

目录

入门指南

Introduction	1.1
介绍	1.2
安装	1.2.1
Hello, World!	1.2.2
猜猜看教程	1.3
通用编程概念	1.4
变量与可变性	1.4.1
数据类型	1.4.2
函数如何工作	1.4.3
注释	1.4.4
控制流	1.4.5
认识所有权	1.5
什么是所有权	1.5.1
引用与借用	1.5.2
Slices	1.5.3
使用结构体来组织相关联的数据	1.6
定义并实例化结构体	1.6.1
一个使用结构体的示例程序	1.6.2
方法语法	1.6.3
枚举与模式匹配	1.7
定义枚举	1.7.1
match 控制流运算符	1.7.2
if let 简洁控制流	1.7.3

基本 Rust 技能

模块	2.1
mod 与文件系统	2.1.1
使用 pub 控制可见性	2.1.2
在不同的模块中引用命名	2.1.3
通用集合类型	2.2
vector	2.2.1
字符串	2.2.2

哈希 map	2.2.3
错误处理	2.3
panic! 与不可恢复的错误	2.3.1
Result 与可恢复的错误	2.3.2
panic! 还是不 panic!	2.3.3
泛型、trait 与生命周期	2.4
泛型数据类型	2.4.1
trait: 定义共享的行为	2.4.2
生命周期与引用有效性	2.4.3
测试	2.5
编写测试	2.5.1
运行测试	2.5.2
测试的组织结构	2.5.3
一个 I/O 项目：构建命令行程序	2.6
接受命令行参数	2.6.1
读取文件	2.6.2
重构以改进模块化与错误处理	2.6.3
采用测试驱动开发完善库的功能	2.6.4
处理环境变量	2.6.5
将错误信息输出到标准错误而不是标准输出	2.6.6

Rust 编程思想

Rust 中的函数式语言功能：迭代器与闭包	3.1
闭包：可以捕获其环境的匿名函数	3.1.1
使用迭代器处理元素序列	3.1.2
改进之前的 I/O 项目	3.1.3
性能比较：循环对迭代器	3.1.4
更多关于 Cargo 和 Crates.io 的内容	3.2
采用发布配置自定义构建	3.2.1
将 crate 发布到 Crates.io	3.2.2
Cargo 工作空间	3.2.3
使用 cargo install 从 Crates.io 安装二进制文件	3.2.4
Cargo 自定义扩展命令	3.2.5
智能指针	3.3
Box 在堆上存储数据，并且可确定大小	3.3.1
Deref Trait 允许通过引用访问数据	3.3.2
Drop Trait 运行清理代码	3.3.3
Rc 引用计数智能指针	3.3.4

RefCell 和内部可变性模式	3.3.5
引用循环和内存泄漏是安全的	3.3.6
无畏并发	3.4
线程	3.4.1
消息传递	3.4.2
共享状态	3.4.3
可扩展的并发: Sync和Send	3.4.4
Rust 是面向对象语言吗?	3.5
什么是面向对象?	3.5.1
为使用不同类型的值而设计的 trait 对象	3.5.2
面向对象设计模式的实现	3.5.3

高级主题

模式用来匹配值的结构	4.1
所有可能会用到模式的位置	4.1.1
Refutability: 何时模式可能会匹配失败	4.1.2
模式的全部语法	4.1.3
高级特征	4.2
不安全的 Rust	4.2.1
高级生命周期	4.2.2
高级 trait	4.2.3
高级类型	4.2.4
高级函数与闭包	4.2.5
最后的项目: 构建多线程 web server	4.3
单线程 web server	4.3.1
慢请求如何影响吞吐率	4.3.2
设计线程池接口	4.3.3
创建线程池并储存线程	4.3.4
使用通道向线程发送请求	4.3.5
Graceful Shutdown 与清理	4.3.6
附录	4.4
A - 关键字	4.4.1
B - 运算符与符号	4.4.2
C - 可导出的 trait	4.4.3
D - 宏	4.4.4
E - 本书翻译	4.4.5
F - 最新功能	4.4.6

Rust 程序设计语言（第二版） 简体中文版

还在施工中...

介绍

ch01-00-introduction.md

commit d06a6a181fd61704cbf7feb55bc61d518c6469f9

欢迎阅读“**Rust** 程序设计语言”，一本介绍 **Rust** 的书。**Rust** 是一门着眼于安全、速度和并发的编程语言。其程序设计兼顾底层语言的性能与控制，并不失高级语言强大的抽象能力。其特性适合那些有类 **C** 语言经验，正在寻找更安全的替代品的开发者；同样适合有着类 **Python** 语言背景，寻求在不牺牲表现力的前提下，编写更高性能代码的开发者。

Rust 编译时执行绝大部分的安全检查和内存管理决策，对运行时性能的影响微不足道。这使其在其他语言不擅长的应用场景中得以大显身手：可预测时间和空间需求的程序，嵌入到其他语言中，以及编写如设备驱动和操作系统这样的底层代码。**Rust** 也很擅长 **web** 程序：它驱动着 **Rust** 包登记网站（package registry site），crates.io！我们由衷期待你使用 **Rust** 进行创作。

本书的目标读者至少应了解一门其它编程语言。阅读本书之后，你应该能自如的编写 **Rust** 程序。我们将通过短小精干、前后呼应的例子来学习 **Rust**，并展示其多样功能的使用方法，同时了解它们幕后如何运行。

为本书做出贡献

本书是开源的。如果你发现任何错误，不要犹豫，在 [GitHub](#) 上发起 issue 或提交 pull request。请查看 [CONTRIBUTING.md](#) 获取更多信息。

译者注：本译本的 [GitHub 仓库](#)，同样欢迎 Issue 和 PR :)

安装

ch01-01-installation.md

commit b6dcc87f2b811c88fc741c86cf6ed3976333acba

使用 **Rust** 的第一步是安装。你需要网络连接来执行本章的命令，因为将要下载 **Rust**。

这里将会展示很多使用终端的命令，这些命令均以 `$` 开头。不需要真的输入 `$`，在这里它们代表每行命令的起始。网上有很多教程和例子遵循这种惯例：`$` 代表以常规用户身份运行命令，`#` 代表需要用管理员身份运行命令。没有以 `$`（或 `#`）起始的行通常是之前命令的输出。

在 Linux 或 Mac 上安装

如果你使用 **Linux** 或 **Mac**，你需要做的全部就是打开一个终端并输入：

```
$ curl https://sh.rustup.rs -sSf | sh
```

这会下载一个脚本并开始安装。可能会提示你输入密码，如果一切顺利，将会出现如下内容：

```
Rust is installed now. Great!
```

当然，如果你不信任采用 `curl URL | sh` 来安装软件，请随意下载、检查和运行这个脚本。

此安装脚本自动将 **Rust** 加入系统 **PATH** 环境变量中，再次登陆时生效。如果你希望立刻（不重新登陆）就开始使用 **Rust**，在 **shell** 中运行如下命令：

```
$ source $HOME/.cargo/env
```

或者，在 `~/.bash_profile` 文件中增加如下行：

```
$ export PATH="$HOME/.cargo/bin:$PATH"
```

在 Windows 上安装

如果你使用 **Windows**，前往 <https://rustup.rs>，按说明下载 `rustup-init.exe`，运行并照其指示操作。

本书中其余 **Windows** 相关的命令，假设你使用 `cmd` 作为 **shell**。如果你使用其它 **shell**，也许可以执行与 **Linux** 和 **Mac** 用户相同的命令。如果不行，请查看该 **shell** 的文档。

自定义安装

无论出于何种理由，如果不愿意使用 `rustup.rs`，请查看 [Rust 安装页面](#) 获取其他选项。

更新

一旦安装了 **Rust**，更新到最新版本是很简单的。在 **shell** 中执行更新脚本：

```
$ rustup update
```

卸载

卸载 Rust 与安装同样简单。在 shell 中执行卸载脚本:

```
$ rustup self uninstall
```

故障排除（Troubleshooting）

安装完 Rust 后，打开 shell 并执行：

```
$ rustc --version
```

应该能看到类似这样格式的版本号、提交哈希和提交日期，对应安装时的最新稳定版：

```
rustc x.y.z (abcabcabc yyyy-mm-dd)
```

如果出现这些内容，Rust 就安装成功了！

恭喜入坑！（此处应该有掌声！）

如果在 Windows 中使用出现问题，检查 Rust（rustc，cargo 等）是否在 `%PATH%` 环境变量所包含的路径中。

如果还是不能解决，有许多地方可以求助。最简单的是 irc.mozilla.org 上的 [#rust IRC 频道](#)，可以使用 [Mibbit](#) 来访问它。然后就能和其他 Rustacean（Rust 用户的称号，有自嘲意味）聊天并寻求帮助。其它给力的资源包括[用户论坛](#)和 [Stack Overflow](#)。

本地文档

安装程序也自带一份文档的本地拷贝，可以离线阅读。运行 `rustup doc` 在浏览器中查看本地文档。

任何时候，如果你拿不准标准库中的类型或函数如何工作，请查看 [API 文档](#)！

Hello, World!

ch01-02-hello-world.md

commit c21a4e7b315d62583470482ab542587a26e1b2e8

现在安装好了 **Rust**，让我们来编写第一个程序。当学习一门新语言的时候，使用该语言在屏幕上打印“**Hello, world!**”是一项传统，这里我们将遵循这个传统。

注意：本书假设你熟悉基本的命令行操作。**Rust** 对于你的编辑器、工具，以及代码位于何处并没有特定的要求，如果相比命令行你更倾向于 IDE，请随意使用合意的 IDE。

创建项目目录

首先，创建一个存放 **Rust** 代码的目录。**Rust** 并不关心代码的位置，不过在本书中，我们建议你在 **home** 目录中创建一个 *projects* 目录，并将你的所有项目置于此处。打开终端并输入如下命令为此项目创建一个目录：

Linux 和 Mac:

```
$ mkdir ~/projects
$ cd ~/projects
$ mkdir hello_world
$ cd hello_world
```

Windows 的 cmd:

```
> mkdir %USERPROFILE%\projects
> cd %USERPROFILE%\projects
> mkdir hello_world
> cd hello_world
```

Windows 的 PowerShell:

```
> mkdir $env:USERPROFILE\projects
> cd $env:USERPROFILE\projects
> mkdir hello_world
> cd hello_world
```

编写并运行 **Rust** 程序

接下来，新建一个叫做 *main.rs* 的源文件。**Rust** 源代码总是以 *.rs* 后缀结尾。如果文件名包含多个单词，使用下划线分隔它们。例如 *my_program.rs*，而不是 *myprogram.rs*。

现在打开刚创建的 *main.rs* 文件，输入如下代码：

文件名: *main.rs*

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

保存文件，并回到终端窗口。在 **Linux** 或 **OSX** 上，输入如下命令：

```
$ rustc main.rs
$ ./main
Hello, world!
```

在 **Windows** 上，运行 `.\main.exe`，而不是 `./main`。不管使用何种系统，都应该在终端看到 `Hello, world!` 字样。如果你做到了，恭喜你！你已经正式编写了一个 **Rust** 程序。现在你成为了一名 **Rust** 程序员！欢迎！

分析 **Rust** 程序

现在，让我们回过头来仔细看看“**Hello, world!**”程序中到底发生了什么。这是拼图的第一片：

```
fn main() {

}
```

这几行定义了一个 **Rust** 函数。`main` 函数是特殊的：它是每个可执行的 **Rust** 程序所首先执行的。第一行代码表示“我声明了一个叫做 `main` 的函数，它没有参数也没有返回值。”如果有参数的话，它们的名称应该出现在括号中，位于 `(` 和 `)` 之间。

还须注意函数体被包裹在花括号中，`{` 和 `}` 之间。**Rust** 要求所有函数体都要用花括号包裹起来（译者注：有些语言，当函数体只有一行时可以省略花括号，但在 **Rust** 中是不行的）。一般来说，将左花括号与函数声明置于同一行并以空格分隔，是良好的代码风格。

在 `main()` 函数中：

```
println!("Hello, world!");
```

这行代码完成这个小程序的所有工作：在屏幕上打印文本。这里有很多细节需要注意。首先 **Rust** 使用 4 个空格的缩进风格，而不是 1 个制表符（`tab`）。

第二个重要的部分是 `println!()`。这称为 **Rust** 宏，**Rust** 元编程（**metaprogramming**）的关键所在。如果是调用函数，则应看起来像这样：`println`（没有 `!`）。我们将在附录 **D** 中更加详细的讨论宏，现在你只需记住，当看到符号 `!` 的时候，就意味着调用的是宏而不是普通函数。

接下来，`"Hello, world!"` 是一个字符串。我们把这个字符串作为一个参数传递给 `println!`，它负责在屏幕上打印这个字符串。轻松加愉快！(☺o☺)

该行以分号结尾（`;`）。`;` 代表一个表达式的结束和下一个表达式的开始。大部分 **Rust** 代码行以 `;` 结尾。

编译和运行是彼此独立的步骤

“编写并运行 **Rust** 程序”部分中展示了如何运行新创建的程序。现在我们将拆分并检查每一步操作。

在运行 **Rust** 程序之前必须先进行编译。可以通过 `rustc` 命令并传递源文件名称来使用 **Rust** 编译器，如下：

```
$ rustc main.rs
```

如果你有 **C** 或 **C++** 背景，就会发现这与 `gcc` 和 `clang` 类似。编译成功后，**Rust** 应该会输出一个二进制可执行文件，在 **Linux** 或 **OSX** 上在 **shell** 中可以通过 `ls` 命令看到如下内容：

```
$ ls
main main.rs
```

在 Windows 上，输入：

```
> dir /B %* the /B option says to only show the file names %*
main.exe
main.rs
```

这表示我们有两个文件：`.rs` 后缀的源文件，和可执行文件（在 Windows 下是 `main.exe`，其它平台是 `main`）。余下需要做的就是运行 `main` 或 `main.exe` 文件，如下：

```
$ ./main # or .\main.exe on Windows
```

如果 `main.rs` 是上文所述的“Hello, world!”程序，它将会在终端上打印 `Hello, world!`。

来自 Ruby、Python 或 JavaScript 这样的动态类型语言背景的同学，可能不太习惯将编译和执行分为两个单独的步骤。Rust 是一种 预编译静态类型（*ahead-of-time compiled*）语言，这意味着你可以编译程序并将其交与他人，它们不需要安装 Rust 即可运行。相反如果你给他们一个 `.rb`、`.py` 或 `.js` 文件，他们需要先分别安装 Ruby、Python、JavaScript 实现（运行时环境，VM），不过你只需要一句命令就可以编译和执行程序。这一切都是语言设计上的权衡取舍。

使用 `rustc` 编译简单程序是没问题的，不过随着项目的增长，你可能需要控制你项目的方方面面，并且更容易地将代码分享给其它人或项目。接下来，我们要介绍一个叫做 **Cargo** 的工具，它会帮助你编写真实世界中的 Rust 程序。

Hello, Cargo!

Cargo 是 Rust 的构建系统和包管理工具，同时 Rustacean 们使用 Cargo 来管理他们的 Rust 项目，因为它使得很多任务变得更轻松。例如，Cargo 负责构建代码、下载依赖库并编译它们。我们把代码所需要的库叫做 依赖（*dependencies*）。

最简单的 Rust 程序，比如我们刚刚编写的，并没有任何依赖，所以目前我们只会用到 Cargo 构建代码那部分的功能。随着编写的程序更加复杂，你会想要添加依赖，如果你一开始就使用 Cargo 的话，事情会变得简单许多。

由于绝大部分 Rust 项目使用 Cargo，本书接下来的部分将假设你使用它。如果使用之前介绍的官方安装包的话，则自带了 Cargo。如果通过其他方式安装的话，可以在终端输入如下命令检查是否安装了 Cargo：

```
$ cargo --version
```

如果出现了版本号，一切 OK！如果出现类似 `command not found` 的错误，你应该查看相应安装文档以确定如何单独安装 Cargo。

使用 Cargo 创建项目

让我们使用 Cargo 来创建一个新项目，然后看看与上面的 `hello_world` 项目有什么不同。回到 `projects` 目录（或者任何你放置代码的目录）：

Linux 和 Mac:

```
$ cd ~/projects
```

Windows:

```
> cd %USERPROFILE%\projects
```

并在任何操作系统下运行：

```
$ cargo new hello_cargo --bin
$ cd hello_cargo
```

我们向 `cargo new` 传递了 `--bin`，因为我们的目标是生成一个可执行程序，而不是一个库。可执行程序是二进制可执行文件，通常就叫做 二进制文件（*binaries*）。项目的名称被定为 `hello_cargo`，同时 Cargo 在一个同名目录中创建它的文件，接着我们可以进入查看。

如果列出 `hello_cargo` 目录中的文件，将会看到 Cargo 生成了一个文件和一个目录：一个 `Cargo.toml` 文件和一个 `src` 目录，`main.rs` 文件位于 `src` 目录中。它也在 `hello_cargo` 目录初始化了一个 git 仓库，以及一个 `.gitignore` 文件；你可以通过 `--vcs` 参数切换到其它版本控制系统（VCS），或者不使用 VCS。

使用文本编辑器（工具请随意）打开 `Cargo.toml` 文件。它应该看起来像这样：

文件名: `Cargo.toml`

```
[package]
name = "hello_cargo"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]

[dependencies]
```

这个文件使用 [TOML](#) (Tom's Obvious, Minimal Language) 格式。TOML 类似于 INI，不过有一些额外的改进之处，并且被用作 Cargo 的配置文件的格式。

第一行，`[package]`，是一个部分标题，表明下面的语句用来配置一个包。随着我们在这个文件增加更多的信息，还将增加其他部分。

接下来的三行设置了三个 Cargo 所需的配置，项目的名称、版本和作者，它们告诉 Cargo 需要编译这个项目。Cargo 从环境中获取你的名称和 email 信息。如果不正确，请修改并保存此文件。

最后一行，`[dependencies]`，是项目依赖的 *crates* 列表（我们称呼 Rust 代码包为 *crate*）部分的开始，这样 Cargo 就知道应该下载和编译它们了。这个项目并不需要任何其他的 *crate*，不过在下一章猜想看教程会得上。

现在看看 `src/main.rs`：

文件名: `src/main.rs`

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

Cargo 为你生成了一个“Hello World!”，正如我们之前编写的那个！目前为止，之前项目与 Cargo 生成项目的区别有：

- 代码位于 `src` 目录
- 项目根目录包含一个 `Cargo.toml` 配置文件

Cargo 期望源文件位于 `src` 目录，将项目根目录留给 README、license 信息、配置文件和其他跟代码无关的文件。这样，Cargo 帮助你保持项目干净整洁，一切井井有条。

如果没有用 **Cargo** 创建项目，比如 *hello_world* 目录中的项目，可以通过将代码放入 *src* 目录，并创建一个合适的 *Cargo.toml*，将其转化为一个 **Cargo** 项目。

构建并运行 **Cargo** 项目

现在让我们看看通过 **Cargo** 构建和运行 **Hello World** 程序有什么不同。为此输入下面的命令：

```
$ cargo build
Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
Finished dev [unoptimized + debuginfo] target(s) in 2.85 secs
```

这应该会创建 *target/debug/hello_cargo* 可执行文件（或者在 **Windows** 上是 *target\debug\hello_cargo.exe*），可以通过这个命令运行：

```
$ ./target/debug/hello_cargo # or .\target\debug\hello_cargo.exe on Windows
Hello, world!
```

很好！如果一切顺利，*Hello, world!* 应该再次打印在终端上。

首次运行 `cargo build` 的时候，**Cargo** 会在项目根目录创建一个新文件，*Cargo.lock*，它看起来像这样：

文件名: *Cargo.lock*

```
[root]
name = "hello_cargo"
version = "0.1.0"
```

Cargo 使用 *Cargo.lock* 来记录程序的依赖。这个项目并没有依赖，所以其内容比较少。事实上，你自己永远也不需要碰这个文件，让 **Cargo** 处理它就行了。

我们刚刚使用 `cargo build` 构建了项目并使用 `./target/debug/hello_cargo` 运行了程序，也可以使用 `cargo run` 同时编译并运行：

```
$ cargo run
Running `target/debug/hello_cargo`
Hello, world!
```

注意这一次并没有出现 **Cargo** 正在编译 *hello_cargo* 的输出。**Cargo** 发现文件并没有被改变，就直接运行了二进制文件。如果修改了源文件的话，**Cargo** 会在运行之前重新构建项目，并会出现像这样的输出：

```
$ cargo run
Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
Finished dev [unoptimized + debuginfo] target(s) in 0.33 secs
Running `target/debug/hello_cargo`
Hello, world!
```

所以现在又出现更多的不同：

- 使用 `cargo build` 构建项目（或使用 `cargo run` 一步构建并运行），而不是使用 `rustc`
- 有别于将构建结果放在与源码相同的目录，**Cargo** 会将其放到 *target/debug* 目录。

Cargo 的另一个优点是，不管你使用什么操作系统其命令都是一样的，所以本书之后将不再为 **Linux** 和 **Mac** 以及 **Windows** 提供相应的命令。

发布（**release**）构建

当项目最终准备好发布了，可以使用 `cargo build --release` 来优化编译项目。这会在 `target/release` 而不是 `target/debug` 下生成可执行文件。这些优化可以让 **Rust** 代码运行的更快，不过启用这些优化也需要消耗更长的编译时间。这也就是为什么会有两种不同的配置：一种为了开发，你需要经常快速重新构建；另一种为了构建给用户最终程序，它们不会重新构建，并且希望程序运行得越快越好。如果你在测试代码的运行时间，请确保运行 `cargo build --release` 并使用 `target/release` 下的可执行文件进行测试。

把 **Cargo** 当作习惯

对于简单项目，**Cargo** 并不比 `rustc` 提供了更多的优势，不过随着开发的深入终将证明其价值。对于拥有多个 `crate` 的复杂项目，让 **Cargo** 来协调构建将更简单。有了 **Cargo**，只需运行 `cargo build`，然后一切将有序运行。即便这个项目很简单，它现在也使用了很多你之后的 **Rust** 生涯将会用得上的实用工具。其实你可以使用下面的命令开始任何你想要从事的项目：

```
$ git clone someurl.com/someproject
$ cd someproject
$ cargo build
```

注意：如果想要了解 **Cargo** 更多的细节，请阅读官方的 [Cargo guide](#)，它覆盖了 **Cargo** 所有的功能。

猜查看

ch02-00-guessing-game-tutorial.md

commit 8a145ebea5c05f07fc240269bc9557340972188f

让我们一起动手完成一个项目，来快速上手 **Rust**！本章将介绍 **Rust** 中常用的一些概念，并通过真实的程序来展示如何运用它们。你将会学到更多诸如 `let`、`match`、方法、关联函数、外部 `crate` 等很多的知识！后继章节会深入探索这些概念的细节。在这一章，我们将练习基础。

我们会实现一个经典的新手编程问题：猜查看游戏。它是这么工作的：程序将会随机生成一个 1 到 100 之间的随机整数。接着它会请玩家猜一个数并输入，然后提示猜测是大了还是小了。如果猜对了，它会打印祝贺信息并退出。

准备一个新项目

要创建一个新项目，进入第一章中创建的 `projects` 目录，使用 **Cargo** 新建一个项目，如下：

```
$ cargo new guessing_game --bin
$ cd guessing_game
```

第一个命令，`cargo new`，它获取项目的名称（`guessing_game`）作为第一个参数。`--bin` 参数告诉 **Cargo** 创建一个二进制项目，与第一章类似。第二个命令进入到新创建的项目目录。

看看生成的 `Cargo.toml` 文件：

文件名: `Cargo.toml`

```
[package]
name = "guessing_game"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]

[dependencies]
```

如果 **Cargo** 从环境中获取的开发者信息不正确，修改这个文件并再次保存。

正如第一章那样，`cargo new` 生成了一个“Hello, world!”程序。查看 `src/main.rs` 文件：

文件名: `src/main.rs`

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

现在编译“Hello, world!”程序，使用 `cargo run` 编译运行一步到位：

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
   Running `target/debug/guessing_game`
Hello, world!
```

`run` 命令适合用于需要快速迭代的项目，而这个游戏便是这样的项目：我们需要在下一步迭代之前快速测试每一步。

重新打开 `src/main.rs` 文件。我们将会在这个文件中编写全部的代码。

处理一次猜测

程序的第一部分请求和处理用户输入，并检查输入是否符合预期的格式。首先，允许用户输入猜测。在 `src/main.rs` 中输入示例 2-1 中的代码。

文件名: `src/main.rs`

```
use std::io;

fn main() {
    println!("Guess the number!");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {}", guess);
}
```

示例 2-1：获取用户猜测并打印的代码

这些代码包含很多信息，我们一点一点地过一遍。为了获取用户输入并打印结果作为输出，我们需要将 `io`（输入/输出）库引入当前作用域。`io` 库来自于标准库（也被称为 `std`）：

```
use std::io;
```

Rust 默认只在每个程序的 *prelude* 中引入少量类型。如果需要的类型不在 *prelude* 中，你必须使用一个 `use` 语句显式的将其引入作用域。`std::io` 库提供很多 `io` 相关的功能，比如接受用户输入的功能。

如第一章所提及，`main` 函数是程序的入口点：

```
fn main() {
```

`fn` 语法声明了一个新函数，`()` 表明没有参数，`{` 作为函数体的开始。

第一章也提及了 `println!` 是一个在屏幕上打印字符串的宏：

```
println!("Guess the number!");

println!("Please input your guess.");
```

这些代码仅仅打印提示，介绍游戏的内容然后请求用户输入。

使用变量储存值

接下来，创建一个地方储存用户输入，像这样：


```
let mut guess = String::new();
```

现在程序开始变得有意思了！这一小行代码发生了很多事。注意这是一个 `let` 语句，用来创建 变量。这里是另外一个例子：

```
let foo = bar;
```

这行代码新建了一个叫做 `foo` 的变量并把它绑定到值 `bar` 上。在 **Rust** 中，变量默认是不可变的。下面的例子展示了如何在变量名前使用 `mut` 来使一个变量可变：

```
let foo = 5; // immutable
let mut bar = 5; // mutable
```

注意：`//` 语法开始一个持续到本行的结尾的注释。**Rust** 忽略注释中的所有内容。

现在我们知道了 `let mut guess` 会引入一个叫做 `guess` 的可变变量。等号（`=`）的右边是 `guess` 所绑定的值，它是 `String::new` 的结果，这个函数会返回一个 `String` 的新实例。`String` 是一个标准库提供的字符串类型，它是 UTF-8 编码的可增长文本块。

`::new` 那一行的 `::` 语法表明 `new` 是 `String` 类型的一个 关联函数（*associated function*）。关联函数是针对类型实现的，在这个例子中是 `String`，而不是 `String` 的某个特定实例。一些语言中把它称为 静态方法（*static method*）。

`new` 函数创建了一个新的空 `String`，你会在很多类型上发现 `new` 函数，这是创建类型实例的惯用函数名。

总结一下，`let mut guess = String::new();` 这一行创建了一个可变变量，当前它绑定到一个新的 `String` 空实例上。

回忆一下，我们在程序的第一行使用 `use std::io;` 从标准库中引入了输入/输出功能。现在调用 `io` 的关联函数 `stdin`：

```
io::stdin().read_line(&mut guess)
    .expect("Failed to read line");
```

如果程序的开头没有 `use std::io;` 这一行，可以把函数调用写成 `std::io::stdin`。`stdin` 函数返回一个 `std::io::stdin` 的实例，这代表终端标准输入句柄的类型。

代码的下一部分，`.read_line(&mut guess)`，调用 `read_line` 方法从标准输入句柄获取用户输入。我们还向 `read_line()` 传递了一个参数：`&mut guess`。

`read_line` 的工作是，无论用户在标准输入中键入什么内容，都将其存入一个字符串中，因此它需要字符串作为参数。这个字符串参数应该是可变的，以便 `read_line` 将用户输入附加上去。

`&` 表示这个参数是一个 引用（*reference*），它允许多处代码访问同一处数据，而无需在内存中多次拷贝。引用是一个复杂的特性，**Rust** 的一个主要优势就是安全而简单的操纵引用。完成当前程序并不需要了解如此多细节：第四章会更全面的解释引用。现在，我们只需知道它像变量一样，默认是不可变的，需要写成 `&mut guess` 而不是 `&guess` 来使其可变。

我们还没有完全分析完这行代码。虽然这是单独一行代码，但它是一个逻辑行（虽然换行了但仍是一个语句）的第一部分。第二部分是这个方法：

```
.expect("Failed to read line");
```

当使用 `.foo()` 语法调用方法时，通过换行并缩进来把长行拆开是明智的。我们完全可以这样写：

```
io::stdin().read_line(&mut guess).expect("Failed to read line");
```

不过，过长的行难以阅读，所以最好拆开来写，两行代码两个方法调用。现在来看看这行代码干了什么。

使用 `Result` 类型来处理潜在的错误

之前提到了 `read_line` 将用户输入附加到传递给它的字符串中，不过它也返回一个值——在这个例子中是 `io::Result`。`Rust` 标准库中有很多叫做 `Result` 的类型。一个 `Result` 泛型以及对应子模块的特定版本，比如 `io::Result`。

`Result` 类型是 **枚举（enumerations）**，通常也写作 *enums*。枚举类型持有固定集合的值，这些值被称为枚举的成员（*variants*）。第六章将介绍枚举的更多细节。

对于 `Result`，它的成员是 `Ok` 或 `Err`，`Ok` 表示操作成功，内部包含成功时产生的值。`Err` 意味着操作失败，并且包含失败的前因后果。

这些 `Result` 类型的作用是编码错误处理信息。`Result` 类型的值，像其他类型一样，拥有定义于其上的方法。`io::Result` 的实例拥有 `expect` 方法。如果 `io::Result` 实例的值是 `Err`，`expect` 会导致程序崩溃，并显示当做参数传递给 `expect` 的信息。如果 `read_line` 方法返回 `Err`，则可能是来源于底层操作系统错误的结果。如果 `io::Result` 实例的值是 `Ok`，`expect` 会获取 `Ok` 中的值并原样返回。在本例中，这个值为用户输入到标准输入中的字节的数量。

如果不调用 `expect`，程序也能编译，不过会出现一个警告：

```
$ cargo build
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
warning: unused `std::result::Result` which must be used
--> src/main.rs:10:5
10 |     io::stdin().read_line(&mut guess);
    |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    = note: #[warn(unused_must_use)] on by default
```

`Rust` 警告我们没有使用 `read_line` 的返回值 `Result`，说明有一个可能的错误没有处理。消除警告的正确做法是实际编写错误处理代码，不过我们就是希望程序在出现问题时立即崩溃，所以直接使用 `expect`。第九章会学习如何从错误中恢复。

使用 `println!` 占位符打印值

除了位于结尾的大括号，目前为止就只有一行代码值得讨论一下了，就是这一行：

```
println!("You guessed: {}", guess);
```

这行代码打印存储用户输入的字符串。第一个参数是格式化字符串，里面的 `{}` 是预留在特定位置的占位符。使用 `{}` 也可以打印多个值：第一对 `{}` 使用格式化字符串之后的第一个值，第二对则使用第二个值，依此类推。调用一次 `println!` 打印多个值看起来像这样：

```
let x = 5;
let y = 10;

println!("x = {} and y = {}", x, y);
```

这行代码会打印出 `x = 5 and y = 10` 。

测试第一部分代码

让我们来测试下猜猜看游戏的第一部分。使用 `cargo run` 运行：

```
$ cargo run
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
  Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
  Running `target/debug/guessing_game`
Guess the number!
Please input your guess.
6
You guessed: 6
```

至此为止，游戏的第一部分已经完成：我们从键盘获取输入并打印了出来。

生成一个秘密数字

接下来，需要生成一个秘密数字，好让用户来猜。秘密数字应该每次都不同，这样重复玩才不会乏味；范围应该在 1 到 100 之间，这样才不会太困难。**Rust** 标准库中尚未包含随机数功能。然而，**Rust** 团队还是提供了一个 `rand` [crate](#)。

使用 **crate** 来增加更多功能

记住 **crate** 是一个 **Rust** 代码的包。我们正在构建的项目是一个 二进制 **crate**，它生成一个可执行文件。`rand` **crate** 是一个 库 **crate**，库 **crate** 可以包含任意能被其他程序使用的代码。

Cargo 对外部 **crate** 的运用是其真正闪光的地方。在我们使用 `rand` 编写代码之前，需要编辑 *Cargo.toml*，声明 `rand` 作为一个依赖。现在打开这个文件并在底部的 `[dependencies]` 部分标题之下添加：

文件名: *Cargo.toml*

```
[dependencies]

rand = "0.3.14"
```

在 *Cargo.toml* 文件中，标题以及之后的内容属同一个部分，直到遇到下一个标题才开始新的部分。`[dependencies]` 部分告诉 **Cargo** 本项目依赖了哪些外部 **crate** 及其版本。本例中，我们使用语义化版本 `0.3.14` 来指定 `rand` **crate**。**Cargo** 理解语义化版本（[Semantic Versioning](#)）（有时也称为 **SemVer**），这是一种定义版本号的标准。`0.3.14` 事实上是 `^0.3.14` 的简写，它表示“任何与 `0.3.14` 版本公有 API 相兼容的版本”。

现在，不修改任何代码，构建项目，如示例 2-2 所示：

```
$ cargo build
  Updating registry `https://github.com/rust-lang/crates.io-index`
  Downloading rand v0.3.14
  Downloading libc v0.2.14
  Compiling libc v0.2.14
  Compiling rand v0.3.14
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
  Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
```

示例 2-2: 增加 `rand crate` 作为依赖之后运行 `cargo build` 的输出

可能会出现不同的版本号（多亏了语义化版本，它们与代码是兼容的！），同时显示顺序也可能会有所不同。

现在我们有了一个外部依赖，Cargo 从 *registry* 上获取所有包的最新版本信息，这是一份来自 [Crates.io](https://crates.io) 的数据拷贝。Crates.io 是 Rust 生态环境中的开发者们向他人贡献 Rust 开源项目的地方。

在更新完 *registry* 后，Cargo 检查 `[dependencies]` 段落并下载缺失的部分。本例中，虽然只声明了 `rand` 一个依赖，然而 Cargo 还是额外获取了 `libc` 的拷贝，因为 `rand` 依赖 `libc` 来正常工作。下载完成后，Rust 编译依赖，然后使用这些依赖编译项目。

如果不做任何修改，立刻再次运行 `cargo build`，则不会有任何输出。Cargo 知道它已经下载并编译了依赖，同时 `Cargo.toml` 文件也没有变动。Cargo 还知道代码也没有任何修改，所以它不会重新编译代码。因为无事可做，它简单的退出了。如果打开 `src/main.rs` 文件，做一些无关紧要的修改，保存并再次构建，只会出现两行输出：

```
$ cargo build
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
```

这一行表示 Cargo 只针对 `src/main.rs` 文件的微小修改而更新构建。依赖没有变化，所以 Cargo 知道它可以复用已经为此下载并编译的代码。它只是重新构建了部分（项目）代码。

Cargo.lock 文件确保构建是可重现的

Cargo 有一个机制来确保任何人在任何时候重新构建代码，都会产生相同的结果：Cargo 只会使用你指定的依赖的版本，除非你又手动指定了别的。例如，如果下周 `rand crate` 的 `v0.3.15` 版本出来了，它修复了一个重要的 bug，同时也含有一个会破坏代码运行的缺陷，这时会发生什么呢？

这个问题的答案是 `Cargo.lock` 文件。它在第一次运行 `cargo build` 时创建，并放在 `guessing_game` 目录。当第一次构建项目时，Cargo 计算出所有符合要求的依赖版本并写入 `Cargo.lock` 文件。当将来构建项目时，Cargo 会发现 `Cargo.lock` 存在并使用其中指定的版本，而不是再次计算所有的版本。这使得你拥有了一个自动化的可重现的构建。换句话说，项目会持续使用 `0.3.14` 直到你显式升级，感谢 `Cargo.lock` 文件。

更新 crate 到一个新版本

当你确实需要升级 crate 时，Cargo 提供了另一个命令，`update`，他会：

1. 忽略 `Cargo.lock` 文件，并计算出所有符合 `Cargo.toml` 声明的最新版本。
2. 如果成功了，Cargo 会把这些版本写入 `Cargo.lock` 文件。

不过，Cargo 默认只会寻找大于 `0.3.0` 而小于 `0.4.0` 的版本。如果 `rand crate` 发布了两个新版本，`0.3.15` 和 `0.4.0`，在运行 `cargo update` 时会出现如下内容：

```
$ cargo update
Updating registry `https://github.com/rust-lang/crates.io-index`
Updating rand v0.3.14 -> v0.3.15
```

这时，你也会注意到的 `Cargo.lock` 文件中的变化无外乎 `rand crate` 现在使用的版本是 `0.3.15`

如果想要使用 `0.4.0` 版本的 `rand` 或是任何 `0.4.x` 系列的版本，必须像这样更新 `Cargo.toml` 文件：

```
[dependencies]

rand = "0.4.0"
```

下一次运行 `cargo build` 时，Cargo 会从 registry 更新可用的 crate，并根据你指定的新版本重新计算。

第十四章会讲到 Cargo 及其生态系统的更多内容，不过目前你只需要了解这么多。通过 Cargo 复用库文件非常容易，因此 Rustacean 能够编写出由很多包组装而成的更轻巧的项目。

生成一个随机数

让我们开始使用 `rand`。下一步是更新 `src/main.rs`，如示例 2-3 所示：

文件名: `src/main.rs`

```
extern crate rand;

use std::io;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {}", guess);
}
```

示例 2-3：为了生成随机数而做的修改

这里在顶部增加一行 `extern crate rand;` 通知 Rust 我们要使用外部依赖。这也会调用相应的 `use rand`，所以现在可以使用 `rand::` 前缀来调用 `rand crate` 中的任何内容。

接下来增加了另一行 `use : use rand::Rng`。`Rng` 是一个 trait，它定义了随机数生成器应实现的方法，想使用这些方法的话此 trait 必须在作用域中。第十章会详细介绍 trait。

另外，中间还新增加了两行。`rand::thread_rng` 函数提供实际使用的随机数生成器：它位于当前执行线程本地，并从操作系统获取 seed。接下来，调用随机数生成器的 `gen_range` 方法。这个方法由刚才引入到作用域的 `Rng trait` 定义。`gen_range` 方法获取两个数字作为参数，并生成一个范围在两者之间的随机数。它包含下限但不包含上限，所以需要指定 `1` 和 `101` 来请求一个 `1` 和 `100` 之间的数。

知道 `use` 哪个 trait 和该从 `crate` 中调用哪个方法并不是你唯一会知道的。`crate` 的使用说明位于其文档中。

Cargo 有一个很棒的功能是：运行 `cargo doc --open` 命令来构建所有本地依赖提供的文档，并在浏览器中打开。例如，假设你对 `rand crate` 中的其他功能感兴趣，`cargo doc --open` 并点击左侧导航栏中的 `rand`。

新增加的第二行代码打印出了秘密数字。这在开发程序时很有用，因为可以测试它，不过在最终版本中会删掉它。游戏一开始就打印出结果就没什么好玩的了！

尝试运行程序几次：

```
$ cargo run
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
Running `target/debug/guessing_game`
```

```

Guess the number!
The secret number is: 7
Please input your guess.
4
You guessed: 4
$ cargo run
    Running `target/debug/guessing_game`
Guess the number!
The secret number is: 83
Please input your guess.
5
You guessed: 5

```

你应该能得到不同的随机数，同时它们应该都是在 1 和 100 之间的。干得漂亮！

比较猜测与秘密数字

现在有了用户输入和一个随机数，我们可以比较它们。这个步骤如示例 2-4 所示：

文件名: `src/main.rs`

```

extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {}", guess);

    match guess.cmp(&secret_number) {
        Ordering::Less => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal => println!("You win!"),
    }
}

```

示例 2-4：处理比较两个数字可能的返回值

新代码的第一行是另一个 `use`，从标准库引入了一个叫做 `std::cmp::Ordering` 的类型。`Ordering` 是一个像 `Result` 一样的枚举，不过它的成员是 `Less`、`Greater` 和 `Equal`。这是比较两个值时可能出现的三种结果。

接着，底部的五行新代码使用了 `Ordering` 类型：

```

match guess.cmp(&secret_number) {
    Ordering::Less => println!("Too small!"),
    Ordering::Greater => println!("Too big!"),

```

```
    Ordering::Equal => println!("You win!"),
}
```

`cmp` 方法用来比较两个值并可以在任何可比较的值上调用。它获取一个被比较值的引用：这里是把 `guess` 与 `secret_number` 做比较。`cmp` 返回一个刚才通过 `use` 引入作用域的 `Ordering` 枚举的成员。使用一个 `match` 表达式，根据对 `guess` 和 `secret_number` 调用 `cmp` 返回的 `Ordering` 成员来决定接下来做什么。

一个 `match` 表达式由分支（**arms**）构成。一个分支包含一个模式（*pattern*）和表达式开头的值与分支模式相匹配时应该执行的代码。`Rust` 获取提供给 `match` 的值并挨个检查每个分支的模式。`match` 结构和模式是 `Rust` 中强大的功能，它体现了代码可能遇到的多种情形，并帮助你确保没有遗漏处理。这些功能将分别在第六章和第十八章详细介绍。

让我们看看使用 `match` 表达式的例子。假设用户猜了 `50`，这时随机生成的秘密数字是 `38`。比较 `50` 与 `38` 时，因为 `50` 比 `38` 要大，`cmp` 方法会返回 `Ordering::Greater`。`Ordering::Greater` 是 `match` 表达式得到的值。它检查第一个分支的模式，`Ordering::Less` 与 `Ordering::Greater` 并不匹配，所以它忽略了这个分支的动作并来到下一个分支。下一个分支的模式是 `Ordering::Greater`，正确匹配！这个分支关联的代码被执行，在屏幕打印出 `Too big!`。`match` 表达式就此终止，因为该场景下没有检查最后一个分支的必要。

然而，示例 2-4 的代码并不能编译，可以尝试一下：

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
error[E0308]: mismatched types
  --> src/main.rs:23:21
   |
23 |     match guess.cmp(&secret_number) {
   |                     ^^^^^^^^^^^^^^^ expected struct `std::string::String`, found integral variable
   |
   = note: expected type `&std::string::String`
   = note:   found type `&{integer}`

error: aborting due to previous error
Could not compile `guessing_game`.
```

错误的核心表明这里有不匹配的类型（*mismatched types*）。`Rust` 有一个静态强类型系统，同时也有类型推断。当我们写出 `let guess = String::new()` 时，`Rust` 推断出 `guess` 应该是一个 `String`，不需要我们写出的类型。另一方面，`secret_number`，是一个数字类型。多种数字类型拥有 1 到 100 之间的值：32 位数字 `i32`；32 位无符号数字 `u32`；64 位数字 `i64` 等等。`Rust` 默认使用 `i32`，所以它是 `secret_number` 的类型，除非增加类型信息，或任何能让 `Rust` 推断出不同数值类型的信息。这里错误的原因在于 `Rust` 不会比较字符串类型和数字类型。

所以必须把从输入中读取到的 `String` 转换为一个真正的数字类型，才好与秘密数字进行比较。这可以通过在 `main` 函数体中增加如下两行代码来实现：

文件名: `src/main.rs`

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);
}
```

```
println!("Please input your guess.");

let mut guess = String::new();

io::stdin().read_line(&mut guess)
    .expect("Failed to read line");

let guess: u32 = guess.trim().parse()
    .expect("Please type a number!");

println!("You guessed: {}", guess);

match guess.cmp(&secret_number) {
    Ordering::Less => println!("Too small!"),
    Ordering::Greater => println!("Too big!"),
    Ordering::Equal => println!("You win!"),
}
}
```

这两行新代码是：

```
let guess: u32 = guess.trim().parse()
    .expect("Please type a number!");
```

这里创建了一个叫做 `guess` 的变量。不过等等，不是已经有了一个叫做 `guess` 的变量了吗？确实如此，不过 Rust 允许 隐藏（*shadow*），用一个新值来隐藏 `guess` 之前的值。这个功能常用在需要转换值类型之类的场景，它允许我们复用 `guess` 变量的名字，而不是被迫创建两个不同变量，诸如 `guess_str` 和 `guess` 之类。（第三章会介绍 *shadowing* 的更多细节。）

`guess` 被绑定到 `guess.trim().parse()` 表达式。表达式中的 `guess` 是包含输入的原始 `String` 类型。`String` 实例的 `trim` 方法会去除字符串开头和结尾的空白。`u32` 只能由数字字符转换，不过用户必须输入 `return` 键才能让 `read_line` 返回，然而用户按下 `return` 键时，会在字符串中增加一个换行（*newline*）符。例如，用户输入 5 并按下 `return`，`guess` 看起来像这样： `5\n`。`\n` 代表“换行”，回车键。`trim` 方法消除 `\n`，只留下 5。

字符串的 `parse` 方法将字符串解析成数字。因为这个方法可以解析多种数字类型，因此需要告诉 Rust 具体的数字类型，这里通过 `let guess: u32` 指定。`guess` 后面的冒号（`:`）告诉 Rust 我们指定了变量的类型。Rust 有一些内建的数字类型；`u32` 是一个无符号的 32 位整型。对于不大的正整数来说，它是不错的类型，第三章还会讲到其他数字类型。另外，程序中的 `u32` 注解以及与 `secret_number` 的比较，意味着 Rust 会推断出 `secret_number` 也是 `u32` 类型。现在可以使用相同类型比较两个值了！

`parse` 调用很容易产生错误。例如，字符串中包含 `A6%`，就无法将其转换为一个数字。因此，`parse` 方法返回一个 `Result` 类型。像之前“使用 `Result` 类型来处理潜在的错误”讨论的 `read_line` 方法那样，再次按部就班的用 `expect` 方法处理即可。如果 `parse` 不能从字符串生成一个数字，返回一个 `Result::Err` 时，`expect` 会使游戏崩溃并打印附带的信息。如果 `parse` 成功地将字符串转换为一个数字，它会返回 `Result::Ok`，然后 `expect` 会返回 `Ok` 中的数字。

现在让我们运行程序！

```
$ cargo run
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
Finished dev [unoptimized + debuginfo] target(s) in 0.43 secs
Running `target/guessing_game`
Guess the number!
The secret number is: 58
Please input your guess.
76
You guessed: 76
Too big!
```


漂亮！即便是在猜测之前添加了空格，程序依然能判断出用户猜测了 **76**。多运行程序几次来检验不同类型输入的相应行为：猜一个正确的数字，猜一个过大的数字和猜一个过小的数字。

现在游戏已经大体上能玩了，不过用户只能猜一次。增加一个循环来改变它吧！

使用循环来允许多次猜测

`loop` 关键字提供了一个无限循环。将其加入后，用户可以反复猜测：

文件名: `src/main.rs`

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .expect("Failed to read line");

        let guess: u32 = guess.trim().parse()
            .expect("Please type a number!");

        println!("You guessed: {}", guess);

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => println!("You win!"),
        }
    }
}
```

如上所示，我们将提示用户猜测之后的所有内容放入了循环。确保这些代码额外缩进了一层，再次运行程序。注意这里有一个新问题，因为程序忠实地执行了我们的要求：永远地请求另一个猜测，用户好像没法退出啊！

用户总能使用 **ctrl-C** 终止程序。不过还有另一个方法跳出无限循环，就是“比较猜测与秘密数字”部分提到的

`parse`：如果用户输入一个非数字答案，程序会崩溃。用户可以利用这一点来退出，如下所示：

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
   Running `target/guessing_game`
Guess the number!
The secret number is: 59
Please input your guess.
45
You guessed: 45
```

```

Too small!
Please input your guess.
60
You guessed: 60
Too big!
Please input your guess.
59
You guessed: 59
You win!
Please input your guess.
quit
thread 'main' panicked at 'Please type a number!: ParseIntError { kind: InvalidDigit }', src/libcore/result.rs:785
note: Run with `RUST_BACKTRACE=1` for a backtrace.
error: Process didn't exit successfully: `target/debug/guess` (exit code: 101)

```

输入 `quit` 确实退出了程序，同时其他任何非数字输入也一样。然而，这并不理想，我们想要当猜测正确的数字时游戏能自动退出。

猜测正确后退出

让我们增加一个 `break`，在用户猜对时退出游戏：

文件名: `src/main.rs`

```

extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .expect("Failed to read line");

        let guess: u32 = guess.trim().parse()
            .expect("Please type a number!");

        println!("You guessed: {}", guess);

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
                println!("You win!");
                break;
            }
        }
    }
}

```

通过在 `You win!` 之后增加一行 `break`，用户猜对了神秘数字后会退出循环。退出循环也意味着退出程序，因为循环是 `main` 的最后一部分。

处理无效输入

为了进一步改善游戏性，不要在用户输入非数字时崩溃，需要忽略非数字，让用户可以继续猜测。可以通过修改 `guess` 将 `String` 转化为 `u32` 那部分代码来实现：

```
let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
};
```

将 `expect` 调用换成 `match` 语句，是从遇到错误就崩溃转换到真正处理错误的惯用方法。须知 `parse` 返回一个 `Result` 类型，而 `Result` 是一个拥有 `Ok` 或 `Err` 成员的枚举。这里使用的 `match` 表达式，和之前处理 `cmp` 方法返回 `Ordering` 时用的样子。

如果 `parse` 能够成功的将字符串转换为一个数字，它会返回一个包含结果数字的 `Ok`。这个 `Ok` 值与 `match` 第一个分支的模式相匹配，该分支对应的动作返回 `Ok` 值中的数字 `num`，最后如愿变成新创建的 `guess` 变量。

如果 `parse` 不能将字符串转换为一个数字，它会返回一个包含更多错误信息的 `Err`。 `Err` 值不能匹配第一个 `match` 分支的 `Ok(num)` 模式，但是会匹配第二个分支的 `Err(_)` 模式： `_` 是一个通配符值，本例中用来匹配所有 `Err` 值，不管其中有何种信息。所以程序会执行第二个分支的动作， `continue` 意味着进入 `loop` 的下一循环，请求另一个猜测。这样程序就有效的忽略了 `parse` 可能遇到的所有错误！

现在万事俱备，只需运行 `cargo run`：

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
   Running `target/guessing_game`
Guess the number!
The secret number is: 61
Please input your guess.
10
You guessed: 10
Too small!
Please input your guess.
99
You guessed: 99
Too big!
Please input your guess.
foo
Please input your guess.
61
You guessed: 61
You win!
```

太棒了！再有最后一个小的修改，就能完成猜数字游戏了：还记得程序依然会打印出秘密数字。在测试时还好，但正式发布时会毁了游戏。删掉打印秘密数字的 `println!`。示例 2-5 为最终代码：

文件名: `src/main.rs`

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;
```

```

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .expect("Failed to read line");

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        println!("You guessed: {}", guess);

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
                println!("You win!");
                break;
            }
        }
    }
}

```

示例 2-5: 猜猜看游戏的完整代码

总结

此时此刻，你顺利完成了猜猜看游戏！恭喜！

这是一个通过动手实践学习 **Rust** 新概念的项目：`let`、`match`、方法、关联函数、使用外部 `crate` 等等，接下来的几章，我们将会继续深入。第三章涉及到大部分编程语言都有的概念，比如变量、数据类型和函数，以及如何在 **Rust** 中使用它们。第四章探索所有权（ownership），这是一个 **Rust** 同其他语言大不相同的功能。第五章讨论结构体和方法的语法，而第六章侧重解释枚举。

通用编程概念

[ch03-00-common-programming-concepts.md](#)

commit 04aa3a45eb72855b34213703718f50a12a3eeec8

本章涉及一些几乎所有编程语言都有的概念，以及它们在 **Rust** 中是如何工作的。很多编程语言的核心概念都是共通的，本章中展示的概念都不是 **Rust** 所特有的，不过我们会在 **Rust** 环境中讨论它们，解释它们的使用习惯。

具体地，我们将会学习变量，基本类型，函数，注释和控制流。这些基础知识将会出现在每一个 **Rust** 程序中，提早学习这些概念会为你奠定坚实的起步基础。

关键字

Rust 语言有一系列保留的 关键字（*keywords*），就像大部分语言一样，它们只能由语言本身使用，你不能使用这些关键字作为变量或函数的名称。大部分关键字有特殊的意义，并被用来完成 **Rust** 程序中的各种任务；一些关键字目前没有相应的功能，是为将来可能添加的功能保留的。可以在附录 A 中找到关键字的列表。

变量和可变性

ch03-01-variables-and-mutability.md

commit 6aad5008b69078a2fc18e6dd7e00ef395170c749

第二章中提到过，变量默认是 不可变（*immutable*）的。这是鼓励你利用 Rust 安全和简单并发的优势来编写代码的一大助力。不过，你仍然可以使用可变变量。让我们探讨一下 Rust 拥抱不可变性的原因及方法，以及何时你不想使用不可变性。

当变量不可变时，意味着一旦值被绑定上一个名称，你就不能改变这个值。作为说明，通过 `cargo new --bin variables` 在 `projects` 目录生成一个叫做 `variables` 的新项目。

接着，在新建的 `variables` 目录，打开 `src/main.rs` 并替换其代码为如下：

文件名: `src/main.rs`

```
fn main() {
    let x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```

保存并使用 `cargo run` 运行程序。应该会看到一个错误信息，如下输出所示：

```
error[E0384]: cannot assign twice to immutable variable `x`
--> src/main.rs:4:5
 |
2 |     let x = 5;
 |         - first assignment to `x`
3 |     println!("The value of x is: {}", x);
4 |     x = 6;
 |     ^^^^^ cannot assign twice to immutable variable
```

这个例子展示了编译器如何帮助你找出程序中的错误。虽然编译错误令人沮丧，那也不过是说程序不能安全的完成你想让它完成的工作；而 不能 说明你是不是一个好程序员！有经验的 Rustacean 们一样会遇到编译错误。

这些错误给出的原因是 不能对不可变变量二次赋值（`cannot assign twice to immutable variable x`），因为我们尝试对不可变变量 `x` 赋第二个值。

在尝试改变预设为不可变的值的时候产生编译错误是很重要的，因为这种情况可能导致 **bug**：如果代码的一部分假设一个值永远也不会改变，而另一部分代码改变了它，第一部分代码就有可能以不可预料的方式运行。不得不承认这种 **bug** 难以跟踪，尤其是第二部分代码只是 有时 改变其值的时候。

Rust 编译器保证，如果声明一个值不会变，它就真的不会变。这意味着当阅读和编写代码时，不需要追踪一个值如何以及哪里可能会被改变，从而使得代码易于推导。

不过可变性也是非常有用的。变量只是默认不可变，可以通过在变量名之前加 `mut` 来使其可变。除了使值可以改变之外，它向读者表明了其他代码将会改变这个变量的意图。

例如，改变 `src/main.rs` 并替换其代码为如下：

文件名: `src/main.rs`

```
fn main() {
```

```
let mut x = 5;
println!("The value of x is: {}", x);
x = 6;
println!("The value of x is: {}", x);
}
```

当运行这个程序，出现如下：

```
$ cargo run
Compiling variables v0.1.0 (file:///projects/variables)
Finished dev [unoptimized + debuginfo] target(s) in 0.30 secs
Running `target/debug/variables`
The value of x is: 5
The value of x is: 6
```

通过 `mut`，允许把绑定到 `x` 的值从 `5` 改成 `6`。在一些情况下，你会想用可变量，因为这样的代码比起只用不可变量的实现更容易编写。

除了避免 `bug` 外，还有很多地方需要权衡取舍。例如，使用大型数据结构时，适当地使用可变量，可能比复制和返回新分配的实例更快。对于较小的数据结构，总是创建新实例，采用更偏向函数式的风格编程，可能会使代码更易理解，为可读性而遭受性能惩罚或许值得。

变量和常量的区别

不允许改变值的变量，可能会使你想起另一个大部分编程语言都有的概念：常量（*constants*）。类似于不可变量，常量也是绑定到一个名称的不允许改变的值，不过常量与变量还是有一些区别。

首先，不允许对常量使用 `mut`：常量不光默认不能变，它总是不能变。

声明常量使用 `const` 关键字而不是 `let`，并且必须注明值的类型。在下一部分，“数据类型”中会涉及到类型和类型注解，现在无需关心这些细节，记住总是标注类型即可。

常量可以在任何作用域声明，包括全局作用域，这在一个值需要被很多部分的代码用到时很有用。

最后一个区别是常量只能用于常量表达式，而不能作为函数调用的结果，或任何其他只在运行时计算的值。

这是一个常量声明的例子，它的名称是 `MAX_POINTS`，值是 `100,000`。（`Rust` 的常量使用下划线分隔的大写字母的命名规范）：

```
const MAX_POINTS: u32 = 100_000;
```

在声明它的作用域之中，常量在整个程序生命周期中都有效，这使得常量可以作为多处代码使用的全局范围的值，例如一个游戏中所有玩家可以获取的最高分或者光速。

将用于整个程序的硬编码的值声明为常量对后来的维护者了解值的意义很有帮助。同时将硬编码的值汇总于一处，也能为将来修改提供方便。

隐藏（Shadowing）

如第二章“猜猜看游戏”所讲的，我们可以定义一个与之前变量重名的新变量，而新变量会隐藏之前的变量。

`Rustacean` 们称之为第一个变量被第二个隐藏了，这意味着使用这个变量时会看到第二个值。可以用相同变量名称来隐藏它自己，以及重复使用 `let` 关键字来多次隐藏，如下所示：

文件名: `src/main.rs`

```
fn main() {
    let x = 5;

    let x = x + 1;

    let x = x * 2;

    println!("The value of x is: {}", x);
}
```

这个程序首先将 `x` 绑定到值 `5` 上。接着通过 `let x =` 隐藏 `x`，获取原始值并加 `1` 这样 `x` 的值就变成 `6` 了。第三个 `let` 语句也隐藏了 `x`，获取之前的值并乘以 `2`，`x` 最终的值是 `12`。运行这个程序，它会有如下输出：

```
$ cargo run
Compiling variables v0.1.0 (file:///projects/variables)
Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
Running `target/debug/variables`
The value of x is: 12
```

这与将变量声明为 `mut` 是有区别的。因为除非再次使用 `let` 关键字，不小心尝试对变量重新赋值会导致编译时错误。我们可以用这个值进行一些计算，不过计算完之后变量仍然是不变的。

`mut` 与隐藏的另一个区别是，当再次使用 `let` 时，实际上创建了一个新变量，我们可以改变值的类型，从而复用这个名字。例如，假设程序请求用户输入空格来提供文本间隔，然而我们真正需要的是将输入存储成数字（多少个空格）：

```
let spaces = " ";
let spaces = spaces.len();
```

这里允许第一个 `spaces` 变量是字符串类型，而第二个 `spaces` 变量，它是一个恰巧与第一个变量同名的崭新变量，是数字类型。隐藏使我们不必使用不同的名字，如 `spaces_str` 和 `spaces_num`；相反，我们可以复用 `spaces` 这个更简单的名字。然而，如果尝试使用 `mut`，如下所示：

```
let mut spaces = " ";
spaces = spaces.len();
```

会导致一个编译错误，因为改变一个变量的类型是不被允许的：

```
error[E0308]: mismatched types
--> src/main.rs:3:14
|
3 |     spaces = spaces.len();
|               ^^^^^^^^^^^ expected &str, found usize
|
= note: expected type `&str`
       found type `usize`
```

现在我们已经了解了变量如何工作，让我们再看看更多变量可以拥有的数据类型。

数据类型

ch03-02-data-types.md

commit ec65990849230388e4ce4db5b7a0cb8a0f0d60e2

在 **Rust** 中，任何值都属于一种明确的 类型（*type*），这告诉了 **Rust** 它被指定为何种数据，以便明确其处理方式。本部分我们将看到一系列内建于语言中的类型。我们将其分为两类：标量（*scalar*）和复合（*compound*）。

Rust 是 静态类型（*statically typed*）语言，也就是说在编译时就必须知道所有变量的类型，这一点将贯穿整个章节。通过值的形式及其使用方式，编译器通常可以推断出我们想要用的类型。多种类型均有可能时，比如第二章中使用 `parse` 将 `String` 转换为数字时，必须增加类型注解，像这样：

```
let guess: u32 = "42".parse().expect("Not a number!");
```

这里如果不添加类型注解，**Rust** 会显示如下错误，这说明编译器需要更多信息，来了解我们想要的类型：

```
error[E0282]: type annotations needed
--> src/main.rs:2:9
  |
2 |     let guess = "42".parse().expect("Not a number!");
  |     ^^^^^
  |     |
  |     cannot infer type for `_`
  |     consider giving `guess` a type
```

在我们讨论各种数据类型时，你会看到不同的类型注解。

标量类型

标量（*scalar*）类型代表一个单独的值。**Rust** 有四种基本的标量类型：整型、浮点型、布尔类型和字符类型。你可能在其他语言中见过它们，不过让我们深入了解它们在 **Rust** 中是如何工作的。

整型

整数 是一个没有小数部分的数字。我们在这一章的前面使用过 `u32` 类型。该类型声明表明，`u32` 关联的值应该是一个占据 32 比特位的无符号整数（有符号整型类型以 `i` 开头而不是 `u`）。表格 3-1 展示了 **Rust** 内建的整数类型。每一种变体都有符号和无符号列（例如，`i8`）可以用来声明对应的整数值。

表格 3-1: **Rust** 中的整型

长度	有符号	无符号
8-bit	<code>i8</code>	<code>u8</code>
16-bit	<code>i16</code>	<code>u16</code>
32-bit	<code>i32</code>	<code>u32</code>
64-bit	<code>i64</code>	<code>u64</code>
arch	<code>isize</code>	<code>usize</code>

每一种变体都可以是有符号或无符号的，并有一个明确的大小。有符号和无符号代表数字能否为负值；换句话说，数字是否需要有一个符号（有符号数），或者永远为正而不需要符号（无符号数）。这有点像在纸上书写数字：当需要考虑符号的时候，数字以加号或减号作为前缀；然而，可以安全地假设为正数时，加号前缀通常省略。有符号数以二进制补码形式（two's complement representation）存储（如果你不清楚这是什么，可以在网上搜索；对其的解释超出了本书的范畴）。

每一个有符号的变体可以储存包含从 $-(2^{n-1})$ 到 $2^{n-1} - 1$ 在内的数字，这里 n 是变体使用的位数。所以 `i8` 可以储存从 $-(2^7)$ 到 $2^7 - 1$ 在内的数字，也就是从 -128 到 127。无符号的变体可以储存从 0 到 $2^n - 1$ 的数字，所以 `u8` 可以储存从 0 到 $2^8 - 1$ 的数字，也就是从 0 到 255。

另外，`isize` 和 `usize` 类型依赖运行程序的计算机架构：64 位架构上它们是 64 位的，32 位架构上它们是 32 位的。

可以使用表格 3-2 中的任何一种形式编写数字字面值。注意除 `byte` 以外的其它字面值允许使用类型后缀，例如 `57u8`，同时也允许使用 `_` 做为分隔符以方便读数，例如 `1_000`。

表格 3-2: Rust 中的整型字面值

数字字面值	例子
Decimal	<code>98_222</code>
Hex	<code>0xff</code>
Octal	<code>0o77</code>
Binary	<code>0b1111_0000</code>
Byte (<code>u8</code> only)	<code>b'A'</code>

那么该使用哪种类型的数字呢？如果拿不定主意，`Rust` 的默认类型通常就很好，数字类型默认是 `i32`：它通常是最快的，甚至在 64 位系统上也是。`isize` 或 `usize` 主要作为某些集合的索引。

浮点型

`Rust` 同样有两个主要的浮点数（*floating-point numbers*）类型，`f32` 和 `f64`，它们是带小数点的数字，分别占 32 位和 64 位比特。默认类型是 `f64`，因为在现代 CPU 中它与 `f32` 速度几乎一样，不过精度更高。

这是一个展示浮点数的实例：

文件名: `src/main.rs`

```
fn main() {
    let x = 2.0; // f64

    let y: f32 = 3.0; // f32
}
```

浮点数采用 IEEE-754 标准表示。`f32` 是单精度浮点数，`f64` 是双精度浮点数。

数字运算符

`Rust` 支持所有数字类型常见的基本数学运算操作：加法、减法、乘法、除法以及取余。下面的代码展示了如何在一个 `let` 语句中使用它们：

文件名: `src/main.rs`

```
fn main() {
    // addition
    let sum = 5 + 10;

    // subtraction
    let difference = 95.5 - 4.3;

    // multiplication
    let product = 4 * 30;

    // division
    let quotient = 56.7 / 32.2;

    // remainder
    let remainder = 43 % 5;
}
```

这些语句中的每个表达式使用了一个数学运算符并计算出了一个值，它们绑定到了一个变量。附录 B 包含了一个 Rust 提供的所有运算符的列表。

布尔型

正如其他大部分编程语言一样，Rust 中的布尔类型有两个可能的值：`true` 和 `false`。Rust 中的布尔类型使用 `bool` 表示。例如：

文件名: `src/main.rs`

```
fn main() {
    let t = true;

    let f: bool = false; // with explicit type annotation
}
```

使用布尔值的主要场景是条件表达式，例如 `if` 表达式。在“控制流”（“Control Flow”）部分将讲到 `if` 表达式在 Rust 中如何工作。

字符类型

目前为止只使用到了数字，不过 Rust 也支持字符。Rust 的 `char` 类型是大部分语言中基本字母字符类型，如下代码展示了如何使用它。注意 `char` 由单引号指定，不同于字符串使用双引号：

文件名: `src/main.rs`

```
fn main() {
    let c = 'z';
    let z = 'Z';
    let heart_eyed_cat = '😺';
}
```

Rust 的 `char` 类型代表了一个 Unicode 标量值（Unicode Scalar Value），这意味着它可以比 ASCII 表示更多内容。拼音字母（Accented letters），中文/日文/韩文等象形文字，emoji（绘文字）以及零长度的空白字符对于 Rust `char` 类型都是有效的。Unicode 标量值包含从 `U+0000` 到 `U+D7FF` 和 `U+E000` 到 `U+10FFFF` 在内的值。不过，“字符”并不是一个 Unicode 中的概念，所以人直觉上的“字符”可能与 Rust 中的 `char` 并不符合。第八章的“字符串”部分将详细讨论这个主题。

复合类型

复合类型（*Compound types*）可以将多个其他类型的值组合成一个类型。**Rust** 有两个原生的复合类型：元组（*tuple*）和数组（*array*）。

将值组合进元组

元组是一个将多个其他类型的值组合进一个复合类型的主要方式。

我们使用一个括号中的逗号分隔的值列表来创建一个元组。元组中的每一个位置都有一个类型，而且这些不同值的类型也不必是相同的。这个例子中使用了额外的可选类型注解：

文件名: `src/main.rs`

```
fn main() {  
    let tup: (i32, f64, u8) = (500, 6.4, 1);  
}
```

`tup` 变量绑定了整个元组，因为元组被认为是一个单独的复合元素。为了从元组中获取单个的值，可以使用模式匹配（*pattern matching*）来解构（*destructure*）元组，像这样：

文件名: `src/main.rs`

```
fn main() {  
    let tup = (500, 6.4, 1);  
  
    let (x, y, z) = tup;  
  
    println!("The value of y is: {}", y);  
}
```

程序首先创建了一个元组并绑定到 `tup` 变量上。接着使用了 `let` 和一个模式将 `tup` 分成了三个不同的变量，`x`、`y` 和 `z`。这叫做 解构（*destructuring*），因为它将一个元组拆成了三个部分。最后，程序打印出了 `y` 的值，也就是 `6.4`。

除了使用模式匹配解构之外，也可以使用点号（`.`）后跟值的索引来直接访问它们。例如：

文件名: `src/main.rs`

```
fn main() {  
    let x: (i32, f64, u8) = (500, 6.4, 1);  
  
    let five_hundred = x.0;  
  
    let six_point_four = x.1;  
  
    let one = x.2;  
}
```

这个程序创建了一个元组，`x`，并接着使用索引为每个元素创建新变量。跟大多数编程语言一样，元组的第一个索引值是 `0`。

数组

另一个获取一个多个值集合的方式是 数组（*array*）。与元组不同，数组中的每个元素的类型必须相同。**Rust** 中的数组与一些其他语言中的数组不同，因为 **Rust** 中的数组是固定长度的：一旦声明，它们的长度不能增长或缩小。

Rust 中数组的值位于中括号中的逗号分隔的列表中：

文件名: src/main.rs

```
fn main() {  
    let a = [1, 2, 3, 4, 5];  
}
```

当你想要在栈（**stack**）而不是在堆（**heap**）上为数据分配空间（第四章将讨论栈与堆的更多内容），或者是想要确保总是有固定数量的元素时，数组非常有用，虽然它并不如 **vector** 类型那么灵活。**vector** 类型是标准库提供的一个允许增长和缩小长度的类似数组的集合类型。当不确定是应该使用数组还是 **vector** 的时候，你可能应该使用 **vector**。第八章会详细讨论 **vector**。

一个你可能想要使用数组而不是 **vector** 的例子是，当程序需要知道一年中月份的名字时，程序不大可能会去增加或减少月份。这时你可以使用数组，因为我们知道它总是含有 12 个元素：

```
let months = ["January", "February", "March", "April", "May", "June", "July",  
              "August", "September", "October", "November", "December"];
```

访问数组元素

数组是一整块分配在栈上的内存。可以使用索引来访问数组的元素，像这样：

文件名: src/main.rs

```
fn main() {  
    let a = [1, 2, 3, 4, 5];  
  
    let first = a[0];  
    let second = a[1];  
}
```

在这个例子中，叫做 `first` 的变量的值是 `1`，因为它是数组索引 `[0]` 的值。变量 `second` 将会是数组索引 `[1]` 的值 `2`。

无效的数组元素访问

如果我们访问数组结尾之后的元素会发生什么呢？比如我们将上面的例子改成下面这样，这可以编译不过在运行时会因为错误而退出：

文件名: src/main.rs

```
fn main() {  
    let a = [1, 2, 3, 4, 5];  
    let index = 10;  
  
    let element = a[index];  
  
    println!("The value of element is: {}", element);  
}
```

使用 `cargo run` 运行代码后会产生如下结果：

```
$ cargo run  
Compiling arrays v0.1.0 (file:///projects/arrays)  
Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
```

```
Running `target/debug/arrays`  
thread '<main>' panicked at 'index out of bounds: the len is 5 but the index is  
10', src/main.rs:6  
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

编译并没有产生任何错误，不过程序会导致一个运行时（*runtime*）错误并且不会成功退出。当尝试用索引访问一个元素时，**Rust** 会检查指定的索引是否小于数组的长度。如果索引超出了数组长度，**Rust** 会 *panic*，这是 **Rust** 中的术语，它用于程序因为错误而退出的情况。

这是第一个在实战中遇到的 **Rust** 安全原则的例子。在很多底层语言中，并没有进行这类检查，这样当提供了一个不正确的索引时，就会访问无效的内存。**Rust** 通过立即退出而不是允许内存访问并继续执行来使你免受这类错误困扰。第九章会讨论更多 **Rust** 的错误处理。

函数如何工作

ch03-03-how-functions-work.md

commit 6aad5008b69078a2fc18e6dd7e00ef395170c749

函数在 **Rust** 代码中应用广泛。你已经见过一个语言中最重要的函数：`main` 函数，它是很多程序的入口点。你也见过了 `fn` 关键字，它用来声明新函数。

Rust 代码使用 *snake case* 作为函数和变量名称的规范风格。在 *snake case* 中，所有字母都是小写并使用下划线分隔单词。这里是一个包含函数定义的程序的例子：

文件名: `src/main.rs`

```
fn main() {  
    println!("Hello, world!");  
  
    another_function();  
}  
  
fn another_function() {  
    println!("Another function.");  
}
```

Rust 中的函数定义以 `fn` 开始并在函数名后跟一对括号。大括号告诉编译器哪里是函数体的开始和结尾。

可以使用定义过的函数名后跟括号来调用任意函数。因为 `another_function` 已经在程序中定义过了，它可以在 `main` 函数中被调用。注意，源码中 `another_function` 在 `main` 函数之后被定义；也可以在其之前定义。**Rust** 不关心函数定义于何处，只要它们被定义了。

让我们开始一个叫做 *functions* 的新二进制项目来进一步探索函数。将上面的 `another_function` 例子写入 `src/main.rs` 中并运行。你应该会看到如下输出：

```
$ cargo run  
Compiling functions v0.1.0 (file:///projects/functions)  
Finished dev [unoptimized + debuginfo] target(s) in 0.28 secs  
Running `target/debug/functions`  
Hello, world!  
Another function.
```

代码在 `main` 函数中按照它们出现的顺序被执行。首先，打印“Hello, world!”信息，接着 `another_function` 被调用并打印它的信息。

函数参数

函数也可以被定义为拥有参数（*parameters*），它们是作为函数签名一部分的特殊变量。当函数拥有参数时，可以为这些参数提供具体的值。技术上讲，这些具体值被称为参数（*arguments*），不过通常的习惯是倾向于在函数定义中的变量和调用函数时传递的具体值都可以用“parameter”和“argument”而不加区别。

如下被重写的 `another_function` 版本展示了 **Rust** 中参数是什么样的：

文件名: `src/main.rs`

```
fn main() {  
    another_function(5);
```



```

}

fn another_function(x: i32) {
    println!("The value of x is: {}", x);
}

```

尝试运行程序，将会得到如下输出：

```

$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
   Finished dev [unoptimized + debuginfo] target(s) in 1.21 secs
    Running `target/debug/functions`
The value of x is: 5

```

`another_function` 的声明有一个叫做 `x` 的参数。`x` 的类型被指定为 `i32`。当 `5` 被传递给 `another_function` 时，`println!` 宏将 `5` 放入格式化字符串中大括号的位置。

在函数签名中 必须 声明每个参数的类型。这是 Rust 设计中一个经过慎重考虑的决定：要求在函数定义中提供类型注解意味着编译器不需要在别的地方要求你注明类型就能知道你的意图。

当一个函数有多个参数时，使用逗号隔开它们，像这样：

文件名: `src/main.rs`

```

fn main() {
    another_function(5, 6);
}

fn another_function(x: i32, y: i32) {
    println!("The value of x is: {}", x);
    println!("The value of y is: {}", y);
}

```

这个例子创建了一个有两个参数的函数，都是 `i32` 类型的。函数打印出了这两个参数的值。注意函数参数并不一定都是相同类型的，这个例子中它们只是碰巧相同罢了。

尝试运行代码。使用上面的例子替换当前 *functions* 项目的 `src/main.rs` 文件，并用 `cargo run` 运行它：

```

$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
   Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
    Running `target/debug/functions`
The value of x is: 5
The value of y is: 6

```

因为我们使用 `5` 作为 `x` 的值 `6` 作为 `y` 的值来调用函数，这两个字符串和它们的值被相应打印出来。

函数体

函数体由一系列的语句和一个可选的表达式构成。目前为止，我们只涉及到了没有结尾表达式的函数，不过我们见过表达式作为了语句的一部分。因为 Rust 是一个基于表达式（**expression-based**）的语言，这是一个需要理解的（不同于其他语言）重要区别。其他语言并没有这样的区别，所以让我们看看语句与表达式有什么区别以及它们是如何影响函数体的。

语句与表达式

我们已经用过语句与表达式了。语句（*Statements*）是执行一些操作但不返回值的指令。表达式（*Expressions*）计算并产生一个值。让我们看一些例子：

使用 `let` 关键字创建变量并绑定一个值是一个语句。在列表 3-1 中，`let y = 6;` 是一个语句：

文件名: `src/main.rs`

```
fn main() {  
    let y = 6;  
}
```

列表 3-1: 包含一个语句的 `main` 函数定义

函数定义也是语句，上面整个例子本身就是一个语句。

语句并不返回值。因此，不能把 `let` 语句赋值给另一个变量，比如下面的例子尝试做的，这会产生一个错误：

文件名: `src/main.rs`

```
fn main() {  
    let x = (let y = 6);  
}
```

当运行这个程序，会得到如下错误：

```
$ cargo run  
   Compiling functions v0.1.0 (file:///projects/functions)  
error: expected expression, found statement (`let`)  
--> src/main.rs:2:14  
   |  
2 |     let x = (let y = 6);  
   |               ^^^  
   = note: variable declaration using `let` is a statement
```

`let y = 6` 语句并不返回值，所以并没有 `x` 可以绑定的值。这与其他语言不同，例如 **C** 和 **Ruby**，它们的赋值语句返回所赋的值。在这些语言中，可以这么写 `x = y = 6` 这样 `x` 和 `y` 的值都是 `6`；这在 **Rust** 中可不行。

表达式计算出一些值，而且它们组成了其余大部分你将会编写的 **Rust** 代码。考虑一个简单的数学运算，比如 `5 + 6`，这是一个表达式并计算出值 `11`。表达式可以是语句的一部分：在列表 3-3 中有这个语句 `let y = 6;`，`6` 是一个表达式，它计算出的值是 `6`。函数调用是一个表达式。宏调用是一个表达式。我们用来创建新作用域的大括号（代码块），`{}`，也是一个表达式，例如：

文件名: `src/main.rs`

```
fn main() {  
    let x = 5;  
  
    let y = {  
        let x = 3;  
        x + 1  
    };  
  
    println!("The value of y is: {}", y);  
}
```

这个表达式：

```
{
    let x = 3;
    x + 1
}
```

是一个代码块，它的值是 `4`。这个值作为 `let` 语句的一部分被绑定到 `y` 上。注意结尾没有分号的那一行，与大部分我们见过的代码行不同。表达式并不包含结尾的分号。如果在表达式的结尾加上分号，他就变成了语句，这也就使其不返回一个值。在接下来的探索中记住函数和表达式都返回值就行了。

函数的返回值

函数可以向调用它的代码返回值。我们并不对返回值命名，不过会在一个箭头（`->`）后声明它的类型。在 **Rust** 中，函数的返回值等同于函数体最后一个表达式的值。这是一个有返回值的函数的例子：

文件名: `src/main.rs`

```
fn five() -> i32 {
    5
}

fn main() {
    let x = five();

    println!("The value of x is: {}", x);
}
```

在函数 `five` 中并没有函数调用、宏、甚至也没有 `let` 语句——只有数字 `5` 自身。这在 **Rust** 中是一个完全有效的函数。注意函数的返回值类型也被指定了，就是 `-> i32`。尝试运行代码；输出应该看起来像这样：

```
$ cargo run
Compiling functions v0.1.0 (file:///projects/functions)
Finished dev [unoptimized + debuginfo] target(s) in 0.30 secs
Running `target/debug/functions`
The value of x is: 5
```

函数 `five` 的返回值是 `5`，也就是为什么返回值类型是 `i32`。让我们仔细检查一下这段代码。这有两个重要的部分：首先，`let x = five();` 这一行表明我们使用函数的返回值来初始化了一个变量。因为函数 `five` 返回 `5`，这一行与如下这行相同：

```
let x = 5;
```

其次，函数 `five` 没有参数并定义了返回值类型，不过函数体只有单一个 `5` 也没有分号，因为这是我们想要返回值的表达式。让我们看看另一个例子：

文件名: `src/main.rs`

```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {}", x);
}

fn plus_one(x: i32) -> i32 {
    x + 1
}
```

运行代码会打印出 `The value of x is: 6`。如果在包含 `x + 1` 的那一行的结尾加上一个分号，把它从表达式变成语句后会怎样呢？

文件名: `src/main.rs`

```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {}", x);
}

fn plus_one(x: i32) -> i32 {
    x + 1;
}
```

运行代码会产生一个错误，如下：

```
error[E0308]: mismatched types
--> src/main.rs:7:28
  |
7 |   fn plus_one(x: i32) -> i32 {
  |   _____^
8 | |       x + 1;
  | |           - help: consider removing this semicolon
9 | | }
  | |_^ expected i32, found ()
  |
= note: expected type `i32`
        found type `()``
```

主要的错误信息，“**mismatched types**”（类型不匹配），揭示了代码的核心问题。函数 `plus_one` 的定义说明它要返回一个 `i32`，不过语句并不返回一个值，这由那个空元组 `()` 表明。因此，这个函数返回了空元组 `()`，这与函数定义相矛盾并导致一个错误。在输出中，**Rust** 提供了一个可能会对修正问题有帮助的信息：它建议去掉分号，这会修复这个错误。

注释

ch03-04-comments.md

commit d4c77666f480edfb960cc9b11a31c42f4b90c745

所有编程语言都力求使其代码易于理解，不过有时需要提供额外的解释。在这种情况下，程序员在源码中留下记录，或者注释（*comments*），编译器会忽略它们不过其他阅读代码的人可能会用得上。

这是一个简单的注释的例子：

```
// Hello, world.
```

在 **Rust** 中，注释必须以两道斜杠开始，并持续到本行的结尾。对于超过一行的注释，需要在每一行都加上 `//`，像这样：

```
// So we're doing something complicated here, long enough that we need
// multiple lines of comments to do it! Whew! Hopefully, this comment will
// explain what's going on.
```

注释也可以在放在包含代码的行的末尾：

文件名: `src/main.rs`

```
fn main() {
    let lucky_number = 7; // I'm feeling lucky today.
}
```

不过你会经常看到它们用于这种格式，也就是位于它所解释的代码行的上面一行：

文件名: `src/main.rs`

```
fn main() {
    // I'm feeling lucky today.
    let lucky_number = 7;
}
```

Rust 还有另一种注释，称为文档注释，我们将在 14 章讨论它。

控制流

ch03-05-control-flow.md

commit ec65990849230388e4ce4db5b7a0cb8a0f0d60e2

通过条件是不是为真来决定是否执行某些代码，或者根据条件是否为真来重复运行一段代码是大部分编程语言的基本组成部分。**Rust** 代码中最常见的用来控制执行流的结构是 `if` 表达式和循环。

if 表达式

`if` 表达式允许根据条件执行不同的代码分支。我们提供一个条件并表示“如果符合这个条件，运行这段代码。如果条件不满足，不运行这段代码。”

在 `projects` 目录创建一个叫做 `branches` 的新项目来学习 `if` 表达式。在 `src/main.rs` 文件中，输入如下内容：

文件名: `src/main.rs`

```
fn main() {  
    let number = 3;  
  
    if number < 5 {  
        println!("condition was true");  
    } else {  
        println!("condition was false");  
    }  
}
```

所有的 `if` 表达式都以 `if` 关键字开头，其后跟一个条件。在这个例子中，条件检查变量 `number` 是否有一个小于 5 的值。在条件为真时希望执行的代码块位于紧跟条件之后的大括号中。`if` 表达式中与条件关联的代码块有时被叫做 **arms**，就像第二章“比较猜测与秘密数字”部分中讨论到的 `match` 表达式中分支一样。也可以包含一个可选的 `else` 表达式来提供一个在条件为假时应当执行的代码块，这里我们就这么做了。如果不提供 `else` 表达式并且条件为假时，程序会直接忽略 `if` 代码块并继续执行下面的代码。

尝试运行代码，应该能看到如下输出：

```
$ cargo run  
Compiling branches v0.1.0 (file:///projects/branches)  
Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs  
Running `target/debug/branches`  
condition was true
```

尝试改变 `number` 的值使条件为假时看看会发生什么：

```
let number = 7;
```

再次运行程序并查看输出：

```
$ cargo run  
Compiling branches v0.1.0 (file:///projects/branches)  
Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs  
Running `target/debug/branches`  
condition was false
```

另外值得注意的是代码中的条件 必须是 `bool` 值。如果想看看条件不是 `bool` 值时会发生什么，尝试运行如下代码：

文件名: `src/main.rs`

```
fn main() {
    let number = 3;

    if number {
        println!("number was three");
    }
}
```

这里 `if` 条件的值是 `3`，`Rust` 抛出了一个错误：

```
error[E0308]: mismatched types
--> src/main.rs:4:8
|
4 |     if number {
|         ^^^^^ expected bool, found integral variable
|
= note: expected type `bool`
       found type `{integer}`
```

这个错误表明 `Rust` 期望一个 `bool` 不过却得到了一个整型。不像 `Ruby` 或 `JavaScript` 这样的语言，`Rust` 并不会尝试自动地将非布尔值转换为布尔值。必须总是显式地使用布尔值作为 `if` 的条件。例如，如果想要 `if` 代码块只在一个数字不等于 `0` 时执行，可以把 `if` 表达式修改成下面这样：

文件名: `src/main.rs`

```
fn main() {
    let number = 3;

    if number != 0 {
        println!("number was something other than zero");
    }
}
```

运行代码会打印出 `number was something other than zero`。

使用 `else if` 实现多重条件

可以将 `else if` 表达式与 `if` 和 `else` 组合来实现多重条件。例如：

文件名: `src/main.rs`

```
fn main() {
    let number = 6;

    if number % 4 == 0 {
        println!("number is divisible by 4");
    } else if number % 3 == 0 {
        println!("number is divisible by 3");
    } else if number % 2 == 0 {
        println!("number is divisible by 2");
    } else {
        println!("number is not divisible by 4, 3, or 2");
    }
}
```

```
}
```

这个程序有四个可能的执行路径。运行后应该能看到如下输出：

```
$ cargo run
  Compiling branches v0.1.0 (file:///projects/branches)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
     Running `target/debug/branches`
number is divisible by 3
```

当执行这个程序，它按顺序检查每个 `if` 表达式并执行第一个条件为真的代码块。注意即使 6 可以被 2 整除，也不会出现 `number is divisible by 2` 的输出，更不会出现 `else` 块中的 `number is not divisible by 4, 3, or 2`。原因是 **Rust** 只会执行第一个条件为真的代码块，并且一旦它找到一个以后，甚至就不会检查剩下的条件了。

使用过多的 `else if` 表达式会使代码显得杂乱无章，所以如果有多于一个 `else if`，最好重构代码。为此第六章会介绍 **Rust** 中一个叫做 `match` 的强大的分支结构（branching construct）。

在 `let` 语句中使用 `if`

因为 `if` 是一个表达式，我们可以在 `let` 语句的右侧使用它，例如在示例 3-2 中：

文件名: `src/main.rs`

```
fn main() {
    let condition = true;
    let number = if condition {
        5
    } else {
        6
    };

    println!("The value of number is: {}", number);
}
```

示例 3-2: 将 `if` 的返回值赋值给一个变量

`number` 变量将会绑定到基于 `if` 表达式结果的值。运行这段代码看看会出现什么：

```
$ cargo run
  Compiling branches v0.1.0 (file:///projects/branches)
    Finished dev [unoptimized + debuginfo] target(s) in 0.30 secs
     Running `target/debug/branches`
The value of number is: 5
```

还记得代码块的值是其最后一个表达式的值，以及数字本身也是一个表达式吗。在这个例子中，整个 `if` 表达式的值依赖哪个代码块被执行。这意味着 `if` 的每个分支的可能的返回值都必须是相同类型；在示例 3-2 中，`if` 分支和 `else` 分支的结果都是 `i32` 整型。如果它们的类型不匹配，如下面这个例子，则会出现一个错误：

文件名: `src/main.rs`

```
fn main() {
    let condition = true;

    let number = if condition {
        5
    } else {
        "six"
    }
}
```



```
};

println!("The value of number is: {}", number);
}
```

当运行这段代码，会得到一个错误。`if` 和 `else` 分支的值类型是不相容的，同时 **Rust** 也准确地表明了程序中的何处发现的这个问题：

```
error[E0308]: if and else have incompatible types
--> src/main.rs:4:18
   |
 4 |         let number = if condition {
   |         ^
 5 | |         5
 6 | |     } else {
 7 | |         "six"
 8 | |     };
   | |____^ expected integral variable, found reference
   |
   = note: expected type `{integer}`
           found type `&str`
```

`if` 代码块的表达式返回一个整型，而 `else` 代码块返回一个字符串。这并不可行，因为变量必须只有一个类型。**Rust** 需要在编译时就确切的知道 `number` 变量的类型，这样它就可以在编译时证明其他使用 `number` 变量的地方它的类型是有效的。**Rust** 并不能够在 `number` 的类型只能在运行时确定的情况下工作；这样会使编译器变得更复杂而且只能为代码提供更少的保障，因为它不得不记录所有变量的多种可能的类型。

使用循环重复执行

多次执行同一段代码是很常用的，**Rust** 为此提供了多种 循环（*loops*）。一个循环执行循环体中的代码直到结尾并紧接着回到开头继续执行。为了实验一下循环，让我们创建一个叫做 *loops* 的新项目。

Rust 有三种循环类型：`loop`、`while` 和 `for`。让我们每一个都试试。

使用 `loop` 重复执行代码

`loop` 关键字告诉 **Rust** 一遍又一遍地执行一段代码直到你明确要求停止。

作为一个例子，将 *loops* 目录中的 `src/main.rs` 文件修改为如下：

文件名: `src/main.rs`

```
fn main() {
    loop {
        println!("again!");
    }
}
```

当执行这个程序，我们会看到 `again!` 被连续的打印直到我们手动停止程序。大部分终端都支持一个键盘快捷键，`ctrl-C`，来终止一个陷入无限循环的程序。尝试一下：

```
$ cargo run
Compiling loops v0.1.0 (file:///projects/loops)
Finished dev [unoptimized + debuginfo] target(s) in 0.29 secs
Running `target/debug/loops`
again!
again!
```

```
again!
again!
^Cagain!
```

符号 `^c` 代表你在这按下了 `ctrl-C`。在 `^c` 之后你可能看到也可能看不到 `again!`，这依赖于在接收到终止信号时代码执行到了循环的何处。

幸运的是，**Rust** 提供了另一个更可靠的方式来退出循环。可以使用 `break` 关键字来告诉程序何时停止执行循环。回忆一下在第二章猜猜看游戏的“猜测正确后退出”部分使用过它来在用户猜对数字赢得游戏后退出程序。

while 条件循环

在程序中计算循环的条件也很常见。当条件为真，执行循环。当条件不再为真，调用 `break` 停止循环。这个循环类型可以通过组合 `loop`、`if`、`else` 和 `break` 来实现；如果你喜欢的话，现在就可以在程序中试试。

然而，这个模式太常见了以至于 **Rust** 为此提供了一个内建的语言结构，它被称为 `while` 循环。下面的例子使用了 `while`：程序循环三次，每次数字都减一。接着，在循环之后，打印出另一个信息并退出：

文件名: `src/main.rs`

```
fn main() {
    let mut number = 3;

    while number != 0 {
        println!("{}", number);

        number = number - 1;
    }

    println!("LIFTOFF!!!");
}
```

这个结构消除了很多需要嵌套使用 `loop`、`if`、`else` 和 `break` 的代码，这样显得更加清楚。当条件为真就执行，否则退出循环。

使用 for 遍历集合

可以使用 `while` 结构来遍历一个元素集合，比如数组。例如，看看如下的示例 3-3：

文件名: `src/main.rs`

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    let mut index = 0;

    while index < 5 {
        println!("the value is: {}", a[index]);

        index = index + 1;
    }
}
```

示例 3-3：使用 `while` 循环遍历集合中的每一个元素

这里代码对数组中的元素进行计数。它从索引 `0` 开始，并接着循环直到遇到数组的最后一个索引（这时，`index < 5` 不再为真）。运行这段代码会打印出数组中的每一个元素：

```
$ cargo run
Compiling loops v0.1.0 (file:///projects/loops)
Finished dev [unoptimized + debuginfo] target(s) in 0.32 secs
Running `target/debug/loops`
the value is: 10
the value is: 20
the value is: 30
the value is: 40
the value is: 50
```

所有数组中的五个元素都如期被打印出来。尽管 `index` 在某一时刻会到达值 `5`，不过循环在其尝试从数组获取第六个值（会越界）之前就停止了。

不过这个过程是容易出错的；如果索引长度不正确会导致程序 **panic**。这也使程序更慢，因为编译器增加了运行时代码来对每次循环的每个元素进行条件检查。

可以使用 `for` 循环来对一个集合的每个元素执行一些代码，来作为一个更有效率的替代。`for` 循环看起来如示例 3-4 所示：

文件名: `src/main.rs`

```
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a.iter() {
        println!("the value is: {}", element);
    }
}
```

示例 3-4：使用 `for` 循环遍历集合中的每一个元素

当运行这段代码，将看到与示例 3-3 一样的输出。更为重要的是，我们增强了代码安全性并消除了出现可能会导致超出数组的结尾或遍历长度不够而缺少一些元素这类 **bug** 的机会。

例如，在示例 3-3 的代码中，如果从数组 `a` 中移除一个元素但忘记更新条件为 `while index < 4`，代码将会 **panic**。使用 `for` 循环的话，就不需要惦记着在更新数组元素数量时修改其他的代码了。

`for` 循环的安全性和简洁性使得它在成为 **Rust** 中使用最多的循环结构。即使是在想要循环执行代码特定次数时，例如示例 3-3 中使用 `while` 循环的倒计时例子，大部分 **Rustacean** 也会使用 `for` 循环。这么做的方式是使用 `Range`，它是标准库提供的用来生成从一个数字开始到另一个数字之前结束的所有数字序列的类型。

下面是一个使用 `for` 循环来倒计时的例子，它还使用了一个我们还未讲到的方法，`rev`，用来反转 `range`：

文件名: `src/main.rs`

```
fn main() {
    for number in (1..4).rev() {
        println!("{}", number);
    }
    println!("LIFTOFF!!!");
}
```

这段代码看起来更帅气不是吗？

总结

你做到了！这是一个大章节：你学习了变量，标量和 `if` 表达式，还有循环！如果你想要实践本章讨论的概念，尝试构建如下的程序：

- 相互转换摄氏与华氏温度
- 生成 n 阶斐波那契数列
- 打印圣诞颂歌 “The Twelve Days of Christmas” 的歌词，并利用歌曲中的重复部分（编写循环）

当你准备好继续的时候，让我们讨论一个其他语言中 并不 常见的概念：所有权（ownership）。

认识所有权

[ch04-00-understanding-ownership.md](#)

commit 4f2dc564851dc04b271a2260c834643dfd86c724

所有权（系统）是 **Rust** 最独特的功能，其令 **Rust** 无需垃圾回收（**garbage collector**）即可保障内存安全。因此，理解 **Rust** 中所有权如何工作是十分重要的。本章我们将讲到所有权以及相关功能：借用、**slice** 以及 **Rust** 如何在内存中布局数据。

什么是所有权

ch04-01-what-is-ownership.md

commit ec65990849230388e4ce4db5b7a0cb8a0f0d60e2

Rust 的核心功能（之一）是所有权（*ownership*）。虽然这个功能说明起来很直观，不过它对语言的其余部分有着更深层的含义。

所有程序都必须管理其运行时使用计算机内存的方式。一些语言中使用垃圾回收在程序运行过程中来时刻寻找不再被使用的内存；在另一些语言中，程序员必须亲自分配和释放内存。**Rust** 则选择了第三种方式：内存被一个所有权系统管理，它拥有一系列的规则使编译器在编译时进行检查。任何所有权系统的功能都不会导致运行时开销。

因为所有权对很多程序员来说都是一个新概念，需要一些时间来适应。好消息是随着你对 **Rust** 和所有权系统的规则越来越有经验，你就越能自然地编写出安全和高效的代码。持之以恒！

当你理解了所有权系统，你就会对这个使 **Rust** 如此独特的功能有一个坚实的基础。在本章中，你将会通过一些例子来学习所有权，他们关注一个非常常见的数据结构：字符串。

栈（Stack）与堆（Heap）

在很多语言中并不经常需要考虑到栈与堆。不过在像 **Rust** 这样的系统编程语言中，值是位于栈上还是堆上在更大程度上影响了语言的行为以及为何必须做出这样的选择。我们会在本章的稍后部分描述所有权与堆与栈相关的部分，所以这里只是一个用来预热的简要解释。

栈和堆都是代码在运行时可供使用的内存部分，不过他们以不同的结构组成。栈以放入值的顺序存储并以相反顺序取出值。这也被称作 后进先出（*last in, first out*）。想象一下一叠盘子：当增加更多盘子时，把他们放在盘子堆的顶部，当需要盘子时，也从顶部拿走。不能从中间也不能从底部增加或拿走盘子！增加数据叫做 进栈（*pushing onto the stack*），而移出数据叫做 出栈（*popping off the stack*）。

操作栈是非常快的，因为它访问数据的方式：永远也不需要寻找一个位置放入新数据或者取出数据因为这个位置总是在栈顶。另一个使得栈快速的性质是栈中的所有数据都必须有一个已知且固定的大小。

相反对于在编译时未知大小或大小可能变化的数据，可以把他们储存在堆上。堆是缺乏组织的：当向堆放入数据时，我们请求一定大小的空间。操作系统在堆的某处找到一块足够大的空位，把它标记为已使用，并返回给我们一个其位置的 指针（*pointer*）。这个过程称作 在堆上分配内存（*allocating on the heap*），并且有时这个过程就简称为“分配”（*allocating*）。向栈中放入数据并不被认为是分配。因为指针是已知的固定大小的，我们可以将指针储存在栈上，不过当需要实际数据时，必须访问指针。

想象一下去餐馆就坐吃饭。当进入时，你说明有几个人，餐馆员工会找到一个够大的空桌子并领你们过去。如果有人来迟了，他们也可以通过询问来找到你们坐在哪。

访问堆上的数据要比访问栈上的数据要慢因为必须通过指针来访问。现代处理器在内存中跳转越少就越快（缓存）。继续类比，假设有一个服务员在餐厅里处理多个桌子的点菜。在一个桌子报完所有菜后再移动到下一个桌子是最有效率的。从桌子 A 听一个菜，接着桌子 B 听一个菜，然后再桌子 A，然后再桌子 B 这样的流程会更加缓慢。出于同样原因，处理器在处理的数据之间彼此较近的时候（比如在栈上）比较远的时候（比如可能在堆上）能更好的工作。在堆上分配大量的空间也可能消耗时间。

当调用一个函数，传递给函数的值（包括可能指向堆上数据的指针）和函数的局部变量被压入栈中。当函数结束时，这些值被移出栈。

记录何处的代码在使用堆上的什么数据，最小化堆上的冗余数据的数量以及清理堆上不再使用的数据以致不至于耗尽空间，这些所有的问题正是所有权系统要处理的。一旦理解了所有权，你就不需要经常考虑栈和堆了，不过理解如何管理堆内存可以帮助我们理解所有权为何存在以及为什么要以这种方式工作。

所有权规则

首先，让我们看一下所有权的规则。请记住它们，我们将讲解一些它们的例子：

1. Rust 中每一个值都有一个称之为其所有者（*owner*）的变量。
2. 值有且只能有一个所有者。
3. 当所有者（变量）离开作用域，这个值将被丢弃。

变量作用域

我们已经在第二章完成过一个 Rust 程序的例子。现在我们已经掌握了基本语法，所以不会在之后的例子中包含 `fn main() {` 代码了，所以如果你是一路跟过来的，必须手动将之后例子的代码放入一个 `main` 函数中。为此，例子将显得更加简明，使我们可以关注具体细节而不是样板代码。

作为所有权的第一个例子，我们看看一些变量的作用域（*scope*）。作用域是一个项(原文：item) 在程序中有效的范围。假设有这样一个变量：

```
let s = "hello";
```

变量 `s` 绑定到了一个字符串字面值，这个字符串值是硬编码进程序代码中的。这个变量从声明的点开始直到当前作用域结束时都是有效的。示例 4-1 的注释标明了变量 `s` 在何处是有效的：

```
{
    // s is not valid here, it's not yet declared
    let s = "hello"; // s is valid from this point forward

    // do stuff with s
} // this scope is now over, and s is no longer valid
```

示例 4-1：一个变量和其有效的作用域

换句话说，这里有两个重要的点：

1. 当 `s` 进入作用域时，它就是有效的。
2. 这一直持续到它离开作用域为止。

目前为止，变量是否有效与作用域的关系跟其他编程语言是类似的。现在我们在此基础上介绍 `String` 类型。

String 类型

为了演示所有权的规则，我们需要一个比第三章讲到的任何一个都要复杂的数据类型。“Data Types”部分涉及到的数据类型都是储存在栈上的并且当离开作用域时被移出栈，不过我们需要寻找一个储存在堆上的数据来探索 Rust 是如何知道该在何时清理数据的。

这里使用 `String` 作为例子并专注于 `String` 与所有权相关的部分。这些方面也同样适用于其他标准库提供的或你自己创建的复杂数据类型。在第八章会更深入地讲解 `String`。

我们已经见过字符串字面值了，它被硬编码进程序里。字符串字面值是很方便的，不过他们并不总是适合所有需要使用文本的场景。原因之一就是他们是不可变的。另一个原因是不是所有字符串的值都能在编写代码时就知道：例如，如果想要获取用户输入并储存该怎么办呢？为此，Rust 有第二个字符串类型，`String`。这个类型储存在堆上。

所以能够储存在编译时未知大小的文本。可以用 `from` 函数从字符串字面值来创建 `String`，如下：

```
let s = String::from("hello");
```

这两个冒号（`::`）运算符允许将特定的 `from` 函数置于 `String` 类型的命名空间（`namespace`）下而不需要使用类似 `string_from` 这样的名字。在第五章的“方法语法”（“`Method Syntax`”）部分会着重讲解这个语法而且在第七章会讲到模块的命名空间。

这类字符串 可以 被修改：

```
let mut s = String::from("hello");

s.push_str(", world!"); // push_str() appends a literal to a String

println!("{}", s); // This will print `hello, world!`
```

那么这里有什么区别呢？为什么 `String` 可变而字面值却不行呢？区别在于两个类型对内存的处理上。

内存与分配

对于字符串字面值的情况，我们在编译时就知道其内容所以它直接被硬编码进最终的可执行文件中，这使得字符串字面值快速且高效。不过这些属性都只来源于其不可变性。不幸的是，我们不能为了每一个在编译时未知大小的文本而将一块内存放入二进制文件中而它的大小还可能随着程序运行而改变。

对于 `String` 类型，为了支持一个可变，可增长的文本片段，需要在堆上分配一块在编译时未知大小的内存来存放内容。这意味着：

1. 内存必须在运行时向操作系统请求。
2. 需要一个当我们处理完 `String` 时将内存返回给操作系统的方法。

第一部分由我们完成：当调用 `String::from` 时，它的实现 (*implementation*) 请求其所需的内存。这在编程语言中是非常通用的。

然而，第二部分实现起来就各有区别了。在有 垃圾回收 (*garbage collector, GC*) 的语言中，`GC` 记录并清除不再使用的内存，而我们作为程序员，并不需要关心他们。没有 `GC` 的话，识别出不再使用的内存并调用代码显式释放就是我们程序员的责任了，正如请求内存的时候一样。从历史的角度上说正确处理内存回收曾经是一个困难的编程问题。如果忘记回收了会浪费内存。如果过早回收了，将会出现无效变量。如果重复回收，这也是个 `bug`。我们需要 `allocate` 和 `free` 一一对应。

`Rust` 采取了一个不同的策略：内存拥有它的变量离开作用域后就被自动释放。下面是示例 4-1 中作用域例子的一个使用 `String` 而不是字符串字面值的版本：

```
{
    let s = String::from("hello"); // s is valid from this point forward

    // do stuff with s
}                                     // this scope is now over, and s is no
                                     // longer valid
```

这里是一个将 `String` 需要的内存返回给操作系统的很自然的位置：当 `s` 离开作用域的时候。当变量离开作用域，`Rust` 为其调用一个特殊的函数。这个函数叫做 `drop`，在这里 `String` 的作者可以放置释放内存的代码。`Rust` 在结尾的 `}` 处自动调用 `drop`。

注意：在 C++ 中，这种 item 在生命周期结束时释放资源的方法有时被称作 资源获取即初始化（*Resource Acquisition Is Initialization (RAII)*）。如果你使用过 RAII 模式的话应该对 Rust 的 `drop` 函数并不陌生。

这个模式对编写 Rust 代码的方式有着深远的影响。现在它看起来很简单，不过在更复杂的场景下代码的行为可能是不可预测的，比如当有多个变量使用在堆上分配的内存时。现在让我们探索一些这样的场景。

变量与数据交互的方式（一）：移动

Rust 中的多个变量可以采用一种独特的方式与同一数据交互。让我们看看示例 4-2 中一个使用整型的例子：

```
let x = 5;
let y = x;
```

示例 4-2：将变量 `x` 赋值给 `y`

根据其他语言的经验我们大致可以猜到这在干什么：“将 `5` 绑定到 `x`；接着生成一个值 `x` 的拷贝并绑定到 `y`”。现在有了两个变量，`x` 和 `y`，都等于 `5`。这也正是事实上发生了的，因为正数是有已知固定大小的简单值，所以这两个 `5` 被放入了栈中。

现在看看这个 `String` 版本：

```
let s1 = String::from("hello");
let s2 = s1;
```

这看起来与上面的代码非常类似，所以我们可能会假设他们的运行方式也是类似的：也就是说，第二行可能会生成一个 `s1` 的拷贝并绑定到 `s2` 上。不过，事实上并不完全是这样。

为了更全面的解释这个问题，让我们看看图 4-1 中 `String` 真正是什么样的。`String` 由三部分组成，如图左侧所示：一个指向存放字符串内容内存的指针，一个长度，和一个容量。这一组数据储存在栈上。右侧则是堆上存放内容的内存部分。

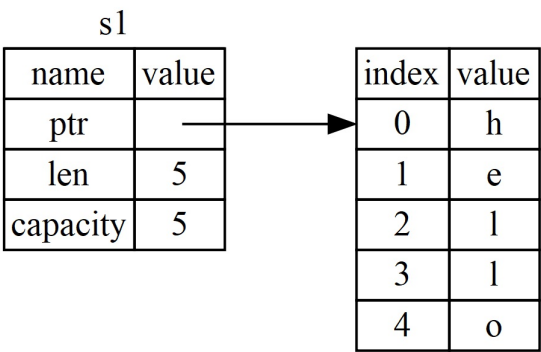


图 4-1：一个绑定到 `s1` 的拥有值 `"hello"` 的 `String` 的内存表现

长度代表当前 `String` 的内容使用了多少字节的内存。容量是 `String` 从操作系统总共获取了多少字节的内存。长度与容量的区别是很重要的，不过这在目前为止的场景中并不重要，所以可以暂时忽略容量。

当我们把 `s1` 赋值给 `s2`，`String` 的数据被复制了，这意味着我们从栈上拷贝了它的指针、长度和容量。我们并没有复制堆上指针所指向的数据。换句话说，内存中数据的表现如图 4-2 所示。

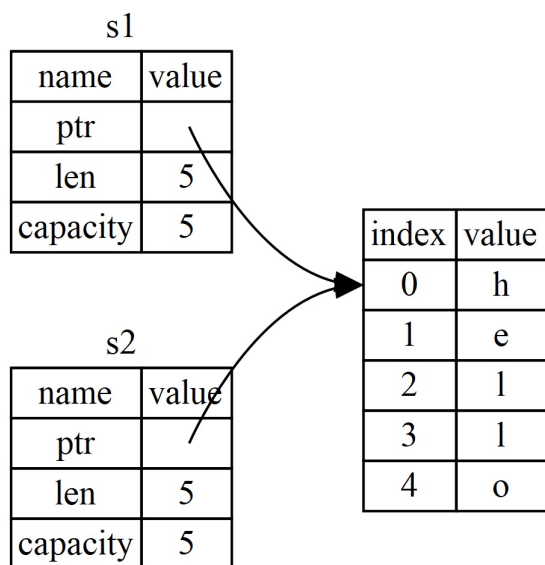


图 4-2: 变量 `s2` 的内存表现，它有一份 `s1` 指针、长度和容量的拷贝

这个表现形式看起来 并不像 图 4-3 中的那样，但是如果 `Rust` 也拷贝了堆上的数据后内存看起来会是如何呢。如果 `Rust` 这么做了，那么操作 `s2 = s1` 在堆上数据比较大的时候可能会对运行时性能造成非常大的影响。

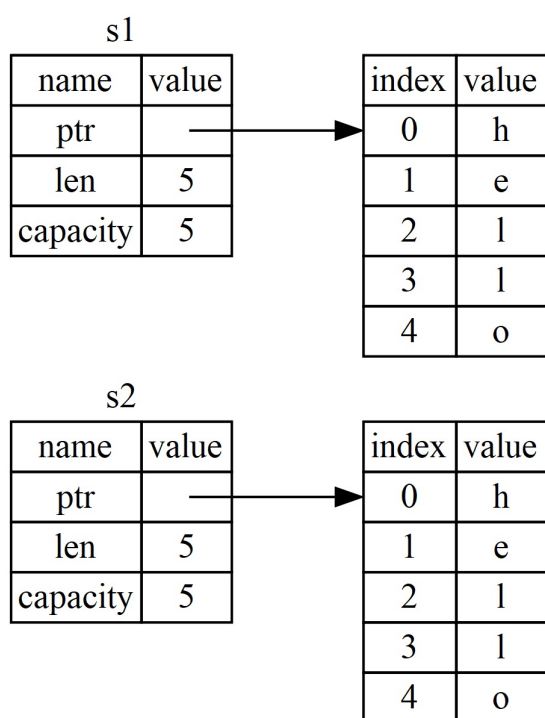


图 4-3: 另一个 `s2 = s1` 时可能的内存表现，如果 `Rust` 同时也拷贝了堆上的数据的话

之前，我们提到过当变量离开作用域后 `Rust` 自动调用 `drop` 函数并清理变量的堆内存。不过图 4-4 展示了两个数据指针指向了同一位置。这就有了一个问题：当 `s2` 和 `s1` 离开作用域，他们都会尝试释放相同的内存。这是一个叫做 二次释放 (*double free*) 的错误，也是之前提到过的内存安全性 bug 之一。两次释放（相同）内存会导致内存污染，它可能会导致潜在的安全漏洞。

为了确保内存安全，这种场景下 Rust 的处理有另一个细节值得注意。与其尝试拷贝被分配的内存，Rust 则认为 `s1` 不再有效，因此 Rust 不需要在 `s1` 离开作用域后清理任何东西。看看在 `s2` 被创建之后尝试使用 `s1` 会发生什么：

```
let s1 = String::from("hello");
let s2 = s1;

println!("{}", world!", s1);
```

你会得到一个类似如下的错误，因为 Rust 禁止你使用无效的引用。

```
error[E0382]: use of moved value: `s1`
--> src/main.rs:5:28
|
3 |     let s2 = s1;
|         -- value moved here
4 |
5 |     println!("{}", world!", s1);
|                               ^^ value used here after move
|
= note: move occurs because `s1` has type `std::string::String`, which does
       not implement the `Copy` trait
```

如果你在其他语言中听说过术语“浅拷贝”（“shallow copy”）和“深拷贝”（“deep copy”），那么拷贝指针、长度和容量而不拷贝数据可能听起来像浅拷贝。不过因为 Rust 同时使第一个变量无效化了，这个操作被称为移动（*move*），而不是浅拷贝。上面的例子可以解读为 `s1` 被移动到了 `s2` 中。那么具体发生了什么，如图 4-4 所示。

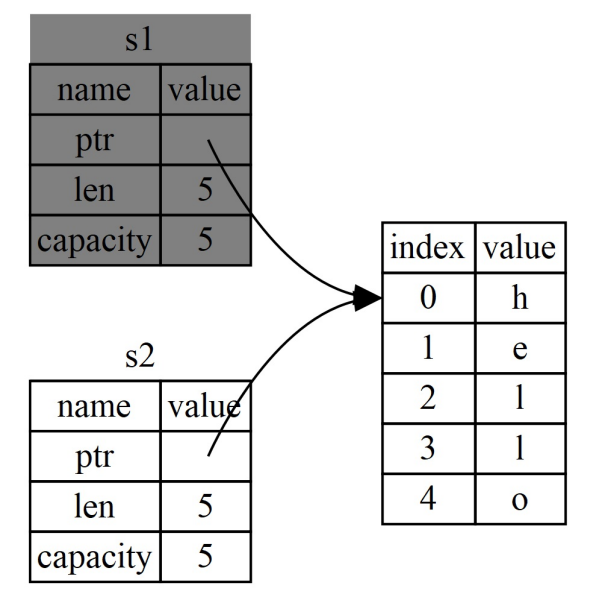


图 4-4： `s1` 无效化之后的内存表现

这样就解决了我们的麻烦！因为只有 `s2` 是有效的，当其离开作用域，它就释放自己的内存，完毕。

另外，这里还隐含了一个设计选择：Rust 永远也不会自动创建数据的“深拷贝”。因此，任何自动的复制可以被认为对运行时性能影响较小。

变量与数据交互的方式（二）：克隆

如果我们 确实 需要深度复制 `String` 中堆上的数据，而不仅仅是栈上的数据，可以使用一个叫做 `clone` 的通用函数。第五章会讨论方法语法，不过因为方法在很多语言中是一个常见功能，所以之前你可能已经见过了。

这是一个实际使用 `clone` 方法的例子：

```
let s1 = String::from("hello");
let s2 = s1.clone();

println!("s1 = {}, s2 = {}", s1, s2);
```

这段代码能正常运行，也是如何显式产生图 4-3 中行为的方式，这里堆上的数据 确实 被复制了。

当出现 `clone` 调用时，你知道一些特定的代码被执行而且这些代码可能相当消耗资源。你很容易察觉到一些不寻常的事情正在发生。

只在栈上的数据：拷贝

这里还有一个没有提到的小窍门。这些代码使用了整型并且是有效的，他们是之前示例 4-2 中的一部分：

```
let x = 5;
let y = x;

println!("x = {}, y = {}", x, y);
```

他们似乎与我们刚刚学到的内容相抵触：没有调用 `clone`，不过 `x` 依然有效且没有被移动到 `y` 中。

原因是像整型这样的在编译时已知大小的类型被整个储存在栈上，所以拷贝其实际的值是快速的。这意味着没有理由在创建变量 `y` 后使 `x` 无效。换句话说，这里没有深浅拷贝的区别，所以这里调用 `clone` 并不会与通常的浅拷贝有什么不同，我们可以不用管它。

Rust 有一个叫做 `Copy trait` 的特殊注解，可以用在类似整型这样的储存在栈上的类型（第十章详细讲解 `trait`）。如果一个类型拥有 `Copy trait`，一个旧的变量在将其赋值给其他变量后仍然可用。**Rust** 不允许自身或其任何部分实现了 `Drop trait` 的类型使用 `Copy trait`。如果我们对其值离开作用域时需要特殊处理的类型使用 `Copy` 注解，将会出现一个编译时错误。关于如何为你的类型增加 `Copy` 注解，请阅读附录 C 中的可导出 `trait`。

那么什么类型是 `Copy` 的呢？可以查看给定类型的文档来确认，不过作为一个通用的规则，任何简单标量值的组合可以是 `Copy` 的，任何需要分配内存，或者本身就是某种形式资源的类型不会是 `Copy` 的。如下是一些 `Copy` 的类型：

- 所有整数类型，比如 `u32`。
- 布尔类型，`bool`，它的值是 `true` 和 `false`。
- 所有浮点数类型，比如 `f64`。
- 元组，当且仅当其包含的类型也都是 `Copy` 的时候。`(i32, i32)` 是 `Copy` 的，不过 `(i32, String)` 就不是。

所有权与函数

将值传递给函数在语义上与给变量赋值相似。向函数传递值可能会移动或者复制，就像赋值语句一样。示例 4-7 是一个展示变量何时进入和离开作用域的例子：

文件名: `src/main.rs`

```
fn main() {
    let s = String::from("hello"); // s comes into scope.

    takes_ownership(s);           // s's value moves into the function...
```

```

// ... and so is no longer valid here.

let x = 5; // x comes into scope.

makes_copy(x); // x would move into the function,
               // but i32 is Copy, so it's okay to still
               // use x afterward.

} // Here, x goes out of scope, then s. But since s's value was moved, nothing
  // special happens.

fn takes_ownership(some_string: String) { // some_string comes into scope.
    println!("{}", some_string);
} // Here, some_string goes out of scope and `drop` is called. The backing
  // memory is freed.

fn makes_copy(some_integer: i32) { // some_integer comes into scope.
    println!("{}", some_integer);
} // Here, some_integer goes out of scope. Nothing special happens.

