

Contents

前言	xv
译者序	xix
1 系统程序员的福音	1
1.1 Rust 替你承担责任	2
1.2 安全的并发编程	3
1.3 Rust 的速度很快	3
1.4 Rust 使协作变得更简单	4
2 Rust 概览	5
2.1 rustup 和 Cargo	6
2.2 Rust 函数	8
2.3 编写并运行单元测试	10
2.4 处理命令行参数	10
2.5 提供 web 页面	14
2.6 并发	20
2.6.1 曼德勃罗集到底是什么	21
2.6.2 解析成对的命令行参数	25
2.6.3 将像素映射到复数	27
2.6.4 绘制曼德勃罗集	29
2.6.5 写入图片文件	30
2.6.6 一个并发的曼德勃罗集程序	32
2.6.7 运行曼德勃罗集绘制器	37
2.6.8 安全性是不可见的	38
2.7 文件系统和命令行工具	39
2.7.1 命令行接口	40
2.7.2 读写文件	42

2.7.3	查找和替换	43
3	基本类型	47
3.1	固定位数的数字类型	50
3.1.1	整数类型	51
3.1.2	Checked、Wrapping、Saturating、Overflowing 算术	54
3.1.3	浮点数	56
3.2	布尔类型	58
3.3	字符	59
3.4	元组	60
3.5	指针类型	62
3.5.1	引用	62
3.5.2	Box	63
3.5.3	原始指针	63
3.6	数组、vector 和切片	64
3.6.1	数组	64
3.6.2	vector	65
3.6.3	切片	68
3.7	字符串类型	70
3.7.1	字符串字面量	70
3.7.2	字节字符串	71
3.7.3	内存中的字符串	71
3.7.4	String	73
3.7.5	使用字符串	74
3.7.6	其他类似字符串的类型	74
3.8	类型别名	75
3.9	基本类型之外	75
4	所有权与 move	77
4.1	所有权	78
4.2	move	83
4.2.1	更多 move 的操作	88
4.2.2	move 和控制流	89
4.2.3	move 和索引	90
4.3	Copy 类型：move 的例外	92
4.4	Rc 和 Arc：共享所有权	95

5 引用	99
5.1 值的引用	99
5.2 使用引用	103
5.2.1 Rust 的引用 vs C++ 的引用	103
5.2.2 对引用赋值	104
5.2.3 引用的引用	105
5.2.4 比较引用	105
5.2.5 引用永不为空	106
5.2.6 借用任意表达式的引用	106
5.2.7 切片和 trait 对象的引用	107
5.3 引用安全	107
5.3.1 借用一个局部变量	108
5.3.2 引用作为函数参数	111
5.3.3 向函数传递引用	113
5.3.4 返回引用	114
5.3.5 包含引用的结构体	115
5.3.6 不同的生命周期参数	117
5.3.7 省略生命周期参数	119
5.4 共享 vs 可变	120
5.5 拿起武器对抗对象之海	129
6 表达式	131
6.1 表达式语言	131
6.2 优先级与结合性	132
6.3 块和分号	135
6.4 声明	136
6.5 if 与 match	138
6.6 if let	140
6.7 循环	141
6.8 循环中的控制流	142
6.9 return 表达式	144
6.10 Rust 为什么会有 loop 循环	145
6.11 函数和方法调用	146
6.12 字段和元素	147
6.13 引用运算符	149
6.14 算术、位运算、比较、逻辑运算符	149
6.15 赋值	150
6.16 类型转换	151

6.17 闭包	152
6.18 继续	152
7 错误处理	153
7.1 panic	153
7.1.1 栈展开	154
7.1.2 中止	155
7.2 Result	155
7.2.1 捕捉错误	156
7.2.2 Result 类型别名	158
7.2.3 打印错误	158
7.2.4 传播错误	160
7.2.5 处理多种错误类型	161
7.2.6 处理“不可能发生”的错误	163
7.2.7 忽略错误	164
7.2.8 在 main() 中处理错误	165
7.2.9 声明自定义错误类型	166
7.2.10 为什么选择 Result	167
8 crate 与模块	169
8.1 Crate	169
8.1.1 版本	173
8.1.2 构建配置	173
8.2 模块	174
8.2.1 嵌套模块	175
8.2.2 单独文件中的模块	177
8.2.3 路径和导入	179
8.2.4 标准 prelude	182
8.2.5 pub use 声明	182
8.2.6 pub 结构体字段	183
8.2.7 静态量和常量	183
8.3 将程序变为库	184
8.4 src/bin 目录	186
8.5 属性	187
8.6 测试和文档	190
8.6.1 集成测试	192
8.6.2 文档	193
8.6.3 文档测试	196

8.7 指定依赖	199
8.7.1 版本	199
8.7.2 Cargo.lock	200
8.8 把 crate 发布到 crates.io	201
8.9 工作空间	203
8.10 更多有趣的事	204
9 结构体	207
9.1 命名字段结构体	207
9.2 类元组结构体	210
9.3 类单元结构体	211
9.4 结构体布局	212
9.5 使用 <code>impl</code> 定义方法	212
9.5.1 以 Box、Rc、Arc 传递 Self	215
9.5.2 类型关联函数	217
9.6 关联常量	218
9.7 泛型结构体	219
9.8 有生命周期参数的结构体	221
9.9 为结构体类型派生常见的 trait	222
9.10 内部可变性	223
10 枚举与模式	229
10.1 枚举	230
10.1.1 带有数据的枚举	233
10.1.2 内存中的枚举	234
10.1.3 使用枚举实现富数据结构	234
10.1.4 泛型枚举	237
10.2 模式	239
10.2.1 模式中的字面量、变量和通配符	241
10.2.2 元组和结构体模式	243
10.2.3 数字和切片模式	245
10.2.4 引用模式	245
10.2.5 匹配守卫	247
10.2.6 匹配多种可能	248
10.2.7 绑定和 @ 模式	249
10.2.8 模式可以用在哪里	250
10.2.9 填充二叉树	251
10.3 宏观视图	253

11 trait 与泛型	255
11.1 使用 trait	257
11.1.1 trait 对象	258
11.1.2 泛型函数和类型参数	260
11.1.3 选择哪一种	263
11.2 定义和实现 trait	265
11.2.1 默认方法	266
11.2.2 trait 和其他人的类型	268
11.2.3 trait 中的 Self	269
11.2.4 子 trait	271
11.2.5 类型关联函数	272
11.3 完全限定方法调用	273
11.4 定义类型关系的 trait	275
11.4.1 关联类型 (或迭代器是如何工作的)	275
11.4.2 泛型 trait (或运算符重载是如何工作的)	278
11.4.3 impl Trait	279
11.4.4 关联常量	281
11.5 逆向工程约束	283
11.6 trait 作为基础	286
 12 运算符重载	 287
12.1 算术和位运算符	287
12.1.1 一元运算符	291
12.1.2 二元运算符	292
12.1.3 复合赋值运算符	293
12.2 相等性比较	294
12.3 顺序性比较	297
12.4 Index 与 IndexMut	300
12.5 其他运算符	302
 13 实用 trait	 305
13.1 Drop	306
13.2 Sized	309
13.3 Clone	312
13.4 Copy	313
13.5 Deref 与 DerefMut	313
13.6 Default	317
13.7 AsRef 与 AsMut	318

13.8 Borrow 与 BorrowMut	320
13.9 From 与 Into	322
13.10 TryFrom 与 TryInto	324
13.11 ToOwned	326
13.12 Borrow 和 ToOwned 的配合: Cow	326
14 闭包	329
14.1 捕获变量	330
14.1.1 借用值的闭包	331
14.1.2 偷取值的闭包	332
14.2 函数和闭包类型	333
14.3 闭包的性能	336
14.4 闭包和安全性	337
14.4.1 杀死值的闭包	337
14.4.2 FnOnce	338
14.4.3 FnMut	340
14.4.4 闭包的 Copy 和 Clone	342
14.5 回调	343
14.6 高效地使用闭包	348
15 迭代器	351
15.1 Iterator 与 IntoIterator trait	352
15.2 创建迭代器	354
15.2.1 iter 和 iter_mut 方法	354
15.2.2 IntoIterator 实现	355
15.2.3 from_fn 和 successors	357
15.2.4 drain 方法	358
15.2.5 其他迭代器源	359
15.3 迭代器适配器	360
15.3.1 map 和 filter	361
15.3.2 filter_map 和 flat_map	363
15.3.3 flatten	365
15.3.4 take 和 take_while	367
15.3.5 skip 和 skip_while	368
15.3.6 peekable	368
15.3.7 fuse	370
15.3.8 可逆迭代器和 rev	371
15.3.9 inspect	372

15.3.10 <code>chain</code>	372
15.3.11 <code>enumerate</code>	373
15.3.12 <code>zip</code>	374
15.3.13 <code>by_ref</code>	374
15.3.14 <code>cloned</code> 和 <code>copied</code>	376
15.3.15 <code>cycle</code>	376
15.4 消耗迭代器	377
15.4.1 简单的累计: <code>count</code> , <code>sum</code> , <code>product</code>	377
15.4.2 <code>max</code> , <code>min</code>	378
15.4.3 <code>max_by</code> , <code>min_by</code>	378
15.4.4 <code>max_by_key</code> , <code>min_by_key</code>	379
15.4.5 比较 item 序列	380
15.4.6 <code>any</code> 和 <code>all</code>	380
15.4.7 <code>position</code> , <code>rposition</code> , <code>ExactSizeIterator</code>	381
15.4.8 <code>fold</code> 和 <code>rfold</code>	381
15.4.9 <code>try_fold</code> 和 <code>try_rfold</code>	382
15.4.10 <code>nth</code> , <code>nth_back</code>	384
15.4.11 <code>last</code>	384
15.4.12 <code>find</code> , <code>rfind</code> , <code>find_map</code>	384
15.4.13 构建集合: <code>collect</code> 和 <code>FromIterator</code>	385
15.4.14 <code>Extend</code> trait	387
15.4.15 <code>partition</code>	388
15.4.16 <code>for_each</code> 和 <code>try_for_each</code>	389
15.5 实现自己的迭代器	390
16 集合	395
16.1 概述	396
16.2 <code>Vec<T></code>	397
16.2.1 访问元素	398
16.2.2 迭代	400
16.2.3 增长和缩减 vector	400
16.2.4 连接	404
16.2.5 切分	404
16.2.6 交换	407
16.2.7 排序和搜索	408
16.2.8 比较切片	410
16.2.9 随机元素	410
16.2.10 Rust 排除了无效性错误	411

16.3 VecDeque<T>	412
16.4 BinaryHeap<T>	414
16.5 HashMap<K, V> 和 BTreeMap<K, V>	415
16.5.1 条目	419
16.5.2 迭代 map	421
16.6 HashSet<T> 和 BTreeSet<T>	422
16.6.1 迭代 set	424
16.6.2 当相等的值不同时	424
16.6.3 集合操作	425
16.7 哈希	426
16.8 使用一个自定义的哈希算法	427
16.9 标准集合之外	429
17 字符串与文本	431
17.1 Unicode 背景知识	431
17.1.1 ASCII, Latin-1, Unicode	432
17.1.2 UTF-8	432
17.2 字符(char)	432
17.3 String 和 str	432
17.3.1 创建 String 值	432
17.3.2 简单的视图	432
17.3.3 附加和插入文本	432
17.4 格式化	432
17.5 正则表达式	432
17.5.1 基本正则使用	432
17.5.2 惰性构建正则值	432
18 输入输出	433
18.1 Reader 和 Writer	433
18.1.1 Reader	435
18.1.2 有缓冲的 Reader	437
18.1.3 读取行	438
18.1.4 收集行	441
18.1.5 Writer	442
18.1.6 File	443
18.1.7 Seek	444
18.1.8 其他 Reader 和 Writer 类型	444
18.1.9 二进制数据, 压缩和序列化	447

18.2 文件和目录	448
18.2.1 OsStr 和 Path	448
18.2.2 Path 和 PathBuf 方法	450
18.2.3 文件系统访问函数	452
18.2.4 读取目录	454
18.2.5 平台特定特性	456
18.3 网络	457
 19 并发	 461
19.1 fork-join 并行	462
19.1.1 spawn 和 join	464
19.1.2 跨线程的错误处理	466
19.1.3 跨线程共享不可变数据	467
19.1.4 Rayon	469
19.1.5 回顾曼德勃罗集	471
19.2 通道	473
19.2.1 发送值	474
19.2.2 接收值	479
19.2.3 运行流水线	480
19.2.4 通道的特性和性能	482
19.2.5 线程安全: Send 和 Sync	483
19.2.6 使用管道串联迭代器	485
19.2.7 超越流水线	487
19.3 共享可变状态	487
19.3.1 互斥锁是什么?	488
19.3.2 Mutex<T>	489
19.3.3 mut 和 Mutex	491
19.3.4 为什么有时互斥锁不是好方案	492
19.3.5 死锁	493
19.3.6 中毒的互斥锁	493
19.3.7 使用互斥锁的多消费者通道	494
19.3.8 读写锁 (RwLock<T>)	495
19.3.9 条件变量 (Condvar)	496
19.3.10 原子量	497
19.3.11 全局变量	499
19.4 Rust 中的 hacking 并发代码是什么样的	501

20 异步编程	503
20.1 从同步到异步	505
20.1.1 Future	506
20.1.2 async 函数和 await 表达式	508
20.1.3 在同步代码中调用异步函数: block_on	510
20.1.4 spawn 异步任务	513
20.1.5 async 块	517
20.1.6 从异步块中构建异步函数	519
20.1.7 在一个线程池中 spawn 异步任务	520
20.1.8 但你的 future 实现了 Send 吗?	521
20.1.9 长时间计算: yield_now 和 spawn_blocking	524
20.1.10 比较异步设计	525
20.1.11 一个真实的异步 HTTP 客户端	526
20.2 一个异步的客户端和服务器	527
20.2.1 Error 和 Result 类型	529
20.2.2 协议	529
20.2.3 获取用户输入: 异步流	531
20.2.4 发送包	533
20.2.5 接收包: 更多异步流	534
20.2.6 客户端的 main 函数	536
20.2.7 服务器的 main 函数	537
20.2.8 处理聊天连接: 异步的 Mutex	538
20.2.9 聊天组表: 同步的 Mutex	541
20.2.10 聊天组: tokio 的广播通道	542
20.3 原语 future 和 executor: 何时一个 future 值得再次 poll	545
20.3.1 调用 waker: spawn_blocking	547
20.3.2 实现 block_on	549
20.4 Pin	551
20.4.1 future 的两个生命阶段	551
20.4.2 Pinned 指针	555
20.4.3 Unpin trait	556
20.5 异步代码什么时候能带来帮助?	557
21 宏	561
21.1 宏基础	562
21.1.1 宏展开基础	563
21.1.2 意外的结果	565
21.1.3 重复	566

21.2 内建的宏	569
21.3 调试宏	571
21.4 构建 json! 宏	572
21.4.1 片段类型	573
21.4.2 宏中的递归	576
21.4.3 宏和 trait	577
21.4.4 作用域和 hygiene	579
21.4.5 导入和导出宏	582
21.5 在匹配时避免语法错误	584
21.6 macro_rules! 之外	585
 22 unsafe 代码	 587
22.1 unsafe 从何而来?	588
22.2 unsafe 块	589
22.3 示例: 一个高效的 ASCII 字符类型	590
22.4 unsafe 函数	592
22.5 unsafe 块还是 unsafe 函数?	594
22.6 未定义行为	595
22.7 unsafe trait	597
22.8 原始指针	599
22.8.1 安全地解引用原始指针	601
22.8.2 示例: RefWithFlag	602
22.8.3 可空的指针	605
22.8.4 类型大小和对齐	605
22.8.5 指针算术	606
22.8.6 移进和移出内存	607
22.8.7 示例: GapBuffer	612
22.8.8 unsafe 代码中的 panic 安全性	619
22.9 使用 union 重新解释内存	620
22.9.1 match union	622
22.10 借用 union 的引用	623
 23 外部函数	 625
23.1 寻找公共的数据表示	625
23.2 声明外部函数和变量	630
23.3 使用库里的函数	631
23.4 一个 libgit2 的原始接口	635
23.5 一个 libgit2 的安全接口	641

23.6 总结	653
-------------------	-----

前言

Rust 是一门系统编程语言。

因为很多业务程序员对系统编程并不是很熟悉，所以这里首先简单解释一下什么是系统编程，为之后的内容奠定基础。

当你合上笔记本电脑时，操作系统检测到了这一行为，然后把所有正在运行的程序挂起、关掉屏幕、并把电脑设置为睡眠；之后，当你打开笔记本电脑时，屏幕和其他组件被再次唤醒，并且每个程序可以在它中断的地方继续运行。我们对此习以为常，但这都多亏了系统程序员为此编写的很多代码。

系统编程被用于以下领域：

- 操作系统
- 各种设备的驱动
- 文件系统
- 数据库
- 在非常廉价或需要极高的可靠性的设备上运行的代码
- 密码学
- 多媒体编解码器（用于读写音频、视频、图片文件的软件）
- 多媒体处理（例如，语音识别或图像处理软件）
- 内存管理（例如，实现一个垃圾回收器）
- 文本渲染（把文本和字体转换为像素点的过程）
- 实现更高级的编程语言（例如 JavaScript 和 Python）
- 网络
- 虚拟化和容器
- 科学仿真
- 游戏

简而言之，系统编程是一种资源受限 (*resource-constrained*) 的编程方式、是一种每个字节和每个 CPU 时钟都需要考虑的编程方式。仅仅是为了支持一个基本的应用所需要的系统代码的数量也是非常惊人的。

本书并不会教你系统编程。事实上，这本书包含了很多有关内存管理的细节，如果你没有自己进行过系统编程，你会感觉这些内容乍一看似乎没有必要。但如果你是一个熟练的系统程序员，你将会发现 Rust 是一门非常优秀的语言：它可以解决困扰了整个工业界几十年的主要问题。

谁应该阅读这本书

如果你已经是一名系统程序员并且已经准备好替换 C++，那么这本书就是为你而生。如果你是一名有其他任何语言的经验的开发者，不管是 C#、Java、Python、JavaScript 还是其他语言，这本书同样适用于你。

然而，这阅读这本书的过程中，你需要学习的不止是 Rust。为了充分利用这门语言，你还需要一些系统编程的经验。我们推荐在阅读这本书的同时用 Rust 实现一些系统编程的项目，构建一些你以前从来没有构建过的东西、一些充分利用 Rust 的速度、并发和安全性的方法。本前言开头的列表也许能给你一些启发。

我们为什么要撰写这本书

早在我们开始学习 Rust 时我们就开始着手编写这本书。我们的目标是首先讲解 Rust 中主要的、新的概念，清晰而深入地呈现它们，以最大限度地避免通过试错来学习。

本书概览

本书的前两章简单地介绍了 Rust，并提供了一些简单的示例。之后我们转到第3章的基本数据类型。第4章、第5章专注于介绍所有权和引用的核心概念。我们推荐按照顺序阅读前5章。

第6到第10章覆盖了语言的基础部分：表达式（第6章），错误处理（第7章），crate 和 模块（第8章），结构体（第9章），枚举和模式（第10章）。在这一部分可以跳过一些内容，但请不要跳过错误处理的章节。第11章包括 trait 和 泛型，这是最后两个必需的重要概念。trait 类似于 Java 或 C# 中的接口。它们也是 Rust 支持把你的类型集成到语言中的主要手段。第12章展示了 trait 怎么支持运算符重载，第13章包括了很多有用的工具 trait。

理解了 trait 和 泛型就可以解锁本书的剩余部分了。闭包和迭代器，这两个你绝对不想错过的强大工具，分别在第14章和第15章中介绍。你可以以任意顺序阅读剩下的章节，或者按需阅读。它们包括了剩余的语言部分：集合（第16章），字符串和文本（第17章），输入和输出（第18章），并发（第19章），异步代码（第20章），宏（第21章），unsafe 代码（第22章）和调用其它语言中的函数（第23章）。

本书中的约定

本书中用到了以下约定：¹

斜体

表示新术语、URL、电子邮件地址、文件名和文件拓展名。

等宽

用于代码环境和在段落中引用代码中的元素例如变量或函数名、数据库、数据类型、环境变量、语句和关键字等。

等宽加粗

表示需要用户逐字输入的命令或其他文本。

¹译者注：因为我比较懒，所以基本上不会用等宽加粗和等宽斜体这两种字体。

等宽斜体

表示需要用户用自己的值或者上下文推断出的值进行替换的文本。

NOTE

这个标志表示一个注意事项。

使用示例代码

补充材料（示例代码，练习等）可以在<https://github.com/ProgrammingRust>下载。

本书旨在帮助你完成工作。一般来说，本书中提供的示例代码都可以直接在自己的编程和文章中使用。除非你需要再次分发大量本书中的代码，否则你不需要联系我们获取授权。例如，编写一个使用了书中部分代码的程序并不需要授权。售卖或者分发 O'Reilly 书籍中的示例代码则需要授权。通过引用本书或书中的示例代码回答问题并不需要授权。将本书中的大量代码纳入你自己的产品文档则需要授权。

我们感激，但并不要求署名。如果要署名的话，应该包含标题、作者、出版社和 ISBN。例如：“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.”

如果你感觉你对示例代码的使用方式不在上述范围内，请放心联系我们permissions@oreilly.com。

O'Reilly 在线学习

NOTE

40 多年以来，O'Reilly Media 提供技术和业务培训、知识和洞察力，以帮助公司取得成功。

我们独特的专家和创意网络会通过书籍、文章、会议和在线学习平台分享他们的知识。O'Reilly 的在线学习平台可以让你按需访问 O'Reilly 及其他 200 多家出版社的实时培训课程、深入学习路线、交互式代码环境。更多信息请访问<http://oreilly.com>。

如何联系我们

请将和本书有关的评论和问题发送给出版社：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

我们有一个本书的 web 页面，我们在那里列出了勘误表、示例和其他附加信息。你可以通过<https://oreil.ly/programming-rust-2e>访问该页面。

评论或有关本书的技术问题可以发送到邮箱bookquestions@oreilly.com。

访问<http://www.oreilly.com>获取更多有关我们的书籍和课程的信息。

我们的 Facebook: <http://facebook.com/oreilly>

我们的 Twitter: <http://twitter.com/oreillymedia>

我们的 YouTube: <http://youtube.com/oreillymedia>

致谢

你手中的这本书从我们的官方技术评审：Brian Anderson、Matt Brubeck、J. David Eisenberg、Ryan Levick、Jack Moffitt、Carol Nichols、Erik Nordin 和翻译：Hidemoto Nakada (中田秀基) (Japanese)、Mr. Songfeng Li (Simplified Chinese)、Adam Bochenek 和 Krzysztof Sawka (Polish) 处受益匪浅。

还有很多非官方的评审阅读了早期的草案并提供了很有价值的反馈。我们想要感谢 Eddy Bruel、Nick Fitzgerald、Graydon Hoare、Michael Kelly、Jeffrey Lim、Jakob Olesen、Gian-Carlo Pascutto、Larry Rabinowitz、Jaroslav Šnajdr、Joe Walker、Yoshua Wuyts 的认真评论。Jeff Walden 和 Nicolas Pierron 花费了宝贵的时间来审阅几乎整本书。就像编程一样，一本编程书籍需要高质量的 bug 报告才能不断成长。感谢你们。

Mozilla 对 Jim 和 Jason 在此项目中的工作非常宽容，尽管这超出了我们的官方职责范围，并会和他们产生竞争。我们非常感谢 Jim 和 Jason 的领导：Dave Camp、Naveed Ihsanullah、Tom Tromey、Joe Walker 的支持。他们从长远的角度看待 Mozilla，我们希望这些结果证明了他们对我们的信任。

我们还想对 O'Reilly 里每个帮助过我们的人表达感谢，尤其是非常有耐心的编辑 Jeff Bleiel 和 Brian MacDonald，以及我们的策划编辑 Zan McQuade。

最重要的是，我们衷心感谢家人们坚定不移的爱、热情和耐心。

译者序

这是我翻译的第二本英文资料。

相识

初次听说 Rust 的传闻还是大二时候听说有位清华的大佬用 Rust 实现了自己的编译器和操作系统作为毕设，当时虽然不懂但大为震撼（笑）。初次接触 Rust 应该还是大三，当时是看官方的书入门的。但我相信一定有很多人和我一样，看完官方的书之后合上书发现自己其实还是什么也写不出来（笑）。

初次接触本书是在 21 年 7 月初从 Rust 中文社区的推送消息中得知，当发现本书有一千多页时推测本书应该是类似 C/C++ Primer [Plus] 那样的比较全面的书籍，事实果然如此。不过因为当时忙于科研，没有空余时间阅读本书，就一直搁置了下去（实名吐槽某实验室让大四刚毕业的学生搞科研搞到 8 月份才能放暑假回家）。

初次开始阅读和翻译本书是在 2021.9.19 日，撰写本序是在 2022.9.18 日晚，历时恰好一年完成了本书的翻译，不得不感叹命运总是如此巧妙。（整整一年，鬼知道我这一年都经历了什么。）

相知

研一刚开学之后忙于科研，到 9 月下旬才有时间开始翻译，之后连续翻译了几个月，直到快到 21 年底的时候科研任务又紧张起来才中断了翻译。再之后就是寒假前后才又有时间继续翻译，在寒假结束之后开学不久我就做出了人生中也许是第二大也是第二正确的选择，随之而来的又是很长一段时间的翻译中断。终于在 22 年 9 月重新拾起了翻译，坚持到了最后。

在翻译本书的前期，我最大的感受只有两个字——绝望。一千多页的压力让我几乎窒息，即使我每天把所有的时间用来翻译，也只能翻译 20-30 页，而当时研一的我科研任务繁重，经常每天只能抽出一两个小时来进行翻译，结果就是每天只能翻译几页。也只有在周末，才能拿出全天时间来翻译，但也只能翻译 20-30 页。就这样持续了两周之后，突然发现自己坚持

了两周把每天所有空闲时间都拿来翻译，竟然只翻译了 $\frac{1}{5}$ 还不到？再加上科研压力越来越大，翻译的时间越来越短，绝望之感尤甚。

还记得心态的转变是在翻译到了 $\frac{1}{3}$ 左右的时候，就像是看到了曙光一样，第一次有了自己可以翻译完这本书的信心。也是从那时起我不再经常犹豫是否还要继续翻译下去。在那之前我也深刻地感受到了坚持做一件看不到希望的事情是多么痛苦。

22年5月于Rust中文社区推送的消息中得知已有人完成了本书的翻译，当时我离弃坑就差那么亿点点的距离：直到那时本译文仍然几乎无人问津，且我已经明白了我以后的工作是不可能用到Rust的，既然我已用不到Rust而且本书也有人翻译过了，那我为什么还要继续下去？继续学习Rust和翻译本书对我而言又有什么意义？幸运的是我很快就得到了这个问题的答案，再加上强迫症的影响，我还是决定继续下去。

尾声

本书于2022.9.18日完成翻译（除了第17章不打算进行翻译），在完成时我感受到的并非是完成了一项艰难的任务之后的成就感和幸福感，而是一种平静和释然的感觉。成就感和幸福感不能说没有，只能说是被一年的时长稀释了，留下的只有平淡如水的心境，似乎翻译逐渐成为了习惯、成为了一有空闲时间就会选择的消磨时间的方式。

我在翻译的过程中可以说是收获满满，（虽然长达一年的时长让我已经几乎完全忘记了自己收获），希望阅读本书的你也能有所收获。

——汪屹硕 (MeouSker77)

(真正的实名，笑)

2022.9.18

致谢

感谢所有在github上star、fork、issue、pr的读者，你们对本译文的关注和支持也是我坚持下去的主要动力之一。

Chapter 1

系统程序员的福音

在某些场景下——例如 *Rust* 的目标场景——比竞争对手快 $10x$ 或者仅仅 $2x$ 是足以决定成败的事情。它决定了一个系统在市场中的命运，就像在硬件市场中一样。

——Graydon Hoare

现在所有的计算机都是并行的……并行编程才是编程。

——Michael McCool et al., Structured Parallel Programming

民族国家的攻击者们利用 *TrueType* 解析器的漏洞来进行监视；所有的软件都对安全很敏感。

——Andy Wingo

我们选择用以上三个引言来开始本书是有原因的。但首先让我们以一个谜题开始。下面的 C 程序做了什么？

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

今天早上在 Jim 的笔记本电脑上，上面的程序输出了：

```
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，它的行为可能会不同。这个过程中到底发生了什么呢？

这段代码是有漏洞的。数组 `a` 的长度为 1，对 `a[3]` 的访问，根据 C 语言标准，是未定义行为：

对于使用不可移植或错误的程序结构或错误的数据的行为，本国际标准不做任何要求

未定义行为不仅仅会导致非预期的结果，语言标准甚至允许程序在这种情况下做任何事情。在我们的例子中，把一个特定的值存在特定数组的第 4 个元素处恰巧破坏了函数的调用栈，因此导致 `main` 函数返回时，并没有正常的退出程序，而是跳转到了 C 标准库里从用户的家目录中的一个文件中读取密码的代码中。这显然是有问题的。

C 和 C++ 有几百条避免未定义行为的规则。它们大多都是一些常识：不要访问不应该访问的内存，不要让算术运算溢出，不要除以零等等。然而编译器并不强制这些规则，它没有义务去检测哪怕明目张胆的违反规则的行为。事实上，上述程序编译时不会有错误和警告。避免未定义行为的责任全部落到你——程序员身上。

根据经验，我们程序员并不能很好的识别出未定义行为。有一位 Utah 大学的学生，研究员 Peng Li 修改了 C 和 C++ 的编译器来让它们在编译时报告它们正在编译的程序中是否含有会导致未定义行为的模式。他发现几乎所有的程序都有，包括那些公认的优秀项目。想要在 C 和 C++ 中避免未定义行为就和仅仅知道规则就想赢得国际象棋比赛一样不切实际。

各种偶然的奇怪信息或崩溃可能只属于质量问题，但自从 1988 年莫里斯蠕虫病毒使用前面显示的技术的一个变种在早期互联网上从一台计算机传播到另一台计算机以来，无意中的未定义行为就成了安全漏洞的一个主要原因。

因此 C 和 C++ 把程序员推到了一个很尴尬的地位：这些语言是系统编程的工业标准，但它们对程序员的要求和限制却只能保证不断出现崩溃和安全问题。回答我们的谜题只是抛出了一个更大的问题：我们不能做的更好吗？

1.1 Rust 替你承担责任

我们的答案对应着我们开头的三个引言。第三个引言引用自一篇报告，这篇报告中，一个叫做 Stuxnet 的计算机蠕虫在 2010 年被发现侵入了工业界的设备，并获取了受害计算机的控制权。它只是利用了解析 word 文档中嵌入的 TrueType 字体的代码中的未定义行为，没有使用任何其他技术。这段代码的作者显然没有预料到这段代码会被以这种形式利用，这说明不仅仅只有操作系统和服务器需要担心安全问题：任何需要处理来自不受信任来源的数据的软件都可能成为受害者。

Rust 语言做了一个简单的保证：如果你的代码通过了编译器的检查，那么它将不会遇到未定义行为。悬垂指针，两次释放，空指针解引用都会在编译期被捕捉到。对数组的引用通过编译期和运行期的双重检查保证安全，当索引越界时，Rust 不会像不幸的 C 语言一样出现缓

冲区溢出，而是会安全地退出程序并打印出错误消息。

Rust 旨在同时实现安全和易于使用。为了对你的程序行为做出更强的保证，Rust 对你的代码施加了比 C 和 C++ 更多的限制，这些限制需要通过一些实践和经验才能习惯。但总体来看这门语言的灵活性和表达力都是很强的。Rust 的应用范围之广已经证明了这一点。

根据我们的经验，在相信语言可以帮助我们捕获错误的情况下，我们将有勇气尝试更有挑战性的项目。修改复杂的大型程序的风险将会降低，因为我们不再需要关注内存管理和指针有效性的问题。调试起来也会简单得多，因为潜在的 bug 不会破坏不相关的代码部分。

当然，还有很多 Rust 也不能检测出的 bug。但在实践中，没有未定义行为可以显著改善开发的现状。

1.2 安全的并发编程

在 C 和 C++ 中并发是众所周知的难，开发者通常只有在已经证明了单线程代码无法达到所需性能的情况下才会考虑并发。但第二个引言则认为并行非常重要以至于现代计算机将它视为基本的操作。

事实证明，Rust 中保证内存安全的限制也可以保证 Rust 程序中不会出现数据竞争。你可以在线程间安全的共享数据，只要它不是正在被修改。被修改的数据只能通过同步原语来访问。你可以使用所有传统的工具：互斥锁、条件变量、通道、原子量等等，Rust 会通过检查确保你正确地使用它们。

这些使 Rust 能够充分利用现代多核机器的性能。Rust 的生态还提供了普通并发原语之外的库来帮助你完成复杂的负载，包括处理器池、无锁同步机制例如 Read-Copy-Update 等。

1.3 Rust 的速度很快

最后，对应我们的第一条引言。Rust 遵循了 Bjarne Stroustrup 在他的文章 “Abstraction and the C++ Machine Model” 中提到的为 C++ 设计的原则：

一般情况下，C++ 的实现遵循 *O* 开销原则：你没有用到的部分，将不会有开销。你用到的部分，你将不能找到更好的代码。

系统编程经常需要考虑如何将机器性能发挥到极限。对于视频游戏，整个机器都需要投入工作来为玩家创造出最好的体验。对于网页浏览，浏览器的性能制约了内容发布者可以做的事情的上限，在机器本身的限制范围内，浏览器需要将尽可能多的内存和处理器资源留给内容本身。同样的原则也适用于操作系统：内核需要把机器的资源尽可能多的留给用户程序，而不是被它们自身消耗。

但当我们说 Rust 很“快”的时候，到底是什么意思？一个人可以用任何通用语言写出非常慢的代码。更准确地说，如果你已经准备好认真设计你的程序来最大限度的利用底层机器的性能，那么 Rust 可以支撑你实现目标。这门语言的效率很高，并且能给予你控制使用多少内存和 CPU 资源的能力。

1.4 Rust 使协作变得更简单

我们在标题中隐藏了第 4 条引言：“系统程序员的福音”。这是在暗示 Rust 对代码共享和重用的支持。

Rust 的包管理器和构建工具 Cargo，使用户可以很容易地使用其他用户发布在 Rust 的公开仓库 crates.io 上的库。你只需要简单地在一个文件中加上库的名字和版本号，cargo 将会自动下载该库和它的依赖，并把它们链接在一起。你可以将 Rust 的 Cargo 视为 NPM 或者 RubyGems 一类的东西，只不过还同时强调完善的版本控制和可复制的构建。有很多流行的 Rust 库可以提供从序列化到 HTTP 客户端和服务器再到现代图形 API 等几乎任何功能。

进一步讲，这门语言本身就被设计为支持协作：Rust 的 trait 和泛型让你能创建出拥有灵活接口的库，它们可以在很多不同的上下文中工作。Rust 的标准库也提供了一组核心的基础类型，为常见的情况建立了共享的约定，使不同的库可以更容易地协同使用。

下一章旨在更具体地说明我们在这一章中提出的观点，我们通过几个小的 Rust 程序作为示例来展示这门语言的强大之处。

Chapter 2

Rust 概览

Rust 给像本书一样的书籍的作者提出了一个挑战：赋予这门语言特色的并不是可以在第一页就展示出来的某些具体的、惊人的特性，而是如何设计这门语言以使它的各个部分可以无缝的协同工作，最终达到我们在上一章提到的目标：安全、高性能的系统编程。这门语言的每一部分在以其他所有部分为背景的前提下都是最合理的。

因此，相比于一次着眼于一种语言特性，我们选择了几个简单但却完整的程序作为概览，每一个程序都会涉及到一些语言特性：

- 作为热身，我们准备了一个简单的计算命令行参数的程序，以及相应的单元测试。这个程序展示了 Rust 的核心类型，并引入了 *trait*。
- 接下来，我们构建了一个 web 服务器。我们将会使用一个第三方库来处理 HTTP 的细节，并引入字符串处理、闭包、错误处理。
- 我们的第三个程序绘制了一个漂亮的图形，将计算分布到多个线程来提高速度。这一部分包括一个泛型函数的示例，阐明了怎么处理一个像素以及类似的概念，并展示了 Rust 对并发的支持。
- 最后，我们展示了一个使用正则表达式处理文件的健壮的命令行程序。这个程序展示了 Rust 标准库中处理文件的设施和最常用的第三方正则表达式库。

Rust 保证在对代码的性能影响最小的情况下防止未定义行为，这一保证影响了整个 Rust 中每个部分的设计，从标准的数据结构例如 vector 和 string 到 Rust 程序员使用第三方库的方式都受此影响。这些具体的细节书中都会提到，但是现在，我们想向你展示 Rust 是一门强大且有趣的语言。

当然，首先你要在你的计算机上安装 Rust。

2.1 rustup 和 Cargo

安装 Rust 的最佳方式是使用 `rustup`。访问<https://rustup.rs>并按照说明进行操作。

或者，你可以访问[Rust 网站](#)来获取预构建好的 Linux、macOS、Windows 上的包。一些操作系统发行版里也包含 Rust。我们推荐使用 `rustup`，因为它是专门用于管理 Rust 安装的工具，就像 Ruby 的 RVM 和 Node 的 NVM 一样。例如，当一个新版本的 Rust 发布时，你只需要输入 `rustup update` 就可以完成更新。

无论如何，一旦你完成了安装，你就应该可以通过命令行访问以下三条新命令：

```
$ 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)
```

这里，`$` 是命令提示符。在 Windows 上，可能是 `C:\>` 或者别的类似的。这里我们运行了安装的三条命令，查询它们的版本。接下来让我们依次讲解每一个命令：

- `cargo` 是 Rust 的编译管理器、包管理器和通用的工具。你可以使用 Cargo 来新建项目、构建并运行程序、管理所有代码中依赖的外部库。
- `rustc` 是 Rust 的编译器。通常我们使用 Cargo 来调用编译器，但有时也需要直接运行它。
- `rustdoc` 是 Rust 的文档工具。如果你在源代码中按照文档注释的格式写了文档，那么 `rustdoc` 可以通过它们构建出漂亮的 HTML 文档。和 `rustc` 一样，我们通常用 Cargo 来调用 `rustdoc`。

使用 Cargo 可以很方便地创建新的 Rust 包，并设置好一些标准元数据：

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

这个命令创建了一个叫做 `hello` 的新的包目录，并准备好构建一个可执行程序。

进入包的顶级目录并查看：

```
$ 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
```

我们可以看到 Cargo 创建了一个文件 `Cargo.toml` 来保存包的元数据。此时这个文件里还没有太多内容：

```
[package]  
name = "hello"  
version = "0.1.0"  
authors = ["You <you@example.com>"]  
edition = "2018"  
  
# See more keys and their definitions at  
# https://doc.rust-lang.org/cargo/reference/manifest.html
```

[dependencies]

如果我们的程序中需要使用第三方库，那么可以把它们添加在这个文件里，Cargo 将会负责下载、构建和更新这些库。我们将在第8章中详细介绍 `Cargo.toml` 文件。

Cargo 已经为我们的包初始化好了 git 版本控制系统，创建了一个 `.git` 元数据目录和一个 `.gitignore` 文件。你可以通过向 `cargo new` 命令传递 `--vcs none` 参数来跳过这一步。

`src` 子目录包含了实际的 Rust 代码：

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

看起来 Cargo 好像已经替我们写好了程序。`main.rs` 中包含以下内容：

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

在 Rust 中，你甚至不需要自己编写 “Hello, World!” 程序。它是 Rust 的样板程序：包含两个文件，总共 13 行。

我们可以从包中的任何目录调用 `cargo run` 命令来构建并运行我们的程序：

```
$ 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 调用了 Rust 的编译器 `rustc`，然后运行了它生成的可执行文件。Cargo 把可执行文件放在了顶层目录的 `target` 子目录下：

```
$ 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!
```

如果需要的话，Cargo 可以为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 的语法借鉴自其他语言。如果你熟悉 C、C++、Java 或者 JavaScript，你可以很快找到自己的方式来理解 Rust 的程序结构。这里有一个使用[欧几里得算法](#)计算两个整数的最大公约数的函数。你可以把它添加到 `src/main.rs` 的最后：

```
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` 关键字（读作 “fun”）创建了一个函数。这里，我们定义了一个叫 `gcd` 的函数，它有两个参数 `m` 和 `n`，类型都是 `u64`，也就是 64 位无符号整数。`->` 词元指明了返回值类型：我们的函数返回一个 `u64` 类型的值。四个空格缩进是 Rust 的标准风格。

Rust 的整数类型的名字代表了它们的大小和符号性：`i32`是有符号 32 位整数；`u8`是无符号 8 位整数（用于表示“字节”）等。`isize` 和 `usize` 类型分别代表可以存下一个指针的有符号和无符号整数，在 32 位平台上它们就是 32 位，在 64 位平台上就是 64 位。Rust 还有两种浮点数类型：`f32` 和 `f64`，分别是 IEEE 标准的单精度和双精度浮点数类型，类似于 C 和 C++ 中的 `float` 和 `double`。

默认情况下，当变量初始化后，它的值就不能再被改变。但如果在参数 `m` 和 `n` 前加上 `mut` 关键字（读作“mute”，`mutable` 的缩写）就可以在函数体中对它们进行赋值。在实践中，大多数变量都不会被重新赋值，因此在阅读代码时 `mut` 关键字将是一个有用的提示。

函数体中首先调用了 `assert!` 宏来确保两个参数都不是 0。! 字符表示这是宏调用，而不是函数调用。类似于 C 和 C++ 中的 `assert` 宏，Rust 中的 `assert!` 宏也会检查参数是否为真，如果不为真则中断程序，并输出一条有用的信息，其中包括断言失败的源码位置。这种终止的方式被称为 *panic*。和 C 和 C++ 中断言可以被跳过不同，Rust 总是检查断言，不管程序怎么编译。还有一个 `debug_assert!` 宏，当程序被编译为 `release` 模式时会被跳过。

我们函数的主体是一个包含一条 `if` 语句和一条赋值语句的 `while` 循环。和 C 和 C++ 不同，Rust 的条件表达式不需要括号，但紧随其后的控制流语句需要花括号。

`let` 语句声明了一个局部变量，例如函数中的 `t`。我们不需要写出 `t` 的类型，因为 Rust 可以通过使用这个值的方式来推断它的类型。在我们的函数中，`t` 只有和 `m`、`n` 相匹配，即是 `u64` 类型时函数才可以正常运行。Rust 只在函数体内推断类型：你必须写出函数参数和返回值的类型，就像我们所做的那样。如果你想指明 `t` 的类型，你可以写：

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

Rust 有 `return` 语句，但是 `gcd` 函数并不需要。如果一个函数体以一个没有分号结尾的表达式结尾，那么这个表达式的值就是函数的返回值。事实上，任何一个花括号包围的语法块都可以作为一个表达式。例如，这里有一个表达式打印出一条消息，然后返回 `x.cos()` 作为它的值：

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

在 Rust 中当控制流到达函数底部时利用这种形式返回值是一种很典型的做法，只有当在函数的中途显式地返回时才会使用 `return` 语句。

2.3 编写并运行单元测试

Rust 语言内建有对测试的支持。为了测试我们的 gcd 函数，我们可以在 `src/main.rs` 的最后添加下列代码：

```
#[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叫 `test_gcd` 的函数，它调用了 `gcd` 函数并检查返回值是否正确。函数上方的 `#[test]` 标记 `test_gcd` 是一个测试函数，这种函数在正常编译时会被跳过，但在使用 `cargo test` 命令时会被编译并自动调用。我们可以在整个源码树的任何位置定义测试函数，`cargo test` 会自动收集它们并运行。

`#[test]` 标记是一个属性 (*attribute*)。属性是一种为函数和其他声明标记额外信息的开放式系统，类似于 C++ 和 C# 中的属性，或者 Java 中的注解。它们被用来控制编译器警告和代码风格检查、条件编译（类似于 C 和 C++ 中的 `#ifdef`）、告诉 Rust 怎么和其它语言编写的代码交互等。随着继续深入我们将会看到更多使用属性的例子。

把 `gcd` 和 `test_gcd` 函数的定义添加到 `hello` 包里之后，我们可以在包内的某个目录下按照如下方式运行测试：

```
$ cargo test
Compiling hello v0.1.0 (/home/jimb/rust/hello)
Finished test [unoptimized + debuginfo] target(s) in 0.35s
Running /home/jimb/rust/hello/target/debug/deps/hello-2375a82d9e9673d7

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

为了让我们的函数能获取一些作为命令行参数传入的数字并打印出他们的最大公约数，我们可以把 `src/main.rs` 中 `main` 函数的代码替换为如下内容：

```
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);
}
```

这是一个很大的代码块，让我们一步步来理解它：

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

第一个 `use` 声明引入了标准库中的 `FromStr trait`。一个 `trait` 就是一些可以被实现的方法的集合。任何实现了 `FromStr trait` 的类型都有一个 `from_str` 方法，这个方法尝试把一个字符串解析为该类型。`u64` 类型实现了 `FromStr`，因此我们将调用 `u64::from_str` 来解析命令行参数。尽管我们在程序中并没有使用到 `FromStr` 这个名字，但为了使用这个 `trait` 的方法，必须将它引入作用域。我们将会在第 11 章讲解 `trait`。

第二个 `use` 声明引入了 `std::env` 模块，它提供了一些和运行环境进行交互的函数和类型，包括 `args` 函数，它可以让我们获取到程序的命令行参数。

接下来移步到程序的 `main` 函数：

```
fn main() {
```

我们的 `main` 函数不返回值，因此我们可以省略 `->` 和返回类型。

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

我们声明了一个可变的局部变量 `numbers`, 并将它初始化为一个空的 `vector`。`Vec` 是 Rust 的可变长的向量类型, 类似于 C++ 的 `std::vector`、Python 的 `list`、或者 JavaScript 的 `array`。即使 `vector` 被设计为动态伸缩, 我们仍然需要将变量标记为 `mut`, 这样 Rust 才允许我们向它的末尾添加元素。

`numbers` 的类型是 `Vec<u64>`, 一个 `u64` 的 `vector`。但和之前一样, 我们不需要写出类型。Rust 将会为我们推断出它的类型: 这里我们把 `u64` 类型的值添加到了 `vector` 里, 而且我们之后还把这个 `vector` 的元素传给了 `gcd` 函数, 而 `gcd` 函数的参数只能是 `u64`。

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

这里我们使用了一个 `for` 循环来处理命令行参数, 依次将每个参数赋值给 `arg` 变量, 然后执行循环体。

`std::env` 模块的 `args` 函数返回一个迭代器, 迭代器可以惰性产生每一个值, 并在迭代结束时告知我们。迭代器在 Rust 中无处不在, 标准库中还包含其他的迭代器, 例如产生 `vector` 中的每个元素、产生文件的每一行、产生通道收到的每一条消息、以及其他几乎所有可以循环处理的东西。Rust 的迭代器非常高效: 编译器通常能将它们翻译为和手写循环一样的代码。我们将会在第 15 章介绍该怎么使用它并给出一些示例。

除了和 `for` 循环一起使用之外, 迭代器还有很多可以直接使用的方法。例如, `args` 方法返回的迭代器的第一个值总是正在运行的程序名, 我们想要跳过它, 因此我们调用了迭代器的 `skip` 方法来生成一个忽略了第一个值的新迭代器。

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

这里我们调用了 `u64::from_str` 来尝试将命令行参数解析为 64 位整数。和通过 `u64` 类型的值调用的方法不同, `u64::from_str` 是一个和 `u64` 类型关联的方法, 类似于 C++ 和 Java 中的静态方法。`from_str` 函数不直接返回一个 `u64` 类型的值, 而是返回一个 `Result` 类型的值来表示解析是否成功。一个 `Result` 类型的值有两种可能:

- 一个写作 `Ok(v)`, 表示解析成功, `v` 就是解析出的值。
- 另一个写作 `Err(e)`, 表示解析失败, `e` 是解释失败的原因。

任何可能会失败的函数, 例如输入输出或其他和操作系统交互的函数, 都会返回 `Result` 值, `Ok` 时会携带成功的结果——读写的字节数、打开的文件等等——`Err` 时会携带错误码来指示错误的原因。和大多数现代编程语言不同, Rust 没有异常: 所有的错误都通过 `Result` 或者 `panic` 来处理, 这会在第 7 章中介绍。

我们使用 Rust 的 `expect` 方法来检查解析是否成功。如果结果是 `Err(e)`, `expect` 会打印出包含 `e` 的描述错误的消息。如果结果是 `Ok(v)`, `expect` 会简单的返回 `v`, 之后我们才能把它添加到 `vector` 的尾部。

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

空的数字集合没有最大公约数, 因此我们检查 `vector` 是否不为空, 如果为空就退出程序。我们使用 `eprintln!` 宏来把错误信息写入到标准错误输出流。

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

这一个循环使用了变量 `d`, 将它更新为目前的最大公约数。和之前一样, 我们将 `d` 标记为可变的, 因此我们在循环里给它赋值。

这个 `for` 循环有两个特别的地方。一个是 `for m in &numbers[1..]`, 操作符 `&` 是什么意思? 另一个是 `gcd(d, *m);`, `*m` 里的 `*` 又是什么意思? 事实上这两个细节是互补的。

到目前为止, 我们的代码只操作过像整数这类固定内存大小的值。但是现在我们要迭代一个 `vector`, 它的值可能是任意大小——有可能非常大。在处理这类值时 Rust 是很谨慎的: 它想让程序员自己控制对内存的消耗、明确每个值的生命周期, 同时确保当内存不再被需要时立即释放内存。

因此当我们在迭代时, 我们想告诉 Rust 这个 `vector` 的所有权仍然属于 `numbers`, 我们只是借用它的值来进行循环。`&numbers[1..]` 中的 `&` 运算符借用了 `vector` 中从第二个元素开始到最后一个元素的引用。`for` 循环迭代引用的那些元素, 每次迭代中用 `m` 借用每一个元素。`*m` 中的 `*` 运算符解引用了 `m`, 返回了它所指向的值, 也就是我们传递给 `gcd` 的第二个值。最后, 因为 `numbers` 拥有 `vector` 的所有权, 当 `numbers` 离开 `main` 的作用域时 Rust 会自动释放它的内存。

Rust 对所有权和引用的规则是 Rust 的内存管理和安全并发的关键。我们将在[第4章](#)讨论它们, 并在[第5章](#)讨论它们的伙伴。

要想舒服的使用 Rust, 你必须习惯这些规则, 但在这篇概览中, 你只需要知道 `&x` 是借用 `x` 的引用, `*r` 返回引用 `r` 指向的值。

继续我们的程序:

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

迭代完 `numbers` 的元素之后，程序把结果打印到标准输出流。`println!` 宏接收一个模板字符串，用剩余参数替换掉模板字符串里的`{...}`，并把结果写入到标准输出流。

与 C 和 C++ 中的 `main` 函数成功执行结束时要返回 0、执行失败时返回非 0 不同，Rust 假设不管 `main` 返回什么值都代表成功执行结束。只有当显式调用 `expect` 或 `std::process::exit` 之类的才会导致程序以错误的状态码终止。

`cargo run` 命令允许我们向程序传递参数，因此我们可以测试我们的命令行程序：

```
$ 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
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 ...
```

我们在这一节中用到了 Rust 标准库中的一小部分特性。如果你对其他的部分很好奇，我们强烈建议你去尝试 Rust 的在线文档。它的搜索功能很有用，甚至还包括到源代码的链接。安装 Rust 时 `rustup` 命令会自动在你的计算机上安装一份文档的拷贝。你可以在 Rust 的[网站](#)上查阅标准库的文档，或者通过如下命令在你的浏览器中查阅：

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

2.5 提供 web 页面

Rust 的一个强项就是发布在网站[crates.io](#)上的可以自由使用的包。`cargo` 命令让你可以方便的使用 `crates.io` 的包：它将会下载正确的版本并按需构建和更新包。一个 Rust 包，无论是库还是可执行文件，都被称为一个 *crate*。Cargo 和 `crates.io` 都是因为这个术语而得名。

为了展示它的工作方式，我们将会使用 `actix-web` 网络框架 crate，`serde` 序列化 crate 和其它它们依赖的 crate 来构建一个简单的 web 服务器。如图 2-1 所示，我们的网站将会提示用户输入两个数字，然后计算它们的最大公约数。

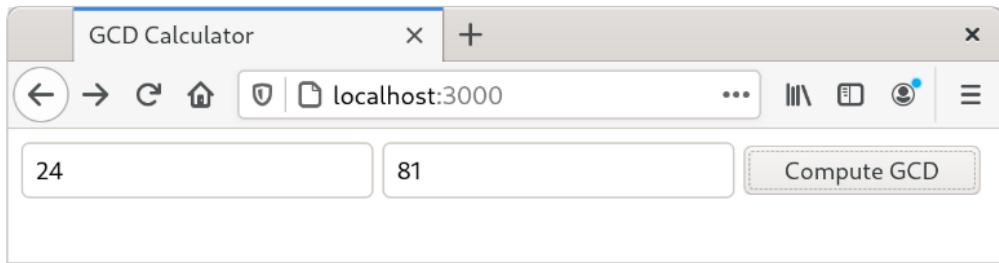


图 2-1: 提供计算最大公约数功能的 web 页面

首先，我们需要使用 Cargo 创建一个新的包，名称为 `actix-gcd`:

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

然后，我们将编辑项目中的 `Cargo.toml` 文件来列举出我们需要的包，它的内容如下所示：

```
[package]
name = "actix-gcd"
version = "0.1.0"
authors = ["You <you@example.com>"]
edition = "2018"

# See more keys and their definitions at
# https://doc.rust-lang.org/cargo/reference/manifest.html

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

`Cargo.toml` 中 `dependencies` 节的每一行都有一个 crates.io 上的 crate 的名字和需要使用的版本。在这个例子中，我们需要 `actix-web` crate 的 1.0.8 版本和 `serde` crate 的 1.0 版本。crates.io 上这两个 crate 可能还有更新的版本，但是通过指定我们测试成功的特定版本，可以保证即使这两个 crate 发布了新的版本，代码仍然可以正常工作。我们将会在第 8 章中详细讨论版本管理。

crate 有一些可选的特性：这些特性是有些用户可能用不到、但仍然需要包含在 crate 中的部分接口或实现。`serde` 提供了一个非常简洁的方式来处理 web 表单中的数据，但是根据 `serde` 的文档，只有当我们选择了这个 crate 的 `derive` 特性，才可以使用这种方式，因此我们在 `Cargo.toml` 文件中指定了这个特性。

注意我们只需要指明那些我们直接使用的 crate，cargo 会自动下载它们依赖的其他 crate。

在我们的第一个版本中，我们将保持 web 服务器的简洁：它将只提供一个页面，提示用户输入要计算的数字。将 `actix-gcd/src/main.rs` 中的内容替换为如下：

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

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

    println!("Servering 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>
            "#
        )
}
```

我们以一条 `use` 声明开始，引入一些 `actix-web` 的定义。当我们写下 `use actix_web::{...}` 时，花括号里的每一个名称都被导入到作用域中，这样我们就可以直接使用 `HttpResponse`，而不用每次都写出全名 `actix_web::HttpResponse`。（我们稍后才会使用 `serde` crate）

我们的 `main` 函数很简单：它首先调用 `HttpServer::new` 来创建一个服务器，这个服务器会响应对 "/" 路径的 `get` 方法；然后打印出一条消息；最后让服务器监听本地机器上的 3000 TCP 端口。

我们传递给 `HttpServer::new` 的参数是 Rust 的闭包表达式 `|| { App::new() ... }`。闭包

是一种可以被当作函数来调用的值。这里定义的闭包没有参数，但如果需要参数的话，要写在 || 之间。{ ... } 是闭包的函数体。当我们启动服务器时，Actix 会启动一个线程池来处理到达的请求。每个线程都会调用我们的闭包来获取一份 App 的拷贝，App 的值将告诉它如何路由和处理请求。

闭包会调用 App::new 来创建一个新的 App 并调用它的 route 方法来添加一个路径 "/" 的路由。web::get().to(get_index) 为这个路由了添加处理函数，这样当遇到 HTTP GET 请求时会调用函数 get_index。route 方法在增加新的路由之后会返回调用它的 App 自身。因为闭包的结尾处没有分号，因此 App 就是闭包的返回值，HttpServer 线程将会使用它。

get_index 函数构建了一个 HttpResponse 类型的值作为 HTTP GET / 请求的响应。HttpResponse::Ok() 代表 HTTP 200 OK 状态，表示请求被成功处理。我们还调用了它的 content_type 和 body 方法来填充相应的细节，这两个方法都会返回调用它们的 HttpResponse。最后，body 方法的返回值作为 get_index 的返回值。

因为响应文本中包含很多双引号，所以我们使用了 Rust 的 raw string 语法：开头是字母 r+0 个或多个井号 (#字符) + 一个双引号，然后是 string 的内容，最后以另一个双引号加上和开头处相同数量的井号结尾。raw string 中出现的任何字符都不会被转义，包括双引号；事实上，像 \" 这样的转义序列也不会被转义。我们可以通过增多井号的数量来保证终止的标记不会出现在字符串内容中。

编写完 main.rs 之后，我们可以使用 cargo run 命令来运行它：它会自动获取所需的 crate 并编译它们，然后编译我们自己的程序，再链接在一起，最后运行程序：

```
$ 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-htt p 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...
```

到此我们已经可以访问给定的 URL 并看到如图 2-1 所示的页面。

不幸的是，点击 GCD 并不会做任何事，而且会把我们的浏览器导航到一个空页面。接下来让我们来修复它，通过向我们的 App 添加另一个路由来处理我们表单的 POST 请求的响应。

终于到了使用我们在 `Cargo.toml` 中列出的 `serde` crate 的时候了：它提供了一种帮助我们处理表单数据的简单方法。首先，我们需要在 `src/main.rs` 中添加如下 `use` 声明：

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

Rust 程序员通常会把 `use` 声明集中在文件开始处，但这并不是必须的：Rust 允许 `use` 声明以任何顺序出现，只要它们出现在正确的嵌套层级。

接下来让我们定义一个 Rust 结构体类型来表示我们希望从表单中接收到的数据：

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

这里定义了一个新的类型叫做 `GcdParameters`，它有两个字段 `n` 和 `m`，都是 `u64` 类型，和 `gcd` 函数的参数类型保持一致。

`struct` 定义上方的注解是一个属性，类似于我们之前用来标记为测试函数时使用的 `#[test]` 属性。在一个类型定义前加上 `#[derive(Deserialize)]` 属性可以告诉 `serde` crate 在程序编译时自动为该类型生成代码来把 HTML POST 请求的表单数据转为该类型的值。事实上，这个属性可以让你从几乎所有结构化的数据：JSON、YAML、TOML 或其他文本或二进制格式中解析出一个 `GcdParameters` 类型的值。`serde` crate 还提供一个 `Serialize` 属性来做相反的事，即把 Rust 值写成结构化的格式。

有了上面的定义，我们可以简单地写出我们的处理函数：

```
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)
}

```

作为 Actix 请求的处理函数，它的参数的类型必须是 Actix 知道怎么从中提取出 HTTP 请求的类型。我们的 `post_gcd` 函数有一个参数 `form`，它的类型是 `web::Form<GcdParameters>`。当且仅当 `T` 可以由 HTML 的 POST 表单数据反序列化出来时，Actix 才知道怎么从 `web::Form<T>` 类型中提取出值。因为我们在 `GcdParameters` 类型的定义前加上了 `#[derive(Deserialize)]` 属性，所以 Actix 可以从表单数据中反序列化出它，因此请求的处理函数可以使用 `web::Form<GcdParameters>` 作为参数。这些类型和函数之间的关系都是在编译期处理的，如果你用了 Actix 不知道该如何处理的类型作为参数的类型，Rust 编译器将会立刻告诉你这个错误。

再看 `post_gcd` 的实现：如果有参数的值为 0，这个函数会返回一个 HTTP 401 BAD REQUEST 错误，因为这种情况下我们的 `gcd` 函数会 panic。然后，它使用 `format!` 宏创建了一个响应。`format!` 宏类似于 `println!` 宏，只不过它不会把字符串写入到标准输出，而是会返回字符串。当响应的文本就绪后，`post_gcd` 用一个 HTTP 200 OK 响应来包装它，设置好 content type 之后，就把它返回到请求方。

我们还要把 `post_gcd` 注册为表单的处理函数。我们要把 `main` 函数替换为如下内容：

```

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

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

```

唯一的变化在于多了一个 `route` 的调用，把 `web::post().to(post_gcd)` 注册为路径 "/gcd" 的 handler。

最后剩余的部分是我们之前编写的 `gcd` 函数，要把它添加到 `actix-gcd/src/main.rs` 文件中。完成这些之后，你可以停止之前运行的服务器并重新启动程序：

```

$ 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，通过访问 `http://localhost:3000`，输入一些数字，然后点击 Compute GCD 按钮，你应该能看到如下的结果（图 2-2）。

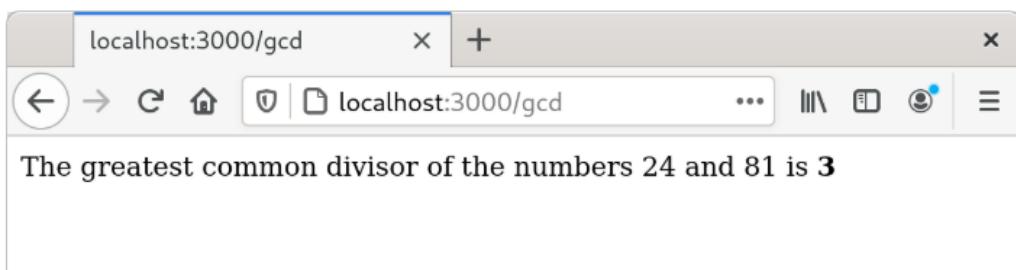


图 2-2: 显示计算 GCD 结果的 web 页面

2.6 并发

Rust 的另一项长处是对并发编程的支持。Rust 中用于避免内存安全问题的规则同样可以保证线程之间在没有数据竞争的情况下共享数据。例如：

- 如果你使用互斥锁来同步需要修改共享数据结构的线程，Rust 保证只有当你持有锁的情况下才可以访问数据，并且当你处理完数据后自动释放锁。在 C 和 C++ 中，互斥锁和要保护的数据之间的关系一般都写在注释里。
- 如果你想在几个线程中共享只读数据，Rust 保证你不可能意外修改数据。在 C 和 C++ 中，类型系统也可以帮我们做到这一点，但很容易出错。
- 如果你把一个数据结构的所有权从一个线程转移到另一个线程，Rust 会保证你确实已经失去了对它的访问权。在 C 和 C++ 中，需要由你自己来保证发送线程不会再次访问该数据。如果你没有正确做到这些，那么结果将会取决于处理器的缓存和你最近写入了多少内存。

在这一节中，我们将带领你编写你的第二个多线程程序。

你已经编写过第一个多线程程序了：你用来实现最大公约数服务器的 Actix web 框架使用了线程池来运行请求的处理函数。如果服务器同时收到很多请求，它会立刻在若干个线程里运行 `get_form` 和 `post_gcd` 函数。这可能让我们有些震惊，因为当我们编写那些函数时我们完全没有并发的概念。

不过 Rust 保证这么做是安全的，不管你的服务器变得多复杂：只要你的程序能够编译，它

就能免于数据竞争。所有的 Rust 函数都是线程安全的。

这一节的程序将绘制曼德勃罗集，它是一种通过迭代一个简单的复数函数得到的分形。绘制曼德勃罗集经常被称为 *embarrassingly parallel* 算法，因为线程之间的通信太过简单；我们将会在第 19 章中讲述更为复杂的模式，但这个例子已经可以展示出一些核心的部分。

开始之前，我们要创建一个新的 Rust 项目：

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

所有的代码都会添加到 `mandelbrot/src/main.rs` 里，我们将会把一些依赖添加到 `mandelbrot/Cargo.toml`。

在开始实现并行的曼德勃罗集之前，我们需要先描述一下我们准备实现的计算过程。

2.6.1 曼德勃罗集到底是什么

了解这一点可以帮我们在阅读代码时更加清楚它要做什么，因此首先让我们先探讨一些纯数学知识。我们将以一个简单的例子开始，并逐渐添加复杂的细节，直到我们讲到曼德勃罗集的核心。

首先这里有一个无限循环，用 Rust 的语法来写的话就是一个 `loop` 语句：

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

事实上，Rust 可能会看出 `x` 的值从来没有被使用过，因此并不执行计算。但一开始，首先让我们假设代码按照我们所写的运行。`x` 的值将会发生什么变化？任何小于 1 的数平方都会变得更小，因此它会接近于 0；1 的平方还是 1；大于 1 的数平方会变得更大，因此它会接近无限大；负数的平方将会使它变为正数，然后它的变化就和前面说的一样（图 2-3）。

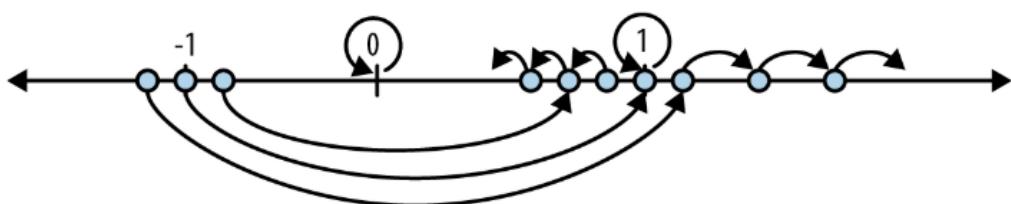


图 2-3：重复平方一个数的结果

因此根据传递给 `square_loop` 的值不同，`x` 可能会保持 0 或者 1 不变，也可能接近 0 或者接近无限大。

现在让我们考虑一个稍有不同的循环：

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

这一次，`x` 从 0 开始，并且每次迭代时平方之后再加上 `c`。这导致我们更难看出 `x` 会怎么变化，但一些实验表明如果 `c` 大于 0.25 或者小于 -2.0，那么 `x` 将会变得接近无穷大；否则它会保持在接近 0 的某个区间。

更进一步，如果不使用 `f64` 类型的值，考虑使用复数来执行相同的循环。crates.io 上的 `num` crate 提供了一个我们可以使用的复数类型，因此我们必须在我们程序的 `Cargo.toml` 文件的 `[dependencies]` 节中添加一行对 `num` 的引用。这是到目前为止这个文件里的全部内容（稍后我们还会添加一些内容）：

```
[package]
name = "mandelbrot"
version = "0.1.0"
authors = ["You <you@example>"]
edition = "2018"

# See more keys and their definitions at
# https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
num = "0.4"
```

现在我们可以写出循环的倒数第二个版本：

```
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;
    }
}
```

用 `z` 表示复数是一种习惯，因此我们重命名了变量。表达式 `Complex { re: 0.0, im: 0.0 }` 创建了一个 `num` crate 的 `Complex` 类型的复数 0 值。`Complex` 是一个 Rust 的结构体类型，定义类似于如下：

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

上面的代码定义了一个叫 `Complex` 的结构体，有两个字段：`re` 和 `im`。`Complex` 是一个泛型结构体：你可以将类型名后的 `<T>` 理解为“任意类型 `T`”。例如，`Complex<f64>` 是一个 `re` 和 `im` 字段都是 `f64` 类型的复数，`Complex<f32>` 则是 32 位浮点数，等等。有了这个定义之后，类似于 `Complex { re: 0.24, im: 0.3 }` 这样的表达式将会产生一个 `re` 字段初始化为 0.24、`im` 字段初始化为 0.3 的 `Complex` 类型的值。

`num` crate 为 `Complex` 类型的值提供了 `*`、`+` 和其他的算术运算符，因此剩余的代码就和之前一样，除了它现在是操作复数类型，而不是实数。我们将会在[第 12 章](#)解释怎么为自定义类型定义运算符。

最后，我们终于完成了纯数学的部分。曼德勃罗集被定义为使得 `z` 不会变为无穷大的 `c` 的集合。我们最初版本的循环很容易预测结果：任何大于 1 或小于 -1 的数都不满足曼德勃罗集的条件。每次迭代平方之后再 `+ c` 则让程序的行为更难预测：正如我们之前所说，如果 `c` 的值大于 0.25 或者小于 -2 会导致 `z` 变为无穷大。但如果扩展到复数领域事情就会变得有趣起来，并且会生成漂亮的图形，我们将绘制这个图形。

因为一个复数 `c` 同时有实部 `c.re` 和虚部 `c.im`，因此我们将它当作笛卡尔坐标系中的 `x` 和 `y` 坐标，然后如果 `c` 在曼德勃罗集里就把点设置为黑色，否则设置为白色。因此对于我们图片中的每个像素点，我们必须运行复数版本的循环，看看它最终是会变为无穷大还是围绕在原点附近，并以此判断点的颜色。

无限循环会运行很长时间，但有两个技巧可以加快它。首先，如果我们放弃无限循环而是只迭代有限次，实验表明我们仍然可以得到一个不错的近似结果。迭代多少次取决于我们想要多精确的绘制边界。其次，已经被证明的一点是，如果 `z` 有一次离开了以原点为中心半径为 2 的圆形范围，它将会逐渐变为无穷大。因此这是我们最终版本的循环，也是我们程序的核心：

```
use num::Complex;
```

```

/// 尝试判断`c`是否在曼德勃罗集里，最多迭代`limit`次。
///
/// 如果`c`不在曼德勃罗集里，就返回`Some(i)`，其中`i`是
/// `z`离开以原点为圆心、2为半径的圆所需的迭代次数。
/// 如果`c`似乎在曼德勃罗集里（更确切地说是迭代了`limit`次
/// 之后仍无法证明`c`不在曼德勃罗集里），就返回`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
}

```

这个函数接收我们想测试是否在曼德勃罗集中的复数 `c` 和迭代次数上限作为参数。函数的返回值是 `Option<usize>` 类型。Rust 的标准库中 `Option` 类型定义如下：

```

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

```

`Option` 是一个枚举类型，通常被称为 *enum*，因为它的定义列举出了一个该类型的实例可能的值：对于任何类型 `T`，一个 `Option<T>` 类型的值要么是 `Some(v)`（其中 `v` 是 `T` 类型的值），要么是 `None`，表示没有可用的 `T` 类型的值。类似于我们之前讨论的 `Complex` 类型，`Option` 也是泛型类型：你可以用 `Option<T>` 来代替任何类型的可选值。

在我们的例子中，`escape_time` 返回一个可选的 `Option<usize>` 来表示 `c` 是否在曼德勃罗集里，并且如果不在的话，我们希望知道迭代多少次才能判断出来它不在。如果 `c` 不在曼德勃罗集里，`escape_time` 会返回 `Some(i)`，其中 `i` 是 `z` 离开半径为 2 的圆所需的迭代次数。否则，如果 `c` 在曼德勃罗集里，`escape_time` 会返回 `None`。

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

更早的例子中展示过使用 `for` 循环迭代命令行参数和 `vector` 的元素；这里的 `for` 循环迭代一个从 `0` 到 `limit`（但不包含）的范围内的整数。

`z.norm_sqr()` 方法会返回 `z` 与原点的距离的平方。为了判断 `z` 是否离开了半径为 2 的圆，我们不需要求平方根，只需要比较距离的平方和 4.0 即可，这样会更快一些。

你可能已经注意到我们在函数定义的上方使用了`///`来标记注释行；`Complex`结构体的成员上方的注释也是以`///`开头。这些是文档注释；`rustdoc`工具知道该如何解析它们、收集它们中对代码的解释、并生成在线文档。Rust 标准库的文档就是用这种方式书写的。我们将在[第 8 章](#)中详细介绍文档注释。

程序的剩余部分就是在多大的分辨率范围内计算曼德勃罗集和把负载分发到若干线程来加快计算。

2.6.2 解析成对的命令行参数

程序将会接收几个命令行参数来控制生成图片的分辨率和图中曼德勃罗集所占的部分。因为这些命令行参数都遵循相同的形式，这里定义了一个解析它们的函数：

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

/// 将`s`解析为一个坐标对，例如`"400x6000"`、或者`"1.0,0.5"`

///
/// 确切地说，`s`的形式应该是<left><sep><right>，其中<sep>是由
/// `separator`参数指定的字符，<left>和<right>都是可以被`T::from_str`
/// 解析的字符串，`separator`必须是个ASCII字符。
///
/// 如果`s`的格式正确，就返回`Some<(x, y)>`。
/// 如果不能正确解析，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",      ','), 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)> {
```

你可以将语法 `<T: FromStr>` 读作：“对于任意实现了 `FromStr` trait 的类型 `T...`”。这可以让我们方便地一次定义一系列函数：`parse_pair::<i32>` 是一个把字符串解析为 `i32` 值对的函数，`parse_pair::<f64>` 是一个把字符串解析为浮点数对的函数，等等。这很像 C++ 中的函数模板。一个 Rust 程序员会把 `T` 称为 `parse_pair` 的类型参数。当你使用泛型函数时，Rust 通常能推导出类型参数，你不需要像我们测试代码中那样指明类型。

我们的返回类型是 `Option<(T, T)>`：要么是 `None` 要么是 `Some((v1, v2))`，`(v1, v2)` 是一个有两个值的元组，两个值类型都是 `T`。`parse_pair` 函数没有使用显式的 `return` 语句，因此它的返回值就是函数体中最后一条表达式的值：

```

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

```

`String` 类型的 `find` 方法会在字符串中搜索匹配 `separator` 的字符。如果 `find` 返回 `None`，表示字符串中不存在分隔字符，整个 `match` 语句将会求值为 `None`，表示解析失败；否则，`index` 就是分隔字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
}

```

从这里就能看出 `match` 表达式的强大，要匹配的参数是这个元组表达式：

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

表达式 `&s[..index]` 和 `&s[index + 1..]` 是字符串切片，分别是分隔字符之前和之后的部分。参数 `T` 的类型关联的 `from_str` 函数接收参数并尝试将它们解析为 `T` 类型的值，最后生成一个元组。我们要匹配的模式是：

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

只有当元组的两个值都是 `Result` 类型的 `Ok` 值才可以匹配这个模式，表示两个解析都成功了此时 `Some((l, r))`。就是 `match` 表达式的值，也是函数的返回值。

```
_ => None
```

通配模式 `_` 匹配任何情况并忽略值。如果我们到达这个地方，说明 `parse_pair` 失败了，因此求值为 `None`，并作为函数的返回值。

现在我们已经有了 `parse_pair`，很容易就可以写出一个解析一对浮点数并返回一个 `Complex<f64>` 值的函数：

```
/// 把一对逗号分隔的浮点数解析为一个复数。
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` 函数会调用 `parse_pair` 函数，如果参数成功解析就构建一个 `Complex` 值，如果失败就向调用者传播。

如果你仔细阅读代码，你可能已经注意到我们使用了一种简写的方式来构建 `Complex` 值。用同名的变量来初始化结构体的字段是很常见的，因此相比于强迫你写 `Complex { re: re, im: im }`，Rust 允许你简写为 `Complex { re, im }`。这是受 JavaScript 和 Haskell 中类似写法的启发。

2.6.3 将像素映射到复数

我们的程序需要处理两个相关联的坐标空间：输出图片中的每个像素对应复平面中的一个点。这两个空间的关系取决于我们要绘制的曼德勃罗集的部分和图片的分辨率，这些都是由命令行参数指定的。下面的函数实现从图片空间到复数空间的转换：

```
/// 给定输出图片中一个像素的行和列，返回复平面中相应的点。
///
/// `bounds` 是一个指定图片的宽和高的值对。
```

```

/// `pixel`是一个(行,列)对,指定图片中的某个像素。
/// `upper_left`和`lower_right`参数是复平面上的点,
/// 指定我们的图像覆盖的区域。
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
        //为什么这里是减法?当我们向下时pixel.1会增大,但虚部会减小。
    }
}

#[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 展示了 `pixel_to_point` 进行的计算。

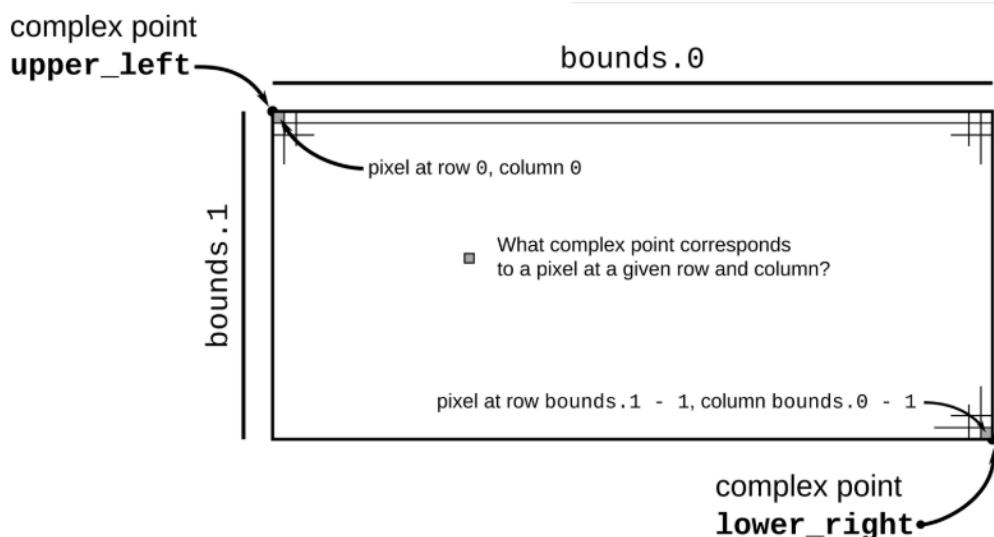


图 2-4: 复数平面和图片像素的关系

`pixel_to_point` 中的代码只是简单的计算，因此我们不会详细解释。然而，还是有一点要提一下。下面的表达式展示了引用元组元素的方式：

```
pixel.0
```

上面的表达式引用了元组 `pixel` 的第一个元素。

```
pixel.0 as f64
```

这是 Rust 的类型转换语法：把 `pixel.0` 转换为 `f64` 类型的值。与 C 和 C++ 不同，Rust 禁止数字类型之间的隐式转换，你必须显式写明类型转换。这可能会很枯燥，但显式指明类型转换有时会很有用。整数之间的隐式转换看起来似乎没有问题，但历史上它们曾在真实世界中的 C 和 C++ 代码中导致了很多 bug 和安全漏洞。

2.6.4 绘制曼德勃罗集

为了绘制曼德勃罗集，对于图像中的每个像素点，我们需要用它在复平面中对应的点来调用 `escape_time`，然后根据结果决定像素点的颜色：

```
/// 把一个矩形区域内的曼德勃罗集渲染到像素的缓冲区里。
///
/// `bounds` 参数指定了缓冲区 `pixels` 的宽度和高度，每个字节存储一个
/// 灰度像素。`upper_left` 和 `lower_right` 参数指定了复平面中对应
/// 像素缓冲区左上角和右下角的两个点。
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
                };
        }
    }
}
```

这一段代码看起来应该很熟悉：

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

```

如果 `escape_time` 函数判定 `point` 属于曼德勃罗集，`render` 将会把相应像素设为黑色。否则，`render` 根据点要迭代多少次才能离开圆来决定像素的颜色有多深。

2.6.5 写入图片文件

`image` crate 提供读写很多图片格式的函数，还有一些对图片进行简单操作的函数。它包含了一个 PNG 图片格式的编码器，我们的程序将用它来保存最后的结果。为了使用 `image`，在 `Cargo.toml` 的 `[dependencies]` 节添加下面这一行：

```
image = "0.13.0"
```

然后再加上下面的代码：

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

/// 把缓冲区`pixels`写入到文件`filename`，它的宽和高由`bounds`指定。
fn write_image(filename: &str, pixels: &[u8], bounds: (u32, u32))
    -> 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(())
}
```

这个函数的操作非常直观：它打开一个文件，然后尝试写入图片。我们向编码器传递了 `pixels` 中的像素数据，图片的宽和高则来自 `bounds`，编码器的最后一个参数表明该如何解析 `pixels` 的字节：`ColorType::Gray(8)` 表示每个字节都是 8 比特的灰度值。

这些都很直观，有趣的地方是它处理错误的方式。如果我们遇到了一个错误，我们需要把错误返回给调用者。正如我们之前提到的那样，Rust 里可能会失败的函数应该返回一

个 `Result` 值，成功时是 `Ok(s)` (`s` 是成功后的结果)，或者失败时是 `Err(e)` (`e` 是错误码)。那么 `write_image` 的成功和错误类型是什么呢？

在一切顺利的情况下，我们的 `write_image` 函数没有需要返回的值，因为它把数据都写入到了文件里。因此它的成功类型是单元类型 `()`，这么称呼是因为这种类型只有唯一的值，这个唯一的值也被写作 `()`。单元类型类似于 C 和 C++ 中的 `void`。

当出现错误时，不管是因为 `File::create` 不能创建文件还是 `encoder.encode` 不能写入图片，I/O 操作都会返回一个错误码。`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>`，因此两者有相同的错误类型：`std::io::Error`。我们的 `write_image` 函数也把错误类型设置为同样的类型，当两者中任何一个出现错误时，函数应该立即返回，并把描述错误信息的 `std::io::Error` 向上传播。

因此为了正确地处理 `File::create` 的结果，我们需要 `match` 它的返回值，类似于这样：

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

当成功时，将 `output` 赋值为 `Ok` 里携带的 `File` 值。当失败时，把错误传播给调用者。

这种模式的 `match` 语句在 Rust 中太过常见，因此 Rust 提供了一个 `?` 操作符作为这种模式的缩写。因此，我们不需要再每次都显式地写出这样的逻辑来处理可能的错误，我们可以用下面这种等价但简洁的多的语法：

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

如果 `File::create` 失败了，`?` 操作符会使 `write_image` 函数返回，并把错误向上传播。否则，`output` 将被赋值为成功打开的 `File`。

NOTE

初学者的一个常见错误是尝试在 `main` 函数中使用 `?`。然而，因为 `main` 函数自身不返回值，所以这种写法是错误的。正确的做法是使用 `match` 表达式或者类似于 `unwrap` 或 `expect` 这样的缩写方法。还有一种方式是把 `main` 改为返回 `Result`，我们之后会介绍这种方式。

2.6.6 一个并发的曼德勃罗集程序

所有的部分都已经就位，我们将在 `main` 函数中加入并发的部分。首先，一个简单的无并发版本如下：

```
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 -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");
}
```

把命令行参数收集为一个 `String` 的 `vector` 之后，我们逐个解析每个参数，然后开始计算。

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

宏调用 `vec![v; n]` 会创建一个有 `n` 个元素，并且每个元素都被初始化为 `v` 的 `vector`，因此上面的代码会创建一个长度为 `bounds.0 * bounds.1`，元素全为 0 的 `vector`，其中 `bounds` 是通过解析命令行参数得到的图片的分辨率。我们将把这个 `vector` 用作数组来存储一节的灰度像素值，如图 2-5 所示。

下一行有趣的代码如下：

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

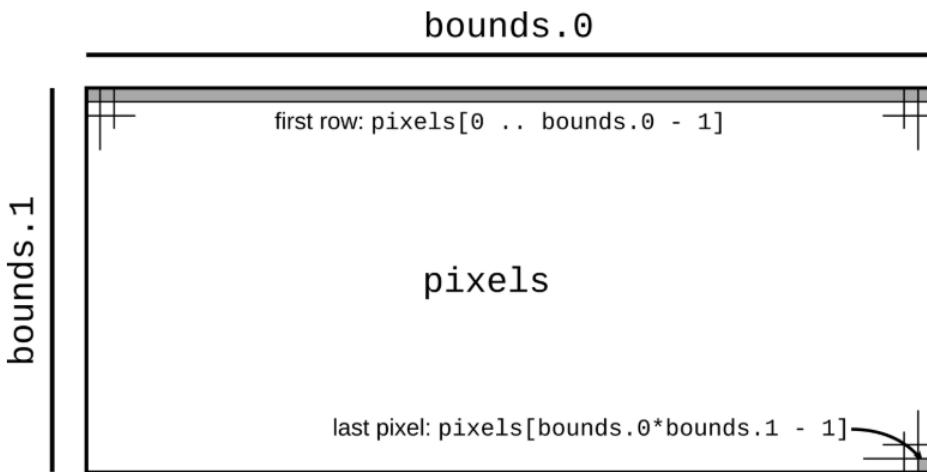


图 2-5: 将 vector 用作矩形像素数组

这里调用了 `render` 函数来计算图片。表达式 `&mut pixels` 获取像素缓冲区的可变引用，这样可以让 `render` 用计算出的灰度值填充像素缓冲区，同时 `pixels` 仍然保有 `vector` 的所有权。剩余的参数传递了图像的维度和我们要绘制的复平面对应的矩形。

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

最后，我们把像素缓冲区以 PNG 格式写入到磁盘上。在这个例子中，我们传递了一个缓冲区的共享（不可变）引用，因为 `write_image` 不需要修改缓冲区的内容。

到了这里，我们可以以 `release` 模式构建并运行应用，这样可以允许编译器进行更多优化。几秒之后，它将会生成一幅漂亮的图片 `mandel.png`：

```
$ 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
```

这条命令会创建一个叫 `mandel.png` 的文件，你可以用系统的图片查看器或者在浏览器中打开。如果一切正常的话，它应该看起来类似于图 2-6。

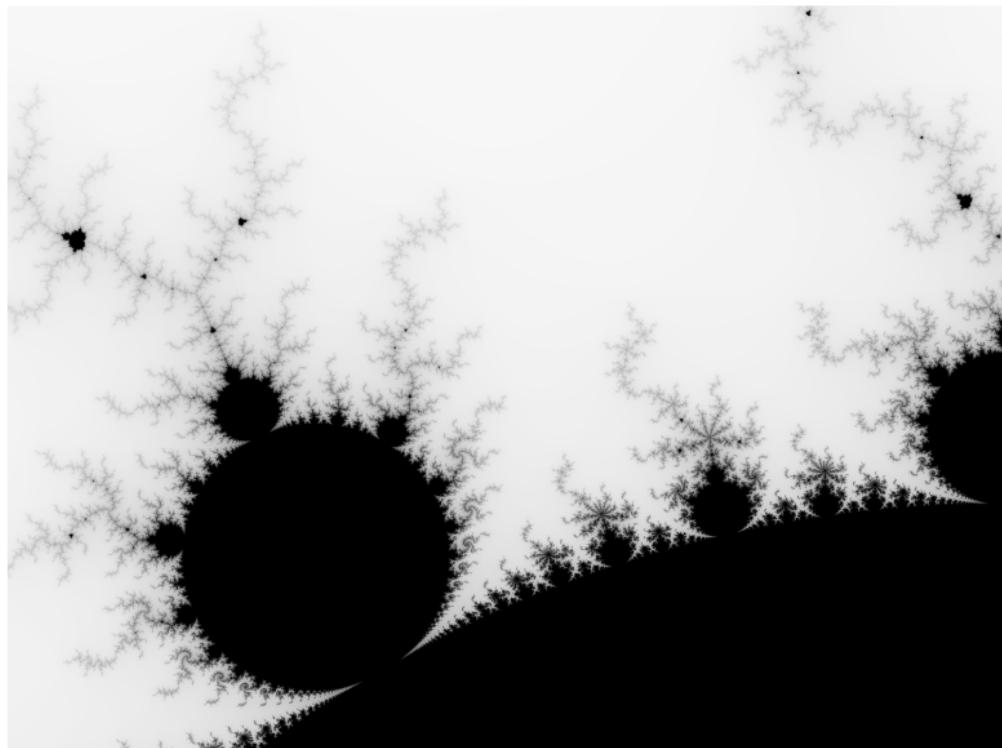


图 2-6: 曼德勃罗集程序的结果

在上面的测试中，我们使用了 Unix 的 `time` 程序分析了程序的运行时间：它需要花费 5 秒才能对图片中的每个像素进行曼德勃罗计算。但几乎所有现代机器都有很多个处理核心，而这个程序只利用到了一个。如果我们可以把计算任务分配给机器提供的所有计算资源，我们将能更快的完成图片的计算。

为了实现这一点，我们把图片分成了几个部分，每一个处理单元负责计算一个部分的像素值。为了简单，我们把图片分成水平的条带，如图 2-7 所示。当所有的处理单元都计算完成之后，我们再把结果写入到磁盘。

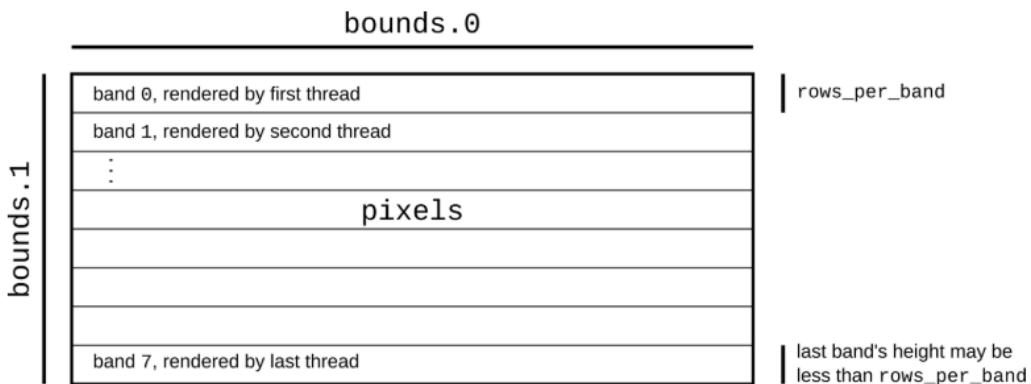


图 2-7: 把像素缓冲区分成若干条带来并行渲染

`crossbeam` crate 提供了很多并行的设施，包括我们恰好需要的作用域线程 (*scoped thread*)。为了使用它，我们需要在 `Cargo.toml` 中加上下面这一行：

```
crossbeam = "0.8"
```

然后我们需要删除调用 `render` 的那一行，将它替换为如下内容：

```
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();
}

```

和之前一样对这段代码进行分解：

```

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

```

这里我们决定使用 8 个线程¹。然后我们计算每一个条带包含多少个像素，我们将结果加一来保证即使图片的高度不是 `threads` 的倍数的情况下，这些条带仍然能包含整个图像。

```

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

```

这里我们把像素缓冲区划分为了若干条带。缓冲区的 `chunks_mut` 方法返回了一个产生可变引用的迭代器，每一个引用都是缓冲区的切片，这些引用互相不重叠，切片里包含 `rows_per_band * bounds.0` 个像素——即 `rows_per_band` 行的像素。`chunks_mut` 产生的最后一个切片包含的行数可能会比较少，但每一行的像素数量都是相同的。最后，使用迭代器的 `collect` 方法创建一个 `vector` 来保存这些可变且互不相交的切片。

现在我们可以使用 `crossbeam` 库了：

```

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

```

参数 `|spawner| { ... }` 是一个 Rust 的闭包，这个闭包接收一个参数 `spawner`。注意，和使用 `fn` 声明的函数不同，我们不需要指明闭包中的参数的类型；Rust 将会自动推断参数和返回值的类型。在这个例子中，`crossbeam::scope` 会调用这个闭包，并将闭包中创建新线程需要使用的值作为 `spawner` 参数传入。`crossbeam::scope` 会等待所有的线程都结束之后才会返回。这样可以确保线程不会在 `pixels` 离开作用域之后再访问它的数据，并且可以确保当 `crossbeam::scope` 返回时，计算图片的过程已经完成。如果一切顺利，`crossbeam::scope` 会返回 `Ok(())`，但如果任何一个线程 `panic` 了，它会返回一个 `Err`。我们对返回的 `Result` 调用 `unwrap`，这样当遇到线程 `panic` 的情况时程序也会 `panic`。

¹`num_cpus` crate 提供了一个函数返回当前系统中可用的 CPU 核的数量。

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

这里我们迭代了像素缓冲区的条带。`into_iter()` 迭代器在循环体的每一次迭代时获取一个条带的所有权，以确保同一时间内只有一个线程可以写入它。我们将在第5章详细讲解它是如何工作的。之后，`enumerate` 适配器产生一个包含迭代元素和它的索引的 tuple。

```
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);
```

给定索引和每个条带的实际大小（再次提醒一下最后一个条带可能比其他的要短），我们可以构造出一个 `render` 所需的矩形区域，不同的是现在每一个区域指向缓冲区的条带，而不是整个图像。类似于之前，我们再次调用 `pixel_to_point` 函数来找出区域的左上角和右下角在复平面中的位置。

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

最后，我们创建了一个线程来运行闭包 `move |_| { ... }`。前边的 `move` 关键字表示这个闭包需要获取它用到的变量的所有权；在闭包获取了所有权之后，就只有闭包可以使用可变的切片 `band`。参数列表 `|_|` 表示这个闭包有一个参数，但不会用到这个参数（另一个用于嵌套线程的 `spawner`）。

如同我们之前提到的一样，`crossbeam::scope` 调用会保证在它返回之前所有的线程都已经结束，这意味着我们可以安全的把图片保存到文件，正如我们接下来要做的一样。

2.6.7 运行曼德勃罗集绘制器

我们在程序中使用了好几个外部 crate：`num` 用于复数的数学计算、`image` 用于写入 PNG 文件、`crossbeam` 用于创建 scoped thread。这是最终的包含所有这些依赖的 `Cargo.toml` 文件：

```
[package]
name = "mandelbrot"
version = "0.1.0"
authors = ["You <you@example.com>"]
```

```
edition = "2018"

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

这些都就绪之后，我们可以构建并运行程序：

```
$ cargo build --release
    Updating crates.io index
    Compiling crossbeam-queue v0.3.1
    Compiling crossbeam v0.8.0
    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，我们再次使用了 `time` 来查看程序运行了多长时间；注意尽管还是花费了接近 5 秒的处理器时间，但实际经过的时间只有 1.5 秒左右。这其中有一部分时间用来写入图片文件，你可以试着注释掉相应的代码并再次测量。在我们的测试笔记本上，计算曼德勃罗集的并发版本只需要大概 1/4 的时间。我们将会在第 19 章中展示如何从本质上加快速度。

和之前一样，程序会创建一个 `mandel.png` 文件。有了这个更快的版本，你可以更加轻松地通过改变命令行参数来探索曼德勃罗集。

2.6.8 安全性是不可见的

在最后，我们的最终版本的并行程序和其它语言编写的并没有本质上的不同：我们把像素缓冲区分成不同部分，每一个处理器核心都单独计算它自己的那一部分，并在全部结束时保存结果。所以 Rust 的并发支持到底有什么特殊之处？

我们在这里并没有展示不能编写的代码。我们在这一章中编写的代码正确的把缓冲区划分到了各个线程中，但这段代码还有很多不正确的变体（会导致数据竞争）；所有这些变体都不可能通过 Rust 编译器的静态检查。一个 C 或 C++ 编译器将会乐于帮助你探索带有数据竞争的程序的广阔空间；Rust 则会在错误发生之前就告诉你。

在第 4 章和第 5 章中，我们描述了 Rust 保证内存安全的规则。第 19 章将会介绍这些规则怎么保证并发的安全性。

2.7 文件系统和命令行工具

Rust 在命令行工具的世界中找到了合适的应用场景。作为一个现代、安全、高性能的系统编程语言，它给程序员提供了一套用于创建命令行接口的工具箱，这些新的接口可以替代或者扩展现有的基础工具。例如，`bat` 命令行工具提供了一个自带语法高亮并内建对分页程序的支持的`cat` 命令替代品，`hyperfine` 可以自动测量任何可以通过命令或管道运行的程序的性能。

虽然这么复杂的东西超出了本书的范围，但 Rust 使你可以轻松的涉足人体工程学命令行程序的世界。在本节中，我们将向你展示如何构建自己的搜索-替换工具，并带有彩色的输出和友好的错误信息。

开始之前，我们要创建一个新的 Rust 项目：

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

我们的程序需要两个其他的 crate：`text-colorizer` 用于在终端中创建彩色输出，`regex` 用于实际的搜索-替换功能。和之前一样，我们把这些 crate 写在 `Cargo.toml` 中来告诉 `cargo` 我们需要它们：

```
[package]
name = "quickreplace"
version = "0.1.0"
authors = ["You <you@example.com>"]
edition = "2018"

# See more keys and their definitions at
# https://doc.rust-lang.org/cargo/reference/manifest.html

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

Rust 中版本到达了 1.0 以上的 crate 都遵循“语义版本”的规则：只要主版本号 1 不变，新版本将是兼容之前版本的扩展。因此如果我们用例如 1.2 版本的 crate 测试成功，那么换成版本 1.3、1.4 之后代码将仍然能够工作；但版本 2.0 将会引入不兼容的更改。当我们简单地在 `Cargo.toml` 中把所需版本写为 1 时，`Cargo` 将会使用 2.0 之前的最新的可用版本。

2.7.1 命令行接口

这个程序的接口非常简单。它只需要 4 个参数：一个需要搜索的字符串（或正则表达式）、一个用于替换的字符串（或正则表达式）、输入文件的名称、输出文件的名称。我们将以一个包含这些参数的 struct 开始 `main.rs` 文件：

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

`#[derive(Debug)]` 属性告诉编译器生成额外的代码，以允许我们在 `println!` 中使用 `{:?}` 来格式化 `Arguments` 结构体。

当用户输入的参数数量错误时，我们需要打印出信息来解释如何使用程序。我们将使用一个叫 `print_usage` 的简单的函数来实现这一点，并且导入 `text-colorizer` 中的内容，这样我们可以添加一些颜色：

```
use text_colorizer::*;

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

简单地在字符串字面量的最后添加上 `.green()` 就可以返回一个用 ANSI 转义代码包装后的字符串，它在终端模拟器中会显示为绿色。这个字符串在打印之前会被插入到消息中。

现在我们收集并处理程序的参数：

```
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()
    }
}
```

为了获取用户传递的命令行参数，我们使用了和上面例子中相同的 `args` 迭代器。`.skip(1)` 跳过迭代器的第一个值（正在运行的程序名）因此结果只有命令行参数。

`collect()` 方法返回一个参数组成的 `Vec`。我们之后检查参数的数量是否正确，如果不正确，打印出消息并终止程序。我们再次把输出的部分消息带上颜色并使用 `.bold()` 来加粗文本。如果参数的数量正确，我们将它们存储到 `Arguments` 结构体中并返回。

之后我们添加一个 `main` 函数，函数里简单的调用 `parse_args` 并打印出结果：

```
fn main() {
    let args = parse_args();
    println!("{:?}", args);
}
```

到此，我们可以运行程序并看到它打印出正确的错误信息：

```
$ 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: expect 4, got 0
```

如果给这个程序传递一些参数，它会打印出 `Arguments` 结构体的表示：

```
$ 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" }
```

这是一个很好的开始！参数被正确提取出来并存储到 `Arguments` 结构体中。

2.7.2 读写文件

接下来，我们需要一些方法来从文件系统中获取数据，处理之后再写回到文件系统中。Rust 有健全的输入输出工具，但标准库的设计者知道读写文件是非常常见的，因此他们特意将这些工具设计得非常易用。我们需要的只是导入 `std::fs`，这样我们就可以使用 `read_to_string` 和 `write` 函数：

```
use std::fs;

std::fs::read_to_string 返回一个 Result<String, std::io::Error>。如果函数成功执行，它会产生一个 String。如果失败了，它会产生一个 std::io::Error，标准库用这个类型表示 I/O 错误。与之类似，std::fs::write 返回一个 Result<(), std::io::Error>：成功时什么也不返回，失败时返回错误详情。
```

```
use std::fs;

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 '{}': {:?}", args.filename, e);
            std::process::exit(1);
        }
    };

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

```
};  
}
```

这里，我们使用了我们之前编写的 `parse_args()` 函数，并把解析出的文件名传递给 `read_to_string` 和 `write`。 `match` 语句优雅地处理错误，打印出错误的文件名、错误的原因、还有一点颜色来引起用户的注意。

有了更新之后的 `main` 函数，我们就可以运行程序了。当然，新文件的内容和旧文件的内容完全相同：

```
$ 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`
```

程序确实读取了输入文件 `Cargo.toml`，并且确实写入到输出文件 `Copy.toml`，但因为我们没有写任何实际做查找和替换工作的代码，因此输出文件和输入文件没有任何不同。我们可以通过 `diff` 命令来检查：

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

2.7.3 查找和替换

这个程序的最终目标是实现它实际的功能：查找和替换。为了这一点，我们使用了 `regex` crate 来编译和处理正则表达式。它提供了一个叫 `Regex` 的结构体来表示一个编译后的正则表达式。`Regex` 有一个方法 `replace_all`，它的行为就像名称一样：在一个字符串中搜索每一个匹配的子串并替换为指定的模式。我们可以把这部分逻辑放入一个函数：

```
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())  
}
```

注意这个函数的返回值类型。像我们之前使用的标准库函数一样，`replace_all` 返回一个 `Result`，这一次它的错误类型由 `regex` crate 提供。

`Regex::new` 编译用户提供的正则表达式，如果给定的表达式无效时可能会失败。像在曼德勃罗集程序中一样，我们使用 `?` 来短路处理 `Regex::new` 失败的情况，但这个例子中返回一个 `regex` crate 中特定的错误类型。当正则表达式编译完后，它的 `replace_all` 方法把 `text` 中的所有匹配项替换为指定的字符串。

如果 `replace_all` 找到了匹配项，它会返回一个替换后的新的 `String`。否则，`replace_all` 返回一个指向原始文本的指针，这样可以避免不必要的内存分配和拷贝。然而在这个例子中，我们总是想要一个独立的拷贝，所以我们使用 `to_string` 方法来获取一个 `String`，然后用 `Result::Ok` 包装之后返回。

现在，是时候在 `main` 函数的代码中插入我们的新函数了：

```
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);
        }
    };
}
```

有了最终的版本，程序已经完全就绪，你可以像下面这样测试它：

```
$ 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
```

当然，错误处理也已经就绪，可以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，这个简单的演示中缺少了很多特性，但基础的功能都已经有了。你可以看到如何读写文件、传播和显示错误、使用彩色输出来提高用户的终端体验。

后续的章节会介绍更多高级的应用开发的技术：从数据集合和使用迭代器的函数式编程到为了实现极致并发效率的异步编程。但首先，你需要在下一章中为 Rust 的基本数据类型打下坚实的基础。

Chapter 3

基本类型

这个世界上有很多很多不同类型的书，这是一件好事。但也有很多很多不同类型的人，每个人都想读到一些不同的东西。

——Lemony Snicket

在很大程度上，Rust 语言是围绕它的类型来设计的。它对高性能代码的支持源于让开发者选择不同情况下最合适的数据表示，并在简单性和成本之间取得适当的平衡。Rust 的内存和线程安全也依赖于类型系统的健全性，Rust 的灵活性则来自于它的泛型和 trait。

这一章将介绍 Rust 的基本类型。这些源码级别的类型都有对应的成本和性能可预测的机器级的组件。尽管 Rust 并不保证它会完全按照你的要求精确地表示数据，但只有当有更可靠的改进时它才会违背你的要求。

与 JavaScript 或 Python 这种动态类型语言相比，Rust 要求你事先就进行更多规划。你必须自己写出函数的参数和返回值、结构体的字段、以及一些其他结构的类型。然而，Rust 的两个特性使得这些工作比你想象的要简单很多：

- 有了你指明的类型，Rust 的类型推导将会为你推导出剩余的大部分类型。在实践中，通常只有一个类型能够满足给定的变量或表达式。在这种情况下，Rust 允许你留空，或者说省略这个类型。例如，你可以像下面这样写出一个函数里的所有类型：

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

但这样做既杂乱又冗余。在指定了返回值类型之后，很明显 v 必须是 `Vec<i16>` 类型，即一个 16 位有符号整数的 vector，因为没有其他的类型可以满足语义。并且据此可以推

出 vector 的每个元素必须是 i16 类型。这恰好是 Rust 的类型推导适用的场景，所以你可以改为：

```
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价的，Rust 将会生成完全相同的机器代码。类型推导可以回馈一部分动态类型语言的可读性，并且仍能在编译时捕捉到类型错误。

- 函数可以是泛型的：一个函数可以同时处理很多不同类型的值。

在 Python 和 JavaScript 中，所有的函数天然都是泛型的：一个函数可以操作任何类型的值，只要这个类型有函数体中需要的属性和方法。（这种特性通常被称作鸭子类型：如果它像鸭子一样叫，那它就是一只鸭子。）但正是这种灵活性也导致这些语言很难检测出类型错误，在这些语言里测试通常是唯一一种捕捉类型错误的方式。Rust 的泛型函数给予了这门语言某种程度上和动态类型同样的灵活性，并且仍能在编译期捕获所有的类型错误。

除了灵活性之外，泛型函数和非泛型的函数一样高效。例如，为每个整数类型都编写一个 sum 函数与编写一个处理所有整数类型的泛型 sum 函数相比，并没有性能上的优势。我们将在第 11 章种详细讨论泛型函数。

这一章的剩余部分将会自上而下的覆盖 Rust 的类型，从最简单的数字类型例如整数和浮点数到持有很多个值的复合类型：box、tuple、数组和字符串。

这里有一个 Rust 中类型的汇总。[表 3-1](#)显示了 Rust 的原始类型，包括一些来自标准库的基本类型，和一些用户自定义类型的示例。

表 3-1: Rust 中的类型示例

类型	描述	值
i8, i16, i32, i64, i128, u8, u16, u32, u64, u128	指定位数的有符号和无符号整数	42, -5i8, 0x400u16, 0o100i16, 20_922_789_888_000u64, b'*'(u8 字节字面量)

<code>isize, usize</code>	有符号和无符号整数，和机器里的一个指针一样大（32位或64位）	137, -0b0101_0010 isize, 0xffff_fc00 usize
<code>f32, f64</code>	IEEE 浮点数，单精度和双精度	1.61803, 3.14f32, 6.0221e23f64
<code>bool</code>	布尔值	true, false
<code>char</code>	Unicode 字符，32位	'*', '\n', '字', \x7f, '\u{CA0}'
<code>(char, u8, i32)</code>	Tuple: 把类型混合在一起	('%', 0x7f, -1)
<code>()</code>	“单元值”（空 tuple）	()
<code>struct S { x: f32, y: f32 }</code>	命名字段结构体	S { x: 120.0, y: 209.0 }
<code>struct T (i32, char);</code>	元组结构体	T(120, 'X')
<code>struct E;</code>	元组结构体，无字段	E
<code>enum Attend { OnTime, Late(u32) }</code>	枚举，代数数据类型	Attend::Late(5), Attend::OnTime
<code>Box<Attend></code>	Box: 持有一个堆上的值的指针	Box::new(Late(15))
<code>&i32, &mut i32</code>	共享和可变引用：生命周期不能超过所引用对象的无所有权的指针	&s.y, &mut v
<code>String</code>	UTF-8 字符串，动态大小	"ラーメン: ramen" .to_string()
<code>&str</code>	str 的引用：指向 UTF-8 字符串的无所有权的指针	"そば: soba", &s[0..12]
<code>[f64; 4], [u8; 256]</code>	固定长度的数组，所有元素的类型都必须相同	[1.0, 0.0, 0.0, 1.0], [b' '; 256]
<code>Vec<f64></code>	可变长度的 vector，所有元素的类型都必须相同	vec![0.367, 2.718, 7.389]
<code>&[u8], &mut [u8]</code>	切片的引用：指向数组或 vector 的一部分，包含指针和长度	&v[10..20], &mut a[..]

<code>Option<&str></code>	可选值: <code>None</code> (无值) 或 <code>Some(v)</code> (有值, <code>Some("Dr.", None)</code> 值为 <code>v</code>)	
<code>Result<u64, Error></code>	可能会失败的操作的结果: 成功时 <code>Ok(4096)</code> , 是 <code>Ok(v)</code> , 失败时是 <code>Err(e)</code>	<code>Err(Error::last_os_error())</code>
<code>&dyn Any, &mut dyn Read</code>	trait 对象: 指向一个实现了给定方法的任何值	<code>value as &dyn Any, &mut file as &mut dyn Read</code>
<code>fn(&str) -> bool</code> (闭包类型)	函数指针 闭包	<code>str::is_empty</code> <code> a, b { a*a + b*b }</code>

这些类型中的大部分都会在这一章中介绍，除了下面这些：

- 我们将在第 9 章中单独介绍 `struct` 类型。
- 我们将在第 10 章中单独介绍枚举类型。
- 我们将在第 11 章中介绍 trait 对象。
- 我们将在这里介绍 `String` 和 `&str` 的基础，但在第 17 章中介绍更多细节。
- 我们将在第 14 章介绍函数和闭包类型。

3.1 固定位数的数字类型

Rust 类型系统的基础是一组固定宽度的数字类型的集合，这些类型和现代处理器中的硬件类型相匹配。

固定宽度的数字类型可能会溢出或失去精度，但它们适用于大多数的场景，并且比任意精度的整数和精确小数快几千倍。如果你需要那些类型的数字，可以在 `num` crate 找到相应的支持。

Rust 的数字类型的名称遵循通用的模式，即宽度加上类型（表 3-2）。

表 3-2: Rust 的数字类型

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

这里，机器字是运行代码的机器上的一个指针的大小，32位或者64位。

3.1.1 整数类型

Rust 的无符号整数使用全部的范围来表示正数和0（表3-3）。

表 3-3: Rust 无符号整数类型

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 1 万 8 千亿)
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 的有符号整数使用两种互补的表示方法，使用和无符号类型相对应的位模式来表示一个包含正数和负数的范围（表3-4）。

表 3-4: Rust 的有符号整数类型

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 $2^{31} - 1$, 或者 -2^{63} 到 $2^{63} - 1$

Rust 使用 u8 类型来表示一个字节的值。例如，从二进制文件或者套接字读取数据就会返回 u8 类型的数据流。

与 C 和 C++ 不同，Rust 区分了字符和数字类型：char 不是 u8，也不是 u32（尽管它是 32 位）。我们将会在字符这一节介绍 Rust 的 char 类型。

usize 和 isize 类似于 C 和 C++ 中的 size_t 和 ptrdiff_t 类型。它们的位数和目标机器上地址空间的位数相同：在 32 位架构上就是 32 位，在 64 位架构上就是 64 位。Rust 要求数组索引为 usize 类型的值。数组或 vector 或其他任何含有多个元素的数据结构的长度都是 usize 类型。

Rust 中的整数字面量可以有一个后缀来指示类型：42u8 是一个 u8 类型的值，1729isize 是一个 isize 类型的值。如果一个整数字面量没有类型后缀，Rust 将会延迟决定它的类型，直

到可以从它的使用中推断出它的类型：存储到一个已知类型的变量中、作为参数传递给一个参数类型已知的函数、和一个已知类型的值比较、以及类似的情况。如果到最后还是有很多类型可以满足，此时如果 `i32` 是其中一种可能，Rust 将推断它为 `i32` 类型。否则，Rust 会报歧义错误。

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

为了让长数字更可读，你可以在数字中间插入下划线。例如，你可以把最大的 `u32` 值写作 `4_294_967_295`。下划线放置的位置并不重要，所以你可以每四位插入一个下划线来把十六进制和二进制数字分组，例如 `0xffff_ffff` 或者在最后插入下划线分隔类型后缀，例如 `127_u8`。[表 3-5](#) 给出了一些整数字面量的例子。

[表 3-5: 整数字面量的例子](#)

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，Rust 确实提供了字节字面量：很像字符字面量的 `u8` 值：`b'X'` 代表字符 X 的 ASCII 码值，但是是 `u8` 类型的值。例如，因为 A 的 ASCII 码值是 65，字面量 `b'A'` 和 `65u8` 是等价的。只有 ASCII 字符可以出现在字节字面量中。

这里有一些不能用单个字符表示的字符，因为它们要么会导致歧义要么很难看出来。[表 3-6](#) 中的字符只能用反斜杠转移的方式写出来。

[表 3-6: 需要转义的字符](#)

字符	字节字面量	等价的数字值
单引号, '	<code>b'\''</code>	<code>39u8</code>
反斜杠, \	<code>b'\\'</code>	<code>92u8</code>
换行	<code>b'\n'</code>	<code>10u8</code>
回车	<code>b'\r'</code>	<code>13u8</code>
制表符	<code>b'\t'</code>	<code>9u8</code>

对于那些难以写出或看出的字符，你可以用它们的十六进制码代替。一个字节字面量的形式是 `b'\xHH'`，其中 HH 是两个十六进制的数字，代表值是 HH 的字节。例如，你可以将 ASCII 的“escape”字符的字节字面量写作 `b'\x1b'`，因为“escape”的 ASCII 码是 27，也就是 16 进制的 1B。因为字节字面量只是 `u8` 类型值的另一种表示方式，考虑使用数字字面量可能可读性

会更强：只有当你想表示 ASCII 码时 `b'\x1b'` 才会比 27 更有意义。

你可以将一种整数类型转换为另一种整数类型。我们将会在[类型转换](#)这一节中介绍转换的原理，这里有一些例子：

```
assert_eq!( 10_i8 as u16,    10_u16); // 在类型的表示范围内
assert_eq!( 2525_u16 as i16, 2525_i16); // 在类型的表示范围内

assert_eq!( -1_i16 as i32,   -1_i32); // 符号扩展
assert_eq!(65535_u16 as i32, 65535_i32); // 0 扩展

// 转换一个超出目标类型范围的值
// 等价于原值对  $2^N$  取模
// N 是目标类型的位数
// 这有时也被称为“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);
```

标准库提供一些整数的方法来进行操作。例如：

```
assert_eq!(2_u16.pow(4), 16);           // 求指数幂
assert_eq!((-4_i32).abs(), 4);         // 求绝对值
assert_eq!(0b101101_u8.count_ones(), 4); // 位计数
```

你可以在在线文档中找到这些。但是注意，文档中 `i32`（原始类型）和模块导入的类型（搜索 `std::i32`）有不同的单独页面。

在实际编码时，你不需要像我们在这里一样写出类型后缀，因为上下文会自动推断出类型。当推断不出来时，错误信息可能会让你很惊讶。例如，下面的代码不能编译：

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

Rust 报错：

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

这可能有点迷惑：所有的整数类型都有 `abs` 方法，所以问题在哪呢？出于技术原因，Rust 需要先知道某个值的类型，然后才能调用它的方法。只有当所有的方法调用都被解析之后仍然存在歧义才会使用默认的 `i32` 类型，而在这里，在解析 `abs` 方法时就需要知道 `-4` 的类型，默认推导为 `i32` 的规则在此时不能生效。解决方法是指明类型，要么加上类型后缀，要么使用类型特定的函数：

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

注意函数调用的优先级高于一元前缀运算符，所以当对负数调用方法时一定要小心。如果这个地方第一个表达式里`-4_i32`两侧没有括号，`-4_i32.abs()`将会对4调用`abs`方法，返回正数4，然后求负数返回-4。

3.1.2 Checked、Wrapping、Saturating、Overflowing 算术

当整数运算溢出时，如果是在 debug 模式下 Rust 会 panic。在 release 模式下，运算结果会回环 (*wraps around*)：它会返回正确的值对结果类型能表示的范围取余之后的结果。（这两种情况下，溢出都不像在 C 和 C++ 中一样是未定义行为）。

例如，下面的代码在 debug 模式下会 panic：

```
let mut i = 1;
loop {
    i *= 10;      // panic: 尝试乘到溢出
    // (但只有在 debug 模式会 panic!)
}
```

在 release 模式下，溢出时乘法会回环成负数，然后循环会无限执行。

如果默认行为不是你希望的结果，整数类型提供了一个方法让你指定想要做什么。例如，下面的代码在任何构建模式下都会 panic：

```
let mut i: i32 = 1;
loop {
    // panic: 乘法溢出 (在任何构建模式下)
    i = i.checked_mul(10).expect("multiplication overflowed");
}
```

这些整数运算的方法可以被分为四个通用的类别：

- *Checked* 操作返回一个结果的 Option 值：如果运算结果可以被结果类型正确表示就返回 `Some(v)`，否则返回 `None`。例如：

```
// 10 和 20 的结果可以用 u8 表示。
assert_eq!(10_u8.checked_add(20), Some(30));

// 不幸的是，100 和 200 的和不能用 u8 表示。
assert_eq!(100_u8.checked_add(200), None);

// 求和，如果溢出就 panic。
let sum = x.checked_add(y).unwrap();
```

```
// 奇怪的是，在一种特定情况下，有符号除法也可能导致溢出。
// 一个有符号整数能表示 $-2^{n-1}$ ，但不能表示 $2^{n-1}$ 。
assert_eq!((-128_i8).checked_div(-1), None);
```

- *Wrapping* 操作返回正确的值对结果类型能表示的范围的余数：

```
// 第一个积可以用 u16 来表示。
// 第二个不能，因此我们得到 250000 对 2^16 取模。
assert_eq!(100_u16.wrapping_mul(200), 20000);
assert_eq!(500_u16.wrapping_mul(500), 53392);

// 有符号数的操作可能会回环成负数。
assert_eq!(500_i16.wrapping_mul(500), -12144);

// 在移位操作中，移动的位数会回环到该类型的位数之内
// 因此对 16 位的数字移动 17 位等于移动 1 位
assert_eq!(5_i16.wrapping_shl(17), 10);
```

正如解释的那样，这就是 release 模式下算术操作的行为。使用这种写法的好处是在所有的构建模式下代码的行为都一致。

- *Saturating* 操作会返回最接近正确结果的表示。换句话说，结果被“截断”到这个类型能表示的最大或最小值：

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

没有 saturating 的除法、取余、位移操作。

- *Overflowing* 操作返回一个 tuple (`result`, `overflowed`)，其中 `result` 是回环版本的方法返回的结果，而 `overflowed` 是一个指示是否发生溢出的 `bool` 值：

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

`overflowing_shl` 和 `overflowing_shr` 稍微有些偏离这个模式：只有当移位距离恰好等于类型的位宽度时 `overflowed` 才为 `true`。实际的移位距离等于要求的距离对位宽度取余后的结果：

```
// 对 `u16` 来说移动 17 位太多了，17 对 16 取余等于 1。
assert_eq!(5_u16.overflowing_shl(17), (10, true));
```

表 3-7 中列出了以 `checked_`, `wrapping_`, `saturating_`, `overflowing_` 为前缀的方法。

表 3-7: 操作的名称

操作	名称后缀	示例
加法	add	<code>100_i8.checked_add(27) == Some(127)</code>
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>
取余	rem	<code>(-32768_i16).wrapping_rem(-1) == 0</code>
负数	neg	<code>(-128_i8).checked_neg() == None</code>
绝对值	abs	<code>(-32768_i16).wrapping_abs() == -32768</code>
指数	pow	<code>3_u8.checked_pow(4) == Some(81)</code>
左移	shl	<code>10_u32.wrapping_shl(34) == 40</code>
右移	shr	<code>40_u64.wrapping_shr(66) == 10</code>

3.1.3 浮点数

Rust 提供 IEEE 的单精度和双精度浮点数。这两个类型还包括正无穷、负无穷、正 0、负 0 和非数值。(表 3-8)

表 3-8: IEEE 单精度和双精度浮点数类型

类型	精度	范围
f32	IEEE 单精度浮点数 (至少 6 位十进制数字)	大约 -3.4×10^{38} 到 $+3.4 \times 10^{38}$
f64	IEEE 双精度浮点数 (至少 15 位十进制数字)	大约 -1.8×10^{308} 到 $+1.8 \times 10^{308}$

Rust 的 f32 和 f64 分别对应 C 和 C++ (如果实现支持 IEEE 浮点数的话) 以及 Java (总是使用 IEEE 浮点数) 里的 float 和 double 类型。

浮点数字面量的一般形式如图 3-1。

整数部分之后的部分都是可选的，但小数部分、指数、或者类型后缀至少需要有一个，才能和整数字面量区分开。小数部分可以只有一个单独的小数点，因此 5. 是一个有效的浮点数。

如果一个浮点数字面量缺少类型后缀，和处理整数一样，Rust 会检查上下文来查看这个值是如何使用的。如果最后它发现两种浮点数类型都可以满足语义，那么它会默认选择 f64。

为了实现类型推导，Rust 把整数字面量和浮点数字面量区分为不同的种类：它从来不会把一个浮点数类型推断为整数类型，反之亦然。表 3-9 展示了一些浮点数字面量的例子。

f32 和 f64 类型还关联了 IEEE 要求的特殊常量值例如 INFINITY、NEG_INFINITY (负无穷)、NAN (非数值)、MIN 和 MAX (最小和最大的有限值)：

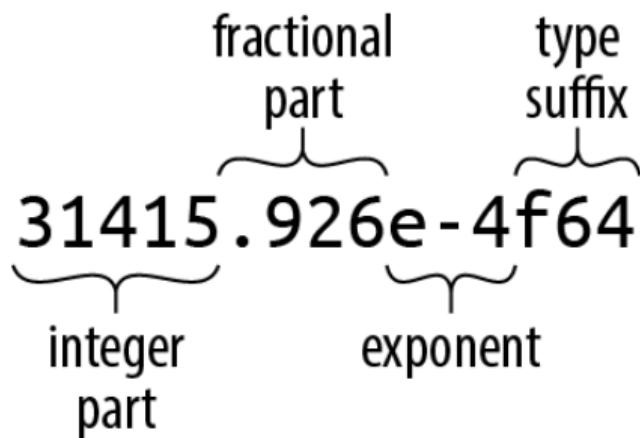


图 3-1: 浮点数字面量

表 3-9: 浮点数字面量的示例

字面量	类型	数值
-1.5625	自动推断	$-1\frac{9}{16}$
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	大约是 $9.10938356 \times 10^{-31}$

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

f32 和 f64 类型提供了完整的数值计算的方法；例如，`2f64.sqrt()` 是 2 的平方根。还有一些示例：

```
assert_eq!(5f32.sqrt() * 5f32.sqrt(), 5.); // 精确的 5.0
assert_eq!((-1.01f64).floor(), -2.0);
```

再重复一次，方法调用的优先级高于前缀运算符，因此对负数调用方法时确保要用括号括起来。

`std::f32::consts` 和 `std::f64::consts` 模块提供了常用的数学常数，例如 E、PI、2 的平方根。

如果要查找浮点数的文档，记得文档里既有类型的文档，名称叫“f32(primitive type)”和“f64(primitive type)”，又有模块的文档，名称叫 `std::f32` 和 `std::f64`。

和整数一样，在实际编码时通常你不需要写出浮点数字面量的类型后缀，但如果你要写，那么只需要指明变量和函数其中一个类型即可：

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

和 C 和 C++ 不同，Rust 中几乎没有隐式类型转换。如果一个函数接收 `f64` 类型的参数，传递 `i32` 的值作为参数将是一个错误。事实上，Rust 甚至不允许从 `i16` 到 `i32` 这样的隐式转换，即使每一个 `i16` 值也都是一个合法的 `i32` 值。但你总是可以使用 `as` 运算符来进行显式转换：`i as f64`，或者 `x as i32`。

缺少隐式类型转换导致 Rust 的表达式可能会比 C 和 C++ 中类似的表达式更加冗长。然而，隐式整数转换经常导致 bug 和安全漏洞。尤其是用来表示内存中某个东西的长度的整数，可能会导致意外的溢出。在我们的实践中，在 Rust 中显式写出类型转换可以提醒我们可能忽略的问题。

我们会在 [类型转换](#) 一节中介绍转换的原理。

3.2 布尔类型

Rust 的布尔类型 `bool` 只有两个值：`true` 和 `false`。比较运算符例如 `==` 和 `<` 会产生 `bool` 类型的结果：`2 < 5` 的结果是 `true`。

许多语言都很宽容，允许在需要布尔值的上下文中使用其他类型：C 和 C++ 隐式把字符、整数、浮点数和指针转换为布尔值，因此它们可以直接用作 `if` 或 `while` 语句的条件。Python 还允许 `string`、`list`、`字典`、甚至 `集合` 用作布尔值，如果不为空时视为 `true`。然而 Rust 非

常严格：像 `if` 和 `while` 这样的控制流的条件必须是 `bool` 表达式，短路求值运算符 `&&` 和 `||` 也是这样。你必须写 `if x != 0 { ... }`，而不能写 `if x { ... }`。

Rust 的 `as` 运算符可以把 `bool` 值转换为整数值：

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

然而，`as` 不能反过来把整数值转换为 `bool` 值。你必须显式写出比较运算例如 `x != 0`。

尽管 `bool` 类型只需要单个比特来表示，Rust 还是使用整个字节来表示 `bool`，因此你可以创建指向它的指针。

3.3 字符

Rust 的字符类型 `char` 代表一个单独的 Unicode 字符，是一个 32 位的值。

Rust 使用 `char` 表示单个字符，但使用 UTF-8 编码字符串和文本流。因此，`String` 表示它的文本是一个 UTF-8 字节序列，而不是字符的数组。

字符字面量是被单括号包围的单个字符，例如 '`'8'` 或 '`'!'`'。你可以使用 Unicode 范围内的任何字符：`'𠮷'` 是一个 `char` 字面量代表日语汉字中的 *sabi* (rust)¹。

和字节字面量一样，一些字符需要反斜杠转义（表 3-10）。

表 3-10: 需要反斜杠转义的字符

字符	Rust 字符字面量
单引号，'	'\''
反斜杠，\	'\\'
换行	'\n'
回车	'\r'
制表符	'\t'

如果你喜欢的话，你可以以十六进制的方式写出一个字符的 Unicode 编码：

- 如果字符的码点在 U+0000 到 U+007F 之间（可以据此判断是否在 ASCII 字符集之中），那么你可以将字符写作 '`\xHH`'，`HH` 是一个两位的十六进制数字。例如，字符字面量 '`*`' 和 '`\x2A`' 是等价的，因为字符 * 的码点是 42，十六进制是 2A。
- 你可以用 '`\uHHHHHHH`' 形式写出任何 Unicode 字符，`HHHHHHH` 是一个最长 6 位的十六进制数字，可以用下划线分隔。²

¹译者注：看起来这个字符可能无法正常显示，可以在源码里看到这个字符原本的样子。

²译者注：此处原文中给了一个例子，但译者不知道该怎么打出卡纳达语里的字符，复制粘贴也不行，就省略了这个例子。

一个 `char` 总是存储一个在 0x0000 到 0xD7FF 或 0xE000 到 0x10FFFF 之间的 Unicode 码点。一个 `char` 绝不会在两个范围之间（即 0xD800 到 0xDFFF），也不会超出 Unicode 的编码空间（即大于 0x10FFFF）。Rust 使用类型系统和动态检查来确保 `char` 值总是在允许的范围内。

Rust 永远不会进行 `char` 和其他任何类型之间的隐式转换。你可以使用 `as` 转换运算符来把 `char` 转换为整数类型，对于小于 32 位的类型，字符值的高位会被截断：³

```
assert_eq!('*' as i32, 42);
```

另外，`u8` 是唯一可以用 `as` 运算符转换成 `char` 的整数类型：Rust 只会对开销很低并且不可能失败的转换使用 `as` 运算符，但任何 `u8` 之外的整数类型都包含不是有效的 Unicode 码点的值，因此这些转换需要运行时检查。所以，标准库提供了函数 `std::char::from_u32` 接受任何 `u32` 值，并返回 `Option<char>`：如果 `u32` 的值不是合法的 Unicode 码点，`from_u32` 会返回 `None`；否则，它会返回 `Some(c)`，`c` 就是作为转换结果的 `char`。

标准库为字符类型提供了一些有用的方法，你可以在文档中的“`char(primitive type)`”和模块“`std::char`”的页面中查找这些方法。例如：

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

当然，单个字符显然没有字符串和文本流有趣。我们将会在[字符串类型](#)中介绍 Rust 的标准 `String` 类型和常用的文本处理操作。

3.4 元组

元组是两个、或三个、四个、五个、……不同类型的值的组合。你可以将元组看作被逗号分隔和括号包围的元素序列。例如，`("Brazil", 1985)` 是一个元组，它的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`，你可以通过 `t.0`、`t.1`、……来访问它的元素。

某种程度上，元组类似于数组：这两个类型都代表一系列有固定顺序的值。许多编程语言合并或结合了这两种概念，但在 Rust 中，它们是完全独立的。一方面，元组的每个元素可以拥有不同的类型，而数组的所有元素必须有相同的类型。另外，元组只允许常数索引，例如 `t.4`。你不可能写 `t.i` 或者 `t[i]` 来获取第 `i` 个元素。

Rust 代码中经常使用元组类型来返回多个值。例如，字符串切片中的 `split_at` 方法（用于将一个字符串切分为两半并返回）被声明为类似如下形式：

³译者注：这个例子中也省略了卡纳达语字符相关的内容。

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

返回类型 (`&str`, `&str`) 是两个字符串切片组成的元组。你可以使用模式匹配语法来把返回的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");
```

这比下边的等价代码可读性更强：

```
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");
```

你也可以将元组视为一种极简的结构体类型。例如，在[第2章](#)的曼德勃罗集程序中，我们需要向函数传递要绘制的图片的宽度和高度。我们可以声明一个有 `width` 和 `height` 成员的结构体，但这么简单的事没有必要搞得这么复杂，因此我们用了一个元组：

```
/// 写入缓冲区`pixels`，它的大小由`bounds`给出，
/// 写入的文件名是`filename`。
fn write_image(filename: &str, pixels: &[u8], bounds: (usize, usize))
    -> Result<(), std::io::Error>
{ ... }
```

参数 `bounds` 的类型是 `(usize, usize)`，一个有两个 `usize` 值的元组。诚然，我们可以直接使用单独的 `width` 和 `height` 参数，生成的机器代码也会完全相同。这么写的目的只是想表明，我们把图片的大小看成一个值，而不是两个值，使用元组的写法可以清晰的表现出这一点。

元组的另一个常见用法是 0 元组 ()。这通常被称为单元类型因为它只有一个取值，也写作 ()。Rust 会在上下文要求某种类型，但没有有意义的值要传递的情况下使用单元类型。

例如，一个不返回值的函数的返回类型是 ()。标准库中的 `std::mem::swap` 函数没有有意义的返回值；它只是交换两个参数的值。`std::mem::swap` 的声明如下：

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

`<T>` 意味着 `swap` 是泛型的：你可以将它用于任何类型 `T` 的引用。但签名中省略了 `swap` 的返回类型，这实际上是返回单元类型的缩写：

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

与此类似，我们之前提到的 `write_image` 中的返回值类型是 `Result<(), std::io::Error>`，这意味着如果出错时函数返回 `std::io::Error` 类型的值，如果成功时返回无值。

如果你想的话，你可以在元组的最后一个元素之后加上一个逗号：类型 `(&str, i32,)` 和 `(&str, i32)` 是等价的，`("Brazil", 1985,)` 和 `("Brazil", 1985)` 也是。Rust 允许任何逗号分隔的值列表最后加上一个额外的逗号：函数参数、数组、结构体和枚举定义，等等。这对人类来说可能看起来很奇怪，但当需要在最后添加或删除条目时会变得方便一些。

为了一致性，还有只包含单个值的元组。字面量 `("lonely hearts",)` 是一个包含单个字符串的元组，它的类型是 `(&str,)`。这里，最后的逗号是必须的，这是为了和单纯的用括号把表达式括起来相区分。

3.5 指针类型

Rust 有几个表示内存地址的类型。

这是 Rust 和其他大多数有垃圾回收语言的不同之处。在 Java 中，如果 `class Rectangle` 包含一个字段 `Vector2D upperLeft`，那么 `upperLeft` 实际上是对另一个单独创建的 `Vector2D` 对象的引用。在 Java 中一个对象从来不会真的包含其他对象。

Rust 里则不同。Rust 被设计为尽量减少内存分配。默认情况下 Rust 里是值嵌套的，值 `((0, 0), (1440, 900))` 被存储为四个相邻的整数。如果你把它赋值给一个局部变量，那么你将得到一个 4 个整数大小的局部变量，不会在堆上分配任何内存。

这有助于提高内存效率，但它会导致当 Rust 程序需要指向其他值的指针时，必须显式使用指针类型。好消息是 safe Rust 里的指针类型有一些约束来保证不会出现未定义行为，因此 Rust 中的指针比 C++ 中的更容易正确使用得多。

我们将会在这里讨论三种指针类型：引用、box、unsafe 指针。

3.5.1 引用

一个 `&String`（读作“ref String”）类型的值是一个 `String` 值的引用，一个 `&i32` 类型的值是一个 `i32` 值的引用，等等。

把引用想象成 Rust 的基本指针类型可以让我们更容易理解。在运行时，一个 `i32` 的引用是一个单独的机器字，里面存储的是指向的 `i32` 值的地址，可能在栈上也可能在堆上。表达式 `&x` 产生一个 `x` 的引用；在 Rust 的术语中，我们称它借用了 `x` 的引用。给定引用 `r`，表达式 `*r` 就是 `r` 指向的值。这些类似于 C 和 C++ 中的 `&` 和 `*` 运算符。和 C 指针类似，当引用离开作用域时并不会自动释放任何资源。

然而和 C 指针不同的是，Rust 的引用永远不为空：在 safe Rust 中没有任何办法产生一个

空引用。而且和 C 指针不同，Rust 通过追踪值的所有权和生命周期，在编译期就可以杜绝悬垂指针、double free 和指针失效的情况。

Rust 的指针有以下两种类型：

`&T`

一个不可变的共享引用。你可以同时拥有同一个值的多个共享的引用，但它们都是只读的：修改它们指向的值是禁止的，就像 C 中的 `const T*` 一样。

`&mut T`

一个可变的、独占的引用。你可以读写它指向的值，类似于 C 中的 `T*`。但只要这个引用存在，你不能再持有任何这个值的其他任何类型的引用。事实上，这时候你唯一可以访问这个值的方法就是通过这个可变引用。

Rust 使用这种方式来区分共享和可变的引用，以此来强制执行“单个写者或多个读者”规则：要么你可以读写值，要么它只能被任何数量的读者共享。这种分隔由编译器检查来强制执行，它是 Rust 安全保证的核心。[第 5 章](#)解释了 Rust 中使用安全引用的规则。

3.5.2 Box

最简单的在堆上分配内存的方式是使用 `Box::new`：

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

`t` 的类型是 `(i32, &str)`，因此 `b` 的类型是 `Box<(i32, &str)>`。调用 `Box::new` 会在堆上分配足够的内存来存储元组。当 `b` 离开作用域时，内存会被立即释放，除非 `b` 被 `move` 了——例如被返回了。`move` 对 Rust 处理堆上分配的值的方式至关重要，我们将在[第 4 章](#)详解介绍这一点。

3.5.3 原始指针

Rust 也有原始指针类型 `*mut T` 和 `*const T`。原始指针类似于 C++ 中的指针。使用原始指针是不安全的，因为 Rust 无法追踪它指向的到底是什么。例如，原始指针可能是空、或者可能指向被释放的内存、或者现在指向一个和之前不同类型的值。所有 C++ 中经典的指针错误都有可能。

然而，你只能在 `unsafe` 块中解引用原始指针。`unsafe` 块是 Rust 中的可选机制，它的安全性取决于你自己。如果你的代码没有 `unsafe` 块（或者有 `unsafe` 块但里面的代码都是完全正确的），那么整本书中强调的安全性保证都适用。细节见[第 22 章](#)。

3.6 数组、vector 和切片

Rust 有三种表示一系列值的类型：

- 类型 `[T; N]` 表示一个有 `N` 个元素的数组，每个元素的类型都是 `T`。每个数组的长度必须在编译期已知，并且是类型的一部分。你不能添加新元素或者缩减元素。
- 类型 `Vec<T>`，是类型 `T` 的向量。它是动态分配、可增长的类型 `T` 的值的序列。`vector` 的元素存储在堆中，所以你可以按需更改 `vector` 的大小：可以添加新的元素、附加其它 `vector`、删除元素等。
- 类型 `&[T]` 和 `&mut [T]` 分别是类型 `T` 的共享切片和可变切片，它们是数组或 `vector` 等其它值中的一部分元素的引用。你可以把切片当做一个指向第一个元素的指针再加上一个可访问的元素数量。一个可变的切片 `&mut [T]` 让你可以读取并修改元素，但不能被共享；共享的切片 `&[T]` 允许你在多个读者间共享数据，但不能修改元素。

如果 `v` 是这三种类型中的任意一种，那么表达式 `v.len()` 返回 `v` 中元素的数量，`v[i]` 返回 `v` 中的第 `i+1` 个元素。第一个元素是 `v[0]`，最后一个元素是 `v[v.len() - 1]`。Rust 会检查 `i` 是否在范围内，如果超过了长度范围，表达式会 panic。`v` 的长度可能是 0，这时任何索引操作都会 panic。`i` 必须是 `usize` 类型的值，你不能使用任何其它整数类型当作索引。

3.6.1 数组

创建一个数组有几种方式。最简单的方式是用方括号括起来的一系列值：

```
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);
```

如果想要初始化定长数组，可以使用 `[V; N]` 的写法，其中 `V` 是每一个元素的初始值，`N` 是长度。例如，`[true; 10000]` 代表含有 10,000 个 `bool` 类型元素的数组，每个元素都被初始化为 `true`：

```
let mut sieve = [true; 1000];
for i in 2..100 {
    if sieve[i] {
        let mut j = i * i;
        while j < 1000 {
            sieve[j] = false;
            j += 1;
        }
    }
}
```

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

你会发现这种语法还可以用于创建固定大小的缓冲区：`[0u8; 1024]`可以用作一个1K的缓冲区，所有字节全部初始化为0。Rust里没有创建未初始化数组的语法。（更广泛的说，Rust保证代码不可能访问任何未初始化的值。）

一个数组的长度是类型的一部分，并且在编译期就已经固定。如果n是一个变量，你不能用`[true; n]`这种语法来得到一个有n个元素的数组。当你需要运行时才能确定长度的数组时，使用vector来代替。

数组有一些有用的方法——迭代元素、搜索、排序、填充、过滤等等——事实上这些方法都是切片的，而不是数组的。但在搜索方法时Rust会隐式的把数组的引用转换为切片，因此你可以直接用数组调用切片的方法：

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

这里，`sort`方法实际上是由切片定义的，但因为它是以调用者的引用为参数，所以Rust隐式的创建了一个引用整个数组的`&mut [i32]`类型的切片，然后在这个切片上调用`sort`方法。事实上，我们之前提到的`len`方法最早的时候也是切片的方法。我们将在[切片](#)这一节中介绍切片。

3.6.2 vector

一个`Vec<T>`是一个长度可变的类型T的数组，它的元素都存储在堆上。

有几种创建vector的方法。最简单的方法是使用`vec!`宏，这种语法很像数组字面量：

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

当然，这是vector，不是数组，因此我们可以动态添加元素：

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

你也可以再次使用类似于数组字面量的语法来创建重复给定值若干次的vector：

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

`vec!` 宏等价于调用 `Vec::new` 创建一个新的空 vector，然后向其中添加元素：

```
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"]);
```

另一种创建 vector 的方法是通过迭代器创建：

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

通常使用 `collect` 时你需要给出类型，因为它可以生成不同类型的集合，不只是 vector。但指明了 `v` 的类型之后，我们需要的集合类型就没有歧义了。

类似于数组，你也可以对 vector 调用切片的方法：

```
// 回文
let mut palindrome = vec!["a man", "a plan", "a canal", "panama"];
palindrome.reverse();
// 合理但令人失望
assert_eq!(palindrome, vec!["panama", "a canal", "a plan", "a man"]);
```

这里，`reverse` 方法实际上是由切片定义的，但这里调用隐式地借用了 vector 的 `&mut [&str]` 切片，然后对它调用了 `reverse`。

`Vec` 是 Rust 中不可或缺的一个类型——它被用在几乎所有需要动态大小列表的场景——所以有很多其他的方法创建构造新的 vector 或扩展现有的列表，我们将在第 16 章介绍它们。

一个 `Vec<T>` 由 3 个值组成：一个指针指向堆上存储元素的缓冲区，这个缓冲区的所有权属于这个 `Vec<T>`；缓冲区可以存储的元素的数量；它现在实际已经拥有的元素的数量（也就是它的长度）。当缓冲区的元素到达最大容量时，继续添加元素会导致 vector 重新分配一个更大的缓冲区，再把已有元素都拷贝过去，然后更新 vector 的指针和容量，最后释放旧的缓冲区。

如果你提前知道 vector 需要存储的元素的数量，你可以使用 `Vec::with_capacity` 来创建一个从一开始就拥有足以存下它们的缓冲区的 vector；然后你可以向 vector 中添加元素，并且不会导致重新分配内存。`vec!` 宏就使用了类似这样的方法，因为它知道最终的 vector 会有

多少个元素。注意这决定了 vector 的初始大小，如果你继续添加元素，vector 会像通常一样变大。

很多库函数都在寻找使用 `Vec::with_capacity` 代替 `Vec::new` 的机会。例如，在 `collect` 的例子中，迭代器 `0..5` 提前知道它会产生 5 个值，`collect` 函数就可以利用这一点提前分配足够容量的 vector。我们将在第 15 章中介绍这些。

就像 `len` 方法会返回持有的元素数量一样，它的 `capacity` 方法会返回它在不重新分配的前提下能存储的最大元素数量：

```
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());
```

最后打印出的容量并不保证是 4，但通常至少是 3，因为 vector 已经持有 3 个元素了。

你可以在 vector 中任意插入或移除元素，尽管这些操作会移动修改位置前面或者后面的元素，因此如果 vector 很长的时候这些操作会很慢：

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

// 在索引为 3 的地方插入 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]);
```

你可以使用 `pop` 方法移除最后一个元素并返回它。更确切地说，从 `Vec<T>` 中弹出只会返回一个 `Option<T>`：如果 vector 已经为空是 `None`，如果最后一个元素是 `v` 就是 `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);
```

你可以使用 `for` 循环来迭代 `vector`:

```
// 把命令行参数收集为 String 的 vector
let languages: Vec<String> = std::env::args().skip(1).collect();
for l in languages {
    println!("{}: {}", l,
        if l.len() % 2 == 0 {
            "functional"
        } else {
            "imperative"
    });
}
```

用一些编程语言的名字来运行这个程序，你就会发现有趣的地方：

```
$ 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，我们得到了术语函数式语言的一个令人满意的定义。⁴

尽管 `Vec` 非常重要，但它只是 Rust 中定义的一个普通类型，不是语言内建的类型。我们将在[第 22 章](#)中介绍实现这样的类型的技术。

3.6.3 切片

一个切片写作 `[T]`，没有长度。它表示数组或 `vector` 的一部分。因为切片可以是任意长度，因此切片不能直接存储在变量中或者作为参数传递。切片必须通过引用传递。

切片的引用是胖指针：一个包含指向切片中第一个元素的指针和切片中元素数量的双字值。

假设你正在运行下列代码：

```
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];
```

⁴译者注：此处作者应该是开玩笑。

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

在最后两行中，Rust 自动把 `&Vec<f64>` 和 `&[f64; 4]` 转换成了切片的引用。

这段代码运行完后，内存布局类似图 3-2 中所示：

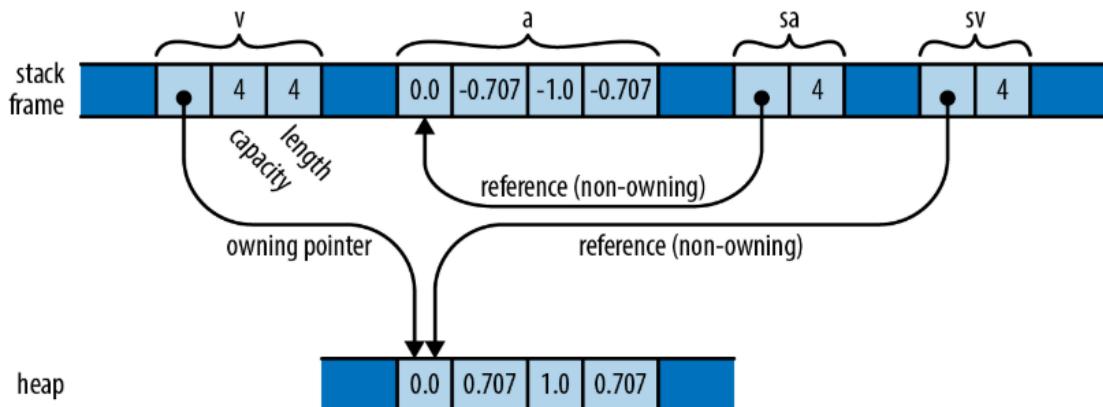


图 3-2: 一个 vector `v` 和一个数组 `a`，以及分别指向它们的切片 `sv` 和 `sa` 的内存布局

一个普通的引用是一个指向单个值的无所有权指针，而一个切片的引用是一个指向内存中连续的范围的指针。这使得如果你想写一个处理数组或 vector 的函数，那么切片引用将是一个很好的选择。例如，这里有一个函数打印出一系列值，每个单独一行：

```
fn print(n: &[f64]) {
    for elt in n {
        println!("{}", elt);
    }
}

print(&a); // 可以用于数组
print(&v); // 可以用于 vector
```

因为这个函数的参数是切片引用，你可以像示例中一样传递数组或者 vector 的引用。事实上，很多你可能以为是数组或 vector 的方法实际上都是切片的方法：例如 `sort` 和 `reverse` 方法，实际上都是切片类型 `[T]` 的方法。

你可以通过索引一个数组或 vector 或一个现有切片的一个范围来得到一个切片引用：

```
print(&v[0..2]); // 打印出 v 的前两个元素
print(&a[2..]); // 打印出从 a[2] 开始剩余的元素
print(&sv[1..3]); // 打印出 v[1] 和 v[2]
```

类似于普通的数组访问，Rust 会检查索引的有效性。尝试借用超过数据结尾的切片会导致 panic。

因为切片类型总是以引用的方式出现，我们也经常用“切片”来指代 `&[T]` 或 `&str` 这样的类型，用最短的名称表达最常用的概念。

3.7 字符串类型

熟悉 C++ 的程序员应该都知道 C++ 里有两种字符串类型：字符串字面量的类型 `const char *`，标准库里还提供了一个类 `std::string`，用于在运行时动态创建字符串。

Rust 的设计与此类似。在这一节中，我们将首先展示字符串字面量的所有写法，然后介绍 Rust 的两种字符串类型。我们将在第 17 章中详细讨论字符串和文本的处理。

3.7.1 字符串字面量

字符串字面量被双引号包裹，和 `char` 字面量一样是通过反斜杠来转义：

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

在字符串字面量里和 `char` 字面量里不同的一点是，单引号不需要使用反斜杠转义，但双引号需要。

一个字符串可以占多行：

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

这个字符串里的换行符被包括在字符串里，因此也会被输出。第二行开头的空格也是。

如果字符串中的一行以反斜杠结尾，那么换行符和下一行的前导空格就会被丢弃掉：

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

这段代码会打印出单行文本。打印出的字符串里“and”和“there”之间有一个空格，因为第一行的反斜杠前边有一个空格，而连字符和“more”之间没有空格。

在少数情况下，在字符串里用反斜杠转义反斜杠是一件很烦的事情。（典型的例子就是正则表达式和 Windows 的路径）对于这种情况，Rust 提供了原始字符串，原始字符串用小写字母 `r` 来标记。原始字符串中的所有反斜杠和空白字符都被原样包含在字符串里，不会识别任何转义：

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

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

你不能通过在前面加上反斜杠来把双引号包含在原始字符串中——记住，我们说的是不会识别任何转义。然而，这种情况也有解决的方法。原始字符串的开头和结尾可以用任意相同数量的井号来标记：

```
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 ('###').  
"###);
```

您可以根据需要添加井号，以明确原始字符串的结束位置。

3.7.2 字节字符串

以**b**前缀开头的字符串字面量是字节字符串。这种字符串实际上是u8值的切片——也就是字节流——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]`：它是一个有三个字节的数组的引用。它没有任何我们之后要讨论的字符串的方法。它和字符串最像的地方就是我们创建它时用的语法。

字节字符串可以使用我们之前使用过的所有语法：它也可以跨多行、使用反斜杠转义、使用反斜杠连接多行。还有以 `br"` 开头的原始字节字符串。

字节字符串不能包含任意 Unicode 字符，它只能包含 ASCII 和用 `\xHH` 形式表示的字符。

3.7.3 内存中的字符串

Rust 的字符串是 Unicode 字符的序列，但它并不是作为 `char` 的数组存储在内存中。事实上它使用 UTF-8 编码存储，这是一种可变长度的编码。每一个 ASCII 字符被存储为一个字节，其他字符可能会占据多个字节。

图 3-3 显示了下面代码创建的 `String` 和 `&str`: ⁵

```
let noodles = "noodles".to_string();  
let oodles = &noodles[1..];  
let poodles = "某些卡纳达语字符";
```

