

O'REILLY®

TURING

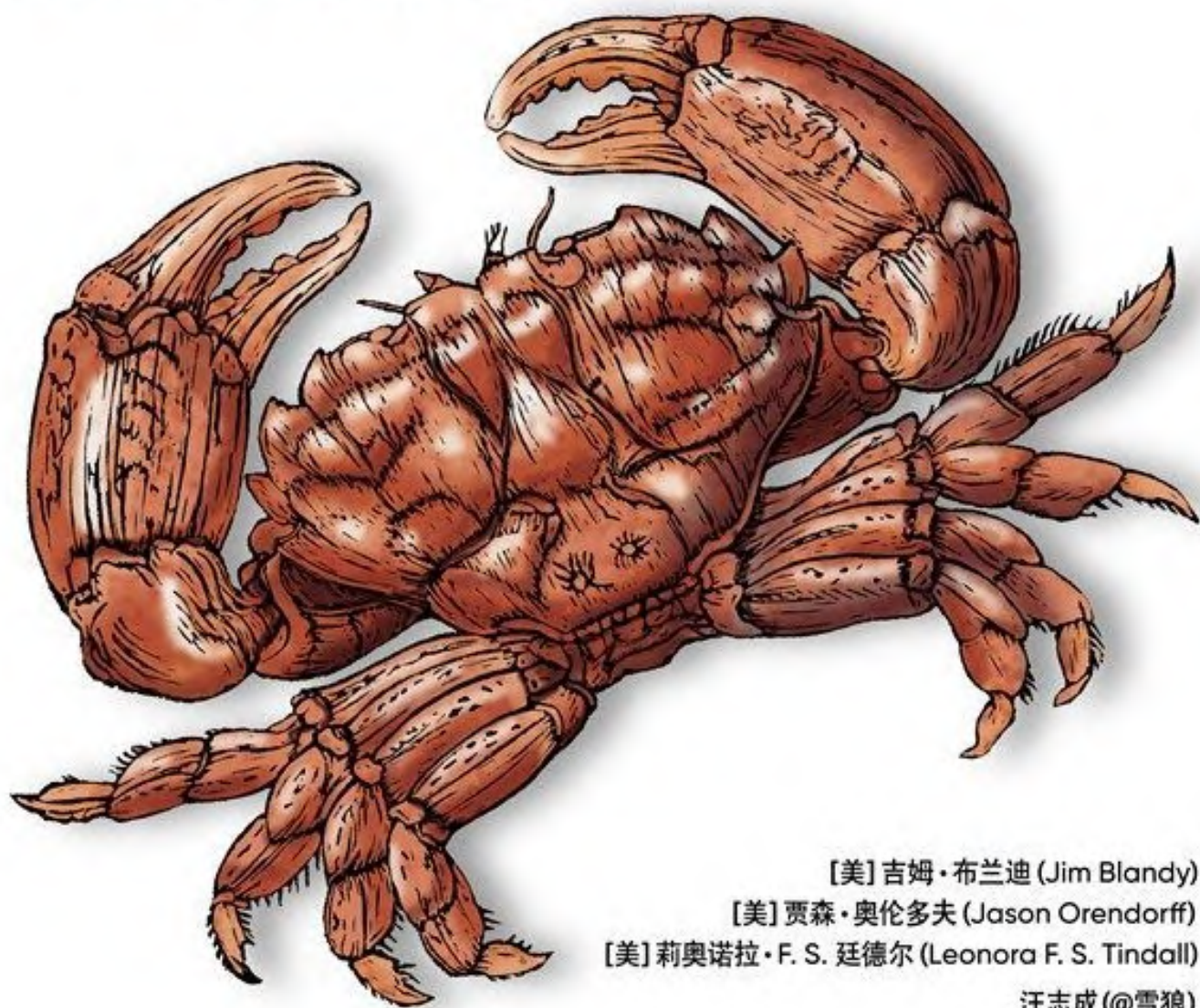
图灵程序设计丛书

第2版
基于Rust 2021 Edition

Rust程序设计

Programming Rust, Second Edition

构建安全、快速、高并发的应用



[美] 吉姆·布兰迪 (Jim Blandy)

[美] 贾森·奥伦多夫 (Jason Orendorff) 著

[美] 莉奥诺拉·F. S. 廷德尔 (Leonora F. S. Tindall)

汪志成 (@雪狼) 译



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

版权信息

书名：Rust 程序设计（第2版）

作者：[美] 吉姆·布兰迪（Jim Blandy） 贾森·奥伦多夫（Jason Orendorff） 莉奥诺拉·F. S. 廷德尔（Leonora F. S. Tindall）

译者：汪志成（@雪狼）

ISBN：978-7-115-62181-8

本书由北京图灵文化发展有限公司发行数字版。版权所有，侵权必究。

您购买的图灵电子书仅供您个人使用，未经授权，不得以任何方式复制和传播本书内容。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

版权声明

O'Reilly Media, Inc. 介绍

专家推荐

译者序

致谢

前言

读者对象

写作初衷

浏览本书

排版约定

使用代码示例

联系我们

致谢

中文版审读致谢

第 1 章 系统程序员也能享受美好

1.1 Rust 为你负重前行

1.2 高效并行编程

1.3 性能毫不妥协

1.4 协作无边无界

第 2 章 Rust 导览

2.1 rustup 与 Cargo

2.2 Rust 函数

2.3 编写与运行单元测试

2.4 处理命令行参数

2.5 搭建 Web 服务器

2.6 并发

2.6.1 什么是曼德博集

2.6.2 解析并配对命令行参数

- 2.6.3 从像素到复数的映射
- 2.6.4 绘制曼德博集
- 2.6.5 写入图像文件
- 2.6.6 并发版曼德博程序
- 2.6.7 运行曼德博绘图器
- 2.6.8 大“安”无形
- 2.7 文件系统与命令行工具
 - 2.7.1 命令行界面
 - 2.7.2 读写文件
 - 2.7.3 查找并替换

第 3 章 基本数据类型

- 3.1 固定宽度的数值类型
 - 3.1.1 整型
 - 3.1.2 检查算法、回绕算法、饱和算法和溢出算法
 - 3.1.3 浮点类型
- 3.2 布尔类型
- 3.3 字符
- 3.4 元组
- 3.5 指针类型
 - 3.5.1 引用
 - 3.5.2 Box
 - 3.5.3 裸指针
- 3.6 数组、向量和切片
 - 3.6.1 数组
 - 3.6.2 向量
 - 3.6.3 切片
- 3.7 字符串类型
 - 3.7.1 字符串字面量

- 3.7.2 字节串
 - 3.7.3 内存中的字符串
 - 3.7.4 `String`
 - 3.7.5 使用字符串
 - 3.7.6 其他类似字符串的类型
- 3.8 类型别名
- 3.9 前路展望
- 第 4 章 所有权与移动 ¹
 - 4.1 所有权
 - 4.2 移动
 - 4.2.1 更多移动类操作
 - 4.2.2 移动与控制流
 - 4.2.3 移动与索引内容
 - 4.3 `Copy` 类型：关于移动的例外情况
 - 4.4 `Rc` 与 `Arc`：共享所有权
- 第 5 章 引用
 - 5.1 对值的引用
 - 5.2 使用引用
 - 5.2.1 `Rust` 引用与 `C++` 引用
 - 5.2.2 对引用变量赋值
 - 5.2.3 对引用进行引用
 - 5.2.4 比较引用
 - 5.2.5 引用永不为空
 - 5.2.6 借用任意表达式结果值的引用
 - 5.2.7 对切片和特型对象的引用
 - 5.3 引用安全
 - 5.3.1 借用局部变量
 - 5.3.2 将引用作为函数参数

- 5.3.3 把引用传给函数
- 5.3.4 返回引用
- 5.3.5 包含引用的结构体
- 5.3.6 不同的生命周期参数
- 5.3.7 省略生命周期参数

5.4 共享与可变

5.5 应对复杂对象关系

第 6 章 表达式

6.1 表达式语言

6.2 优先级与结合性

6.3 块与分号

6.4 声明

6.5 if 与 match

6.5.1 if let

6.5.2 循环

6.6 循环中的控制流

6.7 return 表达式

6.8 为什么 Rust 中会有 loop

6.9 函数与方法调用

6.10 字段与元素

6.11 引用运算符

6.12 算术运算符、按位运算符、比较运算符和逻辑运算符

6.13 赋值

6.14 类型转换

6.15 闭包

6.16 前路展望

第 7 章 错误处理

7.1 panic

- 7.1.1 展开调用栈
 - 7.1.2 中止
- 7.2 Result
 - 7.2.1 捕获错误
 - 7.2.2 Result 类型别名
 - 7.2.3 打印错误
 - 7.2.4 传播错误
 - 7.2.5 处理多种 Error 类型
 - 7.2.6 处理“不可能发生”的错误
 - 7.2.7 忽略错误
 - 7.2.8 处理 main() 中的错误
 - 7.2.9 声明自定义错误类型
 - 7.2.10 为什么是 Result
- 第 8 章 crate 与模块
 - 8.1 crate
 - 8.1.1 版本
 - 8.1.2 创建配置文件
 - 8.2 模块
 - 8.2.1 嵌套模块
 - 8.2.2 单独文件中的模块
 - 8.2.3 路径与导入
 - 8.2.4 标准库预导入
 - 8.2.5 公开 use 声明
 - 8.2.6 公开结构体字段
 - 8.2.7 静态变量与常量
 - 8.3 将程序变成库
 - 8.4 src/bin 目录
 - 8.5 属性

- 8.6 测试与文档
 - 8.6.1 集成测试
 - 8.6.2 文档
 - 8.6.3 文档测试
- 8.7 指定依赖项
 - 8.7.1 版本
 - 8.7.2 Cargo.lock
- 8.8 将 crate 发布到 crates.io
- 8.9 工作空间
- 8.10 更多好资源

第 9 章 结构体

- 9.1 具名字段型结构体
- 9.2 元组型结构体
- 9.3 单元型结构体
- 9.4 结构体布局
- 9.5 用 impl 定义方法
 - 9.5.1 以 Box、Rc 或 Arc 形式传入 self
 - 9.5.2 类型关联函数
- 9.6 关联常量
- 9.7 泛型结构体
- 9.8 带生命周期参数的泛型结构体
- 9.9 带常量参数的泛型结构体
- 9.10 让结构体类型派生自某些公共特型
- 9.11 内部可变性

第 10 章 枚举与模式

- 10.1 枚举
 - 10.1.1 带数据的枚举
 - 10.1.2 内存中的枚举

- 10.1.3 用枚举表示富数据结构
 - 10.1.4 泛型枚举
 - 10.2 模式
 - 10.2.1 模式中的字面量、变量和通配符
 - 10.2.2 元组型模式与结构体型模式
 - 10.2.3 数组型模式与切片型模式
 - 10.2.4 引用型模式
 - 10.2.5 匹配守卫
 - 10.2.6 匹配多种可能性
 - 10.2.7 使用@模式绑定
 - 10.2.8 模式能用在哪里
 - 10.2.9 填充二叉树
 - 10.3 大局观
- 第 11 章 特型与泛型
- 11.1 使用特型
 - 11.1.1 特型对象
 - 11.1.2 泛型函数与类型参数
 - 11.1.3 使用哪一个
 - 11.2 定义与实现特型
 - 11.2.1 默认方法
 - 11.2.2 特型与其他人的类型
 - 11.2.3 特型中的 `self`
 - 11.2.4 子特型
 - 11.2.5 类型关联函数
 - 11.3 完全限定的方法调用
 - 11.4 定义类型之间关系的特型
 - 11.4.1 关联类型（或迭代器的工作原理）
 - 11.4.2 泛型特型（或运算符重载的工作原理）

- 11.4.3 `impl Trait`
 - 11.4.4 关联常量
 - 11.5 逆向工程求限界
 - 11.6 以特型为基础
- 第 12 章 运算符重载
 - 12.1 算术运算符与按位运算符
 - 12.1.1 一元运算符
 - 12.1.2 二元运算符
 - 12.1.3 复合赋值运算符
 - 12.2 相等性比较
 - 12.3 有序比较
 - 12.4 `Index` 与 `IndexMut`
 - 12.5 其他运算符
- 第 13 章 实用工具特型
 - 13.1 `Drop`
 - 13.2 `Sized`
 - 13.3 `Clone`
 - 13.4 `Copy`
 - 13.5 `Deref` 与 `DerefMut`
 - 13.6 `Default`
 - 13.7 `AsRef` 与 `AsMut`
 - 13.8 `Borrow` 与 `BorrowMut`
 - 13.9 `From` 与 `Into`
 - 13.10 `TryFrom` 与 `TryInto`
 - 13.11 `ToOwned`
 - 13.12 `Borrow` 与 `ToOwned` 的实际运用：谦卑²的 `Cow`
- 第 14 章 闭包

14.1 捕获变量

14.1.1 借用值的闭包

14.1.2 “窃取”值的闭包

14.2 函数与闭包的类型

14.3 闭包性能

14.4 闭包与安全

14.4.1 “杀死”闭包

14.4.2 FnOnce

14.4.3 FnMut

14.4.4 对闭包的 Copy 与 Clone

14.5 回调

14.6 高效地使用闭包

第 15 章 迭代器

15.1 Iterator 特型与 IntoIterator 特型

15.2 创建迭代器

15.2.1 iter 方法与 iter_mut 方法

15.2.2 IntoIterator 的实现

15.2.3 from_fn 与 successors

15.2.4 drain 方法

15.2.5 其他迭代器源

15.3 迭代器适配器

15.3.1 map 与 filter

15.3.2 filter_map 与 flat_map

15.3.3 flatten

15.3.4 take 与 take_while

15.3.5 skip 与 skip_while

15.3.6 peekable

- 15.3.7 fuse
- 15.3.8 可逆迭代器与 rev
- 15.3.9 inspect
- 15.3.10 chain
- 15.3.11 enumerate
- 15.3.12 zip
- 15.3.13 by_ref
- 15.3.14 cloned 与 copied
- 15.3.15 cycle

15.4 消耗迭代器

- 15.4.1 简单累加: count、sum 和 product
- 15.4.2 min 与 max
- 15.4.3 max_by 与 min_by
- 15.4.4 max_by_key 与 min_by_key
- 15.4.5 对条目序列进行比较
- 15.4.6 any 与 all
- 15.4.7 position、rposition 和

ExactSizeIterator

- 15.4.8 fold 与 rfold
- 15.4.9 try_fold 与 try_rfold
- 15.4.10 nth 与 nth_back
- 15.4.11 last
- 15.4.12 find、rfind 和 find_map
- 15.4.13 构建集合: collect 与 FromIterator
- 15.4.14 Extend 特型
- 15.4.15 partition
- 15.4.16 for_each 与 try_for_each

- 15.5 实现自己的迭代器
- 第 16 章 集合
 - 16.1 概述
 - 16.2 Vec<T>
 - 16.2.1 访问元素
 - 16.2.2 迭代
 - 16.2.3 扩大向量与收缩向量
 - 16.2.4 联结
 - 16.2.5 拆分
 - 16.2.6 交换
 - 16.2.7 填充
 - 16.2.8 排序与搜索
 - 16.2.9 比较切片
 - 16.2.10 随机元素
 - 16.2.11 Rust 中不存在失效型错误
 - 16.3 VecDeque<T>
 - 16.4 BinaryHeap<T>
 - 16.5 HashMap<K, V> 与 BTreeMap<K, V>
 - 16.5.1 条目
 - 16.5.2 对 Map 进行迭代
 - 16.6 HashSet<T> 与 BTreeSet<T>
 - 16.6.1 对 Set 进行迭代
 - 16.6.2 当相等的值不完全相同时
 - 16.6.3 针对整个 Set 的运算
 - 16.7 哈希
 - 16.8 使用自定义哈希算法
 - 16.9 在标准集合之外
- 第 17 章 字符串与文本

- 17.1 一些 Unicode 背景知识
 - 17.1.1 ASCII、Latin-1 和 Unicode
 - 17.1.2 UTF-8 编码
 - 17.1.3 文本方向性
- 17.2 字符 (char)
 - 17.2.1 字符分类
 - 17.2.2 处理数字
 - 17.2.3 字符大小写转换
 - 17.2.4 与整数之间的转换
- 17.3 String 与 str
 - 17.3.1 创建字符串值
 - 17.3.2 简单探查
 - 17.3.3 追加文本与插入文本
 - 17.3.4 移除文本与替换文本
 - 17.3.5 搜索与迭代的约定
 - 17.3.6 搜索文本的模式
 - 17.3.7 搜索与替换
 - 17.3.8 遍历文本
 - 17.3.9 修剪
 - 17.3.10 字符串的大小写转换
 - 17.3.11 从字符串中解析出其他类型
 - 17.3.12 将其他类型转换为字符串
 - 17.3.13 借用其他类似文本的类型
 - 17.3.14 以 UTF-8 格式访问文本
 - 17.3.15 从 UTF-8 数据生成文本
 - 17.3.16 推迟分配
 - 17.3.17 把字符串当作泛型集合
- 17.4 格式化各种值

- 17.4.1 格式化文本值
- 17.4.2 格式化数值
- 17.4.3 格式化其他类型
- 17.4.4 格式化值以进行调试
- 17.4.5 格式化指针以进行调试
- 17.4.6 按索引或名称引用参数
- 17.4.7 动态宽度与动态精度
- 17.4.8 格式化自己的类型
- 17.4.9 在自己的代码中使用格式化语言
- 17.5 正则表达式
 - 17.5.1 `Regex` 的基本用法
 - 17.5.2 惰性构建正则表达式值
- 17.6 规范化
 - 17.6.1 规范化形式
 - 17.6.2 `unicode-normalization crate`

第 18 章 输入与输出

- 18.1 读取器与写入器
 - 18.1.1 读取器
 - 18.1.2 缓冲读取器
 - 18.1.3 读取行
 - 18.1.4 收集行
 - 18.1.5 写入器
 - 18.1.6 文件
 - 18.1.7 寻址
 - 18.1.8 其他读取器与写入器类型
 - 18.1.9 二进制数据、压缩和序列化
- 18.2 文件与目录
 - 18.2.1 `OsStr` 与 `Path`

18.2.2 Path 与 PathBuf 的方法

18.2.3 访问文件系统的函数

18.2.4 读取目录

18.2.5 特定于平台的特性

18.3 网络

第 19 章 并发

19.1 分叉与合并并行

19.1.1 启动与联结

19.1.2 跨线程错误处理

19.1.3 跨线程共享不可变数据

19.1.4 rayon

19.1.5 重温曼德博集

19.2 通道

19.2.1 发送值

19.2.2 接收值

19.2.3 运行管道

19.2.4 通道的特性与性能

19.2.5 线程安全：Send 与 Sync

19.2.6 绝大多数迭代器能通过管道传给通道

19.2.7 除管道之外的用法

19.3 共享可变状态

19.3.1 什么是互斥锁

19.3.2 Mutex<T>

19.3.3 mut 与互斥锁

19.3.4 为什么互斥锁不是“银弹”

19.3.5 死锁

19.3.6 “中毒”的互斥锁

19.3.7 使用互斥锁的多消费者通道

19.3.8 读/写锁 (RwLock<T>)

19.3.9 条件变量 (Condvar)

19.3.10 原子化类型

19.3.11 全局变量

19.4 在 Rust 中编写并发代码的一点儿经验

第 20 章 异步编程

20.1 从同步到异步

20.1.1 Future

20.1.2 异步函数与 await 表达式

20.1.3 从同步代码调用异步函数: block_on

20.1.4 启动异步任务

20.1.5 异步块

20.1.6 从异步块构建异步函数

20.1.7 在线程池中启动异步任务

20.1.8 你的 Future 实现 Send 了吗

20.1.9 长时间运行的计算: yield_now 与
spawn_blocking

20.1.10 对几种异步设计进行比较

20.1.11 一个真正的异步 HTTP 客户端

20.2 异步客户端与服务器

20.2.1 Error 类型与 Result 类型

20.2.2 协议

20.2.3 获取用户输入: 异步流

20.2.4 发送数据包

20.2.5 接收数据包: 更多异步流

20.2.6 客户端的 main 函数

20.2.7 服务器的 main 函数

20.2.8 处理聊天连接: 异步互斥锁

- 20.2.9 群组表：同步互斥锁
 - 20.2.10 聊天组：tokio 的广播通道
 - 20.3 原始 Future 与执行器：Future 什么时候值得再次轮询
 - 20.3.1 调用唤醒器：spawn_blocking
 - 20.3.2 实现 block_on
 - 20.4 固定 (Pin)⁵
 - 20.4.1 Future 生命周期的两个阶段
 - 20.4.2 固定指针
 - 20.4.3 Unpin 特型
 - 20.5 什么时候要用异步代码
- 第 21 章 宏
- 21.1 宏基础
 - 21.1.1 宏展开的基础
 - 21.1.2 意外后果
 - 21.1.3 重复
 - 21.2 内置宏
 - 21.3 调试宏
 - 21.4 构建 json! 宏
 - 21.4.1 片段类型
 - 21.4.2 宏中的递归
 - 21.4.3 将特型与宏一起使用
 - 21.4.4 作用域界定与卫生宏
 - 21.4.5 导入宏和导出宏
 - 21.5 在匹配过程中避免语法错误
 - 21.6 超越 macro_rules!
- 第 22 章 不安全代码
- 22.1 不安全因素来自哪里

- 22.2 不安全块
- 22.3 示例：高效的 ASCII 字符串类型
- 22.4 不安全函数
- 22.5 不安全块还是不安全函数
- 22.6 未定义行为
- 22.7 不安全特型
- 22.8 裸指针
 - 22.8.1 安全地解引用裸指针
 - 22.8.2 示例：RefWithFlag
 - 22.8.3 可空指针
 - 22.8.4 类型大小与对齐方式
 - 22.8.5 指针运算
 - 22.8.6 移动入和移动出内存
 - 22.8.7 示例：GapBuffer
 - 22.8.8 不安全代码中的 panic 安全性
- 22.9 用联合体重新解释内存
- 22.10 匹配联合体
- 22.11 借用联合体
- 第 23 章 外部函数
 - 23.1 寻找共同的数据表示
 - 23.2 声明外部函数与变量
 - 23.3 使用库中的函数
 - 23.4 libgit2 的裸接口
 - 23.5 libgit2 的安全接口
 - 23.6 结论

作者介绍

封面介绍

版权声明

Copyright © 2021 Jim Blandy, Leonora F. S. Tindall, Jason Orendorff. All rights reserved.

Simplified Chinese edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2023. Authorized translation of the English edition, 2023 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版, 2021。

简体中文版由人民邮电出版社有限公司出版, 2023。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有, 未得书面许可, 本书的任何部分和全部不得以任何形式重制。

O'Reilly Media, Inc. 介绍

O'Reilly 以“分享创新知识、改变世界”为己任。40 多年来我们一直向企业、个人提供成功所必需之技能及思想，激励他们创新并做得更好。

O'Reilly 业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、互动学习、认证体验、图书及视频等，使客户更容易获取业务成功所需的专业知识。几十年来 O'Reilly 图书一直被视为学习开创未来之技术的权威资料。我们所做的一切是为了帮助各领域的专业人士学习最佳实践，发现并塑造科技行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的领域，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，那就走小路。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

专家推荐

这本书是深入探索 Rust 语言的权威指南，也是 Rust 编程思维的启示录。其最大价值是深入浅出地介绍了 Rust 的所有关键特性，从基本语法到并发和异步编程，每一章都配备了实用的示例代码和细致的解析。愿这本书引领你走进 Rust 的世界，体验其独特的魅力！

——陈天，Tubi SVP

非常高兴看到《Rust 程序设计（第 2 版）》引进出版。这本书对 Rust 知识做了全面而细致的介绍，特别花大量篇幅介绍了日常开发中用得最多的字符串、集合、迭代器等基础知识——对 Rust 初学者来说，这无疑是非常重要的。预祝这本书的面世为 Rust 语言在国内的普及打开新局面。

——唐刚（@Mike Tang），Rust 语言中文社区联合创始人

多年前，我们就被 Rust 的无 GC 内存安全和零开销抽象所吸引。2019 年以来，随着 Rust 的成熟，我们开始逐渐用 Rust 重写项目，以 Kata Agent 为例，从 Go 换成 Rust 之后代码体积缩小到了之前的 1/20 以下，这对任何系统软件来说都是无法拒绝的。在我的团队里，优秀的内核和系统开发人员都会毫不拒绝从 C、Go 或其他语言转向 Rust，并边开发边学习。但是，Rust 被公认是一门上手难的语言，尤其是对于很多初学者，一本系统的好书才是最佳的入门阶梯。非常开心图灵公司引进《Rust 程序设计（第 2 版）》，相信这本书对于大家学习 Rust，乃至提高国内系统程序的安全性，都会有很大的帮助。

——王旭，Kata Containers 项目联合发起人、开放基础设施基金会董事、
蚂蚁集团容器基础设施团队负责人

这本书是不可或缺的 Rust 编程资源，适合各个层次的开发者。书中深入解析了 Rust 的核心概念，如所有权、类型系统、并发等，配以实例代码，使得理解更为直观。如果你渴望深入探索 Rust，这本书将是你的最佳伙伴。

——张汉东，资深 Rust 独立咨询师、Rust 中文社区布道者、
《Rust 编程之道》作者

我一直在寻找一本先实践后理论，能够提供有深度且实践性强的 Rust 编程知识的教程。当阅读这本书时，我发现了自己一直在寻找的东西。

——张炎泼，分布式系统专家

译者序

我曾是一名“前端”，在那之前，我是一名“后端”，而在刚入行的时候，我是一名系统程序员。现在，我是一名“锈儿”（Rust 的英文是“铁锈”的意思）。

在系统编程领域，一场变革正在悄然到来，而这次的主题，叫作“安全”。

2017 年，英国劳合社（Lloyd's of London）在一份报告中称，一场全球范围的大规模网络攻击造成的经济损失平均达 530 亿美元，这与一场特大自然灾害 [比如 2012 年的美国超级飓风桑迪（Sandy）] 造成的损失相当。而前几年轰动全球的勒索病毒 WannaCry，在单次事件中造成的直接经济损失就高达 80 亿美元。

这些，都还没有包含企业形象、政府形象等方面的无形损失，以及给受害者造成的次生灾害。

是时候改变这一切了。

与其四处漏水之后再“亡羊补牢”，不如从一开始就让安全内建在软件中。这个理念早已是业界共识。然而，好用的工具太少了，特别是对系统程序员来说，而系统程序恰恰是被攻击的重灾区。

造成这种局面的原因很多：一是系统程序往往工作在底层，容易获得很高的权限，一旦被突破，无论上层如何封堵也作用有限；二是系统程序普遍使用汇编、C、C++ 等语言编写，这些语言本身不仅过于复杂，也过于灵活，导致很难开发出全面的安全辅助工具；三是系统程序员数量少、任务重，很难抽出时间仔细思考安全问题。

中国的科技产业要想扎下根、扎稳根，必然要涉足系统程序开发的领域。如今，这种需求比以往更加紧迫。

作为 Thoughtworks 的咨询师，我服务过很多企业。前些年，我们的大部分客户是做上层应用开发的，特别是在互联网和金融方面；这些

年，越来越多的制造业企业开始找到我们。这固然有我们市场品牌推广的功劳，但从另一个角度看，也说明制造业企业正在步入“深水区”，开始有意识地把软件能力融入自身的技术内核。

对制造业企业来说，管理软件固然重要，但是真正的“老大难问题”是嵌入式软件。这是因为只有这些代码才会真正运行在目标场景中，一旦出现问题，可能导致上百万乃至上千万元的损失，甚至会以生命为代价。所有人都知道有问题，但没有人敢动，也没有人知道该怎么动。这，就是无奈的现实！

然而，没有“银弹”。即使本书中介绍的 Rust，也只是候选工具箱中的一个。诚然，它打开了一扇新的大门，开启了一种新的思路，但这还远远不够。安全是一项系统工程，走向“内建安全”的变革更是一个混沌领域：技术、市场、人力、供应链等问题搅在一起，牵一发而动全身。任何人都无法以一己之力改变它，哪怕是总工程师、总经理，甚至董事长。但是我们应该从现在开始就进行知识储备和安全科普，一点点积累共识，逐渐形成变革的合力。

相对而言，编码方面的问题也许才是最单纯的。毕竟，只要能编译成机器码，CPU 才不会管那么多。编码，有机会成为变革的突破口——安全漏洞减少 => 开发效率提升 => 市场竞争力提升 => 利润提升 => 进一步投资于安全，开启一个良性循环。

产业升级的关键是先进制造业的升级，而先进制造业升级的第一步就是形成差异化竞争力。随着智能设备的爆发和万物互联的持续推进，网络攻击必然进一步广泛化，安全也必将在每个消费者的心中成为产品的核心属性。那时候的你，将是引领者还是跟随者呢？

千里之行，始于足下。是时候好好规划你的未来了。

愿本书能有幸成为你的上马石！

与君共勉！

最后是不可或缺的致谢环节。

致谢

首先当然要感谢我可爱的女儿冬冬，在枯燥的译书过程中，是她不时把我拉回现实世界，让我能站起来活动活动，进而规避了很多健康风险。译书期间正值春节，少了很多陪伴她的时间，就拿本书作为迟到的新年礼物吧。

接着要感谢我的爱人春娜，是她在生活中给了我很多支持，让我能把精力投入对翻译质量的追求上。

感谢我的老友叶志敏，在一些我拿不准的翻译上，他给了我很多帮助，并深度参与了本书前期的审阅和修订工作。

感谢我的朋友罗超然（Manonloki），他是全栈工程师，某浏览器产品负责人，主要从事 Tauri+Angular 的桌面端应用及浏览器开发。他和他的团队深度参与了审阅过程，并提出了一些宝贵意见。

感谢我的新朋友马申彦，他是腾讯文档桌面端技术负责人，也是 Rust 在国内的早期用户之一。在内部审阅过程中，他提出了一些很有价值的意见，为提高本书的翻译质量做出了很大的贡献。

感谢我的东家 Thoughtworks 和同事们，这里的技术氛围是我这种高龄码农仍能不断学习新技术的依托和动力。

感谢我所在的 Angular 中文社区，是社区中这群始终追求技术卓越的小伙伴让我得以在心态上永葆青春。

感谢后期审读团队的各位老师，是你们的博学严谨让我见识了什么才叫专业！

最后要感谢人民邮电出版社图灵公司的策划编辑刘美英、项目编辑张海艳和其他工作人员，是他们让我接触到这本好书，并非常专业地完成了翻译工作。

值此新春佳节之际，祝我的父母和岳父母，身体健康，吉祥如意！祝天下的老人得享天年！

雪狼

2023 年春节

前言

Rust 是一门用于系统编程的语言。

事实上，大多数程序员并不熟悉系统编程，尽管它是我们日常工作的基础。

假设你合上了笔记本电脑，操作系统检测到这个行为，随即暂停了所有正在运行的程序、关闭屏幕，并让计算机进入睡眠状态。稍后，当你又打开笔记本电脑时，屏幕和其他组件再次启动，每个程序都能从断点处恢复运行。这些我们习以为常的功能，离不开系统程序员编写的大量系统程序。

系统编程用于：

- 操作系统
- 各种设备驱动程序
- 文件系统
- 数据库
- 在成本极低或可靠性要求极高的设备上运行的代码
- 加密解密
- 媒体编解码器（用于读取和写入音频、视频和图像文件的软件）
- 媒体处理（例如，语音识别或照片编辑软件）
- 内存管理（例如，实现垃圾回收器）
- 文本渲染（将文本和字体转换为像素）
- 实现高级编程语言（如 JavaScript 和 Python）
- 网络编程
- 虚拟化和软件容器
- 科学仿真
- 游戏

简而言之，系统编程是资源受限条件下的编程。每字节、每个 CPU 周期，对于程序正常运行都尤为重要。

即便是支持一个最基本的应用程序，涉及的系统代码量也是惊人的。

本书并不会教你如何进行系统编程。虽然本书涵盖了内存管理的许多细节，但是如果你还没有亲自做过系统编程，那么这些细节乍一看可能会有点儿过于深奥。如果你是一名经验丰富的系统程序员，就会看出 Rust 的卓尔不群：这种新工具可以解决困扰整个行业数十年的、众所周知的重大问题。

读者对象

如果你已经是系统程序员并且正准备物色一款 C++ 的替代品，那么本书适合你。如果你是任何一种编程语言的资深开发人员，无论是 C#、Java、Python、JavaScript，还是其他语言，那么本书同样适合你。

不过，你不能仅仅学习 Rust 语言本身。要充分发挥该语言的价值，还需要接触一些系统编程知识。我们建议你在阅读本书的同时，使用 Rust 实现一些系统编程领域的业余项目，利用 Rust 的速度、并发和安全，构建出你以前从未构建过的东西。开头那个主题列表应该会给你一些启发。

写作初衷

最初学习 Rust 时，我们曾期待有这样一本书，这正是本书的写作初衷。我们的目标是直面 Rust 中一些重大而新颖的概念，清晰而深入地呈现它们，从而减少学习中的试错成本。

浏览本书

本书前两章简要介绍了 Rust 的背景知识，接着在第 3 章开始介绍基本数据类型。第 4 章和第 5 章讨论了“所有权”和“引用”这两个核心概念。建议你按顺序通读前 5 章。

第 6 章到第 10 章涵盖了 Rust 这门语言的基础知识：表达式（第 6 章）、错误处理（第 7 章）、crate 与模块（第 8 章）、结构体（第 9 章），以及枚举与模式（第 10 章）。这几章可以稍微读快一些，但不要跳过第 7 章。

第 11 章介绍了特型与泛型，这是最后两个你需要了解的重要概念。特型就像 Java 或 C# 中的接口。它们也是 Rust 用来将你的类型集成到语言本身的主要方式。第 12 章展示了如何用特型支持运算符重载，第 13 章介绍了更多的实用工具特型。

了解特型和泛型可以帮你解锁本书其余部分的知识。不容错过的两个强大工具——闭包和迭代器，会分别在第 14 章和第 15 章中进行介绍。剩下的几章你可以按任意顺序阅读，或根据需要深入阅读。它们涵盖了这门语言的其余部分：集合（第 16 章）、字符串与文本（第 17 章）、输入与输出（第 18 章）、并发（第 19 章）、异步编程（第 20 章）、宏（第 21 章）、不安全代码（第 22 章），以及调用来自其他语言的函数（第 23 章）。

排版约定

本书使用以下排版约定。

黑体

表示新术语或重点强调的内容。

等宽字体 (`constant width`)

表示程序片段，以及正文中出现的变量名、函数名、数据库、数据类型、环境变量、语句和关键字等。

加粗等宽字体 (**`constant width bold`**)

表示应该由用户输入的命令或其他文本。

等宽斜体 (*`constant width italic`*)

表示应该由用户输入的值或根据上下文确定的值替换的文本。



此图标表示一般性注释。

使用代码示例

补充材料（代码示例、练习等）可在
<https://github.com/ProgrammingRust> 下载。

本书旨在帮助你完成工作。一般来说，你可以在自己的程序或文档中使用本书提供的示例代码。除非需要复制大量代码，否则无须联系我们获得许可。比如，使用本书中的几个代码片段编写程序无须获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无须获得许可，将本书中的大量示例代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如 “*Programming Rust, Second Edition* by Jim Blandy, Jason Orendorff, and Leonora F. S. Tindall (O'Reilly). Copyright 2021 Jim Blandy, Leonora F. S. Tindall, and Jason Orendorff, 978-1-492-05259-3” 。

如果你对示例代码的用法超出了上述的许可范围，欢迎你通过 permissions@oreilly.com 与我们联系。

联系我们

如有与本书有关的评价或问题，请联系出版社。

美国：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室
(100035)

奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码¹以及其他信息。本书的网页是
<https://oreil.ly/programming-rust-2e>。

¹也可到图灵社区本书主页 ituring.com.cn/book/2846 下载示例代码。——编者注

对于本书的评论和技术性问题，请发送电子邮件至
errata@oreilly.com.cn。

要了解更多 O'Reilly 图书、培训课程和新闻的信息，请访问以下网站：
<https://www.oreilly.com>。

我们在 Facebook 的地址如下：<http://facebook.com/oreilly>。

请关注我们的 Twitter 动态：<http://twitter.com/oreillymedia>。

我们的 YouTube 视频地址如下：

<http://www.youtube.com/oreillymedia>。

致谢

本书得以面世得益于我们的官方技术审稿人的关注，感谢 Brian Anderson、Matt Brubeck、J. David Eisenberg、Ryan Levick、Jack Moffitt、Carol Nichols 和 Erik Nordin。还要感谢本书各语种版本的翻译人员：Hidemoto Nakada（中田秀基）（日语）、李松峰（简体中文第 1 版）、汪志成（简体中文第 2 版），以及 Adam Bochenek 和 Krzysztof Sawka（波兰语）。

另外，许多非官方审稿人也阅读了早期的草稿并提供了宝贵的反馈。感谢 Eddy Bruel、Nick Fitzgerald、Graydon Hoare、Michael Kelly、Jeffrey Lim、Jakob Olesen、Gian-Carlo Pascutto、Larry Rabinowitz、Jaroslav Šnajdr、Joe Walker 和 Yoshua Wuyts 对本书给出的深思熟虑的评论。Jeff Walden 和 Nicolas Pierron 花大量时间几乎审阅了全书。与任何编程冒险一样，编程书总是因高质量的错误报告而日益兴盛。感谢你们。

Mozilla 非常支持吉姆和贾森在这个项目上的工作，尽管这不属于我们的官方职责范围，并且一定程度上会分散我们的注意力。非常感谢吉姆和贾森的经理 Dave Camp、Naveed Ihsanullah、Tom Tromey 和 Joe Walker 的支持。他们从长远的角度看待 Mozilla 的意义，我们希望这些成果能证明他们对我们的信任是对的。

还要感谢 O'Reilly 出版社帮助这个项目取得成果的每一个人，尤其是我们极富耐心的编辑 Jeff Bleiel 和 Brian MacDonald，以及我们的策划编辑 Zan McQuade。

最重要的是，衷心感谢我们的家人，感谢他们坚定不移的爱、热情和耐心。

中文版审读致谢

作为 Rust 与系统编程领域的经典，本书翻译难度较大。在此，诚挚地感谢在第 2 版中文版出版过程中参与公开审读活动的 30 位开发者（含一线专家与爱好者）。

各位审读专家针对译文提出了大量宝贵的意见与建议，你们严谨的治学态度和对 Rust 语言的深刻理解让译者与编辑受益匪浅，也让本书的译文质量更上一层楼。在此以表格形式列出 30 位审读专家的姓名（按姓氏字母排序），各章及章名与为之贡献的专家对应。

审读划分	章名	审读专家
第 1~3 章	第 1 章 系统程序员也能享受美好	陈骛、姜子龙、尚卓燃（PsiACE）、唐刚（@Mike Tang）
	第 2 章 Rust 导览	
	第 3 章 基本数据类型	
第 4~6 章	第 4 章 所有权与移动	陈翔（@翔翔的学习频道）、刘燚、陶叶港（@scrue1）、杨思杰
	第 5 章 引用	
	第 6 章 表达式	
第 7~9 章	第 7 章 错误处理	寸志、胡屹、苏胤榕、张卫滨
	第 8 章 crate 与模块	
	第 9 章 结构体	
第 10~12 章	第 10 章 枚举与模式	刘久武、张鑫明、阴钰（@yxonic）
	第 11 章 特型与泛型	
	第 12 章 运算符重载	
第 13~15 章	第 13 章 实用工具特型	曹洪伟、柳佳龙、尹钢、张罗东（@zhangll）
	第 14 章 闭包	
	第 15 章 迭代器	
第 16~18 章	第 16 章 集合	Kyle C、杜小豆、钱宇超（@stdrc）、杨光宇
	第 17 章 字符串与文本	

	第 18 章 输入与输出	
第 19~20 章	第 19 章 并发	耿腾、刘祺、逢振洲、赵梓淇 (BugenZ)
	第 20 章 异步编程	
第 21~23 章	第 21 章 宏	艾尼、大妈 (ZoomQuiet)、刘鑫
	第 22 章 不安全代码	
	第 23 章 外部函数	

本书虽已出版，但内容品质的提升不会终止。译者、编辑、审读专家虽已尽力，但疏漏可能在所难免，如果大家在阅读过程中发现任何问题，欢迎将其提交到图灵社区本书的勘误处

(ituring.com.cn/book/2846)。勘误经编辑确认之后会更正在重印书中。

第 1 章 系统程序员也能享受美好

在某些情况（例如 Rust 的目标环境）下，比竞争对手快 10 倍，哪怕只快两倍就能成为决胜的关键。速度决定了一个系统在市场命运，就像在硬件市场上一样。

——Graydon Hoare

现在所有的计算机都支持并行……并行编程就是编程。

——《结构化并程序序设计：高效计算模式》，Michael McCool 等

就连 TrueType 解析器的缺陷都会被攻击者用于监视。安全性对所有软件都很重要。

——Andy Wingo

我们用这 3 条引言作为本书的开篇是别有深意的。但还是先从一个“谜题”开始吧。下面的 C 程序是做什么的？

```
int main(int argc, char **argv) {  
    unsigned long a[1];  
    a[3] = 0x7ffff7b36cebUL;  
    return 0;  
}
```

今天早上，这个程序在 Jim 的笔记本电脑计算机上打印出了下列内容：

```
undef: Error: .netrc file is readable by others.  
undef: Remove password or make file unreadable by others.
```


然后它就崩溃了。如果你在自己的机器上尝试运行这个程序，则可能会出现不一样的结果。这是怎么回事呢？

这个程序存在缺陷。数组 `a` 只有一个元素长，所以根据 C 编程语言标准，使用 `a[3]` 是未定义行为（undefined behavior）¹：

¹ 对于未定义行为，没有任何标准规定这种情况下该怎么做，因此任何后果都有可能出现。
——译者注

当使用一个不可移植的或错误的程序结构或者数据时，如果本国国际标准对其行为没有强制要求，那么此行为就是未定义的。

未定义行为的后果不仅仅是产生难以预料的结果，事实上，它甚至会允许程序做任何事。在我们的例子中，将这个奇怪的数值存储在数组 `a` 的第四个元素中恰好会破坏 `main` 函数的调用栈，这样，当从 `main` 函数返回时，它不会正常退出程序，而是会跳转到 C 标准库中用以从用户主目录的文件中获取密码的代码。这可不妙。

C 和 C++ 给出了数百条规则来规避未定义行为，其中大多数是常识，比如不要访问不应该访问的内存、不要让算术运算溢出、不要除以 0，等等。但是编译器并不会强制执行这些规则——它甚至没有义务去“揭露”这些明显的违规行为。事实上，前面的程序在编译时就没有抛出错误或发出警告。作为程序员，你需要打起十二分的精神来避免这种未定义行为。

从既往经验来看，作为程序员，我们在这方面可没有什么好记录。在犹他大学学习期间，研究员 Peng Li 修改了 C 编译器和 C++ 编译器，让它们“翻译”过的程序能报告自己是否执行了某种形式的未定义行为。他发现几乎所有程序都执行过某种未定义行为，包括那些一直以坚持高标准而声名卓著的项目。幻想靠自己就能避开 C 和 C++ 中的未定义行为，就像幻想只要知道了象棋的规则就能赢得比赛一样。

偶尔出现的奇怪消息或崩溃可能仅仅是质量问题，但自从 1988 年莫里斯（Morris）蠕虫利用类似上述内存越界的技术变体破译用户口

令，继而在早期互联网上大规模扩散以来，未定义行为一直是造成各种安全漏洞的主要原因。

因此 C 和 C++ 将程序员置于这样一种尴尬的境地：这些语言是系统编程的行业标准，但它们对程序员的要求实在太高了，这就注定系统会源源不断地出现崩溃和安全问题。要解开这个“谜题”，只会引出一个更大的问题：难道我们不能做得更好吗？

1.1 Rust 为你负重前行

本章开头的 3 条引言给出了我们的答案。第三条引言来自关于震网（Stuxnet）病毒的报道，这是一种计算机蠕虫病毒，2010 年人们发现它入侵了工业控制设备。它使用许多技术来控制受害者的计算机，其中就包括负责解析文字处理软件中内嵌 TrueType 字体的相关代码的未定义行为。可以肯定的是，这段代码的作者并没有料到它以这种方式被利用，这表明需要担心安全问题的不仅仅是操作系统和服务器的，任何可能处理“来源不可信数据”的软件都会成为漏洞利用的目标。

Rust 语言给了我们一个简单的承诺：只要程序通过了编译器的检查，就不会存在未定义行为。悬空指针、双重释放和空指针解引用都能在编译期捕获。数组引用则会受到编译期检查和运行期检查的双重保护，因此不会存在缓冲区溢出：想想那个不幸的 C 语言程序，它的 Rust 版代码会安全地退出并给出一条错误消息。

另外，Rust 的目标是既安全又易用。为了更好地保障程序的行为，Rust 对代码施加了比 C 和 C++ 更多的限制，而只有靠实践和经验，我们才能逐渐适应这种限制。但就整体而言，这门语言还是灵活且富有表现力的。用 Rust 编写的代码范围之宽、应用领域之广就是证明。

根据我们的经验，如果该语言能发现更多错误，那么我们就可以尝试一些更具雄心的项目。如果内存管理和指针有效性问题已得到妥善处理，那么修改大型复杂程序时的风险就会降低。如果代码里的 bug 不会破坏程序中的其他部分，那么调试也会简单得多。

当然，仍然有很多 Rust 无法检测到的 bug。但实际上，只要从可能出现的问题清单中排除了未定义行为，就会让编程更美好。

1.2 高效并行编程

众所周知，在 C 和 C++ 中恰当使用并发的难度极大。通常只有在确信单线程代码无法达到预期的性能时，开发人员才会转向并发。但正如本章第二条引言所指出的，并行性对现代机器来说太重要了，不应该等到迫不得已时才考虑它。

事实证明，Rust 用来确保内存安全的那些限制同样能确保 Rust 程序避免产生数据竞争（data race）。只要数据不可变，你就可以在线程之间自由地共享这些数据。会发生变化的数据则只能使用同步原语访问。所有传统的并发工具仍然可用：互斥锁、条件变量、通道、原子等。Rust 只负责检查你是否正确地使用了它们。

这就让 Rust 成了一门能充分发挥现代多核机器能力的优秀语言。Rust 的生态系统提供了一些超乎于常规并发原语的库，可帮助你在处理器池之间均匀分布复杂负载、使用无锁同步机制（如读取-复制-更新）等。

1.3 性能毫不妥协

最后，正如本章第一条引言所说的那样，Rust 同样怀揣着 Bjarne Stroustrup 在他的论文 “Abstraction and the C++ Machine Model”（抽象和 C++ 机器模型）中为 C++ 阐明的雄心。

通常，C++ 的各种实现会遵循零开销原则：没用到的，就没有开销；要用到的，你也无法手写出更好的代码。

系统编程通常会涉及极限压榨机器性能。对电子游戏来说，整台机器应该极力为玩家创造最佳体验。对浏览器来说，浏览器的效率决定了内容作者可用能力的上限。在机器的固有限制下，必须将尽可能多的内存和处理器能力留给内容本身。同样的原则也适用于操作系统：内核应该尽可能把机器资源留给用户程序，而不是自己来消耗它们。

但是当我们说 Rust “快” 的时候，到底意味着什么呢？毕竟人们可以用任何通用语言来写出慢速代码。更准确一点儿说，如果你准备让程序充分发挥底层机器的能力，那么 Rust 就会为你提供支持。这门语言设计了一些 “高性能” 的默认选项，并赋予你自主控制内存使用和处理器算力分配方式的能力。

1.4 协作无边无界

本章的标题中还隐藏了第四条引言：“系统程序员也能享受美好”。这是指 Rust 对代码共享和复用的支持。

作为 Rust 的包管理器和构建工具，Cargo 能让你轻松使用别人在 Rust 的公共包存储库 crates.io 网站上发布的各种库。只需将库的名称和所需的版本号添加到文件中，Cargo 就会负责下载这个库以及它所用到的任何其他库，并将所有内容链接在一起。你可以将 Cargo 视为 Rust 下的 NPM 或 RubyGems，其侧重于实现健全的版本管理和可重现的构建。一些广为流行的 Rust 库提供了包括开箱即用的序列化功能、HTTP 客户端和服务端功能以及现代图形 API 功能在内的一切。

再说一步，Rust 这门语言本身也旨在支持协作：借助 Rust 的特型（trait）²和泛型，我们可以创建具有灵活接口的库，将其用在许多不同的上下文中。Rust 的标准库提供了一组最核心的基本类型，这些类型为一些常见的情况建立了共享规约，以方便不同的库彼此协作。

²特型，表示一组具有指定特点的类型。这里之所以把 trait 译为“特型”，是因为特型和泛型是紧密相关的，我们希望读者顾名思义就能在两者之间建立联系；而其他中文译法，或者缺乏特异性，或者难以望文生义，因此并未采纳。关于 trait 翻译与否及译法是个非常有争议的话题，我们在审读群曾经进行过至少 3 次深度讨论，但并未形成共识。对于一个常用专有名词，有一个合适的中文表述是非常必要的。但对于一个词的新译法，一开始，大家可能还不太习惯，不妨试着给新词些时间。——译者注

第 2 章旨在通过介绍几个小的 Rust 程序来具体展示本章所述的这几项优势。

第 2 章 Rust 导览

写这样一本书并不容易：Rust 这门语言如此卓尔不群，我们固然有能力在一开始就展示出其独特的、令人惊叹的特性，但更为重要的是它的各个部分之间能够良好协作，共同服务于我们在第 1 章中设定的目标——安全、高性能的系统编程。该语言的每个部分都与其他部分配合得天衣无缝。

因此，我们并不打算每次讲透一个语言特性，而是准备了一些小而完备的程序作为导览，每个程序都会在其上下文中介绍该语言的更多特性。

- 作为暖场，我们会设计一个简单的程序，它可以解析命令行参数并进行简单计算，而且带有单元测试。这会展示 Rust 的一些核心类型并引入特型的概念。
- 接下来，我们一起构建一个 Web 服务器。我们将使用第三方库来处理 HTTP 的细节，并介绍字符串处理、闭包和错误处理功能。
- 第三个程序会绘制一张美丽的分形图，将计算工作分派到多个线程以提高效率。这包括一个泛型函数的示例，以说明该如何处理像素缓冲区之类的问题，并展示 Rust 对并发的支持。
- 最后，我们会展示一个强大的命令行工具，它利用正则表达式来处理文件。这展示了 Rust 标准库的文件处理功能，以及最常用的第三方正则表达式库。

Rust 承诺会在对性能影响最小的情况下防止未定义行为，这在潜移默化中引导着每个部分的设计——从标准数据结构（如向量和字符串）到使用第三方库的方式。关于如何做好这些的细节会贯穿全书。但目前而言，我们只想向你证明 Rust 是一门功能强大且易于使用的语言。

当然，你要先在计算机上安装 Rust。

2.1 rustup 与 Cargo

安装 Rust 的最佳方式是使用 rustup。请转到 rustup.rs 网站并按照那里的说明进行操作。

还可以到 Rust 网站获取针对 Linux、macOS 和 Windows 的预构建包。Rust 也已经包含在某些操作系统的发行版中。建议使用 rustup，因为它是专门管理 Rust 安装的工具，就像 Ruby 中的 RVM 或 Node 中的 NVM。例如，当 Rust 发布新版本时，你就可以通过键入 `rustup update` 来实现一键升级。

无论采用哪种方式，完成安装之后，你的命令行中都会有 3 条新命令：

```
$ cargo --version
cargo 1.49.0 (d00d64df9 2020-12-05)
$ rustc --version
rustc 1.49.0 (e1884a8e3 2020-12-29)
$ rustdoc --version
rustdoc 1.49.0 (e1884a8e3 2020-12-29)
```

在这里，`$` 是命令提示符，在 Windows 上，则会是 `C:\>` 之类的文本。在刚才的记录中，我们运行了 3 条已安装的命令，并要求每条命令报告其版本号。下面来逐个看看每条命令。

- `cargo` 是 Rust 的编译管理器、包管理器和通用工具。可以用 Cargo 启动新项目、构建和运行程序，并管理代码所依赖的任何外部库。
- `rustc` 是 Rust 编译器。通常 Cargo 会替我们调用此编译器，但有时也需要直接运行它。
- `rustdoc` 是 Rust 文档工具。如果你在程序源代码中以适当形式的注释编写文档，那么 `rustdoc` 就可以从中构建出格式良好的 HTML。与 `rustc` 一样，通常 Cargo 会替我们运行 `rustdoc`。

为便于使用，Cargo 可以为我们创建一个新的 Rust 包，并适当准备一些标准化的元数据：

```
$ cargo new hello
    Created binary (application) `hello` package
```

该命令会创建一个名为 hello 的新包目录，用于构建命令行可执行文件。

查看包的顶层目录：

```
$ cd hello
$ ls -la
total 24
drwxrwxr-x.  4 jimb jimb 4096 Sep 22 21:09 .
drwx----- 62 jimb jimb 4096 Sep 22 21:09 ..
drwxrwxr-x.  6 jimb jimb 4096 Sep 22 21:09 .git
-rw-rw-r--.  1 jimb jimb   7 Sep 22 21:09 .gitignore
-rw-rw-r--.  1 jimb jimb  88 Sep 22 21:09 Cargo.toml
drwxrwxr-x.  2 jimb jimb 4096 Sep 22 21:09 src
```

我们看到 Cargo 已经创建了一个名为 Cargo.toml 的文件来保存此包的元数据。目前这个文件还没有多少内容：

```
[package]
name = "hello"
version = "0.1.0"
edition = "2021"

# 请到“The Cargo Book”查看更多的键及其定义

[dependencies]
```

如果程序依赖于其他库，那么可以把它们记录在这个文件中，Cargo 将为我们下载、构建和更新这些库。第 8 章会详细介绍 Cargo.toml 文件。

Cargo 已将我们的包设置为与版本控制系统 git 一起使用，并为此创建了一个元数据子目录 .git 和一个 .gitignore 文件。可以通过

在命令行中将 `--vcs none` 传给 `cargo new` 来要求 Cargo 跳过此步骤。

`src` 子目录包含实际的 Rust 代码：

```
$ cd src
$ ls -l
total 4
-rw-rw-r--. 1 jimb jimb 45 Sep 22 21:09 main.rs
```

Cargo 似乎已经替我们写好一部分程序了。`main.rs` 文件包含以下文本：

```
fn main() {
    println!("Hello, world!");
}
```

在 Rust 中，你甚至不需要编写自己的“Hello, World!”程序。这是 Rust 新程序样板的职责，该程序样板包括两个文件，总共 13 行代码。

可以在包内的任意目录下调用 `cargo run` 命令来构建和运行程序：

```
$ cargo run
   Compiling hello v0.1.0 (/home/jimb/rust/hello)
   Finished dev [unoptimized + debuginfo] target(s) in 0.28s
   Running `/home/jimb/rust/hello/target/debug/hello`
Hello, world!
```

这里 Cargo 先调用 Rust 编译器 `rustc`，然后运行了它生成的可执行文件。Cargo 将可执行文件放在此包顶层的 `target` 子目录中：

```
$ ls -l ../target/debug
total 580
drwxrwxr-x. 2 jimb jimb 4096 Sep 22 21:37 build
drwxrwxr-x. 2 jimb jimb 4096 Sep 22 21:37 deps
drwxrwxr-x. 2 jimb jimb 4096 Sep 22 21:37 examples
-rwxrwxr-x. 1 jimb jimb 576632 Sep 22 21:37 hello
-rw-rw-r--. 1 jimb jimb 198 Sep 22 21:37 hello.d
drwxrwxr-x. 2 jimb jimb 68 Sep 22 21:37 incremental
```

```
$ ../target/debug/hello  
Hello, world!
```

完工之后，Cargo 还可以帮我们清理生成的文件。

```
$ cargo clean  
$ ../target/debug/hello  
bash: ../target/debug/hello: No such file or directory
```

2.2 Rust 函数

Rust 在语法设计上刻意减少了原创性。如果你熟悉 C、C++、Java 或 JavaScript，那么就能通过 Rust 程序的一般性构造找到自己的快速学习之道。这是一个使用欧几里得算法计算两个整数的最大公约数的函数。可以将这些代码添加到 `src/main.rs` 的末尾：

```
fn gcd(mut n: u64, mut m: u64) -> u64 {
    assert!(n != 0 && m != 0);
    while m != 0 {
        if m < n {
            let t = m;
            m = n;
            n = t;
        }
        m = m % n;
    }
    n
}
```

`fn`（发音为 /fʌn/）关键字引入了一个函数。这里我们定义了一个名为 `gcd` 的函数，它有两个参数（`n` 和 `m`），每个参数都是 `u64` 类型，即一个无符号的 64 位整数。`->` 标记后面紧跟着返回类型，表示此函数返回一个 `u64` 值。4 空格缩进是 Rust 的标准风格。

Rust 的“机器整数类型名”揭示了它们的大小和符号：`i32` 是一个带符号的 32 位整数，`u8` 是一个无符号的 8 位整数（“字节”值），以此类推。`isize` 类型和 `usize` 类型保存着恰好等于“指针大小”的有符号整数和无符号整数，在 32 位平台上是 32 位长，在 64 位平台上则是 64 位长。Rust 还有 `f32` 和 `f64` 这两种浮点类型，它们分别是 IEEE 单精度浮点类型和 IEEE 双精度浮点类型，就像 C 和 C++ 中的 `float` 和 `double`。

默认情况下，一经初始化，变量的值就不能再改变了，但是在参数 `n` 和 `m` 之前放置 `mut`（发音为 /mjuːt/，是 `mutable` 的缩写）关键字将会准许我们在函数体中赋值给它们。实际上，大多数变量是不需要

被赋值的，而对于那些确实需要被赋值的变量，`mut` 关键字相当于用一个醒目的提示来帮我们阅读代码。

函数的主体始于一次 `assert!` 宏调用，以验证这两个参数都不为 0。这里的 `!` 字符标明此句为宏调用，而不是函数调用。就像 C 和 C++ 中的 `assert` 宏一样，Rust 的 `assert!` 会检查其参数是否为真，如果非真，则终止本程序并提供一条有帮助的信息，其中包括导致本次检查失败的源代码位置。这种突然的终止在 Rust 中称为 `panic`。与可以跳过断言的 C 和 C++ 不同，Rust 总是会检查这些断言，而不管程序是如何编译的。还有一个 `debug_assert!` 宏，在编译发布版程序时会跳过其断言以提高速度。

这个函数的核心是一个包含 `if` 语句和赋值语句的 `while` 循环。与 C 和 C++ 不同，Rust 不需要在条件表达式周围使用圆括号，但必须在受其控制的语句周围使用花括号。

`let` 语句会声明一个局部变量，比如本函数中的 `t`。只要 Rust 能从变量的使用方式中推断出 `t` 的类型，就不需要标注其类型。在此函数中，通过匹配 `m` 和 `n`，可以推断出唯一适用于 `t` 的类型是 `u64`。Rust 只会推断函数体内部的类型，因此必须像之前那样写出函数参数的类型和返回值的类型。如果想明确写出 `t` 的类型，那么可以这样写：

```
let t: u64 = m;
```

Rust 有 `return` 语句，但这里的 `gcd` 函数并不需要。如果一个函数体以没有尾随着分号的表达式结尾，那么这个表达式就是函数的返回值。事实上，花括号包起来的任意代码块都可以用作表达式。例如，下面是一个打印了一条信息然后以 `x.cos()` 作为其值的表达式：

```
{
    println!("evaluating cos x");
    x.cos()
}
```

在 Rust 中，当控制流“正常离开函数的末尾”时，通常会以上述形式创建函数的返回值，`return` 语句只会用在从函数中间显式地提前返回的场景中。

2.3 编写与运行单元测试

Rust 语言内置了对测试的简单支持。为了测试 `gcd` 函数，可以在 `src/main.rs` 的末尾添加下面这段代码：

```
#[test]
fn test_gcd() {
    assert_eq!(gcd(14, 15), 1);

    assert_eq!(gcd(2 * 3 * 5 * 11 * 17,
                  3 * 7 * 11 * 13 * 19),
              3 * 11);
}
```

这里我们定义了一个名为 `test_gcd` 的函数，该函数会调用 `gcd` 并检查它是否返回了正确的值。此定义顶部的 `#[test]` 将 `test_gcd` 标记为“测试函数”，在正常编译时会跳过它，但如果用 `cargo test` 命令运行我们的程序，则会自动包含并调用它。可以让测试函数分散在源代码树中，紧挨着它们所测试的代码，`cargo test` 会自动收集并运行它们。

`#[test]` 标记是属性（attribute）的示例之一。属性是一个开放式体系，可以用附加信息给函数和其他声明做标记，就像 C++ 和 C# 中的属性或 Java 中的注解（annotation）一样。属性可用于控制编译器警告和代码风格检查、有条件地包含代码（就像 C 和 C++ 中的 `#ifdef` 一样）、告诉 Rust 如何与其他语言编写的代码互动，等等。后面还会介绍更多的属性示例。

将 `gcd` 和 `test_gcd` 的定义添加到本章开头创建的 `hello` 包中，如果当前目录位于此包子树中的任意位置，可以用如下方式运行测试。

```
$ cargo test
  Compiling hello v0.1.0 (/home/jimb/rust/hello)
    Finished test [unoptimized + debuginfo] target(s) in 0.35s
     Running unittests
(/home/jimb/rust/hello/target/debug/deps/hello-2375...)
```

```
running 1 test  
test test_gcd ... ok
```

```
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0  
filtered out
```


2.4 处理命令行参数

为了让我们的程序接收一系列数值作为命令行参数并打印出它们的最大公约数，可以将 `src/main.rs` 中的 `main` 函数替换为以下内容：

```
use std::str::FromStr;
use std::env;

fn main() {
    let mut numbers = Vec::new();

    for arg in env::args().skip(1) {
        numbers.push(u64::from_str(&arg)
            .expect("error parsing argument"));
    }

    if numbers.len() == 0 {
        eprintln!("Usage: gcd NUMBER ...");
        std::process::exit(1);
    }

    let mut d = numbers[0];
    for m in &numbers[1..] {
        d = gcd(d, *m);
    }

    println!("The greatest common divisor of {:?} is {}",
        numbers, d);
}
```

我们来逐段分析一下：

```
use std::str::FromStr;
use std::env;
```

第一个 `use` 声明将标准库中的 `FromStr` 特型引入了当前作用域。特型是可以由类型实现的方法集合。任何实现了 `FromStr` 特型的类型都有一个 `from_str` 方法，该方法会尝试从字符串中解析这个类型的值。`u64` 类型实现了 `FromStr`，所以我们将调用 `u64::from_str` 来解析程序中的命令行参数。尽管我们从未在程序

的其他地方用到 `FromStr` 这个名字，但仍然要 `use`（使用）它，因为要想使用某个特型的方法，该特型就必须在作用域内。第 11 章会详细介绍特型。

第二个 `use` 声明引入了 `std::env` 模块，该模块提供了与执行环境交互时会用到的几个函数和类型，包括 `args` 函数，该函数能让我们访问程序中的命令行参数。

继续看程序中的 `main` 函数：

```
fn main() {
```

`main` 函数没有返回值，所以可以简单地省略 `->` 和通常会跟在参数表后面的返回类型。

```
let mut numbers = Vec::new();
```

我们声明了一个可变的局部变量 `numbers` 并将其初始化为空向量。`Vec` 是 Rust 的可增长向量类型，类似于 C++ 的 `std::vector`、Python 的列表或 JavaScript 的数组。虽然从设计上说向量可以动态扩充或收缩，但仍然要标记为 `mut`，这样 Rust 才能把新值压入末尾。

`numbers` 的类型是 `Vec<u64>`，这是一个可以容纳 `u64` 类型的值的向量，但和以前一样，不需要把类型写出来。Rust 会推断它，一部分原因是我们将 `u64` 类型的值压入了此向量，另一部分原因是我们将此向量的元素传给了 `gcd`，后者只接受 `u64` 类型的值。

```
for arg in env::args().skip(1) {
```

这里使用了 `for` 循环来处理命令行参数，依次将变量 `arg` 指向每个参数并运行循环体。

`std::env` 模块的 `args` 函数会返回一个迭代器，此迭代器会按需生成¹每个参数，并在完成时给出提示。各种迭代器在 Rust 中无处不

在，标准库中也包括一些迭代器，这些迭代器可以生成向量的元素、文件每一行的内容、通信信道上接收到的信息，以及几乎任何有意义的循环变量。Rust 的迭代器非常高效，编译器通常能将它们翻译成与手写循环相同的代码。第 15 章会展示迭代器的工作原理并给出相关示例。

¹ “生成”只是沿袭普遍译法，事实上，这里并不会创建任何新条目，只是把已有条目提供给消费者。——译者注

除了与 `for` 循环一起使用，迭代器还包含大量可以直接使用的方法。例如，`args` 返回的迭代器生成的第一个值永远是正在运行的程序的名称。如果想跳过它，就要调用迭代器的 `skip` 方法来生成一个新的迭代器，新迭代器会略去第一个值。

```
numbers.push(u64::from_str(&arg)
              .expect("error parsing argument"));
```

这里我们调用了 `u64::from_str` 来试图将命令行参数 `arg` 解析为一个无符号的 64 位整数。`u64::from_str` 并不是 `u64` 值上的某个方法，而是与 `u64` 类型相关联的函数，类似于 C++ 或 Java 中的静态方法。`from_str` 函数不会直接返回 `u64`，而是返回一个指明本次解析已成功或失败的 `Result` 值。`Result` 值是以下两种变体之一：

- 形如 `Ok(v)` 的值，表示解析成功了，`v` 是所生成的值；
- 形如 `Err(e)` 的值，表示解析失败了，`e` 是解释原因的错误值。

执行任何可能会失败的操作（例如执行输入或输出或者以其他方式与操作系统交互）的函数都会返回一个 `Result` 类型，其 `Ok` 变体会携带成功结果（传输的字节数、打开的文件等），而其 `Err` 变体会携带错误码，以指明出了什么问题。与大多数现代语言不同，Rust 没有异常（exception）：所有错误都使用 `Result` 或 `panic` 进行处理，详见第 7 章。

我们用 `Result` 的 `expect` 方法来检查本次解析是否成功。如果结果是 `Err(e)`，那么 `expect` 就会打印出一条包含 `e` 的消息并直接退出程序。但如果结果是 `Ok(v)`，则 `expect` 会简单地返回 `v` 本身，最终我们会将其压入这个数值向量的末尾。

```
if numbers.len() == 0 {
    eprintln!("Usage: gcd NUMBER ...");
    std::process::exit(1);
}
```

空数组没有最大公约数，因此要检查此向量是否至少包含一个元素，如果没有则退出程序并报错。这里我们用 `eprintln!` 宏将错误消息写入标准错误流。

```
let mut d = numbers[0];
for m in &numbers[1..] {
    d = gcd(d, *m);
}
```

该循环使用 `d` 作为其运行期间的值，不断地把它更新为已处理的所有数值的最大公约数。和以前一样，必须将 `d` 标记为可变，以便在循环中给它赋值。

这个 `for` 循环有两个值得注意的地方。首先，我们写了 `for m in &numbers[1..]`，那么这里的 `&` 运算符有什么用呢？其次，我们写了 `gcd(d, *m)`，那么 `*m` 中的 `*` 又有何用呢？这两个细节是紧密相关的。

迄今为止，我们的代码只是在对简单的值（例如适合固定大小内存块的整数）进行操作。但现在我们要迭代一个向量，它可以是任意大小，而且可能会非常大。Rust 在处理这类值时非常慎重：它想让程序员控制内存消耗，明确每个值的生存时间，同时还要确保当不再需要这些值时能及时释放内存。

所以在进行迭代时，需要告诉 Rust，该向量的所有权应该留在 `numbers` 上，我们只是为了本次循环而借用它的元素。

`&numbers[1..]` 中的 `&` 运算符会从向量中借用从第二个元素开始

的引用。for 循环会遍历这些被引用的元素，让 m 依次借出每个元素。`*m` 中的 `*` 运算符会将 m 解引用，产生它所引用的值，这就是要传给 gcd 的下一个 u64。最后，由于 numbers 拥有着此向量，因此当 main 末尾的 numbers 超出作用域时，Rust 会自动释放它。

Rust 的所有权规则和引用规则是 Rust 内存管理和并发安全的关键所在，第 4 章和第 5 章会对此进行详细讨论。只有熟悉了这些规则，才算熟练掌握了 Rust。但是对于这个介绍性的导览，你只需要知道 `&x` 借用了对 `x` 的引用，而 `*r` 访问的是 `r` 所引用的值就足够了。

继续我们的程序：

```
println!("The greatest common divisor of {:?} is {}",
        numbers, d);
```

遍历 numbers 的元素后，程序会将结果打印到标准输出流。

`println!` 宏会接受一个模板字符串，在模板字符串中以 `{...}` 形式标出的位置按要求格式化并插入剩余的参数，最后将结果写入标准输出流。

C 和 C++ 要求 main 在程序成功完成时返回 0，在出现问题时返回非零的退出状态，而 Rust 假设只要 main 完全返回，程序就算成功完成。只有显式地调用像 `expect` 或 `std::process::exit` 这样的函数，才能让程序以表示错误的状态码终止。

`cargo run` 命令可以将参数传给程序，因此可以试试下面这些命令行处理：

```
$ cargo run 42 56
   Compiling hello v0.1.0 (/home/jimb/rust/hello)
   Finished dev [unoptimized + debuginfo] target(s) in 0.22s
   Running `/home/jimb/rust/hello/target/debug/hello 42 56`
The greatest common divisor of [42, 56] is 14
$ cargo run 799459 28823 27347
   Compiling hello v0.1.0 (/home/jimb/rust/hello)
   Finished dev [unoptimized + debuginfo] target(s) in 0.02s
   Running `/home/jimb/rust/hello/target/debug/hello 799459
28823 27347`
The greatest common divisor of [799459, 28823, 27347] is 41
```

```
$ cargo run 83
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
    Running `/home/jimb/rust/hello/target/debug/hello 83`
The greatest common divisor of [83] is 83
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
    Running `/home/jimb/rust/hello/target/debug/hello`
Usage: gcd NUMBER ...
```

本节使用了 Rust 标准库中的一些特性。如果你好奇还有哪些别的特性，强烈建议看看 Rust 的在线文档。它具有实时搜索功能，能让你的探索更容易，其中还包括指向源代码的链接。安装 Rust 时，`rustup` 命令会自动在你的计算机上安装一份文档副本。你既可以在 Rust 网站上查看标准库文档，也可以使用以下命令打开浏览器查看。

```
$ rustup doc --std
```

2.5 搭建 Web 服务器

Rust 的优势之一是在 crates.io 网站上发布的大量免费的可用包。cargo 命令可以让你的代码轻松使用 crates.io 上的包：它将下载包的正确版本，然后会构建包，并根据用户的要求更新包。一个 Rust 包，无论是库还是可执行文件，都叫作 crate（发音为 /kreɪt/，意思是“板条箱”）²。Cargo 和 crates.io 的名字都来源于这个术语。

²相对于包，crate 更强调自己的硬边界，也暗示着更高的安全性。助记：马车（Cargo）上装着一些板条箱。——译者注

为了展示这种工作过程，我们将使用 actix-web（Web 框架 crate）、serde（序列化 crate）以及它们所依赖的各种其他 crate 来组装出一个简单的 Web 服务器。如图 2-1 所示，该网站会提示用户输入两个数值并计算它们的最大公约数。

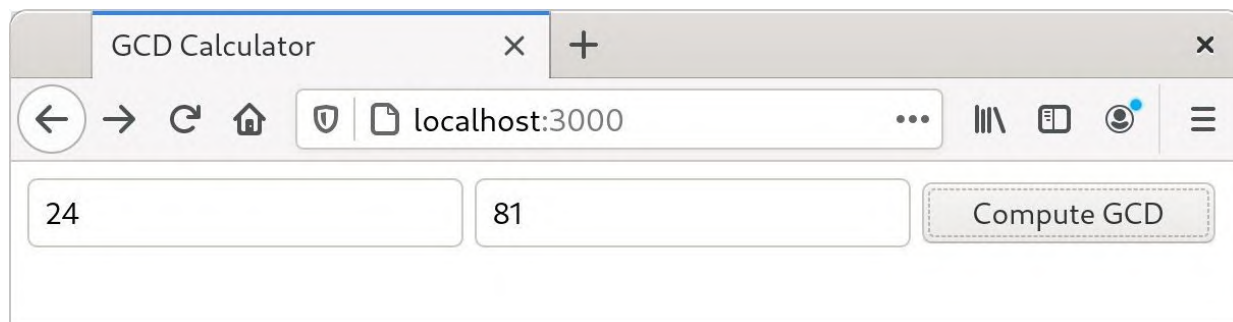


图 2-1：计算最大公约数的网页

首先，让 Cargo 创建一个新包，命名为 actix-gcd：

```
$ cargo new actix-gcd
   Created binary (application) `actix-gcd` package
$ cd actix-gcd
```

然后，编辑新项目的 Cargo.toml 文件以列出所要使用的包，其内容应该是这样的：

```
[package]
name = "actix-gcd"
version = "0.1.0"
edition = "2021"

# 请到“The Cargo Book”查看更多的键及其定义

[dependencies]
actix-web = "1.0.8"
serde = { version = "1.0", features = ["derive"] }
```

Cargo.toml 中 [dependencies] 部分的每一行都给出了 crates.io 上的 crate 名称，以及我们想要使用的那个 crate 的版本。在本例中，我们需要 1.0.8 版的 actix-web crate 和 1.0 版的 serde crate。crates.io 上这些 crate 的版本很可能比此处展示的版本新，但通过指明在测试此代码时所使用的特定版本，可以确保即使发布了新版本的包，这些代码仍然能继续编译。³第 8 章会更详细地讨论版本管理。

³理想情况下确实如此，但现实中并不是每个第三方包都能实现这种兼容性。——译者注

crate 可能具备某些可选特性：一部分接口或实现不是所有用户都需要的，但将其包含在那个 crate 中仍然有意义。例如，serde crate 就提供了一种非常简洁的方式来处理来自 Web 表单的数据，但根据 serde 的文档，只有选择了此 crate 的 derive 特性时它才可用，因此我们在 Cargo.toml 文件中请求了它。

请注意，只需指定要直接用到的那些 crate 即可，cargo 会负责把它们自身依赖的所有其他 crate 带进来。

在第一次迭代中，我们将实现此 Web 服务器的一个简单版本：它只会给出让用户输入要计算的数值的页面。actix-gcd/src/main.rs 的内容如下所示：

```
use actix_web::{web, App, HttpResponse, HttpServer};

fn main() {
    let server = HttpServer::new(|| {
        App::new()
```



```

        .route("/", web::get().to(get_index))
    });

    println!("Serving on http://localhost:3000...");
    server
        .bind("127.0.0.1:3000").expect("error binding server to
address")
        .run().expect("error running server");
}

fn get_index() -> HttpResponse {
    HttpResponse::Ok()
        .content_type("text/html")
        .body(
            r#"
                <title>GCD Calculator</title>
                <form action="/gcd" method="post">
                <input type="text" name="n"/>
                <input type="text" name="m"/>
                <button type="submit">Compute GCD</button>
                </form>
            "#,
        )
}

```

use 声明可以让来自 `actix-web` crate 的定义用起来更容易些。当我们写下 `use actix_web::{...}` 时，花括号中列出的每个名称都可以直接用在代码中，而不必每次都拼出全名，比如 `actix_web::HttpResponse` 可以简写为 `HttpResponse`。（稍后还会提及 `serde` crate。）

`main` 函数很简单：它调用 `HttpServer::new` 创建了一个响应单个路径 `"/"` 请求的服务器，打印了一条信息以提醒我们该如何连接它，然后监听本机的 TCP 端口 3000。

我们传给 `HttpServer::new` 的参数是 Rust 闭包表达式 `|| { App::new() ... }`。闭包是一个可以像函数一样被调用的值。这个闭包没有参数，如果有参数，那么可以将参数名放在两条竖线 `||` 之间。`{ ... }` 是闭包的主体。当我们启动服务器时，Actix 会启动一个线程池来处理传入的请求。每个线程都会调用这个闭包来获取 `App` 值的新副本，以告诉此线程该如何路由这些请求并处理它们。

闭包会调用 `App::new` 来创建一个新的空白 `App`，然后调用它的 `route` 方法为路径 `"/"` 添加一个路由。提供给该路由的处理程序 `web::get().to(get_index)` 会通过调用函数 `get_index` 来处理 HTTP 的 GET 请求。`route` 方法的返回值就是调用它的那个 `App`，不过其现在已经有了新的路由。由于闭包主体的末尾没有分号，因此此 `App` 就是闭包的返回值，可供 `HttpServer` 线程使用。

`get_index` 函数会构建一个 `HttpResponse` 值，该值表示对 HTTP GET / 请求的响应。`HttpResponse::Ok()` 表示 HTTP 200 OK 状态，意味着请求成功。我们会调用它的 `content_type` 方法和 `body` 方法来填入该响应的细节，每次调用都会返回在前一次基础上修改过的 `HttpResponse`。最后会以 `body` 的返回值作为 `get_index` 的返回值。

由于响应文本包含很多双引号，因此我们使用 Rust 的“原始字符串”语法来编写它：首先是字母 `r`、0 到多个井号（`#`）标记、一个双引号，然后是字符串本体，并以另一个双引号结尾，后跟相同数量的 `#` 标记。任何字符都可以出现在原始字符串中而不被转义，包括双引号。事实上，Rust 根本不认识像 `\"` 这样的转义序列。我们总是可以在引号周围使用比文本内容中出现过的 `#` 更多的 `#` 标记，以确保字符串能在期望的地方结束。

编写完 `main.rs` 后，可以使用 `cargo run` 命令来执行为运行它而要做的一切工作：获取所需的 `crate`、编译它们、构建我们自己的程序、将所有内容链接在一起，最后启动 `main.rs`。

```
$ cargo run
  Updating crates.io index
Downloading crates ...
  Downloaded serde v1.0.100
  Downloaded actix-web v1.0.8
  Downloaded serde_derive v1.0.100
...
  Compiling serde_json v1.0.40
  Compiling actix-router v0.1.5
  Compiling actix-http v0.2.10
  Compiling awc v0.2.7
  Compiling actix-web v1.0.8
```

```
Compiling gcd v0.1.0 (/home/jimb/rust/actix-gcd)
  Finished dev [unoptimized + debuginfo] target(s) in 1m 24s
  Running `/home/jimb/rust/actix-gcd/target/debug/actix-gcd`
Serving on http://localhost:3000...
```

此刻，在浏览器中访问给定的 URL 就会看到图 2-1 所示的页面。

但很遗憾，单击“Compute GCD”除了将浏览器导航到一个空白页面外，没有做任何事。为了继续解决这个问题，可以往 App 中添加另一个路由，以处理来自表单的 POST 请求。

现在终于用到我们曾在 Cargo.toml 文件中列出的 `serde crate` 了：它提供了一个便捷工具来协助处理表单数据。首先，将以下 `use` 指令添加到 `src/main.rs` 的顶部：

```
use serde::Deserialize;
```

Rust 程序员通常会将所有的 `use` 声明集中放在文件的顶部，但这并非绝对必要：Rust 允许这些声明以任意顺序出现，只要它们出现在适当的嵌套级别即可。

接下来，定义一个 Rust 结构体类型，用以表示期望从表单中获得的值：

```
#[derive(Deserialize)]
struct GcdParameters {
    n: u64,
    m: u64,
}
```

上述代码定义了一个名为 `GcdParameters` 的新类型，它有两个字段（`n` 和 `m`），每个字段都是一个 `u64`，这是我们的 `gcd` 函数想要的参数类型。

此 `struct` 定义上面的注解是一个属性，就像之前用来标记测试函数的 `#[test]` 属性一样。在类型定义之上放置一个 `#[derive(Deserialize)]` 属性会要求 `serde crate` 在程序编译时检查此类型并自动生成代码，以便从 HTML 表单 POST 提交过来的

格式化数据中解析出此类型的值。事实上，该属性足以让你从几乎任何种类的结构化数据（JSON、YAML、TOML 或许多其他文本格式和二进制格式中的任何一种）中解析 `GcdParameters` 的值。`serde` crate 还提供了一个 `Serialize` 属性，该属性会生成代码来执行相反的操作，获取 `Rust` 值并以结构化的格式序列化它们。

有了这个定义，就可以很容易地编写处理函数了：

```
fn post_gcd(form: web::Form<GcdParameters>) -> HttpResponse {
    if form.n == 0 || form.m == 0 {
        return HttpResponse::BadRequest()
            .content_type("text/html")
            .body("Computing the GCD with zero is boring.");
    }

    let response =
        format!("The greatest common divisor of the numbers {} and
        {} \
                is <b>{}</b>\n",
                form.n, form.m, gcd(form.n, form.m));
    HttpResponse::Ok()
        .content_type("text/html")
        .body(response)
}
```

对于用作 `Actix` 请求处理程序的函数，其参数必须全都是 `Actix` 知道如何从 HTTP 请求中提取出来的类型。`post_gcd` 函数接受一个参数 `form`，其类型为 `web::Form<GcdParameters>`。当且仅当 `T` 可以从 HTML 表单提交过来的数据反序列化时，`Actix` 才能知道该如何从 HTTP 请求中提取任意类型为 `web::Form<T>` 的值。由于我们已经将 `#[derive(Deserialize)]` 属性放在了 `GcdParameters` 类型定义上，`Actix` 可以从表单数据中反序列化它，因此请求处理程序可以要求以 `web::Form<GcdParameters>` 值作为参数。这些类型和函数之间的关系都是在编译期指定的。如果使用了 `Actix` 不知道该如何处理的参数类型来编写处理函数，那么 `Rust` 编译器会直接向你报错。

来看看 `post_gcd` 内部，如果任何一个参数为 0，则该函数会先行返回 HTTP 400 BAD REQUEST 错误，因为如果它们为 0，我们的 `gcd`

函数将崩溃。同时，`post_gcd` 会使用 `format!` 宏来为此请求构造出响应体。`format!` 与 `println!` 很像，但它不会将文本写入标准输出，而是会将其作为字符串返回。一旦获得响应文本，`post_gcd` 就会将其包装在 HTTP 200 OK 响应中，设置其内容类型，并将它返回给请求者。

还必须将 `post_gcd` 注册为表单处理程序。为此，可以将 `main` 函数替换成以下这个版本：

```
fn main() {
    let server = HttpServer::new(|| {
        App::new()
            .route("/", web::get().to(get_index))
            .route("/gcd", web::post().to(post_gcd))
    });

    println!("Serving on http://localhost:3000...");
    server
        .bind("127.0.0.1:3000").expect("error binding server to address")
        .run().expect("error running server");
}
```

这里唯一的变化是添加了另一个 `route` 调用，确立 `web::post().to(post_gcd)` 作为路径 `"/gcd"` 的处理程序。

最后剩下的部分是我们之前编写的 `gcd` 函数，它位于 `actix-gcd/src/main.rs` 文件中。有了它，你就可以中断运行中的服务器，重新构建并启动程序了：

```
$ cargo run
Compiling actix-gcd v0.1.0 (/home/jimb/rust/actix-gcd)
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running `target/debug/actix-gcd`
Serving on http://localhost:3000...
```

这一次，访问 `http://localhost:3000`，输入一些数值，然后单击“Compute GCD”按钮，应该会看到一些实质性结果，如图 2-2 所示。

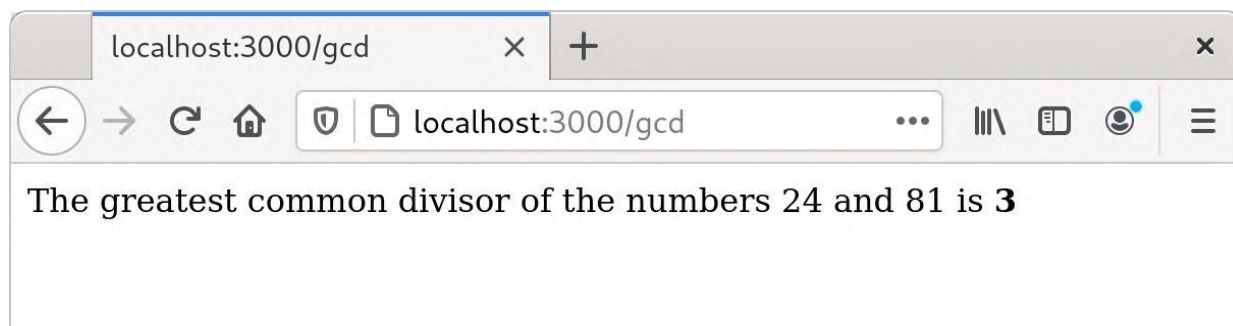


图 2-2：展示计算最大公约数结果的网页

2.6 并发

Rust 的一大优势是它对并发编程的支持。Rust 中用来确保内存安全的那些规则也同样可以让线程在共享内存的时候避免数据竞争。

- 如果使用互斥锁来协调对共享数据结构进行更改的多个线程，那么 Rust 会确保只有持有锁才能访问这些数据，并会在完工后自动释放锁。而在 C 和 C++ 中，互斥锁和它所保护的数据之间的联系只能体现在注释中。
- 如果想在多个线程之间共享只读数据，那么 Rust 能确保你不会意外修改数据。而在 C 和 C++ 中，虽然类型系统也可以帮你解决这个问题，但很容易出错。
- 如果将数据结构的所有权从一个线程转移给另一个线程，那么 Rust 能确保你真的放弃了对它的所有访问权限。而在 C 和 C++ 中，要由你来检查发送线程上的任何代码是否会再次接触数据。如果你弄错了，那么后果可能取决于处理器缓存中正在发生什么，以及你最近对内存进行过多少次写入。我们或多或少都在这方面吃过一点儿苦头。

本节将引导你写出第二个多线程程序。

你已经写完了第一个程序：用 Actix Web 框架实现的最大公约数服务器，它使用线程池来运行请求处理函数。如果服务器同时收到多个请求，那么它就会在多个线程中同时运行 `get_index` 函数和 `post_gcd` 函数。这可能有点儿令人震撼，因为我们在编写这些函数时甚至都没有考虑过并发。但 Rust 能确保这样做是安全的，无论你的服务器变得多么复杂：只要程序编译通过了，就一定不会出现数据竞争。所有 Rust 函数都是线程安全的。

本节的程序绘制了曼德博集（一组分形几何图形，包括著名的海龟图等），这是一种对复数反复运行某个简单函数而生成的分形图。人们通常把“绘制曼德博集”称为易并行算法，因为其线程之间的通信模式非常简单，第 19 章会介绍更复杂的模式，但这里的任务已足以演示一些基本要素了。

首先，创建一个新的 Rust 项目：

```
$ cargo new mandelbrot
      Created binary (application) `mandelbrot` package
$ cd mandelbrot
```

所有代码都将放在 `mandelbrot/src/main.rs` 中，我们将向 `mandelbrot/Cargo.toml` 添加一些依赖项。

在进入并发曼德博实现之前，先来讲一下接下来将要执行的计算。

2.6.1 什么是曼德博集

在阅读代码时，具体了解一下它要执行的任务是很有帮助的，所以，我们可以稍微了解一点儿纯数学。先从一个简单的案例开始，然后添加复杂的细节，直到抵达曼德博集最核心的计算领域。

下面是一个使用 Rust 特有语法实现的 loop 语句无限循环：

```
fn square_loop(mut x: f64) {
    loop {
        x = x * x;
    }
}
```

在现实世界中，Rust 能看出 `x` 从未用来做任何事，因此不会计算它的值。但目前，假设代码能按编写的方式运行。那么 `x` 的值会如何变化呢？对任何小于 1 的数值求平方会使它变得更小，因此它会趋近于 0；1 的平方会得到 1；对大于 1 的数值求平方会使它变大，因此它会趋近于无穷大；对一个负数求平方会先使其变为正数，之后它的变化情况和前面的情况类似，如图 2-3 所示。

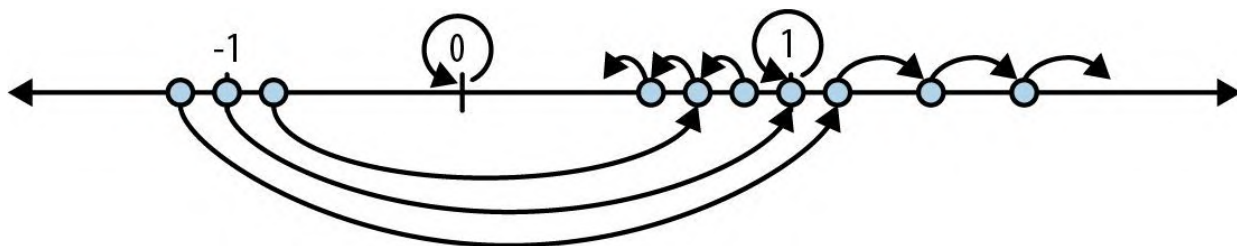


图 2-3：重复对数值求平方的效果

因此，根据传给 `square_loop` 的值，`x` 的取值为 0 或 1、趋近 0 或趋近无穷大。

现在考虑一个略有不同的循环：

```
fn square_add_loop(c: f64) {  
    let mut x = 0.;  
    loop {  
        x = x * x + c;  
    }  
}
```

这一次，`x` 从 0 开始，我们通过对它求平方后再加上 `c` 来调整它在每次迭代中的进度。这更难看出 `x` 的变化情况了，但通过一些实验会发现，如果 `c` 大于 0.25 或小于 -2.0，那么 `x` 最终会变得无限大，否则，它就会停留在 0 附近的某个地方。

下一个问题：如果不再使用 `f64` 值而是改用复数做同样的循环会怎样？`crates.io` 上的 `num crate` 已经提供了开箱即用的复数类型，因此要在程序的 `Cargo.toml` 文件的 `[dependencies]` 部分添加一行 `num`。这是迄今为止的整个文件（稍后会添加更多）：

```
[package]  
name = "mandelbrot"  
version = "0.1.0"  
edition = "2021"  
  
# 请到“The Cargo Book”查看更多的键及其定义  
  
[dependencies]  
num = "0.4"
```

现在可以编写此循环的倒数第二个版本了：

```
use num::Complex;  
  
fn complex_square_add_loop(c: Complex<f64>) {  
    let mut z = Complex { re: 0.0, im: 0.0 };  
}
```

```
    loop {  
        z = z * z + c;  
    }  
}
```

传统上会用 z 来代表复数，因此我们重命名了循环变量。表达式 `Complex { re: 0.0, im: 0.0 }` 是使用 `num crate` 的 `Complex` 类型编写复数 0 的方式。`Complex` 是一种 Rust 结构体类型（或 `struct`），其定义如下：

```
struct Complex<T> {  
    /// 复数的实部  
    re: T,  
  
    /// 复数的虚部  
    im: T,  
}
```

上述代码定义了一个名为 `Complex` 的结构体，该结构体有两个字段，即 `re` 和 `im`。`Complex` 是一种泛型结构体：可以把在类型名称之后的 `<T>` 读作“对于任意类型 `T`”。例如，`Complex<f64>` 是一个复数，其 `re` 字段和 `im` 字段为 `f64` 值，`Complex<f32>` 则使用 32 位浮点数，等等。根据此定义，像 `Complex { re: 0.24, im: 0.3 }` 这样的表达式就会生成一个 `Complex` 值，其 `re` 字段已初始化为 0.24，`im` 字段已初始化为 0.3。

`num crate` 支持用 `*`、`+` 和其他算术运算符来处理 `Complex` 值，因此该函数的其余部分仍然像之前的版本那样工作，只是它会将数值视作复平面上而不是实数轴上的点进行运算。第 12 章会讲解如何让 Rust 的运算符与自定义类型协同工作。

我们终于抵达了纯数学之旅的终点。曼德博集的定义是：令 z 不会“飞到”无穷远的复数 c 的集合。我们最初的简单平方循环是可以预测的：任何大于 1 或小于 -1 的数值都会“飞”出去。把 $+c$ 放入每次迭代中会使变化情况更难预测：正如前面所说，大于 0.25 或小于 -2.0 的 c 值会导致 z “飞”出去。但是将此游戏推广到复数就会生成真正奇异而美丽的图案，这就是我们所要绘制的分形图。

由于复数 `c` 具有实部 `c.re` 和虚部 `c.im`，因此可以把它们视为笛卡儿平面上某个点的 `x` 坐标和 `y` 坐标，如果 `c` 在曼德博集中，就在其中用黑色着色，否则就用浅色。因此，对于图像中的每个像素，必须在复平面上的相应点位运行前面的循环，看看它是逃逸到无穷远还是永远绕着原点运行，并相应地将其着色。

无限循环需要一段时间才能完成，但是对缺乏耐心的人来说有两个小技巧。首先，如果不再永远运行循环而只是尝试一些有限次数的迭代，事实证明仍然可以获得该集合的一个不错的近似值。我们需要多少次迭代取决于想要绘制的边界的精度。其次，业已证明，一旦 `z` 离开了以原点为中心的半径为 2 的圆，它最终就一定会“飞到”无穷远的地方。所以下面是循环的最终版本，也是程序的核心：

```
use num::Complex;

/// 尝试测定`c`是否位于曼德博集中，使用最多`limit`次迭代来判定
///
/// 如果`c`不是集合成员之一，则返回`Some(i)`，其中的`i`是`c`离开以原点
/// 为中心的半径为2的圆时所需的迭代次数。如果`c`似乎是集合成员之一（确
/// 切而言是达到了迭代次数限制但仍然无法证明`c`不是成员），则返回`None`
fn escape_time(c: Complex<f64>, limit: usize) -> Option<usize> {
    let mut z = Complex { re: 0.0, im: 0.0 };
    for i in 0..limit {
        if z.norm_sqr() > 4.0 {
            return Some(i);
        }
        z = z * z + c;
    }

    None
}
```

此函数会接受两个参数：`c` 是我们要测试其是否属于曼德博集的复数，`limit` 是要尝试的迭代次数上限，一旦超出这个次数就放弃并认为 `c` 可能是成员。

该函数的返回值是一个 `Option<usize>`。Rust 的标准库中对 `Option` 类型的定义如下所示：

```
enum Option<T> {  
    None,  
    Some(T),  
}
```

`Option` 是一种枚举类型（通常称为“枚举”，`enum`），因为它的定义枚举了这个类型的值可能是几种变体之一：对于任意类型 `T`，`Option<T>` 类型的值要么是 `Some(v)`，其中 `v` 的类型为 `T`；要么是 `None`，表示没有可用的 `T` 值。与之前讨论的 `Complex` 类型一样，`Option` 是一种泛型类型：你可以使用 `Option<T>` 来表示任何一种类型 `T` 的可选值。

在这个例子中，`escape_time` 返回一个 `Option<usize>` 来指示 `c` 是否在曼德博集中——如果不在，是迭代了多少次才发现的。如果 `c` 不在集合中，那么 `escape_time` 就会返回 `Some(i)`，其中 `i` 是 `z` 在离开半径为 2 的圆之前的迭代次数。否则，`c` 显然在集合中，并且 `escape_time` 返回 `None`。

```
for i in 0..limit {
```

前面的示例展示了如何用 `for` 循环遍历命令行参数和向量元素，这个 `for` 循环则只是遍历从 0 开始到 `limit`（不含）的整数范围。

`z.norm_sqr()` 方法调用会返回 `z` 与原点距离的平方。要判断 `z` 是否已经离开半径为 2 的圆，不必计算平方根，只需将此距离的平方与 4.0 进行比较即可，这样速度更快。

你可能已经注意到我们使用了 `///` 来标记函数定义上方的注释行，`Complex` 结构体成员上方的注释同样以 `///` 开头。这些叫作文档型注释，`rustdoc` 实用程序知道如何解析它们和它们所描述的代码，并生成在线文档。Rust 的标准库文档就是以这种形式编写的。第 8 章会详细讲解文档型注释。

该程序的其余部分所“关心”的是决定以何种分辨率绘制此集合中的哪个部分，并将此项工作分发给多个线程以加快计算速度。

2.6.2 解析并配对命令行参数

该程序会接受几个命令行参数来控制我们要写入的图像的分辨率以及要绘制曼德博集里哪部分的图像。由于这些命令行参数遵循着一种共同的格式，因此我们写了一个解析它们的函数：

```
use std::str::FromStr;

/// 把字符串`s`（形如`"400x600"`或`"1.0,0.5"`）解析成一个坐标对
///
/// 具体来说，`s`应该具有<left><sep><right>的格式，其中<sep>是由
/// `separator`
/// 参数给出的字符，而<left>和<right>是可以被`T::from_str`解析的字符串。
/// `separator`必须是ASCII字符
///
/// 如果`s`具有正确的格式，就返回`Some<(x, y)>`；如果无法正确解析，就返回
/// `None`
fn parse_pair<T: FromStr>(s: &str, separator: char) -> Option<(T,
T)> {
    match s.find(separator) {
        None => None,
        Some(index) => {
            match (T::from_str(&s[..index]), T::from_str(&s[index +
1..])) {
                (Ok(l), Ok(r)) => Some((l, r)),
                _ => None
            }
        }
    }
}

#[test]
fn test_parse_pair() {
    assert_eq!(parse_pair::<i32>("", ' '), None);
    assert_eq!(parse_pair::<i32>("10,", ' '), None);
    assert_eq!(parse_pair::<i32>(",10", ' '), None);
    assert_eq!(parse_pair::<i32>("10,20", ' '), Some((10, 20)));
    assert_eq!(parse_pair::<i32>("10,20xy", ' '), None);
    assert_eq!(parse_pair::<f64>("0.5x", 'x'), None);
    assert_eq!(parse_pair::<f64>("0.5x1.5", 'x'), Some((0.5,
1.5)));
}
```

`parse_pair` 的定义是一个泛型函数：

```
fn parse_pair<T: FromStr>(s: &str, separator: char) -> Option<(T, T)> {
```

可以把 `<T: FromStr>` 子句读作“对于实现了 `FromStr` 特型的任意类型 `T`……”，这样就能高效地一次定义出整个函数家族：

`parse_pair::<i32>` 是能解析一对 `i32` 值的函数、

`parse_pair::<f64>` 是能解析一对 `f64` 浮点值的函数，等等。这很像 C++ 中的函数模板。Rust 程序员会将 `T` 称作 `parse_pair` 的**类型参数**。当使用泛型函数时，Rust 通常能帮我们推断出类型参数，并且我们不必像这里的测试代码那样把它们明确写出来。

我们的返回类型是 `Option<(T, T)>`：它或者是 `None`，或者是一个值 `Some((v1, v2))`，其中 `(v1, v2)` 是由两个 `T` 类型的值构成的元组。`parse_pair` 函数没有使用显式 `return` 语句，因此它的返回值是其函数体中最后一个（也是唯一的一个）表达式的值：

```
match s.find(separator) {
    None => None,
    Some(index) => {
        ...
    }
}
```

`String` 类型的 `find` 方法会在字符串中搜索与 `separator` 相匹配的字符。如果 `find` 返回 `None`，那么就意味着字符串中没有出现分隔符，这样整个 `match` 表达式的计算结果就为 `None`，表明解析失败。否则，`index` 值就是此分隔符在字符串中的位置。

```
match (T::from_str(&s[..index]), T::from_str(&s[index + 1..])) {
    (Ok(l), Ok(r)) => Some((l, r)),
    _ => None
}
```

这里初步展现了 `match`（匹配）表达式的强大之处。`match` 的参数是如下元组表达式：

```
(T::from_str(&s[..index]), T::from_str(&s[index + 1..]))
```

表达式 `&s[..index]` 和 `&s[index + 1..]` 都是字符串的切片，分别位于分隔符之前和之后。类型参数 `T` 的关联函数 `from_str` 会获取其中的每一个元素并尝试将它们解析为类型 `T` 的值，从而生成结果元组。下面是我们要匹配的目标：

```
(Ok(l), Ok(r)) => Some((l, r)),
```

仅当此元组的两个元素都是 `Result` 类型的 `Ok` 变体时，该模式才能匹配上，这表明两个解析都成功了。如果是这样，那么 `Some((l, r))` 就是匹配表达式的值，也就是函数的返回值。

```
_ => None
```

通配符模式 `_` 会匹配任意内容并忽略其值。如果运行到此处，则表明 `parse_pair` 已然失败，因此其值为 `None`，并继而作为本函数的返回值。

现在有了 `parse_pair`，就很容易编写一个函数来解析一对浮点坐标并将它们作为 `Complex<f64>` 值返回：

```
/// 把一对用逗号分隔的浮点数解析为复数
fn parse_complex(s: &str) -> Option<Complex<f64>> {
    match parse_pair(s, ',') {
        Some((re, im)) => Some(Complex { re, im }),
        None => None
    }
}

#[test]
fn test_parse_complex() {
    assert_eq!(parse_complex("1.25,-0.0625"),
               Some(Complex { re: 1.25, im: -0.0625 }));
    assert_eq!(parse_complex(", -0.0625"), None);
}
```

`parse_complex` 函数调用了 `parse_pair`，如果坐标解析成功则构建一个 `Complex` 值，如果失败则传回给它的调用者。

如果你很细心，可能会注意到我们用了简写形式来构建 `Complex` 值。用同名变量来初始化结构体中的字段是很常见的写法，所以 Rust 不会强迫你写成 `Complex { re: re, im: im }`，而会让你简写成 `Complex { re, im }`。这是从 JavaScript 和 Haskell 中的类似写法借鉴来的。

2.6.3 从像素到复数的映射

我们的程序需要在两个彼此相关的坐标空间中运行：输出图像中的每个像素对应于复平面上的一个点。这两个空间之间的关系取决于要绘制曼德博集的哪一部分以及所请求图像的分辨率，这些都要通过命令行参数指定。以下函数会将图像空间转换为复数空间：

```
/// 给定输出图像中像素的行和列，返回复平面中对应的坐标
///
/// `bounds` 是一个 `pair`，给出了图像的像素宽度和像素高度。`pixel` 是表示该
/// 图像中特定像素的 (column, row) 二元组。`upper_left` 参数和 `lower_right`
/// 参数是在复平面中表示指定图像覆盖范围的点
fn pixel_to_point(bounds: (usize, usize),
                  pixel: (usize, usize),
                  upper_left: Complex<f64>,
                  lower_right: Complex<f64>)
    -> Complex<f64>
{
    let (width, height) = (lower_right.re - upper_left.re,
                           upper_left.im - lower_right.im);
    Complex {
        re: upper_left.re + pixel.0 as f64 * width / bounds.0 as
f64,
        im: upper_left.im - pixel.1 as f64 * height / bounds.1 as
f64
        // 为什么这里要用减法？这是因为在屏幕坐标系中 pixel.1 是
        // 向下递增的，但复数的虚部是向上递增的
    }
}

#[test]
fn test_pixel_to_point() {
    assert_eq!(pixel_to_point((100, 200), (25, 175),
                              Complex { re: -1.0, im: 1.0 },
                              Complex { re: 1.0, im: -1.0 }),
               Complex { re: -0.5, im: -0.75 });
}
```


图 2-4 说明了 `pixel_to_point` 所执行的计算规则。

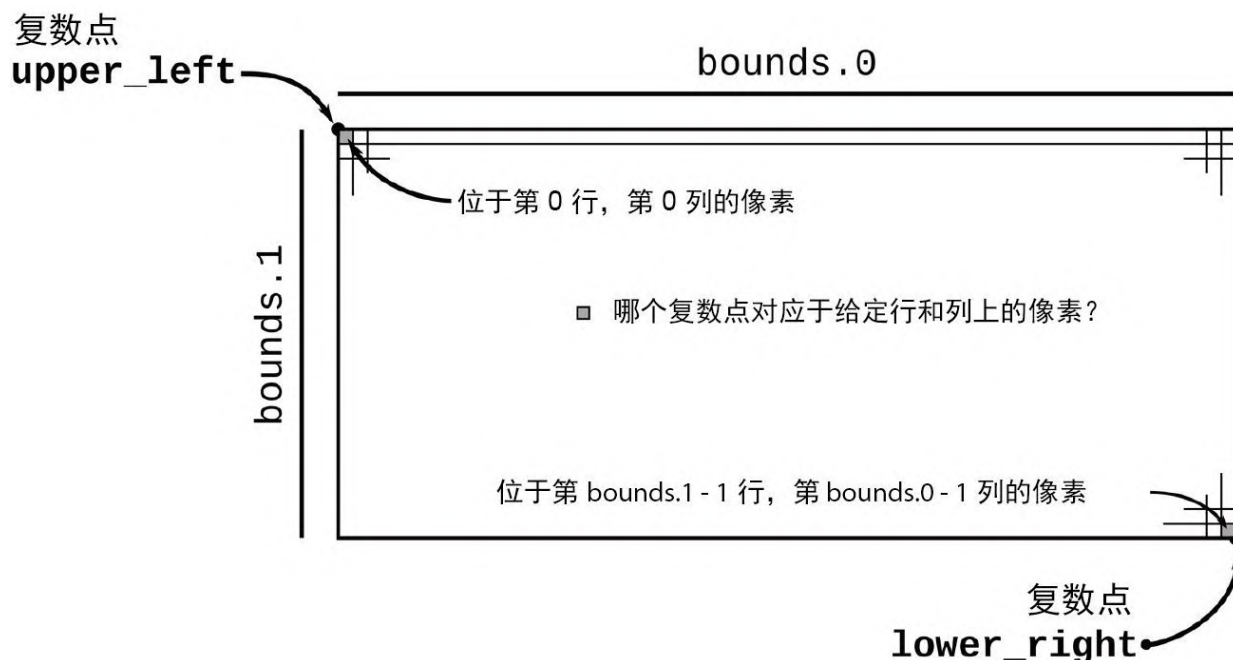


图 2-4: 复平面与图像像素的对应关系

`pixel_to_point` 的代码只是简单的计算，就不详细解释了。但是，有几点需要指出一下。下列形式的表达式引用的是元组中的元素：

```
pixel.0
```

这里引用的是 `pixel` 元组的第一个元素。

```
pixel.0 as f64
```

这是 Rust 的类型转换语法：这会将 `pixel.0` 转换为 `f64` 值。与 C 和 C++ 不同，Rust 通常会拒绝在数值类型之间进行隐式转换，因此你必须写出所需的转换。这可能有些烦琐，但明确说明发生了哪些转换以及发生于何时是非常有帮助的。隐式整数转换看似“人畜无害”，但从历史上看，它们一直是现实世界 C 和 C++ 代码中缺陷和安全漏洞的常见来源。

2.6.4 绘制曼德博集

要绘制出曼德博集，只需对复平面上的每个点调用 `escape_time`，并根据其结果为图像中的像素着色：

```
/// 将曼德博集对应的矩形渲染到像素缓冲区中
///
/// `bounds` 参数会给出缓冲区 `pixels` 的宽度和高度，此缓冲区的每字节都
/// 包含一个灰度像素。`upper_left` 参数和 `lower_right` 参数分别指定了
/// 复平面中对应于像素缓冲区左上角和右下角的点
fn render(pixels: &mut [u8],
          bounds: (usize, usize),
          upper_left: Complex<f64>,
          lower_right: Complex<f64>)
{
    assert!(pixels.len() == bounds.0 * bounds.1);

    for row in 0..bounds.1 {
        for column in 0..bounds.0 {
            let point = pixel_to_point(bounds, (column, row),
                                       upper_left, lower_right);
            pixels[row * bounds.0 + column] =
                match escape_time(point, 255) {
                    None => 0,
                    Some(count) => 255 - count as u8
                };
        }
    }
}
```

此刻，这一切看起来都很熟悉。

```
pixels[row * bounds.0 + column] =
    match escape_time(point, 255) {
        None => 0,
        Some(count) => 255 - count as u8
    };
```

如果 `escape_time` 认为该 `point` 属于本集合，`render` 就会将相应像素的颜色渲染为黑色（0）。否则，`render` 会将需要更长时间才能逃离圆圈的数值渲染为较深的颜色。

2.6.5 写入图像文件

image crate 提供了读取和写入各种图像格式的函数，以及一些基本的图像处理函数。特别是，此 crate 包含一个 PNG 图像文件格式的编码器，该程序使用这个编码器来保存计算的最终结果。为了使用 image，请将下面这行代码添加到 Cargo.toml 的 [dependencies] 部分：

```
image = "0.13.0"
```

然后可以这样写：

```
use image::ColorType;
use image::png::PNGEncoder;
use std::fs::File;

/// 把`pixels`缓冲区（其尺寸由`bounds`给出）写入名为`filename`的文件中
fn write_image(filename: &str, pixels: &[u8], bounds: (usize,
    usize))
    -> Result<(), std::io::Error>
{
    let output = File::create(filename)?;

    let encoder = PNGEncoder::new(output);
    encoder.encode(pixels,
        bounds.0 as u32, bounds.1 as u32,
        ColorType::Gray(8))?;

    Ok(())
}
```

这个函数的操作一目了然：它打开一个文件并尝试将图像写入其中。我们给编码器传入来自 pixels 的实际像素数据、来自 bounds 的宽度和高度，然后是最后一个参数，以说明如何解释 pixels 中的字节：值 ColorType::Gray(8) 表示每字节都是一个 8 位的灰度值。

这些也同样一目了然。该函数值得一看的地方在于当出现问题时它是如何处理的。一旦遇到错误，就要将错误报告给调用者。正如之前提过的，Rust 中的容错函数应该返回一个 Result 值，成功时为

`Ok(s)`（其中 `s` 是成功值），失败时为 `Err(e)`（其中 `e` 是错误代码）。那么 `write_image` 的成功类型和错误类型是什么呢？

当一切顺利时，`write_image` 函数只是把所有值得一看的东西都写到了文件中，没有任何有用的返回值。所以它的成功类型就是单元（`unit`）类型 `()`，而如此命名是因为这个类型只有一个值 `()`。单元类型类似于 C 和 C++ 中的 `void`。

如果发生错误，那么可能是因为 `File::create` 无法创建文件或 `encoder.encode` 无法将图像写入其中，此 I/O 操作就会返回错误代码。`File::create` 的返回类型是 `Result<std::fs::File, std::io::Error>`，而 `encoder.encode` 的返回类型是 `Result<(), std::io::Error>`，所以两者共享着相同的错误类型，即 `std::io::Error`。`write_image` 函数也应该这么做。在任何情况下，失败都应导致立即返回，并传出用以描述错误原因的 `std::io::Error` 值。

所以，为了正确处理 `File::create` 的结果，需要 `match` 它的返回值，如下所示：

```
let output = match File::create(filename) {  
    Ok(f) => f,  
    Err(e) => {  
        return Err(e);  
    }  
};
```

成功时，就将 `output` 赋值为 `Ok` 值中携带的 `File`。失败时，就将错误透传给调用者。

这种 `match` 语句在 Rust 中是一种非常常见的模式，所以该语言提供了 `?` 运算符作为它的简写形式。因此，与其每次在尝试可能失败的事情时都明确地写出这个逻辑，不如使用以下等效且更易读的语句：

```
let output = File::create(filename)?;
```

如果 `File::create` 失败，那么 `?` 运算符就会从 `write_image` 返回，并传出此错误。否则，`output` 就会持有已成功打开的 `File`。



新手常犯的一个错误就是试图在 `main` 函数中使用 `?`。但是，由于 `main` 本身不返回值，因此这样做行不通。应该使用 `match` 语句，或者像 `unwrap` 和 `expect` 这样的简写方法。还可以选择简单地把 `main` 改成返回一个 `Result`，稍后会介绍这种方式。

2.6.6 并发版曼德博程序

万事俱备，可以展示一下 `main` 函数了，我们可以在其中利用并发来完成任务。为简单起见，先来看一个非并发版本：

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();

    if args.len() != 5 {
        eprintln!("Usage: {} FILE PIXELS UPPERLEFT LOWERRIGHT",
            args[0]);
        eprintln!("Example: {} mandel.png 1000x750 -1.20,0.35\n-1,0.20",
            args[0]);
        std::process::exit(1);
    }

    let bounds = parse_pair(&args[2], 'x')
        .expect("error parsing image dimensions");
    let upper_left = parse_complex(&args[3])
        .expect("error parsing upper left corner point");
    let lower_right = parse_complex(&args[4])
        .expect("error parsing lower right corner point");

    let mut pixels = vec![0; bounds.0 * bounds.1];

    render(&mut pixels, bounds, upper_left, lower_right);

    write_image(&args[1], &pixels, bounds)
```

```
}  
    .expect("error writing PNG file");  
}
```

将命令行参数收集到一个 `String` 向量中后，我们会解析每个参数，然后开始计算。

```
let mut pixels = vec![0; bounds.0 * bounds.1];
```

宏调用 `vec![v; n]` 创建了一个 `n` 元素长的向量，其元素会被初始化为 `v`，因此前面的代码创建了一个长度为 `bounds.0 * bounds.1` 的全零向量，其中 `bounds` 是从命令行解析得来的图像分辨率。我们将使用此向量作为单字节灰度像素值的矩形数组，如图 2-5 所示。

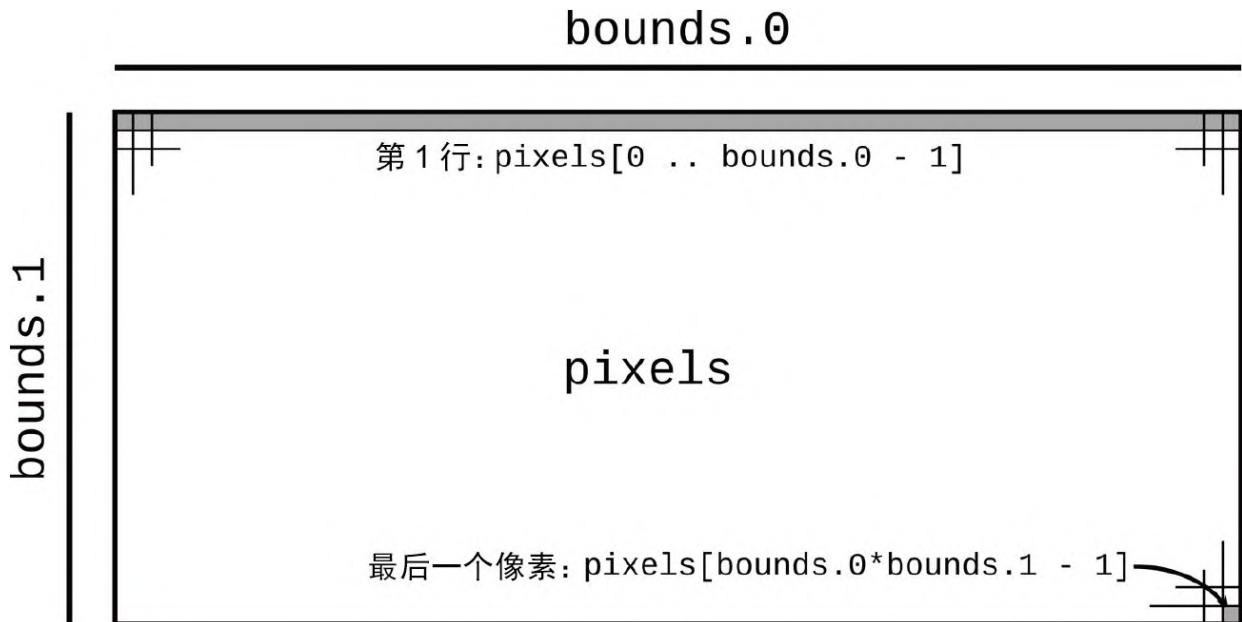


图 2-5：使用向量作为矩形像素阵列

下一行值得关注的代码是：

```
render(&mut pixels, bounds, upper_left, lower_right);
```

这会调用 `render` 函数来实际计算图像。表达式 `&mut pixels` 借用了一个对像素缓冲区的可变引用，以允许 `render` 用计算出来的灰

度值填充它，不过 `pixels` 仍然是此向量的拥有者。其余的参数传入了图像的维度和要绘制的复平面矩形。

```
write_image(&args[1], &pixels, bounds)
    .expect("error writing PNG file");
```

最后，将这个像素缓冲区作为 PNG 文件写入磁盘。在这个例子中，我们向缓冲区传入了一个共享（不可变）引用，因为 `write_image` 不需要修改缓冲区的内容。

此时，可以在发布模式下构建和运行程序，它启用了许多强力的编译器优化，几秒后会在文件 `mandel.png` 中写入一个漂亮的图像：

```
$ cargo build --release
  Updating crates.io index
  Compiling autocfg v1.0.1
  ...
  Compiling image v0.13.0
  Compiling mandelbrot v0.1.0 ($RUSTBOOK/mandelbrot)
  Finished release [optimized] target(s) in 25.36s
$ time target/release/mandelbrot mandel.png 4000x3000 -1.20,0.35
-1,0.20
real    0m4.678s
user    0m4.661s
sys     0m0.008s
```

此命令会创建一个名为 `mandel.png` 的文件，你可以使用系统的图像查看器或在 Web 浏览器中查看该文件。如果一切顺利，它应该如图 2-6 所示。

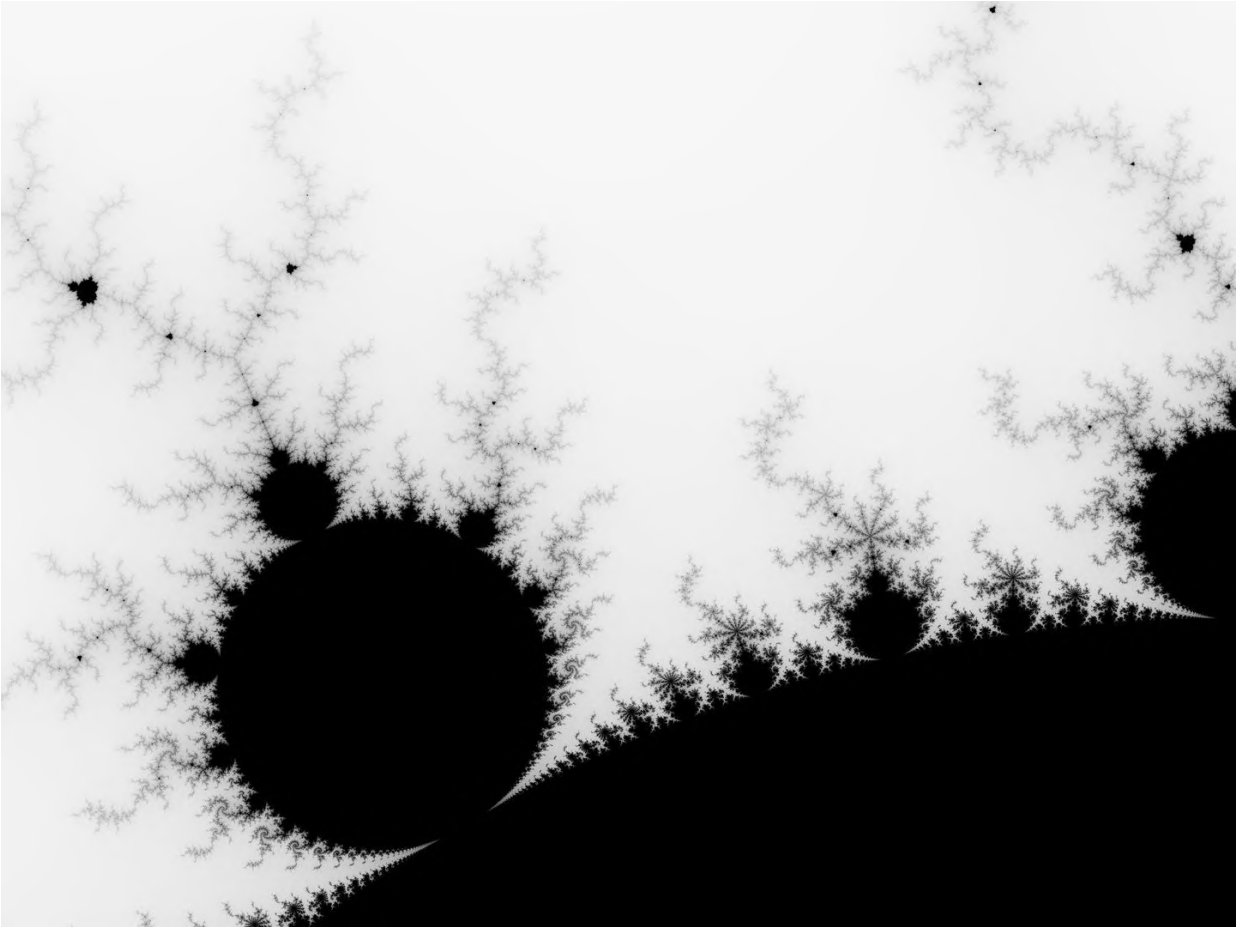


图 2-6：并行曼德博程序的结果

在之前的记录中，我们使用过 Unix 的 `time` 程序来分析程序的运行时间——对图像的每个像素运行曼德博计算总共需要大约 5 秒。但是几乎所有的现代机器都有多个处理器核心，而这个程序只使用了一个。如果可以将此工作分派给机器提供的所有计算资源，则应该能更快地画完图像。

为此，可以将图像分成多个部分（每个处理器一个），并让每个处理器为分派给它的像素着色。为简单起见，可以将其分成一些水平条带，如图 2-7 所示。当所有处理器都完成后，可以将像素写入磁盘中。

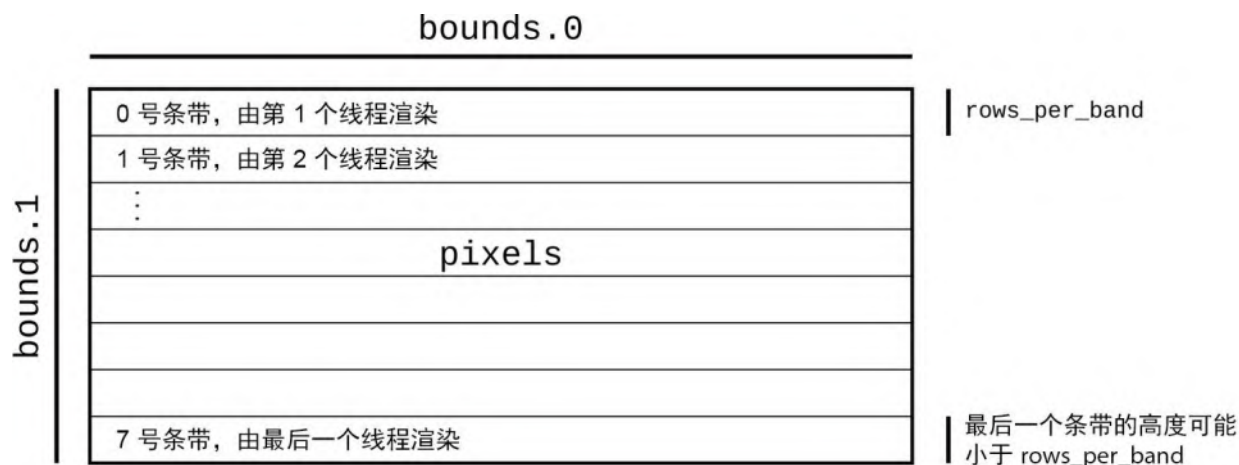


图 2-7：将像素缓冲区划分为一些条带以进行并行渲染

`crossbeam crate` 提供了许多有价值的并发设施，包括这里正需要的一个作用域线程设施。要使用此设施，必须将下面这行代码添加到 `Cargo.toml` 文件中：

```
crossbeam = "0.8"
```

然后要找出调用 `render` 的代码行并将其替换为以下内容：

```
let threads = 8;
let rows_per_band = bounds.1 / threads + 1;

{
    let bands: Vec<&mut [u8]> =
        pixels.chunks_mut(rows_per_band * bounds.0).collect();
    crossbeam::scope(|spawner| {
        for (i, band) in bands.into_iter().enumerate() {
            let top = rows_per_band * i;
            let height = band.len() / bounds.0;
            let band_bounds = (bounds.0, height);
            let band_upper_left =
                pixel_to_point(bounds, (0, top), upper_left,
lower_right);
            let band_lower_right =
                pixel_to_point(bounds, (bounds.0, top + height),
                                upper_left, lower_right);

            spawner.spawn(move |_| {
                render(band, band_bounds, band_upper_left,
band_lower_right);
            });
        }
    });
}
```

```
        });  
    }  
    }).unwrap();  
}
```

仍以刚才的方式分步进行讲解：

```
let threads = 8;  
let rows_per_band = bounds.1 / threads + 1;
```

这里我们决定使用 8 个线程。⁴然后会计算每个条带应该有多少行像素。我们向上舍入行数以确保条带覆盖整个图像，即使其高度并不是 `threads` 的整数倍。

⁴`num_cpus` crate 提供了一个返回当前系统可用 CPU 数量的函数。

```
let bands: Vec<&mut [u8]> =  
    pixels.chunks_mut(rows_per_band * bounds.0).collect();
```

这里我们将像素缓冲区划分为几个条带。缓冲区的 `chunks_mut` 方法会返回一个迭代器，该迭代器会生成此缓冲区的可变且不重叠的切片，每个切片都包含 `rows_per_band * bounds.0` 个像素，换句话说，`rows_per_band` 包含整行的像素。`chunks_mut` 生成的最后一个切片包含的行数可能少一些，但每一行都包含同样数量的像素。最后，此迭代器的 `collect` 方法会构建一个向量来保存这些可变且不重叠的切片。

现在可以使用 `crossbeam` 库了：

```
crossbeam::scope(|spawner| {  
    ...  
}).unwrap();
```

参数 `|spawner| { ... }` 是 Rust 闭包，它需要一个参数 `spawner`。请注意，与使用 `fn` 声明的函数不同，无须声明闭包参数的类型，Rust 将推断它们及其返回类型。在这里，`crossbeam::scope` 调用了此闭包，并将一个值作为 `spawner` 参

数传给闭包，以便闭包使用 `spawner` 来创建新线程。`crossbeam::scope` 函数会等待所有线程执行完毕后返回。这种机制能让 Rust 确保这些线程不会在 `pixels` 超出作用域后再访问分配给自己的那部分，并能让我们确保当 `crossbeam::scope` 返回时，图像的计算已然完成。如果一切顺利，那么 `crossbeam::scope` 就会返回 `Ok(())`，但如果我们启动的任何线程发生了 `panic`，则它会返回一个 `Err`。我们会对该 `Result` 调用 `unwrap`，这样一来，在那种情况下我们也会发生 `panic`，并且用户会收到报告。

```
for (i, band) in bands.into_iter().enumerate() {
```

在这里，我们遍历了像素缓冲区的各个条带。`into_iter()` 迭代器会为循环体的每次迭代赋予独占一个条带的所有权，确保一次只有一个线程可以写入它（第 5 章会详细解释 `into_iter()` 迭代器的工作原理）。然后，枚举适配器生成了一些元组，将向量中的元素与其索引配对。

```
let top = rows_per_band * i;
let height = band.len() / bounds.0;
let band_bounds = (bounds.0, height);
let band_upper_left =
    pixel_to_point(bounds, (0, top), upper_left, lower_right);

let band_lower_right =
    pixel_to_point(bounds, (bounds.0, top + height),
                    upper_left, lower_right);
```

给定索引和条带的实际大小（回想一下，最后一个条带可能比其他条带矮），可以生成 `render` 需要的一个边界框，但它只会引用缓冲区的这个条带，而不是整个图像。同样，我们会重新调整渲染器的 `pixel_to_point` 函数的用途，以找出条带的左上角和右下角落在复平面上的位置。

```
spawner.spawn(move |_| {
    render(band, band_bounds, band_upper_left, band_lower_right);
});
```

最后，创建一个线程，运行 `move |_| { ... }` 闭包。前面的 `move` 关键字表示这个闭包会接手它所用变量的所有权，特别是，只有此闭包才能使用可变切片 `band`。参数列表 `|_|` 意味着闭包会接受一个参数，但不使用它（另一个用以启动嵌套线程的启动器）。

如前所述，`crossbeam::scope` 调用会确保所有线程在它返回之前都已完成，这意味着将图像保存到文件中是安全的，这就是我们下一步要做的。

2.6.7 运行曼德博绘图器

我们在这个程序中使用了几个外部 `crate`：`num` 用于复数运算，`image` 用于写入 PNG 文件，`crossbeam` 用于提供“作用域线程创建”原语。下面是包含所有这些依赖项的最终 `Cargo.toml` 文件：

```
[package]
name = "mandelbrot"
version = "0.1.0"
edition = "2021"

[dependencies]
num = "0.4"
image = "0.13"
crossbeam = "0.8"
```

接下来就可以构建并运行程序了：

```
$ cargo build --release
    Updating crates.io index
   Compiling crossbeam-queue v0.3.2
   Compiling crossbeam v0.8.1
   Compiling mandelbrot v0.1.0 ($RUSTBOOK/mandelbrot)
    Finished release [optimized] target(s) in #.## secs
$ time target/release/mandelbrot mandel.png 4000x3000 -1.20,0.35
-1,0.20
real    0m1.436s
user    0m4.922s
sys     0m0.011s
```

这里我们再次使用 `time` 来查看程序运行所需的时间，请注意，尽管我们仍然花费了将近 5 秒的处理器时间，但实际运行时间仅为 1.5 秒左右。你可以通过注释掉执行此操作的代码并再次进行测量来验证这部分时间是否花在了写入图像文件上。在测试此代码的笔记本电脑上，并发版本将曼德博计算时间缩短了近 3/4。第 19 章会展示如何对此做实质性改进。

和以前一样，该程序会创建一个名为 `mandel.png` 的文件。有了这个更快的版本，你可以根据自己的喜好更改命令行参数，更轻松地探索曼德博集了。

2.6.8 大“安”无形

这个并行程序与用任何其他语言写出来的程序并没有本质区别：我们将像素缓冲区的片段分给不同的处理器，由每个处理器单独处理，并在它们都完工时展示结果。那么 Rust 的并发支持有什么独到之处呢？

这里并没有展示那些被编译器一票否决的 Rust 程序。本章中展示的代码能正确地在线程之间对缓冲区进行分区，但是这些代码的许多小型变体将无法正确进行分区（因此会导致数据竞争），不过这些变体里没有一个能逃过 Rust 编译器的静态检查。C 编译器或 C++ 编译器将乐于帮助你探索具有微妙数据竞争的广阔程序空间，而 Rust 会预先告诉你什么时候可能出错。

第 4 章和第 5 章会讲解 Rust 的内存安全规则。第 19 章会讲解这些规则如何确保适当的安全并发环境。

2.7 文件系统与命令行工具

Rust 在命令行工具领域构筑了重要的基本应用场景。作为一种现代、安全、快速的系统编程语言，它为程序员提供了一个工具箱，他们可以用这个工具箱组装出灵活的命令行界面，从而复现或扩展现有工具的功能。例如，bat 命令⁵就提供了一个支持语法高亮的替代方案 cat，并内置了对分页工具的支持，而 hyperfine 可以自动对任何通过命令或管道运行的程序执行基准测试。

⁵ 这是一个跨平台的命令行工具，可以用来浏览和交互式地查看文件内容。——译者注

虽然如此复杂的内容已经超出了本书的范畴，但 Rust 可以让你轻松步入符合工效学的命令行领域。本节将向你展示如何构建自己的搜索与替换工具，并内置丰富多彩的输出和友好的错误消息。

首先，创建一个新的 Rust 项目：

```
$ cargo new quickreplace
    Created binary (application) `quickreplace` package
$ cd quickreplace
```

我们的程序要用到另外两个 crate：用于在终端中创建彩色输出的 text-colorizer 以及执行实际搜索和替换的 regex。和以前一样，将这些 crate 放在 Cargo.toml 中，告诉 cargo 我们需要它们：

```
[package]
name = "quickreplace"
version = "0.1.0"
edition = "2021"

# 请到“The Cargo Book”查看更多的键及其定义

[dependencies]
text-colorizer = "1"
regex = "1"
```

凡是达到 1.0 版的 Rust crate 都会遵循“语义化版本控制”规则：在主版本号 1 发生变化之前，所有更新都应当在兼容前序版本的基础上扩展。因此，如果针对某个 crate 的 1.2 版测试过我们的程序，那它应该仍然适用于 1.3、1.4 等版本，但 2.0 版可能会引入不兼容的变更。如果在 Cargo.toml 文件中只是请求版本 "1" 的 crate，那么 Cargo 就会使用 2.0 之前的 crate 里最新的可用版本。

2.7.1 命令行界面

这个程序的界面非常简单。它有 4 个参数：要搜索的字符串（或正则表达式）、要替换成的字符串（或正则表达式）、输入文件的名称和输出文件的名称。我们将从包含这些参数的结构体开始写 main.rs 文件：

```
#[derive(Debug)]
struct Arguments {
    target: String,
    replacement: String,
    filename: String,
    output: String,
}
```

`#[derive(Debug)]` 属性会让编译器生成一些额外的代码，这能让我们在 `println!` 中使用 `{:?}` 来格式化 `Arguments` 结构体。

如果用户输入的参数个数不对，那么通常会打印出一份关于如何使用本程序的简单说明。我们会使用一个名为 `print_usage` 的简单函数来完成此操作，并从 `text-colorizer` 导入所有内容，以便为这些输出添加一些颜色：

```
use text_colorizer::*;

fn print_usage() {
    eprintln!("{}", - change occurrences of one string into
another",
                "quickreplace".green());
    eprintln!("Usage: quickreplace <target> <replacement> <INPUT>
<OUTPUT>");
}
```

只要将 `.green()` 添加到字符串字面量的末尾，就可以生成包裹在适当 ANSI 转义码中的字符串，从而在终端模拟器中显示为绿色。然后，在打印之前将生成的字符串插到信息中的其他部分。

现在可以开始收集并处理程序的参数了：

```
use std::env;

fn parse_args() -> Arguments {
    let args: Vec<String> = env::args().skip(1).collect();

    if args.len() != 4 {
        print_usage();
        eprintln!("{}", wrong number of arguments: expected 4, got
{}).",
            "Error:".red().bold(), args.len());
        std::process::exit(1);
    }

    Arguments {
        target: args[0].clone(),
        replacement: args[1].clone(),
        filename: args[2].clone(),
        output: args[3].clone()
    }
}
```

为了获取用户输入的参数，我们会使用与前面例子中相同的 `args` 迭代器。`.skip(1)` 会跳过迭代器的第一个值（正在运行的程序的名称），让结果中只含命令行参数。

首先 `collect()` 方法会生成一个 `Vec` 参数。然后我们会检查它的参数个数是否正确，如果不正确，则打印一条信息并以返回一个错误代码的形式退出。接下来我们再次对部分信息进行着色，并用 `.bold()` 把这段文本加粗。如果参数个数正确，就把它放入一个 `Arguments` 结构体中，并返回该结构体。

下面添加一个只会调用 `parse_args` 并打印输出的 `main` 函数：

```
fn main() {
    let args = parse_args();
```



```
println!("{:?}", args);  
}
```

现在，运行本程序，可以看到它正常输出了错误消息：

```
$ cargo run  
  Updating crates.io index  
Compiling libc v0.2.82  
Compiling lazy_static v1.4.0  
Compiling memchr v2.3.4  
Compiling regex-syntax v0.6.22  
Compiling thread_local v1.1.0  
Compiling aho-corasick v0.7.15  
Compiling atty v0.2.14  
Compiling text-colorizer v1.0.0  
Compiling regex v1.4.3  
Compiling quickreplace v0.1.0 (/home/jimb/quickreplace)  
Finished dev [unoptimized + debuginfo] target(s) in 6.98s  
Running `target/debug/quickreplace`  
quickreplace - change occurrences of one string into another  
Usage: quickreplace <target> <replacement> <INPUT> <OUTPUT>  
Error: wrong number of arguments: expected 4, got 0
```

如果传给程序的参数个数正确，那么它就会打印出 `Arguments` 结构体的文本表示：

```
$ cargo run "find" "replace" file output  
  Finished dev [unoptimized + debuginfo] target(s) in 0.01s  
  Running `target/debug/quickreplace find replace file output`  
Arguments { target: "find", replacement: "replace", filename:  
"file", output:  
"output" }
```

这是一个很好的开端。这些参数都已被正确提取并放置在 `Arguments` 结构体的正确部分中。

2.7.2 读写文件

接下来，我们需要用某种方法从文件系统中实际获取数据，以便进行处理，并在完工后将数据写回去。Rust 有一套健壮的输入 / 输出工具，但标准库的设计者知道读写文件是很常用的操作，所以刻意简化

了它。我们所要做的是导入模块 `std::fs`，然后就可以访问 `read_to_string` 函数和 `write` 函数了：

```
use std::fs;
```

`std::fs::read_to_string` 会返回一个 `Result<String, std::io::Error>`。如果此函数成功，就会生成一个 `String`；如果失败，就会生成一个 `std::io::Error`，这是标准库中用来表示 I/O 问题的类型。类似地，`std::fs::write` 会返回一个 `Result<(), std::io::Error>`：在成功的时候不返回任何内容，一旦出现问题就返回错误详情。

```
fn main() {
    let args = parse_args();

    let data = match fs::read_to_string(&args.filename) {
        Ok(v) => v,
        Err(e) => {
            eprintln!("{}", "failed to read from file '{}': {:?}",
                          "Error:".red().bold(), args.filename, e);
            std::process::exit(1);
        }
    };

    match fs::write(&args.output, &data) {
        Ok(_) => {},
        Err(e) => {
            eprintln!("{}", "failed to write to file '{}': {:?}",
                          "Error:".red().bold(), args.output, e);
            std::process::exit(1);
        }
    };
}
```

在这里，我们使用前面写好的 `parse_args()` 函数并将生成的文件名传给 `read_to_string` 和 `write`。对这些函数的输出使用 `match` 语句可以优雅地处理错误，打印出文件名、错误原因，并用一点儿醒目的颜色引起用户的注意。

有了这个改写后的 `main` 函数，运行程序时就可以看到下面这些了，当然，新旧文件的内容是完全相同的：

```
$ cargo run "find" "replace" Cargo.toml Copy.toml
   Compiling quickreplace v0.1.0 (/home/jimb/rust/quickreplace)
   Finished dev [unoptimized + debuginfo] target(s) in 0.01s
   Running `target/debug/quickreplace find replace Cargo.toml
Copy.toml`
```

该程序**确实**读取了输入文件 `Cargo.toml`，也**确实**写入了输出文件 `Copy.toml`，但是由于我们尚未编写任何代码来实际进行查找和替换，因此输出中没有任何变化。通过运行 `diff` 命令轻松进行查验，该命令确实没有检测到任何差异。

```
$ diff Cargo.toml Copy.toml
```

2.7.3 查找并替换

这个程序的最后一步是实现它的实际功能：查找并替换。为此，我们将使用 `regex crate`，它会编译并执行正则表达式。它提供了一个名为 `Regex` 的结构体，表示已编译的正则表达式。`Regex` 有一个 `replace_all` 方法，该方法名副其实：在一个字符串中搜索此正则表达式的所有匹配项，并用给定的替代字符串替换每个匹配项。可以将这段逻辑提取到一个函数中：

```
use regex::Regex;
fn replace(target: &str, replacement: &str, text: &str)
    -> Result<String, regex::Error>
{
    let regex = Regex::new(target)?;
    Ok(regex.replace_all(text, replacement).to_string())
}
```

注意看这个函数的返回类型。就像之前使用过的标准库函数一样，`replace` 也会返回一个 `Result`，但这次它携带着 `regex crate` 提供的错误类型。

`Regex::new` 会编译用户提供的正则表达式，如果给定的字符串无效，那么它就会失败。与曼德博程序中一样，我们使用 `?` 符号在 `Regex::new` 失败的情况下短路它，但该函数将返回 `regex crate` 特有的错误类型。一旦正则表达式编译完成，它的 `replace_all` 方法就能用给定的替代字符串替换 `text` 中的任何匹配项。

如果 `replace_all` 找到了匹配项，那么它就会返回一个新的 `String`，而这些匹配项会被替换成我们给它的文本。否则，`replace_all` 就会返回指向原始文本的指针，以回避不必要的内存分配和复制。然而，在这个例子中，我们想要一个独立的副本，因此无论是哪种情况，都要使用 `to_string` 方法来获取 `String` 并返回包裹在 `Result::Ok` 中的字符串，就像其他函数中的做法一样。

现在，是时候将这个新函数合并到 `main` 代码中了：

```
fn main() {
    let args = parse_args();

    let data = match fs::read_to_string(&args.filename) {
        Ok(v) => v,
        Err(e) => {
            eprintln!("{}", failed_to_read_from_file '{}': {:?} ",
                "Error:".red().bold(), args.filename, e);
            std::process::exit(1);
        }
    };

    let replaced_data = match replace(&args.target,
&args.replacement, &data) {
        Ok(v) => v,
        Err(e) => {
            eprintln!("{}", failed_to_replace_text: {:?} ",
                "Error:".red().bold(), e);
            std::process::exit(1);
        }
    };

    match fs::write(&args.output, &replaced_data) {
        Ok(v) => v,
        Err(e) => {
            eprintln!("{}", failed_to_write_to_file '{}': {:?} ",
                "Error:".red().bold(), args.filename, e);
        }
    }
}
```

```
        std::process::exit(1);
    }
};
}
```

完成了最后一步，程序已经就绪，你可以测试它了：

```
$ echo "Hello, world" > test.txt
$ cargo run "world" "Rust" test.txt test-modified.txt
   Compiling quickreplace v0.1.0 (/home/jimb/rust/quickreplace)
   Finished dev [unoptimized + debuginfo] target(s) in 0.88s
   Running `target/debug/quickreplace world Rust test.txt test-
modified.txt`

$ cat test-modified.txt
Hello, Rust
```

错误处理做得也很到位，它优雅地向用户报告错误：

```
$ cargo run "[[a-z]" "0" test.txt test-modified.txt
   Finished dev [unoptimized + debuginfo] target(s) in 0.01s
   Running `target/debug/quickreplace '[[a-z]' 0 test.txt test-
modified.txt`
Error: failed to replace text: Syntax(
~~~~~
~~~~~
regex parse error:
  [[a-z]
  ^
error: unclosed character class
~~~~~
~~~~~
)
```

当然，这个简单的演示中还缺少许多特性，但已经“五脏俱全”。至此，你已经了解了如何读取和写入文件、传播和显示错误，以及为输出着色以提升终端程序里的用户体验。

在未来的章节中，我们将探讨应用程序开发中的高级技术，从数据的集合以及使用迭代器进行函数式编程到可实现高效并发的异步编程技术，但首先，你得在第 3 章的 Rust 基本数据类型方面打下坚实的基础。

第 3 章 基本数据类型

之所以世界上有很多很多类型的书，是因为世界上有很多很多类型的人，而每个人都在追求不同类型的知识。

——Lemony Snicket

在很大程度上，可以说 Rust 语言就是围绕其类型来设计的。Rust 对高性能代码的支持，源自它能让开发人员选择最适合当前场景的数据表示法，并在简单性和成本之间进行合理的权衡。Rust 的内存和线程安全保障也依赖于其类型系统的健全性，而 Rust 的灵活性则源于其泛型类型和特型。

本章涵盖了 Rust 用以表示值的基本数据类型。这些源代码级类型都有其对应的机器级表示法，具有可预测的成本和性能。尽管 Rust 无法保证会完全按你的要求去表示各项事物，但只有当它确信能做出可靠的改进时才会谨慎地偏离你的原始要求。

与 JavaScript 或 Python 等动态类型语言相比，Rust 需要你提前做出更多规划。你必须明确写出各个函数参数和返回值的类型、结构体字段以及一些其他结构体。然而，Rust 的以下两个特性让这项工作比你预想的要轻松一点儿。

- 基于已明确写出的类型，Rust 的**类型推断**会帮你推断出剩下的大部分类型。实际上，对于给定的变量或表达式，通常只会有一种恰当的类型，在这种情况下，Rust 允许你省略类型。例如，你可以明确写出函数中的每一种类型，如下所示。

```
fn build_vector() -> Vec<i16> {  
    let mut v: Vec<i16> = Vec::<i16>::new();  
    v.push(10i16);  
    v.push(20i16);  
    v  
}
```

但这既凌乱又啰唆。既然已知函数的返回类型，那么显然 `v` 只能是一个 `Vec<i16>`，也就是 16 位有符号整数的向量，其他类型都不行。由此可知该向量的每个元素都必须是 `i16`。这正是 Rust 的类型推断所擅长的那种推理方式，这样一来，你可以将上述代码改写成下面这样。

```
fn build_vector() -> Vec<i16> {  
    let mut v = Vec::new();  
    v.push(10);  
    v.push(20);  
    v  
}
```

这两个定义是完全等效的，无论采用哪种方式，Rust 都会生成相同的机器码。类型推断让 Rust 具备了与动态类型语言相近的易读性，并且仍然能在编译期捕获类型错误。

- 函数可以是泛型的：单个函数就可以处理许多不同类型的值。

在 Python 和 JavaScript 中，所有函数都天生如此：函数可以对任何具备该函数所要求的属性和方法的值进行操作。（这就是通常称为**鸭子类型**的特征：如果它叫得像鸭子，走路像鸭子，那它就是鸭子。）但也正是这种灵活性让这些语言很难及早发现类型错误，而测试通常是发现此类错误的唯一途径。Rust 的泛型函数为该语言提供了一定程度的灵活性，而且仍然能在编译期捕获所有的类型错误。

虽然泛型函数更灵活，但其效率仍然与非泛型函数一样高。相较于编写能处理所有整数的泛型函数，为每种整数编写一个专用的 `sum` 函数并没有性能方面的内在优势。第 11 章会详细讨论泛型函数。

本章的其余部分自下向上介绍了 Rust 的一些类型，从简单的数值类型（如整数和浮点值）开始，转而介绍包含更多数据的类型：Box、元组（tuple）、数组和字符串。

接下来是你将在 Rust 中看到的各种类型的摘要。表 3-1 展示了 Rust 的原始类型、标准库中一些最常见的类型以及一些用户定义类型的示例。

表 3-1：Rust 中的类型示例

类型	说明	值
<code>i8</code> 、 <code>i16</code> 、 <code>i32</code> 、 <code>i64</code> 、 <code>i128</code> 、 <code>u8</code> 、 <code>u16</code> 、 <code>u32</code> 、 <code>u64</code> 、 <code>u128</code>	给定位宽的有符号整数和无符号整数	<code>42</code> 、 <code>-5i8</code> 、 <code>0x400u16</code> 、 <code>0o100i16</code> 、 <code>20_922_789_888_0</code> 字节字面量）
<code>isize</code> 、 <code>usize</code>	与机器字（32 位或 64 位）一样大的有符号整数和无符号整数	<code>137</code> 、 <code>-0b0101_0010isize</code> 、 <code>0xffff_fc00usize</code>
<code>f32</code> 、 <code>f64</code>	单精度 IEEE 浮点数和双精度 IEEE 浮点数	<code>1.61803</code> 、 <code>3.14f32</code> 、 <code>6.0221e23f64</code>
<code>bool</code>	布尔值	<code>true</code> 、 <code>false</code>
<code>char</code>	Unicode 字符，32 位宽（4 字节）	<code>'*'</code> 、 <code>'\n'</code> 、 <code>'字'</code> 、 <code>'\x7f'</code> 、 <code>'\u{CA0}'</code>
<code>(char, u8, i32)</code>	元组，允许混合类型	<code>('%', 0x7f, -1)</code>
<code>()</code>	“单元”（空元组）	<code>()</code>
<code>struct S { x: f32, y: f32 }</code>	具名字段型结构体	<code>S { x: 120.0, y: 209.0}</code>
<code>struct T(i32, char);</code>	元组型结构体	<code>T (120, 'x')</code>
<code>struct E;</code>	单元型结构体，无字段	<code>E</code>
<code>enum Attend { OnTime, Late(u32) }</code>	枚举，或代数数据类型	<code>Attend::Late(5)</code> 、 <code>Attend::OnTime</code>

类型	说明	值
<code>Box<Attend></code>	Box: 指向堆中值的拥有型指针	<code>Box::new(Late(15))</code>
<code>&i32</code> 、 <code>&mut i32</code>	共享引用和可变引用: 非拥有型指针, 其生命周期不能超出引用目标	<code>&s.y</code> 、 <code>&mut v</code>
<code>String</code>	UTF-8 字符串, 动态分配大小	<code>" ラーメン : ramen".to_string()</code>
<code>&str</code>	对 <code>str</code> 的引用: 指向 UTF-8 文本的非拥有型指针	<code>" そば : soba"&,&s[0..12]</code>
<code>[f64; 4]</code> 、 <code>[u8; 256]</code>	数组, 固定长度, 其元素类型都相同	<code>[1.0, 0.0, 0.0, 1.0]</code> 、 <code>[b' '; 256]</code>
<code>Vec<f64></code>	向量, 可变长度, 其元素类型都相同	<code>vec![0.367, 2.718, 7.389]</code>
<code>&[u8]</code> 、 <code>*mut [u8]</code>	对切片 (数组或向量某一部分) 的引用, 包含指针和长度	<code>&v[10..20]</code> 、 <code>&mut a[..]</code>
<code>Option<&str></code>	可选值: 或者为 <code>None</code> (无值), 或者为 <code>Some(v)</code> (有值, 其值为 <code>v</code>)	<code>Some("Dr.")</code> 、 <code>None</code>
<code>Result<u64, Error></code>	可能失败的操作结果: 或者为成功值 <code>Ok(v)</code> , 或者为错误值 <code>Err(e)</code>	<code>Ok(4096)</code> 、 <code>Err(Error::last_os_error())</code>
<code>&dyn Any</code> 、 <code>&mut dyn Read</code>	特型对象, 是对任何实现了一组给定方法的值的引用	<code>value as &dyn Any</code> 、 <code>&mut file as &mut dyn Read</code>
<code>fn(&str) -> bool</code>	函数指针	<code>str::is_empty</code>

类型	说明	值
(闭包类型没有显式书写形式)	闭包	<code> a, b a * a + b * b</code>

本章会对上述大多数类型进行介绍，但以下类型除外。

- 结构体 (struct) 类型 (参见第 9 章)。
- 枚举类型 (参见第 10 章)。
- 特型对象 (参见第 11 章)。
- 函数和闭包类型 (参见第 14 章)。
- String 和 &str 的更多细节 (参见第 17 章)。

3.1 固定宽度的数值类型

Rust 类型系统的根基是一组固定宽度的数值类型，选用这些类型是为了匹配几乎所有现代处理器都已直接在硬件中实现的类型。

固定宽度的数值类型可能会溢出或丢失精度，但它们足以满足大多数应用程序的需求，并且要比任意精度整数和精确有理数等表示法快数千倍。如果需要后面提到的那些类型的数值的表示法，可以到 `num crate` 中找到它们。

Rust 中数值类型的名称都遵循着一种统一的模式，也就是以“位”数表明它们的宽度，以前缀表明它们的用法，如表 3-2 所示。

表 3-2: Rust 数值类型

大小（位）	无符号整数	有符号整数	浮点数
8	u8	i8	
16	u16	i16	
32	u32	i32	f32
64	u64	i64	f64
128	u128	i128	
机器字	usize	isize	

在这里，**机器字**是一个值，其大小等于运行此代码的机器上“地址”的大小，可能是 32 位，也可能是 64 位。

3.1.1 整型

Rust 的无符号整型会使用它们的完整范围来表示正值和 0，如表 3-3 所示。

表 3-3：Rust 无符号整型

类型	范围
u8	0 到 2^8-1 (0 到 255)
u16	0 到 $2^{16}-1$ (0 到 65 535)
u32	0 到 $2^{32}-1$ (0 到 4 294 967 295)
u64	0 到 $2^{64}-1$ (0 到 18 446 744 073 709 551 615 或 1844 京)
u128	0 到 $2^{128}-1$ (0 到大约 3.4×10^{38})
usize	0 到 $2^{32}-1$ 或 $2^{64}-1$

Rust 的有符号整型会使用二进制补码表示，使用与相应的无符号类型相同的位模式来覆盖正值和负值的范围，如表 3-4 所示。

表 3-4：Rust 有符号整型

类型	范围
i8	-2^7 到 2^7-1 (-128 到 127)
i16	-2^{15} 到 $2^{15}-1$ (-32 768 到 32 767)
i32	-2^{31} 到 $2^{31}-1$ (-2 147 483 648 到 2 147 483 647)
i64	-2^{63} 到 $2^{63}-1$ (-9 223 372 036 854 775 808 到 9 223 372 036 854 775 807)
i128	-2^{127} 到 $2^{127}-1$ (大约 -1.7×10^{38} 到 $+1.7 \times 10^{38}$)
isize	-2^{31} 到 $2^{31}-1$ 或 -2^{63} 到 $2^{63}-1$

Rust 会使用 u8 类型作为字节值。例如，从二进制文件或套接字中读取数据时会产生一个 u8 值构成的流。

与 C 和 C++ 不同，Rust 会把字符视为与数值截然不同的类型：char 既不是 u8，也不是 u32（尽管它确实有 32 位长）。稍后 3.3 节会详细讲解 Rust 的 char 类型。

usize 类型和 isize 类型类似于 C 和 C++ 中的 size_t 和 ptrdiff_t。它们的精度与目标机器上地址空间的大小保持一致，即在 32 位架构上是 32 位长，在 64 位架构上则是 64 位长。Rust 要求数组索引是 usize 值。用来表示数组或向量大小或某些数据结构中元素数量的值通常也是 usize 类型。

Rust 中的整型字面量可以带上一个后缀来指示它们的类型：42u8 是 u8 类型，1729isize 是 isize 类型。如果整型字面量没有带类

型后缀，那么 Rust 就会延迟确定其类型，直到找出一处足以认定其类型的使用代码，比如存储在特定类型的变量中、传给期待特定类型的函数、与具有特定类型的另一个值进行比较，等等。最后，如果有多种候选类型，那么 Rust 就会默认使用 `i32`（如果是候选类型之一的话）。如果无法认定类型，那么 Rust 就会将此歧义报告为错误。

前缀 `0x`、`0o` 和 `0b` 分别表示十六进制字面量、八进制字面量和二进制字面量。

为了让长数值更易读，可以在数字之间任意插入下划线。例如，可以将 `u32` 的最大值写为 `4_294_967_295`。下划线的具体位置无关紧要，因此也可以将十六进制数或二进制数按 4 位数字而非 3 位数字进行分组（如 `0xffff_ffff`），或分隔开数字的类型后缀（如 `127_u8`）。表 3-5 中展示了整型字面量的一些示例。

表 3-5：整型字面量示例

字面量	类型	十进制值
<code>116i8</code>	<code>i8</code>	116
<code>0xcafeu32</code>	<code>u32</code>	51966
<code>0b0010_1010</code>	推断	42
<code>0o106</code>	推断	70

尽管数值类型和 `char` 类型是不同的，但 Rust 确实为 `u8` 值提供了字节字面量。与字符字面量类似，`b'x'` 表示以字符 `x` 的 ASCII 码作为 `u8` 值。例如，由于 `A` 的 ASCII 码是 65，因此字面量 `b'A'` 和 `65u8` 完全等效。只有 ASCII 字符才能出现在字节字面量中。

有几个字符不能简单地放在单引号后面，因为那样在语法上会有歧义或难以阅读。表 3-6 中的字符只能以反斜杠开头的替代符号来书写。

表 3-6：需要替代符号的字符

字符	字节字面量	等效的数值
单引号 (')	b'\''	39u8
反斜杠 (\)	b'\\'	92u8
换行 (lf)	b'\n'	10u8
回车 (cr)	b'\r'	13u8
制表符 (tab)	b'\t'	9u8

对于难于书写或阅读的字符，可以将其编码改为十六进制。这种字节字面量形如 b'\xHH'，其中 HH 是任意两位十六进制数，表示值为 HH 的字节。例如，你可以将 ASCII 控制字符 escape 的字节字面量写成 b'\x1b'，因为 escape 的 ASCII 码为 27，即十六进制的 1B。由于字节字面量只是 u8 值的表示法之一，因此还应该考虑使用一个整型字面量是否更易读：只有当你要强调该值表示的是 ASCII 码时，才应该使用 b'\x1b' 而不是简单明了的 27。

可以使用 as 运算符将一种整型转换为另一种整型。6.14 节会详细讲解类型转换的原理，这里先举一些例子：

```
assert_eq!( 10_i8 as u16, 10_u16); // 范围内转换
assert_eq!( 2525_u16 as i16, 2525_i16); // 范围内转换

assert_eq!( -1_i16 as i32, -1_i32); // 带符号扩展
```

```
assert_eq!(65535_u16 as i32, 65535_i32); // 填零扩展

// 超出目标范围的转换生成的值等于原始值对 $2^N$ 取模的值，
// 其中N是按位算的目标宽度。有时这也称为“截断”
assert_eq!(1000_i16 as u8, 232_u8);
assert_eq!(65535_u32 as i16, -1_i16);

assert_eq!(-1_i8 as u8, 255_u8);
assert_eq!(255_u8 as i8, -1_i8);
```

标准库提供了一些运算，可以像整型的方法一样使用。例如：

```
assert_eq!(2_u16.pow(4), 16); // 求幂
assert_eq!((-4_i32).abs(), 4); // 求绝对值
assert_eq!(0b101101_u8.count_ones(), 4); // 求二进制1的个数
```

可以在在线文档中找到这些内容。但请注意，该文档在“i32（原始类型）”和此类型的专有模块（搜索“std::i32”）下的单独页面中分别含有此类型的信息。

在实际编码中，通常不必像刚才那样写出类型后缀，因为其上下文将决定类型。但是，如果没有类型后缀且无法决定类型，那么错误消息可能会令人惊讶。例如，以下代码无法编译：

```
println!("{}", (-4).abs());
```

Rust 会报错：

```
error: can't call method `abs` on ambiguous numeric type
`{integer}`
```

这令人不解：明明所有的有符号整型都有一个 abs 方法，那么问题出在哪里呢？出于技术原因，Rust 在调用类型本身的方法之前必须确切地知道一个值属于哪种整型。只有在解析完所有方法调用后类型仍然不明确的时候，才会默认为 i32，但这里并没有其他方法可供解析，因此 Rust 提供不了帮助。解决方案是加上后缀或使用特定类型的函数来明确写出希望的类型：


```
println!("{}", (-4_i32).abs());  
println!("{}", i32::abs(-4));
```

请注意，方法调用的优先级高于一元前缀运算符，因此在将方法应用于负值时要小心。如果第一个语句中 `-4_i32` 周围没有圆括号，则 `-4_i32.abs()` 会先针对正值 4 调用 `abs` 方法，生成正值 4，再根据负号取负，得到 `-4`。

3.1.2 检查算法、回绕算法、饱和算法和溢出算法

当整型算术运算溢出时，Rust 在调试构建中会出现 `panic`。而在发布构建中，运算会回绕：它生成的值等于“数学意义上正确的结果”对“值类型范围”取模的值。（在任何情况下都不会像 C 和 C++ 中那样出现“溢出未定义”的行为。）

例如，以下代码在调试构建中会出现 `panic`：

```
let mut i = 1;  
loop {  
    i *= 10; // panic: 试图进行可能溢出的乘法（但只会在调试构建中出现）  
}
```

在发布构建中，此乘法会返回负数，并且循环会无限运行。

如果这种默认行为不是你想要的，则整型提供的某些方法可以让你准确地阐明自己期望的行为。例如，在任意构建中都会出现下列 `panic`：

```
let mut i: i32 = 1;  
loop {  
    // panic: 乘法溢出（在任意构建中出现）  
    i = i.checked_mul(10).expect("multiplication overflowed");  
}
```

这些整型算术方法分为 4 大类。

- 检查运算会返回结果的 `Option` 值：如果数学意义上正确的结果可以表示为该类型的值，那么就为 `Some(v)`，否则为

None。

```
// 10与20之和可以表示为u8
assert_eq!(10_u8.checked_add(20), Some(30));

// 很遗憾，100与200之和不能表示为u8
assert_eq!(100_u8.checked_add(200), None);

// 做加法。如果溢出，则会出现panic
let sum = x.checked_add(y).unwrap();

// 奇怪的是，在某种特殊情况下，带符号的除法也会溢出。
// 带符号的n位类型可以表示 $-2^{n-1}$ ，但不足以表示 $2^{n-1}$ 
assert_eq!((-128_i8).checked_div(-1), None);
```

- **回绕**运算会返回与“数学意义上正确的结果”对“值类型范围”取模的值相等的值。

```
// 第一个结果可以表示为u16，第二个则不能，所以会得到250000 对 $2^{16}$ 的模
assert_eq!(100_u16.wrapping_mul(200), 20000);
assert_eq!(500_u16.wrapping_mul(500), 53392);

// 对有符号类型的运算可能会回绕为负值
assert_eq!(500_i16.wrapping_mul(500), -12144);

// 在移位运算中，移位距离会在值的大小范围内回绕，
// 所以在16位类型中移动17位就相当于移动了1位
assert_eq!(5_i16.wrapping_shl(17), 10);
```

如前所述，这就是普通算术运算符在发布构建中的行为。这些方法的优点是它们在所有构建中的行为方式都是相同的。

- **饱和**运算会返回最接近“数学意义上正确结果”的可表达值。换句话说，结果“紧贴着”该类型可表达的最大值和最小值。

```
assert_eq!(32760_i16.saturating_add(10), 32767);
assert_eq!((-32760_i16).saturating_sub(10), -32768);
```

不存在饱和除法¹、饱和求余法²或饱和位移法³。

- 溢出运算会返回一个元组 (result, overflowed)，其中 result 是函数的回绕版本所返回的内容，而 overflowed 是一个布尔值，指示是否发生过溢出。

```
assert_eq!(255_u8.overflowing_sub(2), (253, false));
assert_eq!(255_u8.overflowing_add(2), (1, true));
```

overflowing_shl 和 overflowing_shr 稍微偏离了这种模式：只有当移位距离与类型本身的位宽一样大或比其更大时，它们才会为 overflowed 返回 true。实际应用的移位数是所请求的移位数对类型位宽取模的结果。

```
// 移动17位对`u16`来说太大了，而17对16取模就是1
assert_eq!(5_u16.overflowing_shl(17), (10, true));
```

¹ 整数相除一般不会溢出，即使溢出也没有“数学意义上的正确结果”。——译者注

² 饱和是对溢出的一种补救方式，余数不可能溢出，因此饱和也没有意义。——译者注

³ 饱和是对溢出的一种补救方式，移位的溢出在不同情况下补救方式不同，因此无法统一支持。——译者注

前缀 checked_、wrapping_、saturating_ 或 overflowing_ 后面可以跟着的运算名称如表 3-7 所示。

表 3-7：运算名称

运算	名称后缀	例子
加法	add	100_i8.checked_add(27) == Some(127)

运算	名称后缀	例子
减法	sub	<code>10_u8.checked_sub(11) == None</code>
乘法	mul	<code>128_u8.saturating_mul(3) == 255</code>
除法	div	<code>64_u16.wrapping_div(8) == 8</code>
求余	rem	<code>(-32768_i16).wrapping_rem(-1) == 0</code>
取负	neg	<code>(-128_i8).checked_neg() == None</code>
绝对值	abs	<code>(-32768_i16).wrapping_abs() == -32768</code>
求幂	pow	<code>3_u8.checked_pow(4) == Some(81)</code>
按位左移	shl	<code>10_u32.wrapping_shl(34) == 40</code>
按位右移	shr	<code>40_u64.wrapping_shr(66) == 10</code>

3.1.3 浮点类型

Rust 提供了 IEEE 单精度浮点类型和 IEEE 双精度浮点类型。这些类型包括正无穷大和负无穷大、不同的正零值和负零值，以及非数值。如表 3-8 所示。

表 3-8：IEEE 单精度浮点类型和 IEEE 双精度浮点类型

类型	精度	范围
f32	IEEE 单精度（至少 6 位小数）	大约 -3.4×10^{38} 至 $+3.4 \times 10^{38}$
f64	IEEE 双精度（至少 15 位小数）	大约 -1.8×10^{308} 至 $+1.8 \times 10^{308}$

Rust 的 `f32` 和 `f64` 分别对应于 C 和 C++（在支持 IEEE 浮点的实现中）以及 Java（始终使用 IEEE 浮点）中的 `float` 类型和 `double` 类型。

浮点字面量的一般化形式如图 3-1 所示。



图 3-1：浮点字面量

浮点数中整数部分之后的每个部分都是可选的，但必须至少存在小数部分、指数或类型后缀这三者中的一个，以将其与整型字面量区分开来。小数部分可以仅由一个单独的小数点组成，因此 `5.` 也是有效的浮点常量。

如果浮点字面量缺少类型后缀，那么 Rust 就会检查上下文以查看值的使用方式，这与整型字面量非常相似。如果它最终发现这两种浮点类型都适合，就会默认选择 f64。

为了便于类型推断，Rust 会将整型字面量和浮点字面量视为不同的大类：它永远不会把整型字面量推断为浮点类型，反之亦然。表 3-9 展示了浮点字面量的一些示例。

表 3-9：浮点字面量的例子

字面量	类型	数学值
-1.5625	自动推断	$-(1^9/16)$
2.	自动推断	2
0.25	自动推断	$\frac{1}{4}$
1e4	自动推断	10 000
40f32	f32	40
9.109_383_56e-31f64	f64	大约 $9.109\ 383\ 56 \times 10^{-31}$

f32 类型和 f64 类型具有 IEEE 要求的一些特殊值的关联常量，比如 INFINITY（无穷大）、NEG_INFINITY（负无穷大）、NaN（非数值）以及 MIN（最小有限值）和 MAX（最大有限值）：

```
assert!((-1. / f32::INFINITY).is_sign_negative());
assert_eq!(-f32::MIN, f32::MAX);
```

f32 类型和 f64 类型提供了完备的数学计算方法，比如 `2f64.sqrt()` 就是 2 的双精度平方根。下面是一些例子：

```
assert_eq!(5f32.sqrt() * 5f32.sqrt(), 5.); // 按IEEE的规定，它精确等于5.0
assert_eq!((-1.01f64).floor(), -2.0);
```

再次提醒，方法调用的优先级高于前缀运算符，因此在对负值进行方法调用时，请务必正确地加上圆括号。

`std::f32::consts` 模块和 `std::f64::consts` 模块提供了各种常用的数学常量，比如 `E`、`PI` 和 2 的平方根。

在搜索文档时，请记住这两种类型本身都有名为“f32（原始类型）”和“f64（原始类型）”的页面，以及每种类型的单独模块 `std::f32` 和 `std::f64`。

与整数一样，通常不必在实际代码中写出浮点字面量的类型后缀，但如果你想这么做，那么将类型放在字面量或函数上就可以：

```
println!("{}", (2.0_f64).sqrt());
println!("{}", f64::sqrt(2.0));
```

与 C 和 C++ 不同，Rust 几乎不会执行任何隐式的数值转换。如果函数需要 f64 参数，则传入 i32 型参数是错误的。事实上，Rust 甚至不会隐式地将 i16 值转换为 i32 值，虽然每个 i16 值都必然在 i32 范围内。不过，你随时可以用 `as` 运算符写出显式转换：`i as f64` 或 `x as i32`。

缺少隐式转换有时会让 Rust 表达式比类似的 C 或 C++ 代码更冗长。然而，隐式整数转换有着导致错误和安全漏洞的大量“前科”，特别是在用这种整数表示内存中某些内容的大小时，很可能发生意外溢出。根据以往的经验，Rust 这种要求明确写出数值类型转换的行为，会提醒我们注意到一些可能错过的问题。

6.14 节会解释各种类型转换的确切行为。

3.2 布尔类型

Rust 的布尔类型 `bool` 具有此类型常用的两个值 `true` 和 `false`。`==`、`<` 等比较运算符会生成 `bool` 结果，比如 `2 < 5` 的值为 `true`。

许多语言对在要求布尔值的上下文中使用其他类型的值持宽松态度，比如 C 和 C++ 会把字符、整数、浮点数和指针隐式转换成布尔值，因此它们可以直接用作 `if` 语句或 `while` 语句中的条件。Python 允许在布尔上下文中使用字符串、列表、字典甚至 `Set`，如果这些值是非空的，则将它们视为 `true`。然而，Rust 非常严格：像 `if` 和 `while` 这样的控制结构要求它们的条件必须是 `bool` 表达式，短路逻辑运算符 `&&` 和 `||` 也是如此。你必须写成 `if x != 0 { ... }`，而不能只写成 `if x { ... }`。

Rust 的 `as` 运算符可以将 `bool` 值转换为整型：

```
assert_eq!(false as i32, 0);
assert_eq!(true  as i32, 1);
```

但是，`as` 无法进行另一个方向（从数值类型到 `bool`）的转换。相反，你必须显式地写出比较表达式，比如 `x != 0`。

尽管 `bool` 只需要用一个位来表示，但 Rust 在内存中会使用整字节来表示 `bool` 值，因此可以创建指向它的指针。

3.3 字符

Rust 的字符类型 `char` 会以 32 位值表示单个 Unicode 字符。

Rust 会对单独的字符使用 `char` 类型，但对字符串和文本流使用 UTF-8 编码。因此，`String` 会将其文本表示为 UTF-8 字节序列，而不是字符数组。

字符字面量是用单引号括起来的字符，比如 `'8'` 或 `'!'`。还可以使用全角 Unicode 字符：`'錆'` 是一个 `char` 字面量，表示日文汉字中的 `sabi (rust)`。

与字节字面量一样，有些字符需要用反斜杠转义，如表 3-10 所示。

表 3-10：需要用反斜杠转义的字符

字符	Rust 字符字面量
单引号 (')	'\''
反斜杠 (\)	'\\'
换行 (lf)	'\n'
回车 (cr)	'\r'
制表 (tab)	'\t'

如果你愿意，还可以用十六进制写出字符的 Unicode 码点。

- 如果字符的码点在 U+0000 到 U+007F 范围内（也就是说，如果它是从 ASCII 字符集中提取的），就可以把字符写为 `'\xHH'`，其中 HH 是两个十六进制数。例如，字符字面量 `'*'` 和 `'\x2A'` 是等效的，因为字符 `*` 的码点是 42 或十六进制的 2A。
- 可以将任何 Unicode 字符写为 `'\u{HHHHHH}'` 形式，其中 HHHHHH 是最多 6 个十六进制数，可以像往常一样用下划线进行分组。例如，字符字面量 `'\u{CA0}'` 表示字符“ᱠ”，这是 Unicode 中用于表示反对的卡纳达语字符“ᱠ_ᱠ”。同样的字面量也可以简写成 `'ᱠ'`。

`char` 总是包含 0x0000 到 0xD7FF 或 0xE000 到 0x10FFFF 范围内的 Unicode 码点。`char` 永远不会是“半代用区”中的码点（0xD800 到 0xDFFF 范围内的码点，它们不能单独使用）或 Unicode 码点空间之外的值（大于 0x10FFFF 的值）。Rust 使用类型系统和动态检查来确保 `char` 值始终在允许的范围内。

Rust 不会在 `char` 和任何其他类型之间进行隐式转换。可以使用 `as` 转换运算符将 `char` 转换为整型，对于小于 32 位的类型，该字符值的高位会被截断：

```
assert_eq!('*', as i32, 42);
assert_eq!('ᱠ' as u16, 0xca0);
assert_eq!('ᱠ' as i8, -0x60); // U+0CA0截断到8位，有符号
```

从另一个方向来看，`u8` 是唯一能通过 `as` 运算符转换为 `char` 的类型，因为 Rust 刻意让 `as` 运算符只执行开销极低且可靠的转换，但是除 `u8` 之外的每个整型都可能包含 Unicode 码点之外的值，所以这些转换都要做运行期检查。作为替代方案，标准库函数 `std::char::from_u32` 可以接受任何 `u32` 值并返回一个 `Option<char>`：如果此 `u32` 不是允许的 Unicode 码点，那么 `from_u32` 就会返回 `None`，否则，它会返回 `Some(c)`，其中 `c` 是转换成 `char` 后的结果。

标准库为字符提供了一些有用的方法，你可以在“char（原始类型）”和模块“std::char”下的在线文档中找到这些方法。

```
assert_eq!('*'.is_alphabetic(), false);
assert_eq!('β'.is_alphabetic(), true);
assert_eq!('8'.to_digit(10), Some(8));
assert_eq!('ð'.len_utf8(), 3);
assert_eq!(std::char::from_digit(2, 10), Some('2'));
```

孤立的字符自然不如字符串和文本流那么有用。3.7 节会讲解 Rust 的标准 String 类型和文本处理。

3.4 元组

元组是各种类型值的值对或三元组、四元组、五元组等（因此称为 n -元组或元组）。可以将元组编写为一个元素序列，用逗号隔开并包裹在一对圆括号中。例如，`("Brazil", 1985)` 是一个元组，其第一个元素是一个静态分配的字符串，第二个元素是一个整数，它的类型是 `(&str, i32)`。给定一个元组值 `t`，可以通过 `t.0`、`t.1` 等访问其元素。

元组有点儿类似于数组，即这两种类型都表示值的有序序列。许多编程语言混用或结合了这两个概念，但在 Rust 中，它们是截然不同的。一方面，元组的每个元素可以有不同的类型，而数组的元素必须都是相同的类型。另一方面，元组只允许用常量作为索引，比如 `t.4`。不能通过写成 `t.i` 或 `t[i]` 的形式来获取第 `i` 个元素。

Rust 代码通常会用元组类型从一个函数返回多个值。例如，字符串切片上的 `split_at` 方法会将字符串分成两半并返回它们，其声明如下所示：

```
fn split_at(&self, mid: usize) -> (&str, &str);
```

返回类型 `(&str, &str)` 是两个字符串切片构成的元组。可以用模式匹配语法将返回值的每个元素赋值给不同的变量：

```
let text = "I see the eigenvalue in thine eye";
let (head, tail) = text.split_at(21);
assert_eq!(head, "I see the eigenvalue ");
assert_eq!(tail, "in thine eye");
```

这样比其等效写法更易读：

```
let text = "I see the eigenvalue in thine eye";
let temp = text.split_at(21);
let head = temp.0;
let tail = temp.1;
assert_eq!(head, "I see the eigenvalue ");
assert_eq!(tail, "in thine eye");
```

你还会看到元组被用作一种超级小巧的结构体类型。例如，在第 2 章的曼德博程序中，我们要将图像的宽度和高度传给绘制它的函数并将其写入磁盘。为此可以声明一个具有 `width` 成员和 `height` 成员的结构体，但对如此显而易见的事情来说，这种写法相当烦琐，所以我们只用了一个元组：

```
/// 把`pixels`缓冲区（其尺寸由`bounds`给出）写入名为`filename`的文件中
fn write_image(filename: &str, pixels: &[u8], bounds: (usize,
    usize))
    -> Result<(), std::io::Error>
{ ... }
```

`bounds` 参数的类型是 `(usize, usize)`，这是一个包含两个 `usize` 值的元组。当然也可以写成单独的 `width` 参数和 `height` 参数，并且最终的机器码也基本一样。但重点在于思路的清晰度。应该把大小看作一个值，而不是两个，使用元组能更准确地记述这种意图。

另一种常用的元组类型是零元组 `()`。传统上，这叫作**单元类型**，因为此类型只有一个值，写作 `()`。当无法携带任何有意义的值但其上下文仍然要求传入某种类型时，Rust 就会使用单元类型。

例如，不返回值的函数的返回类型为 `()`。标准库的 `std::mem::swap` 函数就没有任何有意义的返回值，它只会交换两个参数的值。`std::mem::swap` 的声明如下所示：

```
fn swap<T>(x: &mut T, y: &mut T);
```

这个 `<T>` 意味着 `swap` 是泛型的：可以将对任意类型 `T` 的值的引用传给它。但此签名完全省略了 `swap` 的返回类型，它是以下完整写法的简写形式：

```
fn swap<T>(x: &mut T, y: &mut T) -> ();
```

类似地，前面提到过的 `write_image` 示例的返回类型是 `Result<(), std::io::Error>`，这意味着该函数在出错时会返回一个 `std::io::Error` 值，但成功时不会返回任何值。

如果你愿意，可以在元组的最后一个元素之后跟上一个逗号：类型 `(&str, i32,)` 和 `(&str, i32)` 是等效的，表达式 `("Brazil", 1985,)` 和 `("Brazil", 1985)` 是等效的。Rust 始终允许在所有能用逗号的地方（函数参数、数组、结构体和枚举定义，等等）添加额外的尾随逗号。这对人类读者来说可能很奇怪，不过一旦在多行列表末尾添加或移除了条目（entry），在显示差异时就会更容易阅读。

为了保持一致性，甚至有包含单个值的元组。字面量 `("lonely hearts",)` 就是一个包含单个字符串的元组，它的类型是 `(&str,)`。在这里，值后面的逗号是必需的，以用于区分单值元组和简单的括号表达式。

3.5 指针类型

Rust 有多种表示内存地址的类型。

这是 Rust 和大多数具有垃圾回收功能的语言之间一个重大的差异。在 Java 中，如果 `class Rectangle` 包含字段 `Vector2D upperLeft;`，那么 `upperLeft` 就是对另一个单独创建的 `Vector2D` 对象的引用。在 Java 中，一个对象永远不会包含其他对象的实际内容。

但 Rust 不一样。该语言旨在帮你将内存分配保持在最低限度。默认情况下值会嵌套。值 `((0, 0), (1440, 900))` 会存储为 4 个相邻的整数。如果将它存储在一个局部变量中，则会得到 4 倍于整数宽度的局部变量。堆中没有分配任何内容。

这可以帮我们高效利用内存，但代价是，当 Rust 程序需要让一些值指向其他值时，必须显式使用指针类型。好消息是，当使用这些指针类型时，安全的 Rust 会对其进行约束，以消除未定义的行为，因此指针在 Rust 中比在 C++ 中更容易正确使用。

接下来将讨论 3 种指针类型：引用、Box 和不安全指针。

3.5.1 引用

`&String` 类型的值（读作“ref String”）是对 `String` 值的引用，`&i32` 是对 `i32` 的引用，以此类推。

最简单的方式是将引用视为 Rust 中的基本指针类型。在运行期间，对 `i32` 的引用是一个保存着 `i32` 地址的机器字，这个地址可能位于栈或堆中。表达式 `&x` 会生成一个对 `x` 的引用，在 Rust 术语中，我们会说它借用了 `x` 的引用。给定一个引用 `r`，表达式 `*r` 会引用 `r` 指向的值。它们非常像 C 和 C++ 中的 `&` 运算符和 `*` 运算符，并且和 C 中的指针一样，当超出作用域时引用不会自动释放任何资源。

然而，与 C 指针不同，Rust 的引用永远不会为空：在安全的 Rust 中根本没有办法生成空引用。与 C 不同，Rust 会跟踪值的所有权和生命周期，因此早在编译期间就排除了悬空指针、双重释放和指针失效等错误。

Rust 引用有两种形式。

`&T`

一个不可变的共享引用。你可以同时拥有多个对给定值的共享引用，但它们是只读的：禁止修改它们所指向的值，就像 C 中的 `const T*` 一样。

`&mut T`

一个可变的、独占的引用。你可以读取和修改它指向的值，就像 C 中的 `T*` 一样。但是只要该引用还存在，就不能对该值有任何类型的其他引用。事实上，访问该值的唯一途径就是使用这个可变引用。

Rust 利用共享引用和可变引用之间的“二选一”机制来强制执行“单个写入者或多个读取者”规则：你或者独占读写一个值，或者让任意数量的读取者共享，但二者只能选择其一。这种由编译器检查强制执行的“二选一”规则是 Rust 安全保障的核心。第 5 章会解释 Rust 的安全引用的使用规则。

3.5.2 Box

在堆中分配值的最简单方式是使用 `Box::new`：

```
let t = (12, "eggs");  
let b = Box::new(t); // 在堆中分配一个元组
```

`t` 的类型是 `(i32, &str)`，所以 `b` 的类型是 `Box<(i32, &str)>`。对 `Box::new` 的调用会分配足够的内存以在堆上容纳此元组。当 `b` 超出作用域时，内存会立即被释放，除非 `b` 已被移动

(move)，比如返回它。移动对于 Rust 处理在堆上分配的值的方式至关重要，第 4 章会对此进行详细解释。

3.5.3 裸指针

Rust 也有裸指针类型 `*mut T` 和 `*const T`。裸指针实际上和 C++ 中的指针很像。使用裸指针是不安全的，因为 Rust 不会跟踪它指向的内容。例如，裸指针可能为空，或者它们可能指向已释放的内存或现在包含不同类型的值。C++ 的所有经典指针错误都可能“借尸还魂”。

但是，你只能在 `unsafe` 块中对裸指针解引用 (dereference)。 `unsafe` 块是 Rust 高级语言特性中的可选机制，其安全性取决于你自己。如果代码中没有 `unsafe` 块（或者虽然有但编写正确），那么本书中强调的安全保证就仍然有效。有关详细信息，请参阅第 22 章。

3.6 数组、向量和切片

Rust 用 3 种类型来表示内存中的值序列。

- 类型 `[T; N]` 表示 `N` 个值的数组，每个值的类型为 `T`。数组的大小是在编译期就已确定的常量，并且是类型的一部分，不能追加新元素或缩小数组。
- 类型 `Vec<T>` 可称为 `T` 的向量，它是一个动态分配且可增长的 `T` 类型的值序列。向量的元素存在于堆中，因此可以随意调整向量的大小：压入新元素、追加其他向量、删除元素等。
- 类型 `&[T]` 和 `&mut [T]` 可称为 `T` 的共享切片和 `T` 的可变切片，它们是对一系列元素的引用，这些元素是某个其他值（比如数组或向量）的一部分。可以将切片视为指向其第一个元素的指针，以及从该点开始允许访问的元素数量的计数。可变切片 `&mut [T]` 允许读取元素和修改元素，但不能共享；共享切片 `&[T]` 允许在多个读取者之间共享访问权限，但不允许修改元素。

给定这 3 种类型中任意一种类型的值 `v`，表达式 `v.len()` 都会给出 `v` 中的元素数，而 `v[i]` 引用的是 `v` 的第 `i` 个元素。`v` 的第一个元素是 `v[0]`，最后一个元素是 `v[v.len() - 1]`。Rust 总是会检查 `i` 是否在这个范围内，如果没在，则此表达式会出现 panic。`v` 的长度可能为 0，在这种情况下，任何对其进行索引的尝试都会出现 panic。`i` 的类型必须是 `usize`，不能使用任何其他整型作为索引。

3.6.1 数组

编写数组值的方法有好几种，其中最简单的方法是在方括号内写入一系列值：

```
let lazy_caterer: [u32; 6] = [1, 2, 4, 7, 11, 16];
let taxonomy = ["Animalia", "Arthropoda", "Insecta"];

assert_eq!(lazy_caterer[3], 7);
assert_eq!(taxonomy.len(), 3);
```

对于要填充一些值的长数组的常见情况，可以写成 `[V; N]`，其中 `V` 是每个元素的值，`N` 是长度。例如，`[true; 10000]` 是一个包含 10 000 个 `bool` 元素的数组，其内容全为 `true`：

```
let mut sieve = [true; 10000];
for i in 2..100 {
    if sieve[i] {
        let mut j = i * i;
        while j < 10000 {
            sieve[j] = false;
            j += i;
        }
    }
}

assert!(sieve[211]);
assert!(!sieve[9876]);
```

你会看到用来声明固定大小缓冲区的语法：`[0u8; 1024]`，它是一个 1 KB 的缓冲区，用 0 填充。Rust 没有任何能定义未初始化数组的写法。（一般来说，Rust 会确保代码永远无法访问任何种类的未初始化值。）

数组的长度是其类型的一部分，并会在编译期固定下来。如果 `n` 是变量，则不能写成 `[true; n]` 以期得到一个包含 `n` 个元素的数组。当你需要一个长度在运行期可变的数组时（通常都是这样），请改用向量。

你在数组上看到的那些实用方法（遍历元素、搜索、排序、填充、过滤等）都是作为切片而非数组的方法提供的。但是 Rust 在搜索各种方法时会隐式地将对数组的引用转换为切片，因此可以直接在数组上调用任何切片方法：

```
let mut chaos = [3, 5, 4, 1, 2];
chaos.sort();
assert_eq!(chaos, [1, 2, 3, 4, 5]);
```

在这里，`sort` 方法实际上是在切片上定义的，但由于它是通过引用获取的操作目标，因此 Rust 会隐式地生成一个引用整个数组的

`&mut [i32]` 切片，并将其传给 `sort` 来进行操作。其实前面提到过的 `len` 方法也是切片的方法之一。3.6.3 节会更详细地介绍切片。

3.6.2 向量

向量 `Vec<T>` 是一个可调整大小的 `T` 类型元素的数组，它是在堆上分配的。

创建向量的方法有好几种，其中最简单的方法是使用 `vec!` 宏，它为我们提供了一个看起来非常像数组字面量的向量语法：

```
let mut primes = vec![2, 3, 5, 7];
assert_eq!(primes.iter().product::<i32>(), 210);
```

当然，这仍然是一个向量，而不是数组，所以可以动态地向它添加元素：

```
primes.push(11);
primes.push(13);
assert_eq!(primes.iter().product::<i32>(), 30030);
```

还可以通过将给定值重复一定次数来构建向量，可以再次使用模仿数组字面量的语法：

```
fn new_pixel_buffer(rows: usize, cols: usize) -> Vec<u8> {
    vec![0; rows * cols]
}
```

`vec!` 宏相当于调用 `Vec::new` 来创建一个新的空向量，然后将元素压入其中，这是另一种惯用法：

```
let mut pal = Vec::new();
pal.push("step");
pal.push("on");
pal.push("no");
pal.push("pets");
assert_eq!(pal, vec!["step", "on", "no", "pets"]);
```

还有一种可能性是从迭代器生成的值构建一个向量：

```
let v: Vec<i32> = (0..5).collect();
assert_eq!(v, [0, 1, 2, 3, 4]);
```

使用 `collect` 时，通常要指定类型（正如此处给 `v` 声明了类型），因为它可以构建出不同种类的集合，而不仅仅是向量。通过指定 `v` 的类型，我们明确表达了自己想要哪种集合。

与数组一样，可以对向量使用切片的方法：

```
// 回文！
let mut palindrome = vec!["a man", "a plan", "a canal", "panama"];
palindrome.reverse();
// 固然合理，但不符合预期：
assert_eq!(palindrome, vec!["panama", "a canal", "a plan", "a
man"]);
```

在这里，`reverse` 方法实际上是在切片上定义的，但是此调用会隐式地从此向量中借用一个 `&mut [&str]` 切片并在其上调用 `reverse`。

`Vec` 是 Rust 的基本数据类型，它几乎可以用在任何需要动态大小的列表的地方，所以还有许多其他方法可以构建新向量或扩展现有向量。第 16 章会介绍这些方法。

`Vec<T>` 由 3 个值组成：指向元素在堆中分配的缓冲区（该缓冲区由 `Vec<T>` 创建并拥有）的指针、缓冲区能够存储的元素数量，以及它现在实际包含的数量（也就是它的长度）。当缓冲区达到其最大容量时，往向量中添加另一个元素需要分配一个更大的缓冲区，将当前内容复制到其中，更新向量的指针和容量以指向新缓冲区，最后释放旧缓冲区。

如果事先知道向量所需的元素数量，就可以调用 `Vec::with_capacity` 而不是 `Vec::new` 来创建一个向量，它的缓冲区足够大，可以从一开始就容纳所有元素。然后，可以逐个将元素添加到此向量中，而不会导致任何重新分配。`vec!` 宏就使用了这

样的技巧，因为它知道最终向量将包含多少个元素。请注意，这只会确定向量的初始大小，如果大小超出了你的预估，则向量仍然会正常扩大其存储空间。

许多库函数会寻求使用 `Vec::with_capacity` 而非 `Vec::new` 的机会。例如，在 `collect` 示例中，迭代器 `0..5` 预先知道它将产生 5 个值，并且 `collect` 函数会利用这一点以正确的容量来预分配它返回的向量。第 15 章会介绍其工作原理。

向量的 `len` 方法会返回它现在包含的元素数，而 `capacity` 方法则会返回在不重新分配的情况下可以容纳的元素数：

```
let mut v = Vec::with_capacity(2);
assert_eq!(v.len(), 0);
assert_eq!(v.capacity(), 2);

v.push(1);
v.push(2);
assert_eq!(v.len(), 2);
assert_eq!(v.capacity(), 2);

v.push(3);
assert_eq!(v.len(), 3);
// 通常会打印出"capacity is now 4":
println!("capacity is now {}", v.capacity());
```

最后打印出的容量不能保证恰好是 4，但至少大于等于 3，因为此向量包含 3 个值。

可以在向量中任意位置插入元素和移除元素，不过这些操作会将受影响位置之后的所有元素向前或向后移动，因此如果向量很长就可能很慢：

```
let mut v = vec![10, 20, 30, 40, 50];

// 在索引为3的元素处插入35
v.insert(3, 35);
assert_eq!(v, [10, 20, 30, 35, 40, 50]);

// 移除索引为1的元素
```

```
v.remove(1);
assert_eq!(v, [10, 30, 35, 40, 50]);
```

可以使用 `pop` 方法移除最后一个元素并将其返回。更准确地说，从 `Vec<T>` 中弹出一个值会返回 `Option<T>`：如果向量已经为空则为 `None`，如果其最后一个元素为 `v` 则为 `Some(v)`。

```
let mut v = vec!["Snow Puff", "Glass Gem"];
assert_eq!(v.pop(), Some("Glass Gem"));
assert_eq!(v.pop(), Some("Snow Puff"));
assert_eq!(v.pop(), None);
```

可以使用 `for` 循环遍历向量：

```
// 将命令行参数作为字符串的向量
let languages: Vec<String> = std::env::args().skip(1).collect();
for l in languages {
    println!("{}", l,
        if l.len() % 2 == 0 {
            "functional"
        } else {
            "imperative"
        });
}
```

以编程语言列表为参数运行本程序就可以说明这个问题：

```
$ cargo run Lisp Scheme C C++ Fortran
   Compiling proglangs v0.1.0 (/home/jimb/rust/proglangs)
   Finished dev [unoptimized + debuginfo] target(s) in 0.36s
   Running `target/debug/proglangs Lisp Scheme C C++ Fortran`
Lisp: functional
Scheme: functional
C: imperative
C++: imperative
Fortran: imperative
$
```

终于可以对术语函数式语言做一个令人满意的定义了。

虽然扮演着基础角色，但 `Vec` 仍然是 Rust 中定义的普通类型，而没有内置在语言中。第 22 章会介绍实现这些类型所需的技术。

3.6.3 切片

切片（写作不指定长度的 `[T]`）是数组或向量中的一个区域。由于切片可以是任意长度，因此它不能直接存储在变量中或作为函数参数进行传递。切片总是通过引用传递。

对切片的引用是一个**胖指针**：一个双字值，包括指向切片第一个元素的指针和切片中元素的数量。

假设你运行以下代码：

```
let v: Vec<f64> = vec![0.0, 0.707, 1.0, 0.707];
let a: [f64; 4] = [0.0, -0.707, -1.0, -0.707];

let sv: &[f64] = &v;
let sa: &[f64] = &a;
```

在最后两行中，Rust 自动把 `&Vec<f64>` 的引用和 `&[f64; 4]` 的引用转换成了直接指向数据的切片引用。

最后，内存布局如图 3-2 所示。

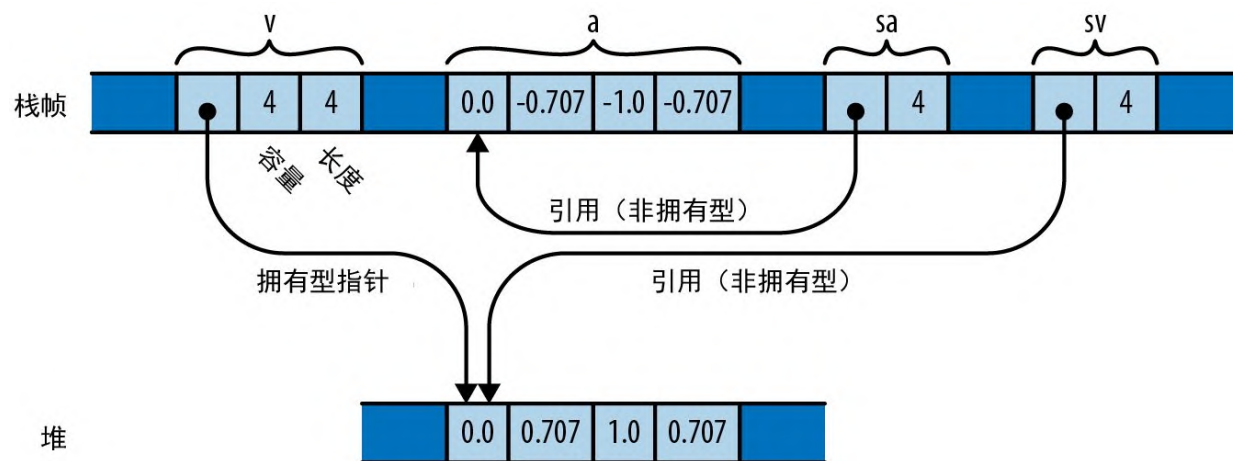


图 3-2：内存中的向量 `v` 和数组 `a` 分别被切片 `sa` 和 `sv` 引用

普通引用是指向单个值的非拥有型指针，而对切片的引用是指向内存中一系列连续值的非拥有型指针。如果要写一个对数组或向量进行操作

作的函数，那么切片引用就是不错的选择。例如，下面是打印一段数值的函数，每行一个：

