介绍

ch01-00-introduction.md
commit c51c14215d2ee2cb481bc8a942a3769c6d9a2e1a

欢迎阅读“Rust 程序设计语言”，一本关于 Rust 的介绍性书籍。Rust 是一个着用于安全、速度和并发的编程语言。它的设计不仅可以使程序获得性能和对底层语言的控制，并且能够享受高级语言强大的抽象能力。这些特性使得 Rust 适合那些有类似 C 语言经验并正在寻找一个更安全的替代者的程序员，同时也适合那些来自类似 Python 语言背景，正在探索在不牺牲表现力的情况下编写更好性能代码的人们。

Rust 在编译时进行其绝大多数的安全检查和内存管理决策，因此程序的运行时性能没有受到影响。这让其在许多其他语言不擅长的应用场景中得以大显身手：有可预测空间和时间要求的程序，嵌入到其他语言中，以及编写底层代码，如设备驱动和操作系统。Rust 也很擅长 web 程序：它驱动着 Rust 包注册网站（package registry site），crates.io！我们期待看到**你**使用 Rust 进行创作。

本书是为已经至少了解一门编程语言的读者而写的。读完本书之后，你应该能自如的编写 Rust 程序。我们将通过小而集中并相互依赖的例子来学习 Rust，并向你展示如何使用 Rust 多样的功能，同时了解它们在后台是如何执行的。

为本书做出贡献

本书是开源的。如果你发现任何错误，请不要犹豫，在 GitHub 上发起 issue 或提交 pull request。

安装

ch01-01-installation.md
commit f828919e62aa542aaaae03c1fb565da42374213e

使用 Rust 的第一步是安装。你需要联网来执行本章的命令，因为我们要从网上下载 Rust。

我们将会展示很多使用终端的命令，并且这些代码都以`$`开头。并不需要真正输入`$`，它们在这里代表每行指令的开头。在网上会看到很多使用这个惯例的教程和例子：`$`代表以常规用户运行命令，`#`代表需要用管理员运行的命令。没有以`$`（或`#`）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，所有需要做的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。你可能被提示要输入密码。如果一切顺利，将会出现如下内容：

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

当然，如果你不赞成`curl | sh`这种模式，可以随意下载、检查和运行这个脚本。

在 Windows 上安装

在 Windows 上，前往https://rustup.rs并按照说明下载`rustup-init.exe`。运行并遵循它提供的其余指示。

本书其余 Windows 相关的命令假设你使用`cmd`作为你的 shell。如果你使用不同的 shell，可能能够执行 Linux 和 Mac 用户相同的命令。如果都不行，查看所使用的 shell 的文档。

自定义安装

如果有理由倾向于不使用 rustup.rs，请查看Rust 安装页面获取其他选择。

卸载

卸载 Rust 同安装一样简单。在 shell 中运行卸载脚本

```
$ rustup self uninstall
```

故障排除

安装完 Rust 后，打开 shell，输入：

```
$ rustc --version
```

应该能看到类似这样的版本号、提交 hash 和提交日期，对应你安装时的最新稳定版本：

```
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！）

如果有问题并且你在使用 Windows，检查 Rust（rustc，cargo 等）是否位于`%PATH%`系统变量中。

如果还是不能运行，有许多可以获取帮助的地方。最简单的是 irc.mozilla.org 上的 IRC 频道 #rust-beginners 和供一般讨论之用的 #rust，我们可以使用 Mibbit 访问。然后我们就可以和其他能提供

帮助的 Rustacean（我们这些人自嘲的绰号）聊天了。其它给力的资源包括用户论坛和Stack Overflow。

本地文档

安装程序也包含一份本地文档的拷贝，你可以离线阅读它们。输入`rustup doc`将在浏览器中打开本地文档。

任何你太确认标准库提供的类型或函数是干什么的时候，使用文档 API 查找！

Hello, World!

> ch01-02-hello-world.md
> commit ccbeea7b9fe115cd545881618fe14229d18b307f

现在你已经安装好了 Rust，让我们来编写你的第一个 Rust 程序。当学习一门新语言的时候，编写一个在屏幕上打印 “Hello, world!” 文本的小程序是一个传统，而在这一部分，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 本身并不对你的编辑器，工具和你的代码存放在何处有什么特定的要求，所以如果你比起命令行更喜欢 IDE，请随意选择你喜欢的 IDE。

创建项目文件夹

首先，创建一个文件夹来编写 Rust 代码。Rust 并不关心你的代码存放在哪里，不过在本书中，我们建议在你的 home 目录创建一个**项目**目录，并把你的所有项目放在这。打开一个终端并输入如下命令来为这个项目创建一个文件夹：

Linux 和 Mac:

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

Windows:

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

编写并运行 Rust 程序

接下来，创建一个新的叫做 *main.rs* 的源文件。Rust 文件总是以 *.rs* 后缀结尾。如果文件名多于一个单词，使用下划线分隔它们。例如，使用 *my_program.rs* 而不是 *myprogram.rs*。

现在打开刚创建的 *main.rs* 文件，并输入如下代码：

Filename: 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，你应该在终端看到`Hello, world!`字符串。如果你做到了，那么恭喜你！你已经正式编写了一个 Rust 程序。你是一名 Rust 程序员了！欢迎入坑。

分析 Rust 程序

现在，让我们回过头来仔细看看你的“Hello, world!”程序到底发生了什么。这是谜题的第一片：

```
fn main() {

}
```

这几行定义了一个 Rust **函数**。`main`函数是特殊的：这是每一个可执行的 Rust 程序首先运行的函数（译者注：入口点）。第一行表示“定义一个叫`main`的函数，没有参数也没有返回值。”如果有参数的话，它们应该出现在括号中，`(`和`)`。

同时注意函数体被包裹在大括号中，`{`和`}`。Rust 要求所有函数体都位于大括号中（译者注：对比有些语言特定情况可以省略大括号）。将前一个大括号与函数声明置于一行，并留有一个空格被认为是一个好的代码风格。

在`main()`函数中：

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

这行代码做了这个小程序的所有工作：它在屏幕上打印文本。有很多需要注意的细节。第一个是 Rust 代码风格使用 4 个空格缩进，而不是 1 个制表符（tab）。

第二个重要的部分是`println!()`。这叫做 Rust **宏**，是如何进行 Rust 元编程（metaprogramming）的关键所在。相反如果调用一个函数的话，它应该看起来像这样：`println`（没有`!`）。我们将在 24 章更加详细的讨论 Rust 宏，不过现在你只需记住当看到符号`!`的时候，就代表在调用一个宏而不是一个普通的函数。

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⊙)

这一行以一个分号结尾（`;`）。`;` 代表这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。

编译和运行是两个步骤

在“编写并运行 Rust 程序”部分，展示了如何运行一个新创建的程序。现在我们将拆分并检查每一步操作。

在运行一个 Rust 程序之前，必须编译它。可以输入 `rustc` 命令来使用 Rust 编译器并像这样传递你源文件的名字：

```
$ rustc main.rs
```

如果你来自 C 或 C++ 背景，你会发现这与 `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 中你可以通过 `ls` 命令看到如下：

```
$ 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*）。这里剩下的操作就只有运行 *main* 或 *main.exe* 文件了，像这样：

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

如果 *main.rs* 是我们的“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，可能不太习惯在分开的步骤编译和执行程序。Rust 是一种 **静态提前编译语言**（*ahead-of-time compiled language*），这意味着可以编译好程序后，把它给任何人，他们都不需要安装 Rust 就可运行。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的权衡取舍。

仅仅使用 `rustc` 编译简单程序是没问题的，不过随着项目的增长，你将想要能够控制你项目拥有的所有选项，并使其易于分享你的代码给别人或别的项目。接下来，我们将介绍一个叫做 Cargo 的工具，它将帮助你编写现实生活中的 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 来管理它们的 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、下载代码依赖的库并编译这些库。我们把代码需要的库叫做 **依赖**（*dependencies*）。

最简单的 Rust 程序，例如我们刚刚编写的，并没有任何依赖，所以目前我们只使用了 Cargo 负责构建代码的部分。随着你编写更加复杂的 Rust 程序，你会想要添加依赖，那么如果你使用 Cargo 开始的话，这将会变得简单许多。

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。如果使用安装章节介绍的官方安装包的话，Rust 自带 Cargo。如果通过其他方式安装 Rust 的话，可以在终端输入如下命令检查是否安装了 Cargo：

```
$ cargo --version
```

如果看到了版本号，一切 OK！如果出现一个类似“`command not found`”的错误，那么你应该查看安装方式的文档来确定如何单独安装 Cargo。

使用 Cargo 创建项目

让我们使用 Cargo 来创建一个新项目并看看与`hello_world`项目有什么不同。回到项目目录（或者任何你决定放置代码的目录）：

Linux 和 Mac:

```
$ cd ~/projects
```

Windows:

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

并在任何操作系统运行：

```
$ 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。可执行文件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（子）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，我们将会看到 Cargo 生成了两个文件和一个目录：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；你可以改为使用不同的版本控制系统，或者不使用，通过`--vcs`参数。

使用你选择的文本编辑器（IDE）打开 *Cargo.toml* 文件。它应该看起来像这样：

Filename: 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，我们还会增加其他部分。

最后一行，`[dependencies]`，是列出项目依赖的 *crates*（我们这么称呼 Rust 代码的包）的部分的开始，这样 Cargo 也就知道去下载和编译它们。这个项目并不需要任何其他的 crate，不过在猜猜看教程章节会需要。

现在看看 *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！目前为止我们所见过的之前项目与 Cargo 生成的项目区别有：

- 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，这样将项目根目录留给 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。

如果没有使用 Cargo 开始项目，正如我们在 *hello_world* 目录中的项目，可以把它转化为一个 Cargo 使用的项目，通过将代码放入 *src* 目录并创建一个合适的 *Cargo.toml*。

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，我们输入如下命令：

```
$ cargo build
   Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
```

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

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

好的！如果一切顺利，`Hello, world!`应该再次打印在终端上。

第一次运行的时候也会使 Cargo 在项目根目录创建一个叫做 *Cargo.lock* 的新文件，它看起来像这样：

Filename: 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，所以内容有一点稀少。事实上，你自己永远也不需要碰这个文件；仅仅让 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`运行了它，不过也可以使用`cargo run`编译并运行：

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

注意这一次，并没有出现告诉我们 Cargo 正在编译`hello_cargo`的输出。Cargo 发现文件并没有被改变，所以只是运行了二进制文件。如果修改了源文件的话，将会出现像这样的输出：

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

所以又出现一些更多的不同：

- 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`
- 不同于将构建结果放在源码相同目录，Cargo 会将它放到 *target/debug* 目录中的文件，我们将会看到

Cargo 的另一个有点是不管你使用什么操作系统它的命令都是一样的，所以之后我们将不再为 Linux 和 Mac 以及 Windows 提供特定的命令。

发布构建

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* 下生成可执行文件，而不是 *target/debug*。这些优化可以让 Rust 代码运行的更快，不过启用他们会让程序花更长的时间编译。这也是为何这是两种不同的配置：一个为了开发，这时你经常想要快速重新构建；另一个构建提供给用户的最终程序，这时并不会重新构建并希望能运行得越快越好。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 并不能比`rustc`提供更多的价值，不过随着开发的进行终将体现它的价值。对于拥有多个 crate 的复杂项目，可以仅仅运行`cargo build`，然后一切将有序运行。即便这个项目很简单，现在它使用了很多接下来你 Rust 程序生涯将会用到的实用工具。事实上，无形中你可以使用下面的命令开始所有你想要从事的项目：

```
$ git clone someurl.com/someproject
$ cd someproject
$ carg
```

> 注意：如果你想要查看 Cargo 的更多细节，请阅读官方的 Cargo guide，它覆盖了其所有的功能。

猜猜看

> ch02-00-guessing-game-tutorial.md
> commit 7c1c935560190fcd64c0851e75dbeabf75fedd19

让我们通过自己动手的方式一起完成一个项目来快速上手 Rust！本章通过展示如何在真实的项目中运用的方式向你介绍一些常用的 Rust 概念。你将会学到 `let`、`match`、方法、关联函数、使用外部 crate 等更多的知识！接下来的章节会探索这些概念的细节。在这一章，我们练习基础。

我们会实现一个经典新手编程问题：猜猜看游戏。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。接着它会提示玩家输入一个猜测。当输入了一个猜测后，它会告诉提示猜测是太大了还是太小了。猜对了，它会打印出祝贺并退出。

准备一个新项目

要创建一个新项目，进入你在第一章创建的**项目**目录，并使用 Cargo 创建它，像这样：

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

第一个命令，`cargo new`，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。

看一样生成的 *Cargo.toml* 文件：

Filename: Cargo.toml

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

[dependencies]
```

如果 Cargo 从环境中获取的作者信息不正确，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：

Filename: src/main.rs

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

现在让我们使用`cargo run`在相同的步骤编译并运行这个“Hello, world!”程序：

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

`run`命令在你需要快速迭代项目时就派上用场了，而这个游戏就正是这么一个项目：我们需要在进行下一步之前快速测试每次迭代。

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

处理一次猜测

程序的第一部分会请求用户输入，处理输入，并检查输入是否为期望的形式。首先，允许玩家输入一个猜测。在 *src/main.rs* 中输入列表 2-1 中的代码。

Filename: 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);
}
```

Listing 2-1: Code to get a guess from the user and print it out

这些代码包含很多信息，所以让我们一点一点地过一遍。为了获取用户输入并接着打印结果作为输出，我们需要将`io`（输入/输出）库引入作用域中。`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* 中引用很少的一些类型。如果想要使用的类型并不在 prelude 中，你必须使用一个`use`语句显式的将其引入到作用域中。使用`std::io`库将提供很多`io`相关的功能，接受用户输入的功能。

正如第一章所讲，`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。

第一章也讲到了，`println!`是一个在屏幕上打印字符串的宏：

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

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

这些代码仅仅打印一个提示，说明游戏的内容并请求用户输入。

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;
```

这行代码会创建一个叫做`foo`的新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：`//`开始一个注释，它持续到本行的结尾。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（`=`）的另一边是`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，它是可增长的、UTF-8 编码的文本块。

`::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的`String`，你会在很多类型上发现`new`函数，因为这是创建某个类型新值的常用函数名。

总结一下，`let mut guess = String::new();`这一行创建了一个可变变量，目前它绑定到一个`String`新的、空的实例上。哟！

回忆一下我们在程序的第一行使用`use std::io;`从标准库中引用输入/输出功能。现在在`io`上调用一个关联函数，`stdin`：

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

如果我们在程序的开头没有`use std::io`这一行，我们可以把函数调用写成`std::io::stdin`这样。`stdin`函数返回一个`std::io::Stdin`的实例，这是一个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`的工作是把获取任何用户键入到标准输入的字符并放入一个字符串中，所以它获取字符串作为一个参数。这个字符串需要是可变的，这样这个方法就可以通过增加用户的输入来改变字符串的内容。

`&`表明这个参数是一个**引用**（*reference*），它提供了一个允许多个不同部分的代码访问同一份数据而不需要在内存中多次拷贝的方法。引用是一个复杂的功能，而 Rust 的一大优势就是它是安全而优雅操纵引用。完成这个程序并不需要知道这么多细节：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。因此，需要写成`&mut guess`而不是`&guess`来使其可变。

这行代码还没有分析完。虽然这是单独一行代码，但它只是一个逻辑上代码行（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，明智的选择是换行并留出空白（缩进）来把长的代码行拆开。我们可以把代码写成这样：

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

不过，过长的代码行难以阅读，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

之前提到过，`read_line`将用户输入放入到传递给它字符串中，不过它也返回一个值——一个`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*。枚举拥有固定值集合的类型，而这些值被称为枚举的**成员**（*variants*）。第六章会更详细的介绍枚举。

对于`Result`，它的成员是`Ok`或`Err`，`Ok`表明操作成功了，同时`Ok`成员之中包含成功生成的值。`Err`意味着操作失败，`Err`之中包含操作是为什么或如何失败的信息。

`Result`类型的作用是编码错误处理信息。`Result`类型的值，正如其他任何类型，拥有定义于其上的方法。`io::Result`的实例拥有`expect`方法可供调用。如果`io::Result`实例的值是`Err`，`expect`会导致程序崩溃并显示显示你作为参数传递给`expect`的信息。如果`io::Result`实例的值是`Ok`，`expect`会获取`Ok`中的值并原原本本的返回给你，这样就可以使用它了。在本例中，返回值是用户输入到标准输入的一些字符。

如果不使用`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)
src/main.rs:10:5: 10:39 warning: unused result which must be used,
#[warn(unused_must_use)] on by default
src/main.rs:10     io::stdin().read_line(&mut guess);
                   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Rust 警告说我们没有使用`read_line`返回的值`Result`，表明程序没有处理一个可能的错误。消除警告的正确方式是老实编写错误处理，不过因为我们仅仅希望程序出现问题就崩溃，可以使用`expect`。你会在第九章学习从错误中恢复。

使用`println!`占位符打印值

除了位于结尾的大括号，目前为止编写的代码就只有一行代码值得讨论一下了，就是这一行：

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

这行代码打印出存储了用户输入的字符串。这对`{}`是一个在特定位置预留值的占位符。可以使用`{}`打印多个值：第一个`{}`对应格式化字符串之后列出的第一个值，第二个对应第二个值，以此类推。用一个`println!`调用打印多个值应该看起来像这样：

```
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`运行它：

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
     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：我们从键盘获取了输入并打印了出来。

生成一个秘密数字

接下来，需要生成一个秘密数字，用户会尝试猜测它。秘密数字应该每次都不同，这样多玩几次才会有意思。生成一个 1 到 100 之间的随机数这样游戏也不会太难。Rust 标准库中还未包含随机数功

能。然而，Rust 团队确实提供了一个 `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*，它包含意在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。在我们可以使用 `rand` 编写代码之前，需要编辑 *Cargo.toml* 来包含 `rand` 作为一个依赖。现在打开这个文件并在 `[dependencies]` 部分标题（Cargo 为你创建了它）的下面添加如下代码：

Filename: Cargo.toml