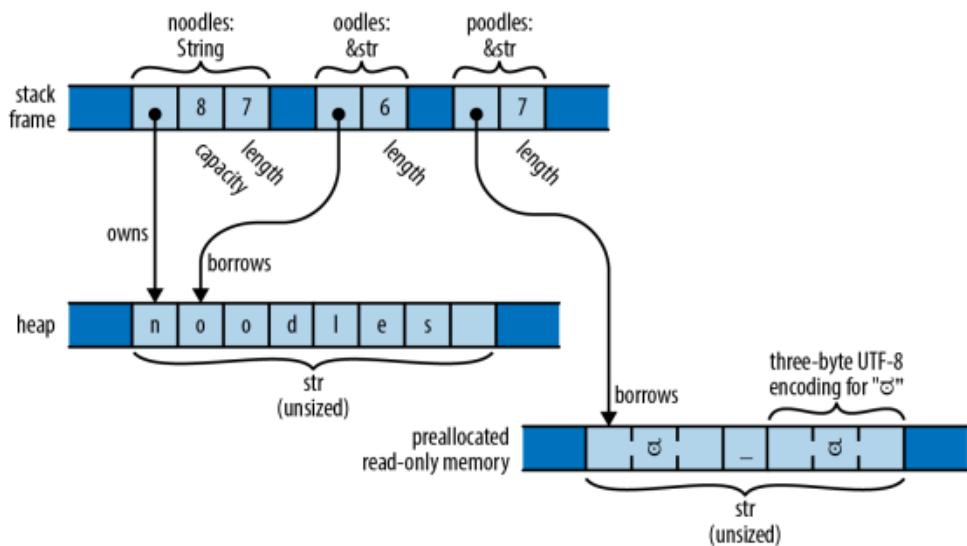


图 3-3: String, &str, str

一个 `String` 有一个大小可变的缓冲区来存储 UTF-8 文本。缓冲区是在堆上分配的，所以它可以按需或按照要求调整缓冲区大小。在这个例子中，`noodles` 是一个拥有 8 个字节大小的缓冲区的 `String`，其中 7 个字节已经被使用。你可以将 `String` 想象为保证内容是有效 UTF-8 编码的 `Vec<u8>`；事实上，`String` 就是这么实现的。

一个 `&str`（读作“stir”或者“字符串切片”）是一个指向其他值拥有的 UTF-8 文本的引用：它“借用”了这段文本。在这个例子中，`oodles` 是一个指向 `noodles` 所持有的文本中的最后六个字节的 `&str`，因此它表示文本“oodles”。就像其他切片引用一样，`&str` 是一种胖指针，包括实际数据的地址和长度。你可以将 `&str` 看作一个保证内容为合法的 UTF-8 编码的 `&[u8]`。

字符串字面量是一个指向预先分配好内存的文本的 `&str`，实际的文本通常和程序的机器码一起存储到只读的内存区域。在之前的例子中，`poodles` 是一个字符串字面量，指向程序运行时就已经被创建好并且持续到程序退出的 7 个字节。

一个 `String` 或 `&str` 的 `len()` 方法返回它的长度。但这个长度是以字节为单位，而不是以字符为单位：

```
assert_eq!("某些卡纳达语字符".len(), 24);
assert_eq!("某些卡纳达语字符".chars().count(), 8);
```

不能修改 `&str`:

```
let mut s = "hello";
s[0] = 'c';      // error: `&str` 不能被修改，以及其他原因
s.push('\n');    // error: `&str` 没有`push`方法
```

⁵译者注：因为不知道怎么打出来这些卡纳达语字符，所以用“某些卡纳达语字符”代替。

要在运行时创建新的字符串，要使用 `String`。

类型 `&mut str` 确实存在，但它并不好用，因为几乎所有对 UTF-8 的操作都可能改变字节长度，而一个切片不能重新分配它指向的参照物。事实上，`&mut str` 唯一能做的操作是 `make_ascii_uppercase` 和 `make_ascii_lowercase`，这两个操作在原址修改文本，并且只影响单个字节的字符。

3.7.4 String

`&str` 和 `&[T]` 很像：都是一个指向某些数据的胖指针。`String` 类似于 `Vec<T>`，如表 3-11 所示。

表 3-11: `Vec<T>` 和 `String` 的比较

	<code>Vec<T></code>	<code>String</code>
自动释放缓冲区	是	是
可增长	是	是
有 <code>::new()</code> 和 <code>::with_capacity()</code> 类型关联函数	是	是
<code>.reserve()</code> 和 <code>.capacity()</code> 方法	是	是
<code>.push()</code> 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>
继承方法	从 <code>&[T]</code>	从 <code>&str</code>

类似于 `Vec`，每个 `String` 都有它自己的在堆上分配的缓冲区，这个缓冲区不和其他任何 `String` 共享。当一个 `String` 变量离开作用域时，缓冲区会自动释放，除非 `String` 被 move 了。

有几种方式创建 `String`:

- `.to_string()` 方法把 `&str` 转换为一个 `String`，这会拷贝字符串：

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

`.to_owned()` 方法做同样的事，你可能会看到它用同样的方式使用。它也可以用于其他类型，正如我们将在第 13 章讨论的一样。

- `format!()` 宏类似于 `println!()`，区别在于它返回一个新的 `String` 而不是把它打印到标准输出，而且它不会在最后自动加上换行符：

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

- 字符串的数组、切片、vector 都有两个方法 `.concat()` 和 `.join(sep)`，把多个字符串组合成一个：

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

有时你需要选择使用 `&str` 还是 `String`。第 5 章将详细介绍这个问题。现在你只需要知道 `&str` 可以指向任何字符串的任何切片，不管是字符串字面量（存储在可执行文件中）还是 `String`（在运行时分配和释放）。这意味着当允许调用者传递任何类型的字符串时 `&str` 更适合用作参数的类型。

3.7.5 使用字符串

字符串支持 `==` 和 `!=` 运算符。只有当两个字符串含有内容和顺序都完全相同的字符时两个字符串才相等（和它们在内存中的位置无关）：

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

字符串还支持比较运算符 `<`, `<=`, `>`, `>=`, 以及很多你可以在在线文档的 “`str(primitive type)`” 或 “`std::str`” 模块的页面中找到的有用的方法和函数（或者直接跳转到第 17 章）。这里有一些例子：

```
assert!("peanut".contains("nut"));
assert_eq!("    clean\n".trim(), "clean");

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

注意，因为 Unicode 的特性，简单的逐字符比较并不总是能给出预期的答案。例如，字符串 "`th\ufe9f`" 和 "`the\ufe0f`" 都是有效的 thé（法语中表示茶的单词）的 Unicode 表示。Unicode 认为它们应该以相同的方式展示和处理，但是 Rust 把它们当作两个完全不同的字符串。类似的，Rust 的比较运算符例如 `<` 只是简单的使用基于字符码点值的字典顺序。这种顺序只是偶尔和用户语言和文化中用于文本的排序一致。我们将在第 17 章中详细讨论这个问题。

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 字符串。有时一个程序可能会需要处理非有效的 Unicode 字符串。这通常发生在 Rust 程序与其他不强制遵循 Unicode 的系统交互时。例如，在大多数操作系统中很容易就能创建一个文件名不是有效的 Unicode 的文件。当 Rust 遇到这样的文件名时该如何处理？

Rust 的方法是为这些场景提供一些类似字符串的类型：

- 对于 Unicode 文本坚持使用 `String` 和 `&str`。
- 当处理文件名时，使用 `std::path::PathBuf` 和 `&Path` 来代替。
- 当处理完全不是 UTF-8 编码的二进制数据时，使用 `Vec<u8>` 和 `&[u8]`。
- 当处理环境变量名称或者命令行参数这些由操作系统提供的内容时，使用 `OsString` 和 `&OsStr`。
- 当和以空字符结尾的 C 库交互时，使用 `std::ffi::CString` 和 `&CStr`。

3.8 类型别名

`type` 关键字可以像 C++ 中的 `typedef` 一样为已存在的类型声明一个新名字：

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

我们在这里声明的 `Bytes` 就是这种特定类型的 `Vec`：

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

3.9 基本类型之外

类型是 Rust 的一个核心部分。我们将在整本书中继续讨论类型并介绍新的类型。特别是，Rust 的用户自定义类型赋予了语言很多自身的风格。有三种用户自定义的类型，我们将分别在三章中介绍它们：[第 9 章](#)介绍结构体，[第 10 章](#)介绍枚举，[第 11 章](#)介绍 trait。

函数和闭包有它们自己的类型，这将在[第 14 章](#)中介绍。标准库中的类型将在整本书中介绍。例如，[第 16 章](#)介绍标准集合类型。

不过，这些都还需要等一会。在我们继续之前，是时候了解一下 Rust 安全规则的核心概念了。

Chapter 4

所有权与 move

在内存管理方面，我们希望编程语言能够具备以下两个特点：

- 我们希望内存能在我们想要释放的时候被及时释放。这样我们可以控制程序的内存消耗。
- 我们永远不希望使用一个指向已经被释放的对象的指针。这会导致未定义行为，进而导致崩溃和安全漏洞。

但这两点看起来似乎是相互矛盾的：释放一个还有指针指向的对象的内存必定会导致悬垂指针。几乎所有的主流编程语言都属于两个阵营之一，取决于它们放弃了哪一点：

- “安全优先”的阵营使用垃圾回收来管理内存，自动释放那些没有指针指向的对象。这种做法通过将对象一直保持到没有指针指向来避免悬垂指针。几乎所有的现代语言都落入了这个阵营，包括 Python、JavaScript、Ruby、Java、C#、Haskell。

但依赖垃圾回收意味着放弃控制对象被回收的精确时间。垃圾收集器通常都令人讨厌，并且理解为什么内存没有如你所料的被释放可能会是一个挑战。

- “控制优先”的阵营让你自己负责释放内存。程序的内存消耗完全由你控制，但如何避免悬垂指针成了你最大的问题。C 和 C++ 是这个阵营里仅有的主流语言。

如果你从没犯过错，那说明你很厉害。但证据表明，你最终还是会犯错。指针的错误使用一直都是那些被报导的安全问题的罪魁祸首。

Rust 旨在同时保证安全和性能，因此这两种阵营都是不可接受的。但如果兼顾两者很简单的话，早就有人做出来了。要想兼顾两者，必须从根本上作出改变。

Rust 以一种令人惊讶的方式打破了这个死锁：严格限制程序使用指针的方法。这一章和接下来将专注于解释这些限制和为什么它们能解决问题。你常用的一些程序结构可能也不符合这些规则，你可能需要寻找替代方案。但这些限制的最终效果是给这种混乱带来了足够的秩序，以允许 Rust 在编译期检查你的程序是否能避免内存安全错误：悬垂指针、两次释放、使用未初始化的内存等。在运行时，你的指针只是简单的地址，就像在 C 和 C++ 中一样。不同

的是你的代码已经被证明是安全的。

这些规则也为 Rust 实现安全的并发编程奠定了基础。Rust 精心设计的线程原语可以让这些保证内存安全的规则也能保证你的代码可以避免数据竞争。Rust 程序中的一个 bug 不可能导致一个线程破坏另一个线程的数据进而导致在不相干的地方出现很难复现的错误。多线程代码中的不确定行为被那些专为它设计的特性——互斥锁、消息通道、原子类型等完全隔离，不会出现在正常的内存访问中。C 和 C++ 中的多线程代码臭名昭著，但 Rust 漂亮地解决了它。

即使有这些限制，你会发现它仍然可以足够灵活地处理几乎所有的任务，而它可以消除内存管理和并发 bug 的优势将证明你需要改变——你需要对自己的风格进行调整。这是 Rust 最大的赌注，也是它的核心和成功之处。本书的作者们看好 Rust，正是因为我们在 C 和 C++ 方面有丰富的经验。对我们来说，遵守 Rust 的规则不费吹灰之力。¹

Rust 的规则可能和你在其他编程语言中看到的不同。了解怎么和它们一起工作并利用它们的优势，在我们看来是学习 Rust 的核心挑战。在这一章中，我们将首先展示相同的潜在问题如何在其他语言中导致问题，以此来深入了解 Rust 规则背后的逻辑和意图。之后，我们将详细解释 Rust 的规则、从概念和机制层面探究所有权的含义、如何在各种场景下追踪所有权的变化、以及一些为了提供更大的灵活性而打破这些规则的类型。

4.1 所有权

如果你读过很多 C 或 C++ 的代码，你可能会看到过有注释说某个类的实例拥有 (*own*) 某些它指向的其他对象。这一般意味着拥有者可以决定何时释放它拥有的对象：当拥有者被销毁时，它会销毁所有它拥有的对象。

例如，假设你写了如下 C++ 代码：

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

字符串 `s` 在内存中的表示通常如图 4-1 所示：

这里，实际上 `std::string` 对象本身总是只有 3 个字长，包括一个指向堆上分配的缓冲区的指针、缓冲区的最大容量（也就是在不重新分配缓冲区的情况下，能存储的最大文本长度），和已经持有的文本的长度。这些都是 `std::string` 的私有字段，使用者不能访问。

一个 `std::string` 拥有它的缓冲区，当程序销毁 `string` 时，它的析构函数会释放缓冲区。以前，一些 C++ 库在多个 `std::string` 值之间共享单个缓冲区，使用一个引用计数来决定缓冲区什么时候应该被释放。较新版本的 C++ 标准有效地排除了这种表示，所有现代的 C++ 库都是用上图中的方式。

¹译者注：此处原文：For us, Rust's deal is a no-brainer.

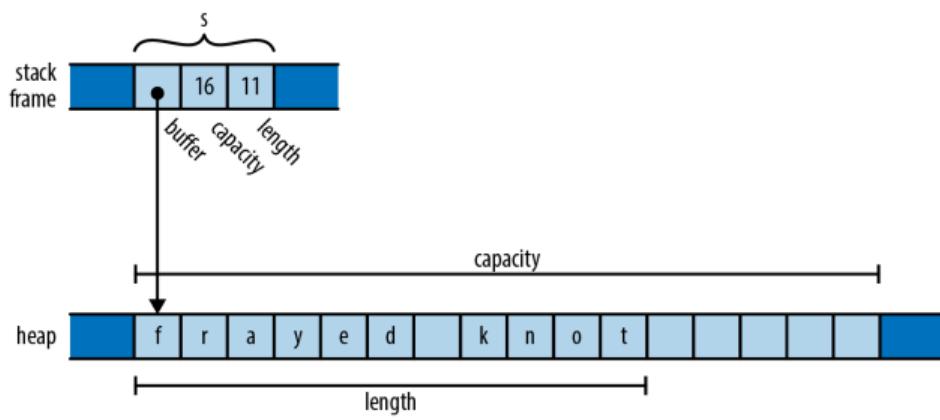


图 4-1: 一个栈上的 C++ `std::string`, 指向它在堆上分配的内存

在这些场景中，人们普遍认为尽管其他代码创建这些被拥有的内存的指针是没问题的，但这些代码有责任确保在所有者决定销毁它拥有的对象之前所有的这种指针都已消失。你可以创建一个指向 `std::string` 的缓冲区的指针，但当 `string` 被销毁后，你的指针就无效了，你必须自己保证不再使用它。拥有者决定所拥有对象的生命周期，所有其他的对象必须尊重它的决定。

我们在这里使用 `std::string` 做为例子展示了 C++ 中的所有权是什么样子的：它只是一个标准库普遍遵守的规范。然而即使语言鼓励你也遵守相似的实践，但如何设计你自己的类型最终还是取决于你。

但在 Rust 中，所有权的概念被内建在语言之中，并且通过编译期检查确保强制执行。每个值都只有一个决定它生命周期的所有者。当所有者被释放——Rust 中的术语叫 *dropped*——它拥有的值也会被 *dropped*。这些规则的目的是让你可以通过检查代码很容易的查明某个值的生命周期，并给你系统语言应有的控制生命周期的能力。

一个变量拥有它的值。当控制流离开了变量声明的语法块，变量会被 drop，因此它的值也会随之一起 drop。例如：

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

变量 `padovan` 的类型是 `Vec<i32>`，一个 32 位整数的 vector。在内存中，`padovan` 看起来将

类似于图 4-2。

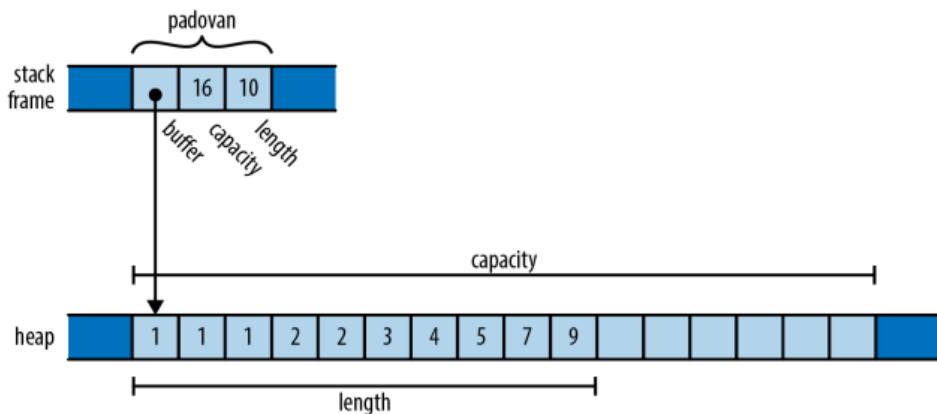


图 4-2: 栈上的 `Vec<i32>`，指向它在堆上的缓冲区

这和我们之前展示的 C++ 的 `std::string` 非常像，除了缓冲区里的元素是 32 位整数，而不是字符。注意存储 `padovan` 的指针、容量和长度的字都在 `print_padovan` 函数的栈帧中，只有 `vector` 的缓冲区是在堆上分配的。

和之前展示的 `string s` 一样，`vector` 拥有它用来存储元素的缓冲区。当变量 `padovan` 在函数结尾处离开作用域时，程序会 drop 这个 `vector`。因为 `vector` 拥有它的缓冲区，缓冲区也会随之 drop。

Rust 的 `Box` 类型是另一个所有权的例子。`Box<T>` 是一个指针，指向一个存储在堆上的类型 `T` 的值，调用 `Box::new(v)` 会在堆上分配一些空间，把值 `v` 移动进去，然后返回一个 `Box` 指向堆上的空间。因为一个 `Box` 拥有它所指向的空间，当 `Box` 被 drop 的时候，堆上的空间也会被释放。

例如，你可以像这样在堆上分配一个元组：

```
{
    let point = Box::new((0.625, 0.5));      // point 在这里分配
    let label = format!("{}:{}", point);       // label 在这里分配
    assert_eq!(label, "(0.625, 0.5)");
}                                         // point 和 label 都在这里 drop
```

当程序调用 `Box::new` 时，它会在堆上为一个由两个 `f64` 值组成的元组分配空间，把它的参数 `(0.625, 0.5)` 移动进去，然后返回一个指向它的指针。当控制流到达 `assert_eq!` 的调用时，栈帧如图 4-3 所示。

栈帧本身存储了变量 `point` 和 `label`，每一个变量都指向自己拥有的堆上的内存。当它们被 drop 时，它们拥有的内存也随之释放。

与变量拥有它们的值类似，结构体拥有它们的字段，元组、数组、vector 拥有它们的元素。

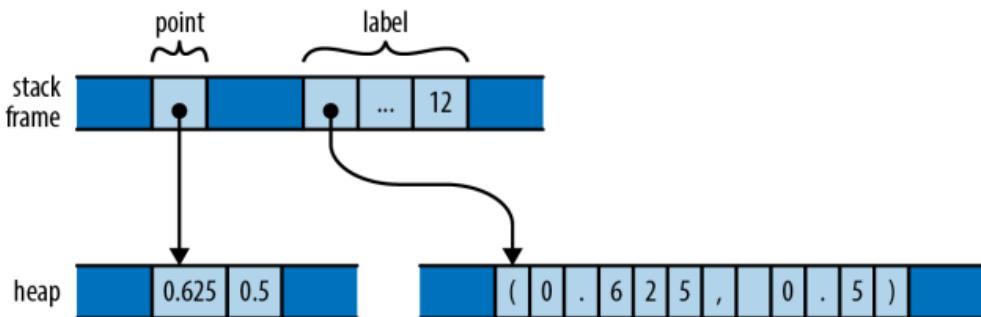


图 4-3: 两个本地变量，每个都拥有堆上的一块内存

```

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);
}

```

这里，`composers`是一个`Vec<Person>`: 一个结构体的vector，每个结构体有一个字符串和数字。在内存中，`composers`的最终结果如图 4-4 所示。

这里有很多的所有权关系，但每一个都很直观：`composers`拥有一个vector，vector拥有它的元素，每一个元素是一个`Person`结构体；每个结构体拥有它的字段；其中的字符串字段拥有它的文本。当控制流离开了`composers`声明的作用域，程序会 drop 它的值，同时 drop 它拥有的所有内容。如果这里还有其他类型的集合，例如`HashMap`、`BTreeSet`，那么过程也是一样的。

到这里，让我们退后一步并思考我们到目前为止展示的所有权关系。每个值都只有一个所有者，这样很容易决定什么时候 drop 这个值。但单个值可能拥有很多其他值：例如，vector`composers`拥有它的所有元素。这些元素也可能反过来拥有其他值：`composers`的每个元素拥有一个字符串，字符串又拥有它的文本。

所有者和它们拥有的值组成了树：值的拥有者是它的父结点，值拥有的值是它的孩子结点。每棵树的根结点是一个变量；当这个变量离开作用域时，整个树都会随之销毁。我们可以在`composers`的图中看到这样一棵所有权的树：它不是搜索树数据结构意义上的“树”、也不是 DOM 元素组成的 HTML 文档树。相反，我们有一个由混合类型构建的树，Rust 的单一

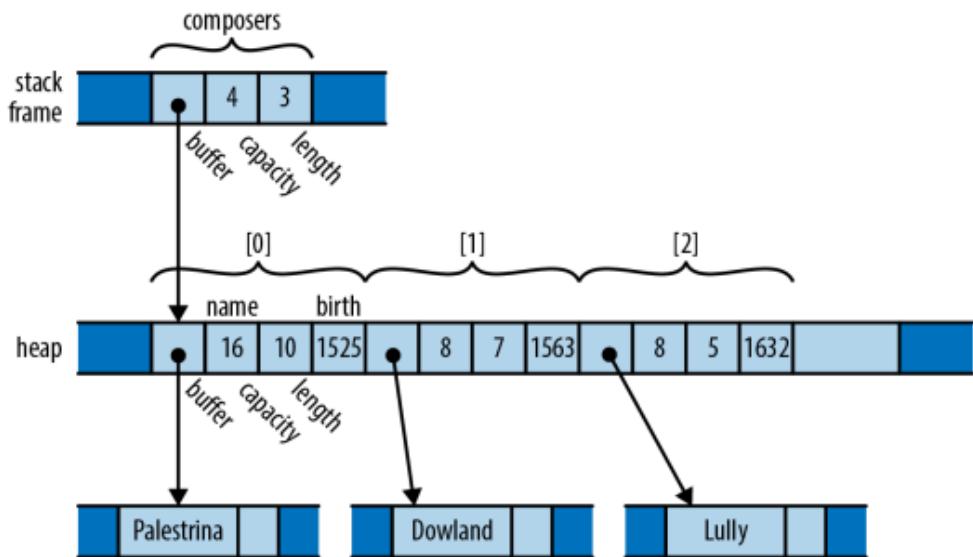


图 4-4: 一个更复杂的所有权树

所有者规则禁止任何可能使布局变得比树更复杂的连接操作。Rust 程序中的每个值都是树中的一个结点，树的根就是变量。

Rust 程序通常完全不需要像 C 和 C++ 程序中使用 `free` 和 `delete` 一样显式 drop 值。Rust 中 drop 值的方式是将它从所有权树移除：当离开作用域时、或者从 vector 中删除元素时、或者类似的情况。这时，Rust 保证值会和它拥有的所有值一起被 drop 掉。

在某种意义上，Rust 不如其他语言强大：每个其他的编程语言都允许你在对象之间构建任意的关系图，这些对象以你认为合适的方式互相指向。但正因为 Rust 不够强大，所以它才可以对你的程序进行更强大的分析。Rust 的安全保证可以实现的原因就是你的代码中可能出现的所有权关系更加容易处理。这是我们之前提到的 Rust 的“激进赌注”的一部分：Rust 声称，在实践中，解决问题时通常有足够的灵活性来保证至少有一些完美的解决方案可以在语言强加的限制范围内实现。

也就是说，我们到目前为止解释的所有权的概念太过死板以至于很难使用。Rust 在以下几个方面扩展了这个简单的想法：

- 你可以将值从一个所有者移动到另一个所有者。这允许你构建、更改、拆除所有权树。
- 很简单的类型例如整数、浮点数和字符被所有权规则排除在外。它们被称为 Copy 类型。
- 标准库提供了引用计数的指针类型 `Rc` 和 `Arc`，它们允许值在一定的限制下可以有多个所有者。
- 你可以“借用一个值的引用”，引用是生命周期受限的非占有的指针。

这些策略中的每一条都改善了所有权模型的灵活性，同时仍然坚持 Rust 的承诺。我们将

依次介绍它们，引用将在下一章介绍。

4.2 move

在 Rust 里对大多数类型来说，赋值给变量、把值传给函数、或者从函数返回值并不会拷贝这个值：它们只会 *move* 它。源对象放弃了值的所有权，把所有权转移给了目的对象，同时源对象变为未初始化的状态；此时目的对象控制值的生命周期。Rust 程序一次一个值、一次 *move* 一个地构建和拆除复杂的结构。

你可能会很惊讶 Rust 改变了这些基础操作的含义。确实赋值操作很早之前就已经有了明确的含义。然而，如果你仔细观察过不同的语言是怎么处理赋值操作的，你就会发现不同语言的处理方式有很大差别。这种差别也让我们能更容易的看出 Rust 的选择的含义和结果。

考虑下面的 Python 代码：

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

每一个 Python 对象都有一个引用计数，用来追踪当前有多少个值指向它。因此在对 `s` 赋值以后，程序的状态如图 4-5 所示（注意有一些内容被省略了）。

因为只有 `s` 指向列表，所以列表的引用计数是 1；因为列表是唯一指向那些字符串的对象，所以每个字符串的引用计数也是 1。

当程序执行到 `t` 和 `u` 的赋值时会发生什么？Python 把赋值操作简单实现为让目标变量也指向源变量指向的对象，然后增加对象的引用计数。因此，这段程序的最终状态如图 4-6 所示：

Python 拷贝了 `s` 的指针，并赋给了 `t` 和 `u`，然后把列表的引用计数更新为 3。Python 中的赋值开销很低，但因为它创建了新的指向对象的引用，我们必须维护引用计数来知道我们什么时候可以释放值。

现在考虑下面类似的 C++ 代码：

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

一开始 `s` 的值在内存中如图 4-7 所示。

当把 `s` 赋值给 `t` 和 `u` 时会发生什么呢？在 C++ 里赋值一个 `std::vector` 会产生一份这个 `vector` 的拷贝；`std::string` 的行为类似。因此当程序到达末尾时，它实际上有 3 个 `vector` 和 9 个字符串（图 4-8）。

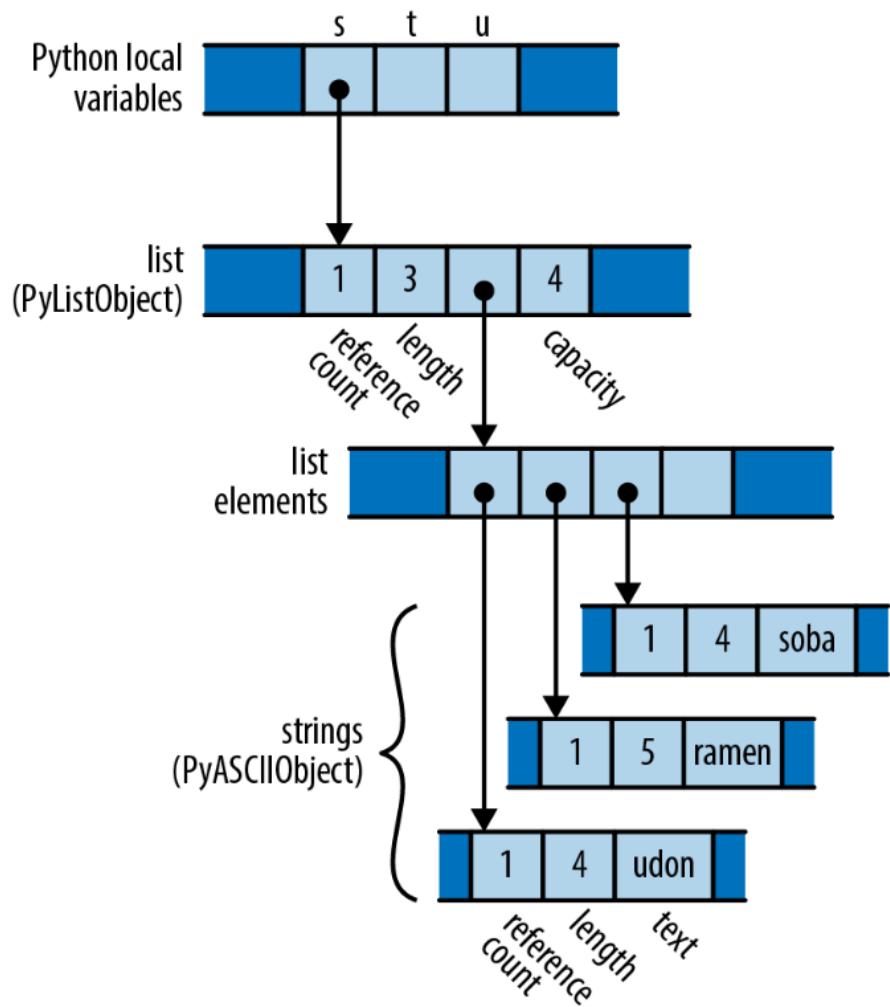


图 4-5: Python 如何在内存中表示一个字符串的列表

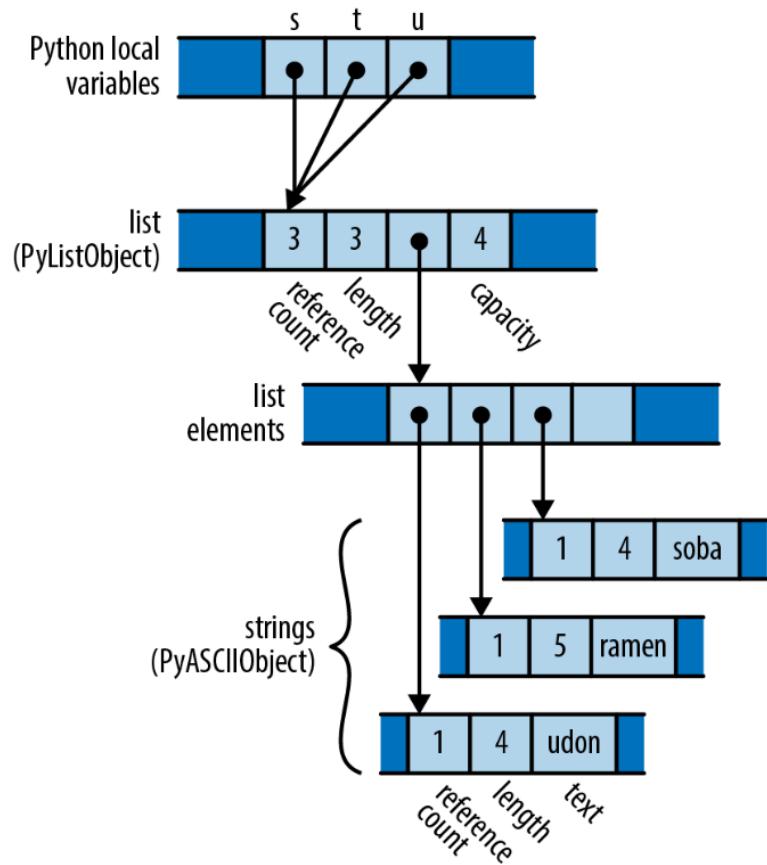


图 4-6: 在 Python 里把 s 赋值给 t 和 u 的结果

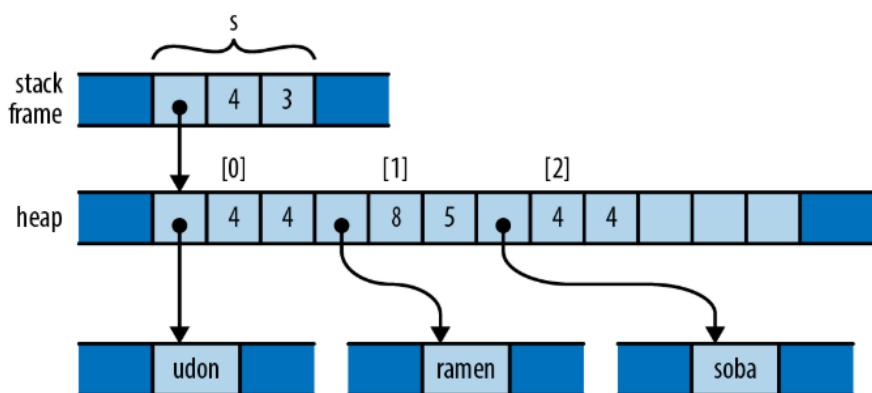


图 4-7: C++ 里一个字符串的 vector 在内存中的表示

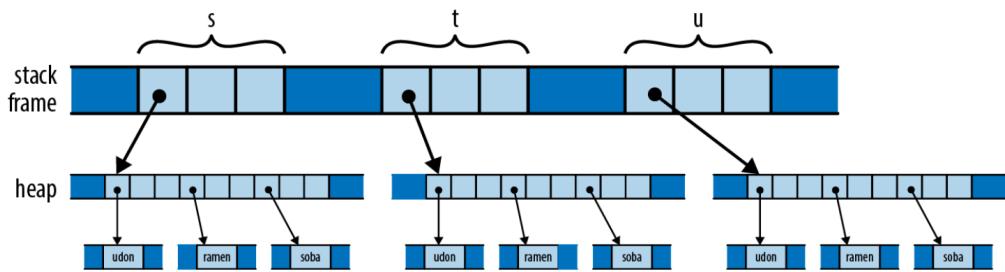


图 4-8: 在 C++ 里把 s 赋值给 t 和 u 的结果

根据值的不同，C++ 里的赋值可能会消耗任意数量的内存和处理器时间。然而，它的优势是，程序可以很容易的决定何时释放这些内存：当变量离开作用域时，所有这里分配的内存都会被自动释放。

某种意义上，C++ 和 Python 选择了相反的策略：Python 里赋值操作开销很小，但引用计数（通用一点的说法，垃圾回收）开销很大。C++ 保持了内存的所有权都很清楚，但赋值时会执行对象的深拷贝导致开销很大。C++ 程序员通常不太热衷于这种选择：深拷贝可能开销很大，通常会有更好的替代方法。

那么 Rust 中的类似程序会怎么做呢？代码如下：

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

类似于 C 和 C++，Rust 把字符串字面量例如"udon" 存储在只读内存中，因此，为了更清楚地与 C++ 和 Python 的例子进行对比，我们调用了 `to_string` 来获得在堆上分配的 String 值。

在 s 的初始化之后，因为 Rust 和 C++ 有相似的 vector 和 string 表示，所以看起来和 C++ 中的情况很像（图 4-9）。

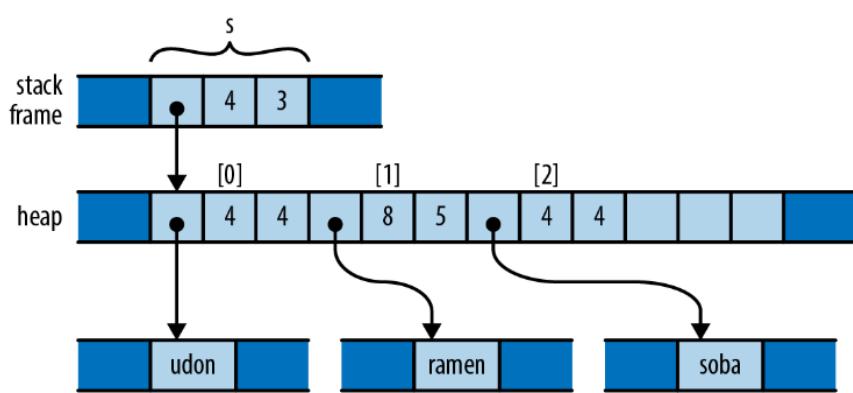


图 4-9: Rust 中一个字符串的 vector 在内存中的表示

但回想一下，Rust 里大多数类型的赋值操作都是把值从源对象移动到目的对象，然后源对象变为未初始化的状态。因此 `t` 初始化完之后，程序的内存状态如图 4-10 所示。

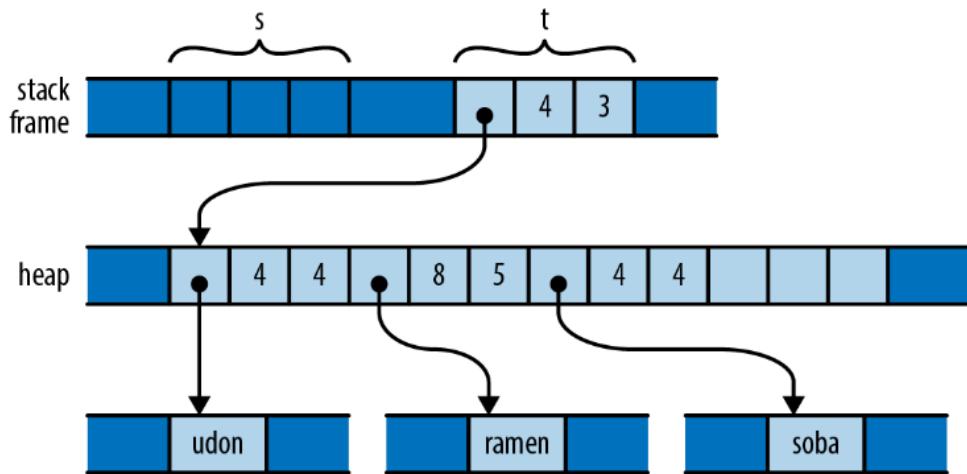


图 4-10: Rust 里把 `s` 赋值给 `t` 之后的结果

这里发生了什么？赋值语句 `let t = s;` 把 `vector` 的三个字段从 `s` 移动到了 `t`；现在 `t` 拥有了这个 `vector`。`vector` 的元素则仍待在原来的位置，`string` 的位置也没有发生变化。每一个值都只有一个所有者，尽管所有者已经变了。不需要调整引用计数，并且编译器现在把 `s` 视作未初始化的状态。

因此当我们到达 `let u = s;` 时会发生什么呢？这将会把 `s` 赋值给 `u`。Rust 禁止使用未初始化的值，所以编译器会报如下错误：

```
error[E0382]: use of moved value: `s`
|
7 |     let s = vec!["udon".to_string(), "ramen".to_string(), "soba".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 after move
```

考虑 Rust 在这里使用 move 的结果。类似于 Python，赋值操作开销很小，程序简单地把 `vector` 的三个字长的头部从一个点移动到了另一个点。但和 C++ 类似，所有权总是很清晰：程序不需要引用计数或者垃圾回收来判断什么时候释放 `vector` 的元素和 `string` 的内容。

而为此付出的代价是如果赋值时想要的是拷贝操作那么必须显式写出。如果你想要最后和 C++ 程序一样的状态，也就是每个变量都有独立的拷贝，那么必须调用 `vector` 的 `clone` 方

法，它会对 vector 和它的元素执行深拷贝：

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

你也可以通过 Rust 的引用计数指针类型复现 Python 代码的行为，我们将在[Rc 和 Arc：共享所有权](#)这一节中简要介绍这一点。

4.2.1 更多 move 的操作

在我们上面展示的初始化例子中，都是在使用 `let` 语句引入变量的同时把值赋给它们。赋值给一个变量将与此有细微的不同，如果你把值移动进一个已经被初始化的变量，Rust 会 drop 变量之前的值。例如：

```
let mut s = "Govinda".to_string();
s = "Siddhartha".to_string(); // 值"Govinda"在这里drop
```

在这段代码中，当程序把 "Siddhartha" 赋值给 `s` 时，它之前的值 "Govinda" 首先被 drop 掉。但考虑下面的代码：

```
let mut s = "Govinda".to_string();
let t = s;
s = "Siddhartha".to_string(); // 这里不会drop任何内容
```

这一次，`t` 拿走了 `s` 中原本的字符串的所有权，因此当我们给 `s` 赋值时，它是未初始化的。在这种场景下，不会发生 drop。

我们在这里使用初始化和赋值的例子是因为它们足够简单，但 Rust 在几乎所有场景下都使用 move。向函数传参会把所有权移动给函数的参数；从函数返回值会把所有权移动给调用者；创建一个元组会把值移动给元组，等等。

你现在可能对我们之前章节给出的例子中到底发生了什么有了更深入的理解。例如，当我们构建作曲家的 vector 时，我们写了：

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

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

这段代码展示了除了初始化和赋值之外，move 发生的几个场景：

从函数返回值

调用 `Vec::new()` 会创建一个新的 `vector` 并返回，返回的并不是指向 `vector` 的指针，而是 `vector` 本身：它的所有权从 `Vec::new` 移动到了变量 `composers`。类似的，`to_string` 调用返回了一个新的 `String` 实例。

构造新的值

新的 `Person` 结构体的 `name` 字段被 `to_string` 的返回值初始化。结构体获得了这个字符串的所有权。

向函数传递值

整个 `Person` 结构体，而不是指向它的指针，被传递给 `vector` 的 `push` 方法，这个方法将值移动到了结构体的尾部。`vector` 获得了 `Person` 的所有权，因此也变成了 `name String` 的间接所有者。

像这样移动值可能听起来并不是很高效，但有两件事需要记住。第一，`move` 只作用于恰当的值，而不作用于它们拥有的堆存储。对于 `vector` 和 `string` 来说，恰当的值是它们三个字长的头部，潜在的很多元素的数组和文本缓冲区仍然停留在堆中原本的位置。第二，Rust 编译器的代码生成部分擅长“看穿”所有这些 `move`；在实践中，机器码通常会直接把值存储到它最终的位置。

4.2.2 move 和控制流

之前的例子中的控制流都很简单，`move` 会如何影响更复杂的代码呢？通用的原则是，如果一个变量的值被移动走并且从此之后没有再被赋予一个新的值，那么它被认为是未初始化的。例如，如果一个变量在 `if` 表达式的条件判断之后还是有值的，那我们在两个分支中都可以使用它：

```
let x = vec![10, 20, 30];
if c {
    f(x);    // ... 在这里移动 x 的值是 ok 的
} else {
    g(x);    // ... 在这里移动 x 的值也是 ok 的
}
h(x);    // 错误：如何任何一个分支使用了 x，那么 x 在此处将是未初始化的
```

出于类似的原因，在循环里移动一个变量的值是禁止的：

```
let x = vec![10, 20, 30];
while f() {
    g(x);    // 错误：x 会在第一次迭代时被移动
```

```
// 第二次迭代时就是未初始化状态
}
```

也就是说，我们需要在每次迭代里都重新赋予它一个新值：

```
let mut x = vec![10, 20, 30];
while f() {
    g(x);      // 移动x的值
    x = h();    // 给x一个新值
}
```

4.2.3 move 和索引

我们已经提到过 move 会将源对象设置为未初始化状态，目的对象会获得值的所有权。但并不是每一种值的所有者都可以设置为未初始化状态。例如，考虑下面的代码：

```
// 创建一个string 的vector: "101", "102", ... "105"
let mut v = Vec::new();
for i in 101 .. 106 {
    v.push(i.to_string());
}

// 从vector 中取出随机的元素
let third = v[2];    // 错误：不能移动Vec 的索引
let fifth = v[4];    // 这里也是一样
```

如果想让这段代码工作，Rust 需要记住这个 vector 的第 3 和第 5 个元素变成了未初始化状态，然后一直追踪这些信息直到这个 vector 被 drop。在一般情况下，vector 需要携带额外的信息来指示哪些元素还可用，哪些变为了未初始化。显然这不是一门系统编程语言应该有的行为；一个 vector 应该只是一个 vector。事实上，Rust 会报错拒绝上面的代码：

```
error[E0507]: 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]`
```

移动到 fifth 的语句也会报类似的错误。在这些错误信息中，Rust 建议使用引用，这样就可以在不移动的情况下访问元素。这通常是你想要的。但如果我真的想从 vector 中移出一个元素呢？你需要找到一些不违反类型限制的方法来做这件事。这里有三种方法：

```
// 创建一个string的vector: "101", "102", ... "105"
let mut v = Vec::new();
for i in 101 .. 106 {
    v.push(i.to_string());
}

// 1. 弹出vector尾部的元素
let fifth = v.pop().expect("vector empty!");
assert_eq!(fifth, "105");

// 2. 移出给定位置的元素，并把最后一个元素移动过来：
let second = v.swap_remove(1);
assert_eq!(second, "102");

// 3. 用另一个值和我们想移出的值交换
let third = std::mem::replace(&mut v[2], "substitute".to_string());
assert_eq!(third, "103");

// 让我们看看vector中还剩下什么。
assert_eq!(v, vec!["101", "104", "substitute"]);
```

这三种方法都从 vector 中移出一个元素，但仍然保证 vector 处于没有空隙的状态，可能长度还会变小。

像 Vec 这样的集合类型也提供方法通过循环消费它们的所有元素：

```
let v = vec![ "liberté".to_string(),
             "égalité".to_string(),
             "fraternité".to_string()];
for mut s in v {
    s.push('!');
    println!("{}!", s);
}
```

当直接把 vector 传给循环时，例如 `for ... in v`，这会把 v 中的所有元素移出 vector，然后 v 变为未初始化。for 循环内部的机制会获取 vector 的所有权，然后把它分解为若干元素。每一次迭代时，循环都会把一个元素移动到变量 s。因为 s 现在拥有这个字符串，所以我们可以直接修改它，然后再打印。因为 vector 本身不再对代码可见，所以在循环的过程中当它部分为空时没有任何东西可以观测到它。

如果你确实发现你需要从所有者中移出一个编译器无法追踪的值，你可以考虑将所有者的类型改为可以动态追踪是否有值的类型。例如，这里有一个之前示例的变体：

```
struct Person { name: Option<String>, birth: i32 }

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

你不能这样做：

```
let first_name = composers[0].name;
```

这只会和犯和之前一样的“不能移出索引”的错误。但因为你把 name 字段的类型从 String 改为了 Option<String>，这意味着 None 是这个字段的一个合法的取值，因此下面的代码可以生效：

```
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 调用移出了 composers[0].name 的值，留下了一个 None，并把原本的值的所有权传递给了调用者。事实上，这种使用 Option 的方法非常普遍，所以这个类型提供了一个 take 方法来实现这个特殊的用途。你可以使用下面的代码更清晰地完成上述操作：

```
let first_name = composers[0].name.take();
```

这里的 take 调用和之前的 replace 调用有相同的效果。

4.3 Copy 类型：move 的例外

到目前为止我们的例子涉及 vector、string、和其他可能潜在地使用大量内存并且拷贝开销很大的类型。move 保证了这些类型的所有权的明晰、也保证了赋值的开销很小。但对于简单类型例如整数或字符，这种谨慎的处理方式事实上并不是必须的。

让我们比较一下赋值一个 String 和赋值一个 i32 值时内存中会发生什么：

```
let string1 = "somnambulance".to_string();
let string2 = string1;

let num1: i32 = 36;
let num2 = num1;
```

运行完这段代码后，内存布局如图 4-11 所示。

类似于之前的 vector，赋值会把 string1 移动到 string2，这样我们就不会得到两个负责释放同一个缓冲区的字符串。然而，num1 和 num2 的情况与此不同。i32 只是在内存中的一种位

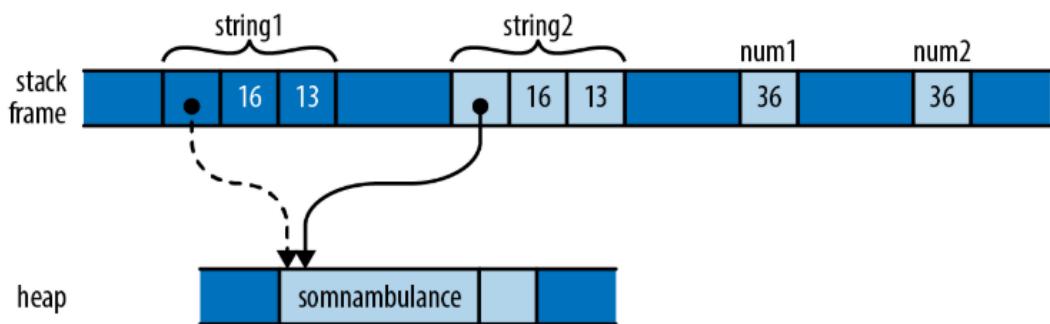


图 4-11: 赋值 String 会移动它, 而赋值 i32 会拷贝它

模式, 它并不拥有任何堆上的资源、也不依赖任何它本身所占的字节之外的东西。此时如果我们把它的位都移动给 num2, 我们就得到了一个和 num1 完全互相独立的拷贝。

移动一个值会导致源对象变为未初始化。但是尽管把 string1 视为无值是一个基本的目的, 但如果对 num1 也这么做是毫无意义的, 继续使用 num1 不会导致任何危害。move 的优势在这里并不适用, 反而变得不够便捷。

之前我们谨慎的说过大多数类型会被移动, 现在我们来到了例外的情况, 也就是 Rust 称之为 `Copy type` 的类型。赋予一个 `Copy` 类型的值会拷贝它, 而不是移动它。源对象仍然保持初始化状态和可用性, 它的值不会发生改变。向函数和构造器传递 `Copy` 类型也类似。

标准的 `Copy` 类型包括所有的机器整数和浮点数类型、`char` 和 `bool` 类型, 以及少数其他类型。所有元素都是 `Copy` 类型的元组或数组也是 `Copy` 类型。

只有简单的逐位拷贝的类型才可以是 `Copy` 类型。正如我们解释过的, `String` 不是 `Copy` 类型, 因为它拥有一个在堆上分配的缓冲区; 与此类似, `Box<T>` 也不是 `Copy` 类型, 它拥有一个堆上分配的对象; 代表操作系统中文件句柄的 `File` 类型, 也不是 `Copy` 类型, 赋值这样一个值意味着向操作系统请求另一个文件句柄。类似的, 表示一个互斥锁的 `MutexGuard` 类型, 也不是 `Copy` 类型, 拷贝这个类型没有任何意义, 因为在一个时间点只有一个线程可以持有锁。

根据经验, 任何在 `drop` 时要做一些特殊事情的类型不可能是 `Copy` 类型: 一个 `Vec` 需要释放它的内存, 一个 `File` 需要关闭它的文件句柄, 一个 `MutexGuard` 需要释放它的锁, 等等。对这些类型逐位拷贝会导致搞不清它们中的哪一个要负责释放原始的资源。

自定义的类型呢? 默认情况下, `struct` 和 `enum` 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);
```

这不能通过编译，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
```

因为 `Label` 不是 `Copy` 类型，将它传递给 `print` 会把值的所有权移动给 `print` 函数的参数，在函数返回时会 drop 掉值。但这非常愚蠢，一个 `Label` 只是一个 `u32` 套壳。没有理由将 `l` 传递给 `print` 时应该移动值。

但用户自定义类型只是默认不是 `Copy` 类型。如果你的结构体的所有字段都是 `Copy` 类型，那么你可以通过在定义上方加上属性 `#[derive(Copy, Clone)]` 来把它变为 `Copy` 类型，像这样：

```
#[derive(Copy, Clone)]
struct Label { number: u32 }
```

这样修改之后，上面的代码就可以通过编译了。然而，如果我们用不是 `Copy` 类型的字段来修改这个类型的话，也不能通过编译。假设我们要编译下面的代码：

```
#[derive(Copy, Clone)]
struct StringLabel { name: String }
```

它会报这样的错误：

```
error[E0204]: the trait `Copy` may not be implemented for this type
--> ownership_string_label.rs:7:10
|
7 | #[derive(Copy, Clone)]
|     ^
8 | struct StringLabel { name: String }
|               ----- this field does not implement `Copy`
```

为什么不默认把用户自定义类型设置为 `Copy` 类型？一个类型是否是 `Copy` 类型对接下来的代码如何使用它有巨大的影响：`Copy` 类型更加的灵活，因为赋值和相关的操作不会导致源对象变得未初始化。但对于类型的实现者来说，恰恰相反：`Copy` 类型所能包含的类型十分有限，

而非 Copy 类型可以使用堆上的内存并且拥有自己的资源。因此将一个类型标记为 Copy 代表着实现者的一个承诺：如果之后它必须要修改为非 Copy 类型，很多使用它的代码都需要修改。

虽然 C++ 允许你重载赋值运算符和自定义拷贝和移动构造函数，但 Rust 不允许这种自定义。在 Rust 里，所有的移动都是逐字节的浅拷贝，同时把源对象设置为未初始化。拷贝与此类似，除了源对象仍然是初始化过的状态。这确实意味着 C++ 类可以提供 Rust 类所不能提供的方便接口：看起来普通的代码会隐式的调整引用计数、推迟开销很大的拷贝操作、或者使用其它复杂的实现技巧。

但 C++ 中的这种灵活性会导致基本操作例如赋值、传参、返回值变得不可预料。例如，这一章之前的部分我们展示过在 C++ 里把一个变量赋给另一个可能会需要任意数量的内存和处理器时间。Rust 的原则之一就是开销对程序员来说必须是明显的。基本的操作必须保持简单。潜在的开销很大的操作必须是显式的，例如在更早的例子中调用 `clone` 来获取 vector 和它包含的 string 的深拷贝。

在这一节中，我们谈到了术语 Copy 和 Clone，模糊地将它们视作类型具有的特征。事实上，它们是 trait 的示例，它是 Rust 的一个开发的工具，你可以通过它根据类型能做什么对类型进行分类。我们将在第 11 章中讨论一般性的 trait，在第 13 章中专门讨论 Copy 和 Clone。

4.4 Rc 和 Arc: 共享所有权

尽管典型的 Rust 代码中几乎所有的值都只有一个所有者，但在一些情况下，很难让每个值的所有者都有你想要的生命周期，这时你可能会希望值一直保持有效，直到每一个所有者都使用完它。对于这种情况，Rust 提供了引用计数类型 `Rc` 和 `Arc`。正如你对 Rust 的期待一样，它们是安全的：你不可能忘记调整引用计数、创建其它指向它的指针、或者出现其他使用 C++ 的引用计数指针类型时可能出现的问题。

`Rc` 和 `Arc` 类型非常相似，它们唯一的不同之处在于 `Arc` 可以直接安全地在线程之间共享——名称 `Arc` 是原子引用计数的缩写——`Rc` 则使用更快一些的非线程安全代码来更新引用计数。如果你不需要在线程之间共享指针，那就没有必要承担 `Arc` 的性能损失，所以你应该使用 `Rc`；Rust 会阻止你无意间在线程之间传递 `Rc`。这两种类型在其他方面都是等价的，因此在本节的剩余部分，我们将只讨论 `Rc`。

之前我们曾经展示过 Python 使用引用计数来管理值的生存周期。你可以使用 `Rc` 来在 Rust 中实现相似的效果。考虑下面的代码：

```
use std::rc::Rc;
```

```
// Rust 可以推断出所有这些类型，写出来是为了更清楚
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，一个 `Rc<T>` 值是一个指向在堆上分配的 T 类型值的指针，同时还附有一个引用计数。克隆一个 `Rc<T>` 类型的值并不意味着拷贝 T，它只是简单的创建另一个指向它的指针，并且递增引用计数。因此上面的代码会产生如图 4-12 所示的内存布局：

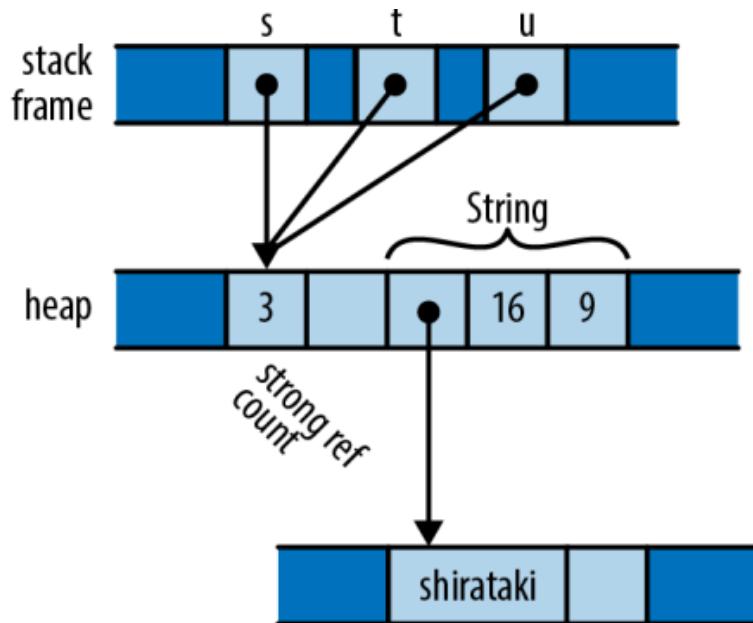


图 4-12: 一个有三个引用的引用计数字符串

这三个 `Rc<String>` 指针都指向内存中的同一块内存，这块内存里存储了一个引用计数和一个 `String`。通常的所有权规也适用于 `Rc` 指针，当最后一个 `Rc` 指针 drop 时，Rust 会同时 drop 掉 `String`。

你可以直接对 `Rc<String>` 使用任何 `String` 的方法：

```
assert!(s.contains("shira"));
assert_eq!(t.find("taki"), Some(5));
println!("{} are quite chewy, almost bouncy, but lack flavor", u);
```

一个 `Rc` 指针拥有的值是不可变的。假设你尝试在字符串的结尾添加文本：

```
s.push_str(" noodles");
```

Rust 将会报错：

```

error: cannot borrow data in an `Rc` as mutable
--> ownership/ownership_rc_mutability.rs:13:5
|
13 |     s.push_str(" noodles");
|     ^ cannot borrow as mutable
|

```

Rust 的内存和线程安全保证依赖于没有值既是共享的又是可变的。Rust 假设 Rc 指针指向的值要被共享，因此它必须是不可变的。我们将在[第 5 章](#)解释为什么限制要这么严格。

使用引用计数来管理内存的一个已知问题就是，如果两个引用计数的值互相指向彼此，那么每一个都会导致对方的引用计数不可能降到 0，因此值永远不会被释放（[图 4-13](#)）。

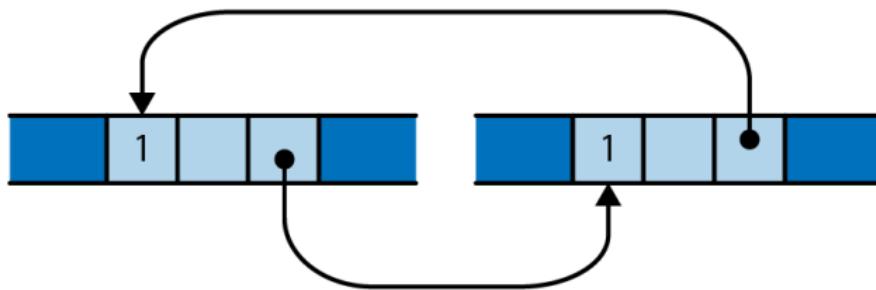


图 4-13: 一个引用计数循环，这些对象永远不会被释放

在 Rust 中出现这种泄漏是可能的，但这种情况非常少见。要想创建这样一个循环，你必须让旧值指向新值，这也意味着旧值要是可变的。因为 Rc 指针指向的值不可变，所以通常是不可能创建这样的循环的。然而，Rust 确实提供了一些方法创建部分可变的值的方法；这被称为内部可变性，我们将在[内部可变性](#)一节中介绍。如果你将那些技术和 Rc 指针结合使用，那么你确实能创建出一个循环，然后造成内存泄漏。

你可以使用弱指针 `std::rc::Weak` 来避免使用 Rc 指针创建循环的情况。然而，我们不会在本书中介绍这些，你可以查看标准库的文档获取详情。

`move` 和引用计数指针是两种缓解所有权树过于死板的方法。在下一章中，我们将看到第三种方法：借用值的引用。一旦你对所有权和借用都感到很舒服，那你就已经跨过了 Rust 的学习曲线中最陡峭的部分，而且已经准备好接受 Rust 独特的优势。

Chapter 5

引用

Libraries cannot provide new inabilities.

——Mark Miller

我们至今为止见过的所有指针类型——简单的 `Box<T>` 堆指针、`String` 和 `Vec` 内部的指针都拥有值：当所有者被 `drop` 时，指针指向的值也会随之消失。Rust 还有非拥有指针类型，称为引用，引用对指向的值的生命周期没有影响。

事实上，正相反，引用绝不应该比它们指向的值活的更长。你必须在你的代码中明确表明引用的寿命比它指向的值更短。为了强调这一点，Rust 将创建某个值的引用称为借用 (*borrow*) 值：最终必须把借走的还给所有者。

如果你在读到“你必须在你的代码中明确表明”时感到一阵怀疑，那说明你很优秀。引用自身并没有什么特殊的——本质上，它们只是地址。但保证它们安全的规则是 Rust 独有的，你以前不可能看到过类似的。尽管这些规则是 Rust 里最难掌握的部分，但它们能防止经典的、日常的 bug 的范围之广令人惊讶，它们对多线程的影响也正在显现。这也是 Rust 的赌注。

这一章中，我们将讨论 Rust 中的引用如何工作，展示引用、函数和自定义类型如何包含生命周期信息来保证它们被安全使用，阐释它怎么能在编译期、不引入运行时开销的同时防止常见的 bug。

5.1 值的引用

举个例子，假设我们要为文艺复兴时期优秀的艺术家和他们的著名作品建一个表格。Rust 的标准库包含一个哈希表类型，所以我们可以像这样定义我们的类型：

```
use std::collections::HashMap;
```

```
type Table = HashMap<String, Vec<String>>;
```

换句话说，这是一个把 `String` 值映射到 `Vec<String>` 值的哈希表，它把艺术家的名字关联到它们的作品的名字。你可以使用 `for` 循环来迭代 `HashMap` 的条目，因此我们可以写一个函数打印出一个 `Table`：

```
fn show(table: Table) {
    for (artist, works) in table {
        println!("works by {}: ", artist);
        for work in works {
            println!("  {}", work);
        }
    }
}
```

构造和打印表格都很直观：

```
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);
}
```

它也能正常工作：

```
$ 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
```

但如果你阅读过上一章中有关 move 的小节，你就会发现 show 的定义有一些问题。首先，HashMap 不是 Copy 类型——它不可能是，因为它持有动态分配的表格。因此当程序调用 show(table) 时，整个结构体都被移动到函数里，变量 table 将变为未初始化。（迭代它时没有特定的顺序，你可能会得到一个不同的顺序，不用担心）如果调用者代码尝试继续使用 table，它会遇到问题：

```
...
show(table);
assert_eq!(table["Gesualdo"][0], "many madrigals");
```

Rust 会报错 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，如果我们仔细查看 show 的定义，会发现外层的 for 循环获取了哈希表的所有权然后完全消费了它，内层的 for 循环对每一个 vector 做了同样的事（我们之前已经在 “liberté, égalité, fraternité” 的例子中见过这种行为了）。因为 move 语义，我们仅仅是为了打印它就已经完全销毁了整个结构体。感谢你，Rust！

正确的处理方式是使用引用。引用让你可以访问一个值，同时不影响它的所有权。引用有两种：

- 共享引用让你能读取但不能修改被引用的值。然而，你可以同时持有多个共享引用。表达式 &e 返回一个指向 e 的值的共享引用，如果 e 的类型是 T，那么 &e 的类型就是 &T，读作 “ref T”。共享引用是 Copy 类型。
- 如果你有一个值的可变引用，你可以读取和修改这个值。然而，你不能同时再有任何其他有效的引用。表达式 &mut e 返回一个指向 e 的值的可变引用，它的类型是 &mut T，读作 “ref mute T”。可变引用不是 Copy 类型。

你可以将共享和可变引用的区别看作是一种在编译期强制多个读者或一个写者的规则的方法。事实上，这个规则不仅适用于引用，还适用于被借用的值的所有者。只要一个值有共

享引用存在，就算是它的拥有者也不能修改它，此时这个值已经被锁定了。当 `show` 正在使用 `table` 时没有人可以修改 `table`。类似的，当一个值有可变引用时，它会排斥其他所有对这个值的访问，此时你不能使用值的拥有者，直到可变引用消失。将共享和可变完全分离开来是内存安全的基础，我们将会在下一章介绍原因。

我们的示例中的打印函数不需要修改表格，只需要读取它的内容。所以调用者可以以共享引用的方式传递表格，像下面这样：

```
show(&table);
```

引用是非拥有指针，所以 `table` 变量仍然保留着整个数据结构的所有权，`show` 只是借用了它。当然，我们还需要调整 `show` 的定义来进行匹配，但你必须仔细看才能看出其中的区别：

```
fn show(table: &Table) {
    for (artist, works) in table {
        println!("works by {}: ", artist);
        for work in works {
            println!("  {}", work);
        }
    }
}
```

`show` 的参数 `table` 的类型从 `Table` 变为了 `&Table`：不再以值传递表格（会导致所有权移动到函数里），我们现在传递一个共享引用。这就是表面上唯一的变化。但是当我们执行函数体时到底是怎么工作的？

我们原先的版本 `for` 循环会获取 `HashMap` 的所有权并消耗它。在新版本中它接受一个 `HashMap` 的共享引用，迭代 `HashMap` 的共享引用被定义为产生每个条目的 `key` 和 `value` 的引用：`artist` 从一个 `String` 变为 `&String`，`works` 从一个 `Vec<String>` 变为 `&Vec<String>`。

内部的循环也有类似的变化。迭代 `vector` 的共享引用被定义为产生它的每个元素的共享引用，因此 `work` 现在是一个 `&String`。这个函数里不再有任何所有权的变化，而是传递各种无所有权的引用。

现在，如果我们想写一个函数来把每个艺术家的作品按字母顺序排列，一个共享引用显然不够，因为共享引用不允许修改。排序函数需要表格的可变引用：

```
fn sort_works(table: &mut Table) {
    for (_artist, works) in table {
        works.sort();
    }
}
```

然后我们需要传递：

```
sort_works(&mut table);
```

这个可变借用赋予了 `sort_works` 读取和修改我们的结构体的能力，以满足 `vector` 的 `sort` 方法的需要。

当我们以会把所有权移动到函数中的方式传参时，我们称这种方式为以值传参。如果我们传递值的引用，我们称之为以引用传参。例如，我们通过将以值传参修改为以引用传参修复了 `show` 函数。许多语言都有这种区别，但它在 Rust 中尤其重要，因为它阐明了所有权如何被引用影响。

5.2 使用引用

上面的例子展示了引用的一个经典应用：允许函数在不获取所有权的情况下访问或者操作一个数据结构。但引用要更加灵活，让我们通过一些例子来进行更深入的了解。

5.2.1 Rust 的引用 vs C++ 的引用

如果你熟悉 C++ 的引用，你会发现它和 Rust 中的引用有很多相似之处。最重要的是，它们在机器层面都只是地址。但在实践中，Rust 的引用使用起来有一种不同的感觉。

在 C++ 中，引用通常通过隐式转换来创建，然后显式解引用：

```
// C++ 代码!
int x = 10;
int &r = x;           // 初始化时隐式创建引用
assert(r == 10);     // 显式解引用 r 来访问 x 的值
r = 20;              // 把 20 存储到 x 中，r 仍然指向 x
```

在 Rust 中，引用通过 `&` 运算符显示创建，通过运算符显式解引用：

```
// 从现在开始回到 Rust 的代码。
let x = 10;
let r = &x;           // &x 是一个 x 的共享引用
assert!(*r == 10);   // 显式解引用 r
```

要想创建可变引用，使用 `&mut` 运算符：

```
let mut y = 32;
let m = &mut y;      // &mut y 是 y 的一个可变引用
*m += 32;            // 显式解引用 m 来访问 y 的值
assert!(*m == 64);   // 查看 y 的新值
```

但是你可能回想起来，我们修改 `show` 函数让它以引用获取艺术家的表格时，我们从来没有使用过 `*` 运算符。这是为什么呢？

因为引用在 Rust 中使用如此广泛，所以如果需要的话，`.` 运算符会隐式解引用它左侧的操作数：

```
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");

// 等价于上面的代码，但显式写出了解引用
assert_eq!((*anime_ref).name, "Aria: The Animation");
```

`show` 函数里使用的 `println!` 宏会展开为使用 `.` 运算符的代码，因此它也利用了这种隐式解引用。

如果方法调用需要的话，`.` 运算符还会隐式借用左侧操作数的引用。例如，`Vec` 的 `sort` 方法会获取 `vector` 的可变引用，因此下面的两个调用时等价的：

```
let mut v = vec![1973, 1968];
v.sort();           // 隐式借用 v 的可变引用
(&mut v).sort();   // 等价写法，不过更详细
```

简而言之，C++ 在引用和左值（指向某个内存地址的表达式）之间隐式转换，如果需要这种转换会在任何地方出现；而在 Rust 中使用 `&` 和 `*` 运算符来创建和解引用，除了 `.` 运算符是个例外，它会自动隐式借用和解引用。

5.2.2 对引用赋值

把一个引用赋值给变量会让这个变量指向新的东西：

```
let x = 10;
let y = 20;
let mut r = &x;
if b { r = &y; }

assert!(*r == 10 || *r == 20);
```

引用 `r` 一开始指向 `x`。但如果 `b` 为 `true`，`r` 将指向 `y`，如图 5-1 所示。

代码的行为似乎太过于简单而不值一提：显然 `r` 现在会指向 `y`，因为我们把 `&y` 赋给了它。但我们指出这一点是因为 C++ 的引用与此差别很大：如之前展示的一样，在 C++ 中对一个引用赋值会把值存储在引用指向的对象。一旦一个 C++ 引用被初始化之后，将没有任何方法让它指向别的东西。

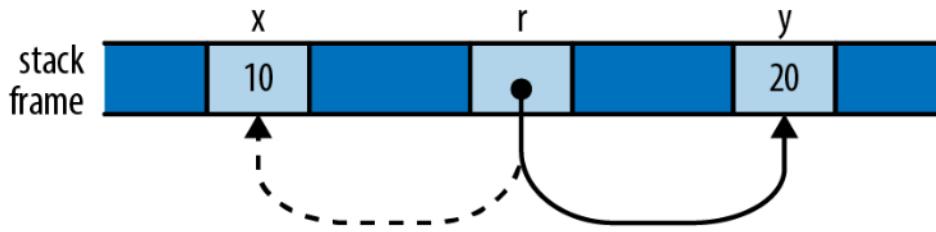


图 5-1: 引用 r 现在指向 y 而不是 x

5.2.3 引用的引用

Rust 允许引用的引用:

```
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;
```

(我们写出引用的类型是为了看得更清楚，但你可以省略它们，Rust 可以推导出这里的所有类型。) . 运算符寻找目标时需要解引用多次:

```
assert_eq!(rrr.y, 729);
```

在内存中，这些引用按照图 5-2 的形式排布。

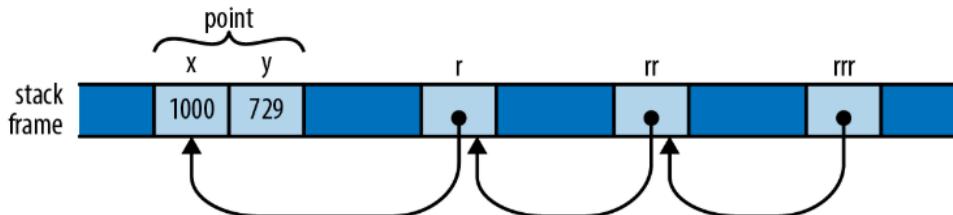


图 5-2: 引用的引用链

因此表达式 `rrr.y`，在 `rrr` 的类型的引导下，实际上进行了 3 次解引用才得到了 `Point` 值，然后才能获得它的 `y` 字段。

5.2.4 比较引用

类似于 . 运算符，Rust 比较运算符可以“看穿”任意数量的引用嵌套:

```
let x = 10;
let y = 10;
```

```

let rx = &x;
let ry = &y;

let rrx = &rx;
let rry = &ry;

assert!(rrx <= rry);
assert!(rrx == rry);

```

这里，最后的断言会成功，尽管 `rrx` 和 `rry` 指向不同的值 (`rx` 和 `ry`)，但因为 `==` 运算符会解除所有的引用然后对最终的值 `x` 和 `y` 进行比较。这几乎总是你想要的行为，尤其是编写泛型函数时。如果你实际上是想知道两个引用是否指向相同的内存位置，你可以使用 `std::ptr::eq`，它会按照地址比较引用：

```

assert!(rx == ry);           // 它们指向的对象相等
assert!(!std::ptr::eq(rx, ry)); // 但是地址不同

```

注意比较运算符的操作数的类型必须完全相同，包括引用：

```

assert!(rx == rrx);      // error: 类型不匹配: `&i32` 和 `&&i32`
assert!(rx == *rrx);     // Ok

```

5.2.5 引用永不为空

Rust 的引用永远不可能为空，没有类似 C 的 `NULL` 或 C++ 的 `nullptr` 的值。引用没有默认初始值（不管什么类型，你不能在一个变量初始化之前使用它）并且 Rust 不允许把整数转换为引用（除了 `unsafe` 代码），因此你不能把 0 转换为引用。

C 和 C++ 经常使用空指针来表示没有值：例如，`malloc` 函数会返回一个指向新内存块的指针，但如果没有任何足够的内存能够满足要求就会返回一个 `nullptr`。在 Rust 里，如果你需要一个可能是引用可能是无的值，你可以使用类型 `Option<&T>`。在机器层面，Rust 将 `None` 表示为空指针，将 `Some(r)`（其中 `r` 是一个 `&T` 类型的值）表示为非 0 的地址，因此 `Option<&T>` 和 C 或 C++ 中的可以为空的指针一样高效，然而它却是安全的：它的类型要求你在使用它之前要先检查它是不是 `None`。

5.2.6 借用任意表达式的引用

C 和 C++ 只允许你对特定种类的表达式使用 `&` 运算符，而 Rust 允许你借用任何表达式的引用：

```
fn factorial(n: usize) -> usize {
    (1..n+1).product()
}

let r = &factorial(6);
// 算术运算符可以看穿一层引用
assert_eq!(r + &1009, 1729);
```

在类似这样的场景中，Rust 简单地创建一个匿名变量来存储表达式的值，然后让引用指向它。匿名变量的生命周期依赖于你使用引用的方式：

- 如果你立刻通过一个 `let` 语句把它赋值给了一个变量（或者将它变为了结构体或数组的一部分），那么 Rust 会让匿名变量的生命周期变得和 `let` 初始化的变量一样长。在上面的例子中，Rust 将会对 `r` 引用的值进行这样的操作。
- 否则，匿名变量将只能生存到语句结束。在我们的示例中，存储 `1009` 的匿名变量只生存到 `assert_eq!` 语句的结尾处。

如果你习惯于 C 或 C++，这可能听起来很容易出错。但记住 Rust 永远不会让你写出产生悬垂引用的代码。如果引用被用在匿名变量的生命周期之外，Rust 总是会在编译期向你报告这个问题。然后你可以把被引用的值保存在一个有恰当的生命周期的命名变量中来修复代码。

5.2.7 切片和 trait 对象的引用

我们至今展示的所有引用都只是简单的地址。然而，Rust 还包括两种类型的胖指针：包含值的地址和使用值所必需的额外信息的两个字的值。

切片的引用是一种胖指针，包括切片的起始地址和它的长度。我们在[第3章](#)中详细讨论了切片。

Rust 的另一种胖指针类型是 *trait 对象*，一个实现了特定 trait 的值的引用。一个 trait 对象包含值的地址和一个指向该值对 trait 的实现的指针，用于调用 trait 的方法。我们将在[trait 对象](#)一节中详细介绍 trait 对象。

除了携带了这些额外信息之外，切片和 trait 对象的引用和我们在这一章中目前见到的所有引用一样：它们并不获取所有权，不允许比引用对象生存得更久，可能是共享或者可变的，等等。

5.3 引用安全

正如我们目前为止展示的一样，引用与 C 和 C++ 中的普通指针看起来非常像。但那些指针是不安全的，Rust 怎么能保证引用的安全呢？可能最简单的发现规则的方法就是尝试去打破规则。

为了展示最基础的思想，我们将以最简单的例子开始，展示 Rust 是怎么保证引用在单个函数体里被正确使用。然后我们会看到在函数之间传递引用和在数据结构中对它们排序。这意味着要给函数和数据类型提供生命周期参数，我们马上会解释它。最后，我们会展示一些 Rust 提供的用于简化常用模式的缩写。在这个过程中，我们会看到 Rust 是怎么指出错误代码并给出建议的解决方岸。

5.3.1 借用一个局部变量

这里有一个非常明显的例子。你不能借用一个局部变量的引用然后把它带出变量的作用域：

```
{
    let r;
    {
        let x = 1;
        r = &x;
    }
    assert_eq!(*r, 1); // bad: reads memory `x` used to occupy
}
```

Rust 的编译器会拒绝这个程序，并给出详细的错误消息：

```
error: `x` does not live long enough
--> references_dangling.rs:8.5
|
7 |         r = &x;
|         ^^^ borrowed value does not live long enough
8 |     }
|     - `x` dropped here while still borrowed
9 |     assert_eq!(*r, 1); // bad: reads memory `x` used to occupy
10| }
```

Rust 说 `x` 只生存到内层块的末尾，然而引用一直生存到外层块的末尾，因此它会变为悬垂指针，这是禁止的。

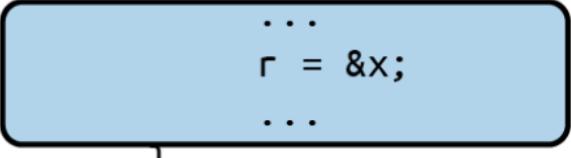
尽管对于人类来说很容易就能看出来这个程序是错误的，但 Rust 是如何得到这个结论的值得探究。每一个简单的例子都展示了 Rust 用于检查更多复杂代码的工具。

Rust 尝试给你的程序中的每一个引用赋予一个生命周期，这个生命周期要满足使用它的方式带来的约束。一个生命周期是一个引用可以安全使用的程序区间：可能是一条语句、一个表达式、也可能是一些变量的作用域，或者类似的区间。生命周期完全是 Rust 的编译期视图。在运行时引用只是一个地址，生命周期是它的类型的一部分，并没有运行时表示。

这个例子中，我们需要搞清楚三个生命周期的关系。变量 `r` 和 `x` 都有一个生命周期，从它们被初始化一直持续到编译器可以证明它们不会再被使用为止。第三个生命周期是一个引用类型的：我们从 `x` 借用并存储到 `r` 中的引用。

有一个规则应该非常明显：如果你有一个变量 `x`，那么对 `x` 的引用不能比 `x` 的生命周期更长，如图 5-3 所示。

```
{
    let r;
    {
        let x = 1;
        ...
        r = &x;
        ...
    }
    assert_eq!(*r, 1);
}
```



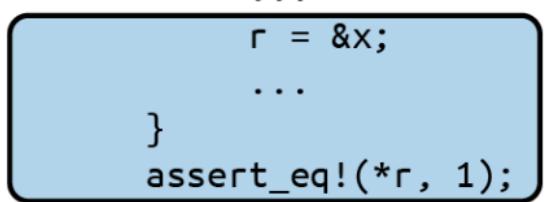
lifetime of &x must not
exceed this range

图 5-3: `&x` 的合法生命周期

当 `x` 离开作用域之后，引用将变为悬垂指针。因此我们说变量的生命周期必须包含或包括它的引用的生命周期。

这里还有另一种约束：如果你把引用存储到变量 `r` 中，引用的生命周期必须覆盖变量的整个生命周期：从它初始化到最后一次使用它，如图 5-4 所示。

```
{
    let r;
    {
        let x = 1;
        ...
        r = &x;
        ...
    }
    assert_eq!(*r, 1);
}
```



lifetime of anything stored in
r must cover at least this range

图 5-4: 存储在 `r` 中的引用的合法生命周期

如果引用不能和变量生存的一样长，那么在某些区域内指针 `r` 会变成悬垂指针。因此我们说引用的生命周期必须包含或包括存储它的变量的生命周期。

第一种约束限制了引用生命周期的上限，第二种约束限制了它的下限。Rust 简单的尝试为每个引用找到一个满足这两个约束的生命周期。然而在我们的例子中，不存在这样的生命周期，如图 5-5 所示。

```
{
    let r;
    {
        let x = 1;
        ...
        r = &x;
        ...
    }
    assert_eq!(*r, 1);
}
```

图 5-5: 一个引用的生命周期的限制互相矛盾

让我们考虑一个可以正常工作的不同的例子。我们有相同的约束：引用的生命周期必须被 `x` 的生命周期包含，但又要包含 `r` 的生命周期。但因为现在 `r` 的生命周期变小了，所以存在这样一个满足约束的生命周期，如图 5-6 所示。

```
{
    let x = 1;
    {
        let r = &x;
        ...
        assert_eq! (*r, 1);
        ...
    }
}
```

图 5-6: 一个引用的生命周期包括 `r` 的生命周期、但被 `x` 的生命周期包含

当你借用一个更大的数据结构中的一部分的引用时这些规则也自然地生效，例如引用 `vector` 的一个元素：

```
let v = vec![1, 2, 3];
let r = &v[1];
```

因为 `v` 拥有 `vector`, `vector` 又拥有它的元素, 所以 `v` 的生命周期必须包含引用 `&v[1]` 的生命周期。类似的, 如果你把引用存储在了某些数据结构之中, 那引用的生命周期必须包含这个数据结构的生命周期。例如, 如果你创建了一个引用的 `vector`, 那么每个引用的生命周期都必须包含拥有这个 `vector` 的变量的生命周期。

这是 Rust 对所有代码进行的实质的处理。如果考虑更多的语言特性——例如复杂的数据结构和函数调用, 只是会引入新的约束, 但核心原则是不变的: 首先, 理解程序使用引用的方式所带来的约束; 然后, 找到满足这些约束的生命周期。这与 C 和 C++ 程序员给自己施加的限制没有太大区别, 区别在于 Rust 自己知道这些规则并且强迫它们执行。

5.3.2 引用作为函数参数

当我们向函数传递引用时, Rust 怎么保证函数安全地使用它? 假设我们有一个函数 `f`, 这个函数接受一个引用, 然后把它存储到一个全局变量中。我们之后会进行一些修改, 不过这里有一个最初的版本:

```
// 这段代码有几个问题, 不能通过编译。
static mut STASH: &i32;
fn f(p: &i32) { STASH = p; }
```

Rust 中的全局变量叫做静态变量: 它在程序开始时就被创建, 直到程序终止时才被销毁。(类似其他的声明, Rust 的模块系统控制静态变量在哪些部分可见, 它们的“全局”只体现在生命周期上, 而不是在可见性上) 我们将在第8章中介绍静态变量, 但现在我们只列出几条代码没有遵守的规则:

- 每一个静态变量必须被初始化
- 可变的静态变量自然是线程不安全的 (因为所有线程都可以在任何时间访问一个静态变量), 即使在单线程程序中, 它们也可能成为其它类型的重入问题的牺牲品。因为这些原因, 你只能在 `unsafe` 块中使用可变的静态变量。在这个例子中我们并不关注那些特定的问题, 我们只是把它放进 `unsafe` 块中, 然后继续。

有了这些修改之后, 我们有了下面的代码:

```
static mut STASH: &i32 = &128;
fn f(p: &i32) { // 仍然不行
    unsafe {
        STASH = p;
    }
}
```

只差一步了, 为了看到剩余的问题, 我们需要写出一些 Rust 帮我们省略的东西。这里写的 `f` 的签名实际上如下签名的缩写:

```
fn f<'a>(p: &'a i32) { ... }
```

这里，生命周期'a（读作“tick A”）是f的一个生命周期参数。你可以将<'a>读作“对于任何生命周期'a”，因此当我们写fn f<'a>(p: &'a i32)时，我们定义了一个接受一个i32类型且带有任何给定生命周期'a的参数的函数。

尽管我们必须允许'a是任何生命周期，但如果它是最小的可能的生命周期的话会更好：例如恰好包含对f的调用。然后赋值语句就成了问题所在：

```
STASH = p;
```

因为STASH在程序的整个执行过程中都存在，所以它持有的引用必须有相同长度的生命周期；Rust把这种生命周期称为'static生命周期。但是p的生命周期是'a，这意味着它可以是任何包含对f的调用的生命周期。因此，Rust拒绝了我们的代码：

```
error: explicit lifetime required in the type of `p`
|
5 | fn f(p: &i32) { // 仍然不行
|     ____ help: add explicit lifetime `<static` to the type of `p`: `&'static i32`}
|         unsafe {
6 |             STASH = p;
|             ^ lifetime `<static` required
```

到这里，已经很明显了，我们的函数不能接受任意引用作为参数。但正如Rust指出的一样，它应该能接受一个有'static生命周期的引用：把这样一个引用存储到STASH不会导致悬垂指针。并且确实，下面的代码可以编译：

```
static mut STASH: &i32 = &10;

fn f(p: &'static i32) {
    unsafe {
        STASH = p;
    }
}
```

这一次，f的签名指明了p必须是一个有'static生命周期的引用，因此将它存储在STASH中没有任何问题。我们只能将f用于其他静态变量的引用，但这是唯一一种保证STASH不会变为悬垂指针的方法。因此我们可以写：

```
static WORTH_POINTING_AT: i32 = 1000;
f(&WORTH_POINTING_AT);
```

因为 `WORTH_POINTING_AT` 是一个静态变量，所以 `&WORTH_POINTING_AT` 的类型是 `&'static i32`，可以安全地传递给 `f`。

稍微后退一步，注意一下我们这种修改的方式让 `f` 的签名发生了什么变化：原本的 `f(p: &i32)` 变为了 `f(p: &'static i32)`。换句话说，如果不在函数签名中表明我们的意图，我们将不能写出一个把引用存储在全局变量中的函数。在 Rust 中，一个函数的签名总是暴露出函数体的行为。

反过来说，如果我们看到了一个函数签名例如 `g(p: &i32)`（或者写出了生命周期的 `g<'a>(p: &'a i32)`），我们将能辨别出它不会把参数 `p` 存储在此次调用之外的变量中。我们不需要查看 `g` 的定义，`g` 的签名已经告诉了我们它能做什么和不能做什么。当你想保持函数调用的安全性时，这一点会很有用。

5.3.3 向函数传递引用

现在我们已经展示了一个函数的签名怎么和它的函数体关联起来，让我们继续解释它怎么和函数的调用者联系起来。假设你有下面的代码：

```
// 这可以被更简洁地写为: fn g(p: &i32)
// 但现在让我们写出生命周期
fn g<'a>(p: &'a i32) { ... }

let x = 10;
g(&x);
```

仅仅从 `g` 的签名，Rust 就知道它不会把 `p` 存储在生命周期超过此次调用的变量中：任何包含此次调用的生命周期 '`a` 都一定能正常工作。因此 Rust 为 `&x` 选择了最小的可能的生命周期：也就是对 `g` 的调用。这满足了所有的约束：它的生命周期没有超过 `x`、包含 `g` 的整个调用。因此这段代码可以通过检查。

注意尽管 `g` 有一个生命周期参数 '`a`，我们在调用 `g` 时不需要提到它。你只需要在定义函数和类型时担心生命周期参数，当使用它们时，Rust 会为你推断生命周期。

如果我们尝试把 `&x` 传递给之前的接受静态引用参数的 `f` 函数，会发生什么呢？

```
fn f(p: &'static i32) { ... }

let x = 10;
f(&x);
```

这样会编译失败：引用 `&x` 不能生存的比 `x` 更久，但传递给 `f` 时，我们约束它至少要和 '`static` 生存的一样久。没有办法同时满足所有约束，所以 Rust 会拒绝这段代码。

5.3.4 返回引用

一种很常见的场景是一个函数接受一些数据结构的引用，然后返回一个指向这个结构中部分数据的引用。例如，这里有一个函数返回一个切片中最小的元素的引用：

```
// 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
}
```

我们按照通常的方式省略了函数签名中的生命周期参数。当一个函数接收单个引用作为参数并返回单个引用时，Rust 假设这两个引用一定有相同的生命周期。如果显式写出生命周期将是：

```
fn smallest<'a>(v: &'a [i32]) -> &'a i32 { ... }
```

假设我们像这样调用 `smallest`：

```
let s;
{
    let parabola = [9, 4, 1, 0, 1, 4, 9];
    s = smallest(&parabola);
}
assert_eq!(*s, 0); // 错误：s 指向的元素已经被 drop
```

通过 `smallest` 的签名，我们能看出它的参数和返回值必须有相同的生命周期 '`a`'。在我们的调用中，参数 `¶bola` 不能比 `parabola` 生存的久，然而 `smallest` 的返回值又至少要和 `s` 生存的一样久。不存在生命周期 '`a`' 可以同时满足这两个约束，因此 Rust 拒绝了代码：

```
error: `parabola` does not live long enough
--> references_lifetimes_propagated.rs:12:5
|
11 |         s = smallest(&parabola);
|             ----- borrow occurs here
12 |
|         ^ `parabola` dropped here while still borrowed
13 |         assert_eq!(*s, 0); // bad: points to element of dropped array
|                         - borrowed value needs to live until here
14 | }
```

移动对 s 的使用来保证它的生命周期被 parabola 的生命周期包含可以解决这个问题：

```
{
    let parabola = [9, 4, 1, 0, 1, 4, 9];
    let s = smallest(&parabola);
    assert_eq!(*s, 0); // fine: parabola still alive
}
```

函数签名中的生命周期让 Rust 能获得传进函数的引用和函数返回的引用的关系，然后保证它们都被安全地使用。

5.3.5 包含引用的结构体

Rust 怎么处理存储在数据结构中的引用？这里有一个我们之前看到的错误的程序，不过现在我们把它存在了一个结构体中：

```
// 不能通过编译
struct S {
    r: &i32
}

let s;
{
    let x = 10;
    s = S { r: &x };
}
assert_eq!(*s.r, 10); // bad: reads from dropped `x`
```

Rust 对引用安全的约束不可能因为我们把引用藏在了结构体里就神奇的消失。这些约束最终也会作用在 S 上。事实上，Rust 是很多疑的：

```
error[E0106]: missing lifetime specifier
--> references_in_struct.rs:7:12
 |
7 |     r: &i32
 |     ^ expected lifetime parameter
```

当一个引用类型出现在其他类型的定义中时，你必须写出它的生命周期。你可以这样写：

```
struct S {
    r: &'static i32
}
```

这意味着 r 只能指向生命周期等于整个程序的 i32 值，这样限制太大了，替代方案是给类型一个生命周期参数'a 并用于 r：

```
struct S<'a> {
    r: &'a i32
}
```

现在类型 S 也有了一个生命周期，就像引用类型一样。你创建的每一个类型 S 的值都有一个新的生命周期'a，它会成为你使用值的约束。你存储在 r 中的引用的生命周期需要包含'a，'a 需要包含存储这个 S 的变量的生命周期。

回到之前的代码，表达式 S { r: &x } 创建了一个生命周期为'a 的新 S。当你把 &x 存储到 r 字段时，你约束了'a 的生命周期必须被 x 的生命周期包含。

赋值语句 s = S { ... } 把 S 存储到了变量中，变量的生命周期直到示例的结尾处，'a 的生命周期需要包含 s 的生命周期。现在 Rust 遇到了和之前一样的矛盾的约束：'a 的生命周期不能超过 x，但又要至少和 s 一样长。没有满足条件的生命周期存在，所以 Rust 拒绝了这段代码。避免了灾难！

如果把带有生命周期参数的类型存储在其他类型里会发生什么？

```
struct D {
    s: S // not adequate
}
```

Rust 是多疑的，正如我们在不指明生命周期的情况下把引用存储在 S 中一样：

```
error[E0106]: missing lifetime specifier
|
8 |     s: S // not adequate
|     ^ expected named lifetime parameter
|
```

我们不能在这里省略 S 的生命周期：Rust 需要知道 D 的生命周期和 S 中的引用的生命周期的关系，这样才能对 D 进行和对 S、普通引用一样的检查。

我们可以给 s 加上'static 生命周期，就可以编译了：

```
struct D {
    s: S<'static>
}
```

有了这个定义之后，S 字段只能借用生命周期等于整个程序的值。这太严格了，它意味着 D 不能借用本地变量，D 的生命周期没有特殊的限制。

Rust 的错误信息建议了另一种更通用的解决方法：

```
help: consider introducing a named lifetime parameter
|
```

```
7 | struct D<'a> {  
8 |     s: S<'a>  
|
```

这里，我们给了 D 自己的生命周期参数并传递给 S：

```
struct D<'a> {  
    s: S<'a>  
}
```

通过接受一个生命周期参数'a 并把它用于 s 的类型，我们就可以允许 Rust 把 D 的值的生命周期关联到它持有的 S 的生命周期。

我们之前展示了函数的签名怎么暴露出它对传入的引用的操作。现在我们展示了关于类型的相似之处：一个类型的生命周期参数总是表明它是否持有一些带有有趣的（也就是非'static' 的）生命周期的引用，以及这些生命周期是什么样的。

例如，假设我们有一个解析函数，这个函数接受一个字节切片，返回一个保存解析结果的结构体：

```
fn parse_record<'i>(input: &'i [u8]) -> Record<'i> { ... }
```

根本不需要查看 Record 类型的定义，我们就可以判断出，如果我们获得了一个 parse_record 返回的 Record，那么它里面包含的引用一定指向我们传入的缓冲区内的内容，而不是别的东西（除了可能的'static' 值）。

事实上，这种内部行为的暴露是 Rust 要求持有引用的类型显式写出生命周期参数的主要原因。否则，Rust 完全可以为结构体中的每个引用标注一个不同的生命周期，让你可以省去写出它们的麻烦。早期版本的 Rust 确实是这么做的，但开发者发现这样会让人很迷惑。知道一个值什么时候从另一个值里借用了部分内容是很有用的，特别是解决 bug 时。

不只有引用和像 S 这样的类型才有生命周期。Rust 中的每个类型都有生命周期，包括 i32 和 String。大多数都是简单的'static'，意味着这些类型的值想要生存多久就能生存多久。例如，一个 Vec<i32> 是自包含的，不需要在其他任何值离开作用于之前 drop。而一个类型例如 Vec<&'a i32> 的生命周期必须被'a 包含：在它引用的值离开作用域之前它必须被 drop。

5.3.6 不同的生命周期参数

假设你定义了一个类似这样的包含两个引用的结构体：

```
struct S<'a> {  
    x: &'a i32,
```

```
y: &'a i32,
}
```

这两个引用的生命周期都是'`a`。如果你想像下面这样做的话这可能会是一个问题：

```
let x = 10;
let r;
{
    let y = 20;
    {
        let s = S { x: &x, y: &y };
        r = s.x;
    }
}
println!("{}", r);
```

这段代码不会制造任何悬垂指针。对`y`的引用存储在`s`中，`s`在`y`离开作用域之前先离开作用域。`x`的引用随着`r`结束，没有超过`x`的生命周期。

然而，如果你尝试编译这段代码，Rust 会报错说`y`生存的不够久，尽管显然它的生命周期已经够了。

Rust 为什么会报错？如果你仔细思考这段代码，你会发现以下原因：

- `s`的两个字段都有相同的生命周期'`a`，所以Rust必须找到一个生命周期同时适用于`s.x`和`s.y`。
- 我们进行了赋值`r = s.x`，这要求'`a`必须包含`r`的生命周期。
- 我们用`&y`初始化了`s.y`，这要求'`a`必须不超过`y`的生命周期。

这些约束不可能同时满足：没有生命周期比`y`的作用域短又比`r`的长。所以Rust报错了。

问题出现的原因是`S`中的两个引用有相同的生命周期'`a`。将`S`的定义修改为每个引用有一个不同的生命周期就可以解决问题：

```
struct S<'a, 'b> {
    x: &'a i32,
    y: &'b i32
}
```

有了这个定义，`s.x`和`s.y`就有了独立的生命周期。我们对`s.x`做什么都不会影响我们存储在`s.y`中的值，因此现在可以很容易地满足约束：`'a`可以直接是`r`的生命周期，`'b`是`s`的生命周期（`y`的生命周期也可以用作`'b`，但Rust尝试选择最小的满足条件的生命周期）。这样一切都没有问题了。

函数的签名也可以有相似的效果。假设我们有一个类似这样的函数：

```
fn f<'a>(r: &'a i32, s: &'a i32) -> &'a i32 { r } // perhaps too tight
```

这里，两个生命周期参数使用了相同的生命周期'a，这可能会像我们之前展示的一样对调用者施加不必要的约束。如果这导致了问题，你可以让每个参数都有独立的生命周期：

```
fn f<'a, 'b>(r: &'a i32, s: &'b i32) -> &'a i32 { r } // looser
```

这样的缺点是添加生命周期可能会导致类型和函数类型更难阅读。你们可以先尝试最简单的定义然后逐渐放松限制直到代码能够编译。因为 Rust 不允许不安全的代码，等 Rust 告诉你哪有问题然后再去修复是一个完美的策略。

5.3.7 省略生命周期参数

目前为止我们已经展示了足够多的返回引用或获取引用参数的函数，但我们通常不需要写出哪个生命周期是哪个。当生命周期很明显的时候 Rust 让我们可以省略它们。

在最简单的例子中，你可能永远都不需要写出参数的生命周期。Rust 会为每一个需要声明周期的位置分配一个生命周期。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
}
```

这个函数签名是如下的缩写：

```
fn sum_r_xy<'a, 'b, 'c>(r: &'a i32, s: S<'b, 'c>) -> i32
```

如果你返回了带有生命周期的参数的引用或其它类型，Rust 仍会尝试允许无歧义的情况下省略。如果你的函数参数中只有一个生命周期参数，那么 Rust 会假设你的返回值中的所有生命周期参数都是那一个：

```
fn first_third(point: &[i32; 3]) -> (&i32, &i32) {
    (&point[0], &point[2])
}
```

如果把所有的生命周期参数都写出来，那么等价的代码如下：

```
fn first_third<'a>(point: &'a [i32; 3]) -> (&'a i32, &'a i32)
```

如果参数里有多个生命周期参数，编译器没有简单的方法推断出返回值中的生命周期参数是哪个，此时 Rust 会要求你指明。

如果你的函数是某个类型的方法，并通过引用获取 `self` 参数，那么会打破这个限制：Rust 会假设是返回值中所有内容的生命周期都和 `self` 的生命周期相同。（Rust 中的 `self` 参数代表调用这个方法的值，等价于 C++、Java、JavaScript 中的 `this`，或者 Python 中的 `self`。我们将在[使用 impl 定义方法](#)一节中介绍方法。）

例如，你可以写如下代码：

```
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` 方法的签名是如下签名的缩写：

```
fn find_by_prefix<'a, 'b>(&'a self, prefix: &'b str) -> Option<&'a String>
```

Rust 假设如果你要借用，那么你会从 `self` 借用。

同样，这些只是缩写，用来提供帮助，并不能带来惊喜。如果它们不是你想要的，你可以显式写出生命周期。

5.4 共享 vs 可变

到目前为止，我们已经讨论了 Rust 如何保证没有引用会指向已经离开作用域的变量。但还有其它方法造成悬垂指针。这里有一个简单的例子：

```
let v = vec![4, 8, 19, 27, 34, 10];
let r = &v;
let aside = v; // 将 vector 移动到 aside
r[0];          // bad: 使用了`v`，`v` 未初始化
```

对 `aside` 的赋值会移动 `vector`, 将 `v` 设为未初始化, 因此 `r` 变成了一个悬垂指针。如图 5-7 所示。

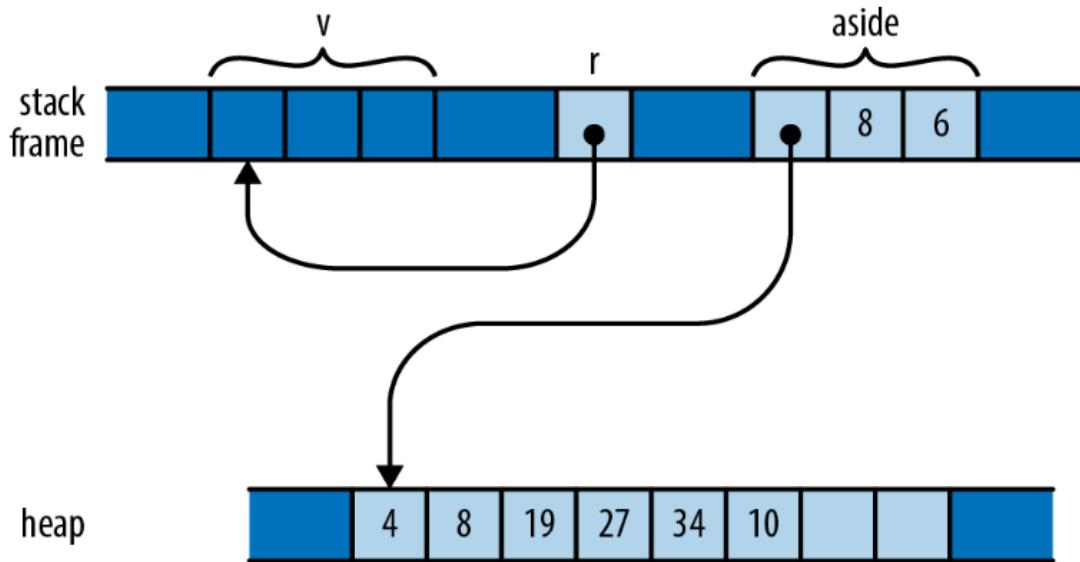


图 5-7: 一个已经被移动的 `vector` 的引用

尽管 `r` 的整个生命周期内 `v` 都还在作用域之内, 但问题是 `v` 的值被移动走了, `v` 变为了未初始化, 而 `r` 仍然指向它。自然, Rust 捕捉到了这个错误:

```
error[E0505]: cannot move out of `v` because it is borrowed
--> references_sharing_vs_mutation_1.rs:10:9
|
9 |     let r = &v;
|         _ borrow of `v` occurs here
10|     let aside = v; // move vector to aside
|         ^^^^^^ move out of `v` occurs here
```

在一个共享引用的整个生命周期内, 它会使它引用的值只读: 你不能对引用的值赋值或者移动它的值。在这段代码中, `r` 的生命周期包括尝试移动 `vector` 的语句, 因此 Rust 拒绝了这段代码。如果你按照下面这样修改, 就不会有问题是:

```
let v = vec![4, 8, 19, 27, 34, 10];
{
    let r = &v;
    r[0]; // ok: vector is still there
}
let aside = v;
```

在这个版本中, `r` 更早离开作用域, 在 `v` 被移动之前引用的生命周期就结束了, 因此没有

任何问题。

这里还有一种导致问题的方式。假设我们有一个函数用一个切片里的元素扩展一个 vector:

```
fn extend(vec: &mut Vec<f64>, slice: &[f64]) {
    for elt in slice {
        vec.push(*elt);
    }
}
```

这是标准库里 vector 的 `extend_from_slice` 的一种不够灵活（也不够优化）的版本。我们可以使用它从其他 vector 或数组的切片构建 vector:

```
let mut wave = Vec::new();
let head = vec![0.0, 1.0];
let tail = [0.0, -1.0];

extend(&mut wave, &head); // 用另一个vector扩展wave
extend(&mut wave, &tail); // 用一个数组扩展wave

assert_eq!(wave, vec![0.0, 1.0, 0.0, -1.0]);
```

这里我们构建了正弦波的一个周期。如果我们想再添加一个周期，我们可以把 vector 附加给它自己吗？

```
extend(&mut wave, &wave);
assert_eq!(wave, vec![0.0, 1.0, 0.0, -1.0,
                     0.0, 1.0, 0.0, -1.0]);
```

粗略地检查一遍，感觉这好像没有问题。但记住当我们把元素添加到 vector 中时，如果缓冲区已经满了，它必须重新分配一个更大的缓冲区。假设 `wave` 一开始有 4 个元素，因此当 `extend` 尝试添加第 5 个元素时 `wave` 必须分配更大的缓冲区。此时内存布局看起来如图 5-8。

`extend` 函数的 `vec` 参数借用了 `wave`, `wave` 已经分配了一个有 8 个元素的缓冲区，而 `slice` 仍然指向旧的 4 个元素的缓冲区，但这个缓冲区已经被 drop 了。

这种问题不是 Rust 独有的：在指向集合的同时修改集合在很多语言里都是很微妙的领域。在 C++ 中，`std::vector` 的规范提醒你 “[vector 的缓冲区的] 重新分配会导致所有其中的元素的引用、指针、迭代器无效”。类似的，Java 中有关修改 `java.util.Hashtable` 对象是这么说的：

如果创建了 `Hashtable` 的迭代器之后，`Hashtable` 被修改了，除了迭代器自身的 `remove` 方法之外，任何操作都会导致迭代器抛出一个 `ConcurrentModificationException` 异常。

导致这种 bug 特别难解决的是它并不是总是发生。在测试中，你的 vector 可能总是恰好有足够的空间，缓冲区从来不重新分配，那这个问题可能一直都不会被曝光。

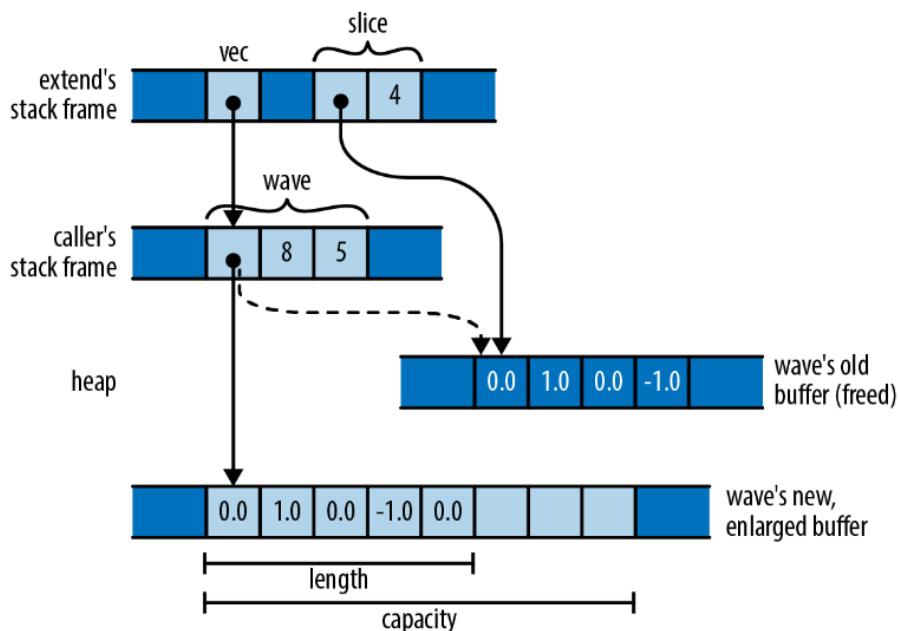


图 5-8: vector 重新分配内存导致一个切片变为悬垂指针

然而 Rust 会在编译期就汇报调用 `extend` 的问题：

```
error[E0502]: cannot borrow `wave` as immutable because it is also
borrowed as mutable
--> references_sharing_vs_mutation_2.rs:9:24
|
9 |     extend(&mut wave, &wave);
|         ^^^^ mutable borrow ends here
|         |
|             |     immutable borrow occurs here
|             mutable borrow occurs here
```

换句话说，我们可以借用 vector 的可变引用，也可以借用它的元素的共享引用，但这两种引用的生命周期不能有重叠的部分。在我们的例子中，两个引用的生命周期都包含 `extend` 的调用，所以 Rust 拒绝了这段代码。

这些错误都违背了 Rust 关于可变和共享的核心规则：

共享访问是只读的访问

共享引用借用的值是只读的。在一个共享引用的生命周期中，不管是它引用的值，还是任何通过这个值可以访问的东西，都不能被任何东西修改。如果一个结构体的所有者是只读的，将不允许指向结构体中任何内容的可变引用。它实际上被冻结了。

可变访问是独占的访问

可变引用借用的值只能通过这个引用访问。在一个可变引用的生命周期内，没有任何其他方法访问被引用的值或通过这个值可以访问到的值。唯一能与可变引用的生命周期出现重叠的引用就是那个可变引用自己。

Rust 会报告 `extend` 的例子违反了第二条规则：因为我们已经借用了 `wave` 的一个可变引用，这个可变引用必须是唯一访问 `vector` 和它的元素的方法。切片的共享引用是另一种访问元素的方式，因此违反了第二条规则。

但 Rust 也可以把我们的 bug 当成违反了第一条规则：因为我们已经获取了 `wave` 的元素的共享引用，元素和 `Vec` 自身都变为只读，所以你不能再借用一个只读值的可变引用。

每种引用都会影响我们可以对被引用的值，以及通过这个值可以访问的值进行的操作（图 5-9）。

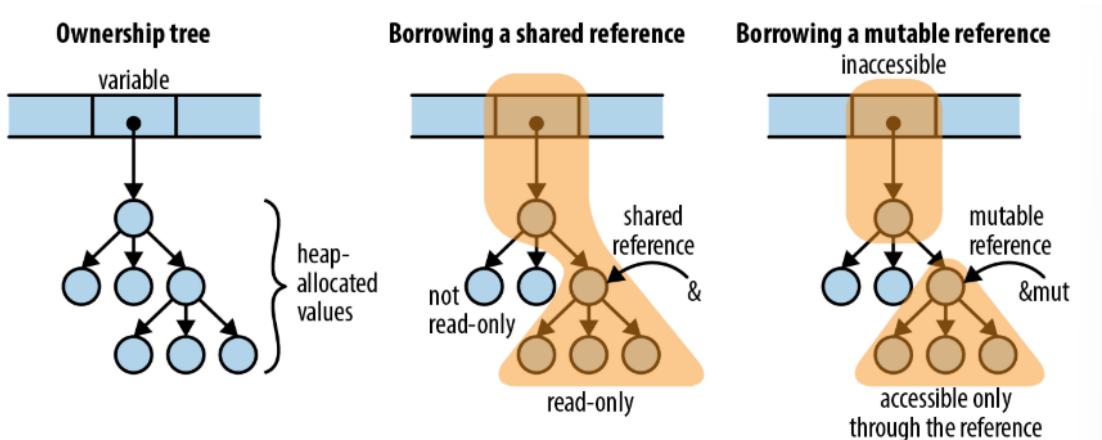


图 5-9: 借用一个引用会影响你可以对同一棵所有权树中的其他值进行的操作

注意在两种情况下，在引用的生命周期内，所有权树中通往被引用值的路径都不可能被修改。对于共享引用，这个路径是只读的；对于一个可变引用，这个路径完全不可访问。因此程序没有任何方法让一个引用变的无效。

将这些原则分解为最简单的示例：

```
let mut x = 10;
let r1 = &x;
let r2 = &x;    // ok: 允许多个共享引用
x += 10;        // error: 不能对 `x` 赋值因为它已经被借用了
let m = &mut x; // error: 不能借用 `x` 的可变引用,
                // 因为它已经被借用了不可变引用
println!("{} , {} , {}", r1, r2, m); // 引用都在这里使用
                                         // 因此它们的生命周期
```

```
// 都只到这里
```

```
let mut y = 20;
let m1 = &mut y;
let m2 = &mut y;      // error: 不能再次借用可变引用
let z = y;           // error: 不能使用`y`，因为它被借用了可变引用
println!("{} , {} , {}", m1, m2, z); // 引用都在这里使用
```

可以从一个共享引用借用一个共享引用：

```
let mut w = (107, 109);
let r = &w;
let r0 = &r.0;          // ok; 从共享引用再次借用共享引用
let m1 = &mut r.1;      // error: 不能从共享引用借用可变引用
println!("{}" , r0);    // r0 在这里使用
```

也可以再次借用一个可变引用：

```
let mut v = (136, 139);
let m = &mut v;
let m0 = &mut m.0;      // ok: 从可变引用重新借用可变引用
*m0 = 137;
let r1 = &m.1;          // ok: 从可变引用重新借用共享引用
                        // 并且和m0没有重叠部分
v.1;                  // error: 通过其他路径访问仍然是禁止的
println!("{}" , r1);    // r1 在这里使用
```

这些限制非常严格。回到我们调用 `extend(&mut wave, &wave)` 的地方，没有方便快速的方法修复代码让它按我们希望的方式运行。Rust 在几乎所有地方应用这些规则：如果我们借用了 `HashMap` 里的一个键的共享引用，我们将不能借用整个 `HashMap` 的可变引用，直到共享引用的生命周期结束。

但有足够的理由这么做：设计一个支持不严格的、可以同时迭代和修改的集合是非常困难的，并且通常没有简单高效的实现。Java 的 `Hashtable` 和 C++ 的 `vector` 并不考虑这些、Python 的字典、JavaScript 的对象也没有定义这种情况下的行为。JavaScript 中的其它集合类型做到了这一点，但也导致了更加笨重的实现。C++ 的 `std::map` 承诺插入新的条目不会使指向 map 中其他条目的指针失效，但为了实现这个承诺，标准中排除了类似 Rust 中的 `BTreeMap` 的更加缓存高效的实现，这种实现在树中的每个节点存储多个条目。

这里有另一个这些规则可以捕获的 bug 的例子。考虑下面的 C++ 代码，用于管理一个文件描述符。为了保持简单，我们只写出构造函数和一个拷贝赋值运算符，并且省略错误处理：

```
struct File {
    int descriptor;
```

```

File(int d) : descriptor(d) {}

File& operator=(const File &rhs) {
    close(descriptor);
    descriptor = dup(rhs.descriptor);
    return *this;
}
}

```

赋值运算符足够简单，但在像这样的特殊情况下会出现问题：

```

File f(open("foo.txt", ...));
...
f = f;

```

如果们把一个 `File` 赋给它自己，那么 `rhs` 和 `*this` 是同一个对象，因此 `operator=` 会关闭它准备 `dup` 的文件描述符。我们销毁了想要拷贝的资源。

在 Rust 中，类似的代码将是：

```

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 中的常用写法。Rust 有优秀的方法给 Rust 中的类型添加构造函数和方法，我们将在第 9 章介绍，但上面的定义对这个例子来说已经足够了。)

如果我们写出与之前使用 `File` 相对应的 Rust 代码，我们有：

```

let mut f = new_file(open("foo.txt", ...));
...
clone_from(&mut f, &f);

```

当然，Rust 拒绝编译代码：

```
error[E0502]: cannot borrow `f` as immutable because it is also
borrowed as mutable
--> references_self_assignment.rs:18:25
|
18 |     clone_from(&mut f, &f);
|             ^-- mutable borrow ends here
|             | |
|             | immutable borrow occurs here
|             mutable borrow occurs here
```

这看起来很熟悉。它暴露出了两种典型的 C++ bug——无法处理自我赋值以及使用无效的迭代器——两者实际上是同一种底层类型的 bug！在这两种情况下，代码都假设它在修改一个值的同时访问另一个值，然而实际上它们是同一个值。如果你曾经偶然在 C 和 C++ 中让 `memcpy` 或 `strcpy` 的调用的源区间和目标区间重叠，那么将会产生另一种形式的 bug。通过要求可变引用必须独占，Rust 可以避免各种各样的日常错误。

共享和可变引用的互斥实际上也展示了编写并发代码时值的状态。只有当值在某些线程中既可写又共享时才可能发生数据竞争——Rust 的引用规则恰好可以避免这种情况。没有 `unsafe` 代码的并发 Rust 程序可以从构造上完全避免数据竞争。我们将在[第 19 章](#)中详细介绍更多有关并发的细节。但总的来说，在 Rust 中使用并发要比大多数其他语言轻松得多。

Rust 的共享引用 vs C 的常量指针

乍一看，Rust 中的共享引用很像 C 和 C++ 中指向 `const` 值的指针。然而，Rust 对共享引用的限制要更严格。例如考虑下面的 C 代码：

```
int x = 42;           // int 变量, 不是 const
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” 的 int 值已经被修改了
```

`p` 是 `const int *` 的事实意味着你不能通过 `p` 自身来改变它指向的值：`(*p)++` 是禁止的。但你可以直接通过 `x` 访问被引用的值，而 `x` 并不是 `const`，所以你可以直接修改它的值。C 家族的 `const` 关键字有它的作用，但它并不能把指向的值变为常量。

在 Rust 中，一个共享引用会禁止对被引用值的任何修改，直到引用的生命周期结束：

```
let mut x = 42;        // 非常量 i32 值
let p = &x;            // i32 的共享引用
assert_eq!(*p, 42);
x += 1;               // 错误：不能对 x 赋值因为它已经被借用了
assert_eq!(*p, 42);    // 如果删除赋值语句，断言将为真
```

为了确保值是常量，我们需要持续追踪所有可以访问到这个值的路径，并确保它们都不允许修改操作或者直接不可用。C 和 C++ 的编译器对指针的检查太过宽松。Rust 的引用总是关联到一个特定的生命周期，这使得在编译期检查它们变得可行。

5.5 拿起武器对抗对象之海

自从 19 世纪 90 年代起自动内存管理兴起之后，所有程序的默认架构都变成了对象之海 (*sea of objects*)，如图 5-10 所示。

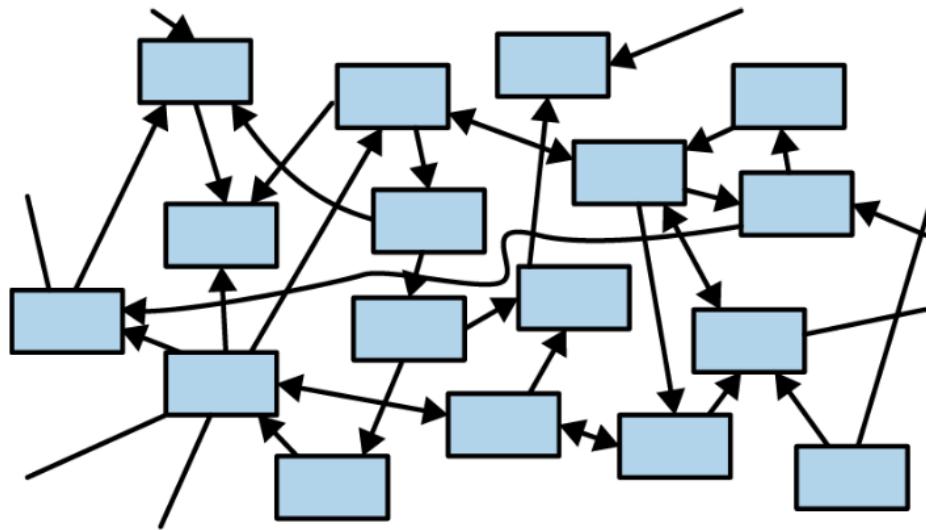


图 5-10: 一个对象之海

这就是如果你不假思索就开始编写程序并且有垃圾回收时的真实情景。我们都在构建这样的系统。

这种架构有很多图中没有显示出的优势：初始化很快、很容易破译里面的逻辑、而且即使几年之后你也可以轻易的完全重写它们。（Cue AC/DC's “Highway to Hell.”（通往地狱的高速公路））。

当然，它也有很多缺陷。当每个东西都依赖别的任何东西的时候，将非常难以测试、推演、甚至单独考虑其中的任何一部分组件。

Rust 迷人的一点就是 Rust 在通往地狱的高速公路上放了一条减速带。在 Rust 中需要花费大量的努力才能制造出环形引用——两个值互相包含指向彼此的引用。你必须使用智能指针例如 Rc 和 内部可变性——一个我们还没有讲到的话题。Rust 更喜欢指针、所有权、数据流朝一个方向流经整个系统，如图 5-11 所示。

我们现在提出这一点的原因是，在阅读完这一章之后，你可能会很自然的想通过 Rc 智能指针创造一片“结构体的海洋”，然后完全重现你熟悉的面向对象的模式。你不能立刻就完成这些，因为 Rust 的所有权模型会给你制造一些麻烦。解决方法是事先进行设计并构建更好的程序。

Rust 就是把理解你的程序的痛苦从未来转移到了现在。它做的非常好：它不仅强迫你理解

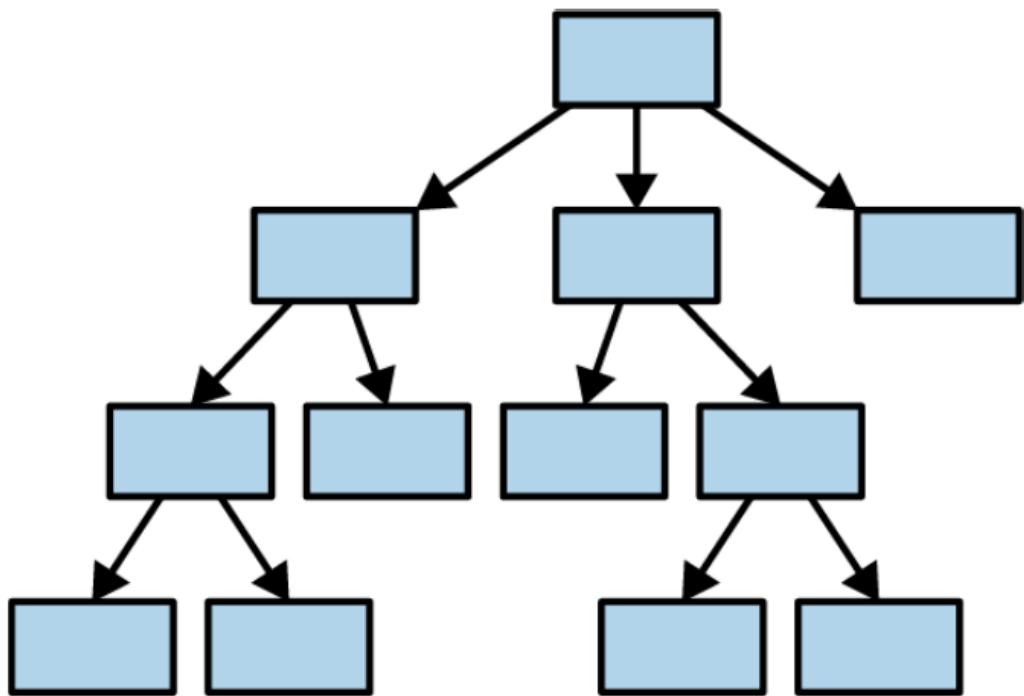


图 5-11: 一棵值组成的树

为什么你的程序是线程安全的，它甚至还能要求一些高层级的架构设计。

Chapter 6

表达式

LISP programmers know the value of everything, but the cost of nothing

——Alan Perlis, epigram #55

在本章中，我们将介绍 Rust 的表达式，它是构成 Rust 函数体和大部分 Rust 代码的构建块。本章将探索表达式的力量以及如何克服它的局限。我们还将介绍控制流，它在 Rust 中完全是以表达式为基础的，最后还要介绍 Rust 中的基本运算符如何单独和组合工作。

还有一些从技术角度应该划入这一类的概念，例如闭包和迭代器。但它们太过重要因此我们之后会用单独的章节介绍它们。现在，我们希望能用尽可能少的页数介绍尽可能多的语法。

6.1 表达式语言

Rust 表面上看上去像 C 家族的语言，但这其实是一个误解。在 C 语言中，表达式和语句之间有很大的不同。表达式是一些像这样的代码：

```
5 * (fahr-32) / 9
```

而语句则是像这样的：

```
for (; begin != end; ++begin) {  
    if (*begin == target)  
        break;  
}
```

表达式有值，但语句没有。

Rust 是一种表达式语言。这意味着它遵循了起源于 Lisp 的传统，即表达式负责完成所有工作。

在C中，`if`和`switch`是语句。它们并不产生值，也不能被用在表达式中间。在Rust中，`if`和`match`可以产生值。我们已经在第2章中看到过一个产生数字值的`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
    };
```

一个`if`表达式可以用于初始化一个变量：

```
let status =
    if cpu.temperature <= MAX_TEMP {
        HttpStatus::Ok
    } else {
        HttpStatus::ServerError // server melted
    };
```

一个`match`表达式可以被用作函数或宏的参数：

```
println!("Inside the vat, you see {}.", 
    match vat.contents {
        Some(brain) => brain.desc(),
        None => "nothing of interest"
    });
});
```

这解释了Rust为什么没有C的三元运算符(`expr1 ? expr2 : expr3`)。在C中，它是一种类似`if`语句的表达式。但在Rust中这种写法是多余的，因为`if`表达式可以同时实现这两种功能。

C中的大部分控制流工具都是语句，而Rust中的控制流则全是表达式。

6.2 优先级与结合性

表6-1总结了Rust的表达式语法。我们将在这一章中介绍所有这些表达式。运算符按照优先级从高到低的顺序列出。(类似于大多数编程语言，Rust使用运算符优先级来决定当表达式中含有多个运算符时的运算顺序。例如，在表达式`limit < 2 * broom.size + 1`中，`.`运算符优先级最高，因此会先访问字段。)

表 6-1: 表达式

表达式类型	示例	相关 trait
数组字面量	[1, 2, 3]	
重复数组字面量	[0; 50]	
元组	(6, "crullers")	
组合	(2 + 2)	
块	{ f(); g() }	
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
方法调用	point.translate(50, 50)	Deref, DerefMut
函数调用	stdin()	Fn(Arg0, ...) -> T, FnMut(Arg0, ...) -> T, FnOnce(Arg0, ...) -> T
索引	arr[0]	Index, IndexMut, Deref, DerefMut
错误检查	create_dir("tmp")?	

逻辑/位 NOT	<code>!ok</code>	Not
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>	
乘法	<code>n * 2</code>	Mul
除法	<code>n / 2</code>	Div
余数 (取模)	<code>n % 2</code>	Rem
加法	<code>n + 1</code>	Add
减法	<code>n - 1</code>	Sub
左移	<code>n << 1</code>	Shl
右移	<code>n >> 1</code>	Shr
位与	<code>n & 1</code>	BitAnd
位异或	<code>n ^ 1</code>	BitXor
位或	<code>n 1</code>	BitOr
小于	<code>n < 1</code>	<code>std::cmp::PartialOrd</code>
小于等于	<code>n <= 1</code>	<code>std::cmp::PartialOrd</code>
大于	<code>n > 1</code>	<code>std::cmp::PartialOrd</code>
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>	
左闭右开区间	<code>start .. stop</code>	
左闭右闭区间	<code>start ..= stop</code>	
赋值	<code>x = val</code>	
复合赋值	<code>x *= 1</code>	MulAssign
	<code>x /= 1</code>	DivAssign
	<code>x %= 1</code>	RemAssign
	<code>x += 1</code>	AddAssign
	<code>x -= 1</code>	SubAssign
	<code>x <= 1</code>	ShlAssign

<code>x >= 1</code>	<code>ShrAssign</code>
<code>x &= 1</code>	<code>BitAndAssign</code>
<code>x ^= 1</code>	<code>BitXorAssign</code>
<code>x = 1</code>	<code>BitOrAssign</code>
闭包	<code> x, y x + y</code>

所有可以链式使用的运算符都是左结合的。也就是说，一条运算链例如 `a - b - c` 被组合为 `(a - b) - c`，而不是 `a - (b - c)`。这些运算符可以被任意组合：

```
* / % + - << >> & ^ | && || as
```

比较运算符、赋值运算符、范围运算符`..`和`..=`不能被链式使用。

6.3 块和分号

块是最通用的表达式。一个块产生一个值，可以被用于任何需要一个值的地方：

```
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) =>`之后的代码是简单的表达式 `author.name()`，`None =>`之后的代码则是一个块表达式。对 Rust 来说，两种表达式没有区别。块表达式的值是它的最后一条表达式的值，也就是 `ip.to_string()`。

注意 `ip.to_string()` 后面没有分号。Rust 中的大部分代码行都以分号或者花括号结尾，类似于 C 和 Java。如果一个块看起来像 C 代码一样在所有的表达式后边都有分号，那它的行为就和 C 块一样，它的值将是`()`。正如我们在第 2 章提到的，当你省略了块中最后一个表达式后边的分号，那么块的值将是最后一个表达式的值，而不是通常的`()`。

在一些语言中，尤其是 Javascript，你可以省略分号，语言会自动为你添加上——这样除了方便一点，并没有任何区别。然而在 Rust 中，分号通常是有实际意义的：

```
let msg = {
    // let 语句：总是需要分号
    let dandelion_control = puffball.open();
```

```
// 表达式 + 分号: 方法被调用, 返回值被丢弃
dandelion_control.release_all_seeds(launch_codes);

// 没有分号的表达式: 方法被调用,
// 返回值被存储到 `msg`
dandelion_control.get_status()

};
```

语句块可以包含声明最后还能产生一个值的能力是一个很有用的特性，而且可以很快习惯。它的一个缺陷是如果你偶然忘记了分号会导致一条错误信息：

```
...
if preferences.changed() {
    page.compute_size() // oops, 缺少分号
}
```

如果你在 C 或者 Java 程序中犯了这种错误，编译器会简单地直接指出你少写了一个分号。然而这是 Rust 的报错：

```
error[E0308]: mismatched types
22 |     page.compute_size() // oops, missing semicolon
|     ^^^^^^^^^^^^^^^^^^- help: try adding a semicolon `;`
|     |
|     expected (), found tuple
|
= note: expected unit type `()`
       found tuple `(u32, u32)`
```

在缺少分号的情况下，块的值将是 `page.compute_size()` 返回的值，但一个没有 `else` 分支的 `if` 语句必须总是返回 `()`。幸运的是，Rust 知道这种类型的错误并建议加上分号。

6.4 声明

除了表达式和分号之外，一个块中可能包含任意数量的声明。最常见的情况是 `let` 声明，它用来声明局部变量：

```
let name: type = expr;
```

类型和初始值是可选的，分号是必须的。

一个 `let` 声明可以在不初始化的情况下声明一个变量。这有时很有用，因为有时候一个变量需要在控制流的中途初始化：

```
let name;
if user.hasNickname() {
    name = user.nickname();
} else {
    name = generateUniqueName();
    user.register(&name);
}
```

局部变量 `name` 有两种不同的初始化路径，但两条路径上它都只会被初始化一次，所以 `name` 不需要声明为 `mut`。

在变量初始化之前使用它会导致错误（这和使用被 move 的值的错误紧密相关，Rust 希望你只在变量的值存在时使用它们！）。

你有时可能会看到代码似乎重新声明一个已经存在的变量，例如：

```
for line in file.lines() {
    let line = line?;
    ...
}
```

`let` 声明创建了一个新的、类型不同的、第二个变量 `line`。第一个 `line` 的类型是 `Result<String, io::Error>`，第二个 `line` 是一个 `String`。第二个声明在块的剩余部分会取代第一个。这被称为遮蔽 (*shadowing*)，在 Rust 程序中非常常见。上面的代码等价于：

```
for line_result in file.lines() {
    let line = line_result?;
    ...
}
```

在本书中，我们将坚持在这种场景中使用 `_result` 后缀，来保证变量的名字不同。

一个块还可以包含 *item declarations*。一个 item 是一个可以出现在全局或模块中的声明，例如 `fn`、`struct`、`use`。

后面的章节将会详细介绍 item。现在，`fn` 足够作为一个例子了。任何块都可以包含 `fn` 声明：

```
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)) // 比较路径
    }

    v.sort_by(cmp_by_timestamp_then_name);
    ...
}

```

当一个 fn 在块内声明的时候，它的作用域是整个块，它可以在整个块内使用。但是一个嵌套的 fn 不能访问外围作用域的局部变量和参数。例如，函数 cmp_by_timestamp_then_name 不能使用 v。（Rust 还有闭包，闭包可以使用外层作用域的变量，见第 14 章。）

一个块甚至可以包含整个模块。这听起来可能有些多余了：我们真的需要把语言的每一部分嵌套在其他部分中的能力吗？——但程序员（尤其是使用宏的程序员）可以找到语言提供的每一个正交碎片的用法。

6.5 if 与 match

if 表达式的形式大家都很熟悉：

```

if condition1 {
    block1
} else if condition2 {
    block2
} else {
    block_n
}

```

每一个 condition 都必须是 bool 类型的表达式，Rust 不会隐式地将数字或者指针类型转换为布尔值。

和 C 不同的是条件表达式不需要括号，事实上如果有不必要的括号的话，rustc 会发出警告。不过花括号是必须的。

else if 块，和最后的 else 都是可选的。一个没有 else 块的 if 表达式类似于一个 else 块为空的 if 表达式。

match 表达式有些类似于 C 的 switch 语句，但是更加灵活。一个简单的例子如下：

```

match code {
    0 => println!("OK"),
    1 => println!("Wires Tangled"),
    2 => println!("User Asleep"),
}

```

```

    _ => println!("Unrecognized Error {}", code)
}

```

这类似于一个 `switch` 语句，`match` 表达式的四条分支里只有一条会执行，取决于 `code` 的值。通配符模式 `_` 可以匹配任何情况，类似于 `switch` 语句中的 `default:` 标签，不过它会覆盖之后的所有模式，它之后的模式将永远不会匹配到任何东西（出现这种情况时编译器也会警告你）。

编译器可以使用跳转表来优化这种 `match` 表达式，类似于 C++ 中的 `switch` 语句。当 `match` 的每个分支都返回常量值时还会有一个类似的优化，这种情况下，编译器会用那些值构建一个数组，然后 `match` 会被编译为一次数组访问，这种情况下除了边界检查之外，编译出的代码将不会有任何条件分支。

每个分支中 `=>` 左侧支持多种模式，这是 `match` 功能强大的根源。上边的例子中，每种模式都只是一个简单的整数。我们还展示过区分 `Option` 的两种值的 `match` 表达式：

```

match params.get("name") {
    Some(name) => println!("Hello, {}!", name);
    None => println!("Greetings, stranger.");
}

```

这只是模式的一个小应用，一个模式可以匹配很多值。它可以解包元组，可以匹配结构体中的每个字段，可以解引用，借用一个值的一部分，等等。Rust 的模式是一种专门的 mini 语言。我们将在第 10 章中介绍它们。

`match` 表达式的通用形式是：

```

match value {
    pattern => expr,
    ...
}

```

如果 `expr` 是一个块的话，分支最后的逗号可以省略。

Rust 会从第一个分支开始，逐个检查 `value` 和给定的模式是否匹配。当有一个模式匹配时，相应的 `expr` 将会被求值，整个 `match` 表达式将完成执行，不会再检查别的模式。必须至少有一个模式可以匹配，Rust 会禁止没有覆盖所有可能情况的 `match` 表达式：

```

let score = match card.rank {
    Jack => 10,
    Queen => 10,
    Ace => 11
}; // 错误：没有穷尽所有模式

```

`if` 表达式的所有块必须产生相同类型的值：

```
let suggested_pet =
  if with_wings { Pet::Buzzard } else { Pet::Hyena }; // ok
let favorite_number =
  if user.is_hobbit() { "eleventy-one" } else { 9 }; // error
let best_sports_team =
  if is_hockey_season() { "Predators" }; // error
```

(最后一个例子会导致错误，因为在7月结果将是()。)¹

类似的，`match`表达式的分支也必须有相同的类型：

```
let suggested_pet =
  match favorite.element {
    Fire => Pet::RedPanda,
    Air => Pet::Buffalo,
    Water => Pet::Orca,
    _ => None // 错误：类型不一致
  }
```

6.6 if let

`if`表达式还有一种形式：`if let`表达式：

```
if let pattern = expr {
  block1
} else {
  block2
}
```

`expr`如果能匹配`pattern`，则执行`block1`，如果不能匹配，则执行`block2`。有时这是一种获取`Option`或`Result`的值的好方法：

```
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();
}
```

¹译者注：7月不是曲棍球赛季？

没有任何场景必须使用 `if let`, 因为 `match` 可以做到任何 `if let` 可以做的事。`if let` 表达式类似于如下只有一个模式的 `match` 表达式的缩写:

```
match expr {  
    pattern => { block1 }  
    _ => { block2 }  
}
```

6.7 循环

有四种循环表达式:

```
while condition {  
    block  
}  
  
while let pattern = expr {  
    block  
}  
  
loop {  
    block  
}  
  
for pattern in iterable {  
    block  
}
```

在 Rust 中循环也是表达式, 不过 `while` 或 `for` 循环的值总是 `()`, 因此它们的值并没有用。如果你指明的话 `loop` 表达式可以产生一个值。

`while` 循环的行为和 C 语言中的基本一样, 除了 `condition` 必须是精确的 `bool` 类型的表达式。

`while let` 循环类似于 `if let`。每一次迭代循环开始时, 如果 `expr` 的值可以匹配 `pattern`, 将会运行表达式块, 否则循环将结束。

使用 `loop` 循环来编写无限循环。它会永远重复执行 `block` (直到遇到 `break` 或 `return` 或线程 `panic`)。

一个 `for` 循环会先求值 `iterable` 表达式, 然后对表达式返回的迭代器产生的每个值执行一次 `block`。很多类型都可以被迭代, 包括所有标准集合例如 `Vec` 和 `HashMap`。标准的 C 语言中的 `for` 循环:

```
for (int i = 0; i < 20; i++) {
    printf("%d\n", i);
}
```

用 Rust 来写的话就是：

```
for i in 0..20 {
    println!("{}", i);
}
```

和 C 语言一样，最后一个打印出的数字是 19。

`..` 运算符会产生一个范围，它是一个只有两个字段 `start` 和 `end` 的简单结构体。`0..20` 等价于 `std::ops::Range { start: 0, end: 20 }`。范围可以用于 `for` 循环，因为 `Range` 是可以迭代的类型，它实现了 `std::iter::IntoIterator` trait，我们将在[第 15 章](#)中讨论这些。标准集合都是可迭代的，数组和切片也是。

为了保持 Rust 中的移动语义，`for` 循环会消耗掉值：

```
let strings: Vec<String> = error_messages();
for s in strings {           // 每一个 String 被移动进 s
    println!("{}", s);
}                           // 并在这里 drop
println!("{} error(s)", strings.len()); // 错误：使用了被 move 的值
```

这样可能会很不方便，一个简单的方法是让循环获取集合的引用。然后循环变量将会变成集合中每一个元素的引用：

```
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`。

迭代一个 `mut` 引用会产生每个元素的 `mut` 引用：

```
for rs in &mut strings {    // rs 的类型是 &mut String
    rs.push('\n'); // 每个字符串添加一个换行符
}
```

[第 15 章](#)会更详细的介绍 `for` 循环并展示其他使用迭代器的方式。

6.8 循环中的控制流

`break` 表达式用来跳出循环（Rust 中 `break` 只能在循环中使用。`match` 表达式中不需要它，这一点和 `switch` 语句不同）。

在一个 `loop` 循环体内，你可以给 `break` 一个表达式，表达式的值就是整个循环的值：

```
// 每一次对`next_line`的调用返回`Some(line)`，其中`line`  
// 是输入的一行；或者返回`None`，表示已经到达输入结尾。  
// 返回第一个以"answer: "开头的行。  
// 如果没有，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` 表达式中的所有 `break` 表达式必须产生相同类型的值，这个类型也会成为 `loop` 本身的类型。

一个 `continue` 表达式跳转到下一次迭代：

```
// 读取一些数据，一次读取一行  
for line in input_lines {  
    let trimmed = trim_comments_and_whitespace(line);  
    if trimmed.is_empty() {  
        // 跳转到loop的开头并  
        // 移动到下一行输入  
        continue;  
    }  
    ...  
}
```

在一个 `for` 循环中，`continue` 会跳转到集合中的下一个值，如果没有值了，循环会停止。类似的，在一个 `while` 循环中 `continue` 会重新检查循环条件，如果现在是 `false`，循环会停止。

一个循环可以用一个生命周期标记。在下面的例子中，`'search'` 是一个外层 `for` 循环的标签。因此 `break 'search'` 会退出外层循环，而不是内层循环：

```
'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;  
        }  
    }  
}
```

```

    }
}

}

```

一个 `break` 可以同时带有一个标签和一个值表达式：

```

// 寻找一列数中的第一个完全平方
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，尝试下一个数
            break;
        }
    }
};


```

标签也可以和 `continue` 一起使用。

6.9 `return` 表达式

`return` 表达式退出当前函数，向调用者返回一个值。

没有值的 `return` 相当于 `return ()`：

```

fn f() {    // 返回类型被省略：默认是()
    return; // 返回值被省略：默认是()
}

```

函数并不一定需要显式的 `return` 表达式。函数体就像是一个块表达式：如果最后一个表达式后边没有分号，它的值将是函数的返回值。事实上，这是在 Rust 中充当返回值的最佳方法。

但这并不意味着 `return` 就毫无作用，或者仅仅是为了符合不熟悉表达式语言的用户的习惯。类似于 `break` 表达式，`return` 可以终止当前的工作。例如，在第 2 章中，在调用了一个可能会失败的函数之后，我们使用了 `? 运算符` 来检查错误：

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

我们解释过这是 `match` 表达式的缩写：

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

这段代码首先调用 `File::create(filename)`。如果返回 `Ok(f)`，那么整个 `match` 表达式将求值为 `f`，因此 `f` 被存储在 `output` 中，然后将继续执行 `match` 之后的代码。

否则，我们会匹配到 `Err(err)`，然后触发 `return` 表达式。此时我们正在求值一个 `match` 表达式，来决定变量 `output` 的值。但这都无所谓，我们会放弃所有任务，直接退出函数，并返回我们从 `File::create()` 得到的错误。

我们将在 [传播错误](#) 一节中详细介绍 `? 运算符`。

6.10 Rust 为什么会有 loop 循环

Rust 编译器的几个部分会检查整个程序中的控制流：

- Rust 会检查一个函数里的所有返回路径是否返回相同类型的值。为了做到这一点，它需要知道控制流是否能到达函数结尾。
- Rust 会检查未初始化的局部变量是否绝不会被使用。这需要检查程序中的每一条路径来确保没有路径会到达未初始化的变量被使用的情况。
- Rust 会警告不可达的代码。函数中没有路径可以到达的代码就是不可达的代码。

这些被称为控制流敏感 (*flow-sensitive*) 分析。这并不是新的概念，Java 很多年前就有一个“确定性赋值 (definite assignment)” 的分析，和 Rust 的检查很相似。

当强迫执行这种规则时，一门语言（的编译器）必须在简洁和智能之间权衡。简洁可以让程序员更容易地理解编译器在说什么；智能则可以帮助消除错误警告和程序完全正确但编译器却报错的情况。Rust 倾向于简洁，它的控制流敏感分析根本就不检查循环条件，而是假设程序中的所有条件都既可能是真又可能是假。

这导致 Rust 会拒绝编译这样的安全程序：

```
fn wait_for_process(process: &mut Process) -> i32 {
    while true {
        if process.wait() {
            return process.exit_code();
        }
    }
} // 错误：不匹配的类型：期望 i32，得到()
```

这里的错误是假的。函数只可能通过 `return` 表达式返回，因此 `while` 循环不会产生 `i32` 值是无关紧要的。

而 `loop` 表达式作为一种“告诉编译器你的意思”的解决方案来解决这个问题。

Rust 的类型系统也会被控制流影响。之前我们说过 `if` 表达式中的所有分支都必须有相同的类型。但如果在以 `break`、`return`、`loop` 表达式或对 `panic!()`、`std::process::exit()` 的调用作为结尾的块中强迫这个规则就会显得很愚蠢。这些表达式的共同点就是它们永远不会像正常的方式一样产生一个值：`break` 或 `return` 会中断并退出当前的块、无限的 `loop` 循环则永远不会结束，等等。

因此在 Rust 中，这些表达式并没有通常的类型。不会正常结束的表达式会被赋予特殊类型 `!`，并且它们不受类型必须匹配的规则的约束。你可以在 `std::process::exit()` 的函数签名中看到 `!`：

```
fn exit(code: i32) -> !
```

`!` 意味着 `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，在某些情况下这会显得非常自然：

```
fn serve_forever(socket: ServerSocket, handler: ServerHandler) -> ! {
    socket.listen();
    loop {
        let s = socket.accept();
        handler.handle(s);
    }
}
```

当然，如果函数能正常返回的话，Rust 会认为这是一个错误。

有了这些大规模控制流的构建块之后，我们可以继续分析常用的更细粒度的表达式，例如函数调用和算术运算。

6.11 函数和方法调用

在 Rust 中调用函数的方法的语法和许多其它语言一样：

```
let x = gcd(1302, 462);           // 函数调用
let room = player.location();    // 方法调用
```

在第二个例子中，`player` 是一个编造的 `Player` 类型的变量，它有一个 `.location()` 方法。（我们将在第 9 章中讨论自定义类型时展示如何定义自己的方法。）

Rust 通常会严格区分引用和它指向的值。如果你向接收 `i32` 的函数传递 `&i32`，将会引发类型错误。不过你会注意到 `.` 运算符放宽了这个限制，在方法调用 `player.location()` 中，`player` 可以是一个 `Player`、也可以是一个引用类型 `&Player`、也可以是智能指针类型

Box<Player>或者Rc<Player>。.location()方法可能会以值也可能会以引用获取参数，同样的.location()语法可以适用于所有情况，因为Rust的.运算符会根据需要自动解引用player或者获取它的引用。

第三种语法用于调用类型关联函数，例如Vec::new():

```
let mut numbers = Vec::new(); // 类型关联函数调用
```

这类似于面向对象语言中的静态方法，方法在值上调用（例如my_vec.len()），而类型关联函数在类型上调用（例如Vec::new()）。

当然，方法调用可以被串联起来：

```
// 取自于第二章的Actix-based web server
server
    .bind("127.0.0.1:3000").expect("error binding server to address")
    .run().expect("error running server");
```

Rust语法中的一个怪异之处是，不能按照泛型类型的通常语法例如Vec<T>来调用函数或者方法：

```
return Vec<i32>::with_capacity(1000); // error: something about chained comparisons
let ramp = (0 .. n).collect<Vec<i32>>(); // 同样的错误
```

问题在于这个表达式中的<被当作小于运算符。Rust编译器会建议使用::<T>来代替<T>，这样可以解决这个问题：

```
return Vec::<i32>::with_capacity(1000); // ok, using ::<
let ramp = (0 .. n).collect::<Vec<i32>>(); // ok, using ::<
```

符号::<...>在Rust中被亲切地称为*turbofish*。

另外，通常也可以省略类型参数，让Rust推断它们：

```
return Vec::with_capacity(10); // ok, 如果函数的返回类型是Vec<i32>
let ramp: Vec<i32> = (0 .. n).collect(); // ok, 指定了变量的类型
```

在可以推断出类型时省略类型是一种好的风格。

6.12 字段和元素

访问结构体中字段的语法大家都很熟悉，元组也一样，不过元组中的字段是数字而不是名称：

```
game.black_pawns    // 结构体字段
coords.1            // 元组元素
```

如果点左边的值是引用或者智能指针类型，它也会像方法调用一样自动解引用。

方括号可以访问数组、切片或 vector 中的元素：

```
pieces[i]          // 数组元素
```

方括号左侧的值也会自动解引用。

类似于这三种的表达式被称为左值，因为它们可以出现在赋值表达式的左边：

```
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 被声明为 mut 变量时才可以这么做。

从一个数组或者 vector 中提取切片非常直观：

```
let second_half = &game_moves[midpoint .. end];
```

这里 game_moves 可能是一个数组、切片或者是 vector，结果将是一个长度为 end - midpoint 的切片，在 second_half 的生命周期内 game_moves 都处于被借用的状态。

.. 运算符允许任意一边的操作数被省略，根据两边是否有操作数，它可以被划分为四种形式：

```
..      // 全部范围
a ..    // 起点范围 { start: a }
.. b    // 终点范围 { end: b }
a .. b // 范围 { start: a, end: b }
```

后两种形式是 *end-exclusive*(右开区间)：终点值将不会被包含。例如，范围 0 .. 3 包括数字 0、1、2。

..= 运算符产生 *end-inclusive*(闭区间) 范围，它会包含终点值：

```
.. = b      // RangeToInclusive { end: b }
a .. = b    // RangeInclusive::new(a, b)
```

例如，范围 0 .. = 3 包含数字 0、1、2、3。

只有包含 start 值的范围才可以迭代，因为一个循环必须有开始的地方。但在数组切片中，所有的六种形式都是有用的。如果起始值或终点值被省略了，将默认包含数组从起点开始的元素或者直到终点的元素。

因此快速排序的一个实现，经典的分治排序算法，将类似于这样：

```

fn quicksort<T: Ord>(slice: &mut [T]) {
    if slice.len() <= 1 {
        return; // 切片为空，不需要排序
    }

    // 将切片分为两个部分，前半部分和后半部分
    let pivot_index = partition(slice);

    // 递归排序`slice`的前半部分
    quicksort(&mut slice[.. pivot_index]);

    // 再排序后半部分
    quicksort(&mut slice[pivot_index + 1 ..]);
}

```

6.13 引用运算符

取地址运算符`&`和`&mut`，已经在第5章中介绍过了。

一元`*`运算符用于获取引用指向的值。正如我们看到的，当你使用`.`运算符访问字段或者方法时，Rust会自动解引用。因此只有在我们想读取或写入引用指向的整个值时，`*`运算符才是必须的。

例如，有时迭代器会产生引用，但程序需要底层的类型：

```

let padovan: Vec<u64> = compute_padovan_sequence(n);
for elem in &padovan {
    draw_triangle(turtle, *elem);
}

```

在这个例子中，`elem`的类型是`&u64`，因此`*elem`是一个`u64`值。

6.14 算术、位运算、比较、逻辑运算符

Rust的二元运算符和其他语言中的很像。为了节省时间，我们假设你熟悉这些语言中的一种，并只专注于Rust中背离传统的点。

Rust有通常的算术运算符`+`，`-`，`*`，`/`，`%`。正如在第3章中说过的一样，整数溢出会被检测到，并在debug模式下造成panic。标准库提供类似`a.wrapping_add(b)`的方法来执行不检查溢出的算术。

整数除法会向0舍入，并且整数除以0时即使在release模式也会导致panic。整数有一个`a.checked_div(b)`方法会返回`Option`(如果`b`是0时为`None`)，而不会panic。

一元的-运算符用于求负数，它支持除无符号整数外所有的数字类型。没有一元的+运算符：

```
println!("{}", -100);      // -100
println!("{}", -100u32);   // 错误：一元`-`不能用于类型`u32`
println!("{}", +100);      // 错误：期望表达式，发现`+`
```

和在 C 中一样，`a % b` 会计算有符号的余数，或者叫取模。结果和左侧的操作数符号一致。注意%也可以像整数一样用于浮点数：

```
let x = 1234.567 % 10.0; // 约为 4.567
```

Rust 还继承了 C 的整数的位运算符 &, |, ^, <<, >>。然而，Rust 使用! 而不是~ 来表示按位取反：

```
let hi: u8 = 0xe0;
let lo = !hi; // 0x1f
```

这意味着!`n` 不能用来判断`n` 是否为 0，必须使用`n == 0`。

有符号整数的位移总是符号扩展，无符号整数的位移的位移总是 0 扩展。因为 Rust 有无符号整数，所以不需要像 Java 中的`>>>`一样的无符号移位运算符。

和 C 语言不同的是，位运算符比比较运算的优先级更高，因此如果你写`x & BIT != 0`，意味着`(x & BIT) != 0`。这比 C 中的含义更有用，C 中将是`x & (BIT != 0)`。

Rust 的比较运算符是`==`, `!=`, `<`, `<=`, `>`, `>=`。比较的两个值类型必须相同。

Rust 也有短路求值的`&&` 和`||` 运算符，所有的操作数必须都是`bool` 类型。

6.15 赋值

= 运算符可以用于给`mut` 变量或它们的字段或元素赋值。但在 Rust 中赋值行为并不像在其他语言中那么普遍，因为变量默认是不可变的。

正如在第 4 章介绍的那样，如果变量是 non-Copy 类型，赋值行为将会把它的值移动到目标变量。值的所有权从源对象移动到目的对象，目的对象之前的值会被丢弃。

也可以使用复合赋值：

```
total += item.price;
```

这等价于`total = total + item.price;`。还支持其他的运算符例如：`-=`, `*=` 等。完整的运算符支持在表 6-1 中列出。

和 C 不同，Rust 不支持链式赋值：你不能写`a = b = 3` 来把 3 同时赋给`a` 和`b`。在 Rust 中赋值很罕见，所以你不会怀念这种缩写的。

Rust 不支持 C 的自增和自减运算符`++` 和`--`。

6.16 类型转换

在 Rust 中把一个类型转换为另一个类型通常需要显式的转换。使用 `as` 关键字来进行转换：

```
let x = 17;           // x 是 i32 类型
let index = x as usize; // 转换为 usize
```

Rust 允许以下几种转换：

- 任何内建的数字类型可以彼此转换。

将一个整数转换为另一个整数总是良定义的。转换为更小的类型会导致截断，有符号数转换为更大的类型会进行符号扩展，无符号数是 0 扩展，等等。总而言之，没有意外的行为。

浮点数转换为整数会向 0 舍入：值 -1.99 转换为 `i32` 是 -1。如果值太大不能用整数类型表示，转换会返回整数类型能表示的最接近真实值的值：值 `1e6` 转换为 `u8` 是 255。

- `bool`、`char`、或者类似 C 的 `enum` 类型可以转换为任意整数类型（我们将在[第 10 章](#)中介绍枚举）。

其他方向的转换是不允许的，因为 `bool`、`char` 和 `enum` 都对它们的值有严格的要求，必须要进行运行时检查。例如，将一个 `u16` 数字转换为 `char` 类型是禁止的，因为一些 `u16` 值例如 `0xd800`，对应 Unicode 的代理码点，因此不是有效的 `char` 类型值。有一个标准的方法 `std::char::from_u32()` 来进行运行时检查并返回 `Option<char>`。但需要指出的是，这种转换的需求非常少见。我们通常一次转换整个字符串或流，有关 Unicode 文本的算法通常是非平凡的，最好留给库来实现。

一个例外是，`u8` 可以转换为 `char` 类型，因为 0 到 255 之间的所有整数都是有效的 Unicode 码点。

- 一些涉及 `unsafe` 指针类型的转换也是允许的。见[原始指针](#)一节。

我们说转换通常需要显式的转换。但少数涉及引用类型的转换非常直观以至于语言可以自动完成而不需要显式转换。一个小例子是把 `mut` 引用转换为 `non-mut` 引用。

还有一些更重要的自动转换可能发生：

- `&String` 类型的值可以自动转换为 `&str` 类型。
- `&Vec<i32>` 类型的值可以自动转换为 `&[i32]`。
- `&Box<Chessboard>` 类型的值可以自动转换为 `&Chessboard`。

这些被称为强制解引用 (*deref coercions*)，因为它们适用于实现了内建的 `Deref` trait 的类型。`Deref` 是为了智能指针类型设计的，例如 `Box`，它的行为尽量和底层的值类型保持一致。得益于 `Deref`，使用 `Box<Chessboard>` 非常像在使用一个普通的 `Chessboard`。

用户自定义的类型也可以实现 `Deref` trait。当你需要编写自己的智能指针类型时，见 [Deref 和 DerefMut](#) 一节。

6.17 闭包

Rust 支持闭包：一种轻量的类似函数的值。一个闭包通常由被竖线包围的参数列表和一个表达式组成：

```
let is_even = |x| x % 2 == 0;
```

Rust 会自动推导参数类型和返回值类型。你可以显式写出它们，就像写函数签名一样。如果你指定了返回值类型，那么为了语法的健全，闭包的主体必须是一个块：

```
let is_even = |x: u64| -> bool x % 2 == 0; // error
```

```
let is_even = |x: u64| -> bool { x % 2 == 0 }; // ok
```

调用一个闭包和调用函数的语法相同：

```
assert_eq!(is_even(14), true);
```

闭包是 Rust 中最令人愉快的特性之一，关于它有很多内容可以说，我们将在[第 14 章](#)中介绍。

6.18 继续

表达式是“要运行的代码 (running code)”。它们是 Rust 中要被编译为机器指令的一部分。然而它们只是整个语言的一小部分。

大多数其他语言也是这样。一个程序的第一个任务是运行起来，但这并不是它唯一的任务。程序之间必须交流，它们也必须可以测试，它们还需要保持有序和灵活以便继续改进，它们必须与其它队伍构建的代码和服务进行互操作。而且即使只是为了运行，像 Rust 这样的典型的静态语言的程序也需要更多的工具来组织数据。

接下来，我们将用数个章节讨论这些领域：让你的程序变得结构化的模块和 crate、让你的数据变得结构化的结构体和枚举。

首先，我们将花费一些篇幅来介绍一个非常重要的话题：错误处理。

Chapter 7

错误处理

I knew if I stayed around long enough, something like this would happen.