```
fn print(n: &[f64]) {  
    for elt in n {  
        println!("{}", elt);  
    }  
}  
  
print(&a); // 打印数组  
print(&v); // 打印向量
```

因为此函数以切片引用为参数，所以也可以给它传入向量或数组。事实上，你以为属于向量或数组的许多方法其实是在切片上定义的，比如会对元素序列进行排序或反转的 `sort` 方法和 `reverse` 方法实际上是切片类型 `[T]` 上的方法。

你可以使用范围值对数组或向量进行索引，以获取一个切片的引用，该引用既可以指向数组或向量，也可以指向一个既有切片：

```
print(&v[0..2]); // 打印v的前两个元素  
print(&a[2..]); // 打印从a[2]开始的元素  
print(&sv[1..3]); // 打印v[1]和v[2]
```

与普通数组访问一样，Rust 会检查索引是否有效。尝试借用超出数据末尾的切片会导致 `panic`。

由于切片几乎总是出现在引用符号之后，因此通常只将 `&[T]` 或 `&str` 之类的类型称为“切片”，使用较短的名称来表示更常见的概念。

3.7 字符串类型

熟悉 C++ 的程序员应该还记得该语言中有两种字符串类型。字符串字面量的指针类型为 `const char *`。标准库还提供了一个 `std::string` 类，用于在运行期动态创建字符串。

Rust 中也有类似的设计。本节将首先展示所有编写字符串字面量的方法，然后介绍 Rust 的两种字符串类型。第 17 章会介绍有关字符串和文本处理的更多信息。

3.7.1 字符串字面量

字符串字面量要用双引号括起来，它们使用与 `char` 字面量相同的反斜杠转义序列：

```
let speech = "\"Ouch!\" said the well.\n";
```

但与 `char` 字面量不同，在字符串字面量中单引号不需要用反斜杠转义，而双引号需要。

一个字符串可能跨越多行：

```
println!("In the room the women come and go,  
Singing of Mount Abora");
```

该字符串字面量中的换行符是字符串的一部分，因此也会包含在输出中。第 2 行开头的空格也是如此。

如果字符串的一行以反斜杠结尾，那么就会丢弃其后的换行符和前导空格：

```
println!("It was a bright, cold day in April, and \  
there were four of us-\  
more or less.");
```

这会打印出单行文本。该字符串在“and”和“there”之间会有一个空格，因为在本程序中，第一个反斜杠之前有一个空格，而在破折号和“more”之间则没有空格。

在少数情况下，需要双写字符串中的每一个反斜杠，这让人不胜其烦。（经典的例子是正则表达式和 Windows 路径。）对于这些情况，Rust 提供了原始字符串。原始字符串用小写字母 `r` 进行标记。原始字符串中的所有反斜杠和空白字符都会逐字包含在字符串中。原始字符串不识别任何转义序列：

```
let default_win_install_path = r"C:\Program Files\Gorillas";

let pattern = Regex::new(r"\d+(\.\d+)*");
```

不能简单地在双引号前面放置一个反斜杠来包含原始字符串——别忘了，前面说过它不识别转义序列。但是，仍有办法解决。可以在原始字符串的开头和结尾添加 `#` 标记：

```
println!(r###"
    This raw string started with 'r###'.
    Therefore it does not end until we reach a quote mark ('')
    followed immediately by three pound signs ('###'):
"###);
```

可以根据需要添加任意多个井号，以标明原始字符串的结束位置。⁴

⁴ 原始字符串不要求井号的具体数量，只要求前后数量一致，且不会和内容相混淆。另外，原始字符串不会去掉前导空格。——译者注

3.7.2 字节串

带有 `b` 前缀的字符串字面量都是字节串。这样的字节串是 `u8` 值（字节）的切片而不是 Unicode 文本：

```
let method = b"GET";
assert_eq!(method, &[b'G', b'E', b'T']);
```

`method` 的类型是 `&[u8; 3]`：它是对 3 字节数组的引用，没有刚刚讨论过的任何字符串方法，最像字符串的地方就是其书写语法，仅此而已。

字节串可以使用前面展示过的所有其他的字符串语法：可以跨越多行、可以使用转义序列、可以使用反斜杠来连接行等。不过原始字节串要以 `br"` 开头。

字节串不能包含任意 Unicode 字符，它们只能使用 ASCII 和 `\xHH` 转义序列。

3.7.3 内存中的字符串

Rust 字符串是 Unicode 字符序列，但它们并没有以 `char` 数组的形式存储在内存中，而是使用了 UTF-8（一种可变宽度编码）的形式。字符串中的每个 ASCII 字符都会存储在单字节中，而其他字符会占用多字节。

图 3-3 展示了由以下代码创建的 `String` 值和 `&str` 值。

```
let noodles = "noodles".to_string();
let oodles = &noodles[1..];
let poodles = "ð_ð";
```

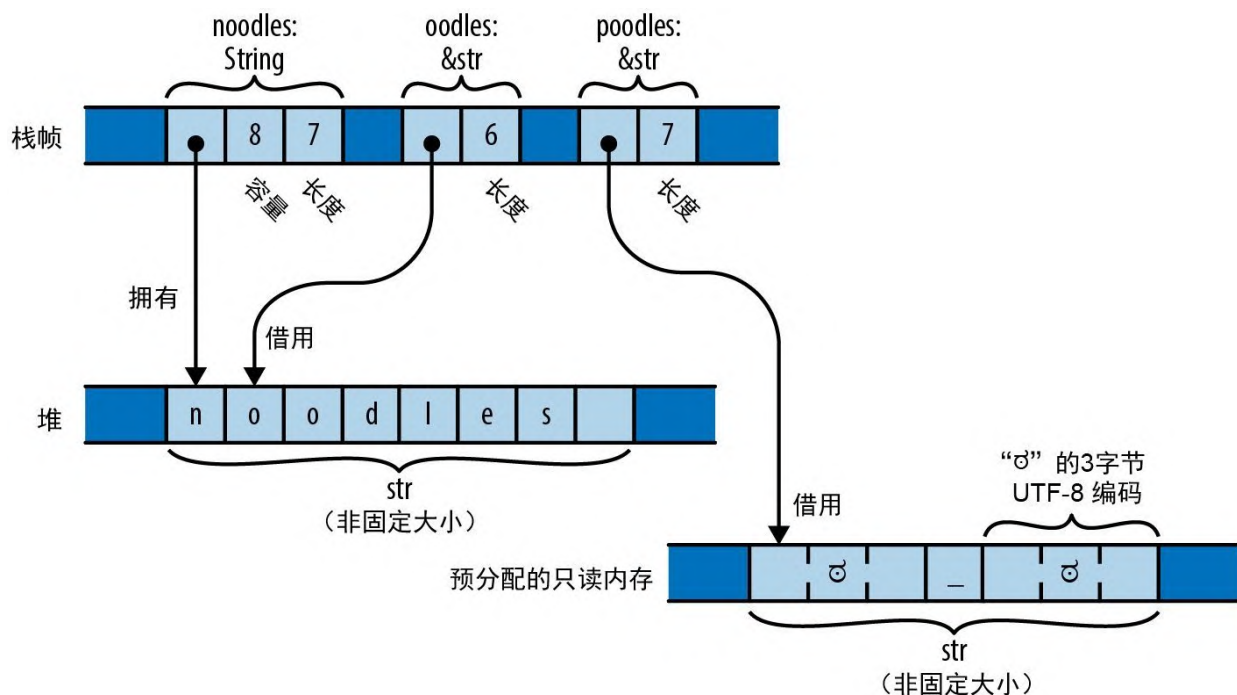


图 3-3: **String**、**&str** 和 **str**

String 有一个可以调整大小的缓冲区，其中包含 UTF-8 文本。缓冲区是在堆上分配的，因此它可以根据需要或请求来调整大小。在示例中，**noodles** 是一个 **String**，它拥有一个 8 字节的缓冲区，其中 7 字节正在使用中。可以将 **String** 视为 **Vec<u8>**，它可以保证包含格式良好的 UTF-8，实际上，**String** 就是这样实现的。

&str（发音为 /stɜːr/ 或 string slice）是对别人拥有的一系列 UTF-8 文本的引用，即它“借用”了这个文本。在示例中，**oodles** 是对 **noodles** 拥有的文本的最后 6 字节的一个 **&str** 引用，因此它表示文本“oodles”。与其他切片引用一样，**&str** 也是一个胖指针，包含实际数据的地址及其长度。可以认为 **&str** 就是 **&[u8]**，但它能保证包含的是格式良好的 UTF-8。

字符串字面量是指预分配文本的 **&str**，它通常与程序的机器码一起存储在只读内存区。在前面的示例中，**poodles** 是一个字符串字面量，指向一块 7 字节的内存，它在程序开始执行时就已创建并一直持续到程序退出。

String 或 &str 的 .len() 方法会返回其长度。这个长度以字节而不是字符为单位：

```
assert_eq!("ð_ð".len(), 7);
assert_eq!("ð_ð".chars().count(), 3);
```

不能修改 &str:

```
let mut s = "hello";
s[0] = 'c';      // 错误：无法修改`&str`，并给出错误原因
s.push('\n');    // 错误：`&str`引用上没有找到名为`push`的方法
```

要在运行期创建新字符串，可以使用 String。

&mut str 类型确实存在，但它没什么用，因为对 UTF-8 的几乎所有操作都会更改其字节总长度，但切片不能重新分配其引用目标的缓冲区。事实上，&mut str 上唯一可用的操作是 make_ascii_uppercase 和 make_ascii_lowercase，根据定义，它们会就地修改文本并且只影响单字节字符。

3.7.4 String

&str 非常像 &[T]，是一个指向某些数据的胖指针。而 String 则类似于 Vec<T>，如表 3-11 所示。

表 3-11: Vec<T> 与 String 对比

	Vec<T>	String
自动释放缓冲区	是	是
可增长	是	是

	Vec<T>	String
类型关联函数 <code>::new()</code> 和 <code>::with_capacity()</code>	是	是
<code>.reserve()</code> 方法和 <code>.capacity()</code> 方法	是	是
<code>.push()</code> 方法和 <code>.pop()</code> 方法	是	是
范围语法 <code>v[start..stop]</code>	是, 返回 <code>&[T]</code>	是, 返回 <code>&str</code>
自动转换	<code>&Vec<T></code> 到 <code>&[T]</code>	<code>&String</code> 到 <code>&str</code>
继承的方法	来自 <code>&[T]</code>	来自 <code>&str</code>

与 `Vec` 一样, 每个 `String` 都在堆上分配了自己的缓冲区, 不会与任何其他 `String` 共享。当 `String` 变量超出作用域时, 缓冲区将自动释放, 除非这个 `String` 已经被移动。

以下是创建 `String` 的几种方法。

- `.to_string()` 方法会将 `&str` 转换为 `String`。这会复制此字符串。

```
let error_message = "too many pets".to_string();
```

`.to_owned()` 方法会做同样的事, 也会以同样的方式使用。这种命名风格也适用于另一些类型, 第 13 章中会讨论。

- `format!()` 宏的工作方式与 `println!()` 类似，但它会返回一个新的 `String`，而不是将文本写入标准输出，并且不会在末尾自动添加换行符。

```
assert_eq!(format!("{}",°{:02}'{:02}"N", 24, 5, 23),
            "24°05'23"N".to_string());
```

- 字符串的数组、切片和向量都有两个方法（`.concat()` 和 `.join(sep)`），它们会从许多字符串中形成一个新的 `String`。

```
let bits = vec!["veni", "vidi", "vici"];
assert_eq!(bits.concat(), "venividivici");
assert_eq!(bits.join(", "), "veni, vidi, vici");
```

有时要选择是使用 `&str` 类型还是使用 `String` 类型。第 5 章会详细讨论这个问题。这里仅指出一点：`&str` 可以引用任意字符串的任意切片，无论它是字符串字面量（存储在可执行文件中）还是 `String`（在运行期分配和释放）。这意味着如果希望允许调用者传递任何一种字符串，那么 `&str` 更适合作为函数参数。

3.7.5 使用字符串

字符串支持 `==` 运算符和 `!=` 运算符。如果两个字符串以相同的顺序包含相同的字符（无论是否指向内存中的相同位置），则认为它们是相等的：

```
assert!("ONE".to_lowercase() == "one");
```

字符串还支持比较运算符 `<`、`<=`、`>` 和 `>=`，以及许多有用的方法和函数，你可以在“`str`（原始类型）”或“`std::str`”模块下的在线文档中找到它们（或直接翻到第 17 章）。下面是一些例子：


```
assert!("peanut".contains("nut"));
assert_eq!("ø_ø".replace("ø", "■"), "■_■");
assert_eq!("    clean\n".trim(), "clean");

for word in "veni, vidi, vici".split(", ") {
    assert!(word.starts_with("v"));
}
```

要记住，考虑到 Unicode 的性质，简单的逐字符比较并不总能给出预期的答案。例如，Rust 字符串 `"th\u{e9}"` 和 `"the\u{301}"` 都是 `thé`（在法语中是“茶”的意思）的有效 Unicode 表示。Unicode 规定它们应该以相同的方式显示和处理，但 Rust 会将它们视为两个完全不同的字符串。类似地，Rust 的排序运算符（如 `<`）也使用基于字符码点值的简单字典顺序。这种排序方式只能说近似于在用户的语言和文化环境中对文本的正确排序方式。⁵第 17 章会更详细地讨论这些问题。

⁵ 比如汉语就有拼音、笔画等排序方式，所以不能靠它做那些需要严格本地化场景下的排序。——译者注

3.7.6 其他类似字符串的类型

Rust 保证字符串是有效的 UTF-8。有时程序确实需要处理并非有效 Unicode 的字符串。这种情况通常发生在 Rust 程序不得不与不强制执行此类规则的其他系统进行互操作时，例如，在大多数操作系统中，很容易创建一个名字不符合 Unicode 规则的文件。当 Rust 程序遇到这种文件名时应该怎么办呢？

Rust 的解决方案是为这些情况提供一些类似字符串的类型。

- 对于 Unicode 文本，坚持使用 `String` 和 `&str`。
- 当使用文件名时，请改用 `std::path::PathBuf` 和 `&Path`。
- 当处理根本不是 UTF-8 编码的二进制数据时，请使用 `Vec<u8>` 和 `&[u8]`。
- 当使用操作系统提供的原生形式的环境变量名和命令行参数时，请使用 `OsString` 和 `&OsStr`。

- 当和使用 `null` 结尾字符串的 C 语言库进行互操作时，请使用 `std::ffi::CString` 和 `&CStr`。

3.8 类型别名

与 C++ 中的 `typedef` 用法类似，可以使用 `type` 关键字来为现有类型声明一个新名称：

```
type Bytes = Vec<u8>;
```

这里声明的类型 `Bytes` 就是这种特定 `Vec` 的简写形式。

```
fn decode(data: &Bytes) {  
    ...  
}
```

3.9 前路展望

类型是 Rust 的核心部分。接下来本书会继续讨论类型并引入一些新的类型。特别是，Rust 的用户定义类型赋予了该语言很多特色，因为各种方法都是在此基础上定义的。用户定义类型共有 3 种，我们将用连续 3 章（第 9 章、第 10 章和第 11 章）介绍它们。

函数和闭包都有自己的类型，第 14 章中会介绍。构成标准库的类型贯穿全书。例如，第 16 章会介绍标准的集合类型。

不过，所有这些都还得再等等。在继续前进之前，我们必须先着手处理 Rust 安全规则的核心概念。

第 4 章 所有权与移动¹

¹这里遵循了社区的习惯译法“移动”，学过 C++ 的读者可能比较熟悉了；对使用其他语言的读者来说，要特别注意这里的“移动”在语义上并非像真实生活中那样简单地挪动物品的位置，而是涉及一个非常重要的概念——所有权。在这个语义下，你可以把它理解为将值从一个所有者移交给另一个所有者，这里的重点是对所有权的转移，而所有权是 Rust 的核心概念。——译者注

谈及内存管理，我们希望编程语言能具备两个特点：

- 希望内存能在我们选定的时机及时释放，这使我们能控制程序的内存消耗；
- 在对象被释放后，我们绝不希望继续使用指向它的指针，这是未定义行为，会导致崩溃和安全漏洞。

但上述情景似乎难以兼顾：只要指向值的指针仍然存在，释放这个值就必然会让这些指针悬空。几乎所有主流编程语言都只能在两个阵营中“二选一”，这取决于它们从中放弃了哪一项。

- “安全优先”阵营会通过垃圾回收机制来管理内存，在所有指向对象的可达指针都消失后，自动释放对象。它通过简单地保留对象，直到再也没有指向它们的指针为止，来消除悬空指针。几乎所有现代语言都属于这个阵营，从 Python、JavaScript 和 Ruby 到 Java、C# 和 Haskell。

但是依赖垃圾回收，就意味着放弃了对于释放对象时机的精准控制，完全委托给回收器代管。一般来说，垃圾回收器就像奇怪的野兽般难以捉摸，要理解内存为何没有在预期的时机被释放可能颇具挑战。

- “控制优先”阵营会让你自己负责释放内存。程序的内存消耗完全掌握在你的手中，但避免悬空指针也完全成了你的责任。C 和 C++ 是这个阵营中仅有的两种主流语言。

如果你永不犯错，这当然是很好的选择，但事实证明，只要是人就会犯错。从已收集的安全报告数据来看，指针滥用一直都是引发问题的罪魁祸首。

Rust 的目标是既安全又高效，所以这两种妥协都是无法接受的。但如果很容易两者兼得，那应该早就有人做到了。看来我们需要做一些根本性的变革。

Rust 通过限制程序使用指针的方式出人意料地打破了这种困局。本章和第 5 章将专门解释这些限制是什么以及它们为什么能起作用。现在，只需要知道一些惯用的常见结构可能不符合这些限制规则，而你要寻找替代方案。施加这些限制的最终目的是在混沌中建立足够的秩序，以便让 Rust 的编译期检查器有能力验证程序中是否存在内存安全错误：悬空指针、重复释放、使用未初始化的内存等。在运行期，指针仅仅是内存中的地址，和在 C 与 C++ 中一样。而不一样的是，Rust 编译器已然证明你的代码在安全地使用它们。

这些规则同样构成了 Rust 支持安全并发编程的基础。使用 Rust 精心设计的线程原语，那些确保代码在正确使用内存的规则，同样可以用于证明代码中不存在数据竞争。Rust 程序中的缺陷不会导致一个线程破坏另一个线程的数据，进而在系统中的无关部分引入难以重现的故障。多线程代码中的固有不确定性被隔离到了那些专门设计来处理它们的线程特性（比如互斥锁、消息通道、原子值等）上，而不必出现在普通的内存引用中。用 C 和 C++ 编写的多线程代码饱受诟病，但 Rust 很好地改变了这种局面。

Rust 的激进“赌注”是其成功的基础和语言的根源。即使有这种限制，该语言依然足够灵活，可以完成几乎所有任务，并且可以消除各种内存管理和并发错误。这些优点将会证明你值得调整自己的风格来适应它。正是因为我们（本书作者）在 C 和 C++ 方面拥有丰富的经验，所以才更加看好 Rust。对我们来说，与 Rust 的这项交易非常划算。

Rust 的一些规则可能与你在其他编程语言中看到的截然不同。在我们看来，学习 Rust 的核心挑战，就是学习如何用好这些规则并转化为你的优势。在本章中，我们将首先展示同一个根本问题在不同语言中

的表现形式，以深入了解 Rust 规则背后的逻辑和意图。然后，我们将详细解释 Rust 的规则，看看所有权在概念层和实现层分别意味着什么、如何在各种场景中跟踪所有权的变化，以及在哪些情况下要改变或打破其中的一些规则，以提供更大的灵活性。

4.1 所有权

如果你读过大量 C 或 C++ 代码，可能遇到过这样的注释，即某个类的实例拥有它指向的某个其他对象。通常，拥有对象意味着可以决定何时释放此对象：当销毁拥有者时，它拥有的对象也会随之销毁。

假如有如下 C++ 代码：

```
std::string s = "frayed knot";
```

通常，字符串 `s` 在内存中的表示如图 4-1 所示。

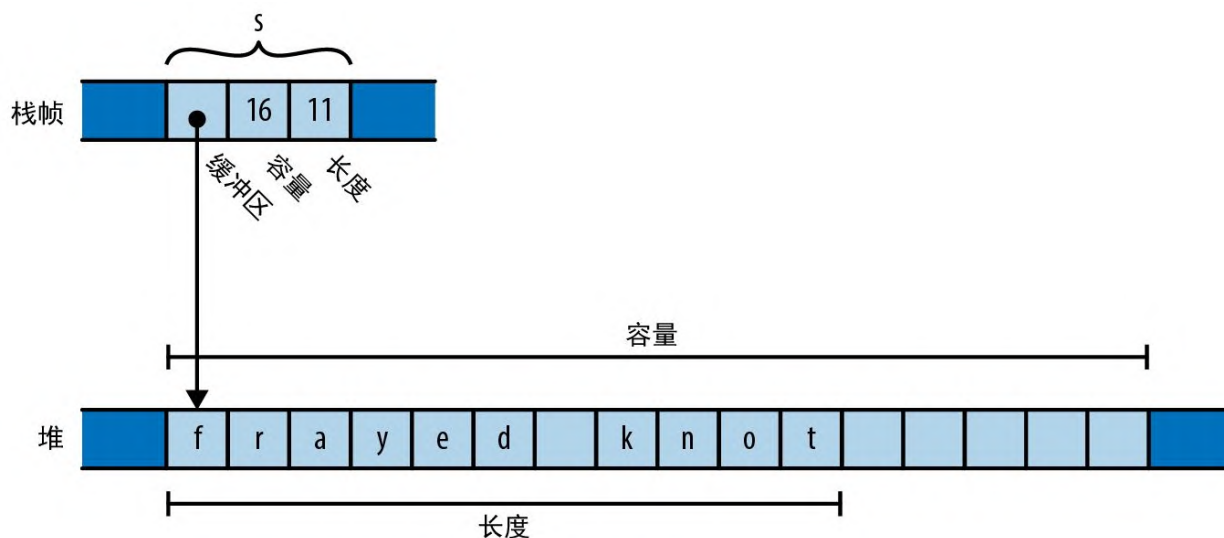


图 4-1：栈上的 C++ `std::string` 值，指向其在堆上分配的缓冲区

在这里，实际的 `std::string` 对象本身总是正好有 3 个机器字长，包括指向分配在堆上的缓冲区的指针、缓冲区的总容量（在不得不为字符串分配更大的缓冲区之前，文本可以增长到多大），以及当前持有的文本的长度。这些都是 `std::string` 类私有的字段，使用者无法访问。

`std::string` 拥有自己的缓冲区：当程序销毁字符串时，字符串的析构函数会释放缓冲区。以前，一些 C++ 库会在多个 `std::string`

值之间共享同一个缓冲区，通过引用计数来决定何时释放此缓冲区。但较新版本的 C++ 规范有效地杜绝了这种表示法，所有现代 C++ 库使用的都是这里展示的方法。

在这些情况下，人们普遍认为，虽然其他代码也可以创建指向所拥有内存的临时指针，但在拥有者决定销毁拥有的对象之前，其他代码有责任确保其指针已消失。也就是说，你可以创建一个指向 `std::string` 的缓冲区中的字符的指针，但是当字符串被销毁时，你也必须让你的指针失效，并且要确保不再使用它。拥有者决定被拥有者的生命周期，其他所有人都必须尊重其决定。

这里使用了 `std::string` 作为 C++ 中所有权的示例：它只是标准库通常遵循的规约，尽管 C++ 鼓励人们都遵循类似的做法，但说到底，如何设计自己的类型还是要由你自己决定。

然而，在 Rust 中，所有权这个概念内置于语言本身，并通过编译期检查强制执行。每个值都有决定其生命周期的唯一拥有者。当拥有者被释放时，它拥有的值也会同时被释放，在 Rust 术语中，释放的行为被称为**丢弃**（drop）。这些规则便于通过检查代码确定任意值的生命周期，也提供了系统级语言本应支持的对生命周期的控制。

变量拥有自己的值，当控制流离开声明变量的块时，变量就会被丢弃，因此它的值也会一起被丢弃。例如：

```
fn print_padovan() {
    let mut padovan = vec![1,1,1]; // 在此分配
    for i in 3..10 {
        let next = padovan[i-3] + padovan[i-2];
        padovan.push(next);
    }
    println!("P(1..10) = {:?}", padovan);
}                                     // 在此丢弃
```

变量 `padovan` 的类型为 `Vec<i32>`，即一个 32 位整数向量。在内存中，`padovan` 的最终值如图 4-2 所示。

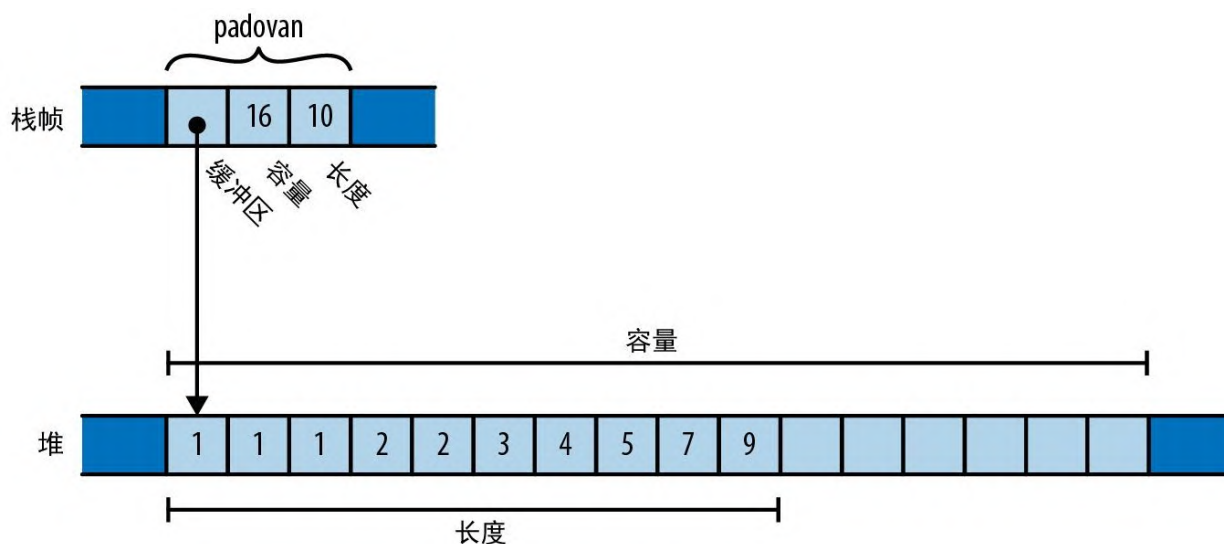


图 4-2: 栈上的 `Vec<i32>`, 指向其在堆中的缓冲区

这和之前展示过的 C++ `std::string` 非常相似，不过缓冲区中的元素都是 32 位整数，而不是字符。请注意，保存 `padovan` 指针、容量和长度的字都直接位于 `print_padovan` 函数的栈帧中，只有向量的缓冲区才分配在堆上。

和之前的字符串 `s` 一样，此向量拥有保存其元素的缓冲区。当变量 `padovan` 在函数末尾超出作用域时，程序将会丢弃此向量。因为向量拥有自己的缓冲区，所以此缓冲区也会一起被丢弃。

Rust 的 `Box` 类型是所有权的另一个例子。`Box<T>` 是指向存储在堆上的 `T` 类型值的指针。可以调用 `Box::new(v)` 分配一些堆空间，将值 `v` 移入其中，并返回一个指向该堆空间的 `Box`。因为 `Box` 拥有它所指向的空间，所以当丢弃 `Box` 时，也会释放此空间。

例如，可以像下面这样在堆中分配一个元组：

```
{
    let point = Box::new((0.625, 0.5)); // 在此分配了point
    let label = format!("{:?}", point); // 在此分配了label
    assert_eq!(label, "(0.625, 0.5)");
}
```

// 在此全都丢弃了

当程序调用 `Box::new` 时，它会在堆上为由两个 `f64` 值构成的元组分配空间，然后将其参数 `(0.625, 0.5)` 移进去，并返回指向该空间的指针。当控制流抵达对 `assert_eq!` 的调用时，栈帧如图 4-3 所示。

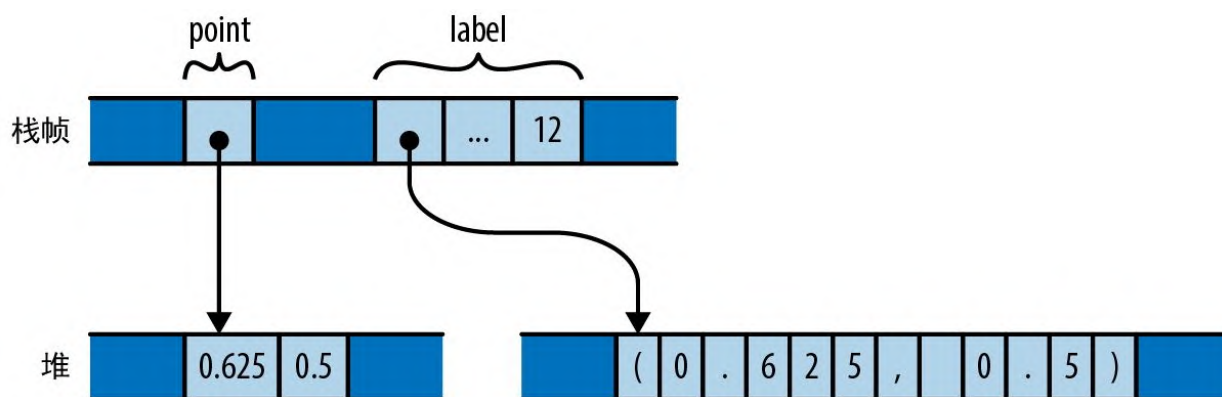


图 4-3：两个局部变量，它们各自在堆中拥有内存

栈帧本身包含变量 `point` 和 `label`，其中每个变量都指向其拥有的堆中内存。当丢弃它们时，它们拥有的堆中内存也会一起被释放。

就像变量拥有自己的值一样，结构体拥有自己的字段，元组、数组和向量则拥有自己的元素。

```
struct Person { name: String, birth: i32 }

let mut composers = Vec::new();
composers.push(Person { name: "Palestrina".to_string(),
                        birth: 1525 });
composers.push(Person { name: "Dowland".to_string(),
                        birth: 1563 });
composers.push(Person { name: "Lully".to_string(),
                        birth: 1632 });
for composer in &composers {
    println!("{}", born {}, composer.name, composer.birth);
}
```

在这里，`composers` 是一个 `Vec<Person>`，即由结构体组成的向量，每个结构体都包含一个字符串和数值。在内存中，`composers` 的最终值如图 4-4 所示。

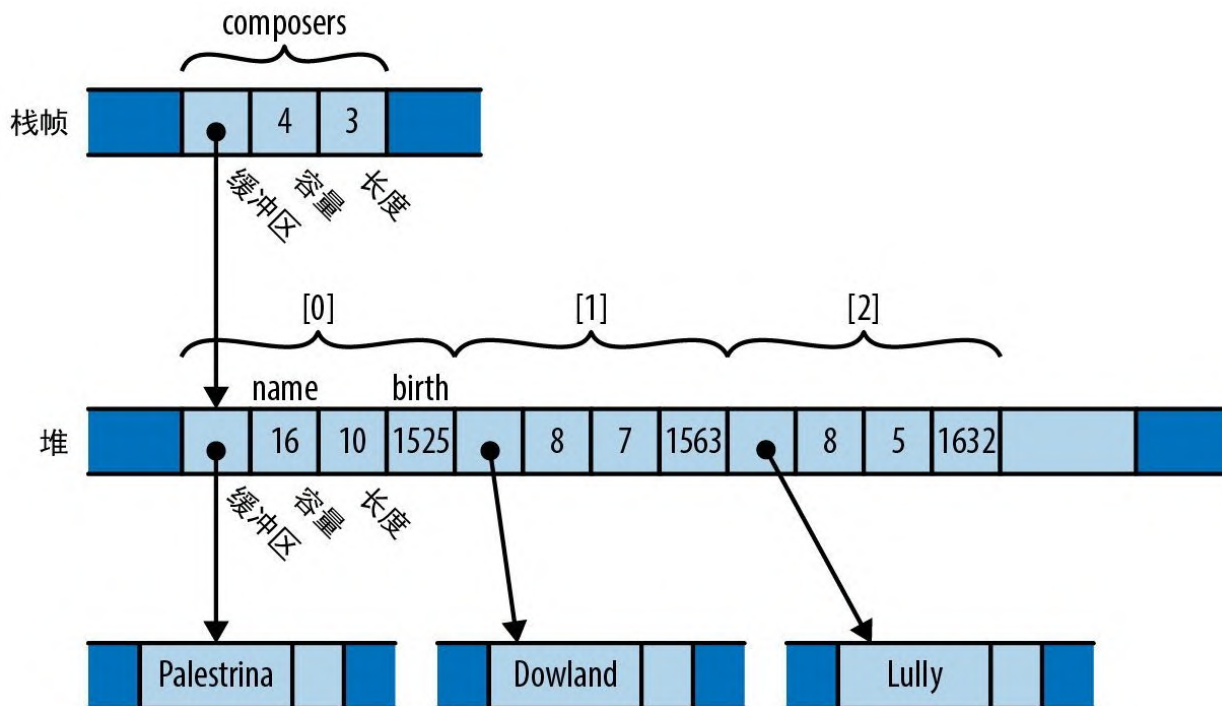


图 4-4：更复杂的所有权树

这里有很多所有权关系，但每个都一目了然：`composers` 拥有一个向量，向量拥有自己的元素，每个元素都是一个 `Person` 结构体，每个结构体都拥有自己的字段，并且字符串字段拥有自己的文本。当控制流离开声明 `composers` 的作用域时，程序会丢弃自己的值并将整棵所有权树一起丢弃。如果还存在其他类型的集合（可能是 `HashMap` 或 `BTreeSet`），那么处理的方式也是一样的。

现在，回过头来思考一下刚刚介绍的这些所有权关系的重要性。每个值都有一个唯一的拥有者，因此很容易决定何时丢弃它。但是每个值可能会拥有许多其他值，比如向量 `composers` 会拥有自己的所有元素。这些值还可能拥有其他值：`composers` 的每个元素都各自拥有一个字符串，该字符串又拥有自己的文本。

由此可见，拥有者及其拥有的那些值形成了一棵树：值的拥有者是值的父节点，值拥有的值是值的子节点。每棵树的总根都是一个变量，当该变量超出作用域时，整棵树都将随之消失。可以在 `composers` 的图中看到这样的所有权树：它既不是“搜索树”那种数据结构意义上的“树”，也不是由 DOM 元素构成的 HTML 文档。相反，我们有一

棵由混合类型构建的树，Rust 的单一拥有者规则将禁止任何可能让它们排列得比树结构更复杂的可能性。Rust 程序中的每一个值都是某棵树的成员，树根是某个变量。

Rust 程序通常不需要像 C 程序和 C++ 程序那样显式地使用 `free` 和 `delete` 来丢弃值。在 Rust 中丢弃一个值的方式就是从所有权树中移除它：或者离开变量的作用域，或者从向量中删除一个元素，或者执行其他类似的操作。这样一来，Rust 就会确保正确地丢弃该值及其拥有的一切。

从某种意义上说，Rust 确实不如其他语言强大：其他所有实用的编程语言都允许你构建出任意复杂的对象图，这些对象可以用你认为合适的方式相互引用。但正是因为 Rust 不那么强大，所以编辑器对你的程序所进行的分析才能更强大。Rust 的安全保证之所以可行，是因为在你的代码中可能出现的那些关系都更可控。这是之前提过的 Rust 的“激进赌注”的一部分：实际上，Rust 声称，解决问题的方式通常是灵活多样的，因此总是会有一些完美的解决方案能同时满足它所施加的限制。

迄今为止，我们已经解释过的这些所有权概念仍然过于严格，还处理不了某些场景。Rust 从几个方面扩展了这种简单的思想。

- 可以将值从一个拥有者转移给另一个拥有者。这允许你构建、重新排列和拆除树形结构。
- 像整数、浮点数和字符这样的非常简单的类型，不受所有权规则的约束。这些称为 `Copy` 类型。
- 标准库提供了引用计数指针类型 `Rc` 和 `Arc`，它们允许值在某些限制下有多个拥有者。
- 可以对值进行“借用”（`borrow`），以获得值的引用。这种引用是非拥有型指针，有着受限的生命周期。

这些策略中的每一个策略都为所有权模型带来了灵活性，同时仍然坚持着 Rust 的那些承诺。接下来我们将依次解释这几种方式，并会在第 5 章中介绍一些参考资料。

4.2 移动

在 Rust 中，对大多数类型来说，像为变量赋值、将其传给函数或从函数返回这样的操作都不会复制值，而是会**移动**值。源会把值的所有权转移给目标并变回未初始化状态，改由目标变量来控制值的生命周期。Rust 程序会以每次只移动一个值的方式建立和拆除复杂的结构。

你可能惊讶于 Rust 会改变这些基本操作的含义，确实如此，历史发展到今天，赋值应该已经是含义最明确的操作了。但是，如果仔细观察不同的语言处理赋值操作的方式，你会发现不同的编程流派之间实际上存在着相当明显的差异。对比这些差异也能很容易看出 Rust 做出这种选择的意义及其重要性。

考虑以下 Python 代码：

```
s = ['udon', 'ramen', 'soba']  
t = s  
u = s
```

每个 Python 对象都有一个引用计数，以用于跟踪当前正引用着此值的数量。因此，在对 s 赋值之后，程序的状态如图 4-5 所示。（请注意，这里忽略了一些字段。）

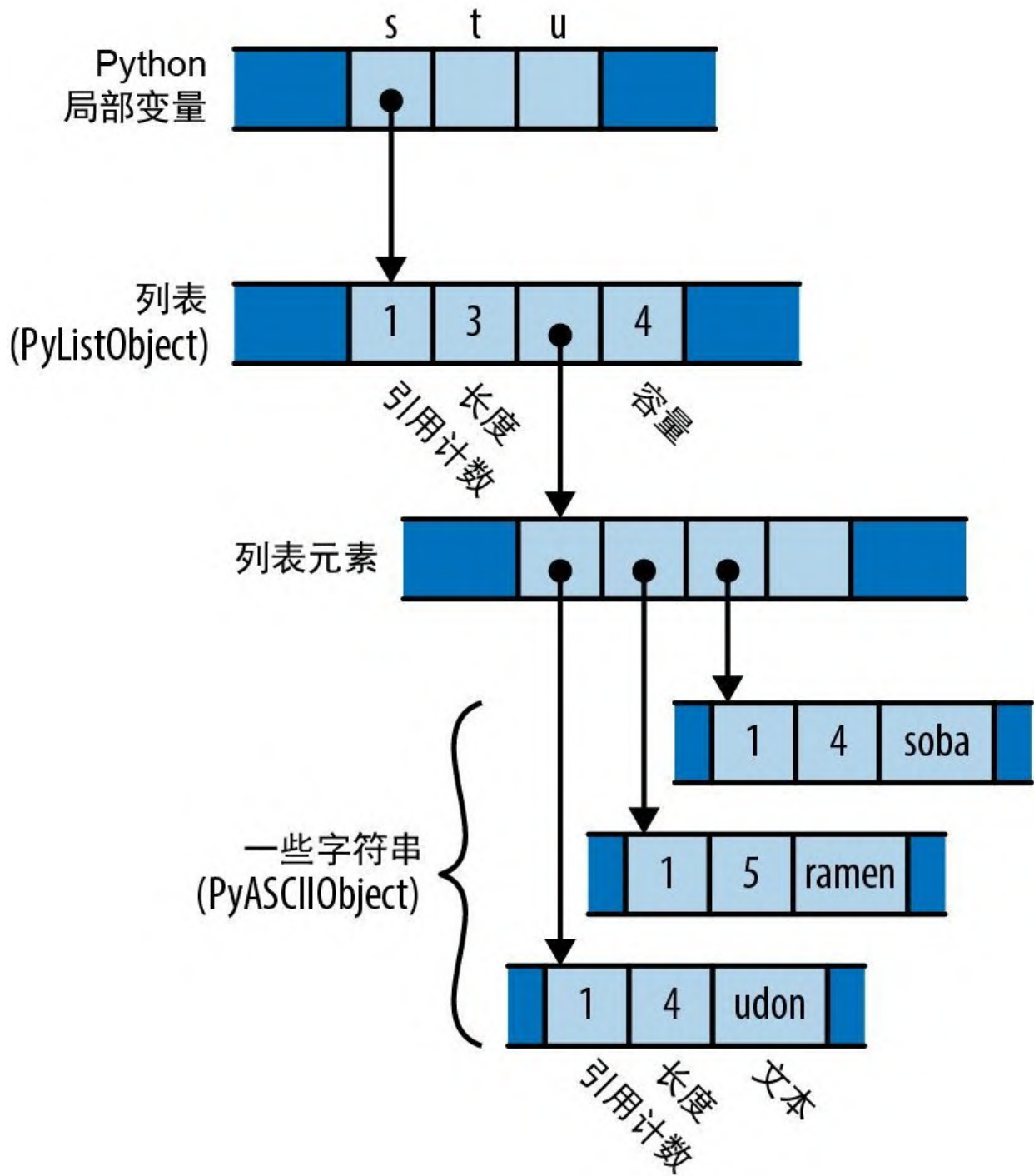


图 4-5: Python 如何在内存中表示字符串列表

由于只有 `s` 指向列表，因此该列表的引用计数为 1。由于列表是唯一指向这些字符串的对象，因此它们各自的引用计数也是 1。

当程序执行对 `t` 和 `u` 的赋值时会发生什么？Python 会直接让目标指向与源相同的对象，并增加对象的引用计数来实现赋值。所以程序的最终状态如图 4-6 所示。

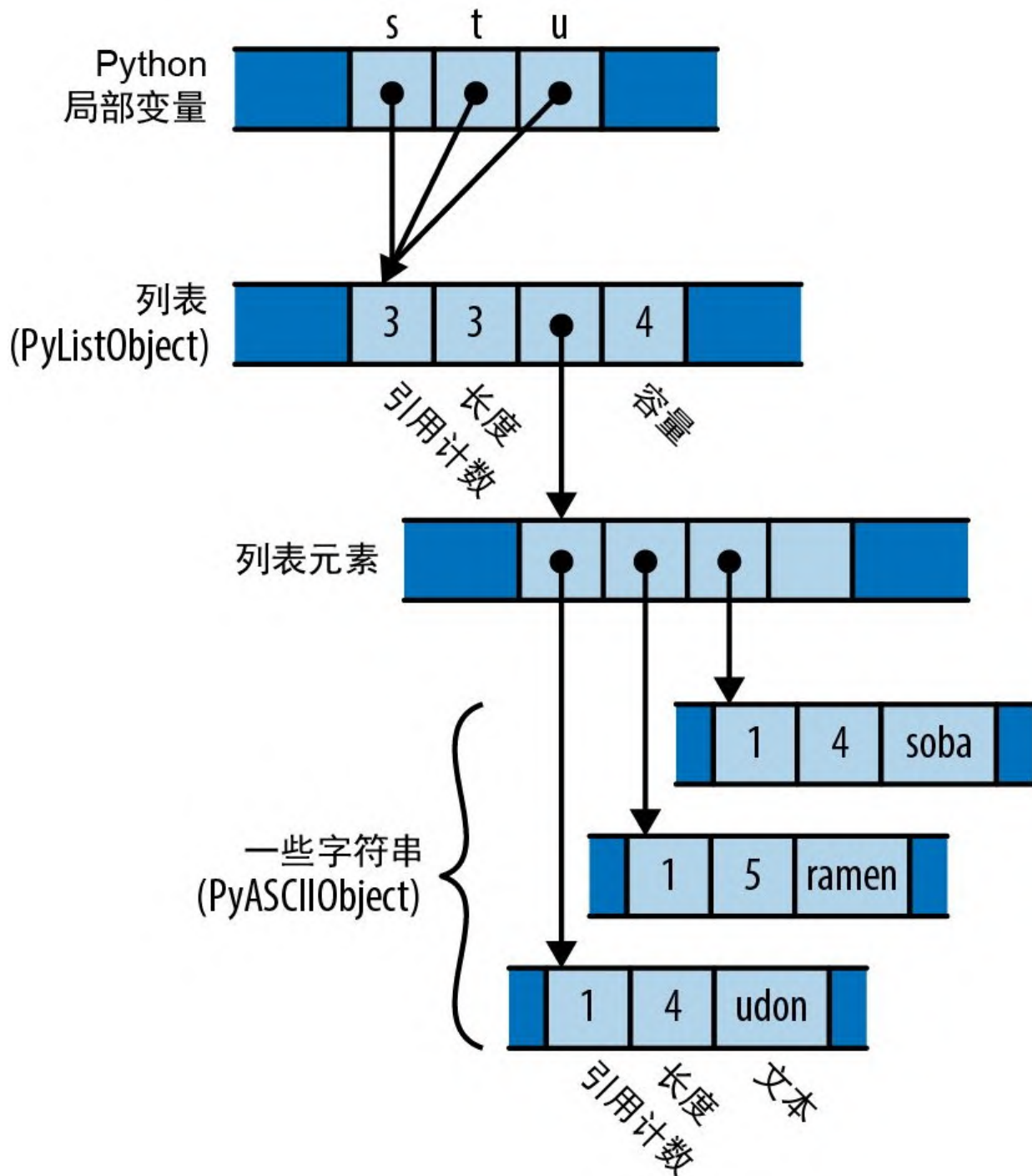


图 4-6: 在 Python 中将 **s** 赋值给 **t** 和 **u** 的结果

Python 已经将指针从 **s** 复制到 **t** 和 **u**，并将此列表的引用计数更新为 3。Python 中的赋值开销极低，但因为它创建了对对象的新引用，所以必须维护引用计数才能知道何时可以释放该值。

现在考虑类似的 C++ 代码：

```
using namespace std;
vector<string> s = { "udon", "ramen", "soba" };
vector<string> t = s;
vector<string> u = s;
```

s 的原始值在内存中如图 4-7 所示。

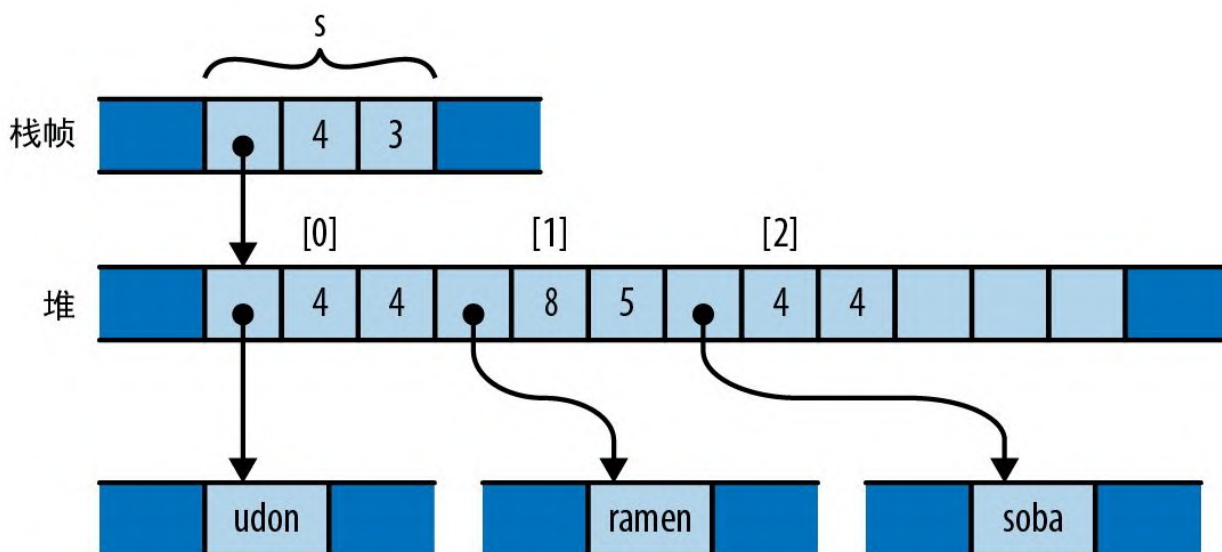


图 4-7: C++ 如何表示内存中的字符串向量

当程序将 **s** 赋值给 **t** 和 **u** 时会发生什么？在 C++ 中，把 `std::vector` 赋值给其他元素会生成一个向量的副本，`std::string` 的行为也类似。所以当程序执行到这段代码的末尾时，它实际上已经分配了 3 个向量和 9 个字符串，如图 4-8 所示。

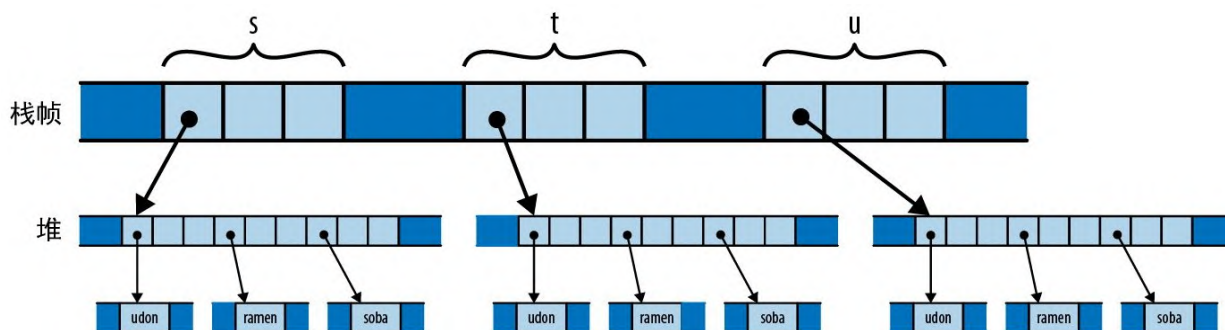


图 4-8：在 C++ 中将 **s** 赋值给 **t** 和 **u** 的结果

理论上，如果涉及某些特定的值，那么 C++ 中的赋值可能会消耗超乎想象的内存和处理器时间。然而，其优点是程序很容易决定何时释放这些内存：当变量超出作用域时，此处分配的所有内容都会自动清除。

从某种意义上说，C++ 和 Python 选择了相反的权衡：Python 以需要引用计数（以及更广泛意义上的垃圾回收）为代价，让赋值的开销变得非常低。C++ 则选择让全部内存的所有权保持清晰，而代价是在赋值时要执行对象的深拷贝。一般来说，C++ 程序员不太热衷这种选择：深拷贝的开销可能很昂贵，而且通常有更实用的替代方案。

那么类似的程序在 Rust 中会怎么做呢？请看如下代码：

```
let s = vec!["udon".to_string(), "ramen".to_string(),
"soba".to_string()];
let t = s;
let u = s;
```

与 C 和 C++ 一样，Rust 会将纯字符串字面量（如 "udon"）放在只读内存中，因此为了与 C++ 示例和 Python 示例进行更清晰的比较，此处调用了 `to_string` 以获取堆上分配的 `String` 值。

在执行了 `s` 的初始化之后，由于 Rust 和 C++ 对向量和字符串使用了类似的表示形式，因此情况看起来就和 C++ 中一样，如图 4-9 所示。

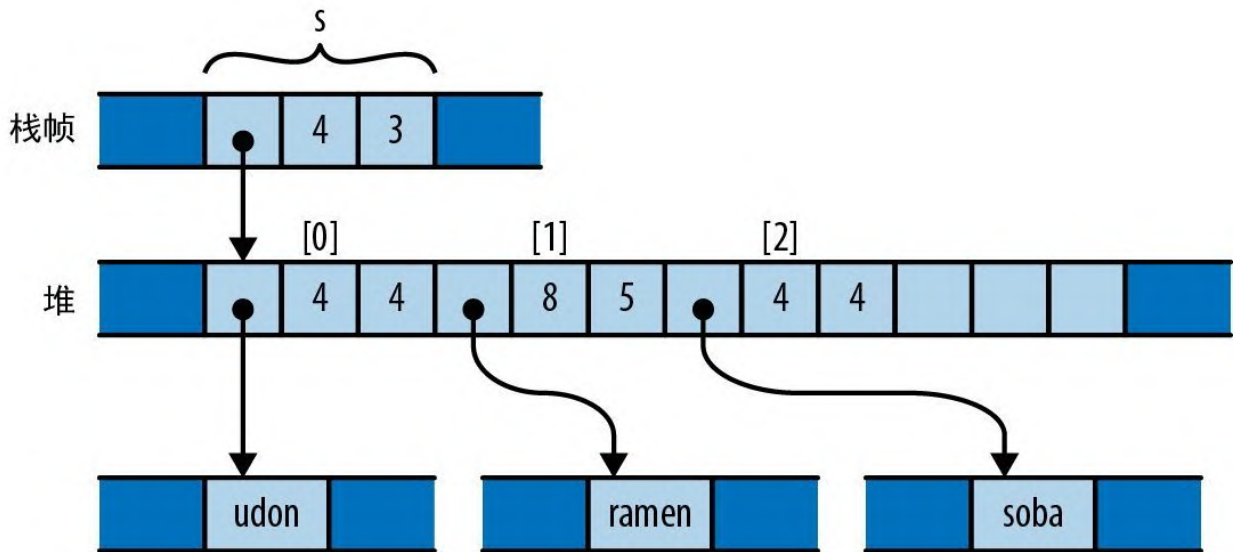


图 4-9: Rust 如何表示内存中的字符串向量

但要记住，在 Rust 中，大多数类型的赋值会将值从源转移给目标，而源会回到未初始化状态。因此在初始化 `t` 之后，程序的内存如图 4-10 所示。

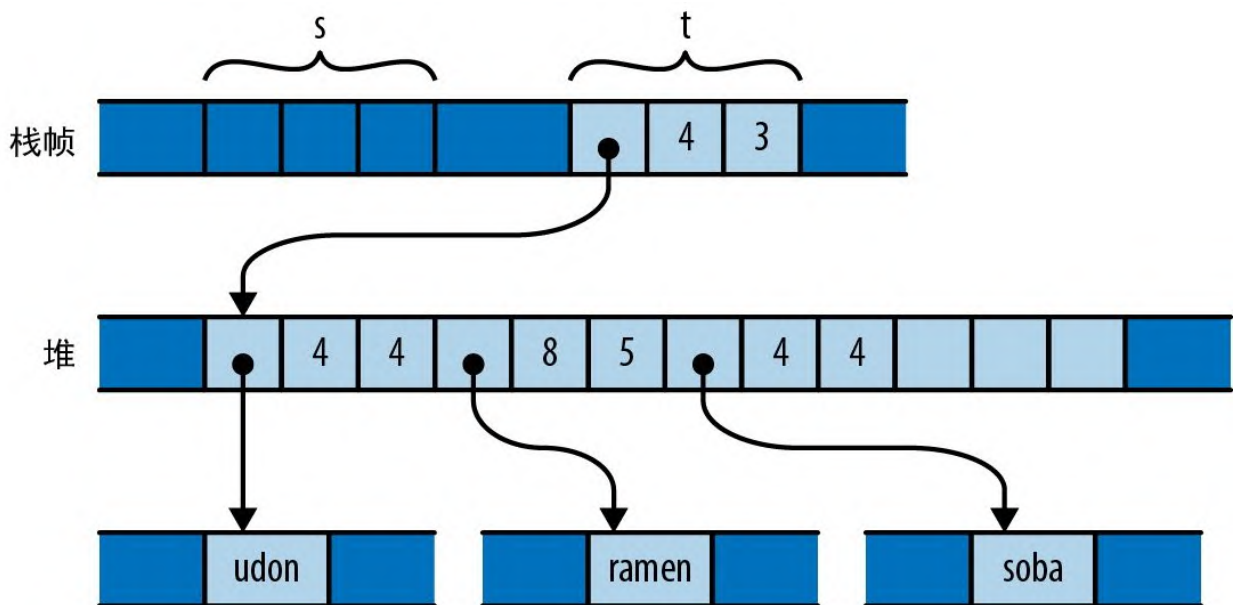


图 4-10: Rust 中将 `s` 赋值给 `t` 的结果

这里发生了什么？初始化语句 `let t = s;` 将向量的 3 个标头字段从 `s` 转移给了 `t`，现在 `t` 拥有此向量。向量的元素保持原样，字符串也没有任何变化。每个值依然只有一个拥有者，尽管其中一个已然易手。整个过程中没有需要调整的引用计数，不过编译器现在会认为 `s` 是未初始化状态。

那么当我们执行初始化语句 `let u = s;` 时会发生什么呢？这会将尚未初始化的值 `s` 赋给 `u`。Rust 明智地禁止使用未初始化的值，因此编译器会拒绝此代码并报告如下错误：

```
error: use of moved value: `s`
  |
7 |     let s = vec!["udon".to_string(), "ramen".to_string(),
  |     - move occurs because `s` has type `Vec<String>`,
  |       which does not implement the `Copy` trait
8 |     let t = s;
  |           - value moved here
9 |     let u = s;
  |           ^ value used here after move
```

思考一下 Rust 在这里使用移动语义的影响。与 Python 一样，赋值操作开销极低：程序只需将向量的三字标头从一个位置移到另一个位置即可。但与 C++ 一样，所有权始终是明确的：程序不需要引用计数或垃圾回收就能知道何时释放向量元素和字符串内容。

代价是如果需要同时访问它们，就必须显式地要求复制。如果想达到与 C++ 程序相同的状态（每个变量都保存一个独立的结构副本），就必须调用向量的 `clone` 方法，该方法会执行向量及其元素的深拷贝：

```
let s = vec!["udon".to_string(), "ramen".to_string(),
"soba".to_string()];
let t = s.clone();
let u = s.clone();
```

还可以使用 Rust 的引用计数来实现与 Python 类似的操作，4.4 节会简要讨论这些内容。

4.2.1 更多移动类操作

在先前的例子中，我们已经展示了如何初始化工作——在变量进入 `let` 语句的作用域时为它们提供值。给变量赋值则与此略有不同，如果你将一个值转移给已初始化的变量，那么 Rust 就会丢弃该变量的先前值。例如：

```
let mut s = "Govinda".to_string();  
s = "Siddhartha".to_string(); // 在这里丢弃了值"Govinda"
```

在上述代码中，当程序将字符串 "Siddhartha" 赋值给 `s` 时，它的先前值 "Govinda" 会首先被丢弃。但请考虑以下代码：

```
let mut s = "Govinda".to_string();  
let t = s;  
s = "Siddhartha".to_string(); // 这里什么也没有丢弃
```

这一次，`t` 从 `s` 接手了原始字符串的所有权，所以当给 `s` 赋值时，它是未初始化状态。这种情况下不会丢弃任何字符串。

我们在这个例子中使用了初始化和赋值，因为它们很简单，但 Rust 还将“移动”的语义应用到了几乎所有对值的使用上。例如，将参数传给函数会将所有权转移给函数的参数、从函数返回一个值会将所有权转移给调用者、构建元组会将值转移给元组。

你现在可以更好地理解 4.1 节的示例中到底发生过什么了。例如，我们在构建 `composers` 向量时，是这样写的：

```
struct Person { name: String, birth: i32 }  
  
let mut composers = Vec::new();  
composers.push(Person { name: "Palestrina".to_string(),  
                        birth: 1525 });
```

这段代码展示了除初始化和赋值之外发生移动的几个地方。

从函数返回值

调用 `Vec::new()` 构造一个新向量并返回，返回的不是指向此向量的指针，而是向量本身：它的所有权从 `Vec::new` 转移给了变量 `composers`。同样，`to_string` 调用返回的是一个新的 `String` 实例。

构造出新值

新 `Person` 结构体的 `name` 字段是用 `to_string` 的返回值初始化的。该结构体拥有这个字符串的所有权。

将值传给函数

整个 `Person` 结构体（不是指向它的指针）被传给了向量的 `push` 方法，此方法会将该结构体移动到向量的末尾。向量接管了 `Person` 的所有权，因此也间接接手了 `name` 这个 `String` 的所有权。

像这样移动值乍一看可能效率低下，但有两点需要牢记。首先，移动的永远是值本身，而不是这些值拥有的堆存储。对于向量和字符串，值本身就是指单独的“三字标头”，幕后的大型元素数组和文本缓冲区仍然位于它们在堆中的位置。其次，`Rust` 编译器在生成代码时擅长“看穿”这一切动作。在实践中，机器码通常会将值直接存储在它应该在的位置。²

²这意味着甚至可以没有运行期开销。——译者注

4.2.2 移动与控制流

前面的例子中都有非常简单的控制流，那么该如何在更复杂的代码中移动呢？一般性原则是，如果一个变量的值有可能已经移走，并且从那以后尚未明确赋予其新值，那么它就可以被看作是未初始化状态。如果一个变量在执行了 `if` 表达式中的条件后仍然有值，那么就可以在这两个分支中使用它：

```
let x = vec![10, 20, 30];
if c {
    f(x); // .....可以在这里移动x
```

```
} else {  
    g(x); // .....也可以在这里移动x  
}  
h(x); // 错误：只要任何一条路径用过它，x在这里就是未初始化状态
```

出于类似的原因，禁止在循环中进行变量移动：

```
let x = vec![10, 20, 30];  
while f() {  
    g(x); // 错误：x已经在第一次迭代中移动出去了，在第二次迭代中，它成了未初始化状态  
}
```

也就是说，除非在下一次迭代中明确赋予 `x` 一个新值，否则就会出错。

```
let mut x = vec![10, 20, 30];  
while f() {  
    g(x);           // 从x移动出去了  
    x = h();        // 赋予x一个新值  
}  
e(x);
```

4.2.3 移动与索引内容

前面提到过，移动会令其来源变成未初始化状态，因为目标将获得该值的所有权。但并非值的每种拥有者都能变成未初始化状态。例如，考虑以下代码：

```
// 构建一个由字符串"101"、"102"....."105"组成的向量  
let mut v = Vec::new();  
for i in 101 .. 106 {  
    v.push(i.to_string());  
}  
  
// 从向量中随机抽取元素  
let third = v[2]; // 错误：不能移动到Vec索引结构之外3  
let fifth = v[4]; // 这里也一样
```

³注意这里写的是 `v[2]` 而非 `&v[2]`。——译者注

为了解决这个问题，Rust 需要以某种方式记住向量的第三个元素和第五个元素是未初始化状态，并要跟踪该信息直到向量被丢弃。通常的解决方案是，让每个向量都携带额外的信息来指示哪些元素是活动的，哪些元素是未初始化的。这显然不是系统编程语言应该做的。向量应该只是向量，不应该携带额外的信息或状态。事实上，Rust 会拒绝前面的代码并报告如下错误：

```
error: cannot move out of index of `Vec<String>`
|
14 |         let third = v[2];
    |                     ^^^^
    |                     |
    |                     move occurs because value has type `String`,
    |                     which does not implement the `Copy` trait
    |                     help: consider borrowing here: `&v[2]`
```

移动第五个元素时 Rust 也会报告类似的错误。在这条错误消息中，Rust 还建议使用引用，因为你可能只是想访问该元素而不是移动它，这通常确实是你想要做的。但是，如果真想将一个元素移出向量该怎么办呢？需要找到一种在遵循类型限制的情况下执行此操作的方法。以下是 3 种可能的方法：

```
// 构建一个由字符串"101"、"102"....."105"组成的向量
let mut v = Vec::new();
for i in 101 .. 106 {
    v.push(i.to_string());
}

// 方法一：从向量的末尾弹出一个值：
let fifth = v.pop().expect("vector empty!");
assert_eq!(fifth, "105");

// 方法二：将向量中指定索引处的值与最后一个值互换，并把前者移动出来：
let second = v.swap_remove(1);
assert_eq!(second, "102");

// 方法三：把要取出的值和另一个值互换：
let third = std::mem::replace(&mut v[2], "substitute".to_string());
assert_eq!(third, "103");

// 看看向量中还剩下什么
assert_eq!(v, vec!["101", "104", "substitute"]);
```


每种方法都能将一个元素移出向量，但仍会让向量处于完全填充状态，只是向量可能会变小。

像 `Vec` 这样的集合类型通常也会提供在循环中消耗所有元素的方法：

```
let v = vec!["liberté".to_string(),
             "égalité".to_string(),
             "fraternité".to_string()];

for mut s in v {
    s.push('!');
    println!("{}", s);
}
```

当我们将向量直接传给循环（如 `for ... in v`）时，会将向量从 `v` 中移动出去，让 `v` 变成未初始化状态。`for` 循环的内部机制会获取向量的所有权并将其分解为元素。在每次迭代中，循环都会将另一个元素转移给变量 `s`。由于 `s` 现在拥有字符串，因此可以在打印之前在循环体中修改它。在循环的过程中，向量本身对代码不再可见，因此也就无法观察到它正处在某种部分清空的状态。⁴

⁴因此不用担心中途修改向量本身之类的问题。——译者注

如果需要从拥有者中移出一个编译器无法跟踪的值，那么可以考虑将拥有者的类型更改为能动态跟踪自己是否有值的类型。例如，下面是前面例子的一个变体：

```
struct Person { name: Option<String>, birth: i32 }

let mut composers = Vec::new();
composers.push(Person { name: Some("Palestrina".to_string()),
                        birth: 1525 });
```

但不能像下面这样做：

```
let first_name = composers[0].name;
```

这只会引发与前面一样的“无法移动到索引结构之外”错误。但是因为已将 `name` 字段的类型从 `String` 改成了 `Option<String>`，

所以这意味着 `None` 也是该字段要保存的合法值。因此，可以像下面这样做：

```
let first_name = std::mem::replace(&mut composers[0].name, None);
assert_eq!(first_name, Some("Palestrina".to_string()));
assert_eq!(composers[0].name, None);
```

`replace` 调用会移出 `composers[0].name` 的值，将 `None` 留在原处，并将原始值的所有权转移给其调用者。事实上，这种使用 `Option` 的方式非常普遍，所以该类型专门为此提供了一个 `take` 方法，以便更清晰地写出上述操作，如下所示：

```
let first_name = composers[0].name.take();
```

这个 `take` 调用与之前的 `replace` 调用具有相同的效果。

4.3 Copy 类型：关于移动的例外情况

迄今为止，本章所展示的值移动示例都涉及向量、字符串和其他可能占用大量内存且复制成本高昂的类型。移动能让这些类型的所有权清晰且赋值开销极低。但对于像整数或字符这样的简单类型，如此谨小慎微的处理方式确实没什么必要。

下面来比较一下用 `String` 进行赋值和用 `i32` 进行赋值时内存中有什么不同：

```
let string1 = "somnambulance".to_string();
let string2 = string1;

let num1: i32 = 36;
let num2 = num1;
```

运行这段代码后，内存如图 4-11 所示。

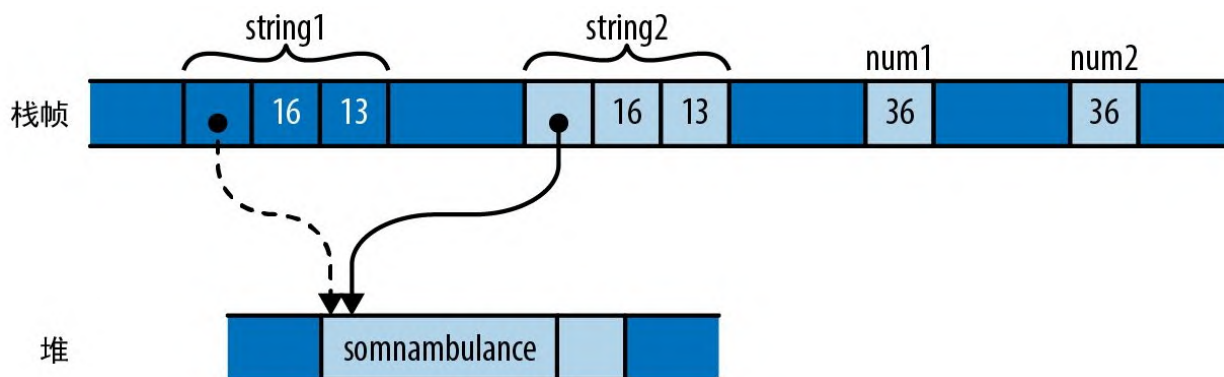


图 4-11：用 `String` 赋值会移动值，而用 `i32` 赋值会复制值

与前面的向量一样，赋值会将 `string1` 转移给 `string2`，这样就不会出现两个字符串负责释放同一个缓冲区的情况。但是，`num1` 和 `num2` 的情况有所不同。`i32` 只是内存中的几字节，它不拥有任何堆资源，也不会实际依赖除本身的字节之外的任何内存。当我们将它的每一位转移给 `num2` 时，其实已经为 `num1` 制作了一个完全独立的副本。

移动一个值会使移动的源变成未初始化状态。不过，尽管将 `string1` 视为未初始化变量确实符合其基本意图，但以这种方式对待 `num1` 毫无意义，继续使用 `num1` 也不会造成任何问题。移动在这里并无好处，反而会造成不便。

之前我们谨慎地说过，大多数类型会被移动，现在该谈谈例外情况了，即那些被 Rust 指定成 **Copy** 类型的类型。对 Copy 类型的值进行赋值会复制这个值，而不会移动它。赋值的源仍会保持已初始化和可用状态，并且具有与之前相同的值。把 Copy 类型传给函数和构造器的行为也是如此。

标准的 Copy 类型包括所有机器整数类型和浮点数类型、`char` 类型和 `bool` 类型，以及某些其他类型。Copy 类型的元组或固定大小的数组本身也是 Copy 类型。

只有那些可以通过简单地复制位来复制其值的类型才能作为 Copy 类型。前面解释过，`String` 不是 Copy 类型，因为它拥有从堆中分配的缓冲区。出于类似的原因，`Box<T>` 也不是 Copy 类型，因为它拥有从堆中分配的引用目标。代表操作系统文件句柄的 `File` 类型不是 Copy 类型，因为复制这样的值需要向操作系统申请另一个文件句柄。类似地，`MutexGuard` 类型表示一个互斥锁，它也不是 Copy 类型：复制这种类型毫无意义，因为每次只能有一个线程持有互斥锁。

根据经验，任何在丢弃值时需要做一些特殊操作的类型都不能是 Copy 类型：`Vec` 需要释放自身元素、`File` 需要关闭自身文件句柄、`MutexGuard` 需要解锁自身互斥锁，等等。对这些类型进行逐位复制会让我们无法弄清哪个值该对原始资源负责。

那么自定义类型呢？默认情况下，`struct` 类型和 `enum` 类型不是 Copy 类型：

```
struct Label { number: u32 }

fn print(l: Label) { println!("STAMP: {}", l.number); }
let l = Label { number: 3 };
```

```
print(l);
println!("My label number is: {}", l.number);
```

这无法编译。Rust 会报错：

```
error: borrow of moved value: `l`
  |
10 |     let l = Label { number: 3 };
   |         - move occurs because `l` has type `main::Label`,
   |         which does not implement the `Copy` trait
11 |     print(l);
   |         - value moved here
12 |     println!("My label number is: {}", l.number);
   |                                         ^^^^^^^^^
   |                                         value borrowed here after move
```

由于 Label 不是 Copy 类型，因此将它传给 print 会将值的所有权转移给 print 函数，然后在返回之前将其丢弃。这样做显然是愚蠢的，Label 中只有一个 u32，因此没有理由在将 l 传给 print 时移动这个值。

但是用户定义的类型不是 Copy 类型这一点只是默认情况而已。如果此结构体的所有字段本身都是 Copy 类型，那么也可以通过将属性 `#[derive(Copy, Clone)]` 放置在此定义之上来创建 Copy 类型，如下所示：

```
#[derive(Copy, Clone)]
struct Label { number: u32 }
```

经过此项更改，前面的代码可以顺利编译了。但是，如果试图在一个其字段不全是 Copy 类型的结构体上这样做，则仍然行不通。假设要编译如下代码：

```
#[derive(Copy, Clone)]
struct StringLabel { name: String }
```

那么就会引发如下错误：

```
error: the trait `Copy` may not be implemented for this type
|
7 | #[derive(Copy, Clone)]
|           ^^^^
8 | struct StringLabel { name: String }
|                               ----- this field does not implement
`Copy`
```

为什么符合条件的用户定义类型不能自动成为 `Copy` 类型呢？这是因为类型是否为 `Copy` 对于在代码中使用它的方式有着重大影响：`Copy` 类型更灵活，因为赋值和相关操作不会把原始值变成未初始化状态。但对类型的实现者而言，情况恰恰相反：`Copy` 类型可以包含的类型非常有限，而非 `Copy` 类型可以在堆上分配内存并拥有其他种类的资源。因此，创建一个 `Copy` 类型代表着实现者的郑重承诺：如果以后确有必要将其改为非 `Copy` 类型，则使用它的大部分代码可能需要进行调整。

虽然 C++ 允许重载赋值运算符以及定义专门的复制构造函数和移动构造函数，但 Rust 并不允许这种自定义行为。在 Rust 中，每次移动都是字节级的一对一浅拷贝，并让源变成未初始化状态。复制也是如此，但会保留源的初始化状态。这确实意味着 C++ 类可以提供 Rust 类型所无法提供的便捷接口，比如可以在看似普通的代码中隐式调整引用计数、把昂贵的复制操作留待以后进行，或使用另一些复杂的实现技巧。

但这种灵活性的代价是，作为一门语言，C++ 的基本操作（比如赋值、传参和从函数返回值）变得更难预测。例如，本章的前半部分展示过在 C++ 中将一个变量赋值给另一个变量时可能需要任意数量的内存和处理器时间。Rust 的一个原则是：各种开销对程序员来说应该是显而易见的。基本操作必须保持简单，而潜在的昂贵操作应该是显式的，比如前面例子中对 `clone` 的调用就是在对向量及其包含的字符串进行深拷贝。

本节用复制（`Copy`）和克隆（`Clone`）这两个模糊的术语描述了某个类型可能具备的特征。它们实际上是特型的示例。特型是 Rust 语言中的开放式工具，用于根据你对类型可以执行的操作来对类型进行分

类。第 11 章会对特型做一般性讲解，第 13 章会专门讲解 Copy 和 Clone 这两个特型。

4.4 Rc 与 Arc: 共享所有权

尽管在典型的 Rust 代码中大多数值会有唯一的拥有者，但在某些情况下，很难为每个值都找到具有所需生命周期的单个拥有者，你会希望某个值只要存续到每个人都用完它就好。对于这些情况，Rust 提供了引用计数指针类型 Rc 和 Arc [Arc 是原子引用计数(atomic reference count) 的缩写]。正如你对 Rust 的期待一样，这些类型用起来完全安全：你不会忘记调整引用计数，不会创建 Rust 无法注意到的指向引用目标的其他指针，也不会偶遇那些常与 C++ 中的引用计数指针如影随形的各种问题。

Rc 类型和 Arc 类型非常相似，它们之间唯一的区别是 Arc 可以安全地在线程之间直接共享，而普通 Rc 会使用更快的非线性安全代码来更新其引用计数。如果不需要在线程之间共享指针，则没有理由为 Arc 的性能损失“埋单”，因此应该使用 Rc，Rust 能防止你无意间跨越线程边界传递它。这两种类型在其他方面都是等效的，所以本节的其余部分只会讨论 Rc。

之前我们展示过 Python 如何使用引用计数来管理值的生命周期。你可以使用 Rc 在 Rust 中获得类似的效果。考虑以下代码：

```
use std::rc::Rc;

// Rust能推断出所有这些类型，这里写出它们只是为了讲解时清晰
let s: Rc<String> = Rc::new("shirataki".to_string());
let t: Rc<String> = s.clone();
let u: Rc<String> = s.clone();
```

对于任意类型 T，Rc<T> 值是指向附带引用计数的在堆上分配的 T 型指针。克隆一个 Rc<T> 值并不会复制 T，相反，它只会创建另一个指向它的指针并递增引用计数。所以前面的代码在内存中会生成图 4-12 所示的结果。

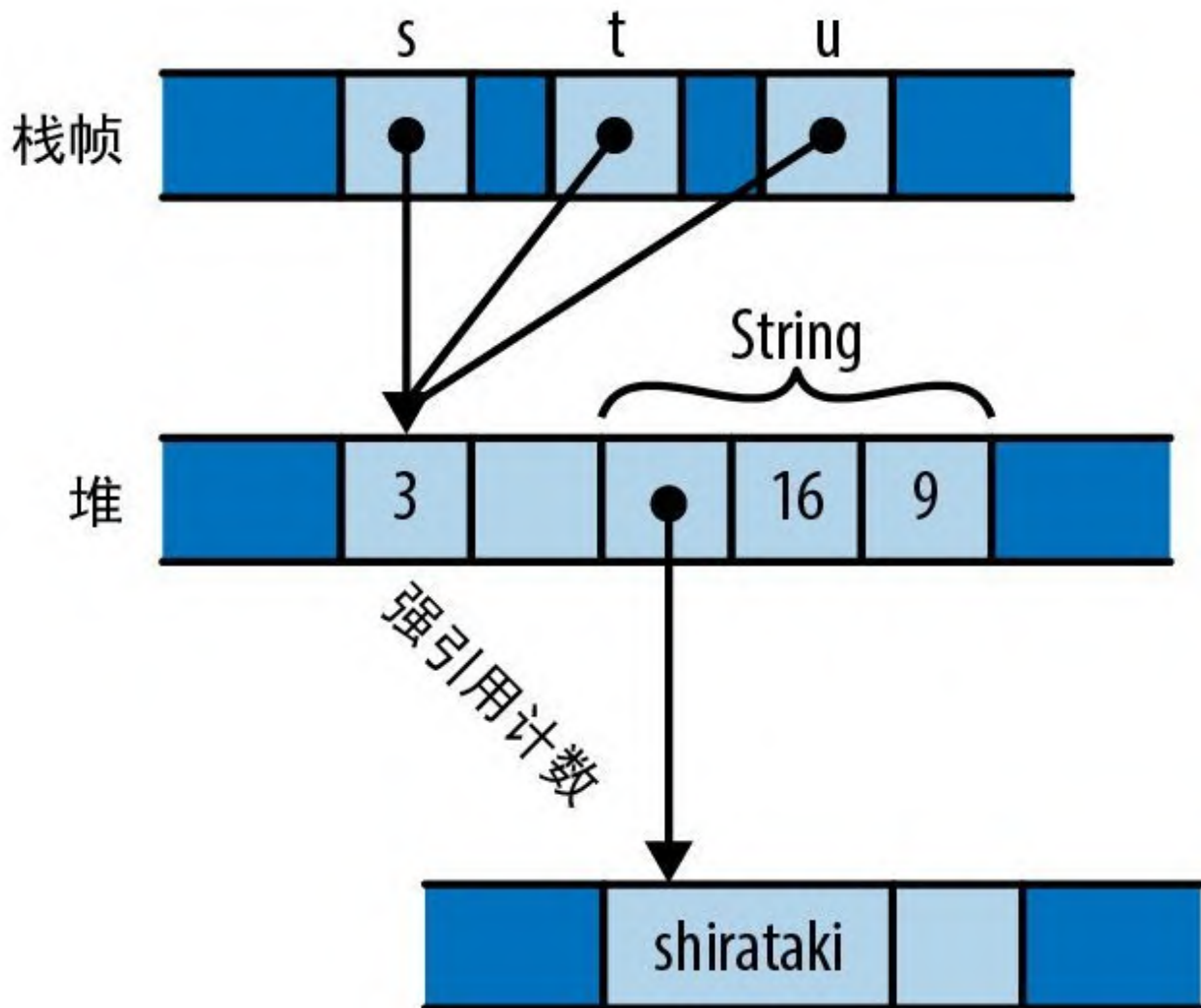


图 4-12: 具有 3 个引用的引用计数字符串

这 3 个 `Rc<String>` 指针指向了同一个内存块，其中包含引用计数和 `String` 本身的空间。通常的所有权规则适用于 `Rc` 指针本身，当丢弃最后一个现有 `Rc` 时，Rust 也会丢弃 `String`。

可以直接在 `Rc<String>` 上使用 `String` 的任何常用方法：

```
assert!(s.contains("shira"));
assert_eq!(t.find("taki"), Some(5));
println!("{}", u);
```

`Rc` 指针拥有的值是不可变的。如果你试图将一些文本添加到字符串的末尾：

```
s.push_str(" noodles");
```

那么 Rust 会拒绝：

```
error: cannot borrow data in an `Rc` as mutable
  |
13 |     s.push_str(" noodles");
  |     ^ cannot borrow as mutable
  |
```

Rust 的内存和线程安全保证的基石是：确保不会有任何值是既共享又可变的。Rust 假定 Rc 指针的引用目标通常都可以共享，因此就不能是可变的。第 5 章会解释为什么这个限制很重要。

使用引用计数管理内存的一个众所周知的问题是，如果有两个引用计数的值是相互指向的，那么其中一个值就会让另一个值的引用计数保持在 0 以上，因此这些值将永远没机会释放，如图 4-13 所示。

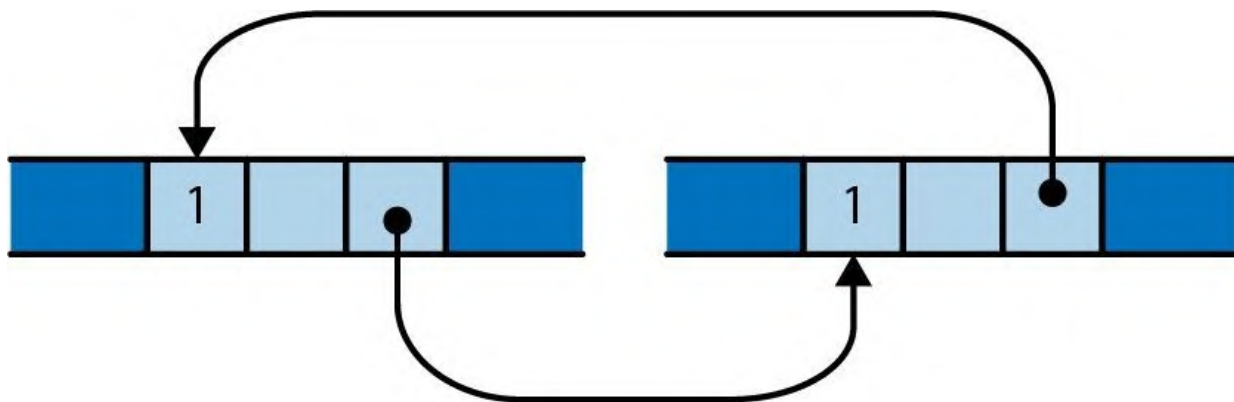


图 4-13：循环引用计数——这些对象都没机会释放

以这种方式在 Rust 中造成值的泄漏也是有可能的，但这种情况非常少见。只要不在某个时刻让旧值指向新值，就无法建立循环。这显然要求旧值是可变的。由于 Rc 指针会保证其引用目标不可变，因此通常不可能建立这种循环引用。但是，Rust 确实提供了创建其他不可变值中的可变部分的方法，这称为内部可变性，9.11 节会详细介绍。如果将这些技术与 Rc 指针结合使用，则确实可以建立循环并造成内存泄漏。

有时可以通过对某些链接使用**弱引用指针** `std::rc::Weak` 来避免建立 Rc 指针循环。但是，本节不会介绍这些内容，有关详细信息，请参阅标准库的文档。

移动和引用计数指针是缓解所有权树严格性问题的两种途径。在第 5 章中，我们将研究第三种途径：借用对值的引用。一旦你熟悉了所有权和借用这两个概念，就已经翻越了 Rust 学习曲线中最陡峭的部分，并为发挥 Rust 的独特优势做好了准备。

第 5 章 引用

图书馆（库）¹无法弥补个人（程序员）能力的不足。

——Mark Miller

¹ “库”与“图书馆”在英文中是同一个词，这里一语双关。——译者注

迄今为止，我们看到的所有指针类型（无论是简单的 `Box<T>` 堆指针，还是 `String` 值和 `Vec` 值内部的指针）都是拥有型指针，这意味着当拥有者被丢弃时，它的引用目标也会随之消失。Rust 还有一种名为引用（reference）的非拥有型指针，这种指针对于引用目标的生命周期毫无影响。

事实上，影响是反过来的：引用的生命周期绝不能超出其引用目标。你的代码必须遵循这样的规则，即任何引用的生命周期都不可能超出它指向的值。为了强调这一点，Rust 把创建对某个值的引用的操作称为借用（borrow）那个值：凡是借用，终须归还。

如果在读到“你的代码必须遵循这样的规则”这句话的时候不知道如何是好，那么你并非特例。引用本身确实没什么特别之处——说到底，它们只是地址而已。但用以让引用保持安全的规则，对 Rust 来说是一种创新，除了一些研究性语言，你不可能在其他编程语言中见到类似的规则。尽管这些规则是 Rust 中掌握起来最费心力的部分，但它们在防止经典的、常见的错误方面的覆盖度令人叹为观止，它们对多线程编程的影响也是革命性的。这又是 Rust 的“激进赌注”。

本章将介绍引用在 Rust 中的工作方式，我们会展开讲解引用、函数和自定义类型是如何通过包含生命周期信息来确保它们被安全使用的，并阐明这些努力为何能在编译期就避免一些常见类别的缺陷，而不必在运行期付出性能方面的代价。

5.1 对值的引用

假设我们要创建一张表格，列出文艺复兴时期某一特定类型的艺术家和他们的作品。Rust 的标准库包含一个哈希表类型，所以可以像下面这样定义我们的类型：

```
use std::collections::HashMap;

type Table = HashMap<String, Vec<String>>;
```

换句话说，这是一个将 String 值映射到 Vec<String> 值的哈希表，用于将艺术家的名字对应到他们作品名称的列表中。由于可以使用 for 循环遍历 HashMap 的条目，因此需要编写一个函数来打印 Table 的内容：

```
fn show(table: Table) {
    for (artist, works) in table {
        println!("works by {}: ", artist);
        for work in works {
            println!("  {}", work);
        }
    }
}
```

构建和打印这个表格的代码也一目了然：

```
fn main() {
    let mut table = Table::new();
    table.insert("Gesualdo".to_string(),
                vec!["many madrigals".to_string(),
                    "Tenebrae Responsoria".to_string()]);
    table.insert("Caravaggio".to_string(),
                vec!["The Musicians".to_string(),
                    "The Calling of St. Matthew".to_string()]);
    table.insert("Cellini".to_string(),
                vec!["Perseus with the head of
Medusa".to_string(),
                    "a salt cellar".to_string()]);
}
```

```
    show(table);  
}
```

一切正常：

```
$ cargo run  
    Running  
`/home/jimb/rust/book/fragments/target/debug/fragments`  
works by Gesualdo:  
    many madrigals  
    Tenebrae Responsoria  
works by Cellini:  
    Perseus with the head of Medusa  
    a salt cellar  
works by Caravaggio:  
    The Musicians  
    The Calling of St. Matthew  
$
```

但是，如果你已经阅读过第 4 章关于“移动”的部分，就会对 `show` 这个函数的定义产生一些疑问。特别是，`HashMap` 不是 `Copy` 类型——也不可能是，因为它拥有能动态分配大小的表格。所以当程序调用 `show(table)` 时，整个结构就移动到了函数中，而变量 `table` 变成了未初始化状态。（而且它还会以乱序迭代其内容，所以如果你看到的顺序与这里不同，请不要担心，这是正常现象。）现在如果调用方代码试图使用 `table`，则会遇到麻烦：

```
...  
show(table);  
assert_eq!(table["Gesualdo"][0], "many madrigals");
```

Rust 会报错说 `table` 不再可用：

```
error: borrow of moved value: `table`  
  |  
20 |     let mut table = Table::new();  
   |         ----- move occurs because `table` has type  
   |         `HashMap<String, Vec<String>>`,  
   |         which does not implement the `Copy` trait  
...  
31 |     show(table);
```

```
|          ----- value moved here
32 |      assert_eq!(table["Gesualdo"][0], "many madrigals");
|          ^^^^^ value borrowed here after move
```

事实上，如果查看 `show` 的定义，你会发现外层的 `for` 循环获取了哈希表的所有权并完全消耗掉了，并且内部的 `for` 循环对每个向量执行了相同的操作。（之前我们在 4.2.3 节的示例中看到过这种行为。）由于移动的语义特点，我们只是想把它打印出来，却完全破坏了整个结构。Rust，你可“真行”！

处理这个问题的正确方式是使用引用。引用能让你在不影响其所有权的情况下访问值。引用分为以下两种。

- **共享引用**允许你读取但不能修改其引用目标。但是，你可以根据需要同时拥有任意数量的对特定值的共享引用。表达式 `&e` 会产生对 `e` 值的共享引用，如果 `e` 的类型为 `T`，那么 `&e` 的类型就是 `&T`，读作“`ref T`”。共享引用是 `Copy` 类型。
- **可变引用**允许你读取和修改值。但是，一旦一个值拥有了可变引用，就无法再对该值创建其他任何种类的引用了。表达式 `&mut e` 会产生一个对 `e` 值的可变引用，可以将其类型写成 `&mut T`，读作“`ref mut T`”。可变引用不是 `Copy` 类型。

可以将共享引用和可变引用之间的区别视为在编译期强制执行“多重读取”或“单一写入”规则的一种手段。事实上，这条规则不仅适用于引用，也适用于所引用值的拥有者。只要存在对一个值的共享引用，即使是它的拥有者也不能修改它，该值会被锁定。当 `show` 正在使用 `table` 时，没有人可以修改它。类似地，如果有某个值的可变引用，那么它就会独占对该值的访问权，在可变引用消失之前，即使拥有者也根本无法使用该值。事实证明，让共享和修改保持完全分离对于内存安全至关重要，本章会在稍后内容中讨论原因。

我们示例中的打印函数不需要修改表格，只需读取其内容即可。所以调用者可以向它传递一个对表的共享引用，如下所示：

```
show(&table);
```

引用是非拥有型指针，因此 `table` 变量仍然是整个结构的拥有者，`show` 刚刚只是借用了一会儿。当然，我们需要调整 `show` 的定义来匹配它，但必须仔细观察才能看出差异：

```
fn show(table: &Table) {
    for (artist, works) in table {
        println!("works by {}: ", artist);
        for work in works {
            println!("  {}", work);
        }
    }
}
```

`show` 中的 `table` 参数的类型已从 `Table` 变成了 `&Table`：现在不再按值传入 `table`（那样会将所有权转移到函数中），而是传入了共享引用。这是代码上的唯一变化。但是当我们深入函数体了解其工作原理时，这会有怎样的影响呢？

在以前的版本中，外部 `for` 循环获取了此 `HashMap` 的所有权并消耗掉了它，但在新版本中，它收到了对 `HashMap` 的共享引用。迭代中对 `HashMap` 的共享引用就是对每个条目的键和值的共享引用：`artist` 从 `String` 变成了 `&String`，而 `works` 从 `Vec<String>` 变成了 `&Vec<String>`。

内层循环也有类似的改变。迭代中对向量的共享引用就是对其元素的共享引用，因此 `work` 现在是 `&String`。此函数的任何地方都没有发生过所有权转移，它只会传递非拥有型引用。

现在，如果想写一个函数来按字母顺序排列每位艺术家的作品，那么只通过共享引用是不够的，因为共享引用不允许修改。而这个排序函数需要对表进行可变引用：

```
fn sort_works(table: &mut Table) {
    for (_artist, works) in table {
        works.sort();
    }
}
```


于是我们需要传入一个：

```
sort_works(&mut table);
```

这种可变借用使 `sort_works` 能够按照向量的 `sort` 方法的要求读取和修改此结构。

当通过将值的所有权转移给函数的方式将这个值传给函数时，就可以说**按值传递**了它。如果改为将值的引用传给函数，就可以说**按引用传递**了它。例如，我们刚刚修复了 `show` 函数，将其改为按引用而不是按值接受 `table`。许多语言中也有这种区分，但在 Rust 中这尤为重要，因为它阐明了所有权是如何受到影响的。

5.2 使用引用

前面的示例展示了引用的一个非常典型的用途：允许函数在不获取所有权的情况下访问或操纵某个结构。但引用比这要灵活得多，下面我们通过一些示例来更详细地了解引用的用法。

5.2.1 Rust 引用与 C++ 引用

如果熟悉 C++ 中的引用，你就会知道它们确实与 Rust 引用有某些共同点。最重要的是，它们都只是机器级别的地址。但在实践中，Rust 的引用会给人截然不同的感觉。

在 C++ 中，引用是通过类型转换隐式创建的，并且是隐式解引用的：

```
// C++代码！
int x = 10;
int &r = x;           // 初始化时隐式创建引用
assert(r == 10);     // 对r隐式解引用，以查看x的值
r = 20;              // 把20存入x，r本身仍然指向x
```

在 Rust 中，引用是通过 & 运算符显式创建的，同时要用 * 运算符显式解引用：

```
// 从这里开始回到Rust代码
let x = 10;
let r = &x;           // &x是对x的共享引用
assert!(*r == 10);    // 对r显式解引用
```

要创建可变引用，可以使用 &mut 运算符：

```
let mut y = 32;
let m = &mut y;       // &mut y是对y的可变引用
*m += 32;             // 对m显式解引用，以设置y的值
assert!(*m == 64);    // 来看看y的新值
```

也许你还记得，当我们修复 show 函数以通过引用而非值来获取艺术家表格时，并未使用过 * 运算符。这是为什么呢？

由于引用在 Rust 中随处可见，因此 `.` 运算符就会按需对其左操作数隐式解引用：

```
struct Anime { name: &'static str, bechdel_pass: bool }
let aria = Anime { name: "Aria: The Animation", bechdel_pass: true };
let anime_ref = &aria;
assert_eq!(anime_ref.name, "Aria: The Animation");

// 与上一句等效，但把解引用过程显式地写了出来
assert_eq!((*anime_ref).name, "Aria: The Animation");
```

`show` 函数中使用的 `println!` 宏会展开成使用 `.` 运算符的代码，因此它也能利用这种隐式解引用的方式。

在进行方法调用时，`.` 运算符也可以根据需求隐式借用对其左操作数的引用。例如，`Vec` 的 `sort` 方法就要求参数是对向量的可变引用，因此这两个调用是等效的：

```
let mut v = vec![1973, 1968];
v.sort();           // 隐式借用对v的可变引用
(&mut v).sort();    // 等效，但是更烦琐
```

简而言之，C++ 会在引用和左值（引用内存中位置的表达式）之间隐式转换，并且这种转换会出现在任何需要转换的地方，而在 Rust 中要使用 `&` 运算符和 `*` 运算符来创建引用（借用）和追踪引用（解引用），不过 `.` 运算符不需要做这种转换，它会隐式借用和解引用。

5.2.2 对引用变量赋值

把引用赋值给某个引用变量会让该变量指向新的地方：

```
let x = 10;
let y = 20;
let mut r = &x;

if b { r = &y; }

assert!(*r == 10 || *r == 20);
```

引用 `r` 最初指向 `x`。但如果 `b` 为 `true`，则代码会把它改为指向 `y`，如图 5-1 所示。

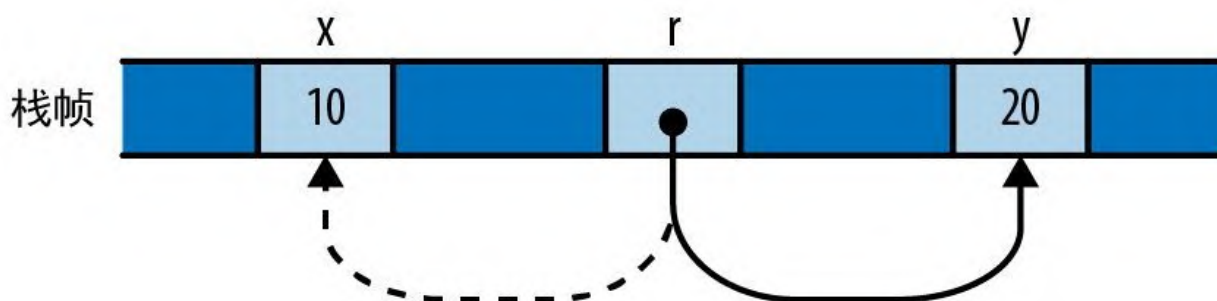


图 5-1：引用 `r` 现在指向 `y` 而不再是 `x`

乍一看，这种行为可能太显而易见，不值一提：现在 `r` 当然会指向 `y`，因为我们在其中存储了 `&y`。但特意指出这一点是因为 C++ 引用的行为与此截然不同：如前所述，在 C++ 中对引用赋值会将新值存储在其引用目标中而非指向新值。C++ 的引用一旦完成初始化，就无法再指向别处了。

5.2.3 对引用进行引用

Rust 允许对引用进行引用：

```
struct Point { x: i32, y: i32 }
let point = Point { x: 1000, y: 729 };
let r: &Point = &point;
let rr: &&Point = &r;
let rrr: &&&Point = &rr;
```

（为了让讲解更清晰，我们标出了引用的类型，但你可以省略它们，这里没有什么是 Rust 无法自行推断的。）运算符会追踪尽可能多层次的引用来找到它的目标：

```
assert_eq!(rrr.y, 729);
```

在内存中，引用的排列方式如图 5-2 所示。

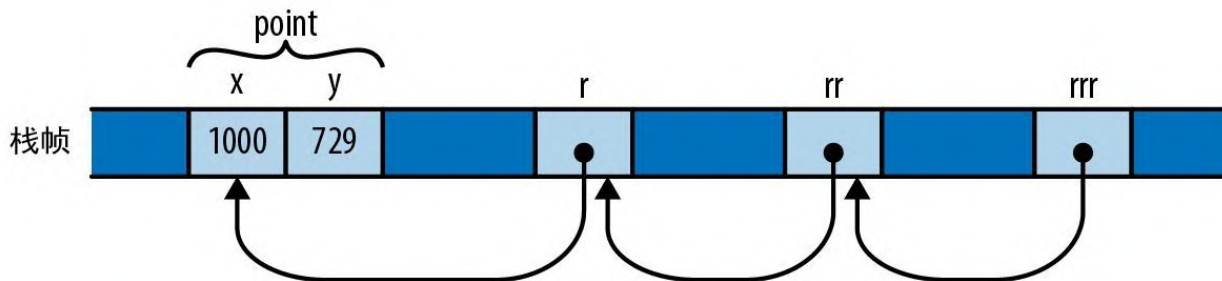


图 5-2：对引用的引用链

在这里，表达式 `rrr.y` 根据 `rrr` 的类型的指引遍历了 3 层引用才取到相应 `Point` 的 `y` 字段。

5.2.4 比较引用

就像 `.` 运算符一样，Rust 的比较运算符也能“看穿”任意数量的引用：

```
let x = 10;
let y = 10;

let rx = &x;
let ry = &y;

let rrx = &rx;
let rry = &ry;

assert!(rrx <= rry);
assert!(rrx == rry);
```

虽然 `rrx` 和 `rry` 指向的是不同的值（`rx` 和 `ry`），这里的断言最终仍然会成功，因为 `==` 运算符会追踪所有引用并对它们的最终目标 `x` 和 `y` 进行比较。这几乎总是你期望的行为，尤其是在编写泛型函数时。如果你真想知道两个引用是否指向同一块内存，可以使用 `std::ptr::eq`，它会将两者作为地址进行比较：

```
assert!(rx == ry);           // 它们引用的目标值相等
assert!(!std::ptr::eq(rx, ry)); // 但所占据的地址（自身的值）不同
```

但要注意，比较运算符的操作数（包括引用型操作数）必须具有完全相同的类型。

```
assert!(rx == rrx);    // 错误：`&i32`与`&&i32`的类型不匹配
assert!(rx == *rrx);   // 这样没问题
```

5.2.5 引用永不为空

Rust 的引用永远都不会为空。与 C 的 NULL 或 C++ 的 nullptr 类似的东西是不存在的。引用没有默认初始值（在初始化之前不能使用任何变量，无论其类型如何），并且 Rust 不会将整数转换为引用（在 unsafe 代码外）。因此，不能将 0 转换成引用。

C 代码和 C++ 代码通常会使用空指针来指示值的缺失：当可用内存充足时，malloc 函数会返回指向新内存块的指针，否则会返回 nullptr。在 Rust 中，如果需要用一个值来表示对某个“可能不存在”事物的引用，请使用类型 Option<&T>。在机器码级别，Rust 会将 None 表示为空指针，将 Some(r) 表示为非零地址（其中 r 是 &T 型的值），因此 Option<&T> 与 C 或 C++ 中的可空指针一样高效，但更安全：它的类型要求你在使用之前必须检查它是否为 None。

5.2.6 借用任意表达式结果值的引用

C 和 C++ 只允许将 & 运算符应用于某些特定种类的表达式，而 Rust 允许借用任意种类的表达式结果值的引用：

```
fn factorial(n: usize) -> usize {
    (1..n+1).product()
}
let r = &factorial(6);
// 数学运算符可以“看穿”一层引用
assert_eq!(r + &1009, 1729);
```

在这种情况下，Rust 会创建一个匿名变量来保存此表达式的值，并让该引用指向它。这个匿名变量的生命周期取决于你对引用做了什么。

- 在 `let` 语句中，如果立即将引用赋值给某个变量（或者使其成为立即被赋值的某个结构体或数组的一部分），那么 Rust 就会让匿名变量存在于 `let` 初始化此变量期间。在前面的示例中，Rust 就会对 `r` 的引用目标这样做。
- 否则，匿名变量会一直存续到所属封闭语句块的末尾。在我们的示例中，为保存 1009 而创建的匿名变量只会存续到 `assert_eq!` 语句的末尾。

如果你习惯于使用 C 或 C++，那么这可能听起来很容易出错。但别忘了，Rust 永远不会让你写出可能生成悬空引用的代码。只要引用可能在匿名变量的生命周期之外被使用，Rust 就一定会在编译期间报告问题，然后你就可以通过将引用保存在具有适当生命周期的命名变量中来修复代码。

5.2.7 对切片和特型对象的引用

迄今为止，我们展示的引用全都是简单地址。但是，Rust 还包括两种胖指针，即携带某个值地址的双字值，以及要正确使用该值所需的某些额外信息。

对切片的引用就是一个胖指针，携带着此切片的起始地址及其长度。第 3 章详细讲解过切片。

Rust 的另一种胖指针是**特型对象**，即对实现了指定特型的值的引用。特型对象会携带一个值的地址和指向适用于该值的特型实现的指针，以便调用特型的方法。11.1.1 节会详细介绍特型对象。

除了会携带这些额外数据，切片和特型对象引用的行为与本章中已展示过的其他引用是一样的：它们并不拥有自己的引用目标、它们的生命周期也不允许超出它们的引用目标、它们可能是可变的或共享的，等等。

5.3 引用安全

正如前面介绍过的那样，引用看起来很像 C 或 C++ 中的普通指针。但普通指针是不安全的，Rust 又如何保持对引用的全面控制呢？或许了解规则的最佳方式就是尝试打破规则。

为了传达基本思想，我们将从最简单的案例开始，展示 Rust 如何确保在单个函数体内正确使用引用。然后我们会看看如何在函数之间传递引用并将它们存储到数据结构中。这需要为函数和数据类型提供生命周期参数（稍后会对其进行解释）。最后我们会介绍 Rust 提供的一些简写形式，以简化常见的使用模式。在整个过程中，我们将展示 Rust 如何找出损坏的代码，并不时提出解决方案。

5.3.1 借用局部变量

这是一个非常浅显的案例。你不能借用对局部变量的引用并将其移出变量的作用域：

```
{
    let r;
    {
        let x = 1;
        r = &x;
    }
    assert_eq!(*r, 1); // 错误：试图读取`x`所占用的内存
}
```

Rust 编译器会拒绝此程序，并显示详细的错误消息：

```
error: `x` does not live long enough
  |
7 |         r = &x;
  |             ^^ borrowed value does not live long enough
8 |     }
  |     - `x` dropped here while still borrowed
9 |     assert_eq!(*r, 1); // 错误：试图读取`x`所占用的内存
10 | }
```


Rust 报错说 `x` 只能存续至内部块的末尾，而引用 (`r`) 会一直存续至外部块的末尾，这就让它成了悬空指针，这是被禁止的。

虽然对人类读者来说这个程序很明显是错误的，但还是值得研究一下 Rust 本身如何得出的这个结论。即使是这么简单的例子，也能展示出 Rust 用来检查更复杂代码的一些逻辑工具。

Rust 会尝试为程序中的每个引用类型分配一个生命周期，以表达根据其使用方式应施加的约束。生命周期是程序的一部分，可以确保引用在下列位置都能被安全地使用：语句中、表达式中、某个变量的作用域中等。生命周期完全是 Rust 在编译期虚构的产物。在运行期，引用只是一个地址，它的使用寿命只是其类型的一部分，不存在运行期表示。

在这个例子中，我们要分析 3 个生命周期之间的关系。变量 `r` 和 `x` 都有各自的生命周期，从它们被初始化的时间点一直延续到足以让编译器断定不再使用它们的时间点。第三个生命周期是引用类型，即借用了 `x` 并存储在 `r` 中的引用类型。

这里有一个显而易见的约束：如果有一个变量 `x`，那么对 `x` 的引用的生命周期不能超出 `x` 本身，如图 5-3 所示。

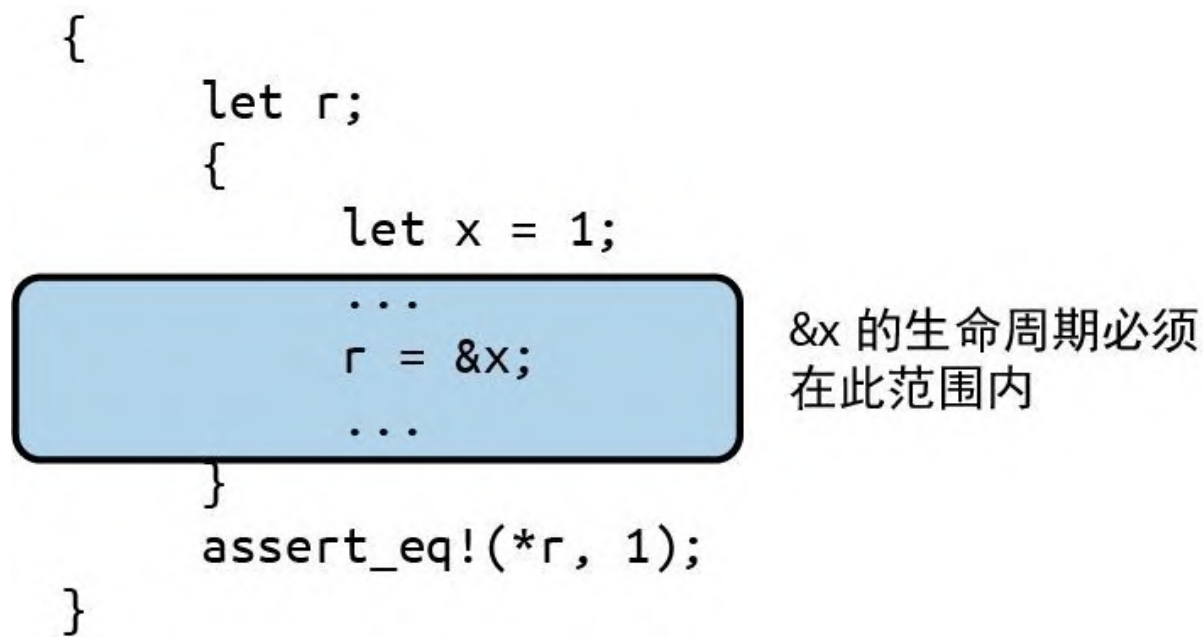


图 5-3: `&x` 的容许生命周期

当 `x` 超出作用域时，其引用将是一个悬空指针。为此，我们说变量的生命周期必须涵盖借用过它的引用的生命周期。

这是另一个约束：如果将引用存储在变量 `r` 中，则引用类型必须在变量 `r` 从初始化到最后一次使用的整个生命周期内都可以访问，如图 5-4 所示。

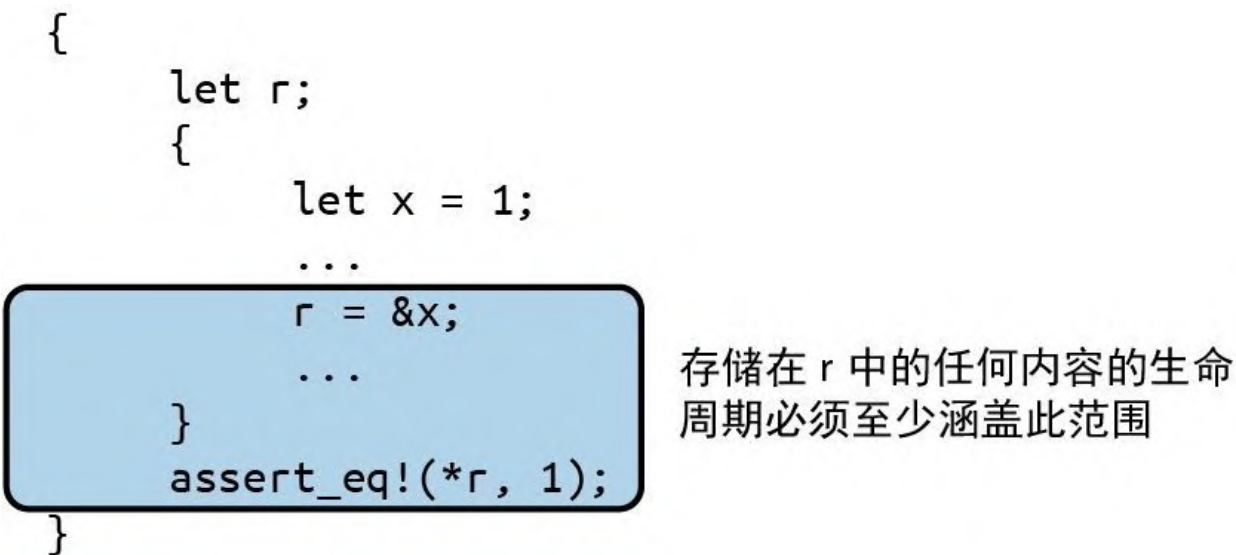


图 5-4: 存储在 `r` 中的引用的容许生命周期

如果引用的生命周期不能至少和变量 `r` 一样长，那么在某些时候变量 `r` 就会变成悬空指针。为此，我们说引用的生命周期必须涵盖变量 `r` 的生命周期。

第一个约束限制了引用的生命周期可以有多大，而第二个约束则限制了它可以有多小。Rust 会尝试找出能让每个引用都满足这两个约束的生命周期。然而我们的示例中并不存在这样的生命周期，如图 5-5 所示。

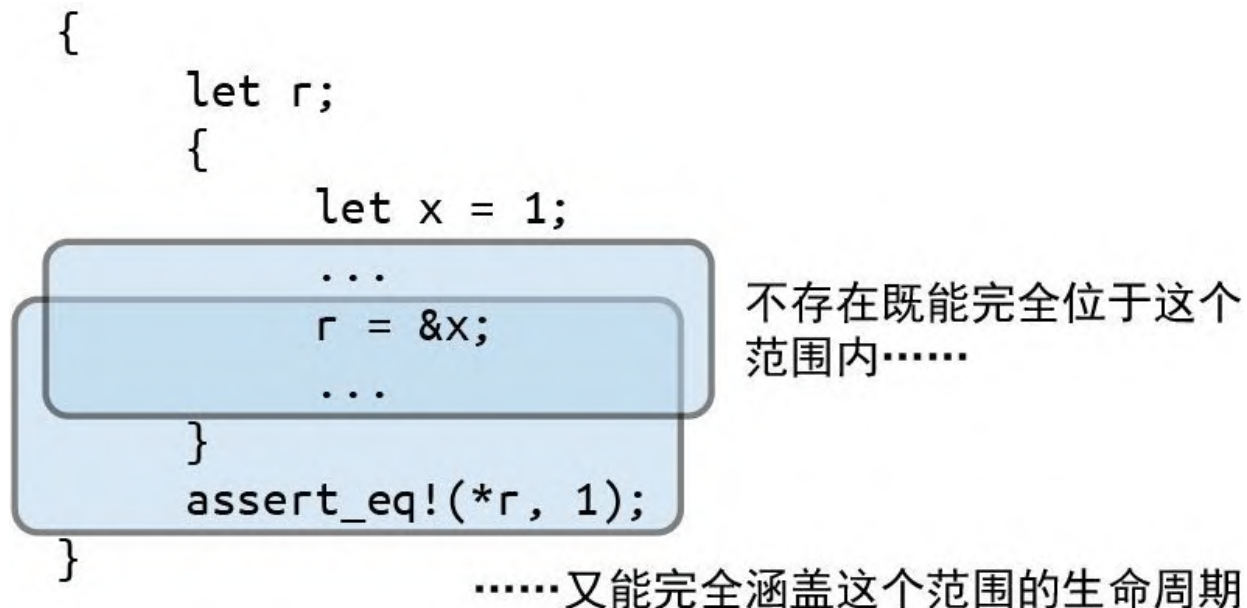


图 5-5：引用的生命周期，其各项约束存在矛盾

现在来看一个不一样的例子，这次就行得通了。还是同样的约束：引用的生命周期必须包含在 `x` 中，但也要完全涵盖 `r` 的生命周期。因为现在 `r` 的生命周期变小了，所以会有一个生命周期满足这些约束，如图 5-6 所示。

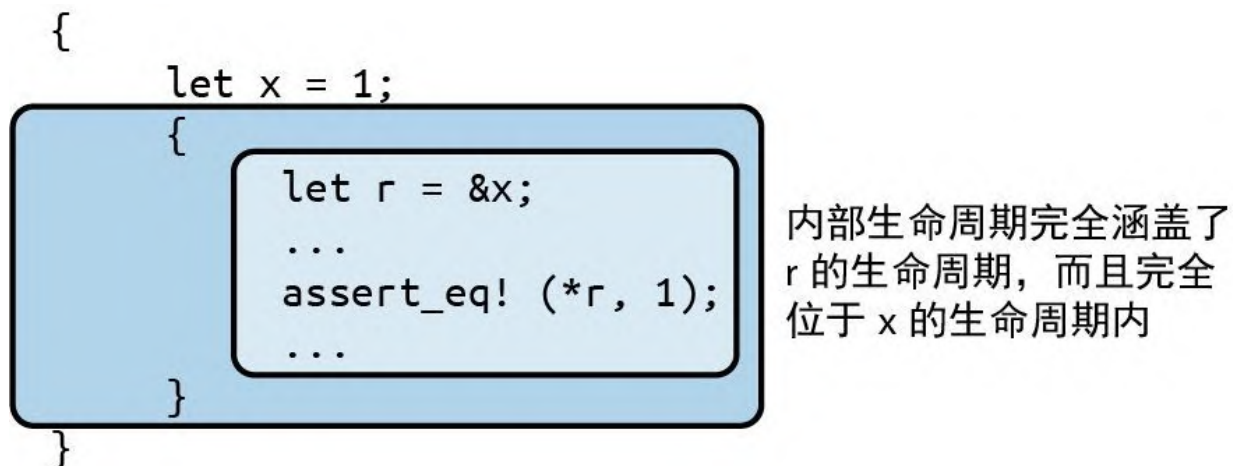


图 5-6：引用的生命周期涵盖了 `r` 的生命周期且同时位于 `x` 的作用域内

当你借用大型数据结构中某些部分（比如向量的元素）的引用时，会自然而然地应用这些规则：

```
let v = vec![1, 2, 3];  
let r = &v[1];
```

由于 `v` 拥有一个向量，此向量又拥有自己的元素，因此 `v` 的生命周期必须涵盖 `&v[1]` 引用类型的生命周期。类似地，如果将一个引用存储于某个数据结构中，则此引用的生命周期也必须涵盖那个数据结构的生命周期。如果构建一个由引用组成的向量，则所有这些引用的生命周期都必须涵盖拥有该向量的变量的生命周期。

这是 Rust 用来处理所有代码的过程的本质。引入更多的语言特性（比如数据结构和函数调用），必然会引入一些全新种类的约束，但基本原则保持不变：首先，要了解程序中使用各种引用的方式带来的约束；其次，找出能同时满足这些约束的生命周期。这与 C 和 C++ 程序员不得不人工担负的过程没有多大区别，唯一的区别是 Rust 知道这些规则并会强制执行。

5.3.2 将引用作为函数参数

当我们传递对函数的引用时，Rust 要如何确保函数能安全地使用它呢？假设我们有一个函数 `f`，它会接受一个引用并将其存储在全局变量中。我们需要对此进行一些修改，下面是第一个版本：

```
// 这段代码有一系列问题，无法编译  
static mut STASH: &i32;  
fn f(p: &i32) { STASH = p; }
```

在 Rust 中，全局变量的等价物称为**静态变量**（`static`）：它是在程序启动时就会被创建并一直存续到程序终止时的值。（与任何其他声明一样，Rust 的模块系统会控制静态变量在何处可见，因此这只代表它们的生命周期是“全局”的，并不表示它们全局可见。）第 8 章会介绍静态变量，现在我们只讲一下刚才展示的代码违反了哪些规则。

- 每个静态变量都必须初始化。

- 可变静态变量本质上不是线程安全的（毕竟，任何线程都可以随时访问静态变量），即使在单线程的程序中，它们也可能成为一些另类可重入性问题的牺牲品。由于这些原因，你只能在 `unsafe` 块中访问可变静态变量。在这个例子中，我们并不关心那几个具体问题，所以只是把可变静态变量扔到了一个 `unsafe` 块中。

经过这些修改，现在我们有了以下内容：

```
static mut STASH: &i32 = &128;
fn f(p: &i32) { // 仍然不够理想
    unsafe {
        STASH = p;
    }
}
```

就快完工了。要查看剩下的问题，就要把 Rust 替我们省略的一些代码写出来。此处所写的 `f` 的签名实际上是以下内容的简写：

```
fn f<'a>(p: &'a i32) { ... }
```

这里，生命周期 `'a`（读作“tick A”）是 `f` 的生命周期参数。`<'a>` 的意思是“对于任意生命周期 `'a`”，因此当我们编写 `fn f<'a>(p: &'a i32)` 时，就定义了一个函数，该函数能接受对具有任意生命周期 `'a` 的 `i32` 型引用。

因为必须允许 `'a` 是任意生命周期，所以如果它是可能的最小生命周期（一个恰好涵盖对 `f` 调用的生命周期），那么问题就能轻易解决。接下来赋值语句就成了争论的焦点：

```
STASH = p;
```

由于 `STASH` 会存续在程序的整个执行过程中，因此它所持有的引用类型必须具有等长的生命周期，Rust 将此称为 **`'static`** 生命周期。但是指向 `p` 的引用的生命周期是 `'a`，它可以是任何能涵盖对 `f` 调用的生命周期。所以，Rust 拒绝了我们的代码：

```
error: explicit lifetime required in the type of `p`
  |
5 |         STASH = p;
  |                   ^ lifetime `'static` required
```

在这个时间点，很明显我们的函数不能接受任意引用作为参数。但正如 Rust 指出的那样，它应当接受具有 `'static` 生命周期的引用：在 `STASH` 中存储这样的引用不会创建悬空指针。事实上，下面的代码就编译得很好：

```
static mut STASH: &i32 = &10;

fn f(p: &'static i32) {
    unsafe {
        STASH = p;
    }
}
```

这一次，`f` 的签名指出 `p` 必须是生命周期为 `'static` 的引用，因此将其存储在 `STASH` 中不会再有任何问题。我们只能用对其他静态变量的引用来调用 `f`，但这是唯一一种肯定不会让 `STASH` 悬空的方式。所以可以像下面这样写：

```
static WORTH_POINTING_AT: i32 = 1000;
f(&WORTH_POINTING_AT);
```

由于 `WORTH_POINTING_AT` 是静态变量，因此 `&WORTH_POINTING_AT` 的类型是 `&'static i32`，将该类型传给 `f` 是安全的。

不过，可以退后一步，来看看在修改成正确方法时，`f` 的签名发生了哪些变化：原来的 `f(p: &i32)` 最后变成了 `f(p: &'static i32)`。换句话说，我们无法编写在全局变量中潜藏一个引用却不在函数签名中明示该意图的函数。在 Rust 中，函数的签名总会揭示出函数体的行为。

相反，如果确实看到一个带有 `g(p: &i32)` 签名的函数（或者带着生命周期写成 `g<'a>(p: &'a i32)`），那么就可以肯定它没有将其

参数 `p` 藏在任何超出此调用点的地方。无须查看 `g` 的具体定义，签名本身就可以告诉我们 `g` 用它的参数能做什么，不能做什么。当你尝试为函数调用建立安全保障时，这一认知会非常有价值。

5.3.3 把引用传给函数

我们刚刚揭示了函数签名与其函数体的关系，下面再来看一下函数签名与其调用者的关系。假设你有以下代码：

```
// 这个函数可以简写为fn g(p: &i32)，但这里还是把它生命周期写出来了
fn g<'a>(p: &'a i32) { ... }

let x = 10;
g(&x);
```

只要看看 `g` 的签名，Rust 就知道它不会将 `p` 保存在生命周期可能超出本次调用的任何地方：包含本次调用的任何生命周期都必须符合 `'a` 的要求。所以 Rust 为 `&x` 选择了尽可能短的生命周期，即调用 `g` 时的生命周期。这满足所有约束：它的使用寿命不会超出 `x`，并且会涵盖对 `g` 的完整调用。所以这段代码通过了审核。

请注意，虽然 `g` 有一个生命周期参数 `'a`，但调用 `g` 时并不需要提及它。只要在定义函数和类型时关心生命周期参数就够了，使用它们时，Rust 会为你推断生命周期。

如果试着将 `&x` 传给之前要求其参数存储在静态变量中的函数 `f` 会怎样呢？

```
fn f(p: &'static i32) { ... }

let x = 10;
f(&x);
```

这无法编译：引用 `&x` 的生命周期不能超出 `x`，但通过将它传给 `f`，又限制了它必须至少和 `'static` 一样长。没办法做到两全其美，所以 Rust 只好拒绝了这段代码。

5.3.4 返回引用

函数通常会接收某个数据结构的引用，然后返回对该结构的某个部分的引用。例如，下面是一个函数，它会返回对切片中最小元素的引用：

```
// v应该至少有一个元素
fn smallest(v: &[i32]) -> &i32 {
    let mut s = &v[0];
    for r in &v[1..] {
        if *r < *s { s = r; }
    }
    s
}
```

我们依惯例在该函数的签名中省略了生命周期。当函数以单个引用作为参数并返回单个引用时，Rust 会假定两者具有相同的生命周期。如果把生命周期明确地写出来，则能看得更清楚：

```
fn smallest<'a>(v: &'a [i32]) -> &'a i32 { ... }
```

假设我们是这样调用 `smallest` 的：

```
let s;
{
    let parabola = [9, 4, 1, 0, 1, 4, 9];
    s = smallest(&parabola);
}
assert_eq!(*s, 0); // 错误：指向了已被丢弃的数组的元素
```

从 `smallest` 的签名可以看出它的参数和返回值必须具有相同的生命周期 `'a`。在我们的调用中，参数 `&parabola` 的生命周期不得超出 `parabola` 本身，但 `smallest` 的返回值的生命周期必须至少和 `s` 一样长。生命周期 `'a` 不可能同时满足这两个约束，因此 Rust 拒绝执行这段代码：

```
error: `parabola` does not live long enough
  |
11 |         s = smallest(&parabola);
  |                        ----- borrow occurs here
```



```

12 |     }
    |     ^ `parabola` dropped here while still borrowed
13 |     assert_eq!(*s, 0); // 错误: 指向了已被丢弃的数组的元素
    |                       - borrowed value needs to live until here
14 |     }

```

移动 `s`，让其生命周期完全包含在 `parabola` 内就可以解决问题：

```

{
    let parabola = [9, 4, 1, 0, 1, 4, 9];
    let s = smallest(&parabola);
    assert_eq!(*s, 0); // 很好: parabola 仍然“活着”
}

```

函数签名中的生命周期能让 Rust 评估你传给函数的引用与函数返回的引用之间的关系，并确保安全地使用它们。

5.3.5 包含引用的结构体

Rust 如何处理存储在数据结构中的引用呢？下面仍然是之前那个出错的程序，但这次我们将引用“藏”在了结构体中：

```

// 这无法编译
struct S {
    r: &i32
}

let s;
{
    let x = 10;
    s = S { r: &x };
}
assert_eq!(*s.r, 10); // 错误: 从已被丢弃的`x`中读取

```

Rust 对引用的安全约束不会因为我们将引用“藏”在结构体中而神奇地消失。无论如何，这些约束最终也必须应用在 `s` 上。的确，Rust 提出了质疑：

```

error: missing lifetime specifier
|

```

```
7 |           r: &i32  
  |           ^ expected lifetime parameter
```

每当一个引用类型出现在另一个类型的定义中时，必须写出它的生命周期。可以这样写：

```
struct S {  
    r: &'static i32  
}
```

这表示 `r` 只能引用贯穿程序整个生命周期的 `i32` 值，这种限制太严格了。还有一种方法是给类型指定一个生命周期参数 `'a` 并将其用在 `r` 上：

```
struct S<'a> {  
    r: &'a i32  
}
```

现在 `S` 类型有了一个生命周期，就像引用类型一样。你创建的每个 `S` 类型的值都会获得一个全新的生命周期 `'a`，它会受到该值的使用方式的限制。你存储在 `r` 中的任何引用的生命周期最好都涵盖 `'a`，并且 `'a` 必须比存储在 `S` 中的任何内容的生命周期都要长。

回到前面的代码，表达式 `S { r: &x }` 创建了一个新的 `S` 值，其生命周期为 `'a`。当你将 `&x` 存储在 `r` 字段中时，就将 `'a` 完全限制在了 `x` 的生命周期内部。

赋值语句 `s = S { ... }` 会将此 `S` 存储在一个变量中，该变量的生命周期会延续到示例的末尾，这种限制决定了 `'a` 比 `s` 的生命周期更长。现在 Rust 遇到了与之前一样矛盾的约束：`'a` 的生命周期不能超出 `x`，但至少和 `s` 一样长。因为找不到两全其美的生命周期，所以 Rust 拒绝执行该代码。一场灾难提前化解了。

如果将具有生命周期参数的类型放置在其他类型中会怎样呢？

```
struct D {  
    s: S // 不合格  
}
```

Rust 提出了质疑，就像试图在 `s` 中放置一个引用而未指定其生命周期一样：

```
error: missing lifetime specifier
  |
8 |         s: S // 不合格
  |           ^ expected named lifetime parameter
  |
```

不能在这里省略 `s` 的生命周期参数：Rust 需要知道 `D` 的生命周期和其引用的 `s` 的生命周期之间是什么关系，以便对 `D` 进行与“`s` 和普通引用”一样的检查。

可以给 `s` 一个 `'static` 生命周期。这样没问题：

```
struct D {
    s: S<'static>
}
```

使用这种定义，`s` 字段只能借用存续于整个程序执行过程中的值。这会带来一定的限制，但它确实表明 `D` 不可能借用局部变量，而 `D` 本身的生命周期并没有特殊限制。

来自 Rust 的错误消息其实建议了另一种方法，这种方法更通用：

```
help: consider introducing a named lifetime parameter
  |
7 | struct D<'a> {
8 |     s: S<'a>
  |
```

这一次，为 `D` 提供生命周期参数并将其传给 `s`：

```
struct D<'a> {
    s: S<'a>
}
```

通过获取生命周期参数 `'a` 并在 `s` 的类型中使用它，我们允许 Rust 将 `D` 值的生命周期和其 `s` 类型字段持有的引用的生命周期关联起来。

我们之前展示过函数的签名如何明确表达出它对我们传给它的引用做了什么。现在我们在类型方面展示了类似的做法：类型的生命周期参数总会揭示它是否包含某些值得关心其生命周期的引用（也就是非 `'static` 的）以及这些生命周期可以是什么。

假设我们有一个解析函数，它会接受一个字节切片并返回一个存有解析结果的结构：

```
fn parse_record<'i>(input: &'i [u8]) -> Record<'i> { ... }
```

不用看 `Record` 类型的定义就可以知道，如果从 `parse_record` 接收到 `Record`，那么它包含的任何引用就必然指向我们传入的输入缓冲区，而不是其他地方（`'static` 静态值除外）。

事实上，Rust 要求包含引用的类型都要接受显式生命周期参数就是为了明示这种内部行为。其实 Rust 原本可以简单地给结构体中的每个引用创建一个不同的生命周期，从而省去把它们写出来的麻烦。实际上，Rust 的早期版本就是这么做的，但开发人员发现这样会令人困惑：了解“某个值是从另一个值中借用出来的”这一点很有帮助，特别是在处理错误时。

不仅像 `s` 这样的引用和类型有生命周期，Rust 中的每个类型都有生命周期，包括 `i32` 和 `String`。它们大多数是 `'static` 的，这意味着这些类型的值可以一直存续下去，例如，`Vec<i32>` 是自包含的，在任何特定变量超出作用域之前都不需要丢弃它。但是像 `Vec<&'a i32>` 这样的类型，其生命周期就必须被 `'a` 涵盖，也就是说必须在引用目标仍然存续的情况下丢弃它。

5.3.6 不同的生命周期参数

假设你已经定义了一个包含两个引用的结构体，如下所示：

```
struct S<'a> {  
    x: &'a i32,  
    y: &'a i32  
}
```

这两个引用使用相同的生命周期 'a。如果这样写代码，那么可能会有问题：

```
let x = 10;  
let r;  
{  
    let y = 20;  
    {  
        let s = S { x: &x, y: &y };  
        r = s.x;  
    }  
}  
println!("{}", r);
```

上述代码不会创建任何悬空指针。对 y 的引用会保留在 s 中，它会在 y 之前超出作用域。对 x 的引用最终会出现在 r 中，它的生命周期不会超出 x。

然而，如果你尝试编译这段代码，那么 Rust 会报错说 y 的存活时间不够长，但其实它看起来是足够长的。为什么 Rust 会担心呢？如果仔细阅读代码，就能明白其推理过程。

- s 的两个字段是具有相同生命周期 'a 的引用，因此 Rust 必须找到一个同时适合 s.x 和 s.y 的生命周期。
- 赋值 r = s.x，这就要求 'a 涵盖 r 的生命周期。
- 用 &y 初始化 s.y，这就要求 'a 不能长于 y 的生命周期。

这些约束是不可能满足的：没有哪个生命周期比 y 短但比 r 长。Rust 被迫止步于此。

出现这个问题是因为 s 中的两个引用具有相同的生命周期 'a。只要更改 s 的定义，让每个引用都有各自的生命周期，就可以解决所有问题：

```
struct S<'a, 'b> {  
    x: &'a i32,  
    y: &'b i32  
}
```

根据这个定义，`s.x` 和 `s.y` 具有独立的生命周期。对 `s.x` 所做的操作不会影响 `s.y` 中存储的内容，因此现在很容易满足约束条件：'`a`' 可以用 `r` 的生命周期，而 '`b`' 可以用 `s` 的生命周期。（虽然 '`b`' 也可以用 `y` 的生命周期，但 Rust 会尝试选择可行的最小生命周期。）一切都好起来了。

函数签名也有类似的情况。假设有这样一个函数：

```
fn f<'a>(r: &'a i32, s: &'a i32) -> &'a i32 { r } // 可能过于严格
```

在这里，两个引用参数使用了相同的生命周期 '`a`'，这可能会给调用者施加不必要的限制，就像前面讲过的那样。如果这确实是问题，可以让各个参数的生命周期独立变化：

```
fn f<'a, 'b>(r: &'a i32, s: &'b i32) -> &'a i32 { r } // 宽松多了
```

这样做的缺点是，添加生命周期会让类型和函数签名更难阅读。我们（本书作者）倾向于先尝试尽可能简单的定义，然后放宽限制，直到代码能编译通过为止。由于 Rust 不允许不安全的代码运行，因此简单地等到报告问题时再修改也是一种完全可以接受的策略。

5.3.7 省略生命周期参数

迄今为止，本书已经展示了很多返回引用或以引用为参数的函数，但通常没必要详细说明每个生命周期。生命周期就在那里，如果它们显而易见，那么 Rust 就允许我们省略。

在最简单的情况下，你可能永远不需要为参数写出生命周期。Rust 会为需要生命周期的每个地方分配不同的生命周期。例如：

```
struct S<'a, 'b> {  
    x: &'a i32,
```

```
        y: &'b i32
    }

fn sum_r_xy(r: &i32, s: S) -> i32 {
    r + s.x + s.y
}
```

此函数的签名是以下代码的简写形式：

```
fn sum_r_xy<'a, 'b, 'c>(r: &'a i32, s: S<'b, 'c>) -> i32
```

如果确实要返回引用或其他带有生命周期参数的类型，那么针对无歧义的情况，Rust 会尽量采用简单的设计。如果函数的参数只有一个生命周期，那么 Rust 就会假设返回值具有同样的生命周期：

```
fn first_third(point: &[i32; 3]) -> (&i32, &i32) {
    (&point[0], &point[2])
}
```

明确写出所有生命周期后的代码如下所示：

```
fn first_third<'a>(point: &'a [i32; 3]) -> (&'a i32, &'a i32)
```

如果函数的参数有多个生命周期，那么就没有理由选择某一个生命周期作为返回值的生命周期，Rust 会要求你明确指定生命周期。

但是，如果函数是某个类型的方法，并且具有引用类型的 `self` 参数，那么 Rust 就会假定返回值的生命周期与 `self` 参数的生命周期相同。（`self` 指的是调用方法的对象，类似于 C++、Java 或 JavaScript 中的 `this` 或者 Python 中的 `self`。9.6 节会介绍这些方法。）

例如，你可以编写以下内容：

```
struct StringTable {
    elements: Vec<String>,
}

impl StringTable {
    fn find_by_prefix(&self, prefix: &str) -> Option<&String> {
```

```
        for i in 0 .. self.elements.len() {
            if self.elements[i].starts_with(prefix) {
                return Some(&self.elements[i]);
            }
        }
        None
    }
}
```

`find_by_prefix` 方法的签名是以下内容的简写形式：

```
fn find_by_prefix<'a, 'b>(&'a self, prefix: &'b str) -> Option<&'a String>
```

Rust 假定无论你借用的是什么，本质上都是从 `self` 借用的。

再次强调，这些都只是简写形式，旨在提供便利且比较直观。如果它们不是你想要的，那么你可以随时明确地写出生命周期。

5.4 共享与可变

迄今为止，本书讨论的都是 Rust 如何确保不会有任何引用指向超出作用域的变量。但是还有其它方法可能引入悬空指针。下面是一个简单的例子：

```
let v = vec![4, 8, 19, 27, 34, 10];  
let r = &v;  
let aside = v; // 把向量转移给aside  
r[0];          // 错误：这里所用的`v`此刻是未初始化状态
```

对 `aside` 的赋值会移动向量、让 `v` 回到未初始化状态，并将 `r` 变为悬空指针，如图 5-7 所示。

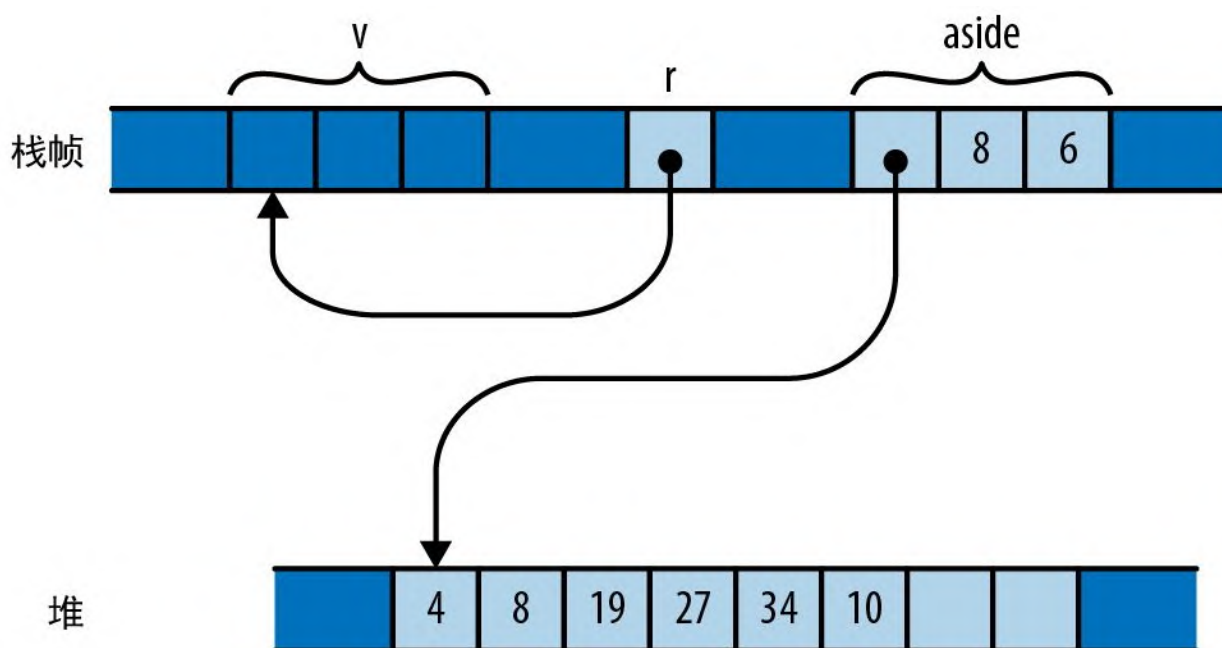


图 5-7：对已移动出去的向量的引用

尽管 `v` 在 `r` 的整个生命周期中都处于作用域内部，但这里的问题是 `v` 的值已经移动到别处，导致 `v` 成了未初始化状态，而 `r` 仍然在引用它。当然，Rust 会捕获错误：

```
error: cannot move out of `v` because it is borrowed
|
9 |     let r = &v;
|           - borrow of `v` occurs here
10 |     let aside = v; // 把向量转移给`aside`
|               ^^^^^ move out of `v` occurs here
```

在共享引用的整个生命周期中，它引用的目标会保持只读状态，即不能对引用目标赋值或将值移动到别处。在上述代码中，`r` 的生命周期内发生了移动向量的操作，Rust 当然要拒绝。如果按如下所示更改程序，就没问题了：

```
let v = vec![4, 8, 19, 27, 34, 10];
{
    let r = &v;
    r[0]; // 正确：向量仍然在那里
}
let aside = v;
```

在这个版本中，`r` 作用域范围更小，在把 `v` 转移给 `aside` 之前，`r` 的生命周期就结束了，因此一切正常。

下面是另一种制造混乱的方式。假设我们随手写了一个函数，它使用切片的元素来扩展某个向量：

```
fn extend(vec: &mut Vec<f64>, slice: &[f64]) {
    for elt in slice {
        vec.push(*elt);
    }
}
```

这是标准库中向量的 `extend_from_slice` 方法的一个不太灵活（并且优化程度较低）的版本。可以用它从其他向量或数组的切片中构建一个向量：

```
let mut wave = Vec::new();
let head = vec![0.0, 1.0];
let tail = [0.0, -1.0];

extend(&mut wave, &head); // 使用另一个向量扩展`wave`
extend(&mut wave, &tail); // 使用数组扩展`wave`
```

```
assert_eq!(wave, vec![0.0, 1.0, 0.0, -1.0]);
```

我们在这里建立了一个正弦波周期。如果想添加另一个周期，那么可以把向量追加到其自身吗？

```
extend(&mut wave, &wave);  
assert_eq!(wave, vec![0.0, 1.0, 0.0, -1.0,  
                      0.0, 1.0, 0.0, -1.0]);
```

乍一看你可能觉得这还不错。但别忘了，在往向量中添加元素时，如果它的缓冲区已满，那么就必须分配一个具有更多空间的新缓冲区。假设开始时 `wave` 有 4 个元素的空间，那么当 `extend` 尝试添加第五个元素时就必须分配更大的缓冲区。内存最终如图 5-8 所示。

`extend` 函数的 `vec` 参数借用了 `wave`（由调用者拥有），而 `wave` 为自己分配了一个新的缓冲区，其中有 8 个元素的空间。但是 `slice` 仍然指向旧的 4 元素缓冲区，该缓冲区已经被丢弃了。

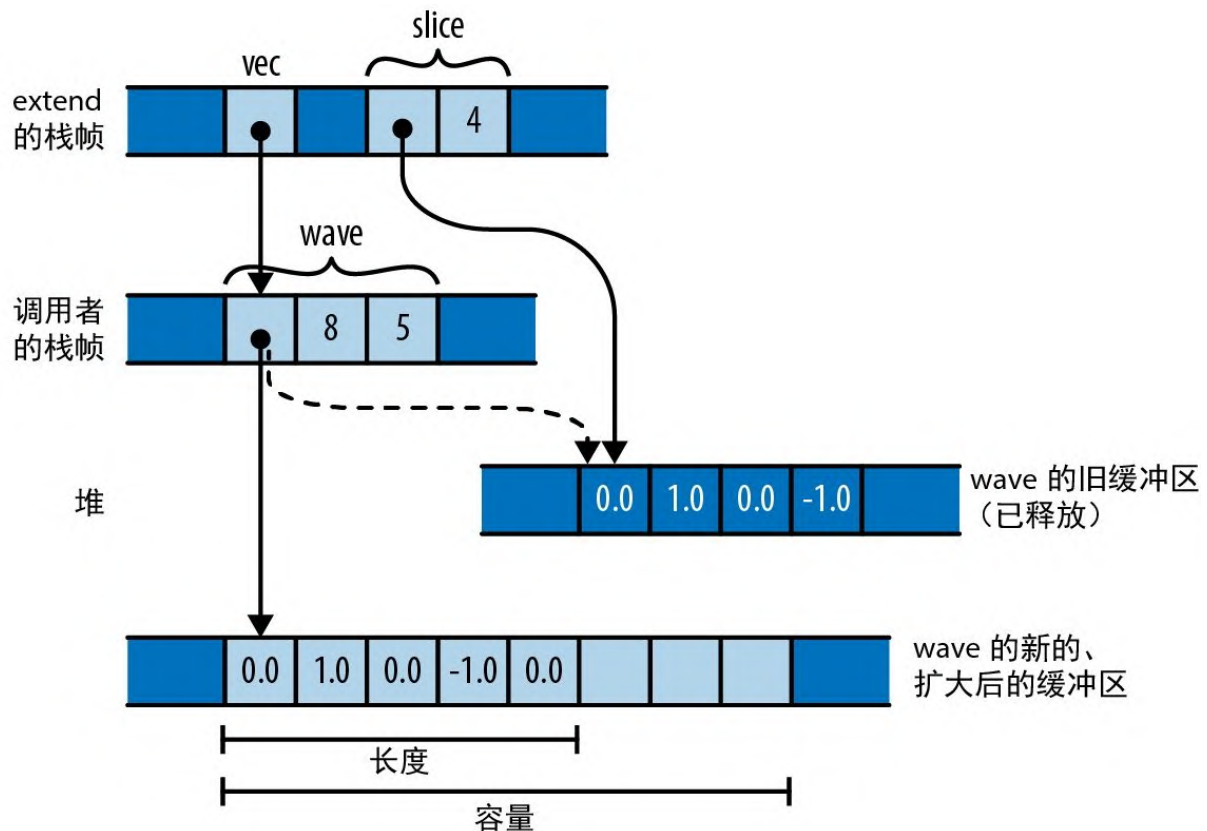


图 5-8：通过向量的重新分配将 `slice` 变成了悬空指针

这种问题并不是 Rust 独有的：在许多语言中，在指向集合的同时修改集合要加倍小心。在 C++ 中，`std::vector` 规范会告诫你“重新分配向量缓冲区会令指向序列中各个元素的所有引用、指针和迭代器失效”。Java 对修改 `java.util.Hashtable` 对象也有类似的说法。

如果在创建迭代器后的任何时间以任何方法（迭代器自身的 `remove` 方法除外）修改了 `Hashtable` 的结构，那么迭代器都将抛出 `ConcurrentModificationException` 异常。

这类错误特别难以调试，因为它只会偶尔发生。在测试中，向量可能总是恰好有足够的空间，缓冲区可能永远都不会重新分配，于是这个问题可能永远都没人发现。

然而，Rust 会在编译期报告调用 `extend` 有问题：

```
error: cannot borrow `wave` as immutable because it is also
      borrowed as mutable
  |
9 |         extend(&mut wave, &wave);
  |                   ---- ^^^^- mutable borrow ends here
  |                   |      |
  |                   |      immutable borrow occurs here
  |                   mutable borrow occurs here
```

换句话说，我们既可以借用向量的可变引用，也可以借用其元素的共享引用，但这两种引用的生命周期不能重叠。在这个例子中，这两种引用的生命周期都包含着对 `extend` 的调用，出现了重叠，因此 Rust 会拒绝执行这段代码。

这些错误都源于违反了 Rust 的“可变与共享”规则。

共享访问是只读访问。

共享引用借用的值是只读的。在共享引用的整个生命周期中，无论是它的引用目标，还是可从该引用目标间接访问的任何值，都不能

被任何代码改变。这种结构中不能存在对任何内容的有效可变引用，其拥有者应保持只读状态，等等。值完全冻结了。

可变访问是独占访问。

可变引用借用的值只能通过该引用访问。在可变引用的整个生命周期中，无论是它的引用目标，还是该引用目标间接访问的任何目标，都没有任何其他路径可访问。对可变引用来说，唯一能和自己的生命周期重叠的引用就是从可变引用本身借出的引用。

Rust 报告说 `extend` 示例违反了第二条规则：因为我们借用了 `wave` 的可变引用，所以该可变引用必须是抵达向量或其元素的唯一方式。而对切片的共享引用本身是抵达这些元素的另一种方式，这违反了第二条规则。

但是 Rust 也可以将我们的错误视为违反了第一条规则：因为我们借用了 `wave` 元素的共享引用，所以这些元素和 `vec` 本身都是只读的。不能对只读值借用出可变引用。

每种引用都会影响到我们可以对“到引用目标从属路径上的值”以及“从引用目标可间接访问的值”所能执行的操作，如图 5-9 所示。

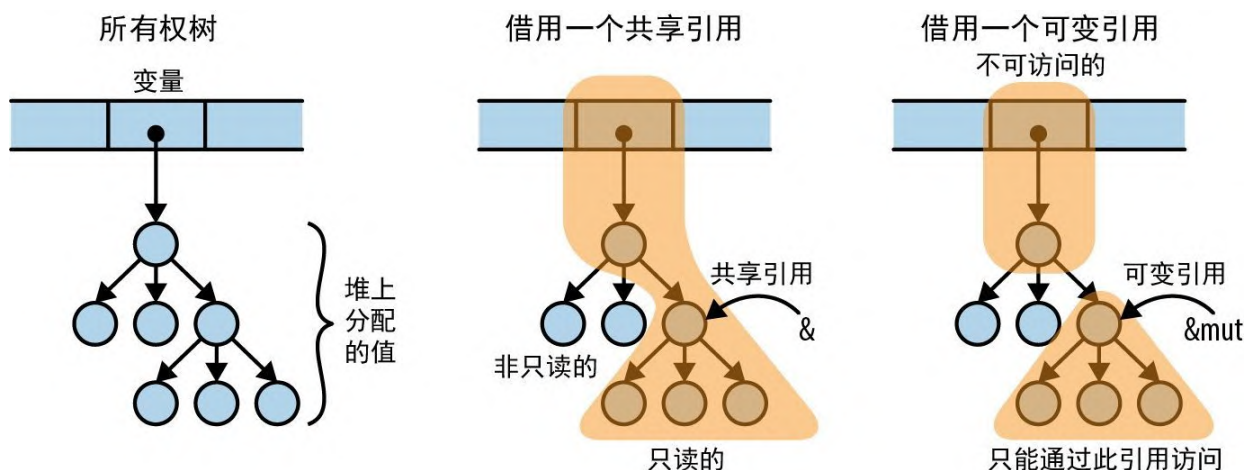


图 5-9：借用引用会影响你对同一所有权树中的其他值执行的操作

请注意，在这两种情况下，指向引用目标的所有权路径在此引用的生命周期内都无法更改。对于共享借用，这条路径是只读的；对于可变借用，这条路径是完全不可访问的。所以程序无法做出任何会使该引用无效的操作。

可以将这些原则分解为一些最简单的示例：

```
let mut x = 10;
let r1 = &x;
let r2 = &x;          // 正确：允许多个共享借用
x += 10;              // 错误：不能赋值给`x`，因为它已被借出
let m = &mut x;        // 错误：不能把`x`借入为可变引用，因为
                      // 它涵盖在已借出的不可变引用的生命周期内
println!("{}", {}, {}, {}, r1, r2, m); // 这些引用是在这里使用的，所以它们
                                         // 的生命周期至少要存续这么长

let mut y = 20;
let m1 = &mut y;
let m2 = &mut y;      // 错误：不能多次借入为可变引用
let z = y;            // 错误：不能使用`y`，因为它涵盖在已借出的可变引用的生命周期内
println!("{}", {}, {}, {}, m1, m2, z); // 在这里使用这些引用
```

可以从共享引用中重新借入共享引用：

```
let mut w = (107, 109);
let r = &w;
let r0 = &r.0;          // 正确：把共享引用重新借入为共享引用
let m1 = &mut r.1;      // 错误：不能把共享引用重新借入为可变
println!("{}", r0);     // 在这里使用r0
```

可以从可变引用中重新借入可变引用：

```
let mut v = (136, 139);
let m = &mut v;
let m0 = &mut m.0;      // 正确：从可变引用中借入可变引用
*m0 = 137;
let r1 = &m.1;          // 正确：从可变引用中借入共享引用，并且不能和m0重叠
v.1;                   // 错误：禁止通过其他路径访问
println!("{}", r1);     // 可以在这里使用r1
```

这些限制非常严格。回过头来看看我们尝试调用 `extend(&mut wave, &wave)` 的地方，没有什么快速而简便的方法来修复代码，以使其按照我们想要的方式工作。Rust 中到处都在应用这些规则：如果要借用对 `HashMap` 中键的共享引用，那么在共享引用的生命周期结束之前就不能再借入对 `HashMap` 的可变引用。

但这么做有充分的理由：要为集合设计出“支持不受限制地在迭代期间修改”的能力是非常困难的，而且往往会导致无法简单高效地实现这些集合。Java 的 `Hashtable` 和 C++ 的 `vector` 就不支持这种访问方式，Python 的字典和 JavaScript 的对象甚至都不曾定义过这种访问方式。JavaScript 中的其他集合类型固然可以做到，不过需要更繁重的实现。C++ 的 `std::map` 承诺插入新条目不会让指向此映射表中其他条目的指针失效，但做出这一承诺的代价是该标准无法提供像 Rust 的 `BTreeMap` 这样更高效的缓存设计方案，因为后者会在树的每个节点中存储多个条目。

下面是通过上述规则捕获各种错误的另一个例子。考虑以下 C++ 代码，它用于管理文件描述符。为了简单起见，这里只展示一个构造函数和复制赋值运算符，并会省略错误处理代码：

```
struct File {
    int descriptor;

    File(int d) : descriptor(d) { }

    File& operator=(const File &rhs) {
        close(descriptor);
        descriptor = dup(rhs.descriptor);
        return *this;
    }
};
```

这个赋值运算符很简单，但在下面这种情况下会执行失败：

```
File f(open("foo.txt", ...));
...
f = f;
```

如果将一个 `File` 赋值给它自己，那么 `rhs` 和 `*this` 就是同一个对象，所以 `operator=` 会关闭它要传给 `dup` 的文件描述符。也就是说，我们销毁了正打算复制的那份资源。

在 Rust 中，类似的代码如下所示：

```
struct File {
    descriptor: i32
}

fn new_file(d: i32) -> File {
    File { descriptor: d }
}

fn clone_from(this: &mut File, rhs: &File) {
    close(this.descriptor);
    this.descriptor = dup(rhs.descriptor);
}
```

（这并不是 Rust 的惯用法。有很多很好的方式可以让 Rust 类型拥有自己的构造函数和方法，第 9 章会对此进行讲解，刚才的定义方式仅仅是为了示范。）

如果编写使用了 `File` 的 Rust 代码，就会得到如下内容：

```
let mut f = new_file(open("foo.txt", ...));
...
clone from(&mut f, &f);
```

当然，Rust 干脆拒绝编译这段代码：

```
error: cannot borrow `f` as immutable because it is also
      borrowed as mutable
      |
18 |     clone_from(&mut f, &f);
      |               ^- mutable borrow ends here
      |               |
      |               | immutable borrow occurs here
      |               mutable borrow occurs here
```


以上错误看起来很熟悉。事实证明，这里的两个经典 C++ 错误（无法处理自赋值和使用无效迭代器）本质上是同一种错误。在这两种情况下，代码都以为自己正在修改一个值，同时在引用另一个值，但实际上两者是同一个值。如果你不小心让调用 `memcpy` 或 `strcpy` 的源和目标在 C 或 C++ 中重叠，则可能会带来另一种错误。通过要求可变访问必须是独占的，Rust 避免了一大类日常错误。

在编写并发代码时，共享引用和可变引用的互斥性确实证明了其价值。只有当某些值既可变又要在线程之间共享时，才可能出现数据竞争，而这正是 Rust 的引用规则所要消除的。一个用 Rust 编写的并发程序，只要避免使用 `unsafe` 代码，就可以在构造之初就避免产生数据竞争。第 19 章在讨论并发时会更详细地对此进行介绍。总而言之，与大多数其他语言相比，并发在 Rust 中更容易使用。

Rust 的共享引用与 C 的 `const` 指针

乍一看，Rust 的共享引用似乎与 C 和 C++ 中指向 `const` 值的指针非常相似。然而，Rust 中共享引用的规则要严格得多。例如，考虑以下 C 代码：

```
int x = 42;           // int变量，不是常量
const int *p = &x;    // 指向const int的指针
assert(*p == 42);
x++;                  // 直接修改变量
assert(*p == 43);     // “常量”指向的值发生了变化
```

`p` 是 `const int *` 这一事实意味着不能通过 `p` 本身修改它的引用目标，也就是说，禁止使用 `(*p)++`。但是可以直接通过 `x` 获取引用目标，`x` 不是 `const`，能以这种方式更改其值。C 家族的 `const` 关键字自有其用处，但与“常量”无关。

在 Rust 中，共享引用禁止对其引用目标进行任何修改，直到其生命周期结束：

```
let mut x = 42;       // 非常量型i32变量
let p = &x;           // 到i32的共享引用
assert_eq!(*p, 42);
```

```
x += 1; // 错误：不能对x赋值，因为它已被借出
assert_eq!(*p, 42); // 如果赋值成功，那么这应该是true
```

为了保证一个值是常量，需要追踪该值的所有可能路径，并确保它们要么不允许修改，要么根本不能使用。C 和 C++ 的指针不受限制，编译器无法对此进行检查。Rust 的引用总是与特定的生命周期相关联，因此可以在编译期检查它们。

5.5 应对复杂对象关系

自 20 世纪 90 年代自动内存管理兴起以来，所有程序都由大量复杂关联的对象构成，如图 5-10 所示。

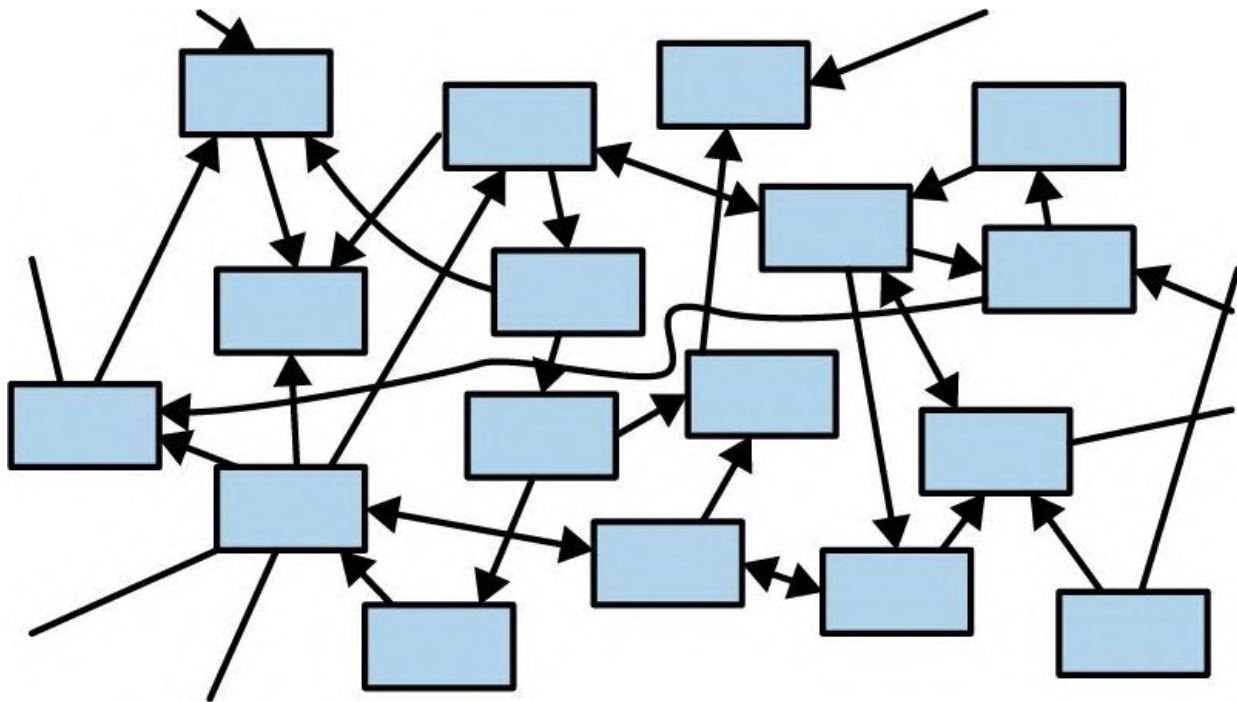


图 5-10：复杂对象关系

如果你采用垃圾回收（自动内存管理）并且在开始编写程序之前不做任何设计，就会发生这种情况。我们都构建过这样的系统。

这种架构有很多从图 5-10 中无法看出的优点：初始的进展迅速；很容易添加新功能；几年以后，你将很容易确定你需要完全重写它。

（让我们来一首澳大利亚摇滚乐队 AC/DC 的“通往地狱的高速公路”。²）

²意思是重写会变得迫在眉睫，但又非常困难。——译者注

当然，这种架构也有缺点。当每个部分都像这样依赖于其他部分时，必然很难测试、迭代，甚至很难单独考虑其中的任何组件。

Rust 令人着迷的地方之一就在于，其所有权模型就好像是在通向地狱的高速公路上铺设了一条减速带。在 Rust 中创建循环引用（两个值，每个值都包含指向另一个值的引用）相当困难。你必须使用智能指针类型（如 `Rc`）和内部可变性（目前为止本书还未涉及这个主题）。Rust 更喜欢让指针、所有权和数据流单向通过系统，如图 5-11 所示。

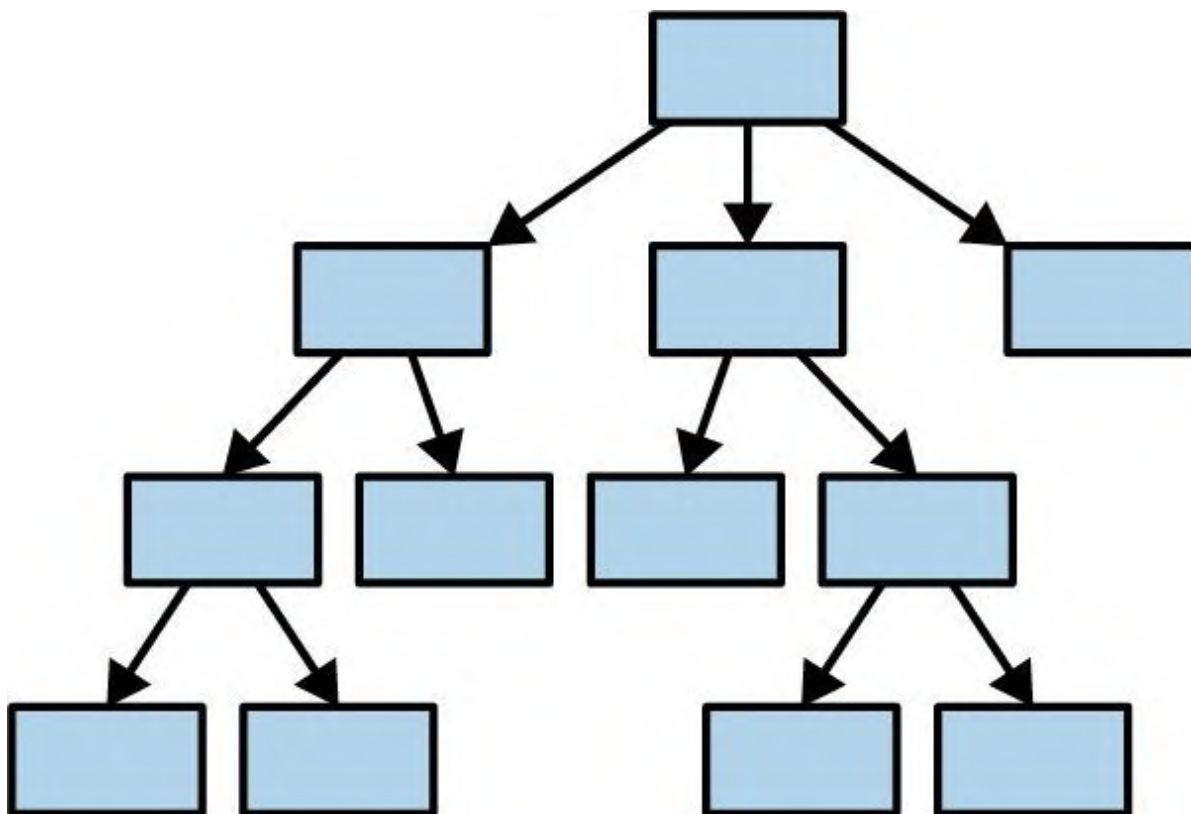


图 5-11：树形对象关系

之所以现在提出这个问题，是因为在阅读本章后，你可能会很自然地想要立即编写代码并创建出大量的对象，所有对象之间使用 `Rc` 智能指针关联起来，最终呈现你熟悉的所有面向对象反模式。但此刻这还行不通。Rust 的所有权模型会不断给你制造麻烦。解决之道是进行一些前期设计并构建出更好的程序。

Rust 就是要把你理解程序的痛苦从将来移到现在。它确实做到了：Rust 不仅会迫使你理解为什么自己的程序是线程安全的，甚至可能还需要你做一些高级架构设计。

第 6 章 表达式

LISP 程序员知道一切的价值 (value)，但不了解其代价。

——Alan Perlis, 警句 #55

本章将介绍 Rust 表达式，它是函数体的组成部分，因而也是大部分 Rust 代码的组成部分。Rust 中的大多数内容是表达式。本章将探索表达式的强大功能以及如何克服它的局限性。我们将介绍在 Rust 中完全面向表达式的控制流，以及 Rust 的基础运算符如何独立工作和组合工作。

某些概念（比如闭包和迭代器）严格来说也属于这一类，但略显深奥，我们稍后将用单独的一章来介绍它们。目前，我们的目标是在这“区区几页”中涵盖尽可能多的语法。

6.1 表达式语言

Rust乍看起来和 C 家族的语言相似，但这只是假象。在 C 语言中，表达式和语句之间有明显的区别，表达式看起来是这样的：

```
5 * (fahr-32) / 9
```

而语句看起来更像这样：

```
for (; begin != end; ++begin) {  
    if (*begin == target)  
        break;  
}
```

表达式有值，而语句没有。

Rust 是所谓的**表达式语言**。这意味着它遵循更古老的传统，可以追溯到 Lisp，在 Lisp 中，表达式能完成所有工作。

在 C 中，if 和 switch 是语句，它们不生成值，也不能在表达式中间使用。而在 Rust 中，if 和 match 可以生成值。第 2 章介绍过一个生成数值的 match 表达式：

```
pixels[r * bounds.0 + c] =  
    match escapes(Complex { re: point.0, im: point.1 }, 255) {  
        None => 0,  
        Some(count) => 255 - count as u8  
    };
```

if 表达式可用于初始化变量：

```
let status =  
    if cpu.temperature <= MAX_TEMP {  
        HttpStatus::Ok  
    } else {  
        HttpStatus::ServerError // 服务程序出错了  
    };
```

match 表达式可以作为参数传给函数或宏：

```
println!("Inside the vat, you see {}.",
    match vat.contents {
        Some(brain) => brain.desc(),
        None => "nothing of interest"
    });
```

这解释了为什么 Rust 没有 C 那样的三元运算符 (*expr1* ? *expr2* : *expr3*)。在 C 语言中，三元运算符是一个表达式级别的类似 if 语句的东西。这在 Rust 中是多余的：if 表达式足以处理这两种情况。

C 中的大多数控制流工具是语句。而在 Rust 中，它们都是表达式。

6.2 优先级与结合性

表 6-1 总结了 Rust 的表达式语法。本章将讨论所有这些类型的表达式。这里的运算符已按优先级顺序列出，从最高到最低。（与大多数编程语言一样，当一个表达式包含多个相邻的运算符时，Rust 会根据运算符优先级来确定运算顺序。例如，在 `limit < 2 * broom.size + 1` 中，`.` 运算符具有最高优先级，因此会最先访问字段。）

表 6-1：表达式

表达式类型	示例	相关特型
数组字面量	<code>[1, 2, 3]</code>	
数组重复表达式	<code>[0; 50]</code>	
元组	<code>(6, "crullers")</code>	
分组	<code>(2 + 2)</code>	
块	<code>{ f(); g() }</code>	

表达式类型	示例	相关特型
控制流表达式	<pre> if ok { f() } if ok { 1 } else { 0 } } if let Some(x) = f() { x } else { 0 } match x { None => 0, _ => 1 } for v in e { f(v); } while ok { ok = f(); } while let Some(x) = it.next() { f(x); } loop { next_event(); } break continue return 0 </pre>	std::iter::IntoIterator
宏调用	println!("ok")	
路径	std::f64::consts::PI	
结构体字面量	Point {x: 0, y: 0}	
元组字段访问	pair.0	Deref、DerefMut
结构体字段访问	point.x	Deref、DerefMut

表达式类型	示例	相关特型
方法调用	<code>point.translate(50, 50)</code>	Deref、DerefMut
函数调用	<code>stdin()</code>	<code>Fn(Arg0, ...) -> T</code> 、 <code>FnMut(Arg0, ...) -> T</code> 、 <code>FnOnce(Arg0, ...) -> T</code>
索引	<code>arr[0]</code>	Index、IndexMutDeref、DerefMut
错误检查	<code>create_dir("tmp")?</code>	
逻辑非 / 按位非	<code>!ok</code>	Not
取负	<code>-num</code>	Neg
解引用	<code>*ptr</code>	Deref、DerefMut
借用	<code>&val</code>	
类型转换	<code>x as u32</code>	
乘	<code>n * 2</code>	Mul
除	<code>n / 2</code>	Div

表达式类型	示例	相关特型
取余 (取模)	$n \% 2$	Rem
加	$n + 1$	Add
减	$n - 1$	Sub
左移	$n \ll 1$	Shl
右移	$n \gg 1$	Shr
按位与	$n \& 1$	BitAnd
按位异或	$n \wedge 1$	BitXor
按位或	$n 1$	BitOr
小于	$n < 1$	std::cmp::PartialOrd
小于等于	$n \leq 1$	std::cmp::PartialOrd
大于	$n > 1$	std::cmp::PartialOrd

表达式类型	示例	相关特型
大于等于	<code>n >= 1</code>	<code>std::cmp::PartialOrd</code>
等于	<code>n == 1</code>	<code>std::cmp::PartialEq</code>
不等于	<code>n != 1</code>	<code>std::cmp::PartialEq</code>
逻辑与	<code>x.ok && y.ok</code>	
逻辑或	<code>x.ok backup.ok</code>	
右开区间范围	<code>start .. stop</code>	
右闭区间范围	<code>start ..= stop</code>	
赋值	<code>x = val</code>	
复合赋值	<code>x *= 1</code> <code>x /= 1</code> <code>x %= 1</code> <code>x += 1</code> <code>x -= 1</code> <code>x <<= 1</code> <code>x >>= 1</code> <code>x &= 1</code> <code>x ^= 1</code> <code>x = 1</code>	<code>DivAssign</code> <code>RemAssign</code> <code>AddAssign</code> <code>SubAssign</code> <code>ShlAssign</code> <code>ShrAssign</code> <code>BitAndAssign</code> <code>BitXorAssign</code> <code>BitOrAssign</code> <code>MulAssign</code>

表达式类型	示例	相关特型
闭包	<code> x, y x + y</code>	

所有可以链式书写的运算符都是左结合的。也就是说，诸如 `a - b - c` 之类的操作链会分组为 `(a - b) - c`，而不是 `a - (b - c)`。所有可以这样链式书写的运算符都遵循左结合规则：

<code>*</code>	<code>/</code>	<code>%</code>	<code>+</code>	<code>-</code>	<code><<</code>	<code>>></code>	<code>&</code>	<code>^</code>	<code> </code>	<code>&&</code>	<code> </code>	<code>as</code>

比较运算符、赋值运算符和范围运算符（`..` 和 `..=`）则根本无法链式书写。

6.3 块与分号

块是一种最通用的表达式。一个块生成一个值，并且可以在任何需要值的地方使用：

```
let display_name = match post.author() {
    Some(author) => author.name(),
    None => {
        let network_info = post.get_network_metadata()?;
        let ip = network_info.client_address();
        ip.to_string()
    }
};
```

`Some(author) =>` 之后的代码是简单表达式 `author.name()`，`None =>` 之后的代码是一个块表达式，它们对 Rust 来说没什么不同。块表达式的值是最后一个表达式 `ip.to_string()` 的值。

请注意，`ip.to_string()` 方法调用后面没有分号。大多数 Rust 代码行以分号或花括号结尾，就像 C 或 Java 一样。如果一个块看起来很像 C 代码，在你熟悉的每个地方都有分号，那么它就会像 C 的块一样运行，并且其值为 `()`。正如第 2 章提到的，当块的最后一行不带分号时，就以最后这个表达式的值而不是通常的 `()` 作为块的值。

在某些语言，尤其是 JavaScript 中，可以省略分号，并且该语言会简单地替你填充分号——这是一个小小的便捷特性。但 Rust 不一样。在 Rust 中，分号是有实际意义的：

```
let msg = {
    // let声明：分号总是必需的
    let dandelion_control = puffball.open();

    // 带分号的表达式：调用方法，丢弃返回值
    dandelion_control.release_all_seeds(launch_codes);

    // 无分号的表达式：调用方法，返回值将存入`msg`
```

```
dandelion_control.get_status()
};
```

块可以包含多个声明并在末尾生成值，这是一个很好的特性，你很快就会适应它。但它的一个缺点是，如果你不小心遗漏了分号，则会导致奇怪的错误消息：

```
...
if preferences.changed() {
    page.compute_size() // 糟糕，丢了分号
}
...
```

如果在 C 或 Java 程序中犯了同样的错误，那么编译器会直接指出你漏了一个分号。但 Rust 会这么说：

```
error: mismatched types
22 |         page.compute_size() // 糟糕，丢了分号
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^- help: try adding a semicolon:
   |         `;`
   |         |
   |         expected (), found tuple
   |
= note: expected unit type `()`
        found tuple `(u32, u32)`
```

由于缺少分号，块的值将是 `page.compute_size()` 返回的任何值，但是没有 `else` 的 `if` 必定返回 `()`。幸好，Rust 已经针对这类错误做出改进，并会建议添加分号。

6.4 声明

除了表达式和分号，块还可以包含任意数量的声明。最常见的是 `let` 声明，它会声明局部变量：

```
let name: type = expr;
```

类型和初始化代码是可选的，分号则是必需的。与 Rust 中的所有标识符一样，变量名必须以字母或下划线开头，并且只能在第一个字符之后包含数字。Rust 中的“字母”是广义的，包括希腊字母、带重音的拉丁字符和更多符号——符合 Unicode 标准中附件 #31 要求的一切字符（也包括中文）。不允许使用表情符号。

`let` 声明可以在不初始化变量的情况下声明变量，然后再用赋值语句来初始化变量。这在某些情况下很有用，因为有时确实应该在某种控制流结构的中间初始化变量：

```
let name;
if user.has_nickname() {
    name = user.nickname();
} else {
    name = generate_unique_name();
    user.register(&name);
}
```

这里有两种初始化局部变量 `name` 的方式，但无论采用哪种方式，都只会初始化一次，所以 `name` 不需要声明为 `mut`。

在初始化之前就使用变量是错误的。（这与“移动后又使用值”的错误紧密相关。Rust 确实非常希望你只使用存在的值。）

你可能偶尔会看到似乎在重新声明现有变量的代码，如下所示：

```
for line in file.lines() {
    let line = line?;
```

```
    ...  
}
```

这个 `let` 声明会创建一个不同类型的、新的（第二个）变量。第一个 `line` 变量的类型是 `Result<String, io::Error>`。第二个 `line` 变量则是 `String`。第二个定义会在所处代码块的其余部分代替第一个定义。这叫作**遮蔽**（`shadowing`），在 Rust 程序中很常见。该代码等效于如下内容：

```
for line_result in file.lines() {  
    let line = line_result?;  
    ...  
}
```

本书会坚持在这种情况下使用 `_result` 后缀，以便让不同变量具有不同的名称。

块还可以包含**语法项声明**（`item declaration`）。语法项是指可以在程序或模块中的任意地方出现的声明，比如 `fn`、`struct` 或 `use`。

后面的章节会详细介绍这些语法项。现阶段，用 `fn` 这个例子就足够了。任何块都可能包含一个 `fn`：

```
use std::io;  
use std::cmp::Ordering;  
  
fn show_files() -> io::Result<()> {  
    let mut v = vec![];  
    ...  
    fn cmp_by_timestamp_then_name(a: &FileInfo, b: &FileInfo) ->  
Ordering {  
        a.timestamp.cmp(&b.timestamp)    // 首先，比较时间戳  
            .reverse()                    // 最新的文件优先  
            .then(a.path.cmp(&b.path))    // 通过路径做二级比较  
    }  
  
    v.sort_by(cmp_by_timestamp_then_name);  
    ...  
}
```

当在块内声明一个 `fn` 时，它的作用域是整个块，也就是说，它可以在整个封闭块内部使用。但是嵌套的 `fn` 无法访问恰好在同一作用域内的局部变量或参数。例如，函数 `cmp_by_timestamp_then_name` 不能直接使用 `v`。（封闭块与闭包不同。Rust 也有闭包，闭包可以看到封闭块作用域内的变量。请参阅第 14 章。）

块甚至可以包含完整的模块。这可能看起来有点儿过分（真的需要把语言的每一部分都嵌进任何其他部分吗？），但是程序员（特别是使用宏的程序员）总是有办法为语言提供的每一种独立语法找到用武之地。

6.5 if 与 match

if 表达式的形式我们很眼熟：

```
if condition1 {  
    block1  
} else if condition2 {  
    block2  
} else {  
    block_n  
}
```

每个 *condition* 都必须是 `bool` 类型的表达式，依照 Rust 的风格，不会隐式地将数值或指针转换为布尔值。

与 C 不同，条件周围不需要圆括号。事实上，如果出现了不必要的圆括号，那么 `rustc` 会给出警告。但花括号是必需的。

`else if` 块以及最后的 `else` 是可选的。没有 `else` 块的 `if` 表达式的行为与具有空的 `else` 块完全相同。

`match` 表达式类似于 C 语言中的 `switch` 语句，但更灵活。下面是一个简单的例子：

```
match code {  
    0 => println!("OK"),  
    1 => println!("Wires Tangled"),  
    2 => println!("User Asleep"),  
    _ => println!("Unrecognized Error {}", code)  
}
```

这类似于 `switch` 语句的用途。它将执行此 `match` 表达式的四个分支之一，具体执行哪个分支取决于 `code` 的值。通配符模式 `_` 会匹配所有内容。这类似于 `switch` 语句中的 `default:` 语句，不过它必须排在最后。将 `_` 模式放在其他模式之前意味着它会优先于其

他模式。这样一来，其他模式将永远没机会匹配到（编译器会发出警告）。

编译器可以使用跳转表来优化这种 `match`，就像 C++ 中的 `switch` 语句一样。当 `match` 的每个分支都生成一个常量值时，也会应用与 C++ 类似的优化。在这种情况下，编译器会构建出这些值的数组，并将各个 `match` 项编译为数组访问。除了边界检查，编译后的代码中根本不存在任何分支。

`match` 的多功能性源于每个分支 `=>` 左侧支持的多种模式

（`pattern`）。在上面的例子中，每个模式只是一个常量整数。我们还展示了用以区分两种 `Option` 值的 `match` 表达式：

```
match params.get("name") {  
    Some(name) => println!("Hello, {}!", name),  
    None => println!("Greetings, stranger.")  
}
```

对模式的强大能力来说，这还只是“冰山一角”。模式可以匹配一系列值，它可以解构元组、可以匹配结构体的各个字段、可以追踪引用、可以借用部分值，等等。甚至可以说，Rust 的模式定义了自己的迷你语言。第 10 章会用一些篇幅来介绍模式。

`match` 表达式的一般形式如下所示：

```
match value {  
    pattern => expr,  
    ...  
}
```

如果 `expr` 是一个块，则可以省略此分支之后的逗号。

Rust 会从第一项开始依次根据每个模式检查给定的 `value`。当模式能够匹配时，对应的 `expr` 会被求值，而当这个 `match` 表达式结束时，不会再检查别的模式。至少要有一个模式能够匹配。Rust 禁止执行未覆盖所有可能值的 `match` 表达式：

```
let score = match card.rank {  
  Jack => 10,  
  Queen => 10,  
  Ace => 11  
}; // 错误：未穷举所有模式
```

if 表达式的所有块都必须生成相同类型的值：

```
let suggested_pet =  
  if with_wings { Pet::Buzzard } else { Pet::Hyena }; // 正确  
  
let favorite_number =  
  if user.is_hobbit() { "eleventy-one" } else { 9 }; // 错误  
  
let best_sports_team =  
  if is_hockey_season() { "Predators" }; // 错误
```

（最后一个示例之所以是错的，是因为在 7 月份结果将是 `()`。）¹

¹因为 7 月份不是冰球赛季 `hockey_season`，所以会“走”入隐藏的 `else` 分支，返回 `()`。——译者注

类似地，`match` 表达式的所有分支都必须具有相同的类型。

```
let suggested_pet =  
  match favorites.element {  
    Fire => Pet::RedPanda,  
    Air => Pet::Buffalo,  
    Water => Pet::Orca,  
    _ => None // 错误：不兼容的类型  
  };
```

6.5.1 if let

还有一种 if 形式，即 `if let` 表达式：

```
if let pattern = expr {  
  block1  
} else {
```

```
    block2
}
```

给定的 *expr* 要么匹配 *pattern*，这时会运行 *block1*；要么无法匹配，这时会运行 *block2*。有时这是从 `Option` 或 `Result` 中获取数据的好办法：

```
if let Some(cookie) = request.session_cookie {
    return restore_session(cookie);
}

if let Err(err) = show_cheesy_anti_robot_task() {
    log_robot_attempt(err);
    politely_accuse_user_of_being_a_robot();
} else {
    session.mark_as_human();
}
```

`if let` 不是必需的，因为凡是 `if let` 可以做到的，`match` 同样可以做到。`if let` 表达式其实是只有一个模式的 `match` 表达式的简写形式。

```
match expr {
    pattern => { block1 }
    _ => { block2 }
}
```

6.5.2 循环

有 4 种循环表达式：

```
while condition {
    block
}

while let pattern = expr {
    block
}
```

```
loop {  
    block  
}  
  
for pattern in iterable {  
    block  
}
```

各种循环都是 Rust 中的表达式，但 while 循环或 for 循环的值总是 `()`，因此它们的值通常没什么用。如果指定了一个值，那么 loop 表达式就能生成一个值。

while 循环的行为与 C 中的等效循环完全一样，只不过其 *condition* 必须是 bool 类型。

while let 循环类似于 if let。在每次循环迭代开始时，*expr* 的值要么匹配给定的 *pattern*，这时会运行循环体 (block)；要么不匹配，这时会退出循环。

可以用 loop 来编写无限循环。它会永远重复执行循环体（直到遇上 break 或 return，或者直到线程崩溃）。

for 循环会对可迭代 (iterable) 表达式求值，然后为结果迭代器中的每个值运行一次循环体。许多类型可以迭代，包括所有标准库集合，比如 Vec 和 HashMap。标准的 C 语言的 for 循环如下所示：

```
for (int i = 0; i < 20; i++) {  
    printf("%d\n", i);  
}
```

在 Rust 中则是这样的：

```
for i in 0..20 {  
    println!("{}", i);  
}
```

与 C 一样，最后打印出的数值是 19。

.. 运算符会生成一个范围（range），即具有两个字段（start 和 end）的简单结构体。0..20 与 std::ops::Range { start: 0, end: 20 } 相同。各种范围都可以与 for 循环一起使用，因为 Range 是一种可迭代类型，它实现了 std::iter::IntoIterator 特型（参见第 15 章）。标准集合都是可迭代的，数组和切片也是如此。

为了与 Rust 的移动语义保持一致，把值用于 for 循环会消耗该值：

```
let strings: Vec<String> = error_messages();
for s in strings {                               // 在这里，每个String都会转移给s.....
    println!("{}", s);
}                                                  // .....并在此丢弃
println!("{}", error(s), strings.len()); // 错误：使用了已移动出去的值
```

这可能很不方便。简单的补救措施是在循环中访问此集合的引用。然后，循环变量也会变成对集合中每个条目的引用：

```
for rs in &strings {
    println!("String {:?} is at address {:p}.", *rs, rs);
}
```

这里 &strings 的类型是 &Vec<String>，rs 的类型是 &String。

遍历（可迭代对象的）可变引用会为每个元素提供一个可变引用：

```
for rs in &mut strings { // rs的类型是&mut String
    rs.push('\n'); // 为每个字符串添加一个换行
}
```

第 15 章会更详细地介绍 for 循环，并展示使用迭代器的许多其他方式。

6.6 循环中的控制流

`break` 表达式会退出所在循环。（在 Rust 中，`break` 只能用在循环中，不能用在 `match` 表达式中，这与 `switch` 语句不同。）

在 `loop` 的循环体中，可以在 `break` 后面跟一个表达式，该表达式的值会成为此 `loop` 的值：

```
// 对`next_line`的每一次调用，或者返回一个`Some(line)`（这里的`line`是
// 输入中的一行），或者当输入已结束返回`None`。最终会返回以"answer: "
// 开头的第1行，如果没找到，就返回"answer: nothing"
let answer = loop {
    if let Some(line) = next_line() {
        if line.starts_with("answer: ") {
            break line;
        }
    } else {
        break "answer: nothing";
    }
};
```

自然，`loop` 中的所有 `break` 表达式也必须生成具有相同类型的值，这样该类型就会成为这个 `loop` 本身的类型。

`continue` 表达式会跳转到循环的下一迭代：

```
// 读取某些数据，每次一行
for line in input_lines {
    let trimmed = trim_comments_and_whitespace(line);
    if trimmed.is_empty() {
        // 跳转回循环的顶部，并移到输入中的下一行
        continue;
    }
    ...
}
```

在 `for` 循环中，`continue` 会前进到集合中的下一个值，如果没有更多值，则退出循环。同样，在 `while` 循环中，`continue` 会重新检查循环条件，如果当前条件为假，就退出循环。

循环可以带有生命周期**标签**。在以下示例中，'search：是外部 for 循环的标签。因此，break 'search 会退出这层循环，而不是退出内部循环：

```
'search:
for room in apartment {
    for spot in room.hiding_spots() {
        if spot.contains(keys) {
            println!("Your keys are {} in the {}.", spot, room);
            break 'search;
        }
    }
}
```

break 可以同时具有标签和值表达式：

```
// 找到此系列中第一个完全平方数的平方根
let sqrt = 'outer: loop {
    let n = next_number();
    for i in 1.. {
        let square = i * i;
        if square == n {
            // 找到了一个平方根
            break 'outer i;
        }
        if square > n {
            // `n`不是完全平方数，尝试下一个
            break;
        }
    }
};
```

标签也可以与 continue 一起使用。

6.7 return 表达式

return 表达式会退出当前函数，并向调用者返回一个值。

不带值的 return 是 return () 的简写：

```
fn f() {           // 省略了返回类型：默认为()
    return;        // 省略了返回值：默认为()
}
```

函数不必有明确的 return 表达式。函数体的工作方式类似于块表达式：如果最后一个表达式后没有分号，则它的值就是函数的返回值。事实上，这是在 Rust 中提供函数返回值的首选方式。

但这并不意味着 return 是无用的，或者仅仅是对不熟悉表达式语言的用户做出的让步。与 break 表达式一样，return 可以放弃进行中的工作。例如，第 2 章就使用过 ? 运算符在调用可能失败的函数后检查错误：

```
let output = File::create(filename)?;
```

我们曾解释说这是 match 表达式的简写形式：

```
let output = match File::create(filename) {
    Ok(f) => f,
    Err(err) => return Err(err)
};
```

上述代码会首先调用 File::create(filename)。如果返回 Ok(f)，则整个 match 表达式的计算结果为 f，因此可以把 f 存储在 output 中，继续执行 match 后的下一行代码。

否则，我们将匹配 Err(err) 并抵达 return 表达式。这时候，对 match 表达式求值的具体结果会决定 output 变量的值。我们会放

弃所有这些并退出所在函数，返回从 `File::create()` 中得到的任何错误。

7.2.4 节会完整讲解 `?` 运算符。

6.8 为什么 Rust 中会有 loop

Rust 编译器中有几个部分会分析程序中的控制流。

- Rust 会检查通过函数的每条路径是否返回了预期返回类型的值。为了正确地做到这一点，它需要知道是否有可能抵达函数的末尾。
- Rust 会检查局部变量有没有在未初始化的情况下使用过。这就要检查通过函数的每一条路径，以确保只要不经过初始化此变量的代码，就无法抵达使用它的地方。
- Rust 会对不可达代码发出警告。如果无法通过函数抵达某段代码，则这段代码不可达。

以上这些称为**流敏感**（flow-sensitive）分析。这不是什么新事物，多年来，Java 一直在采用与 Rust 相似的“显式赋值”分析。

要执行这种规则，语言就必须在简单性和智能性之间取得平衡。简单性使得程序员更容易弄清楚编译器到底在说什么，而智能性有助于消除假警报和编译器拒绝一份完美而安全的程序的情况。Rust 更倾向于简单性，它的流敏感分析根本不会检查循环条件，而会简单地假设程序中的任何条件都可以为真或为假。

这会导致 Rust 可能拒绝某些安全程序：

```
fn wait_for_process(process: &mut Process) -> i32 {  
    while true {  
        if process.wait() {  
            return process.exit_code();  
        }  
    }  
} // 错误：类型不匹配：期待i32，实际找到了()
```

这里的错误是假警报。此函数只会通过 return 语句退出，因此 while 循环无法生成 i32 这个事实无关紧要。

loop 表达式就是这个问题的“有话直说”式解决方案。

Rust 的类型系统也会受到控制流的影响。前面说过，if 表达式的所有分支都必须具有相同的类型。但是，在可能以 break 或 return 表达式、无限 loop，或者调用 panic!() 或 std::process::exit() 等多种方式结束的块上强制执行此规则是不现实的。这些表达式的共同点是它们永远都不会以通常的方式结束并生成一个值。break 或 return 会突然退出当前块、无限 loop 则根本不会结束，等等。

所以，在 Rust 中，这些表达式没有正常类型。不能正常结束的表达式属于一个特殊类型 !，并且它们不受“类型必须匹配”这条规则的约束。可以在 std::process::exit() 的函数签名中看到 !：

```
fn exit(code: i32) -> !
```

此处的 ! 表示 exit() 永远不会返回，它是一个发散函数（divergent function）。

你可以用相同的语法编写自己的发散函数，这在某些情况下是很自然的：

```
fn serve_forever(socket: ServerSocket, handler: ServerHandler) -> ! {
    {
        socket.listen();
        loop {
            let s = socket.accept();
            handler.handle(s);
        }
    }
}
```

当然，如果此函数正常返回了，那么 Rust 就会认为它能正常返回反而是一个错误。

有了这些大规模控制流的构建块，就可以继续处理该流中常用的、更细粒度的表达式（比如函数调用和算术运算符）了。

6.9 函数与方法调用

Rust 中调用函数和方法的语法与许多其他语言中的语法相同：

```
let x = gcd(1302, 462); // 函数调用

let room = player.location(); // 方法调用
```

在此处的第二个示例中，`player` 是虚构类型 `Player` 的变量，它具有虚构的 `.location()` 方法。（第 9 章在讨论用户定义类型时会展示如何定义我们自己的方法。）

Rust 通常会在引用和它们所引用的值之间做出明确的区分。如果将 `&i32` 传给需要 `i32` 的函数，则会出现类型错误。你会注意到 `.` 运算符稍微放宽了这些规则。在调用 `player.location()` 的方法中，`player` 可能是一个 `Player`、一个 `&Player` 类型的引用，也可能是一个 `Box<Player>` 或 `Rc<Player>` 类型的智能指针。`.location()` 方法可以通过值或引用获取 `player`。同一个 `.location()` 语法适用于所有情况，因为 Rust 的 `.` 运算符会根据需要自动对 `player` 解引用或借入一个对它的引用。

第三种语法用于调用类型关联函数，比如 `Vec::new()`：

```
let mut numbers = Vec::new(); // 类型关联函数调用
```

这些语法类似于面向对象语言中的静态方法：普通方法会在值上调用（如 `my_vec.len()`），类型关联函数会在类型上调用（如 `Vec::new()`）。

自然，也支持链式方法调用：

```
// 来自第2章的基于Actix的Web服务器
server
    .bind("127.0.0.1:3000").expect("error binding server to
address")
    .run().expect("error running server");
```


Rust 语法的怪癖之一就是，在函数调用或方法调用中，泛型类型的常用语法 `Vec<T>` 是不起作用的：

```
return Vec<i32>::with_capacity(1000); // 错误：是某种关于“链式比较”的  
错误消息  
  
let ramp = (0 .. n).collect<Vec<i32>>(); // 同样的错误
```

这里的问题在于，在表达式中 `<` 是小于运算符。Rust 编译器建议用 `::<T>` 代替 `<T>`。这样就解决了问题：

```
return Vec::i32::with_capacity(1000); // 正确，改用::i32  
  
let ramp = (0 .. n).collect::i32
```

符号 `::<...>` 在 Rust 社区中被亲切地称为比目鱼（turbofish）。

或者，通常可以删掉类型参数，让 Rust 来推断它们：

```
return Vec::with_capacity(10); // 正确，只要fn的返回类型是Vec<i32>  
  
let ramp: Vec<i32> = (0 .. n).collect(); // 正确，前面已给定变量的类型
```

只要类型可以被推断，就省略类型，这是一种很好的代码风格。

6.10 字段与元素

你可以使用早已熟悉的语法访问结构体的字段。元组也一样，不过它们的字段是数值而不是名称：

```
game.black_pawns    // 结构体字段  
coords.1            // 元组元素
```

如果 `.` 左边的值是引用或智能指针类型，那么它就会像方法调用一样自动解引用。

方括号会访问数组、切片或向量的元素：

```
pieces[i]           // 数组元素
```

方括号左侧的值也会自动解引用。

像下面这样的 3 个表达式叫作左值，因为赋值时它们可以出现在左侧：

```
game.black_pawns = 0x00ff0000_00000000_u64;  
coords.1 = 0;  
pieces[2] = Some(Piece::new(Black, Knight, coords));
```

当然，只有当 `game`、`coords` 和 `pieces` 声明为 `mut` 变量时才允许这样做。

从数组或向量中提取切片的写法很直观：

```
let second_half = &game_moves[midpoint .. end];
```

这里的 `game_moves` 可以是数组、切片或向量，无论哪种方式，结果都是已被借出的长度为 `end - midpoint` 的切片。在 `second_half` 的生命周期内，`game_moves` 要被视为已借出的引用。

`..` 运算符允许省略任何一个操作数，它会根据存在的操作数最多生成 4 种类型的对象：