```
[dependencies]

rand = "0.3.14"
```

在 *Cargo.toml* 文件中，任何标题之后的内容都是属于这个部分的，一直持续到直到另一个部分开始。 `[dependencies]` 部分告诉 Cargo 项目依赖了哪个外部 crate 和需要的 crate 版本。在这个例子中，我们使用语义化版本符号 `0.3.14` 来指定 `rand` crate。Cargo 理解语义化版本（Semantic Versioning）（有时也称为 *SemVer*），这是一个编写版本号的标准。版本号 `0.3.14` 事实上是 `^0.3.14` 的缩写，它的意思是“任何与 0.3.14 版本公有 API 相兼容的版本”。

现在，不用修改任何代码，构建项目，如列表 2-2：

```
$ 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)
```

Listing 2-2: The output from running `cargo build` after adding the rand crate as a dependency

可能会出现不同的版本号（不过多亏了语义化版本，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*（Crates.io）上获取了一份（兼容的）最新版本代码的拷贝。Crates.io 是 Rust 生态环境中的人们向他人贡献他们的开源 Rust 项目的地方。

在更新完 registry（索引）后，Cargo 检查 `[dependencies]` 部分并下载还不存在部分。在这个例子中，虽然只列出了 `rand` 一个依赖，Cargo 也获取了一份 `libc` 的拷贝，因为 `rand` 依赖 `libc` 来正常工作。在下载他们之后，Rust 编译他们接着用这些依赖编译项目。

如果不做任何修改就立刻再次运行 `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* 文件中也没有任何相关修改。Cargo 也知道代码没有做任何修改，所以它也不会重新编译代码。因为无事可做，它简单的退出了。如果打开 *src/main.rs* 文件，并做一些普通的修改，保存并再次构建，只会出现一行输出：

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
```

这一行表明 Cargo 只构建了对 *src/main.rs* 文件做出的微小修改。依赖没有被修改，所以 Cargo 知道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。

The *Cargo.lock* 文件确保构建是可重现的

Cargo 有一个机制来确保每次任何人重新构建代码都会生成相同的成品：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，而它包含一个重要的 bug 修改并也含有一个会破坏代码运行的缺陷的时候会发生什么呢？

这个问题的答案是 *Cargo.lock* 文件，它在第一次运行 `cargo build` 时被创建并位于 *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接着写入 *Cargo.lock* 文件中。当将来构建项目时，Cargo 发现 *Cargo.lock* 存在就会使用这里指定的版本，而不是重新进行所有版本的计算。这使得你拥有了一个自动的可重现的构建。换句话说，项目会继续使用 `0.3.14` 直到你显式升级，多亏了 *Cargo.lock* 文件。我们将会在这个文件编写全部的代码。

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* 中规格的最新版本。
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，值得注意的是 *Cargo.lock* 文件中的一个改变，`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 会更新 registry 中可用的 crate 并根据你指定新版本重新计算 `rand` 的要求。

第十四章会讲到Cargo 和它的生态系统的更多内容，不过目前你只需要了解这么多。Cargo 使得复用库文件变得非常容易，所以 Rustacean 们能够通过组合很多包来编写出更轻巧的项目。

生成一个随机数

让我们开始**使用** `rand`。下一步是更新 *src/main.rs*，如列表 2-3：

Filename: 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);
}
```

Listing 2-3: Code changes needed in order to generate a random number

我们在顶部增加一行`extern crate rand;`来让 Rust 知道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`中的任何内容。

接下来，我们增加了另一行`use`：`use rand::Rng`。`Rng`是一个定义了随机数生成器应实现方法的 trait，如果要使用这些方法的话这个 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`函数会提供具体会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。这个方法由我们使用`use rand::Rng`语句引入到作用域的`Rng` trait 定义。`gen_range`方法获取两个数作为参数并生成一个两者之间的随机数。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。

并不仅仅能够知道该 use 哪个 trait 和该从 crate 中调用哪个方法。如何使用 crate 的说明在每个 crate 的文档中。Cargo 另一个很棒的功能是可以运行`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，如果你对`rand` crate 中的其他功能感兴趣，运行`cargo doc --open`并点击左侧导航栏的`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，因为我们可以去测试它，不过在最终版本我们会删掉它。游戏一开始就打印出结果就没什么可玩的了！

尝试运行程序几次：

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
     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，同时他们应该都是在 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，我们可以比较他们。这个步骤如列表 2-4：

Filename: 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!"),
    }
}
```

Listing 2-4: Handling the possible return values of comparing two numbers

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`的类型到作用域。`Ordering`是另一个枚举，像`Result`一样，不过`Ordering`的成员是`Less`、`Greater`和`Equal`。这是你比较两个值时可能出现三种结果。

接着在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` 方法比较两个值并可以在任何可比较的值上调用。它获取一个任何你想要比较的值的引用：这里是把 `guess` 与 `secret_number` 做比较。 `cmp` 返回一个使用 `use` 语句引入作用域的 `Ordering` 枚举的成员。我们使用一个 `match` 表达式根据对 `guess` 和 `secret_number` 中的值调用 `cmp` 后返回的哪个 `Ordering` 枚举成员来决定接下来干什么。

一个 `match` 表达式由 **分支（arms）** 构成。一个分支包含一个 **模式**（*pattern*）和代码，这些代码在 `match` 表达式开头给出的值符合分支的模式时将被执行。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 中非常强大的功能，它帮助你体现代码可能遇到的多种情形并帮助你处理全部的可能。这些功能将分别在第六章和第十九章详细介绍。

让我们看看一个使用这里的 `match` 表达式会发生什么的例子。假设用户猜了 50，这时随机生成的秘密数字是 38。当代码比较 50 与 38 时， `cmp` 方法会返回 `Ordering::Greater` ，因为 50 比 38 要大。 `Ordering::Greater` 是 `match` 表达式得到的值。它检查第一个分支的模式， `Ordering::Less` ，不过值 `Ordering::Greater` 并不匹配 `Ordering::Less` 。所以它忽略了这个分支的代码并移动到下一个分支。下一个分支的模式， `Ordering::Greater` ，**正确**匹配了 `Ordering::Greater` ！这个分支关联的代码会被执行并在屏幕打印出 `Too big!` 。 `match` 表达式就此终止，因为在这个特定场景下没有检查最后一个分支的必要。

然而，列表 2-4 的代码并不能编译，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 拥有一个静态强类型系统。不过，它也有类型推断。当我们写出 `let guess = String::new()` 时，Rust 能够推断出 `guess` 应该是一个 `String` ，并不需要我们写出类型。另一方面， `secret_number` ，是一个数字类型。一些数字类型拥有 1 到 100 之间的值： `i32` ，一个 32 位的数字； `u32` ，一个 32 位无符号数字； `i64` ，一个 64 位数字；等等。Rust 默认使用 `i32` ，所以 `secret_number` 的类型就是它，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。这里错误的原因是 Rust 不会比较字符串类型和数字类型。

最终我们想要把程序从输入中读取到的 `String` 转换为一个真正的数字类型，这样好与秘密数字向比较。可以通过在 `main` 函数体中增加如下两行代码来实现：

Filename: 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，难道这个程序不是已经有了一个叫做`guess`的变量了吗？确实如此，不过 Rust 允许我们通过**覆盖**（*shadow*）用一个新值来覆盖`guess`之前的值。这个功能经常用在类似需要把一个值从一种类型转换到另一种类型的场景。shadowing 允许我们复用`guess`变量的名字而不是强迫我们创建两个不同变量，比如`guess_str`和`guess`。（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`对应包含输入的`String`类型的原始`guess`。`String`实例的`trim`方法会消除字符串开头和结尾的空白。`u32`只能包含数字字符。不过用户必须输入回车键才能让`read_line`返回。当用户按下回车键时，会在字符串中增加一个换行（newline）字符。例如，如果用户输入 5 并回车，`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`方法解析一个字符串成某个数字。因为这个方法可以解析多种数字类型，需要告诉 Rust 我们需要的具体的数字类型，这里通过`let guess: u32`指定。`guess`后面的冒号（`:`）告诉 Rust 我们指明了变量的类型。Rust 有一些内建的数字类型；这里的`u32`是一个无符号的 32 位整型。它是一个好的较小正整数的默认类型。第三章会讲到其他数字类型。另外，例子程序中的`u32`注解和与`secret_number`的比较意味着 Rust 会推断`secret_number`应该是也是`u32`类型。现在可以使用相同类型比较两个值了！

`parse`调用容易产生错误。例如，如果字符串包含`A👍%`，就无法将其转换为一个数字。因为它可能失败，`parse`方法返回一个`Result`类型，非常像之前在 XX 页“使用`Result`类型来处理潜在的错误”部分讨

论的`read_line`方法。这里再次类似的使用`expect`方法处理这个`Result`类型。如果`parse`因为不能从字符串生成一个数字而返回一个`Err`的`Result`成员时，`expect`会使游戏崩溃并打印提供给它的信息。如果`parse`能成功地将字符串转换为一个数字，它会返回`Result`的`Ok`成员，同时`expect`会返回`Ok`中我们需要的数字。

现在让我们运行程序！

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
     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。增加它后给了用户多次猜测的机会：

Filename: 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。确保这些代码多缩进了四个空格，并再次运行程序。注意这里有一个新问题，因为程序忠实地执行了我们要求它做的：永远地请求另一个猜测！看起来用户没法退出啊！

用户总是可以使用`Ctrl-C`快捷键来终止程序。不过这里还有另一个逃离这个贪得无厌的怪物的方法，就是在 XX 页“比较猜测”部分提到的`parse`：如果用户输入一个非数字回答，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`就会退出程序，同时其他任何非数字输入也一样。然而，毫不夸张的说这是不理想的。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`来在用户胜利时退出游戏：

Filename: 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`，程序在用户猜对了神秘数字后会退出循环。退出循环也就意味着退出程序，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，而不是在用户输入非数字时崩溃，需要让游戏忽略非数字从而用户可以继续猜测。可以通过修改`guess`从`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,
};
```

从 `expect` 调用切换到 `expect` 语句是如何从遇到错误就崩溃到真正处理错误的常用手段。记住 `parse` 返回一个 `Result` 类型，而 `Result` 是一个拥有 `Ok` 或 `Err` 两个成员的枚举。在这里使用 `match` 表达式，就像之前处理 `cmp` 方法返回的 `Ordering` 一样。

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 `Ok` 值。这个 `Ok` 值会匹配第一个分支的模式，这时 `match` 表达式仅仅返回 `parse` 产生的 `Ok` 值之中的 `num` 值。这个数字会最终如期变成新创建的 `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` 值。 `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。 `_` 是一个包罗万象的值；在这个例子中，我们想要匹配所有 `Err` 值，不管其中有何种信息。所以程序会执行第二个分支的代码， `continue` ，这意味着进入 `loop` 的下一次循环并请求另一个猜测。这样程序就有效地忽略了 `parse` 可能遇到的所有错误！

现在万事俱备（只欠东风）了。运行 `cargo run` 来尝试一下：

```
$ 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，就能完成猜猜看游戏了：还记得程序依然会打印出秘密数字。这在测试时还好，但会毁了游戏性。删掉打印秘密数字的 `println!` 。列表 2-5 为最终代码：

Filename: 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;
            }
        }
    }
}
```

Listing 2-5: Complete code of the guessing game

总结一下，

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的方式想你介绍许多 Rust 新知识的项目：`let`、`match`、方法、关联函数，使用外部 crate，等等。接下来的几章，我们将会详细学习这些概念。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 中使用他们。第四章探索所有权（ownership），这是一个 Rust 同其他语言都不相同的功能。第五章讨论结构体和方法语法，而第六章侧重解释枚举。

通用编程概念

---

ch03-00-common-programming-concepts.md
commit 2067b6e2bff990bceb39ae8f35780bd3bed08644

这一章涉及到几乎出现在所有编程语言中的概念，以及他们在 Rust 中如何工作。很多编程语言在核心概念上都是共通的。本章中展示的所有概念没有一个是 Rust 所特有的，不过我们会在 Rust 环境中讨论他们并解释他们的使用习惯。

具体的，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使你在起步时拥有一个核心的基础。

> **关键字**
>
> Rust 语言有一系列被保留为只能被语言使用的**关键字**（*keywords*），如大部分语言一样。注意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，并将会被用来进行 Rust 程序中的多种任务；一些关键字目前没有相关的功能不过为了将来可能添加进 Rust 的功能而被保留。可以在附录 A 中找到一份关键字的列表

变量和可变性

> ch03-01-variables-and-mutability.md
> commit b0fab378c9c6a817d4f0080d7001d085017cdef8

第二章中提到过，变量默认是**不可变**（*immutable*）的。这是 Rust 中许多鼓励以利用 Rust 提供的安全和简单并发优势编写代码的助力之一。不过，仍然有使变量可变的选项。让我们探索一下为什么以及如何鼓励你拥抱不可变性，还有为什么你可能想要弃之不用。

当变量使不可变时，这意味着一旦一个值被绑定上了一个名称，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：

Filename: 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：

```
$ cargo run
   Compiling variables v0.0.1 (file:///projects/variables)
error[E0384]: re-assignment of 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;
  |     ^^^^^ re-assignment of immutable variable
```

这个例子显示了编译器如何帮助你寻找程序中的错误。即便编译器错误可能是令人沮丧的，他们也仅仅意味着程序不能安全的完成你想让它完成的工作；他们**不能**说明你不是一个好的程序员！有经验的 Rustacean 们也会遇到编译器错误。这些错误表明错误的原因是`对不可变变量重新赋值`（`re-assignment of immutable variable`），因为我们尝试对不可变变量`x`赋第二个值。

当尝试去改变之前设计为不可变的值出现编译时错误是很重要的，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，这样第一部分代码就有可能不能像它设计的那样运行。你必须承认这种 bug 难以跟踪，尤其是当第二部分代码只是**有时**当变量使不可变时，这意味着一旦一个值被绑定上了一个名称，你就不能改变这个值。

Rust 编译器保证如果声明一个值不会改变，它就真的不会改变。这意味着当阅读和编写代码时，并不需要记录如何以及在哪可能会被改变，这使得代码易于推导。

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增加`mut`来使其可变。它向之后的读者表明了其他部分的代码将会改变这个变量值的意图。

例如，改变 *src/main.rs* 并替换其代码为如下：

Filename: 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)
    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，你会想要使一个变量可变，因为这比只使用不可变变量实现的代码更易于编写。

除了避免 bug 外，这里还有数个需要权衡取舍的地方。例如，有时使用大型数据结构时，适当地使变量可变可能比复制和返回新分配的实例要更快。对于较小的数据结构，总是创建新实例并采用一种更函数式的编程风格可能会使代码更易理解。所以为了可读性而造成的性能惩罚也许使值得的。

变量和常量的区别

不能改变一个变量的值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*）。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`：常量不光是默认不能改变，它总是不能改变。常量使用`const`关键字而不是`let`关键字声明，而且*必须*注明值的类型。现在我们准备在下一部分，“数据类型”，涉及到类型和类型注解，所以现在无需担心这些细节。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使用到的值。

这是一个常量声明的例子，它的名称是`MAX_POINTS`而它的值是 100,000。Rust 常量的命名规范是使用大写字母和单词间使用下划线：

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

常量在整个程序生命周期中都有效，位于它声明的作用域之中。这使得常量可以用作多个部分的代码可能需要知道的程序范围的值，例如一个游戏中任何玩家可以获得的最高分或者一年的秒数。

将用于整个程序的硬编码的值命名为常量（并编写文档）对为将来代码维护者表明值的意义是很有用的。它也能帮助你将硬编码的值至于一处以便将来可能需要修改他们。

覆盖

如第二章猜猜看游戏所讲到的，我们可以定义一个与之前变量名称相同的新变量，而新变量会**覆盖**之前的变量。Rustacean 们称其为第一个变量被第二个**给覆盖**了，这意味着第二个变量的值是使用这个变量时会看到的值。可以用相同变量名称来覆盖它自己以及重复使用`let`关键字来多次覆盖，如下所示：

Filename: 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 =`覆盖`x`，获取原始值并加`1`这样`x`的值就变成`6`了。第三个`let`语句也覆盖了`x`，获取之前的值并乘以`2`，`x`的最终值是`12`。当运行这个程序，它会有如下输出：

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
     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。

另一个`mut`与覆盖的区别是当再次使用`let`关键字时，事实上创建了一个新变量，我们可以改变值的类型。例如，假设程序请求用户输入空格来提供在一些文本之间需要多少空间来分隔，不过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名字相同的崭新的变量，它是数字类型。因此覆盖使我们不必使用不同的名字，比如`spaces_str`和`spaces_num`；相反，我们可以复用`spaces`这个更简单的名称。然而，如果尝试使用`mut`，如下所示：

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

会导致一个编译时错误，因为不允许改变一个变量的类型：

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

现在我们探索了变量如何工作，让我们看看他们能有多少数据类型。

数据类型

ch03-02-data-types.md
commit 6436ebee2a84820adf77231cead6b5691c8e2744

Rust 中的任何值都有一个具体的**类型**（*type*），这告诉了 Rust 它被指定为何种数据这样 Rust 就知道如何处理这些数据了。这一部分将讲到一些语言内建的类型。我们将这些类型分为两个子集：标量（scalar）和复合（compound）。

贯穿整个部分，请记住 Rust 是一个**静态类型**（*statically typed*）语言，也就是说必须在编译时就知道所有变量的类型。编译器通常可以通过值以及如何使用他们来推断出我们想要用的类型。当多个类型都是可能的时候，比如第二章中`parse`将`String`转换为数字类型，必须增加类型注解，像这样：

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

如果这里不添加类型注解，Rust 会显示如下错误，它意味着编译器需要我们提供更多我们想要使用哪个可能的类型的信息：