——George Bernard Shaw on dying

Rust 的错误处理不同寻常，无法用很短的一个章节来介绍它。其实它里面并没有什么困难的概念，只有一些可能对你来说可能很新的概念。这一章将覆盖 Rust 中两种不同的错误处理：panic 和 Result。

一般的错误使用 Result 类型来处理，Result 通常代表程序之外的东西引起的问题，例如错误的输入、网络中断、权限问题等。这种情况的出现不由我们决定，即使是一个完全没有 bug 的程序也可能随时遇到它们。这一章中的大部分内容都是在讨论这种错误。我们将首先介绍 panic，因为它比较简单。

panic 是另一种错误，一种永远不应该发生的错误。

7.1 panic

当程序遇到一些由程序自身的 bug 导致的非常糟糕的事情时它会 panic。例如：

- 数组访问越界
- 整数除以 0
- 在值为 Err 的 Result 上调用 .expect() 方法
- 断言失败

(还有一个宏 `panic!()`，用于当你的代码自己发现了错误，想要直接触发 panic 的情况。`panic!()` 接受可选的 `println!()` 风格的参数，用于构建错误信息。)

这些条件的共同之处在于——它们都是程序员的错。一条好的经验法则是：“不要 panic”。

但是我们都有犯错误的时候。当这些不该发生的错误发生了的时候，该怎么办？值得注意的是，Rust 给了你一个选择：Rust 可以展开堆栈或者中止进程。栈展开是默认行为。

7.1.1 栈展开

当海盗们瓜分抢来的战利品时，船长将得到一半的战利品。普通的船员们均分剩下的一半。（海盗们讨厌分数，因此如果均分时不能除尽，结果会向下取整，余数将分给船上的鹦鹉。）

```
fn pirate_share(total: u64, crew_size: usize) -> u64 {
    let half = total / 2;
    half / crew_size as u64
}
```

这段代码也许可以工作几个世纪，直到有一天船长是抢劫之后唯一的幸存者。如果我们传递的 `crew_size` 为 0，它将会除以 0。在 C++ 中，这将是未定义行为。在 Rust 中，它会触发 panic，panic 通常会按照如下方式继续：

- 打印一条错误消息到终端：

```
thread 'main' panicked at 'attempt to divide by zero',
pirates.rs:3780
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

如果你设置了 `RUST_BACKTRACE` 环境变量，Rust 还会打印出此时的堆栈信息。

- 堆栈被展开。这和 C++ 中的异常处理很像。

任何当前函数内的临时值、局部变量、或者参数都会按照与它们创建时相反的顺序被 drop 掉。

`drop` 一个值意味着清理它：函数使用过的任何 `String` 或 `Vec` 都会被释放，任何打开的 `File` 都会被关闭，等等。用户自定义的 `drop` 方法也会被调用，见 [Drop](#) 一节。在 `pirate_share()` 的例子中，没有要清理的内容。

一旦当前的函数调用被清理完毕，我们会移动到它的调用者，以同样的方式 `drop` 它的变量和参数。然后我们移动到那个函数的调用者，以此类推。

- 最后，线程退出。如果 panic 的线程是主线程，整个进程会退出（退出代码不为 0）。

对这种有序的处理，也许 `panic` 是一个有误导性的名字。`panic` 并不是崩溃，也不是未定义行为，它更类似于 Java 中的 `RuntimeException` 或 C++ 中的 `std::logic_error`。它的行为都是良定义的，它只是不应该发生。

`panic` 是安全的。它不违背 Rust 中的任何安全规则，即使你设法在一个标准库的方法中引起 `panic`，它也永远不会导致悬垂指针或者初始化到一半的值。关键在于 Rust 在任何错误的

事情发生之前就捕捉到了无效的数组访问或者类似的情况。如果继续下去将是不安全的，所以 Rust 会展开堆栈。但进程的其他部分可以继续运行。

panic 以线程为单位，一个线程可以 panic，而其他线程继续处理它们的业务。在第 19 章中，我们会展示一个父线程怎么查明一个子线程是否 panic 并优雅地处理错误。

还有一种方式捕获栈展开，允许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。我们不会解释如何使用它，但 Rust 的测试工具使用了这个机制，用于在测试时断言失败的情况下恢复执行（当编写可以在 C 或 C++ 中调用的 Rust 代码时这也是必须的，因为在非 Rust 代码中的栈展开是未定义行为，见第 22 章）。

理想情况下，我们希望没有 bug 并且永远不会 panic 的代码。但没有完美的事物，你可以使用线程和 `catch_unwind()` 来处理 panic，让你的程序更加健壮。一个重要的警告是这些工具只会捕获展开堆栈的 panic。不是所有的 panic 都以这种方式处理。

7.1.2 中止

栈展开是默认的 panic 行为，但还有两种情况下 Rust 不会尝试展开堆栈。

如果在 Rust 尝试清理时一个 `.drop()` 方法触发了第二次 panic，Rust 会认为这是致命错误，它会停止栈展开并中止整个进程。

还有，Rust 的 panic 行为可以自定义。如果你以参数 `-C panic=abort` 编译，程序中的第一个 panic 会立即中止进程。（这个选项下，Rust 不需要知道如何展开堆栈，因此可以减小编译出的代码的体积。）

最后总结一下关于 Rust 中 panic 的讨论。没有更多要说的了，因为普通的 Rust 代码没有义务处理 panic。即使你使用了线程或 `catch_unwind()`，所有处理 panic 的代码很可能集中在少数部分。没有理由检查程序中的每一个函数然后预测并处理里面的 bug。其他因素导致的错误则是另一码事。

7.2 Result

Rust 里没有异常，可能会失败的函数可以通过返回 `Result` 来表达类似的含义：

```
fn get_weather(location: LatLng) -> Result<WeatherReport, io::Error>
```

`Result` 类型表明可能会失败。当调用 `get_weather()` 函数时，它可能会返回成功的结果 `Ok(weather)`，其中 `weather` 是一个新的 `WeatherReport` 值；或者返回一个错误的结果 `Err(error_value)`，其中 `error_value` 是一个解释错误的 `io::Error`。

Rust 要求我们每次调用这个函数时都要进行一些错误处理。我们必须对返回的 `Result` 做一些处理，才能得到 `WeatherReport` 值。如果 `Result` 值没有被使用的话编译器也会警告。

在第 10 章中，我们将看到标准库是怎么定义 `Result` 的、以及你该怎么自己定义类似的类型。现在，我们将专注于如何使用 `Result` 来进行错误处理。我们将看到如何捕捉、传播和报告错误，以及一些组织和使用 `Result` 类型的常见模式。

7.2.1 捕捉错误

最通用的处理 `Result` 的方法就是我们在第 2 章中展示的：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();
    }
}
```

这在 Rust 中等价于其它语言的 `try/catch`。当你想要自己处理错误，不把错误传递给调用者时你可以使用这种方式。

`match` 有些繁琐，所以 `Result<T, E>` 提供了一些在常见场景下很有用的方法。这些方法中的每一个的内部实现都用到了 `match` 表达式。（`Result` 的全部方法可以查询在线文档。这里列出的方法是我们最常使用的。）

`result.is_ok()`, `result.is_err()`

返回一个 `bool` 值表示 `result` 是成功还是错误。

`result.ok()`

以 `Option<T>` 类型返回成功的结果。如果 `result` 是一个成功的结果，会返回 `Some(success_value)`；否则会返回 `None`，丢弃错误的值。

`result.err()`

以 `Option<E>` 类型返回错误的值。

`result.unwrap_or(fallback)`

如果 `result` 是成功的结果的话，返回成功的值。否则，返回 `fallback`，丢弃错误的值。

```
// 南加州通常的天气情况。  
const THE_USUAL: WeatherReport = WeatherReport::Sunny(72);  
  
// 获取真实的天气预报。  
// 如果失败，倒退到通常的情况。  
let report = get_weather(los_angeles).unwrap_or(THE_USUAL);  
  
display_weather(los_angeles, &report);
```

这是`.ok()`的一个漂亮的替代，因为返回的类型是`T`而不是`Option<T>`。当然，只有当存在有意义的 fallback 值时这么写才有用。

`result.unwrap_or_else(fallback_fn)`

这和上一个类似，但不再是直接传递 fallback 值，而是传递一个函数或者闭包。通常只有在计算 fallback 需要很长时间的情况下才会用到它，因为只有当结果是错误时，`fallback_fn` 才会被调用。

```
let report =  
    get_weather(hometown)  
    .unwrap_or_else(|_err| vague_prediction(hometown));
```

第 14 章会详细介绍闭包。

`result.unwrap()`

如果 `result` 是成功的结果，会返回成功的值。然而，如果 `result` 是错误的结果，这个方法会 panic。这个方法有它的用途，我们将在之后讨论。

`result.expect(message)`

和`.unwrap()`基本相同，不过你可以提供一条 panic 时打印的错误信息。

最后，还有一些和引用相关的方法：

`result.as_ref()`

把一个`Result<T, E>`转换为`Result<&T, &E>`。

`result.as_mut()`

和上面类似，但借用可变引用。返回类型是`Result<&mut T, &mut E>`。

最后两个方法很有用的一个原因是这里列出的其他所有方法，除了`.is_ok()`和`.is_err()`之外，都会消耗操作的`result`值。也就是说，它们以值获取`self`参数。有时如果能访问`result`里的值而不破坏它也会带来便利，这正是`.as_ref()`和`.as_mut()`做的事情。例如，假设你想调用`result.ok()`，但你需要`result`保持完好。你可以写`result.as_ref().ok()`，它只会借用`result`，返回`Option<&T>`而不是`Option<T>`。

7.2.2 Result 类型别名

有时你会看到 Rust 的文档中看起来像是省略了`Result`的错误类型：

```
fn remove_file(path: &Path) -> Result<()>
```

这意味着这里使用了一个`Result`类型的别名。

类型别名是一种缩写的类型名。模块通常会定义一个`Result`类型的类型别名来避免在模块内几乎每个函数中都重复书写相同的错误类型。例如，标准库的`std::io`模块就包含这样一行代码：

```
pub type Result<T> = result::Result<T, Error>;
```

这里定义了一个公有类型`std::io::Result<T>`。它是`Result<T, E>`的一个别名，把`std::io::Error`硬编码为错误类型。在实践中，这意味着如果你写了`use std::io`，Rust 将会把`io::Result<String>`看作`Result<String, io::Error>`的缩写。

当类似于`Result<()>`的东西出现在在线文档里的时候，你可以点击标识符`Result`来查看使用了哪个类型别名和具体的错误类型。在实践中，通常能从上下文中明显地推断出错误类型。

7.2.3 打印错误

有时处理错误的唯一方法就是把它打印到终端然后继续运行程序。我们已经展示了一种这样做的方法：

```
println!("error querying the weather: {}", err);
```

标准库定义了几个错误类型：`std::io::Error`、`std::fmt::Error`、`std::str::Utf8Error`等等。它们全都实现了一个公共接口`std::error::Error` trait，这意味着它们共享下面的特性和方法：

```
println!()
```

所有的错误类型都可以用它来打印。使用 {} 格式说明符只会显式一条简短的错误消息。作为替代，你可以使用 {::?} 格式说明符来获取错误的 Debug 视图。这对用户不是很友好，但包含更多的额外信息。

```
// `println!("error: {}", err);` 的结果
error: failed to lookup address information: No address associated with
hostname
```

```
// `println!("error: {::?}", err);` 的结果
error: Error { repr: Custom(Custom { kind: Other, error: StringError(
    "failed to lookup address information: No address associated with
    hostname") }) }
```

`err.to_string()`

返回一个 `String` 类型的错误消息。

`err.source()`

返回一个导致 `err` 的底层错误，如果有的话。例如，一个网络错误可能会导致一次银行交易失败，这可能反过来导致你的船被收回。如果 `err.to_string()` 是"boat was repossessed"，那么 `err.source()` 可能返回一个有关失败的交易的错误。那个错误的 `.to_string()` 可能是"failed to transfer \$300 to United Yacht Supply"，它的 `.source()` 可能是一个有关网络中断导致错误的详情的 `io::Error`。第三个错误就是根源，因此它的 `.source()` 方法会返回 `None`。因为标准库只包含低级的特性，所以标准库中的错误的来源通常都是 `None`。

打印一个错误值并不会打印出它的来源。如果你想要打印出所有可用的信息，使用这个函数：

```
use std::error::Error;
use std::io::{Write, stderr};

/// 把一个 error 打印到 `stderr`
///
/// 如果在构建错误消息或者写入到 `stderr` 时发生了
/// 另一个 error，那么它会被忽略。
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!` 宏，除了它把数据写入你指定的流。这里，我们将错误消息写入到了标准错误流 `std::io::stderr`。我们也可以使用 `eprintln!` 宏来做同样的事，不过如果中途出错 `eprintln!` 会 panic。在 `print_error` 中，我们希望忽略写入消息时出现的错误；我们会在这一章中的 [忽略错误](#) 一节中解释原因。

标准库的错误类型不包含堆栈追踪，不过当使用 unstable 版本的 Rust 编译器时，流行的 `anyhow` crate 提供了一个现成的有堆栈跟踪的错误类型（Rust 1.50 版本中，标准库中捕获堆栈追踪的函数还没有被稳定化）。

7.2.4 传播错误

很多时候当我们尝试一些可能会失败的操作时，我们并不想立即捕捉并处理错误，在每一个可能出错的地方都使用 10 行的 `match` 表达式太过繁琐。

取而代之的是，当错误出现时我们通常希望让调用者处理它。我们把错误沿着调用栈往上传播。

Rust 有一个 `?` 运算符来做这个工作。你可以在任何产生 `Result` 的表达式后加上 `?`，例如返回 `Result` 的函数调用：

```
let weather = get_weather(hometown)?;
```

`?` 运算符的行为取决于函数返回成功的结果还是错误的结果：

- 成功时，它会解包 `Result` 来获取成功的值。这里 `weather` 的类型并不是 `Result<WeatherReport, io::Error>`，而是 `WeatherReport`。
- 错误时，它会立刻从函数里返回，把错误沿着调用链向上传递。为了做到这一点，`?` 只能用于返回类型是 `Result` 的函数里的 `Result` 值。

`?` 运算符并没有魔法。你可以使用 `match` 表达式做到相同的事，只不过要繁琐很多：

```

let weather = match get_weather(hometown) {
    Ok(success_value) => success_value,
    Err(err) => return Err(err)
};
```

这和 `?` 运算符的唯一区别是一些涉及类型和转换的细节，我们将在下一节中讨论这些细节。

在旧代码中，你可能还会看到 `try!()` 宏，在 Rust 1.13 引入 `?` 运算符之前它是传播错误的通常方法：

```
let weather = try!(get_weather(hometown));
```

这个宏会展开成类似于上面的 `match` 表达式。

我们很容易忘记程序中的错误究竟有多么无孔不入，尤其是那些与操作系统交互的代码。`?=` 运算符有时会在几乎每一行代码中出现：

```
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()? { // 打开目录可能会失败
        let entry = entry_result?; // 读取目录可能会失败
        let dst_file = dst.join(entry.file_name());
        fs::rename(entry.path(), dst_file)?; // 重命名可能会失败
    }
}
```

`?` 也能用于 `Option` 类型。在一个返回 `Option` 的函数中，你可以使用 `?` 来解包值，并在为 `None` 的情况下直接返回：

```
let weather = get_weather(hometown).ok();
```

7.2.5 处理多种错误类型

通常，可能会出错的不止一件事情。假设我们要从文本文件中读取一个数字：

```
use std::io::{self, BufRead};

/// 从文本文件中读取一个整数。
/// 文件中应该每行一个数字。
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?; // 读取一行可能失败
        numbers.push(line.parse()?); // 解析整数可能失败
    }
    Ok(numbers)
}
```

Rust 会报一个编译时错误：

```
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

```

当我们接触到第 11 章中有关 trait 的内容时才能明白错误消息中的术语。到现在为止，只要知道 Rust 是在说? 运算符不能把 std::num::ParseIntError 转换为 std::io::Error 类型就够了。

这里的问题在于从文件中读取一行和解析整数会产生两种不同的错误类型。line_result 的类型是 Result<String, std::io::Error>。line.parse() 的类型是 Result<i64, std::num::ParseIntError>。我们的 read_numbers() 函数的返回类型只能容纳 io::Error。Rust 尝试把 ParseIntError 转换为 io::Error，但这样的转换并不存在，所以向我们汇报了一个错误。

有几种处理这个错误的方法。例如，我们在第 2 章中用于创建曼德勃罗集的图片文件的 image crate 中就定义了它自己的错误类型 ImageError，并实现了从 io::Error 和几种其他错误类型到 ImageError 的转换。如果你想要走这条路，可以尝试 thiserror crate，它被设计的目的就是帮你只用少量代码定义良好的错误类型。

一个更简单的方法是使用 Rust 内建的机制。所有标准库里的错误类型都可以转换为类型 Box<dyn std::error::Error + Send + Sync + 'static>。这有一些复杂，不过 dyn std::error::Error 代表“任何错误”，而 Send + Sync + 'static 使它可以安全地在线程间传递，这通常也是你想要的¹。为了方便，你可以定义类型别名：

```

type GenericError = Box<dyn std::error::Error + Send + Sync + 'static>;
type GenericResult<T> = Result<T, GenericError>;

```

然后，把 read_numbers() 的返回类型修改为 GenericResult<Vec<i64>>。修改之后，函数就可以编译了。? 运算符会在需要的时候把任何错误类型转换为 GenericError。

顺带，? 运算符通过一个标准库方法来完成自动转换，你也可以手动使用这个方法。可以调用 GenericError::from() 来将任何错误转换为 GenericError 类型：

```

let io_error = io::Error::new(
    io::ErrorKind::Other, "timed out");
return Err(GenericError::from(io_error)); // 手动转换为 GenericError

```

¹ 你也可以考虑使用流行的 anyhow crate，它提供类似于我们的 GenericError 和 GenericResult 的类型，不过还带有一些其他的有用的特性。

我们将在第13章中详细介绍From trait和它的from()方法。

用GenericError方法处理的缺点是返回类型不能精确地告诉调用者到底发生了什么类型的错误。如果你要调用一个返回GenericResult的函数并且想处理某一种特定类型的错误，然后继续传播其它所有类型的错误，可以使用泛型方法error.downcast_ref::<ErrorType>()。如果这个错误正好是你指定的错误类型：

```
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; // try again!
            }
            return Err(err);
        }
    }
}
```

许多语言都有类似这样的内建语法，但它们通常极少被用到。Rust使用一个方法来完成这件事。

7.2.6 处理“不可能发生”的错误

有时我们知道有一些错误不可能发生。例如，假设我们正在编写一个解析配置文件的函数，并且我们发现文件中接下来的内容是一串数字的字符串：

```
if next_char.is_digit(10) {
    let start = current_index;
    current_index = skip_digits(&line, current_index);
    let digits = &line[start..current_index];
    ...
}
```

我们想把这一串数字转换为一个真正的数字。有一个标准的方法可以做到这一点：

```
let num = digits.parse::<u64>();
```

现在的问题是：str.parse::<u64>()方法并不返回u64。它返回一个Result。它可能失败，因为一些字符串不能转换为数字：

```
"bleen".parse::<u64>() // ParseIntError: invalid digit
```

但我们确切的知道，在我们的例子中，digits只包含一串数字。我们应该怎么做？

如果我们写的代码返回一个 `GenericResult`, 我们可以在后边加上一个? 然后忘记它。否则, 我们就必须为不可能发生的错误编写错误处理的代码。此时最佳的选择是使用 `.unwrap()`, 如果结果是 `Err` 的话这个方法会 `panic`, 但如果是 `Ok` 的话则直接返回成功的值:

```
let num = digits.parse::<u64>.unwrap();
```

这和使用? 很像, 不同之处在于如果我们判断错了, 这个错误有可能发生, 那么当它发生时会 `panic`。

事实上, 在某些特殊情况下这个例子也可能出错。如果输入包含足够长的数字串, 那么数字将会过大不能存储在 `u64` 中:

```
"99999999999999999999".parse::<u64>() // 溢出错误
```

在这个特殊情况下使用 `.unwrap()` 将会是一个 bug。错误的输入不应该导致 `panic`。

也就是说, 确实有可能出现 `Result` 值一定不是错误的情况。例如, 在第 18 章中, 你会看到 `Write` trait 为文本和二进制输出定义了一组通用的方法 (`.write()` 等)。所有这些方法返回 `io::Result`, 但如果你要写入到一个 `Vec<u8>`, 那么它们不可能失败。在这种情况下, 使用 `.unwrap()` 或者 `.expect(message)` 来处理 `Result` 是可以接受的。

如果当错误发生时说明出现了非常奇怪的情况, 以至于你确实想直接 `panic`, 那么这些方法也很有用:

```
fn print_file_age(filename: &Path, last_modified: SystemTime) {
    let age = last_modified.elapsed().expect("system clock drift");
}
```

这里, `.elapsed()` 方法只在当前系统时间比文件创建的时间更早的情况下才会失败。如果文件是新创建的, 并且在我们的程序运行期间系统的时钟被往回调了, 就有可能出现这种情况。取决于代码如何使被使用, 在这种情况下 `panic`, 而不是处理错误或者传播给调用者, 是一个合理的判断。

7.2.7 忽略错误

偶尔我们只是想忽略一个错误。例如, 在我们的 `print_error` 函数中, 我们必须处理打印一个错误时触发另一个错误的罕见场景。这有可能发生, 例如, 如果 `stderr` 通过管道连接到其他进程, 并且那个进程被杀死了。我们原本尝试汇报的错误可能更重要, 需要传播; 而 `stderr` 的错误我们想直接忽略, 但 Rust 编译器会警告有未使用的 `Result` 值:

```
writeln!(stderr(), "error: {}", err); // 警告: 未使用的结果
```

惯用写法 `let _ = ...` 可以用来消除这个警告:

```
let _ = writeln!(stderr(), "error: {}", err); // ok, 忽略结果
```

7.2.8 在 main() 中处理错误

在大多数产生 `Result` 的场景中，将错误向上传递给调用者是正确的行为。这也是为什么？在 Rust 中是单个字符。正如我们所见，在一些程序中它被用于很多行代码。

但如果你把一个错误传播的够长了，已经到达了 `main()`，必须要对它做些处理了。通常情况下，`main()` 不能使用？因为它的返回类型不是 `Result`：

```
fn main() {
    calculate_tides()?;
    // error: can't pass the buck any further
}
```

在 `main()` 中处理错误的最简单方法就是使用 `.expect()`：

```
fn main() {
    calculate_tides().expect("error"); // the buck stops here
}
```

如果 `calculate_tides()` 返回一个错误结果，那么 `.expect()` 方法会 `panic`。在主线程中 `panic` 会打印一条错误消息并以非 0 的退出码退出程序。我们将会在小程序中一直使用这种写法。这是开始。

错误消息有一点吓人：

```
$ 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
```

错误消息在干扰中丢失。`RUST_BACKTRACE=1` 也是一个错误的建议。

然而，你可以修改 `main()` 的类型签名，让它返回一个 `Result` 类型，这样你可以使用？：

```
fn main() -> Result<(), TideCalcError> {
    let tides = calculate_tides()?;
    print_tides(tides);
    Ok(())
}
```

这可以用于任何可以以 `:?` 格式符打印的错误类型，所有的标准错误类型例如 `std::io::Error` 都满足条件。这种方法易于使用并且给出更好的错误消息，但并不是理想的方案：

```
$ tidecalc --planet mercury
Error: TideCalcError { error_type: NoMoon, message: "moon not found" }
```

如果你要处理更复杂的错误类型或者想在信息中包含更多信息，那你可以自己打印错误信息：

```
fn main() {
    if let Err(err) = calculate_tides() {
        print_error(&err);
        std::process::exit(1);
    }
}
```

这段代码使用了 `if let` 表达式来打印错误消息，只有当 `calculate_tides()` 的调用返回错误结果时才会打印消息。更多有关 `if let` 表达式的细节，见[第 10 章](#)。`print_error` 函数在[打印错误](#)中列出。

现在输出变得漂亮整洁：

```
$ tidecalc --planet mercury
error: moon not found
```

7.2.9 声明自定义错误类型

假设你在编写一个新的 JSON 解析器，而且你想让它有自己的自定义错误类型。（我们还没有介绍如何自定义类型，不过再过几章就会介绍了。错误类型很方便，因此我们将在此处提供一些先睹为快的预览。）

你需要编写的最少的代码如下：

```
// json/src/error.rs

#[derive(Debug, Clone)]
pub struct JsonError {
    pub message: String,
    pub line: usize,
    pub column: usize,
}
```

这个结构体将被称作 `json::error::JsonError`，当你需要返回一个这种类型的错误时，你可以这么写：

```
return Err(JsonError {
    "message": "expected ']' at end of array".to_string(),
    line: current_line,
    column: current_column
});
```

这没有任何问题。然而，如果你想让你的错误类型能像标准错误类型一样工作，这可能也是你的库的使用者希望的，那么你需要再多做一些工作：

```
use std::fmt;

// 错误需要能打印。
impl fmt::Display for JsonError {
    fn fmt(&self, f: &mut fmt::Formatter) -> Result<(), fmt::Error> {
        write!(f, "{} ({:}:{:})", self.message, self.line, self.column)
    }
}

// 错误需要实现 std::error::Error trait,
// 但 Error 的方法的默认实现已经足够了。
impl std::error::Error for JsonError { }
```

`impl` 关键字、`self` 和其它内容将会在几章之后解释。

和 Rust 语言中的其它很多方面一样，有一些 crate 可以让错误处理变得更加易用和简洁。有很多选择，但其中使用最广泛的是 `thiserror`，它帮你完成上述所有的工作，让你可以像这样定义错误：

```
use thiserror::Error;
#[derive(Error, Debug)]
#[error("{message}:{:} ({line}:{}, {column})")]
pub struct JsonError {
    message: String,
    line: usize,
    column: usize,
}
```

`#[derive(Error)]` 指示会告诉 `thiserror` 生成之前展示的代码，这可以节省很多时间和工作。

7.2.10 为什么选择 Result

现在我们已经知道足够多，可以理解 Rust 为什么选择 `Result` 而不是异常了。这是这个设计的关键点：

- Rust 需要程序员在每个可能出现错误的点做出决策，并记录在代码中。这是一个很好的设计，否则很容易因为疏忽而忘记处理错误。
- 最常见的决策是传播错误，并且使用单个字符? 来完成。因此，因此，错误传播不会像在 C 和 Go 中那样扰乱你的代码。并且它还是可见的：你可以在一大段代码中一眼看到

错误在哪里被传播。

- 因为错误是函数返回值的一部分，所以很容易看出哪些函数可能失败、哪些不可能。如果你把一个函数改成可能失败，你必须修改它的返回类型，因此编译器会让你同时更新下游使用了函数的代码。
- Rust 通过检查确保 `Result` 值被使用，因此你不会偶然遗漏错误（在 C 中这是一种常见的错误）。
- 因为 `Result` 是一个类似其他类型的数据类型，它可以方便的在相同的集合中存储成功和失败的结果。这让它可以很容易的建模部分成功的情况。例如，如果你在编写一个程序从读取一个文本文件中的几百万条记录，并且你需要一种方法来应对大多数会成功但有些会失败的可能结果，你可以使用一个 `Result` 的 vector 来表示这种情形。

代价就是你会发现你在 Rust 中关于错误处理的思考和设计会比在其他语言中更多。和许多其他领域一样，Rust 对错误处理的要求比你习惯的要更严格一点。对系统编程语言来说，这是值得的。

Chapter 8

crate 与模块

This is one note in a Rust theme: systems programmers can have nice things.

——Robert O’Callahan, “Random Thoughts on Rust: crates.io and IDEs”

假设你在编写一个仿真蕨类植物从细胞开始生长的程序。你的程序就像蕨类一样，一开始非常简单，可能所有代码都在单个文件里——就像一个孢子。随着它逐渐成长，它开始逐渐建立起内部的结构，不同的片段负责不同的功能。它将分裂为多个文件，可能覆盖整个目录树。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。对于任何成长到不仅仅是几个数据结构和几百行代码的程序，都必须要对代码进行组织。

这一章将会介绍 Rust 中用于组织程序的特性：crate 和模块。我们还会介绍 Rust crate 的结构和分发相关的话题，包括如何编写文档和测试 Rust 代码，如何禁用不需要的编译器警告，如何使用 Cargo 来管理项目依赖和版本，如何在 Rust 的公开 crate 仓库：crates.io 上发布开源的库，crate 的版本如何演变，等等。我们将使用蕨类仿真程序作为我们的例子。

8.1 Crate

Rust 程序由 *crates* 组成。每一个 crate 都是一个完整的、一体的单元：一个库或可执行文件的所有代码、加上相关的测试、示例、工具、配置、以及一些其他东西。为了编写你自己的蕨类模拟器，你可能需要使用和 3D 图形、生物信息学、并行计算等相关的第三方库。这些库就像箱子一样（见图 8-1）。

查看 crate 是什么以及它们是如何工作的最简单方法就是使用带有`--verbose` 参数的 `cargo build` 来构建一个有一些依赖的程序。我们用一个并发的曼德勃罗集作为示例。结果如下所示：

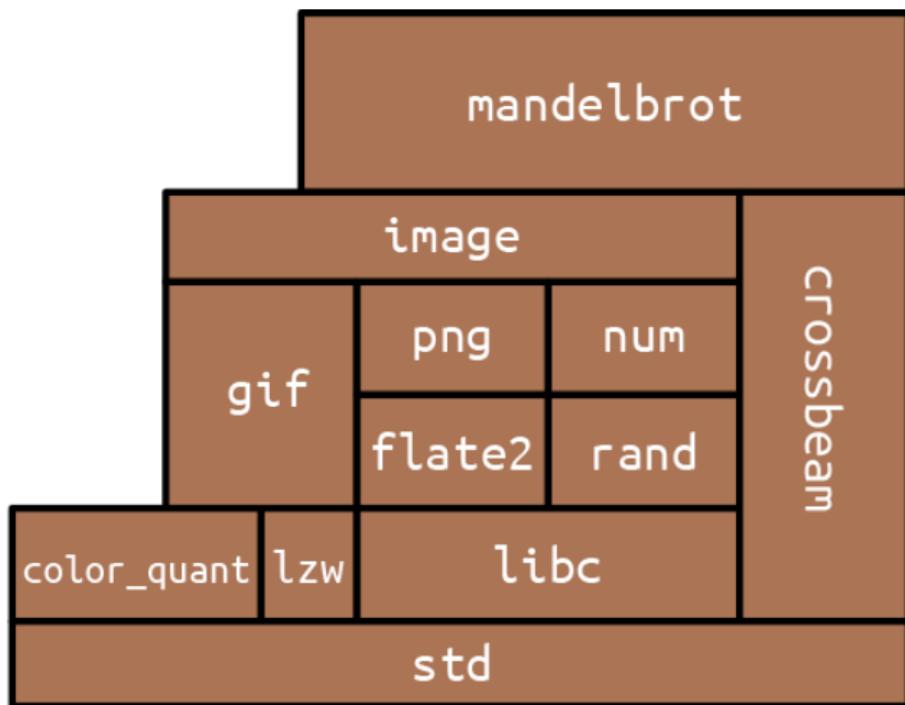


图 8-1: 一个 crate 和它的依赖

```
$ cd mandelbrot
$ cargo clean    # delete previously compiled code
$ 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-a7006d35f1b58972.rmeta
--extern jpeg_decoder=.../libjped_decoder-5c10558d0d57d300.rmeta
Compiling mandelbrot v0.1.0 (/tmp/rustbook-test-files/mandelbrot)
Running `rustc
--edition=2018
--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
```

我们重新格式化了 `rustc` 的命令行来改善可读性，并且删掉了很多和我们的讨论无关的编译器选项，用省略号 (...) 代替了它们。

你可能还记得，当我们完成曼德勃罗集程序时，它的 `main.rs` 包含几个引入其它 crate 的 `use` 声明：

```
use num::Complex;
// ...
use image::ColorType;
use image::png::PNGEncoder;
```

我们还在 `Cargo.toml` 中指定了每个 crate 的版本：

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

这里的依赖指这个程序使用的其它 crate，也就是我们依赖的代码。我们可以在[crates.io](#)中找到这些 crate，那是 Rust 社区用于存放开源的 crate 的网站。例如，我们可以访问 [crates.io](#) 并搜索图片库来找到 `image` 库。`crates.io` 上的每个 crate 的页面上会显示它的 `README.md` 文件和到文档和源代码的链接，还有一行配置例如 `image = "0.13"`，你可以复制这一行并添加到你的 `Cargo.toml` 中。这里显示的版本号直接用了我们在编写这个程序时这三个包的最新版本。

`Cargo` 的输出说明了这些信息是如何被使用的。当我们运行 `cargo build` 时，`Cargo` 会首先从 [crates.io](#) 下载这些 crate 的指定版本的源码。然后，它读取那些 crate 的 `Cargo.toml` 文

件，下载它们的依赖，然后递归操作。例如，`image` crate 的 0.13.0 版本的源代码中包含一个 `Cargo.toml` 文件，内容如下：

```
[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"
```

看到这些内容，Cargo 知道在它可以使用 `image` 之前，它必须先拉取这些 crate。我们称它们为 `mandelbrot` 的间接 (*transitive*) 依赖。所有这些依赖的集合告诉了 Cargo 需要知道的有关如何构建和构建顺序的一切信息，它被称为 crate 的依赖图。Cargo 自动处理依赖图和间接依赖的能力是程序员们付出时间和努力的一大胜利。

当获得了源代码之后，Cargo 会编译所有的 crate。它会运行 Rust 的编译器 `rustc`，一次编译依赖图中的一个 crate。当编译这些库时，Cargo 会使用 `--crate-type lib` 选项。这告诉 `rustc` 不要寻找 `main()` 函数，而是产生一个包含编译过代码的 `.rlib` 文件，这个文件可以被用于创建可执行文件和其他 `.rlib` 文件。

当编译程序时，Cargo 会使用 `--crate-type bin`，编译的结果将是一个目标平台的二进制可执行文件：例如在 Windows 上就是 `mandelbrot.exe`。

对于每一个 `rustc` 命令，Cargo 都会传递 `--extern` 选项，给出 crate 用到的每一个库的名称。这样，当 `rustc` 看到一行类似于 `use image::png::PNGEncoder` 的代码时，它可以分辨出 `image` 是另一个 crate 的名字，而且 Cargo 传递的选项让它知道该从哪里寻找编译好的 crate。Rust 的编译器需要访问这些 `.rlib` 文件，因为它们包含编译好的库中的代码。Rust 将会将代码静态链接到最终的可执行文件中。`.rlib` 还包含类型信息，因此 Rust 可以通过检查确保我们在代码中使用的库的特性确实存在而且被正确使用。它还包含一份 crate 的 public 内联函数、泛型、宏、特性的拷贝，这些东西只有当 Rust 看到我们如何使用它们时才可以将它们编译为机器代码。

`cargo build` 支持各种选项，其中的大部分都超出了本书的范围，不过我们在这里会提到其中一个：`cargo build --release` 会生成优化后的构建。Release 构建运行得更快，但需要更长的时间来编译，而且它们不检查整数溢出、跳过 `debug_assert!()` 断言，并且它们在 `panic` 时生成的堆栈追踪通常不太可靠。

8.1.1 版本

Rust 有极强的兼容性保证。任何在 Rust 1.0 中能编译的代码必须在 Rust 1.50 或者 1.900 (如果发布了的话) 中也能编译。

但有时社区会遇到一些令人信服的扩展语言的建议，这可能会导致旧代码不能再编译。例如，经过了多次讨论之后，Rust 确定了一种支持异步编程的语法，将标识符 `async` 和 `await` 重新用作关键字 (见第 20 章)。但这项语言的改变可能会导致使用 `async` 或者 `await` 作为变量名的代码不能再编译。

为了在不破坏这些现有代码的前提下演变，Rust 使用了版本。Rust 的 2015 版本和 Rust 1.0 兼容。2018 版本将 `async` 和 `await` 改为关键字、精简了模块系统、还引入了一些和 2015 版本不兼容的其它语言更改。每个 crate 在 *Cargo.toml* 文件中的 `package` 节中用一行类似如下的说明指定 Rust 的版本：

```
edition = "2018"
```

如果缺少这个关键字，将会假设使用 2015 版本，因此旧的 crate 完全不需要做任何更改。但如果你想使用异步函数或者新的模块系统，你需要确保 *Cargo.toml* 中有 `edition = "2018"` (或者可能更新的版本)。

Rust 保证编译器将总是接受语言的所有版本，并且程序可以自由混合使用不同版本编写的 crate。即使一个 2015 版本的 crate 依赖一个 2018 版本的 crate 也没有问题。换句话说，一个 crate 的版本只影响它的代码是如何被构建的，版本的区别只体现在代码编译的时候。这意味着没有必要更新旧的版本来适配现代 Rust 的生态。类似的，也没有必要将 crate 保持在旧版本来避免影响到它的用户。你只需要在想使用新的语言特性时更改自己代码中的版本。

版本并不是每年都会更新，只有当 Rust 项目觉得有必要出新版本的时候才会更新。例如，没有 2020 版本。把 `edition` 设置为 "2020" 将会导致错误。[Rust 版本指南](#)介绍了每一个版本中的变化，并提供了版本系统的背景知识。

使用最新版本几乎总是一个好主意，尤其是新编写代码时。`cargo new` 会默认创建最新版本的项目。本书中将始终使用 2018 版本。

如果你有一个用更旧版本的 Rust 编写的 crate，`cargo fix` 命令也许可以帮你自动把代码更新到更新的版本。[Rust 版本指南](#)详细解释了 `cargo fix` 命令。

8.1.2 构建配置

有几个 *Cargo.toml* 中的配置选项可以影响到 `cargo` 生成的 `rustc` 命令行 ([表 8-1](#))。

通常默认的行为就足够了，但我们会发现一个例外是你想使用一个 profiler——一个用于测量程序使用 CPU 时间情况的工具。为了从 profiler 获取最准确的数据，你将同时需要优化

表 8-1: Cargo.toml 配置节

命令行	使用到的 Cargo.toml 节
cargo build	[profile.dev]
cargo build --release	[profile.release]
cargo test	[profile.test]

(通常只在 release 构建中可用) 和调试符号 (通常只在 debug 构建中可用)。为了同时启用两者，在 *Cargo.toml* 中添加：

```
[profile.release]
debug = true      # 允许在 release 构建中启用调试符号
```

debug 设置控制是否给 `rustc` 传递 `-g` 选项。有了这个配置，当你输入 `cargo build --release` 时，你将会得到一个带有调试符号的二进制文件。优化的设置将不会被影响。

[Cargo 文档](#) 中列出了很多其他你可以在 *Cargo.toml* 中调整的设置。

8.2 模块

如果说 crate 决定了项目之间的代码共享，那么模块则决定了项目内部的代码组织。它们扮演了 Rust 中的命名空间——一种包含函数、类型、常量等内容的容器，这些模块组成了你的 Rust 程序或库。一个模块看起来类似于这样：

```
mod spores {
    use cells::{Cell, Gene};

    /// 成熟蕨类植物产生的细胞。它会随着风飘散,
    /// 这也是蕨类生命周期的一部分。一个孢子会成长为一个原叶体——
    /// 一个宽达 5mm 的完整的独立有机体。它会产生受精卵,
    /// 这些受精卵会成长为新的蕨类植物 (植物的性别很复杂)。
    pub struct Spore {

        ...
    }

    /// 模拟通过减数分裂产生孢子的过程。
    pub fn produce_spore(factory: &mut Sporangium) -> Spore {
        ...
    }
}
```

```
// 提取一个孢子中的基因。  
pub(crate) fn genes(spore: &Spore) -> Vec<Gene> {  
    ...  
}  
  
/// 混合基因为减数分裂做准备（分裂间期的一部分）。  
fn recombine(parent: &mut Cell) {  
    ...  
}  
  
...  
}
```

模块是 *item* 的集合，*item* 是命名的特性，例如本例中的 Spore 结构体和两个函数。pub 关键字将 item 设为公有的，因此可以从模块外部访问。

把函数标记为 pub(crate)，意味着它在这个 crate 中任何地方都可以访问，但不作为外部接口的一部分公开。它不能被其他 crate 使用，也不会在 crate 的文档中显示。

任何没有被标记为 pub 的都是私有的，只能在定义它的模块和子模块中使用：

```
let s = spores::produce_spore(&mut factory); // ok  
  
spores::recombine(&mut cell); // 错误：`recombine` 是私有的
```

将 item 标记为 pub 通常称为“导出”这个 item。

这一节的剩余部分将覆盖使用模块所需要了解的细节：

- 我们会展示如果需要的话怎么嵌套模块和把它们分布在不同的文件和目录中。
- 我们会解释 Rust 从其他模块中引用 item 的路径语法，并展示怎么导入 item，这样就不需要每次都写出完整的路径。
- 我们会接触 Rust 对结构体字段的细粒度控制。
- 我们会介绍 prelude 模块，它通过收集几乎所有用户都会用到的常见导入来减少重复的导入。
- 我们会展示常量和静态量，这是两种为了清晰和一致性而设计的定义命名变量的方式。

8.2.1 嵌套模块

模块可以嵌套，事实上一个模块只是一些子模块的集合的情况是很常见的：

```
mod plant_structures {  
    pub mod roots {  
        ...  
    }  
}
```

```

    }
    pub mod stems {
        ...
    }
    pub mod leaves {
        ...
    }
}

```

如果你想要让嵌套模块中的一个 item 对其他 crate 可见，那需要保证将它和所有嵌套包含它的模块标记为 public。否则你会看到一个类似这样的警告：

```

warning: function is never used: `is_square`
--> src/crates_unused_items.rs:23:9
|
23 | /         pub fn is_square(root: &Root) -> bool {
24 | |             root.cross_section_shape().is_square()
25 | |
| |-----^
|

```

可能这个函数这时确实是死代码。但如果你是想将它用在其他 crate 中，Rust 会让你明白它实际上并不可见。你需要保证嵌套包含它的模块也都被标记为 pub。

也可以声明 pub(super)，让一个 item 只在父模块中可见。pub(in <path>) 可以让它在一个指定的父模块和其后代中可见。这在深层嵌套的模块中很有用：

```

mod plant_structures {
    pub mod roots {
        pub mod products {
            pub(in crate::plant_structures::roots) struct Cytokinin {
                ...
            }
        }
    }

    use products::Cytokinin; // ok: 在`roots`模块中
}

use roots::products::Cytokinin; // error: `Cytokinin` 是私有的
}

// error: `Cytokinin` 是私有的
use plant_structures::roots::products::Cytokinin;

```

通过这种方式，我们可以写出一个有数量庞大的代码和完整的模块层次结构的程序，不管这些模块的关系如何，我们都可以将整个程序写在单个文件里。

但实际上以这种方式来工作非常的痛苦，因此还有另一种方案。

8.2.2 单独文件中的模块

一个模块还可以这么写：

```
mod spores;
```

之前，我们还在花括号中包含了 `spores` 模块的主体。这里，我们通过这种方式告诉 Rust 编译器 `spores` 模块在一个单独的叫 `spores.rs` 的文件里：

```
// spores.rs

/// 成熟蕨类植物产生的一个细胞...
pub struct Spore {
    ...
}

/// 模拟减数分裂产生孢子的过程。
pub fn produce_spore(factory: &mut Sporangium) -> Spore {
    ...
}

/// 从一个孢子中提取基因。
pub(crate) fn genes(spore: &Spore) -> Vec<Gene> {
    ...
}

/// 混合基因为减数分裂做准备（分裂间期的一部分）。
fn recombine(parent: &mut Cell) {
    ...
}
```

`spores.rs` 只包含组成模块的 item。它并不需要任何说明来表明它是一个模块。

这个 `spores` 模块和我们在上一节中展示的版本的唯一不同就是代码的位置。有关公有性和私有性的规则和之前完全相同。Rust 从来不会单独编译模块，即使它们在单独的文件里。当你构建一个 Rust 的 crate 的时候，你总是要重新编译它里面所有的模块。

一个模块也可以有自己的目录。当 Rust 看到 `mod spores;` 时，它会检查 `spores.rs` 和 `spores/mod.rs`，如果这两个文件都不存在或者都存在就会报错。本例中因为 `spores` 模块没有

任何子模块，所以我们使用了 `spores.rs`。但考虑一下我们之前写的 `plant_structures` 模块。如果我们决定将那个模块和它的三个子模块分割在单独的文件中，最终的项目看起来就是这样：

```
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 去加载 `plant_structures/mod.rs`，这个文件里又声明了三个子模块：

```
// 在 plant_structures/mod.rs 中
pub mod roots;
pub mod stems;
pub mod leaves;
```

这三个模块的内容都被存储在单独的文件中，分别命名为 `leaves.rs`、`roots.rs`、`stems.rs`，和 `mod.rs` 一样在 `plant_structures` 目录下。

使用同名的文件和目录来组成模块也是可行的。例如，如果 `stems` 需要包含两个分别叫 `xylem` 和 `phloem` 的模块，我们可以选择将 `stems` 保留在 `plant_structures/stems.rs` 中，然后添加一个新的 `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` 中声明这两个新的子模块：

```
// 在 plant_structures/stems.rs 中  
pub mod xylem;  
pub mod phloem;
```

这三种方式——模块在自己单独的文件中、模块在自己同名的目录中的 `mod.rs` 中，模块在自己单独的文件中并有一个同名的目录包含子模块——给了模块系统足够的灵活性来支撑你需要的任何程序结构。

8.2.3 路径和导入

`::` 运算符用于访问模块中的特性。你的项目中任何地方的代码都可以通过路径来引用任何标准库的特性：

```
if s1 > s2 {  
    std::mem::swap(&mut s1, &mut s2);  
}
```

`std` 是标准库的名称。路径 `std` 指向标准库的顶层模块。`std::mem` 是在标准库中的一个子模块，`std::mem::swap` 是 `std::mem` 模块中的一个 public 函数。

你可以始终用这种方式编写所有的代码，每当你需要圆或字典时都写出 `std::f64::consts::PI` 和 `std::collections::HashMap::new`，但这样太过繁琐，而且难以阅读。替代方案是把一些特性导入用到它们的模块：

```
use std::mem;  
  
if s1 > s2 {  
    mem::swap(&mut s1, &mut s2);  
}
```

`use` 声明会导致 `mem` 变为 `std::mem` 在整个块或模块中的一个局部别名。

你也可以写 `std::mem::swap` 来导入 `swap` 函数本身，而不是 `mem` 模块。然而，我们之前的方式被认为是最佳的风格：引入类型、trait 和模块（例如 `std::mem`）然后使用相对路径访问函数、常量和其他成员。

可以一次导入若干个名字：

```
use std::collections::{HashMap, HashSet}; // 导入两个

use std::fs::{self, File}; // 导入`std::fs`和`std::fs::File`

use std::io::prelude::*;

// 导入所有内容
```

也可以写出所有的单独导入：

```
use std::collections::HashMap;
use std::collections::HashSet;

use std::fs;
use std::fs::File;

// std::io::prelude 中的所有 public item:
use std::io::prelude::Read;
use std::io::prelude::Write;
use std::io::prelude::BufRead;
use std::io::prelude::Seek;
```

你可以使用 `as` 导入一个 item 并同时给它一个不同的局部名称：

```
use std::io::Result as IOResult;

// 返回类型等价于`std::io::Result<()>`
fn save_spore(spore: &Spore) -> IOResult<()>
...
```

模块并不会自动从父模块中继承名称。例如，假设我们的 `proteins/mod.rs` 有如下内容：

```
// proteins/mod.rs
pub enum AminoAcid { ... }
pub mod synthesis;
```

那么 `synthesis.rs` 里的代码并不会自动导入类型 `AminoAcid`：

```
// proteins/synthesis.rs
pub fn synthesis(seq: &[AminoAcid]) // 错误：找不到类型`AminoAcid`
```

每一个模块都会以空白的状态开始，必须导入它使用的名称：

```
// proteins/synthesis.rs
use super::AminoAcid; // 显式地从父模块中导入
pub fn synthesize(seq: &[AminoAcid]) // ok
```

默认情况下，路径是相对于当前模块的：

```
// 在proteins/mod.rs 中

// 从子模块中导入
use synthesis::synthesize;
```

`self` 也是当前模块的同义词，因此我们可以写：

```
// 在proteins/mod.rs 中

// 导入一个枚举中的名字
// 这样我们可以用`Lys`来表示赖氨酸，而不是`AminoAcid::Lys`
use self::AminoAcid::*;

或者简写为：
```

```
// 在proteins/mod.rs 中

use AminoAcid::*;

（这里的 AminoAcid 的例子，有些违背我们之前提到的只导入类型、trait 和模块的风格。如果我们的程序包含很长的氨基酸序列，那么根据奥威尔的第六原则：“Break any of these rules sooner than say anything outright barbarous.”（绝不要用粗俗语言，为此可以打破上面任一规则。）这么做也是有道理的。）

路径中的 super 和 crate 关键字有特殊的含义：super 指代父模块，crate 指代包含当前模块的 crate。
```

使用相对于 `crate` 根的路径而不是相对于当前模块的路径可以使在项目中移动代码变得更容易，因为这样的话就算当前模块的路径变了，导入也不会出错。例如，我们可以使用 `crate` 来写 `synthesis.rs`：

```
// proteins/synthesis.rs
use crate::proteins::AminoAcid; // 显式的相对于crate根的导入

pub fn synthesize(seq: &[AminoAcid]) // ok
    ...

子模块可以通过 use super::* 访问父模块中的私有 item。
```

如果你有一个模块和当前正在使用的某一个模块同名，那么引用它们的时候就要小心了。例如，如果你的程序在 `Cargo.toml` 列出了 `image` `crate` 依赖，同时还有一个模块叫 `image`，那么以 `image` 开头的路径将会导致歧义：

```
mod image {
    pub struct Sampler {
        ...
    }
}

// 错误：这是指向`image`模块，还是`image` crate？
use image::Pixels;
```

即使 `image` 模块没有 `Pixels` 类型，这个歧义也会被认为是错误：如果之后又添加了 `Pixels` 的定义，那么可能会改变路径指向的内容，这可能会令人迷惑。

为了解决歧义，Rust 有一种特殊的路径称为绝对路径，它们以`::`开头，将总是指向一个外部的 crate。为了指向 `image` crate 中的 `Pixels` 类型，你可以写：

```
use ::image::Pixels; // `image` crate 的 `Pixels`
```

为了指向你自己的模块中的 ‘`Sampler`’ 类型，你可以写：

```
use self::image::Sampler; // `image` 模块的 `Sampler`
```

模块和文件的概念并不相同，但模块和 Unix 文件系统中的文件和目录存在自然的类比关系。`use` 关键字创建别名，就像 `ln` 命令创建链接。路径类似于文件名，有绝对路径和相对路径两种形式。`self` 和 `super` 类似于`.` 和`..` 这两个特殊的目录。

8.2.4 标准 prelude

我们之前说每个模块都以“空白的状态”开始，但事实上并不是完全空白的状态。

其中一点是，标准库 `std` 被自动链接到每个项目。这意味着你总是可以使用 `std::whatever` 这种方式来引用 `std` 里的 item，例如 `std::mem::swap()`。另外，一些特殊的常见名称，例如 `Vec` 和 `Result` 也被包含在标准 `prelude` 中并且被自动导入。具体的行为就好像是包括根模块在内的每个模块都以如下导入开始：

```
use std::prelude::v1::*;


```

标注的 `prelude` 包含一些通用的 trait 和类型。

在第 2 章中时，我们提到了库有时会提供叫 `prelude` 的模块。但 `std::prelude::v1` 是唯一一个自动导入的。将一个模块命名为 `prelude` 只是一个惯例，告诉用户他应该导入 `*`。

8.2.5 pub use 声明

即使 `use` 声明只是别名，它们也可以是公有的：

```
// in plant_structures/mod.rs
...
pub use self::leaves::Leaf;
pub use self::roots::Root;
```

这意味着 Leaf 和 Root 是 plant_structures 模块里的 public item。它们实际上只是 plant_structures::leaves::Leaf 和 plant_structures::roots::Root 的别名。

标准的 prelude 就是以一系列 pub 导入的方式实现的。

8.2.6 pub 结构体字段

模块可以包含用户用 struct 关键字定义的自定义结构体类型。我们将在[第 9 章](#)中介绍细节，但这是一个了解模块如何和结构体字段的可见性交互的好时机。

一个简单的结构体类似这样：

```
pub struct Fern {
    pub roots: RootSet,
    pub stems: StemSet
}
```

一个结构体的所有字段，包括私有字段，都可以在定义结构体的整个模块及其子模块中访问，在模块之外，只有公有的字段才可以被访问。

事实证明通过模块来实现访问控制，而不是像 Java 和 C++ 那样通过类来实现，对于程序设计将是很大的帮助。不仅能减少重复“getter”和“setter”方法，还能消除类似 C++ 中 friend 声明的需求。一个模块中可以定义几个紧密结合的类型，例如 frond::LeafMap 和 frond::LeafMapIter，让它们能按需互相访问彼此的私有字段，同时对程序中的其他部分隐藏实现的细节。

8.2.7 静态量和常量

除了函数、类型和嵌套模块之外，模块里还可以定义常量和静态量。

const 关键字定义常量，语法类似于 let，除了它必须被标记为 pub 以及要显式写出类型。还有，为了方便，常量一般都用 UPPERCASE_NAMES：

```
pub const ROOM_TEMPERATURE: f64 = 20.0;      // 摄氏度
```

static 关键字定义静态量，和 const 的用法几乎一样：

```
pub static ROOM_TEMPERATURE: f64 = 68.0;      // 华氏温度
```

常量有些类似于 C++ 中的 `#define`，值被编译进代码中每一个使用它的地方。一个静态量就是一个在程序开始之前就初始化并持续到程序退出的变量。在代码中使用常量表示幻数和字符串。使用静态量表示更大规模的数据，或者用于需要借用全局常量的引用时。

没有 `mut` 常量。静态量可以被标记为 `mut`，但就像我们在第 5 章中讨论的一样，Rust 没有任何方法强制实现 `mut` 静态量的独占访问。因此，这种变量天生线程不安全，safe 代码完全不能使用它们：

```
static mut PACKETS_SERVED: usize = 0;

println!("{} served", PACKETS_SERVED); // 错误：使用了可变的静态量
```

Rust 不鼓励全局可变的状态。关于替代方案的讨论，见[全局变量](#)一节。

8.3 将程序变为库

当你的蕨类模拟器完成之后，你发现你需要不止一个程序。假设你还有一个命令行程序运行这个模拟器并把结果保存到文件中。现在，你想要编写其他程序来对保存的结果进行科学分析、实时渲染成长中的植物、渲染逼真的植物等等。所有这些程序都要共享基本的蕨类模拟器的代码。你需要创建一个库。

第一步是将你现有的项目分解为两部分：一个库 crate 和一个可执行文件。前者包含所有的共享代码，后者包含只有命令行程序才需要的代码。

为了展示怎么做到这一点，让我们给出一个简单粗暴的示例程序：

```
struct Fern {
    size: f64,
    growth_rate: f64
}

impl Fern {
    /// 模拟一个蕨类植物一天地成长
    fn grow(&mut self) {
        self.size *= 1.0 + self.growth_rate;
    }
}

/// 运行蕨类模拟器模拟几天的变化
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);
}
```

我们假设这个程序有一个普通的 *Cargo.toml* 文件：

```
[package]
name = "fern_sim"
version = "0.1.0"
authors = ["You <you@example.com>"]
edition = "2018"
```

将这个程序变为库是很简单的。只需要如下步骤：

1. 将文件 *src/main.rs* 重命名为 *src/lib.rs*
2. 给 *src/lib.rs* 里将作为库的公开特性的 item 添加 `pub` 关键字
3. 暂时把 `main` 函数移动到一个别的临时文件中。我们将很快回来处理它。

最后的 *src/lib.rs* 文件看起来像这样：

```
pub struct Fern {
    pub size: f64,
    pub growth_rate: f64
}

impl Fern {
    /// 模拟一个蕨类植物一天地成长
    pub fn grow(&mut self) {
        self.size *= 1.0 + self.growth_rate;
    }
}

/// 运行蕨类模拟器模拟几天的变化
pub fn run_simulation(fern: &mut Fern, days: usize) {
    for _ in 0 .. days {
        fern.grow();
    }
}
```

```

    }
}

```

注意我们不需要更改 *Cargo.toml* 中的任何内容。因为我们的最精简的 *Cargo.toml* 会保持 Cargo 的默认行为。默认情况下，`cargo build` 会查看源代码目录下的文件然后判断要构建什么。当它看到 `src/lib.rs` 文件，它就知道要构建一个库。

`src/lib.rs` 里的代码组成了库的根模块。其他使用我们库的 crate 可以访问根模块里的公有 item。

8.4 src/bin 目录

想让原本的 `fern_sim` 程序再次工作也非常简单：Cargo 有一些内建的支持，可以让较小的程序和库存在同一个 crate 中。

事实上，Cargo 自身就是用这种方式编写的。它的大部分代码都在一个 Rust 库里。我们在本书中至今为止用过的 `cargo` 命令行程序只是一个简单的包装程序，它会调用库来完成真正的工作。库和命令行程序在同一个源代码仓库中。

我们也可以把程序和库放在同一个 crate 中。将以下代码放入 `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);
}

```

这个 `main` 函数就是我们之前放在一边的那个。我们添加了 `use` 声明来使用 `fern_sim` crate 里的一些 item: `Fern` 和 `run_simulation`。也就是说，我们像使用库一样使用了那个 crate。

因为我们把这个文件放在了 `src/bin` 里，所以下次运行 `cargo build` 时，Cargo 将会同时编译 `fren_sim` 库和这个程序。我们可以使用 `cargo run --bin efren` 来运行 `efren` 程序。可以通过使用 `--verbose` 参数来查看 Cargo 运行的命令，大概是这样：

```

$ 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 efren --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
```

我们仍然不需要修改 *Cargo.toml* 中的任何内容，Cargo 的默认行为是查看源文件然后进行判断。它会自动把 *src/bin* 里的 *.rs* 文件当作额外的程序来构建。

我们也可以在 *src/bin* 目录下通过子目录来构建更大的程序。假设我们想要提供另一个程序在屏幕上绘制蕨类植物，但绘制代码比较多而且是模块化的，因此它应该有自己的文件。我们可以将第二个程序放在单独的子目录中：

```
fern_sim/
|-- Cargo.toml
|-- src/
| |-- bin/
|   |-- efern.rs
|   |-- draw_fern/
|     |-- main.rs
|     |-- draw.rs
```

把更大的可执行程序存放在自己的目录中的优势是不会打乱库代码和 *src/bin* 目录。

当然，既然现在 *fern_sim* 是一个库，那么我们还有另一种选择。我们可以把这个程序放入单独的项目中，在一个完全独立的目录中，在它的 *Cargo.toml* 中列出 *fern_sim* 作为依赖：

```
[dependencies]
fern_sim = { path = "../fern_sim" }
```

可能这正是你之后编写其他蕨类模拟程序时采用的方法。*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 item 都可以用属性修饰。属性是 Rust 中向编译器传递指令和建议的语法。例如，假设你得到了如下警告：

```
libgit2.rs: warning: type `git_revspeс` should have a camel case name
such as `GitRevspeс`, #[warn(non_camel_case_types)] on by default
```

但是你选择这个名字是有原因的，你并不希望 Rust 发出警告。你可以通过给类型加上 `#[allow]` 属性来禁用警告：

```
#[allow(non_camel_case_types)]
pub struct git_revspec {
    ...
}
```

条件编译是另一个通过属性来实现的特性，这个属性叫 `#[cfg]`：

```
// 只有当我们在构建Android应用时才在项目中包含这个模块
#[cfg(target_os = "android")]
mod mobile;
```

`#[cfg]` 的完整语法见 [Rust 参考手册](#)，表 8-2 中列出了最常用的选项。

表 8-2: 最常用的 `#[cfg]` 选项

<code>#[cfg]</code> 选项	编译的条件
<code>test</code>	测试模式时（用 <code>cargo test</code> 或者 <code>rustc --test</code> 编译时）
<code>debug_assertions</code>	调试断言开始时（通常在非优化构建中）
<code>unix</code>	为 Unix 系统编译时，包括 macOS
<code>windows</code>	为 Windows 编译时
<code>target_pointer_width = "64"</code>	目标平台是 64 位时。其他可能的值是 "32"
<code>target_arch = "x86_64"</code>	x86-64 架构特定。其他值： "x86", "arm", "aarch64", "powerpc", "powerpc64", "mips"
<code>target_os = "macos"</code>	macOS 特定。其他值： "windows", "ios", "android", "linux", "freebsd", "openbsd", "netbsd", "dragonfly"
<code>feature = "robots"</code>	启用用户自定义特性 "robots" 时（用 <code>cargo build --feature robots</code> 或者 <code>rustc --cfg feature='robots'</code> ）。特性在 Cargo.toml 的 <code>[features]</code> 节中声明。
<code>not(A)</code>	<code>A</code> 不满足时。为了让一个函数有两种不同的实现，可以将其中一个标记为 <code>#[cfg(X)]</code> ，另一个标记为 <code>#[cfg(not(X))]</code> 。
<code>all(A, B)</code>	当 <code>A</code> 和 <code>B</code> 都满足时（等价于 <code>&&</code> ）。
<code>any(A, B)</code>	当 <code>A</code> 或 <code>B</code> 满足时（等价于 <code> </code> ）。

有时，我们需要关心函数的内联展开，这是一种我们通常乐意让编译器实现的一种优化。我们可以使用 `#[inline]` 属性来实现内联：

```
/// 调整相互渗透下，两个相邻细胞中的各种离子的浓度
#[inline]
fn do_osmosis(c1: &mut Cell, c2: &mut Cell) {
    ...
}
```

在一种特殊情况下，如果没有`#[inline]`一定不会发生内联。就是当一个crate中的一个函数或方法在另一个crate中被调用时，除非它是泛型的（有泛型参数）或者被显式标记为`#[inline]`，否则它一定不会被内联展开。

另外，编译器只是把`#[inline]`当作一个建议。Rust也支持`#[inline(always)]`来要求一个函数必须在每一次被调用时内联展开；和`#[inline(never)]`来要求一个函数永远不会被内联展开。

有一些属性，例如`#[cfg]`和`#[allow]`，可以被用于整个模块，然后作用于模块里的所有东西。而另一些，例如`#[test]`和`#[inline]`只能用于单独的item。正如你可能期待的一样，每个属性都是定制的，并且有自己的一组受支持的参数。Rust的参考手册详细列出了所有支持的属性。

为了将一个属性用于整个crate，可以在`main.rs`或者`lib.rs`文件的最前边添加属性，并用`#!`来代替`#`，像这样：

```
// libgit2_sys/lib.rs
#![allow(non_camel_case_types)]

pub struct git_revspec {
    ...
}

pub struct git_error {
    ...
}
```

`#!`告诉Rust将一个属性附加到作用域里的每一个item，而不是下一个item：在这个例子中，`#![allow]`属性作用于整个`libgit2_sys`crate，而不仅仅是`struct git_revspec`。

`#!`还可以在函数体内、结构体定义内等地方使用，但它通常被用在一个文件的开头，来将一个属性附加到整个模块或crate。一些属性总是使用`#!`语法，因为它们只能被用于整个crate。

例如，`#![feature]`属性可以打开Rust语言和库里的`unstable`特性。这些特性都是实验性的，因此可能会有bug或者可能会在将来修改或者删除。例如，正如下面那段代码一样，Rust对追踪宏展开例如`assert!`有实验性的支持，但是因为这个支持是实验性的，因此你必须安装nightly版本的Rust并且显式声明你要使用宏追踪的特性：

```
#![feature(trace_macros)]

fn main() {
    // 我想知道assert_eq!实际上到底被替换成了什么代码
}
```

```

    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 队伍有时会标准化一个实验特性，这样它就会成为语言标准的一部分。然后`#![feature]`特性就会变得多余，Rust 会生成一个警告建议你删除它。

8.6 测试和文档

正如我们在[编写并运行单元测试](#)一节中看到的一样，Rust 中内建了一个简单的单元测试框架。测试是用`#[test]`属性标记的普通函数：

```

#[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

```

(你也可能会看到一些有关“doc-tests”的输出，我们将在稍后讨论它。)

测试的行为和你的 crate 是可执行程序还是库没有关系。你可以通过向 Cargo 传递参数来运行指定的测试：`cargo test math` 会运行名称中包含`math`的所有测试。

测试通常使用 Rust 标准库里的`assert!` 和 `assert_eq!` 宏。若`expr`为真，`assert!(expr)`断言会成功；否则它会 panic，并导致测试失败。`assert_eq!(v1, v2)`类似于`assert!(v1 == v2)`，除了当断言失败时，它会同时打印出两个操作数的值。

也可以在普通的代码中使用这两个宏，但注意即使是 release 构建`assert!` 和 `assert_eq!` 也会被包含进去。使用`debug_assert!` 和 `debug_assert_dq!` 来编写只在 debug 构建时才会检查的断言。

为了测试错误的情况，需要向测试添加`#[should_panic]`属性：

```
/// 正如我们在上一章中说的一样,  
/// 只有当除 0 会导致 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，我们需要添加 `allow` 属性来告诉编译器允许我们写一些可以静态证明一定会 `panic` 的代码，然后进行除法并丢弃结果值。因为通常情况下，编译器会阻止这种愚蠢的代码。

你也可以在测试中返回 `Result<(), E>`。只要错误类型实现了 `Debug` trait（这是通常的情况），你就可以使用 `?Ok` 的情况返回一个 `Result`：

```
use std::num::ParseIntError;  
  
/// 只有当 "1024" 是个有效的数字时这个测试才能通过  
#[test]  
fn main() -> Result<(), ParseIntError> {  
    i32::from_str_radix("1024", 10)?;  
    Ok(())  
}
```

用 `#[test]` 标记的函数会条件编译。普通的 `cargo build` 或者 `cargo build --release` 会跳过测试代码。但当你运行 `cargo test` 时，Cargo 会编译程序两次：一次是用普通方式编译，另一次是在启用测试和测试工具的情况下编译。这意味着如果需要的话，你的单元测试可以和要测试的代码放在一起，这样可以访问内部的实现细节，并且没有运行时开销。然而，这可能会导致一些警告。例如：

```
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));  
}
```

在省略测试代码的构建中，`roughly_equal` 可能未被使用，Rust 会警告：

```
$ cargo build
Compiling math_test v0.1.0 (file:///.../math_test)
warning: function is never used: `roughly_equal`
--> src/crates_unused_testing_function.rs:7:1
|
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，当你的测试足够复杂需要支持的代码的时候，将它们放在一个 `tests` 模块中并使用 `#[cfg]` 属性将整个模块声明为测试模式特定：

```
#[cfg(test)]    // 只有在测试时才包含这个模块
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 的测试工具默认会使用多线程同时运行多个测试，这是你的 Rust 代码线程安全的一个附带的好处。为了禁用并行测试，运行 `cargo test -- --test-threads 1`。（第一个--确保 `cargo test` 会把 `--test-threads` 选项传递给测试的可执行程序）如果想运行单个测试，运行 `cargo test testname`。这意味着从技术上讲，我们在第 2 章中展示的曼德勃罗程序并不是那一章中的第二个多线程程序，而是第三个！编写并运行单元测试一节中运行的 `cargo test` 才是第一个。

通常来说，测试工具只会显示失败的测试的输出。为了显示通过的测试的输出，可以运行 `cargo test -- --no-capture`。

8.6.1 集成测试

你的蕨类模拟器的规模仍然在增长。你决定将所有主要的代码放进一个库里，这样可以被很多可执行程序使用。模拟最终使用 `fern_sim.rlib` 库的用户使用它的方式对它进行一些测试

将是一个好主意。另外，你还有一些首先从二进制文件中加载保存的模拟器的测试，将这些很大的测试文件保存在 `src` 目录中是很尴尬的。集成测试可以帮忙解决这两个问题。

集成测试是 `src` 目录下的 `tests` 目录中的 `.rs` 文件。当你运行 `cargo test` 时，Cargo 会将每一个集成测试编译成独立的 crate，然后和你的库以及 Rust 的测试工具链接。这里有一个例子：

```
// tests/unfurl.rs - Fiddleheads unfurl in sunlight

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，部分原因是它们从 crate 外部看待你的 crate，就像一个用户一样。它们测试 crate 的 public API。

`cargo test` 会同时运行单元测试和集成测试。为了只运行特定文件中的集成测试——例如 `tests/unfurl.rs`——使用命令 `cargo test --test unfurl`。

8.6.2 文档

命令 `cargo doc` 为你的库生成 HTML 文档：

```
$ cargo doc --no-deps --open
Documenting fern_sim v0.1.0 (file:///.../fern_sim)
```

`--no-deps` 选项告诉 Cargo 只为 `fern_sim` 自身生成文档，不要为它依赖的 crate 生成文档。

`--open` 选项告诉 Cargo 稍后在你的浏览器中打开文档。

结果如图 8-2 所示。Cargo 会把新的文档文件保存在 `target/doc` 中，起始页面是 `target/doc/fern_sim/index.html`。

文档从你的库中的 `pub` 特性生成，加上你为它们编写的任何文档注释。我们在这一章中已经看到过一些文档注释了。它们看起来像普通的注释：

```
/// 模拟减数分裂产生孢子的过程。
pub fn produce_spore(factory: &mut Sporangium) -> Spore {
```

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

cells	The simulation of biological cells, which is as low-level as we go.
plant_structures	Higher-level biological structures.
simulation	Overall simulation control.
spores	Fern reproduction.

图 8-2: rustdoc 生成的文档示例

```
    ...
}
```

但当 Rust 看到它们是以三个斜杠开头时，它会把它们当作 `#[doc]` 属性。Rust 将上面的示例看作和下面的代码等价：

```
#[doc = "模拟减数分裂产生孢子的过程。"]
pub fn produce_spore(factory: &mut Sporangium) -> Spore {
    ...
}
```

当你编译一个库或可执行程序时，这些属性并不会改变任何东西，但当你生成文档时，`public` 特性的文档注释将会被包含在输出中。

同样的，以`//!` 开头的注释被当作 `#![doc]` 属性，这个属性会被附加到作用域中，通常是一个模块或者 crate。例如，你的 `fern_sim/src/lib.rs` 文件看起来可能像这样：

```
///! 从独立的细胞开始,
///! 模拟蕨类植物的生长。
```

文档注释的内容被当做 Markdown，一种简单 HTML 格式的速记符号。星号可以用作 `*italics*` 和 `**bold type**`，空行被当作段落结束，等等。你也可以包含 HTML 标签，它们会被逐字符拷贝到格式化文档。

Rust 中的文档注释的一个特殊特性是 Markdown 链接可以使用 Rust 的 item 路径而不是相对 URL 来指明它们引用的 item，例如 `leaves::Leaf`。Cargo 将会查找路径指向的内容，并创建指向正确文档页面中正确位置的链接。例如，这段代码生成的文档会链接到 `VascularPath`、`Leaf` 和 `Root` 的页面：

```
/// 创建并返回一个 `VascularPath` ,
/// 它代表营养从给定的 `Root`[r] 传输到给定的 `Leaf`[leaves::Leaf] 的路径。
///
/// [r]: roots::Root
/// [l]: leaves::Leaf
pub fn trace_path(leaf: &leaves::Leaf, root: &roots::Root) -> VascularPath {
    ...
}
```

你还可以添加搜索别名来使用内建的搜索特性让 item 更容易被找到。在 crate 的文档中搜索“path”或者“route”将会导向 `VascularPath`：

```
#[doc(alias = "route")]
pub struct VascularPath {
    ...
}
```

你可以使用 `backticks` 在文本中间创建代码片段。输出时，这些片段将会用等宽字体显示。更大的代码示例可以通过缩进四个空格来显示：

```
/// 文档注释中的一个代码块
///
/// if samples::everything().works() {
///     println!("ok");
/// }
```

你也可以使用 Markdown-fenced 代码块，效果和上面一样：

```
/// 另一个片段，但是写法不同：
///
/// ``
/// if samples::everything().works() {
///     println!("ok");
/// }
/// ``
```

不管你用哪种方式，当你在文档注释中包含代码块的时候会发生一件有趣的事：Rust 会自动把它当做一个测试。

8.6.3 文档测试

当你运行 Rust 库 crate 中的测试时，Rust 会通过检查确保文档中出现的所有代码都能运行并正常工作。它将文档注释中出现的所有代码作为单独的可执行程序 crate 编译，然后和库链接并运行。

这里有一个文档测试的例子。运行 `cargo new --lib ranges`（`--lib` 参数告诉 Cargo 我们要创建一个库 crate，而不是可执行程序 crate）并将下列代码加到 `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);
///

/// 如果有一个范围是空的，视为不重叠。
///
/// 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
}
```

文档注释中的这两个小代码块将会出现在 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: 文档注释会展示一些文档测试

它们也会成为两个单独的测试：

```
$ 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
```

如果你向 Cargo 传递--verbose 参数，你将会看到它用 rustdoc --test 来运行这两个测试。rustdoc 会把每个示例代码存储在单独的文件中，加上少量的样本代码来生成两个程序。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);
}
```

如果程序能成功编译和运行，那么就可以通过测试。

这两段示例代码都包含了断言，但那只是因为在这种情况下，断言适用于编写文档。文档测试背后的目的不是让你把所有测试都放在注释里，而是让你写出最可能的用法的示例，然后 Rust 确保你文档中的代码示例确实可以编译并运行。

通常一个最小化的能工作的示例还包含一些细节，例如导入或初始化代码。这些是让代码能编译所必需的，但并不重要，不应该在文档中显示。为了隐藏代码示例中的某一行，可以在行的开始加上`#`和一个空格：

```
/// 开启光照，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) {
    ...
}
```

有时在文档中展示一个包含`main`函数的完整的示例程序会很有帮助。如果那些细节代码已经出现在了你的代码示例中，那你显然不会再希望`cargo`自动添加它们，因为这样会导致无法编译。因此`rustdoc`将所有包含字符串`fn main`的代码块视为完整的程序，不会再添加任何东西。

可以为特定的代码块禁用测试。为了告诉 Rust 编译你的示例，但并不真的运行它，可以使用带有`no_run`注解的 fenced 代码块：

```
/// 将所有的本地培养皿上传到在线的 gallery。
///
/// ````no_run
/// let mut session = fern_sim::connect();
/// session.upload_all();
/// ```
pub fn upload_all(&mut self) {
```

```
...  
}
```

如果甚至不希望代码编译，可以将 `no_run` 替换为 `ignore`。使用 `ignore` 标记的块将不会出现在 `cargo test` 的输出中，但 `no_run` 标记的块会出现在测试的输出中，如果能编译就能通过测试。如果代码块完全不是 Rust 代码，那么使用语言的名字来标记。例如 `c++` 或 `sh`，或者普通文本时用 `text`。`rustdoc` 并不知道几百种编程语言的名字，因此，它将所有不认识的注解都当作非 Rust 的代码块，这样会禁用代码高亮和文档测试。

8.7 指定依赖

我们已经看到过一种方式来告诉 Cargo 你的项目依赖的 crate 和版本号。

```
image = "0.6.1"
```

有若干种方式可以指定依赖，并且你可能想问一些类似使用哪个版本的细节问题，因此值得花几页篇幅讨论这个问题。

首先，假设你想使用没有在 crates.io 上发布的依赖。一种方式是指明 Git 仓库的 URL 和修订号：

```
image = { git = "https://github.com/Piston/image.git", rev = "528f19c" }
```

这个 crate 是 GitHub 上的开源仓库，但你也可以指明一个在你自己公司内网里的私有 Git 仓库。正如这里展示的一样，你可以指定要使用的特定的 `rev`、`tag` 或者 `branch`。（这三种方式都可以告诉 Git 要 checkout 到源代码的那个版本。）

另一种方式是指定一个含有 crate 源码的目录：

```
image = { path = "vendor/image" }
```

当你的团队使用单个版本控制仓库来包含若干个甚至整个依赖图的所有 crate 时，这种方式会很方便。每个 crate 都可以使用相对路径指定它的依赖。

这种细粒度的依赖控制方式是一种很强大的功能。如果你发现任何开源的 crate 都不完全符合你的要求，你可以 fork 它：点击 GitHub 上的 Fork 按钮，然后更改 `Cargo.toml` 中的一行。这样你的下一次 `cargo build` 命令将会使用你 fork 的版本而不是官方的版本。

8.7.1 版本

当你在 `Cargo.toml` 中写下类似 `image = "0.13.0"` 这样的内容时，Cargo 会以比较宽松的方式解释它。它会使用和 0.13.0 版本兼容的最新版本。

兼容性规则改编自语义版本号。

- 0.0 开头的版本号太过原始，因此 Cargo 假设它和任意版本都不兼容。
- 一个以 0.x 开头的版本号，其中 x 是非零数字，将会被认为和其他 0.x 系列的版本兼容。¹我们指定了 `image` 的版本是 0.6.1，但 Cargo 有可能会使用 0.6.3 的版本。（这并不是语义版本号标准对 0.x 版本号的说明，但这个规则被证明很有用，因此需要保留。）
- 一旦一个项目到达了 1.0，则只有主版本号会打破兼容性。因此如果你要求 2.0.1 版本，Cargo 可能会使用 2.17.99 版本，但不会是 3.0。

版本号默认是弹性的，因为如果不这样，那么使用哪个版本的问题将会被过度约束。假设库 `libA` 使用了 `num = "0.1.31"`，而库 `libB` 使用了 `num = "0.1.29"`。如果版本号要求精确匹配，将没有任何项目可以同时使用这两个库。允许 Cargo 使用任何兼容的版本是更加实用的默认配置。

不同的项目对依赖和版本有不同的要求。你可以使用运算符指定一个精确的版本号或者一个版本范围，如表 8-3 所示。

表 8-3: 在 `Cargo.toml` 文件中指明版本

<code>Cargo.toml</code> 文件	含义
<code>image = "=0.10.0"</code>	只使用精确的 0.10.0 版本
<code>image = ">=1.0.5"</code>	使用 1.0.5 或者更高的版本（即使是 2.9，如果可用的话）
<code>image = ">1.0.5</code>	使用比 1.0.5 高，但比 1.1.9 低的版本
<code><1.1.9"</code>	
<code>image = "<=2.7.10"</code>	使用任何小于等于 2.7.10 的版本

另一种你偶尔可能会见到的版本指定方式是通配符 *，它告诉 Cargo 可以使用任何版本。除非有别的 `Cargo.toml` 文件中指定了更加具体的版本约束，否则 Cargo 将会使用最新的可用版本。[doc.crates.io 上的 Cargo 文档](#)更加详细地介绍了版本指定的内容。

注意兼容性规则意味着不能纯粹出于营销原因选择版本号。它们必须有真实的含义。这是 crate 的维护者和用户之间的约定。如果你正在维护一个 1.7 版本的 crate，并且你决定删除一个函数或者做出其他不能完全向后兼容的更改，你必须将版本号提升到 2.0。如果你把版本号更新到 1.8，那么你就等于是宣称新版本和 1.7 版本兼容，你的用户可能会发现他们构建时出现错误。

8.7.2 `Cargo.lock`

`Cargo.toml` 中的版本号故意设计的比较灵活，然而我们并不需要每次构建时都让 Cargo 把库更新到最新版本。想象一下你正在一个紧张的调试会话中，突然一次 `cargo build` 把库更

¹译者注：即 `0.x.a` 和 `0.x.b` 兼容，但 `0.x.a` 和 `0.y.a` 不兼容。

新到了最新的版本。这可能会造成很大的破坏，在调试途中任何的改变都会导致问题。事实上，任何时候更新库都不是一个好时机。

Cargo 也有一个内建的机制来解决这个问题。当你第一次构建项目时，Cargo 会输出一个 *Cargo.lock* 文件记录下它使用的每一个 crate 的精确版本。之后的构建将会查询这个文件并继续使用相同的版本。只有当你告诉 Cargo 更新版本时它才会这么做，你可以手动更改 *Cargo.toml* 文件中的版本号或者运行 `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` 只会更新到和你在 *Cargo.toml* 中指定的版本兼容的最新版本。如够你指定了 `image = "0.6.1"`，然后你想更新到 0.10.0，那你必须在 *Cargo.toml* 中更改版本号。当你下一次构建时，Cargo 会更新到 `image` 库的新版本，然后把新的版本号存储在 *Cargo.lock* 中。

上面的示例展示了 Cargo 更新了两个 crates.io 上的 crate。存储在 Git 中的依赖也会发生类似的事情。假设我们的 *Cargo.toml* 文件包含如下内容：

```
image = { git = "https://github.com/Piston/image.git", branch = "master" }
```

如果已经有了 *Cargo.lock* 文件，那么 `cargo build` 将不会从 Git 仓库 pull 新的更改。它会读取 *Cargo.lock* 并且每次都使用和上次相同的版本。但 `cargo update` 将会 pull 最新的 master 分支，因此下一次构建将会使用最新的版本。

Cargo.lock 是自动生成的，通常你不需要手动编辑它。尽管如此，如果你的项目是可执行程序，你应该把 *Cargo.lock* 提交到版本控制系统。这样，任何构建你的项目的人都会使用完全相同的版本。你的 *Cargo.lock* 文件的历史将会记录依赖的更新。

如果你的项目是普通的 Rust 库，不要提交 *Cargo.lock*。你的库的下游用户将会有自己的包含整个依赖图的版本信息的 *Cargo.lock* 文件，他们会忽略你的库中的 *Cargo.lock* 文件。在少数情况下，你的项目可能是一个共享库（即输出是 `.dll`、`.dylib`、`.so` 文件），这时将不会有这种下游的用户，因此你应该提交 *Cargo.lock*。

Cargo.toml 的弹性版本声明让你可以更容易地在自己的项目中使用 Rust 库，并能最大化库之间的兼容性。*Cargo.lock* 的记录可以保持一致性，在不同机器之间复现相同的构建。它们通过协作能有效地帮助你避免依赖地狱。

8.8 把 crate 发布到 crates.io

你已经决定了将你的蕨类模拟器库作为开源软件发布。祝贺你！这个过程非常的简单。

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` 命令会产生一个文件（这里是 `target/package/fern_sim-0.1.0.crate`），这个文件包含你库里的所有源文件，包括 `Cargo.toml`。这是你将要上传到 crates.io 与全世界分享的文件。（你可以使用 `cargo package --list` 来查看包含了哪些文件。）Cargo 会进行二次检查，像你的最终用户一样从 `.crate` 文件构建你的库，来确保它能正常工作。

Cargo 警告说 `Cargo.toml` 的 `[package]` 节缺少了一些对下游用户来说很重要的信息，例如你分发代码所依据的许可证。警告中的 URL 是非常优秀的资源，因此我们不会在这里详细解释所有的字段。简单来说，你可以通过在 `Cargo.toml` 中添加几行内容来修复警告：

```
[package]
name = "fern_sim"
version = "0.1.0"
edition = "2018"
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.
"""

"""

```

NOTE

一旦你在 crates.io 上发布了这个 crate，任何下载了你的 crate 的人都可以看到 `Cargo.toml` 文件。因此如果 `authors` 字段包含了一个你想保持私有的电子邮箱地址，现在是时候修改它了。

这个阶段有时会出现的另一个问题是你的 `Cargo.toml` 文件中可能通过 `path` 指定了别的 crate 的位置，正如指定依赖中展示的一样：

```
image = { path = "vendor/image" }
```

对你和你的团队来说，这可能能正常工作。但显然当其他人下载了 `fern_sim` 库之后，他们的计算机上不会有和你的计算机上一样的文件和目录。因此 Cargo 会忽略自动下载的库中的 `path` 字段，这可能会导致构建错误。然而解决方案也是很直观的：如果你打算在 crates.io 上发布你的库，那么它的依赖也应该发布在 crates.io 上。将 `path` 替换为版本号来指定依赖：

```
image = "0.13.0"
```

如果你喜欢的话，你可以既指定一个 `path` 用于你自己本地的构建，同时为其它用户指定一个 `version`：

```
image = { path = "vendor/image", version = "0.13.0" }
```

当然，这种情况下保持两种版本的同步是你自己的责任。

最后，在发布一个 crate 之前，你需要登录进 crates.io 并获取一个 API key。这一步也很直观：当你在 crates.io 上创建了账号之后，你的“Account Settings”页面将会显示一个 `cargo login` 命令，像这样：

```
$ cargo login 5j0dV54Bj1XBpUUbFIj7G9DvNl1vsWW1
```

Cargo 会把 key 保存在一个配置文件中，API key 应该像密码一样保持私密。因此最好只在你自己的计算机上运行这个命令。

这些都完成之后，最后的步骤是运行 `cargo publish`：

```
$ cargo publish
  Updating registry `https://github.com/rust-lang/crates.io-index`
  Uploading fern_sim v0.1.0 (file:///.../fern_sim)
```

完成之后，你的库就加入了 crates.io 上的其他数以千计的 crate 中。

8.9 工作空间

随着项目的持续增长，你最终会编写很多 crate。它们在一个单独的源代码仓库中逐个排列：

```
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 工作的方式是每一个 crate 都有自己的构建目录 `target`，每一个 `target` 目录都包含了这个 crate 的所有依赖的单独的构建。这些构建是完全独立的。即使两个 crate 有相同的依赖，它们也不会共享任何编译生成的代码。这是一种浪费。

你可以通过使用 Cargo 工作空间来节省编译时间和磁盘空间，一些 crate 可以共享一个公共的构建目录和 `Cargo.lock` 文件。

你需要做的所有事就是在仓库的根目录创建一个 `Cargo.toml` 文件，然后添加几行内容：

```
[workspace]
members = ["fern_sim", "fern_img", "fern_video"]
```

这里 `fern_sim` 等是包含你的 crate 的子目录的名称。然后删除这些子目录里的 `Cargo.lock` 文件和 `target` 目录。

修改完之后，在任何 crate 中运行 `cargo build` 将会自动创建并使用根目录下的共享构建目录（这个例子中是 `fernsoft/target`）。命令 `cargo build --workspace` 会构建当前工作空间内的所有 crate。`cargo test` 和 `cargo doc` 也接受 `--workspace` 选项。

8.10 更多有趣的事

如果你还不满意，Rus 社区还为你准备了更多东西：

- 当你在[crates.io](#)上发布了开源的 crate 之后，你的文档将会被自动呈现并托管在 `doc.rs` 上，这要感谢 Onur Aslan。
- 如果你的项目是在 GitHub 上，Travis CI 可以在你每一次提交时构建并测试你的代码。它非常容易部署，详情见[travis-ci.org](#)。如果你已经对 Travis 很熟悉了，这个 `.travis.yml` 文件可以帮助你上手：

```
language: rust
```

```
rust:
```

```
- stable
```

- 你可以从你的 crate 的顶层文档注释生成一个 *README.md* 文件。这个特性由 Livio Ribeiro 以第三方 Cargo 插件的形式提供。运行 `cargo install cargo-readme` 来安装这个插件，运行 `cargo readme --help` 来学习如何使用它。

我们可以继续了。

Rust 是一门新语言，但它设计的目的之一是支持大型的、复杂的项目。它有强大的工具和活跃的社区。系统程序员将迎来春天。

Chapter 9

结构体

Long ago, when shepherds wanted to see if two herds of sheep were isomorphic, they would look for an explicit isomorphism

——John C. Baez and James Dolan, “[Categorification](#)”

Rust 的结构体，有时也称为 *structure*，类似于 C 和 C++ 中的 `struct` 类型、Python 中的 `class`、JavaScript 中的对象。一个结构体把多个不同类型的值组合成单个值，所以你可以将它们作为一个整体进行处理，你也可以读取并且修改结构体的各个组成部分。一个结构体也可以有一些关联的方法来操作它的组成部分。

Rust 有三种类型的结构体：命名字段 (*name-field*)、类元组 (*tuple-like*)、类单元 (*unit-like*)，它们的区别在于如何引用它们的组成部分：一个命名字段结构体给每一个组件取了一个名字，而类元组结构体用它们出现的顺序来标识它们。类单元结构体没有任何组成部分，它并不常见，但可能比你想象中的更加有用。

在本章中我们将详细解释每一种结构体，并展示它们在内存中的布局。我们将介绍如何给它们添加方法、如何定义可以处理很多不同类型组件的泛型结构体、以及如何让 Rust 为你的结构体生成通用的 trait 的实现。

9.1 命名字段结构体

一个命名字段结构体的定义类似于这样：

```
//> 一个8位灰度像素的矩形
struct GrayscaleMap {
    pixels: Vec<u8>,
    size: (usize, usize)
}
```

这里声明了一个结构体类型 `GrayscaleMap`，它有两个字段分别命名为 `pixels` 和 `size`。Rust 的一个习惯是所有的类型包括结构体，名称中的每一个单词的首字母大写，例如 `GrayscaleMap`，这种习惯称为大驼峰命名法（或帕斯卡命名法）。字段和方法名都是小写，用下划线分隔每个单词。这被称为蛇形命名法。

你可以用结构体表达式构造一个这种类型的值，例如：

```
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开始 (`GrayscaleMap`)，然后在花括号中列出每一个字段的名称和值。还有一种缩写可以用同名的局部变量来充当字段：

```
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 }` 的缩写。你也可以在使用同名字段缩写的同时使用 `key: value` 语法为其它字段赋值。

访问一个结构体的字段需要使用熟悉的 `.` 运算符：

```
assert_eq!(image.size, (1024, 476));
assert_eq!(image.pixels.len(), 1024 * 576);
```

和其他 item 一样，结构体默认是私有的，只在它们声明的模块及其子模块中可见。你可以通过在定义前加 `pub` 来让结构体在模块之外也可见。它的每一个字段也是这样，默认也是私有的：

```
/// 一个 8 位灰度像素的矩形
pub struct GrayscaleMap {
    pub pixels: Vec<u8>,
    pub size: (usize, usize)
}
```

即使结构体被声明为 `pub`，它的字段也可以是私有的：

```
/// 一个 8 位灰度像素的矩形
pub struct GrayscaleMap {
    pixels: Vec<u8>,
```

```
    size: (usize, usize)  
}
```

其他的模块可以使用这个结构体和它的所有公有的关联函数，但不能通过字段名访问私有的字段，也不能通过结构体表达式创建新的 `GrayscaleMap` 值。也就是说，创建一个结构体的值要求所有的结构体字段都可见。这也是为什么你不能通过结构体表达式创建新的 `String` 或者 `Vec`。这些标准类型都是结构体，但它们的字段全都是私有的。要想创建一个这些类型的值，你必须使用公有的类型关联函数，例如 `Vec::new()`。

在创建一个命名字段结构体值的时候，你可以使用另一个相同类型的结构体来提供你省略的字段的值。在一个结构体表达式中，如果命名字段最后跟着一个`.. EXPR`，那么没有提到的字段将从 `EXPR` 中获取值，`EXPR` 必须是另一个相同类型的值。假设我们有一个代表游戏中的怪物的结构体：

```
// 在这个游戏中，连扫帚都是怪物。  
  
struct Broom {  
    name: String,  
    height: u32,  
    health: u32,  
    position: (f32, f32, f32),  
    intent: BroomIntent  
}  
  
/// 一个`Broom`的工作状态有两种可能。  
#[derive(Copy, Clone)]  
enum BroomIntent { FetchWater, DumpWater }
```

对程序员来说最好的童话是魔法师的学徒：一个魔法师学徒制造了一把能替他工作的扫帚，但工作完成之后却不知道该如何停止它。用斧头把扫帚劈成两半会产生两把扫帚，每个只有一半大小，但仍然像之前一样盲目地继续工作：

```
// 以值接受输入的扫帚，会获取所有权  
fn chop(b: Broom) -> (Broom, Broom) {  
    // 用`b`初始化`broom1`的大部分，只修改`height`。因为  
    // `String`不是`Copy`，因此`broom1`会获取`b`的`name`的所有权。  
    let mut broom1 = Broom { height: b.height / 2, .. b };  
  
    // 用`broom1`初始化`broom2`的大部分。因为`String`不是  
    // `Copy`，所以我们必须显式克隆`name`。  
    let mut broom2 = Broom { name: broom1.name.clone(), .. broom1 };  
  
    // 给两半分别起不同的名字。
```

```

        broom1.name.push_str(" I");
        broom2.name.push_str(" II");
        (broom1, broom2)
    }
}

```

这个定义完成之后，我们可以创建一个扫帚，将它劈成两半，然后我们会得到：

```

let hokey = Broom {
    name: "Hokey".to_string(),
    height: 60,
    width: 100,
    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 hokey1 和 hokey2 扫帚接收到了新的调整之后的名字，高度减半，其他的值都和原来一样。

9.2 类元组结构体

第二种结构体类型称为类元组结构体，因为它类似一个元组：

```
struct Bounds(usize, usize);
```

你可以像构建元组一样构造一个这种类型的值，除了必须要包含结构体的名字：

```
let image_bounds = Bounds(1024, 768);
```

类元组结构体持有的值被称为元素，就像元组持有的值一样。你可以像访问元组的元素一样访问它们：

```
assert_eq!(image_bounds.0 * image_bounds.1, 786432);
```

每一个类元组结构体的元素都可以是公有的或者私有的：

```
pub struct Bounds(pub usize, pub usize);
```

表达式 `Bounds(1024, 768)` 看起来像一个函数调用，实际上它就是：定义这个类型也会隐式地定义一个同名函数：

```
fn Bounds(elem0: usize, elem1: usize) -> Bounds { ... }
```

在底层，命名字段结构体和类元组结构体非常相似。到底用哪一个取决于可读性、二义性和简洁性。如果你将频繁使用`.`运算符来获取值的组成部分，那么通过名称来标识字段会增强可读性，也更不容易写错。如果你通常用模式匹配来获取元素，那么类元组结构体可以漂亮地完成工作。

类元组结构体常用于新类型，这种结构体只有单个组件，可以用来获得更严格的类型检查。例如如果你在处理只有 ASCII 的文本，你可以定义一个这样的新类型：

```
struct Ascii(Vec<u8>);
```

使用这种类型表示 ASCII 字符串比简单的传递 `Vec<u8>` 缓冲区好得多，还可以在注释中表明这个类型到底是什么含义。新类型可以帮助 Rust 捕获其他字节缓冲区被传给期望 ASCII 文本的函数的错误。我们将在[第 22 章](#)中给出一个使用新类型来实现高效的类型转换的例子。

9.3 类单元结构体

第三种结构体有一点迷惑：它声明了一个没有任何元素的结构体类型：

```
struct Onesuch;
```

一个这种类型的值不占用任何内存，类似于单元类型 `()`。Rust 不需要考虑怎么在内存中存储类单元结构体，也不需要生成操作它们的代码，因为它可以仅从其类型中得知它可能需要了解的有关值的所有信息。但从逻辑上讲，一个空的结构体和其他的有值的类型没有什么区别——或者更精确地说，一个这样的类型就是一个单独的值：

```
let o = Onesuch;
```

当在[字段和元素](#)一节中介绍`..` 运算符时，你已经遇到过一个类单元结构体了。表达式 `3..5` 是结构体值 `Range { start: 3, end: 5 }` 的缩写，而表达式 `..` 两端都省略的情况下，就是类单元结构体值 `RangeFull` 的缩写。

当和 trait 一起使用时，类单元结构体会变得很有用。我们将在[第 11 章](#)中介绍 trait。

9.4 结构体布局

在内存中，命名字段结构体和类元组结构体是同样的东西：一些可能不同类型的值的集合，以一种特殊的方式分布在内存中。例如，我们之前在这一章中定义的这个结构体：

```
struct GrayscaleMap {
    pixels: Vec<u8>,
    size: (usize, usize)
}
```

一个 GrayscaleMap 按照如图 9-1 的布局分布在内存中。

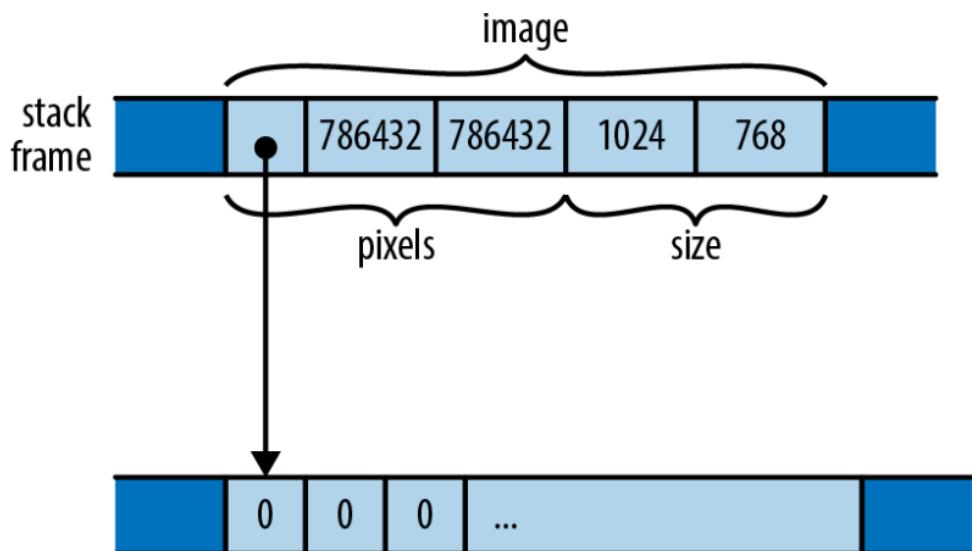


图 9-1: 一个 GrayscaleMap 结构体的内存布局

与 C 和 C++ 不同，Rust 对于如何在内存中布局结构体字段或元素没有任何确切的保证，这里的图只是展示了其中一种可能的排列。然而 Rust 确实保证了直接在结构体的内存块里存储字段的值。JavaScript、Python 和 Java 将会把 pixels 和 size 值分别放在它们自己的堆上分配的内存块中，然后让 GrayscaleMap 的字段指向他们。而 Rust 会直接把 pixels 和 size 嵌入到 GrayscaleMap 值里。只有 pixels vector 持有的堆上分配的缓冲区保留着自己的块。

你可以使用 `#[repr(C)]` 属性来要求 Rust 用一种与 C 和 C++ 兼容的方式布局结构体。我们将会在第 23 章中详细介绍这些。

9.5 使用 `impl` 定义方法

在本书中，我们已经在很多类型的值上调用过方法。我们使用 `v.push(e)` 把元素添加进 `vector`，使用 `v.len()` 获取它的长度，使用 `r.expect("msg")` 检查 `Result` 的值是否是错误

值，等等。你可以为自己的结构体类型定义方法。与 C++ 或 Java 那种直接出现在结构体定义内部的方式不同，Rust 的方法在一个单独的 `impl` 块中定义。

一个 `impl` 块就是一些 `fn` 定义的结合，每一个函数都将成为这个结构体的一个方法。例如，这里我们定义了一个公有的结构体 `Queue`，然后为它定义了两个公有的方法：`push` 和 `pop`：

```
/// 一个先进先出的字符队列
pub struct Queue {
    older: Vec<char>, // 旧的元素，越旧越靠后
    younger: Vec<char> // 新的元素，越新越靠后
}

impl Queue {
    /// 将一个字符添加到队列的尾部。
    pub fn push(&mut self, c: char) {
        self.younger.push(c);
    }

    /// 移出队列最前端的元素，如果有字符被移出就返回`Some(c)`，
    /// 否则如果队列为空就返回`None`。
    pub fn pop(&mut self) -> Option<char> {
        if self.older.is_empty() {
            if self.younger.is_empty() {
                return None;
            }
        }

        // 将新的元素都移动到旧的元素里,
        // 并且反转顺序。
        use std::mem::swap;
        swap(&mut self.older, &mut self.younger);
        self.older.reverse();
    }

    // 现在旧的元素保证不为空。Vec 的 pop 方法
    // 已经返回了一个Option，因此不用再处理
    self.older.pop()
}
}
```

在 `impl` 块中定义的函数被称为关联函数，因为它们被关联到特定的类型。与之相反的是自由函数，也就是不在 `impl` 块中定义的函数。

Rust 将调用方法的值作为第一个参数传给方法，它的参数名必须是 `self`。因为 `self` 的类

型很明显是 `impl` 块外面的结构体类型，或者是其引用，所以 Rust 允许你省略类型，用 `self`、`&self`、`&mut self` 分别作为 `self: Queue`、`self: &Queue`、`self: &mut Queue` 的缩写。如果你喜欢的话也可以使用非缩写的形式，但几乎所有的 Rust 代码都是用缩写形式。

在我们的示例中，`push` 和 `pop` 方法中通过 `self.older` 和 `self.younger` 引用了 `Queue` 的字段。与 C++ 或 Java 中 “this” 对象的成员直接在方法内可见不同，Rust 方法中必须显式使用 `self` 来引用字段，这和 Python 方法中 `self` 的用法、JavaScript 方法中 `this` 的用法类似。

因为 `push` 和 `pop` 需要修改 `Queue`，所以它们都以 `&mut self` 传参。然而，当你调用这两个方法时，你不需要手动借用可变引用，普通的方法调用语法会自动进行隐式转换。因此有了这些定义之后，你可以像这样使用 `Queue`：

```
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);
```

为了满足 `push` 方法的 `self` 参数要求，直接写 `q.push(...)` 会借用 `q` 的可变引用，就像你写了 `(&mut q).push(...)` 一样。

如果一个方法不需要修改 `self`，那么你可以以共享引用获取参数。例如：

```
impl Queue {
    pub fn is_empty(&self) -> bool {
        self.older.is_empty() && self.younger.is_empty()
    }
}
```

方法调用表达式知道要借用哪一种引用：

```
assert!(q.is_empty());
q.push(' ');
assert!(!q.is_empty());
```

或者，如果一个方法想获取 `self` 的所有权，它可以以值传递 `self`：

```
impl Queue {
    pub fn split(self) -> (Vec<char>, Vec<char>) {
        (self.older, self.younger)
```

```

    }
}

```

调用这个 `split` 方法看起来和调用其他方法一样：

```

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']);

```

但是注意，因为 `split` 以值获取 `self`，这会把 `q` 中的 `Queue` 值移动走，导致 `q` 变为未初始化。因为 `split` 的 `self` 现在拥有了这个队列，因此它可以把两个单独的 `vector` 移动出来并返回给调用者。

有时，像这样以值传递 `self`，或者以引用传递都不能满足我们的需求，因此 Rust 还允许你通过智能指针类型传递 `self`。

9.5.1 以 `Box`、`Rc`、`Arc` 传递 `Self`

一个方法的 `self` 参数还可以是 `Box<Self>`、`Rc<Self>`、`Arc<Self>`。这些方法只能在相应指针类型上调用。调用这些方法会传递指针的所有权。

你通常不需要这么做。一个以引用传递 `self` 的方法可以在任何智能指针类型上正常调用：

```

let mut bq = Box::new(Queue::new());

// `Queue::push` 接受一个`&mut Queue`，但`bq` 是`Box<Queue>`。
// 这没有问题：Rust 在调用期间从`Box` 借用了一个`&mut Queue`。
bq.push(' ');

```

对于方法调用和字段访问，Rust 自动从智能指针类型例如 `Box`、`Rc` 和 `Arc` 借用一个引用，因此 `&self` 和 `&mut self` 通常总是正确的方法签名，再加上偶尔用到的 `self`。

但如果这个方法的意图涉及管理指针的所有权呢？假设我们有一个像这样的节点组成的树，类似某种彻底简化的 XML：

```
use std::rc::Rc;
```

```

struct Node {
    tag: String,
    children: Vec<Rc<Node>>
}

impl Node {
    fn new(tag: &str) -> Node {
        Node {
            tag: tag.to_string(),
            children: vec![],
        }
    }
}

```

每一个节点都有一个 tag 来指示它是什么类型的节点，还有一个子节点的 vector 通过引用计数指针来允许共享，并让生命周期变得更灵活。

通常我们会实现一个方法让它向自己的列表中添加一个子节点，但现在让我们把角色颠倒过来，给 Node 实现一个把它自己添加到别的 Node 的子节点中的方法。我们可以写：

```

impl Node {
    fn append(self, parent: &mut Node) {
        parent.children.push(Rc::new(self));
    }
}

```

但这个方法并不能让人满意。这个方法调用了 **Rc**::new 来分配新的堆空间并且把 self 移动进去，但如果调用者已经有了一个 **Rc**<Node>，这些操作就都不是必须的：我们应该只递增引用计数然后把指针加到 vector 里。**Rc** 的全部意义不就是实现共享吗？

我们可以这样写：

```

impl Node {
    fn append_to(self: Rc<Self>, parent: &mut Node) {
        parent.children.push(self);
    }
}

```

如果调用者已经是 **Rc**<Node> 类型，那它可以直接调用 **append_to**，以值传递 **Rc**:

```

let shared_node = Rc::new(Node::new("first"));
shared_node.append_to(&mut parent);

```

这会把 **shared_node** 的所有权传递给方法：引用计数不会发生变化，也不会有新的内存分配。

如果调用者需要保留节点的指针以便之后使用，它可以首先克隆 `Rc` 再调用：

```
shared_node.clone().append_to(&mut parent);
```

克隆 `Rc` 只会递增引用计数，没有堆分配或者拷贝。但当调用返回时 `shared_node` 和 `parent` 的子节点的 `vector` 现在指向同一个 `Node`。

最后，如果调用者现在拥有一个 `Node`，那么它必须先创建一个 `Rc` 再调用方法：

```
let owned = Node::new("owned directly");
Rc::new(owned).append_to(&mut parent);
```

把 `append_to` 方法的签名设为 `Rc<Self>` 可以让调用者知道 `Node` 的需求。然后调用者可以用最小化内存分配和引用计数调整的方式来调用：

- 如果可以传递 `Rc` 的所有权，就直接在指针上调用。
- 如果需要保留 `Rc` 的所有权，就递增引用计数后再调用。
- 如果只拥有 `Node`，那么必须先调用 `Rc::new` 来分配堆空间然后把 `Node` 移动进去。因为 `parent` 必须通过 `Rc<Node>` 指针引用它的子节点，所以这一步最终肯定是必须的。

再重复一遍，对于大多数方法，`&self`、`&mut self`、和 `self`（以值传参）就能满足你的需求。但如果一个方法的目的是影响值的所有权，使用其他指针类型的 `self` 可能是正确的选择。

9.5.2 类型关联函数

`impl` 块里定义的函数也可以没有 `self` 参数。这些参数仍然和类型关联，因为它们也是在 `impl` 块中定义的。但它们不是方法，因为它们没有 `self` 参数。为了将它们和方法区别开来，我们称它们为类型关联函数。

它们通常用于提供构造函数，例如：

```
impl Queue {
    pub fn new() -> Queue {
        Queue { older: Vec::new(), younger: Vec::new() }
    }
}
```

为了使用这个函数，我们通过 `Queue::new` 来引用它：类型名 + 双冒号 + 函数名。现在我们的示例代码变得更加简洁：

```
let mut q = Queue::new();
q.push('*');
...
```

Rust 的传统是构造函数都叫 new，我们已经见过了 Vec::new、Box::new、HashMap::new，等等。但 new 这个名字本身并没有什么特殊的地方。它并不是关键字，而且一个类型经常还有其他名字的关联函数作为构造函数，例如 Vec::with_capacity。

尽管一个类型可以有很多个分开的 impl 块，但它们必须在定义类型的那个 crate 中。然而，Rust 确实允许你将自己定义的方法附加到其他类型，我们将在第 11 章介绍怎么做到这一点。

如果你习惯写 C++ 或 Java，你可能会觉得将类型的方法和定义分离开来很奇怪，但这么做确实有以下优势：

- 你总是能很容易的找到一个类型的数据成员。在很大的 C++ 类定义中，你可能需要浏览几百行成员成员函数的定义来确保你没有遗漏数据成员。而在 Rust 中，所有数据成员都在一个地方。
- 尽管我们可以很容易想象把方法定义添加到命名字段结构体的定义中，但类元组结构体和类单元结构体却不是这样。将方法拿出来放在一个 impl 块中可以让这三种结构体共用唯一一种语法。事实上，Rust 还使用这套语法为不是结构体的类型定义方法，例如 enum 结构体和基本类型例如 i32。（任何类型都可以有方法的事实是 Rust 中不使用术语对象，而是更喜欢把一切称为值的原因之一。）
- 同样的 impl 语法还可以很容易地用于实现 trait，我们将在第 11 章中介绍。

9.6 关联常量

Rust 的类型系统还采用了 C# 和 Java 等语言中的一个特性，就是关联到类型而不是关联到类型实例的值。在 Rust 中，它们被称为关联常量。

正如它的名字一样，关联常量是常量值。它们通常用来表示某一个类型中使用最广泛的值。例如，你可以定义一个二维的向量用于线性代数，并为它定义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 };
}
```

这些值被关联到类型本身，你可以在不创建 Vector2 的实例的情况下使用它们。和类型关联方法一样，引用它们的方法是类型的名称再加上它们的名称：

```
let scaled = Vector2::UNIT.scaled_by(2.0);
```

关联常量的类型并不一定必须是它关联的类型，我们可以使用这个特性来给类型添加 ID 或者名称。例如，如果有几个和 `Vector2` 很像的类型需要写入到文件里，然后加载到内存中，那么关联常量可以为写入的数据添加名称或数字 ID，这样之后可以据此辨识出类型：

```
impl Vector2 {
    const NAME: &'static str = "Vector2";
    const ID: u32 = 18;
}
```

9.7 泛型结构体

我们之前的 `Queue` 定义并不能令人满意：它被用来存储字符，但它却没有任何和字符相关的方法。如果我们要定义另一个存储 `String` 值的结构体，那么除了 `char` 要换成 `String` 之外，剩下的代码完全相同。重复定义将会浪费时间。

幸运的是，Rust 的结构体可以是泛型的，这意味着它们的定义只是一个模板，你可以把任何类型塞进去。例如，这里有一个可以存储任何类型的值的 `Queue` 的定义：

```
pub struct Queue<T> {
    older: Vec<T>,
    younger: Vec<T>
}
```

你可以将 `Queue<T>` 中的 `<T>` 读作“对于任意类型 `T`”。因此这个定义读作“对于任意类型 `T`，一个 `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>`。在 `Queue<char>` 中 `T` 是 `char`，我们就得到了一个和一开始的 `char` 类型特定的定义完全相同的结构体。事实上，`Vec` 本身就是一个用这种方式定义的泛型结构体。

在泛型结构体的定义中，<尖括号>里的类型名被称为类型参数。一个泛型结构体的 `impl` 块看起来像这样：

```
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>` 这一行读作“对于任意类型 `T`, 有一些 `Queue<T>` 可用的关联方法”。然后, 你可以在关联函数的定义中将类型参数 `T` 用作一个类型。

这种语法看起来好像有些重复, 但 `impl<T>` 能更清晰地表达出 `impl` 块覆盖了任意的类型 `T`, 这能将它和为特定种类的 `Queue` 编写的 `impl` 块区分开来, 例如:

```

impl Queue<f64> {
    fn sum(&self) -> f64 {
        ...
    }
}

```

这个 `impl` 块头读作“这里有一些 `Queue<f64>` 特定的关联方法”。这样就给 `Queue<f64>` 添加了一个 `sum` 方法, 而其它种类的 `Queue` 并没有这个方法。

我们已经在之前的代码中使用过 Rust 的 `self` 参数的缩写形式。这里 `Queue<T>` 如果写出类型显得有些冗杂。作为另一种缩写, 每一个 `impl` 块, 不管是不是泛型的, 都定义了特殊的类型参数 `Self` (注意大驼峰风格的名称) 来表示我们要添加方法的类型。在之前的代码中, `Self` 将是 `Queue<T>`, 因此我们可以进一步简化 `Queue::new` 的定义:

```

pub fn new() -> Self {
    Queue { older: Vec::new(), younger: Vec::new() }
}

```

你可能已经注意到了, 在 `new` 的函数体内, 我们不需要在构造表达式中写出类型参数, 简单地写 `Queue { ... }` 就足够了。这是因为 Rust 的类型推导发挥了作用: 因为函数的返回值类型只能有一种可能, 就是 `Queue<T>`, 所以 Rust 会自动为我们提供参数。然而, 你必须总是在函数签名和类型定义中指明类型参数。Rust 不会推断它们, 相反, 它将这些已知的类型作为基础来推断那些函数体内的类型。

`Self` 也可以用于这种方式: 我们可以写 `Self { ... }`。你觉得哪种形式更容易理解就可以用哪种。

对于关联函数的调用, 你可以显式地使用`::<>(涡轮鱼)`注解来提供类型参数:

```
let mut q = Queue::<char>::new();
```

但在实践中，通常可以让 Rust 自动帮你推断类型：

```
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");      // 比特币/USD, 2019 年 6 月
r.push(13764.0);
```

事实上，这正是整本书中我们使用 `Vec` 的方式。

并不只有结构体可以是泛型的。枚举也可以有类型参数，语法也非常的相似。我们将在[枚举](#)一节中详细介绍。

9.8 有生命周期参数的结构体

正如我们在[包含引用的结构体](#)一节中讨论过的一样，如果一个结构体类型中包含引用，你必须指明那些引用的生命周期。例如，这里有一个存储切片中最大和最小元素的引用的结构体：

```
struct Extrema<'elt> {
    greatest: &'elt i32,
    least: &'elt i32
}
```

之前，我们建议你将类似 `struct Queue<T>` 的声明看作是对于任意类型 `T`，你都可以构建一个 `Queue<T>` 来存储该类型。类似的，你可以将 `struct Extrema<'elt>` 看作对于任意生命周期 `'elt`，你都可以构建一个 `Extrema<'elt>` 来存储生命周期为 `'elt` 的引用。

这里有一个函数来扫描一个切片并返回一个指向其中元素的 `Extrema` 值：

```
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 }
}
```

这里，`find_extrema` 借用了 `slice` 的元素。因为 `slice` 的生命周期是 '`s`'，所以我们返回的 `Extrema` 结构体中的引用的生命周期也是 '`s`'。Rust 总是会为调用推断生命周期参数，因此调用 `find_extrema` 并不需要提到它们：

```
let a = [0, -3, 0, 15, 48];
let e = find_extrema(&a);
assert_eq!(*e.least, -3);
assert_eq!(*e.greatest, 48);
```

因为返回值和参数使用相同的生命周期非常常见，所以当只有一种可能时 Rust 允许我们省略生命周期。我们还可以在不改变含义的情况下将 `find_extrema` 的签名写成这样：

```
fn find_extrema(slice: &[i32]) -> Extrema {
    ...
}
```

诚然，我们也可能是想返回 `Extrema<'static>`，但这并不是通常的情况。Rust 只为通常的情况提供了一个缩写。

9.9 为结构体类型派生常见的 trait

可以非常简单地定义一个结构体：

```
struct Point {
    x: f64,
    y: f64
}
```

然而，如果你真的开始使用这个 `Point` 类型，你将很快发现它使用起来很痛苦。按照上面的写法，`Point` 既不能拷贝也不能克隆。你不能用 `println!("{}: {}", point)` 打印它，它也不支持 `==` 和 `!=` 运算符。

在 Rust 中这些特性都有一个名字——`Copy`、`Clone`、`Debug`、`PartialEq`。它们被称为 *trait*。在第 11 章中，我们将展示如何为自己的结构体手动实现 trait。但这个例子中出现的标准 trait，以及其它一些 trait，你不需要手动实现它们，除非你想要一些自定义行为。Rust 可以自动为你实现它们。只需要为结构体添加 `#[derive]` 属性：

```
#[derive(Copy, Clone, Debug, PartialEq)]
struct Point {
    x: f64,
    y: f64
}
```

这些 trait 中的任何一个都可以自动实现，前提是这个结构体的所有字段都实现了那个 trait。我们可以要求 Rust 为 `Point` 派生 `PartialEq`，是因为它的两个字段都是 `f64` 类型，这个类型已经实现了 `PartialEq`。

Rust 还可以派生 `PartialOrd`，它可以支持比较运算符 `<`、`>`、`<=` 和 `>=`。我们没有这么做，是因为比较两个点来判断其中一个是否“小于”另一个是一件很奇怪的事。因此我们选择让 `Point` 值不支持那些比较运算符。像这样的情况也是 Rust 让我们自己写 `##[derive]` 属性而不是直接自动派生所有可用的 trait 的原因之一。另一个原因是实现一个 trait 将自动变为公有的特性。因此可复制性、可克隆性等都将是结构体公共 API 的一部分，应该慎重选择。

我们将在第 13 章详细介绍 Rust 的标准 trait 并解释哪些可以用 `##[derive]` 自动实现。

9.10 内部可变性

可变性和其他东西一样：如果限制太过宽松就会导致问题，但你又总是希望它能放宽一点。例如，假设你的蜘蛛机器人控制系统是一个中心化的结构体 `SpiderRobot`，这个结构体里包含设置和 I/O 处理。当机器人启动时它的值就被初始化，并且永远不会改变：

```
pub struct SpiderRobot {  
    species: String,  
    web_enabled: bool,  
    leg_devices: [fd::FileDesc; 8],  
    ...  
}
```

机器人的每一个主要系统都由一个不同的结构体处理，这些结构体都有一个指向那个 `SpiderRobot` 值的指针：

```
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,  
    ...  
}
```

控制织网、捕食、毒液流动等的结构体都有一个 `Rc<SpiderRobot>` 智能指针。回顾一下，`Rc` 代表引用计数，`Rc` 中的值总是共享的因此总是不可变的。

现在假设你想使用标准的 `File` 类型给 `SpiderRobot` 结构体添加一点日志功能。那么就会有一个问题：`File` 值必须是 `mut` 的。所有写入它的函数都需要 `mut` 引用。

这种情况经常出现。我们需要的是一个不可变 (`SpiderRobot` 结构体) 的值内有一小部分数据可变 (一个 `File`)。这被称为内部可变性。Rust 为此提供了几种方式，这一节中我们将讨论最直观的两种类型：`Cell<T>` 和 `RefCell<T>`，这两个类型都在 `std::cell` 模块中。

`Cell<T>` 结构体只包含单个私有的类型 `T` 的值。`Cell` 唯一特殊的地方是即使你没有对 `Cell` 自身的 `mut` 访问权限，也可以访问和修改它里面的字段值：

`Cell::new(value)`

创建一个新的 `Cell`，将给定的 `value` 移动进去。

`cell.get()`

返回一个 `cell` 中值的拷贝

`cell.set(value)`

将给定的值 `value` 存储在 `cell` 中，丢弃掉之前存储的值。这个方法以非 `mut` 引用获取 `self` 参数：

```
fn set(&self, value: T) // 注意：不是`&mut self`
```

显然，这和普通的 `set` 方法不同。到目前为止，Rust 告诉我们如果想要修改数据就需要 `mut` 访问权限。但同样的道理，这一处不同的细节就是 `Cell` 全部的精髓。它们是一种安全地扭曲不可变性规则的方式——既不多、也不少。

`Cell` 还可以有一些其他的方法，你可以在[它的文档](#)中查阅。

如果你想给你的 `SpiderRobot` 添加一个简单的计数器，那么使用 `Cell` 将会很方便。你可以写：

```
use std::cell::Cell;

pub struct SpiderRobot {
    ...
    hardware_error_count: Cell<u32>,
    ...
}
```

即使是 `SpiderRobot` 的非 `mut` 方法也可以使用 `.get()` 和 `.set()` 来访问 `u32`：

```
impl SpiderRobot {
    /// 错误计数加一
    pub fn add_hardware_error(&self) {
        let n = self.hardware_error_count.get();
```

```
    self.hardware_error_count.set(n + 1);

}

/// 如果有硬件错误的话返回真
pub fn has_hardware_errors(&self) -> bool {
    self.hardware_error_count.get() > 0
}
}
```

这非常简单，但并不能解决我们的日志问题。`Cell`不能让你调用内含值的`mut`方法，因为`.get()`方法返回的是单元里值的拷贝，所以它只能用于实现了`Copy trait`的类型。为了实现日志功能，我们需要一个可变的`File`，而`File`不可拷贝。

这个例子中正确的工具是`RefCell`。类似于`Cell<T>`，`RefCell<T>`也是一个包含单个`T`类型的值的泛型类型。和`Cell`不同的是，`RefCell`支持借用内部的`T`类型的值：

`RefCell::new(value)`

创建一个新的`RefCell`，把`value`移动进去。

`ref_cell.borrow()`

返回一个`Ref<T>`，它本质上是`ref_cell`里存储的值的一个共享引用。如果里面的值已经有可变借用了，这个方法会panic。

`ref_cell.borrow_mut()`

返回一个`RefMut<T>`，它本质上是`ref_cell`里存储的值的一个可变引用。如果里面的值已经被借用了，这个方法会panic。

`ref_cell.try_borrow()`, `ref_cell.try_borrow_mut()`

与`borrow()`和`borrow_mut()`的功能一样，但返回一个`Result`。当值已经有可变借用/被借用时，它们不panic，而是返回一个`Err`值。

同样的，`RefCell`还有一些其它方法，你可以在[它的文档](#)中查阅。

只有当你尝试打破Rust中`mut`引用必须是独占引用的规则时这两个方法才会panic。例如，这样将会panic：

```
use std::cell::RefCell;

let ref_cell: RefCell<String> = RefCell::new("hello".to_string());
```

```

let r = ref_cell.borrow(); // ok, 返回一个Ref<String>
let count = r.len();      // ok, 返回"hello".len()
assert_eq!(count, 5);

let mut w = ref_cell.borrow_mut(); // panic: 已经被借用了
w.push_str(" world");

```

为了避免 panic，你可以把这两个借用放进单独的块里。这样的话，`r`会在你尝试借用 `w` 之前被 drop。

这和普通引用的工作方式很像。唯一的不同在于普通的情况下，当你尝试借用一个变量的引用，Rust 会在编译期确保你必须安全地使用引用。如果检查失败了，你会得到一个编译期错误。RefCell 使用运行时检查来执行同样的规则。如果你尝试打破规则，你会得到一个 panic（或者使用 `try_borrow` 和 `try_borrow_mut` 时会得到一个 Err）。

现在我们已经准备好用 RefCell 来完善我们的 SpiderRobot 了：

```

pub struct SpiderRobot {
    ...
    log_file: RefCell<File>,
    ...
}

impl SpiderRobot {
    /// 向日志文件中写入一行。
    pub fn log(&self, message: &str) {
        let mut file = self.log_file.borrow_mut();
        // `writeln!` 和 `println!` 很像，
        // 但会把字符串写入到给定的文件中。
        writeln!(file, "{}", message).unwrap();
    }
}

```

变量 `file` 的类型是 `RefMut<File>`。它的使用方式就像 `File` 的可变引用一样。有关写入文件的细节，见第 18 章。

Cell 很容易使用。虽然必须调用 `.get()` 和 `.set()` 或者 `.borrow()` 和 `.borrow_mut()` 有点麻烦，但那只是我们扭曲可变性规则的代价。还有一些不是很明显但却更严重的问题：Cell——和任何包含了它们的类型——都不是线程安全的。因此 Rust 不允许多个线程直接访问它们。我们将在第 19 章中讨论到 `Mutex<T>`、原子量和全局变量时介绍多线程中的内部可变性。

不管一个结构体有命名字段还是类元组结构体，它都是一些其他值的聚合体：如果我有一

个 `SpiderSenses` 结构体，那么我就有一个指向共享的 `SpiderRobot` 结构体的 `Rc` 指针、同时还有了 `eyes`、`accelerometer` 等等。因此结构体的本质是单词 “和 (*and*)”：我有一个 X 和 (*and*) 一个 Y。但是是否有另一种类型用于单词 “或 (*or*)” 呢？换句话说，当你有了一个这种类型的值时，你有了一个 X 或 (*or*) 一个 Y？这样的类型太有用了以至于它们在 Rust 中无处不在，它们就是我们下一章要讨论的主题。

Chapter 10

枚举与模式

Surprising how much computer stuff makes sense viewed as tragic deprivation of sum types (cf. deprivation of lambdas).

