目錄

前言

- GitHub: https://github.com/KaiserY/rust-book-chinese
- GitBook: https://www.gitbook.com/book/kaisery/rust-book-chinese
- Rust中文社区: http://rust.cc/
- QQ群: 144605258

1.1.0-stable

贡献者

> 惯例排名不分先后

- armink
- Bluek404
- hczhcz
- JaySon-Huang
- KaiserY
- kenshinji
- leqinotes
- liubin
- liuzhe0223
- ustcscgy
- 1989car

Rust编程语言

欢迎阅读！这本书将教会你使用Rust编程语言。Rust是一个注重安全与速度的现代系统编程语言，通过在没有垃圾回收的情况下保证内存安全来实现它的目标，这使它成为一个在很多其它语言不适合的用例中大展身手的语言：嵌入到其它语言中，在特定的时间和空间要求下编程，和编写底层代码，例如设备驱动和操作系统。它通过一系列的不产生运行时开销的编译时安全检查来提升目前语言所关注的这个领域，同时消除一切数据竞争。Rust同时也意在实现“零开销抽象”，即便在这些抽象看起来比较像一个高级语言的特性。即便如此，Rust也允许你像一个底层语言那样进行精确的控制。

《Rust编程语言》被分为7个部分。这个介绍是第一部分。之后是：

- 准备 - 为你的电脑安装Rust开发环境
- 学习Rust - 通过小项目来学习Rust编程
- 高效Rust - 编写优秀Rust代码的高级内容
- 语法和语义 - Rust各个部分，被拆分成小的部分讲解
- Rust开发版 - 还未出现在稳定版本中的最新功能
- 词汇表 - 书中使用的术语的参考
- 学院派研究 - 影响过Rust的文献

在阅读了介绍这部分之后，你可以根据喜好深入到“学习Rust”或“语法和语义”部分：如果你想通过项目深入了解，可以先选择“学习Rust”;如果你想从头开始，并且学习一个完整的内容再学习另一个，你可以从“语法和语义”开始。丰富的交叉连接将这些部分联系到一起。

贡献

生成这本书的源文件可以在GitHub上找到：github.com/rust-lang/rust/tree/master/src/doc/trpl

Rust简介

Rust是你会感兴趣的语言吗？让我们检查一些小的代码例子来展示它的部分威力。

使Rust显得独一无二的主要概念是“所有权”。考虑这个小例子：

```
fn main() {
    let mut x = vec!["Hello", "world"];
}
```

这个程序创建了一个叫做`x`的变量绑定。这个绑定的值是一个`Vec<T>`，一个vector，我们通过一个定义在标准库中的宏来创建它。这个宏叫做`vec`，并且我们通过一个`!`调用宏。这遵循了Rust的一般原则：让一切明了。宏可以做比函数调用复杂的多的多的工作，并且它们在视觉上也是有区别的。`!`也方便了解析，更容易编写工具，这也是很重要的。

我们使用了`mut`来使`x`可变：再Rust中绑定是默认是不可变的。在下面的例子中这个vector是可变的。

另外值得注意的是这里我们并不需要一个类型注释：因为Rust是静态类型的，我们并不需要显式的标明类

型。Rust拥有类型推断来平衡静态类型的能力和类型注释的冗余。

Rust与堆分配相比倾向于栈分配：`x`被直接储存在栈上。然而，`Vec<T>`类型在堆上为vector的元素分配了空间。如果你并不熟悉这里的区别，目前你可以忽略它，或者看看“栈与堆”。作为一个系统编程语言，Rust给予你控制内存分配的能力，不过当我们上手后，这并不是什么大问题。

之前，我们提到“所有权”是Rust中的一个关键概念。在Rust用语中，`x`被认为“拥有”这个vector。这意味着当`x`离开作用域，vector的内存将被销毁。这由Rust编译器决定，而不是通过类似垃圾回收器这样的机制。换句话说，在Rust中，你并不需要自己调用像`malloc`和`free`这样的函数：编译器静态决定何时你需要分配和销毁内存，并自动调用这些函数。人非圣贤孰能无过,不过编译器永远也不会忘记。

让我们为例子再加一行：

```
fn main() {
    let mut x = vec!["Hello", "world"];

    let y = &x[0];
}
```

我们引入了另一个绑定，`y`。在这个例子中，`y`是对vector第一个元素的“引用”。Rust的引用类似于其它语言中的指针，不过带有额外的编译时安全检查。引用用“借用”它指向的内容，而不是拥有它，来与所有权系统交互。这里的区别是，当一个引用离开作用域，它不会释放之下的内存。如果它这么做了，我们会释放两次，这是很糟的！。

让我们增加第三行。这看起来并不会引起错误，不过实际上会造成一个编译错误：

```
fn main() {
    let mut x = vec!["Hello", "world"];

    let y = &x[0];

    x.push("foo");
}
```

`push`是vector的一个方法，它在vector的末尾附加另一个元素。当我们尝试编译这个程序时，我们得到一个错误：

```
error: cannot borrow `x` as mutable because it is also borrowed as immutable
    x.push(4);
    ^
note: previous borrow of `x` occurs here; the immutable borrow prevents
subsequent moves or mutable borrows of `x` until the borrow ends
    let y = &x[0];
             ^
note: previous borrow ends here
fn main() {

}
^
```

噢！Rust编译器有时给出灰常详细的错误，而这就是其中之一。正如错误所解释的，当我们让绑定可变，我们仍不能调用`push`。这是因为我们已经有了一个vector元素的引用，`y`。当有其它引用存在时改变值是危险的，因为我们可能使这个引用无效。在这个特定的例子中，当我们创建了vector，我们可能只分配了3个元素的空间。增加一个元素意味着将分配一个新的能放下所有4个元素的空间，拷贝旧的值，并更新内部的指针指向这个内存。所有这些都木有问题。问题是`y`并没有被更新，很糟糕地，`y`成了一个“悬垂指针”(dangling pointer)。因此，在这个例子中任何对`y`的使用都会引起错误，而编译器会为我们捕获了这个错误。

那么我们应该如何解决这个问题呢？这里我们可以采取两个方法。第一个方法是使用拷贝而不使用引用：

```
fn main() {
    let mut x = vec!["Hello", "world"];

    let y = x[0].clone();

    x.push("foo");
}
```

Rust默认拥有移动语义，所以如果我们想要拷贝一些数据，我们调用`clone()`方法。在这个例子中，`y`不再是一个储存在`x`中vector的一个引用，而是它第一个元素的拷贝，`"hello"`。现在我们并不拥有一个引用，所以`push()`就能正常工作。

如果我们真心需要一个引用，我们需要另一种方法：确保在我们尝试修改之前，让引用离开作用域。如下：

```
fn main() {
    let mut x = vec!["Hello", "world"];

    {
        let y = &x[0];
    }

    x.push("foo");
}
```

我们用一对大括号创建了一个内部作用域，`y`会在我们调用`push()`之前离开作用域，所以我们不会碰到问题。

所有权的概念并不仅仅善于防止悬垂指针，也解决了一整个系列的相关问题，比如迭代器无效，并发和其它问题。

准备

本书的第一（二）部分将会带你了解Rust和它的工具。第一，我们将会安装Rust。接着：经典的“‘Hello World”程序。最后，我们谈谈Cargo，Rust的系统和包管理器。

安装Rust

开始使用Rust的第一步是安装它！有很多种安装Rust的方法，其中最简单的是使用*rustup*脚本.如果你使用Linux或者Mac系统，你只需要输入以下脚本（注意你并不需要输入`$`，它们代表每行指令的开头）：

```
$ curl -L https://static.rust-lang.org/rustup.sh | sudo sh
```

如果你担心使用`curl | sudo sh`的潜在不安全性，请继续阅读并查看我们下面的免责声明。并且你也可以随意使用下面这个两步安装脚本以便可以检查我们的安装脚本：

```
$ curl -L https://static.rust-lang.org/rustup.sh -O
$ sudo sh rustup.sh
```

如果你用Windows，请直接下载32位或者64位安装包然后运行即可。

卸载

如果不幸的，你再也不想使用Rust了:(，当然这不要紧。也许Rust不是你的菜（原文：不是所有人都会认为什么语言非常好）。运行下面的卸载脚本：

```
$ sudo /usr/local/lib/rustlib/uninstall.sh
```

如果你使用Windows安装包进行安装的话，重新运行`.exe`文件，它会提供一个卸载选项。

你可以在任何时候重新运行脚本来更新Rust。在现在这个时间，你将会频繁更新Rust，因为Rust还未发布1，0版本，经常更新人们会认为你在使用最新版本的Rust。

不过这带来了另外一个问题（传说中的免责声明？）：一些同学确实有理由对我们让他们运行`curl | sudo sh`感到非常反感。他们理应如此。从根本上说，当你运行上面的脚本时，代表你相信是一些好人在维护Rust，他们不会黑了你的电脑做坏事。对此保持警觉是一个很好的天性。如果你是这些强迫症患者（大雾），请检阅以下文档，从源码编译Rust或者官方二进制文件下载。

当然，我们需要提到官方支持的平台：

- Windows (7, 8, Server 2008 R2)
- Linux (2.6.18 or later, various distributions), x86 and x86-64
- OSX 10.7 (Lion) or greater, x86 and x86-64

Rust在以上平台进行了广泛的测试，当然还在一些其他平台，比如Android。不过进行了越多测试的环境，越有可能正常工作。

最后，一关于Windows说明。Rust将Windows作为第一级平台来发布，不过说实话，WIndows平台整体体验并没有Linux/OS X那么好。我们正在改善它！如果有情况它不能工作了，这是一个bug。如果这种发生了，请让我知道。所有和每一次提交都在Windows下进行了测试，就像其它平台一样。

如果你已安装Rust，你可以打开一个Shell，然后输入：

```
$ rustc --version
```

你应该看到版本号，提交的hash值，提交时间和构建时间：

```
rustc 1.0.0-beta (9854143cb 2015-04-02) (built 2015-04-02)
```

如果你做到了，那么Rust已经正确安装！此处应该有掌声！

安装程序（脚本）也会在本地安装一份文档拷贝，所以你可以离线阅读它们。在UN*X系统上，位置是`/usr/local/share/doc/rust`。在Windows，它位于你Rust安装目的`share/doc`文件夹。

如果你遇到什么错误，这里有几个地方你可以获取帮助。最简单的是通过Mibbit访问Rust IRC频道irc.mozilla.org。点击上面的链接，你就可以与其它Rustaceans（简单理解为Ruster吧）聊天，我们会帮助你。其它的地方有the /r/rust subreddit和Stack Overflow。

Hello, world!

现在你已经安装好了Rust，让我们开始写第一个Rust程序吧。作为一个传统，你任何新语言的第一个程序应该在屏幕上打印出“Hello, world!”。写这么一个小程序有一个好处就是你可以确认你安装的Rust编译器不仅仅是装上了而已。并且在屏幕上打印信息是一件非常常见的事情。

你需要做的第一件事就是创建一个用来写代码的文件。我喜欢在home目录下创建一个projects文件夹，用来存放我的所有项目。Rust并不关心你的代码位于何处。

这里确实有另一个导致我担心的地方：这个教程假定你熟悉基本的命令行操作。Rust并不对你使用的编辑工具和代码位置有特定的要求。如果相比命令行你更倾向于IDE，你也许想要看看SolidOak，或者你最喜欢的IDE相关插件。这里有一系正在开发中的来自社区的不同质量的扩展。Rust团队也发布了不同编辑器的插件。特别的，配置你的编辑器或IDE超出了本教程的范围，所以在你开始前查看文档。

既然这么说了，让我们在projects目录下新建一个目录

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

如果你使用Windows并且没有用PowerShell，`~`可能无效。查询你所使用的Shell的文档以获取更多信息。

接下来让我们新建一个源文件。我们叫我们的文件`main.rs`。Rust源文件总是使用`.rs`后缀。如果你使用了多个词，使用下划线分隔。写成`hello_world.rs`而不是`helloworld.rs`。

现在打开你的源文件，键入如下代码：

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

保存文件，然后在终端中输入如下命令：

```
$ rustc main.rs
$ ./main # or main.exe on Windows
Hello, world!
```

搞定这些，让我们回过头来看看到底发生了什么

```
fn main() {

}
```

这几行定义了一个Rust函数。`main`函数是特殊的：这是所有Rust程序的开始。第一行表示“我定义了一个

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（`(` 和 `)`）中。因为我们并没有返回任何东西，我们可以完全省略返回类型。我们稍后将谈论这些。

你会注意到函数被大括号（`{` 和 `}`）包围起来。Rust要求大括号中包含所有函数体。将左大括号与函数定义置于一排并留有一个空格被认为是一个好的代码格式。

接下来是这一行：

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

这一行做了我们这小程序的所有工作。这里有一些细节比较重要。第一，缩进是4个空格，而不是制表符（Tab）。请设置你的编辑器Tab键为插入4个空格。我们提供多种编辑器的配置例子。

第二点是`println!()`部分。这是一个Rust宏，是Rust元编程的关键所在。如果你使用函数的话应该写成`println()`。在这里，我们并不用担心这两者的区别。只需记住，有时你会看到一个`!`，那代表你调用了一个宏而不是一个普通的函数。有很多理由使得Rust实现了`println!`宏而不只是一个函数，不过这涉及一些非常高端的话题。你会在我们谈论宏的时候了解更多。最后提醒一点：Rust宏与C语言的宏有显著区别，万一你用过C语言宏的话。使用宏时请不要恐惧。我们最终会仔细研究它，现在你只需相信我们。

下一部分，`"Hello, world!"`是一个字符串。在一门系统编程语言中，字符串是一个复杂得令人惊讶的话题。这是一个静态分配的字符串。我们将在后面讨论更多不同的分配方式。我们把这个字符串作为参数传递给`println!`，而它负责在屏幕上打印字符串。就这么简单！

最后，这一行以一个分号结尾（;）。Rust是一门面向表达式（expression oriented）的语言，也就是说大部分语句都是表达式。分号用来表示一个表达式的结束，另一个新表达式的开始。大部分Rust代码行以分号结尾。我们将在后面更深层次地讨论这个问题。

最后，让我们实际编译和运行我们的程序。我们可以使用编译器`rustc`，用我们的源文件名作为参数：

```
$ rustc main.rs
```

这跟`gcc`或`clang`类似，如果你有C或C++知识背景。Rust会输出一个可执行文件。你可以用`ls`查看它：

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

或者在Windows下：

```
$ dir
main.exe  main.rs
```

现在这里有两个文件：我们`.rs`后缀的源文件，和可执行文件（在Windows下为`main.exe`，其它平台是`main`）。

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

这将在终端上打印出`Hello, world!`。

对于来自像Ruby，Python或者JavaScript这样的动态类型语言的同学，可能不太习惯这样将编译和执行分开的行为。Rust是一门预先编译语言（*ahead-of-time compiled language*），这意味着你可以编译一个程序，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，他们需要先安装Ruby/Python/JavaScript，不过你只需要一句命令就可以编译和执行你的程序。这一切都是语言设计的权衡取舍，而Rust已经做出了它的选择。

祝贺你！你已经正式完成了一个Rust程序。你现在是一名Rust程序员了！欢迎入坑←_←

接下来，我将向你介绍另外一个工具，Cargo，它用来编写真实世界的Rust程序。仅仅使用`rustc`可以很好应对简单的程序，但是当你的项目不断增长，你将会需要一个工具帮助你管理所有可能的选项，帮助你更加容易的与别人和别的项目分享代码。

Hello, Cargo!

Cargo是一个用来帮助Rustacean们管理Rust项目的工具。和Rust一样Cargo仍处于Alpha阶段，正在开发中。不过，对于许多Rust项目来说它已经足够用了，并且我们假设这些Rust项目将会从一开始就使用Cargo。

Cargo管理3个方面：构建你的代码，下载你代码所需的依赖和构建这些依赖。最开始，你的项目没有任何依赖，所以我们只使用它的第一部分机能。最终，我们会添加更多依赖。因为我们从一开始就使用Cargo，这将有利于我们后面添加依赖。

如果你通过官方安装器安装的Rust的话，你已经拥有了Cargo。如果你用的其它方式安装的Rust，你可能需要查看Cargo README获取特定的脚本安装Cargo。

转换到Cargo

让我们将Hello Wold项目转换到Cargo。

你需要做两件事来Cargo化我们的项目：创建一个`Cargo.toml`配置文件；将我们的源文件放到正确的地方。让我们先做移动文件那部分：

```
$ mkdir src
$ mv main.rs src/main.rs
```

注意因为我们创建了一个可执行文件，我们用了`main.rs`。如果我们想要创建一个库，我们应该使用`lib.rs`。入口点自定义的文件位置可以通过下面描述的TOML文件的[[[lib]]或[bin]]部分指定。

Cargo期望你的源文件位于`src`目录下。这样将项目根目录留给像README，license信息和其它与代码无关的文件。Cargo帮助我们保持项目干净整洁。一切各得其所。

接下来，我们的配置文件

```
$ editor Cargo.toml
```

请确保文件名正确：你需要一个大写的C！

在配置文件中添加：

```
[package]

name = "hello_world"
version = "0.0.1"
authors = [ "Your name " ]
```

配置文件使用TOML格式。让我们向你解释一下：

> TOML旨在作为一个最小化的配置文件格式，明确的语义，易于阅读。TOML被设计成可以无二义地映射到哈希表中。TOML应该能轻易解析成许多类型语言的数据类型。

TOML非常像INI文件，不过有一些其它优点。

一旦你设置好了配置文件，我们应该可以构建了！试试这些：

```
$ cargo build
   Compiling hello_world v0.0.1 (file:///home/yourname/projects/hello_world)
$ ./target/hello_world
Hello, world!
```

OK！我们运行`cargo build`构建了项目，并用`./target/debug/hello_world`运行了它。我们可以用`cargo run`一步到位：

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

注意这次我们并没有重新构建项目。Cargo注意到我们并未改变源文件，所以它只是运行了二进制文件。如果我们做了一个修改，我将会看到两个都做了：

```
$ cargo run
   Compiling hello_world v0.0.1 (file:///home/yourname/projects/hello_world)
     Running `target/debug/hello_world`
Hello, world!
```

这并没有比简单的使用`rustc`给我们带来了多少方便，不过想象一下未来：当我们的项目变得更复杂，我们将需要更多的工作来让所有的部分能够编译。通过Cargo，随着我们的项目增长，你可以只是`cargo build`，这样它就能正常工作。

当你的项目最终准备好发布了，我们可以使用`cargo build --release`来开启优化并编译你的项目。

你还需要注意到Cargo创建了一个新文件：`Cargo.lock`。

```
[root]
name = "hello_world"
version = "0.0.1"
```

这个文件被Cargo用来记录你程序中的依赖。现在，我们没有任何依赖，所以它的内容比较少。我们永远也不需要自己修改这个文件，让Cargo处理这些。

好了！我们成功利用Cargo构建了`hello_world`。虽然我们的项目很简单，但它用上许多实际的工具，你将会在余下的Rust生涯中一直使用。你可以期望这么做来开始几乎所有的Rust项目。

```
$ git clone someurl.com/foo
```

```
$ cd foo
$ cargo build
```

一个新项目

你不需要每次创建项目时都把这些操作整个做一遍。Cargo能够生成一个你可以直接进行开发的骨架项目目录。

用Cargo开始一个新项目，运行`cargo new`：

```
$ cargo new hello_world --bin
```

我们传递了一个`--bin`参数因为我们要构建一个二进制程序：如果你想创建一个库文件则不需要这个参数。

让我们看看Cargo为我们生成了什么：

```
$ cd hello_world
$ tree .
.
├── Cargo.toml
└── src
    └── main.rs

1 directory, 2 files
```

如果你没有`tree`命令，你应该能从你发行版的包管理软件中下载一个。这虽然不是必须的，但确实很有用。

这就是开始时所需的全部。首先，让我们查看下`Cargo.toml`：

```
[package]

name = "hello_world"
version = "0.0.1"
authors = ["Your Name "]
```

Cargo根据你传递的参数和git全局配置生成了合理的默认信息。你可能注意到了Cargo已经将`hello_world`目录初始化为了一个`git`仓库。

下面是`src/main.rs`的内容：

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

Cargo已经为我们生成了一个”Hello World!“，你可以进行coding了。你可以在这里获取一个更加深入的教程。

现在你学会了Cargo，让我们着手学习Rust语言吧。这是一些你使用Rust会一直用的到的基础。

（现在）你有两个选择：在“学习Rust”中深入一个项目，或者自底向上学习“语法和语义”。更有经验的系统程序员可能会喜欢“学习Rust”，而来自动态语言背景同学可能会喜欢另一个。不同的人有不同的学习方式！选择最适合你的。

学习Rust

欢迎！这一部分有一些教程通过构建项目来教你Rust。你会得到一个高层次的概念，不过我们会掠过细节。

如果你更喜欢一个“自底向上”风格的经验，查看语法和语义。

猜猜看

作为我们的第一个项目，我们来实现一个经典新手编程问题：猜猜看游戏。它是这么工作的：我们的程序将会随机生成一个1到100之间的随机数。它接着会提示我们猜一个数。当我们猜了一个数之后，它会告诉我们是大了还是小了。当我们猜对了，它会祝贺我们。听起来如何？

准备

我们准备一个新项目。进入到你的项目目录。还记得我们曾经创建我们`hello_world`的项目目录和`Cargo.toml`文件吗？Cargo有一个命令来为我们做这些。让我们试试：

```
$ cd ~/projects
$ cargo new guessing_game --bin
$ cd guessing_game
```

我们将项目名字传递给`cargo new`，然后用了`--bin`标记，因为我们要创建一个二进制文件，而不是一个库文件。

查看生成的`Cargo.toml`文件：

```
[package]

name = "guessing_game"
version = "0.0.1"
authors = ["Your Name "]
```

Cargo从环境变量中获取这些信息。如果这不对，赶紧修改它。

最后，Cargo为我们生成了应给“Hello, world!”。查看`src/main.rs`文件：

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

让我们编译Cargo为我们生成的项目：

```
$ cargo build
   Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
```

很好，再次打开你的`src/main.rs`文件。我们会将所有代码写在这个文件里。稍后我们会讲到多文件项目。

在我们继续之前，让我们再告诉你一个新的Cargo命令：`run`。`cargo run`跟`cargo build`类似，并且还会运行我们刚生成的可执行文件。试试它：

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

很好！`run`命令在我们需要快速重复运行一个项目是非常方便。我们的游戏就是这么一个项目，在我们添加新内容之前我们需要经常快速测试项目。

处理一次猜测

让我们开始吧！我们需要做的第一件事是让我们的玩家输入一个猜测。把这些放入你的`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)
        .ok()
        .expect("Failed to read line");

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

这有好多东西！让我们一点一点的过一遍。

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

我们需要获取用户输入，并接着打印结果作为输出。为此，我们需要标准库的`io`库。Rust为所有程序只导入了很少一些东西，'prelude'。如果它不在预先导入中，你将不得不直接`use`它。

```
fn main() {
```

就像你之前见过的，`main()`是你程序的入口点。`fn`语法声明了一个新函数，`()`表明这里没有参数，而`{`开始了函数体。因为我们不包含返回类型，它假设是`()`，一个空的元组。

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

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

我们之前学过`println!()`是一个在屏幕上打印字符串的宏。

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

现在我们遇到有意思的东西了！这一小行有很多内容。第一个我们需要注意到的是let语句，它用来创建“变量绑定”。它使用这个形式：

```
let foo = bar;
```

这回创建一个叫做 `foo` 的新绑定，并绑定它到 `bar` 这个值上。在很多语言中，这叫做一个“变量"，不过Rust的变量绑定暗藏玄机。

例如，它们默认是不可变的。这时为什么我们的例子使用了 `mut` ：它让一个绑定可变，而不是不可变。 `let` 并不从左手边获取一个名字，事实上他接受一个模式（pattern）。我们会在后面更多的使用模式。现在它使用起来非常简单：

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

噢，同时 `//` 会开始一个注释，直到这行的末尾。Rust忽略注释中的任何内容。

那么现在我们知道了 `let mut guess` 会引入一个叫做 `guess` 的可变绑定，不过我们不得不看看 `=` 的右侧，它绑定的内容： `String::new()` 。

`String` 是一个字符串类型，由标准库提供。String是一个可增长的，UTF-8编码的文本。

`::new()` 语法用了 `::` 因为它是一个特定类型的”关联函数“。这就是说，它与 `String` 自身关联，而不是与一个特定的 `String` 实例关联。一些语言管这叫一个”静态方法“。

这个函数叫做 `new()` ，因为它创建了一个新的，空的 `String` 。你会在很多类型上找到 `new()` 函数，因为它是创建一些类型新值的通常名称。

让我们继续：

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

这稍微有点多！让我们一点一点来。第一行有两部分。这是第一部分：

```
io::stdin()
```

还记得我们如何在程序的第一行 `use std::io` 的吗？现在我们调用了一个与之相关的函数。如果我们不 `use std::io` ，那么我们就得写成 `std::io::stdin()` 。

这个特殊的函数返回一个指向你终端标准输入的句柄。更具体的，可参考std::io::Stdin。

下一部分将用这个句柄去获取用户输入：

```
.read_line(&mut guess)
```

这里，我们对我们的句柄调用了read_line()方法。方法就像关联函数，不过只在一个类型的特定实例上可用，而不是这个类型本身。我们也向`read_line()`传递了一个参数：`&mut guess`。

还记得我们上面怎么绑定`guess`的吗？我们说它是可变的。然而，`read_line`并不接收`String`作为一个参数：它接收一个`&mut String`。Rust有一个叫做“引用”的功能，它允许你对一片数据有多个引用，用它可以减少拷贝。引用是一个复杂的功能，因为Rust的一个主要卖点就是它如何安全和便捷的使用引用。然而，目前我们还不需要知道很多细节来完成我们的程序。现在，所有我们需要了解的是像`let`绑定，引用默认是不可变的。因此，我们需要写成`&mut guess`，而不是`&guess`。

为什么`read_line()`会需要一个字符串的可变引用呢？它的工作是从标准输入获取用户输入，并把它放入一个字符串。所以它用字符串作为参数，为了可以增加输入，它必须是可变的。

不过我们还未完全看完这行代码。虽然它是单独的一行代码，它是只是这个单独逻辑代码行的开头部分：

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

当你用`.foo()`语法调用一个函数的时候，你可能会引入一个新行符或其它空白。这帮助我们拆分长的行。我们可以这么干：

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

不过这样会难以阅读。所以我们把它分开，3行对应3个方法调用。我们已经谈论过了`read_line()`，不过`ok()`和`expect()`呢？好吧，我们已经提到过`read_line()`将用户输入放入我们传递给它的`&mut String`中。不过它也返回一个值：在这个例子中，一个io::Result。Rust的标准库中有很多叫做`Result`的类型：一个泛型Result，然后是子库的特殊版本，例如`io::Result`。

这个`Result`类型的作用是编码错误处理信息。`Result`类型的值，像任何（其它）类型，有定义在其上的方法。在这个例子中，`io::Result`有一个`ok()`方法，它说“我们想假设这个值是一个成功的值。如果不是，就抛出错误信息”。为什么要抛出错误呢？好吧，对于一个基础的程序，我们只想打印出一个通用错误，因为基本上任何问题意味着我们不能继续。ok()方法返回一个值，有另一个方法定义在其上：`expect()`。expect()方法获取调用它的值，而且如果它不是一个成功的值，panic!并带有你传递给它的信息。这样的`panic!`会使我们的程序崩溃，显示（我们传递的）信息。

如果我们去掉这两个函数调用，我们的程序会编译通过，不过我们会得到一个警告：

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///home/you/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警告我们我们并未使用`Result`的值。这个警告来自`io::Result`的一个特殊注解。Rust尝试告诉你你并未处理一个可能的错误。阻止错误的正确方法是老实编写错误处理。幸运的是，如果我们只是想如果这有一个问题就崩溃的话，我们可以用这两个小方法。如果我们想从错误中恢复什么的，我们得做点别的，不过我们会把它留给接下来的项目。

这是我们第一个例子仅剩的一行：

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

这打印出我们保存输入的字符串。`{}`是一个占位符，所以我们传递`guess`作为一个参数。如果我们有多个`{}`，我们应该传递多个参数：

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

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

简单加愉快。

总而言之，这只是一个观光。我们可以用`cargo run`运行我们写的：

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/debug/guessing_game`
Guess the number!
Please input your guess.
6
You guessed: 6
```

好的！我们的第一部分完成了：我们可以从键盘获取输入，并把它打印回去。

生成一个秘密数字

接下来，我们要生成一个秘密数字。Rust标准库中还未包含一个随机数功能。然而，Rust团队确实提供了一个`rand` crate。一个“包装箱”（crate）是一个Rust代码的包。我们已经构建了一个”二进制包装箱“，它是一个可执行文件。`rand`是一个”库包装箱“，它包含被认为应该被其它程序使用的代码。

使用外部包装箱是Cargo的亮点。在我们使用`rand`编写代码之前，我们需要修改我们的`Cargo.toml`。打开它，并在末尾增加这几行：

```
[dependencies]

rand="0.3.0"
```

`Cargo.toml`的`[dependencies]`部分就像`[package]`部分：所有之后的东西都是它的一部分，直到下一个部分开始。Cargo使用依赖部分来知晓你用的外部包装箱的依赖，和你要求的版本。在这个例子中，我们用了`0.3.0`版本。Cargo理解语义化版本，它是一个编写版本号的标准。如果我们想要使用最新版本我们可以使用`*`或者我们可以使用一个范围的版本。Cargo文档包含更多细节。

现在，在不修改任何我们代码的情况下，让我们构建我们的项目：

```
$ cargo build
    Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading rand v0.3.8
 Downloading libc v0.1.6
   Compiling libc v0.1.6
   Compiling rand v0.3.8
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
```

（当然，你可能会看到不同的版本）

很多新的输出！现在我们有了一个外部依赖，Cargo从记录中获取了所有东西的最新版本，它们是来自Crates.io的一份拷贝。Crates.io是Rust生态系统中人们发表开源Rust项目供他人使用的地方。

在更新了记录后，Cargo检查我们的`[dependencies]`并下载任何我们还没有的东西。在这个例子中，虽然我们只说了我们要依赖`rand`，我们也获取了一份`libc`的拷贝。这是因为`rand`依赖`libc`工作。在下载了它们之后，它编译它们，然后接着编译我们的项目。

如果我们再次运行`cargo build`，我们会得到不同的输出：

```
$ cargo build
```

没错，没有输出！Cargo知道我们的项目被构建了，并且所有它的依赖也被构建了，所以没有理由再做一遍所有这一些。没有事情做，它简单的退出了。如果我们再打开`src/main.rs`，做一个无所谓的修改，然后接着再保存，我们就会看到一行：

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

所以，我们告诉Cargo我们需要任何`0.3.x`版本的`rand`，并且因此它获取在本文被编写时的最新版，`v0.3.8`。不过你瞧瞧当下一周，`v0.3.9`出来了，带有一个重要的bug修改吗？虽然bug修改很重要，不过如果`0.3.9`版本包含破坏我们代码的回归呢？

这个问题的回答是现在你会在你项目目录中找到的`Cargo.lock`。当你第一次构建你的项目的时候，Cargo查明所有符合你的要求的版本，并接着把它们写到了`Cargo.lock`文件里。当你在未来构建你的项目的时候，Cargo会注意到`Cargo.lock`的存在，并接着使用指定的版本而不是再次去做查明版本的所有工作。这让你有了一个可重复的自动构建。换句话说，我们会保持在`0.3.8`直到我们显式的升级，这对任何使用我们共享的代码的人同样有效，感谢锁文件。

当我们确实想要使用`v0.3.9`怎么办？Cargo有另一个命令，`update`，它代表“忽略锁，搞清楚所有我们指定的最新版本。如果这能工作，将这些版本写入锁文件”。不过，默认，Cargo只会寻找大于`0.3.0`小于`0.4.0`的版本。如果你想要移动到`0.4.x`，我们不得不直接更新`Cargo.toml`文件。当我们这么做，下一次我们`cargo build`，Cargo会更新索引并重新计算我们的`rand`要求。

关于 Cargo和它的生态系统有很多东西要说，不过眼下，这是我们需要知道的一切。Cargo让重用库变得真正的简单，并且Rustacean们可以编写更小的由很多子包组装成的项目。

让我们确实的使用`rand`。这是我们的下一步：

```
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)
        .ok()
        .expect("failed to read line");

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

我们做的第一件事是修改第一行。现在它是`extern crate rand`。因为我们在我们的`[dependencies]`声明了`rand`，我们可以用`extern crate`来让Rust知道我们正在使用它。这也等同于一个`use rand;`，所以我们可以通过`rand::`前缀使用`rand`包装箱中的一切。

下一步，我们增加了另一行`use`：`use rand::Rng`。我们一会将要使用一个方法，并且它要求`Rng`在作用域中才能工作。这个基本观点是：方法定义在一些叫做“特性（traits，也有译作特质）”的东西上面，而为了让方法能够工作，需要这个特性位于作用域中。关于更多细节，阅读特性部分。

这里还有两行我们增加的，在中间：

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

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

我们用`rand::thread_rng()`函数来获取一个随机数生成器的拷贝，它位于我们特定的执行线程的本地。因为我们`use rand::Rng`了，我们有一个`gen_range()`方法可用。这个函数获取两个参数，并产生一个位于其间的数字。它包含下限，不过不包含上限，所以我们需要`1`和`101`来生成一个`1`和`100`之间的数。

第二行仅仅打印出了秘密数字。这在我们开发我们的程序时很有用，所以我们可以简单的测试出来。不过在最终版本中我们会删除它。在开始就打印出结果就没什么可玩的了！

尝试运行我们的新程序几次：

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/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
```

好的！接下来：让我们比较我们的猜测和秘密数字。

比较猜测

现在我们得到了用户输入，让我们比较我们的猜测和随机值。这是我们的下一步，虽然它还不能正常工作：

```
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)
        .ok()
        .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!"),
    }
}
```

这有一些新东西。第一个是另一个`use`。我们带来了一个叫做`std::cmp::Ordering`类型到作用域中。接着，底部5行代码使用了它：

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

`cmp()`可以在任何能被比较的值上调用，并且它获取你想要比较的值的引用。它返回我们之前`use`的`Ordering`类型。我们使用一个match语句来决定具体是哪种`Ordering`。`Ordering`是一个枚举（enum），它看起来像这样：

```
enum Foo {
    Bar,
    Baz,
}
```

通过这个定义，任何`Foo`可以是`Foo::Bar`或者`Foo::Baz`。我们用`::`来表明一个特定`enum`变量的命名空间。

Ordering枚举有3个可能的变量：`Less`，`Equal`和`Greater`。`match`语句获取类型的值，并让你为每个可能的值创建一个“分支”。因为我们有3种类型的`Ordering`，我们有3个分支：

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

如果它是`Less`，我们打印`Too small!`，如果它是`Greater`，`Too big!`，而如果`Equal`，`You win!`。`match`真的非常有用，并且在Rust中经常使用。

我确实提到过我们还不能正常运行，虽然。让我们试试：

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
src/main.rs:28:21: 28:35 error: mismatched types:
 expected `&collections::string::String`,
    found `&_`
(expected struct `collections::string::String`,
    found integral variable) [E0308]
src/main.rs:28     match guess.cmp(&secret_number) {
                                   ^~~~~~~~~~~~~~
```

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

噢！这是一个大错误。它的核心是我们有“不匹配的类型”。Rust有一个强大的静态类型系统。然而，它也有类型推断。当我们写`let guess = String::new()`，Rust能够推断出`guess`应该是一个`String`，并因此不需要我们写出类型。而我们的`secret_number`，这有很多类型可以有从`1`到`100`的值：`i32`，一个32位数，或者`u32`，一个无符号的32位值，或者`i64`，一个64位值。或者其它什么的。目前为止，这并不重要，所以Rust默认为`i32`。然而，这里，Rust并不知道如何比较`guess`和`secret_number`。它们必须是相同的类型。最终，我们想要我们作为输入读到的`String`转换为一个真正的数字类型，来进行比较。我们可以用额外3行来搞定它。这是我们的新程序：

```
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)
        .ok()
        .expect("failed to read line");

    let guess: u32 = guess.trim().parse()
        .ok()
        .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!"),
    }
}
```

新的3行是：

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

稍等，我认为我们已经用过了一个`guess`？确实，不过Rust允许我们用新值“遮盖（shadow）”之前

的 `guess`。这在这种具体的情况中经常被用到，`guess` 开始是一个 `String`，不过我们想要把它转换为一个 `u32`。遮盖（Shadowing）让我们重用 `guess` 名字，而不是强迫我们想出两个独特的像 `guess_str` 和 `guess`，或者别的什么的。

我们绑定 `guess` 到一个看起来像我们之前写的表达式：

```
guess.trim().parse()
```

后跟一个 `ok().expect()` 调用。这里，`guess` 引用旧的 `guess`，那个我们输入用到的 `String`。`String` 的 `trim()` 方法会去掉我们字符串开头和结尾的任何空格。这很重要，因为我们不得不按“回车”键来满足 `read_line()`。这意味着如果我们输入 `5` 并按回车，`guess` 看起来像这样：`5\n`。`\n` 代表“新行”，回车键。`trim()` 去掉这些，保留 `5` 给我们的字符串。字符串的 `parse()` 方法将字符串解析为一些类型的数字。因为它可以解析多种数字，我们需要给Rust一些提醒作为我们具体想要的数字的类型。因此，`let guess: u32`。`guess` 后面的分号（`:`）告诉Rust我们要标注它的类型。`u32` 是一个无符号的，32位整型。Rust有一系列内建数字类型，不过我们选择了 `u32`。它是一个小的正数的好的默认选择。

就像 `read_line()`，我们调用 `parse()` 可能产生一个错误。如果我们的字符串包含 `A%?` 呢？并不能将它们转换成一个数字。为此，我们将做我们在 `read_line()` 时做的相同的事：使用 `ok()` 和 `expect()` 方法来在这里有错误时崩溃。

让我们尝试下我们的程序！

```
$ cargo run
   Compiling guessing_game v0.0.1 (file:///home/you/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!
```

很好！你可以看到我甚至在我的猜测前加上了空格，不过它仍然识别出我猜了76。运行这个程序几次，并检测猜测正确的值，和小的值。

现在我们让游戏大体上能玩了，不过我们只能猜一次。让我们增加循环来改变它！

循环

`loop` 关键字给我们一个无限循环。让我们加上它：

```
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)
            .ok()
            .expect("failed to read line");

        let guess: u32 = guess.trim().parse()
            .ok()
            .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!"),
        }
    }
}
```

并试试看。不过稍等，难道我们仅仅加上一个无限循环吗？是的。记得我们我们关于`parse()`的讨论吗？如果我们给出一个非数字回答，明显我们会`return`并退出：

```
$ cargo run
   Compiling guessing_game v0.0.1 (file:///home/you/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!'
```

啊哈！`quit`确实退出了。就像任何其它非数字输入。好吧，这至少不是最差的想法。首先，如果你赢得了

游戏，那我们就真的退出它：

```
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)
            .ok()
            .expect("failed to read line");

        let guess: u32 = guess.trim().parse()
            .ok()
            .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!` 后增加 `break`，我们将在你赢了后退出循环。退出循环也意味着退出程序，因为它是 `main()` 中最后的东西。我们仅仅需要再做一个小修改：当谁输入了一个非数字，我们并不想退出，我们就像忽略它。我们可以这么做：