```

示例 4-3：带有所有权和作用域标注的函数

当尝试在调用 `takes_ownership` 后使用 `s` 时，**Rust** 会抛出一个编译时错误。这些静态检查使我们免于犯错。试试在 `main` 函数中添加使用 `s` 和 `x` 的代码来看看哪里能使用他们，以及所有权规则会在哪里阻止我们这么做。

返回值与作用域

返回值也可以转移作用域。这里是一个拥有与示例 4-3 中类似标注的例子：

文件名: `src/main.rs`

```

fn main() {
    let s1 = gives_ownership(); // gives_ownership moves its return
                               // value into s1.

    let s2 = String::from("hello"); // s2 comes into scope.

    let s3 = takes_and_gives_back(s2); // s2 is moved into
                                       // takes_and_gives_back, which also
                                       // moves its return value into s3.
} // Here, s3 goes out of scope and is dropped. s2 goes out of scope but was
  // moved, so nothing happens. s1 goes out of scope and is dropped.

fn gives_ownership() -> String { // gives_ownership will move its
                                // return value into the function
                                // that calls it.

    let some_string = String::from("hello"); // some_string comes into scope.

    some_string // some_string is returned and
                // moves out to the calling
                // function.
}

// takes_and_gives_back will take a String and return one.
fn takes_and_gives_back(a_string: String) -> String { // a_string comes into
                                                        // scope.

    a_string // a_string is returned and moves out to the calling function.
}

```

变量的所有权总是遵循相同的模式：将值赋值给另一个变量时移动它。当持有堆中数据值的变量离开作用域时，其值将通过 `drop` 被清理掉，除非数据被移动为另一个变量所有。

在每一个函数中都获取并接着返回所有权可能有些冗余。如果我们想要函数使用一个值但不获取所有权该怎么办呢？如果我们还要接着使用它的话，每次都传递出去再传回来就有点烦人了，另外我们也可能想要返回函数体产生的任何（不止一个）数据。

可以使用元组来返回多个值，像这样：

文件名: `src/main.rs`

```
fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    println!("The length of '{}' is {}.", s2, len);
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() returns the length of a String.

    (s, length)
}
```

但是这未免有些形式主义，而且这种场景应该很常见。幸运的是，**Rust** 对此提供了一个功能，叫做 引用（*references*）。

引用与借用

ch04-02-references-and-borrowing.md
commit d06a6a181fd61704cbf7feb55bc61d518c6469f9

有这样一个问题：在上一部分的结尾处的使用元组的代码中，因为 `String` 先被移动到了 `calculate_length` 内，因此我们不得不将 `String` 作为返回值，这样在调用 `calculate_length` 后才仍然可以使用 `String`。

下面是如何定义并使用一个（新的） `calculate_length` 函数，它以一个对象的引用作为参数而不是获取值的所有权：

Filename: src/main.rs

```
fn main() {  
    let s1 = String::from("hello");  
  
    let len = calculate_length(&s1);  
  
    println!("The length of '{}' is {}.", s1, len);  
}  
  
fn calculate_length(s: &String) -> usize {  
    s.len()  
}
```

首先，注意变量声明和函数返回值中的所有元组代码都消失了。其次，注意我们传递 `&s1` 给 `calculate_length`，同时在函数定义中，我们获取 `&String` 而不是 `String`。

这些 `&` 符号就是引用，他们允许你使用值但不获取它的所有权。图 4-8 展示了一个图解。

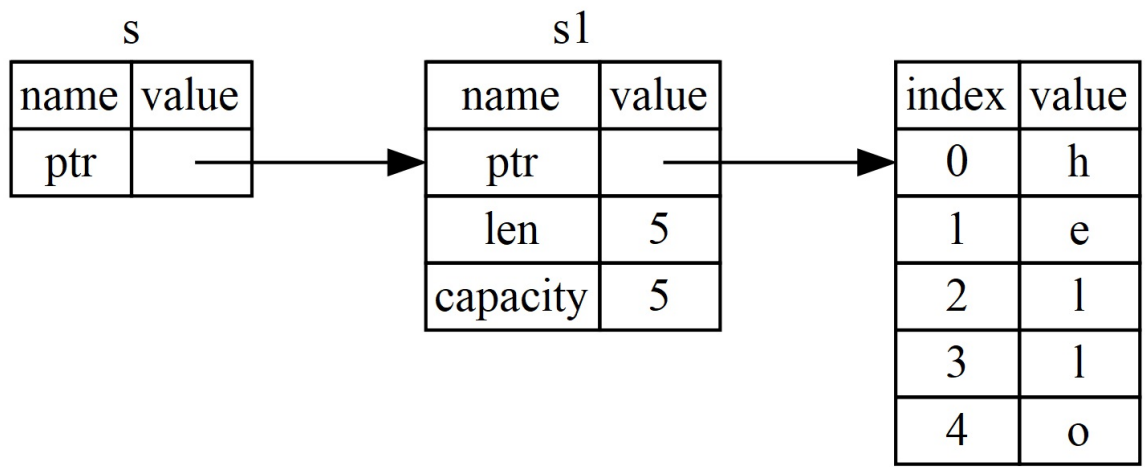


图 4-8: `&String s` 指向 `String s1`

仔细看看这个函数调用：

```
# fn calculate_length(s: &String) -> usize {  
#     s.len()  
# }  
let s1 = String::from("hello");  
  
let len = calculate_length(&s1);
```

`&s1` 语法允许我们创建一个 参考 值 `s1` 的引用，但是并不拥有它。因为并不拥有这个值，当引用离开作用域它指向的值也不会被丢弃。

同理，函数签名使用了 `&` 来表明参数 `s` 的类型是一个引用。让我们增加一些解释性的注解：

```
fn calculate_length(s: &String) -> usize { // s is a reference to a String
    s.len()
} // Here, s goes out of scope. But because it does not have ownership of what
    // it refers to, nothing happens.
```

变量 `s` 有效的作用域与函数参数的作用域一样，不过当引用离开作用域后并不丢弃它指向的数据因为我们没有所有权。函数使用引用而不是实际值作为参数意味着无需返回值来交还所有权，因为就不曾拥有所有权。

我们将获取引用作为函数参数称为 借用（*borrowing*）。正如现实生活中，如果一个人拥有某样东西，你可以从他那里借来。当你使用完毕，必须还回去。

如果我们尝试修改借用的变量呢？尝试示例 4-9 中的代码。剧透：这行不通！

文件名: `src/main.rs`

```
fn main() {
    let s = String::from("hello");

    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

示例 4-9：尝试修改借用的值

这里是错误：

```
error: cannot borrow immutable borrowed content `*some_string` as mutable
--> error.rs:8:5
  |
8 |     some_string.push_str(", world");
  |     ^^^^^^^^^^^^^^^
```

正如变量默认是不可变的，引用也一样。不允许修改引用的值。

可变引用

可以通过一个小调整来修复在示例 4-9 代码中的错误，在示例 4-9 的代码中：

文件名: `src/main.rs`

```
fn main() {
    let mut s = String::from("hello");

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```


首先，必须将 `s` 改为 `mut`。然后必须创建一个可变引用 `&mut s` 和接受一个可变引用 `some_string: &mut String`。

不过可变引用有一个很大的限制：在特定作用域中的特定数据有且只有一个可变引用。这些代码会失败：

文件名: `src/main.rs`

```
let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s;
```

错误如下：

```
error[E0499]: cannot borrow `s` as mutable more than once at a time
--> borrow_twice.rs:5:19
   |
4 |     let r1 = &mut s;
   |               - first mutable borrow occurs here
5 |     let r2 = &mut s;
   |               ^ second mutable borrow occurs here
6 | }
   | - first borrow ends here
```

这个限制允许可变性，不过是以一种受限制的方式。新 **Rustacean** 们经常与此作斗争，因为大部分语言任何时候变量都是可变的。这个限制的好处是 **Rust** 可以在编译时就避免数据竞争。

数据竞争 (***data race***) 是一种特定类型的竞争状态，它可由这三个行为造成：

1. 两个或更多指针同时访问同一数据。
2. 至少有一个这样的指针被用来写入数据。
3. 不存在同步数据访问的机制。

数据竞争会导致未定义行为，难以在运行时追踪，并且难以诊断和修复；**Rust** 避免了这种情况的发生，因为它直接拒绝编译存在数据竞争的代码！

一如既往，可以使用大括号来创建一个新的作用域来允许拥有多个可变引用，只是不能同时拥有：

```
let mut s = String::from("hello");

{
    let r1 = &mut s;

} // r1 goes out of scope here, so we can make a new reference with no problems.

let r2 = &mut s;
```

当结合可变和不可变引用时有一个类似的规则存在。这些代码会导致一个错误：

```
let mut s = String::from("hello");

let r1 = &s; // no problem
let r2 = &s; // no problem
let r3 = &mut s; // BIG PROBLEM
```

错误如下：

```

error[E0502]: cannot borrow `s` as mutable because it is also borrowed as
immutable
--> borrow_thrice.rs:6:19
|
4 |     let r1 = &s; // no problem
|               - immutable borrow occurs here
5 |     let r2 = &s; // no problem
6 |     let r3 = &mut s; // BIG PROBLEM
|               ^ mutable borrow occurs here
7 | }
| - immutable borrow ends here

```

哇哦！我们也不能在拥有不可变引用的同时拥有可变引用。不可变引用的用户可不希望在它的眼皮底下值突然就被改变了！然而，多个不可变引用是没有问题的因为没有哪个只能读取数据的人有能力影响其他人读取到的数据。

尽管这些错误有时使人沮丧，但请牢记这是 **Rust** 编译器在提早指出一个潜在的 **bug**（在编译时而不是运行时）并明确告诉你问题在哪，而不是任由你去追踪为何有时数据并不是你想象中的那样。

悬垂引用

在存在指针的语言中，容易通过释放内存时保留指向它的指针而错误地生成一个悬垂指针（*dangling pointer*），所谓悬垂指针是其指向的内存可能已经被分配给其它持有者。相比之下，在 **Rust** 中编译器确保引用永远也不会变成悬垂状态：当我们拥有一些数据的引用，编译器确保数据不会在其引用之前离开作用域。

让我们尝试创建一个悬垂引用：

Filename: src/main.rs

```

fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");

    &s
}

```

这里是错误：

```

error[E0106]: missing lifetime specifier
--> dangle.rs:5:16
|
5 | fn dangle() -> &String {
|               ^^^^^^^
|
= help: this function's return type contains a borrowed value, but there is no
value for it to be borrowed from
= help: consider giving it a 'static lifetime

error: aborting due to previous error

```

错误信息引用了一个我们还未涉及到的功能：生命周期（*lifetimes*）。第十章会详细介绍生命周期。不过，如果你不理睬生命周期的部分，错误信息确实包含了为什么代码是有问题的关键：

```

this function's return type contains a borrowed value, but there is no value
for it to be borrowed from.

```

让我们仔细看看我们的 `dangle` 代码的每一步到底放生了什么：

```
fn dangle() -> &String { // dangle returns a reference to a String

    let s = String::from("hello"); // s is a new String

    &s // we return a reference to the String, s
} // Here, s goes out of scope, and is dropped. Its memory goes away.
// Danger!
```

因为 `s` 是在 `dangle` 创建的，当 `dangle` 的代码执行完毕后，`s` 将被释放。不过我们尝试返回一个它的引用。这意味着这个引用会指向一个无效的 `String`！这可不对。**Rust** 不会允许我们这么做的。

这里的解决方法是直接返回 `String`：

```
fn no_dangle() -> String {
    let s = String::from("hello");

    s
}
```

这样就可以没有任何错误的运行了。所有权被移动出去，所以没有值被释放掉。

引用的规则

简要的概括一下对引用的讨论：

1. 在任意给定时间，只能 拥有如下中的一个：
 - 一个可变引用。
 - 任意数量的不可变引用。
2. 引用必须总是有效的。

接下来，我们来看看一种不同类型的引用：**slice**。

Slices

ch04-03-slices.md

commit df9e3b922335ec2c76b6c1c4ede31b7742103c48

另一个没有所有权的数据类型是 **slice**。**slice** 允许你引用集合中一段连续的元素序列，而不用引用整个集合。

这里有一个小的编程问题：编写一个获取一个字符串并返回它在其中找到的第一个单词的函数。如果函数没有在字符串中找到一个空格，就意味着整个字符串是一个单词，所以整个字符串都应该返回。

让我们看看这个函数的签名：

```
fn first_word(s: &String) -> ?
```

`first_word` 这个函数有一个参数 `&String`。因为我们不需要所有权，所以这没有问题。不过应该返回什么呢？我们并没有一个真正获取 部分 字符串的办法。不过，我们可以返回单词结尾的索引。让我们试试如示例 4-10 所示的代码：

Filename: src/main.rs

```
fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }

    s.len()
}
```

示例 4-10: `first_word` 函数返回 `String` 参数的一个字节索引值

让我们将代码分解成小块。因为需要一个元素一个元素的检查 `String` 中的值是否是空格，需要用 `as_bytes` 方法将 `String` 转化为字节数组：

```
let bytes = s.as_bytes();
```

接下来，使用 `iter` 方法在字节数组上创建一个迭代器：

```
for (i, &item) in bytes.iter().enumerate() {
```

第十三章将讨论迭代器的更多细节。现在，只需知道 `iter` 方法返回集合中的每一个元素，而 `enumerate` 包装 `iter` 的结果并返回一个元组，其中每一个元素是元组的一部分。返回元组的第一个元素是索引，第二个元素是集合中元素的引用。这比我们自已计算索引要方便一些。

因为 `enumerate` 方法返回一个元组，我们可以使用模式来解构它，就像 `Rust` 中其他地方一样。所以在 `for` 循环中，我们指定了一个模式，其中 `i` 是元组中的索引而 `&item` 是单个字节。因为从 `.iter().enumerate()` 中获取了集合元素的引用，我们在模式中使用 `&`。

我们通过字节的字面值来寻找代表空格的字节。如果找到了，返回它的位置。否则，使用 `s.len()` 返回字符串的长度：

```

        if item == b' ' {
            return i;
        }
    }
    s.len()

```

现在有了一个找到字符串中第一个单词结尾索引的方法了，不过这有一个问题。我们返回了单独一个 `usize`，不过它只在 `&String` 的上下文中才是一个有意义的数字。换句话说，因为它是一个与 `String` 相分离的值，无法保证将来它仍然有效。考虑一下示例 4-11 中使用了示例 4-10 中 `first_word` 函数的程序：

文件名: `src/main.rs`

```

# fn first_word(s: &String) -> usize {
#     let bytes = s.as_bytes();
#
#     for (i, &item) in bytes.iter().enumerate() {
#         if item == b' ' {
#             return i;
#         }
#     }
#
#     s.len()
# }
#
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s); // word will get the value 5.

    s.clear(); // This empties the String, making it equal to "".

    // word still has the value 5 here, but there's no more string that
    // we could meaningfully use the value 5 with. word is now totally invalid!
}

```

示例 4-11：储存 `first_word` 函数调用的返回值紧接着改变 `String` 的内容

这个程序编译时没有任何错误，而且在调用 `s.clear()` 之后使用 `word` 也不会出错。这时 `word` 与 `s` 状态就没有联系了，所以 `word` 仍然包含值 `5`。可以尝试用值 `5` 来提取变量 `s` 的第一个单词，不过这是有 `bug` 的，因为在我们将 `5` 保存到 `word` 之后 `s` 的内容已经改变。

我们不得不时刻担心 `word` 的索引与 `s` 中的数据不再同步，这是冗余且容易出错的！如果编写这么一个 `second_word` 函数的话，管理索引这件事将更加容易出问题。它的签名看起来像这样：

```

fn second_word(s: &String) -> (usize, usize) {

```

现在我们跟踪了一个开始索引 和一个结尾索引，同时有了更多从数据的某个特定状态计算而来的值，他们也完全没有与这个状态相关联。现在有了三个飘忽不定的不相关变量都需要被同步。

幸运的是，`Rust` 为这个问题提供了一个解决方案：字符串 `slice`。

字符串 `slice`

字符串 `slice` (`string slice`) 是 `String` 中一部分值的引用，它看起来像这样：

```

let s = String::from("hello world");

```

```
let hello = &s[0..5];
let world = &s[6..11];
```

这类似于获取整个 `String` 的引用不过带有额外的 `[0..5]` 部分。不同于整个 `String` 的引用，这是一个包含 `String` 内部的一个位置和所需元素数量的引用。`start..end` 语法代表一个以 `start` 开头并一直持续到但不包含 `end` 的 `range`。

使用一个由中括号中的 `[starting_index..ending_index]` 指定的 `range` 创建一个 `slice`，其中 `starting_index` 是包含在 `slice` 的第一个位置，`ending_index` 则是 `slice` 最后一个位置的后一个值。在其内部，`slice` 的数据结构储存了开始位置和 `slice` 的长度，长度对应 `ending_index` 减去 `starting_index` 的值。所以对于 `let world = &s[6..11];` 的情况，`world` 将是一个包含指向 `s` 第 6 个字节的指针和长度值 5 的 `slice`。

图 4-12 展示了一个图例。

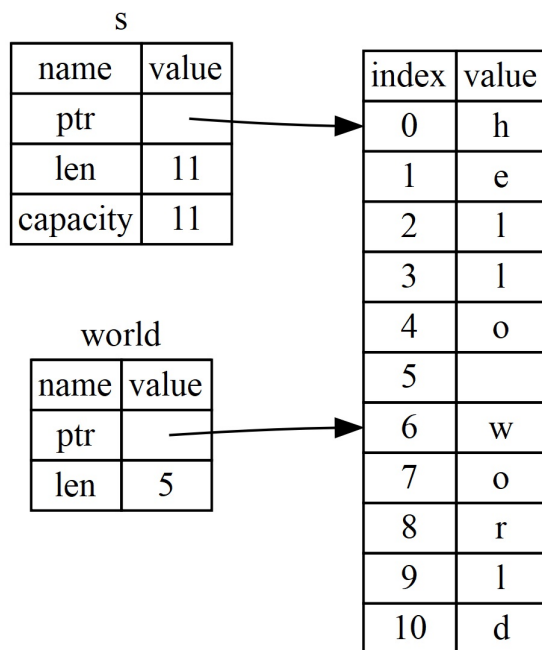


图 4-12: 引用了部分 `String` 的字符串 `slice`

对于 Rust 的 `.. range` 语法，如果想要从第一个索引（0）开始，可以不写两个点号之前的值。换句话说，如下两个语句是相同的：

```
let s = String::from("hello");

let slice = &s[0..2];
let slice = &s[..2];
```

由此类推，如果 `slice` 包含 `String` 的最后一个字节，也可以舍弃尾部的数字。这意味着如下也是相同的：

```
let s = String::from("hello");

let len = s.len();

let slice = &s[3..len];
let slice = &s[3..];
```

也可以同时舍弃这两个值来获取一个整个字符串的 **slice**。所以如下亦是相同的：

```
let s = String::from("hello");

let len = s.len();

let slice = &s[0..len];
let slice = &s[..];
```

在记住所有这些知识后，让我们重写 `first_word` 来返回一个 **slice**。“字符串 **slice**”的类型签名写作 `&str`：

文件名: `src/main.rs`

```
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

我们使用跟示例 4-10 相同的方式获取单词结尾的索引，通过寻找第一个出现的空格。当我们找到一个空格，我们返回一个索引，它使用字符串的开始和空格的索引来作为开始和结束的索引。

现在当调用 `first_word` 时，会返回一个单独的与底层数据相联系的值。这个值由一个 **slice** 开始位置的引用和 **slice** 中元素的数量组成。

`second_word` 函数也可以改为返回一个 **slice**：

```
fn second_word(s: &String) -> &str {
```

现在我们有了一个不易混杂的直观的 **API** 了，因为编译器会确保指向 `String` 的引用保持有效。还记得示例 4-11 程序中，那个当我们获取第一个单词结尾的索引不过接着就清除了字符串所以索引就无效了的 **bug** 吗？那些代码逻辑上是不正确的，不过却没有表现出任何直接的错误。问题会在之后尝试对空字符串使用第一个单词的索引时出现。**slice** 就不可能出现这种 **bug** 并让我们更早的知道出问题了。使用 **slice** 版本的 `first_word` 会抛出一个编译时错误：

Filename: `src/main.rs`

```
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s);

    s.clear(); // Error!
}
```

这里是编译错误：

```
17:6 error: cannot borrow `s` as mutable because it is also borrowed as
      immutable [E0502]
      s.clear(); // Error!
      ^
```

```

15:29 note: previous borrow of `s` occurs here; the immutable borrow prevents
        subsequent moves or mutable borrows of `s` until the borrow ends
    let word = first_word(&s);
                        ^
18:2 note: previous borrow ends here
fn main() {

}
^

```

回忆一下借用规则，当拥有某值的不可变引用时，就不能再获取一个可变引用。因为 `clear` 需要清空 `String`，它尝试获取一个可变引用，它失败了。**Rust** 不仅使得我们的 API 简单易用，也在编译时就消除了一整类的错误！

字符串字面值就是 **slice**

还记得我们讲到过字符串字面值被储存在二进制文件中吗。现在知道 **slice** 了，我们就可以正确的理解字符串字面值了：

```
let s = "Hello, world!";
```

这里 `s` 的类型是 `&str`：它是一个指向二进制程序特定位置的 **slice**。这也就是为什么字符串字面值是不可变的；`&str` 是一个不可变引用。

字符串 **slice** 作为参数

在知道了能够获取字面值和 `String` 的 **slice** 后引起了另一个对 `first_word` 的改进，这是它的签名：

```
fn first_word(s: &String) -> &str {
```

相反一个更有经验的 **Rustacean** 会写下如下这一行，因为它使得可以对 `String` 和 `&str` 使用相同的函数：

```
fn first_word(s: &str) -> &str {
```

如果有一个字符串 **slice**，可以直接传递它。如果有一个 `String`，则可以传递整个 `String` 的 **slice**。定义一个获取字符串 **slice** 而不是字符串引用的函数使得我们的 API 更加通用并且不会丢失任何功能：

Filename: src/main.rs

```

# fn first_word(s: &str) -> &str {
#     let bytes = s.as_bytes();
#
#     for (i, &item) in bytes.iter().enumerate() {
#         if item == b' ' {
#             return &s[0..i];
#         }
#     }
#
#     &s[..]
# }
fn main() {
    let my_string = String::from("hello world");

    // first_word works on slices of `String`s
    let word = first_word(&my_string[..]);

    let my_string_literal = "hello world";

```



```
// first_word works on slices of string literals
let word = first_word(&my_string_literal[..]);

// since string literals are string slices already,
// this works too, without the slice syntax!
let word = first_word(my_string_literal);
}
```

其他 slice

字符串 **slice**，正如你想象的那样，是针对字符串的。不过也有更通用的 **slice** 类型。考虑一下这个数组：

```
let a = [1, 2, 3, 4, 5];
```

就跟我们想要获取字符串的一部分那样，我们也会想要引用数组的一部分，而我们可以这样做：

```
let a = [1, 2, 3, 4, 5];

let slice = &a[1..3];
```

这个 **slice** 的类型是 `&[i32]`。它跟以跟字符串 **slice** 一样的方式工作，通过储存第一个元素的引用和一个长度。你可以对其他所有类型的集合使用这类 **slice**。第八章讲到 **vector** 时会详细讨论这些集合。

总结

所有权、借用和 **slice** 这些概念是 **Rust** 何以在编译时保障内存安全的关键所在。**Rust** 像其他系统编程语言那样给予你对内存使用的控制，但拥有数据所有者在离开作用域后自动清除其数据的功能意味着你无须额外编写和调试相关的控制代码。

所有权系统影响了 **Rust** 中其他很多部分如何工作，所以我们还会继续讲到这些概念，这将贯穿本书的余下内容。让我们开始下一个章节，来看看如何将多份数据组合进一个 `struct` 中。

使用结构体组织相关联的数据

[ch05-00-structs.md](#)

commit d06a6a181fd61704cbf7feb55bc61d518c6469f9

`struct`，是 *structure* 的缩写，是一个允许我们命名并将多个相关值包装进一个有意义的组合的自定义类型。如果你来自一个面向对象编程语言背景，`struct` 就像对象中的数据属性（字段）。在本章中，我们会比较元组与结构体，展示如何使用结构体，并讨论如何在结构体上定义方法和关联函数来指定与结构体数据相关的行为。结构体和枚举（*enum*）（将在第六章讲到）是为了充分利用 Rust 的编译时类型检查来在程序范围内创建新类型的基本组件。

定义并实例化结构体

ch05-01-defining-structs.md

commit 56352c28cf3fe0402fa5a7cba73890e314d720eb

我们在第三章讨论过，结构体与元组类似。就像元组，结构体的每一部分可以是不同类型。不同于元组，结构体需要命名各部分数据以便能清楚的表明其值的意义。由于有了这些名字使得结构体比元组更灵活：不需要依赖顺序来指定或访问实例中的值。

定义结构体，需要使用 `struct` 关键字并为整个结构体提供一个名字。结构体的名字需要描述它所组合的数据的意义。接着，在大括号中，定义每一部分数据的名字，它们被称作 字段（*field*），并定义字段类型。例如，示例 5-1 展示了一个储存用户账号信息的结构体：

```
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}
```

示例 5-1: `User` 结构体定义

一旦定义了结构体后为了使用它，通过为每个字段指定具体值来创建这个结构体的 实例。创建一个实例需要以结构体的名字开头，接着在大括号中使用 `key: value` 对的形式提供字段，其中 `key` 是字段的名称，`value` 是需要储存在字段中的数据值。实例中具体说明字段的顺序不需要和它们在结构体中声明的顺序一致。换句话说，结构体的定义就像一个类型的通用模板，而实例则会在这个模板中放入特定数据来创建这个类型的值。例如，可以像示例 5-2 这样来声明一个特定的用户：

```
# struct User {
#     username: String,
#     email: String,
#     sign_in_count: u64,
#     active: bool,
# }
#
let user1 = User {
    email: String::from("someone@example.com"),
    username: String::from("someusername123"),
    active: true,
    sign_in_count: 1,
};
```

示例 5-2: 创建 `User` 结构体的实例

为了从结构体中获取某个值，可以使用点号。如果我们只想要用户的邮箱地址，可以用 `user1.email`。要更改结构体中的值，如果结构体的实例是可变的，我们可以使用点号并对对应的字段赋值。示例 5-3 展示了如何改变一个可变的 `User` 实例 `email` 字段的值：

```
# struct User {
#     username: String,
#     email: String,
#     sign_in_count: u64,
#     active: bool,
# }
#
```

```
let mut user1 = User {
    email: String::from("someone@example.com"),
    username: String::from("someusername123"),
    active: true,
    sign_in_count: 1,
};

user1.email = String::from("anotheremail@example.com");
```

示例 5-3: 改变 `User` 结构体 `email` 字段的值

与其他任何表达式一样，我们可以在函数体的最后一个表达式构造一个结构体，从函数隐式的返回一个结构体的新实例。表 5-4 显示了一个返回带有给定的 `email` 与 `username` 的 `User` 结构体的实例的 `build_user` 函数。`active` 字段的值为 `true`，并且 `sign_in_count` 的值为 `1`。

```
# struct User {
#     username: String,
#     email: String,
#     sign_in_count: u64,
#     active: bool,
# }
#
fn build_user(email: String, username: String) -> User {
    User {
        email: email,
        username: username,
        active: true,
        sign_in_count: 1,
    }
}
```

示例 5-4: `build_user` 函数获取 `email` 和用户名并返回 `User` 实例

不过，重复 `email` 字段与 `email` 变量的名字，同样的对于 `username`，感觉有一点无趣。将函数参数起与结构体字段相同的名字是可以理解的，但是如果结构体有更多字段，重复它们是十分烦人的。幸运的是，这里有一个方便的语法！

变量与字段同名时的字段初始化语法

如果有变量与字段同名的话，你可以使用 字段初始化语法 (*field init shorthand*)。这可以让创建新的结构体实例的函数更为简练。

在示例 5-4 中，名为 `email` 与 `username` 的参数与结构体 `User` 的字段 `email` 和 `username` 同名。因为名字相同，我们可以写出不重复 `email` 和 `username` 的 `build_user` 函数，如示例 5-5 所示。这个版本的函数与示例 5-4 中代码的行为完全相同。这个字段初始化语法可以让这类代码更简洁，特别是当结构体有很多字段的时候。

```
# struct User {
#     username: String,
#     email: String,
#     sign_in_count: u64,
#     active: bool,
# }
#
fn build_user(email: String, username: String) -> User {
    User {
        email,
        username,
        active: true,
        sign_in_count: 1,
    }
}
```

```
}  
}
```

示例 5-5: `build_user` 函数使用了字段初始化语法, 因为 `email` 和 `username` 参数与结构体字段同名

使用结构体更新语法从其他对象创建对象

可以从老的对象创建新的对象常常是很有帮助的, 即复用大部分老对象的值并只改变一部分。示例 5-6 展示了一个设置 `email` 与 `username` 的值但其余字段使用与示例 5-2 中 `user1` 实例相同的值以创建新的 `User` 实例 `user2` 的例子:

```
# struct User {  
#     username: String,  
#     email: String,  
#     sign_in_count: u64,  
#     active: bool,  
# }  
#  
# let user1 = User {  
#     email: String::from("someone@example.com"),  
#     username: String::from("someusername123"),  
#     active: true,  
#     sign_in_count: 1,  
# };  
#  
let user2 = User {  
    email: String::from("another@example.com"),  
    username: String::from("anotherusername567"),  
    active: user1.active,  
    sign_in_count: user1.sign_in_count,  
};
```

示例 5-6: 创建 `User` 新实例, `user2`, 并将一些字段的值设置为 `user1` 同名字段的值

结构体更新语法 (***struct update syntax***) 可以利用更少的代码获得与示例 5-6 相同的效果。结构体更新语法利用 `..` 以指定未显式设置的字段应有与给定实例对应字段相同的值。示例 5-7 中的代码同样地创建了有着不同的 `email` 与 `username` 值但 `active` 和 `sign_in_count` 字段与 `user1` 相同的实例 `user2` :

```
# struct User {  
#     username: String,  
#     email: String,  
#     sign_in_count: u64,  
#     active: bool,  
# }  
#  
# let user1 = User {  
#     email: String::from("someone@example.com"),  
#     username: String::from("someusername123"),  
#     active: true,  
#     sign_in_count: 1,  
# };  
#  
let user2 = User {  
    email: String::from("another@example.com"),  
    username: String::from("anotherusername567"),  
    ..user1  
};
```

示例 5-7：使用结构体更新语法为 `User` 实例设置新的 `email` 和 `username` 值，但使用 `user1` 变量中剩下字段的值

使用没有命名字段的元组结构体创建不同的类型

也可以定义与元组相像的结构体，称为元组结构体（*tuple structs*），有着结构体名称提供的含义，但没有具体的字段名只有字段的类型。元组结构体的定义仍然以 `struct` 关键字与结构体名称，接下来是元组的类型。如以下是命名为 `Color` 与 `Point` 的元组结构体的定义与使用：

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

let black = Color(0, 0, 0);
let origin = Point(0, 0, 0);
```

注意 `black` 和 `origin` 变量是不同的类型，因为它们是不同的元组结构体的实例。我们定义的每一个结构体有着自己的类型，即使结构体中的字段有相同的类型。在其他方面，元组结构体类似我们在第三章提到的元组。

没有任何字段的类单元结构体

我们也可以定义一个没有任何字段的结构体！它们被称为类单元结构体（*unit-like structs*）因为它们类似于 `()`，即 `unit` 类型。类单元结构体常常在你想要在某个类型上实现 `trait` 但不需要在类型内存储数据的时候发挥作用。我们将在第十章介绍 `trait`。

结构体数据的所有权

在示例 5-1 中的 `User` 结构体的定义中，我们使用了自身拥有所有权的 `String` 类型而不是 `&str` 字符串 `slice` 类型。这是一个有意而为之的选择，因为我们想要这个结构体拥有它所有的数据，为此只要整个结构体是有效的话其数据也应该是有效的。

可以使结构体储存被其他对象拥有的数据的引用，不过这么做的话需要用上 生命周期（*lifetimes*），这是一个第十章会讨论的 `Rust` 功能。生命周期确保结构体引用的数据有效性跟结构体本身保持一致。如果你尝试在结构体中储存一个引用而不指定生命周期，比如这样：

文件名: `src/main.rs`

```
struct User {
    username: &str,
    email: &str,
    sign_in_count: u64,
    active: bool,
}

fn main() {
    let user1 = User {
        email: "someone@example.com",
        username: "someusername123",
        active: true,
        sign_in_count: 1,
    };
}
```

编译器会抱怨它需要生命周期标识符：

```
error[E0106]: missing lifetime specifier
-->
|
2 |     username: &str,
|               ^ expected lifetime parameter

error[E0106]: missing lifetime specifier
-->
|
3 |     email: &str,
|           ^ expected lifetime parameter
```

第十章会讲到如何修复这个问题以便在结构体中储存引用，不过现在，我们会使用像 `String` 这类拥有所有权的类型来替代 `&str` 这样的引用以修正这个错误。

一个使用结构体的示例程序

[ch05-02-example-structs.md](#)

commit d06a6a181fd61704cbf7feb55bc61d518c6469f9

为了理解何时会需要使用结构体，让我们编写一个计算长方形面积的程序。我们会从单独的变量开始，接着重构程序直到使用结构体替代他们为止。

使用 **Cargo** 来创建一个叫做 *rectangles* 的新二进制程序，它会获取一个长方形以像素为单位的长度和宽度并计算它的面积。示例 5-2 中是项目的 *src/main.rs* 文件中为此实现的一个小程序：

文件名: *src/main.rs*

```
fn main() {
    let length1 = 50;
    let width1 = 30;

    println!(
        "The area of the rectangle is {} square pixels.",
        area(length1, width1)
    );
}

fn area(length: u32, width: u32) -> u32 {
    length * width
}
```

示例 5-8：通过指定长方形的长宽变量来计算长方形面积

尝试使用 `cargo run` 运行程序：

```
The area of the rectangle is 1500 square pixels.
```

使用元组重构

虽然示例 5-8 可以运行，并调用 `area` 函数用长方形的每个维度来计算出面积，不过我们可以做的更好。长度和宽度是相关联的，因为他们在一起才能定义一个长方形。

这个做法的问题突显在 `area` 的签名上：

```
fn area(length: u32, width: u32) -> u32 {
```

函数 `area` 本应该计算一个长方形的面积，不过函数却有两个参数。这两个参数是相关联的，不过程序自身却哪里也没有表现出这一点。将长度和宽度组合在一起将更易懂也更易处理。第三章的“将值组合进元组”部分已经讨论过了一种可行的方法：元组。示例 5-9 是另一个使用元组的版本：

文件名: *src/main.rs*

```
fn main() {
    let rect1 = (50, 30);

    println!(
        "The area of the rectangle is {} square pixels.",
        area(rect1)
    );
}
```



```

    });
}

fn area(dimensions: (u32, u32)) -> u32 {
    dimensions.0 * dimensions.1
}

```

示例 5-8: 使用元组来指定长方形的长宽

在某种程度上说这样好一点了。元组帮助我们增加了一些结构性，现在在调用 `area` 的时候只用传递一个参数。不过在另一方面这个方法却更不明确了：元组并没有给出它元素的名称，所以计算变得更费解了，因为不得不使用索引来获取元组的每一部分：

在面积计算时混淆长宽并没有什么问题，不过当在屏幕上绘制长方形时就有问题了！我们将不得不记住元组索引 `0` 是 `length` 而 `1` 是 `width`。如果其他人要使用这些代码，他们也不得不搞清楚并记住他们。容易忘记或者混淆这些值而造成错误，因为我们没有表明代码中数据的意义。

使用结构体重构：增加更多意义

现在是引入结构体的时候了。我们可以将元组转换为一个有整体名称而且每个部分也有对应名字的数据类型，如示例 5-10 所示：

文件名: `src/main.rs`

```

struct Rectangle {
    length: u32,
    width: u32,
}

fn main() {
    let rect1 = Rectangle { length: 50, width: 30 };

    println!(
        "The area of the rectangle is {} square pixels.",
        area(&rect1)
    );
}

fn area(rectangle: &Rectangle) -> u32 {
    rectangle.length * rectangle.width
}

```

示例 5-10: 定义 `Rectangle` 结构体

这里我们定义了一个结构体并称其为 `Rectangle`。在 `{}` 中定义了字段 `length` 和 `width`，都是 `u32` 类型的。接着在 `main` 中，我们创建了一个长度为 50 和宽度为 30 的 `Rectangle` 的具体实例。

函数 `area` 现在被定义为接收一个名叫 `rectangle` 的参数，它的类型是一个结构体 `Rectangle` 实例的不可变借用。第四章讲到过，我们希望借用结构体而不是获取它的所有权这样 `main` 函数就可以保持 `rect1` 的所有权并继续使用它，所以这就是为什么在函数签名和调用的地方会有 `&`。

`area` 函数访问 `Rectangle` 的 `length` 和 `width` 字段。`area` 的签名现在明确的表明了我们的意图：通过其 `length` 和 `width` 字段，计算一个 `Rectangle` 的面积。这表明了长度和宽度是相互联系的，并为这些值提供了描述性的名称而不是使用元组的索引值 `0` 和 `1`。结构体胜在更清晰明了。

通过衍生 `trait` 增加实用功能

如果能够在调试程序时打印出 `Rectangle` 实例来查看其所有字段的值就更好了。示例 5-11 像往常一样使用 `println!` 宏：

文件名: `src/main.rs`

```
struct Rectangle {
    length: u32,
    width: u32,
}

fn main() {
    let rect1 = Rectangle { length: 50, width: 30 };

    println!("rect1 is {}", rect1);
}
```

示例 5-11: 尝试打印出 `Rectangle` 实例

如果运行代码，会出现带有如下核心信息的错误：

```
error[E0277]: the trait bound `Rectangle: std::fmt::Display` is not satisfied
```

`println!` 宏能处理很多类型的格式，不过，`{}`，默认告诉 `println!` 使用被称为 `Display` 的格式：直接提供给终端用户查看的输出。目前为止见过的基本类型都默认实现了 `Display`，因为它就是向用户展示 `1` 或其他任何基本类型的唯一方式。不过对于结构体，`println!` 应该用来输出的格式是不明确的，因为这有更多显示的可能性：是否需要逗号？需要打印出结构体的 `{}` 吗？所有字段都应该显示吗？因为这种不确定性，`Rust` 不尝试猜测我们的意图所以结构体并没有提供一个 `Display` 实现。

但是如果我们继续阅读错误，将会发现这个有帮助的信息：

```
note: `Rectangle` cannot be formatted with the default formatter; try using
`:?` instead if you are using a format string
```

让我们来试试！现在 `println!` 宏调用看起来像 `println!("rect1 is {:?}", rect1)`；这样。在 `{}` 中加入 `:?` 指示符告诉 `println!` 我们想要使用叫做 `Debug` 的输出格式。`Debug` 是一个 `trait`，它允许我们在调试代码时以一种对开发者有帮助的方式打印出结构体。

让我们试试运行这个变化。见鬼了！仍然能看到一个错误：

```
error: the trait bound `Rectangle: std::fmt::Debug` is not satisfied
```

不过编译器又一次给出了一个有帮助的信息！

```
note: `Rectangle` cannot be formatted using `:?`; if it is defined in your
crate, add `#[derive(Debug)]` or manually implement it
```

`Rust` 确实 包含了打印出调试信息的功能，不过我们必须为结构体显式选择这个功能。为此，在结构体定义之前加上 `#[derive(Debug)]` 注解，如示例 5-12 所示：

文件名: `src/main.rs`

```
#[derive(Debug)]
struct Rectangle {
    length: u32,
    width: u32,
```

```

}

fn main() {
    let rect1 = Rectangle { length: 50, width: 30 };

    println!("rect1 is {:?}", rect1);
}

```

示例 5-12: 增加注解来导出 `Debug` trait

现在再运行这个程序时，不会有任何错误并会出现如下输出：

```
rect1 is Rectangle { length: 50, width: 30 }
```

好极了！这并不是最漂亮的输出，不过它显示这个实例的所有字段，毫无疑问这对调试有帮助。如果想要输出再好看和易读一点，可以将 `println!` 的字符串中的 `{:?}",` 替换为 `{:#?}"`，这对更大的结构体会有帮助。如果在这个例子中使用了 `{:#?}"` 风格的话，输出会看起来像这样：

```
rect1 is Rectangle {
  length: 50,
  width: 30
}
```

Rust 为我们提供了很多可以通过 `derive` 注解来使用的 `trait`，他们可以为我们的自定义类型增加实用的行为。这些 `trait` 和行为在附录 C 中列出。第十章会涉及到如何通过自定义行为来实现这些 `trait`，同时还有如何创建你自己的 `trait`。

我们的 `area` 函数是非常特化的，它只是计算了长方形的面积。如果这个行为与 `Rectangle` 结构体再结合得更紧密一些就更好了，因为它不能用于其他类型。现在让我们看看如何继续重构这些代码，来将 `area` 函数协调进 `Rectangle` 类型定义的 `area` 方法中。

方法语法

ch05-03-method-syntax.md

commit 44bf3afd93519f8b0f900f21a5f2344d36e13448

方法与函数类似：它们使用 `fn` 关键字和名字声明，可以拥有参数和返回值，同时包含一段该方法在某处被调用时会执行的代码。不过方法与函数是不同的，因为它们在结构体的上下文中被定义（或者是枚举或 `trait` 对象的上下文，将分别在第六章和第十七章讲解），并且它们第一个参数总是 `self`，它代表调用该方法的结构体实例。

定义方法

让我们把前面实现的获取一个 `Rectangle` 实例作为参数的 `area` 函数，改写成一个定义于 `Rectangle` 结构体上的 `area` 方法，如示例 5-13 所示：

文件名: `src/main.rs`

```
#[derive(Debug)]
struct Rectangle {
    length: u32,
    width: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.length * self.width
    }
}

fn main() {
    let rect1 = Rectangle { length: 50, width: 30 };

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

示例 5-13: 在 `Rectangle` 结构体上定义 `area` 方法

为了使函数定义于 `Rectangle` 的上下文中，我们开始了一个 `impl` 块（`impl` 是 *implementation* 的缩写）。接着将函数移动到 `impl` 大括号中，并将签名中的第一个（在这里也是唯一一个）参数和函数体中其他地方的对应参数改成 `self`。然后在 `main` 中我们将我们先前调用 `area` 方法并传递 `rect1` 作为参数的地方，改成使用方法语法（*method syntax*）在 `Rectangle` 实例上调用 `area` 方法。方法语法获取一个实例并加上一个点号，后跟方法名、括号以及任何参数。

在 `area` 的签名中，开始使用 `&self` 来替代 `rectangle: &Rectangle`，因为该方法位于 `impl Rectangle` 上下文中所以 `Rust` 知道 `self` 的类型是 `Rectangle`。注意仍然需要在 `self` 前面加上 `&`，就像 `&Rectangle` 一样。方法可以选择获取 `self` 的所有权，或者像我们这里一样不可变地借用 `self`，或者可变地借用 `self`，就跟其他别的参数一样。

这里选择 `&self` 跟在函数版本中使用 `&Rectangle` 出于同样的理由：我们并不想获取所有权，只希望能够读取结构体中的数据，而不是写入。如果想要在方法中改变调用方法的实例，需要将第一个参数改为 `&mut self`。很少见到通过仅仅使用 `self` 作为第一个参数来使方法获取实例的所有权；这种技术通常用在当方法将 `self` 转换成别的实例的时候，这时我们想要防止调用者在转换之后使用原始的实例。

尽量使用方法替代函数，除了使用的方法语法和不需要在每个函数签名中重复 `self` 类型之外，其主要好处在于组织性。我们将某个类型实例能做的所有事情都一起放入 `impl` 块中，而不是让将来的用户在我们的代码中到处寻找 `Rectangle` 的功能。

-> 运算符到哪去了？

像在 C++ 这样的语言中，有两个不同的运算符来调用方法：`.` 直接在对象上调用方法，而 `->` 在一个对象的指针上调用方法，这时需要先解引用（*dereference*）指针。换句话说，如果 `object` 是一个指针，那么 `object->something()` 就像 `(*object).something()` 一样。

Rust 并没有一个与 `->` 等效的运算符；相反，Rust 有一个叫自动引用和解引用（*automatic referencing and dereferencing*）的功能。方法调用是 Rust 中少数几个拥有这种行为的地方。

这是它如何工作的：当使用 `object.something()` 调用方法时，Rust 会自动增加 `&`、`&mut` 或 `*` 以便使 `object` 符合方法的签名。也就是说，这些代码是等同的：

```
#[derive(Debug, Copy, Clone)]
struct Point {
    x: f64,
    y: f64,
}

impl Point {
    fn distance(&self, other: &Point) -> f64 {
        let x_squared = f64::powi(other.x - self.x, 2);
        let y_squared = f64::powi(other.y - self.y, 2);
        f64::sqrt(x_squared + y_squared)
    }
}

let p1 = Point { x: 0.0, y: 0.0 };
let p2 = Point { x: 5.0, y: 6.5 };
p1.distance(&p2);
(&p1).distance(&p2);
```

第一行看起来简洁的多。这种自动解引用的行为之所以能行得通是因为方法有一个明确的接收者——`self` 类型。在给出接收者和方法名的前提下，Rust 可以明确地计算出方法是仅仅读取（`&self`），做出修改（`&mut self`）或者是获取所有权（`self`）。Rust 这种使得借用对方法接收者来说是隐式的做法是其所有权系统程序员友好性实践的一大部分。

带有更多参数的方法

让我们更多的实践一下方法，通过为 `Rectangle` 结构体实现第二个方法。这回，我们让一个 `Rectangle` 的实例获取另一个 `Rectangle` 实例并返回 `self` 能否完全包含第二个长方形，如果能则返回 `true`，如果不能则返回 `false`。一旦定义了 `can_hold` 方法，就可以运行示例 5-14 中的代码了：

文件名: `src/main.rs`

```
fn main() {
    let rect1 = Rectangle { length: 50, width: 30 };
    let rect2 = Rectangle { length: 40, width: 10 };
    let rect3 = Rectangle { length: 45, width: 60 };

    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
}
```

示例 5-14: 展示还未实现的 `can_hold` 方法的应用

我们希望看到如下输出，因为 `rect2` 的长宽都小于 `rect1`，而 `rect3` 比 `rect1` 要宽：

```
Can rect1 hold rect2? true
Can rect1 hold rect3? false
```

因为我们想定义一个方法，所以它应该位于 `impl Rectangle` 块中。方法名是 `can_hold`，并且它会获取另一个 `Rectangle` 的不可变借用作为参数。通过观察调用位置的代码可以看出参数是什么类型的：`rect1.can_hold(&rect2)` 传入了 `&rect2`，它是一个 `Rectangle` 的实例 `rect2` 的不可变借用。这是可以理解的，因为我们只需要读取 `rect2`（而不是写入，这意味着我们需要一个可变借用）而且希望 `main` 保持 `rect2` 的所有权这样就可以在调用这个方法后继续使用它。`can_hold` 的返回值是一个布尔值，其实现会分别检查 `self` 的长宽是否都大于另一个 `Rectangle`。让我们在示例 5-13 的 `impl` 块中增加这个新方法，如示例 5-15 所示：

文件名: `src/main.rs`

```
# #[derive(Debug)]
# struct Rectangle {
#     length: u32,
#     width: u32,
# }
#
impl Rectangle {
    fn area(&self) -> u32 {
        self.length * self.width
    }

    fn can_hold(&self, other: &Rectangle) -> bool {
        self.length > other.length && self.width > other.width
    }
}
```

示例 5-15: 在 `Rectangle` 上实现 `can_hold` 方法，它获取另一个 `Rectangle` 实例作为参数

如果结合示例 5-14 的 `main` 函数来运行，就会看到想要得到的输出。方法可以在 `self` 后增加多个参数，而且这些参数就像函数中的参数一样工作。

关联函数

`impl` 块的另一个有用的功能是：允许在 `impl` 块中定义不以 `self` 作为参数的函数。这被称为关联函数（*associated functions*），因为它们与结构体相关联。即便如此它们仍是函数而不是方法，因为它们并不作用于一个结构体的实例。你已经使用过一个关联函数了：`String::from`。

关联函数经常被用作返回一个结构体新实例的构造函数。例如我们可以提供一个关联函数，它接受一个维度参数并且同时用来作为长和宽，这样可以更轻松的创建一个正方形 `Rectangle` 而不必指定两次同样的值：

Filename: `src/main.rs`

```
# #[derive(Debug)]
# struct Rectangle {
#     length: u32,
#     width: u32,
# }
#
impl Rectangle {
    fn square(size: u32) -> Rectangle {
        Rectangle { length: size, width: size }
    }
}
```

```
}
```

使用结构体名和 `::` 语法来调用这个关联函数：比如 `let sq = Rectangle::square(3);`。这个方法位于结构体的命名空间中：`::` 语法用于关联函数和模块创建的命名空间，第七章会讲到模块。

多个 `impl` 块

每个结构体都允许拥有多个 `impl` 块。例如，示例 5-15 等同于示例 5-16 的代码，这里每个方法有其自己的 `impl` 块：

```
# #[derive(Debug)]
# struct Rectangle {
#     length: u32,
#     width: u32,
# }
#
impl Rectangle {
    fn area(&self) -> u32 {
        self.length * self.width
    }
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.length > other.length && self.width > other.width
    }
}
```

示例 5-16：使用多个 `impl` 块重写示例 5-15

总结

结构体让我们可以在自己的范围内创建有意义的自定义类型。通过结构体，我们可以将相关联的数据片段联系起来并命名它们，这样可以使得代码更加清晰。方法允许为结构体实例指定行为，而关联函数将特定功能置于结构体的命名空间中并且无需一个实例。

结构体并不是创建自定义类型的唯一方法；让我们转向 **Rust** 的枚举功能并为自己的工具箱再添一个工具。

枚举和模式匹配

[ch06-00-enums.md](#)

commit 4f2dc564851dc04b271a2260c834643dfd86c724

本章介绍 枚举 (*enumerations*)，也被称作 *enums*。枚举允许你通过列举可能的值来定义一个类型。首先，我们会定义并使用一个枚举来展示它是如何连同数据一起编码信息的。接下来，我们会探索一个特别有用的枚举，叫做 `Option`，它代表一个值要么是某个值要么什么都不是。然后会讲到在 `match` 表达式中用模式匹配，针对不同的枚举值编写相应要执行的代码。最后会涉及到 `if let`，另一个简洁方便处理代码中枚举的结构。

枚举是一个很多语言都有的功能，不过不同语言中的功能各不相同。`Rust` 的枚举与 `F#`、`OCaml` 和 `Haskell` 这样的函数式编程语言中的 代数数据类型 (*algebraic data types*) 最为相似。

定义枚举

ch06-01-defining-an-enum.md

commit d06a6a181fd61704cbf7feb55bc61d518c6469f9

让我们在一个实际场景中看看在这个特定场景下，使用枚举比使用结构体更合适。假设我们要处理 IP 地址。目前被广泛使用的两个主要 IP 标准：IPv4（version four）和 IPv6（version six）。这是我们的程序可能会遇到的所有可能的 IP 地址类型：所以可以枚举出所有可能的值，这也正是它名字的由来。

任何一个 IP 地址要么是 IPv4 的要么是 IPv6 的，而且不能两者都是。IP 地址的这个特性使得枚举数据结构非常适合这个场景，因为枚举值只可能是其中一个成员。IPv4 和 IPv6 从根本上讲仍是 IP 地址，所以当代码在处理申请任何类型的 IP 地址的场景时应该把它们当作相同的类型。

可以通过在代码中定义一个 `IpAddrKind` 枚举来表现这个概念并列出可能的 IP 地址类型，`v4` 和 `v6`。这被称为枚举的成员（*variants*）：

```
enum IpAddrKind {  
    V4,  
    V6,  
}
```

现在 `IpAddrKind` 就是一个可以在代码中使用的自定义类型了。

枚举值

可以像这样创建 `IpAddrKind` 两个不同成员的实例：

```
# enum IpAddrKind {  
#     V4,  
#     V6,  
# }  
#  
let four = IpAddrKind::V4;  
let six = IpAddrKind::V6;
```

注意枚举的成员位于其标识符的命名空间中，并使用两个冒号分开。这么设计的益处是现在 `IpAddrKind::V4` 和 `IpAddrKind::V6` 都是 `IpAddrKind` 类型的。例如，接着可以定义一个函数来获取任何 `IpAddrKind`：

```
# enum IpAddrKind {  
#     V4,  
#     V6,  
# }  
#  
fn route(ip_type: IpAddrKind) { }
```

现在可以使用任意成员来调用这个函数：

```
# enum IpAddrKind {  
#     V4,  
#     V6,  
# }  
#  
# fn route(ip_type: IpAddrKind) { }
```

```
#
route(IPAddrKind::V4);
route(IPAddrKind::V6);
```

使用枚举甚至还有更多优势。进一步考虑一下我们的 IP 地址类型，目前没有一个储存实际 IP 地址数据的方法；只知道它是什么类型的。考虑到已经在第五章学习过结构体了，你可能会像示例 6-1 那样修改这个问题：

```
enum IPAddrKind {
    V4,
    V6,
}

struct IPAddr {
    kind: IPAddrKind,
    address: String,
}

let home = IPAddr {
    kind: IPAddrKind::V4,
    address: String::from("127.0.0.1"),
};

let loopback = IPAddr {
    kind: IPAddrKind::V6,
    address: String::from("::1"),
};
```

示例 6-1：将 IP 地址的数据和 `IPAddrKind` 成员储存在一个 `struct` 中

这里我们定义了一个有两个字段的结构体 `IPAddr`：`kind` 字段是 `IPAddrKind`（之前定义的枚举）类型的而 `address` 字段是 `String` 类型的。这里有两个结构体的实例。第一个，`home`，它的 `kind` 的值是 `IPAddrKind::V4` 与之相关联的地址数据是 `127.0.0.1`。第二个实例，`loopback`，`kind` 的值是 `IPAddrKind` 的另一个成员，`V6`，关联的地址是 `::1`。我们使用了一个结构体来将 `kind` 和 `address` 打包在一起，现在枚举成员就与值相关联了。

我们可以使用一种更简洁的方式来表达相同的概念，仅仅使用枚举并将数据直接放进每一个枚举成员而不是将枚举作为结构体的一部分。`IPAddr` 枚举的新定义表明了 `V4` 和 `V6` 成员都关联了 `String` 值：

```
enum IPAddr {
    V4(String),
    V6(String),
}

let home = IPAddr::V4(String::from("127.0.0.1"));

let loopback = IPAddr::V6(String::from("::1"));
```

我们直接将数据附加到枚举的每个成员上，这样就不需要一个额外的结构体了。

用枚举替代结构体还有另一个优势：每个成员可以处理不同类型和数量的数据。`IPv4` 版本的 IP 地址总是含有四个值在 0 和 255 之间的数字部分。如果我们想要将 `v4` 地址储存为四个 `u8` 值而 `v6` 地址仍然表现为一个 `String`，这就不能使用结构体了。枚举可以轻易处理的这个情况：

```
enum IPAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}

let home = IPAddr::V4(127, 0, 0, 1);
```

```
let loopback = IpAddr::V6(String::from("::1"));
```

这些代码展示了使用枚举来储存两种不同 IP 地址的几种可能的选择。然而，事实证明储存和编码 IP 地址实在是太常见了以致标准库提供了一个可供使用的定义！让我们看看标准库如何定义 `IpAddr` 的：它正有着跟我们定义和使用的一样的枚举和成员，不过它将成员中的地址数据嵌入到了两个不同形式的结构体中，它们对不同的成员的定义是不同的：

```
struct Ipv4Addr {  
    // details elided  
}  
  
struct Ipv6Addr {  
    // details elided  
}  
  
enum IpAddr {  
    V4(Ipv4Addr),  
    V6(Ipv6Addr),  
}
```

这些代码展示了可以将任意类型的数据放入枚举成员中：例如字符串、数字类型或者结构体。甚至可以包含另一个枚举！另外，标准库中的类型通常并不比你设想出来的要复杂多少。

注意虽然标准库中包含一个 `IpAddr` 的定义，仍然可以创建和使用我们自己的定义而不会有冲突，因为我们并没有将标准库中的定义引入作用域。第七章会讲到如何导入类型。

来看看示例 6-2 中的另一个枚举的例子：它的成员中内嵌了多种多样的类型：

```
enum Message {  
    Quit,  
    Move { x: i32, y: i32 },  
    Write(String),  
    ChangeColor(i32, i32, i32),  
}
```

示例 6-2：一个 `Message` 枚举，其每个成员都储存了不同数量和类型的值

这个枚举有四个含有不同类型的成员：

- `Quit` 没有关联任何数据。
- `Move` 包含一个匿名结构体
- `Write` 包含单独一个 `String`。
- `ChangeColor` 包含三个 `i32`。

定义一个像示例 6-2 中的枚举类似于定义不同类型的结构体，除了枚举不使用 `struct` 关键字而且所有成员都被组合在一起位于 `Message` 下之外。如下这些结构体可以包含与之前枚举成员中相同的数据：

```
struct QuitMessage; // unit struct  
struct MoveMessage {  
    x: i32,  
    y: i32,  
}  
struct WriteMessage(String); // tuple struct  
struct ChangeColorMessage(i32, i32, i32); // tuple struct
```

不过如果我们使用不同的结构体，它们都有不同的类型，将不能轻易的定义一个获取任何这些信息类型的函数，正如可以使用示例 6-2 中定义的 `Message` 枚举那样，因为它们是一个类型的。

结构体和枚举还有另一个相似点：就像可以使用 `impl` 来为结构体定义方法那样，也可以在枚举上定义方法。这是一个定义于我们 `Message` 枚举上的叫做 `call` 的方法：

```
# enum Message {
#     Quit,
#     Move { x: i32, y: i32 },
#     Write(String),
#     ChangeColor(i32, i32, i32),
# }
#
impl Message {
    fn call(&self) {
        // method body would be defined here
    }
}

let m = Message::Write(String::from("hello"));
m.call();
```

方法体使用了 `self` 来获取调用方法的值。这个例子中，创建了一个拥有类型 `Message::Write("hello")` 的变量 `m`，而且这就是当 `m.call()` 运行时 `call` 方法中的 `self` 的值。

让我们看看标准库中的另一个非常常见且实用的枚举：`Option`。

Option 枚举和其相对于空值的优势

在之前的部分，我们看到了 `IpAddr` 枚举如何利用 `Rust` 的类型系统编码更多信息而不单单是程序中的数据。接下来我们分析一个 `Option` 的案例，`Option` 是标准库定义的另一个枚举。`Option` 类型应用广泛因为它编码了一个非常普遍的场景，即一个值要么是某个值要么什么都不是。从类型系统的角度来表达这个概念就意味着编译器需要检查是否处理了所有应该处理的情况，这样就可以避免在其他编程语言中非常常见的 `bug`。

编程语言的设计经常从其包含功能的角度考虑问题，但是从其所排除在外的功能的角度思考也很重要。`Rust` 并没有很多其他语言中有的空值功能。空值（`Null`）是一个值，它代表没有值。在有空值的语言中，变量总是这两种状态之一：空值和非空值。

在“`Null References: The Billion Dollar Mistake`”中，`Tony Hoare`，`null` 的发明者，曾经说到：

I call it my billion-dollar mistake. At that time, I was designing the first comprehensive type system for references in an object-oriented language. My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

我称之为我十亿美元的错误。当时，我在为一个面向对象语言设计第一个综合性的面向引用的类型系统。我的目标是通过编译器的自动检查来保证所有引用的使用都应该是绝对安全的。不过我未能抗拒引入一个空引用的诱惑，仅仅是因为它是这么的容易实现。这引发了无数错误、漏洞和系统崩溃，在之后的四十多年中造成了数十亿美元的苦痛和伤害。

空值的问题在于当你尝试像一个非空值那样使用一个空值，会出现某种形式的错误。因为空和非空的属性是无处不在的，非常容易出现这类错误。

然而，空值尝试表达的概念仍然是有意义的：空值是一个因为某种原因目前无效或缺失的值。

问题不在于具体的概念而在于特定的实现。为此，`Rust` 并没有空值，不过它确实拥有一个可以编码存在或不存在概念的枚举。这个枚举是 `Option<T>`，而且它[定义于标准库中](#)，如下：

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

`Option<T>` 是如此有用以至于它甚至被包含在了 `prelude` 之中，这意味着我们不需要显式导入它。另外，它的成员也是如此，可以不需要 `Option::` 前缀来直接使用 `Some` 和 `None`。即便如此 `Option<T>` 也仍是常规的枚举，`Some(T)` 和 `None` 仍是 `Option<T>` 的成员。

`<T>` 语法是一个我们还未讲到的 **Rust** 功能。它是一个泛型类型参数，第十章会更详细的讲解泛型。目前，所有你需要知道的就是 `<T>` 意味着 `Option` 枚举的 `Some` 成员可以包含任意类型的数据。这里是一些包含数字类型和字符串类型 `Option` 值的例子：

```
let some_number = Some(5);
let some_string = Some("a string");

let absent_number: Option<i32> = None;
```

如果使用 `None` 而不是 `Some`，需要告诉 **Rust** `Option<T>` 是什么类型的，因为编译器只通过 `None` 值无法推断出 `Some` 变量保留的值的类型。

当有一个 `Some` 值时，我们就知道存在一个值，而这个值保存在 `Some` 中。当有个 `None` 值时，在某种意义上它跟空值是相同的意义：并没有一个有效的值。那么，`Option<T>` 为什么就比空值要好呢？

简而言之，因为 `Option<T>` 和 `T`（这里 `T` 可以是任何类型）是不同的类型，编译器不允许像一个被定义的有效类型那样使用 `Option<T>`。例如，这些代码不能编译，因为它尝试将 `Option<i8>` 与 `i8` 相加：

```
let x: i8 = 5;
let y: Option<i8> = Some(5);

let sum = x + y;
```

如果运行这些代码，将得到类似这样的错误信息：

```
error[E0277]: the trait bound `i8: std::ops::Add<std::option::Option<i8>>` is
not satisfied
-->
|
7 | let sum = x + y;
|             ^^^^^
|
```

哇哦！事实上，错误信息意味着 **Rust** 不知道该如何将 `Option<i8>` 与 `i8` 相加。当在 **Rust** 中拥有一个像 `i8` 这样类型的值时，编译器确保它总是有一个有效的值。我们可以自信使用而无需判空。只有当使用 `Option<i8>`（或者任何用到的类型）的时候需要担心可能没有一个值，而编译器会确保我们在处理值之前处理为空的情况。

换句话说，在对 `Option<T>` 进行 `T` 的运算之前必须转换为 `T`。通常这能帮助我们捕获空值最常见的问题之一：假设某值不为空但实际上为空的情况。

无需担心错过存在非空值的假设让我们对代码更加有信心，为了拥有一个可能为空的值，必须显式的将其放入对应类型的 `Option<T>` 中。接着，当使用这个值时，必须明确的处理值为空的情况。任何地方一个值不是 `Option<T>` 类型的话，可以安全的假设它的值不为空。这是 **Rust** 的一个有意为之的设计选择，来限制空值的泛滥和增加 **Rust** 代码的安全性。

那么当有一个 `Option<T>` 的值时，如何从 `Some` 成员中取出 `T` 的值来使用它呢？`Option<T>` 枚举拥有大量用于各种情况的方法：你可以查看[相关代码](#)。熟悉 `Option<T>` 的方法将对你的 **Rust** 之旅提供巨大的帮助。

总的来说，为了使用 `Option<T>` 值，需要编写处理每个成员的代码。我们想要一些代码只当拥有 `Some(T)` 值时运行，这些代码允许使用其中的 `T`。也希望一些代码当在 `None` 值时运行，这些代码并没有一个可用的 `T` 值。`match` 表达式就是这么一个处理枚举的控制流结构：它会根据枚举的成员运行不同的代码，这些代码可以使用匹配到的值中的数据。

match 控制流运算符

[ch06-02-match.md](#)

commit 01dd4248621c2f510947592e47d16bdab9b14cf0

Rust 有一个叫做 `match` 的极为强大的控制流运算符，它允许我们将一个值与一系列的模式相比较并根据匹配的模式执行相应代码。模式可由字面值、变量、通配符和许多其他内容构成；第十八章会涉及到所有不同种类的模式以及它们的作用。`match` 的力量来源于模式的表现力以及编译器检查，它确保了所有可能的情况都得到处理。

可以把 `match` 表达式想象成某种硬币分类器：硬币滑入有着不同大小孔洞的轨道，每一个硬币都会掉入符合它大小的孔洞。同样地，值也会检查 `match` 的每一个模式，并且在遇到第一个“符合”的模式时，值会进入相关联的代码块并在执行中被使用。

因为刚刚提到了硬币，让我们用它们来作为一个使用 `match` 的例子！我们可以编写一个函数来获取一个未知的（美帝）硬币，并以一种类似验钞机的方式，确定它是何种硬币并返回它的美分值，如示例 6-3 中所示：

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u32 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

示例 6-3：一个枚举和一个以枚举成员作为模式的 `match` 表达式

拆开 `value_in_cents` 函数中的 `match` 来看。首先，我们列出 `match` 关键字后跟一个表达式，在这个例子中是 `coin` 的值。这看起来非常像 `if` 使用的表达式，不过这里有一个非常大的区别：对于 `if`，表达式必须返回一个布尔值。而这里它可以是任何类型的。例子中的 `coin` 的类型是示例 6-3 中定义的 `Coin` 枚举。

接下来是 `match` 的分支。一个分支有两个部分：一个模式和一些代码。第一个分支的模式是值 `Coin::Penny` 而之后的 `=>` 运算符将模式和将要运行的代码分开。这里的代码就仅仅是值 `1`。每一个分支之间使用逗号分隔。

当 `match` 表达式执行时，它将结果值按顺序与每一个分支的模式相比较，如果模式匹配了这个值，这个模式相关联的代码将被执行。如果模式并不匹配这个值，将继续执行下一个分支，非常类似一个硬币分类器。可以拥有任意多的分支：示例 6-3 中的 `match` 有四个分支。

每个分支相关联的代码是一个表达式，而表达式的结果值将作为整个 `match` 表达式的返回值。

如果分支代码较短的话通常不使用大括号，正如示例 6-3 中的每个分支都只是返回一个值。如果想要在分支中运行多行代码，可以使用大括号。例如，如下代码在每次使用 `Coin::Penny` 调用时都会打印出“Lucky penny!”，同时仍然返回代码块最后的值，`1`：

```
# enum Coin {
#     Penny,
#     Nickel,
#     Dime,
#     Quarter,
```

```
# }
#
fn value_in_cents(coin: Coin) -> u32 {
    match coin {
        Coin::Penny => {
            println!("Lucky penny!");
            1
        },
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

绑定值的模式

匹配分支的另一个有用的功能是可以绑定匹配的模式的部分值。这也就是如何从枚举成员中提取值。

作为一个例子，让我们修改枚举的一个成员来存放数据。1999 年到 2008 年间，美帝在 25 美分的硬币的一侧为 50 个州的每一个都印刷了不同的设计。其他的硬币都没有这种区分州的设计，所以只有这些 25 美分硬币有特殊的价值。可以将这些信息加入我们的 `enum`，通过改变 `Quarter` 成员来包含一个 `State` 值，示例 6-4 中完成了这些修改：

```
#[derive(Debug)] // So we can inspect the state in a minute
enum UsState {
    Alabama,
    Alaska,
    // ... etc
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}
```

示例 6-4: `Quarter` 成员也存放了一个 `UsState` 值的 `Coin` 枚举

想象一下我们的一个朋友尝试收集所有 50 个州的 25 美分硬币。在根据硬币类型分类零钱的同时，也可以报告出每个 25 美分硬币所对应的州名称，这样如果我们的朋友没有的话，他可以把它加入收藏。

在这些代码的匹配表达式中，我们在匹配 `Coin::Quarter` 成员的分支的模式中增加了一个叫做 `state` 的变量。当匹配到 `Coin::Quarter` 时，变量 `state` 将会绑定 25 美分硬币所对应州的值。接着在那个分支的代码中使用 `state`，如下：

```
# #[derive(Debug)]
# enum UsState {
#     Alabama,
#     Alaska,
# }
#
# enum Coin {
#     Penny,
#     Nickel,
#     Dime,
#     Quarter(UsState),
# }
#
fn value_in_cents(coin: Coin) -> u32 {
```



```
match coin {
  Coin::Penny => 1,
  Coin::Nickel => 5,
  Coin::Dime => 10,
  Coin::Quarter(state) => {
    println!("State quarter from {:?}!", state);
    25
  },
}
}
```

如果调用 `value_in_cents(Coin::Quarter(UsState::Alaska))`，`coin` 将是 `Coin::Quarter(UsState::Alaska)`。当将值与每个分支相比较时，没有分支会匹配，直到遇到 `Coin::Quarter(state)`。这时，`state` 绑定的将会是值 `UsState::Alaska`。接着就可以在 `println!` 表达式中使用这个绑定了，像这样就可以获取 `Coin` 枚举的 `Quarter` 成员中内部的州的值。

匹配 `Option<T>`

在之前的部分在使用 `Option<T>` 时我们想要从 `Some` 中取出其内部的 `T` 值；也可以像处理 `Coin` 枚举那样使用 `match` 处理 `Option<T>`！与其直接比较硬币，我们将比较 `Option<T>` 的成员，不过 `match` 表达式的工作方式保持不变。

比如我们想要编写一个函数，它获取一个 `Option<i32>` 并且如果其中有一个值，将其加一。如果其中没有值，函数应该返回 `None` 值并不尝试执行任何操作。

得益于 `match`，编写这个函数非常简单，它将看起来像示例 6-5 中这样：

```
fn plus_one(x: Option<i32>) -> Option<i32> {
  match x {
    None => None,
    Some(i) => Some(i + 1),
  }
}

let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

示例 6-5：一个在 `Option<i32>` 上使用 `match` 表达式的函数

匹配 `Some(T)`

让我们更仔细的检查 `plus_one` 的第一行操作。当调用 `plus_one(five)` 时，`plus_one` 函数体中的 `x` 将会是值 `Some(5)`。接着将其与每个分支比较。

```
None => None,
```

值 `Some(5)` 并不匹配模式 `None`，所以继续进行下一个分支。

```
Some(i) => Some(i + 1),
```

`Some(5)` 与 `Some(i)` 匹配吗？为什么不呢！它们是相同的成员。`i` 绑定了 `Some` 中包含的值，所以 `i` 的值是 `5`。接着匹配分支的代码被执行，所以我们将 `i` 的值加一并返回一个含有值 `6` 的新 `Some`。

匹配 `None`

接着考虑下示例 6-5 中 `plus_one` 的第二个调用，这里 `x` 是 `None`。我们进入 `match` 并与第一个分支相比较。

```
None => None,
```

匹配上了！这里没有值来加一，所以程序结束并返回 `=>` 右侧的值 `None`，因为第一个分支就匹配到了，其他的分支将不再比较。

将 `match` 与枚举相结合在很多场景中都是有用的。你会在 **Rust** 代码中看到很多这样的模式：`match` 一个枚举，绑定其中的值到一个变量，接着根据其值执行代码。这在一开始有点复杂，不过一旦习惯了，你会希望所有语言都拥有它！这一直是用户的最爱。

匹配是穷尽的

`match` 还有另一方面需要讨论。考虑一下 `plus_one` 函数的这个版本：

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        Some(i) => Some(i + 1),
    }
}
```

我们没有处理 `None` 的情况，所以这些代码会造成一个 **bug**。幸运的是，这是一个 **Rust** 知道如何处理的 **bug**。如果尝试编译这段代码，会得到这个错误：

```
error[E0004]: non-exhaustive patterns: `None` not covered
-->
|
6 |         match x {
|           ^ pattern `None` not covered
```

Rust 知道我们没有覆盖所有可能的情况甚至知道那些模式被忘记了！**Rust** 中的匹配是 穷尽的（*exhaustive*）：必须穷举到最后的可能性来使代码有效。特别的在这个 `Option<T>` 的例子中，**Rust** 防止我们忘记明确的处理 `None` 的情况，这使我们免于假设拥有一个实际上为空的值，这造成了之前提到过的价值亿万的错误。

_ 通配符

Rust 也提供了一个模式用于不想列举出所有可能值的场景。例如，`u8` 可以拥有 0 到 255 的有效的值，如果我们只关心 1、3、5 和 7 这几个值，就并不想必须列出 0、2、4、6、8、9 一直到 255 的值。所幸我们不必这么做：可以使用特殊的模式 `_` 替代：

```
let some_u8_value = 0u8;
match some_u8_value {
    1 => println!("one"),
    3 => println!("three"),
    5 => println!("five"),
    7 => println!("seven"),
    _ => (),
}
```

`_` 模式会匹配所有的值。通过将其放置于其他分支之后，`_` 将会匹配所有之前没有指定的可能的值。`()` 就是 `unit` 值，所以 `_` 的情况什么也不会发生。因此，可以说我们想要对 `_` 通配符之前没有列出的所有可能的值不做任何处理。

然而，`match` 在只关心一个情况的场景中可能就有有点啰嗦了。为此 `Rust` 提供了 `if let`。

if let 简单控制流

ch06-03-if-let.md

commit 3f2a1bd8dbb19cc48b210fc4fb35c305c8d81b56

`if let` 语法让我们以一种不那么冗长的方式结合 `if` 和 `let`，来处理匹配一个模式的值而忽略其他的值。考虑示例 6-6 中的程序，它匹配一个 `Option<u8>` 值并希望当值为三时执行代码：

```
let some_u8_value = Some(0u8);
match some_u8_value {
    Some(3) => println!("three"),
    _ => (),
}
```

示例 6-6: `match` 只关心当值为 `Some(3)` 时执行代码

我们想要对 `Some(3)` 匹配进行操作不过不想处理任何其他 `Some<u8>` 值或 `None` 值。为了满足 `match` 表达式（穷尽性）的要求，必须在处理完这唯一的成员后加上 `_ => ()`，这样也要增加很多样板代码。

不过我们可以使用 `if let` 这种更短的方式编写。如下代码与示例 6-6 中的 `match` 行为一致：

```
# let some_u8_value = Some(0u8);
if let Some(3) = some_u8_value {
    println!("three");
}
```

`if let` 获取通过 `=` 分隔的一个模式和一个表达式。它的工作方式与 `match` 相同，这里的表达式对应 `match` 而模式则对应第一个分支。

使用 `if let` 意味着编写更少代码，更少的缩进和更少的样板代码。然而，这样会失去 `match` 强制要求的穷尽性检查。`match` 和 `if let` 之间的选择依赖特定的环境以及增加简洁度和失去穷尽性检查的权衡取舍。

换句话说，可以认为 `if let` 是 `match` 的一个语法糖，它当值匹配某一模式时执行代码而忽略所有其他值。

可以在 `if let` 中包含一个 `else`。 `else` 块中的代码与 `match` 表达式中的 `_` 分支块中的代码相同，这样的 `match` 表达式就等同于 `if let` 和 `else`。回忆一下示例 6-4 中 `Coin` 枚举的定义，它的 `Quarter` 成员包含一个 `UsState` 值。如果想要计数所有不是 25 美分的硬币的同时也报告 25 美分硬币所属的州，可以使用这样一个 `match` 表达式：

```
# #[derive(Debug)]
# enum UsState {
#     Alabama,
#     Alaska,
# }
#
# enum Coin {
#     Penny,
#     Nickel,
#     Dime,
#     Quarter(UsState),
# }
# let coin = Coin::Penny;
let mut count = 0;
match coin {
    Coin::Quarter(state) => println!("State quarter from {:?}!", state),
    _ => count += 1,
}
```

```
}
```

或者可以使用这样的 `if let` 和 `else` 表达式：

```
# #[derive(Debug)]
# enum UsState {
#     Alabama,
#     Alaska,
# }
#
# enum Coin {
#     Penny,
#     Nickel,
#     Dime,
#     Quarter(UsState),
# }
# let coin = Coin::Penny;
let mut count = 0;
if let Coin::Quarter(state) = coin {
    println!("State quarter from {:?}!", state);
} else {
    count += 1;
}
```

如果你的程序遇到一个使用 `match` 表达起来过于啰嗦的逻辑，记住 `if let` 也在你的 **Rust** 工具箱中。

总结

现在我们涉及到了如何使用枚举来创建有一系列可列举值的自定义类型。我们也展示了标准库的 `Option<T>` 类型是如何帮助你利用类型系统来避免出错的。当枚举值包含数据时，你可以根据需要处理多少情况来选择使用 `match` 或 `if let` 来获取并使用这些值。

你的 **Rust** 程序现在能够使用结构体和枚举在自己的作用域内表现其内容了。在你的 **API** 中使用自定义类型保证了类型安全：编译器会确保你的函数只会得到它期望的类型的值。

为了向你的用户提供一个组织良好的 **API**，它使用起来很直观并且只向用户暴露他们确实需要的部分，那么现在就让我们转向 **Rust** 的模块系统吧。

使用模块组织和复用代码

ch07-00-modules.md

commit b707dc664960f0ffc495c373900d6b13e434927d

在你刚开始编写 **Rust** 程序时，代码可能仅仅位于 `main` 函数里。随着代码数量的增长，为了复用和更好地组织代码，最终你会将功能移动到其他函数中。通过将代码分隔成更小的块，每一个块代码自身就更易于理解。不过当你发现自己有太多的函数了该怎么办呢？**Rust** 有一个模块系统可以有组织地复用代码。

就跟你将代码行提取到一个函数中一样，也可以将函数（和其他类似结构体和枚举的代码）提取到不同模块中。模块（*module*）是一个包含函数或类型定义的命名空间，你可以选择这些定义能（公有）或不能（私有）在其模块外可见。下面是一个模块如何工作的梗概：

- 使用 `mod` 关键字声明新模块。此模块的代码要么直接位于声明之后的大括号中，要么位于另一个文件。
- 函数、类型、常量和模块默认都是私有的。可以使用 `pub` 关键字将其变成公有并在命名空间之外可见。
- `use` 关键字将模块或模块中的定义引入到作用域中以便于引用它们。

我们会逐一了解这每一部分并学习如何将它们结合在一起。

mod 和文件系统

[ch07-01-mod-and-the-filesystem.md](#)

commit c6a9e77a1b1ed367e0a6d5dcd222589ad392a8ac

我们将通过使用 **Cargo** 创建一个新项目来开始我们的模块之旅，不过这次不再创建一个二进制 **crate**，而是创建一个库 **crate**：一个其他人可以作为依赖导入的项目。第二章猜猜看游戏中作为依赖使用的 **rand** 就是这样的 **crate**。

我们将创建一个提供一些通用网络功能的项目的骨架结构；我们将专注于模块和函数的组织，而不担心函数体中的具体代码。这个项目叫做 **communicator**。**Cargo** 默认会创建一个库 **crate** 除非指定其他项目类型，所以如果不像一直以来那样加入 **--bin** 参数则项目将会是一个库：

```
$ cargo new communicator
$ cd communicator
```

注意 **Cargo** 生成了 **src/lib.rs** 而不是 **src/main.rs**。在 **src/lib.rs** 中我们会找到这些：

文件名: **src/lib.rs**

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

Cargo 创建了一个空的测试来帮助我们开始库项目，不像使用 **--bin** 参数那样创建一个“Hello, world!”二进制项目。在本章之后的“使用 **super** 访问父模块”部分会介绍 **#[]** 和 **mod tests** 语法，目前只需确保它们位于 **src/lib.rs** 底部即可。

因为没有 **src/main.rs** 文件，所以没有可供 **Cargo** 的 **cargo run** 执行的东西。因此，我们将使用 **cargo build** 命令只是编译库 **crate** 的代码。

我们将学习根据编写代码的意图来以不同方法组织库项目代码以适应多种情况。

模块定义

对于 **communicator** 网络库，首先要定义一个叫做 **network** 的模块，它包含一个叫做 **connect** 的函数定义。**Rust** 中所有模块的定义以关键字 **mod** 开始。在 **src/lib.rs** 文件的开头在测试代码的上面增加这些代码：

文件名: **src/lib.rs**

```
mod network {
    fn connect() {
    }
}
```

mod 关键字的后面是模块的名字，**network**，接着是位于大括号中的代码块。代码块中的一切都位于 **network** 命名空间中。在这个例子中，只有一个函数，**connect**。如果想要在 **network** 模块外面的代码中调用这个函数，需要指定模块名并使用命名空间语法 **::**，像这样：**network::connect()**，而不是只是 **connect()**。

也可以在 `src/lib.rs` 文件中同时存在多个模块。例如，再拥有一个 `client` 模块，它也有一个叫做 `connect` 的函数，如示例 7-1 中所示那样增加这个模块：

文件名: `src/lib.rs`

```
mod network {  
    fn connect() {  
    }  
}  
  
mod client {  
    fn connect() {  
    }  
}
```

示例 7-1: `network` 模块和 `client` 一同定义于 `src/lib.rs`

现在我们有 `network::connect` 函数和 `client::connect` 函数。它们可能有着完全不同的功能，同时它们也不会彼此冲突，因为它们位于不同的模块。

在这个例子中，因为我们构建的是一个库，作为库入口点的文件是 `src/lib.rs`。然而，对于创建模块来说，`src/lib.rs` 并没有什么特殊意义。也可以在二进制 `crate` 的 `src/main.rs` 中创建模块，正如在库 `crate` 的 `src/lib.rs` 创建模块一样。事实上，也可以将模块放入其他模块中。这有助于随着模块的增长，将相关的功能组织在一起并又保持各自独立。如何选择组织代码依赖于如何考虑代码不同部分之间的关系。例如，对于库的用户来说，`client` 模块和它的函数 `connect` 可能放在 `network` 命名空间里显得更有道理，如示例 7-2 所示：

文件名: `src/lib.rs`

```
mod network {  
    fn connect() {  
    }  
  
    mod client {  
        fn connect() {  
        }  
    }  
}
```

示例 7-2: 将 `client` 模块移动到 `network` 模块中

在 `src/lib.rs` 文件中，将现有的 `mod network` 和 `mod client` 的定义替换为示例 7-2 中的定义，这里将 `client` 模块作为 `network` 的一个内部模块。现在我们有 `network::connect` 和 `network::client::connect` 函数：同样的，这两个 `connect` 函数也不相冲突，因为它们在不同的命名空间中。

这样，模块之间形成了一个层次结构。`src/lib.rs` 的内容位于最顶层，而其子模块位于较低层次。如下是示例 7-1 中的例子以层次的方式考虑的结构：

```
communicator  
├─ network  
└─ client
```

而这是示例 7-2 中例子的层次结构：

```
communicator  
└─ network  
    └─ client
```


可以看到示例 7-2 中，`client` 是 `network` 的子模块，而不是它的同级模块。更为复杂的项目可以有很多的模块，所以它们需要符合逻辑地组合在一起以便记录它们。在项目中“符合逻辑”的意义全凭你的理解和库的用户对你项目领域的认识。利用我们这里讲到的技术来创建同级模块和嵌套的模块，总有一个会是你喜欢的结构。

将模块移动到其他文件

位于层级结构中的模块，非常类似计算机领域的另一个我们非常熟悉的结构：文件系统！我们可以利用 **Rust** 的模块系统连同多个文件一起分解 **Rust** 项目，这样就不会是所有的内容都落到 `src/lib.rs` 或 `src/main.rs` 中了。为了举例，我们将从示例 7-3 中的代码开始：

文件名: `src/lib.rs`

```
mod client {
    fn connect() {
    }
}

mod network {
    fn connect() {
    }

    mod server {
        fn connect() {
        }
    }
}
```

示例 7-3: 三个模块，`client`、`network` 和 `network::server`，它们都定义于 `src/lib.rs`

`src/lib.rs` 文件有如下层次结构：

```
communicator
├─ client
└─ network
   └─ server
```

如果这些模块有很多函数，而这些函数又很长，将难以在文件中寻找我们需要的代码。因为这些函数被嵌套进一个或多个模块中，同时函数中的代码也会开始变长。这就有充分的理由将 `client`、`network` 和 `server` 每一个模块从 `src/lib.rs` 抽出并放入它们自己的文件中。

首先，将 `client` 模块的代码替换为只有 `client` 模块声明，这样 `src/lib.rs` 看起来应该像这样：

文件名: `src/lib.rs`

```
mod client;

mod network {
    fn connect() {
    }

    mod server {
        fn connect() {
        }
    }
}
```

这里我们仍然 声明 了 `client` 模块，不过将代码块替换为了分号，这告诉了 `Rust` 在 `client` 模块的作用域中寻找另一个定义代码的位置。换句话说，`mod client;` 行意味着：

```
mod client {  
    // contents of client.rs  
}
```

那么现在需要创建对应模块名的外部文件。在 `src/` 目录创建一个 `client.rs` 文件，接着打开它并输入如下内容，它是上一步被去掉的 `client` 模块中的 `connect` 函数：

Filename: `src/client.rs`

```
fn connect() {  
}
```

注意这个文件中并不需要一个 `mod` 声明；因为已经在 `src/lib.rs` 中已经使用 `mod` 声明了 `client` 模块。这个文件仅提供 `client` 模块的内容。如果在这里加上一个 `mod client`，那么就等于给 `client` 模块增加了一个叫做 `client` 的子模块了！

`Rust` 默认只知道 `src/lib.rs` 中的内容。如果要对项目加入更多文件，我们需要在 `src/lib.rs` 中告诉 `Rust` 去寻找其他文件；这就是为什么 `mod client` 需要被定义在 `src/lib.rs` 而不能在 `src/client.rs` 的原因。

现在，一切应该能成功编译，虽然会有一些警告。记住使用 `cargo build` 而不是 `cargo run` 因为这是一个库 `crate` 而不是二进制 `crate`：

```
$ cargo build  
Compiling communicator v0.1.0 (file:///projects/communicator)  
  
warning: function is never used: `connect`, #[warn(dead_code)] on by default  
--> src/client.rs:1:1  
|  
1 | fn connect() {  
|   ^  
  
warning: function is never used: `connect`, #[warn(dead_code)] on by default  
--> src/lib.rs:4:5  
|  
4 |     fn connect() {  
|       ^  
  
warning: function is never used: `connect`, #[warn(dead_code)] on by default  
--> src/lib.rs:8:9  
|  
8 |         fn connect() {  
|           ^
```

这些警告提醒我们有从未被使用的函数。目前不用担心这些警告，在本章后面的“使用 `pub` 控制可见性”部分会解决它们。好消息是，它们仅仅是警告，我们的项目能够被成功编译。

下面使用相同的模式将 `network` 模块提取到自己的文件中。删除 `src/lib.rs` 中 `network` 模块的内容并在声明后加上一个分号，像这样：

Filename: `src/lib.rs`

```
mod client;  
  
mod network;
```

接着新建 `src/network.rs` 文件并输入如下内容：

Filename: `src/network.rs`

```
fn connect() {  
}  
  
mod server {  
    fn connect() {  
    }  
}
```

注意这个模块文件中我们也使用了一个 `mod` 声明；这是因为我们希望 `server` 成为 `network` 的一个子模块。

现在再次运行 `cargo build`。成功！不过我们还需要再提取出另一个模块：`server`。因为这是一个子模块——也就是模块中的模块——目前的将模块提取到对应名字的文件中的策略就不管用了。如果我们仍这么尝试则会出现错误。对 `src/network.rs` 的第一个修改是用 `mod server;` 替换 `server` 模块的内容：

Filename: `src/network.rs`

```
fn connect() {  
}  
  
mod server;
```

接着创建 `src/server.rs` 文件并输入需要提取的 `server` 模块的内容：

Filename: `src/server.rs`

```
fn connect() {  
}
```

当尝试运行 `cargo build` 时，会出现如示例 7-4 中所示的错误：

```
$ cargo build  
Compiling communicator v0.1.0 (file:///projects/communicator)  
error: cannot declare a new module at this location  
--> src/network.rs:4:5  
|  
4 | mod server;  
|   ^^^^^^  
|  
note: maybe move this module `network` to its own directory via `network/mod.rs`  
--> src/network.rs:4:5  
|  
4 | mod server;  
|   ^^^^^^  
note: ... or maybe `use` the module `server` instead of possibly redeclaring it  
--> src/network.rs:4:5  
|  
4 | mod server;  
|   ^^^^^^
```

示例 7-4：尝试将 `server` 子模块提取到 `src/server.rs` 时出现的错误

这个错误说明“不能在这个位置新声明一个模块”并指出 `src/network.rs` 中的 `mod server;` 这一行。看来 `src/network.rs` 与 `src/lib.rs` 在某些方面是不同的；继续阅读以理解这是为什么。

示例 7-4 中间的 `note` 事实上是非常有帮助的，因为它指出了一些我们还未讲到的操作：

```
note: maybe move this module `network` to its own directory via
`network/mod.rs`
```

我们可以按照记录所建议的去操作，而不是继续使用之前的与模块同名文件的模式：

1. 新建一个叫做 `network` 的目录，这是父模块的名字
2. 将 `src/network.rs` 移动到新建的 `network` 目录中并重命名，现在它是 `src/network/mod.rs`
3. 将子模块文件 `src/server.rs` 移动到 `network` 目录中

如下是执行这些步骤的命令：

```
$ mkdir src/network
$ mv src/network.rs src/network/mod.rs
$ mv src/server.rs src/network
```

现在如果运行 `cargo build` 的话将顺利编译（虽然仍有警告）。现在模块的布局看起来仍然与示例 7-3 中所有代码都在 `src/lib.rs` 中时完全一样：

```
communicator
├── client
└── network
    └── server
```

对应的文件布局现在看起来像这样：

```
├── src
│   ├── client.rs
│   ├── lib.rs
│   └── network
│       ├── mod.rs
│       └── server.rs
```

那么，当我们想要提取 `network::server` 模块时，为什么也必须将 `src/network.rs` 文件改名为 `src/network/mod.rs` 文件呢，还有为什么要将 `network::server` 的代码放入 `network` 目录的 `src/network/server.rs` 文件中，而不能将 `network::server` 模块提取到 `src/server.rs` 中呢？原因是如果 `server.rs` 文件在 `src` 目录中那么 Rust 就不能知道 `server` 应当是 `network` 的子模块。为了阐明这里 Rust 的行为，让我们考虑一下有着如下层级的另一个例子，它的所有定义都位于 `src/lib.rs` 中：

```
communicator
├── client
└── network
    └── client
```

在这个例子中，仍然有这三个模块，`client`、`network` 和 `network::client`。如果按照与上面最开始将模块提取到文件中相同的步骤来操作，对于 `client` 模块会创建 `src/client.rs`。对于 `network` 模块，会创建 `src/network.rs`。但是接下来不能将 `network::client` 模块提取到 `src/client.rs` 文件中，因为它已经存在了，对应顶层的 `client` 模块！如果将 `client` 和 `network::client` 的代码都放入 `src/client.rs` 文件，Rust 将无从可知这些代码是属于 `client` 还是 `network::client` 的。

因此，一旦想要将 `network` 模块的子模块 `network::client` 提取到一个文件中，需要为 `network` 模块新建一个目录替代 `src/network.rs` 文件。接着 `network` 模块的代码将进入 `src/network/mod.rs` 文件，而子模块 `network::client` 将拥有其自己的文件 `src/network/client.rs`。现在顶层的 `src/client.rs` 中的代码毫无疑问的都属于 `client` 模块。

模块文件系统的规则

与文件系统相关的模块规则总结如下：

- 如果一个叫做 `foo` 的模块没有子模块，应该将 `foo` 的声明放入叫做 `foo.rs` 的文件中。
- 如果一个叫做 `foo` 的模块有子模块，应该将 `foo` 的声明放入叫做 `foo/mod.rs` 的文件中。

这些规则适用于递归（嵌套），所以如果 `foo` 模块有一个子模块 `bar` 而 `bar` 没有子模块，则 `src` 目录中应该有如下文件：

```
├─ foo
│   ├─ bar.rs (contains the declarations in `foo::bar`)
│   └─ mod.rs (contains the declarations in `foo`, including `mod bar`)
```

模块自身则应该使用 `mod` 关键字定义于父模块的文件中。

接下来，我们讨论一下 `pub` 关键字，并除掉那些警告！

使用 `pub` 控制可见性

[ch07-02-controlling-visibility-with-pub.md](#)

commit 0a4ed5875aeba78a81ae03ac73aeb84d2e2aca86

我们通过将 `network` 和 `network::server` 的代码分别移动到 `src/network/mod.rs` 和 `src/network/server.rs` 文件中解决了示例 7-4 中出现的错误信息。现在，`cargo build` 能够构建我们的项目，不过仍然有一些警告信息，表示 `client::connect`、`network::connect` 和 `network::server::connect` 函数没有被使用：

```
warning: function is never used: `connect`, #[warn(dead_code)] on by default
src/client.rs:1:1
|
1 | fn connect() {
|   ^

warning: function is never used: `connect`, #[warn(dead_code)] on by default
--> src/network/mod.rs:1:1
|
1 | fn connect() {
|   ^

warning: function is never used: `connect`, #[warn(dead_code)] on by default
--> src/network/server.rs:1:1
|
1 | fn connect() {
|   ^
```

那么为什么会出现这些错误信息呢？我们构建的是一个库，它的函数的目的是被用户使用，而不一定要被项目自身使用，所以不应该担心这些 `connect` 函数是未使用的。创建它们的意义就在于被另一个项目而不是被我们自己使用。

为了理解为什么这个程序出现了这些警告，尝试作为另一个项目来使用这个 `connect` 库，从外部调用它们。为此，通过创建一个包含这些代码的 `src/main.rs` 文件，在与库 `crate` 相同的目录创建一个二进制 `crate`：

文件名: `src/main.rs`

```
extern crate communicator;

fn main() {
    communicator::client::connect();
}
```

使用 `extern crate` 指令将 `communicator` 库 `crate` 引入到作用域，因为事实上我们的包现在包含两个 `crate`。Cargo 认为 `src/main.rs` 是一个二进制 `crate` 的根文件，与现存的以 `src/lib.rs` 为根文件的库 `crate` 相区分。这个模式在执行项目中非常常见：大部分功能位于库 `crate` 中，而二进制 `crate` 使用这个库 `crate`。通过这种方式，其他程序也可以使用这个库 `crate`，这是一个很好的关注分离（*separation of concerns*）。

从一个外部 `crate` 的视角观察 `communicator` 库的内部，我们创建的所有模块都位于一个与 `crate` 同名的模块内部，`communicator`。这个顶层的模块被称为 `crate` 的根模块（*root module*）。

另外注意到即便在项目的子模块中使用外部 `crate`，`extern crate` 也应该位于根模块（也就是 `src/main.rs` 或 `src/lib.rs`）。接着，在子模块中，我们就可以像顶层模块那样引用外部 `crate` 中的项了。

我们的二进制 `crate` 如今正好调用了库中 `client` 模块的 `connect` 函数。然而，执行 `cargo build` 会在之前的警告之后出现一个错误：

```
error: module `client` is private
--> src/main.rs:4:5
|
4 |     communicator::client::connect();
|     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

啊哈！这告诉了我们 `client` 模块是私有的，这也正是那些警告的症结所在。这也是我们第一次在 **Rust** 上下文中涉及到公有（*public*）和私有（*private*）的概念。**Rust** 所有代码的默认状态是私有的：除了自己之外别人不允许使用这些代码。如果不在自己的项目中使用一个私有函数，因为程序自身是唯一允许使用这个函数的代码，**Rust** 会警告说函数未被使用。

一旦我们指定一个像 `client::connect` 的函数为公有，不光二进制 **crate** 中的函数调用是允许的，函数未被使用的警告也会消失。将其标记为公有让 **Rust** 知道了我们意在使函数在程序的外部被使用。现在这个可能的理论上的外部可用性使得 **Rust** 认为这个函数“已经被使用”。因此。当某项被标记为公有，**Rust** 不再要求它在程序自身被使用并停止警告某项未被使用。

标记函数为公有

为了告诉 **Rust** 某项为公有，在想要标记为公有的项的声明开头加上 `pub` 关键字。现在我们将致力于修复 `client::connect` 未被使用的警告，以及二进制 **crate** 中“模块 `client` 是私有的”的错误。像这样修改 `src/lib.rs` 使 `client` 模块公有：

文件名: `src/lib.rs`

```
pub mod client;

mod network;
```

`pub` 写在 `mod` 之前。再次尝试构建：

```
error: function `connect` is private
--> src/main.rs:4:5
|
4 |     communicator::client::connect();
|     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

非常好！另一个不同的错误！好的，不同的错误信息也是值得庆祝的（可能是程序员被黑的最惨的一次）。新错误表明“函数 `connect` 是私有的”，那么让我们修改 `src/client.rs` 将 `client::connect` 也设为公有：

文件名: `src/client.rs`

```
pub fn connect() {
}
```

再一次运行 `cargo build`：

```
warning: function is never used: `connect`, #[warn(dead_code)] on by default
--> src/network/mod.rs:1:1
|
1 | fn connect() {
|   ^

warning: function is never used: `connect`, #[warn(dead_code)] on by default
--> src/network/server.rs:1:1
|
```

```
1 | fn connect() {  
  | ^
```

编译通过了，关于 `client::connect` 未被使用的警告消失了！

未被使用的代码并不总是意味着它们需要被设为公有的：如果你不希望这些函数成为公有 API 的一部分，未被使用的代码警告可能是在警告你这些代码不再需要并可以安全的删除它们。这也可能是警告你出 bug 了，如果你刚刚不小心删除了库中所有这个函数的调用。

当然我们的情况是，确实希望另外两个函数也作为 crate 公有 API 的一部分，所以让我们也将其标记为 `pub` 并去掉剩余的警告。修改 `src/network/mod.rs` 为：

文件名: `src/network/mod.rs`

```
pub fn connect() {  
}  
  
mod server;
```

并编译代码：

```
warning: function is never used: `connect`, #[warn(dead_code)] on by default  
--> src/network/mod.rs:1:1  
|  
1 | pub fn connect() {  
  | ^  
  
warning: function is never used: `connect`, #[warn(dead_code)] on by default  
--> src/network/server.rs:1:1  
|  
1 | fn connect() {  
  | ^
```

虽然将 `network::connect` 设为 `pub` 了我们仍然得到了一个未被使用函数的警告。这是因为模块中的函数是公有的，不过函数所在的 `network` 模块却不是公有的。这回我们是自内向外修改库文件的，而 `client::connect` 的时候是自外向内修改的。我们需要修改 `src/lib.rs` 让 `network` 也是公有的：

文件名: `src/lib.rs`

```
pub mod client;  
  
pub mod network;
```

现在编译的话，那个警告就消失了：

```
warning: function is never used: `connect`, #[warn(dead_code)] on by default  
--> src/network/server.rs:1:1  
|  
1 | fn connect() {  
  | ^
```

只剩一个警告了！尝试自食其力修改它吧！

私有性规则

总的来说，有如下项的可见性规则：

1. 如果一个项是公有的，它能被任何父模块访问
2. 如果一个项是私有的，它能被其直接父模块及其任何子模块访问

私有性示例

让我们看看更多例子作为练习。创建一个新的库项目并在新项目的 `src/lib.rs` 输入示例 7-5 中的代码：

文件名: `src/lib.rs`

```
mod outermost {
    pub fn middle_function() {}

    fn middle_secret_function() {}

    mod inside {
        pub fn inner_function() {}

        fn secret_function() {}
    }
}

fn try_me() {
    outermost::middle_function();
    outermost::middle_secret_function();
    outermost::inside::inner_function();
    outermost::inside::secret_function();
}
```

示例 7-5：私有和公有函数的例子，其中部分是不正确的

在尝试编译这些代码之前，猜测一下 `try_me` 函数的哪一行会出错。接着编译项目来看看是否猜对了，然后继续阅读后面关于错误的讨论！

检查错误

`try_me` 函数位于项目的根模块。叫做 `outermost` 的模块是私有的，不过第二条私有性规则说明 `try_me` 函数允许访问 `outermost` 模块，因为 `outermost` 位于当前（根）模块，`try_me` 也是。

`outermost::middle_function` 的调用是正确的。因为 `middle_function` 是公有的，而 `try_me` 通过其父模块 `outermost` 访问 `middle_function`。根据上一段的规则我们可以确定这个模块是可访问的。

`outermost::middle_secret_function` 的调用会造成一个编译错误。`middle_secret_function` 是私有的，所以第二条（私有性）规则生效了。根模块既不是 `middle_secret_function` 的当前模块（`outermost` 是），也不是 `middle_secret_function` 当前模块的子模块。