——Graydon Hoare

这一章的第一个话题将是一个古老的、强大的、可以帮你在短期内完成很多工作的（有代价地）、并且在许多语言中以不同的名字广为人知的特性。但它并不是魔鬼。而是一种用户自定义的数据类型，它是 ML 和 Haskell 程序员们熟知的和类型、也是互斥的联合、还是代数数据类型。在 Rust 中，它们被称为枚举 (*enumerations*)，或者简写为 *enum*。和魔鬼不同的是，它们非常安全、索取的代价也很小。

C++ 和 C# 都有枚举，你可以使用它们来定义自己的类型，这种类型的取值范围是一些命名常量的集合。例如，你可能定义过一个叫 `Color` 的类型，取值范围为 `Red`、`Orange`、`Yellow` 等等。这种枚举在 Rust 中也能工作，但 Rust 进一步扩展了枚举。一个 Rust 枚举可以包含多种不同类型的数据。例如，Rust 的 `Result<String, io::Error>` 类型就是一个枚举：该类型的值要么是一个包含 `String` 的 `Ok` 值要么是一个包含 `io::Error` 的 `Err` 值。这就超出了 C++ 和 C# 中枚举的能力。它更像 C 中的 `union`——但和联合不同的是，Rust 的枚举是类型安全的。

枚举适用于一个值有多种可能的情况。使用它们的“代价”是你必须使用模式匹配来安全地访问数据，这也是我们这一章中的第二个话题。

如果你使用过 Python 的解包或者 JavaScript 中的解构，那你可能觉得模式也很熟悉，但 Rust 同样扩展了模式。Rust 的模式有点像匹配数据的正则表达式。它们被用来测试一个值是否具有特定的期望的形态。它们可以一次从结构体或这元组中提取出多个字段存入局部变量。并且和正则表达式类似，它们很简洁，通常只用单行代码就能完成任务。

这一章将以枚举的基础开始，展示数据怎么被关联到枚举选项以及枚举是怎么存储在内存中的。然后我们会展示 Rust 的模式和 `match` 表达式如何简洁地指定基于枚举、结构体、数组、切片的逻辑。模式也可以包含引用、`move` 和 `if` 条件，这让它们的功能更加强大。

10.1 枚举

简单的 C 风格枚举非常直观：

```
enum Ordering {
    Less,
    Equal,
    Greater,
}
```

这里声明了一个有三个可能的值的 `Ordering` 类型，这些值被称为 *variant* 或者 *constructor*: `Ordering::Less`、`Ordering::Equal`、`Ordering::Greater`。这个枚举是标准库的一部分，因此 Rust 代码可以导入它：

```
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
    }
}
```

或者它的所有 *constructor*:

```
use std::cmp::Ordering::*;

fn compare(n: i32, m: i32) -> Ordering {
    if n < m {
        Less
    } else if n > m {
        Greater
    } else {
        Equal
    }
}
```

导入 constructor 之后，我们可以写 Less 来代替 Ordering::Less 等，但因为这样不够明显，因此一般认为不要导入它们是更好的风格，除非它能使你的代码可读性更强。

为了导入一个在当前模块中声明的枚举的 constructor，可以使用 self：

```
enum Pet {
    Orca,
    Giraffe,
    ...
}

use self::Pet::*;


```

在内存中，C 风格的枚举值被存储为整数。有时告诉 Rust 使用哪些整数会很有用：

```
enum HttpStatus {
    Ok = 200,
    NotModified = 304,
    NotFound = 404,
    ...
}
```

否则，Rust 会从 0 开始自动分配值。

默认情况下，Rust 用能容纳所有值的最小的内建整数类型来存储 C 风格枚举。大多数情况下都是一个单独的字节：

```
use std::mem::size_of;
assert_eq!(size_of::<Ordering>(), 1);
assert_eq!(size_of::<HttpStatus>(), 2); // 404 不能存储在 u8 中
```

你可以通过添加 #[repr] 属性来覆盖 Rust 选择的内存表示方式。更多的细节见 [寻找公共的数据表示](#)。

将 C 风格的枚举转换为整数是允许的：

```
assert_eq!(HttpStatus::Ok as i32, 200);
```

然而，反过来把整数转换为枚举是不允许的。和 C 和 C++ 不同，Rust 保证枚举的值只能是 enum 生命中列出的值之一。未经检查的从整数类型到枚举类型的转换会打破这种保证，所以它是不允许的。你可以写出你自己的带检查的版本：

```
fn http_status_from_u32(n: u32) -> Option<HttpStatus> {
    match n {
        200 => Some(HttpStatus::Ok),
        304 => Some(HttpStatus::NotModified),
        _ => None,
    }
}
```

```

404 => Some(HttpStatus::NotFound),
...
_ => None,
}
}

```

或者使用`enum_primitive` crate。它包含一个宏可以为你自动生成这种类型的转换代码。

和结构体一样，编译器也可以为枚举自动生成类似`==`运算符这样的特性，但需要显式地要求：

```

#[derive(Copy, Clone, Debug, PartialEq, Eq)]
enum TimeUnit {
    Seconds, Minutes, Hours, Days, Months, Years,
}

```

枚举也和结构体一样可以拥有方法：

```

impl TimeUnit {
    /// 返回该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",
        }
    }

    /// 返回该时间单位的单数名词。
    fn singular(self) -> &'static str {
        self.plural().trim_end_matches('s')
    }
}

```

C风格的枚举就这么多了。Rust中最有趣的一类枚举是那些带有数据的枚举。我们将展示这些枚举如何存储在内存中、如何通过添加类型参数将它们变为泛型的，以及如何通过枚举构建复杂的数据结构。

10.1.1 带有数据的枚举

一些程序总是需要显示完整的日期和时间，并且精确到毫秒。但对于大多数程序，显示大概的时间范围会更加友好，例如“两个月以前”。我们可以用之前定义的枚举编写一个新的枚举来实现这一点：

```
/// 一个故意舍入的时间戳，因此我们的程序会显示“6个月以前”
/// 而不是“February 9, 2016, at 9:49 AM”。
#[derive(Copy, Clone, Debug, PartialEq)]
enum RoughTime {
    InThePast(TimeUnit, u32),
    JustNow,
    InTheFuture(TimeUnit, u32),
}
```

这个枚举中的两个variant，即`InThePast`和`InTheFuture`都有参数。这些被称为*tuple variant*。就像类元组结构体一样，它们的constructor是创建新的`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);
```

枚举也可以有*struct variant*，它们和普通的结构体一样拥有命名字段：

```
enum Shape {
    Sphere { center: Point3d, radius: f32 },
    Cuboid { corner1: Point3d, corner2: Point3d },
}

let unit_sphere = Shape::Sphere {
    center: ORIGIN,
    radius: 1.0,
};
```

总的来说，Rust有三种枚举variant，分别对应我们在上一章中展示的三种结构体。没有数据的variant对应类单元结构体。元组variant对应类元组结构体。结构体variant对应有花括号和命名字段的结构体。一个枚举可以同时有这三种variant：

```
enum RelationshipStatus {
    Single,
    InARelationship,
    ItsComplicated(Option<String>),
    ItsExtremelyComplicated {
```

```

        car: DifferentialEquation,
        cdr: EarlyModernistPoem,
    },
}

```

所有种类的 constructor 都和枚举自身有相同的可见性。

10.1.2 内存中的枚举

在内存中，带有数据的枚举被存储为一个很小的整数标签 (tag)，加上一块足够存储所有 variant 中最大的那个的内存。标签字段是 Rust 内部要使用的，它表示是哪一个 constructor 创建了这个值，进而得知这个值有哪些字段。

在 Rust 1.50 中，RoughTime 存储为 8 个字节，如图 10-1 所示。

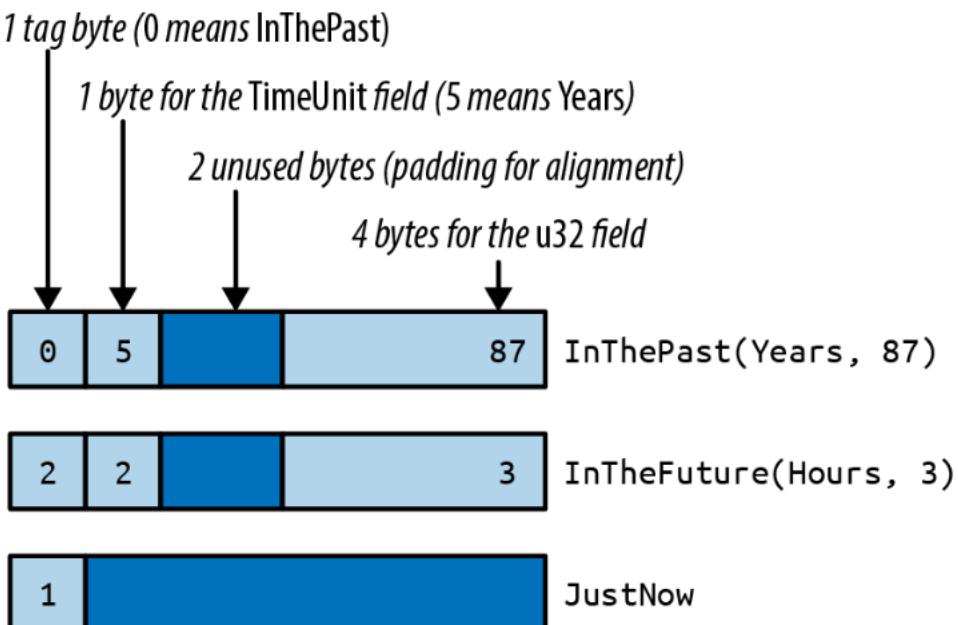


图 10-1: 内存中的 RoughTime 值

对于枚举的布局 Rust 不做任何保证。然而，为了给将来的优化留下余地，在一些情况下它可能会用比图中所示更加高效的方式包装一个枚举。例如，一些泛型结构体可以不用标签存储，我们稍后会讲到它。

10.1.3 使用枚举实现富数据结构

枚举在实现树形结构时也很有用。例如，假设一个 Rust 程序要处理任意的 Json 数据。在内存中，任何 Json 文档都可以被表示为一个这种 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>>),
}
```

与 Rust 代码相比，用英文来解释这个数据结构也不会再有太大的改进了。JSON 标准定义了可以出现在 JSON 文档中的数据类型：null、布尔值、数字、字符串、JSON 值的数组、以及带有字符串键和 JSON 值的对象。这个 `Json` 枚举简单地列出了这些类型。

这并不是一个假想的例子。你可以在 `serde_json` crate 中找到一个非常相似的枚举，它是一个用于 Rust 结构体序列化的库，也是 crates.io 上下载次数最多的 crate 之一。

用于表示 `Object` 的 `HashMap` 外层的 `Box` 只是为了让 `Json` 值更加紧凑。在内存中，`Json` 类型的值将占据 4 个机器字。`String` 和 `Vec` 都是 3 个字，Rust 会再添加一个字节的标签，再加上对齐所以总共是 4 个字。`Null` 和 `Boolean` 值没有足够的数据利用全部的空间，但所有的 `Json` 值大小必须相同，因此这时多余的空间就被浪费了。[图 10-2](#) 展示了一些示例的 `Json` 值在内存中的实际视图。

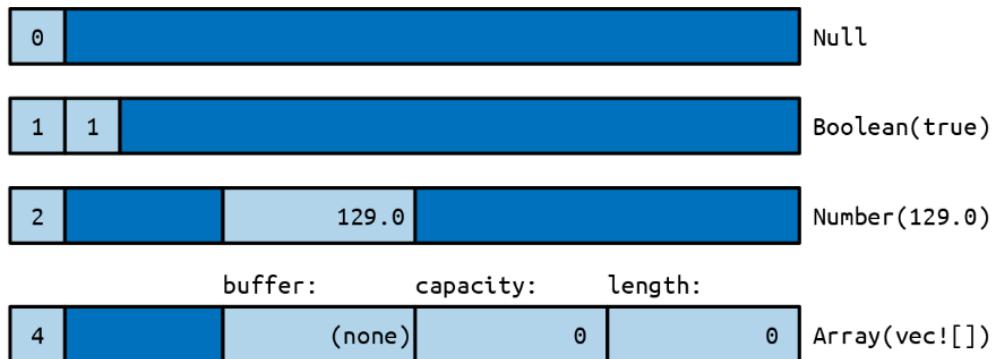


图 10-2: 内存中的 `Json` 值

`HashMap` 会更大一些，如果我们一定要在每一个 `Json` 值中给它留出空间，它会变大到 8 个字。但 `Box<HashMap>` 是单个字：它只是一个指向堆上分配的数据的指针。我们甚至可以通过装箱更多的字段来让 `Json` 变得更加紧凑。

这里优秀的地方在于，我们如此简单的就完成了这一切。如果是在 C++ 中，可能要写一个这样的一个类才行：

```
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(); // 清除string/array/object值  
        tag = Boolean;  
        data.boolean = value;  
    }  
    ...  
};
```

30行代码，我们才刚刚开始。这个类还需要构造函数、析构函数、一个赋值运算符。另一种方案是通过继承，首先创建一个基类 JSON 和它的子类 JSONBoolean、JSONString 等等。无论哪种方式，等到完成之后，我们的 C++ JSON 库都要有一堆代码了。其他程序员需要花费不少精力来阅读和使用它。而 Rust 的整个枚举只需要 8 行代码。

10.1.4 泛型枚举

枚举可以是泛型的。标准库的两个例子几乎是整个语言中使用最广泛的数据类型：

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

enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

到现在这些类型你应该已经很熟悉了，泛型枚举的语法和泛型结构体完全相同。

一个不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 可以省略 Option<T> 的标签字段。因为这些指针类型中的任何一个都不允许为 0，所以 Rust 可以用单个机器字来表示 Option<Box<i32>>：用 0 表示 None，用非 0 表示 Some 指针。这使得这样的 Option 类型与 C 和 C++ 中可以为空的指针值非常相似。不同之处在于 Rust 的类型系统要求你必须先检查 Option 的值是 Some，然后才能使用它内含的值。这有效的避免了空指针解引用。

泛型数据结构体可以用很少的几行代码构建：

```
// 一个`T`类型的有序集合
enum BinaryTree<T> {
    Empty,
    NonEmpty(Box<TreeNode<T>>),
}

// 二叉树的一部分
struct TreeNode<T> {
    element: T,
    left: BinaryTree<T>,
    right: BinaryTree<T>,
}
```

这几行代码定义了一个可以存储任意数量的 T 类型值的 BinaryTree 类型。

这两个定义包含了大量信息，所以我们将花费一些时间把代码翻译为中文。每一个 BinaryTree 值是 Empty 或者 NonEmpty。如果它是 Empty，那么它不包含任何数据。如果是 NonEmpty，那么它会包含一个 Box，这个指针指向一个在堆上分配的 TreeNode 值。

每一个 `TreeNode` 值包含一个实际的元素，和两个 `BinaryTree` 值。这意味着一棵树可以包含子树，因此一个 `NonEmpty` 树可以包含任意数量的后代节点。

一个 `BinaryTree<&str>` 类型的值的视图如图 10-3 所示。因为对于 `Option<Box<T>>`，Rust 会省略标签字段，所以一个 `BinaryTree` 值只占一个机器字。

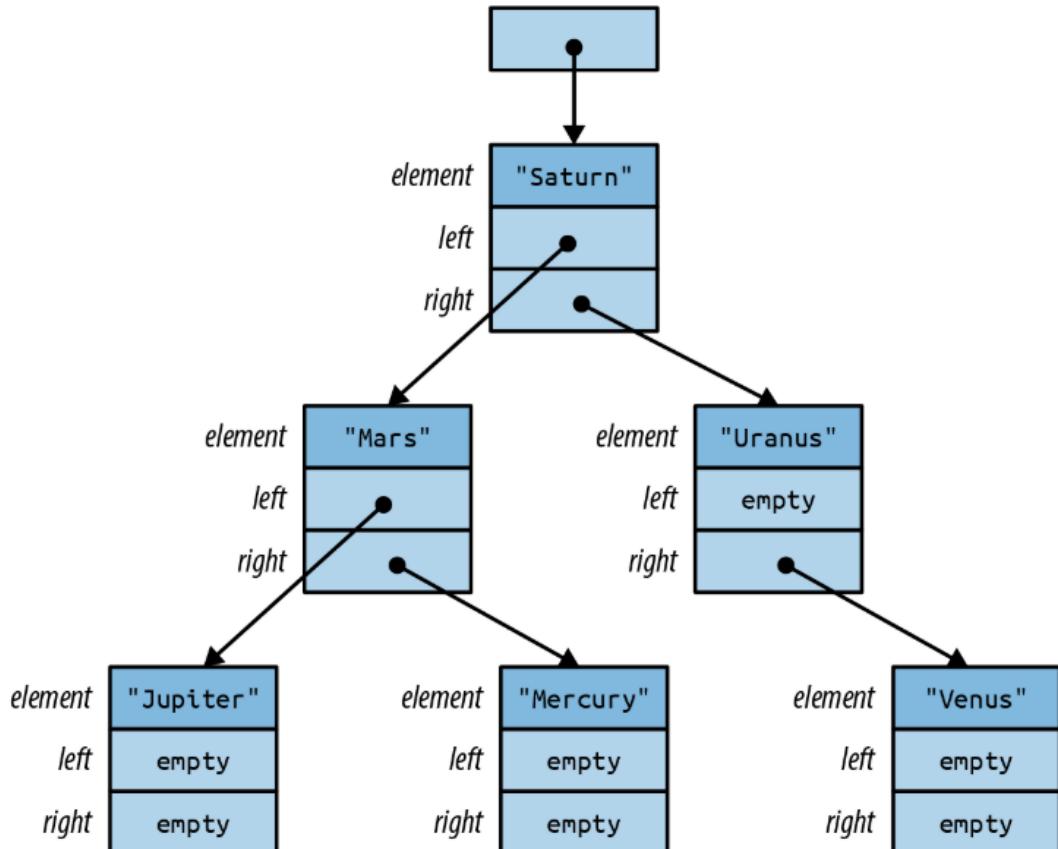


图 10-3: 一个包含 6 个字符串的 `BinaryTree`

构建这棵树中的节点非常直观：

```

use self::BinaryTree::*;

let jupiter_tree = NonEmpty(Box::new(TreeNode {
    element: "Jupiter",
    left: Empty,
    right: Empty,
}));
```

更大的树可以通过较小的树构建：

```

let mars_tree = NonEmpty(Box::new(TreeNode {
    element: "Mars",
```

```
    left: jupiter_tree,
    right: mercury_tree,
});
```

自然地，这个赋值会把 `jupiter_node` 和 `mercury_node` 的所有权移动到新的父节点里。树的其他部分遵循相同的模式。根节点和其它节点不同：

```
let tree = NonEmpty(Box::new(TreeNode {
    element: "Saturn",
    left: mars_tree,
    right: uranus_tree,
}));
```

在这一章的后续部分中，我们将介绍怎么在 `BinaryTree` 类型上实现一个 `add` 方法，这样我们就可以这样写：

```
let mut tree = BinaryTree::Empty;
for planet in planets {
    tree.add(planet);
}
```

无论你之前用什么语言，在 Rust 中创建像 `BinaryTree` 这样的数据结构都需要一些练习。一开始把 `Box` 放在哪可能并不明显。一种寻找设计的方法是画一幅像图 10-3 这样的内存布局图。然后根据图设计代码：每一个矩形都是一个结构体或者元组，每一个箭头都是一个 `Box` 或者其他智能指针。搞清楚每个字段的类型有点困难，但解决难题的回报是控制程序的内存使用。

现在就到了我们在本章开始时提到的“代价”。枚举的标签字段要占用很小的内存，最糟的情况下要占用 8 个字节，但这种情况通常非常少见。枚举真正的缺点（如果它能被称为缺点的话）是 Rust 不能忽略安全性、不管当前的值是什么直接尝试访问字段：

```
let r = shape.radius; // 错误：`Shape` 类型没有字段 `radius`
```

访问枚举中的值的唯一方式是：使用模式，这是一种安全的方式。

10.2 模式

回顾一下我们在本章中定义过的 `RoughTime`：

```
enum RoughTime {
    InThePast(TimeUnit, u32),
    JustNow,
    InTheFuture(TimeUnit, u32),
}
```

假设你有一个 `RoughTime` 值并且你想在网页中显示它。你需要访问值里的 `TimeUnit` 和 `u32` 字段。Rust 不允许你直接通过 `rough_time.0` 和 `rough_time.1` 访问它们，因为毕竟此时值也可能是 `RoughTime::JustNow`，而它没有字段。那么，你怎么获取数据呢？

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` 会进行模式匹配。在这个例子中，模式是第 3、5、7 行中出现在 `=>` 符号左边的部分。匹配 `RoughTime` 值的模式看起来就像是一个创建 `RoughTime` 值的表达式。这并不是巧合。表达式产生值，模式消耗值。它们使用相同的语法。

让我们逐步看看运行这个 `match` 表达式时发生了什么。假设 `rt` 的值是 `RoughTime::InTheFuture(TimeUnit::Months, 1)`。Rust 首先尝试将这个值和第 3 行的模式匹配。正如图 10-4 所示，它并不能匹配。

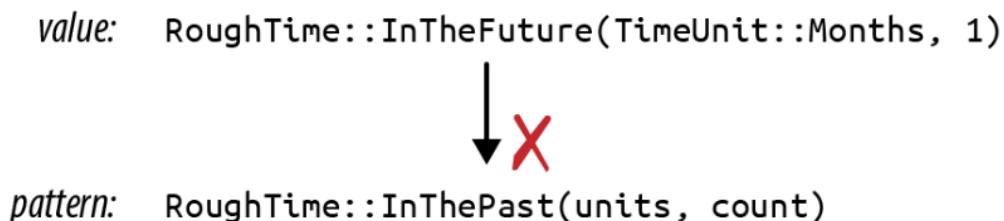


图 10-4: 一个 `RoughTime` 值和不匹配的模式

Rust 中使用模式来匹配一个枚举、结构体或者元组，它的工作原理就好像简单地从左到右扫描，检查模式中的每个部分来看看是不是和值匹配。如果不是，Rust 会移动到下一个模式。

第 3 和第 5 行的模式都匹配失败。但第 7 行的模式成功了（图 10-5）。

如果模式包含像 `units` 和 `count` 这样的简单标识符时，在匹配后的代码中它们会变为局部变量。被匹配的值里的任何内容都会被拷贝或移动到新变量中。Rust 把 `TimeUnit::Months` 存储在 `units` 中，把 1 存储在 `count` 中，然后运行第 8 行的代码，最后返回字符串 "1 months from now"。

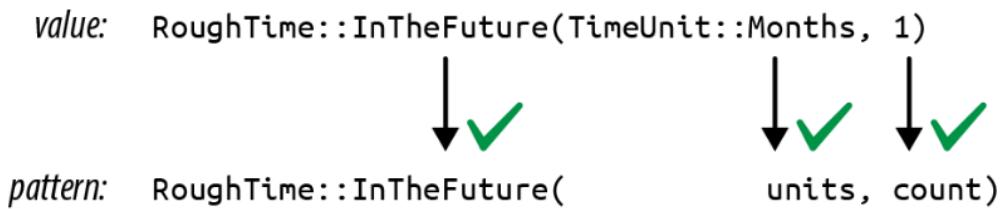


图 10-5: 一个成功的匹配

这个输出有一点语法上的错误，可以通过给 `match` 添加另一个分支来修正：

```
RoughTime::InTheFuture(unit, 1) =>
    format!("a {} from now", unit.singular()),
```

只有当 `count` 字段恰好是 1 时这个分支才能匹配。注意这一段新代码必须添加到第 7 行之前。如果我们把它添加在最后，那么执行流永远不会到达它，因为第 7 行匹配所有的 `InTheFuture` 值。如果你犯了这种错误，Rust 编译器会给出一个“不可达的模式”警告。

即使有了新的代码，`RoughTime::InTheFuture(TimeUnit::Hours, 1)` 仍然有一个问题：结果 "a hour from now" 在英语中并不是完全正确。这可以通过给 `match` 添加另一个分支来修复。

正如这个例子所示，模式匹配和枚举协同工作，甚至可以测试它们包含的值，这使得 `match` 表达式成为 C 的 `switch` 语句的一个更强大、更灵活的替代。

到目前为止，我们只见到了匹配枚举值的模式。其实它还有更多用途。Rust 的模式有它们自己的语言，表 10-1 中进行了总结。我们将用本章中剩下的大部分内容来展示表中的特性。

10.2.1 模式中的字面量、变量和通配符

到目前为止，我们已经展示了 `match` 表达式和枚举一起使用，其实其他类型也可以用模式来匹配。当你需要类似 C 的 `switch` 语句的功能时，可以使用处理整数值的 `match` 表达式。整数字面量例如 0 和 1 可以用作模式：

```
match meadow.count_rabbits() {
    0 => {} // 什么都不输出
    1 => println!("A rabbit is nosing around in the clover."),
    n => println!("There are {} rabbits hopping about in the meadow", n),
}
```

当草地上没有兔子时模式 0 会匹配，当只有一只时 1 会匹配。如果有两只或者更多兔子，就会到达第三个模式 `n`。这个模式只有一个变量名。它可以匹配任何值，被匹配的值会被移动或拷贝进新的局部变量。因此在这个例子中，`meadow.count_rabbits()` 的值被存储在一个新的局部变量 `n` 中，然后我们打印出它。

表 10-1: 模式

模式类型	示例	注释
字面量	100 "name"	匹配一个精确值，也可以使用一个 <code>const</code> 的值的名称
范围	0 ..= 100 'a' ..= 'k'	匹配范围内的任何值，包括终点值
通配符	_	匹配任何值并忽略
变量	name mut count	类似 _ 但是把值移动或拷贝进新的局部变量
ref 变量	ref field ref mut field	借用匹配的值的引用，而不是移动或拷贝它
带子模式的绑定	val @ 0 ..= 99 ref circle @ Shape::Circle { ... } Some(value)	匹配 @ 右侧的模式，使用左侧作为变量名
枚举模式	None Pet::Orca	
元组模式	(key, value) (r, g, b)	
数组模式	[a, b, c, d, e, f, g] [heading, carom, correction]	
切片模式	[first, second] [first, _, third] [first, ..., nth] []	
结构体模式	Color(r, g, b) Point { x, y } Card { suit: Clubs, rank: n } Account { id, name, ... }	
引用	&value &(k, v)	只匹配引用值
多重模式	'a' 'A'	只能用作可反驳的模式 (<code>match</code> , <code>if let</code> , <code>while let</code>)
守卫表达式	x if x * x <= r2	只能在 <code>match</code> 中使用 (在 <code>let</code> 等表达式中无效)

其他的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` 和上个例子中的 `n` 一样用作匹配任何值的模式。这些模式和 `switch` 语句中的 `default` 标签一样，用来匹配其他所有模式都匹配不了的值。

如果你需要一个匹配所有值的模式，但又不关心匹配到的值，你可以使用单个下划线_作为模式，也就是通配模式：

```
let caption = match photo.tagged_pet() {
    Pet::Tyrannosaurus => "RRRAAAAHHHHHH",
    Pet::Samoyed => "*dog thoughts*",
    _ => "I'm cute, love me", // 通用标题，用于任何宠物
};
```

通配模式匹配任何值，但并不存储它。因为 Rust 要求每一个 `match` 表达式要能处理所有可能的值，因此最后通常需要一个通配符。即使你非常确信其他的情况不会发生，你也必须至少添加一个 fallback 分支，这个分支里可以直接 `panic`：

```
// 有很多形状，但我们只支持“选择”文本或者一个矩形区域。
// 你不能选择一个椭圆或者梯形。
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 元组和结构体模式

元组模式匹配元组。当你想在单个 `match` 中获得数据的多个部分时它们会很有用：

```
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",
    }
}

```

结构体模式要使用花括号，就和结构体表达式一样。它们可以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 ({}, {})", x, y),
}

```

在这个例子中，如果第一个分支匹配了，那么 `balloon.location.y` 会被存储到新的局部变量 `height`。

假设 `balloon.location` 是 `Point { x: 30, y: 40 }`。和之前一样，Rust 会按照图 10-6 的顺序检查每一个模式的每一个部分。



图 10-6: 结构体模式匹配

第二个分支可以匹配，因此输出将是 `at (30m, 40m)`。

当匹配结构体时类似 `Point { x: x, y: y }` 的模式非常常见，多余的名字也只会扰乱视觉，因此 Rust 为此支持一种缩写形式 `Point {x, y}`。含义和之前相同，这个模式也会把点的 `x` 字段存储在新的局部变量 `x`、把 `y` 字段存储在新的局部变量 `y`。

即使有了缩写形式，如果我们要匹配一个很大的结构体但又只关心少数组字段时还是会很麻烦：

```

match get_account(id) {
    ...
    Some(Account {
        name, language, // 我们关心的两个字段
        id: _, status: _, address: _, birthday: _, eye_color: _,  

        pet: _, security_question: _, hashed_innermost_secret: _,  

        is_adamantium_preferred_customer: _, }) =>
        language.show_custom_greeting(name),
}

```

为了避免这种情况，可以使用 `..` 告诉 Rust 你不关心其他的字段：

```
Some(Account { name, language, .. }) =>
    language.show_custom_greeting(name),
```

10.2.3 数字和切片模式

数组模式匹配数组。它们被通常被用来过滤出某些特殊值，当数组的不同位置的含义不同时它们也会变得很有用。

例如，当把色相、饱和度、亮度(HSL)颜色值转换为红绿蓝(RGB)颜色值时，亮度为0的颜色就是黑、而亮度为满的颜色就是白。我们可以使用 `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],
        ...
    }
}
```

切片模式与此类似，但和数组不同的是，切片的长度可以变化。因此切片模式并不只匹配值，还要匹配长度。切片模式中的`..` 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 引用模式

Rust 模式支持两种和引用有关的特性。`ref` 模式会借用被匹配的值，`&` 模式匹配引用。我们将首先介绍 `ref` 模式。

匹配一个非拷贝类型的值会移动这个值。继续上面的例子，下面的代码是无效的：

```
match account {
    Account { name, language, .. } => {
        ui.greet(&name, &language);
        ui.show_settings(&account); // error: borrow of moved value: `account`
    }
}
```

这里，字段 `account.name` 和 `account.language` 被移动进局部变量 `name` 和 `language` 中。`account` 的其他部分被丢弃。这就是为什么我们不能再借用它的引用。

如果 `name` 和 `language` 都是可拷贝的值，Rust 将会拷贝字段而不是移动它们，代码将没有问题。但假设它们就是 `String`，那我们该怎么办？

我们需要一种模式借用被匹配的值而不是移动它们。`ref` 关键字就是为此而生：

```
match account {
    Account { ref name, ref language, .. } => {
        ui.greet(name, language);
        ui.show_settings(&account); // ok
    }
}
```

现在局部变量 `name` 和 `language` 都是 `account` 中相应字段的引用。因此 `account` 只是被借用，并没有被消耗，所以继续用它调用方法也是 OK 的。

你可以使用 `ref mut` 来借用 `mut` 引用：

```
match line_result {
    Err(ref err) => log_error(err), // `err` 是 &Error(shared ref)
    Ok(ref mut line) => {          // `line` 是 &mut String(mut ref)
        trim_comments(line);        // 修改 String
        handle(line);
    }
}
```

模式 `Ok(ref mut line)` 匹配任何成功值，并借用存储在里面的成功值的 `mut` 引用。

另一种相反的引用模式是 `&` 模式。一个以 `&` 开始的模式只能匹配引用：

```
match sphere.center() {
    &Point3d { x, y, z } => ...
}
```

在这个例子中，假设 `sphere.center()` 返回一个 `sphere` 的私有字段的引用，这在 Rust 中是很常见的。返回的值是一个 `Point3d` 的引用。如果中心在原点的话，`sphere.center()` 会返回 `&Point3d { x: 0.0, y: 0.0, z: 0.0 }`。

模式匹配按照图 10-7 进行。

这里有一点诡异，因为 Rust 在这里解除了引用，也就是 `*` 运算符的功能。要记住模式和表达式自然是相反的。表达式 `(x, y)` 把两个值放入一个新的元组中，而模式 `(x, y)` 恰好相反：它匹配一个元组然后取出两个值。`&` 也是一样，在表达式里，`&` 创建一个引用；在模式里，`&` 匹配一个引用。

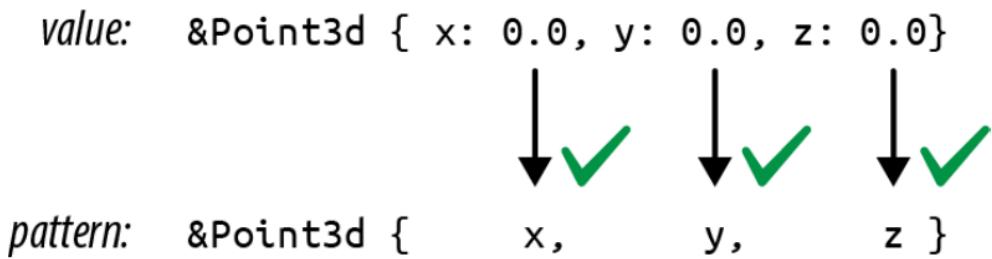


图 10-7: 引用的模式匹配

匹配一个引用遵循我们期望的所有规则：生命周期是强制的、你不能通过共享引用获取 `mut` 访问权限、你不能将值移动出引用，即使是 `mut` 引用。当我们匹配 `&Point3d { x, y, z }` 时，变量 `x`, `y`, `z` 都是坐标的拷贝，原本的 `Point3d` 值还是完整的。只有当这些字段都是拷贝类型才可以正常工作。如果我们想对结构体中一个不可拷贝的字段这么做，我们会遇到错误：

```

match friend.borrow_car() {
    Some(&Car { engine, .. }) => // error: can't move out of borrow
    ...
    None => {}
}

```

把一辆借来的车报废是不好的，Rust 也不会允许这么做。你可以使用 `ref` 模式来借用一个引用，这样就不用拥有它：

```
Some(&Car { ref engine, .. }) => // ok, engine is a reference
```

再来看另一个 `&` 模式的例子。假设我们有一个迭代器 `chars` 迭代一个字符串里的所有字符，并且它有一个方法 `chars.peek()` 返回一个 `Option<&char>`：一个指向下一个字符的引用，如果有的话。（这类迭代器确实返回一个 `Option<&ItemType>`，我们将在第 15 章中见到。）

一个程序可以使用 `&` 来获得指向的字符：

```

match chars.peek() {
    Some(&c) => println!("coming up: {:?}", c),
    None => println!("end of chars"),
}

```

10.2.5 匹配守卫

有时一个匹配分支还有附加的条件必须要满足。假设我们在实现一个六边形空间内的棋子游戏，玩家只需要点击即可移动棋子。为了确认点击是有效的，我们可能要尝试类似这样的代码：

```
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) => // 尝试匹配用户是不是点击了当前位置
            // (这样是错误的：原因如下)
            Err("You are already there! You must click somewhere else."),
        Some(other_hex) =>
            Ok(other_hex)
    }
}
```

这样是错误的，因为模式里的标识符会引入新的变量。模式 `Some(current_hex)` 会创建一个新的叫 `current_hex` 的局部变量，然后遮蔽参数 `current_hex`。Rust 会为这段代码报出好几个警告——尤其是，最后一个 `match` 分支不可达。一种修复这个问题的方法是简单地在分支中使用一个 `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)
        }
    }
}
```

不过 Rust 还提供了匹配守卫 (*match guard*)：模式和分支的 `=>` 词元中间的 `if CONDITION` 条件必须满足才能匹配：

```
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，但条件不满足，那么将会继续匹配下一个分支。

10.2.6 匹配多种可能

竖线 (`|`) 可以用于在一个 `match` 分支中组合多个模式：

```
let at_end = match chars.peek() {
    Some(&'\r') | Some(&'\n') | None => true,
    _ => false,
};
```

在一个表达式中，| 是位或运算符，但这里它的功能就像是普通表达式中的 ||。如果 `chars.peek()` 能匹配三个模式中的任意一个，`at_end` 就会被设为 `true`。

使用 ..= 来匹配范围内的值。范围模式包括起点和终点值，因此 '`0' ..= '9'` 匹配所有的 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（目前）不允许在模式中使用尾开区间例如 `0..100`。

10.2.7 绑定和 @ 模式

最后，`x @ pattern` 用给定的 `pattern` 来匹配，但匹配成功时它会创建单个变量 `x` 并把整个值移动或拷贝进去，而不是为匹配值的每一部分创建一个变量。例如，假设你有下面的代码：

```
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())
    }
}
```

注意第一种情况会解包一个 `Shape::Rect` 值，然后在下一行中重新构建了一个新的 `Shape::Rect` 值。这可以被重写为 @ 模式：

```
rect @ Shape::Rect(..) => {
    optimized_paint(&rect)
}
```

@ 模式和范围一起使用时也很有用：

```
match chars.next() {
    Some(digit @ '0'..='9') => read_number(digit, chars),
```

...
},

10.2.8 模式可以用在哪里

尽管模式最常用在 `match` 表达式中，但它们也能出现在一些其他地方，例如出现在标识符的位置。含义总是相同的：Rust 使用模式匹配来分别取出值中的每一部分，而不是存储在单个变量中。

这意味着模式可以用于……

```
// ... 把一个结构体解包为 3 个局部变量
let Track { album, track_number, title, .. } = song;

// ... 解包一个元组类型的函数参数
fn distance_to((x, y): (f64, f64)) -> f64 { .. }

// ... 迭代一个HashMap 的键和值
for (id, document) in &cache_map {
    println!("Document #{}: {}", id, document.title);
}

// ... 自动解引用闭包的参数,
// 当你想要一份拷贝时, 其他代码可能会传给你一个引用,
// 这时这种写法很方便。
let sum = numbers.fold(0, |a, &num| a + num)
```

这些写法都节省了两到三行重复的样本代码。其它语言中也有相同的概念：JavaScript 中它被称为解构 (*destructuring*)，而在 Python 中它被称为解包 (*unpacking*)。

注意这四个例子中，我们都使用了一定能匹配的模式。模式 `Point3d { x, y, z }` 匹配 `Point3d` 类型的任意值，¹ `(x, y)` 匹配任意的 `(f64, f64)` 类型的值，等等。在 Rust 中总是能匹配的模式是很特殊的，它们被称为不可反驳的模式 (*irrefutable pattern*)，它们也是唯一能出现在这里展示的四种位置的模式（在 `let` 之后、在函数参数中、在 `for` 之后、在闭包的参数中）。

一个可反驳的模式 (*refutable pattern*) 是那些可能不能匹配的模式，例如 `Ok(x)`，它不能匹配一个错误的 `Result`，或者 `'0' ..= '9'`，它不能匹配字符 '`Q`'。可反驳的模式可以在 `match` 分支中，因为 `match` 就是为它们设计的：在 `match` 中如果匹配失败，那么接下来要怎么做很明显。而上面的四个例子中的位置，Rust 不允许匹配失败。

可反驳的模式还可以用在 `if let` 和 `while let` 表达式中，它们可以用于……

¹译者注：此处应是作者写错了，应该是 `Track` 对应的例子。

```
// ... 只有当是特定类型的枚举 variant 时才进行处理
if let RoughTime::InTheFuture(_, _) = user.data_of_birth() {
    user.set_time_traveler(true);
}

// ... 只有当查找成功时才运行代码
if let Some(document) = cache_map.get(&id) {
    return send_cached_response(document);
}

// ... 重复尝试执行操作直到它成功
while let Err(err) = present_cheesy_anti_robot_task() {
    log_robot_attempt(err);
    // 让用户重试
}

// ... 手动迭代器循环
while let Some(_) = lines.peek() {
    read_paragraph(&mut lines);
}
```

有关这些表达式的细节，见 [if let](#) 和 [循环](#)。

10.2.9 填充二叉树

之前我们说过要展示如何实现一个 `BinaryTree::add()` 方法，它把一个节点添加到 `BinaryTree` 中：

```
// 一个类型`T`的顺序集合
enum BinaryTree<T> {
    Empty,
    NonEmpty(Box<TreeNode<T>>),
}

// 二叉树的一个节点
struct TreeNode<T> {
    element: T,
    left: BinaryTree<T>,
    right: BinaryTree<T>,
}
```

现在你已经知道了编写这个方法所需的关于模式的知识。对二叉搜索树的解释已经超过了

这本书的范围，但对于那些已经对这个话题很熟悉的读者，值得看一下在 Rust 中如何实现它。

```

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                      element: value,
7                      left: BinaryTree::Empty,
8                      right: BinaryTree::Empty,
9                  }));
10             }
11             BinaryTree::NonEmpty(ref mut node) => {
12                 if value <= node.element {
13                     node.left.add(value);
14                 } else {
15                     node.right.add(value);
16                 }
17             }
18         }
19     }
20 }
```

第 1 行告诉了 Rust 我们正在为有序类型的 `BinaryTree` 定义一个方法。这和我们之前为泛型结构体定义方法的语法完全相同，语法的解释见[使用 `impl` 定义方法](#)一节。

如果现有的树 `*self` 是空的，那么就很简单。第 5-9 行把 `Empty` 树变成了一个 `NonEmpty` 的树。`Box::new()` 的调用在堆上分配一个新的 `TreeNode`。当运行完之后，树就有了一个元素，这个元素的左子树和右子树都是 `Empty`。

如果 `*self` 不是空，第 11 行的模式就会匹配：

```
BinaryTree::NonEmpty(ref mut node) => {
```

这个模式会借用一个 `Box<TreeNode<T>>` 的可变引用，因此我们可以访问并修改这个树节点中的数据。引用叫做 `node`，它的作用域是 12 到 16 行。因为这个节点中已经有一个元素了，所以需要递归调用 `.add()` 来给左子树或右子树的节点添加新元素。

新的方法可以像这样使用：

```

let mut tree = BinaryTree::Empty;
tree.add("Mercury");
tree.add("Venus");
...
```

10.3 宏观视图

Rust 的枚举对系统编程来说也许很新，但它们并不是一个新的概念。它有过很多学术性的名称，例如代数数据类型 (*algebraic data types*)，它们在函数式编程语言中已经被使用了 40 多年。不知道为什么遵循 C 传统的语言中只有很少的语言有这种特性。可能只是因为对于一个编程语言的设计者来说，将 variant、引用、可变性和内存安全融合在一起非常有挑战性。函数式语言不需要考虑可变性。C 的 `union` 相反，有 variant、指针和可变性——但它们是如此的不安全以至于即使在 C 中，它们也是最少用到的特性。Rust 的借用检查器是使得在不妥协的情况下把这四者融合在一起变为可能的魔法。

程序就是数据处理的过程。一个小巧、快速、优雅的程序与一个充满缓慢、巨大的类型嵌套和虚拟方法调用的程序之间的区别可能就是是否把数据以正确的方式存储。

这正是枚举专注于解决的问题。它们是一种把数据以正确方式存储的设计工具。对于值可能是这样、也可能是那样、设置可能为空的情况，枚举在每一个维度上都比类层次结构要好：更快、更安全、更少的代码、更容易编写文档。

唯一的限制是灵活性。枚举的最终用户不能添加新的 variant 来扩展它。Variant 只能通过修改枚举的定义来添加，而这么做的话，又会打破现有的代码。每一个单独匹配枚举的每个 variant 的 `match` 表达式都需要修改——需要添加一个新的分支来处理新的 variant。在某些情况下，牺牲灵活性换取简洁性是明智之举。毕竟，像 JSON 这样的结构体预计不会改变。而在某些情况下，当枚举改变时修改所有使用的地方正是我们所期望的。例如，当一个编译器使用 `enum` 来表示语言中的不同运算符时，添加一个新的运算符应该涉及到所有处理操作符的代码。

但有些时候也需要更多的灵活性。对于这些情况，Rust 有 trait，也就是我们下一章要讨论的话题。

Chapter 11

trait 与泛型

[A] computer scientist tends to be able to deal with nonuniform structures—case 1, case 2, case 3—while a mathematician will tend to want one unifying axiom that governs an entire system.

——Donald Knuth

编程界中最伟大的发现之一就是可以编写处理多种不同类型的代码，即使是还没有定义出来的类型也可以。这里有两个例子：

- `Vec<T>` 是泛型的：你可以创建一个任意类型的 vector，包括你自己定义的类型，即使 `Vec` 的作者完全不知道这个类型。
- 很多类型都有 `.write()` 方法，包括 `File` 和 `TcpStream`。你的代码可以通过引用获取一个 `writer`(任意的 writer)，并向它写入数据。你的代码不需要关心那个 `writer` 到底是什么类型。然后，如果有人添加了一个新的 `writer` 类型，你的代码将会自动支持它。

当然，这并不是什么新鲜的功能。它被称为多态 (*polymorphism*)，是 20 世纪 70 年代很热门的新的编程语言技术。但现在它已经非常普遍了。Rust 使用两个相关的特性来支持多态：trait 和 泛型。很多程序员可能已经很熟悉这两个概念了，但 Rust 采用了一种受 Haskell 的 typeclass 启发的新方法。

`trait` 是 Rust 中的接口或抽象基类。首先，它们看起来很像 Java 或 C# 中的接口。用于写入字节的 trait 叫做 `std::io::Write`，它在标准库中的定义看起来像这样：

```
trait Write {  
    fn write(&mut self, buf: &[u8]) -> Result<usize>;  
    fn flush(&mut self) -> Result<()>;  
  
    fn write_all(&mut self, buf: &[u8]) -> Result<()> { ... }  
}
```

```
    ...
}
```

这个 trait 提供了几个方法，我们只展示了前三个。

标准类型 `File` 和 `TcpStream` 都实现了 `std::io::Write`。`Vec<u8>` 也是。这三个类型都提供 `.write()`、`.flush()` 等方法。使用 `writer` 的代码不需要关心它的类型，像这样：

```
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“任何实现了 `Write` trait 的值的可变引用”。我们可以把任何这样的值的可变引用传递给 `say_hello`：

```
use std::fs::File;
let mut local_file = File::create("hello.txt")?;
say_hello(&mut local_file)?; // 可以工作

let mut bytes = vec![];
say_hello(&mut bytes)?; // 也可以工作
assert_eq!(bytes, b"hello world\n");
```

这一章首先展示 trait 怎么使用、怎么工作、怎么定义自己的 trait。但 trait 的用途比我们目前提到的更多。我们将使用它们给现有类型添加扩展的方法，甚至像 `str` 和 `bool` 这种内建类型也可以。我们将会解释为什么给一个类型添加 trait 不会消耗多余的内存，以及如何在没有虚方法开销的情况下使用 trait。我们将看到一些 Rust 提供的用于操作符重载和其他特性的语言内建的 trait。我们还将介绍 `Self` 类型、关联函数、关联类型。Rust 从 Haskell 中提取了这三个特性，它们可以优雅地解决其他语言中需要通过变通的方法或者 hack 才能解决的问题。

泛型是 Rust 中另一种形式的多态。类似于 C++ 的模板，一个泛型函数或类型可以用于多种不同的类型：

```
/// 给定两个值，找出较小的那个
fn min<T: Ord>(value1: T, value2: T) -> T {
    if value1 <= value2 {
        value1
    } else {
        value2
    }
}
```

这个函数中的`<T: Ord>`意味着`min`可以用于任何实现了`Ord` trait 的类型`T`——也就是，任何有序的类型。这样的一个要求被称为约束 (*bound*)，因为它列举出了类型`T`需要满足的限制。编译器会为你实际使用的每一个类型`T`生成自定义的机器代码。

泛型和 trait 紧密相关：泛型函数在约束中使用 trait 来表明它可以用于哪些类型的参数。所以我们还会讨论`&mut dyn Write`和`<T: Write>`有哪些相似和不同之处，以及如何在这种两种使用 trait 的方式中选择。

11.1 使用 trait

一个 trait 就是一个给定的类型可能支持也可能不支持的特性。通常，一个 trait 代表一种能力：一个类型可以做的事情。

- 一个实现了`std::io::Write`的值可以写入字节。
- 一个实现了`std::iter::Iterator`的值可以产生值的序列。
- 一个实现了`std::clone::Clone`的值可以产生自身在内存中的克隆。
- 一个实现了`std::fmt::Debug`可以使用`println!()`的`{:?}`格式说明符进行打印。

这 4 个 trait 都是 Rust 标准库的一部分，有很多标准类型都实现了它们。例如：

- `std::fs::File` 实现了`Write` trait，它把字节写入到本地文件。`std::net::TcpStream`写入到网络连接。`Vec<u8>`也实现了`Write`。在字节`vector`上调用`.write()`会往尾部添加数据。
- `Range<i32>(0..10 的类型)`实现了`Iterator` trait，一些和切片、哈希表等相关联的迭代器类型也实现了这个 trait。
- 大多数标准库类型实现了`Clone`。一些例外主要是像`TcpStream`这样的不仅仅表示内存中的数据的类型。
- 大多数标准库类型支持`Debug`。

有关 trait 方法有一个不寻常的规则：trait 自身必须在作用域里。否则，所有它的方法都会被隐藏：

```
let mut buf: Vec<u8> = vec![];
buf.write_all(b"hello")?; // 错误：没有叫`write_all`的方法
```

这种情况下，编译器会打印出友好的错误消息建议你添加`std::io::Write`，然后确实能修复这个问题：

```
use std::io::Write;

let mut buf: Vec<u8> = vec![];
buf.write_all(b"hello")?; // ok
```

Rust 会有这个规则是因为，正如我们稍后会在本章中看到的，你可以使用 trait 来给任意类型添加新的方法——即使是标准库的类型例如 `u32` 和 `str`。第三方的 crate 也可以做同样的事情。显然，这会导致名称冲突！但因为 Rust 让你自己导入你需要使用的 trait，所以 crate 可以轻松地利用这种强大的功能。要想导致冲突，你需要导入两个 trait，这两个 trait 要给同一个类型添加相同名称的方法。这在实践中是很少见的。（如果你确实陷入了冲突中，你可以使用本章稍后会介绍的完全限定方法调用 来指明你想要使用哪一个。）

`Clone` 和 `Iterator` 的方法不需要特殊的导入是因为它们默认总是在作用域里，它们是标准 prelude 的一部分：Rust 会自动导入每个模块中的名称。事实上，prelude 就是一个精心挑选的 trait 的集合。我们将在第 13 章中介绍更多有关它们的内容。

C++ 和 C# 程序员可能已经注意到了 trait 方法很像虚方法。然而，类似上面的函数调用速度很快，与任何其他方法调用一样快。简单来说，这里面并没有多态性。显然 `buf` 是一个 `vector`，不是一个文件或者网络连接，所以编译器可以简单地生成一个 `Vec<u8>::write()` 的调用。它甚至可以内联这个方法。（C++ 和 C# 通常也会这样，尽管子类化可能性有时会排除这一点。）只有通过 `&mut dyn Write` 的调用才会有动态分发的开销，这种调用也被称为虚方法调用，类型里的 `dyn` 关键字暗示了这一点。`dyn Write` 被称为 *trait 对象 (trait object)*；我们将会在接下来的小节中看到 trait 对象的技术细节，以及它们与泛型函数的比较。

11.1.1 trait 对象

在 Rust 中有两种使用 trait 来编写多态代码的方式：trait 对象和泛型。我们将会首先介绍 trait 对象，在下一节中介绍泛型。

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` 并没有固定的大小
```

一个变量的大小必须在编译期时已知，然而实现了 `Write` 的类型可以是任何大小。

如果你来自 C# 或者 Java 的话可能会感觉很惊讶，但原因其实很简单。在 Java 中，一个 `OutputStream`（Java 中类似 `std::io::Write` 的标准接口）类型的变量是一个任何实现了 `OutputStream` 的对象的引用。它是一个引用的事实不言而喻，C# 以及其他大多数语言中的接口也是一样。

我们在 Rust 中想要的也是一样的，但是在 Rust 中引用是显式的：

```
let mut buf: Vec<u8> = vec![];
let writer: &mut dyn Write = &mut buf; // ok
```

一个 trait 类型的引用，例如 `writer`，被称为一个 *trait 对象*。和其他引用一样，一个 trait 对象指向某个值、它有生命周期、它可以是可变的或者是共享的。

让一个 trait 对象与众不同的是 Rust 在编译期通常不知道被引用值的类型是什么。因此一个 trait 对象包括一点额外的有关被引用值的类型信息。类型信息被严格限制为只有 Rust 自己可以在幕后使用：当你调用 `writer.write(data)` 时，Rust 需要这个类型信息来依据 `*writer` 的类型动态调用正确的 `write` 方法。你不能直接查询类型信息，Rust 也不支持将 trait 对象 `&mut dyn Write` 向下转换回精确的类型例如 `Vec<u8>`。

trait 对象的布局

在内存中，一个 trait 对象是一个胖指针，由指向值的指针加上一个指向表示该值类型的表的指针组成。因此每一个 trait 对象要占两个机器字，如图 11-1 所示。

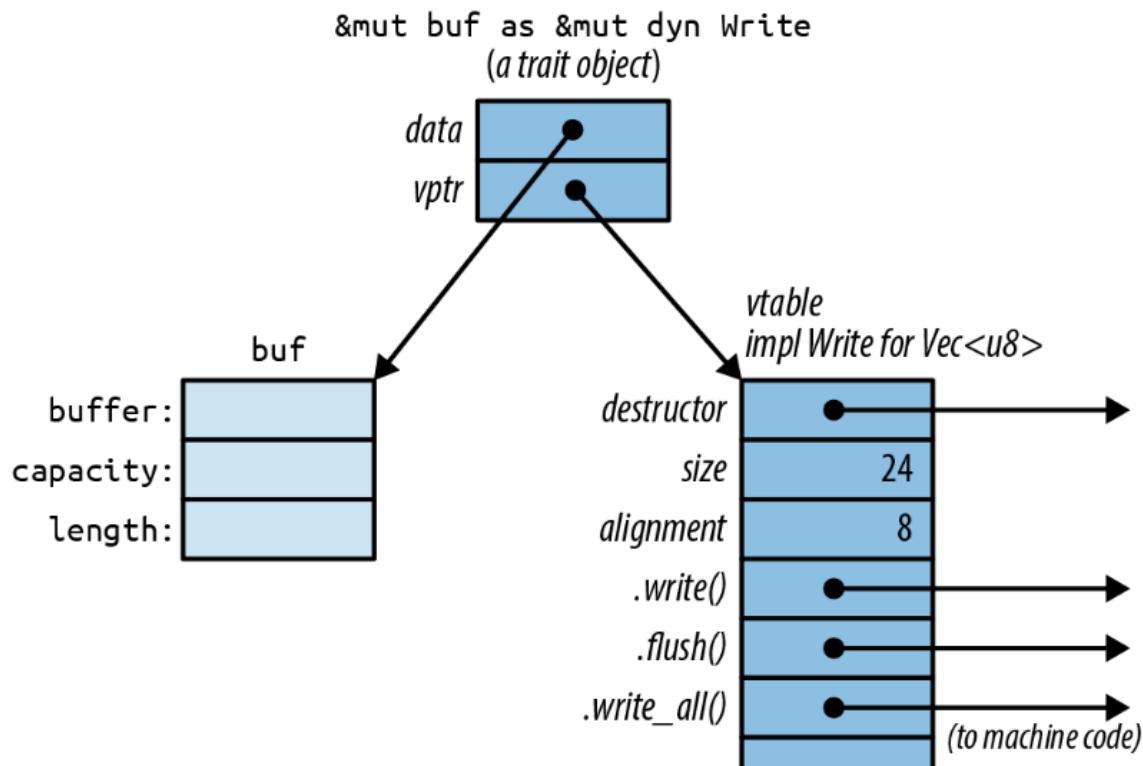


图 11-1: 内存中的 trait 对象

C++ 也有这种运行时的类型信息。它被称为虚表或者 *vtable*。在 Rust 中和在 C++ 中一样，*vtable* 只会在编译期生成一次，然后被所有相同类型的对象共享。图 11-1 中较深颜色的阴影显示的内容，包括 *vtable*，都是 Rust 的私有实现。这些字段和数据结构你不能直接访问。当你调用 trait 对象的方法时语言本身会自动使用 *vtable* 来决定要调用哪个实现。

熟练的 C++ 程序员可能会注意到 Rust 和 C++ 采取的内存策略有些不同。在 C++ 中，虚表指针或者称为 *vptr* 被存储为结构体的一部分，而 Rust 使用胖指针来代替。结构体本身不包含任何自身字段之外的东西。这样，一个结构体可以实现一大堆 trait 而不需要包含一大堆 *vptr*。即使像 *i32* 这样的大小还不足以容纳一个 *vptr* 的类型，也可以实现 trait。

当需要时 Rust 会自动把普通引用转换为 trait 对象。这就是为什么我们能在这个例子中直接把 `&mut local_file` 传递给 `say_hello`:

```
let mut local_file = File::create("hello.txt")?;
say_hello(&mut local_file)?;
```

`&mut local_file` 的类型是 `&mut File`，而 `say_hello` 的参数类型是 `&mut dyn Write`。因为 `File` 是一种 writer，所以 Rust 允许这种普通引用到 trait 对象的转换。

同样的，Rust 也乐于把 `Box<File>` 转换成 `Box<dyn Write>`，它拥有一个在堆上的 writer:

```
let w: Box<dyn Write> = Box::new(local_file);
```

`Box<dyn Write>` 和 `&mut dyn Write` 一样是一个胖指针：它包含 writer 自身的地址和 vtable 的地址。其他指针类型例如 `Rc<dyn Write>` 也一样。

这种转换是唯一创建 trait 对象的方法。编译器做的工作其实很简单，当转换发生时，Rust 知道被引用值的真正类型（这个例子中是 `File`），因此它只是加上了正确的 vtable 的地址、把普通指针变成了胖指针。

11.1.2 泛型函数和类型参数

在这一章的开始处，我们展示了 `say_hello()` 函数，它以 trait 对象为参数。让我们把这个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()
}
```

只有类型签名改变了：

```
fn say_hello(out: &mut dyn Write) // 普通函数
```

```
fn say_hello<W: Write>(out: &mut W) // 泛型函数
```

让函数变为泛型函数的正是 `<W: Write>` 短语，它是一个类型参数。它意味着在整个函数体内，`W` 代表任何实现了 `Write` trait 的类型。按照习惯，类型参数通常是大写字母。

类型 `W` 到底是什么取决于泛型函数如何被调用：

```
say_hello(&mut local_file)?; // 调用 say_hello::<File>
say_hello(&mut bytes)?; // 调用 say_hello::<Vec<u8>>
```

当你把 `&mut local_file` 传递给泛型的 `say_hello()` 函数时，你实际是在调用 `say_hello::<File>()`。Rust 会为这个函数生成机器码，机器码里还会调用 `File::write_all()` 和 `File::flush()`。当你传递 `&mut bytes` 时，你实际是在调用 `say_hello::<Vec<u8>>()`。Rust 会为这个版本的函数生成单独的机器码，然后调用相应的 `Vec<u8>` 的方法。在这两种情况下，Rust 都从参数的类型推导出类型 `W`，这个过程被称为单态化 (*monomorphization*)，编译器会自动进行处理。

你也可以指明类型参数：

```
say_hello::<File>(&mut local_file)?;
```

很少情况下才需要显式写出参数，因为 Rust 通常可以通过参数推断出类型参数。这里，`say_hello` 泛型函数期望一个 `&mut W` 参数，而我们传入了一个 `&mut File`，因此 Rust 推断出 `W = File`。

如果你正在调用的泛型函数并没有足以推断出参数的线索，你需要显式地指明：

```
// 调用一个没有参数的泛型方法 collect<C>()
let v1 = (0 .. 1000).collect(); // 错误：不能推断出类型
let v2 = (0 .. 1000).collect::<Vec<i32>>(); // ok
```

有时我们需要一个类型参数可以支持多种功能。例如，如果我们想打印出一个 vector 中出现次数最多的 10 个值，我们需要这些值可以打印：

```
use std::fmt::Debug;

fn top_ten<T: Debug>(values: &Vec<T>) { ... }
```

但这还不够。我们怎么判断哪个值是出现次数最多的？通常的办法是把每个值当作键存入一个哈希表。这意味着这些值需要支持 `Hash` 和 `Eq` 操作。`T` 的约束还必须包括 `Debug`。这种情况下应该使用的语法是 + 号：

```
use std::hash::Hash;
use std::fmt::Debug;

fn top_ten<T: Debug + Hash + Eq>(values: &Vec<T>) { ... }
```

一些类型实现了 `Debug`、一些实现了 `Hash`、一些支持 `Eq`，还有少数类型例如 `u32` 和 `String`，实现了这三个 trait，如图 11-2 所示。

也可以不给类型参数指定任何约束，但这样的话你几乎不能对它进行任何操作。你只能移动它、将它放在 `box` 或 `vector` 里。

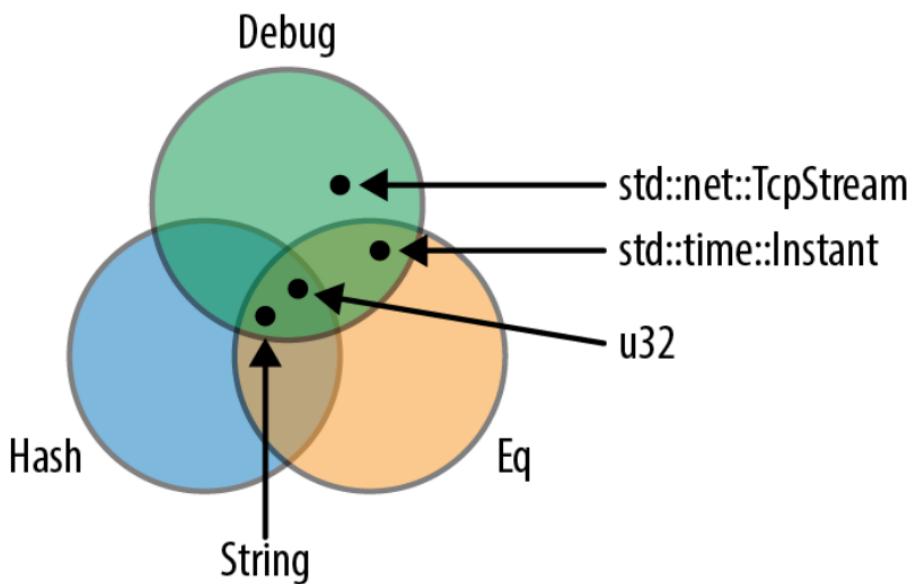


图 11-2: trait 作为类型的集合

泛型函数可以有多个类型参数：

```
/// 在一个大规模的分区数据集上进行查询。
/// 见 <http://research.google.com/archive/mapreduce.html>。
fn run_query<M: Mapper + Serialize, R: Reducer + Serialize>(
    data: &DataSet, map: M, reduce: R) -> Results
{ ... }
```

正如这个例子展示的一样，约束可能太长以至于很难阅读。Rust 提供了使用关键字 `where` 的替代语法：

```
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` 仍然在前边声明，但约束被移动到单独的行。这种 `where` 语法可以用于泛型结构体、泛型枚举、类型别名以及方法——任何允许约束的地方。

当然，`where` 语法的一个替代是保持简单：寻找一种不需要大量使用泛型的方法来编写程序。

引用作为函数参数介绍了生命周期的语法。一个泛型函数可以同时有生命周期参数和类型参数。生命周期参数在前：

```
/// 返回 `candidates` 中距离 `target` 最近的点的引用。
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: `target` 和 `candidates`。它们都是引用, 但我们给了它们不同的生命周期'`t` 和 '`c` (正如在[不同的生命周期参数](#)中讨论的那样)。这个函数可以用于任何实现了 `MeasureDistance` trait 的类型 `P`, 因此我们可以在一个程序中用 `Point2d` 值调用它, 而在另一个程序中用 `Point3d` 值调用它。

生命周期绝不会影响到机器码。两个 `P` 的类型相同但生命周期不同的 `nearest()` 的调用, 将会调用同一个编译好的函数。只有不同的类型才会导致 Rust 编译一个泛型函数的多个拷贝。

当然, 函数并不是 Rust 中唯一的泛型代码:

- 我们已经在[泛型结构体](#)和[泛型枚举](#)中介绍过泛型类型了。
- 一个单独的方法也可以是泛型的, 就算定义它的类型不是泛型的:

```
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>;
```

- 我们将在本章稍后介绍泛型 trait。

所有这一节中介绍的特性——约束、`where` 子句、生命周期参数等——可以被用于所有泛型 item, 而不仅仅是函数。

11.1.3 选择哪一种

选择 trait 对象还是泛型代码是一件很微妙的事情。因为它们都基于 trait, 有很多相似之处。

任何当你需要一个混合类型的值的集合的情况下 trait 对象都是正确的选择。从技术上讲创建泛型的沙拉是可行的:

```
trait Vegetable {
    ...
}
```

```
struct Salad<V: Vegetable> {
    veggies: Vec<V>
}
```

然而，这是一个非常糟糕的设计。每一个这样的沙拉都全部是由单一类型的蔬菜组成的。不是所有人都适合这么做，本书的作者之一曾经为一个 `Salad<IcebergLettuce>` 支付了 \$14 美元，并且直到现在也没有忘记那次经历。

然而我们怎么构建一个更好的沙拉呢？因为 `Vegetable` 值可能是不同大小的，我们不能要求 Rust 创建一个 `Vec<dyn Vegetable>`：

```
struct Salad {
    veggies: Vec<dyn Vegetable> // 错误: `dyn Vegetable` 并
                                    // 没有固定大小
}
```

`trait` 对象就是解决方案：

```
struct Salad {
    veggies: Vec<Box<dyn Vegetable>>
}
```

每一个 `Box<dyn Vegetable>` 可以持有任何类型的蔬菜，但 box 自身的大小是固定的——两个指针——因此可以存储在 `vector` 中。除了在食物里放盒子这个不幸的比喻之外，它确实就是我们需要的。它也同样适用于绘图应用中的形状、游戏中的怪物、网络路由器中的可插拔路由算法等等。

另一个使用 `trait` 对象的可能的原因是减小编译出的代码的体积。Rust 可能需要编译一个泛型函数很多次，因为它要为每一个用到的类型都编译一次。这可能导致生成的二进制文件很大，这种现象在 C++ 圈子里称为代码膨胀 (*code bloat*)。近年来内存越来越充裕，因此我们中的大多数人可以忽略代码的体积，但确实还有一些受限制的环境。

除了涉及到沙拉或者资源受限的环境之外，泛型与 `trait` 对象相比有三个优势。因此在 Rust 中泛型是更加普遍的选择。

第一个优势是速度。注意泛型函数签名中没有 `dyn` 关键字。因为你在编译期指明了确切的类型，不管是显式还是通过类型推导，编译器都知道实际上调用了哪个 `write`。没有使用 `dyn` 关键字是因为没有 `trait` 对象——因此也没有涉及动态分发。

引言中展示的泛型 `min()` 函数就和我们单独编写 `min_u8`、`min_i64`、`min_string` 等函数一样快。编译器还可以像其他函数一样内联它，因此在 `release` 构建中，一个对 `min::<i32>` 的调用可能只有两三条指令。对于常量的调用，例如 `min(5, 3)` 可能会更快：Rust 可以在编译期对它进行求值，因此不会有运行时开销。

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()` 返回一个 `Sink` 类型的 writer，它会偷偷丢弃掉所有写入的字节。

当 Rust 为此生成机器码的时候，它可以产生先调用 `Sink::write_all`、再检查错误、最后调用 `Sink::flush` 的代码。这正是泛型函数体的内容。

或者，Rust 可以查看那些方法，然后意识到下列情况：

- `Sink::write_all()` 什么也不做。
- `Sink::flush()` 什么也不做。
- 两个方法都不可能返回错误。

简单来说，Rust 拥有所有优化掉这个函数调用所需的信息。

相比与 trait 对象的行为，Rust 直到运行时才能知道一个 trait 对象指向的值到底是什么类型。因此即使你传递了一个 `Sink`，虚方法的调用开销和检查错误的开销仍然不可避免。

泛型的第二个优势是有的 trait 不支持 trait 对象。trait 只支持一部分特性，例如关联函数只能使用泛型，这样就完全排除了 trait 对象。当我们讲到这些特性时会指出它们。

泛型的第三个优势是可以很容易地一次给泛型类型参数添加多个 trait 约束，例如我们的 `top_ten` 函数就要求它的参数 `T` 要实现 `Debug + Hash + Eq`。trait 对象不能这么做：Rust 不支持类似 `&mut (dyn Debug + Hash + Eq)` 这样的类型。（你可以用本章中稍后会讲到的子 trait 来实现类似的功能，但这样有点复杂。）

11.2 定义和实现 trait

定义一个 trait 很简单，只需要给出名字和 trait 方法的签名类型。假设我们在编写一个游戏，我们可能会定义像这样的 trait：

```
/// 一个角色、物品、风景等
/// 任何可以显示在屏幕上的游戏世界的物体。
trait Visible {
    /// 在给定的画布上渲染这个对象。
    fn draw(&self, canvas: &mut Canvas);

    /// 如果点击(x, y)会选中这个对象就返回 true。
    fn hit_test(&self, x: i32, y: i32) -> bool;
}
```

为了实现一个 trait，需要使用语法 `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
    }
}
```

注意 `impl` 包含了一份 `Visible` trait 中每个方法的实现，除此之外没有别的内容。在 `trait impl` 中定义的任何东西都必须是 `trait` 的特性。如果我们想要添加一个 `Broom::draw()` 的帮助函数，我们必须在单独的 `impl` 块中定义它：

```
impl Broom {
    /// 下面的 Broom::draw() 用到的帮助函数。
    fn broomstick_range(&self) -> Range<i32> {
        self.y - self.height - 1 .. self.y
    }
}
```

这些帮助函数可以在 `trait 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` writer 可以用少数几行代码实现。首先，我们定义如下类型：

```
/// 一个忽略写入数据的 writer
pub struct Sink;
```

`Sink` 是一个空结构体，因为我们不需要在里面存储任何数据。接下来，我们为 `Sink` 提供了一份 `Write` trait 的实现：

```
use std::io::{Write, Result};

impl Write for Sink {
    fn write(&mut self, buf: &[u8]) -> Result<usize> {
        // 假装成功写入了整个缓冲区
        Ok(buf.len())
    }

    fn flush(&mut self) -> Result<()> {
        Ok(())
    }
}
```

到目前为止，这和 `Visible` trait 很像。但是我们展示过 `Write` trait 还有一个 `write_all` 方法：

```
let mut out = Sink;
out.write_all(b"hello world\n")?;
```

为什么 Rust 允许我们 `impl Write for Sink` 时不定义 `write_all` 方法？答案就是标准库中 `Write` trait 的定义中包含了一个 `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` 和 `flush` 方法是每一个 writer 必须实现的基本方法。一个 writer 可能也实现了 `write_all`，但如果 没有，将会使用我们上边展示的默认实现。

你自己的 trait 也可以使用相同的语法包含默认实现。

默认方法最有戏剧性的使用是在标准库的 `Iterator` trait，它只有一个需要实现的方法 `.next()`，和一堆默认实现的方法。第 15 章中会解释原因。

11.2.2 trait 和其他人的类型

Rust 允许你在任意类型上实现任意 trait，只要 trait 或者类型是在当前 crate 中定义的。这意味着任何时候如果你想给任何类型添加一个方法，你都可以用 trait 来做到这一点：

```
trait IsEmoji {
    fn is_emoji(&self) -> bool;
}

/// 为内建的字符类型实现 IsEmoji 方法
impl IsEmoji for char {
    fn is_emoji(&self) -> bool {
        ...
    }
}

assert_eq!('$'.is_emoji(), false);
```

类似于其他 trait 方法，只有当 `IsEmoji` 在作用域中时新的 `is_emoji` 方法才可见。

这个 trait 的唯一目的就是给现有类型 `char` 添加一个方法。这被称为扩展 trait(*extension trait*)。当然，你可以把这个 trait 添加给其他类型，例如 `impl IsEmoji for str { ... }` 等。

你甚至可以使用泛型 `impl` 块来一次性给一整个家族的类型添加一个扩展 trait。这个 trait 可以在任何类型上实现：

```
use std::io::{self, Write};

/// trait for values to which you can send HTML.
trait WriteHtml {
    fn write_html(&mut self, html: &HtmlDocument) -> io::Result<()>;
}
```

为所有 writer 实现这个 trait，可以为所有 Rust writer 添加这个方法：

```
/// 你可以向任意 std::io writer 写入 HTML
impl<W: Write> WriteHtml for W {
    fn write_html(&mut self, html: &HtmlDocument) -> io::Result<()> {
        ...
    }
}
```

`impl<W: Write> WriteHtml for W`这一行意思是“对于任何实现了`Write`的类型`W`, 这里有一个为`W`编写的`WriteHtml`的实现”。

`serde`库提供了一个很好的例子, 它展示了可以在标准类型上实现用户自定义 trait 这种能力的重要作用。`serde`是一个序列化库。也就是说, 你可以使用它把任何 Rust 数据结构写入到磁盘, 并在稍后加载它们。这个库定义了一个 trait `Serialize`, 库支持所有实现了这个 trait 的数据类型。因此在 `serde`的源码中, 为`bool`, `i8`, `i16`, `i32`, 数组和元组类型等, 包括标准数据结构例如`Vec`和`HashMap`都实现了`Serialize` trait。

这样的结果是 `serde`为所有这些类型添加了一个`.serialize()`方法。它可以像这样使用:

```
use serde::Serialize;
use serde_json;

pub fn save_configuration(config: &HashMap<String, String>)
    -> std::io::Result<()>
{
    // 创建一个JSON序列化器来把数据写入到文件。
    let writer = File::create(config_filename())?;
    let mut serializer = serde_json::Serializer::new(writer);

    // serde的`.serialize()`方法负责剩余的内容。
    config.serialize(&mut serializer)?;

    Ok(())
}
```

我们之前说过当你实现一个 trait 时, trait 和类型至少有一个必须是在当前 crate 中新定义的。这被称为孤儿规则 (*orphan rule*)。它帮助确保 trait 的实现是唯一的。你的代码不能 `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 这么做, 那么不同的 crate 中可能会有不同的`u8`类型的`Write` trait 实现。Rust 将不知道为一个方法调用选择哪种实现。

(C++ 也有一个类似的唯一性约束: 一次定义规则。在传统的 C++ 风格中, 除了最简单的情况之外, 编译器并不会强制这一点, 如果你打破了这个规则会遇到未定义行为。)

11.2.3 trait 中的 Self

trait 中可以将`Self`关键字用作类型。例如标准的`Clone` trait, 看起来像这样 (简化版):

```
pub trait Clone {
    fn clone(&self) -> Self;
```

```
...
}
```

这里使用 `Self` 作为返回类型意味着 `x.clone()` 的返回值类型和 `x` 的类型相同，不管 `x` 是什么。如果 `x` 是一个 `String`，那么 `x.clone()` 的类型就是 `String`——不是 `dyn Clone` 或者别的可克隆的类型。

同样，如果我们定义了这个 trait：

```
pub trait Spliceable {
    fn splice(&self, other: &Self) -> Self;
}
```

还有两个实现：

```
impl Spliceable for CherryTree {
    fn splice(&self, other: &Self) -> Self {
        ...
    }
}

impl Spliceable for Mammoth {
    fn splice(&self, other: &Self) -> Self {
        ...
    }
}
```

在第一个 `impl` 中，`Self` 就是 `CherryTree` 的别名；而在第二个 `impl` 中，它是 `Mammoth` 的别名。这意味着我们可以把两棵樱桃树或者两只猛犸象拼接在一起，而不能创建出樱桃树-猛犸象杂交种。`self` 的类型和 `other` 的类型必须相同。

一个使用了 `Self` 类型的 trait 和 trait 对象不兼容：

```
// 错误：trait `Spliceable` 不能转变为一个对象
fn splice_anything(left: &dyn Spliceable, right: &dyn Spliceable) {
    let combo = left.splice(right);
    // ...
}
```

当我们在深入研究 trait 的高级特性时会多次看到原因。Rust 拒绝这段代码是因为它没有办法对 `left.splice(right)` 调用进行类型检查。关键点在于 trait 对象的类型直到运行时才能知道。Rust 没有办法在编译期知道 `left` 和 `right` 是不是相同的类型。

trait 对象实际上是为最简单的 trait 设计的，就是那种可以用 Java 中的接口或者 C++ 中的抽象基类实现的那种 trait。trait 的还有更多有用的高级特性，但它们不能和现有的 trait 对象

共存。因为使用 trait 对象时，你会丢失 Rust 对程序进行类型检查时必须的类型信息。

现在，假设我们想要一个从基因上讲不可能的拼接，我们可以设计一个 trait 对象友好的 trait：

```
pub trait MegaSpliceable {  
    fn splice(&self, other: &dyn MegaSpliceable) -> Box<dyn MegaSpliceable>;  
}
```

这个 trait 可以和 trait 对象兼容。调用 `.splice()` 方法时的类型检查不会有问题，因此参数 `other` 的类型不需要和 `self` 的类型相同，尽管它们的类型都是 `MegaSpliceable`。

11.2.4 子 trait

我们可以定义一个 trait 作为另一个 trait 的扩展：

```
/// 游戏世界中的某个生物，可能是玩家或者  
/// 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` 意味着所有的生物都是可视的。每一个实现了 `Creature` 的类型都必须实现 `Visible` trait：

```
impl Visible for Broom {  
    ...  
}  
  
impl Creature for Broom {  
    ...  
}
```

我们可以以任何顺序实现这两个 trait，但为一个没有实现 `Visible` 的类型实现 `Creature` 是错误的。这里，我们说 `Creature` 是 `Visible` 的一个子 trait(*subtrait*)，而 `Visible` 是 `Creature` 的父 trait(*supertrait*)。

子 trait 类似 Java 或者 C# 中的子接口，用户可以假定任何实现了子 trait 的值一定也实现了它的父 trait。但在 Rust 中，一个子 trait 不会继承父 trait 中的相关 item，如果你想调用方法的话仍然要确保每个 trait 都在作用域中。

事实上，Rust 的子 trait 只是对 `Self` 的约束的缩写。`Creature` 的定义和下面这个完全等价：

```
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者构造函数，但 trait 可以包含类型关联函数，Rust 中的关联函数类似于静态方法：

```
trait StringSet {
    /// 返回一个空的集合。
    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;

    /// 查找集合是否包含`string`。
    fn contains(&self, string: &str) -> bool;

    /// 向集合中添加一个字符串。
    fn add(&mut self, string: &str);
}
```

每一个实现了 `StringSet` trait 的类型都必须实现这四个关联函数。前两个函数 `new()` 和 `from_slice()`，没有 `self` 参数。它们充当构造函数。在非泛型代码中，这些函数可以使用`::`语法调用，就像其他类型关联函数一样：

```
// 创建两个impl StringSet 的多态类型：
let set1 = SortedStringSet::new();
let set2 = HashedStringSet::new();
```

在泛型代码中也是一样的。除了类型是一个类型变量，因此这里需要调用 `S::new()`：

```
/// 返回`document`中有但`wordlist`中没有的单词的集合。
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
}
```

类似 Java 和 C# 的接口，trait 对象不支持类型关联函数。如果你想使用 `&dyn StringSet` trait 对象，那你必须修改 trait，给那些不接受 `self` 参数的关联函数加上 `where Self: Sized` 约束：

```
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);  
}
```

这个约束告诉 Rust trait 对象不支持这个关联函数。加上之后，你可以创建 `StringSet` 的 trait 对象了，但仍然不能使用 `new` 和 `from_slice`，不过你可以使用它们调用 `.contains()` 和 `.add()`。同样的技巧也适用于其他和 trait 对象不兼容的方法。（从技术上解释为什么会这样是相当乏味的，因此我们不会解释。不过 `Sized` trait 将会在[第 13 章](#)介绍。）

11.3 完全限定方法调用

目前为止我们展示过的所有调用 trait 方法的方式都需要 Rust 自动为我们填充一些缺失的东西。例如，假设你写了如下代码：

```
"hello".to_string();
```

显然这里的 `to_string` 指的是 `ToString` trait 的 `to_string` 方法，而我们调用的是 `str` 类型的实现。因此这场游戏中出现了四个玩家：trait、trait 方法、trait 方法的实现、调用 trait 方法实现的值。我们不需要每次调用方法时都完全写出这四个部分是一件好事，但有些情况下你也可能会需要一种精确的方式来表达你的意思。这种情况下就要用到完全限定方法调用。

首先，要知道方法只是一种特殊的函数。这两种调用是等价的：

```
"hello".to_string()
```

```
str::to_string("hello")
```

第二种形式看起来很像一个关联函数的调用，即使 `to_string` 方法以 `self` 为参数也没有问题，只会简单的传递 `self` 作为函数的第一个参数。

因为 `to_string` 是标准的 `ToString` trait 的方法，所以还有两种调用方式：

```
ToString::to_string("hello")  
  
<str as ToString>::to_string("hello")
```

这四种方法调用功能完全相同。通常你最可能写 `value.method()`。其他的形式是限定 (*qualified*) 方法调用。它们指明了方法关联到的类型或者 trait。最后一种带尖括号的形式同时指明了类型和 trait，这种形式被称为完全限定 (*fully qualified*) 方法调用。

当你写 `"hello".to_string()` 时候，使用 `.` 运算符，你不需要精确地说明你要调用哪个 `to_string` 方法。Rust 有一个依据类型、强制解引用等机制的查找算法来确定是哪个方法。使用完全限定调用，你可以精确地说明你想要调用哪个方法，这可以在一些罕见的情况下有所帮助：

- 当两个方法的名称相同时。经典的例子是 `Outlaw` 有两个来自不同 trait 的 `.draw()` 方法，一个用于在屏幕上绘制它，另一个用于和 law 交互：

```
outlaw.draw();           // error: draw on screen or draw pistol?  
  
Visible::draw(&outlaw);    // ok: draw on screen  
HasPistol::draw(&outlaw);  // ok: corral
```

通常你可能更愿意重命名其中一个方法，但有时你不能这么做。

- 当 `self` 参数的类型不能被推断出来时：

```
let zero = 0;    // 类型未定义：可能是 `i8`、`u8`、...  
  
zero.abs();      // 错误：不能在有歧义的数字类型  
                  // 上调用方法 `abs`  
  
i64::abs(zero); // ok
```

- 当使用函数本身作为函数类型的值的时候：

```
let words: Vec<String> =  
    line.split_whitespace()          // 迭代器会产生 &str 值  
        .map(ToString::to_string)    // ok  
        .collect();
```

- 当在宏中调用 trait 方法时。我们将在第 21 章中解释。

完全限定语法也可以用于关联函数。在之前的小节中，我们用了 `S::new()` 在泛型函数中创建一个新的集合。我们还可以写成 `StringSet::new()` 或者 `<S as StringSet>::new()`。

11.4 定义类型关系的 trait

到目前为止，我们看到过的每个 trait 都是独立的：一个 trait 就是一些可以实现的方法的集合。trait 也可以用于需要多个类型协同工作的场景。它们可以描述类型之间的关系：

- `std::iter::Iterator` trait 将迭代器类型和产生的值的类型联系在了一起。
- `std::ops::Mul` trait 将可以做乘法的类型联系了起来。在表达式 `a * b` 中，值 `a` 和 `b` 可以是相同类型，也可以是不同的类型。
- `rand` crate 包含一个代表随机数生成器的 trait(`rand::Rng`)，和一个代表可以被随机生成的类型的 trait(`rand::Distribution`)。这些 trait 定义了这些类型怎么协同工作。

日常编程中你可能并不需要创建这样的 trait，但你会在标准库和第三方 crate 中看到它们。在这一节中，我们将展示这些例子是怎么实现的、根据需要介绍相关的 Rust 的语言特性。这里最核心的技能就是读懂 trait 和方法签名、并搞清楚它们到底想表达什么意思。

11.4.1 关联类型(或迭代器是如何工作的)

我们将以迭代器开始。到目前为止每一门面向对象的语言都有内建的对迭代器的支持，迭代器是表示遍历一系列值的对象。

Rust 有一个标准的 `Iterator` trait，它的定义如下：

```
pub trait Iterator {  
    type Item;  
  
    fn next(&mut self) -> Option<Self::Item>;  
    ...  
}
```

这个 trait 的第一个特性 `type Item`，是一个关联类型 (*associated type*)。每一个实现了 `Iterator` 的类型都必须指明它产生什么类型的值。

第二个特性 `next()` 方法，在返回值类型中使用了关联类型。`next()` 返回一个 `Option<Self::Item>`：要么是 `Some(item)`，即序列中的下一个值；要么是 `None`，表示已经没有值了。这个类型被写作 `Self::Item`，而不是普通的 `Item`，这是因为 `Item` 是每一个迭代器类型的一个特性，而不是单独的类型。和往常一样，`self` 和 `Self` 类型需要显式地出现在使用它们的字段、方法等的代码中。

这里有个示例为一个类型实现了 `Iterator`：

```
// (这段代码出自 std::env 标准库模块)  
impl Iterator for Args {  
    type Item = String;
```

```

fn next(&mut self) -> Option<String> {
    ...
}

...
}

```

我们在第 2 章中使用过标准库函数 `std::env::args()` 来获取命令行参数，`std::env::Args` 就是它返回的迭代器的类型。它产生 `String` 值，因此 `impl` 块中声明了 `type Item = String;`。泛型代码也可以使用关联类型：

```

/// 循环一个迭代器，把值存储到新的 vector 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，这很棒。但我们必须指明 `collect_into_vector` 的返回类型，而 `Item` 关联类型是唯一的方法。`(Vec<I>)` 显然是错的：它说明函数会返回一个迭代器的 vector！

上面的代码你可能永远不会自己写出来，因为在阅读了第 15 章后，你就会知道迭代器已经有了一个标准方法 `iter.collect()` 来做这件事了。因此在继续之前让我们再看一个例子：

```

/// 打印出一个迭代器产生的所有值
fn dump<I>(iter: I)
    where I: Iterator
{
    for (index, value) in iter.enumerate() {
        println!("{}: {:?}", index, value); // error
    }
}

```

还差一点就完成了。这里只有一个问题：`value` 可能不是一个可打印的类型。

```

error: `<I as Iterator>::Item` doesn't implement `Debug`
|
8 |     println!("{}: {:?}", index, value); // error
|           ^^^^^^

```

```

|           `<I as Iterator>::Item` cannot be formatted
|           using '{:?}' because it doesn't implement 'Debug'
= help: the trait 'Debug' is not implemented for `<I as Iterator>::Item`
= note: required by `std::fmt::Debug::fmt`

help: consider further restricting the associated type
|
5 |     where I: Iterator, <I as Iterator>::Item: Debug
|           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

```

错误信息有一点混淆，因为 Rust 使用了语法 `<I as Iterator>::Item`，这种方式比 `I::Item` 更加显式和详细。这是有效的 Rust 语法，不过你很少会需要用这种方式指明类型。

错误信息的关键是，要想让这段泛型代码能编译，我们必须确保 `I::Item` 实现了 `Debug` trait，这个 trait 用于使用 `{:?}` 格式化值。正如错误信息建议的那样，我们可以通过添加一个 `I::Item` 的约束来解决这个问题：

```

use std::fmt::Debug;

fn dump<I>(iter: I)
    where I: Iterator, I::Item: Debug
{
    ...
}

```

或者，我们可以写“`I` 必须是一个产生 `String` 值的迭代器”：

```

fn dump<I>(iter: I)
    where I: Iterator<Item=String>
{
    ...
}

```

`Iterator<Item=String>` 本身是一个 trait。如果你把 `Iterator` 看作所有可能的迭代器类型的集合，那么 `Iterator<Item=String>` 就是 `Iterator` 的一个子集：产生 `String` 的迭代器类型的集合。这个语法可以用在任何需要一个 trait 名字的位置，包括 trait 对象类型：

```

fn dump(iter: &mut dyn Iterator<Item=String>) {
    for (index, s) in iter.enumerate() {
        println!("{}: {:?}", index, s);
    }
}

```

带有关联类型的 trait，例如 `iterator`，和 trait 对象是兼容的，不过必须像这里展示的一

样指明所有的关联类型才可以。否则，`s` 的类型可能是任何东西，因此 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到迭代器的例子，因为目前迭代器是关联类型最突出的用途。但关联类型在任何 trait 需要涉及方法以外的东西的场景中都很有用：

- 在一个线程池库中，一个 `Task` trait 表示一个工作单元，它可能有一个关联的 `Output` 类型。
- 一个 `Pattern` trait 表示一种搜索字符串的方式，它可能有一个关联的 `Match` 类型，表示字符串中和模式匹配的所有信息：

```
trait Pattern {
    type Match;

    fn search(&self, string: &str) -> Option<Self::Match>;
}

/// 你可以在字符串中搜索一个特定的字符。
impl Pattern for char {
    /// 一个`Match`只是发现字符的位置
    type Match = usize;

    fn search(&self, string: &str) -> Option<usize> {
        ...
    }
}
```

如果你熟悉正则表达式，那么很容易就能看出 `impl Pattern for RegExp` 将会有一个更加精密的 `Match` 类型，可能是一个包含匹配的开始和结尾、匹配的括号组的位置等内容的结构体。

- 一个用于关系型数据库的库可能有一个 `DatabaseConnection` trait，它有一个关联类型表示事务、游标、预处理语句等等。

关联类型完美适用于每一个实现都有一个特定的相关类型的情况：每一个 `Task` 的类型产生一个特定类型的 `Output`；每一个 `Pattern` 的类型查找一个特定的 `Match` 类型。然而，正如我们即将看到的一样，一些类型间的关系并不是这种模式。

11.4.2 泛型 trait (或运算符重载是如何工作的)

Rust 中的乘法使用了这个 trait：

```
/// std::ops::Mul, 用于支持乘法(`*`)的类型
pub trait Mul<RHS> {
```

```
// `*` 运算符产生的结果的类型
type Output;

// `*` 运算符用到的方法
fn mul(self, rhs: RHS) -> Self::Output;
}
```

`Mul` 是一个泛型类型。类型参数 `RHS` 是右手边 (*righthand side*) 的缩写。

这里的类型参数和在结构体或函数中的含义一样：`Mul` 是一个泛型 trait，它实例化出的 `Mul<f64>`、`Mul<String>`、`Mul<Size>` 等都是不同的 trait，正如 `min::<i32>` 和 `min::<String>` 是不同的函数、`Vec<i32>` 和 `Vec<String>` 是不同的类型一样。

单个类型例如 `WindowSize`，可以同时实现 `Mul<f64>` 和 `Mul<i32>`，甚至更多。你可以将一个 `WindowSize` 和很多其它类型相乘。每一个实现都有它自己的关联 `Output` 类型。

泛型 trait 可以不受孤儿规则的约束：你可以为一个外部类型实现一个外部 trait，只要 trait 的类型参数中有一个是在当前 crate 中定义的类型。因此，假设你自己已经定义了 `WindowSize`，你可以为 `f64` 实现 `Mul<WindowSize>`，即使你既没有定义 `Mul` 又没有定义 `f64`。这些实现甚至也可以是泛型的，例如 `impl<T> Mul<WindowSize> for Vec<T>`。之所以可以这样是因为在别的 crate 中没有任何方法可以为任何类型实现 `Mul<WindowSize>`，因此和你的实现之间不可能发生冲突。（我们在 trait 和其他人的类型一节中介绍过孤儿规则。）这正是像 `nalgebra` 这样的 crate 为 vector 定义算术运算的方法。

之前展示的 trait 忽略了一个小细节。真正的 `Mul` trait 看起来像这样：

```
pub trait Mul<RHS=Self> {
    ...
}
```

语法 `RHS=Self` 意思是 `RHS` 的默认值为 `Self`。如果我们写 `impl Mul for Complex`，而不指明 `Mul` 的类型参数，那么意味着 `impl Mul<Complex> for Complex`。如果我们在一个约束中写 `where T: Mul`，那么意味着 `T: Mul<T>`。

在 Rust 中，表达式 `lhs * rhs` 是 `Mul::mul(lhs, rhs)` 的缩写。因此在 Rust 中重载 `*` 运算符和实现 `Mul` trait 一样简单。我们将在下一章中展示示例。

11.4.3 `impl Trait`

你可能想象过，组合使用多种泛型类型可能会变得一团糟。例如，仅仅只使用标准库中的组合器组合几个迭代器会让你的返回类型变得眼花缭乱：

```
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()
}
```

我们可以简单的将返回类型替换为一个 trait 对象：

```
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，仅仅为了避免丑陋的类型签名，就要在每一次调用这个函数时付出动态分发的开销和一次不可避免的堆分配并不是一个好的折衷。

Rust 有一个专为此情形设计的特性叫做 `impl Trait`。`impl Trait` 允许我们“擦除”返回值的类型，只指明它实现的 trait 或 traits，并且没有动态分发或者堆分配：

```
fn cyclical_zip(v: Vec<u8>, u: Vec<u8>) -> impl Iterator<Item=u8> {
    v.into_iter().chain(u.into_iter()).cycle()
}
```

现在，与指明嵌套的迭代器组合器结构体的类型相比，`cyclical_zip` 的签名只简单的说明了它返回一种产生 `u8` 的迭代器。返回类型表达了函数的意图，而不是实现细节。

这确实清理了代码并增强了可读性，但 `impl Trait` 并不只是一个方便的缩写。使用 `impl Trait` 意味着你可以在将来修改实际返回的类型，只要新的类型仍然实现了 `Iterator<Item=u8>`，任何调用了这个函数的代码将仍然能不出错地继续编译。这为库的作者提供了很大的灵活性，因为只有相关的功能被编码进类型签名。

例如，如果一个库的第一版按照上面的方法使用迭代器组合器，然后又发现了一个更好的算法，那么库的作者可以使用不同的迭代器组合器或者甚至返回一个自定义的实现了 `Iterator` 的类型，而库的用户可以在完全不改变代码的情况下享受性能的提升。

使用 `impl Trait` 类似于面向对象语言中广泛使用的工厂模式的静态分发版本，这很有诱惑力。例如，你可以定义一个这样的 trait：

```
trait Shape {
    fn new() -> Self;
    fn area(&self) -> f64;
}
```

在为几个类型实现了它之后，你可能想根据一个运行时的值来决定使用不同的 `Shape`，例如一个用户输入的字符串。使用 `impl Shape` 作为返回类型并不可行：

```
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，像这样的函数并没有什么意义。`impl Trait` 是一种静态分发的版本，因此编译器需要在编译期知道函数内返回的实际类型，这样才能在栈上分配正确数量的空间并调用正确的字段和方法。这里，它可能是 `Circle`、`Triangle` 或者 `Rectangle`，它们的空间大小都不同，而且都有不同的 `area()` 实现。

很重要的一点是要注意 Rust 不允许 trait 方法使用 `impl Trait` 作为返回类型。要想支持这一点需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到特定类型的函数可以使用 `impl Trait` 作为返回值。

`impl Trait` 也可以用来在函数中接受泛型参数。例如，考虑下面的简单泛型代码：

```
fn print<T: Display>(val: T) {
    println!("{}", val);
}
```

它和下面的使用 `impl Trait` 的版本相同：

```
fn print(val: impl Display) {
    println!("{}", val);
}
```

这里有一个很重要的例外。使用泛型允许函数的调用者指定泛型参数的类型，例如 `print::<i32>(42)`，而使用 `impl Trait` 则不行。

每一个 `impl Trait` 参数都会被赋予一个自己的匿名类型参数，因此 `impl Trait` 局限于最简单的泛型函数中，不能表示参数的类型之间的关系。

11.4.4 关联常量

像结构体和枚举一样，trait 也可以有关联常量。你可以用和结构体或枚举一样的语法给 trait 声明关联常量：

```
trait Greet {
    const GREETING: &'static str = "Hello";
    fn greet(&self) -> String;
}
```

trait 中的关联常量也有特殊的作用。像关联类型和函数一样，你可以声明它们但不赋给它们值：

```
trait Float {
    const ZERO: Self;
    const ONE: Self;
}
```

然后，实现这些 trait 的类型可以定义这些值：

```
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
}
```

注意关联常量不能和 trait 对象一起使用，因为编译器依赖实现的类型信息，才能在编译期找出正确的值。

即使是一个没有任何行为的简单 trait，例如 `Float`，也可以给出足够的类型信息，再搭配上少数运算符，就可以实现一些非常普遍的数学函数例如斐波那契数列：

```
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)
    }
}
```

在上面两节中，我们已经展示了用 trait 描述类型间关系的不同方法。所有这些都可以避免虚方法开销和向下转换，因为它们允许 Rust 在编译期就知道精确的类型。

11.5 逆向工程约束

当没有单个 trait 可以满足你的所有需求时，编写泛型代码可能会变得非常困难。假设我们写了这个做一些计算的函数：

```
fn dot(v1: &[i64], v2: &[i64]) -> i64 {
    let mut total = 0;
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

现在我们想用相同的代码来处理浮点数值。我们可能会尝试这样写：

```
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
}
```

这样是行不通的：Rust 会抱怨`*`和`+`的使用以及`0`的类型。我们可以用 `Add` 和 `Mul` trait 来要求 `N` 是一个支持`+`和`*`的类型。对于`0`的使用也要修改，因为在 Rust 中`0`总是整数，而相应的浮点值是`0.0`。幸运的是，那些有默认值的类型有一个标准的 `Default` trait。对于数值类型，默认值总是`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
}
```

这已经接近正确答案了，但还不够：

```
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 `<N as Mul>::Output`
help: consider further restricting this bound
|
5 | fn dot<N: Add + Mul + Default + Mul<Output = N>>(v1: &[N], v2: &[N]) -> N {
|          ^^^^^^^^^^^^^^^^^^

```

我们的新代码假设两个 N 类型的值相乘产生另一个 N 类型的值。这并不是绝对的，你可以重载乘法运算符来返回任何你希望的类型。我们需要一种方式来告诉 Rust 这个泛型函数只能用于有普通乘法的类型，这也就是 $N * N$ 要返回 N 类型的值。错误消息中的建议几乎总是对的：我们可以把 Mul 换成 $\text{Mul}<\text{Output}=\text{N}>$ ，然后 Add 也进行相同的替换：

```

fn dot<N: Add<Output=N> + Mul<Output=N> + Default>(v1: &[N], v2: &[N]) -> N
{
    ...
}

```

这个时候，约束已经开始逐渐累积，让代码变得难以阅读。让我们把约束移动到 where 子句中：

```

fn dot<N>(v1: &[N], v2: &[N]) -> N
    where N: Add<Output=N> + Mul<Output=N> + Default
{
    ...
}

```

很好。但 Rust 仍然会抱怨下面这行代码：

```

error: cannot move out of type `[N]` , a non-copy slice
|
8 |     total = total + v1[i] * v2[i];
|         ^
|         |
|= note: cannot move out of here
|         move occurs because `v1[_]` has type `N` ,
|         which does not implement the `Copy` trait

```

因为我们没有要求 `N` 是一个可拷贝的类型，Rust 把 `v[i]` 解释为尝试把一个值移出切片，这是禁止的。但我们根本不希望修改这个切片；我们只希望拷贝这个值来进行操作。幸运的是，所有 Rust 的内建数值类型都实现了 `Copy`，因此我们可以简单地把它添加到 `N` 的约束中：

```
where N: Add<Output=N> + Mul<Output=N> + Default + Copy
```

这次，代码可以编译运行了。最终的代码看起来像这样：

```
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);
}
```

在 Rust 中偶尔会发生的一种情况是：有一段时间会与编译器激烈斗争，但最后写出来的代码看起来相当不错，好像编写起来轻而易举，并且运行得很漂亮。

我们在这里做的就是对 `N` 的约束进行逆向工程，让编译器来指导并检查我们的工作。这段代码写起来很麻烦是因为标准库中没有单独的 `Number` trait 包含我们需要的所有运算符和方法。有一个流行的开源 crate 叫做 `num` 定义了这样一个 trait！我们已经知道，我们可以在 `Cargo.toml` 中添加 `num` 并编写：

```
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
}
```

正如在面向对象语言中正确的接口让一切变得美好一样，在泛型编程中，正确的 trait 让一切变得美好。

为什么我们会遇到这种问题？为什么 Rust 的设计者不让泛型变得类似于 C++ 的模板一样，把约束隐藏在代码中，à la “duck typing”？

Rust 的方案的一个优势是泛型代码的向前兼容性。你可以修改一个公有的泛型函数或方法的实现，只要你不修改签名，就不会影响到使用它的用户。

约束的另一个优势是当你遇到编译器的错误时，至少编译器可以告诉你错误在哪。C++ 编译器涉及到模板的错误消息比 Rust 的要长很多，并且会指出很多不同行的代码，因为编译器没有办法辨别到底是谁的错误导致了这个问题：是模板、或者是它的调用者？

可能显式写出约束最重要的优势是它们就在代码和文档中。你可以在 Rust 中查看泛型函数的签名，然后看出它到底接受什么类型的参数。而模板则做不到这一点。在像 Boost 这样的 C++ 库中为参数类型编写完整的文档的工作甚至比我们在这里经历的工作更加艰巨。Boost 的开发者们并没有一个可以检查他们的工作的编译器。

11.6 trait 作为基础

trait 是 Rust 最主要的特性之一，并且有充足的理由支持这一观点。设计一个程序或者库时没有什么比设计一个好的接口更重要了。

本章是语法、规则和解释的风暴。现在我们已经铺设好基础了，可以开始讨论 Rust 中更多 trait 和泛型的用法。事实上，我们才刚刚触及皮毛。接下来的两章将介绍标准库提供的通用 trait。再往后的章节介绍闭包、迭代器、输入/输出、并发。trait 和泛型在这些话题中都扮演了中心的角色。

Chapter 12

运算符重载

在第2章的曼德勃罗集绘制器中，我们使用了num crate的Complex类型来表示一个复平面中的点：

```
#[derive(Clone, Copy, Debug)]
struct Complex<T> {
    /// 复数的实部
    re: T,
    /// 复数的虚部
    im: T,
}
```

我们可以使用Rust的+和*运算符，像操作内建类型一样把Complex值相加和相乘：

```
z = z * z + c;
```

你也可以让你自己的类型支持算数和其他运算符，只需要实现一些内建的trait。这被称为运算符重载(*operator overloading*)，它的效果也类似于C++、C#、Python和Ruby中的运算符重载。

如表12-1所示，这些用于重载运算符的trait根据支持的语言部分被分为几个类别。在本章中，我们将介绍每一种类别。我们的目的不只是帮你把自己的类型漂亮地集成到语言中，还是为了让你更好地了解如何编写使用这些运算符的泛型函数，例如在逆向工程约束中介绍的点积函数。本章还会深入了解语言某些功能本身是如何实现的。

12.1 算术和位运算符

在Rust中，表达式`a + b`实际上是`a.add(b)`的缩写，即对标准库中的`std::ops::Add` trait的`add`方法的调用。Rust的标准数值类型都实现了`std::ops::Add`。为了让表达式`a +`

表 12-1: 运算符重载的 trait 汇总

类别	trait	运算符
一元运算符	<code>std::ops::Neg</code>	<code>-x</code>
	<code>std::ops::Not</code>	<code>!x</code>
算术运算符	<code>std::ops::Add</code>	<code>x + y</code>
	<code>std::ops::Sub</code>	<code>x - y</code>
位运算符	<code>std::ops::Mul</code>	<code>x * y</code>
	<code>std::ops::Div</code>	<code>x / y</code>
复合赋值算术运算符	<code>std::ops::Rem</code>	<code>x % y</code>
	<code>std::ops::BitAnd</code>	<code>x & y</code>
复合赋值位运算符	<code>std::ops::BitOr</code>	<code>x y</code>
	<code>std::ops::BitXor</code>	<code>x ^ y</code>
复合赋值位运算符	<code>std::ops::Shl</code>	<code>x << y</code>
	<code>std::ops::Shr</code>	<code>x >> y</code>
比较	<code>std::ops::AddAssign</code>	<code>x += y</code>
	<code>std::ops::SubAssign</code>	<code>x -= y</code>
索引	<code>std::ops::MulAssign</code>	<code>x *= y</code>
	<code>std::ops::DivAssign</code>	<code>x /= y</code>
比较	<code>std::ops::RemAssign</code>	<code>x %= y</code>
	<code>std::ops::BitAndAssign</code>	<code>x &= y</code>
复合赋值位运算符	<code>std::ops::BitOrAssign</code>	<code>x = y</code>
	<code>std::ops::BitXorAssign</code>	<code>x ^= y</code>
索引	<code>std::ops::ShlAssign</code>	<code>x <= y</code>
	<code>std::ops::ShrAssign</code>	<code>x >= y</code>
比较	<code>std::cmp::PartialEq</code>	<code>x == y, x != y</code>
	<code>std::cmp::PartialOrd</code>	<code>x < y, x <= y, x > y, x >= y</code>
索引	<code>std::ops::Index</code>	<code>x[y], &x[y]</code>
	<code>std::ops::IndexMut</code>	<code>x[y] = z, &mut x[y]</code>

`b`能用于`Complex`类型的值，`num` crate 为`Complex`类型实现了这个 trait。其他运算符也有类似的 trait：`a * b`是`a.mul(b)`的缩写，这个方法来自`std::ops::Mul` trait，`std::ops::Neg`包含取负数运算符，等等。

如果你想尝试写`z.add(c)`，你需要在作用域中引入`Add` trait，这样这个方法才可见。然后，你就可以把所有算术看作函数调用：¹

```
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;
}
```

换句话说，`Add<T>` trait 让你的类型可以加上`T`类型的值。例如，为了让你的类型能加上`i32`和`u32`，你的类型必须实现了`Add<i32>`和`Add<u32>`。trait 的类型参数`Rhs`默认是`Self`，因此如果你想实现两个相同类型的值的加法，可以直接实现`Add` trait。关联类型`Output`表示加法结果的类型。

例如，为了能把`Complex<i32>`值相加，`Complex<i32>`必须实现`Add<Complex<i32>>`。因为我们是把一个类型加到同类型的值上，所以可以简单地写`Add`：

```
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，我们不需要单独为`Complex<i32>`、`Complex<f32>`、`Complex<f64>`等实现`Add`。除了类型不同以外所有的定义看起来完全相同，所以我们可以写一个覆盖所有情况的泛型实现，只要实部和虚部的类型支持加法：

¹Lisp程序员狂喜！表达式`<i32 as Add>::add`是`i32`的`+`运算符，被捕获为函数类型的值。

```
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 `where T: Add<Output = T>`, 我们可以把 `T` 限制为可以与同类型的值相加并且返回同类型的值的类型。这个限制是有原因的, 但我们可以进一步放松限制: `Add` trait 并不要求 + 两侧的操作数类型相同, 也不需要返回相同的类型。因此一个最大限度的泛型实现可以让左边的操作数和右边的操作数类型不同, 并让返回值中的实部和虚部的类型是加法返回的类型:

```
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 更倾向于避免支持混合类型的操作。我们的类型参数 `L` 必须要实现 `Add<R>`, 但并没有多少类型实现了与其他类型相加的 trait, 所以一般 `L` 和 `R` 都是相同的类型。因此到最后, 这个极致泛型化的版本可能并不比之前更简单的泛型定义版本有用。

Rust 中为算术和位运算符设计的内建 trait 被分为三组: 一元运算符、二元运算符和复合赋值运算符。每个组中的所有 trait 和它们的方法的形式都相同, 因此我们会从中挑选一个作为示例。

12.1.1 一元运算符

除了解引用运算符 * 将在 [Deref 与 DerefMut](#) 一节中单独介绍之外，Rust 还有可以自定义的一元运算符，如表 12-2 所示。

表 12-2: 内建的一元运算符的 trait

trait 名称	表达式	等价的表达式
<code>std::ops::Neg</code>	<code>-x</code>	<code>x.neg()</code>
<code>std::ops::Not</code>	<code>!x</code>	<code>x.not()</code>

所有 Rust 的有符号数值类型都实现了 `std::ops::Neg`，用于一元运算符-。整数类型和 `bool` 类型实现了 `std::ops::Not`，用于一元运算符!。这些类型的引用也有相应的实现。

注意！会按位取反整数（即反转所有位）值、反转 `bool` 值。它同时提供 C 和 C++ 中的！和 ~ 的功能。

这些 trait 的定义很简单：

```
trait Neg {
    type Output;
    fn neg(self) -> Self::Output;
}

trait Not {
    type Output;
    fn not(self) -> Self::Output;
}
```

求一个复数值的负数只需要简单的求它的每个部分的负数。这里我们可以为 `Complex` 值写一个泛型的求负数实现：

```
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 的二元算术和位运算以及对应的内建 trait 如表 12-3 所示。

表 12-3: 二元运算符的内建 trait

类别	trait 名称	表达式	等价的表达式
算术运算符	std::ops::Add	x + y	x.add(y)
	std::ops::Sub	x - y	x.sub(y)
	std::ops::Mul	x * y	x.mul(y)
	std::ops::Div	x / y	x.div(y)
	std::ops::Rem	x % y	x.rem(y)
位运算符	std::ops::BitAnd	x & y	x.bitand(y)
	std::ops::BitOr	x y	x.bitor(y)
	std::ops::BitXor	x ^ y	x.bitxor(y)
	std::ops::Shl	x << y	x.shl(y)
	std::ops::Shr	x >> y	x.shr(y)

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` 类型实现了位运算符。还有一个操作数是引用或者两个操作数都是引用的版本的实现。

所有这些 trait 都有相同的形式。例如 `std::ops::BitXor`(用于`^`运算符) 的定义看起来像这样：

```

trait BitXor<Rhs = Self> {
    type Output;
    fn bitxor(self, rhs: Rhs) -> Self::Output;
}

```

在本章的开始处，我们还展示了这个分类中的另一个 trait `std::ops::Add`，以及几个示例实现。

你可以使用`+`运算符来连接一个 `String` 和一个 `&str` 切片或者另一个 `String`。然而，Rust 不允许`+`左侧的操作数是 `&str`，以避免通过在左侧反复连接小块来构建长字符串。（这样做的性能很差，需要最终字符串长度平方数量级的时间。）通常来说，`write!` 宏是更好的通过多个小块构建字符串的方式，我们会在[附加和插入文本](#)一节中展示如何做到这一点。

12.1.3 复合赋值运算符

复合赋值运算符例如 `x += y` 或 `x &= y` 需要两个参数，然后对它们进行一些操作例如加法或位与，最后把结果保存到左侧的操作数。在 Rust 中，复合赋值表达式的值总是 `()`，而不是最后被存储的值。

很多语言都有这些运算符，并通常把它们定义为像 `x = x + y` 或 `x = x & y` 这样的表达式的缩写。然而，Rust 并没有采用这种方案。作为代替，`x += y` 是方法调用 `x.add_assign(y)` 的缩写，而 `add_assign` 是 `std::ops::AddAssign` trait 唯一的方法：

```
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以及实现它们的所有内建 trait。

表 12-4: 复合赋值运算符对应的内建 trait

类别	trait 名称	表达式	等价的表达式
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>
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` 实现了位运算复合赋值运算符。一个为我们的 `Complex` 类型的 `AddAssign` 的泛型实现非常直观：

```
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建 trait 和相应的二元运算符的内建 trait 完全独立。实现 `std::ops::Add` 并不会自动实现 `std::ops::AddAssign`。如果你想让 Rust 允许你的类型使用 `+=` 运算符，那你必须自己实现 `AddAssign`。

12.2 相等性比较

Rust 的相等运算符 `==` 和 `!=`，是 `std::cmp::PartialEq` trait 的 `eq` 和 `ne` 方法的缩写：

```

assert_eq!(x == y, x.eq(&y));
assert_eq!(x != y, x.ne(&y));

```

这是 `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)
    }
}

```

因为 `ne` 方法有默认的定义，所以实现 `PartialEq` 时只需要实现 `eq`。这里有一个 `Complex` 类型的实现：

```

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，就是为任何可以比较相等性的类型 `T` 实现 `Complex<T>` 的比较运算。假设我们已经为 `Complex` 实现了 `std::ops::Mul`，那我们现在可以写：

```

let x = Complex { re: 5, im: 2 };
let y = Complex { re: 2, im: 5 };
assert_eq!(x * y, Complex { re: 0, im: 29 });

```

`PartialEq` 的实现通常都是这里展示的形式：比较左右操作数的每一个字段是否都相同。这写起来非常无聊，而且相等性比较是通常都需要支持的操作，所以如果你要求的话，Rust 可

以为你自动生成一个 `PartialEq` 的实现。简单地像这样给这个类型的定义的 `derive` 属性加上 `PartialEq`:

```
#[derive(Clone, Copy, Debug, PartialEq)]
struct Complex<T> {
    ...
}
```

Rust 自动生成的实现跟我们手写的代码基本完全相同，就是依次比较每一个字段或元素。Rust 也可以为 `enum` 类型自动生成 `PartialEq` 实现。当然，这些类型持有的每个值自身必须实现了 `PartialEq`。

与算术和位 trait 以值传递操作数不同，`PartialEq` 以引用获取操作数。这意味着比较像 `String`、`Vec`、`HashMap` 这样的非 `Copy` 值不会导致它们被移动，否则可能会导致问题：

```
let s = "d\x6fv\x65t\x61i\x6c".to_string();
let t = "\x64o\x76e\x74a\x691".to_string();

assert!(s == t); // s 和 t 只会被借用

// 因此它们仍然保持原来的值
assert_eq!(format!("{} {}", s, t), "dovetail dovetail");
```

让我们看看这个 trait 中的 `Rhs` 类型参数的约束，它的类型我们之前从来没有见过：

```
where
    Rhs: ?Sized,
```

这放松了 Rust 通常要求类型参数必须是固定大小类型的要求，所以我们才可以写 `PartialEq<str>` 或 `PartialEq<[T]>` 这样的 trait。`eq` 和 `ne` 方法以类型 `&Rhs` 获取参数，并用 `&str` 或者 `&[T]` 来进行比较是完全合理的。因为 `str` 实现了 `PartialEq<str>`，所以下面的断言是等价的：

```
assert!("ungula" != "ungulate");
assert!("ungula".ne("ungulate"));
```

这里，`Self` 和 `Rhs` 都是无固定大小的 `str` 类型，这导致 `ne` 的 `self` 和 `rhs` 参数都只能是 `&str` 值。我们将在 `Sized` 中讨论固定大小类型、非固定大小类型以及 `Sized` trait。

为什么这个 trait 被称为 `PartialEq`? 相等性是等价关系 (*equivalence relation*) 的传统数学定义中的一种，等价关系需要满足三个要求。对于任意值 `x` 和 `y`:

- 如果 `x == y` 为真，那么 `y == x` 也必须为真。换句话说，交换等价性比较的两侧并不会影响结果。

- 如果 $x == y$ 和 $y == z$, 那么 $x == z$ 也必须为真。给定一个值链, 如果其中相邻的两个值相等, 那么链中的每个值一定等于其他每一个值。相等性有传递性。
- $x == x$ 必须总是为真。

最后一个要求看起来似乎太明显以至于不值得列出来, 但这正是导致事情变得复杂的地方。Rust 的 f32 和 f64 是 IEEE 标准的浮点数类型。根据这个标准, 像 $0.0/0.0$ 以及其他没有合适的结果的表达式必须产生一个特殊的非数 (*not-a-number*) 值, 通常被称为 NaN 值。标准还要求一个 NaN 值必须和其他任何值都不相等——包括它自己。例如, 这个标准要求以下行为:

```
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);
```

另外, 任何与 NaN 的顺序性比较都必须返回假:

```
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);
```

因此尽管 Rust 的 == 运算符满足前两个等价关系的要求, 但使用 IEEE 浮点数值显然不满足第三个要求。这被称为部分等价关系 (*partial equivalence relation*), 因此 Rust 使用名称 PartialEq 作为内建的 == 运算符。如果你写泛型代码时类型参数已知是 PartialEq, 那你应该假设它们满足前两个要求, 但你不应该假设那些值总是和它们自身相等。

这有点违反直觉, 如果你不保持警惕可能会导致 bug。如果你希望你的泛型代码满足完全的等价关系, 你可以使用 std::cmp::Eq trait 作为约束, 它代表完全的等价关系: 如果一个类型实现了 Eq, 那么对于任何该类型的值 x, $x == x$ 一定为 true。在实践中, 几乎所有实现了 PartialEq 的类型也实现了 Eq, f32 和 f64 是标准库中仅有的实现了 PartialEq 但却没有实现 Eq 的类型。

标准库将 Eq 定义为 PartialEq 的扩展, 没有添加任何新方法:

```
trait Eq: PartialEq<Self> {}
```

如果你的类型是 PartialEq 并且你希望它也是 Eq, 那你必须显式地实现 Eq, 即使你并不需要为此再定义任何新的方法或类型。因此为我们的 Complex 类型实现 Eq 非常迅速:

```
impl<T: Eq> Eq for Complex<T> {}
```

我们也可以直接在 Complex 类型定义中的 derive 属性里加上 Eq:

```
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
struct Complex<T> {
    ...
}
```

在泛型类型上派生实现会依赖类型参数。有了这个 `derive` 属性，`Complex<i32>` 将会实现 `Eq`，因为 `i32` 实现了 `Eq`；但 `Complex<f32>` 将只实现 `PartialEq`，因为 `f32` 并没有实现 `Eq`。

当你自己实现 `std::cmp::PartialEq` 时，Rust 无法检查你的 `eq` 和 `ne` 方法的定义是否真的满足等价关系的部分或全部要求。它们的行为可以是任意的。Rust 简单地认为你实现等价性的方式满足用户的需求。

尽管 `PartialEq` 的定义为 `ne` 提供了默认的实现，如果你愿意的话也可以提供自己的实现。然而，你必须要保证 `ne` 和 `eq` 彼此完全互补。`PartialEq` trait 的用户也会这么假设。

12.3 顺序性比较

Rust 用单个 trait `std::cmp::PartialOrd` 来指定比较运算符 `<`, `>`, `<=`, `>=` 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 { ... }
}
```

注意 `PartialOrd<Rhs>` 扩展了 `PartialEq<Rhs>`：你只能对可以比较相等性的类型比较顺序性。

你唯一需要实现的 `PartialOrd` 的方法就是 `partial_cmp`。当 `partial_cmp` 返回 `Some(o)` 时，`o` 表示 `self` 和 `other` 的关系：

```
enum Ordering {
    Less,           // self < other
    Equal,          // self == other
    Greater,        // self > other
}
```

但如果 `partial_cmp` 返回 `None`, 那么意味着 `self` 和 `other` 无法比较顺序: 每一个都不比另一个大, 也不相等。在所有的 Rust 基本类型中, 只有浮点数的比较可能会返回 `None`: 确切地说, 在比较 `Nan`(not-a-number) 和其他任何值时会返回 `None`。我们已经在 [相等性比较](#) 中给出了一些关于 `Nan` 值的背景。

和其他二元运算符一样, 为了比较两个类型 `Left` 和 `Right`, `Left` 必须实现了 `PartialOrd<Right>`。像 `x < y` 或 `x >= y` 这样的表达式是对 `PartialOrd` 方法调用的缩写, 如表 12-5 所示。

表 12-5: 顺序性比较运算符和 `PartialOrd` 的方法

表达式	等价的方法调用	默认实现
<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) Some(Equal))</code>
<code>x >= y</code>	<code>x.ge(y)</code>	<code>matches!(x.partial_cmp(&y), Some(Greater) Some(Equal))</code>

和之前的例子一样, 这里展示的顺序性方法调用代码假设 `std::cmp::PartialOrd` 和 `std::cmp::Ordering` 在作用域里。

如果你知道某个类型的两个值总是可以互相比较顺序性, 那么你可以实现更加严格的 `std::cmp::Ord` trait:

```
trait Ord: Eq + PartialOrd<Self> {
    fn cmp(&self, other: &Self) -> Ordering;
}
```

`cmp` 方法直接返回 `Ordering`, 而不是像 `partial_cmp` 一样返回 `Option<Ordering>`: `cmp` 总是返回两个参数相等或它们的相对顺序。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` 是仅有的例外。

因为复数没有自然的顺序性, 我们不能使用上一节的 `Complex` 类型来展示 `PartialOrd` 的示例实现。作为替代, 假设你正在使用一个类型, 这个类型表示表示一个半开区间内的数字的集合:

```
#[derive(Debug, PartialEq)]
struct Interval<T> {
    lower: T, // 包括
    upper: T, // 不包括
}
```

你可能会希望让这种类型是部分有序的: 如果一个区间 A 完全落在另一个区间 B 里, 那么 A 小于 B; 如果两个不同的区间交叉, 那么它们是无序的: 每个区间里都有部分元素小于

另一个区间里的部分元素；如果两个区间相同则相等。下面的 `PartialOrd` 的实现实现了这些规则：

```
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，你可以写下面的代码：

```
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 });

// 交叉的区间彼此无序
let left = Interval { lower: 10, upper: 30 };
let right = Interval { lower: 20, upper: 40 };
assert!(!(left < right));
assert!(!(left >= right));
```

尽管你通常见到的是 `PartialOrd`，但 `Ord` 定义的全序关系在有些场景下也是必要的，例如标准库中实现的排序算法。例如，只有 `PartialOrd` 实现的情况下不能对区间排序。如果你确实想排序它们，你需要消除无序的情况。例如，你可以根据上界排序，使用 `sort_by_key` 可以很容易做到这一点：

```
intervals.sort_by_key(|i| i.upper);
```

实现 `Ord` 的类型还可以使用 `Reverse` 包装类型来反转顺序。对于任何实现了 `Ord` 的类型 `T`，`std::cmp::Reverse<T>` 也会自动实现 `Ord`，但是是以相反的顺序。例如，将我们的区间按照下界降序排列会很简单：

```
use std::cmp::Reverse;
intervals.sort_by_key(|i| Reverse(i.lower));
```

12.4 Index 与 IndexMut

你可以通过为你的类型实现 `std::ops::Index` 和 `std::ops::IndexMut` trait 来指明索引表达式例如 `a[i]` 的行为。数组直接支持 `[]` 运算符，但对于任何其他类型，表达式 `a[i]` 通常是 `*a.index(i)` 的缩写，其中 `index` 是 `std::ops::Index` trait 的一个方法。然而，如果表达式被赋值或者可变借用，那么将是 `*a.index_mut(i)` 的缩写，它是 `std::ops::IndexMut` trait 的一个方法。

这是这两个 trait 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;
}
```

注意这些 trait 将索引的类型作为参数。你可以用单个 `usize` 值索引一个切片，来得到单个元素的引用，因为切片实现了 `Index<usize>`。但你可以通过像 `a[i..j]` 这样的表达式来引用一个子切片，因为它们也实现了 `Index<Range<usize>>`。这个表达式是如下表达式的缩写：

```
*a.index(std::ops::Range { start: i, end: j })
```

Rust 的 `HashMap` 和 `BTreeMap` 集合让你可以用任何可哈希或可比较的类型作为索引。下面的代码能正常工作，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("亿", 1_0000_0000);

assert_eq!(m["十"], 10);
assert_eq!(m["千"], 1000);
```

那两个索引表达式等价于：

```
use std::ops::Index;
assert_eq!(*m.index("十"), 10);
assert_eq!(*m.index("千"), 1000);
```

`Index` trait 的关联类型 `Output` 指定了索引表达式产生的值的类型：对于这里的 `HashMap`, `Index` 实现的 `Output` 类型是 `i32`。

`IndexMut` trait 扩展了 `Index`, 增加了一个 `index_mut` 方法，这个方法获取 `self` 的引用，并返回一个指向 `Output` 值的可变引用。当索引表达式出现在一个必须可变的上下文中时，Rust 会自动选择 `index_mut`。例如，假设我们写了下面的代码：

```
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` 方法在 `&mut self` 上进行操作，最后的两行等价于：

```
use std::ops::IndexMut;
(*desserts.index_mut(0)).push_str("(fictional)");
(*desserts.index_mut(1)).push_str("(real)");
```

`IndexMut` 的一个限制是，设计上它必须返回一个值的可变引用，所以你不能用类似 `m["十"] = 10`; 这样的表达式在 `HashMap m` 中插入一个值：哈希表需要首先为“十”创建一个值为默认的条目，然后返回指向它的可变引用。但并不是所有的类型都有开销很低的默认值，还有一些类型 `drop` 时可能开销很大。如果这种赋值先创建一个临时值然后立刻 `drop` 它将是一种浪费。（有计划在后续的语言版本中改善这种情况。）

索引最常见的用途是集合。例如，假设我们正在处理类似于我们在第 2 章中创建的曼德勃罗集绘制器的位图。回想一下我们的程序中包含这样的代码：

```
pixels[row * bounds.0 + column] = ...;
```

如果用 `Image<u8>` 类型来充当二维数组显然会更好，这样我们就可以在不需要写出算术运算的情况下访问像素：

```
image[row][column] = ...;
```

为了做到这一点，我们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片。
    fn new(width: usize, height: usize) -> Image<P> {
```

```

    Image {
        width,
        pixels: vec![P::default(); width * height],
    }
}
}
}

```

Index 和 IndexMut 的实现将是：

```

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]
    }
}

```

当你索引一个 Image 时，你会得到一个像素的切片，再索引切片你会得到单独的像素。

注意当我们写 `image[row][column]` 时，如果 `row` 越界了，我们的 `.index()` 方法会尝试索引越界的 `self.pixels`，触发一个 panic。这是 Index 和 IndexMut 的实现应有的行为：越界访问将被检测到并触发一个 panic，当你索引一个越界的数组、切片或 vector 时也是这样。

12.5 其他运算符

Rust 中并不是所有运算符都可以被重载。在 Rust 1.50 中，错误检查的?运算符只能用于 `Result` 和 `Option` 值，尽管有工作正在将其扩展到用户自定义的类型。类似的，逻辑运算符 `&&` 和 `||` 只能用于布尔值。`..` 和 `..=` 运算符总是创建一个表示范围的结构体、`&` 运算符总是借用引用、`=` 运算符总是移动或者拷贝值。它们都不能被重载。

解引用运算符 `*val` 以及访问字段和方法的点运算符 `val.field` 和 `val.method()` 可以使用 **Deref** 与 **DerefMut** trait 来重载，我们将在下一节介绍它。（我们没有在这里介绍是因为这些 trait 不仅仅只是重载一些运算符。）

Rust 不支持重载函数调用运算符 `f(x)`。作为代替，当你需要一个可调用的值时，你通常只

需要写一个闭包。我们将在第 14 章中解释这是怎么工作的，并介绍特殊的 Fn、FnMut 和 FnOnce trait。

Chapter 13

实用 trait

Science is nothing else than the search to discover unity in the wild variety of nature—or, more exactly, in the variety of our experience. Poetry, painting, the arts are the same search, in Coleridge’s phrase, for unity in variety.

——Jacob Bronowski

这一章将介绍 Rust 中的“实用” trait，它们是标准库中能够显著影响到编写 Rust 代码的方式的 trait，因此你需要熟悉它们才能写出惯用的代码并设计出你的用户会觉得是“Rustic”的 crate 接口。它们可以分为三大类：

语言扩展 trait

正如我们上一章介绍的运算符重载 trait 可以让你对自己的类型使用 Rust 的表达式运算符，还有几个其他的标准库 trait 充当 Rust 的扩展，让你可以把自己的类型更紧密地集成到语言中。这一类包括 Drop、Deref 和 DerefMut，以及转换用的 trait From 和 Into。我们将在本章介绍所有这些 trait。

标记 trait

有几个 trait 通常用于约束泛型参数来表达一些特殊的约束。这一类包括 Sized 和 Copy。

公开的词汇表 trait

这些 trait 并没有神秘的编译器集成，你可以在自己的代码中定义等价的 trait。但它们服务于为常见问题制定常规解决方案的重要目标。这些 trait 在 crate 和模块之间的公共接口中特别有价值：通过减少不必要的变化，它们让接口更容易理解，它们还增加

了不同 crate 的特性可以简单地集成在一起的可能性，并且无需样板或自定义的粘合代码。这一类包括 Default、引用借用 trait AsRef、AsMut、Borrow、BorrowMut，可能失败的转换 trait TryFrom 和 TryInto，以及 ToOwned trait，它是 Clone 的泛化。

表 13-1 是对它们的总结。

表 13-1: 实用 trait 汇总

trait	说明
Drop	析构器。当一个值被 drop 时 Rust 会自动运行的清理代码。
Sized	标记 trait，标记一个类型有一个编译期已知的固定大小，与动态大小的类型（例如切片）相反。
Clone	支持克隆的类型。
Copy	标记 trait，标记一个类型可以通过按位拷贝包含值的内存来克隆新值。
Deref 与 DerefMut	为智能指针类型准备的 trait。
Default	有一个有意义的“默认值”的类型。
AsRef 与 AsMut	用于从一个类型的值借用另一个类型的引用的转换 trait。
Borrow 与 BorrowMut	转换 trait，类似于 Asref/AsMut，但额外保证一致的哈希性、顺序性和相等性。
From 与 Into	用于将一个类型的值转换为另一个类型的值的转换 trait。
TryFrom 与 TryInto	用于将一个类型的值转换为另一个类型的值的转换 trait，用于可能失败的转换。
ToOwned	将一个引用转换为一个有所有权的值的转换 trait。

还有一些其它重要的标准库 trait。我们将在第 15 章中介绍 Iterator 和 IntoIterator。用于计算哈希值的 Hash trait 将在第 16 章中介绍。还有一对标记线程安全类型的 trait Send 和 Sync，将在第 19 章中介绍。

13.1 Drop

当一个值的所有者消失时，我们说 Rust *drop* 了这个值。*drop* 一个值意味着释放这个值拥有的所有其他值、堆上的存储空间和系统资源。*drop* 会在各种情况下发生：当变量离开作用域时、处于表达式语句的末尾时、截断 vector 时从尾部移除元素时，等等。

在大多数情况下，Rust 会自动为你处理 *drop* 过程。例如，假设你定义了下面的类型：

```
struct Appellation {
    name: String,
    nicknames: Vec<String>
}
```

字符串的内容和 vector 的元素缓冲区都在堆上分配了空间，Appellation 拥有这些空间。

当一个 Appellation 被 drop 时，Rust 会清理所有这些内容，你不需要编写任何代码。然而，如果你想的话，你可以通过实现 std::ops::Drop trait 来自定义 Rust 如何 drop 你的类型的值：

```
trait Drop {
    fn drop(&mut self);
}
```

Drop 的实现类似于 C++ 中的析构函数，或者其它语言中的终结函数。当一个值被 drop 时，如果它实现了 std::ops::Drop，Rust 会在清理它的字段或元素之前先调用它的 drop 方法。这种 drop 的隐式调用是唯一一种调用这个方法的方式，如果你尝试显式地调用这个方法，Rust 会标记为错误。

因为 Rust 会在 drop 一个值的字段或方法之前先用这个值调用 Drop::drop，所以这个方法接收到的值总是保持完全初始化的状态。我们的 Appellation 类型的一个 Drop 的实现可以充分利用它的字段：

```
impl Drop for Appellation {
    fn drop(&mut self) {
        print!("Dropping {}", self.name);
        if !self.nicknames.is_empty() {
            print!(" (AKA {})", self.nicknames.join(", "));
        }
        println!("");
    }
}
```

有了这个实现，我们可以写出下列代码：

```
{
    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");
}
```

当我们把第二个 Appellation 赋给 a 的时候，第一个值会被 drop，当我们离开 a 的作用域时，第二个值也会被 drop。这段代码会打印出如下内容：

```

before assignment
Dropping Zeus (AKA cloud collector, king of the gods)
at end of block
Dropping Hera

```

既然我们的 `Appellation` 的 `std::ops::Drop` 实现只打印了一条消息，那么它的内存到底是怎么被精确地清理掉的？`Vec` 类型也实现了 `Drop`，`drop` 它的每个元素，然后释放堆上分配的缓冲区。一个 `String` 在内部使用 `Vec<u8>` 来保存文本，因此 `String` 自身没有实现 `Drop`，它让它的 `Vec` 来清理字符。同样的规则也适用于 `Appellation` 值：当一个值被 `drop` 时，它的 `Vec` 的 `Drop` 实现负责清理每一个字符串的内容，并最终释放存储元素的缓冲区。保存 `Appellation` 值的内存本身也有一个拥有者，可能是一个局部变量或者一些数据结构，它们负责释放它。

如果一个变量的值被移动走，导致当它离开作用域时是未初始化的状态，那么 Rust 会避免 `drop` 这个变量：它里面没有值可以 `drop`。即使按照控制流一个变量的值可能被移动走、也可能没有的情况下，这个原则也会生效。Rust 会使用一个不可见的标记来追踪变量的状态，它指示变量的值是否需要被 `drop`：

```

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 `true` 还是 `false`，`p` 或者 `q` 最后将会拥有这个 `Appellation`，另一个变为未初始化。这个值最终落在哪个变量里决定了它会在 `println!` 之前还是之后被 `drop`。因为 `q` 在 `println!` 之前离开作用域，而 `p` 在之后。尽管一个值可能会被移来移去，但 Rust 只会 `drop` 它一次。

你通常不需要实现 `std::ops::Drop`，除非你想定义一个拥有一些 Rust 不知道的资源的类型。例如，在 Unix 系统上，Rust 的标准库内部使用下面的类型来表示一个操作系统文件描述符：

```

struct FileDesc {
    fd: c_int,
}

```

`FileDesc` 的 `fd` 字段就是当程序使用完它之后应该被关闭的文件描述符的序号。`c_int` 是 `i32` 的一个别名。标准库中按照如下方式为 `FileDesc` 实现了 `Drop`:

```
impl Drop for FileDesc {
    fn drop(&mut self) {
        let _ = unsafe { libc::close(self.fd) };
    }
}
```

这里，`libc::close` 是 C 库中的 `close` 函数的 Rust 名称。Rust 代码只能在 `unsafe` 块中调用 C 函数，因此这里标准库使用了 `unsafe` 块。

如果一个类型实现了 `Drop`，它就不能再实现 `Copy`。如果一个类型是 `Copy` 的，那意味着按位复制就可以创建一个新的独立拷贝。但通常在同样的数据上调用同一个 `drop` 方法不止一次是一个错误。

标注 `prelude` 中包含了一个 `drop` 值的函数 `drop`，但它的定义一点也不神奇：

```
fn drop<T>(_x: T) {}
```

换句话说，它以值接收参数，从调用者那里获取所有权——然后什么也不做。当 `_x` 离开作用域时 Rust 会 `drop` 它的值，正如它对其他任何变量做的一样。

13.2 Sized

固定大小的类型 (*sized type*) 是指那些所有实例值都占用相同大小的内存空间的类型。Rust 中几乎所有的类型都是固定大小的：每一个 `u64` 都是 8 字节，每一个 (`f32`, `f32`, `f32`) 类型的值都占 12 个字节。即使枚举也是固定大小的：不管它当前实际的 variant 是哪一个，一个枚举总是占用能存下最大的 variant 的空间。即使 `Vec<T>` 拥有一个大小可变的堆上缓冲区，`Vec` 值本身的大小就是一个缓冲区的指针，加上容量和长度。因此 `Vec<T>` 是一个固定大小的类型。

所有的固定大小的类型都实现了 `std::marker::Sized` trait，它没有任何方法或关联类型。Rust 为所有适合的类型自动实现它，你不能自己实现它。`Sized` 唯一的用途是作为类型参数的约束：一个类似 `T: Sized` 的约束要求 `T` 是一个大小在编译期已知的类型。这种类型的 trait 被称为标记 trait(*marker trait*)，因为 Rust 语言本身使用它们来标记有特定特点的类型。

然而，Rust 还有少量大小不固定的类型 (*unsized type*)，它们的值的大小并不相同。例如，字符串切片类型 `str`（注意，没有 `&`）就是大小不固定的。字符串 "diminutive" 和 "big" 分别是占据了 10 个和 3 个字节的 `str` 切片的引用。如图 13-1 所示。数组切片类型例如 `[T]`（再次注意，这里也没有 `&`）也是大小不固定的：一个共享引用例如 `&[u8]` 可以指向一个任意大小的 `[u8]` 切片。因为 `str` 和 `[T]` 类型表示不同大小的值的集合，因此它们是大小不固定的类型。

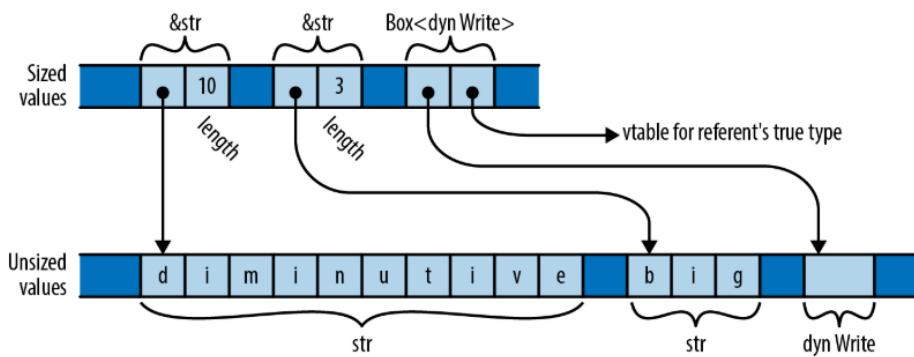


图 13-1: 指向大小不固定的值的引用

Rust 中另一种常见的大小不固定类型是 `dyn` 类型，它是 trait 对象引用的目标。正如我们在 [trait 对象](#) 中解释的一样，一个 trait 对象是一个指向实现了给定 trait 的值的指针。例如，类型 `&dyn std::io::Write` 和 `Box<dyn std::io::Write>` 是指向实现了 `Write` trait 的值的指针。被引用的目标可能是一个文件或者网络套接字，或者是实现了 `Write` 的自定义类型。因为实现了 `Write` 的类型的集合是开放的，所以 `dyn Write` 也是大小不固定的类型：它的值可能有任意的大小。

Rust 不能在变量中存储大小不固定的值或者将它们传递为参数。你只能通过指针例如 `&str` 或 `Box<dyn Write>` 来处理它们，指针本身是固定大小的。正如图 13-1 所示，一个指向大小不固定的值的指针总是一个胖指针 (*fat pointer*)，占用两个字节：一个指向切片的指针加上切片的长度、一个 trait 对象加上一个指向方法实现的 vtable 的指针。

trait 对象和切片的指针是对称的。在这两种情况下，都缺乏相应的类型信息：你不能在不知道 `[u8]` 长度的情况下索引它，也不能在不知道被指向的值的具体 `Write` 实现的情况下调用 `Box<dyn Write>` 的方法。在这两种情况下，胖指针填充了类型缺失的信息，加上了一个长度或者 vtable 的指针。被省略的静态信息被替换为了动态信息。

因为大小不固定的类型限制太多，所以大多数泛型类型参数都被限制为 `Sized` 类型。事实上，它几乎总是必须的，因此在 Rust 中它是默认的：如果你写 `struct S<T> { ... }`，Rust 认为你的意思是 `struct S<T: Sized> { ... }`。如果你不想让 `T` 有这个约束，你必须显式写出来，即 `struct S<T: ?Sized> { ... }`。`?Sized` 语法专门用于这种场景，含义是“不需要是 `Sized`”。例如，如果你写了 `struct S<T: ?Sized> { b: Box<T> }`，那么 Rust 将允许你写 `S<Str>` 和 `S<dyn Write>`，这时 `b` 将是一个胖指针；而 `S<i32>` 和 `S<String>` 中，`b` 是一个普通指针。

抛开它们的限制不谈，大小不固定的类型让 Rust 的类型系统工作得更加顺畅。如果阅读标准库的文档，你偶尔会看到类型参数中的 `?Sized` 约束，这几乎总是意味着给定的类型只能被

指向，同时允许相关的代码既能处理普通类型、又能处理切片和 trait 对象。当一个类型参数有?Sized 约束时，人们通常会说它是可能大小不固定 (*questionably sized*)：它可能是 Sized，也可能不是。

除了切片和 trait 对象之外，还有另一种大小不固定的类型。一个结构体的最后一个字段（也只有最后一个字段）可能是大小不固定的，此时这个结构体本身也是大小不固定的。例如，Rc<T>引用计数指针内部被实现为私有类型 RcBox<T> 的指针，它存储了 T 和引用计数。这里有一个 RcBox 的简化版的定义：

```
struct RcBox<T: ?Sized> {
    ref_count: usize,
    value: T,
}
```

value 字段就是 Rc<T> 引用的 T 值，Rc<T> 解引用之后就是一个这个字段的指针。ref_count 字段保存引用计数。

真正的 RcBox 是标准库的实现细节，不能用于公开使用。但假设我们在处理一个上面的定义。你可以将 RcBox 和固定大小的类型一起使用，例如 RcBox<String>，结果将是一个固定大小的结构体类型。或者你可以将它和大小不固定的类型一起使用，例如 RcBox<dyn std::fmt::Display>（其中 Display 表示可以被 println! 以及类似的宏格式化的类型），RcBox<dyn Display> 是一个大小不固定的结构体类型。

你不能直接创建一个 RcBox<dyn Display> 值。你必须先创建一个普通的、固定大小的 RcBox，它的 value 字段的类型需要实现 Display，例如 RcBox<String>。然后 Rust 允许你把一个它的引用 &RcBox<String> 转换为胖指针引用 &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;
```

当向函数传递参数时会隐式发生这个转换，因此你可以向接受 &RcBox<dyn Display> 参数的函数传递一个 &RcBox<String>：

```
fn display(boxed: &RcBox<dyn Display>) {
    println!("For your enjoyment: {}", &boxed.value);
}

display(&boxed_lunch);
```

这将会产生下列输出：

```
For your enjoyment: lunch
```

13.3 Clone

`std::clone::Clone` trait 用于那些可以拷贝自身的类型。`Clone` 的定义如下：

```
trait Clone: Sized {
    fn clone(&self) -> Self;
    fn clone_from(&mut self, source: &Self) {
        *self = source.clone()
    }
}
```

`clone` 方法应该构建一个 `self` 的独立拷贝并返回它。因为这个方法的返回类型是 `Self`，并且函数不能返回大小不固定的值，因此 `Clone` trait 扩展了 `Sized` trait：它会约束实现的 `Self` 类型是 `Sized`。

拷贝一个值通常意味着拷贝它拥有的所有内容，因此 `clone` 可能在时间和内存上的开销都比较大。例如，克隆一个 `Vec<String>` 不止要拷贝 vector，还要拷贝它的每一个 `String` 元素。这就是为什么 Rust 不会自动拷贝值，而是要求你显式地调用方法来拷贝。引用计数指针例如 `Rc<T>` 和 `Arc<T>` 是例外：拷贝它们只会简单地递增引用计数并返回给你一个新的指针。

`clone_from` 方法将 `self` 修改为 `source` 的拷贝。`clone_from` 的默认实现简单地拷贝 `source`，然后将它移动进 `*self`。这总是能正确工作，但对于某些类型，还有更快的方法达成相同的效果。例如，假设 `s` 和 `t` 都是 `String`。语句 `s = t.clone();` 必须先克隆 `t`，`drop` 掉 `s` 的旧值，然后将克隆的值移动进 `s`，因此这里面有一次堆分配和堆释放。但如果 `s` 原本的堆缓冲区有足够的容量存下 `t` 的内容，那么没有必要进行释放和分配操作：可以简单地将 `t` 的文本拷贝进 `s` 的缓冲区，然后调整它的长度。在泛型代码中，你应该使用 `clone_from`，这样可以充分利用优化过的实现的优势。

如果你的类型的 `Clone` 实现只是简单地调用每一个字段或者元素的 `clone` 方法，然后利用这些克隆的值构造一个新的值，那么 `clone_from` 的默认定义就已经够了，Rust 将会为你实现它：只要在类型的定义上方加上 `#[derive(Clone)]`。

标准库中几乎所有应该能拷贝的类型都实现了 `Clone`。基本类型例如 `bool` 和 `i32` 实现了。容器类型例如 `String`, `Vec<T>`, `HashMap` 也实现了。一些不应该能拷贝的类型例如 `std::sync::Mutex` 没有实现 `Clone`。一些类型例如 `std::fs::File` 可以拷贝，但如果操作系统没有足够的资源那么拷贝可能会失败，这些类型也没有实现 `Clone`，因为 `clone` 不允许失败。

作为替代，`std::fs::File`提供了一个`try_clone`方法，它返回一个`std::io::Result<File>`来报告失败。

13.4 Copy

在第4章中，我们解释过，对于大多数类型，赋值操作会移动它的值，而不是拷贝它们。移动值让我们可以更容易地追踪它们拥有的资源。但在Copy类型：move的例外中，我们指出了例外情况：不持有任何资源的简单类型可以是Copy类型，这种类型的赋值操作会拷贝源值，而不是移动值并把源值设为未初始化。

那个时候，我们并没有确切地说明Copy到底是什么，但现在我们可以告诉你：如果一个类型实现了`std::marker::Copy`标记trait，那么它就是Copy的，Copy的定义如下：

```
trait Copy: Clone { }
```

很容易就可以为自己的类型实现它：

```
impl Copy for MyType { }
```

但因为Copy是一个有特殊含义的标记trait，所以Rust只允许可以通过逐字节的浅拷贝来拷贝自身的类型实现Copy。如果一个类型拥有任何其他资源，例如堆缓冲区或者操作系统句柄，那么它将不能实现Copy。

任何实现了Drop trait的类型不能实现Copy。Rust假定如果一个类型需要特殊的清理代码，那么它肯定也需要特殊的拷贝代码，所以不能是Copy。

和Clone一样，你可以使用`#[derive(Copy)]`来让Rust为你实现Copy。你通常能一次性看到它们两个，即`#[derive(Copy, Clone)]`。

在将类型变为Copy之前请仔细思考。尽管这样做会让类型更容易使用，但却对类型本身的实现添加了很大的限制。而且隐式的拷贝可能会有很大的开销。我们已经在Copy类型：move的例外中详细解释了这些因素。

13.5 Deref与DerefMut

你可以通过实现`std::ops::Deref`和`std::ops::DerefMut`trait来指定解引用运算符`*`和`.`的行为。像`Box<T>`和`Rc<T>`这样的指针类型都实现了这些trait，因此它们的行为可以和Rust内建的指针类型一样。例如，如果你有一个`Box<Complex>`类型的值`b`，那么`*b`就是`b`指向的`Complex`值，`b.re`就是它的实部。如果上下文赋值给被引用的对象，或者借用了被引用对象的可变引用，那么Rust会自动使用`DerefMut`（“可变地解引用”）trait；否则，只读的访问就已经足够了，所以它会使用`Deref`。

这个 trait 的定义类似于这样：

```
trait Deref {
    type Target: ?Sized;
    fn deref(&self) -> &Self::Target;
}

trait DerefMut: Deref {
    fn deref_mut(&mut self) -> &mut Self::Target;
}
```

`deref` 和 `deref_mut` 方法以 `Self` 的引用为参数，并返回 `Self::Target` 的引用。`Target` 应该是 `Self` 包含、拥有或指向的东西：对于 `Box<Complex>`，`Target` 类型就是 `Complex`。注意 `DerefMut` 扩展了 `Deref`：如果你可以解引用某个值并修改它，那么你当然也应该能借用它的共享引用。因为方法返回一个和 `&self` 生命周期相同的引用，只要返回的引用还存在，`self` 就会保持被借用的状态。

`Deref` 和 `DerefMut` trait 还扮演着另一个角色。因为 `deref` 以 `Self` 的引用作为参数并返回 `Self::Target` 的引用，所以 Rust 会自动把前者的类型转换为后者。换句话说，如果插入一个 `deref` 方法调用可以让类型变得匹配，那么 Rust 就会自动插入。实现 `DerefMut` 允许可变引用的相应转换。这些被称为强制解引用 (*deref coercions*)：一个类型被“强迫”表现得和另一个类型一样。

尽管你也可以自己显式写出转换来代替强制解引用，但它们确实很方便：

- 如果你有一个 `Rc<String>` 的值 `r`，并想用它调用 `String::find`，你可以简单地写 `r.find('?')` 来代替 `(*r).find('?')`：这个方法调用隐式地借用 `r`，`&Rc<String>` 被强迫为 `&String`，因为 `Rc<T>` 实现了 `Deref<Target=T>`。
- 你可以对 `String` 值调用类似 `split_at` 的方法，即使它原本是 `str` 切片类型的方法，这是因为 `String` 实现了 `Deref<Target=str>`。没有必要让 `String` 重新实现 `str` 的所有方法，因为你可以把一个 `&String` 强迫为一个 `&str`。
- 如果你有一个字节的 vector `v`，然后你想把它传递给一个接受 `&[u8]` 字节序列的函数，那你只需要传递 `&v` 作为参数，因为 `Vec<T>` 实现了 `Deref<Target=[T]>`。

如果需要的话，Rust 会尝试多次强制解引用。例如，使用上面提到的强制解引用，你可以直接用 `Rc<String>` 调用 `split_at` 方法，因为 `&Rc<String>` 解引用为 `&String`，它再解引用为 `&str`，而 `&str` 有 `split_at` 方法。

例如，假设你有下面的类型：

```
struct Selector<T> {
    /// 这个`Selector`中可用的元素
```

```

elements: Vec<T>,
/// 当前的元素在`elements`中的索引。
/// 一个`Selector`的行为类似于当前的元素的指针。
current: usize
}

```

为了让 Selector 的行为像文档中声明的一样，你必须为它实现 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]
    }
}

```

有了上面的实现之后，你可以像这样使用一个 Selector：

```

let mut s = Selector { elements: vec!['x', 'y', 'z'],
                       current: 2 };

// 因为`Selector`实现了`Deref`，我们可以使用`*`运算符来
// 引用它的当前的元素。
assert_eq!(*s, 'z');

// 断言`z`是字母，通过强制解引用直接在`Selector`上调用`char`的方法
assert!(s.is_alphabetic());

// 通过给`Selector`的引用赋值把`z`改为`w`。
*s = 'w';

assert_eq!(s.elements, ['x', 'y', 'w']);

```

Deref 和 DerefMut trait 是为智能指针例如 Box, Rc, Arc, 以及那些经常以引用的方式使用并且充当另一种类型的类型准备的，例如 Vec<T> 和 String 充当 [T] 和 str。你不应该仅仅

为了让一个类型能自动使用 Target 类型的方法（类似于 C++ 的子类可以使用基类的方法）而实现 Deref 和 DerefMut。这可能不会按照你预想的工作，并且当事情变得复杂时可能会变得令人迷惑。

强制解引用有时可能会导致困惑：Rust 使用它们来解决类型冲突，但不把它们用于满足类型变量的约束。例如，下面的代码可以正常工作：

```
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)` 中，Rust 看到了一个 `&Selector<&str>` 类型的实参，和一个 `&str` 类型的形参，然后它发现了 `Deref<Target=str>` 实现，并根据需要把调用重写为 `show_it(s.deref())`。

然而，如果你把 `show_it` 改为一个泛型函数，Rust 突然变得不配合了：

```
use std::fmt::Display;
fn show_it_generic<T: Display>(thing: T) { println!("{}", thing); }
show_it_generic(&s);
```

Rust 会报错：

```
error: `Selector<&str>` doesn't implement `std::fmt::Display`
|
33 |     fn show_it_generic<T: Display>(thing: T) { println!("{}", thing); }
|             ----- required by this bound in
|                     `show_it_generic`
34 |     show_it_generic(&s);
|             ^
|             |
|             `Selector<&str>` cannot be formatted with the
|             default formatter
|             help: consider adding dereference here: `&*s`
|
```

这可能看起来非常奇怪：为什么把函数改为泛型的就会导致这么一个错误？事实上，`Selector<&str>` 自身没有实现 `Display`，但它可以解引用成 `&str`，后者实现了。

因为你传递了一个 `&Selector<&str>` 类型的参数，而函数的形参类型是 `&T`，因此类型参数 `T` 必须是 `Selector<&str>`。然后，Rust 会检查 `T: Display` 约束是否满足：因为 Rust 不会使用强制解引用来自满足类型参数的约束，所以检查会失败。

为了解决这个问题，你可以使用 `as` 运算符显式转换：

```
show_it_generic(&s as &str);
```

或者，像编译器建议的一样，你可以使用 `&*` 来强制解引用：

```
show_it_generic(&*s);
```

13.6 Default

一些类型会有一些有意义的默认值：vector 或 string 的默认值就是空，默认的数字是 0，默认的 Option 是 None，等等。这样的类型可以实现 `std::default::Default` trait：

```
trait Default {
    fn default() -> Self;
}
```

`default` 方法简单地返回一个 `Self` 类型的新值。例如，`String` 的 `Default` 实现非常的直观：

```
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`，它们的 `default()` 方法都返回一个空集合。当你需要创建一个集合来存储一些值，但你希望代码的调用者来决定创建什么类型的集合时这会非常有用。例如，`Iterator` trait 的 `partition` 方法把迭代器产生的值划分为两个集合，使用一个闭包来决定每个值被分到哪个集合：

```
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` 使用了位操作来分辨一个数字是不是 2 的幂，`partition` 使用它来产生两个 `HashSet`。当然，`partition` 并不是只能用于 `HashSet`，你可以用它产生任何类型的集合，只要这个集合实现了 `Default` 来创建空的集合，以及 `Extend<T>` 来向集合中添加一个 `T` 类型的值。`String` 实现了 `Default` 和 `Extend<char>`，所以你可以写：

```
let (upper, lower): (String, String)
    = "Great Teacher Onizuka".chars().partition(|&c| c.is_uppercase());
```

```
assert_eq!(upper, "GTO");
assert_eq!(lower, "reat eather nizuka");
```

`Default` 的另一个常见用法是为一个结构体生成默认值，它代表一些固定的参数的集合。例如，`glum` crate 为强大且复杂的 OpenGL 图形库提供 Rust 绑定。`glum::DrawParameters` 结构体包含 24 个字段，每一个字段都控制一部分 OpenGL 渲染图形位的细节。`glum` 里的 `draw` 函数接受一个 `DrawParameters` 结构体作为参数。因为 `DrawParameters` 实现了 `Default`，你可以创建一个默认的值，然后只修改想要修改的字段，最后传给 `draw`：

```
let params = glum::DrawParameters {
    line_width: Some(0.02),
    point_size: Some(0.02),
    .. Default::default()
};

target.draw(..., &params).unwrap();
```

`Default::default()` 调用创建了一个 `DrawParameters` 的默认值，然后使用 `..` 语法创建一个新的只有 `line_width` 和 `point_size` 字段改变了的新值，最后把它传给 `target.draw`。

如果一个类型 `T` 实现了 `Default`，那么标准库会自动为 `Rc<T>`、`Arc<T>`、`Box<T>`、`Cell<T>`、`RefCell<T>`、`Cow<T>`、`Mutex<T>`、`RwLock<T>` 实现 `Default`。例如，`Rc<T>` 类型的默认值，是一个类型 `T` 的默认值的 `Rc` 指针。

如果一个元组类型的所有元素的类型都实现了 `Default`，那么元组也会自动实现 `Default`，它的默认值就是每个元素的默认值组成的元组。

Rust 不会隐式为结构体类型实现 `Default`，但如果一个结构体的所有字段都实现了 `Default`，你可以使用 `#[derive(Default)]` 来为结构体自动生成 `Default`。

13.7 AsRef 与 AsMut

如果一个类型实现了 `AsRef<T>`，那么意味着你可以从它高效地借用一个 `&T`。`AsMut` 用于可变引用。它们的定义如下：

```
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]>`，`String`实现了`AsRef<str>`。你可以从一个`String`的内容借用一个字节的数组，因为`String`也实现了`AsRef<[u8]>`。

`AsRef`通常用于让函数在接受参数时更加灵活。例如，`std::fs::File::open`函数的声明如下：

```
fn open<P: AsRef<Path>>(path: P) -> Result<File>
```

`open`真正想要的是一个`&Path`，这个类型表示一个文件系统路径。但有了这个签名以后，`open`接受任何可以被借用一个`&Path`的类型——也就是说，实现了`AsRef<Path>`的任何类型。这样的类型包括`String`和`str`，操作系统接口字符串类型`OsString`和`OsStr`、当然还有`PathBuf`和`Path`。完整的列表见标准库的文档。它允许你向`open`传递字符串字面量：

```
let dot_emacs = std::fs::File::open("/home/jimb/.emacs")?;
```

标准库中所有的文件系统函数都用这种方式接受路径参数。对于调用者来说，它的作用就像是C++中重载的函数一样，尽管Rust用了不同的方法来表示可以接受哪些类型的参数。

但只有这些还不够。字符串字面量是`&str`，但实现了`AsRef<Path>`的类型是`str`，没有`&`。正如我们在[Deref 与 DerefMut](#)一节中解释的一样，Rust不会尝试通过强制解引用满足类型参数的约束，因此它们也不能解决这个问题。

幸运的是，标准库中包含了全面的实现：

```
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，对于任意的类型`T`和`U`，如果有`T: AsRef<U>`，那么也会有`&T: AsRef<U>`：简单地先解引用然后就和之前一样继续。这里，因为有`str: AsRef<Path>`，所以也会有`&str: AsRef<Path>`。某种意义上，这是一种在检查类型参数中的`AsRef`约束时受限的强制解引用方式。

你可能会假设如果一个类型实现了`AsRef<T>`，那它也应该实现`AsMut<T>`。然而，还有一些不适合这样的情况。例如，我们提到过`String`实现了`AsRef<[u8]>`；这是有意义的，因为每一个`String`都包含一个字节缓冲区，可以被当做二进制数据访问。然而，`String`还保证这些字节是有效的UTF-8文本；如果`String`实现了`AsMut<[u8]>`，那么调用者将能把`String`的字

节缓冲区修改为任何内容，那么你将不能再信任 `String` 里存储的是有效的 UTF-8。只有那些修改返回的 `T` 不会破坏一致性的类型实现 `AsMut<T>` 才有意义。

尽管 `AsRef` 和 `AsMut` 非常简单，但它提供了一种标准化、泛型的方式来进行引用转换，进而避免了更多特化的转换 trait。当实现 `AsRef<Foo>` 就足够的时候，你应该避免实现你自己的 `AsFoo` trait。

13.8 Borrow 与 BorrowMut

`std::borrow::Borrow` trait 和 `AsRef` 很相似：如果一个类型实现了 `Borrow<T>`，那么它的 `borrow` 方法可以高效地借用一个 `&T`。但 `Borrow` 有更多的限制：只有当借用的 `&T` 和被借用的值的哈希值和比较方式相同时，这个类型才应该实现 `Borrow<T>`。（Rust 并不强迫这一点，这只是这个 trait 被设计的意图。）这让 `Borrow` 在处理哈希表和树的 key 时非常有用，在处理其他可能会被哈希或者比较的值也很有用。

当从 `String` 借用时，这种不同很重要，例如：`String` 实现了 `AsRef<str>`、`AsRef<[u8]>`、`AsRef<Path>`，但这三种类型会有不同的哈希值。只有 `&str` 保证和等价的 `String` 有相同的哈希值，因此 `String` 只实现了 `Borrow<str>`。

`Borrow` 的定义和 `AsRef` 的定义几乎完全相同，只有名字变了：

```
trait Borrow<Borrowed: ?Sized> {
    fn borrow(&self) -> &Borrowed;
}
```

`Borrow` 被设计用来解决泛型哈希表和其他关联集合类型的特性问题。例如，假设你有一个 `std::collections::HashMap<String, i32>`，把字符串映射到数字。这个表的 key 是 `String`，所以每一个表项都拥有一个 `String`。那么在一个这样的表中查找一个表项的方法的签名应该是什么样的？这是第一次尝试：

```
impl<K, V> HashMap<K, V> where K: Eq + Hash
{
    fn get(&self, key: K) -> Option<&V> { ... }
```

这么做是有道理的：为了查找一个表项，你必须提供一个和表的 key 相同的类型。但在这种情况中，`K` 是 `String`，这样的签名会强迫你每次调用 `get` 时以值传递一个 `String`，显然这很浪费。你实际上只需要一个 key 的引用：

```
impl<K, V> HashMap<K, V> where K: Eq + Hash
{
```

```
fn get(&self, key: &K) -> Option<&V> { ... }
```

这样好一点，但你现在必须传递 `&String` 作为 key，所以如果你想查找一个常量字符串，你必须这么写：

```
hashtable.get(&"twenty-two".to_string());
```

这显然很离谱：它在堆上分配了一个 `String` 的缓冲区并把文本拷贝进去，然后借用它得到一个 `&String`，传给 `get` 之后再 drop 掉。

如果可以传递任何可以被哈希或者可以与 key 比较的类型，那么显然会更好。例如，一个 `&str` 完美满足条件。因此这是最终的版本，你可以在标准库中找到它：

```
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，如果你可以将一个表项的 key 借用为一个 `&Q`，并且引用的哈希值和比较结果就和用 key 本身所得的结果一样，那么 `&Q` 就是一个可以接受的 key 类型。因为 `String` 实现了 `Borrow<str>` 和 `Borrow<String>`，所以 `get` 的最终版本可以传递 `&String` 或者 `&str` 作为 key。

`Vec<T>` 和 `[T; N]` 实现了 `Borrow<[T]>`。每一个类似字符串的类型都允许借用相应的切片类型：`String` 实现了 `Borrow<str>`，`PathBuf` 实现了 `Borrow<Path>`，等等。所有标准库的关联集合类型都使用了 `Borrow` 来决定哪些类型可以被传入查找函数。

标准库包含了完备的实现，因此每一个类型 `T` 都可以从它自身借用：`T: Borrow<T>`。这保证了在一个 `HashMap<K, V>` 中查找表项时 `&K` 总是可接受的类型。

为了方便，每一个 `&mut T` 类型也实现了 `Borrow<T>`，返回一个共享引用 `&T`。这允许你直接向集合的查找函数传递可变引用，不需要重新借用共享引用，它模拟了 Rust 通常把可变引用转换为共享引用的隐式强制解引用。

`BorrowMut` trait 是 `Borrow` 的可变引用版本：

```
trait BorrowMut<Borrowed: ?Sized>: Borrow<Borrowed> {
    fn borrow_mut(&mut self) -> &mut Borrowed;
}
```

`Borrow` 应该满足的条件也适用于 `BorrowMut`。

13.9 From 与 Into

`std::convert::From` 和 `std::convert::Into` trait 代表消耗一个值并返回另一个值的转换。与 `AsRef` 和 `AsMut` trait 从一个类型借用另一个类型的引用不同，`From` 和 `Into` 会获取参数的所有权，对它进行变换，然后把最后的结果的所有权返回给调用着。

它们的定义完美对称：

```
trait Into<T>: Sized {
    fn into(self) -> T;
}

trait From<T>: Sized {
    fn from(other: T) -> Self;
}
```

标准库自动实现每一个类型到自身的转换：每个类型 `T` 都实现了 `From<T>` 和 `Into<T>`。

尽管这两个 trait 看起来像是提供了两种不同的方式做同一件事情，但实际上它们用不同的用途。

通常使用 `Into` 来让函数可以更灵活的接收参数。例如，如果你写：

```
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` 不止可以接受 `Ipv4Addr` 作为参数，还可以接受一个 `u32` 或者一个 `[u8; 4]` 数组，因为这些类型都实现了 `Into<Ipv4Addr>`。（有时把 IPv4 地址当成单个 32 位整数或者一个四个字节的数组会很有用）因为 `ping` 唯一知道的就是 `address` 实现了 `Into<Ipv4Addr>`，所以当你调用 `into` 时没有必要指明你想要什么类型。只有一种类型可能成功，所以类型推导会自动为你填充它。

类似于上一节的 `AsRef`，它的效果也很像 C++ 中的重载函数。有了上面的 `ping` 定义，我们可以像下面这样调用：

```
println!("{}:", ping(Ipv4Addr::new(23, 21, 68, 141))); // 传递一个 Ipv4Addr
println!("{}:", ping([66, 146, 219, 98]));           // 传递一个 [u8; 4]
println!("{}:", ping(0xd076eb94_u32));             // 传递一个 u32
```

From trait 则扮演不同的角色。from 方法充当泛型的构造函数，从别的值构造一个该类型的实例。例如，Ipv4Addr 简单地实现了 From<[u8; 4]> 和 From<u32>，而不是定义两个名叫 from_array 和 from_u32 的方法：

```
let addr1 = Ipv4Addr::from([66, 146, 219, 98]);  
let addr2 = Ipv4Addr::from(0xd076eb94_u32);
```

我们可以让类型推导来决定使用哪个实现。

给定一个合适的 From 实现，标准库会自动生成相应的 Into trait 的实现。当你定义自己的类型时，如果它需要单个参数的构造函数，你可以将它写成 From<T> 的实现，而不是定义一个新的构造函数。你可以免费得到相应的 Into 实现。

因为 from 和 into 转换会获取参数的所有权，转换可以重用原始值的资源来构造转换后的值。例如，假设你写了：

```
let text = "Beautiful Soup".to_string();  
let bytes: Vec<u8> = text.into();
```

String 的 Into<Vec<u8>> 的实现简单地获取 String 的堆缓冲区的所有权，并重用它作为返回的 vector 的元素缓冲区。这个转换中不需要分配或拷贝文本。这是另一个使用 move 来实现高效的方法的例子。

这些转换还提供了一种很好的方式来放松类型的约束，灵活的同时又没有减弱被约束类型的保证。例如，一个 String 保证它的内容总是有效的 UTF-8；它的方法都有很谨慎的限制，来保证你不能引入无效的 UTF-8。但这个例子高效地把一个 String 转换成了一个普通的字节块，你可以对它做任何事：也许你准备压缩它，或者将它和其他不是 UTF-8 的二进制数据混合。因为 into 会获取参数的所有权，转换之后 text 将变为未初始化，这意味着我们可以自由地访问之前的 String 的缓冲区，而不需要破坏任何现有的 String。

然而，低开销的转换并不是 Into 和 From 的约定的一部分。与 AsRef 和 AsMut 被期望开销很小不同，From 和 Into 转换可能会分配、赋值、或者处理值的内容。例如，String 实现了 From<&str>，它把字符串切片拷贝到 String 分配的新的堆上的缓冲区。还有 std::collections::BinaryHeap<T> 实现了 From<Vec<T>>，它根据算法的要求对元素进行比较和重新排序。

? 运算符使用 From 和 Into 通过按需将特定的错误类型自动转换为通用的错误类型来帮忙处理函数中可能会失败的代码。

例如，想象一个系统需要读取二进制数据并把其中一部分被写作 UTF-8 文本的 10 进制数字转换回数字。这意味着要使用 std::str::from_utf8 和 i32 的 FromStr 实现，它们可能会返回不同类型的错误。假设我们使用了在第 7 章中定义的 GenericError 和 GenericResult 类型，那么? 运算符将会为我们做这个转换：

```

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>()?)
}

```

类似大多数错误类型，`Utf8Error` 和 `ParseIntError` 实现了 `Error` trait，并且标准库给了我们完备的 `From` impl 来把任何实现了 `Error` 的类型转换为 `Box<dyn Error>`，`?` 将会自动使用它：

```

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)
    }
}

```

它将原本需要两个 `match` 语句的复杂函数转变为只需要一行。

在 `From` 和 `Into` 被添加进标准库中之前，Rust 代码中充满了单独的转换 trait 和构造方法，每一个都用于单个类型。`From` 提供了标准的转换方式，你可以使用它们来让你的类型更加易于使用，因为你的用户已经对它们很熟悉了。其他的库和语言本身也可以使用这些 trait 作为约定的、标准的方式来处理转换。

`From` 和 `Into` 是不可失败的 trait——它们的 API 要求转换不能失败。不幸的是，很多转换更加复杂。例如，大整数例如 `i64` 可以存储比 `i32` 大很多的数字，将一个类似 `2_000_000_000_000i64` 这样的数字转换为 `i32` 没有意义。如果直接把前 32 位丢掉，那么并不会总是返回我们期望的结果：

```

let huge = 2_000_000_000_000i64;
let smaller = huge as i32;
println!("{}", smaller); // -1454759936

```

有很多处理这种问题的方法。取决于上下文，这种“折断”的方式也可能很合适。另一方面，数字信号处理和控制系统等应用通常可以使用“饱和”转换：超过最大可表示的值的数字会被限制为可表示的最大值。

13.10 TryFrom 与 TryInto

因为这种转换到底该怎么做并不是很清楚，所以 Rust 并没有为 `i32` 实现 `From<i64>`，也没有实现任何其他可能丢失信息的数制转换。作为代替，`i32` 实现了 `TryFrom<i64>`。

TryFrom 和 TryInto 是 From 和 Into 的可能失败的版本，实现 TryFrom 意味着 TryInto 也会自动实现。

它们的定义只比 From 和 Into 复杂一点：

```
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()` 方法返回一个 `Result`，因此我们可以选择在失败的情况下做什么，例如一个数字太大不能存放在结果类型中：

```
use std::convert::TryInto;
// 溢出时选择饱和，而不是折断
let smaller: i32 = huge.try_into().unwrap_or(i32::MAX);
```

如果我们想同时处理负数的情况，我们可以使用 `Result` 的 `unwrap_or_else()` 的方法：

```
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可能失败的转换也非常简单。`Error` 类型既可以很简单，也可以像某个应用的需求一样复杂。这里标准库使用了一个空的结构体，没有提供转换失败的原因相关的信息，因为唯一可能失败的原因就是溢出。另一方面，在更复杂的类型之间转换可能会需要返回更多的信息：

```
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` 将只需要简单的转换的类型联系起来，`TryFrom` 和 `TryInto` 用 `Result` 的强大的错误处理能力扩展了 `From` 和 `Into` 的简单转换。这四个 trait 可以一起使用来联系一个 crate 中的多种类型。

13.11 ToOwned

给定一个引用，产生一个有所有权的拷贝的通常方式是调用 `clone`，假设这个类型实现了 `std::clone::Clone`。但如果你想拷贝一个 `&str` 或者一个 `&[i32]` 呢？你想要的可能是一个 `String` 或者一个 `Vec<i32>`，但 `Clone` 的定义不允许那样做：从定义上讲，拷贝一个 `&T` 必须返回一个 `T` 类型的值，而 `str` 和 `[u8]` 是大小不固定的，它们甚至不能作为函数的返回类型。

`std::borrow::ToOwned` trait 提供了一种更简单的方式来把一个引用转换成一个有所有权的值：

```
trait ToOwned {
    type Owned: Borrow<Self>;
    fn to_owned(&self) -> Self::Owned;
}
```

与 `clone` 必须返回 `Self` 不同，`to_owned` 可以返回任何你可能借用一个 `&Self` 的对象：`Owned` 类型必须实现了 `Borrow<Self>`。你可以从 `Vec<T>` 借用一个 `&[T]`，所以 `[T]` 可以实现 `ToOwned<Owned=Vec<T>>`，只要 `T` 实现了 `Clone`，因此我们可以把切片的元素拷贝进 `vector` 中。类似的，`str` 实现了 `ToOwned<Owned=String>`，`Path` 实现了 `ToOwned<Owned=PathBuf>`，等等。

13.12 Borrow 和 ToOwned 的配合：Cow

想用好 Rust 就必须想好所有权的问题、例如一个函数应该以值还是以引用接受参数。通常你可以选择其中一种，并且类型的参数反应出了你的决定。但在某些情况下，在程序实际运行之前你都不能决定到底是借用还是拥有；`std::borrow::Cow` 类型（用于“写时克隆”）提供了一个处理这种情况的方法。

它的定义如下所示：

```
enum Cow<'a, B: ?Sized>
where B: ToOwned
{
```

```
Borrowed(&'a B),
Owned(<B as ToOwned>::Owned);
}
```

一个 `Cow` 可能是一个 `B` 的一个共享引用，也可能拥有一个我们可以借用到这样一个引用的值。因为 `Cow` 实现了 `Deref`，你可以将它当做 `B` 的共享引用来调用方法；如果它是 `Owned`，它会借用一个拥有的值的共享引用；如果它是 `Borrowed`，它直接使用它存储的引用。

你也可以通过调用 `to_mut` 方法来获取一个 `Cow` 值的可变引用，它会返回一个 `&mut B`。如果 `Cow` 恰巧是 `Cow::Borrowed`，`to_mut` 简单地调用 `to_owned` 方法来获取它引用的值的拷贝，然后把 `Cow` 修改为 `Cow::Owned`，然后借用新的拥有的值的可变引用。这是这个类型的名字表示的“写时克隆”的行为。

类似的，`Cow` 还有一个 `into_owned` 方法，如果需要的话它会通过引用拷贝一个值，然后返回它，把所有权移动给调用者，然后在这个过程中消耗掉 `Cow`。

`Cow` 的一个常见用途是返回一个静态分配的字符串常量或者一个计算之后得到的字符串。例如，假设你需要把一个表示错误的枚举转换为消息，其中大多数 variant 都用固定的字符串来处理，但有些 variant 应该在消息里包含一些数据。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::FileNotFoundException(path) => {
            format!("file not found: {}", path.display()).into()
        }
    }
}
```

这段代码使用了 `Into` 的 `Cow` 实现来构造值。这个 `match` 语句的大多数分支都返回一个指向静态分配字符串的 `Cow::Borrowed`。但当我们到达 `FileNotFoundException` variant 时，我们使用了 `format!` 来构造一个插入了给定文件名的消息。这个 `match` 语句的分支会产生一个 `Cow::Owned` 值。

`describe` 的调用者不需要修改值，就可以把 `Cow` 用作 `&str`：

```
println!("Disaster has struct: {}", describe(&error));
```

如果调用者需要一个拥有所有权的值，那么它可以产生一个：

```
let mut log: Vec<String> = Vec::new();  
...  
log.push(describe(&error).into_owned());
```

使用 Cow 可以帮助 describe 和它的调用者推迟内存分配，直到必须进行分配的时候。

Chapter 14

闭包

Save the environment! Create a closure today!