```
..          // RangeFull
a ..        // RangeFrom { start: a }
.. b        // RangeTo { end: b }
a .. b      // Range { start: a, end: b }
```

后两种形式是排除结束值（或半开放）的：结束值不包含在所表示的范围内。例如，范围 `0 .. 3` 包括数值 0、1 和 2，但不包括 3。

`..=` 运算符会生成包含结束值（或封闭）的范围，其中包括结束值：

```
..= b       // RangeToInclusive { end: b }
a ..= b     // RangeInclusive::new(a, b)
```

例如，范围 `0 ..= 3` 包括数值 0、1、2 和 3。

只有包含起始值的范围才是可迭代的，因为循环必须从某处开始。但是在数组切片中，这 6 种形式都可以使用。如果省略了范围的起点或末尾，则默认为被切片数据的起点或末尾。

因此，经典的分治算法快速排序 `quicksort` 的实现部分看起来可能像下面这样。

```
fn quicksort<T: Ord>(slice: &mut [T]) {
    if slice.len() <= 1 {
        return; // 无可排序
    }

    // 把slice分成两部分：前半片和后半片
    let pivot_index = partition(slice);

    // 对slice的前半片递归排序
    quicksort(&mut slice[.. pivot_index]);

    // 对slice的后半片递归排序
    quicksort(&mut slice[pivot_index + 1 ..]);
}
```

6.11 引用运算符

地址运算符 `&` 和 `&mut` 已在第 5 章中介绍过。

一元 `*` 运算符用于访问引用所指向的值。如你所见，当使用 `.` 运算符访问字段或方法时，Rust 会自动追踪引用，因此只有想要读取或写入引用所指的整个值时才需要用 `*` 运算符。

例如，有时迭代器会生成引用，但程序需要访问底层值：

```
let padovan: Vec<u64> = compute_padovan_sequence(n);
for elem in &padovan {
    draw_triangle(turtle, *elem);
}
```

在此示例中，`elem` 的类型为 `&u64`，因此 `*elem` 为 `u64`。

6.12 算术运算符、按位运算符、比较运算符和逻辑运算符

Rust 的二元运算符与许多其他语言中的二元运算符类似。为了节省时间，这里假设你熟悉其中某一种语言，并专注于 Rust 与传统语言不同的几个点。

Rust 有一些常用的算术运算符：`+`、`-`、`*`、`/` 和 `%`。如第 3 章所述，在调试构建中会检测到整数溢出并引发 `panic`。标准库为此提供了一些非检查（unchecked）的算术方法，比如 `a.wrapping_add(b)`。

整数除法会向 0 取整，而整数除以 0 会触发 `panic`，即使在发布构建中也是如此。标准库为整数提供了一个 `a.checked_div(b)` 方法，它将返回一个 `Option`（如果 `b` 为 0 则返回 `None`），并且不会引发 `panic`。

一元 `-` 运算符会对一个数取负。它支持除无符号整数之外的所有数值类型。没有一元 `+` 运算符。

```
println!("{}", -100);           // -100
println!("{}", -100u32);        // 错误：不能在类型`u32`上使用一元`-`运算符
println!("{}", +100);           // 错误：期待表达式，但发现了`+`
```

与在 C 中一样，`a % b` 会计算向 0 四舍五入的有符号余数或模数。其结果与左操作数的符号相同。注意，`%` 既能用于整数，也能用于浮点数：

```
let x = 1234.567 % 10.0; // 约等于4.567
```

Rust 还继承了 C 的按位整数运算符 `&`、`|`、`^`、`<<` 和 `>>`。但是，Rust 会使用 `!` 而不是 `~` 表示按位非：

```
let hi: u8 = 0xe0;
let lo = !hi; // 0x1f
```

这意味着对于整数 `n`，不能用 `!n` 来表示“`n` 为 0”，而是应该写成 `n == 0`。

移位总是对有符号整数类型进行符号扩展，对无符号整数类型进行零扩展。由于 Rust 具有无符号整数，因此它不需要诸如 Java 的 `>>>` 运算符之类的无符号移位运算符。

与 C 不同，Rust 中按位运算的优先级高于比较运算，因此如果编写 `x & BIT != 0`，那么就意味着 `(x & BIT) != 0`，正如预期的那样。这比在 C 中解释成的 `x & (BIT != 0)` 有用得多，后者会测试错误的位。

Rust 的比较运算符是 `==`、`!=`、`<`、`<=`、`>` 和 `>=`，参与比较的两个值必须具有相同的类型。

Rust 还有两个短路逻辑运算符 `&&` 和 `||`，它们的操作数都必须具有确切的 `bool` 类型。

6.13 赋值

`=` 运算符用于给 `mut` 变量及其字段或元素赋值。但是赋值在 Rust 中不像在其他语言中那么常见，因为默认情况下变量是不可变的。

如第 4 章所述，如果值是非 `Copy` 类型的，则赋值会将其移动到目标位置。值的所有权会从源转移给目标。目标的先前值（如果有的话）将被丢弃。

Rust 支持复合赋值：

```
total += item.price;
```

这等效于 `total = total + item.price;`。Rust 也支持其他运算符：`-=`、`*=` 等。完整列表参见表 6-1。

与 C 不同，Rust 不支持链式赋值：不能编写 `a = b = 3` 来将值 3 同时赋给 `a` 和 `b`。赋值在 Rust 中非常罕见，你是不会想念这种简写形式的。

Rust 没有 C 的自增运算符 `++` 和自减运算符 `--`。

6.14 类型转换

在 Rust 中，将值从一种类型转换为另一种类型通常需要进行显式转换。这种转换要使用 `as` 关键字：

```
let x = 17;                // x是i32类型的
let index = x as usize;    // 转换成usize
```

Rust 允许进行好几种类型的转换。

- 数值可以从任意内置数值类型转换为其他内置数值类型。

将一种整数类型转换为另一种整数类型始终是明确定义的。转换为更窄的类型会导致截断。转换为更宽类型的有符号整数会进行符号扩展，转换为无符号整数会进行零扩展，等等。简而言之，没有意外。

从浮点类型转换为整数类型会向 0 舍入，比如 `-1.99 as i32` 就是 `-1`。如果值太大而无法容纳整数类型，则转换会生成整数类型可以表示的最接近的值，比如 `1e6 as u8` 就是 `255`。

- `bool` 类型或 `char` 类型的值或者类似 C 的 `enum` 类型的值可以转换为任何整数类型。（第 10 章会介绍枚举。）

不允许向相反方向转换，因为 `bool` 类型、`char` 类型和 `enum` 类型都对其值有限制，必须通过运行期检查强制执行。例如，禁止将 `u16` 转换为 `char` 类型，因为某些 `u16` 值（如 `0xd800`）对应于 Unicode 的半代用区码点，因此无法生成有效的 `char` 值。有一个标准方法 `std::char::from_u32()`，它会执行运行期检查并返回一个 `Option<char>`，但更重要的是，这种转变的需求已经越来越少了。我们通常会一次转换整个字符串或流，Unicode 文本的算法通常比较复杂，最好留给库去实现。

作为例外，`u8` 可以转换为 `char` 类型，因为从 0 到 255 的所有整数都是 `char` 能持有的有效 Unicode 码点。

- 一些涉及不安全指针类型的转换也是允许的。参见 22.8 节。

我们说过通常需要进行强制转换。但一些涉及引用类型的转换非常直观，Rust 甚至无须强制转换就能执行它们。一个简单的例子是将可变引用转换为不可变引用。

不过，还可能会发生几个更重要的自动转换。

- `&String` 类型的值会自动转换为 `&str` 类型，无须强制转换。
- `&Vec<i32>` 类型的值会自动转换为 `&[i32]`。
- `&Box<Chessboard>` 类型的值会自动转换为 `&Chessboard`。

这些称为**隐式解引用**，因为它们适用于所有实现了内置特型 `Deref` 的类型。`Deref` 隐式转换的目的是使智能指针类型（如 `Box`）的行为尽可能像其底层值。多亏了 `Deref`，`Box<Chessboard>` 的用法基本上和普通 `Chessboard` 的用法一样。

用户定义类型也可以实现 `Deref` 特型。当你需要编写自己的智能指针类型时，请参阅 13.5 节。

6.15 闭包

Rust 也有闭包，即轻量级的类似函数的值。闭包通常由一个参数列表组成，在两条竖线之间列出，后跟一个表达式：

```
let is_even = |x| x % 2 == 0;
```

Rust 会推断其参数类型和返回类型。你也可以像写函数一样显式地写出它们。如果确实指定了返回类型，那么为了语法的完整性，闭包的主体必须是一个块：

```
let is_even = |x: u64| -> bool x % 2 == 0; // 错误  
let is_even = |x: u64| -> bool { x % 2 == 0 }; // 正确
```

调用闭包和调用函数的语法是一样的：

```
assert_eq!(is_even(14), true);
```

闭包是 Rust 最令人愉悦的特性之一，关于它们还有很多内容可以讲，第 14 章会详细介绍。

6.16 前路展望

表达式就是我们心目中的“可执行代码”，它们是 Rust 程序中编译成机器指令的那部分。然而，表达式也只是整个语言的一小部分。

大多数编程语言也是如此。程序的首要任务是执行，但这不是它唯一的任务。程序必须进行通信，必须是可测试的，必须保持组织性和灵活性，这样它们才能持续演进。程序还需要与其他团队构建的代码和服务进行互操作。就算只是为了执行，像 Rust 这样的静态类型语言的程序也需要更多的工具来组织数据，而不能仅仅使用元组和数组。

接下来，本书会用几章内容来讨论这些特性：首先是模块与 `crate`，它们会帮你对程序进行组织；然后是结构体与枚举，它们会帮你对数据进行组织。

不过，在此之前我们先来简单讨论一下“错误处理”这个重要主题。

第 7 章 错误处理

我知道只要活得足够久，这种事就一定会发生。

——萧伯纳论死亡

Rust 的错误处理方法很不寻常，值得用单独的一章来讨论。这里没有什么深奥的编程思想，只是对你来说可能有点儿新而已。本章介绍了 Rust 中的两类错误处理：panic 和 Result。

普通错误使用 Result 类型来处理。Result 通常用以表示由程序外部的事物引发的错误，比如错误的输入、网络中断或权限问题。这些问题并不是意料之中的，在没有任何 bug 的程序中也可能会不时出现。虽然本章大部分内容是关于 Result 的，但我们会先介绍 panic，因为它相对来说比较简单。

panic 针对的是另一种错误，即那种永远不应该发生的错误。¹

¹Rust 之所以会用一个新词（panic）而不是沿用“异常”来表达，是因为两者并不等价。
——译者注

7.1 panic

当程序遇到下列问题的时候，就可以断定程序自身存在 bug，故而会引发 panic：

- 数组越界访问；
- 整数除以 0；
- 在恰好为 Err 的 Result 上调用 .expect()；
- 断言失败。

(panic!()) 是一种宏，用于处理程序中出现错误的情况。当你的代码检测到出现错误并需要立即触发 panic 时，就可以使用这个宏。panic!() 可以接受类似于 println!() 的可选参数表，用于构建错误消息。)

以上情况的共同点是它们都由（不客气地说）程序员的错误所导致。而行之有效的一条经验法则是：“不要 panic”。

但每个人都会失误。如果这些不该发生的错误真的发生了，那么该怎么办呢？Rust 为你提供了一种选择。Rust 既可以在发生 panic 时展开调用栈，也可以中止进程。展开调用栈是默认方案。

7.1.1 展开调用栈

当海盗瓜分战利品时，船长会先分得一半，普通船员再对另一半进行平分。（众所周知，海盗讨厌小数，所以如果不能恰好平分，那么其结果将向下舍入，剩下的归船上的鹦鹉所有。）

```
fn pirate_share(total: u64, crew_size: usize) -> u64 {
    let half = total / 2;
    half / crew_size as u64
}
```

几个世纪以来，上述算法都能正常工作，直到有一天……船长成了抢劫后唯一的幸存者。如果将 0 作为 crew_size 传给此函数，那么

它将除以 0。在 C++ 中，这将是未定义行为。而在 Rust 中，这会触发 panic，通常会按如下方式处理。

- 把一条错误消息打印到终端。

```
thread 'main' panicked at 'attempt to divide by zero',  
pirates.rs:3780  
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

如果设置了 `RUST_BACKTRACE` 环境变量，那么就像这条消息中建议的，Rust 也会在这里转储当前的调用栈。

- 展开调用栈。这很像 C++ 的异常处理。

当前函数使用的任何临时值、局部变量或参数都将按照与创建它们时相反的顺序被丢弃。丢弃一个值仅仅意味着随后会进行清理：程序正在使用的任何字符串或向量都将被释放，所有打开的文件都将被关闭，等等。还会调用由用户定义的 `drop` 方法，请参阅 13.1 节。就 `pirate_share()` 函数而言，没有需要清理的内容。

清理了当前函数调用后，我们将继续执行到其调用者中，以相同的方式丢弃其变量和参数。然后再“走到”那个调用者的调用者中，在调用栈中逐级向上，以此类推。

- 最后，线程退出。如果 panic 线程是主线程，则整个进程退出（使用非零退出码）。

也许把 panic 作为这个有序过程的名称有误导性。panic 不是崩溃，也不是未定义行为。它更像是 Java 中的 `RuntimeException` 或 C++ 中的 `std::logic_error`。其行为是明确定义的，只是本就不该发生罢了。

panic 是安全的，没有违反 Rust 的任何安全规则，即使你故意在标准库方法的中间引发 panic，它也永远不会在内存中留下悬空指针或半初始化的值。Rust 的设计理念是要在出现任何意外之前捕获诸如无效数组访问之类的错误。继续往下执行显然是不安全的，所以 Rust 会展开这个调用栈。但是进程的其余部分可以继续运行。

panic 是基于线程的。一个线程 panic 时，其他线程可以继续做自己的事。第 19 章会展示父线程如何发现子线程中的 panic 并优雅地处理错误。

还有一种方法可以捕获调用栈展开，让线程“存活”并继续运行。标准库函数 `std::panic::catch_unwind()` 可以做到这一点。本章不会介绍如何使用它，但这是 Rust 的测试工具用于在测试中断言失败时进行恢复的机制。（在编写可以从 C 或 C++ 调用的 Rust 代码时，这种机制是必需的，因为跨越非 Rust 代码展开调用栈是未定义行为，详情请参阅第 22 章。）

人无完人，没有 bug 且不会出现 panic 的代码只存在于理想之中。为了使程序更加健壮，可以使用线程和 `catch_unwind()` 来处理 panic。但有一个重要的限制，即这些工具只能捕获那些会展开调用栈的 panic。不过，并非所有 panic 都会展开调用栈。

7.1.2 中止

调用栈展开是默认的 panic 处理行为，但在两种情况下 Rust 不会试图展开调用栈。

如果 Rust 在试图清理第一个 panic 时，`.drop()` 方法触发了第二个 panic，那么这个 panic 就是致命的。Rust 会停止展开调用栈并中止整个进程。

此外，Rust 处理 panic 的行为是可定制的。如果使用 `-C panic=abort` 参数进行编译，那么程序中的第一个 panic 会立即中止进程。（如果使用这个选项，那么 Rust 就不需要知道如何展开调用栈，故此可以减小编译后的代码的大小。）

对 Rust 中 panic 机制的讨论到此结束。需要说的就这些，因为普通的 Rust 代码没有处理 panic 的义务。即使你确实使用了线程或 `catch_unwind()`，所有的 panic 处理代码也会集中在几个地方。期望程序中的每个函数都能预测并处理其自身代码中的 bug 是不合理的。但由其他因素引起的错误就是另一回事了。

7.2 Result

Rust 中没有异常。相反，函数执行失败时会有像下面这样的返回类型：

```
fn get_weather(location: LatLng) -> Result<WeatherReport,
io::Error>
```

`Result` 类型会指示出可能的失败。当我们调用 `get_weather()` 函数时，它要么返回一个**成功结果** `Ok(weather)`，其中的 `weather` 是一个新的 `WeatherReport` 值；要么返回一个**错误结果** `Err(error_value)`，其中的 `error_value` 是一个 `io::Error`，用来解释出了什么问题。

每当调用此函数时，Rust 都会要求我们编写某种错误处理代码。如果不对 `Result` 执行某些操作，就无法获取 `WeatherReport`；如果未使用 `Result` 值，就会收到编译器警告。

第 10 章将介绍标准库如何定义 `Result` 以及我们如何自定义出类似的类型。本章将采用类似“食谱”的方式并专注于使用 `Result` 来实现你期望的错误处理行为。你将了解如何捕获错误、传播错误和报告错误，以及关于组织和使用 `Result` 类型的常见模式。

7.2.1 捕获错误

第 2 章中已经展示过 `Result` 最彻底的处理方式：使用 `match` 表达式。

```
match get_weather(hometown) {
    Ok(report) => {
        display_weather(hometown, &report);
    }
    Err(err) => {
        println!("error querying the weather: {}", err);
        schedule_weather_retry();
    }
}
```

```
}  
}
```

这相当于其他语言中的 `try/catch`。如果想直接处理错误而不是将错误传给调用者，就可以使用这种方式。

`match` 有点儿冗长，因此 `Result<T, E>` 针对一些常见的特定场景提供了多个有用的方法，每个方法在其实现中都有一个 `match` 表达式。（有关 `Result` 方法的完整列表，请查阅在线文档。此处列出的是最常用的方法。）

`result.is_ok()`（已成功）和 `result.is_err()`（已出错）

返回一个 `bool`，告知此结果是成功了还是出错了。

`result.ok()`（成功值）

以 `Option<T>` 类型返回成功值（如果有的话）。如果 `result` 是成功的结果，就返回 `Some(success_value)`；否则，返回 `None`，并丢弃错误值。

`result.err()`（错误值）

以 `Option<E>` 类型返回错误值（如果有的话）。

`result.unwrap_or(fallback)`（解包或回退值）

如果 `result` 为成功结果，就返回成功值；否则，返回 `fallback`，丢弃错误值。

```
// 对南加州而言，这是一则十拿九稳的天气预报  
const THE_USUAL: WeatherReport = WeatherReport::Sunny(72);  
  
// 如果可能，就获取真实的天气预报；如果不行，就回退到常见状态  
let report = get_weather(los_angeles).unwrap_or(THE_USUAL);  
display_weather(los_angeles, &report);
```

这是 `.ok()` 的一个很好的替代方法，因为返回类型是 `T`，而不是 `Option<T>`。当然，只有存在合适的回退值时，才能用这个方法。

`result.unwrap_or_else(fallback_fn)`（解包，否则调用）

这个方法也一样，但不会直接传入回退值，而是传入一个函数或闭包。它针对的是大概率不会用到回退值且计算回退值会造成浪费的情况。只有在得到错误结果时才会调用 `fallback_fn`。

```
let report =  
    get_weather(hometown)  
    .unwrap_or_else(|_err| vague_prediction(hometown));
```

（第 14 章会详细介绍闭包。）

`result.unwrap()`（解包）

如果 `result` 是成功结果，那么此方法同样会返回成功值。但如果 `result` 是错误结果，则会引发 `panic`。此方法有其应用场景，后面会详细讨论。

`result.expect(message)`（期待）

与 `.unwrap()` 相同，但此方法允许你提供一条消息，在发生 `panic` 时会打印该消息。

最后是处理 `Result` 引用的两个方法。

`result.as_ref()`（转引用）

将 `Result<T, E>` 转换为 `Result<&T, &E>`。

`result.as_mut()`（转可变引用）

与上一个方法一样，但它借入了一个可变引用，其返回类型是 `Result<&mut T, &mut E>`。

最后这两个方法之所以有用，是因为前面列出的所有其他方法，除了 `.is_ok()` 和 `.is_err()`，都在消耗 `result`。也就是说，它们会按值接受 `self` 参数。有时在不破坏 `result` 的情况下访问 `result` 中的数据是非常方便的，这就是 `.as_ref()` 和 `.as_mut()` 的用武之地。假设你想调用 `result.ok()`，但要让 `result` 保持不可变状态，那么就可以写成 `result.as_ref().ok()`，它只会借用 `result`，返回 `Option<&T>` 而非 `Option<T>`。

7.2.2 Result 类型别名

有时你会看到 Rust 文档中似乎忽略了 `Result` 中的错误类型：

```
fn remove_file(path: &Path) -> Result<()>
```

这意味着正在使用 `Result` 的类型别名。

类型别名是类型名称的一种简写形式。模块通常会定义一个 `Result` 类型的别名，以免重复编写模块中几乎每个函数都要用到的 `Error` 类型。例如，标准库的 `std::io` 模块包括下面这行代码：

```
pub type Result<T> = result::Result<T, Error>;
```

这定义了一个公共类型 `std::io::Result<T>`，它是 `Result<T, E>` 的别名，但将错误类型硬编码为 `std::io::Error`。实际上，这意味着如果你写下 `use std::io;`，那么 Rust 就会将 `io::Result<String>` 当作 `Result<String, io::Error>` 的简写形式。

当在线文档中出现类似 `Result<()>` 的内容时，可以单击标识符 `Result` 以查看正在使用的类型别名并了解其错误类型。实践中，错

误类型在上下文中通常是显而易见的。

7.2.3 打印错误

有时处理错误的唯一方法是将其转储到终端并继续执行。前面已经展示过这样处理的一种方法：

```
println!("error querying the weather: {}", err);
```

标准库定义了几种名称平平无奇的错误类型：`std::io::Error`、`std::fmt::Error`、`std::str::Utf8Error` 等。它们都实现了一个公共接口，即 `std::error::Error` 特型，这意味着它们都有以下特性和方法。

`println!()`（打印）

所有错误类型都可以通过 `println!()` 打印出来。使用格式说明符 `{}` 打印错误通常只会显示一条简短的错误消息。或者，也可以使用格式说明符 `{:?}`，以获得该错误的 Debug 视图。虽然这对用户不太友好，但包含额外的技术信息。

```
// `println!("error: {}", err);`的结果
error: failed to look up address information: No address
associated with
hostname

// `println!("error: {:?}", err);`的结果
error: Error { repr: Custom(Custom { kind: Other, error:
StringError(
"failed to look up address information: No address associated with
hostname") }) }
```

`err.to_string()`（转字符串）

以 `String` 形式返回错误消息。

`err.source()`（错误来源）

返回导致 `err` 的底层错误的 `Option`（如果有的话）。例如，网络错误可能导致银行交易失败，进而导致你的游艇被收回。如果 `err.to_string()` 是 "boat was repossessed"，那么 `err.source()` 可能会返回关于本次交易失败的错误。该错误的 `.to_string()` 可能是 "failed to transfer \$300 to United Yacht Supply"，而该错误的 `.source()` 可能是一个 `io::Error`（第二个错误），其中包含导致这一切乱象的特定网络中断的详细信息。第三个错误是根本原因，因此它的 `.source()` 方法应该返回 `None`。由于标准库仅包含相当底层的特性，因此从标准库返回的错误来源（`.source()`）通常都是 `None`。

打印一个错误值并不会打印出其来源。如果想确保打印所有可用信息，请使用下面这个函数：

```
use std::error::Error;
use std::io::{Write, stderr};

/// 把错误消息转储到`stderr`
///
/// 如果在构建此错误消息或将其写入`stderr`期间发生了另一个错误，就忽略新的错误
fn print_error(mut err: &dyn Error) {
    let _ = writeln!(stderr(), "error: {}", err);
    while let Some(source) = err.source() {
        let _ = writeln!(stderr(), "caused by: {}", source);
        err = source;
    }
}
```

`writeln!` 宏类似于 `println!`，但它会将数据写入所选的流。在这里，我们将错误消息写入了标准错误流 `std::io::stderr`。可以使用 `eprintln!` 宏做同样的事情，但是如果 `eprintln!` 中发生了错误，就会 `panic`。在 `print_error` 中，要忽略在写入消息时出现的错误，稍后 7.2.7 节会解释原因。

标准库的这些错误类型不包括调用栈跟踪，但是，当与不稳定版本的 Rust 编译器一起使用时，可以使用广受欢迎的 `anyhow` crate 提供

的一个现成的错误类型。（直到 Rust 1.50 为止，标准库中用于捕获回溯跟踪的函数尚未稳定。）

7.2.4 传播错误

大多数时候，当我们试图做某些可能失败的事情时，可能不想立即捕获并处理错误。如果在每个可能出错的地方都要使用十来行 `match` 语句，那代码就太多了。

因此，当发生某种错误时，我们通常希望让调用者去处理。也就是说，我们希望错误沿着调用栈向上传播。

Rust 的 `?` 运算符可以执行此操作。可以为任何生成 `Result` 的表达式加上一个 `?`，比如将其加在函数调用的结果后面：

```
let weather = get_weather(hometown)?;
```

`?` 的行为取决于此函数是返回了成功结果还是错误结果。

- 如果是成功结果，那么它会解包 `Result` 以获取其中的成功值。这里的 `weather` 类型不是 `Result<WeatherReport, io::Error>`，而是简单的 `WeatherReport`。
- 如果是错误结果，那么它会立即从所在函数返回，将错误结果沿着调用链向上传播。为了确保此操作有效，`?` 只能在返回类型为 `Result` 的函数中的 `Result` 值上使用。

`?` 运算符并无任何神奇之处。可以使用 `match` 表达式来表达同样的意图，只是更冗长：

```
let weather = match get_weather(hometown) {  
    Ok(success_value) => success_value,  
    Err(err) => return Err(err)  
};
```

`?` 运算符与 `match` 表达式唯一的区别在于，它有一些涉及类型和转换的知识点。7.2.5 节会介绍这些细节。

在旧式代码中，你可能还会看到 `try!()` 宏，在 Rust 1.13 引入 `?` 运算符之前，这是传播错误的常用方法：

```
let weather = try!(get_weather(hometown));
```

此宏会扩展为一个 `match` 表达式，就像之前那段代码一样。

人们很难意识到在程序中出现错误的情况有多普遍，尤其是在与操作系统交互的代码中。`?` 运算符有时几乎会出现在函数的每一行中：

```
use std::fs;
use std::io;
use std::path::Path;

fn move_all(src: &Path, dst: &Path) -> io::Result<()> {
    for entry_result in src.read_dir()? { // 打开目录可能失败
        let entry = entry_result?;       // 读取目录可能失败
        let dst_file = dst.join(entry.file_name());
        fs::rename(entry.path(), dst_file)?; // 重命名可能失败
    }
    Ok(()) // 哦.....总算结束了!
}
```

`?` 的作用也与 `Option` 类型相似。在返回 `Option` 类型的函数中，也可以使用 `?` 解包某个值，这样当遇到 `None` 时就会提前返回。

```
let weather = get_weather(hometown).ok()?;
```

7.2.5 处理多种 **Error** 类型

通常，不止一个操作会出错。假设我们只想从文本文件中读取数值：

```
use std::io::{self, BufRead};

/// 从文本文件中读取整数
/// 该文件中应该每行各有一个数值
fn read_numbers(file: &mut dyn BufRead) -> Result<Vec<i64>,
io::Error> {
    let mut numbers = vec![];
    for line_result in file.lines() {
```



```
        let line = line_result?;           // 读取各行可能失败
        numbers.push(line.parse()?);       // 解析整数可能失败
    }
    Ok(numbers)
}
```

Rust 会报告一个编译器错误：

```
error: `?` couldn't convert the error to `std::io::Error`

    numbers.push(line.parse()?);           // 解析整数可能失败
                        ^
                the trait
`std::convert::From<std::num::ParseIntError>`
    is not implemented for `std::io::Error`

note: the question mark operation (`?`) implicitly performs a
conversion
on the error value using the `From` trait
```

当我们读到第 11 章（介绍了相关特型）时，本错误消息中的这些术语会更有意义。现在，只需要注意 Rust 正在报错说？运算符不能将 `std::num::ParseIntError` 值转换为 `std::io::Error` 类型就可以了。

这里的问题在于从文件中读取一行并解析一个整数时会生成两种潜在错误类型。`line_result` 的类型是 `Result<String, std::io::Error>`。`line.parse()` 的类型是 `Result<i64, std::num::ParseIntError>`。而我们的 `read_numbers()` 函数的返回类型只能容纳 `io::Error`。Rust 试图将 `ParseIntError` 转换为 `io::Error`，但是无法进行这样的转换，所以我们得到了一个类型错误。

有几种方法可以解决这个问题。例如，第 2 章中用于创建曼德博集图像文件的 `image crate` 定义了自己的错误类型 `ImageError`，并实现了从 `io::Error` 和其他几种错误类型到 `ImageError` 的转换。如果你想采用这种方法，那么可以试试 `thiserror crate`，它旨在帮助你用几行代码就定义出良好的错误类型。

还有一种更简单的方法是使用 Rust 中内置的特性。所有标准库中的错误类型都可以转换为类型 `Box<dyn std::error::Error + Send + Sync + 'static>`。这虽然有点儿啰唆，不过也不算难：`dyn std::error::Error` 表示“任何错误”，`Send + Sync + 'static` 表示可以安全地在线程之间传递，而这往往是我们的要求。²为便于使用，还可以定义类型别名：

²还应该考虑使用比较受欢迎的 `anyhow` crate，它提供的错误类型和结果类型与 `GenericError` 和 `GenericResult` 非常相似，但有一些不错的附加特性。

```
type GenericError = Box<dyn std::error::Error + Send + Sync +  
'static>;  
type GenericResult<T> = Result<T, GenericError>;
```

然后，将 `read_numbers()` 的返回类型改为 `GenericResult<Vec<i64>>`。这样一来，函数就可以编译了。`?` 运算符会根据需要自动将任意类型的错误转换为 `GenericError`。

顺便说一句，`?` 运算符使用了一种标准方法进行这种自动转换。你也可以使用这种方法将任何错误转换为 `GenericError` 类型，为此，可以调用 `GenericError::from()`：

```
let io_error = io::Error::new(           // 制作自己的io::Error  
    io::ErrorKind::Other, "timed out");  
return Err(GenericError::from(io_error)); // 手动转换成GenericError
```

第 13 章会全面介绍 `From` 特型及其 `from()` 方法。

`GenericError` 方法的缺点是返回类型不再准确地传达调用者可预期的错误类型。调用者必须做好应对任何情况的准备。

如果你正在调用一个返回 `GenericResult` 的函数，并且想要处理一种特定类型的错误，而让所有其他错误传播出去，那么可以使用泛型方法 `error.downcast_ref::<ErrorType>()`。如果这个错误恰好是你要找的那种类型的错误，那么该方法就会借用对它的引用：

```

loop {
    match compile_project() {
        Ok(()) => return Ok(()),
        Err(err) => {
            if let Some(mse) = err.downcast_ref::<
<MissingSemicolonError>>() {
                insert_semicolon_in_source_code(mse.file(),
mse.line())?;
                continue; // 再试一次!
            }
            return Err(err);
        }
    }
}

```

许多语言提供了内置语法来执行此操作，但事实证明没什么必要。Rust 可以用 `downcast_ref` 方法代替它。

7.2.6 处理“不可能发生”的错误

有时我们明确知道某个错误不可能发生。假设我们正在编写代码来解析配置文件，并且确信文件中接下来的内容肯定是一串数字：

```

if next_char.is_digit(10) {
    let start = current_index;
    current_index = skip_digits(&line, current_index);
    let digits = &line[start..current_index];
    ...
}

```

我们想将这个数字串转换为实际的数值。有一个标准方法可以做到这一点：

```

let num = digits.parse::();

```

现在的问题是：`str.parse::()` 方法不返回 `u64`，而是返回了一个 `Result`。转换可能会失败，因为某些字符串不是数值：

```

"bleen".parse::() // ParseIntError: 无效的数字

```

但我们碰巧知道，在这种情况下，`digits` 一定完全由数字组成。那么应该怎么办呢？

如果我们正在编写的代码已经返回了 `GenericResult`，那么就可以添加一个 `?`，并且忽略这个错误。否则，我们将不得不为处理不可能发生的错误而烦恼。最好的选择是使用 `Result` 的 `.unwrap()` 方法。如果结果是 `Err`，就会 `panic`；但如果成功了，则会直接返回 `Ok` 中的成功值：

```
let num = digits.parse::() .unwrap();
```

这和 `?` 的用法很相似，但如果我们对这个错误有没有可能发生的理解是错误的，也就是说如果它其实有可能发生，那么这种情况就会报 panic。

事实上，对于刚才这个例子，我们确实理解错了。如果输入中包含足够长的数字串，则这个数值会因为太大而无法放入 `u64` 中：

```
"99999999999999999999".parse::() // 溢出错误
```

因此，在这种特殊情况下使用 `.unwrap()` 存在 bug。这种有 bug 的输入本不应该引发 panic。

话又说回来，确实会出现 `Result` 值不可能是错误的情况。例如，在第 18 章中，你会看到 `Write` 特型为文本和二进制输出定义了一组泛型方法（`.write()` 等）。所有这些方法都会返回 `io::Result`，但如果你碰巧正在写入 `Vec<u8>`，那么它们就不可能失败。在这种情况下，可以使用 `.unwrap()` 或 `.expect(message)` 来简化 `Result` 的处理。

当错误表明情况相当严重或异乎寻常，理当用 `panic` 对它进行处理时，这些方法也很有用：

```
fn print_file_age(filename: &Path, last_modified: SystemTime) {
    let age = last_modified.elapsed().expect("system clock
drift");
```

```
...  
}
```

在这里，仅当系统时间早于文件创建时间时，`.elapsed()` 方法才会失败。如果文件是最近创建的，并且在程序运行期间系统时钟往回调整过，就会发生这种情况。根据这段代码的使用方式，在这种情况下，调用 `panic` 是一个合理的选择，而不必处理该错误或将该错误传播给调用者。

7.2.7 忽略错误

有时我们只想完全忽略一个错误。例如，在 `print_error()` 函数中，我们必须处理打印一个错误时会触发另一个错误等罕见情况。如果将 `stderr` 通过管道传给另一个进程，而那个进程已终止，就可能发生这种情况。因为要报告的原始错误可能更值得传播，所以我们只想忽略 `stderr` 带来的这些小麻烦，但 Rust 编译器会警告你有未使用的 `Result` 值：

```
writeln!(stderr(), "error: {}", err); // 警告：未使用的结果
```

惯用法 `let _ = ...` 可用来消除这种警告。

```
let _ = writeln!(stderr(), "error: {}", err); // 正确，忽略结果
```

7.2.8 处理 `main()` 中的错误

在大多数生成 `Result` 的地方，让错误冒泡到调用者通常是正确的行为。这就是为什么？在 Rust 中会设计成单字符语法。正如我们所见，在某些程序中，它曾连续用于多行代码。

但是，如果你传播错误的距离足够远，那么最终它就会抵达 `main()`，后者必须对其进行处理。通常来说，`main()` 不能使用 `?`，因为它的返回类型不是 `Result`：

```
fn main() {
    calculate_tides()?; // 错误: main()无法再推卸责任了
}
```

处理 `main()` 中错误的最简单方式是使用 `.expect()`:

```
fn main() {
    calculate_tides().expect("error"); // 责任止于此
}
```

如果 `calculate_tides()` 返回错误结果, 那么 `.expect()` 方法就会 panic。主线程中的 panic 会打印出一条错误消息, 然后以非零的退出码退出, 大体上, 这就是我们期望的行为。在一般的小型程序中我们都是这样做的。这是一个开始。

不过, 错误消息有点儿吓人:

```
$ tidecalc --planet mercury
thread 'main' panicked at 'error: "moon not found"',
src/main.rs:2:23
note: run with `RUST_BACKTRACE=1` environment variable to display
a backtrace
```

错误消息淹没在噪声中。此外, 在这种特殊情况下, `RUST_BACKTRACE=1` 是一个糟糕的建议。

但是, 也可以更改 `main()` 的类型签名以返回 `Result` 类型, 这样就可以使用 `?` 了:

```
fn main() -> Result<(), TideCalcError> {
    let tides = calculate_tides()?;
    print_tides(tides);
    Ok(())
}
```

这适用于任何能用 `{:?}` 格式说明符打印的错误类型, 也就是说, 所有标准错误类型 (如 `std::io::Error`) 都适用。这种技巧易于使用, 并能提供更好的错误消息, 但它并不理想:

```
$ tidecalc --planet mercury
Error: TideCalcError { error_type: NoMoon, message: "moon not found" }
```

如果你有更复杂的错误类型或想要在消息中包含更多信息，那么可以自己打印错误消息：

```
fn main() {
    if let Err(err) = calculate_tides() {
        print_error(&err);
        std::process::exit(1);
    }
}
```

上述代码使用了 `if let` 表达式，以便仅在对 `calculate_tides()` 的调用返回错误结果时才打印错误消息。有关 `if let` 表达式的详细信息，请参阅第 10 章。`print_error` 函数在 7.2.3 节中介绍过。

现在的输出就相当整洁了。

```
$ tidecalc --planet mercury
error: moon not found
```

7.2.9 声明自定义错误类型

假设你正在编写一个新的 JSON 解析器，并且希望它有自己的错误类型。（到目前为止，本书尚未介绍用户定义类型，后面我们将用几章的篇幅进行介绍。声明错误类型很简单，我们先在此处大概预览一下。）

要编写的代码大概也就下面几行：

```
// json/src/error.rs

#[derive(Debug, Clone)]
pub struct JsonError {
    pub message: String,
    pub line: usize,
```

```
pub column: usize,  
}
```

这个结构体叫作 `json::error::JsonError`。当你想引发这种类型的错误时，可以像下面这样写：

```
return Err(JsonError {  
    message: "expected ']' at end of array".to_string(),  
    line: current_line,  
    column: current_column  
});
```

这没什么问题。但是，如果你希望达到你的库用户的预期，确保这个错误类型像标准错误类型一样工作，那么还有一点儿额外的工作要做：

```
use std::fmt;  
  
// 错误应该是可打印的  
impl fmt::Display for JsonError {  
    fn fmt(&self, f: &mut fmt::Formatter) -> Result<(),  
    fmt::Error> {  
        write!(f, "{} ({}:{})", self.message, self.line,  
self.column)  
    }  
}  
  
// 错误应该实现std::error::Error特型，但使用Error各个方法的默认定义就够了  
impl std::error::Error for JsonError { }
```

`impl` 关键字、`self` 和其他所有关键字的含义在后面的几章中会进行解释。

与 Rust 语言的许多方面一样，各种 `crate` 的存在是为了让错误处理更容易、更简洁。`crate` 种类繁多，但最常用的一个是 `thiserror`，它会帮你完成之前的所有工作，让你像下面这样编写错误定义：

```
use thiserror::Error;  
  
#[derive(Error, Debug)]
```



```
#[error("{message:} ({line:}, {column})")]
pub struct JsonError {
    message: String,
    line: usize,
    column: usize,
}
```

`#[derive(Error)]` 指令会让 `thiserror` 生成前面展示过的代码，这可以节省大量的时间和精力。

7.2.10 为什么是 **Result**

现在我们已经足够了解为何 Rust 会优先选择 `Result` 而非异常了。以下是此设计的几个要点。

- Rust 要求程序员在每个可能发生错误的地方做出某种决策，并将其记录在代码中。这样做很好，否则容易因为疏忽而无法正确处理错误。
- 最常见的决策是让错误继续传播，而这用单个字符 `?` 就可以实现。因此，错误处理管道不会像在 C 和 Go 中那样让你的代码混乱不堪，而且它还具有可见性：在浏览一段代码时，你一眼就能看出错误是从哪里传出来的。
- 是否可能出错是每个函数的返回类型的一部分，因此哪些函数会失败、哪些不会失败非常清晰。如果你将一个函数改为可能出错的，那么就要同时更改它的返回类型，而编译器会让你随之修改该函数的各个下游使用者。
- Rust 会检查 `Result` 值是否被用过了，这样你就不会意外地让错误悄悄溜过去（C 中的常见失误）。
- 由于 `Result` 是一种与任何其他数据类型没有本质区别的数据类型，因此很容易将成功结果和错误结果存储在同一个集合中，也很容易对“部分成功”的情况进行模拟。如果你正在编写一个从文本文件加载数百万条记录的程序，并且需要一种方法来处理大多数时候会成功但偶尔也会失败的可能结果，就可以用向量 `Result` 在内存中表达出现这种结果时的情形。

这样设计的代价是，你会发现自己在 Rust 中要比在其他语言中做更多的思考和工程化的错误处理。与许多其他领域一样，Rust 在错误处理方面比你所习惯的要严格一些。但对系统编程来说，这绝对是值得的。

第 8 章 crate 与模块

这是 Rust 主题下的小小理念之一：系统程序员也能享受美好。

——Robert O’Callahan, “Random Thoughts on Rust:
crates.io and IDEs”

假设你正在编写一个程序，用来模拟蕨类植物从单个细胞开始的生长过程。你的程序，也像蕨类植物一样，一开始会非常简单，所有的代码也许都在一个文件中——这只是想法的萌芽。随着成长，它将开始分化出内部结构，不同的部分会有不同的用途。然后它将分化成多个文件，可能遍布在各个目录中。随着时间的推移，它可能会成为整个软件生态系统的重要组成部分。对于任何超出几个数据结构或几百行代码的程序，进行适当的组织都是很有必要的。

本章介绍了 Rust 的一些特性（crate 与模块），这些特性有助于你的程序保持井井有条。我们还将涵盖其他与 Rust crate 的结构和分发有关的主题，包括如何记录与测试 Rust 代码、如何消除不必要的编译器警告、如何使用 Cargo 管理项目依赖项和版本控制、如何在 Rust 的公共 crate 存储库 crates.io 上发布开源库、Rust 如何通过语言版本进行演进等。本章将使用蕨类模拟器作为运行示例。

8.1 crate

Rust 程序由 crate（板条箱）组成。每个 crate 都是既完整又内聚的单元，包括单个库或可执行程序的所有源代码，以及任何相关的测试、示例、工具、配置和其他杂项。对于蕨类植物模拟器，我们可以使用第三方库完成 3D 图形、生物信息学、并行计算等工作。这些库以 crate 的形式分发，如图 8-1 所示。

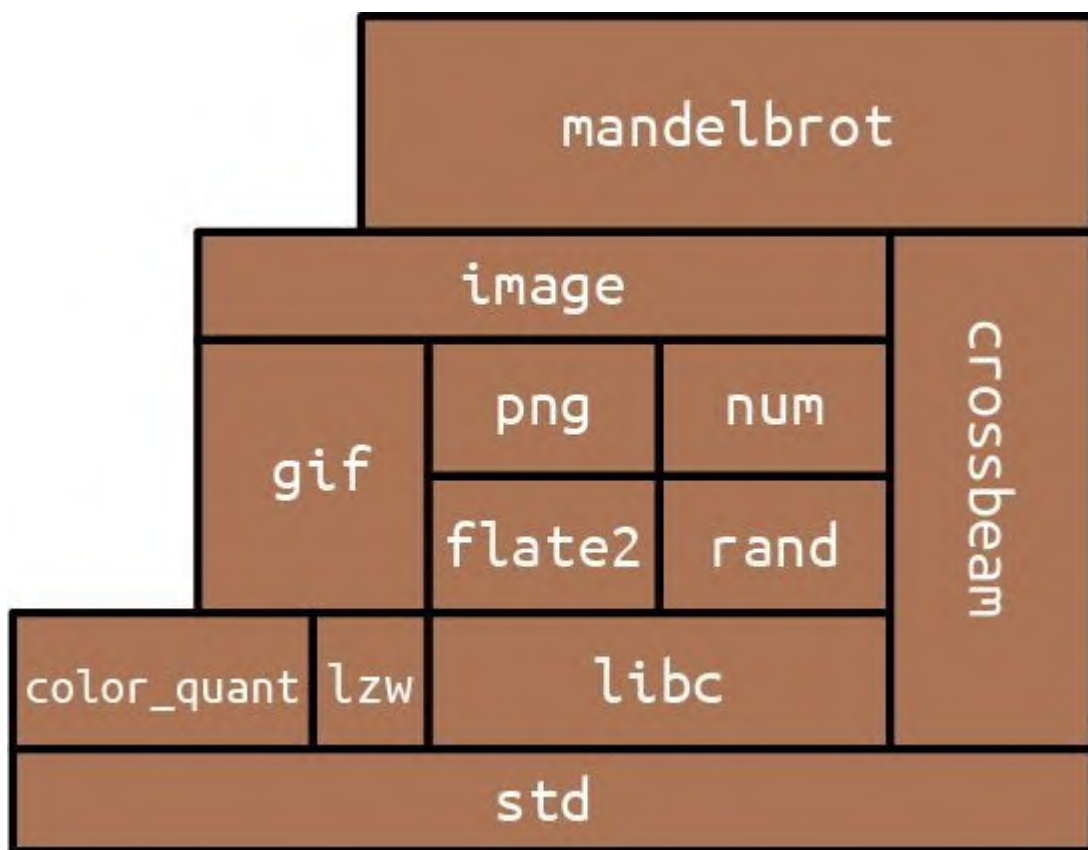


图 8-1：一个 crate 及其依赖

弄清楚 crate 是什么以及它们如何协同工作的最简单途径是，使用带有 `--verbose` 标志的 `cargo build` 来构建具有某些依赖项的现有项目。我们以 2.6.6 节的“并发曼德博程序”为例来执行此操作。结果如下所示：

```
$ cd mandelbrot
$ cargo clean      # 删除之前编译的代码
$ cargo build --verbose
    Updating registry `https://github.com/rust-lang/crates.io-index`
  Downloading autocfg v1.0.0
  Downloading semver-parser v0.7.0
  Downloading gif v0.9.0
  Downloading png v0.7.0

... (downloading and compiling many more crates)

Compiling jpeg-decoder v0.1.18
  Running `rustc
    --crate-name jpeg_decoder
    --crate-type lib
    ...
    --extern byteorder=.../libbyteorder-29efdd0b59c6f920.rmeta
    ...
  Compiling image v0.13.0
  Running `rustc
    --crate-name image
    --crate-type lib
    ...
    --extern byteorder=.../libbyteorder-29efdd0b59c6f920.rmeta
    --extern gif=.../libgif-a7006d35f1b58927.rmeta
    --extern jpeg_decoder=.../libjpeg_decoder-
5c10558d0d57d300.rmeta
  Compiling mandelbrot v0.1.0 (/tmp/rustbook-test-
files/mandelbrot)
  Running `rustc
    --edition=2021
    --crate-name mandelbrot
    --crate-type bin
    ...
    --extern crossbeam=.../libcrossbeam-f87b4b3d3284acc2.rlib
    --extern image=.../libimage-b5737c12bd641c43.rlib
    --extern num=.../libnum-1974e9a1dc582ba7.rlib -C link-
arg=-fuse-ld=lld`
  Finished dev [unoptimized + debuginfo] target(s) in 16.94s
$
```

这里重新格式化了 `rustc` 命令行以提高可读性，我们删除了很多与本次讨论无关的编译器选项，然后将它们替换成了省略号（...）。

你可能还记得，当“并发曼德博程序”的例子完成时，曼德博程序的 `main.rs` 包含几个来自其他 crate 的语法项的 `use` 声明：

```
use num::Complex;
// ...
use image::ColorType;
use image::png::PNGEncoder;
```

并且 `Cargo.toml` 文件中指定了我们想要的每个 crate 的版本：

```
[dependencies]
num = "0.4"
image = "0.13"
crossbeam = "0.8"
```

单词 `dependencies` 在这里是指这个项目使用的其他 crate，也就是我们所依赖的代码。我们在 `crates.io`（Rust 社区的开源 crate 站点）上找到了这些 crate。例如，通过访问 `crates.io` 我们搜索并找到了 `image` 库。`crates.io` 上的每个 crate 页面都显示了其 `README.md` 文件及文档和源代码的链接，以及像 `image = "0.13"` 这样的可以复制并添加到 `Cargo.toml` 中的配置行。这里展示的版本号只是我们编写本程序时这 3 个包的最新版本。

Cargo 清单讲述了如何使用这些信息。运行 `cargo build` 时，Cargo 首先会从 `crates.io` 下载这些 crate 的指定版本的源代码。然后，它会读取这些 crate 的 `Cargo.toml` 文件、下载它们的依赖项，并递归地进行。例如，0.13.0 版的 `image` crate 的源代码包含一个 `Cargo.toml` 文件，其中列出了如下内容：

```
[dependencies]
byteorder = "1.0.0"
num-iter = "0.1.32"
num-rational = "0.1.32"
num-traits = "0.1.32"
enum_primitive = "0.1.0"
```

看到这个文件，Cargo 就知道在它使用 `image` 之前，必须先获取这些 crate。稍后你将看到如何要求 Cargo 从 Git 存储库或本地文件

系统而非 crates.io 获取源代码。

由于 mandelbrot 通过使用 image crate 间接依赖于这些 crate，因此我们称它们为 mandelbrot 的传递依赖。所有这些依赖关系的集合，会告诉 Cargo 关于要构建什么 crate 以及应该按什么顺序构建的全部知识，这叫作该 crate 的依赖图。Cargo 对依赖图和传递依赖的自动处理可以显著节省程序员的时间和精力。

一旦有了源代码，Cargo 就会编译所有的 crate。它会为项目依赖图中的每个 crate 都运行一次 rustc (Rust 编译器)。编译库时，Cargo 会使用 `--crate-type lib` 选项。这会告诉 rustc 不要寻找 `main()` 函数，而是生成一个 `.rlib` 文件，其中包含一些已编译代码，可用于创建二进制文件和其他 `.rlib` 文件。

编译程序时，Cargo 会使用 `--crate-type bin`，结果是目标平台的二进制可执行文件，比如 Windows 上的 `mandelbrot.exe`。

对于每个 rustc 命令，Cargo 都会传入 `--extern` 选项，给出 crate 将使用的每个库的文件名。这样，当 rustc 看到一行代码（如 `use image::png::PNGEncoder`）时，就可以确定 image 是另一个 crate 的名称。感谢 Cargo，它知道在哪里可以找到磁盘上已编译的 crate。Rust 编译器需要访问这些 `.rlib` 文件，因为它们包含库的已编译代码。Rust 会将代码静态链接到最终的可执行文件中。`.rlib` 也包含一些类型信息，这样 Rust 就可以检查我们在代码中使用的库特性是否确实存在于 crate 中，以及我们是否正确使用了它们。`.rlib` 文件中还包含此 crate 的公共内联函数、泛型和宏这三者的副本，在 Rust 知道我们将如何使用它们之前，这些特性无法完全编译为机器码。

`cargo build` 支持各种选项，其中大部分超出了本书的范畴，我们在这里只提一个：`cargo build --release` 会生成优化过的程序。这种程序运行得更快，但它们的编译时间更长、运行期不会检查整数溢出、会跳过 `debug_assert!()` 断言，并且在 panic 时生成的调用栈跟踪通常不太可靠。

8.1.1 版本

Rust 具有极强的兼容性保证。任何能在 Rust 1.0 上编译的代码在 Rust 1.50 或 Rust 1.900（如果已经发布的话）上都必然编译得一样好。

但有时会有一些必要的扩展提议加入语言中，这有可能导致旧代码无法再编译。例如，经过多次讨论，Rust 确定了一种支持异步编程的语法，该语法将标识符 `async` 和 `await` 重新用作关键字（参见第 20 章）。但是这种语言更改会破坏任何使用 `async` 或 `await` 作为变量名的现有代码。

为了在不破坏现有代码的情况下继续演进，Rust 会使用版本。2015 版 Rust 会与 Rust 1.0 兼容。2018 版 Rust 将 `async` 和 `await` 改为关键字并精简了模块系统，而 2021 版 Rust 则提升了数组对人类的友好性，并默认让一些广泛使用的库定义随处可用。虽然这些都是对该语言的重要改进，但会破坏现有代码。为避免这种情况，每个 crate 都在其 `Cargo.toml` 文件顶部的 `[package]` 部分使用下面这样的行来表明自己是用哪个版本的 Rust 编写的：

```
edition = "2021"
```

如果该关键字不存在，则假定为 2015 版，因此旧 crate 根本不必更改。但是如果你想使用异步函数或新的模块系统，就要在 `Cargo.toml` 文件中添加一句“`edition = "2018"` 或更高版本”。

Rust 承诺编译器将始终接受该语言的所有现存版本，并且程序可以自由混用以不同版本编写的 crate。2015 版的 crate 甚至可以依赖 2021 版的 crate。换句话说，crate 的版本只影响其源代码的解释方式，编译代码时，版本的差异已然消失。这意味着你无须为了继续参与到现代 Rust 生态系统中而更新旧的 crate。同样，你也不必为了避免给用户带来不便而被迫使用旧的 crate。当你想在自己的代码中使用新的语言特性时，只要更改版本就可以了。

Rust 项目组不会每年都发布新版本，只有认为确有必要时才会发布。例如，没有 2020 版 Rust。将 `edition` 设置为 `"2020"` 会导致错

误。Rust 版本指南涵盖了每个版本中引入的更改，并提供了有关版本体系的完善的背景知识。

使用最新版本几乎总是更好的做法，尤其是对于新代码。cargo new 默认会在最新版本上创建新项目。本书全程使用 2021 版。

如果你有一个用旧版本的 Rust 编写的 crate，则 cargo fix 命令能帮助你自动将代码升级到新版本。Rust 版本指南详细解释了 cargo fix 命令。

8.1.2 创建配置文件

你可以在 Cargo.toml 文件中放置几个配置设定区段，这些设定会影响 cargo 生成的 rustc 命令行，如表 8-1 所示。

表 8-1: Cargo.toml 的配置设定区段

命令行	使用的 Cargo.toml 区段
cargo build	[profile.dev]
cargo build --release	[profile.release]
cargo test	[profile.test]

通常情况下，默认设置是可以使用的，但当你想要使用剖析器（一种用来测量程序在哪些地方消耗了 CPU 时间的工具）时会出现例外情况。要从剖析器中获得最佳数据，需要同时启用优化（通常仅在发布构建中启用）和调试符号（通常仅在调试构建中启用）这两个选项。要同时启用它们，请将如下代码添加到你的 Cargo.toml 中：

```
[profile.release]
debug = true # 在release构建中启用debug符号
```

debug 设定会控制 `rustc` 的 `-g` 选项。通过这个配置，当你键入 `cargo build --release` 时，将获得带有调试符号的二进制文件。而优化设置未受影响。

Cargo 文档中列出了可以在 `Cargo.toml` 中调整的许多其他设定。

8.2 模块

`crate` 是关于项目间代码共享的，而**模块**是关于项目内代码组织的。它们扮演着 Rust 命名空间的角色，是构成 Rust 程序或库的函数、类型、常量等的容器。一个模块看起来是这样的：

```
mod spores {
    use cells::{Cell, Gene};

    /// 由成年蕨类植物产生的细胞。作为蕨类植物生命周期的一部分，细胞会随风
    /// 传播。一个孢子会长成原叶体（一个完整的独立有机体，最大直径达5毫米），
    /// 原叶体产生的受精卵会长成新的蕨类植物（植物的性别很复杂）
    pub struct Spore {
        ...
    }

    /// 模拟减数分裂产生孢子
    pub fn produce_spore(factory: &mut Sporangium) -> Spore {
        ...
    }

    /// 提取特定孢子中的基因
    pub(crate) fn genes(spore: &Spore) -> Vec<Gene> {
        ...
    }

    /// 混合基因以准备减数分裂（细胞分裂间期的一部分）
    fn recombine(parent: &mut Cell) {
        ...
    }

    ...
}
```

模块是一组语法项的集合，这些语法项具有命名的特性，比如此示例中的 `Spore` 结构体和 3 个函数。`pub` 关键字会使某个语法项声明为公共项，这样它就可以从模块外部访问了。

如果把一个函数标记为 `pub(crate)`，那么就意味着它可以在这个 `crate` 中的任何地方使用，但不会作为外部接口的一部分公开。它不能被其他 `crate` 使用，也不会出现在这个 `crate` 的文档中。

任何未标记为 `pub` 的内容都是私有的，只能在定义它的模块及其任意子模块中使用：

```
let s = spores::produce_spore(&mut factory); // 正确
spores::recombine(&mut cell); // 错误: `recombine`是私有的
```

将某个语法项标记为 `pub` 通常称为“导出”该语法项。

本节的其余部分涵盖了要想充分利用模块所需了解的详细信息。

- 我们会展示如何嵌套模块并在需要时将它们分散到不同的文件和目录中。
- 我们会解释 Rust 使用的路径语法，以引用来自其他模块的语法项，并展示如何导入这些语法项，以便你使用它们而不必写出其完整路径。
- 我们会触及 Rust 对结构体字段的细粒度控制。
- 我们会介绍预导入（prelude，原意为“序曲”）模块，它们通过收集几乎所有用户都需要的常用导入，减少了样板代码的编写。
- 为了提高代码的清晰性和一致性，我们还会介绍常量和静态变量这两种定义命名值的方法。

8.2.1 嵌套模块

模块可以嵌套，通常可以看到某个模块仅仅是一组子模块集合：

```
mod plant_structures {
    pub mod roots {
        ...
    }
    pub mod stems {
        ...
    }
    pub mod leaves {
        ...
    }
}
```

如果你希望嵌套模块中的语法项对其他 crate 可见，请务必将它和它所在的模块标记为公开的。否则可能会看到这样的警告：

```
warning: function is never used: `is_square`
  |
23 | /           pub fn is_square(root: &Root) -> bool {
24 | |           root.cross_section_shape().is_square()
25 | |           }
   | |_____^
```

也许这个函数目前确实是无用的代码。但是，如果你打算在其他 crate 中使用它，那么 Rust 就会提示你这个函数对它们实际上是不可见的。你应该确保它所在的模块也是 pub 形式。

也可以指定 pub(super)，让语法项只对其父模块可见。还可以指定 pub(in <path>)，让语法项在特定的父模块及其后代中可见。这对于深度嵌套的模块特别有用：

```
mod plant_structures {
    pub mod roots {
        pub mod products {
            pub(in crate::plant_structures::roots) struct
Cytokinin {
                ...
            }
        }

        use products::Cytokinin; // 正确：在`roots`模块中可见
    }

    use roots::products::Cytokinin; // 错误：`Cytokinin`是私有的
}

// 错误：`Cytokinin`是私有的
use plant_structures::roots::products::Cytokinin;
```

通过这种方式，我们可以写出一个完整的程序，把大量代码和完整的模块层次结构以我们想要的任何方式关联起来，并放在同一个源文件中。

但实际上，以这种方式写代码相当痛苦，因此还有另一种选择。

8.2.2 单独文件中的模块

模块还可以这样写：

```
mod spores;
```

前面我们一直把 `spores` 模块的主体代码包裹在花括号中。在这里，我们告诉 Rust 编译器，`spores` 模块保存在一个单独的名为 `spores.rs` 的文件中：

```
// spores.rs

/// 由成年蕨类植物产生的细胞.....
pub struct Spore {
    ...
}

/// 模拟减数分裂产生孢子
pub fn produce_spore(factory: &mut Sporangium) -> Spore {
    ...
}

/// 提取特定孢子中的基因
pub(crate) fn genes(spore: &Spore) -> Vec<Gene> {
    ...
}

/// 混合基因以准备减数分裂（细胞分裂间期的一部分）
fn recombine(parent: &mut Cell) {
    ...
}
```

`spores.rs` 仅包含构成该模块的那些语法项，它不需要任何样板代码来声明自己是一个模块。

代码的位置是这个 `spores` 模块与 8.2.1 节中展示的版本之间的唯一区别。Rust 遵循同样的规则，以决定什么是公共的以及什么是私有

的。而且即便模块在单独的文件中，Rust 也永远不会分别编译它们，因为只要你构建 Rust crate，就会重新编译它的所有模块。

模块可以有自己的目录。当 Rust 看到 `mod spore;` 时，会同时检查 `spores.rs` 和 `spores/mod.rs`，如果两个文件都不存在，或者都存在，就会报错。对于这个例子，我们使用了 `spores.rs`，因为 `spores` 模块没有任何子模块。但是考虑一下我们之前编写的 `plant_structures` 模块。如果将该模块及其 3 个子模块拆分到它们自己的文件中，则会生成如下项目：

```
fern_sim/
├── Cargo.toml
└── src/
    ├── main.rs
    ├── spores.rs
    └── plant_structures/
        ├── mod.rs
        ├── leaves.rs
        ├── roots.rs
        └── stems.rs
```

在 `main.rs` 中，我们声明了 `plant_structures` 模块：

```
pub mod plant_structures;
```

这会导致 Rust 加载 `plant_structures/mod.rs`，该文件声明了 3 个子模块：

```
// 在plant_structures/mod.rs中
pub mod roots;
pub mod stems;
pub mod leaves;
```

这 3 个模块的内容存储在 `leaves.rs`、`roots.rs` 和 `stems.rs` 这 3 个单独的文件中，与 `mod.rs` 一样位于 `plant_structures` 目录下。

也可以使用同名的文件和目录来组成模块。如果 `stems`（茎）需要包含称为 `xylem`（木质部）和 `phloem`（韧皮部）的模块，那么可以

选择将 `stems` 保留在 `plant_structures/stems.rs` 中并添加一个 `stems` 目录：

```
fern_sim/
├── Cargo.toml
├── src/
│   ├── main.rs
│   ├── spores.rs
│   └── plant_structures/
│       ├── mod.rs
│       ├── leaves.rs
│       ├── roots.rs
│       ├── stems/
│       │   ├── phloem.rs
│       │   └── xylem.rs
│       └── stems.rs
```

然后，在 `stems.rs` 中，我们声明了两个新的子模块：

```
// 在plant_structures/stems.rs中
pub mod xylem;
pub mod phloem;
```

这 3 种选项（模块位于自己的文件中、模块位于自己的带有 `mod.rs` 的目录中，以及模块在自己的文件中，并带有包含子模块的补充目录）为模块系统提供了足够的灵活性，以支持你可能用到的几乎任何项目结构。

8.2.3 路径与导入

`::` 运算符用于访问模块中的各项特性。项目中任何位置的代码都可以通过写出其路径来引用标准库特性：

```
if s1 > s2 {
    std::mem::swap(&mut s1, &mut s2);
}
```

`std` 是标准库的名称。路径 `std` 指的是标准库的顶层模块。
`std::mem` 是标准库中的子模块，而 `std::mem::swap` 是该模块

中的公共函数。

可以用这种方式编写所有代码：如果你想要一个圆或字典，就明确写出 `std::f64::consts::PI` 或 `std::collections::HashMap::new`。但这样做会很烦琐并且难以阅读。另一种方法是将这些特性导入使用它们的模块中：

```
use std::mem;

if s1 > s2 {
    mem::swap(&mut s1, &mut s2);
}
```

这条 `use` 声明导致名称 `mem` 在整个封闭块或模块中成了 `std::mem` 的本地别名。

可以通过写 `use std::mem::swap;` 来导入 `swap` 函数本身，而不是 `mem` 模块。然而，我们之前的编写风格通常被认为是最好的：导入类型、特型和模块（如 `std::mem`），然后使用相对路径访问其中的函数、常量和成员。

可以一次导入多个名称：

```
use std::collections::{HashMap, HashSet}; // 同时导入两个模块

use std::fs::{self, File}; // 同时导入`std::fs`和`std::fs::File`

use std::io::prelude::*; // 导入所有语法项
```

上述代码只是对“明确写出所有单独导入”的简写：

```
use std::collections::HashMap;
use std::collections::HashSet;

use std::fs;
use std::fs::File;

// std::io::prelude中的全部公开语法项：
use std::io::prelude::Read;
use std::io::prelude::Write;
```

```
use std::io::prelude::BufRead;
use std::io::prelude::Seek;
```

可以使用 `as` 导入一个语法项，但在本地赋予它一个不同的名称：

```
use std::io::Result as IOResult;

// 这个返回类型只是`std::io::Result<()>`的另一种写法：
fn save_spore(spore: &Spore) -> IOResult<()>
...
```

模块不会自动从其父模块继承名称。假设 `proteins/mod.rs` 中有如下代码：

```
// proteins/mod.rs
pub enum AminoAcid { ... }
pub mod synthesis;
```

那么 `synthesis.rs` 中的代码不会自动“看到”类型 ``AminoAcid``：

```
// proteins/synthesis.rs
pub fn synthesize(seq: &[AminoAcid]) // 错误：找不到类型AminoAcid
...
```

其实，每个模块都会以“白板”开头，并且必须导入它使用的名称：

```
// proteins/synthesis.rs
use super::AminoAcid; // 从父模块显式导入

pub fn synthesize(seq: &[AminoAcid]) // 正确
...
```

默认情况下，路径是相对于当前模块的：

```
// in proteins/mod.rs

// 从某个子模块导入
use synthesis::synthesize;
```

`self` 也是当前模块的同义词，所以可以这样写：

```
// in proteins/mod.rs

// 从枚举中导入名称，因此可以把赖氨酸写成`Lys`，而不是`AminoAcid::Lys`
use self::AminoAcid::*;
```

或者简单地写成如下形式。

```
// 在proteins/mod.rs中

use AminoAcid::*;
```

（当然，这里的 `AminoAcid` 示例偏离了之前提到过的仅导入类型、特型和模块的样式规则。如果我们的程序中包含长氨基酸序列，那么这种调整就符合奥威尔第六规则：“为了表达准确，宁可打破上述规则”。）

关键字 `super` 和 `crate` 在路径中有着特殊的含义：`super` 指的是父模块，`crate` 指的是当前模块所在的 `crate`。

使用相对于 `crate` 根而不是当前模块的路径可以更容易地在项目中移动代码，因为如果当前模块的路径发生了变化，则不会破坏任何导入。例如，我们可以使用 `crate` 编写 `synthesis.rs`：

```
// proteins/synthesis.rs
use crate::proteins::AminoAcid; // 显式导入相对于crate根路径的语法项

pub fn synthesize(seq: &[AminoAcid]) // 正确
    ...
```

子模块可以使用 `use super::*` 访问其父模块中的私有语法项。

如果有一个与你正使用的 `crate` 同名的模块，那么引用它们的内容时有一些注意事项。如果你的程序在其 `Cargo.toml` 文件中将 `image` `crate` 列为依赖项，但还有另一个名为 `image` 的模块，那么以 `image` 开头的路径就是有歧义的：

```
mod image {
    pub struct Sampler {
        ...
    }
}

// 错误：它引用的是我们的`image`模块还是`image crate`？
use image::Pixels;
```

即使 `image` 模块中没有 `Pixels` 类型，这种歧义仍然是有问题的：如果模块中稍后添加了这样的定义，则可能会默默地改变程序中其他地方引用到的路径，而这将给人带来困扰。

为了解决歧义，Rust 有一种特殊的路径，称为**绝对路径**，该路径以 `::` 开头，总会引用外部 `crate`。要引用 `image crate` 中的 `Pixels` 类型，可以这样写：

```
use ::image::Pixels;           // `image crate`中的`Pixels`
```

要引用你自己模块中的 `Sampler` 类型，可以这样写：

```
use self::image::Sampler;     // `image`模块中的`Sampler`
```

模块与文件不是一回事，但模块与 Unix 文件系统中的目录和文件有些相似之处。`use` 关键字会创建别名，就像用 `ln` 命令创建链接一样。路径和文件名一样，有绝对和相对两种形式。`self` 和 `super` 类似于 `.` 和 `..` 这样的特殊目录。

8.2.4 标准库预导入

我们刚才说过，就导入的名称而言，每个模块都以“白板”开头。但这个“白板”并不是完全空白的。

一方面，标准库 `std` 会自动链接到每个项目。这意味着你始终可以使用 `use std::whatever`，或者就按名称引用 `std` 中的语法项，比如代码中内联的 `std::mem::swap()`。另一方面，还有一些特别的便捷名称（如 `Vec` 和 `Result`）会包含在标准库预导入中并自动

导入。Rust 的行为就好像每个模块（包括根模块）都用以下导入语句开头一样：

```
use std::prelude::v1::*;
```

标准库预导入包含几十个常用的特型和类型。

我们在第 2 章中提到的那些库有时会提供一些名为 `prelude`（预导入）的模块。但 `std::prelude::v1` 是唯一会自动导入的预导入。把一个模块命名为 `prelude` 只是一种约定，旨在告诉用户应该使用 `*` 导入它。

8.2.5 公开 `use` 声明

虽然 `use` 声明只是个别名，但也可以公开它们：

```
// 在plant_structures/mod.rs中
...
pub use self::leaves::Leaf;
pub use self::roots::Root;
```

这意味着 `Leaf` 和 `Root` 是 `plant_structures` 模块的公共语法项。它们还是 `plant_structures::leaves::Leaf` 和 `plant_structures::roots::Root` 的简单别名。

标准库预导入就是像这样编写的一系列 `pub` 导入。

8.2.6 公开结构体字段

模块可以包含用户定义的一些结构体类型（使用 `struct` 关键字引入）。第 9 章会详细介绍这些内容，但现在可以简单讲讲模块与结构体字段的可见性之间是如何相互作用的。

一个简单的结构体如下所示：

```
pub struct Fern {  
    pub roots: RootSet,  
    pub stems: StemSet  
}
```

结构体的字段，甚至是私有字段，都可以在声明该结构体的整个模块及其子模块中访问。在模块之外，只能访问公共字段。

事实证明，通过模块而不是像 Java 或 C++ 那样通过类来强制执行访问控制对软件设计非常有帮助。它减少了样板“getter”方法和“setter”方法，并且在很大程度上消除了对诸如 C++ friend（友元）声明等语法的需求。单个模块可以定义多个紧密协作的类型，例如 `frond::LeafMap` 和 `frond::LeafMapIter`，它们可以根据需要访问彼此的私有字段，同时仍然对程序的其余部分隐藏这些实现细节。

8.2.7 静态变量与常量

除了函数、类型和嵌套模块，模块还可以定义常量和静态变量。

关键字 `const` 用于引入常量，其语法和 `let` 一样，只是它可以标记为 `pub`，并且必须写明类型。此外，常量的命名规约是

`UPPERCASE_NAMES`:

```
pub const ROOM_TEMPERATURE: f64 = 20.0; // 摄氏度
```

`static` 关键字引入了一个静态语法项，跟常量几乎是一回事：

```
pub static ROOM_TEMPERATURE: f64 = 68.0; // 华氏度
```

常量有点儿像 C++ 的 `#define`：该值在每个使用了它的地方都会编译到你的代码中。静态变量是在程序开始运行之前设置并持续到程序退出的变量。在代码中对魔数和字符串要使用常量，而在处理大量的数据或需要借用常量值的引用时则要使用静态变量。

没有 `mut` 常量。静态变量可以标记为 `mut`，但正如第 5 章所述，Rust 没有办法强制执行其关于 `mut` 静态变量的独占访问规则。因此，`mut` 静态变量本质上是非线程安全的，安全代码根本不能使用它们：

```
static mut PACKETS_SERVED: usize = 0;

println!("{}", served, PACKETS_SERVED); // 错误：使用了mut静态变量
```

Rust 不鼓励使用全局可变状态。有关备选方案的讨论，请参阅 19.3.11 节。

8.3 将程序变成库

随着蕨类植物模拟器成功运行，你会发现你所需要的不仅仅是单个程序。假设你有一个运行此模拟并将结果保存在文件中的命令程序。现在，你想编写其他程序对这些保存下来的结果进行科学分析、实时显示正在生长的植物的 3D 渲染图、渲染足以乱真的图片，等等。所有这些程序都需要共享基本的蕨类植物模拟代码。这时候你应该建立一个库。

第一步是将现有的项目分为两部分：一个库 crate，其中包含所有共享代码；一个可执行文件，其中只包含你现在的命令程序才需要的代码。

为了展示如何做到这一点，要使用一个极度简化的示例程序：

```
struct Fern {
    size: f64,
    growth_rate: f64
}

impl Fern {
    /// 模拟一株蕨类植物在一天内的生长
    fn grow(&mut self) {
        self.size *= 1.0 + self.growth_rate;
    }
}

/// 执行days天内某株蕨类植物的模拟
fn run_simulation(fern: &mut Fern, days: usize) {
    for _ in 0 .. days {
        fern.grow();
    }
}

fn main() {
    let mut fern = Fern {
        size: 1.0,
        growth_rate: 0.001
    };
    run_simulation(&mut fern, 1000);
}
```



```
println!("final fern size: {}", fern.size);
}
```

假设这个程序有一个简单的 Cargo.toml 文件。

```
[package]
name = "fern_sim"
version = "0.1.0"
authors = ["You <you@example.com>"]
edition = "2021"
```

很容易将这个程序变成库，步骤如下。

01. 将文件 src/main.rs 重命名为 src/lib.rs。
02. 将 pub 关键字添加到 src/lib.rs 中的语法项上，这些语法项将成为这个库的公共特性。
03. 将 main 函数移动到某个临时文件中。（我们暂时不同管它。）

生成的 src/lib.rs 文件如下所示：

```
pub struct Fern {
    pub size: f64,
    pub growth_rate: f64
}

impl Fern {
    /// 模拟一株蕨类植物在一天内的生长
    pub fn grow(&mut self) {
        self.size *= 1.0 + self.growth_rate;
    }
}

/// 执行days天内某株蕨类植物的模拟
pub fn run_simulation(fern: &mut Fern, days: usize) {
    for _ in 0 .. days {
        fern.grow();
    }
}
```

请注意，不需要更改 Cargo.toml 中的任何内容。这是因为这个最小化的 Cargo.toml 文件只是为了让 Cargo 保持默认行为而已。默认设定下，cargo build 会查看源目录中的文件并根据文件名确定要构

建的内容。当它发现存在文件 `src/lib.rs` 时，就知道要构建一个库。

`src/lib.rs` 中的代码构成了库的**根模块**。其他使用这个库的 `crate` 只能访问这个根模块的公共语法项。

8.4 src/bin 目录

要让原来的命令程序 `fern_sim` 再次运行起来也很简单，因为 Cargo 对和库位于同一个 crate 中的小型程序有一些内置支持。

其实 Cargo 本身就是用这样的方式编写的。它的大部分代码在一个 Rust 库中。本书中一直使用的 `cargo` 命令程序只是一个很薄的包装程序，它会调用库来完成所有繁重的工作。库和命令程序都位于同一个源代码存储库中。

你也可以将自己的程序和库放在同一个 crate 中。请将下面这段代码放入名为 `src/bin/efern.rs` 的文件中：

```
use fern_sim::{Fern, run_simulation};

fn main() {
    let mut fern = Fern {
        size: 1.0,
        growth_rate: 0.001
    };
    run_simulation(&mut fern, 1000);
    println!("final fern size: {}", fern.size);
}
```

`main` 函数就是之前搁置的那个。我们为 `fern_sim` crate 中的一些语法项（`Fern` 和 `run_simulation`）添加了 `use` 声明。换句话说，我们在把这个 crate 当库来用。

因为我们已将这个文件放入了 `src/bin` 中，所以 Cargo 将在我们下次运行 `cargo build` 时同时编译 `fern_sim` 库和这个程序。可以使用 `cargo run --bin efern` 来运行 `efern` 程序。下面是它的输出，使用 `--verbose` 可以展示 Cargo 正在运行的命令：

```
$ cargo build --verbose
   Compiling fern_sim v0.1.0 (file:///.../fern_sim)
     Running `rustc src/lib.rs --crate-name fern_sim --crate-type lib ...`
     Running `rustc src/bin/efern.rs --crate-name efern --crate-
```

```
type bin ...`
$ cargo run --bin efern --verbose
    Fresh fern_sim v0.1.0 (file:///.../fern_sim)
    Running `target/debug/efern`
final fern size: 2.7169239322355985
```

我们仍然不必对 Cargo.toml 进行任何修改，因为 Cargo 的默认设定就是查看你的源文件并自行决定做什么。Cargo 会自动将 src/bin 中的 .rs 文件视为应该构建的额外程序。

还可以利用子目录在 src/bin 目录中构建更大的程序。假设我们要提供第二个在屏幕上绘制蕨类植物的程序，但绘制代码规模很大而且是模块化的，因此它拥有自己的文件。我们可以给第二个程序建立它自己的子目录：

```
fern_sim/
├── Cargo.toml
└── src/
    └── bin/
        ├── efern.rs
        └── draw_fern/
            ├── main.rs
            └── draw.rs
```

这样做的好处是能让更大的二进制文件拥有自己的子模块，而不会弄乱库代码或 src/bin 目录。

不过，既然 fern_sim 现在是一个库，那么我们就多了一种选择：把这个程序放在它自己的独立项目中，再保存到一个完全独立的目录中，然后在它自己的 Cargo.toml 中将 fern_sim 列为依赖项：

```
[dependencies]
fern_sim = { path = "../fern_sim" }
```

也许这就是你以后要为其他蕨类植物模拟程序专门做的事。src/bin 目录只适合像 efern 和 draw_fern 这样的简单程序。

8.5 属性

Rust 程序中的任何语法项都可以用属性进行装饰。属性是 Rust 的通用语法，用于向编译器提供各种指令和建议。假设你收到了如下警告：

```
libgit2.rs: warning: type `git_revspec` should have a camel case name
such as `GitRevspec`, #[warn(non_camel_case_types)] on by default
```

但是你选择这个名字是有特别原因的，只希望 Rust 对此“闭嘴”。那么通过在此类型上添加 `#[allow]` 属性就可以禁用这条警告：

```
#[allow(non_camel_case_types)]
pub struct git_revspec {
    ...
}
```

条件编译是使用名为 `#[cfg]` 的属性编写的另一项特性：

```
// 只有当我们为Android构建时才在项目中包含此模块
#[cfg(target_os = "android")]
mod mobile;
```

`#[cfg]` 的完整语法可以到 Rust 参考手册中查看，表 8-2 列出了最常用的选项。

表 8-2：最常用的 `#[cfg]` 选项

<code>#[cfg(...)]</code> 选项	当启用时
<code>test</code>	启用测试（使用 <code>cargo test</code> 或 <code>rustc --test</code> 编译）
<code>debug_assertions</code>	启用调试断言（通常在非优化构建中）
<code>unix</code>	为 Unix（包括 macOS）编译
<code>windows</code>	为 Windows 编译
<code>target_pointer_width = "64"</code>	针对 64 位平台。另一个可能的值是 "32"
<code>target_arch = "x86_64"</code>	特别针对 x86-64。其他值有："x86"、"arm"、"aarch64"、"powerpc"、"powerpc64" 和 "mips"

#[cfg(...)] 选项	当启用时
target_os = "macos"	为 macOS 编译。其他值有: "windows"、"ios"、"android"、"linux"、"freebsd"、"openbsd"、"netbsd" 和 "dragonfly"
feature = "robots"	启用名为 "robots" 的用户自定义特性（用 cargo build --feature robots 或 rustc --cfg feature="robots" 编译）。这些特性是在 Cargo.toml 的 [features] 区段中声明的
not(A)	不满足条件 A 时。如果要提供某函数的两种实现，请将其中一个标记为 #[cfg(X)]，另一个标记为 #[cfg(not(X))]
all(A, B)	同时满足 A 和 B（相当于 &&）
any(A, B)	只要满足 A 或 B 之一（相当于 ）

有时，可能需要对函数的内联展开进行微观管理，但我们通常会把这种优化留给编译器。可以使用 #[inline] 属性进行微观管理：

```
/// 由于相邻细胞之间存在渗透作用，因此需要调整它们的离子水平等
#[inline]
fn do_osmosis(c1: &mut Cell, c2: &mut Cell) {
    ...
}
```

在一种特定的情况下，如果没有 #[inline]，就不会发生内联。当在一个 crate 中定义的函数或方法在另一个 crate 中被调用时，Rust 不会将其内联，除非它是泛型的（具有类型参数）或明确标记为 #[inline]。

在其他情况下，编译器只会将 #[inline] 视为建议。Rust 还支持更坚定的 #[inline(always)]（要求函数在每个调用点内联展开）和 #[inline(never)]（要求函数永不内联）。

一些属性（如 #[cfg] 和 #[allow]）可以附着到整个模块上并对其中的所有内容生效。另一些属性（如 #[test] 和 #[inline]）则必须附着到单个语法项上。正如你对这种包罗万象的语法特性的预期一样，每个属性都是定制的，并且有自己所支持的一组参数。Rust 参考文档详细记录了它支持的全套属性。

要将属性附着到整个 crate 上，请将其添加到 main.rs 文件或 lib.rs 文件的顶部，放在任何语法项之前，并写成 #!，而不是 #，就像这样：

```
// libgit2_sys/lib.rs
#![allow(non_camel_case_types)]

pub struct git_revspec {
    ...
}
```

```
}

pub struct git_error {
    ...
}
```

#! 要求 Rust 将一个属性附着到整个封闭区中的语法项而不只是紧邻其后的内容上：在这种情况下，#![allow] 属性会附着到整个 libgit2_sys 包而不仅仅是 struct git_revspec 上

#! 也可以在函数、结构体等内部使用（但 #! 通常只用在文件的开头，以将属性附着到整个模块或 crate 上）。某些属性始终使用 #! 语法，因为它们只能应用于整个 crate。

例如，#![feature] 属性用于启用 Rust 语言和不稳定特性，这些特性是实验性的，因此可能有 bug 或者未来可能会被更改或移除。例如，在我们撰写本章时，Rust 实验性地支持跟踪像 assert! 这样的宏的展开。但由于此支持是实验性的，因此你只能通过两种方式使用：安装 Rust 的夜间构建版或明确声明你的 crate 使用宏跟踪。

```
#![feature(trace_macros)]

fn main() {
    // 我想知道这个使用assert_eq!的代码替换（展开）后会是什么样子！
    trace_macros!(true);
    assert_eq!(10*10*10 + 9*9*9, 12*12*12 + 1*1*1);
    trace_macros!(false);
}
```

随着时间的推移，Rust 团队有时会将实验性特性稳定下来，使其成为语言标准的一部分。那时这个 #![feature] 属性就会变得多余，因此 Rust 会生成一个警告，建议你将其移除。

8.6 测试与文档

正如 2.3 节所述，Rust 中内置了一个简单的单元测试框架。测试是标有 `#[test]` 属性的普通函数：

```
#[test]
fn math_works() {
    let x: i32 = 1;
    assert!(x.is_positive());
    assert_eq!(x + 1, 2);
}
```

`cargo test` 会运行项目中的所有测试：

```
$ cargo test
  Compiling math_test v0.1.0 (file:///.../math_test)
    Running target/release/math_test-e31ed91ae51ebf22

running 1 test
test math_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0
filtered out
```

（你还会看到一些关于“文档测试”的输出，我们稍后会讲到。）

无论你的 `crate` 是可执行文件还是库，你都可以通过将参数传给 Cargo 来运行特定测试：`cargo test math` 会运行名称中包含 `math` 的所有测试。

测试通常会使用 `assert!` 和 `assert_eq!` 这两个来自 Rust 标准库的宏。如果 `expr` 为真，那么 `assert!(expr)` 就会成功；否则，它会 `panic`，导致测试失败。`assert_eq!(v1, v2)` 和 `assert!(v1 == v2)` 基本等效，但当断言失败时，其错误消息会展示两个值。

你可以在普通代码中使用这些宏来检查不变条件，但请注意 `assert!` 和 `assert_eq!` 会包含在发布构建中。因此，可以改用

`debug_assert!` 和 `debug_assert_eq!` 来编写仅在调试构建中检查的断言。

要测试各种出错情况，请将 `#[should_panic]` 属性添加到你的测试中：

```
/// 正如我们在第7章中所讲的那样，只有当除以零导致panic时，这个测试才能通过
#[test]
#[allow(unconditional_panic, unused_must_use)]
#[should_panic(expected="divide by zero")]
fn test_divide_by_zero_error() {
    1 / 0; // 应该panic!
}
```

在这个例子中，还需要添加一个 `allow` 属性，以让编译器允许我们做一些它本可以静态证明而无法触发 `panic` 的事情，然后才能执行除法并丢弃答案，因为在正常情况下，它会试图阻止这种愚蠢行为。

还可以从测试中返回 `Result<(), E>`。只要错误变体实现了 `Debug` 特型（通常都实现了），你就可以简单地使用 `?` 抛弃 `Ok` 变体以返回 `Result`：

```
use std::num::ParseIntError;

/// 如果"1024"是一个有效的数值（这里正是如此），那么本测试就会通过
#[test]
fn explicit_radix() -> Result<(), ParseIntError> {
    i32::from_str_radix("1024", 10)?;
    Ok(())
}
```

标有 `#[test]` 的函数是有条件编译的。普通的 `cargo build` 或 `cargo build --release` 会跳过测试代码。但是当你运行 `cargo test` 时，Cargo 会分两次来构建你的程序：一次以普通方式，一次带着你的测试和已启用的测试工具。这意味着你的单元测试可以与它们所测试的代码一起使用，按需访问内部实现细节，而且没有运行期成本。但是，这可能会导致一些警告。

```
fn roughly_equal(a: f64, b: f64) -> bool {
    (a - b).abs() < 1e-6
}

#[test]
fn trig_works() {
    use std::f64::consts::PI;
    assert!(roughly_equal(PI.sin(), 0.0));
}
```

在省略了测试代码的构建中，`roughly_equal` 似乎从未使用过，于是 Rust 会报错：

```
$ cargo build
   Compiling math_test v0.1.0 (file:///.../math_test)
warning: function is never used: `roughly_equal`
  |
7 | / fn roughly_equal(a: f64, b: f64) -> bool {
8 | |     (a - b).abs() < 1e-6
9 | | }
  | |_^
  |
  = note: #[warn(dead_code)] on by default
```

因此，当你的测试变得很庞大以至于需要支撑性代码时，应该按照惯例将它们放在 `tests` 模块中，并使用 `#[cfg]` 属性声明整个模块仅用于测试：

```
#[cfg(test)]    // 只有在测试时才包含此模块
mod tests {
    fn roughly_equal(a: f64, b: f64) -> bool {
        (a - b).abs() < 1e-6
    }

    #[test]
    fn trig_works() {
        use std::f64::consts::PI;
        assert!(roughly_equal(PI.sin(), 0.0));
    }
}
```

Rust 的测试工具会使用多个线程同时运行好几个测试，这是 Rust 代码默认线程安全的附带好处之一。要禁用此功能，请运行单个测试

`cargo test testname` 或运行 `cargo test -- --test-threads 1`。（第一个 `--` 确保 `cargo test` 将 `--test-threads` 选项透传给实际执行测试的可执行文件。）这意味着，从严格意义上说，我们在第 2 章中展示的曼德博程序不是该章中第二个而是第三个多线程程序。2.3 节的 `cargo test` 运行的才是第一个多线程程序。

通常，测试工具只会显示失败测试的输出。如果也想展示成功测试的输出，请运行 `cargo test -- --nocapture`。

8.6.1 集成测试

你的蕨类植物模拟器继续“成长”。你已决定将所有主要功能放入可供多个可执行文件使用的库中。如果能从最终使用者的视角编写一些测试，把 `fern_sim.rlib` 像外部 `crate` 那样链接进来，那仿真度就更高了。此外，你有一些测试是通过从二进制文件加载已保存的模拟记录开始的，同时把这些大型测试文件放在 `src` 目录中会很棘手。集成测试有助于解决这两个问题。

集成测试是 `.rs` 文件，位于项目的 `src` 目录旁边的 `tests` 目录中。当你运行 `cargo test` 时，Cargo 会将每个集成测试编译为一个独立的 `crate`，与你的库和 Rust 测试工具链接在一起。下面是一个例子：

```
// tests/unfurl.rs—蕨菜在阳光下舒展开

use fern_sim::Terrarium;
use std::time::Duration;

#[test]
fn test_fiddlehead_unfurling() {
    let mut world =
Terrarium::load("tests/unfurl_files/fiddlehead.tm");
    assert!(world.fern(0).is_furled());
    let one_hour = Duration::from_secs(60 * 60);
    world.apply_sunlight(one_hour);
    assert!(world.fern(0).is_fully_unfurled());
}
```

集成测试之所以有价值，部分原因在于它们会从外部视角看待你的 crate，就像用户的做法一样。集成测试会测试你的 crate 的公共 API。

`cargo test` 会运行单元测试和集成测试。如果仅运行某个特定文件（如 `tests/unfurl.rs`）中的集成测试，请使用命令 `cargo test -test unfurl`。

8.6.2 文档

命令 `cargo doc` 会为你的库创建 HTML 文档：

```
$ cargo doc --no-deps --open
Documenting fern_sim v0.1.0 (file:///.../fern_sim)
```

`--no-deps` 选项会要求 Cargo 只为 `fern_sim` 本身生成文档，而不会为它依赖的所有 crate 生成文档。

`--open` 选项会要求 Cargo 随后在浏览器中打开此文档。

你可以在图 8-2 中看到结果。Cargo 会将这个新文档文件保存在 `target/doc` 中。起始页是 `target/doc/fern_sim/index.html`。

Click or press 'S' to search, '?' for more options...

Crate `fern_sim`

[\[-\]](#) [\[src\]](#)

[\[-\]](#) Simulate the growth of ferns, from the level of individual cells on up.

Reexports

```
pub use plant_structures::Fern;  
pub use simulation::Terrarium;
```

Modules

<code>cells</code>	The simulation of biological cells, which is as low-level as we go.
<code>plant_structures</code>	Higher-level biological structures.
<code>simulation</code>	Overall simulation control.
<code>spores</code>	Fern reproduction.

图 8-2: `rustdoc` 生成的文档示例

该文档是根据你库中的 `pub` 特性以及附加的所有文档型注释生成的。我们已经在本章中看到了一些文档型注释。它们看起来很像注释：

```
/// 模拟减数分裂产生孢子
pub fn produce_spore(factory: &mut Sporangium) -> Spore {
    ...
}
```

但是当 Rust 看到以 3 个斜杠开头的注释时，会将它们视为 `#[doc]` 属性。如下代码的效果和 Rust 处理前面那个例子的效果完全一样：

```
#[doc = "模拟减数分裂产生孢子。"]
pub fn produce_spore(factory: &mut Sporangium) -> Spore {
    ...
}
```

当你编译一个库或二进制文件时，这些属性不会改变任何东西，但是当你生成文档时，关于公共特性的文档型注释会包含在其输出中。

同样，以 `//!` 开头的注释会被视为 `#![doc]` 属性并附着到封闭块一级的特性（通常是模块或 `crate`）上。例如，你的 `fern_sim/src/lib.rs` 文件可能从这句话开始：

```
//! 模拟蕨类植物从单个细胞开始的生长过程
```

文档型注释的内容是 Markdown 格式的，这是一种等效于简单 HTML 格式的简写法。星号用于表示 **斜体** 和 ****粗体****，空行用于表示分段符，等等。你还可以使用 HTML 标记，这些标记会原样复制到格式化后的文档中。

Rust 中文档型注释的一大特色是，Markdown 链接中可以使用 Rust 语法项路径（如 `leaves::Leaf`），而不是相对 URL，来指示它们所指的内容。Cargo 会查找路径所指的内容，并在相应的文档页面中将其替换为正确的链接。例如，从如下代码生成的文档会链接到 `VascularPath`、`Leaf` 和 `Root` 的文档页面：

```
/// 创建并返回一个[`VascularPath`]，它表示从给定的[`Root`][r]到
/// 给定的[`Leaf`] (leaves::Leaf)之间的营养路径
///
/// [r]: roots::Root
pub fn trace_path(leaf: &leaves::Leaf, root: &roots::Root) ->
VascularPath {
```

```
}    ...  
}
```

你还可以添加搜索别名，以便使用内置搜索功能更轻松地查找内容。在此 crate 的文档中搜索“path”或“route”都能找到 VascularPath:

```
#[doc(alias = "route")]  
pub struct VascularPath {  
    ...  
}
```

为了处理较长的文档块或者简化工作流，还可以在文档中包含外部文件。如果存储库的 README.md 文件中包含与准备用作 crate 的顶层文档相同的文本，那么可以将下面这句话放在 lib.rs 或 main.rs 的顶部:

```
#![doc = include_str!("../README.md")]
```

你可以使用反引号 (``) 来标出文本中的少量代码。在输出中，这些片段会格式化为等宽字体。还可以通过“4 空格缩进”来添加更大的代码示例:

```
/// 文档型注释里的代码块:  
///  
///     if samples::everything().works() {  
///         println!("ok");  
///     }
```

也可以使用 Markdown 的三重反引号 (```) 来标记代码块。效果完全一样:

```
/// 另一种代码片段，代码一样，但写法不同:  
///  
/// ```  
/// if samples::everything().works() {  
///     println!("ok");  
/// }  
/// ```
```

无论使用哪种格式，当你在文档型注释里包含一段代码时，都会发生一些有意思的事。Rust 会自动将它转成一个测试。

8.6.3 文档测试

当你在 Rust 库 crate 中运行测试时，Rust 会检查文档中出现的所有代码是否真能如预期般工作。为此，Rust 会获取文档型注释中出现的每个代码块，然后将其编译为单独的可执行包，再与你的库链接在一起，最后运行。

这是一个独立的文档测试示例。通过运行 `cargo new --lib ranges` 创建一个新项目（`--lib` 标志告诉 Cargo 我们正在创建一个库 crate，而不是一个可执行 crate）并将以下代码放入 `ranges/src/lib.rs` 中：

```
use std::ops::Range;

/// 如果两个范围重叠，就返回true
///
///      assert_eq!(ranges::overlap(0..7, 3..10), true);
///      assert_eq!(ranges::overlap(1..5, 101..105), false);
///
/// 如果任何一个范围为空，则它们不会被看作是重叠的
///
///      assert_eq!(ranges::overlap(0..0, 0..10), false);
///
pub fn overlap(r1: Range<usize>, r2: Range<usize>) -> bool {
    r1.start < r1.end && r2.start < r2.end &&
        r1.start < r2.end && r2.start < r1.end
}
```

文档型注释中的这两小段代码会出现在 `cargo doc` 生成的文档中，如图 8-3 所示。

Function `ranges::overlap`

[–][src]

```
pub fn overlap(r1: Range<usize>, r2: Range<usize>) -> bool
```

[–] Return true if two ranges overlap.

```
assert_eq!(ranges::overlap(0..7, 3..10), true);
assert_eq!(ranges::overlap(1..5, 101..105), false);
```

If either range is empty, they don't count as overlapping.

```
assert_eq!(ranges::overlap(0..0, 0..10), false);
```

图 8-3：展示一些文档测试的文档

它们也会成为两个独立的测试：

```
$ cargo test
  Compiling ranges v0.1.0 (file:///.../ranges)
...
  Doc-tests ranges

running 2 tests
test overlap_0 ... ok
test overlap_1 ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0
filtered out
```

如果将 `--verbose` 标志传给 Cargo，你会看到它正在使用 `rustdoc --test` 运行这两个测试。`rustdoc` 会将每个代码示例存储在一个单独的文件中，并给它们添加几行样板代码，以生成两个程序。这是第一个：

```
use ranges;
fn main() {
    assert_eq!(ranges::overlap(0..7, 3..10), true);
}
```

```
    assert_eq!(ranges::overlap(1..5, 101..105), false);  
}
```

这是第二个：

```
use ranges;  
fn main() {  
    assert_eq!(ranges::overlap(0..0, 0..10), false);  
}
```

如果这些程序能成功编译并运行，则测试通过。

这两个代码示例都包含一些断言，但这只是因为在这种情况下，断言构成了不错的文档。文档测试背后的理念并不是将所有测试都放入注释中。相反，你应该编写尽可能好的文档，Rust 只会帮你确保文档中的代码示例能实际编译和运行。

通常，哪怕一个最小的工作示例都会包含一些细节，像导入或设置代码这样的细节是编译代码所必需的，只是还没重要到值得在文档中展示。要隐藏代码示例中的一行，请在该行的开头放置一个 `#`，后跟一个空格：

```
/// 让阳光照进来，并运行模拟器一段时间  
///  
/// # use fern_sim::Terrarium;  
/// # use std::time::Duration;  
/// # let mut tm = Terrarium::new();  
/// tm.apply_sunlight(Duration::from_secs(60));  
///  
pub fn apply_sunlight(&mut self, time: Duration) {  
    ...  
}
```

有时在文档中展示完整的示例程序（包含 `main` 函数）会很有帮助。显然，如果这些代码片段出现在代码示例中，你也不希望 `rustdoc` 自动添加包装代码，因为那样会导致编译无法通过。因此，`rustdoc` 会将所有精确包含字符串 `fn main` 的代码块视为一个完整的程序，不向其中添加任何代码。

可以针对特定的代码块禁用测试。要告诉 Rust 编译你的示例，但不实际运行它，请使用带有 `no_run` 注释的多行代码块：

```
/// 将本地玻璃栽培箱的所有照片上传到在线画廊
///
/// ```no_run
/// let mut session = fern_sim::connect();
/// session.upload_all();
/// ```
pub fn upload_all(&mut self) {
    ...
}
```

如果代码甚至都不希望编译，请改用 `ignore` 而不是 `no_run`。标记为 `ignore` 的块不会出现在 `cargo run` 的输出中，但 `no_run` 测试如果编译了就会显示为已通过。如果代码块根本不是 Rust 代码，请使用语言的名称，比如 `c++` 或 `sh`，或用 `text` 表示纯文本。`rustdoc` 并不了解数百种编程语言的名称，相反，它将任何自己无法识别的注解都理解为该代码块不是 Rust。这会禁用代码的高亮显示和文档测试功能。

8.7 指定依赖项

前面我们看到过告诉 Cargo 从哪里获取项目所依赖的 crate 源代码的一种方法：通过版本号。

```
image = "0.6.1"
```

还有几种指定依赖项的方法，以及一些关于要使用哪个版本的微妙细节，因此值得在这上面花一些时间来讲一下。

首先，你可能想要使用根本没发布在 crates.io 上的依赖项。一种方法是指定 Git 存储库 URL 和修订号：

```
image = { git = "https://github.com/Piston/image.git", rev =  
"528f19c" }
```

这个特定的 crate 是开源的，托管在 GitHub 上，但你也可以轻松地指向托管在公司内网上的私有 Git 存储库。如上述代码所示，你可以指定要使用的特定版本（rev）、标签（tag）或分支名（branch）。（这些都是告诉 Git 要检出哪个源代码版本的方法。）

另一种方法是指定一个包含 crate 源代码的目录：

```
image = { path = "vendor/image" }
```

如果你的团队有个做版本控制的单一存储库，而且其中包含多个 crate 的源代码，或者可能包含整个依赖图，那么这种方法就很方便。每个 crate 都可以使用相对路径指定其依赖项。

对依赖项进行这种层级的控制是一项非常强大的特性。如果你要使用的任何开源 crate 都不完全符合你的喜好，那么可以简单地对其进行分叉：只需点击 GitHub 上的 Fork 按钮并更改 Cargo.toml 文件中的一行即可。你的下一次 cargo build 将自然使用此 crate 的分叉版本而非官方版本。

8.7.1 版本

当你在 `Cargo.toml` 文件中写入类似 `image = "0.13.0"` 这样的内容时，Cargo 会相当宽松地理解它。它会使用自认为与版本 0.13.0 兼容的最新版本的 `image`。

这些兼容性规则改编自语义化版本规范。

- 以 0.0 开头的版本号还非常原始，所以 Cargo 永远不会假定它能与任何其他版本兼容。
- 以 0.*x* (*x* 不为 0) 开头的版本号，可认为与 0.*x* 系列的版本兼容。前面我们指定了 `image` 版本为 0.6.1，但如果可用，则 Cargo 会使用 0.6.3。（这跟语义化版本规范所说的 0.*x* 版本号规则不太一样，但事实证明这条规则太有用了，不能遗漏。）
- 一旦项目达到 1.0，只有出现新的主版本号时才会破坏兼容性。因此，如果你要求版本为 2.0.1，那么 Cargo 可能会使用 2.17.99，但不会使用 3.0。

默认情况下版本号是灵活的，否则使用哪个版本的问题很快就会变成过度的束缚。假设一个库 `libA` 使用 `num = "0.1.31"`，而另一个库 `libB` 使用 `num = "0.1.29"`。如果版本号需要完全匹配，则没有项目能够同时使用这两个库。允许 Cargo 使用任何兼容版本是一个更实用的默认设定。

不过，不同的项目在依赖性和版本控制方面有不同的需求。你可以使用一些运算符来指定确切的版本或版本范围，如表 8-3 所示。

表 8-3：在 `Cargo.toml` 文件中指定版本

Cargo.toml 行	含义
<code>image = "=0.10.0"</code>	仅使用确切的版本 0.10.0

Cargo.toml 行	含义
<code>image = ">=1.0.5"</code>	使用 1.0.5 或更高版本（甚至 2.9，如果其可用的话）
<code>image = ">1.0.5 <1.1.9"</code>	使用高于 1.0.5 但低于 1.1.9 的版本
<code>image = "<=2.7.10"</code>	使用 2.7.10 或更早的任何版本

你偶尔会看到的另一种版本规范是使用通配符 `*`。它会告诉 Cargo 任何版本都可以。除非其他 Cargo.toml 文件包含更具体的约束，否则 Cargo 将使用最新的可用版本。doc.crates.io 上的 Cargo 文档更详细地介绍了这些版本规范。

请注意，兼容性规则意味着不能纯粹出于营销目的而选择版本号。这实际上意味着它们是 crate 的维护者与其用户之间的契约。如果你维护一个版本为 1.7 的 crate，并且决定移除一个函数或进行任何其他不完全向后兼容的更改，则必须将版本号提高到 2.0。如果你将其称为 1.8，就相当于声称这个新版本与 1.7 兼容，而用户可能会发现构建失败了。

8.7.2 Cargo.lock

Cargo.toml 中的版本号是刻意保持灵活的，但我们不希望每次构建 Cargo 时都将其升级到最新的库版本。想象一下，当你正处于紧张的调试过程中时，`cargo build` 突然升级到了新版本的库。这可能带来难以估量的破坏性。调试过程中引入的任何变数都是坏事。事实上，对库而言，从来就没有什么好的时机进行意料之外的改变。

因此，Cargo 有一种内置机制来防止发生这种情况。当第一次构建项目时，Cargo 会输出一个 Cargo.lock 文件，以记录它使用的每个 crate 的确切版本。以后的构建都将参考此文件并继续使用相同的版

本。仅当你要求 Cargo 升级时它才会升级到更新版本，方法是手动增加 Cargo.toml 文件中的版本号或运行 cargo update:

```
$ cargo update
  Updating registry `https://github.com/rust-lang/crates.io-index`
  Updating libc v0.2.7 -> v0.2.11
  Updating png v0.4.2 -> v0.4.3
```

cargo update 只会升级到与你在 Cargo.toml 中指定的内容兼容的最新版本。如果你指定了 image = "0.6.1", 并且想要升级到版本 0.10.0, 则必须自己在 Cargo.toml 中进行更改。下次构建时, Cargo 会将 image 库更新到这个新版本并将新版本号存储在 Cargo.lock 中。

前面的示例展示 Cargo 更新了托管在 crates.io 上的两个 crate。存储在 Git 中的依赖项会发生非常相似的情况。假设 Cargo.toml 文件包含以下内容:

```
image = { git = "https://github.com/Piston/image.git", branch = "master" }
```

如果 cargo build 发现我们有一个 Cargo.lock 文件, 那么它不会从 Git 存储库中拉取新的更改。相反, 它会读取 Cargo.lock 并使用与上次相同的修订版。但是 cargo update 会重新从 master 中拉取, 以便我们的下一个构建使用最新版本。

Cargo.lock 是自动生成的, 通常不用手动编辑。不过, 如果此项目是可执行文件, 那你就应该将 Cargo.lock 提交到版本控制。这样, 构建项目的每个人总是会获得相同的版本。Cargo.lock 文件的版本历史中会记录这些依赖项更新。

如果你的项目是一个普通的 Rust 库, 请不要费心提交 Cargo.lock。你的库的下游用户拥有包含其整个依赖图版本信息的 Cargo.lock 文件, 他们将忽略这个库的 Cargo.lock 文件。在极少数情况下, 你的项目是一个共享库 (比如输出是 .dll 文件、.dylib 文件或 .so 文

件)，它没有这样的下游 cargo 用户，这时候就应该提交 Cargo.lock。

Cargo.toml 灵活的版本说明符使你可以在项目中使用 Rust 库，并最大限度地提高库之间的兼容性。Cargo.lock 中的这些详尽记录可以支持跨机器的一致且可重现的构建。它们会共同帮你避开“依赖地狱”的问题。

8.8 将 crate 发布到 crates.io

如果你已决定将这个蕨类植物模拟库作为开源软件发布，那么，恭喜！这部分很简单。

首先，确保 Cargo 可以为你打包 crate。

```
$ cargo package
warning: manifest has no description, license, license-file,
documentation,
homepage or repository. See
http://doc.crates.io/manifest.html#package-metadata
for more info.
   Packaging fern_sim v0.1.0 (file:///.../fern_sim)
   Verifying fern_sim v0.1.0 (file:///.../fern_sim)
   Compiling fern_sim v0.1.0
(file:///.../fern_sim/target/package/fern_sim-0.1.0)
```

`cargo package` 命令会创建一个文件（在本例中为 `target/package/fern_sim-0.1.0.crate`），其中包含所有库的源文件（包括 `Cargo.toml`）。这是你要上传到 `crates.io` 以便与全世界分享的文件。（可以使用 `cargo package --list` 来查看包含哪些文件。）然后 Cargo 会像最终用户一样，从 `.crate` 文件构建这个库，以仔细检查其工作。

Cargo 会警告 `Cargo.toml` 的 `[package]` 部分缺少一些对下游用户很重要的信息，比如你分发代码所依据的许可证。警告中给出的 URL 是一个很好的资源，因此我们不会在这里详细解释所有字段。简而言之，你可以通过向 `Cargo.toml` 添加几行代码来修复此警告。

```
[package]
name = "fern_sim"
version = "0.1.0"
edition = "2021"
authors = ["You <you@example.com>"]
license = "MIT"
homepage = "https://fernsim.example.com/"
repository = "https://gitlair.com/sporeador/fern_sim"
documentation = "http://fernsim.example.com/docs"
```

```
description = ""
Fern simulation, from the cellular level up.
""
```



一旦在 crates.io 上发布了这个 crate，任何下载你的 crate 的人都能看到此 Cargo.toml 文件。因此，如果 authors 字段包含你希望保密的电子邮件地址，那么现在是更改它的时候了。

这个阶段有时会出现的另一个问题是你的 Cargo.toml 文件可能通过 path 指定其他 crate 的位置，如 8.7 节所示：

```
image = { path = "vendor/image" }
```

对你和你的团队来说，这可能没什么问题。但当其他人下载 fern_sim 库时，他们的计算机上可能不会有与你一样的文件和目录。因此，Cargo 会忽略自动下载的库中的 path 键，而这可能会导致构建错误。解决方案一目了然：如果你的库要发布在 crates.io 上，那么它的依赖项也应该在 crates.io 上。应该指定版本号而不是 path：

```
image = "0.13.0"
```

如果你愿意，可以同时指定一个 path（供你自己在本地构建时优先使用）和一个 version（供所有其他用户使用）：

```
image = { path = "vendor/image", version = "0.13.0" }
```

当然，在这种情况下，你有责任确保两者保持同步。

最后，在发布 crate 之前，你需要登录 crates.io 并获取 API 密钥。这一步很简单：一旦你在 crates.io 上有了账户，其“账户设置”页面就会展示一条 cargo login 命令，就像这样：

```
$ cargo login 5j0dV54BjlXBpUUbfIj7G9DvNl1vsWW1
```

Cargo 会把密钥保存在配置文件中，API 密钥应该像密码一样保密。因此，你应该只在自己控制的计算机上运行此命令。

这些都完成后，最后一步是运行 `cargo publish`:

```
$ cargo publish
  Updating registry `https://github.com/rust-lang/crates.io-index`
  Uploading fern_sim v0.1.0 (file:///.../fern_sim)
```

做完这一步，你的库就成为 crates.io 中成千上万个库中的一员了。

8.9 工作空间

随着项目不断“成长”，你最终会写出很多 crate。它们并存于同一个源代码存储库中：

```
fernsoft/
├── .git/...
├── fern_sim/
│   ├── Cargo.toml
│   ├── Cargo.lock
│   ├── src/...
│   └── target/...
├── fern_img/
│   ├── Cargo.toml
│   ├── Cargo.lock
│   ├── src/...
│   └── target/...
└── fern_video/
    ├── Cargo.toml
    ├── Cargo.lock
    ├── src/...
    └── target/...
```

Cargo 的工作方式是，每个 crate 都有自己的构建目录 `target`，其中包含该 crate 的所有依赖项的单独构建。这些构建目录是完全独立的。即使两个 crate 具有共同的依赖项，它们也不能共享任何已编译的代码。这好像有点儿浪费。

你可以使用 Cargo 工作空间来节省编译时间和磁盘空间。Cargo 工作空间是一组 crate，它们共享着公共构建目录和 `Cargo.lock` 文件。

你需要做的就是存储库的根目录中创建一个 `Cargo.toml` 文件，并将下面这两行代码放入其中：

```
[workspace]
members = ["fern_sim", "fern_img", "fern_video"]
```

这里的 `fern_sim` 等是那些包含你的 `crate` 的子目录名。这些子目录中所有残存的 `Cargo.lock` 文件和 `target` 目录都需要删除。

完成此操作后，任何 `crate` 中的 `cargo build` 都会自动在根目录（在本例中为 `fernsoft/target`）下创建和使用共享构建目录。命令 `cargo build --workspace` 会构建当前工作空间中的所有 `crate`。`cargo test` 和 `cargo doc` 也能接受 `--workspace` 选项。

8.10 更多好资源

如果你仍然意犹未尽，Rust 社区还准备了一些你可能感兴趣的资源。

- 当你在 `crates.io` 上发布一个开源 `crate` 时，你的文档会自动渲染并托管在 `docs.rs` 上，这要归功于 Onur Aslan。
- 如果你的项目在 GitHub 上，那么 Travis CI 可以在每次推送时构建和测试你的代码。设置起来非常容易，有关详细信息，请参阅 `travis-ci.org`。如果你已经熟悉 Travis，则可以从下面这个 `.travis.yml` 文件开始。

```
language: rust
rust:
  - stable
```

- 你可以从 `crate` 的顶层文档型注释生成 `README.md` 文件。此特性是由 Livio Ribeiro 作为第三方 Cargo 插件提供的。运行 `cargo install cargo-readme` 来安装此插件，然后运行 `cargo readme --help` 来学习如何使用它。

我们将继续前行。

虽然 Rust 是一门新语言，但它旨在支持大型、雄心勃勃的项目。它有很棒的工具和活跃的社区。系统程序员也能享受美好。

第 9 章 结构体

很久以前，当牧羊人想要了解两个羊群是否相似时，会挨个对它们进行比对。

——John C. Baez, James Dolan, “Categorification”

Rust 中的结构体（`struct/structure`）类似于 C 和 C++ 中的 `struct` 类型、Python 中的类和 JavaScript 中的对象。结构体会将多个不同类型的值组合成一个单一的值，以便你能把它们作为一个单元来处理。给定一个结构体，你可以读取和修改它的各个组件。结构体也可以具有关联的方法，以对其组件进行操作。

Rust 有 3 种结构体类型：**具名字段型结构体**、**元组型结构体**和**单元型结构体**。这 3 种结构体在引用组件的方式上有所不同：具名字段型结构体会为每个组件命名；元组型结构体会按组件出现的顺序标识它们；单元型结构体则根本没有组件。单元型结构体虽然不常见，但它们比你想象的更有用。

本章将详细解释每种类型并展示它们在内存中的样子；介绍如何向它们添加方法、如何定义适用于不同组件类型的泛型结构体类型，以及如何让 Rust 为你的结构体生成常见的便捷特型的实现。

9.1 具名字段型结构体

具名字段型结构体的定义如下所示：

```
/// 由8位灰度像素组成的矩形
struct GrayscaleMap {
    pixels: Vec<u8>,
    size: (usize, usize)
}
```

它声明了一个 `GrayscaleMap` 类型，其中包含两个给定类型的字段，分别名为 `pixels` 和 `size`。Rust 中的约定是，所有类型（包括结构体）的名称都将每个单词的第一个字母大写（如 `GrayscaleMap`），这称为大驼峰格式（`CamelCase` 或 `PascalCase`）。字段和方法是小写的，单词之间用下划线分隔，这称为蛇形格式（`snake_case`）。

你可以使用结构体表达式构造出此类型的值，如下所示：

```
let width = 1024;
let height = 576;
let image = GrayscaleMap {
    pixels: vec![0; width * height],
    size: (width, height)
};
```

结构体表达式以类型名称（`GrayscaleMap`）开头，后跟一对花括号，其中列出了每个字段的名称和值。还有用来从与字段同名的局部变量或参数填充字段的简写形式：

```
fn new_map(size: (usize, usize), pixels: Vec<u8>) -> GrayscaleMap
{
    assert_eq!(pixels.len(), size.0 * size.1);
    GrayscaleMap { pixels, size }
}
```

结构体表达式 `GrayscaleMap { pixels, size }` 是 `GrayscaleMap { pixels: pixels, size: size }` 的简写形

式。你可以对某些字段使用 `key: value` 语法，而对同一结构体表达式中的其他字段使用简写语法。

要访问结构体的字段，请使用我们熟悉的 `.` 运算符：

```
assert_eq!(image.size, (1024, 576));
assert_eq!(image.pixels.len(), 1024 * 576);
```

与所有其他语法项一样，结构体默认情况下是私有的，仅在声明它们的模块及其子模块中可见。你可以通过在结构体的定义前加上 `pub` 来使结构体在其模块外部可见。结构体中的每个字段默认情况下也是私有的：

```
/// 由8位灰度像素组成的矩形
pub struct GrayscaleMap {
    pub pixels: Vec<u8>,
    pub size: (usize, usize)
}
```

即使一个结构体声明为 `pub`，它的字段也可以是私有的：

```
/// 由8位灰度像素组成的矩形
pub struct GrayscaleMap {
    pixels: Vec<u8>,
    size: (usize, usize)
}
```

其他模块可以使用此结构体及其任何公共的关联函数，但不能按名称访问私有字段或使用结构体表达式来创建新的 `GrayscaleMap` 值。也就是说，要创建结构体型的值，就需要结构体的所有字段都可见。这就是为什么你不能编写结构体表达式来创建新的 `String` 或 `Vec`。这些标准类型都是结构体，但它们的所有字段都是私有的。如果想创建一个值，就必须使用公共的类型关联函数，比如 `Vec::new()`。

创建具名字段结构体的值时，可以使用另一个相同类型的结构体为省略的那些字段提供值。在结构体表达式中，如果具名字段后面跟着 `.. EXPR`，则任何未提及的字段都会从 `EXPR`（必须是相同结构体类

型的另一个值）中获取它们的值。假设我们有一个代表游戏中怪物的结构体：

```
// 在这个游戏中，怪物是一些扫帚。你会看到：
struct Broom {
    name: String,
    height: u32,
    health: u32,
    position: (f32, f32, f32),
    intent: BroomIntent
}

/// `Broom`可以支持的两种用途
#[derive(Copy, Clone)]
enum BroomIntent { FetchWater, DumpWater }
```

对程序员来说，最好的童话故事是 *The Sorcerer's Apprentice*（《魔法师的学徒》）：一个新手魔法师对一把扫帚施了魔法，让它为自己工作，但工作完成后不知道如何让它停下来。于是，他用斧头将扫帚砍成了两半，结果一把扫帚变成了两把，虽然每把扫帚的大小只有原始扫帚的一半，但仍然具有和原始扫帚一样的“工作热情”。

```
// 按值接收输入的Broom（扫帚），并获得所有权
fn chop(b: Broom) -> (Broom, Broom) {
    // 主要从`b`初始化`broom1`，只修改`height`。由于`String`
    // 不是`Copy`类型，因此`broom1`获得了`b`中`name`的所有权
    let mut broom1 = Broom { height: b.height / 2, .. b };

    // 主要从`broom1`初始化`broom2`。由于`String`不是`Copy`类型，
    // 因此我们显式克隆了`name`
    let mut broom2 = Broom { name: broom1.name.clone(), .. broom1 };

    // 为每一半扫帚分别起一个名字
    broom1.name.push_str(" I");
    broom2.name.push_str(" II");

    (broom1, broom2)
}
```

有了这个定义，我们就可以制作一把扫帚，把它一分为二，然后看看会得到什么：

```
let hokey = Broom {  
  name: "Hokey".to_string(),  
  height: 60,  
  health: 100,  
  position: (100.0, 200.0, 0.0),  
  intent: BroomIntent::FetchWater  
};  
  
let (hokey1, hokey2) = chop(hokey);  
assert_eq!(hokey1.name, "Hokey I");  
assert_eq!(hokey1.height, 30);  
assert_eq!(hokey1.health, 100);  
  
assert_eq!(hokey2.name, "Hokey II");  
assert_eq!(hokey2.height, 30);  
assert_eq!(hokey2.health, 100);
```

新的扫帚 hokey1 和 hokey2 获得了修改后的名字，长度只有原来的一半，但生命值都跟原始扫帚一样。

9.2 元组型结构体

第二种结构体类型称为元组型结构体，因为它类似于元组：

```
struct Bounds(usize, usize);
```

构造此类型的值与构造元组非常相似，只是必须包含结构体名称：

```
let image_bounds = Bounds(1024, 768);
```

元组型结构体保存的值称为元素，就像元组的值一样。你可以像访问元组一样访问它们：

```
assert_eq!(image_bounds.0 * image_bounds.1, 786432);
```

元组型结构体的单个元素可以是公共的，也可以不是：

```
pub struct Bounds(pub usize, pub usize);
```

表达式 `Bounds(1024, 768)` 看起来像一个函数调用，实际上它确实是，即定义这种类型时也隐式定义了一个函数：

```
fn Bounds(elem0: usize, elem1: usize) -> Bounds { ... }
```

在最基本的层面上，具名字段型结构体和元组型结构体非常相似。选择使用哪一个需要考虑易读性、无歧义性和简洁性。如果你喜欢用 `.` 运算符来获取值的各个组件，那么用名称来标识字段就能为读者提供更多信息，并且更容易防范拼写错误。如果你通常使用模式匹配来查找这些元素，那么元组型结构体会更好用。

元组型结构体适用于创造**新类型**（newtype），即建立一个只包含单组件的结构体，以获得更严格的类型检查。如果你正在使用纯 ASCII 文本，那么可以像下面这样定义一个新类型：

```
struct Ascii(Vec<u8>);
```

将此类型用于 ASCII 字符串比简单地传递 `Vec<u8>` 缓冲区并在注释中解释它们的内容要好得多。在将其他类型的字节缓冲区传给需要 ASCII 文本的函数时，这种新类型能帮 Rust 捕获错误。我们会在第 22 章中给出一个使用新类型进行高效类型转换的例子。

9.3 单元型结构体

第三种结构体有点儿晦涩难懂，因为它声明了一个根本没有元素的结构体类型：

```
struct Onesuch;
```

这种类型的值不占用内存，很像单元类型 `()`。Rust 既不会在内存中实际存储单元型结构体的值，也不会生成代码来对它们进行操作，因为仅通过值的类型它就能知道关于值的所有信息。但从逻辑上讲，空结构体是一种可以像其他任何类型一样有值的类型。或者更准确地说，空结构体是一种只有一个值的类型：

```
let o = Onesuch;
```

在阅读 6.10 节中有关 `..` 范围运算符的内容时，你已经遇到过单元型结构体。像 `3..5` 这样的表达式是结构体值 `Range { start: 3, end: 5 }` 的简写形式，而表达式 `..`（一个省略两个端点的范围）是单元型结构体值 `RangeFull` 的简写形式。

单元型结构体在处理特型时也很有用，第 11 章会对此进行描述。

9.4 结构体布局

在内存中，具名字段型结构体和元组型结构体是一样的：值（可能是混合类型）的集合以特定方式在内存中布局。例如，在本章前面我们定义了下面这个结构体：

```
struct GrayscaleMap {  
    pixels: Vec<u8>,  
    size: (usize, usize)  
}
```

GrayscaleMap 值在内存中的布局如图 9-1 所示。

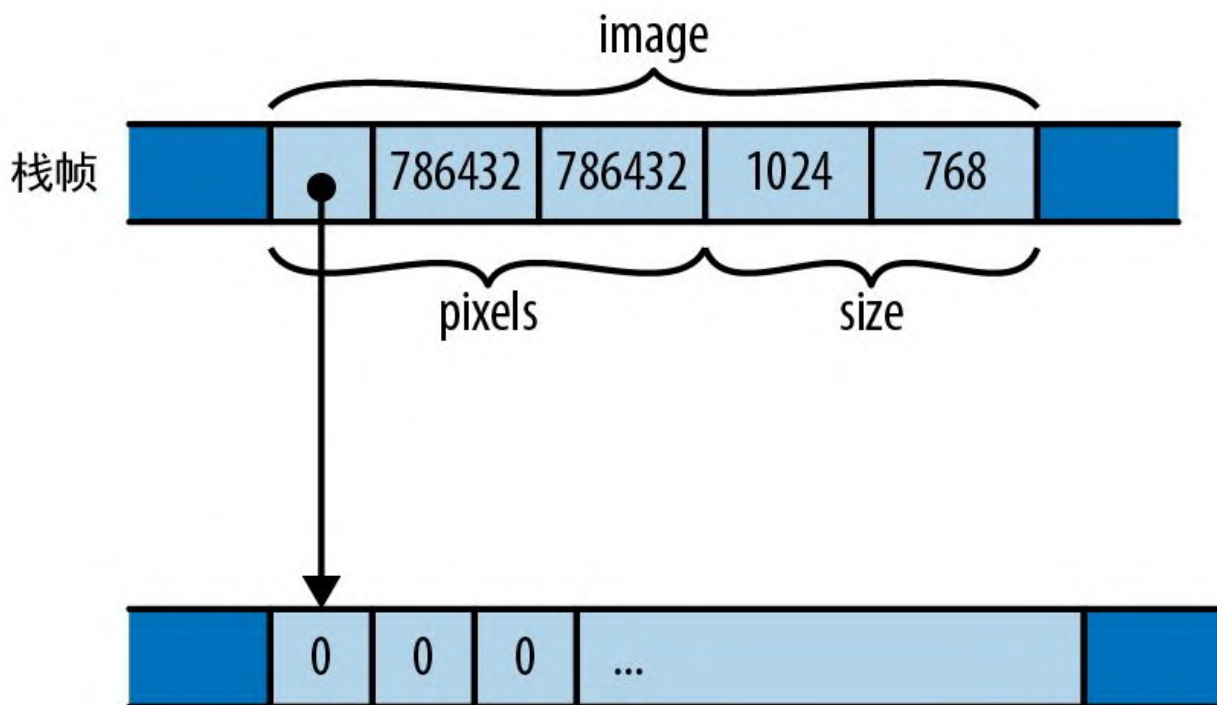


图 9-1：内存中的 `GrayscaleMap` 结构体

与 C 和 C++ 不同，Rust 没有具体承诺它将如何在内存中对结构体的字段或元素进行排序，图 9-1 仅展示了一种可能的安排。然而，Rust 确实承诺会将字段的值直接存储在结构体本身的内存块中。

JavaScript、Python 和 Java 会将 `pixels` 值和 `size` 值分别放在它们自己的分配在堆上的块中，并让 `GrayscaleMap` 的字段指向它

们，而 Rust 会将 `pixels` 值和 `size` 值直接嵌入 `GrayscaleMap` 值中。只有由 `pixels` 向量拥有的在堆上分配的缓冲区才会留在它自己的块中。

你可以使用 `#[repr(C)]` 属性要求 Rust 以兼容 C 和 C++ 的方式对结构体进行布局，第 23 章会对此进行详细介绍。

9.5 用 `impl` 定义方法

在本书中，我们一直在对各种值调用方法，比如使用 `v.push(e)` 将元素推送到向量上、使用 `v.len()` 获取向量的长度、使用 `r.expect("msg")` 检查 `Result` 值是否有错误，等等。你也可以在自己的结构体类型上定义方法。Rust 方法不会像 C++ 或 Java 中的方法那样出现在结构体定义中，而是会出现在单独的 `impl` 块中。

`impl` 块只是 `fn` 定义的集合，每个定义都会成为块顶部命名的结构体类型上的一个方法。例如，这里我们定义了一个公共的 `Queue` 结构体，然后为它定义了 `push` 和 `pop` 这两个公共方法：

```
/// 字符的先入先出队列
pub struct Queue {
    older: Vec<char>,    // 较旧的元素，最早进来的在后面
    younger: Vec<char>   // 较新的元素，最后进来的在后面
}

impl Queue {
    /// 把字符推入队列的最后
    pub fn push(&mut self, c: char) {
        self.younger.push(c);
    }

    /// 从队列的前面弹出一个字符。如果确实有要弹出的字符，
    /// 就返回`Some(c)`；如果队列为空，则返回`None`
    pub fn pop(&mut self) -> Option<char> {
        if self.older.is_empty() {
            if self.younger.is_empty() {
                return None;
            }

            // 将younger中的元素移到older中，并按照所承诺的顺序排列它们
            use std::mem::swap;
            swap(&mut self.older, &mut self.younger);
            self.older.reverse();
        }

        // 现在older能保证有值了。Vec的pop方法已经
        // 返回一个Option，所以可以放心使用了
        self.older.pop()
    }
}
```

```
}  
}
```

在 `impl` 块中定义的函数称为关联函数，因为它们是与特定类型相关联的。与关联函数相对的是自由函数，它是未定义在 `impl` 块中的语法项。

Rust 会将调用关联函数的结构体值作为第一个参数传给方法，该参数必须具有特殊名称 `self`。由于 `self` 的类型显然就是在 `impl` 块顶部命名的类型或对该类型的引用，因此 Rust 允许你省略类型，并以 `self`、`&self` 或 `&mut self` 作为 `self: Queue`、`self: &Queue` 或 `self: &mut Queue` 的简写形式。如果你愿意，也可以使用完整形式，但如前所述，几乎所有 Rust 代码都会使用简写形式。

在我们的示例中，`push` 方法和 `pop` 方法会通过 `self.older` 和 `self.younger` 来引用 `Queue` 的字段。在 C++ 和 Java 中，“`this`”对象的成员可以在方法主体中直接可见，不用加 `this.` 限定符，而 Rust 方法中则必须显式使用 `self` 来引用调用此方法的结构体值，这类似于 Python 方法中使用 `self` 以及 JavaScript 方法中使用 `this` 的方式。

由于 `push` 和 `pop` 需要修改 `Queue`，因此它们都接受 `&mut self` 参数。然而，当调用一个方法时，你不需要自己借用可变引用，常规的方法调用语法就已经隐式处理了这一点。因此，有了这些定义，你就可以像下面这样使用 `Queue` 了：

```
let mut q = Queue { older: Vec::new(), younger: Vec::new() };  
  
q.push('0');  
q.push('1');  
assert_eq!(q.pop(), Some('0'));  
  
q.push('∞');  
assert_eq!(q.pop(), Some('1'));  
assert_eq!(q.pop(), Some('∞'));  
assert_eq!(q.pop(), None);
```

只需编写 `q.push(...)` 就可以借入对 `q` 的可变引用，就好像你写的是 `(&mut q).push(...)` 一样，因为这是 `push` 方法的 `self` 参数所要求的。

如果一个方法不需要修改 `self`，那么可以将其定义为接受共享引用：

```
impl Queue {
    pub fn is_empty(&self) -> bool {
        self.older.is_empty() && self.younger.is_empty()
    }
}
```

同样，方法调用表达式知道要借用哪种引用：

```
assert!(q.is_empty());
q.push('⊙');
assert!(!q.is_empty());
```

或者，如果一个方法想要获取 `self` 的所有权，就可以通过值来获取 `self`：

```
impl Queue {
    pub fn split(self) -> (Vec<char>, Vec<char>) {
        (self.older, self.younger)
    }
}
```

调用这个 `split` 方法看上去和调用其他方法是一样的：

```
let mut q = Queue { older: Vec::new(), younger: Vec::new() };

q.push('P');
q.push('D');
assert_eq!(q.pop(), Some('P'));
q.push('X');

let (older, younger) = q.split();

// q现在是未初始化状态
```

```
assert_eq!(older, vec!['D']);
assert_eq!(younger, vec!['X']);
```

但请注意，由于 `split` 通过值获取 `self`，因此这会将 `Queue` 从 `q` 中移动出去，使 `q` 变成未初始化状态。由于 `split` 的 `self` 现在拥有此队列，因此它能够将这些单独的向量移出队列并返回给调用者。

有时，像这样通过值或引用获取 `self` 还是不够的，因此 Rust 还允许通过智能指针类型传递 `self`。

9.5.1 以 `Box`、`Rc` 或 `Arc` 形式传入 `self`

方法的 `self` 参数也可以是 `Box<Self>` 类型、`Rc<Self>` 类型或 `Arc<Self>` 类型。这种方法只能在给定的指针类型值上调用。调用该方法会将指针的所有权传给它。

你通常不需要这么做。如果一个方法期望通过引用接受 `self`，那它在任何指针类型上调用时都可以正常工作：

```
let mut bq = Box::new(Queue::new());

// `Queue::push` 需要一个 `&mut Queue`，但 `bq` 是一个 `Box<Queue>`
// 这没问题：Rust 在调用期间从 `Box` 借入了 `&mut Queue`
bq.push('■');
```

对于方法调用和字段访问，Rust 会自动从 `Box`、`Rc`、`Arc` 等指针类型中借入引用，因此 `&self` 和 `&mut self` 几乎总是（偶尔也会用一下 `self`）方法签名里的正确选择。

但是如果某些方法确实需要获取指向 `Self` 的指针的所有权，并且其调用者手头恰好有这样一个指针，那么 Rust 也允许你将它作为方法的 `self` 参数传入。为此，你必须明确写出 `self` 的类型，就好像它是普通参数一样。

```
impl Node {
    fn append_to(self: Rc<Self>, parent: &mut Node) {
```

```
        parent.children.push(self);  
    }  
}
```

9.5.2 类型关联函数

给定类型的 `impl` 块还可以定义根本不以 `self` 为参数的函数。这些函数仍然是关联函数，因为它们在 `impl` 块中，但它们不是方法，因为它们不接受 `self` 参数。为了将它们与方法区分开来，我们称其为类型关联函数。

它们通常用于提供构造函数，如下所示：

```
impl Queue {  
    pub fn new() -> Queue {  
        Queue { older: Vec::new(), younger: Vec::new() }  
    }  
}
```

要使用此函数，需要写成 `Queue::new`，即类型名称 + 双冒号 + 函数名称。现在我们的示例代码简洁一点儿了：

```
let mut q = Queue::new();  
  
q.push('*');  
...
```

在 Rust 中，构造函数通常按惯例命名为 `new`，我们已经见过 `Vec::new`、`Box::new`、`HashMap::new` 等。但是 `new` 这个名字并没有什么特别之处，它不是关键字。类型通常还有其他关联函数作为构造函数，比如 `Vec::with_capacity`。

虽然对于一个类型可以有許多独立的 `impl` 块，但它们必须都在定义该类型的同一个 `crate` 中。不过，Rust 确实允许你将自己的方法附加到其他类型中，第 11 章会解释具体做法。

如果你习惯了用 C++ 或 Java，那么将类型的方法与其定义分开可能看起来很不寻常，但这样做有几个优点。

- 找出一个类型的数据成员总是很容易。在大型 C++ 类定义中，你可能需要浏览数百行成员函数的定义才能确保没有遗漏该类的任何数据成员，而在 Rust 中，它们都在同一个地方。
- 尽管可以把方法放到具名字段型结构体中，但对元组型结构体和单元型结构体来说这看上去不那么简洁。将方法提取到一个 `impl` 块中可以让所有这 3 种结构体使用同一套语法。事实上，Rust 还使用相同的语法在根本不是结构体的类型（比如 `enum` 类型和像 `i32` 这样的原始类型）上定义方法。（任何类型都可以有方法，这是 Rust 很少使用**对象**这个术语的原因之一，它更喜欢将所有东西都称为**值**。）
- 同样的 `impl` 语法也可以巧妙地用于实现特型，第 11 章会对此进行介绍。

9.6 关联常量

Rust 在其类型系统中的另一个特性也采用了类似于 C# 和 Java 的思想，有些值是与类型而不是该类型的特定实例关联起来的。在 Rust 中，这些叫作关联常量。

顾名思义，关联常量是常量值。它们通常用于表示指定类型下的常用值。例如，你可以定义一个用于线性代数的二维向量和一个关联的单位向量：

```
pub struct Vector2 {
    x: f32,
    y: f32,
}

impl Vector2 {
    const ZERO: Vector2 = Vector2 { x: 0.0, y: 0.0 };
    const UNIT: Vector2 = Vector2 { x: 1.0, y: 0.0 };
}
```

这些值是和类型本身相关联的，你可以在不必引用 `Vector2` 的任一实例的情况下使用它们。这与关联函数非常相似，使用的名字是与其关联的类型名，后面跟着它们自己的名字：

```
let scaled = Vector2::UNIT.scaled_by(2.0);
```

关联常量的类型不必是其所关联的类型，我们可以使用此特性为类型添加 ID 或名称。如果有多种类似于 `Vector2` 的类型需要写入文件然后加载到内存中，则可以使用关联常量来添加名称或数值 ID，这些名称或数值 ID 可以写在数据旁边以标识其类型。

```
impl Vector2 {
    const NAME: &'static str = "Vector2";
    const ID: u32 = 18;
}
```

9.7 泛型结构体

前面对 `Queue` 的定义并不令人满意：它是为存储字符而写的，但是它的结构体或方法根本没有任何专门针对字符的内容。如果我们要定义另一个包含 `String` 值的结构体，那么除了将 `char` 替换为 `String` 外，其余代码可以完全相同。这纯属浪费时间。

幸运的是，`Rust` 结构体可以是泛型的，这意味着它们的定义是一个模板，你可以在其中插入任何自己喜欢的类型。例如，下面是 `Queue` 的定义，它可以保存任意类型的值：

```
pub struct Queue<T> {
    older: Vec<T>,
    younger: Vec<T>
}
```

你可以把 `Queue<T>` 中的 `<T>` 读作“对于任意元素类型 `T`……”。所以上面的定义可以这样解读：“对于任意元素类型 `T`，`Queue<T>` 有两个 `Vec<T>` 类型的字段。”例如，在 `Queue<String>` 中，`T` 是 `String`，所以 `older` 和 `younger` 的类型都是 `Vec<String>`。而在 `Queue<char>` 中，`T` 是 `char`，我们最终得到的结构体与最初那个针对 `char` 定义的结构体是一样的。事实上，`Vec` 本身也是一个泛型结构体，它就是这样定义的。

在泛型结构体定义中，尖括号（`<>`）中的类型名称叫作类型参数。泛型结构体的 `impl` 块如下所示：

```
impl<T> Queue<T> {
    pub fn new() -> Queue<T> {
        Queue { older: Vec::new(), younger: Vec::new() }
    }

    pub fn push(&mut self, t: T) {
        self.younger.push(t);
    }
}
```



```
pub fn is_empty(&self) -> bool {
    self.older.is_empty() && self.younger.is_empty()
}

...
}
```

你可以将 `impl<T> Queue<T>` 这一行解读为“对于任意元素类型 `T`，这里有一些在 `Queue<T>` 上可用的关联函数。”然后，你可以使用类型参数 `T` 作为关联函数定义中的类型。

语法可能看起来有点儿累赘，但 `impl<T>` 可以清楚地表明 `impl` 块能涵盖任意类型 `T`，这便能将它与为某种特定类型的 `Queue` 编写的 `impl` 块区分开来，如下所示：

```
impl Queue<f64> {
    fn sum(&self) -> f64 {
        ...
    }
}
```

这个 `impl` 块标头表明“这里有一些专门用于 `Queue<f64>` 的关联函数”。这为 `Queue<f64>` 提供了一个 `sum` 方法，不过该方法在其他类型的 `Queue` 上不可用。

我们在前面的代码中使用了 Rust 的 `self` 参数简写形式，如果到处都写成 `Queue<T>`，则让人觉得拗口且容易分心。作为另一种简写形式，每个 `impl` 块，无论是不是泛型，都会将特殊类型的参数 `Self`（注意这里是驼峰 `CamelCase`）定义为我们要为其添加方法的任意类型。对前面的代码来说，`Self` 就应该是 `Queue<T>`，因此我们可以进一步缩写 `Queue::new` 的定义：

```
pub fn new() -> Self {
    Queue { older: Vec::new(), younger: Vec::new() }
}
```

你可能注意到了，在 `new` 的函数体中，不需要在构造表达式中写入类型参数，简单地写 `Queue { ... }` 就足够了。这是 Rust 的类

型推断在起作用：由于只有一种类型适用于该函数的返回值（`Queue<T>`），因此 Rust 为我们补齐了该类型参数。但是，你始终都要在函数签名和类型定义中提供类型参数。Rust 不会推断这些，相反，它会以这些显式类型为基础，推断函数体内的类型。

`Self` 也可以这样使用，我们可以改写成 `Self { ... }`。你觉得哪种写法最容易理解就写成哪种。

在调用关联函数时，可以使用 `::<>`（比目鱼）表示法显式地提供类型参数：

```
let mut q = Queue::::new();
```

但实际上，通常可以让 Rust 帮你推断出来：

```
let mut q = Queue::new();
let mut r = Queue::new();

q.push("CAD"); // 显然是Queue<&'static str>
r.push(0.74);  // 显然是Queue<f64>

q.push("BTC"); // 2019年6月一比特币值多少美元
r.push(13764.0); // Rust可没能力检测出非理性繁荣
```

事实上，我们在本书中经常这样使用另一种泛型结构体类型 `Vec`。

不仅结构体可以是泛型的，枚举同样可以接受类型参数，而且语法也非常相似。10.1 节会详细介绍“枚举”。

9.8 带生命周期参数的泛型结构体

正如我们在 5.3.5 节中讨论的那样，如果结构体类型包含引用，则必须为这些引用的生命周期命名。例如，下面这个结构体可能包含对某个切片的最大元素和最小元素的引用：

```
struct Extrema<'elt> {  
    greatest: &'elt i32,  
    least: &'elt i32  
}
```

早些时候，我们建议你像 `struct Queue<T>` 这样的声明理解为：给定任意类型 `T`，都可以创建一个持有该类型的 `Queue<T>`。同样，可以将 `struct Extrema<'elt>` 理解为：给定任意生命周期 `'elt`，都可以创建一个 `Extrema<'elt>` 来持有对该生命周期的引用。

下面这个函数会扫描切片并返回一个 `Extrema` 值，这个值的各个字段会引用其中的元素：

```
fn find_extrema<'s>(slice: &'s [i32]) -> Extrema<'s> {  
    let mut greatest = &slice[0];  
    let mut least = &slice[0];  
  
    for i in 1..slice.len() {  
        if slice[i] < *least { least = &slice[i]; }  
        if slice[i] > *greatest { greatest = &slice[i]; }  
    }  
    Extrema { greatest, least }  
}
```

在这里，由于 `find_extrema` 借用了 `slice` 的元素，而 `slice` 有生命周期 `'s`，因此我们返回的 `Extrema` 结构体也使用了 `'s` 作为其引用的生命周期。Rust 总会为各种调用推断其生命周期参数，所以调用 `find_extrema` 时不需要提及它们：

```
let a = [0, -3, 0, 15, 48];
let e = find_extrema(&a);
assert_eq!(*e.least, -3);
assert_eq!(*e.greatest, 48);
```

因为返回类型的生命周期与参数的生命周期相同是很常见的情况，所以如果有一个显而易见的候选者，那么 Rust 就允许我们省略生命周期。因此也可以把 `find_extrema` 的签名写成如下形式，意思不变：

```
fn find_extrema(slice: &[i32]) -> Extrema {
    ...
}
```

当然，我们的意思也可能是 `Extrema<'static>`，但这很不寻常。Rust 只为最常见的情况提供了简写形式。

9.9 带常量参数的泛型结构体

泛型结构体也可以接受常量值作为参数。例如，你可以定义一个表示任意次数多项式的类型，如下所示：

```
/// N - 1次多项式
struct Polynomial<const N: usize> {
    /// 多项式的系数
    ///
    /// 对于多项式  $a + bx + cx^2 + \dots + zx^{n-1}$ ，其第`i`个元素是 $x^i$ 的系数
    coefficients: [f64; N]
}
```

例如，根据这个定义，`Polynomial<3>` 是一个二次多项式。这里的 `<const N: usize>` 子句表示 `Polynomial` 类型需要一个 `usize` 值作为它的泛型参数，以此来决定要存储多少个系数。

与通过字段保存长度和容量而将元素存储在堆中的 `Vec` 不同，`Polynomial` 会将其系数 (`coefficients`) 直接存储在值中，再无其他字段。长度直接由类型给出。（这里不需要容量的概念，因为 `Polynomial` 不能动态增长。）

也可以在类型的关联函数中使用参数 `N`：

```
impl<const N: usize> Polynomial<N> {
    fn new(coefficients: [f64; N]) -> Polynomial<N> {
        Polynomial { coefficients }
    }

    /// 计算`x`处的多项式的值
    fn eval(&self, x: f64) -> f64 {
        // 秦九韶算法在数值计算上稳定、高效且简单：
        //  $c_0 + x(c_1 + x(c_2 + x(c_3 + \dots x(c_{n-1} + x c_n))))$ 
        let mut sum = 0.0;
        for i in (0..N).rev() {
            sum = self.coefficients[i] + x * sum;
        }
    }
}
```

```
        sum
    }
}
```

这里，`new` 函数会接受一个长度为 `N` 的数组，并将其元素作为新 `Polynomial` 值的系数。`eval` 方法将在 `0..N` 范围内迭代以找到给定点 `x` 处的多项式值。

与类型参数和生命周期参数一样，Rust 通常也能为常量参数推断出正确的值：

```
use std::f64::consts::FRAC_PI_2;    //  $\pi/2$ 

// 用近似法对`sin`函数求值:  $\sin x \cong x - 1/6 x^3 + 1/120 x^5$ 
// 误差几乎为0, 相当精确!
let sine_poly = Polynomial::new([0.0, 1.0, 0.0, -1.0/6.0, 0.0,
                                1.0/120.0]);
assert_eq!(sine_poly.eval(0.0), 0.0);
assert!((sine_poly.eval(FRAC_PI_2) - 1.).abs() < 0.005);
```

由于我们向 `Polynomial::new` 传递了一个包含 6 个元素的数组，因此 Rust 知道必须构造出一个 `Polynomial<6>`。`eval` 方法仅通过查询其 `Self` 类型就知道 `for` 循环应该运行多少次迭代。由于长度在编译期是已知的，因此编译器可能会用一些顺序执行的代码完全替换循环。

常量泛型参数可以是任意整数类型、`char` 或 `bool`。不允许使用浮点数、枚举和其他类型。

如果结构体还接受其他种类的泛型参数，则生命周期参数必须排在第一位，然后是类型，接下来是任何 `const` 值。例如，一个包含引用数组的类型可以这样声明：

```
struct LumpOfReferences<'a, T, const N: usize> {
    the_lump: [&'a T; N]
}
```

常量泛型参数是 Rust 的一个相对较新的功能，目前它们的使用受到了一定的限制。例如，像下面这样定义 Polynomial 显然更好：

```
/// 一个N次多项式
struct Polynomial<const N: usize> {
    coefficients: [f64; N + 1]
}
```

然而，Rust 会拒绝这个定义：

```
error: generic parameters may not be used in const operations
|
6 |     coefficients: [f64; N + 1]
|                        ^ cannot perform const operation using
`N`
|
= help: const parameters may only be used as standalone
arguments, i.e. `N`
```

虽然 `[f64; N]` 没问题，但像 `[f64; N + 1]` 这样的类型显然对 Rust 来说太过激进了。所以 Rust 暂时施加了这个限制，以避免遇到像下面这样的问题：

```
struct Ketchup<const N: usize> {
    tomayto: [i32; N & !31],
    tomahto: [i32; N - (N % 32)],
}
```

通过计算可知，不管 `N` 取何值，`N & !31` 和 `N - (N % 32)` 总是相等的，因此 `tomayto` 和 `tomahto` 始终具有相同的类型。例如，应该允许将任何一个赋值给另一个。但是，如果能让 Rust 的类型检查器识别这种位运算，就需要把一些令人困惑的极端情况引入这种本已相当复杂的语言中，而这会带来复杂度失控的风险。当然，支持像 `N + 1` 这样的简单表达式是没问题的，并且也确实已经有人在努力教 Rust 顺利处理这些问题。

由于此处关注的是类型检查器的行为，因此这种限制仅适用于出现在类型中的常量参数，比如数组的长度。在普通表达式中，可以随意使用 `N`：像 `N + 1` 和 `N & !31` 这样的写法是完全可以的。

如果要为 `const` 泛型参数提供的值不仅仅是字面量或单个标识符，那么就必须将其括在花括号中，就像 `Polynomial<{5 + 1}>` 这样。此规则能让 Rust 更准确地报告语法错误。

9.10 让结构体类型派生自某些公共特型

结构体很容易编写：

```
struct Point {  
    x: f64,  
    y: f64  
}
```

但是，如果你要开始使用这种 `Point` 类型，很快就会发现它有点儿难用。像这样写的话，`Point` 不可复制或克隆，不能用 `println!("{}", point);` 打印，而且不支持 `==` 运算符和 `!=` 运算符。

这些特性中的每一个在 `Rust` 中都有名称——`Copy`、`Clone`、`Debug` 和 `PartialEq`，它们被称为**特型**。第 11 章会展示如何为自己的结构体手动实现特型。但是对于这些标准特型和其他一些特型，无须手动实现，除非你想要某种自定义行为。`Rust` 可以自动为你实现它们，而且结果准确无误。只需将 `#[derive]` 属性添加到结构体上即可：

```
#[derive(Copy, Clone, Debug, PartialEq)]  
struct Point {  
    x: f64,  
    y: f64  
}
```

这些特型中的每一个都可以为结构体自动实现特型，但前提是结构体的每个字段都实现了该特型。我们可以要求 `Rust` 为 `Point` 派生 `PartialEq`，因为它的两个字段都是 `f64` 类型，而 `f64` 类型已经实现了 `PartialEq`。

`Rust` 还可以派生 `PartialOrd`，这将增加对比较运算符 `<`、`>`、`<=` 和 `>=` 的支持。我们在这里并没有这样做，因为比较两个点以了解一个点是否“小于”另一个点是一件很奇怪的事情。毕竟点和点之间并没有任何常规意义上的顺序可言。所以我们选择不让 `Point` 值支持这些运算符。这种特例就是 `Rust` 让我们自己编写 `#[derive]` 属性而不会自动为它派生每一个可能特型的原因之一。而另一个原因是，

只要实现某个特型就会自动让它成为公共特性，因此可复制性、可克隆性等都会成为该结构体的公共 API 的一部分，应该慎重选择。

第 13 章会详细描述 Rust 的标准特型并解释哪些可用于 `#[derive]`。

9.11 内部可变性

可变性与其他任何事物一样：过犹不及，而你通常只需要一点点就够了。假设你的蜘蛛机器人控制系统有一个中心结构体 `SpiderRobot`，其中包含一些设置和 I/O 句柄。该结构体会在机器人启动时设置好，并且值永不改变：

```
pub struct SpiderRobot {
    species: String,
    web_enabled: bool,
    leg_devices: [fd::FileDesc; 8],
    ...
}
```

机器人的每个主要系统由不同的结构体处理，它们都有一个指向 `SpiderRobot` 的指针：

```
use std::rc::Rc;

pub struct SpiderSenses {
    robot: Rc<SpiderRobot>, // <--指向设置和I/O的指针
    eyes: [Camera; 32],
    motion: Accelerometer,
    ...
}
```

织网、捕食、毒液流量控制等结构体也都有一个 `Rc<SpiderRobot>` 智能指针。回想一下，`Rc` 代表引用计数（reference counting），并且 `Rc` 指向的值始终是共享的，因此将始终不可变。

现在假设你要使用标准 `File` 类型向 `SpiderRobot` 结构体添加一点儿日志记录。但有一个问题：`File` 必须是可变的。所有用于写入的方法都需要一个可变引用。

这种情况经常发生。我们需要一个不可变值（`SpiderRobot` 结构体）中的一丁点儿可变数据（一个 `File`）。这称为内部可变性。

Rust 提供了多种可选方案，本节将讨论两种最直观的类型，即 `Cell<T>` 和 `RefCell<T>`，它们都在 `std::cell` 模块中。¹

¹`cell` 意思是“隔离室、单元格”，引申为“细胞”。——译者注

`Cell<T>` 是一个包含类型 `T` 的单个私有值的结构体。`Cell` 唯一的特殊之处在于，即使你对 `Cell` 本身没有 `mut` 访问权限，也可以获取和设置这个私有值字段。

`Cell::new(value)`（新建）

创建一个新的 `Cell`，将给定的 `value` 移动进去。

`cell.get()`（获取）

返回 `cell` 中值的副本。

`cell.set(value)`（设置）

将给定的 `value` 存储在 `cell` 中，丢弃先前存储的值。

此方法接受一个不可变引用型的 `self`。

```
fn set(&self, value: T)    // 注意：不是`&mut self`
```

当然，这对名为 `set` 的方法来说是相当不寻常的。迄今为止，Rust 一直在告诉我们如果想更改数据，就需要 `mut` 型访问。但出于同样的原因，这个不寻常的细节正是 `Cell` 的全部意义所在。`Cell` 只是改变不变性规则的一种安全方式——一丝不多，一毫不少。

`cell` 还有其他一些方法，你可以查阅其文档进行了解。

如果你想在 `SpiderRobot` 中添加一个简单的计数器，那么 `Cell` 是一个不错的工具。可以写成如下形式：

```
use std::cell::Cell;

pub struct SpiderRobot {
    ...
    hardware_error_count: Cell<u32>,
    ...
}
```

然后，即使 `SpiderRobot` 中的非 `mut` 方法也可以使用 `.get()` 方法和 `.set()` 方法访问 `u32`：

```
impl SpiderRobot {
    /// 把错误计数递增1
    pub fn add_hardware_error(&self) {
        let n = self.hardware_error_count.get();
        self.hardware_error_count.set(n + 1);
    }

    /// 如果报告过任何硬件错误，则为true
    pub fn has_hardware_errors(&self) -> bool {
        self.hardware_error_count.get() > 0
    }
}
```

这很容易，但它无法解决我们的日志记录问题。`Cell` 不允许在共享值上调用 `mut` 方法。`.get()` 方法会返回 `Cell` 中值的副本，因此它仅在 `T` 实现了 `Copy` 特型时才有效。对于日志记录，我们需要一个可变的 `File`，但 `File` 不是 `Copy` 类型。

在这种情况下，正确的工具是 `RefCell`。与 `Cell<T>` 一样，`RefCell<T>` 也是一种泛型类型，它包含类型 `T` 的单个值。但与 `Cell` 不同，`RefCell` 支持借用对其 `T` 值的引用。

`RefCell::new(value)`（新建）

创建一个新的 `RefCell`，将 `value` 移动进去。

`ref_cell.borrow()`（借用）

返回一个 `Ref<T>`，它本质上只是对存储在 `ref_cell` 中值的共享引用。

如果该值已被以可变的方式借出，则此方法会 `panic`，详细信息稍后会解释。

`ref_cell.borrow_mut()`（可变借用）

返回一个 `RefMut<T>`，它本质上是对 `ref_cell` 中值的可变引用。

如果该值已被借出，则此方法会 `panic`，详细信息稍后会解释。

`ref_cell.try_borrow()`（尝试借用）和
`ref_cell.try_borrow_mut()`（尝试可变借用）

行为与 `borrow()` 和 `borrow_mut()` 一样，但会返回一个 `Result`。如果该值已被以可变的方式借出，那么这两个方法不会 `panic`，而是返回一个 `Err` 值。

同样，`RefCell` 也有一些其他的方法，你可以在其文档中进行查找。

仅当你试图打破“可变引用必须独占”的 Rust 规则时，这两个 `borrow` 方法才会 `panic`。例如，以下代码会引起 `panic`：

```
use std::cell::RefCell;

let ref_cell: RefCell<String> = RefCell::new("hello".to_string());

let r = ref_cell.borrow();           // 正确，返回Ref<String>
let count = r.len();                 // 正确，返回"hello".len()
assert_eq!(count, 5);

let mut w = ref_cell.borrow_mut();  // panic: 已被借出
w.push_str(" world");
```

为避免 panic，可以将这两个借用放入不同的块中。这样，在你尝试借用 w 之前，r 已经被丢弃了。

这很像普通引用的工作方式。唯一的区别是，通常情况下，当你借用一个变量的引用时，Rust 会在**编译期**进行检查，以确保你在安全地使用该引用。如果检查失败，则会出现编译错误。RefCell 会使用运行期检查强制执行相同的规则。因此，如果你违反了规则，就会收到 panic（对于 try_borrow 和 try_borrow_mut 则会显示 Err）。

现在我们已经准备好把 RefCell 用在 SpiderRobot 类型中了：

```
pub struct SpiderRobot {
    ...
    log_file: RefCell<File>,
    ...
}

impl SpiderRobot {
    /// 往日志文件中写一行消息
    pub fn log(&self, message: &str) {
        let mut file = self.log_file.borrow_mut();
        // `writeln!`很像`println!`，但会把输出发送到给定的文件中
        writeln!(file, "{}", message).unwrap();
    }
}
```

变量 file 的类型为 RefMut<File>，我们可以像使用 File 的可变引用一样使用它。有关写入文件的详细信息，请参阅第 18 章。

Cell 很容易使用。虽然不得不调用 .get() 和 .set() 或 .borrow() 和 .borrow_mut() 略显尴尬，但这就是我们为违反规则而付出的代价。还有一个缺点虽不太明显但更严重：Cell 以及包含它的任意类型都不是线程安全的。因此 Rust 不允许多个线程同时访问它们。第 19 章会讲解内部可变性的线程安全风格，届时我们会讨论“Mutex<T>”（参见 19.3.2 节）、“原子化类型”（参见 19.3.10 节）和“全局变量”（参见 19.3.11 节）这几项技术。

无论一个结构体是具名字段型的还是元组型的，它都是其他值的聚合：如果我有一个 `SpiderSenses` 结构体，那么就有了指向共享 `SpiderRobot` 结构体的 `Rc` 指针、有了眼睛、有了陀螺仪，等等。所以结构体的本质是“和”这个字：我有 `x` 和 `y`。但是如果围绕“或”这个字构建另一种类型呢？也就是说，当你拥有这种类型的值时，你就拥有了 `x` 或 `y`。这种类型也非常有用，在 `Rust` 中无处不在，它们是第 10 章的主题。

第 10 章 枚举与模式

在计算机领域，总和类型（sum type）长期悲剧性缺位，很多事情却依然行得通，这简直不可思议（参见 Lambda 的缺位）。¹

——Graydon Hoare

¹意思是 Lambda 在主流编程领域的长期缺位造就了大量“烂”代码，总和类型的缺位同样如此。——译者注

本章的第一个主题强劲有力且非常“古老”，它能帮助你在短期内完成很多事（但要付出一定代价），并且许多文化中有关于它的传说。我要说的不是“恶魔”，而是一种用户定义数据类型，长期以来被 ML 社区和 Haskell 社区的黑客们称为总和类型、可区分的联合体（union）或代数数据类型。在 Rust 中，它们被称为枚举。与“恶魔”不同，它们相当安全，而且也不用付出多少代价。

C++ 和 C# 都有枚举，你可以使用它们来定义自己的类型，其值是一组命名常量。例如，你可以定义一个名为 `Color` 的类型，其值为 `Red`、`Orange`、`Yellow` 等。这种枚举也适用于 Rust，但是 Rust 的枚举远不止于此。Rust 枚举还可以包含数据，甚至是不同类型的数据。例如，Rust 的 `Result<String, io::Error>` 类型就是一个枚举，这样的值要么是包含 `String` 型的 `Ok` 值，要么是包含 `io::Error` 的 `Err` 值。C++ 枚举和 C# 枚举则不具备这样的能力。Rust 枚举更像是 C 的联合体，但不同之处在于它是类型安全的。

只要值可能代表多种事物，枚举就很有用。使用枚举的“代价”是你必须通过模式匹配安全地访问数据，这是本章后半部分的主题。

如果你用过 Python 中的解包或 JavaScript 中的解构，那么应该很熟悉“模式”这个词，但 Rust 的模式不止于此。Rust 模式有点儿像

针对所有数据的正则表达式。它们用于测试一个值是否具有特定的目标形态，可以一次从结构体或元组中把多个字段提取到局部变量中。

和正则表达式一样，模式很简洁，通常能在一行代码中完成全部工作。

本章从枚举的基础知识讲起，首先展示数据如何关联到枚举的各个变体，以及枚举如何存储在内存中；然后展示 Rust 的模式和 `match`（匹配）语句如何基于枚举、结构体、数组和切片简洁地表达逻辑。模式中还可以包含引用、移动和 `if` 条件，来让自己更加强大。

10.1 枚举

Rust 中简单的 C 风格枚举很直观：

```
enum Ordering {  
    Less,  
    Equal,  
    Greater,  
}
```

这声明了一个具有 3 个可能值的 `Ordering` 类型，称为变体或构造器：`Ordering::Less`、`Ordering::Equal` 和 `Ordering::Greater`。这个特殊的枚举是标准库的一部分，因此 Rust 代码能够直接导入它：

```
use std::cmp::Ordering;  
  
fn compare(n: i32, m: i32) -> Ordering {  
    if n < m {  
        Ordering::Less  
    } else if n > m {  
        Ordering::Greater  
    } else {  
        Ordering::Equal  
    }  
}
```

或连同其所有构造器一同导入：

```
use std::cmp::Ordering::{self, *};    // ``*`` 导入所有子项  
  
fn compare(n: i32, m: i32) -> Ordering {  
    if n < m {  
        Less  
    } else if n > m {  
        Greater  
    } else {  
        Equal  
    }  
}
```

导入构造器后，我们就可以写成 `Less` 而非 `Ordering::Less`，等等，但是因为这样写意思不太明确，所以通常认为不导入构造器的那种风格更好，除非导入它们能让你的代码更具可读性。

要导入当前模块中声明的枚举的构造器，请使用 `self`：

```
enum Pet {
    Orca,
    Giraffe,
    ...
}

use self::Pet::*;
```

在内存中，C 风格枚举的各个值会存储为整数。有时告诉 Rust 要使用哪几个整数是很有用的：

```
enum HttpStatus {
    Ok = 200,
    NotModified = 304,
    NotFound = 404,
    ...
}
```

否则 Rust 会从 0 开始帮你分配数值。

默认情况下，Rust 会使用可以容纳它们的最小内置整数类型来存储 C 风格枚举。最适合的是单字节：

```
use std::mem::size_of;
assert_eq!(size_of::<Ordering>(), 1);
assert_eq!(size_of::<HttpStatus>(), 2); // 404不适合存入u8
```

你可以通过向枚举添加 `#[repr]` 属性来覆盖 Rust 对内存中表示法的默认选择。有关详细信息，请参阅 23.1 节。

可以将 C 风格枚举转换为整数：

```
assert_eq!(HttpStatus::Ok as i32, 200);
```

从整数到枚举的反向转换则行不通。与 C 和 C++ 不同，Rust 会保证枚举值必然是 `enum` 声明中阐明的值之一。从整数类型到枚举类型的非检查转换可能会破坏此保证，因此不允许这样做。你可以编写自己的“检查完再转换”逻辑：

```
fn http_status_from_u32(n: u32) -> Option<HttpStatus> {
    match n {
        200 => Some(HttpStatus::Ok),
        304 => Some(HttpStatus::NotModified),
        404 => Some(HttpStatus::NotFound),
        ...
        _ => None,
    }
}
```

或者借助 `enum_primitive` crate。它包含一个宏，可以帮你自动生成这类转换代码。

与结构体一样，编译器能为你实现 `==` 运算符等特性，但你必须明确提出要求：

```
#[derive(Copy, Clone, Debug, PartialEq, Eq)]
enum TimeUnit {
    Seconds, Minutes, Hours, Days, Months, Years,
}
```

枚举可以有方法，就像结构体一样：

```
impl TimeUnit {
    /// 返回此时间单位的复数名词
    fn plural(self) -> &'static str {
        match self {
            TimeUnit::Seconds => "seconds",
            TimeUnit::Minutes => "minutes",
            TimeUnit::Hours => "hours",
            TimeUnit::Days => "days",
            TimeUnit::Months => "months",
            TimeUnit::Years => "years",
        }
    }

    /// 返回此时间单位的单数名词
```

```
fn singular(self) -> &'static str {
    self.plural().trim_end_matches('s')
}
}
```

至此，C 风格枚举就介绍完了。更有趣的 Rust 枚举类型是其变体中能持有数据的类型。我们将展示如何将它们存储在内存中、如何通过添加类型参数来泛化它们，以及如何运用枚举构建复杂的数据结构。

10.1.1 带数据的枚举

有些程序总是要显示精确到毫秒的完整日期和时间，但对大多数应用程序来说，使用粗略的近似值（比如“两个月前”）对用户更友好。我们可以使用之前定义的枚举来编写一个新的 enum，以帮忙解决这个问题：

```
/// 刻意四舍五入后的时间戳，所以程序会显示“6个月前”
/// 而非“2016年2月9日上午9点49分”
#[derive(Copy, Clone, Debug, PartialEq)]
enum RoughTime {
    InThePast(TimeUnit, u32),
    JustNow,
    InTheFuture(TimeUnit, u32),
}
```

此枚举中的两个变体 `InThePast` 和 `InTheFuture` 能接受参数。这种变体叫作元组型变体。与元组型结构体一样，这些构造器也是可创建新 `RoughTime` 值的函数：

```
let four_score_and_seven_years_ago =
    RoughTime::InThePast(TimeUnit::Years, 4 * 20 + 7);

let three_hours_from_now =
    RoughTime::InTheFuture(TimeUnit::Hours, 3);
```

枚举还可以有结构体型变体，就像普通结构体一样包含一些具名字段：

```
enum Shape {
    Sphere { center: Point3d, radius: f32 },
```

```
Cuboid { corner1: Point3d, corner2: Point3d },
}

let unit_sphere = Shape::Sphere {
    center: ORIGIN,
    radius: 1.0,
};
```

总而言之，Rust 有 3 种枚举变体，这与我们在第 9 章中展示的 3 种结构体相呼应。没有数据的变体对应于单元型结构体。元组型变体的外观和功能很像元组型结构体。结构体型变体具有花括号和具名字段。单个枚举中可以同时有 3 种类型的变体：

```
enum RelationshipStatus {
    Single,
    InARelationship,
    ItsComplicated(Option<String>),
    ItsExtremelyComplicated {
        car: DifferentialEquation,
        cdr: EarlyModernistPoem,
    },
}
```

枚举的所有构造器和字段都与枚举本身具有相同的可见性。

10.1.2 内存中的枚举

在内存中，带有数据的枚举会以一个小型整数标签加上足以容纳最大变体中所有字段的内存块的格式进行存储。标签字段供 Rust 内部使用。它会区分由哪个构造器创建了值，进而决定这个值应该有哪些字段。

从 Rust 1.50 开始，RoughTime 会占用 8 字节，如图 10-1 所示。

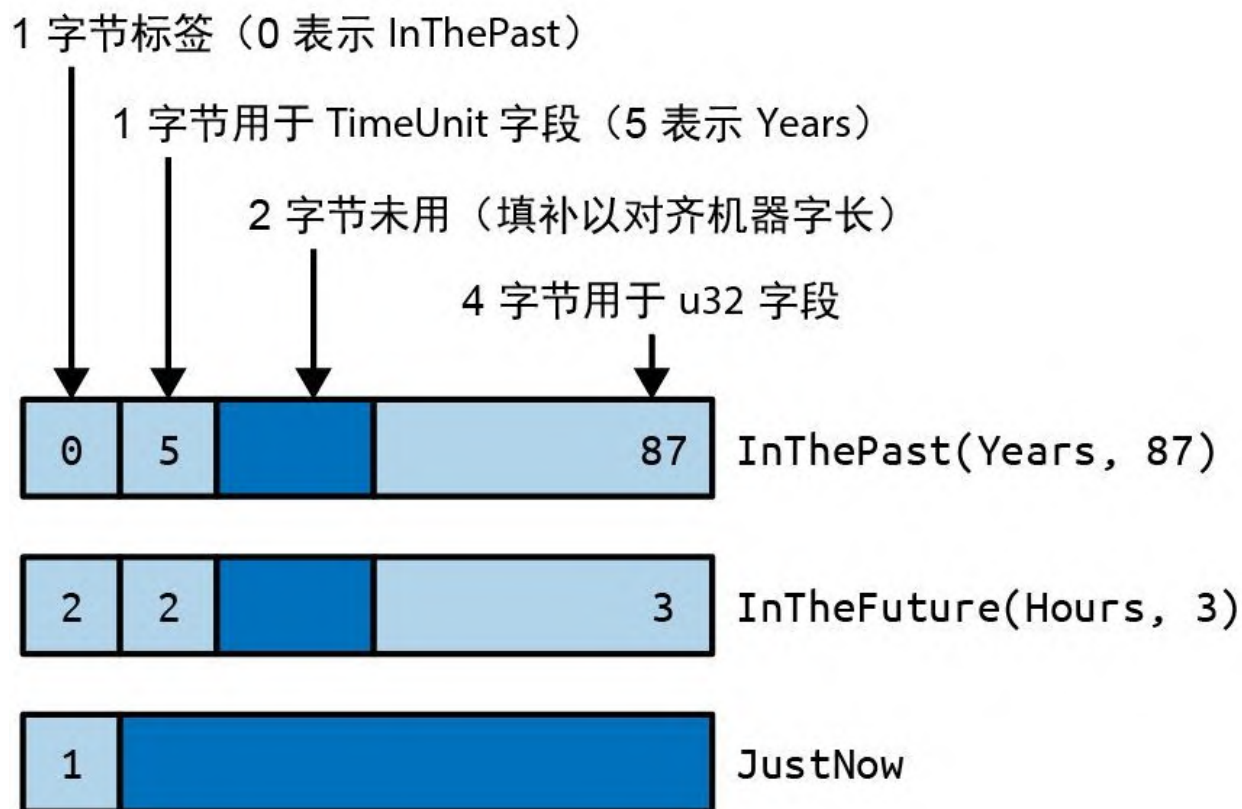


图 10-1: 内存中的 **RoughTime** 值

不过，为了给将来的优化留下余地，Rust 并没有对枚举的内存布局做出任何承诺。在某些情况下，Rust 可以比图 10-1 中展示的布局更有效地打包枚举。例如，有些泛型结构体可以在不需要标签的情况下存储，稍后我们会介绍。

10.1.3 用枚举表示富数据结构

枚举对于快速实现树形数据结构也很有用。假设一个 Rust 程序需要处理任意 JSON 数据。在内存中，任何 JSON 文档都可以表示为这种 Rust 类型的值：

```
use std::collections::HashMap;

enum Json {
    Null,
    Boolean(bool),
    Number(f64),
```



```
String(String),
Array(Vec<Json>),
Object(Box<HashMap<String, Json>>),
}
```

用自然语言解释这种数据结构还不如直接看 Rust 代码。JSON 标准指定了可以出现在 JSON 文档中的不同数据类型：null、布尔值、数值、字符串、各种 JSON 值的数组以及具有字符串键名和 JSON 值的对象。这里的 Json 枚举只是简单地列出了这些类型而已。

这不是一个假想的例子。可以在 `serde_json` 中找到一个非常相似的枚举，`serde_json` 是 Rust 的结构体序列化库，是 `crates.io` 上最常下载的 crate 之一。

这里在表示 Object 的 HashMap 周围加 Box 只是为了让所有 Json 值更紧凑。在内存中，Json 类型的值占用 4 个机器字。而 String 值和 Vec 值占用 3 个机器字，Rust 又添加了一个标签字节。Null 值和 Boolean 值中没有足够的数据来用完所有空间，但所有 Json 值的大小必须相同。因此，额外的空间就用不上了。图 10-2 展示了 Json 值在内存中的实际布局的一些示例。



图 10-2：内存中的 **Json** 值

HashMap 则更大。如果必须在每个 Json 值中为它留出空间，那么将会非常大，在 8 个机器字左右。但是 `Box<HashMap>` 是 1 个机

器字：它只是指向堆中分配的数据的指针。我们甚至可以通过装箱更多字段来让 `Json` 更加紧凑。

这里值得注意的是用 `Rust` 建立这个结构有多么容易。在 `C++` 中，可能要为此编写一个类：

```
class JSON {
private:
    enum Tag {
        Null, Boolean, Number, String, Array, Object
    };
    union Data {
        bool boolean;
        double number;
        shared_ptr<string> str;
        shared_ptr<vector<JSON>> array;
        shared_ptr<unordered_map<string, JSON>> object;

        Data() {}
        ~Data() {}
        ...
    };

    Tag tag;
    Data data;

public:
    bool is_null() const { return tag == Null; }
    bool is_boolean() const { return tag == Boolean; }
    bool get_boolean() const {
        assert(is_boolean());
        return data.boolean;
    }
    void set_boolean(bool value) {
        this->~JSON(); // 清理string/array/object值
        tag = Boolean;
        data.boolean = value;
    }
    ...
};
```

写了 30 行代码，我们才刚开了个头。这个类将需要构造函数、析构函数和赋值运算符。还有一种方法是创建一个具有基类 `JSON` 和子类 `JSONBoolean`、`JSONString` 等的类层次结构。无论采用哪种方

法，操作完成时，我们的 C++ 版 JSON 库都将有十几个方法。其他程序员需要阅读一定的内容才能掌握并使用它。而整个 Rust 枚举才 8 行代码。

10.1.4 泛型枚举

枚举可以是泛型的。Rust 标准库中的两个例子是该语言中最常用的数据类型：

```
enum Option<T> {  
    None,  
    Some(T),  
}  
  
enum Result<T, E> {  
    Ok(T),  
    Err(E),  
}
```

现在你已经很熟悉这些类型了，泛型枚举的语法与泛型结构体是一样的。

一个不太明显的细节是，当类型 `T` 是引用、`Box` 或其他智能指针类型时，Rust 可以省掉 `Option<T>` 的标签字段。由于这些指针类型都不允许为 0，因此 Rust 可以将 `Option<Box<i32>>` 表示为单个机器字：0 表示 `None`，非零表示 `Some` 指针。这能让 `Option` 类型的值尽量接近于可能为空的 C 或 C++ 指针。不同之处在于 Rust 的类型系统要求你在使用其内容之前检查 `Option` 是否为 `Some`。这有效地避免了对空指针解引用。

只需几行代码就可以构建出泛型数据结构：

```
// `T`组成的有序集合  
enum BinaryTree<T> {  
    Empty,  
    NonEmpty(Box<TreeNode<T>>),  
}  
  
// BinaryTree的部件  
struct TreeNode<T> {
```

```
    element: T,  
    left: BinaryTree<T>,  
    right: BinaryTree<T>,  
}
```

这几行代码定义了一个 `BinaryTree` 类型，它可以存储任意数量的 `T` 类型的值。

这两个定义中包含了大量信息，因此我们需要花点儿时间逐字解释这些代码。每个 `BinaryTree` 值要么是 `Empty`，要么是 `NonEmpty`。如果是 `Empty`，那它根本不含任何数据。如果是 `NonEmpty`，那它就会有一个 `Box`，即一个指向堆上分配的 `TreeNode` 的指针。

每个 `TreeNode` 值包含一个实际元素以及另外两个 `BinaryTree` 值。这意味着树可以包含子树，因此 `NonEmpty` 树可以有任意数量的后代。

`BinaryTree<&str>` 类型值的示意图如图 10-3 所示。与 `Option<Box<T>>` 一样，Rust 在这里也省略了标签字段，因此 `BinaryTree` 值占用一个机器字。

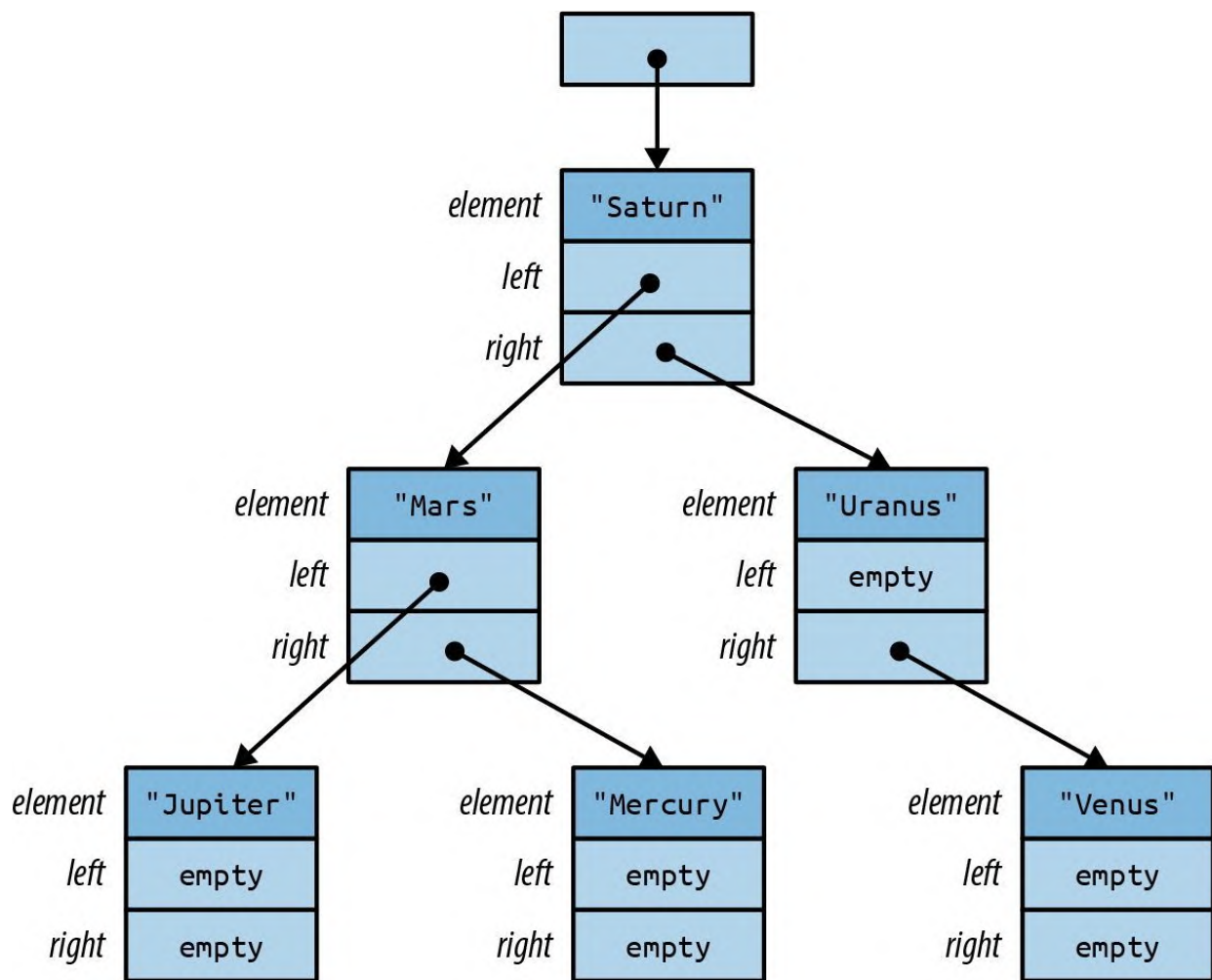


图 10-3: 包含 6 个字符串的 **BinaryTree**

在此树中构建任何特定节点都很简明直观:

```
use self::BinaryTree::*;
let jupiter_tree = NonEmpty(Box::new(TreeNode {
    element: "Jupiter",
    left: Empty,
    right: Empty,
}));
```

较大的树可以基于较小的树来构建:

```
let mars_tree = NonEmpty(Box::new(TreeNode {
    element: "Mars",
    left: jupiter_tree,
```

```
        right: mercury_tree,
    }));
```

自然，此赋值会将 `jupiter_node` 和 `mercury_node` 的所有权转移给它们的新父节点。

树的其余部分都遵循同样的模式。根节点与其他节点没有什么区别：

```
let tree = NonEmpty(Box::new(TreeNode {
    element: "Saturn",
    left: mars_tree,
    right: uranus_tree,
}));
```

稍后本章将展示如何在 `BinaryTree` 类型上实现一个 `add` 方法，以便像下面这样写：

```
let mut tree = BinaryTree::Empty;
for planet in planets {
    tree.add(planet);
}
```

无论你的语言背景如何，在 Rust 中创建像 `BinaryTree` 这样的数据结构都可能需要做一些练习。起初并不容易看出应该把这些 `Box` 放在哪里。找到可行设计方案的方法之一是画出图 10-3 那样的图，展示你希望这些数据在内存中如何布局。然后从图片倒推出代码。每组方块都表示一个结构体或元组，每个箭头都是一个 `Box` 或其他智能指针。弄清楚每个字段的类型虽然有点儿难，但仍然是可以解决的。解决此难题的回报是对程序内存进行了更好的控制。

现在再来说一下本章开头提过的“代价”。枚举的标签字段会占用一点儿内存，最坏情况下可达 8 字节，但这通常可以忽略不计。枚举的真正缺点（如果一定要算的话）是，虽然这些字段真的存在于值中，但 Rust 代码不允许你直接访问它们：

```
let r = shape.radius; // 错误：在`Shape`类型上没有`radius`字段
```

只能用一种安全的方式来访问枚举中的数据，即使用模式。

10.2 模式

回忆一下本章前面定义过的 `RoughTime` 类型：

```
enum RoughTime {  
    InThePast(TimeUnit, u32),  
    JustNow,  
    InTheFuture(TimeUnit, u32),  
}
```

假设你有一个 `RoughTime` 值并希望把它显示在网页上。你需要访问值内的 `TimeUnit` 字段和 `u32` 字段。Rust 不允许你通过编写 `rough_time.0` 和 `rough_time.1` 来直接访问它们，因为毕竟 `rough_time` 也可能是没有字段的，比如 `RoughTime::JustNow`。那么，怎样才能获得数据呢？

你需要一个 `match` 表达式：

```
1  fn rough_time_to_english(rt: RoughTime) -> String {  
2      match rt {  
3          RoughTime::InThePast(units, count) =>  
4              format!("{}", {} ago", count, units.plural()),  
5          RoughTime::JustNow =>  
6              format!("just now"),  
7          RoughTime::InTheFuture(units, count) =>  
8              format!("{}", {} from now", count, units.plural()),  
9      }  
10 }
```

`match` 会执行模式匹配，在此示例中，**模式**就是第 3 行、第 5 行和第 7 行中出现在 `=>` 符号前面的部分。匹配 `RoughTime` 值的模式很像用于创建 `RoughTime` 值的表达式。这是刻意的设计。表达式会生成值，模式会消耗值。两者刻意使用了很多相同的语法。

我们分步了解一下此 `match` 表达式在运行期会发生什么。假设 `rt` 是 `RoughTime::InTheFuture(TimeUnit::Months, 1)` 的值。Rust 会首先尝试将这个值与第 3 行的模式相匹配。如图 10-4 所示，二者不匹配。

值: `RoughTime::InTheFuture(TimeUnit::Months, 1)`



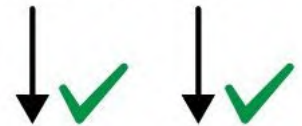
模式: `RoughTime::InThePast(units, count)`

图 10-4: 不匹配的 `RoughTime` 值和模式

对于枚举、结构体或元组类型的匹配，Rust 的工作方式就像简单地从左到右进行扫描一样，会检查模式的每个组件以查看该值是否与之匹配。如果不匹配，Rust 就会接着尝试下一个模式。

第 3 行和第 5 行的模式都不匹配，但是第 7 行的模式匹配成功了，如图 10-5 所示。

值: `RoughTime::InTheFuture(TimeUnit::Months, 1)`



模式: `RoughTime::InTheFuture(` `units, count)`

图 10-5: 一次成功的匹配

模式中包含的简单标识符（如 `units` 和 `count`）会成为模式后面代码中的局部变量。值中存在的任何内容都会复制或移动到新变量中。Rust 会在 `units` 中存储 `TimeUnit::Months`，在 `count` 中存储 `1`，然后运行第 8 行代码，并返回字符串 `"1 months from now"`。

该输出有一个小小的英语语法问题（未处理复数），可以通过在 `match` 中添加另一个分支来解决：

```
RoughTime::InTheFuture(unit, 1) =>  
    format!("a {} from now", unit.singular()),
```


仅当 `count` 字段恰好为 1 时，才会匹配此分支。请注意，这行新代码必须添加到第 7 行之前。如果将其添加到末尾，那么 Rust 将永远无法访问它，因为第 7 行的模式会匹配所有 `InTheFuture` 值。如果你犯了这种错误，那么 Rust 编译器将警告发现了 `"unreachable pattern"`（无法抵达的模式）。

即使用了新代码，
`RoughTime::InTheFuture(TimeUnit::Hours, 1)` 仍然存在问题：`"a hour from now"` 这个结果不太正确。唉，这就是英语啊。这也可以通过在 `match` 中添加另一个分支来解决。

如本示例所示，模式匹配可以和枚举协同工作，甚至可以测试它们包含的数据，这让 `match` 成了 C 的 `switch` 语句的强大而灵活的替代品。迄今为止，我们只看到了匹配枚举值的模式。但模式的类型不止于此，Rust 模式还有它们自己的小型语言，如表 10-1 所示。我们将用本章剩余的大部分内容来介绍此表中展示的特性。

表 10-1：模式

模式类型	例子	注意事项
字面量	100 "name"	匹配一个确切的值；也允许匹配常量名称
范围	0 ..= 100 'a' ..= 'k' 256..	匹配范围内的任何值，包括可能给定的结束值
通配符	_	匹配任何值并忽略它
变量	name mut count	类似于 _，但会把值移动或复制到新的局部变量中

模式类型	例子	注意事项
引用变量	<pre>ref field ref mut field</pre>	借用对匹配值的引用，而不是移动或复制它
与子模式绑定	<pre>val @ 0 ..= 99 ref circle @ Shape::Circle { .. }</pre>	使用 @ 左边的变量名，匹配其右边的模式
枚举型模式	<pre>Some(value) None Pet::Orca</pre>	
元组型模式	<pre>(key, value) (r, g, b)</pre>	
数组型模式	<pre>[a, b, c, d, e, f, g] [heading, carom, correction]</pre>	
切片型模式	<pre>[first, second] [first, _, third] [first, .., nth] []</pre>	
结构体型模式	<pre>Color(r, g, b) Point { x, y } Card { suit: Clubs, rank: n } Account { id, name, .. }</pre>	

模式类型	例子	注意事项
引用	<code>&value</code> <code>&(k, v)</code>	仅匹配引用值
或多个模式	<code>'a' 'A'</code> <code>Some("left" "right")</code>	
守卫表达式	<code>x if x * x <= r2</code>	只用在 <code>match</code> 表达式中（不能用在 <code>let</code> 语句等处）

10.2.1 模式中的字面量、变量和通配符

迄今为止，我们已经展示了如何借助 `match` 表达式来使用枚举。`match` 也可用来匹配其他类型。当你需要类似 C 语言的 `switch` 语句的内容时，可以使用针对整数值的 `match`。像 0 和 1 这样的整型字面量都可以作为模式使用：

```
match meadow.count_rabbits() {
  0 => {} // 无话可说
  1 => println!("A rabbit is nosing around in the clover."),
  n => println!("There are {} rabbits hopping about in the
meadow", n),
}
```

如果草地上没有兔子，就匹配模式 0；如果只有一只兔子，就匹配模式 1；如果有两只或更多的兔子，就匹配第三个模式，即模式 `n`。模式 `n` 只是一个变量名，它可以匹配任何值，匹配的值会移动或复制到一个新的局部变量中。所以在这种情况下，`meadow.count_rabbits()` 的值会存储在一个新的局部变量 `n` 中，然后打印出来。

其他字面量也可以用作模式，包括布尔值、字符，甚至字符串：

```
let calendar = match settings.get_string("calendar") {
    "gregorian" => Calendar::Gregorian,
    "chinese" => Calendar::Chinese,
    "ethiopian" => Calendar::Ethiopian,
    other => return parse_error("calendar", other),
};
```

在这个例子中，`other` 就像上个例子中的 `n` 一样充当了包罗万象的模式。这些模式与 `switch` 语句中的 `default` 分支起着相同的作用，用于匹配与任何其他模式都无法匹配的值。

如果你需要一个包罗万象的模式，但又不关心匹配到的值，那么可以用单个下划线 `_` 作为模式，这就是通配符模式：

```
let caption = match photo.tagged_pet() {
    Pet::Tyrannosaur => "RRRAAAAHHHHHHH",
    Pet::Samoyed => "*dog thoughts*",
    _ => "I'm cute, love me", // 一般性捕获，对任意Pet都生效
};
```

这里的通配符模式能匹配任意值，但不会将其存储到任何地方。由于 Rust 要求每个 `match` 表达式都必须处理所有可能的值，因此最后往往需要一个通配符模式。即使你非常确定其他情况不会发生，也必须至少添加一个后备分支，也许是 `panic` 的分支。

```
// 有很多种形状 (Shape)，但我们只支持“选中”一些文本框
// 或者矩形区域中的所有内容。不能选择椭圆或梯形
match document.selection() {
    Shape::TextSpan(start, end) => paint_text_selection(start,
end),
    Shape::Rectangle(rect) => paint_rect_selection(rect),
    _ => panic!("unexpected selection type"),
}
```

10.2.2 元组型模式与结构体型模式

元组型模式匹配元组。每当你想要在单次 `match` 中获取多条数据时，元组型模式都非常有用：

```
fn describe_point(x: i32, y: i32) -> &'static str {
    use std::cmp::Ordering::*;
    match (x.cmp(&0), y.cmp(&0)) {
        (Equal, Equal) => "at the origin",
        (_, Equal) => "on the x axis",
        (Equal, _) => "on the y axis",
        (Greater, Greater) => "in the first quadrant",
        (Less, Greater) => "in the second quadrant",
        _ => "somewhere else",
    }
}
```

结构体型模式使用花括号，就像结构体表达式一样。结构体型模式包含每个字段的子模式：

```
match balloon.location {
    Point { x: 0, y: height } =>
        println!("straight up {} meters", height),
    Point { x: x, y: y } =>
        println!("at ({}m, {}m)", x, y),
}
```

在此示例中，如果匹配了第一个分支，则 `balloon.location.y` 会存储在新的局部变量 `height` 中。

假设 `balloon.location` 的值是 `Point { x: 30, y: 40 }`。像往常一样，Rust 会依次检查每个模式的每个组件，如图 10-6 所示。



图 10-6：与结构体的模式匹配

这会匹配第二个分支，所以输出是 `at (30m, 40m)`。

像 `Point { x: x, y: y }` 这样的模式在匹配结构体时很常见，而冗余的名称会造成视觉上的混乱，所以 Rust 对此有一个简写形

式: `Point {x, y}`。二者的含义是一样的。`Point {x, y}` 仍会将某个点的 `x` 字段和 `y` 字段分别存储在新的本地变量 `x` 和 `y` 中。

即使用了简写形式, 当我们只关心几个字段时, 匹配大型结构体仍然很麻烦:

```
match get_account(id) {
    ...
    Some(Account {
        name, language, // <---这两个变量才是我们关心的
        id: _, status: _, address: _, birthday: _, eye_color:
    _',
        pet: _, security_question: _, hashed_innermost_secret:
    _',
        is_adamantium_preferred_customer: _, }) =>
        language.show_custom_greeting(name),
}
```

为避免这种情况, 可以使用 `..` 告诉 Rust 你不关心任何其他字段。

```
Some(Account { name, language, .. }) =>
    language.show_custom_greeting(name),
```

10.2.3 数组型模式与切片型模式

数组型模式匹配数组。数组型模式通常用于过滤一些特殊情况的值, 并且在处理那些不同位置的值具有不同含义的数组时也非常有用。

例如, 在将 HSL (色相、饱和度和亮度) 颜色值转换为 RGB (红色、绿色和蓝色) 颜色值时, 具有零亮度或全亮度的颜色只会是黑色或白色。可以使用 `match` 表达式来简单地处理这些情况。

```
fn hsl_to_rgb(hsl: [u8; 3]) -> [u8; 3] {
    match hsl {
        [_, _, 0] => [0, 0, 0],
        [_, _, 255] => [255, 255, 255],
        ...
    }
}
```

切片型模式也与此相似，但与数组不同，切片具有可变长度，因此切片型模式不仅匹配值，还匹配长度。... 在切片型模式中能匹配任意数量的元素。

```
fn greet_people(names: &[&str]) {
    match names {
        [] => { println!("Hello, nobody.") },
        [a] => { println!("Hello, {}. ", a) },
        [a, b] => { println!("Hello, {} and {}. ", a, b) },
        [a, .., b] => { println!("Hello, everyone from {} to {}. ",
a, b) }
    }
}
```

10.2.4 引用型模式

Rust 模式提供了两种特性来支持引用。`ref` 模式会借用已匹配值的一部分。`&` 模式会匹配引用。我们会先介绍 `ref` 模式。

匹配不可复制的值会移动该值。继续以 `account` 为例，以下代码是无效的：

```
match account {
    Account { name, language, .. } => {
        ui.greet(&name, &language);
        ui.show_settings(&account); // 错误：借用已移动的值`account`
    }
}
```

在这里，字段 `account.name` 和 `account.language` 会移动到局部变量 `name` 和 `language` 中。`account` 的其余部分均已丢弃。这就是为什么我们之后不能再借用它的引用。

如果 `name` 和 `language` 都是可复制的值，则 Rust 会复制字段而非移动它们，这时上述代码就是有效的。但假设这些是 `String` 类型，那我们可以做些什么呢？

我们需要一种借用而非移动匹配值的模式。`ref` 关键字就是这样做的：

```
match account {
    Account { ref name, ref language, .. } => {
        ui.greet(name, language);
        ui.show_settings(&account); // 正确
    }
}
```

现在局部变量 `name` 和 `language` 是对 `account` 中相应字段的引用。由于 `account` 只是被借入而没有被消耗，因此继续调用它的方法是没问题的。

还可以使用 `ref mut` 来借入可变引用：

```
match line_result {
    Err(ref err) => log_error(err), // `err`是&Error类型的（共享引用）
    Ok(ref mut line) => {           // `line`是&mut String类型的（可变引用）
        trim_comments(line);        // 就地修改此字符串
        handle(line);
    }
}
```

模式 `Ok(ref mut line)` 能匹配任何成功的结果，并借入其成功值的可变引用。

与 `ref` 模式相对²的引用型模式是 `&` 模式。以 `&` 开头的模式会匹配引用：

² “相对”是指 `ref` 从值取引用，`&` 从引用取值。——译者注

```
match sphere.center() {
    &Point3d { x, y, z } => ...
}
```

在此示例中，假设 `sphere.center()` 会返回对 `sphere` 中的私有字段的引用，这是 Rust 中的常见模式。返回的值是 `Point3d` 的地址。如果中心位于原点，则 `sphere.center()` 会返回 `&Point3d { x: 0.0, y: 0.0, z: 0.0 }`。

模式匹配过程如图 10-7 所示。

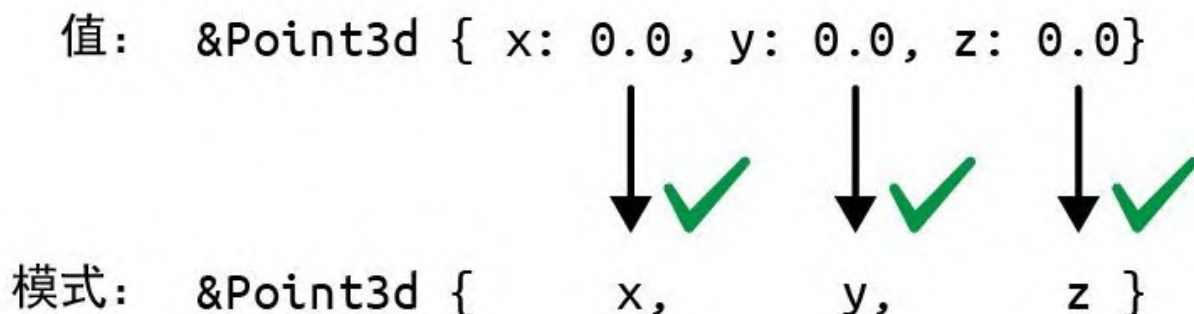


图 10-7: 与引用的模式匹配

这有点儿棘手，因为 Rust 在这里会追踪一个指针，我们通常会将追踪指针的操作与 `*` 运算符而不是 `&` 运算符联系起来。但要记住，模式和表达式是恰恰相反的。表达式 `(x, y)` 会把两个值放入一个新的元组中，而模式 `(x, y)` 则会匹配一个元组并分解成两个值。`&` 的逻辑也是如此。在表达式中，`&` 会创建一个引用。在模式中，`&` 则会匹配一个引用。