叫做 `inside` 的模块是私有的且没有子模块，所以它只能被当前模块 `outermost` 访问。这意味着 `try_me` 函数不允许调用 `outermost::inside::inner_function` 或 `outermost::inside::secret_function` 中的任何一个。

修改错误

这里有一些尝试修复错误的代码修改意见。在你尝试它们之前，猜测一下它们哪个能修复错误，接着编译查看你是否猜对了，并结合私有性规则理解为什么。

- 如果 `inside` 模块是公有的？
- 如果 `outermost` 是公有的而 `inside` 是私有的？
- 如果在 `inner_function` 函数体中调用 `::outermost::middle_secret_function()`？（开头的两个冒号意味着从根模

块开始引用模块。)

请随意设计更多的实验并尝试理解它们！

接下来，让我们讨论一下使用 `use` 关键字将模块项目引入作用域。

导入命名

ch07-03-importing-names-with-use.md

commit d06a6a181fd61704cbf7feb55bc61d518c6469f9

我们已经讲到了如何使用模块名称作为调用的一部分，来调用模块中的函数，如示例 7-6 中所示的 `nested_modules` 函数调用。

文件名: `src/main.rs`

```
pub mod a {
    pub mod series {
        pub mod of {
            pub fn nested_modules() {}
        }
    }
}

fn main() {
    a::series::of::nested_modules();
}
```

示例 7-6: 通过完全指定模块中的路径来调用函数

如你所见，指定函数的完全限定名称可能会非常冗长。所幸 **Rust** 有一个关键字使得这些调用显得更简洁。

使用 `use` 的简单导入

Rust 的 `use` 关键字的工作是缩短冗长的函数调用，通过将想要调用的函数所在的模块引入到作用域中。这是一个将 `a::series::of` 模块导入一个二进制 `crate` 的根作用域的例子：

文件名: `src/main.rs`

```
pub mod a {
    pub mod series {
        pub mod of {
            pub fn nested_modules() {}
        }
    }
}

use a::series::of;

fn main() {
    of::nested_modules();
}
```

`use a::series::of;` 这一行的意思是每当想要引用 `of` 模块时，不必使用完整的 `a::series::of` 路径，可以直接使用 `of`。

`use` 关键字只将指定的模块引入作用域；它并不会将其子模块也引入。这就是为什么想要调用 `nested_modules` 函数时仍然必须写成 `of::nested_modules`。

也可以将函数本身引入到作用域中，通过如下在 `use` 中指定函数的方式：

```
pub mod a {
```

```
pub mod series {
    pub mod of {
        pub fn nested_modules() {}
    }
}

use a::series::of::nested_modules;

fn main() {
    nested_modules();
}
```

这使得我们可以忽略所有的模块并直接引用函数。

因为枚举也像模块一样组成了某种命名空间，也可以使用 `use` 来导入枚举的成员。对于任何类型的 `use` 语句，如果从一个命名空间导入多个项，可以使用大括号和逗号来列举它们，像这样：

```
enum TrafficLight {
    Red,
    Yellow,
    Green,
}

use TrafficLight::{Red, Yellow};

fn main() {
    let red = Red;
    let yellow = Yellow;
    let green = TrafficLight::Green;
}
```

使用 `*` 的全局引用导入

为了一次导入某个命名空间的所有项，可以使用 `*` 语法。例如：

```
enum TrafficLight {
    Red,
    Yellow,
    Green,
}

use TrafficLight::*;

fn main() {
    let red = Red;
    let yellow = Yellow;
    let green = Green;
}
```

`*` 被称为全局导入（*glob*），它会导入命名空间中所有可见的项。全局导入应该保守的使用：它们是方便的，但是也可能会引入多于你预期的内容从而导致命名冲突。

使用 `super` 访问父模块

正如我们已经知道的，当创建一个库 `crate` 时，Cargo 会生成一个 `tests` 模块。现在让我们来深入了解一下。在 `communicator` 项目中，打开 `src/lib.rs`。

文件名: `src/lib.rs`

```
pub mod client;

pub mod network;

#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

第十一章会更详细的解释测试，不过其部分内容现在应该可以理解了：有一个叫做 `tests` 的模块紧邻其他模块，同时包含一个叫做 `it_works` 的函数。即便存在一些特殊注解，`tests` 也不过是另外一个模块！所以我们的模块层次结构看起来像这样：

```
communicator
├── client
├── network
│   └── client
└── tests
```

测试是为了检验库中的代码而存在的，所以让我们尝试在 `it_works` 函数中调用 `client::connect` 函数，即便现在不准测试任何功能：

文件名: `src/lib.rs`

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        client::connect();
    }
}
```

使用 `cargo test` 命令运行测试：

```
$ cargo test
   Compiling communicator v0.1.0 (file:///projects/communicator)
error[E0433]: failed to resolve. Use of undeclared type or module `client`
--> src/lib.rs:9:9
  |
 9 |         client::connect();
  |         ^^^^^^^^^^^^^^^^^ Use of undeclared type or module `client`
```

编译失败了，不过为什么呢？并不需要像 `src/main.rs` 那样将 `communicator::` 置于函数前，因为这里肯定是在 `communicator` 库 `crate` 之内的。失败的原因是路径是相对于当前模块的，在这里就是 `tests`。唯一的例外就是 `use` 语句，它默认是相对于 `crate` 根模块的。我们的 `tests` 模块需要 `client` 模块位于其作用域中！

那么如何在模块层次结构中回退一级模块，以便在 `tests` 模块中能够调用 `client::connect` 函数呢？在 `tests` 模块中，要么可以在开头使用双冒号来让 `Rust` 知道我们想要从根模块开始并列出整个路径：

```
::client::connect();
```

要么可以使用 `super` 在层级中获取当前模块的上一级模块：

```
super::client::connect();
```

在这个例子中这两个选择看不出有多么大的区别，不过随着模块层次的更加深入，每次都从根模块开始就会显得很长了。在这些情况下，使用 `super` 来获取当前模块的同级模块是一个好的捷径。再加上，如果在代码中的很多地方指定了从根开始的路径，那么当通过移动子树或到其他位置来重新排列模块时，最终就需要更新很多地方的路径，这就非常乏味无趣了。

在每一个测试中总是不得不编写 `super::` 也会显得很恼人，不过你已经见过解决这个问题的利器了：`use ! super::` 的功能改变了提供给 `use` 的路径，使其不再相对于根模块而是相对于父模块。

为此，特别是在 `tests` 模块，`use super::something` 是常用的手段。所以现在的测试看起来像这样：

文件名: `src/lib.rs`

```
#[cfg(test)]
mod tests {
    use super::client;

    #[test]
    fn it_works() {
        client::connect();
    }
}
```

如果再次运行 `cargo test`，测试将会通过而且测试结果输出的第一部分将会是：

```
$ cargo test
   Compiling communicator v0.1.0 (file:///projects/communicator)
   Running target/debug/communicator-92007ddb5330fa5a

running 1 test
test tests::it_works ... ok

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

总结

现在你掌握了组织代码的核心科技！利用它们将相关的代码组合在一起、防止代码文件过长并将一个整洁的公有 API 展现给库的用户。

接下来，让我们看看一些标准库提供的集合数据类型，你可以利用它们编写出漂亮整洁的代码。

通用集合类型

ch08-00-common-collections.md

commit d06a6a181fd61704cbf7feb55bc61d518c6469f9

Rust 标准库中包含一系列被称为 集合 (*collections*) 的非常有用的数据结构。大部分其他数据类型都代表一个特定的值，不过集合可以包含多个值。不同于内建的数组和元组类型，这些集合指向的数据是储存在堆上的，这意味着数据的数量不必在编译时就可知并且可以随着程序的运行增长或缩小。每种集合都有着不同能力和代价，而为所处的场景选择合适的集合则是你将要始终发展的技能。在这一章里，我们将详细的了解三个在 **Rust** 程序中被广泛使用的集合：

- **vector** 允许我们一个挨着一个地储存一系列数量可变的值
- 字符串 (*string*) 是一个字符的集合。我们之前见过 `String` 类型，现在将详细介绍它。
- 哈希 **map** (*hash map*) 允许我们将值与一个特定的键 (**key**) 相关联。这是一个叫做 *map* 的更通用的数据结构的特定实现。

对于标准库提供的其他类型的集合，请查看[文档](#)。

我们将讨论如何创建和更新 **vector**、字符串和哈希 **map**，以及它们有什么不同。

vector

ch08-01-vectors.md

commit 6c24544ba718bce0755bdaf03423af86280051d5

我们要讲到的第一个类型是 `Vec<T>`，也被称为 **vector**。**vector** 允许我们在一个单独的数据结构中储存多于一个值，它在内存中彼此相邻地排列所有的值。**vector** 只能储存相同类型的值。它们在拥有一系列项的场景下非常实用，例如文件中的文本行或是购物车中商品的价格。

新建 vector

为了创建一个新的空 **vector**，可以调用 `Vec::new` 函数：

```
let v: Vec<i32> = Vec::new();
```

注意这里我们增加了一个类型注解。因为没有向这个 **vector** 中插入任何值，**Rust** 并不知道我们想要储存什么类型的元素。这是一个非常重要的点。**vector** 是同质的（homogeneous）：它们可以储存很多值，不过这些值必须都是相同类型的。**vector** 是用泛型实现的，第十章会涉及到如何对你自己的类型使用它们。现在，所有你需要知道的就是 `Vec` 是一个由标准库提供的类型，它可以存放任何类型，而当 `Vec` 存放某个特定类型时，那个类型位于尖括号中。这里我们告诉 **Rust** `v` 这个 `Vec` 将存放 `i32` 类型的元素。

在实际的代码中，一旦插入值 **Rust** 就可以推断出想要存放的类型，所以你很少会需要这些类型注解。更常见的做法是使用初始值来创建一个 `Vec`，而且为了方便 **Rust** 提供了 `vec!` 宏。这个宏会根据我们提供的值来创建一个新的 `Vec`。如下代码会新建一个拥有值 `1`、`2` 和 `3` 的 `Vec<i32>`：

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

因为我们提供了 `i32` 类型的初始值，**Rust** 可以推断出 `v` 的类型是 `Vec<i32>`，因此类型注解就不是必须的。接下来让我们看看如何修改一个 **vector**。

更新 vector

对于新建一个 **vector** 并向其增加元素，可以使用 `push` 方法：

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

v.push(5);
v.push(6);
v.push(7);
v.push(8);
```

如第三章中讨论的任何变量一样，如果想要能够改变它的值，必须使用 `mut` 关键字使其可变。放入其中的所有值都是 `i32` 类型的，而且 **Rust** 也根据数据如此判断，所以不需要 `Vec<i32>` 注解。

丢弃 vector 时也会丢弃其所有元素

类似于任何其他 `struct`，**vector** 在其离开作用域时会被释放：

```
{
    let v = vec![1, 2, 3, 4];
}
```



```
// do stuff with v

} // <- v goes out of scope and is freed here
```

当 **vector** 被丢弃时，所有其内容也会被丢弃，这意味着这里它包含的整数将被清理。这可能看起来非常直观，不过一旦开始使用 **vector** 元素的引用，情况就变得有些复杂了。下面让我们处理这种情况！

读取 **vector** 的元素

现在你知道如何创建、更新和销毁 **vector** 了，接下来的一步最好了解一下如何读取它们的内容。有两种方法引用 **vector** 中储存的值。为了更加清楚的说明这个例子，我们标注这些函数返回的值的类型。

这个例子展示了访问 **vector** 中一个值的两种方式，索引语法或者 `get` 方法：

```
let v = vec![1, 2, 3, 4, 5];

let third: &i32 = &v[2];
let third: Option<&i32> = v.get(2);
```

这里有一些需要注意的地方。首先，我们使用索引值 `2` 来获取第三个元素，索引是从 `0` 开始的。其次，这两个不同的获取第三个元素的方式分别为：使用 `&` 和 `[]` 返回一个引用；或者使用 `get` 方法以索引作为参数来返回一个 `Option<&T>`。

Rust 有两个引用元素的方法的原因是程序可以选择如何处理当索引值在 **vector** 中没有对应值的情况。例如如下情况，如果有一个有五个元素的 **vector** 接着尝试访问索引为 `100` 的元素，程序该如何处理：

```
let v = vec![1, 2, 3, 4, 5];

let does_not_exist = &v[100];
let does_not_exist = v.get(100);
```

当运行这段代码，你会发现对于第一个 `[]` 方法，当引用一个不存在的元素时 **Rust** 会造成 `panic!`。这个方法更适合当程序认为尝试访问超过 **vector** 结尾的元素是一个严重错误的情况，这时应该使程序崩溃。

当 `get` 方法被传递了一个数组外的索引时，它不会 `panic` 而是返回 `None`。当偶尔出现超过 **vector** 范围的访问属于正常情况的时候可以考虑使用它。接着你的代码可以有处理 `Some(&element)` 或 `None` 的逻辑，如第六章讨论的那样。例如，索引可能来源于用户输入的数字。如果它们不慎输入了一个过大的数字那么程序就会得到 `None` 值，你可以告诉用户 `Vec` 当前元素的数量并再请求它们输入一个有效的值。这就比因为输入错误而使程序崩溃要友好的多！

无效引用

一旦程序获取了一个有效的引用，借用检查器将会执行第四章讲到的所有权和借用规则来确保 **vector** 内容的这个引用和任何其他引用保持有效。回忆一下不能在相同作用域中同时存在可变和不可变引用的规则。这个规则适用于这个例子，当我们获取了 **vector** 的第一个元素的不可变引用并尝试在 **vector** 末尾增加一个元素的时候：

```
let mut v = vec![1, 2, 3, 4, 5];

let first = &v[0];

v.push(6);
```

编译会给出这个错误：

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as
immutable
  |
4 | let first = &v[0];
  |             - immutable borrow occurs here
5 |
6 | v.push(6);
  |   ^ mutable borrow occurs here
7 | }
  |   - immutable borrow ends here
```

这些代码看起来应该能够运行：为什么第一个元素的引用会关心 **vector** 结尾的变化？不能这么做的原因是由于 **vector** 的工作方式。在 **vector** 的结尾增加新元素时，在没有足够空间将所有元素依次相邻存放的情况下，可能会要求分配新内存并将老的元素拷贝到新的空间中。这时，第一个元素的引用就指向了被释放的内存。借用规则阻止程序陷入这种状况。

注意：关于更多内容，查看 Nomicon <https://doc.rust-lang.org/stable/nomicon/vec.html>

使用枚举来储存多种类型

在本章的开始，我们提到 **vector** 只能储存相同类型的值。这是很不方便的；绝对会有需要储存一系列不同类型的值的用例。幸运的是，枚举的成员都被定义为相同的枚举类型，所以当需要在 **vector** 中储存不同类型值时，我们可以定义并使用一个枚举！

例如，假如我们想要从电子表格的一行中获取值，而这一行的有些列包含数字，有些包含浮点值，还有些是字符串。我们可以定义一个枚举，其成员会存放这些不同类型的值，同时所有这些枚举成员都会被当作相同类型，那个枚举的类型。接着可以创建一个储存枚举值的 **vector**，这样最终就能够储存不同类型的值了：

```
enum SpreadsheetCell {
    Int(i32),
    Float(f64),
    Text(String),
}

let row = vec![
    SpreadsheetCell::Int(3),
    SpreadsheetCell::Text(String::from("blue")),
    SpreadsheetCell::Float(10.12),
];
```

示例 8-1：定义一个枚举，以便能在 **vector** 中存放不同类型的数据

Rust 在编译时就必须准确的知道 **vector** 中类型的原因在于它需要知道储存每个元素到底需要多少内存。第二个好处是可以准确的知道这个 **vector** 中允许什么类型。如果 **Rust** 允许 **vector** 存放任意类型，那么当对 **vector** 元素执行操作时一个或多个类型的值就有可能会造成错误。使用枚举外加 `match` 意味着 **Rust** 能在编译时就保证总是会处理所有可能的情况，正如第六章讲到的那样。

如果在编写程序时不能确切无遗地知道运行时会储存进 **vector** 的所有类型，枚举技术就行不通了。相反，你可以使用 **trait** 对象，第十七章会讲到它。

现在我们了解了一些使用 **vector** 的最常见的方式，请一定去看看标准库中 `Vec` 定义的很多其他实用方法的 **API** 文档。例如，除了 `push` 之外还有一个 `pop` 方法，它会移除并返回 **vector** 的最后一个元素。让我们继续下一个集合类型：`String`！

字符串

ch08-02-strings.md

commit 692c4a78aac93670bc6f1fa5d33f71ed161b9339

第四章已经讲过一些字符串的内容，不过现在让我们更深入地了解它。字符串是新晋 **Rustacean** 们通常会被困住的领域。这是由于三方面内容的结合：**Rust** 倾向于确保暴露出可能的错误，字符串是比很多程序员所想象的要更为复杂的数据结构，以及 **UTF-8**。所有这些结合起来对于来自其他语言背景的程序员就可能显得很困难了。

字符串出现在集合章节的原因是，字符串是作为字节的集合外加一些方法实现的，当这些字节被解释为文本时，这些方法提供了实用的功能。在这一部分，我们会讲到 `String` 中那些任何集合类型都有的操作，比如创建、更新和读取。也会讨论 `String` 与其他集合不一样的地方，例如索引 `String` 是很复杂的，由于人和计算机理解 `String` 数据方式的不同。

什么是字符串？

在开始深入这些方面之前，我们需要讨论一下术语 字符串 的具体意义。**Rust** 的核心语言中事实上就只有一种字符串类型：`str`，字符串 `slice`，它通常以被借用的形式出现，`&str`。第四章讲到了 字符串 `slice`：它们是一些储存在别处的 **UTF-8** 编码字符串数据的引用。比如字符串字面值被储存在程序的二进制输出中，字符串 `slice` 也是如此。

称作 `String` 的类型是由标准库提供的，而没有写进核心语言部分，它是可增长的、可变的、有所有权的、**UTF-8** 编码的字符串类型。当 **Rustacean** 们谈到 **Rust** 的“字符串”时，它们通常指的是 `String` 和字符串 `slice` `&str` 类型，而不是其中一个。这一部分大部分是关于 `String` 的，不过这些类型在 **Rust** 标准库中都被广泛使用。`String` 和字符串 `slice` 都是 **UTF-8** 编码的。

Rust 标准库中还包含一系列其他字符串类型，比如 `OsString`、`OsStr`、`CString` 和 `CStr`。相关库 `crate` 甚至会提供更多储存字符串数据的选择。与 `*String` / `*Str` 的命名类似，它们通常也提供有所有权和可借用的变体，比如说 `String` / `&str`。这些字符串类型在储存的编码或内存表现形式上可能有所不同。本章将不会讨论其他这些字符串类型；查看 **API** 文档来更多的了解如何使用它们以及各自适合的场景。

新建字符串

很多 `Vec` 可用的操作在 `String` 中同样可用，从以 `new` 函数创建字符串开始，像这样：

```
let mut s = String::new();
```

这新建了一个叫做 `s` 的空字符串，接着我们可以向其中装载数据。

通常字符串会有初始数据，因为我们希望一开始就有这个字符串。为此，使用 `to_string` 方法，它能用于任何实现了 `Display` `trait` 的类型，对于字符串字面值是这样：

```
let data = "initial contents";

let s = data.to_string();

// the method also works on a literal directly:
let s = "initial contents".to_string();
```

这会创建一个包含 `initial contents` 的字符串。

也可以使用 `String::from` 函数来从字符串字面值创建 `String`。下面代码等同于使用 `to_string`：

```
let s = String::from("initial contents");
```

因为字符串使用广泛，这里有很多不同的用于字符串的通用 **API** 可供选择。它们有些可能显得有些多余，不过都有其用武之地！在这个例子中，`String::from` 和 `.to_string` 最终做了完全相同的工作，所以如何选择就是风格问题了。

记住字符串是 **UTF-8** 编码的，所以可以包含任何可以正确编码的数据：

```
let hello = "السلام عليكم";
let hello = "Dobry den";
let hello = "Hello";
let hello = "こんにちは";
let hello = "नमस्ते";
let hello = "こんにちは";
let hello = "안녕하세요";
let hello = "你好";
let hello = "Olá";
let hello = "Здравствуйте";
let hello = "Hola";
```

更新字符串

`String` 的大小可以增长其内容也可以改变，就像可以放入更多数据来改变 `Vec` 的内容一样。另外，`String` 实现了 `+` 运算符作为级联运算符以便于使用。

使用 **push** 附加字符串

可以通过 `push_str` 方法来附加字符串 **slice**，从而使 `String` 变长：

```
let mut s = String::from("foo");
s.push_str("bar");
```

执行这两行代码之后 `s` 将会包含“foobar”。`push_str` 方法获取字符串 **slice**，因为我们并不需要获取参数的所有权。例如，如果将 `s2` 的内容附加到 `s1` 中后自身不能被使用就糟糕了：

```
let mut s1 = String::from("foo");
let s2 = String::from("bar");
s1.push_str(&s2);
```

`push` 方法被定义为获取一个单独的字符作为参数，并附加到 `String` 中：

```
let mut s = String::from("lo");
s.push('l');
```

执行这些代码之后，`s` 将会包含“lol”。

使用 **+** 运算符或 **format!** 宏级联字符串

通常我们希望将两个已知的字符串合并在一起。一种办法是像这样使用 `+` 运算符：

```
let s1 = String::from("Hello, ");
```

```
let s2 = String::from("world!");
let s3 = s1 + &s2; // Note that s1 has been moved here and can no longer be used
```

执行完这些代码之后字符串 `s3` 将会包含 `Hello, world!`。 `s1` 在相加后不再有效的原因，和使用 `s2` 的引用的原因与使用 `+` 运算符时调用的方法签名有关，这个函数签名看起来像这样：

```
fn add(self, s: &str) -> String {
```

这并不是标准库中实际的签名；那个 `add` 使用泛型定义。这里我们看到的 `add` 的签名使用具体类型代替了泛型，这也正是当使用 `String` 值调用这个方法会发生的。第十章会讨论泛型。这个签名提供了理解 `+` 运算那奇怪的部分的线索。

首先，`s2` 使用了 `&`，意味着我们使用第二个字符串的引用与第一个字符串相加。这是因为 `add` 函数的 `s` 参数：只能将 `&str` 和 `String` 相加，不能将两个 `String` 值相加。不过等一下——正如 `add` 的第二个参数所指定的，`&s2` 的类型是 `&String` 而不是 `&str`。那么为什么代码还能编译呢？之所以能够在 `add` 调用中使用 `&s2` 是因为 `&String` 可以被强转（*coerced*）成 `&str`——当 `add` 函数被调用时，`Rust` 使用了一个被称为解引用强制多态（*deref coercion*）的技术，你可以将其理解为它把 `&s2` 变成了 `&s2[..]` 以供 `add` 函数使用。第十五章会更深入的讨论解引用强制多态。因为 `add` 没有获取参数的所有权，所以 `s2` 在这个操作后仍然是有效的 `String`。

其次，可以发现签名中 `add` 获取了 `self` 的所有权，因为 `self` 没有使用 `&`。这意味着上面例子中的 `s1` 的所有权将被移动到 `add` 调用中，之后就不再有效。所以虽然 `let s3 = s1 + &s2;` 看起来就像它会复制两个字符串并创建一个新的字符串，而实际上这个语句会获取 `s1` 的所有权，附加上从 `s2` 中拷贝的内容，并返回结果的所有权。换句话说，它看起来好像生成了很多拷贝不过实际上并没有：这个实现比拷贝要更高效。

如果想要级联多个字符串，`+` 的行为就显得笨重了：

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = s1 + "-" + &s2 + "-" + &s3;
```

这时 `s` 的内容会是 `"tic-tac-toe"`。在有这么多 `+` 和 `"` 字符的情况下，很难理解具体发生了什么。对于更为复杂的字符串链接，可以使用 `format!` 宏：

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = format!("{}", s1, s2, s3);
```

这些代码也会将 `s` 设置为 `"tic-tac-toe"`。`format!` 与 `println!` 的工作原理相同，不过不同于将输出打印到屏幕上，它返回一个带有结果的 `String`。这个版本就好理解的多，并且不会获取任何参数的所有权。

索引字符串

在很多语言中，通过索引来引用字符串中的单独字符是有效且常见的操作。然而在 `Rust` 中，如果我们尝试使用索引语法访问 `String` 的一部分，会出现一个错误。比如如下代码：

```
let s1 = String::from("hello");
let h = s1[0];
```

会导致如下错误：

```
error: the trait bound `std::string::String: std::ops::Index<_>` is not
satisfied [--explain E0277]
  |>
  |>   let h = s1[0];
  |>           ^^^^^
note: the type `std::string::String` cannot be indexed by `_`
```

错误和提示说明了全部问题：**Rust** 的字符串不支持索引。那么接下来的问题是，为什么不支持呢？为了回答这个问题，我们必须先聊一聊 **Rust** 是如何在内存中储存字符串的。

内部表示

`String` 是一个 `Vec<u8>` 的封装。让我们看看之前一些正确编码的字符串的例子。首先是这一个：

```
let len = String::from("Hola").len();
```

在这里，`len` 的值是四，这意味着储存字符串“Hola”的 `Vec` 的长度是四个字节：每一个字符的 **UTF-8** 编码都占用一个字节。那下面这个例子又如何呢？

```
let len = String::from("Здравствуйτε").len();
```

当问及这个字符是多长的时候有人可能会说是 12。然而，**Rust** 的回答是 24。这是使用 **UTF-8** 编码“Здравствуйτε”所需要的字节数，这是因为每个 **Unicode** 标量值需要两个字节存储。因此一个字符串字节值的索引并不总是对应一个有效的 **Unicode** 标量值。

作为演示，考虑如下无效的 **Rust** 代码：

```
let hello = "Здравствуйτε";
let answer = &hello[0];
```

`answer` 的值应该是什么呢？它应该是第一个字符 `з` 吗？当使用 **UTF-8** 编码时，`з` 的第一个字节 `208`，第二个是 `151`，所以 `answer` 实际上应该是 `208`，不过 `208` 自身并不是一个有效的字母。返回 `208` 可不是一个请求字符串第一个字母的人所希望看到的，不过它是 **Rust** 在字节索引零位置所能提供的唯一数据。返回字节值可能不是人们希望看到的，即便是只有拉丁字母时：`&"hello"[0]` 会返回 `104` 而不是 `h`。为了避免返回意想不到值并造成不能立刻发现的 **bug**。**Rust** 选择不编译这些代码并及早杜绝了误会的发生。

字节、标量值和字形簇！天呐！

这引起了关于 **UTF-8** 的另外一个问题：从 **Rust** 的角度来讲，事实上有三种相关方式可以理解字符串：字节、标量值和字形簇（最接近人们眼中“字母”的概念）。

比如这个用梵文书写的印度语单词“नमस्ते”，最终它储存在 `Vec` 中的 `u8` 值看起来像这样：

```
[224, 164, 168, 224, 164, 174, 224, 164, 184, 224, 165, 141, 224, 164, 164,
224, 165, 135]
```

这里有 18 个字节，也就是计算机最终会储存的数据。如果从 **Unicode** 标量值的角度理解它们，也就像 **Rust** 的 `char` 类型那样，这些字节看起来像这样：

```
['न', 'म', 'स्', 'ते', 'ः']
```

这里有六个 `char`，不过第四个和第六个都不是字母，它们是发音符本身并没有任何意义。最后，如果以字形簇的角度理解，就会得到人们所说的构成这个单词的四个字母：

```
["न", "म", "स्", "ते"]
```

Rust 提供了多种不同的方式来解释计算机储存的原始字符串数据，这样程序就可以选择它需要的表现方式，而无所谓是何种人类语言。

最后一个 **Rust** 不允许使用索引获取 `String` 字符的原因是索引操作预期总是需要常数时间 ($O(1)$)。但是对于 `String` 不可能保证这样的性能，因为 **Rust** 不得不检查从字符串的开头到索引位置的内容来确定这里有多少有效的字符。

字符串 slice

因为字符串索引应该返回的类型是不明确的，而且索引字符串通常也是一个坏点子，所以 **Rust** 不建议这么做，而如果你确实需要它的话则需要更加明确一些。比使用 `[]` 和单个值的索引更加明确的方式是使用 `[]` 和一个 `range` 来创建包含特定字节的字符串 slice：

```
let hello = "Здравствуйте";  
  
let s = &hello[0..4];
```

这里，`s` 是一个 `&str`，它包含字符串的头四个字节。早些时候，我们提到了这些字母都是两个字节长的，所以这意味着 `s` 将会是“Зд”。

如果获取 `&hello[0..1]` 会发生什么呢？答案是：在运行时会 **panic**，就跟访问 `vector` 中的无效索引时一样：

```
thread 'main' panicked at 'index 0 and/or 1 in `Здравствуйте` do not lie on  
character boundary', ../src/libcore/str/mod.rs:1694
```

你应该小心谨慎的使用这个操作，因为它可能会使你的程序崩溃。

遍历字符串的方法

幸运的是，这里还有其他获取字符串元素的方式。

如果你需要操作单独的 **Unicode** 标量值，最好的选择是使用 `chars` 方法。对“नमस्ते”调用 `chars` 方法会将其分开并返回六个 `char` 类型的值，接着就可以遍历结果来访问每一个元素了：

```
for c in "नमस्ते".chars() {  
    println!("{}", c);  
}
```

这些代码会打印出如下内容：

```
न  
म  
स्  
ते  
ँ
```


`bytes` 方法返回每一个原始字节，这可能会适合你的使用场景：

```
for b in "नमस्ते".bytes() {  
    println!("{}", b);  
}
```

这些代码会打印出组成 `String` 的 18 个字节，开头是这样的：

```
224  
164  
168  
224  
// ... etc
```

不过请记住有效的 **Unicode** 标量值可能会由不止一个字节组成。

从字符串中获取字形簇是很复杂的，所以标准库并没有提供这个功能。crates.io 上有些提供这样功能的 `crate`。

字符串并不简单

总而言之，字符串还是很复杂的。不同的语言选择了不同的向程序员展示其复杂性的方式。**Rust** 选择了以准确的方式处理 `String` 数据作为所有 **Rust** 程序的默认行为，这意味着程序员们必须更多的思考如何在前台处理 **UTF-8** 数据。这种权衡取舍相比其他语言更多的暴露出了字符串的复杂性，不过也使你在开发生命周期中免于处理涉及非 **ASCII** 字符的错误。

现在让我们转向一些不太复杂的集合：哈希 `map`！

哈希 map

ch08-03-hash-maps.md

commit d06a6a181fd61704cbf7feb55bc61d518c6469f9

最后介绍的常用集合类型是 哈希 **map** (*hash map*)。 `HashMap<K, V>` 类型储存了一个键类型 `K` 对应一个值类型 `V` 的映射。它通过一个 哈希函数 (*hashing function*) 来实现映射，决定如何将键和值放入内存中。很多编程语言支持这种数据结构，不过通常有不同的名字：哈希、map、对象、哈希表或者关联数组，仅举几例。

哈希 **map** 可以用于需要任何类型作为键来寻找数据的情况，而不是像 **vector** 那样通过索引。例如，在一个游戏中，你可以将每个团队的分数记录到哈希 **map** 中，其中键是队伍的名字而值是每个队伍的分数。给出一个队名，就能得到它们的得分。

本章我们会介绍哈希 **map** 的基本 API，不过还有更多吸引人的功能隐藏于标准库中 `HashMap` 定义的函数中。请一如既往地查看标准库文档来了解更多信息。

新建一个哈希 map

可以使用 `new` 创建一个空的 `HashMap`，并使用 `insert` 增加元素。这里我们记录两支队伍的分数，分别是蓝队和黄队。蓝队开始有 10 分而黄队开始有 50 分：

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

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);
```

注意必须首先 `use` 标准库中集合部分的 `HashMap`。在这三个常用集合中，`HashMap` 是最不常用的，所以并没有被 `prelude` 自动引用。标准库中对 `HashMap` 的支持也相对较少，例如，并没有内建的构建宏。像 **vector** 一样，哈希 **map** 将它们的数据储存在堆上，这个 `HashMap` 的键类型是 `String` 而值类型是 `i32`。同样类似于 **vector**，哈希 **map** 是同质的：所有的键必须是相同类型，值也必须都是相同类型。

另一个构建哈希 **map** 的方法是使用一个元组的 **vector** 的 `collect` 方法，其中每个元组包含一个键值对。`collect` 方法可以将数据收集进一系列的集合类型，包括 `HashMap`。例如，如果队伍的名字和初始分数分别在两个 **vector** 中，可以使用 `zip` 方法来创建一个元组的 **vector**，其中“Blue”与 10 是一对，依此类推。接着就可以使用 `collect` 方法将这个元组 **vector** 转换成一个 `HashMap`：

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

let teams = vec![String::from("Blue"), String::from("Yellow")];
let initial_scores = vec![10, 50];

let scores: HashMap<_, _> = teams.iter().zip(initial_scores.iter()).collect();
```

这里 `HashMap<_, _>` 类型注解是必要的，因为可能 `collect` 很多不同的数据结构，而除非显式指定否则 **Rust** 无从得知你需要的类型。但是对于键和值的类型参数来说，可以使用下划线占位，而 **Rust** 能够根据 **vector** 中数据的类型推断出 `HashMap` 所包含的类型。

哈希 map 和所有权

对于像 `i32` 这样的实现了 `Copy` `trait` 的类型，其值可以拷贝进哈希 `map`。对于像 `String` 这样拥有所有权的值，其值将被移动而哈希 `map` 会成为这些值的所有者：

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

let field_name = String::from("Favorite color");
let field_value = String::from("Blue");

let mut map = HashMap::new();
map.insert(field_name, field_value);
// field_name and field_value are invalid at this point
```

当 `insert` 调用将 `field_name` 和 `field_value` 移动到哈希 `map` 中后，将不能使用这两个绑定。

如果将值的引用插入哈希 `map`，这些值本身将不会被移动进哈希 `map`。但是这些引用指向的值必须至少在哈希 `map` 有效时也是有效的。第十章生命周期部分将会更多的讨论这个问题。

访问哈希 `map` 中的值

可以通过 `get` 方法并提供对应的键来从哈希 `map` 中获取值：

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

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

let team_name = String::from("Blue");
let score = scores.get(&team_name);
```

这里，`score` 是与蓝队分数相关的值，应为 `Some(10)`。因为 `get` 返回 `Option<V>`，所以结果被装进 `Some`；如果某个键在哈希 `map` 中没有对应的值，`get` 会返回 `None`。这时就要用某种第六章提到的方法来处理 `Option`。

可以使用与 `vector` 类似的方式来遍历哈希 `map` 中的每一个键值对，也就是 `for` 循环：

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

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

for (key, value) in &scores {
    println!("{: {}:", key, value);
}
```

这会以任意顺序打印出每一个键值对：

```
Yellow: 50
Blue: 10
```

更新哈希 `map`

尽管键值对的数量是可以增长的，不过任何时候，每个键只能关联一个值。当我们想要改变哈希 `map` 中的数据时，必须决定如何处理一个键已经有值了的情况。可以选择完全无视旧值并用新值代替旧值。可以选择保留旧值而忽略新值，并只在键 没有 对应值时增加新值。或者可以结合新旧两值。让我们看看这分别该如何处理！

覆盖一个值

如果我们插入了一个键值对，接着用相同的键插入一个不同的值，与这个键相关联的旧值将被替换。即便下面的代码调用了两次 `insert`，哈希 `map` 也只会包含一个键值对，因为两次都是对蓝队的键插入的值：

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

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Blue"), 25);

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

这会打印出 `{"Blue": 25}`。原始的值 `10` 将被覆盖。

只在键没有对应值时插入

我们经常会检查某个特定的键是否有值，如果没有就插入一个值。为此哈希 `map` 有一个特有的 API，叫做 `entry`，它获取我们想要检查的键作为参数。`entry` 函数的返回值是一个枚举，`Entry`，它代表了可能存在也可能不存在的值。比如说我们想要检查黄队的键是否关联了一个值。如果没有，就插入值 `50`，对于蓝队也是如此。使用 `entry` API 的代码看起来像这样：

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

let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);

scores.entry(String::from("Yellow")).or_insert(50);
scores.entry(String::from("Blue")).or_insert(50);

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

`Entry` 的 `or_insert` 方法在键对应的值存在时就返回这个值的 `Entry`，如果不存在则将参数作为新值插入并返回修改过的 `Entry`。这比编写自己的逻辑要简明的多，另外也与借用检查器结合得更好。

这段代码会打印出 `{"Yellow": 50, "Blue": 10}`。第一个 `entry` 调用会插入黄队的键和值 `50`，因为黄队并没有一个值。第二个 `entry` 调用不会改变哈希 `map` 因为蓝队已经有了值 `10`。

根据旧值更新一个值

另一个常见的哈希 `map` 的应用场景是找到一个键对应的值并根据旧的值更新它。例如，如果我们想要计数一些文本中每一个单词分别出现了多少次，就可以使用哈希 `map`，以单词作为键并递增其值来记录我们遇到过几次这个单词。如果是第一次看到某个单词，就插入值 `0`。

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

let text = "hello world wonderful world";

let mut map = HashMap::new();
```

```

for word in text.split_whitespace() {
    let count = map.entry(word).or_insert(0);
    *count += 1;
}

println!("{:?}", map);

```

这会打印出 `{"world": 2, "hello": 1, "wonderful": 1}`，`or_insert` 方法事实上会返回这个键的值的可变引用（`&mut V`）。这里我们将这个可变引用储存在 `count` 变量中，所以为了赋值必须首先使用星号（`*`）解引用 `count`。这个可变引用在 `for` 循环的结尾离开作用域，这样所有这些改变都是安全的并符合借用规则。

哈希函数

`HashMap` 默认使用一种密码学安全的哈希函数，它可以抵拒绝服务（**Denial of Service, DoS**）攻击。然而并不是最快的，不过为了更高的安全性值得付出一些性能的代价。如果性能监测显示此哈希函数非常慢，以致于你无法接受，你可以指定一个不同的 *hasher* 来切换为其它函数。*hasher* 是一个实现了 `BuildHasher trait` 的类型。第十章会讨论 `trait` 和如何实现它们。你并不需要从头开始实现你自己的 *hasher*；crates.io 有其他人分享的实现了许多常用哈希算法的 *hasher* 的库。

总结

`vector`、字符串和哈希 `map` 会在你的程序需要储存、访问和修改数据时帮助你。这里有一些你应该能够解决的练习问题：

- 给定一系列数字，使用 `vector` 并返回这个列表的平均数（**mean, average**）、中位数（排列数组后位于中间的值）和众数（**mode**，出现次数最多的值；这里哈希函数会很有帮助）。
- 将字符串转换为 **Pig Latin**，也就是每一个单词的第一个辅音字母被移动到单词的结尾并增加“**ay**”，所以“**first**”会变成“**irst-fay**”。元音字母开头的单词则在结尾增加“**hay**”（“**apple**”会变成“**apple-hay**”）。牢记 **UTF-8** 编码！
- 使用哈希 `map` 和 `vector`，创建一个文本接口来允许用户向公司的部门中增加员工的名字。例如，“**Add Sally to Engineering**”或“**Add Amir to Sales**”。接着让用户获取一个部门的所有员工的列表，或者公司每个部门的所有员工按照字母顺排序的列表。

标准库 **API** 文档中描述的这些类型的方法将有助于你进行这些练习！

我们已经开始接触可能会有失败操作的复杂程序了，这也意味着接下来是一个了解错误处理的绝佳时机！

错误处理

[ch09-00-error-handling.md](#)

commit 4f2dc564851dc04b271a2260c834643dfd86c724

Rust 对可靠性的执着也扩展到了错误处理。错误对于软件来说是不可避免的，所以当出现错误时，**Rust** 有很多特性来处理当前情况。在很多情况下，**Rust** 要求你承认出错的可能性并在编译代码之前就采取行动。通过确保不会只有在将代码部署到生产环境之后才会发现错误来使得程序更可靠。

Rust 将错误组合成两个主要类别：可恢复错误（*recoverable*）和不可恢复错误（*unrecoverable*）。可恢复错误通常代表向用户报告错误和重试操作是合理的情况，比如未找到文件。不可恢复错误通常是 **bug** 的同义词，比如尝试访问超过数组结尾的位置。

大部分语言并不区分这两类错误，并采用类似异常这样方式统一处理他们。**Rust** 并没有异常。相反，对于可恢复错误有 `Result<T, E>` 值，以及 `panic!`，它在遇到不可恢复错误时停止程序执行。这一章会首先介绍 `panic!` 调用，接着会讲到如何返回 `Result<T, E>`。最后，我们会讨论当决定是尝试从错误中恢复还是停止执行时需要顾及的权衡考虑。

panic! 与不可恢复的错误

[ch09-01-unrecoverable-errors-with-panic.md](#)

commit 8d24b2a5e61b4eea109d26e38d2144408ae44e53

突然有一天，糟糕的事情发生了，而你对此束手无策。对于这种情况，Rust 有 `panic!` 宏。当执行这个宏时，程序会打印出一个错误信息，展开并清理栈数据，然后接着退出。出现这种情况的场景通常是检测到一些类型的 bug 而且程序员并不清楚该如何处理它。

Panic 中的栈展开与终止

当出现 `panic!` 时，程序默认会开始展开（*unwinding*），这意味着 Rust 会回溯栈并清理它遇到的每一个函数的数据，不过这个回溯并清理的过程有很多工作。另一种选择是直接终止（*abort*），这会不清理数据就退出程序。那么程序所使用的内存需要由操作系统来清理。如果你需要项目的最终二进制文件越小越好，可以由 `panic` 时展开切换为终止，通过在 *Cargo.toml* 的 `[profile]` 部分增加 `panic = 'abort'`。例如，如果你想要在发布模式中 `panic` 时直接终止：

```
[profile.release]
panic = 'abort'
```

让我们在一个简单的程序中调用 `panic!`：

文件名: `src/main.rs`

```
fn main() {
    panic!("crash and burn");
}
```

运行程序将会出现类似这样的输出：

```
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
   Finished dev [unoptimized + debuginfo] target(s) in 0.25 secs
   Running `target/debug/panic`
thread 'main' panicked at 'crash and burn', src/main.rs:2
note: Run with `RUST_BACKTRACE=1` for a backtrace.
error: Process didn't exit successfully: `target/debug/panic` (exit code: 101)
```

最后三行包含 `panic!` 造成的错误信息。第一行显示了 `panic` 提供的信息并指明了源码中 `panic` 出现的位置：`src/main.rs:2` 表明这是 `src/main.rs` 文件的第二行。

在这个例子中，被指明的那一行是我们代码的一部分，而且查看这一行的话就会发现 `panic!` 宏的调用。换句话说，`panic!` 可能会出现在我们的代码调用的代码中。错误信息报告的文件名和行号可能指向别人代码中的 `panic!` 宏调用，而不是我们代码中最终导致 `panic!` 的那一行。可以使用 `panic!` 被调用的函数的 *backtrace* 来寻找（我们代码中出问题的地方）。

使用 `panic!` 的 *backtrace*

让我们来看看另一个因为我们代码中的 bug 引起的别的库中 `panic!` 的例子，而不是直接的宏调用：

文件名: `src/main.rs`

```
fn main() {
    let v = vec![1, 2, 3];

    v[100];
}
```

这里尝试访问 **vector** 的第一个元素，不过它只有三个元素。这种情况下 **Rust** 会 **panic**。[] 应当返回一个元素，不过如果传递了一个无效索引，就没有可供 **Rust** 返回的正确的元素。

这种情况下其他像 **C** 这样语言会尝试直接提供所要求的值，即便这可能不是你期望的：你会得到任何对应 **vector** 中这个元素的内存位置的值，甚至是这些内存并不属于 **vector** 的情况。这被称为 **缓冲区溢出**（**buffer overread**），并可能会导致安全漏洞，比如攻击者可以像这样操作索引来读取储存在数组后面不被允许的数据。

为了使程序远离这类漏洞，如果尝试读取一个索引不存在的元素，**Rust** 会停止执行并拒绝继续。尝试运行上面的程序会出现如下：

```
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
   Finished dev [unoptimized + debuginfo] target(s) in 0.27 secs
    Running `target/debug/panic`
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 100', /stable-dist-rustc/build/src/libcollections/vec.rs:1362
note: Run with `RUST_BACKTRACE=1` for a backtrace.
error: Process didn't exit successfully: `target/debug/panic` (exit code: 101)
```

这指向了一个不是我们编写的文件，**libcollections/vec.rs**。这是标准库中 `Vec<T>` 的实现。这是当对 **vector** **v** 使用 [] 时 **libcollections/vec.rs** 中会执行的代码，也是真正出现 **panic!** 的地方。

接下来的几行提醒我们可以设置 **RUST_BACKTRACE** 环境变量来得到一个 **backtrace** 来调查究竟是什么导致了错误。让我们来试试看。示例 9-1 显示了其输出：

```
$ RUST_BACKTRACE=1 cargo run
   Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running `target/debug/panic`
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 100', /stable-dist-rustc/build/src/libcollections/vec.rs:1392
stack backtrace:
 1: 0x560ed90ec04c - std::sys::imp::backtrace::tracing::imp::write::hf33ae72d0baa11ed
    at /stable-dist-rustc/build/src/libstd/sys/unix/backtrace/tracing/gcc_s.rs:42
 2: 0x560ed90ee03e - std::panicking::default_hook:{{closure}}::h59672b733cc6a455
    at /stable-dist-rustc/build/src/libstd/panicking.rs:351
 3: 0x560ed90edc44 - std::panicking::default_hook::h1670459d2f3f8843
    at /stable-dist-rustc/build/src/libstd/panicking.rs:367
 4: 0x560ed90ee41b - std::panicking::rust_panic_with_hook::hcf0ddb069e7abcd7
    at /stable-dist-rustc/build/src/libstd/panicking.rs:555
 5: 0x560ed90ee2b4 - std::panicking::begin_panic::hd6eb68e27bdf6140
    at /stable-dist-rustc/build/src/libstd/panicking.rs:517
 6: 0x560ed90ee1d9 - std::panicking::begin_panic_fmt::abcd5965948b877f8
    at /stable-dist-rustc/build/src/libstd/panicking.rs:501
 7: 0x560ed90ee167 - rust_begin_unwind
    at /stable-dist-rustc/build/src/libstd/panicking.rs:477
 8: 0x560ed911401d - core::panicking::panic_fmt::hc0f6d7b2c300cdd9
    at /stable-dist-rustc/build/src/libcore/panicking.rs:69
 9: 0x560ed9113fc8 - core::panicking::panic_bounds_check::h02a4af86d01b3e96
    at /stable-dist-rustc/build/src/libcore/panicking.rs:56
10: 0x560ed90e71c5 - <collections::vec::Vec<T> as core::ops::Index<usize>>::index::h98abcd4e2a74c41
    at /stable-dist-rustc/build/src/libcollections/vec.rs:1392
11: 0x560ed90e727a - panic::main::h5d6b77c20526bc35
    at /home/you/projects/panic/src/main.rs:4
12: 0x560ed90f5d6a - __rust_maybe_catch_panic
```



```

                                at /stable-dist-rustc/build/src/libpanic_unwind/lib.rs:98
13: 0x560ed90ee926 - std::rt::lang_start::hd7c880a37a646e81
                                at /stable-dist-rustc/build/src/libstd/panicking.rs:436
                                at /stable-dist-rustc/build/src/libstd/panic.rs:361
                                at /stable-dist-rustc/build/src/libstd/rt.rs:57
14: 0x560ed90e7302 - main
15: 0x7f0d53f16400 - __libc_start_main
16: 0x560ed90e6659 - _start
17: 0x0 - <unknown>

```

示例 9-1: 当设置 `RUST_BACKTRACE` 环境变量时 `panic!` 调用所生成的 `backtrace` 信息

这里有大量的输出！`backtrace` 第 11 行指向了我们程序中引起错误的行：`src/main.rs` 的第四行。`backtrace` 是一个执行到目前位置所有被调用的函数的列表。`Rust` 的 `backtrace` 跟其他语言中的一样：阅读 `backtrace` 的关键是从头开始读直到发现你编写的文件。这就是问题的发源地。这一行往上是你的代码调用的代码；往下则是调用你的代码的代码。这些行可能包含核心 `Rust` 代码，标准库代码或用到的 `crate` 代码。

如果你不希望我们的程序 `panic`，第一个提到我们编写的代码行的位置是你应该开始调查的，以便查明是什么值如何在这个地方引起了 `panic`。在上面的例子中，我们故意编写会 `panic` 的代码来演示如何使用 `backtrace`，修复这个 `panic` 的方法就是不要尝试在一个只包含三个项的 `vector` 中请求索引是 100 的元素。当将来你的代码出现了 `panic`，你需要搞清楚在这特定的场景下代码中执行了什么操作和什么值导致了 `panic`，以及应当如何处理才能避免这个问题。

本章的后面会再次回到 `panic!` 并讲到何时应该何时不应该使用这个方式。接下来，我们来看看如何使用 `Result` 来从错误中恢复。

Result 与可恢复的错误

[ch09-02-recoverable-errors-with-result.md](#)

commit e6d6caab41471f7115a621029bd428a812c5260e

大部分错误并没有严重到需要程序完全停止执行。有时，一个函数会因为一个容易理解并做出反应的原因失败。例如，如果尝试打开一个文件不过由于文件并不存在而操作失败，这时我们可能想要创建这个文件而不是终止进程。

回忆一下第二章“使用 `Result` 类型来处理潜在的错误”部分中的那个 `Result` 枚举，它定义有如下两个成员，`Ok` 和 `Err`：

```
enum Result<T, E> {  
    Ok(T),  
    Err(E),  
}
```

`T` 和 `E` 是泛型类型参数；第十章会详细介绍泛型。现在你需要知道的就是 `T` 代表成功时返回的 `Ok` 成员中的数据的类型，而 `E` 代表失败时返回的 `Err` 成员中的错误的类型。因为 `Result` 有这些泛型类型参数，我们可以将 `Result` 类型和标准库中为其定义的函数用于很多不同的场景，这些情况中需要返回的成功值和失败值可能会各不相同。

让我们调用一个返回 `Result` 的函数，因为它可能会失败：如示例 9-2 所示打开一个文件：

文件名: `src/main.rs`

```
use std::fs::File;  
  
fn main() {  
    let f = File::open("hello.txt");  
}
```

示例 9-2：打开文件

如何知道 `File::open` 返回一个 `Result` 呢？我们可以查看标准库 API 文档，或者直接问编译器！如果给 `f` 某个我们知道不是函数返回值类型的类型注解，接着尝试编译代码，编译器会告诉我们类型不匹配。然后错误信息会告诉我们 `f` 的类型应该是什么，为此我们将 `let f` 语句改为：

```
let f: u32 = File::open("hello.txt");
```

现在尝试编译会给出如下错误：

```
error[E0308]: mismatched types  
--> src/main.rs:4:18  
  |  
4 |     let f: u32 = File::open("hello.txt");  
  |                   ^^^^^^^^^^^^^^^^^^^^^^^^^ expected u32, found enum  
  |  
  = note: expected type `u32`  
  = note: found type `std::result::Result<std::fs::File, std::io::Error>`  
`std::result::Result`  
  |  
  = note: expected type `u32`  
  = note: found type `std::result::Result<std::fs::File, std::io::Error>`
```

这就告诉我们了 `File::open` 函数的返回值类型是 `Result<T, E>`。这里泛型参数 `T` 放入了成功值的类型 `std::fs::File`，它是一个文件句柄。`E` 被用在失败值上时其类型是 `std::io::Error`。

这个返回值类型说明 `File::open` 调用可能会成功并返回一个可以进行读写的文件句柄。这个函数也可能会失败：例如，文件可能并不存在，或者可能没有访问文件的权限。`File::open` 需要一个方式告诉我们是成功还是失败，并同时提供给我们文件句柄或错误信息。而这些信息正是 `Result` 枚举可以提供的。

当 `File::open` 成功的情况下，变量 `f` 的值将会是一个包含文件句柄的 `Ok` 实例。在失败的情况下，`f` 会是一个包含更多关于出现了何种错误信息的 `Err` 实例。

我们需要在示例 9-2 的代码中增加根据 `File::open` 返回值进行不同处理的逻辑。示例 9-3 展示了一个使用基本工具处理 `Result` 的例子：第六章学习过的 `match` 表达式。

文件名: `src/main.rs`

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

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => {
            panic!("There was a problem opening the file: {:?}", error)
        },
    };
}
```

示例 9-3：使用 `match` 表达式处理可能的 `Result` 成员

注意与 `Option` 枚举一样，`Result` 枚举和其成员也被导入到了 `prelude` 中，所以就不需要在 `match` 分支中的 `Ok` 和 `Err` 之前指定 `Result::`。

这里我们告诉 `Rust` 当结果是 `Ok` 时，返回 `Ok` 成员中的 `file` 值，然后将这个文件句柄赋值给变量 `f`。`match` 之后，我们可以利用这个文件句柄来进行读写。

`match` 的另一个分支处理从 `File::open` 得到 `Err` 值的情况。在这种情况下，我们选择调用 `panic!` 宏。如果当前目录没有一个叫做 `hello.txt` 的文件，当运行这段代码时会看到如下来自 `panic!` 宏的输出：

```
thread 'main' panicked at 'There was a problem opening the file: Error { repr:
Os { code: 2, message: "No such file or directory" } }', src/main.rs:8
```

匹配不同的错误

示例 9-3 中的代码不管 `File::open` 是因为什么原因失败都会 `panic!`。我们真正希望的是对不同的错误原因采取不同的行为：如果 `File::open` 因为文件不存在而失败，我们希望创建这个文件并返回新文件的句柄。如果 `File::open` 因为任何其他原因失败，例如没有打开文件的权限，我们仍然希望像示例 9-3 那样 `panic!`。让我们看看示例 9-4，其中 `match` 增加了另一个分支：

文件名: `src/main.rs`

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(ref error) if error.kind() == ErrorKind::NotFound => {
```

```

        match File::create("hello.txt") {
            Ok(fc) => fc,
            Err(e) => {
                panic!(
                    "Tried to create file but there was a problem: {:?}",
                    e
                )
            },
        }
    },
    Err(error) => {
        panic!(
            "There was a problem opening the file: {:?}",
            error
        )
    },
};
}

```

示例 9-4: 使用不同的方式处理不同类型的错误

`File::open` 返回的 `Err` 成员中的值类型 `io::Error`，它是一个标准库中提供的结构体。这个结构体有一个返回 `io::ErrorKind` 值的 `kind` 方法可供调用。`io::ErrorKind` 是一个标准库提供的枚举，它的成员对应 `io` 操作可能导致的的不同错误类型。我们感兴趣的成员是 `ErrorKind::NotFound`，它代表尝试打开的文件并不存在。

条件 `if error.kind() == ErrorKind::NotFound` 被称作 **match guard**：它是一个进一步完善 `match` 分支模式的额外的条件。这个条件必须为真才能使分支的代码被执行；否则，模式匹配会继续并考虑 `match` 中的下一个分支。模式中的 `ref` 是必须的，这样 `error` 就不会被移动到 `guard` 条件中而仅仅是引用它。第十八章会详细解释为什么在模式中使用 `ref` 而不是 `&` 来获取一个引用。简而言之，在模式的上下文中，`&` 匹配一个引用并返回它的值，而 `ref` 匹配一个值并返回一个引用。

在 `match guard` 中我们想要检查的条件是 `error.kind()` 是否是 `ErrorKind` 枚举的 `NotFound` 成员。如果是，尝试用 `File::create` 创建文件。然而 `File::create` 也可能会失败，还需要增加一个内部 `match` 语句。当文件不能被打开，会打印出一个不同的错误信息。外部 `match` 的最后一个分支保持不变这样对任何除了文件不存在的错误会使程序 `panic`。

失败时 `panic` 的捷径： `unwrap` 和 `expect`

`match` 能够胜任它的工作，不过它可能有点冗长并且不总是能很好的表明意图。`Result<T, E>` 类型定义了很多辅助方法来处理各种情况。其中之一叫做 `unwrap`，它的实现就类似于示例 9-3 中的 `match` 语句。如果 `Result` 值是成员 `Ok`，`unwrap` 会返回 `Ok` 中的值。如果 `Result` 是成员 `Err`，`unwrap` 会为我们调用 `panic!`。

```

use std::fs::File;

fn main() {
    let f = File::open("hello.txt").unwrap();
}

```

如果调用这段代码时不存在 `hello.txt` 文件，我们将会看到一个 `unwrap` 调用 `panic!` 时提供的错误信息：

```

thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: Error {
  repr: Os { code: 2, message: "No such file or directory" } }',
/stable-dist-rustc/build/src/libcore/result.rs:868

```

还有另一个类似于 `unwrap` 的方法它还允许我们选择 `panic!` 的错误信息：`expect`。使用 `expect` 而不是 `unwrap` 并提供一个好的错误信息可以表明你的意图并有助于追踪 `panic` 的根源。`expect` 的语法看起来像这样：

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

fn main() {
    let f = File::open("hello.txt").expect("Failed to open hello.txt");
}
```

`expect` 与 `unwrap` 的使用方式一样：返回文件句柄或调用 `panic!` 宏。`expect` 用来调用 `panic!` 的错误信息将会作为参数传递给 `expect`，而不像 `unwrap` 那样使用默认的 `panic!` 信息。它看起来像这样：

```
thread 'main' panicked at 'Failed to open hello.txt: Error { repr: Os { code:
2, message: "No such file or directory" } }',
/stable-dist-rustc/build/src/libcore/result.rs:868
```

传播错误

当编写一个其实现会调用一些可能会失败的操作的函数时，除了在这个函数中处理错误外，还可以选择让调用者知道这个错误并决定该如何处理。这被称为传播（*propagating*）错误，这样能更好的控制代码调用，因为比起你代码所拥有的上下文，调用者可能拥有更多信息或逻辑来决定应该如何处理错误。

例如，示例 9-5 展示了一个从文件中读取用户名的函数。如果文件不存在或不能读取，这个函数会将这些错误返回给调用它的代码：

```
use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<String, io::Error> {
    let f = File::open("hello.txt");

    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut s = String::new();

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}
```

示例 9-5：一个函数使用 `match` 将错误返回给代码调用者

首先让我们看看函数的返回值：`Result<String, io::Error>`。这意味着函数返回一个 `Result<T, E>` 类型的值，其中泛型参数 `T` 的具体类型是 `String`，而 `E` 的具体类型是 `io::Error`。如果这个函数没有出任何错误成功返回，函数的调用者会收到一个包含 `String` 的 `Ok` 值——函数从文件中读取到的用户名。如果函数遇到任何错误，函数的调用者会收到一个 `Err` 值，它储存了一个包含更多这个问题相关信息的 `io::Error` 实例。这里选择 `io::Error` 作为函数的返回值是因为它正好是函数体中那两个可能会失败的操作的错误返回值：`File::open` 函数和 `read_to_string` 方法。

函数体以 `File::open` 函数开头。接着使用 `match` 处理返回值 `Result`，类似于示例 9-3 中的 `match`，唯一的区别是当 `Err` 时不再调用 `panic!`，而是提早返回并将 `File::open` 返回的错误值作为函数的错误返回值传递给调用者。如果 `File::open` 成功了，我们将文件句柄储存在变量 `f` 中并继续。

接着我们在变量 `s` 中创建了一个新 `String` 并调用文件句柄 `f` 的 `read_to_string` 方法来将文件的内容读取到 `s` 中。`read_to_string` 方法也返回一个 `Result` 因为它也可能会失败：哪怕是 `File::open` 已经成功了。所以我们需要另一个 `match` 来处理这个 `Result`：如果 `read_to_string` 成功了，那么这个函数就成功了，并返回文件中的用户名，它现在位于被封装进 `Ok` 的 `s` 中。如果 `read_to_string` 失败了，则像之前处理 `File::open` 的返回值的 `match` 那样返回错误值。并不需要显式的调用 `return`，因为这是函数的最后一个表达式。

调用这个函数的代码最终会得到一个包含用户名的 `Ok` 值，或者一个包含 `io::Error` 的 `Err` 值。我们无从得知调用者会如何处理这些值。例如，如果他们得到了一个 `Err` 值，他们可能会选择 `panic!` 并使程序崩溃、使用一个默认的用户名或者从文件之外的地方寻找用户名。我们没有足够的信息知晓调用者具体会如何尝试，所以将所有的成功或失败信息向上传播，让他们选择合适的处理方法。

这种传播错误的模式在 **Rust** 是如此的常见，以至于有一个更简便的专用语法：`?`。

传播错误的捷径：`?`

示例 9-6 展示了一个 `read_username_from_file` 的实现，它实现了与示例 9-5 中的代码相同的功能，不过这个实现是使用了问号运算符的：

```
use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<String, io::Error> {
    let mut f = File::open("hello.txt")?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}
```

示例 9-6：一个使用 `?` 向调用者返回错误的函数

`Result` 值之后的 `?` 被定义为与示例 9-5 中定义的处理 `Result` 值的 `match` 表达式有着完全相同的工作方式。如果 `Result` 的值是 `Ok`，这个表达式将会返回 `Ok` 中的值而程序将继续执行。如果值是 `Err`，`Err` 中的值将作为整个函数的返回值，就好像使用了 `return` 关键字一样，这样错误值就被传播给了调用者。

在示例 9-6 的上下文中，`File::open` 调用结尾的 `?` 将会把 `Ok` 中的值返回给变量 `f`。如果出现了错误，`?` 会提早返回整个函数并将一些 `Err` 值传播给调用者。同理也适用于 `read_to_string` 调用结尾的 `?`。

`?` 消除了大量样板代码并使得函数的实现更简单。我们甚至可以在 `?` 之后直接使用链式方法调用来进一步缩短代码：

```
use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<String, io::Error> {
    let mut s = String::new();

    File::open("hello.txt")?.read_to_string(&mut s)?;

    Ok(s)
}
```

在 `s` 中创建新的 `String` 被放到了函数开头；这没有什么变化。我们对 `File::open("hello.txt")?` 的结果直接链式调用了 `read_to_string`，而不再创建变量 `f`。仍然需要 `read_to_string` 调用结尾的 `?`，而且当 `File::open` 和 `read_to_string` 都成功没有失败时返回包含用户名 `s` 的 `Ok` 值。其功能再一次与示例 9-5 和示例 9-6 保持一致，

不过这是一个与众不同且更符合工程学的写法。

? 只能被用于返回 `Result` 的函数

? 只能被用于返回值类型为 `Result` 的函数，因为他被定义为与示例 9-5 中的 `match` 表达式有着完全相同的工作方式。`match` 的 `return Err(e)` 部分要求返回值类型是 `Result`，所以函数的返回值必须是 `Result` 才能与这个 `return` 相兼容。

让我们看看在 `main` 函数中使用 ? 会发生什么，如果你还记得的话它的返回值类型是 `()`：

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

fn main() {
    let f = File::open("hello.txt");
}
```

当编译这些代码，会得到如下错误信息：

```
error[E0308]: mismatched types
-->
|
3 |     let f = File::open("hello.txt");
|               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected (), found enum
`std::result::Result`
|
= note: expected type `()`
= note:   found type `std::result::Result<_, _>`
```

错误指出存在不匹配的类型：`main` 函数返回一个 `()` 类型，而 ? 返回一个 `Result`。编写不返回 `Result` 的函数时，如果调用其他返回 `Result` 的函数，需要使用 `match` 或者 `Result` 的方法之一来处理它，而不能用 ? 将潜在的错误传播给调用者。

现在我们讨论过了调用 `panic!` 或返回 `Result` 的细节，是时候返回他们各自适合哪些场景的话题了。

panic! 还是不 panic!

[ch09-03-to-panic-or-not-to-panic.md](#)

commit 88dee413c0792dcaf563d753e00400cc191f1fbe

那么，该如何决定何时应该 `panic!` 以及何时应该返回 `Result` 呢？如果代码 `panic`，就没有恢复的可能。你可以选择对任何错误场景都调用 `panic!`，不管是否有可能恢复，不过这样就是你代替调用者决定了这是不可恢复的。选择返回 `Result` 值的话，就将选择权交给了调用者，而不是代替他们做出决定。调用者可能会选择以符合他们场景的方式尝试恢复，或者也可能干脆就认为 `Err` 是不可恢复的，所以他们也可能会调用 `panic!` 并将可恢复的错误变成了不可恢复的错误。因此返回 `Result` 是定义可能会失败的函数的一个好的默认选择。

有一些情况 `panic` 比返回 `Result` 更为合适，不过他们并不常见。让我们讨论一下为何在示例、代码原型和测试中，以及那些人们认为不会失败而编译器不这么看的情况下，`panic` 是合适的，最后会总结一些在库代码中如何决定是否要 `panic` 的通用指导原则。

示例、代码原型和测试：非常适合 `panic`

当你编写一个示例来展示一些概念时，在拥有健壮的错误处理代码的同时也会使得例子不那么明确。例如，调用一个类似 `unwrap` 这样可能 `panic!` 的方法可以被理解为一个你实际希望程序处理错误方式的占位符，它根据其余代码运行方式可能会各不相同。

类似的，`unwrap` 和 `expect` 方法在原型设计时非常方便，在你决定该如何处理错误之前。他们在代码中留下了明显的记号，以便你准备使程序变得更健壮时作为参考。

如果方法调用在测试中失败了，我们希望这个测试都失败，即便这个方法并不是需要测试的功能。因为 `panic!` 是测试如何被标记为失败的，调用 `unwrap` 或 `expect` 都是非常有道理的。

当你比编译器知道更多的情况

当你有一些其他的逻辑来确保 `Result` 会是 `Ok` 值的时候调用 `unwrap` 也是合适的，虽然编译器无法理解这种逻辑。仍然会有一个 `Result` 值等着你处理：总的来说你调用的任何操作都有失败的可能性，即便在特定情况下逻辑上是不可能的。如果通过人工检查代码来确保永远也不会出现 `Err` 值，那么调用 `unwrap` 也是完全可以接受的，这里是一个例子：

```
use std::net::IpAddr;

let home = "127.0.0.1".parse::<IpAddr>().unwrap();
```

我们通过解析一个硬编码的字符来创建一个 `IpAddr` 实例。可以看出 `127.0.0.1` 是一个有效的 IP 地址，所以这里使用 `unwrap` 是没有问题的。然而，拥有一个硬编码的有效字符串也不能改变 `parse` 方法的返回值类型：它仍然是一个 `Result` 值，而编译器仍然就好像还是有可能出现 `Err` 成员那样要求我们处理 `Result`，因为编译器还没有智能到可以识别出这个字符串总是一个有效的 IP 地址。如果 IP 地址字符串来源于用户而不是硬编码进程中的话，那么就确实有失败的可能性，这时就绝对需要我们以一种更健壮的方式处理 `Result` 了。

错误处理指导原则

在当有可能导致有害状态的情况下建议使用 `panic!` ——在这里，有害状态是指当一些假设、保证、协议或不可变性被打破的状态，例如无效的值、自相矛盾的值或者被传递了不存在的值——外加如下几种情况：

- 有害状态并不包含 预期 会偶尔发生的错误

- 之后的代码的运行依赖于不再处于这种有害状态
- 当没有可行的手段来将有害状态信息编码进所使用的类型中的情况

如果别人调用你的代码并传递了一个没有意义的值，最好的情况也许就是 `panic!` 并警告使用你的库的人他的代码中有 `bug` 以便他能在开发时就修复它。类似的，`panic!` 通常适合调用不能够控制的外部代码时，这时无法修复其返回的无效状态。

无论代码编写的多么好，当有害状态是预期会出现时，返回 `Result` 仍要比调用 `panic!` 更为合适。这样的例子包括解析器接收到错误数据，或者 `HTTP` 请求返回一个表明触发了限流的状态。在这些例子中，应该通过返回 `Result` 来表明失败预期是可能的，这样将有害状态向上传播，这样调用者就可以决定该如何处理这个问题。使用 `panic!` 来处理这些情况就不是最好的选择。

当代码对值进行操作时，应该首先验证值是有效的，并在其无效时 `panic!`。这主要是出于安全的原因：尝试操作无效数据会暴露代码漏洞，这就是标准库在尝试越界访问数组时会 `panic!` 的主要原因：尝试访问不属于当前数据结构的内存是一个常见的安全隐患。函数通常都遵循 契约（*contracts*）：他们的行为只有在输入满足特定条件时才能得到保证。当违反契约时 `panic` 是有道理的，因为这通常代表调用方的 `bug`，而且这也不是那种你希望调用方必须处理的错误。事实上也没有合理的方式来恢复调用方的代码：调用方的 程序员 需要修复他的代码。函数的契约，尤其是当违反它会造成 `panic` 的契约，应该在函数的 `API` 文档中得到解释。

虽然在所有函数中都拥有许多错误检查是冗长而烦人的。幸运的是，可以利用 `Rust` 的类型系统（以及编译器的类型检查）为你进行很多检查。如果函数有一个特定类型的参数，可以在知晓编译器已经确保其拥有一个有效值的前提下进行你的代码逻辑。例如，如果你使用了一个不同于 `Option` 的类型，而且程序期望它是有值的并且不是空值。你的代码无需处理 `Some` 和 `None` 这两种情况，它只会有一种情况就是绝对会有一个值。尝试向函数传递空值的代码甚至根本不能编译，所以你的函数在运行时没有必要判空。另外一个例子是使用像 `u32` 这样的无符号整型，也会确保它永远不为负。

创建自定义类型作为验证

让我们借用 `Rust` 类型系统的思想来进一步确保值的有效性，并尝试创建一个自定义类型作为验证。回忆一下第二章的猜猜看游戏，它的代码请求用户猜测一个 1 到 100 之间的数字，在将其与秘密数字做比较之前我们事实上从未验证用户的猜测是位于这两个数字之间的，只保证它为正。在当前情况下，其影响并不是很严重：“Too high”或“Too low”的输出仍然是正确的。但是这是一个很好的引导用户得出有效猜测的辅助，例如当用户猜测一个超出范围的数字或者输入字母时采取不同的行为。

一种实现方式是将猜测解析成 `i32` 而不仅仅是 `u32`，来默许输入负数，接着检查数字是否在范围内：

```
loop {
    // snip

    let guess: i32 = match guess.trim().parse() {
        Ok(num) => num,
        Err(_) => continue,
    };

    if guess < 1 || guess > 100 {
        println!("The secret number will be between 1 and 100.");
        continue;
    }

    match guess.cmp(&secret_number) {
        // snip
    }
}
```

`if` 表达式检查了值是否超出范围，告诉用户出了什么问题，并调用 `continue` 开始下一次循环，请求另一个猜测。`if` 表达式之后，就可以在知道 `guess` 在 1 到 100 之间的情况下与秘密数字作比较了。

然而，这并不是一个理想的解决方案：程序只处理 1 到 100 之间的值是绝对不可取的，而且如果有很多函数都有这样的要求，在每个函数中都有这样的检查将是非常冗余的（并可能潜在的影响性能）。

相反我们可以创建一个新类型来将验证放入创建其实例的函数中，而不是到处重复这些检查。这样就可以安全的在函数签名中使用新类型并相信他们接收到的值。示例 9-8 中展示了一个定义 `Guess` 类型的方法，只有在 `new` 函数接收到 1 到 100 之间的值时才会创建 `Guess` 的实例：

```
pub struct Guess {
    value: u32,
}

impl Guess {
    pub fn new(value: u32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess {
            value
        }
    }

    pub fn value(&self) -> u32 {
        self.value
    }
}
```

示例 9-8：一个 `Guess` 类型，它只在值位于 1 和 100 之间时才继续

首先，我们定义了一个包含 `u32` 类型字段 `value` 的结构体 `Guess`。这里是储存猜测值的地方。

接着在 `Guess` 上实现了一个叫做 `new` 的关联函数来创建 `Guess` 的实例。`new` 定义为接收一个 `u32` 类型的参数 `value` 并返回一个 `Guess`。`new` 函数中代码的测试确保了其值是在 1 到 100 之间的。如果 `value` 没有通过测试则调用 `panic!`，这会警告调用这个函数的程序员有一个需要修改的 `bug`，因为创建一个 `value` 超出范围的 `Guess` 将会违反 `Guess::new` 所遵循的契约。`Guess::new` 会出现 `panic` 的条件应该在其公有 `API` 文档中被提及；第十四章会涉及到在 `API` 文档中表明 `panic!` 可能性的相关规则。如果 `value` 通过了测试，我们新建一个 `Guess`，其字段 `value` 将被设置为参数 `value` 的值，接着返回这个 `Guess`。

接着，我们实现了一个借用了 `self` 的方法 `value`，它没有任何其他参数并返回一个 `u32`。这类方法有时被称为 **getter**，因为它的目的就是返回对应字段的数据。这样的公有方法是必要的，因为 `Guess` 结构体的 `value` 字段是私有的。私有的字段 `value` 是很重要的，这样使用 `Guess` 结构体的代码将不允许直接设置 `value` 的值：调用者必须使用 `Guess::new` 方法来创建一个 `Guess` 的实例，这就确保了不会存在一个 `value` 没有通过 `Guess::new` 函数的条件检查的 `Guess`。

如此获取一个参数并只返回 1 到 100 之间数字的函数就可以声明为获取或返回一个 `Guess`，而不是 `u32`，同时其函数体中也无需进行任何额外的检查。

总结

`Rust` 的错误处理功能被设计为帮助你编写更加健壮的代码。`panic!` 宏代表一个程序无法处理的状态，并停止执行而不是使用无效或不正确的值继续处理。`Rust` 类型系统的 `Result` 枚举代表操作可能会在一种可以恢复的情况下失败。可以使用 `Result` 来告诉代码调用者他需要处理潜在的成功或失败。在适当的场景使用 `panic!` 和 `Result` 将会使你的代码在面对无处不在的错误时显得更加可靠。

现在我们已经见识过了标准库中 `Option` 和 `Result` 泛型枚举的能力了，让我们聊聊泛型是如何工作的，以及如何在你的代码中利用他们。

泛型、**trait** 和生命周期

ch10-00-generics.md

commit f65676e17d7fc4c0c7cd7275a7bf15447364831a

每一个编程语言都有高效的处理重复概念的工具；在 **Rust** 中工具之一就是 泛型（**generics**）。泛型是具体类型或其他属性的抽象替代。我们可以表达泛型的属性，比如他们的行为或如何与其他泛型相关联，而不需要在编写和编译代码时知道他们在这里实际上代表什么。

同理为了编写一份可以用于多种具体值的代码，函数并不知道其参数为何值，这时就可以让函数获取泛型而不是像 `i32` 或 `String` 这样的具体值。我们已经使用过第六章的 `Option<T>`，第八章的 `Vec<T>` 和 `HashMap<K, V>`，以及第九章的 `Result<T, E>` 这些泛型了。本章会探索如何使用泛型定义我们自己的类型、函数和方法！

首先，我们将回顾一下提取函数以减少代码重复的机制。接着使用一个只在参数类型上不同的泛型函数来实现相同的功能。我们也会讲到结构体和枚举定义中的泛型。

之后，我们讨论 **trait**，这是一个定义泛型行为的方法。**trait** 可以与泛型结合来将泛型限制为拥有特定行为的类型，而不是任意类型。

最后介绍 生命周期（**lifetimes**），它是一类允许我们向编译器提供引用如何相互关联的泛型。**Rust** 的生命周期功能允许在很多场景下借用值的同时仍然使编译器能够检查这些引用的有效性。

提取函数来减少重复

在介绍泛型语法之前，首先来回顾一个不使用泛型的处理重复的技术：提取一个函数。当熟悉了这个技术以后，我们将使用相同的机制来提取一个泛型函数！如同你识别出可以提取到函数中重复代码那样，你也会开始识别出能够使用泛型的重复代码。

考虑一下这个寻找列表中最大值的小程序，如示例 10-1 所示：

文件名: `src/main.rs`

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let mut largest = number_list[0];

    for number in number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {}", largest);
    # assert_eq!(largest, 100);
}
```

示例 10-1：在一个数字列表中寻找最大值的函数

这段代码获取一个整型列表，存放在变量 `number_list` 中。它将列表的第一项放入了变量 `largest` 中。接着遍历了列表中的所有数字，如果当前值大于 `largest` 中储存的值，将 `largest` 替换为这个值。如果当前值小于目前为止的最大值，`largest` 保持不变。当列表中所有值都被考虑到之后，`largest` 将会是最大值，在这里也就是 100。

如果需要在两个不同的列表中寻找最大值，我们可以重复示例 10-1 中的代码，这样程序中就会存在两段相同逻辑的代码，如示例 10-2 所示：

Filename: src/main.rs

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let mut largest = number_list[0];

    for number in number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {}", largest);

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let mut largest = number_list[0];

    for number in number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {}", largest);
}
```

示例 10-2: 寻找两个数字列表最大值的代码

虽然代码能够执行，但是重复的代码是冗余且容易出错的，并且意味着当更新逻辑时需要修改多处地方的代码。

为了消除重复，我们可以创建一层抽象，在这个例子中将表现为一个获取任意整型列表作为参数并对其进行处理函数。这将增加代码的简洁性并让我们将表达和推导寻找列表中最大值的这个概念与使用这个概念的特定位置相互独立。

在示例 10-3 的程序中将寻找最大值的代码提取到了一个叫做 `largest` 的函数中。这个程序可以找出两个不同数字列表的最大值，不过示例 10-1 中的代码只存在于一个位置：

文件名: src/main.rs

```
fn largest(list: &[i32]) -> i32 {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);
    # assert_eq!(result, 100);
}
```

```

let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

let result = largest(&number_list);
println!("The largest number is {}", result);
# assert_eq!(result, 6000);
}

```

示例 10-3: 抽象后的寻找两个数字列表最大值的代码

这个函数有一个参数 `list`，它代表会传递给函数的任何具体的 `i32` 值的 `slice`。函数定义中的 `list` 代表任何 `&[i32]`。当调用 `largest` 函数时，其代码实际上运行于我们传递的特定值上。

从示例 10-2 到示例 10-3 中涉及的机制经历了如下几步：

1. 我们注意到了重复代码。
2. 我们将重复代码提取到了一个函数中，并在函数签名中指定了代码中的输入和返回值。
3. 我们将两个具体的存在重复代码的位置替换为了函数调用。

在不同的场景使用不同的方式，我们也可以利用相同的步骤和泛型来减少重复代码。与函数体中现在作用于一个抽象的 `list` 而不是具体值一样，使用泛型的代码也会作用于抽象类型。支持泛型背后的概念与你已经了解的支持函数的概念是一样的，不过是实现方式不同。

如果我们有两个函数，一个寻找一个 `i32` 值的 `slice` 中的最大项而另一个寻找 `char` 值的 `slice` 中的最大项该怎么办？该如何消除重复呢？让我们拭目以待！

泛型数据类型

[ch10-01-syntax.md](#)

commit 56352c28cf3fe0402fa5a7cba73890e314d720eb

泛型用于通常我们放置类型的位置，比如函数签名或结构体，允许我们创建可以代替许多具体数据类型的结构体定义。让我们看看如何使用泛型定义函数、结构体、枚举和方法，并且在本部分的结尾我们会讨论泛型代码的性能。

在函数定义中使用泛型

定义函数时可以在函数签名的参数数据类型和返回值中使用泛型。以这种方式编写的代码将更灵活并能向函数调用者提供更多功能，同时不引入重复代码。

回到 `largest` 函数上，示例 10-4 中展示了两个提供了相同的寻找 `slice` 中最大值功能的函数。第一个是从示例 10-3 中提取的寻找 `slice` 中 `i32` 最大值的函数。第二个函数寻找 `slice` 中 `char` 的最大值：

文件名: `src/main.rs`

```
fn largest_i32(list: &[i32]) -> i32 {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn largest_char(list: &[char]) -> char {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest_i32(&number_list);
    println!("The largest number is {}", result);
    # assert_eq!(result, 100);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest_char(&char_list);
    println!("The largest char is {}", result);
    # assert_eq!(result, 'y');
}
```

示例 10-4：两个只在名称和签名中类型有所不同的函数

这里 `largest_i32` 和 `largest_char` 有着完全相同的函数体，所以如果能够将这两个函数变成一个来减少重复就好了。所幸通过引入一个泛型参数就能实现！

为了参数化要定义的函数的签名中的类型，我们需要像给函数的值参数起名那样为这类型参数起一个名字。这里选择了名称 `T`。任何标识符都可以作为类型参数名，选择 `T` 是因为 **Rust** 的类型命名规范是骆驼命名法（**CamelCase**）。另外泛型类型参数的规范也倾向于简短，经常仅仅是一个字母。`T` 作为“**type**”的缩写是大部分 **Rust** 程序员的首选。

当需要在函数体中使用一个参数时，必须在函数签名中声明这个参数以便编译器能知道函数体中这个名称的意义。同理，当在函数签名中使用一个类型参数时，必须在使用它之前就声明它。类型参数声明位于函数名称与参数列表中间的尖括号中。

我们将要定义的泛型版本的 `largest` 函数的签名看起来像这样：

```
fn largest<T>(list: &[T]) -> T {
```

这可以理解：函数 `largest` 有泛型类型 `T`。它有一个参数 `list`，它的类型是一个 `T` 值的 **slice**。`largest` 函数将会返回一个与 `T` 相同类型的值。

示例 10-5 展示一个在签名中使用了泛型的统一的 `largest` 函数定义，并向我们展示了如何对 `i32` 值的 **slice** 或 `char` 值的 **slice** 调用 `largest` 函数。注意这些代码还不能编译！

文件名: `src/main.rs`

```
fn largest<T>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

示例 10-5：一个还不能编译的使用泛型参数的 `largest` 函数定义

如果现在就尝试编译这些代码，会出现如下错误：

```
error[E0369]: binary operation `>` cannot be applied to type `T`
  |
5 |         if item > largest {
  |             ^^^^
  |
note: an implementation of `std::cmp::PartialOrd` might be missing for `T`
```


注释中提到了 `std::cmp::PartialOrd`，这是一个 *trait*。下一部分会讲到 *trait*，不过简单来说，这个错误表明 `largest` 的函数体不能适用于 `T` 的所有可能的类型；因为在函数体需要比较 `T` 类型的值，不过它只能用于我们知道如何排序的类型。标准库中定义的 `std::cmp::PartialOrd` *trait* 可以实现类型的排序功能。在下一部分会再次回到 *trait* 并讲解如何为泛型指定一个 *trait*，不过让我们先把这个例子放在一边并探索其他那些可以使用泛型类型参数的地方。

结构体定义中的泛型

同样也可以使用 `<>` 语法来定义拥有一个或多个泛型参数类型字段的结构体。示例 10-6 展示了如何定义和使用一个可以存放任何类型的 `x` 和 `y` 坐标值的结构体 `Point`：

文件名: `src/main.rs`

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

示例 10-6: `Point` 结构体存放了两个 `T` 类型的值 `x` 和 `y`

其语法类似于函数定义中使用泛型。首先，必须在结构体名称后面的尖括号中声明泛型参数的名称。接着在结构体定义中可以指定具体数据类型的位置使用泛型类型。

注意 `Point` 的定义中只使用了一个泛型类型，我们想要表达的是结构体 `Point` 对于一些类型 `T` 是泛型的，而且字段 `x` 和 `y` 都是相同类型的，无论它具体是何类型。如果尝试创建一个有不同类型值的 `Point` 的实例，像示例 10-7 中的代码就不能编译：

文件名: `src/main.rs`

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let wont_work = Point { x: 5, y: 4.0 };
}
```

示例 10-7: 字段 `x` 和 `y` 必须是相同类型，因为他们都有相同的泛型类型 `T`

尝试编译会得到如下错误：

```
error[E0308]: mismatched types
-->
|
7 |     let wont_work = Point { x: 5, y: 4.0 };
|                                     ^^^ expected integral variable, found
floating-point variable
= note: expected type `{integer}`
= note:   found type `{float}`
```

当我们将 `5` 赋值给 `x`，编译器就知道这个 `Point` 实例的泛型类型 `T` 是一个整型。接着我们将 `y` 指定为 `4.0`，而它被定义为与 `x` 有着相同的类型，所以出现了类型不匹配的错误。

如果想要定义一个 `x` 和 `y` 可以有不同类型且仍然是泛型的 `Point` 结构体，我们可以使用多个泛型类型参数。在示例 10-8 中，我们修改 `Point` 的定义为拥有两个泛型类型 `T` 和 `U`。其中字段 `x` 是 `T` 类型的，而字段 `y` 是 `U` 类型的：

文件名: `src/main.rs`

```
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```

示例 10-8: 使用两个泛型的 `Point`，这样 `x` 和 `y` 可能是不同类型

现在所有这些 `Point` 实例都是被允许的了！你可以在定义中使用任意多的泛型类型参数，不过太多的话代码将难以阅读和理解。如果你处于一个需要很多泛型类型的位置，这可能是一个需要重新组织代码并分隔成一些更小部分的信号。

枚举定义中的泛型数据类型

类似于结构体，枚举也可以在其成员中存放泛型数据类型。第六章我们使用过了标准库提供的 `Option<T>` 枚举，现在这个定义看起来就更容易理解了。让我们再看看：

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

换句话说 `Option<T>` 是一个拥有泛型 `T` 的枚举。它有两个成员：`Some`，它存放了一个类型 `T` 的值，和不存在任何值的 `None`。标准库中只有这一个定义来支持创建任何具体类型的枚举值。“一个可能的值”是一个比具体类型的值更抽象的概念，而 `Rust` 允许我们不引入重复代码就能表现抽象的概念。

枚举也可以拥有多个泛型类型。第九章使用过的 `Result` 枚举定义就是一个这样的例子：

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

`Result` 枚举有两个泛型类型，`T` 和 `E`。`Result` 有两个成员：`Ok`，它存放一个类型 `T` 的值，而 `Err` 则存放一个类型 `E` 的值。这个定义使得 `Result` 枚举能很方便的表达任何可能成功（返回 `T` 类型的值）也可能失败（返回 `E` 类型的值）的操作。回忆一下示例 9-2 中打开一个文件的场景：当文件被成功打开 `T` 被放入了 `std::fs::File` 类型而当打开文件出现问题时 `E` 被放入了 `std::io::Error` 类型。

当发现代码中有多个只有存放的值的类型有所不同的结构体或枚举定义时，你就应该像之前的函数定义中那样引入泛型类型来减少重复代码。

方法定义中的枚举数据类型

可以像第五章介绍的那样来为其定义中带有泛型的结构体或枚举实现方法。示例 10-9 中展示了示例 10-6 中定义的结构体 `Point<T>`。接着我们在 `Point<T>` 上定义了一个叫做 `x` 的方法来返回字段 `x` 中数据的引用：

文件名: `src/main.rs`

```
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

fn main() {
    let p = Point { x: 5, y: 10 };

    println!("p.x = {}", p.x());
}
```

示例 10-9: 在 `Point<T>` 结构体上实现方法 `x`，它返回 `T` 类型的字段 `x` 的引用

注意必须在 `impl` 后面声明 `T`，这样就可以在 `Point<T>` 上实现的方法中使用它了。在 `impl` 之后声明泛型 `T`，这样 Rust 就知道 `Point` 的尖括号中的类型是泛型而不是具体类型。例如，可以选择为 `Point<f32>` 实例实现方法，而不是为泛型 `Point` 实例。示例 10-10 展示了一个没有在 `impl` 之后（的尖括号）声明泛型的例子，这里使用了一个具体类型，`f32`：

```
# struct Point<T> {
#     x: T,
#     y: T,
# }
#
# impl Point<f32> {
#     fn distance_from_origin(&self) -> f32 {
#         (self.x.powi(2) + self.y.powi(2)).sqrt()
#     }
# }
```

示例 10-10: 构建一个只用于拥有泛型参数 `T` 的结构体的具体类型的 `impl` 块

这段代码意味着 `Point<f32>` 类型会有一个方法 `distance_from_origin`，而其他 `T` 不是 `f32` 类型的 `Point<T>` 实例则没有定义此方法。这个方法计算点实例与另一个点坐标之间的距离，它使用了只能用于浮点型的数学运算符。

结构体定义中的泛型类型参数并不总是与结构体方法签名中使用的泛型是同一类型。示例 10-11 中在示例 10-8 中的结构体 `Point<T, U>` 上定义了一个方法 `mixup`。这个方法获取另一个 `Point` 作为参数，而它可能与调用 `mixup` 的 `self` 是不同的 `Point` 类型。这个方法用 `self` 的 `Point` 类型的 `x` 值（类型 `T`）和参数的 `Point` 类型的 `y` 值（类型 `U`）来创建一个新 `Point` 类型的实例：

文件名: `src/main.rs`

```
struct Point<T, U> {
    x: T,
    y: U,
}
```

```
impl<T, U> Point<T, U> {
    fn mixup<V, W>(self, other: Point<V, W>) -> Point<T, W> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 5, y: 10.4 };
    let p2 = Point { x: "Hello", y: 'c' };

    let p3 = p1.mixup(p2);

    println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```

示例 10-11：方法使用了与结构体定义中不同类型的泛型

在 `main` 函数中，定义了一个有 `i32` 类型的 `x`（其值为 `5`）和 `f64` 的 `y`（其值为 `10.4`）的 `Point`。 `p2` 则是一个有着字符串 `slice` 类型的 `x`（其值为 `"Hello"`）和 `char` 类型的 `y`（其值为 `c`）的 `Point`。在 `p1` 上以 `p2` 作为参数调用 `mixup` 会返回一个 `p3`，它会有一个 `i32` 类型的 `x`，因为 `x` 来自 `p1`，并拥有一个 `char` 类型的 `y`，因为 `y` 来自 `p2`。 `println!` 会打印出 `p3.x = 5, p3.y = c`。

注意泛型参数 `T` 和 `U` 声明于 `impl` 之后，因为他们与结构体定义相对应。而泛型参数 `V` 和 `W` 声明于 `fn` `mixup` 之后，因为他们只是相对于方法本身的。

泛型代码的性能

在阅读本部分的内容的同时你可能会好奇使用泛型类型参数是否会有运行时消耗。好消息是：**Rust** 实现泛型的方式意味着你的代码使用泛型类型参数相比指定具体类型并没有任何速度上的损失。

Rust 通过在编译时进行泛型代码的 单态化（*monomorphization*）来保证效率。单态化是一个将泛型代码转变为实际放入的具体类型的特定代码的过程。

编译器所做的工作正好与示例 10-5 中我们创建泛型函数的步骤相反。编译器寻找所有泛型代码被调用的位置并使用泛型代码针对具体类型生成代码。

让我们看看一个使用标准库中 `Option` 枚举的例子：

```
let integer = Some(5);
let float = Some(5.0);
```

当 **Rust** 编译这些代码的时候，它会进行单态化。编译器会读取传递给 `Option` 的值并发现有两种 `Option<T>`：一个对应 `i32` 另一个对应 `f64`。为此，它会将泛型定义 `Option<T>` 展开为 `Option_i32` 和 `Option_f64`，接着将泛型定义替换为这两个具体的定义。

编译器生成的单态化版本的代码看起来像这样，并包含将泛型 `Option` 替换为编译器创建的具体定义后的用例代码：

文件名: `src/main.rs`

```
enum Option_i32 {
    Some(i32),
    None,
}
```

```
enum Option_f64 {  
    Some(f64),  
    None,  
}  
  
fn main() {  
    let integer = Option_i32::Some(5);  
    let float = Option_f64::Some(5.0);  
}
```

我们可以使用泛型来编写不重复的代码，而 **Rust** 将会为每一个实例编译其特定类型的代码。这意味着在使用泛型时没有运行时开销；当代码运行，它的执行效率就跟好像手写每个具体定义的重复代码一样。这个单态化过程正是 **Rust** 泛型在运行时极其高效的原因。

trait: 定义共享的行为

ch10-02-traits.md

commit 1cbcc277af6931d3091fe46a8f379fefae7202db

trait 允许我们进行另一种抽象：他们让我们可以抽象类型所通用的行为。**trait** 告诉 **Rust** 编译器某个特定类型拥有可能与其他类型共享的功能。在使用泛型类型参数的场景中，可以使用 **trait bounds** 在编译时指定泛型可以是任何实现了某个 **trait** 的类型，并由此在这个场景下拥有我们想要的功能。

注意：**trait** 类似于其他语言中的常被称为 接口（*interfaces*）的功能，虽然有一些不同。

定义 trait

一个类型的行为由其可供调用的方法构成。如果可以对不同类型调用相同的方法的话，这些类型就可以共享相同的行为了。**trait** 定义是一种将方法签名组合起来的方法，目的是定义一个实现某些目的所必需的行为的集合。

例如，这里有多存放了不同类型和属性文本的结构体：结构体 `NewsArticle` 用于存放发生于世界各地的新闻故事，而结构体 `Tweet` 最多只能存放 140 个字符的内容，以及像是否转推或是否是对推友的回复这样的元数据。

我们想要创建一个多媒体聚合库用来显示可能储存在 `NewsArticle` 或 `Tweet` 实例中的数据的总结。每一个结构体都需要的行为是他们是能够被总结的，这样的话就可以调用实例的 `summary` 方法来请求总结。示例 10-12 中展示了一个表现这个概念的 `Summarizable trait` 的定义：

文件名: lib.rs

```
pub trait Summarizable {  
    fn summary(&self) -> String;  
}
```

示例 10-12: `Summarizable trait` 定义，它包含由 `summary` 方法提供的行为

使用 `trait` 关键字来声明一个 **trait**，后面是 **trait** 的名字，在这个例子中是 `Summarizable`。在大括号中声明描述实现这个 **trait** 的类型所需要的方法签名，在这个例子中是 `fn summary(&self) -> String`。在方法签名后跟分号，而不是在大括号中提供其实现。接着每一个实现这个 **trait** 的类型都需要提供其自定义行为的方法体，编译器也会确保任何实现 `Summarizable trait` 的类型都拥有与这个签名的定义完全一致的 `summary` 方法。

trait 体中可以有多方法，一行一个方法签名且都以分号结尾。

为类型实现 trait

现在我们定义了 `Summarizable trait`，接着就可以在多媒体聚合库中需要拥有这个行为的类型上实现它了。示例 10-12 中展示了 `NewsArticle` 结构体上 `Summarizable trait` 的一个实现，它使用标题、作者和创建的位置作为 `summary` 的返回值。对于 `Tweet` 结构体，我们选择将 `summary` 定义为用户名后跟推文的全部文本作为返回值，并假设推文内容已经被限制为 140 字符以内。

文件名: lib.rs

```
# pub trait Summarizable {  
#     fn summary(&self) -> String;  
# }  
#  
pub struct NewsArticle {  
    pub headline: String,
```

```

    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summarizable for NewsArticle {
    fn summary(&self) -> String {
        format!("{}", by {} ({}), self.headline, self.author, self.location)
    }
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summarizable for Tweet {
    fn summary(&self) -> String {
        format!("{}", self.username, self.content)
    }
}

```

示例 10-13: 在 `NewsArticle` 和 `Tweet` 类型上实现 `Summarizable` trait

在类型上实现 `trait` 类似于实现与 `trait` 无关的方法。区别在于 `impl` 关键字之后，我们提供需要实现 `trait` 的名称，接着是 `for` 和需要实现 `trait` 的类型的名称。在 `impl` 块中，使用 `trait` 定义中的方法签名，不过不再后跟分号，而是需要在括号中编写函数体来为特定类型实现 `trait` 方法所拥有的行为。

一旦实现了 `trait`，我们就可以用与 `NewsArticle` 和 `Tweet` 实例的非 `trait` 方法一样的方式调用 `trait` 方法了：

```

let tweet = Tweet {
    username: String::from("horse_ebooks"),
    content: String::from("of course, as you probably already know, people"),
    reply: false,
    retweet: false,
};

println!("1 new tweet: {}", tweet.summary());

```

这会打印出 `1 new tweet: horse_ebooks: of course, as you probably already know, people`。

注意因为示例 10-12 中我们在相同的 `lib.rs` 里定义了 `Summarizable` `trait` 和 `NewsArticle` 与 `Tweet` 类型，所以他们是位于同一作用域的。如果这个 `lib.rs` 是对应 `aggregator` `crate` 的，而别人想要利用我们 `crate` 的功能外加为其 `WeatherForecast` 结构体实现 `Summarizable` `trait`，在实现 `Summarizable` `trait` 之前他们首先就需要将其导入其作用域中，如示例 10-14 所示：

文件名: `lib.rs`

```

extern crate aggregator;

use aggregator::Summarizable;

struct WeatherForecast {
    high_temp: f64,
    low_temp: f64,
    chance_of_precipitation: f64,
}

impl Summarizable for WeatherForecast {

```

```
fn summary(&self) -> String {
    format!("The high will be {}, and the low will be {}. The chance of
    precipitation is {}%.", self.high_temp, self.low_temp,
    self.chance_of_precipitation)
}
}
```

示例 10-14: 在另一个 `crate` 中将 `aggregator crate` 的 `Summarizable trait` 引入作用域

另外这段代码假设 `Summarizable` 是一个公有 `trait`，这是因为示例 10-12 中 `trait` 之前使用了 `pub` 关键字。

`trait` 实现的一个需要注意的限制是：只能在 `trait` 或对应类型位于我们 `crate` 本地的时候为其实现 `trait`。换句话说，不允许对外部类型实现外部 `trait`。例如，不能在 `Vec` 上实现 `Display trait`，因为 `Display` 和 `Vec` 都定义于标准库中。允许在像 `Tweet` 这样作为我们 `aggregator crate` 部分功能的自定义类型上实现标准库中的 `trait Display`。也允许在 `aggregator crate` 中为 `Vec` 实现 `Summarizable`，因为 `Summarizable` 定义于此。这个限制是我们称为孤儿规则（*orphan rule*）的一部分，如果你感兴趣的可以在类型理论中找到它。简单来说，它被称为 *orphan rule* 是因为其父类型不存在。没有这条规则的话，两个 `crate` 可以分别对相同类型实现相同的 `trait`，因而这两个实现会相互冲突：`Rust` 将无从得知应该使用哪一个。因为 `Rust` 强制执行 *orphan rule*，其他人编写的代码不会破坏你代码，反之亦是如此。

默认实现

有时为 `trait` 中的某些或全部方法提供默认的行为，而不是在每个类型的每个实现中都定义自己的行为是很有用的。这样当为某个特定类型实现 `trait` 时，可以选择保留或重载每个方法的默认行为。

示例 10-15 中展示了如何为 `Summarize trait` 的 `summary` 方法指定一个默认的字符串值，而不是像示例 10-12 中那样只是定义方法签名：

文件名: `lib.rs`

```
pub trait Summarizable {
    fn summary(&self) -> String {
        String::from("(Read more...)")
    }
}
```

示例 10-15: `Summarizable trait` 的定义，带有一个 `summary` 方法的默认实现

如果想要对 `NewsArticle` 实例使用这个默认实现，而不是像示例 10-13 中那样定义一个自己的实现，则可以指定一个空的 `impl` 块：

```
impl Summarizable for NewsArticle {}
```

即便选择不再直接为 `NewsArticle` 定义 `summary` 方法了，因为 `summary` 方法有一个默认实现而且 `NewsArticle` 被指定为实现了 `Summarizable trait`，我们仍然可以对 `NewsArticle` 的实例调用 `summary` 方法：

```
let article = NewsArticle {
    headline: String::from("Penguins win the Stanley Cup Championship!"),
    location: String::from("Pittsburgh, PA, USA"),
    author: String::from("Iceburgh"),
    content: String::from("The Pittsburgh Penguins once again are the best
    hockey team in the NHL."),
};

println!("New article available! {}", article.summary());
```


这段代码会打印 `New article available! (Read more...)`。

将 `Summarizable` `trait` 改变为拥有默认 `summary` 实现并不要求对示例 10-13 中 `Tweet` 和示例 10-14 中 `WeatherForecast` 的 `Summarizable` 实现做任何改变：重载一个默认实现的语法与实现没有默认实现的 `trait` 方法时完全一样的。

默认实现允许调用相同 `trait` 中的其他方法，哪怕这些方法没有默认实现。通过这种方法，`trait` 可以实现很多有用的功能而只需实现一小部分特定内容。我们可以选择让 `Summarizable` `trait` 也拥有一个要求实现的 `author_summary` 方法，接着 `summary` 方法则提供默认实现并调用 `author_summary` 方法：

```
pub trait Summarizable {
    fn author_summary(&self) -> String;

    fn summary(&self) -> String {
        format!("(Read more from {}...)", self.author_summary())
    }
}
```

为了使用这个版本的 `Summarizable`，只需在实现 `trait` 时定义 `author_summary` 即可：

```
impl Summarizable for Tweet {
    fn author_summary(&self) -> String {
        format!("{}", self.username)
    }
}
```

一旦定义了 `author_summary`，我们就可以对 `Tweet` 结构体的实例调用 `summary` 了，而 `summary` 的默认实现会调用我们提供的 `author_summary` 定义。

```
let tweet = Tweet {
    username: String::from("horse_ebooks"),
    content: String::from("of course, as you probably already know, people"),
    reply: false,
    retweet: false,
};

println!("1 new tweet: {}", tweet.summary());
```

这会打印出 `1 new tweet: (Read more from @horse_ebooks...)`。

注意在重载过的实现中调用默认实现是不可能的。

trait bounds

现在我们定义了 `trait` 并在类型上实现了这些 `trait`，也可以对泛型类型参数使用 `trait`。我们可以限制泛型不再适用于任何类型，编译器会确保其被限制为那些实现了特定 `trait` 的类型，由此泛型就会拥有我们希望其类型所拥有的功能。这被称为指定泛型的 *trait bounds*。

例如在示例 10-13 中为 `NewsArticle` 和 `Tweet` 类型实现了 `Summarizable` `trait`。我们可以定义一个函数 `notify` 来调用 `summary` 方法，它拥有一个泛型类型 `T` 的参数 `item`。为了能够在 `item` 上调用 `summary` 而不出现错误，我们可以在 `T` 上使用 `trait bounds` 来指定 `item` 必须是实现了 `Summarizable` `trait` 的类型：

```
pub fn notify<T: Summarizable>(item: T) {
    println!("Breaking news! {}", item.summary());
}
```

trait bounds 连同泛型类型参数声明一同出现，位于尖括号中的冒号后面。由于 `T` 上的 **trait bounds**，我们可以传递任何 `NewsArticle` 或 `Tweet` 的实例来调用 `notify` 函数。示例 10-14 中使用我们 `aggregator crate` 的外部代码也可以传递一个 `WeatherForecast` 的实例来调用 `notify` 函数，因为 `WeatherForecast` 同样也实现了 `Summarizable`。使用任何其他类型，比如 `String` 或 `i32`，来调用 `notify` 的代码将不能编译，因为这些类型没有实现 `Summarizable`。

可以通过 `+` 来为泛型指定多个 **trait bounds**。如果我们能够需要在函数中使用 `T` 类型的显示格式的同时也能使用 `summary` 方法，则可以使用 **trait bounds** `T: Summarizable + Display`。这意味着 `T` 可以是任何实现了 `Summarizable` 和 `Display` 的类型。

对于拥有多个泛型类型参数的函数，每一个泛型都可以有其自己的 **trait bounds**。在函数名和参数列表之间的尖括号中指定很多的 **trait bound** 信息将是难以阅读的，所以有另外一个指定 **trait bounds** 的语法，它将其移动到函数签名后的 `where` 从句中。所以相比这样写：

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: T, u: U) -> i32 {
```

我们也可以使用 `where` 从句：

```
fn some_function<T, U>(t: T, u: U) -> i32
  where T: Display + Clone,
        U: Clone + Debug
{
```

这就显得不那么杂乱，同时也使这个函数看起来更像没有很多 **trait bounds** 的函数。这时函数名、参数列表和返回值类型都离得很近。

使用 **trait bounds** 来修复 `largest` 函数

所以任何想要对泛型使用 **trait** 定义的行为的时候，都需要在泛型参数类型上指定 **trait bounds**。现在我们就可以修复示例 10-5 中那个使用泛型类型参数的 `largest` 函数定义了！当我们将其放置不管的时候，它会出现这个错误：

```
error[E0369]: binary operation `>` cannot be applied to type `T`
  |
5 |         if item > largest {
  |                ^^^^
  |
note: an implementation of `std::cmp::PartialOrd` might be missing for `T`
```

在 `largest` 函数体中我们想要使用大于运算符比较两个 `T` 类型的值。这个运算符被定义为标准库中 `trait std::cmp::PartialOrd` 的一个默认方法。所以为了能够使用大于运算符，需要在 `T` 的 **trait bounds** 中指定 `PartialOrd`，这样 `largest` 函数可以用于任何可以比较大小的类型的 `slice`。因为 `PartialOrd` 位于 `prelude` 中所以并不需要手动将其引入作用域。

```
fn largest<T: PartialOrd>(list: &[T]) -> T {
```

但是如果编译代码的话，会出现不同的错误：

```
error[E0508]: cannot move out of type `[T]`, a non-copy array
--> src/main.rs:4:23
  |
4 |     let mut largest = list[0];
  |     ----- ^^^^^^^ cannot move out of here
  |     |
```

```
|             hint: to prevent move, use `ref largest` or `ref mut largest`

error[E0507]: cannot move out of borrowed content
--> src/main.rs:6:9
|
6 |     for &item in list.iter() {
|         ^----
|         ||
|         |hint: to prevent move, use `ref item` or `ref mut item`
|         cannot move out of borrowed content
```

错误的核心是 `cannot move out of type [T], a non-copy array`，对于非泛型版本的 `largest` 函数，我们只尝试了寻找最大的 `i32` 和 `char`。正如第四章讨论过的，像 `i32` 和 `char` 这样的类型是已知大小的并可以储存在栈上，所以他们实现了 `Copy trait`。当我们将 `largest` 函数改成使用泛型后，现在 `list` 参数的类型就有可能是没有实现 `Copy trait` 的，这意味着我们可能不能将 `list[0]` 的值移动到 `largest` 变量中。

如果只想对实现了 `Copy` 的类型调用这些代码，可以在 `T` 的 `trait bounds` 中增加 `Copy`！示例 10-16 中展示了一个可以编译的泛型版本的 `largest` 函数的完整代码，只要传递给 `largest` 的 `slice` 值的类型实现了 `PartialOrd` 和 `Copy` 这两个 `trait`，例如 `i32` 和 `char`：

文件名: `src/main.rs`

```
use std::cmp::PartialOrd;

fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

示例 10-16：一个可以用于任何实现了 `PartialOrd` 和 `Copy trait` 的泛型的 `largest` 函数

如果并不希望限制 `largest` 函数只能用于实现了 `Copy trait` 的类型，我们可以在 `T` 的 `trait bounds` 中指定 `Clone` 而不是 `Copy`，并克隆 `slice` 的每一个值使得 `largest` 函数拥有其所有权。但是使用 `clone` 函数潜在意味着更多的堆分配，而且堆分配在涉及大量数据时可能会相当缓慢。另一种 `largest` 的实现方式是返回 `slice` 中一个 `T` 值的引用。如果我们将函数返回值从 `T` 改为 `&T` 并改变函数体使其能够返回一个引用，我们将不需要任何 `Clone` 或 `Copy` 的 `trait bounds` 而且也不会有任何的堆分配。尝试自己实现这种替代解决方式吧！

使用 `trait bound` 有条件的实现方法

通过使用带有 **trait bound** 的泛型 **impl** 块，可以有条件的只为实现了特定 **trait** 的类型实现方法。例如，示例 10-17 中的类型 `Pair<T>` 总是实现了 `new` 方法，不过只有 `Pair<T>` 内部的 `T` 实现了 `PartialOrd` **trait** 来允许比较和 `Display` **trait** 来启用打印，才会实现 `cmp_display`：

```
use std::fmt::Display;

struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self {
            x,
            y,
        }
    }
}

impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}
```

示例 10-17：根据 **trait bound** 在泛型上有条件的实现方法

也可以对任何实现了特定 **trait** 的类型有条件的实现 **trait**。对任何满足特定 **trait bound** 的类型实现 **trait** 被称为 *blanket implementations*，他们被广泛的用于 **Rust** 标准库中。例如，标准库为任何实现了 `Display` **trait** 的类型实现了 `ToString` **trait**。这个 **impl** 块看起来像这样：

```
impl<T: Display> ToString for T {
    // ...snip...
}
```

因为标准库有了这些 **blanket implementation**，我们可以对任何实现了 `Display` **trait** 的类型调用由 `ToString` 定义的 `to_string` 方法。例如，可以将整型转换为对应的 `String` 值，因为整型实现了 `Display`：