```
error[E0282]: unable to infer enough type information about `_`
 --> src/main.rs:2:5
  |
2 | let guess = "42".parse().expect("Not a number!");
  |     ^^^^^ cannot infer type for `_`
  |
  = note: type annotations or generic parameter binding required
```

在我们讨论各种数据类型时会看到不同的类型注解。

标量类型

标量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他们，不过让我们深入了解他们在 Rust 中时如何工作的。

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一个整型，`i32`类型。这个类型声明表明在 32 位系统上它关联的值应该是一个有符号整数（因为这个`i`，与`u`代表的无符号相对）。表格 3-1 展示了 Rust 内建的整数类型。每一个变体的有符号和无符号列（例如，*i32*）可以用来声明对应的整数值。

Table 3-1: Integer Types in Rust

Length	Signed	Unsigned
8-bit	i8	u8
16-bit	i16	u16
32-bit	i32	u32
64-bit	i64	u64
arch	isize	usize

每一种变体都可以是有符号或无符号的并有一个显式的大小。有符号和无符号代表数字是否能够是正数或负数；换句话说，数字是否需要有一个符号（有符号数）或者永远只需要是正的这样就可以不用符号（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，数字前面会加上一个加号或减号；然而，当可以安全地假设为正数时，可以不带符号（加号）。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类型（构架）：64 位构架他们是 64 位的而 32 位构架他们就是 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。注意除了字节字面值以外的数字字面值允许使用类型后缀，例如`57u8`，而`_`是可视化分隔符（visual separator），例如`1_000`位的。

Table 3-2: Integer Literals in Rust

Number literals	Example
Decimal	`98_222`
Hex	`0xff`
Octal	`0o77`
Binary	`0b1111_0000`
Byte (`u8` only)	`b'A'`

那么如何知晓该使用哪种类型的数字呢？如果对此拿不定主意，Rust 的默认类型通常就是一个很好的选择，这个默认数字类型是`i32`：它通常是最快的，甚至是在 64 位系统上。使用`isize`或`usize`的主要场景是索引一些集合。

浮点型

Rust 也有两个主要的**浮点数**（*floating-point numbers*）类型，他们是有小数点的数字。Rust 的浮点数类型是`f32`和`f64`，分别是 32 位和 64 位大小。默认类型是`f64`，因为它基本上与`f32`一样快不过精度更高。在 32 位系统上使用`f64`是可能的，不过会比`f32`要慢。大部分情况，牺牲潜在可能的更低性能来换取更高的精度是一个合理的首要选择，同时如果怀疑浮点数的大小有问题的时候应该对代码进行性能测试。

这是一个展示浮点数的实例：

Filename: 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、除法以及余数。如下代码展示了如何使用一个`let`语句来使用他们：

Filename: src/main.rs

```rust
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，他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：

Filename: src/main.rs

```rust
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语句，例如`if`。在“控制流”（“Control Flow”）部分将讲到`if`语句在 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：

Filename: src/main.rs

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

Rust 的`char`类型代表了一个 Unicode 变量值（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/汉语等象形文字，emoji（絵文字）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`之间的值。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组要方式。

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，而且这写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：

Filename: 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：

Filename: 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。例如：

Filename: 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，他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：

Filename: src/main.rs

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

数组在想要在栈（stack）而不是在堆（heap）上为数据分配空间时十分有用（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。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。程序不大可能回去增加或减少月份，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：

Filename: 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。`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为如下：

Filename: src/main.rs

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

    let element = a[10];

    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)
     Running `target/debug/arrays`
thread '<main>' panicked at 'index out of bounds: the len is 5 but the index is
 10', src/main.rs:4
note: Run with `RUST_BACKTRACE=1` for a backtrace.
error: Process didn't exit successfully: `target/debug/arrays` (exit code: 101)
```

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 52b7fcbfdd35915cb21e6d492fb6c86764f53b47

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，它时很多程序的入口点。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。这里是一个函数定义程序的例子：

Filename: 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`在程序中已经定义过了，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；也可以在之前定义。Rust 不关心函数定义于何处，只要他们被定义了。

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)
     Running `target/debug/functions`
Hello, world!
Another function.
```

代码在`main`函数中按照他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*），他们是作为函数签名一部分的特殊变量。当函数拥有参数，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：

Filename: 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)
     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再也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，使用逗号隔开他们，像这样：

Filename: 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。注意函数参数并一定都是相同的————这个例子中他们只是碰巧相同。

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

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
     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`的值和`6`作为`y`的值来调用函数，这两个字符串使用这些值并被打印出来。

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他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 3-3 中，`let y = 6;`是一个语句：

Filename: src/main.rs

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

Listing 3-3: A `main` function declaration containing one statement.

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：

Filename: 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，他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。

表达式进行计算而且他们组成了其余大部分 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。我们用来创新建作用域的大括号（代码块），`{}`，也是一个表达式，例如：

Filename: 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
}
```

这个代码块的值是`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

可以向调用它的代码返回值。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：

Filename: 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`它子集。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)
    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;
```

再次，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：

Filename: 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？

```
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[E0269]: not all control paths return a value
 --> src/main.rs:7:1
  |
7 | fn plus_one(x: i32) -> i32 {
  | ^
  |
help: consider removing this semicolon:
 --> src/main.rs:8:10
  |
8 |     x + 1;
  |          ^
```

主要的错误信息，“并非所有控制路径都返回一个值”（“not all control paths return a value,”），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。因此，这个函数没有返回任何值，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 74d6fc999b986b74bf94edd6dcbb5a08a16c12de

所有编程语言都力求使他们的代码易于理解，不过有时额外的解释需要得到保障。在这种情况下，程序员在源码中留下记录，或者**注释**（*comments*），编译器会忽略他们不过其他阅读代码的人可能会用得上。

这是一个注释的例子：

```
// 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结尾：

Filename: src/main.rs

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

不过你会经常看到他们被以这种格式使用，也就是位于它解释的代码行的上面一行：

Filename: src/main.rs

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

这就是注释的全部。并没有什么特别复杂的。

控制流

ch03-05-control-flow.md
commit 784a3ec5e8b9c6bff456ab9f0efd4dabcc180dda

通过条件是不是真来决定是否某些代码，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：

Filename: src/main.rs

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

    if number < 5 {
        println!("condition was true");
    } else {
        println!("condition was false");
    }
}
```

所有`if`表达式以`if`关键字开头，它后跟一个条件。在这个例子中，条件检查`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，这里我们就这么做了，来提供一个在条件为假时应当执行的代码块。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)
     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)
     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`。如果像看看条件不是`bool`值时会发生什么，尝试运行如下代码：

Filename: 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：

```
   Compiling branches v0.1.0 (file:///projects/branches)
error[E0308]: mismatched types
 --> src/main.rs:4:8
  |
4 |     if number {
  |        ^^^^^^ expected bool, found integral variable
  |
  = note: expected type `bool`
  = note:    found type `{integer}`

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

这个错误表明 Rust 期望一个`bool`不过却得到了一个整型。Rust 并不会尝试自动地将非布尔值转换为布尔值，不像例如 Ruby 和 JavaScript 这样的语言。必须总是显式地使用`boolean`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`表达式修改为如下：

Filename: 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：

Filename: 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)
    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，并且它一旦找到一个以后，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。为此第六章介绍了 Rust 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，例如列表 3-4：

Filename: src/main.rs

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

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

Listing 3-4: Assigning the result of an `if` expression to a variable

`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)
    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；在列表 3-4 中，`if` 分支和 `else` 分支的结果都是 `i32` 整型。不过如果像下面的例子一样这些类型并不相同会怎么样呢？

Filename: 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 也准确地表明了在程序中的何处发现的这个问题：

```
   Compiling branches v0.1.0 (file:///projects/branches)
error[E0308]: if and else have incompatible types
 --> src/main.rs:4:18
  |
4 |     let number = if condition {
  |                  ^ expected integral variable, found reference
  |
  = note: expected type `{integer}`
  = note:    found type `&'static 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完成这些功能；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

多次执行一段代码是很常用的。为了这个功能，Rust 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从回到开头继续执行。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 一遍又一遍的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：

Filename: 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)
     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`again!`也可能看不到，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。还记得我们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吗。

`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，这个模式太常见了所以 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：

Filename: 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：

Filename: 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;
    }
}
```

Listing 3-5: Looping through each element of a collection using a `while` loop

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)
     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替代。`for`循环看起来像这样：

Filename: src/main.rs

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

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

Listing 3-6: Looping through each element of a collection using a `for` loop

当运行这段代码，将看到与列表 3-5 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 机会。

例如，在列表 3-5 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，例如列表 3-5 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：

Filename: 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！这是一个相当可观的章节：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 759067b651a48a4a66485fe0876d318d398fb4fe

所有权（系统）是 Rust 最独特的功能，它令 Rust 可以无需垃圾回收（garbage collector）就能保障内存安全。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、slices 以及 Rust 如何在内存中安排数据。

什么是所有权

ch04-01-what-is-ownership.md
commit cc053d91f41793e54d5321abe027b0c163d735b8

Rust 的核心功能（之一）是**所有权**（*ownership*）。虽然这个功能理解起来很直观，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他们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 这样的系统变成语言中，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特定的选择。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是一个已知的固定的大小。

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，并返回给我们一个它位置的指针。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。现代的处理器在内存中跳转越少就越快。继续类比，假设有一台服务器来处理来自多个桌子的订单。它在处理完一个桌子的所有订单后再移动到下一个桌子是最有效率的。从桌子 A 获取一个订单，接着再从桌子 B 获取一个订单，然后再从桌子 A，然后再从桌子 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用完空间，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什么存在以及为什么以它的方式工作。

所有权规则

首先，让我们看一下所有权的规则。记住这些规则正如我们将完成一些说明这些规则的例子：

---

1. Rust 中的每一个值都有一个叫做它的**所有者**（*owner*）的变量。
2. 同时一次只能有一个所有者
3. 当所有者变量离开作用域，这个值将被丢弃。

变量作用域

我们在第二章已经完成过一个 Rust 程序的例子了。现在我们已经掌握了基本语法，所以不会在所有的例子中包含`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，例子将显得更加具体，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 item 在程序中有效的范围。假如有一个这样的变量：

```
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我们程序代码中的。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。列表 4-1 的注释标明了变量`s`在哪里是有效的：

```
{                      // 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
```

Listing 4-1: A variable and the scope in which it is valid

换句话说，这里有两个重要的点：

1. 当`s`**进入作用域**，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变成语言是类似的。现在我们要在此基础上介绍`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。之前出现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 如何知道该在何时清理数据。

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，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储存在编译时未知大小的文本。可以用`from`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

字符串字面值的情况，我们在编译时就知道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和高效。不过这些属性都只来源于它的不可变形。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，它的实现请求它需要的内存。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（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：内存在拥有它的变量离开作用域后就被自动释放。下面是列表 4-1 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`函数不陌生。

这个模式对编写 Rust 代码的方式有着深远的影响。它现在看起来很简单，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：移动

Rust 中的多个变量以一种独特的方式与同一数据交互。让我们看看列表 4-2 中一个使用整型的例子：

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

Listing 4-2: Assigning the integer value of variable `x` to `y`

根据其他语言的经验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 4-3 中`String`真正是什么样。`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。

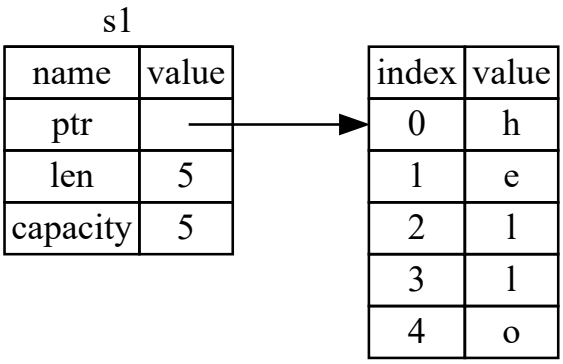

Figure 4-3: Representation in memory of a `String` holding the value `"hello"` bound to `s1`

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 4-4 所示。

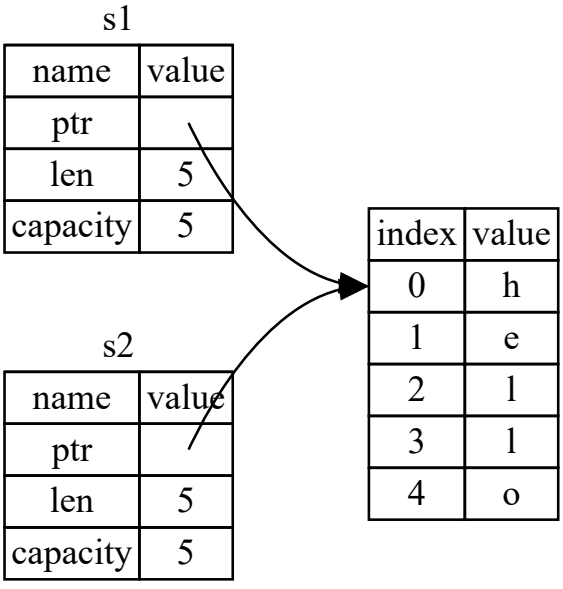

Figure 4-4: Representation in memory of the variable `s2` that has a copy of the pointer, length, and capacity of `s1`

这个表现形式看起来**并不像**图 4-5 中的那样，它是如果 Rust 也拷贝了堆上的数据后内存看起来是怎么样的。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。

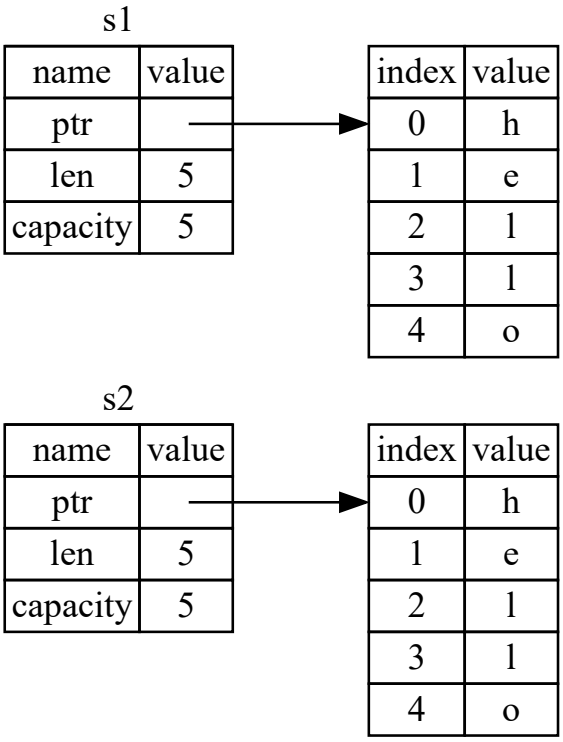

Figure 4-5: Another possibility of what `s2 = s1` might do if Rust copied the heap data as well

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 *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`会发生生么：

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

println!("{}", 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:4:27
  |
3 |     let s2 = s1;
  |         -- value moved here
4 |     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，这个操作被成为**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 4-6 所示。

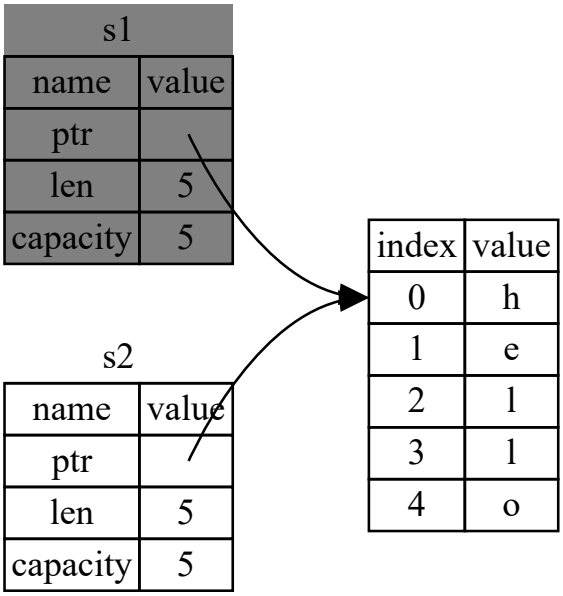

Figure 4-6: Representation in memory after `s1` has been invalidated

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：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 4-5 中行为的方式，这里堆上的数据**被复制了**。

当出现`clone`调用时，你知道一些特有的代码被执行而且这些代码可能相当消耗资源。所以它作为一个可视化的标识代表了不同的行为。

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，他们是之前列表 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向抵触：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（重新）赋值后仍然可用。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。

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，任何不需要分配内存或类似形式资源的类型是`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言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。列表 4-7 是一个带有变量何时进入和离开作用域标注的例子：

Filename: 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.
```

Listing 4-7: Functions with ownership and scope annotated

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，和哪里所有权规则会阻止我们这么做。

返回值与作用域

返回值也可以转移作用域。这里是一个有与列表 4-7 中类似标注的例子：

Filename: 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，并且当变量值的堆书卷离开作用域时，如果数据的所有权没有被移动到另外一个变量时，其值将通过`drop`被清理掉。