```
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)
            .ok()
            .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;
            }
        }
    }
}
```

这是改变了的行：

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

这是你如何大体上从“错误就崩溃”移动到“确实处理错误”，通过从 `ok().expect()` 切换到一个 `match` 语句。 `parse()` 返回的 `Result` 就是一个像 `Ordering` 一样的枚举，不过在这个例子中，每个变量有一些数据与之相关： `Ok` 是一个成功，而 `Err` 是一个失败。每一个都包含更多信息：成功的解析为整型，或一个错误类型。在这个例子中，我们 `match` 为 `Ok(num)` ，它设置了 `Ok` 内叫做 `num` 的值，接着我们在右侧返回它。在 `Err` 的情况下，我们并不关心它是什么类型的错误，所以我们仅仅使用 `_` 而不是一个名字。这忽略错误，并 `continue` 造成我们进行 `loop` 的下一次迭代。

现在我们应该搞定了！让我们试试看：

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/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!
```

狂拽炫酷！通过一个最后的修改，我们就完成了我们的猜猜看游戏。你能想到它是什么吗？对了，我们并不想打印出秘密数字。它有利于测试，不过有点毁游（san）戏（guan）的味道。这是我们的最终源码：

```
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)
            .ok()
            .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;
            }
        }
    }
}
```

完成！

此刻，你成功的构建了猜猜看游戏！恭喜！

这第一个项目向你展示了很多：`let`，`match`，方法，关联函数，使用外部包装箱以及更多。我们下一个项目将会向你展示更多。

哲学家就餐问题

作为我们的第二个项目，让我们看看一个经典的并发问题。它叫做“进餐（ji）的哲学家”。它最初由Dijkstra于1965年（网上说1971年←_←）提出，不过我们会使用Tony Hoare写于1985年的这篇论文的版本

> 在远古时代，一个富有的慈善家捐赠了一个学院来为5名知名的哲学家提供住处。每个哲学家都有一个房间来进行他专业的思考活动；这也有一个共用的餐厅，布置了一个圆桌，周围放着5把椅子，每一把都标出了坐在这的哲学家的名字。哲学家们按逆时针顺序围绕桌子做下。每个哲学家的左手边放着一个金叉子，而在桌子中间有一大碗意大利面，它会不时的被补充。哲学家期望用他大部分的时间思考；不过当他饿了的时候，他走向餐厅，坐在它自己的椅子上，拿起他左手边自己的叉子，然后把它插进意大利面。不过乱成一团的意大利面需要第二把叉子才能吃到嘴里。因此哲学家不得不拿起他右手边的叉子。当他吃完了他会放下两把叉子，从椅子上起来，并继续思考。当然，一把叉子一次同时只能被一名哲学家使用。如果其他哲学家需要它，他必须等待直到叉子再次可用。

这个经典的问题展示了一些不同的并发元素。原因是事实上实现它需要一些技巧：一个简单的实现可能会死锁。例如，让我们考虑一个可能解决这个问题的简单算法：

1. 一个哲学家拿起左手边的叉子
2. 他接着拿起右手边的叉子
3. 他吃
4. 他返回叉子

现在，让我们想象一下事件的序列：

1. 哲学家1开始算法，拿起他左手边的叉子
2. 哲学家2开始算法，拿起他左手边的叉子
3. 哲学家3开始算法，拿起他左手边的叉子
4. 哲学家4开始算法，拿起他左手边的叉子
5. 哲学家5开始算法，拿起他左手边的叉子
6. 。。。？所有的叉子都被拿走了，不过没人在吃（意大利面）！

有不同方法可以解决这个问题。在教程中我们用我们自己的解决办法。现在，让我们自己来为问题建模。我将从哲学家开始：

```
struct Philosopher {
    name: String,
}

impl Philosopher {
    fn new(name: &str) -> Philosopher {
        Philosopher {
            name: name.to_string(),
        }
    }
}

fn main() {
    let p1 = Philosopher::new("Baruch Spinoza");
    let p2 = Philosopher::new("Gilles Deleuze");
```

```
    let p3 = Philosopher::new("Karl Marx");
    let p4 = Philosopher::new("Friedrich Nietzsche");
    let p5 = Philosopher::new("Michel Foucault");
}
```

这里，我们创建了一个struct来代表一个哲学家。目前，我们只需要一个名字。我们选择String类型作为名字，而不是`&str`。通常来说，处理一个拥有它自己数据的类型要比使用引用的数据来的简单。

让我们继续：

```
impl Philosopher {
    fn new(name: &str) -> Philosopher {
        Philosopher {
            name: name.to_string(),
        }
    }
}
```

`impl`块让我们在`Philosopher`上定义方法。在这个例子中，我们定义了一个叫做`new`的“关联函数”。第一行看起来像这样：

```
fn new(name: &str) -> Philosopher {
```

我们获取了一个参数，`name`，`&str`类型的。这是另一个字符串的引用。它返回了一个我们`Philosopher`结构体的实例。

```
Philosopher {
    name: name.to_string(),
}
```

这创建了一个新的`Philosopher`，并把它的`name`设置为我们的`name`参数。不仅仅是参数自身，虽然，因为我们在它上面调用了`.to_string()`。这将创建一个我们`&str`指向的字符串的拷贝，并给我们一个新的`String`，它是我们`Philosopher`的`name`字段的类型。

为什么不直接接受一个`String`呢？它更方便调用。如果我们获取一个`String`，而我们的调用者有一个`&str`，它就不得不自己调用这个方法。这个灵活性的缺点是我们总是生成了一个拷贝。对于我们这个小程序，这并不是特别的重要，因为我们知道我们只会用短小的字符串。

你要注意到的最后一件事：我们刚刚定义了一个`Philosopher`，不过好像并没有对它做什么。Rust是一个“基于表达式”的语言，它意味着Rust中几乎所有的东西都是一个表达式并返回一个值。这对函数也适用，最后的表达式是自动返回的。因为我们创建了一个新的`Philosopher`作为这个函数最后的表达式，我们最终返回了它。

这个名字，`new()`，在Rust中并没有什么特殊性。不过它是创建一个结构体新实例的函数的传统名称。在我们讨论为什么之前，让我们再看看`main()`：

```
fn main() {
    let p1 = Philosopher::new("Baruch Spinoza");
    let p2 = Philosopher::new("Gilles Deleuze");
    let p3 = Philosopher::new("Karl Marx");
    let p4 = Philosopher::new("Friedrich Nietzsche");
    let p5 = Philosopher::new("Michel Foucault");
}
```

这里，我们创建了5个新哲学家的变量绑定。这是我最崇拜的5个，不过你可以替换为任何你想要的。如果我们没有定义`new()`函数，它将看起来像这样：

```
fn main() {
    let p1 = Philosopher { name: "Baruch Spinoza".to_string() };
    let p2 = Philosopher { name: "Gilles Deleuze".to_string() };
    let p3 = Philosopher { name: "Karl Marx".to_string() };
    let p4 = Philosopher { name: "Friedrich Nietzche".to_string() };
    let p5 = Philosopher { name: "Michel Foucault".to_string() };
}
```

这看起来更乱。使用`new`还有别的优点，不过即便在这个简单的例子，它也被证明是易于使用的。

现在我们在这了解到了基础，这里有多种办法可以让我们可以处理更广泛的问题。首先我想从结尾开始：让我们准备让每个哲学家能吃完的方法。作为一个小步骤，让我们写一个方法，并接着循环遍历所有的哲学家，调用这个方法：

```
struct Philosopher {
    name: String,
}

impl Philosopher {
    fn new(name: &str) -> Philosopher {
        Philosopher {
            name: name.to_string(),
        }
    }

    fn eat(&self) {
        println!("{} is done eating.", self.name);
    }
}

fn main() {
    let philosophers = vec![
        Philosopher::new("Baruch Spinoza"),
        Philosopher::new("Gilles Deleuze"),
        Philosopher::new("Karl Marx"),
        Philosopher::new("Friedrich Nietzsche"),
        Philosopher::new("Michel Foucault"),
    ];

    for p in &philosophers {
        p.eat();
    }
```

```
}
```

让我们先看看`main()`。与其为我们的哲学家写5个独立的变量绑定，相反我们为它们创建了一个`Vec<T>`。`Vec<T>`也叫做一个“vector”，它是一个可增长的数组类型。接着我们用`for`循环遍历vector，顺序获取每个哲学家的引用。

在循环体中，我们调用`p.eat()`，它定义在上面：

```
fn eat(&self) {
    println!("{} is done eating.", self.name);
}
```

在Rust中，方法显式获取一个`self`参数。这就是为什么`eat()`是一个方法，而`new`是一个关联函数：`new()`没有用到`self`。在我们第一个版本的`eat()`，我们仅仅打印出哲学家的名字，并提到他们吃完了。运行这个程序应该会给你如下的输出：

```
Baruch Spinoza is done eating.
Gilles Deleuze is done eating.
Karl Marx is done eating.
Friedrich Nietzsche is done eating.
Michel Foucault is done eating.
```

十分简单的，他们都吃完了！然而我们还没有实际上实现真正的问题，所以我们还没完！

下一步，我们想要让我们的哲学家不光说吃完了，而是实际上的吃（意大利面）。这是下一个版本：

```
use std::thread;

struct Philosopher {
    name: String,
}

impl Philosopher {
    fn new(name: &str) -> Philosopher {
        Philosopher {
            name: name.to_string(),
        }
    }

    fn eat(&self) {
        println!("{} is eating.", self.name);

        thread::sleep_ms(1000);

        println!("{} is done eating.", self.name);
    }
}

fn main() {
    let philosophers = vec![
        Philosopher::new("Baruch Spinoza"),
```

```
        Philosopher::new("Gilles Deleuze"),
        Philosopher::new("Karl Marx"),
        Philosopher::new("Friedrich Nietzsche"),
        Philosopher::new("Michel Foucault"),
    ];

    for p in &philosophers {
        p.eat();
    }
}
```

只有一些变化，让我们拆开来看。

```
use std::thread;
```

`use`将名称引入作用域。我们将开始使用标准库的`thread`模块，所以我们需要`use`它。

```
    fn eat(&self) {
        println!("{} is eating.", self.name);

        thread::sleep_ms(1000);

        println!("{} is done eating.", self.name);
    }
```

现在我们打印出两个信息，有一个`sleep_ms()`在中间。这会模拟哲学家吃面的时间。

如果你运行这个程序，你应该会看到每个哲学家依次进餐：

```
Baruch Spinoza is eating.
Baruch Spinoza is done eating.
Gilles Deleuze is eating.
Gilles Deleuze is done eating.
Karl Marx is eating.
Karl Marx is done eating.
Friedrich Nietzsche is eating.
Friedrich Nietzsche is done eating.
Michel Foucault is eating.
Michel Foucault is done eating.
```

好极了！我们做到了。这仅有一个问题：我们实际上没有进行并发处理，而这才是我们问题的核心！

为了让哲学家并发的进餐。我们需要做一个小的修改。这是下一次迭代：

```
use std::thread;

struct Philosopher {
    name: String,
}
```

```
impl Philosopher {
    fn new(name: &str) -> Philosopher {
        Philosopher {
            name: name.to_string(),
        }
    }

    fn eat(&self) {
        println!("{} is eating.", self.name);

        thread::sleep_ms(1000);

        println!("{} is done eating.", self.name);
    }
}

fn main() {
    let philosophers = vec![
        Philosopher::new("Baruch Spinoza"),
        Philosopher::new("Gilles Deleuze"),
        Philosopher::new("Karl Marx"),
        Philosopher::new("Friedrich Nietzsche"),
        Philosopher::new("Michel Foucault"),
    ];

    let handles: Vec<_> = philosophers.into_iter().map(|p| {
        thread::spawn(move || {
            p.eat();
        })
    }).collect();

    for h in handles {
        h.join().unwrap();
    }
}
```

所有我们做的是改变了`main()`中的循环，并增加了第二个循环！这里是第一个变化：

```
let handles: Vec<_> = philosophers.into_iter().map(|p| {
    thread::spawn(move || {
        p.eat();
    })
}).collect();
```

虽然这只有5行，它们有4行密集的代码。让我们分开看。

```
let handles: Vec<_> =
```

我们引入了一个新的绑定，叫做`handles`。我们用这个名字因为我们将创建一些新的线程，并且它们会返回一些这些线程句柄来让我们控制它们的行为。然而这里我们需要显式注明类型，因为一个我们之后会介绍的问题。`_`是一个类型占位符。我们是在说“`handles`是一些东西的vector，不过Rust你自己应该能发现这些东西是什么"。

```
philosophers.into_iter().map(|p| {
```

我们获取了哲学家列表并在其上调用 `into_iter()`。它创建了一个迭代器来获取每个哲学家的所有权。我们需要这样做来把它们传递给我们的线程。我们取得这个迭代器并在其上调用 `map`，他会获取一个闭包作为参数并按顺序在每个元素上调用这个闭包。

```
    thread::spawn(move || {
        p.eat();
    })
```

这就是并发发生的地方。`thread::spawn` 获取一个闭包作为参数并在一个新线程执行这个闭包。这个闭包需要一个额外的标记，`move`，来表明这个闭包将会获取它获取的值的所有权。主要指 `map` 函数的 `p` 变量。

在线程中，所有我们做的就是在 `p` 上调用 `eat()`。

```
}).collect();
```

最后，我们获取所有这些 `map` 调用的结果并把它们收集起来。`collect()` 将会把它们放入一个某种类型的集合，这也就是为什么我们要表明返回值的类型：我们需要一个 `Vec<T>`。这些元素是 `thread::spawn` 调用的返回值，它们就是这些线程的句柄。噢！

```
for h in handles {
    h.join().unwrap();
}
```

在 `main()` 的结尾，我们遍历这些句柄并在其上调用 `join()`，它会阻塞执行直到线程完成执行。这保证了在程序结束之前这些线程都完成了它们的工作。

如果你运行这个程序，你将会看到哲学家们无序的进餐！我们有了多线程！

```
Gilles Deleuze is eating.
Gilles Deleuze is done eating.
Friedrich Nietzsche is eating.
Friedrich Nietzsche is done eating.
Michel Foucault is eating.
Baruch Spinoza is eating.
Baruch Spinoza is done eating.
Karl Marx is eating.
Karl Marx is done eating.
Michel Foucault is done eating.
```

不过叉子怎么办呢？我们还没有模型化它们呢。

为此，让我们创建一个新的 `struct`：

```
use std::sync::Mutex;

struct Table {
    forks: Vec<Mutex<()>>,
}
```

这个`Table`有一个`Mutex`的vector，一个互斥锁是一个控制并发的方法：一次只有一个线程能访问它的内容。这正是我们需要叉子拥有的属性。我们用了一个空元组，`()`，在互斥锁的内部，因为我们实际上并不准备使用这个值，只是要持有它。

让我们修改程序来使用`Table`：

```
use std::thread;
use std::sync::{Mutex, Arc};

struct Philosopher {
    name: String,
    left: usize,
    right: usize,
}

impl Philosopher {
    fn new(name: &str, left: usize, right: usize) -> Philosopher {
        Philosopher {
            name: name.to_string(),
            left: left,
            right: right,
        }
    }

    fn eat(&self, table: &Table) {
        let _left = table.forks[self.left].lock().unwrap();
        let _right = table.forks[self.right].lock().unwrap();

        println!("{} is eating.", self.name);

        thread::sleep_ms(1000);

        println!("{} is done eating.", self.name);
    }
}

struct Table {
    forks: Vec<Mutex<()>>,
}

fn main() {
    let table = Arc::new(Table { forks: vec![
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
    ]});

    let philosophers = vec![
```

```
        Philosopher::new("Baruch Spinoza", 0, 1),
        Philosopher::new("Gilles Deleuze", 1, 2),
        Philosopher::new("Karl Marx", 2, 3),
        Philosopher::new("Friedrich Nietzsche", 3, 4),
        Philosopher::new("Michel Foucault", 0, 4),
    ];

    let handles: Vec<_> = philosophers.into_iter().map(|p| {
        let table = table.clone();

        thread::spawn(move || {
            p.eat(&table);
        })
    }).collect();

    for h in handles {
        h.join().unwrap();
    }
}
```

大量的修改！然而，通过这次迭代，我们有了一个可以工作的程序。让我摸看看细节：

```
use std::sync::{Mutex, Arc};
```

我们将用到`std::sync`包中的另一个结构：`Arc<T>`。我们在用到时再详细解释。

```
struct Philosopher {
    name: String,
    left: usize,
    right: usize,
}
```

我们需要在我们的`Philosopher`中增加更多的字段。每个哲学家将拥有两把叉子：一个拿左手，一个拿右手。我们将用`usize`来表示它们，因为它是你的vector的索引的类型。这两个值将会是我们`Table`中的`forks`的索引。

```
fn new(name: &str, left: usize, right: usize) -> Philosopher {
    Philosopher {
        name: name.to_string(),
        left: left,
        right: right,
    }
}
```

现在我们需要构造这些`left`和`right`的值，所以我们把它们加到`new()`里。

```
fn eat(&self, table: &Table) {
    let _left = table.forks[self.left].lock().unwrap();
    let _right = table.forks[self.right].lock().unwrap();
```

```
        println!("{} is eating.", self.name);

        thread::sleep_ms(1000);

        println!("{} is done eating.", self.name);
    }
```

我们有两个新行。我们也增加了一个参数，`table`。我们访问`Table`的叉子列表，接着使用`self.left`和`self.right`来访问特定索引位置的叉子。这让我们访问索引位置的`Mutex`，并且我们在其上调用`lock()`。如果互斥锁目前正在被别人访问，我们将阻塞直到它可用为止。

`lock()`可能会失败，而且如果它失败了，我们想要程序崩溃。在这个例子中，互斥锁可能发生的错误是“被污染了（poisoned）”，它发生于当线程在持有锁的同时线程恐慌了。因为这不应该发生，所以我们仅仅是使用`unwrap()`。

这些代码还有另一个奇怪的事情：我们命名结果为`_left`和`_right`。为啥要用下划线？好吧，我们并不打算在锁中“使用”这些值。我们仅仅想要获取它。为此，Rust会警告我们从未使用这些值。通过使用下划线，我们告诉Rust这是我们意图做的，这样它就不会产生一个警告。

那怎么释放锁呢？好吧，这会在`_left`和`_right`离开作用域时发生，自动的。

```
    let table = Arc::new(Table { forks: vec![
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
    ]});
```

接下来，在`main()`中，我们创建了一个新`Table`并封装在一个`Arc<T>`中。“arc”代表“原子引用计数”，并且我们需要在多个线程间共享我们的`Table`。因为我们共享了它，引用计数会增长，而当每个线程结束，它会减少。

```
let philosophers = vec![
    Philosopher::new("Baruch Spinoza", 0, 1),
    Philosopher::new("Gilles Deleuze", 1, 2),
    Philosopher::new("Karl Marx", 2, 3),
    Philosopher::new("Friedrich Nietzsche", 3, 4),
    Philosopher::new("Michel Foucault", 0, 4),
];
```

我们需要传递我们的`left`和`right`的值给我们的`Philosopher`们的构造函数。不过这里有另一个细节，并且是“非常”重要。如果你观察它的模式，它们从头到尾全是连续的。米歇尔·福柯应该使用`4`，`0`作为参数，不过我们用了`0`，`4`。这事实上是为了避免死锁：我们的哲学家中有一个左撇子！这是解决这个问题的一个方法，并且在我看来，是最简单的方法。

```
let handles: Vec<_> = philosophers.into_iter().map(|p| {
    let table = table.clone();
```

```
    thread::spawn(move || {
        p.eat(&table);
    })
}).collect();
```

最后，在我们的`map()`/`collect()`循环中，我们调用了`table.clone()`。`Arc<T>`的`clone()`方法用来增加引用计数，而当它离开作用域，它减少计数。你会注意到这里我们可以引入一个新的`table`的绑定，而且它应该覆盖旧的一个。这经常用在你不想整出两个不同的名字的时候。

通过这些，我们的程序能工作了！任何同一时刻只有两名哲学家能进餐，因此你会得到像这样的输出：

```
Gilles Deleuze is eating.
Friedrich Nietzsche is eating.
Friedrich Nietzsche is done eating.
Gilles Deleuze is done eating.
Baruch Spinoza is eating.
Karl Marx is eating.
Baruch Spinoza is done eating.
Michel Foucault is eating.
Karl Marx is done eating.
Michel Foucault is done eating.
```

恭喜！你用Rust实现了一个经典的并发问题。

其他语言中的Rust

作为我们的第三个项目，我们决定选择一些可以展示Rust最强实力的东西：缺少实质的运行时。

当组织增长，他们越来越依赖大量的编程语言。不同的编程语言有不同的能力和弱点，而一个多语言栈让你在某个特定的编程语言的优点起作用的时候能使用它，当它有缺陷时使用其他编程语言。

一个非常常见的问题是很多编程语言在程序的运行时性能是很差的。通常来讲，使用一个更慢的，不过提供了更强大的程序员生产力的语言是一个值得的权衡的问题。为了帮助缓和这个问题，它们提供了一个用C编写部分你的系统，然后接着用高级语言编写的代码调用C代码。这叫做一个“外部语言接口”，通常简写为“FFI”。

Rust在所有两个方向支持FFI：它可以简单的调用C代码，而且至关重要的，它也可以简单的在C中被调用。与Rust缺乏垃圾回收和底层运行时要求相结合，这让Rust成为一个当你需要嵌入其他语言中以提供一些额外的活力时的强大的候选。

这有一个全面专注于FFI的章节和详情位于本书的其他位置。不过在这一章，我们会检查这个特定的FFI用例，通过3个例子，分别在Ruby，Python和JavaScript中。

问题

这里有很多不同的项目我们可以选择，不过我们将选择一个Rust相比其他语言有明确优势的例子：数值计算和线程。

很多语言，为了一致性，将数字放在堆上，而不是放在栈上。特别是在专注面向对象编程和使用垃圾回收的语言中，堆分配是默认行为。有时优化会栈分配特定的数字，不过与其依赖优化器做这个工作，我们可能想要确保我们总是使用原始类型而不是使用各种对象类型。

第二，很多语言有一个“全局解释器锁（GIL）”，它在很多情况下限制了并发。这在安全的名义下被使用，也有一定的积极影响，不过它限制了同时可以进行的工作的数量，这是一个很负面的影响。

为了强调这两方面，我们将创建一个大量使用这两方面的项目。因为这个例子关注的是将Rust嵌入到其他语言中，而不是问题自身，我们只使用一个玩具例子：

> 启动10个线程。在每个线程中，从1数到500万。在所有10个线程结束后，打印“done”。

我选择500万基于我特定的电脑。这里是一个例子的Ruby代码：

```
threads = []

10.times do
  threads << Thread.new do
    count = 0

    5_000_000.times do
      count += 1
    end
  end
```

```
end

threads.each {|t| t.join }
puts "done!"
```

尝试运行这个例子，并选择一个将运行几秒钟的数字。基于你电脑的硬件配置，你可能需要增大或减小这个数字。

在我的系统中，运行这个例子花费`2.156`秒。并且，如果我用一些进程监视工具，像`top`，我可以看到它只用了我的机器的一个核。这是GIL在起作用。

虽然这确实是一个虚构的程序，你可以想象许多问题与现实世界中的问题相似。为了我们的目标，启动一些繁忙的线程来代表一些并行的，昂贵的计算。

一个Rust库

让我们用Rust重写这个问题。首先，让我们用Cargo创建一个新项目：

```
$ cargo new embed
$ cd embed
```

这个程序在Rust中很好写：

```
use std::thread;

fn process() {
    let handles: Vec<_> = (0..10).map(|_| {
        thread::spawn(|| {
            let mut _x = 0;
            for _ in (0..5_000_001) {
                _x += 1
            }
        })
    }).collect();

    for h in handles {
        h.join().ok().expect("Could not join a thread!");
    }
}
```

一些代码可能与前面的例子类似。我们启动了10个线程，把它们收集到一个`handles`向量中。在每一个线程里，我们循环500万次，并每次给`_x`加一。为什么用下划线呢？好吧，如果我们去掉它并编译：

```
$ cargo build
   Compiling embed v0.1.0 (file:///home/steve/src/embed)
src/lib.rs:3:1: 16:2 warning: function is never used: `process`, #[warn(dead_code)] on by
 default
src/lib.rs:3 fn process() {
src/lib.rs:4     let handles: Vec<_> = (0..10).map(|_| {
src/lib.rs:5         thread::spawn(|| {
```

```
src/lib.rs:6             let mut x = 0;
src/lib.rs:7             for _ in (0..5_000_001) {
src/lib.rs:8                 x += 1
             ...
src/lib.rs:6:17: 6:22 warning: variable `x` is assigned to, but never used, #[warn(unused
_variables)] on by default
src/lib.rs:6             let mut x = 0;
                             ^~~~~
```

第一个警告是因为我们正在构建一个库。如果我们有一个函数的测试函数，就不会有警告了。不过目前它并不会被调用。

第二个与 `x` VS `_x` 相关。因为实际上我们从未对 `x` 进行任何处理，我们为此得到一个警告。在我们的情况中，这完全没有问题，因为我们只是想浪费CPU循环。对 `x` 使用下划线前缀将移除这个警告。

最后，我们同步每个线程。

然而现在，这是一个Rust库，而且它并没有暴露任何可以从C中调用的东西。如果现在我们尝试在别的语言中链接这个库，这并不能工作。我们只需做两个小的改变来修复这个问题，第一个是修改我们代码的开头：

```
#[no_mangle]
pub extern fn process() {
```

我们必须增加一个新的属性， `no_mangle` 。当你创建了一个Rust库，编译器会在输出中修改函数的名称。这么做的原因超出了本教程的范围，不过为了其他语言能够知道如何调用这些函数，我们需要禁止这么做。这个属性将它关闭。

另一个变化是 `pub extern` 。 `pub` 意味着这个函数应当从模块外被调用，而 `extern` 说它应当能被C调用。这就是了！没有更多的修改。

我们需要做的第二件事是修改我们的 `Cargo.toml` 的一个设定。在底部加上这些：

```
[lib]
name = "embed"
crate-type = ["dylib"]
```

这告诉Rust我们想要将我们的库编译为一个标准的动态库。默认，Rust编译为一个“rlib”，一个Rust特定的格式。

现在让我们构建这个项目：

```
$ cargo build --release
   Compiling embed v0.1.0 (file:///home/steve/src/embed)
```

我们选择了 `cargo build --release` ，它打开了优化进行构建。我们想让它越快越好！你可以在 `target/release` 找到输出的库：

```
$ ls target/release/
build  deps  examples  libembed.so  native
```

那个`libembed.so`就是我们的“共享目标（动态）”库。我们可以像任何用C写的动态库使用这个文件！另外，这也可能有`embed.dll`或`libembed.dylib`，基于不同的平台。

现在我们构建了我们的Rust库，让我们在Ruby中使用它。

Ruby

在我们的项目中打开一个`embed.rb`文件。并这么做：

```
require 'ffi'

module Hello
  extend FFI::Library
  ffi_lib 'target/release/libembed.so'
  attach_function :process, [], :void
end

Hello.process

puts "done!”
```

在我们可以运行之前，我们需要安装`ffi`gem：

```
$ gem install ffi # this may need sudo
Fetching: ffi-1.9.8.gem (100%)
Building native extensions.  This could take a while...
Successfully installed ffi-1.9.8
Parsing documentation for ffi-1.9.8
Installing ri documentation for ffi-1.9.8
Done installing documentation for ffi after 0 seconds
1 gem installed
```

最后，我们可以尝试运行它：

```
$ ruby embed.rb
done!
$
```

哇哦，这很快欸！在我系统中，它花费了`0.086`秒，而不是纯Ruby所需的2秒。让我们分析下我们的Ruby代码：

```
require 'ffi'
```

首先我们需要 `ffi` gem。这让我们可以像C库一样使用Rust的接口。

```
module Hello
  extend FFI::Library
  ffi_lib 'target/release/libembed.so'
```

`ffi` gem的作者建议使用一个模块来限定我们从共享库导入的函数的作用域。在其中，我们 `extend` 必要的 `FFI::Library` 模块，接着调用 `ffi_lib` 加载我们的动态库。我们仅仅传递我们库存储的路径，它是我们之前见过的，是 `target/release/libembed.so` 。

```
attach_function :process, [], :void
```

`attach_function` 方法由 `ffi` gem提供。它用来把我们Rust中 `process()` 连接到Ruby中同名函数。因为 `process()` 没有参数，第二个参数是一个空数组，并且因为它也没有返回值，我传递 `:void` 作为最后的参数。

```
Hello.process
```

这是实际的Rust调用。我们的 `module` 和 `attach_function` 调用的组合设置了环境。它看起来像一个Ruby函数，不过它实际是Rust！

```
puts "done!"
```

最后，作为我们每个项目的要求，我们打印 `done!` 。

这就是全部！就像我们看到的，连接两个语言真是很简单，并为我们带来了很多性提升。

接下来，让我们试试Python！

Python

在这个目录中创建一个 `embed.py` ，并写入这些：

```
from ctypes import cdll

lib = cdll.LoadLibrary("target/release/libembed.so")

lib.process()

print("done!")
```

甚至更简单了！我们使用 `ctypes` 模块的 `cdll` 。之后是一个快速的 `LoadLibrary` ，然后我可以调用 `process()` 。

在我的系统，这花费了`0.017`秒。非常快！

Node.js

Node并不是一个语言，不过目前它是服务端JavaScript居统治地位的实现。

为了在Node中进行FFI，首先我们需要安装这个库：

```
$ npm install ffi
```

安装之后，我们就可以使用它了：

```
var ffi = require('ffi');

var lib = ffi.Library('target/release/libembed', {
  'process': [ 'void', []  ]
});

lib.process();

console.log("done!");
```

这看起来比Python的例子更像Ruby的例子。我们使用`ffi`模块来获取`ffi.Library()`，它加载我们的动态库。我们需要标明函数的返回值和参数值，它们是返回“void”，和一个空数组表明没有参数。这样，我们就可以调用它并打印结果。

在我的系统上，这会花费`0.092`秒,很快。

结论

如你所见，基础操作是很简单的。当然，这里有很多我们可以做的。查看FFI章节以了解更多细节。

高效Rust

那么你已经学会了如何写一些Rust代码了。不过能写一些Rust代码和能写好Rust代码还是有区别的。

这个部分包含一些相对独立的教程，它们向你展示如何将你的Rust带入下一个等级。常见模式和标准库功能将被介绍。你可以选择任意顺序阅读这一部分。

栈和堆

作为一个系统语言，Rust在底层运作。如果你来自一个高级语言（背景），这可能有一些你不太熟悉的系统编程方面的内容。最重要的一个是内存如何工作，通过栈和堆。如果你熟悉类C语言如何使用栈分配，这个章节将是一个复习。如果你不太了解，你将会学到这个更通用的概念，不过是专注于Rust的。

内存管理

这里有两个术语是关于内存管理的。栈和堆是帮助你决定何时分配和释放内存的抽象（概念）。

这是一个高层次的比较：

栈非常快速，并且是Rust默认分配内存的地方。不过这个分配位于函数调用的本地，并有大小限制。堆，在另一方面，更慢，并且需要被你的程序显式分配。不过它无事实上的大小限制，并且是全局可访问的。

栈

让我们讨论下这个Rust程序：

```
fn main() {
    let x = 42;
}
```

这个程序有一个变量绑定，`x`。这个内存需要在什么地方被分配？Rust默认“栈分配”，也就意味着基本（类型）值“出现在栈上”。这意味着什么呢？

好吧，当函数被调用时，一些内存被分配给所有它的本地变量和一些其它信息。这叫做一个“栈帧（stack frame）”，而为了这个教程的目的，我们将忽略这些额外信息并仅仅考虑我们分配的局部变量。所以在这个例子中，当`main()`运行时，我们将为我们的栈帧分配一个单独的32位整型。如你所见，这会自动为你处理，我们并不必须写任何特殊的Rust代码或什么的。

当这个函数结束时，它的栈帧被释放。这也是自动发生的，在这里我们也不必要做任何特殊的事情。

这就是关于这个简单程序的一切。在这里你需要理解的关键是栈的分配非常快。因为我们知道所有的局部变量是预先分配的，我们可以一次获取所有的内存。并且因为我们也会同时把它们都扔了，我们可以快速的释放它们。

缺点是如果我们需要它们活过一个单独的函数调用，我们并不能保留它们的值。我们也还没有聊聊这个名字，“栈”意味着什么。为此，我们需要一个稍微更复杂一点的例子：

```
fn foo() {
    let y = 5;
    let z = 100;
}

fn main() {
```

```
    let x = 42;

    foo();
}
```

这个程序总共有3个变量：`foo()`中两个，`main()`中一个。就像之前一样，当`main()`被调用时，在它的栈帧中被分配了一个单独的整型。不过在我们展示当`foo()`被调用后发生了什么之前，我们需要想象一下内存中发生了什么。你的操作系统为你的程序提供了一个非常简单内存视图：一个巨大的地址列表。从`0`到一个很大的数字，代表你的电脑有多少内存。例如，如果你有1GB的内存，你的地址从`0`到`1,073,741,824`。这个数字来自2^{30}，1GB的字节数。

这个内存有点像一个巨型数组：地址从`0`开始一直增大到最终的数字。所以这是一个我们第一个栈帧的图表：

地址	名称	值
0	x	42

我们有位于地址`0`的`x`，它的值是`42`。

当`foo()`被调用，一个新的栈帧被分配：

地址	名称	值
2	z	100
1	y	5
0	x	42

因为`0`被第一个帧占有，`1`和`2`被用于`foo()`的栈帧。随着我们调用更多函数，它往上增长。

这有一些我们不得不注意的重要的内容。数字`0`，`1`和`2`都仅仅用于说明目的，并且与编译器会实际使用的具体数字没有关系。特别的，现实中这一系列的地址将会被一定数量的用于分隔地址的字节分隔开，并且这些分隔的字节可能甚至会超过被存储的值的大小。

在`foo()`结束后，它的帧被释放：

地址	名称	值
0	x	42

接着，在`main()`之后，就连最后一个值也木有了。简单明了！

它被叫做“栈”因为它就像一叠餐盘一样工作：最先放进去的盘子是最后一个你能取出来的。为此栈有时被叫做“后进，先出队列”，因为你放入栈的最后值是第一个你能取出来的值。

让我们试试第三个更深入的例子：

```
fn bar() {
    let i = 6;
}
```

```
fn foo() {
    let a = 5;
    let b = 100;
    let c = 1;

    bar();
}

fn main() {
    let x = 42;

    foo();
}
```

好的，第一步，我们调用`main()`：

地址	名称	值
0	x	42

接下来，`main()`调用`foo()`：

地址	名称	值
3	c	1
2	b	100
1	a	5
0	x	42

接着`foo()`调用`bar()`：

地址	名称	值
4	i	6
3	c	1
2	b	100
1	a	5
0	x	42

噢！我们的栈变得很高了。

在`bar()`结束后，它的帧被释放，只留下`foo()`和`main()`：

地址	名称	值
3	c	1
2	b	100
1	a	5
0	x	42

然后接着`foo()`结束，只剩下`main()`的了：

地址	名称	值
0	x	42

接下来我们完事了。找到了窍门了吗？这就像堆盘子：你在顶部增加，从顶部取走。

堆

目前为止，它能出色的工作，不过并非所有事情都能这么运作。有时，你需要在不同函数间传递一些内存，或者让它活过一次函数执行。为此，我们可以使用堆。

在Rust中，你可以使用`Box<T>`类型在堆上分配内存。这是一个例子：

```
fn main() {
    let x = Box::new(5);
    let y = 42;
}
```

这是当`main()`被调用时内存中发生了什么：

地址	名称	值
1	y	42
0	x	??????

我们在栈上分配了两个变量的空间。`y`是`42`，一如既往，不过`x`怎么样呢？好吧，`x`是一个`Box<i32>`，而装箱在堆上分配内存。装箱的实际值是一个带有指向“堆”指针的结构。当我们开始执行这个函数，然后`Box::new()`被调用，它在堆上分配了一些内存，并把`5`放在这。现在内存看起来像这样：

地址	名称	值
2^{30}		5
...	...	...
1	y	42
0	x	2^{30}

在我们假设的带有1GB内存（RAM）的电脑上我们有2^{30}个地址。并且因为我们的栈从`0`开始增长，分配内存的最简单的位置是内存的另一头。所以我们第一个值位于内存的最顶端。而在`x`的结构的值有一个裸指针指向我们在堆上分配的位置，所以`x`的值是2^{30}，我们请求的内存位置。

我们还没有过多的讨论在这个上下文中分配和释放内存具体意味着什么。深入非常底层的细节超出了这个教程的范围，不过需重要指出的是这里的堆不仅仅就是一个相反方向增长的栈。在本书的后面我们会有一个例子，不过因为堆可以以任意顺序分配和释放，它最终会产生“洞”。这是一个已经运行了一段时间的程序的内存图表：

地址	名称	值

2^{30}		5
$(2^{30}) - 1$		
$(2^{30}) - 2$		
$(2^{30}) - 3$		42
...	...	...
3	y	$(2^{30}) - 3$
2	y	42
1	y	42
0	x	2^{30}

在这个例子中，我们在堆上分配了4个东西，不过释放了它们中的两个。在2^{30}和$(2^{30}) - 3$之间有一个目前并没有被使用的断片（gap）。如何和为什么这会发生的具体细节依赖你用来管理堆的何种策略。不同的程序可以使用不同的“内存分配器”，它们是为你管理（内存）的库。Rust程序为此选择了 使用了jemalloc。

不管怎么说，回到我们的例子。因为这些内存在堆上，它可以比分配装箱的函数活的更久。然而在这个例子中，它并不如此。[移动]当函数结束，我们需要为`main()`释放栈帧。然而`Box<T>`有一些玄机：Drop。`Box`的`Drop`实现释放了当它创建时分配的内存。很好！所以当`x`消失时，它首先释放了分配在堆上的内存：

地址	名称	值
1	y	42
0	x	??????