```
let s = 3.to_string();
```

blanket implementation 会出现在 **trait** 文档的“Implementers”部分。

trait 和 **trait bound** 让我们使用泛型类型参数来减少重复，并仍然能够向编译器明确指定泛型类型需要拥有哪些行为。因为我们向编译器提供了 **trait bound** 信息，它就可以检查代码中所用到的具体类型是否提供了正确的行为。在动态类型语言中，如果我们尝试调用一个类型并没有实现的方法，会在运行时出现错误。**Rust** 将这些错误移动到了编译时，甚至在代码能够运行之前就强迫我们修复错误。另外，我们也无需编写运行时检查行为的代码，因为在编译时就已经检查过了，这样相比其他那些不愿放弃泛型灵活性的语言有更好的性能。

这里还有一种泛型，我们一直在使用它甚至都没有察觉它的存在，这就是 生命周期 (*lifetimes*)。不同于其他泛型帮助我们确保类型拥有期望的行为，生命周期则有助于确保引用在我们需要他们的时候一直有效。让我们学习生命周期是如何做到这些的。

生命周期与引用有效性

ch10-03-lifetime-syntax.md

commit aa4be9389d18c31f587ebf75cbbb6af39ff4247d

当在第四章讨论引用时，我们遗漏了一个重要的细节：**Rust** 中的每一个引用都有其 生命周期（*lifetime*），也就是引用保持有效的作用域。大部分时候生命周期是隐含并可以推断的，正如大部分时候类型也是可以推断的一样。类似于当因为有多种可能类型的时候必须注明类型，也会出现引用的生命周期以一些不同方式相关联的情况，所以 **Rust** 需要我们使用泛型生命周期参数来注明他们的关系，这样就能确保运行时实际使用的引用绝对是有效的。

好吧，这有点不太寻常，而且也不同于其他语言中使用的工具。生命周期，从某种意义上说，是 **Rust** 最与众不同的功能。

生命周期是一个很广泛的话题，本章不可能涉及到它全部的内容，所以这里我们会讲到一些通常你可能会遇到的生命周期语法以便你熟悉这个概念。第十九章会包含生命周期所有功能的更高级的内容。

生命周期避免了悬垂引用

生命周期的主要目标是避免悬垂引用，它会导致程序引用了非预期引用的数据。考虑一下示例 10-18 中的程序，它有一个外部作用域和一个内部作用域，外部作用域声明了一个没有初值的变量 `r`，而内部作用域声明了一个初值为 5 的变量 `x`。在内部作用域中，我们尝试将 `r` 的值设置为一个 `x` 的引用。接着在内部作用域结束后，尝试打印出 `r` 的值：

```
{
    let r;

    {
        let x = 5;
        r = &x;
    }

    println!("r: {}", r);
}
```

示例 10-18：尝试使用离开作用域的值引用

未初始化变量不能被使用

接下来的一些例子中声明了没有初始值的变量，以便这些变量存在于外部作用域。这看起来好像和 **Rust** 不允许存在空值相冲突。然而这是可以的，如果我们尝试在给它一个值之前使用这个变量，会出现一个编译时错误。请自行尝试！

当编译这段代码时会得到一个错误：

```
error: `x` does not live long enough
  |
6 |         r = &x;
  |             - borrow occurs here
7 |     }
  |     ^ `x` dropped here while still borrowed
...
10 | }
   | - borrowed value needs to live until here
```

变量 `x` 并没有“存在的足够久”。为什么呢？好吧，`x` 在到达第 7 行的大括号的结束时就离开了作用域，这也是内部作用域的结尾。不过 `r` 在外部作用域也是有效的；作用域越大我们就说它“存在的越久”。如果 **Rust** 允许这段代码工作，`r` 将会引用在 `x` 离开作用域时被释放的内存，这时尝试对 `r` 做任何操作都会不能正常工作。那么 **Rust** 是如何决定这段代码是不被允许的呢？

借用检查器

编译器的这一部分叫做 借用检查器 (*borrow checker*)，它比较作用域来确保所有的借用都是有效的。示例 10-19 展示了与示例 10-18 相同的例子不过带有变量声明周期的注释：

```
{
    let r;           // -----+-- 'a
                    //      |
    {               //      |
        let x = 5;   // -+-----+-- 'b
        r = &x;     //  |   |
    }              // -+   |
                    //      |
    println!("r: {}", r); //  |
                    //      |
                    // -----+
}
```

示例 10-19: `r` 和 `x` 的生命周期注解，分别叫做 `'a` 和 `'b`

我们将 `r` 的生命周期标记为 `'a` 并将 `x` 的生命周期标记为 `'b`。如你所见，内部的 `'b` 块要比外部的生命周期 `'a` 小得多。在编译时，**Rust** 比较这两个生命周期的大小，并发现 `r` 拥有声明周期 `'a`，不过它引用了一个拥有生命周期 `'b` 的对象。程序被拒绝编译，因为生命周期 `'b` 比生命周期 `'a` 要小：被引用的对象比它的引用者存活的时间更短。

让我们看看示例 10-20 中这个并没有产生悬垂引用且可以正确编译的例子：

```
{
    let x = 5;       // -----+-- 'b
                    //      |
    let r = &x;       // -+-----+-- 'a
                    //  |   |
    println!("r: {}", r); //  |   |
                    // -+   |
                    //      |
                    // -----+
}
```

示例 10-20: 一个有效的引用，因为数据比引用有着更长的生命周期

这里 `x` 拥有生命周期 `'b`，比 `'a` 要大。这就意味着 `r` 可以引用 `x`：**Rust** 知道 `r` 中的引用在 `x` 有效的时候也总是有效的。

现在我们已经在具体的例子中展示了引用的生命周期位于何处，并讨论了 **Rust** 如何分析生命周期来保证引用总是有效的，接下来让我们聊聊在函数的上下文中参数和返回值的泛型生命周期。

函数中的泛型生命周期

让我们来编写一个返回两个字符串 `slice` 中较长者的函数。我们希望能够通过传递两个字符串 `slice` 来调用这个函数，并希望返回一个字符串 `slice`。一旦我们实现了 `longest` 函数，示例 10-21 中的代码应该会打印出 `The longest string is abcd`：

文件名: `src/main.rs`

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}
```

示例 10-21: `main` 函数调用 `longest` 函数来寻找两个字符串 `slice` 中较长的一个

注意函数期望获取字符串 `slice`（如第四章所讲到的这是引用）因为我们并不希望 `longest` 函数获取其参数的所有权。我们希望函数能够接受 `String` 的 `slice`（也就是变量 `string1` 的类型）以及字符串字面值（也就是变量 `string2` 包含的值）。

参考之前第四章中的“字符串 `slice` 作为参数”部分中更多关于为什么上面例子中的参数正符合我们期望的讨论。

如果尝试像示例 10-22 中那样实现 `longest` 函数，它并不能编译：

文件名: `src/main.rs`

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

示例 10-22: 一个 `longest` 函数的实现，它返回两个字符串 `slice` 中较长者，现在还不能编译

将会出现如下有关生命周期的错误：

```
error[E0106]: missing lifetime specifier
  |
1 | fn longest(x: &str, y: &str) -> &str {
  |                                   ^ expected lifetime parameter
  |
= help: this function's return type contains a borrowed value, but the
signature does not say whether it is borrowed from `x` or `y`
```

提示文本告诉我们返回值需要一个泛型生命周期参数，因为 `Rust` 并不知道将要返回的引用是指向 `x` 或 `y`。事实上我们也不知道，因为函数体中 `if` 块返回一个 `x` 的引用而 `else` 块返回一个 `y` 的引用。

虽然我们定义了这个函数，但是并不知道传递给函数的具体值，所以也不知道到底是 `if` 还是 `else` 会被执行。我们也不知道传入的引用的具体生命周期，所以也就不能像示例 10-19 和 10-20 那样通过观察作用域来确定返回的引用是否总是有效。借用检查器自身同样也无法确定，因为它不知道 `x` 和 `y` 的生命周期是如何与返回值的生命周期相关联的。接下来我们将增加泛型生命周期参数来定义引用间的关系以便借用检查器可以进行分析。

生命周期注解语法

生命周期注解并不改变任何引用的生命周期的长短。与当函数签名中指定了泛型类型参数后就可以接受任何类型一样，当指定了泛型生命周期后函数也能接受任何生命周期的引用。生命周期注解所做的就是将多个引用的生命周期联系起来。

生命周期注解有着一个不太常见的语法：生命周期参数名称必须以撇号（`'`）开头。生命周期参数的名称通常全是小写，而且类似于泛型类型，其名称通常非常短。`'a` 是大多数人默认使用的名称。生命周期参数注解位于引用的 `&` 之后，并有一个空格来将引用类型与生命周期注解分隔开。

这里有一些例子：我们有一个没有生命周期参数的 `i32` 的引用，一个有叫做 `'a` 的生命周期参数的 `i32` 的引用，和一个生命周期也是 `'a` 的 `i32` 的可变引用：

```
&i32          // a reference
&'a i32       // a reference with an explicit lifetime
&'a mut i32   // a mutable reference with an explicit lifetime
```

单个的生命周期注解本身没有多少意义：生命周期注解告诉 **Rust** 多个引用的泛型生命周期参数如何相互联系。如果函数有一个生命周期 `'a` 的 `i32` 的引用的参数 `first`，还有另一个同样是生命周期 `'a` 的 `i32` 的引用的参数 `second`，这两个生命周期注解有相同的名称意味着 `first` 和 `second` 必须与这相同的泛型生命周期存在得一样久。

函数签名中的生命周期注解

来看看我们编写的 `longest` 函数的上下文中的生命周期。就像泛型类型参数，泛型生命周期参数需要声明在函数名和参数列表间的尖括号中。这里我们想要告诉 **Rust** 关于参数中的引用和返回值之间的限制是他们都必须拥有相同的生命周期，就像示例 10-23 中在每个引用中都加上了 `'a` 那样：

文件名: `src/main.rs`

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

示例 10-23: `longest` 函数定义指定了签名中所有的引用必须有相同的生命周期 `'a`

这段代码能够编译并会产生我们想要使用示例 10-21 中的 `main` 函数得到的结果。

现在函数签名表明对于某些生命周期 `'a`，函数会获取两个参数，他们都是与生命周期 `'a` 存在的一样长的字符串 `slice`。函数会返回一个同样也与生命周期 `'a` 存在的一样长的字符串 `slice`。这就是我们告诉 **Rust** 需要其保证的协议。

我们并不会通过“在函数签名中指定生命周期参数”这种方式来改变任何参数或返回值的生命周期，而是指出任何不遵守这个协议的传入值都将被借用检查器拒绝。这个函数并不知道（或需要知道）`x` 和 `y` 具体会存在多久，而只需要知道有某个可以被 `'a` 替代的作用域将会满足这个签名。

当在函数中使用生命周期注解时，这些注解出现在函数签名中，而不存在于函数体中的任何代码中。这是因为 **Rust** 能够分析函数中代码而不需要任何协助，不过当函数引用或被函数之外的代码引用时，参数或返回值的生命周期可能在每次函数被调用时都不同。这可能会产生惊人的消耗并且对于 **Rust** 来说通常是不可能分析的。在这种情况下，我们需要自己标注生命周期。

当具体的引用被传递给 `longest` 时，被 `'a` 所替代的具体生命周期是 `x` 的作用域与 `y` 的作用域相重叠的那一部分。因为作用域总是嵌套的，所以换一种说法就是泛型生命周期 `'a` 的具体生命周期等同于 `x` 和 `y` 的生命周期中较小的那一个。因为我们用相同的生命周期参数标注了返回的引用值，所以返回的引用值就能保证在 `x` 和 `y` 中较短的那个生命周期结束之前保持有效。

让我们看看如何通过传递拥有不同具体生命周期的引用来限制 `longest` 函数的使用。示例 10-24 是一个应该在任何编程语言中都很直观的例子：`string1` 直到外部作用域结束都是有效的，`string2` 则在内部作用域中是有效的，而 `result` 则引用了一些直到内部作用域结束都是有效的值。借用检查器认可这些代码；它能够编译和运行，并打印出 `The longest string is long string is long`：

文件名: `src/main.rs`

```
# fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
#     if x.len() > y.len() {
#         x
#     } else {
#         y
#     }
# }
#
fn main() {
    let string1 = String::from("long string is long");

    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_str(), string2.as_str());
        println!("The longest string is {}", result);
    }
}
```

示例 10-24：通过拥有不同的具体生命周期的 `String` 值调用 `longest` 函数

接下来，让我们尝试一个 `result` 的引用的生命周期肯定比两个参数的要短的例子。将 `result` 变量的声明从内部作用域中移动出来，但是将 `result` 和 `string2` 变量的赋值语句一同放在内部作用域里。接下来，我们将使用 `result` 的 `println!` 移动到内部作用域之外，就在其结束之后。注意示例 10-25 中的代码不能编译：

文件名: `src/main.rs`

```
fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is {}", result);
}
```

示例 10-25：在 `string2` 离开作用域之后使用 `result` 的尝试不能编译

如果尝试编译会出现如下错误：

```
error: `string2` does not live long enough
  |
6 |         result = longest(string1.as_str(), string2.as_str());
  |                                     ----- borrow occurs here
7 |     }
  |     ^ `string2` dropped here while still borrowed
8 |     println!("The longest string is {}", result);
9 | }
  | - borrowed value needs to live until here
```

错误表明为了保证 `println!` 中的 `result` 是有效的，`string2` 需要直到外部作用域结束都是有效的。Rust 知道这些是因为（`longest`）函数的参数和返回值都使用了相同的生命周期参数 `'a`。

以我们的理解 `string1` 更长，因此 `result` 会包含指向 `string1` 的引用。因为 `string1` 尚未离开作用域，对于 `println!` 来说 `string1` 的引用仍然是有效的。然而，我们通过生命周期参数告诉 **Rust** 的是 `longest` 函数返回的引用的生命周期应该与传入参数的生命周期中较短那个保持一致。因此，借用检查器不允许示例 10-25 中的代码，因为它可能会存在无效的引用。

请尝试更多采用不同的值和不同生命周期的引用作为 `longest` 函数的参数和返回值的实验。并在开始编译前猜想你的实验能否通过借用检查器，接着编译一下看看你的理解是否正确！

深入理解生命周期

指定生命周期参数的正确方式依赖函数具体的功能。例如，如果将 `longest` 函数的实现修改为总是返回第一个参数而不是最长的字符串 `slice`，就不需要为参数 `y` 指定一个生命周期。如下代码将能够编译：

文件名: `src/main.rs`

```
fn longest<'a>(x: &'a str, y: &str) -> &'a str {
    x
}
```

在这个例子中，我们为参数 `x` 和返回值指定了生命周期参数 `'a`，不过没有为参数 `y` 指定，因为 `y` 的生命周期与参数 `x` 和返回值的生命周期没有任何关系。

当从函数返回一个引用，返回值的生命周期参数需要与一个参数的生命周期参数相匹配。如果返回的引用没有指向任何一个参数，那么唯一的可能就是它指向一个函数内部创建的值，它将会是一个悬垂引用，因为它将会在函数结束时离开作用域。尝试考虑这个并不能编译的 `longest` 函数实现：

文件名: `src/main.rs`

```
fn longest<'a>(x: &str, y: &str) -> &'a str {
    let result = String::from("really long string");
    result.as_str()
}
```

即便我们为返回值指定了生命周期参数 `'a`，这个实现却编译失败了，因为返回值的生命周期与参数完全没有关联。这里是会出现的错误信息：

```
error: `result` does not live long enough
  |
3 |     result.as_str()
  |     ^^^^^ does not live long enough
4 | }
  | - borrowed value only lives until here
  |
note: borrowed value must be valid for the lifetime 'a as defined on the block
at 1:44...
  |
1 | fn longest<'a>(x: &str, y: &str) -> &'a str {
  |                                     ^
```

出现的问题是 `result` 在 `longest` 函数的结尾将离开作用域并被清理，而我们尝试从函数返回一个 `result` 的引用。无法指定生命周期参数来改变悬垂引用，而且 **Rust** 也不允许我们创建一个悬垂引用。在这种情况下，最好的解决方案是返回一个有所有权的数据类型而不是一个引用，这样函数调用者就需要负责清理这个值了。

从结果上看，生命周期语法是关于如何联系函数不同参数和返回值的生命周期的。一旦他们形成了某种联系，**Rust** 就有了足够的信息来允许内存安全的操作并阻止会产生悬垂指针亦或是违反内存安全的行为。

结构体定义中的生命周期注解

目前为止，我们只定义过有所有权类型的结构体。也可以定义存放引用的结构体，不过需要为结构体定义中的每一个引用添加生命周期注解。示例 10-26 中有一个存放了一个字符串 `slice` 的结构体 `ImportantExcerpt`：

文件名: `src/main.rs`

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.')
        .next()
        .expect("Could not find a '.");
    let i = ImportantExcerpt { part: first_sentence };
}
```

示例 10-26：一个存放引用的结构体，所以其定义需要生命周期注解

这个结构体有一个字段，`part`，它存放了一个字符串 `slice`，这是一个引用。类似于泛型参数类型，必须在结构体名称后面的尖括号中声明泛型生命周期参数，以便在结构体定义中使用生命周期参数。

这里的 `main` 函数创建了一个 `ImportantExcerpt` 的实例，它存放了变量 `novel` 所拥有的 `String` 的第一个句子的引用。

生命周期省略

在这一部分，我们知道了每一个引用都有一个生命周期，而且需要为使用了引用的函数或结构体指定生命周期。然而，第四章的“字符串 `slice`”部分有一个函数，我们在示例 10-27 中再次展示出来，它没有生命周期注解却能成功编译：

文件名: `src/lib.rs`

```
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

示例 10-27：第四章定义了一个没有使用生命周期注解的函数，即便其参数和返回值都是引用

这个函数没有生命周期注解却能编译是由于一些历史原因：在早期 1.0 之前版本的 `Rust` 中，这的确是不能编译的。每一个引用都必须有明确的生命周期。那时的函数签名将会写成这样：

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

在编写了很多 Rust 代码后，Rust 团队发现在特定情况下 Rust 程序员们总是重复地编写一模一样的生命周期注解。这些场景是可预测的并且遵循几个明确的模式。接着 Rust 团队就把这些模式编码进了 Rust 编译器中，如此借用检查器在这些情况下就能推断出生命周期而不再强制程序员显式的增加注解。

这里我们提到一些 Rust 的历史是因为更多的明确的模式被合并和添加到编译器中是完全可能的。未来只需要更少的生命周期注解。

被编码进 Rust 引用分析的模式被称为 生命周期省略规则 (*lifetime elision rules*)。这并不是需要程序员遵守的规则；这些规则是一系列特定的场景，此时编译器会考虑，如果代码符合这些场景，就无需明确指定生命周期。

省略规则并不提供完整的推断：如果 Rust 在明确遵守这些规则的前提下变量的生命周期仍然是模棱两可的话，它不会猜测剩余引用的生命周期应该是什么。在这种情况下，编译器会给出一个错误，这可以通过增加对应引用之间相联系的生命周期注解来解决。

首先，介绍一些定义：函数或方法的参数的生命周期被称为 输入生命周期 (*input lifetimes*)，而返回值的生命周期被称为 输出生命周期 (*output lifetimes*)。

现在介绍编译器用于判断引用何时不需要明确生命周期注解的规则。第一条规则适用于输入生命周期，后两条规则适用于输出生命周期。如果编译器检查完这三条规则后仍然存在没有计算出生命周期的引用，编译器将会停止并生成错误。

1. 每一个是引用的参数都有它自己的生命周期参数。换句话说就是，有一个引用参数的函数有一个生命周期参数： `fn foo<'a>(x: &'a i32)`，有两个引用参数的函数有两个不同的生命周期参数， `fn foo<'a, 'b>(x: &'a i32, y: &'b i32)`，依此类推。
2. 如果只有一个输入生命周期参数，那么它被赋给所有输出生命周期参数： `fn foo<'a>(x: &'a i32) -> &'a i32`。
3. 如果方法有多个输入生命周期参数，不过其中之一因为方法的缘故为 `&self` 或 `&mut self`，那么 `self` 的生命周期被赋给所有输出生命周期参数。这使得方法写起来更简洁。

假设我们自己就是编译器并来计算示例 10-25 `first_word` 函数的签名中的引用的生命周期。开始时签名中的引用并没有关联任何生命周期：

```
fn first_word(s: &str) -> &str {
```

接着我们（作为编译器）应用第一条规则，也就是每个引用参数都有其自己的生命周期。我们像往常一样称之为 `'a`，所以现在签名看起来像这样：

```
fn first_word<'a>(s: &'a str) -> &str {
```

对于第二条规则，因为这里正好只有一个输入生命周期参数所以是适用的。第二条规则表明输入参数的生命周期将被赋予输出生命周期参数，所以现在签名看起来像这样：

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

现在这个函数签名中的所有引用都有了生命周期，而编译器可以继续它的分析而无须程序员标记这个函数签名中的生命周期。

让我们再看看另一个例子，这次我们从示例 10-22 中没有生命周期参数的 `longest` 函数开始：

```
fn longest(x: &str, y: &str) -> &str {
```

再次假设我们自己就是编译器并应用第一条规则：每个引用参数都有其自己的生命周期。这次有两个参数，所以就有两个生命周期：

```
fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {
```

再来应用第二条规则，它并不适用因为存在多于一个输入生命周期。再来看第三条规则，它同样也不适用因为没有 `self` 参数。然后我们就没有更多规则了，不过还没有计算出返回值的类型的生命周期。这就是为什么在编译示例 10-22 的代码时会出现错误的原因：编译器使用所有已知的生命周期省略规则，不过仍然不能计算出签名中所有引用的生命周期。

因为第三条规则真正能够适用的就只有方法签名，现在就让我们看看那种情况中的生命周期，并看看为什么这条规则意味着我们经常不需要在方法签名中标注生命周期。

方法定义中的生命周期注解

当为带有生命周期的结构体实现方法时，其语法依然类似示例 10-11 中展示的泛型类型参数的语法：声明和使用生命周期参数的位置依赖于生命周期参数是否同结构体字段或方法参数和返回值相关。

（实现方法时）结构体字段的生命周期必须总是在 `impl` 关键字之后声明并在结构体名称之后被使用，因为这些生命周期是结构体类型的一部分。

`impl` 块里的方法签名中，引用可能与结构体字段中的引用相关联，也可能是独立的。另外，生命周期省略规则也经常让我们无需在方法签名中使用生命周期注解。让我们看看一些使用示例 10-26 中定义的结构体

`ImportantExcerpt` 的例子。

首先，这里有一个方法 `level`。其唯一的参数是 `self` 的引用，而且返回值只是一个 `i32`，并不引用任何值：

```
# struct ImportantExcerpt<'a> {
#     part: &'a str,
# }
#
impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 {
        3
    }
}
```

`impl` 之后和类型名称之后的生命周期参数是必要的，不过因为第一条生命周期规则我们并不必须标注 `self` 引用的生命周期。

这里是一个适用于第三条生命周期省略规则的例子：

```
# struct ImportantExcerpt<'a> {
#     part: &'a str,
# }
#
impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part(&self, announcement: &str) -> &str {
        println!("Attention please: {}", announcement);
        self.part
    }
}
```

这里有两个输入生命周期，所以 **Rust** 应用第一条生命周期省略规则并给予 `&self` 和 `announcement` 他们各自的生命周期。接着，因为其中一个参数是 `&self`，返回值类型被赋予了 `&self` 的生命周期，这样所有的生命周期都被计算出来了。

静态生命周期

这里有一种特殊的生命周期值得讨论：`'static`。`'static` 生命周期存活于整个程序期间。所有的字符串面值都拥有 `'static` 生命周期，我们也可以选择像下面这样标注出来：

```
let s: &'static str = "I have a static lifetime.";
```

这个字符串的文本被直接储存在程序的二进制文件中而这个文件总是可用的。因此所有的字符串面值都是 `'static` 的。

你可能在错误信息的帮助文本中见过使用 `'static` 生命周期的建议，不过将引用指定为 `'static` 之前，思考一下这个引用是否真的在整个程序的生命周期里都有效（或者哪怕你希望它一直有效，如果可能的话）。大部分情况，代码中的问题是尝试创建一个悬垂引用或者可用的生命周期不匹配，请解决这些问题而不是指定一个 `'static` 的生命周期。

结合泛型类型参数、**trait bounds** 和生命周期

让我们简单的看一下在同一函数中指定泛型类型参数、**trait bounds** 和生命周期的语法！

```
use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(x: &'a str, y: &'a str, ann: T) -> &'a str
    where T: Display
{
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

这个是示例 10-23 中那个返回两个字符串 `slice` 中较长者的 `longest` 函数，不过带有一个额外的参数 `ann`。`ann` 的类型是泛型 `T`，它可以被放入任何实现了 `where` 从句中指定的 `Display trait` 的类型。这个额外的参数会在函数比较字符串 `slice` 的长度之前被打印出来，这也就是为什么 `Display trait bound` 是必须的。因为生命周期也是泛型，所以生命周期参数 `'a` 和泛型类型参数 `T` 都位于函数名后的同一尖括号列表中。

总结

这一章介绍了很多的内容！现在你知道了泛型类型参数、**trait** 和 **trait bounds** 以及泛型生命周期类型，你已经准备好编写既不重复又能适用于多种场景的代码了。泛型类型参数意味着代码可以适用于不同的类型。**trait** 和 **trait bounds** 保证了即使类型是泛型的，这些类型也会拥有所需要的行为。由生命周期注解所指定的引用生命周期之间的关系保证了这些灵活多变的代码不会出现悬垂引用。而所有的一切发生在编译时所以不会影响运行时效率！

你可能不会相信，这个领域还有更多需要学习的内容：第十七章会讨论 **trait** 对象，这是另一种使用 **trait** 的方式。第十九章会涉及到生命周期注解更复杂的场景。第二十章讲解一些高级的类型系统功能。不过接下来，让我们聊聊如何在 **Rust** 中编写测试，来确保代码的所有功能能像我们希望的那样工作！

测试

[ch11-00-testing.md](#)

commit 1047433147b27d19e5acc068a0ebca5782d64f99

Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.

Edsger W. Dijkstra, "The Humble Programmer" (1972)

软件测试是证明 bug 存在的有效方法，而证明它们不存在时则显得令人绝望的不足。

Edsger W. Dijkstra, 【谦卑的程序员】（1972）

程序的正确性意味着代码如我们期望的那样运行。**Rust** 是一个非常注重正确性的编程语言，不过正确性是一个难以证明的复杂主题。**Rust** 的类型系统在此问题上下了很大的功夫，不过它不可能捕获所有种类的错误。为此，**Rust** 也在语言本身包含了编写软件测试的支持。

例如，我们可以编写一个叫做 `add_two` 的将传递给它的值加二的函数。它的签名有一个整型参数并返回一个整型值。当实现和编译这个函数时，**Rust** 会进行所有目前我们已经见过的类型检查和借用检查，例如，这些检查会确保我们不会传递 `String` 或无效的引用给这个函数。**Rust** 所不能检查的是这个函数是否会准确的完成我们期望的工作：返回参数加二后的值，而不是比如说参数加 10 或减 50 的值！这也就是测试出场的地方。

我们可以编写测试断言，比如说，当传递 3 给 `add_two` 函数时，应该得到 5。当对代码进行修改时可以运行测试来确保任何现存的正确行为没有被改变。

测试是一项复杂的技能，而且我们也不能期望在一本书的一个章节中就涉及到编写好的测试的所有内容，所以这里仅仅讨论 **Rust** 测试功能的机制。我们会讲到编写测试时会用到的注解和宏，**Rust** 提供用来运行测试的默认行为和选项，以及如何将测试组织成单元测试和集成测试。

编写测试

ch11-01-writing-tests.md

commit db08b34db5f1c78b4866b391c802344ec94ecc38

测试用来验证非测试的代码是否按照期望的方式运行的 **Rust** 函数。测试函数体通常包括一些设置，运行需要测试的代码，接着断言其结果是我们所期望的。让我们看看 **Rust** 提供的专门用来编写测试的功能：`test` 属性、一些宏和 `should_panic` 属性。

测试函数剖析

作为最简单例子，**Rust** 中的测试就是一个带有 `test` 属性注解的函数。属性（**attribute**）是关于 **Rust** 代码片段的元数据：第五章中结构体中用到的 `derive` 属性就是一个例子。为了将一个函数变成测试函数，需要在 `fn` 行之前加上 `#[test]`。当使用 `cargo test` 命令运行测试函数时，**Rust** 会构建一个测试执行者二进制文件用来运行标记了 `test` 属性的函数并报告每一个测试是通过还是失败。

第七章当使用 **Cargo** 新建一个库项目时，它会自动为我们生成一个测试模块和一个测试函数。这有助于我们开始编写测试，因为这样每次开始新项目时不必去查找测试函数的具体结构和语法了。当然也可以额外增加任意多的测试函数以及测试模块！

我们将先通过对自动生成的测试模板做一些试验来探索测试如何工作方面的一些内容，而不实际测试任何代码。接着会写一些真实的测试来调用我们编写的代码并断言他们的行为是否正确。

让我们创建一个新的库项目 `adder`：

```
$ cargo new adder
   Created library `adder` project
$ cd adder
```

`adder` 库中 `src/lib.rs` 的内容应该看起来像这样：

文件名: `src/lib.rs`

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
    }
}
```

示例 11-1：由 `cargo new` 自动生成的测试模块和函数

现在让我们暂时忽略 `tests` 模块和 `#[cfg(test)]` 注解并只关注函数来了解其如何工作。注意 `fn` 行之前的 `#[test]`：这个属性表明这是一个测试函数，这样测试执行者就知道将其作为测试处理。也可以在 `tests` 模块中拥有非测试的函数来帮助我们建立通用场景或进行常见操作，所以需要使用 `#[test]` 属性标明哪些函数是测试。

这个函数目前没有任何内容，这意味着没有代码会使测试失败；一个空的测试是可以通过的！让我们运行一下看看它是否通过了。

`cargo test` 命令会运行项目中所有的测试，如示例 11-2 所示：

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
```

```

    Finished dev [unoptimized + debuginfo] target(s) in 0.22 secs
    Running target/debug/deps/adder-ce99bcc2479f4607

running 1 test
test tests::it_works ... ok

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

    Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured

```

示例 11-2：运行自动生成测试的输出

Cargo 编译并运行了测试。在 `Compiling`、`Finished` 和 `Running` 这几行之后，可以看到 `running 1 test` 这一行。下一行显示了生成的测试函数的名称，它是 `it_works`，以及测试的运行结果，`ok`。接着可以看到全体测试运行结果的总结：`test result: ok.` 意味着所有测试都通过了。`1 passed; 0 failed` 表示通过或失败的测试数量。

这里并没有任何被标记为忽略的测试，所以总结表明 `0 ignored`。在下一部分关于运行测试的不同方式中会讨论忽略测试。`0 measured` 统计是针对测试性能的性能测试的。性能测试（**benchmark tests**）在编写本书时，仍只能用于 **Rust** 开发版（**nightly Rust**）。请查看附录 D 来了解更多 **Rust** 开发版的信息。

测试输出中以 `Doc-tests adder` 开头的这一部分是所有文档测试的结果。现在并没有任何文档测试，不过 **Rust** 会编译任何出现在 **API** 文档中的代码示例。这个功能帮助我们使文档和代码保持同步！在第十四章的“文档注释”部分会讲到如何编写文档测试。现在我们将忽略 `Doc-tests` 部分的输出。

让我们改变测试的名称并看看这如何改变测试的输出。给 `it_works` 函数起个不同的名字，比如 `exploration`，像这样：

Filename: `src/lib.rs`

```

#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
    }
}

```

并再次运行 `cargo test`。现在输出中将出现 `exploration` 而不是 `it_works`：

```

running 1 test
test tests::exploration ... ok

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

```

让我们增加另一个测试，不过这一次是一个会失败的测试！当测试函数中出现 **panic** 时测试就失败了。每一个测试都在一个新线程中运行，当主线程发现测试线程异常了，就将对应测试标记为失败。第九章讲到了最简单的造成 **panic** 的方法：调用 `panic!` 宏！写入新函数后 `src/lib.rs` 现在看起来如示例 11-3 所示：

文件名: `src/lib.rs`

```

#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
    }
}

```

```
#[test]
fn another() {
    panic!("Make this test fail");
}
}
```

示例 11-3：增加第二个测试：他会失败因为调用了 `panic!` 宏

再次 `cargo test` 运行测试。输出应该看起来像示例 11-4，它表明 `exploration` 测试通过了而 `another` 失败了：

```
running 2 tests
test tests::exploration ... ok
test tests::another ... FAILED

failures:

---- tests::another stdout ----
thread 'tests::another' panicked at 'Make this test fail', src/lib.rs:9
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
    tests::another

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured

error: test failed
```

示例 11-4：一个测试通过和一个测试失败的测试结果

`test tests::another` 这一行是 `FAILED` 而不是 `ok` 了。在单独测试结果和总结之间多了两个新的部分：第一个部分显示了测试失败的详细原因。在这个例子中，`another` 因为 `panicked at 'Make this test fail'` 而失败，这位于 `src/lib.rs` 的第 9 行。下一部分仅仅列出了所有失败的测试，这在有很多测试和很多失败测试的详细输出时很有帮助。可以使用失败测试的名称来只运行这个测试，这样比较方便调试；下一部分会讲到更多运行测试的方法。

最后是总结行：总体上讲，一个测试结果是 `FAILED` 的。有一个测试通过和一个测试失败。

现在我们见过不同场景中测试结果是什么样子的了，再来看看除 `panic!` 之外的一些在测试中有帮助的宏吧。

使用 `assert!` 宏来检查结果

`assert!` 宏由标准库提供，在希望确保测试中一些条件为 `true` 时非常有用。需要向 `assert!` 宏提供一个计算为布尔值的参数。如果值是 `true`，`assert!` 什么也不做同时测试会通过。如果值为 `false`，`assert!` 调用 `panic!` 宏，这会导致测试失败。这是一个帮助我们检查代码是否以期望的方式运行的宏。

回忆一下第五章中，示例 5-9 中有一个 `Rectangle` 结构体和一个 `can_hold` 方法，在示例 11-5 中再次使用他们。将他们放进 `src/lib.rs` 而不是 `src/main.rs` 并使用 `assert!` 宏编写一些测试。

文件名: `src/lib.rs`

```
#[derive(Debug)]
pub struct Rectangle {
    length: u32,
    width: u32,
}

impl Rectangle {
    pub fn can_hold(&self, other: &Rectangle) -> bool {
        self.length > other.length && self.width > other.width
    }
}
```

```
}  
}
```

示例 11-5: 第五章中 `Rectangle` 结构体和其 `can_hold` 方法

`can_hold` 方法返回一个布尔值，这意味着它完美符合 `assert!` 宏的使用场景。在示例 11-6 中，让我们编写一个 `can_hold` 方法的测试来作为练习，这里创建一个长为 8 宽为 7 的 `Rectangle` 实例，并假设它可以放得下另一个长为 5 宽为 1 的 `Rectangle` 实例：

文件名: `src/lib.rs`

```
#[cfg(test)]  
mod tests {  
    use super::*;  
  
    #[test]  
    fn larger_can_hold_smaller() {  
        let larger = Rectangle { length: 8, width: 7 };  
        let smaller = Rectangle { length: 5, width: 1 };  
  
        assert!(larger.can_hold(&smaller));  
    }  
}
```

示例 11-6: 一个 `can_hold` 的测试，检查一个较大的矩形确实能放得下一个较小的矩形

注意在 `tests` 模块中新增加了一行: `use super::*;`。 `tests` 是一个普通的模块，它遵循第七章介绍的通常的可见性规则。因为这是一个内部模块，需要将外部模块中被测试的代码引入到内部模块的作用域中。这里选择使用全局导入使得外部模块定义的所有内容在 `tests` 模块中都是可用的。

我们将测试命名为 `larger_can_hold_smaller`，并创建所需的两个 `Rectangle` 实例。接着调用 `assert!` 宏并传递 `larger.can_hold(&smaller)` 调用的结果作为参数。这个表达式预期会返回 `true`，所以测试应该通过。让我们拭目以待！

```
running 1 test  
test tests::larger_can_hold_smaller ... ok  
  
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

它确实通过了！再来增加另一个测试，这一回断言一个更小的矩形不能放下一个更大的矩形：

文件名: `src/lib.rs`

```
#[cfg(test)]  
mod tests {  
    use super::*;  
  
    #[test]  
    fn larger_can_hold_smaller() {  
        let larger = Rectangle { length: 8, width: 7 };  
        let smaller = Rectangle { length: 5, width: 1 };  
  
        assert!(larger.can_hold(&smaller));  
    }  
  
    #[test]  
    fn smaller_cannot_hold_larger() {  
        let larger = Rectangle { length: 8, width: 7 };  
        let smaller = Rectangle { length: 5, width: 1 };  

```

```

        assert!(!smaller.can_hold(&larger));
    }
}

```

因为这里 `can_hold` 函数的正确结果是 `false`，我们需要将这个结果取反后传递给 `assert!` 宏。这样的话，测试就会通过而 `can_hold` 将返回 `false`：

```

running 2 tests
test tests::smaller_cannot_hold_larger ... ok
test tests::larger_can_hold_smaller ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured

```

两个通过的测试！现在让我们看看如果引入一个 **bug** 的话测试结果会发生什么。将 `can_hold` 方法中比较长度时本应使用大于号的地方改成小于号：

```

#[derive(Debug)]
pub struct Rectangle {
    length: u32,
    width: u32,
}

impl Rectangle {
    pub fn can_hold(&self, other: &Rectangle) -> bool {
        self.length < other.length && self.width > other.width
    }
}

```

现在运行测试会产生：

```

running 2 tests
test tests::smaller_cannot_hold_larger ... ok
test tests::larger_can_hold_smaller ... FAILED

failures:

---- tests::larger_can_hold_smaller stdout ----
thread 'tests::larger_can_hold_smaller' panicked at 'assertion failed:
larger.can_hold(&smaller)', src/lib.rs:22
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
    tests::larger_can_hold_smaller

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured

```

我们的测试捕获了 **bug**！因为 `larger.length` 是 8 而 `smaller.length` 是 5，`can_hold` 中的长度比较现在因为 8 不小于 5 而返回 `false`。

使用 `assert_eq!` 和 `assert_ne!` 宏来测试相等

测试功能的一个常用方法是将需要测试代码的值与期望值做比较，并检查是否相等。可以通过向 `assert!` 宏传递一个使用 `==` 运算符的表达式来做到。不过这个操作实在是太常见了，以至于标注库提供了一对宏来方便处理这些操作：`assert_eq!` 和 `assert_ne!`。这两个宏分别比较两个值是相等还是不相等。当断言失败时他们也会打印出这两个值具体是什么，以便于观察测试为什么失败，而 `assert!` 只会打印出它从 `==` 表达式中得到了 `false` 值，而不是导致 `false` 的两个值。

示例 11-7 中，让我们编写一个对其参数加二并返回结果的函数 `add_two`。接着使用 `assert_eq!` 宏测试这个函数：

文件名: `src/lib.rs`

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_adds_two() {
        assert_eq!(4, add_two(2));
    }
}
```

示例 11-7：使用 `assert_eq!` 宏测试 `add_two`

测试通过了！

```
running 1 test
test tests::it_adds_two ... ok

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

传递给 `assert_eq!` 宏的第一个参数，4，等于调用 `add_two(2)` 的结果。我们将会看到这个测试的那一行说 `test tests::it_adds_two ... ok`，`ok` 表明测试通过了！

在代码中引入一个 `bug` 来看看使用 `assert_eq!` 的测试失败是什么样的。修改 `add_two` 函数的实现使其加 3：

```
pub fn add_two(a: i32) -> i32 {
    a + 3
}
```

再次运行测试：

```
running 1 test
test tests::it_adds_two ... FAILED

failures:

---- tests::it_adds_two stdout ----
thread 'tests::it_adds_two' panicked at 'assertion failed: `(left == right)` (left: `4`, right: `5`)', src/lib.rs:11
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
    tests::it_adds_two

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured
```

测试捕获到了 `bug`！`it_adds_two` 测试失败并显示信息 `assertion failed: `(left == right)` (left: `4`, right: `5`)`。这个信息有助于我们开始调试：它说 `assert_eq!` 的 `left` 参数是 4，而 `right` 参数，也就是 `add_two(2)` 的结果，是 5。

注意在一些语言和测试框架中，断言两个值相等的函数的参数叫做 `expected` 和 `actual`，而且指定参数的顺序是需要注意的。然而在 **Rust** 中，他们则叫做 `left` 和 `right`，同时指定期望的值和被测试代码产生的值的顺序并不重要。这个测试中的断言也可以写成 `assert_eq!(add_two(2), 4)`，这时错误信息会变成 `assertion failed: `(left == right)` (left: `5`, right: `4`)`。

`assert_ne!` 宏在传递给它的两个值不相等时通过而在相等时失败。这个宏在代码按照我们期望运行时不确定值会是什么，不过知道他们绝对不会是什么的时候最有用处。例如，如果一个函数确定会以某种方式改变其输出，不过这种方式由运行测试是星期几来决定，这时最好的断言可能就是函数的输出不等于其输入。

`assert_eq!` 和 `assert_ne!` 宏在底层分别使用了 `==` 和 `!=`。当断言失败时，这些宏会使用调试格式打印出其参数，这意味着被比较的值必需实现了 `PartialEq` 和 `Debug` `trait`。所有的基本类型和大部分标准库类型都实现了这些 `trait`。对于自定义的结构体和枚举，需要实现 `PartialEq` 才能断言他们的值是否相等。需要实现 `Debug` 才能断言失败时打印他们的值。因为这两个 `trait` 都是可推导 `trait`，如第五章所提到的，通常可以直接在结构体或枚举上添加 `#[derive(PartialEq, Debug)]` 注解。附录 C 中有更多关于这些和其他可推导 `trait` 的详细信息。

自定义错误信息

也可以向 `assert!`、`assert_eq!` 和 `assert_ne!` 宏传递一个可选的参数来增加用于打印的自定义错误信息。任何在 `assert!` 必需的一个参数和 `assert_eq!` 和 `assert_ne!` 必需的两个参数之后指定的参数都会传递给第八章讲到的 `format!` 宏，所以可以传递一个包含 `{}` 占位符的格式字符串和放入占位符的值。自定义信息有助于记录断言的意义，这样到测试失败时，就能更好的理解代码出了什么问题。

例如，比如说有一个根据人名进行问候的函数，而我们希望测试将传递给函数的人名显示在输出中：

Filename: src/lib.rs

```
pub fn greeting(name: &str) -> String {
    format!("Hello {}", name)
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn greeting_contains_name() {
        let result = greeting("Carol");
        assert!(result.contains("Carol"));
    }
}
```

这个程序的需求还没有被确定，而我们非常确定问候开始的 `Hello` 文本不会改变。我们决定并不想在人名改变时不得不更新测试，所以相比检查 `greeting` 函数返回的确切的值，我们将仅仅断言输出的文本中包含输入参数。

让我们通过将 `greeting` 改为不包含 `name` 来在代码中引入一个 `bug` 来测试失败时是怎样的，

```
pub fn greeting(name: &str) -> String {
    String::from("Hello!")
}
```

运行测试会产生：

```
running 1 test
test tests::greeting_contains_name ... FAILED

failures:
```



```

---- tests::greeting_contains_name stdout ----
thread 'tests::greeting_contains_name' panicked at 'assertion failed:
result.contains("Carol")', src/lib.rs:12
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
  tests::greeting_contains_name

```

这仅仅告诉了我们断言失败了和失败的行号。一个更有用的错误信息应该打印出从 `greeting` 函数得到的值。让我们改变测试函数来使用一个由包含占位符的格式字符串和从 `greeting` 函数取得的值组成的自定义错误信息：

```

#[test]
fn greeting_contains_name() {
    let result = greeting("Carol");
    assert!(
        result.contains("Carol"),
        "Greeting did not contain name, value was `{}`", result
    );
}

```

现在如果再次运行测试，将会看到更有价值的错误信息：

```

---- tests::greeting_contains_name stdout ----
thread 'tests::greeting_contains_name' panicked at 'Greeting did not contain
name, value was `Hello`', src/lib.rs:12
note: Run with `RUST_BACKTRACE=1` for a backtrace.

```

可以在测试输出中看到所取得的确切的值，这会帮助我们理解发生了什么而不是期望发生什么。

使用 `should_panic` 检查 `panic`

除了检查代码是否返回期望的正确的值之外，检查代码是否按照期望处理错误情况也是很重要的。例如，考虑第九章示例 9-8 创建的 `Guess` 类型。其他使用 `Guess` 的代码依赖于 `Guess` 实例只会包含 1 到 100 的值的保证。可以编写一个测试来确保创建一个超出范围的值的 `Guess` 实例会 `panic`。

可以通过对函数增加另一个属性 `should_panic` 来实现这些。这个属性在函数中的代码 `panic` 时会通过，而在其中的代码没有 `panic` 时失败。

示例 11-8 展示了如何编写一个测试来检查 `Guess::new` 按照我们的期望出现的错误情况：

文件名: `src/lib.rs`

```

struct Guess {
    value: u32,
}

impl Guess {
    pub fn new(value: u32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess {
            value
        }
    }
}

```

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[should_panic]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

示例 11-8: 测试会造成 `panic!` 的条件

`#[should_panic]` 属性位于 `#[test]` 之后和对应的测试函数之前。让我们看看测试通过时它是什么样子:

```
running 1 test
test tests::greater_than_100 ... ok

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

看起来不错! 现在在代码中引入 `bug`, 移除 `new` 函数在值大于 100 时会 `panic` 的条件:

```
# struct Guess {
#     value: u32,
# }
#
impl Guess {
    pub fn new(value: u32) -> Guess {
        if value < 1 {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess {
            value
        }
    }
}
```

如果运行示例 11-8 的测试, 它会失败:

```
running 1 test
test tests::greater_than_100 ... FAILED

failures:

failures:
    tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured
```

这回并没有得到非常有用的信息, 不过一旦我们观察测试函数, 会发现它标注了 `#[should_panic]`。这个错误意味着代码中函数 `Guess::new(200)` 并没有产生 `panic`。

然而 `should_panic` 测试可能是非常含糊不清的, 因为他们只是告诉我们代码并没有产生 `panic`。`should_panic` 甚至在测试因为其他不同的原因而不是我们期望发生的情况而 `panic` 时也会通过。为了使 `should_panic` 测试更精确, 可以给 `should_panic` 属性增加一个可选的 `expected` 参数。测试工具会确保错误信息中包含其提供的文本。例如, 考虑示例 11-9 中修改过的 `Guess`, 这里 `new` 函数根据其值是过大还或者过小而提供不同的 `panic` 信息:

文件名: `src/lib.rs`

```

struct Guess {
    value: u32,
}

impl Guess {
    pub fn new(value: u32) -> Guess {
        if value < 1 {
            panic!("Guess value must be greater than or equal to 1, got {}. ",
                value);
        } else if value > 100 {
            panic!("Guess value must be less than or equal to 100, got {}. ",
                value);
        }

        Guess {
            value
        }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[should_panic(expected = "Guess value must be less than or equal to 100")]
    fn greater_than_100() {
        Guess::new(200);
    }
}

```

示例 11-9：一个会带有特定错误信息的 `panic!` 条件的测试

这个测试会通过，因为 `should_panic` 属性中 `expected` 参数提供的值是 `Guess::new` 函数 `panic` 信息的子字符串。我们可以指定期望的整个 `panic` 信息，在这个例子中是 `Guess value must be less than or equal to 100, got 200.`。这依赖于 `panic` 有多独特或动态，和你希望测试有多准确。在这个例子中，错误信息的子字符串足以确保函数在 `else if value > 100` 的情况下运行。

为了观察带有 `expected` 信息的 `should_panic` 测试失败时会发生什么，让我们再次引入一个 `bug`，将 `if value < 1` 和 `else if value > 100` 的代码块对换：

```

if value < 1 {
    panic!("Guess value must be less than or equal to 100, got {}. ", value);
} else if value > 100 {
    panic!("Guess value must be greater than or equal to 1, got {}. ", value);
}

```

这一次运行 `should_panic` 测试，它会失败：

```

running 1 test
test tests::greater_than_100 ... FAILED

failures:

---- tests::greater_than_100 stdout ----
thread 'tests::greater_than_100' panicked at 'Guess value must be greater
than or equal to 1, got 200.', src/lib.rs:10
note: Run with `RUST_BACKTRACE=1` for a backtrace.
note: Panic did not include expected string 'Guess value must be less than or
equal to 100'

```

```
failures:
  tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured
```

错误信息表明测试确实如期望 **panic** 了，不过 **panic** 信息是 `did not include expected string 'Guess value must be less than or equal to 100'`。可以看到我们得到的 **panic** 信息，在这个例子中是 `Guess value must be greater than or equal to 1, got 200.`。这样就可以开始寻找 **bug** 在哪了！

现在我们讲完了编写测试的方法，让我们看看运行测试时会发生什么并讨论可以用于 `cargo test` 的不同选项。

运行测试

ch11-02-running-tests.md

commit db08b34db5f1c78b4866b391c802344ec94ecc38

就像 `cargo run` 会编译代码并运行生成的二进制文件一样，`cargo test` 在测试模式下编译代码并运行生成的测试二进制文件。这里有一些选项可以用来改变 `cargo test` 的默认行为。例如，`cargo test` 生成的二进制文件的默认行为是并行的运行所有测试，并捕获测试运行过程中产生的输出避免他们被显示出来，使得阅读测试结果相关的内容变得更容易。你可以指定命令行参数来改变这些默认行为。

这些选项的一部分可以传递给 `cargo test`，而另一些则需要传递给生成的测试二进制文件。为了分隔两种类型的参数，首先列出传递给 `cargo test` 的参数，接着是分隔符 `--`，再之后是传递给测试二进制文件的参数。运行 `cargo test --help` 会告诉你 `cargo test` 的相关参数，而运行 `cargo test -- --help` 则会告诉你位于分隔符 `--` 之后的相关参数。

并行或连续的运行测试

当运行多个测试时，他们默认使用线程来并行的运行。这意味着测试会更快的运行完毕，所以可以更快的得到代码能否工作的反馈。因为测试是在同时运行的，你应该小心测试不能相互依赖或依赖任何共享状态，包括类似于当前工作目录或者环境变量这样的共享环境。

例如，每一个测试都运行一些代码在硬盘上创建一个 `test-output.txt` 文件并写入一些数据。接着每一个测试都读取文件中的数据并断言这个文件包含特定的值，而这个值在每个测试中都是不同的。因为所有测试都是同时运行的，一个测试可能会在另一个测试读写文件过程中覆盖了文件。那么第二个测试就会失败，并不是因为代码不正确，而是因为测试并行运行时相互干涉。一个解决方案是使每一个测试读写不同的文件；另一个是一次运行一个测试。

如果你不希望测试并行运行，或者想要更加精确的控制使用线程的数量，可以传递 `--test-threads` 参数和希望使用线程的数量给测试二进制文件。例如：

```
$ cargo test -- --test-threads=1
```

这里将测试线程设置为 1，告诉程序不要使用任何并行机制。这也会比并行运行花费更多时间，不过测试就不会在存在共享状态时潜在的相互干涉了。

显示函数输出

如果测试通过了，Rust 的测试库默认会捕获打印到标准输出的任何内容。例如，如果在测试中调用 `println!` 而测试通过了，我们将不会在终端看到 `println!` 的输出：只会看到说明测试通过的行。如果测试失败了，就会看到所有标准输出和其他错误信息。

例如，示例 11-20 有一个无意义的函数它打印出其参数的值并接着返回 10。接着还有一个会通过的测试和一个会失败的测试：

文件名: `src/lib.rs`

```
fn prints_and_returns_10(a: i32) -> i32 {
    println!("I got the value {}", a);
    10
}

#[cfg(test)]
```

```
mod tests {
    use super::*;

    #[test]
    fn this_test_will_pass() {
        let value = prints_and_returns_10(4);
        assert_eq!(10, value);
    }

    #[test]
    fn this_test_will_fail() {
        let value = prints_and_returns_10(8);
        assert_eq!(5, value);
    }
}
```

示例 11-10: 一个调用了 `println!` 的函数的测试

运行 `cargo test` 将会看到这些测试的输出:

```
running 2 tests
test tests::this_test_will_pass ... ok
test tests::this_test_will_fail ... FAILED

failures:

---- tests::this_test_will_fail stdout ----
    I got the value 8
thread 'tests::this_test_will_fail' panicked at 'assertion failed: `(left ==
right)` (left: `5`, right: `10`)', src/lib.rs:19
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
    tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured
```

注意输出中哪里也不会出现 `I got the value 4`，这是当测试通过时打印的内容。这些输出被捕获。失败测试的输出，`I got the value 8`，则出现在输出的测试总结部分，同时也显示了测试失败的原因。

如果你希望也能看到通过的测试中打印的值，捕获输出的行为可以通过 `--nocapture` 参数来禁用:

```
$ cargo test -- --nocapture
```

使用 `--nocapture` 参数再次运行示例 11-10 中的测试会显示:

```
running 2 tests
I got the value 4
I got the value 8
test tests::this_test_will_pass ... ok
thread 'tests::this_test_will_fail' panicked at 'assertion failed: `(left ==
right)` (left: `5`, right: `10`)', src/lib.rs:19
note: Run with `RUST_BACKTRACE=1` for a backtrace.
test tests::this_test_will_fail ... FAILED

failures:

failures:
    tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured
```

注意测试的输出和测试结果的输出是相互交叉的；这是由于上一部分讲到的测试是并行运行的。尝试一同使用 `--test-threads=1` 和 `--nocapture` 功能来看看输出是什么样子！

通过名称来运行测试的子集

有时运行整个测试集会耗费很长时间。如果你负责特定位置的代码，你可能会希望只运行这些代码相关的测试。可以向 `cargo test` 传递希望运行的测试的（部分）名称作为参数来选择运行哪些测试。

为了展示如何运行测试的子集，示例 11-11 使用 `add_two` 函数创建了三个测试来供我们选择运行哪一个：

文件名: `src/lib.rs`

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn add_two_and_two() {
        assert_eq!(4, add_two(2));
    }

    #[test]
    fn add_three_and_two() {
        assert_eq!(5, add_two(3));
    }

    #[test]
    fn one_hundred() {
        assert_eq!(102, add_two(100));
    }
}
```

示例 11-11：不同名称的三个测试

如果没有传递任何参数就运行测试，如你所见，所有测试都会并行运行：

```
running 3 tests
test tests::add_two_and_two ... ok
test tests::add_three_and_two ... ok
test tests::one_hundred ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured
```

运行单个测试

可以向 `cargo test` 传递任意测试的名称来只运行这个测试：

```
$ cargo test one_hundred
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running target/debug/deps/adder-06a75b4a1f2515e9

running 1 test
test tests::one_hundred ... ok
```

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

不能像这样指定多个测试名称，只有传递给 `cargo test` 的第一个值才会被使用。

过滤运行多个测试

然而，可以指定测试的部分名称，这样任何名称匹配这个值的测试会被运行。例如，因为头两个测试的名称包含 `add`，可以通过 `cargo test add` 来运行这两个测试：

```
$ cargo test add
  Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
   Running target/debug/deps/adder-06a75b4a1f2515e9

running 2 tests
test tests::add_two_and_two ... ok
test tests::add_three_and_two ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured
```

这运行了所有名字中带有 `add` 的测试。同时注意测试所在的模块作为测试名称的一部分，所以可以通过模块名来过滤运行一个模块中的所有测试。

除非指定否则忽略某些测试

有时一些特定的测试执行起来是非常耗费时间的，所以在大多数运行 `cargo test` 的时候希望能排除他们。与其通过参数列举出所有希望运行的测试，也可以使用 `ignore` 属性来标记耗时的测试来排除他们：

文件名: `src/lib.rs`

```
#[test]
fn it_works() {
    assert!(true);
}

#[test]
#[ignore]
fn expensive_test() {
    // code that takes an hour to run
}
```

对想要排除的测试的 `#[test]` 之后增加了 `#[ignore]` 行。现在如果运行测试，就会发现 `it_works` 运行了，而 `expensive_test` 没有运行：

```
$ cargo test
  Compiling adder v0.1.0 (file:///projects/adder)
  Finished dev [unoptimized + debuginfo] target(s) in 0.24 secs
   Running target/debug/deps/adder-ce99bcc2479f4607

running 2 tests
test expensive_test ... ignored
test it_works ... ok

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

   Doc-tests adder

running 0 tests
```



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

`expensive_test` 被列为 `ignored`，如果只希望运行被忽略的测试，可以使用 `cargo test -- --ignored` 来请求运行他们：

```
$ cargo test -- --ignored
  Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
  Running target/debug/deps/adder-ce99bcc2479f4607

running 1 test
test expensive_test ... ok

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

通过控制运行哪些测试，可以确保运行 `cargo test` 的结果是快速的。当某个时刻需要检查 `ignored` 测试的结果而且你也有时间等待这个结果的话，可以选择执行 `cargo test -- --ignored`。

测试的组织结构

ch11-03-test-organization.md

commit 0665bc5646339a6bcda21838f46d4357b9435e75

正如之前提到的，测试是一个很广泛的学科，而且不同的开发者也采用不同的技术和组织。**Rust** 社区倾向于根据测试的两个主要分类来考虑问题：单元测试（*unit tests*）与集成测试（*integration tests*）。单元测试倾向于更小而更专注，在隔离的环境中一次测试一个模块，也可以测试私有接口。集成测试对于你的库来说则完全是外部的。他们与其他用户使用相同的方式使用你的代码，他们只针对公有接口而且每个测试都会测试多个模块。

编写这两类测试对于从独立和整体的角度保证你的库符合期望是非常重要的。

单元测试

单元测试的目的是在与其他部分隔离的环境中测试每一个单元的代码，以便于快速而准确的定位代码位于何处和是否符合预期。单元测试位于 `src` 目录中，与他们要测试的代码存在于相同的文件中。传统做法是在每个文件中创建包含测试函数的 `tests` 模块，并使用 `cfg(test)` 标注模块。

测试模块和 `cfg(test)`

测试模块的 `#[cfg(test)]` 注解告诉 **Rust** 只在执行 `cargo test` 时才编译和运行测试代码，而在运行 `cargo build` 时不这么做。这在只希望构建库的时候可以节省编译时间，并能节省编译产物的空间因为他们并没有包含测试。我们将会看到因为集成测试位于另一个文件夹，他们并不需要 `#[cfg(test)]` 注解。但是因为单元测试位于与源码相同的文件中，所以使用 `#[cfg(test)]` 来指定他们不应该被包含进编译产物中。

还记得本章第一部分新建的 `adder` 项目吗？**Cargo** 为我们生成了如下代码：

文件名: `src/lib.rs`

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
    }
}
```

这里自动生成了测试模块。`cfg` 属性代表 *configuration*，它告诉 **Rust** 其之后的项只被包含进特定配置中。在这个例子中，配置是 `test`，**Rust** 所提供的用于编译和运行测试的配置。通过使用这个属性，**Cargo** 只会在我们主动使用 `cargo test` 运行测试时才编译测试代码。除了标注为 `#[test]` 的函数之外，还包括测试模块中可能存在的帮助函数。

测试私有函数

测试社区中一直存在关于是否应该对私有函数进行单元测试的论战，而其他语言中难以甚至不可能测试私有函数。不过无论你坚持哪种测试意识形态，**Rust** 的私有性规则确实允许你测试私有函数，由于私有性规则。考虑示例 11-12 中带有私有函数 `internal_adder` 的代码：

文件名: `src/lib.rs`

```
pub fn add_two(a: i32) -> i32 {
    internal_adder(a, 2)
}
```

```
fn internal_adder(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn internal() {
        assert_eq!(4, internal_adder(2, 2));
    }
}
```

示例 11-12：测试私有函数

注意 `internal_adder` 函数并没有标记为 `pub`，不过因为测试也不过是 `Rust` 代码同时 `tests` 也仅仅是另一个模块，我们完全可以在测试中导入和调用 `internal_adder`。如果你并不认为私有函数应该被测试，`Rust` 也不会强迫你这么

集成测试

在 `Rust` 中，集成测试对于需要测试的库来说是完全独立。他们同其他代码一样使用库文件，这意味着他们只能调用作为库公有 `API` 的一部分函数。他们的目的是测试库的多个部分能否一起正常工作。每个能单独正确运行的代码单元集成在一起也可能会出现

tests 目录

为了编写集成测试，需要在项目根目录创建一个 `tests` 目录，与 `src` 同级。`Cargo` 知道如何去寻找这个目录中的集成测试文件。接着可以随意在这个目录中创建任意多的测试文件，`Cargo` 会将每一个文件当作单独的 `crate` 来编译。

让我们试一试吧！保留示例 11-12 中 `src/lib.rs` 的代码。创建一个 `tests` 目录，新建一个文件 `tests/integration_test.rs`，并输入示例 11-13 中的代码。

文件名: `tests/integration_test.rs`

```
extern crate adder;

#[test]
fn it_adds_two() {
    assert_eq!(4, adder::add_two(2));
}
```

示例 11-13：一个 `adder` `crate` 中函数的集成测试

我们在顶部增加了 `extern crate adder`，这在单元测试中是不需要的。这是因为每一个 `tests` 目录中的测试文件都是完全独立的 `crate`，所以需要在每一个文件中导入库。集成测试就像其他库使用者那样通过导入 `crate` 并只使用公有 `API`。

并不需要将 `tests/integration_test.rs` 中的任何代码标注为 `#[cfg(test)]`。`Cargo` 对 `tests` 文件夹特殊处理并只会在运行 `cargo test` 时编译这个目录中的文件。现在就试试运行 `cargo test`：

```
cargo test
```

```

Compiling adder v0.1.0 (file:///projects/adder)
Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
Running target/debug/deps/adder-abcabcabc

running 1 test
test tests::internal ... ok

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

    Running target/debug/deps/integration_test-ce99bcc2479f4607

running 1 test
test it_adds_two ... ok

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

    Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured

```

现在有了三个部分的输出：单元测试、集成测试和文档测试。第一部分单元测试与我们之前见过的一样：每一个单元测试一行（示例 11-12 中有一个叫做 `internal` 的测试），接着是一个单元测试的总结行。

集成测试部分以行 `Running target/debug/deps/integration-test-ce99bcc2479f4607`（输出最后的哈希值可能不同）开头。接着是每一个集成测试中的测试函数一行，以及一个就在 `Doc-tests adder` 部分开始之前的集成测试的总结行。

注意在任意 `src` 文件中增加更多单元测试函数会增加更多单元测试部分的测试结果行。在我们创建的集成测试文件中增加更多测试函数会增加更多集成测试部分的行。每一个集成测试文件有其自己的部分，所以如果在 `tests` 目录中增加更多文件，这里就会有更多集成测试部分。

我们仍然可以通过指定测试函数的名称作为 `cargo test` 的参数来运行特定集成测试。为了运行某个特定集成测试文件中的所有测试，使用 `cargo test` 的 `--test` 后跟文件的名称：

```

$ cargo test --test integration_test
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running target/debug/integration_test-952a27e0126bb565

running 1 test
test it_adds_two ... ok

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

```

这些只是 `tests` 目录中我们指定的文件中的测试。

集成测试中的子模块

随着集成测试的增加，你可能希望在 `tests` 目录增加更多文件，例如根据测试的功能来将测试分组。正如我们之前提到的，每一个 `tests` 目录中的文件都被编译为单独的 `crate`。

将每个集成测试文件当作其自己的 `crate` 来对待有助于创建更类似与终端用户使用 `crate` 那样的单独的作用域。然而，这意味着考虑到第七章学习的如何将代码分隔进模块和文件的知识，`tests` 目录中的文件不能像 `src` 中的文件那样共享相同的行为，。

对于 **tests** 目录中不同文件的行为，通常在如果有一系列有助于多个集成测试文件的帮助函数，而你尝试遵循第七章的步骤将他们提取到一个通用的模块中时显得很明显。例如，如果我们创建了 **tests/common.rs** 并将 `setup` 函数放入其中，这里将放入一些希望能够在多个测试文件的多个测试函数中调用的代码：

文件名: **tests/common.rs**

```
pub fn setup() {  
    // setup code specific to your library's tests would go here  
}
```

如果再次运行测试，将会在测试结果中看到一个对应 **common.rs** 文件的新部分，即便这个文件并没有包含任何测试函数，或者没有任何地方调用了 `setup` 函数：

```
running 1 test  
test tests::internal ... ok  
  
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured  
  
Running target/debug/deps/common-b8b07b6f1be2db70  
  
running 0 tests  
  
test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured  
  
Running target/debug/deps/integration_test-d993c68b431d39df  
  
running 1 test  
test it_adds_two ... ok  
  
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured  
  
Doc-tests adder  
  
running 0 tests  
  
test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

common 出现在测试结果中并显示 `running 0 tests`，这不是我们想要的；我们只是希望能够在其他集成测试文件中分享一些代码罢了。

为了使 **common** 不出现在测试输出中，需要使用第七章学习到的另一个将代码提取到文件的方式：不再创建 **tests/common.rs**，而是创建 **tests/common/mod.rs**。当将 `setup` 代码移动到 **tests/common/mod.rs** 并去掉 **tests/common.rs** 文件之后，测试输出中将不会出现这一部分。**tests** 目录中的子目录不会被作为单独的 **crate** 编译或作为一部分出现在测试输出中。

一旦拥有了 **tests/common/mod.rs**，就可以将其作为模块来在任何集成测试文件中使用。这里是一个 **tests/integration_test.rs** 中调用 `setup` 函数的 `it_adds_two` 测试的例子：

文件名: **tests/integration_test.rs**

```
extern crate adder;  
  
mod common;  
  
#[test]  
fn it_adds_two() {  
    common::setup();  
    assert_eq!(4, adder::add_two(2));  
}
```

注意 `mod common;` 声明与第七章中的模块声明相同。接着在测试函数中就可以调用 `common::setup()` 了。

二进制 `crate` 的集成测试

如果项目是二进制 `crate` 并且只包含 `src/main.rs` 而没有 `src/lib.rs`，这样就不可能在 `tests` 创建集成测试并使用 `extern crate` 导入 `src/main.rs` 中的函数了。只有库 `crate` 向其他 `crate` 暴露了可供调用和使用的函数：二进制 `crate` 只意在单独运行。

这也是 `Rust` 二进制项目明确采用 `src/main.rs` 调用 `src/lib.rs` 中逻辑这样的结构的原因之一。通过这种结构，集成测试就可以使用 `extern crate` 测试库 `crate` 中的主要功能了，而如果这些重要的功能没有问题的话，`src/main.rs` 中的少量代码也就会正常工作且不需要测试。

总结

`Rust` 的测试功能提供了一个如何确保即使函数做出改变也能继续以指定方式运行的途径。单元测试独立的验证库的不同部分并能够测试私有实现细节。集成测试则涉及多个部分结合起来工作时的用例，并像其他外部代码那样测试库的公有 `API`。即使 `Rust` 的类型系统和所有权规则可以帮助避免一些 `bug`，不过测试对于减少代码是否符合期望相关的逻辑 `bug` 仍然是很重要的。

接下来让我们结合本章所学和其他之前章节的知识，在下一章一起编写一个项目！

一个 I/O 项目：构建一个命令程序

ch12-00-an-io-project.md

commit 096db3d06b25692ee701750e1f995ba00d0f5c55

本章既是一个目前所学的很多技能的概括，也是一个更多标准库功能的探索。我们将构建一个与文件和命令行输入/输出交互的命令行工具来练习现在一些你已经掌握的 Rust 技能。

Rust 的运行速度、安全性、单二进制文件 输出和跨平台支持使其成为创建命令程序的绝佳选择，所以我们的项目将创建一个我们自己版本的经典命令行工具：`grep`。`grep` 是“**G**lobally search a **R**egular **E**xpression and **P**rint.”的首字母缩写。`grep` 最简单的使用场景是在特定文件中搜索指定字符串。为此，`grep` 获取一个文件名和一个字符串作为参数，接着读取文件并找到其中包含字符串参数的行，然后打印出这些行。

在这个过程中，我们会展示如何让我们的命令行工具利用很多命令行工具中用到的终端功能。读取环境变量来使得用户可以配置工具的行为。打印到标准错误控制流（`stderr`）而不是标准输出（`stdout`），例如这样用户可以选择将成功输出重定向到文件中而仍然在屏幕上显示错误信息。

一位 Rust 社区的成员，**Andrew Gallant**，已经创建了一个功能完整且非常快速的 `grep` 版本，叫做 `ripgrep`。相比之下，我们的 `grep` 版本将非常简单，本章将教会你一些帮助理解像 `ripgrep` 这样真实项目的背景知识。

这个项目将会结合之前所学的一些内容：

- 代码组织（使用第七章学习的模块）
- `vector` 和字符串（第八章，集合）
- 错误处理（第九章）
- 合理的使用 `trait` 和生命周期（第十章）
- 测试（第十一章）

另外还会简要的讲到闭包、迭代器和 `trait` 对象，他们分别会在第十三章和第十七章中详细介绍。

接受命令行参数

ch12-01-accepting-command-line-arguments.md

commit 50658e654fb6a9208b635179cdd79939aa0ab133

一如之前使用 `cargo new` 新建一个项目。我们称之为 `minigrep` 以便与可能已经安装在系统上的 `grep` 工具相区别：

```
$ cargo new --bin minigrep
   Created binary (application) `minigrep` project
$ cd minigrep
```

第一个任务是让 `minigrep` 能够接受两个命令行参数：文件名和要搜索的字符串。也就是说我们希望能够使用 `cargo run`、要搜索的字符串和被搜索的文件的路径来运行程序，像这样：

```
$ cargo run searchstring example-filename.txt
```

现在 `cargo new` 生成的程序忽略任何传递给它的参数。`crates.io` 上有一些现成的库可以帮助我们接受命令行参数，不过因为正在学习，让我们自己来实现一个。

读取参数值

首先我们需要程序能够获取传递给它的命令行参数的值，为此需要一个 `Rust` 标准库提供的函数：`std::env::args`。这个函数返回一个传递给程序的命令行参数的迭代器（*iterator*）。我们还未讨论到迭代器，第十三章会全面的介绍他们。但是对于我们现在的目的来说只需要明白两点：迭代器生成一系列的值，可以在迭代器上调用 `collect` 方法将其转换为一个 `vector`，比如包含所有迭代器产生元素的 `vector`。

让我们尝试一下：使用示例 12-1 中的代码来读取任何传递给 `minigrep` 的命令行参数并将其收集到一个 `vector` 中。

文件名: `src/main.rs`

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
    println!("{:?}", args);
}
```

示例 12-1：将命令行参数收集到一个 `vector` 中并打印出来

首先使用 `use` 语句来将 `std::env` 模块引入作用域以便可以使用它的 `args` 函数。注意 `std::env::args` 函数被嵌套进了两层模块中。正如第七章讲到的，当所需函数嵌套了多于一层模块时，通常将父模块引入作用域，而不是其自身。这便于我们利用 `std::env` 中的其他函数。这比增加了 `use std::env::args;` 后仅仅使用 `args` 调用函数要更明确一些；这样容易被错认成一个定义于当前模块的函数。

args 函数和无效的 Unicode

注意 `std::env::args` 在其任何参数包含无效 `Unicode` 字符时会 `panic`。如果你需要接受包含无效 `Unicode` 字符的参数，使用 `std::env::args_os` 代替。这个函数返回 `OsString` 值而不是 `String` 值。这里出于简单考虑使用了 `std::env::args`，因为 `OsString` 值每个平台都不一样而且比 `String` 值处理起来更复杂。

在 `main` 函数的第一行，我们调用了 `env::args`，并立即使用 `collect` 来创建了一个包含迭代器所有值的 `vector`。`collect` 可以被用来创建很多类型的集合，所以这里显式注明 `args` 的类型来指定我们需要一个字符串 `vector`。虽然在 `Rust` 中我们很少会需要注明类型，`collect` 就是一个经常需要注明类型的函数，因为 `Rust` 不能推断出你想要什么类型的集合。

最后，我们使用调试格式 `:?` 打印出 `vector`。让我们尝试不用参数运行代码，接着用两个参数：

```
$ cargo run
["target/debug/minigrep"]

$ cargo run needle haystack
...snip...
["target/debug/minigrep", "needle", "haystack"]
```

你可能注意到了 `vector` 的第一个值是 `"target/debug/minigrep"`，它是我们二进制文件的名称。这与 `C` 中的参数列表的行为相符合，并使得程序可以在执行过程中使用它的名字。能够访问程序名称在需要在信息中打印时，或者需要根据执行程序所使用的命令行别名来改变程序行为时显得很方便，不过考虑到本章的目的，我们将忽略它并只保存所需的两个参数。

将参数值保存进变量

打印出参数 `vector` 中的值展示了程序可以访问指定为命令行参数的值。现在需要将这两个参数的值保存进变量这样就可以在程序的余下部分使用这些值。让我们如示例 12-2 这样做：

文件名: `src/main.rs`

```
use std::env;

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

    let query = &args[1];
    let filename = &args[2];

    println!("Searching for {}", query);
    println!("In file {}", filename);
}
```

示例 12-2: 创建变量来存放查询参数和文件名参数

正如之前打印出 `vector` 时所看到的，程序的名称占据了 `vector` 的第一个值 `args[0]`，所以我们从索引 `1` 开始。`minigrep` 获取的第一个参数是需要搜索的字符串，所以将其将第一个参数的引用存放在变量 `query` 中。第二个参数将是文件名，所以将第二个参数的引用放入变量 `filename` 中。

我们将临时打印出这些变量的值，再一次证明代码如我们期望的那样工作。让我们使用参数 `test` 和 `sample.txt` 再次运行这个程序：

```
$ cargo run test sample.txt
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running `target/debug/minigrep test sample.txt`
Searching for test
In file sample.txt
```

好的，它可以工作！我们将所需的参数值保存进了对应的变量中。之后会增加一些错误处理来应对类似用户没有提供参数的情况，不过现在我们将忽略他们并开始增加读取文件功能。

读取文件

ch12-02-reading-a-file.md

commit b693c8400817f1022820fd63e3529cbecc35070c

接下来我们将读取由命令行文件名参数指定的文件。首先，需要一个用来测试的示例文件——用来确保 `minigrep` 正常工作的最好的文件是拥有少量文本和多个行且有一些重复单词的文件。示例 12-3 是一首艾米莉·狄金森（Emily Dickinson）的诗，它正适合这个工作！在项目根目录创建一个文件 `poem.txt`，并输入诗 "I'm nobody! Who are you?":

文件名: `poem.txt`

```
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

示例 12-3: 艾米莉·狄金森的诗 "I'm nobody! Who are you?", 一个好的测试用例

创建完这个文件之后，修改 `src/main.rs` 并增加如示例 12-4 所示的打开文件的代码：

文件名: `src/main.rs`

```
use std::env;
use std::fs::File;
use std::io::prelude::*;

fn main() {
    # let args: Vec<String> = env::args().collect();
    #
    # let query = &args[1];
    # let filename = &args[2];
    #
    # println!("Searching for {}", query);
    // ...snip...
    println!("In file {}", filename);

    let mut f = File::open(filename).expect("file not found");

    let mut contents = String::new();
    f.read_to_string(&mut contents)
        .expect("something went wrong reading the file");

    println!("With text:\n{}", contents);
}
```

示例 12-4: 读取第二个参数所指定的文件内容

首先，我们增加了更多的 `use` 语句来引入标准库中的相关部分：需要 `std::fs::File` 来处理文件，而 `std::io::prelude::*` 则包含许多对于 I/O 包括文件 I/O 有帮助的 `trait`。类似于 Rust 有一个通用的 `prelude` 来自动引入特定内容，`std::io` 也有其自己的 `prelude` 来引入处理 I/O 时所需的通用内容。不同于默认的 `prelude`，必须显式 `use` 位于 `std::io` 中的 `prelude`。

在 `main` 中，我们增加了三点内容：第一，通过传递变量 `filename` 的值调用 `File::open` 函数来获取文件的可变句柄。创建了叫做 `contents` 的变量并将其设置为一个可变的，空的 `String`。它将会存放之后读取的文件的内容。第三，对文件句柄调用 `read_to_string` 并传递 `contents` 的可变引用作为参数。

在这些代码之后，我们再次增加了临时的 `println!` 打印出读取文件后 `contents` 的值，这样就可以检查目前为止的程序能否工作。

尝试运行这些代码，随意指定一个字符串作为第一个命令行参数（因为还未实现搜索功能的部分）而将 *poem.txt* 文件将作为第二个参数：

```
$ cargo run the poem.txt
   Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
   Running `target/debug/minigrep the poem.txt`
Searching for the
In file poem.txt
With text:
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

好的！代码读取并打印出了文件的内容。虽然它还有一些瑕疵：`main` 函数有着多个职能，通常函数只负责一个功能的话会更简洁并易于维护。另一个问题是没有尽可能的处理错误。虽然我们的程序还很小，这些瑕疵并不是什么大问题，不过随着程序功能的丰富，将会越来越难以用简单的方法修复他们。在开发程序时，及早开始重构是一个最佳实践，因为重构少量代码时要容易的多，所以让我们现在就开始吧。

重构改进模块性和错误处理

ch12-03-improving-error-handling-and-modularity.md

commit 5908c59a5a4cc58fd863605b80b295a335c2cbdf

为了改善我们的程序这里有四个问题需要修复，而且他们都与程序的组织方式和如何处理潜在错误有关。

第一，`main` 现在进行了两个任务：它解析了参数并打开了文件。对于一个这样的小函数，这并不是一个大问题。然而如果 `main` 中的功能持续增加，`main` 函数处理的独立的任务也会增加。当函数承担了更多责任，它就更难以推导，更难以测试，并且更难以在不破坏其他部分的情况下做出修改。最好能分离出功能以便每个函数就负责一个任务。

这同时也关系到第二个问题：`search` 和 `filename` 是程序中的配置变量，而像 `f` 和 `contents` 则用来执行程序逻辑。随着 `main` 函数的增长，就需要引入更多的变量到作用域中，而当作用域中有更多的变量时，将更难以追踪每个变量的目的。最好能将配置变量组织进一个结构这样就能使他们的目的更明确了。

第三个问题是如果打开文件失败我们使用 `expect` 来打印出错误信息，不过这个错误信息只是说 `file not found`。除了缺少文件之外还有很多打开文件可能失败的方式：例如，文件可能存在，不过可能没有打开它的权限。如果我们现在就出于这种情况，打印出的 `file not found` 错误信息就给了用户错误的建议！

第四，我们不停的使用 `expect` 来处理不同的错误，如果用户没有指定足够的参数来运行程序，他们会从 `Rust` 得到 "index out of bounds" 错误，而这并不能明确的解释问题。如果所有的错误处理都位于一处这样将来的维护者在需要修改错误处理逻辑时就只需要考虑这一处代码。将所有的错误处理都放在一处也有助于确保我们打印的错误信息对终端用户来说是有意义的。

让我们通过重构项目来解决这些问题。

二进制项目的关注分离

`main` 函数负责多个任务的组织问题在许多二进制项目中很常见。所以 `Rust` 社区开发出一类在 `main` 函数开始变得庞大时进行二进制程序的关注分离的指导性过程。这些过程有如下步骤：

1. 将程序拆分成 `main.rs` 和 `lib.rs` 并将程序的逻辑放入 `lib.rs` 中。
2. 当命令行解析逻辑比较小时，可以保留在 `main.rs` 中。
3. 当命令行解析开始变得复杂时，也同样将其从 `main.rs` 提取到 `lib.rs` 中。
4. 经过这些过程之后保留在 `main` 函数中的责任应该被限制为：
 - 使用参数值调用命令行解析逻辑
 - 设置任何其他的配置
 - 调用 `lib.rs` 中的 `run` 函数
 - 如果 `run` 返回错误，则处理这个错误

这个模式的一切就是为了关注分离：`main.rs` 处理程序运行，而 `lib.rs` 处理所有的真正的任务逻辑。因为不能直接测试 `main` 函数，这个结构通过将所有的程序逻辑移动到 `lib.rs` 的函数中使得我们可以测试他们。仅仅保留在 `main.rs` 中的代码将足够小以便阅读就可以验证其正确性。

提取参数解析器

首先，我们将解析参数的功能提取到一个 `main` 将会调用的函数中，为将命令行解析逻辑移动到 `src/lib.rs` 做准备。示例 12-5 中展示了新 `main` 函数的开头，它调用了新函数 `parse_config`。目前它仍将定义在 `src/main.rs` 中：

文件名: `src/main.rs`

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

    let (query, filename) = parse_config(&args);

    // ...snip...
}

fn parse_config(args: &[String]) -> (&str, &str) {
    let query = &args[1];
    let filename = &args[2];

    (query, filename)
}
```

示例 12-5: 从 `main` 中提取出 `parse_config` 函数

我们仍然将命令行参数收集进一个 **vector**，不过不同于在 `main` 函数中将索引 1 的参数值赋值给变量 `query` 和将索引 2 的值赋值给变量 `filename`，我们将整个 **vector** 传递给 `parse_config` 函数。接着 `parse_config` 函数将包含决定哪个参数该放入哪个变量的逻辑，并将这些值返回到 `main`。仍然在 `main` 中创建变量 `query` 和 `filename`，不过 `main` 不再负责处理命令行参数与变量如何对应。

这对我们这小程序可能有点大材小用，不过我们将采用小的、增量的步骤进行重构。在做出这些改变之后，再次运行程序并验证参数解析是否仍然正常。经常验证你的进展是一个好习惯，这样在遇到问题时能帮助你定位问题的成因。

组合配置值

我们可以采取另一个小的步骤来进一步改善这个函数。现在函数返回一个元组，不过立刻又就将元组拆成了独立的部分。这是一个我们可能没有进行正确抽象的信号。

另一个表明还有改进空间的迹象是 `parse_config` 名称的 `config` 部分，它暗示了我们返回的两个值是相关的并都是一个配置值的一部分。目前除了将这两个值组合进元组之外并没有表达这个数据结构的意义：我们可以将这两个值放入一个结构体并给每个字段一个有意义的名字。这会让未来的维护者更容易理解不同的值如何相互关联以及他们的目的。

注意：一些同学将这种拒绝使用相对而言更为合适的复合类型而使用基本类型的模式称为 **基本类型偏执**（*primitive obsession*）。

示例 12-6 展示了新定义的结构体 `Config`，它有字段 `query` 和 `filename`。我们也改变了 `parse_config` 函数来返回一个 `Config` 结构体的实例，并更新 `main` 来使用结构体字段而不是单独的变量：

文件名: `src/main.rs`

```
# use std::env;
# use std::fs::File;
#
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = parse_config(&args);

    println!("Searching for {}", config.query);
    println!("In file {}", config.filename);

    let mut f = File::open(config.filename).expect("file not found");

    // ...snip...
```

```

}

struct Config {
    query: String,
    filename: String,
}

fn parse_config(args: &[String]) -> Config {
    let query = args[1].clone();
    let filename = args[2].clone();

    Config { query, filename }
}

```

示例 12-6: 重构 `parse_config` 返回一个 `Config` 结构体实例

`parse_config` 的签名表明它现在返回一个 `Config` 值。在 `parse_config` 的函数体中，之前返回引用了 `args` 中 `String` 值的字符串 `slice`，现在我们选择定义 `Config` 来包含拥有所有权的 `String` 值。`main` 中的 `args` 变量是参数值的所有者并只允许 `parse_config` 函数借用他们，这意味着如果 `Config` 尝试获取 `args` 中值的所有权将违反 `Rust` 的借用规则。

还有许多不同的方式可以处理 `String` 的数据，而最简单但有些不太高效的方式是调用这些值的 `clone` 方法。这会生成 `Config` 实例可以拥有的数据的完整拷贝，不过会比储存字符串数据的引用消耗更多的时间和内存。不过拷贝数据使得代码显得更加直白因为无需管理引用的生命周期，所以在这种情况下牺牲一小部分性能来换取简洁性的取舍是值得的。

使用 `clone` 的权衡取舍

由于其运行时消耗，许多 `Rustacean` 之间有一个趋势是倾向于避免使用 `clone` 来解决所有权问题。在关于迭代器的第十三章中，我们将会学习如何更有效率的处理这种情况，不过现在，复制一些字符串来取得进展是没有问题的，因为只会进行一次这样的拷贝，而且文件名和要搜索的字符串都比较短。在第一轮编写时拥有一个可以工作但有点低效的程序要比尝试过度优化代码更好一些。随着你对 `Rust` 更加熟练，将能更轻松的直奔合适的方法，不过现在调用 `clone` 是完全可以接受的。

我们更新 `main` 将 `parse_config` 返回的 `Config` 实例放入变量 `config` 中，并将之前分别使用 `search` 和 `filename` 变量的代码更新为现在的使用 `Config` 结构体的字段的代码。

现在代码更明确的表现了我们的意图，`query` 和 `filename` 是相关联的并且他们的目的是配置程序如何工作的。任何使用这些值的代码就知道在 `config` 实例中对应目的的字段的名称中寻找他们。

创建一个 `Config` 构造函数

目前为止，我们将负责解析命令行参数的逻辑从 `main` 提取到了 `parse_config` 函数中，这有助于我们看清值 `query` 和 `filename` 是相互关联的并应该在代码中表现这种关系。接着我们增加了 `Config` 结构体来描述 `query` 和 `filename` 的相关性，并能够从 `parse_config` 函数中将这些值的名称作为结构体字段名称返回。

所以现在 `parse_config` 函数的目的是创建一个 `Config` 实例，我们可以将 `parse_config` 从一个普通函数变为一个叫做 `new` 的与结构体关联的函数。做出这个改变使得代码更符合习惯：可以像标准库中的 `String` 调用 `String::new` 来创建一个该类型的实例那样，将 `parse_config` 变为一个与 `Config` 关联的 `new` 函数。示例 12-7 展示了需要做出的修改：

文件名: `src/main.rs`

```

# use std::env;
#
fn main() {

```

```

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

let config = Config::new(&args);

// ...snip...
}

# struct Config {
#     query: String,
#     filename: String,
# }
#
// ...snip...

impl Config {
    fn new(args: &[String]) -> Config {
        let query = args[1].clone();
        let filename = args[2].clone();

        Config { query, filename }
    }
}

```

示例 12-7: 将 `parse_config` 变为 `Config::new`

这里将 `main` 中调用 `parse_config` 的地方更新为调用 `Config::new`。我们将 `parse_config` 的名字改为 `new` 并将其移动到 `impl` 块中, 这使得 `new` 函数与 `Config` 相关联。再次尝试编译并确保它可以工作。

修复错误处理

现在我们开始修复错误处理。回忆一下之前提到过如果 `args` `vector` 包含少于 3 个项并尝试访问 `vector` 中索引 1 或索引 2 的值会造成程序 `panic`。尝试不带任何参数运行程序; 这将看起来像这样:

```

$ cargo run
   Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running `target/debug/minigrep`
thread 'main' panicked at 'index out of bounds: the len is 1
but the index is 1', /stable-dist-rustc/build/src/libcollections/vec.rs:1307
note: Run with `RUST_BACKTRACE=1` for a backtrace.

```

`index out of bounds: the len is 1 but the index is 1` 是一个针对程序员的错误信息, 然而这并不能真正帮助终端用户理解发生了什么和他们应该做什么。现在就让我们修复它吧。

改善错误信息

在示例 12-8 中, 在 `new` 函数中增加了一个检查在访问索引 1 和 2 之前检查 `slice` 是否足够长。如果 `slice` 不够长, 我们使用一个更好的错误信息 `panic` 而不是 `index out of bounds` 信息:

文件名: `src/main.rs`

```

// ...snip...
fn new(args: &[String]) -> Config {
    if args.len() < 3 {
        panic!("not enough arguments");
    }
    // ...snip...
}

```

示例 12-8: 增加一个参数数量检查

这类似于示例 9-8 中的 `Guess::new` 函数，那里如果 `value` 参数超出了有效值的范围就调用 `panic!`。不同于检查值的范围，这里检查 `args` 的长度至少是 3，而函数的剩余部分则可以在假设这个条件成立的基础上运行。如果 `args` 少于 3 个项，则这个条件将为真，并调用 `panic!` 立即终止程序。

有了 `new` 中这几行额外的代码，再次不带任何参数运行程序并看看现在错误看起来像什么：

```
$ cargo run
   Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
   Running `target/debug/minigrep`
thread 'main' panicked at 'not enough arguments', src/main.rs:29
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

这个输出就好多了，现在有了一个合理的错误信息。然而，我们还有一堆额外的信息不希望提供给用户。所以在这里使用示例 9-8 中的技术可能不是最好的；正如第九章所讲到的一样，`panic!` 的调用更趋向于程序上的问题而不是使用上的问题。相反我们可以使用那一章学习的另一个技术：返回一个可以表明成功或错误的 `Result`。

从 `new` 中返回 `Result` 而不是调用 `panic!`

我们可以选择返回一个 `Result` 值，它在成功时会包含一个 `Config` 的实例，而在错误时会描述问题。当 `Config::new` 与 `main` 交流时，可以使用 `Result` 类型来表明这里存在问题。接着修改 `main` 将 `Err` 成员转换为对用户更友好的错误，而不是 `panic!` 调用产生的关于 `thread 'main'` 和 `RUST_BACKTRACE` 的文本。

示例 12-9 展示了为了返回 `Result` 在 `Config::new` 的返回值和函数体中所需的改变：

文件名: `src/main.rs`

```
impl Config {
    fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let filename = args[2].clone();

        Ok(Config { query, filename })
    }
}
```

示例 12-9：从 `Config::new` 中返回 `Result`

现在 `new` 函数返回一个 `Result`，在成功时带有一个 `Config` 实例而在出现错误时带有一个 `&'static str`。回忆一下第十章“静态生命周期”中讲到 `&'static str` 是一个字符串面值，也是目前的错误信息。

`new` 函数体中有两处修改：当没有足够参数时不再调用 `panic!`，而是返回 `Err` 值。同时我们将 `Config` 返回值包装进 `Ok` 成员中。这些修改使得函数符合其新的类型签名。

通过让 `Config::new` 返回一个 `Err` 值，这就允许 `main` 函数处理 `new` 函数返回的 `Result` 值并在出现错误的情况更明确的结束进程。

`Config::new` 调用并处理错误

为了处理错误情况并打印一个对用户友好的信息，我们需要像示例 12-10 那样更新 `main` 函数来处理现在 `Config::new` 返回的 `Result`。另外还需要负责手动实现 `panic!` 的使用非零错误码退出命令行工具的工作。非零的退出状态是一个告诉调用程序的进程我们的程序以错误状态退出的惯例信号。

文件名: `src/main.rs`

```
use std::process;

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

    let config = Config::new(&args).unwrap_or_else(|err| {
        println!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    // ...snip...
```

示例 12-10: 如果新建 `Config` 失败则使用错误码退出

在上面的示例中，使用了一个之前没有涉及到的方法：`unwrap_or_else`，它定义于标准库的 `Result<T, E>` 上。使用 `unwrap_or_else` 可以进行一些自定义的非 `panic!` 的错误处理。当 `Result` 是 `Ok` 时，这个方法的行为类似于 `unwrap`：它返回 `Ok` 内部封装的值。然而，当其值是 `Err` 时，该方法会调用一个闭包（*closure*），也就是一个我们定义的作为参数传递给 `unwrap_or_else` 的匿名函数。第十三章会更详细的介绍闭包。现在你需要理解的是 `unwrap_or_else` 会将 `Err` 的内部值，也就是示例 12-9 中增加的 `not enough arguments` 静态字符串的情况，传递给闭包中位于两道竖线间的参数 `err`。闭包中的代码在其运行时可以使用这个 `err` 值。

我们新增了一个 `use` 行来从标准库中导入 `process`。在错误的情况闭包中将被运行的代码只有两行：我们打印出了 `err` 值，接着调用了 `std::process::exit`。`process::exit` 会立即停止程序并将传递给它的数字作为退出状态码。这类似于示例 12-8 中使用的基于 `panic!` 的错误处理，除了不会再得到所有的额外输出了。让我们试试：

```
$ cargo run
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.48 secs
Running `target/debug/minigrep`
Problem parsing arguments: not enough arguments
```

非常好！现在输出对于用户来说就友好多了。

从 `main` 提取逻辑

现在我们完成了配置解析的重构：让我们转向程序的逻辑。正如“二进制项目的关注分离”部分所展开的讨论，我们将提取一个叫做 `run` 的函数来存放目前 `main` 函数中不属于设置配置或处理错误的所有逻辑。一旦完成这些，`main` 函数将简明的足以通过观察来验证，而我们将能够为所有其他逻辑编写测试。

示例 12-11 展示了提取出来的 `run` 函数。目前我们只进行小的增量式的提取函数的改进。我们仍将在 `src/main.rs` 中定义这个函数：

文件名: `src/main.rs`

```
fn main() {
    // ...snip...

    println!("Searching for {}", config.query);
    println!("In file {}", config.filename);

    run(config);
}

fn run(config: Config) {
    let mut f = File::open(config.filename).expect("file not found");
```

```

    let mut contents = String::new();
    f.read_to_string(&mut contents)
        .expect("something went wrong reading the file");

    println!("With text:\n{}", contents);
}

// ...snip...