在每一个函数中都获取并接着返回所有权是冗余乏味的。如果我们想要函数使用一个值但不获取所有权改怎么办呢？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。

使用元组来返回多个值是可能的，像这样：

Filename: 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)
}
```

但是这不免有些形式主义，同时这离一个通用的观点还有很长距离。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 c9fd8eb1da7a79deee97020e8ad49af8ded78f9c

在上一部分的结尾处的使用元组的代码是有问题的，我们需要将`String`返回给调用者函数这样就可以在调用`calculate_length`后仍然可以使用`String`了，因为`String`先被移动到了`calculate_length`。

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。

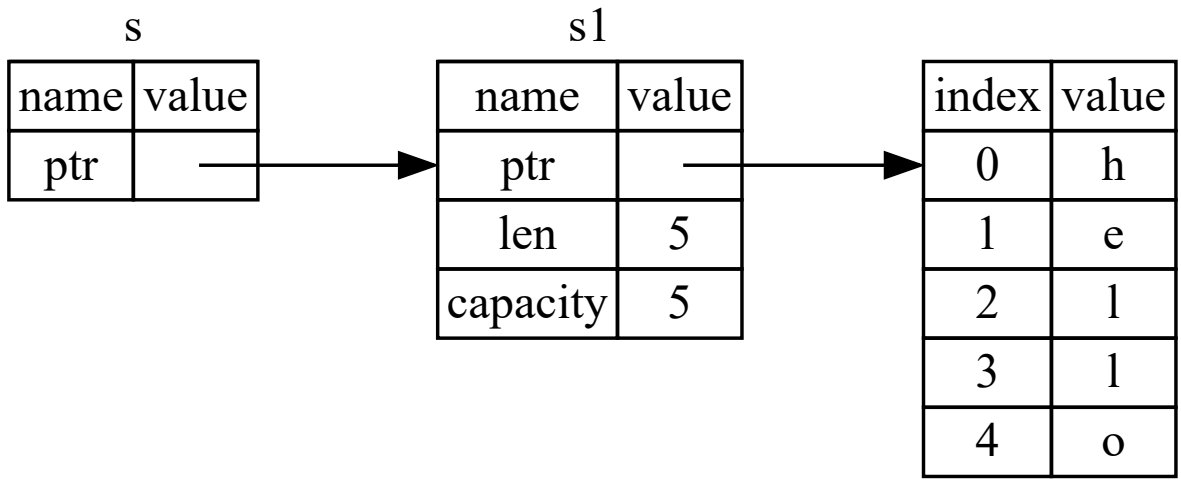

Figure 4-8: `&String s` pointing at `String s1`

仔细看看这个函数调用：

```
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，因为就不曾拥有它。

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，你可以从它哪里借来。当你使用完毕，必须还回去。

那么如果我们尝试修改借用的变量呢？尝试列表 4-9 中的代码。剧透：这行不通！

Filename: src/main.rs

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

    change(&s);
}

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

Listing 4-9: Attempting to modify a borrowed value

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列表 4-9 代码中的错误，在列表 4-9 的代码中：

Filename: 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：

Filename: src/main.rs

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

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

具体错误如下：

```
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（data races）。

数据竞争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相同的数据。
2. 至少有一个指针被用来写数据。
3. 没有被用来同步数据访问的机制。

数据竞争会导致未定义行为并且当在运行时尝试追踪时可能会变得难以诊断和修复；Rust 阻止了这种情况的发生，因为存在数据竞争的代码根本就不能编译！

一如既往，使用大括号来创建一个新的作用域，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读取数据的人有能力影响其他人读取到的数据。

即使这些错误有时是使人沮丧的。记住这是 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，容易错误地生成一个**悬垂指针**（*dangling pointer*），一个引用某个内存位置的指针，这个内存可能已经因为被分配给别人，因为释放内存时指向内存的指针被保留了下来。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，如果你不理会生命周期的部分，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`！这可不好。Rust 不会允许我们这么做的。

正确的代码是直接返回`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. 特定时间，**只能**拥有如下中的一个：

- 一个可变引用。
- 任意属性的不可变引用。

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：slices。

Slices

---

ch04-03-slices.md
commit c9fd8eb1da7a79deee97020e8ad49af8ded78f9c

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。让我们试试如列表 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()
> }
> ```
>
> Listing 4-10: The `first_word` function that returns a byte index value into the `String` parameter

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();
```

Next, we create an iterator over the array of bytes using the `iter` method :

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

第十六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。这比我们自己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，我们在模式中使用了`&`。

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。我们返回了单单一个`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`像分离的值，无法保证将来它仍然有效。考虑一下列表 4-11 中使用了列表 4-10 `first_word`函数的程序：

Filename: src/main.rs

```
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!
}
```

Listing 4-11: Storing the result from calling the `first_word` function then changing the `String` contents

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。

不得不担心`word`的索引与`s`中的数据不再同步是乏味且容易出错的！如果编写一个`second_word`函数的话管理索引将更加容易出问题。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。

我们使用一个 range `[starting_index..ending_index]`来创建 slice，不过 slice 的数据结构实际上储存了开始位置和 slice 的长度。所以就`let world = &s[6..11];`来说，`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

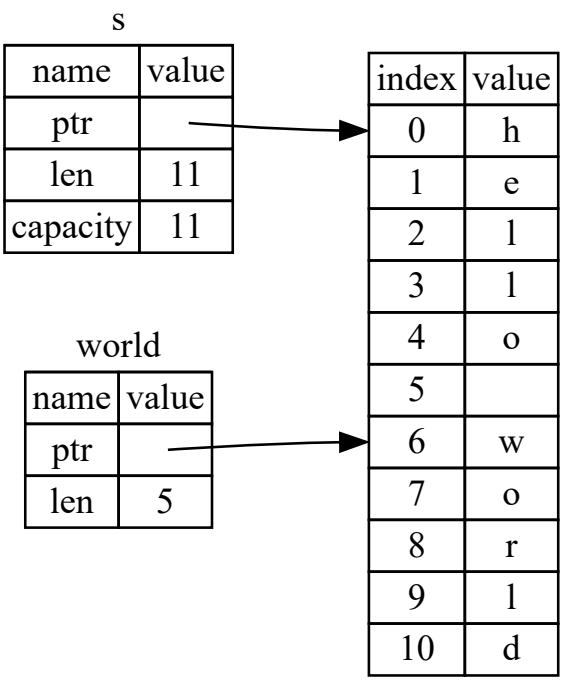

Figure 4-12: String slice referring to part of a `String`

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[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”的签名写作`&str`：

Filename: 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[..]
}
```

我们使用跟列表 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。还记得列表 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？那些代码逻辑上时不正确的，不过却没有任何直观的错误。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。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个错误类型！

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 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，所以我们会继续讲到这些概念，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。

结构体

---

ch05-00-structs.md
commit 255b44b409585e472e14c396ebc75d28f540a1ac

`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）。在这一章的下一部分会讲到如何在结构体上定义方法；方法是如何为结构体数据指定**行为**的函数。`struct`和`enum`（将在第六章讲到）是为了充分利用 Rust 的编译时类型检查，来在程序范围创建新类型的基本组件。

对结构体的一种看法是他们与元组类似，这个我们在第三章讲过了。就像元组，结构体的每一部分可以是不同类型。可以命令各部分数据所以能更清楚的知道其值是什么意思。由于有了这些名字使得结构体更灵活：不需要依赖顺序来指定或访问实例中的值。

为了定义结构体，通过`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，他们被称作**字段**（*fields*），并定义字段类型。例如，列表 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,
}
```

Listing 5-1: A `User` struct definition

一旦定义后为了使用它，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 是字段的名字而 value 是需要储存在字段中的数据值。这时字段的顺序并不必要与在结构体中声明他们的顺序一致。换句话说，结构体的定义就像一个这个类型的通用模板。例如，我们可以像这样来声明一个特定的用户：

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

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`。

结构体数据的所有权

在列表 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*），这是第十章会讨论的一个 Rust 的功能。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：

Filename: 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说明符：

```
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，通过通过从像`&str`这样的引用切换到像`String`这类拥有所有权的类型来修改修改这个错误。

一个示例程序

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。列表 5-2 中是项目的 *src/main.rs* 文件中为此实现的一个小程序：

Filename: 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
}
```

Listing 5-2: Calculating the area of a rectangle specified by its length and width in separate variables

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

我们的小程序能正常运行；它调用`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。

第三章已经讨论过了一种可行的方法：元组。列表 5-3 是一个使用元组的版本：

Filename: 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
}
```

Listing 5-3: Specifying the length and width of the rectangle with a tuple

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，现在在调用`area`的时候只用传递一个参数。不过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：

```
dimensions.0 * dimensions.1
```

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后再记住。容易忘记或者混淆这些值而造成错误，因为我们没有表达我们代码中数据的意义。

使用结构体重构：增加更多意义

现在引入结构体。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，如列表 5-4 所示：

Filename: 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
}
```

Listing 5-4: Defining a `Rectangle` struct

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：计算一个`Rectangle`的面积，通过其`length`和`width`字段。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`。这是明确性的胜利。

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。列表 5-5 尝试像往常一样使用`println!`宏：

Filename: src/main.rs

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

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

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

Listing 5-5: Attempting to print a `Rectangle` instance

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`，所以它就是向用户展示`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`的实现。

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!`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
```

虽然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，如列表 5-6 所示：

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

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

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

Listing 5-6: Adding the annotation to derive the `Debug` trait and printing the `Rectangle` instance using debug formatting

此时此刻运行程序，运行这个程序，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，这对更大的结构体会有帮助，可以将`println!`的字符串中的`{:?}`替换为`{:#?}`。如果在这个例子中使用了美化的调试风格的话，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有益的行为。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`函数是非常明确的————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，因为这明显就是`Rectangle`类型的行为。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

ch05-01-method-syntax.md
commit c9fd8eb1da7a79deee97020e8ad49af8ded78f9c

方法与函数类似：他们使用`fn`关键和名字声明，他们可以拥有参数和返回值，同时包含一些代码会在某处被调用时执行。不过方法与方法是不同的，因为他们在结构体（或者枚举或者 trait 对象，将分别在第六章和第十三章讲解）的上下文中被定义，并且他们第一个参数总是`self`，它代表方法被调用的结构体的实例。

定义方法

让我们将获取一个`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，如列表 5-7 所示：

Filename: 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()
    );
}
```

Listing 5-7: Defining an `area` method on the `Rectangle` struct

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`中将我们调用`area`方法并传递`rect1`作为参数的地方，改成使用**方法语法**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，像我们这里一样不可变的借用`self`，或者可变的借用`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。如果想要能够在方法中改变调用方法的实例的话，需要将抵押给参数改为`&mut self`。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，不过这是很少见的；这通常用在当方法将`self`转换成别的实例的时候，同时我们想要防止调用者在转换之后使用原始的实例。

使用方法而不是函数，除了使用了方法语法和不需要在每个函数签名中重复`self`类型外，其主要好处在于组织性。我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，又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：

```
p1.distance(&p2);
(&p1).distance(&p2);
```

第一行看起来简洁的多。这种自动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` 的类型。在给出接收者和方法名的前提下，Rust 可以明确的计算出方法是仅仅读取（所以需要 `&self`），做出修改（所以是 `&mut self`）或者是获取所有权（所以是 `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人体工程学实践的一大部分。

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，如果能返回 `true` 若不能则返回 `false`。当我们定义了 `can_hold` 方法，就可以运行列表 5-8 中的代码了：

Filename: 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));
}
```

Listing 5-8: Demonstration of using the as-yet-unwritten `can_hold` method

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。通过观察调用点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` 的长宽是够都大于另一个 `Rectangle`。让我们在列表 5-7 的 `impl` 块中增加这个新方法：

Filename: src/main.rs

```
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
    }
}
```

如果结合列表 5-8 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`块的另一个好用的功能是：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*），因为他们与结构体相关联。即便如此他们也是函数而不是方法，因为他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。例如我们可以一个关联函数，它获取一个维度参数并且用来作为长宽，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

```
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，第七章会讲到后者。

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他们来使得代码更清晰。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`enum`功能并为自己的工具箱再填一个工具。

枚举和模式匹配

---

ch06-00-enums.md
commit 396e2db4f7de2e5e7869b1f8bc905c45c631ad7d

本章介绍**枚举**，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一些值要么什么都不是。然后会讲到`match`表达式中的模式匹配如何使对枚举不同的值运行不同的代码变得容易。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 的枚举与像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 396e2db4f7de2e5e7869b1f8bc905c45c631ad7d

让我们通过一用代码来表现的场景，来看看为什么这里枚举是有用的而且比结构体更合适。比如我们要处理 IP 地。目前被广泛使用的两个主要 IP 标准：IPv4（version four）和 IPv6（version six）。这是我们的程序只可能会遇到两种 IP 地址：我们可以**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，因为枚举值尽可能是其一个成员。IPv4 和 IPv6 从根本上讲都是 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他们当作相同的类型。

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：

```
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`是相同类型的：`IpAddrKind`。例如，接着我们可以顶一个函数来获取`IpAddrKind`：

```
fn route(ip_type: IpAddrKind) { }
```

现在可以使用任意成员来调用这个函数：

```
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，你可以想如列表 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"),
};
```

Listing 6-1: Storing the data and `IpAddrKind` variant of an IP address using a `struct`

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`。我们使用了要给结构体来将 `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。

使用枚举而不是结构体还有另外一个优势：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，不过它将成员种的地址数据嵌入到了两个不同形式的结构体中，他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可能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。

来看看列表 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),
}
```

Listing 6-2: A `Message` enum whose variants each store different amounts and types of values

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`。

定义一个像列表 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。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，他们都有不同的类型，将不能轻易的定义一个获取任何这些信息类型的函数，正如可以使用列表 6-2 中定义的`Message`枚举那样因为他们是一个类型的。

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：

```
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和实用的枚举：`Option`。

`Option`枚举和其相对空值的优势

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。这一部分探索一个`Option`的案例分析，它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，就是一个值可能是某个值或者什么都不是。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，但是从不包含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：

> 我称之为我万亿美元的错误。当时，我在在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应有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以万计美元的苦痛和伤害。

空值的为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。

问题不在于实际的概念而在于具体的实现。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：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`成员的类型。

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的类型那样使用`Option<T>`。例如，这些代码不能编译，因为它尝试将`Option<i8>`与`i8`相比：

```
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）是需要担心可能没有一个值，而编译器会确保我们在使用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。

无需担心错过非空值的假设（和处理）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 396e2db4f7de2e5e7869b1f8bc905c45c631ad7d

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；第十八章会讲到所有不同种类的模式以及他们的作用。`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。

把`match`表达式想象成某种硬币分啦机：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，让我们用他们来作为一个使用`match`的例子！我们可以编写一个函数来获取一个未知的（美国）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，如列表 6-3 中所示：

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> i32 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

Listing 6-3: An enum and a `match` expression that has the variants of the enum as its patterns.

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`的类型是列表 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。

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可以不适用大括号，正如列表 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`：

```
fn value_in_cents(coin: Coin) -> i32 {
    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，列表 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),
}
```

Listing 6-4: A `Coin` enum where the `Quarter` variant also holds a `UsState` value

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，这样如何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。接着在代码那个分支中使用`state`，如下：

```
fn value_in_cents(coin: Coin) -> i32 {
    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，没有分支会匹配知道遇到`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。

编写这个函数非常简单，得益于`match`，它将看起来像列表 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);
```

Listing 6-5: A function that uses a `match` expression on an `Option<i32>`

匹配 `Some(T)`

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！他们是相同的成员。`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`

接着考虑下列表 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。这在一开有点复杂，不过一旦习惯了，你将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 396e2db4f7de2e5e7869b1f8bc905c45c631ad7d

`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。考虑列表 6-6 中的程序，它匹配一个`Option<u8>`值并只希望当值是三时执行代码：

```
let some_u8_value = Some(0u8);
match some_u8_value {
    Some(3) => println!("three"),
    _ => (),
}
```

Listing 6-6: A `match` that only cares about executing code when the value is `Some(3)`

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。如下代码与列表 6-6 中的`match`行为一致：

```
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`强制要求的穷进行检查。`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`。回忆一下列表 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：

```
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：

```
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。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，它使用直观且只向用户暴露他们确实需要的部分，那么让我们转向 Rust 的模块系统吧。

模块

> ch07-00-modules.md
> commit e2a129961ae346f726f8b342455ec2255cdfed68

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，最终你会将功能移动到其他函数中，为了复用也为了更好的组织。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 有一个模块系统来处理编写可复用代码同时保持代码组织度的问题。

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是能（公有）还是不能（私有）在其模块外可见。这是一个模块如何工作的概括：

- 使用`mod`关键字声明模块
- 默认所有内容都是私有的（包括模块自身）。可以使用`pub`关键字将其变成公有并在其命名空间外可见。
- `use`关键字允许引入模块、或模块中的定义到作用域中以便于引用他们。

我们会逐一了解这每一部分并学习如何将他们结合在一起。

`mod`和文件系统

> ch07-01-mod-and-the-filesystem.md
> commit e2a129961ae346f726f8b342455ec2255cdfed68

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，不过我们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。第二章我们见过的`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：

Filename: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
    }
}
```

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。稍后一点会介绍`#[]`和`mod tests`语法，目前只需确保他们位于 *src/lib.rs* 中。

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选择不同的织库项目代码组织来适应多种场景。

模块定义

对于`communicator`网络库，首先我们要定义一个叫做`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：

Filename: 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，如列表 7-1 中所示那样增加这个模块：

Filename: src/lib.rs

```
mod network {
    fn connect() {
    }
}

mod client {
    fn connect() {
    }
}
```