[移动]：我们可以通过转移所有权来让内存活的更久，这有时叫做“走出盒子”。我们将在后面涉及更复杂的例子。

接着栈帧消失，释放所有的内存。

参数和借用

我们有了一些关于栈和堆运行的基础例子，不过函数参数和借用又怎么样呢？这是一个小的Rust程序：

```
fn foo(i: &i32) {
    let z = 42;
}

fn main() {
    let x = 5;
    let y = &x;

    foo(y);
}
```

当我们进入`main()`，内存看起来像这样：

地址	名称	值
1	y	0
0	x	5

`x`是一个普通的`5`，而`y`是一个指向`x`的引用。所以它的值是`x`的所在内存位置，它在这是`0`。

那么当我们调用`foo()`，传递`y`作为一个参数会怎么样呢？

地址	名称	值
3	z	42
2	i	0
1	y	0
0	x	5

栈帧不再仅仅是本地绑定了，它也有参数。所以在这里，我们需要有`i`参数，和`z`，我们本地的变量绑定。`i`是参数`y`的一个拷贝。因为`y`的值是`0`，`i`也是。

为什么要借用一个变量的一个原因是不需要分配任何内存：一个引用的值仅仅是一个内存位置的指针。如果我们溢出任何底层内存，事情就不能这么顺利工作了。

一个复杂的例子

好的，让我们一步一步过一遍这个复杂程序：

```
 fn foo(x: &i32) {
    let y = 10;
    let z = &y;

    baz(z);
    bar(x, z);
}

fn bar(a: &i32, b: &i32) {
    let c = 5;
    let d = Box::new(5);
    let e = &d;

    baz(e);
}

fn baz(f: &i32) {
    let g = 100;
}

fn main() {
    let h = 3;
    let i = Box::new(20);
    let j = &h;

    foo(j);
}
```

首先，我们调用`main()`：

地址	名称	值
2^{30}		20
...	...	...
2	j	0
1	i	2^{30}
0	h	3

我们为`j`，`i`和`h`分配内存。`i`在堆上，所以这里我们有一个指向它的值。

下一步，在`main()`的末尾，`foo()`被调用：

地址	名称	值
2^{30}		20
...	...	...
5	z	4
4	y	10
3	x	0
2	j	0
1	i	2^{30}
0	h	3

为`x`，`y`和`z`分配了空间。参数`x`和`j`有相同的值，因为这是我们传递给它的。它是一个指向`0`地址的指针，因为`j`指向`h`。

接着，`foo()`调用`baz()`，传递`z`：

地址	名称	值
2^{30}		20
...	...	...
7	g	100
6	f	4
5	z	4
4	y	10
3	x	0
2	j	0
1	i	2^{30}
0	h	3

我们为`f`和`g`分配了内存。`baz()`非常短，所以当它结束时，我们移除了它的栈帧：

地址	名称	值
2^{30}		20
...	...	...
5	z	4
4	y	10
3	x	0
2	j	0
1	i	2^{30}
0	h	3

接下来，`foo()`调用`bar()`并传递`x`和`z`：

地址	名称	值
2^{30}		20
$(2^{30}) - 1$		5
...	...	...
10	e	9
9	d	$(2^{30}) - 1$
8	c	5
7	b	4
6	a	0
5	z	4
4	y	10
3	x	0
2	j	0
1	i	2^{30}
0	h	3

我们最终在堆上分配了另一个值，所以我们必须从2^{30}减一。它比直接写`1,073,741,823`更简单。在任何情况下，我们通常用这个值。

在`bar()`的末尾，它调用了`baz()`：

地址	名称	值
2^{30}		20
$(2^{30}) - 1$		5
...	...	...
12	g	100

11	f	9
10	e	9
9	d	$(2^{30}) - 1$
8	c	5
7	b	4
6	a	0
5	z	4
4	y	10
3	x	0
2	j	0
1	i	2^{30}
0	h	3

这样，我们就到达最深的位置！噢！恭喜你一路跟了过来。

在`baz()`结束后，我们移除了`f`和`g`：

地址	名称	值
2^{30}		20
$(2^{30}) - 1$		5
...	...	...
10	e	9
9	d	$(2^{30}) - 1$
8	c	5
7	b	4
6	a	0
5	z	4
4	y	10
3	x	0
2	j	0
1	i	2^{30}
0	h	3

接下来，我们从`bar()`返回。在这里`d`是一个`Box<T>`，所以它也释放了它指向的内存空间：$(2^{30}) - 1$。

地址	名称	值
2^{30}		20
...	...	...
5	z	4
4	y	10

3	x	0
2	j	0
1	i	2^{30}
0	h	3

而在这之后，`foo()`返回：

地址	名称	值
2^{30}		20
...	...	...
2	j	0
1	i	2^{30}
0	h	3

接着，最后，`main()`，它清理了剩下的东西。当`i`被`Drop`时，它也会清理最后的堆空间。

其它语言怎么做？

大部分语言有一个默认堆分配的垃圾回收器。这意味着每个值都是装箱的。有很多原因为什么要这么做，不过这超出了这个教程的范畴。这也有一些优化会使得这些规则不是100%的时间为真。与其依赖栈和`Drop`来清理内存，垃圾回收器用处理堆来代替。

该用啥？（**Which to use?**）

那么如果栈是更快并更易于管理的，那么我们为啥还需要堆呢？一个大的原因是只有栈分配的话意味着你只有先进后出语义的获取存储的方法。堆分配严格的说更通用，允许以任意顺序从池中取出和返回存储，不过有复杂度开销。

一般来说，你应该倾向于栈分配，因此，Rust默认栈分配。栈的先进后出模型在基础层面上更简单。这有两个重大的影响：运行时效率和语义影响。

运行时效率

管理栈的内存是平凡的（trivial：来自C++的概念）：机器只是增加和减少一个单独的值，所谓的“栈指针”。管理堆的内存是不平凡的（non-trivial）：堆分配的内存在任意时刻被释放，而且每个堆分配内存的块可以是任意大小，内存管理器通常需要更多工作来识别出需要重用的内存。

如果你想深入这个主题的更多细节，这篇文章是一个很好的介绍。

语义影响（**Semantic impact**）

栈分配影响Rust语言自身，因此还有开发者的心智模型（mental model：想起苍蓝的握爪）。先进后出语义驱动了Rust语言如何处理自动内存管理。甚至是一个单独所有权堆分配的装箱的释放也可以被基于栈的先进后出语义驱动，就像本章中的讨论一样。非先进后出语义的灵活性（也就是说：表现力）意味着大体

上讲编译器不能在编译时自动推断出哪些内存应该被释放；它不得不依赖动态协议，可能来自于语言之外，来驱动释放（引用计数，就像是用`Rc<T>`和`Arc<T>`，是这个的一个例子）。

当考虑到极端情况，堆分配的增加的表现力带来了要么是显著的运行时支持（例如，以垃圾回收器的形式）要么是显著的程序猿努力（以要求进行Rust编译器并未提供的验证的显式的内存管理调用的形式）的开销。

测试

> 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）

让我们讨论一下如何测试Rust代码。在这里我们不会讨论什么是测试Rust代码的正确方法。这里有很多关于写测试好坏方法的流派。所有的这些途径都使用相同的基本工具，所以我们会向你展示他们的语法。

`test`属性（The test attribute）

简单的说，测试是一个标记为`test`属性的函数。让我们用Cargo来创建一个叫`adder`的项目：

```
$ cargo new adder
$ cd adder
```

在你创建一个新项目时Cargo会自动生成一个简单的测试。下面是`src/lib.rs`的内容：

```
#[test]
fn it_works() {
}
```

注意这个`#[test]`。这个属性表明这是一个测试函数。它现在没有函数体。它肯定能编译通过！让我们用`cargo test`运行测试：

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

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编译和运行了我们的测试。这里有两部分输出：一个是我们写的测试，另一个是文档测试。我们稍后

再讨论这些。现在，看看这行：

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

注意那个`it_works`。这是我们函数的名字：

```
fn it_works() {
```

然后我们有一个总结行：

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

那么为啥我们这个啥都没干的测试通过了呢？任何没有`panic!`的测试通过，`panic!`的测试失败。让我们的测试失败：

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

`assert!`是Rust提供的一个宏，它接受一个参数：如果参数是`true`，啥也不会发生。如果参数是`false`，它会`panic!`。让我们再次运行我们的测试：

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... FAILED

failures:

---- it_works stdout ----
        thread 'it_works' panicked at 'assertion failed: false', /home/steve/tmp/adder/sr
c/lib.rs:3



failures:
    it_works

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

thread '<main>' panicked at 'Some tests failed', /home/steve/src/rust/src/libtest/lib.rs:
247
```

Rust指出我们的测试失败了：

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

这反映在了总结行上：

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

我们也得到了一个非0的状态值：

```
$ echo $?
101
```

这在你想把`cargo test`集成进其它工具是非常有用。

我们可以使用另一个属性反转我们的失败的测试：`should_panic`：

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

现在即使我们`panic!`了测试也会通过，并且如果我们的测试通过了则会失败。让我试一下：

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

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
```

Rust提供了另一个宏，`assert_eq!`用来比较两个参数：

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

那个测试通过了吗？因为那个`should_panic`属性，它通过了：

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

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
```

`should_panic`测试是脆弱的，因为很难保证测试是否会因什么不可预测原因并未失败。为了解决这个问题，`should_panic`属性可以添加一个可选的`expected`参数。这个参数可以确保失败信息中包含我们提供的文字。下面是我们例子的一个更安全的版本：

```
#[test]
#[should_panic(expected = "assertion failed")]
fn it_works() {
    assert_eq!("Hello", "world");
}
```

这就是全部的基础内容！让我们写一个“真实”的测试：

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

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

`assert_eq!`是非常常见的；用已知的参数调用一些函数然后与期望的输出进行比较。

`tests`模块

然而以这样的方式来实现我们的测试的例子并不是地道的做法：它缺少`tests`模块。如果要实现我们的测试实例，一个比较惯用的做法应该是如下的：

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

#[cfg(test)]
mod tests {
```

```
    use super::add_two;

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

这里产生了一些变化。第一个变化是引入了一个`cfg`属性的`mod tests`。这个模块允许我们把所有测试集中到一起，并且需要的话还可以定义辅助函数，它们不会成为我们包装箱的一部分。`cfg`属性只会在我们尝试去运行测试时才会编译测试代码。这样可以节省编译时间，并且也确保我们的测试代码完全不会出现在我们的正式构建中。

第二个变化是`use`声明。因为我们在一个内部模块中，我们需要把我们要测试的函数导入到当前空间中。如果你有一个大型模块的话这会非常烦人，所以这里有经常使用一个`glob`功能。让我们修改我们的`src/lib.rs`来使用这个：

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

#[cfg(test)]
mod tests {
    use super::*;

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

注意`use`行的变化。现在运行我们的测试：

```
$ cargo test
    Updating registry `https://github.com/rust-lang/crates.io-index`
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
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
```

它能工作了！

目前的习惯是使用`test`模块来存放你的“单元测试”。任何只是测试一小部分功能的测试理应放在这里。那

么“集成测试”怎么办呢？我们有`tests`目录来处理这些。

`tests`目录

为了进行集成测试，让我们创建一个`tests`目录，然后放一个`tests/lib.rs`文件进去，输入如下内容：

```
extern crate adder;

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

这看起来与我们刚才的测试很像，不过有些许的不同。我们现在有一行`extern crate adder`在开头。这是因为在`tests`目录中的测试另一个完全不同的包装箱，所以我们需要导入我们的库。这也是为什么`tests`是一个写集成测试的好地方：它们就像其它程序一样使用我们的库。

让我们运行一下：

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

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

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

     Running target/lib-c18e7d3494509e74

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
```

现在我们有了三个部分：我们之前的两个测试，然后还有我们新添加的。

这就是`tests`目录的全部内容。它不需要`test`模块因为它整个就是关于测试的。

让我们最后看看第三部分：文档测试。

文档测试

没有什么是比带有例子的文档更好的了。当然也没有什么比不能工作的例子更糟的，因为文档完成之后代

码已经被改写。为此，Rust支持自动运行你文档中的例子。这是一个完整的有例子的`src/lib.rs`：

```
//! The `adder` crate provides functions that add numbers to other numbers.
//!
//! # Examples
//!
//! ` ` ` 因为gitbook排版问题，这里多写了两个空格
//! assert_eq!(4, adder::add_two(2));
//! ` ` ` 因为gitbook排版问题，这里多写了两个空格

/// This function adds two to its argument.
///
/// # Examples
///
/// ` ` ` 因为gitbook排版问题，这里多写了两个空格
/// use adder::add_two;
///
/// assert_eq!(4, add_two(2));
/// ` ` ` 因为gitbook排版问题，这里多写了两个空格
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

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

注意模块级的文档以`//!`开头然后函数级的文档以`///`开头。Rust文档在注释中支持Markdown语法，所以它支持3个反单引号代码块语法。想上面例子那样，加入一个`# Examples`部分被认为是一个惯例。

让我们再次运行测试：

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/steve/tmp/adder)
     Running target/adder-91b3e234d4ed382a

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

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

     Running target/lib-c18e7d3494509e74

running 1 test
test it_works ... ok

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

   Doc-tests adder
```

```
running 2 tests
test add_two_0 ... ok
test _0 ... ok

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

现在我们运行了3种测试！注意文档测试的名称：`_0`生成为模块测试，而`add_two_0`函数测试。如果你添加更多用例的话它们会像`add_two_1`这样自动加一。

条件编译

Rust有一个特殊的属性，`#[cfg]`，它允许你基于一个传递给编译器的标记编译代码。它有两种形式：

```
#[cfg(foo)]

#[cfg(bar = "baz")]
```

它还有一些帮助选项：

```
#[cfg(any(unix, windows))]

#[cfg(all(unix, target_pointer_width = "32"))]

#[cfg(not(foo))]
```

这些选项可以任意嵌套：

```
#[cfg(any(not(unix), all(target_os="macos", target_arch = "powerpc")))]
```

至于如何启用和禁用这些开关，如果你使用Cargo的话，它们可以在你`Cargo.toml`中的`[features]`部分设置：

```
[features]
# no features by default
default = []

# The “secure-password” feature depends on the bcrypt package.
secure-password = ["bcrypt"]
```

当你这么做的时候，Cargo传递给`rustc`一个标记：

```
--cfg feature="${feature_name}"
```

这些`cfg`标记集合会决定哪些功能被启用，并且因此，哪些代码会被编译。让我们看看这些代码：

```
#[cfg(feature = "foo")]
mod foo {
}
```

如果你用`cargo build --features "foo"`编译，他会向`rustc`传递`--cfg feature="foo"`标记，并且输出中将会包含`mod foo`。如果我们使用常规的`cargo build`编译，则不会传递额外的标记，因此，（输出）不会存在`foo`模块。

cfg_attr

你也可以通过一个基于`cfg`变量的`cfg_attr`来设置另一个属性：

```
#[cfg_attr(a, b)]
```

如果`a`通过`cfg`属性设置了的话这与`#[b]`相同，否则不起作用。

cfg!

`cfg!`语法扩展也让你可以在你的代码中使用这类标记：

```
if cfg!(target_os = "macos") || cfg!(target_os = "ios") {
    println!("Think Different!");
}
```

这会在编译时被替换为一个`true`或`false`，依配置设定而定。

文档

> 注意：由于gitbook的markdown对代码块中的3重音符的解析与github不一致（估计是bug），故在本章中在其间加空格，届时会有说明

文档是任何软件项目中重要的一部分，并且它在Rust中是一级重要的。让我们讨论下Rust提供给我们编写项目文档的的工具。

关于 `rustdoc`

Rust发行版中包含了一个工具，`rustdoc`，它可以生成文档。`rustdoc`也可以在Cargo中通过`cargo doc`。

文档可以使用两种方法生成：从源代码，或者从单独的Markdown文件。

文档化源代码

文档化Rust项目的主要方法是在源代码中添加注释。为了这个目标你可以这样使用文档注释：

```
/// Constructs a new `Rc<T>`.
///
/// # Examples
///
/// ` ` ` 实际不应有空格
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ` ` ` 实际不应有空格
pub fn new(value: T) -> Rc<T> {
    // implementation goes here
}
```

这段代码产生像这样的文档。我忽略了函数的实现，而是留下了一个标准的注释。第一个需要注意的地方是这个注释：它使用了`///`，而不是`//`。三斜线表明这是文档注释。

文档注释用Markdown语法编写。

Rust会记录这些注释，并在生成文档时使用它们。这在文档化像枚举这样的结构时很重要：

```
/// The `Option` type. See [the module level documentation](../) for more.
enum Option<T> {
    /// No value
    None,
    /// Some value `T`
    Some(T),
}
```

上面的代码可以工作，但这个不行：

```
/// The `Option` type. See [the module level documentation](../) for more.
enum Option<T> {
    None, /// No value
    Some(T), /// Some value `T`
}
```

你会得到一个错误：

```
hello.rs:4:1: 4:2 error: expected ident, found `}`
hello.rs:4 }
           ^
```

这个不幸的错误是有道理的：文档注释适用于它后面的内容，而在在最后的注释后面没有任何内容。

编写文档注释

不管怎样，让我们来详细了解一下注释的每一部分：

```
/// Constructs a new `Rc<T>`.
```

文档注释的第一行应该是它功能的一个简要总结。一句话。只包括基础。高层次。

```
///
/// Other details about constructing `Rc<T>`s, maybe describing complicated
/// semantics, maybe additional options, all kinds of stuff.
///
```

我们原始的例子只有一行总结，不过如果有更多东西要写，我们在一个新的段落增加更多解释。

特殊部分

```
/// # Examples
```

下面，是特殊部分。它由一个标头表明，`#`。有三种经常使用的标头。它们不是特殊的语法，只是传统，目前为止。

```
/// # Panics
```

不可恢复的函数滥用（比如，程序错误）在Rust中通常用恐慌表明，它至少会杀死整个当前的线程。如果你的函数有这样有意义的被识别为或者强制为恐慌的约定，记录文档是非常重要的。

```
/// # Failures
```

如果你的函数或方法返回`Result<T, E>`，那么描述何种情况下它会返回`Err(E)`是件好事。这并不如`Panics`重要，因为失败被编码进了类型系统，不过仍旧是件好事。

```
/// # Safety
```

如果你的函是`unsafe`的，你应该解释调用者应该支持哪种不可变量。

```
/// # Examples
///
/// ` ` ` 实际不应有空格
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ` ` ` 实际不应有空格
```

第三个，`Examples`。包含一个或多个使用你函数的例子，这样你的用户会为此感（ai）谢（shang）你的。这些例子写在代码块注释中，我们稍后会讨论到，并且可以有不止一个部分：

```
/// # Examples
///
/// Simple `&str` patterns:
///
/// ` ` ` 实际不应有空格
/// let v: Vec<&str> = "Mary had a little lamb".split(' ').collect();
/// assert_eq!(v, vec!["Mary", "had", "a", "little", "lamb"]);
/// ` ` ` 实际不应有空格
///
/// More complex patterns with a lambda:
///
/// ` ` ` 实际不应有空格
/// let v: Vec<&str> = "abc1def2ghi".split(|c: char| c.is_numeric()).collect();
/// assert_eq!(v, vec!["abc", "def", "ghi"]);
/// ` ` ` 实际不应有空格
```

让我们聊聊这些代码块的细节。

代码块注释

在注释中编写Rust代码，使用三重重音号：

```
/// ` ` ` 实际不应有空格
/// println!("Hello, world");
/// ` ` ` 实际不应有空格
```

如果你想要一些不是Rust的代码，你可以加上一个注释：

```
/// ` ` `c  实际不应有空格
/// printf("Hello, world\n");
/// ` ` ` 实际不应有空格
```

这回根据你选择的语言高亮代码。如果你只是想展示普通文本，选择`text`。

选择正确的注释是很重要的，因为`rustdoc`用一种有意思的方法是用它：它可以用来实际测试你的代码，这样你的注释就不会过时。如果你写了些C代码不过`rustdoc`会认为它是Rust代码由于你忽略了注释，`rustdoc`会在你生成文档时提示。

文档作为测试

让我们看看我的例子文档的样例：

```
/// ` ` ` 实际不应有空格
/// println!("Hello, world");
/// ` ` ` 实际不应有空格
```

你会注意到你并不需要`fn main()`或者别的什么函数。`rustdoc`会自动一个`main()`包装你的代码，并且在正确的位置。例如：

```
/// ` ` ` 实际不应有空格
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ` ` ` 实际不应有空格
```

这回作为测试：

```
fn main() {
    use std::rc::Rc;
    let five = Rc::new(5);
}
```

这里是`rustdoc`用来后处理例子的完整的算法：

1. 任何`#![foo]`开头的属性会被完整的作为包装箱属性
2. 一些通用的`allow`属性被插入，包括`unused_variables`，`unused_assignments`，`unused_mut`，`unused_attributes`和`dead_code`。小的例子经常触发这些lint检查
3. 如果例子并未包含`extern crate`，那么`extern crate <mycrate>;`被插入
4. 最后，如果例子不包含`fn main`，剩下的文本将被包装到`fn main() { your_code }`中

有时，这是不够的。例如，我们已经考虑到了所有`///`开头的代码样例了吗？普通文本：

```
/// Some documentation.
# fn foo() {}
```

与它的输出看起来有些不同：

```
/// Some documentation.
```

是的，你猜对了：你写的以`#`开头的行会在输出中被隐藏，不过会在编译你的代码时被使用。你可以利用这一点。在这个例子中，文档注释需要适用于一些函数，所以我只想向你展示文档注释，我需要在下面增加一些函数定义。同时，这只是用来满足编译器的，所以省略它会使得例子看起来更清楚。你可以使用这个技巧来详细的解释较长的例子，同时保留你文档的可测试行。例如，这些代码：

```
let x = 5;
let y = 6;
println!("{}", x + y);
```

这是一个被渲染出来的解释：

首先，我们把`x`设置为`5`：

```
let x = 5;
```

接着，我们把`y`设置为`6`：

```
let y = 6;
```

最后，我们打印`x`和`y`的和：

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

这是同样的解释的原始文本：

```
首先，我们把`x`设置为`5`：

    let x = 5;
    # let y = 6;
    # println!("{}", x + y);

接着，我们把`y`设置为`6`：

    # let x = 5;
    let y = 6;
    # println!("{}", x + y);

最后，我们打印`x`和`y`的和：
```

```
    # let x = 5;
    # let y = 6;
    println!("{}", x + y);
```

通过重复例子的所有部分，你可以确保你的例子仍能编译，同时只显示与你解释相关的部分。

文档化宏

下面是一个宏的文档例子：

```
/// Panic with a given message unless an expression evaluates to true.
///
/// # Examples
///
/// ` ` ` 实际不应有空格
/// # #[macro_use] extern crate foo;
/// # fn main() {
/// panic_unless!(1 + 1 == 2, “Math is broken.”);
/// # }
/// ` ` ` 实际不应有空格
///
/// ` ` `should_panic 实际不应有空格
/// # #[macro_use] extern crate foo;
/// # fn main() {
/// panic_unless!(true == false, “I’m broken.”);
/// # }
/// ` ` ` 实际不应有空格
#[macro_export]
macro_rules! panic_unless {
    ($condition:expr, $($rest:expr),+) => ({ if ! $condition { panic!($($rest),+); } });
}
```

你会注意到3个地方：我们需要添加我们自己的`extern crate`行，这样我们可以添加`#[macro_use]`属性。第二，我们也需要添加我们自己的`main()`。最后，用`#`机智的注释掉这两个代码，这样它们不会出现在输出中。

运行文档测试

要运行测试，要么

```
$ rustdoc --test path/to/my/crate/root.rs
# or（或者）
$ cargo test
```

对了，`cargo test`也会测试嵌入的文档。

这还有一些注释有利于帮助`rustdoc`在测试你的代码时正常工作：

```
/// ` ` `ignore 实际不应有空格
```

```
/// fn foo() {
/// ` ` ` 实际不应有空格
```

`ignore`指令告诉Rust忽略你的代码。这几乎不会是你想要的，因为这是最不受支持的。相反，考虑注释为`text`如果不是代码的话，或者使用`#`来形成一个只显示你关心部分的例子。

```
/// ` ` `should_panic 实际不应有空格
/// assert!(false);
/// ` ` `
```

`should_panic`告诉`rustdoc`这段代码应该正确编译，但是作为一个测试则不能通过。

```
/// ` ` `no_run 实际不应有空格
/// loop {
///     println!("Hello, world");
/// }
/// ` ` ` 实际不应有空格
```

`no_run`属性会编译你的代码，但是不运行它。这对像如“如何开始一个网络服务”这样的例子很重要，你会希望确保它能够编译，不过它可能会无限循环的执行！

文档化模块

Rust有另一种文档注释，`//!`。这种注释并不文档化接下来的内容，而是包围它的内容。换句话说：

```
mod foo {
    //! This is documentation for the `foo` module.
    //!
    //! # Examples

    // ...
}
```

这是你会看到`//!`最常见的用法：作为模块文档。如果你在`foo.rs`中有一个模块，打开它你常常会看到这些：

```
//! A module for using `foo`s.
//!
//! The `foo` module contains a lot of useful functionality blah blah blah
```

文档注释风格

查看RFC 505以了解文档风格和格式的惯例。

其它文档

所有这些行为都能在非Rust代码文件中工作。因为注释是用Markdown编写的，它们通常是`.md`文件。

当你在Markdown文件中写文档时，你并不需要加上注释前缀。例如：

```
/// # Examples
///
/// ` ` ` 实际不应有空格
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ` ` ` 实际不应有空格
```

就是

> ```
> # Examples
>
> use std::rc::Rc;
>
> let five = Rc::new(5);
> ```

当在一个Markdown文件中。不过这里有个窍门：Markdown文件需要有一个像这样的标题：

```
% The title

This is the example documentation.
```

`%`行需要放在文件的第一行。

`doc`属性

在更底层，文档注释是文档属性的语法糖：

```
/// this

#[doc="this"]
```

跟下面这个是相同的：

```
//! this

#![doc="/// this"]
```

写文档时你不会经常看见这些属性，不过当你要改变一些选项，或者写一个宏的时候比较有用。

重导出（Re-exports）

`rustdoc`会将公有部分的文档重导出：

```
extern crate foo;

pub use foo::bar;
```

这回在`foo`包装箱中生成文档，也会在你的包装箱中生成文档。它会在两个地方使用相同的内容。

这种行文可以通过`no_inline`来阻止：

```
extern crate foo;

#[doc(no_inline)]
pub use foo::bar;
```

控制HTML

你可以通过`#![doc]`属性控制`rustdoc`生成的THML文档的一些方面：

```
#![doc(html_logo_url = "http://www.rust-lang.org/logos/rust-logo-128x128-blk-v2.png",
       html_favicon_url = "http://www.rust-lang.org/favicon.ico",
       html_root_url = "http://doc.rust-lang.org/")];
```

这里设置了一些不同的选项，带有一个logo，一个收藏夹，和一个根URL。

通用选项

`rustdoc`也提供了一些其他命令行选项，以便进一步自定义：

- `--html-in-header FILE`：在`<head>...</head>`部分的末尾加上`FILE`内容
- `--html-before-content FILE`：在`<body>`之后，在渲染内容之前加上`FILE`内容
- `--html-after-content FILE`：在所有渲染内容之后加上`FILE`内容

注解安全

文档注释中的Markdown会被不加处理的放置于最终的网页中。注意HTML文本（XSS？）：

```
/// <script>alert(document.cookie)</script>
```

迭代器

让我们讨论一下循环。

还记得Rust的`for`循环吗？这是一个例子：

```
for x in 0..10 {
    println!("{}", x);
}
```

现在我们更加了解Rust了，我们可以谈谈这里的具体细节了。这个范围（`0..10`）是“迭代器”。我们可以重复调用迭代器的`.next()`方法，然后它会给我们一个数据序列。

就像这样：

```
let mut range = 0..10;

loop {
    match range.next() {
        Some(x) => {
            println!("{}", x);
        },
        None => { break }
    }
}
```

我们创建了一个`range`的可变绑定，它是我们的迭代器。我们接着`loop`，它包含一个`match`。`match`用来匹配`range.next()`的结果，它给我们迭代器的下一个值。`next`返回一个`Option<i32>`，在这个例子中，如果有值,它会返回`Some(i32)`然后当我们循环完毕,就会返回`None`。如果我们得到`Some(i32)`，我们就会打印它，如果我们得到`None`，我们`break`出循环。

这个代码例子基本上和我们的`loop`版本一样。`for`只是`loop/match/break`结构的简便写法。

然而，`for`循环并不是唯一使用迭代器的结构。编写你自己的迭代器涉及到实现`Iterator`特性。然而特性不是本章教程的涉及范围，不过Rust提供了一系列的有用的迭代器帮助我们完成各种任务。在我们开始讲解之前，我们需要看看一个Rust的反面模式。这就是如此使用范围。

是的，我们刚刚谈论到范围是多么的酷。不过范围也是非常原始的。例如，如果你想迭代一个向量的内容，你可能尝试这么写：

```
let nums = vec![1, 2, 3];

for i in 0..nums.len() {
    println!("{}", nums[i]);
}
```

这严格的说比使用现成的迭代器还要糟。你可以直接在向量上迭代。所以这么写：

```
let nums = vec![1, 2, 3];

for num in &nums {
    println!("{}", num);
}
```

这么写有两个原因。第一，它更明确的表明了我们的意图。我们迭代整个向量，而不是先迭代向量的索引，再按索引迭代向量。第二，这个版本也更有效率：第一个版本会进行额外的边界检查因为它使用了索引，`nums[i]`。因为我们利用迭代器获取每个向量元素的引用，第二个例子中并没有边界检查。这在迭代器中非常常见：我们可以忽略不必要的边界检查，不过仍然知道我们是安全的。

这里还有一个细节不是100%清楚的就是`println!`是如何工作的。`num`是`&i32`类型。也就是说，它是一个`i32`的引用，并不是`i32`本身。`println!`为我们处理了解引用，所以我们并没有看到它。下面的代码也能工作：

```
let nums = vec![1, 2, 3];

for num in &nums {
    println!("{}", *num);
}
```

现在我们显式的解引用了`num`。为什么`&nums`会给我们一个引用呢？首先，因为我们显式的使用了`&`。再次，如果它给我们数据，我们就是它的所有者了，这会涉及到生成数据的拷贝然后返回给我们拷贝。通过引用，我们只是借用了一个数据的引用，所以仅仅是传递了一个引用，并不涉及数据的移动。

那么，既然现在我们已经明确了范围通常不是我们需要的，让我们来讨论下你需要什么。

这里涉及到大体上相关的3类事物：迭代器，迭代适配器（*iterator adapters*）和消费者（*consumers*）。下面是一些定义：

- 迭代器给你一个值的序列
- 迭代适配器操作迭代器，产生一个不同输出序列的新迭代器
- 消费者操作迭代器，产生最终值的集合

让我们先看看消费者，因为我们已经见过范围这个迭代器了。

消费者（Consumers）

消费者操作一个迭代器，返回一些值或者几种类型的值。最常见的消费者是`collect()`。这个代码还不能编译，不过它表明了我们的意图：

```
let one_to_one_hundred = (1..101).collect();
```

如你所见，我们在迭代器上调用了`collect()`。`collect()`从迭代器中取得尽可能多的值，然后返回结果

的集合。那么为什么这不能编译呢？因为Rust不能确定你想收集什么类型的值，所以你需要让它知道。下面是一个可以编译的版本：

```
let one_to_one_hundred = (1..101).collect::<Vec<i32>>();
```

如果你还记得，`::<>`语法允许我们给出一个类型提示，所以我们可以告诉编译器我们需要一个整形的向量。但是你并不总是需要提供完整的类型。使用`_`可以让你提供一个部分的提示：

```
let one_to_one_hundred = (1..101).collect::<Vec<_>>();
```

这是指“请把值收集到`Vec<T>`，不过自行推断`T`类型”。为此`_`有时被称为“类型占位符”。

`collect()`是最常见的消费者，不过这还有其它的消费者。`find()`就是一个：

```
let greater_than_forty_two = (0..100)
                             .find(|x| *x > 42);

match greater_than_forty_two {
    Some(_) => println!("We got some numbers!"),
    None => println!("No numbers found :("),
}
```

`find`接收一个闭包，然后处理迭代器中每个元素的引用。如果这个元素是我们要找的，那么这个闭包返回`true`，如果不是就返回`false`。因为我们可能不能找到任何元素，所以`find`返回`Option`而不是元素本身。

另一个重要的消费者是`fold`。他看起来像这样：

```
let sum = (1..4).fold(0, |sum, x| sum + x);
```

`fold()`看起来像这样：`fold(base, |accumulator, element| ...)`。它需要两个参数：第一个参数叫做基数（*base*）。第二个是一个闭包，它自己也需要两个参数：第一个叫做累计数（*accumulator*），第二个叫元素（*element*）。每次迭代，这个闭包都会被调用，返回值是下一次迭代的累计数。在我们的第一次迭代，基数是累计数。

好吧，这有点混乱。让我们检查一下这个迭代器中所有这些值：

基数	累计数	元素	闭包结果
0	0	1	1
0	1	2	3
0	3	3	6

我们可以使用这些参数调用`fold()`：

```
.fold(0, |sum, x| sum + x);
```

那么，`0`是我们的基数，`sum`是累计数，`x`是元素。在第一次迭代，我们设置`sum`为`0`，然后`x`是`nums`的第一个元素，`1`。我们接着把`sum`和`x`相加，得到`0 + 1 = 1`。在我们第二次迭代，`sum`成为我们的累计值，元素是数组的第二个值，`2`，`1 + 2 = 3`，然后它就是最后一次迭代的累计数。在这次迭代中，`x`是最后的元素，`3`，那么`3 + 3 = 6`，就是我们和的最终值。`1 + 2 + 3 = 6`，这就是我们的结果。

（口哨）。最开始你见到`fold`的时候可能觉得有点奇怪，不过一旦你习惯了它，你就会在到处都用它。任何时候你有一个列表，然后你需要一个单一的结果，`fold`就是合适的。

消费者很重要还因为另一个我们没有讨论到的迭代器的属性：惰性。让我们更多的讨论一下迭代器，你就知道为什么消费者重要了。

迭代器（Iterators）

正如我们之前说的，迭代器是一个我们可以重复调用它的`.next()`方法，然后它会给我们一个数据序列的结构。因为你需要调用函数，这意味着迭代器是懒惰（*lazy*）的并且不需要预先生成所有的值。例如，下面的代码并没有真正的生成`1-100`这些数，而是创建了一个值来代表这个序列：

```
let nums = 1..100;
```

因为我们没有用范围做任何事，它并生成序列。让我们加上消费者：

```
let nums = (1..100).collect::<Vec<i32>>();
```

现在，`collect()`会要求范围生成一些值，接着它会开始产生序列。

范围是你会见到的两个基本迭代器之一。另一个是`iter()`。`iter()`可以把一个向量转换为一个简单的按顺序给出每个值的迭代器：

```
let nums = [1, 2, 3];

for num in nums.iter() {
   println!("{}", num);
}
```

这两个基本迭代器应该能胜任你的工作。这还有一些高级迭代器，包括一个是无限的。

足够关于迭代器的知识了。迭代适配器是关于迭代器最后一个要介绍的内容了。让我们开始吧！

迭代适配器（Iterator adapters）

迭代适配器（*Iterator adapters*）获取一个迭代器然后按某种方法修改它，并产生一个新的迭代器。最简单

的是一个是`map`：

```
(1..100).map(|x| x + 1);
```

在其他迭代器上调用`map`，然后产生一个新的迭代器，它的每个元素引用被调用了作为参数的闭包。所以它会给我们`2-100`这些数字。好吧，看起来是这样。如果你编译这个例子，你会得到一个警告：

```
warning: unused result which must be used: iterator adaptors are lazy and
         do nothing unless consumed, #[warn(unused_must_use)] on by default
(1..100).map(|x| x + 1);
 ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

又是惰性！那个闭包永远也不会执行。这个例子也不会打印任何数字：

```
(1..100).map(|x| println!("{}", x));
```

如果你尝试在一个迭代器上执行带有副作用的闭包，不如直接使用`for`。

这里有大量有趣的迭代适配器。`take(n)`会返回一个源迭代器下`n`个元素的新迭代器，注意这对源迭代器没有副作用。让我们试试我们之前的无限迭代器，`count()`：

```
for i in std::iter::count(1, 5).take(5) {
    println!("{}", i);
}
```

这会打印：

```
1
6
11
16
21
```

`filter()`是一个带有一个闭包参数的适配器。这个闭包返回`true`或`false`。`filter()`返回的新迭代器只包含闭包返回`true`的元素：

```
for i in (1..100).filter(|&x| x % 2 == 0) {
    println!("{}", i);
}
```

这会打印出1到100之间所有的偶数。（注意因为`filter`并不消费它迭代的元素，它传递每个元素的引用，所以过滤器使用`&x`来获取其中的整形数据。）

你可以链式的调用所有三种结构：以一个迭代器开始，适配几次，然后处理结果。看看下面的：

```
(1..1000)
    .filter(|&x| x % 2 == 0)
    .filter(|&x| x % 3 == 0)
    .take(5)
    .collect::<Vec<i32>>();
```

这会给你一个包含`6`，`12`，`18`，`24`和`30`的向量。

这只是一个迭代器，迭代适配器和消费者如何帮助你的小尝试。这里有很多非常实用的迭代器，当然你也可以编写你自己的迭代器。迭代器提供了一个安全，高效的处理所有类型列表的方法。最开始它们显得比较不寻常，不过如果你玩转了它们，你就会上瘾的。关于不同迭代器和消费者的列表，查看迭代器模块文档。

并发

并发与并行是计算机科学中相当重要的两个主题，并且在当今生产环境中也十分热门。计算机正拥有越来越多的核心，然而很多程序员还没有准备好去完全的利用它们。

Rust的内存安全功能也适用于并发环境。甚至并发的Rust程序也会是内存安全的，并且没有数据竞争。Rust的类型系统也能胜任，并且在编译时能提供你强大的方式去推论并发代码。

在我们讨论Rust提供的并发功能之前，理解一些问题是很重要的：Rust非常底层以至于所有这些都是由标准库，而不是由语言提供的。这意味着如果你在某些方面不喜欢Rust处理并发的方式，你可以自己实现一个。mio是关于这个原则实践的一个实际的例子。

背景：`Send`和`Sync`

并发难以推理。在Rust中，我们有一个强大，静态类型系统来帮助我们推理我们的代码。Rust自身提供了两个特性来帮助我们理解可能是并发的代码的意思。

`Send`

第一个我们要谈到的特性是Send。当一个`T`类型实现了`Send`，它向编译器表明这个类型的所有权可以在进程间安全的转移。

强制实施一些通用的限制是很重要的。例如，我们有一个连接两个线程的通道，我们想要能够向通道发送些数据到另一个线程。因此，我们要确保这个类型实现了`Send`。

相反的，如果我们通过FFI封装了一个不是线程安全的库，我们并不想实现`Send`，那么编译器会帮助我们强制确保它不会离开当前线程。

`Sync`

第二个特性是`Sync`。当一个类型`T`实现了`Sync`，它向编译器表明这个类型在多线程并发时没有导致内存不安全的可能性。

例如，使用一个原子引用来共享可变数据是线程安全的。Rust提供了一个这样的类型，`Arc<T>`，并且它实现了`Sync`，所以它可以安全的在线程间共享。

这两个特性允许你使用类型系统来确保你代码在并发环境的特性。在我们演示为什么之前，我们需要先学会如何创建一个并发Rust程序！

线程

Rust标准库提供了一个“线程”库，它允许你并行的执行Rust代码。这是一个使用`std::thread`的基本例子：

```
use std::thread;
```

```
fn main() {
    thread::spawn(|| {
        println!("Hello from a thread!");
    });
}
```

`thread::spawn()`方法接受一个闭包，它将会在一个新线程中执行。它返回一线程的句柄，这个句柄可以用来等待子线程结束并提取它的结果：

```
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        "Hello from a thread!"
    });

    println!("{}", handle.join().unwrap());
}
```

很多语言有执行多线程的能力，不过是很不安全的。这里有完整的书籍关于如何避免在共享可变状态下出现错误。Rust也用它的类型系统帮助我们，通过在编译时避免数据竞争。让我们具体讨论下如何在线程间共享数据。

安全的共享可变状态（**Safe Shared Mutable State**）

根据Rust的类型系统，我们有个听起来类似谎言的概念叫做：“安全的共享可变状态”。很多程序员都同意共享可变状态是非常，非常不好的。

有人曾说道：

> 共享可变状态是一切罪恶的根源。大部分语言尝试解决这个问题的“可变”部分，而Rust则尝试解决“共享”部分。

同样所有权系统也通过防止不当的使用指针来帮助我们排除数据竞争，最糟糕的并发bug之一。

作为一个例子，这是一个在很多语言中会产生数据竞争的Rust版本程序。它不能编译：

```
use std::thread;

fn main() {
    let mut data = vec![1u32, 2, 3];

    for i in 0..3 {
        thread::spawn(move || {
            data[i] += 1;
        });
    }

    thread::sleep_ms(50);
}
```

这会给我们一个错误：

```
8:17 error: capture of moved value: `data`
        data[i] += 1;
        ^~~~
```

在这个例子中：我们知道我们的代码应该是安全的，不过Rust并不确定。并且它确实不安全：如果每个线程中都有一个`data`的引用，并且这些线程获取了引用的所有权，我们就有了3个所有者！这是很糟糕的。我们可以用`Arc<T>`类型来修复它，它是一个原子引用计数的指针。“原子”部分是指它可以安全的跨线程共享。

`Arc<T>`假设它的内容有另一个属性来确保它可以跨线程共享：它假设它的内容实现了`Sync`。不过在我们的例子中，我们想要可以改变它的值。我们需要一个同一时间只有一个人可以修改它的值的类型。为此，我们可以使用`Mutex<T>`类型。下面是我们代码的第二版。它还是不能工作，不过是因为另外的原因：