```

示例 12-11：提取 `run` 函数来包含剩余的程序逻辑

现在 `run` 函数包含了 `main` 中从读取文件开始的剩余的所有逻辑。`run` 函数获取一个 `Config` 实例作为参数。

从 `run` 函数中返回错误

通过将剩余的逻辑分离进 `run` 函数而不是留在 `main` 中，就可以像示例 12-9 中的 `Config::new` 那样改进错误处理。不再通过 `expect` 允许程序 `panic`，`run` 函数将会在出错时返回一个 `Result<T, E>`。这让我们进一步以一种对用户友好的方式统一 `main` 中的错误处理。示例 12-12 展示了 `run` 签名和函数体中的改变：

文件名: `src/main.rs`

```

use std::error::Error;

// ...snip...

fn run(config: Config) -> Result<(), Box<Error>> {
    let mut f = File::open(config.filename)?;

    let mut contents = String::new();
    f.read_to_string(&mut contents)?;

    println!("With text:\n{}", contents);

    Ok(())
}

```

示例 12-12：修改 `run` 函数返回 `Result<(), Box<Error>>`。之前这个函数返回 `unit` 类型 `()`，现在它仍然保持作为 `Ok` 时的返回值。

对于错误类型，使用了 `trait` 对象 `Box<Error>`（在开头使用了 `use` 语句将 `std::error::Error` 引入作用域）。第十七章会涉及 `trait` 对象。目前只需知道 `Box<Error>` 意味着函数会返回实现了 `Error trait` 的类型，不过无需指定具体将会返回的值的类型。这提供了在不同的错误场景可能有不同类型的错误返回值的灵活性。

第二个改变是去掉了 `expect` 调用并替换为第九章讲到的 `?`。不同于遇到错误就 `panic!`，这会从函数中返回错误值并让调用者来处理它。

第三个修改是现在成功时这个函数会返回一个 `Ok` 值。因为 `run` 函数签名中声明成功类型返回值是 `()`，这意味着需要将 `unit` 类型值包装进 `Ok` 值中。`Ok(())` 一开始看起来有点奇怪，不过这样使用 `()` 是表明我们调用 `run` 只是为了它的副作用的惯用方式；它并没有返回什么有意义的值。

上述代码能够编译，不过会有一个警告：

```

warning: unused result which must be used, #[warn(unused_must_use)] on by
default
--> src/main.rs:39:5
   |
39 |     run(config);
   |     ^^^^^^^^^^^

```

Rust 提示我们的代码忽略了 `Result` 值，它可能表明这里存在一个错误。虽然我们没有检查这里是否有一个错误，而编译器提醒我们这里应该有一些错误处理代码！现在就让我们修正他们。

处理 `main` 中 `run` 返回的错误

我们将检查错误并使用类似示例 12-10 中 `Config::new` 处理错误的技术来处理他们，不过有一些细微的不同：

文件名: `src/main.rs`

```
fn main() {
    // ...snip...

    println!("Searching for {}", config.query);
    println!("In file {}", config.filename);

    if let Err(e) = run(config) {
        println!("Application error: {}", e);

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

我们使用 `if let` 来检查 `run` 是否返回一个 `Err` 值，不同于 `unwrap_or_else`，并在出错时调用 `process::exit(1)`。`run` 并不返回像 `Config::new` 返回的 `Config` 实例那样需要 `unwrap` 的值。因为 `run` 在成功时返回 `()`，而我们只关心发现一个错误，所以并不需要 `unwrap_or_else` 来返回未封装的值，因为它只会是 `()`。

不过两个例子中 `if let` 和 `unwrap_or_else` 的函数体都一样：打印出错误并退出。

将代码拆分到库 `crate`

现在项目看起来好多了！现在我们将要拆分 `src/main.rs` 并将一些代码放入 `src/lib.rs`，这样就能测试他们并拥有一个拥有更少功能的 `main` 函数。

让我们将如下代码片段从 `src/main.rs` 移动到新文件 `src/lib.rs` 中：

- `run` 函数定义
- 相关的 `use` 语句
- `Config` 的定义
- `Config::new` 函数定义

现在 `src/lib.rs` 的内容应该看起来像示例 12-13（为了简洁省略了函数体）：

文件名: `src/lib.rs`

```
use std::error::Error;
use std::fs::File;
use std::io::prelude::*;

pub struct Config {
    pub query: String,
    pub filename: String,
}

impl Config {
    pub fn new(args: &[String]) -> Result<Config, &'static str> {
        // ...snip...
    }
}
```

```

}

pub fn run(config: Config) -> Result<(), Box<Error>> {
    // ...snip...
}

```

示例 12-13: 将 `Config` 和 `run` 移动到 `src/lib.rs`

这里使用了公有的 `pub` : 在 `Config`、其字段和其 `new` 方法, 以及 `run` 函数上。现在我们有了一个拥有可以测试的公有 API 的库 `crate` 了。

现在需要在 `src/main.rs` 中使用 `extern crate minigrep` 将移动到 `src/lib.rs` 的代码引入二进制 `crate` 的作用域。接着我们将增加一个 `use minigrep::Config` 行将 `Config` 类型引入作用域, 并使用库 `crate` 的名称作为 `run` 函数的前缀, 如示例 12-14 所示:

Filename: `src/main.rs`

```

extern crate minigrep;

use std::env;
use std::process;

use minigrep::Config;

fn main() {
    // ...snip...
    if let Err(e) = minigrep::run(config) {
        // ...snip...
    }
}

```

示例 12-14: 将 `minigrep` `crate` 引入 `src/main.rs` 的作用域

为了将库 `crate` 引入二进制 `crate`, 我们使用 `extern crate minigrep`。接着增加 `use minigrep::Config` 将 `Config` 类型引入作用域, 并使用 `crate` 名作为 `run` 函数的前缀。通过这些重构, 所有功能应该能够联系在一起并运行了。运行 `cargo run` 来确保一切都正确的衔接在一起。

哇哦! 这可有很多的工作, 不过我们为将来成功打下了基础。现在处理错误将更容易, 同时代码也更加模块化。从现在开始几乎所有的工作都将在 `src/lib.rs` 中进行。

让我们利用这些新创建的模块的优势来进行一些在旧代码中难以展开的工作, 他们在新代码中却很简单: 编写测试!

测试库的功能

ch12-04-testing-the-librarys-functionality.md

commit 5908c59a5a4cc58fd863605b80b295a335c2cbdf

现在我们将逻辑提取到了 `src/lib.rs` 并将所有的参数解析和错误处理留在了 `src/main.rs` 中，为代码的核心功能编写测试将更加容易。我们可以直接使用多种参数调用函数并检查返回值而无需从命令行运行二进制文件了。如果你愿意的话，请自行为 `Config::new` 和 `run` 函数的功能编写一些测试。

在这一部分，我们将遵循测试驱动开发（Test Driven Development, TTD）的模式来逐步增加 `minigrep` 的搜索逻辑。这是一个软件开发技术，它遵循如下步骤：

1. 编写一个会失败的测试，并运行它以确保其因为你期望的原因失败。
2. 编写或修改刚好足够的代码来使得新的测试通过。
3. 重构刚刚增加或修改的代码，并确保测试仍然能通过。
4. 重复上述步骤！

这只是众多编写软件的方法之一，不过 TDD 有助于驱动代码的设计。在编写能使测试通过的代码之前编写测试有助于在开发过程中保持高测试覆盖率。

我们将测试驱动实现实际在文件内容中搜索查询字符串并返回匹配的行示例的功能。我们将在一个叫做 `search` 的函数中增加这些功能。

编写失败测试

首先，去掉 `src/lib.rs` 和 `src/main.rs` 中的 `println!` 语句，因为不再真正需要他们了。接着我们会像第十一章那样增加一个 `test` 模块和一个测试函数。测试函数指定了 `search` 函数期望拥有的行为：它会获取一个需要查询的字符串和用来查询的文本，并只会返回包含请求的文本行。示例 12-15 展示了这个测试：

文件名: `src/lib.rs`

```
# fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
#     vec![]
# }
#
#[cfg(test)]
mod test {
    use super::*;

    #[test]
    fn one_result() {
        let query = "duct";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.";

        assert_eq!(
            vec!["safe, fast, productive."],
            search(query, contents)
        );
    }
}
```

示例 12-15：创建一个我们期望的 `search` 函数的失败测试

这里选择使用 "duct" 作为这个测试中需要搜索的字符串。用来搜索的文本有三行，其中只有一行包含 "duct"。我们断言 `search` 函数的返回值只包含期望的那一行。

我们还不能运行这个测试并看到它失败，因为它甚至都还不能编译！我们将增加足够的代码来使其能够编译：一个总是会返回空 `vector` 的 `search` 函数定义，如示例 12-16 所示。一旦有了它，这个测试应该能够编译并因为空 `vector` 并不匹配一个包含一行 "safe, fast, productive." 的 `vector` 而失败。

文件名: `src/lib.rs`

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {  
    vec![]  
}
```

示例 12-16: 刚好足够使测试通过编译的 `search` 函数定义

注意需要在 `search` 的签名中定义一个显式生命周期 `'a` 并用于 `contents` 参数和返回值。回忆一下第十章中讲到生命周期参数指定哪个参数的生命周期与返回值的生命周期相关联。在这个例子中，我们表明返回的 `vector` 中应该包含引用参数 `contents`（而不是参数 `query`）`slice` 的字符串 `slice`。

换句话说，我们告诉 **Rust** 函数 `search` 返回的数据将与 `search` 函数中的参数 `contents` 的数据存在的一样久。这是非常重要的！为了使这个引用有效那么被 `slice` 引用的数据也需要保持有效；如果编译器认为我们是在创建 `query` 而不是 `contents` 的字符串 `slice`，那么安全检查将是不正确的。

如果尝试不用生命周期编译的话，我们将得到如下错误：

```
error[E0106]: missing lifetime specifier  
--> src/lib.rs:5:47  
  |  
5 | fn search(query: &str, contents: &str) -> Vec<&str> {  
  |                                         ^ expected lifetime parameter  
  |  
= help: this function's return type contains a borrowed value, but the  
signature does not say whether it is borrowed from `query` or `contents`
```

Rust 不可能知道我们需要的是哪一个参数，所以需要告诉它。因为参数 `contents` 包含了所有的文本而且我们希望返回匹配的那部分文本，所以我们知道 `contents` 是应该要使用生命周期语法来与返回值相关联的参数。

其他语言中并不需要你在函数签名中将参数与返回值相关联，所以这么做可能仍然感觉有些陌生，随着时间的推移会越来越容易。你可能想要将这个例子与第十章中生命周期语法部分做对比。

现在试尝试运行测试：

```
$ cargo test  
...warnings...  
    Finished dev [unoptimized + debuginfo] target(s) in 0.43 secs  
    Running target/debug/deps/minigrep-abcabcabc  
  
running 1 test  
test test::one_result ... FAILED  
  
failures:  
  
---- test::one_result stdout ----  
thread 'test::one_result' panicked at 'assertion failed: `(left == right)`  
(left: `["safe, fast, productive."]`, right: `[]`)', src/lib.rs:16  
note: Run with `RUST_BACKTRACE=1` for a backtrace.  
  
failures:
```

```
test::one_result

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured

error: test failed
```

好的，测试失败了，这正是我们所期望的。修改代码来让测试通过吧！

编写使测试通过的代码

目前测试之所以会失败是因为我们总是返回一个空的 `vector`。为了修复并实现 `search`，我们的程序需要遵循如下步骤：

- 遍历每一行文本。
- 查看这一行是否包含要搜索的字符串。
- 如果有，将这一行加入返回示例中。
- 如果没有，什么也不做。
- 返回匹配到的示例

让我们一步一步的来，从遍历每行开始。

使用 `lines` 方法遍历每一行

`Rust` 有一个有助于一行一行遍历字符串的方法，出于方便它被命名为 `lines`，它如示例 12-17 这样工作：

Filename: `src/lib.rs`

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        // do something with line
    }
}
```

示例 12-17：遍历 `contents` 的每一行

`lines` 方法返回一个迭代器。第十三章会深入了解迭代器，不过我们已经在示例 3-6 中见过使用迭代器的方法，在那里使用了一个 `for` 循环和迭代器在一个集合的每一项上运行了一些代码。

用查询字符串搜索每一行

接下来将会增加检查当前行是否包含查询字符串的功能。幸运的是，字符串类型为此也有一个叫做 `contains` 的实用方法！如示例 12-18 所示在 `search` 函数中加入 `contains` 方法调用：

文件名: `src/lib.rs`

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        if line.contains(query) {
            // do something with line
        }
    }
}
```

示例 12-18：增加检查文本行是否包含 `query` 中字符串的功能

存储匹配的行

最后我们需要一个方法来存储包含查询字符串的行。为此可以在 `for` 循环之前创建一个可变的 `vector` 并调用 `push` 方法在 `vector` 中存放一个 `line`。在 `for` 循环之后，返回这个 `vector`，如示例 12-19 所示：

文件名: `src/lib.rs`

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}
```

示例 12-19: 储存匹配的行以便可以返回他们

现在 `search` 函数应该返回只包含 `query` 的那些行，而测试应该会通过。让我们运行测试：

```
$ cargo test
running 1 test
test test::one_result ... ok

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

测试通过了，很好，它可以工作了！

现在测试通过了，我们可以考虑一下重构 `search` 的实现并时刻保持测试通过来保持其功能不变的机会了。`search` 函数中的代码并不坏，不过并没有利用迭代器的一些实用功能。第十三章将回到这个例子并深入探索迭代器并看看如何改进代码。

在 `run` 函数中使用 `search` 函数

现在 `search` 函数是可以工作并测试通过了的，我们需要实际在 `run` 函数中调用 `search`。需要将 `config.query` 值和 `run` 从文件中读取的 `contents` 传递给 `search` 函数。接着 `run` 会打印出 `search` 返回的每一行：

文件名: `src/lib.rs`

```
pub fn run(config: Config) -> Result<(), Box<Error>> {
    let mut f = File::open(config.filename)?;

    let mut contents = String::new();
    f.read_to_string(&mut contents)?;

    for line in search(&config.query, &contents) {
        println!("{}", line);
    }

    Ok(())
}
```

这里仍然使用了 `for` 循环获取了 `search` 返回的每一行并打印出来。

现在整个程序应该可以工作了！让我们试一试，首先使用一个只会在艾米莉·狄金森的诗中返回一行的单词 `"frog"`：

```
$ cargo run frog poem.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.38 secs
Running `target/debug/minigrep frog poem.txt`
How public, like a frog
```

好的！接下来，像 **"the"** 这样会匹配多行的单词会怎么样呢：

```
$ cargo run the poem.txt
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running `target/debug/minigrep the poem.txt`
Then there's a pair of us – don't tell!
To tell your name the livelong day
```

最后，让我们确保搜索一个在诗中哪里都没有的单词时不会得到任何行，比如 **"monomorphization"**：

```
$ cargo run monomorphization poem.txt
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running `target/debug/minigrep monomorphization poem.txt`
```

非常好！我们创建了一个属于自己的迷你版经典工具，并学习了很多如何组织程序的知识。我们还学习了一些文件输入输出、生命周期、测试和命令行解析的内容。

为了使这个项目章节更丰满，我们将简要的展示如何处理环境变量和打印到标准错误，这两者在编写命令行程序时都很有用。现在如果你希望的话请随意移动到第十三章。

处理环境变量

[ch12-05-working-with-environment-variables.md](#)

commit adababc48956f4d39c97c8b6fc14a104d90e20dc

我们将用一个额外的功能来改进我们的工具：一个通过环境变量启用的大小写不敏感搜索的选项。我们可以将其设计为一个命令行参数并要求用户每次需要时都加上它，不过相反我们将使用环境变量。这允许用户设置环境变量一次之后在整个终端会话中所有的搜索都将是大小写不敏感的。

编写一个大小写不敏感 `search` 函数的失败测试

首先，增加一个新函数，当设置了环境变量时会调用它。

这里将继续遵循上一部分开始使用的 TDD 过程，其第一步是再次编写一个失败测试。我们将为新的大小写不敏感搜索函数新增一个测试函数，并将老的测试函数从 `one_result` 改名为 `case_sensitive` 来更清楚的表明这两个测试的区别，如示例 12-20 所示：

文件名: `src/lib.rs`

```
#[cfg(test)]
mod test {
    use super::*;

    #[test]
    fn case_sensitive() {
        let query = "duct";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.
Duct tape.";

        assert_eq!(
            vec!["safe, fast, productive."],
            search(query, contents)
        );
    }

    #[test]
    fn case_insensitive() {
        let query = "rUsT";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.
Trust me.";

        assert_eq!(
            vec!["Rust:", "Trust me."],
            search_case_insensitive(query, contents)
        );
    }
}
```

示例 12-20：为准备添加的大小写不敏感函数新增失败测试

注意我们也改变了老测试中 `contents` 的值。还新增了一个含有文本 "Duct tape" 的行，它有一个大写的 D，这在大小写敏感搜索时不应该匹配 "duct"。我们修改这个测试以确保不会意外破坏已经实现的大小写敏感搜索功能；这个测试现在应该能通过并在处理大小写不敏感搜索时应该能一直通过。

大小写 不敏感 搜索的新测试使用 "rUsT" 作为其查询字符串。在我们将要增加的 `search_case_insensitive` 函数中，“rUsT” 查询应该包含 “Rust:” 包含一个大写的 R 还有 “Trust me.” 这两行，即便他们与查询的大小写都不同。这个测试现在会编译失败因为还没有定义 `search_case_insensitive` 函数。请随意增加一个总是返回空 `vector` 的骨架实现，正如示例 12-16 中 `search` 函数为了使测试编译并失败时所做的那样。

实现 `search_case_insensitive` 函数

`search_case_insensitive` 函数，如示例 12-21 所示，将与 `search` 函数基本相同。唯一的区别是它会将 `query` 变量和每一 `line` 都变为小写，这样不管输入参数是大写还是小写，在检查该行是否包含查询字符串时都会是小写。

文件名: `src/lib.rs`

```
fn search_case_insensitive<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let query = query.to_lowercase();
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.to_lowercase().contains(&query) {
            results.push(line);
        }
    }

    results
}
```

示例 12-21: 定义 `search_case_insensitive` 函数，它在比较查询和每一行之前将他们都转换为小写

首先我们将 `query` 字符串转换为小写，并将其覆盖到同名的变量中。对查询字符串调用 `to_lowercase` 是必需的，这样不管用户的查询是 “rust”、“RUST”、“Rust” 或者 “rUsT”，我们都将其当作 “rust” 处理并对大小写不敏感。

注意 `query` 现在是一个 `String` 而不是字符串 `slice`，因为调用 `to_lowercase` 是在创建新数据，而不是引用现有数据。如果查询字符串是 “rUsT”，这个字符串 `slice` 并不包含可供我们使用的小写的 “u” 或 “t”，所以必需分配一个包含 “rust” 的新 `String`。现在当我们将 `query` 作为一个参数传递给 `contains` 方法时，需要增加一个 `&` 因为 `contains` 的签名被定义为获取一个字符串 `slice`。

接下来在检查每个 `line` 是否包含 `search` 之前增加了一个 `to_lowercase` 调用将他们都变为小写。现在我们将 `line` 和 `query` 都转换成了小写，这样就可以不管查询的大小写进行匹配了。

让我们看看这个实现能否通过测试：

```
running 2 tests
test test::case_insensitive ... ok
test test::case_sensitive ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured
```

好的！现在，让我们在 `run` 函数中实际调用新 `search_case_insensitive` 函数。首先，我们将在 `Config` 结构体中增加一个配置项来切换大小写敏感和大小写不敏感搜索：

文件名: `src/lib.rs`

```
pub struct Config {
```

```
pub query: String,
pub filename: String,
pub case_sensitive: bool,
}
```

这里增加了 `case_sensitive` 字段来存放一个布尔值。接着我们需要 `run` 函数检查 `case_sensitive` 字段的值并使用它来决定是否调用 `search` 函数或 `search_case_insensitive` 函数，如示例 12-22 所示：

文件名: `src/lib.rs`

```
# use std::error::Error;
# use std::fs::File;
# use std::io::prelude::*;
#
# fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
#     vec![]
# }
#
# fn search_case_insensitive<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
#     vec![]
# }
#
# struct Config {
#     query: String,
#     filename: String,
#     case_sensitive: bool,
# }
#
pub fn run(config: Config) -> Result<(), Box<Error>>{
    let mut f = File::open(config.filename)?;

    let mut contents = String::new();
    f.read_to_string(&mut contents)?;

    let results = if config.case_sensitive {
        search(&config.query, &contents)
    } else {
        search_case_insensitive(&config.query, &contents)
    };

    for line in results {
        println!("{}", line);
    }

    Ok(())
}
```

示例 12-22：根据 `config.case_sensitive` 的值调用 `search` 或 `search_case_insensitive`

最后需要实际检查环境变量。处理环境变量的函数位于标准库的 `env` 模块中，所以我们需要在 `src/lib.rs` 的开头增加一个 `use std::env;` 行将这个模块引入作用域中。接着在 `Config::new` 中使用 `env` 模块的 `var` 方法来检查一个叫做 `CASE_INSENSITIVE` 的环境变量，如示例 12-23 所示：

文件名: `src/lib.rs`

```
use std::env;
# struct Config {
#     query: String,
#     filename: String,
#     case_sensitive: bool,
# }
```

```
// ...snip...

impl Config {
    pub fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let filename = args[2].clone();

        let case_sensitive = env::var("CASE_INSENSITIVE").is_err();

        Ok(Config { query, filename, case_sensitive })
    }
}
```

示例 12-23：检查叫做 `CASE_INSENSITIVE` 的环境变量

这里创建了一个新变量 `case_sensitive`。为了设置它的值，需要调用 `env::var` 函数并传递我们需要寻找的环境变量名称，`CASE_INSENSITIVE`。`env::var` 返回一个 `Result`，它在环境变量被设置时返回包含其值的 `Ok` 成员，并在环境变量未被设置时返回 `Err` 成员。

我们使用 `Result` 的 `is_err` 方法来检查其是否是一个 `error`（也就是环境变量未被设置的情况），这也就意味着我们需要进行一个大小写敏感搜索。如果 `CASE_INSENSITIVE` 环境变量被设置为任何值，`is_err` 会返回 `false` 并将进行大小写不敏感搜索。我们并不关心环境变量所设置的值，只关心它是否被设置了，所以检查 `is_err` 而不是 `unwrap`、`expect` 或任何我们已经见过的 `Result` 的方法。

我们将变量 `case_sensitive` 的值传递给 `Config` 实例，这样 `run` 函数可以读取其值并决定是否调用 `search` 或者示例 12-22 中实现的 `search_case_insensitive`。

让我们试一试吧！首先不设置环境变量并使用查询“to”运行程序，这应该会匹配任何全小写的单词“to”的行：

```
$ cargo run to poem.txt
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running `target/debug/minigrep to poem.txt`
Are you nobody, too?
How dreary to be somebody!
```

看起来程序仍然能够工作！现在将 `CASE_INSENSITIVE` 设置为 1 并仍使用相同的查询“to”，这回应该得到包含可能有 uppercase 的“to”的行：

```
$ CASE_INSENSITIVE=1 cargo run to poem.txt
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running `target/debug/minigrep to poem.txt`
Are you nobody, too?
How dreary to be somebody!
To tell your name the livelong day
To an admiring bog!
```

好极了，我们也得到了包含“To”的行！现在 `minigrep` 程序可以通过环境变量控制进行大小写不敏感搜索了。现在你知道了如何管理由命令行参数或环境变量设置的选项了！

一些程序允许对相同配置同时使用参数和环境变量。在这种情况下，程序来决定参数和环境变量的优先级。作为一个留给你的测试，尝试通过一个命令行参数或一个环境变量来控制大小写不敏感搜索。并在运行程序时遇到矛盾值时决定命令行参数和环境变量的优先级。

`std::env` 模块还包含了更多处理环境变量的实用功能；请查看官方文档来了解其可用的功能。

输出到 `stderr` 而不是 `stdout`

[ch12-06-writing-to-stderr-instead-of-stdout.md](#)

commit 7db14aa689553706198ffcb11a8c60b478e752fe

目前为止，我们将所有的输出都 `println!` 到了终端。大部分终端都提供了两种输出：标准输出（*standard output*）对应大部分信息（有时在代码中使用缩写 `stdout`），标准错误（*standard error*）则用于错误信息（`stderr`）。这种区别允许用户选择将程序正常输出定向到一个文件中并仍将错误信息打印到屏幕上。

但是 `println!` 函数只能够打印到标准输出，所以我们必需使用其他方法来打印到标准错误。

检查错误应该写入何处

首先，让我们观察一下目前 `minigrep` 打印的所有内容都被写入了标准输出，包括应该被写入标准错误的错误信息。可以通过将标准输出流重定向到一个文件同时有意产生一个错误来做到这一点。我们没有重定向标准错误流，所以任何发送到标准错误的内容将会继续显示在屏幕上。命令行程序被期望将错误信息发送到标准错误流，这样即便选择将标准输出流重定向到文件中时仍然能看到错误信息。目前我们的程序并不符合期望；我们将看到相反它将错误信息输出保存到了文件中。

展示这种行为的方式是通过 `>` 和文件名 `output.txt` 来与运行程序，这个文件是期望重定向标准输出流的位置。并不传递任何参数这样应该会产生一个错误：

```
$ cargo run > output.txt
```

`>` 语法告诉 `shell` 将标准输出的内容写入到 `output.txt` 文件中而不是屏幕上。我们并没有看到期望的错误信息打印到屏幕上，所以这意味着它一定被写入了文件中。让我们看看 `output.txt` 包含什么：

```
Problem parsing arguments: not enough arguments
```

是的，错误信息被打印到了标准输出中。像这样的错误信息被打印到标准错误中将有多得多，并在重定向标准输出时只将成功运行的信息写入文件。我们将改变他们。

将错误打印到标准错误

让我们如示例 12-24 所示的代码改变错误信息是如何被打印的。得益于本章早些时候的重构，所有打印错误信息的代码都位于 `main` 一个函数中。标准库提供了 `eprintln!` 宏来打印到标准错误流，所以将两个调用 `println!` 打印错误信息的维持替换为 `eprintln!`：

文件名: `src/main.rs`

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

    let config = Config::new(&args).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    if let Err(e) = minigrep::run(config) {
        eprintln!("Application error: {}", e);

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



```
}  
}
```

示例 12-24: 使用 `eprintln!` 将错误信息写入标准错误而不是标准输出

将 `println!` 改为 `eprintln!` 之后, 让我们再次尝试用同样的方式运行程序, 不使用任何参数并通过 `>` 重定向标准输出:

```
$ cargo run > output.txt  
Problem parsing arguments: not enough arguments
```

现在看到了屏幕上的错误信息, 同时 `output.txt` 里什么也没有, 这也就是命令程序所期望的行为。

如果使用不会造成错误的参数再次运行程序, 不过仍然将标准输出重定向到一个文件:

```
$ cargo run to poem.txt > output.txt
```

我们并不会在终端看到任何输出, 同时 `output.txt` 将会包含其结果:

文件名: `output.txt`

```
Are you nobody, too?  
How dreary to be somebody!
```

这一部分展示了现在我们适当的使用成功时产生的标准输出和错误时产生的标准错误。

总结

在这一章中, 我们回顾了目前为止的一些主要章节并涉及了如何在 **Rust** 环境中进行常规的 I/O 操作。通过使用命令行参数、文件、环境变量和打印错误的 `eprintln!` 宏, 现在你已经准备好编写命令程序了。通过结合前几章的知识, 你的代码将会是组织良好的, 并能有效的将数据存储到合适的数据结构中、更好的处理错误, 并且还是经过良好测试的。

接下来, 让我们探索如何利用一些 **Rust** 中受函数式编程语言影响的功能: 闭包和迭代器。

Rust 中的函数式语言功能 —— 迭代器和闭包

[ch13-00-functional-features.md](#)

commit d06a6a181fd61704cbf7feb55bc61d518c6469f9

Rust 的设计灵感来源于很多现存的语言和技术。其中一个显著的影响就是 函数式编程（*functional programming*）。函数式编程风格通常包含将函数作为参数值或其他函数的返回值、将函数赋值给变量以供之后执行等等。我们不会在这里讨论函数式编程是或不是什么问题，而是展示 Rust 的一些在功能上与其他语言类似的特性。

更具体的，我们将要涉及：

- 闭包（*Closures*），一个可以储存在变量里的类似函数的结构。
- 迭代器（*Iterators*），一种处理元素序列的方式。。
- 如何使用这些功能来改进第十二章的 I/O 项目。
- 这些功能的性能。剧透高能： 他们的速度超乎你的想象！

还有其它受函数式风格影响的 Rust 功能，比如模式匹配和枚举，这些已经在其他章节中讲到过了。掌握闭包和迭代器则是编写符合语言风格的高性能 Rust 代码的重要一环，所以我们将专门用一整章来讲解他们。

闭包：可以捕获环境的匿名函数

ch13-01-closures.md

commit 56352c28cf3fe0402fa5a7cba73890e314d720eb

Rust 的闭包是可以保存进变量或作为参数传递给其他函数的匿名函数。可以在一个地方创建闭包，然后在以不同的上下文中执行闭包运算。不同于函数，闭包允许捕获调用者作用域中的值。我们将展示闭包的这些功能如何复用代码和自定义行为。

使用闭包创建行为的抽象

让我们看看一个展示储存闭包并在之后执行如何有用的情形的例子。其间我们会讨论闭包的语法、类型推断和 trait。

这个假想的情况如下：我们在一个通过 **app** 生成自定义健身计划的初创企业工作。其后端使用 Rust 编写，而生成健身计划的算法需要考虑很多不同的因素，比如用户的年龄、身体质量指数（Body Mass Index）、用户喜好、最近的健身活动和用户指定的强度系数。本例中实际的算法并不重要。我们只希望在需要时调用算法，并且只希望调用一次，这样就不会让用户等得太久。这里将通过调用 `simulated_expensive_calculation` 函数来模拟调用假象的算法，如示例 13-1 所示，它会打印出 `calculating slowly...`，等待两秒，并接着返回传递给它的数字：

文件名: src/main.rs

```
use std::thread;
use std::time::Duration;

fn simulated_expensive_calculation(intensity: i32) -> i32 {
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    intensity
}
```

示例 13-1：一个用来代替假象计算的函数，它大约会执行两秒

接下来，`main` 函数中将会包含本例的健身 **app** 中的重要部分。这代表当用户请求健身计划时 **app** 会调用的代码。因为与 **app** 前端的交互与闭包的使用并不相关，所以我们将硬编码代表程序输入的值并打印输出。

程序的输入有：

- 一个来自用户的 `intensity` 数字，请求健身计划时指定，它代表用户喜好低强度还是高强度健身
- 一个会在健身计划中生成变化的随机数

程序的输出将会是建议的锻炼计划。

示例 13-2 展示了我们将要使用的 `main` 函数。处于简单考虑这里硬编码了 `simulated_user_specified_value` 变量的值为 10 和 `simulated_random_number` 变量的值为 7；一个实际的程序会从 **app** 前端获取强度系数并使用 `rand crate` 来生成随机数，正如第二章的猜猜看游戏所做的那样。`main` 函数使用模拟的输入值调用 `generate_workout` 函数：

文件名: src/main.rs

```
fn main() {
    let simulated_user_specified_value = 10;
    let simulated_random_number = 7;

    generate_workout(simulated_user_specified_value, simulated_random_number);
}
```

```
# fn generate_workout(intensity: i32, random_number: i32) {}
```

示例 13-2: `main` 函数包含了用于 `generate_workout` 函数的模拟用户输入和模拟随机数输入

这就是我们的执行上下文。示例 13-3 中的 `generate_workout` 函数包含我们最关心的 `app` 业务逻辑。本例中余下的代码修改都将在这个函数中:

文件名: `src/main.rs`

```
# use std::thread;
# use std::time::Duration;
#
# fn simulated_expensive_calculation(num: i32) -> i32 {
#     println!("calculating slowly...");
#     thread::sleep(Duration::from_secs(2));
#     num
# }
#
fn generate_workout(intensity: i32, random_number: i32) {
    if intensity < 25 {
        println!(
            "Today, do {} pushups!",
            simulated_expensive_calculation(intensity)
        );
        println!(
            "Next, do {} situps!",
            simulated_expensive_calculation(intensity)
        );
    } else {
        if random_number == 3 {
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            println!(
                "Today, run for {} minutes!",
                simulated_expensive_calculation(intensity)
            );
        }
    }
}
```

示例 13-3: 程序的业务逻辑, 它根据输入并调用 `simulated_expensive_calculation` 函数来打印出健身计划

示例 13-3 中的代码有多处慢计算函数的调用。第一个 `if` 块调用了 `simulated_expensive_calculation` 两次, 外部 `else` 中的 `if` 完全没有调用它, `else` 中的 `else` 调用了它一次。

`generate_workout` 函数的期望行为是首先检查用户需要低强度(由小于 25 的系数表示)锻炼还是高强度(25 或以上)锻炼。低强度锻炼计划会根据由 `simulated_expensive_calculation` 函数所模拟的复杂算法建议一定数量的俯卧撑和仰卧起坐, 此函数需要强度系数作为输入。

如果用户需要高强度锻炼, 这里有一些额外的逻辑: 如果 `app` 生成的随机数刚好是 3, `app` 相反会建议用户稍做休息并补充水分。如果不是, 则用户会从复杂算法中得到数分钟跑步的高强度锻炼计划。

数据科学部门的同学告知我们将来会对调用算法的方式做出一些改变。为了在要做这些改动的时候简化更新步骤, 我们将重构代码来让它只调用 `simulated_expensive_calculation` 一次。同时还希望去掉目前多余的连续两次函数调用, 并不希望在计算过程中增加任何其他此函数的调用。也就是说, 我们不需要在完全无需其结果的情况调用函数, 不过最终仍然需要调用函数一次。

有多种方法可以重构此程序。我们首先尝试的是将重复的慢计算函数调用提取到一个变量中, 如示例 13-4 所示:

文件名: `src/main.rs`

```

# use std::thread;
# use std::time::Duration;
#
# fn simulated_expensive_calculation(num: i32) -> i32 {
#     println!("calculating slowly...");
#     thread::sleep(Duration::from_secs(2));
#     num
# }
#
fn generate_workout(intensity: i32, random_number: i32) {
    let expensive_result =
        simulated_expensive_calculation(intensity);

    if intensity < 25 {
        println!(
            "Today, do {} pushups!",
            expensive_result
        );
        println!(
            "Next, do {} situps!",
            expensive_result
        );
    } else {
        if random_number == 3 {
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            println!(
                "Today, run for {} minutes!",
                expensive_result
            )
        }
    }
}

```

示例 13-4：将 `simulated_expensive_calculation` 调用提取到一个位置，位于 `if` 块之前并将结果储存在变量 `expensive_result` 中

这个修改统一了 `simulated_expensive_calculation` 调用并解决了第一个 `if` 块中不必要的两次调用函数的问题。不幸的是，现在所有的情况下都需要调用函数并等待结果，包括那个完全不需要这一结果的内部 `if` 块。

我们希望能够程序的一个位置指定某些代码，并只在程序的某处实际需要结果的时候执行这些代码。这正是闭包的用武之地！

闭包储存了之后会执行的代码

不同于总是在 `if` 块之前调用 `simulated_expensive_calculation` 函数并储存其结果，我们可以定义一个闭包并将其储存在变量中，如示例 13-5 所示。实际上可以选择将整个 `simulated_expensive_calculation` 函数体移动到这里引入的闭包中：

文件名: `src/main.rs`

```

# use std::thread;
# use std::time::Duration;
#
let expensive_closure = |num| {
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    num
};
# expensive_closure(5);

```

示例 13-5：使用慢计算的函数体定义一个闭包并储存到变量 `expensive_closure` 中

闭包定义是 `expensive_closure` 赋值的 `=` 之后的部分。闭包的定义以一对竖线（`|`）开始。在竖线中指定闭包的参数；之所以选择这个语法是因为它与 **Smalltalk** 和 **Ruby** 的闭包定义类似。这个闭包有一个参数 `num`；如果多于一个参数，可以使用逗号分隔，比如 `|param1, param2|`。

参数之后是存放闭包体的大括号。如果闭包体只有一行则大括号是可以省略的。大括号之后，需要用于 `let` 语句的分号。闭包体的最后一行（`num`）返回的值将是调用闭包时返回的值，因为最后一行没有分号，正如函数体中的一样。

注意这个 `let` 语句意味着 `expensive_closure` 包含一个匿名函数的定义，不是调用匿名函数的返回值。回忆一下使用闭包的原因是我们需要在一个位置定义代码，储存代码，并在之后的位置实际调用它；期望调用的代码现在储存在 `expensive_closure` 中。

现在我们定义了闭包，可以改变 `if` 块中的代码来调用闭包以执行代码并获取结果值。调用闭包看起来非常类似调用函数；指定存放闭包定义的变量名并后跟包含期望使用的参数的括号，如示例 13-6 所示：

文件名: `src/main.rs`

```
# use std::thread;
# use std::time::Duration;
#
fn generate_workout(intensity: i32, random_number: i32) {
    let expensive_closure = |num| {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        num
    };

    if intensity < 25 {
        println!(
            "Today, do {} pushups!",
            expensive_closure(intensity)
        );
        println!(
            "Next, do {} situps!",
            expensive_closure(intensity)
        );
    } else {
        if random_number == 3 {
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            println!(
                "Today, run for {} minutes!",
                expensive_closure(intensity)
            )
        }
    }
}
```

示例 13-6：调用定义的 `expensive_closure`

现在我们达成了将慢计算统一到一个地方的目标，并只会在需要结果的时候执行改代码。然而，我们又重新引入了示例 13-3 中的问题：仍然在第一个 `if` 块中调用了闭包两次，这会调用慢计算两次并使用户多等待一倍的时间。可以通过在 `if` 块中创建一个本地变量存放闭包调用的结果来解决这个问题，不过正因为使用了闭包还有另一个解决方案。稍后会回到这个方案上；首先讨论一下为何闭包定义中和所涉及的 `trait` 中没有类型注解。

闭包类型推断和注解

闭包与由 `fn` 关键字定义的函数有一些区别。第一是不要求像 `fn` 函数那样在参数和返回值上注明类型。

函数中需要类型注解是因为他们是暴露给用户的显式接口的一部分。严格的定义这些接口对于保证所有人都认同函数使用和返回值的类型来说是很重要的。但是闭包并不用于这样暴露在外的接口：他们储存在变量中并被使用，不用命名他们或暴露给库的用户调用。

另外，闭包通常很短并只与对应相对任意的场景较小的上下文中。在这些有限制的上下文中，编译器能可靠的推断参数和返回值的类型，类似于它是如何能够推断大部分变量的类型一样。强制在这些小的匿名函数中注明类型是很恼人的，并且与编译器已知信息存在大量的重复。

类似于变量，如果相比严格的必要性你更希望增加明确性并变得更啰嗦，可以选择增加类型注解；为示例 13-4 中定义的闭包标注类型将看起来像示例 13-7 中的定义：

文件名: `src/main.rs`

```
# use std::thread;
# use std::time::Duration;
#
let expensive_closure = |num: i32| -> i32 {
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    num
};
```

示例 13-7：为闭包的参数和返回值增加可选的类型注解

有了类型注解闭包的语法就更类似函数了。如下是一个对其参数加一的函数的定义与拥有相同行为闭包语法的纵向对比。这里增加了一些空格来对齐相应部分。这展示了闭包语法如何类似于函数语法，除了使用竖线而不是括号以及几个可选的语法：

```
fn add_one_v1   (x: i32) -> i32 { x + 1 }
let add_one_v2 = |x: i32| -> i32 { x + 1 };
let add_one_v3 = |x|           { x + 1 };
let add_one_v4 = |x|           x + 1 ;
```

第一行展示了一个函数定义，而第二行展示了一个完整标注的闭包定义。第三行闭包定义中省略了类型注解，而第四行去掉了可选的大括号，因为闭包体只有一行。

闭包定义会为每个参数和返回值推断一个具体类型。例如，示例 13-8 中展示了仅仅将参数作为返回值的简短的闭包定义。除了作为示例的目的这个闭包并不是很实用。注意其定义并没有增加任何类型注解：如果尝试调用闭包两次，第一次使用 `String` 类型作为参数而第二次使用 `i32`，则会得到一个错误：

文件名: `src/main.rs`

```
let example_closure = |x| x;

let s = example_closure(String::from("hello"));
let n = example_closure(5);
```

示例 13-8：尝试调用一个被推断为两个不同类型的闭包

编译器给出如下错误：

```
error[E0308]: mismatched types
--> src/main.rs
|
| let n = example_closure(5);
```

```
|
    ^ expected struct `std::string::String`, found
integral variable
|
= note: expected type `std::string::String`
       found type `{integer}`
```

第一次使用 `String` 值调用 `example_closure` 时，编译器推断 `x` 和此闭包返回值的类型为 `String`。接着这些类型被锁定进闭包 `example_closure` 中，如果尝试对同一闭包使用不同类型则会得到类型错误。

使用带有泛型和 `Fn trait` 的闭包

回到我们的健身计划生成 `app`，在示例 13-6 中的代码仍然调用了多于需要的慢计算闭包。在全部代码中的每一个需要多个慢计算闭包结果的地方，可以将结果保存进变量以供复用，这样就可以使用变量而不是再次调用闭包。但是这样就会有重复的保存结果变量的地方。

然而，因为拥有一个慢计算的闭包，我们还可以采取另一个解决方案。可以创建一个存放闭包和调用闭包结果的结构体。该结构体只会在需要结果时执行闭包，并会缓存结果值，这样余下的代码就不必再负责保存结果并可以复用该值。你可能见过这种模式被称 *memoization* 或 *lazy evaluation*。

为了让结构体存放闭包，我们需要能够指定闭包的类型。每一个闭包实例有其自己独有的匿名类型：也就是说，即便两个闭包有着相同的签名，他们的类型仍然可以被认为是不同。为了定义使用闭包的结构体、枚举或函数参数，需要像第十章讨论的那样使用泛型和 `trait bound`。

`Fn` 系列 `trait` 由标准库提供。所有的闭包都实现了 `trait Fn`、`FnMut` 或 `FnOnce` 中的一个。在下一部分捕获环境部分我们会讨论这些 `trait` 的区别；在这个例子中可以使用 `Fn trait`。

为了满足 `Fn trait bound` 我们增加了代表闭包所必须的参数和返回值类型的类型。在这个例子中，闭包有一个 `i32` 的参数并返回一个 `i32`，这样所指定的 `trait bound` 就是 `Fn(i32) -> i32`。

示例 13-9 展示了存放了闭包和一个 `Option` 结果值的 `Cacher` 结构体的定义：

文件名: `src/main.rs`

```
struct Cacher<T>
where T: Fn(i32) -> i32
{
    calculation: T,
    value: Option<i32>,
}
```

示例 13-9：定义一个 `Cacher` 结构体来在 `calculation` 中存放闭包并在 `value` 中存放 `Option` 值

结构体 `Cacher` 有一个泛型 `T` 的字段 `calculation`。`T` 的 `trait bound` 指定了 `T` 是一个使用 `Fn` 的闭包。任何我们希望储存在 `Cacher` 实例的 `calculation` 字段的闭包必须有一个 `i32` 参数（由 `Fn` 之后的括号的内容指定）并必须返回一个 `i32`（由 `->` 之后的内容）。

`value` 是 `Option<i32>` 类型的。在执行闭包之前，`value` 将是 `None`。如果使用 `Cacher` 的代码请求闭包的结果，这时会执行闭包并将结果储存在 `value` 字段的 `Some` 成员中。接着如果代码再次请求闭包的结果，这时不再执行闭包，而是会返回存放在 `Some` 成员中的结果。

刚才讨论的有关 `value` 字段逻辑定义于示例 13-10：

文件名: `src/main.rs`

```
# struct Cacher<T>
# where T: Fn(i32) -> i32
# {
```



```

#     calculation: T,
#     value: Option<i32>,
# }
#
impl<T> Cacher<T>
    where T: Fn(i32) -> i32
{
    fn new(calculation: T) -> Cacher<T> {
        Cacher {
            calculation,
            value: None,
        }
    }

    fn value(&mut self, arg: i32) -> i32 {
        match self.value {
            Some(v) => v,
            None => {
                let v = (self.calculation)(arg);
                self.value = Some(v);
                v
            },
        }
    }
}

```

示例 13-10：一个 `Cacher` 的关联函数 `new` 和管理缓存逻辑的 `value` 方法的实现

`Cacher` 结构体的字段是私有的，因为我们希望 `Cacher` 管理这些值而不是任由调用代码潜在的直接改变他们。`Cacher::new` 函数获取一个泛型参数 `T`，它定义于 `impl` 块上下文中并与 `Cacher` 结构体有着相同的 **trait bound**。`Cacher::new` 返回一个在 `calculation` 字段中存放了指定闭包和在 `value` 字段中存放了 `None` 值的 `Cacher` 实例，因为我们还未执行闭包。

当调用代码需要闭包的执行结果时，不同于直接调用闭包，它会调用 `value` 方法。这个方法会检查 `self.value` 是否已经有了一个 `Some` 的结果值；如果有，它返回 `Some` 中的值并不会再次执行闭包。

如果 `self.value` 是 `None`，则会调用 `self.calculation` 中储存的闭包，将结果保存到 `self.value` 以便将来使用，并同时返回结果值。

示例 13-11 展示了如何在示例 13-6 的 `generate_workout` 函数中利用 `Cacher` 结构体：

文件名: `src/main.rs`

```

# use std::thread;
# use std::time::Duration;
#
# struct Cacher<T>
#     where T: Fn(i32) -> i32
# {
#     calculation: T,
#     value: Option<i32>,
# }
#
# impl<T> Cacher<T>
#     where T: Fn(i32) -> i32
# {
#     fn new(calculation: T) -> Cacher<T> {
#         Cacher {
#             calculation,
#             value: None,
#         }
#     }
# }

```

```

#
# fn value(&mut self, arg: i32) -> i32 {
#     match self.value {
#         Some(v) => v,
#         None => {
#             let v = (self.calculation)(arg);
#             self.value = Some(v);
#             v
#         },
#     }
# }
# }
#
fn generate_workout(intensity: i32, random_number: i32) {
    let mut expensive_result = Cacher::new(|num| {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        num
    });

    if intensity < 25 {
        println!(
            "Today, do {} pushups!",
            expensive_result.value(intensity)
        );
        println!(
            "Next, do {} situps!",
            expensive_result.value(intensity)
        );
    } else {
        if random_number == 3 {
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            println!(
                "Today, run for {} minutes!",
                expensive_result.value(intensity)
            )
        }
    }
}

```

示例 13-11: 在 `generate_workout` 函数中利用 `Cacher` 结构体来抽象出缓存逻辑

不同于直接将闭包保存进一个变量，我们保存一个新的 `Cacher` 实例来存放闭包。接着，在每一个需要结果的地方，调用 `Cacher` 实例的 `value` 方法。可以调用 `value` 方法任意多次，或者一次也不调用，而慢计算最多只会运行一次。尝试使用示例 13-2 中的 `main` 函数来运行这段程序，并将 `simulated_user_specified_value` 和 `simulated_random_number` 变量中的值来验证在所有情况下在多个 `if` 和 `else` 块中，闭包打印的 `calculating slowly...` 只会在需要时出现并只会出现一次。

`Cacher` 负责确保不会调用超过所需的慢计算所需的逻辑，这样 `generate_workout` 就可以专注业务逻辑了。值缓存是一种更加广泛的实用行为，我们可能希望在代码中的其他闭包中也使用他们。然而，目前 `Cacher` 的实现存在一些小问题，这使得在不同上下文中复用变得很困难。

第一个问题是 `Cacher` 实例假设对于 `value` 方法的任何 `arg` 参数值总是会返回相同的值。也就是说，这个 `Cacher` 的测试会失败：

```

#[test]
fn call_with_different_values() {
    let mut c = Cacher::new(|a| a);

    let v1 = c.value(1);
    let v2 = c.value(2);
}

```

```
    assert_eq!(v2, 2);
}
```

这个测试使用返回传递给它的值的闭包创建了一个新的 `Cacher` 实例。使用为 1 的 `arg` 和为 2 的 `arg` 调用 `Cacher` 实例的 `value` 方法，同时我们期望使用为 2 的 `arg` 调用 `value` 会返回 2。

使用示例 13-9 和示例 13-10 的 `Cacher` 实现运行测试，它会在 `assert_eq!` 失败并显示如下信息：

```
thread 'call_with_different_arg_values' panicked at 'assertion failed:
  `(left == right)` (left: `1`, right: `2`)', src/main.rs
```

这里的问题是第一次使用 1 调用 `c.value`，`Cacher` 实例将 `Some(1)` 保存进 `self.value`。在这之后，无论传递什么值调用 `value`，它总是会返回 1。

尝试修改 `Cacher` 存放一个哈希 `map` 而不是单独一个值。哈希 `map` 的 `key` 将是传递进来的 `arg` 值，而 `value` 则是对应 `key` 调用闭包的结果值。相比之前检查 `self.value` 是 `Some` 还是 `None` 值，现在 `value` 会在哈希 `map` 中寻找 `arg`，如果存在就返回它。如果不存在，`Cacher` 会调用闭包并将结果值保存在哈希 `map` 对应 `arg` 值的位置。

当前 `Cacher` 实现的另一个问题是它的应用被限制为只接受获取一个 `i32` 值并返回一个 `i32` 值的闭包。比如说，我们可能需要能够缓存一个获取字符串 `slice` 并返回 `usize` 值的闭包的结果。请尝试引入更多泛型参数来增加 `Cacher` 功能的灵活性。

闭包会捕获其环境

在健身计划生成器的例子中，我们只将闭包作为内联匿名函数来使用。不过闭包还有另一个函数所没有的功能：他们可以捕获其环境并访问定义他们的作用域的变量。

示例 13-12 有一个储存在 `equal_to_x` 变量中闭包的例子，它使用了闭包环境中的变量 `x`：

文件名: `src/main.rs`

```
fn main() {
    let x = 4;

    let equal_to_x = |z| z == x;

    let y = 4;

    assert!(equal_to_x(y));
}
```

示例 13-12：一个引用了其周围作用域中变量的闭包示例

这里，即便 `x` 并不是 `equal_to_x` 的一个参数，`equal_to_x` 闭包也被允许使用变量 `x`，因为它与 `equal_to_x` 定义于相同的作用域。

函数则不能做到同样的事，让我们试试会发生什么：

文件名: `src/main.rs`

```
fn main() {
    let x = 4;

    fn equal_to_x(z: i32) -> bool { z == x }
```

```

    let y = 4;

    assert!(equal_to_x(y));
}

```

这会得到一个错误：

```

error[E0434]: can't capture dynamic environment in a fn item; use the || { ... }
closure form instead
-->
|
4 |     fn equal_to_x(z: i32) -> bool { z == x }
|                                     ^

```

编译器甚至会提示我们这只能用于闭包！

当闭包从环境中捕获一个值，闭包会在闭包体中储存这个值以供使用。这会使用内存并产生额外的开销，当执行不会捕获环境的更通用的代码场景中我们不希望有这些开销。因为函数从未允许捕获环境，定义和使用函数也就从不会有这些额外开销。

闭包可以通过三种方式捕获其环境，他们直接对应函数的三种获取参数的方式：获取所有权，不可变借用和可变借用。这三种捕获值的方式被编码为如下三个 `Fn trait`：

- `FnOnce` 消费从周围作用域捕获的变量（闭包周围的作用域被称为其环境，*environment*）。为了消费捕获到的变量，闭包必须获取其所有权并在定义闭包时将其移动进闭包。其名称的 `Once` 部分是因为闭包不能多次获取相同变量的所有权，所以它只能被调用一次。
- `Fn` 从其环境不可变的借用值
- `FnMut` 可变的借用值所以可以改变其环境

当创建一个闭包时，`Rust` 根据其如何使用环境中变量来推断我们希望如何引用环境。在示例 13-12 中，`equal_to_x` 闭包不可变的借用了 `x`（所以 `equal_to_x` 使用 `Fn trait`），因为闭包体只需要读取 `x` 的值。

如果我们希望强制闭包获取其使用的环境值的所有权，可以在参数列表前使用 `move` 关键字。这在将闭包传递给新线程以便将数据移动到新线程中时最为实用。第十六章讨论并发时会展示更多 `move` 闭包的例子，不过现在这里修改了示例 13-12 中的代码（作为演示），在闭包定义中增加 `move` 关键字并使用 `vector` 代替整型，因为整型可以被拷贝而不是移动：

文件名: `src/main.rs`

```

fn main() {
    let x = vec![1, 2, 3];

    let equal_to_x = move |z| z == x;

    println!("can't use x here: {:?}", x);

    let y = vec![1, 2, 3];

    assert!(equal_to_x(y));
}

```

这个例子并不能编译：

```

error[E0382]: use of moved value: `x`
--> src/main.rs:6:40
|
4 |     let equal_to_x = move |z| z == x;
|

```

```

|           ----- value moved (into closure) here
5 |
6 |     println!("can't use x here: {:?}", x);
|                                     ^ value used here after move
|
= note: move occurs because `x` has type `std::vec::Vec<i32>`, which does not
       implement the `Copy` trait

```

`x` 被移动进了闭包，因为闭包使用 `move` 关键字定义。接着闭包获取了 `x` 的所有权，同时 `main` 就不再允许使用它了。去掉 `println!` 即可修复问题。

大部分需要指定一个 `Fn` **trait bound** 的时候，可以从 `Fn` 开始，编译器会根据闭包体中的情况告诉你是否需要 `FnMut` 或 `FnOnce`。

为了展示闭包作为函数参数时捕获其环境的作用，让我们移动到下一个主题：迭代器。

迭代器

ch13-02-iterators.md

commit 40910f557c328858f230123d1234c1cb3029dda3

迭代器模式允许你对一个项的序列进行某些处理。迭代器（*iterator*）负责遍历序列中的每一项和决定序列何时结束的逻辑。当使用迭代器时，我们无需重新实现这些逻辑。

在 **Rust** 中，迭代器是惰性的（*lazy*），这意味着直到调用方法消费迭代器之前它都不会有效果。例如，示例 13-13 中的代码通过调用定义于 `Vec` 上的 `iter` 方法在一个 `vector` `v1` 上创建了一个迭代器。这段代码本身没有任何用处：

```
let v1 = vec![1, 2, 3];

let v1_iter = v1.iter();
```

示例 13-13：创建了迭代器

创建迭代器之后，可以选择用多种方式利用它。在示例 3-6 中，我们实际上使用了迭代器和 `for` 循环在每一个项上执行了一些代码，直到现在我们才解释了 `iter` 调用做了什么。示例 13-14 中的例子将迭代器的创建和 `for` 循环中的使用分开。迭代器被储存在 `v1_iter` 变量中，而这时没有进行迭代。一旦 `for` 循环开始使用 `v1_iter`，接着迭代器中的每一个元素被用于循环的一次迭代，这会打印出其每一个值：

```
let v1 = vec![1, 2, 3];

let v1_iter = v1.iter();

for val in v1_iter {
    println!("Got: {}", val);
}
```

示例 13-14：在一个 `for` 循环中使用迭代器

在标准库中没有提供迭代器的语言中，我们可能会使用一个从 0 开始的索引变量，使用这个变量索引 `vector` 中的值，并循环增加其值直到达到 `vector` 的元素数量。迭代器为我们处理了所有这些逻辑，这减少了重复代码并潜在的消除了混乱。另外，迭代器的实现方式提供了对多种不同的序列使用相同逻辑的灵活性，而不仅仅是像 `vector` 这样可索引的数据结构。让我们看看迭代器是如何做到这些的。

Iterator trait 和 next 方法

迭代器都实现了一个叫做 `Iterator` 的定义于标准库的 `trait`。这个 `trait` 的定义看起来像：

```
trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;

    // methods with default implementations elided
}
```

这里有一下我们还未讲到的新语法：`type Item` 和 `Self::Item`，他们定义了 `trait` 的关联类型（*associated type*）。第十九章会深入讲解关联类型，不过现在只需知道这段代码表明实现 `Iterator trait` 要求同时定义一个 `Item` 类型，这个 `Item` 类型被用作 `next` 方法的返回值类型。换句话说，`Item` 类型将是迭代器返回元素的类型。

`next` 是 `Iterator` 实现者被要求定义的唯一方法。`next` 一次返回迭代器中的一个项，封装在 `Some` 中，当迭代器结束时，它返回 `None`。如果你希望的话可以直接调用迭代器的 `next` 方法；示例 13-15 有一个测试展示了重复调用由 `vector` 创建的迭代器的 `next` 方法所得到的值：

文件名: `src/lib.rs`

```
#[test]
fn iterator_demonstration() {
    let v1 = vec![1, 2, 3];

    let mut v1_iter = v1.iter();

    assert_eq!(v1_iter.next(), Some(&1));
    assert_eq!(v1_iter.next(), Some(&2));
    assert_eq!(v1_iter.next(), Some(&3));
    assert_eq!(v1_iter.next(), None);
}
```

示例 13-15：在迭代器上（直接）调用 `next` 方法

注意 `v1_iter` 需要是可变的：在迭代器上调用 `next` 方法改变了迭代器中用来记录序列位置的状态。换句话说，代码消费（consume）了，或使用了迭代器。每一个 `next` 调用都会从迭代器中吃掉一个项。使用 `for` 循环时无需使 `v1_iter` 可变因为 `for` 循环会获取 `v1_iter` 的所有权并在后台使 `v1_iter` 可变。

另外需要注意到从 `next` 调用中得到的值是 `vector` 的不可变引用。`iter` 方法生成一个不可变引用的迭代器。如果我们需要一个获取 `v1` 所有权并返回拥有所有权的迭代器，则可以调用 `into_iter` 而不是 `iter`。类似的，如果我们希望迭代可变引用，则可以调用 `iter_mut` 而不是 `iter`。

Iterator trait 中消费迭代器的方法

`Iterator trait` 有一系列不同的由标准库提供默认实现的方法；你可以在 `Iterator trait` 的标准库 API 文档中找到所有这些方法。一些方法在其定义中调用了 `next` 方法，这也就是为什么在实现 `Iterator trait` 时要求实现 `next` 方法的原因。

这些调用 `next` 方法的方法被称为消费适配器（*consuming adapters*），因为调用他们会消耗迭代器。一个消费适配器的例子是 `sum` 方法。这个方法获取迭代器的所有权并反复调用 `next` 来遍历迭代器，因而会消费迭代器。当其遍历每一个项时，它将每一个项加总到一个总和并在迭代完成时返回总和。示例 13-16 有一个展示 `sum` 方法使用的测试：

文件名: `src/lib.rs`

```
#[test]
fn iterator_sum() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    let total: i32 = v1_iter.sum();

    assert_eq!(total, 6);
}
```

示例 13-16: 调用 `sum` 方法获取迭代器所有项的总和

调用 `sum` 之后不再允许使用 `v1_iter` 因为调用 `sum` 时它会获取迭代器的所有权。

Iterator trait 中产生其他迭代器的方法

`Iterator trait` 中定义的另一类方法会产生其他的迭代器。这些方法被称为 迭代器适配器 (*iterator adaptors*)，他们允许我们将当前迭代器变为不同类型的迭代器。可以链式调用多个迭代器适配器。不过因为所有的迭代器都是惰性的，必须调用一个消费适配器方法以便获取迭代器适配器调用的结果。示例 13-17 展示了一个调用迭代器适配器方法 `map` 的例子，该 `map` 方法使用闭包来调用每个元素以生成新的迭代器。这里的闭包创建了一个新的迭代器，其中 `vector` 中的每个元素都被加1。不过这些代码会产生一个警告：

文件名: `src/main.rs`

```
let v1: Vec<i32> = vec![1, 2, 3];

v1.iter().map(|x| x + 1);
```

示例 13-17: 调用迭代器适配器 `map` 来创建一个新迭代器

得到的警告是：

```
warning: unused result which must be used: iterator adaptors are lazy and do
nothing unless consumed
--> src/main.rs:4:1
 |
4 | v1.iter().map(|x| x + 1);
 | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
 |
 = note: #[warn(unused_must_use)] on by default
```

示例 13-17 中的代码实际上并没有做任何事；所指定的闭包从未被调用过。警告提醒了我们为什么：迭代器适配器是惰性的，而这里我们可能意在消费迭代器。

为了修复这个警告并消费迭代器获取有用的结果，我们将使用第十二章简要讲到的 `collect` 方法。这个方法消费迭代器并将结果收集到一个数据结构中。在示例 13-18 中，我们将遍历由 `map` 调用生成的迭代器的结果收集到一个 `vector` 中，它将会含有原始 `vector` 中每个元素加一的结果：

文件名: `src/main.rs`

```
let v1: Vec<i32> = vec![1, 2, 3];

let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();

assert_eq!(v2, vec![2, 3, 4]);
```

示例 13-18: 调用 `map` 方法创建一个新迭代器，接着调用 `collect` 方法消费新迭代器并创建一个 `vector`

因为 `map` 获取一个闭包，可以指定任何希望在遍历的每个元素上执行的操作。这是一个展示如何使用闭包来自定义行为同时又复用 `Iterator trait` 提供的迭代行为的绝佳例子。

使用闭包获取环境与迭代器

现在我们介绍了迭代器，让我们展示一个通过使用 `filter` 迭代器适配器和捕获环境的闭包的常规用例。迭代器的 `filter` 方法获取一个使用迭代器的每一个项并返回布尔值的闭包。如果闭包返回 `true`，其值将会包含在 `filter` 提供的新迭代器中。如果闭包返回 `false`，其值不会包含在结果迭代器中。示例 13-19 展示了使用 `filter` 和一个捕获环境中变量 `shoe_size` 的闭包，这样闭包就可以遍历一个 `Shoe` 结构体集合以便只返回指定大小的鞋子：

文件名: `src/lib.rs`

```
#[derive(PartialEq, Debug)]
struct Shoe {
    size: i32,
    style: String,
}

fn shoes_in_my_size(shoes: Vec<Shoe>, shoe_size: i32) -> Vec<Shoe> {
    shoes.into_iter()
        .filter(|s| s.size == shoe_size)
        .collect()
}

#[test]
fn filters_by_size() {
    let shoes = vec![
        Shoe { size: 10, style: String::from("sneaker") },
        Shoe { size: 13, style: String::from("sandal") },
        Shoe { size: 10, style: String::from("boot") },
    ];

    let in_my_size = shoes_in_my_size(shoes, 10);

    assert_eq!(
        in_my_size,
        vec![
            Shoe { size: 10, style: String::from("sneaker") },
            Shoe { size: 10, style: String::from("boot") },
        ]
    );
}
```

示例 13-19：使用 `filter` 方法和一个捕获 `shoe_size` 的闭包

`shoes_in_my_size` 函数获取一个鞋子 **vector** 的所有权和一个鞋子大小作为参数。它返回一个只包含指定大小鞋子的 **vector**。在 `shoes_in_my_size` 函数体中调用了 `into_iter` 来创建一个获取 **vector** 所有权的迭代器。接着调用 `filter` 将这个迭代器适配成只含有闭包返回 `true` 元素的新迭代器。我们指定的闭包从环境中捕获了 `shoe_size` 变量并使用其值与每一只鞋的大小作比较，只保留指定大小的鞋子。最终，调用 `collect` 将迭代器适配器返回的值收集进一个 **vector** 并返回。

这个测试展示当调用 `shoes_in_my_size` 时，我们只会得到与指定值相同大小的鞋子。

实现 **Iterator trait** 来创建自定义迭代器

我们已经展示了可以通过在 **vector** 上调用 `iter`、`into_iter` 或 `iter_mut` 来创建一个迭代器。也可以用标准库中其他的集合类型创建迭代器，比如哈希 **map**。另外，可以实现 **Iterator trait** 来创建任何我们希望的迭代器。正如之前提到的，定义中唯一要求提供的方法就是 `next` 方法。一旦定义了它，就可以使用所有其他由 **Iterator trait** 提供的拥有默认实现的方法来创建自定义迭代器了！

我们将要创建的迭代器只会从 1 数到 5。首先，我们会创建一个结构体来存放一些值，接着实现 **Iterator trait** 将这个结构体放入迭代器中并在此实现中使用其值。

示例 13-20 有一个 `Counter` 结构体定义和一个创建 `Counter` 实例的关联函数 `new`：

文件名: src/lib.rs

```
struct Counter {
    count: u32,
}

impl Counter {
    fn new() -> Counter {
        Counter { count: 0 }
    }
}
```

示例 13-20: 定义 `Counter` 结构体和一个创建 `count` 初值为 0 的 `Counter` 实例的 `new` 函数

`Counter` 结构体有一个字段 `count`。这个字段存放一个 `u32` 值，它会记录处理 1 到 5 的迭代过程中的位置。`count` 是私有的因为我们希望 `Counter` 的实现来管理这个值。`new` 函数通过总是从为 0 的 `count` 字段开始新实例来确保我们需要的行为。

接下来将为 `Counter` 类型实现 `Iterator` trait，通过定义 `next` 方法来指定使用迭代器时的行为，如示例 13-21 所示：

文件名: src/lib.rs

```
# struct Counter {
#     count: u32,
# }
#
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        self.count += 1;

        if self.count < 6 {
            Some(self.count)
        } else {
            None
        }
    }
}
```

示例 13-21: 在 `Counter` 结构体上实现 `Iterator` trait

这里将迭代器的关联类型 `Item` 设置为 `u32`，意味着迭代器会返回 `u32` 值集合。再一次，这里仍无需担心关联类型，第十九章会讲到。我们希望迭代器对其内部状态加一，这也就是为何将 `count` 初始化为 0：我们希望迭代器首先返回 1。如果 `count` 值小于 6，`next` 会返回封装在 `Some` 中的当前值，不过如果 `count` 大于或等于 6，迭代器会返回 `None`。

使用 `Counter` 迭代器的 `next` 方法

一旦实现了 `Iterator` trait，我们就有了一个迭代器！示例 13-22 展示了一个测试用来演示现在我们可以使用 `Counter` 结构体的迭代器功能，通过直接调用 `next` 方法，正如示例 13-15 中从 `vector` 创建的迭代器那样：

文件名: src/lib.rs

```
# struct Counter {
#     count: u32,
# }
```

```

#
# impl Iterator for Counter {
#     type Item = u32;
#
#     fn next(&mut self) -> Option<Self::Item> {
#         self.count += 1;
#
#         if self.count < 6 {
#             Some(self.count)
#         } else {
#             None
#         }
#     }
# }
#
#[test]
fn calling_next_directly() {
    let mut counter = Counter::new();

    assert_eq!(counter.next(), Some(1));
    assert_eq!(counter.next(), Some(2));
    assert_eq!(counter.next(), Some(3));
    assert_eq!(counter.next(), Some(4));
    assert_eq!(counter.next(), Some(5));
    assert_eq!(counter.next(), None);
}

```

示例 13-22: 测试 `next` 方法实现的功能

这个测试在 `counter` 变量中新建了一个 `Counter` 实例并接着反复调用 `next` 方法，来验证我们实现的行为符合这个迭代器返回从 1 到 5 的值的预期。

使用自定义迭代器中其他 `Iterator trait` 方法

通过定义 `next` 方法实现 `Iterator trait`，我们现在就可以使用任何标准库定义的拥有默认实现的 `Iterator trait` 方法了，因为他们都使用了 `next` 方法的功能。

例如，出于某种原因我们希望获取 `Counter` 实例产生的值，将这些值与另一个 `Counter` 实例在省略了第一个值之后产生的值配对，将每一对值相乘，只保留那些可以被三整除的结果，然后将所有保留的结果相加，这可以如示例 13-23 中的测试这样做：

文件名: `src/lib.rs`

```

# struct Counter {
#     count: u32,
# }
#
# impl Counter {
#     fn new() -> Counter {
#         Counter { count: 0 }
#     }
# }
#
# impl Iterator for Counter {
#     // Our iterator will produce u32s
#     type Item = u32;
#
#     fn next(&mut self) -> Option<Self::Item> {
#         // increment our count. This is why we started at zero.
#         self.count += 1;
#
#         // check to see if we've finished counting or not.

```

```

#         if self.count < 6 {
#             Some(self.count)
#         } else {
#             None
#         }
#     }
# }
#
#[test]
fn using_other_iterator_trait_methods() {
    let sum: u32 = Counter::new().zip(Counter::new().skip(1))
        .map(|(a, b)| a * b)
        .filter(|x| x % 3 == 0)
        .sum();

    assert_eq!(18, sum);
}

```

示例 13-23：使用自定义的 `Counter` 迭代器的多种方法

注意 `zip` 只产生4对值；理论上第五对值 `(5, None)` 从未被产生，因为 `zip` 在任一输入迭代器返回 `None` 时也返回 `None`。

所有这些方法调用都是可能的，因为我们通过指定 `next` 如何工作来实现 `Iterator` `trait` 而标准库则提供其他调用 `next` 的默认方法实现。

改进 I/O 项目

[ch13-03-improving-our-io-project.md](#)

commit 714be7f0d6b2f6110afe8808a7f528f9eae75c61

我们可以使用迭代器来改进第十二章中 I/O 项目的实现来使得代码更简洁明了。让我们看看迭代器如何能够改进 `Config::new` 函数和 `search` 函数的实现。

使用迭代器并去掉 `clone`

在示例 12-6 中，我们增加了一些代码获取一个 `String slice` 并创建一个 `Config` 结构体的实例，他们索引 `slice` 中的值并克隆这些值以便 `Config` 结构体可以拥有这些值。在示例 13-24 中原原本本的重现了第十二章结尾

`Config::new` 函数的实现：

文件名: `src/lib.rs`

```
impl Config {
    pub fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let filename = args[2].clone();

        let case_sensitive = env::var("CASE_INSENSITIVE").is_err();

        Ok(Config { query, filename, case_sensitive })
    }
}
```

示例 13-24：重现第十二章结尾的 `Config::new` 函数

这时可以不必担心低效的 `clone` 调用了，因为将来可以去掉他们。好吧，就是现在！

起初这里需要 `clone` 的原因是参数 `args` 中有一个 `String` 元素的 `slice`，而 `new` 函数并不拥有 `args`。为了能够返回 `Config` 实例的所有权，我们需要克隆 `Config` 中字段 `query` 和 `filename` 的值，这样 `Config` 实例就能拥有这些值。

通过迭代器的新知识，我们可以将 `new` 函数改为获取一个有所有权的迭代器作为参数而不是借用 `slice`。我们将使用迭代器功能之前检查 `slice` 长度和索引特定位置的代码。这会清理 `Config::new` 的工作因为迭代器会负责访问这些值。

一旦 `Config::new` 获取了迭代器的所有权并不再使用借用的索引操作，就可以将迭代器中的 `String` 值移动到 `Config` 中，而不是调用 `clone` 分配新的空间。

直接使用 `env::args` 返回的迭代器

在 I/O 项目的 `src/main.rs` 中，让我们修改第十二章结尾 `main` 函数中的这些代码：

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

    let config = Config::new(&args).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {}", err);
    });
}
```

```

        process::exit(1);
    });

    // ...snip...
}

```

将他们改为如示例 13-25 所示：

文件名: `src/main.rs`

```

fn main() {
    let config = Config::new(env::args()).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    // ...snip...
}

```

示例 13-25：将 `env::args` 的返回值传递给 `Config::new`

`env::args` 函数返回一个迭代器！不同于将迭代器的值收集到一个 `vector` 中接着传递一个 `slice` 给 `Config::new`，现在我们直接将 `env::args` 返回的迭代器的所有权传递给 `Config::new`。

接下来需要更新 `Config::new` 的定义。在 I/O 项目的 `src/lib.rs` 中，将 `Config::new` 的签名改为如示例 13-26 所示：

文件名: `src/lib.rs`

```

impl Config {
    pub fn new(args: std::env::Args) -> Result<Config, &'static str> {
        // ...snip...
    }
}

```

示例 13-26：更新 `Config::new` 的签名来接受一个迭代器

`env::args` 函数的标准库文档展示了其返回的迭代器类型是 `std::env::Args`。需要更新 `Config::new` 函数的签名中 `args` 参数的类型为 `std::env::Args` 而不是 `&[String]`。

使用 `Iterator trait` 方法带起索引

接下来修复 `Config::new` 的函数体。标准库文档也提到了 `std::env::Args` 实现了 `Iterator trait`，所以可以在其上调用 `next` 方法！示例 13-27 更新了示例 12-23 中的代码为使用 `next` 方法：

文件名: `src/lib.rs`

```

# use std::env;
#
# struct Config {
#     query: String,
#     filename: String,
#     case_sensitive: bool,
# }
#
impl Config {
    pub fn new(mut args: std::env::Args) -> Result<Config, &'static str> {
        args.next();

        let query = match args.next() {
            Some(arg) => arg,
            None => return Err("Didn't get a query string"),
        };
    }
}

```

```

    };

    let filename = match args.next() {
        Some(arg) => arg,
        None => return Err("Didn't get a file name"),
    };

    let case_sensitive = env::var("CASE_INSENSITIVE").is_err();

    Ok(Config {
        query, filename, case_sensitive
    })
}

```

示例 13-27: 修改 `Config::new` 的函数体为使用迭代器方法

请记住 `env::args` 返回值的第一个值是程序的名称。我们希望忽略它并获取下一个值，所以首先调用 `next` 并不对返回值做任何操作。之后对希望放入 `Config` 中字段 `query` 调用 `next`。如果 `next` 返回 `Some`，使用 `match` 来提取其值。如果它返回 `None`，则意味着没有提供足够的参数并通过 `Err` 值提早返回。对 `filename` 值进行同样的操作。

使用迭代器适配器来使代码更简明

I/O 项目中其他可以利用迭代器优势的地方位于 `search` 函数，在示例 13-28 中重现了第十二章结尾的此函数定义：

文件名: `src/lib.rs`

```

pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}

```

示例 13-28: 第十二章结尾 `search` 函数的定义

可以通过使用迭代器适配器方法来编写更短的代码。这也避免了一个可变的中间 `results` `vector` 的使用。函数式编程风格倾向于最小化可变状态的数量来使代码更简洁。去掉可变状态可能会使得将来进行并行搜索的增强变得更容易，因为我们不必管理 `results` `vector` 的并发访问。示例 13-29 展示了该变化：

文件名: `src/lib.rs`

```

pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    contents.lines()
        .filter(|line| line.contains(query))
        .collect()
}

```

示例 13-29: 在 `search` 函数实现中使用迭代器适配器

回忆 `search` 函数的目的是返回所有 `contents` 中包含 `query` 的行。类似于示例 13-19 中的 `filter` 例子，可以使用 `filter` 适配器只保留 `line.contains(query)` 为真的那些行。接着使用 `collect` 将匹配行收集到另一个 `vector` 中。这样就容易多了！请随意对 `search_case_insensitive` 函数做出同样的使用迭代器方法的修改。

接下来的逻辑问题就是在代码中应该选择哪种风格：示例 13-28 中的原始实现，或者是示例 13-29 中使用迭代器的版本。大部分 **Rust** 程序员倾向于使用迭代器风格。开始这有点难以理解，不过一旦你对不同迭代器的工作方式有了感觉之后，迭代器可能会更容易理解。相比摆弄不同的循环并创建新 `vector`，（迭代器）代码则更关注循环的目的。这抽象出了那些老生常谈的代码，这样就更容易看清代码所特有的概念，比如迭代器中每个元素必须面对的过滤条件。

不过这两种实现真的完全等同吗？直觉上的假设是更底层的循环会更快一些。让我们聊聊性能吧。

性能对比：循环 VS 迭代器

ch13-04-performance.md

commit 40910f557c328858f230123d1234c1cb3029dda3

为了决定使用哪个实现，我们需要知道哪个版本的 `search` 函数更快：直接使用 `for` 循环的版本还是使用迭代器的版本。

我们运行了一个性能测试，通过将阿瑟·柯南·道尔的“福尔摩斯探案集”的全部内容加载进 `String` 并寻找其中的单词“the”。如下是 `for` 循环版本和迭代器版本的 `search` 函数的性能测试结果：

```
test bench_search_for ... bench: 19,620,300 ns/iter (+/- 915,700)
test bench_search_iter ... bench: 19,234,900 ns/iter (+/- 657,200)
```

结果迭代器版本还要稍微快一点！这里我们将不会查看性能测试的代码，我们的目的并不是为了证明他们是完全等同的，而是得出一个怎样比较这两种实现方式性能的基本思路。对于一个更全面的性能测试，将会检查不同长度的文本、不同的搜索单词、不同长度的单词和所有其他的可变情况。这里所要表达的是：迭代器，作为一个高级的抽象，被编译成了与手写的底层代码大体一致性能代码。迭代器是 Rust 的零成本抽象（*zero-cost abstractions*）之一，它意味着抽象并不会强加运行时开销，它与本贾尼·斯特劳斯特卢普，C++ 的设计和实现者所定义的零开销（*zero-overhead*）如出一辙：

In general, C++ implementations obey the zero-overhead principle: What you don't use, you don't pay for. And further: What you do use, you couldn't hand code any better.

- Bjarne Stroustrup "Foundations of C++"

从整体来说，C++ 的实现遵循了零开销原则：你不需要的，无需为他们买单。更有甚者的是：你需要的时候，也不可能找到其他更好的代码了。

- 本贾尼·斯特劳斯特卢普 "Foundations of C++"

作为另一个例子，这里有一些取自于音频解码器的代码。这些代码使用迭代器链来对作用域中的三个变量进行了某种数学计算：一个叫 `buffer` 的数据 slice、一个有 12 个元素的数组 `coefficients`、和一个代表位移位数的 `qlp_shift`。例子中声明了这些变量但并没有提供任何值；虽然这些代码在其上下文之外没有什么意义，不过仍是一个简明的现实中的例子，来展示 Rust 如何将高级概念转换为底层代码：

```
let buffer: &mut [i32];
let coefficients: [i64; 12];
let qlp_shift: i16;

for i in 12..buffer.len() {
    let prediction = coefficients.iter()
        .zip(&buffer[i - 12..i])
        .map(|(&c, &s)| c * s as i64)
        .sum::<i64>() >> qlp_shift;

    let delta = buffer[i];
    buffer[i] = prediction as i32 + delta;
}
```

为了计算 `prediction` 的值，这些代码遍历了 `coefficients` 中的 12 个值，使用 `zip` 方法将系数与 `buffer` 的前 12 个值组合在一起。接着将每一对值相乘，再将所有结果相加，然后将总和右移 `qlp_shift` 位。

像音频解码器这样的程序通常最看重计算的性能。这里，我们创建了一个迭代器，使用了两个适配器，接着消费了其值。**Rust** 代码将会被编译为什么样的汇编代码呢？好吧，在编写本书的这个时候，它被编译成与手写的相同的汇编代码。遍历 `coefficients` 的值完全用不到循环：**Rust** 知道这里会迭代 12 次，所以它“展开”（`unroll`）了循环。展开是一种移除循环的控制代码开销并替换为每个迭代中的重复代码的优化。

所有的系数都被储存在了寄存器中，这意味着访问他们非常快。这里也没有运行时数组访问边界检查。所有这些 **Rust** 能够提供的优化使得结果代码极为高效。

现在知道这些了，请放心大胆的使用迭代器和闭包吧！他们使得代码看起来更高级，但并不为此引入运行时性能损失。

总结

闭包和迭代器是 **Rust** 受函数式编程语言观念所启发的功能。他们对 **Rust** 以底层的性能来明确的表达高级概念的能力有很大贡献。闭包和迭代器的实现达到了不影响运行时性能的程度。这正是 **Rust** 竭力提供零成本抽象的目标的一部分。

现在我们改进了我们 I/O 项目的（代码）表现力，让我们看一看更多 `cargo` 的功能，他们将帮助我们准备好将项目分享给世界。

进一步认识 **Cargo** 和 **Crates.io**

[ch14-00-more-about-cargo.md](#)

commit db6129a30d7c7baed34dd38dbc56f7ed8a66ae92

目前为止我们只使用过 **Cargo** 构建、运行和测试代码的最基本功能，不过它还可以做到更多。这里我们将了解一些 **Cargo** 其他更高级的功能，他们将展示如何：

- 使用发布配置来自定义构建
- 将库发布到 **crates.io**
- 使用工作空间来组织更大的项目
- 从 **crates.io** 安装二进制文件
- 使用自定义的命令来扩展 **Cargo**

相比本章能够涉及的工作 **Cargo** 甚至还可以做到更多，关于其功能的全部解释，请查看[文档](#)

发布配置

ch14-01-release-profiles.md
commit db6129a30d7c7baed34dd38dbc56f7ed8a66ae92

在 Rust 中 发布配置（*release profiles*）是预定义的、可定制的带有不同选项的配置，他们允许程序员更多的控制代码编译的多种选项。每一个配置都彼此相互独立。

Cargo 定义了四种有着良好默认值的可用于各自使用场景的配置。Cargo 根据运行的命令来选择不同的配置。不同命令所对应的配置如表格 14-1 所示：

命令	配置
cargo build	dev
cargo build --release	release
cargo test	test
cargo doc	doc

表格 14-1：运行不同 Cargo 命令所使用的配置

这可能很熟悉，他们出现在构建的输出中，他们展示了构建中所使用的配置：

```
$ cargo build
  Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
$ cargo build --release
  Finished release [optimized] target(s) in 0.0 secs
```

这里的“dev”和“release”提示表明编译器在使用不同的配置。

定制发布配置

Cargo 对每一个配置都有默认设置，当项目的 Cargo.toml 文件的 [profile.*] 部分没有指定时使用。通过增加任何希望定制的配置对应的 [profile.*] 部分，我们可以选择覆盖任意默认设置的子集。例如，如下是 dev 和 release 配置的 opt-level 设置的默认值：

```
[profile.dev]
opt-level = 0

[profile.release]
opt-level = 3
```

opt-level 设置控制 Rust 会对代码进行何种程度的优化。这个配置的值从 0 到 3。越高的优化级别需要更多的时间编译，所以如果你在进行开发并经常编译，可能会希望在牺牲一些代码性能的情况下编译得快一些。这就是为什么 dev 的 opt-level 默认为 0。当你准备发布时，花费更多时间在编译上则更好。只需要在发布模式编译一次，而编译出来的程序则会运行很多次，所以发布模式用更长的编译时间换取运行更快的代码。这正是为什么 release 配置的 opt-level 默认为 3。

我们可以选择通过在 Cargo.toml 增加不同的值来覆盖任何默认设置。比如，如果我们想要在开发配置中使用级别 1 的优化，则可以在 Cargo.toml 中增加这两行：

文件名: Cargo.toml

```
[profile.dev]
opt-level = 1
```

这会覆盖默认的设置 `0`。现在运行 `cargo build` 时，Cargo 将会使用 `dev` 的默认配置加上定制的 `opt-level`。因为 `opt-level` 设置为 `1`，Cargo 会比默认进行更多的优化，但是没有发布构建那么多。

对于每个配置的设置和其默认值的完整列表，请查看[Cargo 的文档](#)。

将 crate 发布到 Crates.io

ch14-02-publishing-to-crates-io.md

commit 56352c28cf3fe0402fa5a7cba73890e314d720eb

我们曾经在项目中增加 crates.io 上的包作为依赖，不过你也可以通过发布自己的包来向它人分享代码。Crates.io 用来分发包的源代码，所以它主要托管开源代码。

Rust 和 Cargo 有一些帮助它人更方便找到和使用你发布的包的功能。我们将介绍一些这样的功能，接着讲到如何发布一个包。

编写有用的文档注释

准确的包文档有助于其他用户立即如何以及何时使用他们，所以花一些时间编写文档是值得的。第三章中我们讨论了如何使用 `///` 注释 Rust 代码。Rust 也有特定的用于文档的注释类型，通常被称为 文档注释（*documentation comments*），他们会生成 HTML 文档。这些 HTML 展示公有 API 文档注释的内容，他们意在让对库感兴趣的程序员理解如何使用这个 crate，而不是它是如何被实现的。

文档注释使用 `///` 而不是 `//` 并支持 Markdown 注解来格式化文本。文档注释就位于需要文档的项的之前。示例 14-2 展示了一个 `my_crate` crate 中 `add_one` 函数的文档注释：

文件名: `src/lib.rs`

```
/// Adds one to the number given.
///
/// # Examples
///
///
```

```
/// let five = 5; /// /// assert_eq!(6, my_crate::add_one(5)); /// ``` pub fn add_one(x: i32) -> i32 { x + 1 }
```

示例 14-2: 一个函数的文档注释

这里，我们提供了一个 `add_one` 函数工作的描述，接着开始了一个标题为“Examples”的部分，和展示如何使用 `add_one` 函数的代码。可以运行 `cargo doc` 来生成这个文档注释的 HTML 文档。这个命令运行由 Rust 分发的工具 `rustdoc` 并将生成的 HTML 文档放入 `*target/doc*` 目录。

为了方便起见，运行 `cargo doc --open` 会构建当前 crate 文档（同时还有所有 crate 依赖的文档）的 HTML 并在浏览器中打开。导航到 `add_one` 函数将会发现文档注释的文本是如何渲染的，如图 13-3 所示：

图 14-3: add_one 函数的文档注释 HTML

常用（文档注释）部分

示例 14-2 中使用了 `# Examples` Markdown 标题在 HTML 中创建了一个以“Examples”为标题的部分。一些其他经常在文档注释中使用的部分有：

- **Panics:** 这个函数可能会 `panic!` 的场景。并不希望程序崩溃的函数调用者应该确保他们不会在这些情况下调用此函数。
- **Errors:** 如果这个函数返回 `Result`，此部分描述可能会出现何种错误以及什么情况会造成这些错误，这有助于调用者编写代码来采用不同的方式处理不同的错误。
- **Safety:** 如果这个函数使用 `unsafe` 代码（这会在第十九章讨论），这一部分应该会涉及到期望函数调用者支持的确保 `unsafe` 块中代码正常工作的不变条件（*invariants*）。

大部分文档注释不需要所有这些部分，不过这是一个提醒你检查调用你代码的人有兴趣了解的内容的列表。

```
#### 文档注释作为测试
```

在文档注释中添加示例代码块是一个清楚的表明如何使用库的方法，这么做还有一个额外的好处：`cargo test` 也会像测试那样运行文档中的示例代码！没有什么比有例子的文档更好的了！也没有什么比不能正常工作的例子更糟的了，因为代码在编写文档时已经改变。尝试 `cargo test` 运行像示例 14-2 中 `add_one` 函数的文档：应该在测试结果中看到像这样的部分：

```
```text
Doc-tests my_crate

running 1 test
test src/lib.rs - add_one (line 5) ... ok

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

现在尝试改变函数或例子来使例子中的 `assert_eq!` 产生 `panic`。再次运行 `cargo test`，你将会看到文档测试捕获到了例子与代码不再同步！

## 注释包含项的结构

还有另一种风格的文档注释，`///`，这为包含注释的项，而不是注释之后的项增加文档。这通常用于 `crate` 根文件或模块的根文件为 `crate` 或模块整体提供文档。

作为一个例子，如果我们希望增加描述包含 `add_one` 函数的 `my_crate` `crate` 目的的文档，可以在 `src/lib.rs` 开头增加以 `///` 开头的注释，如示例 14-4 所示：

文件名: `src/lib.rs`

```
/// # My Crate
///
/// `my_crate` is a collection of utilities to make performing certain
/// calculations more convenient.
///
/// Adds one to the number given.
// ...snip...
```

示例 14-4: `my_crate` `crate` 整体的文档

注意 `///` 的最后一行之后没有任何代码。因为他们以 `///` 开头而不是 `///`，这是属于包含此注释的项而不是注释之后项的文档。在这个情况中，包含这个注释的项是 `src/lib.rs` 文件，也就是 `crate` 根文件。这些注释描述了整个 `crate`。

如果运行 `cargo doc --open`，将会发现这些注释显示在 `my_crate` 文档的首页，位于 `crate` 中公有项列表之上，如图 14-5 所示：

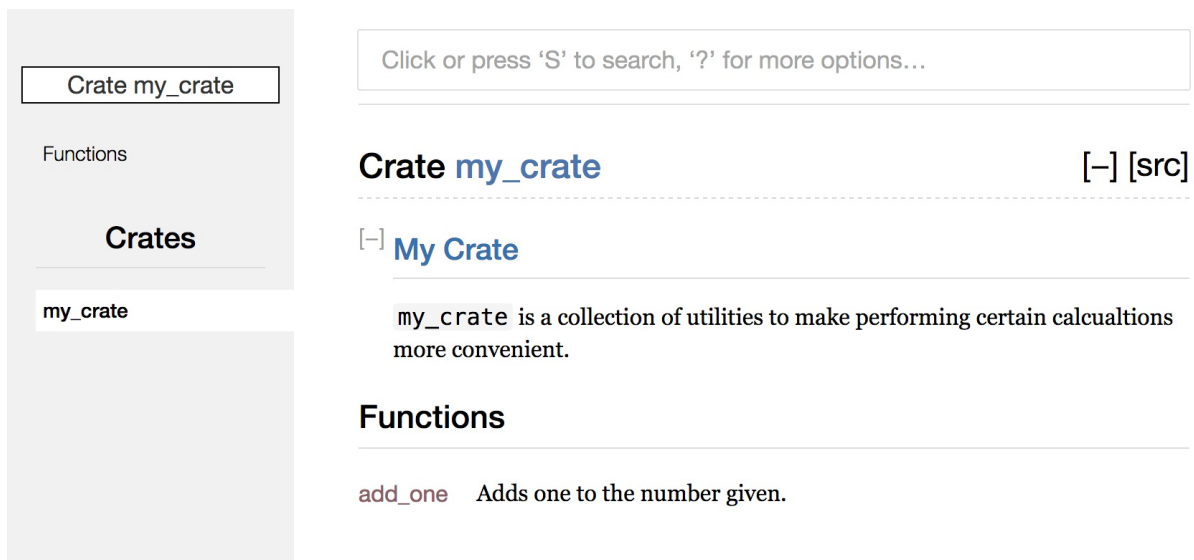


图 14-5: 包含 `my_crate` 整体描述的注释所渲染的文档

位于项之中的文档注释对于描述 `crate` 和模块特别有用。使用他们描述其容器整体的目的来帮助 `crate` 用户理解你的代码组织。

## 使用 `pub use` 导出合适的公有 API

第七章介绍了如何使用 `mod` 关键字来将代码组织进模块中，如何使用 `pub` 关键字将项变为公有，和如何使用 `use` 关键字将项引入作用域。然而对你开发来说很有道理的结果可能对用户来说就不太方便了。你可能希望将结构组织进有多个层次的层级中，不过想要使用被定义在很深层级中的类型的人可能很难发现这些类型是否存在。他们也可能厌烦 `use my_crate::some_module::another_module::UsefulType;` 而不是 `use my_crate::UsefulType;` 来使用类型。

公有 API 的结构是你发布 `crate` 时主要需要考虑的。`crate` 用户没有你那么熟悉其结构，并且如果模块层级过大他们可能会难以找到所需的部分。

好消息是，如果结果对于用户来说不是很方便，你也无需重新安排内部组织：你可以选择使用 `pub use` 重导出（`re-export`）项来使公有结构不同于私有结构。重导出获取位于一个位置的公有项并将其公开到另一个位置，好像它就定义在这个新位置一样。

例如，假设我们创建了一个模块化了充满艺术化气息的库 `art`。在这个库中是一个包含两个枚举 `PrimaryColor` 和 `SecondaryColor` 的模块 `kinds`，以及一个包含函数 `mix` 的模块 `utils`，如示例 14-6 所示：

文件名: `src/lib.rs`

```

/// # Art
///
/// A library for modeling artistic concepts.

pub mod kinds {
 /// The primary colors according to the RYB color model.
 pub enum PrimaryColor {
 Red,
 Yellow,
 Blue,
 }

 /// The secondary colors according to the RYB color model.
 pub enum SecondaryColor {
 Orange,
 }
}

```



```

 Green,
 Purple,
 }
}

pub mod utils {
 use kinds::*;

 /// Combines two primary colors in equal amounts to create
 /// a secondary color.
 pub fn mix(c1: PrimaryColor, c2: PrimaryColor) -> SecondaryColor {
 // ...snip...
 }
}
SecondaryColor::Green
}
}

```

示例 14-6: 一个库 `art` 其组织包含 `kinds` 和 `utils` 模块

`cargo doc` 所生成的 `crate` 文档首页如图 14-7 所示:

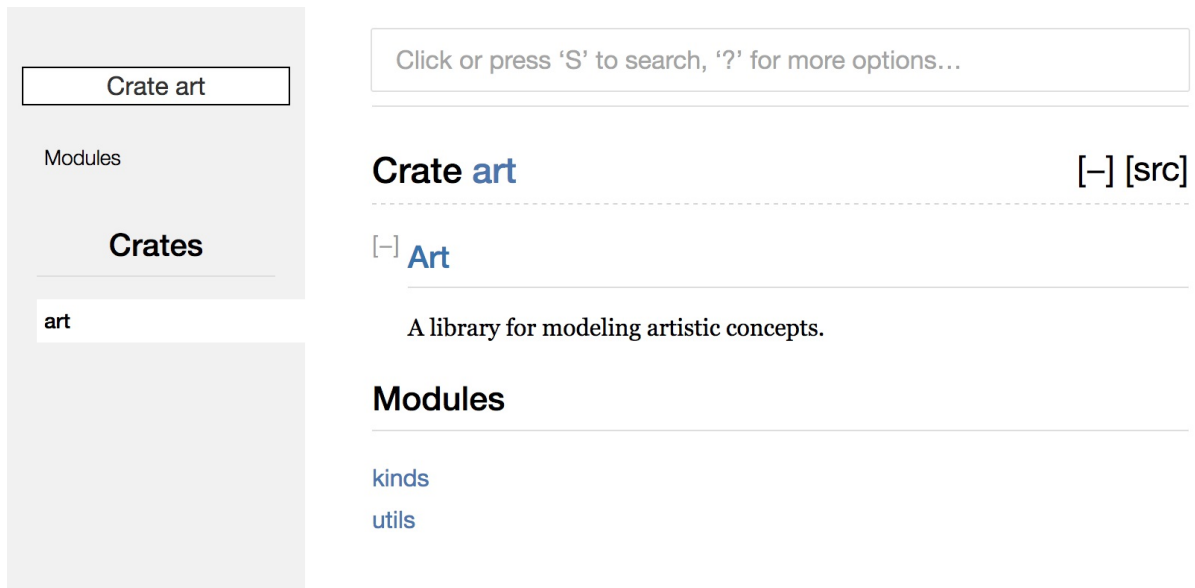


图 14-7: 包含 `kinds` 和 `utils` 模块的库 `art` 的文档首页

注意 `PrimaryColor` 和 `SecondaryColor` 类型没有在首页中列出, `mix` 函数也是。必须点击 `kinds` 或 `utils` 才能看到他们。

另一个依赖这个库的 `crate` 需要 `use` 语句来导入 `art` 中的项, 这包含指定其当前定义的模块结构。示例 14-8 展示了一个使用 `art` `crate` 中 `PrimaryColor` 和 `mix` 项的 `crate` 的例子:

文件名: `src/main.rs`

```

extern crate art;

use art::kinds::PrimaryColor;
use art::utils::mix;

fn main() {
 let red = PrimaryColor::Red;
 let yellow = PrimaryColor::Yellow;
 mix(red, yellow);
}

```

示例 14-8: 一个通过导出内部结构使用 `art crate` 中项的 `crate`

示例 14-8 中使用 `art crate` 代码的作者不得不搞清楚 `PrimaryColor` 位于 `kinds` 模块而 `mix` 位于 `utils` 模块。`art crate` 的模块结构相比使用它的开发者来说对编写它的开发者更有意义。其内部的 `kinds` 模块和 `utils` 模块的组织结构并没有对尝试理解如何使用它的人提供任何有价值的信息。`art crate` 的模块结构因不得不搞清楚所需的内容在何处和必须在 `use` 语句中指定模块名称而显得混乱和不便。

为了从公有 API 中去掉 `crate` 的内部组织，我们可以采用示例 14-6 中的 `art crate` 并增加 `pub use` 语句来重导出项到顶层结构，如示例 14-9 所示：

文件名: `src/lib.rs`

```
//! # Art
//!
//! A library for modeling artistic concepts.

pub use kinds::PrimaryColor;
pub use kinds::SecondaryColor;
pub use utils::mix;

pub mod kinds {
 // ...snip...
}

pub mod utils {
 // ...snip...
}
```

示例 14-9: 增加 `pub use` 语句重导出项

现在此 `crate` 由 `cargo doc` 生成的 API 文档会在首页列出重导出的项以及其链接，如图 14-10 所示，这就使得这些类型易于查找。

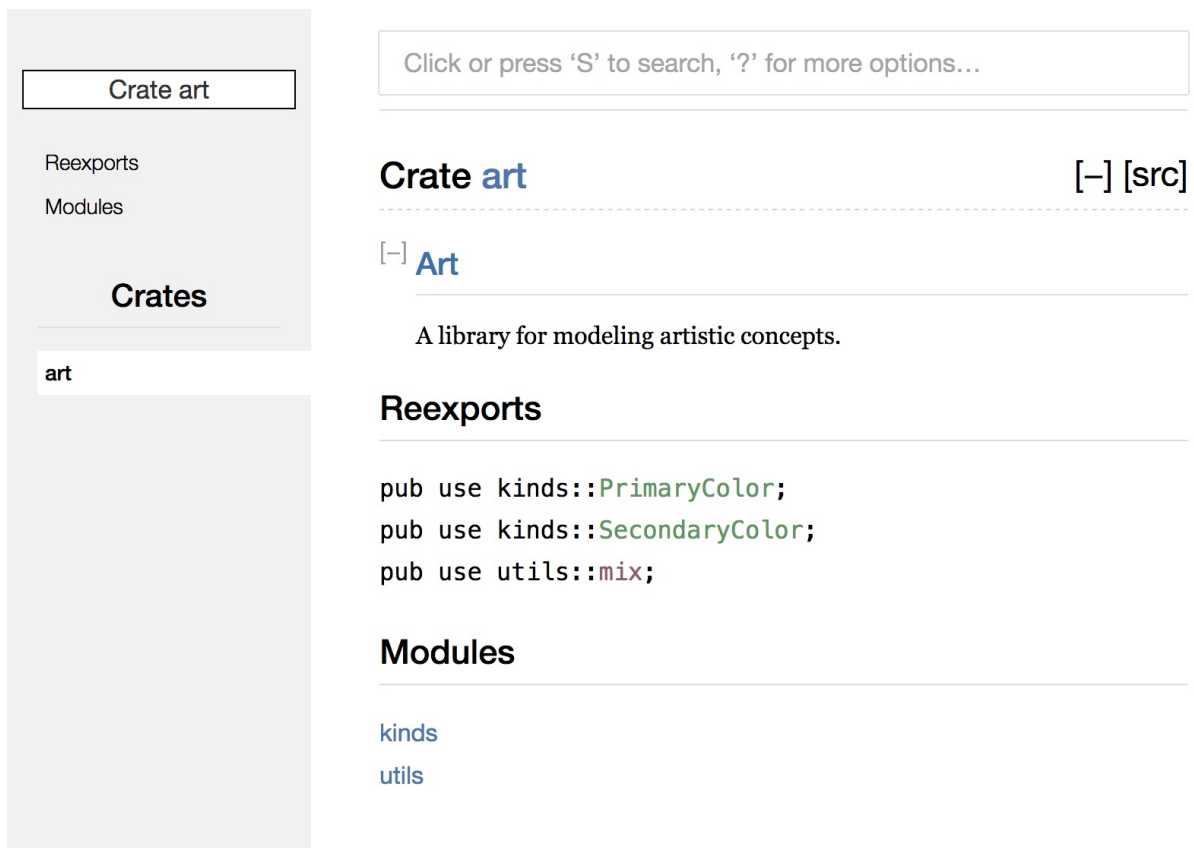


图 14-10: `art` 文档的首页，这里列出了重导出的项

`art crate` 的用户仍然可以看见和选择使用示例 14-8 中的内部结构，或者可以使用示例 14-9 中更为方便的结构，如示例 14-11 所示：

文件名: `src/main.rs`

```
extern crate art;

use art::PrimaryColor;
use art::mix;

fn main() {
 // ...snip...
}
```

示例 14-11: 一个使用 `art crate`

对于有很多嵌套模块的情况，使用 `pub use` 将类型重导出到顶级结构对于使用 `crate` 的人来说将会是大为不同的体验。

创建一个有用的公有 API 结构更像是一门艺术而非科学，你可以反复检视他们来找出最适合用户的 API。选择 `pub use` 提供了组织 `crate` 内部结构和与终端用户体现解耦的灵活性。观察一些你所安装的 `crate` 的代码来看看其内部结构是否不同于公有 API。

## 创建 Crates.io 账号

在你发布任何 `crate` 之前，需要在 [crates.io](https://crates.io) 上注册账号并获取一个 API token。为此，访问位于 <https://crates.io> 的官网并使用 GitHub 账号登陆——目前 GitHub 账号是必须的，不过将来该网站可能会支持其他创建账号的方法。一旦登陆之后，查看位于 <https://crates.io/me> 的账户设置页面并获取 API token。接着使用

该 API token 运行 `cargo login` 命令，像这样：

```
$ cargo login abcdefghijklmnopqrstuvwxyz012345
```

这个命令会通知 Cargo 你的 API token 并将其储存在本地的 `~/.cargo/config` 文件中。注意这个 token 是一个秘密（**secret**）且不应该与其他人共享。如果因为任何原因与他人共享了这个信息，应该立即重新生成这个 token。

## 发布新 crate 之前

有了账号之后，比如说你已经有一个希望发布的 crate。在发布之前，你需要在 crate 的 *Cargo.toml* 文件的 `[package]` 部分增加一些本 crate 的元信息（**metadata**）。

首先 crate 需要一个唯一的名称。虽然在本地开发 crate 时，可以使用任何你喜欢的名称。不过 Crates.io 上的 crate 名称遵守先到先得的分配原则。一旦某个 crate 名称被使用，其他人就不能再发布这个名称的 crate 了。请在网站上搜索你希望使用的名称来找出它是否已被使用。如果没有，修改 *Cargo.toml* 中 `[package]` 里的名称为你希望用于发布的名称，像这样：

```
[package]
name = "guessing_game"
```

即使你选择了一个唯一的名称，如果此时尝试运行 `cargo publish` 发布该 crate 的话，会得到一个警告接着是一个错误：

```
$ cargo publish
Updating registry `https://github.com/rust-lang/crates.io-index`
warning: manifest has no description, license, license-file, documentation,
homepage or repository.
...snip...
error: api errors: missing or empty metadata fields: description, license.
```

这是因为我们缺少一些关键信息：关于该 crate 用途的描述和用户可能在何种条款下使用该 crate 的 license。为了修正这个错误，需要在 *Cargo.toml* 中引入这些信息。

描述通常是一两句话，因为它会出现在 crate 的搜索结果中和 crate 页面里。对于 `license` 字段，你需要一个 **license** 标识符值（*license identifier value*）。Linux 基金会位于 <http://spdx.org/licenses/> 的 Software Package Data Exchange (SPDX) 列出了可以使用的标识符。例如，为了指定 crate 使用 MIT License，增加 MIT 标识符：

```
[package]
name = "guessing_game"
license = "MIT"
```

如果你希望使用不存在于 SPDX 的 license，则需要将 license 文本放入一个文件，将该文件包含进项目中，接着使用 `license-file` 来指定文件名而不是使用 `license` 字段。

关于项目所适用的 license 指导超出了本书的范畴。很多 Rust 社区成员选择与 Rust 自身相同的 license，这是一个双许可的 MIT/Apache-2.0 ——这展示了也可以通过斜杠来分隔来指定多个 license 标识符。

那么，有了唯一的名称、版本号、由 `cargo new` 新建项目时增加的作者信息、描述和所选择的 license，已经准备好发布的项目的 *Cargo.toml* 文件可能看起来像这样：

```
[package]
name = "guessing_game"
version = "0.1.0"
```

```
authors = ["Your Name <you@example.com>"]
description = "A fun game where you guess what number the computer has chosen."
license = "MIT/Apache-2.0"

[dependencies]
```

[Cargo 的文档](#) 描述了其他可以指定的元信息，他们可以帮助你的 `crate` 更容易被发现和使用！

## 发布到 Crates.io

现在我们创建了一个账号，保存了 `API token`，为 `crate` 选择了一个名字，并指定了所需的元数据，你已经准备好发布了！发布 `crate` 会上传特定版本的 `crate` 到 `crates.io` 以供他人使用。

发布 `crate` 时请多加小心，因为发布是永久性的（*permanent*）。对应版本不可能被覆盖，其代码也不可能被删除。`Crates.io` 的一个主要目标是作为一个代码的永久文档服务器，这样所有依赖 `Crates.io` 中 `crate` 的项目都能一直正常工作。允许删除版本将不可能满足这个目标。然而，可以被发布的版本号却没有限制。

让我们再次运行 `cargo publish` 命令。这次它应该会成功：

```
$ cargo publish
Updating registry `https://github.com/rust-lang/crates.io-index`
Packaging guessing_game v0.1.0 (file:///projects/guessing_game)
Verifying guessing_game v0.1.0 (file:///projects/guessing_game)
Compiling guessing_game v0.1.0
(file:///projects/guessing_game/target/package/guessing_game-0.1.0)
Finished dev [unoptimized + debuginfo] target(s) in 0.19 secs
Uploading guessing_game v0.1.0 (file:///projects/guessing_game)
```

恭喜！你现在向 `Rust` 社区分享了代码，而且任何人都可以轻松的将你的 `crate` 加入他们项目的依赖。

## 发布现存 crate 的新版本

当你修改了 `crate` 并准备好发布新版本时，改变 `Cargo.toml` 中 `version` 所指定的值。请使用 [语义化版本规则](#) 来根据修改的类型决定下一个版本号。接着运行 `cargo publish` 来上传新版本。

## 使用 cargo yank 从 Crates.io 撤回版本

虽然你不能删除之前版本的 `crate`，但是可以阻止任何将来的项目将他们加入到依赖中。这在某个版本因为这样或那样的原因被破坏的情况很有用。对于这种情况，`Cargo` 支持 撤回（*yanking*）某个版本。

撤回某个版本会阻止新项目开始依赖此版本，不过所有现存此依赖的项目仍然能够下载和依赖这个版本。从本质上说，撤回意味着所有带有 `Cargo.lock` 的项目的依赖不会被破坏，同时任何新生成的 `Cargo.lock` 将不能使用被撤回的版本。

为了撤回一个 `crate`，运行 `cargo yank` 并指定希望撤回的版本：

```
$ cargo yank --vers 1.0.1
```

也可以撤销撤回操作，并允许项目可以再次开始依赖某个版本，通过在命令上增加 `--undo`：

```
$ cargo yank --vers 1.0.1 --undo
```

撤回 并没有 删除任何代码。举例来说，撤回功能并不意在删除不小心上传的秘密信息。如果出现了这种情况，请立即重新设置这些秘密信息。



# Cargo 工作空间

[ch14-03-cargo-workspaces.md](#)

commit 6e53771a409794d9933c2a31310d78149b7e0534

第十二章中，我们构建一个包含二进制 `crate` 和库 `crate` 的包。你可能会发现，随着项目开发的深入，库 `crate` 持续增大，而你希望将其进一步拆分成多个库 `crate`。对于这种情况，Cargo 提供了一个叫工作空间（*workspaces*）的功能，它可以帮助我们管理多个相关的并行开发的包。

工作空间 是一系列共享同样的 *Cargo.lock* 和输出目录的包。让我们使用工作空间创建一个项目，这里采用常见的代码这样就可以关注工作空间的结构了。这里有一个使用了两个库的二进制项目：一个库会提供 `add_one` 方法而第二个会提供 `add_two` 方法。让我们以为这个二进制项目创建一个新 `crate` 作为开始：

```
$ cargo new --bin adder
 Created binary (application) `adder` project
$ cd adder
```

我们需要修改二进制包的 *Cargo.toml* 并增加一个 `[workspace]` 部分来告诉 Cargo 包 `adder` 是一个工作空间。在文件末尾增加如下内容：

```
[workspace]
```

类似于很多 Cargo 的功能，工作空间支持配置惯例：只要遵循这些惯例就无需在 *Cargo.toml* 中增加更多的配置来定义工作空间了。

## 指定工作空间的依赖

工作空间惯例表明任何顶级 `crate` 依赖的位于任意子目录的 `crate` 都是工作空间的一部分。任何 `crate`，无论是否在工作空间中，可以在 *Cargo.toml* 中使用 `path` 属性来指定它拥有本地目录中的 `crate` 作为依赖。如果 `crate` 拥有 `[workspace]` 部分并指定了路径依赖，而这些路径是 `crate` 的子目录，则这些相关的 `crate` 被认为是工作空间的一部分。让我们在顶级的 `adder` `crate` 的 *Cargo.toml* 中为其指定位于 `add-one` 子目录的 `add-one` `crate` 作为依赖，通过这样修改 *Cargo.toml*：

```
[dependencies]
add-one = { path = "add-one" }
```

如果在 *Cargo.toml* 中增加依赖但没有指定对应 `path`，则这些将是不属于工作空间的假设来自于 Crates.io 的常规依赖。

## 在工作空间中创建第二个 `crate`

接下来，在 `adder` 目录中生成 `add-one` `crate`：

```
$ cargo new add-one
 Created library `add-one` project
```

现在 `adder` 目录应该有如下目录和文件：

```
├─ Cargo.toml
```

```

├─ add-one
│ └─ Cargo.toml
│ └─ src
│ └─ lib.rs
└─ src
 └─ main.rs

```

在 `add-one/src/lib.rs` 中增加 `add_one` 函数的实现：

文件名: `add-one/src/lib.rs`

```

pub fn add_one(x: i32) -> i32 {
 x + 1
}

```

打开 `adder` 的 `src/main.rs` 并增加一行 `extern crate` 将新的 `add-one` 库引入作用域，并修改 `main` 函数来调用 `add_one` 函数，如示例 14-12 所示：

```

extern crate add_one;

fn main() {
 let num = 10;
 println!("Hello, world! {} plus one is {}!", num, add_one::add_one(num));
}

```

示例 14-12：使用来自 `adder` `crate` 的库 `crate add-one`

在 `adder` 目录下运行 `cargo build` 来构建 `adder` `crate`！

```

$ cargo build
Compiling add-one v0.1.0 (file:///projects/adder/add-one)
Compiling adder v0.1.0 (file:///projects/adder)
Finished dev [unoptimized + debuginfo] target(s) in 0.68 secs

```

注意这会构建 `adder` `crate` 和 `adder/add-one` 中的 `add-one` `crate`。现在 `adder` 目录中应该有这些文件：

```

├─ Cargo.lock
├─ Cargo.toml
├─ add-one
│ └─ Cargo.toml
│ └─ src
│ └─ lib.rs
├─ src
│ └─ main.rs
└─ target

```

工作空间在顶级目录有一个 `target` 目录；`add-one` 并没有自己的 `target` 目录。即使进入 `add-one` 目录运行 `cargo build`，构建结果也位于 `adder/target` 而不是 `adder/add-one/target`。因为工作空间中的 `crate` 之间相互依赖。如果每个 `crate` 有其自己的 `target` 目录，为了在自己的 `target` 目录中生成构建结果，工作空间中的每一个 `crate` 都不得不相互重新编译其他 `crate`。通过共享一个 `target` 目录，工作空间可以避免其他 `crate` 多余的重复构建。

## 在工作空间中依赖外部 `crate`



还需注意的是工作空间只有一个 **Cargo.lock**，而不是拥有一个顶级的 **Cargo.lock** 和一个 **add-one/Cargo.lock**。这确保了所有的 **crate** 都使用完全相同版本的依赖。如果在 **Cargo.toml** 和 **add-one/Cargo.toml** 中都增加 **rand** **crate**，则 **Cargo** 会将其都解析为同一版本并记录到唯一的 **Cargo.lock** 中。使得工作空间中的所有 **crate** 都使用相同的依赖意味着其中的 **crate** 都是相互减重的。现在就让我们来试一试。

让我们在 **add-one/Cargo.toml** 中的 **[dependencies]** 部分增加 **rand** **crate** 以便能够在 **add-one** **crate** 中使用 **rand** **crate**：

文件名: **add-one/Cargo.toml**

```
[dependencies]

rand = "0.3.14"
```

现在就可以在 **add-one/src/lib.rs** 中增加 **extern crate rand;** 了，接着在 **adder** 目录运行 **cargo build** 构建整个工作空间就会引入并编译 **rand** **crate**：

```
$ cargo build
 Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading rand v0.3.14
 ...snip...
 Compiling rand v0.3.14
 Compiling add-one v0.1.0 (file:///projects/adder/add-one)
 Compiling adder v0.1.0 (file:///projects/adder)
 Finished dev [unoptimized + debuginfo] target(s) in 10.18 secs
```

现在顶级的 **Cargo.lock** 包含了 **add-one** 的 **rand** 依赖的信息。然而，即使 **rand** 被用于工作空间的某处，也不能在其他 **crate** 中使用它，除非也在他们的 **Cargo.toml** 中加入 **rand**。例如，如果在顶级的 **adder** **crate** 的 **src/main.rs** 中增加 **extern crate rand;**，则会得到一个错误：

```
$ cargo build
 Compiling adder v0.1.0 (file:///projects/adder)
 error[E0463]: can't find crate for `rand`
 --> src/main.rs:1:1
 |
 1 | extern crate rand;
 | ^^^^^^^^^^^^^^^^^^ can't find crate
```

为了修复这个错误，修改顶级 **adder** **crate** 的 **Cargo.toml** 来表明 **rand** 也是这个 **crate** 的依赖。构建 **adder** **crate** 会将 **rand** 加入到 **Cargo.lock** 中 **adder** 的依赖列表中，但是这并不会下载 **rand** 的额外拷贝。**Cargo** 确保了工作空间中任何使用 **rand** 的 **crate** 都采用相同的版本。在整个工作空间中使用相同版本的 **rand** 节省了空间，因为这样就无需多个拷贝并确保了工作空间中的 **crate** 将是相互兼容的。

## 为工作空间增加测试

作为另一个提升，让我们为 **add\_one** **crate** 中的 **add\_one::add\_one** 函数增加一个测试：

文件名: **add-one/src/lib.rs**

```
pub fn add_one(x: i32) -> i32 {
 x + 1
}

#[cfg(test)]
mod tests {
 use super::*;
```

```
#[test]
fn it_works() {
 assert_eq!(3, add_one(2));
}
}
```

在顶级 **adder** 目录运行 `cargo test` :

```
$ cargo test
Compiling adder v0.1.0 (file:///projects/adder)
Finished dev [unoptimized + debuginfo] target(s) in 0.27 secs
Running target/debug/adder-f0253159197f7841

running 0 tests

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

等等，零个测试？我们刚刚增加了一个测试！如果我们观察输出，就不难发现在工作空间中的 `cargo test` 只运行顶级 **crate** 的测试。为了运行工作空间中所有 **crate** 的测试，需要使用 `--all` 参数：

```
$ cargo test --all
Finished dev [unoptimized + debuginfo] target(s) in 0.37 secs
Running target/debug/deps/add_one-abcabcabc

running 1 test
test tests::it_works ... ok

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

Running target/debug/deps/adder-abcabcabc

running 0 tests

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

Doc-tests add-one

running 0 tests

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

当传递了 `--all` 时，`cargo test` 会运行工作空间中所有 **crate** 的测试。也可以选择在顶级目录运行工作空间中特定 **crate** 的测试，通过使用 `-p` 参数并指定希望测试的 **crate** 的名称：

```
$ cargo test -p add-one
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running target/debug/deps/add_one-b3235fea9a156f74

running 1 test
test tests::it_works ... ok

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

Doc-tests add-one

running 0 tests

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

输出显示了 `cargo test` 只运行了 `add-one crate` 的测试而没有运行 `adder crate` 的测试。

如果你选择向 `crates.io` 发布工作空间中的 `crate`，每一个工作空间中的 `crate` 将会单独发布。`cargo publish` 命令并没有 `--all` 或者 `-p` 参数，所以必须进入每一个 `crate` 的目录并运行 `cargo publish` 来发布工作空间中的每一个 `crate`。

现在尝试以类似 `add-one crate` 的方式向工作空间增加 `add-two crate` 来作为更多的练习！

随着项目增长，考虑使用工作空间：每一个更小的组件比一大块代码要容易理解。将 `crate` 保持在工作空间中易于协调他们的改变，如果他们一起运行并经常需要同时被修改的话。

## 使用 `cargo install` 从 **Crates.io** 安装二进制文件

[ch14-04-installing-binaries.md](#)

commit 6e53771a409794d9933c2a31310d78149b7e0534

`cargo install` 命令用于在本地安装和使用二进制 `crate`。它并不打算替换系统中的包；它意在作为一个方便 **Rust** 开发者们安装其他人已经在 **crates.io** 上共享的工具的手段。只有拥有二进制目标文件的包能够被安装。二进制目标文件是在 `crate` 有 `src/main.rs` 或者其他指定为二进制文件时所创建的可执行程序，这不同于自身不能执行但适合包含在其他程序中的库目标。通常 `crate` 的 `README` 文件中有该 `crate` 是库、二进制目标还是两者都是的信息。

所有来自 `cargo install` 的二进制文件都安装到 **Rust** 安装根目录的 `bin` 文件夹中。如果你使用 `rustup.rs` 安装的 **Rust** 且没有自定义任何配置，这将是 `$HOME/.cargo/bin`。将这个目录添加到 `$PATH` 环境变量中就能够运行通过 `cargo install` 安装的程序了。

例如，第十二章提到的叫做 `ripgrep` 的用于搜索文件的 `grep` 的 **Rust** 实现。如果想要安装 `ripgrep`，可以运行如下：

```
$ cargo install ripgrep
Updating registry `https://github.com/rust-lang/crates.io-index`
Downloading ripgrep v0.3.2
...snip...
 Compiling ripgrep v0.3.2
 Finished release [optimized + debuginfo] target(s) in 97.91 secs
Installing ~/.cargo/bin/rg
```

最后一行输出展示了安装的二进制文件的位置和名称，在这里 `ripgrep` 被命名为 `rg`。只要你像上面提到的那样将安装目录加入 `$PATH`，就可以运行 `rg --help` 并开始使用一个更快更 **Rust** 的工具来搜索文件了！

# Cargo 自定义扩展命令

[ch14-05-extending-cargo.md](#)

commit 6e53771a409794d9933c2a31310d78149b7e0534

Cargo 被设计为可扩展的，通过新的子命令而无须修改 Cargo 自身。如果 `$PATH` 中有类似 `cargo-something` 的二进制文件，就可以通过 `cargo something` 来像 Cargo 子命令一样运行它。像这样的自定义命令也可以运行 `cargo --list` 来展示出来。能够通过 `cargo install` 向 Cargo 安装扩展并可以如内建 Cargo 工具那样运行他们是 Cargo 设计上的一个非常方便的优点！

## 总结

通过 Cargo 和 crates.io 来分享代码是使得 Rust 生态环境可以用于许多不同的任务的重要组成部分。Rust 的标准库是小而稳定的，不过 crate 易于分享和使用，并采用一个不同语言自身的时间线来提供改进。不要羞于在 crates.io 上共享对你有用的代码；因为它很有可能对别人也很有用！

# 智能指针

ch15-00-smart-pointers.md

commit d06a6a181fd61704cbf7feb55bc61d518c6469f9

指针（*pointer*）是一个常见的编程概念，它代表一个指向储存其他数据的位置。第四章学习了 Rust 的引用；他们是一类很平常的指针，以 `&` 符号为标志并借用了他们所指向的值。智能指针（*smart pointers*）是一类数据结构，他们的表现类似指针，但是也拥有额外的元数据和功能，比如说引用计数。智能指针模式起源于 C++。在 Rust 中，普通引用和智能指针的一个额外的区别是引用是一类只借用数据的指针；相反大部分情况，智能指针拥有他们指向的数据。

本书中已经出现过一些智能指针，虽然当时我们并不这么称呼他们。例如在某种意义上说，第八章的 `String` 和 `Vec<T>` 都是智能指针。他们拥有一些数据并允许你修改他们，并带有元数据（比如他们的容量）和额外的功能或保证（`String` 的数据总是有效的 UTF-8 编码）。智能指针区别于常规结构体的显著特性在于他们实现了 `Deref` 和 `Drop` trait，而本章会讨论这些 trait 以及为什么对于智能指针来说他们很重要。

考虑到智能指针是一个在 Rust 经常被使用的通用设计模式，本章并不会覆盖所有现存的智能指针。很多库都有自己的智能指针而你也可以编写属于你自己的。这里将会讲到的是来自标准库中最常用的一些：

- `Box<T>`，用于在堆上分配值
- `Rc<T>`，一个引用计数类型，其数据可以有多个所有者
- `RefCell<T>`，其本身并不是智能指针，不过它管理智能指针 `Ref` 和 `RefMut` 的访问，在运行时而不是在编译时执行借用规则。

同时我们还将涉及：

- 内部可变性（*interior mutability*）模式，当一个不可变类型暴露出改变其内部值的 API，这时借用规则适用于运行时而不是编译时。
- 引用循环，它如何会泄露内存，以及如何避免他们

让我们开始吧！

## Box<T> 在堆上存储数据，并且可确定大小

ch15-01-box.md

commit 68267b982a226fa252e9afa1a5029396ccf5fa03

最简单直接的智能指针是 **box**，它的类型是 `Box<T>`。**box** 允许你将一个值放在堆上而不是栈上。留在栈上的则是指向堆数据的指针。如果你想回顾一下栈与堆的区别请参考第四章。

除了数据被储存在堆上而不是栈上之外，**box** 没有性能损失，不过也没有很多额外的功能。他们多用于如下场景：

- 当有一个在编译时未知大小的类型，而又需要在需要确切大小的上下文中使用这个类型的值的时候
- 当有大量数据并希望确保数据不被拷贝的情况下转移所有权的时候
- 当希望拥有一个值并只关心它的类型是否实现了特定 **trait** 而不是其具体类型的时候

我们将在本部分的余下内容中展示第一种应用场景。作为对另外两个情况更详细的说明：在第二种情况中，转移大量数据的所有权可能会花费很长的时间，因为数据在栈上进行了拷贝。为了改善这种情况下的性能，可以通过 **box** 将这些数据储存在堆上。接着，只有少量的指针数据在栈上被拷贝。第三种情况被称为 **trait** 对象（*trait object*），第十七章刚好有一整个部分专门讲解这个主题。所以这里所学的内容会在第十七章再次用上！

### 使用 Box<T> 在堆上储存数据

在开始 `Box<T>` 的用例之前，让我们熟悉一下语法和如何与储存在 `Box<T>` 中的值交互。

示例 15-1 展示了如何使用 **box** 在堆上储存一个 `i32`：

文件名: src/main.rs

```
fn main() {
 let b = Box::new(5);
 println!("b = {}", b);
}
```