Listing 7-1: The `network` module and the `client` module defined side-by-side in *src/lib.rs*

现在我们有了`network::connect`函数和`client::connect`函数。他们可能有着完全不同的功能，同时他们也不会彼此冲突因为他们位于不同的模块。

虽然在这个例子中，我们构建了一个库，但是 *src/lib.rs* 并没有什么特殊意义。也可以在 *src/main.rs* 中使用子模块。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，如列表 7-2 所示：

Filename: src/lib.rs

```
mod network {
    fn connect() {
    }

    mod client {
        fn connect() {
        }
    }
}
```

Listing 7-2: Moving the `client` module inside of the `network` module

在 *src/lib.rs* 文件中，将现有的 `mod network` 和 `mod client` 的定义替换为 `client` 模块作为 `network` 的一个内部模块。现在我们有了 `network::connect` 和 `network::client::connect` 函数：又一次，这两个 `connect` 函数也不相冲突，因为他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的层次。这是列表 7-1 中的例子以这种方式考虑的组织结构：

```
communicator
  ├─── network
  └─── client
```

而这是列表 7-2 中例子的的结构：

```
communicator
  └─── network
      └─── client
```

可以看到列表 7-2 中，`client` 是 `network` 的子模块，而不是它的同级模块。更为负责的项目可以有很多的模块，所以他们需要符合逻辑地组合在一起以便记录他们。在项目中“符合逻辑”的意义全凭你得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将是你会喜欢的结构。

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，这样就不是所有的内容都落到 *src/lib.rs* 中了。作为例子，我们将从列表 7-3 中的代码开始：

Filename: src/lib.rs

```
mod client {
    fn connect() {
    }
}

mod network {
    fn connect() {
    }

    mod server {
        fn connect() {
        }
    }
}
```

Listing 7-3: Three modules, `client`, `network`, and `network::server`, all defined in *src/lib.rs*

这是模块层次结构：

```
communicator
 ├─── client
 └─── network
     └─── server
```

这里我们仍然**定义**了`client`模块，不过去掉了大括号和`client`模块中的定义并替换为一个分号，这使得 Rust 知道去其他地方寻找模块中定义的代码。

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`client`模块中被去掉的`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！

Rust 默认只知道 *src/lib.rs* 中的内容。如果想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* 而不是在 *src/client.rs*。

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；在本章的后面会解决他们。好消息是，他们仅仅是警告；我们的项目能够被成功编译。

下面使用相同的模式将`network`模块提取到它自己的文件中。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，会出现如列表 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;
  |     ^^^^^^
```

Listing 7-4: Error when trying to extract the `server` submodule into *src/server.rs*

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；让我们继续阅读以理解这是为什么。

列表 7-4 中间的记录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的文件的模式：

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列表 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* 文件改名成 *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。为了更清除的说明为什么 Rust 不知道，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 e2a129961ae346f726f8b342455ec2255cdfed68

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* 文件中解决了列表 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函数是未被使用的。创建他们的意义就在于被另一个项目而不是被自己使用。

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，从外部调用他们。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：

Filename: 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。这个模式在可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**和**私有**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 中的函数调用会被允许，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我们程序的外部被使用。现在这个可能的理论上的外部可用性使 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：

Filename: src/lib.rs

```
pub mod client;

mod network;
```

`pub`写在`mod`之前。再次尝试构建：

```
<warnings>
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：

Filename: src/client.rs

```
pub fn connect() {
}
```

再再一次运行`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他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他们。这也可能是警告你出 bug，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：

Filename: src/network/mod.rs

```
pub fn connect() {
}

mod server;
```

并编译：

```
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() {
  | ^
```

恩，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：

Filename: src/lib.rs

```
pub mod client;

pub mod network;
```

现在再编译的话，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，它只能被当前模块或其子模块访问

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* 输入列表 7-5 中的代码：

Filename: 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();
}
```

Listing 7-5: Examples of private and public functions, some of which are incorrect

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`通过其父模块访问`middle_function`，`outermost`。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访问，`outermost`。这意味着`try_me`函数不允许调用`outermost::inside::inner_function`或`outermost::inside::secret_function`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。在你尝试他们之前，猜测一下他们哪个能修复错误，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他们！

接下来，让我们讨论一下使用`use`关键字来将项引入作用域。

导入命名

ch07-03-importing-names-with-use.md
commit e2a129961ae346f726f8b342455ec2255cdfed68

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，如列表 7-6 中所示的`nested_modules`函数调用。

Filename: src/main.rs

```
pub mod a {
    pub mod series {
        pub mod of {
            pub fn nested_modules() {}
        }
    }
}

fn main() {
    a::series::of::nested_modules();
}
```

Listing 7-6: Calling a function by fully specifying its enclosing module's namespaces

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：

Filename: 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，不用使用完整的 `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他们，像这样：

```
enum TrafficLight {
    Red,
    Yellow,
    Green,
}

use TrafficLight::{Red, Yellow};

fn main() {
    let red = Red;
    let yellow = Yellow;
    let green = TrafficLight::Green; // because we didn't `use` 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：他们是方便的，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*。

Filename: src/lib.rs

```
pub mod client;

pub mod network;

#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
    }
}
```

第十二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，即便现在不准备测试任何功能：

Filename: 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`

warning: function is never used: `connect`, #[warn(dead_code)] on by default
 --> src/network/server.rs:1:1
  |
1 | fn connect() {
  | ^
```

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。之所以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：

Filename: 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！利用他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，你可以利用他们编写出漂亮整洁的代码。

通用集合类型

ch08-00-common-collections.md
commit 0d229cc5a3da341196e15a6761735b2952281569

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，以及他们何以如此特殊。

vector

> ch08-01-vectors.md
> commit 0d229cc5a3da341196e15a6761735b2952281569

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的排列所有的值。vector 只能储存相同类型的值。他们在拥有一系列的场景下非常实用，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，空的 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）：他们可以储存很多值，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他们。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

类似于任何其他的`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他们的内容。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。如果他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他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 的结尾增加新元素是，在没有足够空间将所有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),
];
```

Rust 在编译时就必须准确的知道 vector 中类型的原因是它需要知道储存每个元素到底需要多少内存。第二个优点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的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 65f52921e21ad2e1c79d620fcfd01bde3ee30571

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一下它。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` 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` 于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` 数据的不同方式。

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**：他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，他们通常指的是`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，他们通常也提供有所有权和可借用的变体，就比如说`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他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 s = String::new();
```

这新建了一个叫做`s`的空的字符串，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();
```

这会创建一个包好`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`。如下等同于使用`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。他们有些可能显得有些多于，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 = "Dobrý den";
let hello = "Hello";
let hello = "שָׁלוֹם";
let hello = "नमस्ते";
let hello = "こんにちは";
let hello = "안녕하세요";
let hello = "你好";
let hello = "Olá";
let hello = "З д р а в с т в у й т е";
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。

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。回忆之前第四章我们讲到`&String`是如何被强转为`&str`的：写成`&s2`的话`String`将会被强转成一个合适的类型`&str`。又因为方法没有获取参数的所有权，所以`s2`在这个操作后仍然有效。

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 如何在内存中储存字符串。

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("З д р а в с т в у й т е").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“Здравствуйте” 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 = "З д р а в с т в у й т е";
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`的第一个字节是`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放生。

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 标量值的角度理解他们，也就像 Rust 的`char`类型那样，这些字节看起来像这样：

```
['न', 'म', 'स', '्', 'त', 'े']
```

这里有六个`char`，不过第四个和第六个都不是字母，他们是发音符号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 = "З д р а в с т в у й т е";

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"。

那么如果获取`&hello[0..1]`会发生什么呢？回答是：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 `З д р а в с т в у й т е` 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。堆"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
स
्
त
े
```

`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 0d229cc5a3da341196e15a6761735b2952281569

最后要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，它决定了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，就能得到他们的得分。

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的`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`来增加元素。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，这个是最不常用的，所以并不包含在被 prelude 自动引用的功能中。标准库中对哈希 map 的支持也相对较少；例如，并没有内建的用于构建的宏。

就像 vector 一样，哈希 map 将他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`进很多不同的数据结构，而除非显式指定 Rust 无从得知你需要的类型。但是对于键和值的参数来说，可以使用下划线而 Rust 可以根据 vector 中数据的类型推断出哈希 map 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`将会是与蓝队分数相关的值，而这个值将是`Some(10)`。因为`get`返回`Option<V>`所以结果被封装进`Some`；如果某个键在哈希 map 中没有对应的值，`get`会返回`None`。程序将需要采用第六章提到的方法中之一来处理`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

虽然键值对的数量是可以增长的，不过每个单独的键同时只能关联一个值。当你想要改变哈希 map 中的数据时，必须选择是用新值替代旧值，还是完全无视旧值。我们也可以选择保留旧值而忽略新值，并只在键**没有**对应一个值时增加新值。或者可以结合新值和旧值。让我们看看着每一种方式是如何工作的！

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一个可变引用（`&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被借用规则所允许。

哈希函数

`HashMap`默认使用一个密码学上是安全的哈希函数，它可以提供抵抗拒绝服务（Denial of Service, DoS）攻击的能力。这并不是现有最快的哈希函数，不过为了更好的安全性带来一些性能下降也是值得的。如果你监控你的代码并发现默认哈希函数对你来说非常慢，可以通过指定一个不同的*hasher* 来切换为另一个函数。hasher 是一个实现了`BuildHasher` trait 的类型。第十章会讨论 trait 和如何实现他们。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！

我们已经开始解除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 fc825966fabaa408067eb2df3aa45e4fa6644fb6

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 Rust 有很多功能来处理当现错误的情况。在很多情况下，Rust 要求你承认出错的可能性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>`值和`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 380e6ee57c251f5ffa8df4c58b3949405448d914

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，并接着退出。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!`：

Filename: 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 debug [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!`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：

Filename: src/main.rs

```
fn main() {
    let v = vec![1, 2, 3];

    v[100];
}
```

这里尝试访问 vector 的第一百个元素，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 这样语言会尝直接试提供所要求的值，即便这可能不是你期望的：你会得到对任何应 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 debug [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。列表 9-1 显示了其输出：

```
$ RUST_BACKTRACE=1 cargo run
    Finished debug [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:1395
stack backtrace:
   1:        0x10922522c -
std::sys::imp::backtrace::tracing::imp::write::h1204ab053b688140
   2:        0x10922649e -
std::panicking::default_hook::{{closure}}::h1204ab053b688140
   3:        0x109226140 - std::panicking::default_hook::h1204ab053b688140
   4:        0x109226897 -
std::panicking::rust_panic_with_hook::h1204ab053b688140
   5:        0x1092266f4 - std::panicking::begin_panic::h1204ab053b688140
   6:        0x109226662 - std::panicking::begin_panic_fmt::h1204ab053b688140
   7:        0x1092265c7 - rust_begin_unwind
   8:        0x1092486f0 - core::panicking::panic_fmt::h1204ab053b688140
   9:        0x109248668 -
core::panicking::panic_bounds_check::h1204ab053b688140
  10:        0x1092205b5 - <collections::vec::Vec<T> as
core::ops::Index<usize>>::index::h1204ab053b688140
  11:        0x10922066a - panic::main::h1204ab053b688140
  12:        0x1092282ba - __rust_maybe_catch_panic
  13:        0x109226b16 - std::rt::lang_start::h1204ab053b688140
  14:        0x1092206e9 - main
```

Listing 9-1: The backtrace generated by a call to `panic!` displayed when the environment variable `RUST_BACKTRACE` is set

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。当将来你得代码出现了 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

ch09-01-unrecoverable-errors-with-panic.md
commit 0c1d55ef48e5f6cf6a3b221f5b6dd4c922130bb1

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回应的原因失败。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就失败，这是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，它定义有如下连个成员，`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：如列表 9-2 所示打开一个文件：

Filename: src/main.rs

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

fn main() {
    let f = File::open("hello.txt");
}
```

Listing 9-2: Opening a file

如何知道`File::open`返回一个`Result`呢？我们可以查看标准库 API 文档，或者可以直接问编译器！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
`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。

我们需要在列表 9-2 的代码中增加根据 `File::open` 返回值进行不同处理的逻辑。列表 9-3 展示了一个处理 `Result` 的基本工具：第六章学习过的 `match` 表达式。

Filename: 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)
        },
    };
}
```

Listing 9-3: Using a `match` expression to handle the `Result` variants we might have

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`，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

列表 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，我们仍然希望像列表 9-3 那样 `panic!`。让我们看看列表 9-4，其中 `match` 增加了另一个分支：

Filename: 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
            )
        },
    };
}
```

Listing 9-4: Handling different kinds of errors in different ways

`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。

`if error.kind() == ErrorKind::NotFound`条件被称作 *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 条件中而只是仅仅引用它。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，我们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并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`，它的实现就类似于列表 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传递给 `expect` 的参数，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，列表 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：

```rust
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),
    }
}
```

Listing 9-5: A function that returns errors to the calling code using `match`

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。我们选择`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`，类似于列表 9-3 中的`match`，唯一的区别是不再当`Err`时调用`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，亦或一个包含`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：`?`

列表 9-6 展示了一个`read_username_from_file`的实现，它实现了与列表 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：

```
use std::io;
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)
}
```

Listing 9-6: A function that returns errors to the calling code using `?`

`Result`值之后的`?`被定义为与列表 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。

在列表 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任何`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。其功能再一次与列表 9-5 和列表 9-5 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，因为他被定义为与列表 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")?;
}
```

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的函数。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。

在列表 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，不过列表 10-1 中的代码只存在于一个位置：

Filename: 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 numbers = vec![34, 50, 25, 100, 65];

    let result = largest(&numbers);
    println!("The largest number is {}", result);

    let numbers = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let result = largest(&numbers);
    println!("The largest number is {}", result);
}
```

Listing 10-3: Abstracted code to find the largest number in two lists

这个函数有一个参数`list`，它代表会传递给函数的任何具体`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。

从列表 10-2 到列表 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泛型也可以利用相同的步骤来减少重复代码。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 55d9e75ffec92e922273c997026bb10613a76578

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，列表 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。第一个是从列表 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：

Filename: 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 numbers = vec![34, 50, 25, 100, 65];

    let result = largest_i32(&numbers);
    println!("The largest number is {}", result);

    let chars = vec!['y', 'm', 'a', 'q'];

    let result = largest_char(&chars);
    println!("The largest char is {}", result);
}
```

Listing 10-4: Two functions that differ only in their names and the types in their signatures

这里`largest_i32`和`largest_char`有着完全相同的函数体，所以能够将这两个函数变成一个来减少重复就太好了。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`。任何标识符抖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"是大部分 Rust 程序员的首选。

当需要再函数体中使用一个参数时，必须再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 {
```

这可以理解为：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。

列表 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！

Filename: 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 numbers = vec![34, 50, 25, 100, 65];

    let result = largest(&numbers);
    println!("The largest number is {}", result);

    let chars = vec!['y', 'm', 'a', 'q'];

    let result = largest(&chars);
    println!("The largest char is {}", result);
}
```

Listing 10-5: A definition of the `largest` function that uses generic type parameters but doesn't compile yet

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`的函数体对`T`的所有可能的类型都无法工作；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。列表 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`：

Filename: 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 };
}
```

Listing 10-6: A `Point` struct that holds `x` and `y` values of type `T`

其语法类似于函数定义中的泛型应用。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`的定义中是使用了要给泛型类型，我们想要表达的是结构体`Point`对于一些类型`T`是泛型的，而且无论这个泛型是什么，字段`x`和`y`**都是**相同类型的。如果尝试创建一个有不同类型值的`Point`的实例，像列表 10-7 中的代码就不能编译：

Filename: src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let wont_work = Point { x: 5, y: 4.0 };
}
```

Listing 10-7: The fields `x` and `y` must be the same type because both have the same generic data type `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。在列表 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：

Filename: 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 };
}
```

Listing 10-8: A `Point` generic over two types so that `x` and `y` may be values of different types

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。回忆一下列表 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。

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。列表 10-9 中展示了列表 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：

Filename: 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());
}
```

Listing 10-9: Implementing a method named `x` on the `Point<T>` struct that will return a reference to the `x` field, which is of type `T`.

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。

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。列表 10-10 中在列表 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`W`）来创建一个新`Point`类型的实例：

Filename: 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);
}
```

Listing 10-10: Methods that use different generic types than their struct's definition

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`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，因为他们于结构体定义相对应。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列表 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：

Filename: 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 会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 709eb1eaca48864fafd9263042f5f9d9d6ffe08d

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希望的功能。

> 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须行为的集合。

例如，这里有多个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。列表 10-11 中展示了一个表现这个概念的`Summarizable` trait 的定义：

Filename: lib.rs

```
pub trait Summarizable {
    fn summary(&self) -> String;
}
```

Listing 10-11: Definition of a `Summarizable` trait that consists of the behavior provided by a `summary` method

使用`trait`关键字来定义一个 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 的类型所需要的行为的方法签名，在这个例子中是是`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 体中可以有多个方法，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。列表 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。

Filename: lib.rs

```
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)
    }
}
```

Listing 10-12: Implementing the `Summarizable` trait on the `NewsArticle` and `Tweet` types

在类型上实现 trait 类似与实现与 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，而是需要在大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`。

注意因为列表 10-12 中我们在相同的`lib.rs`力定义了`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，如列表 10-13 所示：