匹配引用时会遵循我们所期望的一切规则。生命周期规则仍然有效。你 cannot 通过共享引用获得可变访问权限，而且不能将值从引用中移动出去，即使对可变引用也是如此。当我们匹配 `&Point3d { x, y, z }` 时，变量 `x`、`y` 和 `z` 会接受坐标的副本，而原始 `Point3d` 的值保持不变。这种写法之所以有效，是因为这些字段都是可复制的。如果试图在具有不可复制字段的结构体上这么做，就会出错：

```
match friend.borrow_car() {
    Some(&Car { engine, .. }) => // 错误：不能把借用的值移动出去
    ...
    None => {}
}
```

从借来的汽车上搜刮零件可不是君子所为，Rust 同样不会容忍这么做。你可以使用 `ref` 模式来借用对部件的引用，但并不拥有它：

```
Some(&Car { ref engine, .. }) => // 正确，engine是一个引用
```

再来看一个 `&` 模式的例子。假设我们有一个遍历字符串中各字符的迭代器 `chars`，并且它有一个返回 `Option<&char>`（如果有，则是对下一个字符的引用）的方法 `chars.peek()`。（`Peekable` 迭代器实际上会返回 `Option<&ItemType>`，我们在第 15 章中会看到。）

程序可以使用 `&` 模式来获取它所指向的字符。

```
match chars.peek() {
    Some(&c) => println!("coming up: {:?}", c),
    None => println!("end of chars"),
}
```

10.2.5 匹配守卫

有时，匹配分支会有一些额外的条件，必须满足这些条件才能视为匹配成功。假设我们正在实现一款棋类游戏，它的棋盘是由六边形组成的，而玩家刚刚通过点击移动了一枚棋子。为了确认点击是有效的，我们可能会做如下尝试：

```
fn check_move(current_hex: Hex, click: Point) -> game::Result<Hex>
{
    match point_to_hex(click) {
        None =>
            Err("That's not a game space."),
        Some(current_hex) => // 如果用户单击current_hex，就会尝试匹配
                           // （其实它不起作用：请参见下面的解释）
            Err("You are already there! You must click somewhere
else."),
        Some(other_hex) =>
            Ok(other_hex)
    }
}
```

这失败了，因为模式中的标识符引入了新变量。这里的模式 `Some(current_hex)` 创建了一个新的局部变量 `current_hex`，它遮蔽了同名参数 `current_hex`。Rust 发出了几个关于此代码的警告——特别是，`match` 的最后一个分支是不可达的。解决此问题的一种简单方式是在匹配分支中使用 `if` 表达式：

```
match point_to_hex(click) {
    None => Err("That's not a game space."),
    Some(hex) => {
        if hex == current_hex {
            Err("You are already there! You must click somewhere
else")
        } else {
            Ok(hex)
        }
    }
}
```

但 Rust 还提供了匹配守卫，额外的条件必须为真时才能应用此匹配分支，在模式及其分支的 `=>` 标记之间写上 `if CONDITION`：

```
match point_to_hex(click) {
    None => Err("That's not a game space."),
    Some(hex) if hex == current_hex =>
        Err("You are already there! You must click somewhere
else"),
    Some(hex) => Ok(hex)
}
```

如果模式匹配成功，但此条件为假，就会继续尝试匹配下一个分支。

10.2.6 匹配多种可能性

对于形如 `pat1 | pat2` 的模式，如果能匹配其中的任何一个子模式，则认为匹配成功：

```
let at_end = match chars.peek() {
    Some(&'\\r' | &'\\n') | None => true,
    _ => false,
};
```

在表达式中，`|` 是按位或运算符，但在这里，它更像正则表达式中的 `|` 符号。如果 `chars.peek()` 为 `None`，或者是某个持有回车符、换行符的 `Some`，则把 `at_end` 设置为 `true`。

使用 `..=` 匹配整个范围的值。范围型模式包括开始值和结束值，因此 `'0' ..= '9'` 会匹配所有 ASCII 数字：

```
match next_char {
    '0'..='9' => self.read_number(),
    'a'..='z' | 'A'..='Z' => self.read_word(),
    ' ' | '\t' | '\n' => self.skip_whitespace(),
    _ => self.handle_punctuation(),
}
```

Rust 中还允许使用像 `x..` 这样的范围型模式，该模式会匹配从 `x` 到其类型最大值的任何值。但是，目前模式中还不允许使用其他的开区间范围（如 `0..100` 或 `..100`）以及无限范围（如 `..`）。

10.2.7 使用@模式绑定

最后，`x @ pattern` 会与给定的 `pattern` 精确匹配，但成功时，它不会为匹配到的值的各个部分创建变量，而是会创建单个变量 `x` 并将整个值移动或复制到其中。假设你有如下代码：

```
match self.get_selection() {
    Shape::Rect(top_left, bottom_right) => {
        optimized_paint(&Shape::Rect(top_left, bottom_right))
    }
    other_shape => {
        paint_outline(other_shape.get_outline())
    }
}
```

请注意，第一个分支解包出一个 `Shape::Rect` 值，却只是为了在下一行重建一个相同的 `Shape::Rect` 值。像这种代码可以用 `@` 模式重写：

```
rect @ Shape::Rect(..) => {
    optimized_paint(&rect)
}
```

`@` 模式对于各种范围模式也很有用。

```
match chars.next() {
    Some(digit @ '0'..'9') => read_number(digit, chars),
    ...
},
```

10.2.8 模式能用在哪儿

尽管模式在 `match` 表达式中作用最为突出，但它们也可以出现在其他一些地方，通常用于代替标识符。但无论出现在哪里，其含义都是一样的：Rust 不是要将值存储到单个变量中，而是使用模式匹配来拆分值。

这意味着模式可用于：

```
// 把结构体解包成3个局部变量.....
let Track { album, track_number, title, .. } = song;

// .....解包某个作为函数参数传入的元组
fn distance_to((x, y): (f64, f64)) -> f64 { ... }

// .....迭代某个HashMap上的键和值
for (id, document) in &cache_map {
    println!("Document #{}: {}", id, document.title);
}

// .....自动对闭包参数解引用（当其他代码给你传入引用，
// 而你更想要一个副本时会很有用）
let sum = numbers.fold(0, |a, &num| a + num);
```

上述示例中的每一个都节省了两三行样板代码。同样的概念也存在于其他一些语言中：JavaScript 中叫作**解构**，而 Python 中叫作**解包**。

请注意，上述 4 个示例中都使用了确保匹配的模式。模式 `Point3d { x, y, z }` 会匹配 `Point3d` 结构体类型的每个可能值，`(x, y)` 会匹配任何一个 `(f64, f64)` 值对，等等。这种始终都可以匹配的模式在 Rust 中是很特殊的，它们叫作**不可反驳模式**，是唯一能同时用于此处展示的 4 个位置（`let` 之后、函数参数中、`for` 之后，以及闭包参数中）的模式。

可反驳模式是一种可能不会匹配的模式，比如 `Ok(x)` 不会匹配错误结果，而 `'0' ..= '9'` 不会匹配字符 `'Q'`。可反驳模式可以用在 `match` 的分支中，因为 `match` 就是为此而设计的：如果一个模式无法匹配，那么很清楚接下来会发生什么。在 Rust 程序中，前面的 4 个示例确实是模式可以派上用场的地方，但在这些地方语言不允许匹配失败。

`if let` 表达式和 `while let` 表达式中也允许使用可反驳模式，这些模式可用于：

```
// .....处理只有一个枚举值的特例
if let RoughTime::InTheFuture(_, _) = user.date_of_birth() {
    user.set_time_traveler(true);
}

// .....只有当查表成功时才运行某些代码
if let Some(document) = cache_map.get(&id) {
    return send_cached_response(document);
}

// .....重复尝试某些事，直到成功
while let Err(err) = present_cheesy_anti_robot_task() {
    log_robot_attempt(err);
    // 让用户再试一次（此用户仍然可能是人类）
}

// .....在某个迭代器上手动循环
while let Some(_) = lines.peek() {
    read_paragraph(&mut lines);
}
```

有关这些表达式的详细信息，请参阅 6.5.1 节和 6.5.2 节。

10.2.9 填充二叉树

早些时候我们曾承诺要展示如何实现方法 `BinaryTree::add()`，它能将一个节点添加到如下的 `BinaryTree` 类型中：

```
// `T` 的有序集合
enum BinaryTree<T> {
    Empty,
```

```

        NonEmpty(Box<TreeNode<T>>),
    }

// BinaryTree的部件
struct TreeNode<T> {
    element: T,
    left: BinaryTree<T>,
    right: BinaryTree<T>,
}

```

你现在对模式的了解已经足以写出此方法了。对二叉搜索树的解释超出了本书的范畴，如果你已经很熟悉这个主题，可以自己看看它在 Rust 中的表现。

```

1  impl<T: Ord> BinaryTree<T> {
2      fn add(&mut self, value: T) {
3          match *self {
4              BinaryTree::Empty => {
5                  *self = BinaryTree::NonEmpty(Box::new(TreeNode
6                      {
7                          element: value,
8                          left: BinaryTree::Empty,
9                          right: BinaryTree::Empty,
10                     })))
11             }
12             BinaryTree::NonEmpty(ref mut node) => {
13                 if value <= node.element {
14                     node.left.add(value);
15                 } else {
16                     node.right.add(value);
17                 }
18             }
19         }
20     }

```

第 1 行告诉 Rust 我们正在为有序类型的 `BinaryTree` 定义一个方法。这与我们在泛型结构体上定义方法的语法是完全相同的，详见 9.5 节。

如果现有的树 `*self` 是空的，那就很简单了。运行第 5~9 行代码，将 `Empty` 树更改为 `NonEmpty` 树即可。此处对 `Box::new()` 的调

用在堆中分配了一个新的 `TreeNode`。当完成时，树就会包含一个元素。它的左右子树都是 `Empty`。

如果 `*self` 不为空，那么我们会匹配第 11 行代码的模式：

```
BinaryTree::NonEmpty(ref mut node) => {
```

该模式借用了 `Box<TreeNode<T>>` 的可变引用，因此我们可以访问和修改该树节点中的数据。该引用名为 `node`，位于第 12~16 行代码的作用域内。由于此节点中已经有了一个元素，因此代码必须递归调用 `.add()` 以将此新元素添加到左子树或右子树中。

新方法可以像下面这样使用。

```
let mut tree = BinaryTree::Empty;
tree.add("Mercury");
tree.add("Venus");
...
```


10.3 大局观

Rust 的枚举对系统编程来说可能是新的，但它并不是新思想。它一直顶着各种听起来就很学术的名字（比如代数数据类型）在传播，已经在函数式编程语言中存在四十多年了。目前还不清楚为什么在 C 系列的传承中很少有其他语言支持这种枚举。或许只是因为对编程语言的设计者来说，要将变体、引用、可变性和内存安全这 4 项内容结合使用极具挑战性。函数式编程语言抵触可变性。与之相反，C 的联合体具有变体、指针和可变性——但非常不安全，即使在 C 中，它们也只会迫不得已时使用。Rust 的借用检查器简直就是魔法，它不必做丝毫妥协就能将上述 4 项内容结合起来。

编程就是数据处理。一个小巧、快速、优雅的程序与一个庞大、缓慢、杂乱无章、充斥着各种补丁和虚拟方法调用的程序之间的区别在于，数据是否被转换成了正确的形态。

这就是枚举所针对的“问题空间”。它们是将数据表达为正确形态的设计工具。对于值可能是 A、可能是 B，也可能两者都不是的情况，枚举在每个维度上都比类层次结构表现得要好：更快、更安全、代码更少且更容易文档化。

这里的限制因素是灵活性。枚举的最终用户无法通过扩展枚举来添加新变体，只能通过更改枚举声明来添加。当这种情况发生时，现有代码就会被破坏。我们必须重新审视任何单独匹配枚举的每个变体的 `match` 表达式，因为它需要一个新的分支来处理这个新变体。在某些情况下，为了简单性而牺牲灵活性是很明智的。毕竟，JSON 的语法结构已经定型，不需要灵活性了。在另外一些情况下，当枚举发生变化时，重新审视枚举的所有使用场合正是我们应该做的。例如，当在编译器中使用 `enum` 来表示编程语言的各种运算符时，添加新运算符本来就应该涉及处理运算符的所有代码。

但有时确实需要更大的灵活性。针对这些情况，Rust 设计了一些特型，这就是第 11 章的主题。

第 11 章 特型与泛型

计算机科学家倾向于处理非统一性结构（情形 1、情形 2、情形 3），而数学家则倾向于找一个统一的公理来管理整个体系。

——高德纳

编程领域的伟大发现之一，就是可以编写能对许多不同类型（甚至是尚未发明的类型）的值进行操作的代码。下面是两个例子。

- `Vec<T>` 是泛型的：你可以创建任意类型值的向量，包括在你的程序中定义的连 `Vec` 的作者都不曾设想的类型。
- 许多类型有 `.write()` 方法，包括 `File` 和 `TcpStream`。你的代码可以通过引用来获取任意写入器，并向它发送数据。你的代码不必关心写入器的类型。以后，如果有人添加了一种新型写入器，你的代码也能够直接支持。

当然，这种能力对 Rust 来说并不是什么新鲜事。这就是所谓的多态性，在 20 世纪 70 年代，它是最新且最热门的编程语言技术。到了现在，多态性实际上已经成了通用技术。Rust 通过两个相关联的特性来支持多态：特型和泛型。许多程序员熟悉这些概念，但 Rust 受到 Haskell 类型类（`typeclass`）的启发，采用了一种全新的方式。

特型是 Rust 体系中的接口或抽象基类。乍一看，它们和 Java 或 C# 中的接口差不多。写入字节的特型称为 `std::io::Write`，它在标准库中的定义开头部分是这样的：

```
trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()> { ... }
    ...
}
```

此特型还提供了几个方法，我们只展示了前 3 个。

`File` 和 `TcpStream` 这两个标准类型以及 `Vec<u8>` 都实现了 `std::io::Write`。这 3 种类型都提供了 `.write()`、`.flush()` 等方法。使用写入器而不关心其具体类型的代码如下所示：

```
use std::io::Write;

fn say_hello(out: &mut dyn Write) -> std::io::Result<()> {
    out.write_all(b"hello world\n")?;
    out.flush()
}
```

`out` 的类型是 `&mut dyn Write`，意思是“对实现了 `Write` 特型的任意值的可变引用”。我们可以将任何此类值的可变引用传给 `say_hello`：

```
use std::fs::File;
let mut local_file = File::create("hello.txt")?;
say_hello(&mut local_file)?; // 正常

let mut bytes = vec![];
say_hello(&mut bytes)?; // 同样正常
assert_eq!(bytes, b"hello world\n");
```

本章展示特型的用法、工作原理，以及如何定义你自己的特型。其实，除了前面介绍的用法，特型还有很多其他用法：我们会使用特型为现有类型添加扩展方法，甚至可以在 `str`、`bool` 等内置类型上添加；我们会解释为什么向类型添加特型不需要额外的内存，以及如何在不需要虚方法调用开销的情况下使用特型；我们会看到，某些内置特型其实是 Rust 为运算符重载和其他特性提供的语言级钩子；我们会介绍 `Self` 类型、关联函数和关联类型，这是 Rust 从 Haskell 中借来的三大特性，它们优雅地解决了其他语言中要通过变通和入侵才能解决的问题。

泛型是 Rust 中多态的另一种形式。与 C++ 模板一样，泛型函数或泛型类型可以和不同类型的值一起使用：

```
/// 给定两个值，找出哪个更小
fn min<T: Ord>(value1: T, value2: T) -> T {
    if value1 <= value2 {
        value1
    } else {
        value2
    }
}
```

此函数中的 `<T: Ord>` 表明 `min` 函数可以与实现了 `Ord` 特型的任意类型（任意有序类型）`T` 的参数一起使用。像这样的要求称为**限界**，因为它对 `T` 可能的类型范围做了限制。编译器会针对你实际用到的每种类型 `T` 生成一份单独的机器码。

泛型和特型紧密相关：泛型函数会在限界中使用特型来阐明它能针对哪些类型的参数进行调用。因此，我们还会讨论 `&mut dyn Write` 和 `<T: Write>` 的相似之处、不同之处，以及如何在特型的这两种使用方式之间做出选择。

11.1 使用特型

特型是一种语言特性，我们可以说某类型支持或不支持某个特型。大多数情况下，特型代表着一种能力，即一个类型能做什么。

- 实现了 `std::io::Write` 的值能写出一些字节。
- 实现了 `std::iter::Iterator` 的值能生成一系列值。
- 实现了 `std::clone::Clone` 的值能在内存中克隆自身。
- 实现了 `std::fmt::Debug` 的值能用带有 `{:?}` 格式说明符的 `println!()` 打印出来。

这 4 个特型都是 Rust 标准库的一部分，许多标准类型实现了它们。

- `std::fs::File` 实现了 `Write` 特型，它能将一些字节写入本地文件。`std::net::TcpStream` 能写入网络连接。`Vec<u8>` 也实现了 `Write`，对字节向量的每次 `.write()` 调用都会将一些数据追加到向量末尾。
- `Range<i32>`（表达式 `0..10` 的类型）实现了 `Iterator` 特型，一些与切片、哈希表等关联的迭代器类型同样实现了 `Iterator` 特型。
- 大多数标准库类型实现了 `Clone`。没实现 `Clone` 的主要是一些像 `TcpStream` 这样的类型，因为它们代表的不仅仅是内存中的数据。
- 同样，大多数标准库类型支持 `Debug`。

关于特型方法有一条值得注意的规则：特型本身必须在作用域内。否则，它的所有方法都是不可见的：

```
let mut buf: Vec<u8> = vec![];
buf.write_all(b"hello")?; // 错误：没有名为`write_all`的方法
```

在这种情况下，编译器会打印出一条友好的错误消息，建议添加 `use std::io::Write;`，这确实可以解决问题：

```
use std::io::Write;

let mut buf: Vec<u8> = vec![];
buf.write_all(b"hello")?; // 正确
```

之所以 Rust 会有这条规则，是因为你可以使用特型为任意类型添加新方法——甚至是像 `u32` 和 `str` 这样的标准库类型（正如我们将在本章后面看到的那样）。而第三方 crate 也可以这样做。显然，这可能导致命名冲突。但是由于 Rust 会要求你导入自己想用的特型，因此 crate 可以放心地利用这种超能力。只有导入两个特型，才会发生冲突，将具有相同名称的方法添加到同一个类型中。这在实践中非常罕见。（如果确实遇到冲突，你可以使用带完全限定符的方法名明确写出自己想要的内容，本章后面会介绍。）

`Clone` 和 `Iterator` 的各个方法在没有任何特殊导入的情况下就能工作，因为默认情况下它们始终在作用域中：它们是标准库预导入的一部分，Rust 会把这些名称自动导入每个模块中。事实上，预导入主要就是精心挑选的特型。第 13 章会介绍其中的许多内容。

C++ 程序员和 C# 程序员可能已经看出来了，特型方法类似于虚方法。不过，特型方法的调用仍然很快，与任何其他方法调用一样快。一言以蔽之，这里没有多态。很明显，`buf` 一定是向量，而不可能是文件或网络连接。编译器可以生成对 `Vec<u8>::write()` 的简单调用，甚至可以内联该方法。（C++ 和 C# 通常也会这样做，不过有时会因为子类的缘故而无法做到。）只有通过 `&mut dyn Write` 调用时才会产生动态派发（也叫虚方法调用）的开销，类型上的 `dyn` 关键字指出了这一点。`dyn Write` 叫作**特型对象**，11.1.1 节会介绍特型对象的一些技术细节，以及它们与泛型函数的比较。

11.1.1 特型对象

在 Rust 中使用特型编写多态代码有两种方法：特型对象和泛型。我们将首先介绍特型对象，然后会在 11.1.2 节转向泛型。

Rust 不允许 `dyn Write` 类型的变量：

```
use std::io::Write;

let mut buf: Vec<u8> = vec![];
let writer: dyn Write = buf; // 错误: `Write` 的大小不是常量
```

变量的大小必须是编译期已知的，而那些实现了 `Write` 的类型可以是任意大小。

如果你是 C# 程序员或 Java 程序员，可能会对此感到惊讶，但原因其实很简单。在 Java 中，`OutputStream` 类型（类似于 `std::io::Write` 的 Java 标准接口）的变量其实是对任何实现了 `OutputStream` 的对象的引用。它本身就是引用，无须显式说明。C# 和大多数其他语言中的接口也一样。

我们在 Rust 中也想这么做，但在 Rust 中，引用是显式的：

```
let mut buf: Vec<u8> = vec![];
let writer: &mut dyn Write = &mut buf; // 正确
```

对特型类型（如 `writer`）的引用叫作**特型对象**。与任何其他引用一样，特型对象指向某个值，它具有生命周期，并且可以是可变或共享的。

特型对象的与众不同之处在于，Rust 通常无法在编译期间知道引用目标的类型。因此，特型对象要包含一些关于引用目标类型的额外信息。这仅供 Rust 自己在幕后使用：当你调用 `writer.write(data)` 时，Rust 需要使用类型信息来根据 `*writer` 的具体类型动态调用正确的 `write` 方法。你不能直接查询这些类型信息，Rust 也不支持从特型对象 `&mut dyn Write` 向下转型回像 `Vec<u8>` 这样的具体类型。

特型对象的内存布局

在内存中，特型对象是一个胖指针，由指向值的指针和指向表示该值类型的虚表的指针组成。因此，每个特型对象会占用两个机器字，如图 11-1 所示。

C++ 也有这种运行期类型信息，叫作虚表或 vtable。就像在 C++ 中一样，在 Rust 中，虚表只会在编译期生成一次，并由同一类型的所有对象共享。图 11-1 中深色部分展示的所有内容（包括虚表）都是 Rust 的私有实现细节。同样，这些都不是你可以直接访问的字段和数据结构。当你调用特型对象的方法时，该语言会自动使用虚表来确定要调用哪个实现。

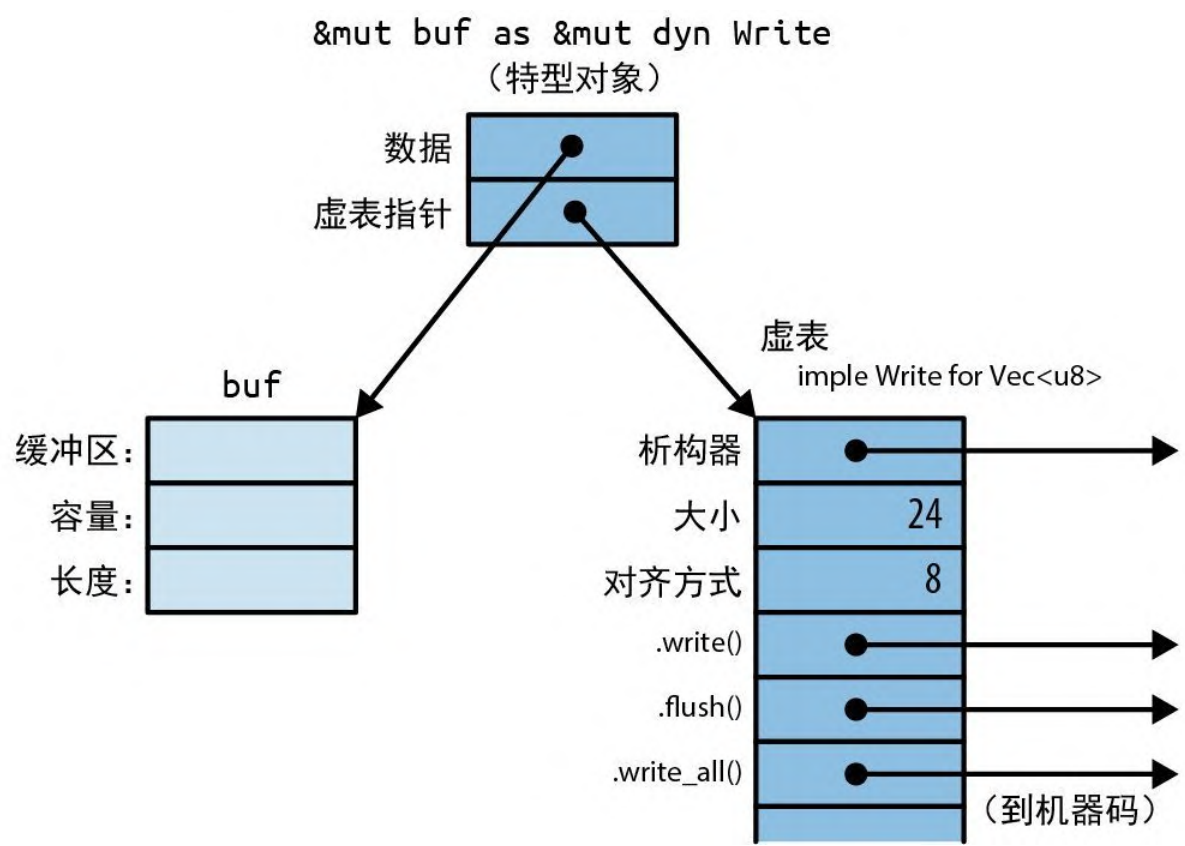


图 11-1：内存中的特型对象

经验丰富的 C++ 程序员可能会注意到 Rust 和 C++ 在内存使用上略有不同。在 C++ 中，虚表指针或 `vptr` 是作为结构体的一部分存储的，而 Rust 使用的是胖指针方案。结构体本身只包含自己的字段。这样一来，每个结构体就可以实现几十个特型而不必包含几十个 `vptr` 了。甚至连 `i32` 这样大小不足以容纳 `vptr` 的类型都可以实现特型。

Rust 在需要时会自动将普通引用转换为特型对象。这就是为什么我们能够在例子中把 `&mut local_file` 传给 `say_hello`:


```
let mut local_file = File::create("hello.txt");  
say_hello(&mut local_file)?;
```

`&mut local_file` 的类型是 `&mut File`，`say_hello` 的参数类型是 `&mut dyn Write`。由于 `File` 也是一种写入器，因此 Rust 允许这样操作，它会自动将普通引用转换为特型对象。

同样，Rust 会愉快地将 `Box<File>` 转换为 `Box<dyn Write>`，这是一个拥有在堆中分配的写入器的值：

```
let w: Box<dyn Write> = Box::new(local_file);
```

和 `&mut dyn Write` 一样，`Box<dyn Write>` 也是一个胖指针，即包含写入器本身的地址和虚表的地址。其他指针类型（如 `Rc<dyn Write>`）同样如此。

这种转换是创建特型对象的唯一方法。编译器在这里真正做的事非常简单。在发生转换的地方，Rust 知道引用目标的真实类型（在本例中为 `File`），因此它只要加上适当的虚表的地址，把常规指针变成胖指针就可以了。

11.1.2 泛型函数与类型参数

本章在开头展示过一个以特型对象为参数的 `say_hello()` 函数。现在，让我们把该函数重写为泛型函数：

```
fn say_hello<W: Write>(out: &mut W) -> std::io::Result<()> {  
    out.write_all(b"hello world\n")?;  
    out.flush()  
}
```

只有类型签名发生了变化：

```
fn say_hello(out: &mut dyn Write)           // 普通函数  
  
fn say_hello<W: Write>(out: &mut W)         // 泛型函数
```

短语 `<W: Write>` 把函数变成了泛型形式。此短语叫作**类型参数**。这意味着在这个函数的整个函数体中，`W` 都代表着某种实现了 `Write` 特型的类型。按照惯例，类型参数通常是单个大写字母。

`W` 代表哪种类型取决于泛型函数的使用方式：

```
say_hello(&mut local_file)?; // 调用say_hello::
say_hello(&mut bytes)?;      // 调用say_hello::
```

当你将 `&mut local_file` 传给泛型函数 `say_hello()` 时，其实调用的是 `say_hello::()`。Rust 会为此函数生成一份机器码，以调用 `File::write_all()` 方法和 `File::flush()` 方法。当你传入 `&mut bytes` 时，其实是在调用 `say_hello::。Rust 会为这个版本的函数生成单独的机器码，以调用相应的 Vec<u8> 方法。在这两种情况下，Rust 都会从参数的类型推断出类型 W，这个过程叫作单态化，编译器会自动处理这一切。`

你总是可以明确写出类型参数：

```
say_hello::(&mut local_file)?;
```

但一般无此必要，因为 Rust 通常可以通过查看参数来推断出类型参数。在这里，`say_hello` 泛型函数需要一个 `&mut W` 参数，我们向它传入了一个 `&mut File`，所以 Rust 断定 `W = File`。

如果你调用的泛型函数没有任何能提供有用线索的参数，则可能需要把它明确写出来：

```
// 调用无参数的泛型方法collect<C>()
let v1 = (0 .. 1000).collect(); // 错误：无法推断类型
let v2 = (0 .. 1000).collect::
```

有时我们需要同一个类型参数的多种能力。如果想打印出向量中前十个最常用的值，那么就要让这些值是可打印的：

```
use std::fmt::Debug;

fn top_ten<T: Debug>(values: &Vec<T>) { ... }
```

但这还不够好。如果我们要确定哪些值是最常用的该怎么办呢？通常的做法是用这些值作为哈希表中的键。这意味着这些值还要支持 Hash 操作和 Eq 操作。T 的类型限界必须包括这些特型，就像 Debug 一样。这种情况下就要使用 + 号语法：

```
use std::hash::Hash;
use std::fmt::Debug;

fn top_ten<T: Debug + Hash + Eq>(values: &Vec<T>) { ... }
```

有些类型实现了 Debug，有些类型实现了 Hash，有些类型支持 Eq，还有一些类型（如 u32 和 String）实现了所有这 3 个，如图 11-2 所示。

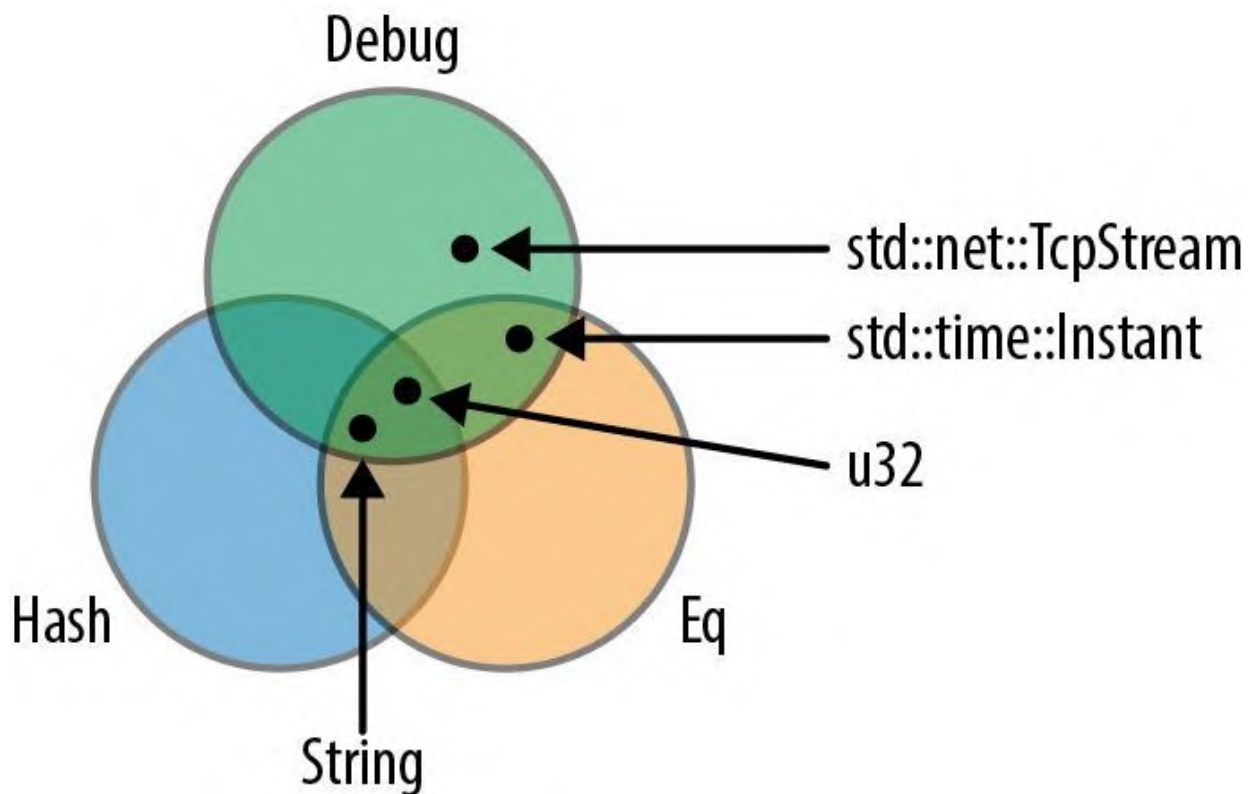


图 11-2：把类型集用作特型

类型参数也可能完全没有限界，但是如果没有为它指定任何限界，则无法对它做什么。你可以移动它，也可以将它放入一个 `Box` 或向量中。但也只能做这些了。

泛型函数可以有多个类型参数：

```
/// 在一个大型且分区的数据集上运行查询
/// 参见<http://research.google.com/archive/mapreduce.html>
fn run_query<M: Mapper + Serialize, R: Reducer + Serialize>(
    data: &DataSet, map: M, reduce: R) -> Results
{ ... }
```

如本示例所示，限界可能会变得很长，让人眼花缭乱。Rust 使用关键字 `where` 提供了另一种语法：

```
fn run_query<M, R>(data: &DataSet, map: M, reduce: R) -> Results
    where M: Mapper + Serialize,
          R: Reducer + Serialize
{ ... }
```

类型参数 `M` 和 `R` 仍然放在前面声明，但限界移到了单独的行中。这种 `where` 子句也允许用于泛型结构体、枚举、类型别名和方法——任何允许使用限界的地方。

当然，替代 `where` 子句的最佳方案是保持简单：找到一种无须大量使用泛型就能编写程序的方式。

5.3.2 节中介绍了生命周期参数的语法。泛型函数可以同时具有生命周期参数和类型参数。生命周期参数要排在前面：

```
/// 返回对`candidates`中最接近`target`的点的引用
fn nearest<'t, 'c, P>(target: &'t P, candidates: &'c [P]) -> &'c P
    where P: MeasureDistance
{
    ...
}
```

这个函数有两个参数，即 `target` 和 `candidates`。两者都是引用，但我们为它们赋予了不同的生命周期 `'t` 和 `'c`（参见 5.3.6

节)。此外，该函数适用于实现了 `MeasureDistance` 特型的任意类型 `P`，因此可以在一个程序中将其用于 `Point2d` 值，而在另一个程序中将其用于 `Point3d` 值。

生命周期永远不会对机器码产生任何影响。如果对 `nearest()` 进行的两次调用使用了相同的类型 `P` 和不同的生命周期，那么就会调用同一个编译结果函数。只有不同的类型才会导致 Rust 编译出泛型函数的多个副本。

除了类型和生命周期，泛型函数也可以接受常量参数，就像 9.9 节中介绍过的 `Polynomial` 结构体：

```
fn dot_product<const N: usize>(a: [f64; N], b: [f64; N]) -> f64 {
    let mut sum = 0.;
    for i in 0..N {
        sum += a[i] * b[i];
    }
    sum
}
```

在这里，短语 `<const N: usize>` 指出函数 `dot_product` 需要一个泛型参数 `N`，该参数必须是一个 `usize`。给定了 `N`，这个函数就会接受两个 `[f64; N]` 类型的参数，并将其对应元素的乘积相加。`N` 与普通 `usize` 参数的区别是，你可以在 `dot_product` 的签名或函数体的类型中使用它。

与类型参数一样，你既可以显式提供常量参数，也可以让 Rust 推断它们。

```
// 显式提供`3`作为`N`的值
dot_product::<3>([0.2, 0.4, 0.6], [0., 0., 1.])

// 让Rust推断`N`必然是`2`
dot_product([3., 4.], [-5., 1.])
```

当然，函数并不是 Rust 中唯一的泛型代码。

- 9.7 节和 10.1.4 节已经介绍过泛型类型。

- 单独的方法也可以是泛型的，即使它并没有定义在泛型类型上。

```
impl PancakeStack {  
    fn push<T: Topping>(&mut self, goop: T) ->  
PancakeResult<()> {  
        goop.pour(&self);  
        self.absorb_topping(goop)  
    }  
}
```

- 类型别名也可以是泛型的。

```
type PancakeResult<T> = Result<T, PancakeError>;
```

- 本章在后面还会介绍一些泛型特型。

本节中介绍的特性（限界、where 子句、生命周期参数等）可用于所有泛型语法项，而不仅仅是函数。

11.1.3 使用哪一个

关于使用特型对象还是泛型代码的选择相当微妙。由于这两个特性都基于特型，因此它们有很多共同点。

当你需要一些混合类型值的集合时，特型对象是正确的选择。制作泛型沙拉在技术上是可行的：

```
trait Vegetable {  
    ...  
}  
  
struct Salad<V: Vegetable> {  
    veggies: Vec<V>  
}
```

然而，这是一种相当严格的设计：每一样沙拉都要完全由一种蔬菜组成。但毕竟众口难调。本书的一位作者就曾花 14 美元买过一份 `Salad<IcebergLettuce>`（球形生菜沙拉），那味道真是让人终生难忘。

怎样才能做出更好的沙拉呢？由于 `Vegetable` 的值可以有各种不同的大小，因此不能要求 Rust 接受 `Vec<dyn Vegetable>` 这种写法：

```
struct Salad {  
    veggies: Vec<dyn Vegetable> // 错误：`dyn Vegetable` 的大小不是常量  
}
```

解决方案是使用特型对象：

```
struct Salad {  
    veggies: Vec<Box<dyn Vegetable>>  
}
```

每个 `Box<dyn Vegetable>` 都可以拥有任意类型的蔬菜，但 `Box` 本身具有适合存储在向量中的常量大小（两个指针）。虽然把盒子（`Box`）放进某个食物里这种隐喻怪怪的，但这正是我们想要的，这种方式同样适用于绘图应用程序中的形状、游戏中的怪物、网络路由器中的可插接路由算法等。

使用特型对象的另一个原因可能是想减少编译后代码的总大小。Rust 可能会不得不多次编译泛型函数，针对用到了它的每种类型各编译一次。而这可能会使二进制文件变大，这种现象在 C++ 圈子里叫作**代码膨胀**。如今，内存资源充裕，大多数人可以不在乎代码大小，但确实仍然存在着某种受限环境。

除了像制作沙拉这种问题或是低资源环境之类的情况，与特型对象相比，泛型具有 3 个重要优势，因此在 Rust 中，泛型是更常见的选择。

第一个优势是速度。请注意泛型函数签名中缺少 `dyn` 关键字。因为你要在编译期指定类型，所以无论是显式写出还是通过类型推断，编译

器都知道要调用哪个 `write` 方法。没有使用 `dyn` 关键字是因为这里不涉及特型对象（因此也不涉及动态派发）。

本章开头展示过的泛型函数 `min()` 运行起来与我们编写的单独的函数 `min_u8`、`min_i64`、`min_string` 等一样快。编译器可以像任何函数一样将其内联，因此在发布构建中，调用 `min::<i32>` 可能只有两三条指令。参数是常数的调用（如 `min(5, 3)`）会更快：Rust 可以在编译期对其求值，因此根本没有运行期开销。

或者考虑这个泛型函数调用：

```
let mut sink = std::io::sink();
say_hello(&mut sink)?;
```

`std::io::sink()` 会返回类型为 `Sink` 的写入器，该写入器会悄悄地丢弃写入其中的所有字节。

当 Rust 为此生成机器码时，它可以生成调用 `Sink::write_all` 的代码，检查错误，然后调用 `Sink::flush`。这就是泛型函数体所要求做的。

Rust 还可以查看这些方法并注意到：

- `Sink::write_all()` 什么都不做；
- `Sink::flush()` 什么都不做；
- 这两种方法都不会返回错误。

简而言之，Rust 拥有完全优化此函数调用所需的全部信息。

这与特型对象的行为不同。在那种方式下，Rust 直到运行期才能知道特型对象指向什么类型的值。因此，即使你传递了 `Sink`，调用虚方法和检查错误的开销仍然存在。

泛型的第二个优势在于并不是每个特型都能支持特型对象。特型支持的几个特性（如关联函数）只适用于泛型：它们完全不支持特型对象。我们会在谈及这些特性时指出这一点。

泛型的第三个优势是它很容易同时指定具有多个特型的泛型参数界限，就像我们的 `top_ten` 函数要求它的 `T` 参数必须实现 `Debug + Hash + Eq` 那样。特型对象不能这样做：Rust 不支持像 `&mut (dyn Debug + Hash + Eq)` 这样的类型。（你可以使用本章稍后定义的特型来解决这个问题，但有点儿复杂。）

11.2 定义与实现特型

定义特型很简单，给它一个名字并列出特型方法的类型签名即可。如果你正在编写游戏，那么可能会有这样的特型：

```
/// 角色、道具和风景的特型——游戏世界中可在屏幕上看见的任何东西
trait Visible {
    /// 在给定的画布上渲染此对象
    fn draw(&self, canvas: &mut Canvas);

    /// 如果单击(x, y)时应该选中此对象，就返回true
    fn hit_test(&self, x: i32, y: i32) -> bool;
}
```

要实现特型，请使用语法 `impl TraitName for Type`：

```
impl Visible for Broom {
    fn draw(&self, canvas: &mut Canvas) {
        for y in self.y - self.height - 1 .. self.y {
            canvas.write_at(self.x, y, '|');
        }
        canvas.write_at(self.x, self.y, 'M');
    }

    fn hit_test(&self, x: i32, y: i32) -> bool {
        self.x == x
        && self.y - self.height - 1 <= y
        && y <= self.y
    }
}
```

请注意，这个 `impl` 包含 `Visible` 特型中每个方法的实现，再无其他。特型的 `impl` 代码中定义的一切都必须真正属于此特型的，如果想添加一个辅助方法来支持 `Broom::draw()`，就必须在单独的 `impl` 块中定义它：

```
impl Broom {
    /// 供下面的Broom::draw()使用的辅助函数
    fn broomstick_range(&self) -> Range<i32> {
        self.y - self.height - 1 .. self.y
    }
}
```

```
}  
}
```

这些辅助函数可以在特型的各个 `impl` 块中使用。

```
impl Visible for Broom {  
    fn draw(&self, canvas: &mut Canvas) {  
        for y in self.broomstick_range() {  
            ...  
        }  
        ...  
    }  
    ...  
}
```

11.2.1 默认方法

我们之前讨论的 `Sink` 写入器类型可以用几行代码来实现。首先，定义如下类型：

```
/// 一个会忽略你写入的任何数据的写入器  
pub struct Sink;
```

`Sink` 是一个空结构体，因为我们不需要在其中存储任何数据。接下来，为 `Sink` 提供 `Write` 特型的实现：

```
use std::io::{Write, Result};  
  
impl Write for Sink {  
    fn write(&mut self, buf: &[u8]) -> Result<usize> {  
        // 声称已成功写入了整个缓冲区  
        Ok(buf.len())  
    }  
  
    fn flush(&mut self) -> Result<()> {  
        Ok(())  
    }  
}
```

迄今为止，这种写法和 `Visible` 特型非常相似。但我们看到 `Write` 特型中还有一个 `write_all` 方法：

```
let mut out = Sink;
out.write_all(b"hello world\n"?);
```

为什么 Rust 允许在未定义此方法的情况下 `impl Write for Sink` 呢？答案是标准库的 `Write` 特型定义中包含了对 `write_all` 的默认实现：

```
trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()> {
        let mut bytes_written = 0;
        while bytes_written < buf.len() {
            bytes_written += self.write(&buf[bytes_written..])?;
        }
        Ok(())
    }

    ...
}
```

`write` 方法和 `flush` 方法是每个写入器必须实现的基本方法。写入器也可以自行实现 `write_all`，但如果没实现，就会使用前面展示过的默认实现。

你的自定义特型也可以包含一些使用同样语法的默认实现。

标准库中对默认方法最引人注目的应用场景是 `Iterator` 特型，它有一个必要方法 (`.next()`) 和几十个默认方法。第 15 章会解释原因。

11.2.2 特型与其他人的类型

Rust 允许在任意类型上实现任意特型，但特型或类型二者必须至少有一个是在当前 `crate` 中新建的。

这意味着任何时候如果你想为任意类型添加一个方法，都可以使用特型来完成：

```
trait IsEmoji {
    fn is_emoji(&self) -> bool;
}

/// 为内置的字符类型实现IsEmoji特型
impl IsEmoji for char {
    fn is_emoji(&self) -> bool {
        ...
    }
}

assert_eq!('$'.is_emoji(), false);
```

与任何其他特型方法一样，只有当 `IsEmoji` 在作用域内时这个新的 `is_emoji` 方法才是可见的。

这个特殊特型的唯一目的是向现有类型 `char` 中添加一个方法。这称为扩展特型。当然，也可以通过 `impl IsEmoji for str { ... }` 等写法将此特型添加到其他类型中。

甚至可以使用一个泛型的 `impl` 块来一次性向整个类型家族添加扩展特型。这个特型可以在任意类型上实现：

```
use std::io::{self, Write};

/// 能让你把HTML写入值里的特型
trait WriteHtml {
    fn write_html(&mut self, html: &HtmlDocument) ->
    io::Result<()>;
}
```

为所有写入器实现特型会令其成为扩展特型，比如为所有 `Rust` 写入器添加一个方法：

```
/// 可以把HTML写入任何一个std::io writer
impl<W: Write> WriteHtml for W {
    fn write_html(&mut self, html: &HtmlDocument) ->
    io::Result<()> {
```

```
    ...  
    }  
}
```

`impl<W: Write> WriteHtml for W` 这一行的意思是“对于每个实现了 `Write` 的类型 `W`，这里有一个适用于 `W` 的 `WriteHtml` 实现”。

`serde` 库提供了一个很好的例子，表明在标准类型上实现用户定义的特型非常有用。`serde` 是一个序列化库。也就是说，你可以使用 `serde` 将 `Rust` 数据结构写入磁盘，稍后再重新加载它们。这个库定义了一个特型 `Serialize`，它为该库支持的每种数据类型都提供了实现。因此在 `serde` 源代码中，有一些代码为 `bool`、`i8`、`i16`、`i32`、数组类型和元组类型等内置类型，以及像 `Vec` 和 `HashMap` 这样的标准数据结构实现了 `Serialize`。

所有这些工作的成果，就是 `serde` 为这些类型添加了一个 `.serialize()` 方法。它可以这样使用：

```
use serde::Serialize;  
use serde_json;  
  
pub fn save_configuration(config: &HashMap<String, String>)  
    -> std::io::Result<()>  
{  
    // 创建一个JSON序列化器以把数据写入文件  
    let writer = File::create(config_filename())?;  
    let mut serializer = serde_json::Serializer::new(writer);  
  
    // serde的`.serialize()`方法会完成剩下的工作  
    config.serialize(&mut serializer)?;  
    Ok(())  
}
```

我们之前说过，在实现特型时，特型或类型二者必须至少有一个是在当前 `crate` 中新建的。这叫作**孤儿规则**。它会帮助 `Rust` 确保特型的实现是唯一的。你的代码不能写成 `impl Write for u8`，因为 `Write` 和 `u8` 都是在标准库中定义的。如果 `Rust` 允许 `crate` 这样

做，那么在不同的 crate 中可能会有多个 u8 的 Write 实现，而 Rust 并没有合理的方法来决定把哪个实现用于给定的方法调用。

（C++ 有一个类似的唯一性限制：单一定义规则。在典型的 C++ 流派中，除了一些最简单的情况，编译器不会强制执行此规则。如果你打破了此规则，就会得到未定义行为。）

11.2.3 特型中的 **Self**

特型可以用关键字 `Self` 作为类型。例如，标准库的 `Clone` 特型看起来是这样的（稍作简化）：

```
pub trait Clone {  
    fn clone(&self) -> Self;  
    ...  
}
```

这里以 `Self` 作为返回类型意味着 `x.clone()` 的类型与 `x` 的类型相同，无论 `x` 是什么。如果 `x` 是 `String`，那么 `x.clone()` 的类型也必须是 `String`——而不能是 `dyn Clone` 或其他可克隆类型。

同样，如果我们定义如下特型

```
pub trait Spliceable {  
    fn splice(&self, other: &Self) -> Self;  
}
```

并有两个实现：

```
impl Spliceable for CherryTree {  
    fn splice(&self, other: &Self) -> Self {  
        ...  
    }  
}  
  
impl Spliceable for Mammoth {  
    fn splice(&self, other: &Self) -> Self {  
        ...  
    }  
}
```

```
}  
}
```

那么在第一个 `impl` 中，`Self` 只是 `CherryTree` 的别名，而在第二个 `impl` 中，它是 `Mammoth` 的别名。这意味着可以将两棵樱桃树或两头猛犸象拼接在一起，但不表示可以创造出猛犸象和樱桃树的混合体。`self` 的类型和 `other` 的类型必须匹配。

使用了 `Self` 类型的特型与特型对象不兼容：

```
// 错误：特型`Spliceable`不能用作特型对象  
fn splice_anything(left: &dyn Spliceable, right: &dyn Spliceable)  
{  
    let combo = left.splice(right);  
    // ...  
}
```

至于其原因，当我们深入研究特型的高级特性时还会一次又一次看到。Rust 会拒绝此代码，因为它无法对 `left.splice(right)` 这个调用进行类型检查。特型对象的全部意义恰恰在于其类型要到运行期才能知道。Rust 在编译期无从了解 `left` 和 `right` 是否为同一类型。

特型对象实际上是为最简单的特型类型而设计的，这些类型都可以使用 Java 中的接口或 C++ 中的抽象基类来实现。特型的高级特性很有用，但它们不能与特型对象共存，因为一旦有了特型对象，就会失去 Rust 对你的程序进行类型检查时所必需的类型信息。

现在，如果想要实现这种在遗传意义上不可能的拼接，可以设计一个对特型对象友好的特型：

```
pub trait MegaSpliceable {  
    fn splice(&self, other: &dyn MegaSpliceable) -> Box<dyn  
MegaSpliceable>;  
}
```

此特型与特型对象兼容。对 `.splice()` 方法的调用可以通过类型检查，因为参数 `other` 的类型不需要匹配 `self` 的类型，只要这两种

类型都是 `MegaSpliceable` 就可以了。

11.2.4 子特型

我们可以声明一个特型是另一个特型的扩展：

```
/// 游戏世界中的生物，既可以是玩家，也可以是
/// 其他小精灵、石像鬼、松鼠、食人魔等
trait Creature: Visible {
    fn position(&self) -> (i32, i32);
    fn facing(&self) -> Direction;
    ...
}
```

短语 `trait Creature : Visible` 表示所有生物都是可见的。每个实现了 `Creature` 的类型也必须实现 `Visible` 特型：

```
impl Visible for Broom {
    ...
}

impl Creature for Broom {
    ...
}
```

可以按任意顺序实现这两个特型，但是如果不为类型实现 `Visible` 只为其实现 `Creature` 则是错误的。在这里，我们说 `Creature` 是 `Visible` 的**子特型**，而 `Visible` 是 `Creature` 的**超特型**。

子特型与 Java 或 C# 中的子接口类似，因为用户可以假设实现了子特型的任何值也会实现其超特型。但是在 Rust 中，子特型不会继承其超特型的关联项，如果你想调用超特型的方法，那么仍然要保证每个特型都在作用域内。

事实上，Rust 的子特型只是对 `Self` 类型限界的简写。像下面这样的 `Creature` 定义与前面的定义完全等效。

```
trait Creature where Self: Visible {
    ...
}
```

```
}
```

11.2.5 类型关联函数

在大多数面向对象语言中，接口不能包含静态方法或构造函数，但特型可以包含类型关联函数，这是 Rust 对静态方法的模拟：

```
trait StringSet {  
    /// 返回一个新建的空集合  
    fn new() -> Self;  
  
    /// 返回一个包含`strings`中所有字符串的集合  
    fn from_slice(strings: &[&str]) -> Self;  
  
    /// 判断这个集合中是否包含特定的`string`  
    fn contains(&self, string: &str) -> bool;  
  
    /// 把一个字符串添加到此集合中  
    fn add(&mut self, string: &str);  
}
```

每个实现了 StringSet 特型的类型都必须实现这 4 个关联函数。前两个函数，即 new() 和 from_slice()，不接受 self 参数。它们扮演着构造函数的角色。在非泛型代码中，可以使用 :: 语法调用这些函数，就像调用任何其他类型关联函数一样：

```
// 创建实现了StringSet的两个假想集合类型：  
let set1 = SortedStringSet::new();  
let set2 = HashedStringSet::new();
```

在泛型代码中，也可以使用 :: 语法，不过其类型部分通常是类型变量，如下面对 S::new() 的调用所示：

```
/// 返回`document`中不存在于`wordlist`中的单词集合  
fn unknown_words<S: StringSet>(document: &[String], wordlist: &S)  
-> S {  
    let mut unknowns = S::new();  
    for word in document {  
        if !wordlist.contains(word) {  
            unknowns.add(word);  
        }  
    }  
}
```

```
}  
    unknowns  
}
```

与 Java 接口和 C# 接口一样，特型对象也不支持类型关联函数。如果想使用 `&dyn StringSet` 特型对象，就必须修改此特型，为每个未通过引用接受 `self` 参数的关联函数加上类型限界 `where Self: Sized`:

```
trait StringSet {  
    fn new() -> Self  
        where Self: Sized;  
  
    fn from_slice(strings: &[&str]) -> Self  
        where Self: Sized;  
  
    fn contains(&self, string: &str) -> bool;  
  
    fn add(&mut self, string: &str);  
}
```

这个限界告诉 Rust，特型对象不需要支持特定的关联函数¹。通过添加这些限界，就能把 `StringSet` 作为特型对象使用了。虽然特型对象仍不支持关联函数 `new` 或 `from_slice`，但你还是可以创建它们并用其调用 `.contains()` 和 `.add()`。同样的技巧也适用于其他与特型对象不兼容的方法。（我们暂且放弃对“为何这么改就行”的枯燥技术解释，但会在第 13 章介绍一下 `Sized` 特型，届时你就懂了。）

¹指带有 `:Sized` 限界的 `new` 和 `from_slice` 这两个关联函数。——译者注

11.3 完全限定的方法调用

迄今为止，我们看到的所有调用特型方法的方式都依赖于 Rust 为你补齐了一些缺失的部分。假设你编写了以下内容：

```
"hello".to_string()
```

Rust 知道 `to_string` 指的是 `ToString` 特型的 `to_string` 方法（我们称之为 `str` 类型的实现）。所以这个游戏里有 4 个“玩家”：特型、特型的方法、方法的实现以及调用该实现时传入的值。很高兴我们不必在每次调用方法时都把它们完全写出来。但在某些情况下，你需要一种方式来准确表达你的意思。完全限定的方法调用符合此要求。

首先，要知道方法只是一种特殊的函数。下面两个调用是等效的：

```
"hello".to_string()  
str::to_string("hello")
```

第二种形式看起来很像关联函数调用。尽管 `to_string` 方法需要一个 `self` 参数，但是仍然可以像关联函数一样调用。只需将 `self` 作为此函数的第一个参数传进去即可。

由于 `to_string` 是标准 `ToString` 特型的方法之一，因此你还可以使用另外两种形式：

```
ToString::to_string("hello")  
<str as ToString>::to_string("hello")
```

所有这 4 种方法调用都会做同样的事情。大多数情况下，只要写 `value.method()` 就可以了。其他形式都是限定方法调用。它们要指定方法所关联的类型或特型。最后一种带有尖括号的形式，同时指定了两者，这就是完全限定的方法调用。

当你写下 `"hello".to_string()` 时，使用的是 `.` 运算符，你并没有确切说明要调用哪个 `to_string` 方法。Rust 有一个“方法查找”算法，它可以根据类型、隐式解引用等来解决这个问题。通过完全限定的调用，你可以准确地指出是哪一个方法，这在一些奇怪的情况下会有所帮助。

- 当两个方法具有相同的名称时。生拼硬凑的经典示例是 `Outlaw`（亡命之徒），它具有来自不同特型的两个 `.draw()` 方法，一个用于将其绘制在屏幕上，另一个用于犯罪。

```
outlaw.draw(); // 错误：画（draw）在屏幕上还是拔出（draw）手枪？

Visible::draw(&outlaw); // 正确：画在屏幕上
HasPistol::draw(&outlaw); // 正确：拔出手枪
```

通常你可以对其中一个方法改名，但有时实在没法改。

- 当无法推断 `self` 参数的类型时。

```
let zero = 0; // 类型未指定：可能为`i8`、`u8`.....

zero.abs(); // 错误：无法在有歧义的数值类型上调用方法`abs`

i64::abs(zero); // 正确
```

- 将函数本身用作函数类型的值时。

```
let words: Vec<String> =
    line.split_whitespace() // 迭代器生成&str值
        .map(ToString::to_string) // 正确
        .collect();
```

- 在宏中调用特型方法时。第 21 章会对此进行解释。

完全限定语法也适用于关联函数。在 11.2.5 节中，我们编写了 `S::new()` 以在泛型函数中创建一个新集合。也可以写成 `StringSet::new()` 或 `<S as StringSet>::new()`。

11.4 定义类型之间关系的特型

迄今为止，我们看到的每个特型都是独立的：特型是类型可以实现的一组方法。特型也可以用于多种类型必须协同工作的场景中。它们可以描述多个类型之间的关系。

- `std::iter::Iterator` 特型会为每个迭代器类型与其生成的值的类型建立联系。
- `std::ops::Mul` 特型与可以相乘的类型有关。在表达式 `a * b` 中，值 `a` 和 `b` 可以是相同类型，也可以是不同类型。
- `rand crate` 中包含随机数生成器的特型（`rand::Rng`）和可被随机生成的类型的特型（`rand::Distribution`）。这些特型本身就准确地定义了它们是如何协同工作的。

你不需要每天都创建这样的特型，但它们在标准库和第三方 `crate` 中随处可见。本节将展示每一个示例是如何实现的，并根据需要来展开讲解相关的 Rust 语言特性。你需要掌握的关键技能是能够阅读这些特型和方法签名，并理解它们对所涉及的类型意味着什么。

11.4.1 关联类型（或迭代器的工作原理）

接下来我们从迭代器讲起。迄今为止，每种面向对象的语言都内置了某种对迭代器的支持，迭代器是用以遍历某种值序列的对象。

Rust 有一个标准的 `Iterator` 特型，定义如下：

```
pub trait Iterator {  
    type Item;  
  
    fn next(&mut self) -> Option<Self::Item>;  
    ...  
}
```

这个特型的第一个特性（`type Item;`）是一个关联类型。实现了 `Iterator` 的每种类型都必须指定它所生成的条目的类型。

第二个特性 (`next()` 方法) 在其返回值中使用了关联类型。`next()` 会返回一个 `Option<Self::Item>`: 或者是序列中的下一个值 `Some(item)`, 或者当没有更多值可供访问时返回 `None`。该类型要写成 `Self::Item`, 而不仅仅是无修饰的 `Item`, 因为这里的 `Item` 是每个迭代器类型下的一个特性, 而不是一个独立的类型。同样, `self` 和 `Self` 类型在代码中任何使用了其字段、方法等的地方都要像这样显式写出来。

下面是为一个类型实现 `Iterator` 的范例:

```
// (来自标准库中std::env模块的代码)
impl Iterator for Args {
    type Item = String;

    fn next(&mut self) -> Option<String> {
        ...
    }
    ...
}
```

`std::env::Args` 是我们在第 2 章中用来访问命令行参数的标准库函数 `std::env::args()` 返回的迭代器类型。它能生成 `String` 值, 因此这个 `impl` 声明了 `type Item = String;`。

泛型代码可以使用关联类型:

```
/// 遍历迭代器, 将值存储在新向量中
fn collect_into_vector<I: Iterator>(iter: I) -> Vec<I::Item> {
    let mut results = Vec::new();
    for value in iter {
        results.push(value);
    }
    results
}
```

在这个函数体中, Rust 为我们推断出了 `value` 的类型, 这固然不错, 但我们还必须明确写出 `collect_into_vector` 的返回类型, 而 `Item` 关联类型是唯一的途径。(用 `Vec<I>` 肯定不对, 因为那样是在宣告要返回一个由迭代器组成的向量。)

前面示例中的代码并不需要你亲自编写，因为在阅读第 15 章之后，你会知道迭代器已经有了一个执行此操作的标准方法：`iter.collect()`。在继续之前，再来看一个例子：

```
/// 打印出迭代器生成的所有值
fn dump<I>(iter: I)
  where I: Iterator
{
  for (index, value) in iter.enumerate() {
    println!("{}", index, value);    // 错误
  }
}
```

这几乎已经改好了。但还有一个问题：`value` 不一定是可打印的类型。

```
error: `::Item` doesn't implement `Debug`
|
8 |         println!("{}", index, value);    // 错误
|                                     ^^^^^
|                                     `::Item` cannot be
formatted
|                                     using `{:?}` because it doesn't
implement `Debug`
|
  = help: the trait `Debug` is not implemented for `::Item`
  = note: required by `std::fmt::Debug::fmt`
help: consider further restricting the associated type
|
5 |         where I: Iterator, <I as Iterator>::Item: Debug
|                                     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

由于 Rust 使用了语法 `<I as Iterator>::Item` (`I::Item` 的一种明确但冗长的说法)，因此错误消息被弄得稍微有点儿乱。虽然这是有效的 Rust 语法，但实际上你很少需要以这种方式书写类型。

这个错误消息的要点是，要编译此泛型函数，就必须确保 `I::Item` 实现了 `Debug` 特型，也就是用 `{:?}` 格式化值时要求的特型。正如错误消息所暗示的那样，我们可以通过在 `I::Item` 上设置一个限界来做到这一点：

```
use std::fmt::Debug;

fn dump<I>(iter: I)
    where I: Iterator, I::Item: Debug
{
    ...
}
```

或者，我们可以说“I 必须是指对 String 值的迭代器”：

```
fn dump<I>(iter: I)
    where I: Iterator<Item=String>
{
    ...
}
```

`Iterator<Item=String>` 本身就是一个特型。如果将 `Iterator` 视为所有迭代器类型的全集，那么 `Iterator<Item=String>` 就是 `Iterator` 的子集：生成 `String` 的迭代器类型的集合。此语法可用于任何可以使用特型名称（包括特型对象类型）的地方：

```
fn dump(iter: &mut dyn Iterator<Item=String>) {
    for (index, s) in iter.enumerate() {
        println!("{index}: {s}", index, s);
    }
}
```

具有关联类型的特型（如 `Iterator`）与特型对象是兼容的，但前提是要把所有关联类型都明确写出来，就像此处所展示的那样。否则，`s` 的类型可能是任意类型，同样，这会导致 Rust 无法对这段代码进行类型检查。

我们已经展示了很多涉及迭代器的示例。之所以花了这么多篇幅，是因为迭代器是迄今为止使用关联类型的最主要场景。但当特型需要包含的不仅仅是方法的时候，关联类型会很有用。

- 在线程池中，`Task` 特型表示一个工作单元，它可以包含一个关联的 `Output` 类型。

- Pattern 特型表示一种搜索字符串的方式，它可以包含一个关联的 Match 类型，后者表示将模式与字符串匹配后收集到的所有信息。

```
trait Pattern {
    type Match;

    fn search(&self, string: &str) -> Option<Self::Match>;
}

/// 你可以在字符串中找一个特定的字符
impl Pattern for char {
    /// Match只是找到的字符的位置
    type Match = usize;

    fn search(&self, string: &str) -> Option<usize> {
        ...
    }
}
```

如果你熟悉正则表达式，那么很容易看出 `impl Pattern for RegExp` 应该有一个更精细的 Match 类型，它可能是一个结构体，其中包含此匹配的开始位置和长度、圆括号组匹配的位置等。

- 用于处理关系型数据库的库可能具有 `DatabaseConnection` 特型，其关联类型表示事务、游标、已准备语句等。

关联类型非常适合每个实现都有一个特定相关类型的情况：每种类型的 `Task` 都会生成特定类型的 `Output`，每种类型的 `Pattern` 都会寻找特定类型的 `Match`。然而，如你所见，类型之间的某些关系并不是这样的。

11.4.2 泛型特型（或运算符重载的工作原理）

Rust 中的乘法使用了以下特型：

```
/// std::ops::Mul, 用于标记支持`*`（乘号）的类型的特型
pub trait Mul<RHS> {
    /// 在应用了`*`运算符后的结果类型
    type Output;

    /// 实现`*`运算符的方法
    fn mul(self, rhs: RHS) -> Self::Output;
}
```

Mul 是一个泛型特型。类型参数 RHS 是右操作数（right-hand side）的缩写。

这里的类型参数与它在结构体或函数上的含义是一样的：Mul 是一个泛型特型，它的实例 Mul<f64>、Mul<String>、Mul<Size> 等都是不同的特型，就像 min::<i32> 和 min::<String> 是不同的函数，Vec<i32> 和 Vec<String> 是不同的类型一样。

单一类型（如 WindowSize）可以同时实现 Mul<f64>、Mul<i32>，等等。然后你就可以将 WindowSize 乘以许多其他类型。每个实现都有自己关联的 Output 类型。

泛型特型在涉及孤儿规则时会得到特殊豁免：你可以为外部类型实现外部特型，只要特型的类型参数之一是当前 crate 中定义的类型即可。因此，如果你自己定义了 WindowSize，则可以为 f64 实现 Mul<WindowSize>，即使你既没有定义 Mul 也没有定义 f64。这些实现甚至可以是泛型的，比如 impl<T> Mul<WindowSize> for Vec<T>。这是可行的，因为其他 crate 不可能在任何东西上定义 Mul<WindowSize>，因此实现之间不可能出现冲突。（11.2.2 节介绍过孤儿规则。）这就是像 nalgebra 这样的 crate 能为向量定义算术运算的原理。

前面展示的特型缺少一个小细节。真正的 Mul 特型是这样的：

```
pub trait Mul<RHS=Self> {
    ...
}
```

语法 `RHS=Self` 表示 `RHS` 默认为 `Self`。如果我写下 `impl Mul for Complex`，而不指定 `Mul` 的类型参数，则表示 `impl Mul<Complex> for Complex`。在类型限界中，如果我写下 `where T: Mul`，则表示 `where T: Mul<T>`。

在 Rust 中，表达式 `lhs * rhs` 是 `Mul::mul(lhs, rhs)` 的简写形式。所以在 Rust 中重载 `*` 运算符就像实现 `Mul` 特型一样简单。第 12 章会展示相关示例。

11.4.3 `impl Trait`

如你所料，由许多泛型类型组合而成的结果可能会极其凌乱。例如，使用标准库的组合器组合上几个迭代器，就会迅速把你的返回类型变成一个“丑八怪”：

```
use std::iter;
use std::vec::IntoIter;
fn cyclical_zip(v: Vec<u8>, u: Vec<u8>) ->
    iter::Cycle<iter::Chain<IntoIter<u8>, IntoIter<u8>>> {
    v.into_iter().chain(u.into_iter()).cycle()
}
```

我们可以很容易地用特型对象替换这个“丑陋的”返回类型：

```
fn cyclical_zip(v: Vec<u8>, u: Vec<u8>) -> Box<dyn
    Iterator<Item=u8>> {
    Box::new(v.into_iter().chain(u.into_iter()).cycle())
}
```

然而，在大多数情况下，如果仅仅是为了避免“丑陋的”类型签名，就要在每次调用这个函数时承受动态派发和不可避免的堆分配开销，可不太划算。

Rust 有一个名为 `impl Trait` 的特性，该特性正是为应对这种情况而设计的。`impl Trait` 允许我们“擦除”返回值的类型，仅指定它实现的一个或多个特型，而无须进行动态派发或堆分配：

```
fn cyclical_zip(v: Vec<u8>, u: Vec<u8>) -> impl Iterator<Item=u8>
{
    v.into_iter().chain(u.into_iter()).cycle()
}
```

现在，`cyclical_zip` 的签名中再也没有那种带着迭代器组合结构的嵌套类型了，而只是声明它会返回某种 `u8` 迭代器。返回类型表达了函数的意图，而非实现细节。

这无疑清理了代码并提高了可读性，但 `impl Trait` 可不止是一个方便的简写形式。使用 `impl Trait` 意味着你将来可以更改返回的实际类型，只要返回类型仍然会实现 `Iterator<Item=u8>`，调用该函数的任何代码就能继续编译而不会出现问题。这就为库作者提供了很大的灵活性，因为其类型签名中只编码了有意义的功能。

如果库的第一个版本像前面那样使用迭代器的组合器，但后来发现了针对同一过程的更好算法，则库作者可能会改用不同的组合器，甚至会创建一个能实现 `Iterator` 的自定义类型，但只要当初使用了 `impl Trait` 来编写签名，库的用户根本不必更改代码就能获得性能改进。

使用 `impl Trait` 来为面向对象语言中常用的工厂模式模仿出一个静态派发版本可能是个诱人的想法。例如，你可能会想定义如下特型：

```
trait Shape {
    fn new() -> Self;
    fn area(&self) -> f64;
}
```

在为几种类型实现了 `Shape` 之后，你可能希望根据某个运行期的值（比如用户输入的字符串）使用不同的 `Shape`。但以 `impl Shape` 作为返回类型并不能实现这一目标：

```
fn make_shape(shape: &str) -> impl Shape {
    match shape {
        "circle" => Circle::new(),
        "triangle" => Triangle::new(), // 错误：不兼容的类型
    }
}
```

```
        "shape" => Rectangle::new(),
    }
}
```

从调用者的角度来看，这样的函数没有多大意义。`impl Trait` 是一种静态派发形式，因此编译器必须在编译期就知道从函数返回的类型，以便在栈上分配正确的空间数量并正确访问该类型的字段和方法。在这里，这个类型可能是 `Circle`、`Triangle` 或 `Rectangle`，它们可能占用不同的空间大小，并且有着不同的 `area()` 实现。

这里的要点是，Rust 不允许特型方法使用 `impl Trait` 作为返回值。要支持这个特性，就要对语言的类型系统进行一些改进。在这项工作完成之前，只有自由函数和关联具体类型的函数才能使用 `impl Trait` 作为返回值。

`impl Trait` 也可以用在带有泛型参数的函数中。例如，考虑下面这个简单的泛型函数：

```
fn print<T: Display>(val: T) {
    println!("{}", val);
}
```

它与使用 `impl Trait` 的版本完全相同：

```
fn print(val: impl Display) {
    println!("{}", val);
}
```

但有一个重要的例外。使用泛型时允许函数的调用者指定泛型参数的类型，比如 `print::<i32>(42)`，而如果使用 `impl Trait` 则不能这样做。

每个 `impl Trait` 参数都被赋予了自己独有的匿名类型参数，因此，只有在最简单的泛型函数中才能把 `impl Trait` 参数用作类型，参数的类型之间不能存在关系。

11.4.4 关联常量

与结构体和枚举一样，特型也可以有关联常量。你可以使用与结构体或枚举相同的语法来声明带有关联常量的特型：

```
trait Greet {  
    const GREETING: &'static str = "Hello";  
    fn greet(&self) -> String;  
}
```

不过，关联常量在特型中具有特殊的功能。与关联类型和函数一样，你也可以声明它们，但不为其定义值：

```
trait Float {  
    const ZERO: Self;  
    const ONE: Self;  
}
```

之后，特型的实现者可以定义这些值：

```
impl Float for f32 {  
    const ZERO: f32 = 0.0;  
    const ONE: f32 = 1.0;  
}  
  
impl Float for f64 {  
    const ZERO: f64 = 0.0;  
    const ONE: f64 = 1.0;  
}
```

你可以编写使用这些值的泛型代码：

```
fn add_one<T: Float + Add<Output=T>>(value: T) -> T {  
    value + T::ONE  
}
```

请注意，关联常量不能与特型对象一起使用，因为为了在编译期选择正确的值，编译器会依赖相关实现的类型信息。

即使是没有任何行为的简单特型（如 `Float`），也可以提供有关类型的足够信息，再结合一些运算符，以实现像斐波那契数列这样常见的数学函数：

```
fn fib<T: Float + Add<Output=T>>(n: usize) -> T {  
    match n {  
        0 => T::ZERO,  
        1 => T::ONE,  
        n => fib::<T>(n - 1) + fib::<T>(n - 2)  
    }  
}
```

在 11.5 节和 11.6 节中，我们将展示用特型描述类型之间关系的不同方式。所有这些都可以看作避免虚方法开销和向下转换的方法，因为它们允许 Rust 在编译期了解更多的具体类型。

11.5 逆向工程求限界

当没有特型可以满足你的所有需求时，编写泛型代码可能会是一件真正的苦差事。假设我们编写了下面这个非泛型函数来进行一些计算：

```
fn dot(v1: &[i64], v2: &[i64]) -> i64 {
    let mut total = 0;
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

现在我们对浮点值使用相同的代码，因此可能会尝试像下面这么做。

```
fn dot<N>(v1: &[N], v2: &[N]) -> N {
    let mut total: N = 0;
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

运气不佳：Rust 会报错说乘法（*）的使用以及 0 的类型有问题。我们可以使用 Add 和 Mul 的特型要求 N 是支持 + 和 * 的类型。但是，对 0 的用法需要改变，因为 0 在 Rust 中始终是一个整数，对应的浮点值为 0.0。幸运的是，对于具有默认值的类型，有一个标准的 Default 特型。对于数值类型，默认值始终为 0：

```
use std::ops::{Add, Mul};

fn dot<N: Add + Mul + Default>(v1: &[N], v2: &[N]) -> N {
    let mut total = N::default();
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

离成功更近了，但仍未完全解决：

```
error: mismatched types
  |
5 | fn dot<N: Add + Mul + Default>(v1: &[N], v2: &[N]) -> N {
  |           - this type parameter
...
8 |         total = total + v1[i] * v2[i];
  |                               ^^^^^^^^^^^^^^^^^ expected type parameter
`N`,
  |                               found associated type
  |
  = note: expected type parameter `N`
           found associated type `::Output`
help: consider further restricting this bound
  |
5 | fn dot<N: Add + Mul + Default + Mul<Output = N>>(v1: &[N], v2:
  | &[N]) -> N {
  |                               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

我们的新代码中假定将两个 `N` 类型的值相乘会生成另一个 `N` 类型的值。但事实并非如此。你可以重载乘法运算符以返回想要的任意类型。我们需要以某种方式让 Rust 知道这个泛型函数只适用于那些支持正常乘法规范的类型，其中 `N * N` 一定会返回 `N`。错误消息中的建议几乎是正确的：我们可以通过将 `Mul` 替换为 `Mul<Output=N>` 来做到这一点，对 `Add` 的处理也是一样的：

```
fn dot<N: Add<Output=N> + Mul<Output=N> + Default>(v1: &[N], v2: &[N]) -> N
{
    ...
}
```

此时，类型限界积累得越来越多，使得代码难以阅读。我们来把限界移动到 `where` 子句中：

```
fn dot<N>(v1: &[N], v2: &[N]) -> N
    where N: Add<Output=N> + Mul<Output=N> + Default
{
    ...
}
```

漂亮！但是 Rust 仍然会对这行代码报错：

```
error: cannot move out of type `[N]`, a non-copy slice
  |
8 |         total = total + v1[i] * v2[i];
  |                               ^^^^^
  |                               |
  |                               cannot move out of here
  |                               move occurs because `v1[_]` has type
  |                               `[N]`,
  |                               which does not implement the `Copy`
  |                               trait
```

由于我们没有要求 `N` 是可复制的类型，因此 Rust 将 `v1[i]` 解释为试图将值从切片中移动出去（Rust 中禁止这样做）。但是我们根本就没想修改切片，只是想将这些值复制出来以便对它们进行操作。幸运的是，Rust 的所有内置数值类型都实现了 `Copy`，因此可以简单地将其添加到对 `N` 的约束中：

```
where N: Add<Output=N> + Mul<Output=N> + Default + Copy
```

这样代码就可以编译并运行了。最终代码如下所示：

```
use std::ops::{Add, Mul};

fn dot<N>(v1: &[N], v2: &[N]) -> N
    where N: Add<Output=N> + Mul<Output=N> + Default + Copy
{
    let mut total = N::default();
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}

#[test]
fn test_dot() {
    assert_eq!(dot(&[1, 2, 3, 4], &[1, 1, 1, 1]), 10);
    assert_eq!(dot(&[53.0, 7.0], &[1.0, 5.0]), 88.0);
}
```

上述现象在 Rust 中偶有发生：虽然经历了一段与编译器的激烈拉锯战，但最后代码看起来相当不错，就仿佛这场拉锯战从未发生过一样，运行起来也很令人满意。

我们在这里所做的就是对 N 的限界进行逆向工程，使用编译器来指导和检查我们的工作。这个过程有点儿痛苦，因为标准库中没有那么一个 `Number` 特型包含我们想要使用的所有运算符和方法。碰巧的是，有一个名为 `num` 的流行开源 crate 定义了这样的一个特型。如果我们能提前知道，就可以将 `num` 添加到 `Cargo.toml` 中并这样写：

```
use num::Num;

fn dot<N: Num + Copy>(v1: &[N], v2: &[N]) -> N {
    let mut total = N::zero();
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

就像在面向对象编程中正确的接口能令一切变得美好一样，在泛型编程中正确的特型也能令一切变得美好。

不过，为什么要这么麻烦呢？为什么 Rust 的设计者不让泛型更像 C++ 模板中的“鸭子类型”那样在代码中隐含约束呢？

Rust 的这种方式的一个优点是泛型代码的前向兼容性。你可以更改公共泛型函数或方法的实现，只要没有更改签名，对它的用户就没有任何影响。

类型限界的另一个优点是，当遇到编译器错误时，至少编译器可以告诉你问题出在哪里。涉及模板的 C++ 编译器错误消息可能比 Rust 的错误消息要长得多，并且会指向许多不同的代码行，因为编译器无法判断谁应该为此问题负责：是模板，还是其调用者？而调用者也可能是模板，或者调用者模板的模板……

也许明确写出限界的最重要的优点是限界就这么写在代码和文档中。你可以查看 Rust 中泛型函数的签名，并准确了解它能接受的参数类型。而使用模板则做不到这些。在像 Boost 这样的 C++ 库中完整记录参数类型的工作比我们在这里经历的还要艰巨得多。Boost 开发人员可没有能帮他们检查工作成果的编译器。

11.6 以特型为基础

特型成为 Rust 中最主要的组织特性之一是有充分理由的，因为良好的接口在设计程序或库时尤为重要。

本章是关于语法、规则和解释的“狂风骤雨”。现在我们已经奠定了基础，可以开始讨论在 Rust 代码中使用特型和泛型的多种方式了。事实上，我们才刚刚开始入门。接下来的第 12 章和第 13 章将介绍标准库提供的公共特型。之后的各章将涵盖闭包、迭代器、输入 / 输出和并发。特型和泛型在所有这些主题中都扮演着核心角色。

第 12 章 运算符重载

在第 2 章展示的曼德博集绘图器中，我们使用了 `num crate` 的 `Complex` 类型来表示复平面上的数值：

```
#[derive(Clone, Copy, Debug)]
struct Complex<T> {
    /// 复数的实部
    re: T,

    /// 复数的虚部
    im: T,
}
```

使用 Rust 的 `+` 运算符和 `*` 运算符，可以像对任何内置数值类型一样对 `Complex` 进行加法运算和乘法运算：

```
z = z * z + c;
```

你也可以让自己的类型支持算术运算符和其他运算符，只要实现一些内置特型即可。这叫作**运算符重载**，其效果跟 C++、C#、Python 和 Ruby 中的运算符重载很相似。

运算符重载的特型可以根据其支持的语言特性分为几类，如表 12-1 所示。本章将逐一介绍每个类别。我们不仅要帮你把自己的类型很好地集成到语言中，而且要让你更好地了解如何编写泛型函数，比如 11.1.2 节讲过的 `dot_product` 函数，该函数能使用运算符自然而然地对自定义类型进行运算。本章还会深入讲解语言本身的某些特性是如何实现的。

表 12-1：运算符重载的特型汇总表

类别	特型	运算符

类别	特型	运算符
一元运算符	<code>std::ops::Neg</code> <code>std::ops::Not</code>	<code>-x</code> <code>!x</code>
算术运算符	<code>std::ops::Add</code> <code>std::ops::Sub</code> <code>std::ops::Mul</code> <code>std::ops::Div</code> <code>std::ops::Rem</code>	<code>x + y</code> <code>x - y</code> <code>x * y</code> <code>x / y</code> <code>x % y</code>
按位运算符	<code>std::ops::BitAnd</code> <code>std::ops::BitOr</code> <code>std::ops::BitXor</code> <code>std::ops::Shl</code> <code>std::ops::Shr</code>	<code>x & y</code> <code>x y</code> <code>x ^ y</code> <code>x << y</code> <code>x >> y</code>
复合赋值算术运算符	<code>std::ops::AddAssign</code> <code>std::ops::SubAssign</code> <code>std::ops::MulAssign</code> <code>std::ops::DivAssign</code> <code>std::ops::RemAssign</code>	<code>x += y</code> <code>x -= y</code> <code>x *= y</code> <code>x /= y</code> <code>x %= y</code>
复合赋值按位运算符	<code>std::ops::BitAndAssign</code> <code>std::ops::BitOrAssign</code> <code>std::ops::BitXorAssign</code> <code>std::ops::ShlAssign</code> <code>std::ops::ShrAssign</code>	<code>x &= y</code> <code>x = y</code> <code>x ^= y</code> <code>x <<= y</code> <code>x >>= y</code>
比较	<code>std::cmp::PartialEq</code> <code>std::cmp::PartialOrd</code>	<code>x == y</code> 、 <code>x != y</code> <code>x < y</code> 、 <code>x <= y</code> 、 <code>x > y</code> 、 <code>x >= y</code>

类别	特型	运算符
索引	<code>std::ops::Index</code> <code>std::ops::IndexMut</code>	<code>x[y]</code> 、 <code>&x[y]</code> <code>x[y] = z</code> 、 <code>&mut x[y]</code>

12.1 算术运算符与按位运算符

在 Rust 中，表达式 `a + b` 实际上是 `a.add(b)` 的简写形式，也就是对标准库中 `std::ops::Add` 特型的 `add` 方法的调用。Rust 的标准数值类型都实现了 `std::ops::Add`。为了使表达式 `a + b` 适用于 `Complex` 值，`num crate` 也为 `Complex` 实现了这个特型。还有一些类似的特型覆盖了其他运算符：`a * b` 是 `a.mul(b)` 的简写形式，也就是对 `std::ops::Mul` 特型的 `mul` 方法的调用，`std::ops::Neg` 实现了前缀取负运算符 `-`，等等。

如果试图写出 `z.add(c)`，就要将 `Add` 特型引入作用域，以便它的方法在此可见。做完这些，就可以将所有算术运算视为函数调用了：¹

¹Lisp 程序员应该很熟悉这种写法。表达式 `<i32 as Add>::add` 就是 `i32` 的 `+` 运算符，其结果是函数的返回值。

```
use std::ops::Add;

assert_eq!(4.125f32.add(5.75), 9.875);
assert_eq!(10.add(20), 10 + 20);
```

这是 `std::ops::Add` 的定义：

```
trait Add<Rhs = Self> {
    type Output;
    fn add(self, rhs: Rhs) -> Self::Output;
}
```

也就是说，`Add<T>` 特型代表给自己的类型加上一个 `T` 值的能力。如果希望把 `i32` 和 `u32` 型的值加到你的类型上，就必须同时实现 `Add<i32>` 和 `Add<u32>`。特型的类型参数 `Rhs` 默认为 `Self`，因此如果想在两个相同类型的值之间实现加法，那么可以仅为这种情况编写 `Add`。关联类型 `Output` 描述了加法结果的类型。

例如，为了能把两个 `Complex<i32>` 值加到一起，`Complex<i32>` 就必须实现 `Add<Complex<i32>>`。由于我们想为自身加上同类型的值，因此只需像下面这样编写 `Add` 即可：

```
use std::ops::Add;

impl Add for Complex<i32> {
    type Output = Complex<i32>;
    fn add(self, rhs: Self) -> Self {
        Complex {
            re: self.re + rhs.re,
            im: self.im + rhs.im,
        }
    }
}
```

当然，不必为 `Complex<i32>`、`Complex<f32>`、`Complex<f64>` 等逐个实现 `Add`。除了所涉及的类型不一样，所有定义看起来都完全相同，因此我们可以写一个涵盖所有这些定义的泛型实现，只要复数的各个组件本身的类型都支持加法就可以：

```
use std::ops::Add;

impl<T> Add for Complex<T>
where
    T: Add<Output = T>,
{
    type Output = Self;
    fn add(self, rhs: Self) -> Self {
        Complex {
            re: self.re + rhs.re,
            im: self.im + rhs.im,
        }
    }
}
```

通过编写 `where T: Add<Output=T>`，我们将 `T` 限定到能与自身相加并产生另一个 `T` 值的类型。虽然这是一个合理的限制，但还可以将条件进一步放宽，因为 `Add` 特型不要求 `+` 的两个操作数具有相同的类型，也不限制结果类型。因此，一个尽可能泛化的实现可以让

左右操作数独立变化，并生成加法所能生成的任何组件类型的 Complex 值：

```
use std::ops::Add;

impl<L, R> Add<Complex<R>> for Complex<L>
where
    L: Add<R>,
{
    type Output = Complex<L::Output>;
    fn add(self, rhs: Complex<R>) -> Self::Output {
        Complex {
            re: self.re + rhs.re,
            im: self.im + rhs.im,
        }
    }
}
```

然而，在实践中，Rust 更倾向于避免混合类型运算。由于我们的类型参数 L 必须实现 Add<R>，因此通常情况下 L 和 R 将是相同的类型。这是因为对 L 而言，该限制太严格了，不大可能实现其他类型的 Add。最终，这个尽可能泛化版本的泛型定义，其实并不比之前的简单版本有用多少。

Rust 的算术运算符和按位运算符的内置特型分为 3 组：一元运算符、二元运算符和复合赋值运算符。在每一组中，特型及其方法都具有相同的形式，因此接下来我们将各举一例。

12.1.1 一元运算符

除了我们将在 13.5 节单独介绍的解引用运算符 *，Rust 还有两个可以自定义的一元运算符，如表 12-2 所示。

表 12-2：一元运算符的内置特型

特型名称	表达式	等效表达式

特型名称	表达式	等效表达式
<code>std::ops::Neg</code>	<code>-x</code>	<code>x.neg()</code>
<code>std::ops::Not</code>	<code>!x</code>	<code>x.not()</code>

Rust 的所有带符号数值类型都实现了 `std::ops::Neg`，以支持一元取负运算符 `-`；整数类型和 `bool` 实现了 `std::ops::Not`，以支持一元取反运算符 `!`。还有一些是针对这些类型的引用的实现。

请注意，`!` 运算符会对 `bool` 值进行取反，而对整数执行按位取反，它同时扮演着 C 和 C++ 中的 `!` 运算符和 `~` 运算符的角色。

这些特型的定义很简单：

```
trait Neg {
    type Output;
    fn neg(self) -> Self::Output;
}

trait Not {
    type Output;
    fn not(self) -> Self::Output;
}
```

对一个复数取负就是对它的每个组件取负。以下是对 `Complex` 值进行取负的泛型实现。

```
use std::ops::Neg;

impl<T> Neg for Complex<T>
where
    T: Neg<Output = T>,
{
    type Output = Complex<T>;
    fn neg(self) -> Complex<T> {
        Complex {
```

```

        re: -self.re,
        im: -self.im,
    }
}
}

```

12.1.2 二元运算符

Rust 的二元算术运算符和二元按位运算符及它们对应的内置特型参见表 12-3。

表 12-3：二元运算符的内置特型

类别	特型名称	表达式	等效表达式
算术运算符	<code>std::ops::Add</code> <code>std::ops::Sub</code> <code>std::ops::Mul</code> <code>std::ops::Div</code> <code>std::ops::Rem</code>	<code>x + y</code> <code>x - y</code> <code>x * y</code> <code>x / y</code> <code>x % y</code>	<code>x.add(y)</code> <code>x.sub(y)</code> <code>x.mul(y)</code> <code>x.div(y)</code> <code>x.rem(y)</code>
按位运算符	<code>std::ops::BitAnd</code> <code>std::ops::BitOr</code> <code>std::ops::BitXor</code> <code>std::ops::Shl</code> <code>std::ops::Shr</code>	<code>x & y</code> <code>x y</code> <code>x ^ y</code> <code>x << y</code> <code>x >> y</code>	<code>x.bitand(y)</code> <code>x.bitor(y)</code> <code>x.bitxor(y)</code> <code>x.shl(y)</code> <code>x.shr(y)</code>

Rust 的所有数值类型都实现了算术运算符。Rust 的整数类型和 `bool` 类型都实现了按位运算符。此外，还有一些运算符能接受“对这些类型的引用”作为一个或两个操作数。

这里的所有特型，其一般化形式都是一样的。例如，对于 `^` 运算符，`std::ops::BitXor` 的定义如下所示：

```

trait BitXor<Rhs = Self> {
    type Output;
}

```

```
fn bitxor(self, rhs: Rhs) -> Self::Output;
}
```

本章开头还展示过此类别中的另一个特型 `std::ops::Add`，以及几个范例实现。

你可以使用 `+` 运算符将 `String` 与 `&str` 切片或另一个 `String` 连接起来。但是，`Rust` 不允许 `+` 的左操作数是 `&str` 类型，以防止通过在左侧重复接入小型片段来构建长字符串。（这种方式性能不佳，其时间复杂度是字符串最终长度的平方。）一般来说，`write!` 宏更适合从小型片段构建出字符串，17.3.3 节会展示如何执行此操作。

12.1.3 复合赋值运算符

复合赋值表达式形如 `x += y` 或 `x &= y`：它接受两个操作数，先对它们执行加法或按位与等操作，然后再将结果写回左操作数。在 `Rust` 中，复合赋值表达式自身的值总是 `()`，而不是所存入的值。

许多语言有这样的运算符，并且通常会将它们定义为 `x = x + y` 或 `x = x & y` 等表达式的简写形式。但是，`Rust` 没有采用这种方式。在 `Rust` 中，`x += y` 是方法调用 `x.add_assign(y)` 的简写形式，其中 `add_assign` 是 `std::ops::AddAssign` 特型的唯一方法：

```
trait AddAssign<Rhs = Self> {
    fn add_assign(&mut self, rhs: Rhs);
}
```

表 12-4 展示了 `Rust` 的所有复合赋值运算符和实现了它们的内置特型。

表 12-4：复合赋值运算符的内置特型

类别	特型名称	表达式	等效表达式

类别	特型名称	表达式	等效表达式
算术运算符	<code>std::ops::AddAssign</code>	<code>x += y</code>	<code>x.add_assign(y)</code>
	<code>std::ops::SubAssign</code>	<code>x -= y</code>	<code>x.sub_assign(y)</code>
	<code>std::ops::MulAssign</code>	<code>x *= y</code>	<code>x.mul_assign(y)</code>
	<code>std::ops::DivAssign</code>	<code>x /= y</code>	<code>x.div_assign(y)</code>
	<code>std::ops::RemAssign</code>	<code>x %= y</code>	<code>x.rem_assign(y)</code>
按位运算符	<code>std::ops::BitAndAssign</code>	<code>x &= y</code>	<code>x.bitand_assign(y)</code>
	<code>std::ops::BitOrAssign</code>	<code>x = y</code>	<code>x.bitor_assign(y)</code>
	<code>std::ops::BitXorAssign</code>	<code>x ^= y</code>	<code>x.bitxor_assign(y)</code>
	<code>std::ops::ShlAssign</code>	<code>x <<= y</code>	<code>x.shl_assign(y)</code>
	<code>std::ops::ShrAssign</code>	<code>x >>= y</code>	<code>x.shr_assign(y)</code>

Rust 的所有数值类型都实现了算术复合赋值运算符。Rust 的整数类型和 `bool` 类型都实现了按位复合赋值运算符。

为 `Complex` 类型实现 `AddAssign` 的泛型代码一目了然：

```
use std::ops::AddAssign;

impl<T> AddAssign for Complex<T>
where
    T: AddAssign<T>,
{
    fn add_assign(&mut self, rhs: Complex<T>) {
        self.re += rhs.re;
        self.im += rhs.im;
    }
}
```

复合赋值运算符的内置特型完全独立于相应二元运算符的内置特型。实现 `std::ops::Add` 并不会自动实现 `std::ops::AddAssign`，如果能让 Rust 允许你的类型作为 `+=` 运算符的左操作数，就必须自行实现 `AddAssign`。

12.2 相等性比较

Rust 的相等性运算符 `==` 和 `!=` 是对调用 `std::cmp::PartialEq` 特型的 `eq` 和 `ne` 这两个方法的简写:

```
assert_eq!(x == y, x.eq(&y));
assert_eq!(x != y, x.ne(&y));
```

下面是 `std::cmp::PartialEq` 的定义:

```
trait PartialEq<Rhs = Self>
where
    Rhs: ?Sized,
{
    fn eq(&self, other: &Rhs) -> bool;
    fn ne(&self, other: &Rhs) -> bool {
        !self.eq(other)
    }
}
```

由于 `ne` 方法有一个默认定义, 因此你只需定义 `eq` 来实现 `PartialEq` 特型即可。下面是 `Complex` 的完整实现:

```
impl<T: PartialEq> PartialEq for Complex<T> {
    fn eq(&self, other: &Complex<T>) -> bool {
        self.re == other.re && self.im == other.im
    }
}
```

换句话说, 对于自身可以做相等性比较的任意组件类型 `T`, 这个实现就能为 `Complex<T>` 提供比较功能。假设我们还在某处为 `Complex` 实现了 `std::ops::Mul`, 那么现在就可以这样写了:

```
let x = Complex { re: 5, im: 2 };
let y = Complex { re: 2, im: 5 };
assert_eq!(x * y, Complex { re: 0, im: 29 });
```

PartialEq 的实现几乎就是这里展示的形式，即将左操作数的每个字段与右操作数的相应字段进行比较。手写这些代码很枯燥，而相等性是一个常见的支持性操作，所以只要提出要求，Rust 就会自动为你生成一个 PartialEq 的实现。只需把 PartialEq 添加到类型定义的 derive 属性中即可，如下所示：

```
[derive(Clone, Copy, Debug, PartialEq)]
struct Complex<T> {
    ...
}
```

Rust 自动生成的实现与手写的代码本质上是一样的，都会依次比较每个字段或类型的元素。Rust 还可以为 enum 类型派生出 PartialEq 实现。同样，该类型含有（对于 enum 则是所有可能含有）的每个值本身必须实现 PartialEq。

与按值获取操作数的算术特型和按位运算特型不同，PartialEq 会通过引用获取其操作数。这意味着在比较诸如 String、Vec 或 HashMap 之类的非 Copy 值时并不会导致它们被移动，否则就会很麻烦：

```
let s = "d\x6fv\x65t\x61i\x6c".to_string();
let t = "\x64o\x76e\x74a\x69l".to_string();
assert!(s == t); // s和t都是借用来的.....

// .....所以，在这里它们仍然拥有自己的值
assert_eq!(format!("{}", s), format!("{}", t), "dovetail dovetail");
```

注意 Rhs 类型参数上的特型限界，这是一种我们从未见过的类型：

```
where
    Rhs: ?Sized,
```

这放宽了 Rust 对类型参数必须有固定大小的常规要求，能让我们写出像 PartialEq<str> 或 PartialEq<[T]> 这样的特型。eq 方法和 ne 方法会接受 &Rhs 类型的参数，因为将某些类型的值与

`&str` 或 `&[T]` 进行比较是完全合理的。由于 `str` 实现了 `PartialEq<str>`，因此以下断言是等效的：

```
assert!("ungula" != "ungulate");  
assert!("ungula".ne("ungulate"));
```

在这里，`Self` 和 `Rhs` 都是无固定大小类型 `str`，这就令 `ne` 的 `self` 参数和 `rhs` 参数都是 `&str` 值。13.2 节会详细讨论固定大小类型、无固定大小类型和 `Sized` 特型。

为什么这个特型叫作 `PartialEq`？这是因为等价关系（相等就是其中之一）的传统数学定义提出了 3 个要求。对于任意值 `x` 和 `y`，需满足以下条件。

- 如果 `x == y` 为真，则 `y == x` 也必然为真。换句话说，交换相等性比较的两个操作数不会影响比较结果。
- 如果 `x == y` 且 `y == z`，则 `x == z` 一定成立。给定任何值组成的链，其中的每个值必然等于下一个值，链中的每个值都直接等于其他值。相等性是可传递的。
- `x == x` 必须始终为真。

最后一个要求可能看起来过于显而易见而不值一提，但这正是容易出错的地方。Rust 的 `f32` 和 `f64` 是 IEEE 标准浮点值。根据该标准，像 `0.0/0.0` 和其他没有适当值的表达式必须生成特殊的非数值，通常叫作 NaN 值。该标准进一步要求将 NaN 值视为与包括其自身在内的所有其他值都不相等。例如，标准要求以下所有断言都成立：

```
assert!(f64::is_nan(0.0 / 0.0));  
assert_eq!(0.0 / 0.0 == 0.0 / 0.0, false);  
assert_eq!(0.0 / 0.0 != 0.0 / 0.0, true);
```

此外，任何值与 NaN 值进行有序比较都必须返回 `false`：

```
assert_eq!(0.0 / 0.0 < 0.0 / 0.0, false);  
assert_eq!(0.0 / 0.0 > 0.0 / 0.0, false);
```

```
assert_eq!(0.0 / 0.0 <= 0.0 / 0.0, false);
assert_eq!(0.0 / 0.0 >= 0.0 / 0.0, false);
```

因此，虽然 Rust 的 `==` 运算符满足等价关系的前两个要求，但当用于 IEEE 浮点值时，它显然不满足第三个要求。这称为部分相等关系，因此 Rust 使用名称 `PartialEq` 作为 `==` 运算符的内置特型。如果要用仅支持 `PartialEq` 类型的参数编写泛型代码，那么可以假设前两个要求一定成立，但不应假设任何值一定等于它自身。

这有点儿反直觉，如果不提高警惕，就可能带来 bug。如果你的泛型代码想要“完全相等”关系，那么可以改用 `std::cmp::Eq` 特型作为限界，它表示完全相等关系：如果类型实现了 `Eq`，则对于该类型的每个值 `x`，`x == x` 都必须为 `true`。实际上，几乎所有实现了 `PartialEq` 的类型都实现了 `Eq`，而 `f32` 和 `f64` 是标准库中仅有的两个属于 `PartialEq` 却不属于 `Eq` 的类型。

标准库将 `Eq` 定义为 `PartialEq` 的扩展，而且未添加新方法：

```
trait Eq: PartialEq<Self> {}
```

如果你的类型是 `PartialEq` 并且希望它也是 `Eq`，就必须显式实现 `Eq`，不过你并不需要实际为此定义任何新函数或类型。所以要为 `Complex` 类型实现 `Eq` 很简单：

```
impl<T: Eq> Eq for Complex<T> {}
```

甚至可以通过在 `Complex` 类型定义的 `derive` 属性中包含 `Eq` 来更简洁地实现它：

```
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
struct Complex<T> {
    ...
}
```

泛型类型的派生实现可能取决于类型参数。使用 `derive` 属性，`Complex<i32>` 将实现 `Eq`，因为 `i32` 实现了它，但

`Complex<f32>` 只能实现 `PartialEq`，因为 `f32` 没有实现 `Eq`。

当你自己实现 `std::cmp::PartialEq` 时，Rust 无法检查你对 `eq` 方法和 `ne` 方法的定义是否真的符合部分相等或完全相等的要求。你的实现可以“为所欲为”。Rust 只会接受你给出的结果，因为它假设你已经以满足特型用户期望的方式实现了相等性。

尽管 `PartialEq` 已经为 `ne` 提供了默认定义，但你也可以根据需要提供自己的实现。不过，你必须确保 `ne` 和 `eq` 彼此精确互补，因为 `PartialEq` 特型的用户会认为理所当然。

12.3 有序比较

Rust 会根据单个特型 `std::cmp::PartialOrd` 来定义全部的有序比较运算符 `<`、`>`、`<=` 和 `>=` 的行为：

```
trait PartialOrd<Rhs = Self>: PartialEq<Rhs>
where
    Rhs: ?Sized,
{
    fn partial_cmp(&self, other: &Rhs) -> Option<Ordering>;
    fn lt(&self, other: &Rhs) -> bool { ... }
    fn le(&self, other: &Rhs) -> bool { ... }
    fn gt(&self, other: &Rhs) -> bool { ... }
    fn ge(&self, other: &Rhs) -> bool { ... }
}
```

请注意，`PartialOrd<Rhs>` 扩展了 `PartialEq<Rhs>`：只有可以比较相等性的类型才能比较顺序性。

`PartialOrd` 中必须自行实现的唯一方法是 `partial_cmp`。当 `partial_cmp` 返回 `Some(o)` 时，`o` 应该指出 `self` 与 `other` 之间的关系：

```
enum Ordering {
    Less,          // self < other
    Equal,         // self == other
    Greater,       // self > other
}
```

但是如果 `partial_cmp` 返回 `None`，那么就意味着 `self` 和 `other` 相对于彼此是无序的，即两者都不大于对方，但也不相等。在 Rust 的所有原始类型中，只有浮点值之间的比较会返回 `None`：具体来说，将 NaN 值与任何其他值进行比较都会返回 `None`。有关 NaN 值的更多背景知识，请参见 12.2 节。

和其他二元运算符一样，如果要比较 `Left` 和 `Right` 这两种类型的值，那么 `Left` 就必须实现 `PartialOrd<Right>`。像 `x < y` 或

`x >= y` 这样的表达式都是调用 `PartialOrd` 方法的简写形式，如表 12-5 所示。

表 12-5：有序比较运算符和 **PartialOrd** 方法

表达式	相等性方法调用	默认定义
<code>x < y</code>	<code>x.lt(y)</code>	<code>x.partial_cmp(&y) == Some(Less)</code>
<code>x > y</code>	<code>x.gt(y)</code>	<code>x.partial_cmp(&y) == Some(Greater)</code>
<code>x <= y</code>	<code>x.le(y)</code>	<code>matches!(x.partial_cmp(&y), Some(Less Equal))</code>
<code>x >= y</code>	<code>x.ge(y)</code>	<code>matches!(x.partial_cmp(&y), Some(Greater Equal))</code>

与前面的示例一样，这里的相等性方法调用代码也假定当前作用域中已经引入了 `std::cmp::PartialOrd` 和 `std::cmp::Ordering`。

如果你知道两种类型的值总能确定相对于彼此的顺序，那么就可以实现更严格的 `std::cmp::Ord` 特型：

```
trait Ord: Eq + PartialOrd<Self> {
    fn cmp(&self, other: &Self) -> Ordering;
}
```

这里的 `cmp` 方法只会返回 `Ordering`，而不会像 `partial_cmp` 那样返回 `Option<Ordering>`： `cmp` 总会声明它的两个参数相等或

指出它们的相对顺序。几乎所有实现了 `PartialOrd` 的类型都应该实现 `Ord`。在标准库中，`f32` 和 `f64` 是该规则的例外情况。

由于复数没有自然顺序，因此我们无法使用前几节中的 `Complex` 类型来展示 `PartialOrd` 的示例实现。相反，假设你正在使用以下类型表示落在给定左闭右开区间内的一组数值：

```
[derive(Debug, PartialEq)]
struct Interval<T> {
    lower: T, // 闭区间
    upper: T, // 开区间
}
```

你希望对这种类型的值进行部分排序，即如果一个区间完全落在另一个区间之前，并且没有重叠，则认为这个区间小于另一个区间。如果两个不相等的区间有重叠（每一侧都有某些元素小于另一侧的某些元素），则认为它们是无序的。而两个相等的区间必然是完全相等的。以下 `PartialOrd` 代码实现了这些规则：

```
use std::cmp::{Ordering, PartialOrd};

impl<T: PartialOrd> PartialOrd<Interval<T>> for Interval<T> {
    fn partial_cmp(&self, other: &Interval<T>) -> Option<Ordering> {
        {
            if self == other {
                Some(Ordering::Equal)
            } else if self.lower >= other.upper {
                Some(Ordering::Greater)
            } else if self.upper <= other.lower {
                Some(Ordering::Less)
            } else {
                None
            }
        }
    }
}
```

有了这个实现，你就可以写出如下代码了：

```
assert!(Interval { lower: 10, upper: 20 } < Interval { lower: 20,
upper: 40 });
assert!(Interval { lower: 7, upper: 8 } >= Interval { lower: 0,
```

```
upper: 1  });  
assert!(Interval { lower: 7,  upper: 8  } <= Interval { lower: 7,  
upper: 8  });  
  
// 两个存在重叠的区间相对彼此没有顺序可言  
let left  = Interval { lower: 10, upper: 30 };  
let right = Interval { lower: 20, upper: 40 };  
assert!(!(left < right));  
assert!(!(left >= right));
```

虽然通常我们会使用 `PartialOrd`，但在某些情况下，用 `Ord` 定义的完全排序也是有必要的，比如在标准库中实现的那些排序方法。但不可能仅通过 `PartialOrd` 来对区间进行排序。如果你确实想对它们进行排序，则必须想办法填补这些无法确定顺序的情况。如果你希望按上限排序，那么很容易用 `sort_by_key` 来实现：

```
intervals.sort_by_key(|i| i.upper);
```

包装器类型 `Reverse` 就利用了这一点，借助一个简单的逆转任何顺序的方法来实现 `Ord`。对于任何实现了 `Ord` 的类型 `T`，`std::cmp::Reverse<T>` 也会实现 `Ord`，只是顺序相反。例如，可以简单地按下限从高到低对这些区间进行排序。

```
use std::cmp::Reverse;  
intervals.sort_by_key(|i| Reverse(i.lower));
```

12.4 Index 与 IndexMut

通过实现 `std::ops::Index` 特型和 `std::ops::IndexMut` 特型，你可以规定像 `a[i]` 这样的索引表达式该如何作用于你的类型。数组可以直接支持 `[]` 运算符，但对其他类型来说，表达式 `a[i]` 通常是 `*a.index(i)` 的简写形式，其中 `index` 是 `std::ops::Index` 特型的方法。但是，如果表达式被赋值或借用成了可变形式，那么 `a[i]` 就是对调用 `std::ops::IndexMut` 特型方法的 `*a.index_mut(i)` 的简写。

以下是 `Index` 和 `IndexMut` 这两个特型的定义：

```
trait Index<Idx> {
    type Output: ?Sized;
    fn index(&self, index: Idx) -> &Self::Output;
}

trait IndexMut<Idx>: Index<Idx> {
    fn index_mut(&mut self, index: Idx) -> &mut Self::Output;
}
```

请注意，这些特型会以索引表达式的类型作为参数。你可以使用单个 `usize` 对切片进行索引，以引用单个元素，因为切片实现了 `Index<usize>`。还可以使用像 `a[i..j]` 这样的表达式来引用子切片，因为切片也实现了 `Index<Range<usize>>`。该表达式是以下内容的简写形式：

```
*a.index(std::ops::Range { start: i, end: j })
```

Rust 的 `HashMap` 集合和 `BTreeMap` 集合允许使用任何可哈希类型或有序类型作为索引。以下代码之所以能运行，是因为 `HashMap<&str, i32>` 实现了 `Index<&str>`：

```
use std::collections::HashMap;
let mut m = HashMap::new();
m.insert("十", 10);
```

```
m.insert("百", 100);
m.insert("千", 1000);
m.insert("万", 1_0000);
m.insert("億", 1_0000_0000);

assert_eq!(m["十"], 10);
assert_eq!(m["千"], 1000);
```

这些索引表达式等效于如下内容：

```
use std::ops::Index;
assert_eq!(*m.index("十"), 10);
assert_eq!(*m.index("千"), 1000);
```

`Index` 特型的关联类型 `Output` 指定了索引表达式要生成的类型：对这个 `HashMap` 而言，`Index` 实现的 `Output` 类型是 `i32`。

`IndexMut` 特型使用 `index_mut` 方法（该方法接受对 `self` 的可变引用）扩展了 `Index`，并返回了对 `Output` 值的可变引用。当索引表达式出现在需要可变引用的上下文中时，`Rust` 会自动选择 `index_mut`。假设我们编写了如下代码：

```
let mut desserts =
    vec!["Howalon".to_string(), "Soan papdi".to_string()];
desserts[0].push_str(" (fictional)");
desserts[1].push_str(" (real)");
```

因为 `push_str` 方法要对 `&mut self` 进行操作，所以最后两行代码等效于如下内容：

```
use std::ops::IndexMut;
(*desserts.index_mut(0)).push_str(" (fictional)");
(*desserts.index_mut(1)).push_str(" (real)");
```

`IndexMut` 有一个限制，即根据设计，它必须返回对某个值的可变引用。这就是不能使用像 `m["十"] = 10`；这样的表达式来将值插入 `m` 这个 `HashMap` 中的原因：该表需要先为 "十" 创建一个带有默认值的条目，然后再返回一个对它的可变引用。但并不是所有的类型都有开销很低的默认值，有些可能开销很高，创建这么一个马上

就会因赋值而被丢弃的值是一种浪费。（Rust 计划在更高版本中对此进行改进。）

索引最常用于各种集合。假设我们要处理第 2 章中曼德博集绘图器那样的位图图像。当时我们的程序中包含如下代码：

```
pixels[row * bounds.0 + column] = ...;
```

如果有一个像二维数组一样的 `Image<u8>` 类型肯定会更好，这样就可以访问像素而不必写出所有的算法了：

```
image[row][column] = ...;
```

为此，需要声明一个结构体：

```
struct Image<P> {
    width: usize,
    pixels: Vec<P>,
}

impl<P: Default + Copy> Image<P> {
    /// 创建一个给定大小的新图像
    fn new(width: usize, height: usize) -> Image<P> {
        Image {
            width,
            pixels: vec![P::default(); width * height],
        }
    }
}
```

以下是符合要求的 `Index` 和 `IndexMut` 的实现：

```
impl<P> std::ops::Index<usize> for Image<P> {
    type Output = [P];
    fn index(&self, row: usize) -> &[P] {
        let start = row * self.width;
        &self.pixels[start..start + self.width]
    }
}

impl<P> std::ops::IndexMut<usize> for Image<P> {
    fn index_mut(&mut self, row: usize) -> &mut [P] {
```

```
        let start = row * self.width;  
        &mut self.pixels[start..start + self.width]  
    }  
}
```

对 `Image` 进行索引时，你会得到一些像素的切片，再索引此切片会返回一个单独的像素。

请注意，在编写 `image[row][column]` 时，如果 `row` 超出范围，那么 `.index()` 方法在试图索引 `self.pixels` 时也会超出范围，从而引发 `panic`。这就是 `Index` 实现和 `IndexMut` 实现的行为方式：检测到越界访问并导致 `panic`，就像索引数组、切片或向量时越界一样。

12.5 其他运算符

并非所有运算符都可以在 Rust 中重载。从 Rust 1.50 开始，错误检查运算符 `?` 仅适用于 `Result` 值和 `Option` 值，不过 Rust 也在努力将其扩展到用户定义类型。同样，逻辑运算符 `&&` 和 `||` 仅限于 `bool` 值。`..` 运算符和 `..=` 运算符总会创建一个表示范围边界的结构体，`&` 运算符总是会借用引用，`=` 运算符总是会移动值或复制值。它们都不能重载。

解引用运算符 `*val` 和用于访问字段和调用方法的点运算符（如 `val.field` 和 `val.method()`）可以用 `Deref` 特型和 `DerefMut` 特型进行重载，这将在第 13 章中介绍。（之所以本章没有包含它们，是因为这两个特型不仅仅是重载几个运算符那么简单。）

Rust 不支持重载函数调用运算符 `f(x)`。当你需要一个可调用的值时，通常只需编写一个闭包即可。第 14 章将解释它是如何工作的，同时会涵盖 `Fn`、`FnMut` 和 `FnOnce` 这几个特殊特型。

第 13 章 实用工具特型

科学无非就是在自然界的多样性中寻求统一性（或者更确切地说，是在我们经验的多样性中寻求统一性）。用 Coleridge 的话说，诗歌、绘画、艺术，同样是在多样性中寻求统一性。

——Jacob Bronowski

本章讲解了所谓的 Rust “实用工具” 特型，这是标准库中各种特型的“百宝箱”，它们对 Rust 的编写方式有相当大的影响，所以，只有熟悉它们，你才能写出符合 Rust 语言惯例的代码并据此为你的 crate 设计公共接口，让用户认为这些接口是符合 Rust 风格的。Rust 实用工具特型可分为三大类。

语言扩展特型

第 12 章中介绍的运算符重载特型能让你在自己的类型上使用 Rust 的表达式运算符，同样，还有其他几个标准库特型也是 Rust 的扩展点，允许你把自己的类型更紧密地集成进语言中。这类特型包括 Drop、Deref 和 DerefMut，以及转换特型 From 和 Into。我们将在本章中讲解它们。

标记特型

这类特型多用作泛型类型变量的限界，以表达无法以其他方式捕获的约束条件。Sized 和 Copy 就属于这类特型。

公共词汇特型

这类特型不涉及任何编译器魔术，你完全可以在自己的代码中定义其等效特型。之所以定义它们，是为了给常见问题制定一些约定俗成的解决方案。这对 crate 和模块之间的公共接口来说特别有价值：通过减少不必要的变体，让接口更容易理解，也增加了把来自不同

crate 的特性轻易插接在一起的可能性，而且无须样板代码或自定义胶水代码。

这类特型包括 Default、引用借用特型 AsRef、AsMut、Borrow 与 BorrowMut、容错的转换特型 TryFrom 与 TryInto，以及 ToOwned 特型（对 Clone 的泛化）。

表 13-1 对上述特型进行了汇总。

表 13-1：实用工具特型汇总表

特型	描述
Drop	析构器。每当丢弃一个值时，Rust 都要自动运行的清理代码
Sized	具有在编译期已知的固定大小类型的标记特型，与之相对的是动态大小类型（如切片）
Clone	用来支持克隆值的类型
Copy	可以简单地通过对包含值的内存进行逐字节复制以进行克隆的类型的标记特型
Deref 与 DerefMut	智能指针类型的特型
Default	具有合理“默认值”的类型
AsRef 与 AsMut	用于从另一种类型中借入一种引用类型的转换特型

特型	描述
Borrow 与 BorrowMut	转换特型，类似 AsRef/AsMut，但能额外保证一致的哈希、排序和相等性
From 与 Into	用于将一种类型的值转换为另一种类型的转换特型
TryFrom 与 TryInto	用于将一种类型的值转换为另一种类型的转换特型，用于可能失败的转换
ToOwned	用于将引用转换为拥有型值的转换特型

还有另一些重要的标准库特型。第 15 章会介绍 Iterator 和 IntoIterator。第 16 章会介绍用于计算哈希值的 Hash 特型。第 19 章会介绍两个用于标记线程安全类型的特型 Send 和 Sync。

13.1 Drop

当一个值的拥有者消失时，Rust 会丢弃（drop）该值。丢弃一个值就必须释放该值拥有的任何其他值、堆存储和系统资源。丢弃可能发生在多种情况下：当变量超出作用域时；在表达式语句的末尾；当截断一个向量时，会从其末尾移除元素；等等。

在大多数情况下，Rust 会自动处理丢弃值的工作。假设你定义了以下类型：

```
struct Appellation {  
    name: String,  
    nicknames: Vec<String>  
}
```

Appellation 拥有用作字符串内容和向量元素缓冲区的堆存储。每当 Appellation 被丢弃时，Rust 都会负责清理所有这些内容，无须你进行任何进一步的编码。但只要你想，也可以通过实现 `std::ops::Drop` 特型来自定义 Rust 该如何丢弃此类型的值：

```
trait Drop {  
    fn drop(&mut self);  
}
```

Drop 的实现类似于 C++ 中的析构函数或其他语言中的终结器。当一个值被丢弃时，如果它实现了 `std::ops::Drop`，那么 Rust 就会调用它的 `drop` 方法，然后像往常一样继续丢弃它的字段或元素拥有的任何值。这种对 `drop` 的隐式调用是调用该方法的唯一途径。如果你试图显式调用该方法，那么 Rust 会将其标记为错误。

Rust 在丢弃某个值的字段或元素之前会先对值本身调用 `Drop::drop`，该方法收到的值仍然是已完全初始化的。因此，在 Appellation 类型的 Drop 实现中可以随意使用其字段：

```
impl Drop for Appellation {  
    fn drop(&mut self) {
```

```
        print!("Dropping {}", self.name);
        if !self.nicknames.is_empty() {
            print!(" (AKA {})", self.nicknames.join(", "));
        }
        println!("");
    }
}
```

基于该实现，可以编写以下内容：

```
{
    let mut a = Appellation {
        name: "Zeus".to_string(),
        nicknames: vec!["cloud collector".to_string(),
                        "king of the gods".to_string()]
    };

    println!("before assignment");
    a = Appellation { name: "Hera".to_string(), nicknames: vec![] };
    println!("at end of block");
}
```

当我们将第二个 `Appellation` 赋值给 `a` 时，就会丢弃第一个 `Appellation`，而当我们离开 `a` 的作用域时，就会丢弃第二个 `Appellation`。上述代码会打印出以下内容：

```
before assignment
Dropping Zeus (AKA cloud collector, king of the gods)
at end of block
Dropping Hera
```

`Appellation` 的 `std::ops::Drop` 实现只打印了一条消息，那么它的内存究竟是如何清理的呢？`Vec` 类型实现了 `Drop`，它会丢弃自己的每一个元素，然后释放它们占用的分配在堆上的缓冲区。`String` 在内部使用 `Vec<u8>` 来保存它的文本，因此 `String` 不需要自己实现 `Drop`，它会让 `Vec` 负责释放这些字符。同样的原则也适用于 `Appellation` 值：当一个值被丢弃时，最终由 `Vec` 的 `Drop` 实现来负责真正释放每个字符串的内容，并释放保存这些向量

元素的缓冲区。至于 Appellation 值本身占用的内存，它的拥有者（可能是某个局部变量或某些数据结构）会负责释放。

如果一个变量的值移动到了别处，以致该变量在超出作用域时正处于未初始化状态，那么 Rust 将不会试图丢弃该变量，因为这里没有需要丢弃的值。

虽然根据控制流才能判断变量的值是否会移动出去，但这一原则仍然成立。在这种情况下，Rust 会使用一个不可见的标志来跟踪此变量的状态，该标志会指出是否需要丢弃此变量的值：

```
let p;
{
    let q = Appellation { name: "Cardamine hirsuta".to_string(),
                          nicknames: vec!["shotweed".to_string(),
"bittercress".to_string()] };
    if complicated_condition() {
        p = q;
    }
}
println!("Sproing! What was that?");
```

根据 complicated_condition 返回的是 true 还是 false，p 或 q 中的一个会最终拥有 Appellation，而另一个则会变成未初始化状态。这种差异决定了它是在 println! 之前还是之后丢弃（因为 q 在 println! 之前就离开了作用域，而 p 则在 println! 之后离开的作用域）。虽然一个值可能会从一个地方移动到另一个地方，但 Rust 只会丢弃它一次。

除非正在定义某个拥有 Rust 不了解的资源类型，通常我们不需要自己实现 std::ops::Drop。例如，在 Unix 系统上，Rust 的标准库在内部使用了以下类型来表示操作系统的文件描述符：

```
struct FileDesc {
    fd: c_int,
}
```

FileDesc 的 fd 字段是当程序完成时应该关闭的文件描述符的编号，c_int 是 i32 的别名。标准库为 FileDesc 实现的 Drop 如下所示：

```
impl Drop for FileDesc {
    fn drop(&mut self) {
        let _ = unsafe { libc::close(self.fd) };
    }
}
```

这里，libc::close 是 C 库中 close 函数的 Rust 名称。Rust 代码只能在 unsafe 块中调用 C 函数，因此在这里使用了一个 unsafe 块。

如果一个类型实现了 Drop，就不能再实现 Copy 特型了。如果类型是 Copy 类型，就表示简单的逐字节复制足以生成该值的独立副本。但是，对同一份数据多次调用同一个 drop 方法显然是错误的。

标准库预导入中包含一个丢弃值的函数 drop，但它的定义一点儿也不神奇：

```
fn drop<T>(_x: T) { }
```

换句话说，它会按值接受参数，从调用者那里获得所有权，然后什么也不做。当 _x 超出作用域时，Rust 自然会丢弃它的值，这跟对任何其他变量的操作一样。

13.2 Sized

固定大小类型是指其每个值在内存中都有相同大小的类型。Rust 中的几乎所有类型都是固定大小的，比如每个 `u64` 占用 8 字节，每个 `(f32, f32, f32)` 元组占用 12 字节。甚至枚举也是有大小的，也就是说，无论实际存在的是哪个变体，枚举总会占据足够的空间来容纳其最大的变体。尽管 `Vec<T>` 拥有一个大小可变的堆分配缓冲区，但 `Vec` 值本身是指向“缓冲区、容量和长度”的指针，因此 `Vec<T>` 也是一个固定大小类型。

所有固定大小类型都实现了 `std::marker::Sized` 特型，该特型没有方法或关联类型。Rust 自动为所有适用的类型实现了 `std::marker::Sized` 特型，你不能自己实现它。`Sized` 的唯一用途是作为类型变量的限界：像 `T: Sized` 这样的限界要求 `T` 必须是在编译期已知的类型。由于 Rust 语言本身会使用这种类型的特型为具有某些特征的类型打上标记，因此我们将其称为**标记特型**。

然而，Rust 也有一些**无固定大小类型**，它们的值大小不尽相同。例如，字符串切片类型 `str`（注意没有 `&`）就是无固定大小的。字符串字面量 `"diminutive"` 和 `"big"` 是对占用了 10 字节和 3 字节的 `str` 切片的引用，两者都展示在图 13-1 中。像 `[T]`（同样没有 `&`）这样的数组切片类型也是无固定大小的，即像 `&[u8]` 这样的共享引用可以指向任意大小的 `[u8]` 切片。因为 `str` 类型和 `[T]` 类型都表示不定大小的值集，所以它们是无固定大小类型。

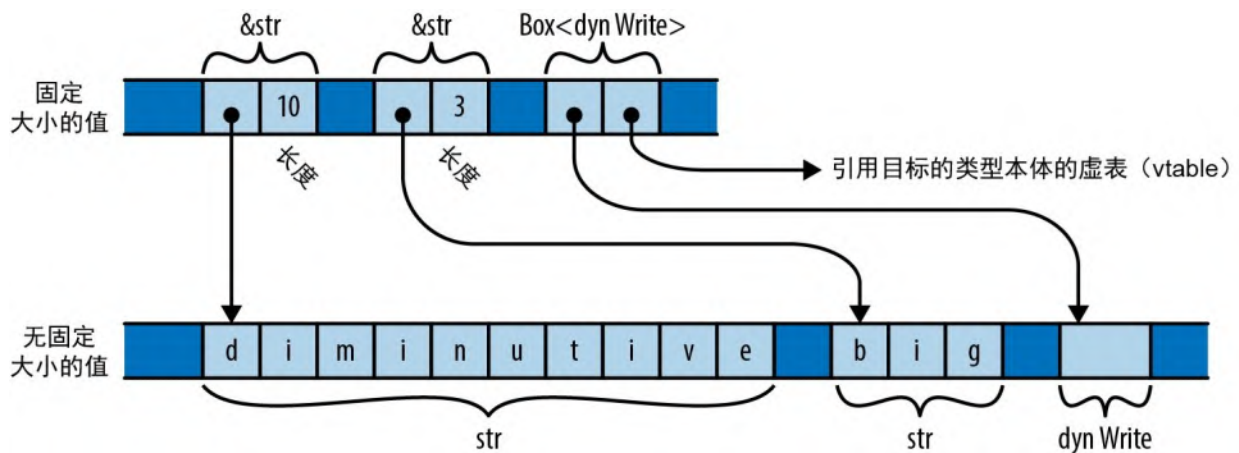


图 13-1：对无固定大小的值的引用

Rust 中另一种常见的无固定大小类型是 `dyn` 类型，它是特型对象的引用目标。正如我们在 11.1.1 节中所解释的那样，特型对象是指向实现了给定特型的某个值的指针。例如，类型 `&dyn std::io::Write` 和 `Box<dyn std::io::Write>` 是指向实现了 `Write` 特型的某个值的指针。引用目标可能是文件、网络套接字，或某种实现了 `Write` 的自定义类型。由于实现了 `Write` 的类型集是开放式的，因此 `dyn Write` 作为一个类型也是无固定大小的，也就是说它的值可以有各种大小。

Rust 不能将无固定大小的值存储在变量中或将它们作为参数传递。你只能通过像 `&str` 或 `Box<dyn Write>` 这样的本身是固定大小的指针来处理它们。如图 13-1 所示，指向无固定大小值的指针始终是一个胖指针，宽度为两个机器字：指向切片的指针带有切片的长度，特型对象带有指向方法实现的虚表的指针。

特型对象和指向切片的指针在结构上很像。这两种类型都缺乏某种在使用它们时必要的信息。换句话说，你无法在不知道其长度的情况下对 `[u8]` 进行索引，也无法在不知道该对某个值使用 `Write` 的哪个具体实现的情况下调用 `Box<dyn Write>` 的方法。对于这两种类型，胖指针都会补齐类型中缺少的信息——它携带着长度或虚表指针。既然欠缺静态信息，那就用动态信息来弥补。

由于无固定大小类型处处受限，因此大多数泛型类型变量应当被限制为固定大小的 `Sized` 类型。事实上，鉴于这种情况的普遍性，`Sized` 已经成为 Rust 中的隐式默认值：如果你写 `struct S<T> { ... }`，那么 Rust 会将其理解为 `struct S<T: Sized> { ... }`。如果你不想以这种方式约束 `T`，就必须将其明确地排除，写成 `struct S<T: ?Sized> { ... }`。`?Sized` 语法专用于这种情况，意思是“不要求固定大小的”。如果你写 `struct S<T: ?Sized> { b: Box<T> }`，那么 Rust 将允许写成 `S<str>` 和 `S<dyn Write>`，这样这两个 `Box` 就变成了胖指针，而不像 `S<i32>` 和 `S<String>` 的 `Box` 那样只是普通指针。

尽管存在一些限制，但无固定大小类型能让 Rust 的类型系统工作得更顺畅。阅读标准库文档时，你偶尔会遇到类型变量上的 `?Sized` 限界，这几乎总是表明“给定的类型只能通过指针访问”，并能让其关联的代码与切片对象和特型对象以及普通值一起使用。当类型变量具有 `?Sized` 限界时，人们认为它的大小不确定，既可能是固定大小，也可能不是。

除了切片对象和特型对象，还有另一种无固定大小类型。结构体类型的最后一个字段（而且只能是最后一个）可以是无固定大小的，并且这样的结构体本身也是无固定大小的。例如，`Rc<T>` 引用计数指针的内部实现是指向私有类型 `RcBox<T>` 的指针，后者把引用计数和 `T` 保存在一起。下面是 `RcBox` 的简化定义：

```
struct RcBox<T: ?Sized> {  
    ref_count: usize,  
    value: T,  
}
```

`Rc<T>` 是引用计数指针，其中的 `value` 字段是 `Rc<T>` 对其进行引用计数的 `T` 类型。`Rc<T>` 会解引用成指向 `value` 字段的指针。`ref_count` 字段会保存其引用计数。

真正的 `RcBox` 只是标准库的一个实现细节，无法在外部使用。但假设我们正在使用前面这种定义，那么就可以将此 `RcBox` 与固定大小类型一起使用，比如 `RcBox<String>` 的结果是一个固定大小的结构

体类型。或者也可以将它与无固定大小类型一起使用，比如 `RcBox<dyn std::fmt::Display>`（其中 `Display` 是可以通过 `println!` 之类的宏进行格式化的类型的特型），结果 `RcBox<dyn Display>` 就成了无固定大小的结构体类型。

不能直接构建 `RcBox<dyn Display>` 值，而应该先创建一个普通的固定大小的 `RcBox`，并让其 `value` 类型实现 `Display`，比如 `RcBox<String>`。然后 Rust 就会允许你将引用 `&RcBox<String>` 转换为胖引用 `&RcBox<dyn Display>`：

```
let boxed_lunch: RcBox<String> = RcBox {
    ref_count: 1,
    value: "lunch".to_string()
};

use std::fmt::Display;
let boxed_displayable: &RcBox<dyn Display> = &boxed_lunch;
```

在将值传给函数时会发生隐式转换，这样你就可以将 `&RcBox<String>` 传给需要 `&RcBox<dyn Display>` 的函数：

```
fn display(boxed: &RcBox<dyn Display>) {
    println!("For your enjoyment: {}", &boxed.value);
}

display(&boxed_lunch);
```

这将生成以下输出。

```
For your enjoyment: lunch
```

13.3 Clone

`std::clone::Clone` 特型适用于可复制自身的类型。`Clone` 定义如下：

```
trait Clone: Sized {
    fn clone(&self) -> Self;
    fn clone_from(&mut self, source: &Self) {
        *self = source.clone()
    }
}
```

`clone` 方法应该为 `self` 构造一个独立的副本并返回它。由于此方法的返回类型是 `Self`，并且函数本来也不可能返回无固定大小的值，因此 `Clone` 特型也是扩展自 `Sized` 特型的，进而导致其实现代码中的 `Self` 类型被限界成了 `Sized`。

克隆一个值通常还需要为它拥有的任何值分配副本，因此 `clone` 无论在时间消耗还是内存占用方面都是相当昂贵的。例如，克隆 `Vec<String>` 不仅会复制此向量，还会复制它的每个 `String` 元素。这就是 Rust 不会自动克隆值，而是要求你进行显式方法调用的原因。像 `Rc<T>` 和 `Arc<T>` 这样的引用计数指针类型属于例外，即克隆其中任何一个都只会增加引用计数并为你返回一个新指针。

`clone_from` 方法会把 `self` 修改成 `source` 的副本。`clone_from` 的默认定义只是克隆 `source`，然后将其转移给 `*self`。这固然可行，但对于某些类型，有一种更快的方法可以获得同样的效果。假设 `s` 和 `t` 都是 `String`。`s = t.clone();` 语句必然会克隆 `t`，丢弃 `s` 的旧值，然后将克隆后的值转移给 `s`，这会进行一次堆分配和一次堆释放。但是如果属于原始 `s` 的堆缓冲区有足够的容量来保存 `t` 的内容，则不需要分配或释放：可以简单地将 `t` 的文本复制到 `s` 的缓冲区并调整长度。在泛型代码中，应该优先使用 `clone_from`，以便充分利用这种优化。

如果你的 `Clone` 实现只需要简单地对类型中的每个字段或元素进行 `clone`，然后从这些克隆结果中构造一个新值，并且认为 `clone_from` 的默认定义已经足够好了，那么 Rust 也可以帮你实现：只要在类型定义上方写 `#[derive(Clone)]` 就可以了。

标准库中几乎所有能合理复制的类型都实现了 `Clone`。不仅 `bool`、`i32` 等原始类型实现了 `Clone`，`String`、`Vec<T>` 和 `HashMap` 等容器类型也实现了 `Clone`。而那些无法合理复制的类型（如 `std::sync::Mutex`）则没有实现 `Clone`。像 `std::fs::File` 这样的类型虽然可以复制，但如果操作系统无法提供必要的资源，则复制可能会失败。这些类型也没有实现 `Clone`，因为 `clone` 必须是不会失败的。作为替代，`std::fs::File` 提供了一个 `try_clone` 方法，该方法会返回一个 `std::io::Result<File>` 值，用以报告失败信息。

13.4 Copy

在第 4 章中，我们曾解释说，对于大多数类型，赋值时会移动值，而不是复制它们。移动值可以更简单地跟踪它们所拥有的资源。但在 4.3 节中，我们指出了例外情况：不拥有任何资源的简单类型可以是 Copy 类型，对这些简单类型赋值会创建源的副本，而不会移动值并使源回到未初始化状态。

当时，我们没有充分解释 Copy 类型到底是什么，现在可以告诉你了：如果一个类型实现了 `std::marker::Copy` 标记特型，那么它就是 Copy 类型，其定义如下所示：

```
trait Copy: Clone { }
```

对于你自己的类型，这当然很容易实现：

```
impl Copy for MyType { }
```

但由于 Copy 是一种对语言有着特殊意义的标记特型，因此只有当类型需要一个浅层的逐字节复制时，Rust 才允许它实现 Copy。拥有任何其他资源（比如堆缓冲区或操作系统句柄）的类型都无法实现 Copy。

任何实现了 Drop 特型的类型都不能是 Copy 类型。Rust 认为如果一个类型需要特殊的清理代码，那么就必然需要特殊的复制代码，因此不能是 Copy 类型。

与 Clone 一样，可以使用 `#[derive(Copy)]` 让 Rust 为你派生出 Copy 实现。你会经常看到同时使用 `#[derive(Copy, Clone)]` 进行派生的代码。

在允许一个类型成为 Copy 类型之前务必慎重考虑。尽管这样做能让该类型更易于使用，但也对其实现施加了严格的限制。如果复制的开

销很高，那么就不适合进行隐式复制。4.3 节曾详细解释过这些因素。

13.5 Deref 与 DerefMut

通过实现 `std::ops::Deref` 特型和 `std::ops::DerefMut` 类型，可以指定像 `*` 和 `.` 这样的解引用运算符在你的类型上的行为。像 `Box<T>` 和 `Rc<T>` 这样的指针类型就实现了这些特型，因此它们可以像 Rust 的内置指针类型那样用。如果你有一个 `Box<Complex>` 型的值 `b`，那么 `*b` 引用的就是 `b` 指向的 `Complex`（复数）值，而 `b.re` 引用的是它的实部。如果上下文对引用目标进行了赋值或借用了可变引用，那么 Rust 就会使用 `DerefMut`（解可变引用）特型，否则，只要通过 `Deref` 进行只读访问就够了。

这两个特型的定义如下所示：

```
trait Deref {
    type Target: ?Sized;
    fn deref(&self) -> &Self::Target;
}

trait DerefMut: Deref {
    fn deref_mut(&mut self) -> &mut Self::Target;
}
```

`deref` 方法会接受 `&Self` 引用并返回 `&Self::Target` 引用，而 `deref_mut` 方法会接受 `&mut Self` 引用并返回 `&mut Self::Target` 引用。`Target` 应该是 `Self` 包含、拥有或引用的资源：对于 `Box<Complex>`，其 `Target` 类型是 `Complex`。请注意 `DerefMut` 扩展了 `Deref`：如果可以解引用并修改某些资源，那么当然也可以借入对它的共享引用。由于这些方法会返回与 `&self` 生命周期相同的引用，因此只要返回的引用还存在，`self` 就会一直处于已借出状态。

`Deref` 特型和 `DerefMut` 特型还扮演着另一个角色。由于 `deref` 会接受 `&Self` 引用并返回 `&Self::Target` 引用，因此 Rust 会利用这一点自动将前一种类型的引用转换为后一种类型的引用。换句

话说，如果只要插入一个 `deref` 调用就能解决类型不匹配问题，那 Rust 就会插入它。实现 `DerefMut` 也可以为可变引用启用相应的转换。这些叫作**隐式解引用**：一种类型被“转换”成了另一种类型。

尽管隐式解引用也可以通过显式的方式写出来，但隐式解引用使用起来更方便。

- 如果你有一个 `Rc<String>` 型的值 `r`，并想对其调用 `String::find`，就可以简单地写成 `r.find('?')`，而不用写成 `(*r).find('?')`：这种方法调用会隐式借入 `r`，并将 `&Rc<String>` 转换为 `&String`，因为 `Rc<T>` 实现了 `Deref<Target=T>`。
- 你可以对 `String` 值使用 `split_at` 之类的方法，虽然 `split_at` 是在 `str` 切片类型上定义的方法，但因为 `String` 实现了 `Deref<Target=str>`，所以可以这样写。`String` 不需要重新实现 `str` 的所有方法，因为可以将 `&String` 隐式转换为 `&str`。
- 如果你有一个字节向量 `v` 并且想将它传给需要字节切片 `&[u8]` 的函数，就可以简单地将 `&v` 作为参数传递，因为 `Vec<T>` 实现了 `Deref<Target=[T]>`。

在必要的情况下，Rust 会连续应用多个隐式解引用。例如，使用前面提到的隐式转换，你可以将 `split_at` 直接应用于 `Rc<String>`，因为 `&Rc<String>` 解引用成了 `&String`，后者又解引用成了 `&str`，而 `&str` 具有 `split_at` 方法。

假设你有以下类型：

```
struct Selector<T> {
    /// 在这个`Selector`中可用的元素
    elements: Vec<T>,

    /// `elements`中“当前”（current）元素的索引
    /// `Selector`的行为类似于指向当前元素的指针
    current: usize
}
```


要让 `Selector` 的行为与文档型注释中声明的一致，就必须为该类型实现 `Deref` 和 `DerefMut`：

```
use std::ops::{Deref, DerefMut};

impl<T> Deref for Selector<T> {
    type Target = T;
    fn deref(&self) -> &T {
        &self.elements[self.current]
    }
}

impl<T> DerefMut for Selector<T> {
    fn deref_mut(&mut self) -> &mut T {
        &mut self.elements[self.current]
    }
}
```

给定上述实现，可以像下面这样使用 `Selector`：

```
let mut s = Selector { elements: vec!['x', 'y', 'z'],
                      current: 2 };

// 因为`Selector`实现了`Deref`，所以可以使用`*`运算符来引用它的当前元素
assert_eq!(*s, 'z');

// 通过隐式解引用直接在`Selector`上使用`char`的方法断言'z'是字母
assert!(s.is_alphabetic());

// 通过对此`Selector`的引用目标赋值，把'z'改成了'w'
*s = 'w';

assert_eq!(s.elements, ['x', 'y', 'w']);
```

`Deref` 特型和 `DerefMut` 特型旨在实现诸如 `Box`、`Rc` 和 `Arc` 之类的智能指针类型，以及其拥有型版本会频繁通过引用来使用的类型（比如 `Vec<T>` 和 `String` 就是 `[T]` 和 `str` 的拥有型版本）。仅仅为了让 `Target` 类型的方法能自动通过类型指针使用（就像 C++ 中那样让基类的方法在子类上可见）就为类型实现 `Deref` 和 `DerefMut` 是不对的。那样做的话并不总能如预期般工作，并且在出错时可能会让人困惑。

隐式解引用有一个容易引起混淆的地方需要注意：Rust 会用它们来解决类型冲突，但并不会将其用于满足类型变量的限界。例如，下面的代码能正常工作：

```
let s = Selector { elements: vec!["good", "bad", "ugly"],
                    current: 2 };

fn show_it(thing: &str) { println!("{}", thing); }
show_it(&s);
```

在调用 `show_it(&s)` 时，Rust 发现了一个类型为 `&Selector<&str>` 的实参（argument）和一个类型为 `&str` 的形参（parameter），据此找到了这个 `Deref<Target=str>` 实现，并根据需要将此调用重写成了 `show_it(s.deref())`。¹

¹在需要做出区分的上下文中，本书选择将 `argument` 和 `parameter` 分别译为“实参”和“形参”。除此之外，在表示一般性概念或不会导致混淆的情况下，二者均译为“参数”。——编者注

但是，如果将 `show_it` 改成泛型函数，Rust 突然就报错了：

```
use std::fmt::Display;
fn show_it_generic<T: Display>(thing: T) { println!("{}", thing); }
show_it_generic(&s);
```

Rust 报错说：

```
error: `Selector<&str>` doesn't implement `std::fmt::Display`
  |
31 |     show_it_generic(&s);
  |                      ^^
  |                      |
  |                      `Selector<&str>` cannot be formatted with
  |                      the default formatter
  |                      help: consider adding dereference here:
  |                      `&*s`
note: required by a bound in `show_it_generic`
  |
30 |     fn show_it_generic<T: Display>(thing: T) { println!("{}",
```

```
thing); }  
|  
|  
|  
^^^^^^ required by this bound  
in `show_it_generic`
```

这可能会令人困惑：为什么仅仅把函数改成泛型形式就会引入错误呢？`Selector<&str>` 本身确实没有实现 `Display`，但它解引用成了 `&str`，而 `&str` 实现了 `Display`。

由于你要传入一个类型为 `&Selector<&str>` 的实参并且函数的形参类型为 `&T`，因此类型变量 `T` 必然是 `Selector<&str>`。然后，Rust 会检查这是否满足 `T: Display` 限界，但因为它不会通过隐式解引用来满足类型变量的限界，所以这个检查失败了。

要解决此问题，可以使用 `as` 运算符进行显式转换：

```
show_it_generic(&s as &str);
```

或者，正如编译器建议的那样，可以使用 `&*` 进行强制转换。

```
show_it_generic(&*s);
```

13.6 Default

显然，某些类型具有合理的默认值：向量或字符串默认为空、数值默认为 0、Option 默认为 None，等等。这样的类型都可以实现 `std::default::Default` 特型：

```
trait Default {  
    fn default() -> Self;  
}
```

`default` 方法只会返回一个 `Self` 类型的新值。为 `String` 实现 `Default` 的代码一目了然：

```
impl Default for String {  
    fn default() -> String {  
        String::new()  
    }  
}
```

Rust 的所有集合类型（Vec、HashMap、BinaryHeap 等）都实现了 `Default`，其 `default` 方法会返回一个空集合。当你需要构建一些值的集合但又想让调用者来决定具体构建何种集合时，这很有用。例如，`Iterator` 特型的 `partition` 方法会将迭代器生成的值分为两个集合，并使用闭包来决定每个值的去向：

```
use std::collections::HashSet;  
let squares = [4, 9, 16, 25, 36, 49, 64];  
let (powers_of_two, impure): (HashSet<i32>, HashSet<i32>)  
    = squares.iter().partition(|&n| n & (n-1) == 0);  
assert_eq!(powers_of_two.len(), 3);  
assert_eq!(impure.len(), 4);
```

闭包 `|&n| n & (n-1) == 0` 会使用一些位操作来识别哪些数值是 2 的幂，并且 `partition` 会使用它来生成两个 `HashSet`。不过，`partition` 显然不是专属于 `HashSet` 的，你可以用它来生成想要的任何种类的集合，只要该集合类型能够实现 `Default` 以生成一个

初始的空集合，并且实现 `Extend<T>` 以将 `T` 添加到集合中就可以。`String` 实现了 `Default` 和 `Extend<char>`，所以你可以这样写：

```
let (upper, lower): (String, String)
    = "Great Teacher Onizuka".chars().partition(|&c|
c.is_uppercase());
assert_eq!(upper, "GTO");
assert_eq!(lower, "reat eacher nizuka");
```

`Default` 的另一个常见用途是为表示大量参数集合的结构体生成默认值，其中大部分参数通常不用更改。例如，`glium crate` 为强大而复杂的 OpenGL 图形库提供了 Rust 绑定。`glium::DrawParameters` 结构体包括 24 个字段，每个字段控制着 OpenGL 应该如何渲染某些图形的不同细节。`glium draw` 函数需要一个 `DrawParameters` 结构体作为参数。由于 `DrawParameters` 已经实现了 `Default`，因此只需提及想要更改的那些字段即可创建一个可以传给 `draw` 的结构体：

```
let params = glium::DrawParameters {
    line_width: Some(0.02),
    point_size: Some(0.02),
    .. Default::default()
};

target.draw(..., &params).unwrap();
```

这会调用 `Default::default()` 来创建一个 `DrawParameters` 值，该值会使用其所有字段的默认值进行初始化，然后使用结构体的 `..` 语法创建一个更改了 `line_width` 字段和 `point_size` 字段的新值，最后就可以把它传给 `target.draw` 了。

如果类型 `T` 实现了 `Default`，那么标准库就会自动为 `Rc<T>`、`Arc<T>`、`Box<T>`、`Cell<T>`、`RefCell<T>`、`Cow<T>`、`Mutex<T>` 和 `RwLock<T>` 实现 `Default`。例如，类型 `Rc<T>` 的默认值就是一个指向类型 `T` 的默认值的 `Rc`。

如果一个元组类型的所有元素类型都实现了 `Default`，那么该元组类型也同样会实现 `Default`，这个元组的默认值包含每个元素的默认值。

Rust 不会为结构体类型隐式实现 `Default`，但是如果结构体的所有字段都实现了 `Default`，则可以使用 `#[derive(Default)]` 为此结构体自动实现 `Default`。

13.7 AsRef 与 AsMut

如果一个类型实现了 `AsRef<T>`，那么就意味着你可以高效地从中借入 `&T`。`AsMut` 是 `AsRef` 针对可变引用的对应类型。它们的定义如下所示：

```
trait AsRef<T: ?Sized> {  
    fn as_ref(&self) -> &T;  
}  
  
trait AsMut<T: ?Sized> {  
    fn as_mut(&mut self) -> &mut T;  
}
```

例如，`Vec<T>` 实现了 `AsRef<[T]>`，而 `String` 实现了 `AsRef<str>`。还可以把 `String` 的内容借入为字节数组，因此 `String` 也实现了 `AsRef<[u8]>`。

`AsRef` 通常用于让函数更灵活地接受其参数类型。例如，`std::fs::File::open` 函数的声明如下：

```
fn open<P: AsRef<Path>>>(path: P) -> Result<File>
```

`open` 真正想要的是 `&Path`，即代表文件系统路径的类型。有了这个函数签名，`open` 就能接受可以从中借入 `&Path` 的一切，也就是实现了 `AsRef<Path>` 的一切。这些类型包括 `String` 和 `str`、操作系统接口字符串类型 `OsString` 和 `OsStr`，当然还有 `PathBuf` 和 `Path`。有关完整列表，请参阅标准库的文档。这样你才能给 `open` 传入字符串字面量：

```
let dot_emacs = std::fs::File::open("/home/jimb/.emacs")?;
```

标准库的所有文件系统访问函数都会以这种方式接受路径参数。对调用者来说，其效果类似于 C++ 中的重载函数，只不过 Rust 采用的是另一种方式来确定可接受的参数类型。

但这还不是全部。字符串字面量是 `&str`，实现了 `AsRef<Path>` 的类型是 `str`，并没有 `&`。正如我们在 13.5 节中解释的那样，Rust 不会试图通过隐式解引用来满足类型变量限界，因此就算它们在这里也无济于事。

幸运的是，标准库包含了其通用实现：

```
impl<'a, T, U> AsRef<U> for &'a T
    where T: AsRef<U>,
           T: ?Sized, U: ?Sized
{
    fn as_ref(&self) -> &U {
        (*self).as_ref()
    }
}
```

换句话说，对于任意类型 `T` 和 `U`，只要满足 `T: AsRef<U>`，就必然满足 `&T: AsRef<U>`：只需追踪引用并像以前那样继续处理即可。特别是，如果满足 `str: AsRef<Path>`，那么也会满足 `&str: AsRef<Path>`。从某种意义上说，这是一种在检查类型变量的 `AsRef` 限界时获得受限隐式解引用的方法。

你可能会认为，如果一个类型实现了 `AsRef<T>`，那么它也应该实现 `AsMut<T>`。但是，这在某些情况下是不合适的。例如，我们已经提到 `String` 实现了 `AsRef<[u8]>`，这是合理的，因为每个 `String` 肯定都有一个可以作为二进制数据访问的字节缓冲区。但是，`String` 要进一步保证这些字节是表示 Unicode 文本的一段格式良好的 UTF-8 编码，如果 `String` 实现了 `AsMut<[u8]>`，那么就会允许调用者将 `String` 的字节更改为他们想要的任何内容，这样你就不能再相信 `String` 一定是格式良好的 UTF-8 了。只有修改给定的 `T` 肯定不会违反此类型的不变性约束时，实现 `AsMut<T>` 的类型才有意义。

尽管 `AsRef` 和 `AsMut` 非常简单，但为引用转换提供标准的泛型特型可避免更专用的转换特型数量激增。只要能实现 `AsRef<Foo>`，就要尽量避免定义自己的 `AsFoo` 特型。

13.8 Borrow 与 BorrowMut

`std::borrow::Borrow` 特型类似于 `AsRef`：如果一个类型实现了 `Borrow<T>`，那么它的 `borrow` 方法就能高效地从自身借入一个 `&T`。但是 `Borrow` 施加了更多限制：只有当 `&T` 能通过与它借来的值相同的方式进行哈希和比较时，此类型才应实现 `Borrow<T>`。

（Rust 并不强制执行此限制，它只是记述了此特型的意图。）这使得 `Borrow` 在处理哈希表和树中的键或者处理因为某些原因要进行哈希或比较的值时非常有用。

这在区分对 `String` 的借用时很重要，比如 `String` 实现了 `AsRef<str>`、`AsRef<[u8]>` 和 `AsRef<Path>`，但这 3 种目标类型通常具有不一样的哈希值。只有 `&str` 切片才能保证像其等效的 `String` 一样进行哈希，因此 `String` 只实现了 `Borrow<str>`。

`Borrow` 的定义与 `AsRef` 的定义基本相同，只是名称变了：

```
trait Borrow<Borrowed: ?Sized> {  
    fn borrow(&self) -> &Borrowed;  
}
```

`Borrow` 旨在解决具有泛型哈希表和其他关联集合类型的特定情况。假设你有一个 `std::collections::HashMap<String, i32>`，用于将字符串映射到数值。这个表的键是 `String`，每个条目都有一个键。在这个表中查找某个条目的方法的签名应该是什么呢？下面是第一次尝试。

```
impl<K, V> HashMap<K, V> where K: Eq + Hash  
{  
    fn get(&self, key: K) -> Option<&V> { ... }  
}
```

这很合理：要查找条目，就必须为表提供适当类型的键。但在这里，`K` 是 `String`，这种签名会强制你将 `String` 按值传给对 `get` 的每次调用，这显然是一种浪费。你真正需要的只是此键的引用：

```
impl<K, V> HashMap<K, V> where K: Eq + Hash
{
    fn get(&self, key: &K) -> Option<&V> { ... }
}
```

这稍微好一点儿了，但现在你必须将键作为 `&String` 传递，所以如果想查找常量字符串，就必须像下面这样写。

```
hashtable.get(&"twenty-two".to_string())
```

这相当荒谬：它会在堆上分配一个 `String` 缓冲区并将文本复制进去，这样才能将其作为 `&String` 借用出来，传给 `get`，然后将其丢弃。

它应该只要求传入任何可以哈希并与我们的键类型进行比较的类型。例如，`&str` 就完全够用了。所以下面是最后一次迭代，也正是你在标准库中所看到的：

```
impl<K, V> HashMap<K, V> where K: Eq + Hash
{
    fn get<Q: ?Sized>(&self, key: &Q) -> Option<&V>
        where K: Borrow<Q>,
              Q: Eq + Hash
    { ... }
}
```

换句话说，只要可以借入一个条目的键充当 `&Q`，并且对生成的引用进行哈希和比较的方式与键本身一致，`&Q` 显然就是可接受的键类型。由于 `String` 实现了 `Borrow<str>` 和 `Borrow<String>`，因此最终版本的 `get` 允许按需传入 `&String` 型或 `&str` 型的 `key`。

`Vec<T>` 和 `[T; N]` 实现了 `Borrow<[T]>`。每个类似字符串的类型都能借入其相应的切片类型：`String` 实现了 `Borrow<str>`、`PathBuf` 实现了 `Borrow<Path>`，等等。标准库中所有关联集合类型都使用 `Borrow` 来决定哪些类型可以传给它们的查找函数。

标准库中包含一个通用实现，因此每个类型 `T` 都可以从自身借用：`T: Borrow<T>`。这确保了在 `HashMap<K, V>` 中查找条目时 `&K` 总是可接受的类型。

为便于使用，每个 `&mut T` 类型也都实现了 `Borrow<T>`，它会像往常一样返回一个共享引用 `&T`。这样你就可以给集合的查找函数传入可变引用，而不必重新借入共享引用，以模拟 Rust 通常会从可变引用到共享引用进行的隐式转换。

`BorrowMut` 特型则类似于针对可变引用的 `Borrow`：

```
trait BorrowMut<Borrowed: ?Sized>: Borrow<Borrowed> {  
    fn borrow_mut(&mut self) -> &mut Borrowed;  
}
```

刚才讲过的对 `Borrow` 的要求同样适用于 `BorrowMut`。

13.9 From 与 Into

`std::convert::From` 特型和 `std::convert::Into` 特型表示类型转换，这种转换会接受一种类型的值并返回另一种类型的值。`AsRef` 特型和 `AsMut` 特型用于从一种类型借入另一种类型的引用，而 `From` 和 `Into` 会获取其参数的所有权，对其进行转换，然后将转换结果的所有权返回给调用者。

`From` 和 `Into` 的定义是对称的：

```
trait Into<T>: Sized {
    fn into(self) -> T;
}

trait From<T>: Sized {
    fn from(other: T) -> Self;
}
```

标准库自动实现了从每种类型到自身的简单转换：每种类型 `T` 都实现了 `From<T>` 和 `Into<T>`。

虽然这两个特型看起来只是为做同一件事提供了两种方式，但其实它们有不同的用途。

你通常可以使用 `Into` 来让你的函数在接受参数时更加灵活。如果你写如下代码：

```
use std::net::Ipv4Addr;

fn ping<A>(address: A) -> std::io::Result<bool>
    where A: Into<Ipv4Addr>
{
    let ipv4_address = address.into();
    ...
}
```

那么 `ping` 不仅可以接受 `Ipv4Addr` 作为参数，还可以接受 `u32` 或 `[u8; 4]` 数组，因为这些类型都恰好实现了 `Into<Ipv4Addr>`。（有时将 IPv4 地址视为单个 32 位值或 4 字节数组会很有用。）因为 `ping` 对 `address` 的唯一了解就是它要实现 `Into<Ipv4Addr>`，所以在调用 `into` 时无须指定想要的是哪种类型。因为只会存在一种有效类型，所以类型推断会替你补全它。

与 13.7 节中的 `AsRef` 一样，其效果很像 C++ 中的函数重载。使用之前的 `ping` 定义，可以进行以下任何调用：

```
println!("{:?}", ping(Ipv4Addr::new(23, 21, 68, 141))); // 传入一个
Ipv4Addr
println!("{:?}", ping([66, 146, 219, 98]));             // 传入一个
[u8; 4]
println!("{:?}", ping(0xd076eb94_u32));                 // 传入一个
u32
```

而 `From` 特型扮演着另一种角色。`from` 方法会充当泛型构造函数，用于从另一个值生成本类型的实例。例如，虽然 `Ipv4Addr` 有两个名为 `from_array` 和 `from_u32` 的方法，但 `From` 只是简单地实现了 `From<[u8;4]>` 和 `From<u32>`，于是我们就能这样写：

```
let addr1 = Ipv4Addr::from([66, 146, 219, 98]);
let addr2 = Ipv4Addr::from(0xd076eb94_u32);
```

可以让类型推断找出适用于此的实现。

给定适当的 `From` 实现，标准库会自动实现相应的 `Into` 特型。当你定义自己的类型时，如果它具有某些单参数构造函数，那么就应该将它们写成适当类型的 `From<T>` 的实现，这样你就会自动获得相应的 `Into` 实现。

因为转换方法 `from` 和 `into` 会接手它们的参数的所有权，所以此转换可以复用原始值的资源来构造出转换后的值。假设你写如下代码：

```
let text = "Beautiful Soup".to_string();  
let bytes: Vec<u8> = text.into();
```

`String` 的 `Into<Vec<u8>>` 的实现只是获取 `String` 的堆缓冲区，并在不进行任何更改的情况下将其重新用作所返回向量的元素缓冲区。此转换既不需要分配内存，也不需要复制文本。这是通过移动进行高性能实现的另一个例子。

这些转换还提供了一种很好的方式来将受限类型的值放宽为更灵活的值，而不会削弱受限类型提供的保证。例如，`String` 会保证其内容始终是有效的 UTF-8，它的可变方法会受到严格限制，以确保你所做的任何事情都不会引入错误的 UTF-8。但是这个例子有效地将 `String` “降级”为一个普通字节块，你可以用它做任何喜欢的事情：既可以压缩它，也可以将它与其他非 UTF-8 的二进制数据组合使用。因为 `into` 会按值接手其参数，所以转换后的 `text` 就成了未初始化状态，这意味着我们可以自由访问前一个 `String` 的缓冲区，而不会破坏任何现有 `String`。

然而，`Into` 和 `From` 契约并不要求这种转换是低开销的。尽管对 `AsRef` 和 `AsMut` 的转换可以预期开销极低，但 `From` 和 `Into` 的转换可能会分配内存、复制或以其他方式处理值的内容。例如，`String` 实现了 `From<&str>`，它会将字符串切片复制到 `String` 在堆上分配的新缓冲区中。

`std::collections::BinaryHeap<T>` 实现了 `From<Vec<T>>`，它能根据算法的要求对元素进行比较和重新排序。

通过在需要时自动将具体错误类型转换为一般错误类型，运算符 `?` 可以使用 `From` 和 `Into` 来帮助清理可能以多种方式失败的函数中的代码。

假设一个系统需要读取二进制数据并将其中的某些部分从 UTF-8 文本中作为十进制数值转换出来。这意味着要使用 `std::str::from_utf8` 和 `i32` 的 `FromStr` 实现，它们都可以返回不同类型的错误。假设讨论错误处理时使用的是第 7 章中定义的

GenericError 类型和 GenericResult 类型，那么运算符 `?` 将为我们进行这种转换：

```
type GenericError = Box<dyn std::error::Error + Send + Sync +
'static>;
type GenericResult<T> = Result<T, GenericError>;

fn parse_i32_bytes(b: &[u8]) -> GenericResult<i32> {
    Ok(std::str::from_utf8(b)?.parse::<i32>())?
}
```

与大多数错误类型一样，Utf8Error 和 ParseIntError 也实现了 Error 特型，标准库为我们提供了 From 的通用实现，用于将任何实现了 Error 的类型转换为 Box<dyn Error> 类型，`?` 运算符会自动使用这种转换：

```
impl<'a, E: Error + Send + Sync + 'a> From<E>
for Box<dyn Error + Send + Sync + 'a> {
    fn from(err: E) -> Box<dyn Error + Send + Sync + 'a> {
        Box::new(err)
    }
}
```

这能把具有两个 match 语句的大函数变成单行函数。

在 From 和 Into 被加入标准库之前，Rust 代码充满了专用的转换特型和构造方法，每一个都专用于一种类型。为了让你的类型更容易使用，From 和 Into 明确写出了可以遵循的约定，因为你的用户已经熟悉它们了。其他库以及语言自身也可以依赖这些特型，将其作为一种规范化、标准化的方式来对转换进行编码。

From 和 Into 是不会失败的特型——它们的 API 要求这种转换不会失败。不过很遗憾，许多转换远比这复杂得多。例如，像 i64 这样的大整数可以存储比 i32 大得多的数值，如果没有一些额外的信息，那么将像 2_000_000_000_000i64 这样的数值转换成 i32 就没有多大意义。如果进行简单的按位转换，那么其中前 32 位就会被丢弃，通常不会产生我们预期的结果：

```
let huge = 2_000_000_000_000i64;  
let smaller = huge as i32;  
println!("{}", smaller); // -1454759936
```

有很多选项可以处理这种情况。根据上下文的不同，“回绕型”转换可能比较合适。另外，像数字信号处理和控制系统这样的应用程序通常会使用“饱和型”转换，它会把比可能的最大值还要大的数值限制为最大值。

13.10 TryFrom 与 TryInto

由于转换的行为方式不够清晰，因此 Rust 没有为 `i32` 实现 `From<i64>`，也没有实现任何其他可能丢失信息的数值类型之间的转换，而是为 `i32` 实现了 `TryFrom<i64>`。`TryFrom` 和 `TryInto` 是 `From` 和 `Into` 的容错版“表亲”，这种转换同样是双向的，实现了 `TryFrom` 也就意味着实现了 `TryInto`。

`TryFrom` 和 `TryInto` 的定义比 `From` 和 `Into` 稍微复杂一点儿。

```
pub trait TryFrom<T>: Sized {
    type Error;
    fn try_from(value: T) -> Result<Self, Self::Error>;
}

pub trait TryInto<T>: Sized {
    type Error;
    fn try_into(self) -> Result<T, Self::Error>;
}
```

`try_into()` 方法给了我们一个 `Result`，因此我们可以选择在异常情况下该怎么做，比如处理一个因为太大而无法放入结果类型的数值：

```
// 溢出时饱和，而非回绕
let smaller: i32 = huge.try_into().unwrap_or(i32::MAX);
```

如果还想处理负数的情况，那么可以使用 `Result` 的 `unwrap_or_else()` 方法：

```
let smaller: i32 = huge.try_into().unwrap_or_else(|_|{
    if huge >= 0 {
        i32::MAX
    } else {
        i32::MIN
    }
});
```

为你自己的类型实现容错的转换也很容易。Error 类型既可以很简单，也可以很复杂，具体取决于特定应用程序的要求。标准库使用的是一个空结构体，除了发生过错误这一事实之外没有提供任何信息，因为唯一可能的错误就是溢出。另外，更复杂类型之间的转换可能需要返回更多信息：

```
impl TryInto<LinearShift> for Transform {
    type Error = TransformError;

    fn try_into(self) -> Result<LinearShift, Self::Error> {
        if !self.normalized() {
            return Err(TransformError::NotNormalized);
        }
        ...
    }
}
```

From 和 Into 可以将类型与简单转换关联起来，而 TryFrom 和 TryInto 通过 Result 提供的富有表现力的错误处理扩展了 From 和 Into 的简单转换。这 4 个特型可以一起使用，在同一个 crate 中关联多个类型。

13.11 ToOwned

给定一个引用，如果此类型实现了 `std::clone::Clone`，则生成其引用目标的拥有型副本的常用方法是调用 `clone`。但是当你想克隆一个 `&str` 或 `&[i32]` 时该怎么办呢？你想要的可能是 `String` 或 `Vec<i32>`，但 `Clone` 的定义不允许这样做：根据定义，克隆 `&T` 必须始终返回 `T` 类型的值，并且 `str` 和 `[u8]` 是无固定大小类型，它们甚至都不是函数所能返回的类型。

`std::borrow::ToOwned` 特型提供了一种稍微宽松的方式来将引用转换为拥有型的值：

```
trait ToOwned {  
    type Owned: Borrow<Self>;  
    fn to_owned(&self) -> Self::Owned;  
}
```

与必须精确返回 `Self` 类型的 `clone` 不同，`to_owned` 可以返回任何能让你从中借入 `&Self` 的类型：Owned 类型必须实现 `Borrow<Self>`。你可以从 `Vec<T>` 借入 `&[T]`，所以只要 `T` 实现了 `Clone`，`[T]` 就能实现 `ToOwned<Owned=Vec<T>>`，这样就可以将切片的元素复制到向量中了。同样，`str` 实现了 `ToOwned<Owned=String>`，`Path` 实现了 `ToOwned<Owned=PathBuf>`，等等。

13.12 Borrow 与 ToOwned 的实际运用：谦卑²的 Cow

²指不会主动占有资源，直到确有必要。——译者注

要想用好 Rust，就必然涉及对所有权问题的透彻思考，比如函数应该通过引用还是值接受参数。通常你可以任选一种方式，让参数的类型反映你的决定。但在某些情况下，在程序开始运行之前你无法决定是该借用还是该拥有，`std::borrow::Cow` 类型（用于“写入时克隆”，clone on write 的缩写）提供了一种兼顾两者的方式。

`std::borrow::Cow` 的定义如下所示：

```
enum Cow<'a, B: ?Sized>
    where B: ToOwned
{
    Borrowed(&'a B),
    Owned(<B as ToOwned>::Owned),
}
```

`Cow<B>` 要么借入对 `B` 的共享引用，要么拥有可供借入此类引用的值。由于 `Cow` 实现了 `Deref`，因此你可以像对 `B` 的共享引用一样调用它的方法：如果它是 `Owned`，就会借入对拥有值的共享引用；如果它是 `Borrowed`，就会转让自己持有的引用。

还可以通过调用 `Cow` 的 `to_mut` 方法来获取对 `Cow` 值的可变引用，这个方法会返回 `&mut B`。如果 `Cow` 恰好是 `Cow::Borrowed`，那么 `to_mut` 只需调用引用的 `to_owned` 方法来获取其引用目标的副本，将 `Cow` 更改为 `Cow::Owned`，并借入对新创建的这个拥有型值的可变引用即可。这就是此类型名称所指的“写入时克隆”行为。

类似地，`Cow` 还有一个 `into_owned` 方法，该方法会在必要时提升对所拥有值的引用并返回此引用，这会将所有权转移给调用者并在此过程中消耗掉 `Cow`。

Cow 的一个常见用途是返回静态分配的字符串常量或由计算得来的字符串。假设你需要将错误枚举转换为错误消息。大多数变体可以用固定字符串来处理，但有些也需要在消息中包含附加数据。你可以返回一个 Cow<'static, str>:

```
use std::path::PathBuf;
use std::borrow::Cow;
fn describe(error: &Error) -> Cow<'static, str> {
    match *error {
        Error::OutOfMemory => "out of memory".into(),
        Error::StackOverflow => "stack overflow".into(),
        Error::MachineOnFire => "machine on fire".into(),
        Error::Unfathomable => "machine bewildered".into(),
        Error::FileNotFound(ref path) => {
            format!("file not found: {}", path.display()).into()
        }
    }
}
```

上述代码使用了 Cow 的 Into 实现来构造出值。此 match 语句的大多数分支会返回 Cow::Borrowed 来引用静态分配的字符串。但是当我们得到一个 FileNotFound 变体时，会使用 format! 来构建包含给定文件名的消息。match 语句的这个分支会生成一个 Cow::Owned 值。

如果 describe 的调用者不打算更改值，就可以直接把此 Cow 看作 &str:

```
println!("Disaster has struck: {}", describe(&error));
```

如果调用者确实需要一个拥有型的值，那么也能很容易地生成一个:

```
let mut log: Vec<String> = Vec::new();
...
log.push(describe(&error).into_owned());
```

使用 Cow, describe 及其调用者可以把分配的时机推迟到确有必要的时候。

第 14 章 闭包

拯救¹环境！现在就创建闭包！

——Cormac Flanagan

¹save（有“拯救”和“保存”两个含义）是英文双关语。——译者注

对整型向量进行排序很容易：

```
integers.sort();
```

遗憾的是，当我们想对一些数据进行排序时，它们几乎从来都不是整型向量。例如，对某种记录型数据来说，内置的 `sort` 方法一般不适用：

```
struct City {
    name: String,
    population: i64,
    country: String,
    ...
}

fn sort_cities(cities: &mut Vec<City>) {
    cities.sort(); // 出错：你到底想怎么排序？
}
```

Rust 会报错说 `City` 没有实现 `std::cmp::Ord`。我们需要指定排序顺序，如下所示：

```
/// 按照人口数量对城市进行排序的辅助函数
fn city_population_descending(city: &City) -> i64 {
    -city.population
}

fn sort_cities(cities: &mut Vec<City>) {
```

```
cities.sort_by_key(city_population_descending); // 正确
}
```

辅助函数 `city_population_descending` 会接受 `City` 型记录并提取其键，该键是我们对数据进行排序时要依据的字段。（它会返回一个负数，因为 `sort` 会按升序排列数值，而我们想要按降序排列：让人口最多的城市在前。）`sort_by_key` 方法会将这个取键函数作为参数。

这固然可行，但如果将辅助函数写成闭包（匿名函数表达式）则会更简洁：

```
fn sort_cities(cities: &mut Vec<City>) {
    cities.sort_by_key(|city| -city.population);
}
```

这里的 `|city| -city.population` 就是闭包。它会接受一个参数 `city` 并返回 `-city.population`。Rust 会从闭包的使用方式中推断出其参数类型和返回类型。下面是标准库中接受闭包的其他例子。

- 像 `map` 和 `filter` 这样的 `Iterator` 方法，可用于处理序列数据。第 15 章会介绍这些方法。
- 像 `thread::spawn` 这样的线程 API，会启动一个新的系统线程。并发就是要将工作转移给其他线程，而闭包能方便地表示这些工作单元。第 19 章会介绍这些特性。
- 一些需要根据条件计算默认值的方法，比如 `HashMap` 条目的 `or_insert_with` 方法。此方法用于获取或创建 `HashMap` 中的条目，当默认值的计算成本很高时就要使用闭包。默认值会作为闭包传入，只有当不得不创建新条目时才会调用此闭包。

当然，如今匿名函数无处不在，甚至连 Java、C#、Python、C++ 等最初没有匿名函数的语言中也有了其“身影”。从现在开始，我们假定你之前已经了解匿名函数，只想知道 Rust 的闭包与匿名函数有何不同。本章将介绍 3 种类型的闭包，你将学习如何将闭包与标准库方法一起使用、闭包如何“捕获”其作用域内的变量、如何编写自己的以

闭包作为参数的函数和方法，以及如何存储闭包供以后用作回调。我们还将解释 Rust 闭包是如何实现的，以及它们为什么比你预想的要快。

14.1 捕获变量

闭包可以使用属于其所在函数的数据：

```
/// 根据任何其他的统计标准排序
fn sort_by_statistic(cities: &mut Vec<City>, stat: Statistic) {
    cities.sort_by_key(|city| -city.get_statistic(stat));
}
```

这里的闭包使用了 `stat`，该函数由其所在函数 `sort_by_statistic` 所拥有。于是我们说这个闭包“捕获”了 `stat`。这是闭包的经典特性之一，Rust 当然也支持。但在 Rust 中，此特性略有不同。

在大多数支持闭包的语言中，垃圾回收扮演着重要角色。例如，考虑下面这段 JavaScript 代码：

```
// 启动重新排列城市所在表行的动画
function startSortingAnimation(cities, stat) {
    // 用来对表格进行排序的辅助函数
    // 注意此函数引用了 stat
    function keyfn(city) {
        return city.get_statistic(stat);
    }

    if (pendingSort)
        pendingSort.cancel();

    // 现在开始动画，把 keyfn 传给它
    // 排序算法稍后会调用 keyfn
    pendingSort = new SortingAnimation(cities, keyfn);
}
```

闭包 `keyfn` 存储在了新的 `SortingAnimation` 对象中。当 `startSortingAnimation` 返回后就会调用它。通常，当一个函数返回时，它的所有变量和参数都会超出作用域并被丢弃。但是在这里，JavaScript 引擎必须以某种方式保留 `stat`，因为闭包会使用

它。大多数 JavaScript 引擎的实现方式是在堆中分配 `stat` 并等垃圾回收器稍后回收。

Rust 没有垃圾回收。这可如何是好？为了回答这个问题，我们来看两个例子。

14.1.1 借用值的闭包

首先，我们重复一下本节开头的例子：

```
/// 根据任何其他的统计标准排序
fn sort_by_statistic(cities: &mut Vec<City>, stat: Statistic) {
    cities.sort_by_key(|city| -city.get_statistic(stat));
}
```

在这种情况下，当 Rust 创建闭包时，会自动借入对 `stat` 的引用。这很合理，因为闭包引用了 `stat`，所以闭包必须包含对 `stat` 的引用。

剩下的就简单了。闭包同样遵循第 5 章中讲过的关于借用和生命周期的规则。特别是，由于闭包中包含对 `stat` 的引用，因此 Rust 不会让它的生命周期超出 `stat`。因为闭包只会在排序期间使用，所以这个例子是适用的。

简而言之，Rust 会使用生命周期而非垃圾回收来确保安全。Rust 的方式更快，因为即使是最快的垃圾回收器在分配内存时也会比把 `stat` 保存在栈上慢，本例中 Rust 就把 `stat` 保存在栈上。

14.1.2 “窃取”值的闭包

第二个例子比较棘手：

```
use std::thread;

fn start_sorting_thread(mut cities: Vec<City>, stat: Statistic)
    -> thread::JoinHandle<Vec<City>>
{
    ...
}
```

```

    let key_fn = |city: &City| -> i64 { -city.get_statistic(stat)
};

    thread::spawn(|| {
        cities.sort_by_key(key_fn);
        cities
    })
}

```

这有点儿像我们的 JavaScript 示例所做的：thread::spawn 会接受一个闭包并在新的系统线程中调用它。请注意 || 是闭包的空参数列表。

新线程会和调用者并行运行。当闭包返回时，新线程退出。（闭包的返回值会作为 JoinHandle 值发送回调用线程。第 19 章会介绍 JoinHandle。）

同样，闭包 key_fn 包含对 stat 的引用。但这一次，Rust 不能保证此引用的安全使用。因此 Rust 会拒绝编译这个程序：

```

error: closure may outlive the current function, but it borrows
`stat`,
      which is owned by the current function
|
33 | let key_fn = |city: &City| -> i64 { -city.get_statistic(stat)
|;
|           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
|           |                                                                 `stat` is
borrowed here                                                                    may outlive borrowed value `stat`
|

```

其实这里还有一个问题，因为 cities 也被不安全地共享了。简单来说，thread::spawn 创建的新线程无法保证在 cities 和 stat 被销毁之前在函数末尾完成其工作。

这两个问题的解决方案是一样的：要求 Rust 将 cities 和 stat 移动到使用它们的闭包中，而不是借入对它们的引用。

```

fn start_sorting_thread(mut cities: Vec<City>, stat: Statistic)
-> thread::JoinHandle<Vec<City>>

```

```
{
    let key_fn = move |city: &City| -> i64 { -
city.get_statistic(stat) };

    thread::spawn(move || {
        cities.sort_by_key(key_fn);
        cities
    })
}
```

这里唯一的改动是在两个闭包之前都添加了 `move` 关键字。`move` 关键字会告诉 Rust，闭包并不是要借入它用到的变量，而是要“窃取”它们。

第一个闭包 `key_fn` 取得了 `stat` 的所有权。第二个闭包则取得了 `cities` 和 `key_fn` 的所有权。

因此，Rust 为闭包提供了两种从封闭作用域中获取数据的方法：移动和借用。实际上，关于移动和借用，闭包所遵循的正是第 4 章和第 5 章中已经介绍过的规则。下面举几个例子。

- 就像 Rust 这门语言中的其他地方一样，如果闭包要移动可复制类型的值（如 `i32`），那么就会复制该值。因此，如果 `Statistic` 恰好是可复制类型，那么即使在创建了要使用 `stat` 的 `move` 闭包之后，我们仍可以继续使用 `stat`。
- 不可复制类型的值（如 `Vec<City>`）则确实会被移动，比如前面的代码就通过 `move` 闭包将 `cities` 转移给了新线程。在创建此闭包后，Rust 就不允许再通过 `cities` 访问它了。
- 实际上，在闭包将 `cities` 移动之后，此代码就不需要再使用它了。但是，即使我们确实需要在此之后使用 `cities`，解决方法也很简单：可以要求 Rust 克隆 `cities` 并将副本存储在另一个变量中。闭包将只会“窃取”其中一个副本，即它所引用的那个副本。

通过遵循 Rust 的严格规则，我们也收获颇丰，那就是线程安全。正是因为向量是被移动的，而不是跨线程共享的，所以我们知道旧线程肯定不会在新线程正在修改向量的时候释放它。

14.2 函数与闭包的类型

在本章中，函数和闭包都在被当作值使用。自然，这就意味着它们有自己的类型。例如：

```
fn city_population_descending(city: &City) -> i64 {  
    -city.population  
}
```

该函数会接受一个参数（&City）并返回 i64。所以它的类型是 `fn(&City) -> i64`。

你可以像对其他值一样对函数执行各种操作。你可以将函数存储在变量中，也可以使用所有常用的 Rust 语法来计算函数值：

```
let my_key_fn: fn(&City) -> i64 =  
    if user.prefs.by_population {  
        city_population_descending  
    } else {  
        city_monster_attack_risk_descending  
    };  
  
cities.sort_by_key(my_key_fn);
```

结构体也可以有函数类型的字段。像 `Vec` 这样的泛型类型可以存储大量的函数，只要它们共享同一个 `fn` 类型即可。而且函数值占用的空间很小，因为 `fn` 值就是函数机器码的内存地址，就像 C++ 中的函数指针一样。

一个函数还可以将另一个函数作为参数：

```
/// 给定一份城市列表和一个测试函数，返回有多少个城市通过了测试  
fn count_selected_cities(cities: &Vec<City>,  
                        test_fn: fn(&City) -> bool) -> usize  
{  
    let mut count = 0;  
    for city in cities {  
        if test_fn(city) {  
            count += 1;  
        }  
    }  
}
```

```

        }
    }
    count
}

/// 测试函数的示例。注意，此函数的类型是`fn(&City) -> bool`，
/// 与`count_selected_cities`的`test_fn`参数相同
fn has_monster_attacks(city: &City) -> bool {
    city.monster_attack_risk > 0.0
}

// 有多少个城市存在被怪兽袭击的风险？
let n = count_selected_cities(&my_cities, has_monster_attacks);

```

如果你熟悉 C/C++ 中的函数指针，就会发现 Rust 的函数值简直跟它一模一样。

知道了这些之后，说闭包与函数不是同一种类型可能会让人大吃一惊：

```

let limit = preferences.acceptable_monster_risk();
let n = count_selected_cities(
    &my_cities,
    |city| city.monster_attack_risk > limit); // 错误：类型不匹配

```

第二个参数会导致类型错误。为了支持闭包，必须更改这个函数的类型签名。要改成下面这样：

```

fn count_selected_cities<F>(cities: &Vec<City>, test_fn: F) ->
usize
where F: Fn(&City) -> bool
{
    let mut count = 0;
    for city in cities {
        if test_fn(city) {
            count += 1;
        }
    }
    count
}

```

这里只更改了 `count_selected_cities` 的类型签名，而没有更改函数体。新版本是泛型函数。只要 `F` 实现了特定的特型 `Fn(&City) -> bool`，该函数就能接受任意 `F` 型的 `test_fn`。以单个 `&City` 为参数并返回 `bool` 值的所有函数和大多数闭包会自动实现这个特型：

```
fn(&City) -> bool    // fn类型（只接受函数）
Fn(&City) -> bool    // Fn特型（既接受函数也接受闭包）
```

这种特殊的语法内置于语言中。`->` 和返回类型是可选的，如果省略，则返回类型为 `()`。

`count_selected_cities` 的新版本会接受函数或闭包：

```
count_selected_cities(
    &my_cities,
    has_monster_attacks); // 正确

count_selected_cities(
    &my_cities,
    |city| city.monster_attack_risk > limit); // 同样正确
```

为什么第一次尝试没有成功？好吧，闭包确实是可调用的，但它不是 `fn`。闭包 `|city| city.monster_attack_risk > limit` 有它自己的类型，但不是 `fn` 类型。

事实上，你编写的每个闭包都有自己的类型，因为闭包可以包含数据：从封闭作用域中借用或“窃取”的值。这既可以是任意数量的变量，也可以是任意类型的组合。所以每个闭包都有一个由编译器创建的特殊类型，大到足以容纳这些数据。任何两个闭包的类型都不相同。但是每个闭包都会实现 `Fn` 特型，我们示例中的闭包就实现了 `Fn(&City) -> i64`。

因为每个闭包都有自己的类型，所以使用闭包的代码通常都应该是泛型的，比如 `count_selected_cities`。每次都明确写出泛型类型确实有点儿笨拙，如果了解这种设计的优点，请继续往下阅读。

14.3 闭包性能

Rust 中闭包的设计目标是要快：比函数指针还要快，快到甚至可以在对性能敏感的热点代码中使用它们。如果你熟悉 C++ 的 `lambda` 表达式，就会发现 Rust 闭包也一样快速而紧凑，但更安全。

在大多数语言中，闭包会在堆中分配内存、进行动态派发以及进行垃圾回收。因此，创建、调用和收集每一个闭包都会花费一点点额外的 CPU 时间。更糟的是，闭包往往难以内联，而内联是编译器用来消除函数调用开销并实施大量其他优化的关键技术。总而言之，闭包在这些语言中确实慢到值得手动将它们从节奏紧凑的内层循环中去掉。

Rust 闭包则没有这些性能缺陷。它们没有垃圾回收。与 Rust 中的其他所有类型一样，除非你将闭包放在 `Box`、`Vec` 或其他容器中，否则它们不会被分配到堆上。由于每个闭包都有不同的类型，因此 Rust 编译器只要知道你正在调用的闭包的类型，就可以内联该闭包的代码。这使得在节奏紧凑的循环中使用闭包成为可能，并且各种 Rust 程序经常会满怀热情地刻意这么做，你会在第 15 章中亲身体会到这一点。

图 14-1 展示了 Rust 闭包在内存中的布局方式。在图的顶部，我们展示了闭包要引用的两个局部变量：字符串 `food` 和简单的枚举 `weather`，枚举的数值恰好是 27。

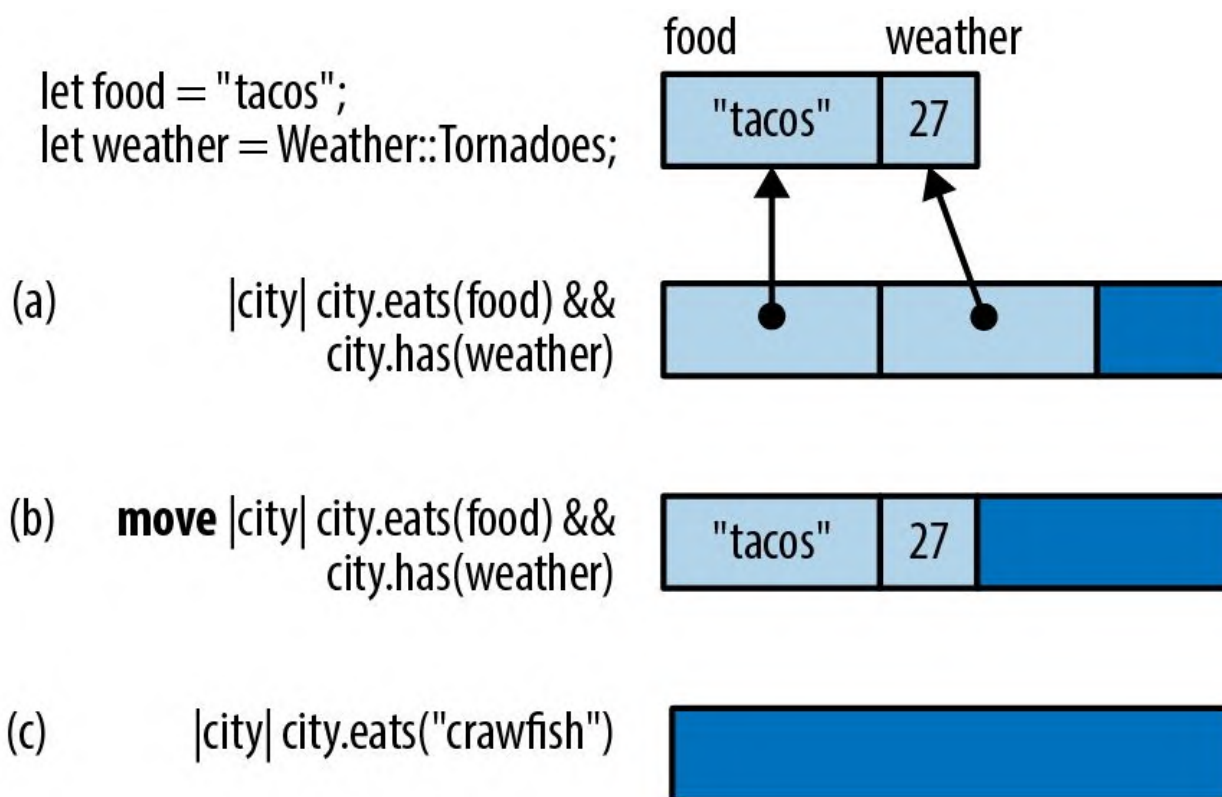


图 14-1：闭包在内存中的布局

闭包 (a) 使用了上述两个变量。显然，我们正在寻找既有炸玉米饼（taco）又有龙卷风（tornado）的城市。在内存中，这个闭包看起来像一个小结构体，其中包含对其所用变量的引用。

请注意，这个闭包并不包含指向其代码的指针。这种指针毫无必要：只要 Rust 知道闭包的类型，就知道在调用此闭包时该运行哪些代码。

闭包 (b) 与闭包 (a) 完全相同，只不过它是一个 `move` 闭包，因此会包含值而非引用。

闭包 (c) 不会使用其环境中的任何变量。该结构体是空的，所以这个闭包根本不会占用任何内存。

如图 14-1 所示，这些闭包不会占用多少空间。但在实践中，即使那区区几字节可能也不是必要的。通常，编译器会内联所有对闭包的调用，然后连图中所示的小结构体也优化掉。

14.5 节会展示如何使用特型对象在堆中分配闭包并动态调用它们。虽然这种方法有点儿慢，但仍然和特型对象的其他方法一样快。

14.4 闭包与安全

到目前为止，我们已经讨论了 Rust 如何确保闭包在从周围代码中借用或移动变量时遵守语言的安全规则。但是还有一些更复杂的后果并不那么显而易见。本节将稍微解释一下当闭包丢弃或修改其捕获的值时会发生什么。

14.4.1 “杀死”闭包

我们已经见过借用值的闭包和“窃取”值的闭包，如果沿着这种思路一直走下去，那么早晚会“出事”²。

² 因为没释放，早晚会“偷光”。——译者注

当然，“杀死”并不是正确的术语。在 Rust 中，我们会说丢弃值。最直观的方法是调用 `drop()`：

```
let my_str = "hello".to_string();
let f = || drop(my_str);
```

当调用 `f` 时，`my_str` 会被丢弃。

那么，如果调用它两次会发生什么呢？

```
f();
f();
```

我们来深入思考一下。当第一次调用 `f` 时，它丢弃了 `my_str`，这意味着存储该字符串的内存已经被释放，交还给了系统。当第二次调用 `f` 时，发生了同样的事情。这是 C++ 编程中会触发未定义行为的经典错误：双重释放。

在 Rust 中丢弃同一个 `String` 两次当然也会出事。幸运的是，Rust 可没那么容易被愚弄：

```
f(); // 正确
f(); // 错误：使用了已移动的值
```

Rust 知道这个闭包不能调用两次。

一个只能调用一次的闭包看起来很不寻常，但这种现象的根源在于本书一直在讲的所有权和生命周期。值会被消耗掉（移动）的想法是 Rust 的核心概念之一，它对闭包与其他语法一视同仁。

14.4.2 FnOnce

我们再试一次欺骗 Rust，让它把同一个 String 丢弃两次。这次将使用下面这个泛型函数：

```
fn call_twice<F>(closure: F) where F: Fn() {
    closure();
    closure();
}
```

可以将这个泛型函数传给任何实现了特型 `Fn()` 的闭包，即不带参数且会返回 `()` 的闭包。（与函数一样，返回类型如果是 `()` 则可以省略，`Fn()` 是 `Fn() -> ()` 的简写形式。）

现在，如果将不安全的闭包传给这个泛型函数会发生什么呢？

```
let my_str = "hello".to_string();
let f = || drop(my_str);
call_twice(f);
```

同样，此闭包将在调用后丢弃 `my_str`。调用它两次将导致双重释放。但 Rust 仍然没有被愚弄：

```
error: expected a closure that implements the `Fn` trait, but
      this closure only implements `FnOnce`
   |
8 | let f = || drop(my_str);
   |           ^^^^^^^^^^^^^^
   |           |           |
   |           |           closure is `FnOnce` because it moves the
```

```
variable `my_str`  
    |           |           out of its environment  
    |           this closure implements `FnOnce`, not `Fn`  
9 | call_twice(f);  
    | ----- the requirement to implement `Fn` derives from  
here
```

这条错误消息为我们揭示了关于 Rust 如何处理“清理型闭包”的更多信息。Rust 本可以在语言中完全禁止这种闭包，但清理闭包有时候是很有用的。因此，Rust 只是限制了它们的使用场景。像 `f` 这种会丢弃值的闭包不允许实现 `Fn`。从字面上看，它们也确实不是 `Fn`。它们实现了一个不那么强大的特型 `FnOnce`，即只能调用一次的闭包特型。

第一次调用 `FnOnce` 闭包时，闭包本身也会被消耗掉。这是因为 `Fn` 和 `FnOnce` 这两个特型是这样定义的：

```
// 无参数的`Fn`特型和`FnOnce`特型的伪代码  
trait Fn() -> R {  
    fn call(&self) -> R;  
}  
  
trait FnOnce() -> R {  
    fn call_once(self) -> R;  
}
```

正如算术表达式 `a + b` 是方法调用 `Add::add(a, b)` 的简写形式一样，Rust 也会将 `closure()` 视为前面示例中的两个特型方法之一的简写形式。对于 `Fn` 闭包，`closure()` 会扩展为 `closure.call()`。此方法会通过引用获取 `self`，因此闭包不会被移动。但是如果闭包只能安全地调用一次，那么 `closure()` 就会扩展为 `closure.call_once()`。该方法会按值获取 `self`，因此这个闭包就会被消耗掉。

当然，这里是故意使用 `drop()` 挑起的麻烦。而在实践中，你通常都会在无意中遇到这种情况。虽然不会经常发生，但偶尔你还是会写出一些无意中消耗掉一个值的闭包代码：

```

let dict = produce_glossary();
let debug_dump_dict = || {
    for (key, value) in dict { // 糟糕!
        println!("{:?} - {:?}", key, value);
    }
};

```

然后，当你多次调用 `debug_dump_dict()` 时，就会收到如下错误消息：

```

error: use of moved value: `debug_dump_dict`
   |
19 |     debug_dump_dict();
   |     ----- `debug_dump_dict` moved due to this call
20 |     debug_dump_dict();
   |     ^^^^^^^^^^^^^^^^^ value used here after move
   |
note: closure cannot be invoked more than once because it moves the
variable
`dict` out of its environment
   |
13 |         for (key, value) in dict {
   |                               ^^^^

```

要调试上述错误，就必须弄清楚此闭包为什么是 `FnOnce`。这里使用了哪个值？编译器友好地指出它是 `dict`，在这种情况下该值是我们唯一引用的值。啊，果然有一个 bug：它通过直接迭代消耗掉了 `dict`。我们应该遍历 `&dict`，而不是普通的 `dict`，以便通过引用访问值：

```

let debug_dump_dict = || {
    for (key, value) in &dict { // 不要消耗掉dict
        println!("{:?} - {:?}", key, value);
    }
};

```

这样就修复了错误，现在这个函数是 `Fn` 并且可以调用任意次数了。

14.4.3 FnMut

还有一种包含可变数据或可变引用的闭包。

Rust 认为不可变值可以安全地跨线程共享，但是包含可变数据的不可变闭包不能安全共享——从多个线程调用这样的闭包可能会导致各种竞态条件，因为多个线程会试图同时读取和写入同一份数据。

Rust 还有另一类名为 `FnMut` 的闭包，也就是可写入的闭包。`FnMut` 闭包会通过可变引用来调用，其定义如下所示：

```
// `Fn` 特型、`FnMut` 特型和 `FnOnce` 特型的伪代码
trait Fn() -> R {
    fn call(&self) -> R;
}

trait FnMut() -> R {
    fn call_mut(&mut self) -> R;
}

trait FnOnce() -> R {
    fn call_once(self) -> R;
}
```

任何需要对值进行可变访问但不会丢弃任何值的闭包都是 `FnMut` 闭包。例如：

```
let mut i = 0;
let incr = || {
    i += 1; // incr借入了对i的一个可变引用
    println!("Ding! i is now: {}", i);
};
call_twice(incr);
```

按照 `call_twice` 的调用方式，它会要求传入一个 `Fn`。由于 `incr` 是 `FnMut` 而非 `Fn`，因此上述代码无法通过编译。不过，有一种简单的解决方法。为了理解此修复，我们先回过头来总结一下 Rust 闭包的 3 种类别。

- `Fn` 是可以不受限制地调用任意多次的闭包和函数系列。此最高类别还包括所有 `fn` 函数。
- `FnMut` 是本身会被声明为 `mut`，并且可以多次调用的闭包系列。

- `FnOnce` 是如果其调用者拥有此闭包，它就只能调用一次的闭包系列。

每个 `Fn` 都能满足 `FnMut` 的要求，每个 `FnMut` 都能满足 `FnOnce` 的要求。如图 14-2 所示，它们不是 3 个彼此独立的类别。