示例 15-1：使用 **box** 在堆上储存一个 `i32` 值

这里定义了变量 `b`，其值是一个指向被分配在堆上的值 `5` 的 `Box`。这个程序会打印出 `b = 5`。在这个例子中，我们可以像数据是储存在栈上的那样访问 **box** 中的数据。正如任何拥有数据所有权的值那样，当像 `b` 这样的 **box** 在 `main` 的末尾离开作用域时，它将被释放。这个释放过程作用于 **box** 本身（位于栈上）和它所指向的数据（位于堆上）。

将一个单独的值存放在堆上并不是很有意义，所以像示例 15-1 这样单独使用 **box** 并不常见。将像单个 `i32` 这样的值储存在栈上，也就是其默认存放的地方在大部分使用场景中更为合适。让我们看看一个 **box** 定义一个不使用 **box** 时无法定义的类型例子。

### box 允许创建递归类型

Rust 需要在编译时知道类型占用多少空间。一种无法在编译时知道大小的类型是 递归类型（*recursive type*），其值的一部分可以是相同类型的另一个值。这种值的嵌套理论上可以无限的进行下去，所以 Rust 不知道递归类型需要多少空间。不过 **box** 有一个已知的大小，所以通过在循环类型定义中插入 **box**，就可以创建递归类型了。

让我们探索一下 `cons list`，一个函数式编程语言中的常见类型，来展示这个（递归类型）概念。除了递归之外，我们将要定义的 `cons list` 类型是很直白的，所以这个例子中的概念在任何遇到更为复杂的涉及到递归类型的场景时都很实用。

`cons list` 是一个每一项都包含两个部分的列表：当前项的值和下一项。其最后一项值包含一个叫做 `Nil` 的值并没有下一项。

## cons list 的更多内容

`cons list` 是一个来源于 Lisp 编程语言及其方言的数据结构。在 Lisp 中，`cons` 函数（“construct function”的缩写）利用两个参数来构造一个新的列表，他们通常是一个单独的值和另一个列表。

`cons` 函数的概念涉及到更通用的函数式编程术语：“将 `x` 与 `y` 连接”通常意味着构建一个新的容器而将 `x` 的元素放在新容器的开头，其后则是容器 `y` 的元素。

`cons list` 通过递归调用 `cons` 函数产生。代表递归的终止条件（base case）的规范名称是 `Nil`，它宣布列表的终止。注意这不同于第六章中的“null”或“nil”的概念，他们代表无效或缺失的值。

注意虽然函数式编程语言经常使用 `cons list`，但是它并不是一个 Rust 中常见的类型。大部分在 Rust 中需要列表的时候，`Vec<T>` 是一个更好的选择。其他更为复杂的递归数据类型 确实 在 Rust 的很多场景中很有用，不过通过以 `cons list` 作为开始，我们可以探索如何使用 `box` 毫不费力的定义一个递归数据类型。

示例 15-2 包含一个 `cons list` 的枚举定义。注意这还不能编译因为这个类型没有已知的大小，之后我们会展示：

文件名: `src/main.rs`

```
enum List {
 Cons(i32, List),
 Nil,
}
```

示例 15-2：第一次尝试定义一个代表 `i32` 值的 `cons list` 数据结构的枚举

注意：出于示例的我们选择实现一个只存放 `i32` 值的 `cons list`。也可以用泛型实现它，正如第十章讲到的，来定义一个可以存放任何类型值的 `cons list` 类型。

使用这个 `cons list` 来储存列表 `1, 2, 3` 将看起来如示例 15-3 所示：

文件名: `src/main.rs`

```
use List::{Cons, Nil};

fn main() {
 let list = Cons(1, Cons(2, Cons(3, Nil)));
}
```

示例 15-3：使用 `List` 枚举储存列表 `1, 2, 3`

第一个 `Cons` 储存了 `1` 和另一个 `List` 值。这个 `List` 是另一个包含 `2` 的 `Cons` 值和下一个 `List` 值。接着又有另一个存放了 `3` 的 `Cons` 值和最后一个值为 `Nil` 的 `List`，非递归成员代表了列表的结尾。

如果尝试编译上面的代码，会得到如示例 15-4 所示的错误：

```
error[E0072]: recursive type `List` has infinite size
-->
|
1 | enum List {
| ^^^^^^^ recursive type has infinite size
2 | Cons(i32, List),
| ----- recursive without indirection
|
= help: insert indirection (e.g., a `Box`, `Rc`, or `&`) at some point to
```



```
make `List` representable
```

示例 15-4：尝试定义一个递归枚举时得到的错误

这个错误表明这个类型“有无限的大小”。其原因是 `List` 的一个成员被定义为是递归的：它直接存放了另一个相同类型的值。这意味着 `Rust` 无法计算为了存放 `List` 值到底需要多少空间。让我们一点一点来看：首先了解一下 `Rust` 如何决定需要多少空间来存放一个非递归类型。

## 计算非递归类型的大小

回忆一下第六章讨论枚举定义时示例 6-2 中定义的 `Message` 枚举：

```
enum Message {
 Quit,
 Move { x: i32, y: i32 },
 Write(String),
 ChangeColor(i32, i32, i32),
}
```

当 `Rust` 需要知道要为 `Message` 值分配多少空间时，它可以检查每一个成员并发现 `Message::Quit` 并不需要任何空间，`Message::Move` 需要足够储存两个 `i32` 值的空间，依此类推。因此，`Message` 值所需的空间等于储存其最大成员的空间大小。

与此相对当 `Rust` 编译器检查像示例 15-2 中的 `List` 这样的递归类型时会发生什么呢。编译器尝试计算出储存一个 `List` 枚举需要多少内存，并开始检查 `Cons` 成员，那么 `Cons` 需要的空间等于 `i32` 的大小加上 `List` 的大小。为了计算 `List` 需要多少内存，它检查其成员，从 `Cons` 成员开始。`Cons` 成员储存了一个 `i32` 值和一个 `List` 值，这样的计算将无限进行下去，如图 15-5 所示：

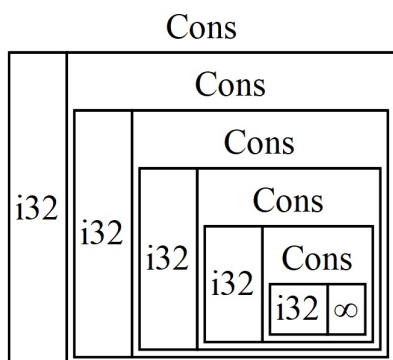


图 15-5：一个包含无限个 `Cons` 成员的无限 `List`

## 使用 `Box<T>` 给递归类型一个已知的大小

`Rust` 无法计算出要为定义为递归的类型分配多少空间，所以编译器给出了示例 15-4 中的错误。这个错误也包括了有用的建议：

```
= help: insert indirection (e.g., a `Box`, `Rc`, or `&`) at some point to
 make `List` representable
```

在建议中，“indirection”意味着不同于直接储存一个值，我们将间接的储存一个指向值的指针。

因为 `Box<T>` 是一个指针，我们总是知道它需要多少空间：指针的大小并不会根据其指向的数据量而改变。

所以可以将 `Box` 放入 `Cons` 成员中而不是直接存放另一个 `List` 值。`Box` 会指向另一个位于堆上的 `List` 值，而不是存放在 `Cons` 成员中。从概念上讲，我们仍然有一个通过在其中“存放”其他列表创建的列表，不过现在实现这个概念的方式更像是一个项挨着另一项，而不是一项包含另一项。

我们可以修改示例 15-2 中 `List` 枚举的定义和示例 15-3 中对 `List` 的应用，如示例 15-6 所示，这是可以编译的：

文件名: `src/main.rs`

```
enum List {
 Cons(i32, Box<List>),
 Nil,
}

use List::{Cons, Nil};

fn main() {
 let list = Cons(1,
 Box::new(Cons(2,
 Box::new(Cons(3,
 Box::new(Nil))))));
}
```

示例 15-6：为了已知大小而使用 `Box<T>` 的 `List` 定义

`Cons` 成员将会需要一个 `i32` 的大小加上储存 `box` 指针数据的空间。`Nil` 成员不储存值，所以它比 `Cons` 成员需要更少的空间。现在我们知道了任何 `List` 值最多需要一个 `i32` 加上 `box` 指针数据的大小。通过使用 `box`，打破了这无限递归的连锁，这样编译器就能够计算出储存 `List` 值需要的大小了。图 15-7 展示了现在 `Cons` 成员看起来像什么：

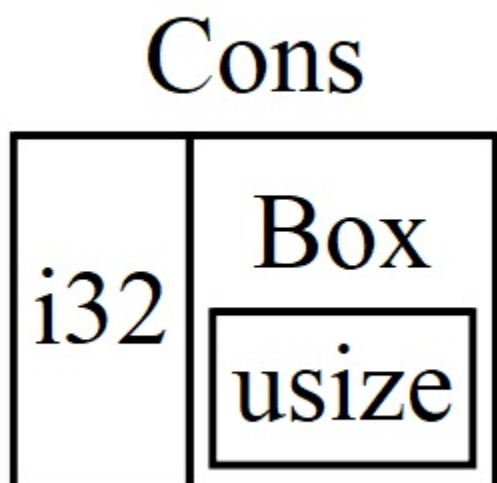


图 15-7：因为 `Cons` 存放一个 `Box` 所以 `List` 不是无限大小的了

`box` 只提供了间接存储和堆分配；他们并没有任何其他特殊的功能，比如我们将会见到的其他智能指针。他们也没有这些特殊功能带来的性能损失，所以他们可以用于像 `cons list` 这样间接存储是唯一所需功能的场景。我们还将将在第十七章看到 `box` 的更多应用场景。

`Box<T>` 类型是一个智能指针，因为它实现了 `Deref trait`，它允许 `Box<T>` 值被当作引用对待。当 `Box<T>` 值离开作用域时，由于 `Box<T>` 类型 `Drop trait` 的实现，`box` 所指向的堆数据也会被清除。让我们更详细的探索一下这两个 `trait`；这些 `trait` 在本章余下讨论的其他智能指针所提供的功能中将会更为重要。

## Deref Trait 允许通过引用访问数据

ch15-02-deref.md

commit d06a6a181fd61704cbf7feb55bc61d518c6469f9

第一个智能指针相关的重要 **trait** 是 **Deref**，它允许我们重载 **\***，解引用运算符（不同于乘法运算符或全局引用运算符）。重载智能指针的 **\*** 能使访问其持有的数据更为方便，在本章结束前谈到解引用强制多态（**deref coercions**）时我们会说明方便意味着什么。

第八章的哈希 **map** 的“根据旧值更新一个值”部分简要的提了解引用运算符。当时有一个可变引用，而我們希望改变这个引用所指向的值。为此，首先我们必须解引用。这是另一个使用 **i32** 值引用的例子：

```
let mut x = 5;
{
 let y = &mut x;

 *y += 1
}

assert_eq!(6, x);
```

我们使用 **\*y** 来访问可变引用 **y** 所指向的数据，而不是可变引用本身。接着可以修改它的数据，在这里是对其加一。

引用并不是智能指针，他们只是引用指向的一个值，所以这个解引用操作是很直接的。智能指针还会储存指针或数据的元数据。当解引用一个智能指针时，我们只想要数据，而不需要元数据，因为解引用一个常规的引用只能给我们数据而不是元数据。我们希望能在使用常规引用的地方也能使用智能指针。为此，可以通过实现 **Deref trait** 来重载 **\*** 运算符的行为。

示例 15-7 展示了一个定义为储存 **mp3** 数据和元数据的结构体通过 **Deref trait** 来重载 **\*** 的例子。**Mp3**，在某种意义上是一个智能指针：它拥有包含音频的 **Vec<u8>** 数据。另外，它储存了一些可选的元数据，在这个例子中是音频数据中艺术家和歌曲的名称。我们希望能够方便的访问音频数据而不是元数据，所以需要实现 **Deref trait** 来返回音频数据。实现 **Deref trait** 需要一个叫做 **deref** 的方法，它借用 **self** 并返回其内部数据：

文件名: src/main.rs

```
use std::ops::Deref;

struct Mp3 {
 audio: Vec<u8>,
 artist: Option<String>,
 title: Option<String>,
}

impl Deref for Mp3 {
 type Target = Vec<u8>;

 fn deref(&self) -> &Vec<u8> {
 &self.audio
 }
}

fn main() {
 let my_favorite_song = Mp3 {
 // we would read the actual audio data from an mp3 file
 audio: vec![1, 2, 3],
 artist: Some(String::from("Nirvana")),
 }
```

```

 title: Some(String::from("Smells Like Teen Spirit")),
 };

 assert_eq!(vec![1, 2, 3], *my_favorite_song);
}

```

示例 15-7：一个存放 mp3 文件数据和元数据的结构体上的 `Deref trait` 实现

大部分代码看起来都比较熟悉：一个结构体、一个 `trait` 实现、和一个创建了结构体实例的 `main` 函数。其中有一部分我们还未全面的讲解：类似于第十三章学习迭代器 `trait` 时出现的 `type Item`，`type Target = T`；语法用于定义关联类型，第十九章会更详细的介绍。不必过分担心例子中的这一部分；它只是一个稍显不同的定义泛型参数的方式。

在 `assert_eq!` 中，我们验证 `vec![1, 2, 3]` 是否为 `Mp3` 实例 `*my_favorite_song` 解引用的值，结果正是如此，因为我们实现了 `deref` 方法来返回音频数据。如果没有为 `Mp3` 实现 `Deref trait`，`Rust` 将不会编译

`*my_favorite_song`：会出现错误说 `Mp3` 类型不能被解引用。

没有 `Deref trait` 的话，编译器只能解引用 `&` 引用，而 `my_favorite_song` 并不是（它是一个 `Mp3` 结构体）。通过 `Deref trait`，编译器知道实现了 `Deref trait` 的类型有一个返回引用的 `deref` 方法（在这个例子中，是 `&self.audio` 因为示例 15-7 中的 `deref` 的定义）。所以为了得到一个 `*` 可以解引用的 `&` 引用，编译器将 `*my_favorite_song` 展开为如下：

```

*(my_favorite_song.deref())

```

其结果就是 `self.audio` 中的值。`deref` 返回一个引用并接下来必需解引用而不是直接返回值的原因是所有权：如果 `deref` 方法直接返回值而不是引用，其值将被移出 `self`。和大部分使用解引用运算符的地方相同，这里并不想获取 `my_favorite_song.audio` 的所有权。

注意将 `*` 替换为 `deref` 调用和 `*` 调用的过程在每次使用 `*` 的时候都会发生一次。`*` 的替换并不会无限递归进行。最终的数据类型是 `Vec<u8>`，它与示例 15-7 中 `assert_eq!` 的 `vec![1, 2, 3]` 相匹配。

## 函数和方法的隐式解引用强制多态

`Rust` 倾向于偏爱明确而不是隐晦，不过一个情况下这并不成立，就是函数和方法的参数的解引用强制多态（*`deref coercions`*）。解引用强制多态会自动的将指针或智能指针的引用转换为指针内容的引用。解引用强制多态发生于当传递给函数的参数类型不同于函数签名中定义参数类型的时候。解引用强制多态的加入使得 `Rust` 调用函数或方法时无需很多显式使用 `&` 和 `*` 的引用和解引用。

使用示例 15-7 中的 `Mp3` 结构体，如下是一个获取 `u8 slice` 并压缩 `mp3` 音频数据的函数签名：

```

fn compress_mp3(audio: &[u8]) -> Vec<u8> {
 // the actual implementation would go here
}

```

如果 `Rust` 没有解引用强制多态，为了使用 `my_favorite_song` 中的音频数据调用此函数，必须写成：

```

compress_mp3(my_favorite_song.audio.as_slice())

```

也就是说，必须明确表明需要 `my_favorite_song` 中的 `audio` 字段而且我们希望有一个 `slice` 来引用这整个 `Vec<u8>`。如果有很多地方需要用相同的方式处理 `audio` 数据，那么 `.audio.as_slice()` 就显得冗长重复了。

然而，因为解引用强制多态和 `Mp3` 的 `Deref trait` 实现，我们可以使用如下代码使用 `my_favorite_song` 中的数据调用这个函数：

```
let result = compress_mp3(&my_favorite_song);
```

只有 `&` 和实例，好的！我们可以把智能指针当成普通的引用那样使用。也就是说解引用强制多态意味着 **Rust** 利用了 **Deref** 实现的优势：**Rust** 知道 `Mp3` 实现了 **Deref trait** 并从 `deref` 方法返回 `&Vec<u8>`。它也知道标准库实现了 `Vec<T>` 的 **Deref trait**，其 `deref` 方法返回 `&[T]`（我们也可以通过查阅 `Vec<T>` 的 **API** 文档来发现这一点）。所以，在编译时，**Rust** 会发现它可以调用两次 `Deref::deref` 来将 `&Mp3` 变成 `&Vec<u8>` 再变成 `&[T]` 来满足 `compress_mp3` 的签名。这意味着我们可以少写一些代码！**Rust** 会多次分析 `Deref::deref` 的返回值类型直到它满足参数的类型，只要相关类型实现了 **Deref trait**。这些间接转换在编译时进行，所以利用解引用强制多态并没有运行时惩罚！

类似于如何使用 **Deref trait** 重载 `&T` 的 `*` 运算符，**DerefMut trait** 用于重载 `&mut T` 的 `*` 运算符。

**Rust** 在发现类型和 **trait** 实现满足三种情况时会进行解引用强制多态：

- 当 `T: Deref<Target=U>` 时从 `&T` 到 `&U`。
- 当 `T: DerefMut<Target=U>` 时从 `&mut T` 到 `&mut U`。
- 当 `T: Deref<Target=U>` 时从 `&mut T` 到 `&U`。

头两个情况除了可变性之外是相同的：如果有一个 `&T`，而 `T` 实现了返回 `U` 类型的 **Deref**，则可以直接得到 `&U`。对于可变引用也是一样。最后一个有些微妙：如果有一个可变引用，它也可以强转为一个不可变引用。反之则是 不可能 的：不可变引用永远也不能强转为可变引用。

**Deref trait** 对于智能指针模式十分重要的原因在于智能指针可以被看作普通引用并被用于期望使用普通引用的地方。例如，无需重新定义方法和函数来直接获取智能指针。

## Drop Trait 运行清理代码

ch15-03-drop.md

commit 3f2a1bd8dbb19cc48b210fc4fb35c305c8d81b56

对于智能指针模式来说另一个重要的 `trait` 是 `Drop`。`Drop` 运行我们在值要离开作用域时执行一些代码。智能指针在被丢弃时会执行一些重要的清理工作，比如释放内存或减少引用计数。更一般的来讲，数据类型可以管理多于内存的资源，比如文件或网络连接，而使用 `Drop` 在代码处理完他们之后释放这些资源。我们在智能指针上下文中讨论 `Drop` 是因为其功能几乎总是用于实现智能指针。

在其他一些语言中，我们不得不记住在每次使用完智能指针实例后调用清理内存或资源的代码。如果忘记的话，运行代码的系统可能会因为负荷过重而崩溃。在 `Rust` 中，可以指定一些代码应该在值离开作用域时被执行，而编译器会自动插入这些代码。这意味着无需记住在所有处理完这些类型实例后调用清理代码，而仍然不会泄露资源！

指定在值离开作用域时应该执行的代码的方式是实现 `Drop trait`。`Drop trait` 要求我们实现一个叫做 `drop` 的方法，它获取一个 `self` 的可变引用。

示例 15-8 展示了并没有实际功能的结构体 `CustomSmartPointer`，不过我们会在创建实例之后打印出 `CustomSmartPointer created.`，而在实例离开作用域时打印出 `Dropping CustomSmartPointer!`，这样就能看出每一段代码是何时被执行的。实际的项目中，我们应该在 `drop` 中清理任何智能指针运行所需要的资源，而不是这个例子中的 `println!` 语句：

Filename: src/main.rs

```
struct CustomSmartPointer {
 data: String,
}

impl Drop for CustomSmartPointer {
 fn drop(&mut self) {
 println!("Dropping CustomSmartPointer!");
 }
}

fn main() {
 let c = CustomSmartPointer { data: String::from("some data") };
 println!("CustomSmartPointer created.");
 println!("Wait for it...");
}
```

Listing 15-8: A `CustomSmartPointer` struct that implements the `Drop` trait, where we could put code that would clean up after the `CustomSmartPointer`.

`Drop trait` 位于 `prelude` 中，所以无需导入它。`drop` 方法的实现调用了 `println!`：这里是你需要放入实际关闭套接字代码的地方。在 `main` 函数中，我们创建一个 `CustomSmartPointer` 的新实例并打印出 `CustomSmartPointer created.` 以便在运行时知道代码运行到此处。在 `main` 的结尾，`CustomSmartPointer` 的实例会离开作用域。注意我们没有显式调用 `drop` 方法：

当运行这个程序，我们会看到：

```
CustomSmartPointer created.
Wait for it...
Dropping CustomSmartPointer!
```

被打印到屏幕上，它展示了 **Rust** 在实例离开作用域时自动调用了 `drop`。

可以使用 `std::mem::drop` 函数来在值离开作用域之前丢弃它。这通常是不必要的；整个 **Drop trait** 的要点在于它自动的帮我们处理清理工作。在第十六章讲到并发时我们会看到一个需要在离开作用域之前丢弃值的例子。现在知道这是可能的即可，`std::mem::drop` 位于 `prelude` 中所以可以如示例 15-9 所示直接调用 `drop`：

Filename: src/main.rs

```
fn main() {
 let c = CustomSmartPointer { data: String::from("some data") };
 println!("CustomSmartPointer created.");
 drop(c);
 println!("Wait for it...");
}
```

Listing 15-9: Calling `std::mem::drop` to explicitly drop a value before it goes out of scope

运行这段代码会打印出如下内容，因为 `Dropping CustomSmartPointer!` 在 `CustomSmartPointer created.` 和 `Wait for it...` 之间被打印出来，表明析构代码被执行了：

```
CustomSmartPointer created.
Dropping CustomSmartPointer!
Wait for it...
```

注意不允许直接调用我们定义的 `drop` 方法：如果将示例 15-9 中的 `drop(c)` 替换为 `c.drop()`，会得到一个编译错误表明 `explicit destructor calls not allowed`。不允许直接调用 `Drop::drop` 的原因是 **Rust** 在值离开作用域时会自动插入 `Drop::drop`，这样就会丢弃值两次。丢弃一个值两次可能会造成错误或破坏内存，所以 **Rust** 就不允许这么做。相应的可以调用 `std::mem::drop`，它的定义是：

```
pub mod std {
 pub mod mem {
 pub fn drop<T>(x: T) { }
 }
}
```

这个函数对于 `T` 是泛型的，所以可以传递任何值。这个函数的函数体并没有任何实际内容，所以它也不会利用其参数。这个空函数的作用在于 `drop` 获取其参数的所有权，它意味着在这个函数结尾 `x` 离开作用域时 `x` 会被丢弃。

使用 **Drop trait** 实现指定的代码在很多方面都使得清理值变得方便和安全：比如可以使用它来创建我们自己的内存分配器！通过 **Drop trait** 和 **Rust** 所有权系统，就无需担心之后清理代码，因为 **Rust** 会自动考虑这些问题。如果代码在值仍被使用时就清理它会出现编译错误，因为所有权系统确保了引用总是有效的，这也就保证了 `drop` 只会在值不再被使用时被调用一次。

现在我们学习了 `Box<T>` 和一些智能指针的特性，让我们聊聊一些其他标准库中定义的拥有各种实用功能的智能指针。



## Rc<T> 引用计数智能指针

ch15-04-rc.md

commit 3f2a1bd8dbb19cc48b210fc4fb35c305c8d81b56

大部分情况下所有权是非常明确的：可以准确的知道哪个变量拥有某个值。然而并不总是如此；有时确实可能需要多个所有者。为此，**Rust** 有一个叫做 `Rc<T>` 的类型。它的名字是引用计数（*reference counting*）的缩写。引用计数意味着它记录一个值引用的数量来知晓这个值是否仍在被使用。如果这个值有零个引用，就知道可以在没有有效引用的前提下清理这个值。

根据现实生活场景来想象的话，它就像一个客厅的电视。当一个人进来看电视时，他打开电视。其他人也会进来看电视。当最后一个人离开房间时，他关掉电视因为它不再被使用了。如果某人在其他人还在看的时候关掉了电视，正在看电视人肯定会抓狂的！

`Rc<T>` 用于当我们希望在堆上分配一些内存供程序的多个部分读取，而且无法在编译时确定程序的那一部分会最后结束使用它。如果我们知道的话那么常规的所有权规则会在编译时强制起作用。

注意 `Rc<T>` 只能用于单线程场景；下一章并发会涉及到如何在多线程程序中进行引用计数。如果尝试在多线程中使用 `Rc<T>` 则会得到一个编译错误。

### 使用 `Rc<T>` 分享数据

让我们回到示例 15-5 中的 `cons list` 例子。在示例 15-11 中尝试使用 `Box<T>` 定义的 `List`。首先创建了一个包含 5 接着是 10 的列表实例。之后我们想要创建另外两个列表：一个以 3 开始并后接第一个包含 5 和 10 的列表，另一个以 4 开始其后也是第一个列表。换句话说，我们希望这两个列表共享第三个列表的所有权，概念上类似于图 15-10：

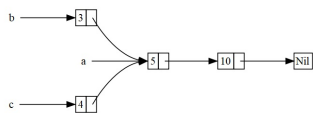


Figure 15-10: Two lists, `b` and `c`, sharing ownership of a third list, `a`

尝试使用 `Box<T>` 定义的 `List` 并不能工作，如示例 15-11 所示：

Filename: src/main.rs

```
enum List {
 Cons(i32, Box<List>),
 Nil,
}

use List::{Cons, Nil};

fn main() {
 let a = Cons(5,
 Box::new(Cons(10,
 Box::new(Nil))));
```

```

 let b = Cons(3, Box::new(a));
 let c = Cons(4, Box::new(a));
}

```

Listing 15-11: Having two lists using `Box<T>` that try to share ownership of a third list won't work

编译会得出如下错误：

```

error[E0382]: use of moved value: `a`
 --> src/main.rs:13:30
 |
12 | let b = Cons(3, Box::new(a));
 | - value moved here
13 | let c = Cons(4, Box::new(a));
 | ^ value used here after move
 |
 = note: move occurs because `a` has type `List`, which does not
 implement the `Copy` trait

```

`Cons` 成员拥有其储存的数据，所以当创建 `b` 列表时将 `a` 的所有权移动到了 `b`。接着当再次尝试使用 `a` 创建 `c` 时，这不被允许因为 `a` 的所有权已经被移动。

相反可以改变 `Cons` 的定义来存放一个引用，不过接着必须指定生命周期参数，而且在构造列表时，也必须使列表中的每一个元素都至少与列表本身存在的一样久。否则借用检查器甚至都不会允许我们编译代码。

如示例 15-12 所示，可以将 `List` 的定义从 `Box<T>` 改为 `Rc<T>`：

Filename: src/main.rs

```

enum List {
 Cons(i32, Rc<List>),
 Nil,
}

use List::{Cons, Nil};
use std::rc::Rc;

fn main() {
 let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
 let b = Cons(3, a.clone());
 let c = Cons(4, a.clone());
}

```

Listing 15-12: A definition of `List` that uses `Rc<T>`

注意必须为 `Rc` 增加 `use` 语句因为它不在 `prelude` 中。在 `main` 中创建了存放 5 和 10 的列表并将其存放在一个叫做 `a` 的新的 `Rc` 中。接着当创建 `b` 和 `c` 时，我们对 `a` 调用了 `clone` 方法。

## 克隆 `Rc<T>` 会增加引用计数

之前我们见过 `clone` 方法，当时使用它来创建某些数据的完整拷贝。但是对于 `Rc<T>` 来说，它并不创建一个完整的拷贝。`Rc<T>` 存放了引用计数，也就是说，一个存在多少个克隆的计数器。让我们像示例 15-13 那样在创建 `c` 时增加一个内部作用域，并在不同的位置打印出关联函数 `Rc::strong_count` 的结果。`Rc::strong_count` 返回传递给它的 `Rc` 值的引用计数，而在本章的稍后部分介绍避免引用循环时讲到它为什么叫做 `strong_count`。

Filename: src/main.rs

```

enum List {

```

```

Cons(i32, Rc<List>),
Nil,
}
#
use List::{Cons, Nil};
use std::rc::Rc;
#
fn main() {
 let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))))
 println!("rc = {}", Rc::strong_count(&a));
 let b = Cons(3, a.clone());
 println!("rc after creating b = {}", Rc::strong_count(&a));
 {
 let c = Cons(4, a.clone());
 println!("rc after creating c = {}", Rc::strong_count(&a));
 }
 println!("rc after c goes out of scope = {}", Rc::strong_count(&a));
}

```

Listing 15-13: Printing out the reference count

这会打印出：

```

rc = 1
rc after creating b = 2
rc after creating c = 3
rc after c goes out of scope = 2

```

不难看出 `a` 的初始引用计数是一。接着每次调用 `clone`，计数会加一。当 `c` 离开作用域时，计数减一，这发生在 `Rc<T>` 的 `Drop` `trait` 实现中。这个例子中不能看到的是当 `b` 接着是 `a` 在 `main` 函数的结尾离开作用域时，包含 5 和 10 的列表的引用计数会是 0，这时列表将被丢弃。这个策略允许拥有多个所有者，而引用计数会确保任何所有者存在时这个值保持有效。

在本部分的开始，我们说 `Rc<T>` 只允许程序的多个部分读取 `Rc<T>` 中 `T` 的不可变引用。如果 `Rc<T>` 允许一个可变引用，我们将遇到第四章讨论的借用规则所不允许的问题：两个指向同一位置的可变借用会导致数据竞争和不一致。不过可变数据是非常有用的！在下一部分，我们将讨论内部可变性模式和 `RefCell<T>` 类型，它可以与 `Rc<T>` 结合起来处理不可变性的限制。

## RefCell<T> 和内部可变性模式

ch15-05-interior-mutability.md

commit 3f2a1bd8dbb19cc48b210fc4fb35c305c8d81b56

内部可变性 (*Interior mutability*) 是 Rust 中的一个设计模式，它允许你即使在有不可变引用时改变数据，这通常是借用规则所不允许。内部可变性模式涉及到在数据结构中使用 `unsafe` 代码来模糊 Rust 通常的可变性和借用规则。我们还未讲到不安全代码；第十九章会学习他们。内部可变性模式用于当你可以确保代码在运行时也会遵守借用规则，哪怕编译器也不能保证的情况。引入的 `unsafe` 代码将被封装进安全的 API 中，而外部类型仍然是不可变的。

让我们通过遵循内部可变性模式的 `RefCell<T>` 类型来开始探索。

### RefCell<T> 拥有内部可变性

不同于 `Rc<T>`，`RefCell<T>` 代表其数据的唯一的所有权。那么是什么让 `RefCell<T>` 不同于像 `Box<T>` 这样的类型呢？回忆一下第四章所学的借用规则：

1. 在任意给定时间，只能拥有如下中的一个：
  - 一个可变引用。
  - 任意数量的不可变引用。
2. 引用必须总是有效的。

对于引用和 `Box<T>`，借用规则的不可变性作用于编译时。对于 `RefCell<T>`，这些不可变性作用于运行时。对于引用，如果违反这些规则，会得到一个编译错误。而对于 `RefCell<T>`，违反这些规则会 `panic!`。

Rust 编译器执行的静态分析天生是保守的。代码的一些属性则不可能通过分析代码发现：其中最著名的就是停机问题 (Halting Problem)，这超出了本书的范畴，不过如果你感兴趣的话这是一个值得研究的有趣主题。

因为一些分析是不可能的，Rust 编译器在其不确定的时候甚至都不尝试猜测，所以说它是保守的而且有时会拒绝事实上不会违反 Rust 保证的正确的程序。换句话说，如果 Rust 接受不正确的程序，那么人们也就不会相信 Rust 所做的保证了。如果 Rust 拒绝正确的程序，会给程序员带来不便，但不会带来灾难。`RefCell<T>` 正是用于当你知道代码遵守借用规则，而编译器不能理解的时候。

类似于 `Rc<T>`，`RefCell<T>` 只能用于单线程场景。在并发章节会介绍如何在多线程程序中使用 `RefCell<T>` 的功能。现在所有你需要知道的就是如果尝试在多线程上下文中使用 `RefCell<T>`，会得到一个编译错误。

对于引用，可以使用 `&` 和 `&mut` 语法来分别创建不可变和可变的引用。不过对于 `RefCell<T>`，我们使用 `borrow` 和 `borrow_mut` 方法，它是 `RefCell<T>` 拥有的安全 API 的一部分。`borrow` 返回 `Ref` 类型的智能指针，而 `borrow_mut` 返回 `RefMut` 类型的智能指针。这两个类型实现了 `Deref` 所以可以被当作常规引用处理。`Ref` 和 `RefMut` 动态的借用所有权，而他们的 `Drop` 实现也动态的释放借用。

示例 15-14 展示了如何使用 `RefCell<T>` 来使函数不可变的和可变的借用它的参数。注意 `data` 变量使用 `let data` 而不是 `let mut data` 来声明为不可变的，而 `a_fn_that_mutably_borrows` 则允许可变的借用数据并修改它！

Filename: src/main.rs

```
use std::cell::RefCell;

fn a_fn_that_immutably_borrows(a: &i32) {
 println!("a is {}", a);
}

fn a_fn_that_mutably_borrows(b: &mut i32) {
```

```

 *b += 1;
}

fn demo(r: &RefCell<i32>) {
 a_fn_that_immutably_borrows(&r.borrow());
 a_fn_that_mutably_borrows(&mut r.borrow_mut());
 a_fn_that_immutably_borrows(&r.borrow());
}

fn main() {
 let data = RefCell::new(5);
 demo(&data);
}

```

Listing 15-14: Using `RefCell<T>`, `borrow`, and `borrow_mut`

这个例子打印出：

```

a is 5
a is 6

```

在 `main` 函数中，我们新声明了一个包含值 `5` 的 `RefCell<T>`，并储存在变量 `data` 中，声明时并没有使用 `mut` 关键字。接着使用 `data` 的一个不可变引用来调用 `demo` 函数：对于 `main` 函数而言 `data` 是不可变的！

在 `demo` 函数中，通过调用 `borrow` 方法来获取到 `RefCell<T>` 中值的不可变引用，并使用这个不可变引用调用了 `a_fn_that_immutably_borrows` 函数。更为有趣的是，可以通过 `borrow_mut` 方法来获取 `RefCell<T>` 中值的可变引用，而 `a_fn_that_mutably_borrows` 函数就允许修改这个值。可以看到下一次调用 `a_fn_that_immutably_borrows` 时打印出的值是 `6` 而不是 `5`。

## RefCell<T> 在运行时检查借用规则

回忆一下第四章因为借用规则，尝试使用常规引用在同一作用域中创建两个可变引用的代码无法编译：

```

let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s;

```

这会得到一个编译错误：

```

error[E0499]: cannot borrow `s` as mutable more than once at a time
-->
|
5 | let r1 = &mut s;
| - first mutable borrow occurs here
6 | let r2 = &mut s;
| ^ second mutable borrow occurs here
7 | }
| - first borrow ends here

```

与此相反，使用 `RefCell<T>` 并在同一作用域调用两次 `borrow_mut` 的代码是可以编译的，不过它会在运行时 `panic`。如下代码：

```

use std::cell::RefCell;

fn main() {
 let s = RefCell::new(String::from("hello"));
}

```

```

 let r1 = s.borrow_mut();
 let r2 = s.borrow_mut();
}

```

能够编译不过在 `cargo run` 运行时会出现如下错误：

```

Finished dev [unoptimized + debuginfo] target(s) in 0.83 secs
Running `target/debug/refcell`
thread 'main' panicked at 'already borrowed: BorrowMutError',
/stable-dist-rustc/build/src/libcore/result.rs:868
note: Run with `RUST_BACKTRACE=1` for a backtrace.

```

这个运行时 `BorrowMutError` 类似于编译错误：它表明我们已经可变得借用过一次 `s` 了，所以不允许再次借用它。我们并没有绕过借用规则，只是选择让 **Rust** 在运行时而不是编译时执行他们。你可以选择在任何时候任何地方使用 `RefCell<T>`，不过除了不得不编写很多 `RefCell` 之外，最终还是可能会发现其中的问题（可能是在生产环境而不是开发环境）。另外，在运行时检查借用规则有性能惩罚。

## 结合 `Rc<T>` 和 `RefCell<T>` 来拥有多个可变数据所有者

那么为什么要权衡考虑选择引入 `RefCell<T>` 呢？好吧，还记得我们说过 `Rc<T>` 只能拥有一个 `T` 的不可变引用吗？考虑到 `RefCell<T>` 是不可变的，但是拥有内部可变性，可以将 `Rc<T>` 与 `RefCell<T>` 结合起来创建一个既有引用计数又可变的类型。示例 15-15 展示了一个这么做的例子，再次回到示例 15-5 中的 `cons list`。在这个例子中，不同于在 `cons list` 中储存 `i32` 值，我们储存一个 `Rc<RefCell<i32>>` 值。希望储存这个类型是因为其可以拥有不属于列表一部分的这个值的所有者（`Rc<T>` 提供的多个所有者功能），而且还可以改变内部的 `i32` 值（`RefCell<T>` 提供的内部可变性功能）：

Filename: src/main.rs

```

#[derive(Debug)]
enum List {
 Cons(Rc<RefCell<i32>>, Rc<List>),
 Nil,
}

use List::{Cons, Nil};
use std::rc::Rc;
use std::cell::RefCell;

fn main() {
 let value = Rc::new(RefCell::new(5));

 let a = Cons(value.clone(), Rc::new(Nil));
 let shared_list = Rc::new(a);

 let b = Cons(Rc::new(RefCell::new(6)), shared_list.clone());
 let c = Cons(Rc::new(RefCell::new(10)), shared_list.clone());

 *value.borrow_mut() += 10;

 println!("shared_list after = {:?}", shared_list);
 println!("b after = {:?}", b);
 println!("c after = {:?}", c);
}

```

Listing 15-15: Using `Rc<RefCell<i32>>` to create a `List` that we can mutate

我们创建了一个值，它是 `Rc<RefCell<i32>>` 的实例。将其储存在变量 `value` 中因为我们希望之后能直接访问它。接着在 `a` 中创建了一个拥有存放了 `value` 值的 `Cons` 成员的 `List`，而且 `value` 需要被克隆因为我们希望除了 `a` 之外还拥有 `value` 的所有权。接着将 `a` 封装进 `Rc<T>` 中这样就可以创建都引用 `a` 的有着不同开头的列表 `b` 和 `c`，类似示例 15-12 中所做的那样。

一旦创建了 `shared_list`、`b` 和 `c`，接下来就可以通过解引用 `Rc<T>` 和对 `RefCell` 调用 `borrow_mut` 来将 10 与 5 相加了。

当打印出 `shared_list`、`b` 和 `c` 时，可以看到他们都拥有被修改的值 15：

```
shared_list after = Cons(RefCell { value: 15 }, Nil)
b after = Cons(RefCell { value: 6 }, Cons(RefCell { value: 15 }, Nil))
c after = Cons(RefCell { value: 10 }, Cons(RefCell { value: 15 }, Nil))
```

这是非常巧妙的！通过使用 `RefCell<T>`，我们可以拥有一个表面上不可变的 `List`，不过可以使用 `RefCell<T>` 中提供内部可变性的方法来在需要时修改数据。`RefCell<T>` 的运行时借用规则检查也确实保护我们免于出现数据竞争，而且我们也决定牺牲一些速度来换取数据结构的灵活性。

`RefCell<T>` 并不是标准库中唯一提供内部可变性的类型。`Cell<T>` 有点类似，不过不同于 `RefCell<T>` 那样提供内部值的引用，其值被拷贝进和拷贝出 `Cell<T>`。`Mutex<T>` 提供线程间安全的内部可变性，下一章并发会讨论它的应用。请查看标准库来获取更多细节和不同类型的区别。

## 引用循环和内存泄漏是安全的

[ch15-06-reference-cycles.md](#)

commit 9430a3d28a2121a938d704ce48b15d21062f880e

我们讨论过 **Rust** 做出的一些保证，例如永远也不会遇到一个空值，而且数据竞争也会在编译时被阻止。**Rust** 的内存安全保证也使其更难以制造从不被清理的内存，这被称为内存泄露。然而 **Rust** 并不是不可能出现内存泄漏，避免内存泄露并不是 **Rust** 的保证之一。换句话说，内存泄露是安全的。

在使用 `Rc<T>` 和 `RefCell<T>` 时，有可能创建循环引用，这时各个项相互引用并形成环。这是不好的因为每一项的引用计数将永远也到不了 0，其值也永远也不会被丢弃。让我们看看这是如何发生的以及如何避免它。

在示例 15-16 中，我们将使用示例 15-5 中 `List` 定义的另一个变体。我们将回到储存 `i32` 值作为 `Cons` 成员的第一个元素。现在 `Cons` 成员的第二个元素是 `RefCell<Rc<List>>`：这时就不能修改 `i32` 值了，但是能够修改 `Cons` 成员指向的那个 `List`。还需要增加一个 `tail` 方法来方便我们在拥有一个 `Cons` 成员时访问第二个项：

Filename: src/main.rs

```
#[derive(Debug)]
enum List {
 Cons(i32, RefCell<Rc<List>>),
 Nil,
}

impl List {
 fn tail(&self) -> Option<&RefCell<Rc<List>>> {
 match *self {
 Cons(_, ref item) => Some(item),
 Nil => None,
 }
 }
}
```

Listing 15-16: A cons list definition that holds a `RefCell` so that we can modify what a `Cons` variant is referring to

接下来，在示例 15-17 中，我们将在变量 `a` 中创建一个 `List` 值，其内部是一个 `5`, `Nil` 的列表。接着在变量 `b` 创建一个值 `10` 和指向 `a` 中列表的 `List` 值。最后修改 `a` 指向 `b` 而不是 `Nil`，这会创建一个循环：

Filename: src/main.rs

```
###[derive(Debug)]
enum List {
Cons(i32, RefCell<Rc<List>>),
Nil,
}
#
impl List {
fn tail(&self) -> Option<&RefCell<Rc<List>>> {
match *self {
Cons(_, ref item) => Some(item),
Nil => None,
}
}
}
#
use List::{Cons, Nil};
use std::rc::Rc;
use std::cell::RefCell;
```



```
fn main() {

 let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));

 println!("a initial rc count = {}", Rc::strong_count(&a));
 println!("a next item = {:?}", a.tail());

 let b = Rc::new(Cons(10, RefCell::new(a.clone())));

 println!("a rc count after b creation = {}", Rc::strong_count(&a));
 println!("b initial rc count = {}", Rc::strong_count(&b));
 println!("b next item = {:?}", b.tail());

 if let Some(ref link) = a.tail() {
 *link.borrow_mut() = b.clone();
 }

 println!("b rc count after changing a = {}", Rc::strong_count(&b));
 println!("a rc count after changing a = {}", Rc::strong_count(&a));

 // Uncomment the next line to see that we have a cycle; it will
 // overflow the stack
 // println!("a next item = {:?}", a.tail());
}
```

Listing 15-17: Creating a reference cycle of two `List` values pointing to each other

使用 `tail` 方法来获取 `a` 中 `RefCell` 的引用，并将其放入变量 `link` 中。接着对 `RefCell` 使用 `borrow_mut` 方法将其中的值从存放 `Nil` 值的 `Rc` 改为 `b` 中的 `Rc`。这创建了一个看起来像图 15-18 所示的引用循环：

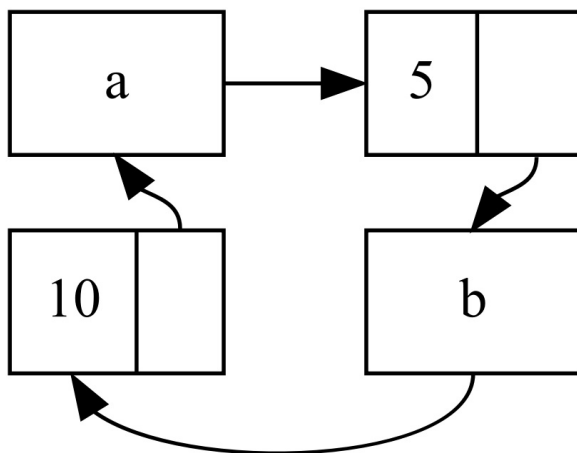


Figure 15-18: A reference cycle of lists `a` and `b` pointing to each other

如果你注释掉最后的 `println!`，`Rust` 会尝试打印出 `a` 指向 `b` 指向 `a` 这样的循环直到栈溢出。

观察最后一个 `println!` 之前的打印结果，就会发现在将 `a` 改变为指向 `b` 之后 `a` 和 `b` 的引用计数都是 2。在 `main` 的结尾，`Rust` 首先会尝试丢弃 `b`，这会使 `Rc` 的引用计数减一，但是这个计数是 1 而不是 0，所以 `Rc` 在堆上的内存不会被丢弃。它只是会永远的停留在 1 上。这个特定例子中，程序立马就结束了，所以并不是一个问题，不过如果是一个更加复杂的程序，它在这个循环中分配了很多内存并占有很长时间，这就是个问题了。这个程序会使用多于它所需要的内存，并有可能压垮系统并造成没有内存可供使用。

现在，如你所见，在 `Rust` 中创建引用循环是困难和繁琐的。但并不是不可能：避免引用循环这种形式的内存泄漏并不是 `Rust` 的保证之一。如果你有包含 `Rc<T>` 的 `RefCell<T>` 值或类似的嵌套结合了内部可变性和引用计数的类型，请务必小心确保你没有形成一个引用循环。在示例 15-14 的例子中，可能解决方式就是不要编写像这样可能造

成引用循环的代码，因为我们希望 `Cons` 成员拥有他们指向的列表。

举例来说，对于像图这样的数据结构，为了创建父节点指向子节点的边和以相反方向从子节点指向父节点的边，有时需要创建这样的引用循环。如果一个方向拥有所有权而另一个方向没有，对于模拟这种数据关系的一种不会创建引用循环和内存泄露的方式是使用 `Weak<T>`。接下来让我们探索一下！

## 避免引用循环：将 `Rc<T>` 变为 `Weak<T>`

`Rust` 标准库中提供了 `Weak<T>`，一个用于存在引用循环但只有一个方向有所有权的智能指针。我们已经展示过如何克隆 `Rc<T>` 来增加引用的 `strong_count`；`Weak<T>` 是一种引用 `Rc<T>` 但不增加 `strong_count` 的方式：相反它增加 `Rc` 引用的 `weak_count`。当 `Rc` 离开作用域，其内部值会在 `strong_count` 为 0 的时候被丢弃，即便 `weak_count` 不为 0。为了能够从 `Weak<T>` 中获取值，首先需要使用 `upgrade` 方法将其升级为 `Option<Rc<T>>`。升级 `Weak<T>` 的结果在 `Rc` 还未被丢弃时是 `Some`，而在 `Rc` 被丢弃时是 `None`。因为 `upgrade` 返回一个 `Option`，我们知道 `Rust` 会确保 `Some` 和 `None` 的情况都被处理并不会尝试使用一个无效的指针。

不同于示例 15-17 中每个项只知道它的下一项，假如我们需要一个树，它的项知道它的子项和父项。

让我们从一个叫做 `Node` 的存放拥有所有权的 `i32` 值和其子 `Node` 值的引用的结构体开始：

```
use std::rc::Rc;
use std::cell::RefCell;

#[derive(Debug)]
struct Node {
 value: i32,
 children: RefCell<Vec<Rc<Node>>>,
}
```

我们希望能够 `Node` 拥有其子节点，同时也希望变量可以拥有每个节点以便可以直接访问他们。这就是为什么 `Vec` 中的项是 `Rc<Node>` 值。我们也希望能够修改其他节点的子节点，这就是为什么 `children` 中 `Vec` 被放进了 `RefCell` 的原因。在示例 15-19 中创建了一个叫做 `leaf` 的带有值 3 并没有子节点的 `Node` 实例，和另一个带有值 5 和以 `leaf` 作为子节点的实例 `branch`：

Filename: src/main.rs

```
fn main() {
 let leaf = Rc::new(Node {
 value: 3,
 children: RefCell::new(vec![]),
 });

 let branch = Rc::new(Node {
 value: 5,
 children: RefCell::new(vec![leaf.clone()]),
 });
}
```

Listing 15-19: Creating a `leaf` node and a `branch` node where `branch` has `leaf` as one of its children but `leaf` has no reference to `branch`

`leaf` 中的 `Node` 现在有两个所有者：`leaf` 和 `branch`，因为我们克隆了 `leaf` 中的 `Rc` 并储存在了 `branch` 中。`branch` 中的 `Node` 知道它与 `leaf` 相关联因为 `branch` 在 `branch.children` 中有 `leaf` 的引用。然而，`leaf` 并不知道它与 `branch` 相关联，而我们希望 `leaf` 知道 `branch` 是其父节点。

为了做到这一点，需要在 `Node` 结构体定义中增加一个 `parent` 字段，不过 `parent` 的类型应该是什么呢？我们知道它不能包含 `Rc<T>`，因为这样 `leaf.parent` 将会指向 `branch` 而 `branch.children` 会包含 `leaf` 的指针，这会形成引用循环。`leaf` 和 `branch` 不会被丢弃因为他们总是引用对方且引用计数永远也不会是零。

所以在 `parent` 的类型中使用 `Weak<T>` 而不是 `Rc`，具体来说是 `RefCell<Weak<Node>>`：

Filename: src/main.rs

```
use std::rc::{Rc, Weak};
use std::cell::RefCell;

#[derive(Debug)]
struct Node {
 value: i32,
 parent: RefCell<Weak<Node>>,
 children: RefCell<Vec<Rc<Node>>>,
}
```

这样，一个节点就能够在拥有父节点时指向它，而并不拥有其父节点。一个父节点哪怕在拥有指向它的子节点也会被丢弃，只要是其自身也没有一个父节点就行。现在将 `main` 函数更新为如示例 15-20 所示：

Filename: src/main.rs

```
fn main() {
 let leaf = Rc::new(Node {
 value: 3,
 parent: RefCell::new(Weak::new()),
 children: RefCell::new(vec![]),
 });

 println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());

 let branch = Rc::new(Node {
 value: 5,
 parent: RefCell::new(Weak::new()),
 children: RefCell::new(vec![leaf.clone()]),
 });

 *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

 println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
}
```

Listing 15-20: A `leaf` node and a `branch` node where `leaf` has a `Weak` reference to its parent, `branch`

创建 `leaf` 节点是类似的；因为它作为开始并没有父节点，这里创建了一个新的 `Weak` 引用实例。当尝试通过 `upgrade` 方法获取 `leaf` 父节点的引用时，会得到一个 `None` 值，如第一个 `println!` 输出所示：

```
leaf parent = None
```

类似的，`branch` 也有一个新的 `Weak` 引用，因为也没有父节点。`leaf` 仍然作为 `branch` 的一个子节点。一旦在 `branch` 中有了一个新的 `Node` 实例，就可以修改 `leaf` 将一个 `branch` 的 `Weak` 引用作为其父节点。这里使用了 `leaf` 中 `parent` 字段里的 `RefCell` 的 `borrow_mut` 方法，接着使用了 `Rc::downgrade` 函数来从 `branch` 中的 `Rc` 值创建了一个指向 `branch` 的 `Weak` 引用。

当再次打印出 `leaf` 的父节点时，这一次将会得到存放了 `branch` 的 `Some` 值。另外需要注意到这里并没有打印出类似示例 15-14 中那样最终导致栈溢出的循环：`Weak` 引用仅仅打印出 `(Weak)`：

```
leaf parent = Some(Node { value: 5, parent: RefCell { value: (Weak) },
children: RefCell { value: [Node { value: 3, parent: RefCell { value: (Weak) },
children: RefCell { value: [] } }] } })
```

没有无限的输出（或直到栈溢出）的事实表明这里并没有引用循环。另一种证明的方式是观察调用 `Rc::strong_count` 和 `Rc::weak_count` 的值。在示例 15-21 中，创建了一个新的内部作用域并将 `branch` 的创建放入其中，这样可以观察 `branch` 被创建时和离开作用域被丢弃时发生了什么：

Filename: src/main.rs

```
fn main() {
 let leaf = Rc::new(Node {
 value: 3,
 parent: RefCell::new(Weak::new()),
 children: RefCell::new(vec![]),
 });

 println!(
 "leaf strong = {}, weak = {}",
 Rc::strong_count(&leaf),
 Rc::weak_count(&leaf),
);

 {
 let branch = Rc::new(Node {
 value: 5,
 parent: RefCell::new(Weak::new()),
 children: RefCell::new(vec![leaf.clone()]),
 });
 *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

 println!(
 "branch strong = {}, weak = {}",
 Rc::strong_count(&branch),
 Rc::weak_count(&branch),
);

 println!(
 "leaf strong = {}, weak = {}",
 Rc::strong_count(&leaf),
 Rc::weak_count(&leaf),
);
 }

 println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
 println!(
 "leaf strong = {}, weak = {}",
 Rc::strong_count(&leaf),
 Rc::weak_count(&leaf),
);
}
```

Listing 15-21: Creating `branch` in an inner scope and examining strong and weak reference counts of `leaf` and `branch`

创建 `leaf` 之后，强引用计数是 1（用于 `leaf` 自身）而弱引用计数是 0。在内部作用域中，在创建 `branch` 和关联 `leaf` 和 `branch` 之后，`branch` 的强引用计数为 1（用于 `branch` 自身）而弱引用计数为 1（因为 `leaf.parent` 通过一个 `Weak<T>` 指向 `branch`）。`leaf` 的强引用计数为 2，因为 `branch` 现在有一个 `leaf` 克隆的 `Rc` 储存在 `branch.children` 中。`leaf` 的弱引用计数仍然为 0。

当内部作用域结束，`branch` 离开作用域，其强引用计数减少为 0，所以其 `Node` 被丢弃。来自 `leaf.parent` 的弱引用计数 1 与 `Node` 是否被丢弃无关，所以并没有产生内存泄露！

如果在内部作用域结束后尝试访问 `leaf` 的父节点，会像 `leaf` 拥有父节点之前一样得到 `None` 值。在程序的末尾，`leaf` 的强引用计数为 1 而弱引用计数为 0，因为现在 `leaf` 又是唯一指向其自己的值了。

所有这些管理计数和值是否应该被丢弃的逻辑都通过 `Rc` 和 `Weak` 和他们的 `Drop trait` 实现来控制。通过在定义中指定从子节点到父节点的关系为一个 `Weak<T>` 引用，就能够拥有父节点和子节点之间的双向引用而不会造成引用循环和内存泄露。

## 总结

现在我们学习了如何选择不同类型的智能指针来选择不同的保证并与 **Rust** 的常规引用向取舍。`Box<T>` 有一个已知的大小并指向分配在堆上的数据。`Rc<T>` 记录了堆上数据的引用数量这样就可以拥有多个所有者。`RefCell<T>` 和其内部可变性使其可以用于需要不可变类型，但希望在运行时而不是编译时检查借用规则的场景。

我们还介绍了提供了很多智能指针功能的 `trait Deref` 和 `Drop`。同时探索了形成引用循环和造成内存泄漏的可能性，以及如何使用 `Weak<T>` 避免引用循环。

如果本章内容引起了你的兴趣并希望现在就实现你自己的智能指针的话，请阅读 [The Nomicon](#) 来获取更多有用的信息。

接下来，让我们谈谈 **Rust** 的并发。我们还会学习到一些新的对并发有帮助的智能指针。

# 无畏并发

[ch16-00-concurrency.md](#)

commit da15de39eaabd50100d6fa662c653169254d9175

确保内存安全并不是 **Rust** 的唯一目标：更好的处理并发和并行编程一直是 **Rust** 的另一个主要目标。并发编程（**concurrent programming**）代表程序的不同部分相互独立的执行，而并行编程代表程序不同部分同时执行，这两个概念在计算机拥有更多处理器可供程序利用时变得更加重要。由于历史的原因，在此类上下文中编程一直是困难且容易出错的：**Rust** 希望能改变这一点。

最开始，我们认为内存安全和防止并发问题是需要通过两个不同的方法解决的两个相互独立的挑战。然而，随着时间的推移，我们发现所有权和类型系统是一系列解决内存安全和并发问题的强用力的工具！通过改进所有权和类型检查，很多并发错误在 **Rust** 中都是编译时错误，而不是运行时错误。我们给 **Rust** 的这一部分起了一个绰号无畏并发（**fearless concurrency**）。无畏并发意味着 **Rust** 不光允许你自信代码不会出现诡异的错误，也让你可以轻易重构这种代码而无需担心会引入新的 **bug**。

注意：对于 **Rust** 的口号无畏并发，这里用并发指代很多问题而不是更精确的区分并发和（或）并行，是出于简化问题的原因。如果这是一本专注于并发和/或并行的书，我们肯定会更精确的。对于本章，当我们谈到并发时，请自行替换为并发和（或）并行。

很多语言所提供的处理并发问题的解决方法都非常有特色，尤其是对于更高级的语言，这是一个非常合理的策略。然而对于底层语言则没有奢侈的选择。在任何给定的情况下，我们都期望底层语言可以提供最高的性能，并且对硬件有更薄的抽象。因此，**Rust** 给了我们多种工具，并以适合实际情况和需求的方式来为问题建模。

如下是本章将要涉及到的内容：

- 如何创建线程来同时运行多段代码。
- 并发消息传递（**Message passing**），其中通道（**channel**）被用来在线程间传递消息。
- 并发共享状态（**Shared state**），其中多个线程可以访问同一片数据。
- **Sync** 和 **Send trait**，他们允许 **Rust** 的并发保证能被扩展到用户定义的和标准库中提供的类型中。

# 使用线程同时运行代码

ch16-01-threads.md

commit 55b294f20fc846a13a9be623bf322d8b364cee77

在今天使用的大部分操作系统中，当程序执行时，操作系统运行代码的上下文称为进程（*process*）。操作系统可以运行很多进程，而操作系统也管理这些进程使得多个程序可以在电脑上同时运行。

我们可以将每个进程运行一个程序的概念再往下抽象一层：程序也可以在其上下文中同时运行独立的部分。这个功能叫做线程（*thread*）。

将程序需要执行的计算拆分到多个线程中可以提高性能，因为程序可以在同时进行很多工作。不过使用线程会增加程序复杂性。因为线程是同时运行的，所以无法预先保证不同线程中的代码的执行顺序。这可能会由于线程以不一致的顺序访问数据或资源而导致竞争状态，或由于两个线程相互阻止对方继续运行而造成死锁，以及仅仅出现于特定场景并难以稳定重现的 **bug**。**Rust** 减少了这些或那些使用线程的负面影响，不过在多线程上下文中编程，相比只期望在单个线程中运行的程序，仍然要采用不同的思考方式和代码结构。

编程语言有一些不同的方法来实现线程。很多操作系统提供了创建新线程的 **API**。另外，很多编程语言提供了自己的特殊的线程实现。编程语言提供的线程有时被称作轻量级（*lightweight*）或绿色（*green*）线程。这些语言将一系列绿色线程放入不同数量的操作系统线程中执行。因为这个原因，语言调用操作系统 **API** 创建线程的模型有时被称为 **1:1**，一个 **OS** 线程对应一个语言线程。绿色线程模型被称为 **M:N** 模型，**M** 个绿色线程对应 **N** 个 **OS** 线程，这里 **M** 和 **N** 不必相同。

每一个模型都有其自己的优势和取舍。对于 **Rust** 来说最重要的取舍是运行时支持。运行时是一个令人迷惑的概念；在不同上下文中它可能有不同的含义。这里其代表二进制文件中包含的语言自身的代码。对于一些语言，这些代码是庞大的，另一些则很小。通俗的说，“没有运行时”通常被人们用来指代“小运行时”，因为任何非汇编语言都存在一定数量的运行时。更小的运行时拥有更少的功能不过其优势在于更小的二进制输出。更小的二进制文件更容易在更多上下文中与其他语言结合。虽然很多语言觉得增加运行时来换取更多功能没有什么问题，但是 **Rust** 需要做到几乎没有运行时，同时为了保持高性能必需能够调用 **C** 语言，这点也是不能妥协的。

绿色线程模型功能要求更大的运行时来管理这些线程。为此，**Rust** 标准库只提供了 **1:1** 线程模型实现。因为 **Rust** 是这么一个底层语言，所以有相应的 **crate** 实现了 **M:N** 线程模型，如果你宁愿牺牲性能来换取例如更好的线程运行控制和更低的上下文切换成本。

现在我们明白了 **Rust** 中的线程是如何定义的，让我们开始探索如何使用标准库提供的线程相关的 **API**吧。

## 使用 **spawn** 创建新线程

为了创建一个新线程，调用 `thread::spawn` 函数并传递一个闭包（第十三章学习了闭包），它包含希望在新线程运行的代码。列表 16-1 中的例子在新线程中打印了一些文本而其余的文本在主线程中打印：

Filename: src/main.rs

```
use std::thread;

fn main() {
 thread::spawn(|| {
 for i in 1..10 {
 println!("hi number {} from the spawned thread!", i);
 }
 });

 for i in 1..5 {
 println!("hi number {} from the main thread!", i);
 }
}
```

```
}
}
```

Listing 16-1: Creating a new thread to print one thing while the main thread is printing something else

注意这个函数编写的方式，当主线程结束时，它也会停止新线程。这个程序的输出每次可能都略微不同，不过它大体上看起来像这样：

```
hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the main thread!
hi number 2 from the spawned thread!
hi number 3 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the main thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
```

这些线程可能会轮流运行，不过并不保证如此。在这里，主线程先行打印，即便新创建线程的打印语句位于程序的开头。甚至即便我们告诉新建的线程打印直到 `i` 等于 9，它的主线程结束之前也只打印到了 5。如果你只看到了一个线程，或没有出现重叠打印的现象，尝试增加 `range` 的数值来增加线程暂停并切换到其他线程运行的机会。

## 使用 `join` 等待所有线程结束

由于主线程先于新建线程结束，不仅列表 16-1 中的代码大部分时候不能保证新建线程执行完毕，甚至不能实际保证新建线程会被执行！可以通过保存 `thread::spawn` 的返回值来解决这个问题，这是一个 `JoinHandle`。这看起来如列表 16-2 所示：

Filename: src/main.rs

```
use std::thread;

fn main() {
 let handle = thread::spawn(|| {
 for i in 1..10 {
 println!("hi number {} from the spawned thread!", i);
 }
 });

 for i in 1..5 {
 println!("hi number {} from the main thread!", i);
 }

 handle.join();
}
```

Listing 16-2: Saving a `JoinHandle` from `thread::spawn` to guarantee the thread is run to completion

`JoinHandle` 是一个拥有所有权的值，它可以等待一个线程结束，这也正是 `join` 方法所做的。通过调用这个句柄的 `join`，当前线程会阻塞直到句柄所代表的线程结束。因为我们将 `join` 调用放在了主线程的 `for` 循环之后，运行这个例子将产生类似这样的输出：

```
hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 1 from the spawned thread!
hi number 3 from the main thread!
hi number 2 from the spawned thread!
hi number 4 from the main thread!
```



```
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
```

这两个线程仍然会交替执行，不过主线程会由于 `handle.join()` 调用会等待直到新建线程执行完毕。

如果将 `handle.join()` 放在主线程的 `for` 循环之前，像这样：

Filename: src/main.rs

```
use std::thread;

fn main() {
 let handle = thread::spawn(|| {
 for i in 1..10 {
 println!("hi number {} from the spawned thread!", i);
 }
 });

 handle.join();

 for i in 1..5 {
 println!("hi number {} from the main thread!", i);
 }
}
```

主线程会等待直到新建线程执行完毕之后才开始执行 `for` 循环，所以输出将不会交替出现：

```
hi number 1 from the spawned thread!
hi number 2 from the spawned thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 3 from the main thread!
hi number 4 from the main thread!
```

稍微考虑一下将 `join` 放置于何处会影响线程是否同时运行。

## 线程和 `move` 闭包

第十三章有一个我们没有讲到的闭包功能，它经常用于 `thread::spawn` : `move` 闭包。第十三章中讲到：

获取他们环境中值的闭包主要用于开始新线程的场景

现在我们正在创建新线程，所以让我们讨论一下获取环境值的闭包吧！

注意列表 16-1 中传递给 `thread::spawn` 的闭包并没有任何参数：并没有在新建线程代码中使用任何主线程的数据。为了在新建线程中使用来自于主线程的数据，需要新建线程的闭包获取它需要的值。列表 16-3 展示了一个尝试在主线程中创建一个 `vector` 并用于新建线程的例子，不过这么写还不能工作：

Filename: src/main.rs

```
use std::thread;

fn main() {
 let v = vec![1, 2, 3];

 let handle = thread::spawn(|| {
 println!("Here's a vector: {:?}", v);
 });

 handle.join();
}
```

### Listing 16-3: Attempting to use a vector created by the main thread from another thread

闭包使用了 `v`，所以闭包会获取 `v` 并使其成为闭包环境的一部分。因为 `thread::spawn` 在一个新线程中运行这个闭包，所以可以在新线程中访问 `v`。

然而当编译这个例子时，会得到如下错误：

```
error[E0373]: closure may outlive the current function, but it borrows `v`,
which is owned by the current function
-->
|
6 | let handle = thread::spawn(|| {
| ^^ may outlive borrowed value `v`
7 | println!("Here's a vector: {:?}", v);
| - `v` is borrowed here
|
help: to force the closure to take ownership of `v` (and any other referenced
variables), use the `move` keyword, as shown:
| let handle = thread::spawn(move || {
```

当在闭包环境中获取某些值时，**Rust** 会尝试推断如何获取它。`println!` 只需要 `v` 的一个引用，所以闭包尝试借用 `v`。但是这有一个问题：我们并不知道新建线程会运行多久，所以无法知道 `v` 是否一直时有效的。

考虑一下列表 16-4 中的代码，它展示了一个 `v` 的引用很有可能不再有效的场景：

Filename: src/main.rs

```
use std::thread;

fn main() {
 let v = vec![1, 2, 3];

 let handle = thread::spawn(|| {
 println!("Here's a vector: {:?}", v);
 });

 drop(v); // oh no!

 handle.join();
}
```

### Listing 16-4: A thread with a closure that attempts to capture a reference to `v` from a main thread that drops `v`

这些代码可以运行，而新建线程则可能直接就出错了并完全没有机会运行。新建线程内部有一个 `v` 的引用，不过主线程仍在执行：它立刻丢弃了 `v`，使用了第十五章提到的显式丢弃其参数的 `drop` 函数。接着，新建线程开始执行，现在 `v` 是无效的了，所以它的引用也就是无效的。噢，这太糟了！

为了修复这个问题，我们可以听取错误信息的建议：

```
help: to force the closure to take ownership of `v` (and any other referenced
variables), use the `move` keyword, as shown:
| let handle = thread::spawn(move || {
```

通过在闭包之前增加 `move` 关键字，我们强制闭包获取它使用的值的所有权，而不是引用借用。列表 16-5 中展示的对列表 16-3 代码的修改可以按照我们的预期编译并运行：

Filename: src/main.rs

```
use std::thread;

fn main() {
 let v = vec![1, 2, 3];

 let handle = thread::spawn(move || {
 println!("Here's a vector: {:?}", v);
 });

 handle.join();
}
```

Listing 16-5: Using the `move` keyword to force a closure to take ownership of the values it uses

那么列表 16-4 中那个主线程调用了 `drop` 的代码该怎么办呢？如果在闭包上增加了 `move`，就将 `v` 移动到了闭包的环境中，我们将不能对其调用 `drop` 了。相反会出现这个编译时错误：

```
error[E0382]: use of moved value: `v`
-->
|
6 | let handle = thread::spawn(move || {
| ----- value moved (into closure) here
...
10 | drop(v); // oh no!
| ^ value used here after move
|
= note: move occurs because `v` has type `std::vec::Vec<i32>`, which does
not implement the `Copy` trait
```

Rust 的所有权规则又一次帮助我们！

现在我们有一个线程和线程 API 的基本了解，让我们讨论一下使用线程实际可以做什么吧。

# 使用消息传递在线程间传送数据

ch16-02-message-passing.md

commit da15de39eaabd50100d6fa662c653169254d9175

最近人气正在上升的一个并发方式是消息传递（*message passing*），这里线程或 **actor** 通过发送包含数据的消息来沟通。这个思想来源于口号：

Do not communicate by sharing memory; instead, share memory by communicating.

不要共享内存来通讯；而是要通讯来共享内存。

--Effective Go

实现这个目标的主要工具是通道（*channel*）。通道有两部分组成，一个发送者（**transmitter**）和一个接收者（**receiver**）。代码的一部分可以调用发送者和想要发送的数据，而另一部分代码可以在接收的那一端收取消息。

我们将编写一个例子使用一个线程生成值并向通道发送他们。主线程会接收这些值并打印出来。

首先，如列表 16-6 所示，先创建一个通道但不做任何事：

Filename: src/main.rs

```
use std::sync::mpsc;

fn main() {
 let (tx, rx) = mpsc::channel();
 # tx.send(()).unwrap();
}
```

Listing 16-6: Creating a channel and assigning the two halves to `tx` and `rx`

`mpsc::channel` 函数创建一个新的通道。`mpsc` 是多个生产者，单个消费者（*multiple producer, single consumer*）的缩写。简而言之，可以有多个产生值的发送端，但只能有一个消费这些值的接收端。现在我们以一个单独的生产者开始，不过一旦例子可以工作了就会增加多个生产者。

`mpsc::channel` 返回一个元组：第一个元素是发送端，而第二个元素是接收端。由于历史原因，很多人使用 `tx` 和 `rx` 作为发送者和接收者的缩写，所以这就是我们将用来绑定这两端变量的名字。这里使用了一个 `let` 语句和模式来解构了元组。第十八章会讨论 `let` 语句中的模式和解构。

让我们将发送端移动到一个新建线程中并发送一个字符串，如列表 16-7 所示：

Filename: src/main.rs

```
use std::thread;
use std::sync::mpsc;

fn main() {
 let (tx, rx) = mpsc::channel();

 thread::spawn(move || {
 let val = String::from("hi");
 tx.send(val).unwrap();
 });
}
```

Listing 16-7: Moving `tx` to a spawned thread and sending "hi"

正如上一部分那样使用 `thread::spawn` 来创建一个新线程。并使用一个 `move` 闭包来将 `tx` 移动进闭包这样新建线程就是其所有者。

通道的发送端有一个 `send` 方法用来获取需要放入通道的值。`send` 方法返回一个 `Result<T, E>` 类型，因为如果接收端被丢弃了，将没有发送值的目标，所以发送操作会出错。在这个例子中，我们简单的调用 `unwrap` 来忽略错误，不过对于一个真实程序，需要合理的处理它。第九章是你复习正确错误处理策略的好地方。

在列表 16-8 中，让我们在主线程中从通道的接收端获取值：

Filename: src/main.rs

```
use std::thread;
use std::sync::mpsc;

fn main() {
 let (tx, rx) = mpsc::channel();

 thread::spawn(move || {
 let val = String::from("hi");
 tx.send(val).unwrap();
 });

 let received = rx.recv().unwrap();
 println!("Got: {}", received);
}
```

Listing 16-8: Receiving the value "hi" in the main thread and printing it out

通道的接收端有两个有用的方法：`recv` 和 `try_recv`。这里，我们使用了 `recv`，它是 *receive* 的缩写。这个方法会阻塞执行直到从通道中接收一个值。一旦发送了一个值，`recv` 会在一个 `Result<T, E>` 中返回它。当通道发送端关闭，`recv` 会返回一个错误。`try_recv` 不会阻塞；相反它立刻返回一个 `Result<T, E>`。

如果运行列表 16-8 中的代码，我们将会看到主线程打印出这个值：

```
Got: hi
```

## 通道与所有权如何交互

现在让我们做一个试验来看看通道与所有权如何在一起工作：我们将尝试在新建线程中的通道中发送完 `val` 之后再使用它。尝试编译列表 16-9 中的代码：

Filename: src/main.rs

```
use std::thread;
use std::sync::mpsc;

fn main() {
 let (tx, rx) = mpsc::channel();

 thread::spawn(move || {
 let val = String::from("hi");
 tx.send(val).unwrap();
 println!("val is {}", val);
 });

 let received = rx.recv().unwrap();
 println!("Got: {}", received);
}
```

### Listing 16-9: Attempting to use `val` after we have sent it down the channel

这里尝试在通过 `tx.send` 发送 `val` 到通道中之后将其打印出来。这是一个坏主意：一旦将值发送到另一个线程后，那个线程可能会在我们在此使用它之前就修改或者丢弃它。这会由于不一致或不存在的数据而导致错误或意外的结果。

尝试编译这些代码，`Rust` 会报错：

```
error[E0382]: use of moved value: `val`
 --> src/main.rs:10:31
 |
 9 | tx.send(val).unwrap();
 | --- value moved here
10 | println!("val is {}", val);
 | ^^^ value used here after move
 |
 = note: move occurs because `val` has type `std::string::String`, which does
 not implement the `Copy` trait
```

我们的并发错误会造成一个编译时错误！`send` 获取其参数的所有权并移动这个值归接收者所有。这个意味着不可能意外的在发送后再次使用这个值；所有权系统检查一切是否合乎规则。

在这一点上，消息传递非常类似于 `Rust` 的单所有权系统。消息传递的拥护者出于相似的原因支持消息传递，就像 `Rustacean` 们欣赏 `Rust` 的所有权一样：单所有权意味着特定类型问题的消失。如果一次只有一个线程可以使用某些内存，就没有出现数据竞争的机会。

## 发送多个值并观察接收者的等待

列表 16-8 中的代码可以编译和运行，不过这并不是很有趣：通过它难以看出两个独立的线程在一个通道上相互通讯。列表 16-10 则有一些改进会证明这些代码是并发执行的：新建线程现在会发送多个消息并在每个消息之间暂停一段时间。

Filename: `src/main.rs`

```
use std::thread;
use std::sync::mpsc;
use std::time::Duration;

fn main() {
 let (tx, rx) = mpsc::channel();

 thread::spawn(move || {
 let vals = vec![
 String::from("hi"),
 String::from("from"),
 String::from("the"),
 String::from("thread"),
];

 for val in vals {
 tx.send(val).unwrap();
 thread::sleep(Duration::new(1, 0));
 }
 });

 for received in rx {
 println!("Got: {}", received);
 }
}
```

#### Listing 16-10: Sending multiple messages and pausing between each one

这一次，在新建线程中有一个字符串 `vector` 希望发送到主线程。我们遍历他们，单独的发送每一个字符串并通过一个 `Duration` 值调用 `thread::sleep` 函数来暂停一秒。

在主线程中，不再显式的调用 `recv` 函数：而是将 `rx` 当作一个迭代器。对于每一个接收到的值，我们将其打印出来。当通道被关闭时，迭代器也将结束。

当运行列表 16-10 中的代码时，将看到如下输出，每一行都会暂停一秒：

```
Got: hi
Got: from
Got: the
Got: thread
```

在主线程中并没有任何暂停或位于 `for` 循环中用于等待的代码，所以可以说主线程是在等待从新建线程中接收值。

## 通过克隆发送者来创建多个生产者

差不多在本部分的开头，我们提到了 `mpsc` 是 *multiple producer, single consumer* 的缩写。可以扩展列表 16-11 中的代码来创建都向同一接收者发送值的多个线程。这可以通过克隆通道的发送端在来做到，如列表 16-11 所示：

Filename: src/main.rs

```
use std::thread;
use std::sync::mpsc;
use std::time::Duration;
#
fn main() {
// ...snip...
let (tx, rx) = mpsc::channel();

let tx1 = tx.clone();
thread::spawn(move || {
 let vals = vec![
 String::from("hi"),
 String::from("from"),
 String::from("the"),
 String::from("thread"),
];

 for val in vals {
 tx1.send(val).unwrap();
 thread::sleep(Duration::new(1, 0));
 }
});

thread::spawn(move || {
 let vals = vec![
 String::from("more"),
 String::from("messages"),
 String::from("for"),
 String::from("you"),
];

 for val in vals {
 tx.send(val).unwrap();
 thread::sleep(Duration::new(1, 0));
 }
});
// ...snip...
```

```
#
for received in rx {
println!("Got: {}", received);
}
}
```

#### Listing 16-11: Sending multiple messages and pausing between each one

这一次，在创建新线程之前，我们对通道的发送端调用了 `clone` 方法。这会给我们一个可以传递给第一个新建线程的发送端句柄。我们会将原始的通道发送端传递给第二个新建线程，这样每个线程将向通道的接收端发送不同的消息。

如果运行这些代码，你可能会看到这样的输出：

```
Got: hi
Got: more
Got: from
Got: messages
Got: for
Got: the
Got: thread
Got: you
```

虽然你可能会看到这些以不同的顺序出现。这依赖于你的系统！这也就是并发既有趣又困难的原因。如果你拿 `thread::sleep` 做实验，在不同的线程中提供不同的值，就会发现他们的运行更加不确定并每次都会产生不同的输出。

现在我们见识过了通道如何工作，再看看共享内存并发吧。



# 共享状态并发

ch16-03-shared-state.md

commit 9df612e93e038b05fc959db393c15a5402033f47

虽然消息传递是一个很好的处理并发的方式，但并不是唯一的一个。再次考虑一下它的口号：

Do not communicate by sharing memory; instead, share memory by communicating.

不要共享内存来通讯；而是要通讯来共享内存。

那么“共享内存来通讯”是怎样的呢？共享内存并发有点像多所有权：多个线程可以同时访问相同的内存位置。第十五章介绍了智能指针如何使得多所有权成为可能，然而这会增加额外的复杂性，因为需要以某种方式管理这些不同的所有者。

不过 **Rust** 的类型系统和所有权可以很好的帮助我们，正确的管理它们。以共享内存中更常见的并发原语：互斥器（**mutexes**）为例，让我们看看具体的情况。

## 互斥器一次只允许一个线程访问数据

互斥器（**mutex**）是一种用于共享内存的并发原语。它是“**mutual exclusion**”的缩写，也就是说，任意时间，它只允许一个线程访问某些数据。互斥器以难以使用著称，因为你不得不记住：

1. 在使用数据之前尝试获取锁。
2. 处理完被互斥器所保护的数据之后，必须解锁数据，这样其他线程才能够获取锁。

现实中也有互斥器的例子，想象一下在一个会议中，只有一个麦克风。如果一个成员要发言，他必须请求使用麦克风。一旦得到了麦克风，他可以畅所欲言，然后将麦克风交给下一个希望讲话的成员。如果成员在没有麦克风的时候就开始叫喊，或者在其他成员发言结束之前就拿走麦克风，是很不合适的。如果这个共享的麦克风因为此类原因而出现问题，会议将无法正常进行。

正确的管理互斥器异常复杂，这也是许多人之所以热衷于通道的原因。然而，在 **Rust** 中，得益于类型系统和所有权，我们不会在锁和解锁上出错。

## Mutex<T> 的 API

让我们看看列表 16-12 中使用互斥器的例子，现在不涉及多线程：

Filename: src/main.rs

```
use std::sync::Mutex;

fn main() {
 let m = Mutex::new(5);

 {
 let mut num = m.lock().unwrap();
 *num = 6;
 }

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

Listing 16-12: Exploring the API of `Mutex<T>` in a single threaded context for simplicity

像很多类型一样，我们使用关联函数 `new` 来创建一个 `Mutex<T>`。使用 `lock` 方法获取锁，以访问互斥器中的数据。这个调用会阻塞，直到我们拥有锁为止。如果另一个线程拥有锁，并且那个线程 `panic` 了，则这个调用会失败。类似于列表 16-6 那样，我们暂时使用 `unwrap()` 进行错误处理，或者使用第九章中提及的更好的工具。

一旦获取了锁，就可以将返回值（在这里是 `num`）作为一个数据的可变引用使用了。观察 `Rust` 类型系统如何保证使用值之前必须获取锁：`Mutex<i32>` 并不是一个 `i32`，所以必须获取锁才能使用这个 `i32` 值。我们是不会忘记这么做的，因为类型系统不允许。

你也许会怀疑，`Mutex<T>` 是一个智能指针？是的！更准确的说，`lock` 调用返回一个叫做 `MutexGuard` 的智能指针。类似我们在第十五章见过的智能指针，它实现了 `Deref` 来指向其内部数据。另外 `MutexGuard` 有一个用来释放锁的 `Drop` 实现。这样就不会忘记释放锁了。这在 `MutexGuard` 离开作用域时会自动发生，例如它发生于列表 16-12 中内部作用域的结尾。接着可以打印出互斥器的值并发现能够将其内部的 `i32` 改为 6。

## 在线程间共享 `Mutex<T>`

现在让我们尝试使用 `Mutex<T>` 在多个线程间共享值。我们将启动十个线程，并在各个线程中对同一个计数器值加一，这样计数器将从 0 变为 10。注意，接下来的几个例子会出现编译错误，而我们将通过这些错误来学习如何使用 `Mutex<T>`，以及 `Rust` 又是如何辅助我们以确保正确。列表 16-13 是最开始的例子：

Filename: src/main.rs

```
use std::sync::Mutex;
use std::thread;

fn main() {
 let counter = Mutex::new(0);
 let mut handles = vec![];

 for _ in 0..10 {
 let handle = thread::spawn(|| {
 let mut num = counter.lock().unwrap();

 *num += 1;
 });
 handles.push(handle);
 }

 for handle in handles {
 handle.join().unwrap();
 }

 println!("Result: {}", *counter.lock().unwrap());
}
```

Listing 16-13: The start of a program having 10 threads each increment a counter guarded by a `Mutex<T>`

这里创建了一个 `counter` 变量来存放内含 `i32` 的 `Mutex<T>`，类似列表 16-12 那样。接下来使用 `range` 创建了 10 个线程。使用了 `thread::spawn` 并对所有线程使用了相同的闭包：他们每一个都将调用 `lock` 方法来获取 `Mutex<T>` 上的锁，接着将互斥器中的值加一。当一个线程结束执行，`num` 会离开闭包作用域并释放锁，这样另一个线程就可以获取它了。

在主线程中，我们像列表 16-2 那样收集了所有的 `join` 句柄，调用它们的 `join` 方法来确保所有线程都会结束。之后，主线程会获取锁并打印出程序的结果。

之前提示过这个例子不能编译，让我们看看为什么！

```
error[E0373]: closure may outlive the current function, but it borrows
`counter`, which is owned by the current function
```

```

-->
|
9 | let handle = thread::spawn(|| {
| ^^ may outlive borrowed value `counter`
10 | let mut num = counter.lock().unwrap();
| ----- `counter` is borrowed here
|
help: to force the closure to take ownership of `counter` (and any other
referenced variables), use the `move` keyword, as shown:
| let handle = thread::spawn(move || {

```

这类似于列表 16-5 中解决了的问题。考虑到启动了多个线程，**Rust** 无法知道这些线程会运行多久，而在每一个线程尝试借用 `counter` 时它是否仍然有效。帮助信息提醒了我们如何解决它：可以使用 `move` 来给予每个线程其所有权。尝试在闭包上做一点改动：

```
thread::spawn(move || {
```

再次编译。这回出现了一个不同的错误！

```

error[E0382]: capture of moved value: `counter`
-->
|
9 | let handle = thread::spawn(move || {
| ----- value moved (into closure) here
10 | let mut num = counter.lock().unwrap();
| ^^^^^^^ value captured here after move
|
= note: move occurs because `counter` has type `std::sync::Mutex<i32>`,
which does not implement the `Copy` trait

error[E0382]: use of moved value: `counter`
-->
|
9 | let handle = thread::spawn(move || {
| ----- value moved (into closure) here
...
21 | println!("Result: {}", *counter.lock().unwrap());
| ^^^^^^^ value used here after move
|
= note: move occurs because `counter` has type `std::sync::Mutex<i32>`,
which does not implement the `Copy` trait

error: aborting due to 2 previous errors

```

`move` 并没有像列表 16-5 中那样解决问题。为什么呢？错误信息有点难懂，因为它表明 `counter` 被移动进了闭包，接着它在调用 `lock` 时被捕获。这似乎是我们希望的，然而不被允许。

让我们推理一下。这次不再使用 `for` 循环创建 10 个线程，只创建两个线程，看看会发生什么。将列表 16-13 中第一个 `for` 循环替换为如下代码：

```

let handle = thread::spawn(move || {
 let mut num = counter.lock().unwrap();

 *num += 1;
});
handles.push(handle);

let handle2 = thread::spawn(move || {
 let mut num2 = counter.lock().unwrap();

```

```

 *num2 += 1;
 });
 handles.push(handle2);

```

这里创建了两个线程，并将第二个线程所用的变量改名为 `handle2` 和 `num2`。我们简化了例子，看是否能理解错误信息。此次编译给出如下信息：

```

error[E0382]: capture of moved value: `counter`
 -->
 |
8 | let handle = thread::spawn(move || {
 | ----- value moved (into closure) here
...
16 | let mut num = counter.lock().unwrap();
 | ^^^^^^^ value captured here after move
 |
 = note: move occurs because `counter` has type `std::sync::Mutex<i32>`,
 which does not implement the `Copy` trait

error[E0382]: use of moved value: `counter`
 -->
 |
8 | let handle = thread::spawn(move || {
 | ----- value moved (into closure) here
...
26 | println!("Result: {}", *counter.lock().unwrap());
 | ^^^^^^^ value used here after move
 |
 = note: move occurs because `counter` has type `std::sync::Mutex<i32>`,
 which does not implement the `Copy` trait

error: aborting due to 2 previous errors

```

啊哈！第一个错误信息中说，`counter` 被移动进了 `handle` 所代表线程的闭包中。因此我们无法在第二个线程中对其调用 `lock`，并将结果储存在 `num2` 中时捕获 `counter`！所以 **Rust** 告诉我们不能将 `counter` 的所有权移动到多个线程中。这在之前很难看出，因为我们在循环中创建了多个线程，而 **Rust** 无法在每次迭代中指明不同的线程（没有临时变量 `num2`）。

## 多线程和多所有权

在第十五章中，我们通过使用智能指针 `Rc<T>` 来创建引用计数的值，以便拥有多所有权。同时第十五章提到了 `Rc<T>` 只能在单线程环境中使用，不过还是在这里试用 `Rc<T>` 看看会发生什么。列表 16-14 将 `Mutex<T>` 装进了 `Rc<T>` 中，并在移入线程之前克隆了 `Rc<T>`。再用循环来创建线程，保留闭包中的 `move` 关键字：

Filename: src/main.rs

```

use std::rc::Rc;
use std::sync::Mutex;
use std::thread;

fn main() {
 let counter = Rc::new(Mutex::new(0));
 let mut handles = vec![];

 for _ in 0..10 {
 let counter = counter.clone();
 let handle = thread::spawn(move || {
 let mut num = counter.lock().unwrap();

 *num += 1;

```

```

 });
 handles.push(handle);
}

for handle in handles {
 handle.join().unwrap();
}

println!("Result: {}", *counter.lock().unwrap());
}

```

Listing 16-14: Attempting to use `Rc<T>` to allow multiple threads to own the `Mutex<T>`

再一次编译并...出现了不同的错误！编译器真是教会了我们很多！

```

error[E0277]: the trait bound `std::rc::Rc<std::sync::Mutex<i32>>:
std::marker::Send` is not satisfied
-->
|
11 | let handle = thread::spawn(move || {
| ^^^^^^^^^^^^^ the trait `std::marker::Send` is not
implemented for `std::rc::Rc<std::sync::Mutex<i32>>`
|
= note: `std::rc::Rc<std::sync::Mutex<i32>>` cannot be sent between threads
safely
= note: required because it appears within the type
`[closure@src/main.rs:11:36: 15:10
counter:std::rc::Rc<std::sync::Mutex<i32>>]`
= note: required by `std::thread::spawn`

```

哇哦，太长不看！说重点：第一个提示表明 `Rc<Mutex<i32>>` 不能安全的在线程间传递。理由也在错误信息中，“不满足 `Send trait bound`”（the trait bound `Send` is not satisfied）。下一部分将会讨论 `Send`，它是确保许多用在多线程中的类型，能够适合并发环境的 `trait` 之一。

不幸的是，`Rc<T>` 并不能安全的在线程间共享。当 `Rc<T>` 管理引用计数时，它必须在每一个 `clone` 调用时增加计数，并在每一个克隆被丢弃时减少计数。`Rc<T>` 并没有使用任何并发原语，来确保改变计数的操作不会被其他线程打断。在计数出错时可能会导致诡异的 `bug`，比如可能会造成内存泄漏，或在使用结束之前就丢弃一个值。如果一个类型与 `Rc<T>` 相似，又以一种线程安全的方式改变引用计数，会怎么样呢？

## 原子引用计数 `Arc<T>`

答案是肯定的，确实有一个类似 `Rc<T>` 并可以安全的用于并发环境的类型：`Arc<T>`。字母“a”代表原子性（*atomic*），所以这是一个原子引用计数（*atomically reference counted*）类型。原子性是另一类这里还未涉及到的并发原语；请查看标准库中 `std::sync::atomic` 的文档来获取更多细节。其中的要点就是：原子性类型工作起来类似原始类型，不过可以安全的在线程间共享。

为什么不是所有的原始类型都是原子性的？为什么不是所有标准库中的类型都默认使用 `Arc<T>` 实现？线程安全带来性能惩罚，我们希望只在必要时才为此买单。如果只是在单线程中对值进行操作，原子性提供的保证并无必要，代码可以因此运行的更快。

回到之前的例子：`Arc<T>` 和 `Rc<T>` 除了 `Arc<T>` 内部的原子性之外没有区别。其 `API` 也相同，所以可以修改 `use` 行和 `new` 调用。列表 16-15 中的代码最终可以编译和运行：

Filename: src/main.rs

```

use std::sync::{Mutex, Arc};
use std::thread;

```

```
fn main() {
 let counter = Arc::new(Mutex::new(0));
 let mut handles = vec![];

 for _ in 0..10 {
 let counter = counter.clone();
 let handle = thread::spawn(move || {
 let mut num = counter.lock().unwrap();

 *num += 1;
 });
 handles.push(handle);
 }

 for handle in handles {
 handle.join().unwrap();
 }

 println!("Result: {}", *counter.lock().unwrap());
}
```

Listing 16-15: Using an `Arc<T>` to wrap the `Mutex<T>` to be able to share ownership across multiple threads

这会打印出：

```
Result: 10
```

成功了！我们从 0 数到了 10，这可能并不是很显眼，不过一路上我们学习了很多关于 `Mutex<T>` 和线程安全的内容！这个例子中构建的结构可以用于比增加计数更为复杂的操作。能够被分解为独立部分的计算可以像这样被分散到多个线程中，并可以使用 `Mutex<T>` 来允许每个线程在他们自己的部分更新最终的结果。

你可能注意到了，因为 `counter` 是不可变的，不过可以获取其内部值的可变引用，这意味着 `Mutex<T>` 提供了内部可变性，就像 `Cell` 系列类型那样。正如第十五章中使用 `RefCell<T>` 可以改变 `Rc<T>` 中的内容那样，同样的可以使用 `Mutex<T>` 来改变 `Arc<T>` 中的内容。

回忆一下 `Rc<T>` 并没有避免所有可能的问题：我们也讨论了当两个 `Rc<T>` 相互引用时的引用循环的可能性，这可能造成内存泄露。`Mutex<T>` 有一个类似的问题：死锁。死锁（*deadlock*）是一个场景中操作需要锁定两个资源，而两个线程分别拥有一个锁并永远相互等待的问题。如果你对这个主题感兴趣，尝试编写一个带有死锁的 `Rust` 程序，接着研究任何其他语言中使用互斥器的死锁规避策略并尝试在 `Rust` 中实现他们。标准库中 `Mutex<T>` 和 `MutexGuard` 的 API 文档会提供有用的信息。

`Rust` 的类型系统和所有权规则，确保了线程在更新共享值时拥有独占的访问权限，所以线程不会以不可预测的方式覆盖彼此的操作。虽然为了使一切正确运行而在编译器上花了一些时间，但是我们节省了未来的时间，尤其是线程以特定顺序执行才会出现的诡异错误难以重现。

接下来，为了丰富本章的内容，让我们讨论一下 `Send` 和 `Sync` `trait` 以及如何对自定义类型使用他们。

# 使用 `Sync` 和 `Send` `trait` 的可扩展并发

[ch16-04-extensible-concurrency-sync-and-send.md](#)

commit 9430a3d28a2121a938d704ce48b15d21062f880e

Rust 的并发模型中一个有趣的方面是：语言本身对并发知之甚少。我们之前讨论的几乎所有内容，都属于标准库，而不是语言本身的内容。由于不需要语言提供并发相关的基础设施，并发方案不受标准库或语言所限：我们可以编写自己的或使用别人编写的。

我们说“几乎所有内容都不属于语言本身”，那么属于语言本身的是什么呢？是两个 `trait`，都位于 `std::marker`：

`Sync` 和 `Send`。

## `Send` 用于表明所有权可能被传送给其他线程

`Send` 标记 `trait` 表明类型的所有权可能被在线程间传递。几乎所有的 Rust 类型都是 `Send` 的，不过有一些例外。比如标准库中提供的 `Rc<T>`：如果克隆 `Rc<T>` 值，并尝试将克隆的所有权传递给另一个线程，这两个线程可能会同时更新引用计数。正如上一部分提到的，`Rc<T>` 被实现为用于单线程场景，这时不需要为拥有线程安全的引用计数而付出性能代价。

因为 `Rc<T>` 没有标记为 `Send`，Rust 的类型系统和 `trait bound` 会确保我们不会错误的把一个 `Rc<T>` 值不安全的在线程间传递。列表 16-14 曾尝试这么做，不过得到了一个错误，`the trait Send is not implemented for Rc<Mutex<i32>>`。而使用标记为 `Send` 的 `Arc<T>` 时，就没有问题了。

任何完全由 `Send` 的类型组成的类型也会自动被标记为 `Send`：几乎所有基本类型都是 `Send` 的，大部分标准库类型是 `Send` 的，除了 `Rc<T>`，以及第十九章将会讨论的裸指针（raw pointer）。

## `Sync` 表明多线程访问是安全的

`Sync` 标记 `trait` 表明一个类型可以安全的在多个线程中拥有其值的引用。换一种方式来说，对于任意类型 `T`，如果 `&T`（`T` 的引用）是 `Send` 的话 `T` 就是 `Sync` 的，这样其引用就可以安全的发送到另一个线程。类似于 `Send` 的情况，基本类型是 `Sync` 的，完全由 `Sync` 的类型组成的类型也是 `Sync` 的。

`Rc<T>` 也不是 `Sync` 的，出于其不是 `Send` 的相同的原因。`RefCell<T>`（第十五章讨论过）和 `Cell<T>` 系列类型不是 `Sync` 的。`RefCell<T>` 在运行时所进行的借用检查也不是线程安全的。`Mutex<T>` 是 `Sync` 的，正如上一部分所讲的它可以被用来在多线程中共享访问。

## 手动实现 `Send` 和 `Sync` 是不安全的

通常并不需要实现 `Send` 和 `Sync` `trait`，由属于 `Send` 和 `Sync` 的类型组成的类型，自动就是 `Send` 和 `Sync` 的。因为他们是标记 `trait`，甚至都不需要实现任何方法。他们只是用来加强并发相关的不可变性的。

实现这些标记 `trait` 涉及到编写不安全的 Rust 代码，第十九章将会讲述具体的方法；当前重要的是，在创建新的由不是 `Send` 和 `Sync` 的部分构成的并发类型时需要多加小心，以确保维持其安全保证。[The Nomicon](#) 中有更多关于这些保证以及如何维持他们的信息。

## 总结

这不会是本书最后一个出现并发的章节；第二十章的项目会在更现实的场景中使用这些概念，而不像本章中讨论的这些小子。

正如我们提到的，因为 **Rust** 本身很少有处理并发的部分内容，有很多的并发方案都由 **crate** 实现。他们比标准库要发展的更快；请在网上搜索当前最新的用于多线程场景的 **crate**。

**Rust** 提供了用于消息传递的通道，和像 `Mutex<T>` 和 `Arc<T>` 这样可以安全的用于并发上下文的智能指针。类型系统和借用检查器会确保这些场景中的代码，不会出现数据竞争和无效的引用。一旦代码可以编译了，我们就可以坚信这些代码可以正确的运行于多线程环境，而不会出现其他语言中经常出现的那些难以追踪的 **bug**。并发编程不再是什么可怕的概念：无所畏惧地并发吧！

接下来，让我们讨论一下当 **Rust** 程序变得更大时，有哪些符合语言习惯的问题建模方法和结构化解决方案，以及 **Rust** 的风格是如何与面向对象编程（**Object Oriented Programming**）中那些你所熟悉的概念相联系的。



# Rust 是一个面向对象的编程语言吗？

[ch17-00-oop.md](#)

commit 759801361bde74b47e81755fff545c66020e6e63

面向对象编程（Object-Oriented Programming）是一种起源于 20 世纪 60 年代的 Simula 编程语言的模式化编程方式，然后在 90 年代随着 C++ 语言开始流行。关于 OOP 是什么有很多相互矛盾的定义：在一些定义下，Rust 是面向对象的；在其他定义下，Rust 不是。在本章节中，我们会探索一些被普遍认为是面向对象的特性和这些特性是如何体现在 Rust 语言习惯中的。

# 什么是面向对象？

ch17-01-what-is-oo.md

commit 2a9b2a1b019ad6d4832ff3e56fbcba5be68b250e

关于一个语言被称为面向对象所需的功能，在编程社区内并未达成一致意见。**Rust** 被很多不同的编程范式影响；我们探索了十三章提到的来自函数式编程的特性。面向对象编程语言所共享的一些特性往往是对象、封装和继承。让我们看一下这每一个概念的含义以及 **Rust** 是否支持他们。

## 对象包含数据和行为

**Design Patterns: Elements of Reusable Object-Oriented Software** 这本书被俗称为 **The Gang of Four book**，是面向对象编程模式的目录。它这样定义面向对象编程：

Object-oriented programs are made up of objects. An *object* packages both data and the procedures that operate on that data. The procedures are typically called *methods* or *operations*.

面向对象的程序是由对象组成的。一个对象包含数据和操作这些数据的过程。这些过程通常被称为方法或操作。

在这个定义下，**Rust** 是面向对象的：结构体和枚举包含数据而 **impl** 块提供了在结构体和枚举之上的方法。虽然带有方法的结构体和枚举并不被称为对象，但是他们提供了与对象相同的功能，参考 **Gang of Four** 中对象的定义。

## 隐藏了实现细节的封装

另一个通常与面向对象编程相关的方面是封装（*encapsulation*）的思想：对象的实现细节不能被使用对象的代码获取到。唯一与对象交互的方式是通过对象提供的公有 **API**；使用对象的代码无法深入到对象内部并直接改变数据或者行为。封装使得改变和重构对象的内部时无需改变使用对象的代码。

就像我们在第七章讨论的那样，可以使用 **pub** 关键字来决定模块、类型函数和方法是公有的，而默认情况下一切都是私有的。比如，我们可以定义一个包含一个 **i32** 类型的 **vector** 的结构体 **AveragedCollection**。结构体也可以有一个字段，该字段保存了 **vector** 中所有值的平均值。这样，希望知道结构体中的 **vector** 的平均值的人可以随时获取它，而无需自己计算。**AveragedCollection** 会为我们缓存平均值结果。列表 17-1 有 **AveragedCollection** 结构体的定义：

文件名: **src/lib.rs**

```
pub struct AveragedCollection {
 list: Vec<i32>,
 average: f64,
}
```

列表 17-1: **AveragedCollection** 结构体维护了一个整型列表和集合中所有元素的平均值。

注意，结构体自身被标记为 **pub**，这样其他代码可以使用这个结构体，但是在结构体内部的字段仍然是私有的。这是非常重要的，因为我们希望保证变量被增加到列表或者被从列表删除时，也会同时更新平均值。可以通过在结构体上实现 **add**、**remove** 和 **average** 方法来做到这一点，如列表 17-2 所示：

文件名: **src/lib.rs**

```
pub struct AveragedCollection {
list: Vec<i32>,
average: f64,
```

```
}
impl AveragedCollection {
 pub fn add(&mut self, value: i32) {
 self.list.push(value);
 self.update_average();
 }

 pub fn remove(&mut self) -> Option<i32> {
 let result = self.list.pop();
 match result {
 Some(value) => {
 self.update_average();
 Some(value)
 },
 None => None,
 }
 }

 pub fn average(&self) -> f64 {
 self.average
 }

 fn update_average(&mut self) {
 let total: i32 = self.list.iter().sum();
 self.average = total as f64 / self.list.len() as f64;
 }
}
```

列表 17-2: 在 `AveragedCollection` 结构体上实现了 `add`、`remove` 和 `average` 公有方法

公有方法 `add`、`remove` 和 `average` 是修改 `AveragedCollection` 实例的唯一方式。当使用 `add` 方法把一个元素加入到 `list` 或者使用 `remove` 方法来删除它时，这些方法的实现同时会调用私有的 `update_average` 方法来更新 `average` 字段。因为 `list` 和 `average` 是私有的，没有其他方式来使得外部的代码直接向 `list` 增加或者删除元素，直接操作 `list` 可能会引发 `average` 字段不同步。`average` 方法返回 `average` 字段的值，这使得外部的代码只能读取 `average` 而不能修改它。

因为我们已经封装好了 `AveragedCollection` 的实现细节，将来可以轻松改变类似数据结构这些方面的内容。例如，可以使用 `HashSet` 代替 `Vec` 作为 `list` 字段的类型。只要 `add`、`remove` 和 `average` 公有函数的签名保持不变，使用 `AveragedCollection` 的代码就无需改变。如果将 `list` 暴露给外部代码时，未必都是这样，因为 `HashSet` 和 `Vec` 使用不同的方法增加或移除项，所以如果要想直接修改 `list` 的话，外部的代码可能不得不修改。

如果封装是一个语言被认为是面向对象语言所必要的方面的话，那么 `Rust` 就满足这个要求。在代码中不同的部分使用或者不使用 `pub` 决定了实现细节的封装。

## 作为类型系统的继承和作为代码共享的继承

继承 (*Inheritance*) 是一个很多编程语言都提供的机制，一个对象可以定义为继承另一个对象的定义，这使其可以获得父对象的数据和行为，而不用重新定义。一些人定义面向对象语言时，认为继承是一个特色。

如果一个语言必须有继承才能被称为面向对象语言的话，那么 `Rust` 就不是面向对象的。无法定义一个结构体继承自另外一个结构体，从而获得父结构体的成员和方法。然而，如果你过去常常在你的编程工具箱使用继承，根据你希望使用继承的原因，`Rust` 也提供了其他的解决方案。

使用继承有两个主要的原因。第一个是为了重用代码：一旦为一个类型实现了特定行为，继承可以对一个不同的类型重用这个实现。相反 `Rust` 代码可以使用默认 `trait` 方法实现来进行共享，在列表 10-14 中我们见过在 `Summarizable trait` 上增加的 `summary` 方法的默认实现。任何实现了 `Summarizable trait` 的类型都可以使

用 `summary` 方法而无须进一步实现。这类似于父类有一个方法的实现，而通过继承子类也拥有这个方法的实现。当实现 `Summarizable` `trait` 时也可以选择覆盖 `summary` 的默认实现，这类似于子类覆盖从父类继承的方法实现。

第二个使用继承的原因与类型系统有关：用来表现子类型可以在父类型被使用的地方使用。这也被称为多态（*polymorphism*），意味着如果多种对象有一个相同的形态大小，它们可以替代使用。

虽然很多人使用“多态”（*polymorphism*）来描述继承，但是它实际上是一种特殊的多态，称为“子类型多态”（*sub-type polymorphism*）。也有很多种其他形式的多态，在 Rust 中带有泛型参数的 `trait bound` 也是多态，更具体的说是“参数多态”（*parametric polymorphism*）。不同类型多态的确切细节在这里并不关键，所以不要过于担心细节，只需要知道 Rust 有多种多态相关的特色就好，不同于很多其他 OOP 语言。

为了支持这种模式，Rust 有 `trait` 对象（*trait objects*），这样就可以使用任意类型的值，只要这个值实现了指定的 `trait`。

继承最近在很多编程语言的设计方案中失宠了。使用继承来实现代码重用，会共享更多非必需的代码。子类不应该总是共享其父类的所有特性，然而继承意味着子类得到了其父类全部的数据和行为。这使得程序的设计更不灵活，并产生了无意义的方法调用或子类，以及由于方法并不适用于子类，却必需从父类继承而可能造成的错误。另外，某些语言只允许子类继承一个父类，进一步限制了程序设计的灵活性。

因为这些原因，Rust 选择了一个另外的途径，使用 `trait` 对象替代继承。让我们看一下在 Rust 中 `trait` 对象是如何实现多态的。

# 为使用不同类型的值而设计的 **trait** 对象

ch17-02-trait-objects.md

commit 67876e3ef5323ce9d394f3ea6b08cb3d173d9ba9

在第八章中，我们谈到了 **vector** 只能存储同种类型元素的局限。在列表 8-1 中有一个例子，其中定义了一个拥有分别存放整型、浮点型和文本型成员的枚举类型 `SpreadsheetCell`，使用这个枚举的 **vector** 可以在每一个单元格（`cell`）中储存不同类型的数据，并使得 **vector** 整体仍然代表一行（`row`）单元格。这当编译代码时就知道希望可以交替使用的类型为固定集合的情况下是可行的。

有时，我们希望使用的类型的集合对于使用库的程序员来说是可扩展的。例如，很多图形用户接口（**GUI**）工具具有一个项目列表的概念，它通过遍历列表并调用每一个项目的 `draw` 方法来将其绘制到屏幕上。我们将要创建一个叫做 `rust_gui` 的库 `crate`，它含一个 **GUI** 库的结构。这个 **GUI** 库包含一些可供开发者使用的类型，比如 `Button` 或 `TextField`。使用 `rust_gui` 的程序员会想要创建更多可以绘制在屏幕上的类型：其中一些可能会增加一个 `Image`，而另一些可能会增加一个 `SelectBox`。本章节并不准备实现一个功能完善的 **GUI** 库，不过会展示其中各个部分是如何结合在一起的。

编写 `rust_gui` 库时，我们并不知道其他程序员想要创建的全部类型，所以无法定义一个 `enum` 来包含所有这些类型。我们所要做的是使 `rust_gui` 能够记录一系列不同类型的值，并能够对其中每一个值调用 `draw` 方法。**GUI** 库不需要知道当调用 `draw` 方法时具体会发生什么，只需提供这些值可供调用的方法即可。

在拥有继承的语言中，我们可能定义一个名为 `Component` 的类，该类上有一个 `draw` 方法。其他的类比如 `Button`、`Image` 和 `SelectBox` 会从 `Component` 派生并因此继承 `draw` 方法。它们各自都可以覆盖 `draw` 方法来定义自己的行为，但是框架会把所有这些类型当作是 `Component` 的实例，并在其上调用 `draw`。

## 定义通用行为的 **trait**

不过，在 **Rust** 中，我们可以定义一个 `Draw trait`，包含名为 `draw` 的方法。接着可以定义一个存放 **trait** 对象（*trait object*）的 **vector**，**trait** 对象是一个位于某些指针，比如 `&` 引用或 `Box<T>` 智能指针，之后的 **trait**。第十九章会讲到为何 **trait** 对象必须位于指针之后的原因。

之前提到过，我们并不将结构体与枚举称之为“对象”，以便与其他语言中的对象相区别。结构体与枚举和 `impl` 块中的行为是分开的，不同于其他语言中将数据和行为组合进一个称为对象的概念中。**trait** 对象将由指向具体对象的指针构成的数据和定义于 **trait** 中方法的行为结合在一起，从这种意义上说它则更类似其他语言中的对象。不过 **trait** 对象与其他语言中的对象是不同的，因为不能向 **trait** 对象增加数据。**trait** 对象并不像其他语言中的对象那么通用：他们（**trait** 对象）的作用是允许对通用行为的抽象。

**trait** 对象定义了给定情况下所需的行为。接着就可以在要使用具体类型或泛型的地方使用 **trait** 来作为 **trait** 对象。**Rust** 的类型系统会确保任何我们替换为 **trait** 对象的值都会实现了 **trait** 的方法。这样就无需在编译时就知道所有可能的类型，就能够用同样的方法处理所有的实例。列表 17-3 展示了如何定义一个带有 `draw` 方法的 **trait**

Draw :

文件名: `src/lib.rs`