Filename: 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)
    }
}
```

Listing 10-13: Bringing the `Summarizable` trait from our `aggregator` crate into scope in another crate

另外这段代码假设`Summarizable`是一个公有 trait，这是因为列表 10-11 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`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`定义与此。这个限制是我们称为 *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是实现相同的 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。

列表 10-14 中展示了如何为 `Summarize` trait 的 `summary` 方法指定一个默认的字符串值，而不是像列表 10-11 中那样只是定义方法签名：

> Filename: lib.rs
>
> ```
> pub trait Summarizable {
>     fn summary(&self) -> String {
>         String::from("(Read more...)")
>     }
> }
> ```
>
> Listing 10-14: Definition of a `Summarizable` trait with a default implementation of the `summary` method

如果想要对 `NewsArticle` 实例使用这个默认实现，而不是像列表 10-12 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` 实现并不要求对列表 10-12 中的 `Tweet` 和列表 10-13 中的 `WeatherForecast` 对 `Summarizable` 的实现做任何改变：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么实现了特定 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*。

例如在列表 10-12 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。列表 10-13 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。如果我们需要能够在函数中使用`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

何是实现了`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列表 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`的类型调用这些带啊吗，可以在`T`的 trait bounds 中增加`Copy`！列表 10-15 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`：

Filename: 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 numbers = vec![34, 50, 25, 100, 65];

    let result = largest(&numbers);
    println!("The largest number is {}", result);

    let chars = vec!['y', 'm', 'a', 'q'];

    let result = largest(&chars);
    println!("The largest char is {}", result);
}
```

Listing 10-15: A working definition of the `largest` function that works on any generic type that implements the `PartialOrd` and `Copy` traits

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！

trait 和 trait bounds 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 bounds 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 d7a4e99554da53619dd71044273535ba0186f40a

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**，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多种不同方式向关联的情况，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并非其期望引用的数据。考虑一下列表 10-16 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);
}
```

Listing 10-16: An attempt to use a reference whose value has gone out of scope

---

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。列表 10-17 展示了与列表 10-16 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); // |
                   //        |
                   // -------+
}
```

Listing 10-17: Annotations of the lifetimes of `x` and `r`, named `'a` and `'b` respectively

我们将`r`的声明周期标记为`'a`而将`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：引用者没有比被引用者存在的更久。

让我们看看列表 10-18 中这个并没有产生悬垂引用且可以正常编译的例子：

```
{
    let x = 5;            // ----------+-- 'b
                          //           |
    let r = &x;           // --+-------+-- 'a
                          //   |       |
    println!("r: {}", r); //   |       |
                          // --+       |
}                         // ----------+
```

Listing 10-18: A valid reference because the data has a longer lifetime than the reference

`x`拥有生命周期`'b`，在这里它比`'a`要大。这就意味着`r`可以引用`x`：Rust 知道`r`中的引用在`x`有效的时候也会一直有效。

现在我们已经在一个具体的例子中展示了引用的声明周期位于何处，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 中最长的那一个的函数。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，列表 10-19 中的代码应该会打印出`The longest string is abcd`：

Filename: 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);
}
```

Listing 10-19: A `main` function that calls the `longest` function to find the longest of two string slices

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`函数获取其参数的引用。我们希望函数能够接受`String`的 slice（也就是变量`string1`的类型）和字符串字面值（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是我们想要的讨论。

如果尝试像列表 10-20 中那样实现`longest`函数，它并不能编译：

Filename: src/main.rs

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Listing 10-20: An implementation of the `longest` function that returns the longest of two string slices, but does not yet compile

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，所以也就不能像列表 10-17 和 10-18 那样通过观察作用域来确定返回的引用总是有效的。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相关分析。

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，和一个也有的生命周期参数`'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
```

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加括号中。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，就像列表 10-21 中在每个引用中都加上了`'a`那样：

> Filename: src/main.rs
>
> ```
> fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
>     if x.len() > y.len() {
>         x
>     } else {
>         y
>     }
> }
> ```
>
> Listing 10-21: The `longest` function definition that specifies all the references in the signature must have the same lifetime, `'a`

这段代码能够编译并会产生我们想要使用列表 10-19 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。

通过在函数签名中指定生命周期参数，我们不会改变任何参数或返回值的生命周期，不过我们说过任何不坚持这个协议的类型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，不过只需要知道一些可以使用`'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 来说经常都是不可能分析的。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，具体被`'a`所替代的生命周期是`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。

让我们如何通过传递拥有不同具体生命周期的引用来观察他们是如何限制`longest`函数的使用的。列表 10-22 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`则引用了一些直到外部作用域结束都是有效的值。借用检查器赞同这些代码；它能够编译和运行，并打印出`The longest string is long string is long`：

> Filename: src/main.rs

```
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);
    }
}
```

Listing 10-22: Using the `longest` function with references to `String` values that have different concrete lifetimes

接下来，让我们尝试一个`result`的引用的生命周期必须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，不过将`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。注意列表 10-23 中的代码不能编译：

Filename: 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);
}
```

Listing 10-23: Attempting to use `result` after `string2` has gone out of scope won't compile

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`还未离开作用域，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，借用检查器不允许列表 10-23 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是否是正确的！

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：

Filename: 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：

Filename: 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，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。列表 10-24 中有一个存放了一个字符串 slice 的结构体 `ImportantExcerpt`：

Filename: 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 };
}
```

Listing 10-24: A struct that holds a reference, so its definition needs a lifetime annotation

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，我们在列表 10-25 中再次展示它，没有生命周期注解却能成功编译：

Filename: 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[..]
}
```

Listing 10-25: A function we defined in Chapter 4 that compiled without lifetime annotations, even though the parameter and return type are references

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 之前的版本的 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将被合并和添加到编译器中是完全可能的。未来将会需要越来越少的生命周期注解。

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，此时编译器会会考虑，如果代码符合这些场景，就不需要明确指定生命周期。

这些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，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，而两条规则则适用于输出生命周期。如果编译器检查完这三条规则并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。话句话说就是，有一个引用参数的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，而且它被赋予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，不过其中之一是`&self`或`&mut self`，那么`self`的生命周期被赋予所有输出生命周期参数。这使得方法看起来更简洁。

假设我们自己就是编译器并来计算列表 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，这次我们从列表 10-20 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，所有就有两个生命周期：

```
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列表 10-20 的代码时会出现错误的原因：编译器适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，其语法依然类似列表 10-10 中展示的泛型类型参数的语法：包括声明生命周期参数的位置和以及生命周期参数是否与结构体字段或方法的参数与返回值相关联。

（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适用，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列表 10-24 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：

```
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：

```
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。所有的字符串字面值都拥有 `'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。因此所有的字符串字面值都是 `'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
    }
}
```

这个是列表 10-21 中那个返回两个字符串 slice 中最长者的 `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，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。而所有的这一切，发生在运行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 c9fd8eb1da7a79deee97020e8ad49af8ded78f9c

> 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）

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 在其类型系统上下了很大的功夫，来确保程序像我们希望的那样运行，不过它并不有助于所有情况。为此，Rust 也包含为语言自身编写软件测试的支持。

例如，我们可以编写一个叫做`add_two`的函数，它的签名有一个整型参数并返回一个整型值。我们可以实现并编译这个函数，而 Rust 也会进行所有的类型检查和借用检查，正如我们之前见识过的那样。Rust 所**不能**检查的是，我们实现的这个函数是否返回了参数值加二后的值，而不是加 10 或者减 50！这也就是测试出场的地方。例如可以编写传递`3`给`add_two`函数并检查我们是否得到了`5`。任何时候修改了代码我们都可以运行测试来确保没有改变任何现有测试所指定的行为。

测试是一项技能，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。然而我们可以讨论的是 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 77370c073661548dd56bbcb43cc64713585acbba

测试是一种使用特定功能的 Rust 函数，它用来验证非测试的代码按照期望的方式运行。我们讨论过的任何 Rust 代码规则都适用于测试！让我们看看 Rust 提供的具体用来编写测试的功能：`test`属性、一些宏和`should_panic`属性。

`test`属性

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。让我们使用 Cargo 来创建一个新的库项目`adder`：

```
$ cargo new adder
     Created library `adder` project
$ cd adder
```

Cargo 在创建新的库项目时自动生成一个简单的测试。这是`src/lib.rs`中的内容：

Filename: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
    }
}
```

现在让我们暂时忽略`tests`模块和`#[cfg(test)]`注解并只关注函数。注意它之前的`#[test]`：这个属性表明这是一个测试函数。这个函数目前没有任何内容，所以绝对是可以通过的！使用`cargo test`来运行测试：

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished debug [unoptimized + debuginfo] target(s) in 0.22 secs
     Running target/debug/deps/adder-abcabcabc

running 1 test
test 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
```

Cargo 编译并运行了测试。这里有两部分输出：本章我们将关注第一部分。第二部分是文档测试的输出，第十四章会介绍他们。现在注意看这一行：

```
test it_works ... ok
```

`it_works`文本来源于测试函数的名称。

这里也有一行总结告诉我们所有测试的聚合结果：

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

`assert!`宏

空的测试函数之所以能通过是因为任何没有`panic!`的测试都是通过的，而任何`panic!`的测试都算是失败。让我们使用`assert!宏来使测试失败：

Filename: src/lib.rs

```
#[test]
fn it_works() {
    assert!(false);
}
```

`assert!`宏由标准库提供，它获取一个参数，如果参数是`true`，什么也不会发生。如果参数是`false`，这个宏会`panic!`。再次运行测试：

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished debug [unoptimized + debuginfo] target(s) in 0.22 secs
     Running target/debug/deps/adder-abcabcabc

running 1 test
test it_works ... FAILED

failures:

---- it_works stdout ----
    thread 'it_works' panicked at 'assertion failed: false', src/lib.rs:5
note: Run with `RUST_BACKTRACE=1` for a backtrace.


failures:
    it_works

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured

error: test failed
```

Rust 表明测试失败了：

```
test it_works ... FAILED
```

并展示了测试是因为src/lib.rs`的第 5 行`assert!`宏得到了一个`false`值而失败的：

```
thread 'it_works' panicked at 'assertion failed: false', src/lib.rs:5
```

失败的测试也体现在了总结行中：

```
test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured
```

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`==`宏的表达式来做到。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编译处理这些操作：`assert_eq!`和`assert_ne!`。这两个宏分别比较两个值是相等还是不相等。使用这些宏的另一个优势是当断言失败时他们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。

下面是分别使用这两个宏其会测试通过的例子：

Filename: src/lib.rs

```
#[test]
fn it_works() {
    assert_eq!("Hello", "Hello");

    assert_ne!("Hello", "world");
}
```

也可以对这些宏指定可选的第三个参数，它是一个会加入错误信息的自定义文本。这两个宏展开后的逻辑看起来像这样：

```
// assert_eq! - panic if the values aren't equal
if left_val != right_val {
    panic!(
        "assertion failed: `(left == right)` (left: `{:?}`, right: `{:?}`): {}"
        left_val,
        right_val,
        optional_custom_message
    )
}

// assert_ne! - panic if the values are equal
if left_val == right_val {
    panic!(
        "assertion failed: `(left != right)` (left: `{:?}`, right: `{:?}`): {}"
        left_val,
        right_val,
        optional_custom_message
    )
}
```

看看这个因为`hello`不等于`world`而失败的测试。我们还增加了一个自定义的错误信息，`greeting operation failed`：

Filename: src/lib.rs

```
#[test]
fn a_simple_case() {
    let result = "hello"; // this value would come from running your code
    assert_eq!(result, "world", "greeting operation failed");
}
```

毫无疑问运行这个测试会失败，而错误信息解释了为什么测试失败了并且带有我们的指定的自定义错误信息：

```
---- a_simple_case stdout ----
    thread 'a_simple_case' panicked at 'assertion failed: `(left == right)`
    (left: `"hello"`, right: `"world"`): greeting operation failed',
    src/main.rs:4
```

`assert_eq!`的两个参数被称为 "left" 和 "right" ，而不是 "expected" 和 "actual" ；值的顺序和硬编码的值并没有什么影响。

因为这些宏使用了`==`和`!=`运算符并使用调试格式打印这些值，进行比较的值必须实现`PartialEq`和`Debug` trait。Rust 提供的类型实现了这些 trait，不过自定义的结构体和枚举则需要自己实现`PartialEq`以便能够断言这些值是否相等，和实现`Debug`以便在断言失败时打印出这些值。因为第五章提到过这两个 trait 都是 derivable trait，所以通常可以直接在结构体或枚举上加上`#[derive(PartialEq, Debug)]`注解。查看附录 C 来寻找更多关于这些和其他 derivable trait 的信息。

使用`should_panic`测试期望的失败

可以使用另一个属性来反转测试中的失败：`should_panic`。这在测试调用特定的函数会产生错误的函数时很有帮助。例如，让我们测试第八章中的一些我们知道会 panic 的代码：尝试使用 range 语法和并不组成完整字母的字节索引来创建一个字符串 slice。在有`#[test]`属性的函数之前增加`#[should_panic]`属性，如列表 11-1 所示：

Filename: src/lib.rs

```
#[test]
#[should_panic]
fn slice_not_on_char_boundaries() {
    let s = "З д р а в с т в у й т е";
    &s[0..1];
}
```

Listing 11-1: A test expecting a `panic!`

这个测试是成功的，因为我们表示代码应该会 panic。相反如果代码因为某种原因没有产生`panic!`则测试会失败。

使用`should_panic`的测试是脆弱的，因为难以保证测试不会因为一个不同于我们期望的原因失败。为了帮助解决这个问题，`should_panic`属性可以增加一个可选的`expected`参数。测试工具会确保错误信息里包含我们提供的文本。一个比列表 11-1 更健壮的版本如列表 11-2 所示：

Filename: src/lib.rs

```
#[test]
#[should_panic(expected = "do not lie on character boundary")]
fn slice_not_on_char_boundaries() {
    let s = "З д р а в с т в у й т е";
    &s[0..1];
}
```

Listing 11-2: A test expecting a `panic!` with a particular message

请自行尝试当`should_panic`的测试出现 panic 但并不符合期望的信息时会发生什么：在测试中因为不同原因造成`panic!`，或者将期望的 panic 信息改为并不与字母字节边界 panic 信息相匹配。

运行测试

ch11-02-running-tests.md
commit cf52d81371e24e14ce31a5582bfcb8c5b80d26cc

类似于`cargo run`会编译代码并运行生成的二进制文件，`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。`cargo test`生成的二进制文件默认会并行的运行所有测试并在测试过程中捕获生成的输出，这样就更容易阅读测试结果的输出。

可以通过指定命令行选项来改变这些运行测试的默认行为。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。分隔这些参数的方法是`--`：`cargo test`之后列出了传递给`cargo test`的参数，接着是分隔符`--`，之后是传递给测试二进制文件的参数。

并行运行测试

测试使用线程来并行运行。为此，编写测试时需要注意测试之间不要相互依赖或者存在任何共享状态。共享状态也可能包含在运行环境中，比如当前工作目录或者环境变量。

如果你不希望它这样运行，或者想要更加精确的控制使用线程的数量，可以传递`--test-threads`参数和线程的数量给测试二进制文件。将线程数设置为 1 意味着没有任何并行操作：

```
$ cargo test -- --test-threads=1
```

捕获测试输出

Rust 的测试库默认捕获并丢弃标准输出和标准错误中的输出，除非测试失败了。例如，如果在测试中调用了`println!`而测试通过了，你将不会在终端看到`println!`的输出。这个行为可以通过向测试二进制文件传递`--nocapture`参数来禁用：

```
$ cargo test -- --nocapture
```

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多时间。如果你负责特定位置的代码，你可能会希望只与这些代码相关的测试。`cargo test`有一个参数允许你通过指定名称来运行特定的测试。

列表 11-3 中创建了三个如下名称的测试：

Filename: src/lib.rs

```
#[test]
fn add_two_and_two() {
    assert_eq!(4, 2 + 2);
}

#[test]
fn add_three_and_two() {
    assert_eq!(5, 3 + 2);
}

#[test]
fn one_hundred() {
    assert_eq!(102, 100 + 2);
}
```

Listing 11-3: Three tests with a variety of names

使用不同的参数会运行不同的测试子集。没有参数的话，如你所见会运行所有的测试：

```
$ cargo test
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/deps/adder-abcabcabc

running 3 tests
test add_three_and_two ... ok
test one_hundred ... ok
test add_two_and_two ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured
```

可以传递任意测试的名称来只运行那个测试：

```
$ cargo test one_hundred
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/deps/adder-abcabcabc

running 1 test
test one_hundred ... ok

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

也可以传递名称的一部分，`cargo test`会运行所有匹配的测试：

```
$ cargo test add
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/deps/adder-abcabcabc

running 2 tests
test add_three_and_two ... ok
test add_two_and_two ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured
```

模块名也作为测试名的一部分，所以类似的模块名也可以用来指定测试特定模块。例如，如果将我们的代码组织成一个叫`adding`的模块和一个叫`subtracting`的模块并分别带有测试，如列表 11-4 所示：

Filename: src/lib.rs

```
mod adding {
    #[test]
    fn add_two_and_two() {
        assert_eq!(4, 2 + 2);
    }

    #[test]
    fn add_three_and_two() {
        assert_eq!(5, 3 + 2);
    }

    #[test]
    fn one_hundred() {
        assert_eq!(102, 100 + 2);
    }
}

mod subtracting {
    #[test]
    fn subtract_three_and_two() {
        assert_eq!(1, 3 - 2);
    }
}
```

Listing 11-4: Tests in two modules named `adding` and `subtracting`

执行`cargo test`会运行所有的测试，而模块名会出现在输出的测试名中：

```
$ cargo test
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/deps/adder-abcabcabc

running 4 tests
test adding::add_two_and_two ... ok
test adding::add_three_and_two ... ok
test subtracting::subtract_three_and_two ... ok
test adding::one_hundred ... ok
```

运行`cargo test adding`将只会运行对应模块的测试而不会运行任何 subtracting 模块中的测试：