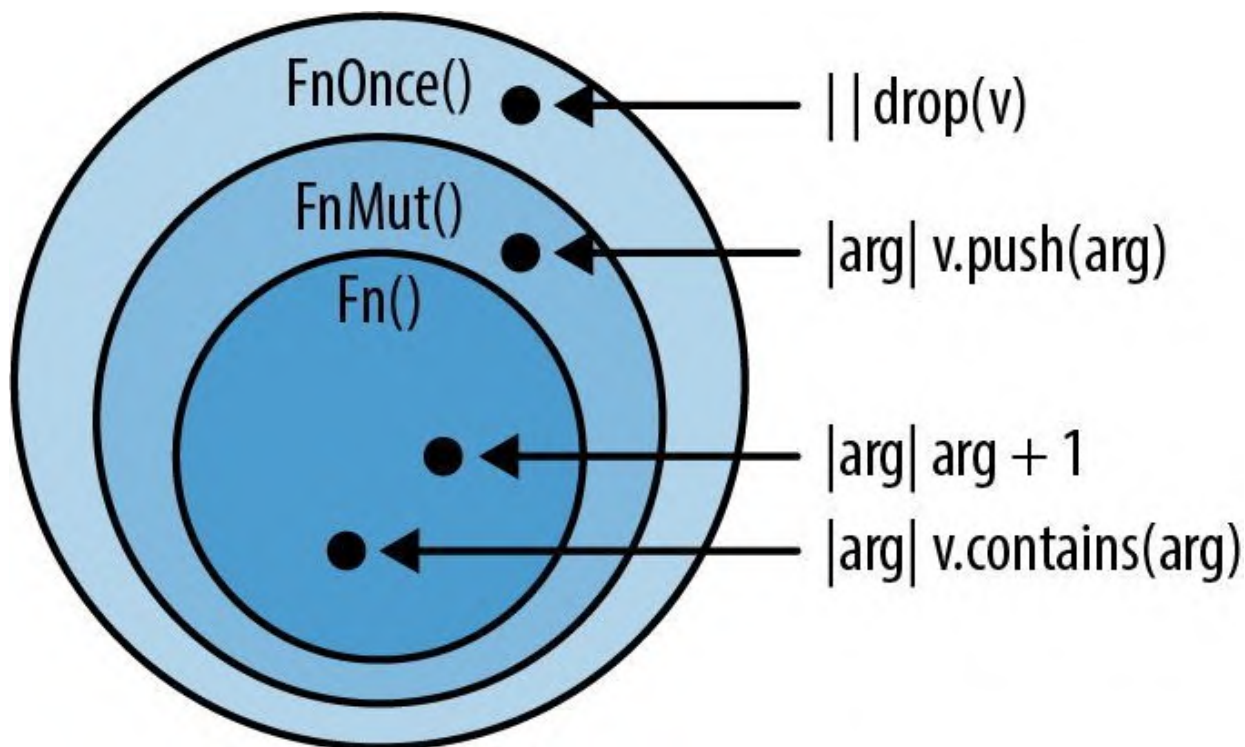


图 14-2: 3 个闭包类别的维恩图

应该说，`Fn()` 是 `FnMut()` 的子特型，而 `FnMut()` 是 `FnOnce()` 的子特型。这使得 `Fn` 成了最严格且最强大的类别。`FnMut` 和 `FnOnce` 是更宽泛的类别，其中包括某些具有使用限制的闭包。

现在我们已经厘清了思路，很显然为了接受尽可能宽泛的闭包，`call_twice` 函数应该接受所有 `FnMut` 闭包，如下所示：

```
fn call_twice<F>(mut closure: F) where F: FnMut() {  
    closure();  
    closure();  
}
```

第 1 行的限界原来是 `F: Fn()`，现在是 `F: FnMut()`。通过此更改，我们仍然能接受所有 `Fn` 闭包，并且还可以在会修改数据的闭包上使用 `call_twice`。

```
let mut i = 0;
call_twice(|| i += 1); // 正确!
assert_eq!(i, 2);
```

14.4.4 对闭包的 **Copy** 与 **Clone**

就像能自动找出哪些闭包只能调用一次一样，Rust 也能找出哪些闭包可以实现 `Copy` 和 `Clone`，哪些则不可以实现。

正如之前所解释的，闭包是表示包含它们捕获的变量的值（对于 `move` 闭包）或对值的引用（对于非 `move` 闭包）的结构体。闭包的 `Copy` 规则和 `Clone` 规则与常规结构体的规则是一样的。一个不修改变量的非 `move` 闭包只持有共享引用，这些引用既能 `Clone` 也能 `Copy`，所以闭包也能 `Clone` 和 `Copy`：

```
let y = 10;
let add_y = |x| x + y;
let copy_of_add_y = add_y; // 此闭包能`Copy`，所以.....
assert_eq!(add_y(copy_of_add_y(22)), 42); // .....可以调用它两次
```

另外，一个会修改值的非 `move` 闭包在其内部表示中也可以有可变引用。可变引用既不能 `Clone`，也不能 `Copy`，使用它们的闭包同样如此：

```
let mut x = 0;
let mut add_to_x = |n| { x += n; x };

let copy_of_add_to_x = add_to_x; // 这会进行移动而非复制
assert_eq!(add_to_x(copy_of_add_to_x(1)), 2); // 错误：使用了已移动出去的值
```

对于 `move` 闭包，规则更简单。如果 `move` 闭包捕获的所有内容都能 `Copy`，那它就能 `Copy`。如果 `move` 闭包捕获的所有内容都能

Clone，那它就能 Clone。例如：

```
let mut greeting = String::from("Hello, ");
let greet = move |name| {
    greeting.push_str(name);
    println!("{}", greeting);
};
greet.clone()("Alfred");
greet.clone()("Bruce");
```

这里的 `.clone()(...)` 语法有点儿奇怪，其实它只是表示克隆此闭包并调用其克隆体。这个程序会输出如下内容：

```
Hello, Alfred
Hello, Bruce
```

当在 `greet` 中使用 `greeting` 时，`greeting` 被移动到了内部表示 `greet` 的结构体中，因为它是一个 `move` 闭包。所以，当我们克隆 `greet` 时，它里面的所有东西同时被克隆了。`greeting` 有两个副本，它们会在调用 `greet` 的克隆时分别被修改。这种行为本身并不是很有用，但是在你需要将同一个闭包传给多个函数的场景中，它会非常有帮助。

14.5 回调

很多库会在其 API 中使用回调函数，即由用户提供某些函数，供库稍后调用。事实上，你已经在本书中看到过一些类似的 API。在第 2 章，我们曾使用 `actix-web` 框架编写过一个简单的 Web 服务器。那个程序的一个重要部分是路由器，它看起来是这样的：

```
App::new()
    .route("/", web::get().to(get_index))
    .route("/gcd", web::post().to(post_gcd))
```

这个路由器的目的是将从互联网传入的请求路由到处理特定类型请求的那部分 Rust 代码中。在本示例中，`get_index` 和 `post_gcd` 是我们在程序其他地方使用 `fn` 关键字声明的函数名称。其实也可以在这里传入闭包，就像这样：

```
App::new()
    .route("/", web::get().to(|| {
        HttpResponse::Ok()
            .content_type("text/html")
            .body("<title>GCD Calculator</title>...")
    }))
    .route("/gcd", web::post().to(|form: web::Form<GcdParameters>|
{
    HttpResponse::Ok()
        .content_type("text/html")
        .body(format!("The GCD of {} and {} is {}.",
            form.n, form.m, gcd(form.n, form.m)))
}))
```

这是因为 `actix-web` 设计成了可以接受任何线程安全的 `Fn` 作为参数的形式。

那么，如何在自己的程序中做到这一点呢？可以试着从头开始编写自己的简易路由器，而不使用来自 `actix-web` 的任何代码。可以首先声明一些类型来表示 HTTP 请求和响应：

```

struct Request {
    method: String,
    url: String,
    headers: HashMap<String, String>,
    body: Vec<u8>
}

struct Response {
    code: u32,
    headers: HashMap<String, String>,
    body: Vec<u8>
}

```

现在路由器所做的只是存储一个将 URL 映射到回调的表，以便按需调用正确的回调。（为简单起见，只允许用户创建与单个 URL 精确匹配的路由。）

```

struct BasicRouter<C> where C: Fn(&Request) -> Response {
    routes: HashMap<String, C>
}

impl<C> BasicRouter<C> where C: Fn(&Request) -> Response {
    /// 创建一个空路由器
    fn new() -> BasicRouter<C> {
        BasicRouter { routes: HashMap::new() }
    }

    /// 给路由器添加一个路由
    fn add_route(&mut self, url: &str, callback: C) {
        self.routes.insert(url.to_string(), callback);
    }
}

```

遗憾的是，我们犯了一个错误。你注意到了吗？

如果只给路由器添加一个路由，那么它是可以正常工作的：

```

let mut router = BasicRouter::new();
router.add_route("/", |_| get_form_response());

```

这段代码可以编译和运行。不过很遗憾，如果再添加一个路由：

```

router.add_route("/gcd", |req| get_gcd_response(req));

```

就会得到一些错误：

```
error: mismatched types
  |
41 | router.add_route("/gcd", |req| get_gcd_response(req));
  |                               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |                               expected closure, found a different
closure
  |
    = note: expected type `[closure@closures_bad_router.rs:40:27:
40:50]`
           found type `[closure@closures_bad_router.rs:41:30:
41:57]`
note: no two closures, even if identical, have the same type
help: consider boxing your closure and/or using it as a trait
object
```

我们所犯的错误在于如何定义 `BasicRouter` 类型：

```
struct BasicRouter<C> where C: Fn(&Request) -> Response {
    routes: HashMap<String, C>
}
```

这里声明的每个 `BasicRouter` 都带有一个回调类型 `C`，并且 `HashMap` 中的所有回调都是此类型的。11.1.3 节曾展示过一个具有相同问题的 `Salad` 类型：

```
struct Salad<V: Vegetable> {
    veggies: Vec<V>
}
```

这里的解决方案与 `Salad` 的解决方案一样：因为要支持多种类型，所以需要使用 `Box` 和特型对象：

```
type BoxedCallback = Box<dyn Fn(&Request) -> Response>;

struct BasicRouter {
    routes: HashMap<String, BoxedCallback>
}
```

每个 Box 可以包含不同类型的闭包，因此单个 HashMap 可以包含各种回调。请注意，类型参数 C 消失了。

这需要对此方法进行一些调整。

```
impl BasicRouter {
    // 创建一个空路由器
    fn new() -> BasicRouter {
        BasicRouter { routes: HashMap::new() }
    }
    // 给路由器添加一个路由
    fn add_route<C>(&mut self, url: &str, callback: C)
        where C: Fn(&Request) -> Response + 'static
    {
        self.routes.insert(url.to_string(), Box::new(callback));
    }
}
```



请注意 add_route 的类型签名中 C 的两个限界：特定的 Fn 特型和 'static 生命周期。Rust 要求我们添加这个 'static 限界。如果没有它，那么对 Box::new(callback) 的调用就会出错，因为如果闭包包含对即将超出作用域的变量的已借用引用，那么存储闭包就是不安全的。

最后，我们的简单路由器已准备好处理传入请求了：

```
impl BasicRouter {
    fn handle_request(&self, request: &Request) -> Response {
        match self.routes.get(&request.url) {
            None => not_found_response(),
            Some(callback) => callback(request)
        }
    }
}
```

以牺牲一些灵活性为代价，我们还可以写出此路由器的更省空间的版本：它并不存储特型对象，而是使用函数指针或 fn 类型。这些类型（如 fn(u32) -> u32）的行为很像闭包：

```
fn add_ten(x: u32) -> u32 {
    x + 10
}
let fn_ptr: fn(u32) -> u32 = add_ten;
let eleven = fn_ptr(1); // 11
```

事实上，不从其环境中捕获任何内容的闭包与函数指针是一样的，因为它们不需要保存有关捕获变量的任何额外信息。如果在绑定或函数签名中指定了适当的 `fn` 类型，则编译器很乐意让你以这种方式使用它们：

```
let closure_ptr: fn(u32) -> u32 = |x| x + 1;
let two = closure_ptr(1); // 2
```

与捕获型闭包不同，这类函数指针只会占用一个 `usize`。

持有函数指针的路由表如下所示：

```
struct FnPointerRouter {
    routes: HashMap<String, fn(&Request) -> Response>
}
```

在这里，`HashMap` 只会为每个 `String` 键存储一个 `usize` 值，更关键的是，没有 `Box`。除了 `HashMap` 自身，根本不存在动态分配。当然，方法也要相应调整：

```
impl FnPointerRouter {
    // 创建一个空路由器
    fn new() -> FnPointerRouter {
        FnPointerRouter { routes: HashMap::new() }
    }

    // 给路由器添加一个路由
    fn add_route(&mut self, url: &str, callback: fn(&Request) ->
Response)
    {
        self.routes.insert(url.to_string(), callback);
    }
}
```


如图 14-1 所示，闭包具有独特的类型，因为每个闭包会捕获不同的变量，所以和别的语法元素一样，它们各自具有不同的大小。但是，如果闭包没有捕捉到任何东西，那就没有什么要存储的了。通过在接受回调的函数中使用 `fn` 指针，可以限制调用者仅使用这些非捕获型闭包，以牺牲调用者的灵活性为代价，在接受回调的代码中换取一定的性能和灵活性。

14.6 高效地使用闭包

正如我们所见，Rust 的闭包不同于大多数其他语言中的闭包。最大的区别是，在具有垃圾回收的语言中，你可以在闭包中使用局部变量，而无须考虑生命周期或所有权的问题。但如果没有垃圾回收，那么情况就不同了。一些在 Java、C# 和 JavaScript 中常见的设计模式如果不进行改变将无法在 Rust 中正常工作。

如图 14-3 所示，我们以模型-视图-控制器设计模式（简称 MVC）为例。对于用户界面的每个元素，MVC 框架都会创建 3 个对象：表示该 UI 元素状态的**模型**、负责其外观的**视图**和处理用户交互的**控制器**。多年来，MVC 模式已经出现了无数变体，但总体思路仍是 3 个对象以某种方式分担了 UI 的职责。

这就是问题所在。通常，每个对象都会直接或通过回调对其他对象中的一个或两个进行引用，如图 14-3 所示。每当 3 个对象中的一个对象发生变化时，它会通知其他两个对象，因此所有内容都会及时更新。哪个对象“拥有”其他对象之类的问题永远不会出现。

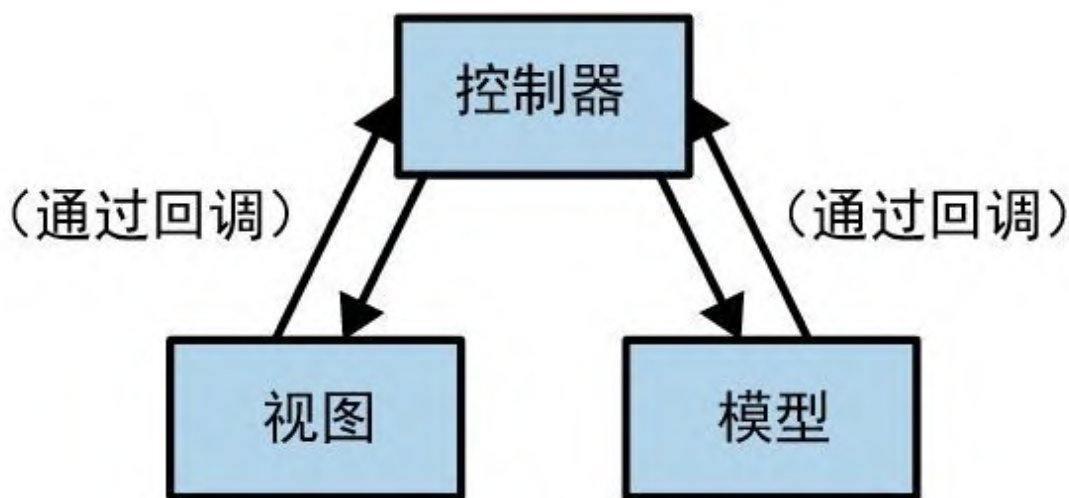


图 14-3：模型-视图-控制器设计模式

如果不进行更改，就无法在 Rust 中实现此模式。所有权必须明晰，循环引用也必须消除。模型和控制器不能相互直接引用。

Rust 的“激进赌注”是基于“必然存在好的替代设计”这个假设的。有时你可以通过让每个闭包接受它需要的引用作为参数，来解决闭包所有权和生命周期的问题。有时你可以为系统中的每个事物分配一个编号，并传递这些编号而不是传递引用。或者你可以实现 MVC 的众多变体之一，其中的对象并非都相互引用。或者你可以将工具包建模为具有单向数据流的非 MVC 系统，比如 Facebook 的 Flux 架构，如图 14-4 所示。

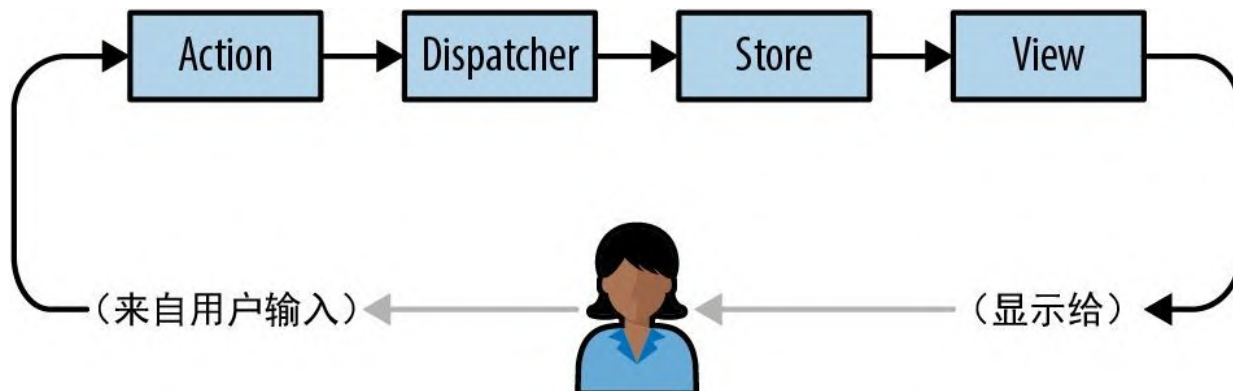


图 14-4：MVC 的替代方案——Flux 架构

简而言之，如果你试图使用 Rust 闭包来应对复杂对象关系，就会遇到困难，但还有其他选择。在这种情况下，软件工程这门学科似乎更倾向于使用替代方案，因为它们更简单。

第 15 章将讨论真正让闭包大放异彩的话题。我们将编写一种代码，以充分利用 Rust 闭包的简洁性、速度和效率。这些代码编写起来很有趣，易于阅读，而且非常实用。第 15 章将介绍 Rust 迭代器。

第 15 章 迭代器

漫长的一天结束了。

——Phil

迭代器是一个值，它可以生成一系列值，通常用来执行循环操作。Rust 的标准库不仅提供了用于遍历向量、字符串、哈希表和其他集合的迭代器，还提供了“从输入流中产生文本行”“从网络服务器中产生新的入站连接”“从通信通道中其他线程接收的值”等迭代器。当然，你也可以出于自己的目的实现迭代器。Rust 的 `for` 循环为使用迭代器提供了一种自然的语法，但迭代器本身也提供了一组丰富的方法，比如映射（`map`）、过滤（`filter`）、连接（`join`）、收集（`collect`）等。

Rust 的各种迭代器灵活、富有表现力而且高效。考虑以下函数，它会返回前 `n` 个正整数之和（通常称为第 `n` 个三角形数）：

```
fn triangle(n: i32) -> i32 {
    let mut sum = 0;
    for i in 1..=n {
        sum += i;
    }
    sum
}
```

表达式 `1..=n` 是一个 `RangeInclusive<i32>` 型的值。而 `RangeInclusive<i32>` 是一个迭代器，可以生成其起始值到结束值（包括两者）之间的整数，因此你可以将它用作 `for` 循环的操作数来对从 1 到 `n` 的值求和。

但是迭代器还有一个 `fold` 方法，可以实现完全一样的效果：

```
fn triangle(n: i32) -> i32 {
    (1..=n).fold(0, |sum, item| sum + item)
}
```

开始运行时以 0 作为总和，fold 会获取 `1..=n` 生成的每个值，并以总和（sum）跟当前值（item）为参数调用闭包 `|sum, item| sum + item`。闭包的返回值会作为新的总和。它返回的最后一个值就是 fold 自身要返回的值——在这个例子中，也就是整个序列的总和。如果你用惯了 for 循环和 while 循环，这种写法可能看起来很奇怪，但一旦习惯了 fold，你就会发现 fold 的表达方式更加清晰和简洁。

这就是函数式编程语言的标准风格，非常注重表达能力。Rust 的迭代器都经过了精心设计，以确保编译器可以把它们翻译成优秀的机器码。例如前面展示的第二个定义，在发行版中，Rust 会理解 fold 的定义并将其内联到 triangle 中。接下来是将闭包 `|sum, item| sum + item` 内联到 triangle 中。最后，Rust 会检查合并后的代码并意识到有一种更简单的方法可以对从 1 到 n 的数值求和：其总和总会等于 $n * (n+1) / 2$ 。于是 Rust 将 triangle 的整个函数体，包括循环、闭包和所有内容，翻译成了单个乘法指令和几个算术运算。

虽然这个例子只涉及简单的算术运算，但迭代器在重度使用时也同样表现出色。它们是 Rust 提供的另一种灵活抽象，在典型应用中几乎不会产生额外开销。

在本章中，我们将解释以下核心知识点。

- Iterator 特型和 IntoIterator 特型，两者是 Rust 迭代器的基础。
- 一个典型的迭代器流水线通常有 3 个阶段：从某种“值源”创建迭代器，通过选择或处理从中流过的值来将一种迭代器适配成另一种迭代器，然后消耗此迭代器生成的值。
- 如何为自己的类型实现迭代器。

迭代器的方法非常多，不过只要你掌握了其基本思想，就可以粗略浏览相应的部分。但迭代器在 Rust 惯用法中非常常见，熟悉它们附带的工具对于掌握这门语言至关重要。

15.1 Iterator 特型与 IntoIterator 特型

迭代器是实现了 `std::iter::Iterator` 特型的任意值：

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
    ..... // 很多默认方法
}
```

`Item` 是迭代器所生成的值的类型。`next` 方法要么返回 `Some(v)`（其中 `v` 是迭代器的下一个值），要么返回 `None`（作为序列结束的标志）。这里我们省略了 `Iterator` 的许多默认方法，本章的其余小节会分别介绍它们。

只要可以用某种自然的方式来迭代某种类型，该类型就可以实现 `std::iter::IntoIterator`，其 `into_iter` 方法会接受一个值并返回一个迭代器：

```
trait IntoIterator where Self::IntoIter: Iterator<Item=Self::Item>
{
    type Item;
    type IntoIter: Iterator;
    fn into_iter(self) -> Self::IntoIter;
}
```

`IntoIter` 是迭代器本身的类型，而 `Item` 是它生成的值的类型。任何实现了 `IntoIterator` 的类型都称为可迭代者，因为你可以随意迭代它。

Rust 的 `for` 循环会将所有这些部分很好地结合在一起。要遍历向量的元素，你可以这样写：

```
println!("There's:");
let v = vec!["antimony", "arsenic", "aluminum", "selenium"];

for element in &v {
```

```
println!("{}", element);  
}
```

在幕后，每个 `for` 循环都只是调用 `IntoIterator` 和 `Iterator` 中某些方法的简写形式：

```
let mut iterator = (&v).into_iter();  
while let Some(element) = iterator.next() {  
    println!("{}", element);  
}
```

`for` 循环会使用 `IntoIterator::into_iter` 将其操作数 `&v` 转换为迭代器，然后重复调用 `Iterator::next`。每次返回 `Some(element)` 时，`for` 循环都会执行其循环体，如果返回 `None`，则循环结束。

先记住这个例子，下面介绍迭代器的一些术语。

- 正如我们所说，**迭代器**是实现了 `Iterator` 的任意类型。
- **可迭代者**是任何实现了 `IntoIterator` 的类型：你可以通过调用它的 `into_iter` 方法来获得一个迭代器。在这里，向量引用 `&v` 就是可迭代者。
- 迭代器能**生成值**。
- 迭代器生成的值是**条目**。在这里，条目是 `"antimony"`、`"arsenic"` 等。
- 接收迭代器所生成条目的代码是**消费者**。在这里，`for` 循环体就是消费者。

虽然 `for` 循环总会在其操作数上调用 `into_iter`，但你也可以直接把迭代器传给 `for` 循环，比如，在遍历 `Range` 时就是这样的。所有迭代器都自动实现了 `IntoIterator`，并带有一个直接返回迭代器的 `into_iter` 方法。

如果在返回 `None` 后再次调用迭代器的 `next` 方法，则 `Iterator` 特型没有规定它应该做什么。大多数迭代器只会再次返回 `None`，但

也有例外。（如果这个过程中出了问题，可以参考一下 15.3.7 节中介绍的 fuse 适配器。）

15.2 创建迭代器

虽然 Rust 标准库文档对每个种类的迭代器类型都进行了详细解释，但标准库还是遵循了一些通用的约定，来帮助你定位并找到想要的东西。

15.2.1 `iter` 方法与 `iter_mut` 方法

大多数集合类型提供了 `iter`（迭代器）方法和 `iter_mut`（可变动迭代器）方法，它们会返回该类型的自然迭代器，为每个条目生成共享引用或可变动引用。像 `&[T]` 和 `&mut [T]` 这样的数组切片也有 `iter` 方法和 `iter_mut` 方法。如果你不打算让 `for` 循环替你跟迭代器打交道，`iter` 方法和 `iter_mut` 方法就是获取迭代器最常用的方法：

```
let v = vec![4, 20, 12, 8, 6];
let mut iterator = v.iter();
assert_eq!(iterator.next(), Some(&4));
assert_eq!(iterator.next(), Some(&20));
assert_eq!(iterator.next(), Some(&12));
assert_eq!(iterator.next(), Some(&8));
assert_eq!(iterator.next(), Some(&6));
assert_eq!(iterator.next(), None);
```

这个迭代器的条目类型是 `&i32`：每次调用 `next` 都会生成对下一个元素的引用，直到抵达向量的末尾。

每种类型都可以自由选用任何符合其设计意图的方式实现 `iter` 和 `iter_mut`。`std::path::Path` 上的 `iter` 方法会返回一个迭代器，一次生成一个路径组件：

```
use std::ffi::OsStr;
use std::path::Path;

let path = Path::new("C:/Users/JimB/Downloads/Fedora.iso");
let mut iterator = path.iter();
assert_eq!(iterator.next(), Some(OsStr::new("C:")));
```

```
assert_eq!(iterator.next(), Some(OsStr::new("Users")));
assert_eq!(iterator.next(), Some(OsStr::new("JimB")));
...
```

这个迭代器的条目类型是 `&std::ffi::OsStr`，是从操作系统调用所要求的那类字符串借用的切片类型。

如果类型有不只一种常用的遍历方式，该类型通常会为每种遍历方式提供一个专门的方法，因为普通的 `iter` 方法会产生歧义。例如，`&str` 字符串切片类型就没有 `iter` 方法——如果 `s` 是 `&str`，则 `s.bytes()` 会返回一个能生成 `s` 中每字节的迭代器，而 `s.chars()` 则会将内容解释为 UTF-8 并生成每个 Unicode 字符。

15.2.2 IntoIterator 的实现

如果一个类型实现了 `IntoIterator`，你也可以自行调用它的 `into_iter` 方法，就像 `for` 循环一样：

```
// 大家通常会使用HashSet，但它的迭代顺序是不确定的，
// 因此在这个示例中用了BTreeSet，它的演示效果更好些
use std::collections::BTreeSet;
let mut favorites = BTreeSet::new();
favorites.insert("Lucy in the Sky With Diamonds".to_string());
favorites.insert("Liebesträume No. 3".to_string());
let mut it = favorites.into_iter();
assert_eq!(it.next(), Some("Liebesträume No. 3".to_string()));
assert_eq!(it.next(), Some("Lucy in the Sky With
Diamonds".to_string()));
assert_eq!(it.next(), None);
```

大多数集合实际上提供了 `IntoIterator` 的几种实现，用于共享引用（`&T`）、可变引用（`&mut T`）和移动（`T`）。

- 给定一个集合的共享引用，`into_iter` 会返回一个迭代器，该迭代器会生成对其条目的共享引用。例如，在前面的代码中，`(&favorites).into_iter()` 会返回一个 `Item` 类型为 `&String` 的迭代器。

- 给定对集合的可变引用，`into_iter` 会返回一个迭代器，该迭代器会生成对其条目的可变引用。如果 `vector` 是某个 `Vec<String>`，则调用 `(&mut vector).into_iter()` 会返回一个 `Item` 类型为 `&mut String` 的迭代器。
- 当按值传递集合时，`into_iter` 会返回一个迭代器，该迭代器会获取集合的所有权并按值返回这些条目，这些条目的所有权会从集合转移给消费者，原始集合在此过程中已被消耗掉了。例如，前面代码中的 `favorites.into_iter()` 调用返回了一个迭代器，该迭代器会按值生成每个字符串，消费者会获得每个字符串的所有权。当迭代器被丢弃时，`BTreeSet` 中剩余的所有元素都将被丢弃，并且该集合的空壳也将被丢弃。

由于 `for` 循环会将 `IntoIterator::into_iter` 作为它的操作对象，因此这 3 种实现创建了以下惯用法，用于迭代对集合的共享引用或可变引用，或者消耗该集合并获取其元素的所有权：

```
for element in &collection { ... }  
for element in &mut collection { ... }  
for element in collection { ... }
```

其中每种用法最终都会调用此处列出的三种 `IntoIterator` 实现之一。

并非每种类型都提供了这 3 种实现，比如，`HashSet`、`BTreeSet` 和 `BinaryHeap` 不会在可变引用上实现 `IntoIterator`，因为修改它们的元素可能会违反类型自身的不变性规则——修改后的值很可能有不同的哈希值，或者相对于其邻居的顺序改变了，所以修改它会让该类型处于错误状态。另一部分类型确实支持修改，但只支持修改一部分，比如，`HashMap` 和 `BTreeMap` 会生成对其条目值的可变引用，但只能提供对其键的共享引用，原因与前面给出的相似。

总体原则是，迭代应该是高效且可预测的，因此 `Rust` 不会提供昂贵或可能表现出意外行为的实现。（例如，对修改过的 `HashSet` 条目重新进行哈希，可能会导致在迭代中稍后再次遇到这些条目。）

切片实现了 3 个 `IntoIterator` 变体中的两个，由于切片并不拥有自己的元素，因此不存在“按值”引用的情况。`&[T]` 和 `&mut [T]` 各自的 `into_iter` 会分别返回一个迭代器，该迭代器会生成对其元素的共享引用和可变引用。如果你将底层的切片类型 `[T]` 想象成某种集合，那它就完美地遵循了集合的总体模式。¹

¹Rust 的总体模式是指，对于任何集合，都会有一种标准的方式可以迭代它的元素。但这种方式可能因具体集合类型而异。——译者注

你可能已经注意到了，对于共享引用和可变引用，前两个 `IntoIterator` 变体等效于在引用目标上调用 `iter` 或 `iter_mut`。为什么 Rust 要同时提供 `into_iter` 和 `iter` 这两种方式呢？

`IntoIterator` 是确保 `for` 循环工作的关键，显而易见它是必要的。但当我们不用 `for` 循环时，写 `favorites.iter()` 会比 `(&favorites).into_iter()` 更清晰。我们会频繁通过共享引用进行迭代，因此 `iter` 和 `iter_mut` 仍然具有很高的工效学价值。

`IntoIterator` 在泛型代码中也很有用：你可以使用像 `T: IntoIterator` 这样的限界来将类型变量 `T` 限制为可以迭代的类型，还可以编写 `T: IntoIterator<Item=U>` 来进一步要求迭代时生成具有特定类型 `U` 的条目。例如，`dump` 函数可以转储任何其条目可用 `"{:?}"` 格式打印的可迭代者的值：

```
use std::fmt::Debug;

fn dump<T, U>(t: T)
    where T: IntoIterator<Item=U>,
          U: Debug
{
    for u in t {
        println!("{:?}", u);
    }
}
```

但你不能使用 `iter` 和 `iter_mut` 来编写这个泛型函数，因为它们不是任何特型的方法：只是大多数可迭代类型恰好具有叫这两个名字的方法而已。

15.2.3 `from_fn` 与 `successors`

要生成一系列值，有一种简单而通用的方法，那就是提供一个能返回这些值的闭包。

给定返回 `Option<T>` 的函数，`std::iter::from_fn`（来自 `fn`）就会返回一个迭代器，该迭代器会调用 `fn` 来生成条目。例如：

```
use rand::random; // 在Cargo.toml中添加dependencies: rand = "0.7"
use std::iter::from_fn;

// 产生1000条端点均匀分布在区间[0, 1]上的随机线段的长度（这并不是
// `rand_distr` crate中能找到的分布类型，但你可以轻易实现一个）
let lengths: Vec<f64> =
    from_fn(|| Some((random::<f64>() - random::<f64>()).abs()))
        .take(1000)
        .collect();
```

它会调用 `from_fn` 来让迭代器产生随机数。由于迭代器总是返回 `Some`，因此序列永不结束，但我们调用 `take(1000)` 时会将其限制为前 1000 个元素。然后 `collect` 会从这 1000 次迭代中构建出向量。这是构造已初始化向量的有效方式，我们会在 15.4.13 节中解释原因。

如果每个条目都依赖于其前一个条目，那么 `std::iter::successors` 函数很实用。只需要提供一个初始条目和一个函数，且该函数能接受一个条目并返回下一个条目的 `Option`。如果返回 `None`，则迭代结束。例如，下面是编写第 2 章中的曼德博集绘图器的 `escape_time` 函数的另一种方式：

```
use num::Complex;
use std::iter::successors;

fn escape_time(c: Complex<f64>, limit: usize) -> Option<usize> {
```

```

let zero = Complex { re: 0.0, im: 0.0 };
successors(Some(zero), |&z| { Some(z * z + c) })
    .take(limit)
    .enumerate()
    .find(|(_i, z)| z.norm_sqr() > 4.0)
    .map(|(i, _z)| i)
}

```

从零开始，`successors`（后继者）调用会通过反复对最后一个点求平方再加上参数 `c` 来生成复平面上的一系列点。在绘制曼德博集时，我们想看看这个序列是永远在原点附近打转还是“飞向”无穷远。调用 `take(limit)` 确定了我们追踪序列的次数限制，然后 `enumerate` 对每个点进行编号，将每个点 `z` 变成元组 `(i, z)`。我们使用 `find` 来寻找距离原点足够远的第一个点以判断是否逃逸。`find` 方法会返回一个 `Option`：如果这样的点存在就返回 `Some((i, z))`，否则返回 `None`。调用 `Option::map` 会将 `Some((i, z))` 变成 `Some(i)`，但不会改变 `None`，因为这正是我们想要的返回值。

`from_fn` 和 `successors` 都接受 `FnMut` 闭包，因此你的闭包可以捕获和修改周边作用域中的变量。例如，下面的 `fibonacci` 函数就用 `move` 闭包来捕获一个变量并将其用作自己的运行状态：

```

fn fibonacci() -> impl Iterator<Item=usize> {
    let mut state = (0, 1);
    std::iter::from_fn(move || {
        state = (state.1, state.0 + state.1);
        Some(state.0)
    })
}

assert_eq!(fibonacci().take(8).collect::<Vec<_>>(),
    vec![1, 1, 2, 3, 5, 8, 13, 21]);

```

需要注意的是，`from_fn` 方法和 `successors` 方法非常灵活，你几乎可以将任何对迭代器的使用改写成对其中之一的调用，通过传递复杂的闭包来得到你想要的行为。但这样做浪费了迭代器提供的机会，即使用常见模式的标准名称来更清晰地表达计算中的数据流。在

使用这两个方法之前，请确保你已经熟悉本章中的其他迭代器方法，通常其他迭代器是更好的选择。²

²简而言之，这两个方法应该作为我们最后的选项，而不是一上来就使用。——译者注

15.2.4 `drain` 方法

有许多集合类型提供了 `drain`（抽取）方法。`drain` 会接受一个对集合的可变引用，并返回一个迭代器，该迭代器会将每个元素的所有权传给消费者。然而，与按值获取并消耗掉集合的 `into_iter()` 方法不同，`drain` 只会借入对集合的可变引用，当迭代器被丢弃时，它会从集合中移除所有剩余元素以清空集合。

对于可以按范围索引的类型（如 `String`、向量和 `VecDeque`），`drain` 方法可指定要移除的元素范围，而不是“抽干”整个序列：

```
let mut outer = "Earth".to_string();
let inner = String::from_iter(outer.drain(1..4));

assert_eq!(outer, "Eh");
assert_eq!(inner, "art");
```

如果确实需要“抽干”整个序列，使用整个范围（`..`）作为参数即可。

15.2.5 其他迭代器源

前面的内容主要关注集合类型（如向量和 `HashMap`），但标准库中还有许多其他类型也支持迭代。表 15-1 总结了一些比较有趣的迭代器，限于篇幅，还有更多没有列出来。我们在专门讲解特定类型的章（第 16 章、第 17 章和第 18 章）中会更详细地介绍其中的一些方法。

表 15-1：标准库中的其他迭代器

类型或特型	表达式	注意事项
<code>std::ops::Range</code>	<code>1..10</code> <code>(1..10).step_by(2)</code>	两个端点必须是可迭代的整数类型。范围包括起始值，不包括结束值 生成 1、3、5、7、9
<code>std::ops::RangeFrom</code>	<code>1..</code>	无界迭代。起点必须是一个整数。如果值达到了该类型的上限，可能会发生 panic 或溢出
<code>std::ops::RangeInclusive</code>	<code>1..=10</code>	和 Range 类似，但包含结束值
<code>Option<T></code>	<code>Some(10).iter()</code>	表现得像一个长度为 0 (None) 或 1 (Some(v)) 的向量
<code>Result<T, E></code>	<code>Ok("blah").iter()</code>	类似于 Option，但生成 Ok 值

类型或特型	表达式	注意事项
<code>Vec<T>, &[T]</code>	<code>v.windows(16)</code> <code>v.chunks(16)</code> <code>v.chunks_mut(1024)</code> <code>v.split(byte byte & 1 != 0)</code> <code>v.split_mut(...)</code> <code>v.rsplit(...)</code> <code>v.splitn(n, ...)</code>	从左到右生成给定长度的所有连续切片。窗口之间会有重叠 从左到右生成给定长度的不重叠的连续切片 和 <code>chunks</code> 类似，但生成的切片是可变的 生成由匹配给定谓词的元素分隔的切片 同上，但生成可变切片 与 <code>split</code> 类似，但从右到左生成切片 与 <code>split</code> 类似，但最多生成 n 个切片

类型或特型	表达式	注意事项
String, &str	<pre> s.bytes() s.chars() s.split_whitespace() s.lines() s.split('/') s.matches(char::is_numeric) </pre>	生成一些 UTF-8 格式的字节 生成一些 UTF-8 表示的字符 按空白字符拆分字符串，并生成非空白字符的切片 生成字符串各行的切片 用给定模式拆分字符串，用匹配项之间的部分生成切片。模式可以有很多种：字符、String 和闭包 生成与给定模式匹配的切片
std::collections::HashMap, std::collections::BTreeMap	<pre> map.keys(), map.values() map.values_mut() </pre>	生成对该 map 的键或值的共享引用 生成对条目值的可变引用
std::collections::HashSet, std::collections::BTreeSet	<pre> set1.union(set2) set1.intersection(set2) </pre>	生成对 set1 和 set2 并集元素的共享引用 生成对 set1 和 set2 交集元素的共享引用

类型或特型	表达式	注意事项
<code>std::sync::mpsc::Receiver</code>	<code>recv.iter()</code>	生成从位于另一个线程的对端发送者发来的值
<code>std::io::Read</code>	<code>stream.bytes()</code> <code>stream.chars()</code>	从 I/O 流中生成一些字节 将流解析为 UTF-8，并生成一些字符
<code>std::io::BufRead</code>	<code>bufstream.lines()</code> <code>bufstream.split(0)</code>	将流解析为 UTF-8，并按行生成一些 String 使用给定的字节拆分流，生成该字节间的 Vec<u8> 缓冲区
<code>std::fs::ReadDir</code>	<code>std::fs::read_dir(path)</code>	生成目录条目
<code>std::net::TcpListener</code>	<code>listener.incoming()</code>	生成传入的网络连接
自由函数	<code>std::iter::empty()</code> <code>std::iter::once(5)</code> <code>std::iter::repeat("#9")</code>	立即返回 None 生成给定的值然后结束 总是生成给定的值

15.3 迭代器适配器

一旦你手头有了迭代器，迭代器的 `Iterator` 特型就会提供大量适配器方法（也可以简称为适配器）。适配器会消耗某个迭代器并构建一个实现了特定行为的新迭代器。为了阐明适配器的工作原理，我们将从两个最流行的适配器 `map` 和 `filter` 开始，然后介绍其他适配器，涵盖了你能想到的从其他序列生成值序列的几乎所有方式：截断、跳过、组合、反转、连接、重复等。

15.3.1 `map` 与 `filter`

`Iterator` 特型的 `map`（映射）适配器能针对迭代器的各个条目调用闭包来帮你转换迭代器。`filter` 适配器能使用闭包来帮你从迭代器中过滤某些条目，由闭包决定保留和丢弃哪些条目。

假设你正在逐行遍历文本并希望去掉每一行的前导空格和尾随空格。标准库的 `str::trim` 方法能从单个 `&str` 中丢弃前导空格和尾随空格，返回一个新的、修剪过的 `&str` 借用。你可以通过 `map` 适配器将 `str::trim` 应用于迭代器中的每一行：

```
let text = " ponies \n giraffes\niguanas\nsquid".to_string();
let v: Vec<&str> = text.lines()
    .map(str::trim)
    .collect();
assert_eq!(v, ["ponies", "giraffes", "iguanas", "squid"]);
```

`text.lines()` 调用会返回一个生成字符串中各行的迭代器。在该迭代器上调用 `map` 会返回第二个迭代器，第二个迭代器会对每一行调用 `str::trim` 并将生成的结果作为自己的条目。最后，`collect` 会将这些条目收集到一个向量中。

`map` 返回的迭代器本身当然也可以进一步适配。如果你想将结果中的 `iguanas` 排除，可以这样写：

```
let text = "  ponies  \n  giraffes\niguanas\nsquid".to_string();
let v: Vec<&str> = text.lines()
    .map(str::trim)
    .filter(|s| *s != "iguanas")
    .collect();
assert_eq!(v, ["ponies", "giraffes", "squid"]);
```

在这里，`filter` 会返回第三个迭代器，它只会从 `map` 迭代器的结果中生成闭包 `|s| *s != "iguanas"` 返回 `true` 的那些条目。迭代器的适配器链条就像 Unix shell 中的管道：每个适配器都有单一用途，并且很清楚此序列是如何在从左到右读取时进行转换的。

`map` 和 `filter` 的适配器的签名如下所示：

```
fn map<B, F>(self, f: F) -> impl Iterator<Item=B>
    where Self: Sized, F: FnMut(Self::Item) -> B;

fn filter<P>(self, predicate: P) -> impl Iterator<Item=Self::Item>
    where Self: Sized, P: FnMut(&Self::Item) -> bool;
```

在标准库中，`map` 和 `filter` 实际上返回的是名为 `std::iter::Map` 和 `std::iter::Filter` 的专用不透明（隐藏了实现细节的）`struct` 类型。然而，仅仅看到这些名字并不能告诉我们更多信息，所以在本书中，我们会写成 `-> impl Iterator<Item=...>`，因为这揭示了我们真正关心的事情：此方法返回了能生成给定类型条目的 `Iterator`。

大多数适配器会按值接受 `self`，这就要求 `Self` 必须是固定大小的（`Sized`）（所有常见的迭代器都是这样的）。

`map` 迭代器会按值将每个条目传给闭包，然后将闭包结果的所有权转移给自己的消费者。`filter` 迭代器会通过共享引用将每个条目传给闭包，并保留所有权以便再把选定的条目传给自己的消费者。这就是为什么该示例必须解引用 `s` 以便将其与 `"iguanas"` 进行比较：`filter` 迭代器的条目类型是 `&str`，因此闭包参数 `s` 的类型是 `&&str`。

关于迭代器适配器，有两点需要特别注意。

第一个要点是，单纯在迭代器上调用适配器并不会消耗任何条目，只会返回一个新的迭代器，新迭代器会根据需要从第一个迭代器中提取条目，以生成自己的条目。在适配器的适配链中，实际完成任何工作（同时消耗条目）的唯一方法是在最终的迭代器上调用 `next`。

因此，在我们之前的示例中，方法调用 `text.lines()` 本身实际上并不会解析字符串中的任何一行，它只是返回了一个迭代器，当需要时才会解析这些行。同样，`map` 和 `filter` 也只会返回新的迭代器，当需要时，它们才会映射或过滤。在由 `collect` 调用 `filter` 迭代器上的 `next` 之前，不会进行任何实际的工作。

如果你在使用有副作用的适配器，这一点尤为重要。例如，以下代码根本不会输出任何内容：

```
["earth", "water", "air", "fire"]
    .iter().map(|elt| println!("{}", elt));
```

`iter` 调用会返回数组元素的迭代器，`map` 调用会返回第二个迭代器，第二个迭代器会对第一个迭代器生成的每个值调用闭包。但是这里没有任何代码会实际用到整个链条的值，所以 `next` 方法永远不会执行。事实上，Rust 会就此发出警告：

```
warning: unused `std::iter::Map` that must be used
|
7 | /      ["earth", "water", "air", "fire"]
8 | |      .iter().map(|elt| println!("{}", elt));
| |      ^
|
= note: iterators are lazy and do nothing unless consumed
```

错误消息中的术语 `lazy`（惰性）不是贬义词，只是用来表示“推迟计算，直到需要该值”这种机制的行话。Rust 的惯例是迭代器应该做尽可能少的必要工作来满足每次对 `next` 的调用，在这个例子中，根本没有这样的调用，所以什么也没做。

第二个要点是，迭代器的适配器是一种零成本抽象。由于 `map`、`filter` 和其他类似的适配器都是泛型的，因此将它们应用于迭代器就会专门针对所涉及的特定迭代器类型生成特化代码。这意味着 Rust 会有足够的信息将每个迭代器的 `next` 方法内联到它的消费者中，然后将这一组功能作为一个单元翻译成机器代码。因此，我们之前展示的迭代器的 `lines/ map/ filter` 链条会和手写代码一样高效：

```
for line in text.lines() {
    let line = line.trim();
    if line != "iguanas" {
        v.push(line);
    }
}
```

下面我们接着介绍可用于 `Iterator` 特型上的各种适配器。

15.3.2 `filter_map` 与 `flat_map`

在每个传入条目都会生成一个传出条目的情况下，`map` 适配器很实用。但是，如果想从迭代中删除而不是处理某些条目，或想用零个或多个条目替换单个条目时该怎么办？`filter_map`（过滤映射）适配器和 `flat_map`（展平映射）适配器为你提供了这种灵活性。

`filter_map` 适配器与 `map` 类似，不同之处在于前者允许其闭包将条目转换为新条目（就像 `map` 那样）或从迭代中丢弃该条目。因此，它有点儿像 `filter` 和 `map` 的组合。它的签名如下所示：

```
fn filter_map<B, F>(self, f: F) -> impl Iterator<Item=B>
    where Self: Sized, F: FnMut(Self::Item) -> Option<B>;
```

它和 `map` 的签名基本相同，不同之处在于这里的闭包会返回 `Option<B>`，而不只是 `B`。当闭包返回 `None` 时，该条目就会从本迭代中丢弃；当返回 `Some(b)` 时，`b` 就是 `filter_map` 迭代器生成的下一个条目。

假设你要扫描字符串，以查找可解析为数值且以空格分隔的单词，然后处理该数值，忽略其他单词。可以这样写：

```
use std::str::FromStr;

let text = "1\nfrond .25 289\n3.1415 estuary\n";
for number in text
    .split_whitespace()
    .filter_map(|w| f64::from_str(w).ok())
{
    println!("{:4.2}", number.sqrt());
}
```

上面的代码会输出以下内容：

```
1.00
0.50
17.00
1.77
```

传给 `filter_map` 的闭包会尝试使用 `f64::from_str` 来解析每个以空格分隔的切片。结果是一个 `Result<f64, ParseFloatError>`，再调用 `.ok()` 就会把它变成 `Option<f64>`：如果解析错误就会变成 `None`；如果成功就会变成 `Some(v)`。`filter_map` 迭代器会丢弃所有 `None` 值并为每个 `Some(v)` 生成值 `v`。

但是像这样把 `map` 和 `filter` 融合成一个操作，而非直接使用两种适配器，意义何在？实际上，刚才的例子已经展示了 `filter_map` 适配器的价值。如果只有先试着实际处理一下条目才能决定是否应该在迭代中包含它，该适配器就会派上用场。只用 `filter` 和 `map` 也可以做同样的事，但略显烦琐：

```
text.split_whitespace()
    .map(|w| f64::from_str(w))
    .filter(|r| r.is_ok())
    .map(|r| r.unwrap())
```

我们可以将 `flat_map` 视为与 `filter_map` 功能类似的适配器，即对 `map` 的功能延伸，只是现在这个闭包不仅可以像 `map` 那样返回一个条目，或像 `filter_map` 那样返回零个或一个条目，还可以返回任意数量的条目序列。也就是说，`flat_map` 迭代器会生成此闭包返回的序列串联后的结果。

`flat_map` 的签名如下所示：

```
fn flat_map<U, F>(self, f: F) -> impl Iterator<Item=U::Item>
    where F: FnMut(Self::Item) -> U, U: IntoIterator;
```

传给 `flat_map` 的闭包必须返回一个可迭代者，但可以返回任意种类的可迭代者。³

³事实上，由于 `Option` 也是一个可迭代者，其行为类似于有零个或一个条目的序列，因此 `iterator.filter_map(closure)` 等效于 `iterator.flat_map(closure)`，这里假设 `closure` 会返回一个 `Option<T>`。

假设有一个将国家映射成其主要城市的表。给定一个国家列表，如何遍历这些国家的主要城市呢？

```
use std::collections::HashMap;

let mut major_cities = HashMap::new();
major_cities.insert("Japan", vec!["Tokyo", "Kyoto"]);
major_cities.insert("The United States", vec!["Portland",
"Nashville"]);
major_cities.insert("Brazil", vec!["São Paulo", "Brasília"]);
major_cities.insert("Kenya", vec!["Nairobi", "Mombasa"]);
major_cities.insert("The Netherlands", vec!["Amsterdam",
"Utrecht"]);

let countries = ["Japan", "Brazil", "Kenya"];

for &city in countries.iter().flat_map(|country|
&major_cities[country]) {
    println!("{}", city);
}
```

上面的代码会输出以下内容：

```
Tokyo  
Kyoto  
São Paulo  
Brasília  
Nairobi  
Mombasa
```

可以这样理解，对于每个国家，我们都会检索其城市的向量，将所有向量串联成一个序列，然后打印出来。

但请记住，迭代器是惰性的：只有当 `for` 循环调用了 `flat_map` 迭代器的 `next` 方法时才会实际工作。完整的串联序列从未在内存中构建过。不过，这里有一个小小的状态机，它会从城市迭代器中逐个提取，直到用完，然后才为下一个国家/地区生成一个新的城市迭代器。其效果实际上和嵌套循环一样，但封装成了迭代器以便使用。⁴

⁴这就是迭代器提供的抽象能力和价值。——译者注

15.3.3 `flatten`

`flatten`（展平）适配器会串联起迭代器的各个条目，这里假设每个条目本身都是可迭代者：

```
use std::collections::BTreeMap;  
  
// 一个把城市映射为城市中停车场的表格：每个值都是一个向量  
let mut parks = BTreeMap::new();  
parks.insert("Portland", vec!["Mt. Tabor Park", "Forest Park"]);  
parks.insert("Kyoto", vec!["Tadasu-no-Mori Forest", "Maruyama Koen"]);  
parks.insert("Nashville", vec!["Percy Warner Park", "Dragon Park"]);  
  
// 构建一个表示全部停车场的向量。`values`给出了一个能生成  
// 向量的迭代器，然后`flatten`会依次生成每个向量的元素  
let all_parks: Vec<_> =  
parks.values().flatten().cloned().collect();  
  
assert_eq!(all_parks,  
           vec!["Tadasu-no-Mori Forest", "Maruyama Koen", "Percy
```

```
Warner Park",  
    "Dragon Park", "Mt. Tabor Park", "Forest Park"]]);
```

名称 "flatten" 来自将二级结构展平为一级结构的直观图景：BTreeMap 及其表示停车场名称的 Vec 会被展平为一个能生成所有名称的迭代器。

flatten 的签名如下所示：

```
fn flatten(self) -> impl Iterator<Item=Self::Item::Item>  
    where Self::Item: IntoIterator;
```

这意味着，底层迭代器的条目必须自行实现 IntoIterator 才能真正形成一个由序列组成的序列。然后，flatten 方法会返回一个迭代器，该迭代器会生成这些序列的串联。当然，这都是惰性执行的，只有当我们迭代完上一个序列才会从 self 中提取一个新序列。

flatten 方法还有一些看似不寻常的用法。如果你只想从一个 Vec<Option<...>> 中迭代出 Some 的值，那么 flatten 可以漂亮地完成此任务：

```
assert_eq!(vec![None, Some("day"), None, Some("one")]  
    .into_iter()  
    .flatten()  
    .collect::<Vec<_>>(),  
    vec!["day", "one"]);
```

这是因为 Option 本身也实现了 IntoIterator，表示由 0 个或 1 个元素组成的序列。None 元素对迭代没有任何贡献，而每个 Some 元素都会贡献一个值。同样，也可以用 flatten 来迭代 Option<Vec<...>> 值：其中 None 的行为等同于空向量。

Result 也实现了 IntoIterator，其中 Err 表示一个空序列，因此将 flatten 应用于 Result 值的迭代器有效地排除了所有 Err 并将它们丢弃，进而产生了一个解包装过的由成功值组成的流。通常不应该忽略代码中的错误，但如果你很清楚自己在做什么，那么这可能是个有用的小技巧。

如果你发现自己随手用了 `flatten`，这个时候你真正需要的可能是 `flat_map`。例如，标准库的 `str::to_uppercase` 方法可以将字符串转换为大写，其工作方式如下所示：

```
fn to_uppercase(&self) -> String {
    self.chars()
        .map(char::to_uppercase)
        .flatten() // 使用flat_map更好
        .collect()
}
```

用 `flatten` 的原因是 `ch.to_uppercase()` 返回的不是单个字符，而是会生成一个或多个字符的迭代器。将每个字符都映射成其对应的大写字母会生成由字符迭代器组成的迭代器，而 `flatten` 负责将它们拼接在一起，形成我们最终可以收集（`collect`）到 `String` 中的内容。

不过 `map` 和 `flatten` 的组合非常常见，因此 `Iterator` 为这种情况提供了 `flat_map` 适配器。（事实上，`flat_map` 比 `flatten` 先纳入标准库。）因此，可以把前面的代码改写成如下形式。

```
fn to_uppercase(&self) -> String {
    self.chars()
        .flat_map(char::to_uppercase)
        .collect()
}
```

15.3.4 `take` 与 `take_while`

`Iterator` 特型的 `take`（取出）适配器和 `take_while`（当……时取出）适配器的作用是当条目达到一定数量或闭包决定中止时结束迭代。它们的签名如下所示：

```
fn take(self, n: usize) -> impl Iterator<Item=Self::Item>
    where Self: Sized;

fn take_while<P>(self, predicate: P) -> impl
```

```
Iterator<Item=Self::Item>
    where Self: Sized, P: FnMut(&Self::Item) -> bool;
```

两者都会接手某个迭代器的所有权并返回一个新的迭代器，新的迭代器会从第一个迭代器中传递条目，并可能提早终止序列。`take` 迭代器会在最多生成 `n` 个条目后返回 `None`。`take_while` 迭代器会针对每个条目调用 `predicate`，并对 `predicate` 返回了 `false` 的首个条目以及其后的每个条目都返回 `None`。

例如，给定一封电子邮件，其中有一个空行将标题与邮件正文分隔开，如果只想遍历标题就可以使用 `take_while`：

```
let message = "To: jimb\r\n\
               From: superego <editor@oreilly.com>\r\n\
               \r\n\
               Did you get any writing done today?\r\n\
               When will you stop wasting time plotting fractals?
\r\n";
for header in message.lines().take_while(|l| !l.is_empty()) {
    println!("{}", header);
}
```

回想一下 3.7.1 节，当字符串中的一行以反斜杠结尾时，Rust 中不会包含字符串中下一行的缩进，因此字符串中的任何一行都没有前导空格。这意味着 `message` 的第 3 行是空白的。`take_while` 适配器一看到空行就会中止此迭代，所以此代码只会打印两行。

```
To: jimb
From: superego <editor@oreilly.com>
```

15.3.5 `skip` 与 `skip_while`

Iterator 特型的 `skip`（跳过）和 `skip_while`（当……时跳过）是与 `take` 和 `take_while` 互补的方法：`skip` 从迭代开始时就丢弃一定数量的条目，`skip_while` 则一直丢弃条目直到闭包终于找到一个可接受的条目为止，然后将剩下的条目按照原样传递出来。它们的签名如下所示：

```
fn skip(self, n: usize) -> impl Iterator<Item=Self::Item>
    where Self: Sized;

fn skip_while<P>(self, predicate: P) -> impl
    Iterator<Item=Self::Item>
    where Self: Sized, P: FnMut(&Self::Item) -> bool;
```

`skip` 适配器的常见用途之一是在迭代程序的命令行参数时跳过命令本身的名称。在第 2 章中，我们的最大公约数计算器曾用如下代码循环其命令行参数：

```
for arg in std::env::args().skip(1) {
    ...
}
```

`std::env::args` 函数会返回一个迭代器，该迭代器会将程序的各个参数生成为一些 `String` 型条目，首个条目是程序本身的名称，但这并不是我们要在循环中处理的字符串。对该迭代器调用 `skip(1)` 会返回一个新的迭代器，新迭代器会在首次调用时丢弃程序名称，然后生成所有后续参数。

`skip_while` 适配器会使用闭包决定从序列的开头丢弃多少个条目。你还可以像下面这样遍历 15.3.4 节中消息正文的各行：

```
for body in message.lines()
    .skip_while(|l| !l.is_empty())
    .skip(1) {
    println!("{}", body);
}
```

这会使用 `skip_while` 来跳过非空行，但迭代器本身还是会生成一个空行——毕竟，闭包对该空行返回了 `false`。所以我们还要使用 `skip` 方法来丢弃它，并返回一个迭代器，其第一个条目是消息正文的第 1 行。与 15.3.4 节中的 `message` 声明合用，此代码会输出如下内容。

```
Did you get any writing done today?
When will you stop wasting time plotting fractals?
```

15.3.6 peekable

peekable（可窥视）迭代器的功能是允许我们窥视即将生成的下一个条目，而无须实际消耗它。调用 `Iterator` 特型的 `peekable` 方法可以将任何迭代器变成 `peekable` 迭代器：

```
fn peekable(self) -> std::iter::Peekable<Self>
    where Self: Sized;
```

在这里，`Peekable<Self>` 是一个实现了 `Iterator<Item=Self::Item>` 的结构体，而 `Self` 是底层迭代器的类型。

`peekable` 迭代器有一个额外的方法 `peek`，该方法会返回一个 `Option<&Item>`：如果底层迭代器已耗尽，那么返回值就为 `None`；否则为 `Some(r)`，其中 `r` 是对下一个条目的共享引用。

（注意，如果迭代器的条目类型已经是对某个值的引用了，则最终产出就会是对引用的引用。）

调用 `peek` 会尝试从底层迭代器中提取下一个条目，如果条目存在，就将其缓存，直到下一次调用 `next` 时给出。`Peekable` 上的所有其他 `Iterator` 方法都知道这个缓存，比如，`peekable` 迭代器 `iter` 上的 `iter.last()` 就知道要在耗尽底层迭代器后检查此缓存。

有时候，只有超前一点儿才能决定应该从迭代器中消耗多少个条目，在这种情况下，`peekable` 迭代器就变得至关重要。如果要从字符流中解析数值，那么在看到数值后面的第一个非数值字符之前是无法确定该数值的结束位置的：

```
use std::iter::Peekable;

fn parse_number<I>(tokens: &mut Peekable<I>) -> u32
    where I: Iterator<Item=char>
{
    let mut n = 0;
    loop {
```



```

        match tokens.peek() {
            Some(r) if r.is_digit(10) => {
                n = n * 10 + r.to_digit(10).unwrap();
            }
            _ => return n
        }
        tokens.next();
    }
}

let mut chars = "226153980,1766319049".chars().peekable();
assert_eq!(parse_number(&mut chars), 226153980);
// 注意, `parse_number` 并没有消耗这个逗号, 所以我们能看到它
assert_eq!(chars.next(), Some(','));
assert_eq!(parse_number(&mut chars), 1766319049);
assert_eq!(chars.next(), None);

```

`parse_number` 函数会使用 `peek` 来检查下一个字符, 只有当它是数字时才消耗它。如果它不是数字或迭代器已消耗完 (也就是说, `peek` 返回了 `None`), 我们将返回已解析的数值并将下一个字符留在迭代器中, 以供使用。

15.3.7 fuse

`Iterator` 特型并没有规定一旦 `next` 返回 `None` 之后, 再次调用 `next` 方法时应该如何行动。大多数迭代器只是再次返回 `None`, 但也有例外。如果你的代码依赖于“再次返回 `None`”这种行为, 那么遇到例外可能会让你大吃一惊。

`fuse` (保险丝) 适配器能接受任何迭代器并生成一个确保在第一次返回 `None` 后继续返回 `None` 的迭代器:

```

struct Flaky(bool);

impl Iterator for Flaky {
    type Item = &'static str;
    fn next(&mut self) -> Option<Self::Item> {
        if self.0 {
            self.0 = false;
            Some("totally the last item")
        } else {
            ...

```

```

        self.0 = true; // 糟糕!
        None
    }
}

let mut flaky = Flaky(true);
assert_eq!(flaky.next(), Some("totally the last item"));
assert_eq!(flaky.next(), None);
assert_eq!(flaky.next(), Some("totally the last item"));

let mut not_flaky = Flaky(true).fuse();
assert_eq!(not_flaky.next(), Some("totally the last item"));
assert_eq!(not_flaky.next(), None);
assert_eq!(not_flaky.next(), None);

```

`fuse` 适配器在需要使用不明来源迭代器的泛型代码中非常有用。与其奢望要处理的每个迭代器都表现良好，还不如使用 `fuse` 加上保险。

15.3.8 可逆迭代器与 `rev`

有的迭代器能够从序列的两端抽取条目，使用 `rev`（逆转）适配器可以反转此类迭代器。例如，向量上的迭代器就可以像从头开始一样轻松地 从向量的末尾抽取条目。这样的迭代器可以实现 `std::iter::DoubleEndedIterator` 特型，它扩展了 `Iterator`：

```

trait DoubleEndedIterator: Iterator {
    fn next_back(&mut self) -> Option<Self::Item>;
}

```

你可以将双端迭代器想象成用两根手指分别标记序列的当前首端和尾端。从任何一端提取条目都会让该手指向另一端前进，当两者相遇时，迭代就完成了：

```

let bee_parts = ["head", "thorax", "abdomen"];

let mut iter = bee_parts.iter();
assert_eq!(iter.next(), Some(&"head"));

```

```
assert_eq!(iter.next_back(), Some(&"abdomen"));
assert_eq!(iter.next(),      Some(&"thorax"));

assert_eq!(iter.next_back(), None);
assert_eq!(iter.next(),      None);
```

基于切片迭代器的结构，这种行为实现起来非常简单：切片迭代器实际上是一对指向我们尚未生成的元素范围的起始指针和结束指针，`next` 和 `next_back` 所做的只是从起始指针或结束指针中提取一个条目而已。`BTreeSet` 和 `BTreeMap` 等有序集合的迭代器也是双端的：它们的 `next_back` 方法会首先提取最大的元素或条目。总体而言，只要有可能，标准库就会提供双端迭代能力。

但并非所有迭代器都能如此轻松地实现，比如，从通道的 `Receiver` 中生成来自其他线程的值的迭代器显然无法预测最后接收的值可能是什么。一般来说，你需要查看标准库的文档以了解哪些迭代器实现了 `DoubleEndedIterator`，哪些没有实现。

如果迭代器是双端的，就可以用 `rev` 适配器将其逆转：

```
fn rev(self) -> impl Iterator<Item=Self>
    where Self: Sized + DoubleEndedIterator;
```

返回的迭代器也是双端的，只是互换了 `next` 方法和 `next_back` 方法：

```
let meals = ["breakfast", "lunch", "dinner"];

let mut iter = meals.iter().rev();
assert_eq!(iter.next(), Some(&"dinner"));
assert_eq!(iter.next(), Some(&"lunch"));
assert_eq!(iter.next(), Some(&"breakfast"));
assert_eq!(iter.next(), None);
```

大多数适配器，如果应用到某个可逆迭代器上，就会返回另一个可逆迭代器，比如，`map` 和 `filter` 都会保留可逆性。

15.3.9 inspect

`inspect`（探查）适配器为调试迭代器适配器的流水线提供了便利，但在生产代码中用得不多。`inspect` 只是对每个条目的共享引用调用闭包，然后传递该条目。闭包不会影响条目，但可以做一些事情，比如打印它们或对它们进行断言。

本示例演示了将字符串转换为大写会更改其长度的情况：

```
let upper_case: String = "große".chars()
    .inspect(|c| println!("before: {:?}", c))
    .flat_map(|c| c.to_uppercase())
    .inspect(|c| println!(" after:      {:?}", c))
    .collect();
assert_eq!(upper_case, "GROSSE");
```

小写德语字母 `ß` 的大写形式是 `SS`，这就是 `char::to_uppercase` 会返回一个字符迭代器而不是单个字符的原因。前面的代码使用了 `flat_map` 将 `to_uppercase` 返回的所有序列连接成一个 `String`，同时输出以下内容。

```
before: 'g'
after:   'G'
before: 'r'
after:   'R'
before: 'o'
after:   'O'
before: 'ß'
after:   'S'
after:   'S'
before: 'e'
after:   'E'
```

15.3.10 **chain**

`chain`（链接）适配器会将一个迭代器追加到另一个迭代器之后。更准确地说，`i1.chain(i2)` 会返回一个迭代器，该迭代器从 `i1` 中提取条目，直到用尽，然后从 `i2` 中提取条目。

`chain` 适配器的签名如下所示：

```
fn chain<U>(self, other: U) -> impl Iterator<Item=Self::Item>
    where Self: Sized, U: IntoIterator<Item=Self::Item>;
```

换句话说，你可以将迭代器与任何会生成相同条目类型的可迭代者链接在一起。

例如：

```
let v: Vec<i32> = (1..4).chain([20, 30, 40]).collect();
assert_eq!(v, [1, 2, 3, 20, 30, 40]);
```

如果 `chain` 的两个底层迭代器都是可逆的，则其结果迭代器也是可逆的：

```
let v: Vec<i32> = (1..4).chain([20, 30, 40]).rev().collect();
assert_eq!(v, [40, 30, 20, 3, 2, 1]);
```

`chain` 迭代器会跟踪这两个底层迭代器是否返回了 `None` 并按需把其中一个迭代器的 `next` 和 `next_back` 重定向到另一个迭代器的 `next` 和 `next_back`。

15.3.11 `enumerate`

`Iterator` 特型的 `enumerate`（枚举）适配器会将运行索引附加到序列中，它接受某个迭代器生成的条目 `A`, `B`, `C`, ... 并返回生成的值对 `(0, A)`, `(1, B)`, `(2, C)`, ...。乍看起来，这微不足道，但其使用频率相当惊人。

消费者可以使用上述索引将一个条目与另一个条目区分开来，并建立处理每个条目时的上下文。例如，第 2 章中的曼德博集绘图器会将图像分成 8 个水平条带，并将每个条带分配给不同的线程。该代码就使用了 `enumerate` 来告诉每个线程其条带对应于图像的哪个部分。

从一个矩形像素缓冲区开始：

```
let mut pixels = vec![0; columns * rows];
```

接下来使用 `chunks_mut` 将图像拆分为一些水平条带，每个线程负责一个：

```
let threads = 8;
let band_rows = rows / threads + 1;
...
let bands: Vec<&mut [u8]> = pixels.chunks_mut(band_rows *
columns).collect();
```

然后遍历条带，为每个条带启动一个线程：

```
for (i, band) in bands.into_iter().enumerate() {
    let top = band_rows * i;
    // 启动一个线程来渲染`top..top + band_rows`范围内的行
    ...
}
```

每次迭代都会得到一个 `(i, band)` 值对，其中 `band` 是 `&mut [u8]` 类型的像素缓冲区切片，表示线程应该绘制的区域，而 `i` 是该条带在整个图像中的索引，由 `enumerate` 适配器提供。绘图的边界和条带的大小足以让线程确定分配给它的是图像中的哪个部分，从而确定要在 `band` 中绘制什么。

可以将 `enumerate` 生成的 `(index, item)` 值对视为在迭代 `HashMap` 或其他关联集合时获得的 `(key, value)` 值对。如果在切片或向量上进行迭代，则 `index` 就是 `item` 对应的 `key`。

15.3.12 **zip**

`zip`（拉合）适配器会将两个迭代器组合成一个迭代器，新的迭代器会生成值对，每个底层迭代器各提供一个值，就像把拉链的两侧拉合起来一样。当两个底层迭代器中的任何一个已结束，拉合后的迭代器就结束了。

例如，可以通过将无尽范围 `0..` 与一个迭代器拉合起来获得与 `enumerate` 适配器相同的效果：

```
let v: Vec<_> = (0..).zip("ABCD".chars()).collect();
assert_eq!(v, vec![(0, 'A'), (1, 'B'), (2, 'C'), (3, 'D')]);
```

从这个意义上说，你可以将 `zip` 视为 `enumerate` 的泛化版本：`enumerate` 会将索引附加到序列，而 `zip` 能附加来自任意迭代器的条目。之前我们建议用 `enumerate` 在处理条目时协助提供上下文，而 `zip` 提供了一种更灵活的方式来实现同样的效果。

`zip` 的参数本身不一定是迭代器，可以是任意可迭代者。

```
use std::iter::repeat;

let endings = ["once", "twice", "chicken soup with rice"];
let rhyme: Vec<_> = repeat("going")
    .zip(endings)
    .collect();
assert_eq!(rhyme, vec![("going", "once"),
                        ("going", "twice"),
                        ("going", "chicken soup with rice")]);
```

15.3.13 `by_ref`

前面我们一直在将适配器附加到迭代器上。一旦开始这样做，还能再取下适配器吗？一般来说是不能，因为适配器会接手底层迭代器的所有权，并且没有提供归还所有权的方法。

迭代器的 `by_ref`（按引用）方法会借入迭代器的可变引用，便于将各种适配器应用于该引用。一旦消耗完适配器中的条目，就会丢弃这些适配器，借用也就结束了，然后你就能重新获得对原始迭代器的访问权。

例如，在本章前面我们展示过如何使用 `take_while` 和 `skip_while` 来处理邮件消息的标题行和正文。但是，如果想让两者使用同一个底层迭代器来处理邮件消息，该怎么办呢？借助 `by_ref`，我们就可以使用 `take_while` 来处理邮件头，完成这些之后，取回底层迭代器，此时 `take_while` 恰好位于处理消息正文的适当位置：

```

let message = "To: jimb\r\n\
               From: id\r\n\
               \r\n\
               Ooooooh, donuts!!\r\n";

let mut lines = message.lines();

println!("Headers:");
for header in lines.by_ref().take_while(|l| !l.is_empty()) {
    println!("{}", header);
}

println!("\nBody:");
for body in lines {
    println!("{}", body);
}

```

调用 `lines.by_ref()` 会借出一个对迭代器的可变引用，`take_while` 迭代器会取得这个引用的所有权。该迭代器在第一个 `for` 循环结束时超出了作用域，表示本次借用已结束，这样你就能在第二个 `for` 循环中再次使用 `lines` 了。上述代码将输出以下内容：

```

Headers:
To: jimb
From: id

Body:
Ooooooh, donuts!!

```

`by_ref` 适配器的定义很简单：它会返回对迭代器的可变引用。然后，标准库中还包含了这个神奇的小实现：

```

impl<'a, I: Iterator + ?Sized> Iterator for &'a mut I {
    type Item = I::Item;
    fn next(&mut self) -> Option<I::Item> {
        (**self).next()
    }
    fn size_hint(&self) -> (usize, Option<usize>) {
        (**self).size_hint()
    }
}

```


换句话说，如果 `I` 是某种迭代器类型，那么 `&mut I` 就同样是一个迭代器，其 `next` 方法和 `size_hint` 方法会转发给其引用目标。当你在此迭代器的可变引用上调用某个适配器时，适配器会取得引用（而不是迭代器本身）的所有权。当适配器超出作用域时，本次借用就会结束。

15.3.14 `cloned` 与 `copied`

`cloned`（克隆后）适配器会接受一个生成引用的迭代器，并返回一个会生成从这些引用克隆而来的值的迭代器，就像 `iter.map(|item| item.clone())`。当然，引用目标的类型也必须实现了 `Clone`：

```
let a = ['1', '2', '3', '∞'];

assert_eq!(a.iter().next(),          Some(&'1'));
assert_eq!(a.iter().cloned().next(), Some('1'));
```

`copied`（复制后）适配器的设计思想同样如此，但限制更严格，它要求引用目标的类型必须实现了 `Copy`。像 `iter.copied()` 这样的调用与 `iter.map(|r| *r)` 大致相同。由于每个实现了 `Copy` 的类型也必定实现了 `Clone`，因此 `cloned` 更通用。但根据条目类型的不同，`clone` 调用可能会进行任意次数的分配和复制。如果你认为由于条目类型很简单，因而永远不会发生内存分配，那么最好使用 `copied` 来让类型检查器帮你验证这种假设。

15.3.15 `cycle`

`cycle`（循环）适配器会返回一个迭代器，它会无限重复底层迭代器生成的序列。底层迭代器必须实现 `std::clone::Clone`，以便 `cycle` 保存其初始状态并且在每次循环重新开始时复用它。下面是一个例子：

```
let dirs = ["North", "East", "South", "West"];
let mut spin = dirs.iter().cycle();
```

```
assert_eq!(spin.next(), Some(&"North"));
assert_eq!(spin.next(), Some(&"East"));
assert_eq!(spin.next(), Some(&"South"));
assert_eq!(spin.next(), Some(&"West"));
assert_eq!(spin.next(), Some(&"North"));
assert_eq!(spin.next(), Some(&"East"));
```

或者，下面是纯粹为了演示迭代器的用法而编写的代码：

```
use std::iter::{once, repeat};

let fizzes = repeat("").take(2).chain(once("fizz")).cycle();
let buzzes = repeat("").take(4).chain(once("buzz")).cycle();
let fizzes_buzzes = fizzes.zip(buzzes);

let fizz_buzz = (1..100).zip(fizzes_buzzes)
    .map(|tuple|
        match tuple {
            (i, ("", "")) => i.to_string(),
            (_, (fizz, buzz)) => format!("{}", fizz, buzz)
        });

for line in fizz_buzz {
    println!("{}", line);
}
```

这是一个儿童文字游戏，现在有时会用作程序员的求职面试问题：玩家轮流数数，将任何可被 3 整除的数值替换为单词 `fizz`；将任何可被 5 整除的数值替换为单词 `buzz`；能被两者整除的数值则替换为单词 `fizzbuzz`。

15.4 消耗迭代器

迄今为止，我们已经介绍了创建迭代器并将它们适配成新迭代器的方法，在这里，我们会通过展示如何消耗它们来完成整个处理过程。

当然，你可以使用带有 `for` 循环的迭代器，也可以显式调用 `next`，但有许多常见任务不必一遍又一遍地写出来。`Iterator` 类型提供了一大组可选方法来涵盖其中的许多任务。

15.4.1 简单累加：**`count`**、**`sum`** 和 **`product`**

`count`（计数）方法会从迭代器中提取条目，直到迭代器返回 `None`，并报告提取的条目数。下面是一个计算标准输入行数的小程序：⁵

⁵要结束输入，请用 `Ctrl-Z`（Windows）或 `Ctrl-D`（Linux/Mac）组合键。——译者注

```
use std::io::prelude::*;

fn main() {
    let stdin = std::io::stdin();
    println!("{}", stdin.lock().lines().count());
}
```

`sum`（求和）方法和 `product`（乘积）方法会分别计算迭代器条目之和与乘积，结果必须是整数或浮点数。

```
fn triangle(n: u64) -> u64 {
    (1..=n).sum()
}
assert_eq!(triangle(20), 210);

fn factorial(n: u64) -> u64 {
    (1..=n).product()
}
assert_eq!(factorial(20), 2432902008176640000);
```

（为方便与其他类型一起工作，可以通过实现 `std::iter::Sum` 特型和 `std::iter::Product` 特型来扩展 `sum` 和 `product`，这里我们就不展开讲解这些特型了。）

15.4.2 `min` 与 `max`

Iterator 上的 `min`（最小）方法和 `max`（最大）方法会分别返回迭代器生成的最小条目与最大条目。迭代器的条目类型必须实现 `std::cmp::Ord`，这样条目之间才能相互比较：

```
assert_eq!([-2, 0, 1, 0, -2, -5].iter().max(), Some(&1));
assert_eq!([-2, 0, 1, 0, -2, -5].iter().min(), Some(&-5));
```

这些方法会返回一个 `Option<Self::Item>` 以便当迭代器不再生成任何条目时能返回 `None`。

就像 12.2 节中所讲的，Rust 的浮点类型 `f32` 和 `f64` 仅实现了 `std::cmp::PartialOrd` 而没有实现 `std::cmp::Ord`，因此不能使用 `min` 方法和 `max` 方法来计算浮点数序列中的最小值和最大值。这在 Rust 设计中并不讨喜，却是经过深思熟虑的——因为不清楚这些函数该如何处理 IEEE 的 NaN 值，如果只是简单地忽略则可能会掩盖代码中更严重的问题。

如果知道如何处理 NaN 值，则可以改用 `max_by` 和 `min_by` 迭代器方法，这样你就可以提供自己的比较函数了。

15.4.3 `max_by` 与 `min_by`

`max_by`（据……最大）方法和 `min_by`（据……最小）方法会分别返回迭代器生成的最大条目与最小条目，由你提供的比较函数确定规则：

```
use std::cmp::Ordering;

// 比较两个f64值，如果其一是NaN，则引发panic
fn cmp(lhs: &f64, rhs: &f64) -> Ordering {
```

```
        lhs.partial_cmp(rhs).unwrap()
    }

let numbers = [1.0, 4.0, 2.0];
assert_eq!(numbers.iter().copied().max_by(cmp), Some(4.0));
assert_eq!(numbers.iter().copied().min_by(cmp), Some(1.0));

let numbers = [1.0, 4.0, std::f64::NAN, 2.0];
assert_eq!(numbers.iter().copied().max_by(cmp), Some(4.0)); // 引发 panic
```

`max_by` 方法和 `min_by` 方法会通过引用将条目传给比较函数，这样一来，这两个方法就可以与任意种类的迭代器一起高效配合使用。在上面的代码中，虽然我们已经在 `copied` 获取了会生成 `f64` 条目的迭代器，但 `cmp` 函数还是期望通过引用获取其参数。

15.4.4 `max_by_key` 与 `min_by_key`

使用 `Iterator` 上的 `max_by_key`（据键最大）方法和 `min_by_key`（据键最小）方法可以选择最大条目或最小条目，由针对每个条目调用的闭包确定。闭包可以选择条目的某些字段或对此条目执行某些计算。由于你通常只对与某些最小值或最大值相关的数据感兴趣，而不仅仅是极值本身，因此这两个函数通常比 `max` 和 `min` 更有用。它们的签名如下所示：

```
fn min_by_key<B: Ord, F>(self, f: F) -> Option<Self::Item>
    where Self: Sized, F: FnMut(&Self::Item) -> B;

fn max_by_key<B: Ord, F>(self, f: F) -> Option<Self::Item>
    where Self: Sized, F: FnMut(&Self::Item) -> B;
```

也就是说，给定一个接受某条目并返回任意有序类型 `B` 的闭包，则返回那个调用闭包时所返回的 `B` 为最大或最小的条目；如果没有生成任何条目，则返回 `None`。

例如，要扫描城市的哈希表，分别查找人口最多和最少的城市，可以这样写：

```

use std::collections::HashMap;

let mut populations = HashMap::new();
populations.insert("Portland", 583_776);
populations.insert("Fossil", 449);
populations.insert("Greenhorn", 2);
populations.insert("Boring", 7_762);
populations.insert("The Dalles", 15_340);

assert_eq!(populations.iter().max_by_key(|&(_name, pop)| pop),
           Some((&"Portland", &583_776)));
assert_eq!(populations.iter().min_by_key(|&(_name, pop)| pop),
           Some((&"Greenhorn", &2)));

```

闭包 `|&(_name, pop)| pop` 会针对迭代器生成的每个条目进行调用并返回要用于比较的值——在本例中为城市人口。这两个方法返回的值是整个条目，而不仅仅是闭包返回的值。（当然，如果要频繁进行这样的条目查询，最好用一种比在表中进行线性查找更高效的方式。）

15.4.5 对条目序列进行比较

如果字符串、向量和切片的各个元素是可比较的，我们就可以使用 `<` 运算符和 `==` 运算符来对它们进行比较。但是，比较运算符不能用来比较迭代器，这项工作是由像 `eq` 和 `lt` 这样的方法来完成的。相关方法从迭代器中成对取出条目并比较，直到得出结果：

```

let packed = "Helen of Troy";
let spaced = "Helen   of       Troy";
let obscure = "Helen of Sandusky"; // 好人，只是不出名

assert!(packed != spaced);
assert!(packed.split_whitespace().eq(spaced.split_whitespace()));

// 此断言为真，因为' ' < 'o'
assert!(spaced < obscure);

// 此断言为真，因为'Troy' > 'Sandusky'
assert!(spaced.split_whitespace().gt(obscure.split_whitespace()));

```

调用 `split_whitespace` 会返回字符串中以空白字符分隔的单词的迭代器。在这些迭代器上使用 `eq` 方法和 `gt` 方法会进行逐词比较（而不是逐字符比较），因为 `&str` 实现了 `PartialOrd` 和 `PartialEq`。

迭代器提供的比较方法既包括用于相等比较的 `eq` 方法和 `ne` 方法，也包括用于有序比较的 `lt` 方法、`le` 方法、`gt` 方法和 `ge` 方法。而 `cmp` 方法和 `partial_cmp` 方法的行为类似于 `Ord` 特型和 `PartialOrd` 特型的相应方法。

15.4.6 `any` 与 `all`

`any`（任意）方法和 `all`（所有）方法会将闭包应用于迭代器生成的每个条目。如果闭包对任意条目返回了 `true` 或对所有条目都返回了 `true`，则相应的方法返回 `true`：

```
let id = "Iterator";

assert!(id.chars().any(char::is_uppercase));
assert!(!id.chars().all(char::is_uppercase));
```

这两个方法只会消耗确定答案所需的尽可能少的条目。如果闭包已经为给定条目返回了 `true`，则 `any` 会立即返回 `true`，不会再从迭代器中提取更多条目。

15.4.7 `position`、`rposition` 和 `ExactSizeIterator`

`position`（位置）方法会针对迭代器中的每个条目调用闭包，并返回调用结果为 `true` 的第一个条目的索引。确切而言，它会返回关于此索引的 `Option`：如果闭包没有为任何条目返回 `true`，则 `position` 返回 `None`。一旦闭包返回 `true`，`position` 就会停止提取条目：

```
let text = "Xerxes";
assert_eq!(text.chars().position(|c| c == 'e'), Some(1));
assert_eq!(text.chars().position(|c| c == 'z'), None);
```

`rposition`（右起位置）方法也是一样的，只是从右侧开始搜索：

```
let bytes = b"Xerxes";
assert_eq!(bytes.iter().rposition(|&c| c == b'e'), Some(4));
assert_eq!(bytes.iter().rposition(|&c| c == b'X'), Some(0));
```

`rposition` 方法要求使用可逆迭代器，以便它能从此序列的右端提取条目。另外，它也要求这是确切大小迭代器，以便能像 `position` 一样对索引进行赋值，最左边的索引值为 0。所谓确切大小迭代器就是实现了 `std::iter::ExactSizeIterator` 特型的迭代器：

```
trait ExactSizeIterator: Iterator {
    fn len(&self) -> usize { ... }
    fn is_empty(&self) -> bool { ... }
}
```

`len` 方法会返回剩余的条目数，而 `is_empty` 方法会在迭代完成时返回 `true`。

自然，也不是每个迭代器都能预知要生成的条目数，比如，之前使用的 `str::chars` 迭代器就不能（UTF-8 是可变宽度编码），因此不能在字符串上使用 `rposition`。但是字节数组上的迭代器必定知道数组的长度，因此它可以实现 `ExactSizeIterator`。

15.4.8 `fold` 与 `rfold`

`fold`（折叠）方法是一种非常通用的工具，用于在迭代器生成的整个条目序列上累积某种结果。给定一个初始值（我们称之为累加器）和一个闭包，`fold` 会以当前累加器和迭代器中的下一个条目为参数反复调用这个闭包。闭包返回的值被视为新的累加器，并将其与下一个条目一起传给闭包。最终，累加器的值就是 `fold` 本身返回的值。如果序列为空，则 `fold` 只返回初始累加器。

使用迭代器值的许多其他方法可以改写成对 `fold` 的使用：

```
let a = [5, 6, 7, 8, 9, 10];

assert_eq!(a.iter().fold(0, |n, _| n+1), 6);           // 计数
assert_eq!(a.iter().fold(0, |n, i| n+i), 45);         // 求和
assert_eq!(a.iter().fold(1, |n, i| n*i), 151200);     // 乘积

// 最大值
assert_eq!(a.iter().cloned().fold(i32::min_value(),
std::cmp::max),
10);
```

`fold` 方法的签名如下所示：

```
fn fold<A, F>(self, init: A, f: F) -> A
    where Self: Sized, F: FnMut(A, Self::Item) -> A;
```

这里，`A` 是累加器的类型。闭包的第一个参数 `init` 及其返回值都是类型 `A`，`fold` 自身的返回值也是类型 `A`。

请注意，累加器的值会移动进闭包或者从闭包中移动出来，因此你可以将 `fold` 与各种非 `Copy` 的累加器类型一起使用：

```
let a = ["Pack", "my", "box", "with",
        "five", "dozen", "liquor", "jugs"];

// 另见切片的`join`方法，它不会像这里一样帮你在末尾添加额外的空格
let pangram = a.iter()
    .fold(String::new(), |s, w| s + w + " ");
assert_eq!(pangram, "Pack my box with five dozen liquor jugs ");
```

`rfold`（右起折叠）方法与 `fold` 方法基本相同，但需要一个双端迭代器，并从后往前处理各个条目。

```
let weird_pangram = a.iter()
    .rfold(String::new(), |s, w| s + w + " ");
assert_eq!(weird_pangram, "jugs liquor dozen five with box my Pack
");
```

15.4.9 try_fold 与 try_rfold

`try_fold`（尝试折叠）方法与 `fold` 方法基本相同，不过迭代可以提前退出，无须消耗迭代器中的所有值。传给 `try_fold` 的闭包返回的值会指出它是应该立即返回，还是继续折叠迭代器的条目。

闭包可以返回多种类型的值，根据类型值，`try_fold` 方法可判断继续折叠的方式。

- 如果闭包返回 `Result<T, E>`，可能是因为它执行了 I/O 或其他一些容易出错的操作，那就返回 `Ok(v)` 令 `try_fold` 继续折叠，同时将 `v` 作为新的累加器值。如果返回 `Err(e)`，则 `try_fold` 立即停止折叠。折叠后的最终值是一个带有最终累加器值的 `Result`，或者由闭包返回的错误值。
- 如果闭包返回 `Option<T>`，则 `Some(v)` 表示折叠应该以 `v` 作为新的累加器值继续前进，而 `None` 表示迭代应该立即停止。折叠后的最终值也是 `Option` 类型的。
- 最后，闭包还可以返回一个 `std::ops::ControlFlow` 值。这种类型是一个具有两个变体的枚举，即 `Continue(c)` 和 `Break(b)`，分别表示使用新的累加器值 `c` 继续或提前中止迭代。折叠的最终结果是一个 `ControlFlow` 值：如果折叠消耗了整个迭代器，并生成了最终的累加器值 `v`，则为 `Continue(v)`；如果闭包中途返回了值 `b`，则为 `Break(b)`。

`Continue(c)` 和 `Break(b)` 的行为与 `Ok(c)` 和 `Err(b)` 完全一样。使用 `ControlFlow` 而不用 `Result` 的优点在于，有时候提前退出并不表示出错了，而只是表明提前得出了答案，这种情况下它会让代码可读性更好。我们接下来会展示一个例子。

下面是一段对从标准输入读取的数值进行求和的程序：

```
use std::error::Error;
use std::io::prelude::*;
use std::str::FromStr;

fn main() -> Result<(), Box<dyn Error>> {
```

```

let stdin = std::io::stdin();
let sum = stdin.lock()
    .lines()
    .try_fold(0, |sum, line| -> Result<u64, Box<dyn Error>> {
        Ok(sum + u64::from_str(&line?.trim())?)
    })?;
println!("{}", sum);
Ok(())
}

```

缓冲输入流上的 `lines` 迭代器会生成 `Result<String, std::io::Error>` 类型的条目，并且将 `String` 解析为整数时也可能出错。在这里使用 `try_fold` 会让闭包返回 `Result<u64, ...>`，所以我们可以使用 `?` 运算符将本次失败从闭包传播到 `main` 函数。

`try_fold` 非常灵活，被用来实现 `Iterator` 的许多其他消费者方法，比如，以下是 `all` 的一种实现：

```

fn all<P>(&mut self, mut predicate: P) -> bool
    where P: FnMut(Self::Item) -> bool,
           Self: Sized
{
    use std::ops::ControlFlow::*;
    self.try_fold((), |_, item| {
        if predicate(item) { Continue(()) } else { Break(()) }
    }) == Continue(())
}

```

请注意，这里无法用普通的 `fold` 来写，`all` 的语义承诺了一旦 `predicate` 返回 `false` 就停止从底层迭代器中消耗条目，但 `fold` 总是会消耗掉整个迭代器。

如果你正在实现自己的迭代器类型，就值得研究一下你的迭代器能否比 `Iterator` 特型的默认定义更高效地实现 `try_fold`。如果可以为 `try_fold` 提速，那么基于它构建的所有其他方法都会受益。

顾名思义，`try_rfold`（尝试右起折叠）方法与 `try_fold` 方法相同，只是从后面而不是前面开始提取值，并且要求传入一个双端迭代

器。

15.4.10 `nth` 与 `nth_back`

`nth`（第 n 个）方法会接受索引参数 n ，从迭代器中跳过 n 个条目，并返回下一个条目，如果序列提前结束了，则返回 `None`。调用 `.nth(0)` 等效于 `.next()`。

`nth` 不会像适配器那样接手迭代器的所有权，因此可以多次调用：

```
let mut squares = (0..10).map(|i| i*i);

assert_eq!(squares.nth(4), Some(16));
assert_eq!(squares.nth(0), Some(25));
assert_eq!(squares.nth(6), None);
```

它的签名如下所示：

```
fn nth(&mut self, n: usize) -> Option<Self::Item>
    where Self: Sized;
```

`nth_back`（倒数第 n 个）方法与 `nth` 方法很像，只是从双端迭代器的后面往前提取。调用 `.nth_back(0)` 等效于 `.next_back()`：返回最后一个条目；如果迭代器为空则返回 `None`。

15.4.11 `last`

`last`（最后一个）方法会返回迭代器生成的最后一个条目，如果为空则返回 `None`。它的签名如下所示：

```
fn last(self) -> Option<Self::Item>;
```

例如：

```
let squares = (0..10).map(|i| i*i);
assert_eq!(squares.last(), Some(81));
```

会从前面开始消耗所有迭代器的条目，即便此迭代器是可逆的也会如此。如果你有一个可逆迭代器并且不想消耗它的所有条目，应该只写 `iter.next_back()`。

15.4.12 `find`、`rfind` 和 `find_map`

`find`（查找）方法会从迭代器中提取条目，返回第一个由给定闭包回复 `true` 的条目，如果序列在找到合适的条目之前就结束了则返回 `None`。它的签名如下所示：

```
fn find<P>(&mut self, predicate: P) -> Option<Self::Item>
    where Self: Sized,
          P: FnMut(&Self::Item) -> bool;
```

`rfind`（右起查找）方法与此类似，但要求迭代器为双端迭代器并从后往前搜索值，返回最后一个给定闭包回复为 `true` 的条目。

例如，使用 15.4.4 节的城市和人口表，你可以这样写：

```
assert_eq!(populations.iter().find(|&(_name, &pop)| pop >
1_000_000),
    None);
assert_eq!(populations.iter().find(|&(_name, &pop)| pop >
500_000),
    Some((&"Portland", &583_776)));
```

这个表中没有任何城市的人口超过一百万，但有一个城市有五十万人口。

有时候，闭包不仅仅是一个简单的谓词——对每个条目进行逻辑判断并继续向前，它还可能是更复杂的东西，比如生成一个有意义的值。在这种情况下就需要使用 `find_map`（查找并映射），它的签名如下所示：

```
fn find_map<B, F>(&mut self, f: F) -> Option<B> where
    F: FnMut(Self::Item) -> Option<B>;
```

`find_map` 和 `find` 很像，但其闭包不会返回 `bool`，而是返回某个值的 `Option`。`find_map` 会返回第一个类型为 `Some` 的 `Option`。

假设有个城市内部公园的数据库，而我们要查看其中的公园是否有火山，有的话就给出公园的名称。

```
let big_city_with_volcano_park = populations.iter()
    .find_map(|(&city, _)| {
        if let Some(park) = find_volcano_park(city, &parks) {
            // find_map会返回下面的值，以便让调用者知道我们找到了哪个公园
            return Some((city, park.name));
        }

        // 拒绝此条目，并继续搜索
        None
    });

assert_eq!(big_city_with_volcano_park,
    Some(("Portland", "Mt. Tabor Park"));
```

15.4.13 构建集合：`collect` 与 `FromIterator`

本书一直在用 `collect`（收集）方法构建包含迭代器条目的向量，比如，在第 2 章中，我们曾调用 `std::env::args()` 获取程序命令行参数的迭代器，然后调用该迭代器的 `collect` 方法把它们收集到一个向量中：

```
let args: Vec<String> = std::env::args().collect();
```

但 `collect` 并不是向量专用的，事实上，它可以构建出 Rust 标准库中任意类型的集合，只要迭代器能生成合适的条目类型即可：

```
use std::collections::{HashSet, BTreeSet, LinkedList, HashMap,
    BTreeMap};
```

```
let args: HashSet<String> = std::env::args().collect();
let args: BTreeSet<String> = std::env::args().collect();
let args: LinkedList<String> = std::env::args().collect();

// 只有键-值对才能收集到Map中，因此对于这个例子，
// 要把字符串序列和整数序列拉合在一起
let args: HashMap<String, usize> =
std::env::args().zip(0..).collect();
let args: BTreeMap<String, usize> =
std::env::args().zip(0..).collect();

// 其他代码略
```

当然，`collect` 本身并不知道如何构造出所有这些类型。相反，如果某些集合类型（如 `Vec` 或 `HashMap`）知道如何从迭代器构造自身，就会自行实现 `std::iter::FromIterator`（来自迭代器）特型，而 `collect` 只是一个便捷的浅层包装而已：

```
trait FromIterator<A>: Sized {
    fn from_iter<T: IntoIterator<Item=A>>(iter: T) -> Self;
}
```

如果一个集合类型实现了 `FromIterator<A>`，那么它的类型关联函数 `from_iter` 就能从一个可迭代者生成的 `A` 类型条目中构建出一个该类型的值。

在最简单的情况下，实现代码可以构造出一个空集合，然后将迭代器中的条目一个个添加进去。例如，`std::collections::LinkedList` 的 `FromIterator` 就是这样实现的。

不过，也有一些类型可以用更好的方式实现，比如，要从某个迭代器 `iter` 构造出一个向量可以非常简单。

```
let mut vec = Vec::new();
for item in iter {
    vec.push(item)
}
vec
```

但这样做的效果并不理想：随着向量的增长，它可能要扩展其缓冲区，进而需要调用堆分配器并复制现有元素。向量固然进行了算法优化来保持这种开销尽可能低，但是如果有某种方法可以直接分配一个正确大小的缓冲区作为起点，那就根本没必要调整大小了。

这是 `Iterator` 特型的 `size_hint` 方法的用武之地：

```
trait Iterator {  
    ...  
    fn size_hint(&self) -> (usize, Option<usize>) {  
        (0, None)  
    }  
}
```

`size_hint` 方法会返回迭代器要生成的条目数的下限与可选上限，默认返回 0 作为下限而没有指定上限，这实际上就是在说“我也不知道上限”。许多迭代器可以做得更好，比如，`Range` 上的迭代器肯定知道它将生成多少个条目，`Vec` 或 `HashMap` 上的迭代器也知道。这类迭代器为 `size_hint` 提供了自己的专有定义。

`Vec` 的 `FromIterator` 实现如果想在一开始就正确设置新向量的缓冲区大小，那么这些上下限信息正是它所需要的。但在插入条目时仍然会检查缓冲区是否足够大，因此即使这种提示不正确，也只会影响性能，而不会影响安全。其他类型也可以采取类似的步骤，比如，`HashSet` 和 `HashMap` 就会使用 `Iterator::size_hint` 来为其哈希表选择合适的初始大小。

关于类型推断的一点说明：在本节的一开始，你曾看到同样是调用 `std::env::args().collect()`，却根据上下文生成了 4 种类型的集合，这可能有点儿奇怪。`collect` 的返回类型就是它的类型参数，所以前两个调用等价于以下代码：

```
let args = std::env::args().collect::<Vec<String>>();  
let args = std::env::args().collect::<HashSet<String>>();
```

但如果只有一种类型可以作为 `collect` 的参数，`Rust` 的类型推断就会为你提供这种类型。当你明确写出 `args` 的类型时，也足以说明

只有一种类型了。

15.4.14 **Extend** 特型

如果一个类型实现了 `std::iter::Extend`（扩展）特型，那么它的 `extend` 方法就能将一些可迭代的条目添加到集合中：

```
let mut v: Vec<i32> = (0..5).map(|i| 1 << i).collect();
v.extend([31, 57, 99, 163]);
assert_eq!(v, [1, 2, 4, 8, 16, 31, 57, 99, 163]);
```

所有的标准集合都实现了 `Extend`，因此它们都有 `extend` 方法，`String` 也实现了，但具有固定长度的数组和切片则未实现。

`Extend` 特型的定义如下所示：

```
trait Extend<A> {
    fn extend<T>(&mut self, iter: T)
        where T: IntoIterator<Item=A>;
}
```

显然，这与 `std::iter::FromIterator` 非常相似：后者会创建新集合，而 `Extend` 会扩展现有集合。事实上，标准库中 `FromIterator` 的好几个实现都只是简单地创建一个新的空集合，然后调用 `extend` 填充它，比如 `std::collections::LinkedList` 的 `FromIterator` 就是这样实现的。

```
impl<T> FromIterator<T> for LinkedList<T> {
    fn from_iter<I: IntoIterator<Item = T>>(iter: I) -> Self {
        let mut list = Self::new();
        list.extend(iter);
        list
    }
}
```

15.4.15 **partition**

`partition`（分区）方法会将迭代器的条目划分到两个集合中，并使用闭包来决定每个条目归属的位置：

```
let things = ["doorknob", "mushroom", "noodle", "giraffe",
"grapefruit"];

// 惊人的事实：在这个列表里生物的名字都是以奇数序的字母开头的
let (living, nonliving): (Vec<&str>, Vec<&str>)
    = things.iter().partition(|name| name.as_bytes()[0] & 1 != 0);

assert_eq!(living,      vec!["mushroom", "giraffe", "grapefruit"]);
assert_eq!(nonliving,   vec!["doorknob", "noodle"]);
```

与 `collect` 一样，`partition` 也可以创建你喜欢的任意种类的集合，但这些集合必须属于同一个类型。和 `collect` 一样，你需要指定返回类型：前面的示例明确写出了 `living` 和 `nonliving` 的类型，并让类型推断为调用 `partition` 选择正确的类型参数。

`partition` 的签名如下所示：

```
fn partition<B, F>(self, f: F) -> (B, B)
    where Self: Sized,
           B: Default + Extend<Self::Item>,
           F: FnMut(&Self::Item) -> bool;
```

`collect` 要求其结果类型实现了 `FromIterator`，而 `partition` 则要求其结果类型实现了 `std::default::Default`，因为所有 Rust 集合都实现了 `std::default::Default`，而 `std::default::Extend` 则用于将元素添加到集合中。

其他语言提供的 `partition` 操作往往只是将迭代器拆分为两个迭代器，而不会构建两个集合——这种处理方法之所以在 Rust 中并非好的选择，是因为那些已从底层迭代器中提取但尚未从已分区迭代器中提取的条目需要在某处进行缓冲，毕竟如果无论如何都要在内部构建某种集合，那还不如直接返回这些集合本身。

15.4.16 `for_each` 与 `try_for_each`

`for_each`（对每一个）方法会简单地对每个条目调用某个闭包：

```
["doves", "hens", "birds"].iter()
  .zip(["turtle", "french", "calling"])
  .zip(2..5)
  .rev()
  .map(|((item, kind), quantity)| {
    format!("{}", quantity, kind, item)
  })
  .for_each(|gift| {
    println!("You have received: {}", gift);
  });
```

输出如下所示：

```
You have received: 4 calling birds
You have received: 3 french hens
You have received: 2 turtle doves
```

这与简单的 `for` 循环非常相似，你同样可以在其中使用像 `break` 和 `continue` 这样的控制结构，但下面这样的适配器长链条调用在 `for` 循环中会显得有点儿笨拙：

```
for gift in ["doves", "hens", "birds"].iter()
  .zip(["turtle", "french", "calling"])
  .zip(2..5)
  .rev()
  .map(|((item, kind), quantity)| {
    format!("{}", quantity, kind, item)
  })
{
  println!("You have received: {}", gift);
}
```

要绑定的模式 `gift` 最终可能离使用它的循环体很远。

如果你的闭包需要容错或提前退出，可以使用 `try_for_each`（尝试对每一个）。

```
...
  .try_for_each(|gift| {
```

```
        writeln!(&mut output_file, "You have received: {}", gift)
    })?;
```

15.5 实现自己的迭代器

你可以为自己的类型实现 `IntoIterator` 特型和 `Iterator` 类型，令本章中展示的所有适配器和消费者都可以使用它，甚至包括许多针对标准迭代器接口编写的库和 `crate` 代码。在本节中，我们将展示两个迭代器：一个简单的迭代器，可以遍历范围类型；一个相对复杂些的迭代器，可以遍历二叉树类型。

假设我们有以下范围类型（从标准库的 `std::ops::Range<T>` 类型简化而来）。

```
struct I32Range {
    start: i32,
    end: i32
}
```

要想迭代 `I32Range` 就需要两个状态：当前值和迭代应该结束的界限。这恰好非常适合 `I32Range` 类型本身，使用 `start` 作为下一个值，并使用 `end` 作为界限。因此，你可以像这样实现 `Iterator`：

```
impl Iterator for I32Range {
    type Item = i32;
    fn next(&mut self) -> Option<i32> {
        if self.start >= self.end {
            return None;
        }
        let result = Some(self.start);
        self.start += 1;
        result
    }
}
```

此迭代器会生成 `i32` 型条目，也就是其 `Item` 类型。如果迭代已完成，则 `next` 会返回 `None`，否则，它会生成下一个值并更新其状态以便为下一次调用做准备。

当然，`for` 循环会使用 `IntoIterator::into_iter` 将其操作数转换为迭代器。但是标准库为每个实现了 `Iterator` 的类型都提供了

IntoIterator 的通用实现，所以 I32Range 可以直接使用：

```
let mut pi = 0.0;
let mut numerator = 1.0;

for k in (I32Range { start: 0, end: 14 }) {
    pi += numerator / (2*k + 1) as f64;
    numerator /= -3.0;
}
pi *= f64::sqrt(12.0);

// IEEE 754精确定义了此结果
assert_eq!(pi as f32, std::f32::consts::PI);
```

但 I32Range 是一个特例，因为其迭代目标和迭代器的类型是一样的。但还有许多情况没这么简单，比如，下面是第 10 章中的二叉树类型：

```
enum BinaryTree<T> {
    Empty,
    NonEmpty(Box<TreeNode<T>>)
}

struct TreeNode<T> {
    element: T,
    left: BinaryTree<T>,
    right: BinaryTree<T>
}
```

遍历二叉树的经典方式是递归，使用函数调用栈来跟踪你当前在树中的位置以及尚未访问的节点。但是当为 BinaryTree<T> 实现 Iterator 时，对 next 的每次调用都必须生成一个值并返回。为了跟踪尚未生成的树节点，迭代器必须维护自己的栈。下面是 BinaryTree 的一种可能的迭代器类型：

```
use self::BinaryTree::*;

// `BinaryTree` 的有序遍历状态
struct TreeIter<'a, T> {
    // 跟踪树节点引用的栈。由于我们使用了 `Vec` 的 `push` 方法
    // 和 `pop` 方法，因此栈顶就是向量的末尾
    //
```

```
// 迭代器接下来要访问的节点位于栈顶，栈顶之下的那些祖先
// 仍未访问。如果栈已空，则本迭代结束
unvisited: Vec<&'a TreeNode<T>>
}
```

创建了一个新的 `TreeIter` 时，它的初始状态应该是即将生成的树中最左侧的叶节点。根据 `unvisited` 栈的规则，它应该让那个叶节点位于栈顶，然后是栈顶节点未访问的祖先：位于树的左边缘的节点。我们可以通过从根到叶遍历树的左边缘并推入我们遇到的每个节点来初始化 `unvisited`（未访问的栈），因此我们将在 `TreeIter` 上定义一个方法来执行此操作：

```
impl<'a, T: 'a> TreeIter<'a, T> {
    fn push_left_edge(&mut self, mut tree: &'a BinaryTree<T>) {
        while let NonEmpty(ref node) = *tree {
            self.unvisited.push(node);
            tree = &node.left;
        }
    }
}
```

写成 `mut tree` 能让代码中的循环在沿着左边缘向下移动时更改 `tree` 指向的节点，但是由于 `tree` 是共享引用，它不能改变节点本身。

有了这个辅助方法，就可以给 `BinaryTree` 提供一个 `iter` 方法，让它返回遍历树的迭代器了：

```
impl<T> BinaryTree<T> {
    fn iter(&self) -> TreeIter<T> {
        let mut iter = TreeIter { unvisited: Vec::new() };
        iter.push_left_edge(self);
        iter
    }
}
```

`iter` 方法会构造一个带有空 `unvisited` 栈的 `TreeIter`，然后调用 `push_left_edge` 进行初始化。根据 `unvisited` 栈规则的要求，最左边的节点位于栈顶。

遵循标准库的做法，也可以通过调用 `BinaryTree::iter` 在对树的共享引用上实现 `IntoIterator`：

```
impl<'a, T: 'a> IntoIterator for &'a BinaryTree<T> {
    type Item = &'a T;
    type IntoIter = TreeIter<'a, T>;
    fn into_iter(self) -> Self::IntoIter {
        self.iter()
    }
}
```

这个 `IntoIter` 定义将 `TreeIter` 定义为针对 `&BinaryTree` 的迭代器类型。

最后，在 `Iterator` 实现中，我们开始实际遍历树。与 `BinaryTree` 的 `iter` 方法一样，迭代器的 `next` 方法也遵循栈规则：

```
impl<'a, T> Iterator for TreeIter<'a, T> {
    type Item = &'a T;
    fn next(&mut self) -> Option<&'a T> {
        // 找到此迭代必须生成的节点，或完成迭代（如果为`None`，
        // 请使用`?`运算符立即返回）
        let node = self.unvisited.pop()?;

        // 在`node`之后，我们接下来生成的条目必须是`node`右子树中最左边
        // 的子节点，所以要从这里向下推进。这个辅助方法恰好是我们想要的
        self.push_left_edge(&node.right);

        // 生成一个对本节点值的引用
        Some(&node.element)
    }
}
```

如果栈为空，则迭代完成。否则，`node` 就是对当前要访问的节点的引用，此调用将返回一个指向其 `element` 字段的引用。但首先，我们必须将迭代器的状态推进到下一个节点。如果这个节点具有右子树，则下一个要访问的节点是该子树最左侧的节点，我们可以使用 `push_left_edge` 将它及其未访问的祖先压入栈。但是如果这个节点没有右子树，则 `push_left_edge` 没有任何效果，这正是我们想

要的：我们希望新的栈顶是 `node` 的第一个未访问的祖先（如果有的话）。

有了 `IntoIterator` 和 `Iterator` 这两个实现，我们终于可以使用 `for` 循环来通过引用迭代 `BinaryTree` 了。使用 10.2.9 节中 `BinaryTree` 的 `add` 方法：

```
// 构建一棵小树
let mut tree = BinaryTree::Empty;
tree.add("jaeger");
tree.add("robot");
tree.add("droid");
tree.add("mecha");

// 对它进行遍历
let mut v = Vec::new();
for kind in &tree {
    v.push(*kind);
}
assert_eq!(v, ["droid", "jaeger", "mecha", "robot"]);
```

图 15-1 展示了当我们遍历示例树时 `unvisited` 栈的行为方式。对于每一步，下一个要访问的节点都会位于栈顶，所有未访问的祖先则位于其下方。

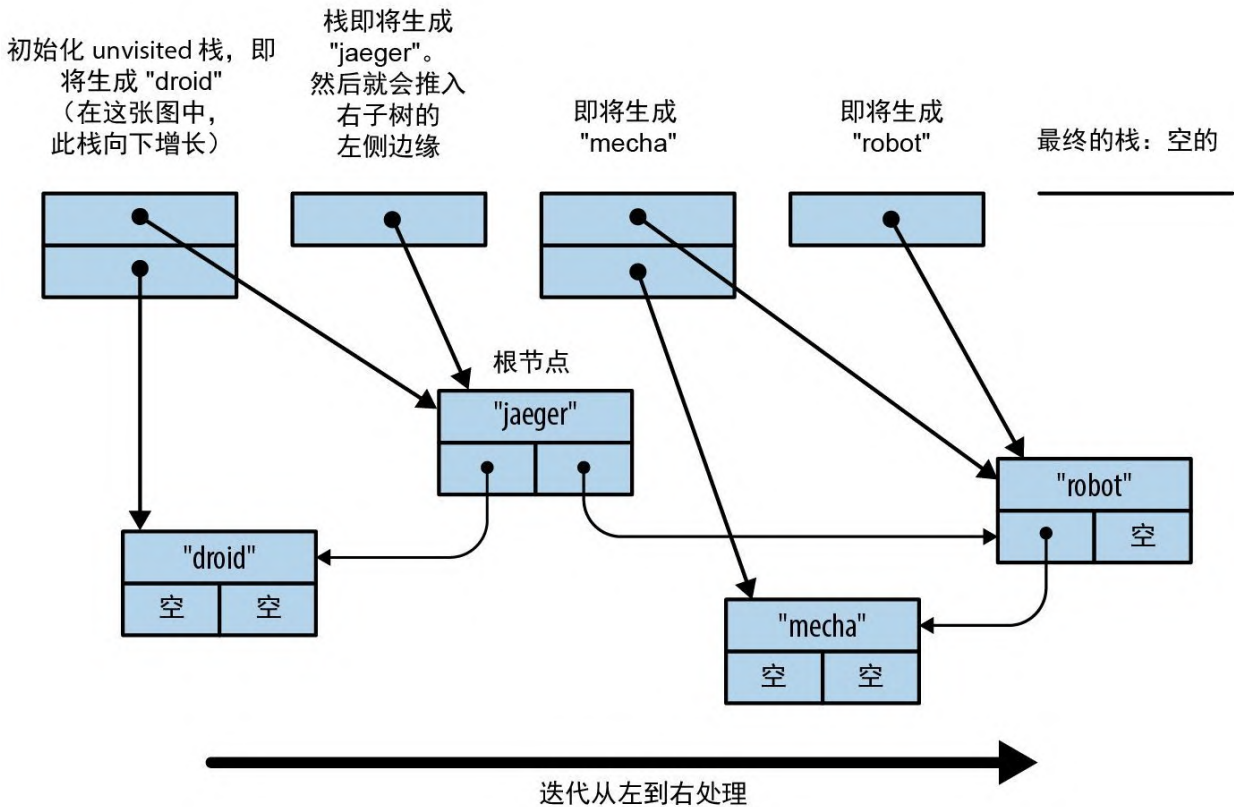


图 15-1：迭代遍历二叉树

所有常用的迭代器适配器和消费者都能用在我们这棵树上：

```
assert_eq!(tree.iter()
    .map(|name| format!("mega-{}", name))
    .collect:::<Vec<_>>(),
    vec!["mega-droid", "mega-jaeger",
        "mega-mecha", "mega-robot"]);
```

各种迭代器正是 Rust 哲学的体现，即提供强大的、零成本的抽象，提升代码的表现力和可读性。迭代器并没有完全取代循环，但确实提供了一个功能强大的基础构件，天然支持惰性求值特性且具有出色的性能。

第 16 章 集合

我们都像物理学假想中的麦克斯韦妖一样活动。正是在日常经验中，我们可以发现一向冷静的物理学家在两个世纪里对这个卡通形象一直难以忘怀的原因。生物体（organism），顾名思义，时刻在组织（organize）。我们分拣邮件、堆造沙堡、拼凑拼图、复盘棋局、收集邮票、给麦穗脱粒、按字母表顺序排列图书、创造对称形式、创作十四行诗和奏鸣曲，以及整理自己的房间。所有这些活动并不需要巨大的能量，只需保障我们能够发挥智能便可。

——James Gleick, 《信息简史》¹

¹ 此书已由人民邮电出版社出版，详见 <http://ituring.cn/book/731>。——编者注

Rust 标准库包含多个集合，这些集合是泛型类型，用于在内存中存储各种数据。在本书中，我们已经用到了一些集合，比如 `Vec` 和 `HashMap`。本章将详细介绍这两种类型的方法，以及另外 6 个标准集合。在开始之前，我们先来辨析一下 Rust 的集合与其他语言的集合之间的一些系统性差异。

首先，移动和借用无处不在。Rust 使用移动来避免对值做深拷贝。这就是 `Vec<T>::push(item)` 方法会按值而非按引用来获取参数的原因。这样值就会移动到向量中。第 4 章中的示意图展示了这在实践中是如何实现的：将 Rust `String` 压入 `Vec<String>` 中会很快，因为 Rust 不必复制字符串的字符数据，并且字符串的所有权始终是明晰的。

其次，Rust 没有失效型错误，也就是当程序仍持有指向集合内部数据的指针时，集合被重新调整大小或发生其他变化而导致的那种悬空指针错误。失效型错误是 C++ 中未定义行为的另一个来源，即使在内存安全的语言中，它们也会偶尔导致

`ConcurrentModificationException`²。Rust 的借用检查器在编译期就可以排除这些错误。

²这是 Java 中的异常类。——译者注

最后，Rust 没有 `null`，因此在其他语言使用 `null` 的地方 Rust 会使用 `Option`。

除了这些差异，Rust 的集合与你预期的差不多。如果你是一位经验丰富的程序员，而且时间有限，那么可以快速浏览本章，但不要错过 16.5.1 节。

16.1 概述

表 16-1 展示了 Rust 的 8 个标准集合，它们都是泛型类型。

表 16-1：标准集合汇总表

集合	描述	其他语言中类似的集合类型		
		C++	Java	Python
<code>Vec<T></code>	可增长数组	<code>vector</code>	<code>ArrayList</code>	<code>list</code>
<code>VecDeque<T></code>	双端队列（可增长的环形缓冲区）	<code>deque</code>	<code>ArrayDeque</code>	<code>collections.deque</code>
<code>LinkedList<T></code>	双向链表	<code>list</code>	<code>LinkedList</code>	—
<code>BinaryHeap<T></code> <code>where T: Ord</code>	最大堆	<code>priority_queue</code>	<code>PriorityQueue</code>	<code>heapq</code>
<code>HashMap<K, V></code> <code>where K: Eq + Hash</code>	键值哈希表	<code>unordered_map</code>	<code>HashMap</code>	<code>dict</code>
<code>BTreeMap<K, V></code> <code>where K: Ord</code>	有序键值表	<code>map</code>	<code>TreeMap</code>	—
<code>HashSet<T></code> <code>where T: Eq + Hash</code>	无序的、基于哈希的集	<code>unordered_set</code>	<code>HashSet</code>	<code>set</code>
<code>BTreeSet<T></code> <code>where T: Ord</code>	有序集	<code>set</code>	<code>TreeSet</code>	—

`Vec<T>`、`HashMap<K, V>` 和 `HashSet<T>` 是最常用的集合类型，其余的都各自有其基本应用场景。本章会依次讨论每种集合类型。

`Vec<T>`（普通向量）

可增长的、分配在堆上的 `T` 类型值数组。本章会用大约一半的篇幅专门介绍 `Vec` 及其众多实用方法。

`VecDeque<T>`（双端队列向量）

与 `Vec<T>` 类似，但更适合用作先入先出队列。它支持在列表的前面和后面高效地添加值和移除值，但代价是会让所有其他的操作都稍微变慢一些。

`BinaryHeap<T>`（二叉堆）

优先级队列。`BinaryHeap` 中的值是精心组织过的，因此始终可以高效地查找和移除其最大值。

`HashMap<K, V>`（哈希 Map）

由键-值对构成的表。通过键查找值很快。其条目会以任意顺序存储。

`BTreeMap<K, V>`（B 树 Map）

与 `HashMap<K, V>` 类似，但它会根据键来对条目进行排序。`BTreeMap<String, i32>` 会以 `String` 的比较顺序来存储其条目。除非需要让条目保持排序状态，否则用 `HashMap` 更快一些。

`HashSet<T>`（哈希 Set）

由 `T` 类型的值组成的 `Set`。它既能很快地添加值和移除值，也能很快地查询给定值是否在此 `Set` 中。

`BTreeSet<T>` (B 树 Set)

与 `HashSet<T>` 类似，但它会让元素按值排序。同样，除非需要让数据保持排序状态，否则用 `HashSet` 更快一些。

因为 `LinkedList` 很少使用（对于大多数用例，在性能和接口方面有更好的替代方案），所以这里就不展开讲解了。

16.2 Vec<T>

因为本书中一直在使用 `Vec`，所以我们假设你对它已经比较熟悉了。有关介绍，请参阅 3.6.2 节。下面我们将详细讲解 `Vec` 的方法及内部工作原理。

创建向量的最简单方法是使用 `vec!` 宏：

```
// 创建一个空向量
let mut numbers: Vec<i32> = vec![];

// 使用给定内容创建一个向量
let words = vec!["step", "on", "no", "pets"];
let mut buffer = vec![0u8; 1024]; // 1024个内容为0的字节
```

如第 4 章所述，向量具有 3 个字段：长度、容量和指向用于存储元素的堆分配内存的指针。图 16-1 展示了前面的向量在内存中的布局方式。空向量 `numbers` 最初的容量为 0。直到添加第一个元素之前，不会为其分配堆内存。

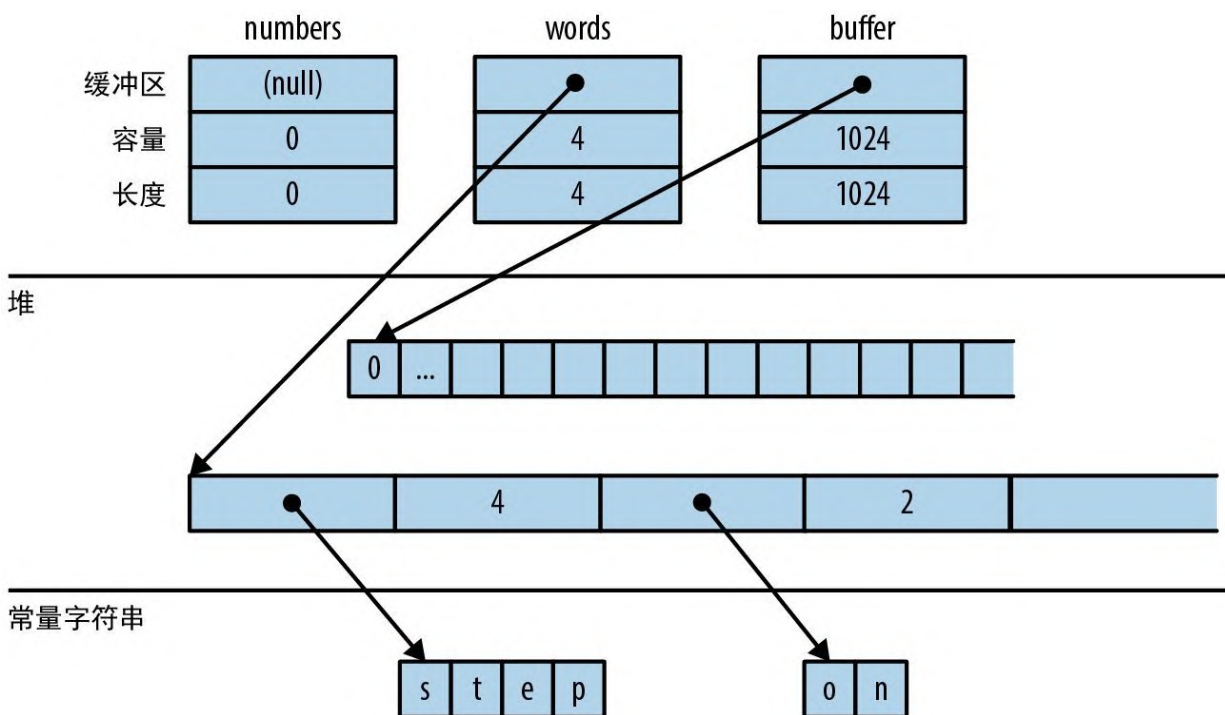


图 16-1：向量的内存布局：**words** 的每个元素都是一个由指针和长度组成的 **&str** 值

与所有集合一样，`Vec` 也实现了 `std::iter::FromIterator`，所以可以使用迭代器的 `.collect()` 方法从任意迭代器创建向量，详情请参阅 15.4.13 节。

```
// 把另一个集合转换成向量
let my_vec = my_set.into_iter().collect::<Vec<String>>();
```

16.2.1 访问元素

通过索引来获取数组、切片或向量的元素非常简单：

```
// 获取某个元素的引用
let first_line = &lines[0];

// 获取某个元素的副本
let fifth_number = numbers[4];           // 要求实现了Copy特型
let second_line = lines[1].clone();      // 要求实现了Clone特型

// 获取切片的引用
let my_ref = &buffer[4..12];

// 获取切片的副本
let my_copy = buffer[4..12].to_vec();    // 要求实现了Clone特型
```

如果索引超出了范围，则所有这些形式都会引发 `panic`。

Rust 对数值类型很挑剔，对向量也不例外。向量的长度和索引都是 `usize` 类型。试图用 `u32`、`u64` 或 `isize` 作为向量索引会导致出错。可以根据需要使用 `as usize` 来转换，详情请参阅 6.14 节。

下面这些方法可以轻松访问向量或切片的特定元素（请注意，所有的切片方法也都适用于数组和向量）。

`slice.first()`（第一个）

返回对 `slice` 的第一个元素的引用（如果有的话）。

返回类型为 `Option<&T>`，所以如果 `slice` 为空则返回值为 `None`，如果不为空则返回 `Some(&slice[0])`。

```
if let Some(item) = v.first() {  
    println!("We got one! {}", item);  
}
```

`slice.last()`（最后一个）

与 `first` 类似，但会返回对最后一个元素的引用。

`slice.get(index)`（获取）

如果其存在，就返回 `slice[index]` 引用的 `Some` 值。如果 `slice` 的元素少于 `index+1` 个，则返回 `None`。

```
let slice = [0, 1, 2, 3];  
assert_eq!(slice.get(2), Some(&2));  
assert_eq!(slice.get(4), None);
```

`slice.first_mut()`（第一个可变）、`slice.last_mut()`（最后一个可变）和 `slice.get_mut(index)`（获取可变）

这些方法是前述 `slice.first()` 等方法的变体，但借入的是可变引用。

```
let mut slice = [0, 1, 2, 3];  
{  
    let last = slice.last_mut().unwrap(); // last的类型是&mut i32  
    assert_eq!(*last, 3);  
    *last = 100;  
}  
assert_eq!(slice, [0, 1, 2, 100]);
```

因为按值返回 `T` 就意味着移动它，所以一些需要就地访问元素的方法通常会按引用返回这些元素。

`.to_vec()` 方法是一个例外，它会复制这些元素。

`slice.to_vec()`（转向量）

克隆整个切片，返回一个新向量：

```
let v = [1, 2, 3, 4, 5, 6, 7, 8, 9];
assert_eq!(v.to_vec(),
           vec![1, 2, 3, 4, 5, 6, 7, 8, 9]);
assert_eq!(v[0..6].to_vec(),
           vec![1, 2, 3, 4, 5, 6]);
```

此方法只能用在元素可以克隆的情况下，也就是需满足 `where T: Clone` 限界。

16.2.2 迭代

向量、数组和切片是可迭代的，要么按值迭代³，要么按引用迭代，但都要遵循 15.2.2 节描述的模式。

³ 作者在这里只是泛泛而言，其实除非元素是 `Copy` 类型，否则切片并不能按值迭代（情形 1）。——译者注

- 遍历 `Vec<T>` 或数组 `[T; N]` 会生成 `T` 类型的条目。这些元素会逐个从向量或数组中移动出来并被消耗掉。
- 遍历 `&[T; N]`、`&[T]` 或 `&Vec<T>` 类型的值（对数组、切片或向量的引用）会生成 `&T` 类型的条目，即对单个元素的引用，这些元素不会移动出来。
- 遍历 `&mut [T; N]`、`&mut [T]` 或 `&mut Vec<T>` 类型的值会生成 `&mut T` 类型的条目。

数组、切片和向量也有 `.iter()` 方法和 `.iter_mut()` 方法（参见 15.2.1 节），以用于创建一个会生成对其元素的引用的迭代器。

稍后本章将在 16.2.5 节中介绍一些更高级的方法来迭代切片。

16.2.3 扩大向量与收缩向量

数组、切片或向量的长度是它们所包含的元素数量。

`slice.len()` (长度)

返回 `slice` 的长度，类型为 `usize`。

`slice.is_empty()` (为空?)

如果 `slice` 未包含任何元素 (`slice.len() == 0`) 则为真。

本节中的其余方法是关于扩大向量和收缩向量的。但数组和切片中没有这些方法，因为数组和切片一旦创建就无法调整大小。

向量的所有元素都存储在连续的、分配在堆上的内存块中。向量的容量就是该内存块可以容纳的最大元素数量。`Vec` 通常会替你管理容量，当需要更多空间时它会自动分配更大的缓冲区并将元素移入其中。下面是一些显式管理容量的方法。

`Vec::with_capacity(n)` (自带容量)

创建一个容量为 `n` 的新的空向量。

`vec.capacity()` (取容量)

返回 `vec` 的容量，类型为 `usize`。`vec.capacity() >= vec.len()` 始终是成立的。

`vec.reserve(n)` (预留)

确保向量至少有足够的备用容量来容纳另外 `n` 个元素，也就是说，`vec.capacity()` 至少等于 `vec.len() + n`。如果已经有足够的空间，就什么也不做。如果没有，则会分配一个更大的缓冲区并将向量的内容移入其中。

`vec.reserve_exact(n)` (精确预留)

与 `vec.reserve(n)` 类似，但要求 `vec` 不要为未来的增长分配任何多于 `n` 的额外容量。调用此方法后，`vec.capacity()` 应该精确等于 `vec.len() + n`。

`vec.shrink_to_fit()`（缩小到刚好够）

如果 `vec.capacity()` 大于 `vec.len()`，则尝试释放额外的内存。

`Vec<T>` 还有许多用来添加或移除元素的方法，它们可以改变向量的长度。所有这些方法都可以通过可变引用获取其 `self` 参数。

下面这两个方法会在向量的末尾添加或移除单个值。

`vec.push(value)`（推入）

将给定 `value` 添加到 `vec` 的末尾。

`vec.pop()`（弹出）

移除并返回最后一个元素。返回类型是 `Option<T>`。如果弹出的元素是 `x`，则返回 `Some(x)`；如果向量已经为空，则返回 `None`。

请注意，`.push()` 会按值而不是按引用接受其参数。同样，`.pop()` 会返回弹出的值，而不是引用。本节中剩下的大部分方法也是如此。它们可以将值移动进和移动出向量。

下面这两个方法会在向量的任意位置添加或移除一个值。

`vec.insert(index, value)`（插入）

在 `vec[index]` 处插入给定的 `value`，将 `vec[index..]` 中的所有当前值向右平移一个位置以腾出空间。

如果 `index > vec.len()`，则会引发 `panic`。

`vec.remove(index)`（移除）

移除并返回 `vec[index]`，将 `vec[index+1..]` 中的所有当前值向左平移一个位置以填补空白。

如果 `index >= vec.len()`，则会引发 `panic`，因为在这种情况下要移除的 `vec[index]` 元素并不存在。

向量越长，这个操作就越慢。如果需要经常执行 `vec.remove(0)`，请考虑使用 `VecDeque`（参见 16.3 节）来代替 `Vec`。

需要移动的元素越多，`.insert()` 和 `.remove()` 的速度就会越慢。

下面这 4 个方法可以把向量的长度更改为特定值。

`vec.resize(new_len, value)`（调整大小）

将 `vec` 的长度设置为 `new_len`。如果该操作会增加 `vec` 的长度，则以 `value` 的副本填补新空间。元素类型必须实现 `Clone` 类型。

`vec.resize_with(new_len, closure)`（以……调整大小）

与 `vec.resize` 类似，但会调用闭包来构造每个新元素。它可用于不可 `Clone` 的元素构成的向量。

`vec.truncate(new_len)`（截断）

将 `vec` 的长度减少到 `new_len`，丢弃 `vec[new_len..]` 范围内的任何元素。

如果 `vec.len()` 已经小于或等于 `new_len`，则什么也不会发生。

`vec.clear()`（清空）

从 `vec` 中移除所有元素。此方法的效果和 `vec.truncate(0)` 一样。

下面这 4 个方法可以一次添加或移除多个值。

`vec.extend(iterable)` (扩展)

按顺序在 `vec` 末尾添加来自给定 `iterable` 值的所有条目。此方法就像 `.push()` 的多值版本。`iterable` 参数可以是实现了 `IntoIterator<Item=T>` 的任何值。

此方法非常有用，所以我们为其定义了一个标准特型 `Extend`，所有标准集合都实现了该特型。不过很遗憾，这会导致 `rustdoc` 在其生成的 HTML 底部将 `.extend()` 与其他特型方法混排在一起，因此在需要时很难找到它。我也只能告诉你：请记住它在那里。有关详细信息，请参见 15.4.14 节。

`vec.split_off(index)` (拆分出)

与 `vec.truncate(index)` 类似，但此方法会返回一个 `Vec<T>`，其中包含从 `vec` 末尾移除的那些值。此方法就像是 `.pop()` 的多值版本。

`vec.append(&mut vec2)` (追加)

这会将 `vec2` 的所有元素移动到 `vec` 中，其中 `vec2` 是 `Vec<T>` 类型的另一个向量。之后，`vec2` 会被清空。

此方法与 `vec.extend(vec2)` 类似，不同之处在于调用 `extend` 之后 `vec2` 仍然存在，其容量也不受影响。

`vec.drain(range)` (抽取)

这将从 `vec` 中移除 `range` 范围内的切片 `vec[range]`，并返回对所移除元素的迭代器，其中 `range` 是范围值，类似 `..` 或 `0..4`。

还有一些略显古怪的方法可以从向量中选择性地移除一些元素。

`vec.retain(test)`（留下）

移除所有未通过给定测试的元素。`test` 参数是实现了 `FnMut(&T) -> bool` 的函数或闭包。针对 `vec` 的每个元素，此方法都会调用 `test(&element)`，如果函数或闭包返回了 `false`，就会从向量中移除并丢弃此元素。

除了性能上略有差异，此方法和下面的写法很像。

```
vec = vec.into_iter().filter(test).collect();
```

`vec.dedup()`（去重）

丢弃重复的元素，类似于 Unix shell 实用程序 `uniq`。此方法会扫描 `vec` 以查找彼此相等的相邻元素，然后会从这些相等值中保留一个并丢弃多余的值：

```
let mut byte_vec = b"Missssssssissippi".to_vec();
byte_vec.dedup();
assert_eq!(&byte_vec, b"Misisipi");
```

请注意，输出中仍然有两个 `'s'` 字符。这是因为此方法只会移除相邻的重复项。要想消除所有重复项，你有 3 个选择：在调用 `.dedup()` 之前对向量进行排序、将数据移动到一个 `Set` 中，或者（为了保持元素的原始顺序）使用如下 `.retain()` 技巧：

```
let mut byte_vec = b"Missssssssissippi".to_vec();

let mut seen = HashSet::new();
byte_vec.retain(|r| seen.insert(*r));

assert_eq!(&byte_vec, b"Misp");
```

上述代码的工作原理是当 `Set` 已经包含我们要插入的条目时 `.insert()` 就会返回 `false`。

`vec.dedup_by(same)`（根据 `same` 调用结果去重）

与 `vec.dedup()` 类似，但此方法会使用函数或闭包 `same(&mut elem1, &mut elem2)` 而不是 `==` 运算符来检查两个元素是否应被视为相等。

`vec.dedup_by_key(key)`（根据 `key` 属性去重）

与 `vec.dedup()` 类似，但此方法会在 `key(&mut elem1) == key(&mut elem2)` 时将两个元素视为相等。

如果 `errors` 是一个 `Vec<Box<dyn Error>>`，你可以这样写：

```
// 移除带有相同信息的多余错误 (error)
errors.dedup_by_key(|err| err.to_string());
```

在本节涵盖的所有方法中，只有 `.resize()` 会克隆值，其他方法都是将值从一个地方移动到另一个地方。

16.2.4 联结

以下两个方法可用于数组的数组，即其元素本身也是数组、切片或向量的数组、切片或向量。

`slices.concat()`（串联）

返回通过串联所有切片组装成的新向量。

```
assert_eq!([[1, 2], [3, 4], [5, 6]].concat(),
            vec![1, 2, 3, 4, 5, 6]);
```

`slices.join(&separator)`（联结）

与 `concat` 类似，只是在这些切片之间插入了值 `separator` 的副本。

```
assert_eq!([[1, 2], [3, 4], [5, 6]].join(&0),
           vec![1, 2, 0, 3, 4, 0, 5, 6]);
```

16.2.5 拆分

同时获得多个对数组、切片或向量中元素的不可变引用是比较容易的：

```
let v = vec![0, 1, 2, 3];
let a = &v[i];
let b = &v[j];

let mid = v.len() / 2;
let front_half = &v[..mid];
let back_half = &v[mid..];
```

但获取多个可变引用就不那么容易了：

```
let mut v = vec![0, 1, 2, 3];
let a = &mut v[i];
let b = &mut v[j]; // 错误：不能同时把`v`借入为多个可变引用

*a = 6;           // 这里用到了引用`a`和引用`b`，
*b = 7;           // 所以它们的生命周期必然重叠
```

Rust 禁止这样做，因为如果 `i == j`，那么 `a` 和 `b` 就是对同一个整数的两个可变引用，这违反了 Rust 的安全规则。（参见 5.4 节。）

Rust 有几种方法可以同时借入对数组、切片或向量的两个或多个部分的可变引用。与前面的代码不同，这些方法是安全的，因为根据设计，它们总会把数据拆分成几个不重叠的区域。这里的大部分方法在处理非 `mut` 切片时也很方便，因此每个方法都有 `mut` 版本和非 `mut` 版本。

图 16-2 展示了这些方法。

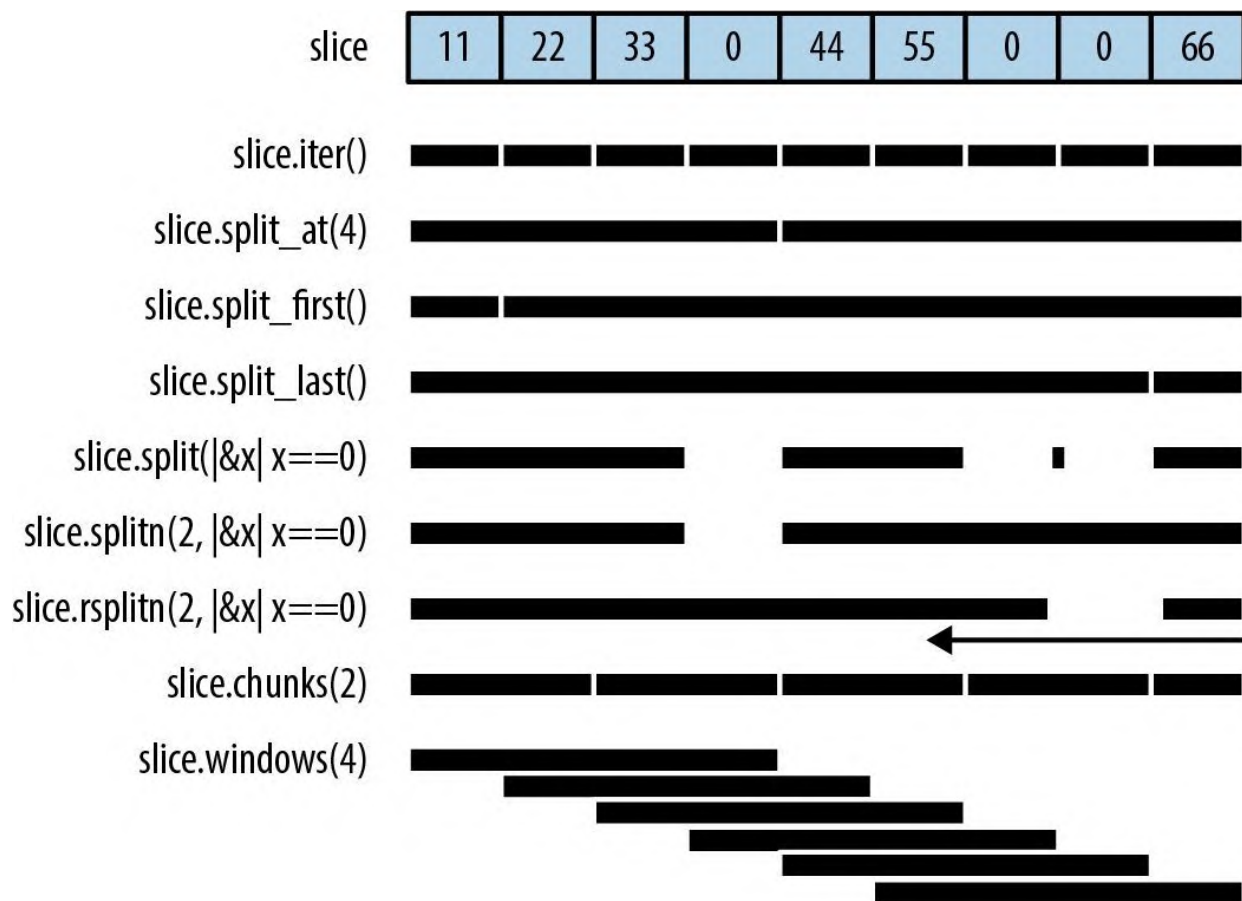


图 16-2: 对几个拆分型方法的说明（注意：**`slice.split(|&x| x==0)`** 输出中的小矩形是由于存在两个相邻的分隔符而生成的空切片，并且 **`rsplittn`** 会按从后向前的顺序生成其输出，这与另外几个方法不同）

这些方法都没有直接修改数组、切片或向量，它们只是返回了对内部数据中各部分的新引用。

`slice.iter()`（迭代器）和 `slice.iter_mut()`（可变迭代器）

生成对 `slice` 中每个元素的引用。16.2.2 节介绍过它们。

`slice.split_at(index)`（拆分子）和
`slice.split_at_mut(index)`（可变拆分子）

将一个切片分成两半，返回一个值对。
`slice.split_at(index)` 等价于 `(&slice[..index], &slice[index..])`。如果 `index` 超出了限界，这两个方法就会发生 panic。

`slice.split_first()`（拆分首个）和
`slice.split_first_mut()`（可变拆分首个）

同样会返回一个值对：对首个元素（`slice[0]`）的引用和对所有其余元素（`slice[1..]`）的切片的引用。

`.split_first()` 的返回类型是 `Option<(&T, &[T])>`，如果 `slice` 为空，则结果为 `None`。

`slice.split_last()`（拆分末尾）和
`slice.split_last_mut()`（可变拆分末尾）

与 `slice.split_first()` 和 `slice.split_first_mut()` 类似，但这两个方法拆分出的是最后一个元素而不是首个元素。

`.split_last()` 的返回类型是 `Option<(&T, &[T])>`。

`slice.split(is_sep)`（拆分）和
`slice.split_mut(is_sep)`（可变拆分）

将 `slice` 拆分为一个或多个子切片，使用函数或闭包 `is_sep` 确定拆分位置。这两个方法会返回一个遍历这些子切片的迭代器。

当你消耗此迭代器时，这些方法会为切片中的每个元素调用 `is_sep(&element)`。如果 `is_sep(&element)` 为 `true`，则认为该元素是分隔符。分隔符不会包含在输出的任何子切片中。

输出总是至少包含一个子切片，每遇到一个分隔符就额外加一个。如果有多个分隔符彼此相邻，或者有分隔符出现在 `slice` 的两端，则每对分隔符和两端的分隔符分别会对应输出一个空的子切片。

`slice.split_inclusive(is_sep)` (拆分, 含分隔符) 和
`slice.split_inclusive_mut(is_sep)` (可变拆分, 含分隔符)

与 `split` 和 `split_mut` 类似, 但这两个方法会在前一个子切片的结尾包含分隔符而不会排除它。

`slice.rsplit(is_sep)` (右起拆分) 和
`slice.rsplit_mut(is_sep)` (右起可变拆分)

与 `split` 和 `split_mut` 类似, 但这两个方法会从切片的末尾开始往前拆分。

`slice.splitn(n, is_sep)` (拆分为 n 片) 和
`slice.splitn_mut(n, is_sep)` (可变拆为 n 片)

与 `slice.rsplit(is_sep)` 和 `slice.rsplit_mut(is_sep)` 类似, 但这两个方法最多会生成 n 个子切片。在找到前 $n-1$ 个切片后, 不会再调用 `is_sep`。最后一个子切片中会包含剩下的所有元素。

`slice.rsplitn(n, is_sep)` (右起拆分为 n 片) 和
`slice.rsplitn_mut(n, is_sep)` (右起可变拆分为 n 片)

与 `.splitn()` 和 `.splitn_mut()` 类似, 但是在使用这两个方法时, 切片会以相反的顺序扫描。也就是说, 这两个方法会在切片中的最后而不是最前 $n-1$ 个分隔符上进行拆分, 并且子切片是从末尾开始向前生成的。

`slice.chunks(n)` (分为长度为 n 的块) 和
`slice.chunks_mut(n)` (分为长度为 n 的可变块)

返回长度为 n 的非重叠子切片上的迭代器。如果 n 不能被 `slice.len()` 整除, 则最后一个块包含的元素将不足 n 个。

`slice.rchunks(n)`（右起分为长度为 n 的块）和
`slice.rchunks_mut(n)`（右起分为长度为 n 的可变块）

与 `slice.chunks` 和 `slice.chunks_mut` 类似，但会从切片的末尾开始向前拆分。

`slice.chunks_exact(n)`（精确分为长度为 n 的块）和
`slice.chunks_exact_mut(n)`（精确分为长度为 n 的可变块）

返回长度为 n 的非重叠子切片上的迭代器。如果 n 不能被 `slice.len()` 整除，则最后一个块（少于 n 个元素）可以在其结果的 `remainder()` 方法中获取。

`slice.rchunks_exact(n)`（右起精确分为长度为 n 的块）和
`slice.rchunks_exact_mut(n)`（右起精确分为长度为 n 的可变块）

与 `slice.chunks_exact` 和 `slice.chunks_exact_mut` 类似，但会从切片的末尾开始拆分。

还有一个迭代子切片的方法。

`slice.windows(n)`（滑动窗口）

返回一个其行为类似于 `slice` 中数据的“滑动窗口”的迭代器。这个迭代器会生成一些横跨此 `slice` 中 n 个连续元素的子切片。它生成的第一个值是 `&slice[0..n]`，第二个值是 `&slice[1..n+1]`，以此类推。

如果 n 大于 `slice` 的长度，则不会生成任何切片。如果 n 为 0，则此方法会发生 `panic`。

如果 `days.len() == 31`，那么就可以通过调用 `days.windows(7)` 来生成 `days` 中所有相隔 7 天的时间段。

大小为 2 的滑动窗口可用于探究数据序列如何从一个数据点变化到下一个数据点：

```
let changes = daily_high_temperatures
    .windows(2)                // 获得两个相邻的最高气温
    .map(|w| w[1] - w[0])      // 气温变化了多少？
    .collect::<Vec<_>>();
```

因为各个子切片会重叠，所以此方法并没有返回可变引用的变体。

16.2.6 交换

下面是交换切片内容的一些便利方法。

`slice.swap(i, j)`（交换元素）

交换 `slice[i]` 和 `slice[j]` 这两个元素。

`slice_a.swap_with_slice(slice_b)`（互换内容）

交换 `slice_a` 和 `slice_b` 的全部内容。`slice_a` 和 `slice_b` 的长度必须相同。

向量有一个关联方法，该方法可以高效地移除任何元素。

`vec.swap_remove(i)`（交换后移除）

移除并返回 `vec[i]`。与 `vec.remove(i)` 类似，但此方法不会将向量中剩余的元素平移过来以填补空缺，而是简单地将 `vec` 的最后一个元素移动到空缺中。如果不关心向量中剩余条目的顺序，那么此方法会很有用，因为性能更高。

16.2.7 填充

下面是替换可变切片内容的两种便利方法。

`slice.fill(value)` (填充)

用 `value` 的克隆体填充切片。

`slice.fill_with(function)` (以 `function` 回调填充)

使用调用给定函数生成的值来填充切片。这对于实现了 `Default` 但未实现 `Clone` 的类型很有用，比如当 `T` 为不可复制类型时的 `Option<T>` 或 `Vec<T>`。

16.2.8 排序与搜索

下面是切片提供的 3 个排序方法。

`slice.sort()` (排序)

将元素按升序排列。此方法仅当元素类型实现了 `Ord` 时才存在。

`slice.sort_by(cmp)` (按 `cmp` 回调排序)

对 `slice` 中的元素按函数或闭包 `cmp` 指定的顺序进行排序。
`cmp` 必须实现 `Fn(&T, &T) -> std::cmp::Ordering`。

手动实现 `cmp` 是一件痛苦的事，不过可以把它委托给别的
`.cmp()` 方法来实现：

```
students.sort_by(|a, b| a.last_name.cmp(&b.last_name));
```

如果想按一个字段排序，但当该字段相同时按另一个字段判定先后，则可以先把它们做成元组然后再进行比较。

```
students.sort_by(|a, b| {  
    let a_key = (&a.last_name, &a.first_name);  
    let b_key = (&b.last_name, &b.first_name);  
    a_key.cmp(&b_key)  
});
```


`slice.sort_by_key(key)`（按 `key` 回调排序）

使用由函数或闭包型参数 `key` 给出的排序键对 `slice` 的元素按递增顺序排序。`key` 的类型必须实现 `Fn(&T) -> K`，这里要满足 `K: Ord`。

这在 `T` 包含一个或多个有序字段时会很有用，因此它可以按多种方式排序：

```
// 按平均学分绩点排序，低分在前
students.sort_by_key(|s| s.grade_point_average());
```

请注意，在排序过程中不会缓存这些排序键值，因此 `key` 函数可能会被调用 n 次以上。

出于技术原因，`key(element)` 无法返回从元素借来的任何引用。下面这种写法行不通：

```
students.sort_by_key(|s| &s.last_name); // 错误：无法推断生命周期
```

Rust 无法推算出生命周期。但在这些情况下，很容易把 `.sort_by()` 作为后备方案。

以上 3 个方法都会执行稳定排序。

要想以相反的顺序排序，可以将 `sort_by` 与交换了两个参数的 `cmp` 闭包一起使用。传入参数 `|b, a|` 而不是 `|a, b|` 可以有效地生成相反的顺序。或者，也可以在排序之后调用 `.reverse()` 方法。

`slice.reverse()`（逆转）

就地逆转切片。

一旦切片排序完毕，就可以高效地进行搜索了。

`slice.binary_search(&value)`（二分搜索）、
`slice.binary_search_by(&value, cmp)`（按 `cmp` 回调二

分搜索) 和 `slice.binary_search_by_key(&value, key)` (按 `key` 闭包二分搜索)

以上 3 个方法都会在给定的已排序 `slice` 中搜索 `value`。请注意, `value` 是按引用传递的。

这些方法的返回类型是 `Result<usize, usize>`。如果在指定排序顺序中 `slice[index]` 等于 `value`, 那么这些方法就会返回 `Ok(index)`。如果找不到这样一个索引, 那么这些方法就会返回 `Err(insertion_point)`, 这样当你在 `insertion_point` 中插入 `value` 后, 向量仍然会保持排好序的状态。

当然, 只有在切片确实已经按指定顺序排序时二分搜索才有效。否则, 结果将是没有意义的, 因为如果输入无效, 则输出也无效。

由于 `f32` 和 `f64` 具有 `NaN` 值, 因此它们无法实现 `Ord` 并且不能直接用作排序和二分搜索方法的键。要获得适用于浮点数据的类似方法, 请使用 `ord_subset crate`。

可以用另一个方法在未排过序的向量中搜索。

`slice.contains(&value)` (包含)

如果 `slice` 中有任何元素等于 `value`, 则返回 `true`。这会简单地检查切片的每个元素, 直到找到匹配项。`value` 同样是按引用传递的。

如果要在切片中查找值的位置 (类似 JavaScript 中的 `array.indexOf(value)`), 请使用迭代器:

```
slice.iter().position(|x| *x == value)
```

这将返回 `Option<usize>`。

16.2.9 比较切片

如果类型 `T` 支持 `==` 运算符和 `!=` 运算符 (`PartialEq` 特型, 参见 12.2 节), 那么数组 `[T; N]`、切片 `[T]` 和向量 `Vec<T>` 也会支持这两个运算符。如果两个切片的长度相同并且对应的元素也相等, 那它们就是相等的。数组和向量也是如此。

如果 `T` 支持运算符 `<`、`<=`、`>` 和 `>=` (`PartialOrd` 特型, 参见 12.3 节), 那么 `T` 的数组、切片和向量也会支持这些运算符。切片之间是按字典序比较的 (从左到右逐个比较)。

下面是执行常见的切片比较的两个便捷方法。

`slice.starts_with(other)` (以 `other` 开头)

如果 `slice` 的起始值序列等于 `other` 切片中的相应元素, 则返回 `true`。

```
assert_eq!([1, 2, 3, 4].starts_with(&[1, 2]), true);
assert_eq!([1, 2, 3, 4].starts_with(&[2, 3]), false);
```

`slice.ends_with(other)` (以 `other` 结尾)

与上一个方法类似, 但会检查 `slice` 的结尾值。

```
assert_eq!([1, 2, 3, 4].ends_with(&[3, 4]), true);
```

16.2.10 随机元素

随机数并未内置在 Rust 标准库中, 但在 `rand crate` 中可以找到它们。`rand crate` 提供了以下两个方法, 用于从数组、切片或向量中获取随机输出。

`slice.choose(&mut rng)` (随机选取)

返回对切片中随机元素的引用。与 `slice.first()` 和 `slice.last()` 类似, 此方法会返回 `Option<&T>`, 只有当切片为空时才返回 `None`。

`slice.shuffle(&mut rng)`（随机洗牌）

就地随机重排切片中的元素。切片必须通过可变引用传递。

这两个都是 `rand::Rng` 特型的方法，所以你需要一个 `Rng`（random number generator，随机数生成器），以便调用它们。幸运的是，通过调用 `rand::thread_rng()` 很容易得到一个生成器。要对向量 `my_vec` 进行洗牌，可以像下面这样写。

```
use rand::seq::SliceRandom;
use rand::thread_rng;

my_vec.shuffle(&mut thread_rng());
```

16.2.11 Rust 中不存在失效型错误

大多数主流编程语言有集合和迭代器，它们为排除失效型错误做了一点儿变化：不要在迭代集合时修改它。例如，在 Python 中，与向量等价的是列表：

```
my_list = [1, 3, 5, 7, 9]
```

假设我们试图从 `my_list` 中移除所有大于 4 的值：

```
for index, val in enumerate(my_list):
    if val > 4:
        del my_list[index] # bug: 在迭代过程中修改列表

print(my_list)
```

（Python 中的 `enumerate` 函数相当于 Rust 中的 `.enumerate()` 方法，参见 15.3.11 节。）

令人惊讶的是，这个程序打印出了 `[1, 3, 7]`。但是 7 显然大于 4。为什么失控了？这就是失效型错误：程序在迭代数据时修改了数据，让迭代器失效了。在 Java 中，结果将是一个异常。在 C++ 中，这是未定义行为。在 Python 中，虽然此行为有明确定义，但不直

观：迭代器会跳过一个元素。这样一来，`val` 永远不会等于 7，因此也就没有机会删除它了。

我们试着在 Rust 中重现这个 bug：

```
fn main() {  
    let mut my_vec = vec![1, 3, 5, 7, 9];  
  
    for (index, &val) in my_vec.iter().enumerate() {  
        if val > 4 {  
            my_vec.remove(index); // 错误：不能把`my_vec`借用为可变的  
        }  
    }  
    println!("{:?}", my_vec);  
}
```

当然，Rust 在编译时就会拒绝这个程序。当我们调用 `my_vec.iter()` 时，它借用了共享（非 `mut`）的向量引用。引用与迭代器的生命周期一样长，直到 `for` 循环结束。当存在不可变引用时，不能通过调用 `my_vec.remove(index)` 来修改向量。

帮你指出错误固然有用，但你还是得想方设法达成自己的目标。最简单的修复方法是写成如下形式：

```
my_vec.retain(|&val| val <= 4);
```

或者，也可以像在 Python 或任何其他语言中那样使用 `filter` 创建一个新向量。

16.3 VecDeque<T>

`Vec` 只支持在末尾高效地添加元素和移除元素。当程序需要一个地方来存储“排队等候”的值时，`Vec` 可能会很慢。

Rust 的 `std::collections::VecDeque<T>` 是一个双端队列（`deque`, `double-ended queue` 的缩写，发音为 /'dek/）。它支持在首端和尾端进行高效的添加操作和移除操作。

`deque.push_front(value)`（队首推入）

在队列的首端添加一个值。

`deque.push_back(value)`（队尾推入）

在队列的尾端添加一个值。（此方法比 `.push_front()` 更常用，因为队列通常的习惯是在尾端添加值，在首端移除值，就像人们在排队等候一样。）

`deque.pop_front()`（队首弹出）

移除并返回队列的首端值，如果队列为空则返回一个为 `None` 的 `Option<T>`，就像 `vec.pop()` 那样。

`deque.pop_back()`（队尾弹出）

移除并返回队列的尾端值，同样返回 `Option<T>`。

`deque.front()`（队首）和 `deque.back()`（队尾）

与 `vec.first()` 和 `vec.last()` 类似，这两个方法会返回对队列首端或尾端元素的引用。返回值是一个 `Option<&T>`，如果队列为空则为 `None`。

`deque.front_mut()`（队首，可变版）和
`deque.back_mut()`（队尾，可变版）

与 `vec.first_mut()` 和 `vec.last_mut()` 类似，这两个方法会返回 `Option<&mut T>`。

`VecDeque` 的实现是一个环形缓冲区，如图 16-3 所示。

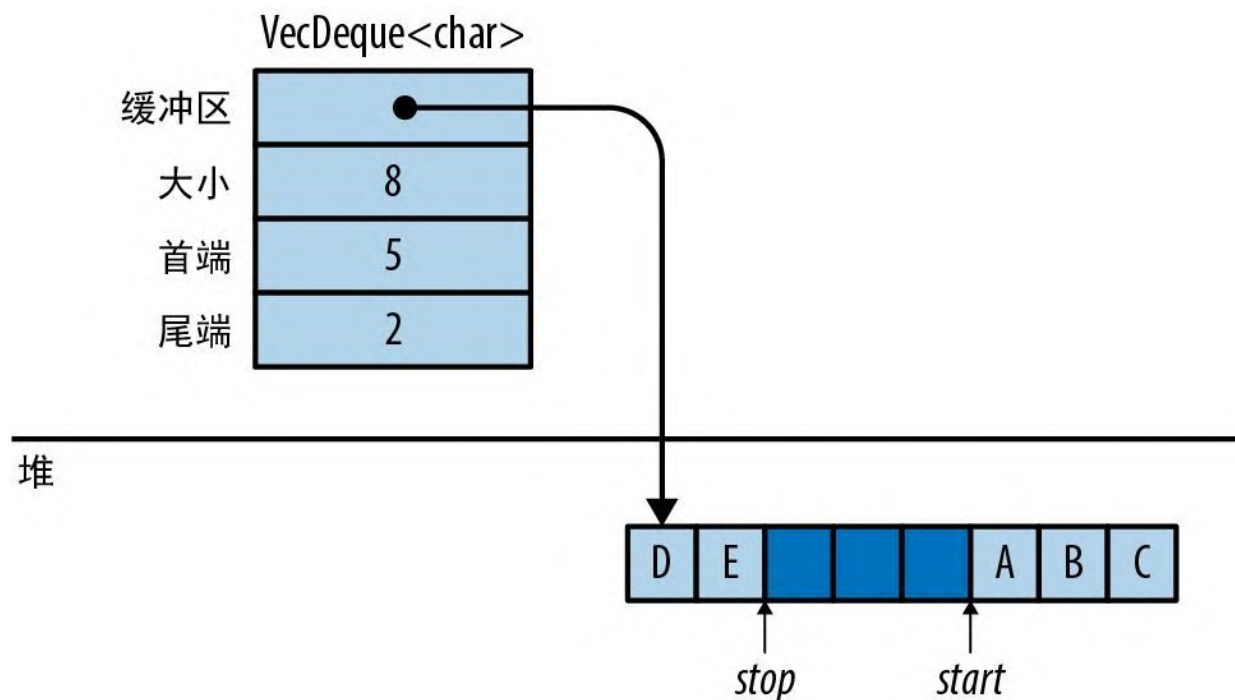


图 16-3: `VecDeque` 在内存中的存储情况

和 `Vec` 一样，`VecDeque` 用一块分配在堆上的内存来存储元素。与 `Vec` 不同，`VecDeque` 的数据并不总是从该区域的开头开始，它可以“回绕”到末尾。这个双端队列的元素按顺序排列是 `['A', 'B', 'C', 'D', 'E']`。`VecDeque` 有两个私有字段，在图 16-3 中被标记为 `start` 和 `stop`，用于记住数据在缓冲区中的首端位置和尾端位置。

从任一端向队列中添加一个值都意味着占用一个未使用的插槽，如图 16-3 中的深灰色色块所示。如果需要，可以回绕或分配更大的内存块。

`VecDeque` 会管理回绕，因此你不必费心于此。图 16-3 展示了 Rust 如何让 `.pop_front()` 更快的幕后原理。

通常，当你需要用到双端队列时，基本上只是需要 `.push_back()` 和 `.pop_front()` 这两个方法。用于创建队列的类型关联函数 `VecDeque::new()` 和 `VecDeque::with_capacity(n)` 和它们在 `Vec` 中的对应函数一样。`VecDeque` 还实现了 `Vec` 中的许多方法，比如 `.len()` 和 `.is_empty()`、`.insert(index, value)`、`.remove(index)`、`.extend(iterable)` 等。

和向量一样，双端队列可以按值、共享引用或可变引用进行迭代。它们有 3 个迭代器方法，即 `.into_iter()`、`.iter()` 和 `.iter_mut()`。它们可以按通常的方式通过索引来访问：`deque[index]`。

因为双端队列不会将自己的元素存储在连续的内存中，所以它们无法继承切片的所有方法。但是，如果你愿意承受移动内容的开销，则可以通过 `VecDeque` 提供的如下方法来解决此问题。

```
deque.make_contiguous() (变连续)
```

获取 `&mut self` 并将 `VecDeque` 重新排列到连续的内存中，返回 `&mut [T]`。

`Vec` 和 `VecDeque` 紧密相关，为了轻松地在两者之间进行转换，标准库提供了两个特型实现。

```
Vec::from(deque) (来自双端队列)
```

`Vec<T>` 实现了 `From<VecDeque<T>>`，因此 `Vec::from(deque)` 能将双端队列变成向量。这个操作的时间复杂度是 $O(n)$ ，因为可能要重新排列元素。

```
VecDeque::from(vec) (来自向量)
```


`VecDeque<T>` 实现了 `From<Vec<T>>`，因此 `VecDeque::from(vec)` 能把向量变成双端队列。这个操作的时间复杂度是 $O(1)$ ，因为 Rust 会直接把向量缓冲区转移给 `VecDeque`，而不会重新分配。

这个方法使得创建具有指定元素的双端队列变得很容易，哪怕并没有标准的 `vec_deque![]` 宏。

```
use std::collections::VecDeque;

let v = VecDeque::from(vec![1, 2, 3, 4]);
```

16.4 BinaryHeap<T>

BinaryHeap（二叉堆）是一种元素组织会保持松散的集合，这样最大值便能总是冒泡到队列的首部。以下是 3 个最常用的 BinaryHeap 方法。

`heap.push(value)`（压入）

向堆中添加一个值。

`heap.pop()`（弹出）

从堆中移除并返回最大值。如果堆为空，则会返回一个为 `None` 的 `Option<T>`。

`heap.peek()`（窥视）

返回对堆中最大值的引用。返回类型是 `Option<&T>`。

`heap.peek_mut()`（窥视，可变版）

返回一个 `PeekMut<T>`，它会返回对堆中最大值的可变引用，并提供类型关联函数 `pop()` 以从堆中弹出该值。使用此方法，我们可以根据最大值来决定是否将其从堆中弹出。

```
use std::collections::binary_heap::PeekMut;

if let Some(top) = heap.peek_mut() {
    if *top > 10 {
        PeekMut::pop(top);
    }
}
```

BinaryHeap 还支持 `Vec` 上的部分方法，包括 `BinaryHeap::new()`、`.len()`、`.is_empty()`、`.capacity()`、`.clear()` 和 `.append(&mut heap2)`。

假设我们用一堆数值填充了 `BinaryHeap`:

```
use std::collections::BinaryHeap;

let mut heap = BinaryHeap::from(vec![2, 3, 8, 6, 9, 5, 4]);
```

值 9 会位于堆顶:

```
assert_eq!(heap.peek(), Some(&9));
assert_eq!(heap.pop(), Some(9));
```

移除了 9 之后也会稍微重新排列其他元素, 以便让 8 位于最前面, 以此类推:

```
assert_eq!(heap.pop(), Some(8));
assert_eq!(heap.pop(), Some(6));
assert_eq!(heap.pop(), Some(5));
...
```

当然, `BinaryHeap` 并不局限于数值。它可以包含实现了内置特型 `Ord` 的任意类型的值。

这使得 `BinaryHeap` 可用作工作队列。你可以定义一个基于优先级实现 `Ord` 的任务结构体, 以便高优先级任务比低优先级任务大一些。然后, 创建一个 `BinaryHeap` 来保存所有待处理的任务。它的 `.pop()` 方法将始终返回最重要的条目, 也就是你的程序下一步就应该处理的任务。

注意: `BinaryHeap` 是可迭代的, 它有一个 `.iter()` 方法, 但此迭代器会以任意顺序而不是从大到小生成堆的元素。要按优先顺序消耗 `BinaryHeap` 中的值, 请使用 `while` 循环。

```
while let Some(task) = heap.pop() {
    handle(task);
}
```

16.5 HashMap<K, V> 与 BTreeMap<K, V>

Map 是键-值对 [称为条目 (entry)] 的集合。任何两个条目都不会有相同的键，并且这些条目会始终按某种数据结构进行组织，从而使你可以通过键在 Map 中高效地查找对应的值。简而言之，Map 就是一个查找表。

Rust 提供了两种 Map 类型：HashMap<K, V> 和 BTreeMap<K, V>。这两种类型共享许多相同的方法，区别在于它们如何组织条目以便进行快速查找。

HashMap 会将键和值存储在哈希表中，因此它需要一个实现了 Hash 和 Eq 的键类型 K，即用来求哈希与判断相等性的标准库特型。

图 16-4 展示了 HashMap 在内存中的排列方式。深灰色区域表示未使用。所有键、值和缓存的哈希码都存储在一个分配在堆上的表中。添加条目最终会迫使 HashMap 分配一个更大的表并将所有数据移入其中。

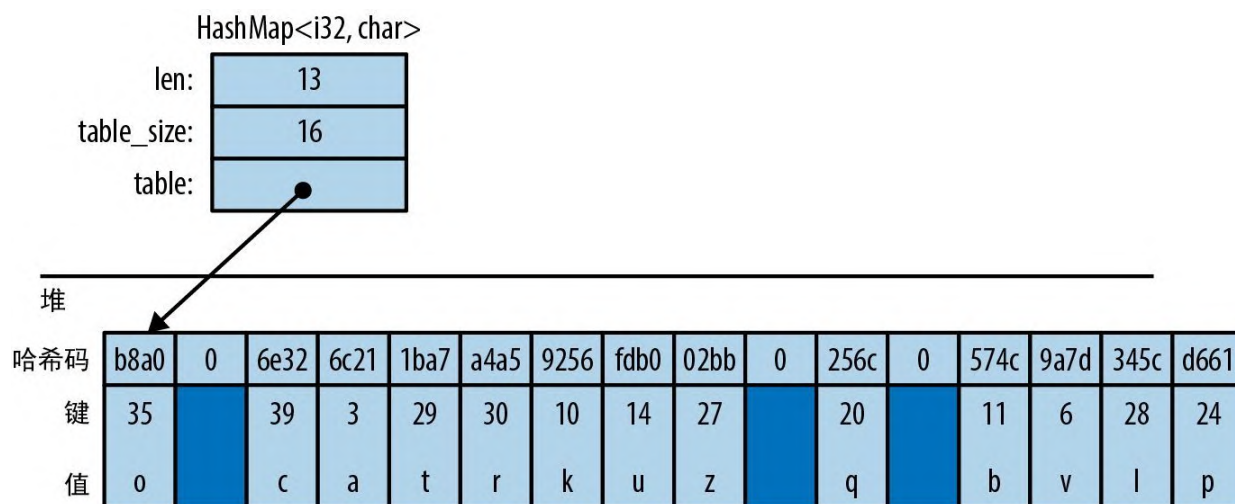


图 16-4：内存中的 HashMap

BTreeMap 会在树结构中按键的顺序存储条目，因此它需要一个实现了 Ord 的键类型 K。图 16-5 展示了一个 BTreeMap。同样，深灰

色区域表示未使用的备用容量。

BTreeMap 中存储条目的单元称为节点。BTreeMap 中的大多数节点仅包含键-值对。非叶节点（比如图 16-5 中所示的根节点）中也有一些空间用于存储指向子节点的指针。（20, 'q'）和（30, 'r'）之间的指针会指向包含 20 和 30 之间所有键的子节点。添加条目通常需要将节点的一些现有条目向右平移，以保持它们的顺序，并且偶尔需要创建新节点。

为了适合页面大小，图 16-5 已略作简化。例如，真正的 BTreeMap 节点会有 11 个条目的空间，而不是 4 个。

Rust 标准库采用了 B 树而不是平衡二叉树，因为 B 树在现代硬件上速度更快。两相对比，二叉树固然在每次搜索时的比较次数较少，但 B 树具有更好的**局部性**（也就是说，内存访问被分组在一起，而不是分散在整个堆中）。这使得 CPU 缓存未命中的情况更为罕见。这会带来显著的速度提升。

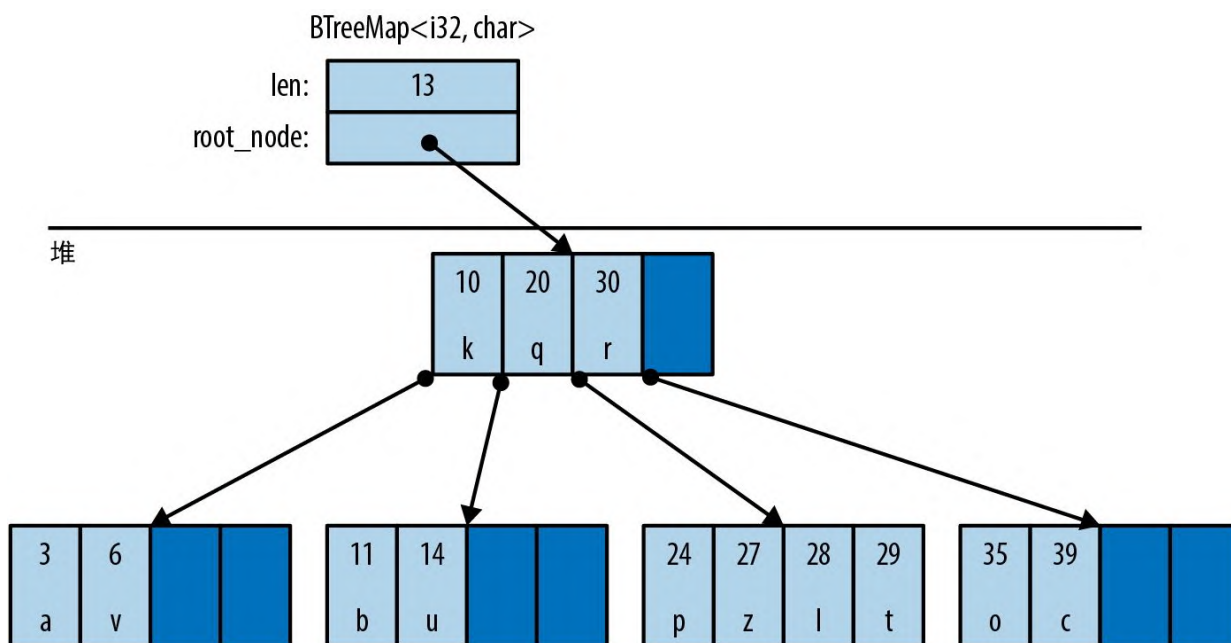


图 16-5：内存中的 BTreeMap

下面是创建 Map 的几种方法。

`HashMap::new()`（新建）和 `BTreeMap::new()`（新建）

创建新的空 `Map`。

`iter.collect()`（收集）

可用于从键-值对创建和填充新的 `HashMap` 或 `BTreeMap`。
`iter` 必须是 `Iterator<Item=(K, V)>` 类型的。

`HashMap::with_capacity(n)`（自带容量）

创建一个新的空 `HashMap`，其中至少有 `n` 个条目的空间。与向量一样，`HashMap` 会将数据存储在分配在堆上的单块内存中，因此它们有容量及其相关方法 `hash_map.capacity()`、`hash_map.reserve(additional)` 和 `hash_map.shrink_to_fit()`。`BTreeMap` 则没有这些。

`HashMap` 和 `BTreeMap` 用于处理键和值的核心方法是一样的。

`map.len()`（长度）

返回条目数。

`map.is_empty()`（为空？）

如果 `map` 没有条目，则返回 `true`。

`map.contains_key(&key)`（包含 `key`？）

如果 `map` 具有给定 `key` 的条目，则返回 `true`。

`map.get(&key)`（按 `key` 获取）

在 `map` 中搜索具有给定 `key` 的条目。如果找到匹配的条目，就返回 `Some(r)`，其中 `r` 是对相应值的引用。如果没找到，则返回 `None`。

`map.get_mut(&key)` (按 key 获取, 可变版)

与 `map.get(&key)` 类似, 但此方法会返回对值的可变引用。

一般来说, Map 允许对其存储的值进行可变访问, 但不允许对键进行可变访问。你可以随意修改这些值, 但键属于 Map 本身, 需要确保它们不会改变, 因为条目是根据对应的键来组织的。对键进行就地修改是错误的。

`map.insert(key, value)` (插入)

将条目 (key, value) 插入 map 并返回旧值 (如果有的话)。返回类型是 `Option<V>`。如果 Map 中已有 key 条目, 则新插入的 value 会覆盖旧值。

`map.extend(iterable)` (用 iterable 扩展)

遍历 iterable 中的 (K, V) 项并将这些键和值逐个插入 map 中。

`map.append(&mut map2)` (从 map2 追加)

将所有条目从 map2 移动到 map 中。之后, map2 会变空。

`map.remove(&key)` (按 key 移除值)

从 map 中查找并移除具有给定 key 的任何条目, 如果存在, 就返回刚刚移除的值。返回类型是 `Option<V>`。

`map.remove_entry(&key)` (按 key 移除条目)

从 map 中查找并移除具有给定 key 的任何条目, 返回刚刚移除的键和值 (如果有的话)。返回类型是 `Option<(K, V)>`。

`map.retain(test)` (留下)

移除所有未通过给定测试的元素。`test` 参数是实现了 `FnMut(&K, &mut V) -> bool` 的函数或闭包。对于 `map` 中的每个元素，都会调用 `test(&key, &mut value)`，如果此函数或闭包返回 `false`，则从 `map` 中移除并丢弃该元素。

除了性能上有些许区别，此方法和下面的写法很像。

```
map = map.into_iter().filter(test).collect();
```

`map.clear()`（清空）

移除所有条目。

也可以使用方括号来查询 `Map`，比如 `map[&key]`。也就是说，`Map` 实现了内置特型 `Index`。但是，如果给定 `key` 还没有条目（就像越界数组访问），则会出现 `panic`，因此只有在要查找的条目肯定已填充过时才应使用此语法。

`.contains_key()`、`.get()`、`.get_mut()` 和 `.remove()` 的 `key` 参数不必具有确切的类型 `&K`。这些方法对可以从 `K` 借用来的类型来说是通用的。可以在 `HashMap<String, Fish>` 上调用 `fish_map.contains_key("conger")`，即便 `"conger"` 不是确切的 `String` 类型也没问题，因为 `String` 实现了 `Borrow<&str>`。有关详细信息，请参阅 13.8 节。

因为 `BTreeMap<K, V>` 会始终保持其条目是根据键排序的，所以它支持一些额外的操作。

`btree_map.split_off(&key)`（拆分出）

把 `btree_map` 一分为二。将键小于 `key` 的条目留在 `btree_map` 中。返回包含其他条目的新 `BTreeMap<K, V>`。

16.5.1 条目

HashMap 和 BTreeMap 都有其对应的 Entry（条目）类型。条目的作用旨在消除冗余的 Map 查找。例如，下面是一些获取或创建学生记录的代码：

```
// 已经有关于此学生的记录了吗？
if !student_map.contains_key(name) {
    // 没有：创建一个
    student_map.insert(name.to_string(), Student::new());
}
// 现在，肯定存在一条记录了
let record = student_map.get_mut(name).unwrap();
...
```

这固然可以正常工作，但它会访问 student_map 两次或 3 次，每次都进行同样的查找。

对于这些条目，我们应该只进行一次查找，生成一个 Entry 值，然后将其用于所有后续操作。下面这个单行代码等效于上一段代码，但它只会执行一次查找：

```
let record =
student_map.entry(name.to_string()).or_insert_with(Student::new);
```

student_map.entry(name.to_string()) 返回的 Entry 值就像对 Map 中某个位置的可变引用，该位置要么由键-值对占用着，要么是空的（意味着那里还没有条目）。如果为空，那么条目的 .or_insert_with() 方法就会插入一个新的 Student。Entry 的大多数用法也是这样的：小而美。

所有 Entry 值都是由同一个方法创建的。

map.entry(key)（按 key 取条目）

返回给定 key 的 Entry。如果 Map 中没有这样的键，则返回一个空的 Entry。

此方法会通过可变引用获取其 self 参数并返回与其生命周期一致的 Entry：

```
pub fn entry<'a>(&'a mut self, key: K) -> Entry<'a, K, V>
```

Entry 类型有一个生命周期参数 'a，因为它实际上是一种奇特的对 Map 的可变引用。只要 Entry 存在，它就拥有对此 Map 的独占访问权。

通过 5.3.5 节，我们已经了解了如何在类型中存储引用以及这会如何影响生命周期。现在我们正在从用户的视角来看待它。Entry 就是一例。

遗憾的是，如果 Map 具有 String 型的键，则无法将 &str 类型的引用传给此方法。在这种情况下，.entry() 方法需要一个真正的 String 型的值。

Entry 值提供了以下 3 个方法来处理空条目。

map.entry(key).or_insert(value)（取条目或插入）

确保 map 包含具有给定 key 的条目，如果需要，就插入具有给定 value 的新条目。此方法会返回对新值或现有值的可变引用。

假设我们需要统计选票。可以这样写：

```
let mut vote_counts: HashMap<String, usize> = HashMap::new();
for name in ballots {
    let count = vote_counts.entry(name).or_insert(0);
    *count += 1;
}
```

.or_insert() 会返回一个可变引用，所以 count 的类型是 &mut usize。

map.entry(key).or_default()（取条目或插入默认值）

确保 map 包含具有给定键的条目，如果需要，就插入一个具有 Default::default() 返回值的新条目。这仅适用于实现了

Default 的类型。与 `or_insert` 类似，此方法会返回对新值或现有值的可变引用。

`map.entry(key).or_insert_with(default_fn)`（取条目或借助 `default_fn` 插入）

与前两个方法类似，不过当需要创建一个新条目时，此方法会调用 `default_fn()` 来生成默认值。如果 `map` 中已经有了 `key` 条目，则不会调用 `default_fn`。

假设我们想知道哪些词出现在了哪些文件中。可以这样写：

```
// 对于每个单词，这个`Map`包含一组曾出现过此单词的文件
let mut word_occurrence: HashMap<String, HashSet<String>> =
    HashMap::new();
for file in files {
    for word in read_words(file)? {
        let set = word_occurrence
            .entry(word)
            .or_insert_with(HashSet::new);
        set.insert(file.clone());
    }
}
```

`Entry` 还提供了一个仅修改现有字段的便捷方法。

`map.entry(key).and_modify(closure)`（取条目并修改）

如果存在具有键 `key` 的条目，则调用 `closure`，并传入对该值的可变引用。此方法会返回 `Entry`，因此可以与其他方法做链式调用。

例如，可以使用此方法来计算字符串中单词出现的次数：

```
// 这个`Map`包含给定字符串的所有单词以及单词的出现次数
let mut word_frequency: HashMap<&str, u32> = HashMap::new();
for c in text.split_whitespace() {
    word_frequency.entry(c)
        .and_modify(|count| *count += 1)
}
```

```
        .or_insert(1);  
    }
```

Entry 类型是为 HashMap 专门定义的一个枚举（BTreeMap 也类似）：

```
// (in std::collections::hash_map)  
pub enum Entry<'a, K, V> {  
    Occupied(OccupiedEntry<'a, K, V>),  
    Vacant(VacantEntry<'a, K, V>)  
}
```

OccupiedEntry 类型和 VacantEntry 类型都有一些无须重复查找即可插入、移除和访问条目的方法。可以在在线文档中找到这些方法。这些额外的方法有时可用于消除一两次冗余查找，不过 .or_insert() 和 .or_insert_with() 已经涵盖了几种常见情况。

16.5.2 对 Map 进行迭代

以下几个方法可以对 Map 进行迭代。

- 按值迭代（for (k, v) in map）以生成 (K, V) 对。这会消耗 Map。
- 按共享引用迭代（for (k, v) in &map）以生成 (&K, &V) 对。
- 按可变引用迭代（for (k, v) in &mut map）以生成 (&K, &mut V) 对。（同样，无法对存储在 Map 中的键进行可变访问，因为这些条目是通过它们的键进行组织的。）

与向量类似，Map 也有 .iter() 方法和 .iter_mut() 方法，它们会返回针对“条目引用”的迭代器，可用来迭代 &map 或 &mut map。此外，还有以下迭代方法。

map.keys()（所有键的迭代器）

返回只有“键引用”的迭代器。

`map.values()`（所有值的迭代器）

返回只有“值引用”的迭代器。

`map.values_mut()`（所有值的可变迭代器）

返回只有“值可变引用”的迭代器。

`map.into_iter()`（转为迭代器）、`map.into_keys()`（转为键迭代器）和 `map.into_values()`（转为值迭代器）

消耗此 `Map`，分别返回遍历键值元组 (K, V) 、键或值的迭代器。

所有 `HashMap` 迭代器都会以任意顺序访问 `Map` 的条目，而 `BTreeMap` 迭代器会按 `key` 的顺序访问它们。

16.6 HashSet<T> 与 BTreeSet<T>

Set 是用于快速进行元素存在性测试的集合：

```
let b1 = large_vector.contains(&"needle");    // 慢，会检查每一个元素
let b2 = large_hash_set.contains(&"needle");  // 快，会按哈希查找
```

Set 中永远不会包含相同值的多个副本。

Map 和 Set 有一些不同的方法，但在幕后，Set 就像一个只有键（而非键-值对）的 Map。事实上，Rust 的两个 Set 类型 HashSet<T> 和 BTreeSet<T> 都是通过对 HashMap<T, ()> 和 BTreeMap<T, ()> 的浅层包装实现的。

HashSet::new()（新建）和 BTreeSet::new()（新建）

创建新 Set。

iter.collect()（收集）

可用于从任意迭代器创建出新 Set。如果 iter 多次生成了同一个值，则重复项将被丢弃。

HashSet::with_capacity(n)（自带容量）

创建一个至少有 n 个值空间的空 HashSet。

HashSet<T> 和 BTreeSet<T> 有以下几个公共的基本方法。

set.len()（长度）

返回 set 中值的数量。

set.is_empty()（为空？）

如果 `set` 不包含任何元素，就返回 `true`。

`set.contains(&value)` (包含)

如果 `set` 包含给定 `value`，就返回 `true`。

`set.insert(value)` (插入)

向 `set` 中添加一个 `value`。如果新增了一个值，就返回 `true`；如果它先前已是此 `set` 的成员，则返回 `false`。

`set.remove(&value)` (移除)

从 `set` 中移除一个 `value`。如果移除了一个值，就返回 `true`；如果它之前不是此 `set` 的成员，则返回 `false`。

`set.retain(test)` (留下)

移除所有未通过给定测试的元素。`test` 参数是实现了 `FnMut(&T) -> bool` 的函数或闭包。对于 `set` 中的每个元素，此方法都会调用 `test(&value)`，如果它返回 `false`，则该元素将被从此 `set` 中移除并丢弃。

除了性能上略有差异，此方法和下面的写法很像。

```
set = set.into_iter().filter(test).collect();
```

与 `Map` 一样，通过引用查找值的方法对于可以从 `T` 借用来的类型都是通用的。有关详细信息，请参阅 13.8 节。

16.6.1 对 **Set** 进行迭代

以下两个方法可以迭代 `Set`。

- 按值迭代 (`for v in set`) 会生成 `Set` 的成员并消耗掉此 `Set`。

- 按共享引用（`for v in &set`）迭代会生成对 `Set` 成员的共享引用。

不支持通过可变引用迭代 `Set`。无法获取对存储在 `Set` 中的值的可变引用。

`set.iter()`（迭代器）

返回 `set` 中成员引用的迭代器。

与 `HashMap` 迭代器类似，`HashSet` 迭代器会以任意顺序生成它们的值。`BTreeSet` 迭代器会按顺序生成值，就像一个排好序的向量。

16.6.2 当相等的值不完全相同时

`Set` 有一些奇怪的方法，只有当你关心“相等”的值之间的差异时才需要使用这些方法。

这种差异确实经常存在。例如，两个内容相同的 `String` 值会将它们的字符存储在内存中的不同位置：

```
let s1 = "hello".to_string();
let s2 = "hello".to_string();
println!("{:p}", &s1 as &str); // 0x7f8b32060008
println!("{:p}", &s2 as &str); // 0x7f8b32060010
```

通常，我们不必在乎这种差异。

但如果确实需要关心这两个 `String` 的存储位置，就可以用以下方法访问存储在 `Set` 中的实际值。如果 `set` 不包含匹配值，则每个方法都会返回一个为 `None` 的 `Option`。

`set.get(&value)`（取值）

返回对等于 `value` 的 `set` 成员的共享引用（如果有的话）。返回类型是 `Option<&T>`。

`set.take(&value)`（拿出值）

与 `set.remove(&value)` 类似，但此方法会返回所移除的值（如果有的话）。返回类型是 `Option<T>`。

`set.replace(value)`（替换为）

与 `set.insert(value)` 类似，但如果 `set` 已经包含一个等于 `value` 的值，那么此方法就会替代并返回原来的值。返回类型是 `Option<T>`。

16.6.3 针对整个 **Set** 的运算

迄今为止，我们看到的大多数 `set` 方法专注于单个 `Set` 中的单个值。`Set` 还有一些对整个 `Set` 进行运算的方法。

`set1.intersection(&set2)`（交集）

返回同时出现在 `set1` 和 `set2` 中的值的迭代器。

如果想打印同时参加脑外科和火箭科学课程的所有学生的姓名，可以这样写：

```
for student in &brain_class {
    if rocket_class.contains(student) {
        println!("{}", student);
    }
}
```

或者，再精简一些：

```
for student in brain_class.intersection(&rocket_class) {
    println!("{}", student);
}
```

令人惊讶的是，有一个运算符能实现同样的效果。

`&set1 & &set2` 会返回一个新 Set，该 Set 是 `set1` 和 `set2` 的交集。这是把“二进制按位与”运算符应用在了两个引用之间。这样就会找到同时存在于 `set1` 和 `set2` 中的值。

```
let overachievers = &brain_class & &rocket_class;
```

`set1.union(&set2)`（并集）

返回存在于 `set1` 或 `set2` 中或者同时存在于两者中的值的迭代器。

`&set1 | &set2` 会返回包含所有这些值的新 Set。它会找出所有存在于 `set1` 或 `set2` 中的值。

`set1.difference(&set2)`（差集）

返回存在于 `set1` 但不在于 `set2` 中的值的迭代器。

`&set1 - &set2` 会返回包含所有此类值的新 Set。

`set1.symmetric_difference(&set2)`（对称差集，异或）

返回存在于 `set1` 或 `set2` 中但不同时存在于两者中的值的迭代器。

`&set1 ^ &set2` 会返回包含所有此类值的新 Set。

以下是测试 Set 之间关系的 3 个方法。

`set1.is_disjoint(set2)`（有交集？）

如果 `set1` 和 `set2` 没有共同的值，就返回 `true`——它们之间的交集为空。

`set1.is_subset(set2)`（是子集？）

如果 `set1` 是 `set2` 的子集，就返回 `true`。也就是说，`set1` 中的所有值都在 `set2` 中。

`set1.is_superset(set2)`（是超集？）

与上一个方法相反：如果 `set1` 是 `set2` 的超集，就返回 `true`。

`Set` 还支持使用 `==` 和 `!=` 进行相等性测试。如果两个 `Set` 包含完全相同的一组值，那它们就是相等的。

16.7 哈希

`std::hash::Hash` 是可哈希类型的标准库特型。`HashMap` 的键和 `HashSet` 的元素都必须实现 `Hash` 和 `Eq`。

大多数实现了 `Eq` 的内置类型也会实现 `Hash`。整数、`char` 和 `String` 都是可哈希的。对元组、数组、切片和向量来说，只要它们的元素是可哈希的，它们自身就是可哈希的。

标准库的一个设计原则是，无论将值存储在何处或如何指向它，都应具有相同的哈希码。因此，引用与其引用的值具有相同的哈希码，而 `Box` 与其封装的值也具有相同的哈希码。向量 `vec` 与包含其所有数据的切片 `&vec[..]` 具有相同的哈希码。`String` 与具有相同字符的 `&str` 具有相同的哈希码。

默认情况下，结构体和枚举没有实现 `Hash`，但可以派生一个实现：

```
/// 大英博物馆藏品中某件物品的ID号
#[derive(Clone, PartialEq, Eq, Hash)]
enum MuseumNumber {
    ...
}
```

只要此类型的字段都是可哈希的，就可以这样用。

如果为一个类型手动实现了 `PartialEq`，那么也应该手动实现 `Hash`。假设我们有一个代表无价历史宝藏的类型：

```
struct Artifact {
    id: MuseumNumber,
    name: String,
    cultures: Vec<Culture>,
    date: RoughTime,
    ...
}
```

如果两个 `Artifact` 具有相同的 ID，那么就认为它们是相等的：

```
impl PartialEq for Artifact {
    fn eq(&self, other: &Artifact) -> bool {
        self.id == other.id
    }
}

impl Eq for Artifact {}
```

由于我们仅是根据这些收藏品的 ID 来比较它们，因此也必须以相同的方式对这些收藏品进行哈希处理：

```
use std::hash::{Hash, Hasher};

impl Hash for Artifact {
    fn hash<H: Hasher>(&self, hasher: &mut H) {
        // 把哈希工作委托给藏品编号
        self.id.hash(hasher);
    }
}
```

（否则，`HashSet<Artifact>` 将无法正常工作。与所有哈希表一样，它要求如果 `a == b`，则必然 `hash(a) == hash(b)`。）

这允许我们创建一个 `Artifact` 的 `HashSet`：

```
let mut collection = HashSet::<Artifact>::new();
```

如上述代码的前一段代码所示，即使要手动实现 `Hash`，也不需要了解任何有关哈希算法的知识。`.hash()` 会接收一个表示哈希算法的 `Hasher` 引用作为参数。你只需将与 `==` 运算符相关的所有数据提供给这个 `Hasher` 即可。`Hasher` 会根据你提供的任何内容计算哈希码。

16.8 使用自定义哈希算法

hash 方法是泛型的，因此 16.7 节展示的 Hash 实现可以将数据提供给实现了 Hasher 的任何类型。这就是 Rust 支持可替换哈希算法的方式。

第三个特型 `std::hash::BuildHasher` 是表示哈希算法初始状态的类型的特型。每个 Hasher 都是单次使用的，就像迭代器一样：用过一次就扔掉了。而 BuildHasher 是可重用的。

每个 HashMap 都包含一个 BuildHasher，每次需要计算哈希码时都会用到。BuildHasher 值包含哈希算法每次运行时所需的键、初始状态或其他参数。

计算哈希码的完整协议如下所示：

```
use std::hash::{Hash, Hasher, BuildHasher};

fn compute_hash<B, T>(builder: &B, value: &T) -> u64
    where B: BuildHasher, T: Hash
{
    let mut hasher = builder.build_hasher(); // 1. 开始此算法
    value.hash(&mut hasher);                // 2. 填入数据
    hasher.finish()                          // 3. 结束，生成u64
}
```

HashMap 每次需要计算哈希码时都会调用这 3 个方法。所有的方法都是可内联的，所以速度非常快。

Rust 的默认哈希算法是著名的 SipHash-1-3 算法。SipHash 的速度很快，而且非常擅长减少哈希冲突。事实上，它也是一个加密算法：目前还没有已知有效方法能刻意生成与 SipHash-1-3 冲突的值。只要每个哈希表使用不同且无法预测的密钥，Rust 就可以安全地抵御一种称为 HashDoS 的拒绝服务攻击，在这种攻击中，攻击者会故意使用哈希冲突来触发服务器的最坏性能。

不过，也许你的应用程序不需要此算法。如果要存储诸如整数或非常短的字符串之类的小型键，则可以实现更快的哈希函数，但代价是要牺牲 HashDoS 的安全性。fnv crate 实现了这样的一个算法，即 Fowler-Noll-Vo (FNV) 哈希。要尝试此算法，请将如下内容添加到你的 Cargo.toml 中：

```
[dependencies]
fnv = "1.0"
```

然后从 fnv 中导入 Map 类型和 Set 类型：

```
use fnv::{FnvHashMap, FnvHashSet};
```

可以使用这两种类型作为 HashMap 和 HashSet 的无缝替代品。浏览一下 fnv 源代码，就会发现它们是如何定义的：

```
/// 使用默认FNV哈希器的`HashMap`
pub type FnvHashMap<K, V> = HashMap<K, V, FnvBuildHasher>;