```
pub trait Draw {
 fn draw(&self);
}
```

列表 17-3: `Draw trait` 的定义

因为第十章已经讨论过如何定义 **trait**，这看起来应该比较眼熟。接下来就是新内容了：列表 17-4 有一个名为 **Screen** 的结构体定义，它存放了一个叫做 **components** 的 **Box<Draw>** 类型的 **vector**。**Box<Draw>** 是一个 **trait** 对象：它是 **Box** 中任何实现了 **Draw trait** 的类型的替身。

文件名: **src/lib.rs**

```
pub trait Draw {
fn draw(&self);
}
#
pub struct Screen {
 pub components: Vec<Box<Draw>>,
}
```

列表 17-4: 一个 **Screen** 结构体的定义，它带有一个字段 **components**，其包含实现了 **Draw trait** 的 **trait** 对象的 **vector**

在 **Screen** 结构体上，我们将定义一个 **run** 方法，该方法会对其 **components** 上的每一个元素调用 **draw** 方法，如列表 17-5 所示：

文件名: **src/lib.rs**

```
pub trait Draw {
fn draw(&self);
}
#
pub struct Screen {
pub components: Vec<Box<Draw>>,
}
#
impl Screen {
 pub fn run(&self) {
 for component in self.components.iter() {
 component.draw();
 }
 }
}
```

列表 17-5: 在 **Screen** 上实现一个 **run** 方法，该方法在每个 **component** 上调用 **draw** 方法

这与定义使用了带有 **trait bound** 的泛型类型参数的结构体不同。泛型类型参数一次只能替代一个具体的类型，而 **trait** 对象则允许在运行时替代多种具体类型。例如，可以像列表 17-6 那样定义使用泛型和 **trait bound** 的结构体 **Screen**：

文件名: **src/lib.rs**

```
pub trait Draw {
fn draw(&self);
}
#
pub struct Screen<T: Draw> {
 pub components: Vec<T>,
}

impl<T> Screen<T>
where T: Draw {
 pub fn run(&self) {
 for component in self.components.iter() {
 component.draw();
 }
 }
}
```

```
}
}
```

列表 17-6: 一种 `Screen` 结构体的替代实现，它的 `run` 方法使用泛型和 `trait bound`

这只允许我们拥有一个包含全是 `Button` 类型或者全是 `TextField` 类型的 `component` 列表的 `Screen` 实例。如果只拥有相同类型的集合，那么使用泛型和 `trait bound` 是更好的，因为在编译时使用具体类型其定义是单态（`monomorphized`）的。

相反对于存放了 `Vec<Box<Draw>>` `trait` 对象的 `component` 列表的 `Screen` 定义，一个 `Screen` 实例可以存放一个既可以包含 `Box<Button>`，也可以包含 `Box<TextField>` 的 `Vec`。让我们看看它是如何工作的，接着会讲到其运行时性能影响。

## 来自我们或者库使用者的 `trait` 实现

现在来增加一些实现了 `Draw trait` 的类型。我们将提供 `Button` 类型，再一次重申，真正实现 `GUI` 库超出了本书的范畴，所以 `draw` 方法体中不会有任何有意义的实现。为了想象一下这个实现看起来像什么，一个 `Button` 结构体可能会拥有 `width`、`height` 和 `label` 字段，如列表 17-7 所示：

文件名: `src/lib.rs`

```
pub trait Draw {
fn draw(&self);
}
#
pub struct Button {
 pub width: u32,
 pub height: u32,
 pub label: String,
}

impl Draw for Button {
 fn draw(&self) {
 // Code to actually draw a button
 }
}
```

列表 17-7: 一个实现了 `Draw trait` 的 `Button` 结构体

在 `Button` 上的 `width`、`height` 和 `label` 字段会和其他组件不同，比如 `TextField` 可能有 `width`、`height`、`label` 以及 `placeholder` 字段。每一个我们希望能在屏幕上绘制的类型都会使用不同的代码来实现 `Draw trait` 的 `draw` 方法，来定义如何绘制像这里的 `Button` 类型（并不包含任何实际的 `GUI` 代码，这超出了本章的范畴）。除了实现 `Draw trait` 之外，`Button` 还可能有一个包含按钮点击如何响应的方法的 `impl` 块。这类方法并不适用于像 `TextField` 这样的类型。

一些库的使用者决定实现一个包含 `width`、`height` 和 `options` 字段的结构体 `SelectBox`。并也为其实现了 `Draw trait`，如列表 17-8 所示：

文件名: `src/main.rs`

```
extern crate rust_gui;
use rust_gui::Draw;

struct SelectBox {
 width: u32,
 height: u32,
 options: Vec<String>,
}
```

```
impl Draw for SelectBox {
 fn draw(&self) {
 // Code to actually draw a select box
 }
}
```

列表 17-8: 在另一个使用 `rust_gui` 的 `crate` 中, 在 `SelectBox` 结构体上实现 `Draw` `trait`

库使用者现在可以在他们的 `main` 函数中创建一个 `Screen` 实例, 并通过将 `SelectBox` 和 `Button` 放入 `Box<T>` 转变为 `trait` 对象来将它们放入屏幕实例。接着可以调用 `Screen` 的 `run` 方法, 它会调用每个组件的 `draw` 方法。列表 17-9 展示了这个实现:

文件名: `src/main.rs`

```
use rust_gui::{Screen, Button};

fn main() {
 let screen = Screen {
 components: vec![
 Box::new(SelectBox {
 width: 75,
 height: 10,
 options: vec![
 String::from("Yes"),
 String::from("Maybe"),
 String::from("No")
],
 }),
 Box::new(Button {
 width: 50,
 height: 10,
 label: String::from("OK"),
 }),
],
 };

 screen.run();
}
```

列表 17-9: 使用 `trait` 对象来存储实现了相同 `trait` 的不同类型的值

即使我们不知道何时何人会增加 `SelectBox` 类型, `Screen` 的实现能够操作 `SelectBox` 并绘制它, 因为 `SelectBox` 实现了 `Draw` `trait`, 这意味着它实现了 `draw` 方法。

只关心值所反映的信息而不是值的具体类型, 这类似于动态类型语言中称为鸭子类型 (*duck typing*) 的概念: 如果它走起来像一只鸭子, 叫起来像一只鸭子, 那么它就是一只鸭子! 在列表 17-5 中 `Screen` 上的 `run` 实现中, `run` 并不需要知道各个组件的具体类型是什么。它并不检查组件实例是 `Button` 或者是 `SelectBox`, 它只是调用组件上的 `draw` 方法。通过指定 `Box<Draw>` 作为 `components` `vector` 中值的类型, 我们就定义了 `Screen` 需要可以在其上调用 `draw` 方法的值。

使用 `trait` 对象和 `Rust` 类型系统来使用鸭子类型的优势是无需在运行时检查一个值是否实现了特定方法或者担心在调用时因为值没有实现方法而产生错误。如果值没有实现 `trait` 对象所需的 `trait` 则 `Rust` 不会编译这些代码。

例如, 列表 17-10 展示了当创建一个使用 `String` 做为其组件的 `Screen` 时发生的情况:

文件名: `src/main.rs`

```
extern crate rust_gui;
use rust_gui::Draw;
```



```
fn main() {
 let screen = Screen {
 components: vec![
 Box::new(String::from("Hi")),
],
 };

 screen.run();
}
```

列表 17-10: 尝试使用一种没有实现 `trait` 对象的 `trait` 的类型

我们会遇到这个错误，因为 `String` 没有实现 `Draw trait`:

```
error[E0277]: the trait bound `std::string::String: Draw` is not satisfied
-->
|
4 | Box::new(String::from("Hi")),
| ^^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait `Draw` is not
 implemented for `std::string::String`
|
 = note: required for the cast to the object type `Draw`
```

这告诉了我们，要么是我们传递了并不希望传递给 `Screen` 的类型并应该提供其他类型，要么应该在 `String` 上实现 `Draw` 以便 `Screen` 可以调用其上的 `draw`。

## trait 对象执行动态分发

回忆一下第十章讨论过的，当对泛型使用 `trait bound` 时编译器所进行单态化处理：编译器为每一个被泛型类型参数代替的具体类型生成了非泛型的函数和方法实现。单态化所产生的代码进行静态分发（*static dispatch*）：当方法被调用时，伴随方法调用的代码在编译时就被确定了，同时寻找这些代码是非常快速的。

当使用 `trait` 对象时，编译器并不进行单态化，因为并不知道所有可能会使用这些代码的类型。相反，`Rust` 记录当方法被调用时可能会用到的代码，并在运行时计算出特定方法调用时所需的代码。这被称为动态分发（*dynamic dispatch*），进行这种代码搜寻是有运行时开销的。动态分发也阻止编译有选择的内联方法的代码，这会禁用一些优化。尽管在编写和支持代码的过程中确实获得了额外的灵活性，但仍然需要权衡取舍。

## Trait 对象要求对象安全

不是所有的 `trait` 都可以被放进 `trait` 对象中；只有对象安全（*object safe*）的 `trait` 才可以。一个 `trait` 只有同时满足如下两点时才被认为是对象安全的：

- `trait` 不要求 `Self` 是 `Sized` 的
- 所有的 `trait` 方法都是对象安全的

`Self` 关键字是我们要实现 `trait` 或方法的类型的别名。`Sized` 是一个类似第十六章中介绍的 `Send` 和 `Sync` 那样的标记 `trait`。`Sized` 会自动为在编译时有已知大小的类型实现，比如 `i32` 和引用。包括 `slice`（`[T]`）和 `trait` 对象这样的没有已知大小的类型则没有。

`Sized` 是一个所有泛型参数类型默认的隐含 `trait bound`。`Rust` 中大部分实用的操作都要求类型是 `Sized` 的，所以将 `Sized` 作为默认 `trait bound` 要求，就可以不必在每一次使用泛型时编写 `T: Sized` 了。然而，如果想要使用在 `slice` 上使用 `trait`，则需要去掉 `Sized trait bound`，可以通过指定 `T: ?Sized` 作为 `trait bound` 来做到这一点。

`trait` 有一个默认的 `bound Self: ?Sized`，这意味着他们可以在是或者不是 `Sized` 的类型上实现。如果创建了一个去掉了 `Self: ?Sized bound` 的 `trait Foo`，它可能看起来像这样：

```
trait Foo: Sized {
 fn some_method(&self);
}
```

`trait Sized` 现在就是 `trait Foo` 的父 `trait` (*supertrait*) 了, 也就意味着 `trait Foo` 要求实现 `Foo` 的类型 (也就是 `Self`) 是 `Sized` 的。我们将在第十九章中更详细的介绍父 `trait`。

像 `Foo` 这样要求 `Self` 是 `Sized` 的 `trait` 不被允许成为 `trait` 对象的原因是, 不可能为 `trait` 对象实现 `Foo trait`: `trait` 对象不是 `Sized` 的, 但是 `Foo` 又要求 `Self` 是 `Sized` 的。一个类型不可能同时既是有确定大小的又是无确定大小的。

关于第二条对象安全要求说到 `trait` 的所有方法都必须是对象安全的, 一个对象安全的方法满足下列条件之一:

- 要求 `Self` 是 `Sized` 的, 或者
- 满足如下三点:
  - 必须不包含任何泛型类型参数
  - 其第一个参数必须是 `Self` 类型或者能解引用为 `Self` 的类型 (也就是说它必须是一个方法而非关联函数, 并且以 `self`、`&self` 或 `&mut self` 作为第一个参数)
  - 必须不能在方法签名中除第一个参数之外的地方使用 `Self`

虽然这些规则有一点形式化, 但是换个角度想一下: 如果方法在它的签名的其他什么地方要求使用具体的 `Self` 类型, 而一个对象又忘记了它具体的类型, 这时方法就无法使用它遗忘的原始的具体类型了。当使用 `trait` 的泛型类型参数被放入具体类型参数时也是如此: 这个具体的类型就成了实现该 `trait` 的类型的一部分。一旦这个类型因使用 `trait` 对象而被擦除掉了之后, 就无法知道放入泛型类型参数的类型是什么了。

一个 `trait` 的方法不是对象安全的例子是标准库中的 `Clone trait`。 `Clone trait` 的 `clone` 方法的参数签名看起来像这样:

```
pub trait Clone {
 fn clone(&self) -> Self;
}
```

`String` 实现了 `Clone trait`, 当在 `String` 实例上调用 `clone` 方法时会得到一个 `String` 实例。类似的, 当调用 `Vec` 实例的 `clone` 方法会得到一个 `Vec` 实例。 `clone` 的签名需要知道什么类型会代替 `Self`, 因为这是它的返回值。

如果尝试在像列表 17-3 中 `Draw` 那样的 `trait` 上实现 `Clone`, 就无法知道 `Self` 将会是 `Button`、`SelectBox` 亦或是将来会实现 `Draw trait` 的其他什么类型。

如果尝试做一些违反有关 `trait` 对象但违反对象安全规则的事情, 编译器会提示你。例如, 如果尝试实现列表 17-4 中的 `Screen` 结构体来存放实现了 `Clone trait` 而不是 `Draw trait` 的类型, 像这样:

```
pub struct Screen {
 pub components: Vec<Box<Clone>>,
}
```

将会得到如下错误:

```
error[E0038]: the trait `std::clone::Clone` cannot be made into an object
-->
|
2 | pub components: Vec<Box<Clone>>,
| ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait `std::clone::Clone` cannot be
made into an object
|
```

= note: the trait cannot require that `Self : Sized`

# 面向对象设计模式的实现

ch17-03-oo-design-patterns.md

commit 67737ff868e3347588cc832eceb8fc237afc5895

让我们看看一个状态设计模式的例子以及如何在 **Rust** 中使用他们。状态模式（*state pattern*）是指一个值有某些内部状态，而它的行为随着其内部状态而改变。内部状态由一系列继承了共享功能的对象表现（我们使用结构体和 **trait** 因为 **Rust** 没有对象和继承）。每一个状态对象负责它自身的行为和当需要改变为另一个状态时的规则。持有任何一个这种状态对象的值对于不同状态的行为以及何时状态转移毫不知情。当将来需求改变时，无需改变值持有状态或者使用值的代码。我们只需更新某个状态对象中的代码来改变它的规则，或者是增加更多的状态对象。

为了探索这个概念，我们将实现一个增量式的发布博文的工作流。这个我们希望发布博文时所应遵守的工作流，一旦完成了它的实现，将为如下：

1. 博文从空白的草案开始。
2. 一旦草案完成，请求审核博文。
3. 一旦博文过审，它将被发表。
4. 只有被发表的博文的内容会被打印，这样就不会意外打印出没有被审核的博文的文本。

任何其他对博文的修改尝试都是没有作用的。例如，如果尝试在请求审核之前通过一个草案博文，博文应该保持未发布的状态。

列表 17-11 展示这个工作流的代码形式。这是一个我们将要在一个叫做 `blog` 的库 `crate` 中实现的 **API** 的使用示例：

文件名: `src/main.rs`

```
extern crate blog;
use blog::Post;

fn main() {
 let mut post = Post::new();

 post.add_text("I ate a salad for lunch today");
 assert_eq!("", post.content());

 post.request_review();
 assert_eq!("", post.content());

 post.approve();
 assert_eq!("I ate a salad for lunch today", post.content());
}
```

列表 17-11: 展示了 `blog` `crate` 期望行为的代码

我们希望能够使用 `Post::new` 创建一个新的博文草案。接着希望能在草案阶段为博文编写一些文本。如果尝试立即打印出博文的内容，将不会得到任何文本，因为博文仍然是草案。这里增加的 `assert_eq!` 用于展示目的。断言草案博文的 `content` 方法返回空字符串将能作为库的一个非常好的单元测试，不过我们并不准备为这个例子编写单元测试。

接下来，我们希望能够请求审核博文，而在等待审核的阶段 `content` 应该仍然返回空字符串，当博文审核通过，它应该被发表，这意味着当调用 `content` 时我们编写的文本将被返回。

注意我们与 `crate` 交互的唯一的类型是 `Post`。博文可能处于的多种状态（草案，等待审核和发布）由 `Post` 内部管理。博文状态依我们在 `Post` 调用的方法而改变，但不必直接管理状态改变。这也意味着不会在状态上犯错，比如忘记了在发布前请求审核。

## 定义 `Post` 并新建一个草案状态的实例

让我们开始实现这个库吧！我们知道需要一个公有 `Post` 结构体来存放一些文本，所以让我们从结构体的定义和一个创建 `Post` 实例的公有关联函数 `new` 开始，如列表 17-12 所示。我们还需定义一个私有 `trait State`。`Post` 将在私有字段 `state` 中存放一个 `Option` 中的 `trait` 对象 `Box<State>`。稍后将会看到为何 `Option` 是必须的。`State trait` 定义了所有不同状态的博文所共享的行为，同时 `Draft`、`PendingReview` 和 `Published` 状态都会实现 `State` 状态。现在这个 `trait` 并没有任何方法，同时开始将只定义 `Draft` 状态因为这是我们希望开始的状态：

文件名: `src/lib.rs`

```
pub struct Post {
 state: Option<Box<State>>,
 content: String,
}

impl Post {
 pub fn new() -> Post {
 Post {
 state: Some(Box::new(Draft {})),
 content: String::new(),
 }
 }
}

trait State {}

struct Draft {}

impl State for Draft {}
```

列表 17-12: `Post` 结构体的定义和新建 `Post` 实例的 `new` 函数，`State trait` 和实现了 `State` 的结构体 `Draft`

当创建新的 `Post` 时，我们将其 `state` 字段设置为一个 `Some` 值，它存放了指向一个 `Draft` 结构体新实例的 `Box`。这确保了无论何时新建一个 `Post` 实例，它会从草案开始。因为 `Post` 的 `state` 字段是私有的，也就无法创建任何其他状态的 `Post` 了！。

## 存放博文内容的文本

在 `Post::new` 函数中，我们设置 `content` 字段为新的空 `String`。在列表 17-11 中，展示了我们希望能够调用一个叫做 `add_text` 的方法并向其传递一个 `&str` 来将文本增加到博文的内容中。选择实现为一个方法而不是将 `content` 字段暴露为 `pub` 是因为我们希望能够通过之后实现的一个方法来控制 `content` 字段如何被读取。`add_text` 方法是非常直观的，让我们在列表 17-13 的 `impl Post` 块中增加一个实现：

文件名: `src/lib.rs`

```
pub struct Post {
content: String,
}
#
impl Post {
 // ...snip...
 pub fn add_text(&mut self, text: &str) {
 self.content.push_str(text);
 }
}
```

```

 }
}

```

列表 17-13: 实现方法 `add_text` 来向博客的 `content` 增加文本

`add_text` 获取一个 `self` 的可变引用, 因为需要改变调用 `add_text` 的 `Post`。接着调用 `content` 中的 `String` 的 `push_str` 并传递 `text` 参数来保存到 `content` 中。这不是状态模式的一部分, 因为它的行为并不依赖博文所处的状态。`add_text` 方法完全不与 `state` 状态交互, 不过这是我们希望支持的行为的一部分。

## 博文草案的内容是空的

调用 `add_text` 并像博文增加一些内容之后, 我们仍然希望 `content` 方法返回一个空字符串 `slice`, 因为博文仍然处于草案状态, 如列表 17-11 的第 8 行所示。现在让我们使用能满足要求的最简单的方式来实现 `content` 方法总是返回一个空字符串 `slice`。当实现了将博文状态改为发布的能力之后将改变这一做法。但是现在博文只能是草案状态, 这意味着其内容总是空的。列表 17-14 展示了这个占位符实现:

文件名: `src/lib.rs`

```

pub struct Post {
content: String,
}
#
impl Post {
 // ...snip...
 pub fn content(&self) -> &str {
 ""
 }
}

```

列表 17-14: 增加一个 `Post` 的 `content` 方法的占位实现, 它总是返回一个空字符串 `slice`

通过增加这个 `content` 方法, 列表 17-11 中直到第 8 行的代码能如期运行。

## 请求审核博文来改变其状态

接下来是请求审核博文, 这应当将其状态由 `Draft` 改为 `PendingReview`。我们希望 `post` 有一个获取 `self` 可变引用的公有方法 `request_review`。接着将调用内部存放的状态的 `request_review` 方法, 而这第二个 `request_review` 方法会消费当前的状态并返回要一个状态。为了能够消费旧状态, 第二个 `request_review` 方法需要能够获取状态值的所有权。这就是 `Option` 的作用: 我们将 `take` 字段 `state` 中的 `Some` 值并留下一个 `None` 值, 因为 `Rust` 并不允许结构体中有空字段。接着将博文的 `state` 设置为这个操作的结果。列表 17-15 展示了这些代码:

文件名: `src/lib.rs`

```

pub struct Post {
state: Option<Box<State>>,
content: String,
}
#
impl Post {
 // ...snip...
 pub fn request_review(&mut self) {
 if let Some(s) = self.state.take() {
 self.state = Some(s.request_review())
 }
 }
}

```

```

trait State {
 fn request_review(self: Box<Self>) -> Box<State>;
}

struct Draft {}

impl State for Draft {
 fn request_review(self: Box<Self>) -> Box<State> {
 Box::new(PendingReview {})
 }
}

struct PendingReview {}

impl State for PendingReview {
 fn request_review(self: Box<Self>) -> Box<State> {
 self
 }
}

```

列表 17-15: 实现 `Post` 和 `State trait` 的 `request_review` 方法

这里给 `State trait` 增加了 `request_review` 方法；所有实现了这个 `trait` 的类型现在都需要实现 `request_review` 方法。注意不用于使用 `self`、`&self` 或者 `&mut self` 作为方法的第一个参数，这里使用了 `self: Box<Self>`。这个语法意味着这个方法调用只对这个类型的 `Box` 有效。这个语法获取了 `Box<Self>` 的所有权，这是我们所希望的，因为需要从老状态转换为新状态，同时希望老状态不再有效。

`Draft` 的方法 `request_review` 的实现返回一个新的，装箱的 `PendingReview` 结构体的实例，这是新引入的用来代表博文处于等待审核状态的类型。结构体 `PendingReview` 同样也实现了 `request_review` 方法，不过它不进行任何状态转换。它返回自身，因为请求审核已经处于 `PendingReview` 状态的博文应该保持 `PendingReview` 状态。

现在能够看出状态模式的优势了：`Post` 的 `request_review` 方法无论 `state` 是何值都是一样的。每个状态负责它自己的规则。

我们将继续保持 `Post` 的 `content` 方法不变，返回一个空字符串 `slice`。现在可以拥有 `PendingReview` 状态而不仅仅是 `Draft` 状态的 `Post` 了，不过我们希望在 `PendingReview` 状态下其也有相同的行为。现在列表 17-11 中直到 11 行的代码是可以执行的！

## 批准博文并改变 `content` 的行为

`Post` 的 `approve` 方法将与 `request_review` 方法类似：它会将 `state` 设置为审核通过时应处于的状态。我们需要为 `State trait` 增加 `approve` 方法，并需新增实现了 `State` 的结构体，`Published` 状态。列表 17-16 展示了新增的代码：

文件名: `src/lib.rs`

```

pub struct Post {
state: Option<Box<State>>,
content: String,
}
#
impl Post {
 // ...snip...
 pub fn approve(&mut self) {
 if let Some(s) = self.state.take() {
 self.state = Some(s.approve())
 }
 }
}

```

```

trait State {
 fn request_review(self: Box<Self>) -> Box<State>;
 fn approve(self: Box<Self>) -> Box<State>;
}

struct Draft {}

impl State for Draft {
 # fn request_review(self: Box<Self>) -> Box<State> {
 # Box::new(PendingReview {})
 # }
 #
 // ...snip...
 fn approve(self: Box<Self>) -> Box<State> {
 self
 }
}

struct PendingReview {}

impl State for PendingReview {
 # fn request_review(self: Box<Self>) -> Box<State> {
 # Box::new(PendingReview {})
 # }
 #
 // ...snip...
 fn approve(self: Box<Self>) -> Box<State> {
 Box::new(Published {})
 }
}

struct Published {}

impl State for Published {
 fn request_review(self: Box<Self>) -> Box<State> {
 self
 }

 fn approve(self: Box<Self>) -> Box<State> {
 self
 }
}

```

列表 17-16: 为 `Post` 和 `State trait` 实现 `approve` 方法

类似于 `request_review`，如果对 `Draft` 调用 `approve` 方法，并没有任何效果，因为它会返回 `self`。当对 `PendingReview` 调用 `approve` 时，它返回一个新的、装箱的 `Published` 结构体的实例。`Published` 结构体实现了 `State trait`，同时对于 `request_review` 和 `approve` 方法来说，它返回自身，因为在这两种情况博文应该保持 `Published` 状态。

现在更新 `Post` 的 `content` 方法：我们希望当博文处于 `Published` 时返回 `content` 字段的值，否则返回空字符串 `slice`。因为目标是将所有像这样的规则保持在实现了 `State` 的结构体中，我们将调用 `state` 中的值的 `content` 方法并传递博文实例（也就是 `self`）作为参数。接着返回 `state` 值的 `content` 方法的返回值，如列表 17-17 所示：

文件名: `src/lib.rs`

```

trait State {
fn content<'a>(&self, post: &'a Post) -> &'a str;
}
pub struct Post {

```



```
state: Option<Box<State>>,
content: String,
}
#
impl Post {
 // ...snip...
 pub fn content(&self) -> &str {
 self.state.as_ref().unwrap().content(&self)
 }
 // ...snip...
}
```

列表 17-17: 更新 `Post` 的 `content` 方法来委托调用 `State` 的 `content` 方法

这里调用 `Option` 的 `as_ref` 方法是因为需要 `Option` 中值的引用。接着调用 `unwrap` 方法，这里我们知道永远也不会 `panic` 因为 `Post` 的所有方法都确保在他们返回时 `state` 会有一个 `Some` 值。这就是一个第十二章讨论过的我们知道 `None` 是不可能的而编译器却不能理解的情况。

`State` `trait` 的 `content` 方法是博文返回什么内容的逻辑所在之处。我们将增加一个 `content` 方法的默认实现来返回一个空字符串 `slice`。这样就无需为 `Draft` 和 `PendingReview` 结构体实现 `content` 了。`Published` 结构体会覆盖 `content` 方法并会返回 `post.content` 的值，如列表 17-18 所示：

文件名: `src/lib.rs`

```
pub struct Post {
content: String
}
trait State {
 // ...snip...
 fn content<'a>(&self, post: &'a Post) -> &'a str {
 ""
 }
}

// ...snip...
struct Published {}

impl State for Published {
 // ...snip...
 fn content<'a>(&self, post: &'a Post) -> &'a str {
 &post.content
 }
}
```

列表 17-18: 为 `State` `trait` 增加 `content` 方法

注意这个方法需要生命周期注解，如第十章所讨论的。这里获取 `post` 的引用作为参数，并返回 `post` 一部分的引用，所以返回的引用的生命周期与 `post` 参数相关。

## 状态模式的权衡取舍

我们展示了 `Rust` 是能够实现面向对象的状态模式的，以便能根据博文所处的状态来封装不同类型的行为。`Post` 的方法并不知道这些不同类型的行为。这种组织代码的方式，为了找到所有已发布的博文不同行为只需查看一处代码：`Published` 的 `State` `trait` 的实现。

一个不使用状态模式的替代实现可能会在 `Post` 的方法中，甚至于在使用 `Post` 的代码中（在这里是 `main` 中）用到 `match` 语句，来检查博文状态并在这里改变其行为。这可能意味着需要查看很多位置来理解处于发布状态的博文的所有逻辑！这在增加更多状态时会变得更糟：每一个 `match` 语句都会需要另一个分支。对于状态模式来

说，`Post` 的方法和使用 `Post` 的位置无需 `match` 语句，同时增加新状态只涉及到增加一个新 `struct` 和为其实现 `trait` 的方法。

这个实现易于增加更多功能。这里是一些你可以尝试对本部分代码做出的修改，来亲身体会一下使用状态模式随着时间的推移维护代码是什么感觉：

- 只允许博文处于 `Draft` 状态时增加文本内容
- 增加 `reject` 方法将博文的状态从 `PendingReview` 变回 `Draft`
- 在将状态变为 `Published` 之前需要两次 `approve` 调用

状态模式的一个缺点是因为状态实现了状态之间的转换，一些状态会相互联系。如果在 `PendingReview` 和 `Published` 之间增加另一个状态，比如 `Scheduled`，则不得不修改 `PendingReview` 中的代码来转移到 `Scheduled`。如果 `PendingReview` 无需因为新增的状态而改变就更好了，不过这意味着切换到另一个设计模式。

这个 `Rust` 中的实现的缺点在于存在一些重复的逻辑。如果能够为 `State trait` 中返回 `self` 的 `request_review` 和 `approve` 方法增加默认实现就好了，不过这会违反对象安全性，因为 `trait` 不知道 `self` 具体是什么。我们希望能够将 `State` 作为一个 `trait` 对象，所以需要这个方法是对象安全的。

另一个最好能去除的重复是 `Post` 中 `request_review` 和 `approve` 这两个类似的实现。他们都委托调用了 `state` 字段中 `Option` 值的同一方法，并在结果中为 `state` 字段设置了新值。如果 `Post` 中的很多方法都遵循这个模式，我们可能会考虑定义一个宏来消除重复（查看附录 E 以了解宏）。

这个完全按照面向对象语言的定义实现的面向对象模式的缺点在于没有尽可能的利用 `Rust` 的优势。让我们看看一些代码中可以做出的修改，来将无效的状态和状态转移变为编译时错误。

## 将状态和行为编码为类型

我们将展示如何稍微反思状态模式来进行一系列不同的权衡取舍。不同于完全封装状态和状态转移使得外部代码对其毫不知情，我们将状态编码进不同的类型。当状态是类型时，`Rust` 的类型检查就会使任何在只能使用发布的博文的地方使用草案博文的尝试变为编译时错误。

让我们考虑一下列表 17-11 中 `main` 的第一部分：

文件名: `src/main.rs`

```
fn main() {
 let mut post = Post::new();

 post.add_text("I ate a salad for lunch today");
 assert_eq!("", post.content());
}
```

我们仍然希望使用 `Post::new` 创建一个新的草案博文，并仍然希望能够增加博文的内容。不过不同于存在一个草案博文时返回空字符串的 `content` 方法，我们将使草案博文完全没有 `content` 方法。这样如果尝试获取草案博文的内容，将会得到一个方法不存在的编译错误。这使得我们不可能在生产环境意外显示出草案博文的内容，因为这样的代码甚至就不能编译。列表 17-19 展示了 `Post` 结构体、`DraftPost` 结构体以及各自的方法的定义：

文件名: `src/lib.rs`

```
pub struct Post {
 content: String,
}

pub struct DraftPost {
 content: String,
}
```

```
impl Post {
 pub fn new() -> DraftPost {
 DraftPost {
 content: String::new(),
 }
 }

 pub fn content(&self) -> &str {
 &self.content
 }
}

impl DraftPost {
 pub fn add_text(&mut self, text: &str) {
 self.content.push_str(text);
 }
}
```

列表 17-19: 带有 `content` 方法的 `Post` 和没有 `content` 方法的 `DraftPost`

`Post` 和 `DraftPost` 结构体都有一个私有的 `content` 字段来储存博文的文本。这些结构体不再有 `state` 字段因为我们将类型编码为结构体的类型。`Post` 将代表发布的博文，它有一个返回 `content` 的 `content` 方法。

仍然有一个 `Post::new` 函数，不过不同于返回 `Post` 实例，它返回 `DraftPost` 的实例。现在不可能创建一个 `Post` 实例，因为 `content` 是私有的同时没有任何函数返回 `Post`。`DraftPost` 上定义了一个 `add_text` 方法，这样就可以像之前那样向 `content` 增加文本，不过注意 `DraftPost` 并没有定义 `content` 方法！所以所有博文都强制从草案开始，同时草案博文没有任何可供展示的内容。任何绕过这些限制的尝试都会产生编译错误。

## 实现状态转移为不同类型的转移

那么如何得到发布的博文呢？我们希望强制的规则是草案博文在可以发布之前必须被审核通过。等待审核状态的博文应该仍然不会显示任何内容。让我们通过增加另一个结构体 `PendingReviewPost` 来实现这个限制，在 `DraftPost` 上定义 `request_review` 方法来返回 `PendingReviewPost`，并在 `PendingReviewPost` 上定义 `approve` 方法来返回 `Post`，如列表 17-20 所示：

文件名: `src/lib.rs`

```
pub struct Post {
content: String,
}
#
pub struct DraftPost {
content: String,
}
#
impl DraftPost {
 // ...snip...

 pub fn request_review(self) -> PendingReviewPost {
 PendingReviewPost {
 content: self.content,
 }
 }
}

pub struct PendingReviewPost {
 content: String,
}

impl PendingReviewPost {
```

```
pub fn approve(self) -> Post {
 Post {
 content: self.content,
 }
}
```

列表 17-20: `PendingReviewPost` 通过调用 `DraftPost` 的 `request_review` 创建, `approve` 方法将 `PendingReviewPost` 变为发布的 `Post`

`request_review` 和 `approve` 方法获取 `self` 的所有权, 因此会消费 `DraftPost` 和 `PendingReviewPost` 实例, 并分别转换为 `PendingReviewPost` 和 发布的 `Post`。这样在调用 `request_review` 之后就不会遗留任何 `DraftPost` 实例, 后者同理。`PendingReviewPost` 并没有定义 `content` 方法, 所以类似 `DraftPost` 尝试读取它的内容是一个编译错误。因为唯一得到定义了 `content` 方法的 `Post` 实例的途径是调用 `PendingReviewPost` 的 `approve` 方法, 而得到 `PendingReviewPost` 的唯一办法是调用 `DraftPost` 的 `request_review` 方法, 现在我们就将发博文的工作流编码进了类型系统。

这也意味着不得不为 `main` 做出一些小的修改。因为 `request_review` 和 `approve` 返回新实例而不是修改被调用的结构体, 我们需要增加更多的 `let post =` 覆盖赋值来保存返回的实例。也不能再断言草稿和等待审核的博文的内容为空字符串了, 我们也不再需要他们: 不能编译尝试使用这些状态下博文内容的代码。更新后的 `main` 的代码如列表 18-21 所示:

Filename: `src/main.rs`

```
extern crate blog;
use blog::Post;

fn main() {
 let mut post = Post::new();

 post.add_text("I ate a salad for lunch today");

 let post = post.request_review();

 let post = post.approve();

 assert_eq!("I ate a salad for lunch today", post.content());
}
```

列表 17-21: `main` 中使用新的博文工作流实现的修改

不得不修改 `main` 来重新赋值 `post` 使得这个实现不再完全遵守面向对象的状态模式: 状态间的转换不再完全封装在 `Post` 实现中。然而, 得益于类型系统和编译时类型检查我们得到了不可能拥有无效状态的属性! 这确保了特定的 `bug`, 比如显示未发布博文的内容, 将在部署到生产环境之前被发现。

尝试在这一部分开始所建议的增加额外需求的任务来体会使用这个版本的代码是何感觉。

即便 `Rust` 能够实现面向对象设计模式, 也有其他像将状态编码进类型这样的模式存在。这些模式有着不同于面向对象模式的权衡取舍。虽然你可能非常熟悉面向对象模式, 重新思考这些问题来利用 `Rust` 提供的像在编译时避免一些 `bug` 这样有益功能。在 `Rust` 中面向对象模式并不总是最好的解决方案, 因为 `Rust` 拥有像所有权这样的面向对象语言所没有的功能。

## 总结

阅读本章后，不管你是否认为 **Rust** 是一个面向对象语言，现在你都见识了 **trait** 对象是一个 **Rust** 中获取部分面向对象功能的方法。动态分发可以通过牺牲一些运行时性能来为你的代码提供一些灵活性。这些灵活性可以用来实现有助于代码可维护性的面向对象模式。**Rust** 也有像所有权这样不同于面向对象语言的功能。面向对象模式并不总是利用 **Rust** 实力的最好方式。

接下来，让我们看看另一个提供了多样灵活性的**Rust**功能：模式。贯穿全书的模式, 我们已经和它们打过照面了，但并没有见识过它们的全部本领。让我们开始探索吧！

# 模式用来匹配值的结构

[ch18-00-patterns.md](#)

commit 3d47ebddad51b0080a19857e1495675a8e9376ef

模式是 **Rust** 中特殊的语法，它用来匹配类型中的结构，无论类型是简单还是复杂。模式由一些常量组成：解构数组、枚举、结构体或者是元组；变量、通配符和占位符。这些部分描述了我们要处理的数据的“形状”。

我们通过将一些值与模式相比较来使用它。如果模式匹配这些值，我们对值部分进行相应处理。回忆一下第六章讨论 `match` 表达式时像硬币分类器那样使用模式。我们可以为形状中的片段命名，就像在第六章中命名出现在二十五美分硬币上的州那样，如果数据符合这个形状，就可以使用这些命名的片段。

本章是所有模式相关内容的参考。我们将涉及到使用模式的有效位置，*refutable* 与 *irrefutable* 模式的区别，和你可能会见到的不同类型的模式语法。

# 所有可能会用到模式的位置

[ch18-01-all-the-places-for-patterns.md](#)

commit 4ca9e513e532a4d229ab5af7dfcc567129623bf4

模式出现在 **Rust** 的很多地方。你已经在不经意间使用了很多模式！本部分是一个所有有效模式位置的参考。

## match 分支

如第六章所讨论的，一个模式常用的位置是 `match` 表达式的分支。在形式上 `match` 表达式由 `match` 关键字、用于匹配的值和一个或多个分支构成。这些分支包含一个模式和在值匹配分支的模式时运行的表达式：

```
match VALUE {
 PATTERN => EXPRESSION,
 PATTERN => EXPRESSION,
 PATTERN => EXPRESSION,
}
```

## 穷尽性和默认模式 `_`

`match` 表达式必须是穷尽的。当我们把所有分支的模式都放在一起，`match` 表达式所有可能的值都应该被考虑到。一个确保覆盖每个可能值的方法是在最后一个分支使用捕获所有的模式，比如一个变量名。一个匹配任何值的名称永远也不会失败，因此可以覆盖之前分支模式匹配剩下的情况。

这有一个额外的模式经常被用于结尾的分支：`_`。它匹配所有情况，不过它从不绑定任何变量。这在例如只希望在某些模式下运行代码而忽略其他值的时候很有用。

## if let 表达式

第六章讨论过了 `if let` 表达式，以及它是如何成为编写等同于只关心一个情况的 `match` 语句的简写的。`if let` 可以对应一个可选的 `else` 和代码在 `if let` 中的模式不匹配时运行。

列表 18-1 展示了甚至可以组合并匹配 `if let`、`else if` 和 `else if let`。这些代码展示了一系列针对不同条件的检查来决定背景颜色应该是什么。为了达到这个例子的目的，我们创建了硬编码值的变量，在真实程序中则可能由询问用户获得。如果用户指定了中意的颜色，我们将使用它作为背景颜色。如果今天是星期二，背景颜色将是绿色。如果用户指定了他们的年龄字符串并能够成功将其解析为数字的话，我们将根据这个数字使用紫色或者橙色。最后，如果没有一个条件符合，背景颜色将是蓝色：

文件名: `src/main.rs`

```
fn main() {
 let favorite_color: Option<&str> = None;
 let is_tuesday = false;
 let age: Result<u8, _> = "34".parse();

 if let Some(color) = favorite_color {
 println!("Using your favorite color, {}, as the background", color);
 } else if is_tuesday {
 println!("Tuesday is green day!");
 } else if let Ok(age) = age {
 if age > 30 {
 println!("Using purple as the background color");
 } else {
 println!("Using orange as the background color");
 }
 }
}
```

```

 }
} else {
 println!("Using blue as the background color");
}
}

```

列表 18-1: 结合 `if let`、`else if`、`else if let` 和 `else`

这个条件结构允许我们支持复杂的需求。使用这里硬编码的值，例子会打印出 `Using purple as the background color`。

注意 `if let` 也可以像 `match` 分支那样引入覆盖变量：`if let Ok(age) = age` 引入了一个新的覆盖变量 `age`，它包含 `Ok` 成员中的值。这也意味着 `if age > 30` 条件需要位于这个代码块内部；不能将两个条件组合为 `if let Ok(age) = age && age > 30`，因为我们希望与 `30` 进行比较的被覆盖的 `age` 直到大括号开始的新作用域才是有效的。

另外注意这样有很多情况的条件并没有 `match` 表达式强大，因为其穷尽性没有为编译器所检查。如果去掉最后的 `else` 块而遗漏处理一些情况，编译器也不会报错。这个例子可能过于复杂以致难以重写为一个可读的 `match`，所以需要额外注意处理了所有的情况，因为编译器不会为我们检查穷尽性。

## while let

一个与 `if let` 类似的结构体是 `while let`：它允许只要模式匹配就一直进行 `while` 循环。列表 18-2 展示了一个使用 `while let` 的例子，它使用 `vector` 作为栈并打以先进后出的方式打印出 `vector` 中的值：

```

let mut stack = Vec::new();

stack.push(1);
stack.push(2);
stack.push(3);

while let Some(top) = stack.pop() {
 println!("{}", top);
}

```

列表 18-2: 使用 `while let` 循环只要 `stack.pop()` 返回 `Some` 就打印出其值

这个例子会打印出 `3`、`2` 和 `1`。`pop` 方法取出 `vector` 的最后一个元素并返回 `Some(value)`，如果 `vector` 是空的，它返回 `None`。`while` 循环只要 `pop` 返回 `Some` 就会一直运行其块中的代码。一旦其返回 `None`，`while` 循环停止。我们可以使用 `while let` 来弹出栈中的每一个元素。

## for 循环

`for` 循环，如同第三章所讲的，是 `Rust` 中最常见的循环结构。那一章所没有讲到的是 `for` 可以获取一个模式。列表 18-3 中展示了如何使用 `for` 循环来解构一个元组。`enumerate` 方法适配一个迭代器来产生元组，其包含值和值的索引：

```

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

for (index, value) in v.iter().enumerate() {
 println!("{}", value, index);
}

```

列表 18-3: 在 `for` 循环中使用模式来解构 `enumerate` 返回的元组

这会打印出：



```
1 is at index 0
2 is at index 1
3 is at index 2
```

第一个 `enumerate` 调用会产生元组 `(0, 1)`。当这个匹配模式 `(index, value)`，`index` 将会是 0 而 `value` 将会是 1。

## let 语句

`match` 和 `if let` 都是本书之前明确讨论过的使用模式的位置，不过他们不是仅有的使用过模式的地方。例如，考虑一下这个直白的 `let` 变量赋值：

```
let x = 5;
```

本书进行了不下百次这样的操作。你可能没有发觉，不过你这正是在使用模式！`let` 语句更为正式的样子如下：

```
let PATTERN = EXPRESSION;
```

我们见过的像 `let x = 5;` 这样的语句中变量名位于 `PATTERN` 位置；变量名不过是形式特别朴素的模式。

通过 `let`，我们将表达式与模式比较，并为任何找到的名称赋值。所以例如 `let x = 5;` 的情况，`x` 是一个模式代表“将匹配到的值绑定到变量 `x`”。同时因为名称 `x` 是整个模式，这个模式实际上等于“将任何值绑定到变量 `x`，不过它是什么”。

为了更清楚的理解 `let` 的模式匹配的方面，考虑列表 18-4 中使用 `let` 和模式解构一个元组：

```
let (x, y, z) = (1, 2, 3);
```

列表 18-4: 使用模式解构元组并一次创建三个变量

这里有一个元组与模式匹配。`Rust` 会比较值 `(1, 2, 3)` 与模式 `(x, y, z)` 并发现值匹配这个模式。在这个例子中，将会把 `1` 绑定到 `x`，`2` 绑定到 `y`，`3` 绑定到 `z`。你可以将这个元组模式看作是将三个独立的变量模式结合在一起。

在第十六章中我们见过另一个解构元组的例子，列表 16-6 中，那里解构 `mpsc::channel()` 的返回值为 `tx`（发送者）和 `rx`（接收者）。

## 函数参数

类似于 `let`，函数参数也可以是模式。列表 18-5 中的代码声明了一个叫做 `foo` 的函数，它获取一个 `i32` 类型的参数 `x`，这看起来应该很熟悉：

```
fn foo(x: i32) {
 // code goes here
}
```

列表 18-5: 在参数中使用模式的函数签名

`x` 部分就是一个模式！类似于之前对 `let` 所做的，可以在函数参数中匹配元组。列表 18-6 展示了如何可以将传递给函数的元组拆分为值：

文件名: `src/main.rs`

```
fn print_coordinates(&(x, y): &(i32, i32)) {
 println!("Current location: ({}, {})", x, y);
}

fn main() {
 let point = (3, 5);
 print_coordinates(&point);
}
```

列表 18-6: 一个在参数中解构元组的函数

这会打印出 `Current location: (3, 5)`。当传递值 `&(3, 5)` 给 `print_coordinates` 时，这个值会匹配模式 `&(x, y)`，`x` 得到了值 3，而 `y` 得到了值 5。

因为如第十三章所讲闭包类似于函数，也可以在闭包参数中使用模式。

在这些可以使用模式的位置中的一个区别是，对于 `for` 循环、`let` 和函数参数，其模式必须是 *irrefutable* 的。接下来让我们讨论这个。

## Refutability(可反驳性): 模式是否会匹配失效

匹配模式有两种形式: **refutable**(可反驳)和**irrefutable**(不可反驳). 对任意可能的值进行匹配都不会失效的模式被称为是**irrefutable**(不可反驳)的, 而对某些可能的值进行匹配会失效的模式被称为是**refutable**(可反驳)的. `let` 语句、函数参数和 `for` 循环被约束为只接受**irrefutable**模式, 因为如果模式匹配失效程序就不会正确运行. `if let` 和 `while let` 表达式被约束为只接受**refutable**模式, 因为它们需要处理可能存在的匹配失效的情况, 并且如果模式匹配永不失效, 那它们就派不上用场了.

通常, 你不用关心**refutable**和**irrefutable**模式的区别, 当你看见它出现在了错误消息中时, 你只要了解可反驳性(**refutability**)的概念即可. 如果你得到一个涉及到可反驳性概念的错误消息, 根据你的代码行为的意图, 你只需改变匹配模式或者是改变你构造模式的方法即可.

让我们来看几个例子. 在本章的前面部分, 我们提到 `let x = 5;` 这里 `x` 就是一个我们被允许使用**irrefutable**的模式: 因为它不可能匹配失效. 相反, 如果用 `let` 来匹配一个枚举的变体, 比如像例18-7中列出的那样从 `Option<T>` 枚举中只匹配 `Some<T>` 这个值:

```
let Some(x) = some_option_value;
```

例18-7: 试试用一个有 `let` 的**refutable**模式

如果 `some_option_value` 的值是 `None`, `some_option_value` 将不会匹配模式 `Some(x)`. 模式 `Some(x)` 是可反驳的(**refutable**), 因为存在一个使它匹配失效的值. 如果 `some_option_value` 的值是 `None`, 那么 `let` 语句就不会产生任何效果. 因此Rust会在编译时会报期望**irrefutable**模式但是却得到了一个**refutable**模式的错误:

```
error[E0005]: refutable pattern in local binding: `None` not covered
--> <anon>:3:5
 |
 3 | let Some(x) = some_option_value;
 | ^^^^^^^ pattern `None` not covered
```

因为我们没有(也不能)覆盖到模式 `Some(x)` 的每一个可能的值, 所以Rust会报错.

如果我们采用**refutable**模式, 使用 `if let` 而不是 `let`. 这样当模式不匹配时, 在花括号中的代码将不执行, 这段代码只有在值匹配模式的时候才会执行, 也仅在此时才有意义. 例18-8显示了如何修正在例18-7中用 `Some(x)` 来匹配 `some_option_value` 的代码. 因为这个例子使用了 `if let`, 因此使用**refutable**模式的 `Some(x)` 就没问题了:

```
let some_option_value: Option<i32> = None;
if let Some(x) = some_option_value {
 println!("{}", x);
}
```

例18-8: 使用 `if let` 和一个有**refutable**模式的代码块来代替 `let`

此外, 如果我们给 `if let` 一个绝对会匹配的**irrefutable**模式, 比如在例18-9中显示的 `x`:

```
if let x = 5 {
 println!("{}", x);
};
```

例18-9: 尝试把一个**irrefutable**模式用到 `if let` 上

Rust将会抱怨把 `if let` 和一个**irrefutable**模式一起使用没有意义:

```
error[E0162]: irrefutable if-let pattern
--> <anon>:2:8
 |
2 | if let x = 5 {
 | ^ irrefutable pattern
```

一般来说, 多数匹配使用`refutable`模式, 除非是那种可以匹配任意值的情况使用`irrefutable`模式. `match` 操作符中如果只有一个`irrefutable`模式分支也没有什么问题, 但这就没什么特别的用处, 此时可以用一个更简单的 `let` 语句来替换. 不管是把表达式关联到 `let` 语句亦或是关联到只有一个`irrefutable`模式分支的 `match` 操作, 代码都肯定会运行, 如果它们的表达式一样的话最终的结果也相同.

目前我们已经讨论了所有可以使用模式的地方, 也介绍了`refutable`模式和`irrefutable`模式的不同, 下面让我们一起去把可以用来创建模式的语法过目一遍吧.

# 所有的模式语法

通过本书我们已领略过一些不同类型模式的例子. 本节会列出所有在模式中有效的语法并且会阐述你为什么可能会用到它们中的每一个.

## 字面量

我们在第6章已经见过, 你可以直接匹配字面量:

```
let x = 1;

match x {
 1 => println!("one"),
 2 => println!("two"),
 3 => println!("three"),
 _ => println!("anything"),
}
```

这段代码会打印 `one` 因为 `x` 的值是 `1`.

## 命名变量

命名变量是可匹配任何值的 `irrefutable` (不可反驳)模式.

与所有变量一样, 模式中声明的变量会屏蔽 `match` 表达式外层的同名变量, 因为一个 `match` 表达式会开启一个新的作用域. 在列表18-10中, 我们声明了一个值为 `Some(5)` 的变量 `x` 和一个值为 `10` 的变量 `y`. 然后是一个值 `x` 上的 `match` 表达式. 看一看匹配分支的模式和结尾的 `println!`, 你可以在继续阅读或运行代码前猜一猜什么会被打印出来:

Filename: src/main.rs

```
fn main() {
 let x = Some(5);
 let y = 10;

 match x {
 Some(50) => println!("Got 50"),
 Some(y) => println!("Matched, y = {:?}", y),
 _ => println!("Default case, x = {:?}", x),
 }

 println!("at the end: x = {:?}", y = {:?}", x, y);
}
```

列表18-10: 引入了一个阴影变量 `y` 的 `match` 语句

让我们看看当 `match` 语句运行的时候发生了什么. 第一个匹配分支是模式 `Some(50)`, `x` 中的值(`Some(5)`)不匹配 `Some(50)`, 所以我们继续. 在第二个匹配分支中, 模式 `Some(y)` 引入了一个可以匹配在 `Some` 里的任意值的新变量 `y`. 因为我们位于 `match` 表达式里面的新作用域中, 所以 `y` 就是一个新变量而不是在开头被声明的其值为`10`的变量 `y`. 这个新的 `y` 绑定将会匹配在 `Some` 中的任意值, 这里也就是 `x` 中的值, 因为 `y` 绑定到 `Some` 中的值是 `x`, 这里是`5`, 所以我们就执行了这个分支中的表达式并打印出 `Matched, y = 5`.

如果 `x` 的值是 `None` 而不是 `Some(5)`，我们将会匹配下划线因为其它两个分支的模式将不会被匹配。在这个匹配分支(下划线)的表达式里，因为我们没有在分支的模式中引入变量 `x`，所以这个 `x` 仍然是 `match` 作用域外部的那个没被屏蔽的 `x`。在这个假想的例子中，`match` 表达式将会打印出 `Default case, x = None`。

一旦 `match` 表达式执行完毕，它的作用域也就结束了，同时 `match` 内部的 `y` 也就结束了。最后的 `println!` 会打印 `at the end: x = Some(5), y = 10`。

为了让 `match` 表达式能比较外部变量 `x` 和 `y` 的值而不是内部引入的阴影变量 `x` 和 `y`，我们需要使用一个有条件的匹配守卫(`guard`)。我们将在本节的后面讨论匹配守卫。

## 多种模式

只有在 `match` 表达式中，你可以通过 `|` 符号匹配多个模式，它代表或(*or*)的意思：

```
let x = 1;

match x {
 1 | 2 => println!("one or two"),
 3 => println!("three"),
 _ => println!("anything"),
}
```

上面的代码会打印 `one or two`。

## 通过 `...` 匹配值的范围

你可以用 `...` 匹配一个值包含的范围：

```
let x = 5;

match x {
 1 ... 5 => println!("one through five"),
 _ => println!("something else"),
}
```

上面的代码中，如果 `x` 是1、2、3、4或5，第一个分支就会匹配。

范围只能是数字或 `char` 类型的值。下面是一个使用 `char` 类型值范围的例子：

```
let x = 'c';

match x {
 'a' ... 'j' => println!("early ASCII letter"),
 'k' ... 'z' => println!("late ASCII letter"),
 _ => println!("something else"),
}
```

上面的代码会打印 `early ASCII letter`。

## 解构并提取值

模式可以用来解构(*destructure*)结构、枚举、元组和引用。解构意味着把一个值分解成它的组成部分。例18-11中的结构 `Point` 有两个字段 `x` 和 `y`，我们可以通过一个模式和 `let` 语句来进行提取：

Filename: src/main.rs

```

struct Point {
 x: i32,
 y: i32,
}

fn main() {
 let p = Point { x: 0, y: 7 };

 let Point { x, y } = p;
 assert_eq!(0, x);
 assert_eq!(7, y);
}

```

#### 例18-11: 用结构的字段来解构

上面的代码创建了匹配 `p` 中的 `x` 和 `y` 字段的变量 `x` 和 `y`。变量的名字必须匹配使用了这个写法中的字段。如果我们想使用不同的变量名字, 我们可以在模式中使用 `field_name: variable_name`。在例18-12中, `a` 会拥有 `Point` 实例的 `x` 字段的值, `b` 会拥有 `y` 字段的值:

Filename: src/main.rs

```

struct Point {
 x: i32,
 y: i32,
}

fn main() {
 let p = Point { x: 0, y: 7 };

 let Point { x: a, y: b } = p;
 assert_eq!(0, a);
 assert_eq!(7, b);
}

```

#### 例18-12: 把结构解构到与字段不同名的变量中

为了测试和使用一个值内部的某个属性, 我们也可以用字面量来解构。例18-13用一个 `match` 语句来判断一个点是位于 `x` (此时 `y = 0`)轴上还是在 `y` (此时 `x = 0`)轴上或者不在两个轴上面:

```

struct Point {
x: i32,
y: i32,
}
#
fn main() {
let p = Point { x: 0, y: 7 };
#
match p {
Point { x, y: 0 } => println!("On the x axis at {}", x),
Point { x: 0, y } => println!("On the y axis at {}", y),
Point { x, y } => println!("On neither axis: ({}, {})", x, y),
}
}

```

#### 例18-13: 解构和匹配一个模式中的字面量

上面的代码会打印 `On the y axis at 7`, 因为 `p` 的 `x` 字段的值是0, 这正好匹配第二个分支。

在第6章中我们对枚举进行了解构, 比如例6-5中, 我们用一个 `match` 表达式来解构一个 `Option<i32>`, 其中被提取出来的一个值是 `Some` 内的变量。

当我们正匹配的值在一个包含了引用的模式里面时, 为了把引用和值分割开我们可以在模式中指定一个 `&` 符号. 在迭代器对值的引用进行迭代时当我们想在闭包中使用值而不是引用的时候这个符号在闭包里特别有用. 例18-14演示了如何在一个向量里迭代 `Point` 实例的引用, 为了能方便地对 `x` 和 `y` 的值进行计算还对引用的结构进行了解构:

```
struct Point {
x: i32,
y: i32,
}
#
let points = vec![
 Point { x: 0, y: 0 },
 Point { x: 1, y: 5 },
 Point { x: 10, y: -3 },
];
let sum_of_squares: i32 = points
 .iter()
 .map(|&Point {x, y}| x * x + y * y)
 .sum();
```

例18-14: 把结构的引用解构到结构的字段值中

因为 `iter` 会对向量里面的项目的引用进行迭代, 如果我们在 `map` 里的闭包的参数上忘了 `&` 符号, 我们将会得到下面的类型不匹配的错误:

```
error[E0308]: mismatched types
-->
|
14 | .map(|Point {x, y}| x * x + y * y)
| ^^^^^^^^^^^^^^^ expected &Point, found struct `Point`
|
= note: expected type `&Point`
 found type `Point`
```

这个报错提示 **Rust** 希望我们的闭包匹配参数匹配 `&Point`, 但是我们却试图用一个 `Point` 的值的模式去匹配它, 而不是一个 `Point` 的引用.

我们可以用更复杂的方法来合成、匹配和嵌套解构模式: 下例中我们通过在一个元组中嵌套结构和元组来解构出所有的基础类型的值:

```
struct Point {
x: i32,
y: i32,
}
#
let ((feet, inches), Point {x, y}) = ((3, 10), Point { x: 3, y: -10 });
```

这使得我们把复杂的类型提取成了它们的组成成分.

## 忽略模式中的值

有一些简单的方法可以忽略模式中全部或部分值: 使用 `_` 模式, 在另一个模式中使用 `_` 模式, 使用一个以下划线开始的名字, 或者使用 `..` 来忽略掉所有剩下的值. 下面让我们来探索如何以及为什么要这么做.

### 用 `_` 忽略整个值



我们已经见过了用下划线作为通配符会匹配任意值,但是它不会绑定值.把下划线模式用作 `match` 表达式的最后一个匹配分支特别有用,我们可以在任意模式中使用它,比如在例18-15中显示的函数参数:

```
fn foo(_: i32) {
 // code goes here
}
```

例18-15: 在一个函数签名中使用 `_`

通常,你应该把这种函数的参数声明改成不用无用参数.如果是要实现这样一个有特定类型签名的 *trait*,使用下划线可以让你忽略一个参数,并且编译器不会像使用命名参数那样警告有未使用的函数参数.

## 用一个嵌套的 `_` 忽略部分值

我们也可以在另一个模式中使用 `_` 来忽略部分值.在例18-16中,第一个 `match` 分支中的模式匹配了一个 `Some` 值,但是却通过下划线忽略掉了 `Some` 变量中的值:

```
let x = Some(5);

match x {
 Some(_) => println!("got a Some and I don't care what's inside"),
 None => (),
}
```

例18-16: 通过使用一个嵌套的下划线忽略 `Some` 变量中的值

当代码关联的 `match` 分支不需要使用被嵌套的全部变量时这很有用.

我们可以在一个模式中多处使用下划线,在例18-17中我们将忽略掉一个五元元组中的第二和第四个数:

```
let numbers = (2, 4, 8, 16, 32);

match numbers {
 (first, _, third, _, fifth) => {
 println!("Some numbers: {}, {}, {}", first, third, fifth)
 },
}
```

例18-17: 忽略元组中的多个部分

上面的代码将会打印出 `Some numbers: 2, 8, 32`,元组中的4和16会被忽略.

## 通过在名字前以一个下划线开头来忽略不使用的变量

如果你创建了一个变量却不使用它, `Rust`通常会给你一个警告,因为这可能会是个bug.如果你正在做原型或者刚开启一个项目,那么你可能会创建一个暂时不用但是以后会使用的变量.如果你面临这个情况并且希望 `Rust`不要对你警告未使用的变量,你可以让那个变量以一个下划线开头.这和其它模式中的变量名没什么区别,只是 `Rust`不会警告你这个变量没用被使用.在例18-18中,我们会得到一个没用使用变量 `y` 的警告,但是我们不会得到没用使用变量 `_x` 的警告:

```
fn main() {
 let _x = 5;
 let y = 10;
}
```

例18-18: 为了消除对未被使用变量的警告以一个下划线开始来命名变量

注意, 只使用 `_` 和使用一个以一个下划线起头的名字是有微妙的不同的: `_x` 仍然会把值绑定到变量上但是 `_` 不会绑定值.

例18-19显示了这种区别的主要地方: `s` 将仍然被转移到 `_s`, 它会阻止我们继续使用 `s`:

```
let s = Some(String::from("Hello!"));

if let Some(_s) = s {
 println!("found a string");
}

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

例18-19: 以下划线起头的未被使用的变量仍然会绑定值, 它也会拥有值的所有权

只使用下划线本身却不会绑定值. 例18-20在编译时将不会报错, 因为 `s` 不会被转移到 `_`:

```
let s = Some(String::from("Hello!"));

if let Some(_) = s {
 println!("found a string");
}

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

例18-20: 使用下划线不会绑定值

上面的代码能很好的运行. 因为我们没有把 `s` 绑定到其它地方, 它没有被转移.

## 用 `..` 忽略剩余的值

对于有多个字段的值而言, 我们可以只提取少数字段并使用 `..` 来代替下划线, 这就避免了用 `_` 把剩余的部分列出来的麻烦. `..` 模式将忽略值中没有被精确匹配值中的其它部分. 在例18-21中, 我们有一个持有三维空间坐标的 `Point` 结构. 在 `match` 表达式里, 我们只想操作 `x` 坐标上的值并忽略 `y` 坐标和 `z` 坐标上的值:

```
struct Point {
 x: i32,
 y: i32,
 z: i32,
}

let origin = Point { x: 0, y: 0, z: 0 };

match origin {
 Point { x, .. } => println!("x is {}", x),
}
```

例18-21: 通过用 `..` 来忽略除了 `x` 以外的所有其它 `Point` 的字段

使用 `..` 比列出 `y: _` 和 `z: _` 写起来更简单. 当一个结构有很多字段但却只需要使用少量字段时 `..` 模式就特别有用.

`..` 将会囊括它能匹配的尽可能多的值. 例18-22显示了一个在元组中使用 `..` 的情况:

```
fn main() {
 let numbers = (2, 4, 8, 16, 32);
```

```
match numbers {
 (first, .., last) => {
 println!("Some numbers: {}, {}", first, last);
 },
}
```

例18-22: 用 `..` 匹配元组中的第一和最后一个值并忽略掉所有的其它值

我们在这里用 `first` 和 `last` 来匹配了第一和最后一个值. `..` 将匹配并忽略中间的所有其它值.

然而使用 `..` 必须清晰明了. 例18-23中的代码就不是很清晰, `Rust` 看不出哪些值时我们想匹配的, 也看不出哪些值是我们想忽略的:

```
fn main() {
 let numbers = (2, 4, 8, 16, 32);

 match numbers {
 (.., second, ..) => {
 println!("Some numbers: {}", second)
 },
 }
}
```

例18-23: 尝试含混不清地使用 `..`

如果我们编译上面的例子, 我们会得到下面的错误:

```
error: `..` can only be used once per tuple or tuple struct pattern
--> src/main.rs:5:22
|
5 | (.., second, ..) => {
| ^^
```

上面的代码中在一个值被匹配到 `second` 之前不可能知道元组中有多少值应该被忽略, 同样在 `second` 被匹配后也不知道应该有多少值被忽略. 我们可以忽略2, 把 `second` 绑定到4, 然后忽略8、16和32, 或者我们也可以忽略2和4, 把 `second` 绑定到8, 然后再忽略16和32. 对`Rust`而言, 变量名 `second` 并不意味着某个确定的值, 因为像这样在两个地方使用 `..` 是含混不清的, 所以我们就得到了一个编译错误.

## 用 `ref` 和 `ref mut` 在模式中创建引用

当你匹配一个模式时, 模式匹配的变量会被绑定到一个值. 也就是说你会把值转移进 `match` (或者是其它你使用了模式的地方), 这是所有权规则的作用. 例18-24提供了一个例子:

```
let robot_name = Some(String::from("Bors"));

match robot_name {
 Some(name) => println!("Found a name: {}", name),
 None => (),
}

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

例18-24: 在一个匹配分支模式里创建的变量会拥有值的所有权

上例的代码不能编译通过, 因为 `robot_name` 中的值被转移到了 `match` 中的 `Some` 的值所绑定的 `name` 里了.

在模式中使用 `&` 会匹配已存在的引用中的值, 我们在"解构并提取值"这一节中已经见过了. 如果你想创建一个引用来借用模式中变量的值, 可以在新变量名前使用 `ref` 关键字, 比如例18-25:

```
let robot_name = Some(String::from("Bors"));

match robot_name {
 Some(ref name) => println!("Found a name: {}", name),
 None => (),
}

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

例18-25: 创建一个引用这样模式中的变量就不会拥有值的所有权

上例可以编译, 因为 `robot_name` 没有被转移到 `Some(ref name)` 匹配分支的 `Some` 变量中; 这个匹配分支只是持有 `robot_name` 中的数据, `robot_name` 并没被转移.

如果要创建一个可变引用, 可以像例18-26那样使用 `ref mut` :

```
let mut robot_name = Some(String::from("Bors"));

match robot_name {
 Some(ref mut name) => *name = String::from("Another name"),
 None => (),
}

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

例18-26: 在模式中使用 `ref mut` 来创建一个值的可变引用

上例可以编译并打印出 `robot_name is: Some("Another name")`. 因为在匹配分支的代码中 `name` 是一个可变引用, 为了能够改变这个值, 我们需要用 `*` 操作符来对它解引用.

## 用了匹配守卫的额外条件

你可以通过在模式后面指定一个额外的 `if` 条件来往匹配分支中引入匹配守卫(*match guards*). 这个条件可以使用模式中创建的变量. 例18-27中的 `match` 表达式的第一个匹配分支就有一个匹配守卫:

```
let num = Some(4);

match num {
 Some(x) if x < 5 => println!("less than five: {}", x),
 Some(x) => println!("{}", x),
 None => (),
}
```

例18-27: 往一个模式中加入匹配守卫

上例会打印 `less than five: 4`. 如果把 `num` 换成 `Some(7)`, 上例将会打印 `7`. 匹配守卫让你能表达出模式不能给予你的更多的复杂的东西.

在例18-10中, 我们见过了模式中的阴影变量, 当一个值等于 `match` 外部的变量时我们不能用模式来表达出这种情况. 例18-28演示了我们如何用一个匹配守卫来解决这个问题:

```
fn main() {
 let x = Some(5);
 let y = 10;
```

```
match x {
 Some(50) => println!("Got 50"),
 Some(n) if n == y => println!("Matched, n = {:?}", n),
 _ => println!("Default case, x = {:?}", x),
}

println!("at the end: x = {:?}", y = {:?}", x, y);
}
```

例18-28: 用一个匹配守卫来测试与外部变量的相等性

上例会打印出 `Default case, x = Some(5)`。因为第二个匹配分支没有往模式中引入新变量 `y`，所以外部变量 `y` 就不会被遮掩，这样我们就可以在匹配守卫中直接使用外部变量 `y`。我们还把 `x` 解构到了内部变量 `n` 中，这样我们就可以在匹配守卫中比较 `n` 和 `y` 了。

如果你在由 `|` 指定的多模式中使用匹配守卫，匹配守卫的条件就会应用到所有的模式上。例18-29演示了在第一个匹配分支中的匹配守卫会在被匹配的全部三个模式的值上生效：

```
let x = 4;
let y = false;

match x {
 4 | 5 | 6 if y => println!("yes"),
 _ => println!("no"),
}
```

例18-29: 用一个匹配守卫来合成多个模式

上例会打印 `no` 因为条件 `if` 会应用到整个模式 `4 | 5 | 6` 上，而不是只应用到最后一个值 `6` 上面。换一种说法，一个与模式关联的匹配守卫的优先级是：

```
(4 | 5 | 6) if y => ...
```

而不是：

```
4 | 5 | (6 if y) => ...
```

## @ 绑定

为了既能测试一个模式的值又能创建一个绑定到值的变量，我们可以使用 `@`。例18-30演示了在匹配分支中我们想测试一个 `Message::Hello` 的 `id` 字段是否位于 `3...7` 之间，同时我们又想绑定这个值这样我们可以在代码中使用它：

```
enum Message {
 Hello { id: i32 },
}

let msg = Message::Hello { id: 5 };

match msg {
 Message::Hello { id: id @ 3...7 } => {
 println!("Found an id in range: {}", id)
 },
 Message::Hello { id: 10...12 } => {
 println!("Found an id in another range")
 },
 Message::Hello { id } => {
```

```
println!("Found some other id: {}", id)
 },
}
```

例18-30: 在测试模式中的值的时候用 `@` 符号来绑定值

上例会打印 `Found an id in range: 5` . 通过在范围前指定 `id @` , 我们就在测试模式的同时又捕获了匹配范围的值. 在第二个分支我们只有一个在模式中指定的范围, 与这个分支关联的代码就不知道 `id` 是10还是11或12, 因为我们没有把 `id` 的值保存在某个变量中: 我们只知道如果匹配分支代码被执行这个值与范围匹配. 在最后一个匹配分支中我们指定了一个无范围的变量, 这个值就可以用在分支代码中, 此时我们没有对这个值进行任何其它的测试. 在一个模式中使用 `@` 让我们可以测试模式中的值并把它保存在一个变量中.

## 总结

模式是Rust的一个很有用的特点, 它帮助区分不同类型的数据. 当被用在 `match` 语句中时, Rust确保你的模式覆盖了每个可能的值. 在 `let` 语句和函数参数中的模式使得这些构造更加强大, 这些模式在赋值给变量的同时可以把值解构成更小的部分.

现在让我们进入倒数第二章吧, 让我们看一下Rust的某些高级特性.

# 高级特征

[ch19-00-advanced-features.md](#)

commit d06a6a181fd61704cbf7feb55bc61d518c6469f9

我们已经走得很远了！现在我们已经学习了 99% 的编写 Rust 时需要了解的内容。在第二十章开始的新项目之前，让我们聊聊你可能会遇到的最后 1% 的内容。你可以随意跳过本章并在遇到这些问题时再回过头来；这里将要学习的特征在某些非常特定的情况下非常有用。我们并不想我们不想舍弃这些特性，但你会发现不会经常用到他们。

本章将覆盖如下内容：

- 不安全 Rust：用于当需要舍弃 Rust 的某些保证并告诉编译器你将会负责维持这些保证
- 高级生命周期：用于负责情形的额外的生命周期语法
- 高级 trait：与 trait 相关的关联类型，默认类型参数，完全限定语法（fully qualified syntax），超（父）trait（supertraits）和 newtype 模式
- 高级类型：关于 newtype 模式的更多内容，类型别名，“never”类型和动态大小类型
- 高级函数和闭包：函数指针和返回闭包

对所有人而言，这都是一个介绍 Rust 迷人特性的宝典！让我们翻开它吧！

# 不安全的Rust

在本书之前的章节, 我们讨论了Rust代码在编译时会强制保证内存安全. 然而, Rust还有另一个隐藏的语言特性, 这就是不安全的Rust, 它不会担保内存安全. 不安全的Rust和常规Rust代码无异, 但是它会给你安全的Rust代码不具备的超能力.

不安全的Rust之所以存在, 本质上是因为编译器对代码的静态分析趋于保守. 代码何时保证内存安全, 何时放权这种担保呢? 把合法的代码拒绝掉通常比接纳非法的代码要好一点. 有些时候你的代码的确没问题, 但是Rust却不这样认为! 这时你可以用不安全的代码告诉编译器, "相信我吧, 我知道我在做什么." 这样缺陷可能就在于你自己了; 如果你的不安全代码发生了错误, 比如对null指针解引用就可能会引发内存出错的大问题.

还有另一个Rust需要不安全代码的原因: 底层电脑硬件固有的不安全性. 如果Rust不让你执行不安全的操作, 那么有些任务你就完成不了. 但是Rust需要你能够做像直接与操作系统交互甚至是写你自己的操作系统这样的底层操作! 这也是Rust语言的一部分目标, 所以我们需要一些来做这些事情的方法.

## 不安全的神力

我们通过使用 `unsafe` 关键字开启一个持有不安全代码的代码块来切换到不安全的Rust. 你可以在不安全的Rust中进行四个安全的Rust做不到的操作. 我们把它们称作"不安全的神力". 之前我们没见过这几个特性是因为它们只在 `unsafe` 代码块中! 它们是:

1. 解引用原生指针
2. 调用一个不安全的函数或方法
3. 访问或修改一个不可变的静态变量
4. 实现一个不安全的trait

记住这一点很重要, `unsafe` 不会关掉借用检查器也不会禁用其它的Rust安全性检查: 如果你在不安全的代码中用了引用, 它仍将会被检查. `unsafe` 关键字做的唯一的一件事是让你存取编译器因内存安全性而没去检查的上述四个特性. 在一个`unsafe`代码块中你仍然会获得某种程度的安全性! 此外, `unsafe` 并不是说代码块中的代码是危险的或者有内存安全性问题: 它只是表明作为程序员的你关掉了编译器检查, 你将确保 `unsafe` 代码块会拥有合理的内存.

人是会犯错误的, 错误总会发生. 在 `unsafe` 代码块中执行上述四个不安全的操作时, 如果你犯了错误并得到一个内存安全性的错误, 你必定会知道它与你使用不安全的代码有关. 这样就更容易处理内存安全性的bug, 因为Rust已经帮我们其它的代码做了检查. 能缩小排查内存安全性bug的出现区域当然好, 所以尽量缩小你的不安全代码的数量吧. 当修正内存安全问题时, `unsafe` 代码块中的任何代码都可能出错: 所以让 `unsafe` 代码块尽可能的小吧, 以后你需要排查的代码也会少一些.

为了尽可能隔离不安全的代码, 在安全的抽象中包含不安全的代码并提供一个安全的API是一个好主意, 当我们学习不安全的函数和方法时我们会讨论它. 标准库中有些不安全的代码被实现为安全的抽象, 它们中的部分已被审核过了. 当你或者你的用户使用通过 `unsafe` 代码实现的功能时, 因为使用一个安全的抽象是安全的, 这样就可以避免到处都是 `unsafe` 字样.

让我们按顺序依次介绍上述四个不安全的神力, 同时我们会见到一些抽象, 它们为不安全的代码提供了安全的接口.

## 解引用原生指针

回到第4章, 我们在哪里学习了引用. 我们知道编译器会确保引用永远合法. 不安全的Rust有两个类似于引用的被称为原生指针(*raw pointers*)的新类型. 和引用一样, 我们可以有一个不可变的原生指针和一个可变的原生指针. 在原生指针的上下文中, "不可变"意味着指针不能被解引用和被赋值. 例19-1演示了如何通过引用来创建一个原生指针:

```
let mut num = 5;
```



```
let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;
```

### 例19-1: 通过引用创建原生指针

上例中 `*const T` 类型是一个不可变的原生指针, `*mut T` 是一个可变的原生指针. 我们通过使用 `as` 把一个可变的和一个不可变的引用转换成它们对应的原生指针类型来创建原生指针. 与引用不同, 这些指针的合法性不能得到保证.

例19-2演示了如何创建一个指向内存中任意地址的原生指针. 试图随便访问内存地址所带来的结果是难以预测的: 也许在那个地址处有数据, 也许在那个地址处没有任何数据, 编译器也可能会优化代码导致那块内存不能访问, 亦或你的程序可能会发生段错误. 虽然可以写出下面的代码, 但是通常找不到好的理由来这样做:

```
let address = 0x012345;
let r = address as *const i32;
```

### 例子19-2: 创建一个指向任意内存地址的原生指针

注意在例19-1和19-2中没有 `unsafe` 代码块. 你可以在安全代码中创建原生指针 **raw pointers in safe code**, 但是你不能在安全代码中解引用(**dereference**)原生指针来读取被指针指向的数据. 如例19-3所示, 对原生指针解引用需要在 `unsafe` 代码块中使用解引用操作符 `*`:

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;

unsafe {
 println!("r1 is: {}", *r1);
 println!("r2 is: {}", *r2);
}
```

### 例19-3: 在 `unsafe` 代码块中解引用原生指针

创建一个指针不会造成任何危险; 只有在你访问指针指向的值时可能出问题, 因为你可能会用它处理无效的值.

注意在19-1和19-3中我们创建的一个 `*const i32` 和一个 `*mut i32` 都指向同一个内存位置, 也就是 `num`. 如果我们尝试创建一个不可变的和可变的 `num` 的引用而不是原生指针, 这就不能被编译, 因为我们不能在使用了不可变引用的同时再对同一个值进行可变引用. 通过原生指针, 我们能创建指向同一个内存位置的可变指针和不可变指针, 我们可以通过可变指针来改变数据, 但是要小心, 因为这可能会产生数据竞争!

既然存在这么多的危险, 为什么我们还要使用原生指针呢? 一个主要的原因是为了与C代码交互, 在下一节的不安全函数里我们将会看到. 另一个原因是创建一个借用检查器理解不了的安全的抽象. 下面让我们介绍不安全的函数, 然后再看一个使用了不安全代码的安全的抽象的例子.

## 调用一个不安全的函数或方法

需要一个不安全的代码块的才能执行的第二个操作是调用不安全的函数. 不安全的函数和方法与常规的函数和方法看上去没有什么异样, 只是他们前面有一个额外的 `unsafe` 关键字. 不安全的函数的函数体自然是 `unsafe` 的代码块.

下例是一个名叫 `dangerous` 的不安全的函数:

```
unsafe fn dangerous() {}

unsafe {
 dangerous();
}
```

如果不用 `unsafe` 代码块来调用 `dangerous` , 我们将会得到下面的错误:

```
error[E0133]: call to unsafe function requires unsafe function or block
--> <anon>:4:5
|
4 | dangerous();
| ^^^^^^^^^^^ call to unsafe function
```

通过把对 `dangerous` 的调用放到 `unsafe` 代码块中, 我们表明我们已经阅读了该函数的文档, 我们明白如何正确的使用它, 并且我们已经验证了调用的正确性.

## 创建一个不安全的代码上的安全的抽象

让我们用标准库中的某个函数比如 `split_at_mut` 来举个例子, 然后来探讨我们如何自己来实现它. 这个方法被定义在一个可变的切片(`slice`)上, 它通过参数指定的索引把一个切片分割成两个, 如例19-4所示:

```
let mut v = vec![1, 2, 3, 4, 5, 6];

let r = &mut v[..];

let (a, b) = r.split_at_mut(3);

assert_eq!(a, &mut [1, 2, 3]);
assert_eq!(b, &mut [4, 5, 6]);
```

例19-4: 使用安全的 `split_at_mut` 函数

用安全的Rust代码是不能实现这个函数的. 如果要试一下用安全的Rust来实现它可以参考例19-5. 简单起见, 我们把 `split_at_mut` 实现成一个函数而不是一个方法, 这个函数只处理 `i32` 类型的切片而不是泛型类型 `T` 的切片:

```
fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
 let len = slice.len();

 assert!(mid <= len);

 (&mut slice[..mid],
 &mut slice[mid..])
}
```

例19-5: 尝试用安全的Rust来实现 `split_at_mut`

该函数先取得切片(`slice`)的长度, 然后通过检查参数是否小于或等于这个长度来断言参数给定的索引位于切片(`slice`)当中. 这个断言意味着如果我们传入的索引比要分割的切片(`slice`)的长度大, 这个函数就会在使用这个索引前中断(`panic`).

接着我们在一个元组中返回两个可变的切片(`slice`): 一个从被分割的切片的头部开始直到 `mid` 索引的前一个元素中止, 另一个从被分割的切片的 `mid` 索引开始直到被分割的切片的末尾结束.

如果我们编译上面的代码, 我们将得到一个错误:

```
error[E0499]: cannot borrow `*slice` as mutable more than once at a time
--> <anon>:6:11
|
5 | (&mut slice[..mid],
| ----- first mutable borrow occurs here
6 | &mut slice[mid..])
```

```

| ^^^^^ second mutable borrow occurs here
7 | }
| - first borrow ends here

```

Rust的借用检查器不能理解为什么我们要借用这个切片(slice)的不同部分; 它只知道我们对同一个切片借用了两次. 借用一个切片(slice)的不同部分在功能上是没有问题的; 而且我们的两个 `&mut [i32]` 也没有重叠. 但是Rust并没有聪明到能明白这一点. 当我们知道有些东西是可以的但是Rust却不知道的时候就是时候使用不安全的代码了.

例子19-6演示了如何用 `unsafe` 代码块、一个原生指针和一个不安全的函数调用来实现 `split_at_mut`:

```

use std::slice;

fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
 let len = slice.len();
 let ptr = slice.as_mut_ptr();

 assert!(mid <= len);

 unsafe {
 (slice::from_raw_parts_mut(ptr, mid),
 slice::from_raw_parts_mut(ptr.offset(mid as isize), len - mid))
 }
}

```

例19-6: 用不安全的代码来实现 `split_at_mut`

回顾一下第4章, 切片(slice)是一个指向某个数据的指针和这个切片(slice)的长度. 我们经常用 `len` 方法来取得切片的长度; 也可以用 `as_mut_ptr` 方法来访问切片的原生指针. 在这个例子里, 因为我们有一个可变的 `i32` 类型的切片, `as_mut_ptr` 返回一个 `*mut i32` 类型的原生指针, 我们把它存放在变量 `ptr` 里.

对索引 `mid` 合法性的断言上面已经介绍过了. 函数 `slice::from_raw_parts_mut` 的行为与 `as_mut_ptr` 和 `len` 方法相反: 它以一个原生指针和一个长度为参数并返回一个切片(slice). 我们调用 `slice::from_raw_parts_mut` 来创建一个从 `ptr` 开始且拥有 `mid` 个元素的切片. 然后我们以 `mid` 为参数调用 `ptr` 上的 `offset` 方法来得到一个从索引 `mid` 开始的原生指针, 然后我们用这个原生指针和索引 `mid` 之后的元素个数为参数创建一个切片.

因为切片(slice)会被检查, 所以一旦我们创建了它就可以安全使用. 函数 `slice::from_raw_parts_mut` 是一个不安全的函数因为它有一个原生指针参数, 而且它相信这个指针是有效的. 原生指针的 `offset` 方法也是不安全的, 因为它相信一个原生指针的位置偏移一些后也是一个有效的指针. 为了能调用 `slice::from_raw_parts_mut` 和 `offset`, 我们把他们的调用放到一个 `unsafe` 代码块中, 我们可以通过查看代码并添加 `mid` 不大于 `len` 的断言来表明 `unsafe` 代码块中的原生指针是指向切片中的数据的有效指针. 这是一个 `unsafe` 恰当用法.

注意结果 `split_at_mut` 函数是安全的: 我们不用在它的前面添加 `unsafe` 关键字, 并且我们可以从安全的Rust代码中调用它. 我们通过写一个使用了 `unsafe` 代码的函数来创建不安全代码的安全抽象, 上例用一种安全的方式通过函数访问的数据来创建有效的指针.

相反, 当使用切片(slice)时, 例19-7中 `slice::from_raw_parts_mut` 的用法很可能会崩溃. 下面的代码用一个随意的内存地址来创建一个有10000个元素的切片:

```

use std::slice;

let address = 0x012345;
let r = address as *mut i32;

let slice = unsafe {
 slice::from_raw_parts_mut(r, 10000)
};

```

例19-7: 通过一个任意的内存位置来创建一个切片

我们不能拥有任意地址的内存, 也不能保证这个代码创建的切片会包含有效的 `i32` 类型的值. 试图使用臆测是有效切片的 `slice` 的行为是难以预测的.

## 调用外部代码的 `extern` 函数是不安全的

有时, 你的Rust代码需要与其它语言交互. Rust有一个 `extern` 关键字可以实现这个功能, 这有助于创建并使用外部功能接口(*Foreign Function Interface*) (FFI). 例19-8演示了如何与定义在一个非Rust语言编写的外部库中的 `some_function` 进行交互. 在Rust中调用 `extern` 声明的代码块永远都是不安全的:

Filename: src/main.rs

```
extern "C" {
 fn some_function();
}

fn main() {
 unsafe { some_function() };
}
```

例19-8: 声明并调用一个用其它语言写成的函数

在 `extern "C"` 代码块中, 我们列出了我们想调用的用其它语言实现的库中定义的函数名和这个函数的特征签名. `"C"` 定义了外部函数使用了哪种应用程序接口(*application binary interface*) (ABI). ABI定义了如何在汇编层调研这个函数. `"C"` 是最常用的遵循C语言的ABI.

调用一个外部函数总是不安全的. 如果我们要调用其他语言, 这种语言却不会遵循Rust的安全保证. 因为Rust不能检查外部代码是否是安全的, 我们只负责检查外部代码的安全性来表明我们已经用 `unsafe` 代码块来调用外部函数了.

通过其它语言来调用Rust函数

`extern` 关键字也总是被用来创建一个允许被其他语言调用的Rust函数. 与 `extern` 代码块不同, 我们可以在 `fn` 关键字之前添加 `extern` 关键字并指定要使用的ABI. 我们也加入注解 `#[no_mangle]` 来告诉Rust编译器不要取消这个函数的名字. 一旦我们把下例的代码编译成一个共享库并链接到C, 这个 `call_from_c` 函数就可以被C代码访问了:

```
#[no_mangle]
pub extern "C" fn call_from_c() {
 println!("Just called a Rust function from C!");
}
```

上例的 `extern` 不需要 `unsafe` 就可以直接用

## 访问或修改一个可变的静态变量

目前为止本书还没有讨论全局变量(*global variables*). 很多语言都支持全局变量, 当然Rust也不例外. 然而全局变量也有问题: 比如, 如果两个线程访问同一个可变的全局变量有可能会发生数据竞争.

全局变量在Rust中被称为是静态(*static*)变量. 例19-9中声明并使用了一个字符串切片类型的静态变量:

Filename: src/main.rs

```
static HELLO_WORLD: &str = "Hello, world!";

fn main() {
 println!("name is: {}", HELLO_WORLD);
}
```

```
}
```

#### 例19-9: 定义和使用一个不可变的静态变量

`static` 变量类似于常量: 按照惯例它们的命名遵从 `SCREAMING_SNAKE_CASE`(用下划线分割的全大写字母) 风格, 我们也必须注明变量的类型, 本例中是 `&'static str`。只有定义为 `'static` 的生命期才可以被存储在一个静态变量中。也正因为此, **Rust**编译器自己就已经很清楚静态变量的生命期了, 所以我们也不需要明确地注明它了。访问不可变的静态变量是安全的。因为静态变量的值有一个固定的内存地址, 所以使用该值的时候总会得到同样的数据。另一方面, 当常量被使用时, 复制它们的数据也是被允许的。

静态变量与常量的另一个不同是静态变量可以是可变的。访问和修改可变的静态变量都是不安全的。例19-10演示了如何声明、访问和修改一个名叫 `COUNTER` 的可变的静态变量:

Filename: src/main.rs

```
static mut COUNTER: u32 = 0;

fn add_to_count(inc: u32) {
 unsafe {
 COUNTER += inc;
 }
}

fn main() {
 add_to_count(3);

 unsafe {
 println!("COUNTER: {}", COUNTER);
 }
}
```

#### 例19-10: 读取或修改一个可变的静态变量是不安全的

与常规变量一样, 我们用 `mut` 关键字来表明这个静态变量是可变的。每次我们对 `COUNTER` 的读写都必须被放到一个 `unsafe` 代码块中。上面的代码编译运行会打印 `COUNTER: 3`, 这正如我们期望的那样, 因为程序现在是一个单线程, 如果有多个线程访问 `COUNTER` 就可能会导致数据竞争。

可全局访问的可变数据难于管理也很难保证没有数据竞争, 这也正是**Rust**认为可变的静态变量是不安全的原因。如果可能, 请使用在第16章中介绍的并发技术和线程安全的智能指针, 这样可以让编译器从不同的线程检查被访问的数据是安全的。

## 实现一个不安全的Trait

最后, 当我们使用 `unsafe` 关键字时最后一个只在不安全的代码中才能做的事是实现一个不安全的**trait**。我们可以在 `trait` 之前添加一个 `unsafe` 关键字来声明一个**trait**是不安全的, 以后实现这个**trait**的时候也必须标记一个 `unsafe` 关键字, 如19-11所示:

```
unsafe trait Foo {
 // methods go here
}

unsafe impl Foo for i32 {
 // method implementations go here
}
```

#### 例19-11: 定义并实现一个不安全的trait

与不安全的函数类似, 一个不安全的`trait`中的方法也有一些编译器无法验证的盲点. 通过使用 `unsafe impl`, 我们就是在说明我们来保证这些有疑虑的地方的安全.

举个例子, 回想一下第16章中的 `Sync` 和 `Send` 这两个标记`trait`, 如果我们的类型全部由 `Send` 和 `Sync` 类型组合而成, 编译器会自动实现它们. 如果我们要实现的一个类型包含了不是 `Send` 或 `Sync` 的东西, 比如原生指针, 若是我们像把我们的类型标记成 `Send` 或 `Sync`, 这就要求使用 `unsafe` 关键字. **Rust**不能验证我们的类型能保证可以安全地跨线程发送或从多个线程访问, 所以我们需要用 `unsafe` 关键字来表明我们会自己来做这些检查.

使用 `unsafe` 来执行这四个动作之一是没有问题的, 因为编译器不能确保内存安全, 所以把 `unsafe` 代码写正确也实属不易. 当你需要使用 `unsafe` 代码时, 你可以这样做, 明确注明 `unsafe`, 这样如果出现问题可以更容易地追踪问题的源头.

# 高级生命周期

ch19-02-advanced-lifetimes.md

commit d06a6a181fd61704cbf7feb55bc61d518c6469f9

回顾第十章，我们学习了怎样使用生命周期参数来注解引用来帮助 **Rust** 理解不同引用的生命周期如何相互联系。见识到了大部分情况 **Rust** 允许我们省略生命周期，不过每一个引用都有一个生命周期。这里有三个生命周期的高级特征我们还未讲到：生命周期子类型（*lifetime subtyping*），生命周期约束（*lifetime bounds*），以及 **trait** 对象生命周期（*trait object lifetimes*）。

## 生命周期子类型

想象一下我们想要编写一个解析器。为此，会有一个储存了需要解析的字符串的引用的结构体，我们称之为结构体 `Context`。解析器将会解析字符串并返回成功或失败。解析器需要借用 `Context` 来进行解析。其实现看起来像列表 19-12 中的代码，它还不能编译，因为目前我们去掉了生命周期注解：

```
struct Context(&str);

struct Parser {
 context: &Context,
}

impl Parser {
 fn parse(&self) -> Result<(), &str> {
 Err(&self.context.0[1..])
 }
}
```

列表 19-12：定义结构体 `Context` 来存放一个字符串 `slice`，结构体 `Parser` 包含一个 `Context` 实例和一个 `parse` 方法，它总是返回一个引用了字符串 `slice` 的错误

为了简单起见，`parse` 方法返回 `Result<(), &str>`。也就是说，成功时不做任何操作，失败时则返回字符串 `slice` 没有正确解析的部分。真实的实现将会包含比这更多的错误信息，也将会在解析成功时返回创建的结果，不过我们将去掉这些部分的实现，因为他们与这个例子的生命周期部分并不相关。我们还定义了 `parse` 总是在第一个字节之后返回错误。注意如果第一个字节并不位于一个有效的字符范围内（比如 **Unicode**）将会 `panic`；我们有一次简化了例子以专注于涉及到的生命周期。

那么我们如何为 `Context` 中的字符串 `slice` 和 `Parser` 中 `Context` 的引用放入生命周期参数呢？最直接的方法是在每处都使用相同的生命周期，如列表 19-13 所示：

```
struct Context<'a>(&'a str);

struct Parser<'a> {
 context: &'a Context<'a>,
}

impl<'a> Parser<'a> {
 fn parse(&self) -> Result<(), &str> {
 Err(&self.context.0[1..])
 }
}
```

列表 19-13：将所有 `Context` 和 `Parser` 的引用标注为相同的生命周期参数

这次可以编译了。接下来，在列表 19-14 中，让我们编写一个获取 `Context` 的实例，使用 `Parser` 来解析其内容，并返回 `parse` 的返回值的函数。这还不能运行：

```
fn parse_context(context: Context) -> Result<(), &str> {
 Parser { context: &context }.parse()
}
```

列表 19-14：一个增加获取 `Context` 并使用 `Parser` 的函数 `parse_context` 的尝试

当尝试编译这段额外带有 `parse_context` 函数的代码时会得到两个相当冗长的错误：

```
error: borrowed value does not live long enough
--> <anon>:16:5
|
16 | Parser { context: &context }.parse()
| ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ does not live long enough
17 | }
| - temporary value only lives until here
|
note: borrowed value must be valid for the anonymous lifetime #1 defined on the
body at 15:55...
--> <anon>:15:56
|
15 | fn parse_context(context: Context) -> Result<(), &str> {
| _____^
16 | | Parser { context: &context }.parse()
17 | | }
| |_^

error: `context` does not live long enough
--> <anon>:16:24
|
16 | Parser { context: &context }.parse()
| ^^^^^^ does not live long enough
17 | }
| - borrowed value only lives until here
|
note: borrowed value must be valid for the anonymous lifetime #1 defined on the
body at 15:55...
--> <anon>:15:56
|
15 | fn parse_context(context: Context) -> Result<(), &str> {
| _____^
16 | | Parser { context: &context }.parse()
17 | | }
| |_^
```

这些错误表明我们创建的两个 `Parser` 实例和 `context` 参数从 `Parser` 被创建开始一直存活到 `parse_context` 函数结束，不过他们都需要在整个函数的生命周期中都有效。

换句话说，`Parser` 和 `context` 需要比整个函数长寿（*outlive*）并在函数开始之前和结束之后都有效以确保代码中的所有引用始终是有效的。虽然两个我们创建的 `Parser` 和 `context` 参数在函数的结尾就离开了作用域（因为 `parse_context` 获取了 `context` 的所有权）。

让我们再次看看列表 19-13 中的定义，特别是 `parse` 方法的签名：

```
fn parse(&self) -> Result<(), &str> {
```

还记得（生命周期）省略规则吗？如果标注了引用生命周期，签名看起来应该是这样：



```
fn parse<'a>(&'a self) -> Result<(), &'a str> {
```

正是如此，`parse` 返回值的错误部分的生命周期与 `Parser` 实例的生命周期（`parse` 方法签名中的 `&self`）相绑定。这就可以理解了，因为返回的字符串 `slice` 引用了 `Parser` 存放的 `Context` 实例中的字符串 `slice`，同时在 `Parser` 结构体的定义中我们指定了 `Parser` 中存放的 `Context` 引用的生命周期和 `Context` 中存放的字符串 `slice` 的生命周期应该一致。

问题是 `parse_context` 函数返回 `parse` 返回值，所以 `parse_context` 返回值的生命周期也与 `Parser` 的生命周期相联系。不过 `parse_context` 函数中创建的 `Parser` 实例并不能存活到函数结束之后（它是临时的），同时 `context` 将会在函数的结尾离开作用域（`parse_context` 获取了它的所有权）。

不允许一个在函数结尾离开作用域的值引用。Rust 认为这是我们想要做的，因为我们将所有生命周期用相同的生命周期参数标记。这告诉了 Rust `Context` 中存放的字符串 `slice` 的生命周期与 `Parser` 中存放的 `Context` 引用的生命周期一致。

`parse_context` 函数并不知道 `parse` 函数里面是什么，返回的字符串 `slice` 将比 `Context` 和 `Parser` 都存活的更久，因此 `parse_context` 返回的引用指向字符串 `slice`，而不是 `Context` 或 `Parser`。

通过了解 `parse` 实现所做的工作，可以知道 `parse` 的返回值（的生命周期）与 `Parser` 相联系的唯一理由是它引用了 `Parser` 的 `Context`，也就是引用了这个字符串 `slice`，这正是 `parse_context` 所需要关心的生命周期。需要一个方法来告诉 Rust `Context` 中的字符串 `slice` 与 `Parser` 中 `Context` 的引用有着不同的生命周期，而且 `parse_context` 返回值与 `Context` 中字符串 `slice` 的生命周期相联系。

我们只能尝试像列表 19-15 那样给予 `Parser` 和 `Context` 不同的生命周期参数。这里选择了生命周期参数名 `'s` 和 `'c` 是为了使得 `Context` 中字符串 `slice` 与 `Parser` 中 `Context` 引用的生命周期显得更明了（英文首字母）。注意这并不能完全解决问题，不过这是一个开始，我们将看看为什么这还不足以能够编译代码。

```
struct Context<'s>(&'s str);

struct Parser<'c, 's> {
 context: &'c Context<'s>,
}

impl<'c, 's> Parser<'c, 's> {
 fn parse(&self) -> Result<(), &'s str> {
 Err(&self.context.0[1..])
 }
}

fn parse_context(context: Context) -> Result<(), &str> {
 Parser { context: &context }.parse()
}
```

列表 19-15：为字符串 `slice` 和 `Context` 的引用指定不同的生命周期参数

这里在与列表 19-13 完全相同的地方标注了引用的生命周期，不过根据引用是字符串 `slice` 或 `Context` 与否使用了不同的参数。另外还在 `parse` 返回值的字符串 `slice` 部分增加了注解来表明它与 `Context` 中字符串 `slice` 的生命周期相关联。

这里是现在得到的错误：

```
error[E0491]: in type `&'c Context<'s>`, reference has a longer lifetime than the data it references
--> src/main.rs:4:5
 |
4 | context: &'c Context<'s>,
 | ^^^^^^^^^^^^^^^^^^^^^^^^^
 |
```

```

note: the pointer is valid for the lifetime 'c as defined on the struct at 3:0
--> src/main.rs:3:1
|
3 | / struct Parser<'c, 's> {
4 | | context: &'c Context<'s>,
5 | | }
 | |_^
note: but the referenced data is only valid for the lifetime 's as defined on the struct at 3:0
--> src/main.rs:3:1
|
3 | / struct Parser<'c, 's> {
4 | | context: &'c Context<'s>,
5 | | }
 | |_^

```

**Rust** 并不知道 `'c` 与 `'s` 之间的任何联系。为了保证有效性，`Context` 中引用的带有生命周期 `'s` 的数据需要遵守它比带有生命周期 `'c` 的 `Context` 的引用存活得更久的保证。如果 `'s` 不比 `'c` 更长久，那么 `Context` 的引用可能不再有效。

这就引出了本部分的要点：**Rust** 有一个叫做生命周期子类型的功能，这是一个指定一个生命周期不会短于另一个的方法。在声明生命周期参数的尖括号中，可以照常声明一个生命周期 `'a`，并通过语法 `'b: 'a` 声明一个不短于 `'a` 的生命周期 `'b`。

在 `Parser` 的定义中，为了表明 `'s`（字符串 `slice` 的生命周期）保证至少与 `'c`（`Context` 引用的生命周期）一样长，需将生命周期声明改为如此：

```

struct Context<'a>(&'a str);
#
struct Parser<'c, 's: 'c> {
 context: &'c Context<'s>,
}

```

现在 `Parser` 中 `Context` 的引用与 `Context` 中字符串 `slice` 就有了不同的生命周期，并且保证了字符串 `slice` 的生命周期比 `Context` 引用的要长。

这是一个非常冗长的例子，不过正如本章的开头所提到的，这类功能是很小众的。你并不会经常需要这个语法，不过当出现类似这样的情形时，却还是有地方可以参考的。

## 生命周期约束(lifetime bound)

在第十章，我们讨论了如何在泛型类型上使用 `trait bound`。也可以像泛型那样为生命周期参数增加限制，这被称为生命周期约束。例如，考虑一下一个封装了引用的类型。回忆一下第十五章的 `RefCell<T>` 类型：其 `borrow` 和 `borrow_mut` 方法分别返回 `Ref` 和 `RefMut` 类型。这些类型是引用的封装，他们在运行时记录检查借用规则。`Ref` 结构体的定义如列表 19-16 所示，现在还不带有生命周期约束：

```

struct Ref<'a, T>(&'a T);

```

列表 19-16：定义结构体来封装泛型的引用；开始时没有生命周期约束

若不限制生命周期 `'a` 为与泛型参数 `T` 有关，会得到一个错误因为 **Rust** 不知道泛型 `T` 会存活多久：

```

error[E0309]: the parameter type `T` may not live long enough
--> <anon>:1:19
|
1 | struct Ref<'a, T>(&'a T);
 | ^^^^^^

```

```
|
= help: consider adding an explicit lifetime bound `T: 'a`...
note: ...so that the reference type `&'a T` does not outlive the data it points at
--> <anon>:1:19
|
1 | struct Ref<'a, T>(&'a T);
| ^^^^^^
```

因为 `T` 可以是任意类型，`T` 自身也可能是一个引用，或者是一个存放了一个或多个引用的类型，而他们各自可能有着不同的生命周期。`Rust` 不能确认 `T` 会与 `'a` 存活的一样久。

幸运的是，`Rust` 提供了这个情况下如何指定生命周期约束 的有用建议：

```
consider adding an explicit lifetime bound `T: 'a` so that the reference type
`&'a T` does not outlive the data it points at.
```

列表 19-17 展示了按照这个建议，在声明泛型 `T` 时指定生命周期约束。现在代码可以编译了，因为 `T: 'a` 指定了 `T` 可以为任意类型，不过如果它包含任何引用的话，其生命周期必须至少与 `'a` 一样长：

```
struct Ref<'a, T: 'a>(&'a T);
```

列表19-17：为 `T` 增加生命周期约束 来指定 `T` 中的任何引用需至少与 `'a` 存活的一样久

我们可以选择不同的方法来解决这个问题，如列表 19-18 中展示的 `StaticRef` 结构体定义所示，通过在 `T` 上增加 `'static` 生命周期约束。这意味着如果 `T` 包含任何引用，他们必须有 `'static` 生命周期：

```
struct StaticRef<T: 'static>(&'static T);
```

列表 19-18：在 `T` 上增加 `'static` 生命周期约束 来限制 `T` 为只拥有 `'static` 引用或没有引用的类型

没有任何引用的类型被算作 `T: 'static`。因为 `'static` 意味着引用必须同整个程序存活的一样长，一个不包含引用的类型满足所有引用都与程序存活的一样长的标准（因为他们没有引用）。可以这样理解：如果借用检查器关心的是引用是否存活的够久，那么没有引用的类型与有永远存在的引用的类型并没有真正的区别；对于确定引用是否比其所引用的值存活得较短的目的来说两者是一样的。

## trait 对象生命周期

在第十七章，我们学习了 `trait` 对象，其中介绍了可以把一个 `trait` 放在一个引用后面来进行动态分发。然而，我们并没有讨论如果 `trait` 对象中实现 `trait` 的类型带有生命周期时会发生什么。考虑一下 19-19，这里有 `trait Foo`，和带有一个实现了 `trait Foo` 的引用（因此还有其生命周期参数）的结构体 `Bar`，我们希望使用 `Bar` 的实例作为 `trait` 对象 `Box<Foo>`：

```
trait Foo { }

struct Bar<'a> {
 x: &'a i32,
}

impl<'a> Foo for Bar<'a> { }

let num = 5;

let obj = Box::new(Bar { x: &num }) as Box<Foo>;
```

列表 19-19: 使用一个带有生命周期的类型作为 **trait** 对象

这些代码能没有任何错误的编译, 即便并没有明确指出 `obj` 中涉及的任何生命周期。这是因为有如下生命周期与 **trait** 对象必须遵守的规则:

- **trait** 对象的默认生命周期是 `'static`。
- 如果有 `&'a x` 或 `&'a mut x`, 则默认(生命周期)是 `'a`。
- 如果只有 `T: 'a`, 则默认是 `'a`。
- 如果有多个类似 `T: 'a` 的从句, 则没有默认值; 必须明确指定。

当必须明确指定时, 可以为像 `Box<Foo>` 这样的 **trait** 对象增加生命周期约束, 根据需要使用语法 `Box<Foo + 'a>` 或 `Box<Foo + 'static>`。正如其他的 **bound**, 这意味着任何 `Foo` **trait** 的实现如果在内部包含有引用, 就必须在 **trait** 对象 **bounds** 中为那些引用指定生命周期。

接下来, 让我们看看一些其他处理 **trait** 的功能吧!

# 高级 trait

ch19-03-advanced-traits.md

commit f8727711388b28eb2f5c852dd83fdbe6d22ab9bb

第十章讲到了 **trait**，不过就像生命周期，我们并没有涉及所有的细节。现在我们更加了解 **Rust** 了，可以深入理解本质了。

## 关联类型

关联类型（*associated types*）是一个将类型占位符与 **trait** 相关联的方式，这样 **trait** 的方法签名中就可以使用这些占位符类型。实现一个 **trait** 的人只需要针对专门的实现在这个类型的位置指定相应的类型即可。

本章描述的大部分内容都非常少见。关联类型则比较适中；它们比本书其他的内容要少见，不过比本章中的很多内容要更常见。

一个带有关联类型的 **trait** 的例子是标准库提供的 **Iterator trait**。它有一个叫做 **Item** 的关联类型来替代遍历的值的类型。第十三章曾提到过 **Iterator trait** 的定义如列表 19-20 所示：

```
pub trait Iterator {
 type Item;
 fn next(&mut self) -> Option<Self::Item>;
}
```

列表 19-20: **Iterator trait** 的定义中带有关联类型 **Item**

这就是说 **Iterator trait** 有一个关联类型 **Item**。**Item** 是一个占位类型，同时 **next** 方法会返回 **Option<Self::Item>** 类型的值。这个 **trait** 的实现者会指定 **Item** 的具体类型，然而不管实现者指定何种类型，**next** 方法都会返回一个包含了这种类型值的 **Option**。

## 关联类型 vs 泛型

当在列表 13-6 中在 **Counter** 结构体上实现 **Iterator trait** 时，将 **Item** 的类型指定为 **u32**：

```
impl Iterator for Counter {
 type Item = u32;

 fn next(&mut self) -> Option<Self::Item> {
```

这感觉类似于泛型。那么为什么 **Iterator trait** 不像列表 19-21 那样定义呢？

```
pub trait Iterator<T> {
 fn next(&mut self) -> Option<T>;
}
```

列表 19-21: 一个使用泛型的 **Iterator trait** 假象定义

区别是在列表 19-21 的定义中，我们也可以实现 **Iterator<String> for Counter**，或者任何其他类型，这样就可以有多个 **Counter** 的 **Iterator** 的实现。换句话说，当 **trait** 有泛型参数时，可以多次实现这个 **trait**，每次需改变泛型参数的具体类型。接着当使用 **Counter** 的 **next** 方法时，必须提供类型注解来表明希望使用 **Iterator** 的哪一个实现。

通过关联类型，不能多次实现 `trait`。使用列表 19-20 中这个 `Iterator` 的具体定义，只能选择一次 `Item` 会是什么类型，因为只能有一个 `impl Iterator for Counter`。当调用 `Counter` 的 `next` 时不必每次指定我们需要 `u32` 值的迭代器。

当 `trait` 使用关联类型时不必指定泛型参数的好处也在另外一些方面得到体现。考虑一下列表 19-22 中定义的两个 `trait`。他们都必须处理一个包含一些节点和边的图结构。`GGraph` 定义为使用泛型，而 `AGraph` 定义为使用关联类型：

```
trait GGraph<Node, Edge> {
 // methods would go here
}

trait AGraph {
 type Node;
 type Edge;

 // methods would go here
}
```

列表 19-22：两个图 `trait` 定义，`GGraph` 使用泛型而 `AGraph` 使用关联类型代表 `Node` 和 `Edge`

比如说想要是实现一个计算任何实现了图 `trait` 的类型中两个节点之间距离的函数。对于使用泛型的 `GGraph trait` 来说，`distance` 函数的签名看起来应该如列表 19-23 所示：

```
trait GGraph<Node, Edge> {}

fn distance<N, E, G: GGraph<N, E>>(graph: &G, start: &N, end: &N) -> u32 {
 # 0
}
```

列表 19-23：`distance` 函数的签名，它使用 `GGraph trait` 并必须指定所有的泛型参数

函数需要指定泛型参数 `N`、`E` 和 `G`，其中 `G` 拥有以 `N` 类型作为 `Node` 和 `E` 类型作为 `Edge` 的 `GGraph trait` 作为 `trait bound`。即便 `distance` 函数无需指定边的类型，我们也强制声明了 `E` 参数，因为需要使用 `GGraph trait`，而 `GGraph` 需要指定 `Edge` 的类型。

与此相对，列表 19-24 中的 `distance` 定义使用列表 19-22 中带有关联类型的 `AGraph trait`：