```
$ cargo test adding
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/deps/adder-abcabcabc

running 3 tests
test adding::add_three_and_two ... ok
test adding::one_hundred ... ok
test adding::add_two_and_two ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured
```

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，所以对于大多数`cargo test`命令，我们希望能排除它。无需为`cargo test`创建一个用来在运行所有测试时排除特定测试的参数并每次都要记得使用它，我们可以对这些测试使用`ignore`属性：

Filename: src/lib.rs

```
#[test]
fn it_works() {
    assert!(true);
}

#[test]
#[ignore]
fn expensive_test() {
    // code that takes an hour to run
}
```

现在运行测试，将会发现`it_works`运行了，而`expensive_test`没有：

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished debug [unoptimized + debuginfo] target(s) in 0.24 secs
     Running target/debug/deps/adder-abcabcabc

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
```

我们可以通过`cargo test -- --ignored`来明确请求只运行那些耗时的测试：

```
$ cargo test -- --ignored
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/deps/adder-abcabcabc

running 1 test
test expensive_test ... ok

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

通过这种方式，大部分时间运行`cargo test`将是快速的。当需要检查`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 cf52d81371e24e14ce31a5582bfcb8c5b80d26cc

正如之前提到的，测试是一个很广泛的学科，而且不同的人有时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**（*unit 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。他们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得代码，他们只针

对共有接口而且每个测试会测试多个模块。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隔离与其他部分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何处的代码是否符合预期。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。他们被分离进每个文件中他们自有的`tests`模块中。

测试模块和`cfg(test)`

通过将测试放进他们自己的模块并对该模块使用`cfg`注解，我们可以告诉 Rust 只在执行`cargo test`时才编译和运行测试代码。这在当我们只希望用`cargo build`编译库代码时可以节省编译时间，并减少编译产物的大小因为并没有包含测试。

还记得上一部分新建的`adder`项目吗？Cargo 为我们生成了如下代码：

Filename: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
    }
}
```

我们忽略了模块相关的信息以便更关注模块中测试代码的机制，不过现在让我们看看测试周围的代码。

首先，这里有一个属性`cfg`。`cfg`属性让我们声明一些内容只在给定特定的**配置**（*configuration*）时才被包含进来。Rust 提供了`test`配置用来编译和运行测试。通过这个属性，Cargo 只会在尝试运行测试时才编译测试代码。

接下来，`tests`包含了所有测试函数，而我们的代码则位于`tests`模块之外。`tests`模块的名称是一个惯例，除此之外这是一个遵守第七章讲到的常见可见性规则的普通模块。因为这是一个内部模块，我们需要将要测试的代码引入作用域。这对于一个大的模块来说是很烦人的，所以这里经常使用全局导入。

从本章到现在，我们一直在为`adder`项目编写并没有实际调用任何代码的测试。现在让我们做一些改变！在 *src/lib.rs* 中，放入`add_two`函数和带有一个检验代码的测试的`tests`模块，如列表 11-5 所示：

Filename: src/lib.rs

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use add_two;

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }
}
```

Listing 11-5: Testing the function `add_two` in a child `tests` module

注意除了测试函数之外，我们还在`tests`模块中添加了`use add_two;`。这将我们想要测试的代码引入到了内部的`tests`模块的作用域中，正如任何内部模块需要做的那样。如果现在使用`cargo test`运行测试，它会通过：

```
running 1 test
test tests::it_works ... ok

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

如果我们忘记将`add_two`函数引入作用域，将会得到一个 unresolved name 错误，因为`tests`模块并不知道任何关于`add_two`函数的信息：

```
error[E0425]: unresolved name `add_two`
 --> src/lib.rs:9:23
  |
9 |         assert_eq!(4, add_two(2));
  |                       ^^^^^^^ unresolved name
```

如果这个模块包含很多希望测试的代码，在测试中列出每一个`use`语句将是很烦人的。相反在测试子模块中使用`use super::*;`来一次将所有内容导入作用域中是很常见的。

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。不过无论你坚持哪种测试意识形态，Rust 确实允许你测试私有函数，由于私有性规则。考虑列表 11-6 中带有私有函数`internal_adder`的代码：

Filename: 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 internal_adder;

    #[test]
    fn internal() {
        assert_eq!(4, internal_adder(2, 2));
    }
}
```

Listing 11-6: Testing a private function

因为测试也不过是 Rust 代码而 `tests` 也只是另一个模块，我们完全可以在一个测试中导入并调用 `internal_adder`。如果你并不认为私有函数应该被测试，Rust 也不会强迫你这么做。

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。他们的目的是测试库的个个部分结合起来能否正常工作。每个能正确运行的代码单元集成在一起也可能会出现问题，所以集成测试的覆盖率也是很重要的。

tests 目录

Cargo 支持位于 *tests* 目录中的集成测试。如果创建它并放入 Rust 源文件，Cargo 会将每一个文件当作单独的 crate 来编译。让我们试一试！

首先，在项目根目录创建一个 *tests* 目录，挨着 *src* 目录。接着新建一个文件 *tests/integration_test.rs*，并写入列表 11-7 中的代码：

Filename: tests/integration_test.rs

```
extern crate adder;

#[test]
fn it_adds_two() {
    assert_eq!(4, adder::add_two(2));
}
```

Listing 11-7: An integration test of a function in the `adder` crate

在开头使用了 `extern crate adder`，单元测试中并不需要它。`tests` 目录中的每一个测试文件都是完全独立的 crate，所以需要在每个文件中导入我们的库。这也就是为何 `tests` 是编写集成测试的绝佳场所：他们像任何其他用户那样，需要将库导入 crate 并只能使用公有 API。

这个文件中也不需要 `tests` 模块。除非运行测试否则整个文件夹都不会被编译，所以无需将任何部分标记为 `#[cfg(test)]`。另外每个测试文件都被隔离进其自己的 crate 中，无需进一步隔离测试代码。

让我们运行集成测试，同样使用`cargo test`来运行：

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
     Running target/debug/deps/adder-abcabcabc

running 1 test
test tests::it_works ... ok

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

     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

   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。注意当在任何 *src* 目录的文件中增加单元测试时，单元测试部分的对应输出也会增加。增加集成测试文件中的测试函数也会对应增加输出。如果在 *tests* 目录中增加集成测试**文件**，则会增加更多集成测试部分：一个文件对应一个部分。

为`cargo test`指定测试函数名称参数也会匹配集成测试文件中的函数。为了只运行某个特定集成测试文件中的所有测试，可以使用`cargo test`的`--test`参数：

```
$ cargo test --test integration_test
    Finished debug [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
```

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，这是可以的，Cargo 会将每一个文件当作一个独立的 crate。

最终，可能会有一系列在所有集成测试中通用的帮助函数，例如建立通用场景的函数。如果你将这些函数提取到 *tests* 目录的一个文件中，比如说 *tests/common.rs*，则这个文件将会像这个目录中的其他包含测试的 Rust 文件一样被编译进一个单独的 crate 中。它也会作为一个独立的部分出现在测试输出中。因为这很可能不是你所希望的，所以建议在子目录中使用 *mod.rs* 文件，比如 *tests/common/mod.rs*，来放置帮助函数。*tests* 的子目录不会被作为单独的 crate 编译或者作为单独的部分出现在测试输出中。

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。这也是 Rust 二进制项目明确采用

src/main.rs 调用 *src/lib.rs* 中逻辑的结构的原因之一。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，而如果这些功能没有问题的话，*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确保即使改变代码函数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每一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时能否使用，并像其他代码那样测试库的公有 API。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的逻辑 bug 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

---

ch12-00-an-io-project.md
commit efd59dd0fe8e3658563fb5fd289af9d862e07a03

之前几个章节我们学习了很多知识。让我们一起运用这些新知识来构建一个项目。在这个过程中，我们还将学习到更多 Rust 标准库的内容。

那么我们应该写点什么呢？这得是一个利用 Rust 优势的项目。Rust 的一个强大的用途是命令行工具：Rust 的运行速度、安全性、“单二进制文件”输出和跨平台支持使得它称为这类工作的绝佳选择。所以我们将创建一个我们自己的经典命令行工具：`grep`。`grep` 有着极为简单的应用场景，它完成如下工作：

1. 它获取一个文件和一个字符串作为参数。
2. 读取文件
3. 寻找文件中包含字符串参数的行
4. 打印出这些行

另外，我们还将添加一个额外的功能：一个环境变量允许我们大小写不敏感的搜索字符串参数。

还有另一个很好的理由使用 `grep` 作为示例项目：Rust 社区的成员，Andrew Gallant，已经使用 Rust 创建了一个功能非常完整的 `grep` 版本。它叫做 `ripgrep`，并且它非常非常快。这样虽然我们的 `grep` 将会非常简单，你也会掌握阅读现实生活中项目的基础知识。

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，我还会简要的讲到闭包、迭代器和 trait 对象，他们分别会在第XX、YY和ZZ章详细介绍。

让我们一如既往的使用`cargo new`创建一个新项目：

```
$ cargo new --bin greprs
     Created binary (application) `greprs` project
$ cd greprs
```

我们版本的`grep`的叫做“greprs”，这样就不会迷惑用户让他们以为这就是可能已经在系统上安装了功能更完整的`grep`。

接受命令行参数

> ch12-01-accepting-command-line-arguments.md
> commit 4f2dc564851dc04b271a2260c834643dfd86c724

第一个任务是让`greprs`接受两个命令行参数。crates.io 上有一些现存的库可以帮助我们，不过因为我们正在学习，我们将自己实现一个。

我们需要调用一个 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。但是对于我们的目的来说，使用他们并不需要知道多少技术细节。我们只需要明白两点：

1. 迭代器生成一系列的值。
2. 在迭代器上调用`collect`方法可以将其生成的元素转换为一个 vector。

让我们试试列表 12-1 中的代码：

Filename: src/main.rs

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
    println!("{:?}", args);
}
```

Listing 12-1: Collect the command line arguments into a vector and print them out

首先使用`use`语句来将`std::env`模块引入作用域。当函数嵌套了多于一层模块时，比如说`std::env::args`，通常使用`use`将父模块引入作用域，而不是引入其本身。`env::args`比单独的`args`要明确一些。当然，如果使用了多余一个`std::env`中的函数，我们也只需要一个`use`语句。

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 vector。这里我们也显式的注明了`args`的类型：`collect`可以被用来创建很多类型的集合。Rust 并不能推断出我们需要什么类型，所以类型注解是必须的。在 Rust 中我们很少会需要注明类型，不过`collect`是就一个通常需要这么做的函数。

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/greprs"]

$ cargo run needle haystack
...snip...
["target/debug/greprs", "needle", "haystack"]
```

你会注意一个有趣的事情：二进制文件的名字是第一个参数。其原因超出了本章介绍的范围，不过这是我们必须记住的。

现在我们有了一个访问所有参数的方法，让我们如列表 12-2 中所示将需要的变量存放到变量中：

Filename: src/main.rs

```
use std::env;

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

    let search = &args[1];
    let filename = &args[2];

    println!("Searching for {}", search);
    println!("In file {}", filename);
}
```

Listing 12-2: Create variables to hold the search argument and filename argument

记住，程序名称是是第一个参数，所以并不需要`args[0]`。我们决定从第一个参数将是需要搜索的字符串，所以将第一个参数的引用放入变量`search`中。第二个参数将是文件名，将其放入变量`filename`中。再次尝试运行程序：

```
$ cargo run test sample.txt
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target\debug\greprs.exe test sample.txt`
Searching for test
In file sample.txt
```

很棒！不过有一个问题。让我们不带参数运行：

```
$ cargo run
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target\debug\greprs.exe`
thread 'main' panicked at 'index out of bounds: the len is 1
but the index is 1', ../src/libcollections\vec.rs:1307
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

因为 vector 中只有一个元素，就是程序名称，不过我们尝试访问第二元素，程序 panic 并提示越界访问。虽然这个错误信息是_准确的_，不过它对程序的用户来说就没有意义了。现在就可以修复这个问题，不过我先继续学习别的内容：在程序结束前我们会改善这个情况。

读取文件

ch12-02-reading-a-file.md
commit 4f2dc564851dc04b271a2260c834643dfd86c724

现在有了一些包含我们需要的信息的变量了，让我们试着使用他们。下一步目标是打开需要搜索的文件。为此，我需要一个文件。在项目的根目录创建一个文件`poem.txt`，并写入一些艾米莉·狄金森（Emily Dickinson）的诗：

Filename: 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!
```

创建完这个文件后，让我们编辑 *src/main.rs* 并增加如列表 12-3 所示用来打开文件的代码：

Filename: src/main.rs

```
use std::env;
use std::fs::File;
use std::io::prelude::*;

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

    let search = &args[1];
    let filename = &args[2];

    println!("Searching for {}", search);
    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);
}
```

Listing 12-3: Read the contents of the file specified by the second argument

这里增加了一些新内容。首先，需要更多的`use`语句来引入标准库中的相关部分：我们需要`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 时需要的内容。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，我们获取了文件的句柄并使用`File::open`函数与第二个参数中指定的文件名来打开这个文件。第二，我们在变量`contents`中创建了一个空的可变的`String`，接着对文件句柄调用`read_to_string`并以`contents`字符串作为参数，`contents`是`read_to_string`将会放置它读取到的数据地方。最后，我们打印出了整个文件的内容，这是一个确认目前为止的程序能够工作的方法。

尝试运行这些代码，随意指定第一个参数（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 debug [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target\debug\greprs.exe 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！我们的代码可以工作了！然而，它还有一些瑕疵。因为程序还很小，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。现在就让我们开始重构而不是等待之后处理。重构在只有少量代码时会显得容易得多。

读取文件

ch12-03-improving-error-handling-and-modularity.md
commit 4f2dc564851dc04b271a2260c834643dfd86c724

为了完善我们程序有四个问题需要修复，而他们都与潜在的错误和程序结构有关。第一个问题是在哪打开文件：我们使用了`expect`来在打开文件失败时指定一个错误信息，不过这个错误信息只是说“文件不存在”。还有很多打开文件失败的方式，不过我们总是假设是由于缺少文件导致的。例如，文件存在但是没有打开它的权限：这时，我们就打印出了错误不符合事实的错误信息！

第二，我们不停的使用`expect`，这就有点类似我们之前在不传递任何命令行参数时索引会`panic!`时注意到的问题：这虽然时_可以工作_的，不过这有点没有原则性，而且整个程序中都需要他们，将错误处理都置于一处则会显得好很多。

第三个问题是`main`函数现在处理两个工作：解析参数，并打开文件。对于一个小的函数来说，这不是什么大问题。然而随着程序中的`main`函数不断增长，`main`函数中独立的任务也会越来越多。因为一个函数拥有很多职责，它将难以理解、难以测试并难以在不破坏其他部分的情况下做出修改。

这也关系到我们的第四个问题：`search`和`filename`是程序中配置性的变量，而像`f`和`contents`则用来执行程序逻辑。随着`main`函数增长，将引入更多的变量到作用域中，而当作用域中有更多的变量，将更难以追踪哪个变量用于什么目的。如果能够将配置型变量组织进一个结构就能使他们的目的更明确了。

让我们重新组成程序来解决这些问题。

二进制项目的关注分离

这类项目组织上的问题在很多相似类型的项目中很常见，所以 Rust 社区开发出一种关注分离的组织模式。这种模式可以用来组织任何用 Rust 构建的二进制项目，所以可以证明应该更早的开始这项重构，以为我们的项目符合这个模式。这个模式看起来像这样：

1. 将程序拆分成 *main.rs* 和 *lib.rs*。
2. 将命令行参数解析逻辑放入 *main.rs*。
3. 将程序逻辑放入 *lib.rs*。
4. `main` 函数的工作是：
    - 解析参数
    - 设置所有配置性变量
    - 调用 *lib.rs* 中的 `run` 函数
    - 如果 `run` 返回错误则处理这个错误

好的！老实说这个模式好像还很复杂。这就是关注分离的所有内容：*main.rs* 负责实际的程序运行，而 *lib.rs* 处理所有真正的任务逻辑。让我们将程序重构成这种模式。首先，提取出一个目的只在于解析参数的函数。列表 12-4 中展示了一个新的开始，`main` 函数调用了一个新函数 `parse_config`，它仍然定义于 *src/main.rs* 中：

Filename: src/main.rs

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

    let (search, filename) = parse_config(&args);

    println!("Searching for {}", search);
    println!("In file {}", filename);

    // ...snip...
}

fn parse_config(args: &[String]) -> (&str, &str) {
    let search = &args[1];
    let filename = &args[2];

    (search, filename)
}
```

Listing 12-4: Extract a `parse_config` function from `main`

这看起来好像有点复杂，不过我们将一点一点的开展重构。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，这样在遇到问题时就能更好地理解什么修改造成了错误。

组合配置值

现在我们有了一个函数了，让我们接着完善它。我们代码还能设计的更好一些：函数返回了一个元组，不过接着立刻就解构成了单独的部分。这些代码本身没有问题，不过有一个地方表明仍有改善的余地：我们调用了 `parse_config` 方法。函数名中的 `config` 部分也表明了返回的两个值应该是组合在一起的，因为他们都是某个配置值的一部分。

> 注意：一些同学将当使用符合类型更为合适的时候使用基本类型当作一种称为**基本类型偏执**（*primitive obsession*）的反模式。

让我们引入一个结构体来存放所有的配置。列表 12-5 中展示了新增的`Config`结构体定义、重构后的`parse_config`和`main`函数中的相关更新：