```
use std::thread;
use std::sync::Mutex;

fn main() {
    let mut data = Mutex::new(vec![1u32, 2, 3]);

    for i in 0..3 {
        let data = data.lock().unwrap();
        thread::spawn(move || {
            data[i] += 1;
        });
    }

    thread::sleep_ms(50);
}
```

下面是错误：

```
<anon>:9:9: 9:22 error: the trait `core::marker::Send` is not implemented for the type `s
td::sync::mutex::MutexGuard<'_, collections::vec::Vec<u32>>` [E0277]
<anon>:11         thread::spawn(move || {
                  ^~~~~~~~~~~~~
<anon>:9:9: 9:22 note: `std::sync::mutex::MutexGuard<'_, collections::vec::Vec<u32>>` can
not be sent between threads safely
<anon>:11         thread::spawn(move || {
                  ^~~~~~~~~~~~~
```

你看，Mutex有一个有如下标记的lock方法：

```
fn lock(&self) -> LockResult<MutexGuard<T>>
```

因为`MutexGuard<T>`没有实现`Send`，我们不能传送守护穿过线程边界，它给了我们这个错误。

我们可以用`Arc<T>`修复这个问题。下面是一个可以工作的版本：

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let data = Arc::new(Mutex::new(vec![1u32, 2, 3]));

    for i in 0..3 {
        let data = data.clone();
        thread::spawn(move || {
            let mut data = data.lock().unwrap();
            data[i] += 1;
        });
    }

    thread::sleep_ms(50);
}
```

我们现在在`Arc`上调用`clone()`，它增加了其内部计数。这个句柄接着被移动到新线程。让我们更仔细的检查一个线程代码：

```
thread::spawn(move || {
    let mut data = data.lock().unwrap();
    data[i] += 1;
});
```

首先，我们调用`lock()`，它获取了互斥锁。因为这可能失败，它返回一个`Result<T, E>`，并且因为这仅仅是一个例子，我们`unwrap()`结果来获得一个数据的引用。现实代码在这里应该有更健壮的错误处理。下面我们可以随意修改它，因为我们持有锁。

最后，在线程运行的同时，我们等待在一个较短的定时器上。不过这并不理想：我们可能选择等待了一个合理的时间不过它更可能比所需的时间要久或并不足够长，这依赖程序运行时线程完成它的计算所需的时间。

一个比定时器更精确的替代是使用一个Rust标准库提供的用来同步各个线程的机制。让我们聊聊其中一个：通道。

通道（Channels）

下面是我们代码使用通道同步的版本，而不是等待特定时间：

```
use std::sync::{Arc, Mutex};
use std::thread;
use std::sync::mpsc;

fn main() {
    let data = Arc::new(Mutex::new(0u32));

    let (tx, rx) = mpsc::channel();
```

```
    for _ in 0..10 {
        let (data, tx) = (data.clone(), tx.clone());

        thread::spawn(move || {
            let mut data = data.lock().unwrap();
            *data += 1;

            tx.send(());
        });
    }

    for _ in 0..10 {
        rx.recv();
    }
}
```

我们使用`mpsc::channel()`方法创建了一个新的通道。我们仅仅向通道中`send`了一个简单的`()`，然后等待它们10个都返回。

因为这个通道只是发送了一个通用信号，我们也可以通过通道发送任何实现了`Send`的数据！

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    for _ in 0..10 {
        let tx = tx.clone();

        thread::spawn(move || {
            let answer = 42u32;

            tx.send(answer);
        });
    }

   rx.recv().ok().expect("Could not receive answer");
}
```

`u32`实现了`Send`因为我们可以拷贝它。所以我们创建了一个线程，让它计算结果，然后通过通道`send()`给我们结果。

恐慌（Panics）

`panic!`会使当前执行线程崩溃。你可以使用Rust的线程来作为一个简单的隔离机制：

```
use std::thread;

let result = thread::spawn(move || {
    panic!("oops!");
```

```
}).join();

assert!(result.is_err());
```

我们的`Thread`返回一个`Result`,它允许我们检查我们的线程是否发生了恐慌。

错误处理

> The best-laid plans of mice and men Often go awry
>
> "Tae a Moose", Robert Burns
>
> 不管是人是鼠,即使最如意的安排设计,结局也往往会出其不意
>
> 《致老鼠》，罗伯特·彭斯

有时，杯具就是发生了。有一个计划来应对不可避免会发生的问题是很重要的。Rust提供了丰富多样的支持来应对你程序中可能（让我们现实点：将会）出现的错误。

主要有两种类型的错误可能出现在你的软件中：失败和恐慌。让我们先看看它们的区别，接着讨论下如何处理他们。再接下来，我们讨论如何将失败升级为恐慌。

失败 vs. 恐慌

Rust使用了两个术语来区别这两种形式的错误：失败和恐慌。失败（*failure*）是一个可以通过某种方式恢复的错误。恐慌（*panic*）是不能够恢复的错误。

“恢复”又是什么意思呢？好吧，大部分情况，一个错误的可能性是可以预料的。例如，考虑一下`parse`函数：

```
"5".parse();
```

这个函数获取一个字符串参数然后把它转换为其它类型。不过因为它是一个字符串，你不能够确定这个转换是否能成功。例如，这个应该转换成什么呢？

```
"hello5world".parse();
```

这不能工作。所以我们知道这个函数只对一些输入能够正常工作。这是我们期望的行为。我们叫这类错误为失败。

另一方面，有时，会出现意料之外的错误，或者我们不能从中恢复。一个典型的例子是`assert!`：

```
assert!(x == 5);
```

我们用`assert!`声明某值为true。如果它不是true，很糟的事情发生了。严重到我们不能再当前状态下继续执行。另一个例子是使用`unreachable!()`宏：

```
enum Event {
    NewRelease,
}
```

```
fn probability(_: &Event) -> f64 {
    // real implementation would be more complex, of course
    0.95
}

fn descriptive_probability(event: Event) -> &'static str {
    match probability(&event) {
        1.00 => "certain",
        0.00 => "impossible",
        0.00 ... 0.25 => "very unlikely",
        0.25 ... 0.50 => "unlikely",
        0.50 ... 0.75 => "likely",
        0.75 ... 1.00 => "very likely",
    }
}

fn main() {
    std::io::println(descriptive_probability(NewRelease));
}
```

这会给我们一个错误：

```
error: non-exhaustive patterns: `_` not covered [E0004]
```

虽然我们知道我们覆盖了所有可能的分支，不过Rust不能确定。它不知道概率是在0.0和1.0之间的。所以我们加上另一个分支：

```
use Event::NewRelease;

enum Event {
    NewRelease,
}

fn probability(_: &Event) -> f64 {
    // real implementation would be more complex, of course
    0.95
}

fn descriptive_probability(event: Event) -> &'static str {
    match probability(&event) {
        1.00 => "certain",
        0.00 => "impossible",
        0.00 ... 0.25 => "very unlikely",
        0.25 ... 0.50 => "unlikely",
        0.50 ... 0.75 => "likely",
        0.75 ... 1.00 => "very likely",
        _ => unreachable!()
    }
}

fn main() {
    println!("{}", descriptive_probability(NewRelease));
}
```

我们永远也不应该触发`_`分支，所以我们使用`unreachable!()`宏来表明它。`unreachable!()`返回一个不同于`Result`的错误。Rust叫这类错误为恐慌。

使用`Option`和`Result`来处理错误

最简单的表明一个函数会失败的方法是使用`Option<T>`类型。例如，字符串的`find`方法尝试在字符串中找到一个模式，并返回一个`Option`：

```
let s = "foo";

assert_eq!(s.find('f'), Some(0));
assert_eq!(s.find('z'), None);
```

这对最简单的情况是合适的，不过在出错时并没有给出足够的信息。如果我们想知道“为什么”转换失败了呢？为此，我们可以使用`Result<T, E>`类型。它看起来像：

```
enum Result<T, E> {
   Ok(T),
   Err(E)
}
```

Rust自身提供了这个枚举，所以你不需要在你的代码中定义它。`Ok(T)`变体代表成功，`Err(E)`代表失败。在所有除了最普通的情况都推荐使用`Result`代替`Option`作为返回值。

这是一个使用`Result`的例子：

```
#[derive(Debug)]
enum Version { Version1, Version2 }

#[derive(Debug)]
enum ParseError { InvalidHeaderLength, InvalidVersion }

fn parse_version(header: &[u8]) -> Result<Version, ParseError> {
    if header.len() < 1 {
        return Err(ParseError::InvalidHeaderLength);
    }
    match header[0] {
        1 => Ok(Version::Version1),
        2 => Ok(Version::Version2),
        _ => Err(ParseError::InvalidVersion)
    }
}

let version = parse_version(&[1, 2, 3, 4]);
match version {
    Ok(v) => {
        println!("working with version: {:?}", v);
    }
    Err(e) => {
        println!("error parsing header: {:?}", e);
```

```
    }
}
```

这个例子使用了个枚举，`ParseError`,来列举各种可能出现的错误。

Debugtrait让我们可以使用`{:?}`格式运算符打印枚举值。

`panic!`和不可恢复错误

当一个错误是不可预料的和不可恢复的时候，`panic!`宏会引起一个恐慌。这会使当前线程崩溃，并给出一个错误：

```
panic!("boom");
```

给出：

```
thread '<main>' panicked at 'boom', hello.rs:2
```

当你运行它的时候。

因为这种情况相对稀少，保守的使用恐慌。

升级失败为恐慌

在特定的情况下，即使一个函数可能失败，我们也想把它当成恐慌。例如，`io::stdin().read_line(&mut buffer)`返回一个`Result<usize>`，当读取行出现错误时。这允许我们处理并尽可能从错误中恢复。

如果你不想处理这个错误，或者只是想终止程序，我们可以使用`unwrap()`方法：

```
io::stdin().read_line(&mut buffer).unwrap();
```

如果`Result`是`Err`的话`unwrap()`会`panic!`。这基本上就是说“给我一个值，然后如果出错了的话，直接崩溃。”这与匹配错误并尝试恢复相比更不稳定，不过它的处理明显更简洁。有时，直接崩溃就行。

这是另一个比`unwrap()`稍微聪明点的做法：

```
let mut buffer = String::new();
let input = io::stdin().read_line(&mut buffer)
                       .ok()
                       .expect("Failed to read line");
```

`ok()`将`Result`转换为`Option`，然后`expect()`做了和`unwrap()`同样的事，不过带有一个信息。这个信息会传递给底层的`panic!`，如果代码出错的话这样能提供更好的错误信息。

使用 `try!`

当编写调用那些返回 `Result` 的函数的代码时，错误处理会是烦人的。 `try!` 宏在调用栈上隐藏了一些衍生错误的样板。

它可以代替这些：

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

struct Info {
    name: String,
    age: i32,
    rating: i32,
}

fn write_info(info: &Info) -> io::Result<()> {
    let mut file = File::create("my_best_friends.txt").unwrap();

    if let Err(e) = writeln!(&mut file, "name: {}", info.name) {
        return Err(e)
    }
    if let Err(e) = writeln!(&mut file, "age: {}", info.age) {
        return Err(e)
    }
    if let Err(e) = writeln!(&mut file, "rating: {}", info.rating) {
        return Err(e)
    }

    return Ok(());
}
```

为下面这些代码:

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

struct Info {
    name: String,
    age: i32,
    rating: i32,
}

fn write_info(info: &Info) -> io::Result<()> {
    let mut file = try!(File::create("my_best_friends.txt"));

    try!(writeln!(&mut file, "name: {}", info.name));
    try!(writeln!(&mut file, "age: {}", info.age));
    try!(writeln!(&mut file, "rating: {}", info.rating));

    return Ok(());
}
```

在`try!`中封装一个表达式会返回一个未封装的正确（`Ok`）值，除非结果是`Err`，在这种情况下`Err`会从当前函数提早返回。

值得注意的是你只能在一个返回`Result`的函数中使用`try!`，这意味着你不能在`main()`中使用`try!`，因为`main()`不返回任何东西。

`try!`使用From特性来确定错误时应该返回什么。

外部函数接口

介绍

本教程会使用snappy压缩/解压缩库来作为一个Rust编写外部语言代码绑定的介绍。目前Rust还不能直接调用C++库，不过snappy库包含一个C接口（记录在snappy-c.h中）。

下面是一个最简单的调用其它语言函数的例子，如果你安装了snappy的话它将能够编译：

```
extern crate libc;
use libc::size_t;

#[link(name = "snappy")]
extern {
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
}

fn main() {
    let x = unsafe { snappy_max_compressed_length(100) };
    println!("max compressed length of a 100 byte buffer: {}", x);
}
```

`extern`块是一个外部库函数标记的列表，在这里例子中是C ABI。`#[link(...)]`属性用来指示链接器链接snappy库来解析符号。

外部函数被假定为不安全的所以调用它们需要包装在`unsafe {}`中，用来向编译器保证大括号中代码是安全的。C库经常提供不是线程安全的接口，并且几乎所有以指针作为参数的函数不是对所有输入时有效的，因为指针可以是垂悬的，而且裸指针超出了Rust安全内存模型的范围。

当声明外部语言的函数参数时，Rust编译器不能检查它是否正确，所以指定正确的类型是保证绑定运行时正常工作的一部分。

`extern`块可以扩展以包括整个snappy API：

```
extern crate libc;
use libc::{c_int, size_t};

#[link(name = "snappy")]
extern {
    fn snappy_compress(input: *const u8,
                       input_length: size_t,
                       compressed: *mut u8,
                       compressed_length: *mut size_t) -> c_int;
    fn snappy_uncompress(compressed: *const u8,
                         compressed_length: size_t,
                         uncompressed: *mut u8,
                         uncompressed_length: *mut size_t) -> c_int;
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
    fn snappy_uncompressed_length(compressed: *const u8,
                                  compressed_length: size_t,
```

```
                                    result: *mut size_t) -> c_int;
    fn snappy_validate_compressed_buffer(compressed: *const u8,
                                         compressed_length: size_t) -> c_int;
}
```

创建安全接口

原始C API需要需要封装才能提供内存安全性和利用像向量这样的高级内容。一个库可以选择只暴露出安全的，高级的接口并隐藏不安全的底层细节。

包装用到了缓冲区的函数涉及使用`slice::raw`模块来将Rust向量作为内存指针来操作。Rust的向量确保是一个连续的内存块。它的长度是当前包含的元素个数，而容量则是分配内存的大小。长度小于或等于容量。

```
pub fn validate_compressed_buffer(src: &[u8]) -> bool {
    unsafe {
        snappy_validate_compressed_buffer(src.as_ptr(), src.len() as size_t) == 0
    }
}
```

上面的`validate_compressed_buffer`封装使用了一个`unsafe`块，不过它通过从函数标记汇总去掉`unsafe`从而保证了对于所有输入调用都是安全的。

`snappy_compress`和`snappy_uncompress`函数更复杂，因为输出也使用了被分配的缓冲区。

`snappy_max_compressed_length`函数可以用来分配一个所需最大容量的向量来存放压缩的输出。接着这个向量可以作为一个输出参数传递给`snappy_compress`。另一个输出参数也被传递进去并设置了长度，可以用它来获取压缩后的真实长度。

```
pub fn compress(src: &[u8]) -> Vec<u8> {
    unsafe {
        let srclen = src.len() as size_t;
        let psrc = src.as_ptr();

        let mut dstlen = snappy_max_compressed_length(srclen);
        let mut dst = Vec::with_capacity(dstlen as usize);
        let pdst = dst.as_mut_ptr();

        snappy_compress(psrc, srclen, pdst, &mut dstlen);
        dst.set_len(dstlen as usize);
        dst
    }
}
```

解压是相似的，因为snappy储存了未压缩的大小作为压缩格式的一部分并且`snappy_uncompressed_length`可以取得所需缓冲区的实际大小。

```
pub fn uncompress(src: &[u8]) -> Option<Vec<u8>> {
    unsafe {
```

```
        let srclen = src.len() as size_t;
        let psrc = src.as_ptr();

        let mut dstlen: size_t = 0;
        snappy_uncompressed_length(psrc, srclen, &mut dstlen);

        let mut dst = Vec::with_capacity(dstlen as usize);
        let pdst = dst.as_mut_ptr();

        if snappy_uncompress(psrc, srclen, pdst, &mut dstlen) == 0 {
            dst.set_len(dstlen as usize);
            Some(dst)
        } else {
            None // SNAPPY_INVALID_INPUT
        }
    }
}
```

作为一个参考，我们在这里使用的例子可以在GitHub的这个库中找到。

析构函数

外部库经常把资源的所有权传递给调用函数。当这发生时，我们必须使用Rust析构函数累提供安全性和确保释放了这些资源（特别是在恐慌的时候）。

在Rust函数中处理C回调（Callbacks from C code to Rust functions）

一些外部库要求使用回调来向调用者反馈它们的当前状态或者即时数据。可以传递在Rust中定义的函数到外部库中。要求是这个回调函数被标记为`extern`并使用正确的调用约定来确保它可以在C代码中被调用。

接着回调函数可以通过一个C库的注册调用传递并在后面被执行。

一个基础的例子：

Rust代码：

```
extern fn callback(a: i32) {
    println!("I'm called from C with value {0}", a);
}

#[link(name = "extlib")]
extern {
   fn register_callback(cb: extern fn(i32)) -> i32;
   fn trigger_callback();
}

fn main() {
    unsafe {
        register_callback(callback);
        trigger_callback(); // Triggers the callback
    }
```

```
}
```

C代码：

```
typedef void (*rust_callback)(int32_t);
rust_callback cb;

int32_t register_callback(rust_callback callback) {
    cb = callback;
    return 1;
}

void trigger_callback() {
  cb(7); // Will call callback(7) in Rust
}
```

这个例子中Rust的`main()`会调用C中的`trigger_callback()`，它会反过来调用Rust中的`callback()`。

在Rust对象上使用回调（Targeting callbacks to Rust objects）

之前的例子展示了一个全局函数是如何在C代码中被调用的。然而我们经常希望回调是针对一个特殊Rust对象的。这个对象可能代表对应C语言中的封装。

这可以通过向C库传递这个对象的不安全指针来做到。C库则可以根据这个这个通知中的指针来取得Rust对象。这允许回调不安全的访问被引用的Rust对象。

Rust代码：

```
#[repr(C)]
struct RustObject {
    a: i32,
    // other members
}

extern "C" fn callback(target: *mut RustObject, a: i32) {
    println!("I'm called from C with value {0}", a);
    unsafe {
        // Update the value in RustObject with the value received from the callback
        (*target).a = a;
    }
}

#[link(name = "extlib")]
extern {
   fn register_callback(target: *mut RustObject,
                        cb: extern fn(*mut RustObject, i32)) -> i32;
   fn trigger_callback();
}

fn main() {
    // Create the object that will be referenced in the callback
    let mut rust_object = Box::new(RustObject { a: 5 });
```

```
    unsafe {
        register_callback(&mut *rust_object, callback);
        trigger_callback();
    }
}
```

C代码：

```
typedef void (*rust_callback)(void*, int32_t);
void* cb_target;
rust_callback cb;

int32_t register_callback(void* callback_target, rust_callback callback) {
    cb_target = callback_target;
    cb = callback;
    return 1;
}

void trigger_callback() {
  cb(cb_target, 7); // Will call callback(&rustObject, 7) in Rust
}
```

异步回调

在之前给出的例子中回调在一个外部C库的函数调用后直接就执行了。在回调的执行过程中当前线程控制权从Rust传到了C又传到了Rust，不过最终回调和和触发它的函数都在一个线程中执行。

当外部库生成了自己的线程并触发回调时情况就变得复杂了。在这种情况下回调中对Rust数据结构的访问时特别不安全的并必须有合适的同步机制。除了想互斥量这种经典同步机制外，另一种可能就是使用通道（在`std::comm`中）来从触发回调的C线程转发数据到Rust线程。

如果一个异步回调指定了一个在Rust地址空间的特殊Rust对象，那么在确保在对应Rust对象被销毁后不会再有回调被C库触发就格外重要了。这一点可以通过在对象的析构函数中注销回调和设计库使其确保在回调被注销后不会再被触发来取得。

链接

在`extern`上的`link`属性提供了基本的构建块来指示`rustc`如何连接到原生库。现在有两种被接受的链接属性形式：

- `#[link(name = "foo")]`
- `#[link(name = "foo", kind = "bar")]`

在这两种形式中，`foo`是我们链接的原生库的名字，而在第二个形式中`bar`是编译器要链接的原生库的类型。目前有3种已知的原生库类型：

- 动态 - `#[link(name = "readline")]`
- 静态 - `#[link(name = "my_build_dependency", kind = "static")]`

- 框架 - `#[link(name = "CoreFoundation", kind = "framework")]`

注意框架只支持OSX平台。

不同 `kind` 的值意味着链接过程中不同原生库的参与方式。从链接的角度看，rust编译器创建了两种组件：部分的（rlib/staticlib）和最终的（dylib/binary）。原生动态库和框架会从扩展到最终组件部分，而静态库则完全不会扩展。

一些关于这些模型如何使用的例子：

- 一个原生构建依赖。有时编写部分Rust代码时需要一些C/C++代码，另外使用发行为库格式的C/C++代码只是一个负担。在这种情况下，代码会被归档为 `libfoo.a` 然后rust包装箱可以通过 `#[link(name = "foo", kind = "static")]` 声明一个依赖。

  不管包装箱输出为何种形式，原生静态库将会包含在输出中，这意味着分配一个原生静态库是没有必要的。

- 一个正常动态库依赖。通用系统库（像 `readline` ）在大量系统上可用，通常你找不到这类库的静态拷贝。当这种依赖被添加到包装箱里时，部分目标（比如rlibs）将不会链接这些库，但是当rlib被包含进最终目标（比如二进制文件）时，原生库将被链接。

在OSX上，框架与动态库有相同的语义。

不安全块（Unsafe blocks）

一些操作，像解引用不安全的指针或者被标记为不安全的函数只允许在unsafe块中使用。unsafe块隔离的不安全性并向编译器保证不安全代码不会泄露到块之外。

不安全函数，另一方面，将它公布于众。一个不安全的函数这样写：

```
unsafe fn kaboom(ptr: *const int) -> int { *ptr }
```

这个函数只能被从 `unsafe` 块中或者 `unsafe` 函数调用。

访问外部全局变量（Accessing foreign globals）

外部API经常导出一个全局变量来进行像记录全局状态这样的工作。为了访问这些变量，你可以在 `extern` 块中用 `static` 关键字声明它们：

```
extern crate libc;

#[link(name = "readline")]
extern {
    static rl_readline_version: libc::c_int;
}

fn main() {
    println!("You have readline version {} installed.",
```

```
            rl_readline_version as int);
}
```

另外，你可能想修改外部结接口提供的全局状态。为了做到这一点，声明为`mut`这样我们就可以改变它了。

```
extern crate libc;

use std::ffi::CString;
use std::ptr;

#[link(name = "readline")]
extern {
    static mut rl_prompt: *const libc::c_char;
}

fn main() {
    let prompt = CString::new("[my-awesome-shell] $").unwrap();
    unsafe {
        rl_prompt = prompt.as_ptr();

        println!("{:?}", rl_prompt);

        rl_prompt = ptr::null();
    }
}
```

注意与`static mut`变量的所有交互都是不安全的，包括读或写。与全局可变量打交道需要足够的注意。

外部调用约定

大部分外部代码导出为一个C的ABI，并且Rust默认使用平台C的调用约定来调用外部函数。一些外部函数，尤其是大部分Windows API，使用其它的调用约定。Rust提供了一个告诉编译器应该用哪种调用约定的方法：

```
extern crate libc;

#[cfg(all(target_os = "win32", target_arch = "x86"))]
#[link(name = "kernel32")]
#[allow(non_snake_case)]
extern "stdcall" {
    fn SetEnvironmentVariableA(n: *const u8, v: *const u8) -> libc::c_int;
}
```

这适用于整个`extern`块。被支持的ABI约束的列表为：

- `stdcall`
- `aapcs`
- `cdecl`
- `fastcall`

- `Rust`
- `rust-intrinsic`
- `system`
- `C`
- `win64`

列表中大部分ABI都是自解释的，不过`system`ABI可能看起来有点奇怪。这个约束会选择任何能和目标库正确交互的ABI。例如，在x86构架的win32，这意味着会使用`stdcall`ABI。在x86_64上，然而，windows使用`C`调用约定，所以`C`会被使用。这意味在我们之前的例子中，我们可以使用`extern "system" { ... }`定义一个适用于所有windows系统的块，而不仅仅是x86系统。

外部代码交互性（Interoperability with foreign code）

只有当`#[repr(C)]`属性被用于结构体时Rust能确保`struct`的布局兼容平台的C的表现。`#[repr(C, packed)]`可以用来不对齐的排列结构体成员。`#[repr(C)]`也可以被用于一个枚举。

Rust拥有的装箱（`Box<T>`）使用非空指针作为指向他包含的对象的句柄。然而，它们不应该手动创建因为它们由内部分分配器托管。引用可以被安全的假设为直接指向数据的非空指针。然而，打破借用检查和可变性规则并不能保证安全，所以倾向于只在需要时使用裸指针（`*`）因为编译器不能为它们做更多假设。

向量和字符串共享同样基础的内存布局，`vec`和`str`模块中可用的功能可以操作C API。然而，字符串不是`\0`结尾的。如果你需要一个NUL结尾的字符串来与C交互，你需要使用`std::ffi`模块中的`CString`类型。

标准库中的`libc`模块包含类型别名和C标准库中的函数定义，Rust默认链接`libc`和`libm`。

“可空指针优化”（The "nullable pointer optimization"）

特定类型被定义为不为`nulll`。这包括引用（`&T`，`&mut T`），装箱（`Box<T>`），和函数指针（`extern "abi" fn()`）。当使用C接口时，可能为空的指针经常被使用。作为一个特殊的例子，一个泛化的`enum`包含两个变体，其中一个没有数据，而另一个包含一个单独的字段，非常适合“可空指针优化”。当这么一个枚举被用一个非空指针类型实例化时，它表现为一个指针，而无数据的变体表现为一个空指针。那么`Option<extern "C" fn(c_int) -> c_int>`可以用来表现一个C ABI中的可空函数指针。

在C中调用Rust代码

你可能会希望这么编译Rust代码以便可以在C中调用。这是很简单的，不过需要一些东西：

```
#[no_mangle]
pub extern fn hello_rust() -> *const u8 {
    "Hello, world!\0".as_ptr()
}
```

`extern`使这个函数遵循C调用约定，就像之前讨论[外部调用约定]https://doc.rust-lang.org/stable/book/ffi.html#foreign-calling-conventions)时一样。`no_mangle`属性关闭Rust的命名改

编，这样它更容易链接。

Borrow 和 AsRef

Borrow 和 AsRef 特性非常相似。这是一个快速的关于这两个特性意义的复习。

Borrow

Borrow 特性用于当你处于某种目的写了一个数据结构，并且你想要使用一个要么拥有要么借用的类型作为它的同义词。

例如， HashMap 有一个用了 Borrow 的 get 方法：

```
fn get<Q: ?Sized>(&self, k: &Q) -> Option<&V>
    where K: Borrow<Q>,
          Q: Hash + Eq
```

这个签名非常复杂。 k 参数是我们感兴趣的。它引用了一个 HashMap 自身的参数：

```
struct HashMap<K, V, S = RandomState> {
```

k 参数是 HashMap 用的 key 类型。所以，再一次查看 get() 的签名，我们可以在键实现了 Borrow<Q> 时使用 get() 。这样，我们可以创建一个 HashMap ，它使用 String 键，不过在我们搜索时使用 &str ：

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

let mut map = HashMap::new();
map.insert("Foo".to_string(), 42);

assert_eq!(map.get("Foo"), Some(&42));
```

这是因为标准库 为 String 实现了Borrow<str> 。

对于多数类型，当你想要获取一个自我拥有或借用的类型， &T 就足够了。不过一个地方 Borrow 是有效的是当这里有多于一种借用的值。片段就是一个这一点特别正确的地方：你可以有 &[T] 或者 &mut [T] 。如果我们想接受这两种类型， Borrow 就是你需要的：

```
use std::borrow::Borrow;
use std::fmt::Display;

fn foo<T: Borrow<i32> + Display>(a: T) {
    println!("a is borrowed: {}", a);
}

let mut i = 5;

foo(&i);
foo(&mut i);
```

这会打印出`a is borrowed: 5`两次。

AsRef

`AsRef`特性是一个转换特性。它用来在泛型中把一些值转换为引用。像这样：

```
let s = "Hello".to_string();

fn foo<T: AsRef<str>>(s: T) {
    let slice = s.as_ref();
}
```

我应该用哪个？

我们可以看到它们有些相似：它们都处理一些类型的自我拥有和借用版本。然而，它们还是有些不同。

选择`Borrow`当你想要抽象不同类型的借用，或者当你创建一个数据结构它把自我拥有和借用的值看作等同的，例如哈希和比较。

选择`AsRef`当你想要直接把一些值转换为引用，和当你在写泛型代码的时候。

发布途径

Rust项目使用一个叫做“发布途径”的概念来管理发布。理解这个选择你的项目应该使用哪个版本的Rust的过程是很重要的。

概览

Rust发布有3种途径：

- 开发版（Nightly）
- 测试版（Beta）
- 稳定版（Stable）

新的开发发布每天创建一次。每6个星期，最后的开发版被提升为“测试版”。在这时，它将只会收到修改重大错误的补丁。6个星期之后，测试版被提升为“稳定版”，而成为下一个版本的`1.x`发布。

这个过程并行发生。所以每6个星期，在同一天，开发变测试，测试变稳定。当`1.x`发布时的同时，`1.(x + 1)-beta`被发布，而开发版变为第一版的`1.(x + 2)-nightly`。

选择一个版本

通常来说，除非你有一个特定的原因，你应该使用稳定发布途径。这个发布意为用于普通用户。

然而，根据你对Rust的兴趣，你可能会选择使用开发构建。基本的权衡是：在开发途径，你可以使用不稳定的，新的Rust功能。然而，不稳定功能倾向于改变，所以任何新的开发版发布可能会破坏你的代码。如果你使用稳定发布，你不能使用实验功能，不过下一个Rust发布将不会因为破环性改变造成显著的问题。

通过持续集成（**CI**）改善生态系统

那么测试版怎么样呢？我们鼓励所有使用稳定发布途径的Rust用户在他们的持续集成系统中也针对测试途径进行测试。这会帮助警告团队以防出现一个意外的退步（regression）。

另外，针对开发版测试能够更快的捕获退步，因此如果你不介意一个第三种构建（环境），我们也会感激你针对开发版进行测试。

语法和语义

这一部分将Rust拆成小的部分，每一个对应一个概念。

如果你想自底向上的学习Rust，顺序阅读这一部分将会有很大帮助。

这些部分也组成了一个各个概念的参考，所以如果你阅读其它教程并发现一些迷惑的问题，你可以在这里找到一些解释。

变量绑定

你将学习的一个要点是变量绑定。它们看起来像这样：

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

在每个例子中都写上`fn main() {`有点冗长，所以之后我们将省略它。如果你是一路看过来的，确保你写了`main()`函数，而不是省略不写。否则，你将得到一个错误。

在许多语言中，这叫做变量。不过Rust的变量绑定有一些不同的巧妙之处。例如`let`表达式的左侧是一个“模式”，而不仅仅是一个变量。这意味着我们可以这样写：

```
let (x, y) = (1, 2);
```

在这个表达式被计算后，`x`将会是1，而`y`将会是2.模式非常强大，并且本书中有关于它的部分。我们现在还不需要这些功能，所以接下来你只需记住有这个东西就行了。

Rust是一个静态类型语言，这意味着我们需要先确定我们需要的类型。那为什么我们第一个例子能编译过呢？好的，Rust有一个叫做类型推断的功能。如果它能确认这是什么类型，Rust不需要你明确地指出来。

若你愿意，我们也可以加上类型。类型写在一个冒号（`:`）后面：

```
let x: i32 = 5;
```

如果我叫你对着全班同学大声读出这一行，你应该大喊“`x`被绑定为`i32类`型，它的值是`5`”。

在这个例子中我们选择`x`代表一个32位的有符号整数。Rust有许多不同的原生整数类型。以`i`开头的代表有符号整数而`u`开头的代表无符号整数。可能的整数大小是8，16，32和64位。

在之后的例子中，我们可能会在注释中注明变量类型。例子看起来像这样：

```
fn main() {
    let x = 5; // x: i32
}
```

注意注释和`let`表达式有类似的语法。理想的Rust代码中不应包含这类注释。不过我们偶尔会这么做来帮助你理解Rust推断的是什么类型。

绑定默认是不可变的（*immutable*）。下面的代码将不能编译：

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

它会给你如下错误：

```
error: re-assignment of immutable variable `x`
     x = 10;
     ^~~~~~
```

如果你想一个绑定是可变的，使用`mut`：

```
let mut x = 5; // mut x: i32
x = 10;
```

不止一个理由使得绑定默认不可变的，不过我们可以通过一个Rust的主要目标来理解它：安全。如果你没有使用`mut`，编译器会捕获它，让你知道你改变了一个你可能并不打算让它改变的值。如果绑定默认是可变的，编译器将不可能告诉你这些。如果你确实想变量可变，解决办法也非常简单：加个`mut`。

尽量避免可变状态有一些其它好处，不过这不在这个教程的讨论范围内。大体上，你总是可以避免显式可变量，并且这也是Rust希望你做的。即便如此，有时，可变量是你需要的，所以这并不是被禁止的。

让我们回到绑定。Rust变量绑定有另一个不同于其它语言的方面：绑定要求在可以使用它之前必须初始化。

让我们尝试一下。将你的`src/main.rs`修改为为如下：

```
fn main() {
    let x: i32;

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

你可以用`cargo build`命令去构建它。它依然会输出“Hello, world!”，不过你会得到一个警告：

```
   Compiling hello_world v0.0.1 (file:///home/you/projects/hello_world)
src/main.rs:2:9: 2:10 warning: unused variable: `x`, #[warn(unused_variable)] on by defau
lt
src/main.rs:2     let x: i32;
                      ^
```

Rust警告我们从未使用过这个变量绑定，但是因为我们从未用过它，无害不罚。然而，如果你确实想使用`x`，事情就不一样了。让我们试一下。修改代码如下：

```
fn main() {
    let x: i32;

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

然后尝试构建它。你会得到一个错误：

```
$ cargo build
   Compiling hello_world v0.0.1 (file:///home/you/projects/hello_world)
src/main.rs:4:39: 4:40 error: use of possibly uninitialized variable: `x`
src/main.rs:4     println!("The value of x is: {}", x);
                                                    ^
note: in expansion of format_args!
<std macros>:2:23: 2:77 note: expansion site
<std macros>:1:1: 3:2 note: in expansion of println!
src/main.rs:4:5: 4:42 note: expansion site
error: aborting due to previous error
Could not compile `hello_world`.
```

Rust是不会让我们使用一个没有经过初始化的值的。接下来，让我们讨论一下我们添加到`println!`中的内容。

如果你输出的字符串中包含一对大括号（`{}`，一些人称之为。。胡须（moustaches）？），Rust将把它解释为插入值的请求。字符串插值（*String interpolation*）是一个计算机科学术语，代表“在字符串中插入值”。我们加上一个逗号，然后是一个`x`，来表示我们想插入`x`的值。逗号用来分隔我们传递给函数和宏的参数，如果你想传递多个参数的话。

当你只写了大括号的时候，Rust会尝试检查值的类型来显示一个有意义的值。如果你想指定详细的语法，这里有很多选项可供选择。现在，让我们保持默认格式，整数并不难打印。

函数

到目前为止你应该见过一个函数，`main`函数：

```
fn main() {
}
```

这可能是最简单的函数声明。就像我们之前提到的，`fn`表示“这是一个函数”，后面跟着名字，一对括号因为这函数没有参数，然后是一对大括号代表函数体。下面是一个叫`foo`的函数：

```
fn foo() {
}
```

那么有参数是什么样的呢？下面这个函数打印一个数字：

```
fn print_number(x: i32) {
    println!("x is: {}", x);
}
```

下面是一个使用了`print_number`函数的完整的程序：

```
fn main() {
    print_number(5);
}

fn print_number(x: i32) {
    println!("x is: {}", x);
}
```

如你所见，函数参数与`let`声明非常相似：参数名加上冒号再加上参数类型。

下面是一个完整的程序，它将两个数相加并打印结果：

```
fn main() {
    print_sum(5, 6);
}

fn print_sum(x: i32, y: i32) {
    println!("sum is: {}", x + y);
}
```

在调用函数和声明函数时，你需要用逗号分隔多个参数。

与`let`不同，你必须为函数参数声明类型。下面代码将不能工作：

```
fn print_sum(x, y) {
    println!("sum is: {}", x + y);
}
```

你会获得如下错误：

```
expected one of `!`, `:`, or `@`, found `)`
fn print_number(x, y) {
```

这是一个有意为之的设计决定。即使像Haskell这样的能够全程序推断的语言，也经常建议注明类型是一个最佳实践。我们同意即使允许在在函数体中推断也要强制函数声明参数类型是一个全推断与无推断的最佳平衡。

如果我们要一个返回值呢？下面这个函数给一个整数加一：

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

Rust函数确实返回一个值，并且你需要在一个“箭头”后面声明类型，它是一个破折号（`-`）后跟一个大于号（`>`）。

注意这里并没有一个分号。如果你把它加上：

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

你将会得到一个错误：

```
error: not all control paths return a value
fn add_one(x: i32) -> i32 {
     x + 1;
}

help: consider removing this semicolon:
     x + 1;
          ^
```

这揭露了关于Rust两个有趣的地方：它是一个基于表达式的语言，并且分号与其它基于“大括号和分号”的语言不同。这两个方面是相关的。

表达式 VS 语句

Rust主要是一个基于表达式的语言。只有两种语句，其它的一切都是表达式。

然而这又有什么区别呢？表达式返回一个值，而语句不是。这就是为什么这里我们以“不是所有控制路径都返回一个值”结束：`x + 1;`语句不返回一个值。Rust中有两种类型的语句：“声明语句”和“表达式语句”。其余的一切是表达式。让我们先讨论下声明语句。

在一些语言中，变量绑定可以被写成一个表达式，不仅仅是语句。例如Ruby：

```
x = y = 5
```

在Rust中，然而，使用`let`引入一个绑定并不是一个表达式。下面的代码会产生一个编译时错误：

```
let x = (let y = 5); // expected identifier, found keyword `let`
```

编译器告诉我们这里它期望看到表达式的开头，而`let`只能开始一个语句，不是一个表达式。

注意赋值一个已经绑定过的变量（例如，`y = 5`）仍是一个表达式，即使它的（返回）值并不是特别有用。不像其它语言中赋值语句返回它赋的值（例如，前面例子中的`5`），在Rust中赋值的值是一个空的元组`()`：

```
let mut y = 5;

let x = (y = 6);  // x has the value `()`, not `6`
```

Rust中第二种语句是表达式语句。它的目的是把任何表达式变为语句。在实践环境中，Rust语法期望语句后跟其它语句。这意味着你用分号来分隔各个表达式。这意味着Rust看起来很像大部分其它要求你使用分号来做每一行的结尾的语言，并且你会看到分号出现在几乎每一行你看到的Rust代码。