/// 使用默认FNV哈希器的`HashSet`
pub type FnvHashSet<T> = HashSet<T, FnvBuildHasher>;
```

标准的 HashMap 集合和 HashSet 集合会接受一个可选的额外类型参数来指定哈希算法，FnvHashMap 和 FnvHashSet 是 HashMap 和 HashSet 的泛型类型别名，用于为那个参数指定一个 FNV 哈希器。

16.9 在标准集合之外

在 Rust 中创建一个新的自定义集合类型和在其他语言中非常相似。你可以通过组合语言提供的部件（结构体和枚举、标准集合、Option、Box 等）来组织数据。有关示例，请参阅 10.1.4 节定义的 BinaryTree<T> 类型。

如果你习惯于在 C++ 中实现数据结构，使用裸指针、手动内存管理、定位放置（placement）new 和显式析构函数调用来获得最佳性能，那么你无疑会发现这在安全的 Rust 中处处受限。所有这些工具本质上都是不安全的。可以在 Rust 中使用它们，但前提是要使用不安全的代码。第 22 章展示了如何通过不安全的代码实现它们，其中包括一个示例，该示例使用了一些不安全的代码来实现安全的自定义集合。

现在，我们将沐浴在标准集合及其安全、高效 API 的和煦阳光中。与 Rust 标准库中的许多 API 一样，这些 API 旨在让你尽可能少写一点儿 unsafe 代码。

第 17 章 字符串与文本

字符串是一个光秃秃的数据结构，其途经之地会出现很多重复的处理。它简直是隐藏重要信息的“完美”手段。

——Alan Perlis，警句 #34

本书一直在使用 Rust 的主要文本类型 `String`、`str` 和 `char`。3.7 节曾讲解过字符和字符串字面量的语法，也展示过字符串在内存中的表示方式。在本章中，我们将更详细地介绍文本处理技术。

本章包括如下内容。

- 提供一些 Unicode 背景知识来帮助你理解标准库的设计。
- 讲解表示单个 Unicode 码点的 `char` 类型。
- 讲解 `String` 类型和 `str` 类型，二者是表示拥有和借用的 Unicode 字符序列。它们有各种各样的方法来构建、搜索、修改和迭代其内容。
- 介绍 Rust 的字符串格式化工具，比如 `println!` 宏和 `format!` 宏。你可以编写自己的宏来处理格式化字符串，并扩展它们以支持自己的类型。
- 概述 Rust 对正则表达式的支持。
- 讨论为什么 Unicode 的规范化很重要，并展示如何在 Rust 中对其进行规范化。

17.1 一些 Unicode 背景知识

本书是关于 Rust 而不是 Unicode 的，后者已经有专门的书籍介绍它了。但是，Rust 的字符类型和字符串类型都是围绕 Unicode 设计的。此处介绍一些 Unicode 的背景知识有助于更好地理解 Rust。

17.1.1 ASCII、Latin-1 和 Unicode

Unicode 和 ASCII 对于从 0 到 0x7f 的所有 ASCII 码点是一一对应的，比如，它们都为字符 * 分配了码点 42。同样，Unicode 也将 0 到 0xff 分配给了与 ISO/IEC 8859-1 字符集相同的字符，这是 ASCII 字符集用于西欧语言的 8 位超集。Unicode 将此码点范围称为 Latin-1 码块，因此我们也将使用耳熟能详的名称 Latin-1 来指代 ISO/IEC 8859-1。

由于 Unicode 是 Latin-1 的超集，因此将 Latin-1 转换为 Unicode 甚至不需要查表：

```
fn latin1_to_char(latin1: u8) -> char {
    latin1 as char
}
```

反向转换也很简单，假设码点落在了 Latin-1 范围内。

```
fn char_to_latin1(c: char) -> Option<u8> {
    if c as u32 <= 0xff {
        Some(c as u8)
    } else {
        None
    }
}
```

17.1.2 UTF-8 编码

Rust 的 String 类型和 str 类型表示使用了 UTF-8 编码形式的文本。UTF-8 会将字符编码为 1~4 字节的序列，如图 17-1 所示。

UTF-8 编码（1~4字节）	表示的码点	范围
0 x x x x x x x	0bxxxxxx	0 ~ 0x7f
1 1 0 x x x x x 1 0 y y y y y y	0bxxxxxyyyyy	0x80 ~ 0x7ff
1 1 1 0 x x x x 1 0 y y y y y y 1 0 z z z z z z	0bxxxxxyyyyyzzzzz	0x800 ~ 0xffff
1 1 1 1 0 x x x 1 0 y y y y y y 1 0 z z z z z z 1 0 w w w w w w	0bxxxxxyyyyyzzzzzwwwww	0x10000 ~ 0x10ffff

图 17-1： UTF-8 编码

格式良好的 UTF-8 序列有两个限制。首先，只有任何给定码点的最短编码才被认为是格式良好的，你不能花费 4 字节来编码原本只需要 3 字节的码点。此规则确保了每个码点只会有唯一一个 UTF-8 编码。其次，格式良好的 UTF-8 不得对从 0xd800 到 0xdfff 或超过 0x10ffff 的数值进行编码：这些数值要么保留用作非字符目的，要么完全超出了 Unicode 的范围。

图 17-2 展示了一些示例。

UTF-8 编码（1~4字节）	表示的码点	范围
0 0 1 0 1 0 1 0	0b0101010 == 0x2a	'*'
1 1 0 0 1 1 1 0 1 0 1 1 1 1 0 0	0b01110_111100 == 0x3bc	'μ'
1 1 1 0 1 0 0 1 1 0 0 0 1 1 0 0 1 0 0 0 1 1 0	0b1001_001100_000110 == 0x9306	'錆' (sabi: rust)
1 1 1 1 0 0 0 0 1 0 0 1 1 1 1 1 1 0 1 0 1 0 0 1 1 0 1 0 0 0 0 0 0	0b000_011111_100110_000000 == 0x1f980	'🦀' (crab emoji)

图 17-2： UTF-8 示例

请注意，虽然在螃蟹表情符号的编码中其前导字节对码点只贡献了一串 0，但是它仍然需要用 4 字节来编码：3 字节的 UTF-8 编码只能表达 16 位码点，而 0x1f980 有 17 位长。

下面是一个包含具有不同编码长度字符的字符串的简单示例：

```
assert_eq!("うどん: udon".as_bytes(),
          &[0xe3, 0x81, 0x86, // う
            0xe3, 0x81, 0xa9, // ど
            0xe3, 0x82, 0x93, // ん
            0x3a, 0x20, 0x75, 0x64, 0x6f, 0x6e // : udon
          ]);
```

图 17-2 还展示了 UTF-8 一些非常有用的属性。

- 由于 UTF-8 会把码点 $0 \sim 0x7f$ 编码为字节 $0 \sim 0x7f$ ，因此一段 ASCII 文本必然是有效的 UTF-8 字符串。反过来，如果 UTF-8 字符串中只包含 ASCII 字符，则它也必然是有效的 ASCII 字符串。

对于 Latin-1 则不是这样的，比如，Latin-1 会将 `é` 编码为字节 `0xe9`，而 UTF-8 会将其解释为三字节编码中的第一字节。

- 通过查看任何字节的高位，就能立刻判断出它是某个字符的 UTF-8 编码的起始字节还是中间字节。
- 编码的第一字节会单独通过其前导位告诉你编码的全长。
- 由于不会有任何编码超过 4 字节，因此 UTF-8 在处理时从不需要无限循环，这在处理不受信任的数据时非常有用。
- 在格式良好的 UTF-8 中，即使从字节中的任意点开始，你也始终可以明确地找出该字符编码的起始位置和结束位置。UTF-8 的第一字节和后面的字节一定不同，所以一段编码不可能从另一段编码的中间开始。第一字节会确定编码的总长度，因此任何一段编码都不可能是另一段编码的前缀。这很有用。例如，要在 UTF-8 字符串中搜索 ASCII 分隔符只需对分隔符的字节进行简单扫描即可。这个分隔符永远不会作为多字节编码的任何部分出现，因此根本不需要跟踪 UTF-8 的结构。类似地，在一个字节串中搜索另一个字节串的算法无须针对 UTF-8 字符串做修改即可正常工作，甚至连那些根本不会检查待搜文本中每字节的算法也没问题。

尽管可变宽度编码比固定宽度编码更复杂，但以上特征让 UTF-8 比预想的更容易使用。标准库会帮你处理绝大部分问题。

17.1.3 文本方向性

拉丁文、西里尔文、泰文等文字是从左向右书写的，而希伯来文、阿拉伯文等文字则是从右向左书写的。Unicode 以写入或读取字符的常规顺序存储字符，因此在这种情况下字符串（如希伯来语文本）中保存的首字节是对写在最右端的字符的编码。

```
assert_eq!("ערב טוֹב".chars().next(), Some('ע'));
```

17.2 字符（char）

Rust 的 `char` 类型是一个包含 Unicode 码点的 32 位值。`char` 保证会落在 `0~0xd7ff` 或 `0xe000~0x10ffff` 范围内，所有用于创建和操作 `char` 值的方法都会确保此规则永远成立。`char` 类型实现了 `Copy` 和 `Clone`，以及用于比较、哈希和格式化的所有常用特型。

字符串切片可以使用 `slice.chars()` 生成针对其字符的迭代器：

```
assert_eq!("カニ".chars().next(), Some('カ'));
```

接下来的讲解中出现的变量 `ch` 全都是 `char` 类型的。

17.2.1 字符分类

`char` 类型的一些方法可以将字符分入几个常见类别，如表 17-1 所示。这些都是从 Unicode 中提取的定义。

表 17-1: `char` 类型的分类方法

方 法	描 述	例 子
<code>ch.is_numeric()</code>	数值字符，包括 Unicode 通用类别中的“数值：数字”和“数值：字母”，但不包括“数值：其他”	<code>'4'.is_numeric()</code> <code>'𐐆'.is_numeric()</code> <code>'⑧'.is_numeric()</code>
<code>ch.is_alphabetic()</code>	字母字符：由 Unicode 字符属性“Alphabetic”派生而来	<code>'q'.is_alphabetic()</code> <code>'七'.is_alphabetic()</code>
<code>ch.is_alphanumeric()</code>	数值或字母，两者的定义如前所述	<code>'9'.is_alphanumeric()</code> <code>'鯔'.is_alphanumeric()</code> <code>!''.is_alphanumeric()</code>
<code>ch.is_whitespace()</code>	空白字符：Unicode 字符属性“WSpace=Y”	<code>' '.is_whitespace()</code> <code>'\n'.is_whitespace()</code> <code>'\u{A0}'.is_whitespace()</code>
<code>ch.is_control()</code>	控制字符：Unicode 的“其他：控制字符”大类	<code>'\n'.is_control()</code> <code>'\u{85}'.is_control()</code>

一组仅限于 ASCII 的方法，对任何非 ASCII `char` 都会返回 `false`，如表 17-2 所示。

表 17-2: `char` 的 ASCII 分类方法

方法	描述	例子

方法	描述	例子
<code>ch.is_ascii()</code>	ASCII 字符：码点介于 0 和 127 之间的字符	<code>'n'.is_ascii()</code> <code>!'ñ'.is_ascii()</code>
<code>ch.is_ascii_alphabetic()</code>	大写或小写 ASCII 字母，在 'A'..'Z' 或 'a'..'z' 范围内	<code>'n'.is_ascii_alphabetic()</code> <code>!'1'.is_ascii_alphabetic()</code> <code>!'ñ'.is_ascii_alphabetic()</code>
<code>ch.is_ascii_digit()</code>	ASCII 数字，在 '0'..'9' 范围内	<code>'8'.is_ascii_digit()</code> <code>!'-'.is_ascii_digit()</code> <code>!'@'.is_ascii_digit()</code>
<code>ch.is_ascii_hexdigit()</code>	'0'..'9'、'A'..'F' 或 'a'..'f' 范围内的任何字符	
<code>ch.is_ascii_alphanumeric()</code>	ASCII 数字或者大写字母或小写字母	<code>'q'.is_ascii_alphanumeric()</code> <code>'0'.is_ascii_alphanumeric()</code>
<code>ch.is_ascii_control()</code>	ASCII 控制字符，包括 DEL	<code>'\n'.is_ascii_control()</code> <code>'\x7f'.is_ascii_control()</code>
<code>ch.is_ascii_graphic()</code>	会在页面上留下墨迹的任何 ASCII 字符：既不是空白字符也不是控制字符	<code>'Q'.is_ascii_graphic()</code> <code>'~'.is_ascii_graphic()</code> <code>!' '.is_ascii_graphic()</code>
<code>ch.is_ascii_uppercase()</code> , <code>ch.is_ascii_lowercase()</code>	ASCII 大写字母和小写字母	<code>'z'.is_ascii_lowercase()</code> <code>'Z'.is_ascii_uppercase()</code>
<code>ch.is_ascii_punctuation()</code>	既不是字母也不是数字的任何 ASCII 图形字符 ¹	
<code>ch.is_ascii_whitespace()</code>	ASCII 空白字符：空格、水平制表符、换行符、换页符或回车符	<code>' '.is_ascii_whitespace()</code> <code>'\n'.is_ascii_whitespace()</code> <code>!' \u{A0}'.is_ascii_whitespace()</code>

¹也就是广义标点。——译者注

所有 `is_ascii_...` 方法也可用于 `u8` 字节类型:

```
assert!(32u8.is_ascii_whitespace());
assert!(b'9'.is_ascii_digit());
```

在使用这些函数来实现现有规范（如编程语言标准或文件格式）时一定要小心，因为这些分类可能存在某些令人吃惊的差异。例如，注意 `is_whitespace` 和 `is_ascii_whitespace` 对某些字符的处理不同:

```
let line_tab = '\u{000b}'; //“行间制表符”，也叫“垂直制表符”
assert_eq!(line_tab.is_whitespace(), true);
assert_eq!(line_tab.is_ascii_whitespace(), false);
```

这是因为 `char::is_ascii_whitespace` 函数实现了许多 Web 标准中通用的空白字符定义，而 `char::is_whitespace` 遵循的是 Unicode 标准。

17.2.2 处理数字

对于数字的处理，可以使用以下方法。

`ch.to_digit(radix)`（转数字）

判断 `ch` 是不是以 `radix` 为基数的数字。如果是，就返回 `Some(num)`，其中 `num` 是 `u32`；否则，返回 `None`。此方法只会识别 ASCII 数字，而不包括 `char::is_numeric` 涵盖的更广泛的字符类别。`radix` 参数的范围可以从 2 到 36。对于大于 10 的基数，会用 ASCII 字母（不分大小写）表示值为 10 到 35 的数字。

`std::char::from_digit(num, radix)`（来自数字）

自由函数，只要有可能，就可以把 `u32` 数字值 `num` 转换为 `char`。如果 `num` 可以表示为 `radix` 中的单个数字，那么 `from_digit` 就会返回 `Some(ch)`，其中 `ch` 是数字。当 `radix` 大于 10 时，`ch` 可以是小写字母。否则，它会返回 `None`。

这是 `to_digit` 的逆函数。如果 `std::char::from_digit(num, radix)` 等于 `Some(ch)`，则 `ch.to_digit(radix)` 等于 `Some(num)`。如果 `ch` 是 ASCII 数字或小写字母，则反之亦成立。

`ch.is_digit(radix)`（是数字？）

如果 `ch` 可以表示以 `radix` 为基数的 ASCII 数字，就返回 `true`。此方法等效于 `ch.to_digit(radix) != None`。

关于上述方法，举例如下。

```
assert_eq!('F'.to_digit(16), Some(15));
assert_eq!(std::char::from_digit(15, 16), Some('f'));
assert!(char::is_digit('f', 16));
```

17.2.3 字符大小写转换

处理字符大小写的方法如下。

`ch.is_lowercase()`（是小写？）和 `ch.is_uppercase()`（是大写？）

指出 `ch` 是小写字母字符还是大写字母字符。这两个方法遵循 Unicode 的派生属性 `Lowercase`（小写字母）和 `Uppercase`（大写字母），因此它们涵盖了非拉丁字母表（如希腊字母和西里尔字母），并给出了和 ASCII 一样的预期结果。

`ch.to_lowercase()`（转小写）和 `ch.to_uppercase()`（转大写）

根据 Unicode 的默认大小写转换算法，返回生成 `ch` 的小写和大写对应字符的迭代器：

```
let mut upper = 's'.to_uppercase();
assert_eq!(upper.next(), Some('S'));
assert_eq!(upper.next(), None);
```

这两个方法会返回迭代器而不是单个字符，因为 Unicode 中的大小写转换并不总是一对一的过程：

```
// 德文字母"ß"的大写形式是"SS":
let mut upper = 'ß'.to_uppercase();
assert_eq!(upper.next(), Some('S'));
assert_eq!(upper.next(), Some('S'));
assert_eq!(upper.next(), None);

// Unicode规定在将带点的土耳其大写字母'i'变为小写时要转成'i'后跟一个
// '\u{307}'，把点组合到字母上，以便在随后转换回大写字母时保留这个点
let ch = 'İ'; // '\u{130}'
let mut lower = ch.to_lowercase();
assert_eq!(lower.next(), Some('i'));
assert_eq!(lower.next(), Some('\u{307}'));
assert_eq!(lower.next(), None);
```

为便于使用，这些迭代器都实现了 `std::fmt::Display` 特型，因此可以将它们直接传给 `println!` 或 `write!` 宏。

17.2.4 与整数之间的转换

Rust 的 `as` 运算符会将 `char` 转换为任何整数类型，并抹掉高位：

```
assert_eq!('B' as u32, 66);
assert_eq!('鰺' as u8, 66); // 截断高位
assert_eq!('二' as i8, -116); // 同上
```

`as` 运算符会将任何 `u8` 值转换为 `char`，并且 `char` 也实现了 `From<u8>`。但是，更宽的整数类型可以表示无效码点，因此对于那部分整数，必须使用 `std::char::from_u32` 进行转换，它会返回 `Option<char>`。

```
assert_eq!(char::from(66), 'B');
assert_eq!(std::char::from_u32(0x9942), Some('鰺'));
assert_eq!(std::char::from_u32(0xd800), None); // 为UTF-16保留的码点
```

17.3 String 与 str

Rust 的 `String` 类型和 `str` 类型会保证自己只包含格式良好的 UTF-8。标准库通过限制你创建 `String` 值和 `str` 值的方式以及可以对它们执行的操作来确保这一点。这样，当引入这些值时一定是格式良好的，而且在使用中也是如此。它们所有的方法都会坚守这个保证：对它们的任何安全操作都不会引入格式错误的 UTF-8。这就简化了处理文本的代码。

Rust 可以将文本处理方法关联到 `str` 或 `String` 上，具体关联到哪个取决于该方法是需要可调整大小的缓冲区还是仅满足于就地使用文本。由于 `String` 可以解引用成 `&str`，因此在 `str` 上定义的每个方法都可以直接在 `String` 上使用。本节会介绍这两种类型的方法，并按其功能粗略分组。

文本处理方法会按字节偏移量索引文本并以字节而不是字符为单位测量其长度。实际上，考虑到 Unicode 的性质，按字符索引并不像看起来那么有用，按字节偏移量索引反而更快且更简单。如果试图使用位于某个字符的 UTF-8 编码中间的字节偏移量，则该方法会发生 panic，因此不能通过这种方式引入格式错误的 UTF-8。

`String` 通过封装 `Vec<u8>` 实现，并确保向量中的内容永远是格式良好的 UTF-8。Rust 永远不会把 `String` 改成更复杂的表示形式，因此你可以假设 `String` 的性能表现始终会和 `Vec` 保持一致。

在后面的讲解里，所有用到的变量都具有表 17-3 中给出的类型。

表 17-3：后面的讲解里要用到的变量类型

变量	预设类型
<code>string</code>	<code>String</code>

变量	预设类型
slice	<code>&str</code> 或对某值（如 <code>String</code> 或 <code>Rc<String></code> ）的解引用
ch	<code>char</code>
n	<code>usize</code> ，长度
i、j	<code>usize</code> ，字节偏移量
range	字节偏移量的 <code>usize</code> 范围，可以像 <code>i..j</code> 一样完全有界，也可以像 <code>i..</code> 、 <code>..j</code> 或 <code>..</code> 一样部分有界
pattern	任何模式类型： <code>char</code> 、 <code>String</code> 、 <code>&str</code> 、 <code>&[char]</code> 或 <code>FnMut(char) -> bool</code>

17.3.6 节会讲解模式类型。

17.3.1 创建字符串值

创建 `String` 值的常见方法有以下几种。

`String::new()`（新建）

返回一个新的空字符串。这时还没有在堆上分配缓冲区，但将来会按需分配。

`String::with_capacity(n)`（自带容量）

返回一个新的空字符串，其中预先分配了一个足以容纳至少 `n` 字节的缓冲区。如果事先知道要构建的字符串的长度，则此构造函数可以让你从一开始就正确设置缓冲区大小，而不是等构建字符串时再进

行调整。如果字符串的长度超过 `n` 字节，则该字符串仍会根据需要增加其缓冲区。与向量一样，字符串也有 `capacity` 方法、`reserve` 方法和 `shrink_to_fit` 方法，但一般来说默认的分配逻辑就很好。

`str_slice.to_string()`（转字符串）

分配一个新的 `String`，其内容是 `str_slice` 的副本。本书一直在使用诸如 `"literal text".to_string()` 之类的表达式来从字符串字面量生成 `String`。

`iter.collect()`（收集）

通过串联迭代器的各个条目构造出字符串，迭代器的条目可以是 `char` 值、`&str` 值或 `String` 值。例如，要从字符串中移除所有空格，可以这样写：

```
let spacey = "man hat tan";
let spaceless: String =
    spacey.chars().filter(|c| !c.is_whitespace()).collect();
assert_eq!(spaceless, "manhattan");
```

以这种方式使用 `collect` 可以充分利用 `String` 对 `std::iter::FromIterator` 特型的实现。

`slice.to_owned()`（转自有）

将 `slice` 的副本作为新分配的 `String` 返回。`str` 类型无法实现 `Clone`：该特型需要在 `&str` 上进行 `clone` 以返回 `str` 值，但 `str` 是无固定大小类型。不过，`&str` 实现了 `ToOwned`，这能让实现者指定其自有（`Owned`）版本的等效类型。

17.3.2 简单探查

下面这些方法可以从字符串切片中获取基本信息。

`slice.len()`（长度）

`slice` 的长度，以字节为单位。

`slice.is_empty()`（为空？）

如果 `slice.len() == 0`，就返回 `True`。

`slice[range]`（范围内切片）

返回借用了 `slice` 给定部分的切片。有界的范围、部分有界的范围和无界的范围都可以。

例如：

```
let full = "bookkeeping";
assert_eq!(&full[..4], "book");
assert_eq!(&full[5..], "eeping");
assert_eq!(&full[2..4], "ok");
assert_eq!(full[..].len(), 11);
assert_eq!(full[5..].contains("boo"), false);
```

请注意，不能索引具有单个位置的字符串切片，比如 `slice[i]`。要想在给定的字节偏移处获取单个字符有点儿笨拙：必须在切片上生成一个 `chars` 迭代器，并要求它解析成单个字符的 UTF-8：

```
let parenthesized = "Rust (鯰)";
assert_eq!(parenthesized[6..].chars().next(), Some('鯰'));
```

不过，你很少需要这样做。Rust 有更好的方法来迭代切片，17.3.8 节会对此进行讲解。

`slice.split_at(i)`（拆分于）

返回从 `slice` 借来的两个共享切片的元组：一个是字节偏移量 `i` 之前的部分，另一个是字节偏移量 `i` 之后的部分。换句话说，这会返回 `(slice[..i], slice[i..])`。

`slice.is_char_boundary(i)`（是字符边界？）

如果字节偏移量 `i` 恰好落在字符边界之间并且适合作为 `slice` 的偏移量，就返回 `True`。

自然，也可以对切片做相等性比较、排序和哈希。有序比较只是将字符串视为一系列 Unicode 码点，并按字典顺序进行比较。

17.3.3 追加文本与插入文本

以下方法会将文本添加到 `String` 中。

`string.push(ch)`（压入）

将字符 `ch` 追加到 `string` 的末尾。

`string.push_str(slice)`（压入字符串）

追加 `slice` 的全部内容。

`string.extend(iter)`（以 `iter` 扩展）

将迭代器 `iter` 生成的条目追加到字符串中。迭代器可以生成 `char` 值、`str` 值或 `String` 值。这是 `String` 对 `std::iter::Extend` 特型的实现。

```
let mut also_spaceless = "con".to_string();
also_spaceless.extend("tri but ion".split_whitespace());
assert_eq!(also_spaceless, "contribution");
```

`string.insert(i, ch)`（插入于）

在 `string` 内的字节偏移量 `i` 处插入单个字符 `ch`。这需要平移 `i` 之后的所有字符以便为 `ch` 腾出空间，因此用这种方式构建字符串的时间复杂度是 $O(n)^2$ 。

²不考虑重新分配内存的情况。——译者注

`string.insert_str(i, slice)`（插入字符串于）

这会在 `string` 内插入 `slice`，但同样需要注意性能问题。

`String` 实现了 `std::fmt::Write`，这意味着 `write!` 宏和 `writeln!` 宏可以将格式化后的文本追加到 `String` 上：

```
use std::fmt::Write;

let mut letter = String::new();
writeln!(letter, "Whose {} these are I think I know",
"rutabagas")?;
writeln!(letter, "His house is in the village though;")?;
assert_eq!(letter, "Whose rutabagas these are I think I know\n\
His house is in the village though;\n");
```

由于 `write!` 和 `writeln!` 是专为写入输出流而设计的，因此它们会返回一个 `Result`，如果你忽略 `Result`，则 Rust 会报错。上述代码使用了 `?` 运算符来处理错误，但实际上写入 `String` 是肯定不会出错的，因此这种情况下也可以调用 `.unwrap()`。

因为 `String` 实现了 `Add<&str>` 和 `AddAssign<&str>`，所以你可以编写如下代码：

```
let left = "partners".to_string();
let mut right = "crime".to_string();
assert_eq!(left + " in " + &right, "partners in crime");

right += " doesn't pay";
assert_eq!(right, "crime doesn't pay");
```

当应用于字符串时，`+` 运算符会按值获取其左操作数，所以实际上它可以重用该 `String` 的缓冲区作为加法的结果。因此，如果左操作数的缓冲区足够容纳结果，那么就不需要分配内存。

遗憾的是，此运算不是对称的，`+` 的左操作数不能是 `&str`，所以不能写成：

```
let parenthetical = "(" + string + ");
```

只能改成：


```
let parenthetical = "(" + string.to_string() + &string + "');
```

不过，此限制确实妨碍了从末尾向开头反向构建字符串的方式。这种方式性能不佳，因为必须反复把文本平移到缓冲区的末尾。

然而，通过向末尾追加小片段的方式从头到尾构建字符串是高效的。`String` 的行为方式与向量是一样的，当它需要更多容量时，总是至少将其缓冲区大小加倍。这就令再次复制的开销与字符串的最终大小成正比。不过，使用 `String::with_capacity` 创建具有正确缓冲区大小的字符串可以完全避免调整大小，并且可以减少对堆分配器的调用次数。

17.3.4 移除文本与替换文本

`String` 有以下几个移除文本的方法。（这些方法不会影响字符串的容量，如果需要释放内存，请使用 `shrink_to_fit`。）

`string.clear()`（清空）

将 `string` 重置为空字符串。

`string.truncate(n)`（截断为 n 个）

丢弃字节偏移量 n 之后的所有字符，留下长度最多为 n 的 `string`。如果 `string` 短于 n 字节，则毫无效果。

`string.pop()`（弹出）

从 `string` 中移除最后一个字符（如果有的话），并将其作为 `Option<char>` 返回。

`string.remove(i)`（移除）

从 `string` 中移除字节偏移量 i 处的字符并返回该字符，将后面的所有字符平移到前面。这个操作所花费的时间与后续字符的数量呈线性关系。

`string.drain(range)` (抽取)

返回给定字节索引范围内的迭代器，并在迭代器被丢弃后移除字符。范围之后的所有字符都会向前平移：

```
let mut choco = "chocolate".to_string();
assert_eq!(choco.drain(3..6).collect::<String>(), "col");
assert_eq!(choco, "choate");
```

如果只是想移除这个范围，则可以立即丢弃此迭代器，而不从中提取任何条目。

```
let mut winston = "Churchill".to_string();
winston.drain(2..6);
assert_eq!(winston, "Chill");
```

`string.replace_range(range, replacement)` (替换范围)

用给定的替代字符串切片替换 `string` 中的给定范围。切片不必与要替换的范围长度相同，但除非要替换的范围已到达 `string` 的末尾，否则将需要移动范围末尾之后的所有字节。

```
let mut beverage = "a piña colada".to_string();
beverage.replace_range(2..7, "kahlua"); // 'ñ' 是两字节的！
assert_eq!(beverage, "a kahlua colada");
```

17.3.5 搜索与迭代的约定

Rust 用于搜索文本和迭代文本的标准库函数遵循了一些命名约定，以便于记忆。

r

大多数操作会从头到尾处理文本，但名称以 **r** 开头的操作会从尾到头处理。例如，`rsplit` 是 `split` 的从尾到头版本。在某些情况

下，改变处理方向不仅会影响值生成的顺序，还会影响值本身。具体示例请参见图 17-3。

`n`

名称以 `n` 结尾的迭代器会将自己限定为只取给定数量的匹配项。

`_indices`³

³`index` 的复数形式。——译者注

名称以 `_indices` 结尾的迭代器会生成通常的迭代值和在此 `slice` 中的字节偏移量组成的值对。

标准库并不会提供每个操作的所有组合。例如，许多操作并不需要 `n` 变体，因为很容易简单地提前结束迭代。

17.3.6 搜索文本的模式

当标准库函数需要搜索、匹配、拆分或修剪文本时，它能接受如下几种类型来表示要查找的内容：

```
let haystack = "One fine day, in the middle of the night";
assert_eq!(haystack.find(','), Some(12));
assert_eq!(haystack.find("night"), Some(35));
assert_eq!(haystack.find(char::is_whitespace), Some(3));
```

这些类型称为**模式**，大多数操作支持它们。

```
assert_eq!("## Elephants"
    .trim_start_matches(|ch: char| ch == '#' ||
ch.is_whitespace()),
    "Elephants");
```

标准库支持 4 种主要的模式。

- 以 `char` 作为模式意味着要匹配该字符。

- 以 `String`、`&str` 或 `&&str` 作为模式，意味着要匹配等于该模式的子串。
- 以 `FnMut(char) -> bool` 闭包作为模式，意味着要匹配该闭包返回 `true` 的单个字符。
- 以 `&[char]`（注意并不是 `&str`，而是 `char` 的切片）作为模式，意味着要匹配该列表中出现的任何单个字符。请注意，如果将此列表写成数组字面量，那么可能要调用 `as_ref()` 来获得正确的类型。

```
let code = "\t    function noodle() { ";
assert_eq!(code.trim_start_matches([' ', '\t'].as_ref()),
           "function noodle() { ");
// 更短的等效形式: &[' ', '\t'][..]4
```

⁴从 Rust 1.51.0 开始，通常可以使用更简短的形式，即 `&[' ', '\t']`。——译者注

如果不这么做，则 Rust 会误以为这是固定大小数组类型 `&[char; 2]`。遗憾的是，`&[char; 2]` 不是有效的模式类型。

在标准库本身的代码中，模式就是实现了 `std::str::Pattern` 特型的任意类型。`Pattern` 的细节还不稳定，所以你不能在稳定版的 Rust 中为自己的类型实现它。但是，将来要支持正则表达式和其他复杂模式也很容易。Rust 可以保证现在支持的模式类型将来仍会继续有效。

17.3.7 搜索与替换

Rust 提供了一些可以在切片中搜索某些模式并可能将其替换成新文本的方法。

`slice.contains(pattern)`（包含）

如果 `slice` 包含 `pattern` 的匹配项，就返回 `true`。

`slice.starts_with(pattern)`（以 `pattern` 开头）和
`slice.ends_with(pattern)`（以 `pattern` 结尾）

如果 `slice` 的起始文本或结尾文本与 `pattern` 相匹配，就返回 `true`。

```
assert!("2017".starts_with(char::is_numeric));
```

`slice.find(pattern)`（查找）和
`slice.rfind(pattern)`（右起查找）

如果 `slice` 包含 `pattern` 的匹配项，就返回 `Some(i)`，其中的 `i` 是模式出现的字节偏移量。`find` 方法会返回第一个匹配项，`rfind` 方法则返回最后一个。

```
let quip = "We also know there are known unknowns";
assert_eq!(quip.find("know"), Some(8));
assert_eq!(quip.rfind("know"), Some(31));
assert_eq!(quip.find("ya know"), None);
assert_eq!(quip.rfind(char::is_uppercase), Some(0));
```

`slice.replace(pattern, replacement)`（替换）

返回新的 `String`，它是通过用 `replacement` 急性⁵替换 `pattern` 的所有匹配项而形成的：

⁵ 急性（eagerly）是惰性（lazily）的反义词。——译者注

```
assert_eq!("The only thing we have to fear is fear itself"
          .replace("fear", "spin"),
          "The only thing we have to spin is spin itself");

assert_eq!("`Borrow` and `BorrowMut`"
          .replace(|ch:char| !ch.is_alphanumeric(), ""),
          "BorrowandBorrowMut");
```

因为替换是急性完成的，所以 `.replace()` 在彼此重叠的几个匹配段上的行为可能令人惊讶。这里有 4 个匹配 `"aba"` 模式的实

例，但在替换了第一个和第三个之后，第二个和第四个就不再匹配了。

```
assert_eq!("cabababababbage"
           .replace("aba", "***"),
           "c***b***babbage")
```

`slice.replacen(pattern, replacement, n)`（替换 n 次）

与上一个方法类似，但最多替换前 n 个匹配项。

17.3.8 遍历文本

标准库提供了几种对切片的文本进行迭代的方法。图 17-3 展示了一些示例。

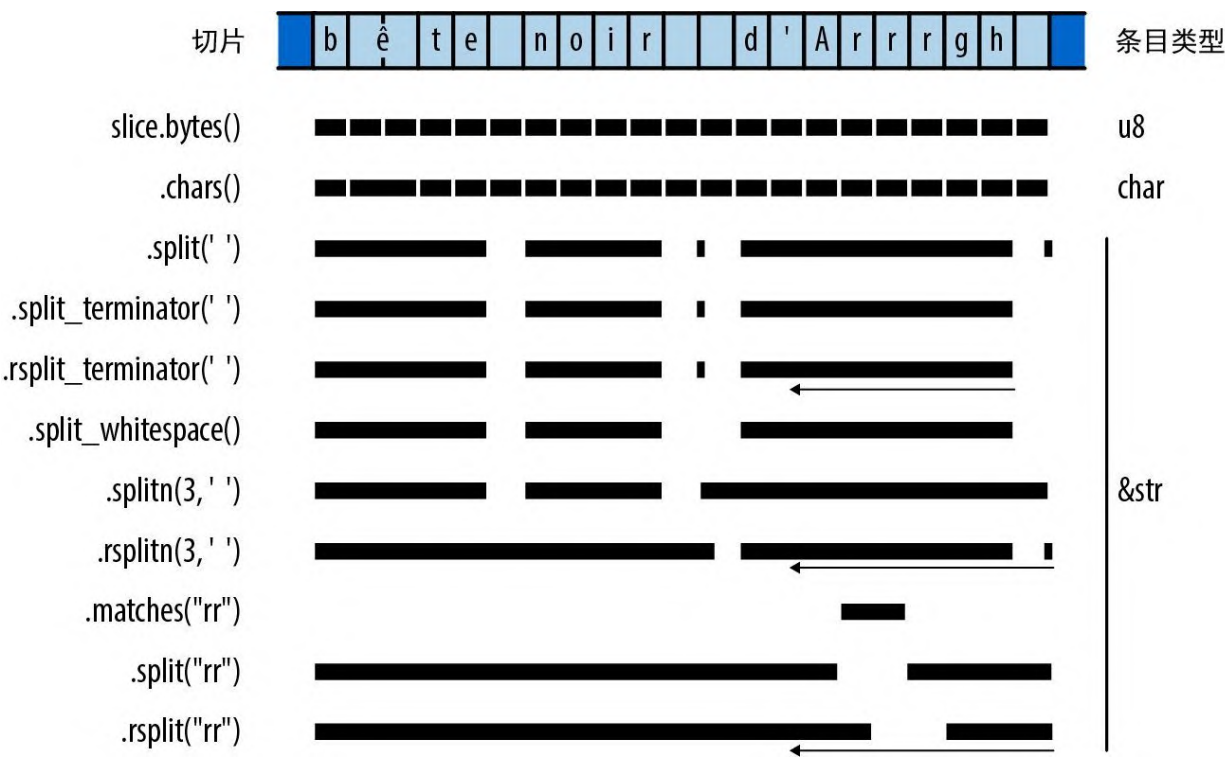


图 17-3：迭代切片的一些方法

`split`（拆分）和 `match`（匹配）系列方法是互补的：拆分取的是匹配项之间的范围。

这些方法中大多数会返回可逆的迭代器（也就是说，它们实现了 `DoubleEndedIterator`）：调用它们的 `.rev()` 适配器方法会为你提供一个迭代器，该迭代器会生成相同的条目，只是顺序相反。

`slice.chars()`（字符迭代器）

返回访问 `slice` 中各个字符的迭代器。

`slice.char_indices()`（字符及其偏移量迭代器）

返回访问 `slice` 中各个字符及其字节偏移量的迭代器：

```
assert_eq!("élan".char_indices().collect::<Vec<_>>(),
           vec![(0, 'é'), // 有一个双字节UTF-8编码
                (2, 'l'),
                (3, 'a'),
                (4, 'n')]);
```

请注意，这并不等同于 `.chars().enumerate()`，因为本方法提供的是每个字符在切片中的字节偏移量，而不仅仅是字符的序号。

`slice.bytes()`（字节迭代器）

返回访问 `slice` 中各字节的迭代器，对外暴露 UTF-8 编码细节。

```
assert_eq!("élan".bytes().collect::<Vec<_>>(),
           vec![195, 169, b'l', b'a', b'n']);
```

`slice.lines()`（文本行迭代器）

返回访问 `slice` 中各行的迭代器。各行以 `"\n"` 或 `"\r\n"` 结尾。生成的每个条目都是从 `slice` 中借入的 `&str`。这些条目不包括行的终止字符。

`slice.split(pattern)`（拆分）

返回一个迭代器，该迭代器会迭代 `slice` 中由 `pattern` 匹配项分隔开的各个部分。这会在紧邻的两个匹配项之间、位于 `slice` 开头的匹配项与头部之间，以及结尾的匹配项与尾部之间生成空字符串。

如果 `pattern` 是 `&str`，则返回的迭代器不可逆，因为这类模式会根据不同的扫描方向生成不同的匹配序列，但可逆迭代器不允许这种行为。可以改用接下来要讲的 `rsplit` 方法。

`slice.rsplit(pattern)`（右起拆分）

与上一个方法类似，但此方法会从尾到头扫描 `slice`，并按该顺序生成匹配项。

`slice.split_terminator(pattern)`（终结符拆分）和
`slice.rsplit_terminator(pattern)`（右起终结符拆分）

与刚刚讲过的拆分方法类似，但这两个方法会把模式视为终结符，而不是分隔符：如果 `pattern` 在 `slice` 的末尾匹配上了，则迭代器不会像 `split` 和 `rsplit` 那样生成表示匹配项和切片末尾之间空字符串的空切片。例如：

```
// 这里把':'字符视为分隔符。注意结尾的""（空串）
assert_eq!("jimb:1000:Jim Blandy:".split(':').collect::<Vec<_>>(),
           vec!["jimb", "1000", "Jim Blandy", ""]);

// 这里把'\n'字符视为终结符
assert_eq!("127.0.0.1 localhost\n\
           127.0.0.1 www.reddit.com\n"
           .split_terminator('\n').collect::<Vec<_>>(),
           vec!["127.0.0.1 localhost",
                "127.0.0.1 www.reddit.com"]);
// 注意，没有结尾的""！
```

`slice.splitn(n, pattern)`（拆分为 n 片）和
`slice.rsplitn(n, pattern)`（右起拆分为 n 片）

与 `split` 和 `rsplit` 类似，但这两个方法会把字符串分成最多 `n` 个切片，拆分位置位于 `pattern` 的第 `n-1` 个 (`split`) 或倒数第 `n-1` 个 (`rsplit`) 匹配项处。

`slice.split_whitespace()` (按空白字符拆分) 和
`slice.split_ascii_whitespace()` (按 ASCII 空白字符拆分)

返回访问 `slice` 中以空白字符分隔的各部分的迭代器。这两个方法会把连续多个空白字符视为单个分隔符。忽略尾部空白字符。

`split_whitespace` 方法会使用 Unicode 的空白字符定义，由 `char` 上的 `is_whitespace` 方法实现。

`split_ascii_whitespace` 方法则会使用只识别 ASCII 空白字符的 `char::is_ascii_whitespace`。

```
let poem = "This is just to say\n\
            I have eaten\n\
            the plums\n\
            again\n";

assert_eq!(poem.split_whitespace().collect::<Vec<_>>(),
           vec!["This", "is", "just", "to", "say",
                "I", "have", "eaten", "the", "plums",
                "again"]);
```

`slice.matches(pattern)` (匹配项)

返回访问 `slice` 中 `pattern` 匹配项的迭代器。
`slice.rmatches(pattern)` 也一样，但会从尾到头迭代。

`slice.match_indices(pattern)` (匹配项及其偏移量) 和
`slice.rmatch_indices(pattern)` (右起匹配项及其偏移量)

和上一个方法很像，但这两个方法生成的条目是 (`offset`, `match`) 值对，其中 `offset` 是匹配的起始字节的偏移量，而 `match` 是匹配到的切片。

17.3.9 修剪

修剪字符串就是从字符串的开头或结尾移除文本（通常是空白字符）。修剪常用于清理从文件中读取的输入，在此文件中，用户可能为了易读性而添加了文本缩进，或者不小心在一行中留下了尾随空白字符。

`slice.trim()`（修剪）

返回略去了任何前导空白字符和尾随空白字符的 `slice` 的子切片。`slice.trim_start()` 只会略去前导空白字符，`slice.trim_end()` 只会略去尾随空白字符。

```
assert_eq!("\t*.rs".trim(), "*.rs");
assert_eq!("\t*.rs".trim_start(), "*.rs ");
assert_eq!("\t*.rs".trim_end(), "\t*.rs");
```

`slice.trim_matches(pattern)`（按匹配修剪）

返回 `slice` 的子切片，该子切片从开头和结尾略去了 `pattern` 的所有匹配项。`trim_start_matches` 方法和 `trim_end_matches` 方法只会对匹配的前导内容或尾随内容执行修剪操作。

```
assert_eq!("001990".trim_start_matches('0'), "1990");
```

`slice.strip_prefix(pattern)`（剥离前缀）和
`slice.strip_suffix(pattern)`（剥离后缀）

如果 `slice` 以 `pattern` 开头，则 `strip_prefix` 会返回一个 `Some`，其中携带了移除匹配文本之后的切片。否则，它会返回 `None`。`strip_suffix` 方法与此类似，但会检查字符串末尾的匹配项。

与 `trim_start_matches` 和 `trim_end_matches` 类似，但这里的两个方法会返回 `Option`，并且只会移除一个匹配 `pattern`

的副本。

```
let slice = "banana";
assert_eq!(slice.strip_suffix("na"),
           Some("bana"))
```

17.3.10 字符串的大小写转换

`slice.to_uppercase()` 方法和 `slice.to_lowercase()` 方法会返回一个新分配的字符串，其中包含已转为大写或小写的 `slice` 文本。结果的长度可能与 `slice` 不同，有关详细信息，请参阅 17.2.3 节。

17.3.11 从字符串中解析出其他类型

Rust 为“从字符串解析出值”和“生成值的文本表示”提供了一些标准特型。

如果一个类型实现了 `std::str::FromStr` 特型，那它就提供了一种从字符串切片中解析出值的标准方法：

```
pub trait FromStr: Sized {
    type Err;
    fn from_str(s: &str) -> Result<Self, Self::Err>;
}
```

所有常见的机器类型都实现了 `FromStr`：

```
use std::str::FromStr;

assert_eq!(usize::from_str("3628800"), Ok(3628800));
assert_eq!(f64::from_str("128.5625"), Ok(128.5625));
assert_eq!(bool::from_str("true"), Ok(true));

assert!(f64::from_str("not a float at all").is_err());
assert!(bool::from_str("TRUE").is_err());
```

`char` 类型也实现了 `FromStr`，用于解析只有一个字符的字符串：

```
assert_eq!(char::from_str("é"), Ok('é'));
assert!(char::from_str("abcdefg").is_err());
```

`std::net::IpAddr` 类型，即包含 IPv4 或 IPv6 互联网地址的 enum，同样实现了 `FromStr`：

```
use std::net::IpAddr;

let address = IpAddr::from_str("fe80::0000:3ea9:f4ff:fe34:7a50")?;
assert_eq!(address,
            IpAddr::from([0xfe80, 0, 0, 0, 0x3ea9, 0xf4ff, 0xfe34,
                          0x7a50]));
```

字符串切片有一个 `parse` 方法，该方法可以将切片解析为你想要的任何类型——只要它实现了 `FromStr`。与 `Iterator::collect` 一样，有时需要明确写出想要的类型，因此用 `parse` 不一定比直接调用 `from_str` 可读性强。

```
let address = "fe80::0000:3ea9:f4ff:fe34:7a50".parse::<IpAddr>()?;
```

17.3.12 将其他类型转换为字符串

将非文本值转换为字符串的方法主要有以下 3 种。

- 那些具有人类可读的自然打印形式的类型可以实现 `std::fmt::Display` 特型，该特型允许在 `format!` 宏的格式中使用 `{}` 格式说明符：

```
assert_eq!(format!("{}", wow, "doge"), "doge, wow");
assert_eq!(format!("{}", true), "true");
assert_eq!(format!("{:.3}, {:.3})", 0.5, f64::sqrt(3.0)/2.0),
            "(0.500, 0.866)");

// 使用上一个例子中的`address`
let formatted_addr: String = format!("{}", address);
assert_eq!(formatted_addr, "fe80::3ea9:f4ff:fe34:7a50");
```

Rust 的所有机器数值类型都实现了 `Display`，字符、字符串和切片也是如此。智能指针类型 `Box<T>`、`Rc<T>` 和 `Arc<T>` 也实现了 `Display`（只要 `T` 本身实现了 `Display`）：它们的显示形式就只是其引用目标的显示形式而已。而像 `Vec` 和 `HashMap` 这样的容器则没有实现 `Display`，因为这些类型没有人类可读的单一自然形式。

- 如果一个类型实现了 `Display`，那么标准库就会自动为它实现 `std::str::ToString` 特型，当你不需要 `format!` 的灵活性时，使用此特型的唯一方法 `to_string` 会更方便：

```
// 接续前面的例子
assert_eq!(address.to_string(), "fe80::3ea9:f4ff:fe34:7a50");
```

Rust 在引入 `Display` 之前就已经引入 `ToString` 特型了，但该特型不太灵活。对于自己的类型，你通常应该实现 `Display` 而非 `ToString`。

- 标准库中的每个公共类型都实现了 `std::fmt::Debug`，这个特型会接受一个值并将其格式化为对程序员有用的字符串。用 `Debug` 生成字符串的最简单方法是使用 `format!` 宏的 `{:?}` 格式说明符：

```
// 接续前面的例子
let addresses = vec![address,
                     IpAddr::from_str("192.168.0.1")?];
assert_eq!(format!("{:?}", addresses),
           "[fe80::3ea9:f4ff:fe34:7a50, 192.168.0.1]");
```

对于本身实现了 `Debug` 的任何类型 `T`，这里利用了 `Vec<T>` 对 `Debug` 的通用实现。Rust 的所有集合类型都有这样的实现。

你也应该为自己的类型实现 `Debug`。通常，最好让 Rust 派生一个实现，就像我们在第 12 章中对 `Complex` 类型所做的那样：

```
#[derive(Copy, Clone, Debug)]
struct Complex { re: f64, im: f64 }
```

`format!` 及其相关宏在把值格式化为文本时用到了很多格式化特型，`Display` 和 `Debug` 只是其中的两个。17.4 节会介绍其他特型，并解释如何实现它们。

17.3.13 借用其他类似文本的类型

可以通过以下两种方式借用切片的内容。

- 切片和 `String` 都实现了 `AsRef<str>`、`AsRef<[u8]>`、`AsRef<Path>` 和 `AsRef<OsStr>`。许多标准库函数会使用这些特型作为参数类型的限界，因此可以直接将切片和字符串传给它们，即便它们真正想要的是其他类型。有关详细解释，请参阅 13.7 节。
- 切片和字符串还实现了 `std::borrow::Borrow<str>` 特型。`HashMap` 和 `BTreeMap` 会借助 `Borrow` 令 `String` 很好地用作表中的键。有关详细信息，请参阅 13.8 节。

17.3.14 以 UTF-8 格式访问文本

获取表示文本的那些字节有两个主要方法，具体取决于你是想获取字节的所有权还是只想借用它们。

`slice.as_bytes()`（用作字节切片）

把 `slice` 的字节借入为 `&[u8]`。由于这不是可变引用，因此 `slice` 可以假定其字节将保持为格式良好的 UTF-8。

`string.into_bytes()`（转为字节切片）

获取 `string` 的所有权并按值返回字符串字节的 `Vec<u8>`。这是一个开销极低的转换，因为它只是移动了字符串一直用作缓冲区的 `Vec<u8>`。由于 `string` 已经不复存在，因此这些字节无须继续保持为格式良好的 UTF-8，而调用者可以随意修改 `Vec<u8>`。

17.3.15 从 UTF-8 数据生成文本

如果你有一个包含 UTF-8 数据的字节块，那么有几个方法可以将其转换为 `String` 或切片，但具体用哪个取决于你希望如何处理错误。

`str::from_utf8(byte_slice)` (来自 `utf8` 切片)

接受 `&[u8]` 字节切片并返回 `Result`：如果 `byte_slice` 包含格式良好的 UTF-8，就返回 `Ok(&str)`，否则，返回错误。

`String::from_utf8(vec)` (来自 `utf8` 向量)

尝试从按值传递的 `Vec<u8>` 中构造字符串。如果 `vec` 持有格式良好的 UTF-8，那么 `from_utf8` 就会返回 `Ok(string)`，其中 `string` 会取得 `vec` 的所有权并将其用作缓冲区。此过程不会发生堆分配或文本复制。

如果这些字节不是有效的 UTF-8，则返回 `Err(e)`，其中 `e` 是 `FromUtf8Error` 型的错误值。调用 `e.into_bytes()` 会返回原始向量 `vec`，因此当转换失败时它并不会丢失：

```
let good_utf8: Vec<u8> = vec![0xe9, 0x8c, 0x86];
assert_eq!(String::from_utf8(good_utf8).ok(),
Some("鏄".to_string()));

let bad_utf8: Vec<u8> = vec![0x9f, 0xf0, 0xa6, 0x80];
let result = String::from_utf8(bad_utf8);
assert!(result.is_err());
// 由于String::from_utf8失败了，因此它不会消耗原始向量，
// 而是通过错误值把原始向量原原本本地还给了我们
assert_eq!(result.unwrap_err().into_bytes(),
vec![0x9f, 0xf0, 0xa6, 0x80]);
```

`String::from_utf8_lossy(byte_slice)` (来自 `utf8`, 宽松版)

尝试从 `&[u8]` 共享字节切片构造一个 `String` 或 `&str`。此转换总会成功, 任何格式错误的 UTF-8 都会被 Unicode 代用字符替换。返回值是一个 `Cow<str>`, 如果它包含格式良好的 UTF-8, 就会直接从 `byte_slice` 借用 `&str`, 否则会拥有一个新分配的 `String`, 其中格式错误的字节会被代用字符替换。因此, 当 `byte_slice` 是格式良好的 UTF-8 时, 不会发生堆分配或复制。17.3.16 节会更详细地讨论 `Cow<str>`。

`String::from_utf8_unchecked(vec)` (来自 `utf8`, 不检查版)

如果你确信此 `Vec<u8>` 包含格式良好的 UTF-8, 那就可以调用这个不安全的函数。此方法只是将 `vec` 包装为一个 `String` 并返回它, 根本不检查字节。你有责任确保没有将格式错误的 UTF-8 引入系统, 这就是此函数被标记为 `unsafe` 的原因。

`str::from_utf8_unchecked(byte_slice)` (来自 `utf8`, 不检查版)

与上一个方法类似, 但此方法会接受 `&[u8]` 并将其作为 `&str` 返回, 而不检查它是否包含格式良好的 UTF-8。与 `String::from_utf8_unchecked` 一样, 你有责任确保 `byte_slice` 是安全的。

17.3.16 推迟分配

假设你想让程序向用户打招呼。在 Unix 上, 可以这样写:

```
fn get_name() -> String {
    std::env::var("USER") // 在Windows上要改成"USERNAME"
        .unwrap_or("whoever you are".to_string())
}

println!("Greetings, {}!", get_name());
```

对于 Unix 用户，这个程序会根据用户名向他们问好。对于 Windows 用户和无名用户，它提供了备用文本。

`std::env::var` 函数会返回一个 `String`——并且有充分的理由这样做，所以我们不会在这里讨论。但这意味着备用文本也必须作为 `String` 返回。这不太理想：当 `get_name` 返回静态字符串时，根本没必要分配内存。

问题的关键在于，`get_name` 的返回值有时应该是拥有型 `String`，有时则应该是 `&'static str`，并且在运行程序之前我们无法知道会是哪一个。这种动态的特点预示着应该考虑使用 `std::borrow::Cow`，这个写入时克隆类型既可以持有拥有型数据也可以持有借入的数据。

正如 13.12 节所述，`Cow<'a, T>` 是一个具有 `Owned` 和 `Borrowed` 两个变体的枚举。`Borrowed` 持有一个引用 `&'a T`，而 `Owned` 持有 `&T` 的拥有型版本：对于 `&str` 是 `String`，对于 `&[i32]` 是 `Vec<i32>`，等等。无论是 `Owned` 还是 `Borrowed`，`Cow<'a, T>` 总能生成一个 `&T` 供你使用。事实上，`Cow<'a, T>` 可以解引用为 `&T`，其行为类似于一种智能指针。

更改 `get_name` 以返回 `Cow`，结果如下所示：

```
use std::borrow::Cow;

fn get_name() -> Cow<'static, str> {
    std::env::var("USER")
        .map(|v| Cow::Owned(v))
        .unwrap_or(Cow::Borrowed("whoever you are"))
}
```

如果读取 "USER" 环境变量成功，那么 `map` 就会将结果 `String` 作为 `Cow::Owned` 返回。如果失败，则 `unwrap_or` 会将其静态 `&str` 作为 `Cow::Borrowed` 返回。调用者可以保持不变：

```
println!("Greetings, {}!", get_name());
```

只要 `T` 实现了 `std::fmt::Display` 特型，显示 `Cow<'a, T>` 的结果就和显示 `T` 的结果是一样的。

当你可能需要也可能不需要修改借用的某些文本时，`Cow` 也很有用。不需要修改时，可以继续借用。但是 `Cow` 名副其实的写入时克隆行为可以根据需要为你提供一个拥有型的、可变的值副本。`Cow` 的 `to_mut` 方法会确保 `Cow` 是 `Cow::Owned`，必要时会应用该值的 `ToOwned` 实现，然后返回对该值的可变引用。

因此，如果你发现某些用户（但不是全部）拥有他们更想使用的头衔，就可以这样写：

```
fn get_title() -> Option<&'static str> { ... }

let mut name = get_name();
if let Some(title) = get_title() {
    name.to_mut().push_str(", ");
    name.to_mut().push_str(title);
}

println!("Greetings, {}!", name);
```

这可能会生成如下输出：

```
$ cargo run
Greetings, jimb, Esq.!
$
```

这样做的好处是，如果 `get_name()` 返回一个静态字符串并且 `get_title` 返回 `None`，那么 `Cow` 只是将静态字符串透传到 `println!`。你已经设法把内存分配推迟到了确有必要的时候，并且代码仍然一目了然。

由于 `Cow` 经常用于字符串，因此标准库对 `Cow<'a, str>` 有一些特殊支持。它提供了来自 `String` 和 `&str` 的 `From` 和 `Into` 这两个转换特型，这样就可以更简洁地编写 `get_name` 了：

```
fn get_name() -> Cow<'static, str> {
    std::env::var("USER")
        .map(|v| v.into())
        .unwrap_or("whoever you are".into())
}
```

Cow<'a, str> 还实现了 std::ops::Add 和 std::ops::AddAssign，因此要将标题添加到名称中，可以这样写：

```
if let Some(title) = get_title() {
    name += ", ";
    name += title;
}
```

或者，因为 String 可以作为 write! 宏的目标，所以也可以这样写：

```
use std::fmt::Write;

if let Some(title) = get_title() {
    write!(name.to_mut(), ", {}", title).unwrap();
}
```

和以前一样，在尝试修改 Cow 之前不会发生内存分配。

请记住，并非每个 Cow<..., str> 都必须是 'static：可以使用 Cow 借用以前计算好的文本，直到需要复制为止。

17.3.17 把字符串当作泛型集合

String 同时实现了 std::default::Default 和 std::iter::Extend：default 返回空字符串，而 extend 可以把字符、字符串切片、Cow<..., str> 或字符串追加到一个字符串尾部。这与 Rust 的其他集合类型（如 Vec 和 HashMap）为其泛型构造模式（如 collect 和 partition）实现的特型组合是一样的。

`&str` 类型也实现了 `Default`，返回一个空切片。这在某些极端情况下很方便，比如，这样可以让包含字符串切片的结构派生于 `Default` (`#[derive(Default)]`)。

17.4 格式化各种值

本书一直在使用像 `println!` 这样的文本格式化宏：

```
println!("{:.3}µs: relocated {} at {:#x} to {:#x}, {} bytes",
         0.84391, "object",
         140737488346304_usize, 6299664_usize, 64);
```

上述调用会生成如下输出：

```
0.844µs: relocated object at 0x7fffffffddcc0 to 0x602010, 64 bytes
```

字符串字面量可以用作输出模板：模板中的每个 `{...}` 都会被其后跟随的某个参数的格式化形式替换。模板字符串必须是常量，以便 Rust 在编译期根据参数的类型检查它。每个参数在检查时必须都用到，否则 Rust 就会报告编译期错误。

以下几个标准库特性中都有这种用于格式化字符串的小型语言。

- `format!` 宏会用它来构建 `String`。
- `println!` 宏和 `print!` 宏会将格式化后的文本写入标准输出流。
- `writeln!` 宏和 `write!` 宏会将格式化后的文本写入指定的输出流。
- `panic!` 宏会使用它构建一个信息丰富的异常终止描述。

Rust 格式化工具的设计是开放式的。你可以通过实现 `std::fmt` 模块的格式化特型来扩展这些宏以支持自己的类型。也可以使用 `format_args!` 宏和 `std::fmt::Arguments` 类型来让自己的函数和宏支持格式化语言。

格式化宏总会借入对其参数的共享引用，但永远不会拥有或修改它们。

模板的 `{...}` 形式称为**格式参数**，具体形式为 `{which:how}`。
Which 和 *how* 都是可选的，很多时候用 `{}` 就行。

which（哪个）值用于选择模板后面的哪个实参应该取代该形参的位置。可以按索引或名称选择实参。没有 *which* 值的形参只会简单地
从左到右与实参配对。

how（如何）值表示应如何格式化参数：如何填补、精度如何、数值
基数等。如果存在 *how*，则需要写上前面的冒号。

表 17-4 给出了一些示例。

表 17-4：格式化字符串示例

模板字符串	参数列表	结果
"number of {}: {}"	"elephants", 19	"number of elephants: 19"
"from {1} to {0}"	"the grave", "the cradle"	"from the cradle to the grave"
"v = {:?}"	vec![0,1,2,5,12,29]	"v = [0, 1, 2, 5, 12, 29]"
"name = {:?}"	"Nemo"	"name = \"Nemo\""
"{:8.2} km/s"	11.186	" 11.19 km/s"
"{:20} {:02x} {:02x}"	"adc #42", 105, 42	"adc #42 69 2a"

模板字符串	参数列表	结果
"{1:02x} {2:02x} {0}"	"adc #42", 105, 42	"69 2a adc #42"
"{lsb:02x} {msb:02x} {insn}"	insn="adc #42", lsb=105, msb=42	"69 2a adc #42"
"{:02?}"	[110, 11, 9]	"[110, 11, 09]"
"{:02x?}"	[110, 11, 9]	"[6e, 0b, 09]"

如果要在输出中包含 { 或 } 字符，可将模板中的这些字符连写两个。

```
assert_eq!(format!("{}", {{a, c}} < {{a, b, c}}"),
           "{a, c} < {a, b, c}");
```

17.4.1 格式化文本值

当格式化像 `&str` 或 `String`（将 `char` 视为单字符字符串）这样的文本类型时，参数的 *how* 值有几个部分，都是可选的。

- **文本长度限制。**如果参数比这个值长，Rust 就会截断它。如果未指定限制，Rust 就使用全文。
- **最小字段宽度。**在完成所有截断之后，如果参数比这个值短，Rust 就会在右边（默认）用空格（默认）填补它以让字段达到这个宽度。如果省略，Rust 则不会填补参数。
- **对齐方式。**如果参数需要填补空白以满足最小字段宽度，那么这个值表示应将文本放置在字段中的什么位置。`<`、`^` 和 `>` 分别会将文本放在开头、中间和结尾。

- 在此填补过程中使用的填补字符。如果省略，Rust 就会使用空格。如果指定了填补字符，则必须同时指定对齐方式。

表 17-5 举例说明了如何编写这些格式字符串及其实际效果。所有这些示例都使用了相同的八字符参数 "bookends"。

表 17-5：文本的格式化字符串指令

使用的特性	模板字符串	结果
默认	"{}"	"bookends"
最小字段宽度	"{:4}" "{:12}"	"bookends" "bookends "
文本长度限制	"{: .4}" "{: .12}"	"book" "bookends"
字段宽度、长度限制	"{:12.20}" "{:4.20}" "{:4.6}" "{:6.4}"	"bookends " "bookends" "booken" "book "
左对齐，宽度	"{:<12}"	"bookends "
居中，宽度	"{: ^12}"	" bookends "
右对齐，宽度	"{: >12}"	" bookends"
用 '=' 填补，居中，宽度	"{: = ^12}"	"==bookends=="

使用的特性	模板字符串	结果
用 '*' 填补，右对齐，宽度，限制	"{:*>12.4}"	"*****book"

Rust 的格式化程序对宽度的处理方式比较“简陋”：它假设每个字符占据一列，而不会考虑组合字符、半角片假名、零宽度空格或 Unicode 的其他乱七八糟的情况。例如：

```
assert_eq!(format!("{:4}", "th\u{e9}"), "th\u{e9} ");
assert_eq!(format!("{:4}", "the\u{301}"), "the\u{301}");
```

尽管 Unicode 规定这两个字符串都等效于 "thé"，但 Rust 的格式化程序可不知道像 '\u{301}' 这样的字符（组合重音符）需要做特殊处理。它正确地填补了第一个字符串，但假设第二个字符串是 4 列宽并且不需要填补。尽管很容易看出 Rust 该如何在这种特定情况下进行改进，但要支持所有 Unicode 脚本的真正多语言文本格式化是一项艰巨的任务。最好依靠所在平台的用户界面工具包来处理，或许也可以通过生成 HTML 和 CSS，让 Web 浏览器来处理。有一个流行的 crate (unicode-width) 可以部分处理这个问题。

除了 &str 和 String，你也可以直接向格式化宏传入带有文本型引用目标的智能指针类型，比如 Rc<String> 或 Cow<'a, str>。

由于文件名路径不一定是格式良好的 UTF-8，因此 std::path::Path 不完全是文本类型，不能将 std::path::Path 直接传给格式化宏。不过，Path 有个 display 方法会返回一个供格式化的 UTF-8 值，以适合所在平台的方式解决问题。

```
println!("processing file: {}", path.display());
```

17.4.2 格式化数值

当格式化参数具有 `usize` 或 `f64` 之类的数值类型时，参数的 `how` 值可以有如下几个部分，它们全是可选的。

- **填补与对齐**，它们和对文本类型的含义一样。
- **+** 字符，要求始终显示数值的符号，即使相应参数是正数。
- **#** 字符，要求加显式基数前缀，比如 `0x` 或 `0b`。参见稍后要讲的“进制符号”那一项。
- **0** 字符，要求通过在数值中包含前导零（而不是通常的填补方式）来满足最小字段宽度。
- **最小字段宽度**。如果格式化后的数值没有这么宽，那么 Rust 会在左侧（默认）用空格（默认）填补它以构成给定宽度的字段。
- **浮点参数的精度**，指示 Rust 应在小数点后包含多少位数字。Rust 会根据需要进行舍入或零扩展以生成要求的小数位。如果省略精度，那么 Rust 会尝试使用尽可能少的数字来准确表示该值。对于整数类型的参数，精度会被忽略。
- **进制符号**。对于整数类型，二进制是 `b`，八进制是 `o`，十六进制是小写字母 `x` 或大写字母 `X`。如果包含 `#` 字符，则它们会包含显式的 Rust 风格的基数前缀 `0b`、`0o`、`0x` 或 `0X`。对于浮点类型，`e` 或 `E` 的基数需要科学记数法，具有归一化系数，使用 `e` 或 `E` 作为指数。如果不指定任何进制符号，则 Rust 会将数值格式化为十进制。

表 17-6 展示了格式化 `i32` 值 `1234` 的一些示例。

表 17-6：格式化整数的字符串指令

使用的特性	模板字符串	结果
默认	"{}"	"1234"
强制正负号	"{:+}"	"+1234"

使用的特性	模板字符串	结果
最小字段宽度	"{:12}" "{:2}"	"1234" "1234"
正负号，宽度	"{:+12}"	"+1234"
前导零，宽度	"{:012}"	"000000001234"
正负号，前导零，宽度	"{:+012}"	"+00000001234"
左对齐，宽度	"{:<12}"	"1234"
居中，宽度	"{: ^12}"	"1234"
右对齐，宽度	"{:>12}"	"1234"
左对齐，正负号，宽度	"{:<+12}"	"+1234"
居中，正负号，宽度	"{: ^+12}"	"+1234"
右对齐，正负号，宽度	"{:>+12}"	"+1234"
用 '=' 填补，居中，宽度	"{: = ^12}"	"====1234===="
二进制表示法	"{:b}"	"10011010010"
宽度，八进制表示法	"{:12o}"	"2322"

使用的特性	模板字符串	结果
正负号，宽度，十六进制表示法	"{:+12x}"	" +4d2"
正负号，宽度，用大写数字的十六进制	"{:+12X}"	" +4D2"
正负号，显式基数前缀，宽度，十六进制	"{:+#12x}"	" +0x4d2"
正负号，基数，前导零，宽度，十六进制	"{:+#012x}" "{:+#06x}"	" +0x0000004d2" "+0x4d2"

如最后两个例子所示，最小字段宽度适用于整个数值、正负号、基数前缀等。

负数总是包含它们的符号。结果和“强制正负号”例子中展示的一样。

当你要求加前导零时，就会忽略对齐和填补字符，因为要用零扩展数值以填补整个字段。

使用参数 1234.5678，可以展示对浮点类型的格式化效果，如表 17-7 所示。

表 17-7：格式化浮点数的字符串指令

使用的特性	模板字符串	结果
默认	"{}"	"1234.5678"
精度	"{: .2}" "{: .6}"	"1234.57" "1234.567800"

使用的特性	模板字符串	结果
最小字段宽度	"{:12}"	" 1234.5678"
最小宽度, 精度	"{:12.2}" "{:12.6}"	" 1234.57" " 1234.567800"
前导零, 最小宽度, 精度	"{:012.6}"	"01234.567800"
科学记数法	"{:e}"	"1.2345678e3"
科学记数法, 精度	"{: .3e}"	"1.235e3"
科学记数法, 最小宽度, 精度	"{:12.3e}" "{:12.3E}"	" 1.235e3" " 1.235E3"

17.4.3 格式化其他类型

除了字符串和数值, 还可以格式化标准库中的其他几种类型。

- 错误类型全都可以直接格式化, 从而很容易地将它们包含在错误消息中。每种错误类型都应该实现 `std::error::Error` 特型, 该特型扩展了默认格式化特型 `std::fmt::Display`。因此, 任何实现了 `Error` 的类型都可以格式化。
- 可以格式化 `std::net::IpAddr`、`std::net::SocketAddr` 等互联网协议地址类型。
- 布尔值 `true` 和 `false` 也可以被格式化, 虽然它们通常不是直接呈现给最终用户的最佳格式。

对上述类型来说, 应该使用与字符串相同类型的格式参数。长度限制、字段宽度和对齐方式控制都会如预期般工作。

17.4.4 格式化值以进行调试

为了帮助调试和记录日志，`{:?}` 参数能以对程序员有帮助的方式格式化 Rust 标准库中的任何公共类型。你可以使用它来检查向量、切片、元组、哈希表、线程和其他数百种类型。

例如，你可以编写如下代码：

```
use std::collections::HashMap;
let mut map = HashMap::new();
map.insert("Portland", (45.5237606, -122.6819273));
map.insert("Shanghai", (31.230416, 121.473701));
println!("{:?}", map);
```

这会打印出如下内容：

```
{"Shanghai": (31.230416, 121.473701), "Portland": (45.5237606, -122.6819273)}
```

`HashMap` 和 `(f64, f64)` 类型都知道该如何格式化自身，你无须额外做什么。

如果你在格式参数中包含了 `#` 字符，Rust 就会优美地打印出该值。将上面那行代码改成 `println!("{:#?}", map)` 会输出如下内容：

```
{
  "Shanghai": (
    31.230416,
    121.473701
  ),
  "Portland": (
    45.5237606,
    -122.6819273
  )
}
```

这些输出的精确格式并不能保证始终如一，比如升级 Rust 版本后就可能发生变化。

供调试用的格式化通常会以十进制打印数值，但可以在问号前放置一个 `x` 或 `X` 以请求十六进制，并且会遵守前导零和字段宽度语法。例如，可以像下面这样写：

```
println!("ordinary: {:02?}", [9, 15, 240]);  
println!("hex:      {:02x?}", [9, 15, 240]);
```

这会打印出如下内容：

```
ordinary: [09, 15, 240]  
hex:      [09, 0f, f0]
```

如前所述，你可以用 `#[derive(Debug)]` 语法让自己的类型支持 `{:?}`：

```
#[derive(Copy, Clone, Debug)]  
struct Complex { re: f64, im: f64 }
```

有了这个定义，就可以使用 `{:?}` 格式来打印 `Complex` 值了：

```
let third = Complex { re: -0.5, im: f64::sqrt(0.75) };  
println!("{:?}", third);
```

这会打印出如下内容：

```
Complex { re: -0.5, im: 0.8660254037844386 }
```

这对调试来说已经很好了，但如果能用 `{}` 以更传统的形式（如 `-0.5 + 0.8660254037844386i`）打印它们就更好了。17.4.8 会展示如何做到这一点。

17.4.5 格式化指针以进行调试

正常情况下，如果将任何种类的指针传给格式化宏（引用、`Box` 或 `Rc`），宏都会简单地追踪指针并格式化它的引用目标，指针本身并不重要。但是在调试时，查看指针有时很有帮助：地址可以用作单个值

的粗略“名称”，这在检查含有循环或共享指针的结构体时可能很有帮助。

`{:p}` 表示法会将引用、`Box` 和其他类似指针的类型格式化为地址：

```
use std::rc::Rc;
let original = Rc::new("mazurka".to_string());
let cloned = original.clone();
let impostor = Rc::new("mazurka".to_string());
println!("text:      {}, {}, {}",      original, cloned,
impostor);
println!("pointers: {:p}, {:p}, {:p}", original, cloned,
impostor);
```

这会打印出如下内容：

```
text:      mazurka, mazurka, mazurka
pointers: 0x7f99af80e000, 0x7f99af80e000, 0x7f99af80e030
```

当然，具体的指针值每次运行时可能都不一样，但即便如此，比较这些地址也能清晰地看出前两个是对同一个 `String` 的引用，而第三个指向了不同的值。

地址确实看起来是可读性很差的十六进制，因此更精致的展现形式可能会更有用，但 `{:p}` 样式仍然是一种有效的快速解决方案。

17.4.6 按索引或名称引用参数

格式参数可以明确选择它要使用的参数。例如：

```
assert_eq!(format!("{1},{0},{2}", "zeroth", "first", "second"),
           "first,zeroth,second");
```

可以在冒号后包含格式参数：

```
assert_eq!(format!("{2:#06x},{1:b},{0:=>10}", "first", 10, 100),
           "0x0064,1010,====first");
```


还可以按名称选择参数。这能让有许多参数的复杂模板更加清晰易读。例如：

```
assert_eq!(format!("{description:.<25}{quantity:2} @ {price:5.2}",
                    price=3.25,
                    quantity=3,
                    description="Maple Turmeric Latte"),
           "Maple Turmeric Latte..... 3 @ 3.25");
```

（这里的命名型参数类似于 Python 中的关键字参数，但它们只是这些格式化宏的独有特性，而不是 Rust 函数调用语法的一部分。）

可以在单个格式化宏中将索引型参数、命名型参数和位置型（没有索引或名称的）参数混用。位置型参数会从左到右与参数配对，就仿佛索引型参数和命名型参数不存在一样（不参与位置编号）：

```
assert_eq!(format!("{mode} {2} {} {}",
                    "people", "eater", "purple", mode="flying"),
           "flying purple people eater");
```

命名型参数必须出现在列表的末尾。

17.4.7 动态宽度与动态精度

参数的最小字段宽度、文本长度限制和数值精度不必总是固定值，也可以在运行期进行选择。

我们一直在研究类似于下面这个表达式的情况，它会生成在 20 个字符宽的字段中右对齐的字符串 `content`：

```
format!("{:>20}", content)
```

但是，如果想在运行期选择字段宽度，则可以这样写：

```
format!("{:>1$}", content, get_width())
```

将最小字段宽度写成 `1$` 就是在告诉 `format!` 使用第二个参数的值作为宽度。它引用的参数必须是 `usize`。还可以按名称引用参数：

```
format!("{:>width$}", content, width=get_width())
```

同样的方法也适用于文本长度限制：

```
format!("{:>width$.limit$}", content,
        width=get_width(), limit=get_limit())
```

要代替文本长度限制或浮点精度，还可以写成 `*`，表示将下一个位置参数作为精度。下面的代码会把 `content` 裁剪成最多 `get_limit()` 个字符：

```
format!("{:.*}", get_limit(), content)
```

用作精度的参数必须是 `usize`。字段宽度没有对应的语法。

17.4.8 格式化自己的类型

格式化宏会使用 `std::fmt` 模块中定义的一组特型将值转换为文本。通过自行实现这些特型中的一个或多个，就可以让 Rust 的格式化宏来格式化你的类型。

格式参数中的符号指示了其参数类型必须实现的特型，如表 17-8 所示。

表 17-8：格式化字符串指令符号

符号	例子	特型	目的
无	<code>{}</code>	<code>std::fmt::Display</code>	文本、数值、错误：通用特型
<code>b</code>	<code>{bits:#b}</code>	<code>std::fmt::Binary</code>	二进制中的数值

符号	例子	特型	目的
o	{:#5o}	std::fmt::Octal	八进制中的数值
x	{:4x}	std::fmt::LowerHex	十六进制中的数值，小写数字
X	{:016X}	std::fmt::UpperHex	十六进制中的数值，大写数字
e	{:.3e}	std::fmt::LowerExp	科学记数法中的浮点数值
E	{:.3E}	std::fmt::UpperExp	同上，但大写 E
?	{:#?}	std::fmt::Debug	调试视图，适用于开发人员
p	{:p}	std::fmt::Pointer	将指针作为地址，适用于开发人员

当你将 `#[derive(Debug)]` 属性放在类型定义上，以期支持 `{:?}` 格式参数时，其实只是在要求 Rust 替你实现 `std::fmt::Debug` 特型。

这些格式化特型都具有相同的结构，只是名称不同而已。我们将以 `std::fmt::Display` 为代表来讲解：

```
trait Display {
    fn fmt(&self, dest: &mut std::fmt::Formatter)
        -> std::fmt::Result;
}
```

`fmt` 方法的任务是为 `self` 生成格式良好的表达形式并将其字符写入 `dest`。除了用作输出流，`dest` 参数还携带着从格式参数解析出的详细信息，比如对齐方式和最小字段宽度。

例如，本章前面曾建议，如果 Complex 值能以通常的 $a + bi$ 形式打印自己则会更好。下面是执行本操作的 Display 实现：

```
use std::fmt;

impl fmt::Display for Complex {
    fn fmt(&self, dest: &mut fmt::Formatter) -> fmt::Result {
        let im_sign = if self.im < 0.0 { '-' } else { '+' };
        write!(dest, "{} {} {}i", self.re, im_sign,
f64::abs(self.im))
    }
}
```

这利用了 Formatter 本身就是一个输出流的事实，所以 write! 宏可以帮我们完成大部分工作。有了这个实现，就可以写出如下代码了：

```
let one_twenty = Complex { re: -0.5, im: 0.866 };
assert_eq!(format!("{}", one_twenty),
            "-0.5 + 0.866i");

let two_forty = Complex { re: -0.5, im: -0.866 };
assert_eq!(format!("{}", two_forty),
            "-0.5 - 0.866i");
```

有时以极坐标形式显示复数会很有帮助：想象在复平面上画一条从原点到数值的线，极坐标形式会给出线的长度，以及线与正向 x 轴之间的顺时针夹角。格式参数中的 # 字符通常会选择某种替代的显示形式，Display 实现可以将其视为要求使用极坐标形式：

```
impl fmt::Display for Complex {
    fn fmt(&self, dest: &mut fmt::Formatter) -> fmt::Result {
        let (re, im) = (self.re, self.im);
        if dest.alternate() {
            let abs = f64::sqrt(re * re + im * im);
            let angle = f64::atan2(im, re) / std::f64::consts::PI
* 180.0;
            write!(dest, "{} ∠ {}°", abs, angle)
        } else {
            let im_sign = if im < 0.0 { '-' } else { '+' };
            write!(dest, "{} {} {}i", re, im_sign, f64::abs(im))
        }
    }
}
```

```
}  
}
```

使用此实现的代码如下所示：

```
let ninety = Complex { re: 0.0, im: 2.0 };  
assert_eq!(format!("{}", ninety),  
           "0 + 2i");  
assert_eq!(format!("{:?}", ninety),  
           "2 ∠ 90°");
```

尽管格式化特型的 `fmt` 方法会返回一个 `fmt::Result` 值（典型的模块专属的 `Result` 类型），但你只能从 `Formatter` 的操作中开始传播错误，就像刚才 `fmt::Display` 的实现中调用 `write!` 时的做法那样。你的格式化函数自身不应该引发错误。这样像 `format!` 这样的宏就可以简单地返回一个 `String` 而非 `Result<String, ...>`，因为将格式化后的文本追加到 `String` 上永远不会出错。这还会确保你从 `write!` 或 `writeln!` 上抛出的错误总能正确地反映出底层 I/O 流的实际问题，而不是某种格式问题。

`Formatter` 还有许多其他的有用的方法，包括一些用于处理结构化数据（如映射、列表等）的方法，本书并没有介绍它们，有关详细信息，请参阅在线文档。

17.4.9 在自己的代码中使用格式化语言

使用 Rust 的 `format_args!` 宏和 `std::fmt::Arguments` 类型，你可以编写能接受格式模板和参数的自定义函数和宏。假设你的程序需要在运行期记录状态消息，并且你想使用 Rust 的文本格式化语言来生成这些消息，那么可以参考以下代码：

```
fn logging_enabled() -> bool { ... }  
  
use std::fs::OpenOptions;  
use std::io::Write;  
  
fn write_log_entry(entry: std::fmt::Arguments) {
```

```
if logging_enabled() {  
    // 尽量保持简单，所以每次只是打开文件  
    let mut log_file = OpenOptions::new()  
        .append(true)  
        .create(true)  
        .open("log-file-name")  
        .expect("failed to open log file");  
  
    log_file.write_fmt(entry)  
        .expect("failed to write to log");  
}  
}
```

可以像这样调用 `write_log_entry`:

```
write_log_entry(format_args!("Hark! {:?}\n", mysterious_value));
```

在编译期，`format_args!` 宏会解析模板字符串并据此检查参数的类型，如果有任何问题则报告错误。在运行期，它会对参数求值并构建一个 `Arguments` 值，其中包含格式化文本时需要的所有信息：模板的预解析形式，以及对参数值的共享引用。

构造一个 `Arguments` 值的代价很低：只是收集一些指针而已。这时尚未进行任何格式化工作，仅收集稍后要用到的信息。这很重要，否则如果未启用日志，那么像把数值转换为十进制、填补值之类的任何开销都会白白浪费。

`File` 类型实现了 `std::io::Write` 特型，该特型的 `write_fmt` 方法会接受一个 `Argument` 并进行格式化，然后将结果写入底层流。

对 `write_log_entry` 的调用并不漂亮。这时宏就可以大显身手了：

```
macro_rules! log { // 在宏定义中的宏名后不需要叹号 (!)  
    ($format:tt, $($arg:expr),*) => (  
        write_log_entry(format_args!($format, $($arg),*))  
    )  
}
```

第 21 章会详细介绍宏。现在，你只需知道这定义了一个新 `log!` 宏并将其参数传给 `format_args!`，然后在生成的 `Arguments` 值上调用 `write_log_entry` 函数即可。诸如 `println!`，`writeln!` 和 `format!` 之类的格式化宏都采用了大致相同的思路。

可以像这样使用 `log!`:

```
log!("O day and night, but this is wondrous strange! {:?}\n",  
    mysterious_value);
```

理论上，这会好看一点儿。

17.5 正则表达式

外部的 `regex crate` 是 Rust 的官方正则表达式库，它提供了通常的搜索函数和匹配函数。该库对 Unicode 有很好的支持，但它也可以搜索字节串。尽管不支持其他正则表达式包中的某些特性（比如反向引用和环视模式），但这些简化允许 `regex` 确保搜索时间始终与表达式的大小、表达式的长度和待搜文本的长度呈线性关系。此外，这些保证还让 `regex` 即使在搜索不可信文本的不可信表达式时也能安全地使用。

本书将只提供 `regex` 的概述。有关详细信息，可以查阅其在线文档。

尽管 `regex crate` 不在 `std` 中，但它是由 Rust 库团队维护的，该团队也负责维护标准库 `std`。要使用 `regex`，请将下面这行代码放在 `crate` 的 `Cargo.toml` 文件的 `[dependencies]` 部分：

```
regex = "1"
```

在以下内容中，我们将假设你已做完了此项更改。

17.5.1 **Regex** 的基本用法

`Regex` 值表示已经解析好的正则表达式。`Regex::new` 构造函数会尝试将 `&str` 解析为正则表达式，并返回一个 `Result`：

```
use regex::Regex;

// 语义化版本号，比如0.2.1
// 可以包含预发行版本后缀，比如0.2.1-alpha
// （为简洁起见，没有“构建编号”元信息后缀）
//
// 注意，使用原始字符串语法r"... "是为了避免一大堆反斜杠
let semver = Regex::new(r"(\d+)\.(\d+)\.(\d+) (-[.[:alnum:]]*)?")?;
```



```
// 简单搜索，返回布尔型结果
let haystack = r#"regex = "0.2.5"#"#;
assert!(semver.is_match(haystack));
```

`Regex::captures` 方法会在字符串中搜索第一个匹配项并返回一个 `regex::Captures` 值，其中包含表达式中每个组的匹配信息：

```
// 可以检索各个捕获组：
let captures = semver.captures(haystack)
    .ok_or("semver regex should have matched")?;
assert_eq!(&captures[0], "0.2.5");
assert_eq!(&captures[1], "0");
assert_eq!(&captures[2], "2");
assert_eq!(&captures[3], "5");
```

如果所请求的组不匹配，则对 `Captures` 值进行索引就会出现 panic。要测试特定组是否匹配，可以调用 `Captures::get`，它会返回 `Option<regex::Match>`，其中的 `Match` 值会记录单个组的匹配信息：

```
assert_eq!(captures.get(4), None);
assert_eq!(captures.get(3).unwrap().start(), 13);
assert_eq!(captures.get(3).unwrap().end(), 14);
assert_eq!(captures.get(3).unwrap().as_str(), "5");
```

可以遍历字符串中的所有匹配项：

```
let haystack = "In the beginning, there was 1.0.0. \
    For a while, we used 1.0.1-beta, \
    but in the end, we settled on 1.2.4.";

let matches: Vec<&str> = semver.find_iter(haystack)
    .map(|match_| match_.as_str())
    .collect();
assert_eq!(matches, vec!["1.0.0", "1.0.1-beta", "1.2.4"]);
```

`find_iter` 迭代器会为表达式的每个非重叠匹配生成一个 `Match` 值，从字符串的开头走到结尾。`captures_iter` 方法也类似，但会生成记录了所有捕获组的 `captures` 值。当必须报告出捕获组时搜

索速度会变慢，因此如果并不实际需要捕获组，那么最好使用某个不返回它们的方法。

17.5.2 惰性构建正则表达式值

`Regex::new` 构造函数的开销可能很高：在速度较快的开发机器上为 1200 个字符的正则表达式构造一个 `Regex` 会花费差不多 1 毫秒时间，即使是一个微不足道的表达式也要花费几微秒时间。最好让 `Regex` 构造远离繁重的计算循环，这就意味着应该只构建一次 `Regex`，然后重复使用它。

`lazy_static` crate 提供了一种在首次使用时惰性构造静态值的好办法。首先，请注意 `Cargo.toml` 文件中的依赖项：

```
[dependencies]
lazy_static = "1"
```

这个 crate 提供了一个宏来声明这样的变量：

```
use lazy_static::lazy_static;

lazy_static! {
    static ref SEMVER: Regex
        = Regex::new(r"(\d+)\.(\d+)\.(\d+) (-[-.[:alnum:]]*)?")
            .expect("error parsing regex");
}
```

该宏会扩展成名为 `SEMVER` 的静态变量的声明，但其类型不完全是 `Regex`，而是一个实现了 `Deref<Target=Regex>` 的由宏生成的类型，并公开了与 `Regex` 相同的全部方法。第一次解引用 `SEMVER` 时，会执行初始化程序，并保存该值供以后使用。由于 `SEMVER` 是一个静态变量，而不仅仅是局部变量，因此每次执行程序时初始化器都最多运行一次。

有了这个声明，使用 `SEMVER` 就很简单了：

```
use std::io::BufRead;

let stdin = std::io::stdin();
for line_result in stdin.lock().lines() {
    let line = line_result?;
    if let Some(match_) = SEMVER.find(&line) {
        println!("{}", match_.as_str());
    }
}
```

可以把 `lazy_static!` 声明放在模块中，甚至可以放在使用 `Regex` 的函数内部（如果这就是最合适的作用域的话）。无论采用哪种方式，每当程序执行时，正则表达式都只会编译一次。

17.6 规范化

大多数用户误以为法语单词 `thé`（意为“茶”）的长度是 3 个字符。然而，Unicode 实际上有两种方式来表示这个单词。

- 在组合形式中，“`thé`”包含 3 个字符，即 `'t'`、`'h'` 和 `'é'`，其中 `'é'` 是码点为 `0xe9` 的单个 Unicode 字符。
- 在分解形式中，“`thé`”包含 4 个字符，即 `'t'`、`'h'`、`'e'` 和 `'\u{301}'`，其中的 `'e'` 是纯 ASCII 字符，没有重音符号，而码点 `0x301` 是“结合性锐音符号”字符，它会为它前面的任意字符添加一个锐音符号。

Unicode 并不认为 `é` 的组合形式或分解形式是“正确的”形式，相反，它认为它们是同一抽象字符的等价表示。Unicode 规定这两种形式应该以相同的方式显示，并且允许文本输入法生成任何一种形式，因此用户通常不知道他们正在查看或输入的是哪种形式。（Rust 允许直接在字符串字面量中使用 Unicode 字符，因此如果不关心自己获得的是哪种编码，则可以简单地写成 `"thé"`。但为了清楚起见，这里我们会使用 `\u` 转义符。）

然而，作为 Rust 的 `&str` 值或 `String` 值，`"th\u{e9}"` 和 `"the\u{301}"` 是完全不同的。它们具有不同的长度，比较起来不相等，具有不同的哈希值，并且相对于其他字符串会以不同的方式排序：

```
assert!("th\u{e9}" != "the\u{301}");
assert!("th\u{e9}" > "the\u{301}");

// 哈希器旨在累积求出一系列值的哈希值，因此仅哈希一个值有点儿大材小用
use std::hash::{Hash, Hasher};
use std::collections::hash_map::DefaultHasher;
fn hash<T: ?Sized + Hash>(t: &T) -> u64 {
    let mut s = DefaultHasher::new();
    t.hash(&mut s);
    s.finish()
}
```

```
// 这些值可能会在将来的Rust版本中发生变化
assert_eq!(hash("th\u{e9}"), 0x53e2d0734eb1dff3);
assert_eq!(hash("the\u{301}"), 0x90d837f0a0928144);
```

显然，如果打算比较用户提供的文本或者将其用作哈希表或 B 树中的键，则需要先将每个字符串转换成某种规范形式。

幸运的是，Unicode 指定了字符串的规范化形式。每当根据 Unicode 规则应将两个字符串视为等同时，它们的规范化形式是逐字符全同的。当使用 UTF-8 编码时，它们是逐字节全同的。这意味着可以使用 `==` 来比较规范化后的字符串，可以将它们用作 `HashMap` 或 `HashSet` 中的键，等等，这样就能获得 Unicode 规定的相等性概念了。

如果未做规范化，则甚至会产生安全隐患。如果你的网站对用户名在某些情况下做了规范化，但在其他情况下未做规范化，那么最终可能会出现两个名为 `bananasflambé` 的不同用户，你的一部分代码会将其视为同一用户，但另一部分代码会认为这是两个用户，导致一个人的权限被错误地扩展到另一个人身上。当然，有很多方法可以避开这种问题，但历史表明也有很多方法不能避开。

17.6.1 规范化形式

Unicode 定义了 4 种规范化形式，每一种都适用于不同的用途。这里要回答两个问题。

- 第一个问题是：你更喜欢让字符尽可能组合还是尽可能分解？
例如，越南语单词 `Phở` 最常用的组合表示是三字符字符串 `"Ph\u{1edf}"`，其中声调标记 `’` 和元音标记 `’` 都应用于基本字符 `“o”` 上，而其单个 Unicode 字符是 `'\u{1edf}'`，Unicode 很质朴地将其命名为“带角和钩形的拉丁文小写字母 `o`”。

最常用的分解表示是将基本字母及其两个标记拆分为 3 个单独的 Unicode 字符：`'o'`、`'\u{31b}'`（组合角符）和 `'\u{309}'`（组合上钩符），其结果就是

"Pho\u{31b}\u{309}"。（每当组合标记作为单独的字符出现，而不是作为组合字符的一部分时，所有规范化形式都指定了它们必须以固定顺序出现，因此即使字符有多个重音符号，也能很好地进行规范化。）

组合形式通常具有较少的兼容性问题，因为它更接近于在 Unicode 建立之前用于其文本的大多数语言的表示。它也可以更好地与简单的字符串格式化特性（如 Rust 的 `format!` 宏）协作。而分解形式可能更适合显示文本或搜索，因为它使文本的详细结构更加明确。

- 第二个问题是：如果两个字符序列表示相同的基础文本，但文本的格式化方式不同，那么你是要将它们视为等同的还是坚持认为有差异？

Unicode 对普通数字 5、上标数字 ⁵（或 '

兼容性等效对搜索很有意义：搜索仅使用了 ASCII 字符的 "difficult"，应该匹配使用了 ffi 连字符的字符串 "di\u{fb03}cult"。对后一个字符串应用兼容性分解会将连字替换为 3 个纯字母 "ffi"，从而让搜索更容易。但是将文本规范化为其兼容的等效形式可能会丢失重要信息，因此不应草率应用。例如，在大多数情况下将 "2⁵" 存储为 "25" 是不正确的。

Unicode 规范化形式 C (NFC) 和规范化形式 D (NFD) 会使用每个字符的最大组合形式和最大分解形式，但不会试图统一兼容性等价序列。NFKC 规范化形式和 NFKD 规范化形式类似于 NFC 和 NFD，但它们会将所有兼容性等效序列规范化为各自的一些简单表示法。

万维网联盟的“WWW 字符模型”建议对所有内容都使用 NFC。Unicode 标识符和模式语法附件则建议使用 NFKC 作为编程语言中的标识符，并提供了在必要时适配此形式的原则。

17.6.2 unicode-normalization crate

Rust 的 `unicode-normalization` crate 提供了一个特型，可以将方法添加到 `&str` 中，以便将文本转成四种规范化形式中的任何一种。要使用这个 crate，请将下面这行代码添加到 `Cargo.toml` 文件的 `[dependencies]` 部分：

```
unicode-normalization = "0.1.17"
```

有了这个声明，`&str` 就有了 4 个新方法，它们会返回字符串的特定规范化形式的迭代器：

```
use unicode_normalization::UnicodeNormalization;

// 不管左边的字符串使用哪种表示形式（无法仅仅通过观察得知），这些断言都成立
assert_eq!("Phở".nfd().collect::(), "Pho\u{31b}\u{309}");
assert_eq!("Phở".nfc().collect::(), "Ph\u{1edf}");

// 左侧使用了"ffi"连字符
assert_eq!("@ Di\u{fb03}culty".nfkc().collect::(), "1
Difficulty");
```

接受规范化的字符串并以相同的形式再次对其进行规范化可以保证返回相同的文本。

尽管规范化字符串的任何子字符串本身也是规范化的，但两个规范化字符串拼接起来不一定是规范化的。例如，第二个字符串可能以组合字符开头，并且这个字符按规范应该排在第一个字符串末尾的组合字符之前。

只要文本在规范化时没有使用未分配的码点，Unicode 就承诺其规范化形式在标准的未来版本中不会改变。这意味着规范化形式通常可以安全地用于持久存储，即使 Unicode 标准在不断发展也不会受影响。

第 18 章 输入与输出

Doolittle: 你有什么具体证据能证明你的存在?

炸弹-20: 嗯……好吧……我思故我在。

Doolittle: 很好。非常好。但你又如何知道其他事物的存在呢?

炸弹-20: 我的感官感受到了。

——科幻喜剧《暗星》 (*Dark Star*)

Rust 标准库中的输入和输出的特性是围绕 3 个特型组织的, 即 `Read`、`BufRead` 和 `Write`。

- 实现了 `Read` 的值具有面向字节的输入方法。它们叫作**读取器**。
- 实现了 `BufRead` 的值是**缓冲读取器**。它们支持 `Read` 的所有方法, 外加读取文本行等方法。
- 实现了 `Write` 的值能支持面向字节和 UTF-8 文本的输出。它们叫作**写入器**。

图 18-1 展示了这 3 个特型以及几个读取器类型和写入器类型的示例。

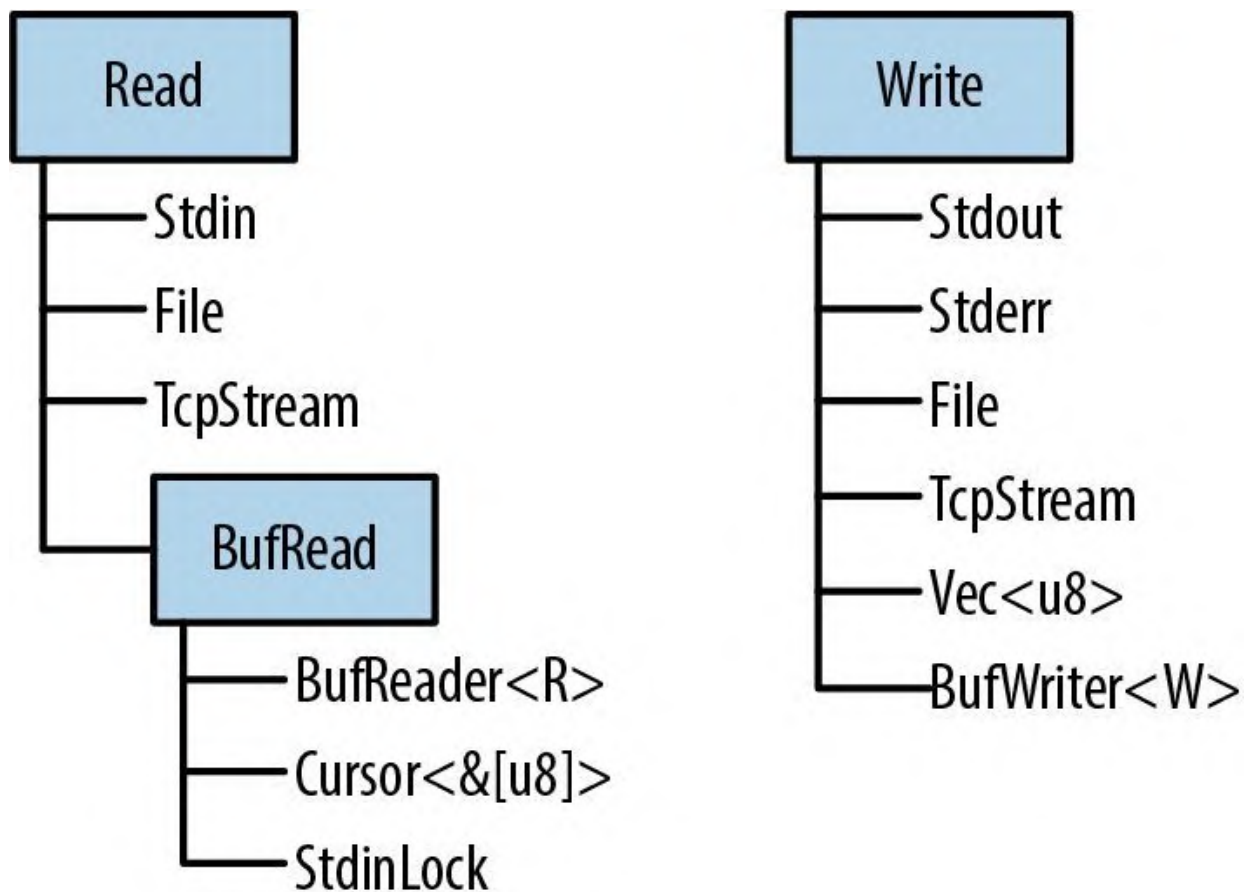


图 18-1: Rust 的 3 个主要 I/O 特型和一些实现了它们的类型

在本章中，我们会讲解如何使用这些特型及其方法，涵盖了图 18-1 中所示的这些读取器类型和写入器类型，并展示了与文件、终端和网络进行交互的其他方式。

18.1 读取器与写入器

你的程序可以从读取器中读取一些字节。例如：

- 使用 `std::fs::File::open(filename)` 打开的文件；
- `std::net::TcpStream`，用于通过网络接收数据；
- `std::io::stdin()`，用于从进程的标准输入流中进行读取；
- `std::io::Cursor<&[u8]>` 值和 `std::io::Cursor<Vec<u8>>` 值，它们是从已存在于内存中的字节数组或向量中进行“读取”的读取器。

写入器则可以向其中写入一些字节。例如：

- 使用 `std::fs::File::create(filename)` 打开的文件；
- `std::net::TcpStream`，用于通过网络发送数据；
- `std::io::stdout()` 和 `std::io::stderr()`，用于写入到终端；
- `Vec<u8>` 也是一个写入器，它的 `write` 方法会把内容追加到向量；
- `std::io::Cursor<Vec<u8>>`，它与 `Vec<u8>` 很像，但允许读取数据和写入数据，并能在向量中寻找不同的位置；
- `std::io::Cursor<&mut [u8]>`，它与 `std::io::Cursor<Vec<u8>>` 很像，不过不能增长缓冲区，因为它只是一些现有字节数组的切片。

由于读取器和写入器都有标准特型（`std::io::Read` 和 `std::io::Write`），因此编写适用于各种输入通道或输出通道的泛型代码是很常见的。例如，下面是一个将所有字节从任意读取器复制到任意写入器的函数：

```
use std::io::{self, Read, Write, ErrorKind};

const DEFAULT_BUF_SIZE: usize = 8 * 1024;

pub fn copy<R: ?Sized, W: ?Sized>(reader: &mut R, writer: &mut W)
    -> io::Result<u64>
```

```

    where R: Read, W: Write
{
    let mut buf = [0; DEFAULT_BUF_SIZE];
    let mut written = 0;
    loop {
        let len = match reader.read(&mut buf) {
            Ok(0) => return Ok(written),
            Ok(len) => len,
            Err(ref e) if e.kind() == ErrorKind::Interrupted =>
continue,
            Err(e) => return Err(e),
        };
        writer.write_all(&buf[..len])?;
        written += len as u64;
    }
}

```

这是 Rust 标准库中 `std::io::copy()` 的实现。由于是泛型的，因此可以使用它将数据从 `File` 复制到 `TcpStream`，从标准输入复制到内存中的 `Vec<u8>`，等等。

如果对此处的错误处理代码不清楚，可以重温一下第 7 章。我们接下来还要不断使用 `Result` 类型，了解它的工作原理很重要。

`std::io` 的 `Read`、`BufRead` 和 `Write` 这 3 个特型以及 `Seek` 非常常用，下面是一个只包含这些特型的 `prelude` 模块：

```
use std::io::prelude::*;
```

你会在本章中看到一两次这种写法。我们还习惯于导入 `std::io` 模块本身：

```
use std::io::{self, Read, Write, ErrorKind};
```

此处的 `self` 关键字将 `io` 声明成了 `std::io` 模块的别名。这样，`std::io::Result` 和 `std::io::Error` 就可以更简洁地写为 `io::Result` 和 `io::Error` 了。

18.1.1 读取器

`std::io::Read` 有以下几个读取数据的方法。所有这些方法都需要对读取器本身进行可变引用。

```
reader.read(&mut buffer) (读取)
```

从数据源中读取一些字节并将它们存储在给定的 `buffer` 中。`buffer` 参数的类型是 `&mut[u8]`。此方法最多会读取 `buffer.len()` 字节。

返回类型是 `io::Result<u64>`，它是 `Result<u64, io::Error>` 的类型别名。成功时，这个 `u64` 的值是已读取的字节数——可能等于或小于 `buffer.len()`，就算数据源突然涌入更多数据也不会超出。`Ok(0)` 则意味着没有更多输入可以读取了。

出错时，`.read()` 会返回 `Err(err)`，其中 `err` 是一个 `io::Error` 值。为了对人类友好，`io::Error` 是可打印的；为了便于程序处理，它有一个 `.kind()` 方法，该方法会返回 `io::ErrorKind` 类型的错误代码。此枚举的成员都有 `PermissionDenied` 和 `ConnectionReset` 之类的名称。大多数表示严重的错误，不容忽视，但有一种错误需要特殊处理。`io::ErrorKind::Interrupted` 对应于 Unix 错误码 `EINTR`，表示读取恰好被某种信号中断了。除非程序的设计目标之一就是对信号中断做一些巧妙的处理，否则就应该再次尝试读取。18.1 节中的 `copy()` 代码就展示了这种示例。

如你所见，`.read()` 方法是非常底层的，甚至还继承了底层操作系统的某些怪癖。如果你正在为一种新型数据源实现 `Read` 特型，那么这会给你很大的发挥空间。但如果你想读取一些数据，则会很痛苦。因此，Rust 提供了几个更高级的便捷方法。这些方法是 `Read` 中的默认实现，而且都处理了 `ErrorKind::Interrupted`，这样你就不必自己处理了。

```
reader.read_to_end(&mut byte_vec) (读至末尾)
```

从这个读取器读取所有剩余的输入，将其追加到 `Vec<u8>` 型的 `byte_vec` 中。返回 `io::Result<usize>`，即已读取的字节数。

此方法对要装入向量中的数据量没有任何限制，因此不要在不受信任的来源上使用它。（可以用接下来要讲的 `.take()` 方法加以限制。）

```
reader.read_to_string(&mut string) (读取字符串)
```

和上一个方法类似，但此方法会将数据附加到给定的 `String`。如果流不是有效的 UTF-8，则返回 `ErrorKind::InvalidData` 错误。

在某些编程语言中，字节输入和字符输入是由不同的类型处理的。如今，UTF-8 占据了绝对主导地位，所以 Rust 承认这一事实标准并且处处支持 UTF-8。开源的 `encoding crate` 可以支持其他字符集。

```
reader.read_exact(&mut buf) (精确读满)
```

读取足够的数据来填充给定的缓冲区。参数类型是 `&[u8]`。如果读取器在读够 `buf.len()` 字节之前就耗尽了数据，那么此方法就会返回 `ErrorKind::UnexpectedEof` 错误。

以上这些是 `Read` 特型的主要方法。此外，还有 3 个适配器方法可以按值获取 `reader`，将其转换为迭代器或另一种读取器。

```
reader.bytes() (字节迭代器)
```

返回输入流中各字节的迭代器。条目类型是 `io::Result<u8>`，因此每字节都需要进行错误检查。此外，此方法会为每字节调用一次 `reader.read()`，如果没有缓冲，则会非常低效。

```
reader.chain(reader2) (串联)
```

返回新的读取器，先生成来自 `reader` 的所有输入，然后再生成来自 `reader2` 的所有输入。

`reader.take(n)`（取出 n 个）

返回新的读取器，从与 `reader` 相同的数据源读取，但仅限于前 n 字节的输入。

没有关闭读取器的方法。读取器和写入器通常会实现 `Drop` 以便自行关闭。

18.1.2 缓冲读取器

为了提高效率，可以对读取器和写入器进行缓冲，这基本上意味着它们有一块内存（缓冲区），用于保存一些输入数据或输出数据。这可以减少一些系统调用，如图 18-2 所示。应用程序会从 `BufReader` 中读取数据，在本例中是通过调用其 `.read_line()` 方法实现的。`BufReader` 会依次从操作系统获取更大块的输入。

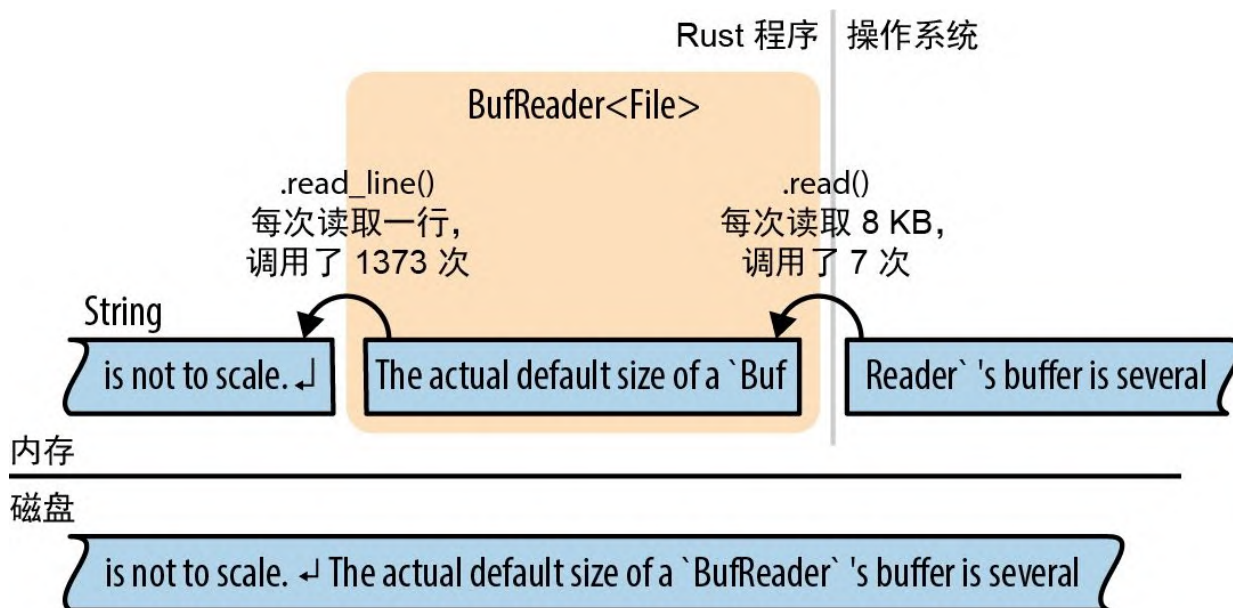


图 18-2：缓冲文件读取器

图 18-2 未按比例绘制。BufReader 缓冲区的实际大小默认为几千字节，因此一次 read 系统调用就可以服务数百次 .read_line() 调用。这很重要，因为系统调用很慢。

（如图 18-2 所示，操作系统也有自己的缓冲区。同理：系统调用固然慢，但从磁盘读取数据更慢。）

缓冲读取器同时实现了 Read 特型和 BufRead 特型，后者添加了以下方法。

```
reader.read_line(&mut line) (读一行)
```

读取一行文本并将其追加到 line，line 是一个 String。行尾的换行符 '\n' 包含在 line 中。

如果输入带有 Windows 风格的行尾结束符号 "\r\n"，则这两个字符都会包含在 line 中。

返回值是 io::Result<usize>，表示已读取的字节数，包括行尾结束符号（如果有的话）。

如果读取器在输入的末尾，则会保持 line 不变并返回 Ok(0)。

```
reader.lines() (文本行迭代器)
```

返回生成各个输入行的迭代器。条目类型是 io::Result<String>。字符串中不包含换行符。如果输入带有 Windows 风格的行尾结束符号 "\r\n"，则这两个字符都会被去掉。

在绝大多数场景中，这个方法足够你进行文本输入了。接下来的 18.1.3 节和 18.1.4 节展示了它的一些使用示例。

```
reader.read_until(stop_byte, &mut byte_vec) (读到 stop_byte 为止) 和 reader.split(stop_byte) (根据 stop_byte 拆分)
```


与 `.read_line()` 和 `.lines()` 类似，但这两个方法是面向字节的，会生成 `Vec<u8>` 而不是 `String`。你要自选分隔符 `stop_byte`。

`BufRead` 还提供了 `.fill_buf()` 和 `.consume(n)`，这是一对底层方法，用于直接访问读取器的内部缓冲区。有关这两个方法的更多信息，请参阅在线文档。

接下来的 18.1.3 节和 18.1.4 节将更详细地介绍缓冲读取器。

18.1.3 读取行

下面是一个用于实现 Unix `grep` 实用程序的函数，该函数会在多行文本（通常是通过管道从另一条命令输入的文本）中搜索给定字符串：

```
use std::io;
use std::io::prelude::*;

fn grep(target: &str) -> io::Result<()> {
    let stdin = io::stdin();
    for line_result in stdin.lock().lines() {
        let line = line_result?;
        if line.contains(target) {
            println!("{}", line);
        }
    }
    Ok(())
}
```

因为要调用 `.lines()`，所以需要实现了一个实现了 `BufRead` 的输入源。在这种情况下，可以调用 `io::stdin()` 来获取通过管道传输给我们的数据。但是，Rust 标准库使用互斥锁保护着 `stdin`。因此要调用 `.lock()` 来锁定 `stdin` 以供当前线程独占使用，这会返回一个实现了 `BufRead` 的 `StdinLock` 值。在循环结束时，`StdinLock` 会被丢弃，释放互斥锁。（如果没有互斥锁，那么当两个线程试图同时从 `stdin` 读取时就会导致未定义行为。C 语言也有相同的问题，且以相

同的方式来解决：所有 C 标准输入函数和输出函数都会在幕后获得锁。唯一的不同在于，在 Rust 中，锁是 API 的一部分。）

该函数的其余部分都很简单：它会调用 `.lines()` 并对生成的迭代器进行循环。因为这个迭代器会生成 `Result` 值，所以要用 `?` 运算符检查错误。

假如我们想完善这个 `grep` 程序，让它支持在磁盘上搜索文件。可以把这个函数变成泛型函数：

```
fn grep<R>(target: &str, reader: R) -> io::Result<()>
    where R: BufRead
{
    for line_result in reader.lines() {
        let line = line_result?;
        if line.contains(target) {
            println!("{}", line);
        }
    }
    Ok(())
}
```

现在可以将 `StdinLock` 或带缓冲的 `File` 传给它：

```
let stdin = io::stdin();
grep(&target, stdin.lock())?; // 正确

let f = File::open(file)?;
grep(&target, BufReader::new(f))?; // 同样正确
```

请注意，`File` 不会自动缓冲。`File` 实现了 `Read` 但没实现 `BufRead`。但是，为 `File` 或任意无缓冲读取器创建缓冲读取器很容易。`BufReader::new(reader)` 就是做这个的。（要设置缓冲区的大小，请使用 `BufReader::with_capacity(size, reader)`。）

在大多数语言中，文件默认是带缓冲的。如果想要无缓冲的输入或输出，就必须弄清楚如何关闭缓冲。在 Rust 中，`File` 和 `BufReader`

是两个独立的库特性，因为有时你想要不带缓冲的文件，有时你想要不带文件的缓冲（例如，你可能想要缓冲来自网络的输入）。

下面是一个包括错误处理和一些粗略的参数解析的完整程序。

```
// grep—搜索stdin或其他文件，以便使用给定的字符串进行逐行匹配

use std::error::Error;
use std::io::{self, BufReader};
use std::io::prelude::*;
use std::fs::File;
use std::path::PathBuf;

fn grep<R>(target: &str, reader: R) -> io::Result<()>
    where R: BufRead
{
    for line_result in reader.lines() {
        let line = line_result?;
        if line.contains(target) {
            println!("{}", line);
        }
    }
    Ok(())
}

fn grep_main() -> Result<(), Box<dyn Error>> {
    // 获取命令行参数。第一个参数是要搜索的字符串，其他参数是一些文件名
    let mut args = std::env::args().skip(1);
    let target = match args.next() {
        Some(s) => s,
        None => Err("usage: grep PATTERN FILE...")?
    };
    let files: Vec<PathBuf> = args.map(PathBuf::from).collect();

    if files.is_empty() {
        let stdin = io::stdin();
        grep(&target, stdin.lock())?;
    } else {
        for file in files {
            let f = File::open(file)?;
            grep(&target, BufReader::new(f))?;
        }
    }

    Ok(())
}
```

```
fn main() {
    let result = grep_main();
    if let Err(err) = result {
        eprintln!("{}", err);
        std::process::exit(1);
    }
}
```

18.1.4 收集行

有些读取器方法（包括 `.lines()`）会返回生成 `Result` 值的迭代器。当你第一次想要将文件的所有行都收集到一个大型向量中时，就会遇到如何摆脱 `Result` 的问题：

```
// 正确，但不是你想要的
let results: Vec<io::Result<String>> = reader.lines().collect();

// 错误：不能把Result的集合转换成Vec<String>
let lines: Vec<String> = reader.lines().collect();
```

第二次尝试无法编译：遇到这些错误怎么办？最直观的解决方法是编写一个 `for` 循环并检查每个条目是否有错：

```
let mut lines = vec![];
for line_result in reader.lines() {
    lines.push(line_result?);
}
```

这固然没错，但这里最好还是用 `.collect()`，事实上确实可以做到。只要知道该请求哪种类型就可以了：

```
let lines = reader.lines().collect::<io::Result<Vec<String>>>()?;
```

这是怎么做到的呢？标准库中包含了 `Result` 对 `FromIterator` 的实现（在线文档中这很容易被忽略），这个实现让一切成为可能：

```
impl<T, E, C> FromIterator<Result<T, E>> for Result<C, E>
    where C: FromIterator<T>
{
```

```
...  
}
```

这需要仔细阅读，但确实是一个很好的技巧。假设 `C` 是任意集合类型，比如 `Vec` 或 `HashSet`。只要已经知道如何从 `T` 值的迭代器构建出 `C`，就可以从生成 `Result<T, E>` 值的迭代器构建出 `Result<C, E>`。只需从迭代器中提取各个值并从 `Ok` 结果构建出集合即可，但一旦看到 `Err`，就停止并将其传出。

换句话说，`io::Result<Vec<String>>` 是一种集合类型，因此 `.collect()` 方法可以创建并填充该类型的值。

18.1.5 写入器

如前所述，输入主要是用方法完成的，而输出略有不同。

本书一直在使用 `println!()` 生成纯文本输出：

```
println!("Hello, world!");  
  
println!("The greatest common divisor of {:?} is {}",  
         numbers, d);  
  
println!(); // 打印空行
```

还有不会在行尾添加换行符的 `print!()` 宏，以及会写入标准错误流的 `eprintln!` 宏和 `eprint!` 宏。所有这些宏的格式化代码都和 `format!` 宏一样，17.4 节曾讲解过。

要将输出发送到写入器，请使用 `write!()` 宏和 `writeln!()` 宏。它们和 `print!()` 和 `println!()` 类似，但有两点区别：

```
writeln!(io::stderr(), "error: world not helloable");  
  
writeln!(&mut byte_vec, "The greatest common divisor of {:?} is  
{}",  
         numbers, d)?;
```

一是每个 `write` 宏都接受一个额外的写入器作为第一参数。二是它们会返回 `Result`，因此必须处理错误。这就是为什么要在每行末尾使用 `?` 运算符。

`print` 宏不会返回 `Result`，如果写入失败，它们只会 `panic`。由于写入的是终端，所以极少失败。

`Write` 特型有以下几个方法。

`writer.write(&buf)`（写入）

将切片 `buf` 中的一些字节写入底层流。此方法会返回 `io::Result<usize>`。成功时，这给出了已写入的字节数，如果流突然提前关闭，那么这个值可能会小于 `buf.len()`。

和 `Reader::read()` 一样，这是一个要避免直接使用的底层方法。

`writer.write_all(&buf)`（写入全部）

将切片 `buf` 中的所有字节都写入。返回 `Result<()>`。

`writer.flush()`（刷新缓冲区）

将可能被缓冲在内存中的数据刷新到底层流中。返回 `Result<()>`。

请注意，虽然 `println!` 宏和 `eprintln!` 宏会自动刷新 `stdout` 流和 `stderr` 流的缓冲区，但 `print!` 宏和 `eprint!` 宏不会。使用它们时，可能要手动调用 `flush()`。

与读取器一样，写入器也会在被丢弃时自动关闭。

正如 `BufReader::new(reader)` 会为任意读取器添加缓冲区一样，`BufWriter::new(writer)` 也会为任意写入器添加缓冲区：

```
let file = File::create("tmp.txt"?;
let writer = BufWriter::new(file);
```

要设置缓冲区的大小，请使用

`BufWriter::with_capacity(size, writer)`。

当丢弃 `BufWriter` 时，所有剩余的缓冲数据都将写入底层写入器。但是，如果在此写入过程中发生错误，则错误会被忽略。（由于错误发生在 `BufWriter` 的 `.drop()` 方法内部，因此没有合适的地方来报告。）为了确保应用程序会注意到所有输出错误，请在丢弃带缓冲的写入器之前将它手动 `.flush()` 一下。

18.1.6 文件

下面是本书已经介绍过的打开文件的两个方法。

`File::open(filename)`（打开）

打开现有文件进行读取。此方法会返回一个 `io::Result<File>`，如果该文件不存在则报错。

`File::create(filename)`（创建）

创建一个用于写入的新文件。如果存在具有给定文件名的文件，则会将其截断。

请注意，`File` 类型位于文件系统模块 `std::fs` 中，而不是 `std::io` 中。

当这两个方法都不符合需求时，可以使用 `OpenOptions` 来指定所期望的确切行为：

```
use std::fs::OpenOptions;

let log = OpenOptions::new()
    .append(true) // 如果文件已存在，则追加到末尾
    .open("server.log"?;
```

```
let file = OpenOptions::new()
    .write(true)
    .create_new(true) // 如果文件已存在，则失败
    .open("new_file.txt")?;
```

方法 `.append()`、`.write()`、`.create_new()` 等是可以链式调用的：每个方法都会返回 `self`。这种链式调用的设计模式很常见，所以在 Rust 中它有一个专门的名字——**构建器**（builder）。另一个例子是 `std::process::Command`。有关 `OpenOptions` 的详细信息，请参阅在线文档。

`File` 打开后的行为就和任何读取器或写入器一样。如果需要，可以为它添加缓冲区。`File` 在被丢弃时会自动关闭。

18.1.7 寻址

`File` 还实现了 `Seek` 特型，这意味着你可以在 `File` 中“跳来跳去”，而不是从头到尾一次性读取或写入。`Seek` 的定义如下：

```
pub trait Seek {
    fn seek(&mut self, pos: SeekFrom) -> io::Result<u64>;
}

pub enum SeekFrom {
    Start(u64),
    End(i64),
    Current(i64)
}
```

此枚举让 `seek` 方法表现得很好：可以用 `file.seek(SeekFrom::Start(0))` 倒回到开头，还能用 `file.seek(SeekFrom::Current(-8))` 回退几字节，等等。

在文件中寻址很慢。无论使用的是硬盘还是固态驱动器（SSD），每一次寻址的开销都接近于读取数兆字节的数据。

18.1.8 其他读取器与写入器类型

迄今为止，本章一直在使用 `File` 作为示范的主力，但还有许多其他有用的读取器类型和写入器类型。

`io::stdin()`（标准输入）

返回标准输入流的读取器。类型是 `io::Stdin`。由于它被所有线程共享，因此每次读取都会获取和释放互斥锁。

`Stdin` 有一个 `.lock()` 方法，该方法会获取互斥锁并返回 `io::StdinLock`，这是一个带缓冲的读取器，在被丢弃之前会持有互斥锁。因此，对 `StdinLock` 的单个操作就避免了互斥开销。18.1.3 节展示过使用此方法的示例代码。

由于技术原因，不能直接调用 `io::stdin().lock()`¹。这个锁持有对 `Stdin` 值的引用，这意味着此 `Stdin` 值必须存储在某个能让它“活得”足够长的地方：

¹从 Rust 1.8 开始，已经可以直接调用 `io::stdin().lock()` 了。——译者注

```
let stdin = io::stdin();
let lines = stdin.lock().lines(); // 正确
```

`io::stdout()`（标准输出）和 `io::stderr()`（标准错误）

返回标准输出流（`Stdout`）类型和标准错误流（`Stderr`）类型的写入器。它们也有互斥锁和 `.lock()` 方法。

`Vec<u8>`（`u8` 向量）

实现了 `Write`。写入 `Vec<u8>` 会使用新数据扩展向量。

（但是，`String` 没有实现 `Write`。要使用 `Write` 构建字符串，需要首先写入 `Vec<u8>`，然后使用 `String::from_utf8(vec)` 将向量转换为字符串。）

`Cursor::new(buf)`（新建）

创建一个 `Cursor`（游标，一个从 `buf` 读取数据的缓冲读取器）。这样你就创建了一个能读取 `String` 的读取器。参数 `buf` 可以是实现了 `AsRef<[u8]>` 的任意类型，因此也可以传递 `&[u8]`、`&str` 或 `Vec<u8>`。

`Cursor` 的内部平平无奇，只有两个字段：`buf` 本身和一个整数，该整数是 `buf` 中下一次读取开始的偏移量。此位置的初始值为 0。

`Cursor` 实现了 `Read`、`BufRead` 和 `Seek`。如果 `buf` 的类型是 `&mut [u8]` 或 `Vec<u8>`，那么 `Cursor` 也实现了 `Write`。写入游标会覆盖 `buf` 中从当前位置开始的字节。如果试图直接写到超出 `&mut [u8]` 末尾的位置，就会导致一次“部分写入”或一个 `io::Error`。不过，使用游标写入 `Vec<u8>` 的结尾就没有这个问题：它会增长此向量。因此，`Cursor<&mut [u8]>` 和 `Cursor<Vec<u8>>` 实现了所有这 4 个 `std::io::prelude` 特型。

`std::net::TcpStream`（TCP 流）

表示 TCP 网络连接。由于 TCP 支持双向通信，因此它既是读取器又是写入器。

类型关联函数 `TcpStream::connect(("hostname", PORT))` 会尝试连接到服务器并返回 `io::Result<TcpStream>`。

`std::process::Command`（命令）

支持启动子进程并通过管道将数据传输到其标准输入，如下所示：

```
use std::process::{Command, Stdio};

let mut child =
    Command::new("grep")
        .arg("-e")
        .arg("a.*e.*i.*o.*u")
```

```
        .stdin(Stdio::piped())
        .spawn()?;

let mut to_child = child.stdin.take().unwrap();
for word in my_words {
    writeln!(to_child, "{}", word)?;
}
drop(to_child); // 关闭grep的stdin, 以便让它退出
child.wait()?;
```

`child.stdin` 的类型是 `Option<std::process::ChildStdin>`, 这里在建立子进程时使用了 `.stdin(Stdio::piped())`, 因此当 `.spawn()` 成功时, `child.stdin` 必然已经就位。如果没提供, 那么 `child.stdin` 就是 `None`。

`Command` 还有两个类似的方法 `.stdout()` 和 `.stderr()`, 可用于请求 `child.stdout` 和 `child.stderr` 中的读取器。

`std::io` 模块还提供了一些返回普通读取器和写入器的函数。

`io::sink()` (地漏)

这是无操作写入器。所有的写入方法都会返回 `Ok`, 但只是把数据扔掉了。

`io::empty()` (空白)

这是无操作读取器。读取总会成功, 但只会返回“输入结束”(EOF)。

`io::repeat(byte)` (重复)

返回一个会无限重复给定字节的读取器。

18.1.9 二进制数据、压缩和序列化

许多开源 `crate` 建立在 `std::io` 框架之上, 以提供额外的特性。

byteorder crate 提供了 ReadBytesExt 特型和 WriteBytesExt 特型，为所有读取器和写入器添加了二进制输入和输出的方法：

```
use byteorder::{ReadBytesExt, WriteBytesExt, LittleEndian};

let n = reader.read_u32::()?;
writer.write_i64::(n as i64)?;
```

flate2 crate 提供了用于读取和写入 gzip 数据的适配器方法：

```
use flate2::read::GzDecoder;
let file = File::open("access.log.gz")?;
let mut gzip_reader = GzDecoder::new(file);
```

serde crate 及其关联的格式类 crate（如 serde_json）实现了序列化和反序列化：它们在 Rust 结构体和字节之间来回转换。

11.2.2 节曾简单提到过这个 crate。现在可以仔细看看了。

假设我们有一些数据（文字冒险游戏中的映射表）存储在 HashMap 中：

```
type RoomId = String;                                // 每个房间都有唯一的名字
type RoomExits = Vec<(char, RoomId)>;                // .....并且存在一个出口列表
type RoomMap = HashMap<RoomId, RoomExits>;           // 房间名和出口的简单映射表

// 创建一个简单映射表
let mut map = RoomMap::new();
map.insert("Cobble Crawl".to_string(),
           vec![('W', "Debris Room".to_string())]);
map.insert("Debris Room".to_string(),
           vec![('E', "Cobble Crawl".to_string()),
                ('W', "Sloping Canyon".to_string())]);
...
```

将此数据转换为 JSON 输出只需一行代码：

```
serde_json::to_writer(&mut std::io::stdout(), &map)?;
```

在内部，`serde_json::to_writer` 使用了 `serde::Serialize` 特型的 `serialize` 方法。该库会将 `serde::Serialize` 特型附加到所有它知道如何序列化的类型中，包括我们的数据中出现过的类型：字符串、字符、元组、向量和 `HashMap`。

`serde` 很灵活。在这个程序中，输出是 JSON 数据，因为我们选择了 `serde_json` 序列化器。其他格式（如 `MessagePack`）也有对应的序列化器支持。同样，可以将此输出发送到文件、`Vec<u8>` 或任意写入器。前面的代码会将数据打印到 `stdout`。具体内容如下所示：

```
{ "Debris Room": [ [ "E", "Cobble Crawl" ], [ "W", "Sloping Canyon" ] ], "Cobble Crawl": [ [ "W", "Debris Room" ] ] }
```

`serde` 还包括对供派生的两个关键 `serde` 特型的支持：

```
#[derive(Serialize, Deserialize)]
struct Player {
    location: String,
    items: Vec<String>,
    health: u32
}
```

这里的 `#[derive]` 属性会让编译多花费一点儿时间，所以当你在 `Cargo.toml` 文件中将它列为依赖项时，要明确要求 `serde` 支持它。下面是要用在上述代码中的内容：

```
[dependencies]
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
```

有关详细信息，请参阅 `serde` 文档。简而言之，构建系统会为 `Player` 自动生成 `serde::Serialize` 和 `serde::Deserialize` 的实现，因此序列化 `Player` 的值很简单：

```
serde_json::to_writer(&mut std::io::stdout(), &player)?;
```

输出如下所示。

```
{"location":"Cobble Crawl","items":["a wand"],"health":3}
```

18.2 文件与目录

前面已经展示了如何使用读取器和写入器，接下来的几节将介绍 Rust 用于处理文件和目录的特性，这些特性位于 `std::path` 模块和 `std::fs` 模块中。所有这些特性都涉及使用文件名，因此我们将从文件名类型开始。

18.2.1 `OsStr` 与 `Path`

麻烦的是，操作系统并不会强制要求其文件名是有效的 Unicode。下面是创建文本文件的两个 Linux shell 命令。第一个使用了有效的 UTF-8 文件名，第二个则没有：

```
$ echo "hello world" > ô.txt
$ echo "O brave new world, that has such filenames in't" >
$'\xf4'.txt
```

这两个命令都没有任何报错就通过了，因为 Linux 内核并不检查 UTF-8 的格式有效性。对内核来说，任意字节串（除了 null 字节和斜杠）都是可接受的文件名。在 Windows 上的情况类似：几乎任意 16 位“宽字符”字符串都是可接受的文件名，即使字符串不是有效的 UTF-16 也可以。操作系统处理的其他字符串也是如此，比如命令行参数和环境变量。

Rust 字符串始终是有效的 Unicode。文件名在实践中几乎总是 Unicode，但 Rust 必须以某种方式处理罕见的例外情况。这就是 Rust 会有 `std::ffi::OsStr` 和 `OsString` 的原因。

`OsStr` 是一种字符串类型，它是 UTF-8 的超集。`OsStr` 的任务是表示当前系统上的所有文件名、命令行参数和环境变量，无论它们是不是有效的 Unicode。在 Unix 上，`OsStr` 可以保存任意字节序列。在 Windows 上，`OsStr` 使用 UTF-8 的扩展格式存储，可以对任意 16 位值序列（包括不符合标准的半代用区码点）进行编码。

所以我们有两种字符串类型：`str` 用于实际的 Unicode 字符串，而 `OsStr` 用于操作系统可能抛出的任意文字。还有用于文件名的 `std::path::Path`，这纯粹是一个便捷名称。`Path` 与 `OsStr` 完全一样，只是添加了许多关于文件名的便捷方法，18.2.2 节会介绍这些方法。绝对路径和相对路径都使用 `Path` 表示。对于路径中的单个组件，请使用 `OsStr`。

最后，每种字符串类型都有对应的拥有型版本：`String` 拥有分配在堆上的 `str`，`std::ffi::OsString` 拥有分配在堆上的 `OsStr`，而 `std::path::PathBuf` 拥有分配在堆上的 `Path`。表 18-1 概述了每种类型的一些特性。

表 18-1：文件名类型

	str	OsStr	Path
无固定大小类型，总是通过引用传递	是	是	是
可以包含任意 Unicode 文本	是	是	是
通常看起来和 UTF-8 一样	是	是	是
可以包含非 Unicode 数据	否	是	是
文本处理类方法	是	否	否
文件名相关方法	否	否	是

	str	OsStr	Path
拥有型、可增长且分配在堆上的等价类型	String	OsString	PathBuf
转换为拥有型类型	.to_string()	.to_os_string()	.to_path_buf()

所有这 3 种类型都实现了一个公共特型 `AsRef<Path>`，因此我们可以轻松地声明一个接受“任意文件名类型”作为参数的泛型函数。这使用了 13.7 节中展示过的技巧：

```
use std::path::Path;
use std::io;

fn swizzle_file<P>(path_arg: P) -> io::Result<()>
    where P: AsRef<Path>
{
    let path = path_arg.as_ref();
    ...
}
```

所有接受 `path` 参数的标准函数和方法都使用了这种技巧，因此可以直接将字符串字面量传给它们中的任意一个。

18.2.2 Path 与 PathBuf 的方法

`Path` 提供了以下方法。

`Path::new(str)`（新建）

将 `&str` 或 `&OsStr` 转换为 `&Path`。这不会复制字符串。新的 `&Path` 会指向与原始 `&str` 或 `&OsStr` 相同的字节。

```
use std::path::Path;
let home_dir = Path::new("/home/fwolfe");
```

（类似的方法 `OsStr::new(str)` 会将 `&str` 转换为 `&OsStr`。）

`path.parent()`（父目录）

返回路径的父目录（如果有的话）。返回类型是 `Option<&Path>`。

这不会复制路径。`path` 的父路径一定是 `path` 的子串。

```
assert_eq!(Path::new("/home/fwolfe/program.txt").parent(),
           Some(Path::new("/home/fwolfe")));
```

`path.file_name()`（文件名）

返回 `path` 的最后一个组件（如果有的话）。返回类型是 `Option<&OsStr>`。

典型情况下，`path` 由目录、斜杠和文件名组成，此方法会返回文件名。

```
use std::ffi::OsStr;
assert_eq!(Path::new("/home/fwolfe/program.txt").file_name(),
           Some(OsStr::new("program.txt")));
```

`path.is_absolute()`（是绝对路径？）和
`path.is_relative()`（是相对路径？）

这两个方法会指出文件是绝对路径（如 Unix 路径 `/usr/bin/advent` 或 Windows 路径 `C:\Program Files`）还是相对路径（如 `src/main.rs`）。

`path1.join(path2)`（联结）

联结两个路径，返回一个新的 `PathBuf`：

```
let path1 = Path::new("/usr/share/dict");
assert_eq!(path1.join("words"),
           Path::new("/usr/share/dict/words"));
```

如果 `path2` 本身是绝对路径，则只会返回 `path2` 的副本，因此该方法可用于将任意路径转换为绝对路径。

```
let abs_path = std::env::current_dir()?.join(any_path);
```

`path.components()`（组件迭代器）

返回从左到右访问给定路径各个组件的迭代器。这个迭代器的条目类型是 `std::path::Component`，这是一个枚举，可以代表所有可能出现在文件名中的不同部分：

```
pub enum Component<'a> {
    Prefix(PrefixComponent<'a>), // 驱动器路径或共享路径（在Windows
    // 上）
    RootDir, // 根目录，`/`或`\\`
    CurDir, // 特殊目录`.`
    ParentDir, // 特殊目录`..`
    Normal(&'a OsStr) // 普通文件或目录名
}
```

例如，Windows 路径 `\\venice\Music\A Love Supreme\04-Psalm.mp3` 包含一个表示 `\\venice\Music` 的 `Prefix`，后跟一个 `RootDir`，然后是表示 `A Love Supreme` 和 `04-Psalm.mp3` 的两个 `Normal` 组件。

有关详细信息，请参阅在线文档。

`path.ancestors()`（祖先迭代器）

返回一个从 `path` 开始一直遍历到根路径的迭代器。生成的每个条目也都是 `Path`：首先是 `path` 本身，然后是它的父级，接下来是它的祖父级，以此类推：

```
let file = Path::new("/home/jimb/calendars/calendar-18x18.pdf");
assert_eq!(file.ancestors().collect::<Vec<_>>(),
           vec![Path::new("/home/jimb/calendars/calendar-
18x18.pdf"),
               Path::new("/home/jimb/calendars"),
               Path::new("/home/jimb"),
               Path::new("/home"),
               Path::new("/")]);
```

这就像在重复调用 `parent` 直到它返回 `None`。最后一个条目始终是根路径或前缀路径（`Prefix`）。

这些方法只针对内存中的字符串进行操作。Path 也有一些能查询文件系统的方

法：`.exists()`、`.is_file()`、`.is_dir()`、`.read_dir()`、`.canonicalize()` 等。请参阅在线文档以了解更多信息。

将 Path 转换为字符串有以下 3 个方法，每个方法都容许 Path 中存在无效 UTF-8。

`path.to_str()`（转字符串）

将 Path 转换为字符串，返回 `Option<&str>`。如果 path 不是有效的 UTF-8，则返回 `None`。

```
if let Some(file_str) = path.to_str() {
    println!("{}", file_str);
} // .....否则就跳过这种名称古怪的文件
```

`path.to_string_lossy()`（转字符串，宽松版）

基本上和上一个方法一样，但该方法在任何情况下都会设法返回某种字符串。如果 path 不是有效的 UTF-8，则该方法会制作一个副本，用 Unicode 代用字符 `U+FFFD`（'❖'）替代每个无效的字节序列。

返回类型为 `std::borrow::Cow<str>`：借用或拥有的字符串。要从此值获取 `String`，请使用其 `.to_owned()` 方法。（有关 `Cow` 的更多信息，请参阅 13.12 节。）

path.display()（转显示）

用于打印路径：

```
println!("Download found. You put it in: {}", dir_path.display());
```

此方法返回的值不是字符串，但它实现了 `std::fmt::Display`，因此可以与 `format!()`、`println!()` 和类似的宏一起使用。如果路径不是有效的 UTF-8，则输出可能包含  字符。

18.2.3 访问文件系统的函数

表 18-2 展示了 `std::fs` 中的一些函数及其在 Unix 和 Windows 上的近乎等价方式。所有这些函数都会返回 `io::Result` 值。除非另行说明，否则它们的返回值都是 `Result<()>`。

表 18-2：文件系统访问函数汇总表

	Rust 函数	Unix	Windows
创建和删除	<code>create_dir(path)</code> <code>create_dir_all(path)</code> <code>remove_dir(path)</code> <code>remove_dir_all(path)</code> <code>remove_file(path)</code>	<code>mkdir()</code> 类似 <code>mkdir</code> <code>-p</code> <code>rmdir()</code> 类似 <code>rm -r</code> <code>unlink()</code>	<code>CreateDirectory()</code> 类似 <code>mkdir</code> <code>RemoveDirectory()</code> 类似 <code>rmdir /s</code> <code>DeleteFile()</code>
复制、移动和链接	<code>copy(src_path, dest_path) -> Result<u64></code> <code>rename(src_path, dest_path)</code> <code>hard_link(src_path, dest_path)</code>	类似 <code>cp -p</code> <code>rename()</code> <code>link()</code>	<code>CopyFileEx()</code> <code>MoveFileEx()</code> <code>CreateHardLink()</code>

	Rust 函数	Unix	Windows
检查	<code>canonicalize(path) -> Result<PathBuf></code> <code>metadata(path) -> Result<Metadata></code> <code>symlink_metadata(path) -> Result<Metadata></code> <code>read_dir(path) -> Result<ReadDir></code> <code>read_link(path) -> Result<PathBuf></code>	<code>realpath()</code> <code>stat()</code> <code>lstat()</code> <code>opendir()</code> <code>readlink()</code>	<code>GetFinalPathNameByHandle()</code> <code>GetFileInformationByHandle()</code> <code>GetFileInformationByHandle()</code> <code>FindFirstFile()</code> <code>FSCTL_GET_REPARSE_POINT</code>
权限	<code>set_permissions(path, perm)</code>	<code>chmod()</code>	<code>SetFileAttributes()</code>

(`copy()` 返回的数值是已复制文件的大小，以字节为单位。要创建符号链接，请参阅 18.2.5 节。)

如你所见，Rust 会努力提供可移植的函数，这些函数可以在 Windows 以及 macOS、Linux 和其他 Unix 系统上如预期般工作。

关于文件系统的完整教程超出了本书的范畴，如果你对这些函数中的任何一个感到好奇，可以轻松地在网上找到更多相关信息。18.2.4 节中会展示一些示例。

所有这些函数都是通过调用操作系统实现的。例如，`std::fs::canonicalize(path)` 不仅会使用字符串处理来从给定的 `path` 中消除 `.` 和 `..`，还会使用当前工作目录解析相对路径，并追踪符号链接。如果路径不存在，则会报错。

`std::fs::metadata(path)` 和 `std::fs::symlink_metadata(path)` 生成的 `Metadata` 类型包含文件类型和大小、权限、时间戳等信息。同样，可以参阅在线文档了解详细信息。

为便于使用，Path 类型将其中一些内置成了方法，比如，`path.metadata()` 和 `std::fs::metadata(path)` 是一样的。

18.2.4 读取目录

要列出目录的内容，请使用 `std::fs::read_dir` 或 Path 中的等效方法 `.read_dir()`：

```
for entry_result in path.read_dir()? {  
    let entry = entry_result?;  
    println!("{}", entry.file_name().to_string_lossy());  
}
```

注意，在这段代码中有两行用到了 `?` 运算符。第 1 行检查了打开目录时的错误。第 2 行检查了读取下一个条目时的错误。

`entry` 的类型是 `std::fs::DirEntry`，这个结构体提供了数个方法。

`entry.file_name()`（文件名）

文件或目录的名称，是 `OsString` 类型的。

`entry.path()`（路径）

与 `entry.file_name()` 基本相同，但 `entry.path()` 联结了原始路径，生成了一个新的 `PathBuf`。如果正在列出的目录是 `"/home/jimb"`，并且 `entry.file_name()` 是 `".emacs"`，那么 `entry.path()` 将返回 `PathBuf::from("/home/jimb/.emacs")`。

`entry.file_type()`（文件类型）

返回 `io::Result<FileType>`。FileType 有 `.is_file()` 方法、`.is_dir()` 方法和 `.is_symlink()` 方法。

`entry.metadata()`（元数据）

获取有关此条目的其他元数据。

特殊目录 `.` 和 `..` 在读取目录时不会列出。

下面是一个更接近现实的例子。以下代码会递归地将目录树从磁盘上的一个位置复制到另一个位置。

```
use std::fs;
use std::io;
use std::path::Path;

/// 把现有目录`src`复制到目标路径`dst`
fn copy_dir_to(src: &Path, dst: &Path) -> io::Result<()> {
    if !dst.is_dir() {
        fs::create_dir(dst)?;
    }

    for entry_result in src.read_dir()? {
        let entry = entry_result?;
        let file_type = entry.file_type()?;
        copy_to(&entry.path(), &file_type,
&dst.join(entry.file_name()))?;
    }

    Ok(())
}
```

独立的 `copy_to` 函数用于复制单个目录条目。

```
/// 把`src`中的任何内容复制到目标路径`dst`
fn copy_to(src: &Path, src_type: &fs::FileType, dst: &Path)
    -> io::Result<()>
{
    if src_type.is_file() {
        fs::copy(src, dst)?;
    } else if src_type.is_dir() {
        copy_dir_to(src, dst)?;
    } else {
        return Err(io::Error::new(io::ErrorKind::Other,
format!("don't know how to copy:
{}"),
src.display())));
    }
}
```



```
}  
Ok()  
}
```

18.2.5 特定于平台的特性

到目前为止，`copy_to` 函数既可以复制文件也可以复制目录。接下来我们还打算在 Unix 上支持符号链接。

没有可移植的方法来创建同时适用于 Unix 和 Windows 的符号链接，但标准库提供了一个特定于 Unix 的 `symlink` 函数：

```
use std::os::unix::fs::symlink;
```

有了这个函数，我们的工作就很容易了。只需向 `copy_to` 中的 `if` 表达式添加一个分支即可：

```
...  
} else if src_type.is_symlink() {  
    let target = src.read_link()?;  
    symlink(target, dst)?;  
...  

```

如果只是为 Unix 系统（如 Linux 和 macOS）编译我们的程序，那么就能这么用。

`std::os` 模块包含各种特定于平台的特性，比如 `symlink`。
`std::os` 在标准库中的实际主体如下所示（这里为了看起来整齐，调整了代码格式）：

```
//! 特定于操作系统的功能  
  
#[cfg(unix)]           pub mod unix;  
#[cfg(windows)]        pub mod windows;  
#[cfg(target_os = "ios")] pub mod ios;  
#[cfg(target_os = "linux")] pub mod linux;  
#[cfg(target_os = "macos")] pub mod macos;  
...  

```

`#[cfg]` 属性表示条件编译：这些模块中的每一个仅在某些平台上存在。这就是为什么使用 `std::os::unix` 修改后的程序只能针对 Unix 成功编译，因为在其他平台上 `std::os::unix` 不存在。

如果希望代码在所有平台上编译，并支持 Unix 上的符号链接，则必须也在程序中使用 `#[cfg]`。在这种情况下，最简单的方法是在 Unix 上导入 `symlink`，同时在其他系统上定义自己的 `symlink` 模拟实现：

```
#[cfg(unix)]
use std::os::unix::fs::symlink;

/// 在未提供`symlink`的平台上提供的模拟实现
#[cfg(not(unix))]
fn symlink<P: AsRef<Path>, Q: AsRef<Path>>(src: P, _dst: Q)
    -> std::io::Result<()>
{
    Err(io::Error::new(io::ErrorKind::Other,
                       format!("can't copy symbolic link: {}",
                               src.as_ref().display())))
}
```

事实上，`symlink` 只是特殊情况。大多数特定于 Unix 的特性不是独立函数，而是将新方法添加到标准库类型的扩展特型（11.2.2 节介绍过扩展特型）。`prelude` 模块可用于同时启用所有这些扩展：

```
use std::os::unix::prelude::*;
```

例如，在 Unix 上，这会向 `std::fs::Permissions` 添加 `.mode()` 方法，从而支持表达 Unix 权限所需的底层 `u32` 值。同样，这还会给 `std::fs::Metadata` 添加一些访问器方法，从而得以访问底层 `struct stat` 的字段（比如 `.uid()` 可获得文件所有者的用户 ID）为 `std::fs::Metadata` 添加了访问器。

总而言之，`std::os` 中的内容非常基础。更多特定于平台的功能可通过第三方 crate 获得，比如用于访问 Windows 注册表的 `winreg`。

18.3 网络

关于网络的教程远远超出了本书的范畴。但是，如果你已经对网络编程有所了解，那么本节能帮你开始使用 Rust 中的网络支持。

要编写较底层的网络代码，可以使用 `std::net` 模块，该模块为 TCP 网络和 UDP 网络提供了跨平台支持。可以使用 `native_tls` crate 来支持 SSL/TLS。

这些模块为网络上直接的、阻塞型的输入和输出提供了一些基础构件。用几行代码就可以编写一个简单的服务器，只要使用 `std::net` 并为每个连接启动一个线程即可。例如，下面是一个“回显”（echo）服务器：

```
use std::net::TcpListener;
use std::io;
use std::thread::spawn;

/// 不断接受连接，为每个连接启动一个线程
fn echo_main(addr: &str) -> io::Result<()> {
    let listener = TcpListener::bind(addr)?;
    println!("listening on {}", addr);
    loop {
        // 等待客户端连入
        let (mut stream, addr) = listener.accept()?;
        println!("connection received from {}", addr);

        // 启动一个线程来处理此客户端
        let mut write_stream = stream.try_clone()?;
        spawn(move || {
            // 回显从`stream`中收到的一切
            io::copy(&mut stream, &mut write_stream)
                .expect("error in client thread: ");
            println!("connection closed");
        });
    }
}

fn main() {
    echo_main("127.0.0.1:17007").expect("error: ");
}
```

回显服务器会简单地重复发给它的所有内容。这种代码与用 Java 或 Python 编写的代码没有太大区别。（第 19 章会介绍 `std::thread::spawn()`。）

但是，对于高性能服务器，需要使用异步输入和输出。第 20 章会介绍 Rust 对异步编程的支持，并展示网络客户端和服务器的完整代码。

更高层级的协议由第三方 crate 提供支持。例如，`reqwest` crate 为 HTTP 客户端提供了一个漂亮的 API。下面是一个完整的命令程序，该程序可以通过 `http:` 或 `https:` URL 获取对应文档，并将其内容打印到你的终端。此代码是使用 `reqwest = "0.11"` 编写的，并启用了其 `"blocking"` 特性。`reqwest` 还提供了一个异步接口。

```
use std::error::Error;
use std::io;

fn http_get_main(url: &str) -> Result<(), Box<dyn Error>> {
    // 发送HTTP请求并获取响应
    let mut response = reqwest::blocking::get(url)?;
    if !response.status().is_success() {
        Err(format!("{}", response.status()))?;
    }

    // 读取响应体并写到标准输出
    let stdout = io::stdout();
    io::copy(&mut response, &mut stdout.lock())?;
    Ok(())
}

fn main() {
    let args: Vec<String> = std::env::args().collect();
    if args.len() != 2 {
        eprintln!("usage: http-get URL");
        return;
    }

    if let Err(err) = http_get_main(&args[1]) {
        eprintln!("error: {}", err);
    }
}
```

actix-web 框架为 HTTP 服务器提供了一些高层次抽象，比如 Service 特型和 Transform 特型，这两个特型可以帮助你从一些可插接部件组合出应用程序。websocket crate 实现了 WebSocket 协议。Rust 是一门年轻的语言，拥有繁荣的开源生态系统，对网络的支持正在迅速扩展。

第 19 章 并发

长远看来，不建议使用面向机器的语言编写大型并发程序，因为面向机器的语言允许不受限制地使用存储位置及其地址。这就意味着，即使借助复杂的硬件机制，我们也根本没有能力确保程序的可靠性。

——Per Brinch Hansen（1977 年）

“通信”的模式就是“并行”的模式。

——Whit Morriss

如果你看待并发的态度在职业生涯中发生过变化，那你并不是特例。这种现象太常见了。

起初，编写并发代码看起来既简单又有趣。线程、锁、队列等工具很容易上手，固然其中也有很多陷阱，但幸运的是我们知道都有哪些陷阱，并且多加小心就不会出错。

我们总有不得不断其他人的多线程代码的情况，这时候，你只能得出以下结论：**某些人确实不适合使用这些工具。**

但迟早，你不得不断自己的多线程代码。

过去的教训告诉你，如果还没有对多线程技术彻底失望，那么至少也应该对所有多线程代码保持适度的警惕。你偶尔会碰上几篇文章详细解释为什么一些看似正确的多线程惯用法却根本不起作用（与“内存模型”有关），这又进一步强化了这种警惕。但是，你最终会找到一种自己用起来顺手且不会经常出错的并发惯用法。你会把几乎所有经验都塞进那个惯用法中，并且，如果你真的很厉害，那么还能对凭空增加的复杂性说“不”。

当然，还有很多惯用法。系统程序员常用的方法包括以下几种。

- 具有单一作业的后台线程，需要定期唤醒执行作业。
- 通过任务队列与客户端通信的通用工作池。
- 管道，数据在其中从一个线程流向下一个线程，每个线程只负责一部分工作。
- 数据并行处理，假设（无论对错）整个计算机只进行一次主要的大型计算，将这次计算分成 n 个部分且在 n 个线程上运行，并期望机器的所有 n 个核心都能立即开始工作。
- 同步复杂对象关系，其中多个线程可以访问相同的数据，并且使用基于互斥锁等底层原语的临时加锁方案避免了竞争。（Java 内置了对此模型的支持，它曾在 20 世纪 90 年代和 21 世纪初非常流行。）
- 原子化整数操作允许多个核心借助一个机器字大小的字段传递信息来进行通信。（这种惯用法比其他所有方法更难正确使用，除非要交换的数据恰好是整数值，但实际上，数据通常是指针。）

随着时间的推移，你已经对其中的几种方法非常娴熟，还能彼此相安无事地组合使用它们——简直就是艺术大师！如果其他人不会以任何方式修改你的系统，那么就能岁月静好——然而，尽管这些程序可以很好地利用线程，但其中充满了“潜规则”。

Rust 提供了一种更好的并发处理方式，不是强制所有程序采用单一风格（对系统程序员来说这可算不上什么解决方案），而是安全地支持多种风格。通过代码把“潜规则”写出来并由编译器强制执行。

你可能听说过 Rust 能让你编写安全、快速、并发的程序。本章将向你展示它是如何做到的。我们将介绍 3 种使用 Rust 线程的方法。

- 分叉与合并（fork-join）并行
- 通道
- 共享可变状态

在此过程中，你会用上迄今为止学过的有关 Rust 语言的所有知识。Rust 对引用、可变性和生命周期的处理方式在单线程程序中已经足够有价值了，但在并发编程中，这些规则的意义才开始真正显现。它们

会扩展你的工具箱，让快速而正确地编写各种风格的多线程代码成为可能——不再怀疑，不再愤世嫉俗，不再恐惧。

19.1 分叉与合并并行

当我們有几个完全独立的任務想要同时完成时，线程最简单的用例就出现了。

假设我们正在对大量文档进行自然语言处理。可以写这样一个循环：

```
fn process_files(filenamees: Vec<String>) -> io::Result<()> {  
    for document in filenamees {  
        let text = load(&document)?;    // 读取源文件  
        let results = process(text);    // 计算统计信息  
        save(&document, results)?;      // 写入输出文件  
    }  
    Ok(())  
}
```

图 19-1 展示了这个程序的执行过程。

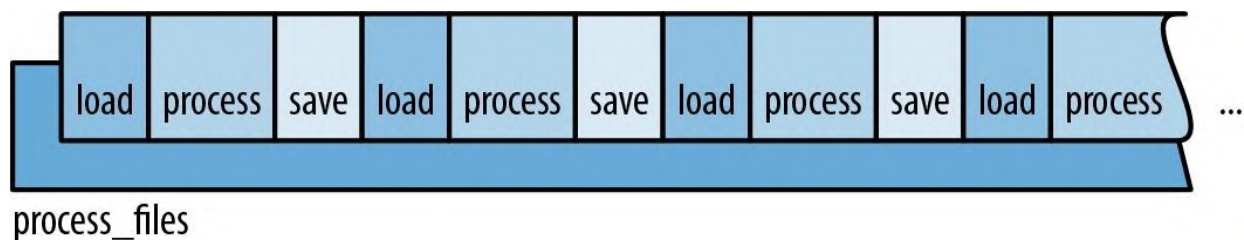


图 19-1: `process_files()` 的单线程执行

由于每个文档都是单独处理的，因此要想加快任务处理速度，可以将语料库分成多个块并在单独的线程上处理每个块，如图 19-2 所示。

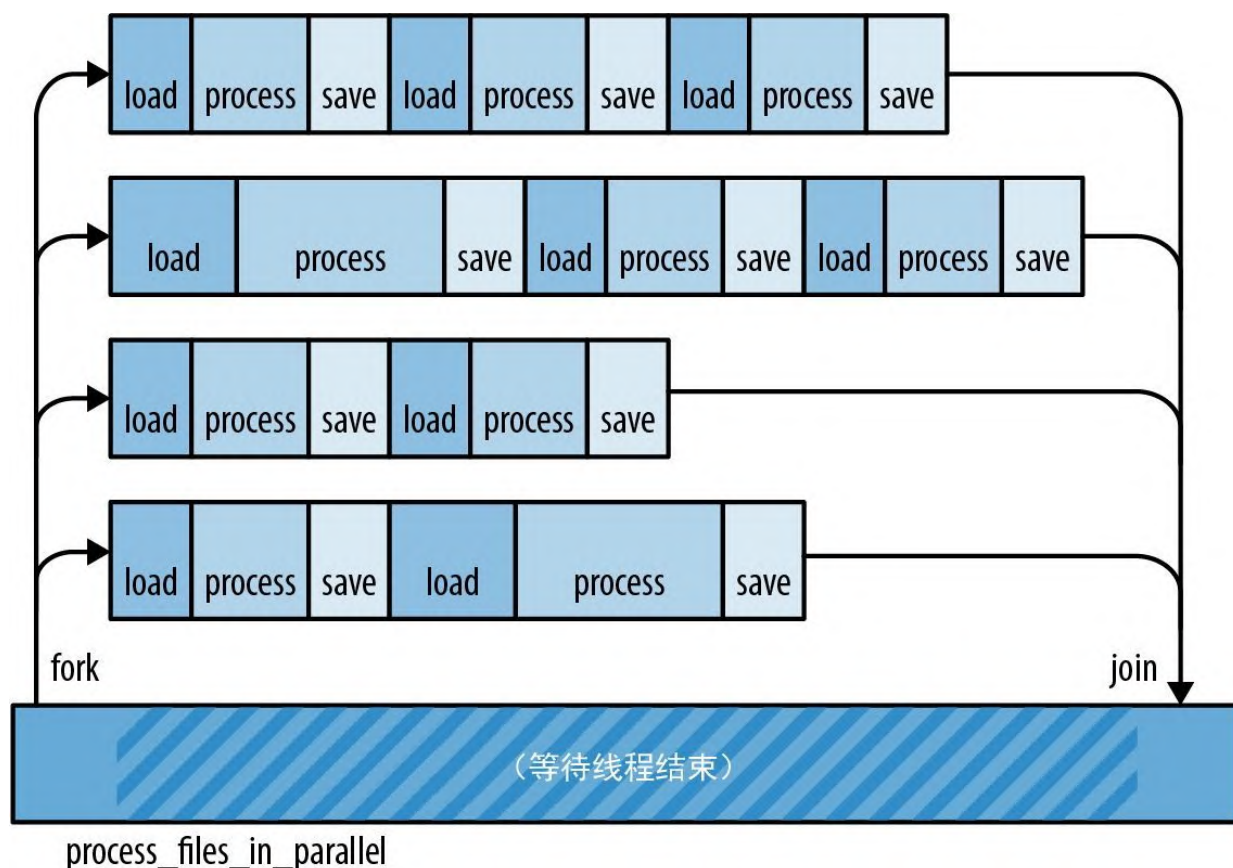


图 19-2：使用分叉与合并方法的多线程文件处理

这种模式称为分叉与合并并行。`fork`（分叉）是启动一个新线程，`join`（合并）是等待线程完成。我们已经见过这种技术：第 2 章中曾用它来加速曼德博程序。

出于以下几个原因，分叉与合并并行很有吸引力。

- 非常简单。分叉与合并很容易实现，在 Rust 中更不容易写错。
- 避免了瓶颈。分叉与合并中没有对共享资源的锁定。任何线程只会在最后一步才不得不等待另一个线程。同时，每个线程都可以自由运行。这有助于降低任务切换开销。
- 这种模式在性能方面的数学模型对程序员来说比较直观。在最好的情况下，通过启动 4 个线程，我们只花 $1/4$ 的时间就能完成原本的工作。图 19-2 展示了不应该期望这种理想加速的一个原因：我们可能无法在所有线程之间平均分配工作。另一个需要注意的原因是，有时分叉与合并程序必须在线程联结后花费一些时

间来组合各线程的计算结果。也就是说，完全隔离这些任务可能会产生一些额外的工作。不过，除了这两个原因，任何具有独立工作单元的 CPU 密集型程序都可以获得显著的性能提升。

- 很容易推断出程序是否正确。只要线程真正隔离了，分叉与合并程序就是**确定性的**，就像曼德博程序中的计算线程一样。无论线程速度如何变化，程序总会生成相同的结果。这是一个没有竞态条件的并发模型。

分叉与合并的主要缺点是要求工作单元彼此隔离。本章在后面会考虑一些无法完全隔离的问题。

现在，继续以自然语言处理为例。我们将展示几种将分叉与合并模式应用于 `process_files` 函数的方法。

19.1.1 启动与联结

函数 `std::thread::spawn` 会启动一个新线程：

```
use std::thread;

thread::spawn(|| {
    println!("hello from a child thread");
});
```

它会接受一个参数，即一个 `FnOnce` 闭包或函数型的参数。Rust 会启动一个新线程来运行该闭包或函数的代码。新线程是一个真正的操作系统线程，有自己的栈，就像 C++、C#、Java 中的线程一样。

下面是一个更实际的例子，它使用 `spawn` 实现了之前的 `process_files` 函数的并行版本：

```
use std::{thread, io};

fn process_files_in_parallel(filenamees: Vec<String>) ->
io::Result<()> {
    // 把工作拆分成几块
    const NTHREADS: usize = 8;
    let worklists = split_vec_into_chunks(filenamees, NTHREADS);
```

```

// 分叉：启动一个线程来处理每一个块
let mut thread_handles = vec![];
for worklist in worklists {
    thread_handles.push(
        thread::spawn(move || process_files(worklist))
    );
}

// 联结：等待所有线程结束
for handle in thread_handles {
    handle.join().unwrap()?;
}

Ok(())
}

```

下面来逐行分析一下这个函数。

```

fn process_files_in_parallel(filenamees: Vec<String>) ->
io::Result<()> {

```

我们的新函数与原始 `process_files` 具有相同的类型签名，这样它就是一个方便的无缝替代品了。

```

// 把工作拆分成几块
const NTHREADS: usize = 8;
let worklists = split_vec_into_chunks(filenamees, NTHREADS);

```

我们使用了尚未展示过的实用函数 `split_vec_into_chunks` 来拆分工作。它的返回值 `worklists` 是由向量组成的向量，其中包含从原始向量 `filenamees` 中均分出来的 8 个部分。

```

// 分叉：启动一个线程来处理每一个块
let mut thread_handles = vec![];
for worklist in worklists {
    thread_handles.push(
        thread::spawn(move || process_files(worklist))
    );
}

```

我们会为每个 `worklist` 启动一个线程。`spawn()` 会返回一个名为 `JoinHandle` 的值，稍后会用到。现在，先将所有 `JoinHandle` 放

入一个向量中。

请注意我们是如何将文件名列表放入工作线程的。

- 在父线程中，通过 `for` 循环来定义和填充 `worklist`。
- 一旦创建了 `move` 闭包，`worklist` 就会被移动到此闭包中。
- 然后 `spawn` 会将闭包（内含 `worklist` 向量）转移给新的子线程。

这些操作开销很低。就像第 4 章中讨论过的 `Vec<String>` 移动一样，`String` 没有被克隆。事实上，这个过程中并没有发生任何分配和释放。唯一移动的数据是 `vec` 本身，只有 3 个机器字。

我们创建的每个线程几乎都需要代码和数据才能启动。Rust 闭包可以方便地包含我们想要的任何代码和数据。

继续看下面的代码：

```
// 联结：等待所有线程结束
for handle in thread_handles {
    handle.join().unwrap()?;
}
```

我们使用之前收集的 `JoinHandle` 的 `.join()` 方法来等待所有 8 个线程完成。联结这些线程对于保证程序的正确性是必要的，因为 Rust 程序会在 `main` 返回后立即退出，即使其他线程仍在运行。这些线程并不会调用析构器，而是直接被“杀死”了。如果这不是你想要的结果，请确保在从 `main` 返回之前联结了任何你关心的线程。

如果我们通过了这个循环，则意味着所有 8 个子线程都成功完成了。因此，该函数会以返回 `Ok(())` 结束。

```
Ok(())
}
```

19.1.2 跨线程错误处理

由于要做错误处理，我们在示例中用于联结子线程的代码比看起来更棘手。再重温一下那行代码：

```
handle.join().unwrap()?;
```

`.join()` 方法为我们做了两件事。

首先，`handle.join()` 会返回 `std::thread::Result`，如果子线程出现了 `panic`，就返回一个错误（`Err`）。这使得 Rust 中的线程比 C++ 中的线程更加健壮。在 C++ 中，越界数组访问是未定义行为，并且无法保护系统的其余部分免受后果的影响。在 Rust 中，`panic` 是安全且局限于每个线程的。线程之间的边界充当着 `panic` 的防火墙，`panic` 不会自动从一个线程传播到依赖它的其他线程。相反，一个线程中的 `panic` 在其他线程中会报告为错误型 `Result`。程序整体而言很容易恢复。

不过，在本程序中，我们不会尝试任何花哨的 `panic` 处理，而是会立即在 `Result` 上使用 `.unwrap()`，断言它是一个 `Ok` 结果而不是 `Err` 结果。如果一个子线程确实发生了 `panic`，那么这个断言就会失败，所以父线程也会出现 `panic`。如此一来，我们就显式地将 `panic` 从子线程传播到了父线程。

其次，`handle.join()` 会将子线程的返回值传回父线程。我们传给 `spawn` 的闭包的返回类型是 `io::Result<()>`，因为它就是 `process_files` 返回值的类型。此返回值不会被丢弃。当子线程完成时，它的返回值会被保存下来，并且 `JoinHandle::join()` 会把该值传回父线程。

在这个程序中，`handle.join()` 返回的完整类型是 `std::thread::Result<std::io::Result<()>>`。`thread::Result` 是 `spawn/joinAPI` 的一部分，而 `io::Result` 是我们的应用程序的一部分。

在这个例子中，展开（`unwrap`）`thread::Result` 之后，我们就用 `io::Result` 上的 `?` 运算符显式地将 I/O 错误从子线程传播到了父线程。

所有这些看起来可能相当琐碎。但如果只把它当作一行代码，则可以与别的语言对比一下。Java 和 C# 中的默认行为是子线程中的异常会转储到终端，然后被遗忘。在 C++ 中，默认行为是中止进程。在 Rust 中，错误是 `Result` 值（数据）而不是异常（控制流）。它们会像其他值一样跨线程传递。每当你使用底层线程 API 时，最终都必须仔细编写错误处理代码，但如果不得不编写错误处理代码，那么 `Result` 是非常合适的选择。

19.1.3 跨线程共享不可变数据

假设我们正在进行的分析需要一个大型的英语单词和短语的数据库：

```
// 之前
fn process_files(filenamees: Vec<String>)

// 之后
fn process_files(filenamees: Vec<String>, glossary: &GigabyteMap)
```

这个 `glossary` 会很大，所以要通过引用传递它。该如何修改 `process_files_in_parallel` 以便将词汇表传给工作线程呢？

想当然的改法是不行的：

```
fn process_files_in_parallel(filenamees: Vec<String>,
                             glossary: &GigabyteMap)
    -> io::Result<()>
{
    ...
    for worklist in worklists {
        thread_handles.push(
            spawn(move || process_files(worklist, glossary)) // 错
        );
    }
    ...
}
```

我们只给此函数添加了一个 `glossary` 参数并将其传给 `process_files`。Rust 报错说：

```
error: explicit lifetime required in the type of `glossary`
|
38 |         spawn(move || process_files(worklist, glossary))
// 错误
|         ^^^^^ lifetime `'_static` required
```

Rust 对传给 `spawn` 的闭包的生命周期报了错，而编译器在此处显示的“有用”消息实际上根本没有帮助。

`spawn` 会启动独立线程。Rust 无法知道子线程要运行多长时间，因此它假设了最坏的情况：即使在父线程完成并且父线程中的所有值都消失后，子线程仍可能继续运行。显然，如果子线程要持续那么久，那么它运行的闭包也需要持续那么久。但是这个闭包有一个有限的生命周期，它依赖于 `glossary` 引用，而此引用不需要永久存在。

请注意，Rust 拒绝编译此代码是对的。按照我们编写这个函数的方式，一个线程确实有可能遇到 I/O 错误，导致 `process_files_in_parallel` 在其他线程完成之前退出。在主线程释放词汇表后，子线程可能仍然会试图使用词汇表。这将是一场竞赛，如果主线程获胜，就会赢得“未定义行为”这份大奖。而 Rust 不允许发生这种事。

`spawn` 似乎过于开放了，无法支持跨线程共享引用。事实上，我们已经在 14.1.2 节中看到过这样的情况。那时候，解决方案是用 `move` 闭包将数据的所有权转移给新线程。但在这里行不通，因为有许多线程要使用同一份数据。一种安全的替代方案是为每个线程都克隆整个词汇表，但由于词汇表很大，我们不希望这么做。幸运的是，标准库提供了另一种方式：原子化引用计数。

4.4 节介绍过 `Arc`。是时候使用它了：

```
use std::sync::Arc;

fn process_files_in_parallel(filenamees: Vec<String>,
                             glossary: Arc<GigabyteMap>)
    -> io::Result<()>
{
    ...
    for worklist in worklists {
```



```

        // 对.clone()的调用只会克隆Arc并增加引用计数，并不会克隆
GigabyteMap
        let glossary_for_child = glossary.clone();
        thread_handles.push(
            spawn(move || process_files(worklist,
&glossary_for_child))
        );
    }
    ...
}

```

我们更改了 `glossary` 的类型：要执行并行分析，调用者就必须传入 `Arc<GigabyteMap>`，这是指向已使用 `Arc::new(giga_map)` 移入堆中的 `GigabyteMap` 的智能指针。

调用 `glossary.clone()` 时，我们是在复制 `Arc` 智能指针而不是整个 `GigabyteMap`。这相当于增加一次引用计数。

通过此项更改，程序可以编译并运行了，因为它不再依赖于引用的生命周期。只要任何线程拥有 `Arc<GigabyteMap>`，它就会让 `GigabyteMap` 保持存活状态，即使父线程提前退出也没问题。不会有任何数据竞争，因为 `Arc` 中的数据是不可变的。

19.1.4 rayon

标准库的 `spawn` 函数是一个重要的基础构件，但它并不是专门为分叉与合并的并行而设计的，基于它，我们可以封装出更好的分叉与合并式 API。例如，在第 2 章中，我们使用 `crossbeam` 库将一些工作拆分为 8 个线程。`crossbeam` 的作用域线程能非常自然地支持分叉与合并并行。

由 Niko Matsakis 和 Josh Stone 设计的 `rayon`¹ 库是另一个例子。它提供了两种运行并发任务的方式：

¹`rayon` 意为“人造丝”，引申为“人造线程”。——译者注

```
use rayon::prelude::*;
```

```
// “并行做两件事”
let (v1, v2) = rayon::join(fn1, fn2);

// “并行做N件事”
giant_vector.par_iter().for_each(|value| {
    do_thing_with_value(value);
});
```

`rayon::join(fn1, fn2)` 只是调用这两个函数并返回两个结果。`.par_iter()` 方法会创建 `ParallelIterator`，这是一个带有 `map`、`filter` 和其他方法的值，很像 Rust 的 `Iterator`。在这两种情况下，`rayon` 都会用自己的工作线程池来尽可能拆分工作。只要告诉 `rayon` 哪些任务可以并行完成就可以了，`rayon` 会管理这些线程并尽其所能地分派工作。

图 19-3 展示了对 `giant_vector.par_iter().for_each(...)` 调用的两种思考方式。(a) `rayon` 表现得就好像它为向量中的每个元素启动了一个线程。(b) 在幕后，`rayon` 在每个 CPU 核心上都有一个工作线程，这样效率更高。这个工作线程池由程序中的所有线程共享。当成千上万个任务同时进来时，`rayon` 会拆分这些工作。

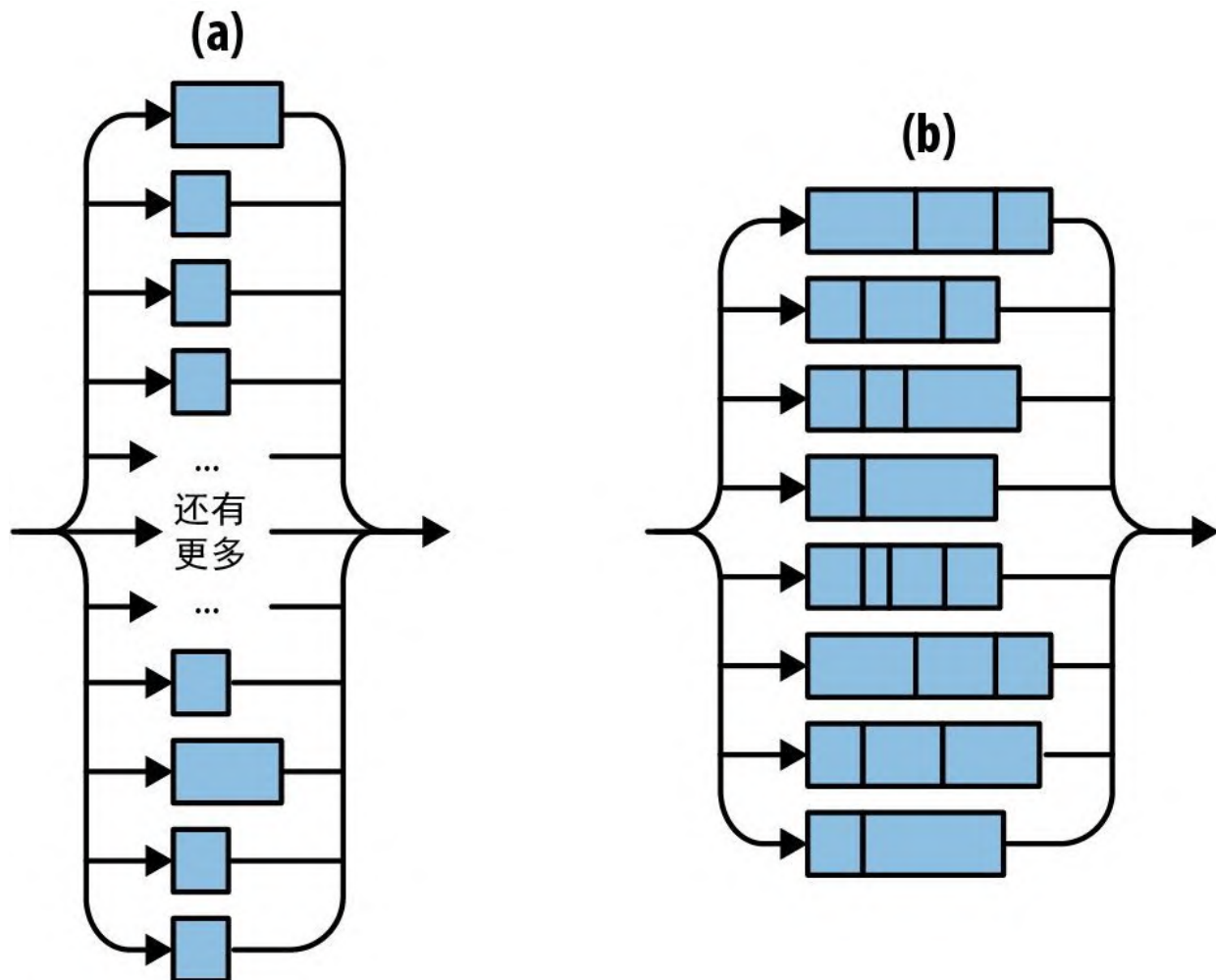


图 19-3：理论上与实践中的 **rayon**

下面是一个使用 rayon 的 `process_files_in_parallel` 版本和一个接受 `Vec<String>` 型而非 `&str` 型参数的 `process_file`:

```
use rayon::prelude::*;

fn process_files_in_parallel(filenamees: Vec<String>, glossary:
&GigabyteMap)
    -> io::Result<()>
{
    filenamees.par_iter()
        .map(|filename| process_file(filename, glossary))
        .reduce_with(|r1, r2| {
            if r1.is_err() { r1 } else { r2 }
        })
}
```

```
    })  
    .unwrap_or(Ok(()))  
}
```

比起使用 `std::thread::spawn` 的版本，这段代码更简短，也不需要很多技巧。我们一行一行地看。

- 首先，用 `filenames.par_iter()` 创建一个并行迭代器。
- 然后，用 `.map()` 在每个文件名上调用 `process_file`。这会在一系列 `io::Result<()>` 型的值上生成一个 `ParallelIterator`。
- 最后，用 `.reduce_with()` 来合并结果。在这里，我们会保留第一个错误（如果有的话）并丢弃其余错误。如果想累积所有的错误或者打印它们，也可以在这里修改。

当传递一个能在成功时返回有用值的 `.map()` 闭包时，`.reduce_with()` 方法也非常好用。这时可以给 `.reduce_with()` 传入一个闭包，指定如何组合两个成功结果。

- `reduce_with` 只有在 `filenames` 为空时才会返回一个为 `None` 的 `Option`。在这种情况下，我们会用 `Option` 的 `.unwrap_or()` 方法来生成结果 `Ok(())`。

在幕后，`rayon` 使用了一种叫作工作窃取的技术来动态平衡线程间的工作负载。相比于 19.1.1 节的手动预先分配工作的方式，这通常能更好地让所有 CPU 都处于忙碌状态。

另外，`rayon` 还支持跨线程共享引用。幕后发生的任何并行处理都能确保在 `reduce_with` 返回时完成。这解释了为什么即使该闭包会在多个线程上调用，也能安全地将 `glossary` 传给 `process_file`。

（顺便说一句，这里使用 `map` 和 `reduce` 这两个方法名并非巧合。由 Google 和 Apache Hadoop 推广的 MapReduce 编程模型与分叉与合并有很多共同点。可以将其看作查询分布式数据的分叉与合并方法。）

19.1.5 重温曼德博集

回想第 2 章，我们曾用分叉与合并并发来渲染曼德博集。这让渲染速度提升了 4 倍，令人印象非常深刻。但考虑到我们让程序在 8 核机器上启动了 8 个工作线程，因此这个速度还不够快。

问题的根源在于我们没有平均分配工作量。计算图像的一个像素相当于运行一个循环（参见 2.6.1 节）。事实上，图像的浅灰色部分（循环会快速退出的地方）比黑色部分（循环会运行整整 255 次迭代的地方）渲染速度要快得多。因此，虽然我们将整个区域划分成了大小相等的水平条带，但创建了不均等的工作负载，如图 19-4 所示。

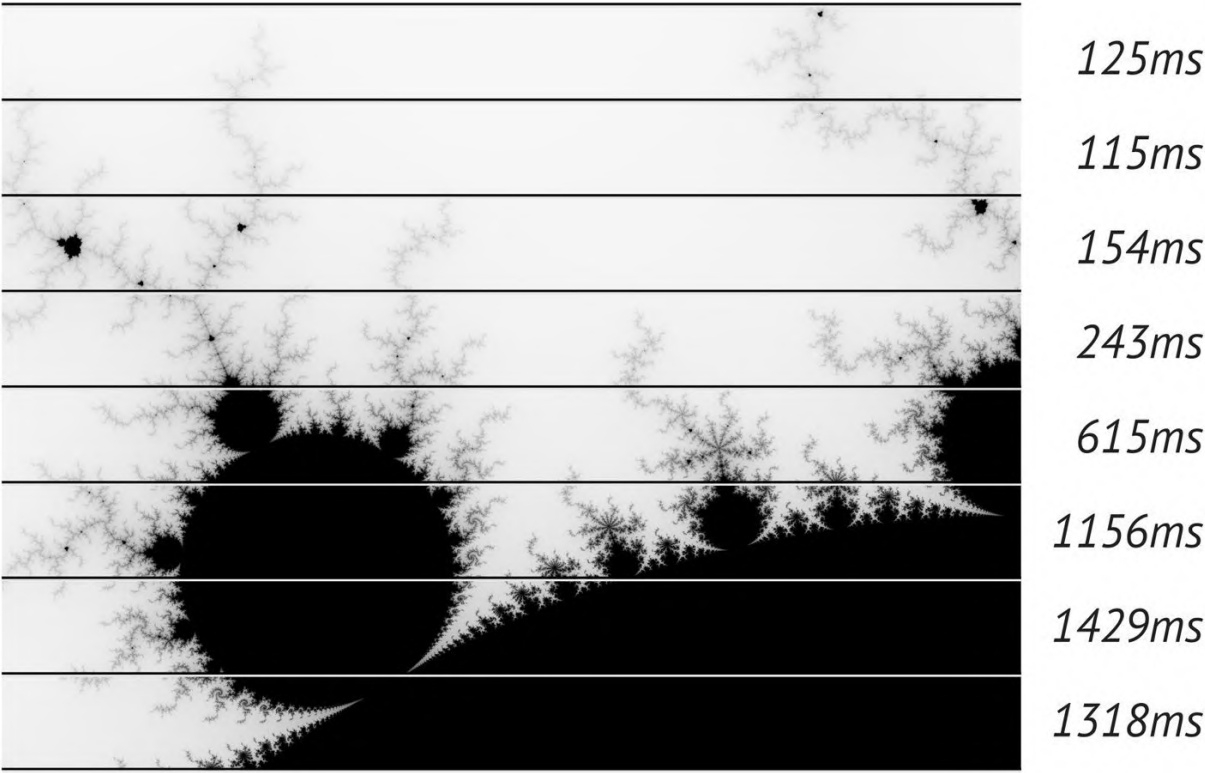


图 19-4：曼德博程序中的工作分配不均等

使用 rayon 很容易解决这个问题。我们可以为输出中的每一行像素启动一个并行任务。这会创建数百个任务，而 rayon 可以在其线程中分配这些任务。有了工作窃取机制，任务的规模是无关紧要的。rayon 会对这些工作进行平衡。

下面是实现代码。第 1 行和最后一行是 2.6.6 节中展示过的 main 函数的一部分，但我们更改了这两行之间的渲染代码：

```
let mut pixels = vec![0; bounds.0 * bounds.1];

// 把`pixels`拆分成一些水平条带
{
    let bands: Vec<(usize, &mut [u8])> = pixels
        .chunks_mut(bounds.0)
        .enumerate()
        .collect();

    bands.into_par_iter()
        .for_each(|(i, band)| {
            let top = i;
            let band_bounds = (bounds.0, 1);
            let band_upper_left = pixel_to_point(bounds, (0, top),
                                                    upper_left,
lower_right);
            let band_lower_right = pixel_to_point(bounds,
(bounds.0, top + 1),
                                                    upper_left,
lower_right);
            render(band, band_bounds, band_upper_left,
band_lower_right);
        });
}

write_image(&args[1], &pixels, bounds).expect("error writing PNG
file");
```

首先，创建 bands，也就是要传给 rayon 的任务集合。每个任务只是一个元组类型 (usize, &mut [u8])：第一个是计算所需的行号，第二个是要填充的 pixels 切片。我们使用 chunks_mut 方法将图像缓冲区分成一些行，enumerate 则会给每一行添加行号，然后 collect 会将所有数值切片对放入一个向量中。（这里需要一个向量，因为 rayon 只能从数组和向量中创建并行迭代器。）

接下来，将 `bands` 转成一个并行迭代器，并使用 `.for_each()` 方法告诉 `rayon` 我们要完成的工作。

由于我们在使用 `rayon`，因此必须将下面这行代码添加到 `main.rs` 中：

```
use rayon::prelude::*;
```

下面是要添加到 `Cargo.toml` 中的内容：

```
[dependencies]
rayon = "1"
```

通过这些更改，现在该程序在 8 核机器上使用了大约 7.75 个核心。速度比以前手动分配工作时快 75%，而且代码更简短。这体现出了让 `crate` 负责工作分配而不是我们自己去完成的好处。

19.2 通道

通道是一种单向管道，用于将值从一个线程发送到另一个线程。换句话说，通道是一个线程安全的队列。

图 19-5 说明了如何使用通道。通道有点儿像 Unix 管道：一端用于发送数据，另一端用于接收数据。两端通常由两个不同的线程拥有。但是，Unix 管道用于发送字节，而通道用于发送 Rust 值。

`sender.send(item)` 会将单个值放入通道，`receiver.recv()` 则会移除一个值。值的所有权会从发送线程转移给接收线程。如果通道为空，则 `receiver.recv()` 会一直阻塞到有值发出为止。

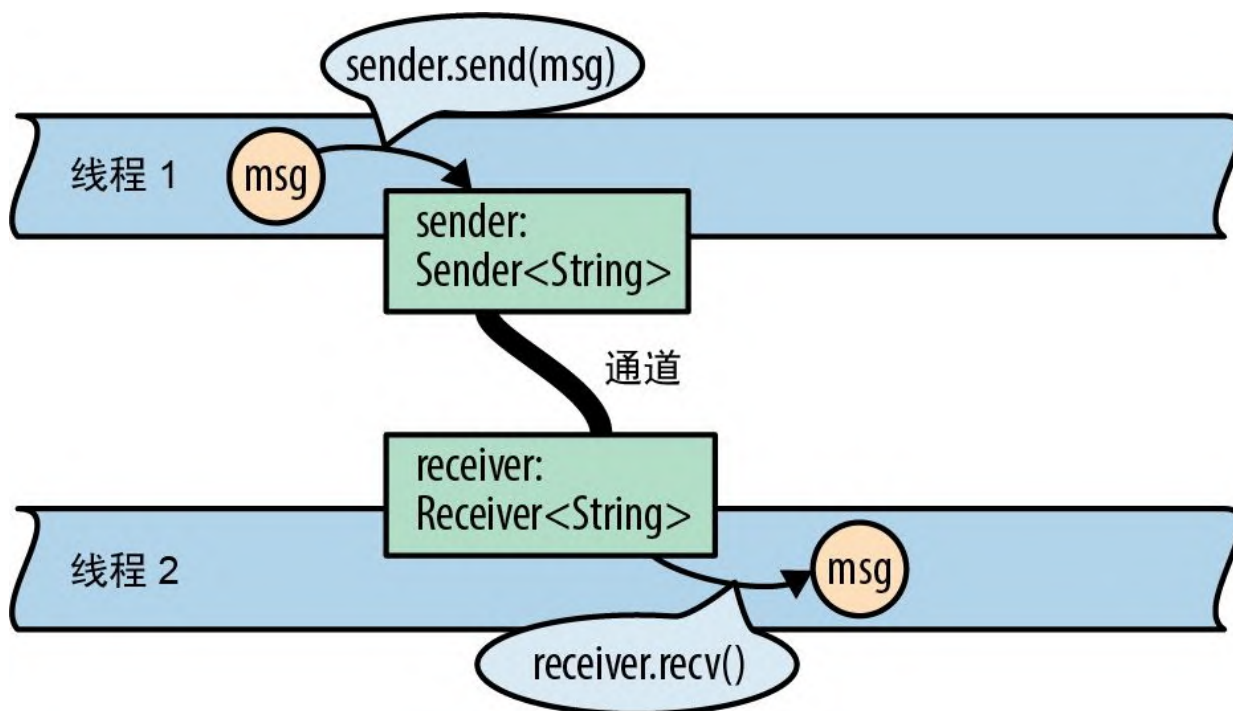


图 19-5: `String` 的通道：字符串 `msg` 的所有权从线程 1 转移给线程 2

使用通道，线程可以通过彼此传值来进行通信。这是线程协同工作的一种非常简单的方法，无须使用锁或共享内存。

这并不是一项新技术。Erlang 中的独立进程和消息传递已经有 30 年历史了。Unix 管道已经有将近 50 年历史了。我们一般会认为管道具有灵活性和可组合性，而没有意识到它还具有并发的特性，但事实上，管道具有上述所有特性。图 19-6 展示了一个 Unix 管道的例子。当然，这 3 个程序也可以同时工作。

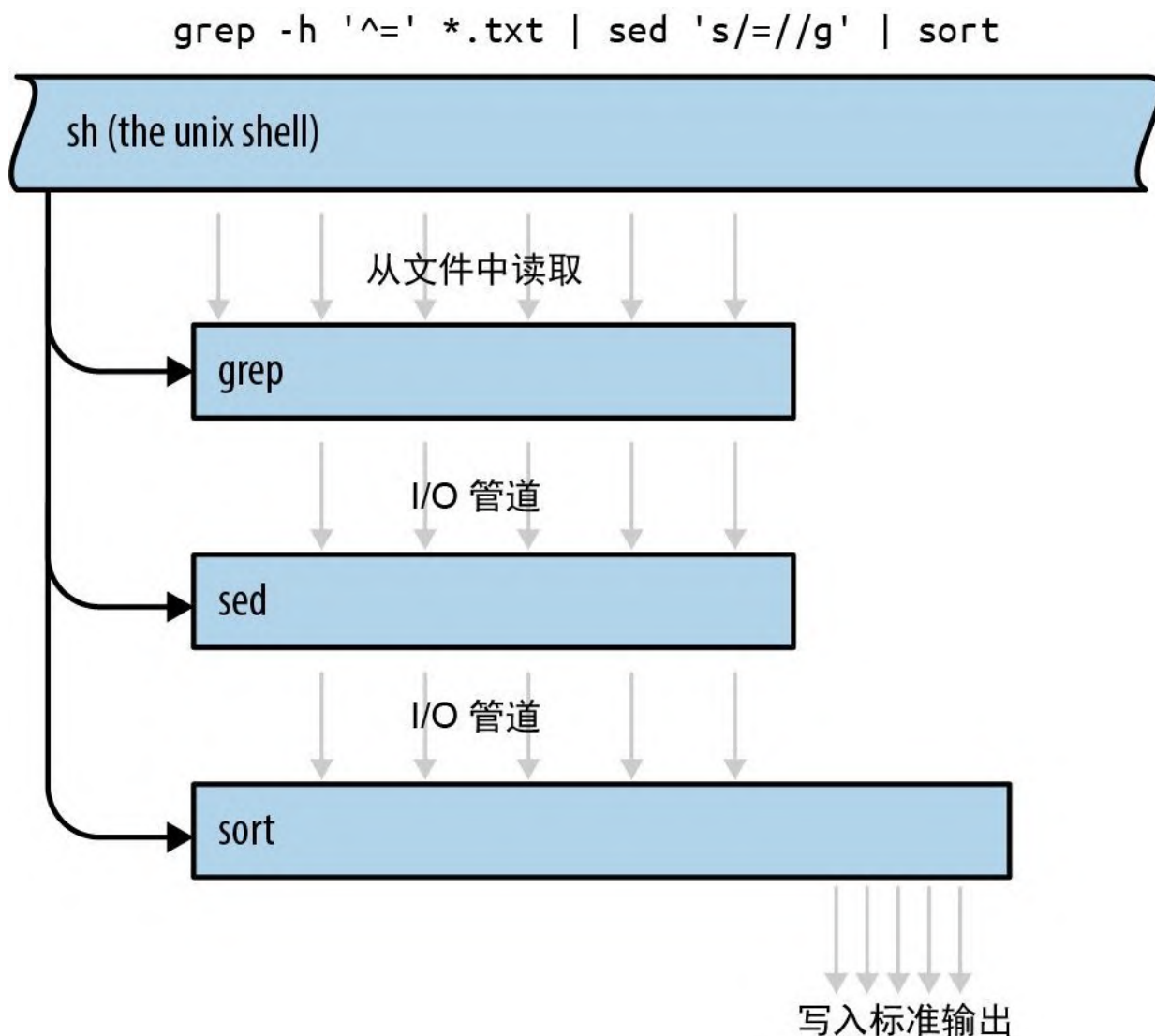


图 19-6: Unix 管道的执行过程

Rust 通道比 Unix 管道更快。发送值只是移动而不是复制，即使要移动的数据结构包含数兆字节数据速度也很快。

19.2.1 发送值

在接下来的几节中，我们将使用通道来构建一个创建倒排索引的并发程序，倒排索引是搜索引擎的关键组成部分之一。每个搜索引擎都会处理特定的文档集合。倒排索引是记录“哪些词出现在哪里”的数据库。

我们将展示与线程和通道有关的部分代码。完整的程序（参见本书在 GitHub 网站上的页面）也不长，大约 1000 行代码。

我们的程序结构是管道式的，如图 19-7 所示。管道只是使用通道的众多方法之一（稍后会讨论其他几种方式），但它们是将并发引入现有单线程程序的最直观方式。

读取

将整个文件读取成字符串

索引

小型内存索引

在内存中合并

大型内存索引

写入

索引文件名

文件合并

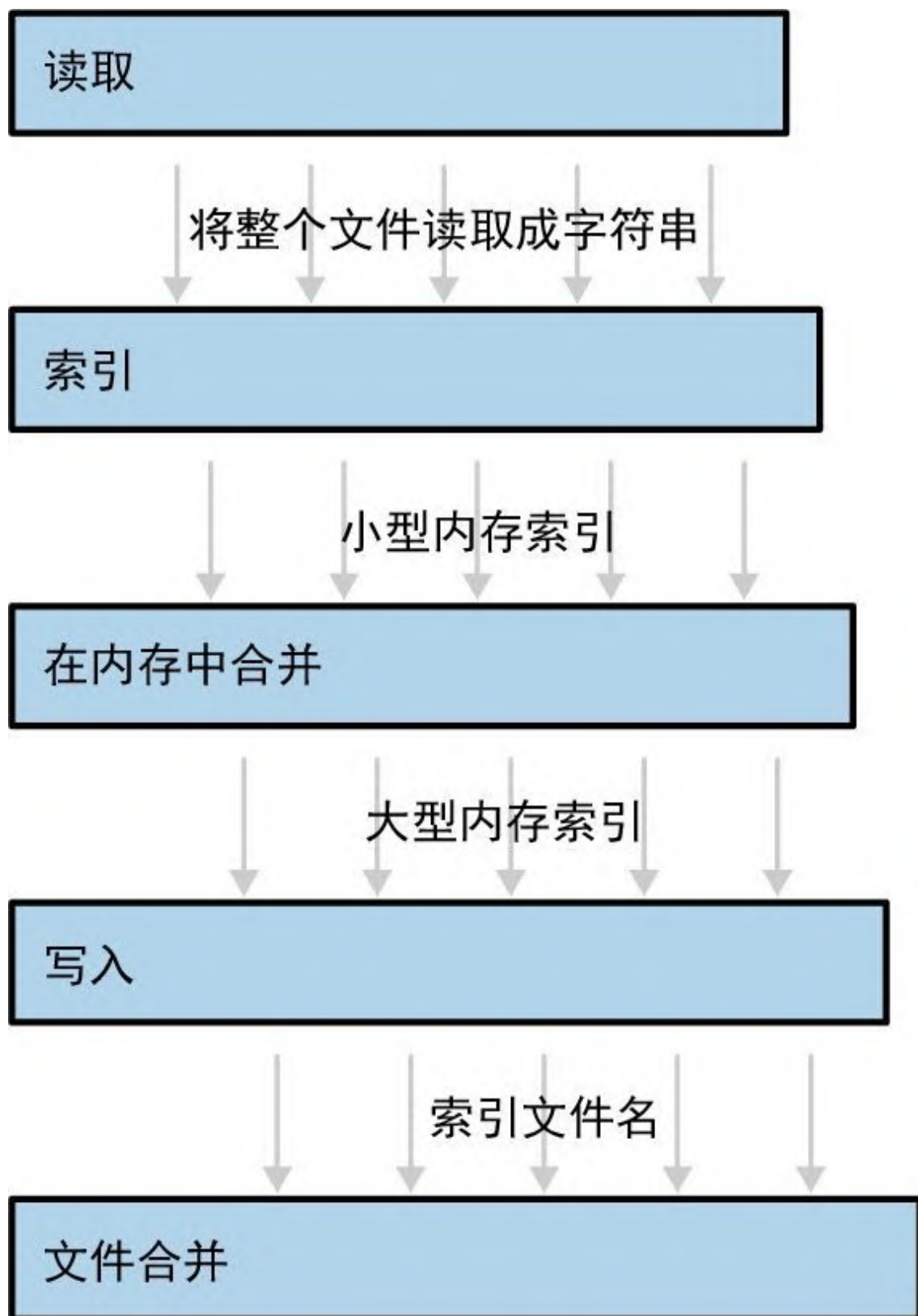


图 19-7：索引构建器管道，其中箭头表示通过通道将值从一个线程发送到另一个线程（未展示磁盘 I/O）

这个程序使用总共 5 个线程分别执行了不同的任务。每个线程在程序的生命周期内不断地生成输出。例如，第一个线程只是将源文档从磁盘逐个读取到内存中。（之所以用一个线程来做这件事，是因为我们想在这里编写尽可能简单的代码，该代码只会调用像 `fs::read_to_string` 这样的阻塞式 API。在磁盘工作时，我们不希望 CPU 闲置。）该阶段会为每个文档输出一个表示其内容的长 `String`，因此这个线程与下一个线程可以通过 `String` 型通道连接。

我们的程序将从启动读取文件的线程开始。假设 `documents` 是一个 `Vec<PathBuf>`，即一个文件名向量。启动读取文件线程的代码如下所示：