——Cormac Flanagan

排序一个整数的 vector 非常简单：

```
integers.sort();
```

然而一个悲伤的事实是，当我们想要对一些数据排序时，它们基本从来都不只是整数。我们通常要排序某种记录，内建的 `sort` 方法通常不能工作：

```
struct City {
    name: String,
    population: i64,
    country: String,
    ...
}

fn sort_cities(cities: &mut Vec<City>) {
    cities.sort(); // 错误：你想让它们怎么排序？
}
```

Rust 会报错 `City` 没有实现 `std::cmp::Ord`。我们需要像这样指明排序的顺序：

```
/// 按照人口排序城市的辅助函数
fn city_population_descending(city: &City) -> i64 {
    -city.population
}

fn sort_cities(cities: &mut Vec<City>) {
```

```

    cities.sort_by_key(city_population_descending); // ok
}

```

这个辅助函数 `city_population_descending`, 获取一个 `City` 记录并提取出 `key`, 我们根据这个字段来排序数据。(它返回一个负数是因为 `sort` 以递增顺序排序, 但我想以降序排序: 人多最多的城市优先。) `sort_by_key` 方法以这个返回 `key` 的函数为参数。

它可以很好地工作, 但如果将辅助函数写成一个闭包 (*closure*) (一个匿名的函数表达式) 会更简洁:

```

fn sort_cities(cities: &mut Vec<City>) {
    cities.sort_by_key(|city| -city.population);
}

```

这里的闭包是 `|city| -city.population`, 它的参数是 `city`, 返回值是 `-city.population`。Rust 会从闭包的使用中推断出参数和返回值的类型。

标准库中其他接受闭包的特性的例子包括:

- `Iterator` 的方法例如 `map` 和 `filter`。我们将在[第 15 章](#)介绍这些方法。
- 线程的 API 例如 `thread::spawn`, 它会创建一个新的系统线程。并发就是把工作移动到其他的线程, 闭包能方便地表示这种工作单元。我们将在[第 19 章](#)中介绍这些特性。
- 一些需要根据条件计算默认值的方法, 例如 `HashMap` 条目的 `or_insert_with` 方法。这个方法获取一个 `HashMap` 的表项, 或者创建一个表项, 当计算默认值有很大开销时会使用这个方法。只有当必须创建一个新表项的时候作为闭包传入的默认值才会被调用。

当然, 现在匿名函数随处可见, 即使像 Java、C#、Python、C++ 这些一开始没有的语言现在也有了。从现在开始我们将假设你已经见过匿名函数, 并专注于介绍 Rust 的闭包的独特之处。本章中, 你将学习到三种不同类型的闭包、如何将闭包和标准库方法一起使用、一个闭包如何“捕获”作用域内的变量、如何编写自己的以闭包为参数的函数和方法、以及如何存储闭包以待之后用于回调。我们将解释 Rust 的闭包是怎么实现的, 以及为什么它们比你想象的更加快速。

14.1 捕获变量

一个闭包可以使用封闭函数内的数据。例如:

```

/// 根据统计数据排序
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`，它属于当前的封闭函数 `sort_by_statistic`。我们说这个闭包“捕获”了 `stat`。这是闭包最经典的特性之一。Rust 自然支持它，但在 Rust 中，这个特性有很多需要注意的地方。

在大多数有闭包的语言中，垃圾收集扮演了重要的角色。例如，考虑这段 JavaScript 代码：

```
// 开始一段动画，重新排序表格中的城市
function startSortingAnimation(cities, stat) {
    // 排序表格时用到的辅助函数。
    // 注意这个函数用到了 stat
    function keyfn(city) {
        return city.get_statistic(stat);
    }

    if (pendingSort)
        pendingSort.cancel();

    // 现在开始动画，向它传递 keyfn。
    // 排序算法之后将会调用 keyfn。
    pendingSort = new SortingAnimation(cities, keyfn);
}
```

这个闭包 `keyfn` 被存储在新的 `SortingAnimation` 对象中。这意味着它可能会在 `startSortingAnimation` 返回之后被调用。通常来讲当一个函数返回时，它的所有变量和参数都会离开作用域并且被丢弃。但这里，JavaScript 引擎必须一直保留 `stat`，因为闭包中用到了它。大多数 JavaScript 引擎通过在堆上分配 `stat`，然后让垃圾收集器之后再回收它来做到这一点。

Rust 没有垃圾收集。那么这样的代码会如何运作？为了回答这个问题，我们先看两个例子。

14.1.1 借用值的闭包

首先，让我们重复这一节开始时的例子：

```
/// 根据统计数据排序
fn sort_by_statistic(cities: &mut Vec<City>, stat: Statistic) {
    cities.sort_by_key(|city| -city.get_statistic(stat));
}
```

在这个例子中，当 Rust 创建闭包时，它会自动借用一个 `stat` 的引用。按理来说，闭包用到了 `stat`，所以它必须有一个指向它的引用。

剩余的部分就很简单了。闭包仍然遵循我们在第 5 章中介绍的有关借用和生命周期的规则。

另外，因为闭包包含一个 `stat` 的引用，所以 Rust 不允许它比 `stat` 活得更长。因为这个闭包只在排序中使用，所以这个例子没有问题。

简单来说，Rust 通过使用生命周期代替垃圾收集来保证安全性。Rust 的方式更加快速：即使是一个很快的 GC 分配器也比 Rust 这种把 `stat` 存在栈上的方式慢。

14.1.2 偷取值的闭包

第二个例子更加棘手：

```
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
    })
}
```

这和我们上面的 JavaScript 的例子有一些像：`thread::spawn` 获取闭包并且在一个新的系统线程中调用它。注意 `||` 是闭包的空参数列表。

新的线程和调用者并行运行。当闭包返回时，新的线程也会退出。（闭包的返回值会通过一个 `JoinHandle` 值返还给调用者。我们将在[第 19 章](#)中讨论它）。

闭包 `key_fn` 仍然包含一个 `stat` 的引用。但这一次，Rust 不能保证引用会被安全使用。因此 Rust 会拒绝这个程序：

```
error[E0373]: closure may outlive the current function, but it borrows `stat`,
which is owned by the current function
--> closures_sort_thread.rs:33:18
|
33 |     let key_fn = |city: &City| -> i64 { -city.get_statistic(stat) };
|           ^^^^^^^^^^^^^^^^^^^^^^          ^^^
|           |                           `stat` is borrowed here
|           may outlive borrowed value `stat`
```

（事实上，这里有两个问题，因为 `cities` 也被不安全地共享。）简单来说，`thread::spawn` 新创建的线程不能保证在 `cities` 和 `stat` 被销毁之前完成工作。

解决这两个问题的方法是一样的：告诉 Rust 把 `cities` 和 `stat` 移动 (*move*) 进新的闭包，而不是借用它们的引用：

```

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
    })
}

```

我们唯一修改的地方就是在两个闭包前面都加上了 `move` 关键字。`move` 关键字告诉 Rust 闭包并不是借用它用到的值，而是偷取它们。

第一个闭包 `key_fn`，获取了 `stat` 的所有权。然后第二个闭包获取了 `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式：移动和借用。实话说没有什么更多要说的了，闭包遵守我们在第 4 章和第 5 章中提到的移动和借用的规则。这里还有一些情况的解释：

- 就像语言中其他任何部分一样，如果一个闭包 `move` 一个可拷贝类型，例如 `i32`，那么它会拷贝这个值。因此如果 `Statistic` 恰巧是个可拷贝类型，那么即使创建了使用了它的 `move` 闭包之后，我们仍然可以继续使用 `stat`。
- 非拷贝类型，例如 `Vec<City>`，会真的被移动：上面的代码会通过 `move` 闭包把 `cities` 移动到新线程。Rust 不允许我们在创建了这个闭包之后再访问 `cities`。
- 这里，在闭包移动 `cities` 之后，我们不需要再使用 `cities`。如果我们还需要用到它，那么我们可以先克隆 `cities` 并把拷贝存储到不同的变量中。闭包将只偷取拷贝中的一个——不管它用的是哪一个。

Rust 的严格规则给我们带来了很重要的一个保证：线程安全。正因为 vector 被移动了，而不是在线程之间共享，所以我们可以知道旧的线程不会在新的线程修改 vector 时释放它。

14.2 函数和闭包类型

在本章中，我们已经见到过一些用作值的函数和闭包。自然地，这意味着它们也有类型。例如：

```

fn city_population_descending(city: &City) -> i64 {
    -city.population
}

```

这个函数接受一个参数（一个`&City`），然后返回一个`i64`。所以它的类型是`fn(&City) -> i64`。

你可以像操作其他值一样对函数进行各种操作。你可以将它们存储在变量中。你可以使用通常的 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。泛型类型例如`Vec`可以存储函数，只要它们有相同的`fn`类型。而且一个函数类型的值非常小：一个`fn`值只是函数的机器码的内存地址，就像 C++ 中的函数指针一样。

一个函数可以获取另一个函数作为参数。例如：

```
/// 给定一个城市的列表和一个测试函数,
/// 返回有多少城市通过了测试。
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
}

/// 一个测试函数的例子。注意这个函数的类型是
/// `fn(&City) -> bool`，和`count_selected_cities`'
/// 的`test_fn`参数的类型一样。
fn has_monster_attacks(city: &City) -> bool {
    city.monster_attack_risk > 0.0
}

// 有多少城市面临被怪物袭击的危险？
let n = count_selected_cities(&my_cities, has_monster_attacks);
```

如果你熟悉 C/C++ 中的函数指针，你会发现 Rust 的函数类型的值其实就是一样的东西。除了这些之外，你可能会很惊讶闭包的类型和函数不同：

```
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，我们必须修改函数的签名。它需要看起来像这样：

```
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
}
```

我们只修改了 `count_selected_cities` 的类型签名，没有修改函数体。新的版本是泛型的。它接受任何 `F` 类型的参数 `test_fn`，只要 `F` 实现了特殊的 trait `Fn(&City) -> bool`。接受单个 `&City` 并返回一个布尔值的所有函数和大部分闭包会自动实现这个 trait：

```
fn(&City) -> bool // fn 类型（只限函数）
Fn(&City) -> bool // Fn trait（函数和闭包）
```

这个特殊的语法是语言内建的。`->` 和返回类型是可选的。如果省略，返回类型是 `()`。

新版本的 `count_selected_cities` 接受一个函数或者闭包：

```
count_selected_cities(
    &my_cities,
    has_monster_attacks); // ok

count_selected_cities(
    &my_cities,
    |city| city.monster_attack_risk > limit); // 也 ok
```

为什么第一次尝试不能工作？这是因为闭包虽然可以调用，但它不是一个 `fn`。闭包 `|city| city.monster_attack_risk > limit` 有自己的类型，不是一个 `fn` 类型。

事实上，你写的每一个闭包都有自己的类型，因为一个闭包可能包含数据：从作用域中借用或偷取的值。这可能是任意数量、任意类型的变量。因此每一个闭包都有一个编译器创建的特定的类型，大到足够存储数据。没有类型完全相同的两个闭包。但每一个闭包都实现了 Fn trait；我们的例子中的闭包实现了 Fn(&City) -> i64。

因为每一个闭包都有自己的类型，所以处理闭包的代码通常要是泛型的，比如 count_selected_cities。每次都要写出这种泛型类型有些笨拙，但要了解这种设计的优点，请继续阅读。

14.3 闭包的性能

Rust 的闭包被设计得很快：比函数指针更快，快到你可以在调用非常频繁、性能敏感的代码中使用它们。如果你熟悉 C++ 的 lambda，你将会发现 Rust 的闭包和它一样快和紧凑，但更安全。

在大多数语言中，闭包都在堆上分配、动态分发、被垃圾收集器回收。因此创建、调用、回收每个闭包都需要消耗额外的 CPU 时间。更糟的是，闭包常常会排除内联 (*inline*)，这是一种编译器用于消除函数调用的开销并为其它优化提供支持的关键技术。总的来说，那些语言中的闭包太慢，以至于需要从紧密的内层循环中手动移除它们。

Rust 的闭包没有任何这些性能缺陷。它们没有垃圾回收。和 Rust 中的其他所有东西一样，除非你把它们放在 Box、Vec 或其他容器中，否则它们不会在堆上分配。并且因为每个闭包都有不同的类型，任何时候 Rust 编译器都知道你要调用的闭包的类型，它可以为那个特殊的闭包内联代码。这让闭包完全可以在紧密的循环中使用，并且 Rust 程序通常确实会踊跃地这么做，正如你将在第 15 章中看到的一样。

图 14-1 展示了 Rust 闭包在内存中的布局。在图片的顶部，我们展示了我们的闭包引用的两个局部变量：一个字符串 food 和一个简单的枚举 weather，它的数字值恰好是 27。

闭包 (a) 使用了两个变量。显然我们在查找同时有炸玉米和龙卷风的城市。在内存中，这个闭包看起来像一个包含两个引用的结构体。

注意并没有一个指向它的代码的指针！这没有必要：只要 Rust 知道闭包的类型，它就知道当你调用闭包时要运行哪里的代码。

闭包 (b) 基本相同，除了它是一个 move 闭包，因此它包含值而不是引用。

闭包 (c) 并没有使用环境中的任何变量。结构体是空的，因此这个闭包不会占用任何内存。

正如图中所示，这些闭包并不会占用太多空间。但在实践中即使这一点空间有时候也不需要。通常，编译器可以内联一个闭包的所有调用，并且即使上面的图中展示的小结构体也可能被优化掉。

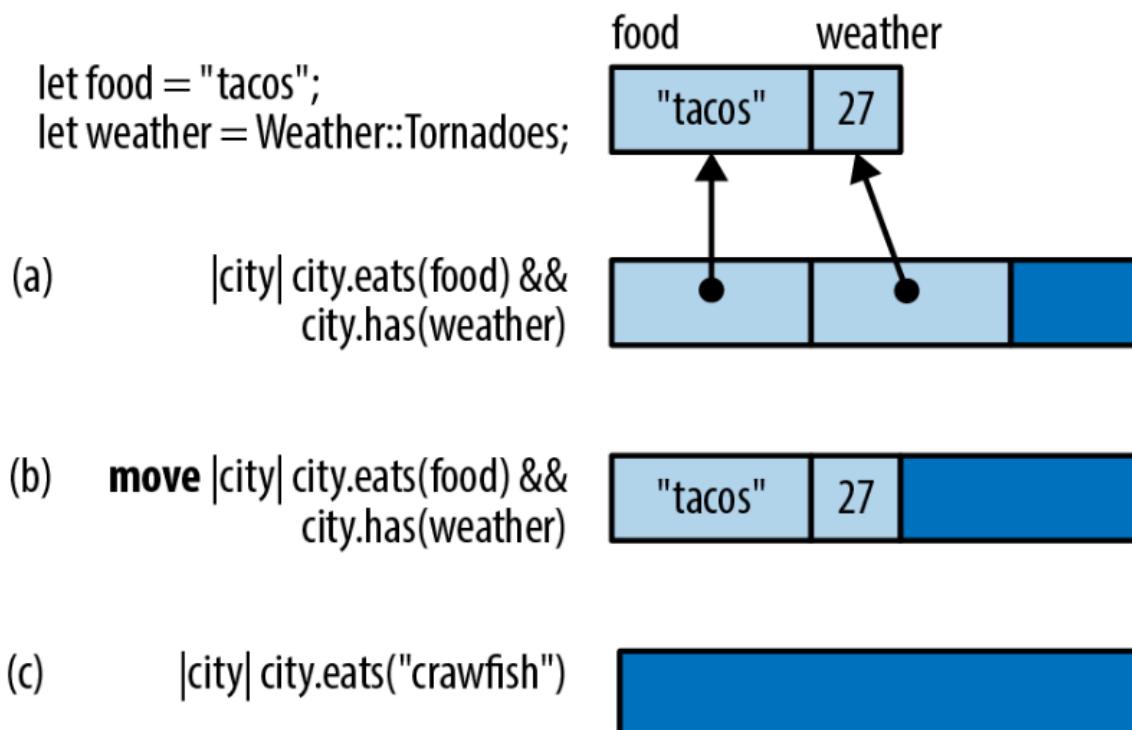


图 14-1: 闭包的内存布局

在[回调](#)中，我们将展示怎么在堆上分配闭包并使用 trait 对象动态地调用它们。这样会稍微慢一点，但它仍然和其他的 trait 对象的方法一样快。

14.4 闭包和安全性

本章中到目前为止，我们已经讨论了 Rust 怎么保证闭包在借用或者移动环境中的值时遵循语言的安全规则。但还有更多不明显的情况。在这一节中，我们会解释当一个闭包 drop 或修改一个捕获的值时会发生什么。

14.4.1 杀死值的闭包

我们已经看到过借用值和偷取值的闭包，但只有它们还不够，还需要更多种类的闭包。

当然，[杀死](#) (*kill*) 并不是真正正确的术语。在 Rust 中，我们丢弃 (*drop*) 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 `drop`。

所以如果我们调用它两次会发生什么？

```
f();
f();
```

让我们深入思考它。第一次调用 `f` 时，它 `drop` 了 `my_str`，这意味着存储字符串的内存已经被释放了，返还给了系统。第二次调用 `f` 时，会发生同样的事情。这是两次释放 (*double free*)，在 C++ 编程中这是一种会导致未定义行为的经典错误。

`drop` 一个 `String` 两次在 Rust 中也是同样的错误行为。幸运的是，Rust 不会这么简单就被骗过：

```
f();    // ok
f();    // 错误：使用了被 move 的值
```

Rust 知道这个闭包不能被调用两次。

一个只能被调用一次的闭包看起来像是一个很特殊的东西，但是我们已经在整本书中都讨论过所有权和生命周期了。值被消耗（即 `move`）的 idea 是 Rust 的核心概念之一。它在闭包中的表现和其他情况中一样。

14.4.2 FnOnce

让我们再一次尝试骗过 Rust、丢弃一个 `String` 两次。这次，我们使用这个泛型函数：

```
fn call_twice<F>(closure: F) where F: Fn() {
    closure();
    closure();
}
```

这个函数可能被传入任何实现了 `Fn()` trait 的闭包：即没有参数并且返回 `()` 的闭包。（和函数一样，当返回值是 `()` 时可以省略；`Fn()` 是 `Fn() -> ()` 的缩写。）

现在如果我们把我们的不安全的闭包传递给这个泛型函数会发生什么？

```
let my_str = "hello".to_string();
let f = || drop(my_str);
call_twice(f);
```

这个闭包仍然在调用时 `drop my_str`。调用它两次将是两次释放。不过 Rust 仍然没有被迷惑：

```
error[E0525]: expected a closure that implements the `Fn` trait, but this closure only implements `FnOnce`
--> closure_twice.rs:12:13
|
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
```

错误信息告诉了我们更多有关 Rust 如何处理“杀死值的闭包”的信息。原本 Rust 可以直接禁止这类闭包，它们有时也是有用的。因此，Rust 限制了它们的使用：drop 值的闭包，例如 `f`，不允许实现 `Fn`（它们显然也不应该是 `Fn`）。它们实现了一个相对弱一些的 trait `FnOnce`，表示只能调用一次的闭包。

当你第一次调用 `FnOnce` 闭包时，闭包本身会被消耗掉 (*the closure itself is used up*)。`Fn` 和 `FnOnce` 这两个 trait 就好像是以如下方式定义的：

```
// 没有参数的`Fn`和`FnOnce` trait 的伪代码
trait Fn() -> R {
    fn call(&self) -> R;
}

trait FnOnce() -> R {
    fn call_once(self) -> R;
}
```

就像算术表达式例如 `a + b` 是方法调用 `Add::add(a, b)` 的缩写一样，Rust 把 `closure()` 看做是上面的例子中展示的两个闭包方法之一。对于一个 `Fn` 闭包，`closure()` 会展开为 `closure.call()`。这个方法以引用获取 `self`，因此闭包本身没有被移动。但如果闭包只有第一次调用时是安全的，那么 `closure()` 会展开为 `closure.call_once()`。这个方法以值获取 `self` 参数，因此闭包会被消耗掉。

当然，我们一直在故意使用 `drop` 制造麻烦。在实际中，你只会偶尔遇到这种情况。它并不会经常发生，但偶尔你会不经意间编写出消耗掉一个值的闭包代码：

```
let dict = produce_glossary();
let debug_dump_dict = || {
    for (key, value) in dict { // oops!
        println!("{} - {}", key, value);
    }
};
```

然后，当你调用 `debug_dump_dict()` 不止一次时，你会得到一个类似这样的错误信息：

```

error[E0382]: use of moved value: `debug_dump_dict`
--> closures_debug_dump_dict.rs:18:5
|
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
--> src/main.rs:13:29
|
13 |         for (key, value) in dict {
|             ^

```

为了调试这个错误，我们需要搞清楚为什么这个闭包是 FnOnce。这里什么值被消耗了？编译器友好地指出了是 dict，在这个例子中，它也是我们唯一使用的变量。哦，这个 bug 是：我们直接迭代 dict 消耗了它。我们应该迭代 &dict，以引用访问值，而不是迭代 dict：

```

let debug_dump_dict = || {
    for (key, value) in &dict { // 不会消耗 dict
        println!("{} - {}", key, value);
    }
}

```

这样就修复了错误；这个函数现在是 Fn，可以被调用任意次。

14.4.3 FnMut

还有一种闭包，这种闭包包含可变的数据或者 mut 引用。

Rust 认为 non-mut 的值可以安全在线程间共享。但在线程间共享包含 mut 数据的 non-mut 闭包不是安全的：在多个线程中调用这种闭包可能会导致各种数据竞争，就和多个线程同时读写同一份数据一样。

因此，Rust 又分出了一种闭包类别 FnMut，这个类别用于有写入操作的闭包。FnMut 闭包以 mut 引用调用，就好像它们被定义为这样：

```

// `Fn`、`FnMut`、`FnOnce` trait 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;
}

```

所有需要值的 `mut` 方法，但不会 drop 任何值的闭包，都是一个 `FnMut` 闭包。例如：

```

let mut i = 0;
let incr = || {
    i += 1; // incr 借用了 i 的一个可变引用
    println!("Ding! i is now: {}", i);
};

call_twice(incr);

```

我们编写的 `call_twice` 需要一个 `Fn`。因为 `incr` 是一个 `FnMut` 而不是 `Fn`，所以这段代码会编译失败。然而，有一种简单的修复方法。为了理解这种修复方法，让我们后退一步，总结一下你学到的有关 Rust 的三种闭包的知识：

- `Fn` 是你可以没有限制地调用多次的闭包和函数家族。这个最高的类别还包括所有的 `fn` 函数。
- `FnMut` 是如果闭包本身被声明为 `mut` 时可以调用多次的闭包家族。
- `FnOnce` 是当调用者拥有它时可以调用一次的闭包家族。

每一个 `Fn` 都满足 `FnMut` 的要求，每一个 `FnMut` 都满足 `FnOnce` 的要求。如图 14-2 所示，它们并不是独立的三个类别。

`Fn()` 是 `FnMut()` 的一个子集，`FnMut()` 又是 `FnOnce()` 的一个子集。这使得 `Fn` 是最独特和强大的分类。`FnMut` 和 `FnOnce` 是范围更广一些的分类，它们包含有一些使用限制的闭包。

现在我们已经梳理了我们所知的内容，显然为了尽可能接受更多的闭包类型，我们的 `call_twice` 实际上应该接受所有的 `FnMut` 闭包，像这样：

```

fn call_twice<F>(mut closure: F) where F: FnMut() {
    closure();
    closure();
}

```

原本第一行的约束是 `F: Fn()`，现在是 `F: FnMut()`。有了这个修改之后，我们仍然可以接受所有的 `Fn` 闭包，并且现在还可以对可变的数据调用 `call_twice`：

```

let mut i = 0;
call_twice(|| i += 1); // ok!
assert_eq!(i, 2);

```

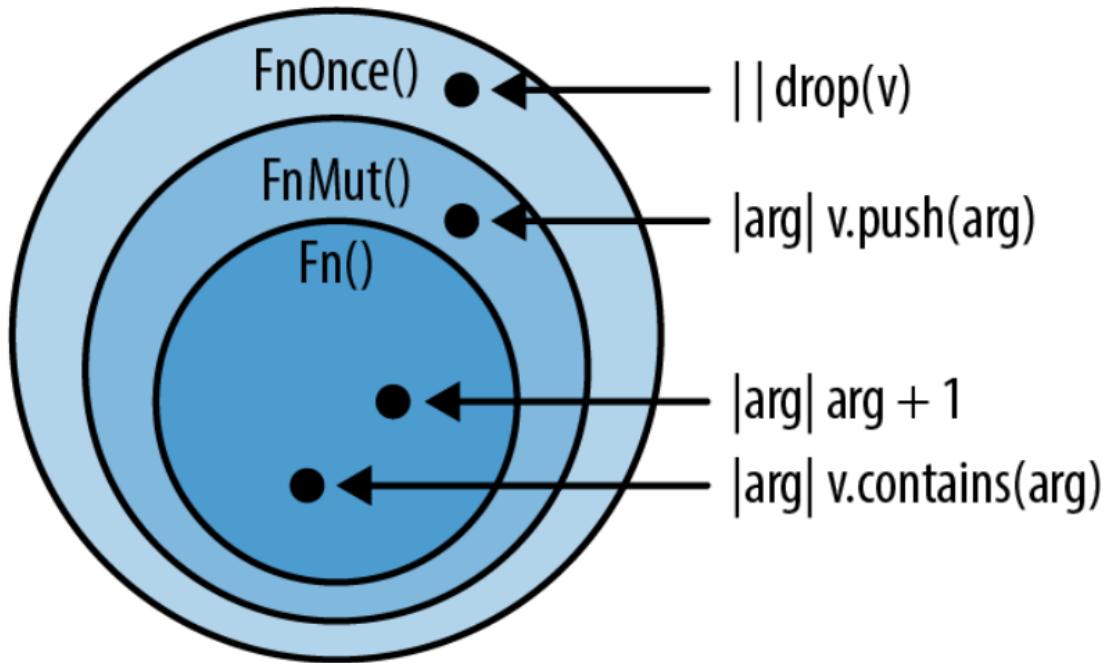


图 14-2: 三种闭包类别的维恩图

14.4.4 闭包的 Copy 和 Clone

正如 Rust 能自动分辨出哪些闭包只能调用一次一样，它也能分辨出哪些闭包可以实现 Copy 和 Clone，哪些不能。

正如我们之前解释的一样，闭包被表示为包含它们捕获的值（move 闭包）或者引用（non-move 闭包）的结构体。闭包的 Copy 和 Clone 的规则就类似于普通结构体的 Copy 和 Clone 的规则。一个没有可变变量的 non-move 的闭包只有共享引用，共享引用是 Clone 和 Copy，所以这种闭包也是 Clone 和 Copy：

```
let y = 10;
let add_y = |x| x + y;
let copy_of_add_y = add_y;           // 这个闭包是`Copy`，因此...
assert_eq!(add_y(copy_of_add_y(22)), 42); // ...我们可以使用这两个。
```

另一方面，一个有可变值的 non-move 闭包在内部的表示中包含可变引用。可变引用既不是 Clone 也不是 Copy，因此这样的一个闭包既不是 Copy 也不是 Clone：

```
let mut x = 0;
let mut add_to_x = |n| { x += n; x };

let copy_of_add_to_x = add_to_x;           // 移动，而不是拷贝
assert_eq!(add_to_x(copy_of_add_to_x(1)), 2); // 错误：使用了被移动的值
```

对于 move 闭包来说，规则变得更简单了。如果一个 move 闭包捕获的所有值都是 Copy，那么它也是 Copy。如果它捕获的所有值都是 Clone，那么它也是 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");
```

这个`.clone()(...)`语法有一点奇怪，但它只是因为我们克隆了闭包然后调用它。这个程序输出：

```
Hello, Alfred
Hello, Bruce
```

当 `greeting` 在 `greet` 中被用到时，它会被移动进表示 `greet` 的内部结构体里，因为它是 move 闭包。因此，当我们克隆 `greet` 时，它里面的所有内容都会被克隆。这里有两个 `greeting` 的拷贝，当调用克隆的 `greet` 时它们会被独立地修改。它本身用处不大，但当你需要把同样的闭包传递给不止一个函数时，它会很有用。

14.5 回调

很多库使用回调 (*callback*) 作为部分 API：一种由用户提供、之后会被库调用的函数。事实上，你已经在这本书中看到过一些这样的 API 了。回顾第 2 章，我们使用了 `actix-web` 框架编写了一个简单的 web 服务器。程序中很重要的一部分就是路由，看起来像这样：

```
App::new()
    .route("/", web::get().to(get_index))
    .route("/gcd", web::post().to(post_gcd))
```

路由的目的是把到来的网络请求发送到处理特定请求的函数。在这个例子中，`get_index` 和 `post_gcd` 是我们在程序里其他地方用 `fn` 关键字声明的函数的名字，但我们也可以传递一个闭包，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` 被设计为可以接受任何线程安全的 Fn 作为参数。

那么我们怎么在自己的程序中做到这一点呢？让我们尝试写出我们自己的简单的路由，不使用任何 `actix-web` 的代码。我们首先声明一些表示 HTTP 请求和响应的类型：

```

struct Request {
    method: String,
    url: String,
    headers: HashMap<String, String>
    body: Vec<u8>
}

struct Response {
    code: u32,
    headers: HashMap<String, String>
    body: Vec<u8>
}

```

现在一个路由器的任务就是简单地存储一个把 URL 映射到回调函数的表，这样可以按需调用正确的回调函数。（为了简单起见，我们只允许用户创建匹配单个具体 URL 的路由。）

```

struct BasicRouter<C> where C: Fn(&Request) -> Response {
    routes: HashMap<String, C>
}

impl<C> BasicRouter<C> where C: Fn(&Request) -> Response {
    /// 创建一个空的路由器。
    fn new() -> BasicRouter<C> {
        BasicRouter { routes: HashMap::new() }
    }

    /// 向路由器中添加一条路由。
    fn add_route(&mut self, url: &str, callback: C) {
        self.routes.insert(url.to_string(), callback);
    }
}

```

不幸的是，我们犯了一个错误。你注意到它了吗？

如果我们只添加一个路由，那么这个路由器可以工作的很好：

```
let mut router = BasicRouter::new();
router.add_route("/", |_| get_form_response());
```

这可以正常编译并运行。然而，如果我们添加另一个路由：

```
router.add_route("/gcd", |req| get_gcd_response(req));
```

那我们会得到错误：

```
error[E0308]: mismatched types
--> closures_bad_router.rs:41:30
|
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
```

我们的错误在于定义 `BasicRouter` 类型的方式：

```
struct BasicRouter<C> where C: Fn(&Request) -> Response {
    routes: HashMap<String, C>
}
```

我们不知不觉中声明了每个 `BasicRouter` 都只有单个回调类型 `C`，并且 `HashMap` 里的所有回调函数都是这个类型。回顾[选择哪一种](#)，我们展示了一个有同样问题的 `Salad` 类型：

```
struct Salad<V: Vegetable> {
    veggies: Vec<V>
}
```

这里的解决方法和沙拉问题的解决方法一样：因为我们想支持很多类型，我们需要使用 `box` 和 `trait` 对象：

```
type BoxedCallback = Box<dyn Fn(&Request) -> Response>

struct BasicRouter {
    routes: HashMap<String, BoxedCallback>
}
```

每一个 box 都可以包含一个不同类型的闭包，因此一个 `HashMap` 可以包含很多种类的回调函数。注意类型参数 `C` 消失了。

这需要对方法进行一些调整：

```
impl BasicRouter {
    // 创建一个空的路由器。
    fn new() -> BasicRouter {
        BasicRouter { routes: HashMap::new() }
    }

    // 向路由器中添加一条路由。
    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));
    }
}
```

NOTE

注意 `add_route` 的类型签名中 `C` 的两个约束：一个特定的 `Fn` trait 和一个 `'static` 生命周期。Rust 让我们添加这个 `'static` 约束。如果没有它，`Box::new(callback)` 的调用将会导致错误，因为如果一个闭包包含可能会离开作用域的变量的引用，那么存储这样的闭包是不安全的。

最后，我们的简单路由已经准备好处理到来的请求：

```
impl BasicRouter {
    fn handle_request(&self, request: &Request) -> Response {
        match self.routes.get(&request.url) {
            None => not_found_response(),
            Some(callback) => callback(request)
        }
    }
}
```

作为灵活性的代价，我们也可以使用函数指针 (*function pointer*) 或者 `fn` 类型来代替 trait 对象，这样空间效率会高一点。像 `fn(u32) -> u32` 这样的类型，和闭包很像：

```
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，不捕获环境中任何变量的闭包和函数指针完全相同，因为它们不需要存储关于被捕获变量的额外信息。如果你指定了合适的 fn 类型，不管是在绑定中还是在函数签名中，编译器都会乐于让你使用它们：

```
let closure_ptr: fn(u32) -> u32 = |x| x + 1;
let two = closure_ptr(1); // 2
```

与那些捕获的闭包不同，这些函数指针只占据一个 `usize` 的空间。

函数指针也可以用于实现我们自己的动态分发，而不是使用编译器建议的 `Box dyn Fn()`：

```
struct FnPointerRouter {
    routes: HashMap<String, fn(&Request) -> Response>
}
```

这里，`HashMap` 只为每一个 `String` 存储一个 `usize`，这里没有 `Box`。除了 `HashMap` 自身，没有任何动态分配。当然，这些方法也需要被调整：

```
impl FnPointerRouter {
    // 创建一个空的路由器。
    fn new() -> FnPointerRouter {
        FnPointerRouter { routes: HashMap::new() }
    }

    // 向路由器中添加一条路由。
    fn add_route(&mut self, url: &str, callback: fn(&Request) -> Response)
    {
        self.routes.insert(url.to_string(), callback);
    }
}
```

如图 14-1 所示，闭包会有单独的类型是因为每一个闭包都捕获了不同的变量，因此它们每一个的大小都不同。如果它们不捕获任何内容，就没有任何东西需要存储。在接受回调的函数中使用 `fn` 指针，可以限制调用只能使用没有捕获东西的闭包，这样在代码内能获取一些性能和灵活性的改善，但对于使用你的 API 的用户来说要付出灵活性的代价。

14.6 高效地使用闭包

正如我们所见，Rust 的闭包和大多数其他语言的闭包不同。最大的不同是在有 GC 的语言中，你可以在闭包中使用局部变量，并且不需要考虑生命周期和所有权。没有了 GC，事情就变得不一样了。一些在 Java、C#、JavaScript 中很普遍的设计模式在 Rust 中如果不做修改将不能工作。

以图 14-3 中的模型-视图-控制器设计模式（简称为 MVC）为例。对于用户界面的每一个元素，MVC 框架会创建三个对象：一个模型 (*model*) 表示 UI 元素的状态，一个视图 (*view*) 负责它的外观，一个控制器 (*controller*) 处理用户的交互。过去这些年里有不计其数的 MVC 的变体被实现，但总体思路都是三个对象以某种方式分配 UI 职责。

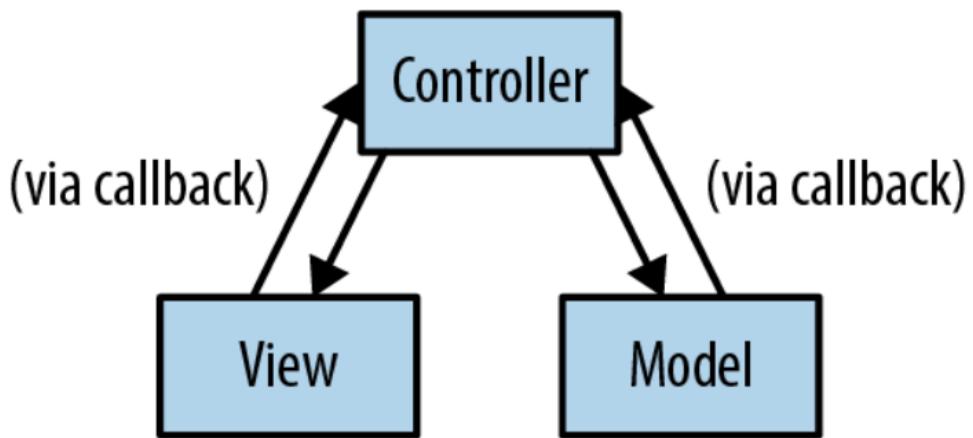


图 14-3: 模型-视图-控制器设计模式

这就是问题。通常来讲，如图 14-3 所示，每个对象有另外一个或者另外两个的引用，直接的或者是通过回调函数的。当其中一个发生了变化时，它会通知其他的两个，因此它们也会一起更新。哪一个对象“拥有”其他两个对象的问题永远不会出现。

在 Rust 中如果不做修改将不能实现这个模式。必须明确所有权，必须消除循环引用。模型和控制器不能有直接到彼此的引用。

Rust 的赌注是有更好的替代设计。有时你可以让每一个闭包接受引用作为参数来解决闭包的所有权和生命周期的问题。有时你可以给系统中的每个部分赋予一个数字，然后传递这些数字来代替引用。或者你可以实现一种其中的对象并不是都有彼此引用的 MVC 的变体。或者将你的工具集建模成单向的数据流架构的非 MVC 系统，例如 Facebook 的 Flux 架构，如图 14-4 所示。

简单来说，如果你尝试使用 Rust 的闭包来制造一个“对象之海”，你会遇到很多困难。但有替代的方案。在例子中的这个场景，软件工程这门学科已经开始倾向于替代方案，因为它

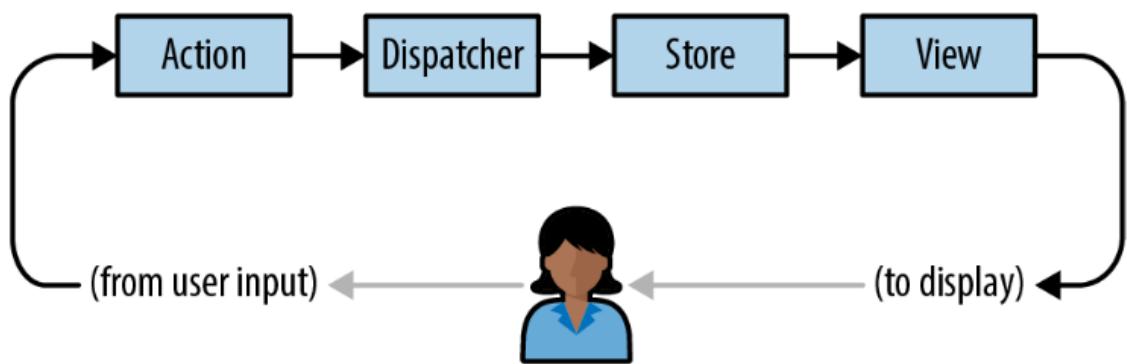


图 14-4: Flux 架构, MVC 的一个替代

们更简单。

在下一章中，我们将开始一个闭包真正闪耀的话题。我们将书写一种充分利用 Rust 闭包的简洁、快速、高效的优势的代码。并且它们写起来很有趣、读起来很轻松、非常实用。下一章：Rust 迭代器。

Chapter 15

迭代器

It was the end of a very long day.

——Phil

一个迭代器 (*iterator*) 可以产生一个值的序列，通常会使用一个循环来进行处理。Rust 的标准库提供了遍历 vector、字符串、哈希表和其他集合的迭代器，以及从一个输入流中产生若干行文本的迭代器、到达网络服务器的连接的迭代器、通过通道从其他线程接收到的值的迭代器，等等。当然，你可以实现自己的迭代器。Rust 的 `for` 循环提供了一种自然地使用迭代器的语法，但迭代器自身也提供了丰富的方法集合用于映射、过滤、连接、收集等用途。

Rust 的迭代器灵活、表达力强、高效。考虑下面的函数，它返回前 n 个正数的和（通常也被称为第 n 个三角数 (*nth triangle number*)）：

```
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>` 值。一个 `RangeInclusive<i32>` 是一个产生从起点到终点的所有整数的迭代器（包含起点和终点），因此你可以将它用作 `for` 循环的操作数来求 1 到 n 的和。

但迭代器也有一个 `fold` 方法，你可以使用它实现如下的等价定义：

```
fn triangle(n: i32) -> i32 {
    (1..=n).fold(0, |sum, item| sum + item)
}
```

以 0 作为起始的总和，`fold` 会获取 `1..=n` 产生的每个值，然后用总和和产生的值调用闭包 `|sum, item| sum + item`，每一次闭包的返回值就是新的总和。它最后返回的值就是 `fold` 自身返回的值——在这个例子中，就是整个序列的总和。如果你习惯使用 `for` 和 `while` 循环，那么这看起来会有些奇怪，但一旦你习惯了它，`fold` 就是一个可读性强而简洁的替代方案。

这种写法是函数式编程语言的标准写法，这使得表达式有更强的表现力。但 Rust 的迭代器是精心设计的，为了保证编译器可以把它们翻译成优秀的机器代码。在 `release` 构建模式下构建上面第二个定义时，Rust 知道 `fold` 的定义，并且把它内联进 `triangle`。然后，闭包 `|sum, item| sum + item` 也会被内联。最后，Rust 会检查组合之后的代码，然后发现有一种更简单的方法计算从 1 到 `n` 的和：和总是等于 $n * (n+1) / 2$ 。Rust 会把 `triangle` 的整个函数体，包括循环、闭包等所有内容，变成一次乘法指令和一些其他的位运算。

这个例子恰巧可以转换成简单的算术，但在更复杂的使用中迭代器也可以表现的很好。它们是 Rust 提供灵活抽象的同时只有很小甚至没有开销的另一个例子。

在本章中，我们将会解释：

- `Iterator` 和 `IntoIterator` trait，它们是 Rust 迭代器的基础
- 经典迭代器流水线的三个阶段：从初始的值创建一个迭代器；通过选择或处理值将一种迭代器变成另一种；消耗迭代器产生的值
- 如何为自己的类型实现迭代器

迭代器有很多方法，所以一旦你了解了大概的思路，就可以跳过那一节。但迭代器在 Rust 的习惯用法中非常普遍，熟悉这些随附的工具对掌握这门语言至关重要。

15.1 Iterator 与 IntoIterator trait

一个迭代器是任何实现了 `std::iter::Iterator` trait 的类型：

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
    ... // 很多默认方法
}
```

`Item` 是迭代器产生的值的类型。`next` 方法可能返回 `Some(v)`，其中 `v` 是迭代器的下一个值；或者返回 `None`，表示已经到达序列的终点。这里我们省略了 `Iterator` 的很多默认方法；我们将在本章的剩余部分分别介绍它们。

如果某个类型有一种很自然的迭代方法，那么它可以实现 `std::iter::IntoIterator`，它的 `into_iter` 方法获取一个值并返回一个迭代它的迭代器：

```
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，`Item` 是它产生的值的类型。我们称所有实现了 `IntoIterator` 的类型为可迭代对象 (*iterable*)，因为你可以迭代它。

Rust 的 `for` 循环将这些部分漂亮地组合在一起。为了迭代一个迭代器的元素，你可以写：

```
println!("There's:");
let v = vec!["antimony", "arsenic", "alumium", "selenium"];

for element in &v {
    println!("{}", element);
}
```

在底层，每一个 `for` 循环只是 `IntoIterator` 和 `Iterator` 的方法调用的缩写：

```
let mut iterator = (&v).into_iter();
while let Some(element) = iterator.next() {
    println!("{}", element);
}
```

`for` 循环使用了 `IntoIterator::into_iter` 来把操作数 `&v` 转换成一个迭代器，然后重复调用 `Iterator::next`。每一次返回 `Some(element)` 时，`for` 循环会执行循环体；如果它返回 `None`，循环会终止。

考虑这个例子，其中有一些迭代器的术语：

- 正如我们所说，迭代器 (*iterator*) 是任何实现了 `Iterator` 的类型。
- 可迭代对象 (*iterable*) 是任何实现了 `IntoIterator` 的类型：你可以调用它的 `into_iter` 方法获得一个迭代它的迭代器。这个例子中 `vector` 的引用 `&v` 就是可迭代对象。
- 一个迭代器产生 (*produce*) 值。
- 迭代器产生的值是 *item*。这里 *item* 是 "antimony", "arsenic"，等等。
- 接受迭代器产生的 *item* 的代码是消费者 (*consumer*)。这个例子中，`for` 循环就是消费者。

尽管 `for` 循环总是调用操作数的 `into_iter`，你也可以直接向 `for` 循环传递迭代器；例如，当你在 `Range` 上循环时就是这种情况。所有的迭代器都会自动实现 `IntoIterator`，它们的 `into_iter` 方法简单地返回迭代器自身。

如果在迭代器返回了 `None` 之后，你再调用它的 `next` 方法，那么 `Iterator` trait 并没有指定这种情况下该怎么做。大多数会再次返回 `None`，但不是所有。（如果这导致了问题，`fuse` 中介绍的 `fuse` 适配器可能会有帮助。）

15.2 创建迭代器

Rust 标准库文档中详细解释了每种类型提供哪些种类的迭代器，但标准库提供了一些通用的约定来帮助你找到需要的迭代器。

15.2.1 `iter` 和 `iter_mut` 方法

大多数集合类型提供 `iter` 和 `iter_mut` 方法，它们返回一个迭代器，迭代器会产生每一个 item 的共享引用或可变引用。数组切片例如 `&[T]` 和 `&mut [T]` 也有 `iter` 和 `iter_mut` 方法。除了使用 `for` 循环自动处理之外，这些方法是最常用的获得迭代器的方法：

```
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 item 类型是 `&i32`：每一次调用 `next` 都会产生下一个元素的引用，直到到达 vector 的终点。

每一个类型都可以实现 `iter` 和 `iter_mut`，不管它们的实现是否有意义。`std::path::Path` 的 `iter` 返回的迭代器一次产生路径的一段：

```
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 item 类型是 `&std::ffi::OsStr`，它是操作系统调用接受的一种字符串类型的引用切片。

如果某个类型有不止一种迭代方式，那么这个类型通常为每种遍历方式提供特定的方法，因为这时普通的 `iter` 方法将会导致歧义。例如，`&str` 字符串切片类型没有 `iter` 方法。作为替代，假设 `s` 是 `&str`，那么 `s.bytes()` 返回一个产生 `s` 的每个字节的迭代器，而 `s.chars()` 会以 UTF-8 编码解析它的内容，然后产生每一个 Unicode 字符。

15.2.2 IntoIterator 实现

当一个类型实现了 IntoIterator 之后，你可以自己调用它的 `into_iter` 方法，正如 for 循环做的一样：

```
// 你通常应该使用 HashSet，但它的迭代顺序是不确定的，  
// 因此这个例子中BTreeSet 会工作得更好。  
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);
```

大多数集合实际上都提供了好几个 IntoIterator 的实现，分别是为共享引用 (&T)、可变引用 (&mut T)、移动 (T) 提供的实现：

- 给定一个集合的共享引用 (*shared reference*)，`into_iter` 返回一个产生 item 的共享引用的迭代器。例如，在上面的代码中，`(&favorites).into_iter()` 将会返回一个 Item 类型是 `&String` 的迭代器。
- 给定一个集合的可变引用 (*mutable reference*)，`into_iter` 返回一个产生 item 的可变引用的迭代器。例如，如果 `vector` 是 `Vec<String>`，那么 `(&mut vector).into_iter()` 将返回一个 Item 类型是 `&mut String` 的迭代器。
- 当集合以值传递时，`into_iter` 返回一个获取集合所有权并返回 item 自身的迭代器；item 的所有权从集合移动到消费者，原来的集合在这个过程中被消耗。例如，上面代码中的 `favorites.into_iter()` 会返回一个产生每个字符串值的迭代器；消费者会接受每个字符串的所有权。当迭代器被 drop 时，`BTreeSet` 中剩余的所有元素也都会被 drop，并且集合会变为未初始化。

因为 for 循环会对操作数调用 `IntoIterator::into_iter`，这三种实现会导致有下面三种迭代方式：迭代集合的共享引用、迭代集合的可变引用、或者消耗集合并获取它的元素的所有权：

```
for element in &collection { ... }  
for element in &mut collection { ... }  
for element in collection { ... }
```

这三种写法会调用上面列出的 IntoIterator 实现之一。

并不是每个类型都提供了全部这三种实现。例如，`HashSet`、`BTreeSet`、`BinaryHeap`没有实现可变引用的`IntoIterator`，因为修改它们的元素可能会破坏类型的不变量：修改后的值可能会有不同的哈希值、或者和它的邻居的顺序关系会改变，因此修改元素会导致它们被放在错误的地方。其他的类型支持可变性，但只支持部分。例如，`HashMap`和`BTreeMap`产生表项的`value`的可变引用，以及`key`的共享引用，原因和上面类似。

一般的准则是迭代应该高效和可预测，因此Rust不提供开销很大或者可能展现出令人惊讶的行为的实现（例如，重新哈希被修改的`HashSet`条目并因此导致之后的迭代中可能再次遇到它们）。

切片实现了三种`IntoIterator`变体中的两个；因为它们并不拥有自己引用的元素，因此没有“以值”的实现。作为代替，`&[T]`和`&mut [T]`的`into_iter`返回一个产生共享引用和可变引用的迭代器。如果你把底层切片类型`[T]`想象成一种集合，那么它就落入了之前的模式。

你可能已经注意到前两种`IntoIterator`的变体产生共享和可变的引用，这和调用`iter`或者`iter_mut`是等价的。为什么Rust同时提供两者？

`IntoIterator`使`for`循环能正常工作，因此它显然是必要的。但当你不使用`for`循环时，使用`favorites.iter()`比`(&favorites).into_iter()`更加清晰。你可能会频繁需要以共享引用迭代，因此`iter`和`iter_mut`也很有用。

`IntoIterator`在泛型代码中也很有用：你可以使用一个约束例如`T: IntoIterator`来限制类型参数`T`必须是可以迭代的类型。或者，你可以写`T: IntoIterator<Item=U>`来进一步要求迭代会产生`U`类型的值。例如，这个函数打印出任何`item`可以用`"{:?}"`格式打印的可迭代对象：

```
use std::fmt::Debug;

fn dump<T, U>(t: T)
    where T: IntoIterator<Item=U>,
          U: Debug
{
    for u in t {
        println!("{:?}", u);
    }
}
```

你不能在这个泛型函数中使用`iter`或者`iter_mut`，因为它们不是任何trait的方法，大多数可迭代类型只是恰好有这两个方法。

15.2.3 from_fn 和 successors

一个简单而通用的产生一个值序列的方式是提供一个返回它们的闭包。

给定一个返回 `Option<T>` 的函数，`std::iter::from_fn` 返回一个迭代器，它简单地调用那个函数来产生 item。例如：

```
use rand::random; // 在Cargo.toml中添加依赖: rand = "0.7"
use std::iter::from_fn;

// 产生1000个随机数，在[0, 1]之间均匀分布。
// (这并不是你想在`rand_distr` crate中找到的分布,
// 但你可以很容易地自己实现它)
let lengths: Vec<f64> =
    from_fn(|| Some((random::f64() - random::f64()).abs()))
        .take(1000)
        .collect();
```

这里调用了 `from_fn` 来制作一个产生随机数的迭代器。因为这个迭代器总是返回 `Some`，因此这个序列永远不会终止，但我们调用了 `take(1000)` 来限制只要前 1000 个元素。然后 `collect` 从最后的迭代器构建一个 vector。这是一种高效地构建初始化的 vector 的方式。我们将在本章稍后的 [构建集合：collect 和 FromIterator](#) 中介绍为什么。

如果每一个 item 都依赖上一个，那么 `std::iter::successors` 函数可以漂亮地工作。你需要提供一个初始 item 和一个函数，这个函数要获取上一个 item 并返回下一个 item 的 `Option`。如果返回 `None`，那么迭代终止。例如，这里有另一种编写[第2章](#)中的曼德勃罗集绘制器的 `escape_time` 函数的方法：

```
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 `zero` 开始，`successors` 调用通过重复平方再加上参数 `c` 来产生一个复平面上点的序列。当绘制曼德勃罗集时，我们希望知道这个序列会一直在原点附近还是远离原点。`take(limit)` 调用设置了序列长度的限制，`enumerate` 为每一个点加上一个序号、把每个

点`z`变为元组`(i, z)`。然后我们使用`find`来查找第一个离原点足够远可以逃离的点。如果存在这样的点，`find`方法返回一个`Option::Some((i, z))`，否则返回`None`。`Option::map`的调用会把`Some((i, z))`变为`Some(i)`，但不会改变`None`：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并修改作用域中的变量。例如，这个`fibonacci`函数使用一个`move`闭包来捕获一个变量并使用它作为运行状态：

```
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]);
```

注意：`from_fn`和`successors`方法非常灵活，你可以把对迭代器的任何使用变成对其中一个的单一调用，通过复杂的闭包来获得你需要的行为。但这样做会忽略迭代器提供的表明数据流动和使用标准名称用于通用模式的能力。在你使用这两个函数之前请确保你已经熟悉了本章中的其他迭代器方法，它们通常是更好的完成工作的方式。

15.2.4 `drain`方法

很多集合类型提供一个`drain`方法来获取集合的可变引用，并返回一个迭代器把每个元素的所有权传递给消费者。然而，和`into_iter()`以值获取集合并消耗它不同，`drain`借用一个集合的可变引用，并且当迭代器被`drop`时，它会移除集合中剩余的所有元素，让集合变为空。

对于可以用范围索引的类型，例如`String`、`vector`、`VecDeque`，`drain`方法获取一个要移除的元素的范围，而不是消耗整个序列：

```
use std::iter::FromIterator;

let mut outer = "Earth".to_string();
let inner = String::from_iter(outer.drain(1..4));

assert_eq!(outer, "Eh");
assert_eq!(inner, "art");
```

如果你确实要消耗整个序列，使用整个范围`..`作为参数。

15.2.5 其他迭代器源

上面的几节基本都是关于像 vector 和 HashMap 这样的集合类型的，但标准库中还有很多其他类型支持迭代。表 15-1 总结了一些有趣的类型，但还有更多没有列出。我们将在专门介绍特定类型的章节（即第 16 章、第 17 章、第 18 章）中详细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 trait	表达式	注意
std::ops::Range	1..10	端点必须是整数才能迭代。包括起点但不包括终点。
	(1..10).step_by(2)	产生 1, 3, 5, 7, 9。
std::ops::RangeFrom	1..	无限迭代。起点必须是整数。当值到达了这种类型的极限时可能会 panic 或者溢出。
std::ops::RangeInclusive	1..=10	类似 Range，但包括终点值。
Option<T>	Some(10).iter()	类似于一个长度为 0(None) 或 1 的 vector(Some(v))。
Result<T, E>	Ok("blah").iter()	类似于 Option，产生 Ok 值。
Vec<T>, &[T]	v.windows(16)	从左到右产生重叠的、连续的给定长度的切片。
	v.chunks(16)	从左到右产生非重叠的、连续的给定长度的切片。
String, &str	v.chunks_mut(1024)	类似 chunks，不过切片是可变的。
	v.split(byte byte & 1 != 0)	产生被满足条件的元素分隔的切片。
	v.split_mut(...)	同上，但产生可变切片。
	v.rsplit(...)	类似 split，但从右向左产生切片。
	v.splitn(n, ...)	类似 split，但最多产生 n 个切片。
String, &str	s.bytes()	产生 UTF-8 字符串的字节。
	s.chars()	产生 UTF-8 字符串的 char。
	s.split_whitespace()	以空格分隔字符串，产生非空字符们的切片。

	<code>s.lines()</code>	产生字符串的每一行的切片。
	<code>s.split('/')</code>	用给定的模式分隔字符串，产生每两个匹配之间的内容的切片。模式可以是字符、字符串或者闭包。
	<code>s.matches(char::is_numeric)</code>	产生匹配给定模式的切片。
<code>std::collections::HashMap, std::collections::BTreeMap</code>	<code>map.keys(), map.values()</code>	产生 map 的 key 或 value 的共享引用。
	<code>map.values_mut()</code>	产生条目的 value 的可变引用。
<code>std::collections::HashSet, std::collections::BTreeSet</code>	<code>set1.union(set2)</code>	产生 set1 和 set2 的并集的元素的共享引用。
	<code>set1.intersection(set2)</code>	产生 set1 和 set2 的交集的元素的共享引用。
<code>std::sync::mpsc::Receiver</code>	<code>rev.iter()</code>	产生另一个线程通过相应的 Sender 发送的值。
<code>std::io::Read</code>	<code>stream.bytes()</code>	产生来自 I/O 流的字节。
	<code>stream.chars()</code>	以 UTF-8 解析流，产生 char。
<code>std::io::BufRead</code>	<code>bufstream.lines()</code>	以 UTF-8 解析流，产生 String。
	<code>bufstream.split(0)</code>	用给定的字节划分流，产生 <code>Vec<u8></code> 缓冲区。
<code>std::fs::ReadDir</code>	<code>std::fs::read_dir(path)</code>	产生目录项。
<code>std::net::TcpListener</code>	<code>listener.incoming()</code>	产生到来的网络连接。
自由函数	<code>std::iter::empty()</code>	立即返回 None。
	<code>std::iter::once(5)</code>	产生给定值然后结束。
	<code>std::iter::repeat("#9")</code>	永远产生给定值。

15.3 迭代器适配器

一旦你得到了一个迭代器，`Iterator` trait 还提供了广泛的适配器方法 (*adapter method*)，或者简称为适配器 (*adapter*)，它们消耗一个迭代器然后构建一个新的迭代器。为了展示适配器如何工作，我们将从两个最流行的适配器 `map` 和 `filter` 开始。然后我们会介绍其他的适配器，它们包括几乎所有你能想到的把一个序列的值变成另一个序列的方法：截断、跳过、组合、反向、连接、重复，等等。

15.3.1 map 和 filter

`Iterator` trait 的 `map` 适配器让你通过对每一个 item 应用一个闭包来产生新迭代器。`filter` 迭代器让你通过一个闭包决定保留哪些 item 丢弃哪些 item，以此过滤迭代器中的某些 item。

例如，假设你在迭代文本的每一行，并且想省略每一行的前导和尾部的空格。标准库的 `str::trim` 方法排除一个 `&str` 中的前导和尾部空格，返回一个新的借用 `&str`。你可以使用 `map` 适配器来对迭代器返回的每一行应用 `str::trim`:

```
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()` 调用返回一个产生每一行的迭代器。对迭代器调用 `map` 返回第二个迭代器，它会对每一行调用 `str::trim`，然后将结果作为产生的 item。最后，`collect` 把所有 item 聚集成为一个 vector。

当然，`map` 返回的迭代器，本身也可以继续适配。如果你想从结果中排除 “iguanas”，你可以像下面这样写：

```
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"]);
```

这里 `filter` 返回第三个迭代器，只有当 `map` 返回的迭代器产生的 item 调用闭包 `|s| *s != "iguanas"` 后返回 `true` 时，第三个迭代器才会产生这个 item。一个这样的迭代器适配器链就像 Unix shell 中的管道：每一个适配器都有单个功能，很容易就能看清楚值的序列是如何从左到右转换的。

这两个适配器的签名如下：

```
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` 实际上返回特定的不透明 `struct` 类型，分别是 `std::iter::Map` 和 `std::iter::Filter`。然而它们的名字提供的信息量很少，所以在本书中我们将用 `-> impl Iterator<Item=...>` 来代替，因为它们能告诉我们我们实际想要知道的信息：这个方法返回一个产生给定类型的 item 的 `Iterator`。

因为大多数适配器以值获取 `self`，所以它们需要 `Self` 是 `Sized`（大多数迭代器都是）。

`map` 迭代器会依次把所有 item 以值传递给闭包，然后把结果返回给消费者。`filter` 迭代器以共享引用把所有的 item 传递给闭包，保留选中的 item 的所有权，然后把它们传递给消费者。这就是为什么上面的例子要先解引用 `s` 再和 "iguanas" 比较：`filter` 迭代器的 item 类型是 `&str`，所以闭包参数的类型是 `&&str`。

有关迭代器适配器有两个重要的点。

首先，在一个迭代器上调用适配器并不会消耗任何 item，它只会返回一个新的迭代器，这个迭代器按需处理第一个迭代器产生的 item 来产生自己的 item。在一个适配器链中，唯一会消耗 item 的方式就是对最后的迭代器调用 `next`。

因此在我们之前的例子中，`text.lines()` 方法调用本身并不从字符串解析行，它只是返回一个迭代器，只有当需要的时候这个迭代器才会解析行。类似的，`map` 和 `filter` 只是返回需要时才会映射或过滤的新迭代器。在最后一个 `collect` 开始对 `filter` 迭代器调用 `next` 之前，将不会有任何计算发生。

当你的适配器有副作用时这一点尤其重要。例如，下面的代码什么也不打印：

```
[ "earth", "water", "air", "fire"]
    .iter().map(|elt| println!("{}", elt));
```

`iter` 调用返回一个迭代数组元素的迭代器，`map` 调用返回第二个迭代器，第二个迭代器对第一个迭代器产生的每个值调用闭包。但如果整个链中没有要求产生值的操作，那么将不会有 `next` 方法被调用。事实上，Rust 会警告你这种情况：

```
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” 并不是贬义词；它只是对任何直到需要时才进行计算的机制的一种称呼。迭代器应该做最少的必要的工作来满足 `next` 调用是 Rust 的习惯；在这个例子中，并没有 `next` 调用，因此不会有任何计算发生。

第二个重要的点是迭代器适配器是0成本抽象。因为`map`、`filter`以及它们的同伴都是泛型的，将它们用于迭代器会生成特定迭代器类型的代码。这意味着Rust有足够的信息把每一个迭代器`next`方法内联到消费者中，然后把整个操作作为一个单元翻译为机器码。因此我们上面展示的`lines/map/filter`迭代器链和你手写的代码一样高效：

```
for line in text.lines() {
    let line = line.trim();
    if line != "iguanas" {
        v.push(line);
    }
}
```

这一节剩余的部分将介绍`Iterator` trait 可用的适配器。

15.3.2 `filter_map` 和 `flat_map`

`map`适配器适用于一个输入item产生一个输出item的情况。但如果你想删除迭代中的某些item而不是处理它们，或者想将一个item替换成0个或更多的item时该怎么做呢？`filter_map`和`flat_map`适配器赋予了你这种灵活性。

`filter_map`适配器类似于`map`，除了它的闭包要么将一个item转换成一个新的item（和`map`一样），要么从迭代中丢弃这个item。因此，它有些像`filter`和`map`的结合。它的签名如下：

```
fn filter_map<B, F>(self, f: F) -> impl Iterator<Item=B>
where Self: Sized, F: FnMut(Self::Item) -> Option<B>;
```

除了闭包返回`Option`之外，而不是`B`之外，它和`map`的签名是一样的。当闭包返回`None`时，这个item会从迭代器中丢弃；当它返回`Some(b)`时，`b`就是`filter_map`迭代器产生的下一个item。

例如，假设你想扫描一个字符串中空格分隔的单词，找到其中可以被解析为数字的并处理它，然后丢弃其他单词。那你可以写：

```
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());
}
```

打印结果如下：

```
1.00
0.50
17.00
1.77
```

传给 `filter_map` 的闭包尝试对每一个空格分隔的切片调用 `f64::from_str`。这会返回一个 `Result<f64, ParseFloatError>`，它的 `.ok()` 返回一个 `Option<f64>`：解析错误时是 `None`，解析成功时是 `Some(v)`。`filter_map` 迭代器丢弃所有的 `None` 值，然后对每一个 `Some(v)` 产生值 `v`。

但为什么要将 `map` 和 `filter` 融合成这样的单个操作，而不是直接使用两个适配器？`filter_map` 适配器适用于刚刚展示过的这种情况，即只有实际尝试处理过 `item` 才知道应不应该包含这个 `item` 的情况。你可以只用 `filter` 和 `map` 做到同样的事情，但这样会很笨拙：

```
text.split_whitespace()
    .map(|w| f64::from_str(w))
    .filter(|r| r.is_ok())
    .map(|r| r.unwrap())
```

你可以认为 `flat_map` 适配器和 `map`、`filter_map` 是同一类的，区别在于现在闭包不是只能返回一个 `item`(`map`) 或者 0 或 1 个 `item`(`filter_map`)，而是可以返回任意数量的 `item`。`flat_map` 迭代器产生闭包返回的序列的串联。

`flat_map` 的签名如下：

```
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对象，但任何类型的可迭代对象都可以。¹

例如，假设我们有一个把国家映射到主要城市的表。给定一个国家的列表，那我们怎么遍历它们的主要城市？

```
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", "Brasilia"]);
major_cities.insert("Kenya", vec!["Nairobi", "Mombasa"]);
```

¹事实上，因为 `Option` 也是一个可迭代对象，行为就像一个有 0 个或者 1 个 item 的序列。所以假设 `closure` 返回一个 `Option<T>`，那么 `iterator.filter_map(closure)` 等价于 `iterator.flat_map(closure)`。

```
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!("{}: {}", country, city);  
}
```

这会打印出下列内容：

```
Tokyo  
Kyoto  
São Paulo  
Brasilia  
Nairobi  
Mombasa
```

这段代码的意思是，对于每一个国家，我们都获取它的城市的 vector，然后将所有 vector 连接成单个序列，然后打印出来。

但记住迭代是惰性的：只有当 `for` 循环调用了 `flat_map` 迭代器的 `next` 方法时才会开始计算。完全连接的序列从来不会在内存中构造。实际上，这里只有一个小的状态机，对于每一个城市迭代器，一次打印一个 item，直到耗尽，然后为下一个国家产生一个新的城市迭代器。效果就类似于嵌套的循环，但被打包用作迭代器。

15.3.3 flatten

`flatten` 适配器把迭代器的 item 连接起来，假设每一个 item 都是可迭代对象：

```
use std::collections::BTreeMap;  
  
// 把城市映射到公园的表：每一个 value 都是一个 vector。  
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", "Dargon Park"]);  
  
// 构建一个所有公园的 vector。`values` 返回一个产生 vector 的迭代器，  
// 然后 `flatten` 按顺序产生每一个 vector 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"]);
```

“flatten”这个名字来自于想象把一个两层的结构压扁成一层的结构：BTreeMap和它的Vec的元素被压成一个产生所有元素的迭代器。

`flatten`的签名如下：

```
fn flatten(self) -> impl Iterator<Item=Self::Item::Item>
where Self::Item: IntoIterator;
```

换句话说，迭代器的item自身必须实现了`IntoIterator`，这样它才是一个高效的序列的序列。`flatten`方法返回一个这些序列连接之后的迭代器。当然，这都是惰性完成的，只有当我们迭代完了一个序列才会从`self`产生一个新的item。

`flatten`方法还有一些令人惊讶的用法。如果你有一个`Vec<Option<...>>`并且你想只迭代其中的`Some`值，那么`flatten`可以漂亮地工作：

```
assert_eq!(vec![None, Some("day"), None, Some("one")]
    .into_iter()
    .flatten()
    .collect::<Vec<_>>(),
    vec!["day", "one"]);
```

这种方式可以工作是因为`Option`自身实现了`IntoIterator`，代表一个有0或1个元素的序列。`None`元素对迭代过程没有贡献，而每一个`Some`元素贡献一个值。类似的，你可以使用`flatten`来迭代`Option<Vec<...>>`: `None`和空vector的行为一样。

`Result`也实现了`IntoIterator`，`Err`时代表一个空的序列，因此对一个产生`Result`值的迭代器调用`flatten`可以高效地排除所有`Err`，产生一个解包之后的成功值的序列。我们不推荐在代码中忽略错误，但当用户知道自己在做什么时这是一个巧妙的技巧。

当你需要`flatten`时你可能会发现你真正需要的是`flat_map`。例如，标准库的`str::to_uppercase`方法把一个字符串转换成大写，工作方式类似于下面的代码：

```
fn to_uppercase(&self) -> String {
    self.chars()
        .map(char::to_uppercase)
        .flatten() // 有更好的方式
        .collect()
}
```

这里必须使用`flatten`的原因是`char::to_uppercase()`并不是返回单个字符，而是返回一个可能产生一个或更多字符的迭代器。`map`调用会返回一个产生字符迭代器的迭代器，`flatten`将它们拼接在一起，因此我们最后可以调用`collect`把它们转换为一个`String`。

但这种`map`和`flatten`的组合使用如此普遍，以至于`Iterator`提供了`flat_map`适配器来处理这种情况。（事实上，`flat_map`比`flatten`更先加入标准库。）因此上面的代码可以写成：

```
fn to_uppercase(&self) -> String {
    self.chars()
        .flat_map(char::to_uppercase)
        .collect()
}
```

15.3.4 `take` 和 `take_while`

Iterator trait 的 `take` 和 `take_while` 适配器让你可以在迭代了一定的次数或者当一个闭包决定截断时停止迭代。它们的签名如下：

```
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;
```

这两个方法都获取一个迭代器的所有权，返回一个新的迭代器，新的迭代器从第一个 item 开始迭代，可能会提前终止序列。在产生最多 n 个 item 之后 `take` 迭代器会返回 `None`。`take_while` 迭代器对每个 item 引用 `predicate`，当有一个 item 使 `predicate` 返回 `false` 时返回 `None`，之后对 `next` 的调用也都会返回 `None`。

例如，给定一个电子邮件的消息，其中消息头和消息主体用一个空行分隔，那么你就可以使用 `take_while` 来只迭代器消息头：

```
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);
}
```

回顾字符串字面量，当字符串行以反斜杠结尾时，Rust 并不会把下一行的缩进包含进字符串里，因此这个字符串中的任何一行都没有前导空格。这意味着 `message` 的第三行是空行。`take_while` 适配器第一次看到空行时就会停止迭代，因此这段代码只会打印出两行：

```
To: jimb
From: superego <editor@oreilly.com>
```

15.3.5 skip 和 skip_while

`Iterator` trait 的 `skip` 和 `skip_while` 方法是 `take` 和 `take_while` 的补充：它们丢弃迭代起始的一定数量的 item，或者直到一个闭包找到一个可接受的 item 时，按原样传递这个和剩余的 item。它们的签名如下：

```
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` 适配器的一个常见用法是在迭代命令行参数时跳过命令的名字。在第 2 章中，我们的最大公约数计算器就用了下面的代码来迭代它的命令行参数：

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

`std::env::args` 函数返回一个迭代器，以 `String` 类型产生程序的参数，其中第一个 item 就是程序自身的名字。我们并不想在这个循环中处理它。对这个迭代器调用 `skip(1)` 返回一个新的迭代器，新的迭代器会丢弃程序名，然后产生剩余的参数。

`skip_while` 适配器使用闭包来决定丢弃序列开头的多少 item。你可以像这样迭代上一节的消息的主体行：

```
for body in message.lines()
    .skip_while(|l| !l.is_empty())
    .skip(1)
    println!("{}", body);
```

这里使用了 `skip_while` 来跳过非空的行，但这个迭代器会产生那个空行——也就是使闭包返回 `false` 的那个 item。因此我们使用了 `skip` 方法来丢弃那个空行，这样最后的迭代器的第一个 item 就是消息主体的第一行。结合上一节中的 `message`，这段代码会打印出：

```
Did you get any writing done today?
When will you stop wasting time plotting fractals?
```

15.3.6 peekable

`peekable` 迭代器让你可以窥视下一个将被产生的 item，而不实际消耗它。你可以通过调用 `Iterator` trait 的 `peekable` 方法把一个迭代器转换成一个 `peekable` 迭代器：

```
fn peekable(self) -> std::iter::Peekable<Self>
    where Self: Sized;
```

这里，`Peekable<Self>`是一个实现了`Iterator<Item=Self::Item>`的`struct`，其中`Self`是底层迭代器的类型。

`Peekable`迭代器有一个额外的`peek`方法返回一个`Option<&Item>`：如果底层迭代器已经迭代完就返回`None`，否则返回`Some(r)`，其中`r`是下一个item的共享引用。(注意如果底层迭代器的item的类型已经是一个引用，那么最后返回的将是一个引用的引用。)

调用`peek`会尝试获取底层迭代器的下一个item，如果确实还有item，就缓存它直到下一次调用`next`。`Peekable`的其他`Iterator`的方法都知道这个缓存：例如，一个`peekable`迭代器`iter`的调用`iter.last()`知道在耗尽了底层的迭代器之后检查缓存。

当你必须要遍历之后才知道到底要消耗多少item时，`peekable`迭代器就很重要了。例如，如果你正在一个字符流解析数字，只有当你看到了数字之后的第一个非数字字符时你才能知道数字在哪里结束：

```
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);
// `parse_number`不会消耗逗号！因此下面不会出错。
assert_eq!(chars.next(), Some(','));
assert_eq!(parse_number(&mut chars), 1766319049);
assert_eq!(chars.next(), None);
```

`parse_number`函数使用`peek`方法来检查下一个字符，并且只有当它是数字时才消耗它。

如果它不是数字或者迭代器被耗尽（也就是，如果 `peek` 返回 `None`），我们将会返回已经解析的数字，将下一个字符留在迭代器里，之后再消耗。

15.3.7 fuse

一旦一个 `Iterator` 返回 `None`，trait 并没有指定如果你继续调用 `next` 方法时它的行为。大多数迭代器都是再次返回 `None`，但并不是所有。如果你的代码依赖这种行为，那你有时可能会遇到奇怪的行为。

`fuse` 适配器接受任何迭代器，并产生一个保证第一次返回 `None` 之后一直返回 `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; // D'oh!
            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` 适配器最有用的场景可能是需要使用不确定来源的迭代器的泛型代码。与其希望每一个要处理的迭代器都是良定义的，不如使用 `fuse` 来确保这种行为。

15.3.8 可逆迭代器和 rev

一些迭代器可以从序列的任意一端产生 item。你可以通过使用 rev 适配器来反转这样的迭代器。例如，一个 vector 的迭代器可以简单的从尾部到头部产生 item。这样的迭代器可以实现 std::iterator::DoubleEndedIterator trait，它是 Iterator 的扩展：

```
trait DoubleEndedIterator: Iterator {
    fn next_back(&mut self) -> Option<Self::Item>;
}
```

你可以将这样的双端迭代器看做有两个标记分别指向正向和反向迭代的位置。每次从其中一个产生 item，都会让它向另一个的方向移动；当它们交汇的时候，迭代就结束了：

```
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_back(), None);
```

切片的迭代器可以很容易的实现这种行为：它利用两个指针分别指向还未产生的元素范围的起点和终点；next 和 next_back 简单地从其中一个产生一个 item。有序集合的迭代器例如 BTreeSet 和 BTreeMap 也是双端迭代器：它们的 next_back 方法以最大元素或条目优先的顺序返回 item。一般来说，标准库在可行时都会提供双端迭代器。

但并不是所有的迭代器都可以简单地做到这一点：一个产生其他线程发给通道的 Receiver 的值的迭代器没有办法预测最后一个到达的值会是什么。一般来说，你应该检查标准库的文档来看看哪些迭代器实现了 DoubleEndedIterator，哪些没有。

如果一个迭代器是双端的，你可以使用 rev 适配器来反转它：

```
fn rev(self) -> impl Iterator<Item=Self>
where Self: Sized + DoubleEndedIterator;
```

返回的迭代器也是一个双端迭代器：它的 next 和 next_back 方法被简单地交换了：

```
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);
```

大多数迭代器适配器如果被用于可逆迭代器，会返回另一个可逆迭代器。例如，`map` 和 `filter` 会保留可逆性。

15.3.9 inspect

`inspect` 适配器在调试迭代器适配器的流水线时很有用，但通常不用在生产代码中。它简单地对每一个 item 的共享引用应用一个闭包然后传递它们。闭包不能影响到 item，但可以做一些类似于打印或进行断言的操作。

这个例子展示了一个将字符串转换为大写会改变长度的例子：

```
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”，这就是为什么 `char::to_uppercase` 会返回一个产生字符的迭代器，而不是单个字符。上面的代码中使用 `flat_map` 来把所有 `to_uppercase` 序列连接成单个 `String`，它会打印出下列内容：

```
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` 迭代器适配器将一个迭代器附加到另一个之后。更确切地说，`i1.chain(i2)` 返回的迭代器会首先产生 `i1` 的 item，当 `i1` 耗尽后，再继续产生 `i2` 的 item。

`chain` 适配器的签名如下：

```
fn chain<U>(self, other: U) -> impl Iterator<Item=Self::Item>
where Self: Sized, U: IntoIterator<Item=Self::Item>;
```

换句话说，你可以将任何产生相同类型 item 的迭代器连接起来。例如：

```
let v: Vec<i32> = (1..4).chain(vec![20, 30, 40]).collect();
assert_eq!(v, [1, 2, 3, 20, 30, 40]);
```

如果两个底层迭代器都是可逆的，那么 chain 迭代器也是可逆的：

```
let v: Vec<i32> = (1..4).chain(vec![20, 30, 40]).rev().collect();
assert_eq!(v, [40, 30, 20, 3, 2, 1]);
```

一个 chain 迭代器会追踪是否两个底层迭代器返回过 None，并正确地调用 next 或者 next_back。

15.3.11 enumerate

Iterator trait 的 enumerate 适配器把一个索引附加到序列中，它以一个产生 A, B, C, ... 的迭代器为参数，返回一个产生 (0, A), (1, B), (2, C), ... 的迭代器。第一眼可能会觉得它用处不大，但实际上它非常常用。

消费者可以使用索引来区分不同的 item 并建立起处理每个 item 的上下文。例如，第 2 章中的曼德勃罗集绘制器将图像分割成了 8 个水平的条带，并把每个条带分到不同的线程中进行处理。那段代码使用了 enumerate 来告诉每个线程它的条带是图像中的哪一部分。

它以一个矩形像素的缓冲区开始：

```
let mut pixels = vec![0; columns * rows];
```

然后，它使用了 chunks_mut 把图像分割成水平的条带，每个线程一个条带：

```
let threads = 8;
let band_rows = rows / threads + 1;
...
let bands: Vec<&mut [u8]> = pixels.chunks_mut(band_rows * columns).collect();
```

然后它迭代条带，为每个条带开启一个新线程：

```
for (i, band) in bands.into_iter().enumerate() {
    let top = band_rows * i;
    // 启动一个线程渲染 `top..top + band_rows` 范围内的行
    ...
}
```

每一次迭代会得到一个(`i, band`)，其中`band`是这个线程应该绘制的像素缓冲区的`&mut[u8]`切片，`i`是条带在整个图片中的索引，由`enumerate`适配器产生。给定图像的长宽和条带的大小，一个线程就获取了足够的信息来判断它要处理的是图像的那一部分，并且因此能正确绘制`band`。

你可以将`enumerate`产生的(`index, item`)类比迭代`HashMap`或其他关联集合时得到的(`key, value`)对。如果你在迭代一个切片或者`vector`，那么`index`就是`item`出现的“key”。

15.3.12 zip

`zip`适配器将两个迭代器组合成单个迭代器，它一次产生一个pair，pair里分别是两个迭代器产生的item，就像是一条拉链一样把两边汇聚成单条缝隙。当两个底层迭代器中有任何一个结束时，`zip`迭代器也会结束。

例如，你可以通过把无限范围`0..`和另一迭代器`zip`在一起达到`enumerate`适配的效果：

```
let v: Vec<_> = (0..).zip("ABCD".chars()).collect();
assert_eq!(v, vec![(0, 'A'), (1, 'B'), (2, 'C'), (3, 'D')]);
```

从这个意义上讲，你可以将`zip`看做`enumerate`的泛化版：`enumerate`把索引附加序列中，`zip`把任意另一个迭代器的item附加到序列中。我们之前建议过`enumerate`可以帮助提供处理item的上下文；`zip`是一种更灵活的实现同样功能的方式。

`zip`的参数不需要是一个和调用者自己同类型的迭代器；它可以是任意的可迭代对象：

```
use std::iter::repeat;

let endings = vec!["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

在本节中，我们已经见过很多次把适配器附加到迭代器上。但一旦这么做了之后，还可以把适配器去除掉吗？通常是不行的：适配器会获取底层迭代器的所有权，而且不提供方法返回底层迭代器。

迭代器的`by_ref`方法借用一个迭代器的可变引用，然后你可以对这个引用应用适配器。当你消耗完这些适配器的item之后，你会`drop`它们，然后借用结束，你可以重新访问原本的迭代器。

例如，本章前面我们展示过怎么使用 `take_while` 和 `skip_while` 来处理邮件消息的消息头或者主体。但如果你想用同一个底层迭代器同时处理两者呢？使用 `by_ref`，我们可以使用 `take_while` 来处理消息头，然后等它结束之后重新获得底层迭代器的访问权，而 `take_while` 结束之后底层迭代器恰好留在消息主体的位置上：

```
let message = "To: jimbo\r\n\
                From: id\r\n\
                \r\n\
                Oooooh, donuts!!\r\n";\n\nlet mut lines = message.lines();\n\nprintln!("Headers:");\nfor header in lines.by_ref().take_while(|l| !l.is_empty()) {\n    println!("{}", header);\n}\n\nprintln!("\nBody:");\nfor body in lines {\n    println!("{}", body);\n}
```

`line.by_ref()` 调用借用了迭代器的可变引用，然后 `take_while` 获取的是这个可变引用的所有权。当第一个 `for` 循环结束时，这个引用就离开了作用域，这意味着借用结束了，因此你可以在第二个 `for` 循环中再次使用 `lines`。这会打印出下面的内容：

```
Headers:\nTo: jimbo\nFrom: id\n\nBody:\nOooooh, donuts!
```

`by_ref` 适配器的定义很简单，它返回一个迭代器的可变引用。然后，标准库包含这个奇怪的实现：

```
impl<'a, I: Iterator + ?Sized> Iterator for &'a mut I {\n    type Item = I::Item;\n    fn next(&mut self) -> Option<I::Item> {\n        (**self).next()\n    }\n    fn size_hint(&self) -> (usize, Option<usize>) {
```

```

        (**self).size_hint()
    }
}

```

换句话说，如果 I 是迭代器，那么 &mut I 也是一个迭代器，它的 next 和 size_hint 方法指向它引用的对象的相应方法。当你在一个迭代器的可变引用上调用适配器时，适配器会获取引用的所有权，而不是迭代器本身的所有权。它只是一个当适配器离开作用域时就会结束的借用。

15.3.14 cloned 和 copied

cloned 适配器获取一个产生引用的迭代器，然后返回一个产生这些引用的拷贝操作返回的值的迭代器。就类似于 `iter.map(|item| item.clone())`。自然地，被引用的类型必须实现 Clone。例如：

```

let a = ['1', '2', '3', '='];
assert_eq!(a.iter().next(), Some(&'a'));
assert_eq!(a.iter().cloned().next(), Some('1'));

```

copied 适配器也是相同的思路，但更加严格：被引用的类型必须实现 Copy。

`iter.copied()` 调用类似于 `iter.map(|r| *r)`。因为每个实现了 Copy 的类型都实现了 Clone，所以 cloned 通常更一般化，但根据 item 的类型不同，clone 调用可以做任意次的分配和拷贝操作。如果你假设你的 item 的类型足够简单所以永远不会发生这种情况，那么最好的方法还是使用 copied 来让类型检查器检查你的假设。

15.3.15 cycle

cycle 适配器返回一个无限重复底层迭代器产生的序列的迭代器。底层迭代器必须实现了 `std::clone::Clone`，这样 cycle 才可以保存它的初始状态，然后在每一次循环开始时重用它。

例如：

```

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，完全没有任何开销地使用它：

```
use std::iter::{once, repeat};

let fizzes = repeat("").take(2).chain(once("fize")).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);
}
```

这样就完成了一个小孩子的单词游戏，现在有时也会被用作程序员的面试题目。在这个游戏中，玩家要轮流计数，把任何能被3整除的数替换成单词 `fizz`，任何能被5整除的数替换成 `fizzbuzz`。能被3和5同时整除的数替换为 `fizzbuzz`。

15.4 消耗迭代器

到目前为止，我们已经介绍了创建迭代器和用适配器将它们包装成新的迭代器；接下来我们将展示如何消耗它们来结束这个过程。

当然，你可以使用 `for` 循环来消耗一个迭代器，或者显式地调用 `next`，但有很多常见的任务没必要一次又一次地写出来。`Iterator` trait 提供了许多的方法来处理很多常用的情况。

15.4.1 简单的累计：`count`, `sum`, `product`

`count` 方法不断从一个迭代器获取 item 直到它返回 `None`，然后告诉你它到底获得了多少个 item。这里有一个简短的程序来统计标准输入的行数：

```
use std::io::prelude::*;

fn main() {
    let stdin = std::io::stdin();
    println!("{}", stdin.lock().lines().count());
}
```

`sum` 和 `product` 方法计算迭代器的 item 的和或者积，item 必须是整数或者浮点数：

```
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);
```

(你可以通过实现 `std::iter::Sum` 和 `std::iter::Product` trait 来扩展 `sum` 和 `product` 以支持其他类型，但本书中不会介绍。)

15.4.2 `max`, `min`

`Iterator` 的 `min` 和 `max` 方法分别返回迭代器产生的 item 中最小的和最大的。item 的类型必须实现了 `std::cmp::Ord`，这样才可以互相比较。例如：

```
assert_eq!([-2, 0, 1, 0, -2, -5].iter().max(), Some(&1));
assert_eq!([-2, 0, 1, 0, -2, -5].iter().min(), Some(&-5));
```

这些方法返回 `Option<Self::Item>`，这样如果迭代器没产生值的时候可以返回 `None`。

正如相等性比较中解释的一样，Rust 的浮点数类型 `f32` 和 `f64` 只实现了 `std::cmp::PartialOrd`，没有实现 `std::cmp::Ord`，因此你不能使用 `min` 和 `max` 方法来计算浮点数序列中的最小值或最大值。这不是 Rust 设计中的优势，但它是故意的：因为不清楚这样的函数如果遇到了 IEEE NaN 值时该怎么处理。简单地忽略它们可能会导致代码中更严重的问题。

如果你知道如何处理 NaN 值，你可以使用 `max_by` 和 `min_by` 迭代器方法作为代替，它允许你提供自己的比较函数。

15.4.3 `max_by`, `min_by`

`max_by` 和 `min_by` 方法分别返回迭代器产生的最大值和最小值，通过一个你提供的比较函数来判断大小：

```
use std::cmp::Ordering;

// 比较两个 f64 值，如果有 NaN 就 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)); // panic

```

`max_by` 和 `min_by` 方法以引用的方式把 item 传递给比较函数，这样不管迭代器产生什么类型的值它们都可以高效工作。因此这里 `cmp` 要以引用获取参数，即使我们已经使用 `copied` 获取过一个产生 `f64` item 的迭代器。

15.4.4 `max_by_key`, `min_by_key`

`Iterator` 的 `max_by_key` 和 `min_by_key` 方法通过比较对 item 调用闭包返回的值来分别返回最大值和最小值。闭包可以选择 item 的某些字段或者对 item 进行计算。因为你通常对最大和最小值相关联的数据感兴趣，而不是对极值本身感兴趣，因此这些方法通常比 `max` 和 `min` 更有用。它们的签名如下：

```

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;

```

这两个方法接受一个闭包 `f` 作为参数，`f` 需要接受一个 `item` 并返回一个有序的类型 `B`。然后这两个方法对每个 `item` 都调用 `f`，根据返回的 `B` 类型的值来比较大小，最后分别返回最大和最小的 `item`，或者如果没有 `item` 产生时返回 `None`。

例如，如果你需要扫描一个城市的哈希表来查找人口最多和最少的城市，你可以写：

```

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` 会应用到迭代器产生的每个 item，然后返回用于比较的值——在这个例子中，就是城市的人口。最后返回的值是整个 item，而不仅仅是闭包返回的值。（当然，如果你经常进行这样的查询，你可能会需要一种更高效的存储方式来查找，而不是在这样的表上进行线性查找。）

15.4.5 比较 item 序列

你可以使用 `<` 和 `==` 运算符来比较字符串、vector、切片，假设它们的元素可以被比较。尽管迭代器不支持 Rust 的比较运算符，但它们确实提供了方法例如 `eq` 和 `lt` 来做相同的工作，这些方法每次从两个迭代器各取一个 item 进行比较，直到可以比较出结果。例如：

```
let packed = "Helen of Troy";
let spaced = "Helen    of    Troy";
let obscure = "Helen of Sandusky"; // 好人，只是没名气

assert!(packed != spaced);
assert!(packed.split_whitespace().eq(spaced.split_whitespace()));

// 比较为真，因为' ' < 'o'
assert!(spaced < obscure);

// 比较为真，因为'Troy' > 'Sandusky'
assert!(spaced.split_whitespace().gt(obscure.split_whitespace()));
```

`split_whitespace` 的调用返回一个迭代字符串中空格分隔的单词的迭代器。对这些迭代器使用 `eq` 和 `gt` 方法会进行逐单词的比较，而不是逐字符比较。这些比较都是可行的，因为 `&str` 实现了 `PartialOrd` 和 `PartialEq`。

迭代器提供 `eq` 和 `ne` 方法进行相等性比较，以及 `lt`, `le`, `gt`, `ge` 方法用于顺序性比较。`cmp` 和 `partial_cmp` 方法的行为类似于 `Ord` 和 `PartialOrd` trait 的相应方法。

15.4.6 any 和 all

`any` 和 `all` 方法对迭代器产生的每一个 item 都应用一个闭包，分别当任意 item 使闭包返回 `true` 和所有 item 都使闭包返回 `true` 时返回 `true`:

```
let id = "Iterator";

assert!(id.chars().any(char::is_uppercase));
assert!(!id.chars().all(char::is_uppercase));
```

只有当需要的时候这两个方法才会继续消耗 item。例如，如果已经有一个 item 让闭包返回 true，那么 any 会立刻返回 true，不会再继续消耗剩余的 item。

15.4.7 position, rposition, ExactSizeIterator

position 方法应用到迭代器的每一个 item 上，返回第一个使闭包返回 true 的 item 的索引。更确切地说，它返回一个索引的 Option：如果没有 item 使闭包返回 true，position 会返回 None。只要闭包有一次返回 true 它就会立刻停止。例如：

```
let text = "Xerxes";
assert_eq!(text.chars().position(|c| c == 'e'), Some(1));
assert_eq!(text.chars().position(|c| c == 'z'), None);
```

rposition 方法功能相同，不过它是从最后一个元素往前开始搜索。例如：

```
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 方法要求一个可逆迭代器这样它才可以从最后往前遍历。它还要求一个 exact-size 迭代器这样它才可以返回一个和 position 一样的索引，即从前往后数情况下的索引。exact-size 迭代器是一个实现了 std::iter::ExactSizeIterator trait 的迭代器：

```
trait ExactSizeIterator: Iterator {
    fn len(&self) -> usize { ... }
    fn is_empty(&self) -> bool { ... }
}
```

len 方法返回剩余 item 的数量，当迭代结束时 is_empty 方法返回 true。

当然，不是每一个迭代器都知道它还会产生多少个 item。例如，之前使用的 str::chars 迭代器就做不到（因为 UTF-8 是可变宽度的编码），因此你不能对字符串使用 rposition。但一个迭代字节数组的迭代器当然知道数组的长度，因此它可以实现 ExactSizeIterator。

15.4.8 fold 和 rfold

在进行某种需要累计所有 item 的计算时 fold 方法是非常通用的工具。给定一个初始值（我们称之为累加器 (accumulator)）和一个闭包，fold 会重复对当前的累加器和下一个 item 应

用这个闭包。闭包返回的值被用作新的累加器，然后和下一个 item 一起传递给闭包。最终累加器的值就是 fold 自身的返回值。如果序列为空，fold 直接返回初始的累加器。

很多其他消耗迭代器的方法都可以用 fold 来实现：

```
let a = [5, 6, 7, 8, 9, 10];

assert_eq!(a.iter().fold(0, |n, _| n+1), 6);           // count
assert_eq!(a.iter().fold(0, |n, i| n+i), 45);          // sum
assert_eq!(a.iter().fold(1, |n, i| n*i), 151200);       // product

// max
assert_eq!(a.iter().cloned().fold(i32::min_value(), std::cmp::max), 10);
```

fold 方法的签名如下：

```
fn fold<A, F>(self, init: A, f: F) -> A
where Self: Sized, F: FnMut(A, Self::Item) -> A;
```

其中，A 是累加器的类型。init 参数就是一个 A 类型的值，A 也是闭包的第一个参数和返回值的类型，也是 fold 本身的返回值的类型。

注意累加器值被移动进和移出闭包，因此你可以用非 Copy 的累加器类型调用 fold：

```
let a = ["Pack", "my", "box", "with",
        "five", "dozen", "liquor", "jugs"];

// See also: 切片的`join`方法，最后不会有多余的空格。
let pangram = a.iter().fold(String::new(), |s, w| s + w + " ");
assert_eq!(pangram, "Pack my box with five dozen liquor jugs");
```

rfold 方法和 fold 方法相同，除了它需要双端迭代器，并且从最后开始往前处理 item：

```
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 和 try_rfold

try_fold 方法和 fold 基本一样，除了它的迭代过程可以提前退出，不需要消耗迭代器里的所有值。传递给 try_fold 的闭包需要返回一个 Result：如果它是 Err(e)，try_fold 会立刻返回 Err(e) 作为返回值。否则，它会继续处理成功的值。闭包也可以返回一个 Option：返回 None 会提前退出，最后的返回值是个 Option。

这里有一个求来自标准输入的数字之和的程序：

```

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(())
}

```

输入流的`lines`迭代器产生`Result<String, std::io::Error>`类型的`item`, 把`String`解析为整数也可能会失败。这里使用`try_fold`需要闭包返回`Result<u64, ...>`, 所以我们可以使用`? 运算符`来把闭包里的错误传播到`main`函数中。

因为`try_fold`如此灵活, 它常被用于实现`Iterator`的很多其他消耗方法。例如, 这里有一个`all`的实现:

```

fn all<P>(&mut self, mut predicate: P) -> bool
    where P: FnMut(Self::Item) -> bool,
          Self: Sized
{
    self.try_fold((), |_| item {
        if predicate(item) { Some(()) } else { None }
    }).is_some()
}

```

注意这个实现不能使用普通的`fold`: `all`保证只要有一个`item`使`predicate`返回`false`就会停止消耗`item`, 但`fold`总是消耗整个迭代器。

如果你正在实现自己的迭代器类型, 那么思考一下你的迭代器是否能比`Iterator trait`的默认实现更高效地实现`try_fold`方法是值得的。如果你可以加速`try_fold`, 所有其他基于它的方法都会受益。

`try_rfold`方法, 就如它的名字所示, 和`try_fold`一样, 除了它从最后开始往前处理`item`, 并因此需要一个双端迭代器。

15.4.10 nth, nth_back

`nth`方法接受一个索引 `n`, 跳过迭代器中接下来 `n` 个 item, 然后返回下一个 item, 或者如果迭代器已经在这个过程中到达终点就返回 `None`。调用 `.nth(0)` 等价于调用 `.next()`。

它不会像很多适配器一样获取迭代器的所有权, 所以你可以调用它很多次:

```
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:

```
fn nth(&mut self, n: usize) -> Option<Self::Item>
where Self: Sized;
```

`nth_back`方法类似, 除了它是从双端迭代器的尾部往前移动。调用 `.nth_back(0)` 等价于调用 `.next_back()`: 它返回最后一个 item, 或者如果迭代器已经为空时返回 `None`。

15.4.11 last

`last`方法返回迭代器产生的最后一个 item, 或者如果迭代器为空时返回 `None`。它的签名如下:

```
fn last(self) -> Option<Self::Item>;
```

例如:

```
let squares = (0..10).map(|i| i*i);
assert_eq!(squares.last(), Some(81));
```

它会消耗迭代器的所有 item, 即使迭代器可逆。如果你有一个可逆迭代器并且不想让它消耗所有 item, 那你应该调用 `iter.next_back()`。

15.4.12 find, rfind, find_map

`find`方法从迭代器查找 item, 返回第一个使给定闭包返回 `true` 的 item, 或者如果没有任何 item 满足条件就返回 `None`。它的签名是:

```
fn find<P>(&mut self, predicate: P) -> Option<Self::Item>
where Self: Sized,
P: FnMut(&Self::Item) -> bool;
```

`rfind`方法类似，不过它需要双端迭代器并且从最后往前搜索，返回最后一个使闭包返回 `true` 的 item。

例如，使用`max_by`, `min_by`中的城市和人口的表，你可以写：

```
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)));
```

表中没有人口超过一千万的城市，但有超过50万的城市。

有时你的闭包并不是一个简单的对 item 进行布尔判断的判别式：它可能会更加复杂，并且返回一个它自己产生的有趣的值。在这种情况下，`find_map`正是你需要的。它的签名是：

```
fn find_map<B, F>(&mut self, f: F) -> Option<B> where
    F: FnMut(Self::Item) -> Option<B>;
```

这类似于 `find`，区别在于它接受的闭包不是返回 `bool`，而是返回一个 `Option`。`find_map` 返回第一个是 `Some` 的 `Option`。

例如，如果我们有一个每个城市的公园的数据库，我们可能想看看其中是否有火山，并且如果说有的话返回公园的名字：

```
let big_city_with_volcano_park = populations.iter()
    .find_map(|(&city, _)| {
        if let Some(park) = find_volcano_park(city, &parks) {
            // find_map 返回这个值,
            // 因此我们的调用者会知道我们找到了 *哪个* 公园
            return Some((city, park.name));
        }

        // 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` 和 `FromIterator`

在这本书中，我们曾多次看见使用 `collect` 方法来构建 `vector`。例如，在第2章中，我们调用了 `std::env::args()` 来获取一个程序命令行参数的迭代器，然后调用了迭代器的 `collect` 方法来把它们收集到一个 `vector` 中：

```
let args: Vec<String> = std::env::args().collect();
```

但 `collect` 并不是只能用于 `vector`: 事实上, 它可以用于构建 Rust 标准库中的任意集合, 只要迭代器产生合适的 item 类型:

```
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();

// 收集 map 需要 (key, value) 对, 因此在这个例子中,
// 使用 zip 将字符串序列和整数序列绑定。
let args: HashMap<String, usize> = std::env::args().zip(0..).collect();
let args: BTreeMap<String, usize> = std::env::args().zip(0..).collect();

// 等等
```

当然, `collect` 方法本身并不知道如何构建这些类型。当一些集合类型例如 `Vec` 或者 `HashMap` 知道如何从一个迭代器构建自身时, 它会实现 `std::iter::FromIterator` trait, `collect` 只是它的便捷用法:

```
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` 可以从一个产生 `A` 类型值的迭代器构建一个自身类型的集合。

在最简单的情况下, 这个 trait 的实现可以只简单地构建一个空集合, 然后把迭代器产生的 item 一个个添加进去。例如, `std::collections::LinkedList` 的 `FromIterator` 实现就是按这种方式工作。

然而, 一些方式可以有更好的实现。例如, 从一个迭代器 `iter` 构造一个 `vector` 可以非常简单地实现:

```
let mut vec = Vec::new();
for item in iter {
    vec.push(item);
}
vec
```

但这并不是个好主意: 随着 `vector` 的增长: 它可能需要扩展缓冲区, 需要调用堆分配器, 还要拷贝现有元素。`vector` 确实通过算法措施来保证这个开销很低, 但如果有更简单的方式直接在最开始就分配一个正确大小的缓冲区, 那么就完全没必要再次调整大小。

这就是迭代器的 `Iterator` trait 的 `size_hint` 方法出现的原因：

```
trait Iterator {
    ...
    fn size_hint(&self) -> (usize, Option<usize>) {
        (0, None)
    }
}
```

这个方法返回迭代器可以产生的 item 数量的下界和一个可选的上界。默认的实现返回 0 作为下界，并拒绝指定上界，实际上就是在说“我不知道”，但很多迭代器可以做得更好。例如 `Range` 的迭代器就精确地知道它将产生多少个 item，`Vec` 和 `HashMap` 的迭代器也一样。这样的迭代器会提供自己的 `size_hint` 的特化定义。

这两个界限正是 `Vec` 的 `FromIterator` 的实现所需要知道的信息，这样它就能从一开始分配正确大小的缓冲区。插入新元素仍然会检查缓冲区是否足够大，因此即使提示不正确，也只会影响性能，而不会影响安全性。其他类型也可以选择类似的策略：例如，`HashSet` 和 `HashMap` 也实现了 `Iterator::size_hint` 来为哈希表选择合适的初始大小。

有关于类型推导的一个提示：在本节的开始处，同样的调用 `std::env::args().collect()` 根据上下文会产生四种不同类型的集合，这看起来可能有些奇怪。`collect` 的返回值类型是它的类型参数，因此前两个调用等价于下面的调用：

```
let args = std::env::args().collect::<Vec<String>>();
let args = std::env::args().collect::<HashSet<String>>();
```

但如果 `collect` 的类型参数只有唯一一种可能，那么 Rust 会自动为你填充它。当你写出 `args` 的类型时，就是确保了这种情况。

15.4.14 Extend trait

如果一个类型实现了 `std::iter::Extend` trait，那么它的 `extend` 方法可以把一个可迭代对象的 item 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`，因此它们都有这个方法，就算是 `String` 也有。而数组和切片这种长度固定的集合没有这个方法。

这个 trait 的定义如下：

```
trait Extend<A> {
    fn extend<T>(&mut self, iter: T)
        where T: IntoIterator<Item=A>;
}
```

显然，这和 `std::iter::FromIterator` 非常相似：不过后者创建一个新集合，而 `Extend` 扩展一个已有集合。事实上，标准库中好几个类型的 `FromIterator` 实现都是简单地创建一个新的空集合，然后调用 `extend` 来填充它。例如，`std::collections::LinkedList` 的 `FromIterator` 就是以这种方式工作：

```
impl<T> FromIterator<T> for LinkedList<T> {
    fn from_iter<I: IntoIter<Item = T>>(iter: I) -> Self {
        let mut list = Self::new();
        list.extend(iter);
        list
    }
}
```

15.4.15 partition

`partition` 方法将一个迭代器的 item 分成两个集合，使用一个闭包来决定每个 item 属于哪个集合：

```
let things = ["doorknob", "mushroom", "noodle", "giraffe", "grapefruit"];

// 一个令人震惊的事实：活物的名字都以奇数字母开头。
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"]);
```

类似于 `collect`，`partition` 也可以创建任意类型的集合，不过两个集合的类型必须相同。类似于 `collect`，你必须指明返回类型：上面的例子中写出了 `living` 和 `nonliving` 的类型并让类型推导选择合适的类型参数来调用 `partition`。

`partition` 的签名如下：

```
fn partition<B, F>(&self, f: F) -> (B, B)
    where Self: Sized,
          B: Default + Extend<Self::Item>,
          F: FnMut(&Self::Item) -> bool;
```

`collect` 要求返回类型实现了 `FromIterator`, 而 `partition` 要求 `std::default::Default` (所有的 Rust 集合都实现了它来返回一个空集合) 和 `std::default::Extend`。

其他语言提供把一个迭代器分成两个迭代器的 `partition` 操作, 而不是构建两个集合。但这并不适合 Rust: 如果要实现分成两个迭代器, 那么那些由底层迭代器产生但还未被分类后的迭代器产生的 item 必须被缓存; 无论如何, 最终都需要在内部构建某种类型的集合。

15.4.16 `for_each` 和 `try_for_each`

`for_each` 方法简单地对每一个 item 应用一个闭包:

```
[ "doves", "hens", "birds" ].iter()
    .zip([ "turtle", "french", "calling" ].iter())
    .zip(2..5)
    .rev()
    .map(|((item, kind), quantity)| {
        format!("{} {} {}", quantity, kind, item)
    })
    .for_each(|gift| {
        println!("You have received: {}", gift);
    });
}
```

这会打印出:

```
You have received: 4 calling birds
You have received: 3 french hens
You have received: 2 turtle doves
```

它非常类似于一个简单的 `for` 循环, 在循环中你还可以使用 `break` 和 `continue` 控制语句。但使用 `for` 循环处理这种很长的链式适配器调用会有一点尴尬:

```
for gift in [ "doves", "hens", "birds" ].iter()
    .zip([ "turtle", "french", "calling" ].iter())
    .zip(2..5)
    .rev()
    .map(|((item, kind), quantity)| {
        format!("{} {} {}", quantity, kind, item)
    })
{
    println!("You have received: {}", gift);
}
```

用于绑定的模式 `gift` 最终可能会距离使用它的循环体非常远。

如果你的闭包需要容错或者提前退出, 你可以使用 `try_for_each`:

```

    ...
    .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` 和 `Iterator` trait，这样本章之前展示过的所有适配器和消耗器，还有其他可以和标准库迭代器接口协同工作的库和 crate 都可以使用。在本节中，我们将展示一个简单的范围类型的迭代器，然后展示一个更复杂的二叉树类型的迭代器。

假设我们有下面的范围类型（标准库的 `std::ops::Range<T>` 类型的简化）：

```

struct I32Range {
    start: i32,
    end: i32
}

```

迭代一个 `I32Range` 需要两个状态：当前的值和终点值。`I32Range` 类型本身非常适合存储这两个状态：使用 `start` 作为下一个值，`end` 作为终点。因此你可以这么实现 `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
    }
}

```

这个迭代器产生 `i32` item，所以 `i32` 就是 `Item` 类型。如果迭代已经结束，`next` 应该返回 `None`；否则，它产生下一个值并更新当前的状态准备好下一次调用。

当然，`for` 循环会使用 `IntoIterator::into_iter` 来把操作数转换成迭代器。但标准库为每一个实现 `Iterator` 的类型自动提供了 `IntoIterator` 实现，因此 `I32Range` 已经可以使用：

```

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 标准指定了精确的结果。
assert_eq!(pi as f32, std::f32::consts::PI);

```

但 `I32Range` 是一种特殊情况，它的可迭代对象和迭代器恰好是同一种类型。很多情况并不是这么简单。例如，这里有一个第 10 章的二叉树类型：

```

enum BinaryTree<T> {
    Empty,
    NonEmpty(Box<TreeNode<T>>)
}

struct TreeNode<T> {
    element: T,
    left: BinaryTree<T>,
    right: BinaryTree<T>
}

```

经典的遍历二叉树的方法是递归，使用函数调用栈来追踪在树中的位置和已经访问过的节点。但当为 `BinaryTree<T>` 实现 `Iterator` 时，每一次 `next` 调用都必须产生一个值并且返回。为了追踪要产生的树节点，迭代器必须保持自己的栈。这里有一个 `BinaryTree` 的可能的迭代器类型：

```

use self::BinaryTree::*;

// 中序遍历`BinaryTree`时的状态
struct TreeIter<'a, T> {
    // 一个树节点的引用的栈。因为我们要使用`Vec`的
    // `push` 和 `pop` 方法，栈的顶部是 `vector` 的尾部。
    //
    // 迭代器下一个要访问的节点在栈顶，
    // 那些还没访问的祖先节点在它下面。如果栈为空，
    // 就代表迭代结束了。
    unvisited: Vec<&'a TreeNode<T>>
}

```

当我们创建一个新的 `TreeIter` 时，它的初始状态应该是即将产生树中最左边的节点。根据 `unvisited` 栈的规则，它的栈顶应该是那个叶节点，再往下就是它的祖先节点：树中左侧

边缘上的节点。我们可以从根节点访问左侧边缘一直到叶节点，把遇到的所有节点都入栈，来初始化 `unvisited`，因此我们为 `TreeIter` 定义一个方法来实现这个过程：

```
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;
        }
    }
}
```

用 `mut tree` 可以让循环在遍历左侧边缘的过程中改变 `tree` 指向的节点，但因为 `tree` 是一个共享引用，所以它不能改变那些节点本身。

有了这个辅助方法，我们可以给 `BinaryTree` 一个 `iter` 方法来返回一个迭代树的迭代器：

```
impl<T> BinaryTree<T> {
    fn iter(&self) -> TreeIter<T> {
        let mut iter = TreeIter { unvisited: Vec::new() };
        iter.push_left_edge(self);
        iter
    }
}
```

`iter` 方法先构造了一个 `unvisited` 栈为空的 `TreeIter`，然后调用 `push_left_edge` 来初始化它。按照 `unvisited` 栈的规则，最左侧的节点在栈顶。

遵循标准库的实践，我们可以为二叉树的引用实现 `IntoIterator`，然后在里面调用 `BinaryTree::iter`：

```
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()
    }
}
```

`IntoIter` 定义将 `TreeIter` 设置为 `&BinaryTree` 的迭代器类型。

最后，在 `Iterator` 的实现中，我们需要实际遍历这棵树。类似于 `BinaryTree` 的 `iter` 方法，迭代器的 `next` 方法也需要遵守栈的规则：

```
impl<'a, T> Iterator for TreeIter<'a, T> {
    type Item = &'a T;
```

```

fn next(&mut self) -> Option<&'a T> {
    // 找到这一次迭代要产生的节点,
    // 或者结束迭代。(如果结果是`None`就通过
    // ``?`运算符立即返回。)
    let node = self.unvisited.pop()?;
    // 在`node`之后, 下一个产生的应该是`node`的右子树
    // 中的最左侧的节点, 因此添加这条线上的节点。我们的辅助函数
    // 恰好就是我们需要的。
    self.push_left_edge(&node.right);
    // 产生这个节点的值的引用。
    Some(&node.element);
}
}

```

如果栈为空, 就表明迭代结束了。否则, `node` 是现在要访问的节点的引用, 这次调用会返回一个它的 `element` 字段的引用。但首先, 我们必须把迭代器的状态更新到下一个节点。如果这个节点有一个右子树, 那么下一个要访问的就是这棵子树中最左侧的节点, 我们可以调用 `push_left_edge` 来添加它和它的未访问的祖先节点到栈里。但如果这个节点没有右子树, 那么 `push_left_edge` 没有效果, 这正是我们想要的: 这时新的栈顶就是 `node` 的第一个未被访问的祖先节点。

有了 `IntoIterator` 和 `Iterator` 实现, 我们最终可以使用一个 `for` 循环来迭代 `BinaryTree` 的引用。这里用到了填充二叉树中 `BinaryTree` 的 `add` 方法:

```

// 建造一棵小树。
let mut tree = BinaryTree::Empty;
tree.add("Jaeger");
tree.add("robot");
tree.add("droid");
tree.add("mecha");

// 迭代它。
let mut v = Vec::new();
for kind in &tree {
    v.push(*kind);
}
assert_eq!(v, ["droid", "Jaeger", "mecha", "robot"]);

```

图 15-1 展示了当我们遍历这棵树的过程中 `unvisited` 栈的行为。在每一次迭代中, 下一个要被访问的节点都是栈顶, 所有还未访问过的祖先节点都在它之下。

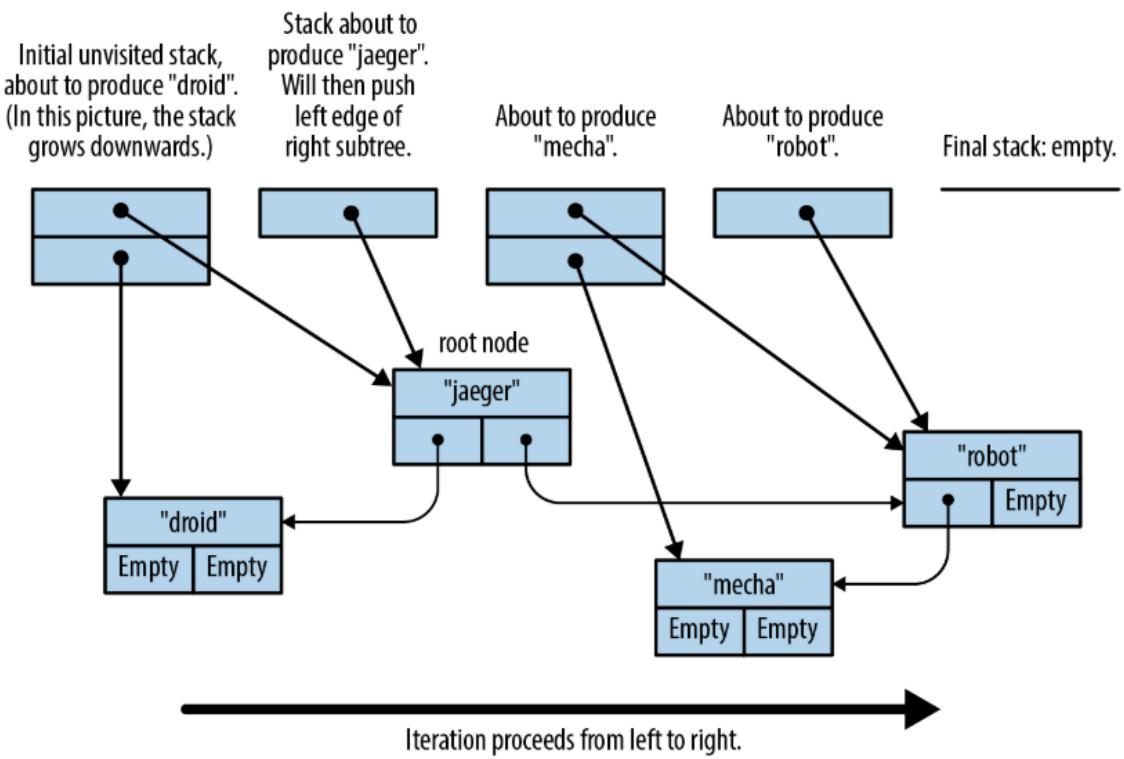


图 15-1: 迭代二叉树

所有常用的迭代器适配器和消耗器都已经准备好在我们的树上使用：

```
assert_eq!(tree.iter()
    .map(|name| format!("mega-{}", name))
    .collect::<Vec<_>>(),
    vec!["mega-droid", "mega-jaeger",
        "mega-mecha", "mega-robot"]);
```

迭代器是 Rust 哲学的体现，即提供强大的、零成本、能提高代码的表现力和可读性的抽象。迭代器并不能完全替代循环，但它们提供了一种功能强大的原语，同时具有内建的惰性求值和高性能的特点。

Chapter 16

集合

We all behave like Maxwell' s demon. Organisms organize. In everyday experience lies the reason sober physicists across two centuries kept this cartoon fantasy alive. We sort the mail, build sand castles, solve jigsaw puzzles, separate wheat from chaff, rearrange chess pieces, collect stamps, alphabetize books, create symmetry, compose sonnets and sonatas, and put our rooms in order, and all this we do requires no great energy, as long as we can apply intelligence.

——James Gleick, The Information: A History, a Theory, a Flood

Rust 标准库里包含几种集合 (*collection*)，它们是在内存中存储数据的泛型类型。我们已经在本书的很多地方使用过集合，例如 `Vec` 和 `HashMap`。在本章中，我们将详细介绍这两种类型的方法，以及其他六种标准集合。但在我们开始之前，让我们先讨论一下 Rust 的集合和其他语言中的集合的一些不同之处。

首先，移动和借用无处不在。Rust 使用移动来避免深拷贝。这就是为什么 `Vec<T>::push(item)` 方法以值获取参数，而不是以引用。值会被移动进 vector。[第 4 章](#) 中的图展示了实践中的表现：在 Rust 中把一个 `String` 添加到 `Vec<String>` 中很快，因为 Rust 不需要拷贝字符串的字符数据，字符串的所有权归属也总是很清楚。

其次，当集合改变大小或者被修改的同时还有指向它们的数据的指针时，Rust 不会有无效性错误——即悬垂指针。无效性错误是 C++ 中另一种未定义行为的来源，即使在内存安全的语言中也可能导致 `ConcurrentModificationException`。Rust 借用检查器会在编译器检查出它们。

最后，Rust 没有 `null`，因此我们将在其他语言中需要 `null` 的地方看到 `Option`。

除了这些不同之外，Rust 的集合可能和你预期的差不多。如果你是经验丰富的程序员并且时间不多，你可以跳过这部分，但不要跳过[条目](#)。

16.1 概述

表 16-1 展示了 Rust 的 8 种标准集合。它们都是泛型类型。

表 16-1: 标准集合总结

集合	说明	其他语言中的类似集合类型		
		C++	Java	Python
<code>Vec<T></code>	可增长的数组	<code>vector</code>	<code>ArrayList</code>	<code>list</code>
<code>VecDeque<T></code>	双端队列(可增长环形缓冲区)	<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> where T: Ord</code>	大顶堆	<code>priority_queue</code>	<code>PriorityQueue</code>	<code>heapq</code>
<code>HashMap<K, V> where K: Eq + Hash</code>	键值哈希表	<code>unordered_map</code>	<code>HashMap</code>	<code>dict</code>
<code>BTreeMap<K, V> where K: Ord</code>	有序键值表	<code>map</code>	<code>TreeMap</code>	—
<code>HashSet<T> where T: Eq + Hash</code>	基于哈希的无序集合	<code>unordered_set</code>	<code>HashSet</code>	<code>set</code>
<code>BTreeSet<T> where T: Ord</code>	有序集合	<code>set</code>	<code>TreeSet</code>	—

`Vec<T>`、`HashMap<K, V>`、`HashSet<T>`是最常用的集合类型。其他的集合都有适用的场景。这一章将轮流讨论每一个集合类型：

`Vec<T>`

一个可增长的、在堆上分配的、`T` 类型值的数组。本章中大约一半的篇幅专门介绍 `Vec` 和它的常用方法。

`VecDeque<T>`

类似于 `Vec<T>`，但是用作先进先出队列会更好。它支持高效地在首部和尾部添加或移除元素，但这种能力的代价是其他操作会稍微慢一点。

`BinaryHeap<T>`

一个优先队列。`BinaryHeap` 中的值按照一定结构组织，因此总是可以高效地找到和移除最大值。

HashMap<K, V>

一个键值对的表。通过键查找值很快速。表中的条目以任意顺序存储。

BTreeMap<K, V>

类似于 HashMap<K, V>, 但按键的顺序保持条目有序。一个 BTreeMap<String, i32> 按照 String 的比较顺序存储条目。除非你需要条目保持有序，否则 HashMap 会更快。

HashSet<T>

类型 T 的值的集合。添加和删除元素都很快，查询一个值是否在集合中也很快。

BTreeSet<T>

类似于 HashSet<T>, 但保持元素有序。同样，除非你想要数据保持有序，否则 HashSet 会更快。

因为 LinkedList 很少使用（并且在大多数情况下都有更好的替代，无论是性能还是接口），因此我们不会在这里介绍它。

16.2 Vec<T>

我们假设你对 Vec 已经有了一定了解，因为我们在本书的很多地方都已经使用过它。简要的介绍见 [vector](#)。这里我们只会描述它的方法以及深入它的内部工作原理。

最简单的创建 vector 的方式是使用 `vec!` 宏：

```
// 创建一个空的 vector
let mut numbers: Vec<i32> = vec![];

// 用给定的内容创建一个 vector
let words = vec!["step", "on", "no", "pets"];
let mut buffer = vec![0u8; 1024]; // 1024 个 0 字节
```

正如我们在[第 4 章](#)所述，vector 有三个字段：长度、容量、和一个指向堆上分配的缓冲区的指针。[图 16-1](#)展示了上面的 vector 在内存中的视图。空 vector `numbers` 的初始长度为 0。在它添加第一个元素之前不会有堆内存被分配。

类似于所有集合，Vec 实现了 `std::iter::FromIterator`，因此你可以对任何迭代器调用 `.collect()` 方法来创建一个 vector，正如[构建集合：collect 和 FromIterator](#) 中所述：

```
// 将一个其他集合转换成 vector
let my_vec = my_set.into_iter().collect::<Vec<String>>();
```

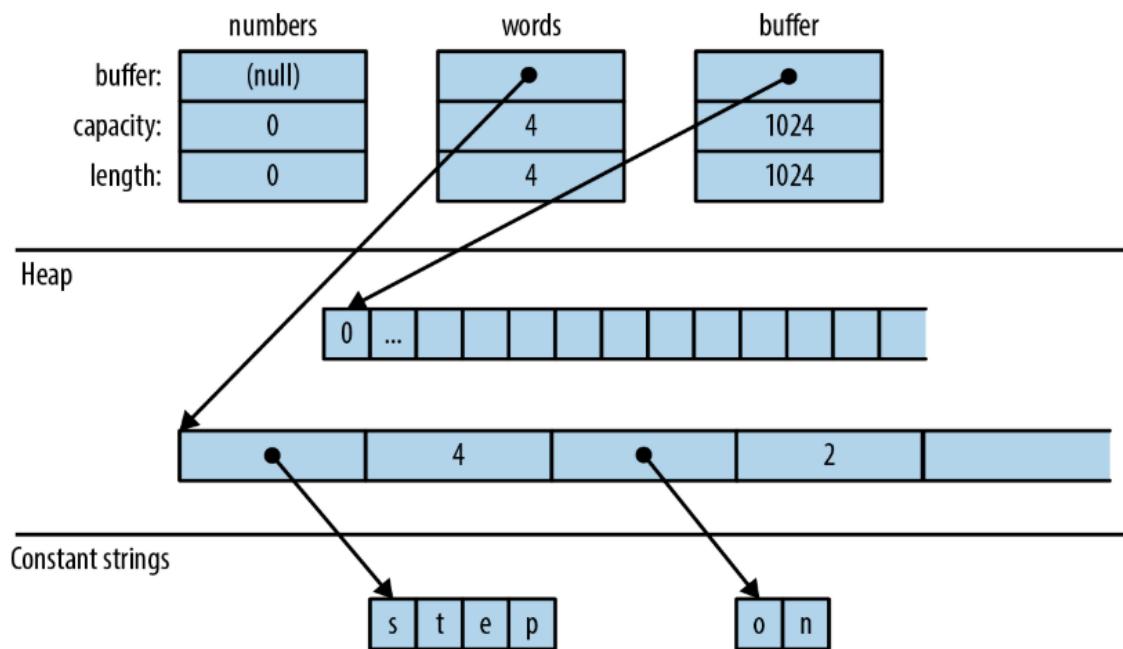


图 16-1: vector 的内存布局: words 的每个元素是一个由指针和长度组成的 &str 值

16.2.1 访问元素

通过索引访问数组、切片或 vector 的元素非常直观：

```
// 获取一个元素的引用
let first_line = &lines[0];

// 获取一个元素的拷贝
let fifth_number = numbers[4];           // 需要 Copy
let second_number = lines[1].clone();     // 需要 Clone

// 获取一个切片的引用
let my_ref = &buffer[4..12];

// 获取一个切片的拷贝
let my_copy = buffer[4..12].to_vec();    // 需要 Clone
```

当索引越界时所有这些方式都会 panic。

Rust 对数字类型很挑剔，vector 也不例外。vector 的长度和索引都是 `usize` 类型。尝试使用 `u32`、`u64`、`isize` 作为 vector 的索引会导致错误。必要时你可以使用 `n as usize` 来转换，见 [类型转换](#)。

有几种方法提供了便捷地访问 vector 或切片的特定元素的方法（注意所有的切片方法都能

用于数组和 vector)：

slice.first()

返回 slice 的第一个元素的引用。返回类型是 Option<&T>，因此如果 slice 为空时返回值为 None，不为空时返回值为 Some(&slice[0]):

```
if let Some(item) = v.first() {  
    println!("We got one! {}", item);  
}
```

slice.last()

和上边相似，不过返回最后一个元素的引用。

slice.get(index)

返回 slice[index] 的引用，如果存在的话。如果 slice 的元素数量小于 index+1，那么返回 None:

```
let slice = [0, 1, 2, 3];  
assert_eq!(slice.get(2), Some(&2));  
assert_eq!(slice.get(4), None);
```

slice.first_mut(), slice.last_mut(), slice.get_mut(index)

与上面的类似，不过借用 mut 引用：

```
let mut slice = [0, 1, 2, 3];  
{  
    let last = slice.last_mut().unwrap(); // 最后一个元素类型: &mut i32  
    assert_eq!(*last, 3);  
    *last = 100;  
}
```

因为以值返回 T 意味着移动它，因此访问元素的方法通常返回元素的引用。

一个例外是 .to_vec() 方法，它获取拷贝：

slice.to_vec()

克隆整个切片，返回一个新的 vector：

```
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]);
```

只有当元素可以拷贝时这个方法才可用，即 where `T: Clone`。

16.2.2 迭代

`vector` 和切片可以以值或者以引用迭代，遵循 `IntoIterator` 实现中介绍的模式：

- 迭代 `Vec<T>` 会产生 `T` 类型的 item。元素被逐个移出 `vector` 消耗掉。
- 迭代 `&[T; N]`, `&[T]`, `&Vec<T>`——即数组、切片或 `vector` 的引用——会产生 `&T` 类型的 item，每一个 item 指向一个元素，不会移动元素。
- 迭代 `&mut [T; N]`, `&mut [T]`, `&mut Vec<T>` 产生 `&mut T` 类型的 item。

数组、切片和 `vector` 还有 `.iter()` 和 `.iter_mut()` 方法（见 `iter` 和 `iter_mut` 方法）创建产生元素的引用的迭代器。

我们将在 `切分` 中介绍一些更有趣的迭代切片的方法。

16.2.3 增长和缩减 `vector`

数组、切片或 `vector` 的长度 (`length`) 是它包含的元素的数量：

`slice.len()`

返回一个 `slice` 的长度，类型为 `usize`。

`slice.is_empty()`

当 `slice` 不包含元素时为真（即 `slice.len() == 0`）。

本节剩余的方法都是关于增长和缩减 `vector`。它们不能用于数组和切片，因为这两种类型一旦被创建之后就不能改变大小。

`vector` 的所有元素都存储在一个在堆上分配的连续内存块中。`vector` 的容量 (`capacity`) 是指当前的内存块最多能存储的元素数量。`Vec` 通常会替你管理容量，当需要增长时它会自动分配更大的缓冲区并把元素都移动过去。还有一些显式管理容量的方法：

`Vec::with_capacity(n)`

创建一个容量为 `n` 的新的空 `vector`。

`vec.capacity()`

返回 `vec` 的容量，类型是 `usize`。`vec.capacity() >= vec.len()` 总是为真。

`vec.reserve(n)`

保证 `vector` 的剩余空间至少还能再存储 `n` 个或更多元素：即 `vec.capacity()` 至少是 `vec.len() + n`。如果已经有足够的空间，它不做任何事。否则，它会分配一个更大的缓冲区并且把 `vector` 的内容移动过去。

`vec.reserve_exact(n)`

类似于 `vec.reserve(n)`，但告诉 `vec` 不要为未来的增长分配额外的空间。调用它之后，`vec.capacity()` 等于 `vec.len() + n`。

`vec.shrink_to_fit()`

当 `vec.capacity()` 大于 `vec.len()` 时尝试释放额外的内存。

`Vec<T>` 有很多添加或移除元素的方法，同时改变 `vector` 的长度。所有这些方法都以 `mut` 引用获取 `self` 参数。

下面这两个方法在 `vector` 的末尾添加或移除一个元素：

`vec.push(value)`

把 `value` 添加到 `vec` 的末尾。

`vec.pop()`

移除并返回最后一个元素。返回类型是 `Option<T>`。当 `vector` 已经为空时返回 `None`，否则返回 `Some(x)`。

注意 `.push()` 以值而不是以引用获取参数。类似的，`.pop()` 返回被弹出的值，而不是引用。本节中剩余的大部分方法也是这样。它们从 `vector` 移出或移进值。

这两个方法向 `vector` 中添加值或者从 `vector` 中移出值：

`vec.insert(index, value)`

在 `vec[index]` 处插入给定的 `value`，把 `vec[index..]` 中的值都向后移动一个位置来腾出空间。如果 `index > vec.len()` 会 panic。

`vec.remove(index)`

移除并返回 `vec[index]`, 把 `vec[index+1..]` 中的值向左移动一个位置来消除缝隙。

`.insert()` 和 `.remove()` 都很慢, 因为有很多元素需要移动。

有四个方法可以将 `vector` 的长度调整为指定值:

```
vec.resize(new_len, value)
```

将 `vec` 的长度设为 `new_len`。如果这会增大 `vec` 的长度, 将会用 `value` 的拷贝填充新空间。元素的类型必须实现了 `Clone` trait。

```
vec.resize_with(new_len, closure)
```

类似于 `vec.resize`, 但调用闭包来构造每一个新元素。它可以用于元素没有实现 `Clone` 的 `vector`。

```
vec.truncate(new_len)
```

将 `vec` 的长度缩减到 `new_len`, 丢弃 `vec[new_len..]` 范围内的所有元素。若 `vec.len()` 小于等于 `new_len`, 则什么也不做。

```
vec.clear()
```

删除 `vec` 的所有元素。等价于 `vec.truncate(0)`。

有四个方法可以一次添加或移除很多元素:

```
vec.extend(iterable)
```

将 `iterable` 的所有 `item` 按顺序添加到 `vec` 的末尾。它类似于多值版本的 `.push()`。`iterable` 参数可以是任何实现了 `IntoIterator<Item=T>`。

这个方法如此有用以至于有一个专门的 trait `Extend`, 所有的标准集合都实现了它。不幸的是, 这导致 `rustdoc` 将 `.extend()` 和其他 trait 的方法放在生成的 HTML 底部的一堆方法中, 因此当你需要它时很难找到它。你必须记住它! 更多内容见 [Extend trait](#)。

```
vec.split_off(index)
```

类似于 `vec.truncate(index)`, 除了它返回一个 `Vec<T>` 包含 `vec` 尾部被移除的元素。它类似于 `.pop()` 的多值版本。

```
vec.append(&mut vec2)
```

这会把 `vec2` 的所有元素移动进 `vec`，其中 `vec2` 是另一个 `Vec<T>` 类型的 vector。调用之后，`vec2` 变为空。

这类似于 `vec.extend(vec2)`，除了调用之后 `vec2` 仍然存在，并且容量不变。

`vec.drain(range)`

这会从 `vec` 中移除范围 `vec[range]`，并返回一个迭代被移除元素的迭代器，其中 `range` 是一个范围值，例如 `..` 或 `0..4`。

还有一些选择性移除 vector 元素的古怪方法：

`vec.retain(test)`

移除所有没有通过给定测试的方法。`test` 参数是一个实现了 `FnMut(&T) -> bool` 的函数或闭包。对于 `vec` 的每一个元素，它会调用 `test(&element)`，如果返回 `false`，元素将会被移出 vector 然后丢弃。

不考虑性能的话，这类似于：

```
vec = vec.into_iter().filter(test).collect();
```

`vec.dedup()`

丢弃相邻的重复元素。它类似于 Unix 的 `uniq` shell 工具。它会扫描 `vec` 中寻找相邻的重复元素，然后丢弃掉多余的重复值，只留下一个：

```
let mut byte_vec = b"Mississippi".to_vec();
byte_vec.dedup();
assert_eq!(&byte_vec, b"Missipi");
```

注意最后的输出中仍然有两个's'字符。这个方法只移除相邻的 (*adjacent*) 重复值。为了移除所有的重复值，你有三种选择：调用 `.dedup()` 之前先排序 vector，将数据移动到一个 `HashSet<T>` 和 `BTreeSet<T>`，或者（为了保持元素原本的顺序）使用这个 `.retain()` 技巧：

```
let mut byte_vec = b"Mississippi".to_vec();

let mut seen = HashSet::new();
byte_vec.retain(|r| seen.insert(*r));

assert_eq!(&byte_vec, b"Misp");
```

这段代码的原理是当集合中已经包含要插入的 item 时 `.insert()` 会返回 `false`。

```
vec.dedup_by(same)
```

类似于 `vec.dedup()`, 但它使用函数或者闭包 `same(&mut elem1, &mut elem2)`, 而不是 `==` 运算符, 来检查两个相邻元素是否被认为相等。

```
vec.dedup_by_key(key)
```

类似于 `vec.dedup()`, 但当 `key(&mut elem1) == key(&mut elem2)` 时它认为两个元素相等。

例如, 如果 `errors` 是一个 `Vec<Box<dyn Error>>`, 你可以写:

```
// 移除消息重复的错误。
errors.dedup_by_key(|err| err.to_string());
```

这一节介绍的所有方法中, 只有 `.resize()` 可能会拷贝值。其他的通过移动值来工作。

16.2.4 连接

两个方法可以用于数组的数组 (*array of array*), 即元素类型是数组、切片、vector 的数组、切片、vector:

```
slices.concat()
```

返回一个所有切片连接成的 vector:

```
assert_eq!([[1, 2], [3, 4], [5, 6]].concat(),
           vec![1, 2, 3, 4, 5, 6]);
```

```
slices.join(&separator)
```

同上, 除了会在切片之间插入 `separator` 的拷贝:

```
assert_eq!([[1, 2], [3, 4], [5, 6]].join(&0),
           vec![1, 2, 0, 3, 4, 0, 5, 6]);
```

16.2.5 切分

很容易一次获得数组、切片、vector 中的很多元素的非 `mut` 引用:

```
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..];
```

但一次获得多个 mut 引用不是这么容易：

```
let mut v = vec![0, 1, 2, 3];
let a = &mut v[i];
let b = &mut v[j]; // error: 不能同时借用`v`的
                  // 多个可变引用。

*a = 6;          // 引用`a`和`b`在这里使用了,
*b = 7;          // 因此它们的生命周期一定会重叠。
```

Rust 禁止这样，因为如果 $i == j$ ，那么 a 和 b 将是同一个整数的两个 mut 引用，这违背了 Rust 的安全性的规则。（见 [共享 vs 可变](#)）。

Rust 有几个方法可以一次借用数组、切片、vector 的两个或更多元素的 mut 引用。和上面的代码不同，这些方法是安全的，因为它们从设计上保证了只会把数组分割成非重叠 (*nonoverlapping*) 区域。这些方法也可以用于非 mut 切片，因此它们有 mut 和非 mut 版本。

[图 16-2](#) 展示了这些方法。

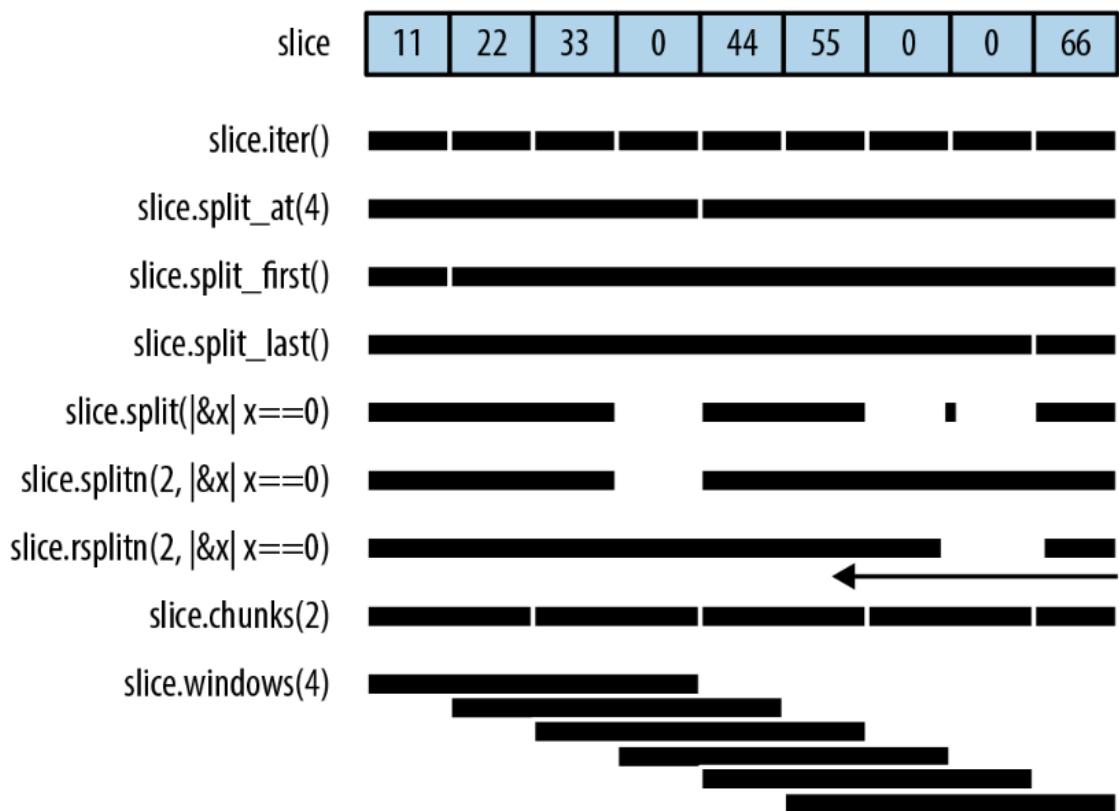


图 16-2：分割方法展示（注意：`slice.split()` 输出中的小矩形是空的切片，因为两侧都是分隔符，`rsplitn` 的输出是按照从后往前的顺序，这一点和其他的不同）。

这些方法中没有一个会直接修改数组、切片或 vector，它们都返回部分数据的引用：

`slice.iter()`, `slice.iter_mut()`

产生 `slice` 的每个元素的引用。我们在迭代中已经介绍过它们。

`slice.split_at(index)`, `slice.split_at_mut(index)`

把切片分成两半，返回一个 pair。`slice.split_at(index)` 等价于 `(&slice[..index], &slice[index..])`。如果 `index` 越界会 panic。

`slice.split_first()`, `slice.split_first_mut()`

返回一个 pair：第一个元素的引用 (`slice[0]`) 和其余所有元素的切片 (`slice[1..]`)。
`.split_first()` 的返回值类型是 `Option<(&T, &[T])>`；如果 `slice` 为空，返回 `None`。

`slice.split_last()`, `slice.split_last_mut()`

类似上一个，不过划分出最后一个元素而不是第一个。
`.split_last()` 的返回类型也是 `Option<(&T, &[T])>`。

`slice.split(is_sep)`, `slice.split_mut(is_sep)`

将 `slice` 切分成一个或更多子切片，使用函数或闭包 `is_sep` 来判断在哪里切分。它们返回一个迭代子切片的迭代器。

当消耗迭代器时，它们会对切片中的每个元素调用 `is_sep(&element)`。如果返回 `true`，那么这个元素就是一个分隔符。分隔符不包含在任何输出的字切片中。

输出总是包含至少一个子切片，每有一个分隔符就加一个子切片。如果有相邻的分隔符或者 `slice` 的两端是分隔符都会产生空的子切片。

`slice.rsplit(is_sep)`, `slice.rsplit_mut(is_sep)`

类似于 `slice` 和 `slice_mut`，但从最后一个切片开始。

`slice.splitn(n, is_sep)`, `slice.splitn_mut(n, is_sep)`

类似上面的方法，不过最多产生 `n` 个子切片。当发现了前 `n-1` 个切片之后就不会再调用 `is_sep`。最后一个子切片将包含剩余的所有元素。

`slice.rsplitn(n, is_sep)`, `slice.rsplitn_mut(n, is_sep)`

类似于 `.splitn()` 和 `.splitn_mut()`，除了反向扫描切片。就是说，这个方法会在切片中最后 `n-1` 个分隔符处切分，而不是前 `n-1` 个，并且从尾部开始产生子切片。

```
slice.chunks(n), slice.chunks_mut(n)
```

返回一个产生长度为 n 的非重叠子切片的迭代器。如果 n 不能整除 slice.len()，最后一个块的元素数量将小于 n。

```
slice.rchunks(n), slick.rchunks_mut(n)
```

类似于 slice.chunks() 和 slice.chunks_mut()，但是从切片的尾部开始。

```
slice.chunks_exact(n), slice.chunks_exact_mut(n)
```

返回一个产生长度为 n 的非重叠子切片的迭代器。如果 n 不能整除 slice.len()，最后一个块（元素数量小于 n）可以通过结果的 remainder() 方法获得。

```
slice.rchunks_exact(n), slice.rchunks_exact_mut(n)
```

类似于 slice.chunks_exact 和 slice.chunks_exact_mut，但从切片的尾部开始。

还有一些其他迭代子切片的方法：

```
slice.windows(n)
```

返回一个效果类似于“滑动窗口”的迭代器。它产生 slice 中相邻的 n 个元素的子切片。第一个产生的值是 &slice[0..n]，第二个是 &slice[1..n+1]，以此类推。

如果 n 大于 slice 的长度，将不会产生切片。如果 n 是 0，这个方法会 panic。

例如，如果 days.len() == 31，那么我们可以调用 days.windows(7) 产生 days 中所有 7 天的区间。

在探索数据列的变化趋势时一个大小为 2 的滑动窗口会很有用：

```
let changes = daily_high_temperatures
    .windows(2)           // 获得相邻天的温度
    .map(|w| w[1] - w[0]) // 温度改变了多少
    .collect::<Vec<_>>();
```

因为子切片是重叠的，所以这个方法没有返回 mut 引用的版本。

16.2.6 交换

有一些交换切片内容的便捷方法：

```
slice.swap(i, j)
```

交换元素 slice[i] 和 slice[j]。

```
slice_a.swap(&mut slice_b)
```

交换 `slice_a` 和 `slice_b` 的全部内容。`slice_a` 和 `slice_b` 长度必须相同。`vector` 有一个高效地移除任何元素的方法：

```
vec.swap_remove(i)
```

移除并返回 `vec[i]`。这类似于 `vec.remove(i)`，除了它不是把剩余的元素往前移动来消除间隙，而是把 `vec` 的最后一个元素移动到间隙。当你不关心 `vector` 中元素的顺序时这很有用。

16.2.7 排序和搜索

切片提供了三个用于排序的方法：

```
slice.sort()
```

按增序排序元素。只有当元素类型实现了 `Ord`，这个方法才可用。

```
slice.sort_by(cmp)
```

使用函数或闭包 `cmp` 指定顺序来排序 `slice` 的元素。`cmp` 必须实现了 `Fn(&T, &T) -> std::cmp::Ordering`。

手动实现 `cmp` 很麻烦，除非你用 `.cmp` 方法：

```
students.sort_by(|a, b| a.last_name.cmp(&b.last_name));
```

如果要先比较一个字段，再比较第二个字段，可以直接比较元组：

```
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)
```

以给定的函数或闭包 `key` 作为排序键将 `slice` 的元素按照增序排序。`key` 的类型必须实现了 `Fn(&T) -> K where K: Ord`。当 `T` 包含一个或更多有序字段时这很有用，因为它可以按照多种方式排序：

```
// 按照平均绩点排序，低的靠前
students.sort_by_key(|s| s.grade_point_average());
```

注意这些排序键在排序过程中并不会被缓存，因此 `key` 函数可能会被调用超过 `n` 次。出于技术上的原因，`key(element)` 不能返回任何从元素借用的引用。这样不能工作：

```
students.sort_by_key(|s| &s.last_name); // 错误：无法推断生命周期
```

Rust 无法查明生命周期。但在这种情况下，可以调用 `.sort_by()` 作为替代。这三种方法都是稳定性排序。

为了实现反向排序，你可以使用 `sort_by`，然后在 `cmp` 闭包中交换两个参数。将参数写为 `|b, a|` 而不是 `|a, b|` 可以高效地产生相反的顺序。或者你可以在排序之后调用 `.reverse()` 方法：

```
slice.reverse()
```

反转一个切片。

一旦一个切片被排过序，它就可以被高效地搜索：

```
slice.binary_search(&value), slice.binary_search_by(&value, cmp),  
slice.binary_search_by_key(&value, key)
```

这些方法都在有序的 `slice` 中搜索 `value`。注意 `value` 是以引用传递。

它们的返回类型都是 `Result<usize, usize>`。如果 `slice[index]` 在指定的排序顺序下等于 `value` 它们会返回 `Ok(index)`。如果没有这样的元素会返回 `Err(insertion_point)`，在 `insertion_point` 处插入 `value` 将保持有序。

当然，二分搜索只在切片在指定的顺序下有序时才能工作。否则，结果将是任意值——garbage in, garbage out。

因为 `f32` 和 `f64` 有 `Nan` 值，它们没有实现 `Ord`，因此不能被直接用作排序或二分搜索的键。为了得到可以用于浮点数的类似方法，使用 `ord_subset crate`。

还有一个在无序 `vector` 中搜索元素的方法：

```
slice.contains(&value)
```

如果 `slice` 中有任何一个元素等于 `value` 则返回 `true`。它简单地检查切片的每个元素，直到找到要查找的值。另外，`value` 也是以引用传递。

为了查找一个切片中某一个值的位置，类似于 JavaScript 中的 `array.indexOf(value)`，使用迭代器：