```
trait AGraph {
 # type Node;
 # type Edge;
}

fn distance<G: AGraph>(graph: &G, start: &G::Node, end: &G::Node) -> u32 {
 # 0
}
```

列表 19-24：`distance` 函数的签名，它使用 `trait AGraph` 和关联类型 `Node`

这样就清楚多了。只需指定一个泛型参数 `G`，带有 `AGraph trait bound`。因为 `distance` 完全不需要使用 `Edge` 类型，无需每次都指定它。为了使用 `AGraph` 的关联类型 `Node`，可以指定为 `G::Node`。

## 带有关联类型的 `trait` 对象

你可能会好奇为什么不在列表 19-23 和 19-24 的 `distance` 函数中使用 `trait` 对象。当使用 `trait` 对象时使用泛型 `GGraph trait` 的 `distance` 函数的签名确实更准确了一些：

```
trait GGraph<Node, Edge> {}
#
fn distance<N, E>(graph: &GGraph<N, E>, start: &N, end: &N) -> u32 {
0
}
```

与列表 19-24 相比较可能更显公平。不过依然需要指定 `Edge` 类型，这意味着列表 19-24 仍更为合适，因为无需指定并不需要的类型。

不可能改变列表 19-24 来对图使用 `trait` 对象，因为这样就无法引用 `AGraph trait` 中的关联类型。

但是一般而言常见的情形是使用带有关联类型 `trait` 的 `trait` 对象；列表 19-25 展示了一个函数 `traverse`，它无需在其他参数中使用关联类型。然而这种情况必须指定关联类型的具体类型。这里选择接受以 `usize` 作为 `Node` 和以两个 `usize` 值的元组作为 `Edge` 的实现了 `AGraph trait` 的类型：

```
trait AGraph {
type Node;
type Edge;
}
#
fn traverse(graph: &AGraph<Node=usize, Edge=(usize, usize)>) {}
```

虽然 `trait` 对象意味着无需在编译时就知道 `graph` 参数的具体类型，但是我们确实需要在 `traverse` 函数中通过具体的关联类型来限制 `AGraph trait` 的使用。如果不提供这样的限制，`Rust` 将不能计算出用哪个 `impl` 来匹配这个 `trait` 对象，因为关联类型可以作为方法签名的一部分，`Rust` 需要在虚函数表(vtable)中查找它们。

## 运算符重载和默认类型参数

`<PlaceholderType=ConcreteType>` 语法也可以以另一种方式使用：用来指定泛型的默认类型。这种情况的一个非常好的例子是用于运算符重载。

`Rust` 并不允许创建自定义运算符或重载任意运算符，不过 `std::ops` 中所列出的运算符和相应的 `trait` 可以通过实现运算符相关 `trait` 来重载。例如，列表 19-25 中展示了如何在 `Point` 结构体上实现 `Add trait` 来重载 `+` 运算符，这样就可以将两个 `Point` 实例相加了：

文件名: `src/main.rs`

```
use std::ops::Add;

#[derive(Debug, PartialEq)]
struct Point {
 x: i32,
 y: i32,
}

impl Add for Point {
 type Output = Point;

 fn add(self, other: Point) -> Point {
 Point {
 x: self.x + other.x,
 y: self.y + other.y,
 }
 }
}

fn main() {
 assert_eq!(Point { x: 1, y: 0 } + Point { x: 2, y: 3 },

```

```

 Point { x: 3, y: 3 });
 }

```

列表 19-25: 实现 `Add` 来重载 `Point` 的 `+` 运算符

这里实现了 `add` 方法将两个 `Point` 实例的 `x` 值和 `y` 值分别相加来创建一个新的 `Point`。`Add trait` 有一个叫做 `Output` 的关联类型，它用来决定 `add` 方法的返回值类型。

让我们更仔细的看看 `Add trait`。这里是其定义：

```

trait Add<RHS=Self> {
 type Output;

 fn add(self, rhs: RHS) -> Self::Output;
}

```

这看来应该很熟悉；这是一个带有一个方法和一个关联类型的 `trait`。比较陌生的部分是尖括号中的 `RHS=Self`：这个语法叫做默认类型参数（*default type parameters*）。`RHS` 是一个泛型参数（“right hand side”的缩写），它用于 `add` 方法中的 `rhs` 参数。如果实现 `Add trait` 时不指定 `RHS` 的具体类型，`RHS` 的类型将是默认的 `Self` 类型（在其上实现 `Add` 的类型）。

让我们看看另一个实现了 `Add trait` 的例子。想象一下我们拥有两个存放不同的单元值的结构体，`Millimeters` 和 `Meters`。可以如列表 19-26 所示那样用不同的方式为 `Millimeters` 实现 `Add trait`：

```

use std::ops::Add;

struct Millimeters(u32);
struct Meters(u32);

impl Add for Millimeters {
 type Output = Millimeters;

 fn add(self, other: Millimeters) -> Millimeters {
 Millimeters(self.0 + other.0)
 }
}

impl Add<Meters> for Millimeters {
 type Output = Millimeters;

 fn add(self, other: Meters) -> Millimeters {
 Millimeters(self.0 + (other.0 * 1000))
 }
}

```

列表 19-26: 在 `Millimeters` 上实现 `Add`，以能够将 `Millimeters` 与 `Millimeters` 相加和将 `Millimeters` 与 `Meters` 相加

如果将 `Millimeters` 与其他 `Millimeters` 相加，则无需为 `Add` 参数化 `RHS` 类型，因为默认的 `Self` 正是我们希望的。如果希望实现 `Millimeters` 与 `Meters` 相加，那么需要声明为 `impl Add<Meters>` 来设定 `RHS` 类型参数的值。

默认参数类型主要用于如下两个方面：

1. 扩展类型而不破坏现有代码。
2. 允许以一种大部分用户都不需要的方法进行自定义。



`Add trait` 就是第二个目的的一个例子：大部分时候你会将两个相似的类型相加。在 `Add trait` 定义中使用默认类型参数使得实现 `trait` 变得更容易，因为大部分时候无需指定这额外的参数。换句话说，这样就去掉了一些实现的样板代码。

第一个目的是相似的，但过程是反过来的：因为现有 `trait` 实现并没有指定类型参数，如果需要为现有 `trait` 增加类型参数，为其提供一个默认值将允许我们在不破坏现有实现代码的基础上扩展 `trait` 的功能。

## 完全限定语法与消歧义

`Rust` 既不能避免一个 `trait` 与另一个 `trait` 拥有相同名称的方法，也不能阻止为同一类型同时实现这两个 `trait`。甚至也可以直接在类型上实现相同名称的方法！那么为了能使用相同的名称调用每一个方法，需要告诉 `Rust` 我们希望使用哪个方法。考虑一下列表 19-27 中的代码，`trait Foo` 和 `Bar` 都拥有方法 `f`，并在结构体 `Baz` 上实现了这两个 `trait`，结构体也有一个叫做 `f` 的方法：

文件名: `src/main.rs`

```
trait Foo {
 fn f(&self);
}

trait Bar {
 fn f(&self);
}

struct Baz;

impl Foo for Baz {
 fn f(&self) { println!("Baz's impl of Foo"); }
}

impl Bar for Baz {
 fn f(&self) { println!("Baz's impl of Bar"); }
}

impl Baz {
 fn f(&self) { println!("Baz's impl"); }
}

fn main() {
 let b = Baz;
 b.f();
}
```

列表 19-27：实现两个拥有相同名称的方法的 `trait`，同时还有直接定义于结构体的（同名）方法

对于 `Baz` 的 `Foo trait` 中方法 `f` 的实现，它打印出 `Baz's impl of Foo`。对于 `Baz` 的 `Bar trait` 中方法 `f` 的实现，它打印出 `Baz's impl of Bar`。直接定义于 `Baz` 的 `f` 实现打印出 `Baz's impl`。当调用 `b.f()` 时会发生什么呢？在这个例子中，`Rust` 总是会使用直接定义于 `Baz` 的实现并打印出 `Baz's impl`。

为了能够调用 `Foo` 和 `Baz` 中的 `f` 方法而不是直接定义于 `Baz` 的 `f` 实现，则需要使用完全限定语法（*fully qualified syntax*）来调用方法。它像这样工作：对于任何类似如下的方法调用：

```
receiver.method(args);
```

可以像这样使用完全限定的方法调用：

```
<Type as Trait>::method(receiver, args);
```

所以为了消歧义并能够调用列表 19-27 中所有的 `f` 方法，需要在尖括号中指定每个希望 `Baz` 作为的 `trait`，接着使用双冒号，接着传递 `Baz` 实例作为第一个参数并调用 `f` 方法。列表 19-28 展示了如何调用 `Foo` 中的 `f`，和 `Bar` 中与 `b` 中的 `f`：

文件名: `src/main.rs`

```
trait Foo {
fn f(&self);
}
trait Bar {
fn f(&self);
}
struct Baz;
impl Foo for Baz {
fn f(&self) { println!("Baz's impl of Foo"); }
}
impl Bar for Baz {
fn f(&self) { println!("Baz's impl of Bar"); }
}
impl Baz {
fn f(&self) { println!("Baz's impl"); }
}
#
fn main() {
 let b = Baz;
 b.f();
 <Baz as Foo>::f(&b);
 <Baz as Bar>::f(&b);
}
```

列表 19-28: 使用完全限定语法调用作为 `Foo` 和 `Bar` `trait` 一部分的 `f` 方法

这会打印出：

```
Baz's impl
Baz's impl of Foo
Baz's impl of Bar
```

只在存在歧义时才需要 `Type as` 部分，只有需要 `Type as` 时才需要 `<>` 部分。所以如果在作用域中只有定义于 `Baz` 和 `Baz` 上实现的 `Foo` `trait` 的 `f` 方法的话，则可以使用 `Foo::f(&b)` 调用 `Foo` 中的 `f` 方法，因为无需与 `Bar` `trait` 相区别。

也可以使用 `Baz::f(&b)` 调用直接定义于 `Baz` 上的 `f` 方法，不过因为这个定义是在调用 `b.f()` 时默认使用的，并不要求调用此方法时使用完全限定的名称。

## 父 `trait` 用于在另一个 `trait` 中使用某 `trait` 的功能

有时我们希望当实现某 `trait` 时依赖另一个 `trait` 也被实现，如此这个 `trait` 就可以使用其他 `trait` 的功能。这个所需的 `trait` 是我们实现的 `trait` 的父（超）`trait`（*supertrait*）。

例如，加入我们希望创建一个带有 `outline_print` 方法的 `trait OutlinePrint`，它会打印出带有星号框的值。也就是说，如果 `Point` 实现了 `Display` 并返回 `(x, y)`，调用以 1 作为 `x` 和 3 作为 `y` 的 `Point` 实例的 `outline_print` 会显示如下：

```

* *
* (1, 3) *
* *
```

```

```

在 `outline_print` 的实现中，因为希望能够使用 `Display` `trait` 的功能，则需要说明 `OutlinePrint` 只能用于同时也实现了 `Display` 并提供了 `OutlinePrint` 需要的功能的类型。可以在 `trait` 定义中指定 `OutlinePrint: Display` 来做到这一点。这类似于为 `trait` 增加 `trait bound`。列表 19-29 展示了一个 `OutlinePrint` `trait` 的实现：

```
use std::fmt;

trait OutlinePrint: fmt::Display {
 fn outline_print(&self) {
 let output = self.to_string();
 let len = output.len();
 println!("{}", ".repeat(len + 4));
 println!("{}", " ".repeat(len + 2));
 println!("{}", " {} ", output);
 println!("{}", " ".repeat(len + 2));
 println!("{}", ".repeat(len + 4));
 }
}
```

列表 19-29：实现 `OutlinePrint` `trait`，它要求来自 `Display` 的功能

因为指定了 `OutlinePrint` 需要 `Display` `trait`，则可以在 `outline_print` 中使用 `to_string`（`to_string` 会为任何实现 `Display` 的类型自动实现）。如果不在 `trait` 名后增加 `: Display` 并尝试在 `outline_print` 中使用 `to_string`，则会得到一个错误说在当前作用域中没有找到用于 `&Self` 类型的方法 `to_string`。

如果尝试在一个没有实现 `Display` 的类型上实现 `OutlinePrint`，比如 `Point` 结构体：

```
trait OutlinePrint {}
struct Point {
 x: i32,
 y: i32,
}

impl OutlinePrint for Point {}
```

则会得到一个错误说 `Display` 没有被实现而 `Display` 被 `OutlinePrint` 所需要：

```
error[E0277]: the trait bound `Point: std::fmt::Display` is not satisfied
--> src/main.rs:20:6
|
20 | impl OutlinePrint for Point {}
| ^^^^^^^^^^^^^ the trait `std::fmt::Display` is not implemented for
| `Point`
|
= note: `Point` cannot be formatted with the default formatter; try using
`:?` instead if you are using a format string
= note: required by `OutlinePrint`
```

一旦在 `Point` 上实现 `Display` 并满足 `OutlinePrint` 要求的限制，比如这样：

```
struct Point {
x: i32,
y: i32,
}
#
use std::fmt;
```

```
impl fmt::Display for Point {
 fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
 write!(f, "({}, {})", self.x, self.y)
 }
}
```

那么在 `Point` 实现 `OutlinePrint` `trait` 将能成功编译并可以在 `Point` 实例上调用 `outline_print` 来显示位于星号框中的点的值。

## newtype 模式用以在外部类型上实现外部 `trait`

在第十章中，我们提到了孤儿规则（`orphan rule`），它说明只要 `trait` 或类型对于当前 `crate` 是本地的话就可以在此类型上实现该 `trait`。一个绕开这个限制的方法是使用 **newtype** 模式（*newtype pattern*），它涉及到使用一个元组结构体来创建一个新类型，它带有一个字段作为希望实现 `trait` 的类型的简单封装。接着这个封装类型对于 `crate` 是本地的，这样就可以在这个封装上实现 `trait`。“Newtype”是一个源自 `Haskell` 编程语言的概念。使用这个模式没有运行时性能惩罚。这个封装类型在编译时被省略了。

例如，如果想要在 `Vec` 上实现 `Display`，可以创建一个包含 `Vec` 实例的 `Wrapper` 结构体。接着可以如列表 19-30 那样在 `Wrapper` 上实现 `Display` 并使用 `Vec` 的值：

文件名: `src/main.rs`

```
use std::fmt;

struct Wrapper(Vec<String>);

impl fmt::Display for Wrapper {
 fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
 write!(f, "[{}]", self.0.join(", "))
 }
}

fn main() {
 let w = Wrapper(vec![String::from("hello"), String::from("world")]);
 println!("w = {}", w);
}
```

列表 19-30：创建 `Wrapper` 类型封装 `Vec<String>` 以便实现 `Display`

`Display` 的实现使用 `self.0` 来访问其内部的 `Vec`，接着就可以使用 `Wrapper` 中 `Display` 的功能了。

此方法的缺点是因为 `Wrapper` 是一个新类型，它没有定义于其值之上的方法；必须直接在 `Wrapper` 上实现 `Vec` 的所有方法，如 `push`、`pop` 等等，并代理到 `self.0` 上以便可以将 `Wrapper` 完全当作 `Vec` 处理。如果希望新类型拥有其内部类型的每一个方法，为封装类型实现第十五章讲到的 `Deref` `trait` 并返回其内部类型是一种解决方案。如果不希望封装类型拥有所有内部类型的方法，比如为了限制封装类型的行为，则必须自行实现所需的方法。

上面便是 **newtype** 模式如何与 `trait` 结合使用的；还有一个不涉及 `trait` 的实用模式。现在让我们将话题的焦点转移到一些与 `Rust` 类型系统交互的高级方法上来吧。

# 高级类型

ch19-04-advanced-types.md

commit e084e1773667c8eae28d9aab6d4939348eec0092

Rust 的类型系统有一些我们曾经提到或用到但没有讨论过的功能。我们从有关 `trait` 的 `newtype` 模式开始讨论：首先从一个关于为什么 `newtype` 与类型一样有用的更宽泛的讨论开始。接着会转向类型别名（`type aliases`），一个类似于 `newtype` 但有着稍微不同的语义的功能。我们还会讨论 `!` 类型和动态大小类型。

## 为了类型安全和抽象而使用 `newtype` 模式

在“高级 `trait`”部分最后开始的 `newtype` 模式的讨论中，我们以一个包含一个封装了某类型的字段的元组结构体创建了一个新类型，这对于静态的确保其值不被混淆也是有帮助的，并且它经常用来表示一个值的单元。实际上列表 19-26 中已有一个例子：`Millimeters` 和 `Meters` 结构体都将 `u32` 值封装进了新类型。如果编写了一个有 `Millimeters` 类型参数的函数，不小心使用 `Meters` 或普通的 `u32` 值来调用该函数的程序是不能编译的。

另一个使用 `newtype` 模式的原因是用来抽象掉一些类型的实现细节：例如，封装类型可以暴露出与直接使用其内部私有类型时所不同的 `API`，以便限制其功能。新类型也可以隐藏其内部的泛型类型。例如，可以提供了一个封装了 `HashMap<i32, String>` 的 `People` 类型，用来储存人名以及相应的 `ID`。使用 `People` 的代码只需与提供的公有 `API` 交互即可，比如向 `People` 集合增加名字字符串的方法，这样这些代码就无需知道在内部我们将一个 `i32` `ID` 赋予了这个名字了。`newtype` 模式是一种实现第十七章所讨论的隐藏实现细节的封装的轻量级方法。

## 类型别名用来创建同义类型

`newtype` 模式涉及到创建新结构体来作为新的、单独的类型。Rust 还提供了声明类型别名（`type alias`）的能力，使用 `type` 关键字来给予现有类型另一个名字。例如，可以像这样创建 `i32` 的别名 `Kilometers`：

```
type Kilometers = i32;
```

这意味着 `Kilometers` 是 `i32` 的同义词（*synonym*）；不同于列表 19-26 中创建的 `Millimeters` 和 `Meters` 类型。`Kilometers` 不是一个新的、单独的类型。`Kilometers` 类型的值将被完全当作 `i32` 类型值来对待：

```
type Kilometers = i32;

let x: i32 = 5;
let y: Kilometers = 5;

println!("x + y = {}", x + y);
```

因为 `Kilometers` 是 `i32` 的别名，他们是同一类型。可以将 `i32` 与 `Kilometers` 相加，可以将 `Kilometers` 传递给获取 `i32` 参数的函数。但无法获得上一部分讨论的 `newtype` 模式所提供的类型检查的好处。

类型别名的主要用途是减少重复。例如，可能会有这样很长的类型：

```
Box<FnOnce() + Send + 'static>
```

在函数签名或类型注解中每次都书写这个类型将是枯燥且易于出错的。想象一下如列表 19-31 这样全是如此代码的项目：

```
let f: Box<FnOnce() + Send + 'static> = Box::new(|| println!("hi"));
```

```
fn takes_long_type(f: Box<FnOnce() + Send + 'static>) {
 // ...
}

fn returns_long_type() -> Box<FnOnce() + Send + 'static> {
 // ...
 # Box::new(|| ())
}
```

列表 19-31：在大部分地方使用名称很长的类型

类型别名通过减少项目中重复代码的数量来使其更加易于控制。这里我们为这个冗长的类型引入了一个叫做 `Thunk` 的别名，这样就可以如列表 19-32 所示将所有使用这个类型的地方替换为更短的 `Thunk`：

```
type Thunk = Box<FnOnce() + Send + 'static>;

let f: Thunk = Box::new(|| println!("hi"));

fn takes_long_type(f: Thunk) {
 // ...
}

fn returns_long_type() -> Thunk {
 // ...
 # Box::new(|| ())
}
```

列表 19-32：引入类型别名 `Thunk` 来减少重复

这样就读写起来就容易多了！为类型别名选择一个好名字也可以帮助你表达意图（单词 *think* 表示会在之后被计算的代码，所以这是一个存放闭包的合适的名字）。

类型别名的另一个常用用法是与 `Result<T, E>` 结合。考虑一下标准库中的 `std::io` 模块。I/O 操作通常会返回一个 `Result<T, E>`，因为这些操作可能会失败。`std::io::Error` 结构体代表了所有可能的 I/O 错误。`std::io` 中大部分函数会返回 `Result<T, E>`，其中 `E` 是 `std::io::Error`，比如 `Write trait` 中的这些函数：

```
use std::io::Error;
use std::fmt;

pub trait Write {
 fn write(&mut self, buf: &[u8]) -> Result<usize, Error>;
 fn flush(&mut self) -> Result<(), Error>;

 fn write_all(&mut self, buf: &[u8]) -> Result<(), Error>;
 fn write_fmt(&mut self, fmt: fmt::Arguments) -> Result<(), Error>;
}
```

这里出现了很多的 `Result<..., Error>`。为此，`std::io` 有这个类型别名声明：

```
type Result<T> = Result<T, std::io::Error>;
```

因为这位位于 `std::io` 中，可用的完全限定的别名是 `std::io::Result<T>`；也就是说，`Result<T, E>` 中 `E` 放入了 `std::io::Error`。 `Write trait` 中的函数最终看起来像这样：

```
pub trait Write {
 fn write(&mut self, buf: &[u8]) -> Result<usize>;
 fn flush(&mut self) -> Result<()>;
}
```

```
fn write_all(&mut self, buf: &[u8]) -> Result<>>;
fn write_fmt(&mut self, fmt: Arguments) -> Result<>>;
}
```

类型别名在两个方面有帮助：易于编写并在整个 `std::io` 中提供了一致的接口。因为这是一个别名，它只是另一个 `Result<T, E>`，这意味着可以在其上使用 `Result<T, E>` 的任何方法，以及像 `?` 这样的特殊语法。

## 从不返回的 `!`，`never type`

Rust 有一个叫做 `!` 的特殊类型。在类型理论术语中，它被称为 *empty type*，因为它没有值。我们更倾向于称之为 *never type*。这个名字描述了它的作用：在函数从不返回的时候充当返回值。例如：

```
fn bar() -> ! {
```

这读作“函数 `bar` 从不返回”，而从不返回的函数被称为发散函数（*diverging functions*）。不能创建 `!` 类型的值，所以 `bar` 也不可能返回。一个不能创建值的类型有什么用呢？如果你回想一下第二章，曾经有一些看起来像这样的代码，如列表 19-33 所重现的：

```
let guess = "3";
loop {
let guess: u32 = match guess.trim().parse() {
 Ok(num) => num,
 Err(_) => continue,
};
break;
}
```

列表 19-33： `match` 语句和一个以 `continue` 结束的分支

当时我们忽略了一些代码细节。在第六章中，我们学习了 `match` 的分支必须返回相同的类型。如下代码不能工作：

```
let guess = match guess.trim().parse() {
 Ok(_) => 5,
 Err(_) => "hello",
}
```

这里的 `guess` 会是什么类型呢？它必须既是整型也是字符串，而 Rust 要求 `guess` 只能是一个类型。那么 `continue` 返回了什么呢？为什么列表 19-33 中会允许一个分支返回 `u32` 而另一个分支却以 `continue` 结束呢？

正如你可能猜到的，`continue` 的值是 `!`。也就是说，当 Rust 要计算 `guess` 的类型时，它查看这两个分支。前者是 `u32` 值，而后者是 `!` 值。因为 `!` 并没有一个值，Rust 认为这是可行的，并决定 `guess` 的类型是 `u32`。描述 `!` 的行为的正式方式是 *never type* 可以与其他任何类型联合。允许 `match` 的分支以 `continue` 结束是因为 `continue` 并不真正返回一个值；相反它把控制权交回上层循环，所以在 `Err` 的情况，事实上并未对 `guess` 赋值。

*never type* 的另一个用途是 `panic!`。还记得 `Option<T>` 上的 `unwrap` 函数吗？它产生一个值或 `panic`。这里是它的定义：

```
impl<T> Option<T> {
 pub fn unwrap(self) -> T {
 match self {
 Some(val) => val,
 None => panic!("called `Option::unwrap()` on a `None` value"),
 }
 }
}
```

```
 }
 }
}
```

这里与列表 19-33 中的 `match` 发生的相同的情况：我们知道 `val` 是 `T` 类型，`panic!` 是 `!` 类型，所以整个 `match` 表达式的结果是 `T` 类型。这能工作是因为 `panic!` 并不产生一个值：它终止程序。对于 `None` 的情况，`unwrap` 并不返回一个值，所以这些代码是有效。

最后的表达式在 `loop` 中使用了 `!` 类型：

```
print!("forever ");

loop {
 print!("and ever ");
}
```

这里，循环永远也不结束，所以此表达式的值是 `!`。但是如果引入 `break` 这就不为真了，因为循环在执行到 `break` 后就会终止。

## 动态大小类型和 `Sized` trait

因为 `Rust` 需要知道类似内存布局之类的信息，在其类型系统的一个特定的角落可能令人迷惑，这就是动态大小类型（*dynamically sized types*）的概念。这有时被称为“DST”或“unsized types”，这些类型允许我们处理只有在运行时才知道大小的类型。

让我们深入研究一个贯穿本书都在使用的动态大小类型的细节：`str`。没错，不是 `&str`，而是 `str` 本身。`str` 是一个 DST；直到运行时我们都不知道字符串有多长。因为不能知道大小，也就不能创建 `str` 类型的变量，也不能获取 `str` 类型的参数。考虑一下这些代码，他们不能工作：

```
let s1: str = "Hello there!";
let s2: str = "How's it going?";
```

这两个 `str` 值需要有完全一样的内存布局，不过他们却有不同长度：`s1` 需要 12 字节来存储，而 `s2` 需要 15 字节。这样就是为什么不可能创建一个存放动态大小类型的变量。

那么该怎么办呢？好吧，在这个例子中你已经知道了答案：`s1` 和 `s2` 的类型是 `&str` 而不是 `str`。如果你回想第四章，我们这样描述 `&str`：

... 这是一个字符串内部位置和其引用的元素的数量引用。

所以虽然 `&T` 是一个储存了 `T` 所在的内存位置的单个值，`&str` 则是两个值：`str` 的地址和其长度。这样，`&str` 就有了一个在编译时可以知道的大小：它是 `usize` 长度的两倍。也就是说，我们总是知道 `&str` 的大小，而无论其引用的字符串是多长。这里是 `Rust` 中动态大小类型的常规用法：他们有一些额外的元信息来储存动态信息的大小。这引出了动态大小类型的黄金规则：必须将动态大小类型的值置于某种指针之后。

虽然我们总是说 `&str`，但是可以将 `str` 与所有类型的指针结合：比如 `Box<str>` 或 `Rc<str>`。事实上，之前已经见过了，不过是另一个动态大小类型：`trait`。每一个 `trait` 都是一个可以通过 `trait` 名称来引用的动态大小类型。在第十七章中，我们提到了为了将 `trait` 用于 `trait` 对象，必须将他们放入指针之后，比如 `&Trait` 或 `Box<Trait>`（`Rc<Trait>` 也可以）。

### Sized trait



为了处理 DST，Rust 有一个 `trait` 来决定一个类型的大小是否在编译时可知，这就是 `Sized`。这个 `trait` 自动为编译器在编译时就知道大小的类型实现。另外，Rust 隐式的为每一个泛型函数增加了 `Sized` bound。也就是说，对于如下泛型函数定义：

```
fn generic<T>(t: T) {
```

实际上被当作如下处理：

```
fn generic<T: Sized>(t: T) {
```

泛型函数默认只能用于在编译时已知大小的类型。然而可以使用如下特殊语法来放松这个限制：

```
fn generic<T: ?Sized>(t: &T) {
```

`?Sized` `trait bound` 与 `Sized` 相对；也就是说，它可以读作“`T` 可能是也可能不是 `Sized` 的”。这个语法只能用于 `Sized`，而不是其他 `trait`。

另外注意我们将 `t` 参数的类型从 `T` 变为了 `&T`：因为其类型可能不是 `Sized` 的，所以需要将其置于某种指针之后。在这个例子中选择了引用。

接下来，让我们讨论一下函数和闭包！

# 高级函数与闭包

ch19-05-advanced-functions-and-closures.md

commit d06a6a181fd61704cbf7feb55bc61d518c6469f9

最后让我们讨论一些有关函数和闭包的高级功能：函数指针、发散函数和返回值闭包。

## 函数指针

我们讨论过了如何向函数传递闭包，不过也可以向函数传递常规的函数！函数的类型是 `fn`，使用小写的“**f**”以便不与 `Fn` 闭包 `trait` 向混淆。`fn` 被称为函数指针（*function pointer*）。指定参数为函数指针的语法类似于闭包，如列表 19-34 所示：

文件名: src/main.rs

```
fn add_one(x: i32) -> i32 {
 x + 1
}

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
 f(arg) + f(arg)
}

fn main() {
 let answer = do_twice(add_one, 5);

 println!("The answer is: {}", answer);
}
```

列表 19-34：使用 `fn` 类型接受函数指针作为参数

这会打印出 `The answer is: 12`。`do_twice` 中的 `f` 被指定为一个接受一个 `i32` 参数并返回 `i32` 的 `fn`。接着就可以在 `do_twice` 函数体中调用 `f`。在 `main` 中，可以将函数名 `add_one` 作为第一个参数传递给 `do_twice`。

不同于闭包，`fn` 是一个类型而不是一个 `trait`，所以直接指定 `fn` 作为参数而不是声明一个带有 `Fn` 作为 `trait bound` 的泛型参数。

函数指针实现了所有三个闭包 `trait`（`Fn`、`FnMut` 和 `FnOnce`），所以总是可以在调用期望闭包的函数时传递函数指针作为参数。倾向于编写使用泛型和闭包 `trait` 的函数，这样它就能接受函数或闭包作为参数。一个只期望接受 `fn` 的情况的例子是与不存在闭包的外部代码交互时：`C` 语言的函数可以接受函数作为参数，但没有闭包。

比如，如果希望使用 `map` 函数将一个数字 `vector` 转换为一个字符串 `vector`，就可以使用闭包：

```
let list_of_numbers = vec![1, 2, 3];
let list_of_strings: Vec<String> = list_of_numbers
 .iter()
 .map(|i| i.to_string())
 .collect();
```

或者可以将函数作为 `map` 的参数来代替闭包：

```
let list_of_numbers = vec![1, 2, 3];
let list_of_strings: Vec<String> = list_of_numbers
 .iter()
 .map(ToString::to_string)
```

```
.collect();
```

注意这里必须使用“高级 **trait**”部分讲到的完全限定语法，因为存在多个叫做 `to_string` 的函数；这里使用定义于 `ToString` **trait** 的 `to_string` 函数，标准库为所有实现了 `Display` 的类型实现了这个 **trait**。

一些人倾向于函数风格，一些人喜欢闭包。他们最终都会产生同样的代码，所以请使用你更明白的吧。

## 返回闭包

因为闭包以 **trait** 的形式体现，返回闭包就有点微妙了；不能直接这么做。对于大部分需要返回 **trait** 的情况，可以使用是实现了期望返回的 **trait** 的具体类型替代函数的返回值。但是这不能用于闭包。他们没有一个可返回的具体类型；例如不允许使用函数指针 `fn` 作为返回值类型。

这段代码尝试直接返回闭包，它并不能编译：

```
fn returns_closure() -> Fn(i32) -> i32 {
 |x| x + 1
}
```

编译器给出的错误是：

```
error[E0277]: the trait bound `std::ops::Fn(i32) -> i32 + 'static':
std::marker::Sized` is not satisfied
--> <anon>:2:25
|
2 | fn returns_closure() -> Fn(i32) -> i32 {
| ^^^^^^^^^^^^^^^^^ the trait `std::marker::Sized` is
not implemented for `std::ops::Fn(i32) -> i32 + 'static`
|
= note: `std::ops::Fn(i32) -> i32 + 'static` does not have a constant size
known at compile-time
= note: the return type of a function must have a statically known size
```

又是 **Sized trait**！**Rust** 并不知道需要多少空间来储存闭包。不过我们在上一部分见过这种情况的解决办法：可以使用 **trait** 对象：

```
fn returns_closure() -> Box<Fn(i32) -> i32> {
 Box::new(|x| x + 1)
}
```

关于 **trait** 对象的更多内容，请参考第十八章。

## 总结

好的！现在我们学习了 **Rust** 并不常用但你可能用得着的功能。我们介绍了很多复杂的主题，这样当你在错误信息提示或阅读他人代码时遇到他们时，至少可以说已经见过这些概念和语法了。

现在，让我们再开始一个项目，将本书所学的所有内容付与实践！



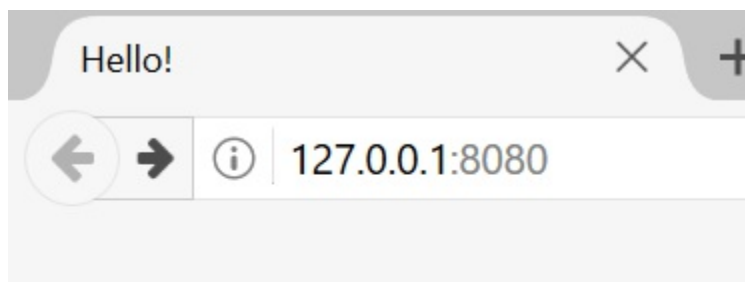
## 最后的项目：构建多线程 **web server**

[ch20-00-final-project-a-web-server.md](#)

commit 08e50d5e147ad290d88efd5c58365000723626df

这是一次漫长的旅途，不过我们做到了！这一章便是本书的结束。离别是如此甜蜜的悲伤。不过在我们结束之前，再来一起构建另一个项目，来展示最后几章所学，同时复习更早的章节。

下面就是我们将要做的：一个简单的 **web server**：



# Hello!

## Hi from Rust

为此我们将：

1. 学习一些 **TCP** 与 **HTTP** 知识
2. 在套接字 (**socket**) 上监听 **TCP** 请求
3. 解析少量的 **HTTP** 请求
4. 创建一个合适的 **HTTP** 响应
5. 通过线程池改善 **server** 的吞吐量

在开始之前，需要提到一点：如果你曾在生产环境中编写过这样的代码，还有很多更好的做法。特别需要指出的是，**crates.io** 上提供了很多更完整健壮的 **web server** 和 线程池实现，要比我们编写的好很多。

然而，本章的目的在于学习，而不是走捷径。因为 **Rust** 是一个系统编程语言，能够选择处理什么层次的抽象。我们能够选择比其他语言可能或可用的层次更低的层次。所以我们将自己编写一个基础的 **HTTP server** 和线程池，以便学习将来可能用到的 **crate** 背后的通用理念和技术。

# 单线程 web server

[ch20-01-single-threaded.md](#)

commit 2e269ff82193fd65df8a87c06561d74b51ac02f7

首先让我们创建一个可运行的单线程 **web server**。我们将处理 **TCP** 和 **HTTP** 请求和响应的原始字节来从 **server** 向浏览器发送 **HTML**。首先快速了解一下涉及到的协议。

超文本传输协议（*Hypertext Transfer Protocol*, **HTTP**）驱动着现在的互联网，它构建于 传输控制协议（*Transmission Control Protocol*, **TCP**）的基础上。这里并不会过多的涉及细节，只做简单的概括：**TCP** 是一个底层协议，**HTTP** 是 **TCP** 之上的高级协议。这两个都是一种被称为 请求-响应协议（*request-response protocol*）的协议，也就是说，有 客户端（*client*）来初始化请求，并有 服务端（*server*）监听请求并向客户端提供响应。请求与响应的内容由协议本身定义。

**TCP** 描述了信息如何从一个 **server** 到另一个的底层细节，不过它并不指定信息是什么；它仅仅是一堆 0 和 1。**HTTP** 构建于 **TCP** 之上，它定义了请求和响应的内容。为此，技术上讲可将 **HTTP** 用于其他协议之上，不过对于绝大部分情况，**HTTP** 通过 **TCP** 传输。

所以我们的 **web server** 所需做的第一件事便是能够监听 **TCP** 连接。标准库有 `std::net` 模块处理这些功能。现在创建一个新项目：

```
$ cargo new hello --bin
 Created binary (application) `hello` project
$ cd hello
```

并在 `src/main.rs` 放入列表 20-1 中的代码作为开始。这段代码会在地址 `127.0.0.1:8080` 上监听传入的 **TCP** 流。当获取到传入的流，它会打印出 `Connection established!`：

文件名: `src/main.rs`

```
use std::net::TcpListener;

fn main() {
 let listener = TcpListener::bind("127.0.0.1:8080").unwrap();

 for stream in listener.incoming() {
 let stream = stream.unwrap();

 println!("Connection established!");
 }
}
```

列表 20-1：监听传入的流并在接收到流时打印信息

`TcpListener` 用于监听 **TCP** 连接。我们选择监听地址 `127.0.0.1:8080`。冒号之前的部分是一个代表本机的 **IP** 地址，而 `8080` 是端口。选择这个端口是因为通常 **HTTP** 监听 80 端口，不过连接 80 端口需要管理员权限。普通用户可以监听大于 1024 的端口；8080 端口易于记忆因为它重复了 **HTTP** 的 80 端口两次。

`bind` 函数类似于 `new` 因为它返回一个新的 `TcpListener` 实例，不过 `bind` 是一个符合这个领域术语的描述性名称。在网络领域，人们通常说“绑定到一个端口”（“binding to a port”），所以标准库中将创建新 `TcpListener` 的函数定义为 `bind`。

`bind` 函数返回 `Result<T, E>`。绑定可能会失败，例如，如果不是管理员尝试连接 `80` 端口。另一个绑定会失败的情况是两个程序监听相同的端口，这可能发生于运行两个本程序的实例时。因为我们编写的是一个基础的 `server`，将不会担心处理这类错误，`unwrap` 使得出现这些情况时直接停止程序。

`TcpListener` 的 `incoming` 方法返回一个迭代器，它提供了一系列的流（更准确的说是 `TcpStream` 类型的流）。流（*stream*）代表一个客户端和服务端之间打开的连接。连接（*connection*）代表客户端连接服务端、服务端生成响应以及服务端关闭连接的全部请求 / 响应过程。为此，`TcpStream` 允许我们读取它来查看客户端发送了什么，并可以编写响应。所以这个 `for` 循环会依次处理每个连接并产生一系列的流供我们处理。

目前为止，处理流意味着调用 `unwrap` 在出现任何错误时终止程序，接着打印信息。因为我们实际上没有遍历连接，而是遍历 连接尝试（*connection attempts*），所以可能出现错误。连接可能会因为很多原因不能成功，大部分是操作系统相关的。例如，很多系统限制同时打开的连接数；新连接尝试产生错误，直到一些打开的连接关闭为止。

让我们试试这段代码！首先在终端执行 `cargo run`，接着在浏览器中加载 `127.0.0.1:8080`。浏览器会显示出看起来类似于“连接重置”（“**Connection reset**”）的错误信息，因为目前并没响应任何数据。但是如果我们观察终端，会发现当浏览器连接 `server` 时会打印出一系列的信息！

```
Running `target/debug/hello`
Connection established!
Connection established!
Connection established!
```

对于一次浏览器请求会打印出多条信息；这些连接可能是浏览器请求页面和请求出现在浏览器 `tab` 页中的 `favicon.ico`，或者浏览器可能重试了连接。浏览器期望使用 `HTTP` 交互，不过我们并未回复任何内容。当 `stream` 在循环的结尾离开作用域并被丢弃，其连接将被关闭，作为 `TcpStream` 的 `drop` 实现的一部分。浏览器有时通过重连来处理关闭的连接，因为这些问题可能是暂时的。现在重要的是我们成功的处理了 `TCP` 连接！

记得当运行完特定版本的代码后使用 `ctrl-C` 来停止程序，并在做出最新的代码修改之后执行 `cargo run` 重启服务。

## 读取请求

下面读取浏览器的请求！因为我们增加了出于处理连接目的的功能。开始一个新函数来将设置 `server` 和连接，与处理每个请求分离，践行关注分离原则。在这个新的 `handle_connection` 函数中，从 `stream` 中读取数据并打印出来以便观察浏览器发送过来的数据。将代码修改为如列表 20-2 所示：

文件名: `src/main.rs`

```
use std::io::prelude::*;
use std::net::TcpListener;
use std::net::TcpStream;

fn main() {
 let listener = TcpListener::bind("127.0.0.1:8080").unwrap();

 for stream in listener.incoming() {
 let stream = stream.unwrap();

 handle_connection(stream);
 }
}

fn handle_connection(mut stream: TcpStream) {
 let mut buffer = [0; 512];

 stream.read(&mut buffer).unwrap();
```

```
println!("Request: {}", String::from_utf8_lossy(&buffer[..]));
}
```

列表 20-2: 读取 `TcpStream` 并打印数据

在开头增加 `std::io::prelude` 以便将读写流所需的 `trait` 引入作用域。相比在 `main` 的 `for` 中在获取到连接时打印信息，现在调用新的 `handle_connection` 函数并向其传递 `stream`。

在 `handle_connection` 中，通过 `mut` 关键字将 `stream` 参数变为可变。我们将从流中读取数据，所以它需要是可修改的。

接下来，需要实际读取流。这里分两步进行：首先，在栈上声明一个 `buffer` 来存放读取到的数据。这里创建了一个 512 字节的缓冲区，它足以存放基本请求的数据。这对于本章的目的来说是足够的。如果希望处理任意大小的请求，管理所需的缓冲区将更复杂，不过现在一切从简。接着将缓冲区传递给 `stream.read`，它将从 `TcpStream` 中读取字节并放入缓冲区中。

接下来将缓冲区中的字节转换为字符串并打印出来。`String::from_utf8_lossy` 函数获取一个 `&[u8]` 并产生一个 `String`。函数名的“lossy”部分来源于当其遇到无效的 UTF-8 序列时的行为：它使用 `U+FFFD REPLACEMENT CHARACTER`，来代替无效序列。你可能会在缓冲区的剩余部分看到这些替代字符，因为他们没有被请求数据填满。

让我们试一试！启动程序并再次在浏览器中发起请求。注意浏览器中仍然会出现错误页面，不过终端中程序的输出现在看起来像这样：

```
$ cargo run
Compiling hello v0.1.0 (file:///projects/hello)
Finished dev [unoptimized + debuginfo] target(s) in 0.42 secs
Running `target/debug/hello`
Request: GET / HTTP/1.1
Host: 127.0.0.1:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:52.0) Gecko/20100101
Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
█
```

根据使用的浏览器不同可能会出现稍微不同的数据。也可能会看到请求重复出现。现在我们打印出了请求数据，可以通过观察 `Request: GET` 之后的路径来解释为何会从浏览器得到多个连接。如果重复的连接都是请求 `/`，就知道了浏览器尝试重复获取 `/` 因为它没有得到响应。

拆开请求数据来理解浏览器向我们请求了什么。`HTTP` 是一个基于文本的协议，而一个请求有如下格式：

```
Method Request-URI HTTP-Version CRLF
headers CRLF
message-body
```

第一行叫做 请求行 (*request line*)，它存放了客户端请求了什么的信息。请求行的第一部分是 *method*，比如 `GET` 或 `POST`，这描述了客户端如何进行请求。

接着是请求的 *URI*，它代表 统一资源标识符 (*Uniform Resource Identifier*)，*URI* 大体上类似，但也不完全类似于 *URL* (统一资源定位符, *Uniform Resource Locators*)，我们通常将其称为输入到浏览器中的地址。`HTTP` 规范使用术语 *URI*，而 *URI* 和 *URL* 之间的区别对于本章的目的来说并不重要，所以心理上将 *URL* 替换为 *URI* 即可。



接下来，是客户端使用的 HTTP 版本，接着请求行以一个 CRLF 序列结尾。CRLF 序列也可以写作 `\r\n`：`\r` 是回车（*carriage return*）而 `\n` 是换行（*line feed*）。这些术语来自打字机时代！CRLF 序列将请求行与其他请求数据分开。

看看代码打印出的请求行的数据：

```
GET / HTTP/1.1
```

`GET` 是 **method**，`/` 是请求 **URI**，而 `HTTP/1.1` 是版本。

从 `Host:` 开始的其余的行是 **headers**；`GET` 请求没有 **body**。

如果你希望的话，尝试用不同的浏览器发送请求，或请求不同的地址，比如 `127.0.0.1:8080/test`，来观察请求数据如何变化。

现在我们知道了浏览器请求了什么。让我们返回一些数据！

## 编写响应

现在向浏览器返回数据以响应请求。响应有如下格式：

```
HTTP-Version Status-Code Reason-Phrase CRLF
headers CRLF
message-body
```

第一行叫做 **状态行**（*status line*），它包含响应的 HTTP 版本、一个数字状态码用以总结请求的结果和一个描述之前状态码的文本原因。CRLF 序列之后是任意 **header**，另一个 CRLF 序列，和响应的 **body**。

这里是一个使用 HTTP 1.1 版本的响应例子，其状态码为 `200`，原因描述为 `OK`，没有 **header**，也没有 **body**：

```
HTTP/1.1 200 OK\r\n\r\n
```

这些文本是一个微型的成功 HTTP 响应。让我们把这些写入流！去掉打印请求信息的 `println!` 行，并在这里增加如列表 20-3 所示的代码：

文件名: `src/main.rs`

```
use std::io::prelude::*;
use std::net::TcpStream;
fn handle_connection(mut stream: TcpStream) {
 let mut buffer = [0; 512];

 stream.read(&mut buffer).unwrap();

 let response = "HTTP/1.1 200 OK\r\n\r\n";

 stream.write(response.as_bytes()).unwrap();
 stream.flush().unwrap();
}
```

列表 20-3：将一个微型成功 HTTP 响应写入流

新代码中的第一行定义了变量 `response` 来存放将要返回的成功响应的数据。接着，在 `response` 上调用 `as_bytes`，因为 `stream` 的 `write` 方法获取一个 `&[u8]` 并直接将这些字节发送给连接。

`write` 可能会失败，所以 `write` 返回 `Result<T, E>`；我们继续使用 `unwrap` 以继续本章的核心内容而不是处理错误。最后，`flush` 会等待直到所有字节都被写入连接中；`TcpStream` 包含一个内部缓冲区来最小化对底层操作系统的调用。

有了这些修改，运行我们的代码并进行请求！我们不再向终端打印任何数据，所以不会再看到除了 **Cargo** 以外的任何输出。不过当在浏览器中加载 `127.0.0.1:8080` 时，会得到一个空页面而不是错误。太棒了！我们刚刚手写了一个 HTTP 请求与响应。

## 返回真正的 HTML

让我们不只是返回空页面。在项目根目录创建一个新文件，*hello.html*，也就是说，不是在 `src` 目录。在此可以放入任何你期望的 HTML；列表 20-4 展示了本书作者所采用的：

文件名: *hello.html*

```
<!DOCTYPE html>
<html lang="en">
 <head>
 <meta charset="utf-8">
 <title>Hello!</title>
 </head>
 <body>
 <h1>Hello!</h1>
 <p>Hi from Rust</p>
 </body>
</html>
```

列表 20-4：一个简单的 HTML 文件用来作为响应

这是一个极小化的 HTML 5 文档，它有一个标题和一小段文本。如列表 20-5 所示修改 `handle_connection` 来读取 HTML 文件，将其加入到响应的 `body` 中，并发送：

文件名: *src/main.rs*

```
use std::io::prelude::*;
use std::net::TcpStream;
use std::fs::File;

// ...snip...

fn handle_connection(mut stream: TcpStream) {
 let mut buffer = [0; 512];
 stream.read(&mut buffer).unwrap();

 let mut file = File::open("hello.html").unwrap();

 let mut contents = String::new();
 file.read_to_string(&mut contents).unwrap();

 let response = format!("HTTP/1.1 200 OK\r\n\r\n{}", contents);

 stream.write(response.as_bytes()).unwrap();
 stream.flush().unwrap();
}
```

列表 20-5：将 *hello.html* 的内容作为响应 `body` 发送

在开头增加了一行来将标准库中的 `File` 引入作用域，打开和读取文件的代码应该看起来很熟悉，因为第十二章 I/O 项目的列表 12-4 中读取文件内容时出现过类似的代码。

接下来，使用 `format!` 将文件内容加入到将要写入流的成功响应的 `body` 中。

使用 `cargo run` 运行程序，在浏览器加载 `127.0.0.1:8080`，你应该会看到渲染出来的 `HTML` 文件！

注意目前忽略了 `buffer` 中的请求数据并无条件的发送了 `HTML` 文件的内容。尝试在浏览器中请求 `127.0.0.1:8080/something-else` 也会得到同样的 `HTML`。对于所有请求都发送相同的响应其作用是非常有限的，也不是大部分 `server` 所做的：让我们检查请求并只对格式良好（`well-formed`）的请求 `/` 发送 `HTML` 文件。

## 验证请求并有选择的响应

目前我们的 `server` 不管客户端请求什么都会返回相同的 `HTML` 文件。让我们检查浏览器是否请求 `/`，并在其请求其他内容时返回错误。如列表 20-6 所示修改 `handle_connection`，它增加了所需的那部分代码。这一部分将接收到的请求的内容与已知的 `/` 请求的一部分做比较，并增加了 `if` 和 `else` 块来加入处理不同请求的代码：

Filename: `src/main.rs`

```
use std::io::prelude::*;
use std::net::TcpStream;
use std::fs::File;
// ...snip...

fn handle_connection(mut stream: TcpStream) {
 let mut buffer = [0; 512];
 stream.read(&mut buffer).unwrap();

 let get = b"GET / HTTP/1.1\r\n";

 if buffer.starts_with(get) {
 let mut file = File::open("hello.html").unwrap();

 let mut contents = String::new();
 file.read_to_string(&mut contents).unwrap();

 let response = format!("HTTP/1.1 200 OK\r\n\r\n{}", contents);

 stream.write(response.as_bytes()).unwrap();
 stream.flush().unwrap();
 } else {
 // some other request
 };
}
```

列表 20-6：将请求与期望的 `/` 请求内容做匹配，并设置对 `/` 和其他请求的条件化处理

这里在变量 `get` 中硬编码了所需的请求相关的数据。因为我们从缓冲区中读取原始字节，所以使用了字节字符串，使用 `b""` 使得 `get` 也是一个字节字符串。接着检查 `buffer` 是否以 `get` 中的字节开头。如果是，这就是一个格式良好的 `/` 请求，也就是 `if` 块中期望处理的成功情况。`if` 块中包含列表 20-5 中增加的返回 `HTML` 文件内容的代码。

如果 `buffer` 不以 `get` 中的字节开头，就说明是其他请求。对于所有其他请求都将使用 `else` 块中增加的代码来响应。

如果运行代码并请求 `127.0.0.1:8080`，就会得到 `hello.html` 中的 `HTML`。如果进行其他请求，比如 `127.0.0.1:8080/something-else`，则会得到像运行列表 20-1 和 20-2 中代码那样的连接错误。

如列表 20-7 所示向 `else` 块增加代码来返回一个带有 `404` 状态码的响应，这代表了所请求的内容没有找到。接着也会返回一个 `HTML` 向浏览器终端用户表明此意：

Filename: `src/main.rs`

```

use std::io::prelude::*;
use std::net::TcpStream;
use std::fs::File;
fn handle_connection(mut stream: TcpStream) {
if true {
// ...snip...

} else {
 let header = "HTTP/1.1 404 NOT FOUND\r\n\r\n";
 let mut file = File::open("404.html").unwrap();
 let mut contents = String::new();

 file.read_to_string(&mut contents).unwrap();

 let response = format!("{}", header, contents);

 stream.write(response.as_bytes()).unwrap();
 stream.flush().unwrap();
}
}

```

列表 20-7: 对于任何不是 `/` 的请求返回 `404` 状态码的响应和错误页面

这里，响应头有状态码 `404` 和原因短语 `NOT FOUND`。仍然没有任何 `header`，而其 `body` 将是 `404.html` 文件中的 HTML。也在 `hello.html` 同级目录创建 `404.html` 文件作为错误页面；这一次也可以随意使用任何 HTML 或使用列表 20-8 中的示例 HTML：

文件名: `404.html`

```

<!DOCTYPE html>
<html lang="en">
 <head>
 <meta charset="utf-8">
 <title>Hello!</title>
 </head>
 <body>
 <h1>Oops!</h1>
 <p>Sorry, I don't know what you're asking for.</p>
 </body>
</html>

```

列表 20-8: 任何 `404` 响应所返回错误页面内容样例

有了这些修改，再次运行 `server`。请求 `127.0.0.1:8080` 应该会返回 `hello.html`，而对于任何其他请求，比如 `127.0.0.1:8080/foo`，应该会返回 `404.html` 中的错误 HTML！

`if` 和 `else` 块中的代码有很多的重复：他们都读取文件并将其内容写入流。这两个情况唯一的区别是状态行和文件名。将这些区别分别提取到一行 `if` 和 `else` 中，对状态行和文件名变量赋值；然后在读取文件和写入响应的代码中无条件的使用这些变量。重构后代码后的结果如列表 20-9 所示：

文件名: `src/main.rs`

```

use std::io::prelude::*;
use std::net::TcpStream;
use std::fs::File;
// ...snip...

fn handle_connection(mut stream: TcpStream) {
let mut buffer = [0; 512];
stream.read(&mut buffer).unwrap();

```

```

#
let get = b"GET / HTTP/1.1\r\n";
// ...snip...

let (status_line, filename) = if buffer.starts_with(get) {
 ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
} else {
 ("HTTP/1.1 404 NOT FOUND\r\n\r\n", "404.html")
};

let mut file = File::open(filename).unwrap();
let mut contents = String::new();

file.read_to_string(&mut contents).unwrap();

let response = format!("{}", status_line, contents);

stream.write(response.as_bytes()).unwrap();
stream.flush().unwrap();
}

```

列表 20-9: 重构代码使得 `if` 和 `else` 块中只包含两个情况所不同的代码

这里，`if` 和 `else` 块所做的唯一的事就是在一个元组中返回合适的状态行和文件名的值；接着使用第十八章讲到的使用模式的 `let` 语句通过解构元组的两部分给 `filename` 和 `header` 赋值。

读取文件和写入响应的冗余代码现在位于 `if` 和 `else` 块之外，并会使用变量 `status_line` 和 `filename`。这样更易于观察这两种情况真正有何不同，并且如果需要改变如何读取文件或写入响应时只需要更新一处的代码。列表 20-9 中代码的行为与列表 20-8 完全一样。

好极了！我们有了一个 40 行左右 Rust 代码的小而简单的 `server`，它对一个请求返回页面内容而对所有其他请求返回 404 响应。

不过因为 `server` 运行于单线程中，它一次只能处理一个请求。让我们模拟一些慢请求来看看这如何会称为一个问题。

# 慢请求如何影响吞吐率

ch20-02-slow-requests.md

commit d06a6a181fd61704cbf7feb55bc61d518c6469f9

目前 **server** 会依次处理每一个请求。这对于向我们这样并不期望有非常大量请求的服务来说是可行的，不过随着程序变得更复杂，这样的串行处理并不是最优的。

因为当前的程序顺序处理连接，在完成第一个连接的处理之前不会处理第二个连接。如果一个请求花费很长时间来处理，这段时间接收的请求则不得不等待这个长请求结束，即便这些新请求可以很快就处理完。让我们实际尝试一下。

## 在当前 **server** 实现中模拟慢请求

让我们看看一个花费很长时间处理的请求对当前的 **server** 实现有何影响。列表 20-10 展示了对另一个请求的响应代码，`/sleep`，它会使 **server** 在响应之前休眠五秒。这将模拟一个慢请求以便体现出 **server** 在串行的处理请求。

文件名: `src/main.rs`

```
use std::thread;
use std::time::Duration;
use std::io::prelude::*;
use std::net::TcpStream;
use std::fs::File;
// ...snip...

fn handle_connection(mut stream: TcpStream) {
 # let mut buffer = [0; 512];
 # stream.read(&mut buffer).unwrap();
 // ...snip...

 let get = b"GET / HTTP/1.1\r\n";
 let sleep = b"GET /sleep HTTP/1.1\r\n";

 let (status_line, filename) = if buffer.starts_with(get) {
 ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
 } else if buffer.starts_with(sleep) {
 thread::sleep(Duration::from_secs(5));
 ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
 } else {
 ("HTTP/1.1 404 NOT FOUND\r\n\r\n", "404.html")
 };

 // ...snip...
}
```

列表 20-10: 通过识别 `/sleep` 并休眠五秒来模拟慢请求

这段代码有些凌乱，不过对于模拟的目的来说已经足够！这里创建了第二个请求 `sleep`，我们会识别其数据。在 `if` 块之后增加了一个 `else if` 来检查 `/sleep` 请求，当发现这个请求时，在渲染欢迎页面之前会先休眠五秒。

现在就可以真切的看出我们的 **server** 有多么的原始：真实的库将会以更简洁的方式处理多请求识别问题。

使用 `cargo run` 启动 **server**，并接着打开两个浏览器窗口：一个请求 `http://localhost:8080/` 而另一个请求 `http://localhost:8080/sleep`。如果像之前一样多次请求 `/`，会发现响应的比较快速。不过如果请求 `/sleep` 之后在请求 `/`，就会看到 `/` 会等待直到 `sleep` 休眠完五秒之后才出现。

这里有多种办法来改变我们的 **web server** 使其避免所有请求都排在慢请求之后；其一便是实现一个线程池。

## 使用线程池改善吞吐量

线程池 (*thread pool*) 是一组预先分配的用来处理任务的线程。当程序收到一个新任务，线程池中的一个线程会被分配任务并开始处理。其余的线程则可用于处理在第一个线程处理任务的同时处理其他接收到的任务。当第一个线程处理完任务时，它会返回空闲线程池中等待处理新任务。

线程池允许我们并发处理连接：可以在老连接处理完之前就开始处理新连接。这增加了 **server** 的吞吐量。

如下是我们将要实现的：不再等待每个请求处理完才开始下一个，我们将每个连接的处理发送给不同的线程。这些线程来此程序启动时分配的四个线程的线程池。限制较少的线程数的原因是如果为每个新来的请求都创建一个新线程，则千万级的请求就造成灾难，他们会用尽服务器的资源并导致所有请求的处理都被终止。

不同于分配无限的线程，线程池中将有固定数量的等待线程。当新进请求时，将请求发送到线程池中做处理。线程池会维护一个接收请求的队列。每一个线程会从队列中取出一个请求，处理请求，接着向对队列索取另一个请求。通过这种设计，则可以并发处理  $N$  个请求，其中  $N$  为线程数。这仍然意味着  $N$  个慢请求会阻塞队列中的请求，不过确实将能够处理的慢请求数量从一增加到了  $N$ 。

这个设计是多种改善 **web server** 吞吐量的方法之一。不过本书并不是有关 **web server** 的，所以这一种方法是我们将要涉及的。其他的方法有 **fork/join** 模型和单线程异步 **I/O** 模型。如果你对这个主题感兴趣，则可以阅读更多关于其他解决方案的内容并尝试用 **Rust** 实现他们；对于一个像 **Rust** 这样的底层语言，所有这些方法都是可能的。

# 设计线程池接口

ch20-03-designing-the-interface.md

commit d06a6a181fd61704cbf7feb55bc61d518c6469f9

让我们讨论一下线程池看起来怎样。库作者们经常会发现，当尝试设计一些代码时，首先编写客户端接口确实有助于指导代码设计。以期望的调用方式来构建 **API** 代码的结构，接着在这个结构之内实现功能，而不是先实现功能再设计公有 **API**。

类似于第十二章项目中使用的测试驱动开发。这里将要使用编译器驱动开发（**Compiler Driven Development**）。我们将编写调用所期望的函数的代码，接着依靠编译器告诉我们接下来需要修改什么。编译器错误信息会指导我们的实现。

## 如果使用 `thread::spawn` 的代码结构

首先，让我们探索一下为每一个连接都创建一个线程看起来如何。这并不是最终方案，因为正如之前讲到的它会潜在的分配无限的线程，不过这是一个开始。列表 20-11 展示了 `main` 的改变，它在 `for` 循环中为每一个流分配了一个新线程进行处理：

Filename: src/main.rs

```
use std::thread;
use std::io::prelude::*;
use std::net::TcpListener;
use std::net::TcpStream;
#
fn main() {
 let listener = TcpListener::bind("127.0.0.1:8080").unwrap();

 for stream in listener.incoming() {
 let stream = stream.unwrap();

 thread::spawn(|| {
 handle_connection(stream);
 });
 }
}
fn handle_connection(mut stream: TcpStream) {}
```

列表 20-11：为每一个流新建一个线程

正如第十六章讲到的，`thread::spawn` 会创建一个新线程并运行闭包中的代码。如果运行这段代码并在两个浏览器标签页中加载 `/sleep` 和 `/`，确实会发现 `/` 请求并没有等待 `/sleep` 结束。不过正如之前提到的，这最终会使系统崩溃因为我们无限制的创建新线程。

## 为 `ThreadPool` 创建一个类似的接口

我们期望线程池以类似且熟悉的方式工作，以便从线程切换到线程池并不会对运行于线程池中的代码做出较大的修改。列表 20-12 展示我们希望用来替换 `thread::spawn` 的 `ThreadPool` 结构体的假想接口：

文件名: src/main.rs

```
use std::thread;
use std::io::prelude::*;
```



```

use std::net::TcpListener;
use std::net::TcpStream;
struct ThreadPool;
impl ThreadPool {
fn new(size: u32) -> ThreadPool { ThreadPool }
fn execute<F>(&self, f: F)
where F: FnOnce() + Send + 'static {}
}
#
fn main() {
 let listener = TcpListener::bind("127.0.0.1:8080").unwrap();
 let pool = ThreadPool::new(4);

 for stream in listener.incoming() {
 let stream = stream.unwrap();

 pool.execute(|| {
 handle_connection(stream);
 });
 }
}
fn handle_connection(mut stream: TcpStream) {}

```

列表 20-12: 如何使用我们将要实现的 `ThreadPool`

这里使用 `ThreadPool::new` 来创建一个新的线程池，它有一个可配置的线程数的参数，在这里是四。这样在 `for` 循环中，`pool.execute` 将会以类似 `thread::spawn` 的方式工作。

## 采用编译器驱动开发来驱动 **API** 的编译

继续并对列表 20-12 中的 `src/main.rs` 做出修改，并利用编译器错误来驱动开发。下面是我们得到的第一个错误：

```

$ cargo check
 Compiling hello v0.1.0 (file:///projects/hello)
error[E0433]: failed to resolve. Use of undeclared type or module `ThreadPool`
 --> src/main.rs:10:16
 |
10 | let pool = ThreadPool::new(4);
 | ^^^^^^^^^^^^^^^^^ Use of undeclared type or module
 | `ThreadPool`

error: aborting due to previous error

```

好的，我们需要一个 `ThreadPool`。将 `hello crate` 从二进制 `crate` 转换为库 `crate` 来存放 `ThreadPool` 实现，因为线程池实现与我们的 `web server` 的特定工作相独立。一旦写完了线程池库，就可以在任何工作中使用这个功能，而不仅仅是处理网络请求。

创建 `src/lib.rs` 文件，它包含了目前可用的最简单的 `ThreadPool` 定义：

文件名: `src/lib.rs`

```
pub struct ThreadPool;
```

接着创建一个新目录，`src/bin`，并将二进制 `crate` 根文件从 `src/main.rs` 移动到 `src/bin/main.rs`。这使得库 `crate` 成为 `hello` 目录的主要 `crate`；不过仍然可以使用 `cargo run` 运行 `src/bin/main.rs` 二进制文件。移动了 `main.rs` 文件之后，修改文件开头加入如下代码来引入库 `crate` 并将 `ThreadPool` 引入作用域：

文件名: `src/bin/main.rs`

```
extern crate hello;
use hello::ThreadPool;
```

再次尝试运行来得到下一个需要解决的错误：

```
$ cargo check
Compiling hello v0.1.0 (file:///projects/hello)
error: no associated item named `new` found for type `hello::ThreadPool` in the
current scope
--> src/main.rs:13:16
|
13 | let pool = ThreadPool::new(4);
| ^^^^^^^^^^^^^^^^^^^
|
```

好的，下一步是为 `ThreadPool` 创建一个叫做 `new` 的关联函数。我们还知道 `new` 需要有一个参数可以接受 `4`，而且 `new` 应该返回 `ThreadPool` 实例。让我们实现拥有此特征的最小化 `new` 函数：

文件夹: `src/lib.rs`

```
pub struct ThreadPool;

impl ThreadPool {
 pub fn new(size: u32) -> ThreadPool {
 ThreadPool
 }
}
```

这里的 `size` 参数是 `u32` 类型，因为我们知道为负的线程数没有意义。`u32` 是一个很好的默认值。一旦真正实现了 `new`，我们将考虑实现需要选择什么类型，目前我们仅仅处理编译器错误。

再次编译检查这段代码：

```
$ cargo check
Compiling hello v0.1.0 (file:///projects/hello)
warning: unused variable: `size`, #[warn(unused_variables)] on by default
--> src/lib.rs:4:16
|
4 | pub fn new(size: u32) -> ThreadPool {
| ^^^^

error: no method named `execute` found for type `hello::ThreadPool` in the
current scope
--> src/main.rs:18:14
|
18 | pool.execute(|| {
| ^^^^^^^
```

好的，一个警告和一个错误。暂时先忽略警告，错误是因为并没有 `ThreadPool` 上的 `execute` 方法。让我们来定义一个，它应该能接受一个闭包。如果你还记得第十三章，闭包作为参数时可以使用三个不同的 `trait`: `Fn`、`FnMut` 和 `FnOnce`。那么应该用哪一种闭包呢？好吧，最终需要实现的类似于 `thread::spawn`；`thread::spawn` 的签名在其参数中使用了何种 `bound` 呢？查看文档会发现：

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
 where
 F: FnOnce() -> T + Send + 'static,
 T: Send + 'static
```

`F` 是我们这里关心的参数；`T` 与返回值有关所以我们并不关心。考虑到 `spawn` 使用 `FnOnce` 作为 `F` 的 `trait bound`，这可能也是我们需要的，因为最终会将传递给 `execute` 的参数传给 `spawn`。因为处理请求的线程只会执行闭包一次，这也进一步确认了 `FnOnce` 是我们需要的 `trait`。

`F` 还有 `trait bound` `Send` 和生命周期绑定 `'static`，这对我们的情况也是有意义的：需要 `Send` 来将闭包从一个线程转移到另一个线程，而 `'static` 是因为并不知道线程会执行多久。让我们编写一个使用这些 `bound` 的泛型参数 `F` 的 `ThreadPool` 的 `execute` 方法：

文件名: `src/lib.rs`

```
pub struct ThreadPool;
impl ThreadPool {
 // ...snip...

 pub fn execute<F>(&self, f: F)
 where
 F: FnOnce() + Send + 'static
 {

 }
}
```

`FnOnce` `trait` 仍然需要之后的 `()`，因为这里的 `FnOnce` 代表一个没有参数也没有返回值的闭包。正如函数的定义，返回值类型可以从签名中省略，不过即便没有参数也需要括号。

因为我们仍在努力使接口能够编译，这里增加了 `execute` 方法的最小化实现，它没有做任何工作。再次进行检查：

```
$ cargo check
Compiling hello v0.1.0 (file:///projects/hello)
warning: unused variable: `size`, #[warn(unused_variables)] on by default
--> src/lib.rs:4:16
|
4 | pub fn new(size: u32) -> ThreadPool {
| ^^^^

warning: unused variable: `f`, #[warn(unused_variables)] on by default
--> src/lib.rs:8:30
|
8 | pub fn execute<F>(&self, f: F)
| ^
```

现在就只有警告了！能够编译了！注意如果尝试 `cargo run` 运行程序并在浏览器中发起请求，仍会在浏览器中出现本章开始时那样的错误。这个库实际上还没有调用传递给 `execute` 的闭包！

一个你可能听说过的关于像 `Haskell` 和 `Rust` 这样有严格编译器的语言的说法是“如果代码能够编译，它就能工作”。这是一个提醒大家的好时机，这只是一个说法和一种有时存在的感觉，实际上并不是完全正确的。我们的项目可以编译，不过它绝对没有做任何工作！如果构建一个真实且功能完整的项目，则需花费大量的时间来开始编写单元测试来检查代码能否编译并且拥有期望的行为。

# 创建线程池并储存线程

ch20-04-storing-threads.md

commit d06a6a181fd61704cbf7feb55bc61d518c6469f9

之前的警告是因为在 `new` 和 `execute` 中没有对参数做任何操作。让我们用期望的实际行为实现他们。

## 验证池中的线程数

以考虑 `new` 作为开始。之前提到使用无符号类型作为 `size` 参数的类型，因为为负的线程数没有意义。然而，零个线程同样没有意义，不过零是一个完全有效的 `u32` 值。让我们在返回 `ThreadPool` 之前检查 `size` 是否大于零，并使用 `assert!` 宏在得到零时 `panic`，如列表 20-13 所示：

文件名: `src/lib.rs`

```
pub struct ThreadPool;
impl ThreadPool {
 /// Create a new ThreadPool.
 ///
 /// The size is the number of threads in the pool.
 ///
 /// # Panics
 ///
 /// The `new` function will panic if the size is zero.
 pub fn new(size: u32) -> ThreadPool {
 assert!(size > 0);

 ThreadPool
 }

 // ...snip...
}
```

列表 20-13: 实现 `ThreadPool::new` 在 `size` 为零时 `panic`

趁着这个机会我们用文档注释为 `ThreadPool` 增加了一些文档。注意这里遵循了良好的文档实践并增加了一个部分提示函数会 `panic` 的情况，正如第十四章所讨论的。尝试运行 `cargo doc --open` 并点击 `ThreadPool` 结构体来查看生成的 `new` 的文档看起来如何！

相比像这里使用 `assert!` 宏，也可以让 `new` 像之前 I/O 项目中列表 12-9 中 `Config::new` 那样返回一个 `Result`，不过在这里我们选择创建一个没有任何线程的线程池应该是要给不可恢复的错误。如果你想做的更好，尝试编写一个采用如下签名的 `new` 版本来感受一下两者的区别：

```
fn new(size: u32) -> Result<ThreadPool, PoolCreationError> {
```

## 在线程池中储存线程

现在有了一个有效的线程池线程数，就可以实际创建这些线程并在返回之前将他们储存在 `ThreadPool` 结构体中。

这引出了另一个问题：如何“储存”一个线程？让我们再看看 `thread::spawn` 的签名：

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
 where
 F: FnOnce() -> T + Send + 'static,
```

```
T: Send + 'static
```

`spawn` 返回 `JoinHandle<T>`，其中 `T` 是闭包返回的类型。尝试使用 `JoinHandle` 来看看会发生什么。在我们的情况中，传递给线程池的闭包会处理连接并不返回任何值，所以 `T` 将会是单元类型 `()`。

这还不能编译，不过考虑一下列表 20-14 所示的代码。我们改变了 `ThreadPool` 的定义来存放一个 `thread::JoinHandle<()>` 的 `vector` 实例，使用 `size` 容量来初始化，并设置一个 `for` 循环来运行创建线程的代码，并返回包含这些线程的 `ThreadPool` 实例：

文件名: `src/lib.rs`

```
use std::thread;

pub struct ThreadPool {
 threads: Vec<thread::JoinHandle<()>>,
}

impl ThreadPool {
 // ...snip...
 pub fn new(size: u32) -> ThreadPool {
 assert!(size > 0);

 let mut threads = Vec::with_capacity(size);

 for _ in 0..size {
 // create some threads and store them in the vector
 }

 ThreadPool {
 threads
 }
 }

 // ...snip...
}
```

列表 20-14：为 `ThreadPool` 创建一个 `vector` 来存放线程

这里将 `std::thread` 引入库 `crate` 的作用域，因为使用了 `thread::JoinHandle` 作为 `ThreadPool` 中 `vector` 元素的类型。

在得到了有效的数量之后，就可以新建一个存放 `size` 个元素的 `vector`。本书还未使用过 `with_capacity`；它与 `Vec::new` 做了同样的工作，不过有一个重要的区别：它为 `vector` 预先分配空间。因为已经知道了 `vector` 中需要 `size` 个元素，预先进行分配比仅仅 `Vec::new` 要稍微有效率一些，因为 `Vec::new` 随着插入元素而重新改变大小。因为一开始就用所需的确定大小来创建 `vector`，为其增减元素时不会改变底层 `vector` 的大小。

如果代码能够工作就应是如此效果，不过他们还不能工作！如果检查他们，会得到一个错误：

```
$ cargo check
 Compiling hello v0.1.0 (file:///projects/hello)
error[E0308]: mismatched types
 --> src/main.rs:70:46
 |
70 | let mut threads = Vec::with_capacity(size);
 | ^^^^^ expected usize, found u32

error: aborting due to previous error
```

`size` 是 `u32`，不过 `Vec::with_capacity` 需要一个 `usize`。这里有两个选择：可以改变函数签名，或者可以将 `u32` 转换为 `usize`。如果你还记得定义 `new` 时，并没有仔细考虑有意义的数值类型，只是随便选了一个。现在来进行一些思考吧。考虑到 `size` 是 `vector` 的长度，`usize` 就很有道理了。甚至他们的名字都很类似！改变 `new` 的签名，这会使得列表 20-14 的代码能够编译：

```
fn new(size: usize) -> ThreadPool {
```

如果再次运行 `cargo check`，会得到一些警告，不过应该能成功编译。

列表 20-14 的 `for` 循环中留下了一个关于创建线程的注释。如何实际创建线程呢？这是一个难题。这些线程应该做什么呢？这里并不知道他们需要做什么，因为 `execute` 方法获取闭包并传递给了线程池。

让我们稍微重构一下：不再储存一个 `JoinHandle<()>` 实例的 `vector`，将创建一下新的结构体来代表 `worker` 的概念。`worker` 会接收 `execute` 方法，并会处理实际的闭包调用。另外储存固定 `size` 数量的还不知道将要执行什么闭包的 `Worker` 实例，也可以为每一个 `worker` 设置一个 `id`，这样就可以在日志和调试中区别线程池中的不同 `worker`。

让我们做出如下修改：

1. 定义 `Worker` 结构体存放 `id` 和 `JoinHandle<()>`
2. 修改 `ThreadPool` 存放一个 `Worker` 实例的 `vector`
3. 定义 `Worker::new` 函数，它获取一个 `id` 数字并返回一个带有 `id` 和用空闭包分配的线程的 `Worker` 实例，之后会修复这些
4. 在 `ThreadPool::new` 中，使用 `for` 循环来计数生成 `id`，使用这个 `id` 新建 `Worker`，并储存进 `vector` 中

如果你渴望挑战，在查看列表 20-15 中的代码之前尝试自己实现这些修改。

准备好了吗？列表 20-15 就是一个做出了这些修改的例子：

文件名: `src/lib.rs`

```
use std::thread;

pub struct ThreadPool {
 workers: Vec<Worker>,
}

impl ThreadPool {
 // ...snip...
 pub fn new(size: usize) -> ThreadPool {
 assert!(size > 0);

 let mut workers = Vec::with_capacity(size);

 for id in 0..size {
 workers.push(Worker::new(id));
 }

 ThreadPool {
 workers
 }
 }
 // ...snip...
}

struct Worker {
 id: usize,
 thread: thread::JoinHandle<()>,
}
```

```
impl Worker {
 fn new(id: usize) -> Worker {
 let thread = thread::spawn(|| {});

 Worker {
 id,
 thread,
 }
 }
}
```

列表 20-15: 修改 `ThreadPool` 存放 `Worker` 实例而不是直接存放线程

这里选择将 `ThreadPool` 中字段名从 `threads` 改为 `workers`，因为我们改变了存放内容为 `Worker` 而不是 `JoinHandle<()>`。使用 `for` 循环中的计数作为 `Worker::new` 的参数，并将每一个新建的 `Worker` 储存在叫做 `workers` 的 `vector` 中。

`Worker` 结构体和其 `new` 函数是私有的，因为外部代码（比如 `src/bin/main.rs` 中的 `server`）并不需要 `ThreadPool` 中使用 `Worker` 结构体的实现细节。`Worker::new` 函数使用 `id` 参数并储存了使用一个空闭包创建的 `JoinHandle<()>`。

这段代码能够编译并用指定给 `ThreadPool::new` 的参数创建储存了一系列的 `Worker` 实例，不过仍然没有处理 `execute` 中得到的闭包。让我们聊聊接下来怎么做。

## 使用通道向线程发送请求

[ch20-05-sending-requests-via-channels.md](#)

commit 2e269ff82193fd65df8a87c06561d74b51ac02f7

下一个需要解决的问题是（线程中的）闭包完全没有做任何工作。我们一直在绕过获取 `execute` 方法中实际期望执行的闭包的问题，不过看起来在创建 `ThreadPool` 时就需要知道实际的闭包。

不过考虑一下真正需要做的：我们希望刚创建的 `Worker` 结构体能够从 `ThreadPool` 的队列中获取任务，并在线程中执行他们。

在第十六章中，我们学习了通道。通道是一个沟通两个线程的良好手段，对于这个例子来说则是绝佳的。通道将充当任务队列的作用，`execute` 将通过 `ThreadPool` 向其中线程正在寻找工作的 `Worker` 实例发送任务。如下是这个计划：

1. `ThreadPool` 会创建一个通道并充当发送端。
2. 每个 `Worker` 将会充当通道的接收端。
3. 新建一个 `Job` 结构体来存放用于向通道中发送的闭包。
4. `ThreadPool` 的 `execute` 方法会在发送端发出期望执行的任务。
5. 在线程中，`Worker` 会遍历通道的接收端并执行任何接收到的任务。

让我们以在 `ThreadPool::new` 中创建通道并让 `ThreadPool` 实例充当发送端开始，如列表 20-16 所示。`Job` 是将在通道中发出的类型；目前它是一个没有任何内容的结构体：

文件名: `src/lib.rs`

```
use std::thread;
// ...snip...
use std::sync::mpsc;

pub struct ThreadPool {
 workers: Vec<Worker>,
 sender: mpsc::Sender<Job>,
}

struct Job;

impl ThreadPool {
 // ...snip...
 pub fn new(size: usize) -> ThreadPool {
 assert!(size > 0);

 let (sender, receiver) = mpsc::channel();

 let mut workers = Vec::with_capacity(size);

 for id in 0..size {
 workers.push(Worker::new(id));
 }

 ThreadPool {
 workers,
 sender,
 }
 }
 // ...snip...
}
#
```



```

struct Worker {
id: usize,
thread: thread::JoinHandle<()>,
}
#
impl Worker {
fn new(id: usize) -> Worker {
let thread = thread::spawn(|| {});
#
Worker {
id,
thread,
}
}
}

```

列表 20-16: 修改 `ThreadPool` 来储存一个发送 `Job` 实例的通道发送端

在 `ThreadPool::new` 中，新建了一个通道，并接着让线程池在接收端等待。这段代码能够编译，不过仍有警告。

在线程池创建每个 **worker** 时将通道的接收端传递给他们。须知我们希望在 **worker** 所分配的线程中使用通道的接收端，所以将在闭包中引用 `receiver` 参数。列表 20-17 中展示的代码还不能编译：

文件名: `src/lib.rs`

```

impl ThreadPool {
 // ...snip...
 pub fn new(size: usize) -> ThreadPool {
 assert!(size > 0);

 let (sender, receiver) = mpsc::channel();

 let mut workers = Vec::with_capacity(size);

 for id in 0..size {
 workers.push(Worker::new(id, receiver));
 }

 ThreadPool {
 workers,
 sender,
 }
 }
 // ...snip...
}

// ...snip...

impl Worker {
 fn new(id: usize, receiver: mpsc::Receiver<Job>) -> Worker {
 let thread = thread::spawn(|| {
 receiver;
 });

 Worker {
 id,
 thread,
 }
 }
}

```

列表 20-17: 将通道的接收端传递给 **worker**

这是一些小而直观的修改：将通道的接收端传递进了 `Worker::new`，并接着在闭包中使用他们。

如果尝试检查代码，会得到这个错误：

```
$ cargo check
Compiling hello v0.1.0 (file:///projects/hello)
error[E0382]: use of moved value: `receiver`
--> src/lib.rs:27:42
|
27 | workers.push(Worker::new(id, receiver));
| ^^^^^^^^^ value moved here in
previous iteration of loop
|
= note: move occurs because `receiver` has type
`std::sync::mpsc::Receiver<Job>`, which does not implement the `Copy` trait
```

这些代码还不能编译的原因如上因为它尝试将 `receiver` 传递给多个 `Worker` 实例。回忆第十六章，**Rust** 所提供的通道实现是多生产者，单消费者的，所以不能简单的克隆通道的消费端来解决问题。即便可以我们也不希望克隆消费端；在所有的 `worker` 中共享单一 `receiver` 才是我们希望的在线程间分发任务的机制。

另外，从通道队列中取出任务涉及到修改 `receiver`，所以这些线程需要一个能安全的共享和修改 `receiver` 的方式。如果修改不是线程安全的，则可能遇到竞争状态，例如两个线程因同时在队列中取出相同的任务并执行了相同的工作。

所以回忆一下第十六章讨论的线程安全智能指针，为了在多个线程间共享所有权并允许线程修改其值，需要使用 `Arc<Mutex<T>>`。 `Arc` 使得多个 `worker` 拥有接收端，而 `Mutex` 则确保一次只有一个 `worker` 能从接收端得到任务。列表 20-18 展示了所做的修改：

文件名: `src/lib.rs`

```
use std::thread;
use std::sync::mpsc;
use std::sync::Arc;
use std::sync::Mutex;

// ...snip...

pub struct ThreadPool {
workers: Vec<Worker>,
sender: mpsc::Sender<Job>,
}
struct Job;
#
impl ThreadPool {
 // ...snip...
 pub fn new(size: usize) -> ThreadPool {
 assert!(size > 0);

 let (sender, receiver) = mpsc::channel();

 let receiver = Arc::new(Mutex::new(receiver));

 let mut workers = Vec::with_capacity(size);

 for id in 0..size {
 workers.push(Worker::new(id, receiver.clone()));
 }

 ThreadPool {
 workers,
 sender,
 }
 }
}
```

```

 }
}

// ...snip...
}
struct Worker {
id: usize,
thread: thread::JoinHandle<()>,
}
#
impl Worker {
fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
// ...snip...
let thread = thread::spawn(|| {
receiver;
});
Worker {
id,
thread,
}
}
}
}

```

列表 20-18: 使用 `Arc` 和 `Mutex` 在 `worker` 间共享通道的接收端

在 `ThreadPool::new` 中，将通道的接收端放入一个 `Arc` 和一个 `Mutex` 中。对于每一个新 `worker`，则克隆 `Arc` 来增加引用计数，如此这些 `worker` 就可以共享接收端的所有权了。

通过这些修改，代码可以编译了！我们做到了！

最好让我们实现 `ThreadPool` 上的 `execute` 方法。同时也要修改 `Job` 结构体：它将不再是结构体，`Job` 将是一个有着 `execute` 接收到的闭包类型的 `trait` 对象的类型别名。我们讨论过类型别名如何将长的类型变短，现在就这种情况！看一看列表 20-19:

文件名: `src/lib.rs`

```

// ...snip...
pub struct ThreadPool {
workers: Vec<Worker>,
sender: mpsc::Sender<Job>,
}
use std::sync::mpsc;
struct Worker {}

type Job = Box<FnOnce() + Send + 'static>;

impl ThreadPool {
 // ...snip...

 pub fn execute<F>(&self, f: F)
 where
 F: FnOnce() + Send + 'static
 {
 let job = Box::new(f);

 self.sender.send(job).unwrap();
 }
}

// ...snip...

```

列表 20-19: 为存放每一个闭包的 `Box` 创建一个 `Job` 类型别名, 接着在通道中发出

在使用 `execute` 得到的闭包新建 `Job` 实例之后, 将这些任务从通道的发送端发出。这里调用 `send` 上的 `unwrap`, 因为如果接收端停止接收新消息则发送可能会失败, 这可能发生于我们停止了所有的执行线程。不过目前这是不可能的, 因为只要线程池存在他们就会一直执行。使用 `unwrap` 是因为我们知道失败不可能发生, 即便编译器不这么认为, 正如第九章讨论的这是 `unwrap` 的一个恰当用法。

那我们结束了吗? 不完全是! 在 `worker` 中, 传递给 `thread::spawn` 的闭包仍然还只是引用了通道的接收端。但是我们需要闭包一直循环, 向通道的接收端请求任务, 并在得到任务时执行他们。如列表 20-20 对 `Worker::new` 做出修改:

文件名: `src/lib.rs`

```
// ...snip...

impl Worker {
 fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
 let thread = thread::spawn(move || {
 loop {
 let job = receiver.lock().unwrap().recv().unwrap();

 println!("Worker {} got a job; executing.", id);

 (*job)();
 }
 });

 Worker {
 id,
 thread,
 }
 }
}
```

列表 20-20: 在 `worker` 线程中接收并执行任务

这里, 首先在 `receiver` 上调用了 `lock` 来获取互斥器, 接着 `unwrap` 在出现任何错误时 `panic`。如果互斥器处于一种叫做被污染 (*poisoned*) 的状态时获取锁可能会失败, 这可能发生于其他线程在持有锁时 `panic` 了并没有释放锁。如果当前线程因为这个原因不能得到锁, 调用 `unwrap` 使其 `panic` 也是正确的行为。如果你觉得有意义的话请随意将 `unwrap` 改为带有错误信息的 `expect`。

如果锁定了互斥器, 接着调用 `recv` 从通道中接收 `Job`。最后的 `unwrap` 也绕过了一些错误, `recv` 在通道的发送端关闭时会返回 `Err`, 类似于 `send` 在接收端关闭时返回 `Err` 一样。

调用 `recv` 的代码块; 也就是说, 它还没有任务, 这个线程会等待直到有可用的任务。`Mutex<T>` 确保一次只有一个 `Worker` 线程尝试请求任务。

理论上这段代码应该能够编译。不幸的是, `Rust` 编译器仍不够完美, 会给出如下错误:

```
error[E0161]: cannot move a value of type std::ops::FnOnce() +
std::marker::Send: the size of std::ops::FnOnce() + std::marker::Send cannot be
statically determined
--> src/lib.rs:63:17
|
63 | (*job)();
| ^^^^^^^
```

这个错误非常的神秘，因为这个问题本身就很神秘。为了调用储存在 `Box<T>`（这正是 `Job` 别名的类型）中的 `FnOnce` 闭包，该闭包需要能将自己移动出 `Box<T>`，因为当调用这个闭包时，它获取 `self` 的所有权。通常来说，将值移动出 `Box<T>` 是不被允许的，因为 `Rust` 不知道 `Box<T>` 中的值将会有多大；回忆第十五章能够正常使用 `Box<T>` 是因为我们将未知大小的值储存进 `Box<T>` 从而得到已知大小的值。

第十七章曾见过，列表 17-15 中有使用了 `self: Box<Self>` 语法的方法，它获取了储存在 `Box<T>` 中的 `Self` 值的所有权。这正是我们希望做的，然而不幸的是 `Rust` 调用闭包的那部分实现并没有使用 `self: Box<Self>`。所以这里 `Rust` 也不知道它可以使用 `self: Box<Self>` 来获取闭包的所有权并将闭包移动出 `Box<T>`。

将来列表 20-20 中的代码应该能够正常工作。`Rust` 仍在努力改进提升编译器。有很多像你一样的人正在修复这个以及其他问题！当你结束了本书的阅读，我们希望看到你也成为他们中的一员。

不过目前让我们绕过这个问题。所幸有一个技巧可以显式的告诉 `Rust` 我们处于可以获取使用 `self: Box<Self>` 的 `Box<T>` 中值的所有权的状态，而一旦获取了闭包的所有权就可以调用它了。这涉及到定义一个新 `trait`，它带有一个在签名中使用 `self: Box<Self>` 的方法 `call_box`，为任何实现了 `FnOnce()` 的类型定义这个 `trait`，修改类型别名来使用这个新 `trait`，并修改 `Worker` 使用 `call_box` 方法。这些修改如列表 20-21 所示：

文件名: `src/lib.rs`

```
trait FnBox {
 fn call_box(self: Box<Self>);
}

impl<F: FnOnce()> FnBox for F {
 fn call_box(self: Box<F>) {
 (*self)()
 }
}

type Job = Box<FnBox + Send + 'static>;

// ...snip...

impl Worker {
 fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
 let thread = thread::spawn(move || {
 loop {
 let job = receiver.lock().unwrap().recv().unwrap();

 println!("Worker {} got a job; executing.", id);

 job.call_box();
 }
 });

 Worker {
 id,
 thread,
 }
 }
}
```

列表 20-21：新增一个 `trait FnBox` 来绕过当前 `Box<FnOnce()>` 的限制

首先，新建了一个叫做 `FnBox` 的 `trait`。这个 `trait` 有一个方法 `call_box`，它类似于其他 `Fn*` `trait` 中的 `call` 方法，除了它获取 `self: Box<Self>` 以便获取 `self` 的所有权并将值从 `Box<T>` 中移动出来。

现在我们希望 `Job` 类型别名是任何实现了新 `trait FnBox` 的 `Box`，而不是 `FnOnce()`。这允许我们在得到 `Job` 值时使用 `Worker` 中的 `call_box`。因为我们为任何 `FnOnce()` 闭包都实现了 `FnBox` `trait`，无需对实际在通道中发出的值做任何修改。

最后，对于 `Worker::new` 的线程中所运行的闭包，调用 `call_box` 而不是直接执行闭包。现在 **Rust** 就能够理解我们的行为是正确的了。

这是非常狡猾且复杂的手段。无需过分担心他们并不是非常有道理；总有一天，这一切将是毫无必要的。

通过这些技巧，线程池处于可以运行的状态了！执行 `cargo run` 并发起一些请求：

```
$ cargo run
 Compiling hello v0.1.0 (file:///projects/hello)
warning: field is never used: `workers`
--> src/lib.rs:7:5
|
7 | workers: Vec<Worker>,
| ^^^^^^^^^^^^^^^^^^^^^
|
= note: #[warn(dead_code)] on by default

warning: field is never used: `id`
--> src/lib.rs:61:5
61 | id: usize,
| ^^^^^^^
|
= note: #[warn(dead_code)] on by default

warning: field is never used: `thread`
--> src/lib.rs:62:5
62 | thread: thread::JoinHandle<()>,
| ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
|
= note: #[warn(dead_code)] on by default

 Finished dev [unoptimized + debuginfo] target(s) in 0.99 secs
 Running `target/debug/hello`
Worker 0 got a job; executing.
Worker 2 got a job; executing.
Worker 1 got a job; executing.
Worker 3 got a job; executing.
Worker 0 got a job; executing.
Worker 2 got a job; executing.
Worker 1 got a job; executing.
Worker 3 got a job; executing.
Worker 0 got a job; executing.
Worker 2 got a job; executing.
```

成功了！现在我们有了一个可以异步执行连接的线程池！它绝不会创建超过四个线程，所以当 **server** 收到大量请求时系统也不会负担过重。如果请求 `/sleep`，**server** 也能够通过另外一个线程处理其他请求。

那么这些警告怎么办呢？难道我们没有使用 `workers`、`id` 和 `thread` 字段吗？好吧，目前我们用了所有这些字段存放了一些数据，不过当设置线程池并开始执行代码在通道中向线程发送任务时，我们并没有对数据进行任何实际的操作。但是如果不存放这些值，他们将会离开作用域：比如，如果不将 `Vec<Worker>` 值作为 `ThreadPool` 的一部分返回，这个 `vector` 在 `ThreadPool::new` 的结尾就会被清理。

那么这些警告有错吗？从某种角度上讲是的，这些警告是错误的，因为我们使用这些字段储存一直需要的数据。从另一种角度来说也不对：使用过后我们也没有做任何操作清理线程池，仅仅通过 `ctrl-C` 来停止程序并让操作系统为我们清理。下面让我们实现 **graceful shutdown** 来清理所创建的一切。



# Graceful Shutdown 与清理

ch20-06-graceful-shutdown-and-cleanup.md

commit 2e269ff82193fd65df8a87c06561d74b51ac02f7

列表 20-21 中的代码如期通过使用线程池异步的响应请求。这里有一些警告说存在一些字段并没有直接被使用，这提醒了我们并没有清理任何内容。当使用 **ctrl-C** 终止主线程，所有其他线程也会立刻停止，即便他们正在处理一个请求。

现在我们要为 `ThreadPool` 实现 `Drop` trait 对线程池中的每一个线程调用 `join`，这样这些线程将会执行完他们的请求。接着会为 `ThreadPool` 实现一个方法来告诉线程他们应该停止接收新请求并结束。为了实践这些代码，修改 `server` 在 graceful Shutdown 之前只接受两个请求。

现在开始为线程池实现 `Drop`。当线程池被丢弃时，应该 `join` 所有线程以确保他们完成其操作。列表 20-22 展示了 `Drop` 实现的第一次尝试；这些代码还不能够编译：

文件名: `src/lib.rs`

```
impl Drop for ThreadPool {
 fn drop(&mut self) {
 for worker in &mut self.workers {
 println!("Shutting down worker {}", worker.id);

 worker.thread.join().unwrap();
 }
 }
}
```

列表 20-22：当线程池离开作用域时 `join` 每个线程

这里遍历线程池中的每个 `workers`，这里使用了 `&mut` 因为 `self` 本身是一个可变引用而且也需要能够修改 `worker`。当特定 `worker` 关闭时会打印出说明信息，接着在对应 `worker` 上调用 `join`。如果 `join` 失败了，通过 `unwrap` 将错误变为 `panic` 从而无法进行 graceful Shutdown。

如下是尝试编译代码时得到的错误：

```
error[E0507]: cannot move out of borrowed content
--> src/lib.rs:65:13
 |
65 | worker.thread.join().unwrap();
 | ^^^^^^ cannot move out of borrowed content
```

因为我们只有每个 `worker` 的可变借用，并不能调用 `join`：`join` 获取其参数的所有权。为了解决这个问题，需要一个方法将 `thread` 移动出拥有其所有权的 `Worker` 实例以便 `join` 可以消费这个线程。列表 17-15 中我们曾见过这么做的方法：如果 `Worker` 存放的是 `Option<thread::JoinHandle<>>`，就可以在 `Option` 上调用 `take` 方法将值从 `Some` 成员中移动出来而对 `None` 成员不做处理。换句话说，正在运行的 `Worker` 的 `thread` 将是 `Some` 成员值，而当需要清理 `worker` 时，将 `Some` 替换为 `None`，这样 `worker` 就没有可以运行的线程了。

所以我们知道了需要更新 `Worker` 的定义为如下：

文件名: `src/lib.rs`

```
use std::thread;
struct Worker {
 id: usize,
```



```

 thread: Option<thread::JoinHandle<()>>,
 }

```

现在依靠编译器来找出其他需要修改的地方。我们会得到两个错误：

```

error: no method named `join` found for type
`std::option::Option<std::thread::JoinHandle<()>>` in the current scope
--> src/lib.rs:65:27
 |
65 | worker.thread.join().unwrap();
 | ^^^^^
error[E0308]: mismatched types
--> src/lib.rs:89:21
 |
89 | thread,
 | ^^^^^^ expected enum `std::option::Option`, found
 | struct `std::thread::JoinHandle`
 = note: expected type `std::option::Option<std::thread::JoinHandle<()>>`
 found type `std::thread::JoinHandle<_>`

```

第二个错误指向 `Worker::new` 结尾的代码；当新建 `Worker` 时需要将 `thread` 值封装进 `Some`：

文件名: `src/lib.rs`

```

impl Worker {
 fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
 // ...snip...

 Worker {
 id,
 thread: Some(thread),
 }
 }
}

```

第一个错误有关 `Drop` 实现，而且我们提到过要调用 `Option` 上的 `take` 将 `thread` 移动出 `worker`。如下是代码：

文件名: `src/lib.rs`

```

impl Drop for ThreadPool {
 fn drop(&mut self) {
 for worker in &mut self.workers {
 println!("Shutting down worker {}", worker.id);

 if let Some(thread) = worker.thread.take() {
 thread.join().unwrap();
 }
 }
 }
}

```

如第十七章我们见过的，`Option` 上的 `take` 方法会取出 `Some` 而留下 `None`。使用 `if let` 解构 `Some` 并得到线程，接着在线程上调用 `join`。如果 `worker` 的线程已然是 `None`，就知道此时这个 `worker` 已经清理了其线程且无需做任何操作。

有了这些修改，代码就能编译且没有任何警告。不过也有坏消息，这些代码还不能以我们期望的方式运行。问题的关键在于 `Worker` 中分配的线程所运行的闭包中的逻辑：调用 `join` 并不会关闭线程，因为他们一直 `loop` 来寻找任务。如果采用这个实现来尝试丢弃 `ThreadPool`，则主线程会永远阻塞在等待第一个线程结束上。

为了修复这个问题，修改线程既监听是否有 `Job` 运行也要监听应该停止监听并退出无限循环的信号。所以通道将发送这个枚举的两个成员之一而不再直接使用 `Job` 实例：

文件名: `src/lib.rs`

```
struct Job;
enum Message {
 NewJob(Job),
 Terminate,
}
```

`Message` 枚举要么是存放了线程需要运行的 `Job` 的 `NewJob` 成员，要么是会导致线程退出循环并终止的 `Terminate` 成员。

同时需要修改通道来使用 `Message` 类型值而不是 `Job`，如列表 20-23 所示：

文件名: `src/lib.rs`

```
pub struct ThreadPool {
 workers: Vec<Worker>,
 sender: mpsc::Sender<Message>,
}

// ...snip...

impl ThreadPool {
 // ...snip...
 pub fn new(size: usize) -> ThreadPool {
 assert!(size > 0);

 let (sender, receiver) = mpsc::channel();

 // ...snip...
 }

 pub fn execute<F>(&self, f: F)
 where
 F: FnOnce() + Send + 'static
 {
 let job = Box::new(f);

 self.sender.send(Message::NewJob(job)).unwrap();
 }
}

// ...snip...

impl Worker {
 fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Message>>>) ->
 Worker {

 let thread = thread::spawn(move ||{
 loop {
 let message = receiver.lock().unwrap().recv().unwrap();

 match message {
 Message::NewJob(job) => {
 println!("Worker {} got a job; executing.", id);
```

```

 job.call_box();
 },
 Message::Terminate => {
 println!("Worker {} was told to terminate.", id);

 break;
 },
}
}
});

Worker {
 id,
 thread: Some(thread),
}
}
}

```

列表 20-23: 收发 `Message` 值并在 `Worker` 收到 `Message::Terminate` 时退出循环

需要将 `ThreadPool` 定义、创建通道的 `ThreadPool::new` 和 `Worker::new` 签名中的 `Job` 改为 `Message`。 `ThreadPool` 的 `execute` 方法需要发送封装进 `Message::NewJob` 成员的任务，当获取到 `NewJob` 时会处理任务而收到 `Terminate` 成员时则会退出循环。

通过这些修改，代码再次能够编译并按照期望的行为运行。不过还是会得到一个警告，因为并没有在任何消息中使用 `Terminate` 成员。如列表 20-14 所示那样修改 `Drop` 实现：

文件名: `src/lib.rs`

```

impl Drop for ThreadPool {
 fn drop(&mut self) {
 println!("Sending terminate message to all workers.");

 for _ in &mut self.workers {
 self.sender.send(Message::Terminate).unwrap();
 }

 println!("Shutting down all workers.");

 for worker in &mut self.workers {
 println!("Shutting down worker {}", worker.id);

 if let Some(thread) = worker.thread.take() {
 thread.join().unwrap();
 }
 }
 }
}

```

列表 20-24: 在对每个 `worker` 线程调用 `join` 之前向 `worker` 发送 `Message::Terminate`

现在遍历了 `worker` 两次，一次向每个 `worker` 发送一个 `Terminate` 消息，一个调用每个 `worker` 线程上的 `join`。如果尝试在同一循环中发送消息并立即 `join` 线程，则无法保证当前迭代的 `worker` 是从通道收到终止消息的 `worker`。

为了更好的理解为什么需要两个分开的循环，想象一下只有两个 `worker` 的场景。如果在一个循环中遍历每个 `worker`，在第一次迭代中 `worker` 是第一个 `worker`，我们向通道发出终止消息并对第一个 `worker` 线程调用 `join`。如果第一个 `worker` 当时正忙于处理请求，则第二个 `worker` 会从通道接收这个终止消息并结束。而我们在

等待第一个 **worker** 结束，不过它永远也不会结束因为第二个线程取走了终止消息。现在我们就阻塞在了等待第一个 **worker** 结束，而无法发出第二条终止消息。死锁！

为了避免此情况，首先从通道中取出所有的 `Terminate` 消息，接着 `join` 所有的线程。因为每个 **worker** 一旦收到终止消息即会停止从通道接收消息，我们就可以确保如果发送同 **worker** 数相同的终止消息，在 `join` 之前每个线程都会收到一个终止消息。

为了实践这些代码，如列表 20-25 所示修改 `main` 在 graceful Shutdown server 之前只接受两个请求：

文件名: `src/bin/main.rs`

```
fn main() {
 let listener = TcpListener::bind("127.0.0.1:8080").unwrap();
 let pool = ThreadPool::new(4);

 let mut counter = 0;

 for stream in listener.incoming() {
 if counter == 2 {
 println!("Shutting down.");
 break;
 }

 counter += 1;

 let stream = stream.unwrap();

 pool.execute(|| {
 handle_connection(stream);
 });
 }
}
```

列表 20-25：在处理两个请求之后通过退出循环来停止 **server**

只处理两次请求并不是生产环境的 **web server** 所期望的行为，不过这可以让我们看清 **graceful shutdown** 和清理起作用了，因为不用再通过 `ctrl-C` 停止 **server** 了。

这里还增加了一个 `counter` 变量在每次收到 **TCP** 流时递增。如果计数到达 2，会停止处理请求并退出 `for` 循环。`ThreadPool` 会在 `main` 的结尾离开作用域，而且还会看到 `drop` 实现的运行。

使用 `cargo run` 启动 **server**，并发起三个请求。第三个请求应该会失败，而终端的输出应该看起来像这样：

```
$ cargo run
Compiling hello v0.1.0 (file:///projects/hello)
Finished dev [unoptimized + debuginfo] target(s) in 1.0 secs
Running `target/debug/hello`
Worker 0 got a job; executing.
Worker 3 got a job; executing.
Shutting down.
Sending terminate message to all workers.
Shutting down all workers.
Shutting down worker 0
Worker 1 was told to terminate.
Worker 2 was told to terminate.
Worker 0 was told to terminate.
Worker 3 was told to terminate.
Shutting down worker 1
Shutting down worker 2
Shutting down worker 3
```

当然，你可能会看到不同顺序的输出。可以从信息中看到服务是如何运行的：**worker 0** 和 **worker 3** 获取了头两个请求，接着在第三个请求时，我们停止接收连接。当 `ThreadPool` 在 `main` 的结尾离开作用域时，其 `Drop` 实现开始工作，线程池通知所有线程终止。每个 **worker** 在收到终止消息时会打印出一个信息，接着线程池调用 `join` 来终止每一个 **worker** 线程。

这个特定的运行过程中一个有趣的地方在于：注意我们向通道中发出终止消息，而在任何线程收到消息之前，就尝试 `join worker 0` 了。**worker 0** 还没有收到终止消息，所以主线程阻塞直到 **worker 0** 结束。与此同时，每一个线程都收到了终止消息。一旦 **worker 0** 结束，主线程就等待其他 **worker** 结束，此时他们都已经收到终止消息并能够停止了。

恭喜！现在我们完成了这个项目，也有了一个使用线程池异步响应请求的基础 **web server**。我们能对 **server** 执行 `graceful shutdown`，它会清理线程池中的所有线程。如下是完整的代码参考：

Filename: `src/bin/main.rs`

```
extern crate hello;
use hello::ThreadPool;

use std::io::prelude::*;
use std::net::TcpListener;
use std::net::TcpStream;
use std::fs::File;
use std::thread;
use std::time::Duration;

fn main() {
 let listener = TcpListener::bind("127.0.0.1:8080").unwrap();
 let pool = ThreadPool::new(4);

 let mut counter = 0;

 for stream in listener.incoming() {
 if counter == 2 {
 println!("Shutting down.");
 break;
 }

 counter += 1;

 let stream = stream.unwrap();

 pool.execute(|| {
 handle_connection(stream);
 });
 }
}

fn handle_connection(mut stream: TcpStream) {
 let mut buffer = [0; 512];
 stream.read(&mut buffer).unwrap();

 let get = b"GET / HTTP/1.1\r\n";
 let sleep = b"GET /sleep HTTP/1.1\r\n";

 let (status_line, filename) = if buffer.starts_with(get) {
 ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
 } else if buffer.starts_with(sleep) {
 thread::sleep(Duration::from_secs(5));
 ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
 } else {
 ("HTTP/1.1 404 NOT FOUND\r\n\r\n", "404.html")
 };
};
```

```

 let mut file = File::open(filename).unwrap();
 let mut contents = String::new();

 file.read_to_string(&mut contents).unwrap();

 let response = format!("{}", status_line, contents);

 stream.write(response.as_bytes()).unwrap();
 stream.flush().unwrap();
}

```

Filename: src/lib.rs

```

use std::thread;
use std::sync::mpsc;
use std::sync::Arc;
use std::sync::Mutex;

enum Message {
 NewJob(Job),
 Terminate,
}

pub struct ThreadPool {
 workers: Vec<Worker>,
 sender: mpsc::Sender<Message>,
}

trait FnBox {
 fn call_box(self: Box<Self>);
}

impl<F: FnOnce()> FnBox for F {
 fn call_box(self: Box<F>) {
 (*self)()
 }
}

type Job = Box<FnBox + Send + 'static>;

impl ThreadPool {
 /// Create a new ThreadPool.
 ///
 /// The size is the number of threads in the pool.
 ///
 /// # Panics
 ///
 /// The `new` function will panic if the size is zero.
 pub fn new(size: usize) -> ThreadPool {
 assert!(size > 0);

 let (sender, receiver) = mpsc::channel();

 let receiver = Arc::new(Mutex::new(receiver));

 let mut workers = Vec::with_capacity(size);

 for id in 0..size {
 workers.push(Worker::new(id, receiver.clone()));
 }

 ThreadPool {
 workers,
 sender,

```

```

 }
}

pub fn execute<F>(&self, f: F)
 where
 F: FnOnce() + Send + 'static
{
 let job = Box::new(f);

 self.sender.send(Message::NewJob(job)).unwrap();
}

}

impl Drop for ThreadPool {
 fn drop(&mut self) {
 println!("Sending terminate message to all workers.");

 for _ in &mut self.workers {
 self.sender.send(Message::Terminate).unwrap();
 }

 println!("Shutting down all workers.");

 for worker in &mut self.workers {
 println!("Shutting down worker {}", worker.id);

 if let Some(thread) = worker.thread.take() {
 thread.join().unwrap();
 }
 }
 }
}

struct Worker {
 id: usize,
 thread: Option<thread::JoinHandle<>>,
}

impl Worker {
 fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Message>>>) ->
 Worker {

 let thread = thread::spawn(move ||{
 loop {
 let message = receiver.lock().unwrap().recv().unwrap();

 match message {
 Message::NewJob(job) => {
 println!("Worker {} got a job; executing.", id);

 job.call_box();
 },
 Message::Terminate => {
 println!("Worker {} was told to terminate.", id);

 break;
 },
 }
 }
 });

 Worker {
 id,
 thread: Some(thread),
 }
 }
}

```

```
}
```

这里还有很多可以做的事！如果你希望继续增强这个项目，如下是一些点子：

- 为 `ThreadPool` 和其公有方法增加更多文档
- 为库的功能增加测试
- 将 `unwrap` 调用改为更健壮的错误处理
- 使用 `ThreadPool` 进行其他不同于处理网络请求的任务
- 在 [crates.io](https://crates.io) 寻找一个线程池 `crate` 并使用它实现一个类似的 `web server`，将其 API 和鲁棒性与我们的实现做对比

## 总结

好极了！你结束了本书的学习！由衷感谢你与我们一道加入这次 **Rust** 之旅。现在你已经准备好出发并实现自己的 **Rust** 项目或帮助他人了。请不要忘记我们的社区，这里有其他 **Rustaceans** 正乐于帮助你迎接 **Rust** 之路上的任何挑战。



## 附录

## **A - 关键字**

## B - 运算符

## C - 可导出的 **trait**

**D - 宏**

## **E - 本书翻译**

## **F - 最新功能**