```
use std::{fs, thread};
use std::sync::mpsc;

let (sender, receiver) = mpsc::channel();

let handle = thread::spawn(move || {
    for filename in documents {
        let text = fs::read_to_string(filename)?;

        if sender.send(text).is_err() {
            break;
        }
    }
    Ok(())
});
```

通道是 `std::sync::mpsc` 模块的一部分，本章稍后会解释这个名字的含义。下面来看这段代码是如何工作的。先创建一个通道：

```
let (sender, receiver) = mpsc::channel();
```

`channel` 函数会返回一个值对：发送者和接收者。底层队列的数据结构是标准库的内部实现细节。

通道是有类型的。我们要使用这个通道来发送每个文件的文本，因此 `sender` 和 `receiver` 的类型分别为 `Sender<String>` 和 `Receiver<String>`。固然可以写成 `mpsc::channel::<String>()` 来明确请求一个字符串型通道。但最好还是让 Rust 的类型推断来解决这个问题。

```
let handle = thread::spawn(move || {
```

和以前一样，使用 `std::thread::spawn` 来启动一个线程。`sender`（而不是 `receiver`）的所有权会通过这个 `move` 闭包转移给新线程。

接下来的几行代码只会从磁盘读取文件：

```
for filename in documents {  
    let text = fs::read_to_string(filename)?;
```

成功读取文件后，要将其文本发送到通道中：

```
    if sender.send(text).is_err() {  
        break;  
    }  
}
```

`sender.send(text)` 会将 `text` 值移动到通道中。最终，通道会再次把 `text` 值转交给接收到该值的任何对象。无论 `text` 包含 10 行文本还是 10 兆字节，此操作都只会复制 3 个机器字（`String` 结构体的大小），相应的 `receiver.recv()` 调用也只会复制 3 个机器字。

`send` 方法和 `recv` 方法都会返回 `Result`，这两种方法只有当通道的另一端已被丢弃时才会失败。如果 `Receiver` 已被丢弃，那么 `send` 调用就会失败，因为如果不失败，则该值会永远存在于通道中：没有 `Receiver`，任何线程都无法再接收它。同样，如果通道中没有值在等待并且 `Sender` 已被丢弃，则 `recv` 调用会失败，因为如果不失败，`recv` 就只能永远等待：

没有 `Sender`，任何线程都无法再发出下一个值。丢弃通道的某一端是正常的“挂断”方式，完成后就会关闭连接。

在我们的代码中，只有当接收者的线程提前退出时，`sender.send(text)` 才会失败。这是使用通道的典型代码。无论接收者是故意退出还是出错退出，读取者线程都可以悄悄地自行关闭。

无论是发生了这种情况还是线程读取完了所有文档，程序都会返回 `Ok()`：

```
Ok()  
});
```

请注意，这个闭包返回了一个 `Result`。如果线程遇到 I/O 错误，则会立即退出，错误会被存储在线程的 `JoinHandle` 中。

当然，就像其他编程语言一样，Rust 在错误处理方面也有许多其他选择。当发生错误时，可以使用 `println!` 将其打印出来，然后再处理下一个文件。还可以通过用于传递数据的同一通道传递错误，把它变成 `Result` 的通道——或者为传递错误创建第二个通道。我们在这里选择的方法既轻量又可靠：我们使用了 `?` 运算符，这样就不会有一堆样板代码，甚至连 Java 中可能看到的显式 `try/catch` 都没有，而且也不会悄无声息地传递错误。

为便于使用，程序会把所有这些代码都包装在一个函数中，该函数会返回至今尚未用到的 `receiver` 和新线程的 `JoinHandle`：

```
fn start_file_reader_thread(documents: Vec<PathBuf>)  
    -> (mpsc::Receiver<String>, thread::JoinHandle<io::Result<()>>)  
{  
    let (sender, receiver) = mpsc::channel();  
  
    let handle = thread::spawn(move || {  
        ...  
    });  
  
    (receiver, handle)  
}
```

请注意，这个函数会启动新线程并立即返回。我们会为管道的每个阶段编写一个类似的函数。

19.2.2 接收值

现在我们有了一个线程来运行发送值的循环。接下来可以启动第二个线程来运行调用 `receiver.recv()` 的循环：

```
while let Ok(text) = receiver.recv() {
    do_something_with(text);
}
```

但 `Receiver` 是可迭代的，所以还有更好的写法：

```
for text in receiver {
    do_something_with(text);
}
```

这两个循环是等效的。无论怎么写，当控制流到达循环顶部时，只要通道恰好为空，接收线程在其他线程发送值之前都会阻塞。当通道为空且 `Sender` 已被丢弃时，循环将正常退出。在我们的程序中，当读取者线程退出时，循环会自然而然地退出。该线程正在运行一个拥有变量 `sender` 的闭包，当闭包退出时，`sender` 会被丢弃。

现在可以为管道的第二阶段编写代码了：

```
fn start_file_indexing_thread(texts: mpsc::Receiver<String>)
    -> (mpsc::Receiver<InMemoryIndex>, thread::JoinHandle<()>)
{
    let (sender, receiver) = mpsc::channel();

    let handle = thread::spawn(move || {
        for (doc_id, text) in texts.into_iter().enumerate() {
            let index = InMemoryIndex::from_single_document(doc_id,
text);
            if sender.send(index).is_err() {
                break;
            }
        }
    });
}
```

```
(receiver, handle)
}
```

这个函数会启动一个线程，该线程会从一个通道（`texts`）接收 `String` 值并将 `InMemoryIndex` 值发送给另一个通道（`sender/receiver`）。这个线程的工作是获取第一阶段加载的每个文件，并将每个文档变成一个小单文件内存倒排索引。

这个线程的主循环很简单。索引文档的所有工作都是由函数 `InMemoryIndex::from_single_document` 完成的。我们不会在这里展示它的源代码，你只要知道它会在单词边界处拆分输入字符串，然后生成从单词到位置列表的映射就可以了。

这个阶段不会执行 I/O，所以不必处理各种 `io::Error`。它会返回 `()` 而非 `io::Result<()>`。

19.2.3 运行管道

其余 3 个阶段的设计也是类似的。每个阶段都会使用上一阶段创建的 `Receiver`。对管道的其余部分，我们设定的目标是将所有小索引合并到磁盘上的单个大索引文件中。最快的方法是将这个任务分为 3 个阶段。我们不会在这里展示代码，只会展示这 3 个函数的类型签名。完整的源代码请参见在线文档。

首先，合并内存中的索引，直到它们变得“笨重”（第三阶段）：

```
fn start_in_memory_merge_thread(file_indexes:
mpsc::Receiver<InMemoryIndex>)
    -> (mpsc::Receiver<InMemoryIndex>, thread::JoinHandle<()>)
```

然后，将这些大型索引写入磁盘（第四阶段）：

```
fn start_index_writer_thread(big_indexes:
mpsc::Receiver<InMemoryIndex>,
                                output_dir: &Path)
    -> (mpsc::Receiver<PathBuf>,
thread::JoinHandle<io::Result<()>>)
```


最后，如果有多个大文件，就用基于文件的合并算法合并它们（第五阶段）：

```
fn merge_index_files(files: mpsc::Receiver<PathBuf>, output_dir:
&Path)
    -> io::Result<()>
```

最后一个阶段不会返回 `Receiver`，因为它是此管道的末尾。这个阶段会在磁盘上生成单个输出文件。它也不会返回 `JoinHandle`，因为我们没有为这个阶段启动线程。这项工作是在调用者的线程上完成的。

现在来看一下启动线程和检查错误的代码：

```
fn run_pipeline(documents: Vec<PathBuf>, output_dir: PathBuf)
    -> io::Result<()>
{
    // 启动管道的所有5个阶段
    let (texts, h1) = start_file_reader_thread(documents);
    let (pints, h2) = start_file_indexing_thread(texts);
    let (gallons, h3) = start_in_memory_merge_thread(pints);
    let (files, h4) = start_index_writer_thread(gallons,
&output_dir);
    let result = merge_index_files(files, &output_dir);

    // 等待这些线程结束，保留它们遇到的任何错误
    let r1 = h1.join().unwrap();
    h2.join().unwrap();
    h3.join().unwrap();
    let r4 = h4.join().unwrap();

    // 返回遇到的第一个错误（如果有的话）（如你所见，h2和h3
    // 不会失败，因为这些线程都是纯粹的内存数据处理）
    r1?;
    r4?;
    result
}
```

和以前一样，使用 `.join().unwrap()` 显式地将 `panic` 从子线程传播到主线程。这里唯一不寻常的事情是：我们没有马上使用 `?`，而是将 `io::Result` 值放在一边，直到所有 4 个线程都联结完成。

这个管道比等效的单线程管道快 40%。这一下午的工作还算小有所成，但与曼德博程序曾获得的 675% 的提升相比就有点儿微不足道了。我们显然没有让系统的 I/O 容量或所有 CPU 核心的工作量饱和。这是怎么回事？

管道就像制造业工厂中的装配流水线，其性能受限于最慢阶段的吞吐量。一条全新的、未调整过的装配线可能和单元化生产一样慢，只有对装配流水线做针对性的调整才能获得回报。在这个例子中，测量表明第二阶段是瓶颈。我们的索引线程使用了 `.to_lowercase()` 和 `.is_alphanumeric()`，因此它会花费大量时间在 Unicode 表中查找。对于索引下游的其他阶段，它们大部分时间在 `Receiver::recv` 中休眠，等待输入。

这意味着应该还可以更快。只要解决了这些瓶颈，并行度就会提高。既然你已经知道如何使用通道，再加上我们的程序是由孤立的代码片段组成的，那么就很容易找到解决第一个瓶颈的方法。可以手动优化第二阶段的代码，就像优化其他代码一样，将工作拆分成两个或更多阶段，或同时运行多个文件索引线程。

19.2.4 通道的特性与性能

`std::sync::mpsc` 中的 `mpsc` 代表多生产者、单消费者（multi-producer, single-consumer），这是对 Rust 通道提供的通信类型的简洁描述。

这个示例程序中的通道会将值从单个发送者传送到单个接收者。这是相当普遍的案例。但是 Rust 通道也支持多个发送者，如果需要的话，你可以用一个线程来处理来自多个客户端线程的请求，如图 19-8 所示。

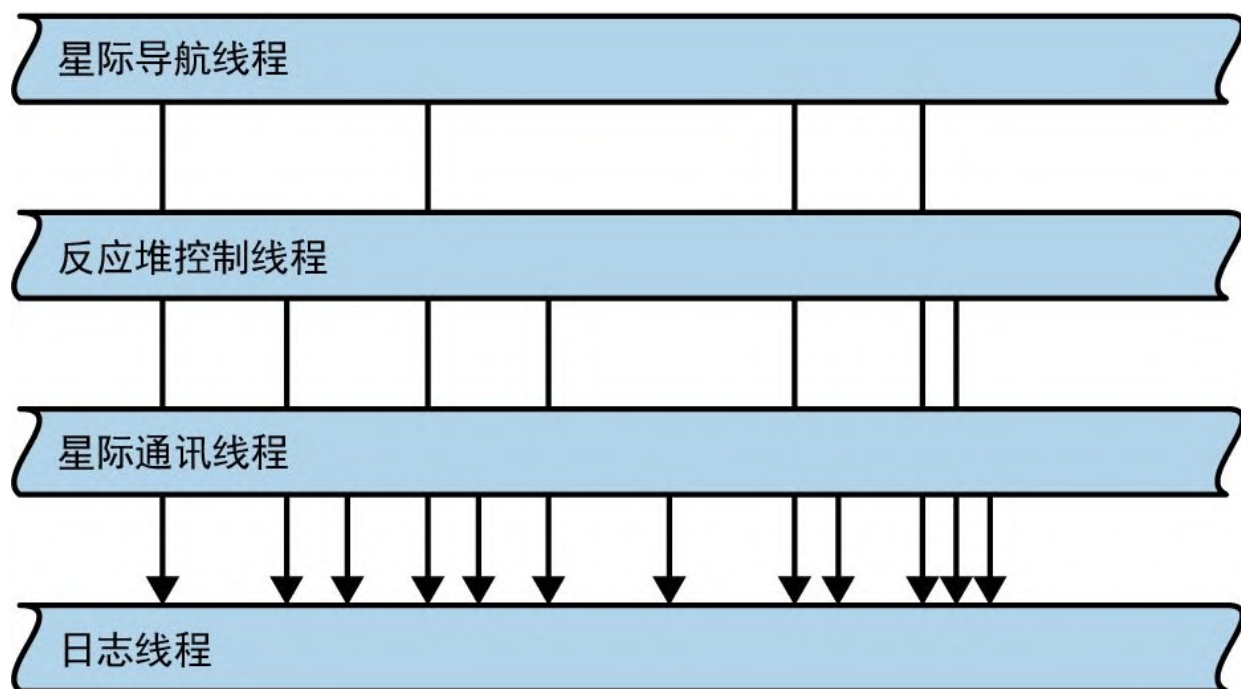


图 19-8：单个通道接收来自多个发送者的请求

`Sender<T>` 实现了 `Clone` 特型。要获得具有多个发送者的通道，只需创建一个常规通道并根据需要多次克隆发送者即可。可以将每个 `Sender` 值转移给不同的线程。

`Receiver<T>` 不能被克隆，所以如果需要让多个线程从同一个通道接收值，就需要使用 `Mutex`。本章后面会展示如何做。

Rust 的通道经过了精心优化。首次创建通道时，Rust 会使用特殊的“一次性”队列实现。如果只通过此通道发送一个对象，那么开销是最低的。如果要发送第二个值，Rust 就会切换到第二种队列实现。实际上，第二种实现就是为长期使用而设计的，它会准备好传输许多值的通道，同时最大限度地降低内存分配开销。如果你克隆了 `Sender`，那么 Rust 就必须回退到第三种实现，使得多个线程可以安全地同时尝试发送值，这种实现是安全的。当然，即便这 3 种实现中最慢的一种也是无锁队列，所以发送或接收一个值最多就是执行几个原子化操作和堆分配，再加上移动本身。只有当队列为空时才需要系统调用，这时候接收线程就会让自己进入休眠状态。当然，在这种情况下，走这个通道的流量无论如何都不会满载。

尽管进行了所有这些优化工作，但应用程序很容易在通道性能方面犯一个错误：发送值的速度快于接收值和处理值的速度。这会导致通道中积压的值不断增长。例如，在这个程序中，我们发现文件读取线程（第一阶段）加载文件的速度比文件索引线程（第二阶段）更快。结果导致数百兆字节的原始数据从磁盘中读取出来后立即填充到了队列中。

这种不当行为会消耗内存并破坏局部性。更糟糕的是，发送线程还会继续运行，耗尽 CPU 和其他系统资源只是为了发出更多的值，而此时却恰恰是接收端最需要资源来处理它们的时候。这显然不对劲。

Rust 再次从 Unix 管道中汲取了灵感。Unix 使用了一个优雅的技巧来提供一些背压，以迫使超速的发送者放慢速度：Unix 系统上的每个管道都有固定的大小，如果进程试图写入暂时已满的管道，那么系统就会简单地阻塞该进程直到管道中有了空间。这在 Rust 中的等效设计称为同步通道：

```
use std::sync::mpsc;

let (sender, receiver) = mpsc::sync_channel(1000);
```

同步通道与常规通道非常像，但在创建时可以指定它能容纳多少个值。对于同步通道，`sender.send(value)` 可能是一个阻塞操作。毕竟，有时候阻塞也不是坏事。在我们的示例程序中，将 `start_file_reader_thread` 中的 `channel` 更改为具有 32 个值空间的 `sync_channel` 后，可将基准数据集上的内存使用量节省 2/3，却不会降低吞吐量。

19.2.5 线程安全：Send 与 Sync

迄今为止，我们一直假定所有值都可以在线程之间自由移动和共享。这基本正确，但 Rust 完整的线程安全故事取决于两个内置特型，即 `std::marker::Send` 和 `std::marker::Sync`。

- 实现了 `Send` 的类型可以安全地按值传给另一个线程。它们可以跨线程移动。

- 实现了 `Sync` 的类型可以安全地将一个值的不可变引用传给另一个线程。它们可以跨线程共享。

这里所说的安全，就是我们一直在强调的意思：没有数据竞争和其他未定义行为。

例如，在本章开头的 `process_files_in_parallel` 示例中，我们使用闭包将 `Vec<String>` 从父线程传给了每个子线程。虽然我们当时没有指出，但这意味着向量及其字符串会在父线程中分配，但会在子线程中释放。`Vec<String>` 实现了 `Send`，这事实上代表一个关于“可以怎么做”的 API 承诺：`Vec` 和 `String` 在内部使用的分配器是线程安全的。

（如果要用快速但非线程安全的分配器编写自己的 `Vec` 类型和 `String` 类型，就不得不使用非 `Send` 的类型（如不安全的指针）来实现它们。然后 Rust 就会推断出 `NonThreadSafeVec` 类型和 `NonThreadSafeString` 类型没有实现 `Send` 而将它们限制为在单线程中使用。但需要这么做的情况非常罕见。）

如图 19-9 所示，大多数类型既实现了 `Send` 也实现了 `Sync`。你甚至不必使用 `#[derive]` 来为程序中的结构体和枚举实现这些特型。Rust 会自动帮你实现。如果结构体或枚举的所有字段都是 `Send` 的，那它自然是 `Send` 的；如果结构体或枚举的所有字段都是 `Sync` 的，那它自然是 `Sync` 的。

有些类型是 `Send` 的但不是 `Sync` 的。这通常是刻意设计的，就像 `mpsc::Receiver` 一样，它是为了保证 `mpsc` 通道的接收端一次只能被一个线程使用。

少数既不是 `Send` 也不是 `Sync` 的类型大多使用了非线程安全的可变性，比如引用计数智能指针类型 `std::rc::Rc<T>`。

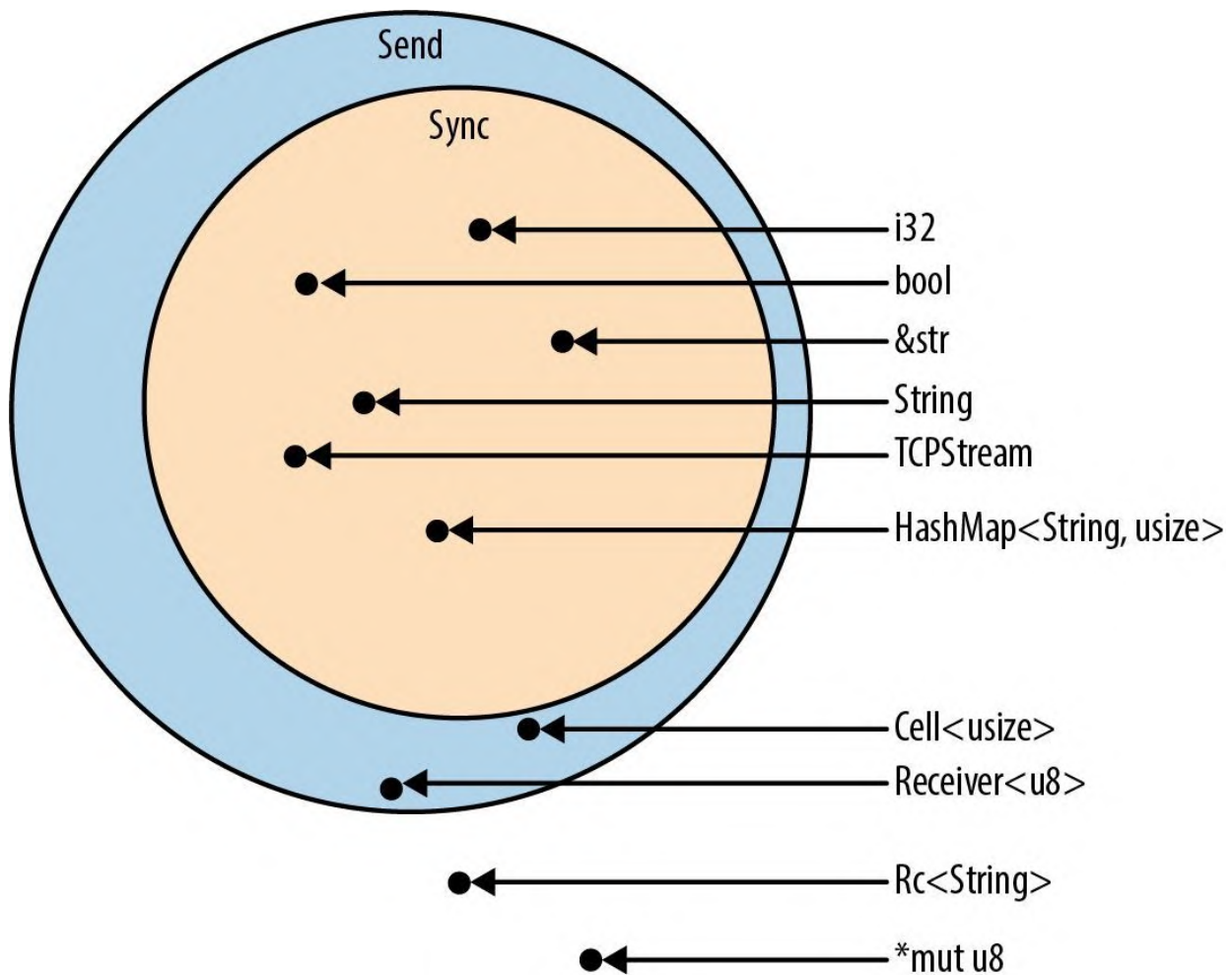


图 19-9: **Send** 类型和 **Sync** 类型

如果 `Rc<String>` 是 `Sync` 的，那么允许线程通过共享引用共享单个 `Rc` 会发生什么呢？如图 19-10 所示，如果两个线程碰巧同时尝试克隆 `Rc`，就会发生数据竞争，因为两个线程都会增加共享引用计数。结果引用计数可能变得不准确，导致释放后仍在使用（`use-after-free`）或稍后出现双重释放，这都是未定义行为。

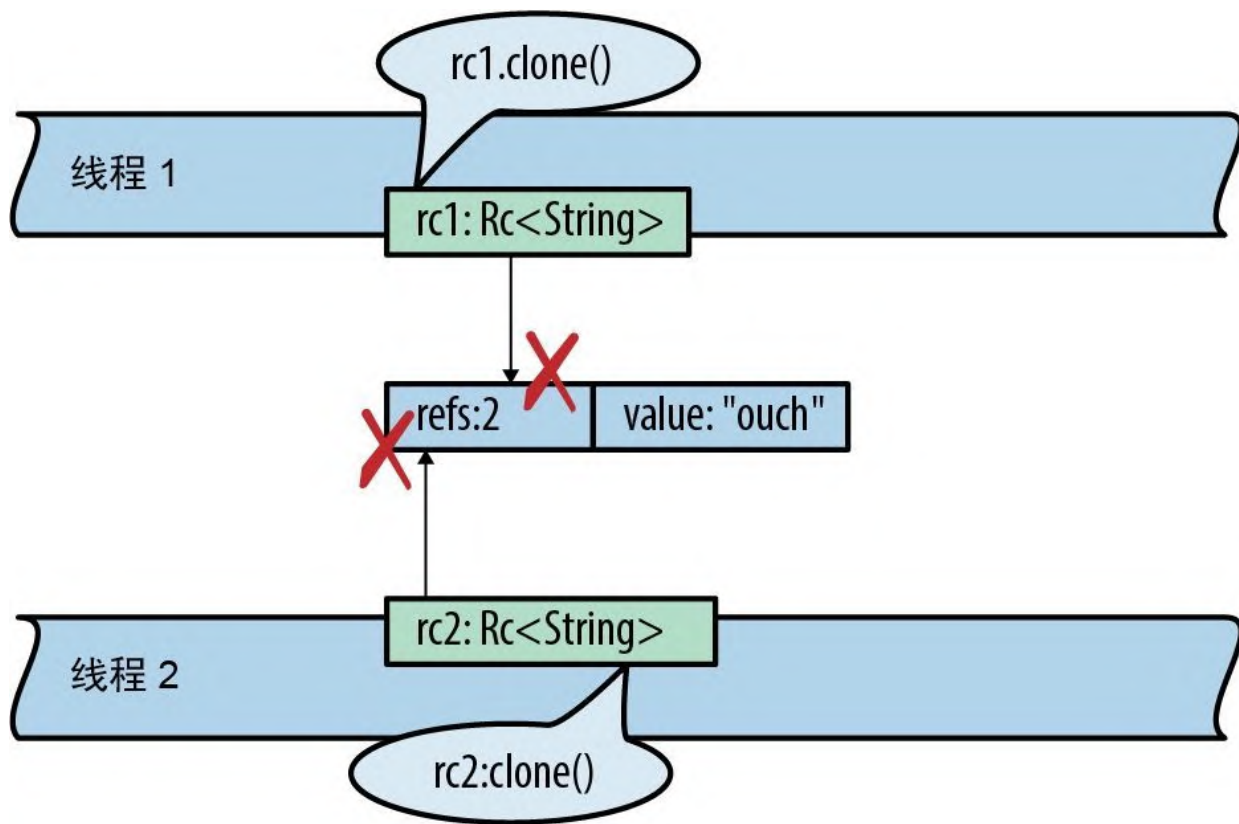


图 19-10：为什么 `Rc<String>` 既非 `Sync` 型也非 `Send` 型

当然，Rust 会阻止这种情况。下面是试图建立这种数据竞争的代码：

```
use std::thread;
use std::rc::Rc;

fn main() {
    let rc1 = Rc::new("ouch".to_string());
    let rc2 = rc1.clone();
    thread::spawn(move || { // 错误
        rc2.clone();
    });
    rc1.clone();
}
```

Rust 会拒绝编译这段代码，并给出详细的错误消息：

```
error: `Rc<String>` cannot be sent between threads safely
|
10 |     thread::spawn(move || { // 错误
```

```

|               ^^^^^ `Rc<String>` cannot be sent between threads
safely
|
|   = help: the trait `std::marker::Send` is not implemented for
`Rc<String>`
|   = note: required because it appears within the type
`[closure@...]'
|   = note: required by `std::thread::spawn`

```

现在可以看出 `Send` 和 `Sync` 如何帮助 Rust 加强线程安全了。对于跨线程边界传输数据的函数，`Send` 和 `Sync` 会作为函数类型签名中的限界。当你生成（`spawn`）一个线程时，传入的闭包必须实现了 `Send` 特型，这意味着它包含的所有值都必须是 `Send` 的。同样，如果要通过通道将值发送到另一个线程，则该值必须是 `Send` 的。

19.2.6 绝大多数迭代器能通过管道传给通道

我们的倒排索引构建器是作为管道构建的。虽然代码很清晰，但需要手动建立通道和启动线程。相比之下，我们在第 15 章中构建的迭代器流水线似乎将更多的工作打包到了几行代码中。可以为线程管道构建类似的东西吗？

如果能统一迭代器流水线和线程管道就好了。这样索引构建器就可以写成迭代器流水线了。它可能是这样开始的：

```

documents.into_iter()
    .map(read_whole_file)
    .errors_to(error_sender)    // 过滤出错误结果
    .off_thread()              // 为上面的工作生成线程
    .map(make_single_file_index)
    .off_thread()              // 为第二阶段生成另一个线程
    ...

```

特型允许我们向标准库类型添加一些方法，所以确实可以这样做。

首先，编写一个特型来声明自己想要的方法：

```

use std::sync::mpsc;

pub trait OffThreadExt: Iterator {
    // ...
}

```



```
/// 将这个迭代器转换为线程外迭代器：`next()`调用发生在
/// 单独的工作线程上，因此该迭代器和循环体会同时运行
fn off_thread(self) -> mpsc::IntoIter<Self::Item>;
}
```

然后，为迭代器类型实现这个特型。mpsc::Receiver 已经是可迭代类型了，对于我们的实现很有帮助：

```
use std::thread;

impl<T> OffThreadExt for T
    where T: Iterator + Send + 'static,
           T::Item: Send + 'static
{
    fn off_thread(self) -> mpsc::IntoIter<Self::Item> {
        // 创建一个通道把条目从工作线程中传出去
        let (sender, receiver) = mpsc::sync_channel(1024);

        // 把这个迭代器转移给新的工作线程，并在那里运行它
        thread::spawn(move || {
            for item in self {
                if sender.send(item).is_err() {
                    break;
                }
            }
        });

        // 返回一个从通道中拉取值的迭代器
        receiver.into_iter()
    }
}
```

此代码中的 where 子句是通过类似于 11.5 节描述的一个流程确定的。起初，我们只有如下内容：

```
impl<T> OffThreadExt for T
```

也就是说，我们希望此实现适用于所有迭代器。而 Rust 说不行。因为要用 spawn 将 T 类型的迭代器转移给新线程，所以必须指定 T: Iterator + Send + 'static。因为要用通道发回条目，所以必须指定 T::Item: Send + 'static。做完这些改动，Rust 很满意。

简而言之，这就是 Rust 的特征：我们可以自由地为该语言中的几乎每个迭代器添加一个提供并发能力的工具，但前提是要理解并用代码说明它在安全使用方面的限制条件。

19.2.7 除管道之外的用法

本节会以管道作为示例，因为管道是使用通道的一种很好、很直白的方式。每个人都能理解它们。管道是具体、实用且具有确定性的。不过，通道不仅仅在管道中 useful，它们也是向同一进程中的其他线程提供异步服务的快捷且简便的方法。

假设你想在自己的线程上进行日志记录，如图 19-8 所示。其他线程可以通过通道将日志消息发送到日志线程。由于你可以克隆通道的 `Sender`，因此许多客户端线程可以具有向同一个日志记录线程发送日志消息的发送器。

在独立线程上运行诸如记录日志之类的服务有一些优势。日志记录线程可以在需要时轮换日志文件。它不必与其他线程进行任何花哨的协调。这些线程也不会被阻塞。消息可以在通道中无害地累积片刻，直到日志线程恢复工作。

通道也可用于一个线程向另一个线程发送请求并要求返回某种响应的情况。第一个线程的请求可以是一个结构体或元组，包含一个 `Sender`，这个 `Sender` 是第二个线程用来发送其回复的一种回邮信封。但这并不意味着这种交互必须是同步的。第一个线程可以自行决定是阻塞并等待响应，还是使用 `.try_recv()` 方法轮询结果。

迄今为止，我们介绍了用于高度并行计算的分叉与合并和用于松散连接组件的通道，这两种工具已经足以应对大部分应用程序。但本章内容还未结束，请接着往下看。

19.3 共享可变状态

自从你在第 8 章发布了 `fern_sim crate`，在之后的几个月里，你的蕨类植物模拟软件真的“火”了。现在你正在创建一个多人即时战略游戏，其中 8 名玩家在模拟的侏罗纪景观中竞相种植大部分真实的同时代蕨类植物。该游戏的服务器是一个大规模并行应用程序，要处理从很多线程涌入的大量请求。一旦有了 8 名待加入的玩家，这些线程该如何相互协调以开始游戏呢？

这里要解决的问题是，很多线程需要访问待加入游戏玩家的共享列表。这个数据必然是可变的，并且会在所有线程之间共享。如果 Rust 不支持共享的可变状态，那我们还能用它做些什么呢？

可以通过创建一个新线程来解决这个问题，该线程的全部工作就是管理这个列表。其他线程将通过通道与它通信。当然，这会多占用一个线程，从而产生一些操作系统开销。

另一种选择是使用 Rust 提供的工具来安全地共享可变数据。这种工具确实存在。它们就是任何使用过线程的系统程序员都熟悉的底层原语。本节将介绍互斥锁、读 / 写锁、条件变量和原子化整数。最后，我们将展示如何在 Rust 中实现全局可变变量。

19.3.1 什么是互斥锁

互斥锁（mutex）或**锁**（lock）用于强制多个线程在访问某些数据时轮流读写。19.3.2 节会介绍 Rust 的互斥锁。先回顾一下互斥锁在其他语言中的用法是有好处的。下面是互斥锁在 C++ 中的简单用法：

```
// C++代码，不是Rust代码
void FernEmpireApp::JoinWaitingList(PlayerId player) {
    mutex.Acquire();

    waitingList.push_back(player);

    // 如果有了足够的待进入玩家，就开始游戏
    if (waitingList.size() >= GAME_SIZE) {
        vector<PlayerId> players;
```

```
        waitingList.swap(players);  
        StartGame(players);  
    }  
  
    mutex.Release();  
}
```

调用 `mutex.Acquire()` 和 `mutex.Release()` 会标记出此代码中**临界区**的开始和结束。对于程序中的每个 `mutex`，一次只能有一个线程在临界区内运行。如果临界区中有一个线程，那么所有调用 `mutex.Acquire()` 的其他线程都将被阻塞，直到第一个线程到达 `mutex.Release()`。

我们说互斥锁能**保护数据**，在这个例子中就是 `mutex` 会保护 `waitingList`。不过，程序员有责任确保每个线程总是在访问数据之前获取互斥锁，并在之后释放它。

互斥锁很有用，原因如下。

- 它们可以防止**数据竞争**，即多个竞争线程同时读取和写入同一内存的情况。数据竞争是 C++ 和 Go 中的未定义行为。Java、C# 等托管语言承诺不会崩溃，但发生数据竞争时，产出的结果仍然没有意义。
- 即使不存在数据竞争，并且所有读取和写入在程序中都是按顺序一个接一个地发生，如果没有互斥锁，不同线程的操作也可能会以任意方式相互交错。想象一下，如何写出即使在运行期被其他线程修改了数据也能照常工作的代码。再想象一下你试图调试这个程序。那简直就像程序在“闹鬼”。
- 互斥锁支持使用**不变条件**进行编程，在初始化设置时，那些关于受保护数据的规则在刚构造出来时就是成立的，并且会让每个临界区负责维护这些规则。

当然，所有这些实际上都基于同一个原因：不受控的竞态条件会让编程变得非常棘手。互斥锁给混乱带来了一些秩序，尽管不如通道或分叉与合并那么有序。

然而，在大多数语言中，互斥锁很容易搞砸。例如，在 C++ 中，数据和锁是彼此独立的对象。理想情况下，可以通过注释来解释每个线程

必须在接触数据之前获取互斥锁：

```
class FernEmpireApp {
    ...

private:
    // 等待加入游戏的玩家列表。通过`mutex`来保护
    vector<PlayerId> waitingList;

    // 请在读写`waitingList`之前获取互斥锁
    Mutex mutex;
    ...
};
```

但即使有这么好的注释，编译器也无法在此处强制执行安全访问。当一段代码忘了获取互斥锁时，就会得到未定义行为。现实中，这意味着极难重现和修复的 bug。

虽然在 Java 中对象和互斥锁之间存在某种概念上的关联，但这种关联也不是很紧密。编译器不会试图强制执行这种关联，实际上，受锁保护的数据在大多数时候不仅仅是相关对象中的几个字段，而是经常包含分布于多个对象中的数据。“锁定”方案依旧很棘手。注释仍然是执行这种关联的主要工具。

19.3.2 **Mutex<T>**

现在我们将展示在 Rust 中如何实现等待列表。在我们的蕨类帝国游戏服务器中，每个玩家都有一个唯一的 ID：

```
type PlayerId = u32;
```

等待列表只是玩家的集合：

```
const GAME_SIZE: usize = 8;

/// 等候列表永远不会超过GAME_SIZE个玩家
type WaitingList = Vec<PlayerId>;
```

等待列表会被存储为 FernEmpireApp 中的一个字段，这是在服务器启动期间在 Arc 中设置的一个单例。每个线程都有一个 Arc 指向它。它包含我们程序中所需的全部共享配置和其他“零件”，其中大部分是只读的。由于等待列表既是共享的又是可变的，因此必须由 Mutex 提供保护：

```
use std::sync::Mutex;

/// 所有线程都可以共享对这个大型上下文结构体的访问
struct FernEmpireApp {
    ...
    waiting_list: Mutex<WaitingList>,
    ...
}
```

与 C++ 不同，在 Rust 中，受保护的数据存储于 Mutex 内部。建立此 Mutex 的代码如下所示：

```
use std::sync::Arc;

let app = Arc::new(FernEmpireApp {
    ...
    waiting_list: Mutex::new(vec![]),
    ...
});
```

创建新的 Mutex 看起来就像创建新的 Box 或 Arc，但是 Box 和 Arc 意味着堆分配，而 Mutex 仅与锁操作有关。如果希望在堆中分配 Mutex，则必须明确写出来，就像这里所做的这样：对整个应用程序使用 Arc::new，而仅对受保护的数据使用 Mutex::new。这两个类型经常一起使用，Arc 用于跨线程共享数据，而 Mutex 用于跨线程共享的可变数据。

现在可以实现使用互斥锁的 join_waiting_list 方法了：

```
impl FernEmpireApp {
    /// 往下一个游戏的等候列表中添加一个玩家。如果有足够
    /// 的待进入玩家，则立即启动一个新游戏
    fn join_waiting_list(&self, player: PlayerId) {
        // 锁定互斥锁，并授予内部数据的访问权。`guard`的作用域是一个临界区
    }
}
```

```
        let mut guard = self.waiting_list.lock().unwrap();

        // 现在开始执行游戏逻辑
        guard.push(player);
        if guard.len() == GAME_SIZE {
            let players = guard.split_off(0);
            self.start_game(players);
        }
    }
}
```

获取数据的唯一方法就是调用 `.lock()` 方法：

```
let mut guard = self.waiting_list.lock().unwrap();
```

`self.waiting_list.lock()` 会阻塞，直到获得互斥锁。这个方法调用所返回的 `MutexGuard<WaitingList>` 值是 `&mut WaitingList` 的浅层包装。多亏了 13.5 节讨论过的“隐式解引用”机制，我们可以直接在此守卫上调用 `WaitingList` 的各种方法：

```
guard.push(player);
```

此守卫甚至允许我们借用对底层数据的直接引用。Rust 的生命周期体系会确保这些引用的生命周期不会超出守卫本身。如果不持有锁，就无法访问 `Mutex` 中的数据。

当 `guard` 被丢弃时，锁就被释放了。这通常会发生在块的末尾，但也可以手动丢弃。

```
if guard.len() == GAME_SIZE {
    let players = guard.split_off(0);
    drop(guard); // 启动游戏时就不必锁定列表了
    self.start_game(players);
}
```

19.3.3 `mut` 与互斥锁

`join_waiting_list` 方法并没有通过可变引用获取 `self`，这可能看起来很奇怪，至少初看上去是这样。它的类型签名如下所示：

```
fn join_waiting_list(&self, player: PlayerId)
```

当你调用底层集合 `Vec<PlayerId>` 的 `push` 方法时，它确实需要一个可变引用，其类型签名如下所示：

```
pub fn push(&mut self, item: T)
```

然而这段代码不仅能编译而且运行良好。这是怎么回事？

在 Rust 中，`&mut` 表示**独占访问**。普通 `&` 表示**共享访问**。

我们习惯于把 `&mut` 访问从父级传到子级，从容器传到内容。只有一开始你就拥有对 `starships` 的 `&mut` 引用 [或者你也可能拥有这些 `starships`（星舰）？如果是这样，那么……恭喜你成了埃隆·马斯克。]，才可以在 `starships[id].engine` 上调用 `&mut self` 方法。这是默认设置，因为如果没有对父项的独占访问权，那么 Rust 通常无法确保你对子项拥有独占访问权。

但是 `Mutex` 有办法确保这一点：锁。事实上，互斥锁只不过是提供对内部数据的**独占**（`mut`）访问的一种方法，即使有许多线程也在**共享**（非 `mut`）访问 `Mutex` 本身时，也能确保一切正常。

Rust 的类型系统会告诉我们 `Mutex` 在做什么。它在动态地强制执行独占访问，而这通常是由 Rust 编译器在编译期间静态完成的。

（你可能还记得 `std::cell::RefCell` 也是这么做的，但它没有试图支持多线程。`Mutex` 和 `RefCell` 是内部可变性的两种形式，详情请参见 9.11 节。）

19.3.4 为什么互斥锁不是“银弹”

在开始使用互斥锁之前，我们就介绍了一些并发方式，如果你是 C++ 用户，那么这些方法可能看起来非常容易正确使用。这并非巧合，因

为这些方法本来就是为了给并发编程中最令人困惑的方面提供强有力的保证。专门使用分叉与合并并行的程序具有确定性，不会死锁。使用通道的程序几乎同样表现良好。那些专门供管道使用的通道（比如我们的索引构建器）也具有确定性：虽然消息传递的时机可能有所不同，但不会影响输出。这些关于多线程编程的保证都很好。

Rust 的 `Mutex` 设计几乎肯定会让你比以往任何时候都更系统、更明智地使用互斥锁。但也值得停下来思考一下 Rust 的安全保证可以帮你做什么，不能帮你做什么。

安全的 Rust 代码不会引发数据竞争，这是一种特定类型的 bug，其中多个线程会同时读写同一内存，并产生无意义的结果。这很好，因为数据竞争总会出 bug，这在真正的多线程程序中并不罕见。

但是，使用互斥锁的线程会遇到 Rust 无法为你修复的另一些问题。

- 有效的 Rust 程序不会有数据竞争，但仍然可能有其他竞态条件——程序的行为取决于各线程之间的运行时间长短，因此可能每次运行时都不一样。有些竞态条件是良性的，有些则表现为普遍的不稳定性和难以修复的 bug。以非结构化方式使用互斥锁会引发竞态条件。你需要确保竞态条件是良性的。
- 共享可变状态也会影响程序设计。通道作为代码中的抽象边界，可以轻松拆出彼此隔离的组件以进行测试，而互斥锁则会鼓励一种“只要再添加一个方法就行了”的工作方式，这可能会导致彼此有联系的代码耦合成一个单体。
- 最后，互斥锁也并不像最初看起来那么简单，接下来的 19.3.5 节和 19.3.6 节会详解介绍。

所有这些问题都是工具本身所固有的。要尽可能使用更结构化的方法，只在必要时使用 `Mutex`。

19.3.5 死锁

线程在尝试获取自己正持有的锁时会让自己陷入死锁：

```
let mut guard1 = self.waiting_list.lock().unwrap();  
let mut guard2 = self.waiting_list.lock().unwrap(); // 死锁
```

假设第一次调用 `self.waiting_list.lock()` 成功，获得了锁。第二次调用时看到锁已被持有，所以线程就会阻塞自己，等待锁被释放。它会永远等下去，因为这个正等待的线程就是持有锁的线程。

换言之，`Mutex` 中的锁并不是递归锁。

这里的 bug 是显而易见的。但在实际程序中，这两个 `lock()` 调用可能位于两个不同的方法中，其中一个会调用另一个。单独来看，每个方法的代码看起来都没什么问题。还有其他方式可以导致死锁，比如涉及多个线程或每个线程同时获取多个互斥锁。`Rust` 的借用系统不能保护你免于死锁。最好的保护是保持临界区尽可能小：进入，开始工作，完成后马上离开。

通道也有可能陷入死锁。例如，两个线程可能会互相阻塞，每个线程都在等待从另一个线程接收消息。然而，再次强调，良好的程序设计可以让你确信这在实践中不会发生。在管道中，就像我们的倒排索引构建器一样，数据流是非循环的。与 `Unix shell` 管道一样，这样的程序不可能发生死锁。

19.3.6 “中毒”的互斥锁

`Mutex::lock()` 返回 `Result` 的原因与 `JoinHandle::join()` 是一样的：如果另一个线程发生 panic，则可以优雅地失败。当我们编写 `handle.join().unwrap()` 时，就是在告诉 `Rust` 将 panic 从一个线程传播到另一个线程。`mutex.lock().unwrap()` 惯用法同样如此。

如果线程在持有 `Mutex` 期间出现 panic，则 `Rust` 会把 `Mutex` 标记为已“中毒”。之后每当试图锁住已“中毒”的 `Mutex` 时都会得到错误结果。如果发生这种情况，我们的 `.unwrap()` 调用就会告诉 `Rust` 发生了 panic，将 panic 从另一个线程传播到本线程。

“中毒”的互斥锁有多糟糕？中毒听起来很致命，但在这个场景中并不一定致命。正如我们在第 7 章中所说，panic 是安全的。一个发生了 panic 的线程能让程序的其余部分仍然留在安全状态。

这样看来，互斥锁因 panic 而“中毒”的原因并非害怕出现未定义行为。相反，它真正的关注点在于你编程时一直在维护不变条件。由于你的程序在未完成其正在执行的操作的情况下发生 panic 并退出临界区，可能更新了受保护数据的某些字段但未更新其他字段，因此不变条件现在有可能已经被破坏了。于是 Rust 决定让这个互斥锁“中毒”，以防止其他线程无意中误入这种已破坏的场景并让情况变得更糟。你仍然可以锁定已“中毒”的互斥锁并访问其中的数据，完全强制运行互斥代码。具体请参阅 `PoisonError::into_inner()` 的文档。但你肯定不会希望这发生在自己的意料之外。

19.3.7 使用互斥锁的多消费者通道

我们之前提到过，Rust 的通道是多生产者、单一消费者。或者更具体地说，一个通道只能有一个 Receiver。如果有一个线程池，则不能让其中的多个线程使用单个 mpsc 通道作为共享工作列表。

其实有一种非常简单的解决方法，只要使用标准库的一点点“能力”就可以。可以在 Receiver 周围包装一个 Mutex 然后再共享。下面就是这样做的一个模块：

```
pub mod shared_channel {
    use std::sync::{Arc, Mutex};
    use std::sync::mpsc::{channel, Sender, Receiver};

    /// 对`Receiver`的线程安全的包装
    #[derive(Clone)]
    pub struct SharedReceiver<T>(Arc<Mutex<Receiver<T>>>);

    impl<T> Iterator for SharedReceiver<T> {
        type Item = T;

        /// 从已包装的接收者中获取下一个条目
        fn next(&mut self) -> Option<T> {
            let guard = self.0.lock().unwrap();
            guard.recv().ok()
        }
    }

    /// 创建一个新通道，它的接收者可以跨线程共享。这会返回一个发送者和一个
    /// 接收者，就像标准库的`channel()`，有时可以作为无缝替代品使用
    pub fn shared_channel<T>() -> (Sender<T>, SharedReceiver<T>) {
```

```
let (sender, receiver) = channel();
    (sender, SharedReceiver(Arc::new(Mutex::new(receiver))))
}
```

我们正在使用 `Arc<Mutex<Receiver<T>>>`。这些泛型简直像俄罗斯套娃。这种情况在 Rust 中比在 C++ 中更常见。虽然这看起来会让人感到困惑，但通常情况下（就像在本例中一样），仅仅读出名称就可以帮你理解发生了什么，如图 19-11 所示。

分配在堆上的原子化引用计数的

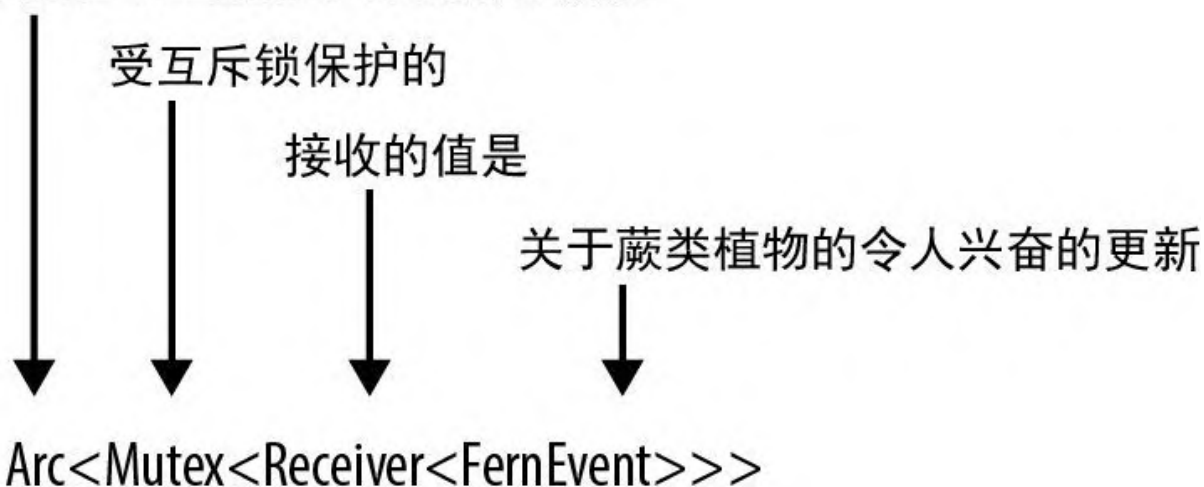


图 19-11：如何阅读复杂类型

19.3.8 读/写锁（`RwLock<T>`）

介绍完互斥锁，下面来看一下 `std::sync` 中提供的其他工具：Rust 标准库的线程同步工具包。我们将快速介绍，因为对这些工具的完整讨论超出了本书的范畴。

服务器程序通常都有一些只加载一次且很少更改的配置信息。大多数线程只会查询此配置，但由于配置可以更改（例如，可能要求服务器从磁盘重新加载其配置），所以无论如何都必须用锁进行保护。在这种情况下，可以使用互斥锁，但它会形成不必要的瓶颈。如果配置没有改变，那么各个线程就不应该轮流查询配置。这时就可以使用读 / 写锁或 `RwLock`。

互斥锁只有一个 `lock` 方法，而读 / 写锁有两个，即 `read` 和 `write`。`RwLock::write` 方法类似于 `Mutex::lock`。它会等待对受保护数据的独占的 `mut` 访问。`RwLock::read` 方法提供了非 `mut` 访问，它的优点是可能不怎么需要等待，因为本就可以让许多线程同时安全地读取。使用互斥锁，在任何给定时刻，受保护的数据都只有一个读取者或写入者（或两者都没有）。使用读 / 写锁，则可以有一个写入者或多个读取者，就像一般的 Rust 引用一样。

`FernEmpireApp` 可能有一个用作配置的结构体，由 `RwLock` 提供保护：

```
use std::sync::RwLock;

struct FernEmpireApp {
    ...
    config: RwLock<AppConfig>,
    ...
}
```

读取配置的方法会使用 `RwLock::read()`：

```
/// 如果应该使用试验性的真菌代码，则为True
fn mushrooms_enabled(&self) -> bool {
    let config_guard = self.config.read().unwrap();
    config_guard.mushrooms_enabled
}
```

重新加载配置的方法就要使用 `RwLock::write()`：

```
fn reload_config(&self) -> io::Result<()> {
    let new_config = AppConfig::load()?;
    let mut config_guard = self.config.write().unwrap();
    *config_guard = new_config;
    Ok(())
}
```

当然，Rust 特别适合在 `RwLock` 数据上执行安全规则。单写者或多读者的概念是 Rust 借用体系的核心。`self.config.read()` 会返回一个守卫，以提供对 `AppConfig` 的非 `mut`（共享）访问。

`self.config.write()` 会返回另一种类型的守卫，以提供 `mut`（独占）访问。

19.3.9 条件变量（**Condvar**）

通常线程需要一直等到某个条件变为真。

- 在关闭服务器的过程中，主线程可能需要等到所有其他线程都完成后才能退出。
- 当工作线程无事可做时，需要一直等待，直到有数据需要处理为止。
- 实现分布式共识协议的线程可能要等到一定数量的对等点给出响应为止。

有时，对于我们想要等待的确切条件，会有一个方便的阻塞式 API，比如服务器关闭示例中的 `JoinHandle::join`。其他情况下，则没有内置的阻塞式 API。程序可以使用条件变量来构建自己的 API。在 Rust 中，`std::sync::Condvar` 类型实现了条件变量。`Condvar` 中有方法 `.wait()` 和 `.notify_all()`，其中 `.wait()` 会阻塞线程，直到其他线程调用了 `.notify_all()`。

但条件变量的用途不止于此，因为说到底条件变量是关于受特定 `Mutex` 保护的某些数据的特定“真或假”条件。因此，`Mutex` 和 `Condvar` 是相关的。对条件变量的完整解释超出了本书的范畴，但为了让曾使用过条件变量的程序员更容易理解，我们将展示代码的两个关键部分。

当所需条件变为真时，就调用 `Condvar::notify_all`（或 `notify_one`）来唤醒所有等待的线程：

```
self.has_data_condvar.notify_all();
```

要进入睡眠状态并等待条件变为真，可以使用 `Condvar::wait()`：

```
while !guard.has_data() {  
    guard = self.has_data_condvar.wait(guard).unwrap();  
}
```

```
}
```

这个 `while` 循环是条件变量的标准用法。然而，`Condvar::wait` 的签名非比寻常。它会按值获取 `MutexGuard` 对象，消耗它，并在成功时返回新的 `MutexGuard`。这种签名给我们的直观感觉是 `wait` 方法会释放互斥锁并在返回之前重新获取它。按值传递 `MutexGuard` 要表达的意思是“我授予你通过 `.wait()` 方法释放互斥锁的独占权限。”

19.3.10 原子化类型

`std::sync::atomic` 模块包含用于无锁并发编程的原子化类型。这些类型与标准 C++ 原子化类型基本相同，但也有一些独特之处。

- `AtomicIsize` 和 `AtomicUsize` 是与单线程 `isize` 类型和 `usize` 类型对应的共享整数类型。
- `AtomicI8`、`AtomicI16`、`AtomicI32`、`AtomicI64` 及其无符号变体（如 `AtomicU8`）是共享整数类型，对应于单线程中的类型 `i8`、`i16` 等。
- `AtomicBool` 是一个共享的 `bool` 值。
- `AtomicPtr<T>` 是不安全指针类型 `*mut T` 的共享值。

正确使用原子化数据超出了本书的范畴。你只要明白多个线程可以同时读取和写入一个原子化的值而不会导致数据竞争就足够了。

与通常的算术运算符和逻辑运算符不同，原子化类型会暴露执行原子化操作的方法，单独的加载、存储、交换和算术运算都会作为一个单元安全地进行，哪怕其他线程也在执行操作同一内存的原子化操作也没问题。递增一个名为 `atom` 的 `AtomicIsize` 的代码如下所示：

```
use std::sync::atomic::{AtomicIsize, Ordering};

let atom = AtomicIsize::new(0);
atom.fetch_add(1, Ordering::SeqCst);
```

这些方法可以编译成专门的机器语言指令。在 x86-64 架构上，这个 `.fetch_add()` 调用会编译为 `lock incq` 指令，而普通 `n += 1` 可以编译为简单的 `incq` 指令或其他各种与此相关的变体。Rust 编译器还必须放弃围绕原子化操作的一些优化，因为与正常的加载或存储不同，它可以立即合法地影响其他线程或被其他线程影响。

参数 `Ordering::SeqCst` 是指内存排序。内存排序类似于数据库中的事务隔离级别。它们告诉系统，相对于性能，你有多关心诸如对因果性的影响和不存在时间循环之类的哲学概念。内存排序对于程序的正确性至关重要，而且很难进行理解和推理。不过令人高兴的是，选择顺序一致性（最严格的内存排序类型）的性能损失通常很低，与将 SQL 数据库置于 `SERIALIZABLE` 模式时的性能损失截然不同。因此，只要拿不准，就尽情使用 `Ordering::SeqCst` 吧。Rust 从标准 C++ 原子化机制继承了另外几种内存排序，分别对存续性和因果性提供了几种保证。我们就不在这里讨论它们了。

原子化的一个简单用途是中途取消。假设有一个线程正在执行一些长时间运行的计算（如渲染视频），我们希望能异步取消它。问题在于如何与希望关闭的线程进行通信。可以通过共享的 `AtomicBool` 来做到这一点：

```
use std::sync::Arc;
use std::sync::atomic::AtomicBool;

let cancel_flag = Arc::new(AtomicBool::new(false));
let worker_cancel_flag = cancel_flag.clone();
```

上述代码会创建两个 `Arc<AtomicBool>` 智能指针，它们都指向分配在堆上的 `AtomicBool`，其初始值为 `false`。第一个名为 `cancel_flag`，将留在主线程中。第二个名为 `worker_cancel_flag`，将转移给工作线程。

下面是工作线程的代码：

```
use std::thread;
use std::sync::atomic::Ordering;

let worker_handle = thread::spawn(move || {
```



```
    for pixel in animation.pixels_mut() {
        render(pixel); // 光线跟踪，需要花几微秒时间
        if worker_cancel_flag.load(Ordering::SeqCst) {
            return None;
        }
    }
    Some(animation)
});
```

渲染完每个像素后，线程会通过调用其 `.load()` 方法检查标志的值：

```
worker_cancel_flag.load(Ordering::SeqCst)
```

如果决定在主线程中取消工作线程，可以将 `true` 存储在 `AtomicBool` 中，然后等待线程退出：

```
// 取消渲染
cancel_flag.store(true, Ordering::SeqCst);

// 放弃结果，该结果有可能是`None`
worker_handle.join().unwrap();
```

当然，还有其他实现方法。此处的 `AtomicBool` 可以替换为 `Mutex<bool>` 或通道。主要区别在于原子化的开销是最低的。原子化操作从不使用系统调用。加载或存储通常会编译为单个 CPU 指令。

原子化是内部可变性的一种形式（就像 `Mutex` 或 `RwLock`），因此它们的方法会通过共享（非 `mut`）引用获取 `self`。这使得它们作为简单的全局变量时非常有用。

19.3.11 全局变量

假设我们正在编写网络代码。我们想要一个全局变量，即一个每当发出数据包时都会递增的计数器：

```
/// 服务器已成功处理的数据包的数量
static PACKETS_SERVED: usize = 0;
```

这可以正常编译，但有一个问题：PACKETS_SERVED 是不可变的，所以我们永远都不能改变它。

Rust 会尽其所能阻止全局可变状态。用 `const` 声明的常量当然是不可变的。默认情况下，静态变量也是不可变的，因此无法获得一个 `mut` 引用。`static` 固然可以声明为 `mut`，但访问它是不安全的。所有这些规则的制定，出发点都是 Rust 对线程安全的坚持。

全局可变状态也有不幸的软件工程后果：它往往使程序的各个部分更紧密耦合，更难测试，以后更难更改。尽管如此，在某些情况下并没有合理的替代，所以最好找到一种安全的方法来声明可变静态变量。

支持递增 PACKETS_SERVED 并保持其线程安全的最简单方式是让它变成原子化整数：

```
use std::sync::atomic::AtomicUsize;

static PACKETS_SERVED: AtomicUsize = AtomicUsize::new(0);
```

一旦声明了这个静态变量，增加数据包计数就很简单了：

```
use std::sync::atomic::Ordering;

PACKETS_SERVED.fetch_add(1, Ordering::SeqCst);
```

原子化全局变量仅限于简单的整数和布尔值。不过，要创建任何其他类型的全局变量，就要解决以下两个问题。

首先，变量必须以某种方式成为线程安全的，否则它就不能是全局变量：为了安全起见，静态变量必须同时是 `Sync` 和非 `mut` 的。幸运的是，我们已经看到了这个问题的解决方案。Rust 具有用于安全地共享变化的值的类型：`Mutex`、`RwLock` 和原子化类型。即使声明为非 `mut`，也可以修改这些类型。这就是它们的用途。（参见 19.3.3 节。）

其次，静态初始化程序只能调用被专门标记为 `const` 的函数，编译器可以在编译期间对其进行求值。换句话说，它们的输出是确定性

的，这些输出只会取决于它们的参数，而不取决于任何其他状态或 I/O。这样，编译器就可以将计算结果作为编译期常量嵌入了。这类似于 C++ 的 `constexpr`。

`Atomic` 类型（`AtomicUsize`、`AtomicBool` 等）的构造函数都是 `const` 函数，这使我们能够更早地创建 `static AtomicUsize`。一些其他类型，比如 `String`、`Ipv4Addr` 和 `Ipv6Addr`，同样有简单的 `const` 构造函数。

还可以直接在函数的签名前加上 `const` 来定义自己的 `const` 函数。Rust 将 `const` 函数可以做的事情限制在一小部分操作上，这些操作足够有用，同时仍然不会带来任何不确定的结果。`const` 函数不能以类型而只能以生命周期作为泛型参数，并且不能分配内存或对裸指针进行操作，即使在 `unsafe` 的块中也是如此。但是，我们可以使用算术运算 [包括回绕型算术 (wrapping arithmetic) 和饱和型算术 (saturating arithmetic)]、非短路逻辑运算和其他 `const` 函数。例如，可以创建便捷函数来更轻松的定义 `static` 和 `const` 并减少代码重复：

```
const fn mono_to_rgba(level: u8) -> Color {
    Color {
        red: level,
        green: level,
        blue: level,
        alpha: 0xFF
    }
}

const WHITE: Color = mono_to_rgba(255);
const BLACK: Color = mono_to_rgba(000);
```

结合这些技术，我们可能会试着像下面这样写：

```
static HOSTNAME: Mutex<String> =
    Mutex::new(String::new()); // 错误：静态调用仅限于常量函数、常量元
    组、
                                // 常量结构体和常量元组变体
```

不过很遗憾，虽然 `AtomicUsize::new()` 和 `String::new()` 是 `const fn`，但 `Mutex::new()` 不是²。为了绕过这些限制，需要使用 `lazy_static crate`。

²在 Rust 1.63 及以上版本中，`Mutex::new()` 已经是 `const fn` 了。——译者注

我们在 17.5.2 节介绍过 `lazy_static crate`。通过 `lazy_static!` 宏定义的变量允许你使用任何喜欢的表达式进行初始化，该表达式会在第一次解引用变量时运行，并保存该值以供后续操作使用。

可以像下面这样使用 `lazy_static` 声明一个全局 `Mutex` 控制的 `HashMap`：

```
use lazy_static::lazy_static;

use std::sync::Mutex;

lazy_static! {
    static ref HOSTNAME: Mutex<String> = Mutex::new(String::new());
}
```

同样的技术也适用于其他复杂的数据结构，比如 `HashMap` 和 `Deque`。对于根本不可变、只是需要进行非平凡初始化³的静态变量，它也非常方便。

³“非平凡初始化”是指初始化一个变量或对象时，需要进行一些复杂的计算或操作，而不是简单的赋值或默认初始化。在计算机编程中，有些变量或对象的初始化需要进行一些计算，比如读取配置文件、解析命令行参数、连接数据库等，这些初始化操作都可以称为“非平凡初始化”。与之相反，简单的赋值操作或者默认初始化可以称为“平凡初始化”。——译者注

使用 `lazy_static!` 会在每次访问静态数据时产生很小的性能成本。该实现使用了 `std::sync::Once`，这是一种专为一次性初始化而设计的底层同步原语。在幕后，每次访问惰性静态数据时，程序都会执行原子化加载指令以检查初始化是否已然发生。（`Once` 有比较特殊的用途，这里不做详细介绍。通常使用 `lazy_static!` 更方

便。但是，`std::sync::Once` 对于初始化非 Rust 库很有用，有关示例，请参阅 23.5 节。）

19.4 在 Rust 中编写并发代码的一点儿经验

本章介绍了在 Rust 中使用线程的 3 种技术：分叉与合并并行、通道和带锁的共享可变状态。我们的目标是好好介绍一下 Rust 提供的这些“零件”，重点在于如何将它们组合到实际程序中。

Rust 坚持安全性，因此从你决定编写多线程程序的那一刻起，重点就是构建安全、结构化的通信。保持线程近乎处于隔离态可以让 Rust 相信你的代码正在做的事是安全的。恰好，隔离也是确保你的代码正确且可维护的好办法。同样，Rust 也会引导你开发优秀的程序。

更重要的是，Rust 能让你组合多种技术并进行实验。你可以快速迭代。换句话说，“在编译器的督促下知错就改”肯定比“等出了问题后再调试数据竞争”能更快地开工并正确运行。

第 20 章 异步编程

假设你要编写一个聊天服务器。对于每个网络连接，都会有一些要解析的传入数据包、要组装的传出数据包、要管理的安全参数、要跟踪的聊天组订阅等。要想同时管理这么多连接，就得进行一定的组织工作。

理论上，可以为传入的每个连接启动一个单独的线程：

```
use std::{net, thread};

let listener = net::TcpListener::bind(address)?;

for socket_result in listener.incoming() {
    let socket = socket_result?;
    let groups = chat_group_table.clone();
    thread::spawn(|| {
        log_error(serve(socket, groups));
    });
}
```

对于每个新连接，这都会启动一个运行 `serve` 函数的新线程，此线程专注于管理单个连接所需的一切。

这确实很好，好得远远超出了预期，直到有一天突然涌入了数万个用户。每个线程都拥有 100 KiB 以上的栈，这很常见，但这可不是你花费数 GB 服务器内存的理由。如果要在多个处理器之间分配工作，那么线程固然好用，而且确有必要。但现在它们的内存需求已经太大了，所以通常在使用线程的同时，还要用一些补充手段来完成这些工作。

可以使用 Rust 异步任务在单个线程或工作线程池中交替执行许多彼此独立的活动。异步任务类似于线程，但其创建速度更快，在它们之间可以更有效地传递控制权，并且其内存开销比线程少一个数量级。在单个程序中同时运行数十万个异步任务是完全可行的。当然，你的应用程序可能仍会受到其他因素的制约，比如网络带宽、数据库速

度、算力，或此工作的固有内存需求，但与线程的开销相比，这些异步任务的固有内存开销只是九牛一毛。

一般来说，异步 Rust 代码看上去很像普通的多线程代码，但实际上那些可能导致阻塞的操作（如 I/O 或获取互斥锁）会以略有不同的方式处理。通过对这些操作进行特殊处理，Rust 能够获得关于这段代码行为的更多信息以辅助优化，这就是它能提高性能的原因。前面代码的异步版本如下所示：

```
use async_std::{net, task};

let listener = net::TcpListener::bind(address).await?;

let mut new_connections = listener.incoming();
while let Some(socket_result) = new_connections.next().await {
    let socket = socket_result?;
    let groups = chat_group_table.clone();
    task::spawn(async {
        log_error(serve(socket, groups).await);
    });
}
```

这里用的是 `async_std` 这个 crate 的网络模块和任务模块，并在可能发生阻塞的调用之后添加了 `.await`。但这段代码的整体结构与基于线程的版本无异。

本章的目标不仅是帮你编写异步代码，还要尽可能详细地展示它的工作原理，以便你可以预知如何在应用程序中执行异步代码以及把它用在哪里最能发挥出其价值。

- 为了展示异步编程的机制，我们会列举一组涵盖所有核心概念的最小语言特性集：Future（未来值）、异步函数、await 表达式、任务以及 `block_on` 执行器和 `spawn_local` 执行器。
- 然后，我们会介绍异步块和 `spawn` 执行器。这些在实际工作中非常重要，但从概念上讲，它们只是刚才提过的那些特性的变体。在此过程中，我们会指出你可能会遇到的一些异步编程特有的问题，并解释该如何处理这些问题。

- 为了展示所有这些“零件”是如何协同工作的，我们还会浏览一遍聊天服务器和客户端的完整代码，前面的代码片段只是其中的一部分。
- 为了说明原生 `Future` 和执行器的工作原理，我们会展示 `spawn_blocking` 和 `block_on` 的简单而实用的实现。
- 最后，我们会解释 `Pin` 类型，该类型在异步接口中会不时出现，以保证异步函数和异步式 `Future` 的安全使用。

20.1 从同步到异步

考虑调用以下（非异步，而是完全传统的）函数时会发生什么：

```
use std::io::prelude::*;
use std::net;

fn cheapo_request(host: &str, port: u16, path: &str)
    -> std::io::Result<String>
{
    let mut socket = net::TcpStream::connect((host, port))?;

    let request = format!("GET {} HTTP/1.1\r\nHost: {}\r\n\r\n",
path, host);
    socket.write_all(request.as_bytes())?;
    socket.shutdown(net::Shutdown::Write)?;

    let mut response = String::new();
    socket.read_to_string(&mut response)?;

    Ok(response)
}
```

这段代码会打开到 Web 服务器的 TCP 连接，以过时的协议向其发送一个极简的 HTTP 请求¹，然后读取其响应。图 20-1 展示了随着时间推移这个函数的执行情况。

¹如果确实需要 HTTP 客户端，请考虑使用像 surf 或 request 这样的众多优秀的 crate 中的任何一个，它们可以正确且异步地完成工作。这里的客户端最多只能得到 HTTPS 重定向的返回。



(续上图)

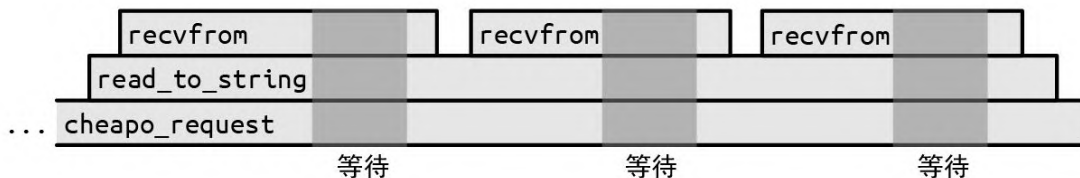


图 20-1：同步 HTTP 请求的进度（深灰色区域表示正在等待操作系统）

图 20-1 展示了当时间从左到右流逝时，函数的调用栈的情况。每个函数调用都是一个方框，叠放在其调用者上方。显然，`cheapo_request` 函数贯穿了整个执行过程。它会调用 Rust 标准库中的函数（如 `TcpStream::connect`）以及由 `TcpStream` 实现的 `write_all` 和 `read_to_string` 这两个特型方法。它们又会依次调用其他函数，但此程序最终会进行一些系统调用，请求操作系统实际完成某些操作，比如打开 TCP 连接，读取或写入一些数据。

深灰色背景表示程序正在等待操作系统完成系统调用的时间。我们没有按比例绘制这些时间。如果按比例绘制，则整张图都会变成深灰色：事实上，这个函数在几乎所有时间里都在等待操作系统。而前面代码的执行时间是系统调用之间的小窄条。

当这个函数正在等待系统调用返回时，它的单个线程是阻塞的，也就是说，在系统调用完成之前，它不能做任何其他事情。一个线程的栈大小有数十或数百 KB 的情况并不罕见，因此如果这是某个更大系统的一小部分，那么就会有許多线程在同时做类似的事情，如果仅仅为了等待而锁定这些线程资源则可能会让开销变得相当高。

为了解决这个问题，就要允许线程在等待系统调用完成期间进行其他工作。但要做到这一点并非易事。例如，我们用来从套接字读取响应的函数签名如下所示：

```
fn read_to_string(&mut self, buf: &mut String) ->
std::io::Result<usize>;
```

它直接在类型签名里表明，这个函数在完成工作或出现问题之前不会返回。因此这个函数是**同步的**：调用者在操作完成后才会继续。如果想在操作系统工作时将此线程用于其他任务，就需要一个新的 I/O 库来提供这个函数的异步版本。

20.1.1 Future

Rust 支持异步操作的方法是引入特型 `std::future::Future`：

```
trait Future {
    type Output;
    // 现在，暂时把`Pin<&mut Self>`当作`&mut Self`
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) ->
Poll<Self::Output>;
}

enum Poll<T> {
    Ready(T),
    Pending,
}
```

`Future` 代表一个你可以测试其完成情况的操作。`Future` 的 `poll`（轮询）方法从来不会等待操作完成，它总是立即返回。如果操作已完成，则 `poll` 会返回 `Poll::Ready(output)`，其中 `output` 是它的最终结果。否则，它会返回 `Pending`。如果 `Future` 值得再次轮询，它承诺会通过调用 `Context` 中提供的回调函数 `waker` 来通知我们。我们将这种实现方式称为异步编程的“皮纳塔²模型”：对于 `Future`，你唯一能做的就是通过轮询来“敲打”它，直到某个值“掉”出来。

²皮纳塔（piñata，西班牙语）是一种纸糊的容器，其中装满玩具与糖果，于节庆或生日宴会上悬挂起来，让人用棍棒敲打，一旦打破，玩具和糖果就会掉落下来。——译者注

所有现代操作系统都包含其系统调用的一些变体，我们可以使用它们来实现这种轮询接口。例如，在 Unix 和 Windows 上，如果将网络套

接字设置为非阻塞模式，那么一旦这些读写发生阻塞，就会返回某种错误。你必须稍后再试。

因此，异步版本的 `read_to_string` 的签名大致如下所示：

```
fn read_to_string(&mut self, buf: &mut String)
    -> impl Future<Output = Result<usize>>;
```

除了返回类型，这与我们之前展示过的签名基本相同：异步版本会返回携带 `Result<usize>` 的 `Future`。你需要轮询这个 `Future`，直到从中获得 `Ready(result)`。每次轮询时，都会尽可能读取更多的内容。最终 `result` 会为你提供成功值或错误值，就像普通的 I/O 操作一样。这是一种通用模式：任何函数的异步版本都会接受与其同步版本完全相同的参数，但返回类型包裹在 `Future` 中。

调用这个版本的 `read_to_string` 并没有实际读取任何内容，它唯一的职责是构建并返回一个 `Future`，该 `Future` 会在轮询时完成其真正的工作。这个 `Future` 必须包含执行调用请求所需的全部信息。例如，此 `read_to_string` 返回的 `Future` 必须记住调用它的输入流，以及附加了传入数据的 `String`。事实上，由于 `Future` 包含 `self` 和 `buf` 的引用，因此 `read_to_string` 的正确签名必然是如下形式：

```
fn read_to_string<'a>(&'a mut self, buf: &'a mut String)
    -> impl Future<Output = Result<usize>> + 'a;
```

这增加了生命周期以表明返回的 `Future` 的生存期只能与 `self` 和 `buf` 借用的值一样长。

`async-std` crate 提供了所有 `std` 中 I/O 设施的异步版本，包括带有 `read_to_string` 方法的异步 `Read` 特型。`async-std` 选择紧紧跟随 `std` 的设计，尽可能在它自己的接口中重用 `std` 的类型，因此 `Error`、`Result`、网络地址和大多数其他相关数据在“两个世界”之间是兼容的。熟悉 `std` 有助于使用 `async-std`，反之亦然。

Future 特型的一个规则是，一旦 Future 返回了 `Poll::Ready`，它就会假定自己永远不会再被轮询（poll）。当某些 Future 被过度轮询时，它们只会永远返回 `Poll::Pending`，而其他 Future 则可能会 panic 或被挂起。（但是，它们绝不会违反内存安全或线程安全规则，或以其他方式导致未定义行为。）Future 特型上的 fuse 适配器方法能把任何 Future 变成被过度轮询时总会返回 `Poll::Pending` 的 Future。但所有常用的 Future 消耗方式都会遵守这一规则，因此通常不必动用 fuse。

完全没必要一听到轮询就觉得效率低下。Rust 的异步架构是经过精心设计的，只要你正确实现了基本的 I/O 函数（如 `read_to_string`），就只会在值得尝试时才轮询 Future。每当调用 poll 时，必然有某个地方的某些代码返回了 Ready，或者至少朝着那个目标前进了一步。20.3 节会对此工作原理进行解释。

但使用 Future 似乎很具挑战性：当轮询时，如果得到了 `Poll::Pending`，应该做些什么呢？你必须四处寻找这个线程暂时可以做的其他工作，还不能忘记稍后回到这个 Future 并再次轮询它。整个程序将充斥着辅助性代码，以跟踪谁在等待处理以及一旦就绪应该做些什么之类的事情。cheapo_request 函数的简单性被破坏了。

好消息是，你大可不必这样做。

20.1.2 异步函数与 **await** 表达式

下面是一个写成异步函数的 cheapo_request 版本：

```
use async_std::io::prelude::*;
use async_std::net;

async fn cheapo_request(host: &str, port: u16, path: &str)
    -> std::io::Result<String>
{
    let mut socket = net::TcpStream::connect((host, port)).await?;

    let request = format!("GET {} HTTP/1.1\r\nHost: {}\r\n\r\n",
```

```
path, host);
    socket.write_all(request.as_bytes()).await?;
    socket.shutdown(net::Shutdown::Write)?;

    let mut response = String::new();
    socket.read_to_string(&mut response).await?;

    Ok(response)
}
```

除了以下几点，这段程序跟我们的原始版本几乎是每个字母都一样。

- 函数以 `async fn` 而不是 `fn` 开头。
- 使用 `async_std` crate 的异步版本的 `TcpStream::connect`、`write_all` 和 `read_to_string`。这些都会返回其结果的 `Future`。（本节中的示例使用了 `async_std` 的 1.7 版。）
- 每次返回 `Future` 的调用之后，代码都会 `.await`。虽然这看起来像是在引用结构体中名为 `await` 的字段，但它实际上是语言中内置的特殊语法，用于等待 `Future` 就绪。`await` 表达式的计算结果为 `Future` 的最终值。这就是函数从 `connect`、`write_all` 和 `read_to_string` 获取结果的方式。

与普通函数不同，当你调用异步函数时，它会在函数体开始执行之前立即返回。显然，调用的最终返回值还没有计算出来，你得到的只是承载它最终值的 `Future`。所以如果执行下面这段代码：

```
let response = cheapo_request(host, port, path);
```

那么 `response` 将是 `std::io::Result<String>` 型的 `Future`，而 `cheapo_request` 的函数体尚未开始执行。你不需要调整异步函数的返回类型，Rust 会自动把 `async fn f(...) -> T` 函数的返回值视为承载 `T` 的 `Future`，而非直接的 `T` 值。

异步函数返回的 `Future` 中包含函数体运行时所需的一切信息：函数的参数、局部变量的内存空间等。（就像是把要调用的栈帧捕获成了一个普通的 Rust 值。）所以 `response` 必须保存传给 `host`、

port 和 path 的值，因为 cheapo_request 的函数体将需要这些值来运行。

Future 的特化类型是由编译器根据函数的主体和参数自动生成的。这种类型没有名字，你只知道它实现了 Future<Output=R>，其中 R 是异步函数的返回类型。从这个意义上说，异步函数的 Future 就像闭包：闭包也有由编译器生成的匿名类型，该类型实现了 FnOnce 特型、Fn 特型和 FnMut 特型。

当你首次轮询 cheapo_request 返回的 Future 时，会从函数体的顶部开始执行，一直运行到 TcpStream::connect 返回的 Future 的第一个 await。await 表达式会轮询 connect 返回的 Future，如果它尚未就绪，则向调用者返回 Poll::Pending：程序不能从这个 await 继续向前运行了，直到对这个 Future 的某次轮询返回了 Poll::Ready。因此，表达式 TcpStream::connect(...).await 大致等价于如下内容：

```
{
    // 注意：这是伪代码，不是有效的Rust
    let connect_future = TcpStream::connect(...);
    'retry_point:
    match connect_future.poll(cx) {
        Poll::Ready(value) => value,
        Poll::Pending => {
            // 安排对`cheapo_request`返回的Future进行
            // 下一次`poll`，以便在'retry_point处恢复执行
            ...
            return Poll::Pending;
        }
    }
}
```

await 表达式会获取 Future 的所有权，然后轮询它。如果已就绪，那么 Future 的最终值就是 await 表达式的值，然后继续执行。否则，此 Future 返回 Poll::Pending。

但至关重要是，下一次对 cheapo_request 返回的 Future 进行轮询时不会从函数的顶部开始，而是会在即将轮询

`connect_future` 的中途时间点恢复执行函数。直到 `Future` 就绪之前，我们都不会继续处理异步函数的其余部分。

随着对其返回的 `Future` 继续进行轮询，`cheapo_request` 将通过函数体从一个 `await` 走到下一个，仅当它等待的子 `Future` 就绪时才会继续。因此，要对 `cheapo_request` 返回的 `Future` 进行多少次轮询，既取决于子 `Future` 的行为，也取决于该函数自己的控制流。`cheapo_request` 返回的 `Future` 会跟踪下一次 `poll` 应该恢复的点，以及恢复该点所需的所有本地状态，比如变量、参数和临时变量。

在函数中间暂停执行稍后再恢复，这种能力是异步函数所独有的。当一个普通函数返回时，它的栈帧就永远消失了。由于 `await` 表达式依赖于这种恢复能力，因此只能在异步函数中使用它们。

在撰写本章时，Rust 还不允许特型具有异步方法。只有自由函数以及从属于具体类型的函数才能是异步的。要解除此限制就要对语言进行一些更改。同时，如果确实需要定义包含异步函数的特型，请考虑使用 `async-trait` crate，它提供了基于宏的解决方案。

20.1.3 从同步代码调用异步函数：`block_on`

从某种意义上说，异步函数就是在转移责任。的确，在异步函数中很容易获得 `Future` 的值：只要使用 `await` 就可以。但是异步函数本身也会返回 `Future`，所以现在调用者的工作是以某种方式进行轮询。但最终还是得有人实际等待一个值。

可以使用 `async_std` 的 `task::block_on` 函数从普通的同步函数（如 `main`）调用 `cheapo_request`，这会接受一个 `Future` 并轮询，直到它生成一个值：

```
fn main() -> std::io::Result<()> {
    use async_std::task;

    let response = task::block_on(cheapo_request("example.com", 80,
"/"));
    println!("{}", response);
}
```

```

Ok ( )
}

```

由于 `block_on` 是一个会生成异步函数最终值的同步函数，因此可以将其视为从异步世界到同步世界的适配器。但 `block_on` 的阻塞式特征意味着我们不应该在异步函数中使用它，因为在值被准备好之前它会一直阻塞整个线程。异步函数中请改用 `await`。

图 20-2 展示了 `main` 的一种可能的执行方式。

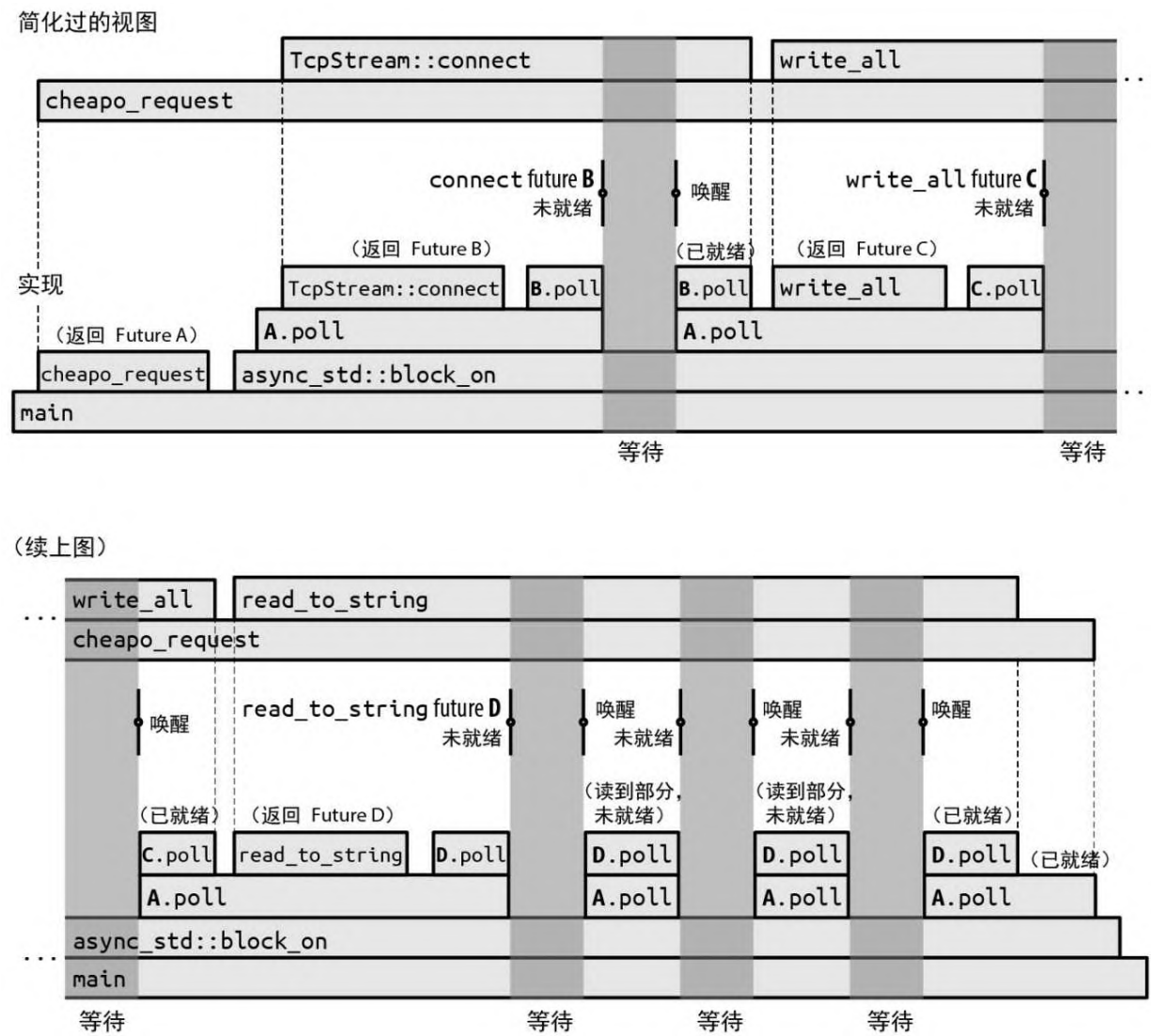


图 20-2：阻塞异步函数

上方的时间线（“简化过的视图”）部分展示了程序异步调用的抽象视图：`cheapo_request` 会首先调用 `TcpStream::connect` 以获得套接字，然后在该套接字上调用 `write_all` 和 `read_to_string`。接下来会返回。这与本章前面的 `cheapo_request` 同步版本的时间线非常相似。

但是其中的每一个异步调用都是一个多步骤的过程：创建一个 `Future`，然后轮询直到它就绪，也许在这个过程中创建并轮询了其他子 `Future`。下方的时间线（“实现”）部分展示了实现此异步行为的实际同步调用。这是了解普通异步执行中究竟发生了什么的一个好机会。

- 首先，`main` 会调用 `cheapo_request`，返回其最终结果的 `Future A`。然后 `main` 会将此 `Future` 传给 `async_std::block_on`，由后者对其进行轮询。
- 轮询 `Future A` 让 `cheapo_request` 的主体开始执行。它会调用 `TcpStream::connect` 来获取套接字的 `Future B`，然后对其进行等待。更准确地说，由于 `TcpStream::connect` 可能会遇到错误，因此 `B` 其实是 `Result<TcpStream, std::io::Error>` 型的 `Future`。
- `Future B` 会被 `await` 轮询。由于尚未建立网络连接，因此 `B.poll` 会返回 `Poll::Pending`，但会安排在套接字就绪后唤醒此调用任务。
- 由于 `Future B` 还没有就绪，因此 `A.poll` 会将 `Poll::Pending` 返回给自己的调用者 `block_on`。
- 由于 `block_on` 没有更好的事情可做，因此它进入了休眠状态。现在整个线程都被阻塞了。
- 如果 `Future B` 的连接就绪，就会唤醒轮询它的任务。这会激发 `block_on` 的行动，并再次尝试轮询 `Future A`。
- 轮询 `Future A` 会导致 `cheapo_request` 在其第一个 `await` 中恢复，并再次轮询 `Future B`。
- 这一次，`Future B` 就绪了：套接字创建完成，因此它将 `Poll::Ready(Ok(socket))` 返回给了 `A.poll`。

- 对 `TcpStream::connect` 的异步调用现已完成。因此 `TcpStream::connect(...).await` 表达式的值成了 `Ok(socket)`。
- `cheapo_request` 函数体正常执行，使用 `format!` 宏构建请求字符串，并将其传给 `socket.write_all`。
- 由于 `socket.write_all` 是一个异步函数，因此它会返回其结果的 `Future C`，而 `cheapo_request` 会等待这个 `Future`。

剩下的部分也类似。在图 20-2 所示的执行中，`socket.read_to_string` 返回的 `Future` 在就绪之前被轮询了 4 次，这些唤醒中的每一个都从套接字中读取了一些数据，但是 `read_to_string` 要求一直读到输入的末尾，这需要做一些操作。

编写一遍又一遍调用 `poll` 的循环听起来并不难。但是 `async_std::task::block_on` 的价值在于，它知道如何进入休眠直到 `Future` 真正值得再次轮询时再启动轮询，而不是浪费处理器时间和电池寿命进行数十亿次无结果的 `poll` 调用。像 `connect` 和 `read_to_string` 这样的基本 I/O 函数返回的 `Future` 保留了由传给 `poll` 的 `Context` 提供的唤醒器。到了应该唤醒 `block_on` 并再次尝试轮询时，就会调用此唤醒器。我们将在 20.3 节通过实现一个简单版本的 `block_on` 来准确地揭示它的工作原理。

与前面介绍的原始同步版本一样，为了等待操作完成，这个异步版本的 `cheapo_request` 花费了几乎所有时间。如果时间轴是按真实比例绘制的，那么图 20-2 将几乎完全是深灰色的，当程序偶尔唤醒时才会出现微小的碎片级计算时间。

这里有大量细节。幸运的是，通常可以只考虑简化过的上层时间线：一些函数调用是同步的，另一些函数调用是异步的且需要 `await`，但它们都只是函数调用。Rust 异步支持的成功之处在于能帮程序员在实践中使用简化过的视图，而不会反复被其具体实现分心。

20.1.4 启动异步任务

在 `Future` 的值就绪之前，`async_std::task::block_on` 函数会一直阻塞。但是把线程完全阻塞在单个 `Future` 上并不比同步调用好：本章的目标是让线程在等待的同时做其他工作。

为此，可以使用 `async_std::task::spawn_local`。该函数会接受一个 `Future` 并将其添加到任务池中，只要正阻塞着 `block_on` 的 `Future` 还未就绪，就会尝试轮询。因此，如果你将一堆 `Future` 传给 `spawn_local`，然后将 `block_on` 应用于最终结果的 `Future`，那么 `block_on` 就会在可以向前推进时轮询每个启动（`spawn`）后的 `Future`，并行执行整个任务池，直到你想要的结果就绪。

在撰写本章时，要想在 `async-std` 中使用 `spawn_local`，就必须启用该 crate 的 `unstable` 特性。为此，需要在 `Cargo.toml` 中使用下面这行代码去引用 `async-std`：

```
async-std = { version = "1", features = ["unstable"] }
```

`spawn_local` 函数是标准库的 `std::thread::spawn` 函数的异步模拟，用于启动线程。

`std::thread::spawn(c)` 会接受闭包 `c` 并启动线程来运行它，然后返回 `std::thread::JoinHandle`，其中 `std::thread::JoinHandle` 的 `join` 方法会等待线程完成并返回 `c` 中返回的任何内容。

`async_std::task::spawn_local(f)` 会接受 `Future f` 并将其添加到当前线程在调用 `block_on` 时要轮询的池中。`spawn_local` 会返回自己的 `async_std::task::JoinHandle` 类型，它本身就是一个 `Future`，你可以等待（`await`）它以获取 `f` 的最终值。

假设我们想同时发出一整套 HTTP 请求。下面是第一次尝试：

```
pub async fn many_requests(requests: Vec<(String, u16, String)>)
    -> Vec<std::io::Result<String>>
```

```

{
    use async_std::task;

    let mut handles = vec![];
    for (host, port, path) in requests {
        handles.push(task::spawn_local(cheapo_request(&host, port,
&path)));
    }

    let mut results = vec![];
    for handle in handles {
        results.push(handle.await);
    }

    results
}

```

该函数会在 `requests` 的每个元素上调用 `cheapo_request`，并将每个调用返回的 `Future` 传给 `spawn_local`。该函数还会将生成的 `JoinHandle` 收集到一个向量中，然后等待每一个 `JoinHandle`。可以用任意顺序等待这些 `JoinHandle`：由于请求已经发出，因此只要此线程调用了 `block_on` 并且没有更有价值的事情可做，请求的 `Future` 就会根据需要进行轮询。所有请求都将并行执行。一旦完成操作，`many_requests` 就会把结果返回给它的调用者。

前面的代码几乎是正确的，但 Rust 的借用检查器报错说它很担心 `cheapo_request` 返回的 `Future` 的生命周期：

```

error: `host` does not live long enough

    handles.push(task::spawn_local(cheapo_request(&host, port,
&path)));
                                           ^^^^^^
--
                                           |
                                           | borrowed value
does not
                                           |
                                           live long enough
argument requires that `host` is borrowed for
`'static`
}
- `host` dropped here while still borrowed

```

`path` 也会出现类似的错误。

自然，如果将引用传给一个异步函数，那么它返回的 `Future` 就必须持有这些引用，因此，安全起见，`Future` 的生命周期不能超出它们借来的值。这 and 任何包含引用的值所受的限制是一样的。

问题是 `spawn_local` 无法确定你会在 `host` 和 `path` 被丢弃之前等待任务完成。事实上，`spawn_local` 只会接受生命周期为 `'static` 的 `Future`，因为你也可以简单地忽略它返回的 `JoinHandle`，并在程序执行其他部分时让此任务继续运行。这不是异步任务独有的问题：如果尝试使用 `std::thread::spawn` 启动一个线程，那么该线程的闭包也会捕获对局部变量的引用，并得到类似的错误。

解决此问题的方法是创建另一个接受这些参数的拥有型版本的异步函数：

```
async fn cheapo_owning_request(host: String, port: u16, path:
String)
                                -> std::io::Result<String> {
    cheapo_request(&host, port, &path).await
}
```

此函数会接受 `String` 引用而不是 `&str` 引用，因此它的 `Future` 拥有 `host` 字符串和 `path` 字符串本身，并且其生命周期为 `'static`。通过借用检查器可以发现它立即开始等待 `cheapo_request` 返回的 `Future`，因此，如果该 `Future` 被轮询，那么它借用的 `host` 变量和 `path` 变量必然仍旧存在。一切顺利。

可以使用 `cheapo_owning_request` 像下面这样分发所有请求：

```
for (host, port, path) in requests {
    handles.push(task::spawn_local(cheapo_owning_request(host,
port, path)));
}
```

可以借助 `block_on` 从同步 `main` 函数中调用 `many_requests`:

```
let requests = vec![
    ("example.com".to_string(), 80, "/".to_string()),
    ("www.red-bean.com".to_string(), 80, "/".to_string()),
    ("en.wikipedia.org".to_string(), 80, "/".to_string()),
];

let results = async_std::task::block_on(many_requests(requests));
for result in results {
    match result {
        Ok(response) => println!("{}", response),
        Err(err) => eprintln!("error: {}", err),
    }
}
```

上述代码会在对 `block_on` 的调用中同时运行所有 3 个请求。每一个都会在某时取得进展，而其他的则会被阻塞，所有这些都发生在调用线程上。图 20-3 展示了对 `cheapo_request` 的 3 个调用的一种可能的执行方式。

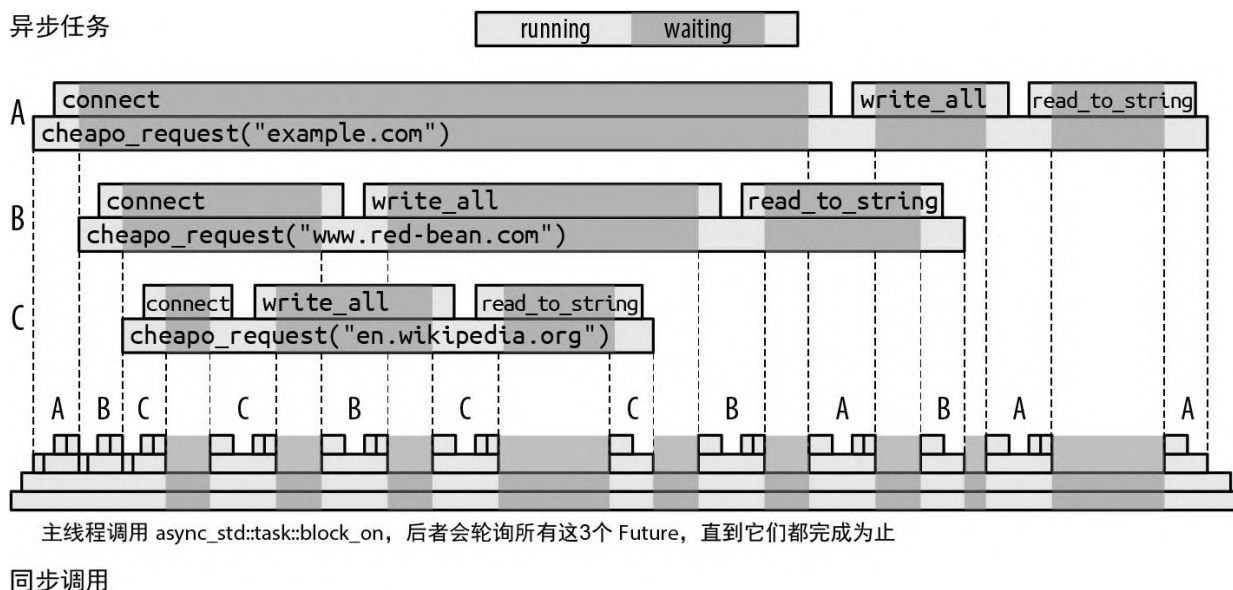


图 20-3：在单个线程上运行 3 个异步任务

（我们鼓励你尝试自己运行此代码，在 `cheapo_request` 的顶部和每个 `await` 表达式之后添加 `eprintln!` 调用，以便看出这些调用

在一次执行与下一次执行之间的交错方式有何不同。)

对 `many_requests` 的调用（为简单起见，图 20-3 中未展示）启动了 3 个异步任务，我们将其标记为 A、B 和 C。`block_on` 首先轮询 A，这样 A 会连接到 `example.com`。一旦返回了 `Poll::Pending`，`block_on` 就会将注意力转向下一个异步任务，轮询 B，并最终轮询 C，这样每个任务都会连接到各自的服务器。

当所有可轮询的 `Future` 都返回了 `Poll::Pending` 时，`block_on` 就会进入休眠状态，直到某个 `TcpStream::connect` 返回的 `Future` 表明它的任务值得再次轮询时才唤醒。

在本次执行中，服务器 `en.wikipedia.org` 比其他服务器响应更快，因此该任务首先完成。当启动的任务完成后，它会将值保存在 `JoinHandle` 中并标记为就绪，以便正在等候的 `many_requests` 可以继续处理。最终，对 `cheapo_request` 的其他调用要么成功了，要么返回了错误，而 `many_requests` 本身也可以返回了。最后，`main` 会从 `block_on` 接收到结果向量。

上述操作发生在同一个线程上，对 `cheapo_request` 的 3 个调用会通过对它们的 `Future` 的连续轮询交错进行。虽然异步调用看起来是单个函数调用一直运行到完成为止，但这种调用其实是通过 `Future` 的 `poll` 方法的一系列同步调用实现的。每个单独的 `poll` 调用都会快速返回，让进程空闲，以便轮询另一个异步调用。

我们终于达成了本章开头设定的目标：让线程在等待 I/O 完成时承担其他工作，这样线程的资源就不会在无所事事中浪费掉。更妙的是，此目标是通过与普通 Rust 代码非常相似的代码实现的：一些函数被标记为 `async`，一些函数调用后面跟着 `.await`，并且改用来自 `async_std` 而不是 `std` 的函数，除此之外，就和普通的 Rust 代码一模一样。

异步任务与线程的一个重要区别是：从一个异步任务到另一个异步任务的切换只会出现在 `await` 表达式处，且只有当等待的 `Future` 返回了 `Poll::Pending` 时才会发生。这意味着如果在

`cheapo_request` 中放置了一个长时间运行的计算，那么传给 `spawn_local` 的其他任务在它完成之前全都没有机会运行。使用线程则不会出现这个问题：操作系统可以在任何时候挂起任何线程，并设置定时器以确保没有哪个线程会独占处理器。异步代码要求共享同一个线程的各个 `Future` 自愿合作。如果想让长时间运行的计算与异步代码共存，可以参考 20.1.9 节讲到的一些选项。

20.1.5 异步块

除了异步函数，Rust 还支持异步块。普通的块语句会返回其最后一个表达式的值，而异步块会返回其最后一个表达式值的 `Future`。可以在异步块中使用 `await` 表达式。

异步块看起来就像普通的块语句，但其前面有 `async` 关键字：

```
let serve_one = async {
    use async_std::net;

    // 监听连接并接受其中一个
    let listener = net::TcpListener::bind("localhost:8087").await?;
    let (mut socket, _addr) = listener.accept().await?;

    // 在`socket`上与客户端对话
    ...
};
```

上述代码会将 `serve_one` 初始化为一个 `Future`（当被轮询时），以侦听并处理单个 TCP 连接。直到轮询 `serve_one` 时才会开始执行代码块的主体，就像直到轮询 `Future` 时才会开始执行异步函数的主体一样。

如果在异步块中使用 `?` 运算符处理错误，那么它只会从块中而不是围绕它的函数中返回。如果前面的 `bind` 调用返回了错误，则 `?` 运算符会将其作为 `serve_one` 的最终值返回。同样，`return` 表达式也会从异步块而不是其所在函数中返回。

如果异步块引用了围绕它的代码中定义的变量，那么它的 `Future` 就会捕获这些变量的值，就像闭包所做的那样。与 `move` 闭包（参见 14.1.2 节）的用法一样，也可以用 `async move` 启动该块以获取捕获的值的所有权，而不仅仅持有对它们的引用。

为了将你想要异步运行的那部分代码分离出去，异步块提供了一种简洁的方法。例如，在 20.1.4 节中，`spawn_local` 需要一个 `'static` 的 `Future`，因此我们定义了包装函数 `cheapo_owning_request` 来为我们提供一个拥有其参数所有权的 `Future`。只需从异步块中调用 `cheapo_request` 即可获得相同的效果，不用花心思去写包装函数：

```
pub async fn many_requests(requests: Vec<(String, u16, String)>)
    -> Vec<std::io::Result<String>>
{
    use async_std::task;

    let mut handles = vec![];
    for (host, port, path) in requests {
        handles.push(task::spawn_local(async move {
            cheapo_request(&host, port, &path).await
        }));
    }
    ...
}
```

由于这是一个 `async move` 块，因此它的 `Future` 获取了 `String` 值 `host` 和 `path` 的所有权，和 `move` 闭包一样。然后该 `Future` 会传递对 `cheapo_request` 的引用。借用检查器可以看到块的 `await` 表达式接手了 `cheapo_request` 返回的 `Future` 的所有权，因此对 `host` 和 `path` 的引用的生命周期不能比它们借来的已捕获变量的生命周期长。对于 `cheapo_owning_request` 所能做的事，`async` 块也能完成，且使用的样板代码更少。

你可能会遇到的一个棘手问题是，与异步函数不同，没有任何语法可用于指定异步块的返回类型。这在使用 `?` 运算符时会导致问题：

```
let input = async_std::io::stdin();
let future = async {
```

```

    let mut line = String::new();

    // 这会返回`std::io::Result<usize>`
    input.read_line(&mut line).await?;

    println!("Read line: {}", line);

    Ok(())
};

```

运行失败并出现以下错误：

```

error: type annotations needed
  |
48 |     let future = async {
  |         ----- consider giving `future` a type
...
60 |         Ok(())
  |         ^^ cannot infer type for type parameter `E` declared
  |            on the enum `Result`

```

Rust 无法判断异步块的返回类型是什么。`read\_line` 方法会返回 `Result<(), std::io::Error>`，但是因为 `?` 运算符会使用 `From` 特型将手头的错误类型转换为场景要求的任何类型，所以异步块的返回类型 `Result<(), E>` 中的 `E` 可以是实现了 `From<std::io::Error>` 的任意类型。

Rust 的未来版本中可能会新增相应的语法来指出 `async` 块的返回类型。目前，可以通过明确写出块的最终 `Ok` 的类型来解决这个问题：

```

let future = async {
    ...
    Ok::<(), std::io::Error> (())
};

```

由于 `Result` 是一个希望以成功类型和错误类型作为其参数的泛型类型，因此，如上例所示，可以在使用 `Ok` 或 `Err` 时指定这些类型参数。

20.1.6 从异步块构建异步函数

异步块为我们提供了另一种实现与异步函数相同效果的方式，并且这种方式更加灵活。例如，可以将我们的 `cheapo_request` 示例改写为一个普通的同步函数，该函数会返回一个异步块的 `Future`：

```
use std::io;
use std::future::Future;

fn cheapo_request<'a>(host: &'a str, port: u16, path: &'a str)
    -> impl Future<Output = io::Result<String>> + 'a
{
    async move {
        .....函数体.....
    }
}
```

当你调用这个版本的函数时，它会立即返回异步块返回值的 `Future`。这会捕获该函数的参数表，并且表现得就像异步函数返回的 `Future` 一样。由于没有使用 `async fn` 语法，因此需要在返回类型中写上 `impl Future`。但就调用者而言，这两个定义是具有相同函数签名的可互换实现。

如果想在调用函数时立即进行一些计算，然后再创建其结果的 `Future`，那么第二种方法会很有用。例如，另一种让 `cheapo_request` 和 `spawn_local` 协同工作的方法是将其变成一个返回 `'static Future` 的同步函数，这会捕获由其参数完全拥有的副本：

```
fn cheapo_request(host: &str, port: u16, path: &str)
    -> impl Future<Output = io::Result<String>> + 'static
{
    let host = host.to_string();
    let path = path.to_string();

    async move {
        .....使用&*host、port和path.....
    }
}
```

这个版本允许异步块将 `host` 和 `path` 捕获为拥有型 `String` 值，而不是 `&str` 引用。由于 `Future` 拥有其运行所需的全部数据，因

此它会在整个 'static 生命周期内有效。（在前面所展示的签名中我们明确写出了 + 'static，但 'static 本来就是各种 -> impl 返回类型的默认值，因此将其省略也不会有任何影响。）

由于这个版本的 cheapo_request 返回的是 'static Future，因此可以将它们直接传给 spawn_local。

```
let join_handle = async_std::task::spawn_local(
    cheapo_request("areweasyncyet.rs", 80, "/")
);

.....其他工作.....

let response = join_handle.await?;
```

20.1.7 在线程池中启动异步任务

迄今为止，我们展示的这些示例把几乎所有时间都花在了等待 I/O 上，但某些工作负载主要是 CPU 任务和阻塞的混合体。当计算量繁重到无法仅靠单个 CPU 满足时，可以使用 `async_std::task::spawn` 在工作线程池中启动 Future，线程池专门用于轮询那些已准备好向前推进的 Future。

`async_std::task::spawn` 用起来很像 `async_std::task::spawn_local`：

```
use async_std::task;

let mut handles = vec![];
for (host, port, path) in requests {
    handles.push(task::spawn(async move {
        cheapo_request(&host, port, &path).await
    }));
}
...
```

与 `spawn_local` 一样，`spawn` 也会返回一个 `JoinHandle` 值，你可以等待它，以获得 Future 的最终值。但与 `spawn_local` 不

同，Future 不必等到调用 `block_on` 才进行轮询。一旦线程池中的某个线程空闲了，该线程就会试着轮询它。

在实践中，`spawn` 比 `spawn_local` 用得更多。这只是因为人们更希望看到他们的工作负载在机器资源上均匀分配，而不关心工作负载的计算和阻塞是如何混杂的。

使用 `spawn` 时要记住一点：线程池倾向于保持忙碌。因此无论哪个线程率先得到轮询的机会，都会轮询到你的 Future。异步调用可能在一个线程上开始执行，阻塞在 `await` 表达式上，然后在另一个线程中恢复。因此，虽然将异步函数调用视为单一的、连续的代码执行是一种合理的简化（实际上，异步函数和 `await` 表达式的设计目标就是鼓励你以这种方式思考），但实际上可能会通过许多不同的线程来承载此次调用。

如果你正在使用线程本地存储，可能会惊讶地看到你在 `await` 表达式之前放置的数据后来被换成了完全不同的东西。这是因为你的任务现在正由线程池中的不同线程轮询。如果你觉得这是一个问题，就应该改用任务本地存储，具体请参阅 `async-std crate` 的 `task_local!` 宏的详细信息。

20.1.8 你的 Future 实现 Send 了吗

`spawn` 具有 `spawn_local` 所没有的一项限制。由于 Future 会被发送到另一个线程运行，因此它必须实现标记特型 `Send`（参见 19.2.5 节）。只有当 Future 包含的所有值都符合 `Send` 要求时，它自己才符合 `Send` 要求：所有函数参数、局部变量，甚至匿名临时值都必须安全地转移给另一个线程。

和生命周期方面的限制一样，这项要求也不是异步任务独有的：如果尝试用 `std::thread::spawn` 启动其闭包以捕获非 `Send` 值的线程，那么也会遇到类似的错误。不同点在于，虽然传给 `std::thread::spawn` 的闭包会留在创建并运行它的线程上，但在线程池中启动的 Future 可以在等待期间的任意时刻从一个线程转移给另一个线程。

这项限制很容易意外触发。例如，下面的代码乍看起来没问题：

```
use async_std::task;
use std::rc::Rc;

async fn reluctant() -> String {
    let string = Rc::new("ref-counted string".to_string());

    some_asynchronous_thing().await;

    format!("Your splendid string: {}", string)
}

task::spawn(reluctant());
```

异步函数的 Future 需要保存足够的信息，以便此函数能从 await 表达式继续。在这种情况下，reluctant 返回的 Future 必须在 await 之后使用 string 的值，因此 Future（至少在某些时刻）会包含一个 Rc<String> 值。由于 Rc 指针不能在线程之间安全地共享，因此 Future 本身也不能是 Send 的。因为 spawn 只接受符合 Send 要求的 Future，所以 Rust 不会接受 Rc 指针：

```
error: future cannot be sent between threads safely
   |
17  |         task::spawn(reluctant());
   |         ^^^^^^^^^^^ future returned by `reluctant` is not `Send`
   |
   |
127 | T: Future + Send + 'static,
   |         ---- required by this bound in
`async_std::task::spawn`
   |
   = help: within `impl Future`, the trait `Send` is not
implemented
           for `Rc<String>`
note: future is not `Send` as this value is used across an await
   |
10  |         let string = Rc::new("ref-counted
string".to_string());
   |         ----- has type `Rc<String>` which is not `Send`
11  |
12  |         some_asynchronous_thing().await;
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
           await occurs here, with `string` maybe used later
```



```
...
15 |     }
    |     - `string` is later dropped here
```

此错误消息很长，包含很多有用的详细信息。

- 解释了为什么 `Future` 需要符合 `Send` 的要求： `task::spawn` 需要它。
- 解释了哪个值不符合 `Send` 的要求：局部变量 `string`，其类型是 `Rc<String>`。
- 解释了为什么 `string` 会影响 `Future`：它的作用域跨越了 `await`³。

³由于 `await` 之前和之后不属于同一个作用域，因此需要保存上下文。——译者注

有两种方法可以解决此问题。一种方法是限制非 `Send` 值的作用域，使其不跨越任何 `await` 表达式的作用域，因此也不需要保存在函数的 `Future` 中：

```
async fn reluctant() -> String {
    let return_value = {
        let string = Rc::new("ref-counted string".to_string());
        format!("Your splendid string: {}", string)
        // `Rc<String>`在此离开了作用域.....
    };

    // .....因此当我们在这里暂停时，它不在周边环境里
    some_asynchronous_thing().await;

    return_value
}
```

另一种方法是简单地使用 `std::sync::Arc` 而非 `Rc`。`Arc` 使用原子更新来管理引用计数，这会让它略慢，但 `Arc` 指针是符合 `Send` 要求的。

虽然最终你将学会识别和避免非 `Send` 类型，但一开始它们可能有点儿令人吃惊。[至少，我们（本书作者）曾感到惊讶。] 例如，旧的 Rust 代码有时会使用下面这样的泛型结果类型：

```
// 别这样做！
type GenericError = Box<dyn std::error::Error>;
type GenericResult<T> = Result<T, GenericError>;
```

这个 `GenericError` 类型使用了装箱过的特型对象来保存实现了 `std::error::Error` 的任意类型的值，但没有对它施加任何进一步的限制：如果有某个非 `Send` 类型实现了 `Error`，那么就可以将该类型的装箱值转换为 `GenericError`。由于这种可能性，`GenericError` 不符合 `Send` 要求，并且下面的代码无法工作：

```
fn some_fallible_thing() -> GenericResult<i32> {
    ...
}

// 这个函数的Future不符合`Send`要求.....
async fn unfortunate() {
    // .....因此调用的值.....
    match some_fallible_thing() {
        Err(error) => {
            report_error(error);
        }
        Ok(output) => {
            // .....其生命周期跨越了这个await.....
            use_output(output).await;
        }
    }
}

// .....因此这个`spawn`会出错
async_std::task::spawn(unfortunate());
```

与前面的示例一样，编译器的错误消息解释了正在发生的事情，并指出 `Result` 类型是罪魁祸首。由于 Rust 认为 `some_fallible_thing` 的结果存在于整个 `match` 语句（包括 `await` 表达式）中，所以它确定 `unfortunate` 返回的 `Future` 不符合 `Send` 的要求。对于这个错误，Rust 过于谨慎了：虽然 `GenericError` 确实不能安全地发送到另一个线程，但 `await` 只有在结果为 `Ok` 时才会发生，因此当我们等待 `use_output` 返回的 `Future` 时其实并不存在错误值。

理想的解决方案是使用更严格的泛型错误类型，比如 7.2.5 节提到的错误类型：

```
type GenericError = Box<dyn std::error::Error + Send + Sync +
'static>;
type GenericResult<T> = Result<T, GenericError>;
```

这个特型对象会明确要求底层错误类型实现 `Send`。一切顺利。

即使你的 `Future` 不符合 `Send` 要求，而且不容易把它变成符合形式，仍然可以使用 `spawn_local` 在当前线程上运行它。当然，你需要确保此线程会在某个时刻调用 `block_on` 以便让它有机会运行，并且你无法受益于跨多个处理器分派工作的能力。

20.1.9 长时间运行的计算：`yield_now` 与 `spawn_blocking`

为了让 `Future` 更好地与其他任务共享线程，它的 `poll` 方法应该总是尽可能快地返回。但是，如果你正在进行长时间的计算，就可能需要很长时间才能到达下一个 `await`，从而让其他异步任务等待的时间比你预想的更久些。

避免这种情况的一种方法是偶尔等待某些事情。

`async_std::task::yield_now` 函数会返回一个为此而设计的简单的 `Future`：

```
while computation_not_done() {
    .....完成一个中等规模的计算步骤.....
    async_std::task::yield_now().await;
}
```

当 `yield_now` 返回的 `Future` 第一次被轮询时，它会返回 `Poll::Pending`，但表示自己很快就值得再次轮询。因此你的异步调用放弃了线程，以使其他任务有机会运行，但很快会再次轮到它。第二次轮询 `yield_now` 返回的 `Future` 时，它会返回 `Poll::Ready(())`，让你的异步函数恢复执行。

然而，这种方法并不总是可行。如果你使用外部 crate 进行长时间运行的计算或者调用 C 或 C++，那么将上述代码更改为异步友好型代码可能并不方便。或者很难确保计算所经过的每条路径一定会时不时地等待一下。

对于这种情况，可以使用

`async_std::task::spawn_blocking`。该函数会接受一个闭包，开始在独立的线程上运行它，并返回携带其返回值的 `Future`。异步代码可以等待那个 `Future`，将其线程让给其他任务，直到本次计算就绪。通过将繁重的工作放在单独的线程上，可以委托给操作系统去负责，让它更友善地分享处理器。

假设我们要根据存储在身份验证数据库中的密码哈希值来检查用户提供的密码。为安全起见，验证密码需要进行大量计算，这样即使攻击者获得了数据库的副本，也无法简单地通过尝试数万亿个可能的密码来查看是否有匹配项。`argonautica` crate 提供了一个专为存储密码而设计的哈希函数：正确生成的 `argonautica` 哈希需要相当一部分时间才能验证。可以在异步应用程序中使用 `argonautica` (0.2 版)，如下所示：

```
async fn verify_password(password: &str, hash: &str, key: &str)
    -> Result<bool, argonautica::Error>
{
    // 制作参数的副本，以使闭包的生命周期是'static
    let password = password.to_string();
    let hash = hash.to_string();
    let key = key.to_string();

    async_std::task::spawn_blocking(move || {
        argonautica::Verifier::default()
            .with_hash(hash)
            .with_password(password)
            .with_secret_key(key)
            .verify()
    }).await
}
```

如果 `password` 与 `hash` 匹配，则返回 `Ok(true)`，给定的 `key` 是整个数据库的键。通过在传给 `spawn_blocking` 的闭包中进行验

证，可以将昂贵的计算推给其各自的线程，确保它不会影响到其他用户请求的响应。

20.1.10 对几种异步设计进行比较

在许多方面，Rust 的异步编程方式与其他语言所采用的方法相似。例如，JavaScript、C# 和 Rust 都有带 `await` 表达式的异步函数。所有这些语言都有代表未完成计算的值：Rust 中叫作“Future”，JavaScript 中叫作“承诺”（Promise），C# 中叫作“任务”（Task），但它们都代表一种你可能不得不等待的值。

然而，Rust 对轮询的使用独树一帜。在 JavaScript 和 C# 中，异步函数在调用后会立即开始运行，并且系统库中内置了一个全局事件循环，可在等待的值可用时恢复挂起的异步函数调用。不过，在 Rust 中，异步调用什么都不会做，直到你将它传给 `block_on`、`spawn` 或 `spawn_local` 之类的函数，这些函数将轮询它并驱动此事直到完成。我们称这些函数为**执行器**，它们承担着与其他语言中全局事件循环类似的职责。

因为 Rust 会让你（程序员）选择一个执行器来轮询你的 Future，所以它并不需要在系统中内置全局事件循环。`async-std` crate 提供了迄今为止本章使用过的这些执行器函数，但是 `tokio` crate（本章稍后会用到）自己定义了一组类似的执行器函数。在本章的末尾，我们将实现自己的执行器。你可以在同一个程序中使用这 3 种执行器。

20.1.11 一个真正的异步 HTTP 客户端

如果不展示一个正确使用异步 HTTP 客户端 crate 的例子，那本章就是不完整的，因为它非常简单，并且确有几个不错的 crate 可供选择，包括 `request` 和 `surf`。

下面是对 `many_requests` 的重写，它甚至比基于 `cheapo_request` 的重写更简单，而且会用 `surf` 同时运行一系列请求。你需要在 `Cargo.toml` 文件中添加如下依赖项：

```
[dependencies]
async-std = "1.7"
surf = "1.0"
```

然后，可以像下面这样定义 `many_requests`:

```
pub async fn many_requests(urls: &[String])
    -> Vec<Result<String, surf::Exception>>
{
    let client = surf::Client::new();

    let mut handles = vec![];
    for url in urls {
        let request = client.get(&url).recv_string();
        handles.push(async_std::task::spawn(request));
    }

    let mut results = vec![];
    for handle in handles {
        results.push(handle.await);
    }

    results
}

fn main() {
    let requests = &["http://example.com".to_string(),
        "https://www.red-bean.com".to_string(),
        "https://en.wikipedia.org/wiki/Main_Page".to_string()];

    let results =
    async_std::task::block_on(many_requests(requests));
    for result in results {
        match result {
            Ok(response) => println!("*** {}\\n", response),
            Err(err) => eprintln!("error: {}\\n", err),
        }
    }
}
```

使用单个 `surf::Client` 发出所有请求可以让我们重用 HTTP 连接（如果其中有多个请求指向同一台服务器的话），并且不需要异步块：因为 `recv_string` 是一个返回 `Send + 'static` 型

`Future` 的异步方法，所以可以将它返回的 `Future` 直接传给 `spawn`。

20.2 异步客户端与服务器

现在，我们要把这些已讨论过的关键思想组合成一个真正可用的程序。在很大程度上，异步应用程序和普通的多线程应用程序非常相似，但在某些需要紧凑而且富有表现力的代码的场合，异步编程可以大显身手。

本节的示例是聊天服务器和客户端。真正的聊天系统是很复杂的，涉及从安全、重新连接到隐私和内部审核的各种问题，但我们已将此系统缩减为一组非常基础的特性，来把注意力聚焦于少数我们感兴趣的要点上。

特别是，我们希望能好好处理背压。也就是说，即使一个客户端的网络连接速度较慢或完全断开连接，也绝不能影响其他客户端按照自己的节奏交换消息。由于“龟速”客户端不应该让服务器花费无限的内存来保存其不断增长的积压消息，因此我们的服务器应该丢弃那些发给掉队客户端的消息，但也有义务提醒他们其信息流不完整。（一个真正的聊天服务器会将消息记录到磁盘并允许客户端检索他们错过的消息，但这里不考虑那样做。）

使用命令 `cargo new --lib async-chat` 启动项目，并将以下文本放入 `async-chat/Cargo.toml` 中：

```
[package]
name = "async-chat"
version = "0.1.0"
authors = ["You <you@example.com>"]
edition = "2021"

[dependencies]
async-std = { version = "1.7", features = ["unstable"] }
tokio = { version = "1.0", features = ["sync"] }
serde = { version = "1.0", features = ["derive", "rc"] }
serde_json = "1.0"
```

我们依赖于 4 个 crate。

- `async-std` crate 是本章中一直在用的异步 I/O 基础构件和实用工具的集合。
- `tokio` crate 是类似于 `async-std` crate 的另一个异步基础构件集合，它也是最古老且最成熟的 crate 之一。`tokio` crate 应用广泛，设计和实现的标准都很高，但使用时需要比 `async-std` crate 更加小心。

`tokio` 是一个大型 crate，但我们只需要其中的一个组件，因此 `Cargo.toml` 依赖行中的 `features = ["sync"]` 字段将 `tokio` 缩减为了我们需要的部分，使其成为一种轻型依赖。当异步库生态系统还不太成熟时，人们会避免在同一个程序中同时使用 `tokio` 和 `async-std`。不过，只要遵循这两个项目各自 crate 文档中的规则，就可以在同一个程序中使用。

- `serde` 和 `serde_json` 是第 18 章中介绍过的两个 crate。它们为我们提供了方便且高效的工具来生成和解析 JSON，我们的聊天协议使用 JSON 来表示网络上的数据。我们想使用 `serde` 的一些可选特性，因此会在提供依赖项时选择它们。

我们的聊天应用程序、客户端和服务器的整体结构如下所示：

```
async-chat
├── Cargo.toml
├── src
│   ├── lib.rs
│   ├── utils.rs
│   └── bin
│       ├── client.rs
│       └── server
│           ├── main.rs
│           ├── connection.rs
│           ├── group.rs
│           └── group_table.rs
```

这个包的布局使用了 8.4 节中提到的一项 Cargo 特性：除了主库 crate `src/lib.rs` 及其子模块 `src/utils.rs`，还包括两个可执行文

件。

- `src/bin/client.rs` 是聊天客户端的单文件可执行文件。
- `src/bin/server` 是服务端的可执行文件，分布在 4 个文件中：`main.rs` 包含 `main` 函数，另外 3 个子模块分别是 `connection.rs`、`group.rs` 和 `group_table.rs`。

我们将在本章中展示每个源文件的内容，如果它们都就位了，那么一旦在此目录树中键入 `cargo build`，就会编译库的 `crate`，然后构建出两个可执行文件。Cargo 会自动包含库的 `crate` 作为依赖项，使其成为放置客户端和服务端共享定义的约定位置。同样，`cargo check` 会检查整棵源代码树。要运行任何一个可执行文件，可以使用如下命令：

```
$ cargo run --release --bin server -- localhost:8088
$ cargo run --release --bin client -- localhost:8088
```

`--bin` 选项会指出要运行哪个可执行文件，而 `--` 选项后面的任何参数都会传给可执行文件本身。我们的客户端和服务端只希望知道服务器的地址和 TCP 端口。

20.2.1 **Error** 类型与 **Result** 类型

库 `crate` 的 `utils` 模块定义了要在整个应用程序中使用的 `Error` 类型与 `Result` 类型。以下来自 `src/utils.rs`：

```
use std::error::Error;

pub type ChatError = Box<dyn Error + Send + Sync + 'static>;
pub type ChatResult<T> = Result<T, ChatError>;
```

这些是我们在 7.2.5 节中建议的泛型错误类型。`async_std` `crate`、`serde_json` `crate` 和 `tokio` `crate` 也分别定义了自己的错误类型，但是 `?` 运算符可以自动将它们全部转换为 `ChatError`，这是借助标准库的 `From` 特型实现的，该特型可以将任何合适的错误类型转换为 `Box<dyn Error + Send + Sync + 'static>` 类

型。类型限界 `Send` 和 `Sync` 会确保，如果在另一个线程中启动的任务失败，那么它可以安全地将错误报告给主线程。

在实际的应用程序中，请考虑使用 `anyhow crate`，它提供了与这里类似的 `Error` 类型和 `Result` 类型。`anyhow crate` 易于使用，而且提供了一些超越 `ChatError` 和 `ChatResult` 的优秀特性。

20.2.2 协议

库 `crate` 以下面这两种类型来支持整个聊天协议，这是在 `lib.rs` 中定义的：

```
use serde::{Deserialize, Serialize};
use std::sync::Arc;

pub mod utils;

#[derive(Debug, Deserialize, Serialize, PartialEq)]
pub enum FromClient {
    Join { group_name: Arc<String> },
    Post {
        group_name: Arc<String>,
        message: Arc<String>,
    },
}

#[derive(Debug, Deserialize, Serialize, PartialEq)]
pub enum FromServer {
    Message {
        group_name: Arc<String>,
        message: Arc<String>,
    },
    Error(String),
}

#[test]
fn test_fromclient_json() {
    use std::sync::Arc;

    let from_client = FromClient::Post {
        group_name: Arc::new("Dogs".to_string()),
        message: Arc::new("Samoyeds rock!".to_string()),
    };
};
```

```

    let json = serde_json::to_string(&from_client).unwrap();
    assert_eq!(json,
               r#"{"Post":{"group_name":"Dogs","message":"Samoyeds
rock!"}}"#);

    assert_eq!(serde_json::from_str::(<FromClient>(&json).unwrap(),
               from_client);
}

```

FromClient 枚举表示可以从客户端发送到服务器的数据包：它可以请求加入一个组或向已加入的任何组发布消息。FromServer 表示可以由服务器发回的内容，即发布到某个组的消息或错误消息。可以使用带引用计数的 Arc<String> 而不是普通的 String，这有助于服务器在管理组和分发消息时避免复制字符串。

#[derive] 属性要求 serde crate 为 FromClient 和 FromServer 生成其 Serialize 特型和 Deserialize 特型的实现。这样一来，就可以调用 serde_json::to_string 将它们转换为 JSON 值，通过网络进行发送，最后再调用 serde_json::from_str 转换回它们的 Rust 形式。

test_fromclient_json 这个单元测试演示了它的用法。给定由 serde 派生的 Serialize 实现，可以调用 serde_json::to_string 将给定的 FromClient 值转换为这样的 JSON：

```

{"Post":{"group_name":"Dogs","message":"Samoyeds rock!"}}

```

然后，派生出的 Deserialize 实现会将其解析回等效的 FromClient 值。请注意，FromClient 中的 Arc 指针对其序列化形式没有任何影响：引用计数字符串会直接显示为 JSON 对象成员的值。

20.2.3 获取用户输入：异步流

我们的聊天客户端的首要职责是读取用户的命令并将相应的数据包发送到服务器。管理一个合适的用户界面超出了本章的范围，所以我们将做最简单可行的事情：直接从标准输入中读取行。以下代码位于 `src/bin/client.rs` 中：

```
use async_std::prelude::*;
use async_chat::utils::{self, ChatResult};
use async_std::io;
use async_std::net;

async fn send_commands(mut to_server: net::TcpStream) ->
ChatResult<()> {
    println!("Commands:\n\
        join GROUP\n\
        post GROUP MESSAGE...\n\
        Type Control-D (on Unix) or Control-Z (on Windows) \
        to close the connection.");

    let mut command_lines =
io::BufReader::new(io::stdin()).lines();
    while let Some(command_result) = command_lines.next().await {
        let command = command_result?;
        // 参见GitHub存储库中对`parse_command`的定义
        let request = match parse_command(&command) {
            Some(request) => request,
            None => continue,
        };

        utils::send_as_json(&mut to_server, &request).await?;
        to_server.flush().await?;
    }

    Ok(())
}
```

这会调用 `async_std::io::stdin` 来获取客户端标准输入的异步句柄，并包装在 `async_std::io::BufReader` 中对其进行缓冲，然后调用 `lines` 逐行处理用户的输入。它会尝试将每一行解析为与某个 `FromClient` 值相对应的命令，如果成功，就将该值发送到服务器。如果用户输入了无法识别的命令，那么 `parse_command` 就会打印一条错误消息并返回 `None`，以便 `send_commands` 可以重新开始循环。如果用户键入了文件结束（EOF）指示符，则 `lines` 流会返

回 `None`，并且 `send_commands` 也会返回。此代码与你在普通同步程序中编写的代码非常相似，只不过它使用的是 `async_std` 版本的库特性。

异步 `BufReader` 的 `lines` 方法很有趣。它没有像标准库那样返回一个迭代器：`Iterator::next` 方法是一个普通的同步函数，因此调用 `command_lines.next()` 会阻塞线程，直到下一行代码就绪。而这里的 `lines` 会返回一个 `Result<String>` 值组成的流。流是迭代器的异步模拟，它会用异步友好的方式按需生成一系列值。下面是 `async_std::stream` 模块中 `Stream` 特型的定义：

```
trait Stream {
    type Item;

    // 现在，把`Pin<&mut Self>`读取为`&mut Self`
    fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<Option<Self::Item>>;
}
```

可以将 `Stream` 特型视为 `Iterator` 特型和 `Future` 特型的混合体。与迭代器一样，`Stream` 也有关联的 `Item` 类型，并使用 `Option` 来指示序列何时结束。同时，与 `Future` 一样，流必须被轮询：要获取下一个条目（或了解流是否结束），就必须调用 `poll_next` 直到它返回 `Poll::Ready`。流的 `poll_next` 实现应该总是快速返回，不会阻塞。如果流返回了 `Poll::Pending`，则必须在值得再次轮询时通过 `Context` 通知调用者。

`poll_next` 方法很难直接使用，不过通常也不需要直接使用该方法。与迭代器一样，流有很多实用方法，比如 `filter` 和 `map`。在这些方法中，有一个 `next` 方法，它会返回流中下一个 `Option<Self::Item>` 的 `Future`。可以调用 `next` 并等待它返回的 `Future`，而不必显式轮询流。

把这些片段结合起来看，`send_commands` 会利用 `while let` 和 `next` 循环遍历输入行组成的流来消耗这个流：

```
while let Some(item) = stream.next().await {  
    .....使用条目.....  
}
```

（Rust 可能会在未来版本中引入可用来消耗流的异步 `for` 循环，就像普通 `for` 循环能消耗 `Iterator` 值一样。）

在流结束后（也就是说，在流返回 `Poll::Ready(None)` 指出流已结束之后）轮询流，就像在迭代器返回 `None` 之后调用 `next`，或者在 `Future` 返回 `Poll::Ready` 之后轮询 `Future: Stream` 特型没有规定此时流应该怎么做，某些流可能行为诡异。与 `Future` 和迭代器一样，流也有一个 `fuse` 方法来确保此类调用的行为在必要时是可预测的。有关详细信息，请参阅在线文档。

使用流时，务必记住使用 `async_std` 预导入：

```
use async_std::prelude::*;
```

这是因为 `Stream` 特型的实用方法（如 `next`、`map`、`filter` 等）实际上并没有定义在自身上，而是单独特型 `StreamExt` 上的默认方法，该特型会自动为所有 `Stream` 实现：

```
pub trait StreamExt: Stream {  
    .....把一些实用工具方法定义为默认方法.....  
}  
  
impl<T: Stream> StreamExt for T { }
```

这是 11.2.2 节描述的扩展特型模式的示例。`async_std::prelude` 模块会将 `StreamExt` 方法引入作用域，因此使用预导入可以确保这些方法在你的代码中可见。

20.2.4 发送数据包

为了在网络套接字上传输数据包，我们的客户端和服务端会使用 `crate` 的 `utils` 模块中的 `send_as_json` 函数：

```
use async_std::prelude::*;
use serde::Serialize;
use std::marker::Unpin;

pub async fn send_as_json<S, P>(outbound: &mut S, packet: &P) ->
ChatResult<()>
where
    S: async_std::io::Write + Unpin,
    P: Serialize,
{
    let mut json = serde_json::to_string(&packet)?;
    json.push('\n');
    outbound.write_all(json.as_bytes()).await?;
    Ok(())
}
```

这个函数会将 `packet` 的 JSON 表示形式构建为 `String`，在末尾添加换行符，然后将其全部写入 `outbound`。

从这个函数的 `where` 子句可以看出 `send_as_json` 非常灵活。要发送的数据包类型 `P` 可以是任何实现了 `serde::Serialize` 的值。输出流 `S` 可以是任何实现了 `async_std::io::Write`（输出流的 `std::io::Write` 特型的异步版本）的值。这足以让我们在异步 `TcpStream` 上发送 `FromClient` 值和 `FromServer` 值。只要遵守 `send_as_json` 的泛型定义，就能确保它不会意外依赖于流类型或数据包类型的细节，因为 `send_as_json` 只能使用来自这些特型的方法。

使用 `write_all` 方法需要满足 `S` 上的 `Unpin` 约束。本章在后面会介绍 `Pin` 和 `Unpin`，但就目前而言，只要有必要时向类型变量中添加 `Unpin` 约束就足够了，如果忘记了，Rust 编译器会帮你指出这些问题。

`send_as_json` 没有将数据包直接序列化到 `outbound` 流，而是将其序列化为临时 `String`，然后写入 `outbound` 中。`serde_json` crate 确实提供了将值直接序列化为输出流的函数，但这些函数只支持同步流。要想写入异步流，就要对 `serde_json` 和 `serde` 这两个 crate 中与格式无关的核心代码进行根本性更改，因为围绕它们设计的特型都有一些同步方法。

与流一样，`async_std` 的 I/O 特型的许多方法实际上是在其扩展特型上定义的，因此在使用它们时请务必记住 `use async_std::prelude::*`。

20.2.5 接收数据包：更多异步流

为了接收数据包，我们的服务器和客户端将使用一个来自 `utils` 模块的函数从异步缓冲的 TCP 套接字

(`async_std::io::BufReader<TcpStream>`) 中接收 `FromClient` 值和 `FromServer` 值：

```
use serde::de::DeserializeOwned;

pub fn receive_as_json<S, P>(inbound: S) -> impl Stream<Item =
ChatResult<P>>
    where S: async_std::io::BufRead + Unpin,
          P: DeserializeOwned,
    {
    inbound.lines()
        .map(|line_result| -> ChatResult<P> {
            let line = line_result?;
            let parsed = serde_json::from_str::<P>(&line)?;
            Ok(parsed)
        })
    }
```

与 `send_as_json` 一样，这个函数的输入流类型和数据包类型是泛型的。

- 流类型 `S` 必须实现 `async_std::io::BufRead`，这是 `std::io::BufRead` 的异步模拟，表示缓冲输入字节流。
- 数据包类型 `P` 必须实现 `DeserializeOwned`，这是 `serde` 的 `Deserialize` 特型的更严格变体。为了提高效率，`Deserialize` 可以生成 `&str` 值和 `&[u8]` 值，这些值会直接从反序列化的缓冲区中借用它们的内容，以免复制数据。然而，在上面的例子中，这样做可不太好：我们要将反序列化后的值返回给调用者，因此它们的生命周期必须超出被解析的缓冲区。实

现了 `DeserializeOwned` 的类型始终独立于被反序列化的缓冲区。

调用 `inbound.lines()` 会为我们提供一个携带 `std::io::Result<String>` 值的 `Stream`。然后，使用流的 `map` 适配器对每个条目应用一个闭包，处理错误并将每一行都解析为 `P` 类型值的 JSON 形式。这就生成了一个携带 `ChatResult<P>` 值的流，我们直接将其返回。该函数的返回类型如下所示：

```
impl Stream<Item = ChatResult<P>>
```

这表示我们返回了某种会异步生成 `ChatResult<P>` 值序列的类型，但我们的调用者无法准确判断是哪种类型。由于传给 `map` 的闭包无论如何都是匿名类型，因此这已经是 `receive_as_json` 可能返回的最具体的类型了。

请注意，`receive_as_json` 本身并不是异步函数，它是会返回一个异步值（一个流）的普通函数。现在，比起“只在某些地方添加 `async` 与 `.await`”，你更深入地理解了 Rust 的异步支持机制，能够写出清晰、灵活和高效的定义，就像刚才这个充分发挥出语言特性的定义一样。

要想了解 `receive_as_json` 的用法，可以看看下面这个来自 `src/bin/client.rs` 的聊天客户端的 `handle_replies` 函数，该函数会从网络接收 `FromServer` 的值流并将它们打印出来供用户查看：

```
use async_chat::FromServer;

async fn handle_replies(from_server: net::TcpStream) ->
ChatResult<()> {
    let buffered = io::BufReader::new(from_server);
    let mut reply_stream = utils::receive_as_json(buffered);

    while let Some(reply) = reply_stream.next().await {
        match reply {
            FromServer::Message { group_name, message } => {
                println!("message posted to {}: {}", group_name,
```

```

message);
    }
    FromServer::Error(message) => {
        println!("error from server: {}", message);
    }
}

Ok(())
}

```

这个函数会接受一个从服务器接收数据的套接字，把它包装进 `BufReader`（请注意，这是 `async_std` 版本），然后将其传给 `receive_as_json` 以获取传入的 `FromServer` 值流。接下来它会用 `while let` 循环来处理传入的回复，检查错误结果并打印每个服务器的回复以供用户查看。

20.2.6 客户端的 `main` 函数

介绍完 `send_commands` 和 `handle_replies`，现在可以展示聊天客户端的 `main` 函数了，该函数来自 `src/bin/client.rs`：

```

use async_std::task;

fn main() -> ChatResult<()> {
    let address = std::env::args().nth(1)
        .expect("Usage: client ADDRESS:PORT");

    task::block_on(async {
        let socket = net::TcpStream::connect(address).await?;
        socket.set_nodelay(true)?;

        let to_server = send_commands(socket.clone());
        let from_server = handle_replies(socket);

        from_server.race(to_server).await?;

        Ok(())
    })
}

```

从命令行获取服务器地址后，main 要调用一系列异步函数，因此它会将函数的其余部分都包装在一个异步块中，并将该块返回的 Future 传给 `async_std::task::block_on` 来运行。

建立连接后，我们希望 `send_commands` 函数和 `handle_replies` 函数双线运行，这样就可以在键入的同时看到别人发来的消息。如果遇到了 EOF 指示器或者与服务器的连接断开了，那么程序就应该退出。

考虑到我们在本章其他地方所做的工作，你可能想要写出这样的代码：

```
let to_server = task::spawn(send_commands(socket.clone()));
let from_server = task::spawn(handle_replies(socket));

to_server.await?;
from_server.await?;
```

但由于我们在等待两个 `JoinHandle`，这会让程序在两个任务都完成后才能退出。但我们希望只要任何一个完成就立即退出。Future 上的 `race`（赛跑）方法可以满足这一要求。调用 `from_server.race(to_server)` 会返回一个新的 Future，它会同时轮询 `from_server` 和 `to_server`，并在二者之一就绪时返回 `Poll::Ready(v)`。这两个 Future 必须具有相同的输出类型，其最终值是先完成的那个 Future 的值。未完成的 Future 会被丢弃。

`race` 方法以及许多其他的便捷工具都是在 `async_std::prelude::FutureExt` 特型上定义的，`async_std::prelude` 能让它对我们可见。

迄今为止，我们唯一没有展示过的客户端代码是 `parse_command` 函数。这是一目了然的文本处理代码，所以这里就不展示它的定义了。有关详细信息，请参阅 Git 库中的完整代码。

20.2.7 服务器的 **main** 函数

以下是服务器主文件 `src/bin/server/main.rs` 的全部内容：

```
use async_std::prelude::*;
use async_chat::utils::ChatResult;
use std::sync::Arc;

mod connection;
mod group;
mod group_table;

use connection::serve;

fn main() -> ChatResult<()> {
    let address = std::env::args().nth(1).expect("Usage: server ADDRESS");

    let chat_group_table =
Arc::new(group_table::GroupTable::new());

    async_std::task::block_on(async {
        // 下面这段代码曾在本章的章节介绍中展示过
        use async_std::{net, task};

        let listener = net::TcpListener::bind(address).await?;

        let mut new_connections = listener.incoming();
        while let Some(socket_result) =
new_connections.next().await {
            let socket = socket_result?;
            let groups = chat_group_table.clone();
            task::spawn(async {
                log_error(serve(socket, groups).await);
            });
        }

        Ok(())
    })
}

fn log_error(result: ChatResult<()>) {
    if let Err(error) = result {
        eprintln!("Error: {}", error);
    }
}
```

服务器的 main 函数和客户端的 main 函数类似：它会先进行一些设置，然后调用 `block_on` 来运行一个异步块以完成真正的工作。为了处理来自客户端的传入连接，它创建了 `TcpListener` 套接字，其 `incoming` 方法会返回一个 `std::io::Result<TcpStream>` 值流。

对于每个传入的连接，我们都会启动一个运行 `connection::serve` 函数的异步任务。每个任务还会收到一个 `GroupTable` 值的引用，该值表示服务器的当前聊天组列表，由所有连接通过 `Arc` 引用计数指针共享。

如果 `connection::serve` 返回错误，我们就会将一条消息记录到标准错误并让任务退出，其他连接则照常运行。

20.2.8 处理聊天连接：异步互斥锁

下面这些位于 `src/bin/server/connection.rs` 的 `connection` 模块中的 `serve` 函数是服务器的主要工作代码：

```
use async_chat::{FromClient, FromServer};
use async_chat::utils::{self, ChatResult};
use async_std::prelude::*;
use async_std::io::BufReader;
use async_std::net::TcpStream;
use async_std::sync::Arc;

use crate::group_table::GroupTable;

pub async fn serve(socket: TcpStream, groups: Arc<GroupTable>)
    -> ChatResult<()>
{
    let outbound = Arc::new(Outbound::new(socket.clone()));

    let buffered = BufReader::new(socket);
    let mut from_client = utils::receive_as_json(buffered);
    while let Some(request_result) = from_client.next().await {
        let request = request_result?;

        let result = match request {
            FromClient::Join { group_name } => {
```

```

        let group = groups.get_or_create(group_name);
        group.join(outbound.clone());
        Ok(())
    }

    FromClient::Post { group_name, message } => {
        match groups.get(&group_name) {
            Some(group) => {
                group.post(message);
                Ok(())
            }
            None => {
                Err(format!("Group '{}' does not exist",
group_name))
            }
        }
    };

    if let Err(message) = result {
        let report = FromServer::Error(message);
        outbound.send(report).await?;
    }

    Ok(())
}

```

这几乎就是客户端的 `handle_replies` 函数的镜像：大部分代码是一个循环，用于处理传入的 `FromClient` 值的流，它是从带有 `receive_as_json` 的缓冲 TCP 流构建出来的。如果发生错误，就会生成一个 `FromServer::Error` 数据包，将坏消息传回给客户端。

除了错误消息，客户端还希望接收来自他们已加入的聊天组的消息，因此需要与每个组共享和客户端的连接。虽然可以简单地为每个人提供一份 `TcpStream` 的克隆，但是如果其中两个源试图同时将数据包写入套接字，那么他们的输出就可能彼此交叉，并且客户端最终会收到乱码 JSON。我们需要对此连接安排安全的并发访问。

这是使用 `Outbound` 类型管理的，在 `src/bin/server/connection.rs` 中的定义如下所示：

```

use async_std::sync::Mutex;

pub struct Outbound(Mutex<TcpStream>);

impl Outbound {
    pub fn new(to_client: TcpStream) -> Outbound {
        Outbound(Mutex::new(to_client))
    }

    pub async fn send(&self, packet: FromServer) -> ChatResult<()>
    {
        let mut guard = self.0.lock().await;
        utils::send_as_json(&mut *guard, &packet).await?;
        guard.flush().await?;
        Ok(())
    }
}

```

Outbound 值在创建时会获得 TcpStream 的所有权并将其包装在 Mutex 中以确保一次只有一个任务可以使用它。serve 函数会将每个 Outbound 包装在一个 Arc 引用计数指针中，以便客户端加入的所有组都可以指向同一个共享的 Outbound 实例。

调用 Outbound::send 时会首先锁定互斥锁，返回一个可解引用为内部 TcpStream 的守卫值。我们使用 send_as_json 来传输 packet，最后会调用 guard.flush() 来确保它不会在某处缓冲区进行不完整传输。（据我们所知，TcpStream 实际上并不会缓冲数据，但 Write 特型确实允许它的实现这样做，所以不应该冒这个险。）

表达式 &mut *guard 可以帮我们解决 Rust 不会通过隐式解引用来满足特型限界的问题。我们会显式解引用互斥锁守卫，得到受保护的 TcpStream，然后借用一个可变引用，生成 send_as_json 所需的 &mut TcpStream。

请注意，Outbound 会使用 async_std::sync::Mutex 类型，而不是标准库的 Mutex。原因有以下 3 点。

首先，如果任务在持有互斥锁守卫时被挂起，那么标准库的 `Mutex` 可能会行为诡异。如果一直运行该任务的线程选择了另一个试图锁定同一 `Mutex` 的任务，那么麻烦就会随之而来：从 `Mutex` 的角度来看，已经拥有它的线程正试图再次锁定它。标准的 `Mutex` 不是为处理这种情况而设计的，因此会发生 `panic` 或死锁。（它永远不会以不恰当的方式授予锁。）`Rust` 团队正在进行的一项工作就是在编译期检测到这个问题，并当 `std::sync::Mutex` 守卫运行在 `await` 表达式中时发出警告。由于 `Outbound::send` 在等待 `send_as_json` 和 `guard.flush` 返回的 `Future` 时需要持有锁，因此它必须使用 `async_std` 的 `Mutex`。

其次，异步 `Mutex` 的 `lock` 方法会返回一个守卫的 `Future`，因此正在等待互斥锁的任务会将其线程让给别的任务使用，直到互斥锁就绪。（如果互斥锁已然可用，则此 `lock` 的 `Future` 会立即就绪，任务根本不会自行挂起。）另外，标准库 `Mutex` 的 `lock` 方法在等待获取锁期间会锁定整个线程。由于前面的代码在通过网络传输数据包时持有互斥锁，因此这种等待可能会持续相当长的时间。

最后，标准库 `Mutex` 必须由锁定它的同一个线程解锁。为了强制执行此操作，标准库互斥锁的守卫类型没有实现 `Send`，它不能传输到其他线程。这意味着持有这种守卫的 `Future` 本身不会实现 `Send`，并且不能传给 `spawn` 以在线程池中运行，它只能与 `block_on` 或 `spawn_local` 一起使用。而 `async_std` `Mutex` 的守卫实现了 `Send`，因此在已启动的任务中使用它没有问题。

20.2.9 群组表：同步互斥锁

但前面所讲的那些并不能导向“在异步代码中应该始终使用 `async_std::sync::Mutex`”这样简单的结论。通常在持有互斥锁时不需要等待任何东西，并且这种锁定不会持续太久。在这种情况下，标准库的 `Mutex` 效率会更高。聊天服务器的 `GroupTable` 类型就说明了这种情况。以下是 `src/bin/server/group_table.rs` 的全部内容：

```

use crate::group::Group;
use std::collections::HashMap;
use std::sync::{Arc, Mutex};

pub struct GroupTable(Mutex<HashMap<Arc<String>, Arc<Group>>>);

impl GroupTable {
    pub fn new() -> GroupTable {
        GroupTable(Mutex::new(HashMap::new()))
    }

    pub fn get(&self, name: &String) -> Option<Arc<Group>> {
        self.0.lock()
            .unwrap()
            .get(name)
            .cloned()
    }

    pub fn get_or_create(&self, name: Arc<String>) -> Arc<Group> {
        self.0.lock()
            .unwrap()
            .entry(name.clone())
            .or_insert_with(|| Arc::new(Group::new(name)))
            .clone()
    }
}

```

GroupTable 只是一个受互斥锁保护的哈希表，它会将聊天组名称映射到实际组，两者都使用引用计数指针进行管理。get 方法和 get_or_create 方法会锁定互斥锁，执行一些哈希表操作，可能还会做一些内存分配，然后返回。

在 GroupTable 中，我们会使用普通的旧式 std::sync::Mutex。此模块中根本没有异步代码，因此无须避免 await。事实上，如果想在這裡使用 async_std::sync::Mutex，就要将 get 和 get_or_create 变成异步函数，这会引入 Future 创建、暂停和恢复的开销，但收益甚微：互斥锁只会有一些哈希操作和可能出现的少量内存分配上锁定。

如果聊天服务器发现自己拥有数百万用户，并且 GroupTable 的互斥锁确实成了瓶颈，那么就算把它变成异步形式也无法解决该问题。

使用某种专门用于并发访问的集合类型来代替 `HashMap` 可能会好一些。例如，`dashmap crate` 就提供了这样一个类型。

20.2.10 聊天组：tokio 的广播通道

在我们的服务器中，`group::Group` 类型代表一个聊天组。该类型只需要支持 `connection::serve` 调用的两个方法：`join` 用于添加新成员，`post` 用于发布消息。发布的每条消息都要分发给所有成员。

现在我们来解决前面提过的背压大挑战。有几项需求相互掣肘。

- 如果一个成员无法跟上发布到群组的消息（比如，其网络连接速度较慢），则群组中的其他成员不应受到影响。
- 即使某个成员掉线了，也应该有办法重新加入对话并以某种方式继续参与。
- 用于缓冲消息的内存不应无限制地增长。

因为这些挑战在实现多对多通信模式时很常见，所以 `tokio crate` 提供了一种广播通道类型，可以对这些挑战进行合理的权衡。`tokio` 广播通道是一个值队列（在这个例子中就是聊天消息），它允许任意数量的不同线程或任务发送值和接收值。之所以称为“广播”通道，是因为每个消费者都会获得这里发出的每个值的副本。（这个值的类型必须实现了 `Clone`。）

通常，广播通道会在队列中把一条消息保留到每个消费者都获得了它的副本为止。但是，如果队列的长度超过通道的最大容量（在创建通道时指定），那么最旧的消息将被丢弃。任何掉队的消费者在下次尝试获取下一条消息时都会收到错误消息，并且通道会让他们赶上仍然可用的最旧消息。

例如，图 20-4 展示了一个最大容量为 16 个值的广播通道。

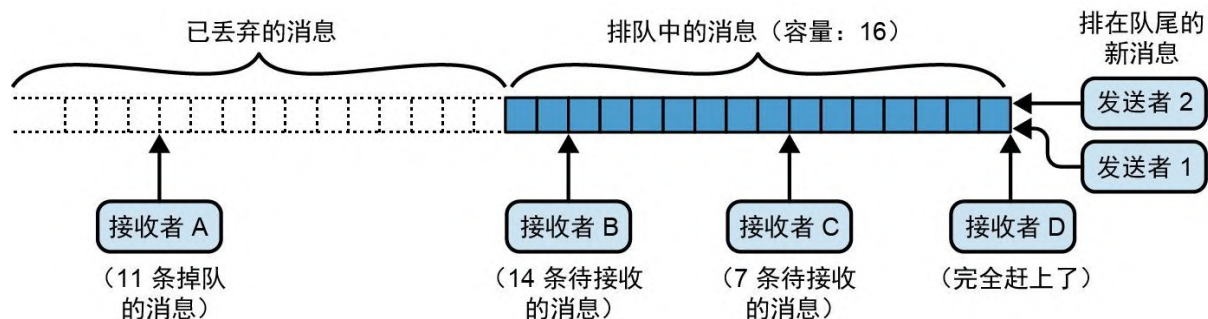


图 20-4: **tokio** 广播通道

有 2 个发送方会将消息排入队列，4 个接收方会将消息从队列中取出——或者更准确地说，是将消息从队列中复制出来。接收者 B 还有 14 条消息要接收，接收者 C 还有 7 条，接收者 D 已经完全赶上了。接收者 A 掉队了，有 11 条消息在它看到之前就被丢弃了。它的下一次接收消息的尝试将失败，然后会返回一个错误以说明情况，并快进到队列的当前尾部。

聊天服务器会将每个聊天组都表示为承载 `Arc<String>` 值的广播通道：向该组发布消息会将消息广播给所有当前成员。下面是 `src/bin/server/group.rs` 中的 `group::Group` 类型的定义：

```
use async_std::task;
use crate::connection::Outbound;
use std::sync::Arc;
use tokio::sync::broadcast;

pub struct Group {
    name: Arc<String>,
    sender: broadcast::Sender<Arc<String>>
}

impl Group {
    pub fn new(name: Arc<String>) -> Group {
        let (sender, _receiver) = broadcast::channel(1000);
        Group { name, sender }
    }

    pub fn join(&self, outbound: Arc<Outbound>) {
        let receiver = self.sender.subscribe();

        task::spawn(handle_subscriber(self.name.clone(),
```

```

                                receiver,
                                outbound));
    }

    pub fn post(&self, message: Arc<String>) {
        // 这只会在没有订阅者时返回错误。连接的发送端可能会退出，并恰好赶在其
        // 接收端回复之前丢弃订阅，这可能会最终导致接收端试图向空组回复消息
        let _ignored = self.sender.send(message);
    }
}

```

Group 结构体中包含聊天组的名称，以及表示组广播通道发送端的 `broadcast::Sender`。

`Group::new` 函数会调用 `broadcast::channel` 创建一个最大容量为 1000 条消息的广播通道。`channel` 函数会返回发送者和接收者，但此时我们不需要接收者，因为组中还没有任何成员。

要向组中添加新成员，`Group::join` 方法会调用发送者的 `subscribe` 方法来为通道创建新的接收者。然后聊天组会在 `handle_subscribe` 函数中启动一个新的异步任务来监视消息的接收者并将它们写回客户端。

有了这些细节，`Group::post` 方法就很简单了：它只是将消息发送到广播通道。由于通道携带的值是 `Arc<String>` 型的值，因此为每个接收者提供自己的消息副本只会增加消息的引用计数，不会进行任何复制或堆分配。一旦所有订阅者都传输了这条消息，引用计数就会降为 0，并且此消息会被释放。

下面是 `handle_subscriber` 的定义：

```

use async_chat::FromServer;
use tokio::sync::broadcast::error::RecvError;

async fn handle_subscriber(group_name: Arc<String>,
                           mut receiver:
broadcast::Receiver<Arc<String>>,
                           outbound: Arc<Outbound>)
{
    loop {
        let packet = match receiver.recv().await {

```

```

        Ok(message) => FromServer::Message {
            group_name: group_name.clone(),
            message: message.clone(),
        },

        Err(RecvError::Lagged(n)) => FromServer::Error(
            format!("Dropped {} messages from {}.", n,
group_name)
        ),

        Err(RecvError::Closed) => break,
    };

    if outbound.send(packet).await.is_err() {
        break;
    }
}
}

```

尽管细节略有不同，但此函数的形式我们很熟悉：它是一个循环，从广播通道接收消息并通过共享的 `Outbound` 值将消息传输回客户端。如果此循环跟不上广播通道，它就会收到一个 `Lagged` 错误，并会尽职尽责地报告给客户端。

如果将数据包发送回客户端时完全失败了，那么可能是因为连接已关闭，`handle_subscriber` 退出其循环并返回，导致异步任务退出。这会丢弃广播通道的 `Receiver`，并取消订阅该通道。这样，当连接断开时，它的每个组成员身份都会在下次该组试图向它发送消息时被清除。

这个聊天组永远不会关闭，因为我们不会从群组表中移除一个组。但为完整性考虑，一旦遇到 `Closed` 错误，`handle_subscriber` 就会退出该任务。

请注意，我们正在为每个客户端的每个组成员创建一个新的异步任务。这之所以可行，是因为异步任务使用的内存要比线程少得多，而且在同一个进程中从一个异步任务切换到另一个异步任务效率非常高。

这就是聊天服务器的完整代码。它有点儿简陋，`async_std crate`、`tokio crate` 和 `futures crate` 中有许多比本书所讲更有价值的特性，但从理论上说，这个扩展示例已经阐明了异步生态系统的一些特性是如何协同工作的：例子中有两种风格的异步任务、流、异步 I/O 特型、通道和互斥锁。

20.3 原始 **Future** 与执行器: **Future** 什么时候值得再次轮询

聊天服务器展示了我们如何使用 `TcpListener`、`broadcast` 通道等异步原语来编写代码，并使用 `block_on`、`spawn` 等执行器来驱动它们的执行。现在来看看这些操作是如何实现的。关键问题是，当一个 `Future` 返回 `Poll::Pending` 时，应该如何与执行器协调，以便在正确的时机再次轮询。

想想当我们从聊天客户端的 `main` 函数运行如下代码时会发生什么：

```
task::block_on(async {
    let socket = net::TcpStream::connect(address).await?;
    ...
})
```

在 `block_on` 第一次轮询异步块的 `Future` 时，几乎可以肯定网络连接没有立即就绪，所以 `block_on` 进入了睡眠状态。那它应该在什么时候醒来呢？一旦网络连接就绪，`TcpStream` 就需要以某种方式告诉 `block_on` 应该再次尝试轮询异步块的 `Future`，因为它知道这一次 `await` 将完成，并且异步块的执行可以向前推进。

当像 `block_on` 这样的执行器轮询 `Future` 时，必须传入一个称为**唤醒器**（`waker`）的回调。如果 `Future` 还没有就绪，那么 `Future` 特型的规则就会要求它必须暂时返回 `Poll::Pending`，并且如果 `Future` 值得再次轮询，就会安排在那时调用唤醒器。

所以 `Future` 的手写实现通常看起来是这样的：

```
use std::task::Waker;

struct MyPrimitiveFuture {
    ...
    waker: Option<Waker>,
}
```



```
impl Future for MyPrimitiveFuture {
    type Output = ...;

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) ->
Poll<...> {
    ...
    if ... future is ready ... {
        return Poll::Ready(final_value);
    }

    // 保存此唤醒器以备后用
    self.waker = Some(cx.waker().clone());
    Poll::Pending
}
}
```

换句话说，如果 Future 的值就绪了，就返回它。否则，将 Context 中唤醒器的克隆体存储在某处，并返回 Poll::Pending。

当 Future 值得再次轮询时，它一定会通过调用其唤醒器的 wake 方法通知最后一个轮询它的执行器：

```
// 如果有一个唤醒器，就调用它，并清除`self.waker`
if let Some(waker) = self.waker.take() {
    waker.wake();
}
```

理论上，执行器和 Future 会轮流轮询和唤醒：执行器会轮询 Future 并进入休眠状态，然后 Future 会调用唤醒器，这样，执行器就会醒来并再次轮询 Future。

异步函数和异步块的 Future 不会处理唤醒器本身，它们只会将自己获得的上下文传给要等待的子 Future，并将保存和调用唤醒器的义务委托给这些子 Future。在我们的聊天客户端中，对异步块返回的 Future 的第一次轮询只会在等待 TcpStream::connect 返回的 Future 时传递上下文（Context）。随后的轮询会同样将自己的上下文传给异步块接下来要等待的任何 Future。

如前面的示例所示，`TcpStream::connect` 返回的 `Future` 会被轮询。也就是说，这些返回的 `Future` 会将唤醒器转移给一个辅助线程，该线程会等待连接就绪，然后调用唤醒器。

`Waker` 实现了 `Clone` 和 `Send`，因此 `Future` 总是可以制作自己的唤醒器副本并根据需要将其发送到其他线程。`Waker::wake` 方法会消耗此唤醒器。还有一个 `wake_by_ref` 方法，该方法不会消耗唤醒器，但某些执行器可以更高效地实现消耗唤醒器的版本。（但这种差异充其量也只是一次 `clone` 而已。）

执行器过度轮询 `Future` 并无害处，只会影响效率。然而，`Future` 应该只在轮询会取得实际进展时才小心地调用唤醒器：虚假唤醒和轮询之间的循环调用可能会阻止执行器完全休眠，从而浪费电量并使处理器对其他任务的响应速度降低。

既然已经展示了执行器和原始 `Future` 是如何通信的，那么接下来我们就自己实现一个原始 `Future`，然后看看 `block_on` 执行器的实现。

20.3.1 调用唤醒器：`spawn_blocking`

本章在前面介绍过 `spawn_blocking` 函数，该函数会启动在另一个线程上运行的给定闭包，并返回携带闭包返回值的 `Future`。现在，我们拥有实现 `spawn_blocking` 所需的所有“零件”。为简单起见，我们的版本会为每个闭包创建一个新线程，而不是像 `async_std` 的版本那样使用线程池。

尽管 `spawn_blocking` 会返回 `Future`，但我们并不会将其写成 `async fn`。相反，它将作为普通的同步函数，返回一个 `SpawnBlocking` 结构体，我们会利用该结构体实现自己的 `Future`。

`spawn_blocking` 的签名如下所示：

```
pub fn spawn_blocking<T, F>(closure: F) -> SpawnBlocking<T>
where F: FnOnce() -> T,
      F: Send + 'static,
      T: Send + 'static,
```

由于需要将闭包发送到另一个线程并带回返回值，因此闭包 `F` 及其返回值 `T` 必须实现 `Send`。由于不知道线程会运行多长时间，因此它们也必须是 `'static` 的。这些限界与 `std::thread::spawn` 自身的强制限界是一样的。

`SpawnBlocking<T>` 是携带闭包返回值的 `Future`。下面是它的定义：

```
use std::sync::{Arc, Mutex};
use std::task::Waker;

pub struct SpawnBlocking<T>(Arc<Mutex<Shared<T>>>);

struct Shared<T> {
    value: Option<T>,
    waker: Option<Waker>,
}
```

`Shared` 结构体必须充当 `Future` 和运行闭包的线程之间的结合点，因此它由 `Arc` 拥有并受 `Mutex` 保护。（同步互斥锁在这里很好用。）轮询此 `Future` 会检查 `value` 是否存在，如果不存在则将唤醒器保存在 `waker` 中。运行闭包的线程会将其返回值保存在 `value` 中，然后调用 `waker`（如果存在的话）。

下面是 `spawn_blocking` 的完整定义：

```
pub fn spawn_blocking<T, F>(closure: F) -> SpawnBlocking<T>
where F: FnOnce() -> T,
      F: Send + 'static,
      T: Send + 'static,
{
    let inner = Arc::new(Mutex::new(Shared {
        value: None,
        waker: None,
    }));
```

```

std::thread::spawn({
    let inner = inner.clone();
    move || {
        let value = closure();

        let maybe_waker = {
            let mut guard = inner.lock().unwrap();
            guard.value = Some(value);
            guard.waker.take()
        };

        if let Some(waker) = maybe_waker {
            waker.wake();
        }
    }
});

SpawnBlocking(inner)
}

```

创建 Shared 值后，就会启动一个线程来运行此闭包，将结果存储在 Shared 的 value 字段中，并调用唤醒器（如果有的话）。

可以为 SpawnBlocking 实现 Future，如下所示：

```

use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll};

impl<T: Send> Future for SpawnBlocking<T> {
    type Output = T;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<T>
    {
        let mut guard = self.0.lock().unwrap();
        if let Some(value) = guard.value.take() {
            return Poll::Ready(value);
        }

        guard.waker = Some(cx.waker().clone());
        Poll::Pending
    }
}

```

轮询 `SpawnBlocking` 来检查闭包的值是否就绪，如果已经就绪，就接手这个值的所有权并返回它。否则，`Future` 仍然处于 `Pending` 状态，因此它在 `Future` 的 `waker` 字段中保存了此上下文中唤醒器的克隆体。

一旦 `Future` 返回了 `Poll::Ready`，就不应该再次对其进行轮询。诸如 `await` 和 `block_on` 之类消耗 `Future` 的常用方式都遵守这条规则。过度轮询 `SpawnBlocking` 型 `Future` 并不会发生什么可怕的事情，因此也不必花费精力来处理这种情况。这就是典型的手写型 `Future`。

20.3.2 实现 `block_on`

除了能够实现原始 `Future`，我们还拥有构建简单执行器所需的全部“零件”。在本节中，我们将编写自己的 `block_on` 版本。它会比 `async_std` 的版本简单很多，比如，它不支持 `spawn_local`、任务局部变量或嵌套调用（从异步代码调用 `block_on`）。但这已足够运行我们的聊天客户端和服务端了。

代码如下所示：

```
use waker_fn::waker_fn;          // Cargo.toml: waker-fn = "1.1"
use futures_lite::pin;           // Cargo.toml: futures-lite = "1.11"
use crossbeam::sync::Parker;     // Cargo.toml: crossbeam = "0.8"
use std::future::Future;
use std::task::{Context, Poll};

fn block_on<F: Future>(future: F) -> F::Output {
    let parker = Parker::new();
    let unparker = parker.unparker().clone();
    let waker = waker_fn(move || unparker.unpark());
    let mut context = Context::from_waker(&waker);

    pin!(future);

    loop {
        match future.as_mut().poll(&mut context) {
            Poll::Ready(value) => return value,
            Poll::Pending => parker.park(),
        }
    }
}
```

```
    }  
  }  
}
```

上述代码虽然很短，但做了很多事，我们慢慢讲。

```
let parker = Parker::new();  
let unparker = parker.unparker().clone();
```

crossbeam crate 的 `Parker` 类型是一个简单的阻塞原语：调用 `parker.park()` 阻塞线程，直到其他人在相应的 `Unparker`（可以通过调用 `parker.unparker()` 预先获得）上调用 `.unpark()`。如果要 `unpark` 一个尚未停泊（`park`）的线程，那么它的下一次 `park` 调用将立即返回，而不会阻塞。这里的 `block_on` 将使用 `Parker` 在 `Future` 未就绪时等待，而我们传给 `Future` 的唤醒器将解除停泊。

```
let waker = waker_fn(move || unparker.unpark());
```

来自 `waker_fn crate` 的 `waker_fn` 函数会从给定的闭包创建一个 `Waker`。在这里，我们制作了一个 `Waker`，当调用它时，它会调用闭包 `move || unparker.unpark()`。还可以通过实现 `std::task::Wake` 特型来创建唤醒器，但这里用 `waker_fn` 更方便一些。

```
pin!(future);
```

给定一个携带 `F` 类型 `Future` 的变量，`pin!` 宏⁴会获取 `Future` 的所有权并声明一个同名的新变量，其类型为 `Pin<&mut F>` 并借入了此 `Future`。这就为我们提供了 `poll` 方法所需的 `Pin<&mut Self>`。异步函数和异步块返回的 `Future` 必须在轮询之前通过 `Pin` 换成引用，20.4 节会对此进行解释。

⁴在 Rust 1.68 及以上版本中，标准库已经内置 `pin!` 宏。——译者注

```
loop {  
  match future.as_mut().poll(&mut context) {  
    Poll::Ready(value) => return value,  
    Poll::Pending => parker.park(),  
  }  
}
```

最后，轮询循环非常简单。以一个携带唤醒器的上下文为入参，我们会轮询 `Future` 直到它返回 `Poll::Ready`。如果返回的是 `Poll::Pending`，我们会暂停此线程，并阻塞到调用了 `waker` 为止。放行后就再重试。

`as_mut` 调用能让我们在不放弃所有权的情况下对 `future` 进行轮询，20.4 节会对此进行详细解释。

20.4 固定 (Pin)⁵

⁵pin 原意为“图钉”，这里指把某个值固定在原位，防止被移动。——译者注

尽管异步函数和异步块对于编写清晰的异步代码至关重要，但处理它们的 Future 时要小心一点儿。Pin 类型有助于确保 Rust 安全地使用它们。

本节首先会展示为什么异步函数调用和异步块的 Future 不能像普通 Rust 值那样随意处理；然后会展示 Pin 如何用作指针的“许可印章”，我们可以依靠这些“盖章指针”来安全地管理此类 Future；最后会展示几种使用 Pin 值的方法。

20.4.1 Future 生命周期的两个阶段

考虑下面这个简单的异步函数：

```
use async_std::io::prelude::*;
use async_std::{io, net};

async fn fetch_string(address: &str) -> io::Result<String> {
    ❶
    let mut socket = net::TcpStream::connect(address).await❷?;
    let mut buf = String::new();
    socket.read_to_string(&mut buf).await❸?;
    Ok(buf)
}
```

这会打开到给定地址的 TCP 连接，并以 String 的形式返回服务器发送的任何内容。标有 ❶、❷ 和 ❸ 的点是恢复点，即异步函数代码中可以暂停执行的点。

假设你调用它，但没有等待，就像下面这样：

```
let response = fetch_string("localhost:6502");
```


现在 `response` 是一个 `Future`，它准备在 `fetch_string` 的开头开始执行，并带有给定的参数。在内存中，`Future` 看起来如图 20-5 所示。

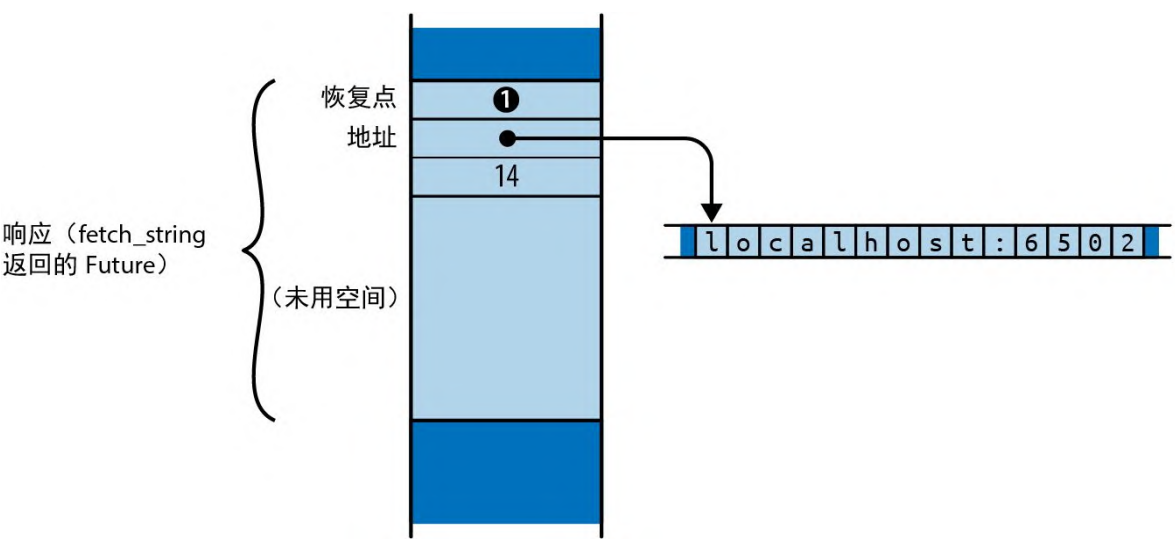


图 20-5：为调用 `fetch_string` 而构建的 `Future`

由于我们刚刚创建了这个 `Future`，因此它认为执行应该从函数体顶部的恢复点 ❶ 开始。在这种状态下，`Future` 唯一能给出的值就是函数参数。

现在假设你对 `response` 进行了几次轮询，并且它在函数体中到达了下面这个点：

```
socket.read_to_string(&mut buf).await❸?;
```

进一步假设 `read_to_string` 的结果尚未就绪，因此轮询会返回 `Poll::Pending`。此时，`Future` 看起来如图 20-6 所示。

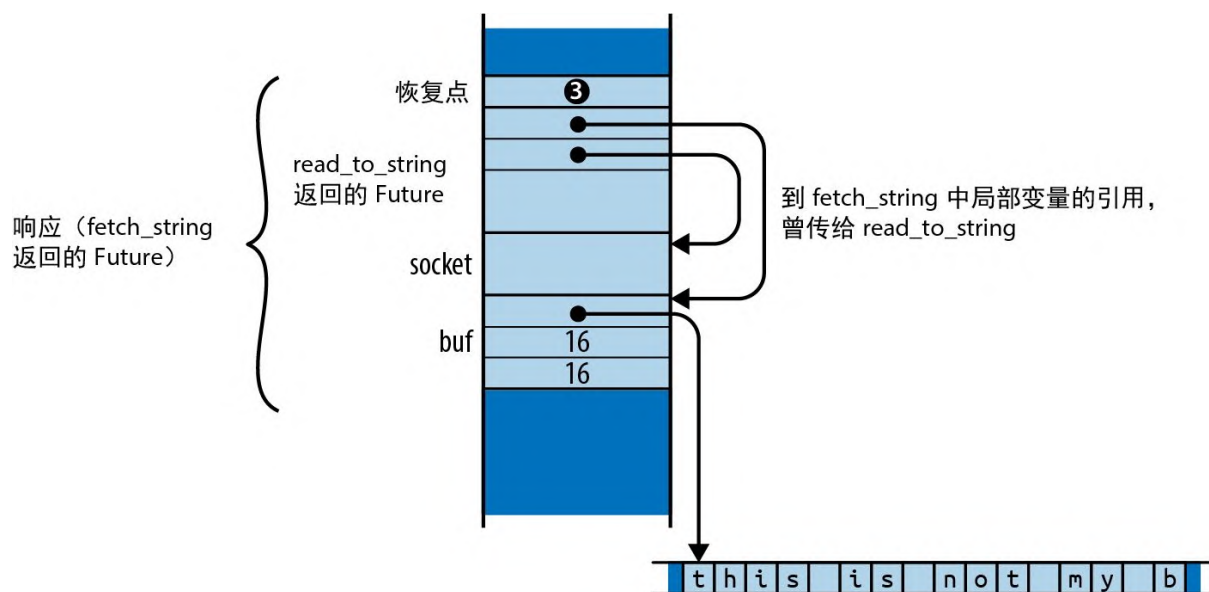


图 20-6: 同一个 **Future**, 正在等待 **read_to_string**

Future 必须始终保存下一次轮询时恢复执行需要的所有信息。在这种情况下是如下内容。

- 恢复点 ③, 表示执行应该在 `await` 处恢复, 那时正在轮询 `read_to_string` 返回的 Future。
- 在那个恢复点处于活动状态的变量: `socket` 和 `buf`。 `address` 的值在 Future 中不会再出现, 因为该函数已不再需要它。
- `read_to_string` 的子 Future, `await` 表达式正在对其进行轮询。

请注意, 对 `read_to_string` 的调用借用了 `socket` 和 `buf` 的引用。在同步函数中, 所有局部变量都存在于栈中, 但在异步函数中, 在 `await` 中仍然存活的局部变量必须位于 Future 中, 这样当再次轮询时它们才是可用的。借入对这样一个变量的引用, 就是借入了 Future 中的一部分。

然而, Rust 要求值在已借出时就不能再移动了。假设要将下面这个 Future 移动到一个新位置:

```
let new_variable = response;
```

Rust 无法找出所有活动引用并相应地调整它们。引用不会指向新位置的 `socket` 和 `buf`，而是继续指向它们在当前处于未初始化状态的 `response` 中的旧位置。它们变成了悬空指针，如图 20-7 所示。

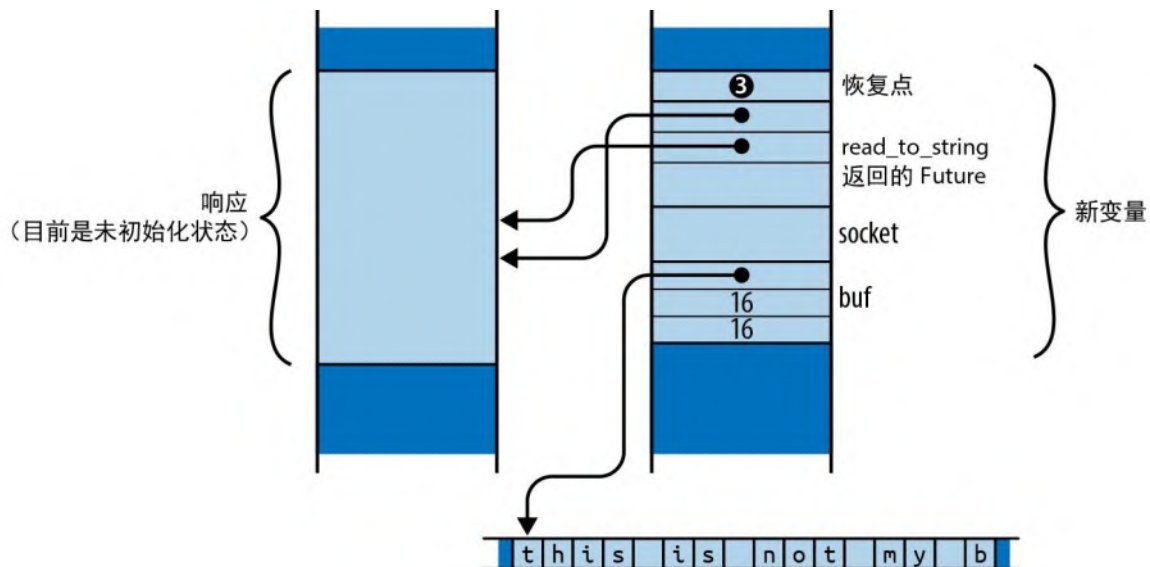


图 20-7: `fetch_string` 返回的 `Future`，在已借出时移动（Rust 会阻止这样做）

防止已借出的值被移动通常是借用检查器的责任。借用检查器会将变量视为所有权树的根。但与存储在栈中的变量不同，如果 `Future` 本身已移动，则存储在 `Future` 中的变量也会移动。这意味着 `socket` 和 `buf` 的借用不仅会影响 `fetch_string` 可以用自己的变量做什么，还会影响其调用者可以安全地用 `response`（也就是持有这些变量的 `Future`）做什么。异步函数的 `Future` 是借用检查器的盲点，如果 Rust 想要保持其内存安全承诺，就必须以某种方式解决这个问题。

Rust 对这个问题的解决方案基于这样一种洞见：`Future` 在首次创建时总是可以安全地移动，只有在轮询时才会变得不安全。在一开始，通过调用异步函数创建的 `Future` 仅包含一个恢复点和参数值。这些仅仅存在于尚未开始执行的异步函数主体的作用域内。只有当轮询 `Future` 时才会借用其内容。

由此可见，每一个 `Future` 的生命周期中都有两个阶段。

- 第一阶段从刚创建 `Future` 时开始。因为函数体还没有开始执行，所以它的任何部分都不可能被借用。在这一点上，移动它和移动其他 `Rust` 值一样安全。
- 第二阶段在第一次轮询 `Future` 时开始。一旦函数的主体开始执行，它就可以借用对存储在 `Future` 中的变量的引用，然后等待，保留对 `Future` 持有的变量的借用。从第一次轮询开始，就必须假设 `Future` 不能被安全地移动了。

第一个生命阶段的灵活性让我们能够将 `Future` 传给 `block_on` 和 `spawn` 并调用适配器方法（如 `race` 和 `fuse`），所有这些都会按值获取 `Future`。事实上，即使最初创建 `Future` 的那次异步函数调用也必须将其返回给调用者，那同样是一次移动。

要进入 `Future` 的第二个生命阶段，就必须对 `Future` 进行轮询。`poll` 方法要求将 `Future` 作为 `Pin<&mut Self>` 值传递。`Pin` 是指针类型（如 `&mut Self`）的包装器，它限制了指针的使用方式，以确保它们的引用目标（如 `Self`）永远不会再次移动。因此，必须首先生成一个指向 `Future` 的以 `Pin` 包装的指针，然后才能对其进行轮询。

这就是 `Rust` 确保 `Future` 安全的策略：`Future` 只有在轮询之前移动才不会有危险，在构建指向 `Future` 的以 `Pin` 包装的指针之前无法轮询 `Future`，一旦这么做了，`Future` 就不可再移动。

“一个无法移动的值”听起来有点儿不可思议，因为在 `Rust` 中移动无处不在。20.4.2 节会详细解释 `Pin` 是如何保护 `Future` 的。

尽管本节讨论的是异步函数，但这里的所有内容也适用于异步块。一个新创建的异步块的 `Future` 只会从它周围的代码中捕获要使用的变量，就像闭包一样。只有轮询 `Future` 时才会创建对其内容的引用，使其移动变得不安全。

请记住，这种移动的脆弱性仅限于异步函数和异步块的 `Future`，以及编译器为它们生成的特殊 `Future` 实现。如果你为自己的类型手动实现了 `Future`，就像我们在 20.3.1 节为 `SpawnBlocking` 类型

所做的那样，那么这样的 `Future` 无论在轮询之前还是之后移动都是完全安全的。在任何手写的 `poll` 实现中，借用检查器会确保当 `poll` 返回时你已借出的任何对 `self` 部分的引用都已消失。正是因为异步函数和异步块有能力在函数调用过程中暂停执行并仍持有借用，所以才必须小心处理它们的 `Future`。

20.4.2 固定指针

`Pin` 类型是指向 `Future` 的指针的包装器，它限制了指针的用法，以确保 `Future` 一旦被轮询就不能移动。这些限制对于不介意被移动的 `Future` 是可以取消的，但对于需要安全地轮询异步函数和异步块的 `Future` 必不可少。

这里的指针指的是任何实现了 `Deref` 或 `DerefMut` 的类型。包裹在指针上的 `Pin` 称为**固定指针**。`Pin<&mut T>` 和 `Pin<Box<T>>` 是典型的固定指针。

标准库中 `Pin` 的定义很简单：

```
pub struct Pin<P> {  
    pointer: P,  
}
```

请注意，`pointer` 字段不是 `pub` 的。这意味着构造或使用 `Pin` 的唯一方法是借助该类型提供的经过精心设计的方法。

给定一个异步函数或异步块返回的 `Future`，只有以下几种方法可以获得指向它的固定指针。

- `pin!` 宏来自 `futures-lite` crate，它会用新的 `Pin<&mut T>` 类型的变量遮蔽 `T` 类型的变量。新变量会指向原始值，而原始值已移至栈中的匿名临时位置。当新变量超出作用域时，原始值会被丢弃。我们用 `pin!` 在 `block_on` 实现中固定了想要轮询的 `Future`。

- 标准库的 `Box::pin` 构造函数能获取任意类型 `T` 值的所有权，将其移动到堆中，并返回 `Pin<Box<T>>`。
- `Pin<Box<T>>` 可以实现 `From<Box<T>>`，因此 `Pin::from(boxed)` 会取得 `boxed` 的所有权，并返回指向堆上同一个 `T` 的固定过的 `Box`。

获得指向这些 `Future` 的固定指针的每一种方法都需要放弃对 `Future` 的所有权，并且无法再取回。当然，固定指针本身可以按照你喜欢的任何方式移动，但移动指针不会移动其引用目标。因此，拥有指向 `Future` 的固定指针足以证明你已经不能移动 `Future` 了。这样就可以安全地对其进行轮询了。这就是我们所要了解的一切。

一旦固定了 `Future`，如果想轮询它，那么所有 `Pin<pointer to T>` 类型都会有一个 `as_mut` 方法，该方法会解引用指针并返回 `poll` 所需的 `Pin<&mut T>`。

`as_mut` 方法还可以帮你在不放弃所有权的情况下轮询 `Future`。`block_on` 的实现中就是出于这个目的而使用它的：

```
pin!(future);

loop {
    match future.as_mut().poll(&mut context) {
        Poll::Ready(value) => return value,
        Poll::Pending => parker.park(),
    }
}
```

在这里，`pin!` 宏已将 `future` 重新声明为 `Pin<&mut F>`，因此可以将其传给 `poll`。但是可变引用不是 `Copy` 类型，因此 `Pin<&mut F>` 也不是 `Copy` 类型，这意味着直接调用 `future.poll()` 将取得 `future` 的所有权，进而导致循环的下一迭代留下未初始化的变量。为了避免这种情况，我们会调用 `future.as_mut()` 为每次循环迭代重新借入新的 `Pin<&mut F>`。

无法获得对已固定 `Future` 的 `&mut` 引用，因为如果可以获得该引用，那么你就能用 `std::mem::replace` 或 `std::mem::swap` 将

其移动出来并在原位置放入另一个 `Future`。

之所以不必担心普通异步代码中的固定 `Future`，是因为获取 `Future` 的最终值的最常见方式（等待它或传给执行器）都要求拥有 `Future` 的所有权并会在内部管理固定指针。例如，我们的 `block_on` 实现会接手 `Future` 的所有权并使用 `pin!` 来生成轮询所需的 `Pin<&mut F>` 的宏。`await` 表达式也会接手 `Future` 的所有权，其内部实现类似于 `pin!` 宏。

20.4.3 Unpin 特型

然而，并不是所有的 `Future` 都需要这样小心翼翼地处理。对于普通类型（如前面提到的 `SpawnBlocking` 类型）的 `Future` 的任何手写实现，这些对构造和使用固定指针方面的限制都是不必要的。

这种耐用类型实现了 `Unpin` 标记特型：

```
trait Unpin { }
```

Rust 中的几乎所有类型都使用编译器中的特殊支持自动实现了 `Unpin`。异步函数和异步块返回的 `Future` 是这条规则的例外情况。

对于各种 `Unpin` 类型，`Pin` 没有任何限制。可以使用 `Pin::new` 从普通指针创建固定指针，然后使用 `Pin::into_inner` 取回该指针。`Pin` 本身会传递指针自己的 `Deref` 实现和 `DerefMut` 实现。

例如，`String` 实现了 `Unpin`，所以可以这样写：

```
let mut string = "Pinned?".to_string();
let mut pinned: Pin<&mut String> = Pin::new(&mut string);

pinned.push_str(" Not");
Pin::into_inner(pinned).push_str(" so much.");

let new_home = string;
assert_eq!(new_home, "Pinned? Not so much.");
```

即使在制作出 `Pin<&mut String>` 之后，仍然可以完全可变地访问字符串，并且一旦这个 `Pin` 被 `into_inner` 消耗，可变引用消失后就可以将其转移给新变量。因此，对 `Unpin` 类型（几乎所有类型）来说，`Pin` 只是指向该类型指针的一个无聊包装器而已。

这意味着当你为自己的 `Unpin` 类型实现 `Future` 时，你的 `poll` 实现可以将 `self` 视为 `&mut Self`，而不是 `Pin<&mut Self>`。`Pin` 成了几乎可以忽略的东西。

令人惊讶的是，即使 `F` 没有实现 `Unpin`，`Pin<&mut F>` 和 `Pin<Box<F>>` 也会实现 `Unpin`。这读起来不太对劲。（`Pin` 怎么可能是 `Unpin` 呢？）但是如果仔细考虑一下每个术语的含义，就能想通了。虽然 `F` 一旦被轮询就不能安全地移动，但指向它的指针总是可以安全地移动，无论其是否被轮询过。不过，只有指针可以移动，其脆弱的引用目标仍然不能。

当你想把一个异步函数或异步块的 `Future` 传给只接受 `Unpin` 类 `Future` 的函数时，知道这一点很重要。（此类函数在 `async_std` 中很少见，但在异步生态中的其他地方并非如此。）无论 `F` 是否实现了 `Unpin`，`Pin<Box<F>>` 都会实现 `Unpin`，因此将 `Box::pin` 应用于异步函数或异步块返回的 `Future` 都会为你提供一个可以在任何地方使用的 `Future`，但代价是要进行堆分配。

使用 `Pin` 时，还有各种不安全的方法，通过这些方法，你可以对指针及其目标做任何想做的事情，甚至对于非 `Unpin` 目标类型也是如此。但是，正如第 22 章所解释的，Rust 无法检查这些方法用的是否正确。因此，你有责任确保用到这些方法的代码的安全性。

20.5 什么时候要用异步代码

异步代码比多线程代码更难写。你必须使用正确的 I/O 和同步原语，手动分解长时间运行的计算或将它们分拆到其他线程上，并管理多线程代码中不会遇到的其他细节，比如固定。那么异步代码到底有哪些优势呢？

下面是你常会听到的两种说法，但它们经不起仔细推敲。

- “异步代码非常适合 I/O。” 这不完全正确。如果应用程序正在花费时间等待 I/O，那么把它变成异步形式并不会让 I/O 运行得更快。如今普遍使用的异步 I/O 接口没有什么比同步接口更高效的地方。对于这两种方式，操作系统会完成同样的工作。（事实上，未就绪的异步 I/O 操作必须稍后重试，因此需要两次系统调用才能完成，而不是一次。）
- “异步代码比多线程代码更容易编写。” 在 JavaScript、Python 等语言中，这很可能是正确的。在这些语言中，程序员使用 `async/await` 作为并发的一种形式：有一个执行线程，并且中断只发生在 `await` 表达式中，因此通常不需要互斥锁来保持数据一致，只是不要在使用它的中途进行 `await`。当只有在你的明确许可下才可能发生任务切换时，代码会更容易理解。

但是这个论点不适用于 Rust，因为在 Rust 中线程用起来几乎不怎么麻烦。一旦程序编译完成，就不会出现数据竞争。非确定性行为仅限于同步特性，比如互斥锁、通道、原子等，它们都是为应对该行为而设计的。因此，异步代码并不能帮你更好地了解其他线程何时会影响你，这在所有安全的 Rust 代码中一目了然。

当然，在与线程结合使用时，Rust 的异步支持真的很出色。如果放弃这种用法实在太可惜了。

那么，异步代码的真正优势是什么呢？

- **异步任务可以使用更少的内存。**在 Linux 上，一个线程的内存使用量至少为 20 KiB，包括用户空间和内核空间的内存使用量。⁶ Future 则小得多：我们的聊天服务器的 Future 只有几百字节大小，并且随着 Rust 编译器的改进还能变得更小。
- **异步任务的创建速度更快。**在 Linux 上，创建一个线程大约需要 15 微秒。而启动一个异步任务大约需要 300 纳秒，仅为创建线程所花费时间的约 1/50。
- **异步任务之间的上下文切换比操作系统线程之间的上下文切换更快。**在 Linux 上这两个操作所需时间分别为 0.2 微秒和 1.7 微秒。⁷然而，这些都是最佳情况下的数值：如果切换是由于 I/O 就绪导致的，则这两个操作的时间都会上升到 1.7 微秒。线程之间的切换和不同处理器核心上的任务之间的切换大不相同：跨核心的通信非常慢。

⁶这包括内核内存，并按为线程分配的物理分页（而非虚拟分页）中尚未分配的数量计算。macOS 和 Windows 上的数量类似。

⁷Linux 上下文切换也曾处于 0.2 微秒范围内，直到其内核由于处理器安全漏洞而被迫使用较慢的技术。

这给了我们一个关于异步代码适合解决哪种问题的提示。例如，异步服务器可能想为每项任务使用更少的内存，以便处理更多的并发连接。（这可能就是异步代码常因“适合 I/O”而享有盛誉的原因。）或者，如果你的设计可以自然地组织成许多相互通信的独立任务，那么每项任务开销低、创建时间短，并且能快速切换上下文都会是重要的优势。这就是为什么聊天服务器是异步编程的经典示例。不过，多人游戏和网络路由器也可能是很好的应用场景。

在其他场景中，要做出是否使用异步编程的决定就不这么显而易见了。如果你的程序有一个线程池来执行繁重的计算或闲置以等待 I/O 完成，那么前面列出的优势可能对其性能影响不大。你必须优化自己的计算，找到更快的网络连接，或者做点儿能实际影响这些限制因素的其他事情。

在实践中，我们能找到的每一个关于实现大容量服务器的说明，都强调了通过测量、调整和不懈努力来识别和消除任务之间产生争用的根

源的重要性。异步架构无法让你跳过这些工作中的任何一项。事实上，虽然很多现成的工具可以评估多线程程序的行为，但这些工具无法识别 Rust 异步任务，因此它们需要自己的工具。（正如一位智者曾经说过的：“现在，你有两个问题了。”）

即使现在不使用异步代码，但如果将来你能有幸比现在忙得多，那么至少了解这个选项的存在也绝对是件好事。

第 21 章 宏

cento（来自拉丁语，意为“拼凑而成”）是一种完全由引自其他诗人的诗句组成的诗。

——Matt Madden

Rust 支持宏。宏是一种扩展语言的方式，它能做到单纯用函数无法做到的一些事。例如，我们已经见过 `assert_eq!` 宏，它是用于测试的好工具：