```
slice.iter().position(|x| *x == value)
```

这会返回 `Option<usize>`。

16.2.8 比较切片

如果类型 T 支持 == 和 != 运算符（`PartialEq` trait，见[相等性比较](#)），那么数组 [T; N]、切片 [T]、vector `Vec<T>` 也支持这些运算符。当两个切片的长度和相应的元素都相等时两个切片才相等。数组和 vector 也是一样。

如果 T 支持运算符 <、<=、>、>=（`PartialOrd` trait，见[顺序性比较](#)），那么 T 的数组、切片和 vector 也支持。切片的比较按照字典序进行。

有两个便捷的方法进行常用的切片比较：

`slice.starts_with(other)`

如果 `slice` 起始的值序列等于 `other` 的元素则返回 `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)`

和上边类似但检查 `slice` 的末尾:

```
assert_eq!([1, 2, 3, 4].ends_with(&[3, 4]), true);
```

16.2.9 随机元素

Rust 的标准库中并不包含随机数。`rand` crate 提供了它们，还提供了两个方法从数组、切片、vector 中获取随机元素：

`slice.choose(&mut rng)`

返回切片中随机一个元素的引用。类似于 `slice.first()` 和 `slice.last()`，除了它返回一个 `Option<&T>`，如果切片为空，返回值为 `None`。

`slice.shuffle(&mut rng)`

随机打乱切片中的元素的顺序。切片必须以 `mut` 引用传递。

这些是 `rand::Rng` trait 的方法，因此为了调用它们你需要一个 `Rng`，它是一个随机数生成器。幸运的是，可以调用 `rand::thread_rng()` 来获得一个。要想打乱 vector `my_vec`，你可以写：

```
use rand::seq::SliceRandom;
use rand::thread_rng;

my_vec.shuffle(&mut thread_rng());
```

16.2.10 Rust 排除了无效性错误

大多数主流编程语言都有集合和迭代器，而且都有这个规则的变体：不要在遍历集合的同时修改它。例如，Python 中 `vecotr` 等价的是 `list`：

```
my_list = [1, 3, 5, 7, 9]
```

假设我们想移除 `my_list` 中大于 4 的元素；

```
for index, val in enumerate(my_list):
    if val > 4:
        del my_list[index] # bug: 在迭代的同时修改

print(my_list)
```

(Python 中的 `enumerate` 函数等价于 Rust 中的 `.enumerate()` 方法，见 [enumerate](#)。)

这个程序令人惊讶地打印出 `[1, 3, 7]`。但 7 比 4 大，为什么它没被移除？这就是无效性错误：程序在迭代数据的同时修改它，将迭代器无效化 (*invalidate*)。在 Java 中，结果可能是一个异常；在 C++ 中是未定义行为。而在 Python 中，这个行为是有定义的，虽然不太直观：迭代器会跳过一个元素。`val` 永远不会是 7。

让我们尝试在 Rust 中复现这个 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); // error: can't borrow `my_vec` as mutable
        }
    }
    println!("{:?}", my_vec);
}
```

Rust 自然会在编译期拒绝这个程序。当我们调用 `my_vec.iter()` 时，它借用了 `vector` 的一个共享（非 `mut`）引用。引用的生命周期和迭代器一样长，也就是直到 `for` 循环的结尾。我们不能在有非 `mut` 引用存在时调用 `my_vec.remove(index)` 修改 `vector`。

编译器能指出这个错误非常好，但当然，你仍然需要一种方法来得到期望的行为！这里最简单的修复方法是：

```
my_vec.retain(|&val| val <= 4);
```

或者，你可以采用在 Python 或其他任何语言中也可以实现的方法：使用 `filter` 来创建一个新的 `vector`。

16.3 VecDeque<T>

Vec 只支持在尾部高效地添加和删除元素。当一个程序需要存储“排队”的值时，Vec 会变得很慢。

Rust 的集合 `std::collections::VecDeque<T>` 是一个双端队列 (*deque*) (读作“deck”)。它支持在头部和尾部高效地添加和移除元素：

`deque.push_front(value)`

在队列的头部添加一个值。

`deque.push_back(value)`

在尾部添加一个值。(这个方法比`.push_front()`用得更多，因为队列的通常用法是从尾部添加并从头部移除，就像人在排队一样。)

`deque.pop_front()`

移除并返回队列头部的值，返回 `Option<T>`，如果队列为空则返回 `None`，类似于 `vec.pop()`。

`deque.pop_back()`

移除并返回尾部的元素，同样返回 `Option<T>`。

`deque.front()`, `deque.back()`

类似于 `vec.first()` 和 `vec.last()`。它们返回队列中头部或者尾部元素的引用。返回类型是 `Option<&T>`，当队列为空时返回 `None`。

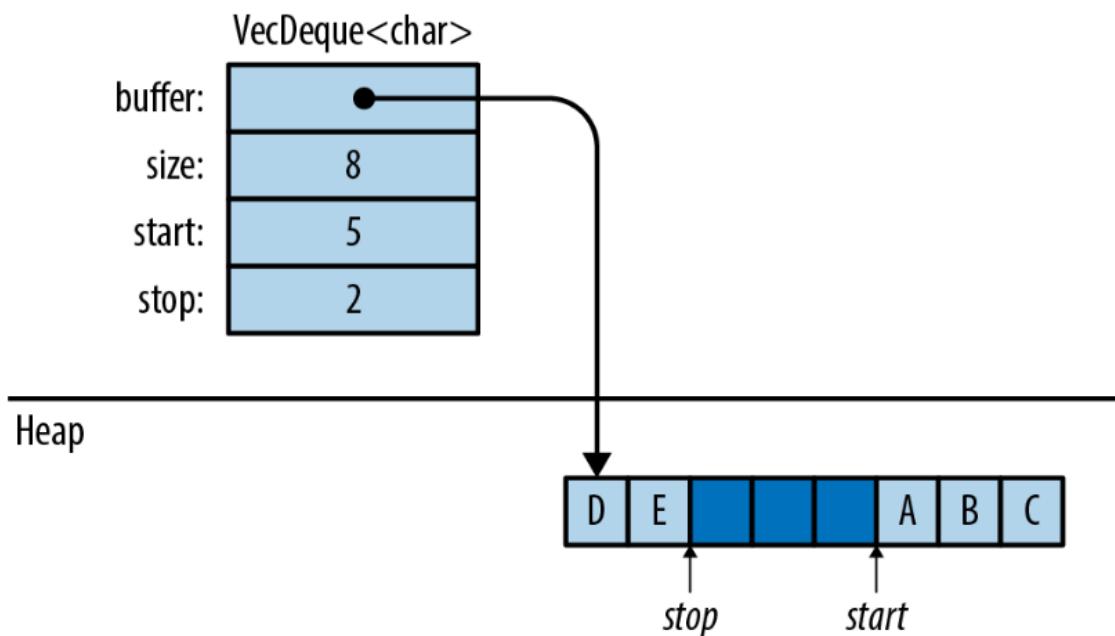
`deque.front_mut()`, `deque.back_mut()`

类似于 `vec.first_mut()` 和 `vec.last_mut()`，返回 `Option<&mut T>`。

`VecDeque` 的实现是一个环形缓冲区，如图 16-3 所示。

类似于 Vec，它有一个堆上分配的缓冲区用来存储元素。和 Vec 不同的是，数据并不总是从区域的起点开始存储，并且可以在结尾处“回环”。图中这个队列的元素，按顺序分别是 `['A', 'B', 'C', 'D', 'E']`。`VecDeque` 有私有的字段标记图中的 `start` 和 `stop`，用来记录缓冲区中数据的起点和终点。

向队列首部或者尾部添加值，意味着要占用一个未使用的位置（图中颜色较深的块），如果需要的话可能会回环或者分配更大的内存块。

图 16-3: `VecDeque` 在内存中如何存储

`VecDeque` 负责管理回环，因此你不需要考虑它。图 16-3 是 Rust 保证 `.pop_front()` 速度很快的幕后视图。

很多时候，当你需要双端队列的时候，你可能只需要 `.push_back()` 和 `.pop_front()` 两个方法。类型关联函数 `VecDeque::new()` 和 `VecDeque::with_capacity(n)` 用于创建队列，类似于 `Vec` 的相应函数。很多 `Vec` 的方法在 `VecDeque` 中都有实现：`.len()`, `.is_empty()`, `.insert(index, value)`, `.remove(index)`, `.extend(iterable)`, 等等。

双端队列和 `vector` 一样可以以值、以共享引用或者以 `mut` 引用迭代。它们都有三个迭代器方法 `.into_iter()`, `.iter()`, `.iter_mut()`。它可以用通常的方式索引：`deque[index]`。

因为双端队列在内存中并不是连续存储元素，因此它不能继承切片的方法。但如果你愿意承受移动元素的开销，`VecDeque` 提供了一个方法来修复它：

```
deque.make_contiguous()
```

以 `&mut self` 为参数，把 `VecDeque` 重新排布到连续的内存中，返回 `&mut [T]`。

`Vec` 和 `VecDeque` 高度相关，标准库提供了两个 trait 实现来轻松地互相转换：

```
Vec::from(deque)
```

`Vec<T>` 实现了 `From<VecDeque<T>>`，因此这会把一个双端队列变成一个 `vector`。这会消耗 $O(n)$ 时间，因为它可能要重新排布元素。

```
VecDeque::from(vec)
```

`VecDeque<T>`实现了`From<Vec<T>>`，因此这会把一个`vector`转换成一个双端队列。这也是 $O(n)$ 复杂度，但它通常会很快，即使`vector`很大，因为`vector`的堆缓冲区可以简单地移动到新的双端队列中。

它可以让我们将简单地用指定元素创建一个双端队列，即使没有标准的`vec_deque![]`宏：

```
use std::collections::VecDeque;

let v = VecDeque::from(vec![1, 2, 3, 4]);
```

16.4 BinaryHeap<T>

`BinaryHeap<T>`集合始终以某种形式组织元素，其中最大的元素总是会被移动到队列的首部。这里是`BinaryHeap`最常用的几个方法：

`heap.push(value)`

向堆中添加一个元素

`heap.pop()`

移除并返回堆中最大的值。它返回`Option<T>`，如果堆为空时返回`None`。

`heap.peek()`

返回堆中最大的值的引用。返回类型是`Option<&T>`。

`heap.peek_mut()`

返回一个`PeekMut<T>`，它可以用作堆中最大值的一个可变引用，并提供类型关联函数`pop()`来从堆中弹出这个值。使用这个方法，我们可以根据最大的元素的值来决定要不要从堆中弹出这个元素：

```
use std::collections::binary_heap::PeekMut;

if let Some(top) = heap.peek_mut() {
    if *top > 10 {
        PeekMut::pop(top);
    }
}
```

BinaryHeap 支持 Vec 的方法的一个子集，包括 BinaryHeap::new()，.len()，.is_empty()，.capacity()，.clear()，.append(&mut heap2)。

例如，假设我们用一些数字填充一个 BinaryHeap：

```
use std::collections::BinaryHeap;

let mut heap = BinaryHeap::from(vec![2, 3, 8, 6, 9, 5, 4]);
```

值 9 在堆的顶部：

```
assert_eq!(heap.peek(), Some(&9));
assert_eq!(heap.pop(), Some(9));
```

移除 9 也会重新排布其他元素，把 8 移动到头部，等等：

```
assert_eq!(heap.pop(), Some(8));
assert_eq!(heap.pop(), Some(6));
assert_eq!(heap.pop(), Some(5));
...
```

当然，BinaryHeap 并不仅限于数字。它可以包含任何实现了内建的 Ord trait 的类型。

这让 BinaryHeap 可以用作一个工作队列。你可以定义一个任务结构体，然后根据任务的优先级实现 Ord，让高优先级的任务大于低优先级的任务。然后，创建一个 BinaryHeap 来保存所有待办的任务。它的.pop() 方法将总是返回最重要的任务。

注意：BinaryHeap 是可迭代的对象，并且它有 .iter() 方法，但这个迭代器以任意顺序产生堆中的元素，而不是按照从大到小的顺序。为了按照大小顺序消耗 BinaryHeap 中的值，可以使用 while 循环：

```
while let Some(task) = heap.pop() {
    handle(task);
}
```

16.5 HashMap<K, V> 和 BTreeMap<K, V>

map(映射)是键值对(称为条目(entry))的集合。任何两个条目的键都不同，所有的条目按照一定结构组织，如果有一个键就可以高效地在 map 中查找到相应的值。简而言之，map 是一个查找表。

Rust 提供两者两种 map 类型：HashMap<K, V> 和 BTreeMap<K, V>。这两种类型共享了很多相同的方法；不同之处在于它们组织条目的方式。

HashMap 把键和值都存储在哈希表中，因此它要求键的类型 K 实现了 Hash 和 Eq，这两个 trait 分别用于哈希和相等性比较。

图 16-4 展示了 HashMap 如何在内存中排布。深色区域表示没有使用。所有的键、值和缓存的哈希值都被存储在单个堆上分配的表中。添加条目最终会迫使 HashMap 分配更大的表，并把所有数据移动进去。

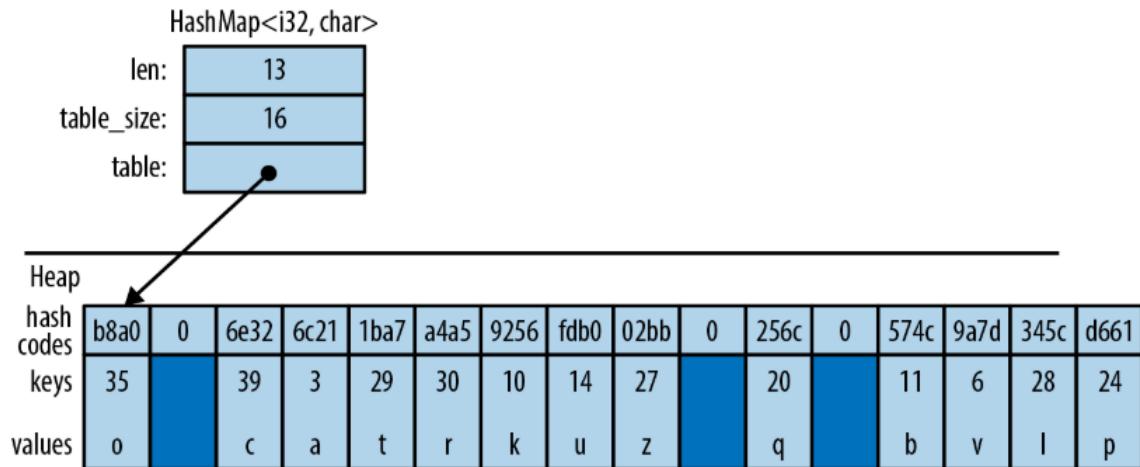


图 16-4: HashMap 的内存布局

BTreeMap 按照键的顺序在树形结构中存储条目，因此它要求键的类型 K 实现了 Ord。图 16-5 展示了一个 BTreeMap。同样，深色区域表示没有被使用的空间。

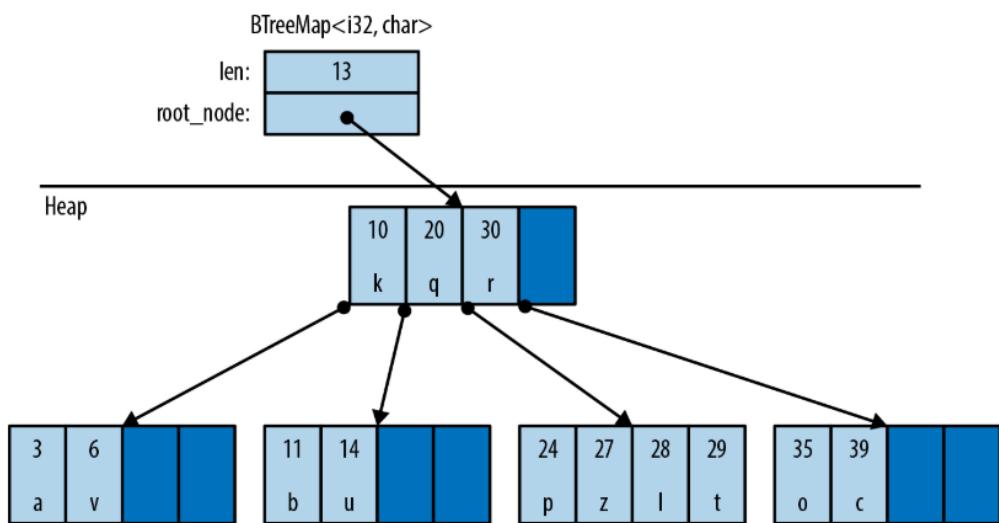


图 16-5: BTreeMap 的内存布局

BTreeMap 把条目存储在节点 (*node*) 中。一个 BTreeMap 中大多数的节点都只包含键值对。而非叶节点，例如图中所示的根节点，还需要存储指向子节点的指针。`(20, 'q')` 和 `(30, 'r')` 之间的指针指向包含 `20` 和 `30` 之间的键的子节点。添加条目通常需要把某个节点的部分

现有条目向右移动，来保持它们有序，有时还需要分配新的节点。

图中的示例经过简化来适应页面。例如，实际的 BTreeMap 节点有 11 个条目的空间，而不是 4 个。

Rust 标准库使用 B 树而不是二叉平衡树，因为在现代硬件上 B 树更快。查找时二叉树可能比 B 树的比较次数更少，但在 B 树中查找有更好的局部性 (*locality*)——即，内存被成组访问，而不是扫描整个堆。这让 CPU 缓存的命中率更高，这能显著地提高速度。

这里有几种创建 map 的方法：

`HashMap::new()`, `BTreeMap::new()`

创建新的空 map。

`iter.collect()`

从键值对创建并填充新的 `HashMap` 或 `BTreeMap`。`iter` 必须是一个 `Iterator<Item=(K, V)>`。

`HashMap::with_capacity(n)`

创建一个新的空哈希表，并分配至少能存储 `n` 个条目的空间。`HashMap` 和 `vector` 一样把数据存储在单个堆上的内存中，因此它也有容量以及相关的方法 `hash_map.capacity()`, `hash_map.reserve(additional)`, `hash_map.shrink_to_fit()`。`BTreeMap` 则没有。

`HashMap` 和 `BTreeMap` 有相同的处理键和值的核心方法：

`map.len()`

返回条目的数量。

`map.is_empty()`

返回 `map` 里是否没有条目

`map.contains_key(&key)`

如果 `map` 里有给定 `key` 的条目则返回 `true`。

`map.get(&key)`

在 `map` 中查找给定 `key` 的条目。如果找到了匹配的条目，就返回 `Some(r)`，其中 `r` 是相应的值的引用。否则返回 `None`。

```
map.get_mut(&key)
```

与上面类似，但返回值的 `mut` 引用。

一般来讲，`map` 让你可以获取值的 `mut` 访问能力，但不能获取键的 `mut` 访问。值属于你，你可以随意修改它。但键术语 `map` 本身，它需要保证键不会改变，因为条目按照键来组织。原地修改键将会导致 bug。

```
map.insert(key, value)
```

向 `map` 中插入条目 `(key, value)`，并返回旧的值（如果有的话）。返回类型是 `Option<V>`，如果 `map` 中已经有了 `key` 的条目，那么新插入的 `value` 会覆盖旧值。

```
map.extend(iterable)
```

迭代 `iterable` 中的 `(K, V)` item，并把每一个键值对插入 `map`。

```
map.append(&mut map2)
```

把 `map2` 中的所有条目移动到 `map` 中。完成之后，`map2` 变为空。

```
map.remove(&key)
```

查找并移除 `map` 中给定的 `key` 的条目，返回被移除的值（如果有的话）。返回类型是 `Option<V>`。

```
map.remove_entry(&key)
```

查找并移除 `map` 中给定的 `key` 的条目，返回被移除的键和值（如果有的话）。返回类型是 `Option<(K, V)>`。

```
map.retain(test)
```

移除所有未通过测试的元素。参数 `test` 参数是一个实现了 `FnMut(&K, &mut V) -> bool` 的函数或者闭包。它会对 `map` 中的每一个元素调用 `test(&key, &mut value)`，如果返回 `false`，就移除掉这个元素并丢弃。

不考虑性能的话，这类似于如下写法：

```
map = map.into_iter().filter(test).collect();
```

```
map.clear()
```

移除所有元素。

map 可以使用方括号 `map[&key]` 进行查询。这是因为 map 实现了内建的 `Index` trait。然而，如果没有给定的 key 的条目存在，这会 panic，就类似越界访问数组一样。因此只有当你确定要查找的条目在 map 中时再使用这个语法。

`.contains_key()`, `.get()`, `.get_mut()`, `.remove()` 方法的 key 参数的类型不一定要是精确的 &K。这些方法都是泛型方法，只要能从 K 类型借用参数的类型的引用即可。假设 `fish_map` 是一个 `HashMap<String, Fish>`, 那么即使 "conger" 并不是 `String`, 我们也可以调用 `fish_map.contains_key("conger")`。因为 `String` 实现了 `Borrow<&str>`, 所以可以从 `String` 借用一个 `&str`。详情见 [Borrow 与 BorrowMut](#)。

因为一个 `BTreeMap<K, V>` 按照键的顺序保存条目，所以它支持一个附加的操作：

```
btree_map.split_off(&key)
```

把 `btree_map` 分割成两个。键小于 `key` 的条目被留在 `btree_map` 中，返回一个包含其余条目的新 `BTreeMap<K, V>`。

16.5.1 条目

`HashMap` 和 `BTreeMap` 都有相应的 `Entry` 类型。这个类型的意义是避免重复的查找。例如，这里有一些代码获取或者创建一个学生的记录：

```
// 我们已经有了这名学生的记录了吗？
if !student_map.contains_key(name) {
    // 没有： 创建一条记录
    student_map.insert(name.to_string(), Student::new());
}

// 现在确定这条记录肯定存在了。
let record = student_map.get_mut(name).unwrap();
...
```

这可以正常工作，但它访问了 `student_map` 两次或者三次，每次都做相同的查找。

条目的思路是我们只查找一次，产生一个 `Entry` 值，然后后续的操作都通过它进行。下面的单行代码等于上面的所有代码，除了它只查找一次：

```
let record = student_map.entry(name.to_string()).or_insert_with(Student::new);
```

`student_map.entry(name.to_string())` 返回的 `Entry` 值就像一个可变引用，它指向 map 中一个已经被键值对占据 (*occupied*) 的位置，或者是空的 (*vacant*)，意思是还没有条目占据这个位置。如果为空，条目的 `.or_insert_with()` 方法会插入一个新的 `Student`。条目的大多数使用都类似这样：简短而方便。

所有的 Entry 值都只能用同一个方法创建：

```
map.entry(key)
```

对给定的 key 返回一个 Entry。如果 map 中没有这个 key，它会返回一个空的 Entry。

这个方法以 mut 引用获取 self 参数，并返回一个生命周期相同的 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，因为它是 map 的一种 mut 借用。只要 Entry 存在，它就有 map 的独占访问权限。

回顾包含引用的结构体，我们看到过如何在一个类型中存储引用以及这样对生命周期的影响。现在我们从用户的视角看看它是什么样的。也正是 Entry 的情况。

不幸的是，如果 map 的键的类型为 String，那么不能向这个方法传递 &str 类型的参数。这种情况下 .entry() 方法需要一个真实的 String。

Entry 值提供了三个处理空条目的方法：

```
map.entry(key).or_insert(value)
```

确保 map 包含给定的 key 的条目，如果需要的话用给定的 value 插入一个新的条目。它返回新插入的或者现有的值的 mut 引用。

假设我们需要计数投票。我们可以写：

```
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() 返回一个可变引用，因此 count 的类型是 &mut usize。

```
map.entry(key).or_default()
```

确保 map 包含给定的 key 的条目，如果需要的话用 Default::default() 返回的值插入一个新条目。只有当值的类型实现了 Default 时这个方法才能工作。类似于 or_insert，这个方法返回新插入的或者现有的值的 mut 引用。

```
map.entry(key).or_insert_with(default_fn)
```

这个方法也一样，除了当它需要创建新的条目时，它会调用 default_fn() 来产生默认值。如果 map 中已经有了 key 的条目，那么 default_fn 将不会被调用。

假设我们想知道哪个单词在哪个文件中出现。我们可以写：

```
// 这个 map 中包含每个单词和出现它的文件的集合。
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 还提供了一个只修改现存条目的便捷方法。

```
map.entry(key).and_modify(closure)
```

如果给定的 key 的条目存在就调用 closure，把值的可变引用传进闭包。它返回一个 Entry，因此它可以和其它方法链式调用。

例如，我们可以使用它来统计一个字符串中每个单词出现的次数：

```
// 这个 map 包含给定字符串中的所有单词,
// 以及它们出现的次数。
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 类型是一个枚举，HashMap 的 Entry 定义类似于这样（BTreeMap 的 Entry 类似）：

```
// (in std::collections::hash_map)
pub enum Entry<'a, K, V> {
    Occupied(OccupiedEntry<'a, K, V>),
    Vacant(VacantEntry<'a, K, V>)
}
```

OccupiedEntry 和 VacantEntry 类型的方法用于在不需要重复查找的情况下插入、移除和访问条目。你可以在在线文档中找到它们。偶尔可以使用这些附加的方法来减少一次或两次查找，但 .or_insert() 和 .or_insert_with() 就能覆盖大多数的情况。

16.5.2 迭代 map

有几种迭代 map 的方法：

1. 以值迭代 (`for (k, v) in map`)，产生 (K, V) 对。这会消耗掉 map。
2. 迭代共享引用 (`for (k, v) in &map`)，产生 (&K, &V) 对。
3. 迭代可变引用 (`for (k, v) in &mut`)，产生 (&K, &mut V) 对。（再提醒一次，没有获取 map 中键的 mut 访问的方法，因为条目是按照键来组织的。）

类似于 vector，map 有 `.iter()` 和 `.iter_mut()` 方法返回以引用迭代的迭代器，就类似于迭代 `&map` 或者 `&mut map`。另外：

`map.keys()`

返回一个只迭代键的迭代器，以引用的形式返回。

`map.values()`

返回一个只迭代值的迭代器，以引用的形式返回。

`map.values_mut()`

返回一个只迭代值的迭代器，以 mut 引用的形式返回。

所有的 HashMap 迭代器都会以任意顺序访问 map 的条目。BTreeMap 的迭代器会按照键的顺序访问它们。

16.6 HashSet<T> 和 BTreeSet<T>

set 是值的集合，它可以快速地测试元素：

```
let b1 = large_vector.contains(&"needle");      // 很慢，检查每一个元素
let b2 = large_hash_set.contains(&"needle");      // 很快，哈希查找
```

一个 set 绝不会包含同一个值的多个拷贝。

map 和 set 有不同的方法，但其实一个 set 就是一个只有键而不是键值对的 map。事实上，Rust 的 HashSet<T> 和 BTreeSet<T> 被实现为 HashMap<T, ()> 和 BTreeMap<T, ()> 的包装。

`HashSet::new()`, `BTreeSet::new()`

创建新的 set。

`iter.collect()`

可以用于从任何迭代器创建 set。如果 `iter` 产生某个值不止一次，那么重复的值将会被丢弃。

```
HashSet::with_capacity(n)
```

创建一个空的 HashSet，有至少能存储 n 个元素的空间。

HashSet<T> 和 BTreeSet<T> 有相同的公共基础方法：

```
set.len()
```

返回 set 中值的数量。

```
set.is_empty()
```

如果 set 不包含任何元素则返回 true。

```
set.contains(&value)
```

如果 set 包含给定的 value 则返回 true。

```
set.insert(value)
```

向 set 中添加一个 value。如果添加了值则返回 true，如果 set 中已经有这个值了则返回 false。

```
set.remove(&value)
```

从 set 中删除 value。如果有值被删除则返回 true，如果没有这个值则返回 false。

```
set.retain(test)
```

删除没有通过测试的元素。test 参数是一个实现了 FnMut(&T) -> bool 的函数或者闭包。对于 set 的每一个元素，都会调用 test(&value)，如果返回 false，这个元素就会从 set 中删除，并且被丢弃。

不考虑性能的话，这类似于：

```
set = set.into_iter().filter(test).collect();
```

和 map 一样，通过引用查找值的方法接受任何可以从 T 借用的类型。细节见 [Borrow](#) 与 [BorrowMut](#)。

16.6.1 迭代 set

有两种迭代 set 的方法:

1. 以值迭代 (`for v in set`) 产生 set 的成员 (并消耗这个 set)。
 2. 以共享引用迭代 (`for v in &set`) 产生 set 中成员的共享引用。
- 不支持以 `mut` 引用迭代 set。没有方法获取 set 中值的 `mut` 引用。

`set.iter()`

返回一个以共享引用方式迭代 set 的迭代器。

`HashSet` 迭代器类似于 `HashMap` 的迭代器，也会以任意顺序产生值。`BTreeset` 迭代器按照顺序产生值，类似于一个排序过的 `vector`。

16.6.2 当相等的值不同时

set 还有一些方法，只有当你关心“相等”值的差异时才会用到。

这样的差异经常出现。例如，两个完全相同的 `String` 值，在内存中的不同位置存储它们的字符：

```
let s1 = "hello".to_string();
let s2 = "hello".to_string();
println!("{:p}", &s1 as &str); // 0x7f8b32060008
println!("{:p}", &s2 as &str); // 0x7f8b32060010
```

通常我们并不会关心这些。

但在你需要的情况下，你可以使用下面的方法获得 set 中存储的值。这些方法都返回一个 `Option`，当 set 不包含匹配的值时为 `None`:

`set.get(&value)`

返回 set 中等于 value 的成员的共享引用，如果有的话。返回 `Option<&T>`。

`set.take(&value)`

类似于 `set.remove(&value)`，但它会返回被删除的值，如果有的话。返回 `Option<T>`。

`set.replace(value)`

类似于 `set.insert(value)`，但如果 set 已经包含一个等于 value 的值，它会替换并返回旧的值。返回 `Option<T>`。

16.6.3 集合操作

到目前为止，我们看到的 set 方法都只关注单个 set 中的单个值。set 还有一些操作整个集合的方法：

```
set1.intersection(&set2)
```

返回一个迭代器，迭代所有既在 set1 又在 set2 中的值。

例如，如果我们想打印出所有同时选择了脑外科和火箭课程的学生的名字，我们可以写：

```
for student in &brain_class {
    if rocket_class.contains(student) {
        println!("{}", student);
    }
}
```

或者，更短的写法：

```
for student in brain_class.intersection(&rocket_class) {
    println!("{}", student);
}
```

令人惊奇的是，有专门的运算符来进行这种操作。

`&set1 & &set2` 返回一个新的 set，它是 set1 和 set2 的交集。这是一元的按位 AND 运算符，用于两个引用。它会找到既在 set1 中又在 set2 中的值：

```
let overachievers = &brain_class & &rocket_class;
```

```
set1.union(&set2)
```

返回一个迭代器，迭代要么在 set1 中要么在 set2 中的值。

`&set1 | &set2` 返回一个新的 set，包含所有在 set1 或 set2 中的值。

```
set1.difference(&set2)
```

返回一个迭代器，迭代在 set1 中但不在 set2 中的值。

`&set1 - &set2` 返回一个新的 set，包含所有这样的值。

```
set1.symmetric_difference(&set2)
```

返回一个迭代器，迭代 set1 或 set2 独有的值。

`&set1 ^ &set2` 返回一个新的 set，包含所有这样的值。

还有三个方法用来测试两个 set 的关系：

```
set1.is_disjoint(set2)
```

如果 `set1` 和 `set2` 没有相同的元素则返回 `true`——即它们的交集为空。

```
set1.is_subset(set2)
```

如果 `set1` 是 `set2` 的子集则返回 `true`——即 `set1` 中的所有值也都在 `set2`。

```
set1.is_superset(set2)
```

和上面相反，如果 `set1` 是 `set2` 的超集则返回 `true`。

`set` 也支持使用 `==` 和 `!=` 进行相等性测试；如果两个 `set` 包含相同的值则它们相等。

16.7 哈希

`std::hash::Hash` 是标准库用于可哈希类型的 trait。`HashMap` 的键和 `HashSet` 的元素必须实现了 `Hash` 和 `Eq`。

大多数实现了 `Eq` 的内建类型也都实现了 `Hash`。整数类型、`char`、`String` 都是可哈希类型；当元素是可哈希类型时，元组、数组、切片、`vector` 也是可哈希类型。

标准库的一个原则是不管你把一个值存储到哪里或者如何指向它，它必须有相同的哈希值。因此，引用和被引用的值有相同的哈希值。`Box` 和被装箱的值有相同的哈希值。一个 `vector` `vec` 和包含它的所有元素的切片 `&vec[..]` 有相同的哈希值。一个 `String` 和一个有相同字符的 `&str` 有相同的哈希值。

结构和枚举默认没有实现 `Hash`，但可以派生实现：

```
/// 大英博物馆藏品中的对象的 ID
#[derive(Clone, PartialEq, Eq, Hash)]
enum MuseumNumber {
    ...
}
```

只要所有的字段都是可哈希的就可以正常工作。

如果你手动为一个类型实现了 `PartialEq`，那你也应该手动实现 `Hash`。例如，假设我们有一个类型表示无价的文物：

```
struct Artifact {
    id: MuseumNumber,
    name: String,
    cultures: Vec<Culture>,
    date: RoughTime,
```

```
    ...  
}
```

并且有相同 ID 的两个 `Artifact` 被认为相等：

```
impl PartialEq for Artifact {  
    fn eq(&self, other: &Artifact) -> bool {  
        self.id == other.id  
    }  
}  
  
impl Eq for Artifact {}
```

因为我们比较两个文物时只根据 ID 进行比较，因此我们必须用相同的方式哈希它们：

```
use std::hash::{Hash, Hasher};  
  
impl Hash for Artifact {  
    fn hash<H: Hasher>(&self, hasher: &mut H) {  
        // 把哈希操作委托给 MuseumNumber  
        self.id.hash(hasher);  
    }  
}
```

(否则，`HashSet<Artifact>` 将不能合适地工作；和所有哈希表一样，它要求如果 `a == b` 那么 `hash(a) == hash(b)`。)

这样我们就可以创建一个 `Artifact` 的 `HashSet`：

```
let mut collection = HashSet::new();
```

如上面的代码所示，即使你手动实现了 `Hash`，你也不需要知道有关哈希算法的信息。`.hash()` 接受一个 `Hasher` 的引用，后者代表哈希算法。你只需要把所有和 `==` 运算符相关的数据喂给 `Hasher`。`Hasher` 会从你给的内容计算出一个哈希值。

16.8 使用一个自定义的哈希算法

`hash` 算法是泛型的，因此上面的 `Hash` 实现可以把数据喂给任何实现了 `Hasher` 的类型。这是 Rust 支持可插拔哈希算法的方式。

第三个 trait, `std::hash::BuildHasher`, 用于表示一个哈希算法的初始状态。每个 `Hasher` 只能使用一次，类似于一个迭代器：你只能使用它一次，然后就丢弃掉它。`BuildHasher` 可以重用。

每一个 `HashMap` 都包含一个 `BuildHasher`，每当需要计算一个哈希值都会使用它。`BuildHasher` 值包含哈希算法每次运行时需要的密钥、初始状态或其他参数。

计算一个哈希值的完整流程看起来像这样：

```
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. 开始算法
    value.hash(&mut hasher); // 2. 喂数据
    hasher.finish() // 3. 结束，产生一个 u64
}
```

`HashMap` 每次需要计算哈希值时都会调用这三个方法。所有的方法都是内联的，因此它的速度很快。

Rust 的默认哈希算法是知名的 SipHash-1-3 算法。SipHash 很快，并且擅长最小化哈希冲突。事实上，它是一种密码学安全算法：没有已知的有效方法来产生 SipHash-1-3 冲突。只要哈希表使用不同、不可预知的键，Rust 可以抵御一种称为 HashDos 的拒绝服务攻击，这种攻击中攻击者故意利用哈希冲突来触发服务器性能最差的情况。

但你自己的应用可能不需要这样。如果你正在存储很多很小的键，例如整数或很短的字符串，那么你可以牺牲 HashDos 安全性来实现更快的哈希函数。`fnv` crate 实现了这样一个算法，Fowler-Noll-Vo(FNV) 哈希。在 `Cargo.toml` 中加上一行来尝试它：

```
[dependencies]
fnv = "1.0"
```

然后从 `fnv` 中导入 `map` 和 `set` 类型：

```
use fnv::{FnvHashMap, FnvHashSet};
```

你可以用这两种类型作为 `HashMap` 和 `HashSet` 的替代。看一眼 `fnv` 的源码就知道它们是如何定义的：

```
/// 一个使用默认 FNV hasher 的 `HashMap`
pub type FnvHashMap<K, V> = HashMap<K, V, FnvBuildHasher>

/// 一个使用默认 FNV hasher 的 `HashSet`
pub type FnvHashSet<T> = HashSet<T, FnvBuildHasher>;
```

标准的 `HashMap` 和 `HashSet` 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，它将最后一个参数指定为 FNV hasher。

16.9 标准集合之外

在 Rust 中创建一个新的自定义集合类型和其他语言中差不多。通过组合语言提供的各个部分来排列数据：结构体和枚举、标准集合、`Option`、`Box` 等等。例如，[泛型枚举](#)中定义了`BinaryTree<T>`类型。

如果你习惯在 C++ 中使用原始指针、手动内存管理、placement `new`、和手动析构调用来实现数据结构以获得最佳的性能，那么毫无疑问你会遇到安全 Rust 中的很多限制。所有这些工具都是天然不安全的。它们在 Rust 中也是可行的，但只能用于 `unsafe` 代码。[第 22 章](#)展示了为什么，并且还包含一个使用 `unsafe` 代码实现 safe 的自定义集合的示例。

到目前为止，我们都沉浸在标准集合及其安全、高效的 API 的温暖光芒中。和 Rust 标准库中很多其他部分一样，它们设计的目的之一是确保 `unsafe` 代码的需求尽可能减少。

Chapter 17

字符串与文本

The string is a stark data structure and everywhere it is passed there is much duplication of process. It is a perfect vehicle for hiding information.

——Alan Perlis, epigram #34

我们已经使用过了 Rust 中的主要文本类型：`String`、`str` 和 `char`。在字符串类型中，我们介绍了字符和字符串字面量的语法并且展示了字符串在内存中如何表示。在本章中，我们将更加详细地介绍文本。

在本章中：

1. 我们会提供一些 Unicode 的背景知识，帮助你更好地理解标准库的设计。
2. 我们会介绍 `char` 类型，它表示单个 Unicode 码点。
3. 我们会介绍 `String` 和 `str` 类型，它们表示有所有权的和借用的 Unicode 字符序列。它们有非常多的方法用于构建、搜索、修改、迭代它们的内容。
4. 我们会介绍 Rust 的字符串格式化设施，例如 `println!` 和 `format!` 宏。你可以编写自己的用于字符串格式化的宏，以及扩展它们来支持你自定义的类型。
5. 我们会给出 Rust 中正则表达式支持的一个概述。
6. 最后我们会讨论为什么 Unicode 规范化很重要，并展示如何在 Rust 中实现它。

17.1 Unicode 背景知识

本书是关于 Rust 的，而不是关于 Unicode 的，事实上已经有整本专门介绍它的书了。但 Rust 的字符和字符串类型被设计为 Unicode。这里有一些有助于理解 Rust 的 Unicode 知识。

17.1.1 ASCII, Latin-1, Unicode

在所有 ASCII 码点范围（0 到 0x7f）内 Unicode 和 ASCII 完全相同：例如，这两种编码中字符 * 都是码点 42。

17.1.2 UTF-8

17.2 字符 (char)

17.3 String 和 str

17.3.1 创建 String 值

17.3.2 简单的视图

17.3.3 附加和插入文本

17.4 格式化

17.5 正则表达式

17.5.1 基本正则使用

17.5.2 惰性构建正则值

Chapter 18

输入输出

Doolittle: *What concrete evidence do you have that you exist?*

Bomb #20: *Hmmmm... well... I think, therefore I am.*

Doolittle: *That's good. That's very good. But how do you know that anything else exists?*

Bomb #20: *My sensory apparatus reveals it to me.*

——Dark Star

Rust 中有关输入输出的特性围绕着三个 trait: `Read`、`BufRead`、`Write` 来组织:

- 实现了 `Read` 的值有读取字节输入的方法。它们被称为读者 (*reader*)。
- 实现了 `BufRead` 的值是 *buffered reader* (有缓存的读者)。它们支持 `Read` 的所有方法，加上读取文本的一行的方法，等等。
- 实现了 `Write` 的值支持字节和 UTF-8 文本输出。它们被称为写者 (*writer*)。

图 18-1 展示了这三个 trait 以及一些 reader 和 writer 类型的示例。

在本章中,我们将解释如何使用这些 trait 和它们的方法,包括图中出现的 reader 和 writer 类型,还有一些其他的和文件、终端、网络交互的方法。

18.1 Reader 和 Writer

Reader 是你的程序可以从中读取字节的值。例如:

- 使用 `std::fs::File::open(filename)` 打开的文件
- 用于从网络中接收数据的 `std::net::TcpStream`
- 进程用来读取标准输入的 `std::io::stdin()`
- `std::io::Cursor<&[u8]>` 和 `std::io::Cursor<Vec<u8>>` 值, 它们是从内存中的字节数组或 `vector` 中“读取”数据的 reader

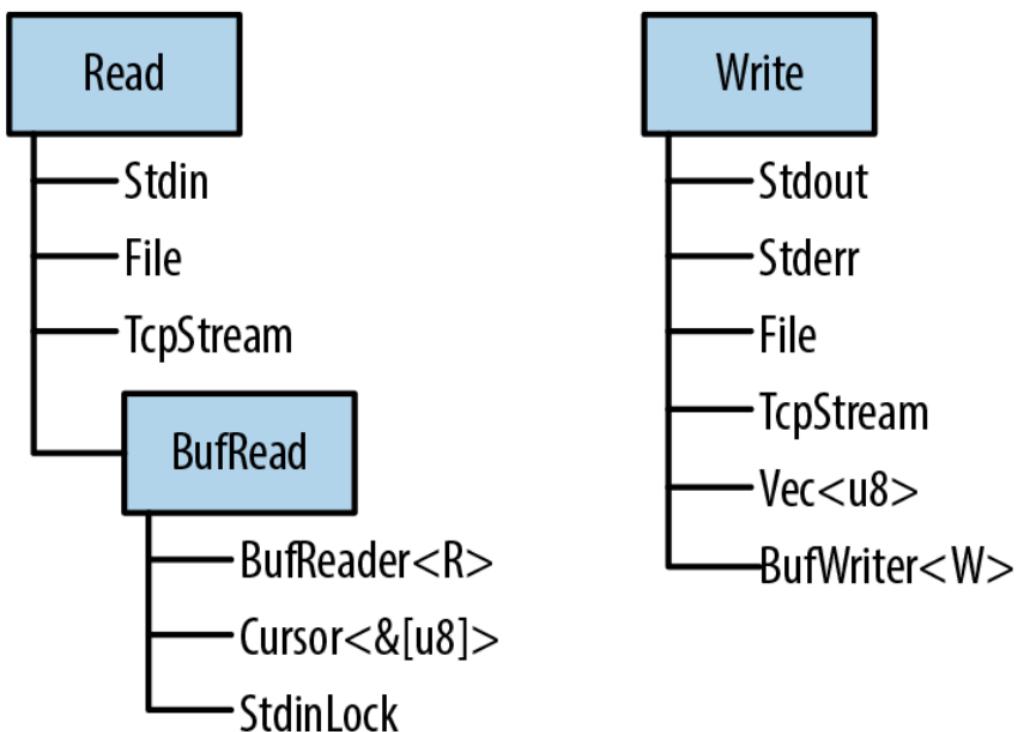


图 18-1: Rust 的三个主要的 I/O trait 以及一些实现了它们的类型

Writer 是你的程序可以向其中写入字节的值。例如：

1. 使用 `std::fs::File::create(filename)` 打开的文件
2. 用于向网络中发送数据的 `std::net::TcpStream`
3. 用于写入到终端的 `std::io::stdout()` 和 `std::io::stderr()`
4. `Vec<u8>`, 它也是一个 writer, 它的 `write` 方法把数据附加到尾部
5. `std::io::Cursor<Vec<u8>>`, 类似于上面, 但允许你同时读取和写入数据, 并可以在 vector 中定位到不同位置
6. `std::io::Cursor<&mut [u8]>`, 和 `std::io::Cursor<Vec<u8>>` 很像, 除了它不能让缓冲区增长, 因为它只是已经存在的字节数组的切片

因为有为 reader 和 writer 设计的标准 trait (`std::io::Read` 和 `std::io::Write`)，所以编写可以处理多种输入输出通道的泛型代码是非常普遍的。例如，这里有一个函数拷贝任意 reader 中的所有字节到任意 writer：

```

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的标准库中的`std::io::copy()`的实现。因为它是泛型的，你可以使用它从`File`中读取数据然后写入到`TcpStream`，或者从`Stdin`读取，然后写入到内存中的`Vec<u8>`，等等。

如果你看不明白这里的错误处理代码，请复习[第7章](#)。我们将在接下来的内容中一直使用`Result`类型，掌握它的工作原理很重要。

这三个`std::io`的trait：`Read`、`BufRead`、`Write`，以及`Seek`如此常用，以至于有一个只包含这些trait的`prelude`模块：

```
use std::io::prelude::*;


```

本章中你还会见到它一到两次。我们通常也习惯导入`std::io`模块自身：

```
use std::io::{self, Read, Write, ErrorKind};


```

这里的`self`关键字声明了`io`作为`std::io`模块的一个别名。这样，`std::io::Result`和`std::io::Error`可以用`io::Result`和`io::Error`更简洁地表示出来，等等。

18.1.1 Reader

`std::io::Read`有几个方法用于读取数据。所有这些方法都通过`mut`引用获取`self`参数。

`reader.read(&mut buffer)`

从数据源读取一些字节，并存储到给定的`buffer`中。`buffer`参数的类型是`&mut [u8]`。它最多读取`buffer.len()`个字节。返回类型是`io::Result<u64>`，它是`Result<u64>`，

`io::Error` 的类型别名。当成功时，`u64` 值是读取到的字节数，它可能等于或者小于 `buffer.len()`，即使还有更多的数据可以读取。`Ok(0)` 意味着没有更多的输入可以读取。

当出错时，`.read()` 返回 `Err(err)`，其中 `err` 是一个 `io::Error` 值。为了便于人类阅读，`io::Error` 是可打印的；而对于程序，它有一个 `.kind()` 方法返回一个 `io::ErrorKind` 类型的错误码。这个枚举的成员有例如 `PermissionDenied` 和 `ConnectionReset`。大多数的错误都不能被忽略，但有一种错误应该进行特殊处理。`io::ErrorKind::Interrupted` 对应 Unix 的错误码 `EINTR`，它意味着读取过程恰好被一个信号打断。除非你的程序想设计为根据信号做一些聪明的操作，否则它应该简单地重试读取操作。上一节中的 `copy()` 的代码，就是一个例子。

如你所见，`.read()` 方法非常底层，甚至直接继承了底层操作系统的怪癖。如果你要为一个新的数据源类型实现 `Read` trait，这会赋予你极大的灵活性。但如果你尝试读取一些数据，就会非常难受。因此，Rust 提供了几个更高级的便捷方法。它们都有基于 `.read()` 的默认实现。它们都处理了 `ErrorKind::Interrupted`，因此你不需要再处理。

```
reader.read_to_end(&mut byte_vec)
```

读取 `reader` 中剩余的所有输入，将读到的数据附加到 `byte_vec` 尾部，`byte_vec` 是一个 `Vec<u8>`。返回一个 `io::Result<uszie>`，表示读取到的字节数。

这个方法读取的数据的大小没有限制，因此不要将它用于不受信任的源。（你可以使用 `.take()` 方法施加限制，如后文所述。）

```
reader.read_to_string(&mut string)
```

和上面相同，不过把数据附加到给定的 `String`。如果流不是有效的 UTF-8，它会返回一个 `ErrorKind::InvalidData` 错误。

在一些编程语言中，字节输入和字符输入由不同的类型来处理。如今，UTF-8 占据主导地位，Rust 承认这一事实标准，并且完全支持 UTF-8。其他字符集由开源的 `encoding` crate 提供支持。

```
reader.read_exact(&mut buf)
```

读取恰好足以填满给定缓冲区的数据。参数的类型是 `&mut [u8]`，如果在读取够 `buf.len()` 个字节之前 `reader` 的数据就已经耗光，那么会返回一个 `ErrorKind::UnexpectedEof` 错误。

上面这些是 Read trait 的主要方法。除此之外，还有三个以值获取 reader 的适配器方法，将它转换为一个迭代器或者一个不同的 reader：

```
reader.bytes()
```

返回一个输入流的字节的迭代器。item 的类型是 `io::Result<u8>`，因此每一个字节都需要进行错误检查。另外，它会逐字节调用 `reader.read()`，因此如果 reader 没有缓存的话会非常低效。

```
reader.chain(reader2)
```

返回一个新的 reader，首先产生 reader 的所有输入，然后产生 reader2 的所有输入。

```
reader.take(n)
```

返回一个新的 reader，从和 reader 相同的数据源读取数据，但最多只读取 n 个字节。
没有关闭 reader 的方法。reader 和 writer 通常实现了 Drop，因此它们会自动关闭。

18.1.2 有缓冲的 Reader

出于性能考虑，reader 和 writer 可以进行缓存 (buffer)，意思是它们有一块内存（缓冲区）用来存储一些输入或输出数据。这样可以减少系统调用的次数，如图 18-2 所示。在这个例子中，应用调用 `.read_line()` 方法从 `BufReader` 中读取数据，`BufReader` 从操作系统获取更大块的输入。

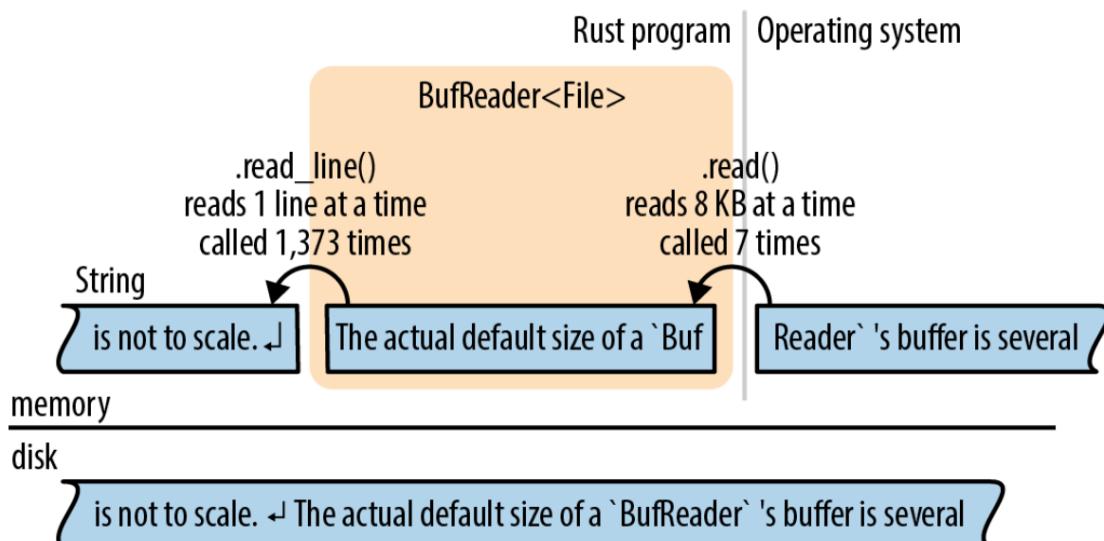


图 18-2: 一个有缓冲的文件 reader

这张图并不是按比例的，一个 `BufReader` 的实际大小是几千字节，因此一次系统的 `read` 调用可以提供上百次 `.read_line()` 调用。这么做之所以能提高性能是因为系统调用很慢。（如图所示，操作系统也有一个缓冲区，原因与此相同：系统调用很慢，但从磁盘读取数据更慢。）

有缓冲的 reader 实现了 `Read` 和另一个 trait `BufRead`，它添加了下面的方法：

```
reader.read_line(&mut line)
```

读取一行文本并将它附加到 `line`，`line` 是一个 `String`。行尾的换行符 '`\n`' 也会包含在 `line` 中。如果输入中有 Windows 风格的换行符 "`\r\n`"，这两个字符都会包含进 `line`。返回值是一个 `io::Result<usize>`，代表读取到的字节数，包括行尾的换行符。如果 `reader` 到达输入结尾，`line` 会保持不变，并返回 `Ok(0)`。

```
reader.lines()
```

返回一个迭代输入中每一行的迭代器。`item` 的类型是 `io::Result<String>`。换行符不包含在字符串中。如果输入中有 Windows 风格的换行符 "`\r\n`"，这两个字符都会被丢弃。

这个方法几乎总是你需要的文本输入方法。下面的两节会通过例子展示如何使用它。

```
reader.read_until(stop_byte, &mut byte_vec), reader.split(stop_byte)
```

这两个方法类似于 `.read_line()` 和 `.lines()`，但是是面向字节的，产生 `Vec<u8>` 而不是 `String`。你可以选择终止符 `stop_byte`。

`BufRead` 还提供两个底层的方法 `.fill_buf()` 和 `.consume(n)`，用来直接访问 `reader` 的内部缓冲区。更多有关这些方法的信息，可以查阅在线文档。

接下来的两节详细介绍了有缓冲的 reader。

18.1.3 读取行

这里有一个实现了 Unix `grep` 工具的函数。它搜索文本的每一行，文本通常通过管道从另一个命令输入。对于一个给定的字符串：

```
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);
        }
    }
}
```

```

        if line.contains(target) {
            println!("{}", line);
        }
    }
    Ok(())
}

```

因为我们想调用`.lines()`, 所以我们需要一个实现了`BufRead`的输入源。在这个例子中, 我们调用了`io::stdin()`来获取通过管道传入的数据。然而, Rust 标准库使用了一个 mutex 来保护 `stdin`, 我们调用`.lock()`来锁住 `stdin`以让当前的线程独占使用, 它返回一个实现了`BufRead`的`StdinLock`值。在循环的结尾, `StdinLock`被丢弃, 释放 mutex。(如果没有 mutex, 那么如果两个线程同时从 `stdin` 中读取数据, 会导致未定义行为。C 里也有这个问题, 它通过这种方式解决它: C 中所有的输入和输出函数会在幕后获取一个锁。Rust 中唯一不同就是锁是 API 的一部分。)

函数的剩余部分非常直观: 它调用`.lines()`并迭代返回的迭代器。因为这个迭代器产生`Result`值, 所以我们使用`?操作符`来检查错误。

假设我们想进一步扩展我们的`grep`程序, 让它支持搜索磁盘中的文件。我们可以把函数修改为泛型的:

```

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(())
}

```

现在我们可以向它传递一个`StdinLock`或者一个有缓存的`File`:

```

let stdin = io::stdin();
grep(&target, stdin.lock())?; // ok

let f = File::open(file)?;
grep(&target, BufReader::new(f))?; // ok

```

注意`File`并不是自动缓存的。`File`实现了`Read`但没有实现`BufRead`。然而, 很容易为`File`或者其他任何无缓存的 reader 创建一个有缓存的 reader。`BufReader::new(reader)`可

以实现这个功能。(可以使用 `BufReader::with_capacity(size, reader)` 设置缓冲区的大小。)

在大多数语言中，文件都是默认有缓存的。如果你想要无缓存的输入或输出，你必须知道如何关闭缓存。在 Rust 中，`File` 和 `BufReader` 是两个单独的库特性，因为有时你可能需要没有缓冲的文件，或者需要缓存文件之外的内容（例如，你可能会想要缓存来自网络的输入）。

包含错误处理和一些参数解析的完整的程序，如下所示：

```
// grep - 搜索 stdin 或文件中匹配给定 string 的行
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
```

```

        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

包括`.lines()`在内的几个 reader 方法返回产生 `Result` 的迭代器。当你第一次尝试将一个文件的每一行收集到一个很大的 `vector` 中时，你可能会遇到需要摆脱 `Result` 的问题：

```

// ok, 但不是你想要的
let results: Vec<io::Result<String>> = reader.lines().collect();

// error: 不能将 Result 的集合转换成 Vec<String>
let lines: Vec<String> = reader.lines().collect();

```

第二次尝试不能编译：哪里出错了？直观的解决方法是编写一个 `for` 循环并为每一个 item 检查错误：

```

let mut lines = vec![];
for line_result in reader.lines() {
    lines.push(line_result?);
}

```

不错；但这里如果使用`.collect()`会更好，并且我们确实可以这么做。我们只需要知道需要什么样的类型：

```
let lines = reader.lines().collect::<io::Result<Vec<String>>>();
```

为什么这能工作？标准库里为 `Result` 包含了一个 `FromIterator` 的实现——在在线文档中容易忽略——让这变为了可能：

```

impl<T, E, C> FromIterator<Result<T, E>> for Result<C, E>
where C: FromIterator<T>

```

{

...

}

这个签名需要仔细阅读，但它是一个漂亮的技巧。假设 C 是任意集合类型，例如 Vec 或者 HashSet。只要我们已经知道了如何从一个产生 T 值的迭代器构建一个 C，我们就可以从一个产生 Result<T, E> 值的迭代器构建一个 Result<C, E>。我们只需要遍历迭代器产生的值，用其中的 Ok 值构建集合，但如何遇到了一个 Err，就停止并传递它。

换句话说，io::Result<Vec<String>> 是一个集合类型，所以.collect() 方法可以创建并填充这种类型的值。

18.1.5 Writer

正如我们所见，使用方法就基本可以完成输入。输出有一些不同。

在整本书中，我们都在使用 println!() 来产生普通文本输出：

```
println!("Hello, world!");

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

println!(); // 打印空自行
```

还有一个 print!() 宏，它不会在最后加上一个换行符，eprintln! 和 eprint! 宏写入到标准错误流。这些函数的格式化代码都和 format! 宏一样，见 [格式化](#)。

使用 write!() 和 writeln!() 宏可以把输出写入一个 writer。它们与 print!() 和 println!() 类似，除了两个不同点：

```
writeln!(io::stderr(), "error: world not helloable")?;

writeln!(&mut byte_vec, "The greatest common divisor of {:?} is {}", numbers, d)?;
```

一是 write 宏有一个额外的第一个参数：writer。另一个不同是它们返回一个 Result，因此必须进行错误处理。这就是为什么我们在每一行的结尾都使用了? 运算符。

print 宏不返回一个 Result，如果写入失败它们会直接 panic。因为它们会写入到终端，写入终端很少会失败。

Write trait 有这些方法：

```
writer.write(&buf)
```

将切片 buf 中的字节写入到底层的流中。它返回一个 `io::Result<usize>`。成功时，它返回写入的字节数量，可能会小于 `buf.len()`，取决于流。类似于 `Reader::read()`，这是一个你应该避免直接使用的底层方法。

```
writer.write_all(&buf)
```

写入切片 buf 中的所有字节。返回 `Result<()>`。

```
writer.flush()
```

冲洗底层流中所有缓存的数据。返回 `Result<()>`。注意尽管 `println!` 和 `eprintln!` 宏会自动冲洗标准输出和标准错误流，但 `print!` 和 `eprint!` 不会。使用它们之后你可能需要手动调用 `flush()`。

类似于 `reader`，`writer` 也是在丢弃时自动关闭。

类似于 `BufReader::new(reader)` 为任意 `reader` 添加缓存，`BufWriter::new(writer)` 为任意 `writer` 添加缓存：

```
let file = File::create("tmp.txt")?;
let writer = BufWriter::new(file);
```

为了设置缓冲区的大小，使用 `BufWriter::with_capacity(size, writer)`。

当 `BufWriter` 被丢弃时，它剩余的所有被缓存的数据都会被写入到底层的 `writer`。然而，如果这次写入时出现了错误，这个错误会被忽略。（因为这个错误是在 `BufWriter` 的 `.drop()` 方法中发生，没有汇报错误的地方。）为了保证你的应用能够注意到所有的输出错误，可以在 `drop` 有缓存的 `writer` 之前手动调用 `.flush()`。

18.1.6 File

我们已经看到过两种打开文件的方式：

```
File::open(filename)
```

打开一个已存在的文件。它返回一个 `io::Result<File>`，如果文件不存在将返回一个错误。

```
File::create(filename)
```

创建一个新的文件用于写入。如果已经有同名文件，它会被截断。

注意 `File` 类型在文件系统模块 `std::fs` 中，而不是在 `std::io` 中。

当这两个文件都不符合要求时，你可以使用 `OpenOptions` 来指定额外的期望行为：

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

let log = OpenOptions::new()
    .append(true) // 如果文件存在，就追加到末尾
    .open("server.log")?;

let file = OpenOptions::new()
    .write(true)
    .create_new(true) // 如果文件存在就失败
    .open("new_file.txt")?;
```

方法.append(), .write(), .create_new() 等，被设计用来进行类似这样的链式调用：每一个都返回 self。这种链式方法的设计模式在 Rust 中太过普遍以至于有一个专门的名字：它被称为 *builder*(构建器)。std::process::Command 是另一个例子。更多关于 OpenOptions 的细节可以查阅在线文档。

File 被打开后，它的行为就类似于其他的 reader 和 writer。如果需要的话你可以添加一个缓冲区。当你 drop 一个 File 时它会自动关闭。

18.1.7 Seek

File 还实现了 Seek trait，它意味着你可以在一个 File 中跳来跳去，而不是只能从开始单调地读到尾。Seek 的定义类似如下：

```
pub trait Seek {
    fn seek(&mut self, pos: SeekFrom) -> io::Result<u64>;
}

pub enum SeekFrom {
    Start(u64),
    End(i64),
    Current(i64)
}
```

得益于这个枚举，seek 方法变得很有表达力：使用 file.seek(SeekFrom::Start(0)) 来定位到开始，使用 file.seek(SeekFrom::Current(-8)) 来回退一些字节，等等。

在一个文件中定位很慢。不管你是在硬盘还是固态盘 (SSD) 上，定位都要消耗和读取几 M 数据一样长的时间。

18.1.8 其他 Reader 和 Writer 类型

目前为止，本章主要使用了 File 作为示例，但还有很多其他有用的 reader 和 writer 类型：

io::stdin()

返回一个标准输入流的 reader。它的类型是 `io::Stdin`。因为它被多个线程共享，所以每一次读取都需要请求并释放 mutex。

`Stdin` 有一个 `.lock()` 方法获取 mutex 并返回一个 `io::StdinLock`，这是一个有缓存的 reader，它会持有 mutex，直到它被丢弃。因此对 `StdinLock` 的单独操作可以避免 mutex 的开销。我们在 [读取行](#) 中展示过使用这个方法的示例代码。

出于技术原因，`io::stdin().lock()` 不能工作。这个锁持有一个 `Stdin` 值的引用，这意味着 `Stdin` 值必须被存储起来，这样它才能生存的足够久：

```
let stdin = io::stdin();
let lines = stdin.lock().lines(); // ok
```

io::stdout(), io::stderr()

返回标准输出和标准错误流的 `Stdout` 和 `Stderr` writer 类型。这两个类型也持有互斥锁和 `.lock()` 方法。

Vec<u8>

实现了 `Write`。写入到一个 `Vec<u8>` 会把新的数据附加到 vector 尾部。

然而，`String` 并没有实现 `Write`。为了使用 `Write` 构建一个字符串，首先要写入到一个 `Vec<u8>`，然后使用 `String::from_utf8(vec)` 来把 vector 转换为字符串。

Cursor::new(buf)

创建一个 `Cursor`，它是一个从 `buf` 中读取的有缓存的 reader。这也是一个创建从 `String` 读取的 reader 的方法。参数 `buf` 可以是任何实现了 `AsRef<[u8]>` 的类型，因此你也可以传递一个 `&[u8]`, `&str`, `Vec<u8>`。`Cursor` 内部的结构非常简单。它只有两个字段：`buf` 和一个整数，用来表示下一次读取开始的偏移量。初始时为 0。

`Cursor` 实现了 `Read`, `BufRead`, `Seek`。如果 `buf` 的类型是 `&mut [u8]` 或者 `Vec<u8>`，那么 `Cursor` 还会实现 `Write`。写入一个 `Cursor` 会覆盖 `buf` 中从当前位置开始的字节。如果你试图越界写入一个 `&mut [u8]`，结果会是部分写入或者一个 `io::Error`。使用 `Cursor` 越界写入一个 `Vec<u8>` 没有问题，因为它会让 vector 变长。因此 `Cursor<&mut [u8]>` 和 `Cursor<Vec<u8>>` 实现了 `std::io::prelude` 中全部的 4 个 trait。

std::net::TcpStream

代表一个 TCP 网络连接。因为 TCP 允许双向连接，所以它既是 reader 又是 writer。

类型关联函数 `TcpStream::connect(("hostname", PORT))` 尝试连接到服务器，并返回一个 `io::Result<TcpStream>`。

`std::process::Command`

支持创建一个子进程并把数据管道连接到它的标准输入，例如：

```
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, 所以它会退出
child.wait()?;

child.stdin的类型是Option<std::process::ChildStdin>; 这里我们在创建子进程的时候使用了.stdin(Stdio::piped()), 因此.spawn()成功后child.stdin肯定是Some。否则child.stdin将是None。
```

`Command` 还有类似的 `.stdout()` 和 `.stderr()` 方法，它们可以用来请求 `child.stdout` 和 `child.stderr` 中的 reader。

`std::io` 模块还提供了很多返回简单 reader 和 writer 的函数：

`io::sink()`

这是一个无操作的 writer。所有的写入方法都会返回 `Ok`，但数据都会被丢弃。

`io::empty()`

这是一个无操作的 reader。所有的读取都会成功，但总是返回输入结束。

`io::repeat(byte)`

返回一个无限重复给定字节的 reader。

18.1.9 二进制数据，压缩和序列化

有很多基于 `std::io` 框架的开源 crate 提供额外的特性。

`byteorder` crate 提供 `ReadBytesExt` 和 `WriteBytesExt` trait，它们为所有 reader 和 writer 添加二进制输入和输出的方法：

```
use byteorder::{ReadBytesExt, WriteBytesExt, LittleEndian};

let n = reader.read_u32::<LittleEndian>()?;
writer.write_i64::<LittleEndian>(n as i64)?;
```

`flate2` crate 提供读取和写入 `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 以及它关联的格式化 crate 例如 `serde_json`，实现了序列化和反序列化：它们在 Rust 结构体和字节流之间来回转换。我们之前在 trait 和其他人的类型 中提到过它们一次。现在让我们仔细看看。

假设我们有一些数据，即一个文字冒险游戏的地图，存储在一个 `HashMap` 中：

```
type RoomId = String; // 每一个房间有一个独一无二的名字
type RoomExits = Vec<(char, RoomId)>; // ... 和一个通向的房间的名字的列表
type RoomMap = HashMap<RoomId, RoomExits>;
```



```
// 创建一个简单的地图。
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())]);
...
```

将这个数据转换为 JSON 并输出只需要一行代码：

```
serde_json::to_writer(&mut std::io::stdout(), &map)?;
```

在内部，`serde_json::to_writer` 使用了 `serde::Serialize` trait 的 `serialize` 方法。这个库给所有它知道如何序列化的类型附加了这个 trait，其中包括我们的数据中出现的类型：字符串、字符、元组、vector、`HashMap`。

`serde` 非常灵活。在我们的程序中，输出是 JSON 数据，因为我们选择了 `serde_json` 序列化器。其他格式例如 `MessagePack` 也是可用的。同样地，你可以把输出送到文件、`Vec<u8>` 或其他任何 `writer` 中。上面的代码通过 `stdout` 打印了数据，内容如下：

```
{"Debris Room": [{"E": "Cobble Crawl"}, {"W": "Sloping Canyon"}], "Cobble Crawl": [{"W": "Debris Room"}]}
```

`serde` 还包括派生两个关键 trait 的支持：

```
#[derive(Serialize, Deserialize)]
struct Player {
    location: String,
    items: Vec<String>,
    health: u32
}
```

这个 `#[derive]` 属性会让编译过程稍微变长，因此当你在 `Cargo.toml` 文件中将 `serde` 列为依赖时需要要求它支持这个特性。这是我们上面的代码用到的依赖：

```
[dependencies]
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
```

更多的细节可以查阅 `serde` 的文档。简单来说，构建系统可以自动为 `Player` 生成 `serde::Serialize` 和 `serde::Deserialize`，因此序列化一个 `Player` 值非常简单：

```
serde_json::to_writer(&mut std::io::stdout(), &player)?;
```

输出看起来是这样的：

```
{"location": "Cobble Crawl", "items": ["a wand"], "health": 3}
```

18.2 文件和目录

现在我们已经展示了如何使用 `reader` 和 `writer`，下面的几节将介绍 Rust 中处理文件和目录的特性，它们在 `std::path` 和 `std::fs` 模块中。这些特性都涉及到文件名，所以我们将以文件名类型开始。

18.2.1 OsStr 和 Path

很不方便的一点是，你的操作系统并不一定强制文件名是有效的 Unicode。这里有两个创建文本文件的 Linux shell 命令。只有第一个是有效的 UTF-8 文件名：

```
$ echo "hello world" > ô.txt
$ echo "O brave new world, that has such filenames in't" > '$'\xf4'.txt
```

两条命令都可以运行，因为Linux内核不知道来自Ogg Vorbis的UTF-8。对于内核来说，任何字节（除了null字节和斜杠）组成的字符串都是可接受的文件名。Windows上也类似：几乎任何16位“宽字符”组成的字符串都是可接受的文件名，即使字符串并不是有效的UTF-16。操作系统处理的其他字符串也是这样，例如命令行参数和环境变量。

Rust的字符串总是有效的Unicode。在实践中文件名几乎总是Unicode，但Rust必须提供方式以应对少数不是Unicode的情况。这就是为什么Rust有`std::ffi::OsStr`和`OsString`。

`OsStr`是一个作为UTF-8超集的字符串类型。它的任务是能表示当前系统中的所有文件名、命令行参数、环境变量，不管它们是不是`Unicode`。在Unix上，`OsStr`可以存储任意字节序列。在Windows上，`OsStr`以UTF-8的扩展格式存储，它可以编码任何16位值的序列。

所以我们有了两种字符串类型：`str`用于实际的Unicode字符串；`OsStr`用于操作系统可能用到的字符串。我们将再介绍一个用于文件名的`std::path::Path`，它纯粹是为了方便。`Path`实际上很像`OsStr`，但它添加了很多和文件名相关的方法，我们将在下一节中介绍。可以使用`Path`表示绝对路径和相对路径。对于路径中每个单独的部分，使用`OsStr`。

最后，每个字符串类型都有一个相应的所有权的(*owning*)类型：`String`拥有一个堆上分配的`str`，一个`std::ffi::OsString`拥有一个堆上分配的`OsStr`，一个`std::path::PathBuf`拥有一个堆上分配的`Path`。[表18-1](#)列出了每个类型的一些特性。

表 18-1: 文件名类型

	str	OsStr	Path
非固定大小类型，总是以引用传递	是	是	是
包含任意Unicode文本	是	是	是
通常看起来就像UTF-8	是	是	是
可以包含非Unicode数据	否	是	是
文本处理方法	是	否	否
文件名相关方法	否	是	是
对应的的所有权、可增长的、堆上分配的类型	<code>String</code>	<code>OsString</code>	<code>PathBuf</code>
转换为所有权的类型	<code>.to_string()</code>	<code>.to_os_string()</code>	<code>.to_path_buf()</code>

所有这些类型都实现了一个公共的trait：`AsRef<Path>`，所以我们可以轻易地声明一个泛型函数接受“任何文件名类型”作为参数。这使用到了我们之前展示过的[AsRef与AsMut](#)：

```
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项技术，因此你可以自由地向它们传递字符串字面量。

18.2.2 Path 和 PathBuf 方法

`Path` 提供了下面这些方法：

`Path::new(str)`

将一个 `&str` 或者 `&OsStr` 转换为 `&Path`。它不会拷贝字符串，新的 `&Path` 和原本的 `&str` 或 `&OsStr` 指向相同的字节流：

```
use std::path::Path;
let home_dir = Path::new("/home/fwolfe");
```

(类似的方法 `OsStr::new(str)` 将 `&str` 转换为 `&OsStr`。)

`path.parent()`

返回 `path` 的父目录，如果有的话。返回类型是 `Option<&Path>`。
它也不会拷贝路径，`path` 的父目录总是 `path` 的一个子串：

```
assert_eq!(Path::new("/home/fwolfe/program.txt").parent(),
           Some(Path::new("/home/fwolfe")));
```

`path.file_name()`

返回 `path` 的最后一个部分，如果有的话。返回类型是 `Option<&OsStr>`。
在通常的情况下，`path` 由一个目录、一个斜杠、然后是一个文件名组成，这会返回文件名：

```
use std::ffi::OsStr;
assert_eq!(Path::new("/home/fwolfe/program.txt").file_name(),
           Some(OsStr::new("program.txt")));
```

`path.is_absolute()`, `path.is_relative()`

这些方法判断路径是绝对的（例如 Unix 路径 `/usr/bin/advent` 或者 Windows 路径 `C:\Program Files`）还是相对的（例如 `src/main.rs`）。

```
path1.join(path2)
```

连接两个路径，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 是一个绝对路径，这会简单地返回 path2 的拷贝，因此这个方法可以用于将任何路径转换为一个绝对路径：

```
let abs_path = std::env::current_dir()?.join(any_path);
```

```
path.components()
```

返回一个从左到右迭代给定路径的所有部分的迭代器。这个迭代器的 item 类型是 std::path::Component，它可以代表任何可能出现在文件名中的部分：

```
pub enum Component<'a> {
    Prefix(PrefixComponent<'a>), // 一个驱动器字母或者共享设备 (在Windows 上)
    RootDir, // 根目录, `/` 或 `\
    CurDir, // `.` 特殊目录
    ParentDir, // `..` 特殊目录
    Normal(&'a OsStr) // 普通的文件和目录名
}
```

例如，Windows 路径 `||venice\Music\A Love Supreme\04-Psalm.mp3` 由一个 Prefix (表示 `||venice\Music`)、后跟一个 RootDir，然后是两个 Normal 组件 (分别是 `A Love Supreme` 和 `04-Psalm.mp3`) 组成。

细节见[在线文档](#)。

```
path.ancestors()
```

返回一个从 path 一直回溯到根目录的迭代器。每一个产生的 item 都是一个 Path：第一个是 path 本身，然后是它的父目录、它的父目录的父目录，等等：

```
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("/")]);
```

这很像一直调用 `parent` 直到它返回 `None`。最终的 `item` 总是一个根目录或者前缀路径。这些方法只考虑内存中的字符串。`Path` 还有一些会查询文件系统的方法：`.exists()`, `.is_file()`, `.is_dir()`, `.read_dir()`, `.canonicalize()` 等等。更多内容请查阅在线文档。

有三个将 `Path` 转换为字符串的方法。每一个都允许 `Path` 中可能含有无效的 UTF-8:

`path.to_str()`

将一个 `Path` 转换成字符串，返回一个 `Option<&str>`。如果 `path` 不是有效的 UTF-8，它返回 `None`:

```
if let Some(file_str) = path.to_str() {
    println!("{}", file_str);
} // ... 否则跳过这个文件名
```

`path.to_string_lossy()`

这个方法功能基本和上面一样，但它在所有情况下都会返回字符串。如果 `path` 不是有效的 UTF-8，这个方法会创建拷贝，然后将每一个无效的字节序列替换为 Unicode 占位字符：U+FFFD('')。

返回值类型是 `std::borrow::Cow<str>`: 可能是字符串的借用也可能是所有权的字符串。为了从这个值得到一个 `String`，使用它的 `.to_owned()` 方法。(更多有关 `Cow` 的内容，见[Borrow 和 ToOwned 的配合: Cow](#)。)

`path.display()`

这用于打印路径:

```
println!("Download found. You put it in: {}", dir_path.display());
```

它返回的值并不是字符串，但它实现了 `std::fmt::Display`，所以它可以和 `format!()`, `println!()` 等一起使用。如果路径不是有效的 UTF-8，输出可能会含有' '字符。

18.2.3 文件系统访问函数

[表 18-2](#)展示了 `std::fs` 中的一些函数以及它们在 Unix 和 Windows 中的类似等价物。所有这些函数都返回 `io::Result` 值。除非特意提及，不然就是 `io::Result<()>`。

(`copy()` 返回的数字是被拷贝的文件的大小，以字节为单位。有关创建符号链接，见[平台特定特性](#)。)

如你所见，Rust 努力提供可以在 Windows、macOS、Linux 以及其他 Unix 系统上工作的可移植函数。

表 18-2: 文件系统访问函数总结

	Rust 函数	Unix	Windows
创建和删除	create_dir(path)	mkdir()	CreateDirectory()
	create_dir_all(path)	类似 mkdir -p	类似 mkdir
	remove_dir(path)	rmdir()	RemoveDirectory()
	remove_dir_all(path)	类似 rm -r	类似 rmdir /s
拷贝，移动和链接	remove_file(path)	unlink()	DeleteFile()
	copy(src_path, dest_path) -> Result<u64>	类似 cp -p	CopyFileEx()
	rename(src_path, dest_path)	rename()	MoveFileex()
	hard_link(src_path, dest_path)	link()	CreateHardLink()
检查	canonicalize(path) -> Result<PathBuf>	realpath	GetFinalPathNameByHandle()
	metadata(path) -> Result<Metadata>	stat()	GetFileInformationByHandle()
	symlink_metadata(path) -> Result<Metadata>	lstat()	GetFileInformationByHandle()
	read_dir(path) -> Result<ReadDir>	opendir()	FindFirstFile()
权限	read_link(path) -> Result<PathBuf>	readlink()	FSCTL_GET_REPARSE_POINT
	set_permission(path, perm)	chmod()	SetFileAttributes()

文件系统的完整说明超出了本书的范围，但如果你对这些函数中的某些更感兴趣，你可以在网上轻松地找到有关他们的更多信息。我们将在下一节中展示更多示例。

所有这些函数都是通过调用操作系统的功能来实现。例如 `std::fs::canonicalize(path)` 不只是使用字符串处理来消除给定的 `path` 中的 `.` 和 `..`。它使用当前的工作目录来解析相对路径，并且它会解析符号链接。如果路径不存在它会报错。

`std::fs::metadata(path)` 和 `std::fs::symlink_metadata(path)` 产生的 `Metadata` 类型包含类似于文件类型和大小、权限、时间戳等信息。同样，详细的内容请查阅文档。

为了方便，`Path` 类型将一些这样的函数内建为方法：例如 `path.metadata()` 和 `std::fs::metadata(path)` 是一样的。

18.2.4 读取目录

可以使用 `std::fs::read_dir` 列出目录中的内容。或者等价的 `Path` 的 `.read_dir()` 方法：

```
for entry_result in path.read_dir()? {
    let entry = entry_result?;
    println!("{} ", entry.file_name().to_string_lossy());
}
```

注意这段代码中`?`的两次使用。第一行的检查打开目录的错误。第二行的检查读取下一个条目的错误。

`entry` 的类型是 `std::fs::DirEntry`，它有如下方法：

`entry.file_name()`

文件或目录的名字，是一个 `OsString`。

`entry.path()`

和上面相同，但和原本的路径连接在一起，产生一个新的 `PathBuf`。如果我们正在列出的目录是`"/home/jimb"`，并且 `entry.file_name()` 是`".emacs"`，那么 `entry.path()` 将会返回 `PathBuf::from("/home/jimb/.emacs")`。

`entry.file_type()`

返回一个 `io::Result<FileType>`。`FileType` 类型有 `.is_file()`、`.is_dir()`、`.is_symlink()` 方法。

`entry.metadata()`

获取这个条目的其他元数据。

在读取目录时特殊目录. 和.. 不会被列出。

这里还有另一个示例。下面的代码递归拷贝磁盘上的一个目录树：

```
use std::fs;
use std::io;
use std::path::Path;

/// 拷贝现有的目录`src`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(())
}
```

用一个单独的函数 `copy_to` 来拷贝单独的目录项：

```
/// 拷贝`src`中的所有东西到目标路径`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 平台特定特性

到目前为止，我们的 `copy_to` 函数可以拷贝文件和目录。假设我们还想在 Unix 上支持符号链接。

目前并没有可移植的方式能创建同时在 Unix 和 Windows 上工作的符号链接，但标准库提供了一个 Unix 特定的 `symlink` 函数：

```
use std::os::unix::fs::symlink;
```

有了这个，我们的工作就变得很简单。我们只需要给 `copy_to` 里的 `if` 表达式添加一个分支：

```
...
} else if src_type.is_symlink() {
    let target = src.read_link()?;
    symlink(target, dst)?;
...
```

只要我们在 Unix 系统例如 Linux 和 macOS 上编译程序，它就可以工作。

`std::os` 模块包含很多平台特定的特性，例如 `symlink`。`std::os` 在标准库中的实际内容看起来像这样（取得了许可）：

```
/// OS 特定的功能

#[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都只在特定平台上可用。这也是为什么我们的修改后使用了 `std::os::unix` 的程序在 Unix 上将会成功编译：在其他平台上，`std::os::unix` 不存在。

如果我们想让我们的代码在所有平台上编译，并且支持 Unix 上的符号链接，我们必须在我们的程序中也使用 `#[cfg]`。在这种情况下，最简单的方法是在 Unix 上时导入 `symlink`，而在其它系统上定义我们自己的 `symlink`：

```
#[cfg(unix)]
use std::os::unix::fs::symlink;

/// 为不支持'symlink'的平台的实现
#[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())))
}
```

事实证明 `symlink` 是一种特殊情况。大多数 Unix 特定的特性并不是单独的函数而是一些扩展的 trait，它们为标准库类型添加了一些的方法（我们在 trait 和其他人的类型 中介绍过扩展 trait）。这里有一个可以一次性启用所有这些扩展的 `prelude` 模块：

```
use std::os::unix::prelude::*;


```

例如，在 Unix 上这会给 `std::fs::Permissions` 添加一个 `.mode()` 方法，它提供 Unix 上表示权限的底层 `u32` 值的访问。类似的，它还扩展了 `std::fs::Metadata`，添加了一些访问底层的 `struct stat` 的字段的方法——例如 `.uid()` 返回文件所有者的 ID。

总而言之，`std::os` 中的内容非常基础。更多的平台特定功能通过第三方 crate 提供，例如 `winreg` 提供了访问 Windows 注册表的支持。

18.3 网络

有关网络编程的教程超出了本书的范围。然而，如果你已经知道一些有关网络编程的知识，那么这一节可以帮助你在 Rust 中开始网络编程。

底层的网络编程需要使用 `std::net` 模块，它提供了 TCP 和 UDP 网络的跨平台支持。使用 `native_tls` crate 来提供 SSL/TLS 支持。

这些模块提供了通过网络的直观的、阻塞式的输入和输出。你可以通过 `std::net` 用很少的代码编写一个简单的服务器，为每一个连接创建一个线程。例如，这里有一个“echo”服务器：

```
use std::net::TcpListener;
use std::io;
use std::thread::spawn;

/// 一直等待并接受连接，为每个连接新建一个线程处理。
fn echo_main(addr: &str) -> io::Result<()> {
    let listener = TcpListener::bind(addr)?;
    println!("listening on {}", addr);
    loop {
```

```

// 等待客户端连接。
let (mut stream, addr) = listener.accept()?;
println!("connection received from {}", addr);

// 创建一个线程来服务这个客户端。
let mut write_stream = stream.try_clone()?;
spawn(move || {
    // 把我们从`stream`接收到的所有内容写回。
    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: ");
}

```

一个回声服务器简单地把你发送给它的数据返回。这些代码和你在 Java 或 Python 中编写的代码并没有多少不同。(我们将在[下一章](#)中介绍 `std::thread::spawn()`)

然而，对于高性能的服务器，你将需要使用异步的输入和输出。[第 20 章](#)会介绍 Rust 对异步编程的支持，并展示编写网络客户端和服务器的完整代码。

更高层的协议由第三方 crate 支持。例如，`reqwest` crate 为 HTTP 客户端提供了一个漂亮的 API。这里有一个完整的命令行程序获取 `http:` 或者 `https:` URL 的文档并输出到终端。这段代码使用 `reqwest = "0.11"` 编写，并启用了它的"blocking" 特性。`reqwest` 还提供了一套异步的接口。

```

use std::error::Error;
use std::io;

fn http_get_main(url: &str) -> Result<(), Box

```

```
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);
    }
}
```

用于HTTP服务器的 `actix-web` 框架提供了更高层的特性，例如 `Service` 和 `Transform` trait，它们可以帮助你通过可组合的部分构建一个 app。`websocket` crate 实现了 WebSocket 协议，等等。Rust 是一门年轻的语言，有一个繁荣的开源生态系统。对网络的支持正在快速扩张。

Chapter 19

并发

In the long run it is not advisable to write large concurrent programs in machine-oriented languages that permit unrestricted use of store locations and their addresses. There is just no way we will be able to make such programs reliable (even with the help of complicated hardware mechanisms).

——Per Brinch Hansen (1977)

Patterns for communication are patterns for parallelism.

——Whit Morris

如果你在职业生涯之中对并发的态度发生了改变，那你并不孤单，这是一种很常见的情况。

一开始的时候，编写并发代码是轻松并且愉快的。那些工具——线程，锁，队列等等——很容易上手和使用。虽然说实话也有很多的陷阱，但幸运的是你知道它们都是什么，所以你可以小心地不犯错误。

但有时，你不得不调试一些别人的多线程代码，然后你被迫得出结论：有些人确实不应该使用这些工具。

然后有时你必须调试你自己的多线程代码。

经验会让你怀疑所有的多线程代码是否健康。少数的文章解释了为什么一些明显正确的多线程惯用写法完全不能工作，它们也许会有帮助。（它与“内存模型”有关。）但你最终会找到一种并发的方式，并且你觉得你能实际使用它且不会一直出错。你可能会把很多东西都塞进这种方法中，并且（如果你真的很优秀）你学会了对添加的复杂性说“不”。

当然，有很多种这样的方法。系统程序员经常使用的方法包括下面这些：

1. 一个只处理单个任务的后台线程 (*background thread*)，周期性地唤醒它执行任务。

2. 通用的线程池 (*worker pool*)，通过任务队列 (*task queue*) 和客户端交互。
3. 流水线 (*pipeline*)，数据从一个线程流向下一个，每个线程都做一些工作。
4. 数据并行 (*data parallelism*)，假设整个计算机主要在做很大型的计算，因此将数据分成 n 片然后在 n 个线程上运行，以让机器的 n 个核心一起工作。
5. 同步对象之海 (*a sea of synchronized objects*)，多个线程都有同一个数据的访问权限，使用基于底层原语例如 mutex 的 ad hoc 锁 (*lock*) 方案来避免竞争。(Java 内建了对这种模型的支持，这种模型在 20 世纪 90 年代和 21 世纪初非常流行。)
6. 原子整数操作 (*atomic integer operation*) 允许多个核通过一个机器字大小的字段传递信息来通信。(这比其他方式更难正确实现，除非交换的数据实际上只是整数值。在实践中，它通常是指针。)

随着时间的推移，你可能可以使用多种方式并安全地组合在一起。这时，你就是大师。如果没有其他人被允许修改系统，那么一切都会正常运作。正确使用线程的程序充满了不成文的规则。

Rust 提供了一种更好的方式来使用并发，它并不强迫所有的程序使用单一的风格（这对系统程序员来说并不是解决方案），而是安全地支持多种风格。不成文的规则现在被写下来了——就在代码中——并且被编译器强迫遵循。

你可能听说过 Rust 可以让你编写安全、快速、并发的程序。本章我们将向你展示如何做到这些。我们将介绍三种使用 Rust 线程的方式：

1. fork-join 并行
2. channel
3. 共享可变状态

在此过程中，你将用到目前为止学过的所有 Rust 语言的知识。Rust 关心的引用、可变性、生命周期在单线程程序中也很有价值，但只有在并发程序中这些规则真正的重要性才会显现出来。它们可以扩展你的工具箱，快速正确地破解多种风格的多线程代码——没有 skepticism，没有 cynicism，没有 fear。

19.1 fork-join 并行

多线程最简单的使用场景是我们有几个想同时执行的完全独立的任务。例如，假设我们要对非常多的文档进行自然语言处理。我们可以编写一个循环：

```
fn process_files(filenames: Vec<String>) -> io::Result<()> {
    for document in filenames {
        let text = load(&document)?;      // 读取源文件
        let results = process(text);     // 计算数据
    }
}
```

```

    save(&document, results)?;      // 写入到输出文件
}
Ok(())
}

```

这个程序的运行情况如图 19-1 所示：

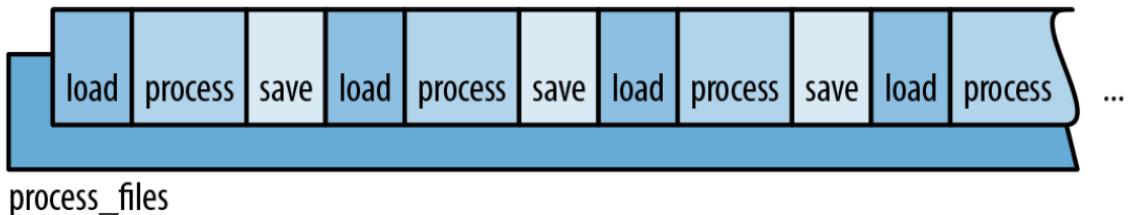


图 19-1: 单线程的 process_files() 执行过程

因为每一个文档都是被单独处理，所以可以很容易的把所有文档分成几块、每一块在单独的线程中进行处理来加快速度，如图 19-2 所示。

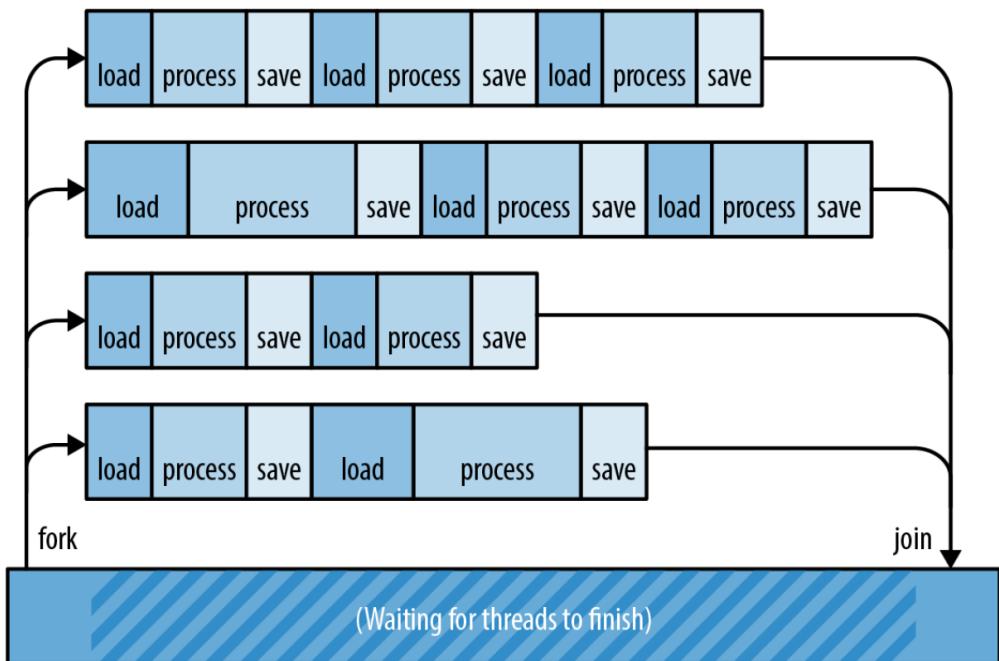


图 19-2: 使用 fork-join 方法的多线程文件处理

这个模式被称为 *fork-join* 并行。*fork* 是启动一个新的线程，*join* 一个线程是等待它结束。我们已经看到过这个技术了：我们在第 2 章中使用了它来加速曼德勃罗集程序。

fork-join 并行的优势主要在以下几个方面：

1. 它太简单了。fork-join 非常容易实现，并且 Rust 能帮你很轻松地保证它正确。
2. 它避免了瓶颈。fork-join 中没有共享资源的加锁，因此线程等待其他线程的唯一情况就是结束时。同时，每一个线程都可以自由地运行。这有助于保证很低的任务切换开销。
3. 性能也非常直观。在最好的情况下，通过启动 4 个线程，我们可以用 $\frac{1}{4}$ 的时间完成任务。
图 19-2 展示了为什么我们不应该期待这种理想的加速：我们可能不能均匀地把工作分布到所有的线程上。另一个要注意的原因是一些 fork-join 程序在所有的线程结束之后还要花费一些时间组合 (combine) 所有线程计算出的结果。即，完全分割任务可能要花费一些额外的工作。如果不考虑这两个原因，一个任务可以分割为独立单元的 CPU 密集的程序可以期待显著的性能提升。
4. 很容易推断程序的正确性。只要每个线程真的被隔离，那么一个 fork-join 程序是确定性的 (deterministic)，例如曼德勃罗集程序中的计算线程。程序总是会产生相同的结果，和线程速度的变化无关。它是一种没有竞争条件的并发模型。

fork-join 的主要缺点是它需要无关的任务单元。本章稍后我们会考虑一些不能分割的这么清晰的问题。

现在，让我们继续看自然语言处理的例子。我们将展示一些将 fork-join 模式应用于 `process_files` 函数的方式。

19.1.1 `spawn` 和 `join`

函数 `std::thread::spawn` 启动一个新线程：

```
use std::thread;

thread::spawn(|| {
    println!("hello from a child thread");
});
```

它接受一个参数：一个 `FnOnce` 闭包或者函数。Rust 启动一个新的线程来运行这个闭包或者函数。新的线程是一个真实的操作系统线程，有自己的栈，就类似于 C++、C#、Java 中的线程。

这里有一个例子，使用 `spawn` 来实现之前的 `process_files` 函数的一个并行版本：

```
use std::{thread, io};

fn process_files_in_parallel(filenames: Vec<String>) -> io::Result<()> {
    // 把工作分成几块。
    const NTHREADS: usize = 8;
    let worklists = split_vec_into_chunks(filenames, NTHREADS);
```

```
// Fork: 创建一个新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))  
    );  
}  
  
// Join: 等待所有线程结束。  
for handle in thread_handles {  
    handle.join().unwrap()?  
}  
  
Ok(())  
}
```

让我们逐行分析这个函数。

```
fn process_files_in_parallel(filenames: Vec<String>) -> io::Result<()> {
```

我们的新函数和原本的 `process_files` 有完全相同的类型签名，这让它可以更轻松地替换原来的函数。

```
// 把工作分成几块。  
const NTHREADS: usize = 8;  
let worklists = split_vec_into_chunks(filenames, NTHREADS);
```

我们使用了一个工具函数 `split_vec_into_chunks` 来分割任务，不过这里没有展示。分割的结果 `worklists` 是一个 `vector` 的 `vector`。它包含原本的 `vector` `filenames` 均匀分配的 8 个部分。

```
// Fork: 创建一个新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))  
    );  
}
```

我们为每一个 `worklist` 启动了一个线程。`spawn()` 返回一个称为 `JoinHandle` 的值，我们之后会使用它。现在，我们只是将所有的 `JoinHandle` 放进一个 `vector` 里。

注意我们如何把文件名的列表传进工作线程里：

1. `worklist` 在父线程的 `for` 循环中定义和初始化。
2. 一旦 `move` 闭包创建完成，`worklist` 就会被移动进闭包里
3. 然后 `spawn` 把闭包（包括 `worklist`）移动进新的子线程。

这些移动操作开销很小。正如我们在第4章中讨论的 `Vec<String>` 的移动一样，`String` 并不会被拷贝。事实上，没有任何东西被分配或者释放。唯一被移动的数据是 `Vec` 自己：三个机器字。

你创建的大多数线程都同时需要代码和数据才能运行。Rust 闭包可以便捷地包含你需要的代码和数据。

继续：

```
// Join: 等待所有线程结束。
for handle in thread_handles {
    handle.join().unwrap()?;
}
```

我们使用了之前收集的 `JoinHandle` 的 `.join()` 方法来等待 8 个线程全部结束。为了保证正确性，`join` 线程通常是必须的，因为一旦 `main` 返回 Rust 程序就会退出，即使其他的线程仍然在运行。析构器不会被调用，额外的线程被简单地杀死。如果这不是你想要的，确保在 `main` 返回之前 `join` 所有你关心的线程。

如果这个循环结束了，那么意味着 8 个子线程都已经成功结束。我们的函数返回一个 `Ok(())` 并结束：

```
Ok(())
}
```

19.1.2 跨线程的错误处理

我们的例子中用来 `join` 子线程的代码比看起来更加有趣，因为还要考虑错误处理。让我们回顾这一行代码：

```
handle.join().unwrap();
```

`.join()` 方法给我们带来了两个问题。

首先，`handle.join()` 返回一个 `std::thread::Result`，如果子线程 `panic` 那么它的值将是一个错误。这使得 Rust 中的线程比 C++ 中更健壮。在 C++ 中，一个越界的数组访问是未定义行为，并且没有办法保护系统中的其他部分不受其影响。在 Rust 中，`panic` 是安全的并且以线程为单位。线程之间的边界充当 `panic` 时的防火墙；`panic` 并不会自动从一个线程传播到依赖它的线程。一个线程中的 `panic` 在其他线程中作为一个值为错误的 `Result` 汇报。整个程序可以不受其影响。

在我们的程序中，我们没有尝试任何 panic 的处理。我们简单地对返回的 `Result` 调用 `.unwrap()`，假定它一定是一个 `Ok` 的结果而不是 `Err` 的结果。如果一个子线程确实 panic 了，那么这个断言会失败，因此父线程也会 panic。我们显式地把子线程的 panic 传播到父线程。

其次，`handle.join()` 会把子线程返回的值传递给父线程。我们传递给 `spawn` 的闭包的返回类型是 `io::Result<()>`，因为 `process_files` 返回这个类型。这个返回值不会被丢弃，当子线程结束时，它的返回值会被保存，`JoinHandle::join()` 会把值传递给父线程。

这个程序中 `handle.join()` 返回的类型是 `std::thread::Result<std::io::Result<()>>`。`thread::Result` 是 `spawn/join API` 的一部分；`io::Result` 是我们程序的一部分。

在我们的例子中，解包了 `thread::Result` 之后，我们对 `io::Result` 使用了 `? 运算符`，显式地把子线程的 I/O 错误传播给父线程。

整个过程看起来可能很复杂，但这其实也只是一行代码的功能。将它与其它语言比较：Java 和 C# 中的默认行为是将子线程的异常转储到终端，然后忽略。在 C++ 中，默认行为是终止进程。在 Rust 中，错误是 `Result` 值（数据）而不是异常（控制流）。它们像其他任何值一样在线程之间传递。每一次使用底层线程的 API 时，你都必须小心地编写错误处理代码，但考虑到你必须编写它们，所以 Rust 在这一点上做得很好。

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非常大的由英文单词和短语组成的数据库：

```
// 之前
fn process_files(filenames: Vec<String>

// 之后
fn process_files_in_parallel(filenames: Vec<String>, glossary: &GigabyteMap)
```

`glossary` 可能很大，所以我们通过引用传递它。我们应该如何更新 `process_files_in_parallel` 来在多个线程间传递词汇表呢？

显而易见的修改方法并不能工作：

```
fn process_files_in_parallel(filenames: Vec<String>,
                               glossary: &GigabyteMap)
    -> io::Result<()>
{
    ...
    for worklist in worklists {
        thread_handles.push(
            spawn(move || process_files(worklist, glossary)) // 错误
        );
    }
}
```

}

我们只是简单地给函数添加了一个 `glossary` 参数并把它传递给 `process_files`。Rust 会报错：

```
error[E0621]: explicit lifetime required in the type of `glossary`
--> src/lib.rs:75:17
|
61 |     glossary: &GigabyteMap)
|           ----- help: add explicit lifetime ``static`` to the
|                               type of `glossary`: ``&'static BTreeMap<String,
|                                         String>``
...
75 |             spawn(move || process_files(worklist, glossary))
|             ^^^^^ lifetime ``static`` required
```

Rust 在抱怨我们传递给 `spawn` 的闭包的生命周期，编译器给出的“有帮助的”信息实际上也没有一点帮助。

`spawn` 启动独立的线程。Rust 没有办法知道子线程会运行多长时间，因此它假设最坏的情况：它假设子线程会一直保持运行，然而在父线程结束之后父线程里的所有值都会消失。显然，如果子线程要持续很长时间，那么它运行所需要的闭包也应该持续这么长时间。但这个闭包有一个有界的生命周期：它依赖于引用 [glossary](#)，但引用并不会一直有效。

注意Rust拒绝这段代码是正确的！按照我们编写这个函数的方式，有可能一个线程遇到I/O错误，导致`process_files_in_parallel`在其他线程结束之前结束。子线程可能会在主线程释放`glossary`之后仍然尝试使用它。这会导致未定义行为。Rust不允许出现这种情况。

看起来 `spawn` 太过开放以至于不能支持跨线程共享引用。事实上，我们已经在 [偷取值的闭包](#) 中见到过一个类似于这样的例子。那一次我们的解决方案是使用 `move` 闭包把数据的所有权移动到新线程里。但那种方法在这里不能生效，因为我们有很多线程需要使用相同的数据。一个安全的替代方案是为每一个线程 `clone` 一份词汇表，但因为它很庞大，我们希望避免这些开销。幸运的是，标准库提供了另一种方案：自动引用计数。

我们在Rc 和 Arc: 共享所有权 中介绍过 Arc。现在正是使用它的时候：

```

-> io::Result<()>
{
    ...
    for worklist in worklists {
        // 这里的.clone() 调用只会克隆 Arc 并递增引用计数。
        // 它并不会克隆 GigabyteMap。
        let glossary_for_child = glossary.clone();
        thread_handles.push(
            spawn(move || process_files(worklist, &glossary_for_child)))
    );
}
...
}

```

我们改变了 `glossary` 的类型：为了进行并行分析，调用者必须传递一个 `Arc<GigabyteMap>`，它是使用 `Arc::new(giga_map)` 创建的一个指向堆上的 `GigabyteMap` 的智能指针。

当我们调用 `glossary.clone()` 时，我们是在创建 `Arc` 智能指针的一个拷贝，而不是整个 `GigabyteMap` 的拷贝。它只会递增一个引用计数。

修改之后，程序就可以编译运行了，因为它不再依赖引用的生命周期。只要有任何线程还持有一个 `Arc<GigabyteMap>`，它就会保持词汇表仍然存在，即使父线程提前退出。也不会有任何数据竞争，因为 `Arc` 里的数据是不可变的。

19.1.4 Rayon

标准库的 `spawn` 函数是一个重要的原语，但它并不是专为 fork-join 并行设计的。还有基于它们构建的更好的 fork-join API。例如，[第 2 章](#) 中我们使用了 Crossbeam 库把一些任务分割给 8 个进程。Crossbeam 的 *scoped thread*(作用域线程) 非常自然地支持 fork-join 并行。

Niko Matsakis 和 Josh Stone 编写的 Rayon 库是另外一个例子。它提供两种并发运行任务的方法：

```

use rayon::prelude::*;

// 并行做 2 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)` 同时调用两个函数并返回两个结果。`.par_iter()` 方法创建一个 `ParallelIterator`，它有 `map`、`filter` 以及其它的方法，非常像 Rust 中的 `Iterator`。在这两种情况下，Rayon 都会尽可能使用它自己的工作线程池来处理任务。你只需要告诉 Rayon 什么样的任务可以并行完成；Rayon 负责管理线程并尽可能好地分发任务。

图 19-3 展示了有关调用 `giant_vector().par_iter().for_each(...)` 的两种思考方式：(a) Rayon 为 vector 中的每个元素创建一个线程。(b) Rayon 在幕后维护等同于 CPU 核数个工作线程，每个工作线程运行在每个 CPU 核上，这样会更加高效。工作线程池被程序的所有线程共享，当同时有上千个任务到达时，Rayon 会划分任务。

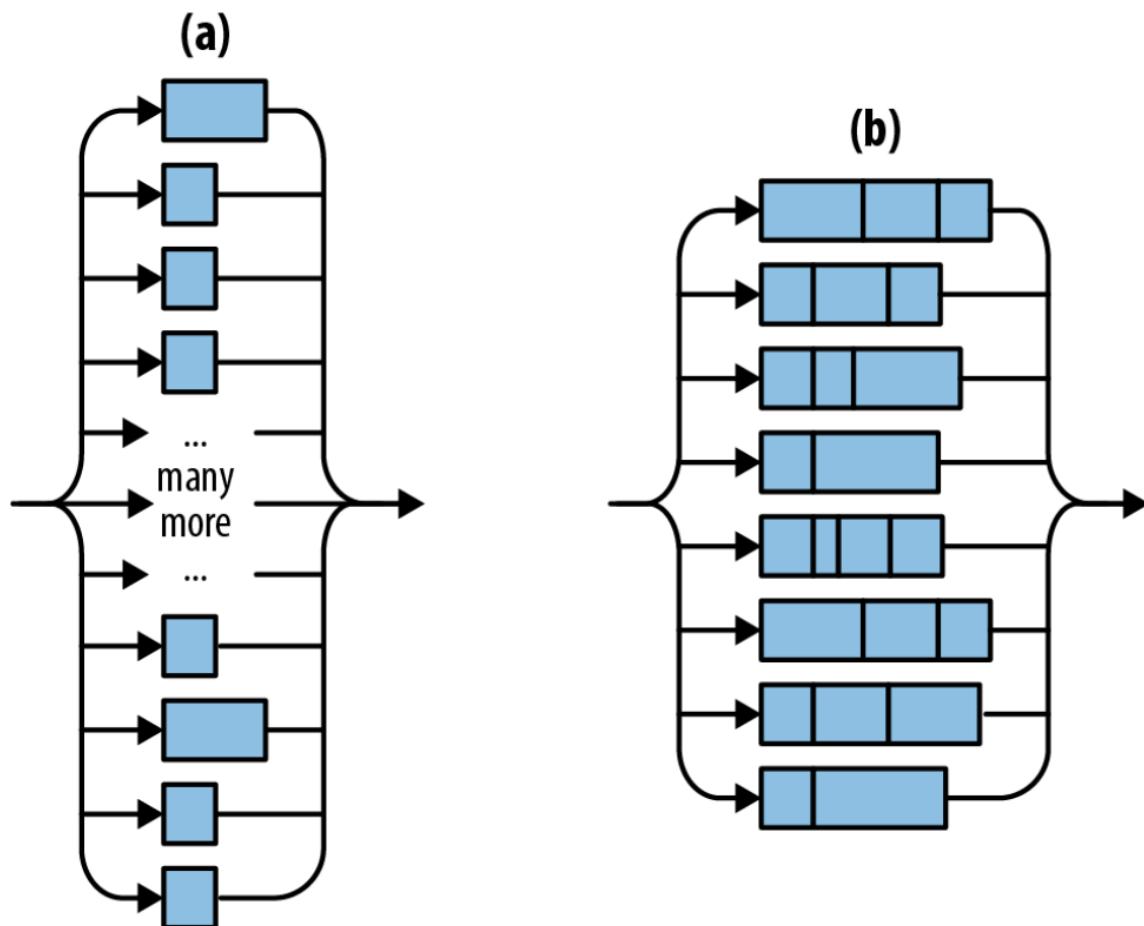


图 19-3: 理论和实践中的 Rayon

这里有一个使用 Rayon 的 `process_files_in_parallel` 的版本，并且它要运行的 `process_file` 函数，不再需要 `Vec<String>`，而是需要 `&str`：

```
use rayon::prelude::*;

fn process_files_in_parallel(filenames: Vec<String>, glossary: &GigabyteMap)
```

```
-> io::Result<()>
{
    filenames.par_iter()
        .map(|filename| process_file(filename, glossary))
        .reduce_with(|r1, r2| {
            if r1.is_err() { r1 } else { r2 }
        })
        .unwrap_or(Ok(()))
}
```

这段代码比使用 `std::thread::spawn` 的版本更加简短，也更加容易理解。让我们逐行分析它：

1. 首先，我们使用了 `filenames.par_iter()` 来创建一个并行迭代器。
2. 我们使用 `.map()` 来对每一个文件名调用 `process_file`。这会返回一个产生 `io::Result<()>` 值的序列的 `ParallelIterator`。
3. 我们使用 `.reduce_with()` 来组合结果。这里我们保留了第一个错误并丢弃剩余的内容。如果我们想累计所有的错误，或者打印它们，我们可以在这里完成。
4. 当你向 `.map()` 传递一个在成功时会返回有用值的闭包时，`.reduce_with()` 方法也非常方便。你可以向 `.reduce_with()` 传递一个知道如何组合两个成功结果的闭包。
5. `reduce_with` 返回一个 `Option`，只有当 `filenames` 为空时它是 `None`。我们使用了 `Option` 的 `.unwrap_or()` 方法以在这种情况下产生一个 `Ok(())`。

在幕后，Rayon 会使用一种叫做 *work-stealing* 的技术动态地实现多个线程的负载均衡。通常保持所有 CPU 核心忙碌比 `spawn` 和 `join` 中的事先手动划分任务效果会更好。

作为奖励，Rayon 支持在线程之间共享引用。所有幕后的并行处理都保证在 `reduce_with` 返回之前结束。这解释了为什么我们可以向 `process_file` 传递 `glossary`，即使这个闭包将会在多个线程间调用。

（顺便说一下，我们使用 `map` 和 `reduce` 方法并不是巧合。被 Google 和 Apache Hadoop 推广的 MapReduce 编程模型和 fork-join 方法有很多相同之处。它可以被看做处理分布式数据的 fork-join 方法。）

19.1.5 回顾曼德勃罗集

回顾第2章，我们使用了 fork-join 并发来渲染曼德勃罗集。这让渲染过程变快了四倍——这很有吸引力，但考虑我们是把程序分割为 8 个工作线程并在一台 8 核机器上运行它，它似乎还能做得更好。

问题在于我们没有平均分配工作负载。计算图像的一个像素点等同于运行一次循环（见曼

德勃罗集到底是什么)。事实证明对于图像的淡灰色部分的像素点，循环会快速退出，比渲染那些需要循环 255 次的黑色部分要快很多。因此尽管我们把区域分成了大小相等的水平条带，实际上我们创建的还是不同等的工作负载，如图 19-4 所示。

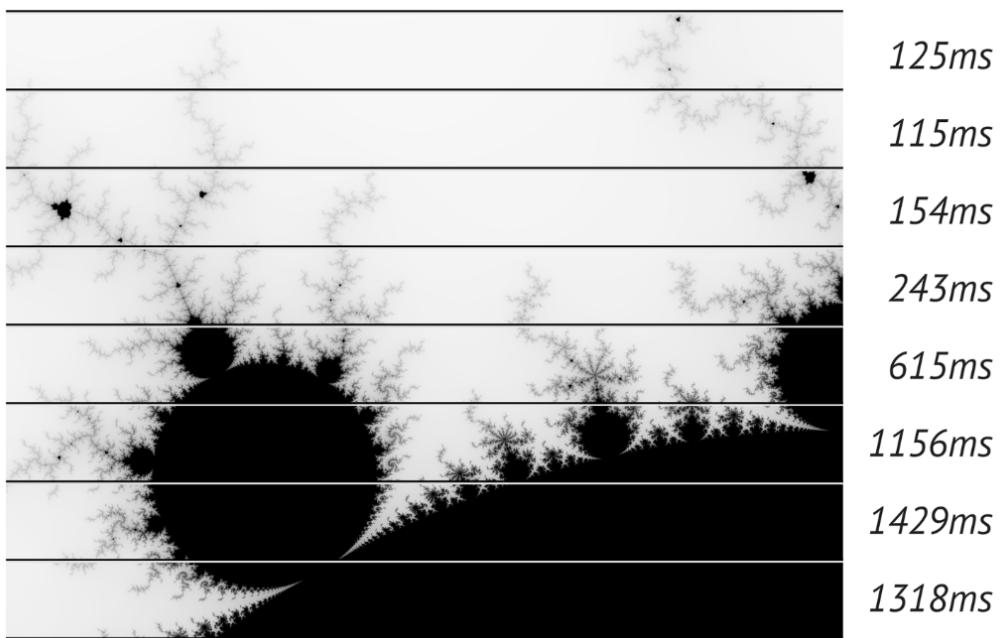


图 19-4: 曼德勃罗集中不均匀的工作划分

使用 Rayon 可以很容易地解决这个问题。我们可以简单地为输出中的每一行启动一个并行任务。这会创建几百个任务，Rayon 可以把它们分配给不同的线程。得益于 work-stealing，不同任务的大小不同并不会导致问题。Rayon 会在运行时均衡任务。

这里是代码。第一行和最后一行是我们在一个并发的曼德勃罗集中展示过的 main 函数的一部分，但我们修改了中间的渲染部分代码：

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

// 将 `pixels` 切割为水平条带的作用域
{
    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");

```

首先我们创建了 `bands`，它是我们要传递给 Rayon 的任务的集合。每一个任务都只是一个类型为 `(usize, &mut [u8])` 的元组：行号（计算时要用到它）以及要填充的 `pixels` 的切片。我们使用了 `chunks_mut` 方法把图像缓冲区分割成很多行，`enumerate` 来给每一行附加一个行号，使用 `collect` 来把所有行号-切片对收集成 `vector`。（我们需要 `vector` 是因为 Rayon 只能根据数组和 `vector` 创建并行迭代器。）

接下来，我们把 `bands` 变为一个并行迭代器，使用 `.for_each()` 方法来告诉 Rayon 我们想要做什么工作。

因为我们在使用 Rayon，因此我们必须在 `main.rs` 中加上下面这一行：

```
use rayon::prelude::*;


```

并在 `Cargo.toml` 中加上：

```
[dependencies]
rayon = "1"
```

有了这些修改，这个程序现在在 8 核机器上大概使用了 7.75 个核。它比之前手动分割任务快了 75%，代码也短了一些，这反映出让 `crate` 完成一些工作（负载分发）比我们自己来做的优势。

19.2 通道

通道 (*channel*) 是一种从一个线程向另一个线程发送数据的管道。换句话说，它是一个线程安全的队列。

图 19-5 展示了通道应该怎么使用。它们和 Unix 管道有些像：一端发送数据，另一端接收。两端通常被两个不同的线程持有。但 Unix 管道用来发送字节，通道用来发送 Rust 值。`sender.send(item)` 把一个值放入通道中；`receiver.recv()` 从通道中取出一个值。所有权从

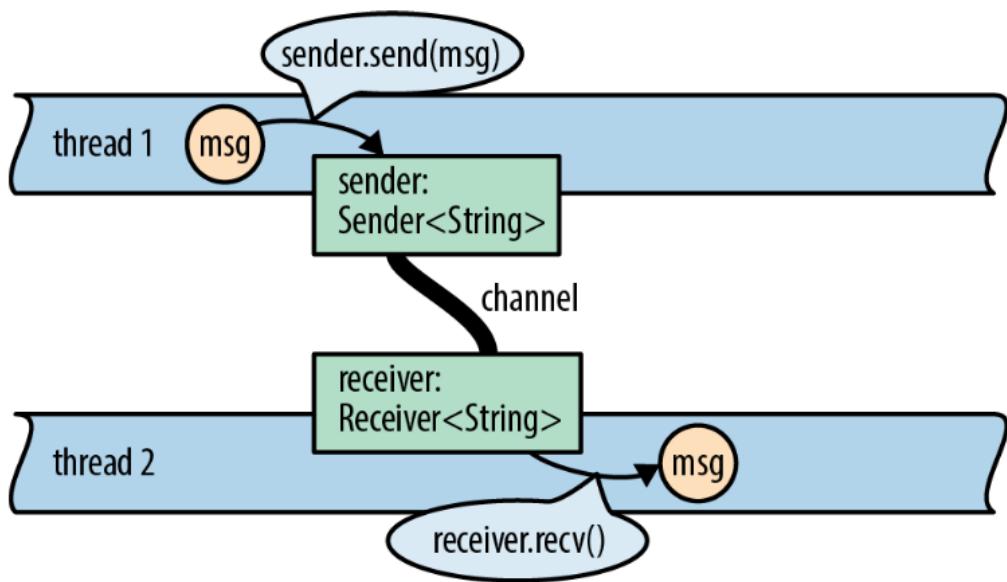


图 19-5: 一个 String 的通道: 字符串 msg 的所有权从线程 1 移动到线程 2

发送线程移动到接收线程。如果通道是空的，`receiver.recv()`会阻塞，直到有一个值被发送。

有了通道，不同的线程可以通过传递值来进行通信。这是一种非常简单的让线程在没有锁或共享内存的情况下协同工作的方法。

这并不是一种新技术。Erlang 有隔离进程和消息传递已经 30 年了。Unix 的管道大概已经 50 年了。我们更倾向于管道是一种提供灵活性和组合性的机制，而不是并发的机制，但事实上，它可以完成所有这些功能。Unix 管道的一个示例如图 19-6 所示。显然三个程序同时工作是可能的。

Rust 的通道比 Unix 的管道更快。发送一个值只是移动它而不是拷贝它，并且即使你要移动一个包含很多 MB 的数据结构也很快速。

19.2.1 发送值

在接下来的几节中，我们会使用通道构建一个创建倒排索引 (*inverted index*) 的并发程序，倒排索引是搜索引擎的一项关键技术。每一个搜索引擎工作在一个文档的集合上。倒排索引是一个指示哪个单词在哪里出现过的数据库。

我们将展示使用线程和通道的部分代码，[完整的程序](#)很短，总共只有大约 1 千行。

我们的程序被组织为流水线架构，如图 19-7 所示。流水线只是使用通道的很多方式中的一种——我们之后将会讨论少数其他的方式——但它们是一种直观地把并发引入现有的单线程程序的方式。

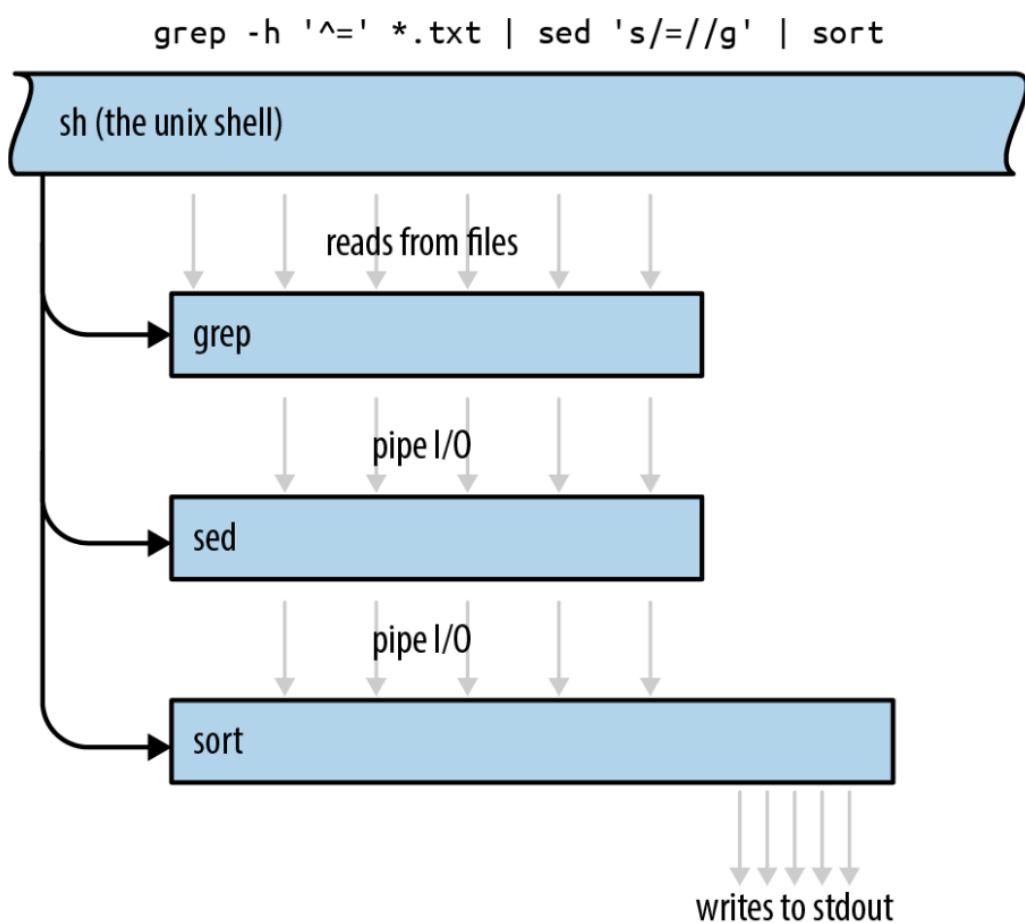


图 19-6: Unix 流水线的执行

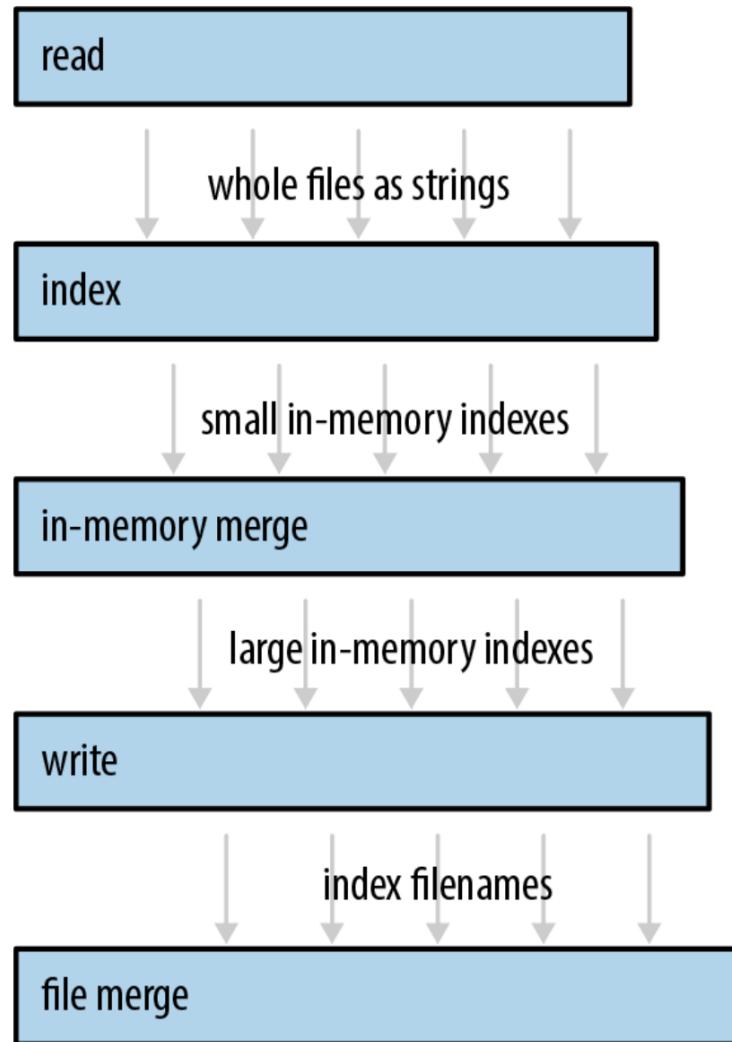


图 19-7: 构建索引的流水线, 箭头代表值通过通道被从一个线程送到另一个线程 (磁盘 I/O 没有显示出来)

我们总共将使用 5 个线程，每一个都做不同的任务。每一个线程都会在整个程序的生命周期中持续产生输出。例如第一个线程，简单地把源文档一个个地从磁盘读取到内存中。（我们用一个单独的线程来执行这个任务，是因为我们要使用 `fs::read_to_string` 来编写尽可能简单的代码，它是一个阻塞的 API。我们不想让 CPU 在磁盘工作时一直闲置。）这一阶段的输出是每一个文档转换成的长 `String`，因此这个线程和下一个线程通过一个 `String` 的通道连接。

我们的程序将以创建一个读取文件的线程开始。假设 `document` 是一个 `Vec<PathBuf>`，即文件名的 vector。启动读取文件的线程的代码如下：

```
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。我们稍后将解释这个名字的含义；首先，让我们看看这段代码怎么工作。我们首先创建一个通道：

```
let (sender, receiver) = mpsc::channel();
```

`channel` 函数返回一个值对：一个 `sender` 和一个 `receiver`。底层的队列数据结构是一个标准库没有暴露的实现细节。

通道是有类型的。我们将使用这个通道来发送每个文件的文本，因此我们有了一个 `Sender<String>` 类型的 `sender` 和一个 `Receiver<String>` 类型的 `receiver`。我们可以通过写 `mpsc::channel::<String>()` 来显式地要求一个字符串的通道。或者也可以让 Rust 的类型推导来判断它的类型。

```
let handle = thread::spawn(move || {
```

和之前一样，我们使用 `std::thread::spawn` 来启动一个线程。`sender` 的所有权通过 `move` 闭包移动到了新的线程（`receiver` 的所有权没有移动）。

接下来的几行代码简单地从磁盘读取文件：

```

for filename in documents {
    let text = fs::read_to_string(filename)?;

```

在成功读取一个文件之后，我们将它的文本送入通道：

```

if sender.send(text).is_err() {
    break;
}
}

```

`sender.send(text)` 把值 `text` 移动到通道里。最后，它被移动到接收值的线程里。不管 `text` 包含 10 行还是 10MB 文本，这个操作都只会拷贝三个机器字（一个 `String` 结构体的大小），相应的 `receiver.recv()` 调用也会拷贝三个机器字。

`send` 和 `recv` 方法都返回 `Result`，但这些方法只在通道的另一端被 `drop` 的情况下才会失败。如果 `Receiver` 被 `drop` 了，`send` 调用会失败，因为如果不这么做，这个值将会永远留在通道中：没有了 `Receiver`，就没有办法让任何线程接收它。类似的，如果通道中没有值并且 `Sender` 被 `drop` 了，`recv` 调用会失败，因为如果不这么做，`recv` 将会永远等待下去：没有了 `Sender`，就没有办法让任何线程发送下一个值。`drop` 通道的一端是通常的“挂断”方式，当你使用完它时用这种方法关闭连接。

在我们的代码中，只有当 `receiver` 的线程提前退出，`sender.send(text)` 才会失败。这是使用通道的代码的典型情况。不管这是故意的还是因为错误，我们的 `reader` 线程直接退出都是没有问题的。

当这种情况发生时，或者线程读取完所有文档后，它会返回 `Ok(())`：

```

Ok(())
});

```

注意这个闭包返回一个 `Result`。如果这个线程遇到了一个 I/O 错误，它会立即退出，并且错误会被存储在该线程的 `JoinHandle` 中。

当然，和其他编程语言一样，Rust 在错误处理方面允许很多其他的可能性。当错误发生时，我们可以简单地使用 `println!` 打印出它，然后继续下一个文件。我们可以通过正在用来传递数据的通道传递这个错误，只需要将它改成 `Result` 的通道——或者为错误创建第二个通道。我们在这里选择的方式既轻量又负责：我们使用了 `? 操作符`，因此不会有样板代码，也不会有你可能会在 Java 中见到的显式的 `try/catch`，并且错误不会被悄悄地忽略。

为了方便，我们的程序把这些代码包装为了一个返回 `receiver`（我们还没有使用过它）和新线程的 `JoinHandle` 的函数：

```

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)
}
```

注意这个函数启动了一个新线程然后立刻返回。我们将为流水线的每个阶段编写一个类似这样的函数。

19.2.2 接收值

现在我们有了一个循环发送值的线程。我们可以第二个线程循环调用 `receiver.recv()`:

```
while let Ok(text) = receiver.recv() {
    do_something_with(text);
}
```

但 `Receiver` 是可迭代对象，因此还有一种更好的写法：

```
for text in receiver {
    do_something_with(text);
}
```

这两个循环是等价的。不管用哪种方式写，如果当控制流到达循环时通道恰好为空，那么 `receiver` 线程会阻塞直到其他线程发送了值。当通道为空并且 `Sender` 被 `drop` 时循环会正常退出。在我们的程序中，当 `reader` 线程退出时就会很自然地变为这种情况。正在运行闭包的线程拥有变量 `sender`；当闭包退出时，`sender` 会被 `drop`。

现在我们可以编写流水线的第二个阶段：

```
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)
}

```

这个函数创建一个从其他线程接收 `String` 值 (`text`) 然后向另一个通道发送 `InMemoryIndex` 值的线程。这个线程的任务是接收第一个阶段加载的文件内容，并把每一个文档转换成一些内存中的倒排索引。

这个线程的主循环很直观。`InMemoryIndex::from_single_document` 函数负责完成所有索引文档的工作。我们并没有在这里展示它的代码，但它把输入的字符串切分为单词，并产生一个单词到其出现位置的映射。这个阶段没有 I/O 操作，因此它不需要处理 `io::Error`。所以它返回 `()` 而不是 `io::Result<()>`。

19.2.3 运行流水线

剩余的三个阶段在设计上很相似，每个阶段都消耗上一个阶段创建的 `Receiver`。流水线剩余部分的目标是把所有小索引合并成一个更大的索引，并保存为磁盘上的文件。我们发现最快的方式是以三个阶段来实现它。我们没有在这里给出代码，只是给出了这三个函数的签名。完整的代码可以在线获得。

首先，我们在内存中合并索引（阶段 3）：

```

fn start_in_memory_thread(file_indexes: mpsc::Receiver<InMemoryIndex>)
-> (mpsc::Receiver<InMemoryIndex>, thread::JoinHandle<()>)

```

然后我们把这些很大的索引写入磁盘（阶段 4）：

```

fn start_index_writer_thread(big_indexes: mpsc::Receiver<InMemoryIndex>,
                             output_dir: &Path)
-> (mpsc::Receiver<PathBuf>, thread::JoinHandle<io::Result<()>>)

```

最后，如果我们有了多个很大的文件，我们使用文件合并算法合并它们（阶段 5）：

```

fn merge_index_files(files: mpsc::Receiver<PathBuf>, output_dir: &Path)
-> io::Result<()>

```

这最后一个阶段不返回 `Receiver`，因为它是流水线的结尾。它产生磁盘上的单个输出文件。它也不返回 `JoinHandle`，因为我们不需要为这个阶段创建新线程，它的工作可以在调用者的线程完成。

现在就到了启动这些线程和检查错误的代码了：

```
fn run_pipeline(documents: Vec<PathBuf>, output_dir: PathBuf)
    -> io::Result<()>
{
    // 启动流水线的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);

    // 等待所有线程结束，处理它们可能遇到的所有错误。
    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。
    // (h2和h3不可能失败：那些线程只进行纯内存中的数据处理。)
    r1?;
    r4?;
    result
}
```

和之前一样，我们使用`.join().unwrap()`来显式地把子线程的 panic 传播到主线程。唯一不同的地方在于我们这里并没有直接使用`?`，而是保存了`io::Result`值，直到`join`了所有线程之后才处理`Result`值。

这个流水线比单线程的实现快了 40%。一下午的时间能做到做一点已经不错了，不过如果和我们之前在曼德勃罗集中取得的 675% 的加速相比，我们显然没有充分利用系统的 I/O 性能或者 CPU 核心。这是为什么呢？

流水线就像生产工厂中的装配线：整体的性能受限于最慢的阶段的吞吐量。在我们的例子中，测量结果表示第二个阶段是瓶颈。我们的索引线程使用了`.to_lowercase()`和`.is_alphanumeric()`，因此它需要很长时间在 Unicode 表中查找。其他的后续阶段把大部分时间浪费在`Receiver::recv`等待输入上。

这意味着我们应该能更快。因为我们定位到了瓶颈，因此并行度还可以提高。现在你已经知道了如何使用通道，我们的程序由很多独立的代码片段组成，很容易找到解决这个瓶颈的方法。我们可以优化第二个阶段，就像其他代码一样：把它的工作划分为两个或更多的阶段，或者同时运行多个文件索引线程。

19.2.4 通道的特性和性能

`std::sync::mpsc` 中的 `mpsc` 代表着多生产者，单消费者 (*multi-producer, single-consumer*)，简洁地描述了 Rust 的通道提供的这种通信方式。

我们的示例程序中的通道将值从单个发送者移动到单个接收者。这是非常常见的情况。但是 Rust 的通道还支持多个发送者的情况，即用单个线程处理来自多个客户端线程的请求，如图 19-8 所示。

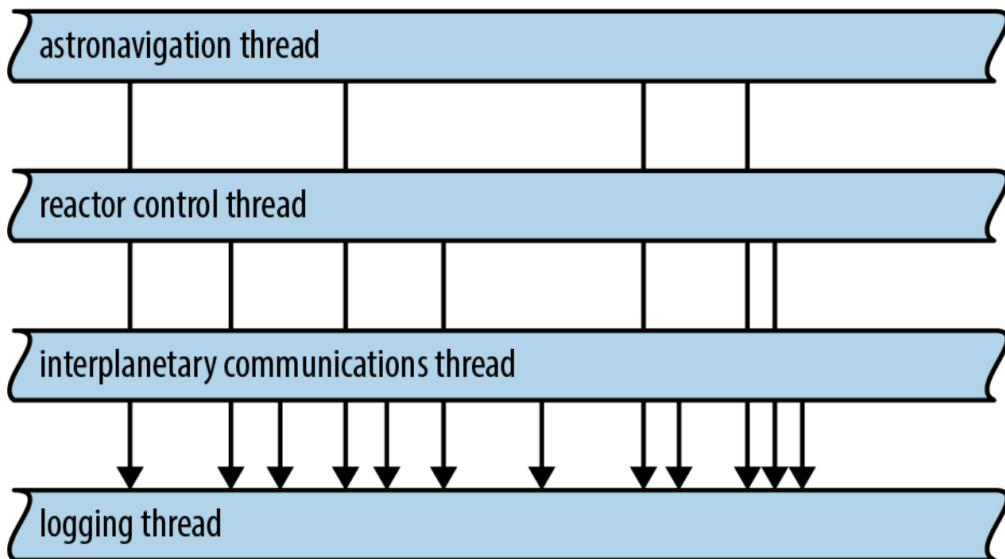


图 19-8: 从很多发送者接收请求的单个通道

`Sender<T>` 实现了 `Clone` trait。为了得到一个多个发送者的通道，简单地创建一个普通的通道然后随意克隆 `sender` 任意次。你可以将每个 `Sender` 值移动到不同的线程。

`Receiver<T>` 不能克隆，因此如果你需要多个线程接受来自单个通道的数据，你需要使用 `Mutex`。我们将在本章稍后展示如何使用它。

Rust 的通道被精心优化过。当通道第一次创建时，Rust 使用了一个特殊的“one-shot (一次性)”队列实现。如果你只通过通道发送一个对象，开销会非常小。如果发送了第二个值，Rust 会切换到不同的队列实现。它会转换为稳定的通道来传输多个值，同时最小化分配开销。如果你克隆了 `Sender`，Rust 必须转换为另一种实现，这种实现在多个线程同时发送值时也是安全的。但即使是这三种实现中最慢的一种也是一个无锁的队列，因此发送或接收值最多就是一些原子操作和一次堆分配，加上移动它的开销。只有当队列为空并且接收线程因此需要睡眠时才会调用系统调用。当然，在这种情况下，通道无论如何都不会限制吞吐量。

除了这些优化工作之外，应用程序很容易犯一个有关通道性能的错误：发送值的速度比接收和处理值更快，这可能会导致通道中累积的值不断增长。例如，在我们的程序中，我们发

现文件读取线程（阶段1）加载文件可能比索引线程（阶段2）构建索引更快。结果就是数百个从磁盘中读取的MB大小的原始数据同时堆积在队列中。

这种错误不仅会消耗内存还会破坏局部性。更糟的是，发送线程在其他线程最需要资源的时候持续运行，耗尽了CPU和其他的系统资源来发送更多值。

这里Rust再次参考了Unix管道。Unix使用了一个优雅的技巧来提供 *backpressure*(背压)，这样快速的发送者会被迫减速：Unix中的每个管道有一个固定大小，如果一个进程尝试写入一个已经满了的管道，系统会简单地阻塞这个进程，直到管道中有空间。Rust中等价的组件是同步通道 (*synchronous channel*)：

```
use std::sync::mpsc;  
  
let (sender, receiver) = mpsc::sync_channel(1000);
```

同步通道实际上就是普通的通道，除了当你创建它时可以指定它可以存储多少值。对于一个同步通道，`sender.send(value)`是一个阻塞操作。毕竟，设计它的idea是阻塞并不是一件坏事。在我们的示例程序中，将`start_file_reader_thread`中的`channel`修改为能存储32个值的`sync_channel`可以在保证吞吐量不变的情况下减少 $\frac{2}{3}$ 的内存占用。

19.2.5 线程安全：`Send` 和 `Sync`

目前为止我们操作的所有值都可以自由地在线程间移动和共享。大部分情况都是这样，但Rust的完整的线程安全取决于两个内建的trait：`std::marker::Send` 和 `std::marker::Sync`。

1. 实现了`Send`的类型可以安全地以值传递到另一个线程。它们可以在线程之间移动。
2. 实现了`Sync`的类型可以安全地以非`mut`引用传递到另一个线程。它们可以在线程之间共享。

这里的安全 (*safe*) 指的是我们通常所说的含义：没有数据竞争和其它未定义行为。

例如，在`process_files_in_parallel`示例中，我们使用了一个闭包来从父线程给每个子线程传递一个`Vec<String>`。我们当时并没有指出，但这意味着`vector`和它的字符串都是在父线程中分配，但却在子线程中释放。事实是`Vec<String>`实现了`Send`，它是一种承诺这样做没问题的API：`Vec`和`String`内部使用的分配器是线程安全的。

（如果你想基于更快但不是线程安全的分配器实现你自己的`Vec`和`String`类型，你可能需要使用不是`Send`的类型来实现它们，例如`unsafe`指针。然后Rust会推断出你的`NonThreadSafeVec`和`NonThreadSafeString`类型不是`Send`，并限制它们只能在单线程中使用。但这种情况很少见。）

如图19-9所示，大多数类型都是`Send`和`Sync`。你甚至不需要使用`#[derive]`来为你的结构体和枚举实现它们。Rust会自动实现。如果结构体或枚举的所有字段都是`Send`/`Sync`，那么

它们也是 Send/Sync。

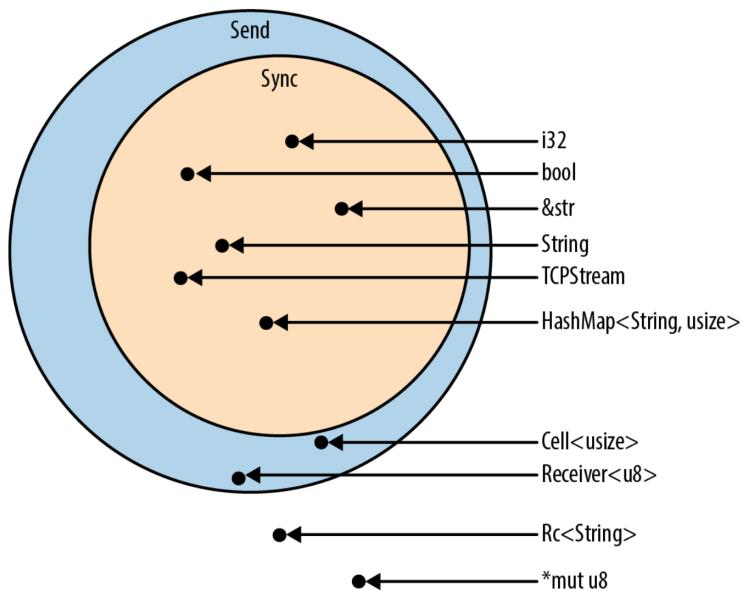


图 19-9: Send 和 Sync 类型

一些类型是 Send，但不是 Sync。这通常是故意的，就像 `mpsc::Receiver` 的例子中，它保证 `mpsc` 通道的接收端同时只能被一个线程使用。

那些少数既不是 Send 又不是 Sync 的类型通常是以线程不安全的方式实现可变性的类型。例如 `std::rc::Rc<T>`，它是引用计数的智能指针类型。

如果 `Rc<String>` 是 Sync、允许多个线程通过共享引用共享一个 `Rc` 会发生什么？如果有多个线程恰好同时克隆 `Rc`，如图 19-10 所示，就会出现数据竞争，因为有多个线程同时想递增引用计数。引用计数就可能变得不精确，进而导致之后出现释放后使用或两次释放——即未定义行为。

当然，Rust 会阻止这种情况。这里有一段产生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 || { // error
        rc2.clone();
    });
    rc1.clone();
}
```

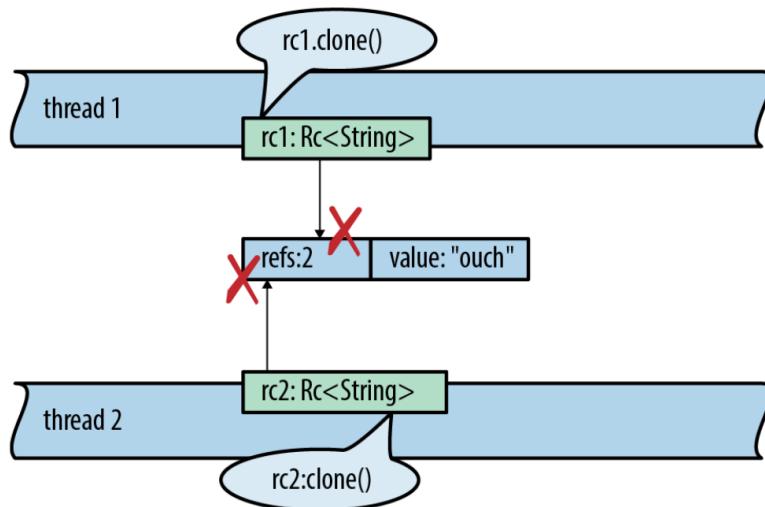


图 19-10: 为什么 `Rc<String>` 既不是 `Sync` 也不是 `Send`

Rust 拒绝编译它，并给出了详细的错误消息：

```
error[E0227]: `Rc<String>` cannot be sent between threads safely
--> concurrency_send_rc.rs:10:5
|
10 |     thread::spawn(move || { // error
|           ^^^^^ `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`
```

现在你可以看到 `Send` 和 `Sync` 如何帮助 Rust 强制线程安全性。它们作为在线程间传递数据的函数签名中的类型约束出现。当你 `spawn` 一个线程时，你传递的闭包必须是 `Send`，这意味着它包含的所有值也必须是 `Send`。类似的，如果你想通过通道给另一个线程发送值，那么这个值必须是 `Send`。

19.2.6 使用管道串联迭代器

我们的倒排索引构造器是按照流水线的方式构建的。代码虽然足够清楚，但它需要我们手动设置通道并启动线程。相反，我们在第 15 章中构建的迭代器流水线似乎在短短几行代码中完成了更多工作。我们能构建类似那样的线程流水线吗？

事实上，如果能统一迭代器流水线和线程流水线那将会非常 nice。这样我们的构造器就可以被写成迭代器流水线的形式。它可能看起来像这样：

```

documents.into_iter()
    .map(read_whole_file)
    .errors_to(error_sender) // 过滤掉错误的结果
    .off_thread()           // 创建一个线程在完成上述工作
    .map(make_single_file_index)
    .off_thread()           // 创建另一个线程执行阶段 2
    ...

```

trait 允许我们给标准库类型添加方法，所以我们确实可以做到这一点。我们首先编写一个声明了我们想要的方法的 trait：

```

use std::sync::mpsc;

pub trait OffThreadExt: Iterator {
    /// 将这个迭代器转换成为一个 off-thread 迭代器：
    /// `next()` 调用发生在一个单独的工作线程里，因此
    /// 迭代器和循环体会并发运行。
    fn off_thread(self) -> mpsc::IntoIter<Self::Item>;
}

```

然后我们为迭代器类型实现这个 trait。`mpsc::Receiver` 本身就是可迭代对象这一点帮了我们不少：

```

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> {
        // 创建一个通道从工作线程里传输数据。
        let (sender, receiver) = mpsc::sync_channel(1024);

        // 把这个迭代器移动到一个新的工作线程里并在这里运行它。
        thread::spawn(move || {
            for item in self {
                if sender.send(item).is_err() {
                    break;
                }
            }
        });
    }

    // 返回一个从通道中取出值的迭代器
}

```

```
    receiver.into_iter()
}
}
```

这段代码中的 `where` 子句是通过类似于我们在 [逆向工程约束](#) 中描述过的方式决定的。一开始，我们只写了：

```
impl<T> OffThreadExt for T
```

就是说，我们想为所有的迭代器都实现它。Rust 会报错。因为我们使用 `spawn` 来把一个 `T` 类型的迭代器移动到新线程，我们必须指明 `T: Iterator + Send + 'static`。因为我们通过通道发送 `item`，所以我们必须指明 `T::Item: Send + 'static`。修改之后，Rust 才会满意。

这就是 Rust 的角色：我们可以放心地为语言中几乎每一种迭代器添加一个强大的并发工具——而且不需要去了解保证安全所需的约束。

19.2.7 超越流水线

在这一节中，我们使用了流水线作为示例程序，因为流水线是一种非常 nice 的使用通道的方式。每个人都可以理解它们。它们是具体的、实用的、确定性的。通道不仅仅在流水线中非常有用。它们还是一种快速、方便地为同一个进程中的其他线程提供任何异步服务的方式。

例如，假设在 [图 19-8](#) 中你想在专门的线程中进行日志工作。其他的线程可以通过一个通道给日志线程发送日志消息；因为你可以克隆通道的 `Sender`，所以很多客户端线程可以持有发送到同一个日志线程的 `sender`。

在单独的线程中运行类似日志这样的服务有很多优势。日志线程可以在需要时轮换日志文件。它不需要和其他线程进行任何的协调，那些线程也不会被阻塞。消息将在通道中无害地累积一段时间，直到日志线程返回来处理。

通道也可以用于一个线程给另一个线程发送请求，并需要得到某种响应的场景。第一个线程的请求可以是一个包含一个 `Sender` 和一些标识自己的信息的结构体或元组，第二个线程可以使用它们发送响应。这并不意味着这种交互必须是同步的，第一个线程可以选择阻塞并等待响应，或者使用 `.try_recv()` 方法进行轮询。

我们目前已经展示过的工具——`fork-join` 用于高度并行化的计算，通道用于弱连接的组件——对非常多的应来说已经足够了。但我们还没有结束。

19.3 共享可变状态

自从你在 [第 8 章](#) 中发布 `fern_sim` crate 之后，你的蕨类模拟器已经确实起飞了。现在你要创建一个多人实时战略游戏，游戏里 8 位玩家竞争在模拟侏罗纪景观中种植大部分那个时期

的蕨类植物。这个游戏的服务器是一个大规模的并行应用程序，在很多线程中处理请求。我们怎么协调这些请求以让它们在 8 位玩家到齐之后立刻开始游戏呢？

这里要解决的问题是很多线程需要能访问一个共享的等待加入游戏的玩家列表。这个数据必须是既是可变的又能在线程之间共享。如果 Rust 没有共享可变状态，这让我们怎么办？

你可以创建一个新线程来专门管理这个列表，其他线程都通过通道和它通信，这确实可以解决问题。但是当然，这会多一个线程，会有一些操作系统的开销。

另一种选择是使用 Rust 提供的用于安全的共享可变数据的工具。这样的工具确实存在，它们是一些底层的原语，任何曾经使用过线程的系统程序员应该都很熟悉。在这一节中，我们将介绍互斥锁 (mutex)，读/写锁 (read/write lock)，条件变量 (condition variable)，和原子整数 (atomic integer)。最后，我们将展示如何在 Rust 中实现全局的可变变量。

19.3.1 互斥锁是什么？

mutex(或 *lock*) 用于强迫多个线程在访问某些数据时按照顺序访问。我们将在下一节介绍 Rust 的互斥锁。首先，有必要回顾一下互斥锁在其他语言中是什么样的。在 C++ 中一个简单的互斥锁的使用示例看起来是这样的：

```
// C++ 代码, 不是 Rust
void FernEmpireApp::JoinWaitingList(PlayerId player) {
    mutex.Acquire();

    waitingList.push_back(player);

    if (waitingList.size() >= GAME_SIZE) {
        vector<PlayerId> players;
        waitingList.swap(players);
        StratGame(players);
    }

    mutex.Release();
}
```

`mutex.Acquire()` 和 `mutex.Release()` 标志着这段代码中临界区 (*critical section*) 的开始和结束。对一个程序中的每个 `mutex`，任意时间只能有一个线程可以在临界区内运行。如果一个线程在临界区内，所有其他调用 `mutex.Acquire()` 的线程都会阻塞，直到第一个线程到达 `mutex.Release()`。

我们说互斥锁保护 (*protect*) 了数据：在这个例子中，`mutex` 保护了 `waitingList`。确保每一个线程访问数据之前总是先获取锁，并在之后释放锁是程序员的责任。

互斥锁有以下作用：

1. 它们阻止了数据竞争 (*data race*)，即多个竞争线程同时读写相同的内存地址。在 C++ 和 Go 中数据竞争是未定义行为。托管语言例如 Java 和 C# 保证不会崩溃，但数据竞争的结果（总的来说）仍然是无意义的。
2. 即使没有数据竞争，即使所有的读写都恰好按照顺序一个一个来，如果没有互斥锁，不同线程的操作也可能会以任意方式交错。想象尝试编写一段即使其他线程修改了它的数据也能工作的代码。想象尝试调试它。就像你的程序正在闹鬼一样。
3. 互斥锁支持使用不变量 (*invariant*) 进行编程，它是一种有关受保护数据的规则，在您构造数据时它为真，并在每个临界区结束后都保持为真。

当然，所有这些作用实际上都是出自同一个原因：不受控制的数据竞争使程序变得不可控。互斥锁为这种混乱带来了某种顺序（尽管比通道或 fork-join 带来的顺序要少）。

然而，在大多数语言中，互斥锁非常容易出错。在 C++ 和其他大多数语言中，数据和锁是单独的对象。理想情况下，注释解释说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。由`mutex`保护。  
        vector<PlayerId> waitingList;  
  
        // 在读取或写入`waitingList`之前需要获取的锁。  
        Mutex mutex;  
}
```