那么我们说“几乎”的例外是神马呢？你已经见过它了，在这写代码中：

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

我们的函数声称它返回一个`i32`，不过带有一个分号，它会返回一个`()`。Rust意识到这可能不是我们想要的，并在我们之前看到的错误中建议我们去掉分号。

提早返回（**Early returns**）

不过提早返回怎么破？Rust确实有这么一个关键字，`return`：

```
fn foo(x: i32) -> i32 {
    return x;

    // we never run this code!
    x + 1
}
```

使用`return`作为函数的最后一行是可行的，不过被认为是一个糟糕的风格：

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

如果你之前没有使用过基于表达式的语言那么前面的没有`return`的定义可能看起来有点奇怪。不过它随着时间的推移它会变得直观。

发散函数（Diverging functions）

Rust有些特殊的语法叫“发散函数”，这些函数并不返回：

```
fn diverges() -> ! {
    panic!("This function never returns!");
}
```

`panic!`是一个宏，类似我们已经见过的`println!()`。与`println!()`不同的是，`panic!()`导致当前的执行线程崩溃并返回指定的信息。

因为这个函数会崩溃，所以它不会返回，所以它拥有一个类型`!`，它代表“发散”。一个发散函数可以是任何类型：

```
let x: i32 = diverges();
let x: String = diverges();
```

原生类型

Rust有一系列被认为是“原生”的类型。这意味着它们是内建在语言中的。Rust被构建为在标准库中也提供了一些建立在这些类型之上的有用的类型，不过它们也是原生的。

布尔型

Rust有一个内建的布尔类型，叫做`bool`。它有两个值，`true`和`false`：

```
let x = true;

let y: bool = false;
```

布尔型通常用在if语句中。

你可以在标准库文档中找到更多关于`bool`的文档。

`char`

`char`类型代表一个单独的Unicode字符的值。你可以用单引号（`'`）创建`char`：

```
let x = 'x';
let two_hearts = '💕';
```

不像其它语言，这意味着Rust的`char`并不是1个字节，而是4个。

你可以在标准库文档中找到更多关于`char`的文档。

数字类型

Rust有一些分类的大量数字类型：有符号和无符号，定长和变长，浮点和整型。

这些类型包含两部分：分类，和大小。例如，`u16`是一个拥有16位大小的无符号类型。更多字节让你拥有更大的数字。

如果一个数字常量没有推断它类型的条件，它采用默认类型：

```
let x = 42; // x has type i32

let y = 1.0; // y has type f64
```

这里有一个不同数字类型的列表，以及它们在标注库中的文档：

- i16

- i32
- i64
- i8
- u16
- u32
- u64
- u8
- isize
- usize
- f32
- f64

让我们按分类重温一遍：

有符号和无符号

整型有两种变体：有符号和无符号。为了理解它们的区别，让我们考虑一个4字节大小的数字。一个有符号，4字节数字你可以储存 `-8` 到 `+7` 的数字。有符号数采用“补码”表示。一个无符号4字节的数字，因为它不需要储存负数，可以出储存 `0` 到 `+15` 的数字。

固定大小类型

固定大小类型在其表现中有特定数量的位。有效的位大小是 `8`， `16`， `32` 和 `64`。那么， `u32` 是无符号的，32位整型，而 `i64` 是有符号，64位整型。

可变大小类型

Rust也提供了依赖底层机器指针大小的类型。这些类型拥有“size”分类，并有有符号和无符号变体。它有两个类型： `isize` 和 `usize`。

浮点类型

Rust也有两个浮点类型： `f32` 和 `f64`。它们对应IEEE-754单精度和双精度浮点数。

数组

像很多编程语言一样，Rust有用来表示数据序列的列表类型。最基本的是数组，一个定长相同类型的元素列表。数组默认是不可变的。

```
let a = [1, 2, 3]; // a: [i32; 3]
let mut m = [1, 2, 3]; // mut m: [i32; 3]
```

数组的类型是 `[T; N]`。我们会在泛型部分的时候讨论这个 `T` 标记。 `N` 是一个编译时常量，代表数组的长度。

这里有一个可以将数组中每一个元素初始化为相同值的简写。在这个例子中，`a`的每个元素都被初始化为`0`：

```
let a = [0; 20]; // a: [i32; 20]
```

你可以用`a.len()`来获取`a`中元素的数量，用`a.iter()`在循环中迭代所有元素。下面的代码会按顺序打印每一个元素：

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

println!("a has {} elements", a.len());
```

你可以用下标（*subscript notation*）来访问特定的元素：

```
let names = ["Graydon", "Brian", "Niko"]; // names: [&str; 3]

println!("The second name is: {}", names[1]);
```

就跟大部分编程语言一个样，下标从0开始，所以第一个元素是`names[0]`，第二个是`names[1]`。上面的例子会打印出`The second name is: Brian`。如果你尝试使用一个不在数组中的下标，你会得到一个错误：数组访问会在运行时进行边界检查。这种不适当的访问时其它系统编程语言中很多bug的根源。

你可以在标准库文档中找到更多关于`array`的文档。

片段（Slices）

一个片段（*slice*）是一个数组的引用（或者“视图”）。它有利于安全，有效的访问数组的一部分而不用进行拷贝。比如，你可能只想要引用读入到内存的文件中的一行。原理上，片段并不是直接创建的，而是引用一个已经存在的变量。片段有长度，可以是可变也可以是不可变的，并且表现起来像一个数组：

```
let a = [0, 1, 2, 3, 4];
let middle = &a[1..4]; // A slice of a: just the elements 1, 2, and 3
```

片段拥有`&[T]`类型。当我们涉及到泛型时会讨论这个`T`。

你可以在标准库文档中找到更多关于`slices`的文档。

`str`

Rust的`str`类型是最原始的字符串类型。作为一个不定长类型，它本身并不是非常有用，不过当它用在引用后是就有用了，例如&str。如你所见，我们到时候再讲。

你可以在标准库文档中找到更多关于`str`的文档。

元组（Tuples）

元组（tuples）是固定大小的有序列表。如下：

```
let x = (1, "hello");
```

这是一个长度为2的元组，有括号和逗号组成。下面也是同样的元组，不过注明了数据类型：

```
let x: (i32, &str) = (1, "hello");
```

如你所见，元组的类型跟元组看起来很像，只不过类型取代的值的位置。细心的读者可能会注意到元组是异质的：这个元组中有一个`i32`和一个`&str`。在系统编程语言中，字符串要比其它语言中来的复杂。现在，可以认为`&str`是一个字符串片段（*string slice*），我们马上会讲到它。

你可以把一个元组赋值给另一个，如果它们包含相同的类型和数量。当元组有相同的长度时它们有相同的数量。

```
let mut x = (1, 2); // x: (i32, i32)
let y = (2, 3); // y: (i32, i32)

x = y;
```

你可以通过一个解构*let*（*destructuring let*）访问元组中的字段。下面是一个例子：

```
let (x, y, z) = (1, 2, 3);

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

还记得我曾经说过`let`语句的左侧远比一个赋值绑定强大吗？这就是证据。我们可以在`let`左侧写一个模式，如果它能匹配右侧的话，我们可以一次写多个绑定。这种情况下，`let`“解构”或“拆开”了元组，并分成了三个绑定。

这个模式是很强大的，我们后面会经常看到它。

你可以一个逗号来消除一个单元素元组和一个括号中的值的歧义：

```
(0,); // single-element tuple
(0); // zero in parentheses
```

元组索引

你也可以用索引语法访问一个元组的字段：

```
let tuple = (1, 2, 3);

let x = tuple.0;
let y = tuple.1;
let z = tuple.2;

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

就像数组索引，它从`0`开始，不过也不像数组索引，它使用`.`，而不是`[]`。

你可以在标准库文档中找到更多关于`tuple`的文档。

函数

函数也有一个类型！它们看起来像这样：

```
fn foo(x: i32) -> i32 { x }

let x: fn(i32) -> i32 = foo;
```

在这个例子中，`x`是一个指向一个获取一个`i32`并返回一个`i32`的函数的“函数指针”。

注释

现在我们写了一些函数，是时候学习一下注释了。注释是你帮助其他程序员理解你的代码的备注。编译器基本上会忽略它们。

Rust有两种需要你了解的注释格式：行注释（*line comments*）和文档注释（*doc comments*）。

```
// Line comments are anything after '//' and extend to the end of the line.

let x = 5; // this is also a line comment.

// If you have a long explanation for something, you can put line comments next
// to each other. Put a space between the // and your comment so that it's
// more readable.
```

另一种注释是文档注释。文档注释使用`///`而不是`//`，并且内建Markdown标记支持：

```
/// Adds one to the number given.
///
/// # Examples
///
///
```

/// let five = 5; /// /// assert_eq!(6, add_one(5)); /// `fn add_one(x: i32) -> i32 { x + 1 }`

当书写文档注释时，加上参数和返回值部分并提供一些用例将是非常，非常有帮助的。你会注意到我们在这里用了一个新的宏：`assert_eq!`。它比较两个值，并当它们不相等时`panic!`。这在文档中是非常有帮助的。还有一个宏，`assert!`，它在传递给它的值是`false`的时候`panic!`。

你可以使用rustdoc工具来将文档注释生成为HTML文档，也可以将代码示例作为测试运行！

If语句

Rust的If并不是特别复杂，不过你会发现它更像动态类型语言而不是更传统的系统语言。所以让我来说说它，以便你能把握这些细节。

If语句是分支这个更加宽泛的概念的一个特定形式。它的名字来源于树的树枝：一个选择点，根据选择的不同，将会使用不同的路径。

在If语句中，这里有一个选择导致了两个路径：

```
let x = 5;

if x == 5 {
    println!("x is five!");
}
```

如果在什么别的地方更改了x的值，这一行将不会输出。更具体一点，如果`if`后面的表达式的值为`true`，这个代码块将被执行。为`false`则不被执行。

如果你想当值为`false`时执行些什么，使用`else`：

```
let x = 5;

if x == 5 {
    println!("x is five!");
} else {
    println!("x is not five :(");
}
```

如果不止一种情况，使用`else if`：

```
let x = 5;

if x == 5 {
    println!("x is five!");
} else if x == 6 {
    println!("x is six!");
} else {
    println!("x is not five or six :(");
}
```

这些都是非常标准的情况。然而你也可以这么写：

```
let x = 5;

let y = if x == 5 {
    10
} else {
```

```
    15
}; // y: i32
```

你可以（或许也应该）这么写：

```
let x = 5;

let y = if x == 5 { 10 } else { 15 }; // y: i32
```

这代码可以被执行是因为`if`是一个表达式。表达式的值是任何被选择的分支的最后一个表达式的值。一个没有`else`的`if`总是返回`()`作为返回值。

for循环

`for`用来循环一个特定的次数。然而，Rust的`for`循环与其它系统语言有些许不同。Rust的`for`循环看起来并不像这个“C语言样式”的`for`循环：

```
for (x = 0; x < 10; x++) {
    printf( "%d\n", x );
}
```

相反，它看起来像这个样子：

```
for x in 0..10 {
    println!("{}", x); // x: i32
}
```

更抽象的形式：

```
for var in expression {
    code
}
```

这个表达式是一个迭代器.迭代器返回一系列的元素。每个元素是循环中的一次重复。然后它的值与`var`绑定，它在循环体中有效。每当循环体执行完后，我们从迭代器中取出下一个值，然后我们再重复一遍。当迭代器中不再有值时，`for`循环结束。

在我们的例子中，`0..10`表达式取一个开始和结束的位置，然后给出一个含有这之间值得迭代器。当然它不包括上限值，所以我们的循环会打印`0`到`9`，而不是到`10`。

Rust没有使用“C语言风格”的`for`循环是有意为之的。即使对于有经验的C语言开发者来说，要手动控制要循环的每个元素也都是复杂并且易于出错的。

while循环

Rust也有一个`while`循环。它看起来像：

```
let mut x = 5; // mut x: u32
let mut done = false; // mut done: bool

while !done {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 {
        done = true;
    }
}
```

`while`循环是当你不确定应该循环多少次时正确的选择。

如果你需要一个无限循环，你可能想要这么写：

```
while true {
```

然而，Rust有一个专用的关键字`loop`来处理这个情况：

```
loop {
```

Rust的控制流分析会区别对待这个与`while true`，因为我们知道它会一直循环。现阶段理解这些细节意味着什么并不是非常重要，基本上，你给编译器越多的信息，越能确保安全和生成更好的代码，所以当你打算无限循环的时候应该总是倾向于使用`loop`。

提早结束迭代（Ending iteration early）

让我们再看一眼之前的`while`循环：

```
let mut x = 5;
let mut done = false;

while !done {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 {
        done = true;
    }
}
```

我们必须使用一个`mut`布尔型变量绑定，`done`,来确定何时我们应该推出循环。Rust有两个关键字帮助我们来修改迭代：`break`和`continue`。

这样，我们可以用`break`来写一个更好的循环：

```
let mut x = 5;

loop {
    x += x - 3;
    println!("{}", x);
    if x % 5 == 0 { break; }
}
```

现在我们用`loop`来无限循环，然后用`break`来提前退出循环。

`continue`比较类似，不过不是退出循环，它直接进行下一次迭代。下面的例子只会打印奇数：

```
for x in 0u32..10 {
    if x % 2 == 0 { continue; }

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

`break`和`continue`在`while`循环和`for`循环中都有效。

所有权

这篇教程是现行3个Rust所有权系统之一。所有权系统是Rust最独特且最引人入胜的特性之一，也是作为Rust开发者应该熟悉的。Rust所追求最大的目标 -- 内存安全，关键在于所有权。所有权系统有一些不同的概念，每个概念独自成章：

- 所有权，你正在阅读的这个章节
- 借用，以及它关联的特性: "引用" (references)
- 生命周期，关于借用的高级概念

这3章依次互相关联，你需要完整地阅读全部3章来对Rust的所有权系统进行全面的了解。

原则（Meta）

在我们开始详细讲解之前，这有两点关于所有权系统重要的注意事项。

Rust注重安全和速度。它通过很多零开销抽象（*zero-cost abstractions*）来实现这些目标，也就是说在Rust中，实现抽象的开销尽可能的小。所有权系统是一个典型的零开销抽象的例子。本文提到所有的分析都是在编译时完成的。你不需要在运行时为这些功能付出任何开销。

然而，这个系统确实有一个开销：学习曲线。很多Rust初学者会经历我们所谓的“与借用检查器作斗争”的过程，也就是指Rust编译器拒绝编译一个作者认为合理的程序。这种“斗争”会因为程序员关于所有权系统如何工作的基本模型与Rust实现的实际规则不匹配而经常发生。当你刚开始尝试Rust的时候，你很可能会有相似的经历。然而有一个好消息：更有经验的Rust开发者反应，一旦他们适应所有权系统一段时间之后，与借用检查器的冲突会越来越少。

记住这些之后，让我们来学习关于所有权的内容。

所有权（Ownership）

Rust中的变量绑定有一个属性：它们有它们所绑定的的值的所有权。这意味着当一个绑定离开作用域，它们绑定的资源就会被释放。例如：

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

当`v`进入作用域，一个新的Vec被创建，向量（vector）也在堆上为它的3个元素分配了空间。
当`v`在`foo()`的末尾离开作用域，Rust将会清理掉与向量（vector）相关的一切，甚至是堆上分配的内存。这在作用域的结尾是一定（deterministically）会发生的。

移动语义（Move semantics）

然而这里有更巧妙的地方：Rust确保了对于任何给定的资源都正好（只）有一个绑定与之对应。例如，如

果我们有一个向量（vector），我们可以把它赋予另外一个绑定：

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

let v2 = v;
```

不过，如果之后我们尝试使用`v`，我们得到一个错误：

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

let v2 = v;

println!("v[0] is: {}", v[0]);
```

它看起来像这样：

```
error: use of moved value: `v`
println!("v[0] is: {}", v[0]);
                        ^
```

当我们定义了一个取得所有权的函数，并尝试在我们把变量作为参数传递给函数之后使用这个变量时，会发生相似的事情：

```
fn take(v: Vec<i32>) {
    // what happens here isn't important.
}

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

take(v);

println!("v[0] is: {}", v[0]);
```

一样的错误：“use of moved value”。当我们把所有权转移给别的别的绑定时，我们说我们“移动”了我们引用的值。这里你并不需要什么类型的特殊注解，这是Rust的默认行为。

细节

在移动了绑定后我们不能使用它的原因是微妙的，也是重要的。当我们写了这样的代码：

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

let v2 = v;
```

第一行为向量（vector）对象和它包含的数据分配了内存。向量对象储存在栈上并包含一个指向堆上`[1, 2, 3]`内容的指针。当我们从`v`移动到`v2`，它为`v2`创建了一个那个指针的拷贝。这意味着这将会有两个

指向向量内容的指针。这将会因为引入了一个数据竞争而违反Rust的安全保证。因此，Rust禁止我们在移动后使用 `v` 。

注意到优化可能会根据情况移除栈上字节（例如上面的向量）的实际拷贝也是很重要的。所以它也许并不像它开始看起来那样没有效率。

`Copy` 类型

我们已经知道了当所有权被转移给另一个绑定以后，你不能再使用原始绑定。然而，这里有一个trait会改变这个行为，它叫做 `Copy` 。我们还没有讨论到trait，不过目前，你可以理解为一个为特定类型增加额外行为的标记。例如：

```
let v = 1;

let v2 = v;

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

在这个情况， `v` 是一个 `i32` ，它实现了 `Copy` 。这意味着，就像一个移动，当我们把 `v` 赋值给 `v2` ,产生了一个数据的拷贝。不过，不像一个移动，我们仍可以在之后使用 `v` 。这是因为 `i32` 并没有指向其它数据的指针，对它的拷贝是一个完整的拷贝。

我们会在trait部分讨论如何编写你自己类型的 `Copy` 。

所有权之外（More than ownership）

当然，如果我们不得不在每个我们写的函数中交还所有权：

```
fn foo(v: Vec<i32>) -> Vec<i32> {
    // do stuff with v

    // hand back ownership
    v
}
```

这将会变得烦人。它在我们获取更多变量的所有权时变得更糟：

```
fn foo(v1: Vec<i32>, v2: Vec<i32>) -> (Vec<i32>, Vec<i32>, i32) {
    // do stuff with v1 and v2

    // hand back ownership, and the result of our function
    (v1, v2, 42)
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let (v1, v2, answer) = foo(v1, v2);
```

额！返回值，返回的代码行（上面的最后一行），和函数调用都变得更复杂了。

幸运的是，Rust提供了一个trait，借用，它帮助我们解决这个问题。这个主题将在下一个部分讨论！

引用和借用

这篇教程是现行3个Rust所有权系统之一。所有权系统是Rust最独特且最引人入胜的特性之一，也是作为Rust开发者应该熟悉的。Rust所追求最大的目标 -- 内存安全，关键在于所有权。所有权系统有一些不同的概念，每个概念独自成章：

- 所有权，关键章节
- 借用，你正在阅读的这个章节
- 生命周期，关于借用的高级概念

这3章依次互相关联，你需要完整地阅读全部3章来对Rust的所有权系统进行全面的了解。

原则（**Meta**）

在我们开始详细讲解之前，这有两点关于所有权系统重要的注意事项。

Rust注重安全和速度。它通过很多零开销抽象（*zero-cost abstractions*）来实现这些目标，也就是说在Rust中，实现抽象的开销尽可能的小。所有权系统是一个典型的零开销抽象的例子。本文提到所有的分析都是在编译时完成的。你不需要在运行时为这些功能付出任何开销。

然而，这个系统确实有一个开销：学习曲线。很多Rust初学者会经历我们所谓的“与借用检查器作斗争”的过程，也就是指Rust编译器拒绝编译一个作者认为合理的程序。这种“斗争”会因为程序员关于所有权系统如何工作的基本模型与Rust实现的实际规则不匹配而经常发生。当你刚开始尝试Rust的时候，你很可能会有相似的经历。然而有一个好消息：更有经验的Rust开发者反应，一旦他们适应所有权系统一段时间之后，与借用检查器的冲突会越来越少。

记住这些之后，让我们来学习关于借用的内容。

借用

在所有权章节的最后，我们有一个看起来像这样的糟糕的函数：

```
fn foo(v1: Vec<i32>, v2: Vec<i32>) -> (Vec<i32>, Vec<i32>, i32) {
    // do stuff with v1 and v2

    // hand back ownership, and the result of our function
    (v1, v2, 42)
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let (v1, v2, answer) = foo(v1, v2);
```

然而这并不是理想的Rust代码，因为它没有利用'借用'这个编程语言的特点。这是它的第一步：

```
fn foo(v1: &Vec<i32>, v2: &Vec<i32>) -> i32 {
    // do stuff with v1 and v2

    // return the answer
    42
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let answer = foo(&v1, &v2);

// we can use v1 and v2 here!
```

与其获取`Vec<i32>`作为我们的参数，我们获取一个引用：`&Vec<i32>`。并与其直接传递`v1`和`v2`，我们传递`&v1`和`&v2`。我们称`&T`类型为一个"引用"，而与其拥有这个资源，它借用了所有权。一个借用变量的绑定在它离开作用域时并不释放资源。这意味着`foo()`调用之后，我们可以再次使用原始的绑定。

引用是不可变的，就像绑定一样。这意味着在`foo()`中，向量完全不能被改变：

```
fn foo(v: &Vec<i32>) {
     v.push(5);
}

let v = vec![];

foo(&v);
```

有如下错误：

```
error: cannot borrow immutable borrowed content `*v` as mutable
v.push(5);
^
```

放入一个值改变了向量，所以我们不允许这样做

`&mut`引用

这有第二种类型的引用：`&mut T`。一个"可变引用"允许你改变你借用的资源。例如：

```
let mut x = 5;
{
    let y = &mut x;
    *y += 1;
}
println!("{}", x);
```

这会打印`6`。我们让`y`是一个`x`的可变引用，接着把`y`指向的值加一。你会注意到`x`也必须被标记

为`mut`，如果它不是，我们不能获取一个不可变值的可变引用。

否则，`&mut`引用就像一个普通引用。这两者之间,以及它们是如何交互的有巨大的区别。你会发现在上面的例子有些不太靠谱，因为我们需要额外的作用域，包围在`{`和`}`之间。如果我们移除它们，我们得到一个错误：

```
error: cannot borrow `x` as immutable because it is also borrowed as mutable
    println!("{}", x);
                   ^
note: previous borrow of `x` occurs here; the mutable borrow prevents
subsequent moves, borrows, or modification of `x` until the borrow ends
        let y = &mut x;
                     ^
note: previous borrow ends here
fn main() {

}
^
```

正如这个例子表现的那样，这里有一些规则是你必须要掌握的。

规则

Rust中的借用有一些规则：

第一，任何借用必须位于比拥有着更小的作用域。第二，你可以有一个或另一个这两种类型的借用，不过不能同时拥有它们（这两种）：

- 0个或N个资源的引用（&T）
- 只有1个可变引用（(&mut T）

你可能注意到这些看起来很眼熟，虽然并不完全一样，它类似于数据竞争的定义：

> 当2个或更多个指针同时访问同一内存位置，当它们中至少有1个在写，同时操作并不是同步的时候存在一个“数据竞争”

通过引用，你可以拥有你像拥有的任意多的引用，因为它们没有一个在写。如果你在写，并且你需要2个或更多相同内存的指针，则你只能一次拥有一个`&mut`。这就是Rust如何在编译时避免数据竞争：我们会得到错误，如果我们打破规则的话。

在记住这些之后，让我们再次考虑我们的例子。

理解作用域（**Thinking in scopes**）

这是代码：

```
let mut x = 5;
let y = &mut x;
```

```
*y += 1;

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

这些代码给我们如下错误：

```
error: cannot borrow `x` as immutable because it is also borrowed as mutable
    println!("{}", x);
                   ^
```

这是因为我们违反了规则：我们有一个指向`x`的`&mut T`，所以我们不允许创建任何`&T`。一个或另一个。错误记录提示了我们应该如何理解这个错误：

```
note: previous borrow ends here
fn main() {

}
^
```

换句话说，可变借用在剩下的例子中一直存在。我们需要的是可变借用在我们尝试调用`println!`之前结束并生成一个不可变借用。在Rust中，借用绑定在借用有效的作用域上。而我们的作用域看起来像这样：

```
let mut x = 5;

let y = &mut x;    // -+ &mut borrow of x starts here
                   //  |
*y += 1;           //  |
                   //  |
println!("{}", x); // -+ - try to borrow x here
                   // -+ &mut borrow of x ends here
```

这些作用域冲突了：我们不能在`y`在作用域中时生成一个`&x`。

所以我们增加了一个大括号：

```
let mut x = 5;

{
    let y = &mut x; // -+ &mut borrow starts here
    *y += 1;        //  |
}                   // -+ ... and ends here

println!("{}", x);  // <- try to borrow x here
```

这就没有问题了。我们的可变借用在我们创建一个不可变引用之前离开了作用域。不过作用域是看清一个借用持续多久的关键。

借用避免的问题（**Issues borrowing prevents**）

为什么要有这些限制性规则？好吧，正如我们记录的，这些规则避免了数据竞争。数据竞争能造成何种问题呢？这里有一些。

迭代器失效（Iterator invalidation）

一个例子是“迭代器失效”，它在当你尝试改变你正在迭代的集合时发生。Rust的借用检查器阻止了这些发生：

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

for i in &v {
    println!("{}", i);
}
```

这会打印出1到3.因为我们在向量上迭代，我们只得到了元素的引用。同时`v`本身作为不可变借用，它意味着我们在迭代时不能改变它：

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

for i in &v {
    println!("{}", i);
    v.push(34);
}
```

这里是错误：

```
error: cannot borrow `v` as mutable because it is also borrowed as immutable
    v.push(34);
    ^
note: previous borrow of `v` occurs here; the immutable borrow prevents
subsequent moves or mutable borrows of `v` until the borrow ends
for i in &v {
          ^
note: previous borrow ends here
for i in &v {
    println!(“{}”, i);
    v.push(34);
}
^
```

我们不能修改`v`因为它被循环借用。

释放后使用

引用必须与它引用的值存活得一样长。Rust会检查你的引用的作用域来保证这是正确的。

如果Rust并没有检查这个属性，我们可能意外的使用了一个无效的引用。例如：

```
let y: &i32;
{
    let x = 5;
    y = &x;
}

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

我们得到这个错误：

```
error: `x` does not live long enough
    y = &x;
         ^
note: reference must be valid for the block suffix following statement 0 at
2:16...
let y: &i32;
{
    let x = 5;
    y = &x;
}

note: ...but borrowed value is only valid for the block suffix following
statement 0 at 4:18
    let x = 5;
    y = &x;
}
```

换句话说，`y`只在`x`存在的作用域中有效。一旦`x`消失，它变成无效的引用。为此，这个错误说借用“并没有存活得足够久”因为它在应该有效的时候是无效的。

当引用在它引用的变量之前声明会导致类似的问题：

```
let y: &i32;
let x = 5;
y = &x;

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

我们得到这个错误：

```
error: `x` does not live long enough
y = &x;
     ^
note: reference must be valid for the block suffix following statement 0 at
2:16...
    let y: &i32;
    let x = 5;
    y = &x;

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

```
note: ...but borrowed value is only valid for the block suffix following
statement 1 at 3:14
    let x = 5;
    y = &x;

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

生命周期

这篇教程是现行3个Rust所有权系统之一。所有权系统是Rust最独特且最引人入胜的特性之一，也是作为Rust开发者应该熟悉的。Rust所追求最大的目标 -- 内存安全，关键在于所有权。所有权系统有一些不同的概念，每个概念独自成章：

- 所有权，关键章节
- 借用，以及它关联的特性: "引用" (references)
- 生命周期，你正在阅读的这个章节

这3章依次互相关联，你需要完整地阅读全部3章来对Rust的所有权系统进行全面的了解。

原则（**Meta**）

在我们开始详细讲解之前，这有两点关于所有权系统重要的注意事项。

Rust注重安全和速度。它通过很多零开销抽象（*zero-cost abstractions*）来实现这些目标，也就是说在Rust中，实现抽象的开销尽可能的小。所有权系统是一个典型的零开销抽象的例子。本文提到所有的分析都是在编译时完成的。你不需要在运行时为这些功能付出任何开销。

然而，这个系统确实有一个开销：学习曲线。很多Rust初学者会经历我们所谓的“与借用检查器作斗争”的过程，也就是指Rust编译器拒绝编译一个作者认为合理的程序。这种“斗争”会因为程序员关于所有权系统如何工作的基本模型与Rust实现的实际规则不匹配而经常发生。当你刚开始尝试Rust的时候，你很可能会有相似的经历。然而有一个好消息：更有经验的Rust开发者反应，一旦他们适应所有权系统一段时间之后，与借用检查器的冲突会越来越少。

记住这些之后，让我们来学习有关生命周期的内容。

生命周期

借出一个其它人所有资源的引用可以是很复杂的。例如，想象一下下列操作：

1. 我获取了一个某种资源的句柄
2. 我借给你了一个关于这个资源的引用
3. 我决定不再需要这个资源了，然后释放了它，这时你仍然持有它的引用
4. 你决定使用这个资源

噢！你的引用指向一个无效的资源。这叫做悬垂指针（*dangling pointer*）或者“释放后使用”，如果这个资源是内存的话。

要修正这个问题的话，我们必须确保第四步永远也不在第三步之后发生。Rust所有权系统通过一个叫生命周期（*lifetime*）的概念来做到这一点，它定义了一个引用有效的作用域。

当我们有一个获取引用作为参数的函数，我们可以隐式或显式涉及到引用的生命周期：

```
// implicit
```

```
fn foo(x: &i32) {
}

// explicit
fn bar<'a>(x: &'a i32) {
}
```

`'a` 读作“生命周期a”。技术上讲，每一个引用都有一些与之相关的生命周期，不过编译器在通常情况让你可以省略它们。在我们讲到它之前，让我们拆开显式的例子看看：

```
fn bar<'a>(...)
```

这一部分声明了我们的生命周期。它说 `bar` 有一个生命周期，`'a`。如果我们有两个生命周期，它看起来像这样：

```
fn bar<'a, 'b>(...)
```

接着在我们的参数列表中，我们使用了我们命名的生命周期：

```
...(x: &'a i32)
```

如果我们想要一个 `&mut` 引用，我们这么做：

```
...(x: &'a mut i32)
```

如果你对比一下 `&mut i32` 和 `&'a mut i32`，他们是一样的，只是后者在 `&` 和 `mut i32` 之间夹了一个 `'a` 生命周期。`&mut i32` 读作“一个 `i32` 的可变引用”，而 `&'a mut i32` 读作“一个带有生命周期'a的i32的可变引用”。

当你处理结构体时你也需要显式的生命周期：

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let y = &5; // this is the same as `let _y = 5; let y = &_y;`
    let f = Foo { x: y };

    println!("{}", f.x);
}
```

如你所见，`struct` 也可以有生命周期。跟函数类似的方法，

```
struct Foo<'a> {
```

声明一个生命周期，接着

```
x: &'a i32,
```

使用它。然而为什么这里我们需要一个生命周期呢？因为我们需要确保任何 `Foo` 的引用不能比它包含的 `i32` 的引用活的更久。

理解作用域（Thinking in scopes）

理解生命周期的一个办法是想象一个引用有效的作用域。例如：

```
fn main() {
    let y = &5;     // -+ y goes into scope
                    //  |
    // stuff        //  |
                    //  |
}                   // -+ y goes out of scope
```

加入我们的 `Foo`：

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let y = &5;           // -+ y goes into scope
    let f = Foo { x: y }; // -+ f goes into scope
    // stuff              //  |
                          //  |
}                         // -+ f and y go out of scope
```

我们的 `f` 生存在 `y` 的作用域之中，所以一切正常。那么如果不是呢？下面的代码不能工作：

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let x;                    // -+ x goes into scope
                              //  |
    {                         //  |
        let y = &5;           // ---+ y goes into scope
        let f = Foo { x: y }; // ---+ f goes into scope
        x = &f.x;             //  | | error here
    }                         // ---+ f and y go out of scope
                              //  |
    println!("{}", x);        //  |
}                             // -+ x goes out of scope
```

噢！就像你在这里看到的一样，`f`和`y`的作用域小于`x`的作用域。不过当我们尝试`x = &f.x`时，我们让`x`引用一些将要离开作用域的变量。

命名作用域用来赋予作用域一个名字。有了名字是我们可以谈论它的第一步。

'static

叫做`static`的作用域是特殊的。它代表某样东西具有横跨整个程序的生命周期。大部分Rust程序员当他们处理字符串时第一次遇到`'static`：

```
let x: &'static str = "Hello, world.";
```

基本字符串是`&'static str`类型的因为它的引用一直有效：它们被写入了最终库文件的数据段。另一个例子是全局量：

```
static FOO: i32 = 5;
let x: &'static i32 = &FOO;
```

它在二进制文件的数据段中保存了一个`i32`，而`x`是它的一个引用。

生命周期省略（Lifetime Elision）

Rust支持强大的在函数体中的局部类型推断，不过这在项签名中是禁止的以便允许只通过项签名本身推导出类型。然而，为了一些人道原因有第二个非常限制的叫做“生命周期省略”的推断算法适用于函数签名。它只基于签名部分自身推断而不涉及函数体，只推断生命周期参数，并且只基于3个易于记忆和无歧义的规则，虽然并不隐藏它涉及到的实际类型因为局部推断可能会适用于它。

当我们讨论生命周期省略的时候，我们使用输入生命周期和输出生命周期（*input lifetime and output lifetime.*）。输入生命周期是关于函数参数的，而输出生命周期是关于函数返回值的。例如，这个函数有一个输入生命周期：

```
fn foo<'a>(bar: &'a str)
```

这个有一个输出生命周期：

```
fn foo<'a>() -> &'a str
```

这个两者皆有：

```
fn foo<'a>(bar: &'a str) -> &'a str
```

这里有3条规则：

- 每一个被省略的函数参数成为一个不同的生命周期参数。
- 如果确实有一个输入生命周期，不管是否省略，这个生命周期被赋予所有函数返回值中被省略的生命周期。
- 如果这里有多个输入生命周期，不过它们当中有一个是`&self`或者`&mut self`，`self`的生命周期被赋予所有省略的输出生命周期。

否则，省略一个输出生命周期将是一个错误。

例子

这里有一些省略了生命周期的函数的例子。我们用它们的扩展形式配对了每个省略了生命周期的例子。

```
fn print(s: &str); // elided
fn print<'a>(s: &'a str); // expanded

fn debug(lvl: u32, s: &str); // elided
fn debug<'a>(lvl: u32, s: &'a str); // expanded

// In the preceding example, `lvl` doesn't need a lifetime because it's not a
// reference (`&`). Only things relating to references (such as a `struct`
// which contains a reference) need lifetimes.

fn substr(s: &str, until: u32) -> &str; // elided
fn substr<'a>(s: &'a str, until: u32) -> &'a str; // expanded

fn get_str() -> &str; // ILLEGAL, no inputs

fn frob(s: &str, t: &str) -> &str; // ILLEGAL, two inputs
fn frob<'a, 'b>(s: &'a str, t: &'b str) -> &str; // Expanded: Output lifetime is unclear

fn get_mut(&mut self) -> &mut T; // elided
fn get_mut<'a>(&'a mut self) -> &'a mut T; // expanded

fn args<T:ToCStr>(&mut self, args: &[T]) -> &mut Command // elided
fn args<'a, 'b, T:ToCStr>(&'a mut self, args: &'b [T]) -> &'a mut Command // expanded

fn new(buf: &mut [u8]) -> BufWriter; // elided
fn new<'a>(buf: &'a mut [u8]) -> BufWriter<'a> // expanded
```

可变性

可变性，可以改变事物的能力，用在Rust中与其它语言有些许不同。可变性的第一方面是它并非默认状态：

```
let x = 5;
x = 6; // error!
```

我们可以使用`mut`关键字来引入可变性：

```
let mut x = 5;

x = 6; // no problem!
```

这是一个可变的变量绑定。当一个绑定是可变的，它意味着你可以改变它指向的内容。所以在上面的例子中，`x`的值并没有多大的变化，不过这个绑定从一个`i32`变成了另外一个。

如果你想改变绑定指向的东西（注：原文跟上面一样，有冲突啊。），你将会需要一个可变引用：

```
let mut x = 5;
let y = &mut x;
```

`y`是一个（指向）可变引用的不可变绑定，它意味着你不能把`y`与其它变量绑定（`y = &mut z`），不过你可以改变`y`绑定变量的值（`*y = 5`）。一个微妙的区别。

当然，如果你想它们都可变：

```
let mut x = 5;
let mut y = &mut x;
```

现在`y`可以绑定到另外一个值，并且它引用的值也可以改变。

很重要的一点是`mut`是模式的一部分，所以你可以这样做：

```
let (mut x, y) = (5, 6);

fn foo(mut x: i32) {
```

内部可变性 VS 外部可变性（Interior vs. Exterior Mutability）

然而，当我们谈到Rust中什么是“不可变”的时候，它并不意味着它不能被改变：我们说它有“外部可变性”。例如，考虑下Arc：

```
use std::sync::Arc;

let x = Arc::new(5);
let y = x.clone();
```

当我们调用 `clone()` 时， `Arc<T>` 需要更新引用计数。以为你并未使用任何 `mut` ， `x` 是一个不可变绑定，并且我们也没有取得 `&mut 5` 或者什么。那么发生了什么呢？

为了解释这些，我们不得不回到Rust指导哲学的核心，内存安全，和Rust用以保证它的机制，所有权系统，和更具体的借用：

> 你可能有这两种类型借用的其中一个，但不同同时拥有：
>
> - 0个或N个对一个资源的引用（ `&T` ）
> - 正好1个可变引用（ `&mut T` ）

因此，这就是是“不可变性”的真正定义：当有两个引用指向同一事物是安全的吗？在 `Arc<T>` 的情况下，是安全的：改变完全包含在结构自身内部。它并不面向用户。为此，它用 `clone()` 分配 `&T` 。如果分配 `&mut T` 的话，那么，这将会是一个问题。

其它类型，像std::cell模块中的这一个，则有相反的属性：内部可变性。例如：

```
use std::cell::RefCell;

let x = RefCell::new(42);

let y = x.borrow_mut();
```

`RefCell` 使用 `borrow_mut()` 方法来分配它内部资源的 `&mut` 引用。这难道不危险吗？如果我们：

```
use std::cell::RefCell;

let x = RefCell::new(42);

let y = x.borrow_mut();
let z = x.borrow_mut();
```

事实上这会在运行时引起恐慌。这是 `RefCell` 如何工作的：它在运行时强制使用Rust的借用规则，并且如果有违反就会 `panic!` 。这让我们绕开了Rust可变性规则的另一方面。让我先讨论一下它。

字段级别可变性（Field-level mutability）

可变性是一个不是借用（ `&mut` ）就是绑定的属性（ `&mut` ）。这意味着，例如，你不能拥有一个一些字段可变而一些字段不可变的结构体：

```
struct Point {
    x: i32,
```

```
    mut y: i32, // nope
}
```

结构体的可变性位于它的绑定上：

```
struct Point {
    x: i32,
    y: i32,
}

let mut a = Point { x: 5, y: 6 };

a.x = 10;

let b = Point { x: 5, y: 6};

b.x = 10; // error: cannot assign to immutable field `b.x`
```

然而，通过使用`Cell<T>`，你可以模拟字段级别的可变性：

```
use std::cell::Cell;

struct Point {
    x: i32,
    y: Cell<i32>,
}

let mut point = Point { x: 5, y: Cell::new(6) };

point.y.set(7);

println!("y: {:?}", point.y);
```