```
assert_eq!(gcd(6, 10), 2);
```

这也可以写成泛型函数，但是 `assert_eq!` 宏能做到一些无法用函数做到的事。一是当断言失败时，`assert_eq!` 会生成一条错误消息，其中包含断言的文件名和行号。函数无法获取这些信息，而宏可以，因为它们的工作方式完全不同。

宏是一种简写形式。在编译期间，在检查类型并生成任何机器码之前，每个宏调用都会被展开。也就是说，每个宏调用都会被替换成一些 Rust 代码。前面的宏调用展开后大致如下所示：

```
match (&gcd(6, 10), &2) {
    (left_val, right_val) => {
        if !(*left_val == *right_val) {
            panic!("assertion failed: `(left == right)`, \
                    (left: `{:?}`, right: `{:?}`)", left_val,
right_val);
        }
    }
}
```

`panic!` 也是一个宏，它本身可以展开为更多的 Rust 代码（此处未展示）。这些代码使用了另外两个宏，即 `file!()` 和 `line!()`。

一旦 crate 中的每个宏调用都已完全展开，Rust 就会进入下一个编译阶段。

在运行期，断言失败时是这样的（同时指出了 gcd() 函数中存在 bug，因为 2 才是正确答案）：

```
thread 'main' panicked at 'assertion failed: `(left == right)`,  
(left: `17`,  
right: `2`)', gcd.rs:7
```

如果你是 C++ 用户，那么可能对宏有过一些不好的体验。Rust 宏采用了完全不同的设计，类似于 Scheme 的 `syntax-rules`。与 C++ 宏相比，Rust 宏能与语言的其余部分更好地集成，因此更不容易出错。宏调用总是标有感叹号，因此在你阅读代码时很容易发现它们，并且当你想要调用函数时也不会意外调用它们。Rust 宏永远不会插入不匹配的方括号或圆括号，它们天生支持模式匹配，因此编写既可维护又易于使用的宏非常容易。

本章会使用几个简单的示例来展示如何编写宏。与 Rust 的大部分内容一样，深入理解宏会获得回报，因此我们将设计一个更复杂的宏，以将 JSON 字面量直接嵌入程序中。但是，由于本书并不能涵盖宏的所有内容，因此本章将在结尾处提供一些指导，以便你进一步学习，内容包括这里展示的这些工具的高级技术，以及称为过程宏的更强大的机制。

21.1 宏基础

图 21-1 展示了 `assert_eq!` 宏的部分源代码。

```
macro_rules! assert_eq {  
    ($left:expr, $right:expr) => ({  
        match (&$left, &$right) {  
            (left_val, right_val) => {  
                if !(*left_val == *right_val) {  
                    panic!("assertion failed: `(left == right)` \\  
                        (left: `{:?}`, right: `{:?}`)",  
                        left_val, right_val)  
                }  
            }  
        }  
    });  
}
```

The diagram shows the macro definition for `assert_eq!`. The pattern `($left:expr, $right:expr)` is highlighted in yellow and labeled "模式" (pattern). The template `{ ... }` is highlighted in orange and labeled "模板" (template).

图 21-1: `assert_eq!` 宏

`macro_rules!` 是在 Rust 中定义宏的主要方式。请注意，这个宏定义中的 `assert_eq` 之后没有 `!`：只有调用宏时才要用到 `!`，定义宏时不用。

但并非所有的宏都是这样定义的：有一些宏是内置于编译器中的，比如 `file!`、`line!` 和 `macro_rules!`。本章会在结尾处讨论另一种方法，称为过程宏。但在本章的大部分内容里，我们会聚焦于 `macro_rules!`，这是迄今为止编写宏的最简单方式。

使用 `macro_rules!` 定义的宏完全借助“模式匹配”方式发挥作用。宏的主体只是一系列规则：

```
( pattern1 ) => ( template1 );  
( pattern2 ) => ( template2 );
```

...

图 21-1 中的 `assert_eq!` 版本只有一个模式和一个模板。

另外，可以在模式或模板周围随意使用方括号或花括号来代替圆括号，这对 Rust 没有影响。同样，在调用宏时，下面这些都是等效的：

```
assert_eq!(gcd(6, 10), 2);  
assert_eq![gcd(6, 10), 2];  
assert_eq!{gcd(6, 10), 2}
```

唯一的区别是花括号后面的分号通常是可选的。按照惯例，在调用 `assert_eq!` 时使用圆括号，在调用 `vec!` 时使用方括号，而在调用 `macro_rules!` 时使用花括号。

刚才我们展示了一个宏展开的简单例子和用来生成它的宏定义，接下来会深入了解实现这些所需的必要细节。

- 详细解释 Rust 是如何在程序中查找和展开宏定义的。
- 指出从宏模板生成代码的过程中固有的一些微妙细节。
- 展示模式是如何处理重复性结构的。

21.1.1 宏展开的基础

Rust 在编译期间的很早阶段就展开了宏。编译器会从头到尾阅读你的源代码，定义并展开宏。你不能在定义宏之前就调用它，因为 Rust 在查看程序的其余部分之前就已经展开了每个宏调用。（相比之下，函数和其他语法项则不必按任何特定顺序排列。调用一个稍后才会会在其 `crate` 中定义的函数是完全可行的。）

Rust 展开 `assert_eq!` 宏调用的过程与对 `match` 表达式求值很像。Rust 会首先将参数与模式进行匹配，如图 21-2 所示。

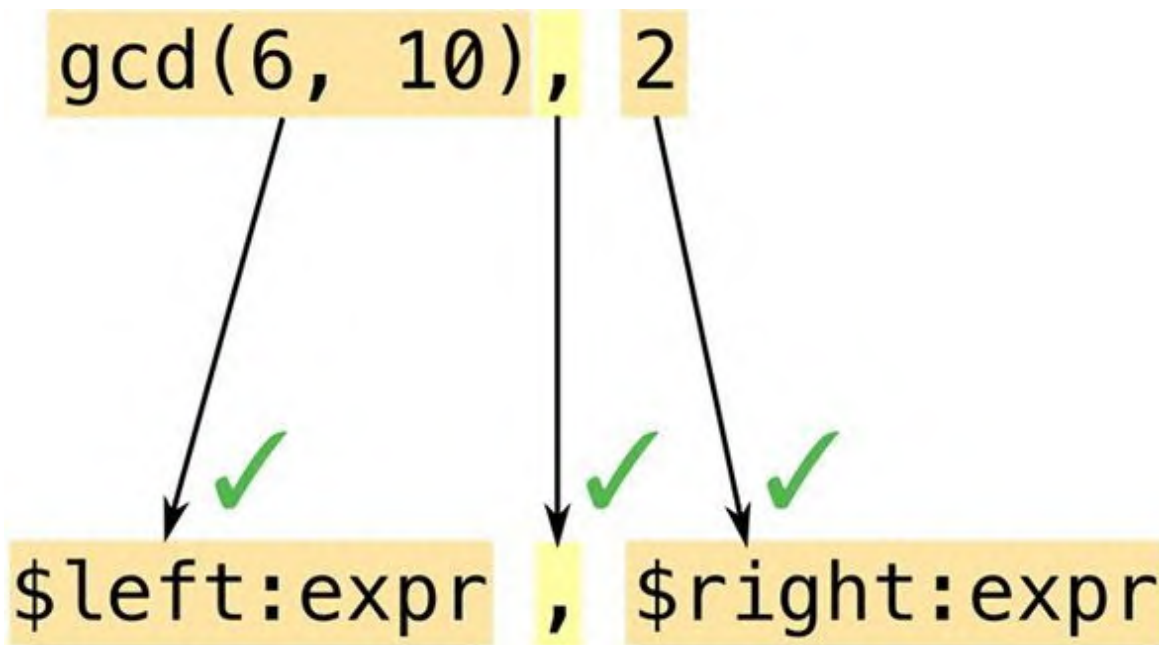


图 21-2：展开宏的第一部分：对参数做模式匹配

宏模式是 Rust 中的一种迷你语言。它们本质上是用来匹配代码的正则表达式。不过正则表达式操作的是字符，而模式操作的是语法标记（Token，包括数值、名称、标点符号等），这些语法标记是 Rust 程序的基础构造块。这意味着可以在宏模式中自由使用注释和空白字符，以尽量提高模式的可读性。因为注释和空白字符不是语法标记，所以不会影响匹配。

正则表达式和宏模式之间的另一个重要区别是圆括号、方括号和花括号在 Rust 中总是成对出现。Rust 会在展开宏之前进行检查，不仅仅在宏模式中检查，而且会贯穿整个语言。

在此示例中，我们的模式包含片段 `$left:expr`，它告诉 Rust 要匹配一个表达式（在本例中是 `gcd(6, 10)`）并将其命名为 `$left`。然后 Rust 会将模式中的逗号与 `gcd` 的参数后面的逗号进行匹配。就像正则表达式一样，模式中只有少数特殊字符会触发有意义的匹配行为；其他字符，比如逗号，则必须逐字匹配，否则匹配就会失败。最后，Rust 会匹配表达式 `2` 并将其命名为 `$right`。

这个模式中的两个代码片段都是 `expr` 类型的，表示它们期待表达式。21.4.1 节会展示其他类型的代码片段。

因为这个模式已经匹配到了所有的参数，所以 Rust 展开了相应的模板，如图 21-3 所示。

```
{
  match (&$left, &$right) {
    (left_val, right_val) => {
      if !(*left_val == *right_val) {
        panic!("assertion failed: `(left == right)` \
          (left: `{:?}`, right: `{:?}`)",
          left_val, right_val)
      }
    }
  }
}
```

替换为 gcd(6, 10)

替换为 2

图 21-3：展开宏的第二部分：填充模板

Rust 会将 `$left` 和 `$right` 替换为它在匹配过程中找到的代码片段。

在输出模板中包含片段类型（比如写成 `$left:expr` 而不仅是 `$left`）是一个常见的错误。Rust 不会立即检测到这种错误。它会将 `$left` 视为替代品，然后将 `:expr` 视为模板中的其他内容——要包含在宏输出中的语法标记。所以宏在被调用之前不会发生错误，然而它将生成实际无法编译的伪输出。如果在使用新宏时收到像 `cannot find type 'expr' in this scope` 和 `help: maybe you meant to use a path separator here` 这样的错误消息，请检查是否存在这种错误。（21.3 节为此类情况提供了更一般化的建议。）

宏模板与 Web 编程中常用的十几种模板语言没有太大差别，唯一的差别（也是重大差别）是它输出的是 Rust 代码。

21.1.2 意外后果

将代码片段插入模板与用来处理值的常规代码略有不同。这些差异起初并不明显。我们一直在讲的宏 `assert_eq!` 就包含一些略显奇怪的代码，其原因大部分和宏编程有关。我们重点看看其中两个比较有意思的部分。

首先，为什么这个宏会创建变量 `left_val` 和 `right_val`？为什么不能将模板简化成下面这样呢？

```
if !($left == $right) {
    panic!("assertion failed: `(left == right)` \
          (left: `{:?}`, right: `{:?}`)", $left, $right)
}
```

要回答这个问题，请尝试在心里展开宏调用 `assert_eq!(letters.pop(), Some('z'))`。它的输出会是什么呢？自然，Rust 会将匹配的表达式插入模板中的多个位置。但是，在构建错误消息时重新计算表达式似乎是个坏主意，不仅仅是因为需要花两倍的时间，更是因为这会导致第二次调用后它的值发生变化（因为 `letters.pop()` 会从向量中移除一个值）。这就是为什么真正的宏只会计算一次 `$left` 和 `$right` 并存储它们的值。

继续第二个问题：为什么这个宏会借用对 `$left` 值和 `$right` 值的引用？为什么不像下面这样将值存储在变量中？

```
macro_rules! bad_assert_eq {
    ($left:expr, $right:expr) => ({
        match ($left, $right) {
            (left_val, right_val) => {
                if !(left_val == right_val) {
                    panic!("assertion failed" /* ... */);
                }
            }
        }
    });
}
```

对于我们一直在考虑的这个特定情况（宏参数是整数），这当然会正常工作。但是，如果调用者将一个 `String` 变量作为 `$left` 或 `$right` 传递，则上述代码会将该值移动出变量。

```
fn main() {
    let s = "a rose".to_string();
    bad_assert_eq!(s, "a rose");
    println!("confirmed: {} is a rose", s); // 错误：使用了已移动出去
    的值 "s"
}
```

我们不希望断言移动值，因此这个宏改成了借入引用。

（你可能想知道为什么这个宏要使用 `match` 而不是 `let` 来定义变量。嗯……我们也想知道。事实证明这样做没有特别的原因。使用 `let` 也可以达到同样的效果。）

简而言之，宏可以做一些令人惊讶的事情。如果你编写的宏周围发生了某些奇怪的事，那么很可能就是宏造成的。

你肯定不会看到下面这个经典的 C++ 宏 bug：

```
// 有bug的C++宏：把数值n加上1
#define ADD_ONE(n) n + 1
```

由于大多数 C++ 程序员很熟悉，因而不值得在这里展开解释的原因，对于像 `ADD_ONE(1) * 10` 或 `ADD_ONE(1 << 4)` 这样不起眼的代码，使用这个宏会产生令人非常吃惊的结果。要修复这个 bug，就要在宏定义中添加更多圆括号。这在 Rust 中是不必要的，因为 Rust 宏能更好地与语言集成。Rust 知道自己什么时候是在处理表达式，因此在将一个表达式粘贴到另一个表达式时能有效地添加合理的圆括号。

21.1.3 重复

标准的 `vec!` 宏有两种形式：

```
// 把一个值重复N次
let buffer = vec![0_u8; 1000];

// 由逗号分隔的值列表
let numbers = vec!["udon", "ramen", "soba"];
```

它可以这样实现：

```
macro_rules! vec {
    ($elem:expr ; $n:expr) => {
        ::std::vec::from_elem($elem, $n)
    };
    ( $( $x:expr ),* ) => {
        <[_]>::into_vec(Box::new([ $( $x ),* ]))
    };
    ( $( $x:expr ),+ , ) => {
        vec![ $( $x ),* ]
    };
}
```

这里有 3 条规则。我们将解释“多规则”宏的工作原理，然后再依次查看每条规则。

Rust 在展开像 `vec![1, 2, 3]` 这样的宏调用时，会先尝试将参数 `1, 2, 3` 与第一条规则的模式相匹配，在本例中就是 `$elem:expr ; $n:expr`。这无法匹配上，因为 `1` 是一个表达式，但模式要求其后面有一个分号，而这里没有。所以 Rust 继续匹配第二条规则，以此类推。如果没有匹配任何规则，则视为错误。

第一条规则处理像 `vec![0u8; 1000]` 这样的用法。碰巧标准库（但未写入文档）函数 `std::vec::from_elem` 完全满足这里的需要，所以这条规则是显而易见的。

第二条规则处理 `vec!["udon", "ramen", "soba"]`。 `$( $x:expr ),*` 模式使用了我们从未见过的一个特性：重复。它会匹配 0 个或多个表达式，以逗号分隔。更一般地说，语法 `$( PATTERN ),*` 可用于匹配任何以逗号分隔的列表，其中列表的每个条目都会匹配 `PATTERN`。

这里的 `*` 与正则表达式中的 `*` 具有相同的含义（“0 或更多”），只是公认的正则表达式中并没有特殊的 `,*` 重复器。还可以使用 `+` 要求至少匹配一次，或者使用 `?` 要求有 0 个或 1 个匹配项。表 21-1 给出了全套的重复模式。

表 21-1：重复模式

模式	含义
<code>\$ (...) *</code>	匹配 0 次或多次，没有分隔符
<code>\$ (...) , *</code>	匹配 0 次或多次，以逗号分隔
<code>\$ (...) ; *</code>	匹配 0 次或多次，以分号分隔
<code>\$ (...) +</code>	匹配 1 次或多次，没有分隔符
<code>\$ (...) , +</code>	匹配 1 次或多次，以逗号分隔
<code>\$ (...) ; +</code>	匹配 1 次或多次，以分号分隔
<code>\$ (...) ?</code>	匹配 0 次或 1 次，没有分隔符

代码片段 `$x` 不是单个表达式，而是一个表达式列表。这条规则的模板也使用了重复语法：

```
<[_]>::into_vec(Box::new([ $( $x ),* ]))
```

同样，有一些标准库方法可以完全满足我们的需要。此代码会创建一个 `Box` 数组，然后使用 `[T]::into_vec` 方法将 `Box` 数组转换为向量。

第一个代码片段 `<[_]>` 是用于编写“某物的切片”类型的一种不寻常的方式，它会期待 `Rust` 推断出元素类型。那些名称为普通标识符的类型可以不经任何修改直接用在表达式中，但是像 `fn()`、`&str` 或 `[_]` 这样的特殊类型必须用尖括号括起来。

重复要出现在模板的末尾，在我们这里是 `$( $x ), *`。这个 `$( ... ), *` 与我们在模式中看到的语法是一样的。它会遍历我们为 `$x` 匹配出的表达式列表，并将它们全部插入模板中，以逗号分隔。

在这个例子中，重复输出看起来和重复输入差不多。但事实并非如此。也可以这样写规则：

```
( $( $x:expr ), * ) => {  
    {  
        let mut v = Vec::new();  
        $( v.push($x); ) *  
        v  
    }  
};
```

在这里，模板中读取 `$( v.push($x); ) *` 的部分会为 `$x` 中的每个表达式插入对 `v.push()` 的调用。宏里的分支可以展开为一系列表达式，但这里只需要一个表达式，所以要把向量的集合包装在一个块中。

与 Rust 的其余部分不同，使用 `$( ... ), *` 的模式不会自动支持可选的尾随逗号。但是，有一个标准技巧，即通过添加额外的规则来支持尾随逗号。这正是 `vec!` 宏的第三条规则的作用：

```
( $( $x:expr ), + , ) => { // 如果出现了尾随逗号，  
    vec! [ $( $x ), * ]    // 就按没有这个逗号时的样子重试  
};
```

我们使用 `$( ... ), + ,` 来匹配带有额外逗号的列表。然后在模板中递归地调用 `vec!`，并剥离额外的逗号。这次会匹配上第二条规则。

21.2 内置宏

Rust 编译器提供了几个内置宏，它们在你定义自己的宏时很有用。这些宏都不能使用 `macro_rules!` 来实现。它们是硬编码在 `rustc` 中的。

`file!()`（文件名）、`line!()`（行号）和 `column!()`（列号）

`file!()` 会展开为字符串字面量，即当前文件名。`line!()` 和 `column!()` 会展开为 `u32` 字面量，以给出当前行号和列号（从 1 开始计数）。

如果一个宏调用了另一个宏，后者又调用了别的宏，并且最后一个宏调用了 `file!()`、`line!()` 或 `column!()`，而它们都在不同的文件中，则最终展开之后的结果指示的是**第一个宏**所在的位置。

`stringify!(...tokens...)`（代码字符串）

该宏会展开为包含给定语法标记的字符串字面量。`assert!` 宏会使用它来生成包含断言代码的错误消息。

它的参数中的宏调用不会展开，比如 `stringify!(line!())` 会展开为字符串 `"line!()"`。

Rust 会从这些语法标记构造出字符串，因此字符串中没有换行符或注释（因为它们都不是语法标记）。

`concat!(str0, str1, ...)`（串联）

该宏会通过串联它的各个参数展开为单个字符串字面量。

Rust 还定义了下面这些用于查询构建环境的宏。

`cfg!(...)`（配置）

该宏会展开为布尔常量，如果当前正构建的配置与圆括号中的条件匹配则为 `true`。如果在启用了调试断言的情况下进行编译，则 `cfg!(debug_assertions)` 为 `true`。

这个宏支持与 8.5 节所讲的 `#[cfg(...)]` 属性完全相同的语法，但其结果不是条件编译，而是给出像 `true` 或 `false` 这样的答案。

`env!("VAR_NAME")`（环境变量）

展开为字符串，即在编译期指定的环境变量的值。如果该变量不存在，则为编译错误。

Cargo 会在编译 `crate` 时设置几个有实质性内容的环境变量，这样此宏才有价值。例如，要获取 `crate` 的当前版本字符串，可以这样写：

```
let version = env!("CARGO_PKG_VERSION");
```

Cargo 文档中有这些环境变量的完整列表。

`option_env!("VAR_NAME")`（可选环境变量）

和 `env!` 基本一样，不过该宏会返回 `Option<&'static str>`，如果没有设置指定的变量，则返回 `None`。

以下 3 个内置宏支持从另一个文件中引入代码或数据。

`include!("file.rs")`（包含代码文件）

该宏会展开为指定文件的内容，这个文件必须是有效的 Rust 代码——表达式或语法项的序列。

`include_str!("file.txt")`（包含字符串）

该宏会展开为包含指定文件中文本的 `&'static str`。可以像这样使用它：


```
const COMPOSITOR_SHADER: &str =
    include_str!("../resources/compositor.glsl");
```

如果文件不存在或不是有效的 UTF-8 格式，你将收到编译错误。

`include_bytes!("file.dat")`（包含一些字节）

和上一个宏基本相同，不过该宏会把文件视为二进制数据而非 UTF-8 文本。结果是 `&'static [u8]`。

与所有宏一样，这些内置宏也都在编译期处理。如果文件不存在或无法读取，则编译失败。它们不会在运行期失败。在任何情况下，如果文件名是相对路径，就会相对于当前文件所在的目录进行解析。

Rust 还提供了几个我们未曾涉及的便捷宏。

`todo!()`（待做）和 `unimplemented!()`（未实现）

这两个宏的作用相当于 `panic!()`，但传达了不同的意图。`unimplemented!()` 适用于 `if` 子句、`match` 分支和其他尚未处理的情况。它总会 `panic`。`todo!()` 大致相同，但传达了这样的想法，即这段代码还没有编写，一些 IDE 会对其进行标记以提请关注。

`matches!(value, pattern)`（匹配）

将值与模式进行比较，如果匹配就返回 `true`，否则返回 `false`。类似于如下写法：

```
match value {
    pattern => true,
    _ => false
}
```

如果你正在寻求编写基本宏的练习，那么该宏是一个很好的复刻目标——特别是你可以直接在标准库文档中找到它的源代码，其实现非常简单。

21.3 调试宏

调试率性而为的宏颇具挑战性。最大的问题是宏展开过程缺乏可见性。Rust 经常会展开所有宏，在发现某种错误后打印一条错误消息，但不会展示包含该错误的完全展开后的代码。

以下是 3 个有助于解决宏问题的工具。（这些特性都还不稳定，但由于它们实际上是为了在开发过程中使用而设计的，而不会出现在要签入的代码中，因此这在实践中并不是什么大问题。）

第一，也是最简单的，可以让 `rustc` 展示代码在展开所有宏后的样子。使用 `cargo build --verbose` 查看 Cargo 如何调用 `rustc`。复制 `rustc` 命令行并添加 `-Z unstable-options --pretty expanded` 选项。完全展开的代码将转储到终端。很遗憾，只有当代码没有语法错误时才能这样做。

第二，Rust 提供了一个 `log_syntax!()` 宏，它只会在编译期将自己的参数打印到终端。你可以将其用于 `println!` 式调试。此宏需要添加 `#![feature(log_syntax)]` 特性标志。

第三，可以要求 Rust 编译器把所有宏调用记录到终端。在代码中的某个地方插入 `trace_macros!(true);`。从那时起，每当 Rust 展开宏时，它都会打印宏的名称和各个参数。例如，考虑下面这个程序：

```
#![feature(trace_macros)]

fn main() {
    trace_macros!(true);
    let numbers = vec![1, 2, 3];
    trace_macros!(false);
    println!("total: {}", numbers.iter().sum::<u64>());
}
```

它会生成这样的输出：

```
$ rustup override set nightly
...
$ rustc trace_example.rs
note: trace_macro
--> trace_example.rs:5:19
  |
5 |         let numbers = vec![1, 2, 3];
  |                               ^^^^^^^^^^^^^^^
  |
= note: expanding `vec! { 1 , 2 , 3 }`
= note: to `< [ _ ] > :: into_vec ( box [ 1 , 2 , 3 ] )`
```

编译器会展示每个宏调用的代码，既包括展开前的也包括展开后的。`trace_macros!(false);` 这一行再次关闭了跟踪，因此不会跟踪对 `println!()` 的调用。

21.4 构建 json! 宏

我们已经讨论过了 `macro_rules!` 的核心特性。本节将逐步开发用于构建 JSON 数据的宏。我们将使用这个例子来展示宏的开发过程，同时会介绍 `macro_rules!` 剩下的几个特性，并就如何确保你的宏如预期般运行提供一些建议。

回想第 10 章，我们介绍过下面这个用于表示 JSON 数据的枚举：

```
#[derive(Clone, PartialEq, Debug)]
enum Json {
    Null,
    Boolean(bool),
    Number(f64),
    String(String),
    Array(Vec<Json>),
    Object(Box<HashMap<String, Json>>)
}
```

遗憾的是，编写 `Json` 值的语法相当冗长：

```
let students = Json::Array(vec![
    Json::Object(Box::new(vec![
        ("name".to_string(), Json::String("Jim
Blandy".to_string())),
        ("class_of".to_string(), Json::Number(1926.0)),
        ("major".to_string(), Json::String("Tibetan throat
singing".to_string()))
    ].into_iter().collect()),
    Json::Object(Box::new(vec![
        ("name".to_string(), Json::String("Jason
Orendorff".to_string())),
        ("class_of".to_string(), Json::Number(1702.0)),
        ("major".to_string(), Json::String("Knots".to_string()))
    ].into_iter().collect())
]);
```

我们希望使用更具 JSON 风格的语法来编写：

```
let students = json!([
    {
```

```
        "name": "Jim Blandy",
        "class_of": 1926,
        "major": "Tibetan throat singing"
    },
    {
        "name": "Jason Orendorff",
        "class_of": 1702,
        "major": "Knots"
    }
];
```

我们想要的是一个将 JSON 值作为参数并展开为 Rust 表达式的 `json!` 宏，就像上面示例中的那样。

21.4.1 片段类型

如果想编写一个复杂的宏，那么第一项工作是弄清楚如何匹配或解析所期望的输入。

我们可以预见 `Json` 宏内部将会有多条规则，因为 JSON 数据有多种类型：对象、数组、数值等。事实上，我们可以合理地猜测每种 JSON 类型都将有一条规则：

```
macro_rules! json {
    (null)      => { Json::Null };
    ([ ... ])  => { Json::Array(...) };
    ({ ... })  => { Json::Object(...) };
    (???)      => { Json::Boolean(...) };
    (???)      => { Json::Number(...) };
    (???)      => { Json::String(...) };
}
```

然而这不太正确，因为宏模式无法区分最后 3 种情况，稍后我们会讨论如何处理。至于前 3 种情况，显然它们是以不同的语法标记开始的，所以我们先从它们开始讨论。

第一条规则已经奏效：

```
macro_rules! json {
    (null) => {
        Json::Null
```

```

    }
}

#[test]
fn json_null() {
    assert_eq!(json!(null), Json::Null); // 通过!
}

```

要添加对 JSON 数组的支持，可以尝试将这些元素匹配为 `expr`:

```

macro_rules! json {
    (null) => {
        Json::Null
    };
    ([ $( $element:expr ),* ]) => {
        Json::Array(vec![ $( $element ),* ])
    };
}

```

很遗憾，这无法匹配所有 JSON 数组。下面是阐明此问题的一个测试：

```

#[test]
fn json_array_with_json_element() {
    let macro_generated_value = json!(
        [
            // 无法匹配`$element:expr`的有效JSON
            {
                "pitch": 440.0
            }
        ]
    );
    let hand_coded_value =
        Json::Array(vec![
            Json::Object(Box::new(vec![
                ("pitch".to_string(), Json::Number(440.0))
            ].into_iter().collect()))
        ]);
    assert_eq!(macro_generated_value, hand_coded_value);
}

```

模式 `$( $element:expr ),*` 表示“以逗号分隔的 Rust 表达式列表”。但是许多 JSON 值，尤其是对象，并不是有效的 Rust 表达式。它们无法匹配。

既然待匹配的每一小段代码并不一定都是表达式，那么 Rust 就肯定要支持另外几种片段类型，如表 21-2 所示。

表 21-2: **macro_rules!** 支持的片段类型

片段类型	匹配（带例子）	后面可以跟……
expr	表达式： 2 + 2、"udon"、x.len()	=>, ;
stmt	表达式或声明，不包括任何尾随分号（很难用，请尝试使用 expr 或 block）	=>, ;
ty	类型： String、Vec<u8>、(&str, bool)、dyn Read + Send	=>, ; = { [: > as where
path	路径（参见 8.2.3 节）： ferns、::std::sync::mpsc	=>, ; = { [: > as where
pat	模式（参见 10.2 节）： _ , Some(ref x)	=>, = if in
item	语法项（参见 6.4 节）： struct Point { x: f64, y: f64 } , mod ferns;	任意
block	块（参见 6.3 节）： { s += "ok\n"; true }	任意
meta	属性的主体（参见 8.5 节）： inline , derive(Copy, Clone) , doc="3D models."	任意

片段类型	匹配（带例子）	后面可以跟……
literal	字面量值： 1024, "Hello, world!", 1_000_000f64	任意
lifetime	生命周期： 'a, 'item, 'static	任意
vis	可见性说明符： pub、pub(crate)、pub(in module::submodule)	任意
ident	标识符： std、Json、longish_variable_name	任意
tt	语法标记树（参见正文）： ;, >= , { } , [0 1 (+ 0 1)]	任意

表 21-2 中的大多数选项会严格执行 Rust 语法。expr 类型只会匹配 Rust 表达式（而不是 JSON 值），ty 只会匹配 Rust 类型，等等。这些选项都不可展开，即无法定义 expr 可以识别的新算术运算符或新关键字。所以无法让它们匹配任意 JSON 数据。

最后两个选项（ident 和 tt）支持匹配看起来不像 Rust 代码的宏参数。ident 能匹配任何标识符。tt 能匹配单个语法标记树：正确匹配的一对括号，比如 (...)、[...] 或 {...}，以及位于两者之间的所有内容，包括嵌套的语法标记树，或者单独的非括号语法标记，比如 1926 或 "Knots"。

看来语法标记树正是我们这个 json! 宏所需要的。每个 JSON 值都是一个语法标记树：数值、字符串、布尔值和 null 是单个语法标记，对象和数组则是有括号的语法标记。所以可以像下面这样写匹配模式：


```
macro_rules! json {
    (null) => {
        Json::Null
    };
    ([ $( $element:tt ),* ]) => {
        Json::Array(...)
    };
    ({ $( $key:tt : $value:tt ),* }) => {
        Json::Object(...)
    };
    ($other:tt) => {
        ... // TODO: 返回Number、String或Boolean
    };
}
```

这个版本的 `json!` 宏可以匹配所有的 JSON 数据。现在只要生成正确的 Rust 代码就好了。

为了使得将来在增加新特性的同时不破坏你今天编写的代码，Rust 限制模式中的语法标记必须出现在片断类型的后面。表 21-2 中的“后面可以跟……”那列展示了哪些语法标记是允许的。例如，模式 `$x:expr ~ $y:expr` 是错误的，因为不允许 `~` 跟在 `expr` 之后。模式 `$vars:pat => $handler:expr` 则是正确的，因为 `$vars:pat` 后面跟着箭头 `=>`，这是 `pat` 允许的语法标记之一，而 `$handler:expr` 后面什么都没有，这总是允许的。

21.4.2 宏中的递归

我们已经看过宏调用自身的一个简单例子：`vec!` 的实现就使用了递归来支持尾随逗号。这里我们会展示一个更重要的例子：`json!` 需要递归调用自己。

我们可能会尝试在不使用递归的情况下支持 JSON 数组，如下所示：

```
([ $( $element:tt ),* ]) => {
    Json::Array(vec![ $( $element ),* ])
};
```

但这是行不通的，因为这样做就会将 JSON 数据（`$element` 语法标记树）直接粘贴到 Rust 表达式中，而它们是两种不同的语言。

因此，我们需要将数组的每个元素从 JSON 格式转换为 Rust。幸运的是，有一个宏可以执行此操作，也就是我们正在写的这个。

```
([ $( $element:tt ),* ]) => {  
    Json::Array(vec![ $( json!($element) ),* ])  
};
```

可以用相同的方式支持对象：

```
(( $( $key:tt : $value:tt ),* )) => {  
    Json::Object(Box::new(vec![  
        $( ($key.to_string(), json!($value)) ),*  
    ].into_iter().collect()))  
};
```

编译器会对宏施加递归限制：默认情况下最多递归 64 层。这对于 `json!` 这样的正常用法足够了，但复杂的递归宏有时会达到这个极限。可以通过在使用宏的 `crate` 顶部添加如下属性来调整它：

```
#![recursion_limit = "256"]
```

我们的 `json!` 宏接近完成了。剩下的就是支持布尔值、数值和字符串值。

21.4.3 将特型与宏一起使用

编写复杂的宏总会给人带来困惑。请务必记住，宏本身并不是你可以使用的唯一解谜工具。

在这里，我们需要支持 `json!(true)`、`json!(1.0)` 和 `json!("yes")`，将值（无论它是什么）转换为适当类型的 `Json` 值。但是宏并不擅长区分类型。可以想象像下面这样写：

```
macro_rules! json {  
    (true) => {
```

```
        Json::Boolean(true)
    };
    (false) => {
        Json::Boolean(false)
    };
    ...
}
```

这种方法马上就会失效。如果只有两个布尔值，这样写当然没问题。但可能还有更多数值，甚至更多字符串，那时候就不能再这样写了。

幸运的是，有一种标准库方法可以将各种类型的值转换为一种指定类型，它就是 `From` 特型（参见 13.9 节）。我们只需要为少数几种类型实现这个特型：

```
impl From<bool> for Json {
    fn from(b: bool) -> Json {
        Json::Boolean(b)
    }
}

impl From<i32> for Json {
    fn from(i: i32) -> Json {
        Json::Number(i as f64)
    }
}

impl From<String> for Json {
    fn from(s: String) -> Json {
        Json::String(s)
    }
}

impl<'a> From<&'a str> for Json {
    fn from(s: &'a str) -> Json {
        Json::String(s.to_string())
    }
}
...
```

事实上，所有 12 种数值类型都有非常相似的实现，所以仅仅为了避免复制粘贴而编写一个宏也是合理的：

```
macro_rules! impl_from_num_for_json {
    ( $( $t:ident )* ) => {
        $(
            impl From<$t> for Json {
                fn from(n: $t) -> Json {
                    Json::Number(n as f64)
                }
            }
        )*
    };
}

impl_from_num_for_json!(u8 i8 u16 i16 u32 i32 u64 i64 u128 i128
                        usize isize f32 f64);
```

现在可以使用 `Json::from(value)` 将任何受支持类型的 `value` 转换为 `Json` 了。在我们的宏中，它看起来是这样的：

```
( $other:tt ) => {
    Json::from($other)    // 处理布尔值、数值和字符串
};
```

将这条规则添加到 `json!` 宏中，让它通过我们迄今已编写的所有测试。将所有部分放在一起，就变成了这样：

```
macro_rules! json {
    (null) => {
        Json::Null
    };
    ([ $( $element:tt ),* ]) => {
        Json::Array(vec![ $( json!($element) ),* ])
    };
    ({ $( $key:tt : $value:tt ),* }) => {
        Json::Object(Box::new(vec![
            $( ($key.to_string(), json!($value)) ),*
        ].into_iter().collect()))
    };
    ( $other:tt ) => {
        Json::from($other)    // 处理布尔值、数值和字符串
    };
}
```

事实证明，宏出乎意料地支持在 JSON 数据中使用变量甚至任意 Rust 表达式，这是一个方便的额外特性：

```
let width = 4.0;
let desc =
    json!({
        "width": width,
        "height": (width * 9.0 / 4.0)
    });
```

因为 `(width * 9.0 / 4.0)` 被圆括号括起来了，所以它是一个单语法标记树，这样宏在解析对象时就能成功地将它与 `$value:tt` 匹配起来。

21.4.4 作用域界定与卫生宏

编写宏时一个非常棘手的问题是需要将来自不同作用域的代码粘贴在一起。所以接下来的内容涵盖了 Rust 处理作用域的两种方式：一种方式用于局部变量和参数，另一种方式用于其他一切。

为了说明为什么这个问题很重要，我们来重写一下解析 JSON 对象的规则（前面展示的 `json!` 宏中的第三条规则）以消除临时向量。可以这样写：

```
(({ $($key:tt : $value:tt),* }) => {
    {
        let mut fields = Box::new(HashMap::new());
        $( fields.insert($key.to_string(), json!($value)); )*
        Json::Object(fields)
    }
});
```

现在不是通过使用 `collect()` 而是通过重复调用 `.insert()` 方法来填充 `HashMap`。这意味着需要将此映射表存储在名为 `fields` 的临时变量中。

但是如果调用 `json!` 时碰巧使用了自己的一个变量，而这个变量也叫 `fields`，会发生什么呢？

```
let fields = "Fields, W.C.";
let role = json!({
    "name": "Larson E. Whipsnade",
    "actor": fields
});
```

展开宏会将两小段代码粘贴在一起，两者都使用 `fields` 这个名字，但表示不同的东西。

```
let fields = "Fields, W.C.";
let role = {
    let mut fields = Box::new(HashMap::new());
    fields.insert("name".to_string(), Json::from("Larson E.
Whipsnade"));
    fields.insert("actor".to_string(), Json::from(fields));
    Json::Object(fields)
};
```

每当宏使用临时变量时，这似乎是一个无法回避的陷阱，你也许已经在考虑可能的修复方法了。也许应该重命名 `json!` 宏，以便定义其调用者不太可能传入的东西，比如可以叫 `__json$fields`。

令人吃惊的是这个宏现在就能正常工作。Rust 会替你重命名此变量。这个特性是首先在 Scheme 语言的宏中实现的，被称为卫生的（hygiene），因此 Rust 被称为支持卫生宏（hygienic macro）的语言。

理解卫生宏的最简单方法是想象每次展开宏时，来自宏本身的展开结果都会被涂上不同的颜色。

然后，不同颜色的变量被视为具有不同的名称：

```
let fields = "Fields, W.C.";
let role = {
    let mut fields = Box::new(HashMap::new());
    fields.insert("name".to_string(), Json::from("Larson E. Whipsnade"));
    fields.insert("actor".to_string(), Json::from(fields));
    Json::Object(fields)
};
```

请注意，由宏调用者传入并粘贴到输出中的那点儿代码（如 "name" 和 "actor"）会保持其原始颜色（黑色）。这里只会对源自宏模板的语法标记进行染色。

现在有一个名为 `fields` 的变量（在调用者中声明）和另一个同样名为 `fields` 的变量（由宏引入）。由于名称是不同的颜色，因此这两个变量不会混淆。

如果宏确实需要引用调用者作用域内的变量，则调用者必须将变量的名称传给宏。

（染色的比喻并不是要准确描述卫生宏的工作原理。真正的机制甚至比这种方式更“聪明一点儿”。只要两个标识符引用的是位于宏及其调用者作用域内的公共变量，不管“染成了什么颜色”，都能识别出它们是相同的。但这种情况在 Rust 中很少见。如果你理解前面的例子，就知道该如何使用卫生宏了。）

你可能已经注意到，随着宏的展开，许多其他标识符（比如 `Box`、`HashMap` 和 `Json`）被染上了不止一种颜色。虽然这些类型名称的颜色不同，但 Rust 仍然毫不费力地识别出了它们。那是因为 Rust 中的卫生工作仅限于局部变量和参数。对于常量、类型、方法、模块、静态值和宏名称，Rust 是“色盲”。

这意味着如果我们的 `json!` 宏在尚未导入 `Box`、`HashMap` 或 `Json` 的模块中使用，那么宏就无法正常工作。21.4.5 节会展示如何避免这一问题。

首先，需要考虑 Rust 的严格卫生机制构成某种障碍的情况，我们要解决这一问题。假设我们有很多包含下面这行代码的函数：

```
let req = ServerRequest::new(server_socket.session());
```

复制和粘贴这行代码很痛苦。可以改用宏吗？

```
macro_rules! setup_req {  
    () => {  
        let req = ServerRequest::new(server_socket.session());  
    }  
}
```

```
    }  
}  
  
fn handle_http_request(server_socket: &ServerSocket) {  
    setup_req!(); // 使用`server_socket`来声明`req`  
    ..... // 使用`req`的代码  
}
```

这样写是不行的。它需要宏中的名称 `server_socket` 来引用函数中声明的局部变量 `server_socket`，对于变量 `req` 也是如此，只不过方向相反。但是卫生机制会防止宏中的名称与其他作用域中的名称“冲突”——不过在这种情况下，你想要的正是变量 `req`。

解决方案是把你打算同时在宏代码内部和外部使用的任何标识符都传给宏：

```
macro_rules! setup_req {  
    ($req:ident, $server_socket:ident) => {  
        let $req = ServerRequest::new($server_socket.session());  
    }  
}  
  
fn handle_http_request(server_socket: &ServerSocket) {  
    setup_req!(req, server_socket);  
    ..... // 使用`req`的代码  
}
```

由于函数现在提供了 `req` 和 `server_socket`，因此它们在该作用域内就有正确的“颜色”了。

卫生机制让这个宏使用起来有点儿冗长，但这是一个特性，而不是 bug：了解到卫生宏不会背着你干扰局部变量，就更容易理解它。如果你在函数中搜索像 `server_socket` 这样的标识符，就会找到使用过它的所有地方，包括宏调用。

21.4.5 导入宏和导出宏

由于宏会在编译早期展开，那时候 Rust 甚至都不知道项目的完整模块结构，因此编译器需要对宏的导入和导出进行特殊支持。

在一个模块中可见的宏也会自动在其子模块中可见。要将宏从当前模块“向上”导出到其父模块，请使用 `#[macro_use]` 属性。假设我们的 `lib.rs` 是这样的：

```
#[macro_use] mod macros;
mod client;
mod server;
```

那么 `macros` 模块中定义的所有宏都会导入 `lib.rs` 中，因此在 `crate` 的其余部分（包括在 `client` 和 `server` 中）都可见。

标有 `#[macro_export]` 的宏会自动成为公共的，并且可以像其他语法项一样通过路径引用。

例如，`lazy_static` crate 提供了一个名为 `lazy_static` 的宏，该宏会被标记为 `#[macro_export]`。要在自己的 `crate` 中使用这个宏，可以这样写：

```
use lazy_static::lazy_static;
lazy_static!{ }
```

一旦导入了宏，就可以像其他任何语法项一样使用它：

```
use lazy_static::lazy_static;

mod m {
    crate::lazy_static! { }
```

当然，实际上做这些就意味着你的宏可能会在其他模块中调用。因此，导出的宏不应该依赖作用域内的任何内容，因为在使用时无从确定它的作用域内会有哪些内容。它甚至可以遮蔽标准库预导入中的特性。

相反，宏应该对它用到的任何名称都使用绝对路径。`macro_rules!` 提供了特殊片段 `$crate` 来帮助解决这个问题。这与 `crate` 不同，`crate` 关键字可以用在任何地方的路径中而不仅仅是宏中。`$crate` 相当于定义此宏的 `crate` 的根模块绝对路径。我们可以写成

`$crate::Json` 而非 `Json` 的形式，这样即使没有导入 `Json` 也能工作。`HashMap` 可以被更改为 `::std::collections::HashMap` 或 `$crate::macros::HashMap`。在后一种情况下，我们将不得不重新导出 `HashMap`，因为 `$crate` 不能用于访问 `crate` 的私有特性。它实际上只是展开成类似于 `::jsonlib` 的普通路径。可见性规则不受影响。

将宏移到它自己的模块 `macros` 并修改为使用 `$crate` 后，代码是下面这样的。这是最终版本：

```
// macros.rs
pub use std::collections::HashMap;
pub use std::boxed::Box;
pub use std::string::ToString;

#[macro_export]
macro_rules! json {
    (null) => {
        $crate::Json::Null
    };
    ([ $( $element:tt ),* ]) => {
        $crate::Json::Array(vec![ $( json!($element) ),* ])
    };
    ({ $( $key:tt : $value:tt ),* }) => {
        {
            let mut fields = $crate::macros::Box::new(
                $crate::macros::HashMap::new());
            $(
                fields.insert($crate::macros::ToString::to_string($key),
                    json!($value));
            )*
            $crate::Json::Object(fields)
        }
    };
    ($other:tt) => {
        $crate::Json::from($other)
    };
}
```

由于 `.to_string()` 方法是标准 `ToString` 特型的一部分，因此我们也会通过 `$crate` 来引用它，而使用的语法是 11.3 节介绍的

`$crate::macros::ToString::to_string($key)`。在这个例子中，这一句并不是必要的，因为 `ToString` 位于标准库预导入中。但是，如果你调用的特型方法可能不在调用宏的作用域内，则使用完全限定的方法调用是最好的办法。

21.5 在匹配过程中避免语法错误

下面的宏看起来很合理，但它给 Rust 带来了一些麻烦：

```
macro_rules! complain {
    ($msg:expr) => {
        println!("Complaint filed: {}", $msg)
    };
    (user : $userid:tt , $msg:expr) => {
        println!("Complaint from user {}: {}", $userid, $msg)
    };
}
```

假设我们这样调用它：

```
complain!(user: "jimb", "the AI lab's chatbots keep picking on
me");
```

在人类眼中，这显然符合第二条规则。但是 Rust 会首先尝试第一条规则，试图将所有输入与 `$msg:expr` 匹配。事情开始变得棘手了。`user: "jimb"` 当然不是表达式，所以我们得到了一个语法错误。Rust 拒绝隐藏语法错误，因为调试宏本来就已经够艰难了，再隐藏语法错误简直要命。因此，它会立即报告并停止编译。

如果无法匹配模式中的任何其他语法标记，Rust 就会继续执行下一条规则。只有语法错误才会导致匹配失败，并且只在试图匹配片段时才会发生。

这里的问题并不难理解：我们是在错误的规则中试图匹配片段 `$msg:expr`。这无法匹配，因为我们本来就不应该在这里匹配。调用者想要匹配其他规则。有两种简单的方法可以避免这种情况。

第一种方法，避免混淆各条规则。例如，可以更改宏，让每个模式都以不同的标识符开头：

```
macro_rules! complain {
    (msg : $msg:expr) => {
```

```
println!("Complaint filed: {}", $msg);  
};  
(user : $userid:tt , msg : $msg:expr) => {  
    println!("Complaint from user {}: {}", $userid, $msg);  
};  
}
```

当宏参数以 `msg` 开头时，就匹配第一条规则。当宏参数以 `user` 开头时，就匹配第二条规则。无论是哪种参数，我们都能知道在尝试匹配片段之前已经找到了正确的规则。

避免虚假语法错误的另一种方法是将更具体的规则放在前面。将 `user:` 规则放在前面就可以解决 `complain!` 的问题，因为永远不会到达导致语法错误的那条规则。

21.6 超越 `macro_rules!`

宏模式固然可以解析比 JSON 更复杂的输入，但我们也发现这种复杂性很快就会失控。

Daniel Keep 等人撰写的 *The Little Book of Rust Macros* 是一本优秀的高级 `macro_rules!` 编程书。该书写得清晰明了，关于宏展开的各个方面都比本章讲得更详尽。另外，该书还提供了几种非常聪明的技巧来借助 `macro_rules!` 模式实现某种玄奥的编程语言，以用这种模式解析复杂的输入。不过我们对这些技巧持保留态度。请小心使用。

Rust 1.15 引入了一种称为过程宏的独立机制。过程宏不仅支持扩展 `#[derive]` 属性以处理自定义派生（参见图 21-4），还支持创建自定义属性和像前面讨论过的 `macro_rules!` 这样的宏。

```
#[derive(Copy, Clone, PartialEq, Eq, IntoJson)]
struct Money {
    dollars: u32,
    cents: u16,
}
```

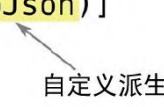


图 21-4：通过 `#[derive]` 属性调用假设的 `IntoJson` 过程宏

没有 `IntoJson` 特型，但这并不重要：过程宏可以利用这个钩子插入它想要的任何代码（在这个例子中，可能是 `impl From<Money> for Json { ... }`）。

让过程宏得名“过程”的原因在于它是作为 Rust 函数而不是声明性规则集实现的。这个函数会通过一个很薄的抽象层与编译器交互，进而实现任意复杂的功能。例如，`diesel` 数据库 `crate` 就会使用过程宏连接到数据库并在编译期根据该数据库的模式（`schema`）生成代码。

因为过程宏要与编译器内部交互，所以要写出有效的宏就要了解编译器是如何运行的，这超出了本书的范畴。有关详细信息，可以查阅 Rust 在线文档。

也许在阅读了所有这些内容后，你开始有点儿讨厌宏了。那么，还有什么其他选择吗？还有一种方法是使用构建脚本生成 Rust 代码。Cargo 文档展示了如何一步一步地做到这一点。它涉及编写一个生成所需 Rust 代码的程序，然后向 Cargo.toml 中添加一行，以便在构建过程中运行该程序并使用 `include!` 将生成的代码放入你的 crate 中。

第 22 章 不安全代码

希望没有人认为我卑微、软弱或顺从，

希望他们明白我与众不同：

对于敌人，我意味着危险；对于朋友，我意味着忠诚。

这便是我荣耀的人生。

——《美狄亚》，Euripides

系统编程的隐秘乐趣在于，在每一种安全语言和精心设计的抽象之下，都是极度不安全的机器语言和按位操作的汹涌暗流。你也可以用 Rust 写出这种代码。

迄今为止，通过类型检查、生命周期检查、限界检查等方法，本书中介绍的这门语言可以确保你的程序完全自动地摆脱内存错误和数据竞争的困扰。但是这种自动化推理有其局限性，因为 Rust 中仍然有许多无法识别为安全的高价值技术。

不安全¹代码能让你告诉 Rust：“我选择使用你无法保证安全的特性。”通过将块或函数标记为不安全的，你可以获得调用标准库中的 `unsafe` 函数、解引用不安全指针以及调用以其他语言（如 C 和 C++）编写的函数等能力。Rust 的其他安全检查仍然适用：类型检查、生命周期检查和索引的边界检查都会正常进行。不安全代码只会启用一小部分附加特性。

¹ 此处的不安全（unsafe）并不表示一定有风险，只是说 Rust 编译器无法保证安全。unsafe 的程度相比中文的“不安全”要弱得多。但这是社区普遍采纳的一个翻译，因此本书跟随这种译法。——译者注

这种跨越 Rust 安全边界的能力使得 Rust 可以实现自身许多最基本的特性，就像 C 和 C++ 被用于实现自己的标准库一样。使用不安全

代码，`Vec` 类型可以更加高效地管理其缓冲区，`std::io` 模块可以和操作系统对话，`std::thread` 模块和 `std::sync` 模块可以提供并发原语。

本章涵盖了使用不安全特性的所有要点。

- Rust 的 `unsafe` 块在普通的、安全的 Rust 代码和使用了不安全特性的代码之间建立了边界。
- 可以将函数标记为 `unsafe`，提醒调用者这里存在必须遵守的额外契约，以避免未定义行为。
- 裸指针及其方法允许不受限制地访问内存，进而构建 Rust 的类型系统原本会禁止的数据结构。Rust 的引用是安全但受限的，而任何 C 或 C++ 程序员都知道，裸指针是一个强大而锋利的工具。
- 理解未定义行为的定义将帮助你理解为什么它会产生比得到不正确的结果还要严重的后果。
- 不安全特型（`unsafe trait`）与 `unsafe` 函数类似，对每个实现而不是每个调用者都强加了必须遵守的契约。

22.1 不安全因素来自哪里

在本书的开头，我们展示过一个因为没有遵守 C 标准规定中的规则而以令人惊讶的方式崩溃的 C 程序。在 Rust 中可以做到同样的事情：

```
$ cat crash.rs
fn main() {
    let mut a: usize = 0;
    let ptr = &mut a as *mut usize;
    unsafe {
        *ptr.offset(3) = 0x7ffff72f484c;
    }
}
$ cargo build
   Compiling unsafe-samples v0.1.0
   Finished debug [unoptimized + debuginfo] target(s) in 0.44s
$ ../../target/debug/crash
crash: Error: .netrc file is readable by others.
crash: Remove password or make file unreadable by others.
Segmentation fault (core dumped)
$
```

这个程序借用了对局部变量 `a` 的可变引用，将其转换为 `*mut usize` 类型的裸指针，然后使用 `offset` 方法在内存中又生成了 3 个字的指针。这恰好是存储 `main` 的返回地址的地方。这个程序用一个常量覆盖了返回地址，这样从 `main` 返回的行为就会令人非常惊讶。导致这次崩溃的原因是程序错误地使用了不安全特性——在这个例子中就是解引用裸指针的能力。

不安全特性是强加了某种**契约**的特性：Rust 不能自动执行这些规则，但你必须遵守这些规则以避免未定义行为。

这种契约超出了常规类型检查和生命周期检查的能力范围，针对该不安全特性强加了更多规则。通常，Rust 本身根本不了解契约，契约只是在该特性的文档中进行了解释。例如，裸指针类型有一个契约，它禁止解引用已超出其原始引用目标末尾的指针。上述例子中的表达式 `*ptr.offset(3) = ...` 破坏了这个契约。但是，正如前面的记录所示，Rust 毫无怨言地编译了这段程序，因为它的安全检查并未检测

到这种违规行为。当使用了不安全特性时，作为程序员，你有责任检查自己的代码是否遵守了它们的契约。

许多特性需要遵守某些规则才能正确使用，但这些规则并不是这里所说的契约，除非违反它们的后果包括未定义行为。未定义行为是

“Rust 坚定地认为你的代码永远不会出现的行为”。例如，Rust 认为你不会用其他内容覆盖函数调用的返回地址。能够通过 Rust 通常的安全检查并遵守其用到的不安全特性的契约的代码不可能做这样的事情。由于前面的程序违反了裸指针契约，因此其行为是未定义的，它已经偏离了轨道。

如果代码表现出未定义行为，那你就已经违背了与 Rust 达成的交易，所以 Rust 无法对其后果负责。从系统库深处挖掘出不相关的错误消息并导致崩溃是一种可能的后果，将计算机的控制权交给攻击者是另一种后果。在没有警告的情况下，从 Rust 的一个版本换到下一个版本可能会产生不同的效果。然而，有时未定义行为并没有明显的后果。如果 `main` 函数永远不会返回（比如调用了 `std::process::exit` 来提前终止程序），那么损坏的返回地址可能无关紧要。

只能在 `unsafe` 块或 `unsafe` 函数中使用不安全特性，我们将在接下来的内容中对两者进行解释。这可以避免在不知觉中使用不安全特性：通过强制编写 `unsafe` 块或函数，Rust 会确保你已经知道在自己的代码中可能要遵守的额外规则。

22.2 不安全块

`unsafe` 块看起来就像前面加了 `unsafe` 关键字的普通 Rust 块，不同之处在于可以在块中使用不安全特性：

```
unsafe {  
    String::from_utf8_unchecked(ascii)  
}
```

如果块前面没有 `unsafe` 关键字，那么 Rust 就会反对使用 `from_utf8_unchecked`，因为这是一个 `unsafe` 函数。有了它周围的 `unsafe` 块，就可以在任何地方使用此代码了。

与普通的 Rust 块一样，`unsafe` 块的值就是其最终表达式的值，如果没有则为 `()`。前面展示的对 `String::from_utf8_unchecked` 的调用提供了该块的值。

`unsafe` 块解锁了 5 个额外的选项。

- 可以调用 `unsafe` 函数。每个 `unsafe` 函数都必须根据自己的目的指定自己的契约。
- 可以解引用裸指针。安全代码可以传递裸指针，比较它们，并从引用（甚至整数）转换成它们，但只有不安全代码才能真正使用它们来访问内存。22.8 节将详细介绍裸指针并解释如何安全地使用它们。
- 可以访问 `union` 的各个字段，编译器无法确定这些字段是否包含其各自类型的有效位模式。
- 可以访问可变的 `static` 变量。如 19.3.11 节所述，Rust 无法确定线程何时使用可变 `static` 变量，因此它们的契约要求你确保所有访问都能正确同步。
- 可以访问通过 Rust 的外部函数接口声明的函数和变量。即使声明为不可变的，这些函数和变量也仍然会被看作 `unsafe` 的，因为它们对于用其他可能不遵守 Rust 安全规则的语言编写的代码仍然是可见的。

将不安全特性限制在 `unsafe` 块中并不能真正阻止你做任何想做的事。你完全可以只将一个 `unsafe` 块粘贴到代码中，然后继续我行我素。该规则的主要目的在于将人们的视线吸引到 Rust 无法保证其安全性的代码上。

- 你不会无意中使用不安全特性，然后发现要对连自己都不知道在哪里的契约负责。
- `unsafe` 块会引起评审者的更多关注。有些项目甚至会通过自动化设施来确保这一点，它们会标记出影响 `unsafe` 块的代码更改以引起特别关注。
- 当你考虑编写 `unsafe` 块时，可以花点儿时间问问自己是否真的需要这样的措施。如果是为了性能，那是否有测量结果表明这确实是一个瓶颈呢？也许在安全的 Rust 中有更好的办法来完成同样的事情。

22.3 示例：高效的 ASCII 字符串类型

下面是 `Ascii` 的定义，它是一种能确保其内容始终为有效 ASCII 的字符串类型。这种类型使用了不安全特性来提供到 `String` 的零成本转换：

```
mod my_ascii {
    /// 一个ASCII编码的字符串
    #[derive(Debug, Eq, PartialEq)]
    pub struct Ascii(
        // 必须只持有格式良好的ASCII文本：字节范围从`0`到`0x7f`
        Vec<u8>
    );

    impl Ascii {
        /// 从`bytes`的ASCII文本中创建`Ascii`。如果`bytes`包含
        /// 任何非ASCII字符，则返回`NotAsciiError`错误
        pub fn from_bytes(bytes: Vec<u8>) -> Result<Ascii,
NotAsciiError> {
            if bytes.iter().any(|&byte| !byte.is_ascii()) {
                return Err(NotAsciiError(bytes));
            }
            Ok(Ascii(bytes))
        }
    }

    // 当转换失败时，给出无法转换的向量。这会实现
    // `std::error::Error`，为保持简洁已省略
    #[derive(Debug, Eq, PartialEq)]
    pub struct NotAsciiError(pub Vec<u8>);

    // 使用不安全代码实现的安全、高效的转换
    impl From<Ascii> for String {
        fn from(ascii: Ascii) -> String {
            // 如果此模块没有bug，这就是安全的，因为格式
            // 良好的ASCII文本必然是格式良好的UTF-8
            unsafe { String::from_utf8_unchecked(ascii.0) }
        }
    }

    ...
}
```

这个模块的关键是 `Ascii` 类型的定义。该类型本身是被标记为 `pub` 的，以令其在 `my_ascii` 模块之外可见。但是该类型的 `Vec<u8>` 元素不是公共的，因此只有 `my_ascii` 模块可以构造 `Ascii` 值或引用其元素。这使得模块的代码可以完全控制允许出现或不允许出现的内容。只要公共构造函数和方法能确保新创建的 `Ascii` 值是格式良好的并在其整个生命周期中都是如此，程序的其余部分就不会违反该规则。事实上，公共构造函数 `Ascii::from_bytes` 在同意从给定的向量中构造 `Ascii` 之前会仔细检查它。为简洁起见，我们没有展示任何方法，但你可以想象有一组文本处理方法，并确保 `Ascii` 值始终包含正确的 ASCII 文本，就像 `String` 的方法会确保其内容始终是格式良好的 UTF-8 一样。

这种安排让我们可以非常高效地为 `String` 实现 `From<Ascii>`。不安全函数 `String::from_utf8_unchecked` 会获取字节向量并从中构建一个 `String`，而不会检查其内容是否为格式良好的 UTF-8 文本，该函数的契约要求其调用者对此负责。幸运的是，`Ascii` 类型强制执行的规则正是应该满足 `from_utf8_unchecked` 契约的规则。正如 17.2 节所解释的那样，任何 ASCII 文本块也是格式良好的 UTF-8，因此 `Ascii` 的底层 `Vec<u8>` 可以立即用作 `String` 的缓冲区。

有了这些定义，便可以这样写：

```
use my_ascii::Ascii;

let bytes: Vec<u8> = b"ASCII and ye shall receive".to_vec();

// 这个调用不需要分配内存或复制文本，只需做扫描
let ascii: Ascii = Ascii::from_bytes(bytes)
    .unwrap(); // 我们知道所选的这些字节肯定是正确的

// 这个调用是零开销的：无须分配内存、复制文本或扫描
let string = String::from(ascii);

assert_eq!(string, "ASCII and ye shall receive");
```

使用 `Ascii` 时不需要 `unsafe` 块。我们已经使用不安全操作实现了一个安全接口，并准备好仅依赖模块自己的代码而不必靠其用户的行为来满足它们的契约。

`Ascii` 只不过是 `Vec<u8>` 的包装器，但隐藏在对其内容实施额外规则的模块中。这种类型称为 `newtype`，这是 `Rust` 中的一种常见模式。`Rust` 自己的 `String` 类型以完全相同的方式定义，不过它的内容被限制为 UTF-8，而不是 ASCII。事实上，标准库中 `String` 的定义是这样的：

```
pub struct String {  
    vec: Vec<u8>,  
}
```

在机器层面，由于不认识 `Rust` 的类型，`newtype` 及其元素在内存中具有相同的表示，因此构造 `newtype` 根本不需要任何机器指令。在 `Ascii::from_bytes` 中，表达式 `Ascii(bytes)` 被简单地看作 `Vec<u8>` 的一种表现，只是它现在持有一个 `Ascii` 值。同理，`String::from_utf8_unchecked` 在内联时可能也不需要机器指令，因为 `Vec<u8>` 现在直接作为 `String` 使用。

22.4 不安全函数

`unsafe` 函数看起来就像前面加了 `unsafe` 关键字的普通函数。
`unsafe` 函数的主体自动被视为 `unsafe` 块。

只能在 `unsafe` 块中调用 `unsafe` 函数。这意味着将函数标记为 `unsafe` 会警告其调用者，为避免未定义行为，该函数具有他们必须满足的契约。

例如，下面是前面介绍的 `Ascii` 类型的新构造函数，它会从字节向量构建 `Ascii`，而不检查其内容是否为有效的 ASCII：

```
// 以下代码必须放在`my_ascii`模块内部
impl Ascii {
    /// 从`bytes`构造`Ascii`值，不检查`bytes`中是否真正包含格式良好的
    ASCII
    ///
    /// 这个构造函数是不会出错的，它会直接返回`Ascii`，而不会像
    /// `from_bytes`那样返回`Result<Ascii, NotAsciiError>`
    ///
    /// # 安全性
    ///
    /// 调用者必须确保`bytes`只包含ASCII字符：各字节
    /// 都不大于0x7f。否则，其行为就是未定义的
    pub unsafe fn from_bytes_unchecked(bytes: Vec<u8>) -> Ascii {
        Ascii(bytes)
    }
}
```

调用 `Ascii::from_bytes_unchecked` 的代码大概已经以某种方式知道了自己手中的向量只会包含 ASCII 字符，因此 `Ascii::from_bytes` 坚持要执行的检查只是浪费时间，调用者也将不得不编写代码来处理他知道永远不会发生的 `Err` 结果。
`Ascii::from_bytes_unchecked` 能让这样的调用者回避检查和错误处理。

但早些时候，为了确保 `Ascii` 值是格式良好的，我们强调了 `Ascii` 的公共构造函数和方法的重要性。`from_bytes_unchecked` 难道不

能履行这一责任吗？

并非如此，其实 `from_bytes_unchecked` 通过它的契约将这些义务推脱给了调用者。这个契约的存在使得将这个函数标记为 `unsafe` 是正确的：虽然函数本身没有执行任何不安全操作，但它的调用者必须遵守某些不能靠 Rust 自动执行的规则来避免未定义行为。

真的可以通过破坏 `Ascii::from_bytes_unchecked` 的契约来导致未定义行为吗？是的。可以构造一个包含格式错误的 UTF-8 的 `String`，如下所示：

```
// 将这个向量想象成用来生成ASCII的一些复杂过程的结果。但这里有问题！
let bytes = vec![0xf7, 0xbf, 0xbf, 0xbf];

let ascii = unsafe {
    // 如果`bytes`中存有非ASCII字节，就违反了这个不安全函数的契约
    Ascii::from_bytes_unchecked(bytes)
};

let bogus: String = ascii.into();

// `bogus`现在持有格式错误的UTF-8。解析其第一个字符会生成一个不是有效
Unicode
// 码点的`char`。这是未定义行为，所以语言无法说明这个断言应该是什么样的行为
assert_eq!(bogus.chars().next().unwrap() as u32, 0x1fffff);
```

在某些平台上某些版本的 Rust 中，会观察到此断言失败并显示以下有趣的错误消息：

```
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `2097151`,
 right: `2097151`, src/main.rs:42:5'
```

这两个数值在我们看来明明是相等的——这不是 Rust 的错，而是前一个 `unsafe` 块所导致的。当我们说未定义行为会导致无法预测的结果时，就是这个意思。

这个例子说明了关于 bug 和不安全代码的两个关键事实。

- 在 **unsafe** 块之前发生的 bug 可能会破坏契约。unsafe 块是否会导致未定义行为不仅取决于块本身的代码，还取决于为其提供操作目标的代码。unsafe 代码为满足契约所依赖的一切都与安全有关。仅当模块的其余部分都能正确维护 Ascii 的不变条件时，基于 `String::from_utf8_unchecked` 从 Ascii 到 String 的转换才是有明确定义的。
- 离开 **unsafe** 区块后，仍可能出现此处违约的后果。由于没有遵守不安全特性的契约而招致的未定义行为通常并不会发生在 unsafe 块内部。如前所述，伪造 String 的行为可能直到程序执行了很久之后才引发问题。

本质上，Rust 的类型检查器、借用检查器和其他静态检查都是在检查你的程序并试图构建出证据，证明它不会表现出未定义行为。如果 Rust 能成功编译程序，那么就意味着它成功地证明了你的代码是正确的。而 unsafe 块是这个证明中的一个缺口，也就是说，unsafe 块就相当于你对 Rust 说：“这段代码很好，请相信我。”你的声明正确与否可能取决于程序中会影响到此 unsafe 块的任意部分，并且其错误的后果也可能出现在受此 unsafe 块影响的任意地点。写出 unsafe 关键字，就相当于你在提醒自己没能充分利用该语言的安全检查。

如果可以选择，你自然更喜欢创建不需要契约的安全接口。这些接口更容易使用，因为用户可以依靠 Rust 的安全检查来确保他们的代码没有未定义行为。即使你的实现使用了不安全特性，最好还是使用 Rust 的类型、生命周期和模块系统来满足它们的契约，同时最好只使用你能自行担保的特性，而不是把责任转嫁给你的调用者。

不过遗憾的是，在实际开发中遇到不安全函数的情况并不少见，而这些函数的文档并没有认真地解释过它们的契约。因此，你要根据自己的经验和对代码行为方式的了解自行推断出规则。如果你曾焦虑不安地想知道用 C API 或 C++ API 所做的事情是否正常，那么对这种感觉肯定也感同身受。

22.5 不安全块还是不安全函数

你可能想知道应该使用 `unsafe` 块还是将整个函数都标记为 `unsafe`。我们推荐的方法是先对函数做一些判定。

- 如果能正常编译，但仍可能导致未定义行为的方式滥用函数，则必须将其标记为不安全。正确使用函数的规则是它的契约，契约的存在意味着函数是不安全的。
- 否则，函数就是安全的。也就是说，对函数的任何类型良好的调用都不会导致未定义行为。这样的函数不应该标记为 `unsafe`。

函数是否在函数体中使用了不安全特性无关紧要，重要的是契约存在与否。之前，我们曾展示过一个没有使用不安全特性的不安全函数，以及一个使用了不安全特性的安全函数。

不要仅仅因为函数体中使用了不安全特性就把安全的函数标记为 `unsafe`。这会让函数更难使用，并使那些期望在某处找到契约说明的读者感到困惑（只要是 `unsafe` 就理当有契约说明）。相反，应该使用 `unsafe` 块，即便整个函数体只有这一个块。

22.6 未定义行为

如本章开头所述，术语未定义行为的意思是“Rust 坚定地认为你的代码永远不会出现的行为”。这是一个奇怪的措辞，特别是因为我们从使用其他语言的经验中就能知道这些行为**确实**会以某种频率偶然发生。为什么这个概念对厘清不安全代码的责任有帮助？

编译器是从一种编程语言到另一种编程语言的翻译器。Rust 编译器是将 Rust 程序翻译成等效的机器语言的程序。但是，说“两个使用完全不同语言的程序是等效的”意味着什么呢？

幸运的是，这个问题对程序员来说比对语言学家更容易理解。我们通常说两个程序是等效的，意思是它们在执行时总是具有相同的可见行为，比如会进行相同的系统调用，以等效的方式与外部库交互，等等。这有点儿像程序界的图灵测试：如果不能分辨自己是在与原文交互还是与译文交互，那它们就是等效的。

现在考虑以下代码：

```
let i = 10;
very_trustworthy(&i);
println!("{}", i * 100);
```

即使对 `very_trustworthy` 的定义一无所知，也可以看到它仅接收对 `i` 的共享引用，因此该调用肯定无法更改 `i` 的值。由于传给 `println!` 的值永远是 1000，因此 Rust 可以将这段代码翻译成机器语言，就像我们写过的一样：

```
very_trustworthy(&10);
println!("{}", 1000);
```

这个转换后的版本与原始版本具有相同的可见行为，而且速度可能更快一点儿。但只有在保证它与原始版本具有相同含义的前提下，才值得去考虑此版本的性能。如果 `very_trustworthy` 的定义是下面这样的该怎么办？

```
fn very_trustworthy(shared: &i32) {
    unsafe {
        // 把共享引用转换成可变指针
        // 这是一种未定义行为
        let mutable = shared as *const i32 as *mut i32;
        *mutable = 20;
    }
}
```

这段代码打破了共享引用的规则：它将 `i` 的值更改为 20，即使这个值应该被冻结（因为 `i` 是为了共享而借用的）。结果，我们对调用者所做的转换现在有了非常明显的效果：如果 Rust 转换这段代码，那么程序就会打印 1000；如果它保留代码并使用 `i` 的新值，则程序会打印 2000。在 `very_trustworthy` 中打破共享引用的规则意味着共享引用在其调用者中的行为可能不再符合预期了。

这种问题几乎会出现在 Rust 可能尝试的每一种转换中，其中就包括：即使把一个函数内联到它的调用点，也仍然可以假设当被调用者完成时，控制流就会返回到调用点。然而我们在本章开头就给出过一个违反了该假设的问题代码示例。

Rust（或任何其他语言）基本上不可能评估出对程序的转换是否保留了其含义，除非它可以相信语言的基本特性会按原本的设计运行。一段代码是否可信，不仅取决于手头的这部分代码，还取决于程序中隔得比较远的其他部分的代码。为了对代码做任何处理，Rust 必须假设你的程序的其余部分是“遵纪守法”的。

下面是 Rust 判断程序是否“遵纪守法”的规则。

- 程序不得读取未初始化的内存。
- 程序不得创建无效的原始值。
 - 引用、Box 值或 fn 指针为空（null）。
 - bool 值非 0 且非 1。
 - enum 值具有无效判别值。
 - char 值无效，比如存在半代用区的 Unicode 码点。
 - str 值不是格式良好的 UTF-8。
 - 胖指针具有无效虚表或 slice 长度。

- `never` 类型的任何值，可以写作 `!`，只能用于不会返回的函数。
- 程序必须遵守第 5 章中解释过的引用规则。任何引用的生命周期都不能超出其引用目标，共享访问是只读访问，可变访问是独占访问。
- 程序不得对空指针、未正确对齐的指针或悬空指针进行解引用。
- 程序不得使用指针访问与此指针关联的分配区之外的内存。22.8.1 节会详细解释此规则。
- 程序必须没有数据竞争。当两个线程在没有同步保护的情况下访问同一个内存位置，并且至少有一个访问是写入时，就会发生数据竞争。
- 程序不得对借助外部函数接口进行的跨语言调用进行栈展开（参见 7.1.1 节）。
- 程序必须遵守标准库函数的契约。

由于还没有针对 `unsafe` 代码的 Rust 语义的完整模型，因此该列表可能会随着时间的推移而演变，但上述这些规则仍然有效。

任何违反这些规则的行为都构成了未定义行为，并让 Rust 试图优化你的程序并将其翻译成机器语言的努力变得不可信。如果你违反最后一条规则并将格式错误的 UTF-8 传给

`String::from_utf8_unchecked`，那么没准儿 2097151 真会不等于 2097151。

未使用不安全特性的 Rust 代码只要编译通过就可以保证会遵守前面的所有规则。（假设编译器自身没有 bug——我们正为之努力，但没有 bug 永远只能是个理想。）只有在使用不安全特性时，遵守这些规则才会成为你的责任。

而在 C 和 C++ 中，你的程序“在编译期没有错误或警告”这件事意义不大。正如本书前面所提到的，即使那些一直坚持着高标准且备受推崇的项目所编写的最好的 C 程序和 C++ 程序，也会在实践中表现出未定义行为。

22.7 不安全特型

不安全特型是一种特型，用于表示这里存在某种 Rust 无法检查也无法强制保障的契约。实现者必须满足它，以规避未定义行为。要实现不安全特型，就必须将实现标记为不安全的。你需要了解此特型的契约并确保自己的类型能满足它。

类型变量以某个不安全特型为限界的函数通常是自身使用了不安全特型的函数，并且只能依靠此不安全特型的契约来满足那些不安全特型的契约。对此特型的不正确实现可能导致这样的函数出现未定义行为。

`std::marker::Send` 和 `std::marker::Sync` 是不安全特型的典型示例。这些特型没有定义任何方法，因此你可以用喜欢的任意类型来轻松实现它们。但它们确实有契约：`Send` 要求实现者能安全地转移给另一个线程，而 `Sync` 要求实现者能安全地通过共享引用在线程之间共享。如果为不合适的类型实现了 `Send`，就会使 `std::sync::Mutex` 在数据竞争中不再安全。

举个简单的例子，Rust 标准库曾包含一个不安全特型 `core::nonzero::Zeroable`，该特型用于标记出可通过将所有字节设置为 0 来进行安全初始化的类型。显然，将 `usize` 变量归零肯定没问题，但将 `&T` 归零就会带来一个空引用，如果解引用，则会导致崩溃。对于 `Zeroable` 的类型，可以进行一些优化：可以使用 `std::ptr::write_bytes`（`memset` 在 Rust 中的等价物）或者用能分配全零内存页的系统调用来快速初始化数组。（`Zeroable` 是不稳定的，在 Rust 1.26 的 `num crate` 中被转移到仅供内部使用，但它是一个优秀、简单且真实的例子。）

`Zeroable` 是一个典型的标记特型，缺少方法或关联类型：

```
pub unsafe trait Zeroable {}
```

它对适用类型的实现同样简单明了：


```
unsafe impl Zeroable for u8 {}
unsafe impl Zeroable for i32 {}
unsafe impl Zeroable for usize {}
// 以及所有整数类型
```

有了这些定义，就可以编写一个函数来快速分配给定长度的包含 `Zeroable` 类型的向量了：

```
use core::nonzero::Zeroable;

fn zeroed_vector<T>(len: usize) -> Vec<T>
    where T: Zeroable
{
    let mut vec = Vec::with_capacity(len);
    unsafe {
        std::ptr::write_bytes(vec.as_mut_ptr(), 0, len);
        vec.set_len(len);
    }
    vec
}
```

这个函数会首先创建一个具有所需容量的空 `Vec`，然后调用 `write_bytes` 以用 0 填充未占用的缓冲区。（`write_byte` 函数会将 `len` 视为 `T` 元素的数量，而不是字节数，因此该调用确实会填充整个缓冲区。）向量的 `set_len` 方法只会更改其长度而不会对缓冲区做任何事，这是不安全的，因为必须保证新的缓冲区空间确实包含已正确初始化的 `T` 类型值。不过这正是 `T: Zeroable` 类型限界所保证的：全零的字节块表示有效的 `T` 值。我们对 `set_len` 的使用是安全的。

下面我们来使用这个函数：

```
let v: Vec<usize> = zeroed_vector(100_000);
assert!(v.iter().all(|&u| u == 0));
```

显然，`Zeroable` 一定是一个不安全特型，因为不遵守其契约的实现可能会导致未定义行为：

```
struct HoldsRef<'a>(&'a mut i32);

unsafe impl<'a> Zeroable for HoldsRef<'a> { }

let mut v: Vec<HoldsRef> = zeroed_vector(1);
*v[0].0 = 1;    // 崩溃：对空指针解引用
```

Rust 不知道 `Zeroable` 意味着什么，所以无从判断它何时会被实现为不合适的类型。与其他任何不安全特性一样，如何理解并遵守不安全特型的契约由你来决定。

请注意，不安全代码不得依赖于普通的、安全的特型在实现上的正确性。假设有一个 `std::hash::Hasher` 特型的实现，它只会返回一个随机哈希值，与被哈希的值无关。该特型要求对一些相同的位进行两次哈希后必须生成相同的哈希值，但此实现无法满足该要求，这根本就不正确。但因为 `Hasher` 并不是不安全特型，所以不安全代码在使用这个哈希器时不得表现出未定义行为。²为了满足“可以使用不安全特性”这条契约，`std::collections::HashMap` 类型是经过精心编写的，但并未考虑哈希器自身行为出错的可能性。当然，这样一来该哈希表将无法正常运行：查找将失败，条目将随机出现和消失。但该哈希表并不存在未定义行为。

² `Hasher` 的行为是有明确定义的，别人实现错了并不等于此特型未定义。——译者注

22.8 裸指针

裸指针在 Rust 中就是一种不受约束的指针。你可以使用裸指针来创建 Rust 的受检查指针类型不能创建的各种结构，比如双向链表或任意对象图。但是因为裸指针非常灵活，Rust 无法判断你是否在安全地使用它们，所以只能在 `unsafe` 块中对它们解引用。

裸指针本质上等效于 C 指针或 C++ 指针，因此在与这些语言编写的代码进行交互时它们也很有用。

裸指针有以下两种类型。

- `*mut T` 是指向 `T` 的允许修改其引用目标的裸指针。
- `*const T` 是指向 `T` 的只允许读取其引用目标的裸指针。

（没有单纯的 `*T` 类型，必须始终指定 `const` 或 `mut`。）

可以把引用转换成裸指针，并使用 `*` 运算符对其解引用：

```
let mut x = 10;
let ptr_x = &mut x as *mut i32;

let y = Box::new(20);
let ptr_y = &*y as *const i32;

unsafe {
    *ptr_x += *ptr_y;
}
assert_eq!(x, 30);
```

与 `Box` 和引用不同，裸指针可以为空，就像 C 中的 `NULL` 或 C++ 中的 `nullptr`：

```
fn option_to_raw<T>(opt: Option<&T>) -> *const T {
    match opt {
        None => std::ptr::null(),
        Some(r) => r as *const T
    }
}
```

```
assert!(!option_to_raw(Some(&("pea", "pod"))).is_null());
assert_eq!(option_to_raw:<i32>(None), std::ptr::null());
```

这个例子中没有 `unsafe` 块：创建裸指针、传递裸指针和比较裸指针都是安全的。只有解引用裸指针是不安全的。

指向无固定大小类型的裸指针是胖指针，就像相应的引用或 `Box` 类型一样。`*const [u8]` 指针包括一个长度和地址，而像 `*mut dyn std::io::Write` 指针这样的特型对象则会携带一个虚表。

尽管 Rust 会在各种情况下隐式解引用安全指针类型，但对裸指针解引用必须是显式的。

- `.` 运算符不会隐式解引用裸指针，必须写成 `(*raw).field` 或 `(*raw).method(...)`。
- 裸指针没有实现 `Deref`，因此隐式解引用不适合它们。
- `==`、`<` 等运算符将裸指针作为地址进行比较：如果两个裸指针指向内存中的相同位置，那它们就相等。类似地，对裸指针进行哈希处理会针对其指向的地址值本身，而不会针对其引用目标的值。
- 像 `std::fmt::Display` 这样的格式化特型会自动追踪引用，但根本不会处理裸指针。`std::fmt::Debug` 和 `std::fmt::Pointer` 是例外，它们会将裸指针展示为十六进制地址，而不会解引用它们。

与 C 和 C++ 中的 `+` 运算符不同，Rust 的 `+` 不会处理裸指针，但可以通过它们的 `offset` 方法和 `wrapping_offset` 方法或更方便的 `add` 方法、`sub` 方法、`wrapping_add` 方法和 `wrapping_sub` 方法执行指针运算。反过来，`offset_from` 方法会以字节为单位求出两个指针之间的距离，不过需要确保开始和结束位于同一个内存区域，比如在同一 `Vec` 中：

```
let trucks = vec!["garbage truck", "dump truck", "moonstruck"];
let first: *const &str = &trucks[0];
let last: *const &str = &trucks[2];
```

```
assert_eq!(unsafe { last.offset_from(first) }, 2);
assert_eq!(unsafe { first.offset_from(last) }, -2);
```

`first` 和 `last` 不需要显式转换，只需指定类型即可。Rust 会将引用隐式转换成裸指针（当然，反过来肯定不成立）。

`as` 运算符允许从引用到裸指针或两个裸指针类型之间几乎所有的合理转换。但是，可能需要将复杂的转换分解为一系列更简单的步骤。例如：

```
&vec![42_u8] as *const String; // 错误：无效的转换
&vec![42_u8] as *const Vec<u8> as *const String; // 这样可以转换
```

请注意，`as` 不会将裸指针转换为引用。这样的转换不安全，`as` 应该保持安全操作。因此，必须在 `unsafe` 块中对裸指针解引用，然后再借用其结果值。

这样操作时要非常小心：以这种方式生成的引用具有不受约束的生命周期，它可以存续多长时间没有限制，因为裸指针没有给 Rust 提供任何能做出这种决定的依据。23.5 节会展示几个如何正确限制生命周期的示例。

许多类型有 `as_ptr` 方法和 `as_mut_ptr` 方法，它们会返回指向其内容的裸指针。例如，数组切片和字符串会返回指向它们第一个元素的指针，而一些迭代器会返回指向它们将生成的下一个元素的指针。像 `Box`、`Rc` 和 `Arc` 这样的拥有型指针类型都有 `into_raw` 函数和 `from_raw` 函数，可以与裸指针相互转换，其中一些方法的契约强加了出人意料的要求，因此在使用之前务必检查一下它们的文档。

还可以通过转换整数来构造裸指针，不过你唯一可以信任的整数通常就是从指针转换来的。22.8.2 节就以这种方式使用了裸指针。

与引用不同，裸指针既不是 `Send` 的也不是 `Sync` 的。因此，在默认情况下，任何包含裸指针的类型都不会实现这些特型。在线程之间发送或共享裸指针本身其实并没有什么不安全的，毕竟，无论它们“走”到哪里，你都需要一个 `unsafe` 块来解引用它们。但是考虑到

裸指针经常扮演的角色，语言设计者认为还是现在这种默认使用方式更好。22.7 节讨论过如何自己实现 `Send` 和 `Sync`。

22.8.1 安全地解引用裸指针

以下是安全使用裸指针的一些常识性指南。

- 解引用空指针或悬空指针是未定义行为，引用未初始化的内存或超出作用域的值也一样。
- 解引用未针对其引用目标的类型正确对齐的指针是未定义行为。
- 只有在遵守了第 5 章解释过的引用安全规则（任何引用的生命周期都不能超出其引用目标，共享访问是只读访问，可变访问是独占访问）的前提下，才能从解引用的裸指针中借用值。（很容易意外违反这条规则，因为裸指针通常用于创建具有非标准共享或所有权的数据结构。）
- 仅当引用目标是所属类型的格式良好的值时，才能使用裸指针的引用目标。例如，必须确保解引用 `*const char` 后会产生一个正确的、不在半代用区的 Unicode 码点。
- 如果想在特定的裸指针上使用 `offset` 方法和 `wrapping_offset` 方法，那么该裸指针只能指向原初（`original`）指针所引用的变量内部的字节或分配在堆上的内存块内部的字节，或者指向上述两个区域之外的第一字节。

如果通过将指针转换为整数，对整数进行运算，然后将其转换回指针的方式进行指针运算，则结果必须是 `offset` 方法的规则允许生成的指针。

- 如果要给裸指针的引用目标赋值，则不得违反引用目标所属的任何类型的不变条件。如果你有一个 `*mut u8` 指向 `String` 中的一字节，那么在该 `u8` 中存储的值必须能让 `String` 保持为格式良好的 UTF-8。

抛开借用规则不谈，上述规则与在 C 或 C++ 中使用指针时必须遵守的规则基本上是一样的。

不得违反类型不变条件的原因应该很清楚。许多 Rust 标准库类型在其实现中使用了不安全代码，但仍然提供了安全接口，前提是 Rust 的安全检查、模块系统和可见性规则能得到遵守。使用裸指针来规避这些保护措施可能会导致未定义行为。

裸指针的完整、准确的契约不容易表述，并且可能随着语言的发展而改变。但本节概要表述的这些原则应该让你处于安全地带。

22.8.2 示例：RefWithFlag

下面这个例子说明了如何采用裸指针实现经典³的位级 hack，并将其包装为完全安全的 Rust 类型。这个模块定义了一个类型 `RefWithFlag<'a, T>`，它同时包含一个 `&'a T` 和一个 `bool`，就像元组 `(&'a T, bool)` 一样，但仍然设法只占用了 1 个机器字而不是两个。这种技术在垃圾回收器和虚拟机中经常使用，其中某些类型（比如表示对象的类型）的数量多到就算只向每个值添加一个机器字都会大大增加内存占用：

³ 好吧，应该说是成就我们（本书作者）的经典。

```
mod ref_with_flag {
    use std::marker::PhantomData;
    use std::mem::align_of;

    /// 包装在单个机器字中的 `&T` 和 `bool`
    /// 类型 `T` 要求必须至少按两字节对齐
    ///
    /// 如果你是那种中规中矩的程序员，从未想过还能从某个指针中偷出
    /// 第 20 位（数据的最低位），那么现在可以安全地做到这一点了！
    /// （“但这样做并不像想象中那么刺激啊……”）
    pub struct RefWithFlag<'a, T> {
        ptr_and_bit: usize,
        behaves_like: PhantomData<&'a T> // 不占空间
    }

    impl<'a, T: 'a> RefWithFlag<'a, T> {
```

```

pub fn new(ptr: &'a T, flag: bool) -> RefWithFlag<T> {
    assert!(align_of::<T>() % 2 == 0);
    RefWithFlag {
        ptr_and_bit: ptr as *const T as usize | flag as
usize,
        behaves_like: PhantomData
    }
}

pub fn get_ref(&self) -> &'a T {
    unsafe {
        let ptr = (self.ptr_and_bit & !1) as *const T;
        &*ptr
    }
}

pub fn get_flag(&self) -> bool {
    self.ptr_and_bit & 1 != 0
}
}

```

这段代码利用了这样一个事实，即许多类型在内存中必须放置在偶数地址：由于偶数地址的最低有效位始终为 0，因此可以在那里存储其他内容，然后通过屏蔽最低位来可靠地重建原始地址。并非所有类型都符合条件，比如类型 `u8` 和 `(bool, [i8; 2])` 可以放在任何地址。但是我们可以检查此类型在构造方面的对齐情况，并拒绝不适用的类型。

可以像下面这样使用 `RefWithFlag`：

```

use ref_with_flag::RefWithFlag;

let vec = vec![10, 20, 30];
let flagged = RefWithFlag::new(&vec, true);
assert_eq!(flagged.get_ref()[1], 20);
assert_eq!(flagged.get_flag(), true);

```

构造函数 `RefWithFlag::new` 会接受一个引用和一个 `bool` 值，并断言此引用具有适当的类型，然后把它转换为裸指针，再转换为 `usize` 类型。`usize` 类型大小的定义是足够在我们正在编译的任何处理器上保存一个指针，因此将裸指针转换为 `usize` 并返回它是有

明确定义的。一旦有了 `usize`，我们就知道它必然是偶数，所以可以使用按位或运算符 `|` 将其与已转换为整数 0 或 1 的 `bool` 值组合起来。

`get_flag` 方法用于提取 `RefWithFlag` 的 `bool` 部分。这很简单：只要取出最低位并检查结果是否非零就可以了
(`self.ptr_and_bit & 1 != 0`)。

`get_ref` 方法用于从 `RefWithFlag` 中提取引用。首先，它会屏蔽 `usize` 的最低位 (`self.ptr_and_bit & !1`) 并将其转换为裸指针。`as` 运算符无法将裸指针转换为引用，但我们可以解引用裸指针（当然是在 `unsafe` 块中）并借用它。借用一个裸指针的引用目标会得到一个无限生命周期的引用：Rust 会赋予引用任何生命周期来检查它周围的代码（如果有的话）。但是，通常还有一些更准确的特定生命周期，因此会发现更多错误。在这个例子中，由于 `get_ref` 的返回类型是 `&'a T`，因此 Rust 认为该引用的生命周期与 `RefWithFlag` 的生命周期参数 `'a` 相同，这正是我们想要的，因为这个生命周期就是最初那个引用的生命周期。

在内存中，`RefWithFlag` 看起来很像 `usize`：由于 `PhantomData`（意思是虚构的数据）是零大小的类型，因此 `behaves_like` 字段并不会占用结构体中的空间。但是，为了让 Rust 知道该如何处理使用 `RefWithFlag` 的代码中的生命周期，`PhantomData` 是必需的。想象一下没有 `behaves_like` 字段的类型会是什么样子：

```
// 这无法编译
pub struct RefWithFlag<'a, T: 'a> {
    ptr_and_bit: usize
}
```

如第 5 章所述，任何包含引用的结构体，其生命周期都不能超出它们借用的值，以免引用变成悬空指针。这个结构体必须遵守适用于其字段的限制。这当然也适用于 `RefWithFlag`：在刚刚看到的示例代码中，`flagged` 的生命周期不能超出 `vec`，因为 `flagged.get_ref()` 会返回对它的引用。但是我们简化版的 `RefWithFlag` 类型根本不包含任何引用，并且从不使用其生命周期

参数 'a，因为这只是一个 `usize`。怎么让 Rust 知道应该如何限制 `flagged` 的生命周期呢？包含一个 `PhantomData<&'a T>` 字段就是为了告诉 Rust 应该将 `RefWithFlag<'a, T>` 视为包含一个 `&'a T`，却不会实际影响此结构体的表示方式。

尽管 Rust 并不真正知道发生了什么（这就是 `RefWithFlag` 不安全的原因），但它会尽力帮助你解决这个问题。如果省略了 `behaves_like` 字段，那么 Rust 就会报错说参数 'a 和 T 未使用，并建议使用 `PhantomData`。

`RefWithFlag` 使用了与之前介绍的 `Ascii` 类型相同的策略来避免其 `unsafe` 块中的未定义行为。类型本身是 `pub` 的，但其字段不是，这意味着只有 `ref_with_flag` 模块中的代码才能创建或查看 `RefWithFlag` 值。你不必检查太多代码就可以确信 `ptr_and_bit` 字段是构造良好的。

22.8.3 可空指针

Rust 中的空裸指针是一个零地址，与 C 和 C++ 中一样。对于任意类型 `T`，`std::ptr::null<T>` 函数会返回一个 `*const T` 空指针，而 `std::ptr::null_mut<T>` 会返回一个 `*mut T` 空指针。

检查裸指针是否为空有几种方法。最简单的是 `is_null` 方法，但 `as_ref` 方法可能更方便。`as_ref` 方法会接受 `*const T` 指针并返回 `Option<&'a T>`，以便将一个空指针变成 `None`。同样，`as_mut` 方法会将 `*mut T` 指针转换为 `Option<&'a mut T>` 值。

22.8.4 类型大小与对齐方式

任何固定大小类型（`Sized`）的值都会在内存中占用固定数量的字节，并且必须放置在由机器体系结构决定的某个对齐值的倍数的地址处。例如，一个 `(i32, i32)` 元组占用 8 字节，而大多数处理器更喜欢将其放置在 4 的倍数地址处。

调用 `std::mem::size_of::<T>()` 会返回类型 `T` 值的大小（以字节为单位），而调用 `std::mem::align_of::<T>()` 会返回其所需的对齐方式。例如：

```
assert_eq!(std::mem::size_of::<i64>(), 8);
assert_eq!(std::mem::align_of::<i32, i32>(), 4);
```

任何类型总是对齐到二的 n 次幂。

即使在技术上可以填入更小的空间，类型的大小也总是会四舍五入为其对齐方式的倍数。例如，尽管像 `(f32, u8)` 这样的元组只需要 5 字节，但 `size_of::<(f32, u8)>()` 是 8，因为 `align_of::<(f32, u8)>()` 是 4。这会确保如果你有一个数组，那么元素类型的大小总能反映出一个元素与其下一个元素的间距。

对于无固定大小类型，其大小和对齐方式取决于手头的值。给定对无固定大小值的引用，`std::mem::size_of_val` 函数和 `std::mem::align_of_val` 函数会返回值的大小和对齐方式。这两个函数可以对固定大小类型和无固定大小类型的引用进行操作。

```
// 指向切片的胖指针包含其引用目标的长度
let slice: &[i32] = &[1, 3, 9, 27, 81];
assert_eq!(std::mem::size_of_val(slice), 20);

let text: &str = "alligator";
assert_eq!(std::mem::size_of_val(text), 9);

use std::fmt::Display;
let unremarkable: &dyn Display = &193_u8;
let remarkable: &dyn Display = &0.0072973525664;

// 这些会返回特型对象指向的值的大小/对齐方式，而不是特型对象
// 本身的大小/对齐方式。此信息来自特型对象引用的虚表
assert_eq!(std::mem::size_of_val(unremarkable), 1);
assert_eq!(std::mem::align_of_val(remarkable), 8);
```

22.8.5 指针运算

Rust 会将数组、切片或向量的元素排布为单个连续的内存块，如图 22-1 所示。元素的间隔很均匀，因此如果每个元素占用 `size` 字节，则第 `i` 个元素就从第 `i * size` 字节开始。

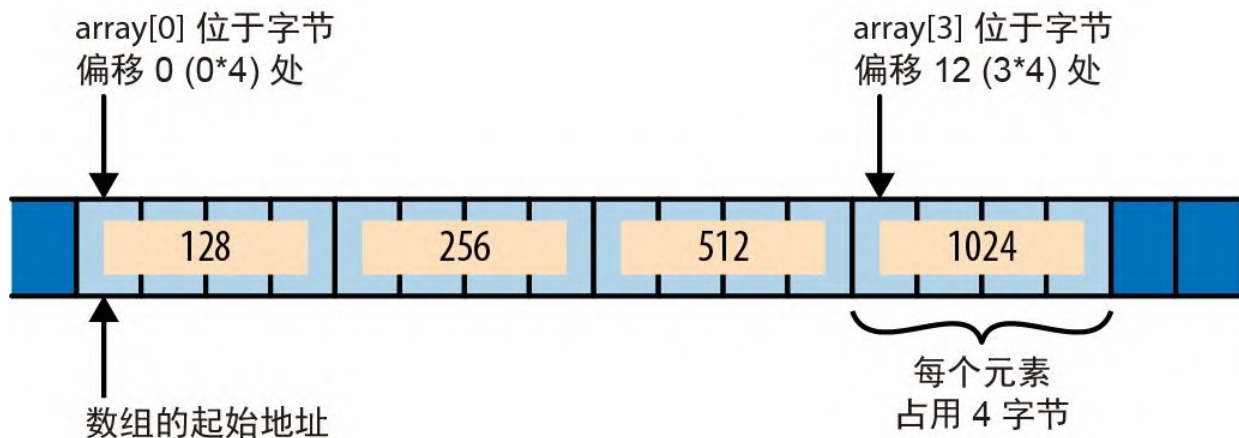


图 22-1：内存中的数组

这样做有一个好处：如果你有两个指向数组元素的裸指针，那么比较指针就会得到与比较元素索引相同的结果。如果 `i < j`，则指向第 `i` 个元素的裸指针一定小于指向第 `j` 个元素的裸指针。这使得裸指针可用作数组遍历的边界。事实上，标准库对切片的简单迭代器最初就是这样定义的：

```
struct Iter<'a, T> {  
    ptr: *const T,  
    end: *const T,  
    ...  
}
```

`ptr` 字段指向迭代应该生成的下一个元素，`end` 字段作为界限：当 `ptr == end` 时，迭代完成。

数组布局的另一个好处是：如果 `element_ptr` 是指向某个数组的第 `i` 个元素的 `*const T` 或 `*mut T` 裸指针，那么 `element_ptr.offset(o)` 就是指向第 `(i + o)` 个元素的裸指针。它的定义等效于如下内容：

```
fn offset<T>(ptr: *const T, count: isize) -> *const T
    where T: Sized
{
    let bytes_per_element = std::mem::size_of::<T>() as isize;
    let byte_offset = count * bytes_per_element;
    (ptr as isize).checked_add(byte_offset).unwrap() as *const T
}
```

`std::mem::size_of::<T>` 函数会返回类型 `T` 的字节大小。根据定义，由于 `isize` 大到足以容纳一个地址，因此可以将基指针转换为 `isize`，对得到的值进行算术运算，然后将结果转换回指针。

可以生成指向数组末尾之后第一字节的指针。虽然不能对这样的指针解引用，但可以用它来表示循环的界限或用于边界检查。

但是，使用 `offset` 生成超出该点或指向数组开头之前的指针是未定义行为，即使从未对它解引用也是如此。为了方便优化，Rust 会假设当 `i` 为正值时 `ptr.offset(i) > ptr`，当 `i` 为负值时 `ptr.offset(i) < ptr`。这个假设似乎是安全的，但如果 `offset` 中的算术溢出了 `isize` 值，那么可能就不成立了。如果把 `i` 限制在 `ptr` 的同一个数组范围内，则肯定不会发生溢出：毕竟数组本身不会溢出地址空间的边界。（为了让指向结尾之后第一字节的指针安全，Rust 从来都不会将值放在地址空间的上端。）

如果确实需要将指针偏移到与其关联的数组的界限之外，则可以使用 `wrapping_offset` 方法。该方法与 `offset` 等效，但 Rust 不会假设 `ptr.wrapping_offset(i)` 和 `ptr` 本身的相对顺序。当然，你仍然不能对此类指针解引用，除非确信它们会落在数组中。

22.8.6 移动入和移动出内存

如果你正在实现的类型需要管理自己的内存，那么就要跟踪内存中哪些部分保存了有效值，而哪些是未初始化的，就像 Rust 处理局部变量一样。考虑下面这段代码：

```
let pot = "pasta".to_string();
let plate = pot;
```

上述代码运行后，情况如图 22-2 所示。

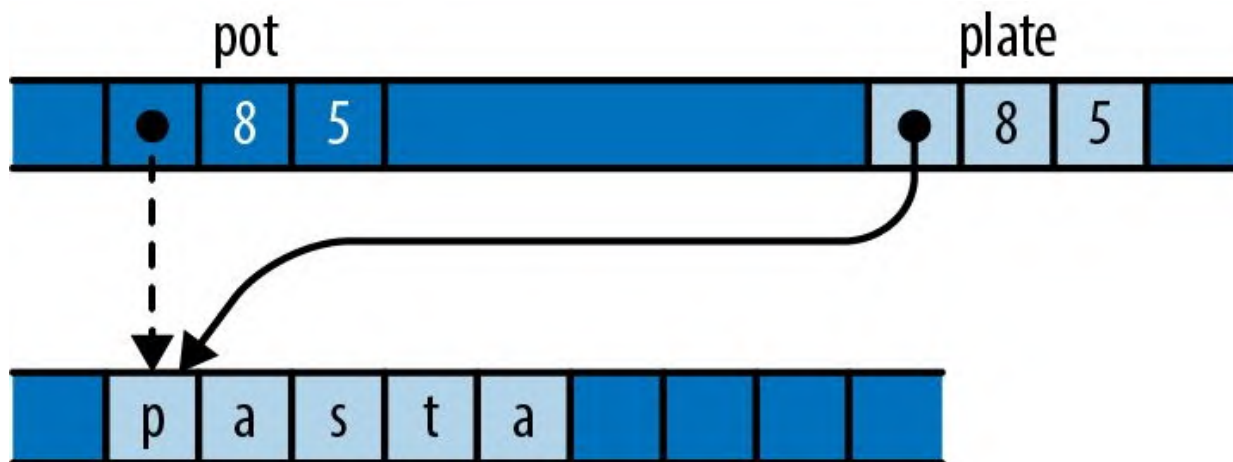


图 22-2：将字符串从一个局部变量转移给另一个局部变量

赋值后，`pot` 处于未初始化状态，而 `plate` 成了字符串的拥有者。

在机器层面，没有指定移动对源值的作用，但实际上它通常什么都不做。该赋值可能会使 `pot` 仍然保存着字符串的指针、容量和长度。当然，如果继续将其视为有效值将是灾难性的，但 Rust 会确保你不会这样做。

同样的考虑也适用于管理自己内存的数据结构。假设你运行了下面这段代码：

```
let mut noodles = vec!["udon".to_string()];  
let soba = "soba".to_string();  
let last;
```

在内存中，状态如图 22-3 所示。

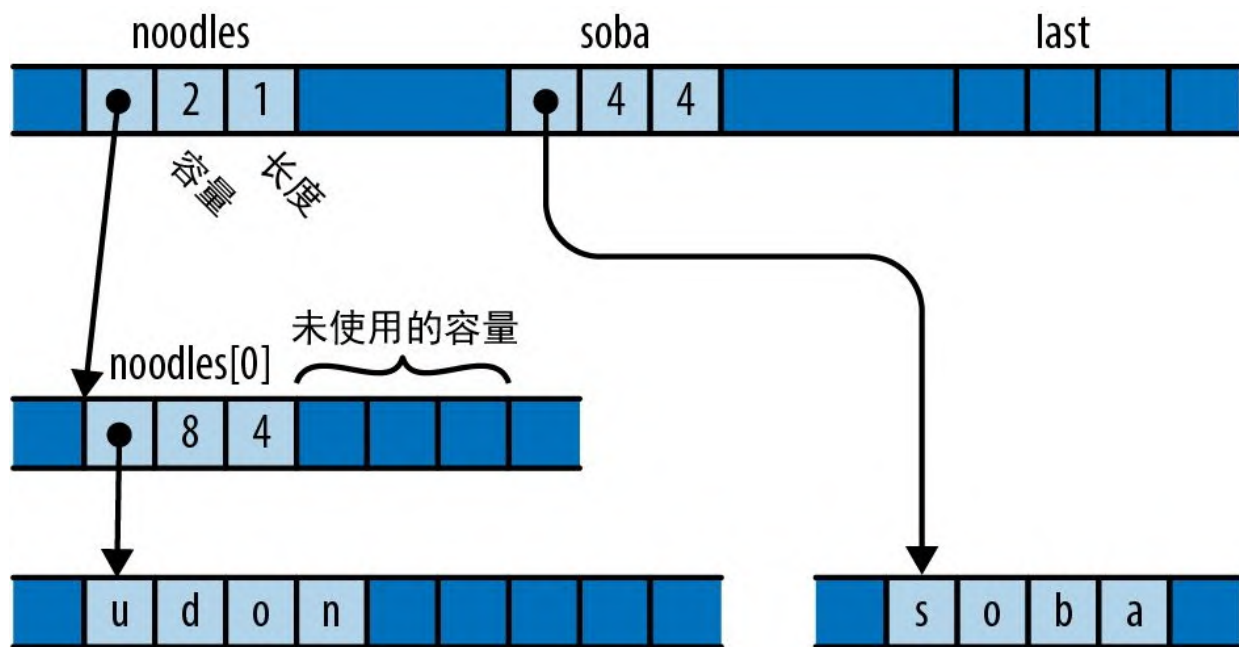


图 22-3：具有未初始化的空闲容量的向量

这个向量有空闲容量可以再容纳一个元素，但空闲容量中存放的是垃圾数据，可能是以前的内存残余。假设你随后运行了如下代码：

```
noodles.push(soba);
```

将字符串压入向量会将未初始化的内存转换为新元素，如图 22-4 所示。

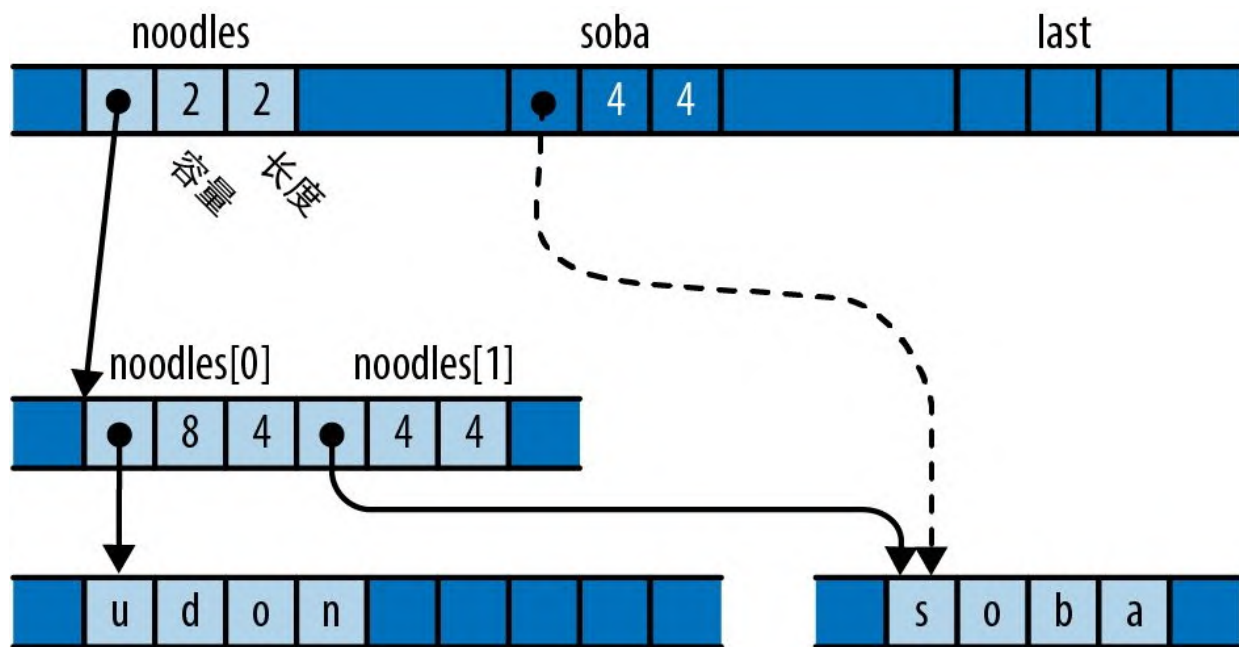


图 22-4：将 **soba** 的值推入向量之后

该向量已初始化其空白空间，以便拥有该字符串，并增加其长度，以便将其标记为新的有效元素。向量现在是字符串的拥有者，你可以引用它的第二个元素了，而丢弃此向量将释放两个字符串。**soba** 现在处于未初始化状态。

最后，考虑一下当从向量中弹出一个值时会发生什么：

```
last = noodles.pop().unwrap();
```

在内存中，现在看起来如图 22-5 所示。

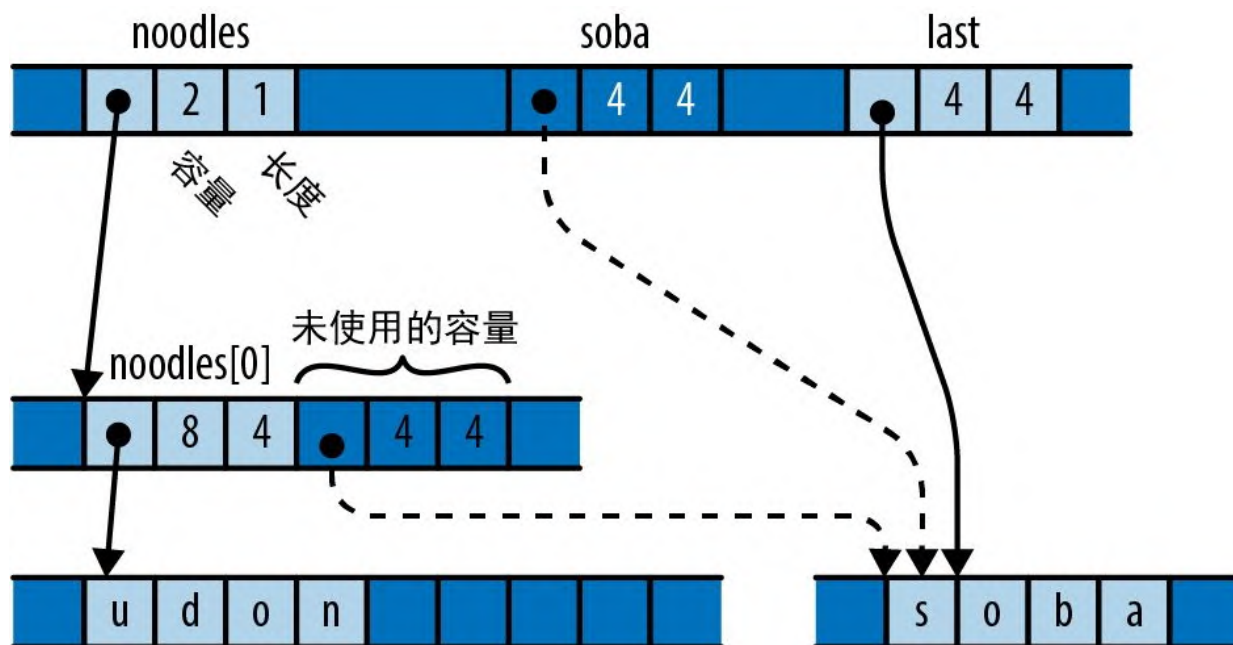


图 22-5：把向量中的一个元素弹出到 **last** 之后

变量 `last` 取得了字符串的所有权。向量已减小其 `length` 以指示用于保存字符串的空间现在未初始化。

就像之前的 `pot` 和 `pasta` 一样，`soba`、`last` 和向量的可用空间这三者可能存有相同的位模式。但只有 `last` 被认为拥有这个值。将其他两个位置中的任何一个视为有效位置都是错误的。

初始化值的真正定义是应视为有效的值。写入值的字节通常是初始化的必要部分，但这只是为了将其视为有效的而做的准备工作。移动和复制对内存的影响是一样的，两者之间的区别在于，在移动之后，源不再被视为有效值，而在复制之后，源和目标都处于有效状态。

Rust 会在编译期跟踪哪些局部变量处于有效状态，并阻止你使用值已转移给其他地方的变量。`Vec`、`HashMap`、`Box` 等类型会动态跟踪它们的缓冲区。如果你实现了一个管理自己内存的类型，则也需要这样做。

Rust 为实现这些类型提供了两个基本操作。

```
std::ptr::read(src) (读取)
```

将值移动出 `src` 指向的位置，将所有权转移给调用者。`src` 参数应该是一个 `*const T` 裸指针，其中 `T` 是一个固定大小类型。调用此函数后，`*src` 的内容不受影响，但除非 `T` 是 `Copy` 类型，否则你必须确保自己的程序会将它们视为未初始化内存。

这是 `Vec::pop` 背后的操作。要弹出一个值，就要调用 `read` 将该值移出缓冲区，然后递减长度以将该空间标记为未初始化容量。

```
std::ptr::write(dest, value) (写入)
```

将 `value` 转移给 `dest` 指向的位置，该位置在调用之前必须是未初始化内存。引用目标现在拥有该值。在这里，`dest` 必须是一个 `*mut T` 裸指针并且 `value` 是一个 `T` 值，其中 `T` 是固定大小类型。

这就是 `Vec::push` 背后的操作。压入一个值会调用 `write` 将值转移给下一个可用空间，然后增加长度以将该空间标记为有效元素。

两者都是自由函数，而不是裸指针类型的方法。

请注意，不能使用任何 Rust 的安全指针类型来执行这些操作。安全指针类型会要求其引用目标始终是初始化的，因此将未初始化内存转换为值或相反的操作都超出了它们的能力范围。而裸指针符合这种要求。

标准库还提供了将值数组从一个内存块移动到另一个内存块的函数。

```
std::ptr::copy(src, dst, count) (复制)
```

将内存中从 `src` 开始的 `count` 个值数组移动到 `dst` 处，就像编写了一个 `read` 和 `write` 调用循环以一次性移动它们一样。调用之前目标内存必须是未初始化的，调用之后源内存要保持未初始化状态。`src` 参数和 `dest` 参数必须是 `*const T` 裸指针和 `*mut T` 裸指针，并且 `count` 必须是 `usize`。

`ptr.copy_to(dst, count)` (复制到)

一个更方便的 `copy` 版本，它会将内存中从 `ptr` 开始的 `count` 个值的数组转移给 `dst`，而不用以其起点作为参数。

`std::ptr::copy_nonoverlapping(src, dst, count)`
(复制，无重叠版)

就像对 `copy` 的类似调用一样，但是它的契约进一步要求源内存块和目标内存块不能重叠。这可能比调用 `copy` 略微快一些。

`ptr.copy_to_nonoverlapping(dst, count)` (复制到，无重叠版)

一个更方便的 `copy_nonoverlapping` 版本，就像 `copy_to`。

还有另外两组 `read` 函数和 `write` 函数，它们也位于 `std::ptr` 模块中。

`read_unaligned` (读取，未对齐版) 和 `write_unaligned`
(写入，未对齐版)

与 `read` 和 `write` 类似，但是这两个函数的指针不需要像引用目标类型通常要求的那样对齐。它们可能比普通的 `read` 函数和 `write` 函数慢一点儿。

`read_volatile` (读取，易变版) 和 `write_volatile` (写入，易变版)

这两个函数对应于 C 或 C++ 中的易变 (`volatile`) 读取和易变写入。

22.8.7 示例：GapBuffer

下面是一个使用刚刚讲过的裸指针函数的示例。

假设你正在编写一个文本编辑器，并且正在寻找一种类型来表示文本。可以选择 `String` 并使用 `insert` 方法和 `remove` 方法在用户键入时插入字符和移除字符。但是如果在一个大文件的开头编辑文本，则这些方法可能开销会很高：插入新字符需要在内存中将整个字符串的其余部分都移到右侧，而删除则要将其全部移回左侧。你希望此类常见操作的开销低一些。

Emacs 文本编辑器使用了一种称为**间隙缓冲区**的简单数据结构，该数据结构可以在恒定时间内插入字符和删除字符。`String` 会将其所有空闲容量保留在文本的末尾，这使得 `push` 和 `pop` 的开销变得更低，而间隙缓冲区会将其空闲容量保留在文本中间，即正在进行编辑的位置。这种空闲容量称为**间隙**。在间隙处插入元素或删除元素的开销很低，只要根据需要缩小或扩大间隙即可。可以通过将文本从间隙的一侧移动到另一侧，来让间隙移动到你喜欢的任何位置。当间隙为空时，就迁移到更大的缓冲区。

虽然间隙缓冲区中的插入和删除速度很快，但如果想更改这些操作发生的位置就要将间隙移动到新位置。移动元素需要的时间与移动的距离成正比。幸运的是，典型的编辑活动通常都会在转移到别处之前，在缓冲区的临近区域中进行一系列更改。

本节将在 `Rust` 中实现间隙缓冲区。为了避免被 UTF-8 分散注意力，我们会让该缓冲区直接存储 `char` 值，但即使以其他形式存储文本，这些操作的原则也是一样的。

首先，我们会展示间隙缓冲区的实际应用。下列代码会创建一个 `GapBuffer`，在其中插入一些文本，然后将插入点移动到最后一个单词之前：

```
let mut buf = GapBuffer::new();
buf.insert_iter("Lord of the Rings".chars());
buf.set_position(12);
```

运行上述代码后，缓冲区如图 22-6 所示。

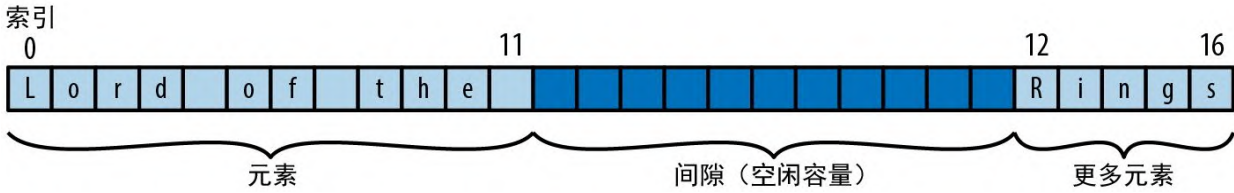


图 22-6：包含一些文本的间隙缓冲区

插入就是要用新文本填补间隙。下面这段代码添加了一个单词并破坏了原句要表达的意思：

```
buf.insert_iter("Onion ".chars());
```

这会导致如图 22-7 所示的状态。

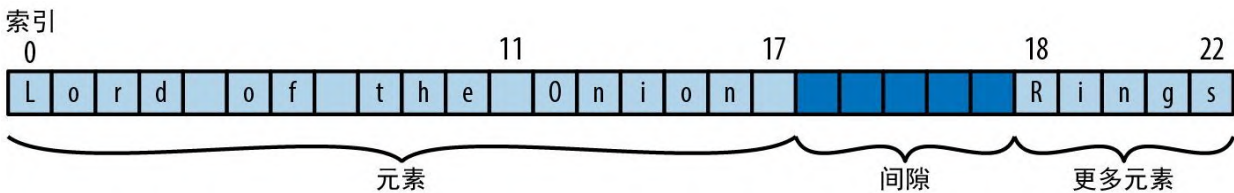


图 22-7：包含更多文本的间隙缓冲区

下面是我们的 GapBuffer 类型：

```
use std;
use std::ops::Range;

pub struct GapBuffer<T> {
    // 元素的存储区。这个存储区具有我们需要的容量，但它的长度始终保持为0。
    // GapBuffer会将其元素和间隙放入此`Vec`的“未使用”容量中
    storage: Vec<T>,

    // `storage`中间未初始化元素的范围
    // 这个范围前后的元素始终是已初始化的
    gap: Range<usize>
}
```

GapBuffer 会以一种奇怪的方式使用它的 storage 字段。⁴它实际上从未在向量中存储任何元素（不过这么说也不太准确），而只是简单地调用 Vec::with_capacity(n) 来获取一块足够大的内存以容

纳 `n` 值，通过向量的 `as_ptr` 方法和 `as_mut_ptr` 方法获得指向该内存的裸指针，然后直接将该缓冲区用于自己的目的。向量的长度始终保持为 0。当 `Vec` 被丢弃时，`Vec` 不会尝试释放自己的元素（因为它不认为自己有任何元素），而只会释放内存块。这正是 `GapBuffer` 想要的行为，它有自己的 `Drop` 实现，知道有效元素在哪里并能正确地丢弃它们。

⁴使用编译器内部 `alloc crate` 中的 `RawVec` 类型能更好地处理这个问题，但那个 `crate` 还不稳定。

`GapBuffer` 中最简单的方法正如你所预期的：

```
impl<T> GapBuffer<T> {
    pub fn new() -> GapBuffer<T> {
        GapBuffer { storage: Vec::new(), gap: 0..0 }
    }

    /// 返回在不重新分配的情况下这个GapBuffer可以容纳的元素数
    pub fn capacity(&self) -> usize {
        self.storage.capacity()
    }

    /// 返回这个GapBuffer当前包含的元素数
    pub fn len(&self) -> usize {
        self.capacity() - self.gap.len()
    }

    /// 返回当前插入点
    pub fn position(&self) -> usize {
        self.gap.start
    }

    ...
}
```

它为后面的很多函数提供了一个工具方法，简化了那些函数的实现。该工具方法会返回指向缓冲区中给定索引处元素的裸指针。为了满足 Rust 的要求，需要为 `mut` 指针和 `const` 指针分别定义一个方法。与前面的方法不同，这些方法都不是公共的。继续看这个 `impl` 块：

```

/// 返回底层存储中第`index`个元素的指针，不考虑间隙
///
/// 安全性：`index`必须是`self.storage`中的有效索引
unsafe fn space(&self, index: usize) -> *const T {
    self.storage.as_ptr().offset(index as isize)
}

/// 返回底层存储中第`index`个元素的可变指针，不考虑间隙
///
/// 安全性：`index`必须是`self.storage`中的有效索引
unsafe fn space_mut(&mut self, index: usize) -> *mut T {
    self.storage.as_mut_ptr().offset(index as isize)
}

```

要找到给定索引处的元素，就必须考虑该索引是落在间隙之前还是之后，并适当调整：

```

/// 返回缓冲区中第`index`个元素的偏移量，并将间隙考虑在内。
/// 这个方法不检查索引是否在范围内，但永远不会返回间隙中的索引
fn index_to_raw(&self, index: usize) -> usize {
    if index < self.gap.start {
        index
    } else {
        index + self.gap.len()
    }
}

/// 返回对第`index`个元素的引用，如果`index`超出了范围，则返回`None`
pub fn get(&self, index: usize) -> Option<&T> {
    let raw = self.index_to_raw(index);
    if raw < self.capacity() {
        unsafe {
            // 刚刚针对self.capacity()检查过`raw`，而index_to_raw
            // 跳过了间隙，所以这是安全的
            Some(&*self.space(raw))
        }
    } else {
        None
    }
}

```

当开始在缓冲区的不同部分进行插入和删除时，需要将间隙移动到新位置。向右移动间隙就要向左移动元素，反之亦然，这就像水平仪中的气泡向一个方向移动而液体会向另一个方向移动一样：

```

/// 将当前插入点设置为`pos`。如果`pos`越界，就panic
pub fn set_position(&mut self, pos: usize) {
    if pos > self.len() {
        panic!("index {} out of range for GapBuffer", pos);
    }

    unsafe {
        let gap = self.gap.clone();
        if pos > gap.start {
            // `pos` 位于间隙之后。通过将间隙之后的元素移动到间隙之前来向右移动间隙

            let distance = pos - gap.start;
            std::ptr::copy(self.space(gap.end),
                           self.space_mut(gap.start),
                           distance);

        } else if pos < gap.start {
            // `pos` 位于间隙之前。通过将间隙之前的元素移动到间隙之后来向左移动间隙

            let distance = gap.start - pos;
            std::ptr::copy(self.space(pos),
                           self.space_mut(gap.end - distance),
                           distance);

        }

        self.gap = pos .. pos + gap.len();
    }
}

```

这个函数使用 `std::ptr::copy` 方法来平移元素，`copy` 要求目标是未初始化的并且会让源保持未初始化。源和目标范围可以重叠，`copy` 会正确处理这种情况。由于间隙是调用前尚未初始化的内存，而这个函数会调整间隙的位置以覆盖 `copy` 腾出的空间，因此可以满足 `copy` 函数的契约。

元素的插入和移除都比较简单。插入会从间隙中为新元素占用一个空间，而移除会将值移出并扩大间隙以覆盖此值曾占据的空间：

```

/// 在当前插入点插入`elt`，并在插入后把插入点后移
pub fn insert(&mut self, elt: T) {
    if self.gap.len() == 0 {
        self.enlarge_gap();
    }

    unsafe {

```



```

        let index = self.gap.start;
        std::ptr::write(self.space_mut(index), elt);
    }
    self.gap.start += 1;
}

/// 在当前插入点插入`iter`生成的元素，并在插入后把插入点后移
pub fn insert_iter<I>(&mut self, iterable: I)
    where I: IntoIterator<Item=T>
{
    for item in iterable {
        self.insert(item)
    }
}

/// 删除插入点之后的元素并返回它，如果插入点位于GapBuffer的末尾，则返回`None`
pub fn remove(&mut self) -> Option<T> {
    if self.gap.end == self.capacity() {
        return None;
    }

    let element = unsafe {
        std::ptr::read(self.space(self.gap.end))
    };
    self.gap.end += 1;
    Some(element)
}

```

与 Vec 使用 `std::ptr::write` 进行 push 和使用 `std::ptr::read` 进行 pop 的方式类似，GapBuffer 使用 `write` 进行 insert，使用 `read` 进行 remove。与 Vec 必须调整其长度以维持已初始化元素和空闲容量之间的边界一样，GapBuffer 也会调整其间隙。

填补此间隙后，insert 方法必须扩大缓冲区以获得更多可用空间。enlarge_gap 方法（impl 块中的最后一个）会处理这个问题：

```

/// 将`self.storage`的容量翻倍
fn enlarge_gap(&mut self) {
    let mut new_capacity = self.capacity() * 2;
    if new_capacity == 0 {
        // 现有向量是空的
        // 选择一个合理的初始容量
        new_capacity = 4;
    }
}

```

```

    }

    // 我们不知道调整vec的大小会对其“（表面看）未使用的”容量
    // 有何影响，所以只好创建一个新向量并把元素移了过去
    let mut new = Vec::with_capacity(new_capacity);
    let after_gap = self.capacity() - self.gap.end;
    let new_gap = self.gap.start .. new.capacity() - after_gap;

    unsafe {
        // 移动位于此间隙之前的元素
        std::ptr::copy_nonoverlapping(self.space(0),
                                      new.as_mut_ptr(),
                                      self.gap.start);

        // 移动位于此间隙之后的元素
        let new_gap_end = new.as_mut_ptr().offset(new_gap.end as
isize);
        std::ptr::copy_nonoverlapping(self.space(self.gap.end),
                                      new_gap_end,
                                      after_gap);
    }

    // 这会释放旧的vec，但不会丢弃任何元素，因为此vec的长度为0
    self.storage = new;
    self.gap = new_gap;
}

```

set_position 必须使用 copy 在间隙中来回移动元素，enlarge_gap 则可以使用 copy_nonoverlapping，因为它会将元素移动到一个全新的缓冲区。

将新向量转移给 self.storage 会丢弃旧向量。由于旧向量的长度为 0，它认为自己没有要丢弃的元素，因此只释放了自己的缓冲区。巧妙的是，copy_nonoverlapping 也有把源变成未初始化状态的语义，因此旧向量的做法恰巧是正确的：现在所有元素都归新向量所有了。

最后，需要确保丢弃 GapBuffer 也会丢弃它的所有元素：

```

impl<T> Drop for GapBuffer<T> {
    fn drop(&mut self) {
        unsafe {
            for i in 0 .. self.gap.start {

```

```

        std::ptr::drop_in_place(self.space_mut(i));
    }
    for i in self.gap.end .. self.capacity() {
        std::ptr::drop_in_place(self.space_mut(i));
    }
}
}
}

```

这些元素都位于间隙前后，因此需要遍历每个区域并使用 `std::ptr::drop_in_place` 函数丢弃每个元素。`drop_in_place` 函数是一个行为类似于 `drop(std::ptr::read(ptr))` 的实用程序，但不会“费心”地将值转移给其调用者（因此适用于无固定大小类型）。就像在 `enlarge_gap` 中一样，当向量 `self.storage` 被丢弃时，它的缓冲区实际上是未初始化的。

与本章展示过的其他类型一样，`GapBuffer` 会确保自己的不变条件足以遵守所使用的每个不安全特性的契约，因此它的所有公共方法都不需要标记为不安全。`GapBuffer` 为无法用安全代码高效编写的特性实现了一个安全的接口。

22.8.8 不安全代码中的 panic 安全性

在 Rust 中，`panic` 通常不会导致未定义行为，`panic!` 宏并不是不安全特性。但是，当你决定使用不安全代码时，就得考虑 `panic` 安全性的问题了。

考虑 22.8.7 节中的 `GapBuffer::remove` 方法：

```

pub fn remove(&mut self) -> Option<T> {
    if self.gap.end == self.capacity() {
        return None;
    }

    let element = unsafe {
        std::ptr::read(self.space(self.gap.end))
    };
    self.gap.end += 1;
}

```

```
    Some(element)
}
```

对 `read` 的调用会将紧随间隙之后的元素移出缓冲区，留下未初始化的空间。此时，`GapBuffer` 处于不一致状态：我们打破了间隙外的所有元素都必须是初始化的这个不变条件。幸运的是，下一条语句扩大了间隙以覆盖这个空间，因此当我们返回时，不变条件会再次成立。

但是请考虑一下，如果在调用 `read` 之后、调整 `self.gap.end` 之前，此代码尝试使用可能引发 `panic` 的特性（如索引切片），那么会发生什么呢？在这两个操作之间的任何地方突然退出该方法都会使 `GapBuffer` 在间隙外留下未初始化的元素。下一次调用 `remove` 可能会尝试再次读取（`read`）它，即使仅仅丢弃 `GapBuffer` 也会尝试运行其 `drop` 方法。这两者都是未定义行为，因为它们会访问未初始化内存。

类型的方法在执行工作时几乎不可避免地会暂时放松类型的不变条件，然后在返回之前让其回到正轨。方法中间出现的 `panic` 可能会中断清理过程，使类型处于不一致状态。

如果类型只使用安全代码，那么这种不一致可能会使类型行为诡异，但并不会引入未定义行为。不过使用不安全特性的代码通常会依赖其不变条件来满足这些特性的契约。破坏不变条件会导致契约破损，从而导致未定义行为。

使用不安全特性时，必须特别注意识别这些暂时放松了不变条件的敏感代码区域，并确保它们不会执行任何可能引起 `panic` 的事情。

22.9 用联合体重新解释内存

虽然 Rust 提供了许多有用的抽象，但最终我们编写的软件只是在操纵字节。联合体是 Rust 最强大的特性之一，用于操纵这些字节并选择如何解释它们。例如，任何 32 位（4 字节）的集合都可以解释为整数或浮点数。任何一种解释都是有效的，不过，将一种数据解释为另一种数据可能会导致其失去意义。

下面是一个用来表示可解释为整数或浮点数的字节集合的联合体：

```
union FloatOrInt {  
    f: f32,  
    i: i32,  
}
```

这是一个包含两个字段（`f` 和 `i`）的联合体。这两个字段可以像结构体的字段一样被赋值，但在构造联合体时，只能选择一个字段，这与结构体不同。结构体的字段会引用内存中的不同位置，而联合体的字段会引用相同位序列的不同解释。赋值给不同的字段只是意味着根据适当的类型覆盖这些位中的一部分或全部。在下面的代码中，`one` 指向的是单个 32 位内存范围，它首先存储一个按简单整数编码的 1，然后存储一个按 IEEE 754 浮点数编码的 1.0。一旦写入了 `f`，先前写入的 `FloatOrInt` 值就会被覆盖：

```
let mut one = FloatOrInt { i: 1 };  
assert_eq!(unsafe { one.i }, 0x00_00_00_01);  
one.f = 1.0;  
assert_eq!(unsafe { one.i }, 0x3F_80_00_00);
```

出于同样的原因，联合体的大小会由其最大字段决定。例如，下面这个联合体的大小为 64 位，虽然 `SmallOrLarge::s` 只是一个 `bool`：

```
union SmallOrLarge {  
    s: bool,  
    l: u64  
}
```

虽然构建联合体或对它的字段赋值是完全安全的，但读取联合体的任何字段都是不安全的：

```
let u = SmallOrLarge { l: 1337 };
println!("{}", unsafe {u.l}); // 打印出1337
```

这是因为与枚举不同，联合体没有标签。编译器不会添加额外的位来区分各种变体。在运行期无法判断 `SmallOrLarge` 是要该解释为 `u64` 还是 `bool`，除非程序有一些额外的上下文。

同时，并没有什么内置手段可以保证给定字段的位模式是有效的。例如，写入 `SmallOrLarge` 值的 `l` 字段将覆盖其 `s` 字段，但它创建的这个位模式并无任何用处，甚至可能都不是有效的 `bool`。因此，虽然写入联合体字段是安全的，但每次读取都需要 `unsafe` 代码。仅当 `s` 字段的各个位可以形成有效的 `bool` 时才允许从 `u.s` 读取，否则，这就是未定义行为。

只要把这些限制牢记在心，联合体仍然可以成为临时重新解释某些数据的有用方法，尤其是在针对值的表观而非值本身进行计算时。例如，前面提到的 `FloatOrInt` 类型可以轻松地打印出浮点数的各个位——即便 `f32` 没有实现过 `Binary` 格式化程序：

```
let float = FloatOrInt { f: 31337.0 };
// 打印出1000110111101001101001000000000
println!("{}", unsafe { float.i });
```

虽然几乎可以肯定这些简单示例会在任何版本的编译器上如预期般工作，但并不能保证任何字段都从特定位置开始，除非将某个属性添加到 `union` 定义中，告诉编译器如何在内存中排布数据。添加属性 `#[repr(C)]` 可以保证所有字段都从偏移量 `0` 而不是编译器喜欢的任何位置开始。有了这个保证，这种改写行为就可以用来提取像整数的符号位这样的单独二进制位了：

```
#[repr(C)]
union SignExtractor {
    value: i64,
```

```
        bytes: [u8; 8]
    }

fn sign(int: i64) -> bool {
    let se = SignExtractor { value: int };
    println!("{:b} ({:?})", unsafe { se.value }, unsafe { se.bytes
});
    unsafe { se.bytes[7] >= 0b10000000 }
}

assert_eq!(sign(-1), true);
assert_eq!(sign(1), false);
assert_eq!(sign(i64::MAX), false);
assert_eq!(sign(i64::MIN), true);
```

在这里，符号位是最高有效字节的最高有效位。因为 x86 处理器是小端（低位在前）的，所以这些字节的顺序是相反的，其最高有效字节不是 `bytes[0]`，而是 `bytes[7]`。通常，这不是 Rust 代码必须处理的事情，但是因为这段代码要直接与 i64 的内存中表示法打交道，所以这些底层细节就变得很重要了。

因为不知道该如何丢弃其内容，所以联合体的所有字段都必须是可 Copy 的。但是，如果必须在联合体中存储一个 `String`，那么也有相应的解决方案，详情请参阅 `std::mem::ManuallyDrop` 的标准库文档。

22.10 匹配联合体

在 Rust 联合体上进行匹配和在结构体上匹配类似，但每个模式必须指定一个字段：

```
unsafe {
    match u {
        SmallOrLarge { s: true } => { println!("boolean true"); }
        SmallOrLarge { l: 2 } => { println!("integer 2"); }
        _ => { println!("something else"); }
    }
}
```

与联合体变体匹配但不指定值的 `match` 分支永远都会成功。如果 `u` 的最后一个写入字段是 `u.i`，则以下代码将导致未定义行为：

```
// 未定义行为！
unsafe {
    match u {
        FloatOrInt { f } => { println!("float {}", f) },
        // 警告：无法抵达的模式
        FloatOrInt { i } => { println!("int {}", i) }
    }
}
```


22.11 借用联合体

借用联合体的一个字段就是借用整个联合体。这意味着，按照正常的借用规则，将一个字段作为可变借用会排斥对该字段或其他字段的任何其他借用，而将一个字段作为不可变借用则意味着对任何字段都不能再进行可变借用。

正如我们将在第 23 章中看到的，Rust 不仅可以帮你为自己的不安全代码构建出安全接口，还可以为用其他语言编写的代码构建出安全接口。从字面来看，“不安全”是充满危险的，但如果谨慎使用，那么也可以构建出高性能代码，同时还能让 Rust 程序员继续享有安全感。

第 23 章 外部函数

赛博空间。难以想象的复杂性。光线交织在思维的非空间、数据的集群和星座中。就像城市的灯光，渐行渐远……

——《神经漫游者》，William Gibson

很遗憾，世界上并不是每个程序都是用 Rust 编写的。我们希望能够 在 Rust 程序中使用以其他语言实现的关键库和接口。Rust 的**外部函数接口**（foreign function interface, FFI）允许 Rust 代码调用以 C 语言编写的（在某些情况下是用 C++ 编写的）函数。由于大多数操作系统提供了 C 接口，因此 Rust 的外部函数接口能帮你立即访问各种底层设施。

在本章中，我们将编写一个与 libgit2 链接的程序，libgit2 是用于 Git 版本控制系统的 C 库。我们将先展示如何借助第 22 章演示的不安全特性在 Rust 中直接使用 C 函数，然后再展示如何构建 libgit2 的安全接口，这些灵感都来自开源的 git2-rs crate。

假设你已经熟悉 C 以及编译和链接 C 程序的机制。如果你擅长使用 C++，那么情况也一样。假设你对 Git 版本控制系统也比较熟悉。

确实存在用于和许多其他语言（包括 Python、JavaScript、Lua 和 Java）通信的 Rust crate。由于篇幅所限，本书就不详细介绍了，但归根结底，所有这些接口都是用 C 的外部函数接口构建的，因此无论要使用哪种语言，本章都会为你提供一个良好的开端。

23.1 寻找共同的数据表示

Rust 和 C 的共同点在于它们都是面向机器的语言，因此为了预测 C 代码的 Rust 值是什么样的（反之亦然），就需要考虑它们的机器层面表示。本书重点展示了值在内存中的实际表示方式，因此你可能已经注意到 C 和 Rust 的数据世界有很多共同点。例如，Rust 的 `usize` 和 C 的 `size_t` 是一样的，而结构体在两种语言中基本上是相同的概念。为了建立 Rust 和 C 类型之间的对应关系，我们将从原始类型开始，然后逐步处理更复杂的类型。

鉴于 C 语言主要用于系统编程，它对类型的表示形式总是出奇地宽松。例如，`int` 通常为 32 位长，但可以更长，也可以短至 16 位；C `char` 可以是有符号的也可以是无符号的。为了应对这种可变性，Rust 的 `std::os::raw` 模块定义了一组保证与某些 C 类型具有相同表示形式的 Rust 类型（参见表 23-1）。这些涵盖了原始整数和字符类型。

表 23-1: Rust 中的 `std::os::raw` 类型

C 类型	相应的 <code>std::os::raw</code> 类型
<code>short</code>	<code>c_short</code>
<code>int</code>	<code>c_int</code>
<code>long</code>	<code>c_long</code>
<code>long long</code>	<code>c_longlong</code>
<code>unsigned short</code>	<code>c_ushort</code>

C 类型	相应的 <code>std::os::raw</code> 类型
<code>unsigned, unsigned int</code>	<code>c_uint</code>
<code>unsigned long</code>	<code>c_ulong</code>
<code>unsigned long long</code>	<code>c_ulonglong</code>
<code>char</code>	<code>c_char</code>
<code>signed char</code>	<code>c_schar</code>
<code>unsigned char</code>	<code>c_uchar</code>
<code>float</code>	<code>c_float</code>
<code>double</code>	<code>c_double</code>
<code>void *</code> 、 <code>const void *</code>	<code>*mut c_void</code> 、 <code>*const c_void</code>

下面是关于表 23-1 的一些特别说明。

- 除了 `c_void`，这里所有的 Rust 类型都是一些原始 Rust 类型的别名，比如 `c_char` 是 `i8` 或 `u8`。
- Rust 的 `bool` 等效于 C 或 C++ 的 `bool`。
- Rust 的 32 位 `char` 类型并不对应于 `wchar_t`，`wchar_t` 的宽度和编码会因不同的具体实现而异。Rust 的 32 位 `char` 类型更接近于 C 的 `char32_t` 类型，但该类型的编码仍然不能保证是 Unicode。

- Rust 的原始 `usize` 类型和 `isize` 类型与 C 的 `size_t` 和 `ptrdiff_t` 具有相同的表示。
- C 和 C++ 的指针与 C++ 的引用对应于 Rust 的裸指针类型 `*mut T` 和 `*const T`。
- 从技术上讲，C 标准允许其实现在 Rust 中没有对应类型的表示，比如 36 位整数、有符号值的原码（sign-and-magnitude）表示法¹等。在实践中，对于每个移植了 Rust 的平台，每个常见的 C 整数类型在 Rust 中都有一个匹配项。

¹我们日常见到的是“有符号数的补码表示法”。——译者注

要定义与 C 结构体兼容的 Rust 结构体类型，可以使用 `#[repr(C)]` 属性。将 `#[repr(C)]` 放在结构体定义之上会要求 Rust 结构体中各个字段的内存布局 and C 中类似结构体的内存布局一样。例如，`libgit2` 的 `git2/errors.h` 头文件定义了以下 C 结构体，以提供有关先前报告的错误的详细信息：

```
typedef struct {
    char *message;
    int klass;
} git_error;
```

可以定义具有相同表示的 Rust 类型，如下所示：

```
use std::os::raw::{c_char, c_int};

#[repr(C)]
pub struct git_error {
    pub message: *const c_char,
    pub klass: c_int
}
```

`#[repr(C)]` 属性只会影响结构体本身的布局，而不会影响其各个字段的表示法，因此要匹配 C 结构体，每个字段也必须使用类 C 类型，比如把 `char *` 表示为 `*const c_char`，把 `int` 表示为 `c_int`。

在这种情况下，`#[repr(C)]` 属性可能不会更改 `git_error` 的布局。确实没有几种有趣的方式来排布指针和整数。但是 C 和 C++ 会保证结构体的成员按照声明的顺序出现在内存中，且每个成员都在不同的地址中，而 Rust 会重新排序字段以将结构体的整体大小最小化，并让零大小的类型不占用空间。`#[repr(C)]` 属性告诉 Rust 要遵循 C 对给定类型的排布规则。

还可以使用 `#[repr(C)]` 来控制 C 风格枚举的表示法：

```
#[repr(C)]
#[allow(non_camel_case_types)]
enum git_error_code {
    GIT_OK          = 0,
    GIT_ERROR       = -1,
    GIT_ENOTFOUND   = -3,
    GIT_EEXISTS     = -4,
    ...
}
```

通常，Rust 在选择枚举的表示方式时会使用各种小技巧。例如，我们曾提到 Rust 用于将 `Option<T>` 存储在单个机器字（如果 T 是固定大小的话）中的技巧。如果没有 `#[repr(C)]`，那么 Rust 就会使用单字节来表示 `git_error_code` 枚举。但如果加了 `#[repr(C)]`，则 Rust 会向 C 看齐，使用 C 中 `int` 大小的值。

还可以要求 Rust 为枚举提供与某些整数类型相同的表示形式。如果前面的定义以 `#[repr(i16)]` 开始，那么它将为你提供一个 16 位类型，其表示形式与以下 C++ 枚举相同：

```
#include <stdint.h>

enum git_error_code: int16_t {
    GIT_OK          = 0,
    GIT_ERROR       = -1,
    GIT_ENOTFOUND   = -3,
    GIT_EEXISTS     = -4,
    ...
};
```

如前所述，`#[repr(C)]` 也适用于联合体。`#[repr(C)]` 联合体的字段总是从联合体的内存的第一位开始（索引为 0）。

假设你有一个 C 结构体，它会使用联合体来保存一些数据和一个标记值，以指示应该使用联合体的哪个字段，就像 Rust 枚举一样。

```
enum tag {
    FLOAT = 0,
    INT    = 1,
};

union number {
    float f;
    short i;
};

struct tagged_number {
    tag t;
    number n;
};
```

通过将 `#[repr(C)]` 应用于枚举、结构体和联合体类型，并使用 `match` 语句根据标记在更大的结构体中选择一个联合体字段，Rust 代码可以与这个结构体进行互操作：

```
#[repr(C)]
enum Tag {
    Float = 0,
    Int = 1
}

#[repr(C)]
union FloatOrInt {
    f: f32,
    i: i32,
}

#[repr(C)]
struct Value {
    tag: Tag,
    union: FloatOrInt
}
```

```
fn is_zero(v: Value) -> bool {
    use self::Tag::*;
    unsafe {
        match v {
            Value { tag: Int, union: FloatOrInt { i: 0 } } =>
true,
            Value { tag: Float, union: FloatOrInt { f: num } } =>
(num == 0.0),
            _ => false
        }
    }
}
```

通过这种技术，即使再复杂的结构也可以轻松地跨越 FFI 边界使用。

在 Rust 和 C 之间传递字符串有点儿困难。C 会将字符串表示为指向字符数组的指针，以空字符结尾。而 Rust 会显式地存储字符串的长度，或是作为 `String` 的字段，或是作为引用 `&str` 的第二个机器字。Rust 字符串不是以空字符结尾的，事实上，它们的内容中可能包含空字符，就像任何其他字符一样。

这意味着不能将 Rust 字符串借用为 C 字符串：如果给 C 代码传入一个 Rust 字符串的指针，那它可能会将嵌入的空字符误认为是字符串的末尾，或者跑到字符串末尾寻找不存在的终止符 `null`。但反过来，你可以借用一个 C 字符串作为 Rust 的 `&str`，只要其内容是格式良好的 UTF-8。

这种情况迫使 Rust 将 C 字符串视为与 `String` 和 `&str` 完全不同的类型。在 `std::ffi` 模块中，`CString` 类型和 `CStr` 类型表示拥有和借用的以空字符结尾的字节数组。与 `String` 和 `str` 相比，`CString` 和 `CStr` 上的方法非常有限，仅限于构造和转换为其他类型。23.2 节会展示这些类型的作用。

23.2 声明外部函数与变量

`extern` 块会声明要在最终 Rust 可执行文件链接的其他库中定义的函数或变量。例如，在大多数平台上，每个 Rust 程序都会链接到标准 C 库，因此可以告诉 Rust，C 库的 `strlen` 函数是这样的：

```
use std::os::raw::c_char;

extern {
    fn strlen(s: *const c_char) -> usize;
}
```

这为 Rust 提供了函数的名称和类型，同时将其定义留待稍后再链接。

Rust 假定在 `extern` 块内声明的函数会使用 C 调用约定来传递参数和接受返回值。这些函数被定义为 `unsafe` 函数。对于 `strlen`，这是正确的选择：它确实是一个 C 函数，C 语言规范要求你向它传递一个指向具有正确结束符的有效字符串指针，而这是 Rust 无法强制执行的契约。（几乎任何接受裸指针的函数都必须是 `unsafe` 的：安全的 Rust 可以从任意整数构造出裸指针，但解引用这样的指针是未定义行为。）

有了这个 `extern` 块，就可以像调用任何其他 Rust 函数一样调用 `strlen` 了，尽管其类型暴露出它只是一个外来者：

```
use std::ffi::CString;

let rust_str = "I'll be back";
let null_terminated = CString::new(rust_str).unwrap();
unsafe {
    assert_eq!(strlen(null_terminated.as_ptr()), 12);
}
```

`CString::new` 函数会构建一个以 `null` 结尾的 C 字符串。它会首先检查其参数是否有嵌入的空字符，因为这些嵌入的空字符不能用 C 字符串表示，如果发现任何字符就返回一个错误（因此需要

unwrap 其结果)。否则，它会在末尾添加一个空字节并返回一个拥有结果字符的 CString。

CString::new 的开销取决于你传入的类型。它可以接受任何实现了 Into<Vec<u8>> 的值。传递 &str 需要一次内存分配和一次复制，因为转换为 Vec<u8> 会让向量拥有一个分配在堆上的字符串副本。但是按值传递 String 只会消耗此字符串并接管其缓冲区，因此除非为了添加额外的空字符要强制调整缓冲区大小，否则本次转换根本不需要复制文本或分配内存。

CString 会解引用成 CStr，后者的 as_ptr 方法会返回指向字符串开头的 *const c_char。这是 strlen 期望的类型。在这个例子中，对字符串调用 strlen 时，会找到 CString::new 放置在那里的空字符，并返回其按字节计算的长度。

还可以在 extern 块中声明全局变量。POSIX 系统有一个名为 environ 的全局变量，该变量会持有进程的环境变量的值。在 C 中，可以将其声明为如下形式：

```
extern char **environ;
```

在 Rust 中，则要这样写：

```
use std::ffi::CStr;
use std::os::raw::c_char;

extern {
    static environ: *mut *mut c_char;
}
```

要打印环境的第一个元素，可以这样写：

```
unsafe {
    if !environ.is_null() && !(*environ).is_null() {
        let var = CStr::from_ptr(*environ);
        println!("first environment variable: {}",
            var.to_string_lossy())
    }
}
```

```
}  
}
```

在确保 `environ` 存在第一个元素后，代码会调用 `CStr::from_ptr` 来构建出借用它的 `CStr`。`to_string_lossy` 方法会返回一个 `Cow<str>`：如果 C 字符串包含格式良好的 UTF-8，则 `Cow` 会将其内容借用为 `&str`，不包括空字节终止符。否则，`to_string_lossy` 就会在堆中复制文本，用 Unicode 的官方替代字符  替换格式错误的 UTF-8 序列，并从中构建一个拥有型 `Cow`。无论是哪种方式，其结果都会实现 `Display`，因此可以使用 `{}` 格式参数打印它。

23.3 使用库中的函数

要使用特定库提供的函数，可以在 `extern` 块的顶部放置一个 `#[link]` 属性，该属性中的 `name` 是 Rust 应该链接进可执行文件的库。例如，下面这个程序会调用 `libgit2` 的初始化和关闭方法，但不会执行任何其他操作：

```
use std::os::raw::c_int;

#[link(name = "git2")]
extern {
    pub fn git_libgit2_init() -> c_int;
    pub fn git_libgit2_shutdown() -> c_int;
}

fn main() {
    unsafe {
        git_libgit2_init();
        git_libgit2_shutdown();
    }
}
```

`extern` 块还是会像以前一样声明外部函数。`#[link(name = "git2")]` 属性会在 `crate` 中留下一个便签，大意是当 Rust 创建最终的可执行文件或共享库时，应该链接到 `git2` 库。Rust 会使用系统的链接器来构建可执行文件。在 Unix 上，它会在链接器命令上传入参数 `-lgit2`；在 Windows 上，它会传入 `git2.LIB`。

`#[link]` 属性也适用于库 `crate`。在构建依赖于其他 `crate` 的程序时，Cargo 会收集整个依赖图中的 `link` 注解，并将它们全部包含在最终链接中。

在这个示例中，如果你想在自己的机器上进行操作，则需要构建 `libgit2`。我们使用的是 `libgit2` 的 0.25.1 版。要编译 `libgit2`，需要安装 CMake 构建工具和 Python 语言，我们使用的是 CMake 3.8.0 和 Python 2.7.13。

构建 `libgit2` 的完整说明可在其网站上找到，构建过程非常简单，我们将在此处展示其基本内容。在 Linux 上，假设你已经将库的源代码解压到目录 `/home/jimb/libgit2-0.25.1` 中：

```
$ cd /home/jimb/libgit2-0.25.1
$ mkdir build
$ cd build
$ cmake ..
$ cmake --build .
```

在 Linux 上，这会生成一个共享库 `/home/jimb/libgit2-0.25.1/build/libgit2.so.0.25.1`，通常有一堆符号链接（可嵌套）指向它，其中一个名为 `libgit2.so`。在 macOS 上，结果相似，但库名是 `libgit2.dylib`。

在 Windows 上，构建也很简单。假设你已将源代码解压到目录 `C:\Users\JimB\libgit2-0.25.1` 中。在 Visual Studio 命令提示符中执行如下命令：

```
> cd C:\Users\JimB\libgit2-0.25.1
> mkdir build
> cd build
> cmake -A x64 ..
> cmake --build .
```

这些命令与在 Linux 上使用的命令相同，只是在第一次运行 CMake 时必须请求进行 64 位构建以匹配你的 Rust 编译器。（如果安装的是 32 位 Rust 工具链，则应该省略第一个 `cmake` 命令的 `-A x64` 标志。）这会在 `C:\Users\JimB\libgit2-0.25.1\build\Debug` 目录中生成一个导入库 `git2.LIB` 和一个动态链接库 `git2.DLL`。（其余说明只针对 Unix，只有在 Windows 上有显著差异的地方才会特别说明。）

在一个单独的目录中创建 Rust 程序：

```
$ cd /home/jimb
$ cargo new --bin git-toy
    Created binary (application) `git-toy` package
```

将本节开头处的代码放入 `src/main.rs` 中。当然，如果现在尝试构建它，则 Rust 不知道去哪里找你刚构建的 `libgit2`：

```
$ cd git-toy
$ cargo run
   Compiling git-toy v0.1.0 (/home/jimb/git-toy)
error: linking with `cc` failed: exit status: 1
 |
 = note: /usr/bin/ld: error: cannot find -lgit2
       src/main.rs:11: error: undefined reference to
'git_libgit2_init'
       src/main.rs:12: error: undefined reference to
'git_libgit2_shutdown'
       collect2: error: ld returned 1 exit status

error: could not compile `git-toy` due to previous error
```

可以通过编写构建脚本来告诉 Rust 在何处搜索库，这是 Cargo 在构建时会编译和运行的 Rust 代码。构建脚本可以做各种各样的事情，比如动态生成代码、编译 C 代码并将其包含在 `crate` 中，等等。在这个例子中，只需将库的搜索路径添加到可执行文件的链接命令中即可。Cargo 在运行构建脚本时会解析构建脚本的输出以获取此类信息，因此构建脚本只需将正确的关键信息打印到其标准输出即可。

要创建构建脚本，请将名为 `build.rs` 的文件添加到 `Cargo.toml` 文件的所属目录中，其中包含如下内容：

```
fn main() {
    println!(r"cargo:rustc-link-search=native=/home/jimb/libgit2-0.25.1/build");
}
```

这是 Linux 上的正确路径，在 Windows 上，可以将文本 `native=` 后面的路径更改为 `C:\Users\JimB\libgit2-0.25.1\build\Debug`。（为了简化这个示例，我们采取了一些不严谨的写法，在实际的应用程序中，应该避免在构建脚本中使用绝对路径。本节末尾引用的文档展示了应该如何正确执行此操作。）

现在差不多可以运行程序了。在 macOS 上，程序应该会立即运行，在 Linux 上，你可能会看到如下内容：

```
$ cargo run
   Compiling git-toy v0.1.0 (/tmp/rustbook-transcript-tests/git-toy)
   Finished dev [unoptimized + debuginfo] target(s)
   Running `target/debug/git-toy`
target/debug/git-toy: error while loading shared libraries:
libgit2.so.25: cannot open shared object file: No such file or
directory
```

这意味着尽管 Cargo 成功地将可执行文件链接到了库，但它不知道运行期在哪里可以找到共享库。Windows 会通过弹出对话框报告此故障。在 Linux 上，必须设置 `LD_LIBRARY_PATH` 环境变量：

```
$ export LD_LIBRARY_PATH=/home/jimb/libgit2-0.25.1/build:$LD_LIBRARY_PATH
$ cargo run
   Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
   Running `target/debug/git-toy`
```

在 macOS 上，则要设置 `DYLD_LIBRARY_PATH`。

在 Windows 上，必须设置 `PATH` 环境变量：

```
> set PATH=C:\Users\JimB\libgit2-0.25.1\build\Debug;%PATH%
> cargo run
   Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
   Running `target/debug/git-toy`
>
```

当然，在已部署的应用程序中，你肯定不希望仅仅为了查找库的代码而不得不设置环境变量。有一种替代方案，可以将 C 库静态链接到 crate。这样就可以将库的目标文件复制到 crate 的 `.rlib` 文件中，与 crate 的 Rust 代码的目标文件和元数据放在一起了。然后整个集合会参与最终链接。

Cargo 约定，提供 C 库访问能力的 crate 应该命名为 `LIB-sys`，其中 `LIB` 是 C 库的名称。`-sys` crate 应该只包含静态链接库以及

带有 `extern` 块和类型定义的 Rust 模块。更高级别的接口则属于依赖 `-sys crate` 的那些 `crate`。这样多个上游 `crate` 就可以依赖同一个 `-sys crate`（假设有一个单一版本的 `-sys crate` 可以满足每个人的需求）。

有关 Cargo 对构建脚本的支持以及与系统库相链接的详细信息，请参见 Cargo 在线文档。文档中展示了如何避免在构建脚本中使用绝对路径、控制编译标志、使用 `pkg-config` 等工具，等等。`git2-rs crate` 也提供了很好的模拟示例，它的构建脚本处理了一些复杂的情况。

23.4 libgit2 的裸接口

要正确使用 libgit2，需要弄清楚以下两个问题。

- 在 Rust 中使用 libgit2 函数需要做什么？
- 如何围绕 libgit2 函数构建安全的 Rust 接口？

我们将分别回答这两个问题。在本节中，我们会编写一个程序，该程序本质上是一个巨大的 unsafe 块，其中充满了非惯用的 Rust 代码，这反映了混合语言中固有的类型系统和使用惯例的冲突。我们将这个程序称为裸接口。虽然代码会很乱，但把 Rust 代码使用 libgit2 时必须发生的所有步骤都说清楚了。

然后，在 23.5 节中，我们会构建一个 libgit2 的安全接口，用 Rust 的类型系统来强制执行 libgit2 强加给其用户的规则。幸运的是，libgit2 是一个设计得非常好的 C 库，所以那些由于 Rust 的安全性要求而让我们不得不问的问题都有很好的答案，从而可以构建出没有 unsafe 函数的 Rust 惯用接口。

我们要编写的程序非常简单，它会以一个路径作为命令行参数，打开那里的 Git 存储库，然后打印出其头部提交。虽然此程序很简单，但足以说明构建出安全的 Rust 惯用接口的关键策略。

对于这个裸接口，程序最终用到的 libgit2 中的函数和类型集比我们之前用过的更大，因此比较合理的做法是将 extern 块移动到它自己的模块中。我们将在 git-toy/src 中创建一个名为 raw.rs 的文件，其内容如下：

```
#![allow(non_camel_case_types)]

use std::os::raw::{c_int, c_char, c_uchar};

#[link(name = "git2")]
extern {
    pub fn git_libgit2_init() -> c_int;
    pub fn git_libgit2_shutdown() -> c_int;
```

```

    pub fn giterr_last() -> *const git_error;

    pub fn git_repository_open(out: *mut *mut git_repository,
                               path: *const c_char) -> c_int;
    pub fn git_repository_free(repo: *mut git_repository);

    pub fn git_reference_name_to_id(out: *mut git_oid,
                                     repo: *mut git_repository,
                                     reference: *const c_char) ->
c_int;

    pub fn git_commit_lookup(out: *mut *mut git_commit,
                             repo: *mut git_repository,
                             id: *const git_oid) -> c_int;

    pub fn git_commit_author(commit: *const git_commit) -> *const
git_signature;
    pub fn git_commit_message(commit: *const git_commit) -> *const
c_char;
    pub fn git_commit_free(commit: *mut git_commit);
}

#[repr(C)] pub struct git_repository { _private: [u8; 0] }
#[repr(C)] pub struct git_commit { _private: [u8; 0] }

#[repr(C)]
pub struct git_error {
    pub message: *const c_char,
    pub klass: c_int
}

pub const GIT_OID_RAWSZ: usize = 20;

#[repr(C)]
pub struct git_oid {
    pub id: [c_uchar; GIT_OID_RAWSZ]
}

pub type git_time_t = i64;

#[repr(C)]
pub struct git_time {
    pub time: git_time_t,
    pub offset: c_int
}

#[repr(C)]
pub struct git_signature {

```

```
pub name: *const c_char,  
pub email: *const c_char,  
pub when: git_time  
}
```

这里的每一项都参考了 `libgit2` 自身的头文件中的声明。例如，`libgit2-0.25.1/include/git2/repository.h` 中包含以下声明：

```
extern int git_repository_open(git_repository **out, const char  
*path);
```

这个函数会尝试打开位于 `path` 的 Git 存储库。如果一切顺利，它会创建一个 `git_repository` 对象，并在 `out` 指向的位置存储一个指向它的指针。等效的 Rust 声明如下所示：

```
pub fn git_repository_open(out: *mut *mut git_repository,  
                           path: *const c_char) -> c_int;
```

`libgit2` 公共头文件将 `git_repository` 类型定义成了一个不完整结构体类型的 `typedef`：

```
typedef struct git_repository git_repository;
```

由于这个类型的细节对库来说是私有的，因此公共头文件不会定义 `struct git_repository`，以确保库的用户永远不能自己构建这种类型的实例。Rust 中用来模拟不完整结构体类型的写法如下所示：

```
#[repr(C)] pub struct git_repository { _private: [u8; 0] }
```

这是一个包含无元素数组的结构体类型。由于 `_private` 字段不是 `pub` 的，因此无法在这个模块外构造此类型的值，这完美地映射了一种只能由 `libgit2` 构造出来的 C 类型，并且此类型只能通过裸指针进行操作。

手动编写大型 `extern` 块可能是一件苦差事。如果你正在为复杂的 C 库创建 Rust 接口，可以试试 `bindgen crate`，它可以使用从构建脚本中调用的函数来解析 C 头文件，并自动生成相应的 Rust 声

明。由于篇幅所限，这里就不展示 bindgen 的实际操作了，但是 bindgen 在 crates.io 的页面上有指向其文档的链接。

接下来我们将完全重写 main.rs。首先需要声明 raw 模块：

```
mod raw;
```

根据 libgit2 的约定，可能出错的函数会返回一个整数代码，正数或 0 表示成功，负数表示失败。如果发生错误，giterr_last 函数会返回一个指向 git_error 结构体的指针，以提供关于出错原因的详细信息。libgit2 拥有这个结构体，所以我们不需要自己释放，但它可能会被下一次库调用覆盖。一个合适的 Rust 接口会使用 Result，但在这个裸接口版本中，我们想按原样使用 libgit2 函数，因此必须编写自己的函数来处理这些错误：

```
use std::ffi::CStr;
use std::os::raw::c_int;

fn check(activity: &'static str, status: c_int) -> c_int {
    if status < 0 {
        unsafe {
            let error = &*raw::giterr_last();
            println!("error while {}: {} ({})",
                    activity,
                    CStr::from_ptr(error.message).to_string_lossy(),
                    error.klass);
            std::process::exit(1);
        }
    }

    status
}
```

我们会使用这个函数来检查 libgit2 调用的结果，如下所示：

```
check("initializing library", raw::git_libgit2_init());
```

这里使用了之前使用过的 `CStr` 方法：`from_ptr` 从 C 字符串构造 `CStr`，并使用 `to_string_lossy` 将其转换为 Rust 可以打印的内容。

接下来需要用一个函数来打印出某次提交：

```
unsafe fn show_commit(commit: *const raw::git_commit) {
    let author = raw::git_commit_author(commit);

    let name = CStr::from_ptr((*author).name).to_string_lossy();
    let email = CStr::from_ptr((*author).email).to_string_lossy();
    println!("{}", <{}>\n", name, email);

    let message = raw::git_commit_message(commit);
    println!("{}", CStr::from_ptr(message).to_string_lossy());
}
```

给定一个指向 `git_commit` 的指针，`show_commit` 会调用 `git_commit_author` 和 `git_commit_message` 来检索需要的信息。这两个函数遵循 `libgit2` 文档中解释过的如下契约：

如果函数返回一个对象作为返回值，那么这个函数就是一个 `getter`，并且该对象的生命周期与其父对象相关联。

在 Rust 术语中，`author` 和 `message` 是从 `commit` 借来的：`show_commit` 不需要自己释放它们，但也不能在 `commit` 被释放后保留它们。由于这个 API 使用了裸指针，所以 Rust 不会替我们检查它们的生命周期。换句话说，如果我们不小心创建了悬空指针，那么可能直到程序崩溃时才能发现。

前面的代码假设这些字段包含 UTF-8 文本，这并不总是正确的。Git 也允许其他编码。正确解释这些字符串可能需要使用 `encoding crate`。为简洁起见，这里不会涉及这些问题。

我们程序的 `main` 函数是这样写的：

```
use std::ffi::CString;
use std::mem;
use std::ptr;
```

```

use std::os::raw::c_char;

fn main() {
    let path = std::env::args().skip(1).next()
        .expect("usage: git-toy PATH");
    let path = CString::new(path)
        .expect("path contains null characters");

    unsafe {
        check("initializing library", raw::git_libgit2_init());

        let mut repo = ptr::null_mut();
        check("opening repository",
            raw::git_repository_open(&mut repo, path.as_ptr()));

        let c_name = b"HEAD\0".as_ptr() as *const c_char;
        let oid = {
            let mut oid = mem::MaybeUninit::uninit();
            check("looking up HEAD",
                raw::git_reference_name_to_id(oid.as_mut_ptr(),
repo, c_name));
            oid.assume_init()
        };

        let mut commit = ptr::null_mut();
        check("looking up commit",
            raw::git_commit_lookup(&mut commit, repo, &oid));

        show_commit(commit);

        raw::git_commit_free(commit);

        raw::git_repository_free(repo);

        check("shutting down library",
raw::git_libgit2_shutdown());
    }
}

```

该函数从处理路径参数和初始化库的代码开始，所有这些我们之前都见过。下面是第一段新代码：

```

let mut repo = ptr::null_mut();
check("opening repository",
    raw::git_repository_open(&mut repo, path.as_ptr()));

```

对 `git_repository_open` 的调用会尝试打开位于给定路径的 Git 存储库。如果成功，就会为其分配一个新的 `git_repository` 对象，并将 `repo` 指向该对象。Rust 隐式地将引用转换为裸指针，因此在这里传递 `&mut repo` 会提供此调用所期待的 `*mut *mut git_repository`。

这展示了使用中的另一个 `libgit2` 契约（同样来自 `libgit2` 文档）：

以第一个参数作为“指向指针的指针”所返回的对象，应该由调用者拥有并负责释放。

在 Rust 术语中，像 `git_repository_open` 这样的函数会将新值的所有权传给调用者。

接下来，再来看看负责查找存储库“当前头部提交”的“对象哈希”的代码：

```
let oid = {
    let mut oid = mem::MaybeUninit::uninit();
    check("looking up HEAD",
        raw::git_reference_name_to_id(oid.as_mut_ptr(), repo,
c_name));
    oid.assume_init()
};
```

`git_oid` 类型会存储一个对象标识符——一个 160 位哈希码，Git 在内部及其用户界面中会使用它来标识提交、文件的各个版本等。对 `git_reference_name_to_id` 的调用会为当前 "HEAD" 提交查找对象标识符。

在 C 语言中，通过将指向变量的指针传给用来填充其值的函数来初始化变量是完全正常的。`git_reference_name_to_id` 对它的第一个参数也会这样处理。但是 Rust 不允许借用对未初始化变量的引用。固然可以用 0 来初始化 `oid`，但这是一种浪费，因为存储在那里的任何值都会被简单地覆盖。

可以让 Rust 为我们提供一份未初始化内存，但是因为在任意时刻读取未初始化内存都是瞬时的未定义行为，所以 Rust 提供了一个抽象 `MaybeUninit` 来简化它的使用。`MaybeUninit<T>` 告诉编译器为类型 `T` 留出足够的内存，但如果你认为这样做不安全，就不要碰它。虽然这个内存由 `MaybeUninit` 拥有，但编译器也会避免某些可能导致未定义行为的优化，即使代码中没有任何地方显式访问这块未初始化内存也会如此。

`MaybeUninit` 提供了一个 `as_mut_ptr()` 方法，该方法会生成一个 `*mut T` 指针以指向它包装的可能未初始化内存。通过将该指针传给可初始化这块内存的外部函数，然后调用 `assume_init`，`MaybeUninit` 就会生成完全初始化过的 `T`，这样就可以避免未定义行为，而无须支付先初始化然后立即丢弃值带来的额外开销。`assume_init` 是不安全的，因为在无法确定内存是否真正被初始化过的情况下在 `MaybeUninit` 上调用它会立即导致未定义行为。

在这个例子中，`assume_init` 是安全的，因为 `git_reference_name_to_id` 已经初始化了 `MaybeUninit` 拥有的内存。也可以将 `MaybeUninit` 用于 `repo` 变量和 `commit` 变量，但由于它们只是单个机器字，因此我们不需要这样做，只需将它们初始化为空指针即可：

```
let mut commit = ptr::null_mut();
check("looking up commit",
      raw::git_commit_lookup(&mut commit, repo, &oid));
```

这会获取提交的对象标识符并查找实际的提交，当成功时会把 `git_commit` 指针存储在 `commit` 中。

`main` 函数的其余部分不言自明。它调用之前定义的 `show_commit` 函数，释放提交对象和存储库对象，最后关闭库。

现在我们可以手头任何现有 Git 存储库上试用该程序了。

```
$ cargo run /home/jimb/rbattle
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
```



```
Running `target/debug/git-toy /home/jimb/rbattle`  
Jim Blandy <jimb@red-bean.com>
```

```
Animate goop a bit.
```

23.5 libgit2 的安全接口

libgit2 的裸接口是不安全特性的一个完美示例：当然可以正确使用它（但我们要知道这些规则——就像这里一样），但 Rust 无法强制执行那些你必须遵循的规则。要为像这样的库设计一个安全的 API，就要识别所有这些规则，然后找出一种方法把任何违反这些规则的行为都转化为类型检查错误或借用检查错误。

下面是 libgit2 为程序正确使用其特性而制定的一些规则。

- 在使用任何其他库函数之前，必须调用 `git_libgit2_init`。调用 `git_libgit2_shutdown` 后不得再使用任何库函数。
- 除了输出参数，传给 libgit2 函数的所有值都必须完全初始化。
- 当调用失败时，用于保存调用结果的输出参数将处于未初始化状态，不得使用它们的值。
- `git_commit` 对象引用的是派生出它的 `git_repository` 对象，所以前者的生命周期不能超出后者。（libgit2 文档中并未对此做明确说明，我们从接口中存在某些函数做出推断，然后通过阅读源代码进行验证。）
- 同样，`git_signature` 总是借用自给定的 `git_commit`，前者的生命周期不能超出后者。（文档中确实涵盖了这种情况。）
- 与提交关联的消息以及作者的姓名和电子邮件地址都是从提交中借用的，并且在提交被释放后不得使用。
- libgit2 对象一旦被释放，就不能再使用了。

事实证明，可以通过 Rust 的类型系统或借助内部管理细节来构建强制执行所有这些规则的 libgit2Rust 接口。

在开始之前，先稍微重构一下本项目。我们想要一个导出安全接口的 git 模块，其中来自先前程序的裸接口是一个私有子模块。

整个源代码树结构如下所示：

```
git-toy/  
├─ Cargo.toml  
├─ build.rs  
└─ src/  
    ├─ main.rs  
    └─ git/  
        ├─ mod.rs  
        └─ raw.rs
```

按照 8.2.2 节中解释过的规则，`git` 模块的源代码应出现在 `git/mod.rs` 中，其 `git::raw` 子模块的源代码应出现在 `git/raw.rs` 中。

我们将再一次完全重写 `main.rs`。它应该以 `git` 模块的声明开始：

```
mod git;
```

然后，需要创建 `git` 子目录并将 `raw.rs` 移入其中：

```
$ cd /home/jimb/git-toy  
$ mkdir src/git  
$ mv src/raw.rs src/git/raw.rs
```

`git` 模块需要声明自己的 `raw` 子模块。文件 `src/git/mod.rs` 中必须写出如下内容：

```
mod raw;
```

因为它不是 `pub` 的，所以这个子模块对主程序不可见。

稍后需要使用 `libc crate` 中的一些函数，因此必须在 `Cargo.toml` 中添加依赖项。完整的文件如下所示：

```
[package]  
name = "git-toy"  
version = "0.1.0"  
authors = ["You <you@example.com>"]  
edition = "2021"
```

```
[dependencies]
libc = "0.2"
```

现在我们已经重构了自己的模块，接下来要考虑错误处理。即使是 `libgit2` 的初始化函数也会返回一个错误代码，所以在开始之前需要先解决这个问题。一个符合惯用法的 Rust 接口需要自己的 `Error` 类型，以捕获 `libgit2` 故障代码以及来自 `giterr_last` 的错误消息和类。一个适当的错误类型必须实现通常的 `Error` 特型、`Debug` 特型和 `Display` 特型。然后，它要有自己的使用此 `Error` 类型的 `Result` 类型。下面是 `src/git/mod.rs` 中必要的定义：

```
use std::error;
use std::fmt;
use std::result;

#[derive(Debug)]
pub struct Error {
    code: i32,
    message: String,
    class: i32
}

impl fmt::Display for Error {
    fn fmt(&self, f: &mut fmt::Formatter) -> result::Result<(),
fmt::Error> {
        // 显示`Error`其实就是显示来自libgit2的消息
        self.message.fmt(f)
    }
}

impl error::Error for Error { }

pub type Result<T> = result::Result<T, Error>;
```

要检查原始库调用的结果，该模块需要一个将 `libgit2` 返回码转换为 `Result` 的函数：

```
use std::os::raw::c_int;
use std::ffi::CStr;

fn check(code: c_int) -> Result<c_int> {
    if code >= 0 {
```

```

        return Ok(code);
    }

    unsafe {
        let error = raw::giterr_last();

        // libgit2会确保(*error).message始终不为空且
        // 以空字符结尾，因此这个调用是安全的
        let message = CStr::from_ptr((*error).message)
            .to_string_lossy()
            .into_owned();

        Err(Error {
            code: code as i32,
            message,
            class: (*error).klass as i32
        })
    }
}

```

这个 `check` 函数与裸接口版本的 `check` 函数之间的主要区别在于，它构造了一个 `Error` 值，而不是打印错误消息并立即退出。

现在可以处理 `libgit2` 的初始化了。安全接口会提供一个 `Repository` 类型，表示打开的 `Git` 存储库，具有解析引用、查找提交等方法。下面是 `git/mod.rs` 中的 `Repository` 的定义：

```

/// Git存储库
pub struct Repository {
    // 这必须始终是一个指向有效`git_repository`结构体的指针
    // 不能有任何其他`Repository`型指针指向它
    raw: *mut raw::git_repository
}

```

`Repository` 的 `raw` 字段不是公共的。由于只有这个模块中的代码可以访问 `raw::git_repository` 指针，因此要正确使用这个模块，就应该确保始终正确使用指针。

如果创建 `Repository` 的唯一方法是成功打开新的 `Git` 存储库，那么就可以确保每个 `Repository` 都指向不同的 `git_repository` 对象：

```

use std::path::Path;
use std::ptr;

impl Repository {
    pub fn open<P: AsRef<Path>>(path: P) -> Result<Repository> {
        ensure_initialized();

        let path = path_to_cstring(path.as_ref())?;
        let mut repo = ptr::null_mut();
        unsafe {
            check(raw::git_repository_open(&mut repo,
path.as_ptr()))?;
        }

        Ok(Repository { raw: repo })
    }
}

```

由于使用安全接口执行任何操作的唯一方法就是先获取 `Repository` 值，并且 `Repository::open` 会以调用 `ensure_initialized` 开始，因此我们可以确信 `ensure_initialized` 会在任何 `libgit2` 函数之前被调用。`ensure_initialized` 的定义如下所示：

```

fn ensure_initialized() {
    static ONCE: std::sync::Once = std::sync::Once::new();
    ONCE.call_once(|| {
        unsafe {
            check(raw::git_libgit2_init())
                .expect("initializing libgit2 failed");
            assert_eq!(libc::atexit(shutdown), 0);
        }
    });
}

extern fn shutdown() {
    unsafe {
        if let Err(e) = check(raw::git_libgit2_shutdown()) {
            eprintln!("shutting down libgit2 failed: {}", e);
            std::process::abort();
        }
    }
}

```

`std::sync::Once` 类型有助于以线程安全的方式运行初始化代码。只有第一个调用 `ONCE.call_once` 的线程才会运行给定的闭包。此线程或任何其他线程的任何后续调用都会阻塞，直到第一个调用完成，然后会立即返回，而不会再次运行此闭包。闭包执行完成后再调用 `ONCE.call_once` 开销很低，只需对存储在 `ONCE` 中的标志进行原子化加载即可。

在前面的代码中，初始化闭包会调用 `git_libgit2_init` 并检查结果。它做了一点儿调整，只使用 `expect` 来确保初始化成功，而不会尝试将错误传播回调用者。

为确保程序不会忘记调用 `git_libgit2_shutdown`，初始化闭包中使用了 C 库的 `atexit` 函数，该函数会接受指向函数的指针留待进程退出之前调用。Rust 闭包不能用作 C 函数指针，因为闭包是某种匿名类型的值，携带着它捕获的任何变量的值或对它们的引用，而 C 函数指针只是一个指针。不过，用 Rust 的 `fn` 类型是没问题的，只要将它们声明为 `extern` 以便 Rust 知道要使用 C 调用约定即可。本地函数 `shutdown` 符合要求并会确保 `libgit2` 正确关闭。

7.1.1 节曾提到让 `panic` 跨越语言边界是未定义行为。从 `atexit` 到 `shutdown` 的调用就是这样的一个边界，所以 `shutdown` 不能引发 `panic` 是非常必要的。这就是为什么 `shutdown` 不能简单地使用 `.expect` 来处理 `raw::git_libgit2_shutdown` 报告的错误。相反，它必须报告错误并终止进程本身。POSIX 禁止在 `atexit` 处理程序中调用 `exit`，因此 `shutdown` 会调用 `std::process::abort` 来突然终止程序。

要尽快安排调用 `git_libgit2_shutdown`，比如当最后一个 `Repository` 值被丢弃时。但无论我们如何安排，调用 `git_libgit2_shutdown` 都是此安全 API 的职责。在调用它的那一刻，任何现存的 `libgit2` 对象都会变得无法安全使用，因此安全 API 不能直接对外暴露这个函数。

`Repository` 的裸指针必须始终指向有效的 `git_repository` 对象。这意味着关闭此存储库的唯一方法就是丢弃拥有它的

Repository 值:

```
impl Drop for Repository {
    fn drop(&mut self) {
        unsafe {
            raw::git_repository_free(self.raw);
        }
    }
}
```

通过只在指向 `raw::git_repository` 的唯一指针即将消失时调用 `git_repository_free`, `Repository` 类型还确保了此指针在释放后永远不再使用。

`Repository::open` 方法会使用一个名为 `path_to_cstring` 的私有函数, 该函数有两个定义, 一个用于类 Unix 系统, 一个用于 Windows:

```
use std::ffi::CString;

#[cfg(unix)]
fn path_to_cstring(path: &Path) -> Result<CString> {
    // `as_bytes` 方法仅存在于类Unix系统中
    use std::os::unix::ffi::OsStrExt;

    Ok(CString::new(path.as_os_str().as_bytes())?)
}

#[cfg(windows)]
fn path_to_cstring(path: &Path) -> Result<CString> {
    // 尝试转换为UTF-8。如果失败, 则libgit2无论如何都无法处理该路径
    match path.to_str() {
        Some(s) => Ok(CString::new(s)?),
        None => {
            let message = format!("Couldn't convert path '{}' to UTF-8",
                                   path.display());
            Err(message.into())
        }
    }
}
```


libgit2 接口使这段代码有点儿棘手。在所有平台上，libgit2 都会接受以 null 结尾的 C 字符串作为路径。在 Windows 上，libgit2 会假定这些 C 字符串包含格式良好的 UTF-8，并会在内部将它们转换为 Windows 实际需要的 16 位路径。这通常是有效的，但并不理想。Windows 允许格式不正确的 Unicode 文件名，因此不能用 UTF-8 表示。如果你有这样的文件，则不可能将其名称传给 libgit2。

在 Rust 中，文件系统路径的正确表示是 `std::path::Path`，它是经过精心设计的，可以处理可能出现在 Windows 或 POSIX 上的任何路径。这意味着 Windows 上存在无法传给 libgit2 的 Path 值，因为它们不是格式良好的 UTF-8。因此，尽管 `path_to_cstring` 的行为不尽如人意，但实际上已经是我们在给定 libgit2 接口的情况下所能做的最佳选择了。

刚刚展示的两个 `path_to_cstring` 定义依赖于对 Error 类型的转换：`?` 运算符会尝试进行此类转换，Windows 版本会显式地调用 `.into()`。这些转换并不明显：

```
impl From<String> for Error {
    fn from(message: String) -> Error {
        Error { code: -1, message, class: 0 }
    }
}

// NulError是当`CString::new`在字符串中嵌入了空字符时返回的内容
impl From<std::ffi::NulError> for Error {
    fn from(e: std::ffi::NulError) -> Error {
        Error { code: -1, message: e.to_string(), class: 0 }
    }
}
```

接下来，先弄清楚如何解析对象标识符代表的 Git 引用。由于对象标识符只是一个 20 字节的哈希值，因此在安全 API 中公开它是完全没问题的：

```
/// 存储在Git对象数据库中的某种对象的标识符：提交、
/// 树、blob、标签等。这是对象内容的宽哈希码
pub struct Oid {
```

```
    pub raw: raw::git_oid
}
```

向 Repository 添加一个方法来执行查找：

```
use std::mem;
use std::os::raw::c_char;

impl Repository {
    pub fn reference_name_to_id(&self, name: &str) -> Result<Oid>
    {
        let name = CString::new(name)?;
        unsafe {
            let oid = {
                let mut oid = mem::MaybeUninit::uninit();
                check(raw::git_reference_name_to_id(
                    oid.as_mut_ptr(), self.raw,
                    name.as_ptr() as *const c_char))?;
                oid.assume_init()
            };
            Ok(Oid { raw: oid })
        }
    }
}
```

尽管 oid 在查找失败时会保持未初始化状态，但这个函数会确保其调用者只要简单地遵循 Rust 的 Result 惯用法就永远不会看到未初始化的值：调用者要么得到一个包含已正确初始化的 Oid 值的 Ok，要么得到一个 Err。

接下来，模块需要一种从存储库中检索提交的方法。我们会定义一个 Commit 类型，如下所示：

```
use std::marker::PhantomData;

pub struct Commit<'repo> {
    // 这必须始终是指向可用`git_commit`结构体的指针
    raw: *mut raw::git_commit,
    _marker: PhantomData<&'repo Repository>
}
```

正如之前提到的，`git_commit` 对象的生命周期绝不能超出从中检索出它的 `git_repository` 对象。Rust 的生命周期让代码精确地捕捉到了这条规则。

上一章的 `RefWithFlag` 示例曾使用 `PhantomData` 字段告诉 Rust，即使一个类型显然未包含任何引用，也要将它视为包含具有给定生命周期的引用。`Commit` 类型也需要这么做。在这个例子中，`_marker` 字段的类型是 `PhantomData<&'repo Repository>`，以表明 Rust 应该将 `Commit<'repo>` 视为持有对某个 `Repository` 的生命周期 `'repo` 的引用。

查找某个提交的方法如下所示：

```
impl Repository {
    pub fn find_commit(&self, oid: &Oid) -> Result<Commit> {
        let mut commit = ptr::null_mut();
        unsafe {
            check(raw::git_commit_lookup(&mut commit, self.raw,
&oid.raw))?;
        }
        Ok(Commit { raw: commit, _marker: PhantomData })
    }
}
```

这里是怎样将 `Commit` 的生命周期与 `Repository` 的生命周期联系起来的呢？根据 5.3.7 节介绍过的规则，`find_commit` 的签名省略了所涉及引用的生命周期。如果把生命周期写出来，那么完整的签名是这样的：

```
fn find_commit<'repo, 'id>(&'repo self, oid: &'id Oid)
    -> Result<Commit<'repo>>
```

这正是我们想要的：Rust 将返回的 `Commit` 视为从 `self` (`Repository`) 借入的值。

当 `Commit` 被丢弃时，必须释放它的 `raw::git_commit`：

```
impl<'repo> Drop for Commit<'repo> {
    fn drop(&mut self) {
        unsafe {
            raw::git_commit_free(self.raw);
        }
    }
}
```

可以从 Commit 中借入 Signature（姓名和电子邮件地址）和提交消息的文本：

```
impl<'repo> Commit<'repo> {
    pub fn author(&self) -> Signature {
        unsafe {
            Signature {
                raw: raw::git_commit_author(self.raw),
                _marker: PhantomData
            }
        }
    }

    pub fn message(&self) -> Option<&str> {
        unsafe {
            let message = raw::git_commit_message(self.raw);
            char_ptr_to_str(self, message)
        }
    }
}
```

下面是 Signature 类型：

```
pub struct Signature<'text> {
    raw: *const raw::git_signature,
    _marker: PhantomData<&'text str>
}
```

git_signature 对象只会从别处借来自己的文本，特别是，git_commit_author 返回的签名从 git_commit 借用了它们的文本。所以我们的安全 Signature 类型包括 PhantomData<&'text str>，以告诉 Rust 此类型要表现得就像含有一个生命周期为 'text 的 &str。和以前一样，无须编写任何

代码，`Commit::author` 就能正确地将它返回的 `Signature` 的 `'text` 生命周期关联到 `Commit` 的 `'text` 生命周期。
`Commit::message` 方法会对持有提交消息的 `Option<&str>` 执行相同的操作。

`Signature` 包括检索作者姓名和电子邮件地址的方法：

```
impl<'text> Signature<'text> {
    /// 将作者姓名以`&str`的形式返回。如果作者姓名不是格式良好的UTF-8，则返回`None`
    pub fn name(&self) -> Option<&str> {
        unsafe {
            char_ptr_to_str(self, (*self.raw).name)
        }
    }

    /// 将作者的电子邮件地址以`&str`的形式返回。如果作者的
    /// 电子邮件地址不是格式良好的UTF-8，则返回`None`
    pub fn email(&self) -> Option<&str> {
        unsafe {
            char_ptr_to_str(self, (*self.raw).email)
        }
    }
}
```

前面的方法都依赖于私有实用函数 `char_ptr_to_str`：

```
/// 尝试从`ptr`中借用`&str`，这里假设`ptr`可能为空或引用了格式错误的UTF-8。
/// 赋予结果一个生命周期，就好像它是从`_owner`那里借来的一样
///
/// 安全性：如果`ptr`不为空，则它必须指向一个以空字符结尾的
/// C字符串，该字符串至少在`_owner`的生命周期内可以安全访问
unsafe fn char_ptr_to_str<T>(_owner: &T, ptr: *const c_char) -> Option<&str> {
    if ptr.is_null() {
        return None;
    } else {
        CStr::from_ptr(ptr).to_str().ok()
    }
}
```

永远不会用到 `_owner` 参数的值，但会用到它的生命周期。这个函数签名中的生命周期把这一点显式地表达了出来：

```
fn char_ptr_to_str<'o, T: 'o>(_owner: &'o T, ptr: *const c_char)
    -> Option<&'o str>
```

`CStr::from_ptr` 函数会返回一个 `&CStr`，其生命周期是完全无界的，因为它是从一个解引用后的裸指针借来的。无界的生命周期几乎不可能准确，因此要尽快收窄它们。包含 `_owner` 参数会导致 Rust 将其生命周期同样应用于返回值的类型，这样调用者就可以接收到更准确的有界引用了。

尽管 `libgit2` 的文档非常好，但无法从 `libgit2` 文档中看出 `git_signature` 的 `email` 指针和 `author` 指针是否可以为空。我们（本书作者）在源代码中“挖掘”了一段时间，但始终无法以这种或那种方式说服自己，最终还是决定，为了以防万一，最好让 `char_ptr_to_str` 能够接受空指针。而如果是 Rust 代码，这种问题立即就可以通过类型回答出来：如果它是 `&str`，那你可以期望一定有字符串；如果它是 `Option<&str>`，那就是可选的。

最后，我们为所需的全部功能都提供了安全接口。`src/main.rs` 中的新 `main` 函数精简了很多，看起来像真正的 Rust 代码了：

```
fn main() {
    let path = std::env::args_os().skip(1).next()
        .expect("usage: git-toy PATH");

    let repo = git::Repository::open(&path)
        .expect("opening repository");

    let commit_oid = repo.reference_name_to_id("HEAD")
        .expect("looking up 'HEAD' reference");

    let commit = repo.find_commit(&commit_oid)
        .expect("looking up commit");

    let author = commit.author();
    println!("{}", <{}>\n",
        author.name().unwrap_or("(none)"),
        author.email().unwrap_or("(none)));
```

```
println!("{}", commit.message().unwrap_or("(none)"));  
}
```

在本章中，我们已经把一个未提供多少安全保证的简单接口，进化成了能将任何违反接口契约的行为转变为某种 Rust 类型错误的安全接口，把本质上不安全的 API 包装成了安全的 API。作为回报，现在 Rust 可以确保你使用的接口的正确性了。在大多数情况下，我们让 Rust 执行的规则就是 C 和 C++ 程序员最终不得不强加于自身的那些规则。你之所以会觉得 Rust 比 C 和 C++ 严格得多，不是因为这些规则太陌生，而是因为对这些规则的执行不仅是机械化的，更是综合而全面的。

23.6 结论

Rust 不是一门简单的语言。它的目标是横跨两个截然不同的世界。它是一门现代编程语言，天生安全，具有闭包和迭代器等便捷特性。但 Rust 旨在让你以最低的运行期开销控制运行计算机的原始能力。

该语言的轮廓就是由这些目标勾勒出来的。Rust 设法用安全代码弥合了大部分差距。它的借用检查器和零成本抽象让你尽可能接近裸机，却不必冒着未定义行为的风险。如果这还不够，或者你想利用现有的 C 代码，不安全代码和外部函数接口会随时待命。但同样，该语言不会止步于只提供这些不安全特性并“尊重”你的命运。它的目标始终是使用不安全特性来构建安全 API。这就是我们刚刚在 `libgit2` 中所做的，也是 Rust 团队在 `Box`、`Vec`、其他集合、通道等方面所做的。标准库充满了安全的抽象，但在幕后实现中使用了一些不安全代码。

像 Rust 这样一门雄心勃勃的语言，也许注定不会成为最简单的工具。Rust 安全、快速、并发且高效。请用它来构建大型、快速、安全、强大的系统，以充分利用它们所运行的硬件的全部能力。请用它让我们的软件变得更好！

作者介绍

吉姆·布兰迪 (Jim Blandy) 从 1981 年开始编程，1990 年开始编写自由软件。他一直是 GNU Emacs 和 GNU Guile 以及 GNU 调试器 GDB 的维护者。吉姆是 Subversion 版本控制系统的最初设计者之一。他目前负责为 Mozilla 处理 Firefox 的图形和渲染引擎。

贾森·奥伦多夫 (Jason Orendorff) GitHub 工程师，专注开发尚未公开的 Rust 项目，曾在 Mozilla 参与 JavaScript 引擎 SpiderMonkey 的开发。他对语法学、烘焙、时间旅行，以及帮助人们理解复杂的主题很感兴趣。

莉奥诺拉·F. S. 廷德尔 (Leonora F. S. Tindall) 软件工程师、类型系统爱好者。她喜欢使用 Rust、Elixir 和其他先进语言在一些关键领域构建健壮且适应性强的系统软件，特别是在医疗保健和数据所有权管理等领域。她从事各种开源项目的开发，从用陌生语言开发程序的遗传算法到 Rust 核心库和 crate 生态系统，并且为富于支持性和多样化的社区项目做出过贡献。在空闲时间，莉奥诺拉会制造用于音频合成的电子设备，同时她是一名狂热的无线电爱好者。她对硬件的热爱也延伸到了软件工程实践中。她用 Rust 和 Python 为 LoRa 收音机构建了应用程序软件，并使用软件和 DIY 硬件在 Eurorack 合成器上创作实验性电子音乐。

封面介绍

本书封面上的动物是蒙塔古蟹（Montagu's crab, *Xantho hydrophilus*），其分布于大西洋东北部和地中海地区。退潮时，你可以在礁石和巨石下发现这种螃蟹。当你掀起石头时，它会攻击性地举起钳子并张开，以使自己显得更强大。

这种看起来很健壮的螃蟹有着肌肉发达的外表和约 70 毫米宽的甲壳。甲壳边缘有褶皱，呈淡黄色或红褐色。它有 10 条腿：前面的一对螯足大小相同，有黑色的尖爪或钳子，后面是 3 对粗壮且相对较短的步足，最后一双腿是用来划水的。它们会侧着身子走路和划水。

这种螃蟹是杂食动物，主要以藻类、蜗牛和其他种类的螃蟹为食。它们大多在夜间活动。雌性会在每年的 3 ~ 7 月产卵，其幼虫在整个夏季的大部分时间里浮游在海上。

O'Reilly 图书封面上的很多动物是濒危生物，它们对这个世界很重要。

本书封面插图由 Karen Montgomery 绘制，基于 *Wood's Natural History* 中的图片。

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：
ebook@turingbook.com。

在这里可以找到我们：

- 微博 @图灵教育 ：好书、活动每日播报
 - 微博 @图灵社区 ：电子书和好文章的消息
 - 微博 @图灵新知 ：图灵教育的科普小组
 - 微信 图灵访谈 ：[ituring_interview](#)，讲述码农精彩人生
 - 微信 图灵教育 ：[turingbooks](#)
-