但即使有了这么好的注释，编译器也不能强迫任何安全的访问。当一段代码忘记获取互斥锁时，会导致未定义行为。在实践中，这种 bug 非常难复现和修复。

即使在 Java 中对象和互斥锁之间有一些概念上的关联，但这种关系也不是很深。编译器也没有尝试强迫它，并且在实践中，被锁保护的数据很少恰好是关联对象的字段。它通常包含多个对象里的数据。上锁的策略仍然很棘手。注释仍然是主要的强迫执行规则的工具。

19.3.2 Mutex<T>

现在我们将展示在 Rust 实现等待列表的一个例子。在我们的孢子帝国游戏服务器中，每一个玩家都有一个独一无二的 ID：

```
type PlayerId = u32;
```

等待列表只是一个玩家的集合：

```
const GAME_SIZE: usize = 8;

/// 一个等待列表的长度永远不会超过 GAME_SIZE
type WaitingList = Vec<PlayerId>;
```

等待列表被存储为 `FernEmpireApp` 的一个字段，在服务器启动时会在一个 `Arc` 中实例化 `FernEmpireApp` 单例。每一个线程都有一个指向它的 `Arc`。它包含我们程序所需的所有共享配置和其他细节。里面的大部分内容都是只读的。因为等待列表既是共享的又是可变的，所以它必须用一个 `Mutex` 保护起来：

```
use std::sync::Mutex;

/// 所有线程都能访问这个大的上下文结构体
struct FernEmpireApp {

    ...
    waiting_list: Mutex<WaitingList>,
    ...
}
```

和 C++ 不同，Rust 中被保护的数据存储在 `Mutex` 的内部。`Mutex` 的初始化看起来像这样：

```
use std::sync::Arc;

let app = Arc::new(FernEmpireApp {
    ...
    waiting_list: Mutex::new(vec![]),
    ...
});
```

创建一个新的 `Mutex` 看起来就像创建一个新的 `Box` 或者 `Arc`，但 `Box` 和 `Arc` 意味着有堆分配，而 `Mutex` 只和锁有关。如果你想让你的 `Mutex` 在堆上分配，你必须显式要求，就像我们这里一样对整个 app 使用 `Arc::new`，对要保护的数据使用 `Mutex::new`。这些类型通常会在一起使用：`Arc` 可以方便地在线程间共享内容，`Mutex` 可以方便地在线程间共享可变的数据。

现在我们可以使用互斥锁实现 `join_waiting_list` 方法：

```
impl FernEmpireApp {
    /// 向下一场游戏的等待队列中添加一个玩家。
    /// 如果有足够的玩家在等待就立刻开始新游戏。
    fn join_waiting_list(&self, player: PlayerId) {
        // 锁住互斥锁并获取里面数据的访问。
        // `guard` 的作用域就是一个临界区。
        let mut guard = self.waiting_list.lock().unwrap();
```

```
// 现在进行游戏的逻辑。
guard.push(player);
if guard.len() == GAME_SIZE {
    let players = guard.split_off(0);
    self.start_game(players);
}
}
```

获取内部数据的唯一方式是调用`.lock()`方法：

```
let mut guard = self.waiting_list.lock().unwrap();
```

`self.waiting_list.lock()`会阻塞，直到获取到互斥锁。这个方法返回的`MutexGuard<WaitingList>`是一个`&mut WaitingList`的简单包装。得益于强制解引用，我们可以直接对`guard`调用`WaitingList`方法：

```
guard.push(player);
```

`guard`甚至允许我们借用底层数据的直接引用。Rust 的生命周期系统会保证这些引用的生命周期不能超过`guard`本身。在不获取锁的情况下，没有任何方法获取`Mutex`内的数据。

当`guard`被`drop`时，锁会被释放。一般会发生在块的结尾，不过你也可以手动`drop`它：

```
if guard.len() == GAME_SIZE {
    let players = guard.split_off(0);
    drop(guard); // 在启动游戏的同时不保持列表被锁定
    self.start_game(players);
}
```

19.3.3 `mut` 和 `Mutex`

有一件事可能很奇怪（第一眼看上去确实会觉得很奇怪），就是我们的`join_waiting_list`方法并不是以`mut`引用获取`self`参数。它的类型签名是：

```
fn join_waiting_list(&self, player: PlayerId)
```

而底层的集合`Vec<PlayerId>`确实需要`mut`引用来调用它的`push`方法。它的类型签名是：

```
pub fn push(&mut self, item: T)
```

然而这段代码确实可以编译并正确运行。到底是怎么回事？

在 Rust 中，`&mut` 意味着独占性访问 (*exclusive access*)。普通的`&`意味着共享访问 (*shared access*)。

我们已经习惯了从父对象到子对象、从容器到内容传递 `&mut` 访问的类型。你可能觉得只有当你有 `starships` 的 `&mut` 引用时，才能对 `starships[id].engine` 调用 `&mut self` 方法。这是默认的情况，因为如果你没有对父对象的独占性访问，Rust 通常不能确保你有对子对象的独占访问。

但 `Mutex` 确实提供了一种方法：锁。事实上，为了能在很多线程都有 `Mutex` 自身的共享（非 `mut`）访问的情况下，仍然能提供对内部数据的独占（`mut`）访问，`mutex` 做的事情还要更多一点。

Rust 的类型系统告诉了我们 `Mutex` 做了什么。它动态地强制独占性访问，而这一点通常是由 Rust 的编译器在编译期静态完成的。

（你可能会回想起来 `std::cell::RefCell` 做了同样的事，除了并不支持多线程。`Mutex` 和 `RefCell` 都是我们之前介绍过的内部可变性的体现。）

19.3.4 为什么有时互斥锁不是好方案

在我们开始互斥锁之前，我们提到过一些并发的方法，如果你是从 C++ 过来的那你可能会感觉它们很容易正确使用。这并非巧合：这些方法旨在为并发编程中最令人困惑的方面提供强有力的保证。只使用 fork-join 并行的程序是确定性的，不可能死锁。只使用通道来实现流水线的程序，例如我们的索引构建器，也是确定性的：消息传递的时机可能不同，但并不会影响输出。多线程程序的安全保证非常 nice！

Rust 的 `Mutex` 的设计几乎肯定会让你比以往更系统、更明智地使用互斥锁。但停下来思考一下 Rust 的安全性保证能做什么、不能做什么是值得的。

安全的 Rust 代码不可能触发数据竞争 (*data race*)，它是一种当多个线程同时读写相同的内存时的 bug，会导致无意义的结果。这非常棒：数据竞争一定是 bug，并且在真实的多线程程序中它们很常见。

然而，使用互斥锁的线程可能还会遇到一些其他 Rust 无法解决的问题：

1. 有效的 Rust 程序不可能有数据竞争，但仍然可能有其他的竞争条件 (*race condition*)——程序的行为依赖于多个线程的执行顺序，并且因此导致每一次运行结果都不同。一些竞争条件是良性的，另一些可能是非常难修复的 bug。以非结构化的方式使用互斥锁会引发竞争条件。你需要确保它们是良性的。
2. 共享的可变状态也会影响程序的设计。通道作为代码中的抽象边界，让你更容易的隔离开不同的组件。而互斥锁鼓励“just-add-a-method”的工作方式，这可能导致相关的代码变成一个大块。
3. 最后，互斥锁并不像它第一眼看上去那么简单，正如接下来两节所示。

这些问题都是互斥锁工具固有的。如果可以的话请使用更加结构化的方式，只有当必要时才使用 `Mutex`。

19.3.5 死锁

如果一个线程尝试获取一个已经持有的锁，那么可能会导致死锁：

```
let mut guard1 = self.waiting_list.lock().unwrap();  
let mut guard2 = self.waiting_list.lock().unwrap(); // 死锁
```

假设对 `self.waiting_list.lock()` 的第一次调用成功获取到了锁。第二次调用发现锁已经被获取了，所以它会阻塞住，等待锁被释放。它将永远等待下去。等待的线程就是持有锁的线程。

换句话说，`Mutex` 里的锁不是可递归的锁。

这个 bug 很明显。在真实的程序中，两个 `lock()` 调用可能发生在两个不同的方法中，其中一个调用另外一个。每一个方法单独看起来都没什么问题。还有一些别的导致死锁的方式：在有多个线程的情况下，每个线程一次获取多个互斥锁。Rust 的借用系统并不能帮助你避免死锁。最好的防护方法就是就是让临界区尽可能小：进入、做工作、然后离开。

使用通道也可能导致死锁。例如，两个线程可能阻塞住，因为它们都在等待从另一个线程中接收消息。然而，好的程序设计可以确保不会发生死锁。在流水线中数据流是非循环的，例如我们的倒排索引构造器。死锁不太可能出现在这样的程序中，正如 Unix shell 管道一样。

19.3.6 中毒的互斥锁

`Mutex::lock()` 返回一个 `Result`，原因和 `JoinHandle::join()` 返回 `Result` 是一样的：如果另一个线程 `panic` 就正常失败。当我们写 `handle.join().unwrap()` 时，我们是在告诉 Rust 把 `panic` 从一个线程传播到另一个线程。`mutex.lock().unwrap()` 也类似。

如果一个线程在持有一个 `Mutex` 的时候 `panic`，Rust 会标记这个 `Mutex` 中毒了 (*poisoned*)。任何后续尝试 `lock` 中毒的 `Mutex` 的操作都会得到错误的结果。我们的 `.unwrap()` 调用告诉 Rust 如果发生了这种情况，就把 `panic` 从另一个线程传播到这个线程。

中毒的互斥锁有多糟糕？中毒听起来是致命的，但实际上这种情况不一定是致命的。正如我们在第 7 章所说，`panic` 是安全的。就算有一个线程 `panic` 程序的其他部分仍然处于安全的状态。

互斥锁在 `panic` 时候中毒的原因，并不是为了防止未定义行为。相反，真正担心的是您可能一直在使用不变量进行编程。因为你的程序 `panic` 了并且导致某个临界区内并没有完成正在做的工作，可能已经更新了被保护数据的一部分字段，但另一部分还没有更新，这可能会

破坏不变量。Rust 让互斥锁中毒是为了防止其他线程不知不觉中陷入这种错误的情况然后让情况变得更糟。通过完全强制互斥，你仍然可以锁住一个中毒的互斥锁，并访问里面的数据，见 `PoisonError::into_inner()` 的文档。但你不可能无意中做到这一点。

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，单消费者。或者更精确地说，一个通道只能有一个 `Receiver`。我们不能创建一个线程池，并用单个 `mpsc` 通道作为共享的工作队列。

然而，事实证明就算只使用标准库的组件，也可以很简单的解决这个问题。我们可以给 `Receiver` 包装上一个 `Mutex` 并且随意共享它。这里有一个这么做的模块：

```
pub mod shared_channel {
    use std::sync::{Arc, Mutex};
    use std::sync::mpsc::{channel, Sender, Receiver};

    /// `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;

        /// 从包装的 receiver 获取下一个 item
        fn next(&Mut self) -> Option<T> {
            let guard = self.0.lock().unwrap();
            guard.recv().ok()
        }
    }

    /// 创建一个 receiver 可以被跨线程共享的通道。
    /// 和标准库的 `channel()` 一样返回一个 sender 和一个 receiver,
    /// 并且有时可以替换标准库的通道正常工作。
    pub fn shared_channel<T>() -> (Sender<T>, SharedReceiver<T>) {
        let (sender, receiver) = channel();
        (sender, SharedReceiver(Arc::new(Mutex::new(receiver))))
    }
}
```

我们正在使用一个 `Arc<Mutex<Receiver<T>>>`。泛型嵌套的有点多。这种情况 Rust 中比在 C++ 中出现的更频繁。看起来这可能会让人困惑，不过通常情况下，就和这里一样，只需要阅读这些名字就可以解释它到底是什么，如图 19-11 所示。

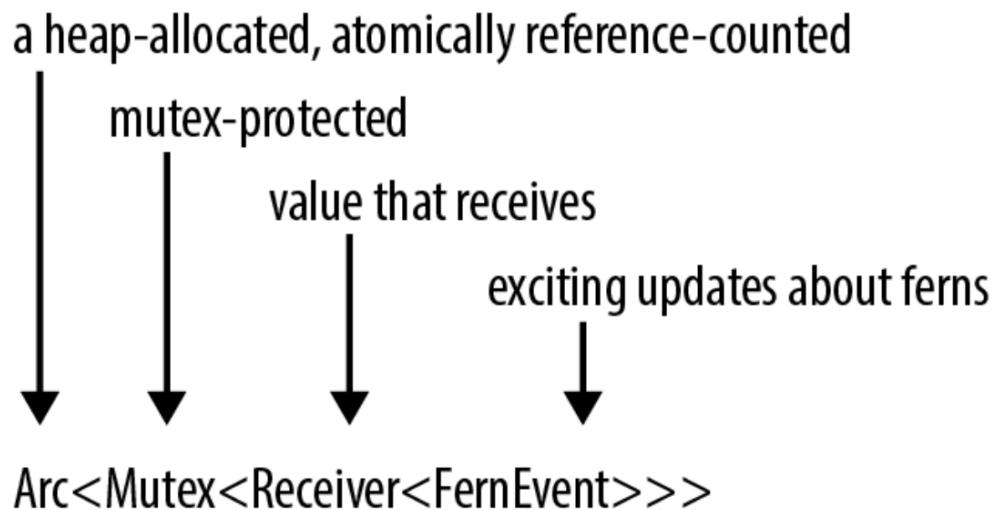


图 19-11: 如何阅读一个复杂的类型

19.3.8 读写锁 (`RwLock<T>`)

现在让我们从互斥锁转到 `std::sync` 里提供的其他工具。我们将会快速地过一遍，因为对这些工具的详细讨论超出了本书的范围。

服务器程序通常有一些只加载一次并且很少会更改的配置信息。大多数线程只会查询配置，但因为配置有可能改变——例如有可能要求服务器从磁盘上重新加载配置——所以它必须用锁保护。在类似这样的情况下，互斥锁可以工作，但会导致不必要的瓶颈。当它不是正在被修改时线程不需要轮流查询配置。这是读写锁即 `RwLock` 的一个使用场景。

互斥锁只有一个 `lock` 方法，而读写锁有两个上锁方法：`read` 和 `write`。`RwLock::write` 方法类似于 `Mutex::lock`。它等待被保护数据的独占、可变的访问。而 `RwLock::read` 方法提供不可变的访问，但它的优势是不一定需要等待，因为多个线程可以安全地同时读。如果使用互斥锁，在任意时刻被访问的数据都只能有一个 reader 或 writer（或者一个也没有）。而使用读写锁，可以有一个 writer 或者多个 reader，和 Rust 的引用非常相似。

`FernEmpireApp` 可能有一个用于配置的结构体，用一个 `RwLock` 进行保护：

```
use std::sync::RwLock;

struct FernEmpireApp {
    ...
    config: RwLock<AppConfig>,
    ...
}
```

读取配置的方法可以使用 `RwLock::read()`：

```
// True if experimental fungus code should be used
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 非常适合在 `RwLock` 数据上强制执行安全规则。“单 writer 或多 reader”是 Rust 的借用系统的核心概念。`self.config.read()` 返回一个提供 `AppConfig` 的非 `mut` (共享) 访问的 `guard`; `self.config.write()` 返回一个不同的提供 `mut` (独占) 访问的 `guard`。

19.3.9 条件变量 (Condvar)

通常一个线程需要等待, 直到某一个确定的条件变为真:

1. 在服务器关机时, 主线程需要等待所有其他线程完成退出。
2. 当一个工作线程没有任务可做的时候, 它需要等待需要处理的数据。
3. 一个实现了分布式共识协议的进程可能需要等待法定人数的 peer 响应。

有时, 有一些方便的阻塞 API 可以用于某些我们需要等待的精确条件, 例如

`JoinHandle::join` 可以用于服务器关机的例子。在其他情况中, 没有内建的阻塞 API。程序可以使用条件变量 (*condition variable*) 来构建自己的 API。在 Rust 中, `std::sync::Condvar` 类型实现了条件变量。一个 `Condvar` 有 `.wait()` 和 `.notify_all()` 方法; `.wait()` 会阻塞直到某些其他线程调用了 `.notify_all()`。

真实的情况还要更复杂一些, 因为条件变量总是与某个特定的 `Mutex` 保护的某些数据相关的条件。因此 `Mutex` 和 `Condvar` 是相关的。我们没有足够的篇幅进行完整的解释, 但为了之前使用过条件变量的程序员, 我们将展示代码中的两个关键部分。

当期望的条件变为真时, 我们会调用 `Condvar::notify_all` (或者 `notify_one`) 唤醒正在等待的线程:

```
self.has_data_condvar.notify_all();
```

为了睡眠并等待一个条件变为真, 我们要使用 `Condvar::wait()`:

```
while !guard.has_data() {
    guard = self.has_data_condvar.wait(guard).unwrap();
}
```

`while` 循环是一种使用条件变量的标准写法。然而，`Condvar::wait` 的签名并不寻常。它以值获取一个 `MutexGuard`，消耗它，并在成功时返回一个新的 `MutexGuard`。这隐含着 `wait` 方法先释放互斥锁然后在返回之前重新获取它。以值传递 `MutexGuard` 像是在说“我把我释放互斥锁的独占性权限赋予你——`.wait()` 方法”。

19.3.10 原子量

`std::sync::atomic` 模块包含用于无锁并发编程的原子类型。这些类型基本和标准的 C++ 原子类型相同，加上一些额外的类型：

1. `AtomicIsize` 和 `AtomicUsize` 是和单线程的 `isize` 和 `usize` 类型相对应的共享整数类型。
2. `AtomicI8`, `AtomicI16`, `AtomicI32`, `Atomic64` 以及它们的无符号变体例如 `AtomicU8` 是和单线程的 `i8`, `i16` 等相对应的共享整数类型。
3. `AtomicBool` 是共享的 `bool` 值。
4. `AtomicPtr<T>` 是共享的 unsafe 指针类型 `*mut T`。

原子类型的正确使用超出了这本书的范围。简单来说多个线程可以同时读写一个原子值并且不会有数据竞争。

作为普通的算术和逻辑运算符的代替，原子类型暴露了一些方法来进行原子操作 (*atomic operation*)：单独的 `load`、`store`、`exchange` 和算术运算，它们可以安全地进行操作，即使有其他线程正在对相同的内存位置进行原子操作。递增一个命名为 `atom` 的 `AtomicIsize` 看起来像这样：

```
use std::sync::atomic::{AtomicIsize, Ordering},

let atom = AtomicIsize::new(0);
atom.fetch_add(1, Ordering::SeqCst);
```

这些方法可能会编译成特定的机器语言指令。在 x86-64 架构上，这个 `.fetch_add()` 调用会编译成一个 `lock incq` 指令，而普通的 `n += 1` 可能会编译成一个普通的 `incq` 指令。Rust 编译器还必须放弃一些围绕原子操作的优化，因为——和普通的 `load` 或 `store` 不同——它可能立即合法地影响或受其他线程的影响。

参数 `Ordering::SeqCst` 是一个内存顺序 (*memory ordering*)。内存顺序类似于数据库中的事务隔离级别。它们告诉系统你有多关心这些概念，而不是性能。内存顺序对程序的正确性至关重要，而且它们很难理解和推理。幸运的是，即使选择顺序一致性（最严格的内存顺序）

的性能惩罚也通常很小——不像 SQL 数据库中 SERIALIZABLE 模式的性能惩罚。因此如果有疑问，就使用 `Ordering::SeqCst`。Rust 还从标准 C++ 原子量继承了其他几个内存顺序，它们对 nature of existence 和因果关系有各种更弱的保证。我们就不在这里讨论了。

原子量的一个简单用途是取消操作。假设我们有一个线程要做长时间的计算，例如渲染一个视频，并且我们想能异步地取消它。问题在于和我们想要停止的线程交互。我们可以通过一个共享的 `AtomicBool` 来实现：

```
use std::sync::Arc;
use std::sync::atomic::AtomicBool;

let cancel_flag = Arc::new(AtomicBool::new(false));
let worker_cancel_flag = cancel_flag.clone();
```

这段代码创建了两个 `Arc<AtomicBool>` 智能指针，它们指向同一个在堆上分配的 `AtomicBool`，它的初始值是 `false`。第一个命名为 `cancel_flag`，它会留在主线程。第二个命名为 `worker_cancel_flag`，将会被移动到工作线程。

这里是工作线程的代码：

```
use std::thread;
use std::sync::atomic::Ordering;

let worker_handle = thread::spawn(move || {
    for pixel in animation.pixels_mut() {
        render(pixel); // 光线追踪，这需要几毫秒
        if worker_cancel_flag.load(Ordering::SeqCst) {
            return None;
        }
    }
    Some(animation)
});
```

在渲染了每一个像素之后，线程会调用 `.load()` 方法检查标记的值：

```
worker_cancel_flag.load(Ordering::SeqCst)
```

如果在主线程中我们决定取消工作线程，我们可以在这个 `AtomicBool` 中 store `true`，然后等待这个线程退出：

```
// 取消渲染
cancel_flag.store(true, Ordering::SeqCst);

// 丢弃结果，可能是`None`
worker_handle.join().unwrap();
```

当然，还有其他的实现方式。这里的 `AtomicBool` 可以用一个 `Mutex<bool>` 或者一个通道替换。主要的不同在于原子量的开销最小。原子操作从不会使用系统调用。一个 `load` 或 `store` 操作通常会编译为单条 CPU 指令。

原子量是另一种形式的内部可变性，类似 `Mutex` 或者 `RwLock`，因此它们的方法以共享（非 `mut`）引用获取 `self`。这使得它们可以用作简单的全局变量。

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：每处理一个包就递增一次的计数器：

```
//> 服务器已经成功处理的包。  
static PACKETS_SERVED: usize = 0;
```

这可以正常编译。只有一个问题。`PACKETS_SERVED` 不是可变的，因此我们永远不能修改它。

Rust 尽其所能阻止全局可变状态。使用 `const` 声明的变量当然是不可变的。静态变量默认也是不可变的，因此没有方法获取它的 `mut` 引用。一个 `static` 变量可以被声明为 `mut`，但访问它是 `unsafe` 的。Rust 对线程安全性的坚持是所有这些规则的一个主要原因。

全局可变状态在软件工程的角度也会导致不幸的后果：它让程序的不同部分耦合的更紧密、更难测试、更难修改。然而，有些情况确实没有更好的替代方案，所以我们最好找到一种安全的方法来声明可变静态变量。

最简单的支持递增 `PACKETS_SERVED` 并同时保证线程安全的方法，是将它变为原子整数：

```
use std::sync::atomic::AtomicUsize;  
  
static PACKETS_SERVED: AtomicUsize = AtomicUsize::new(0);
```

当这个静态变量声明之后，递增包数就非常直观了：

```
use std::sync::atomic::Ordering;  
  
PACKETS_SERVED.fetch_add(1, Ordering::SeqCst);
```

原子全局变量只限于单个整数和布尔值。创建任何其他类型的全局变量需要解决两个问题。

首先，变量必须是线程安全的，否则它不可能是全局的：出于安全性，静态变量必须是 `Sync` 和非 `mut` 的。幸运的是，我们已经见到过这个问题的解决方案。Rust 有安全地共享可能会修改的值的类型：`Mutex`、`RwLock` 和原子类型。这些类型即使被声明为非 `mut` 也可以被修改。这正是它们的功能。（见 [mut 和 Mutex](#)）

第二，静态变量的初始化只能调用被标记为 `const` 的函数，编译器可以在编译期计算出这些函数的值。也就是说，它们的输出是确定性的，只依赖于它们的参数，不依赖任

何其他的状态或 I/O。这样，编译器可以把计算出的结果作为编译期的常量嵌入。这类似于 C++ 的 `constexpr`。

`Atomic` 类型 (`AtomicUsize`, `AtomicBool` 等) 的构造器都是 `const` 函数，因此允许我们像之前一样创建一个 `static AtomicUsize`。一些其他类型，例如 `String`、`Ipv4Addr`、`Ipv6Addr`，也有简单的 `const` 的构造器。

你也可以通过给函数的签名加上 `const` 前缀来定义你自己的 `const` 函数。Rust 限制 `const` 函数只能做很少的操作，这些操作在不破坏确定性结果的同时足够有用。`const` 函数不能接受类型作为泛型参数，除了生命周期，也不能分配内存或操作原始指针，即使在 `unsafe` 块中也不行。然而，我们可以使用算术运算符（包括回环和饱和算术）、不短路的逻辑运算、和其他 `const` 函数。例如，我们可以创建便捷的函数来简化 `static` 和 `const` 的定义并减少代码重复：

```
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);
```

结合这些技术之后，我们可能会尝试写：

```
static HOSTNAME: Mutex<String> =
    Mutex::new(String::new()); // error: calls in statics are limited to constant
                            // functions, tuple structs, and tuple variants
```

不幸的是，虽然 `AtomicUsize::new()` 和 `String::new()` 是 `const fn`，但 `Mutex::new()` 不是。为了绕开这些限制，我们需要使用 `lazy_static` crate。

我们在 [惰性构建正则值](#) 中介绍过 `lazy_static` crate。使用 `lazy_static!` 宏定义静态变量时，你可以使用任何表达式进行初始化；表达式会在变量第一次解引用时运行，值会被存储在变量中以便后续使用。

我们可以使用 `lazy_static` 声明一个全局的 `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());  
}
```

同样的技术还可以用于其他的复杂数据结构例如HashMap和Deque。当静态变量不需要可变，但需要非平凡的初始化的时候，它非常有用。

使用`lazy_static!`会导致每次访问静态数据有微小的性能开销。它的实现里使用了`std::sync::Once`，它是一种用于一次性初始化的底层同步原语。在幕后，每一次访问一个惰性静态变量时，程序都会执行一个原子load指令来检查是否已经初始化过。`(Once的用途很特殊，因此我们不会在这里详细介绍它。使用lazy_static!通常更加方便。它还可以用于方便地初始化非Rust库，一个示例见一个libgit2的安全接口。)`

19.4 Rust 中的 hacking 并发代码是什么样的

我们已经展示了Rust中使用线程的三种技术：fork-join并行，通道和有锁的共享可变状态。我们的目的是提供一个Rust提供的工具的引导，主要聚焦于它们如何组合用于实际的程序中。

Rust坚持安全性，因此当你决定编写一个多线程程序的时候开始，重点就是构建安全、结构化的通信方式。让线程保持最大限度的隔离是一种说服Rust你正在做安全的事的好方法。恰好这种隔离性也能保证你正在做的事是正确和可维护的。再重复一次，Rust引导你实现好的程序。

更重要的是，Rust允许你结合技术和实验。你可以快速迭代：和编译器作斗争比调试数据竞争更能让你更快地启动和正确运行。

Chapter 20

异步编程

假设你正在编写一个聊天服务器。每个网络连接都有很多要解析的到来的包、要组装的发送的包、要管理的安全参数、要追踪的聊天组订阅等。同时为许多连接管理所有这些信息需要进行一些组织。

理想情况下，你可以简单地为每一个到来的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));
    });
}
```

这样对每一个连接都会创建一个新的线程运行 `serve` 函数，这个函数专门处理一个连接的需求。

这可以正常工作，直到一切都比计划的更加顺利很多，然后突然你就已经有了几万名用户。一个线程的栈增长到 100KB 或更多并不罕见，你可能不想就这样花费几 GB 的内存。要把任务分发到多个处理器上，线程是合适并且必须的，但它们的内存需求太大以至于我们通常需要一些补充的方式和线程一起使用，来减小资源占用。

你可以使用 Rust 的异步任务 (*asynchronous task*) 来在单个线程或者线程池中交替执行很多独立的任务。异步任务类似于线程，但可以更快地创建、更高效地传递控制权、并且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，

你的应用仍然可能被其他因素限制，例如网络带宽、数据库速度、计算、或者任务本身内存需求，但使用异步任务的固有内存开销比使用线程的要小很多。

一般来讲，异步 Rust 代码看起来和普通的多线程代码非常相似，除了那些可能阻塞的操作，例如 I/O 或获取锁的处理有一点不同。特殊对待这些操作让 Rust 有更多的信息了解你的代码的行为，这为进一步优化提供了可能。上面代码的异步版本看起来像这样：

```
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);
    });
}
```

这里使用了 `async_std` crate 的网络和任务模块，并在可能阻塞的调用后加上了 `.await`。但整体的结构和基于线程的版本一样。

这一章的目标不止是帮你编写异步代码，还要向你展示它的工作细节，以便你可以预测它在你的应用中的表现，并了解它在哪些方面最有价值。

- 为了展示异步编程的机制，我们列出了涵盖所有核心概念的最小语言功能集：future、异步函数、`await` 表达式、任务、`block_on` 和 `spawn_local` executor。
- 然后我们会展示异步块和 `spawn` executor。它们是真正完成工作的最基础的部分，但从概念上讲，它们只是我们刚才提到过的功能的变体。在这个过程中，我们会指出一些你可能会遇到的异步编程特有的问题并解释如何处理它们。
- 为了展示所有功能的协调工作，我们会展示一个聊天服务器和客户端的完整代码，上面的代码片段就是其中一部分。
- 为了演示原语 `future` 和 `executor` 如何工作，我们会展示 `spawn_blocking` 和 `block_on` 的简单实现。
- 最后，我们介绍了异步接口中经常出现的 `Pin` 类型，它被用来确保异步函数和块 `future` 被安全地使用。

20.1 从同步到异步

考虑当你调用下面的（不是异步的）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)
}
```

这会打开一个到 web 服务器的 TCP 连接，以过时的协议向它发送一个简单的 HTTP 请求¹，然后读取响应。图 20-1 展示了这个函数的执行过程随时间的变化。

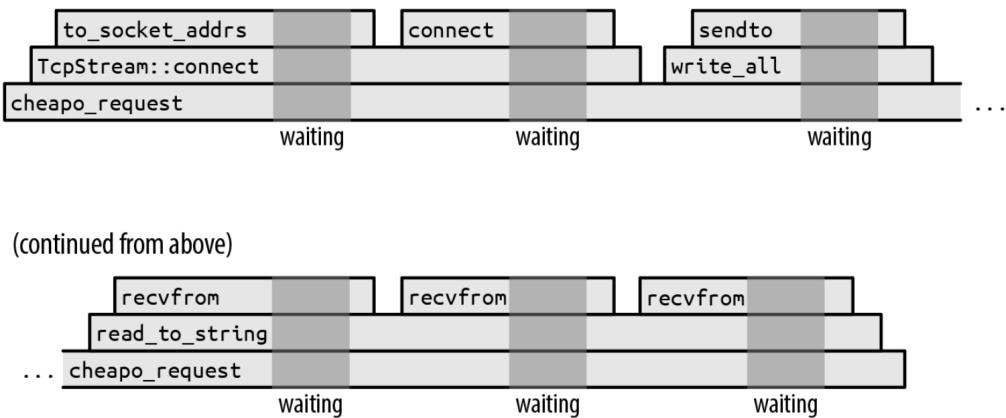


图 20-1: 一个同步 HTTP 请求的过程（深颜色的区域表示等待操作系统）

图中展示了从左到右随着时间的推移，函数的调用栈的变化。每一个函数调用都是一个方块，位于它的调用者上面。显然，`cheapo_request` 函数贯穿整个执行过程。它调用了 Rust 标

¹如果你真的需要一个HTTP客户端，考虑使用一些非常优秀的crate例如surf或reqwest，它们会正确并且异步地完成任务。这个客户端基本只是设法获得HTTPS重定向。

准库里的函数例如 `TcpStream::connect` 和 `TcpStream` 的 `write_all` 和 `read_to_string` 实现。这些对其他函数的调用依次进行，但最终程序会进行系统调用，请求操作系统完成真正的工作，例如打开 TCP 连接，或者读写一些数据。

深灰色的区域表示程序正在等待操作系统完成系统调用。我们并没有按比例绘制这些时间。因为假如我们按比例绘制，整个图都将是深灰色：在实践中，这个函数把几乎所有时间都用在等待操作系统上。上面代码的执行时间将是系统调用之间的窄条。

当函数等待系统调用返回时，它所在的线程会阻塞住：它不能做任何事，直到系统调用结束。一个线程的栈达到几百或几千字节并不罕见，因此如果这是一个更大的系统的一部分，并且有很多线程做类似的任务，锁住这些线程的资源但除了等待什么也不做的代价是非常昂贵的。

为了解决这个问题，一个线程需要能在等待系统调用完成的同时去执行其他的任务。但如何实现这一点并不明显。例如，我们用来从套接字读取响应的函数的签名是：

```
fn read_to_string(&mut self, buf: &mut String) -> std::io::Result<usize>;
```

它的类型表明了：这个函数直到工作完成或者出错时才会返回。这个函数是同步的：当操作完成时调用者才会恢复执行。如果我们想在操作系统进行工作的同时用我们的线程去做别的任务，那么我们需要一个新的提供这个函数的异步版本的 I/O 库。

20.1.1 Future

Rust 支持异步操作的方法是引入一个 trait `std::future::Future`：

```
trait Future {
    type Output;
    // 现在可以先把`Pin<&mut Self>`看作`&mut Self`。
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}

enum Poll<T> {
    Ready(T),
    Pending,
}
```

一个 `Future` 代表一个可以测试是否完成的操作。一个 `future` 的 `poll` 方法从来不会等待操作完成：它总是立即返回。如果操作完成了，`poll` 会返回 `Poll::Ready(output)`，其中 `output` 是最后的结果。否则，它会返回 `Pending`。当且仅当 `future` 值得再次 `poll` 时，它会通过调用一个 `waker` 来通知我们，这是一个由 `Context` 提供的回调函数。我们称之为异步编程的“piñata 模型”：你唯一能对 `future` 做的就是使用 `poll` 敲打它，直到有一个值掉出来。

所有现代的操作系统都包含一些系统调用的变体，我们可以用它们来实现这种 poll 接口。例如在 Unix 和 Windows 上，如果你把网络套接字设置为非阻塞模式，那么如果 read 和 write 会阻塞，它就会返回一个错误，你必须稍后再试。

因此 `read_to_string` 的一个异步版本的签名大概是这样：

```
fn read_to_string(&mut self, buf: &mut String)
-> impl Future<Output = Result<usize>;
```

除了返回类型之外，这和我们之前展示的签名一样：异步的版本返回一个 `Result<usize>` 的 `future`。你需要 poll 这个 `future`，直到从它得到一个 `Ready(result)`。每次它被 poll 时，都会尽可能地继续读取。最后的 `result` 给你成功的值或者错误的值，就像普通的 I/O 操作一样。这是通常的模式：异步版本的任何函数和同步版本的函数获取相同的参数，但返回类型有一个 `Future` 包装。

调用这个版本的 `read_to_string` 并不会真的读取任何内容；它所有的任务就是构造并返回一个 `future`，这个 `future` 会在被 poll 时进行真正的工作。这个 `future` 必须包含处理请求所需的所有信息。例如，这个 `read_to_string` 返回的 `future` 必须记住调用它的输入流，和它需要写入数据的 `String`。事实上，因为这个 `future` 持有了 `self` 和 `buf` 的引用，因此这个 `read_to_string` 的真正的签名必须是：

```
fn read_to_string<'a>(&'a mut self, buf: &'a mut String)
-> impl Future<Output = Result<usize>> + 'a;
```

这个附加的生命周期指示了返回的 `future` 和它借用的 `self` 和 `buf` 的生命周期一样长。

`async-std` crate 提供了 `std` 的所有 I/O 设施的异步版本，包括一个有 `read_to_string` 方法的异步 `Read` trait。`async-std` 密切地遵循了 `std` 的设计，尽可能地在自己的接口中重用 `std` 的类型，因此这两个世界中的错误、结果、网络地址、和其他大多数相关的数据都是兼容的。熟悉 `std` 有助于使用 `async-std`，反之亦然。

`Future` trait 的一个规则是，一旦一个 `future` 返回了 `Poll::Ready`，它会假设它决不会再次被 poll。一些 `future` 在自己被 overpoll 时简单地永远返回 `Poll::Pending`；其它的可能 panic 或者挂起。（它们绝不能违反内存或线程安全性，或者导致未定义行为。）`Future` trait 的 `fuse` 适配器将任何 `future` 转换成 overpoll 时永远返回 `Poll::Pending`。但通常消费 `future` 的方法都遵循这个规则，因此 `fuse` 通常不是必须的。

如果 poll 听起来效率低下，请不必担心。Rust 的异步架构是精心设计的，所以只要你的基本 I/O 函数例如 `read_to_string` 是正确实现的，那么只会在值得 poll 时才会 poll 一个 `future`。每一次 poll 被调用时，某个东西应该返回 `Ready`，或者至少向目标前进一步。我们将在 [原语 future 和 executor：何时一个 future 值得再次 poll](#) 中解释这是如何工作的。

但使用 future 看起来有一个挑战：当你 poll 时，如果你得到了 Poll::Pending 那你应该怎么做？你将不得不四处寻找这个线程暂时可以做的其他工作，并记住一段时间之后返回到这个 future，然后再次 poll。你的整个系统将因为持续追踪谁正在 pending 和当它们完成时应该做什么而变得杂乱无章。我们的 cheapo_request 函数的简洁性会被破坏。

好消息是：它并不是这样的！

20.1.2 `async` 函数和 `await` 表达式

这里有一个异步函数版本的 cheapo_request：

```
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)
}
```

这和之前的版本基本相同，除了：

- 函数以 `async fn` 代替 `fn` 开头。
- 它使用了 `async_std` crate 里的异步版本的 `TcpStream::connect`, `write_all`, `read_to_string`。它们都返回结果的 future (本节中的示例使用了 `async_std` 的 1.7 版本)。
- 每一次调用返回 future 的函数之后，代码都会加上 `.await`。尽管这看起来像是访问一个结构体的 `await` 字段，但它实际上是语言内置的一个特殊语法，它会等待一个 future 直到它准备好。`await` 表达式会求出 future 的最终值。这个函数正是通过它获取 `connect`, `write_all`, `read_to_string` 的结果。

和普通的函数不同，当你调用异步函数时，它会在执行实际的主体代码之前立即返回。显然，调用的返回值还没有被计算出来；你得到的是它的最终值的 *future*。因此如果你执行这

行代码：

```
let response = cheapo_request(host, port, path);
```

那么 `response` 将是一个 `std::io::Result<String>` 的 future，`cheapo_request` 的函数体还没有开始执行。你不需要调整异步函数的返回类型；Rust 会自动把 `async fn f(...) -> T` 看做一个返回 `T` 的 future 而不是直接返回 `T` 的函数。

一个异步函数返回的 future 包含了函数体运行时所需的所有信息：函数的参数、局部变量所需的空间，等等。（就好像你把调用栈捕获为了一个普通的 Rust 值。）因此 `response` 必须包含传入的 `host`, `port`, `path`, 因为 `cheapo_request` 的函数体需要它们才能运行。

`future` 的具体类型由编译器根据函数体和参数自动生成。这个类型并没有名称；你只知道它实现了 `Future<Output=R>`，其中 `R` 是异步函数的返回类型。从这一点来看，异步函数的 future 类似于闭包：闭包也有匿名类型、也是由编译器生成并且实现了 `FnOnce`、`Fn` 和 `FnMut` trait。

当你第一次 poll `cheapo_request` 返回的 future 时，将会从函数体的开头开始运行到第一个由 `TcpStream::connect` 返回的 future 的 `await`。这个 `await` 表达式会 poll `connect` future，如果它还没准备好，那么它会向调用者返回 `Poll::Pending`；直到 `TcpStream::connect` 的 future 返回 `Poll::Ready` 时，对 `cheapo_request` 的 future 的 poll 才能通过第一个 `await` 继续执行。因此表达式 `TcpStream::connect(...).await` 的一个大概等价的写法是：

```
{
    // 注意：这是伪代码，不是有效的Rust代码
    let connect_future = TcpStream::connect(...);

    'retry_point:
    match connect_future.poll(cx) {
        Poll::Ready(value) => value,
        Poll::Pending => {
            // 设置`cheapo_request`的future的下一次`poll`
            // 从'retry_point'处恢复执行。
            ...
            return Poll::Pending
        }
    }
}
```

`await` 表达式会获取 future 的所有权然后 poll 它。如果它已经准备好，那么 future 的最终值就是 `await` 表达式的值，并且会继续往下执行。否则，它向调用者返回 `Poll::Pending`。

但关键的是，下一次 poll `cheapo_request` 的 future 时将不会再次从函数的首部开始：相反，它从 poll `connect_future` 的地方开始恢复 (*resume*) 执行。直到 future 准备好之后我们才

会继续执行这个异步函数的其他部分。

随着 `cheapo_request` 的 future 继续被 poll，它会从函数体里的一个 `await` 开始执行到下一个 `await`，并且只有当它正在等待的子 future ready 时才会继续。因此，`cheapo_request` 的 future 将会被 poll 多少次取决于子 future 的行为和函数本身的控制流。`cheapo_request` 的 future 会追踪下一次 poll 时的恢复点和所有的局部状态——变量、参数、临时值——恢复需要这些。

在函数中间挂起并稍后恢复执行的能力是异步函数独有的。当普通函数返回时，它的栈帧就消失了。因为 `await` 表达式依赖于恢复执行的能力，所以你只能在异步函数里使用它们。

在撰写本书时，Rust 还不允许 trait 有异步方法。只有自由函数和特定类型固有的方法才可以是异步的。取消这个限制需要对语言进行一些修改。在此期间，如果你需要定义包含异步函数的 trait，可以考虑使用 `async-trait` crate，它提供了一个基于宏的解决方案。

20.1.3 在同步代码中调用异步函数：`block_on`

某种意义上讲，异步函数只是在推卸责任。在异步函数里很容易获取一个 future 的值：只需要 `await` 它。但异步函数本身也返回一个 future，因此现在调用者需要负责 poll 它。最后，总有一个地方必须实际等待一个值。

我们可以使用 `async_std` 的 `task::block_on` 函数在普通的同步函数（例如 `main`）中调用 `cheapo_request`，它获取一个 future 并且 poll 它直到它产生一个值：

```
fn main() -> std::io::Result<()> {
    use async_std::task;

    let response = task::block_on(cheapo_request("example.com", 80, "/"))?;
    println!("{}", response);
    Ok(())
}
```

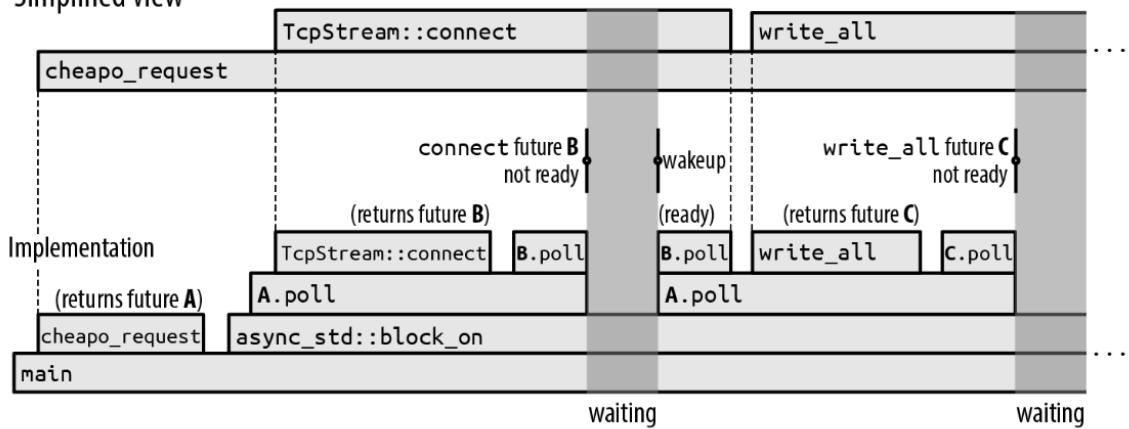
因为 `block_on` 是一个产生异步函数的最终值的同步函数，你可以将它看作是异步世界到同步世界的适配器。但它阻塞的特性也意味着你永远不应该在一个异步函数里使用 `block_on`：它会阻塞整个线程直到值准备好。作为代替，请使用 `await`。

[图 20-2](#)展示了 `main` 的一个可能的执行过程。

上面的时间线，即“简化视图”，展示了程序的异步调用的抽象视图：`cheapo_request` 首先调用了 `TcpStream::connect` 来获取一个套接字，然后对套接字调用了 `write_all` 和 `read_to_string`。然后它返回。这和本章前面的同步版本的 `cheapo_request` 的时间线非常相似。

但这里每一个异步调用都是多阶段的过程：一个 future 被创建，然后被 poll 直到它准备好，可能还会创建并 poll 其他子 future。下面的时间线，即“实现”，展示了实现了这个异步行为

Simplified view



(Continued from above)

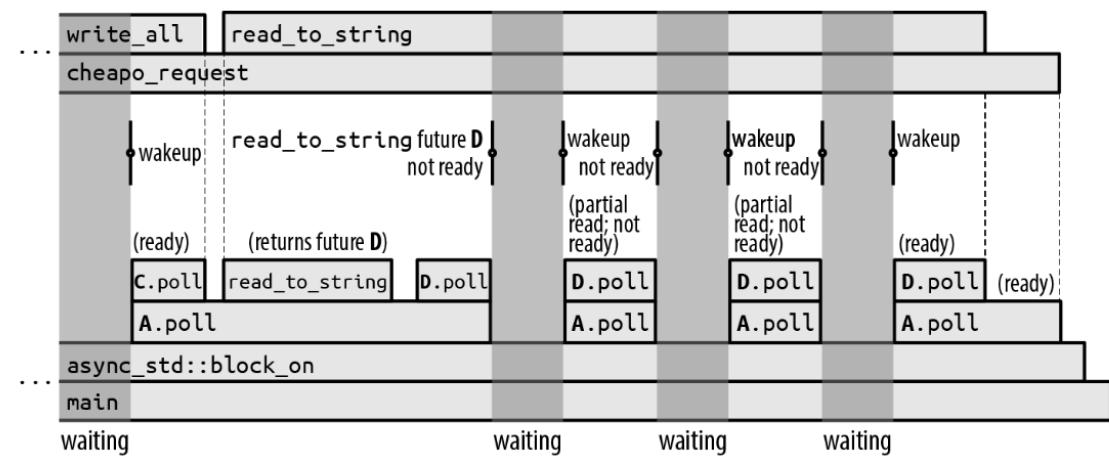


图 20-2: 阻塞等待一个异步函数

的实际同步调用。这是一个介绍普通的异步执行过程中到底发生了什么的好机会：

- 首先，`main` 调用 `cheapo_request`，它返回最终结果的 `future A`。然后 `main` 把这个 `future` 传给了 `async_std::block_on`，它会 `poll A`。
- `poll A` 促使 `cheapo_request` 的函数体开始执行。函数里调用了 `TcpStream::connect` 来获取一个套接字的 `future B` 并 `await` 它。更确切地说，因为 `TcpStream::connect` 可能会遇到错误，因此 `B` 是一个 `Result<TcpStream, std::io::Error>` 的 `future`。
- `future B` 被 `await poll`。因为网络连接还没有建立好，所以 `B.poll` 返回 `Poll::Pending`，但会设置好当套接字准备好后唤醒调用它的任务。
- 因为 `future B` 还没有准备好，`A.poll` 也会向它的调用者 `block_on` 返回 `Poll::Pending`。
- 因为 `block_on` 没有别的事情可做，它会陷入睡眠。这时整个线程会阻塞。
- 当 `B` 的连接准备好之后，它会唤醒 `poll` 它的任务。这促使 `block_on` 开始行动，它会尝试再次 `poll future A`。
- `poll A` 促使 `cheapo_request` 在它的第一个 `await` 处恢复执行，然后再次 `poll B`。
- 这一次 `B` 准备好了：套接字已经创建完毕，因此它返回 `Poll::Ready(Ok(socket))`。
- 到此 `TcpStream::connect` 的异步调用就完成了。`TcpStream::connect(...).await` 表达式的值就是 `Ok(socket)`。
- `cheapo_request` 的函数体会继续正常执行，使用 `format!` 宏构造请求字符串并传递给 `socket.write_all`。
- `socket.write_all` 是一个异步函数，它返回一个 `future C`，然后 `cheapo_request` 会 `await C`。

剩余的流程和之前相似。在图 20-2 所示的执行流程中，`socket.read_to_string` 在准备好之前被 `poll` 了四次，每一次都会从套接字读取一些数据，但 `read_to_string` 被指定为一直读取到输入的末尾，这需要好几次的操作。

听起来编写一个一直调用 `poll` 的循环并不难。但让 `async_std::task::block_on` 真正有价值的是：它知道怎么睡眠到恰好 `future` 值得再次 `poll`，而不是浪费处理器的时间和电量来进行几十亿次无用的 `poll` 调用。基本的 I/O 函数例如 `connect` 和 `read_to_string` 返回的 `future` 保留了传递给 `poll` 的 `Context` 参数提供的唤醒器，并在 `block_on` 应该醒来并再次尝试 `poll` 时调用唤醒器来唤醒它。我们将在原语 `future` 和 `executor`: 何时一个 `future` 值得再次 `poll` 中通过实现一个简单版本的 `block_on` 来展示这具体是怎么工作的。

和我们之前展示的原始的同步版本一样，这个异步版本的 `cheapo_request` 方法也把几乎所有的时间花费在等待操作完成上。如果时间轴是按比例绘制的，那么图将几乎完全是深灰色的，只有当程序被唤醒时会有几个计算过程对应的很细的条。

这里讲了很多细节。幸运的是，你通常可以只考虑简化的上层时间线：一些函数调用是同

步的，其他是异步的并需要一个 `await`，但它们都只是函数调用。Rust 的异步支持的成功取决于帮助程序员在实践中只需要考虑简化的视图，不会被实现的来回跳转干扰。

20.1.4 `spawn` 异步任务

`async_std::task::block_on` 函数会阻塞直到一个 `future` 的值准备好。但在单个 `future` 上完全阻塞一个线程并不比同步调用更好：本章的目的是让线程在等待的同时做别的工作。

为了实现这一点，你可以使用 `async_std::task::spawn_local`。这个函数接受一个 `future` 并把它添加到一个池，当 `block_on` 等待的 `future` 还没准备好时 `block_on` 会 poll 这个池。因此如果你把一堆 `future` 传递给 `spawn_local` 并且之后对最终结果的 `future` 调用 `block_on`，`block_on` 会 poll 每一个被 `spawn` 的 `future`（当它们可以进一步执行时），并发运行整个池，直到结果准备好。

在撰写本书时，只有当启用 `async-std` crate 的 `unstable` 特性时 `spawn_local` 才可用。你需要在 `Cargo.toml` 中用这样的一行引入 `async-std`：

```
async-std = { version = "1", features = ["unstable"] }
```

`spawn_local` 函数是标准中用于启动新线程的 `std::thread::spawn` 函数的异步版本的类似物：

- `std::thread::spawn(c)` 接收闭包 `c` 然后启动一个线程运行它，返回一个 `std::thread::JoinHandle`，它的 `join` 方法会等待线程结束并返回 `c` 返回的内容。
- `async_std::task::spawn_local(f)` 接收 `future f` 并把它添加到当前线程调用 `block_on` 时会 poll 的池里。`spawn_local` 会返回它自己的 `async_std::task::JoinHandle` 类型，它本身是一个 `future`，你可以 `await` 它来获取 `f` 的最终值。

例如，假设我们想让一个 HTTP 请求的集合并发执行。这是第一次尝试：

```
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
}

```

这个函数对 `requests` 的每个元素调用 `cheapo_request`，将每一个调用返回的 `future` 传给 `spawn_local`。它把最后的 `JoinHandle` 收集到一个 `vector` 并且 `await` 每一个。以任意顺序 `await` `join handles` 都是没问题的：因为请求已经被 `spawn`，它们的 `future` 将会被按需 `poll`，即这个线程调用了 `block_on` 并且无事可做时。所有的请求会并发运行。一旦它们完成，`many_requests` 会向调用者返回结果。

上面的代码几乎是正确的，但 Rust 的借用检查器担心 `cheapo_request` 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` 也有一个类似的错误。

自然地，如果我们向异步函数传递引用，那么它们返回的 `future` 就必须持有这些引用，因此出于安全性 `future` 不能比它们借用的值生存的更久。任何持有引用的其他值也有相同的限制。

问题在于 `spawn_local` 不能确保你会在 `host` 和 `path` 被 `drop` 之前等待任务结束。事实上，`spawn_local` 只接受生命周期是 `'static` 的 `future`，因为你可以简单地忽略它返回的 `JoinHandle` 并让任务继续运行程序的剩下部分。这并不是异步任务独有的问题：当你尝试用 `std::thread::spawn` 启动一个线程，并且它的闭包捕获了局部变量的引用时也会遇到类似的问题。

一种解决这个问题的方法是创建另一个版本的获取参数所有权的异步函数：

```

async fn cheapo_owning_request(host: String, port: u16, path: String)
    -> std::io::Result<String> {
    cheapo_request(&host, port, &path).await
}

```

这个函数接收 `String` 而不是 `&str` 引用，因此它的 `future` 自身将拥有 `host` 和 `path`，并且生命周期是 `'static`。借用检查器可以看到它立刻 `await` 了 `cheapo_request` 的 `future`，并且因此如果这个 `future` 被 `poll`，它借用的 `host` 和 `path` 变量肯定还在。一切都没有问题。

使用 `cheapo_owning_request`，你可以像这样 `spawn` 所有的请求：

```
for (host, port, path) in requests {
    handles.push(task::spawn_local(cheapo_owning_request(host, port, path)));
}
```

你可以使用 `block_on` 在同步的 `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),
    }
}
```

这段代码会在 `block_on` 的调用中并发运行三个请求。每个请求会在当其他的请求阻塞时抓住机会继续执行，它们全部在调用者线程中执行。[图 20-3](#) 中展示了三个 `cheapo_request` 调用的可能的执行过程。

(我们鼓励你自己尝试运行这段代码，使用 `eprintln!` 在 `cheapo_request` 开头和每一个 `await` 表达式之后打印消息，这样你可以看到这些调用如何交错执行。)

对 `many_requests` 的调用（为了简单没有展示）`spawn` 了三个异步的任务，分别用 A、B、C 标记。`block_on` 开始时先 `poll` A，A 会开始连接到 `example.com`。这会立刻返回 `Poll::Pending`，`block_on` 会把注意移动到下一个 `spawn` 的任务，然后 `poll` future B，最后是 C，它们会开始连接各自的服务器。

当所有可以 `poll` 的 `future` 都返回了 `Poll::Pending` 之后，`block_on` 会进入睡眠，直到其中一个 `TcpStream::connect` 的 `future` 指示它的任务值得再次 `poll`。

在这次执行中，服务器 `en.wikipedia.org` 比其他的响应得更快，因此这个任务最先完成。当一个 `spawn` 的任务完成时，它会把值保存在 `JoinHandle` 并标记它已经准备好了，这样 `many_requests` `await` 它时无需等待，可以继续执行。最后，其它的 `cheapo_request` 要么

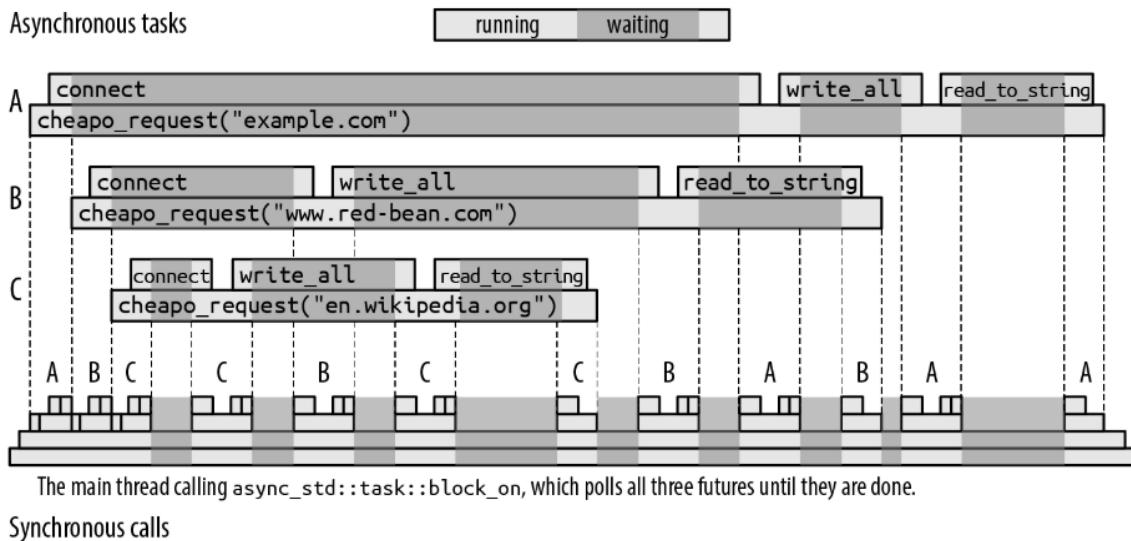


图 20-3: 在单个线程中运行三个异步任务

成功要么返回错误，然后 `many_request` 本身可以返回了。最后，`main` 接收 `block_on` 返回的结果的 `vector`。

所有这些执行都发生在单个线程中，三个 `cheapo_request` 的调用通过对 `future` 的 `poll` 实现交错执行。一个异步调用看起来像是一个运行到完成的单个函数调用，但实际上异步调用由一系列对 `future` 的 `poll` 方法的同步调用实现。每一个单独的 `poll` 调用都可以快速返回，让出线程从而让其他异步调用可以执行。

我们终于达成了我们在本章开头设置的目标：让一个线程在等待 I/O 完成的同时去执行其他的工作，这样线程的资源不会因等待而浪费。更妙的是，达成这个目标的代码看起来非常像普通的 Rust 代码：一些函数被标记为 `async`、一些函数调用后面有 `.await`、使用的函数来自 `async_std` 而不是 `std`，但除此之外，它就是普通的 Rust 代码。

异步任务和线程有一个不同之处需要牢记：异步任务只有在 `await` 表达式中被 `await` 的 `future` 返回 `Poll::Pending` 时才会切换到其他异步任务。这意味着如果你在 `cheapo_request` 中放了一段长时间运行的计算代码，那么在它完成之前，任何传给 `spawn_local` 的其他任务都没有机会运行。而使用线程时没有这个问题：操作系统可以在任何地方挂起任何线程并设置计时器来确保没有线程可以垄断处理器。

异步代码依赖于共享线程的 `future` 的协作。如果你需要让长时间计算和异步代码共存，本章后面的 [长时间计算：yield_now 和 spawn_blocking](#) 中介绍了一些方法。

20.1.5 `async` 块

除了异步函数之外，Rust 还支持异步块 (*asynchronous block*)。与一个普通的块返回最后一个表达式的值不同，一个异步块返回最后一个表达式的值的 *future*。你可以在异步块里使用 `await` 表达式。

异步块看起来就像普通的块表达式，在前边加上 `async` 关键字：

```
let serve_one = async {
    use async_std::net;

    // 监听连接并接受
    let listener = net::TcpListener::bind("localhost:8087").await?;
    let (mut socket, _add) = listener.accept().await?;

    // 通过`socket`与客户端交互
    ...
};
```

这里用一个 *future* 初始化了 `serve_one`，当 `poll` 它时，它会监听并处理单个 TCP 连接。块的代码直到 `serve_one` 被 `poll` 才会执行，就像异步函数只有在它的 *future* 被 `poll` 时才会执行一样。

如果你在异步块里使用了 `?` 操作符，它会从块里返回，而不是从所处的函数返回。例如，如果上面的 `bind` 调用返回一个错误，那么 `?` 操作符会返回它作为 `serve_one` 的最终值。类似的，`return` 表达式会从异步块里返回，而不是从外层的函数返回。

如果一个异步块引用了周围代码里的变量，它的 *future* 会捕获那些变量，就像闭包一样。并且和 `move` 闭包一样（见 [偷取值的闭包](#)），你可以用 `async move` 来创建获取变量所有权的块，而不是持有变量的引用。

异步块提供了一种精确地分离出想要异步运行的部分代码的方法。例如，在上一节中，`spawn_local` 需要 `'static future`，因此我们定义了 `cheapo_owning_request` 包装函数来得到一个获取参数所有权的 *future*。你可以简单地在一个异步块里调用 `cheapo_request` 而不需要分离出包装函数来实现相同的效果：

```
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
        }));
    }
    ...
}

```

因为这是一个 `async move` 块，所以它的 `future` 获取了 `String` 值 `host` 和 `path` 的所有权，就类似 `move` 闭包一样。然后它向 `cheapo_request` 传递引用，借用检查器可以看到块的 `await` 表达式获取了 `cheapo_request` 的 `future` 的所有权，因此 `host` 和 `path` 的引用不可能比它们借用的被捕获的变量生存的更久。异步块和 `cheapo_owning_request` 完成了同样的事，但所需的样板代码更少。

一个你可能遇到的问题是没有语法能指定异步块的返回类型，即异步函数参数后跟的`-> T`。当使用`? 操作符`时这可能会导致问题：

```

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(())
};

```

这会因为如下错误失败：

```

error: type annotations needed
|
42 |     let future = async {
|         ----- consider giving `future` a type
...
46 |         input.read_line(&mut line).await?;
|         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ cannot infer type

```

Rust 不能分辨异步块的返回类型应该是什么。`read_line` 方法返回一个 `Result<(), std::io::Error>`，但因为`? 操作符`使用了 `From` trait 来在需要时转换成指定的错误类型，所以这个异步块的返回类型是 `Result<(), E>`，其中 `E` 可能是任何实现了 `From<std::io::Error>` 的类型。

Rust 未来的版本可能会添加指定 `async` 块的返回类型的语法。但现在，可以通过手动写出最后的 `Ok` 的类型来解决这个问题：

```
let future = async {
    ...
    Ok::<(), std::io::Error>(())
};
```

因为 `Result` 是一个需要成功和错误类型作为参数的泛型类型，我们可以像这里一样使用 `Ok` 或 `Err` 指定那些类型参数。

20.1.6 从异步块中构建异步函数

异步块给了我们另一种实现和异步函数相同效果的方法，并且更加灵活一点。例如，我们可以将我们的 `cheapo_request` 写成一个普通的、同步的返回异步块的 `future` 的函数：

```
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 {
        ... function body ...
    }
}
```

当你调用这个版本的函数时，它会立刻返回异步块的值的 `future`。这个 `future` 会捕获函数的参数并且和异步函数返回的 `future` 的行为一样。因为我们没有使用 `async fn` 语法，我们需要在返回值中写出 `impl Future`，但对调用者来说，这两个定义是同一个函数签名的两种可替换的实现。

如果你想让函数被调用时立刻进行一些计算然后再构造返回的 `future`，那么第二种方法更有用。例如，另一种协调 `cheapo_request` 和 `spawn_local` 的方法是让它变成一个返回 `'static` `future` 的同步函数，并让这个 `future` 捕获参数的拷贝的所有权：

```
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 {
    ... use &host, port, and path ...
}

}

```

这个版本让异步块捕获 `host` 和 `path` 为 `String` 值，而不是 `&str` 引用。因为 `future` 拥有自己运行所需的所有数据，所以它是有效的`'static`生命周期。（我们在上面的签名中写出了`+ 'static`，但`-> impl`返回的类型默认是`'static`的，因此省略它不会有影响。）

因为这个版本的 `cheapo_request` 返回的 `future` 是`'static`的，我们可以直接把它们传递给 `spawn_local`:

```

let join_handle = async_std::task::spawn_local(
    cheapo_request("areweasyncyet.rs", 80, "/")
);

... other work ...

let response = join_handle.await?;

```

20.1.7 在一个线程池中 spawn 异步任务

我们至今为止展示过的例子几乎把所有时间花费在等待 I/O 上，但一些负载是更多处理器工作和阻塞的组合。当你有太多的计算以至于单个处理器不能进行快速处理，你可以使用 `async_std::task::spawn` 来把一个 `future` spawn 到一个工作线程池里，这些线程会 poll 可以进一步执行的 `future`。

`async_std::task::spawn` 的使用方法类似 `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
    }));
}
...

```

类似于 `spawn_local`，`spawn` 也返回一个 `JoinHandle` 值，你可以 `await` 它来获取 `future` 的最终值。但和 `spawn_local` 不同的是，这个 `future` 不会等到你调用 `block_on` 才会被 `poll`，只要线程池中有一个空闲的线程，它就会尝试 `poll` 这个 `future`。

在实践中，`spawn` 比 `spawn_local` 使用得更加广泛，因为人们更希望他们的负载不管计算和阻塞怎么混合，都能在机器上均衡地执行。

当使用 `spawn` 时一个需要记住的点是线程池会尝试保持忙碌，因此只要有一个线程空闲你的 `future` 就会被 poll。一个异步调用可能在一个线程中开始执行，在一个 `await` 表达式处阻塞，最后在另一个不同的线程中恢复执行。因此将一个异步函数调用看作单个函数调用是一个合理的简化（事实上，异步函数和 `await` 表达式的目的就是鼓励你以这种方式思考）。和代码的执行情况有关，异步调用可能实际上会在很多不同线程中移动。

如果你正在使用 `thread-local` 存储，你可能会惊讶地发现你在 `await` 表达式之前放置的一些数据在恢复之后被替换成了某些完全不同的东西，这是因为你的任务现在正在被池中的另一个线程 poll。如果这导致了问题，你应该使用 `task-local storage`；细节见 `async-std` crate 中 `task_local!` 宏的文档。

20.1.8 但你的 `future` 实现了 `Send` 吗？

有一个 `spawn` 要求但 `spawn_local` 不要求的限制。因为 `future` 被送到另一个线程运行，因此 `future` 必须实现了 `Send` 标记 trait。我们在线程安全：`Send` 和 `Sync` 中介绍过 `Send`。只有当 `future` 包含的所有值都是 `Send` 时 `future` 才是 `Send`：所有的函数参数、局部变量、甚至匿名的临时值都必须能安全地移动到另一个线程。

和之前一样，这个要求也不是异步任务独有的：如果你尝试使用 `std::thread::spawn` 启动一个捕获了非 `Send` 值的闭包也会遇到一个类似的错误。不同之处在于，传给 `std::thread::spawn` 的闭包会留在新创建的线程中运行，而 `spawn` 到线程池里的 `future` 可能在 `await` 时从一个线程移动到另一个线程。

这个限制很容易意外触发。例如，下面的代码看起来足够合法：

```
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());
```

一个异步函数的 `future` 必须持有足够的信息来让它可以从一个 `await` 表达式继续执行。在

这个例子中，`reluctant`的future必须在`await`之后使用`string`，因此这个future将会，或至少有时会，包含一个`Rc<String>`值。因为`Rc`指针不能安全地在线程之间共享，所以这个future本身不能是`Send`。并且因为`spawn`只接受`Send`的future，所以Rust会报错：

```
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
```

这一段错误信息很长，但包含很多有用的细节：

- 它解释了为什么future需要是`Send`: `task::spawn`的要求。
- 它解释了什么样的值不是`Send`: 局部变量`string`，它的类型是`Rc<String>`。
- 它解释了为什么`string`会影响future: 它的作用域跨过了`await`。

有两种解决这个问题的方法。一个是限制非`Send`的值的作用域，让它不包含任何`await`表达式，因此就不需要保存在函数的future里：

```
async fn reluctant() -> String {
    let return_value = {
        let string = Rc::new("ref-counted string".to_string());
        format!("Your splendid string: {}", string)
        // `Rc<String>`在这里离开作用域...
    };
}
```

```
// ... 因此当我们在这里挂起时不需要保存它。  
some_asynchronous_thing().await;  
  
return_value  
}
```

另一种解决方案是简单地用 `std::sync::Arc` 替换 `Rc`。`Arc` 使用原子更新来管理它的引用计数，这意味着它会稍微慢一点，不过 `Arc` 指针是 `Send`。

尽管最终你会学会识别和避免非 `Send` 类型，但一开始它们可能令人惊讶。（至少，你的作者经常被惊讶到。）例如，较旧的 Rust 代码有时会像这样使用泛型结果类型：

```
// 不推荐!  
type GenericError = Box;  
type GenericResult<T> = Result<T, GenericError>;
```

这个 `GenericError` 类型使用了一个 trait 对象来存储任何实现了 `std::error::Error` 的类型。但并没有给它施加更严格的限制：如果有一个非 `Send` 类型实现了 `Error`，它们将能转换成一个 `GenericError` 类型。因为这种可能性，`GenericError` 将不是 `Send`，下面的代码将不能工作：

```
fn some_fallible_thing() -> GenericResult<i32> {  
    ...  
}  
  
// 这个函数的 future 不是`Send`...  
async fn unfortunate() {  
    // ... 因为这个调用返回的值...  
    match some_fallible_thing() {  
        Err(error) => {  
            report_error(error);  
        }  
        Ok(output) => {  
            // ... 到这个 await 处仍然存在...  
            use_output(output).await;  
        }  
    }  
}  
  
// ... 因此这个`spawn`会导致错误。  
async_std::task::spawn(unfortunate());
```

和前面的例子一样，编译器的错误消息解释了发生了什么，指出了那个 `Result` 是罪

魁祸首。因为 Rust 考虑到 `some_fallible_thing` 的结果在整个 `match` 表达式中生效，包括 `await` 表达式，它决定了 `unfortunate` 的 `future` 不是 `Send`。这个错误是因为 Rust 过度谨慎：尽管 `GenericError` 不能安全地发送到另一个线程，但 `await` 只会在结果是 `Ok` 的时候发生，因此当我们 `await use_output` 返回的 `future` 时错误的值永远不会有存在。

一个理想的解决方法是使用更加严格的泛型错误类型，例如我们在[处理多种错误类型](#)中建议的这个：

```
type GenericError = Box;
type GenericResult<T> = Result<T, GenericError>;
```

这个 trait 对象显式地要求底层的错误类型要实现了 `Send`，这样就一切顺利了。

如果你的 `future` 不是 `Send` 并且不能方便地将它变成 `Send`，那么你可以使用 `spawn_local` 来在当前线程运行它。当然，你需要保证这个线程在某个地方调用 `block_on`，以给它运行的机会，并且你将不能从多处理器中受益。

20.1.9 长时间计算: `yield_now` 和 `spawn_blocking`

如果一个 `future` 和其他任务共享线程，那么它的 `poll` 方法应该总是尽可能快速地返回。但如果你在进行长时间的计算，它可能需要很长时间才会到达下一个 `await`，让其它的异步任务等待比你预想得更长的时间。

一种避免这种情况的方法是偶尔就 `await` 一次。`async_std::task::yield_now` 函数返回一个为此设计的简单 `future`：

```
while computation_not_done() {
    // ... 进行中等规模的计算 ...
    async_std::task::yield_now().await;
}
```

`yield_now` 的 `future` 第一次被 `poll` 时，它会返回 `Poll::Pending`，但它会很快声明它值得再次 `poll`。效果就是你的异步调用可以放弃线程，其他的任务可以得到运行的机会，但很快又会轮到你的调用。`yield_now` 的 `future` 第二次被 `poll` 时，它会返回 `Poll::Ready(())`，因此你的异步函数可以恢复执行。

然而这个方法并不总是可行。如果你正在使用一个外部的 crate 来做长时间计算或者调用外部的 C 或 C++ 代码，那么并不方便修改代码来变得更加异步友好。或者可能很难确保计算的每一条路径都会经过 `await`。

对于这种情况，你可以使用 `async_std::task::spawn_blocking`。这个函数接受一个闭包，在它自己的线程中运行它，并返回一个返回值的 `future`。异步代码可以 `await` 这个 `future`，把

它的线程让给其他的任务，直到计算完成。通过把困难的任务放在单独的线程，可以让操作系统负责让它很好地共享处理器。

例如，假设我们需要检查用户输入的密码和我们在认证数据库中存储的哈希过的版本是否一致。为了安全性，验证密码需要是计算密集的，这样即使攻击者获取了数据库的拷贝，他们也不能简单地尝试几万亿个可能的密码来看看是否匹配。`argonautica` crate 提供了一个专为存储密码设计的哈希函数：一个正确生成的 `argonautica` 哈希值需要几分之一秒来验证。我们可以像这样在我们的异步应用中使用 `argonautica`（版本 0.2）：

```
async fn verify_password(password: &str, hash: &str, key: &str)
    -> Result<bool, argonautica::Error>
{
    // 获取参数的拷贝，以让闭包变为'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` 匹配 `hash` 它会返回 `Ok(true)`，其中的 `key` 是数据库里的一个键。在传给 `spawn_blocking` 的闭包里进行验证，可以把昂贵的计算放到它自己的线程里，确保它不会影响对其他用户的请求返回响应。

20.1.10 比较异步设计

Rust 的异步编程的方案在很多方面都和其他语言采用的方案很像。例如，JavaScript、C# 和 Rust 都有带有 `await` 表达式的异步函数。所有这些语言都有值来表示还未完成的计算：Rust 称之为“future”，JavaScript 称之为“promise”，C# 称之为“task”，但它们都代表一个可能要等待的值。

然而 Rust 中 `poll` 的使用并不寻常。在 JavaScript 和 C# 中，一个异步函数被调用后会立刻执行，有一个内置在系统库中的全局的事件循环负责当它们等待的值可用时恢复挂起的异步函数调用。然而在 Rust 中，异步函数调用什么都不做，直到把它的 `future` 传递给 `block_on`、

spawn 或者 spawn_local，这些函数会 poll 它并驱动工作完成。这些函数，称为 *executor*，扮演了其它语言中的全局事件循环的角色。

因为 Rust 允许你——程序员来选择一个 executor 来 poll 你的 future，所以 Rust 不需要内置在系统中的全局事件循环。`async-std` crate 提供了我们在本章中用过的 executor 函数，但我们在本章稍后会使用的 `tokio` crate，定义了它自己的类似的 executor 函数集。并且作为本章的终结，我们会实现自己的 executor。你可以在同一个程序中使用这三种 executor。

20.1.11 一个真实的异步 HTTP 客户端

如果不展示一个使用合适的异步 HTTP 客户端 crate 的例子将是我们的疏忽，因为它是如此简单，并且有好几个好的 crate 可以选择，包括 `reqwest` 和 `surf`。

这里有一个使用 `surf` 来并发运行一系列请求的重写的 `many_requests`，甚至比基于 `cheapo_request` 的版本还要简单，你需要在 `Cargo.toml` 中加上这些依赖：

```
[dependencies]
async-std = "1.7"
surf = "1.0"
```

然后，我们可以像这样定义 `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` 来进行所有请求让我们可以在其中某些请求指向同一个服务器时重用 HTTP 连接。并且不需要异步块：因为 `recv_string` 是一个返回 `Send + 'static` future 的异步方法，我们可以直接把它的 future 传给 `spawn`。

20.2 一个异步的客户端和服务器

是时候整理一下我们至今为止讨论过的关键思路并将它们组合成一个可以工作的程序了。很大程度上来说，异步应用类似于普通的多线程应用，但有新机会写出紧凑且富有表现力的代码。

这一节的示例是一个聊天服务器和客户端。[完整的代码](#)见这里。真实的聊天系统很复杂，从安全和重连到隐私和现代化都是需要考虑的因素，但我们将只实现一组简单的功能子集，这样能更加关注几个我们感兴趣的点。

特别地，我们想很好的处理背压 (*backpressure*)。意思是如果一个客户端的网络连接很慢或者完全丢失了连接，必须不影响其他客户端交换信息的能力。并且因为一个慢速的客户端不应该让服务器花费无限制的内存来保存它不断增长的累积消息，我们的服务器应该丢弃一些不能跟上速度的客户端的消息，但要通知它们它们的消息流是不完整的。(一个真实的服务器应该把消息记录到磁盘上并让客户端去获取它们错过的消息，不过我们省略了这个功能。)

我们以命令 `cargo new --lib async-chat` 开始项目，首先把以下内容添加到 `async-chat/Cargo.toml`:

```
[package]
name = "async-chat"
version = "0.1.0"
authors = ["You <you@example.com>"]
edition = "2018"
```

```
[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"
```

我们依赖四个 crate:

- `async-std` crate 是我们在本章中一直在用的异步 I/O 原语和工具的集合。
- `tokio` crate 是另一个类似 `async-std` 的异步原语的集合，它是最古老和成熟的之一。它被广泛使用并保持设计和实现的高标准，但相比 `async-std` 还需要一些别的 crate 才能使用。
`tokio` 是一个很大的 crate，但我们只需要它的一个组件，因此 `Cargo.toml` 中的 `features = ["sync"]` 字段将 `tokio` 削减到只有我们需要的部分，让它更加轻量一些。
当异步库的生态系统不够成熟时，人们会避免同时在一个程序中使用 `tokio` 和 `async-std`，但这两个项目一直在合作来确保可以正确工作，只要遵守它们的文档中的每一条规则。
- `serde` 和 `serde_json` crate 我们之前已经在第 18 章中见过。它们给了我们便利且高效地生成和解析 JSON 的工具，我们的聊天协议将使用 JSON 来在网络中表示数据。我们想使用 `serde` 中的一些可选特性，因此在我们指定依赖时选择了那些特性。

我们的聊天应用的整体架构，包括客户端和服务器，看起来像这样：

```
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我们在 `src/bin` 目录中介绍过的一个 Cargo 的特性：除了主要的库 crate `src/lib.rs` 和它的子模块 `src/utils.rs` 之外，它还包含两个可执行文件：

- `src/bin/client.rs` 是聊天客户端的单文件可执行程序。
- `src/bin/server` 是聊天服务器的可执行程序，它被分成四个文件：`main.rs` 保存 `main` 函数，还有三个子模块 `connection.rs`、`group.rs`、`group_table.rs`。

我们将在本章中展示每个源文件的内容，等它们都就位之后，如果在目录树中输入 cargo build，就会编译库 crate 并且构建两个可执行程序。Cargo 会自动把库 crate 当作一个依赖，这使得它变为一个放置客户端和服务器共享的定义的好地方。类似的，cargo check 会检查整个源码树。为了运行其中某一个可执行程序，你可以使用像这样的命令：

```
$ cargo run --release --bin server -- localhost:8088  
$ cargo run --release --bin client -- localhost:8088
```

--bin 选项指示了要运行哪一个可执行程序，并且任何跟在--选项后面的参数都会被传给可执行程序本身。我们的客户端和服务器需要知道服务器的地址和 TCP 端口。

20.2.1 Error 和 Result 类型

库 crate 的 utils 模块定义了整个应用中用到的结果和错误类型。src/utils.rs:

```
use std::error::Error;  
  
pub type = ChatError = Box<dyn Error + Send + Sync + 'static>;  
pub type = ChatResult<T> = Result<T, ChatError>;
```

这是我们在[处理多种错误类型](#)中建议过的通用的错误类型。async_std、serde_json 和 tokio crate 都定义了它们自己的错误类型，但? 运算符可以自动把它们全部转换成 ChatError，使用标准库的 From trait 的实现可以把任何合适的错误类型转换成 Box<dyn Error + Send + Sync + 'static>。Send 和 Sync 约束确保了如果一个被 spawn 到其他线程的任务失败了，它可以安全地把错误汇报给主线程。

在一个真实的应用中，请考虑使用 anyhow crate，它提供了类似于这里的 Error 和 Result 类型。anyhow crate 易于使用并且提供了一些我们的 ChatError 和 ChatResult 没有的很棒的特性。

20.2.2 协议

库 crate 把整个聊天协议封装在两个类型里，在 lib.rs 中定义：

```
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` 枚举表示一个客户端可能发送给服务器的包：它可以要求加入一个房间并向它加入的房间发送消息。`FromServer` 表示服务器可能返回给客户端的包：被发到聊天组里的消息和错误消息。使用引用计数指针 `Arc<String>` 来代替普通的 `String` 帮助服务器在管理组和分发消息时避免拷贝字符串。

`#[derive]` 属性让 `serde` crate 为 `FromClient` 和 `FromServer` 生成 `Serialize` 和 `Deserialize` trait 的实现。这让我们可以调用 `serde_json::to_string` 来把它们转换成 JSON 值、通过网络发送它们、并且最终调用 `serde_json::from_str` 来把它们转换回 Rust 形式。

`test_fromclient_json` 单元测试展示了这该如何使用。有了 `serde` 生成的 `Serialize` 实现，

我们可以调用 `serde_json::to_string` 来把给定的 `FromClient` 值转换成这个 JSON：

```
{"Post": {"group_name": "Dogs", "message": "Samoyeds rock!"}}
```

然后生成的 `Deserialize` 实现会把它转换成一个等价的 `FromClient` 值。注意 `FromClient` 中的 `Arc` 指针对序列化形式没有影响：引用计数的字符串直接作为 JSON 的对象成员值出现。

20.2.3 获取用户输入：异步流

我们的聊天客户端的第一个功能是读取用户的命令并向服务器发送相应的包。管理一个合适的用户接口超出了本章的范围，因此我们只准备完成能工作的最简单的实现：直接从标准输入读取。下面的代码在 `src/bin/client.rs`：

```
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?;
        // `parse_command` 的定义见 Github 仓库
        let request = match parse_command(&command) {
            Some(request) => request,
            None => continue,
        };
        utils::send_as_json(&mut to_server, &request).await?;
        to_server.flush().await?;
    }
}
```

这段代码中调用了 `async_std::io::stdin` 来获取一个客户端的标准输入的异步 handle，用 `async_std::io::BufReader` 包装它来进行缓冲，然后调用 `lines` 来逐行处理用户的输入。它尝试把输入的每一行命令行解析为 `FromClient` 值，并且如果成功就把值发送给服

务器。如果用户输入了未知的命令，`parse_command`会打印出错误消息并返回 `None`，因此 `send_commands` 可以继续循环。如果用户输入了 end-of-file 标志，那么 `lines` 流会返回 `None`，因此 `send_commands` 会返回。这和以普通的同步程序的方式编写的代码非常相似，除了它使用了 `async_std` 版本的库特性。

异步的 `BufReader` 的 `lines` 方法很有趣。它并不像标准库一样返回一个迭代器：标准库中 `Iterator::next` 方法是一个普通的同步函数，因此调用 `commands.next()` 将会阻塞线程直到读取到下一行。作为代替，它返回一个 `Result<String>` 值的流 (*stream*)。流是异步中和迭代器类似的概念：它按需以一种异步友好的风格产生一个值的序列。这里是 `Stream` trait 的定义，来自于 `async_std::stream` 模块：

```
trait Stream {
    type Item;

    // 现在，把`Pin<&mut Self>`看作`&mut Self`就好。
    fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<Option<Self::Item>>;
}
```

你可以将它看作 `Iterator` 和 `Future` trait 的结合。类似于迭代器，`Stream` 有关联的 `Item` 类型并使用 `Option` 来指示序列何时结束。但类似于 `future`，流必须被 `poll` 才能得到下一个 `item`（或者知道 `stream` 已经结束），你必须调用 `poll_next` 直到它返回 `Poll::Ready`。一个流的 `poll_next` 实现应该总是快速返回，不能阻塞。如果一个流返回 `Poll::Pending`，它必须在值得再次 `poll` 时通过 `Context` 提醒调用者。

`poll_next` 方法直接使用起来很别扭，但你通常不需要这么做。类似迭代器，流也有很多工具方法例如 `filter` 和 `map`。其中一个是 `next` 方法，它返回流的下一个 `Option<Self::Item>` 的 `future`。你可以调用 `next` 并 `await future` 返回而不是显式地 `poll` 流。

将这些组合起来，`send_commands` 通过使用 `next` 和 `while let` 迭代一个流产生的值并消耗输入的行：

```
while let Some(item) = stream.next().await {
    ... use item ...
}
```

（未来的 Rust 版本可能会引入一种 `for` 循环语法的异步变体来消耗流，就像普通的 `for` 循环消耗 `Iterator` 值一样。）

在流结束后 `poll` 它——即在它返回 `Poll::Ready(None)` 来指示流结束之后——就类似于在一个迭代器返回 `None` 之后调用 `next` 或在一个 `future` 返回 `Poll::Ready` 之后 `poll` 它一样。

`Stream` trait 没有指定这种情况下流的行为，因此有些流可能行为不当。类似于 `future` 和迭代器，流有一个 `fuse` 方法来确保这样的调用结果是可预测的，更多细节见文档。

当处理流时，要记得 use `async_std::prelude::*`:

```
use async_std::prelude::*;


```

这是因为 `Stream` trait 的工具方法，例如 `next`, `map`, `filter` 等等，并不是真的定义在 `Stream` 自身里。实际上，它们是另一个 trait `StreamExt` 的默认方法，这个 trait 自动为所有 `Stream` 实现：

```
pub trait StreamExt: Stream {
    // ... 以默认方法的方式定义工具方法 ...
}

impl<T: Stream> StreamExt for T { }
```

这是我们在 trait 和其他人的类型 中介绍过的扩展 trait (extension trait) 的一个例子。`async_std::prelude` 模块把 `StreamExt` 的方法引入作用域，因此要记得 use 这个 prelude 来确保这些方法在你的代码中可见。

20.2.4 发送包

为了通过网络套接字传输包，我们的客户端和服务器使用了库 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(())
}
```

这个函数构建 `packet` 的 JSON `String` 表示，在末尾加上了一个换行符，然后全部写入 `outbound`。

通过它的 where 子句，你可以看到 `send_as_json` 非常灵活。要发送的包的类型 `P`，可以是任何实现了 `serde::Serialize` 的类型。输出流 `S` 可以是任何实现了 `async_std::io::Write` 的类型，这个 trait 是 `std::io::Write` trait 的异步版本。这足够我们在一个异步的 `TcpStream` 上发送 `FromClient` 和 `FromServer` 值。保持 `send_as_json` 的定义是泛型的可以确保它不依赖流或包的类型的细节，不过这样的话 `send_as_json` 就只能使用那些 trait 的方法了。

为了使用 `write_all` 方法，`S` 中的 `Unpin` 约束是必须的。我们将在本章稍后介绍 `pin` 和 `unpin`，但现在只需要给需要的类型参数添加上 `Unpin` 约束即可，如果你哪里忘了，Rust 编译器会指出来的。

`send_as_json` 把包序列化到临时的 `String` 中，然后写入到 `outbound` 中，而不是直接序列化到 `outbound` 流中。`serde_json` crate 确实提供了一些函数把值直接序列化到输出流，但那些函数只支持同步流。写入到异步流需要同时修改 `serde_json` 和 `serde` crate 的格式无关的核心，因为这些 trait 是为同步方法设计的。

和流一样，`async_std` 的 I/O trait 的很多方法实际上也是定义在扩展 trait 中，因此无论何时使用它们都要记得 `use async_std::prelude::*;`

20.2.5 接收包：更多异步流

为了接收包，我们的服务器和客户端将使用 `utils` 模块中的这个函数来接收 `FromClient` 和 `FromServer` 值，它从一个异步的 TCP 套接字，即 `async_std::io::BufReader<TcpStream>` 中接收值：

```
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)
        })
}
```

类似 `send_as_json`，这个函数的输入流和包的类型也是泛型的：

- 流的类型 `S` 必须实现了 `async_std::io::BufRead`，它是 `std::io::BufRead` 的异步版本，表示一个有缓冲的输入字节流。

- 包的类型 P 必须实现了 DeserializeOwned，它是 serde 的 Deserialize trait 的一个更严格的变体。为了性能，Deserialize 可以产生直接从被反序列化的缓冲区借用的 &str 和 &[u8] 值，来避免拷贝数据。然而在我们的例子中，这并不是一个好主意：我们需要向调用者返回反序列化后的值，因此它们的生命周期必须要能超过解析的缓冲区。一个实现了 DeserializeOwned 的类型总是和被反序列化的缓冲区无关。

调用 inbound.lines() 返回一个 std::io::Result<String> 值的 Stream。然后我们使用了流的 map 适配器来对每一个 item 应用一个闭包，处理错误并把每一行当作类型 P 的一个值的 JSON 形式进行解析。这会产生一个 ChatResult<P> 值的流，然后直接返回它。函数的返回类型是：

```
impl Stream<Item = ChatResult<P>>
```

这意味着我们返回的类型异步地产生一个 ChatResult<P> 值的序列，但我们的调用者并不能确定它的精确类型。因为我们传递给 map 的闭包有一个匿名类型，它是 receive_as_json 可能返回的最具体的类型。

注意 receive_as_json 本身并不是一个异步函数。它是一个返回一个异步值，即一个流的普通函数。更加深入地了解 Rust 的异步支持，而不是“简单地到处添加 async 和 .await”为类似这种充分利用语言优势的清晰、灵活和高效的定义开辟了可能性。

为了看看 receiver_as_json 如何使用，这里是我们的 src/bin/client.rs 中的聊天客户端的 handle_replies 函数，它接收一个来自网络的 FromServer 值的流并打印给用户看：

```
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(())
}
```

```
}
```

这个函数接受一个从服务接收数据的套接字，用一个BufReader 包装它（当然，也是async_std的版本），然后传递给receive_as_json来获取一个到来的FromServer值的流。然后它使用了一个while let 循环来处理到来的响应，检查错误并打印出每一条服务器的响应。

20.2.6 客户端的main函数

因为我们已经展示了send_commands 和handle_replies，我们可以展示聊天客户端的主函数了，它的定义在src/bin/client.rs中：

```
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(())
    })
}
```

在从命令行获取了服务器的地址之后，main有一系列要调用的异步函数，因此它把其余的函数包装在一个异步块中并把块的future 传递给async_std::task::block_on来运行。

一旦连接建立之后，我们希望send_commands 和handle_replies 函数串联执行，这样我们可以在输入的时候看到其他人的消息。如果我们输入了文件结束符或者到服务器的连接断开，程序应该退出。

按照我们这一章介绍的内容，你可能会想类似这样的代码：

```
let to_server = task::spawn(send_commands(socket.clone()));
let from_server = task::spawn(handle_replies(socket));
```

```
to_server.await?;
from_server.await?;
```

但因为我们要 await 这两个 join handle，只有当两个任务都结束时程序才会退出。我们希望任何一个任务结束时程序就会退出。future 的 race 方法完成了这个功能。调用 from_server.race(to_server) 返回一个新 future，它同时 poll from_server 和 to_server，并在任何一个准备好时返回 Poll::Ready(v)：最终的值是第一个完成的 future 的值，未完成的 future 被丢弃。

race 方法和很多其它有用的工具，被定义在 `async_std::prelude::FutureExt` trait 中，`async_std::prelude` 让它对我们可见。

到这里，客户端的代码中只剩下 `parse_command` 函数还没有展示过。它只是非常直观的文本处理代码，因此我们不会在这里展示它的定义。细节见 Git 仓库中的完整代码。

20.2.7 服务器的 main 函数

这里是服务器的主文件：`src/bin/server/main.rs` 的完整内容：

```
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 {
        // 这段代码在本章的导论部分展示过
        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 {
            if let Err(e) = socket_result {
                eprintln!("Error accepting connection: {}", e);
            } else {
                let socket = socket_result.unwrap();
                let mut connection = connection::Connection::new(socket);
                task::spawn(connection.run());
            }
        }
    });
}
```

```

        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` 函数和客户端的很像：它进行一些初始化工作，然后调用 `block_on` 来运行一个异步块来做真正的工作。为了处理从客户端到来的连接，它创建了一个 `TcpListener` 套接字，它的 `incoming` 方法返回一个 `std::io::Result<TcpStream>` 值的流。

对每个到来的连接，我们 `spawn` 一个异步任务运行 `connection::serve` 函数。每一个任务还接受一个 `GroupTable` 值的引用，它表示我们的服务器的聊天组的列表，通过一个 `Arc` 引用计数指针被所有连接共享。

如果 `connection::serve` 返回一个错误，我们向标准错误输出记录一条消息并让任务退出。其它的连接继续运行。

20.2.8 处理聊天连接：异步的 Mutex

这里是服务器的主体：`src/bin/server/connection.rs` 中的 `connection` 模块的 `serve` 函数：

```

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 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);
    }
}
}

```

```
{  
    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(())  
}
```

这基本只是客户端的 `handle_replies` 函数的镜像：代码的主体是一个循环，它负责处理一个 `FromClient` 值的流，这个流是使用 `receive_as_json` 从一个缓冲的 TCP 流构建的。如果有错误发生，我们就生成一个 `FromServer::Error` 包将坏消息传回给客户端。

除了错误消息之外，客户端还接收它们加入的聊天组的消息，因此与客户端的连接要被每个组共享。我们可以简单地给每个组一个 `TcpStream` 的克隆，但如果这些源里有两个尝

试同时向套接字里写入包，它们的输出就会穿插在一起，并且客户端最终将会接收到乱码的 JSON。我们需要安排连接的安全并发访问。

我们使用在 `src/bin/server/connection.rs` 里定义的 `Outbound` 类型来完成这一点，它的定义如下：

```
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(())
    }
}
```

在创建 `Outbound` 值时，它会获取一个 `TcpStream` 值得所有权，并把它包装在一个 `Mutex` 里来确保同时只有一个任务可以使用它。`serve` 函数把每个 `Outbound` 包装在一个 `Arc` 引用计数指针里，这样客户端加入的所有组可以共享同一个 `Outbound` 实例。

调用 `Outbound::send` 时首先锁住互斥锁，返回一个可以解引用为内部的 `TcpStream` 值的 `guard` 值。我们使用 `send_as_json` 来传输 `packet`，最后调用 `guard.flush()` 来确保它不会只传输了一半。（据我们所知，`TcpStream` 并不真的缓冲数据，但 `Write` trait 允许它的实现这么做，因此我们不应该冒险。）

表达式 `&mut *guard` 可以帮我们解决 Rust 并不会为了满足 trait 约束来强制解引用的问题。我们显式地解引用互斥锁 `guard` 并且借用 `TcpStream` 的可变引用，产生一个 `send_as_json` 需要的 `&mut TcpStream`。

注意 `Outbound` 使用了 `async_std::sync::Mutex` 类型，而不是标准库的 `Mutex`。这么做有三个原因。

第一，如果一个任务在持有一个互斥锁 `guard` 的时候挂起，标准库的 `Mutex` 可能会有不恰当的行为。如果正在运行任务的线程选择了另一个任务来运行，并且那个任务里尝试锁住同一个 `Mutex`，就会出现问题：从 `Mutex` 的观点来看，已经拥有它的线程再次尝试锁住它。标准的 `Mutex` 设计时并没有考虑到这种情况，因此它会 `panic` 或者死锁。（它永远不会错

误地允许锁定。) 有一些工作让 Rust 能在编译期检测出这个问题并且在 `std::sync::Mutex` guard 的作用域包含一个 `await` 表达式时给出一个警告。因为 `Outbound::send` 需要在 `await send_as_json` 和 `guard.flush` 的同时持有锁，所以它必须使用 `async_std` 的 `Mutex`。

第二，异步的 `Mutex` 的 `lock` 方法返回一个 `guard` 的 `future`，因此一个等待锁住互斥锁的任务会让出线程给其它任务，直到互斥锁准备好。(如果互斥锁已经可用，`lock` future 会立刻准备好，任务也不会挂起自身。) 另一方面，标准的 `Mutex` 的 `lock` 方法在等待获取锁时会定住整个线程。因为上面的代码在通过网络传输一个包时需要持有锁，这可能会需要一段时间。

最后，标准的 `Mutex` 必须被锁住它的线程解锁。为了强迫这一点，标准的互斥锁 `guard` 类型没有实现 `Send`: 它不能被传送到其他线程。这意味着一个包含这样的 `guard` 的 `future` 自身也没有实现 `Send`，因此不能传给 `spawn` 以在线程池中运行；它只能用 `block_on` 或者 `spawn_local` 来运行。一个 `async_std` `Mutex` 的 `guard` 确实实现了 `Send`，因此在被 `spawn` 的任务中使用它不会有问题是。

20.2.9 聊天组表：同步的 `Mutex`

但正确的原则并不是像“总是在异步代码中使用 `async_std::sync::Mutex`”这么简单。通常没有必要在持有锁的同时 `await` 别的东西，上锁通常也不会持续太长时间。在这种情况下，标准库的 `Mutex` 可能更加高效。我们的聊天服务器的 `GroupTable` 类型展示了这种情况。这里是 `src/bin/server/group_table.rs` 的完整内容：

```
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()
}

}

```

一个 GroupTable 就是一个被锁保护的哈希表，它把聊天组的名称映射到实际的组，两者都使用引用计数指针来管理。get 和 get_or_create 方法会锁住互斥锁，进行一些哈希表的操作，可能还会有一些内存分配，并返回。

在 GroupTable 中，我们使用了一个普通的 std::sync::Mutex。这个模块中没有任何异步代码，因此没有需要避免的 await。事实上，如果我们想在这里使用 async_std::sync::Mutex，我们需要把 get 和 get_or_create 变成异步函数，这会引入 future 的创建、挂起、恢复的开销，收益却只有一点：互斥锁只有在一些哈希操作和内存分配操作时会被锁住。

如果我们的聊天服务器发现它有了数百万的用户，并且 GroupTable 互斥锁变成了一个瓶颈，那么把它变为异步的并不能解决问题。使用一些专为并发访问设计的集合类型来代替 HashMap 可能会更好。例如，dashmap crate 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。这个类型只需要支持两个 connection::serve 会调用的方法：join 添加一个成员，post 发送一条消息。每一条消息需要发送给所有的成员。

这里我们要解决前面提到的背压挑战。有几个相关的需求：

- 如果一个成员不能跟上被发送到组里的消息——例如它们的网络连接很慢——其它的成员不应该被影响。
- 即使有成员落后，也应该有办法让他们重新加入对话并以某种方式继续参与。
- 用来缓存消息的内存不应该无限制地增长。

因为这些挑战在实现多到多的通信模式时非常普遍，tokio crate 提供了一个广播通道 (*broadcast channel*) 类型，它实现了一组合理的权衡。一个 tokio 广播通道是一个值的队列（在我们的例子中就是聊天消息），它允许任何数量的不同线程或任务发送和接收值。它被称为“广播”通道，因为每一个消费者都会获得每个值的拷贝。（值的类型必须实现了 Clone。）

通常，一个广播通道会保留队列中的每条消息直到每一个消费者都获得了它的拷贝。但如果队列的长度超过了通道的最大容量（在创建时指定），最旧的消息就会被丢弃。所有不能跟

上的消费者会在尝试获取下一条消息时将得到一个错误值，并且通道会把它们前进到可用的最旧的消息。

例如，图 20-4 展示了一个最大容量为 16 的广播通道。

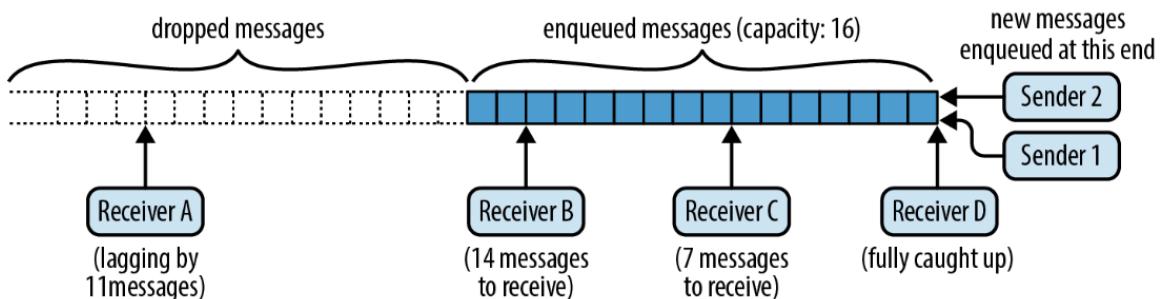


图 20-4: 一个 tokio 广播通道

这里有 2 个 sender 发送消息和 4 个 receiver 接收消息——或者更精确地说，从队列中拷贝消息。Receiver B 有 14 条要接收的消息，Receiver C 有 7 条，Receiver D 完全跟上了。Receiver A 落后了，并且有 11 条消息都被丢弃了。它下一次尝试接收消息时会失败，返回一个错误表示这种情况，然后它会直接前进到当前队列的尾部。

我们的聊天服务器将每一个聊天组表示为一个携带 `Arc<String>` 值的广播通道：向组发送一条消息会向所有的成员广播。这里是 `group::Group` 类型的定义，在 `src/bin/server/group.rs` 中定义：

```
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>) {
    // 当没有订阅者的时候这只会返回一个错误。一个连接的输出端可能在输入端结束发送
    // 之前的短时间内退出并drop掉它的订阅，这可能会导致输入端向一个空的组里发送消息。
    let _ignored = self.sender.send(message);
}
}

```

一个 Group 结构体持有聊天组的名称和一个 broadcast::Sender，它表示组的广播通道的发送端。Group::new 方法调用了 broadcast::channel 来创建一个最大容量为 1000 条消息的广播通道。channel 函数同时返回一个 sender 和一个 receiver，但我们这时还不需要 receiver，因为这个组这时还没有任何成员。

为了向组里添加成员，Group::join 方法会调用 sender 的 subscribe 方法来创建一个新的通道的 receiver。然后它 spawn 一个新的异步任务来监测 receiver 的消息，并把它们写回到客户端，这个功能在 handle_subscriber 函数中完成。

这些细节完成之后，Group::post 方法就很直观了：它简单地向广播通道发送消息。因为通道携带的值是 Arc<String>，给每个 receiver 一个拷贝只会增加消息的引用计数，没有任何拷贝或堆分配。一旦所有的订阅者都接收到了消息，引用计数就会降到 0，消息就会被释放。

这里是 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;
}
}
```

尽管细节不同，但这个函数的形式非常熟悉：它是一个从广播通道接收消息并通过共享的 Outbound 发送回客户端的循环。如果循环不能跟上广播通道，它会接收到一个 Lagged 错误，然后它会报告给客户端。

如果向客户端发送消息失败了，可能是因为连接关闭了，`handle_subscriber`会退出它的循环并返回，导致异步任务退出。这会drop广播通道的Receiver，取消通道的订阅。这样，当连接被drop时，它的每个组成员都会被清除。

我们的聊天组永远不会关闭，因为我们永远不会从组表里移除一个组，但为了完整性，`handle_subscriber`还是通过退出任务来处理一个 `Closed` 错误。

注意我们为每一个组的每一个客户端成员都创建了一个新的异步任务。这是可行的，因为异步任务与线程相比消耗的内存要小很多，并且在一个进程中从一个异步任务切换到另一个非常高效。

这就是聊天服务器的完整代码。它有些简陋，并且 `async_std`、`tokio` 和 `futures` crate 还有很多比我们在本书中介绍的更有价值的特性，但这个示例的理想目的是展示异步生态系统的某些功能如何协同工作：异步任务、流、异步 I/O trait、通道、和两种风格的互斥锁。

20.3 原语 future 和 executor：何时一个 future 值得再次 poll

聊天服务展示了我们怎么使用像 `TcpListener` 和 `broadcast` 通道这样的异步原语编写代码，并用像 `block_on` 和 `spawn` 来驱动它们执行。现在我们可以看看它们是如何实现的。关键的问题在于，当一个 `future` 返回 `Poll::Pending` 时，它怎么跟 `executor` 协调以在正确的时间再次 poll 它？

考虑当我们运行聊天客户端的 main 函数中的这段代码时会发生什么：

```
task::block_on(async {
    let socket = net::TcpStream::connect(address).await?;
    ...
})
```

`block_on`第一次poll异步块的future时，网络连接当然不能立刻准备好，因此`block_on`会陷入睡眠。但应该什么时候唤醒它？一旦网络连接准备好，`TcpStream`需要告诉`block_on`它应该再次poll异步块的future，因为它知道这个时候，`await`将会结束，异步块的执行可以取得进展。

当一个类似`block_on`这样的executor poll一个future时，它必须传递一个回调函数，称为一个`waker`。如果future还没有准备好，`Future` trait的规则要求它必须立刻返回`Poll::Pending`，并设置好当future值得再次poll时调用waker。

因此一个手写的`Future`实现通常看起来像这样：

```
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waker 稍后使用。
        self.waker = Some(cx.waker().clone());
        Poll::Pending
    }
}
```

换句话说，如果future的值已经准备好，就返回它。否则，保存一个`Context`的waker，并返回`Poll::Pending`。

当future值得再次poll时，future必须通过waker通知最后一个poll它的executor：

```
// 如果有一个waker，就调用它，并清除'self.waker'。
if let Some(waker) = self.waker.take() {
    waker.wake();
}
```

理想情况下，executor 和 future 交替 poll 和 wake：executor poll future 然后陷入睡眠，然后 future 调用 waker，唤醒 executor 并再次 poll future。

异步函数和异步块的 future 本身并不处理 waker。它们简单地把上下文传递给它们 await 的子 future，委托它们保存并调用 waker。在我们的聊天客户端中，异步块的 future 第一次被 poll 时简单地把它的上下文传递给它 await 的 `TcpStream::connect` 的 future。后续的 poll 也是类似地把上下文传递给 await 的 future。

`TcpStream::connect` 的 future 按照上一个例子中展示的那样被 poll：它把 waker 交给一个辅助线程，这个线程等待连接完成然后调用 waker。

`Waker` 实现了 `Clone` 和 `Send`，因此当需要时一个 future 总是可以获得 waker 的拷贝并发送给别的线程。`Waker::wake` 方法会消耗 waker。还有一个 `wake_by_ref` 方法不会消耗 waker，但一些 executor 可以把消耗版本实现得更高效一些。（区别最多就是一个 `clone`。）

一个 executor 过度 poll 一个 future 是无害的，只是不够高效。然而，future 应该谨慎调用 waker，直到它们可以取得实际进展的时候再调用：没有意义的唤醒和 poll 循环会导致一个 executor 根本不能睡眠，不仅浪费电量还让处理器无法处理其它任务。

现在我们已经展示了 executor 和 future 是如何通信的，我们将自己实现一个 future，然后介绍一个 `block_on` executor 的实现。

20.3.1 调用 waker: `spawn_blocking`

本章更早的时候，我们描述过 `spawn_blocking` 函数，它启动一个给定的闭包，在另一个线程运行并返回一个返回值的 future。我们现在已经有了实现 `spawn_blocking` 所需的所有知识。为了简单，我们的版本将会每一个闭包创建一个新的线程，而不是像 `async_std` 的版本一样使用一个线程池。

尽管 `spawn_blocking` 返回一个 future，但我们并不准备把它写成一个 `async fn`。相反，它将是一个普通的同步函数，返回一个 `SpawnBlocking` 结构体，我们将基于它实现我们自己的 `Future`。

我们的 `spawn_blocking` 的签名如下：

```
pub fn spawn_blocking<T, F>(closure: F) -> SpawnBlocking<T>
where F: FnOnce() -> T,
      F: Send + 'static,
      T: Send + 'static,
```

因为我们需要把闭包送到另一个线程去运行并把结果带回来，因此闭包 `F` 和它的返回值 `T` 都必须实现了 `Send`。并且因为我们不知道这个线程将会运行多久，因此它们必须都实现了 `'static`。这些约束和 `std::thread::spawn` 的约束一样。

`SpawnBlocking<T>` 是一个闭包将要返回的值的 future。这是它的定义：

```
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交点，因此把它放进 `Arc` 里，并且使用了 `Mutex` 来保护它。(这里使用同步的 `Mutex` 是没问题的。) `poll` 这个 future 会检查 `value` 是否就绪和是否要把 `waker` 保存到 `waker` 中。运行闭包的线程会把返回值保存到 `value` 中，并且如果有 `waker` 的话就调用它。

这里是 `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)
}

```

在创建了 Shared 值之后，这段代码 spawn 了一个线程来运行闭包，把结果存储在 Shared 的 value 字段中，然后调用 waker。

我们可以像下面这样为 SpawnBlocking 实现 Future：

```

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
    }
}

```

poll 一个 SpawnBlocking 会检查闭包的值是否已经准备好了，如果准备好了就获取它的所有权并返回。否则，future 就继续 pending，并在 future 的 waker 字段中保存一个上下文的 waker。

一旦一个 Future 返回 Poll::Ready，你不应该再去 poll 它。通常的消耗 future 的方式，例如 await 和 block_on 都遵守这个规则。如果一个 SpawnBlocking 被过度 poll，不会有特别的麻烦发生，但它也没有特别处理这种情况。对手写的 future 来说这是典型的情况。

20.3.2 实现 block_on

除了能实现 future 原语之外，我们也已经有了构建一个简单的 executor 所需的所有知识。在这一节中，我们将编写我们自己的 block_on。它将比 async_std 的版本简单一些；例如，它不支持 spawn_local、task-local 变量、或者嵌套调用（在异步代码中调用 block_on）。但它

足够运行我们的聊天客户端和服务器。代码如下：

```
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(),
        }
    }
}
```

这段代码很短，但却有很多内容，让我们逐步来分解。

```
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()` 会阻塞当前的线程直到某个别的线程对相应的 `Unparker` 调用 `.unpark()`，`Unparker` 通过调用 `parker.unparker()` 获得。如果你 `unpark` 了一个还没有 `park` 的线程，它的下一次 `park` 调用将没有阻塞、立即返回。我们的 `block_on` 将使用 `Parker` 来在 `future` 还没有准备好时等待，并且传递给 `future` 的 `waker` 将负责 `unpark` 它：

```
let waker = waker_fn(move || unparker.unpark());
```

`waker_fn` crate 中的同名函数将会根据一个给定的闭包创建一个 `Waker`。这里，我们创建了一个 `Waker`，它被调用时会调用闭包 `move || unparker.unpark()`。你可以只使用标准库创建 `waker`，但 `waker_fn` 更加方便一点。

```
pin!(future);
```

给定一个类型 F 的 future, `pin!` 宏会获取 future 的所有权并声明一个同名的新变量, 它的类型是 `Pin<&mut F>`, 并且借用那个 future。这就给了我们 `poll` 方法所需的 `Pin<&mut Self>`。原因我们将在下一节介绍, 但异步函数和块的 future 在可以被 `poll` 之前必须通过 `Pin` 来引用。

```
loop {
    match future.as_mut().poll(&mut context) {
        Poll::Ready(value) => return value,
        Poll::Pending => parker.park(),
    }
}
```

最后, 这个 `poll` 循环非常简单。传递一个携带我们的 `waker` 的上下文之后, 我们 `poll` future 直到它返回 `Poll::Ready`。如果它返回 `Poll::Pending`, 我们就 `park` 线程, 它会阻塞直到 `waker` 被调用。然后我们会再次尝试 `poll`。

`as_mut` 调用让我们可以在不放弃所有权的情况下 `poll` future; 我们将在下一节中更详细地解释这一点。

20.4 Pin

尽管异步函数和块对于编写清晰的异步代码至关重要, 但处理它们的 future 需要非常小心。`Pin` 类型帮助 Rust 确保它们被安全使用。

在这一节中, 我们将展示为什么异步函数调用和块的 future 不能像普通的 Rust 值一样自由处理。然后我们将展示 `Pin` 如何充当一个指针的 “seal of approval”, 以被用来安全地管理这样的 future。最后, 我们将展示一些使用 `Pin` 值的方法。

20.4.1 future 的两个生命阶段

考虑下面简单的异步代码:²

```
use async_std::io::prelude::*;
use async_std::{io, net};

async fn fetch_string(address: &str) -> io::Result<String> {
    // ①
    let mut socket = net::TcpStream::connect(address).await?; // ②
    let mut buf = String::new();
    socket.read_to_string(&mut buf).await?; // ③
}
```

²译者注: 因为某些谜之原因, 译者这里无法在代码里逃逸 (`escapeinside` 无效), 只能 (用 `texcomments`) 在注释里逃逸, 这里原文中②和③都是在 `await` 和?之间的位置。

```
    Ok(buf)
}
```

这会打开一个到给定地址的 TCP 连接并且返回一个服务器发送的 `String`。标记为①②③的点是恢复点 (*resumption point*)，异步函数中的代码在执行时可能会在这些点挂起。

假设你调用了它但却没有 `await`，例如：

```
let response = fetch_string("localhost:6502");
```

现在 `response` 是一个准备在 `fetch_string` 的开头执行的 `future`。在内存中，这个 `future` 看起来像图 20-5 这样。

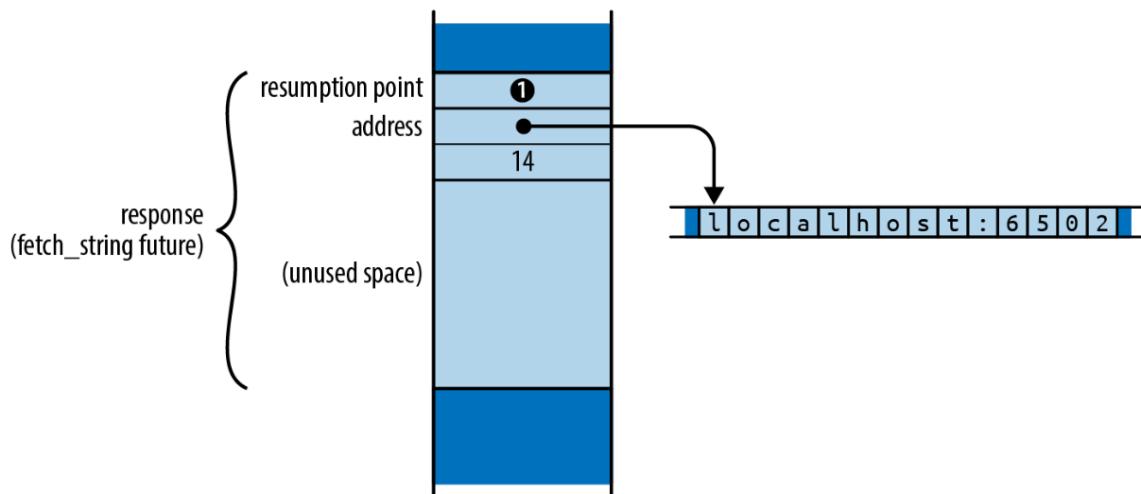


图 20-5: `fetch_string` 的调用构造出的 `future`

因为我们刚刚创建这个 `future`，它应该在恢复点①，即函数体的开头开始执行。这个状态下，`future` 继续运行时唯一需要的值就是函数参数。

现在假设你已经 `poll` 了 `response` 几次，它到达了函数体中的这个点：³

```
socket.read_to_string(&mut buf).await?; // ③
```

进一步假设 `read_to_string` 的结果还没有准备好，因此这一次 `poll` 会返回 `Poll::Pending`。这时，`future` 看起来像图 20-6。

一个 `future` 必须总是持有它下一次被 `poll` 时恢复运行所需的所有信息。在这个情况下这些信息包括：

- 恢复点③，表示应该在 `poll read_to_string` 的 `future` 的 `await` 处恢复。
- 在这个恢复点处还存在的变量：`socket` 和 `buf`。`address` 的值在这个 `future` 中并没有体现，因为这个函数不再需要它。

³同上。

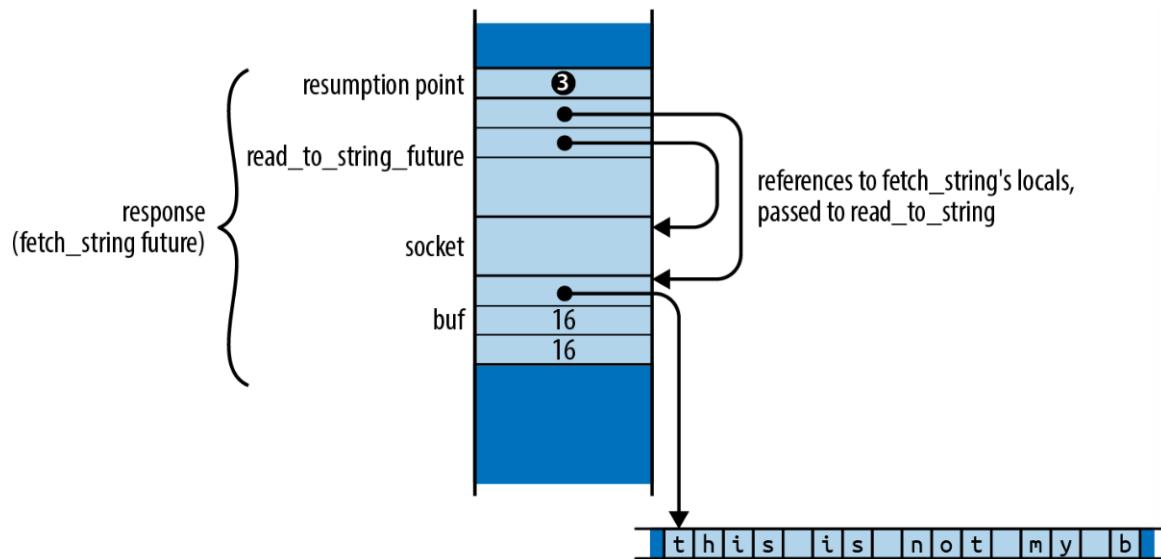


图 20-6: await `read_to_string` 到一半的同一个 future

- `read_to_string` 子 future, `await` 表达式正在 poll 它。

注意对 `read_to_string` 的调用借用了 `socket` 和 `buf` 的引用。在一个同步函数中，所有的局部变量都存在栈上，但在异步函数中，生命周期包含一个 `await` 的局部变量必须放置在 future 中，这样当下一次 poll 时它们才可用。借用这样的变量的引用会借用 future 的一部分。

然而，Rust 要求值在被借用时不能被 move。假设你想把这个 future 移动到一个新位置：

```
let new_variable = response;
```

Rust 没有办法找到所有活跃的引用并调整它们。因此 `socket` 和 `buf` 的引用会继续指向原本的 `response` 里的旧位置，而不是指向新位置。它们就会变成悬垂指针，如图 20-7 所示。

阻止被借用的值被 move 通常是借用检查器的职责。借用检查器把变量看作所有权树的根，但和存储在栈上的变量不同，如果 future 本身被 move 了那么存储在 future 中的变量也会被 move。这意味着 `socket` 和 `buf` 的借用不仅会影响 `fetch_string` 可以对自己的变量进行的操作，还会影响它的调用者可以对 `response`（包含它们的 future）进行的操作。异步函数的 future 是借用检查器的盲点，Rust 为了保持内存安全的保证必须处理这种情况。

Rust 对这个问题的解决方案依赖于一个发现：future 在刚被创建时总是可以安全地 move，只有当它们被 poll 以后才不能安全地 move。一个异步函数调用返回的刚创建的 future 只包含一个恢复点和参数值。这些都属于异步函数的函数体，此时它还没有开始执行。只有当 poll 一个 future 后才有可能借用它的内容。

从这一点来看，我们可以看到每个 future 都有两个生命阶段：

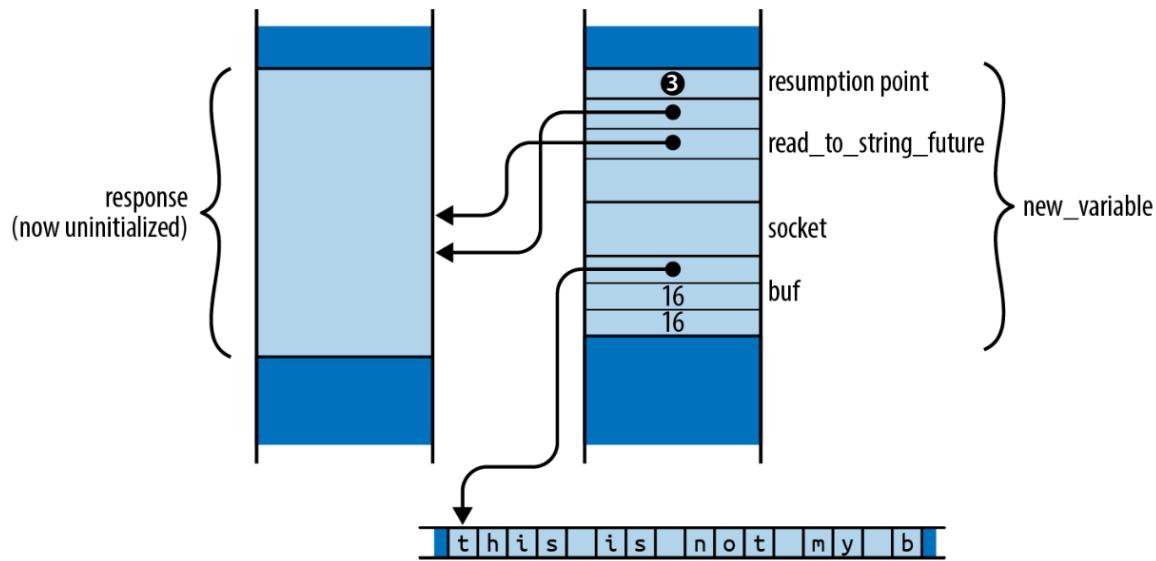


图 20-7: `fetch_string` 的 future，在借用的同时被 move (Rust 会阻止这种情况)

- 当 future 被创建时第一个阶段开始。因为函数体还没有开始执行，它的任何一部分都还没有被借用。这时，它和其他 Rust 值一样可以安全地移动。
- 第二个阶段开始于 future 第一次被 poll。一旦函数体开始执行，它就可能借用存储在 future 中的变量的引用，然后 await，导致 future 的一部分被借用。在 future 第一次被 poll 之后，我们必须假设它已经不能安全地 move 了。

第一个生命阶段的灵活性让我们可以把 future 传递给 `block_on` 和 `spawn`，以及调用适配器方法例如 `race` 和 `fuse`，它们全都是以值接受 future。事实上，创建 future 的异步函数调用也需要把它返回给调用者，这里也有一次 move。

future 只有被 poll 之后才会进入第二个生命阶段。`poll` 方法需要 future 以 `Pin<&mut Self>` 值传递。`Pin` 是一个指针类型的包装（类似于 `&mut Self`），它限制了指针的使用方式，确保它们引用的对象（例如 `Self`）不能再次被 move。因此在 poll 一个 future 之前你必须先创建它的 `Pin` 包装的指针。

这就是 Rust 保证 future 安全的策略：一个 future 在被 poll 之前不可能变得危险；你不能 poll 一个 future，直到你构造一个 `Pin` 包装的指针；并且一旦你这么做了，future 就不能再 move。

“一个不能 move 的值” 听起来好像不可能：在 Rust 中 move 无处不在。我们将在下一节中详细解释 `Pin` 如何保护 future。

尽管这一节是在讨论异步函数，但所有讨论的内容也适用于异步块。一个由异步块创建的新 future 会像闭包一样从周围的环境中捕获它用到的变量。只有当 poll 它的 future 时才会创

建对它的内容的引用，将它标记为不能安全地 move。

请记住，这种 move 的脆弱性仅限于异步函数和块的 future，以及它们特殊的编译器生成的 Future 实现。如果你手动为你的类型实现了 Future，就像我们在 [调用 waker: spawn_blocking](#) 中为我们的 SpawnBlocking 类型做的一样，这样的 future 不管是在 poll 前还是 poll 后都可以完美地 move。在任何手写的 poll 实现中，借用检查器会确保你借用的 self 的部分引用会在 poll 返回时消失。这只是因为异步函数和块有能力在一个函数调用中途挂起执行，如果这个过程中有借用，我们必须谨慎处理它们的 future。

20.4.2 Pinned 指针

Pin 类型是一个 future 的指针的包装，它限制了指针的用法来确保 future 一旦被 poll 之后就不能再 move。对于不会被 move 的 future 这些限制可以被忽略，但它们对安全地 poll 异步函数和块的 future 是至关重要的。

这里的指针，意思是任何实现了 Deref 或者 DerefMut 的类型。Pin 包装的指针被称为 *pinned pointer*。Pin<&mut T> 和 Pin<Box<T>> 是典型的 pinned 指针。

标准库中的 Pin 的定义很简单：

```
pub struct Pin<P> {  
    pointer: P,  
}
```

注意 pointer 字段不是 pub 的。这意味着唯一构造或者使用 Pin 的方法就是该类型提供的精心挑选过的方法。

给定一个异步函数或块的 future，有一些获取 pinned 指针的方法：

- futures-lite crate 提供的 pin! 宏，把一个 T 类型的变量替换成一个 Pin<&mut T> 类型的新变量。新的变量指向原来的值，原来的值被移动到栈上的一个匿名临时变量中，当新变量离开作用域时，值也会被 drop。我们在我们的 block_on 实现中使用了 pin! 来钉住我们想要 poll 的 future。
- 标准库的 Box::pin 构造器获取任意类型 T 的值的所有权，把它移动到堆里，然后返回一个 Pin<Box<T>>。
- Pin<Box<T>> 实现了 From<Box<T>>，因此 Pin::from(boxed) 获取 boxed 的所有权并返回一个 pinned box 指向堆上的同一个 T 值。

每一种获得 future 的 pinned 指针的方式都意味着放弃 future 的所有权，并且没有办法将它转变回来。pinned 指针自身可以 move 到任何地方，但 move 一个指针不会 move 它引用的对象。因此创建一个 pinned 指针的过程就是在证明你永久放弃了 move 这个 future 的能力。这就是想让它可以被安全地 poll 所需的全部内容。

一旦你钉住了一个 future，如果你想 poll 它，所有的 `Pin<pointer to T>` 类型都有一个 `as_mut` 方法解引用指针并返回一个 `Pin<&mut T>`，这正是 `poll` 所需的。

`as_mut` 方法还可以帮助你在不放弃所有权的情况下 poll 一个 future。我们的 `block_on` 实现就用到了这一点：

```
pin!(future);

loop {
    match future.as_mut().poll(&mut context) {
        Poll::Ready(value) => return value,
        Poll::Pending => parker.park(),
    }
}
```

这里，`pin!` 宏重新声明 `future` 为一个 `Pin<&mut F>`，因此我们可以把它传递给 `poll`。但可变引用不是 `Copy`，因此 `Pin<&mut F>` 也不能是 `Copy`，这意味着直接调用 `future.poll()` 将会获取 `future` 的所有权，导致下一次迭代时 `future` 的值变为未初始化状态。为了避免这一点，我们调用了 `future.as_mut()` 来在每一次迭代时重新借用一个新的 `Pin<&mut F>`。

没有办法得到一个被钉住的 `future` 的 `&mut` 引用：因为如果可以的话，你就可以使用 `std::mem::replace` 或者 `std::mem::swap` 把它 move 出来，并在原来的位置放一个新的 `future`，这显然是不允许的。

我们在普通的异步代码中之所以不用考虑钉住的 `future`，是因为大多数情况下获取一个 `future` 的值——`await` 它或者传给一个 `executor`——都会获取 `future` 的所有权并在内部管理 pinning。例如，我们的 `block_on` 实现就获取了 `future` 的所有权，然后使用 `pin!` 宏来产生一个 `poll` 所需的 `Pin<&mut F>`。一个 `await` 表达式也会获取 `future` 的所有权并且在内部使用一种类似 `pin!` 宏的方法。

20.4.3 Unpin trait

然而，不是所有的 `future` 都需要这么谨慎地处理。对于任何为普通类型手写的 `Future` 实现，例如我们之前提到的 `SpawnBlocking` 类型，这种构造和使用 pinned 指针的限制是不必要的。

这样的类型会实现 `Unpin` 标记 trait：

```
trait Unpin { }
```

Rust 中几乎所有的类型都通过编译器的特殊支持实现了 `Unpin`。异步函数和块的 `future` 是这个规则的例外。

对于 `Unpin` 类型，`Pin` 没有施加任何限制。你可以使用 `Pin::new` 从一个普通的指针制造一个 `pinned` 指针，也可以使用 `Pin::into_inner` 转换回指针。`Pin` 本身会传递指针自己的 `Deref` 和 `DerefMut` 实现。

例如，`String` 实现了 `Unpin`，因此我们可以写：

```
let mut string = "Pined?".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.");
```

即使构造出一个 `Pin<&mut String>` 之后，我们也有对原本的字符串的完整可变访问权限，并且一旦 `Pin` 被 `into_inner` 消耗，可变引用就会消失，此时我们还可以把它 move 进一个新的变量中。因此对于那些是 `Unpin` 的类型——几乎所有类型都是——`Pin` 只是一个该类型指针的无聊的包装。

这意味着当你为自己的 `Unpin` 类型实现 `Future` 时，你的 `poll` 实现可以把 `self` 看作是一个 `&mut Self`，而不是一个 `Pin<&mut Self>`。`pin` 基本可以完全忽略。

你可能会很惊讶 `Pin<&mut F>` 和 `Pin<Box<F>>` 实现了 `Unpin`，即使 `F` 没有实现。这听起来好像不对——一个 `Pin` 怎么能是 `Unpin`? ——但如果你仔细想想这两个术语的含义，它确实应该是这样的。即使 `F` 一旦被 `poll` 之后就不能再安全地 `move`，但一个指向它的指针总是可以安全地 `move` 的，不管是否被 `poll` 过。只有指针 `move` 了，它指向的对象并没有。

当你想把异步函数或块的 `future` 传递给一个只接受 `Unpin future` 的函数时知道这些是有用的。(这样的函数在 `async_std` 中很罕见，但在异步生态系统的其他地方较为常见。) 即使 `F` 不是 `Unpin`，`Pin<Box<F>>` 也是 `Unpin`，因此对一个异步函数或块的 `future` 应用 `Box::pin` 可以得到一个可以在任何地方使用的 `future`，代价是一次堆分配。

有很多处理 `Pin` 的 `unsafe` 方法，它们可以让你对指针和它指向的对象做任何事情，即使目标类型不是 `Unpin`。但正如我们在第 22 章中解释的一样，Rust 不能检查这些方法是否被正确使用，确保使用它们的代码的安全性变成了你的责任。

20.5 异步代码什么时候能带来帮助？

编写异步代码比编写多线程代码更加棘手。你需要使用正确的 I/O 和同步原语、手动打断长时间计算或者扔到其他线程去、并处理其他的细节例如 `pin` 这种多线程代码中没有的问题。

因此异步代码到底提供了什么样的优势？

你经常听到的两种说法经不起仔细考察：

- “异步代码非常适合 I/O。”这并不是很正确。如果你的应用把时间都花费在等待 I/O 上，那么将它变为异步的并不能让 I/O 运行得更快。目前通常使用的异步 I/O 接口在任何方面都不比相应的同步接口更高效。两种方式下操作系统要做的工作是一样的。（事实上，一个还没有准备好的异步 I/O 操作必须在之后重试，因此它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。”在类似 JavaScript 和 Python 的语言中，这可能确实是对的。在那些语言中，程序中使用 `async/await` 作为并发的形式：有单个线程负责执行，中断只发生在 `await` 表达式中，因此通常不需要用互斥锁来保证数据的一致性：只要不在使用它的中途 `await` 就行！当任务切换只发生在显式允许的地方的时候，你的代码可以更容易地理解。

但这个观点并不适用于 Rust，在 Rust 中多线程基本不是问题。一旦你的程序能通过编译，它就可以免于数据竞争。不确定的行为仅限于同步原语例如互斥锁、通道、原子量，等等。因此异步代码在帮助你了解其他线程何时可能影响到你的代码行为这方面没有独特的优势；在所有 `safe` 的 Rust 代码中这些都很清楚。

当然，Rust 的异步支持在和多线程一起使用时真的非常出色。如果放弃它确实很遗憾。

因此，异步代码的真正优势是什么？

- 异步任务消耗更少的内存。在 Linux 上，一个线程开始时的内存占用是 20KB⁴。`future` 可以小得多：我们的聊天服务器的 `future` 只有几百字节，并且随着 Rust 编译器的改进还可以变得更小。
- 异步任务可以更快地创建。在 Linux 上，创建一个线程需要大概 15μs。`spawn` 一个异步任务只需要大约 300ns，几乎只有 $\frac{1}{50}$ 的时间。
- 异步任务的上下文切换比操作系统线程的上下文切换更快，在 Linux 上分别是 0.2μs 和 1.7μs⁵。然而，这是两者的最佳情况：如果切换是因为 I/O 就绪，两者都需要花费 1.7μs。切换是否发生在不同处理器上的线程或者任务间也会产生重要的影响：处理器核心之间的通信非常慢。

这提示我们异步代码可以解决什么样的问题。例如，一个异步服务器可能为每一个任务使用更少的内存，因此可以处理更多同时的连接。（这可能也是异步代码因“适合 I/O”而享有盛誉的地方。）或者，如果你的设计可以自然地组织为很多互相通信的独立任务，那么更低的开销、更短的创建时间、快速的上下文切换都是重要的优势。这就是为什么聊天服务器是异

⁴ 这包括内核内存和为线程分配的物理页，而不是虚拟的、尚未分配的页面。macOS 和 Windows 上这个数字也相差不多。

⁵ Linux 上下文切换以前也是大概 0.2μs 的范围，直到内核因为处理器的安全缺陷而被迫使用更慢的技术。

步编程的经典案例，但多人游戏和网络路由可能也是合适的使用场景。

在其他情况下，适合使用异步的条件并不是很清楚。如果你的程序有一个线程池进行繁重的计算或者因为等待 I/O 完成而闲置，那么上面列出的优势可能对性能并没有很大的影响。你必须优化你的计算，找到更快的网络连接，或者进行一些其他能真正影响到限制因素的改进。

在实践中，我们可以找到的每个大规模服务器的说明都强调了测量、调整和不懈努力以识别和消除任务之间的竞争源头的重要性。一个异步架构不能帮你跳过其中任何一点。事实上，虽然有很多现成的工具可用于评估多线程程序的行为，但 Rust 的异步任务对这些工具是不可见的，因此还需要它们自己专门的工具。（正如一位睿智的长者所说：“现在你有了两个问题”。）

即使你现在并不使用异步代码，知道还有这种选择也很 nice，如果你以后有幸比现在忙很多的话。

Chapter 21

宏

A cento (from the Latin for “patchwork”) is a poem made up entirely of lines quoted from another poet.

——Matt Madden

Rust 支持宏 (*macros*)，这是一种普通函数无法做到的扩展语言的方式。例如，我们已经看到过 `assert_eq!` 宏，它可以方便地用来测试：

```
assert_eq!(gcd(6, 10), 2);
```

这也可以写成一个泛型函数，但 `assert_eq!` 可以做到几件函数做不到的事。其中一点是当断言失败时，`assert_eq!` 会生成包含断言所在的文件名和行号的错误消息。函数没有办法获得这些信息，但宏可以，因为它们工作的方式完全不同。

宏是一种缩写。编译期间在类型检查之前、更在生成任何机器码之前，每一个宏调用都会被展开 (*expand*)——即被替换为一些 Rust 代码。上面的宏调用会展开成类似这样的代码：

```
match (&gcd(6, 10), &2) {
    (left_val, right_val) => {
        if !(*left_val == *right_val) {
            panic!("assertion failed: `{} == {}`",
                  left, right);
        }
    }
}
```

`panic!` 也是一个宏，它自己会展开成更多 Rust 代码（这里没有展示）。那些代码里用到了两个别的宏：`file!()` 和 `line!()`。一旦 crate 中的每一个宏调用都被完全展开，Rust 会进入编译的下一个阶段。

在运行时，一个断言失败看起来像这样（并且可能指示 gcd() 函数中的一个 bug，因为 2 是正确结果）：

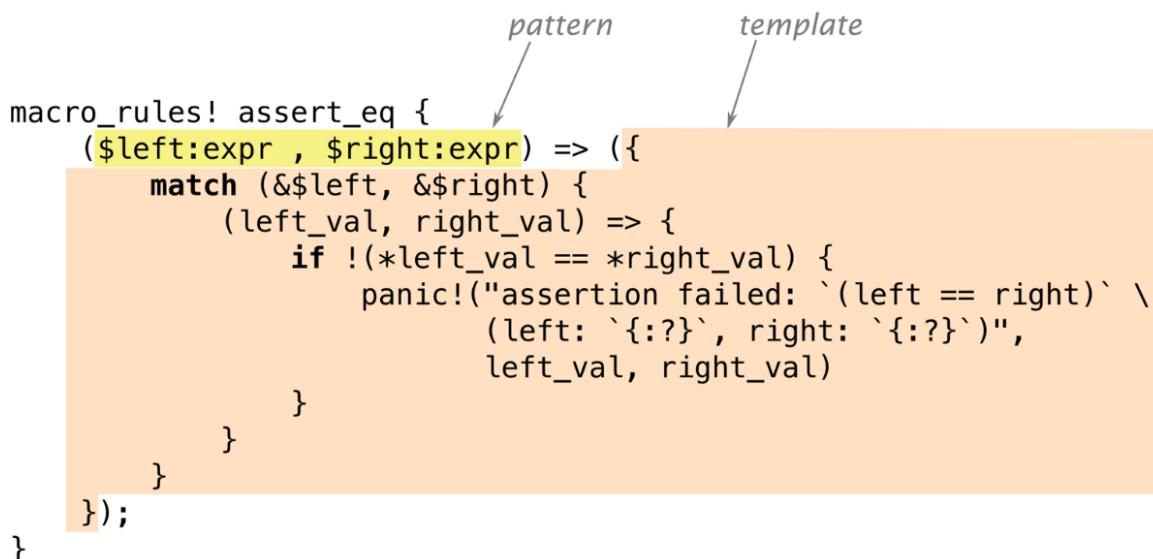
```
thread 'main' panicked at 'assertion failed: `(left == right)`, (left: '17',
right: '2')', gcd.rs:7
```

如果你是从 C++ 来的，你可能经历过一些宏的糟糕体验。Rust 的宏采用了一种不同的方式，类似于 Scheme 的 syntax-rules。相比于 C++ 的宏，Rust 的宏和语言的其他部分集成得更好，并且因此不容易出错。宏调用总是用感叹号标记，这样当你阅读代码时它们会很显眼，并且不会在你想调用函数时偶然错误地调用成了宏。Rust 的宏从来不会插入不匹配的花括号或圆括号。并且 Rust 的宏带有模式匹配，这使得编写既可维护又易于使用的宏变得更容易。

在本章中，我们将通过几个简单的示例展示如何编写宏。但和 Rust 中的很多部分一样，宏值得深入理解，所以我们将介绍一个更复杂的宏的设计，它允许我们直接在我们的程序中嵌入 JSON 字面量。但除了本书中介绍的部分之外，宏还有更多的内容。因此我们将以一些进一步学习的点结束，既有我们将在这里向你展示的高级技巧，也有功能更强大的被称为过程宏 (*procedural macros*) 的设施。

21.1 宏基础

图 21-1 展示了 assert_eq! 宏的部分源码。



```
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)
                }
            }
        }
    });
}
```

图 21-1: assert_eq! 宏

`macro_rules!` 是 Rust 中定义宏的主要方法。注意，宏定义里 `assert_eq!` 后边没有`!`：只有在调用宏时才需要`!`，定义时不需要。

并不是所有的宏都是以这种方式定义的：少数的宏，例如 `file!`、`line!` 和 `macro_rules!` 自身，是编译器内建的。我们将在本章的末尾讨论另一种方法，称为过程宏。但我们的主要精力还是集中在 `macro_rules!`，这是（目前为止）最容易的编写自己的宏的方法。

一个用 `macro_rules!` 定义的宏完全靠模式匹配工作。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。

顺带一提，你可以使用方括号或者花括号来代替模式或模板两侧的圆括号，对 Rust 来说它们并没有任何区别。另外，当你调用一个宏时，这些都是等价的：

```
assert_eq!(gcd(6, 10), 2);
assert_eq![gcd(6, 10), 2];
assert_eq!{gcd(6, 10), 2}
```

唯一的不同是花括号后边的分号是可选的。为了方便，我们在调用 `assert_eq!` 时使用圆括号，调用 `vec!` 时使用方括号，`macro_rules!` 时使用花括号。

现在我们展示了一个宏展开的简单示例和生成这个宏的定义，我们可以深入了解它工作的细节：

- 我们将详细地解释 Rust 是怎么发现和展开你的程序中的宏定义的。
- 我们将指出在根据宏模板生成代码时的一些细节之处。
- 最后，我们将展示模式如何处理重复的结构。

21.1.1 宏展开基础

Rust 会在编译的前期展开宏。编译器会从头到尾读取源码，在这个过程中定义和展开宏。宏只有在定义之后才能被调用，因为 Rust 会立刻展开每一个宏调用，而不会去看程序的剩余部分。（相反，函数和其他的 `item` 的定义不需要按照任何顺序。调用一个后面才定义的函数也是 OK 的。）

当 Rust 展开一个 `assert_eq!` 宏调用时，行为和执行一个 `match` 表达式非常像。Rust 首先根据模式匹配参数，如图 21-2 所示：

宏的模式是 Rust 的 mini 语言。它们本质上是匹配代码的正则表达式。但普通的正则表达式是操作字符，而模式操作 `token`(词元)——数字、变量名、标点符号等 Rust 中的构建块。这意

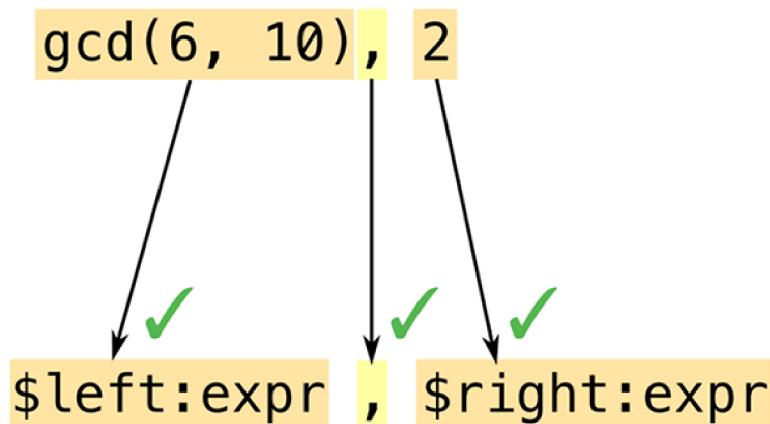


图 21-2: 展开一个宏, 第 1 部分: 用模式匹配参数

味着你可以在宏模式中自由地使用注释和空格来提升它们的可读性。注释和空格不是 token, 因此它们不会影响到匹配。

正则表达式和宏模式的另一个重要不同之处是圆括号、方括号、花括号在 Rust 中总是成对出现。这一点会在宏展开之前就检查, 不仅仅是在宏模式中, 而且是在整个语言中。

在这个例子中, 我们的模式包含了 *fragment(片段)* `$left:expr`, 这告诉 Rust 去匹配一个表达式 (在这个例子中, 就是 `gcd(6, 10)`) 并把它复制到名称 `$left`。然后 Rust 用 `gcd` 调用后边的逗号匹配模式中的逗号。类似于正则表达式, 模式只有少数特殊字符会触发有趣的匹配行为; 其它的所有字符, 例如逗号, 都必须逐字匹配相同的字符, 否则就会匹配失败。

这个模式中的两个代码片段都是 `expr` 类型: 它们代表表达式。我们将在 [片段类型](#) 中看到其他类型的代码片段。

因为这个模式匹配到了所有的参数, Rust 会展开相应的 *template(模板)* ([图 21-3](#)):

Rust 会用匹配阶段发现的代码片段来替换 `$left` 和 `$right`。

一个常见的错误是在输出模板中包含片段的类型: 写 `$left:expr` 而不是 `$left`。Rust 不会立刻检测出这种错误。它会把 `$left` 看做一个整体, 然后把 `:expr` 看作和模板中其他部分一样的东西: 要包含在宏的输出中的词元。因此只有当你调用这个宏时这个错误才会出现; 然后它会生成错误的不能编译的输出。如果你在使用一个新的宏时得到了类似 `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, 可以检查下它是不是有这个错误。[\(调试宏提供了更多类似这种情况的建议。\)](#)

宏模板和 web 编程中常用的很多种模板语言中的任意一种都没有太大的区别。唯一的不同——也是很重要的一点——就是它的输出是 Rust 代码。

```

{
    match (&$left, &$right) {
        (left_val, right_val) => {
            if !(*left_val == *right_val) {
                panic!("assertion failed: `!(left == right)` \\
                    (left: `{:?}` , right: `{:?}` )", \\
                    left_val, right_val)
            }
        }
    }
}

```

replace with gcd(6, 10)
replace with 2

图 21-3: 展开一个宏, 第 2 部分: 填充模板

21.1.2 意外的结果

把代码片段插入模板和普通的处理值的代码有一些区别。这些区别一开始可能并不明显。我们之前看到的 `assert_eq!` 宏包含了一些有些奇怪的代码，这是宏编程特有的。让我们重点看看其中两个有趣的部分。

首先，为什么这个宏创建了两个变量 `left_val` 和 `right_val`? 为什么我们不能把模板简化成这样?