这会打印`y: Cell { value: 7 }`。我们成功的更新了`y`。

结构体

结构体是一个创建更复杂数据类型的方法。例如，如果我们正在进行涉及到2D空间坐标的计算，我们将需要一个`x`和一个`y`值：

```
let origin_x = 0;
let origin_y = 0;
```

结构体让我们组合它们俩为一个单独，统一的数据类型：

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let origin = Point { x: 0, y: 0 }; // origin: Point

    println!("The origin is at ({}, {})", origin.x, origin.y);
}
```

这里有许多细节，让我们分开说。我们使用了`struct`关键字后跟名字来定义了一个结构体。根据传统，结构体使用大写字母开头并且使用驼峰命名法：`PointInSpace`而不要写成`Point_In_Space`。

像往常一样我们用`let`创建了一个结构体的实例，不过我们用`key: value`语法设置了每个字段。这里顺序不必和声明的时候一致。

最后，因为每个字段都有名字，我们可以访问字段通过圆点记法：`origin.x`。

结构体中的值默认是不可变的，就像Rust中其它的绑定一样。使用`mut`使其可变：

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let mut point = Point { x: 0, y: 0 };

    point.x = 5;

    println!("The point is at ({}, {})", point.x, point.y);
}
```

上面的代码会打印`The point is at (5, 0)`。

Rust在语言级别不支持字段可变性，所以你不能像这么写：

```
struct Point {
    mut x: i32,
    y: i32,
}
```

可变性是绑定的一个属性，不是结构体自身的。如果习惯于字段级别的可变性，这开始可能看起来有点奇怪，不过这样明显地简化了问题。它甚至可以让你使变量只可变一小段时间：

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let mut point = Point { x: 0, y: 0 };

    point.x = 5;

    let point = point; // this new binding can't change now

    point.y = 6; // this causes an error
}
```

更新语法（Update syntax）

一个包含 `..` 的 `struct` 表明你想要使用一些其它结构体的拷贝的一些值。例如：

```
struct Point3d {
    x: i32,
    y: i32,
    z: i32,
}

let mut point = Point3d { x: 0, y: 0, z: 0 };
point = Point3d { y: 1, .. point };
```

这给了 `point` 一个新的 `y`，不过保留了 `x` 和 `z` 的值。这也并不必要是同样的 `struct`，你可以在创建新结构体时使用这个语法，并会拷贝你未指定的值：

```
let origin = Point3d { x: 0, y: 0, z: 0 };
let point = Point3d { z: 1, x: 2, .. origin };
```

元组结构体

Rust有像另一个元组和结构体的混合体的数据类型。元组结构体有一个名字，不过它的字段没有：

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

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

这两个是不会相等的，即使它们有一模一样的值：

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

使用结构体几乎总是好于使用元组结构体。我们可以这样重写`Color`和`Point`：

```
struct Color {
    red: i32,
    blue: i32,
    green: i32,
}

struct Point {
    x: i32,
    y: i32,
    z: i32,
}
```

现在，我们有了名字，而不是位置。好的名字是很重要的，使用结构体，我们就可以设置名字。

不过有种情况元组结构体非常有用，就是当元组结构体只有一个元素时。我们管它叫新类型（*newtype*），因为你创建了一个与元素相似的类型：

```
struct Inches(i32);

let length = Inches(10);

let Inches(integer_length) = length;
println!("length is {} inches", integer_length);
```

如你所见，你可以通过一个解构`let`来提取内部的整形，就像我们在讲元组时说的那样，`let Inches(integer_length)`给`integer_length`赋值为`10`。

类单元结构体（Unit-like structs）

你可以定义一个没有任何成员的结构体：

```
struct Electron;
```

这样的结构体叫做“类单元”因为它与一个空元组类似，`()`，这有时叫做“单元”。就像一个元组结构体，它定义了一个新类型。

就它本身来看没什么用（虽然有时它可以作为一个标记类型），不过在与其它功能的结合中，它可以变得

有用。例如，一个库可能请求你创建一个实现了一个特定特性的结构来处理事件。如果你并不需要在结构中存储任何数据，你可以仅仅创建一个类单元结构体。

枚举

Rust中的一个`enum`是一个代表数个可能变量的数据的类型：

```
enum Message {
    Quit,
    ChangeColor(i32, i32, i32),
    Move { x: i32, y: i32 },
    Write(String),
}
```

每个变量都可选关联数据。定义变量的语法与用来定义结构体的语法类似：你可以有不带数据的变量（像类单元结构体），带有命名数据的变量，和带有未命名数据的变量（像元组结构体）。然而，不像单独的结构体定义，一个`enum`是一个单独的类型。一个枚举的值可以匹配任何一个变量。因为这个原因，枚举有时被叫做“集合类型”：枚举可能值的集合是每一个变量可能值的集合的总和。

我们使用`::`语法来使用每个变量的名字：它们包含在`enum`名字自身中。这样的话，以下的情况都是可行的：

```
let x: Message = Message::Move { x: 3, y: 4 };

enum BoardGameTurn {
    Move { squares: i32 },
    Pass,
}

let y: BoardGameTurn = BoardGameTurn::Move { squares: 1 };
```

这两个变量都叫做`Move`，不过他们包含在枚举名字中，他们可以无冲突的使用。

枚举类型的一个值包含它是哪个变量的信息，以及任何与变量相关的数据。这有时被作为一个“标记的联合”被提及。因为数据包括一个“标签”表明它的类型是什么。编译器使用这个信息来确保安全的访问枚举中的数据。例如，我们不能简单的尝试解构一个枚举值，就好像它是一个可能的变量一样：

```
fn process_color_change(msg: Message) {
    let Message::ChangeColor(r, g, b) = msg; // compile-time error
}
```

不支持这些操作（比较操作）可能看起来更像限制。不过这是一个我们可以克服的限制。这里有两种方法：我们自己实现相等（比较），或通过`match`表达式模式匹配变量，你会在下一部分学到它。我们还不够了解Rust如何实现相等，不过我们会在特性找到它们。

匹配

一个简单的`if`/`else`往往是不够的，因为你可能有两个或更多个选项。这样`else`也会变得异常复杂，所以我们该如何解决？

Rust有一个`match`关键字，它可以让你有效的取代复杂的`if`/`else`组。看看下面的代码：

```
let x = 5;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    4 => println!("four"),
    5 => println!("five"),
    _ => println!("something else"),
}
```

`match`使用一个表达式然后基于它的值分支。每个分支都是`val => expression`这种形式。当匹配到一个分支，它的表达式将被执行。`match`属于“模式匹配”的范畴，`match`是它的一个实现。

那么这有什么巨大的优势呢？这确实有优势。第一，`match`强制穷尽性检查（*exhaustiveness checking*）。你看到了最后那个下划线开头的分支了吗？如果去掉它，Rust将会给我们一个错误：

```
error: non-exhaustive patterns: `_` not covered
```

换句话说，Rust试图告诉我们，我们忘记了一个值。因为`x`是一个整型，Rust知道它有很多不同的值，比如，`6`。如果没有`_`分支，那么这就没有分支可以匹配了，Rust就会拒绝编译。`_`就像一个匹配所有的分支。如果其它的分支都没有匹配上，就会选择`_`分支，并且因为我们匹配所有的分支，我们现在就有了一个可以表示`x`所有可能的值的分支了，这样我们的程序就能顺利编译了。

`match`也是一个表达式，也就是说它可以用在`let`绑定的右侧或者其它直接用到表达式的地方：

```
let x = 5;

let number = match x {
    1 => "one",
    2 => "two",
    3 => "three",
    4 => "four",
    5 => "five",
    _ => "something else",
};
```

有时，这是一个把一种类型的数据转换为另一个类型的好方法。

匹配枚举（Matching on enums）

`match`的另一个重要的作用是处理枚举的可能变量：

```
enum Message {
    Quit,
    ChangeColor(i32, i32, i32),
    Move { x: i32, y: i32 },
    Write(String),
}

fn quit() { /* ... */ }
fn change_color(r: i32, g: i32, b: i32) { /* ... */ }
fn move_cursor(x: i32, y: i32) { /* ... */ }

fn process_message(msg: Message) {
    match msg {
        Message::Quit => quit(),
        Message::ChangeColor(r, g, b) => change_color(r, g, b),
        Message::Move { x: x, y: y } => move_cursor(x, y),
        Message::Write(s) => println!("{}", s),
    };
}
```

再一次，Rust编译器检查穷尽性，所以它要求对每一个枚举的变量都有一个匹配分支。如果你忽略了一个，除非你用`_`否则它会给你一个编译时错误。

与之前的`match`的作用不同，你不能用常规的`if`语句来做这些。你可以使用if let语句，它可以被看作是一个`match`的简略形式。

模式

模式在Rust中十分常见。我们在变量绑定，匹配语句和其它一些地方使用它们。让我们开始一个快速的关于模式可以干什么的教程！

快速回顾：你可以直接匹配常量，并且`_`作为“任何”类型：

```
let x = 1;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

这会打印出`one`。

多重模式（Multiple patterns）

你可以使用`|`匹配多个模式：

```
let x = 1;

match x {
    1 | 2 => println!("one or two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

这会输出`one or two`。

范围（Ranges）

你可以用`...`匹配一个范围的值：

```
let x = 1;

match x {
    1 ... 5 => println!("one through five"),
    _ => println!("anything"),
}
```

这会输出`one through five`。

范围经常用在整数和`char`上。

```
let x = '';

match x {
    'a' ... 'j' => println!("early letter"),
    'k' ... 'z' => println!("late letter"),
    _ => println!("something else"),
}
```

这会输出`something else`。

绑定

你可以使用`@`把值绑定到名字上：

```
let x = 1;

match x {
    e @ 1 ... 5 => println!("got a range element {}", e),
    _ => println!("anything"),
}
```

这会输出`got a range element 1`。在你想对一个复杂数据结构进行部分匹配的时候，这个特性十分有用：

```
#[derive(Debug)]
struct Person {
    name: Option<String>,
}

let name = "Steve".to_string();
let mut x: Option<Person> = Some(Person { name: Some(name) });
match x {
    Some(Person { name: ref a @ Some(_), .. }) => println!("{:?}", a),
    _ => {}
}
```

这会输出`Some("Steve")`，因为我们把Person里面的`name`绑定到`a`。

如果你在使用`|`的同时也使用了`@`，你需要确保名字在每个模式的每一部分都绑定名字：

```
let x = 5;

match x {
    e @ 1 ... 5 | e @ 8 ... 10 => println!("got a range element {}", e),
    _ => println!("anything"),
}
```

忽略变量（Ignoring variants）

如果你匹配一个带有变量的枚举，你可以用`..`来忽略变量的值和类型：

```
enum OptionalInt {
    Value(i32),
    Missing,
}

let x = OptionalInt::Value(5);

match x {
    OptionalInt::Value(..) => println!("Got an int!"),
    OptionalInt::Missing => println!("No such luck."),
}
```

这会输出`Got an int!`。

守卫（Guards）

你可以用`if`来引入匹配守卫（*match guards*）：

```
enum OptionalInt {
    Value(i32),
    Missing,
}

let x = OptionalInt::Value(5);

match x {
    OptionalInt::Value(i) if i > 5 => println!("Got an int bigger than five!"),
    OptionalInt::Value(..) => println!("Got an int!"),
    OptionalInt::Missing => println!("No such luck."),
}
```

这会输出`Got an int!`。

`ref`和`ref mut`

如果你想要一个引用，使用`ref`关键字：

```
let x = 5;

match x {
    ref r => println!("Got a reference to {}", r),
}
```

这会输出`Got a reference to 5`。

这里，`match`中的`r`是`&i32`类型的。换句话说，`ref`关键字创建了一个在模式中使用的引用。如果你需要一个可变引用，`ref mut`同样可以做到：

```
let mut x = 5;

match x {
    ref mut mr => println!("Got a mutable reference to {}", mr),
}
```

解构（Destructuring）

如果你有一个复合数据类型，例如一个结构体，你可以在模式中解构它：

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { x: x, y: y } => println!("({},{})", x, y),
}
```

如果你只关心部分值，我们不需要给它们都命名：

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { x: x, .. } => println!("x is {}", x),
}
```

这会输出`x is 0`。

你可以对任何成员进行这样的匹配，不仅仅是第一个：

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { y: y, .. } => println!("y is {}", y),
}
```

这会输出`y is 0`。

这种“解构”行为可以用在任何复合数据类型上，例如元组和枚举

混合与匹配（Mix and Match）

(口哨)！根据你的需求，你可以对上面的多种匹配方法进行组合：

```
match x {
    Foo { x: Some(ref name), y: None } => ...
}
```

模式十分强大。好好使用它们。

方法语法

函数是伟大的，不过如果你在一些数据上调用了一堆函数，这将是令人尴尬的。 考虑下面代码：

```
baz(bar(foo(x)));
```

我们可以从左向右阅读，我们会看到“baz bar foo”。不过这不是函数被调用的顺序，调用应该是从内向外的：“foo bar baz”。如果能这么做不是更好吗？

```
x.foo().bar().baz();
```

幸运的是，正如对上面那个问题的猜测，你可以！Rust通过`impl`关键字提供了使用方法调用语法（*method call syntax*）。

方法调用

这是它如何工作的：

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

fn main() {
    let c = Circle { x: 0.0, y: 0.0, radius: 2.0 };
    println!("{}", c.area());
}
```

这会打印`12.566371`。

我们创建了一个代表圆的结构体。我们写了一个`impl`块，并且在里面定义了一个方法，`area`。

方法的第一参数比较特殊，`&self`。它有3种变体：`self`，`&self`和`&mut self`。你可以认为这第一个参数就是`x.foo()`中的`x`。这3种变体对应`x`可能的3种类型：`self`如果它只是栈上的一个值，`&self`如果它是一个引用，然后`&mut self`如果它是一个可变引用。因为我们我们的`area`以`&self`作为参数，我们就可以可以像其他参数那样使用它。因为我们知道是一个`Circle`，我们可以像任何其他结构体那样访问`radius`字段。

我们应该默认使用`&self`，就像相比获取所有权你应该更倾向于借用，同样相比获取可变引用更倾向于不

可变引用一样。这是一个三种变体的例子：

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn reference(&self) {
       println!("taking self by reference!");
    }

    fn mutable_reference(&mut self) {
       println!("taking self by mutable reference!");
    }

    fn takes_ownership(self) {
       println!("taking ownership of self!");
    }
}
```

链式方法调用（Chaining method calls）

现在我们知道如何调用方法了，例如`foo.bar()`。那么我们最开始的那个例子呢，`foo.bar().baz()`？我们称这个为“方法链”，我们可以通过返回`self`来做到这点。

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }

    fn grow(&self) -> Circle {
        Circle { x: self.x, y: self.y, radius: (self.radius * 10.0) }
    }
}

fn main() {
    let c = Circle { x: 0.0, y: 0.0, radius: 2.0 };
    println!("{}", c.area());

    let d = c.grow().area();
    println!("{}", d);
}
```

注意返回值：

```
fn grow(&self) -> Circle {
```

我们看到我们返回了一个`Circle`。通过这个函数，我们可以增长一个圆的面积到任意大小。

关联函数（Associated functions）

我们也可以定义一个不带`self`参数的关联函数。这是一个Rust代码中非常常见的模式：

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn new(x: f64, y: f64, radius: f64) -> Circle {
        Circle {
            x: x,
            y: y,
            radius: radius,
        }
    }
}

fn main() {
    let c = Circle::new(0.0, 0.0, 2.0);
}
```

这个关联函数（*associated function*）为我们构建了一个新的`Circle`。注意静态函数是通过`Struct::method()`语法调用的，而不是`ref.method()`语法。

创建者模式（Builder Pattern）

我们说我们需要我们的用户可以创建圆，不过我们只允许他们设置他们关心的属性。否则，`x`和`y`将是`0.0`，并且`radius`将是`1.0`。Rust并没有方法重载，命名参数或者可变参数。我们利用创建者模式来代替。它看起像这样：

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

struct CircleBuilder {
    coordinate: f64,
```

```
    radius: f64,
}

impl CircleBuilder {
    fn new() -> CircleBuilder {
        CircleBuilder { coordinate: 0.0, radius: 0.0, }
    }

    fn coordinate(&mut self, coordinate: f64) -> &mut CircleBuilder {
    self.coordinate = coordinate;
    self
    }

    fn radius(&mut self, radius: f64) -> &mut CircleBuilder {
    self.radius = radius;
    self
    }

    fn finalize(&self) -> Circle {
        Circle { x: self.coordinate, y: self.coordinate, radius: self.radius }
    }
}

fn main() {
    let c = CircleBuilder::new()
                .coordinate(10.0)
                .radius(5.0)
                .finalize();


    println!("area: {}", c.area());
}
```

我们在这里又声明了一个结构体，`CircleBuilder`。我们给它定义了一个创建者函数。我们也在`Circle`中定义了`area()`方法。我们还定义了另一个方法`CircleBuilder: finalize()`。这个方法从构造器中创建了我们最后的`Circle`。现在我们使用类型系统来强化我们的考虑：我们可以用`CircleBuilder`来强制生成我们需要的`Circle`。

Vectors

> 注：鉴于将`vector`翻译为`向量`容易引起误解，故决定不再对其进行翻译，如果你在本书中看到“向量”一词，这一定是还未修改过来，请自行脑补为`vector`

“Vector”是一个动态或“可增长”的数组，被实现为标准库类型Vec（其中`<T>`是一个泛型语句）。vector总是在堆上分配数据。vector与切片就像`String`与`&str`一样。你可以使用`vec!`宏来创建它：

```
let v = vec![1, 2, 3]; // v: Vec<i32>
```

（与我们之前使用`println!`宏时不一样，我们在`vec!`中使用中括号`[]`。为了方便，Rust允许你使用上述各种情况。）

对于重复初始值有另一种形式的`vec!`：

```
let v = vec![0; 10]; // ten zeroes
```

访问元素

为了vector特定索引的值，我们使用`[]`：

```
let v = vec![1, 2, 3, 4, 5];

println!("The third element of v is {}", v[2]);
```

索引从`0`开始，所以第3个元素是`v[2]`。

迭代

当你有了一个vector，我可以用`for`来迭代它的元素。这里有3个版本：

```
let mut v = vec![1, 2, 3, 4, 5];

for i in &v {
    println!("A reference to {}", i);
}

for i in &mut v {
    println!("A mutable reference to {}", i);
}

for i in v {
    println!("Take ownership of the vector and its element {}", i);
}
```

vector还有很多有用的方法，你可以看看vector的API文档了解它们。

字符串

对于每一个程序，字符串都是需要掌握的重要内容。由于Rust主要着眼于系统编程，所以它的字符串处理系统与其它语言有些许区别。每当你碰到一个可变大小的数据结构时，情况都会变得很微妙，而字符串正是可变大小的数据结构。这也就是说，Rust的字符串与一些像C这样的系统编程语言也不相同。

让我们进一步了解一下。一个字符串是一串UTF-8字节编码的Unicode量级值的序列。所有的字符串都确保是有效编码的UTF-8序列。另外，字符串并不以null结尾并且可以包含null字节。

Rust有两种主要的字符串类型：`&str`和`String`。让我们先看看`&str`。这叫做字符串片段（*string slices*）。字符串常量是`&'static str`类型的：

```
let string = "Hello there."; // string: &'static str
```

这个字符串是静态分配的，也就是说它储存在我们编译好的程序中，并且整个程序的运行过程中一直存在。这个`string`绑定了一个静态分配的字符串的引用。字符串片段是固定大小的并且不能改变。

一个`String`，相反，是一个在堆上分配的字符串。这个字符串可以增长，并且也保证是UTF-8编码的。`String`通常通过一个字符串片段调用`to_string`方法转换而来。

```
let mut s = "Hello".to_string(); // mut s: String
println!("{}", s);

s.push_str(", world.");
println!("{}", s);
```

`String`可以通过一个`&`强制转换为`&str`：

```
fn takes_slice(slice: &str) {
    println!("Got: {}", slice);
}

fn main() {
    let s = "Hello".to_string();
    takes_slice(&s);
}
```

把`String`转换为`&str`的代价几乎可以忽略不计，不过从`&str`转换到`String`涉及到分配内存。除非必要，没有理由这样做！

索引（Indexing）

因为字符串是有效UTF-8编码的，它不支持索引：

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

```
println!("The first letter of s is {}", s[0]); // ERROR!!!
```

通常，用`[]`访问一个数组是非常快的。不过，字符串中每个UTF-8编码的字符可以是多个字节，你必须遍历字符串来找到字符串的第N个字符。这个操作的代价相当高，而且我们不想误导读者。更进一步来讲，Unicode实际上并没有定义什么“字符”。我们可以选择把字符串看作一个串独立的字节，或者代码点（codepoints）：

```
let hachiko = "忠犬ハチ公";

for b in hachiko.as_bytes() {
print!("{}, ", b);
}

println!("");

for c in hachiko.chars() {
print!("{}, ", c);
}

println!("");
```

这会打印：

```
229, 191, 160, 231, 138, 172, 227, 131, 143, 227, 131, 129, 229, 133, 172,
忠, 犬, ハ, チ, 公,
```

如你所见，这里有比`char`更多的字节。

你可以这样来获取跟索引相似的东西：

```
let dog = hachiko.chars().nth(1); // kinda like hachiko[1]
```

这强调了我们不得不遍历整个`char`的列表。

连接（Concatenation）

如果你有一个`String`，你可以在它后面接上一个`&str`：

```
let hello = "Hello ".to_string();
let world = "world!";

let hello_world = hello + world;
```

不过如果你有两个`String`，你需要一个`&`：

```
let hello = "Hello ".to_string();
let world = "world!".to_string();

let hello_world = hello + &world;
```

这是因为`&String`可以自动转换为一个`&str`。这个功能叫做`Deref`转换。

泛型

有时，当你编写函数或数据类型时，我们可能会希望它能处理多种类型的参数。幸运的是，Rust有一个能给我们更好选择的功能：泛型。泛型在类型理论中叫做参数多态（*parametric polymorphism*），它意味着它们是对于给定参数（parametric）能够有多种形式（`poly`是多，`morph`是形态）的函数或类型。

不管怎么样，类型理论就说这么多，现在我们来看些泛型代码。Rust标准库提供了一个类型，`Option<T>`，它是泛型的：

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

`<T>`部分，你可能见过几次了，代表它是一个泛型数据类型。在我们枚举声明中，每当我们看到`T`，我们用这个类型代替我们泛型中使用的类型。下面是一个使用`Option<T>`的例子，它带有额外的类型标注：

```
let x: Option<i32> = Some(5);
```

在类型声明中，我们看到`Option<i32>`。注意它与`Option<T>`的相似之处。所以在这个特定的`Option`中，`T`是`i32`。在绑定的右侧，我们用了`Some(T)`，`T`是`5`。因为那是个`i32`，两边类型相符，所以皆大欢喜。如果不相符，我们会得到一个错误：

```
let x: Option<f64> = Some(5);
// error: mismatched types: expected `core::option::Option<f64>`,
// found `core::option::Option<_>` (expected f64 but found integral variable)
```

这并不是意味着我们不能写一个`f64`的`Option<T>`！只是类型必须相符：

```
let x: Option<i32> = Some(5);
let y: Option<f64> = Some(5.0f64);
```

这样就好了。一个定义，多种用途。

不一定只有一个类型是泛型的。考虑下Rust内建的`Result<T, E>`类型：

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

这里有两个泛型类型：`T`和`E`。另外，大写字母可以是任何你喜欢的（大写）字母。我们可以定义`Result<T, E>`为：

```
enum Result<A, Z> {
    Ok(A),
    Err(Z),
}
```

如果你想这么做的话。惯例告诉我们第一个泛型参数应该是`T`，代表`type`，然后我们用`E`来代表`error`。然而，Rust并不管这些。

`Result<T, E>`意图作为计算的返回值，并为了能够在不能工作时返回一个错误。

泛型函数

我们可以用熟悉的语法编写一个获取泛型参数的函数：

```
fn takes_anything<T>(x: T) {
    // do something with x
}
```

语法有两部分：`<T>`代表“这个函数带有一个泛型类型”，而`x: T`代表“`x`是`T`类型的”。

多个参数可以有相同的泛型类型：

```
fn takes_two_of_the_same_things<T>(x: T, y: T) {
    // ...
}
```

我们可以写一个获取多个（泛型）类型的版本：

```
fn takes_two_things<T, U>(x: T, y: U) {
    // ...
}
```

泛型函数结合“特性约束”时最有用，我们会在特性部分涉及到它。

泛型结构体（Generic structs）

你也可以在一个`struct`中储存泛型类型：

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

let int_origin = Point { x: 0, y: 0 };
let float_origin = Point { x: 0.0, y: 0.0 };
```

与函数类似，`<T>`是我们声明的泛型参数，而我们也接着在类型定义中使用`x: T`。

Traits

你还记得`impl`关键字吗，曾用方法语法调用方法的那个？

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}
```

trait也很类似，除了我们用函数标记来定义一个trait，然后为结构体实现trait。例如：

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

trait HasArea {
    fn area(&self) -> f64;
}

impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}
```

如你所见，`trait`块与`impl`看起来很像，不过我们没有定义一个函数体，只是函数标记。当我们`impl`一个trait时，我们使用`impl Trait for Item`，而不是仅仅`impl Item`。

那么这有什么重要的呢？还记得我们使用泛型`inverse`函数得到的错误吗？

```
error: binary operation `==` cannot be applied to type `T`
```

我们可以用trait来约束我们的泛型。考虑下这个函数，它不能编译并给出一个类似的错误：

```
fn print_area<T>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}
```

Rust抱怨说：

```
error: type `T` does not implement any method in scope named `area`
```

因为`T`可以是任何类型，我们不能确定它实现了`area`方法。不过我们可以在泛型`T`添加一个*trait*约束（*trait constraint*），来确保它实现了对应方法：

```
fn print_area<T: HasArea>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}
```

`<T: HasArea>`语法是指`any type that implements the HasArea trait`（任何实现了`HasArea`trait的类型）。因为trait定义了函数类型标记，我们可以确定任何实现`HasArea`将会拥有一个`.area()`方法。

这是一个扩展的例子演示它如何工作：

```
trait HasArea {
    fn area(&self) -> f64;
}

struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

struct Square {
    x: f64,
    y: f64,
    side: f64,
}

impl HasArea for Square {
    fn area(&self) -> f64 {
        self.side * self.side
    }
}

fn print_area<T: HasArea>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}

fn main() {
    let c = Circle {
        x: 0.0f64,
        y: 0.0f64,
        radius: 1.0f64,
```

```
    };

    let s = Square {
        x: 0.0f64,
        y: 0.0f64,
        side: 1.0f64,
    };

    print_area(c);
    print_area(s);
}
```

这个程序会输出：

```
This shape has an area of 3.141593
This shape has an area of 1
```

如你所见，`print_area`现在是泛型的了，并且确保我们传递了正确的类型。如果我们传递了错误的类型：

```
print_area(5);
```

我们会得到一个编译时错误：

```
error: failed to find an implementation of trait main::HasArea for int
```

目前为止，我们只在结构体上添加trait实现，不过你可以为任何类型实现一个trait。所以从技术上讲，你可以在`i32`上实现`HasArea`：

```
trait HasArea {
    fn area(&self) -> f64;
}

impl HasArea for i32 {
    fn area(&self) -> f64 {
        println!("this is silly");

        *self as f64
    }
}

5.area();
```

在基本类型上实现方法被认为是不好的设计，即便这是可以的。

这看起来有点像狂野西部（Wild West），不过这还有两个限制来避免情况失去控制。第一是如果trait并不定义在你的作用域，它并不能实现。这是个例子：为了进行文件I/O，标准库提供了一个`Write`trait来为`File`增加额外的功能。默认，`File`并不会有这个方法：

```
let mut f = std::fs::File::open("foo.txt").ok().expect("Couldn't open foo.txt");
let result = f.write("whatever".as_bytes());
```

这里是错误：

```
error: type `std::fs::File` does not implement any method in scope named `write`

let result = f.write(b"whatever");
               ^~~~~~~~~~~~~~~~~
```

我们需要先`use` `Write`trait：

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

let mut f = std::fs::File::open("foo.txt").ok().expect("Couldn't open foo.txt");
let result = f.write("whatever".as_bytes());
```

这样就能无错误的编译了。

这意味着即使有人做了像给`int`增加函数这样的坏事，它也不会影响你，除非你`use`了那个trait。

这还有一个实现trait的限制。不管是trait还是你写的`impl`都只能在你自己的包装箱内生效。所以，我们可以为`i32`实现`HasArea`trait，因为`HasArea`在我们的包装箱中。不过如果我们想为`i32`实现`Float`trait，它是由Rust提供的，则无法做到，因为这个trait和类型都不在我们的包装箱中。

关于trait的最后一点：带有trait限制的泛型函数是单态（*monomorphization*）（mono：单一，morph：形式）的，所以它是静态分发（*statically dispatched*）的。这是什么意思？查看trait对象来了解更多细节。

多trait限定（Multiple trait bounds）

你已经见过你可以用一个trait限定一个泛型类型参数：

```
fn foo<T: Clone>(x: T) {
    x.clone();
}
```

如果你需要多于1个限定，可以使用`+`：

```
use std::fmt::Debug;

fn foo<T: Clone + Debug>(x: T) {
    x.clone();
    println!("{:?}", x);
}
```

`T`现在需要实现`Clone`和`Debug`。

where从句（Where clause）

编写只有少量泛型和trait的函数并不算太糟，不过当它们的数量增加，这个语法就看起来比较诡异了：

```
use std::fmt::Debug;

fn foo<T: Clone, K: Clone + Debug>(x: T, y: K) {
    x.clone();
    y.clone();
    println!("{:?}", y);
}
```

函数的名字在最左边，而参数列表在最右边。限制写在中间。

Rust有一个解决方案，它叫“where从句”：

```
use std::fmt::Debug;

fn foo<T: Clone, K: Clone + Debug>(x: T, y: K) {
    x.clone();
    y.clone();
    println!("{:?}", y);
}

fn bar<T, K>(x: T, y: K) where T: Clone, K: Clone + Debug {
    x.clone();
    y.clone();
    println!("{:?}", y);
}

fn main() {
    foo("Hello", "world");
    bar("Hello", "workd");
}
```

`foo()`使用我们刚才的语法，而`bar()`使用`where`从句。所有你所需要做的就是在定义参数时省略限制，然后在参数列表后加上一个`where`。对于很长的列表，你也可以加上空格：

```
use std::fmt::Debug;

fn bar<T, K>(x: T, y: K)
    where T: Clone,
          K: Clone + Debug {

    x.clone();
    y.clone();
    println!("{:?}", y);
}
```

这种灵活性可以使复杂情况变得简洁。

`where`也比基本语法更强大。例如：

```
trait ConvertTo<Output> {
    fn convert(&self) -> Output;
}

impl ConvertTo<i64> for i32 {
    fn convert(&self) -> i64 { *self as i64 }
}

// can be called with T == i32
fn normal<T: ConvertTo<i64>>(x: &T) -> i64 {
    x.convert()
}

// can be called with T == i64
fn inverse<T>() -> T
        // this is using ConvertTo as if it were "ConvertFrom<i32>"
        where i32: ConvertTo<T> {
    1i32.convert()
}
```

这突显出了`where`从句的额外的功能：它允许限制的左侧可以是任意类型（在这里是`i32`），而不仅仅是一个类型参数（比如`T`）。

默认方法（Default methods）

关于trait还有最后一个我们需要讲到的功能。它简单到只需我们展示一个例子：

```
trait Foo {
    fn bar(&self);

    fn baz(&self) { println!("We called baz."); }
}
```

`Foo` trait的实现者需要实现`bar()`，不过并不需要实现`baz()`。它会使用默认的行为。你也可以选择覆盖默认行为：

```
struct UseDefault;

impl Foo for UseDefault {
    fn bar(&self) { println!("We called bar."); }
}

struct OverrideDefault;

impl Foo for OverrideDefault {
    fn bar(&self) { println!("We called bar."); }
```

```
    fn baz(&self) { println!("Override baz!"); }
}

let default = UseDefault;
default.baz(); // prints "We called bar."

let over = OverrideDefault;
over.baz(); // prints "Override baz!"
```

继承（Inheritance）

有时，实现一个trait要求实现另一个trait：

```
trait Foo {
    fn foo(&self);
}

trait FooBar : Foo {
    fn foobar(&self);
}
```

`FooBar`的实现也必须实现`Foo`，像这样：

```
struct Baz;

impl Foo for Baz {
    fn foo(&self) { println!("foo"); }
}

impl FooBar for Baz {
    fn foobar(&self) { println!("foobar"); }
}
```

如果我们忘了实现`Foo`，Rust会告诉我们：

```
error: the trait `main::Foo` is not implemented for the type `main::Baz` [E0277]
```

Drop

现在我们讨论了trait，让我们看看一个由Rust标准库提供的特殊trait，`Drop`。`Drop` trait提供了一个当一个值离开作用域后运行一些代码的方法。例如：

```
struct HasDrop;

impl Drop for HasDrop {
    fn drop(&mut self) {
        println!("Dropping!");
    }
}

fn main() {
    let x = HasDrop;

    // do stuff

} // x goes out of scope here
```

当在`main()`的末尾`x`离开作用域的时候，`Drop`的代码将会执行。`Drop`有一个方法，他也叫做`drop()`。它获取一个`self`的可变引用。

就是这样！`Drop`的机制非常简单，不过这有一些细节。例如，值会以与它们声明相反的顺序被丢弃（dropped）。这是另一个例子：

```
struct Firework {
    strength: i32,
}

impl Drop for Firework {
    fn drop(&mut self) {
        println!("BOOM times {}!!!", self.strength);
    }
}

fn main() {
    let firecracker = Firework { strength: 1 };
    let tnt = Firework { strength: 100 };
}
```

这会输出：

```
BOOM times 100!!!
BOOM times 1!!!
```

`tnt`在`firecracker`之前离开作用域（原文大意：TNT在爆竹之前爆炸），因为它在之后被声明。先进，后出。

那么`Drop`有什么好处呢？通常来说，`Drop`用来清理任何与`struct`关联的资源。例如，`Arc<T>`类型是一个引用计数类型。当`Drop`被调用，它会减少引用计数，并且如果引用的总数为0，将会清除底层的值。

if let

`if let` 允许你合并 `if` 和 `let` 来减少特定类型模式匹配的开销。

例如，让我们假设我们有一些 `Option<T>` 。我们想让它是 `Some<T>` 时在其上调用一个函数，而它是 `None` 时什么也不做。这看起来像：

```
match option {
    Some(x) => { foo(x) },
    None => {},
}
```

我们并不一定要在这使用 `match` ，例如，我们可以使用 `if` ：

```
if option.is_some() {
    let x = option.unwrap();
    foo(x);
}
```

这两种选项都不是特别吸引人。我们可以使用 `if let` 来优雅地完成相同的功能：

```
if let Some(x) = option {
    foo(x);
}
```

如果一个模式匹配成功，它绑定任何值的合适的部分到模式的标识符中，并计算这个表达式。如果模式不匹配，啥也不会发生。

如果你想在模式不匹配时做点其他的，你可以使用 `else` ：

```
if let Some(x) = option {
    foo(x);
} else {
    bar();
}
```

while let

类似的，当你想一直循环，直到一个值匹配到特定的模式的时候，你可以选择使用 `while let` 。使用 `while let` 可以把类似这样的代码：

```
loop {
    match option {
        Some(x) => println!("{}", x),
```

```
    _ => break,
    }
}
```

变成这样的代码：

```
while let Some(x) = option {
    println!("{}", x);
}
```

trait对象

当涉及到多态的代码时，我们需要一个机制来决定哪个具体的版本应该得到执行。这叫做“分发”（dispatch）。大体上有两种形式的分发：静态分发和动态分发。虽然Rust喜欢静态分发，不过它也提供了一个叫做“trait对象”的机制来支持动态分发。

背景

在本章接下来的内容中，我们需要一个trait和一些实现。让我们来创建一个简单的`Foo`。它有一个返回`String`的方法。

```
trait Foo {
    fn method(&self) -> String;
}
```

我们也在`u8`和`String`上实现了这个trait：

```
impl Foo for u8 {
    fn method(&self) -> String { format!("u8: {}", *self) }
}

impl Foo for String {
    fn method(&self) -> String { format!("string: {}", *self) }
}
```

静态分发

我们可以使用trait的限制来进行静态分发：

```
fn do_something<T: Foo>(x: T) {
    x.method();
}

fn main() {
    let x = 5u8;
    let y = "Hello".to_string();

    do_something(x);
    do_something(y);
}
```

在这里Rust用“单态”来进行静态分发。这意味着Rust会为`u8`和`String`分别创建一个特殊版本的的`do_something()`，然后将对`do_something`的调用替换为这些特殊函数。也就是说，Rust生成了一些像这样的函数：

```
fn do_something_u8(x: u8) {
    x.method();
}

fn do_something_string(x: String) {
    x.method();
}

fn main() {
    let x = 5u8;
    let y = "Hello".to_string();

    do_something_u8(x);
    do_something_string(y);
}
```

这样做的一个很大的优点在于：静态分发允许函数被内联调用，因为调用者在编译时就知道它，内联对编译器进行代码优化十分有利。静态分发能提高程序的运行效率，不过相应的也有它的弊端：会导致“代码膨胀”（code bloat）。因为在编译出的二进制程序中，同样的函数，对于每个类型都会有不同的拷贝存在。

此外，编译器也不是完美的并且“优化”后的代码可能更慢。例如，过度的函数内联会导致指令缓存膨胀（缓存控制着我们周围的一切）。这也是为何要谨慎使用`#[inline]`和`#[inline(always)]`的部分原因。另外一个使用动态分发的原因是，在一些情况下，动态分发更有效率。

然而，常规情况下静态分发更有效率，并且我们总是可以写一个小的静态分发的封装函数来进行动态分发，不过反过来不行，这就是说静态调用更加灵活。因为这个原因标准库尽可能的使用了静态分发。

动态分发

Rust通过一个叫做“trait对象”的功能提供动态分发。比如说`&Foo`、`Box<Foo>`这些就是trait对象。它们是一些值，值中储存实现了特定trait的任意类型。它的具体类型只能在运行时才能确定。

从一些实现了特定`trait`的类型的指针中，可以从通过转型(casting)（例如，`&x as &Foo`）或者强制转型(coercing it)（例如，把`&x`当做参数传递给一个接收`&Foo`类型的函数）来取得trait对象。

这些trait对象的强制多态和转型也适用于类似于`&mut Foo`的`&mut T`以及`Box<Foo>`的`Box<T>`这样的指针，也就是目前为止我们讨论到的所有指针。强制转型和转型是一样的。

这个操作可以被看作“清除”编译器关于特定类型指针的信息，因此trait对象有时被称为“类型清除”（type erasure）。

回到上面的例子，我们可以使用相同的trait，通过trait对象的转型（casting）来进行动态分发：