Filename: src/main.rs

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

    let config = parse_config(&args);

    println!("Searching for {}", config.search);
    println!("In file {}", config.filename);

    let mut f = File::open(config.filename).expect("file not found");

    // ...snip...
}

struct Config {
    search: String,
    filename: String,
}

fn parse_config(args: &[String]) -> Config {
    let search = args[1].clone();
    let filename = args[2].clone();

    Config {
        search: search,
        filename: filename,
    }
}
```

Listing 12-5: Refactoring `parse_config` to return an instance of a `Config` struct

`parse_config`的签名现在表明它返回一个`Config`值。在`parse_config`的函数体中，我们之前返回了`args`中`String`值引用的字符串 slice，不过`Config`定义为拥有两个有所有权的`String`值。因为`parse_config`的参数是一个`String`值的 slice，`Config`实例不能获取`String`值的所有权：这违反了 Rust 的借用规则，因为`main`函数中的`args`变量拥有这些`String`值并只允许`parse_config`函数借用他们。

还有许多不同的方式可以处理`String`的数据；现在我们使用简单但低效率的方式，在字符串 slice 上调用`clone`方法。`clone`调用会生成一个字符串数据的完整拷贝，而且`Config`实例可以拥有它，不过这会消耗更多时间和内存来储存拷贝字符串数据的引用，不过拷贝数据让我们使我们的代码显得更加直白。

使用`clone`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不使用`clone`来解决所有权问题。在关于迭代器的第XX章中，我们将会学习如何更有效率的处理这种情况。现在，为了编写我们的程序拷贝一些字符串是没有问题。我们只进行了一次拷贝，而且文件名和要搜索的字符

串都比较短。随着你对 Rust 更加熟练，将更轻松的省略这个权衡的步骤，不过现在调用 `clone` 是完全可以接受的。

`main` 函数更新为将 `parse_config` 返回的 `Config` 实例放入变量 `config` 中，并将分别使用 `search` 和 `filename` 变量的代码更新为使用 `Config` 结构体的字段。

创建一个 `Config` 构造函数

现在让我们考虑一下 `parse_config` 的目的：这是一个创建 `Config` 示例的函数。我们已经见过了一个创建实例函数的规范：像 `String::new` 这样的 `new` 函数。列表 12-6 中展示了将 `parse_config` 转换为一个 `Config` 结构体关联函数 `new` 的代码：

Filename: src/main.rs

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

    let config = Config::new(&args);

    println!("Searching for {}", config.search);
    println!("In file {}", config.filename);

    // ...snip...


}

// ...snip...

impl Config {
    fn new(args: &[String]) -> Config {
        let search = args[1].clone();
        let filename = args[2].clone();

        Config {
            search: search,
            filename: filename,
        }
    }
}
```

Listing 12-6: Changing `parse_config` into `Config::new`

我们将 `parse_config` 的名字改为 `new` 并将其移动到 `impl` 块中。我们也更新了 `main` 中的调用代码。再次尝试编译并确保程序可以运行。

从构造函数返回 `Result`

这是我们对这个方法最后的重构：还记得当 vector 含有少于三个项时访问索引 1 和 2 会 panic 并给出一个糟糕的错误信息的代码吗？让我们来修改它！列表 12-7 展示了如何在访问这些位置之前检查 slice 是否足够长，并使用一个更好的 panic 信息：

Filename: src/main.rs

```
// ...snip...
fn new(args: &[String]) -> Config {
    if args.len() < 3 {
        panic!("not enough arguments");
    }

    let search = args[1].clone();
    // ...snip...
}
```

Listing 12-7: Adding a check for the number of arguments

通过在`new`中添加这额外的几行代码，再次尝试不带参数运行程序：

```
$ cargo run
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target\debug\greprs.exe`
thread 'main' panicked at 'not enough arguments', src\main.rs:29
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

这样就好多了！至少有个一个符合常理的错误信息。然而，还有一堆额外的信息我们并不希望提供给用户。可以通过改变`new`的签名来完善它。现在它只返回了一个`Config`，所有没有办法表示创建`Config`失败的情况。相反，可以如列表 12-8 所示返回一个`Result`：

Filename: src/main.rs

```
impl Config {
    fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let search = args[1].clone();
        let filename = args[2].clone();

        Ok(Config {
            search: search,
            filename: filename,
        })
    }
}
```

Listing 12-8: Return a `Result` from `Config::new`

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"静态声明周期"中讲到`&'static str`是一个字符串字面值，他也是现在我们的错误信息。

`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。

`Config::new`调用和错误处理

现在我们需要对`main`做一些修改，如列表 12-9 所示：

Filename: src/main.rs

```
// ...snip...
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);
    });

    println!("Searching for {}", config.search);
    println!("In file {}", config.filename);

    // ...snip...
```

Listing 12-9: Exiting with an error code if creating a new `Config` fails

新增了一个`use`行来从标准库中导入`process`。在`main`函数中我们将处理`new`函数返回的`Result`值，并在其返回`Config::new`时以一种更加清楚的方式结束进程。

这里使用了一个之前没有讲到的标准库中定义的`Result<T, E>`的方法：`unwrap_or_else`。当`Result`是`Ok`时其行为类似于`unwrap`：它返回`Ok`内部封装的值。与`unwrap`不同的是，当`Result`是`Err`时，它调用一个**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XX章会更详细的介绍闭包；这里需要理解的重要部分是`unwrap_or_else`会将`Err`的内部值传递给闭包中位于两道竖线间的参数`err`。使用`unwrap_or_else`允许我们进行一些自定义的非`panic!`的错误处理。

上述的错误处理其实只有两行：我们打印出了错误，接着调用了`std::process::exit`。这个函数立刻停止程序的执行并将传递给它的数组作为返回码。依照惯例，零代表成功而任何其他数字表示失败。就结果来说这依然类似于列表 12-7 中的基于`panic!`的错误处理，但是不再会有额外的输出了，让我们试一试：

```
$ cargo run
   Compiling greprs v0.1.0 (file:///projects/greprs)
    Finished debug [unoptimized + debuginfo] target(s) in 0.48 secs
     Running `target\debug\greprs.exe`
Problem parsing arguments: not enough arguments
```

非常好！现在输出就友好多了。

`run`函数中的错误处理

现在重构完了参数解析部分，让我们再改进一下程序的逻辑。列表 12-10 中展示了在`main`函数中调用提取出函数`run`之后的代码。`run`函数包含之前位于`main`中的部分代码：

Filename: src/main.rs

```
fn main() {
    // ...snip...

    println!("Searching for {}", config.search);
    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...
```

Listing 12-10: Extracting a `run` functionality for the rest of the program logic

`run`函数的内容是之前位于`main`中的几行，而且`run`函数获取一个`Config`作为参数。现在有了一个单独的函数了，我们就可以像列表 12-8 中的`Config::new`那样进行类似的改进了。列表 12-11 展示了另一个`use`语句将`std::error::Error`结构引入了作用域，还有使`run`函数返回`Result`的修改：

Filename: 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(())
}
```

Listing 12-11: Changing the `run` function to return `Result`

这里有三个大的修改。第一个是现在`run`函数的返回值是`Result<(), Box<Error>>`类型的。之前，函数返回 unit 类型`()`，现在它仍然是`Ok`时的返回值。对于错误类型，我们将使用`Box<Error>`。这是一个**trait 对象**（*trait object*），第XX章会讲到。现在可以这样理解它：`Box<Error>`意味着函数返回了某个实现了`Error` trait 的类型，不过并没有指定具体的返回值类型。这样就比较灵活，因为在不同的错误场景可能有不同类型的错误返回值。`Box`是一个堆数据的智能指针，第YY章将会详细介绍`Box`。

第二个改变是我们去掉了`expect`调用并替换为第9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`()`，所以需要将 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 尝试告诉我们忽略`Result`，它有可能是一个错误值。让我们现在来处理它。我们将采用类似于列表 12-9 中处理`Config::new`错误的技巧，不过还有少许不同：

Filename: src/main.rs

```
fn main() {
    // ...snip...

    println!("Searching for {}", config.search);
    println!("In file {}", config.filename);

    if let Err(e) = run(config) {
        println!("Application error: {}", e);

        process::exit(1);
    }
}

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(())
}
```

不同于`unwrap_or_else`，我们使用`if let`来检查`run`是否返回`Err`，如果是则调用`process::exit(1)`。为什么呢？这个例子和`Config::new`的区别有些微妙。对于`Config::new`我们关心两件事：

1. 检测出任何可能发生的错误
2. 如果没有出现错误创建一个`Config`

而在这个情况下，因为`run`在成功的时候返回一个`()`，唯一需要担心的就是第一件事：检测错误。如果我们使用了`unwrap_or_else`，则会得到`()`的返回值。它并没有什么用处。

虽然两种情况下`if let`和`unwrap_or_else`的内容都是一样的：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！还有一件我们尚未开始的工作：拆分 *src/main.rs* 并将一些代码放入 *src/lib.rs* 中。让我们现在就开始吧：将 *src/main.rs* 中的`run`函数移动到新建的 *src/lib.rs* 中。还需要移动相关的`use`语句和`Config`的定义，以及其`new`方法。现在 *src/lib.rs* 应该如列表 12-12 所示：

Filename: src/lib.rs

```
use std::error::Error;
use std::fs::File;
use std::io::prelude::*;

pub struct Config {
    pub search: String,
    pub filename: String,
}

impl Config {
    pub fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let search = args[1].clone();
        let filename = args[2].clone();

        Ok(Config {
            search: search,
            filename: filename,
        })
    }
}

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)?;

    println!("With text:\n{}", contents);

    Ok(())
}
```

Listing 12-12: Moving `Config` and `run` into *src/lib.rs*

注意我们还需要使用公有的`pub`：在`Config`和其字段、它的`new`方法和`run`函数上。

现在在 *src/main.rs* 中，我们需要通过`extern crate greprs`来引入现在位于 *src/lib.rs* 的代码。接着需要增加一行`use greprs::Config`来引入`Config`到作用域，并对`run`函数加上 crate 名称前缀，如列表 12-13 所示：

Filename: src/main.rs

```
extern crate greprs;

use std::env;
use std::process;

use greprs::Config;

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);
    });

    println!("Searching for {}", config.search);
    println!("In file {}", config.filename);

    if let Err(e) = greprs::run(config) {
        println!("Application error: {}", e);

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

Listing 12-13: Bringing the `greprs` crate into the scope of *src/main.rs*

通过这些重构，所有代码应该都能运行了。运行几次`cargo run`来确保你没有破坏什么内容。好的！确实有很多的内容，不过已经为将来的成功奠定了基础。我们采用了一种更加优秀的方式来处理错误，并使得代码更模块化了一些。从现在开始几乎所有的工作都将在 *src/lib.rs* 中进行。

让我们利用这新创建的模块的优势来进行一些在旧代码中难以开开展的工作，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 4f2dc564851dc04b271a2260c834643dfd86c724

现在为项目的核心功能编写测试将更加容易，因为我们将逻辑提取到了 *src/lib.rs* 中并将参数解析和错误处理都留在了 *src/main.rs* 里。现在我们可以直接使用多种参数调用代码并检查返回值而不用从命令行运行二进制文件了。

我们将要编写的是一个叫做`grep`的函数，它获取要搜索的项以及文本并产生一个搜索结果列表。让我们从`run`中去掉那行`println!`（也去掉 *src/main.rs* 中的，因为再也不需要他们了），并使用之前收集的选项来调用新的`grep`函数。眼下我们只增加一个空的实现，和指定`grep`期望行为的测试。当然，这个测试对于空的实现来说是会失败的，不过可以确保代码是可以编译的并得到期望的错误信息。列表 12-14 展示了这些修改：

Filename: src/lib.rs

```
// ...snip...

fn grep<'a>(search: &str, contents: &'a str) -> Vec<&'a str> {
    vec![]
}

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)?;

    grep(&config.search, &contents);

    Ok(())
}

#[cfg(test)]
mod test {
    use grep;

    #[test]
    fn one_result() {
        let search = "duct";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.";

        assert_eq!(
            vec!["safe, fast, productive."],
            grep(search, contents)
        );
    }
}
```

Listing 12-14: Creating a function where our logic will go and a failing test for that function

注意需要在`grep`的签名中显式声明声明周期`'a`并用于`contents`参数和返回值。记住，生命周期参数用于指定函数参数于返回值的生命周期的关系。在这个例子中，我们表明返回的 vector 将包含引用参数`contents`的字符串 slice，而不是引用参数`search`的字符串 slice。换一种说法就是我们告诉 Rust 函数`grep`返回的数据将和传递给它的参数`contents`的数据存活的同样久。这是非常重要的！考虑为了使引用有效则 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`search`而不是`contents`的 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:37:46
   |
37 | fn grep(search: &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 `search` 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，而我们知道`contents`是应该要使用生命周期语法来与返回值相关联的参数。

在函数签名中将参数与返回值相关联是其他语言不会让你做的工作，所以不用担心这感觉很奇怪！掌握如何指定生命周期会随着时间的推移越来越容易，熟能生巧。你可能想要重新阅读上一部分或返回与第十章中生命周期语法部分的例子做对比。

现在试试运行测试：

```
$ cargo test
...warnings...
    Finished debug [unoptimized + debuginfo] target(s) in 0.43 secs
     Running target/debug/deps/greprs-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！之所以会失败是因为我们总是返回一个空的 vector。如下是如何实现 `grep` 的步骤：

1. 遍历每一行文本。
2. 查看这一行是否包含要搜索的字符串。
    - 如果有，将这一行加入返回列表中
    - 如果没有，什么也不做
3. 返回匹配到的列表

让我们一步一步的来，从遍历每行开始。字符串类型有一个有用的方法来处理这种情况，它刚好叫做 `lines` ：

Filename: src/lib.rs

```
fn grep<'a>(search: &str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        // do something with line
    }
}
```

我们使用了一个 `for` 循环和 `lines` 方法来依次获得每一行。接下来，让我们看看这些行是否包含要搜索的字符串。幸运的是，字符串类型为此也有一个有用的方法 `contains` ！ `contains` 的用法看起来像这样：

Filename: src/lib.rs

```
fn grep<'a>(search: &str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        if line.contains(search) {
            // do something with line
        }
    }
}
```

最终，我们需要一个方法来存储包含要搜索字符串的行。为此可以在`for`循环之前创建一个可变的 vector 并调用`push`方法来存放一个`line`。在`for`循环之后，返回这个 vector。列表 12-15 中为完整的实现：

Filename: src/lib.rs

```
fn grep<'a>(search: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(search) {
            results.push(line);
        }
    }

    results
}
```

Listing 12-15: Fully functioning implementation of the `grep` function

尝试运行一下：

```
$ cargo test
running 1 test
test test::one_result ... ok

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

     Running target/debug/greprs-2f55ee8cd1721808

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests greprs

running 0 tests

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

非常好！它可以工作了。现在测试通过了，我们可以考虑一下重构`grep`的实现并时刻保持其功能不变。这些代码并不坏，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探索迭代器和如何改进代码。

现在`grep`函数是可以工作的，我们还需在在`run`函数中做最后一件事：还没有打印出结果呢！增加一个`for`循环来打印出`grep`函数返回的每一行：

Filename: 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 grep(&config.search, &contents) {
        println!("{}", line);
    }

    Ok(())
}
```

现在程序应该能正常运行了！试试吧：

```
$ cargo run the poem.txt
   Compiling greprs v0.1.0 (file:///projects/greprs)
    Finished debug [unoptimized + debuginfo] target(s) in 0.38 secs
     Running `target\debug\greprs.exe the poem.txt`
Then there's a pair of us - don't tell!
To tell your name the livelong day

$ cargo run a poem.txt
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target\debug\greprs.exe a poem.txt`
I'm nobody! Who are you?
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!
```

好极了！我们创建了一个属于自己的经典工具，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。

处理环境变量

> ch12-05-working-with-environment-variables.md
> commit 4f2dc564851dc04b271a2260c834643dfd86c724

让我们再增加一个功能：大小写不敏感搜索。另外，这个设定将不是一个命令行参数：相反它将是一个环境变量。当然可以选择创建一个大小写不敏感的命令行参数，不过用户要求提供一个环境变量这样设置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了。

实现并测试一个大小写不敏感 `grep` 函数

首先，让我们增加一个新函数，当设置了环境变量时会调用它。增加一个新测试并重命名已经存在的那个：

```
#[cfg(test)]
mod test {
    use {grep, grep_case_insensitive};

    #[test]
    fn case_sensitive() {
        let search = "duct";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.
Duct tape.";

        assert_eq!(
            vec!["safe, fast, productive."],
            grep(search, contents)
        );
    }

    #[test]
    fn case_insensitive() {
        let search = "rust";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.
Trust me.";

        assert_eq!(
            vec!["Rust:", "Trust me."],
            grep_case_insensitive(search, contents)
        );
    }
}
```

我们将定义一个叫做 `grep_case_insensitive` 的新函数。它的实现与 `grep` 函数大体上相似，不过列表 12-16 展示了一些小的区别：

Filename: src/lib.rs

```
fn grep_case_insensitive<'a>(search: &str, contents: &'a str) -> Vec<&'a str> {
    let search = search.to_lowercase();
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.to_lowercase().contains(&search) {
            results.push(line);
        }
    }

    results
}
```

Listing 12-16: Implementing a `grep_case_insensitive` function by changing the search string and the lines of the contents to lowercase before comparing them

首先，将 `search` 字符串转换为小写，并存放于一个同名的覆盖变量中。注意现在 `search` 是一个 `String` 而不是字符串 slice，所以在将 `search` 传递给 `contains` 时需要加上 &，因为 `contains` 获取一个字符串 slice。