```

if !($left == $right) {
    panic!("assertion failed: `!(left == right)` \\
        (left: `{:?}` , right: `{:?}` )", $left, $right)
}

```

为了回答这个问题，尝试手动展开宏调用 `assert_eq!(letters.pop(), Some('z'))`。输出将会是什么? Rust 会把匹配到的表达式插入模板中的多个位置。看起来在构建错误消息时重新求值表达式是一个坏主意，原因不仅仅是因为它需要消耗两倍的时间：因为 `letters.pop()` 从 vector 中移除一个值，所以在我们第二次调用它时会产生一个和第一次不同的值! 这就是为什么真正的宏只计算一次 `$left` 和 `$right`，然后存储它们的值。

来到第二个问题：为什么这个宏借用了 `$left` 和 `$right` 的值的引用？为什么不直接把它们的值存进变量中，像这样？

```

macro_rules! bad_assert_eq {
    ($left:expr, $right:expr) => ({
        match ($left, $right) {
            (left_val, right_val) => {

```

```

        if !(left_val == right_val) {
            panic!("assertion failed" /* ... */);
        }
    }
}
});
}
}

```

对于我们展示过的特殊情况，即宏的参数是整数的情况下，这可以正常工作。但如果调用者传递了一个 `String` 变量作为 `$left` 或 `$right`，这段代码将会移动变量的值！

```

fn main() {
    let s = "a rose".to_string();
    bad_assert_eq!(s, "a rose");
    println!("confirmed: {} is a rose", s); // error: use of moved value "s"
}

```

因为我们不想让断言移动值，所以宏里使用了引用。

(你可能想知道为什么这个宏使用了 `match` 而不是 `let` 来定义变量。我们也想知道。事实证明并没有特殊的原因这么做。`let` 是等价的。)

简而言之，宏可能做出令人惊讶的行为。如果一个你写的宏附近发生了奇怪的事，那很可能是因为这个宏有问题。

C++ 里的这个经典 bug 你将不会看到：

```

// buggy C++ macro to add 1 to a number
#define ADD_ONE(n) n + 1

```

原因大部分的 C++ 程序员应该很熟悉了，并且不值得详细解释，类似 `ADD_ONE(1) * 10` 或者 `ADD_ONE(1 << 4)` 这样的代码可能会产生令人惊讶的行为。为了修复它，你需要在宏定义中加上更多括号。这在 Rust 中是不必要的，因为 Rust 宏和语言集成的更好。Rust 知道它什么时候是在处理表达式，因此在把一个表达式粘贴到另一个地方时它会自动添加括号。

21.1.3 重复

标准的 `vec!` 宏有两种形式：

```

// 重复一个值N次
let buffer = vec![0_u8; 1000];

// 一个逗号分隔的值的列表
let numbers = vec!["udon", "ramen", "soba"];

```

它可以像这样实现：

```
macro_rules! vec {
    ($elem:expr ; $n:expr) => {
        ::std::vec::from_elem($elem, $n)
    };
    ($($x:expr),*) => {
        <[_]>::into_vec(Box::new([$($x),*]))
    };
    ($($x:expr),+,) => {
        vec![ $($x),* ]
    };
}
```

这里有三个规则。我们将解释多个规则是如何工作的，然后依次看看每一个规则。

当 Rust 展开一个例如 `vec![1, 2, 3]` 的宏调用时，它首先尝试匹配参数 1, 2, 3 和第一条规则的模式，即 `$elem:expr ; $n:expr`。这会匹配失败：1 是一个表达式，但这个模式要求它后边有一个分号，然而并没有。因此 Rust 会移动到第二个规则，等等。如果没有规则可以匹配，就会报错。

第一个规则处理类似 `vec![0u8; 1000]` 这样的调用。恰好有一个标准（但不在文档里的）函数 `std::vec::from_elem`，正好可以完成我们需要的功能，因此这个规则很直观。

第二个规则处理 `vec!["udon", "ramen", "soba"]`。模式 `$($x:expr),*` 使用了一个我们之前没有提过的特性：重复。它匹配 0 个或多个逗号分隔的表达式。更一般地来说，语法 `$(PATTERN),*` 用来匹配任意逗号分割的列表，其中列表中的每一项与 PATTERN 匹配。

这里的 * 和正则表达式中的 * 有相同的含义（“0 次或多次”），尽管正则表达式没有特殊的 * 重复器。你也可以用 + 来要求至少一次匹配，或者用 ? 要求 0 次或 1 次匹配。[表 21-1](#) 列出了全套的重复模式。

代码片段 `$x` 不是单个表达式，而是一个表达式的列表。这个规则的模板也使用了重复语法：

```
<[_]>::into_vec(Box::new([$($x),*]))
```

再一次，恰好有标准方法可以满足我们的需要。这段代码创建了一个装箱的数组，然后使用 `[T]:::into_vec` 方法把这个装箱的数组转换成一个 `vector`。

开头的 `<[_]>`，是一种不常见的写法，它表示类型“某些东西的切片”，由 Rust 来推断元素类型。名字是普通标识符的类型可以直接在表达式中使用，但类似 `fn()`、`&str`，或者 `[_]` 这样的类型必须用尖括号包裹。

表 21-1: 重复模式

模式	含义
<code>\$(...)*</code>	匹配0次或多次，无分隔符
<code>\$(...),*</code>	匹配0次或多次，逗号分隔
<code>\$(...);*</code>	匹配0次或多次，分号分隔
<code>\$(...)+</code>	匹配1次或多次，无分隔符
<code>\$(...),+</code>	匹配1次或多次，逗号分隔
<code>\$(...);+</code>	匹配1次或多次，分号分隔
<code>\$(...)?</code>	匹配0次或1次，无分隔符
<code>\$(...),?</code>	匹配0次或1次，逗号分隔
<code>\$(...);?</code>	匹配0次或1次，分号分隔

重复模式出现在模板的末尾，即 `$($x),*`。这个 `$(...),*` 和我们在模式中看到的是相同的语法。它迭代 `$x` 匹配到的表达式列表并把它们全部插入模板，用逗号分隔。

在这种情况下看，重复的输出看起来和输入一样。但并不总是这样。我们可以编写类似这样的规则：

```
( $( $x:expr ),* ) => {
    let mut v = Vec::new();
    $( v.push($x); )*
    v
};

}
```

这里，模板中 `$(v.push($x);)*` 这一部分会对 `$x` 中的每个表达式插入一个 `v.push()` 调用。一个宏分支可以展开成一个表达式序列，但这里我们只需要单个表达式，所以我们把 vector 的处理包装在一个块中。

和 Rust 中其他部分不同，使用 `$(...),*` 并不能自动支持可选的尾部逗号。然而，有一种标准的技巧是通过添加一个额外的规则来支持尾部逗号。也就是我们的 `vec!` 宏的第三条规则所做的：

```
( $( $x:expr ),+ , ) => {      // 如果存在尾部的逗号,
    vec![ $($x ),* ]           // 重试没有它的情况
}
```

我们使用 `$(...),+`，来匹配一个有额外逗号的列表。然后，我们在模板中递归调用了 `vec!`，但排除了那个逗号。这一次第二条规则将会匹配。

21.2 内建的宏

Rust 编译器提供了几个宏，如果你要定义自己的宏，它们可能会发挥作用。这些宏都不能使用 `macro_rules!` 来实现。它们被硬编码进 `rustc`:

`file!(), line!(), column!()`

`file!()` 展开成一个字符串字面量：当前的文件名。`line!()` 和 `column!()` 展开成 `u32` 字面量，表示当前的行号和列号（从 1 开始计数）。

如果一个宏调用了另一个宏，那个宏又调用了另一个宏，这三个宏在不同的文件中，并且最后一个宏调用了 `file!()`, `line!()` 或者 `column!()`，它会展开成第一个宏调用的位置。

`stringify!(...tokens...)`

展开成一个包含给定 `token` 的字符串字面量。`assert!` 宏就是使用了它来生成一条包含了断言代码的错误信息。

参数中的宏调用不会被展开：`stringify!(line!())` 会展开为 "line!()"。

Rust 根据 `token` 构建字符串，因此生成的字符串里没有换行符或者注释。

`concat!(str0, str1, ...)`

连接参数展开为单个字符串字面量。

Rust 还定义了下面这些宏用来查询构建环境：

`cfg!(...)`

展开为一个 `bool` 常量，如果当前的构建环境满足括号里的条件则为 `true`。例如，如果在编译时启用了调试断言那么 `cfg!(debug_assertions)` 为真。

这个宏和属性中介绍过的 `#[cfg(...)]` 属性的语法完全相同，但它不是条件编译，而是得到一个 `true` 或者 `false`。

`env!("VAR_NAME")`

展开为一个字符串：在编译时该环境变量的值。如果这个变量不存在，会产生编译错误。

除了 Cargo 在编译 crate 时设置的几个有趣的环境变量之外，这个宏毫无价值。例如，为了得到 crate 当前的版本，你可以写：

```
let version = env!("CARGO_PKG_VERSION");
```

这些环境变量的完整列表见[Cargo 文档](#)。

```
option_env!("VAR_NAME")
```

它和 `env!` 宏一样，除了它返回一个 `Option<&'static str>`，当环境变量没有设置时返回 `None`。

还有更多内建的宏可以让你引入另一个文件中的代码或者数据：

```
include!("file.rs")
```

展开为指定文件的内容，必须是有效的 Rust 代码——表达式或者 `item` 的序列。

```
include_str!("file.txt")
```

展开成一个包含指定文件内容的 `&'static str`。你可以像这样使用它：

```
const COMPOSITOR_SHADER: &str =  
    include_str!("../resources/compositor.gls1");
```

如果文件不存在或者不是有效的 UTF-8，会产生编译错误。

```
include_bytes!("file.dat")
```

和上一个基本相同，除了它把文件当作二进制数据而不是 UTF-8 文本，结果是一个 `&'static [u8]`。

和所有的宏一样，这些宏也是在编译时进行处理。如果文件不存在或者不能被读取，就会编译失败。它们不可能在运行时出错。在任何情况下，如果文件名是一个相对路径，它会被解析为相对于当前文件所在的目录的路径。

Rust 还提供了几个方便的宏：

```
todo!(), unimplemented!()
```

这些等价于 `panic!()`，但用于表示不同的意图。`unimplemented!()` 出现在 `if` 分支、`match` 分支，以及其它还未处理的 `case` 中。它总是会 `panic`。`todo!()` 大致相同，但传达的意图是代码还没写完；一些 IDE 使用它来进行标记。

```
matches!(value, pattern)
```

比较一个值和一个模式，当它们匹配时返回 `true`，否则返回 `false`。它等价于写：

```
match value {  
    pattern => true,  
    _ => false  
}
```

如果你在寻找基本的编写宏的练习，这是一个很好的例子——尤其是你可以在标准库文档中看到它的实际实现非常简单。

21.3 调试宏

调试宏可能很有挑战性。最大的问题是在宏展开的过程中缺少可视性。Rust 总是展开所有宏，找到一些错误，然后打印出一条错误信息，但这个错误信息并没有显示出完整的展开后的代码！

这里有三个工具可以帮助你调试宏。（这些特性都是 unstable 的，但因为它们被设计为用在开发的过程中，而不是最后的代码中，因此在实践中这不是一个很大的问题。）

第一个也是最简单的一个，你可以让 `rustc` 显示你的代码在展开所有宏之后是什么样的。使用 `cargo build --verbose` 来看看 Cargo 是怎么调用 `rustc` 的。拷贝 `rustc` 的命令行并加上 `-Z unstable-options --pretty expanded` 选项。完全展开后的代码会输出到终端。不幸的是，只有当你的代码没有语法错误时这种方式才能生效。

第二，Rust 提供了一个 `log_syntax!()` 宏简单地在编译期把它的参数打印到终端。你可以使用它来进行 `println!` 风格的调试。这个宏需要 `#![feature(log_syntax)]` 特性标记。

第三，你可以让 Rust 编译器把所有宏调用记录到终端。在代码中插入 `trace_macros!(true)`，之后每当 Rust 展开一个宏时，它都会打印出宏的名字和参数。例如，考虑这个程序：

```
#![feature(trace_macros)]  
  
fn main() {  
    trace_macros!(true);  
    let numbers = vec![1, 2, 3];  
    trace_macros!(false);  
    println!("total: {}", numbers.iter().sum::<u64>());  
}
```

它会产生如下输出：

```
$ 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 ] )`
```

编译器会显示每一个宏调用的代码，包括展开之前和展开之后的代码。`trace_macros!(false);` 这一行关闭了追踪，因此 `println!()` 的调用不会被追踪。

21.4 构建 json! 宏

我们已经讨论了 `macro_rules!` 的核心特性。在这一节，我们将渐进式开发一个构建 JSON 数据的宏。我们将用这个例子来展示开发一个宏的过程、展示 `macro_rules!` 剩余的部分、并提供一些保证你的宏正确工作的建议。

回到第 10 章，我们使用了这个枚举来代表 JSON 数据：

```
#[derive(Clone, PartialEq, Debug)]
enum Json {
    Null,
    Boolean(bool),
    Number(f64),
    String(String),
    Array(Vec<Json>),
    Object(Box<HashMap<String, Json>>)
}
```

不幸的是，编写 `Json` 值的语法非常复杂：

```
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()))
]);
```

我们希望能使用更类似 JSON 的语法来实现同样的效果：

```
let students = json!([
  {
    "name": "Jim Blandy",
    "class_of": 1926,
    "major": "Tibetan throat singing"
  },
  {
    "name": "Jason Orendorff",
    "class_of": 1702,
    "major": "Knots"
  }
]);
```

我们要实现的是一个 `json!` 宏，它获取一个 JSON 值作为参数并展开为类似于上一个例子中的表达式。

21.4.1 片段类型

编写任何复杂宏的第一步都是指明如何匹配或解析期望的输入。

我们已经能看到这个宏将会有好几条规则，因为 JSON 数据中有几种不同的东西：对象，数组，数字等等。实际上，我们可能会猜测我们将为每一种 JSON 类型编写一条规则：

```
macro_rules! json {
  (null)      => { Json::Null };
  ([ ... ])  => { Json::Array(...); };
  ({ ... })  => { Json::Object(...); };
  (???)       => { Json::Boolean(...); };
  (???)       => { Json::Number(...); };
  (???)       => { Json::String(...); };
}
```

这并不完全正确，因为宏模式没法区分最后三种情况，但我们稍后将会看到怎么处理它们。前三种情况至少很明显地以不同的 token 开头，因此我们可以用这些 token 进行区分。

第一个规则已经可以工作了：

```
macro_rules! json {
  (null) => {
    Json::Null
  }
}
```

```
#[test]
fn json_null() {
    assert_eq!(json!(null), json::Null);      // passes!
}
```

为了支持 JSON 数组，我们可能要尝试匹配元素为 `expr`:

```
macro_rules! json {
    (null) => {
        Json::Null
    };
    ([] $($element:expr),*) => {
        Json::Array(vec![ $($element ),* ])
    };
}
```

不幸的是，这不能匹配所有的 JSON 数组。这里有一个会产生问题的例子:

```
#[test]
fn json_array_with_json_element() {
    let macro_generated_value = json!(
        [
            // 有效的 JSON, 但不匹配`$element:expr`
            {
                "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),*` 意思是“一个逗号分隔的 Rust 表达式的列表”。但很多 JSON 值尤其是对象，不是有效的 Rust 表达式，它们不能匹配。

因为你想要匹配的代码并不都是表达式，所以 Rust 还支持其他几种片段类型，如表 21-2 所示。

表 21-2: `macro_rules!` 支持的片段类型

片段类型	匹配的内容 (及示例)	后边可以跟
<code>expr</code>	一个表达式: <code>2 + 2</code> , <code>"udon"</code> , <code>x.len()</code>	<code>=></code> , ;
<code>stmt</code>	一个表达式或者声明, 不包括尾部的分号 (很难用; 尝试用 <code>expr</code> 或 <code>block</code> 代替)	<code>=></code> , ;
<code>ty</code>	一个类型: <code>String</code> , <code>Vec<u8></code> , <code>(&str, bool)</code> , <code>dyn Read + Send</code>	<code>=></code> , ; { [: > as where
<code>path</code>	一个路径 (已讨论过): <code>ferns</code> , <code>::std::sync::mpsc</code>	<code>=></code> , ; { [: > as where
<code>pat</code>	一个模式 (已讨论过): <code>_</code> , <code>Some(ref x)</code>	<code>=></code> , = if in
<code>item</code>	一个 item (已讨论过): <code>struct Point { x: f64, y: f64 }</code> , <code>mod ferns;</code>	任何东西
<code>block</code>	一个块 (已讨论过): <code>{ s += "ok\n"; true }</code>	任何东西
<code>meta</code>	一个属性的内容 (已讨论过) : <code>inline</code> , <code>derive(Copy, Clone)</code> , <code>doc="3D models."</code>	任何东西
<code>ident</code>	一个标识符: <code>std</code> , <code>Json</code> , <code>longish_variable_name</code>	任何东西
<code>literal</code>	一个字面值: <code>1024</code> , <code>"Hello, world!"</code> , <code>1_000_000f64</code>	任何东西
<code>lifetime</code>	一个生命周期: <code>'a</code> , <code>'item</code> , <code>'static</code>	任何东西
<code>vis</code>	可见性说明符: <code>pub</code> , <code>pub(crate)</code> , <code>pub(in module::submodule)</code>	任何东西
<code>tt</code>	一个 token 树: <code>;</code> , <code>>=</code> , <code>{}</code> , <code>[0 1 (+ 0 1)]</code>	任何东西

表格中的大部分选项严格要求 Rust 语法。`expr` 类型只匹配 Rust 表达式（不是 JSON 值），`ty` 只匹配 Rust 类型，等等。它们是不可扩展的：没有办法定义一种新的可以用 `expr` 识别的运算符或关键字。我们不能用这些来实现匹配任意 JSON 数据。

最后两个，`ident` 和 `tt`，支持匹配不是 Rust 代码的参数。`ident` 匹配任何标识符。`tt` 匹配单个 `token` 树：一对正确匹配的括号：`(...)`，`[...]`，`{...}`，以及其中的任何内容，包括嵌套的 `token` 树；或者一个没有括号的单个 `token`，例如 `1926` 或 `"Knots"`。

`token` 树正是我们的 `json!` 宏所需要的。每一个 JSON 值都是单个 `token` 树：数字、字符串、`bool` 值、`null` 都是单个 `token`，对象和数组是 `token` 树。因此我们可以写出像这样的模式：

```
macro_rules! json {
    (null) => {
        Json::Null
    };
    ([ $( $element:tt ),* ]) => {
        Json::Array(...)

    };
    ({ $($key:tt : $value: tt ),* }) => {
        Json::Object(...)

    };
    ($other:tt) => {
        ... // TODO: 返回数字、字符串、bool 值
    };
}
```

这个版本的 `json!` 宏可以匹配任何 JSON 数据。现在我们只需要产生正确的 Rust 代码。

为了保证 Rust 在未来可以添加新的语法特性而不会破坏任何你今天写的宏，Rust 限制了模式中紧跟在片段之后的 `token`。[表 21-2](#) 中“后边可以跟”这一列展示了哪些 `token` 可以跟在模式后面。例如，模式 `$x:expr ~ $y:expr` 是错的，因为 `~` 不允许出现在 `expr` 后面。模式 `$vars:pat => $handler:expr` 是可以的，因为 `$vars:pat` 后跟的是箭头 `=>`，它允许跟在 `pat` 后面，而 `$handler:expr` 后面没有跟任何东西，这种情况总是允许的。

21.4.2 宏中的递归

我们已经见过了一个在宏里调用自身的小例子：我们的 `vec!` 的实现使用了递归来支持尾部的逗号。这里我们可以展示一个更加显著的例子：`json!` 需要递归调用它自身。

我们可能会尝试在不递归的情况下支持 JSON 数组，像这样：

```
([ $( $element:tt ),* ]) => {
    Json::Array(vec![ $( $element ),* ])
};
```

但这段代码并不能工作。这样只是把 JSON 数据 (`$element` token 树) 粘贴到了 Rust 表达式之中，但它们是两种不同的语言。

我们需要把数组中的每一个元素从 JSON 格式转换为 Rust 代码。幸运的是，有一个宏可以做到这件事：就是我们正在写的这个宏！

```
([ $( $element:tt ),* ]) => {
    Json::Array(vec![ $( json!($element) ),* ])
};
```

这种方式还可以支持对象：

```
{( $key:tt : $value:tt ),* } => {
    Json::Object(Box::new(vec![
        $( ( $key.to_string(), json!($value) ),* )
    ].into_iter().collect())))
};
```

默认情况下编译器会把宏的递归次数上限设置为 64。对于 `json!` 宏的常规使用这肯定已经足够了，但有时复杂的递归宏会达到这个上限。你可以在使用这个宏的 crate 的开头加上这一行来调整上限

```
#![recursion_limit = "256"]
```

我们的 `json!` 宏基本已经完成了。剩下的只有支持布尔、数字和字符串值。

21.4.3 宏和 trait

编写复杂的宏时总是会遇到困难。重要的是要记住，宏并不是解决问题的唯一途径。

这里，我们需要支持 `json!(true)`、`json!(1.0)`、`json!("yes")`，把这些值转换成对应的 `Json` 值。但宏并不擅长区分类型。假如我们这么写：

```
macro_rules! json {
    (true) => {
        Json::Boolean(true)
    };
    (false) => {
        Json::Boolean(false)
    };
    ...
}
```

显然这种方案是行不通的。布尔类型只有两个值，但数字和字符串有很多值。

幸运的是，有一种标准的把多种类型的值转换成另一种特定类型的方法：`From trait`。我们可以简单地为几种类型实现这个 trait：

```

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，所有的12种数字类型应该有非常相似的实现，因此编写一个宏来避免复制粘贴是有意义的：

```

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);

```

现在我们可以使用 `Json::from(value)` 来把 `value` 转换成 `Json`。在我们的宏中，看起来像这样：

```
( $other:tt ) => {
    Json::from($other) // 处理布尔/数字/字符串
};
```

向我们的 `json!` 宏中添加了这个规则之后，它就可以通过我们目前写过的所有测试了。把所有部分汇总起来，它现在看起来像这样：

```
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/数字/字符串
    };
}
```

事实证明，这个宏意外地支持在 JSON 数据中使用变量甚至任意的 Rust 表达式，这是一个有趣的特性：

```
let width = 4.0;
let desc =
    json!({
        "width": width,
        "height": (width * 9.0 / 4.0)
   });
```

因为 `(width * 9.0 / 4.0)` 是用括号括起来的，所以它是单个的 token 树，因此在解析这个对象时宏可以成功地用 `$value:tt` 匹配它。

21.4.4 作用域和 hygiene

在编写宏时一个令人惊讶的问题是它们将来自不同范围的代码粘贴在一起。因此接下来的几页将介绍 Rust 处理作用域的两种方式：一种用于局部变量和参数，另一种用于其他的东

西。

为了展示为什么会有这个问题，让我们重写解析 JSON 对象的规则（之前展示过的 `json!` 宏的第三条规则），让它不再使用临时的 vector。我们可以像这样写：

```
({ $($key:tt) : $value:tt },*) => {
    {
        let mut fields = Box::new(HashMap::new());
        $($fields.insert($key.to_string(), json!{$value})); )*
        Json::Object(fields)
    }
};
```

现在我们不再使用 `collect()`，而是通过重复调用 `.insert()` 方法来填充 `HashMap`。这意味着我们需要把 map 存储在一个临时的变量中，这里命名为 `fields`。

但如果调用 `json!` 宏的代码中恰好也有一个叫 `fields` 的变量，会发生什么呢？

```
let fields = "Fields, W.C.";
let role = json!({
    "name": "Larson E. Whipsnade",
    "actor": fields
});
```

展开这个宏时将会把两段代码拼在一起，两段代码中都用了变量 `fields` 来表示不同的东西！

```
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)
};
```

看起来好像只要宏里使用了临时变量就无法避免这个问题，并且你可能已经正在思考可能的解决方案了。也许我们应该把 `json!` 宏中的变量重命名为一般情况下调用者不会传入的名字：例如用 `__json$fields` 代替 `fields`。

这里令人惊讶的是宏可以正常工作。Rust 会为你重命名变量！这个特性一开始在 Scheme 宏中实现，被称为 *hygiene*，因此 Rust 被认为有 *hygienic* 宏。

理解 hygiene 宏的最简单的方式就是想象每一次展开宏时，展开的部分都被染成了不同的颜色。

不同颜色的变量，就好像有不同的名字一样：注意这里宏的调用者传入并被粘贴到输出中

```
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)
};
```

的代码，例如“name”和“actor”，保持了它们原本的颜色（黑色）。只有宏模板生成的 token 被染成了另一种颜色。

现在这里有一个叫 `fields` 的变量（在调用者中声明）和另一个叫 `fields` 的变量（宏引入的）。因为这两个名字的颜色不同，所以这两个变量不会混淆。

如果一个宏真的需要引用一个在调用者作用域里的变量，那么调用者必须向宏传递这个变量的名字。

（染色的比喻并不意味着 hygiene 真的就是这么工作的。实际的机制要更聪明一些：不管是什颜色，如果两个标识符引用了宏和调用者作用域里的同一个变量，那么就认为它们是相同的。但在 Rust 中像这样的例子非常少见。如果你理解了上面的例子，那么就使用 hygiene 宏来说已经足够了。）

你可能已经注意到了很多别的标识符也被染上了一种或多种的颜色：例如 `Box`、`HashMap`、`Json` 等。就算没有颜色，Rust 也可以辨认出这些类型的名字。这是因为 Rust 中的 hygiene 仅限于局部变量和参数。对于常量、类型、宏、模块、静态变量、宏名来说，Rust 是“色盲”。

这意味着如果我们的 `json!` 宏被用在了 `Box`、`HashMap` 或 `Json` 不在作用域里的模块中时，这个宏将不能工作。我们将在下一节中展示怎么避免这个问题。

首先，我们将考虑一个 Rust 的 hygiene 会产生阻碍的例子，并且我们需要解决它。假设我们有很多函数都包含下面这一行代码：

```
let req = ServerRequest::new(server_socket.session());
```

如果复制粘贴这一行将会很痛苦。我们可以使用宏来代替吗？

```
macro_rules! setup_req {
    () => {
        let req = ServerRequest::new(server_socket.session());
    }
}

fn handle_http_request(server_socket: &ServerSocket) {
    setup_req!(); // 声明`req`，使用了`server_socket`
    ... // 使用`req`的代码
}
```

显然这么写不能工作。必须要让宏里的 `server_socket` 这个名字指向函数里声明的参数 `server_socket`, `req` 变量恰好相反。但 hygiene 会保护宏里的名字不会和其他作用域里的名字“碰撞”——即使这正是你想要的行为。

解决方案是把你计划在宏里宏外都要使用的变量全部传进宏里：

```
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
}
```

因为 `req` 和 `server_socket` 现在由函数提供，所以它们的“颜色”和那个作用域一样。

hygiene 让宏的使用变得更繁琐了一点，但这是一个特性，而不是 bug：知道了它们不会在背后扰乱局部变量之后，可以更容易地推理 hygiene 宏。如果你在一个函数里搜索一个类似 `server_socket` 的标识符，你将会发现所有使用了它的地方，包括宏调用。

21.4.5 导入和导出宏

因为宏在编译的早期被展开，此时 Rust 还不知道你的项目的完整的模块结构，因此编译器有特殊的方法来导入和导出它们。

在一个模块中可见的宏自动地在它的子模块中可见。为了从一个模块“逆向”导出到它的父模块，可以使用 `#[macro_use]` 属性。例如，假设我们的 `lib.rs` 看起来像这样：

```
#[macro_use] mod macros;
mod client;
mod server;
```

`macros` 模块中定义的所有的宏都被导入到了 `lib.rs`，因此它们对 crate 中的其他部分都可见，包括在 `client` 和 `server` 中。

使用 `#[macro_export]` 标记的宏自动地变为 `pub` 并可以和其他 item 一样通过路径引用。

例如，`lazy_static` crate 提供了一个叫 `lazy_static` 的被 `#[macro_export]` 标记的宏。为了在你自己的 crate 中使用这个宏，你可以写：

```
use lazy_static::lazy_static;
lazy_static!{ }
```

一旦一个宏被导入之后，就可以像其他的 item 一样使用它：

```
use lazy_static::lazy_static;

mod m {
    crate::lazy_static!{ }
}
```

当然，实际上这也意味着你的宏可能会在其他的模块中被调用。因此一个导出的宏不应该依赖任何作用域中的东西——没有办法预测使用它的作用域中会有什么。甚至标准 prelude 的特性也可以被遮蔽。

因此，宏应该使用绝对路径来表示它用到的任何名称。为此 `macro_rules!` 提供了特殊的片段 `$crate`。它和 `crate` 不同，后者是一个可以在任何地方的路径中使用的关键字，而不仅仅是在宏里。`$crate` 的含义类似于定义这个宏的 crate 的根模块的绝对路径。我们可以使用 `$crate::Json` 来代替 `Json`，这样即使 `Json` 没有被导入也可以工作。`HashMap` 可以改为 `::std::collections::HashMap` 或者 `$crate::macros::HashMap`。在后者的情况下，我们需要重新导出 `HashMap`，因为 `$crate` 不能用来访问 crate 的私有特性。它实际上会被展开成类似 `::jsonlib` 的普通路径。可见性规则不受影响。

在将宏移动到了它自己的 `macros` 模块，并修改它使用 `$crate` 之后，它看起来像这样。这是最终的版本：

```
// macro.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)
};
}
}

```

因为`.to_string()`方法是标准的`ToString` trait 的一部分，所以我们也使用了`$crate`来引用它，并使用了我们在[完全限定方法调用](#)中介绍过的语法：

`$crate::macros::ToString::to_string($key)`。在我们的例子中，为了让宏能正常工作，这并不是必须的，因为`ToString`在标准 prelude 里。但如果你在调用一个 trait 的方法，并且这个 trait 可能在调用处的作用域里，那么完全限定方法调用将是最好的方式。

21.5 在匹配时避免语法错误

下面的宏看起来是有原因的，但它会给 Rust 带来一些麻烦：

```

macro_rules! complain {
    ($msg:expr) => {
        println!("Complaint filed: {}", $msg);
    };
    (user : $userid:tt , $msg:expr) => {
        println!("Complaint from user {}: {}", $userid, $msg);
    };
}

```

假如我们像这样调用它：

```
complain!(user: "jimb", "the AI lab's chatbots keep picking on me");
```

以人眼来看，显然这会匹配第二个模式。但 Rust 会首先尝试第一个规则，尝试用`$msg:expr`匹配所有的输入。这时事情会变得糟糕起来。`user: "jimb"`显然不是一个表达式，因此我们会得到一个语法错误。Rust 拒绝掩盖语法错误——宏已经足够难调试了。它会立刻报错并中断编译。

如果模式中的其他 token 匹配失败，rust 会移动到下一条规则。只有语法错误会导致终止，并且只有当尝试匹配片段时才会出现语法错误。

这里的问题并不难理解：我们尝试在一条错误的规则中用 \$msg:expr 匹配一个片段。它会匹配失败，因为我们没想过用这条规则来匹配，调用者是希望用另一条规则来匹配。有两种避免这个错误的方式：

第一，避免易混淆的规则。例如，我们可以修改宏，让每一个模式都以一个不同的标识符开头：

```
macro_rules! complain {
    ($msg : $msg:expr) => {
        println!("Complaint filed: {}", $msg);
    };
    ($user : $userid:tt , $msg : $msg:expr) => {
        println!("Complaint from user {}: {}", $userid, $msg);
    };
}
```

当宏的参数以 msg 开头时，我们会得到规则 1。当它们以 user 开头时，我们会得到规则 2。我们现在可以在尝试匹配一个片段之前得到正确的规则。

另一种避免语法错误的方式是把更加具体的规则放在前面。把 user : 规则放在第一个就可以解决 complain! 的问题，会导致语法错误的规则不会再被触发。

21.6 macro_rules! 之外

宏模式可以解析比 JSON 更复杂得多的输入，但我们会发现它的复杂性很快就脱离了控制。

Daniel Keep 等人的 [The Little Book of Rust Macros](#) 是一本优秀的讲解高级 macro_rules! 编程的书。这本书非常的清楚和聪明，它比我们这里更加详细地描述了宏展开的每一个概念。它还提出了几种非常巧妙的技术，将 macro_rules! 模式作为一种编程语言来解析复杂的输入。这一点我们不太热衷。请小心使用。

Rust 1.15 引入了一种单独的机制称为过程宏 (*procedural macros*)。过程宏支持扩展 #[derive] 属性来实现自定义派生，如图 21-4 所示。还可以创造自定义的属性以及新的和之前介绍的 macro_rules! 宏调用方式一样的宏。

```
#[derive(Copy, Clone, PartialEq, Eq, IntoJson)]
struct Money {
    dollars: u32,
    cents: u16,
}
```

A diagram showing a code snippet. The code defines a struct Money with fields dollars (u32) and cents (u16). It uses the #[derive] attribute to implement Copy, Clone, PartialEq, Eq, and IntoJson traits. A callout arrow points from the text "custom derive" to the IntoJson trait in the #[derive] attribute.

图 21-4: 通过 #[derive] 属性来调用一个 IntoJson 过程宏

并不存在 `IntoJson` trait，但没关系：一个过程宏可以使用这种方式来插入它需要的代码（在这个例子中，可能是 `impl From<Money> for Json { ... }`）。

过程宏之所以被称为“过程”是因为它被实现为一个 Rust 函数，而不是声明规则集。这个函数通过一个简单的抽象层和编译器交互，并且可以非常复杂。例如，`diesel` 数据库的库使用了过程宏来连接到一个数据库并在编译期根据该数据库的模式生成代码。

因为过程宏要和编译器内部交互，编写高效的宏需要了解编译器的工作原理，这超出了本书的范围。然而，它在[在线文档](#)中有充分的介绍。

也许当读到这里时，你已经决定了你很讨厌宏。那该怎么办呢？一种替代方案是使用构建脚本来生成 Rust 代码。[Cargo 文档](#)一步步地展示了怎么做到这点。它包含编写一个程序生成你想要的 Rust 代码、在 `Cargo.toml` 中添加一行以在构建过程中运行这个程序、使用 `include!` 把生成的代码引入你的 crate。

Chapter 22

unsafe 代码

*Let no one think of me that I am humble or weak or passive;
Let them understand I am of a different kind: dangerous to my enemies, loyal to my friends.
To such a life glory belongs.*

——Euripides, Medea

系统编程的乐趣在于，在每一个安全的语言和精心设计的抽象之下，都是不安全的机器语言和比特位。你也可以在 Rust 中写出这样的代码。

到目前为止本书中介绍的语言部分，例如类型、生命周期、约束检查等都可以自动保证你的程序完全没有内存错误和数据竞争。但这种自动的技术有它的局限性；Rust 并不能识别出来很多有价值的技术是安全的。

unsafe 代码让你可以告诉 Rust，“我要使用一些你不能保证安全的特性”。把一个块或者函数标记为 *unsafe* 之后，你就可以调用标准库中的 *unsafe* 函数、解引用 *unsafe* 指针、调用其他语言例如 C 和 C++ 编写的函数等。Rust 的其他安全性检查依然生效：类型检查、生命周期检查、约束检查等仍然和之前一样。*unsafe* 代码只是允许了一小部分额外的特性。

正是因为有了允许超出 safe Rust 界限的能力，Rust 才能实现它自身的大部分基础特性，和 C/C++ 一样，Rust 也被用来实现它自己的标准库。*unsafe* 代码可以让 `Vec` 类型更高效地管理它的缓冲区；让 `std::io` 模块和操作系统交互；让 `std::thread` 和 `std::sync` 模块提供并发原语。

本章介绍了 *unsafe* 特性的一些基础：

1. Rust 的 `unsafe` 块区分开了普通的 safe Rust 代码和使用 *unsafe* 特性的代码。
2. 你可以把函数标记为 `unsafe`，提醒调用者他们必须遵守的一些额外约束来避免未定义行为。

3. 原始指针和它们的方法允许对内存进行不受限的访问，并允许你构建 Rust 的类型系统可能禁止的数据结构。Rust 的引用虽然安全，但却是受限制的，原始指针正如每个 C 或 C++ 程序员所知，是一个强大而锋利的工具。
4. 理解未定义行为的定义将会帮助你理解为什么它们比得到错误结果还要糟糕的多。
5. unsafe trait，类似于 unsafe 函数，隐含了每个实现（而不是每个调用者）都要遵守的规则。

22.1 unsafe 从何而来？

在本书的开头处，我们曾经展示过一个 C 程序，它以一种非常令人惊讶的方式崩溃。这是因为它违背了 C 标准的一个规则。你可以在 Rust 中实现相同的效果：

```
$ 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)
$
```

这个程序借用了局部变量 `a` 的一个可变引用，然后把它转换成了 `*mut usize` 类型的原始指针，然后使用了它的 `offset` 方法来产生一个指向三个字之后的位置的指针。这里恰巧存储了 `main` 的返回地址。这个程序用一个常量覆盖了返回地址，因此从 `main` 返回之后程序的行为就很奇怪。这个崩溃之所以可行，是因为程序错误使用了 `unsafe` 的特性——在这个例子中，就是解引用原始指针的能力。

一个 `unsafe` 特性通常隐含了一份合约 (*contract*)：即一组 Rust 不能自动强制，但你必须遵守才能避免未定义行为的规则。

一份合约超出了普通的类型检查和生命周期检查，它们隐含着一些 `unsafe` 特性特定的规则。通常来说，Rust 本身完全不知道这些合约，它们只在 `unsafe` 特性的文档里得到解释。例

如，原始指针类型的合约是禁止解引用一个指向的位置超出原来指向物的末尾的原始指针。这个例子中的表达式 `*ptr.offset(3) = ...` 打破了这个合约。但是，正如上面所示，Rust 没有任何警告，成功编译了这个程序：它的安全检查并没有检测出来这个违规行为。当你使用 unsafe 特性时，你作为程序员，需要负责检查你的代码遵守了它们的合约。

很多特性如果想正确使用，都要遵守一定的规则。但这些规则并不是我们这里说的合约，除非它们可能会导致未定义行为。未定义行为是一种 Rust 假设你的代码中绝对不会出现的行为。例如，Rust 假设你不会用别的值覆盖函数的返回地址。通过了 Rust 通常的安全检查并且遵守了使用到的 unsafe 特性的合约的代码不可能会出现这样的行为。因为这个程序违反了原始指针的合约，因此它的行为变得未定义，并最终崩溃。

如果你的代码出现了未定义行为，说明你打破了你负责的一部分，Rust 拒绝预测结果。从系统库的深处抛出来一个错误并崩溃是一种可能的结果；把你的计算机的控制权交给攻击者是另一种可能的结果。不同的 Rust 版本可能也会有不同的行为。然而，有时未定义行为不一定会产生可见的结果。例如，如果这里的 `main` 函数永远不会返回（可能调用了 `std::process::exit` 来提前终止程序），那么错误的返回地址也无关紧要。

你只能在 unsafe 块或者 unsafe 函数里使用 unsafe 特性；我们将在接下来的小节介绍它们。它们让 unsafe 特性不容易被忽略：通过强迫你写一个 unsafe 块或者函数，Rust 能确保你知道你的代码可能要遵守一些额外的规则。

22.2 unsafe 块

unsafe 块看起来就像一个以 `unsafe` 关键字开头的普通块，区别在于你可以在 unsafe 块里使用 unsafe 特性：

```
unsafe {
    String::from_utf8_unchecked(ascii)
}
```

如果没有块前面的 `unsafe` 关键字，Rust 将会禁止使用 `from_utf8_unchecked`，因为它是一个 `unsafe` 的函数。在 `unsafe` 块中，你可以随意使用它。

和普通的 Rust 块一样，`unsafe` 块的值也是最后一条表达式的值，或者是 `()`。这个例子中 `String::from_utf8_unchecked` 的调用提供了块的值。

一个 `unsafe` 块为你解锁了 5 个额外的功能：

1. 你可以调用 `unsafe` 函数。每一个 `unsafe` 函数都有它自己的合约，这取决于它的功能。
2. 你可以解引用原始指针。`safe` 代码可以传递、比较、通过引用转换创建原始指针，但只有 `unsafe` 代码才可以使用它们来访问内存。我们将在 [原始指针](#) 中详细介绍原始指针并解

释如何安全地使用它们。

3. 你可以访问 `union` 的字段，尽管编译器不能确定它们含有相应类型的有效值。
4. 你可以访问可变的 `static` 变量。正如全局变量中解释的一样，Rust 不能确保什么时候有线程正在使用可变的 `static` 变量，因此它们的合约要求你要确保所有的访问都是正确同步的。
5. 你可以访问通过 Rust 的外部函数接口声明的函数和变量。即使它们是不可变的，也会被认为是 `unsafe` 的，因为它们是使用其他语言编写的，这些语言可能不遵守 Rust 的安全规则。

把 `unsafe` 特性约束在 `unsafe` 块里并不会真的阻止你做任何想做的事。你完全只需要在你的代码里加上一个 `unsafe` 块，然后就可以继续了。这个规则的作用主要是为了把人类的注意力吸引到那些 Rust 不能保证安全性的代码上：

1. 你不会意外地使用到 `unsafe` 特性，然后发现你要为甚至不知道它的存在的合约负责。
2. 一个 `unsafe` 块可以吸引 reviewer 更多的注意力。一些项目甚至有一些自动化流程来确保这一点，例如标记出会影响 `unsafe` 块的代码来吸引更多注意力。
3. 当你正在考虑编写一个 `unsafe` 块时，你可以花费一点时间来问问自己你的任务是否真的需要这些特性。如果是为了性能，你是否有测量数据表明这真的是一个性能瓶颈？可能有一种在 `safe` Rust 中也可以实现相同效果的方法。

22.3 示例：一个高效的 ASCII 字符类型

这里有一个示例，`Ascii` 是一个字符串类型，它确保它的内容总是有效的 ASCII 字符。这个类型使用一个 `unsafe` 特性来提供到 `String` 的 0 开销转换：

```
mod my_ascii {
    /// 一个 ASCII 编码的字符串
    #[derive(Debug, Eq, PartialEq)]
    pub struct Ascii(
        // 它必须只存有有效的 ASCII 文本：从`0`到`0x7f`的字节序列
        Vec<u8>
    );

    impl Ascii {
        /// 从`bytes`中的 Ascii 文本创建一个`Ascii`。
        /// 如果`bytes`中含有任何非 ASCII 字符就返回一个`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，我们会给出不能转换的 vector。
// 它应该实现`std::error::Error`，这里为了简洁就省略了。
#[derive(Debug, Eq, PartialEq)]
pub struct NotAsciiError(pub Vec<u8>);

// 安全、高效的转换，使用 unsafe 代码实现。
impl From<Ascii> for String {
    fn from(ascii: Ascii) -> String {
        // 如果这个模块没有 bug 的话，这里就是安全的，
        // 因为有效的 ASCII 文本也是有效的 UTF-8 文本。
        unsafe { String::from_utf8_unchecked(ascii.0) }
    }
}
...
}
```

这个模块的关键是 `Ascii` 类型的定义。这个类型本身被标记为 `pub`，以让它在 `my_ascii` 模块外可见。但它的 `Vec<u8>` 元素不是 `public` 的，因此只有 `my_ascii` 模块里的方法可以创建一个 `Ascii` 值或者访问它的元素。这完全控制了模块里哪些代码是公开的哪些是不公开的。只要 `public` 的构造器和方法能确保新创建的 `Ascii` 值是有效的，并始终保持有效，那么程序的其他部分就不可能违反规则。并且 `public` 的构造器 `Ascii::from_bytes` 确实小心地检查了给定的 `vector` 来确保能从它构建出一个有效的 `Ascii`。出于简洁性的考虑，我们并没有展示出每一个方法，但你可以想象还有一些处理文本的方法，这些方法同样确保 `Ascii` 的值总是有效的 ASCII 文本，就像 `String` 的方法确保它的内容总是有效的 UTF-8.

这样的安排让我们可以非常高效地为 `String` 实现 `From<Ascii>`。`unsafe` 函数

`String::from_utf8_unchecked` 获取一个字节 `vector` 并根据它的内容构建一个 `String`，并且不检查它的内容是否是有效的 UTF-8 文本；这个函数的合约就是调用者要负责这一点。幸运的是，`Ascii` 类型强迫的规则正是满足 `from_utf8_unchecked` 的合约所需的条件。正如我们在 [UTF-8](#) 中解释的一样，任何有效的 ASCII 文本都是有效的 UTF-8 文本，因此 `Ascii` 内部的 `Vec<u8>` 可以立刻作为一个 `String` 的缓冲区使用。

有了这些定义之后，你可以写这样的代码：

```
use my_ascii::Ascii;
```

```

let bytes: Vec<u8> = b"ASCII and ye shall receive".to_vec();

// 这里的调用没有任何内存分配或者文本拷贝，只进行一次扫描。
let ascii: Ascii = Ascii::from_bytes(bytes)
    .unwrap(); // 我们已经知道了bytes是没问题的。

// 这里的调用是0开销的：没有内存分配、拷贝、扫描。
let string = String::from(ascii);

assert_eq!(string, "ASCII and ye shall receive");

```

使用 `Ascii` 不需要 `unsafe` 块。我们已经使用 `unsafe` 操作实现了一个 `safe` 的接口，并且安排好只依赖模块自己的代码而不是用户的行为来满足它的合约。

`Ascii` 只是一个 `Vec<u8>` 的包装，并在模块里隐藏了一些强迫它的内容需要满足的规则。一个这样的类型被称为 *newtype*，它是 Rust 中非常普遍的一种模式。Rust 自己的 `String` 类型就是用完全相同的方式定义的，区别只有它的内容被限制为 UTF-8，而不是 ASCII。事实上，这是标准库里 `String` 的定义：

```

pub struct String {
    vec: Vec<u8>,
}

```

在机器语言的层面上，是完全没有 Rust 的类型信息的，`newtype` 和它的元素有完全相同的内存表示，因此构建一个 `newtype` 完全不需要任何额外的机器指令。在 `Ascii::from_bytes` 中，表达式 `Ascii(bytes)` 只是表明 `Vec<u8>` 现在的内存表示持有的是一个 `Ascii` 值。类似的，`String::from_utf8_unchecked` 在内联的情况下可能不包含任何机器指令：它只表明 `Vec<u8>` 现在被认为是一个 `String`。

22.4 unsafe 函数

`unsafe` 函数的定义就像一个以 `unsafe` 开头的普通函数。`unsafe` 函数的函数体自动被认为是一个 `unsafe` 块。

你只能在 `unsafe` 块里调用 `unsafe` 函数。这意味着将一个函数标记为 `unsafe` 可以警告调用者这个函数有一个额外的合约，必须满足这个合约才能避免未定义行为。

例如，这里有一个新的 `Ascii` 类的构造器，这个构造器从一个字节 vector 构建一个 `Ascii`，并且不检查内容是否是有效的 ASCII：

```

// 这段代码必须放在`my_ascii`模块中。
impl Ascii {

```

```

    /// 从`bytes`构建一个`Ascii`值，不检查`bytes`是否是有效的 ASCII 文本。
    ///
    /// 这个函数直接返回一个`Ascii`，而不是像`from_bytes`一样返回一个
    /// `Result<Ascii, NotAsciiError>`。
    ///
    /// # 安全性
    ///
    /// 调用者必须确保`bytes`只包含 ASCII 字符：每个字节都不大于 0x7f。
    /// 否则，最后的结果是未定义的。
    pub unsafe fn from_bytes_unchecked(bytes: Vec<u8>) -> Ascii {
        Ascii(bytes)
    }
}

```

如果调用 `Ascii::from_bytes_unchecked` 的代码总是知道 vector 中只包含有效的 ASCII 字符，那么 `Ascii::from_bytes` 里的检查就只是在浪费时间，并且调用者还必须处理永远不会出现的 `Err` 结果。`Ascii::from_bytes` 可以简化这种情况下的调用和错误处理。

但之前我们曾经强调过 `Ascii` 的 public 构造器和方法保证 `Ascii` 的值是有效的的重要性。`from_bytes_unchecked` 是不是没有遵守这个规则？

不完全是：`from_bytes_unchecked` 把它的责任通过它的合约交给了调用者。这个合约的存在正是它应该被标记为 `unsafe` 的原因：虽然这个函数本身没有进行 `unsafe` 的操作，但它的调用者必须遵守一些 Rust 不能强制的规则才能避免未定义行为。

你真的能通过打破 `Ascii::from_bytes_unchecked` 的合约来导致未定义行为吗？是的。你可以像下面这样构造一个无效的 `String`：

```

// 想象这个vector是一些我们认为会产生 ASCII 文本的操作的结果,
// 但这个操作出错了。
let bytes = vec![0xf7, 0xbf, 0xbf, 0xbf];
let ascii = unsafe {
    // 当`bytes`含有非 ASCII 值时这个unsafe 的合约就被打破了
    Ascii::from_bytes_unchecked(bytes)
};

let bogus: String = ascii.into();

// `bogus` 现在包含无效的UTF-8。
// 解析它的第一个字符会产生一个无效的 Unicode 码点的`char`,
// 这是未定义行为，因此Rust 不知道这个断言的行为会是什么样的。
assert_eq!(bogus.chars().next().unwrap() as u32, 0xffff);

```

在特定版本的 Rust 和特定的平台上，这个断言会输出下面的错误信息并失败：

```
thread 'main' panicked at 'assertion failed: `left == right`'  
left: `2097151`  
right: `2097151`, src/main.rs:42:5
```

这两个数字在我看来似乎是相等的，但这不是 Rust 的问题；这是之前的 `unsafe` 块的问题。当我们说未定义行为会导致无法预料的结果时，这就是其中一种情况。

这个例子展示了两个有关 bug 和 `unsafe` 代码的关键事实：

1. `unsafe` 块之前发生的 *bug* 可能会打破合约。一个 `unsafe` 块是否会导致未定义行为可能不仅仅取决于这个块本身，还取决于提供它要操作的值的代码。你的 `unsafe` 代码依赖的任何东西都是和安全相关的。只有当模块的其他部分正确的维护了 `Ascii` 相关的内容时，基于 `String::from_utf_unchecked` 的 `Ascii` 到 `String` 的转换才是安全的。
2. 打破合约的结果可能在你离开 `unsafe` 块之后才会出现。不遵守 `unsafe` 特性而导致的未定义行为通常不会在 `unsafe` 块本身里出现。如上面所示，构造一个 `bogus String` 可能不会有问题是，直到程序执行了一段时间后才出现问题。

本质上讲，Rust 的类型检查、借用检查和其他的静态检查都是在分析你的程序并尝试证明它不可能会出现未定义行为。当 Rust 成功编译你的程序时，这意味着它成功地证明了这一点。一个 `unsafe` 块是这个证明中的例外：等于你在告诉 Rust “它没有问题，相信我”。你的声明是否正确可能依赖程序的任何会影响到 `unsafe` 块的部分，并且出错时产生的结果也可能出现在任何被 `unsafe` 块影响的地方。`unsafe` 关键字也是在提醒你，你无法完全享受到它的安全检查的好处。

如果可以选择的话，你应该尽量选择使用安全的没有合约的接口。它们更容易使用，因为用户可以依赖 Rust 的安全检查来保证他们的代码不可能出现未定义行为。即使你的实现使用了 `unsafe` 特性，最好使用 Rust 的类型、生命周期和模块系统来满足它们的合约，同时只使用你自己可以保证的东西，而不是把责任传递给调用者。

不幸的是，在实际编程中遇到懒得解释它们的合约的 `unsafe` 函数并不罕见。它们期望你能依靠自己的经验和知识自己推导出这些规则。如果你曾经在使用一个 C 或 C++ 的 API 时因为不知道自己用的对不对而感到不安，那么你知道那是一种什么感觉。

22.5 `unsafe` 块还是 `unsafe` 函数？

你可能会想知道是使用 `unsafe` 块还是直接把整个函数标记为 `unsafe`。我们推荐的方法是首先对该函数做出决定：

1. 如果这个函数可能被误用，可以成功编译但可能导致未定义行为，那么你应该将它标记为 `unsafe`。正确使用这个函数的规则就是它的合约，也正是合约的存在让它变得 `unsafe`。

2. 否则，这个函数是 safe 的：没有调用能让它产生未定义行为。它不应该被标记为 `unsafe`。

这个函数在函数体里是否使用 `unsafe` 特性并不这个重要，关键是合约的存在。之前我们展示过一个没有使用 `unsafe` 特性的 `unsafe` 函数，也展示过一个使用了 `unsafe` 特性的 `safe` 函数。

不要只因为你在函数体里使用了 `unsafe` 特性就把 `safe` 的函数标记为 `unsafe`。这只会让函数更难用，并且迷惑调用者，让他以为这里有一个合约。正确的做法是使用一个 `unsafe` 块，即使这个块就是整个函数体。

22.6 未定义行为

在引言中，我们说过术语未定义行为意思是“Rust 假设你的代码绝对不会出现的行为”。这是一个奇怪的说法，尤其是我们通过其他语言积累的经验告诉我们这些行为确实会偶然出现。为什么这个概念有助于规定 `unsafe` 代码的义务？

编译器是从一种编程语言到另一种语言的转换器。Rust 编译器接收一个 Rust 程序并把它翻译成等价的机器语言程序。但两个差别这么大的语言，我们说它们等价到底是什么意思？

幸运的是，相比于语言学家，对程序员来说这个问题简单的多。如果两个程序执行时总是有相同的可见的行为，那么我们说这两个程序是等价的：它们进行相同的系统调用、以等价的方式和外部函数交互等等。这有点像程序的图灵测试：如果你不能分辨出你是在和原始的程序交互还是和翻译后的程序交互，那么它们就是等价的。

现在考虑下面的代码：

```
let i = 10;
very_trustworthy(&i);
println!("{}", i * 100);
```

即使我们完全不知道 `very_trustworthy` 的定义，我们可以看到它只接收一个 `i` 的共享引用，因此这个调用不可能改变 `i` 的值。因此传递给 `println!` 的值将总是 1000，Rust 可以把这段代码翻译成机器语言，就好像我们写的是：

```
very_trustworthy(&10);
println!("{}", 1000);
```

这个转换后的版本和原本的有相同的可见的行为，而且它可能还要更快一点。但只有在我们认为它真的和原始的版本相同的时候考虑它的性能才有意义。如果 `very_trustworthy` 被定义成这样呢？

```
fn very_trustworthy(shared: &i32) {
    unsafe {
        // 把这个共享引用转换成一个可变的指针。
```

```
// 这是未定义行为。  
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：它把 `i` 的值改成了 `20`，但 `i` 是以共享的方式借用的。因此，现在对这个函数的调用者进行转换会产生非常明显的效果：如果 Rust 转换了这段代码，程序会打印出 `1000`；如果它保留了这段代码并使用 `i` 的新值，它会打印出 `2000`。在 `very_trustworthy` 中打破共享引用的规则意味着共享引用的行为并不会如调用者所预期。

这类问题出现在几乎每种 Rust 会尝试进行的转换中。包括把一个函数内联到调用者中、当调用结束后控制流返回到调用处，等等。但是我们以一个打破了这种假设的例子来开始这一章。

对 Rust（或其他任何语言）来说基本不可能判断对程序的转换是否能保持它的含义，除非它可以信任语言的基础特性的行为和预期一样。它们是否会进行这种转换不仅依赖于眼下的代码，还可能依赖潜在的很远之外的代码。为了对你的代码做一点改动，Rust 必须假设程序的其他部分的行为都是正常的。

然后这里是 Rust 对行为正确程序的规则：

1. 程序绝对不能读取未初始化的内存。
2. 程序绝对不能创建无效的基础值：
 - (a) 引用、box 或函数指针为 `null`
 - (b) 既不是 `0` 也不是 `1` 的 `bool` 值
 - (c) 判断值无效的 `enum`
 - (d) 无效的 `char` 值，非 Unicode 码点
 - (e) 内容不是有效的 UTF-8 的 `str` 值
 - (f) 虚表或者切片长度无效的胖指针
 - (g) `!` 类型的任何值
3. 第 5 章 中介绍的引用的规则必须要遵守。不能有引用的生命周期比引用的对象更长；共享的访问是只读访问，可变的访问是独占的访问。
4. 程序绝对不能解引用空的、错误对齐的、或悬垂的指针。
5. 程序绝对不能用一个指针去访问超出这个指针关联的对象的内存范围之外的位置。我们将在 安全地解引用原始指针 中详细解释这个规则。
6. 程序必须没有数据竞争。数据竞争发生在两个线程在未同步的情况下访问相同的内存位置，并且其中至少有一个访问是写入访问。

7. 程序绝对不能在一个其他语言通过外部函数接口所进行的调用中进行栈展开，正如在[栈展开](#)中解释的一样。
8. 程序必须遵守标准库函数的合约。

由于我们还没有 Rust 的 `unsafe` 语义的完整模型，这个列表可能会随着时间的推移而改变，但这些内容很可能仍然是禁止的。

任何违反这些规则的行为都可能构成未定义行为，还会阻止 Rust 优化你的程序并把它们转换成机器语言。如果你打破了最后一个规则把无效的 UTF-8 传递给 `String::from_utf8_unchecked`，那么之后可能 `2097151` 不等于 `2097151`。

不使用 `unsafe` 特性的 Rust 代码只要能编译就能被保证遵守上述所有规则（假设编译器没有 bug，我们正在逐渐靠近这个目标，但曲线和渐近线永远不会相交）。只有当你使用 `unsafe` 特性时，这些规则才会变成你自己的责任。

在 C 和 C++ 中，即使你的程序没有报错成功通过了编译也意义不大；正如我们在这本书的引言中解释的，即使是用那些保持高标准代码的广受欢迎的库编写的最好的 C 和 C++ 程序在实践中也会出现未定义行为。

22.7 unsafe trait

unsafe trait 是一种特殊的 trait，它们有一个 Rust 无法检查或者强制实现必须遵守的规则，实现必须遵守这些规则才能避免未定义行为。为了实现一个 `unsafe trait`，你必须将实现标记为 `unsafe` 的。理解 trait 的合约并确保你的实现满足合约是你的责任。

如果一个泛型函数的类型参数的约束中含有 `unsafe trait`，那么通常这个函数自身也会使用 `unsafe` 特性，并且它们只依赖这些 `unsafe trait` 的合约来满足自己的合约。一个错误的 trait 实现可能会导致这样的函数出现未定义行为。

`std::marker::Send` 和 `std::marker::Sync` 是 `unsafe trait` 的典型例子。这些 trait 并没有定义任何方法，因此可以很容易地为任何类型实现它们。但它们确实有合约：`Send` 要求实现者可以安全地移动到另一个线程中，`Sync` 要求实现者必须能安全地通过共享引用在线程间共享。为一个不恰当的类型实现 `Send` 将会使 `std::sync::Mutex` 不能再保证没有数据竞争。

这里有个简单的例子，Rust 标准库曾经包含了一个叫 `core::nonzero::Zeroable` 的 `unsafe trait`，它用来表示那些可以通过把所有字节置为 0 来安全地初始化的类型。举个例子，把一个 `usize` 置 0 是可以的，但把一个 `&T` 置 0 会产生空引用，如果解引用就会崩溃。对于那些实现了 `Zeroable` 的类型，有一些可行的优化：你可以使用 `std::ptr::write_bytes` (`memset` 在 Rust 中的等价函数) 或者一个分配置 0 内存页的系统调用来快速地初始化它们的数组。`(Zeroable` 是 `unstable` 的，并且在 Rust 1.26 中被移到只在 `num crate` 中内部使用，但它

是一个好的、简单的、真实的例子。)

`Zeroable` 是一个类型标记 trait，没有方法或者关联类型：

```
pub unsafe trait Zeroable {}
```

为恰当的类型实现这个 trait 非常的直观：

```
unsafe impl Zeroable for u8 {}
unsafe impl Zeroable for i32 {}
unsafe impl Zeroable for usize {}
// 其他的整数类型同理
```

有了这些定义，我们可以编写一个函数，它可以快速地分配一个给定长度的 `Zeroable` 类型的 vector：

```
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
}
```

这个函数首先用给定的容量创建一个空的 `Vec`，然后调用 `write_bytes` 用 0 来填充未初始化的缓冲区。（`write_bytes` 函数把 `len` 看做 `T` 类型元素的数量，而不是字节的数量，因此这个调用确实填充了整个缓冲区。）`vector` 的 `set_len` 方法只修改它的长度，不对缓冲区进行任何操作；这是 unsafe 的，因为你必须保证新的缓冲区空间内都包含正确初始化的 `T` 类型的值。但这正是 `T: Zeroable` 约束的：一个 0 字节的内存块代表一个有效的 `T` 值。我们对 `set_len` 的使用是安全的。

这里，我们来使用它：

```
let v: Vec<usize> = zeroed_vector(100_000);
assert!(v.iter().all(|&u| u == 0));
```

显然 `Zeroable` 必须是一个 unsafe 的 trait，因为一个不遵守合约的实现可能导致未定义行为：

```
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：解引用空指针
```

Rust 不知道 `Zeroable` 表示什么，所以它不能分辨出哪些类型的实现是不恰当的。和其他的 `unsafe` 特性一样，理解并遵守 `unsafe trait` 的合约是你的责任。

注意 `unsafe` 代码绝对不能依赖正确实现的普通的 `safe trait`。例如，假设有一个 `std::hash::Hasher` trait 的实现简单地返回一个随机的哈希值，并且这个值和要哈希的值没有一点关系。这个 trait 要求同样的值两次被哈希时必须产生相同的哈希值，但这个实现并不满足这个要求，它很显然是错误的。但因为 `Hasher` 并不是 `unsafe` 的 trait，`unsafe` 代码在使用这个哈希器的时候不应该出现未定义行为。`std::collections::HashMap` 类型是被精心编写的，它遵守所有用到的 `unsafe` 特性的合约，不管哈希器的行为是什么样的。具体来说，即使哈希表不能正确地工作：查找可能失败，表项可能随机出现或者消失，整个表也不会出现未定义行为。

22.8 原始指针

Rust 中原始指针指的是没有约束的指针。你可以使用原始指针来组织 Rust 的普通指针类型无法做到的数据结构，例如双向链表或者任意的图对象。但因为原始指针太过灵活，Rust 无法分辨出你是否正在安全地使用它们，因此你只能在 `unsafe` 块中解引用它们。

原始指针基本等价于 C 或 C++ 中的指针，因此在和这些语言编写的代码交互时原始指针非常有用。

有两种原始指针：

1. `*mut T` 是可以修改引用对象的指针。
2. `*const T` 是只能读取引用对象的指针。

(没有 `*T` 类型，你必须指定 `const` 或者 `mut`。)

你可以通过转换引用来创建原始指针，并使用 `*` 操作符来解引用它：

```
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);
```

和 box 指针以及引用不同，原始指针可以为 null，类似 C 中的 NULL 和 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、传递、比较原始指针都是 safe 的。只有解引用原始指针才是 unsafe 的。

unsized 类型的原始指针是胖指针，就像相应的引用或 Box 指针一样。一个 *const [u8] 的指针除了地址之外还包括长度，一个 trait 对象的原始指针例如 *mut dyn std::io::Write 指针还附带一个虚表。

尽管 Rust 在很多场景可以隐式解引用 safe 的指针类型，但原始指针的解引用必须是显式的：

1. . 运算符不会隐式解引用原始指针，你必须用 (*raw).field 或者 (*raw).method(...)。
2. 原始指针并没有实现 Deref，因此强制解引用并不适用于它们。
3. == 和 < 之类的运算符以地址比较原始指针：只有两个原始指针指向同一个内存位置它们才是相等的。与此类似，哈希一个原始指针会对它指向的地址进行哈希，而不是对它指向的对象的值进行哈希。
4. 格式化 trait 例如 std::fmt::Display 会自动解引用，但无法处理原始指针。例外的是 std::fmt::Debug 和 std::fmt::Pointer，它们会以 16 进制地址的形式显示原始指针，不会解引用它们。

和 C/C++ 中的 + 运算符不同，Rust 的 + 运算符不能用于原始指针，但你可以使用原始指针的 offset、wrapping_offset 或者更方便的 add、sub、wrapping_add、wrapping_sub 方法对它们进行算数操作。offset_from 方法可以给出两个指针之间的距离，不过我们必须确保起点和终点在相同的内存区域（例如在同一个 Vec）里：

```
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` 不需要隐式转换，只要指明类型就够了。Rust 隐式地把引用强制转换为原始指针（当然反过来不行）。

`as` 运算符允许几乎把任何引用转换成原始指针或者转换两个原始指针类型。然而，你必须把一个复杂的转换拆分成一系列简单的转换。例如：

```
&vec![42_u8] as *const String; // 错误：无效转换
&vec![42_u8] as *const Vec<u8> as *const String; // 允许
```

注意 `as` 不能把原始指针转换为引用。这样的转换是 `unsafe` 的，而 `as` 应该保证是 `safe` 的操作。要想做到这一点，你必须解引用原始指针（在一个 `unsafe` 块中）然后借用得到的值的引用。

这么做的时候一定要小心：这种方式产生的引用将会有无限的生命周期：它的生存时间没有任何限制，因为原始指针并没有给 Rust 提供推断这一点的信息。在后面的一个 libgit2 的安全接口一节中，我们将展示几个例子来演示如何正确地约束生命周期。

很多类型都有 `as_ptr` 和 `as_mut_ptr` 方法可以返回它们的内容的原始指针。例如，数组的切片和字符串会返回它们的第一个元素的指针，一些迭代器会返回它们要产生的下一个元素的指针。拥有所有权的指针类型例如 `Box`、`Rc` 和 `Arc` 有 `into_raw` 和 `from_raw` 函数可以转换成或转换自原始指针。其中一些方法的合约有一些令人惊讶的要求，因此在使用之前要仔细阅读它们的文档。

你也可以把整数转换成原始指针，尽管你唯一可以信任的整数是从之前的指针得到整数。
示例：`RefWithFlag` 以这种方式使用了原始指针。

和引用不同，原始指针既没有实现 `Send` 也没有实现 `Sync`。因此，任何包含原始指针的类型默认都没有实现这两个 trait。在线程间发送或者共享原始指针本质上并没有什么不安全的，毕竟，不管它们去了哪，在解引用它们的时候仍然需要一个 `unsafe` 块。但考虑到原始指针通常扮演的角色，语言的设计者认为默认是这样会更有帮助。我们已经在 `unsafe trait` 中讨论过如何自己实现 `Send` 和 `Sync` 了。

22.8.1 安全地解引用原始指针

这里有一些安全使用原始指针的基本规则：

1. 解引用空指针或悬垂指针是未定义行为，指向未初始化内存或超出作用域的值的指针也是如此。
2. 解引用没有按照类型正确对齐的指针是未定义行为。

3. 你可以从解引用原始指针获得的值借用引用，不过只有当这么做满足第 5 章中介绍的引用安全性规则时才可以：引用不能超出被引用对象的生命周期、共享的访问是只读的访问、可变的访问是独占的访问。（这个规则很容易在无意中被违反，因为原始指针通常被用来创建非标准共享或所有权的数据结构。）
4. 只有当一个原始指针指向的对象是正确的该类型的值时你才能使用它指向的对象。例如，你必须确保解引用一个 `*const char` 返回一个正确的 Unicode 码点。
5. 在使用原始指针的 `offset` 和 `wrapping_offset` 方法时你必须确保最后指向的位置还在一开始指向的那个对象的值或者内存块里。
如果你先把一个指针转换成整数，然后进行任何的算术运算，再把它转换回指针，那么结果必须是 `offset` 的规则允许你产生的指针。
6. 如果你对原始指针指向的对象赋值，你必须保证不违反其中任何一个类型的不变量。例如，如果你有一个指向一个 `String` 的字节的 `*mut u8` 指针，你对这个 `u8` 赋的值必须保证 `String` 持有的仍是有效的 UTF-8。

除了借用规则之外，这些都是在 C 和 C++ 中使用指针时必须要遵守的基本规则。

不能违背类型的不变量的原因应该很清楚。很多 Rust 的标准类型在实现里都使用了 unsafe 代码，但仍然提供了 safe 的接口。它们假设 Rust 的安全检查、模块系统和可见性规则都没有被违反。使用原始指针来绕开这些保护措施可能会导致未定义行为。

完整又精确的原始指针的合约很难简单地说清楚，也可能会随着语言的改进发生改变。但这里列出的原则应该能保证代码是安全的。

22.8.2 示例：RefWithFlag

这里有一个例子展示了怎么使用原始指针来进行一些经典的位级操作并把它包装为一个完全安全的 Rust 类型¹。这个模块定义了一个类型 `RefWithFlag<'a, T>`，它持有一个 `&'a T` 和一个 `bool`，类似于元组 `(&'a, bool)` 一样。但它只占用一个机器字而不是两个。这种技术通常用在垃圾回收器和虚拟机中，其中的某些类型（例如表示任意对象的类型）的实例非常多，以至于如果能减小一个机器字就可以大大减少内存占用：

```
mod ref_with_flag {
    use std::marker::PhantomData;
    use std::mem::align_of;

    /// 在单个字中存放一个`&T`和一个`bool`。
    /// 类型`T`必须是至少两字节对齐的。
    ///
}
```

¹在我们的领域它确实是一个经典。

```
/// 如果你喜欢偷取指针的第 $2^n$ 位，那么现在你可以安全地做到这一点！
/// (“但这么做并没有那么刺激...”)
pub struct RefWithFlag<'a, T> {
    ptr_and_bit: usize,
    behaves_like: PhantomData<&'a T> // 不占用空间
}

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
    }
}
```

这段代码利用了很多类型必须放在偶数内存地址处的特点：因为一个偶数地址的最低有效位总是0，因此我们可以在这里存储一些东西，然后只要把它置0就能还原原来的地址。并不是所有类型都能这么做；例如，类型u8和(bool, [i8; 2])可以被放在任何地址处。但我们可以在初始化时检查类型的对齐然后拒绝不适用的类型。

我们可以像这样使用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);
```

`RefWithFlag::new` 接收一个引用和一个 `bool` 值，之后断言引用的类型是合适的，然后把引用转换为一个原始指针，再转换成 `usize`。不管我们在什么处理器上编译，`usize` 类型都足够存储任何指针，因此把一个原始指针转换成 `usize` 再转换回来是良定义的。一旦我们有了一个 `usize`，我们知道它肯定是偶数，因此我们可以使用 `|` 位或运算符把它和 `bool` 值结合在一起，当然要先把 `bool` 值转换成 0 或者 1。

`get_flag` 方法从一个 `RefWithFlag` 中提取出 `bool` 的部分。这很简单，只要看看最低位是否非零。

`get_ref` 方法从一个 `RefWithFlag` 中提取出引用部分。首先，它把 `usize` 的最低位置 0 后转换为原始指针。`as` 运算符不能把原始指针转换成引用，但我们可以解引用原始指针（当然是在 `unsafe` 块中）然后借用引用。借用原始指针指向对象的引用会产生一个没有生命周期约束的引用：如果可行的话，Rust 会给任何引用赋予一个生命周期，用于检查代码。然而，通常有一些生命周期会更加准确，因此也能检查出更多的错误。在这种情况下，因为 `get_ref` 的返回类型是 `&'a T`，Rust 会看到引用的生命周期和 `RefWithFlag` 的生命周期参数 `'a` 相同。这正是我们想要的：这正是一开始的那个引用的生命周期。

在内存中，一个 `RefWithFlag` 看起来就像一个 `usize`：因为 `PhantomData` 是一个 0 字节类型，`behaves_like` 字段不会占用任何空间。但为了让 Rust 知道如何处理使用了 `RefWithFlag` 的代码的生命周期，`PhantomData` 是必须的。想象一下如果没有 `behaves_like` 字段，这个类型的定义会变成什么样：

```
// 不能通过编译。
pub struct RefWithFlag<'a, T: 'a> {
    ptr_and_bit: usize
}
```

在第 5 章中，我们指出过任何包含引用的结构体绝对不能比它们借用的值生存的更久，否则引用会变成悬垂指针。结构体必须在它的字段上遵守这个限制。这当然也适用于 `RefWithFlag`：在我们刚才看过的示例代码中，`flagged` 必须不能比 `vec` 生存的更久，因为 `flagged.get_ref()` 返回一个它的引用。但这里的缩减版的 `RefWithFlag` 类型根本不包含任何引用，甚至都没用到生命周期参数 `'a`，它只是一个 `usize`。Rust 应该如何知道该对 `flagged` 的生命周期进行什么限制？包含一个 `PhantomData<&'a T>` 字段可以告诉 Rust 处理 `RefWithFlag<'a, T>` 时，就好像它还含有一个 `&'a T` 一样，这并不会影响结构体在内存中的表示。

尽管 Rust 并不真的知道发生了什么，它也会尽力帮你实现这些。如果你省略了 `behaves_like` 字段，Rust 将会报错说生命周期 `'a` 和 `T` 都没有用到，然后建议使用 `PhantomData`。`RefWithFlag` 和我们之前展示过的 `Ascii` 类型使用了相同的策略来避免未定义行为。这个

类型本身是 pub 的，但它的字段不是，这意味着只有在 `ref_with_flag` 模块内的代码可以创建或访问 `RefWithFlag` 值。你不需要检查太多代码就能确信 `ptr_and_bit` 字段始终是正确构造的。

22.8.3 可空的指针

Rust 中的空原始指针和 C/C++ 中一样，都是 0 地址。对任何类型 `T`, `std::ptr::null<T>` 函数会返回一个 `*const T` 空指针，`std::ptr::null_mut<T>` 返回一个 `*mut T` 空指针。

有一些方法可以检查一个原始指针是不是空的。最简单的是 `is_null` 方法，但 `as_ref` 方法可能会更加便捷：它接受一个 `*const T` 指针然后返回一个 `Option<&'a T>`，把空指针转换为 `None`。类似的，`as_mut` 方法把一个 `*mut T` 转换成 `Option<&'a mut T>`。

22.8.4 类型大小和对齐

一个 `Sized` 类型的值在内存中的字节数是固定的，并且必须放置在一个是对齐值倍数的地址处。例如，一个 `(i32, i32)` 元组占用 8 个字节，几乎所有的处理器都喜欢把它放在一个 4 的倍数的地址处。

`std::mem::size_of::<T>()` 返回 `T` 类型的字节数，`std::mem::align_of::<T>()` 返回它要求的对齐数。例如：

```
assert_eq!(std::mem::size_of::<i64>(), 8);
assert_eq!(std::mem::align_of::<(i32, i32)>(), 4);
```

任何类型的对齐总是 2 的幂。

一个类型的大小通常向上取它的对齐的倍数，即使从技术上讲它用不了那么多空间。例如，虽然元组 `(f32, u8)` 只需要 5 个字节，但 `size_of::<(f32, u8)>()` 是 8，因为 `align_of<(f32, u8)>()` 是 4。这确保了如果有一个数组，那么元素的 size 总是等于一个元素到下一个元素的距离。

对于 `unsized` 的类型，大小和对齐取决于具体的值。给定一个 `unsized` 值的引用，`std::mem::size_of_val` 和 `std::mem::align_of_val` 函数返回这个值的大小和对齐。这些函数可以同时用于 `Sized` 和 `unsized` 类型的值：

```
// 切片的胖指针会携带长度。
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;

// 这些函数返回 trait 对象指向的值的大小和对齐,
// 而不是 trait 对象本身。
// 这些信息来自于 trait 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 指针计算

Rust 把数组、切片或者 vector 布局为单个连续的内存块，如图 22-1 所示。元素都被按规律放置，这样如果每个元素占用 size 个字节，那么第 i 个元素从 $i * \text{size}$ 处开始。

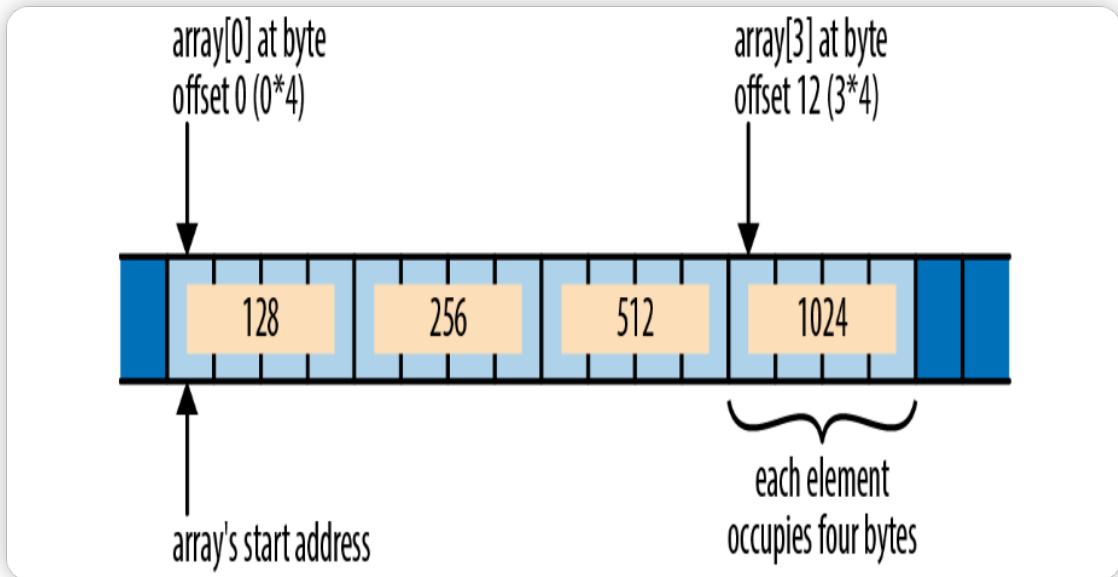


图 22-1: 内存中的数组

这样做的一个好处如果你有两个指向同一个数组中元素的原始指针，比较这两个指针和比较两个元素的索引的结果是一样的：如果 $i < j$ ，那么指向第 i 个元素的原始指针小于指向第 j 个元素的原始指针。这让原始指针作为数组遍历的边界时很有用。事实上，标准库中最简单的迭代切片的迭代器就是这样定义的：

```

struct Iter<'a, T> {
    ptr: *const T,
    end: *const T,
}

```

```
    ...  
}
```

`ptr` 字段指向下一个要产生的元素，`end` 字段作为边界：当 `ptr == end` 时，说明迭代结束了。

数组布局的另一个好处是：如果 `element_ptr` 是一个指向第 `i` 个元素的 `*const T` 或 `*mut T` 原始指针，那么 `element_ptr.offset(o)` 就是指向第 `i + o` 个元素的原始指针。它的定义等价于这样：

```
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>` 函数返回 `T` 类型的字节数。因为 `isize` 足够存储一个地址，因此可以把指针转换为 `isize`，然后对这个值进行算术运算，最后把结果转换为指针。

创建一个指向数组尾部之后第一个字节的指针是可以的。你不能解引用这样的指针，但它可以用于表示循环的结束或者边界检查。

然而，使用 `offset` 创建一个指向数组尾部之后或者头部之前的指针是未定义行为，即使你甚至没有解引用它。为了优化，Rust 会假设当 `i` 是正数时 `ptr.offset(i) > ptr`，当 `i` 是负数时 `ptr.offset(i) < ptr`。这个假设看起来是安全的，但如果 `offset` 中的算术运算溢出了 `isize` 值的范围它可能是不成立的。如果 `i` 被约束为保证最后的结果仍然和 `ptr` 指向同一个数组，那就不可能会溢出：数组自身不可能溢出地址空间的边界。（为了保证指向尾部后第一个字节的指针是安全的，Rust 绝不会把值放置在地址空间的最顶端。）

如果你的确需要把指针偏移到超出指向的数组，你可以使用安全的 `wrapping_offset` 方法。它等价于 `offset`，但 Rust 对 `ptr.wrapping_offset(i)` 和 `ptr` 的相对大小不做任何假设。当然，除非最后的结果还落在数组里，不然你仍然不能解引用它。

22.8.6 移进和移出内存

如果你在实现一个管理它自己的内存的类型，你将需要追踪内存里哪些部分存储了还在生命周期内的值，以及哪些部分是未初始化的，就像 Rust 处理局部变量一样。考虑这段代码：

```
let pot = "pasta".to_string();  
let plate = pot;
```

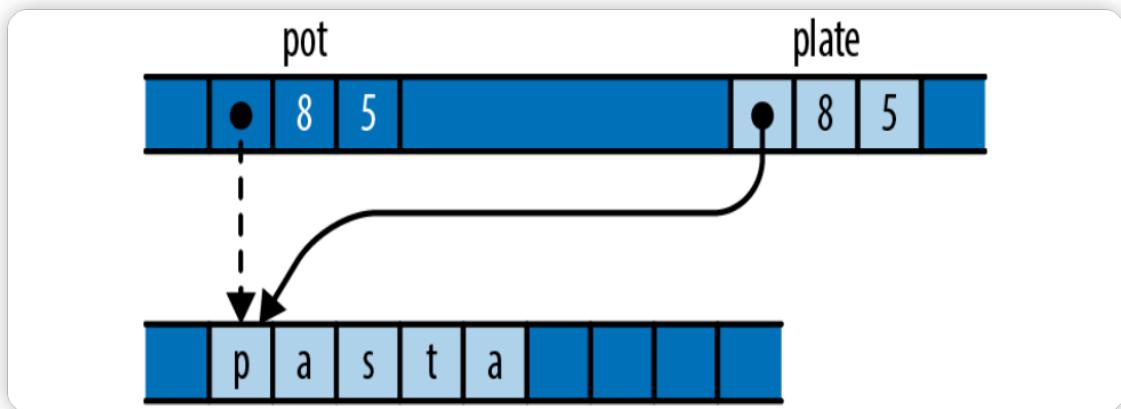


图 22-2: 把一个字符串从一个局部变量移动到另一个局部变量

当这段代码运行之后，内存布局看起来如图 22-2 所示。

在赋值之后，`pot` 是未初始化的，`plate` 拥有了字符串的所有权。

在机器语言的层面，一个 `move` 操作对应什么并不确定，但在实践中它通常什么也不做。这次赋值可能导致 `pot` 仍然持有字符串的指针、容量和长度。当然，如果还把它视为一个在生命周期内的值会带来灾难性的后果，Rust 确保你不会这样做。

同样的考虑也适用于那些管理自己内存的数据结构。假设你运行了这段代码：

```
let mut noodles = vec!["udon".to_string()];
let soba = "soba".to_string();
let last;
```

在内存中的状态可能如图 22-3 所示：

`vector` 还有空闲的空间来存储另一个元素，但这块空间现在的值是无效的，可能是任何之前存在这里的值。假设再运行下面的代码：

```
noodles.push(soba);
```

把字符串 `push` 进 `vector` 会把未初始化的内存转换为一个新的元素，如图 22-4 所示：

`vector` 初始化了空的空间，并且增大了长度来表示这是一个新的、还活着的元素。这个字符串的所有者现在变成了这个 `vector`；你可以通过 `vector` 的第二个元素引用这个字符串，并且 `drop` 这个 `vector` 会释放这两个字符串。并且 `soba` 现在变为未初始化。

最后，如果从 `vector` 中弹出一个值：

```
last = noodles.pop().unwrap();
```

在内存中，看起来像图 22-5。

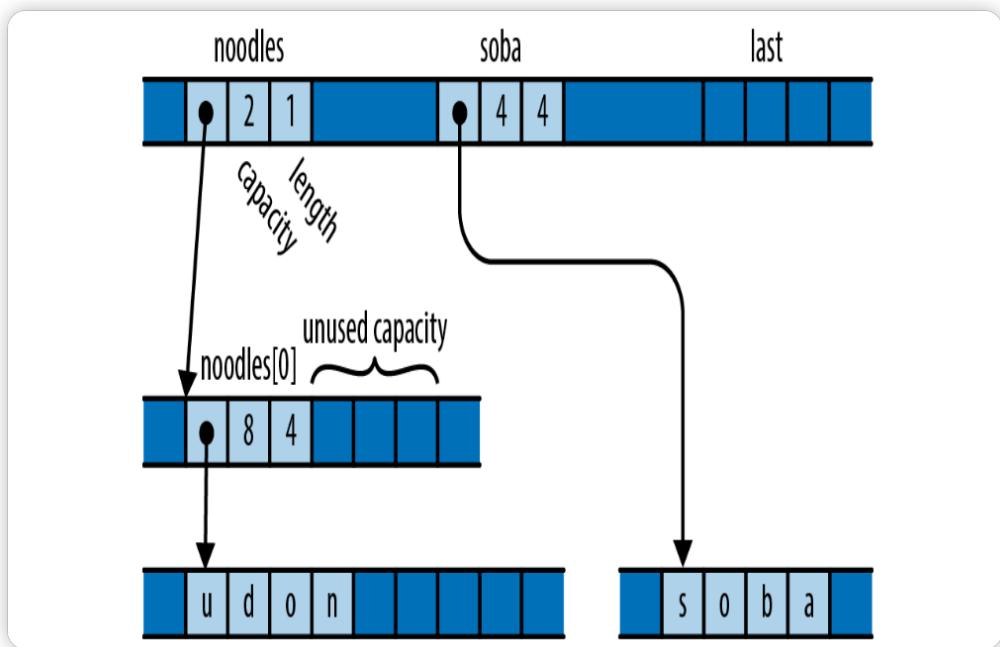


图 22-3: 一个有未初始化的空闲空间的 vector

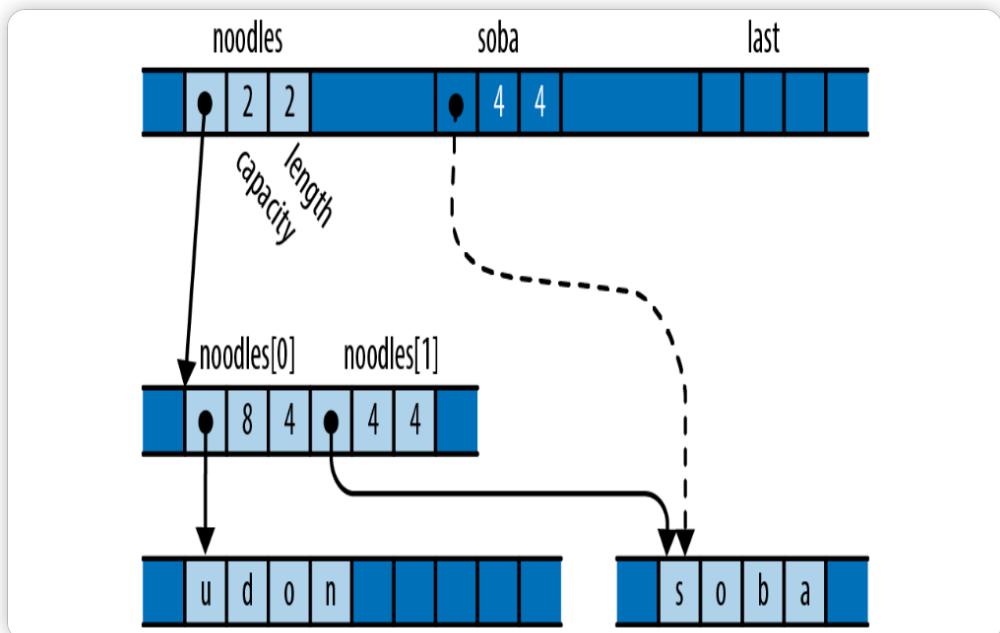


图 22-4: 把 soba 的值 push 进 vector 之后

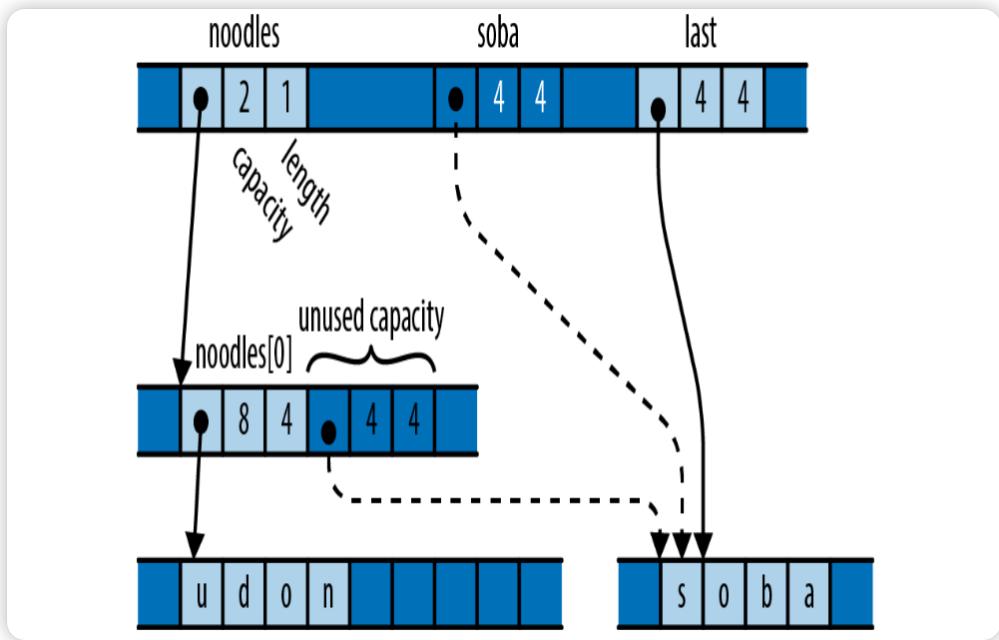


图 22-5: 从 vector 中弹出一个元素到 last 之后

变量 `last` 获得了字符串的所有权。vector 减小了长度来表示用来存储字符串的空间现在变为未初始化。

和之前的 `pot` 和 `pasta` 一样，这里 `soba`、`last` 和 vector 的空闲空间可能有完全相同的比特位。但只有 `last` 被认为拥有这个值，认为另外两个位置有活着的值显然是错误的。

初始化过的值的真正定义是被认为是活着 (*treated as live*) 的值。写入一个值的字节通常是初始化必要的步骤之一，但这只是因为这样做才能让值准备好被认为是活着的。`move` 和 `copy` 在内存中的效果是一样的，区别在于 `move` 之后，源对象不再被认为是活着的，而 `copy` 之后，源对象和目标对象都是活着的。

Rust 会在编译期追踪哪个局部变量是活着的，并阻止你使用那些值被移动走的变量。像 `Vec`、`HashMap`、`Box` 等类型都会动态追踪它们的缓冲区。如果你实现了一个自己管理内存的类，你需要做同样的事。

Rust 为实现这样的类型提供了两个基本的操作：

```
std::ptr::read(src)
```

从 `src` 指向的位置移出一个值，并把它的所有权交给调用者。`src` 参数应该是一个 `*const T` 原始指针，其中 `T` 是一个 `sized` 类型。在调用这个函数之后，`*src` 的内容不受影响，但除非 `T` 实现了 `Copy`，否则你必须保证你的程序不再把它视为活着的值。

这是 `Vec::pop` 背后的操作。`pop` 一个值会调用 `read` 来把这个值移出缓冲区，然后减小长度来把这个空间标记为未初始化。

```
std::ptr::write(dest, value)
```

把 `value` 移动到 `dest` 指向的位置，在调用之前 `dest` 指向的必须是未初始化的内存，调用之后指向的位置将拥有这个值。`dest` 必须是一个 `*mut T` 原始指针，`value` 必须是一个 `T` 值，`T` 是一个 `sized` 类型。

这是 `Vec::push` 背后的操作。`push` 一个值会调用 `write` 把值移动到下一个可用的位置，然后增大长度来表示这个位置现在是一个有效的元素。

它们都是自由函数，不是原始指针类型的方法。

注意你不能对 Rust 的 safe 指针类型做这些操作，它们在任何时候都要求引用对象的是初始化过的，因此把一块未初始化的内存变为一个值，或者反过来，都超出了它们的能力。只有原始指针才符合要求。

标准库还提供了几个函数用来从一个内存块中把数组的值移动到另一个内存块中：

```
std::ptr::copy(src, dst, count)
```

把内存中 `src` 位置开始的 `count` 个值移动到 `dst` 处，就好像你写了一个 `read` 和 `write` 调用的循环来一次一个地移动这些元素一样。目的位置在调用之前必须是未初始化的，调用之后源位置将是未初始化的。`src` 和 `dst` 参数必须是 `*const T` 和 `*mut T` 原始指针，`count` 必须是 `usize`。

```
ptr.copy_to(dst, count)
```

`copy` 的一个更便利的版本，将从 `ptr` 开始处的 `count` 个值移动到 `dst` 位置处，不需要起始点作为参数。

```
std::ptr::copy_nonoverlapping(src, dst, count)
```

类似于 `copy`，除了它的合约进一步要求源地址块和目的地址块不能重叠之外。它可能比 `copy` 快一点。

```
ptr.copy_to_nonoverlapping(dst, count)
```

`copy_nonoverlapping` 的一个更便利的版本，类似于 `copy_to`。

`read` 和 `write` 函数还有另外两个家族，也在 `std::ptr` 模块里：

```
read_unaligned, write_unaligned
```

这两个函数类似于 `read` 和 `write`，除了不像通常的引用类型一样要求指针是对齐的。这两个函数可能比普通的 `read` 和 `write` 函数慢。

`read_volatile`, `write_volatile`

这两个函数等价于 C 和 C++ 中的 `volatile` `read` 和 `write`。

22.8.7 示例：GapBuffer

这里有一个简单的例子用到了刚才介绍的函数。

假设你正在编写一个文本编辑器，并且正在寻找一个类型来表示文本。你可以选择 `String` 并使用 `insert` 和 `remove` 方法来在用户打字时插入或删除字符。但如果他们正在编辑一个大文件的起始位置，这些方法的开销就会很大：插入一个字符需要在内存中向右移动右侧所有的字符，删除一个字符需要向左移动右侧的所有字符。你可能希望这样常用的操作开销能更小。

Emacs 文本编辑器使用了一种叫做 *gap buffer* 的简单数据结构，它可以在常量时间内插入和删除字符。`String` 保持所有空闲空间都在文本的尾部，这样可以让 `push` 和 `pop` 操作的开销很小，而 `gap bufer` 把空闲空间保持在文本的中间，即正在编辑的地方。这样的空闲空间称为 *gap*。在 *gap* 处插入或删除元素的开销很小：只需要简单地缩小或者扩大 *gap*。你只需要把 *gap* 一侧的文本移动到另一侧就可以把 *gap* 移动到任何位置。当 *gap* 为空时，可以迁移到一个更大的缓冲区。

尽管在 *gap buffer* 中插入和删除操作速度都很快，但改变正在编辑的位置需要把 *gap* 移动到一个新的位置。移动元素所需的时间和要移动的距离成正比。幸运的是，典型的编辑活动包括在缓冲区的某一区域进行大量的修改，然后再去其他位置操作文本。

在这一节中我们将用 Rust 实现一个 *gap buffer*。为了避免被 UTF-8 干扰，我们用缓冲区直接存储 `char` 值，但即使以其他格式存储文本，操作的基本原则都是相同的。

首先，我们将用实践来展示一个 *gap buffer*。这段代码创建了一个 `GapBuffer`，在其中插入了一些文本，然后把插入位置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);
```

在运行过这段代码之后，缓冲区如图 22-6 所示。

插入操作会用新的文本填充 *gap*。这行代码添加了一个单词，破坏了现在的布局：

```
buf.insert_iter("Onion ".chars());
```

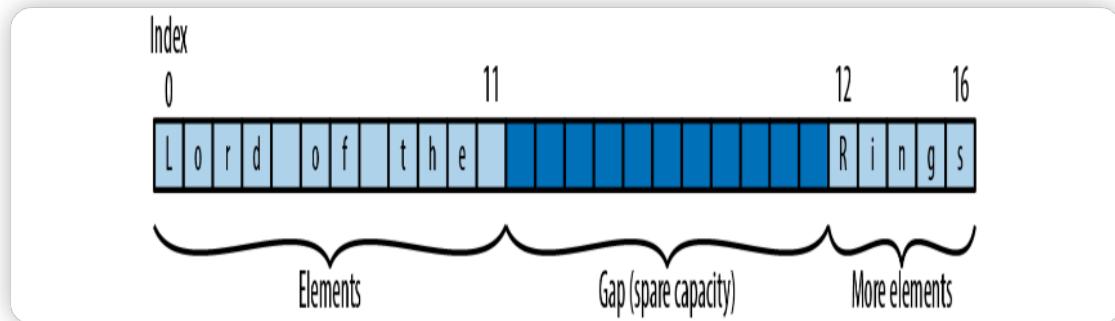


图 22-6: 一个包含一些文本的 gap buffer

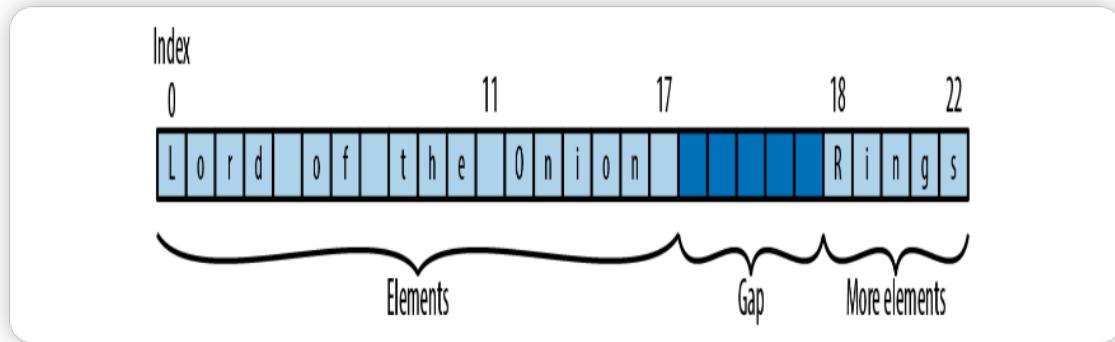


图 22-7: 一个包含更多文本的 gap buffer

结果如图 22-7 所示。

这里是我们的 GapBuffer 类型：

```
use std;
use std::ops::Range;

pub struct GapBuffer<T> {
    // 存储元素。它提供我们需要的容量，但它的长度总是 0。
    // GapBuffer 把它的元素和 gap 放在`Vec`“未使用”的空间里。
    storage: Vec<T>,

    // `storage` 中间未初始化的元素的范围。
    // 这个范围前面和后面的元素总是初始化过的。
    gap: Range<usize>
}
```

GapBuffer 以一种奇怪的方式使用它的 storage 字段²。它永远不会真的在 vector 中存储任何元素。它只是简单的调用 `Vec::with_capacity(n)` 来获得一个可以存储 `n` 个元素的足够大的内存块，通过 `vector` 的 `as_ptr` 和 `as_mut_ptr` 方法获得指向内存块的原始指针，然后直接以它自己的方式来使用这块内存作为缓冲区。`vector` 的长度将始终保持 0. 当 `Vec` 被 drop 时，`Vec` 不会尝试释放它的元素，因为它不知道它持有任何元素，但它确实会释放这个内存块。这正是 GapBuffer 想要的，它有它自己的 `Drop` 实现，这个实现知道有效的元素在哪里并可以正确地 drop 它们。

GapBuffer 的最简单的方法如你所料：

```
impl<T> GapBuffer<T> {
    pub fn new() -> GapBuffer<T> {
        GapBuffer { storage: Vec::new(), gap: 0..0 }
    }

    /// 返回这个 GapBuffer 在不重新分配的情况下可以存储的元素数量
    pub fn capacity(&self) -> usize {
        self.storage.capacity()
    }

    /// 返回这个 GapBuffer 当前持有的元素数量
    pub fn len(&self) -> usize {
        self.capacity() - self.gap.len()
    }
}
```

²还有一种方法可以更好地实现这些功能，这种方法需要使用编译器内部的 alloc crate 里的 `RawVec` 类型，不过这个 crate 仍然是 unstable 的。

```
    /// 返回当前的插入位置
    pub fn position(&self) -> usize {
        self.gap.start
    }
    ...
}
```

它需要很多下列的函数来实现一个方法，这个方法返回指向给定索引位置的元素的原始指针。既然是 Rust，那么我们需要一个返回 `mut` 指针的方法和一个返回 `const` 指针的方法。和上面的方法不同，它们并不是 `public` 的。继续这个 `impl` 块：

```
    /// 返回一个指向底层存储中第`index`个元素的指针。不考虑gap。
    ///
    /// 安全性：`index`必须是`self.storage`中有效的索引
    unsafe fn space(&self, index: usize) -> *const T {
        self.storage.as_ptr().offset(index as isize)
    }

    /// 返回一个指向底层存储中第`index`个元素的可变指针。不考虑gap。
    ///
    /// 安全性：`index`必须是`self.storage`中有效的索引
    unsafe fn space_mut(&mut self, index: usize) -> *mut T {
        self.storage.as_mut_ptr().offset(index as isize)
    }
```

为了查找给定索引处的元素，你必须考虑这个索引是落在 `gap` 之前还是之后，然后进行调整：

```
    /// 返回第`index`个元素的偏移量，考虑gap。
    /// 这个函数并不检查 index 是否在范围内，
    /// 但它永远不会返回一个在 gap 中的位置
    fn index_to_raw(&self, index: usize) -> usize {
        if index < self.gap.start {
            index
        } else {
            index + self.gap.len()
        }
    }

    /// 返回第`index`个元素的引用，
```



```
    self.gap = pos .. pos + gap.len();
}
}
```

这个函数使用了 `std::ptr::copy` 方法来移动元素；`copy` 要求目的位置必须是未初始化的，并且把源位置置为未初始化。源位置和目的位置可以重叠，`copy` 会正确地处理这种情况。因为 `gap` 在这个调用之前是未初始化的内存，并且这个函数会调整 `gap` 的位置来覆盖那些 `copy` 移动走的元素，因此 `copy` 函数的合约得到了满足。

元素的插入和删除相对简单。插入操作从 `gap` 中拿出一个元素的空间来存储新元素，而删除操作把值移出并增大 `gap` 来覆盖空出来的空间：

```
/// 在当前的插入位置插入`elt`,
/// 并把新的插入位置设置为`elt`之后。
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;
}
```

```
/// 在当前的插入位置插入`iter`产生的元素,
/// 并把新的插入位置设置为这些元素之后。
pub fn insert_iter<I>(&mut self, iterable: I)
    where I: IntoIterator<Item=T>
{
    for item in iterable {
        self.insert(item)
    }
}

/// 移除插入位置后的第一个元素并返回它,
/// 如果插入位置是在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)  
}
```

类似于 Vec 使用 `std::ptr::write` 来实现 push、使用 `std::ptr::read` 来实现 pop 一样， GapBuffer 使用 `write` 来实现 insert、使用 `read` 来实现 remove。并且就像 Vec 必须调整长度来维护已初始化元素和空闲空间的边界一样，GapBuffer 也会调整它的 gap。

当 gap 被填满时，`insert` 方法必须增大缓冲区来获取更多的空闲空间。`enlarge_gap` 方法（`impl` 块中的最后一个方法）用来实现这一点：

```
/// 把`self.storage`的容量翻倍
fn enlarge_gap(&mut self) {
    let mut new_capacity = self.capacity() * 2;
    if new_capacity == 0 {
        // 如果现有的vector是空的，就选择一个合适的起始容量。
        new_capacity = 4;
    }
}

// 我们不知道resize一个Vec会对它“未使用的空间进行什么操作。
// 因此简单地创建一个新的vector并把元素移动过去。
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 {
    // 移动落在gap之前的元素。
    std::ptr::copy_nonoverlapping(self.space(0),
                                  new.as_mut_ptr(),
                                  self.gap.start);

    // 移动落在gap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drop任何元素，  
// 因为Vec的长度是0。  
self.storage = new;  
self.gap = new_gap;  
}
```

和 `set_position` 必须使用 `copy` 来移动元素不同，`enlarge_gap` 可以使用 `copy_nonoverlapping`，因为它是把元素移动到一个完全全新的缓冲区。

把新的vector赋值给 `self.storage` 会 drop 旧的vector。但因为旧vector的长度是0，所以它会相信它没有要 drop 的元素，然后简单地释放缓冲区。`copy_nonoverlapping` 会让源地址变为未初始化的，所以旧vector的假设确实是正确的：现在所有的元素的所有权都在新vector里。

最后，我们需要确保 drop 一个 `GapBuffer` 会释放所有元素：

```
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));  
            }  
        }  
    }  
}
```

元素分布在 gap 之前和之后，因此我们迭代这两个区域并使用 `std::ptr::drop_in_place` 函数来 drop 每个元素。`drop_in_place` 函数是一个工具函数，它的行为类似于 `drop(std::ptr::read(ptr))`，但不需要把值 move 给调用者（因此也可以用于 unsized 类型）。在 `enlarge_gap` 中，当 vector `self.storage` 被 drop 时，它的缓冲区实际是未初始化的。

类似我们在本章中展示过的其他类型一样，`GapBuffer` 确保它自己的不变量足够充分，以此来确保它用到的每个 unsafe 特性的合约都被遵守，因此它的所有 public 方法都不需要标记为 unsafe。`GapBuffer` 为一个在 safe 代码中不可能高效实现的特性实现了一个 safe 的接口。

22.8.8 unsafe 代码中的 panic 安全性

在 Rust 中，panic 通常不会导致未定义行为；`panic!` 宏并不是一个 unsafe 特性。但当你决定在 unsafe 代码中使用它时，你就需要考虑 panic 安全性了。

考虑上一节定义的 `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)
}
```

`read` 的调用会把元素立刻移出 `gap` 并留下未初始化的空间。这时 `GapBuffer` 处于一种不一致的状态：我们打破了所有 `gap` 之外的元素都必须是初始化过的这一不变量。幸运的是，下一条语句增大了 `gap` 让它覆盖了未初始化的空间，因此在我们返回时，不变量仍然成立。

但考虑如果在调用 `read` 之后、给 `self.gap.end` 赋值之前尝试使用一些可能 panic 的特性（例如索引元素）会发生什么。在这两个动作之间中断这个方法将会导致 `GapBuffer` 有一个未初始化的元素处于 `gap` 之外。下一次调用 `remove` 时将会再次尝试 `read` 这个元素；即使简单地 drop `GapBuffer` 也会尝试再次 drop 这个元素。这两种情况都是未定义行为，因为它们访问了未初始化的内存。

如果一个类型在方法里临时打破这个类型的不变量，然后再在返回之前恢复不变量，那么这个问题是不可避免的。在方法中途 panic 可能会缩短清理过程，让类型处于不一致的状态。

如果这个类型只使用 safe 代码，那么这种不一致可能会导致类型的行为出错，但不会导致未定义行为。但是用了 unsafe 特性的代码通常依赖它的不变量来满足那些特性的合约。被破坏的不变量会破坏合约，进而导致未定义行为。

当使用 unsafe 特性时，你必须特别注意那些临时打破不变量的区域，并保证它们不会在这些地方 panic。

22.9 使用 union 重新解释内存

Rust 提供了很多有用的抽象，但你编写的代码最终都是在操作字节。Union 是 Rust 最强大的特性之一，它可以操作字节并选择如何解释它们。例如，所有 32 位——4 个字节的类型都可以被解释为一个整数或者一个浮点数。每一种解释都是有效的，尽管这种解释的结果可能是无意义的。

一个 union 代表一些字节的集合，如果像下面这么写，就意味着这些字节可以被解释为一个整数或者一个浮点数：

```
union FloatOrInt {
    f: f32,
    i: i32,
}
```

这是一个有两个字段 `f` 和 `i` 的 union。它们可以像 struct 一样给字段赋值，但和 struct 不同的是当构建一个 union 时，你只能选择其中一个字段初始化。struct 的每个字段指向不同的内存位置，但 union 的不同字段是对同一块字节序列的不同解释。给一个不同的字段赋值意味着覆盖部分或全部的字节。这里，`one` 指向一个 32 位的内存区域，这块区域首先存储了整数 1，然后存储了 IEEE 754 浮点数 1.0。给 `f` 赋值后，之前给 `FloatOrInt` 赋的值立刻被覆盖：

```
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，union 所占的字节数取决于最大的字段。例如，这个 union 占用 64 个比特，尽管 `SmallOrLarge::s` 只是一个 `bool`：

```
union SmallOrLarge {
    s: bool,
    l: u64
}
```

尽管构建一个 union 和向它的字段赋值完全是 safe 的，但读取它的任何字段都是 unsafe 的：

```
let u = SmallOrLarge { l: 1337 };
println!("{}", unsafe { u.l }); // 打印出 1337
```

这是因为和 enum 不同，union 并没有标记。编译器并没有添加额外的位来记录当前的状态。没有办法在运行期判断一个 `SmallOrLarge` 应该被解释为 `u64` 还是 `bool`，除非程序有一些额外的上下文。

给定字段的位模式是否有效也没有任何内建的保证。例如，写入 `SmallOrLarge` 的 `l` 字段会覆盖它的 `s` 字段，这时候的位模式可能并不是有效的 `bool` 值。因此，尽管写入 union 的字段是 safe 的，但每一次读取都需要 unsafe。只有当 `s` 字段的位能构成一个有效的 `bool` 时才允许读取 `u.s`，否则就是未定义行为。

在遵守这些限制的前提下，union 可以作为一种有效的方式来临时解释一些数据，尤其是要对值的位表示进行操作而不是对值本身进行操作的时候。例如，之前提到的 `FloatOrInt` 类型可以用于简单地打印出一个浮点数的每一位，即使 `f32` 并没有实现 `Binary` 格式化器：

```
let float = FloatOrInt { f: 31337.0 };
// 打印出 100011011110100110100100000000
println!("{:b}", unsafe { float.i });
```

尽管这些简单的例子应该在任何版本的编译器上如预期工作，但不能保证每个字段都从同一个地址开始，除非在 union 的定义处添加一个属性告诉编译器如何在内存中摆放数据。添加属性 #[repr(C)] 可以保证所有字段都从 0 偏移处开始，而不是让编译器决定。有了这个保证，覆盖值的行为可以用来提取某些位，例如一个整数的符号位：

```
#[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);
```

这里，符号位是最高有效字节的最高有效位。因为 x86 处理器是小端序，因此这些字节的顺序是反的；最高有效字节不是 bytes[0]，而是 bytes[7]。通常情况下，Rust 代码不需要考虑这些，但因为这段代码直接操作 i64 的内存表示，所以这些底层的细节就变得很重要了。

因为 union 不能识别如何 drop 它们的内容，因此它们的所有字段都必须是 Copy。然而，如果你必须在 union 中存储一个 String，还是有一个解决方法：可以参考标准库中 std::mem::ManuallyDrop 的文档。

22.9.1 match union

match 一个 Rust 的 union 类似于 match 一个 struct，除了每个模式都必须精确地指定一个字段之外：

```
unsafe {
    match u {
        SmallOrLarge { s: true } => { println!("boolean true"); }
```

```
SmallOrLarge { l: 2 } => { println!("integer 2"); }
_ => { println!("something else"); }
}
}
```

指定一个 union 字段但不指定值的 `match` 分支总是能成功匹配。如果 `u` 最后一个被写入的字段是 `u.i`, 下面的代码将会导致未定义行为:

```
// 未定义行为!
unsafe {
    match u {
        FloatOrInt { f } => { println!("float {}", f) },
        // 警告: 不可能到达的模式
        FloatOrInt { i } => { println!("int {}", i) }
    }
}
```

22.10 借用 union 的引用

借用 union 的一个字段的引用会借用整个 union。这意味着根据通常的借用规则, 以可变的方式借用一个字段会禁止对其他任何字段的借用, 以共享的方式借用一个字段导致不能再以可变的方式借用其他字段的引用。

正如我们将在下章介绍的, 除了你自己的 unsafe 代码之外, Rust 还能帮助其他语言编写的代码构建 safe 的接口。unsafe 正如这个名字隐含的一样令人烦恼, 但如果小心使用它, 它可以帮助你构建高性能的代码, 并同时保留 Rust 程序员喜欢的安全性保证。

Chapter 23

外部函数

Cyberspace. Unthinkable complexity. Lines of light ranged in the non-space of the mind, clusters and constellations of data. Like city lights, receding . . .

——William Gibson, *Neuromancer*

不幸的是，世界上不是每个程序都是用 Rust 编写的。我们可能想在我们的 Rust 程序中使用很多用其他语言实现的优秀的库和接口。Rust 的外部语言接口 (*foreign function interface(FFI)*) 让 Rust 代码能调用 C 编写的函数和一部分 C++ 编写的函数。因为大多数操作系统提供 C 接口，Rust 的外部函数接口允许直接访问任何类型的底层设施。

在本章中，我们将编写一个链接到 libgit2 的程序，它是一个用于 Git 版本控制系统的 C 库。首先，我们将展示怎么直接在 Rust 中使用 C 函数，就使用我们上一章中介绍的 unsafe 特性。然后，我们将展示如何构建 libgit2 的 safe 接口，借助开源的 git2-rs crate 的灵感，这个 crate 正好实现了这一点。

我们将假设你熟悉 C 语言和编译链接 C 程序的机制。处理 C++ 也相差不多。我们还假设你熟悉 Git 版本控制系统。

确实有 Rust crate 用于和很多其他语言例如 Python、JavaScript、Lua 和 Java 交互。我们没有足够的篇幅来介绍它们，但所有这些接口最终都是使用 C 外部函数接口实现的，因此无论你想和什么语言交互，这一章都能给你一个开头。

23.1 寻找公共的数据表示

Rust 和 C 的公共基础是机器语言，因此为了预测 Rust 的值在 C 代码中看起来是什么样的，或者反过来，你需要考虑它们的机器级表示。在整本书中，我们一直着重展示一个值在内存中的实际表示，因此你可能已经注意到了 C 和 Rust 的数据世界有很多共通之处：

例如 Rust 的 `usize` 和 C 的 `size_t` 是相同的，两门语言中的结构体也基本相同。为了建立起 Rust 和 C 中相应类型的关系，我们将从基本类型开始，并逐渐扩展到更复杂的类型。

鉴于 C 主要用作系统编程，C 中类型的表示总是令人惊讶的宽松：例如一个 `int` 通常是 32 位，但可能会更长，或者短到 16 位；一个 C 的 `char` 可能是有符号的也可能是无符号的。为了应对这种可变性，Rust 的 `std::os::raw` 模块定义了一些 Rust 的类型，这些类型保证和相应的 C 类型有完全相同的表示（表 23-1）。其中包括基本的整数和字符类型。

表 23-1: `std::os::raw` 中的 Rust 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>
<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 *, const void *</code>	<code>*mut c_void, *const c_void</code>

有关表 23-1 的注意事项：

- 除了 `c_void` 之外，这里所有的 Rust 类型都是某些基本 Rust 类型的别名：例如 `c_char` 是 `i8` 或者 `u8`。
- Rust 的 `bool` 等价于 C 或 C++ 的 `bool`。
- Rust 的 32 位 `char` 类型并不等同于 `wchar_t`，后者的宽度和编码取决于具体实现。C 的 `char32_t` 倒是更接近一点，但它的编码仍然并不保证是 Unicode。
- Rust 的基础 `usize` 和 `isize` 类型和 C 的 `size_t` 和 `ptrdiff_t` 有相同的表示。
- C/C++ 指针和 C++ 的引用对应 Rust 的原始指针类型 `*mut T` 和 `*const T`。
- 从技术上讲，C 标准允许实现使用一些 Rust 中没有相应类型的表示：36 位整数、用符号 + 数字来表示有符号值等等。在实践中，在 Rust 被移植到的每个平台上，每个基本的 C 整数类型在 Rust 中都有对应的类型。

为了定义兼容 C 结构体的 Rust 结构体类型，你可以使用 `#[repr(C)]` 属性。在结构体的定义上面放上 `#[repr(C)]` 可以要求 Rust 按照 C 编译器的方式放置结构体中的字段。例如，`libgit2` 的 `git2/errors.h` 头文件定义了下面的 C 结构体来提供一个错误的详情：

```
typedef struct {
    char *message;
    int klass;
} git_error;
```

你可以按照下面这样定义一个内存表示完全相同的 Rust 类型：

```
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)]` 属性只影响 `struct` 自身的布局，不会影响单个字段的表示，因此为了和 C `struct` 匹配，每一个字段也都要使用 C 风格的类型：例如用 `*const c_char` 替换 `char *`，用 `c_int` 替换 `int`。

在这个特定的例子中，`#[repr(C)]` 属性可能并不会改变 `git_error` 的布局。因为实际上没有那么多放置一个指针和一个整数的方法。但 C 和 C++ 都保证一个结构体的成员按照声明的顺序依次在内存中排布，而 Rust 会按照总结构体大小最小的方式来组织字段，并且 0 大小的类型不占用空间。`#[repr(C)]` 属性告诉 Rust 按照 C 的规则来布局。

你也可以使用 `#[repr(C)]` 来控制 C 风格的 `enum` 的表示：

```
#[repr(C)]
#[allow(non_camel_case_types)]
enum git_error_code {
    GIT_OK          = 0,
    GIT_ERROR       = -1,
    GIT_ENOTFOUND   = -3,
    GIT_EEXISTS     = -4,
    ...
}
```

通常情况下，Rust 在选择如何表示 `enum` 时会使用各种技巧。例如，我们提到过 Rust 在一个单字中存储 `Option<&T>`（如果 `T` 是 `sized`）。如果没有 `#[repr(C)]`，Rust 会使用单个字节来表示 `git_error_code enum`；有了 `#[repr(C)]` 之后，Rust 会和 C 一样用一个 C `int` 一样大的值来存储。

你可以要求 Rust 使用和某些整数相同的表示来存储 enum。上面的定义如果以 `#[repr(i16)]` 开头，将会得到一个和下面 C++ enum 相同的 16 位的表示：

```
#include <stdint.h>

enum git_error_code: int16_t {
    GIT_OK          = 0,
    GIT_ERROR       = -1,
    GIT_ENOTFOUND   = -3,
    GIT_EEXISTS     = -4,
    ...
};
```

正如之前提到的，`#[repr(C)]` 还可以用于 union。`#[repr(C)]` union 的字段总是从 union 的内存的第一个位（0 偏移处）开始。

假设你有一个 C struct 使用一个 union 来存储一些数据，再用一个 tag 值来指示应该使用 union 的哪个值，类似于 Rust 的 enum 一样：

```
enum tag {
    FLOAT = 0,
    INT   = 1,
};

union number {
    float f;
    short i;
};

struct tagged_number {
    tag t;
    number n;
};
```

Rust 可以通过对 enum、struct 和 union 类型都应用 `#[repr(C)]` 来实现一个这样的结构体，然后使用 `match` 语句基于 tag 来选择一个 struct 中的 union 的字段：

```
#[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
        }
    }
}
```

使用这种技术，即使是复杂的结构体也可以很容易地跨 FFI 边界使用。

在 Rust 和 C 之间传递字符串要稍微更难一点。C 使用一个空结尾的字符数组的指针来表示字符串。另一边 Rust 显式地存储字符串的长度，要么是 `String` 的一个字段，要么是一个胖引用 `&str` 的第二个字。Rust 的字符串不是空字符结尾的；事实上，它们的内容里可能包含空字符，这些空字符和其他字符一样，没有区别。

这意味着你不能借用一个 Rust 字符串来当做 C 字符串：如果你传给 C 代码一个 Rust 字符串的指针，它可能错误地把内容中一个空字符当做字符串的结尾，或者越界去查找一个不存在的结尾空字符。从另一个方向考虑，你也许可以借用一个 C 字符串作为一个 Rust 的 `&str`，只要它的内容是有效的 UTF-8。

这种情况强迫 Rust 必须把 C 的字符串看做和 `String`、`&str` 完全不同的类型。在 `std::ffi` 模块中，`CString` 和 `CStr` 类型表示拥有所有权的和借用的空字符结尾的字节数组。和 `String`、`str` 比起来，`CString` 和 `CStr` 的方法非常有限，基本只有构建自身和转换成其他类型的方法。我们将在下一节中用实例展示这些类型。

23.2 声明外部函数和变量

用 `extern` 块来声明其他库中定义的函数和变量，最终的 Rust 编译出的可执行文件会链接到这些库。例如，在大多数平台上，每个 Rust 程序都会链接到标准的 C 库，因此我们可以像这样告诉 Rust C 库里的 `strlen` 函数：

```
use std::os::raw::c_char;

extern {
    fn strlen(s: *const c_char) -> usize;
}
```

这样就告诉了 Rust 这个函数的名字和类型，定义将会在之后进行链接。

Rust 假设在 `extern` 块中声明的函数使用 C 语言的惯例来传递参数和接受返回值。它们被定义为 `unsafe` 函数。对 `strlen` 来说，这是正确的选择：它实际上是一个 C 函数，它的 C 规范要求你传入一个有效的以空字符结尾的字符串指针，而这是一个 Rust 无法强制的合约。（几乎任何接受原始指针作为参数的函数都必须是 `unsafe` 的：safe Rust 可以从任何整数构造原始指针，但解引用这样的指针可能会导致未定义行为。）

有了这个 `extern` 块，我们可以像任何其他 Rust 函数一样调用 `strlen`，尽管它的类型显示了它是别的语言中定义的函数：

```
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` 函数构建出一个空字符结尾的 C 字符串。它首先检查它的内容里是否有空字符，因为如果有空字符就不能用 C 字符串来表示它，如果找到了空字符就返回一个 `error`（因此需要调用 `unwrap`）。否则，它向结尾添加一个空字符并返回一个 `CString`，这个 `CString` 持有最后的字符的所有权。

`CString::new` 的开销取决于参数的类型。它接受任何实现了 `Into<Vec<u8>>` 的类型，传递一个 `&str` 需要一次内存分配和拷贝，因为它到 `Vec<u8>` 的转换需要在堆上构建一份字符串的拷贝，这样 `vector` 才能拥有字符串的所有权。但以值传递一个 `String` 简单地消耗掉这个字符串并获取它的缓冲区的所有权，因此除非向缓冲区中添加一个空字符需要扩大缓冲区，否则这个转换完全不需要拷贝文本或内存分配。

CString 解引用到 CStr，后者的 as_ptr 方法返回一个`*const c_char` 指向字符串的头部。这正是 `strlen` 期望的类型。在这个例子中，`strlen` 遍历字符串，寻找 `CString::new` 添加的空字符，然后返回字符串的长度，即字节数。

你也可以在 `extern` 块中声明全局变量，POSIX 系统有一个叫做 `environ` 的全局变量存储进程的环境变量。在 C 中，它被声明为：

```
extern char **environ;
```

在 Rust 中，你可以写：

```
use std::ffi::CStr;
use std::os::raw::c_char;

extern {
    static environ: *mut *mut c_char;
}
```

为了打印出环境变量的第一个元素，你可以写：

```
unsafe {
    if !environ.is_null() && !(*environ).is_null() {
        let var = CStr::from_ptr(*environ);
        println!("first environment variable: {}",
            var.to_string_lossy())
    }
}
```

在确保了 `environ` 有第一个元素之后，代码调用了 `CStr::from_ptr` 来构建一个借用它的 `CStr`。`to_string_lossy` 方法返回一个 `Cow<str>`：如果 C 字符串包含有效的 UTF-8，`Cow` 会以 `&str` 的形式借用它的内容，不包括结尾的空字符。否则，`to_string_lossy` 在堆上构造一份文本的拷贝，把其中非 UTF-8 的字符序列替换为官方的 Unicode 替换字符，并构造出一个拥有它所有权的 `Cow`。无论是哪种情况，结果都实现了 `Display`，因此你可以使用 `{}` 格式化参数打印出它。

23.3 使用库里的函数

为了使用一个特定的库提供的函数，你可以在 `extern` 块上方加上 `#[link]` 属性来指定 Rust 应该链接的库的名称。例如，这里有一个程序调用 `libgit2` 的初始化和结束函数，但不做任何其他事：

```

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` 和之前一样声明了外部函数。`#[link(name = "git2")]` 属性会要求当 Rust 创建最终的可执行文件或者共享库时，它应该链接到 `git2` 库。Rust 使用系统的链接器来构建可执行文件：在 Unix 上，它会向链接器的命令行传递 `-lgit2` 参数；在 Windows 上，它会传递 `git2.LIB` 参数。

`#[link]` 属性在库 crate 中也可以工作。当你构建一个依赖其他 crate 的程序时，Cargo 会从整个依赖图中抓取所有链接项并在最后的链接中全部加进去。

如果你想要在你自己的机器上继续这个例子，你需要自己构建 `libgit2`。我们使用 `libgit2` 0.25.1 版本。为了编译 `libgit2`，你需要安装 CMake 构建工具和 Python；我们使用了 CMake 3.8.0 版本和 Python 2.7.13 版本。

构建 `libgit2` 的完整文档可以在网站上找到，但因为非常简单所以我们将在这里展示基本的一些步骤。在 Linux 上，假设你已经把库的源码解压到了目录 `/home/jimb/libgit2-0.25.1`：

```

$ 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`，还有一些指向它的符号链接，包括有一个叫 `libgit2.so` 的。在 macOS 上，结果与此类似，但库的名字叫 `libgit2.dylib`。

在 Windows 上也非常简单。假设你把源码解压到了目录 `C:\Users\JimB\libgit2-0.25.1`。在一个 Visual Studio 的命令提示符中：

```
> cd C:\Users\JimB\libgit2-0.25.1
> mkdir build
> cd build
> cmake -A x64 ..
> cmake --build .
```

这些命令和在Linux上用的命令几乎一样，除了在第一次运行CMake的时候必须指定64位的构建来匹配你的Rust编译器。（如果你安装了32位的Rust工具链，那么你应该省略第一条cmake命令的-A x64标记。）这会产生一个导入库git2.LIB和一个动态链接库git2.DLL，都在目录C:\Users\JimB\libgit2-0.25.1\build\Debug。（之后的命令都是以Unix为例，如果Windows上的命令大不相同的话会再加上Windows上的命令。）

在一个单独的目录里创建Rust程序：

```
$ cd /home/jimb
$ cargo new --bin git-toy
Created binary (application) `git-toy` package
```

复制我们之前展示的代码并粘贴到src/main.rs里。当然，如果你尝试构建，Rust会不知道在哪查找你构建的libgit2库：

```
$ cd git-toy
$ cargo run
Compiling git-toy v0.1.0 (/home/jimb/git-toy)
error: linking with `cc` failed: exit code: 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: aborting due to previous error
error: could not compile `git-toy`
To learn more, run the command again with --verbose.
```

你可以通过编写一个构建脚本来告诉Rust在哪搜索这个库，构建脚本是Cargo会在编译期编译并运行的Rust代码。构建脚本可以做很多事：动态生成代码，编译要被包含在crate中的C代码等等。在这个例子中，你需要做的只是向可执行文件的链接命令中添加一个库的搜索路径。当Cargo运行构建脚本时，它会解析构建脚本的输出来获取这类信息，因此构建脚本只需要把正确的信息打印到标准输出就可以了。

为了创建你自己的构建脚本，在Cargo.toml所在的目录下添加一个叫build.rs的文件，内容如下：

```
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，你应该把 `native=` 之后的路径替换为 `C:\Users\JimB\libgit2-0.25.1\build\Debug`。（我们忽略了一些细节来保证这个例子的简洁；在一个真实的应用中，你应该避免在构建脚本中使用绝对路径。我们在本节结尾给出了如何正确实现这一点的文档。）

现在你基本上可以运行这个程序了。在 macOS 上它可能可以立即工作；但在 Linux 系统上你可能会看到类似这样的输出：

```
$ 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
```

意思是说，尽管 Cargo 成功地把可执行文件链接到了库，但它不知道怎么在运行时找到共享库。在 Windows 上通过弹出一个对话框来报告这个错误。在 Linux 上，你必须设置 `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，你可能需要设置 `DYLD_LIBRARY_PATH`。

在 Windows 上，你必须设置 `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，在一个要部署的应用中你可能想避免只为了寻找你的库而修改环境变量。一种替代方案是把 C 库静态链接到你的 crate 里。这会把库的目标文件拷贝到 crate 的 `.rlib` 文件里，和这个 crate 中的 Rust 代码生成的目标文件和元数据放在一起。然后这整个集合会参与最终的链接。

Cargo 的一个惯例是提供 C 库访问的 crate 应该命名为 LIB-sys，其中 LIB 是 C 库的名称。一个-sys crate 应该只包含静态链接的库和包含 `extern` 块和类型定义的 Rust 模块。更高层的接口应该在依赖-sys crate 的 crate 中实现。这允许多个上游的 crate 依赖同一个-sys crate，假设有一个版本的-sys crate 可以满足每个上游的需要的话。

有关 Cargo 的构建脚本和与系统库的链接的完整文档见[在线 Cargo 文档](#)。它展示了如何在构建脚本中避免绝对路径、控制编译选项、使用例如 `pkg-config` 的工具，等等。`git2-rs` crate 也提供了模仿的好例子，它的构建脚本处理了一些很复杂的情况。

23.4 一个 libgit2 的原始接口

如何正确地使用 `libgit2` 可以分解为两个问题：

1. 如何在 Rust 中使用 `libgit2` 的函数？
2. 如何通过它们构建一个 safe Rust 的接口？

我们将逐一解决这两个问题。在本节中，我们将编写一个程序，它基本上是一个巨大的 `unsafe` 块，充满了不规范的 Rust 代码，它反映了类型系统和惯例的冲突，这也是混合语言编程所固有的特点。我们称其为原始 (*raw*) 接口。它的代码可能有些凌乱，但它能展示出 Rust 代码使用 `libgit2` 时必须的所有步骤。

然后，在下一节中，我们将构建一个 `libgit2` 的 safe 接口，它使用 Rust 的类型来强迫 `libgit2` 对使用者隐含的要求。幸运的是，`libgit2` 是一个设计得非常好的 C 库，因此 Rust 的安全性要求的问题都有很好的答案，并且我们可以构建出没有 `unsafe` 函数的规范的 Rust 接口。

我们要写的程序非常简单：它以命令行参数的形式接受一个路径，打开那里的 Git 仓库，然后打印出 head commit 的信息。但这足以用来展示构建 safe 和规范的 Rust 接口的关键策略。

对于原始接口，程序需要很多 `libgit2` 中的函数和类型，因此把 `extern` 块移动到它自己的模块是有意义的。我们将在 `git-toy/src` 中创建一个叫 `raw.rs` 的文件，内容如下：

```
#![allow(non_camel_case_types)]\n\nuse std::os::raw::{c_int, c_char, c_uchar};\n\n#[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
}
```

其中每一项都是根据 libgit2 的头文件来声明的。例如 *libgit2-0.25.1/include/git2/repository.h* 中包含这个声明：

```
extern int git_repository_open(git_repository **out, const char *path);
```

这个函数尝试打开 *path* 处的 Git 仓库。如果一切顺利，它会创建一个 *git_repository* 对象并把该对象的指针存储在 *out* 指向的位置。等价的 Rust 声明如下：

```
pub fn git_repository_open(out: *mut *mut git_repository,
                           path: *const c_char) -> c_int;
```

libgit2 public 的头文件使用 *typedef* 定义了一个不完全的 *git_repository* 类型：

```
typedef struct git_repository git_repository;
```

因为这个类型的详情是 *private* 的，因此 public 的头文件中并没有定义 *struct git_repository*，以确保库的使用者永远不能自己构造这个类型的实例。在 Rust 中一种可行的定义不完全的 *struct* 类型的方法是：

```
#[repr(C)] pub struct git_repository { _private: [u8; 0] }
```

这个 *struct* 类型包含一个没有元素的数组。因为 *_private* 字段不是 *pub* 的，所以不能在这个模块之外构造这个类型的值，正好对应了 libgit2 中不应该手动构造的类型。这种类型只能通过原始指针来进行操作。

手动编写一个巨大的 *extern* 块可能会很繁琐。如果你正在编写一个复杂的 C 库的 Rust 接口，你可能会想尝试 *bindgen* crate，它包含一些可以在构建脚本中使用的函数，这些函数可以解析 C 头文件并自动生成相应的 Rust 声明。这里我们没有篇幅去介绍 *bindgen*，但 crates.io 上的 *bindgen* 的页面包含了它的文档的链接。

接下来我们将完全重写 *main.rs*。首先，我们需要声明 *raw* 模块：

```
mod raw;
```

根据 libgit2 的惯例，可能失败的函数会返回一个整数，成功时整数为正数或者 0，失败时整数为负数。如果有错误发生，*giterr_last* 函数将会返回一个 *git_error* 结构体的指

针，这个结构体会提供更多有关错误的细节。libgit2 持有这个结构体，所以我们不需要自己 free 它，但它可能被后续的调用覆盖。一个恰当的 Rust 接口应该使用 Result，但是在原始的版本中，我们将按照 libgit2 的方式使用它的函数，因此我们将编写我们自己的函数来处理错误：

```
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
}
```

我们将使用这个函数检查 libgit2 调用的结果：

```
check("initializing library", raw::git_libgit2_init());
```

这里使用了之前用过的 CStr 的方法：使用 from_ptr 来从 C 字符串构造一个 CStr，使用 to_string_lossy 来把它转换成 Rust 可以打印的类型。

接下来，我们需要一个函数来打印出一次 commit：

```
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!("{}\n", CStr::from_ptr(message).to_string_lossy());
}
```

给定一个 git_commit 的指针，show_commit 会调用 git_commit_author 和 git_commit_message 来获取它需要的信息。这两个函数遵循了 libgit2 在文档中解释的一个惯例：

如果一个函数返回一个对象作为返回值，那个这个函数是一个 *getter*，并且对象的生命周期被绑定到父对象上。

用Rust的术语来说就是，`author`和`message`是从`commit`的借用，所以`show_commit`不需要自己释放它们，但它绝对不能在`commit`被释放之后仍然持有它们。因为这个API使用了原始指针，所以Rust不会为我们检查它们的生命周期：如果我们意外地创建了悬垂指针，可能直到程序崩溃时我们才会发现。

上面的代码假设这些字段都持有UTF-8文本，但这个假设并不总是正确的。Git也允许其他的编码。正确地解析这些字符串可能需要使用`encoding crate`。为了保持简洁，我们在这里忽略这些问题。

我们程序的`main`函数如下：

```
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);
}

}

```

这段代码首先处理 path 参数并初始化库，都是我们之前见过的内容。第一行有趣的代码是：

```

let mut repo = ptr::null_mut();
check("opening repository",
      raw::git_repository_open(&mut repo, path.as_ptr()));

```

对 `git_repository_open` 的调用会尝试在打开指定路径的 Git 仓库。如果它成功了，它会分配一个新的 `git_repository` 对象并设置 `repo` 指向它。Rust 隐式把引用转换成原始指针，因此这里传递的 `&mut repo` 提供了调用所需的 `*mut *mut git_repository` 参数。

这里展示了另一个 `libgit2` 在使用中的惯例（摘自 `libgit2` 的文档）：

以指针的形式的第一个参数返回的对象由调用者持有，调用者负责释放它们。

用 Rust 的术语来说就是，像 `git_repository_open` 这样的函数会把新值传递给调用者。

接下来，考虑查找仓库当前 head commit 的对象哈希值的代码：

```

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` 类型存储了一个对象的标识符——Git 内部使用的一个 160 位的哈希值，Git 使用它来识别 commit、文件的不同版本，等等。对 `git_reference_name_to_id` 的调用会查找当前 "HEAD" commit 的对象标识符。

在 C 语言中向函数传递一个指针，然后在函数里填充指针指向对象的值是一种非常常用的基本方法，`git_reference_name_to_id` 的第一个参数也是这样的。但是 Rust 不允许我们借用一个未初始化变量的引用。我们可以用 0 值初始化 `oid`，但这是一种浪费：因为之后存储在这里的值都会被覆盖。

要求 Rust 给我们未初始化的内存是可行的，但因为在任何时刻读取未初始化的内存都是未定义行为，因此 Rust 提供了一个抽象 `MaybeUninit` 来方便使用。`MaybeUninit<T>` 告诉编译器为你的类型 `T` 分配足够的内存空间，但并不能访问这块空间，直到你声明现在已经可以安全地访问了。当这块内存的所有权属于 `MaybeUninit` 时，编译器也可以避免一些可能会导致未定义行为的特殊优化，否则即使你没有在代码中显式地访问未初始化的内存，这些优化也可能导致未定义行为。

`MaybeUninit` 提供了一个方法 `as_mut_ptr()`，它产生一个指向它持有的未初始化内存的 `*mut T` 指针。把这个指针传给用于初始化内存的外部函数，然后调用 `MaybeUninit` 的 `unsafe` 的 `assume_init` 方法来产生一个完全初始化过的 `T`。这样既不会有先初始化后立刻重新初始化的开销，也可以避免未定义行为。`assume_init` 是 `unsafe` 的，因为如果在并没有正确初始化的内存上调用它会立刻导致未定义行为。

在这个例子中是安全的，因为 `git_reference_name_to_id` 会初始化 `MaybeUninit` 拥有的内存。我们也可以将 `MaybeUninit` 用于 `repo` 和 `commit` 变量，但因为它们只是单个字，所以我们只是直接将它们初始化为空：

```
let mut commit = ptr::null_mut();
check("looking up commit",
    raw::git_commit_lookup(&mut commit, repo, &oid));
```

这里接受 `commit` 的对象标识符然后查找 `commit`，成功时把一个 `git_commit` 对象的指针存储在 `commit` 中。

`main` 函数的其他部分应该是不言自明的。它调用了之前定义的 `show_commit` 函数，然后释放了 `commit` 和 `repository` 对象，最后关闭了库。

现在我们可以在任何 Git 仓库上尝试我们的程序：

```
$ 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 一个 libgit2 的安全接口

`libgit2` 的原始接口是一个 `unsafe` 特性的完美示例：它完全可以被正确地使用（正如我们在这里所做的一样），但 Rust 不能强迫这些你必须遵守的规则。为一个类似这样的库设置一个 `safe` 的 API 的主要问题是识别所有这些规则，然后将所有违反规则的行为转换为违反借用后检查的错误。

这里是程序使用到的 libgit2 的特性的规则：

1. 在调用库中任何其他函数之前，你必须先调用 `git_libgit2_init`。在调用 `git_libgit2_shutdown` 之后你不能再调用库中任何一个函数。
2. 除了输出参数之外，传递给 `libgit2` 函数的所有值必须是必须是完全初始化的。
3. 当一次调用失败时，用来存储调用结果的输出参数仍然保持未初始化，你绝对不能使用它的值。
4. 一个 `git_commit` 对象引用了派生它的 `git_repository` 对象，因此前者的生命周期绝对不能比后者长。（这一点并没有在 `libgit2` 的文档中说明；我们根据接口中的某些函数的存在推断出这一点，并通过阅读源码证实了它。）
5. 类似的，一个 `git_signature` 也总是借用自一个给定的 `git_commit`，因此前者的生命周期绝对不能比后者长。（文档并没有覆盖这一点。）
6. 一个 commit 关联的 message 和 author 的 name 和 email 总是借用自 commit，因此绝对不能在 commit 被 free 之后再使用它们。
7. 一旦一个 `libgit2` 对象被释放之后，绝不能再次使用它。

事实证明，你可以构建出一个强迫所有这些规则的 `libgit2` 的 Rust 接口，可能是通过 Rust 的类型系统，也可能是通过内部管理细节。

在我们开始之前，让我们重新组织一下这个项目。我们希望有一个 `git` 模块用来导出 safe 的接口，之前的程序中的原始接口作为一个 private 的子模块。

整个源码树看起来像这样：

```
git-toy/
|-- Cargo.toml
|-- build.rs
|-- src/
    |-- main.rs
    |-- git/
        |-- mod.rs
        |-- raw.rs
```

按照我们在[单独文件中的模块](#)中介绍过的规则，`git` 模块的源码应该在 `git/mod.rs`，`git::raw` 子模块的源码应该在 `git/raw.rs`。

我们将完全重写 `main.rs`。它应该以 `git` 模块的声明开始：

```
mod git;
```

然后，我们将需要创建 `git` 子目录并把 `raw.rs` 移动进去：

```
$ cd /home/jimb/git-toy
```

```
$ mkdir src/git  
$ mv src/raw.rs src/git/raw.rs
```

git 模块需要声明它的 raw 子模块。文件 `src/git/mod.rs` 必须声明：

```
mod raw;
```

因为它不是 pub 的，所以这个子模块对主程序不可见。

另外我们将需要使用一些 libc crate 里的函数，因此我们必须在 `Cargo.toml` 中添加依赖。现在这个文件完整的内容是：

```
[package]  
name = "git-toy"  
version = "0.1.0"  
authors = ["You <you@example.com>"]  
edition = "2018"  
  
[dependencies]  
libc = "0.2"
```

现在我们已经重新组织了我们的模块，让我们来考虑下错误处理。就连 libgit2 的初始化函数都可能返回一个错误码，因此我们必须在准备开始之前准备好这一点。一个规范的 Rust 接口需要自己的 `Error` 类型，它需要捕获 libgit2 的错误码和来自 `giterr_last` 的错误类型和消息。一个正确的错误类型必须实现通常的 `Error`、`Debug` 和 `Display` trait。然后，它需要自己的 `Result` 类型来使用这个 `Error` 类型。这里是 `src/git/mod.rs` 中必须的定义：

```
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> {  
        // Display 一个`Error`只需要 display `libgit2` 的错误信息  
        self.message.fmt(f)  
    }  
}
```

```

    }

impl error::Error for Error { }

pub type Result<T> = result::Result<T, Error>;

```

为了检查原始库调用的结果，这个模块需要一个函数用来把一个 libgit2 返回的错误码转换成一个 Result：

```

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 保证(*error).message 总是非空并且以空字符结尾
        // 所以这里的调用是安全的。
        let message = CStr::from_ptr((*error).message)
            .to_string_lossy()
            .into_owned();

        Err(Error {
            code: code as i32,
            message,
            class: (*error).klass as i32
        })
    }
}

```

这个函数和之前原始版本里的 check 函数的主要区别就是它构建一个 error 而不是打印出错误信息然后立即退出。

现在我们已经准备好处理 libgit2 的初始化了。safe 的接口将提供一个 Repository 类型来表示一个打开的 Git 仓库，它有一些方法用来解析引用、查找 commit 等等。继续在 git/mod.rs 中实现 Repository 的定义：

```

/// 一个 Git 仓库
pub struct Repository {

```

```
// 它必须总是指向一个还在生存的`git_repository`结构体,  
// 不能有别的`Repository`也指向同一个结构体。  
raw: *mut raw::git_repository  
}
```

Repository 的 raw 字段并不是 public 的。因为只有这个模块中的代码可以访问 raw::git_repository 指针，所以保证这个模块是正确的就可以保证指针总是被正确地使用。

如果创建一个 Repository 的唯一方法就是成功打开一个新的 Git 仓库，那么将能确保每个 Repository 都指向一个不同的 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 })  
    }  
}
```

因为 safe 接口里想做任何事都必须先创建一个 Repository 值，而且 Repository::open 里首先是一个 ensure_initialized 的调用，所以我们可以确信 ensure_initialized 会在任何 libgit2 函数之前被调用。它的定义如下：

```
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` 类型帮助我们以一种线程安全的方式运行初始化代码。只有第一个调用 `ONCE.call_once` 的线程会运行给定的闭包。任何之后的调用，不管是这个线程还是别的线程，都会阻塞住直到第一次调用结束，然后它们会立刻返回，不会再次运行给定的闭包。一旦闭包结束之后，调用 `ONCE.call_once` 的开销就非常小了，只需要原子地读取一个存储在 `ONCE` 里的标记。

在上面的代码中，初始化闭包调用了 `git_libgit2_init` 然后检查结果。它稍微简化了一下，直接用 `expect` 来保证初始化成功，而没有尝试把错误传播回调用者。

为了确保程序会调用 `git_libgit2_shutdown`，初始化闭包使用了 C 库的 `atexit` 函数，它接受一个在退出进程之前要调用的函数的指针。Rust 的闭包不能用作 C 函数的指针：一个闭包实际上是一个匿名类型的值，它还携带着它捕获或者引用到的值；一个 C 函数指针只是一个指针。然而，Rust 的 `fn` 类型可以用作函数指针，只要你用 `extern` 来声明它们以让 Rust 知道要使用 C 的惯例。本地函数 `shutdown` 里调用 `git_libgit2_shutdown` 并保证 `libgit2` 正确地退出。

在 [栈展开](#) 中，我们提到过跨语言边界的 panic 是未定义行为。从 `atexit` 到 `shutdown` 的调用就是这样一个边界，因此 `shutdown` 不能 panic。这就是为什么 `shutdown` 没有简单地使用 `.expect` 来处理 `raw::git_libgit2_shutdown` 返回的错误，而是手动汇报错误并中断进程。POSIX 禁止在 `atexit` 里调用 `exit`，因此 `shutdown` 调用了 `std::process::abort` 来突然终止程序。

更早地调用 `git_libgit2_shutdown` 也是可行的，即在最后一个 `Repository` 值被 `drop` 的时候调用。但无论如何，调用 `git_libgit2_shutdown` 必须是 safe API 的责任。在调用它时，任何现存的 `libgit2` 对象都会变为不能再使用的状态，因此一个 safe API 绝对不能直接暴露这个函数。

一个 `Repository` 的原始指针必须总是指向一个还在生存的 `git_repository` 对象。这隐含了关闭一个仓库的唯一方法是 `drop` 掉拥有它的 `Repository`：

```

impl Drop for Repository {
    fn drop(&mut self) {
        unsafe {

```

```
    raw::git_repository_free(self.raw);
}
}
}
```

通过在指向 `raw::git_repository` 的指针即将销毁时调用 `git_repository_free`, `Repository` 类型还可以保证这个指针永远不会再在 `free` 掉之后再被使用。

`Repository::open` 方法使用了一个叫做 `path_to_cstring` 的 private 函数, 它有两个定义——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` 方法只在类 Unix 系统中存在。
    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。如果失败了, libgit2 就不能处理这个路径了。
    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 的接口让这段代码变得更棘手一点。在所有的平台上, libgit2 都以空字符结尾的 C 字符串接受路径。在 Windows 上, libgit2 假设这些 C 字符串持有有效的 UTF-8, 并在内部把它们转换成 Windows 实际要求的 16 位路径。这通常是可以工作的, 但并不是最理想的方案。Windows 允许文件名不是有效的 Unicode, 因此也不能用 UTF-8 来表示。如果你有一个这样的文件, 那么不可能把它的名字传递给 libgit2。

在 Rust 中, 文件系统路径的正确表示是 `std::path::Path`, 它被精心设计用来处理任何可能出现在 Windows 或 POSIX 上的路径。这意味着有一些 Windows 上的 Path 值不能传递

给 libgit2，因为它们不是有效的 UTF-8。因此尽管 `path_to_cstring` 的行为不是很理想，但它确实是最好的方案了。

两个 `path_to_cstring` 的定义都依赖到我们的 `Error` 类型的转换：`? 运算符` 会尝试这样的转换，并且 Windows 版本的显式调用了 `.into()`。这些转换并不起眼：

```
impl From<String> for Error {
    fn from(message: String) -> Error {
        Error { code: -1, message, class: 0 }
    }
}

// NulError 是如果字符串里有 0 字节时 `CString::new` 会返回的 Error 类型
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，让我们看看如何把一个 Git 引用解析到一个对象标识符。因为一个对象标识符只是一个 20 字节的哈希值，因此直接在 safe API 中暴露它也是完全没问题的：

```
/// 一些存储在 Git 对象数据库中的对象 (commit、tree、blob、tag 等) 的标识符。
/// 它是对象内容的哈希值。
pub struct Oid {
    pub raw: raw::git_oid
}
```

我们将给 `Repository` 添加一个方法来执行查找功能：

```
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 })
        }
    }
}
```

```
    };  
    Ok(Oid { raw: oid })  
}
```

尽管在查找失败时 `oid` 仍然保持未初始化的状态，但这个函数通过遵循 Rust 的 `Result` 的惯例保证它的调用者永远不可能看到这个未初始化的值：调用者要么得到一个携带着正确初始化的 `Oid` 值的 `Ok`，要么得到一个 `Err`。

接下来，这个模块需要一种方法从仓库里提取 commit，我们的 Commit 类型定义如下：

```
use std::marker::PhantomData;
```

```
pub struct Commit<'repo> {
    // 这个指针总是指向一个可用的`git_commit`结构体。
    raw: *mut raw::git_commit,
    _marker: PhantomData<&'repo Repository>
}
```

正如我们之前提到的一样，一个 `git_commit` 对象的生命周期必须不长于它引用的 `git_repository` 对象。Rust 的生命周期可以让代码精准地捕获这个规则。

本章之前的 RefWithFlag 例子使用了一个 PhantomData 字段来告诉 Rust 在对待一个类型时，把它看成持有一个给定生命周期的引用，即使这个类型显然并不持有这样的引用。 Commit 类型需要类似的操作。在这个例子中，_marker 字段的类型是 PhantomData<&'repo Repository>，它告诉 Rust 应该把 Commit<'repo> 看成好像持有一个生命周期是 'repo 的 Repository 的引用。

查找一个 commit 的方法如下：

```
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 })  
    }  
}
```

这是怎么把 Commit 的生命周期关联到 Repository 的生命周期的？根据省略生命周期参数中列出的规则，`find_commit` 的签名省略了引用的生命周期。如果我们写出生命周期的话，完整的签名应该是：

```
fn find_commit<'repo, 'id>(&'repo self, oid: &'id Oid)
-> Result<Commit<'repo>>
```

这正是我们想要的：Rust 对待返回的 Commit 时就好像它借用了 self（即 Repository）一样。

当一个 Commit 被 drop 时，它必须释放它的 raw::git_commit：

```
impl<'repo> Drop for Commit<'repo> {
    fn drop(&mut self) {
        unsafe {
            raw::git_commit_free(self.raw);
        }
    }
}
```

你可以从一个 Commit 借用一个 Signature（一个 name 和 email 地址）和 commit message 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)
        }
    }
}
```

这是 Signature 类型：

```
pub struct Signature<'text> {
    raw: *const raw::git_signature,
    _marker: PhantomData<&'text str>
}
```

一个`git_signature`对象总是从其他地方借用内容；具体来说，`git_commit_author`返回的签名从`git_commit`借用文本。因此我们的`safe Signature`类型包含一个`PhantomData<&'text str>`来告诉Rust把它看做好像持有一个生命周期是`'text`的`&str`。和之前一样，不需要我们写任何内容，`Commit::author`也可以正确地把它返回的`Signature`的`'text`生命周期关联到那个`Commit`的生命周期。`Commit::message`方法对持有`commit message`的`Option<&str>`也实现了相同的效果。

一个`Signature`包含一些获取`author`的`name`和`email`地址的方法：

```
impl<'text> Signature<'text> {
    /// 以`&str`返回author的name,
    /// 如果不是有效的UTF-8则返回`None`
    pub fn name(&self) -> Option<&str> {
        unsafe {
            char_ptr_to_str(self, (*self.raw).name)
        }
    }

    /// 以`&str`返回author的email,
    /// 如果不是有效的UTF-8则返回`None`
    pub fn email(&self) -> Option<&str> {
        unsafe {
            char_ptr_to_str(self, (*self.raw).email)
        }
    }
}
```

上面的方法依赖于一个`private`的工具函数`char_ptr_to_str`：

```
/// 尝试从`ptr`借用一个`&str`，给定的`ptr`可能为空或者指向无效的UTF-8.
/// 给返回结果赋予一个好像从`_owner`借用的生命周期。
///
/// 安全性：如果`ptr`非空，它必须指向一个空字符结尾的C字符串，
/// 这个字符串的生命周期必须至少和`_owner`一样长。
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()
    }
}
```

`_owner` 参数的值永远不会被使用，但它的生命周期会被用到。如果显式地标注这个函数的生命周期将是：

```
fn char_ptr_to_str<'o, T: 'o>(_owner: &'o T, ptr: *const c_char)
-> Option<&'o str>
```

`CStr::from_ptr` 函数会返回一个生命周期不受限的 `&CStr`，因为它是从原始指针借用的。不受限的生命周期几乎总是不好的，因此应该尽可能地约束它们。包含 `_owner` 参数会导致 Rust 把它的生命周期赋给返回值，因此调用者可以得到一个被更加精确地约束的引用。

尽管 `libgit2` 的文档相当不错，但它并没有说清楚 `git_signature` 的 `email` 和 `author` 是否可以为空。我们在源代码套索掘了一段时间，但无法说服自己它一定不为空，最后决定让 `char_ptr_to_str` 最好为空指针做准备，以防万一。在 Rust 中，这种问题可以通过类型快速回答：如果是 `&str`，那你可以确信一定有一个字符串；如果是 `Option<&str>`，那就可能为空。

最后，我们为所需的所有功能提供了 safe 的接口。`src/main.rs` 中的新 `main` 函数就简单很多，并且看起来更像真正的 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，我们从最简单的不提供很多安全性保证的接口开始，到通过以内部的 unsafe API 为基础，把任何对合约的违反都转换成 Rust 的类型错误来构建出 safe 的 API。最后得到

的结果是一个 Rust 可以保证被正确使用的接口。在大多数情况下，我们让 Rust 强迫的规则都是 C 和 C++ 程序自己隐式遵守的那些规则。Rust 之所以感觉起来比 C 和 C++ 严格得多，并不是因为这些规则很陌生，而是因为这种规则被机械地、全面地强制。

23.6 总结

Rust 并不是一门简单的语言。它的目标横跨了两个不同的世界。它是一门现代化的编程语言，以安全为设计原则，同时还有像闭包和迭代器这样的便利设施，但它的目标是让你能以最小的运行时开销来控制机器的原始能力。

这门语言的轮廓由这些目标决定。Rust 设法用 safe 代码来实现大部分的功能。它的借用检查器和 0 开销抽象让你尽可能地接近裸机，同时还没有未定义行为的风险。当这些还不够或者当你想使用现有的 C 代码时，unsafe 代码和外部函数接口已准备就绪。但再重复一次，它并不是只提供给你 unsafe 特性然后祝你好运，它的目标总是使用 unsafe 特性来构建 safe 的 API。这正是我们基于 `libgit2` 所做的工作。这也是 Rust 的团队对 `Box`、`Vec`、其他集合、`channel` 等等设施所做的工作：标注库充满了用 unsafe 代码实现的 safe 的抽象。

Rust 的雄心也许并不是成为最简单的工具。但 Rust 是安全、快速、并发、高效的。使用它来构建大型、快速、安全、强大的系统，充分利用它们所运行的硬件的力量。使用它来让软件变得更好。