```
fn do_something(x: &Foo) {
    x.method();
}

fn main() {
    let x = 5u8;
    do_something(&x as &Foo);
```

```
}
```

或者通过强制转型（by concercing）：

```
fn do_something(x: &Foo) {
    x.method();
}

fn main() {
    let x = "Hello".to_string();
    do_something(&x);
}
```

一个使用trait对象的函数并没有为每个实现了 `Foo` 的类型专门生成函数：它只有一份函数的代码，一般（但不总是）会减少代码膨胀。然而，因为调用虚函数，会带来更大的运行时开销，也会大大地阻止任何内联以及相关优化的进行。

为什么用指针？

和很多托管语言不一样，Rust默认不用指针来存放数据，因此类型有着不同的大小。在编译时知道值的大小（size），以及了解把值作为参数传递给函数、值在栈上移动、值在堆上分配（或释放）并储存等情况，对于Rust程序员来说是很重要的。

对于 `Foo`，我们需要一个值至少是一个 `String`（24字节）或一个 `u8`（1字节），或者其它crate中可能实现了 `Foo`（任意字节）的其他类型。如果值没有使用指针存储，我们无法保证代码能对其他类型正常运作，因为其它类型可以是任意大小的。

用指针来储存值意味着当我们使用trait对象时值的大小（size）是无关的，只与指针的大小（size）有关。

表现（Representation）

可以在一个trait对象上通过一个特殊的函数指针的记录调用的特性函数通常叫做“虚函数表”（由编译器创建和管理）。

trait对象既简单又复杂：它的核心表现和设计是十分直观的，不过这有一些难懂的错误信息和诡异行为有待发掘。

让我们从一个简单的，带有trait对象的运行时表现开始。 `std::raw` 模块包含与复杂的内建类型有相同结构的结构体，包括trait对象：

```
pub struct TraitObject {
    pub data: *mut (),
    pub vtable: *mut (),
}
```

这就是了，一个trait对象就像包含一个“数据”指针和“虚函数表”指针的 `&Foo`。

数据指针指向trait对象保存的数据（某个未知的类型 `T`），和一个虚表指针指向对应 `T` 的 `Foo` 实现的虚函

数表。

一个虚表本质上是一个函数指针的结构体，指向每个函数实现的具体机器码。一个像 `trait_object.method()` 的函数调用会从虚表中取出正确的指针然后进行一个动态调用。例如：

```
struct FooVtable {
    destructor: fn(*mut ()),
    size: usize,
    align: usize,
    method: fn(*const ()) -> String,
}

// u8:

fn call_method_on_u8(x: *const ()) -> String {
    // the compiler guarantees that this function is only called
    // with `x` pointing to a u8
    let byte: &u8 = unsafe { &*(x as *const u8) };

    byte.method()
}

static Foo_for_u8_vtable: FooVtable = FooVtable {
    destructor: /* compiler magic */,
    size: 1,
    align: 1,

    // cast to a function pointer
    method: call_method_on_u8 as fn(*const ()) -> String,
};


// String:

fn call_method_on_String(x: *const ()) -> String {
    // the compiler guarantees that this function is only called
    // with `x` pointing to a String
    let string: &String = unsafe { &*(x as *const String) };

    string.method()
}

static Foo_for_String_vtable: FooVtable = FooVtable {
    destructor: /* compiler magic */,
    // values for a 64-bit computer, halve them for 32-bit ones
    size: 24,
    align: 8,

    method: call_method_on_String as fn(*const ()) -> String,
};
```

在每个虚表中的 `destructor` 字段指向一个会清理虚表类型的任何资源的函数，对于 `u8` 是普通的，不过对于 `String` 它会释放内存。这对于像 `Box<Foo>` 这类有所有权的trait对象来说是必要的，它需要在离开作用域后清理 `Box` 分配和他内部的类型。 `size` 和 `align` 字段储存需要清除类型的大小和它的对齐情况；它们原理上是无用的因为这些信息已经嵌入了析构函数中，不过在将来会被使用到，因为trait对象正日益变得更

灵活。

假设我们有一些实现了 `Foo` 的值，那么显式的创建和使用 `Foo` trait对象可能看起来有点像这个（忽略不匹配的类型，它们只是指针而已）：

```
let a: String = "foo".to_string();
let x: u8 = 1;

// let b: &Foo = &a;
let b = TraitObject {
    // store the data
    data: &a,
    // store the methods
    vtable: &Foo_for_String_vtable
};

// let y: &Foo = x;
let y = TraitObject {
    // store the data
    data: &x,
    // store the methods
    vtable: &Foo_for_u8_vtable
};

// b.method();
(b.vtable.method)(b.data);

// y.method();
(y.vtable.method)(y.data);
```

如果 `b` 或 `y` 拥有trait对象（ `Box<Foo>` ），在它们离开作用域后会有一个 `(b.vtable.destructor)(b.data)` （相应的还有 `y` 的）调用。

闭包

Rust不只有命名函数，也有匿名函数。有一个相关的环境的匿名函数叫做“闭包”，因为它们包含在同一个环境中。正如我们将看到的，Rust里面有大量闭包的实现。

语法

闭包看起来像这样：

```
let plus_one = |x: i32| x + 1;

assert_eq!(2, plus_one(1));
```

我们创建了一个绑定，`plus_one`，并把它赋予一个闭包。闭包的参数位于管道（`|`）之中，而闭包体是一个表达式，在这个例子中，`x + 1`。记住`{}`是一个表达式，所以我们也可以拥有包含多行的闭包：

```
let plus_two = |x| {
    let mut result: i32 = x;

    result += 1;
    result += 1;

    result
};

assert_eq!(4, plus_two(2));
```

你会注意到闭包的一些方面与用`fn`定义的常规函数有点不同。第一个是我们并不需要标明闭包接收和返回参数的类型。我们可以：

```
let plus_one = |x: i32| -> i32 { x + 1 };

assert_eq!(2, plus_one(1));
```

不过我们并不需要这么写。为什么呢？基本上，这是出于“人体工程学”的原因。因为为命名函数指定全部类型有助于像文档和类型推断，而闭包的类型则很少有文档因为它们是匿名的，并且并不会产生像推断一个命名函数的类型这样的“远距离错误”。

第二个是语法是相似的，不过有点不同。我会增加空格来使它们看起来更像一点：

```
fn  plus_one_v1   (x: i32 ) -> i32 { x + 1 }
let plus_one_v2 = |x: i32 | -> i32 { x + 1 };
let plus_one_v3 = |x: i32 |          x + 1  ;
```

有些小区别，不过仍然是相似的。

闭包及环境

之所以把它称为“闭包”是因为它们“包含在环境中”（close over their environment）。这看起来像：

```
let num = 5;
let plus_num = |x: i32| x + num;

assert_eq!(10, plus_num(5));
```

这个闭包，`plus_num`，引用了它作用域中的`let`绑定：`num`。更明确的说，它借用了绑定。如果我们做一些会与这个绑定冲突的事，我们会得到一个错误。比如这个：

```
let mut num = 5;
let plus_num = |x: i32| x + num;

let y = &mut num;
```

错误是：

```
error: cannot borrow `num` as mutable because it is also borrowed as immutable
    let y = &mut num;
                ^~~
note: previous borrow of `num` occurs here due to use in closure; the immutable
  borrow prevents subsequent moves or mutable borrows of `num` until the borrow
  ends
    let plus_num = |x| x + num;
                   ^~~~~~~~~~
note: previous borrow ends here
fn main() {
    let mut num = 5;
    let plus_num = |x| x + num;

    let y = &mut num;
}
^
```

一个啰嗦但有用的错误信息！如它所说，我们不能取得一个`num`的可变借用因为闭包已经借用了它。如果我们让闭包离开作用域，我们可以：

```
let mut num = 5;
{
    let plus_num = |x: i32| x + num;

} // plus_num goes out of scope, borrow of num ends

let y = &mut num;
```

如果你的闭包需要它，然而，相反Rust会取得所有权并移动环境：

```
let nums = vec![1, 2, 3];

let takes_nums = || nums;

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

这会给我们：

```
note: `nums` moved into closure environment here because it has type
  `[closure(()) -> collections::vec::Vec<i32>]`, which is non-copyable
let takes_nums = || nums;
                    ^~~~~~
```

`Vec<T>`拥有它内容的所有权，而且由于这个原因，当我们在闭包中引用它时，我们必须取得`nums`的所有权。这与我们传递`nums`给一个取得它所有权的函数一样。

`move`闭包

我们可以使用`move`关键字强制使我们的闭包取得它环境的所有权：

```
let num = 5;

let owns_num = move |x: i32| x + num;
```

现在，即便关键字是`move`，变量遵循正常的移动语义。在这个例子中，`5`实现了`Copy`，所以`owns_num`取得一个`5`的拷贝的所有权。那么区别是什么呢？

```
let mut num = 5;

{
    let mut add_num = |x: i32| num += x;

    add_num(5);
}

assert_eq!(10, num);
```

那么在这个例子中，我们的闭包取得了一个`num`的可变引用，然后接着我们调用了`add_num`，它改变了其中的值，正如我们期望的。我们也需要将`add_num`声明为`mut`，因为我们会改变它的环境。

如果我们改为一个`move`闭包，这有些不同：

```
let mut num = 5;

{
    let mut add_num = move |x: i32| num += x;
```

```
    add_num(5);
}

assert_eq!(5, num);
```

我们只会得到 `5` 。与其获取一个我们 `num` 的可变借用，我们取得了一个拷贝的所有权。

另一个理解 `move` 闭包的方法：它给出了一个拥有自己栈帧的闭包。没有 `move` ，一个闭包可能会绑定在创建它的栈帧上，而 `move` 闭包则是独立的。例如，这意味着大体上你不能从函数返回一个非 `move` 闭包。

不过在我们讨论获取或返回闭包之前，我们应该更多的了解一下闭包实现的方法。作为一个系统语言，Rust给予你了大量的控制你代码的能力，而闭包也是一样。

闭包实现

Rust的闭包实现与其它语言有些许不用。它们实际上是trait的语法糖。在这以前你会希望阅读trait章节，和静态和动态分发（已改为trait对象章节）章节，它讲到了trait对象。

都理解吗？很好。

理解闭包底层是如何工作的关键有点奇怪：使用 `()` 调用函数，像 `foo()` ，是一个可重载的运算符。到此，其它的一切都会明了。在Rust中，我们使用trait系统来重载运算符。调用函数也不例外。我们有三个trait来分别重载：

```
pub trait Fn<Args> : FnMut<Args> {
    extern "rust-call" fn call(&self, args: Args) -> Self::Output;
}

pub trait FnMut<Args> : FnOnce<Args> {
    extern "rust-call" fn call_mut(&mut self, args: Args) -> Self::Output;
}

pub trait FnOnce<Args> {
    type Output;

    extern "rust-call" fn call_once(self, args: Args) -> Self::Output;
}
```

你会注意到这些trait之间的些许区别，不过一个大的区别是 `self` ： `Fn` 获取 `&self` ， `FnMut` 获取 `&mut self` ，而 `FnOnce` 获取 `self` 。这包含了所有3种通过通常函数调用语法的 `self` 。不过我们将它们分在3个trait里，而不是单独的1个。这给了我们大量的对于我们可以使用哪种闭包的控制。

闭包的 `|| {}` 语法是上面3个trait的语法糖。Rust将会为了环境创建一个结构体， `impl` 合适的trait，并使用它。

闭包作为参数（**Taking closures as arguments**）

现在我们知道了闭包是trait，我们已经知道了如何接受和返回闭包；就像任何其它的trait！

这也意味着我们也可以选择静态或动态分发。首先，让我们写一个获取可调用结构的函数，调用它，然后返回结果：

```
fn call_with_one<F>(some_closure: F) -> i32
    where F : Fn(i32) -> i32 {

    some_closure(1)
}

let answer = call_with_one(|x| x + 2);

assert_eq!(3, answer);
```

我们传递我们的闭包，`|x| x + 2`，给`call_with_one`。它正做了我们说的：它调用了闭包，`1`作为参数。

让我们更深层的解析`call_with_one`的签名：

```
fn call_with_one<F>(some_closure: F) -> i32
```

我们获取一个参数，而它有类型`F`。我们也返回一个`i32`。这一部分并不有趣。下一部分是：

```
    where F : Fn(i32) -> i32 {
```

因为`Fn`是一个trait，我们可以用它限制我们的泛型。在这个例子中，我们的闭包取得一个`i32`作为参数并返回`i32`，所以我们用泛型限制是`Fn(i32) -> i32`。

还有一个关键点在于：因为我们用一个trait限制泛型，它会是单态的，并且因此，我们在闭包中使用静态分发。这是非常简单的。在很多语言中，闭包固定在堆上分配，所以总是进行动态分发。在Rust中，我们可以在栈上分配我们闭包的环境，并静态分发调用。这经常发生在迭代器和它们的适配器上，它们经常取得闭包作为参数。

当然，如果我们想要动态分发，我们也可以做到。trait对象处理这种情况，通常：

```
fn call_with_one(some_closure: &Fn(i32) -> i32) -> i32 {
    some_closure(1)
}

let answer = call_with_one(&|x| x + 2);

assert_eq!(3, answer);
```

现在我们取得一个trait对象，一个`&Fn`。并且当我们将我们的闭包传递给`call_with_one`时我们必须获取一个引用，所以我们试用`&||`。

返回闭包（**Returning closures**）

对于函数式风格代码来说在各种情况返回闭包是非常常见的。如果你尝试返回一个闭包，你可能会得到一个错误。在刚接触的时候，这看起来有点奇怪，不过我们会搞清楚。当你尝试从函数返回一个闭包的时候，你可能会写出类似这样的代码：

```
fn factory() -> (Fn(i32) -> Vec<i32>) {
    let vec = vec![1, 2, 3];

    |n| vec.push(n)
}

let f = factory();

let answer = f(4);
assert_eq!(vec![1, 2, 3, 4], answer);
```

编译的时候会给出这一长串相关错误：

```
error: the trait `core::marker::Sized` is not implemented for the type
`core::ops::Fn(i32) -> collections::vec::Vec<i32>` [E0277]
f = factory();
^
note: `core::ops::Fn(i32) -> collections::vec::Vec<i32>` does not have a
constant size known at compile-time
f = factory();
^
error: the trait `core::marker::Sized` is not implemented for the type
`core::ops::Fn(i32) -> collections::vec::Vec<i32>` [E0277]
factory() -> (Fn(i32) -> Vec<i32>) {
             ^~~~~~~~~~~~~~~~~~~~
note: `core::ops::Fn(i32) -> collections::vec::Vec<i32>` does not have a constant size kn
own at compile-time
fa ctory() -> (Fn(i32) -> Vec<i32>) {
              ^~~~~~~~~~~~~~~~~~~~
```

为了从函数返回一些东西，Rust需要知道返回类型的大小。不过`Fn`是一个trait，它可以是各种大小(size)的任何东西。比如说，返回值可以是实现了`Fn`的任意类型。一个简单的解决方法是：返回一个引用。因为引用的大小(size)是固定的，因此返回值的大小就固定了。因此我们可以这样写：

```
fn factory() -> &(Fn(i32) -> Vec<i32>) {
    let vec = vec![1, 2, 3];

    |n| vec.push(n)
}

let f = factory();

let answer = f(4);
assert_eq!(vec![1, 2, 3, 4], answer);
```

不过这样会出现另外一个错误：

```
error: missing lifetime specifier [E0106]
fn factory() -> &(Fn(i32) -> i32) {
                ^~~~~~~~~~~~~~~~~
```

对。因为我们有一个引用，我们需要给它一个生命周期。不过我们的`factory()`函数不接收参数，所以省略不能用在这。我们可以使用什么生命周期呢？`'static`：

```
fn factory() -> &'static (Fn(i32) -> i32) {
    let num = 5;

    |x| x + num
}

let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

不过这样又会出现另一个错误：

```
error: mismatched types:
 expected `&'static core::ops::Fn(i32) -> i32`,
    found `[closure <anon>:7:9: 7:20]`
(expected &-ptr,
    found closure) [E0308]
         |x| x + num
         ^~~~~~~~~~
```

这个错误让我们知道我们并没有返回一个`&'static Fn(i32) -> i32`，而是返回了一个`[closure <anon>:7:9: 7:20]`。等等，什么？

因为每个闭包生成了它自己的环境`struct`并实现了`Fn`和其它一些东西，这些类型是匿名的。它们只在这个闭包中存在。所以Rust把它们显示为`closure <anon>`，而不是一些自动生成的名字。

不过为什么我们的闭包没有实现`&'static Fn`呢？正如我们之前讨论的，闭包借用了它们所在的环境。而在这个例子中，我们的环境是基于栈分配的，跟`num`变量绑定的`5`。所以这个借用有一个在栈帧中的生命周期。如果我们返回这个闭包，这一函数调用将会结束，栈帧将会消失，而我们的闭包获取到了一个垃圾内存的环境！

那么我们该怎么做？这个几乎可以成功运行了：

```
fn factory() -> Box<Fn(i32) -> i32> {
    let num = 5;

    Box::new(|x| x + num)
}
let f = factory();

let answer = f(1);
```

```
assert_eq!(6, answer);
```

我们使用一个trait对象，通过`Box`把`Fn`装箱。不过还有最后一个错误：

```
error: `num` does not live long enough
Box::new(|x| x + num)
         ^~~~~~~~~~
```

我们仍有一个指向父栈帧的引用。加上这一个最后的修改后，这段代码可以成功运行了：

```
fn factory() -> Box<Fn(i32) -> i32> {
    let num = 5;

    Box::new(move |x| x + num)
}
let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

通过把内部闭包变为`move Fn`，我们为闭包创建了一个新的栈帧。通过`Box`装箱，我们提供了一个已知大小的返回值，并允许它离开我们的栈帧。

通用函数调用语法

有时，函数可能有相同的名字。就像下面这些代码：

```
trait Foo {
    fn f(&self);
}

trait Bar {
    fn f(&self);
}

struct Baz;

impl Foo for Baz {
    fn f(&self) { println!("Baz's impl of Foo"); }
}

impl Bar for Baz {
    fn f(&self) { println!("Baz's impl of Bar"); }
}

let b = Baz;
```

如果我们尝试调用`b.f()`，我们会得到一个错误：

```
error: multiple applicable methods in scope [E0034]
b.f();
  ^~~
note: candidate #1 is defined in an impl of the trait `main::Foo` for the type
`main::Baz`
    fn f(&self) { println!("Baz's impl of Foo"); }
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
note: candidate #2 is defined in an impl of the trait `main::Bar` for the type
`main::Baz`
    fn f(&self) { println!("Baz's impl of Bar"); }
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

我们需要一个区分我们需要调用哪一函数的方法。这个功能叫做“通用函数调用语法”（universal function call syntax），这看起来像这样：

```
Foo::f(&b);
Bar::f(&b);
```

让我们拆开来看。

```
Foo::
Bar::
```

调用的这一半是两个traits的类型：`Foo`和`Bar`。这样实际上就区分了这两者：Rust调用你使用的trait里面的方法。

```
f(&b)
```

当我们使用方法语法调用像`b.f()`这样的方法时，如果`f()`需要`&self`，Rust实际上会自动地把`b`借用为`&self`。而在这个例子中，Rust并不会这么做，所以我们需要显式地传递一个`&b`。

尖括号形式（Angle-bracket Form）

我们刚才讨论的通用函数调用语法的形式：

```
Trait::method(args);
```

上面的形式其实是一种缩写。这是在一些情况下需要使用的扩展形式：

```
<Type as Trait>::method(args);
```

`<>::`语法是一个提供类型提示的方法。类型位于`<>`中。在这个例子中，类型是`Type as Trait`，表示我们想要`method`的`Trait`版本被调用。在没有二义时`as Trait`部分是可选的。尖括号也是一样。因此上面的形式就是一种缩写的形式。

这是一个使用较长形式的例子。

```
trait Foo {
    fn clone(&self);
}

#[derive(Clone)]
struct Bar;

impl Foo for Bar {
    fn clone(&self) {
        println!("Making a clone of Bar");

        <Bar as Clone>::clone(self);
    }
}
```

这会调用`Clone`trait的`clone()`方法，而不是`Foo`的。

包装箱和模块

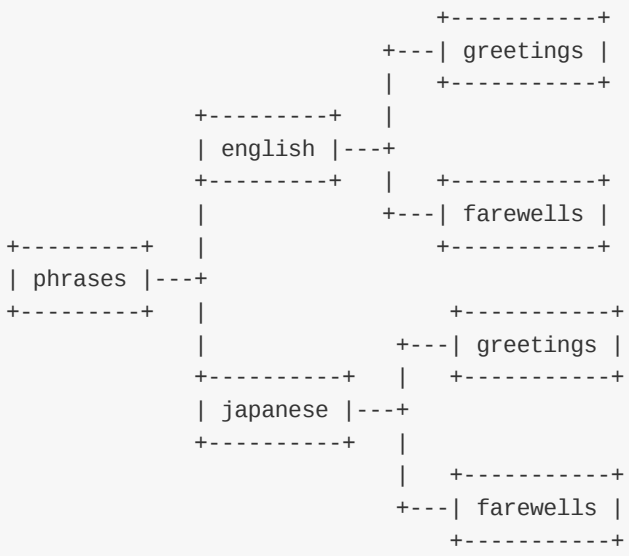
当一个项目开始变得更大，把它分为一堆更小的部分然后再把它们装配到一起被认为是一个好的软件工程实践。另外定义良好的接口也非常重要，这样有些函数是私有的而有些是公有的。Rust有一个模块系统来帮助我们处理这些工作。

基础术语：包装箱和模块

Rust有两个不同的术语与模块系统有关：包装箱（*crate*）和模块（*module*）。包装箱是其它语言中库或包的同义词。因此“Cargo”则是Rust包管理工具的名字：你通过Cargo把你当包装箱交付给别人。包装箱可以根据项目的不同生成可执行文件或库文件。

每个包装箱有一个隐含的根模块（*root module*）包含模块的代码。你可以在根模块下定义一个子模块树。模块允许你为自己模块的代码分区。

作为一个例子，让我们来创建一个短语（*phrases*）包装箱，它会给我们一些不同语言的短语。为了使事情变得简单，我们仅限于“你好”和“再见”这两个短语，并使用英语和日语的短语。我们采用如下模块布局：



在这个例子中，`phrases`是我们包装箱的名字。剩下的所有都是模块。你可以看到它们组成了一个树，它们以包装箱为根，这同时也是树的根：`phrases`。

现在我们有了一个计划，让我们在代码中定义这些模块。让我们以用Cargo创建一个新包装箱作为开始：

```
$ cargo new phrases
$ cd phrases
```

如果你还记得，这会为我们生成一个简单的项目：

```
$ tree .
```

```
.
├── Cargo.toml
└── src
    └── lib.rs

1 directory, 2 files
```

`src/lib.rs`是我们包装箱的根，与上面图表中的`phrases`对应。

定义模块

我们用`mod`关键字来定义我们的每一个模块。让我们把`src/lib.rs`写成这样：

```
// in src/lib.rs

mod english {
    mod greetings {

    }

    mod farewells {

    }
}

mod japanese {
    mod greetings {

    }

    mod farewells {

    }
}
```

在`mod`关键字之后是模块的名字。模块的命名采用Rust其它标识符的命名惯例：`lower_snake_case`。在大括号中（`{}`）是模块的内容。

在`mod`中，你可以定义子`mod`。我们可以用双冒号（`::`）标记访问子模块。我们的4个嵌套模块是`english::greetings`，`english::farewells`，`japanese::greetings`和`japanese::farewells`。因为子模块位于父模块的命名空间中，所以这些不会冲突：`english::greetings`和`japanese::greetings`是不同的，即便它们的名字都是`greetings`。

因为这个包装箱的根文件叫做`lib.rs`，且没有一个`main()`函数。Cargo会把这个包装箱构建为一个库：

```
$ cargo build
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
$ ls target
deps  libphrases-a7448e02a0468eaa.rlib  native
```

`libphrase-hash.rlib`是构建好的包装箱。在我们了解如何使用这个包装箱之前，先让我们把它拆分为多个文件。

多文件包装箱

如果每个包装箱只能有一个文件，这些文件将会变得非常庞大。把包装箱分散到多个文件也非常简单，Rust支持两种方法。

除了这样定义一个模块外：

```
mod english {
    // contents of our module go here
}
```

我们还可以这样定义：

```
mod english;
```

如果我们这么做的话，Rust会期望能找到一个包含我们模块内容的`english.rs`文件，或者包含我们模块内容的`english/mod.rs`文件：

注意在这些文件中，你不需要重新定义这些文件：它们已经由最开始的`mod`定义。

使用这两个技巧，我们可以将我们的包装箱拆分为两个目录和七个文件：

```
$ tree .
.
├── Cargo.lock
├── Cargo.toml
├── src
│   ├── english
│   │   ├── farewells.rs
│   │   ├── greetings.rs
│   │   └── mod.rs
│   ├── japanese
│   │   ├── farewells.rs
│   │   ├── greetings.rs
│   │   └── mod.rs
│   └── lib.rs
└── target
    └── debug
        ├── build
        ├── deps
        ├── examples
        ├── libphrases-a7448e02a0468eaa.rlib
        └── native
```

`src/lib.rs`是我们包装箱的根，它看起来像这样：

```
mod english;
mod japanese;
```

这两个定义告诉Rust去寻找`src/english.rs`和`src/japanese.rs`，或者`src/english/mod.rs`和`src/japanese/mod.rs`，具体根据你的偏好。在我们的例子中，因为我们的模块含有子模块，所以我们选择第二种方式。`src/english/mod.rs`和`src/japanese/mod.rs`都看起来像这样：

```
mod greetings;
mod farewells;
```

再一次，这些定义告诉Rust去寻找`src/english/greetings.rs`和`src/japanese/greetings.rs`，或者`src/english/farewells/mod.rs`和`src/japanese/farewells/mod.rs`。因为这些子模块没有自己的子模块，我们选择`src/english/greetings.rs`和`src/japanese/farewells.rs`。

现在`src/english/greetings.rs`和`src/japanese/farewells.rs`都是空的。让我们添加一些函数。

在`src/english/greetings.rs`添加如下：

```
fn hello() -> String {
    "Hello!".to_string()
}
```

在`src/english/farewells.rs`添加如下：

```
fn goodbye() -> String {
    "Goodbye.".to_string()
}
```

在`src/japanese/greetings.rs`添加如下：

```
fn hello() -> String {
    "こんにちは".to_string()
}
```

当然，你可以从本文复制粘贴这些内容，或者写点别的东西。事实上你写进去“konnichiwa”对我们学习模块系统并不重要。

在`src/japanese/farewells.rs`添加如下：

```
fn goodbye() -> String {
    "さようなら".to_string()
}
```

（这是“Sayōnara”，如果你很好奇的话。）

现在我们在包装箱中添加了一些函数，让我们尝试在别的包装箱中使用它。

导入外部的包装箱

我们有了一个库包装箱。让我们创建一个可执行的包装箱来导入和使用我们的库。

创建一个`src/main.rs`文件然后写入如下：（现在它还不能编译）

```
extern crate phrases;

fn main() {
    println!("Hello in English: {}", phrases::english::greetings::hello());
    println!("Goodbye in English: {}", phrases::english::farewells::goodbye());

    println!("Hello in Japanese: {}", phrases::japanese::greetings::hello());
    println!("Goodbye in Japanese: {}", phrases::japanese::farewells::goodbye());
}
```

`extern crate`声明告诉Rust我们需要编译和链接`phrases`包装箱。然后我们就可以在这里使用`phrases`的模块了。就想我们之前提到的，你可以用双冒号引用子模块和之中的函数。

另外，Cargo假设`src/main.rs`是二进制包装箱的根，而不是库包装箱的。现在我们的包中有两个包装箱：`src/lib.rs`和`src/main.rs`。这种模式在可执行包装箱中非常常见：大部分功能都在库包装箱中，而可执行包装箱使用这个库。这样，其它程序可以只使用我们的库，另外这也是各司其职的良好分离。

现在它还不能很好的工作。我们会得到4个错误，它们看起来像：

```
$ cargo build
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
src/main.rs:4:38: 4:72 error: function `hello` is private
src/main.rs:4     println!("Hello in English: {}", phrases::english::greetings::hello());
                                                    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
note: in expansion of format_args!
<std macros>:2:25: 2:58 note: expansion site
<std macros>:1:1: 2:62 note: in expansion of print!
<std macros>:3:1: 3:54 note: expansion site
<std macros>:1:1: 3:58 note: in expansion of println!
phrases/src/main.rs:4:5: 4:76 note: expansion site
```

Rust的一切默认都是私有的。让我们深入了解一下这个。

导出公用接口

Rust允许你严格的控制你的接口哪部分是公有的，所以它们默认都是私有的。你需要使用`pub`关键字，来公开它。让我们先关注`english`模块，所以让我们像这样减少`src/main.rs`的内容：

```
extern crate phrases;

fn main() {
```

```
    println!("Hello in English: {}", phrases::english::greetings::hello());
    println!("Goodbye in English: {}", phrases::english::farewells::goodbye());
}
```

在我们的`src/lib.rs`，让我们给`english`模块声明添加一个`pub`：

```
pub mod english;
mod japanese;
```

然后在我们的`src/english/mod.rs`中，加上两个`pub`：

```
pub mod greetings;
pub mod farewells;
```

在我们的`src/english/greetings.rs`中，让我们在`fn`声明中加上`pub`：

```
pub fn hello() -> String {
    "Hello!".to_string()
}
```

然后在`src/english/farewells.rs`中：

```
pub fn goodbye() -> String {
    "Goodbye.".to_string()
}
```

这样，我们的包装箱就可以编译了，虽然会有警告说我们没有使用`japanese`的方法：

```
$ cargo run
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
src/japanese/greetings.rs:1:1: 3:2 warning: function is never used: `hello`, #[warn(dead_
code)] on by default
src/japanese/greetings.rs:1 fn hello() -> String {
src/japanese/greetings.rs:2     "こんにちは".to_string()
src/japanese/greetings.rs:3 }
src/japanese/farewells.rs:1:1: 3:2 warning: function is never used: `goodbye`, #[warn(dea
d_code)] on by default
src/japanese/farewells.rs:1 fn goodbye() -> String {
src/japanese/farewells.rs:2     "さようなら".to_string()
src/japanese/farewells.rs:3 }
     Running `target/debug/phrases`
Hello in English: Hello!
Goodbye in English: Goodbye.
```

现在我们的函数是公有的了，我们可以使用它们。好的！然而，`phrases::english::greetings::hello()`非常长并且重复。Rust有另一个关键字用来导入名字到当前空间中，这样我们就可以用更短的名字来引用它们。让我们聊聊`use`。

用 `use` 导入模块

Rust有一个`use`关键字，它允许我们导入名字到我们本地的作用域中。让我们把`src/main.rs`改成这样：

```
extern crate phrases;

use phrases::english::greetings;
use phrases::english::farewells;

fn main() {
    println!("Hello in English: {}", greetings::hello());
    println!("Goodbye in English: {}", farewells::goodbye());
}
```

这两行`use`导入了两个模块到我们本地作用域中，这样我们就可以用一个短得多的名字来引用函数。作为一个传统，当导入函数时，导入模块而不是直接导入函数被认为是一个最佳实践。也就是说，你可以这么做：

```
extern crate phrases;

use phrases::english::greetings::hello;
use phrases::english::farewells::goodbye;

fn main() {
    println!("Hello in English: {}", hello());
    println!("Goodbye in English: {}", goodbye());
}
```

不过这并不理想。这意味着更加容易导致命名冲突。在我们的小程序中，这没什么大不了的，不过随着我们的程序增长，它将会成为一个问题。如果我们有命名冲突，Rust会给我们一个编译错误。举例来说，如果我们将`japanese`的函数设为公有，然后这样尝试：

```
extern crate phrases;

use phrases::english::greetings::hello;
use phrases::japanese::greetings::hello;

fn main() {
    println!("Hello in English: {}", hello());
    println!("Hello in Japanese: {}", hello());
}
```

Rust会给我们一个编译时错误：

```
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
src/main.rs:4:5: 4:40 error: a value named `hello` has already been imported in this modu
le [E0252]
src/main.rs:4 use phrases::japanese::greetings::hello;
                  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

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

如果你从同样的模块中导入多个名字，我们不必写多遍。Rust有一个简便的语法：

```
use phrases::english::greetings;
use phrases::english::farewells;
```

你可以使用大括号：

```
use phrases::english::{greetings, farewells};
```

这两种声明是等同的，不过第二种少打更多字。

使用`pub use`重导出

你不仅可以用`use`来简化标识符。你也可以在包装箱内用它重导出函数到另一个模块中。这意味着你可以展示一个外部接口可能并不直接映射到内部代码结构。

让我们看个例子。修改`src/main.rs`让它看起来像这样：

```
extern crate phrases;

use phrases::english::{greetings,farewells};
use phrases::japanese;

fn main() {
    println!("Hello in English: {}", greetings::hello());
    println!("Goodbye in English: {}", farewells::goodbye());

    println!("Hello in Japanese: {}", japanese::hello());
    println!("Goodbye in Japanese: {}", japanese::goodbye());
}
```

然后修改`src/lib.rs`公开`japanese`模块：

```
pub mod english;
pub mod japanese;
```

接下来，把这两个函数声明为公有，先是`src/japanese/greetings.rs`：

```
pub fn hello() -> String {
    "こんにちは".to_string()
}
```

然后是`src/japanese/farewells.rs`：

```
pub fn goodbye() -> String {
    "さようなら".to_string()
}
```

最后，修改你的`src/japanese/mod.rs`为这样：

```
pub use self::greetings::hello;
pub use self::farewells::goodbye;

mod greetings;
mod farewells;
```

`pub use`声明将这些函数导入到了我们模块结构空间中。因为我们在`japanese`模块内使用了`pub use`，我们现在有了`phrases::japanese::hello()`和`phrases::japanese::goodbye()`函数，即使它们的代码在`phrases::japanese::greetings::hello()`和`phrases::japanese::farewells::goodbye()`函数中。内部结构并不反映外部接口。

这里我们对每个我们想导入到`japanese`空间的函数使用了`pub use`。我们也可以使用通配符来导入`greetings`的一切到当前空间中：`pub use self::greetings::*`。

那么`self`怎么办呢？好吧，默认，`use`声明是绝对路径，从你的包装箱根开始。`self`则使路径相对于你在结构中的当前位置。这里有一个更特殊的`use`形式：你可以使用`use super::`来到达你树中当前位置的上一级。一些同学喜欢把`self`看作`.`而把`super`看作`..`，它们在许多shell表示为当前目录和父目录。

除了`use之`外，路径是相对的：`foo::bar()`引用一个相对我们位置的`foo`中的函数。如果它带有`::`前缀，它引用了一个不同的`foo`，一个从你包装箱根开始的绝对路径。

另外，注意`pub use`出现在`mod`定义之前。Rust要求`use`位于最开始。

构建然后运行：

```
$ cargo run
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
     Running `target/debug/phrases`
Hello in English: Hello!
Goodbye in English: Goodbye.
Hello in Japanese: こんにちは
Goodbye in Japanese: さようなら
```

`const` 和 `static`

Rust有一个用`const`关键字定义常量的方法：

```
const N: i32 = 5;
```

与let绑定不同，你必须标注一个`const`的类型。

常量贯穿于整个程序的生命周期。更具体的，Rust中的常量并没有固定的内存地址。这是因为实际上它们会被内联到用到它们的地方。为此对同一常量的引用并不能保证引用到相同的内存地址。

`static`

Rust以静态量的方式提供了类似“全局变量”的功能。它们与常量类似，不过静态量在使用时并不内联。这意味着对每一个值只有一个实例，并且位于内存中的固定位置。

这是一个例子：

```
static N: i32 = 5;
```

与let绑定不同，你必须标注一个`static`的类型。

静态量贯穿于整个程序的生命周期，因此任何存储在常量中的引用有一个`'static`生命周期：

```
static NAME: &'static str = "Steve";
```

可变性

你可以用`mut`关键字引入可变性：

```
static mut N: i32 = 5;
```

因为这是可变的，一个线程可能在更新`N`同时另一个在读取它，导致内存不安全。因此访问和改变一个`static mut`是不安全（unsafe）的，因此必须在`unsafe`块中操作：

```
unsafe {
    N += 1;

    println!("N: {}", N);
}
```

更进一步，任何存储在`static`的类型必须实现`Sync`。

初始化

`const`和`static`都要求赋予它们一个值。它们只能被赋予一个常量表达式的值。换句话说，你不能用一个函数调用的返回值或任何相似的复合值或在运行时赋值。

我应该用哪个？（**Which construct should I use?**）

几乎所有时候，如果你可以在两者之间选择，选择`const`。实际上你很少需要你的常量关联一个内存位置，而且使用常量允许你不止在在自己的包装箱还可以在下游包装箱中使用像常数扩散这样的优化。

一个常量可以看作一个C中的`#define`：它有元数据开销但无运行时开销。“我应该在C中用一个#define还是一个static呢？”大体上与在Rust你应该用常量还是静态量是一个问题。

属性

在Rust中声明可以用“属性”标注，它们看起来像：

```
#[test]
```

或像这样：

```
#![test]
```

这两者的区别是`!`，它改变了属性作用的对象：

```
#[foo]
struct Foo;

mod bar {
    #![bar]
}
```

`#[foo]`作用于下一个项，在这就是`struct`声明。`#![bar]`作用于包含它的项，在这是`mod`声明。否则，它们是一样的。它们都以某种方式改变它们附加到的项的意义。

例如，考虑一个像这样的函数：

```
#[test]
fn check() {
    assert_eq!(2, 1 + 1);
}
```

它被标记为`#[test]`。这意味着它是特殊的：当你运行测试，这个函数将会执行。当你正常编译时，它甚至不会被包含进来。这个函数现在是一个测试函数。

属性也可以有附加数据：

```
#[inline(always)]
fn super_fast_fn() {
```

或者甚至是键值：

```
#[cfg(target_os = "macos")]
mod macos_only {
```

Rust属性被用在一系列不同的地方。在参考手册中有一个属性的全表。目前，你不能创建你自己的属性，

Rust编译器定义了它们。

`type`别名

`type`关键字让你定义另一个类型的别名：

```
type Name = String;
```

你可以像一个真正类型那样使用这个类型：

```
type Name = String;

let x: Name = "Hello".to_string();
```

然而要注意的是，这一个别名，完全不是一个新的类型。换句话说，因为Rust是强类型的，你可以预期两个不同类型的比较会失败：

```
let x: i32 = 5;
let y: i64 = 5;

if x == y {
   // ...
}
```

这给出

```
error: mismatched types:
 expected `i32`,
    found `i64`
(expected i32,
    found i64) [E0308]
     if x == y {
             ^
```

不过，如果我们有一个别名：

```
type Num = i32;

let x: i32 = 5;
let y: Num = 5;

if x == y {
   // ...
}
```

这会无错误的编译。从任何角度来说，`Num`类型的值与`i32`类型的值都是一样的。

你也可以在泛型中使用类型别名：

```
use std::result;

enum ConcreteError {
    Foo,
    Bar,
}

type Result<T> = result::Result<T, ConcreteError>;
```

这创建了一个特定版本的`Result`类型，它总是有一个`ConcreteError`作为`Result<T, E>`的`E`那部分。这通常用于标准库中创建每个子部分的自定义错误。例如，`io::Result`。

类型转换

Rust，和它对安全的关注，提供了两种不同的在不同类型间转换的方式。第一个，`as`，用于安全转换。相反，`transmute`允许任意的转换，而这是Rust中最危险的功能之一！

as

`as`关键字进行了基本的转换：

```
let x: i32 = 5;

let y = x as i64;
```

然而，它只允许特定类型的转换：

```
let a = [0u8, 0u8, 0u8, 0u8];

let b = a as u32; // four eights makes 32
```

这会报错：

```
error: non-scalar cast: `[u8; 4]` as `u32`
let b = a as u32; // four eights makes 32
        ^~~~~~~~
```

由于这里我们有多个值：数组的4个元素,因而它是一个“混合类型转换”。这种类型的转换灰常危险，因为它们假设了多个底层结构的实现方式。为此，我们需要一些更危险的东西。

transmute

`transmute`函数由编译器固有功能提供，它做的工作非常简单，不过非常可怕。它告诉Rust对待一个类型的值就像它是另一个类型一样。它这样做并不管类型检查系统，并单单完全信任你。

在我们之前的例子中，我们知道一个有4个`u8`的数组可以正常代表一个`u32`，并且我们想要进行转换。使用`transmute`而不是`as`，Rust允许我们：

```
use std::mem;

unsafe {
    let a = [0u8, 0u8, 0u8, 0u8];

    let b = mem::transmute::<[u8; 4], u32>(a);
}
```

为了使它编译通过我们要把这些操作封装到一个`unsafe`块中。技术上讲，只有`mem::transmute`调用自身需要位于块中，不过在这个情况下包含所有相关的内容是有好处的，这样你就知道该看哪了。在这例子中，`a`的细节也是重要的，所以它们放到了块中。你会看到各种风格的代码，有时上下文离得太远，因此在`unsafe`中包含所有的代码并不是一个好主意。

虽然`transmute`做了非常少的检查，至少它确保了这些类型是相同大小的，这个错误：

```
use std::mem;

unsafe {
    let a = [0u8, 0u8, 0u8, 0u8];

    let b = mem::transmute::<[u8; 4], u64>(a);
}
```

和：

```
error: transmute called on types with different sizes: [u8; 4] (32 bits) to u64
(64 bits)
```

除了这些，你可以自行随意转换，只能帮你这么多了！

关联类型

关联类型是Rust类型系统中非常强大的一部分。它涉及到'类型族'的概念，换句话说，就是把多种类型归于一类。这个描述可能比较抽象，所以让我们深入研究一个例子。如果你想编写一个`Graph` trait，你需要泛型化两个类型：点类型和边类型。所以你可能会像这样写一个trait，`Graph<N, E>`：

```
trait Graph<N, E> {
    fn has_edge(&self, &N, &N) -> bool;
    fn edges(&self, &N) -> Vec<E>;
    // etc
}
```

虽然这可以工作，不过显得很尴尬，例如，任何需要一个`Graph`作为参数的函数都需要泛型化的`N'ode`和`E'dge`类型：

```
fn distance<N, E, G: Graph<N, E>>(graph: &G, start: &N, end: &N) -> u32 { ... }
```

我们的距离计算并不需要`Edge`类型，所以函数签名中`E`只是写着玩的。

我们需要的是对于每一种`Graph`类型，都使用一个特定的的`N`ode和`E`dge类型。我们可以用关联类型来做到这一点：

```
trait Graph {
    type N;
    type E;

    fn has_edge(&self, &Self::N, &Self::N) -> bool;
    fn edges(&self, &Self::N) -> Vec<Self::E>;
    // etc
}
```

现在，我们使用一个抽象的`Graph`了：

```
fn distance<G: Graph>(graph: &G, start: &G::N, end: &G::N) -> uint { ... }
```

这里不再需要处理`E`dge类型了。

让我们更详细的回顾一下。

定义关联类型

让我们构建一个`Graph` trait。这里是定义：

```
trait Graph {
```

```
    type N;
    type E;

    fn has_edge(&self, &Self::N, &Self::N) -> bool;
    fn edges(&self, &Self::N) -> Vec<Self::E>;
}
```

十分简单。关联类型使用`type`关键字，并出现在trait体和函数中。

这些`type`声明跟函数定义一样。例如，如果我们想`N`类型实现`Display`，这样我们就可以打印出点类型，我们可以这样写：

```
use std::fmt;

trait Graph {
    type N: fmt::Display;
    type E;

    fn has_edge(&self, &Self::N, &Self::N) -> bool;
    fn edges(&self, &Self::N) -> Vec<Self::E>;
}
```

实现关联类型

就像任何trait，使用关联类型的trait用`impl`关键字来提供实现。下面是一个`Graph`的简单实现：

```
struct Node;

struct Edge;

struct MyGraph;

impl Graph for MyGraph {
    type N = Node;
    type E = Edge;

    fn has_edge(&self, n1: &Node, n2: &Node) -> bool {
        true
    }

    fn edges(&self, n: &Node) -> Vec<Edge> {
        Vec::new()
    }
}
```

这个可笑的实现总是返回`true`和一个空的`Vec<Edge>`，不过它提供了如何实现这类trait的思路。首先我们需要3个`struct`，一个代表图，一个代表点，还有一个代表边。如果使用别的类型更合理，也可以那样做，我们只是准备使用`struct`来代表这3个类型。

接下来是`impl`行，它就像其它任何trait的实现。

在这里，我们使用`=`来定义我们的关联类型。trait使用的名字出现在`=`的左边，而我们`impl`的具体类型出现在右边。最后，我们在函数声明中使用具体类型。

trait对象和关联类型

这里还有另外一个我们需要讨论的语法：trait对象。如果你创建一个关联类型的trait对象，像这样：

```
let graph = MyGraph;
let obj = Box::new(graph) as Box<Graph>;
```

你会得到两个错误：

```
error: the value of the associated type `E` (from the trait `main::Graph`) must
be specified [E0191]
let obj = Box::new(graph) as Box<Graph>;
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
24:44 error: the value of the associated type `N` (from the trait
`main::Graph`) must be specified [E0191]
let obj = Box::new(graph) as Box<Graph>;
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

我们不能这样创建一个trait对象，因为我们并不知道关联的类型。相反，我们可以这样写：

```
let graph = MyGraph;
let obj = Box::new(graph) as Box<Graph<N=Node, E=Edge>>;
```

`N=Node`语法允许我们提供一个具体类型，`Node`，作为`N`类型参数。`E=Edge`也是一样。如果我们不提供这个限制，我们不能确定应该`impl`那个来匹配trait对象。

不定长类型

大部分类型有一个特定的大小，以字节为单位，它们在编译时是已知的。例如，一个`i32`是32位大，或者4个字节。然而，这里有些类型有益于表达，却没有一个定义的大小。它们叫做“不定长”或者“动态大小”类型。一个例子是`[T]`。这个类型代表一个特定数量`t`的序列。不过我们并不知道有多少，所以大小是未知的。

Rust知道几个这样的类型，不过它们有一些限制。这有三个：

1. 我们只能通过指针操作一个不定长类型的实例。`&[T]`刚好能正常工作，不过`[T]`不行。一个`&[T]`能正常工作，不过一个`[T]`不行。
2. 变量和参数不能拥有动态大小类型。
3. 只有一个`struct`的最后一个字段可能拥有一个动态大小类型；其它字段则不可以拥有动态大小类型。枚举变量不可以用动态大小类型作为数据。

所以为什么这很重要？好吧，因为`[T]`只能用在一个指针之后，如果我们没有对不定长类型的语言支持，它将不可能这么写：

```
impl Foo for str {
```

或者

```
impl<T> Foo for [T] {
```

相反，你将不得不这么写：

```
impl Foo for &str {
```

意味深长的是，这个实现将只能用于引用，并且不能用于其它类型的指针。通过`impl for str`，所有指针，包括（在一些地方，这里会有bug需要修复）用户自定义的智能指针，可以使用这个`impl`。

?Sized

如果你想要写一个接受动态大小类型的函数，你可以使用这个特殊的限制，`?Sized`：

```
struct Foo<T: ?Sized> {
    f: T,
}
```

这个`?`，读作“`T`可能是`Sized`的”，意味着这个限制是特殊的：它让我们的匹配更宽松，而不是相反。这几乎像每个`T`都隐式拥有`T: Sized`一样，`?`放松了这个默认（限制）。

运算符与重载

Rust允许有限形式的运算符重载。这里有特定的运算符可以被重载。为了支持一个类型间特定的运算符，这里有一个你可以实现的特定的特性，它接着重载运算符。

例如，`+`运算符可以通过`Add`特性重载：

```
use std::ops::Add;

#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

impl Add for Point {
    type Output = Point;

    fn add(self, other: Point) -> Point {
        Point { x: self.x + other.x, y: self.y + other.y }
    }
}

fn main() {
    let p1 = Point { x: 1, y: 0 };
    let p2 = Point { x: 2, y: 3 };

    let p3 = p1 + p2;

    println!("{:?}", p3);
}
```

在`main`中，我们可以对我们的两个`Point`用`+`号，因为我们已经为`Point`实现了`Add<Output=Point>`。

这里有一系列可以这样被重载的运算符，并且所有与之相关的特性都在`std::ops`模块中。查看它的文档来获取完整的列表。

实现这些特性要遵循一个模式。让我们仔细看看`Add`：

```
pub trait Add<RHS = Self> {
    type Output;

    fn add(self, rhs: RHS) -> Self::Output;
}
```

这里总共涉及到3个类型：你`impl Add`的类型，`RHS`，它默认是`Self`，和`Output`。对于一个表达式`let z = x + y`，`x`是`Self`类型的，`y`是`RHS`，而`z`是`Self::Output`类型。

```
impl Add<i32> for Point {
```

```
    type Output = f64;

    fn add(self, rhs: i32) -> f64 {
        // add an i32 to a Point and get an f64
    }
}
```

将允许你这样做：

```
let p: Point = // ...
let x: f64 = p + 2i32;
```

`Deref` 强制多态

标准库提供了一个特殊的特性，`Deref`。它一般用来重载`*`，解引用运算符：

```
use std::ops::Deref;

struct DerefExample<T> {
    value: T,
}

impl<T> Deref for DerefExample<T> {
    type Target = T;

    fn deref(&self) -> &T {
        &self.value
    }
}

fn main() {
    let x = DerefExample { value: 'a' };
    assert_eq!('a', *x);
}
```

这对编写自定义指针类型很有用。然而，这里有一个与`Deref`相关的语言功能：“解引用强制多态（deref coercions）”。规则如下：如果你有一个`U`类型，和它的实现`Deref<Target=T>`，（那么）`&U`的值将会自动转换为`&T`。这是一个例子：

```
fn foo(s: &str) {
    // borrow a string for a second
}

// String implements Deref<Target=str>
let owned = "Hello".to_string();

// therefore, this works:
foo(&owned);
```

在一个值的前面用`&`号获取它的引用。所以`owned`是一个`String`，`&owned`是一个`&String`，而因为`impl Deref<Target=str> for String`，`&String`将会转换为`&str`，而它是`foo()`需要的。

这就是了。这是Rust唯一一个为你进行一个自动转换的地方，不过它增加了很多灵活性。例如，`Rc<T>`类型实现了`Deref<Target=T>`，所以这可以工作：

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

fn foo(s: &str) {
    // borrow a string for a second
}
```

```
// String implements Deref<Target=str>
let owned = "Hello".to_string();
let counted = Rc::new(owned);

// therefore, this works:
foo(&counted);
```

我们所做的一切就是把我们的`String`封装到了一个`Rc<T>`里。不过现在我们可以传递`Rc<String>`给任何我们有一个`String`的地方。`foo`的签名并无变化，不过它对这两个类型都能正常工作。这个例子有两个转换：`Rc<String>`转换为`String`接着是`String`转换为`&str`。只要类型匹配Rust将可以做任意多次这样的转换。

标准库提供的另一个非常通用的实现是：

```
fn foo(s: &[i32]) {
    // borrow a slice for a second
}

// Vec<T> implements Deref<Target=[T]>
let owned = vec![1, 2, 3];

foo(&owned);
```

向量可以`Deref`为一个片段。

`Deref`和方法调用

当调用一个方法时`Deref`也会出现。换句话说，这两个（应该是`T`和`&T`）在Rust中是一样的：

```
struct Foo;

impl Foo {
    fn foo(&self) { println!("Foo"); }
}

let f = Foo;

f.foo();
```

即便`f`不是一个引用，而`foo`获取`&self`，这也是可以工作的。因为这些都是一样的：

```
f.foo();
(&f).foo();
(&&f).foo();
(&&&&&&&&f).foo();
```

一个`&&&&&&&&&&&&&&&&Foo`类型的值仍然可以调用`Foo`定义的方法，因为编译器会插入足够多的`*`来使类型正确。而正因为它插入`*`，它用了`Deref`。

宏

到目前为止你已经学到了不少Rust提供的抽象和重用代码的工具了。这些代码重用单元有丰富的语义结构。例如，函数有类型标记，类型参数有特性限制并且能重载的函数必须属于一个特定的特性。

这些结构意味着Rust核心抽象拥有强大的编译时正确性检查。不过作为代价的是灵活性的减少。如果你识别出一个重复代码的模式，你会发现把它们解释为泛型函数，特性或者任何Rust语义中的其它结构很难或者很麻烦。

宏允许我们在句法水平上进行抽象。宏是一个“可扩展”句法形式的速记。这个扩展发生在编译的早期，在任何静态检查之前。因此，宏可以实现很多Rust核心抽象不能做到的代码重用模式。

缺点是基于宏的代码更难懂，因为它很少利用Rust的内建规则。就像一个常规函数，一个通用的宏可以在不知道其实现的情况下使用。然而，设计一个通用的宏困难的！另外，在宏中的编译错误更难解释，因为它在扩展代码上描述问题，不是在开发者使用的代码级别。

这些缺点让宏成了所谓“最后求助于的功能”。这并不是说宏的坏话；只是因为它是Rust中需要真正简明，良好抽象的代码的部分。切记权衡取舍。

定义一个宏

你可能见过 `vec!` 宏。用来初始化一个任意数量元素的vector。

```
let x: Vec<u32> = vec![1, 2, 3];
```

这不可能是一个常规函数，因为它可以接受任何数量的参数。不过我们可以想象的到它是这些代码的句法简写：

```
let x: Vec<u32> = {
    let mut temp_vec = Vec::new();
    temp_vec.push(1);
    temp_vec.push(2);
    temp_vec.push(3);
    temp_vec
};
```

我们可以使用宏来实现这么一个简写：[1]

```
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
```

```
    };
}
```

哇哦，这里有好多新语法！让我们分开来看。

```
macro_rules! vec { ... }
```

这里我们定义了一个叫做`vec`的宏，跟用`fn vec`定义一个`vec`函数很相似。再罗嗦一句，我们通常写宏的名字时带上一个感叹号，例如`vec!`。感叹号是调用语法的一部分用来区别宏和常规函数。

匹配

宏通过一系列规则定义，它们是模式匹配的分支。上面我们有：

```
( $( $x:expr ),* ) => { ... };
```

这就像一个`match`表达式分支，不过匹配发生在编译时Rust的语法树中。最后一个分支（这里只有一个分支）的分号是可选的。`=>`左侧的“模式”叫匹配器（*matcher*）。它有自己的语法。

`$x:expr`匹配器将会匹配任何Rust表达式，把它的语法树绑定到元变量`$x`上。`expr`标识符是一个片段分类符（*fragment specifier*）。在宏进阶章节（已被本章合并，坐等官方文档更新）中列举了所有可能的分类符。匹配器写在`$(...)`中，`*`会匹配0个或多个表达式，表达式之间用逗号分隔。

除了特殊的匹配器语法，任何出现在匹配器中的Rust标记必须完全相符。例如：

```
macro_rules! foo {
    (x => $e:expr) => (println!("mode X: {}", $e));
    (y => $e:expr) => (println!("mode Y: {}", $e));
}

fn main() {
    foo!(y => 3);
}
```

将会打印：

```
mode Y: 3
```

而这个：

```
foo!(z => 3);
```

我们会得到编译错误：

```
error: no rules expected the token `z`
```

扩展

宏规则的右边是正常的Rust语法，大部分是。不过我们可以拼接一些匹配器中的语法。例如最开始的例子：

```
$(
    temp_vec.push($x);
)*
```

每个匹配的`$x`表达式都会在宏扩展中产生一个单独`push`语句。扩展中的重复与匹配器中的重复“同步”进行（稍后介绍更多）。

因为`$x`已经在表达式匹配中声明了，我们并不在右侧重复`:expr`。另外，我们并不将用来分隔的逗号作为重复操作的一部分。相反，我们在重复块中使用一个结束用的分号。

另一个细节：`vec!`宏的右侧有两对大括号。它们经常像这样结合起来：

```
macro_rules! foo {
    () => {{
        ...
    }}
}
```

外层的大括号是`macro_rules!`语法的一部分。事实上，你也可以`()`或者`[]`。它们只是用来界定整个右侧结构的。

内层大括号是扩展语法的一部分。记住，`vec!`在表达式上下文中使用。要写一个包含多个语句，包括`let`绑定，的表达式，我们需要使用块。如果你的宏只扩展一个单独的表达式，你不需要内层的大括号。

注意我们从未声明宏产生一个表达式。事实上，直到宏被展开之前我们都无法知道。足够小心的话，你可以编写一个能在多个上下文中扩展的宏。例如，一个数据类型的简写可以作为一个表达式或一个模式。

重复（Repetition）

重复运算符遵循两个原则：

1. `$(...)*`对它包含的所有`$name`都执行“一层”重复
2. 每个`$name`必须有至少这么多的`$(...)*`与其相对。如果多了，它将是多余的。

这个巴洛克宏展示了外层重复中多余的变量。

```
macro_rules! o_O {
    (
```

```
        $(
            $x:expr; [ $( $y:expr ),* ]
        );*
    ) => {
        &[ $($( $x + $y ),*),* ]
    }
}

fn main() {
    let a: &[i32]
        = o_O!(10; [1, 2, 3];
               20; [4, 5, 6]);

    assert_eq!(a, [11, 12, 13, 24, 25, 26]);
}
```

这就是匹配器的大部分语法。这些例子使用了`$(...)*`，它指“0次或多次”匹配。另外你可以用`$(...)+`代表“1次或多次”匹配。每种形式都可以包括一个分隔符，分隔符可以使用任何除了`+`和`*`的符号。

这个系统基于Macro-by-Example（PDF链接）。

卫生（**Hygiene**）

一些语言使用简单的文本替换来实现宏，它导致了很多问题。例如，这个C程序打印`13`而不是期望的`25`。

```
#define FIVE_TIMES(x) 5 * x

int main() {
    printf("%d\n", FIVE_TIMES(2 + 3));
    return 0;
}
```

扩展之后我们得到`5 * 2 + 3`，并且乘法比加法有更高的优先级。如果你经常使用C的宏，你可能知道标准的习惯来避免这个问题，或更多其它的问题。在Rust中，你不需要担心这个问题。

```
macro_rules! five_times {
    ($x:expr) => (5 * $x);
}

fn main() {
    assert_eq!(25, five_times!(2 + 3));
}
```

元变量`$x`被解析成一个单独的表达式节点，并且在替换后依旧在语法树中保持原值。

宏系统中另一个常见的问题是变量捕捉（*variable capture*）。这里有一个C的宏，使用了GNU C 扩展来模拟Rust表达式块。

```
#define LOG(msg) ({ \
```

```
    int state = get_log_state(); \
    if (state > 0) { \
        printf("log(%d): %s\n", state, msg); \
    } \
})
```

这是一个非常糟糕的用例：

```
const char *state = "reticulating splines";
LOG(state)
```

它扩展为：

```
const char *state = "reticulating splines";
int state = get_log_state();
if (state > 0) {
    printf("log(%d): %s\n", state, state);
}
```

第二个叫做`state`的参数参数被替换为了第一个。当打印语句需要用到这两个参数时会出现问题。

等价的Rust宏则会有理想的表现：

```
macro_rules! log {
    ($msg:expr) => {{
        let state: i32 = get_log_state();
        if state > 0 {println!("log({}): {}", state, $msg);
        }
    }};
}

fn main() {
    let state: &str = "reticulating splines";
    log!(state);
}
```

这之所以能工作时因为Rust有一个卫生宏系统。每个宏扩展都在一个不同的语法上下文（*syntax context*）中，并且每个变量在引入的时候都在语法上下文中打了标记。这就好像是`main`中的`state`和宏中的`state`被画成了不同的“颜色”，所以它们不会冲突。

这也限制了宏在被执行时引入新绑定的能力。像这样的代码是不能工作的：

```
macro_rules! foo {
    () => (let x = 3);
}

fn main() {
    foo!();
    println!("{}", x);
}
```

相反你需要在执行时传递变量的名字，这样它会在语法上下文中被正确标记。

```
macro_rules! foo {
    ($v:ident) => (let $v = 3);
}

fn main() {
    foo!(x);
    println!("{}", x);
}
```

这对 `let` 绑定和loop标记有效，对items无效。所以下面的代码可以编译：

```
macro_rules! foo {
    () => (fn x() { });
}

fn main() {
    foo!();
    x();
}
```

递归宏

一个宏扩展中可以包含更多的宏，包括被扩展的宏自身。这种宏对处理树形结构输入时很有用的，正如这这个（简化了的）HTML简写所展示的那样：

```
macro_rules! write_html {
    ($w:expr, ) => (());

    ($w:expr, $e:tt) => (write!($w, "{}", $e));

    ($w:expr, $tag:ident [ $($inner:tt)* ] $($rest:tt)*) => {{
        write!($w, "<{}>", stringify!($tag));
        write_html!($w, $($inner)*);
        write!($w, "</{}>", stringify!($tag));
        write_html!($w, $($rest)*);
    }};
}

fn main() {
    use std::fmt::Write;
    let mut out = String::new();

    write_html!(&mut out,
        html[
            head[title["Macros guide"]]
            body[h1["Macros are the best!"]]
        ]);

    assert_eq!(out,
```

```
        "<html><head><title>Macros guide</title></head>\
         <body><h1>Macros are the best!</h1></body></html>");
}
```

调试宏代码

运行 `rustc --pretty expanded` 来查看宏扩展后的结果。输出表现为一个完整的包装箱，所以你可以把它反馈给 `rustc`，它会有时会比原版产生更好的错误信息。注意如果在同一作用域中有多个相同名字（不过在不同的语法上下文中）的变量的话 `--pretty expanded` 的输出可能会有不同的意义。这种情况下 `--pretty expanded,hygiene` 将会告诉你有关语法上下文的信息。

`rustc` 提供两种语法扩展来帮助调试宏。目前为止，它们是不稳定的并且需要功能入口（feature gates）。

- `log_syntax!(...)` 会打印它的参数到标准输出，在编译时，并且不“扩展”任何东西。
- `trace_macros!(true)` 每当一个宏被扩展时会启用一个编译器信息。在扩展后使用 `trace_macros!(false)` 来关闭它。

句法要求

即使Rust代码中含有未扩展的宏，它也可以被解析为一个完整的语法树。这个属性对于编辑器或其它处理代码的工具来说十分有用。这里也有一些关于Rust宏系统设计的推论。

一个推论是Rust必须确定，当它解析一个宏扩展时，宏是否代替了

- 0个或多个项
- 0个或多个方法
- 一个表达式
- 一个语句
- 一个模式

一个块中的宏扩展代表一些项，或者一个表达式/语句。Rust使用一个简单的规则来解决这些二义性。一个代表项的宏扩展必须是

- 用大括号界定的，例如 `foo! { ... }`
- 分号结尾的，例如 `foo!(...);`

另一个展开前解析的推论是宏扩展必须包含有效的Rust记号。更进一步，括号，中括号，大括号在宏扩展中必须是封闭的。例如， `foo!([)` 是不允许的。这让Rust知道宏何时结束。

更正式一点，宏扩展体必须是一个记号树（*token trees*）的序列。一个记号树是一系列递归的

- 一个由 `()`， `[]` 或 `{}` 包围的记号树序列
- 任何其它单个记号

在一个匹配器中，每一个元变量都有一个片段分类符（*fragment specifier*），确定它匹配的哪种句法。

- `ident`：一个标识符。例如：`x`，`foo`
- `path`：一个合适的名字。例如：`T::SpecialA`
- `expr`：一个表达式。例如：`2 + 2`；`if true then { 1 } else { 2 }`；`f(42)`
- `ty`：一个类型。例如：`i32`；`Vec<(char, String)>`；`&T`
- `pat`：一个模式。例如：`Some(t)`；`(17, 'a')`；`_`
- `stmt`：一个单独语句。例如：`let x = 3`
- `block`：一个大括号界定的语句序列。例如：`{ log(error, "hi"); return 12; }`
- `item`：一个项。例如：`fn foo() { }`，`struct Bar`
- `meta`：一个“元项”，可以在属性中找到。例如：`cfg(target_os = "windows")`
- `tt`：一个单独的记号树

对于一个元变量后面的一个记号有一些额外的规则：

- `expr`变量必须后跟一个`=>`，`,`，`;`
- `ty`和`path`变量必须后跟一个`=>`，`,`，`:`，`=`，`>`，`as`
- `pat`变量必须后跟一个`=>`，`,`，`=`
- 其它变量可以后跟任何记号

这些规则为Rust语法提供了一些灵活性以便将来的扩展不会破坏现有的宏。

宏系统完全不处理解析模糊。例如，`$($t:ty)* $e:expr`语法总是会解析失败，因为解析器会被强制在解析`$t`和解析`$e`之间做出选择。改变扩展在它们之前分别加上一个记号可以解决这个问题。在这个例子中，你可以写成`$(T $t:ty)* E $e:exp`。

范围和宏导入/导出

宏在编译的早期阶段被展开，在命名解析之前。这有一个缺点是与语言中其它结构相比，范围对宏的作用不一样。

定义和扩展都发生在同一个深度优先，字典顺序的包装箱的代码遍历中。那么在模块范围内定义的宏对同模块的接下来的代码是可见的，这包括任何接下来的子`mod`项。

一个定义在`fn`函数体内的宏，或者任何其它不在模块范围内的地方，只在它的范围内可见。

如果一个模块有`subsequent`属性，它的宏在子`mod`项之后的父模块也是可见的。如果它的父模块也有`macro_use`属性那么在父`mod`项之后的祖父模块中也是可见的，以此类推。

`macro_use`属性也可以出现在`extern crate`。在这个上下文中它控制那些宏从外部包装箱中装载，例如

```
#[macro_use(foo, bar)]
extern crate baz;
```

如果属性只是简单的写成`#[macro_use]`，所有的宏都会被装载。如果没有`#[macro_use]`属性那么没有宏被装载。只有被定义为`#[macro_export]`的宏可能被装载。

装载一个包装箱的宏而不链接到输出，使用`#[no_link]`。

一个例子：

```
macro_rules! m1 { () => (()) }

// visible here: m1

mod foo {
    // visible here: m1

    #[macro_export]
    macro_rules! m2 { () => (()) }

    // visible here: m1, m2
}

// visible here: m1

macro_rules! m3 { () => (()) }

// visible here: m1, m3

#[macro_use]
mod bar {
    // visible here: m1, m3

    macro_rules! m4 { () => (()) }

    // visible here: m1, m3, m4
}

// visible here: m1, m3, m4
```

当这个库被用`#[macro_use] extern crate`装载时，只有`m2`会被导入。

Rust参考中有一个宏相关的属性列表。

`$crate`变量

当一个宏在多个包装箱中使用时会产生另一个困难。让我们说`mylib`定义了

```
pub fn increment(x: u32) -> u32 {
    x + 1
}

#[macro_export]
macro_rules! inc_a {
    ($x:expr) => ( ::increment($x) )
}

#[macro_export]
macro_rules! inc_b {
    ($x:expr) => ( ::mylib::increment($x) )
}
```

`inc_a`只能在`mylib`内工作，同时`inc_b`只能在库外工作。进一步说，如果用户有另一个名字导入`mylib`时`inc_b`将不能工作。

Rust（目前）还没有针对包装箱引用的卫生系统，不过它确实提供了一个解决这个问题的变通方法。当从一个叫`foo`的包装箱总导入宏时，特殊宏变量`$crate`会展开为`::foo`。相反，当这个宏在同一包装箱内定义和使用时，`$crate`将展开为空。这意味着我们可以写

```
#[macro_export]
macro_rules! inc {
    ($x:expr) => ( $crate::increment($x) )
}
```

来定义一个可以在库内外都能用的宏。这个函数名字会展开为`::increment`或`::mylib::increment`。

为了保证这个系统简单和正确，`#[macro_use] extern crate ...`应只出现在你包装箱的根中，而不是在`mod`中。这保证了`$crate`扩展为一个单独的标识符。

深入（The deep end）

之前的介绍章节提到了递归宏，但并没有给出完整的介绍。还有一个原因令递归宏是有用的：每一次递归都给你匹配宏参数的机会。

作为一个极端的例子，可以，但极端不推荐，用Rust宏系统来实现一个位循环标记自动机。

```
macro_rules! bct {
    // cmd 0:  d ... => ...
    (0, $($ps:tt),* ; $_d:tt)
        => (bct!($($ps),*, 0 ; ));
    (0, $($ps:tt),* ; $_d:tt, $($ds:tt),*)
        => (bct!($($ps),*, 0 ; $($ds),*));

    // cmd 1p:  1 ... => 1 ... p
    (1, $p:tt, $($ps:tt),* ; 1)
        => (bct!($($ps),*, 1, $p ; 1, $p));
    (1, $p:tt, $($ps:tt),* ; 1, $($ds:tt),*)
        => (bct!($($ps),*, 1, $p ; 1, $($ds),*, $p));

    // cmd 1p:  0 ... => 0 ...
    (1, $p:tt, $($ps:tt),* ; $($ds:tt),*)
        => (bct!($($ps),*, 1, $p ; $($ds),*));

    // halt on empty data string
    ( $($ps:tt),* ; )
        => (());
}
```

练习：使用宏来减少上面`bct!`宏定义中的重复。

常用宏（Common macros）

这里有一些你会在Rust代码中看到的常用宏

`panic!`

这个宏导致当前线程恐慌。你可以传给这个宏一个信息通过：

```
panic!("oh no!");
```

`vec!`

`vec!`的应用遍及本书，所以你可能已经见过它了。它方便创建`Vec<T>`：

```
let v = vec![1, 2, 3, 4, 5];
```

它也让你可以用重复值创建vector。例如，100个`0`：

```
let v = vec![0; 100];
```

`assert!`和`assert_eq!`

这两个宏用在测试中。`assert!`获取一个布尔值，而`assert_eq!`获取两个值并比较它们。对了就通过，错了就`panic!`（注：原书是Truth passes, success panic!s，个人认为不对）。像这样：

```
// A-ok!

assert!(true);
assert_eq!(5, 3 + 2);

// nope :(

assert!(5 < 3);
assert_eq!(5, 3);
```

`try!`

`try!`用来进行错误处理。它获取一些可以返回`Result<T, E>`的数据，并返回`T`如果它是`Ok<T>`，或`return`一个`Err(E)`如果出错了。像这样：

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

fn foo() -> std::io::Result<()> {
    let f = try!(File::create("foo.txt"));

    Ok(())
}
```

它比这么写要更简明：

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

fn foo() -> std::io::Result<()> {
    let f = File::create("foo.txt");

    let f = match f {
        Ok(t) => t,
        Err(e) => return Err(e),
    };

    Ok(())
}
```

`unreachable!`

这个宏用于当你认为一些代码不应该被执行的时候：

```
if false {
    unreachable!();
}
```

有时，编译器可能会让你编写一个不同的你认为将永远不会执行的分支。在这个例子中，用这个宏，这样如果你以错误结尾，你会为此得到一个`panic!`。

```
let x: Option<i32> = None;

match x {
    Some(_) => unreachable!(),
    None => println!("I know x is None!"),
}
```

`unimplemented!`

`unimplemented!`宏可以被用来当你尝试去让你的函数通过类型检查，同时你又不想操心去写函数体的时候。一个这种情况的例子是实现一个要求多个方法的特性，而你只想一次搞定一个。用`unimplemented!`定义其它的直到你准备好去写它们了。

宏程序（**Procedural macros**）

如果Rust宏系统不能做你想要的，你可能想要写一个编译器插件。与`macro_rules!`宏相比，它能做更多的事，接口也更不稳定，并且bug将更难以追踪。相反你得到了可以在编译器中运行任意Rust代码的灵活性。为此语法扩展插件有时被称为宏程序（*procedural macros*）。

---

1. 在libcollections中的`vec!`的实际定义与我们在这展示的有所不同，出于效率和可重用性的考虑。

裸指针

Rust的标准库中有一系列不同的智能指针类型，不过这有两个类型是十分特殊的。Rust的安全大多来源于编译时检查，不过裸指针并没有这样的保证，使用它们是`unsafe`的。

`*const T`和`*mut T`在Rust中被称为“裸指针”。有时当编写特定类型的库时，为了某些原因你需要绕过Rust的安全保障。在这种情况下，你可以使用裸指针来实现你的库，同时暴露一个安全的接口给你的用户。例如，`*`指针允许别名，允许用来写共享所有权类型，甚至是内存安全的共享内存类型（`Rc<T>`和`Arc<T>`类型都是完全用Rust实现的）。

这里有一些你需要记住的裸指针不同于其它指针的地方。它们是：

- 不能保证指向有效的内存，甚至不能保证是非空的（不像`Box`和`&`）；
- 没有任何自动清除，不像`Box`，所以需要手动管理资源；
- 是普通旧式类型，也就是说，它不移动所有权，这又不像`Box`，因此Rust编译器不能保证不出像释放后使用这种bug；
- 被认为是可发送的（如果它的内容是可发送的），因此编译器不能提供帮助确保它的使用是线程安全的；例如，你可以从两个线程中并发的访问`*mut i32`而不用同步。
- 缺少任何形式的生命周期，不像`&`，因此编译器不能判断出悬垂指针；
- 除了通过`*const T`直接不允许改变外，没有别名或可变性的保障。

基础

创建一个裸指针是非常安全的：

```
let x = 5;
let raw = &x as *const i32;

let mut y = 10;
let raw_mut = &mut y as *mut i32;
```

然而，解引用它则不行。这个不能工作：

```
let x = 5;
let raw = &x as *const i32;

println!("raw points at {}", *raw);
```

它给出这个错误：

```
error: dereference of unsafe pointer requires unsafe function or block [E0133]
     println!("raw points at{}", *raw);
                                 ^~~~
```

当你解引用一个裸指针，你要为它并不指向正确的地方负责。为此，你需要`unsafe`：

```
let x = 5;
let raw = &x as *const i32;

let points_at = unsafe { *raw };

println!("raw points at {}", points_at);
```

关于裸指针的更多操作，查看它们的API文档。

FFI

裸指针在FFI中很有用：Rust的`*const T`和`*mut T`分别与C中的`const T*`和`T*`类似。关于它们的应用，查看FFI章节。

引用和裸指针

在运行时，指向一份相同数据的裸指针`*`和引用有相同的表现。事实上，在安全代码中`&T`引用会隐式的转换为一个`*const T`同时它们的`mut`变体也有类似的行为（这两种转换都可以显式执行，分别为`value as *const T`和`value as *mut T`）。

反其道而行之，从`*const`到`&`引用，是不安全的。一个`&T`总是有效的，所以，最少，`*const T`裸指针必须指向一个`T`的有效实例。进一步，结果指针必须满足引用的别名和可变性法则。编译器假设这些属性对任何引用都是有效的，不管它们是如何创建的，因而所以任何从裸指针来的转换都断言它们成立。程序员必须保证它。

推荐的转换方法是

```
let i: u32 = 1;

// explicit cast
let p_imm: *const u32 = &i as *const u32;
let mut m: u32 = 2;

// implicit coercion
let p_mut: *mut u32 = &mut m;

unsafe {
    let ref_imm: &u32 = &*p_imm;
    let ref_mut: &mut u32 = &mut *p_mut;
}
```

与使用`transmute`相比更倾向于`&*x`解引用风格。或者比需要的更强大，并且更严格的操作更难以错误使用；例如，它要求`x`是一个指针（不像`transmute`）。

不安全代码

Rust主要魅力是它强大的静态行为保障。不过安全检查天性保守：有些程序实际上是安全的，不过编译器不能验证它是否是真的。为了写这种类型的程序，我们需要告诉编译器稍微放松它的限制。为此，Rust有一个关键字，`unsafe`。使用`unsafe`的代码比正常代码有更少的限制。

让我们过一遍语法，接着我们讨论语义。`unsafe`用在两个上下文中。第一个标记一个函数为不安全的：

```
unsafe fn danger_will_robinson() {
    // scary stuff
}
```

例如所有从FFI调用的函数都必须标记为`unsafe`。第二个`unsafe`的用途是一个不安全块。

```
unsafe {
    // scary stuff
}
```

显式勾勒出那些可能会有bug并造成大问题的代码是很重要的。如果一个Rust程序段错误了，你可以确认它位于标记为`unsafe`部分的什么地方。

“安全”指什么？（What does ‘safe’ mean?）

安全，在Rust的上下文中，意味着“不做任何不安全的事”。简单明了！

好的，让我再试一下：神马是不安全的事？这里是个列表：

- 数据竞争
- 解引用一个空/悬垂裸指针
- 读`undef`（未初始化）内存
- 使用裸指针打破指针混淆规则
- `&mut T`和`&T`遵循LLVM范围的`noalias`模型，除了如果`&T`包含一个`UnsafeCell<U>`的话。不安全代码必须不能违反这些混淆保证
- 不使用`UnsafeCell<U>`改变一个比可变值/引用
- 通过编译器固有功能调用未定义行为：
  - 使用`std::ptr::offset`（`offset`功能）来索引超过一个对象界限的值，除了结尾后一个字节，这是允许的
  - 使用`std::ptr::copy_nonoverlapping_memory`（`memcpy32/memcpy64`功能）来重叠缓冲区
- 原生类型的无效值，即使是在私有字段/本地变量中：
  - 空/悬垂引用或装箱
  - `bool`中一个不是`false`（`0`）或`true`（`1`）的值
  - `enum`中一个并不包含在类型定义中判别式
  - `char`中一个等于或大于`char::MAX`的值
  - `str`中非UTF-8字节序列

- 在外部代码中使用Rust或在Rust中使用外部语言

这里有好多东西！注意到所有有些行为确实是不好的也是很重要的，不过明显不是不安全的：

- 死锁
- 从私有字段读取数据
- 由于引用计数循环导致的泄露
- 退出但未调用析构函数
- 发送信号
- 访问/修改文件系统
- 整形溢出

Rust不能避免所有类型的软件错误。有bug的代码可能并将会出现在Rust中。这些事并不很光彩，不过它们并不特别识别为`unsafe`。

不安全的超级力量（Unsafe Superpowers）

在不安全函数和不安全块，Rust将会让你做3件通常你不能做的事：只有3件。它们是：

1. 访问和更新一个静态可变变量
2. 解引用一个裸指针
3. 调用不安全函数。这是最NB的能力

这就是全部。注意到`unsafe`不能，例如，“关闭借用检查”是很重要的。为随机的Rust代码加上`unsafe`并不会改变它的语义，它并不会开始接受任何东西。

不过确实它会让你写的东西打破一些规则。让我们按顺序过一遍这3个能力。

访问和更新一个静态可变变量

Rust有一个叫`static mut`的功能，它允许改变全局状态。这么做可能造成一个数据竞争，所以它天生是不安全的。关于更多细节，查看静态量部分。

解引用一个裸指针

裸指针让你做任意的指针算数，并会产生一系列不同的内存安全和保密问题。在某种意义上，解引用一个任意指针的能力是你可以做的最危险的事之一。更多关于裸指针，查看它的部分。

调用不安全函数

最后的能力能用于`unsafe`的两个方面：你只能在一个不安全块中调用被标记为`unsafe`的函数。

这个能力是强力和多变的。Rust暴露了一些作为不安全函数的编译器固有功能，并且一些不安全函数绕开了安全检查，用安全换速度。

我在重复一遍：即便你可以在一个不安全块和函数中做任何事并不意味你应该这么做。编译器会表现得像你在保持它不变一样（The compiler will act as though you’re upholding its invariants），所以请小心。

Rust开发版

Rust提供了3种发行渠道：开发版（每日构建），beta版和稳定版。不稳定功能只在Rust开发版中可用。对于这个进程的更多细节，查看作为可支付的稳定性。

要安装Rust开发版，你可以使用`rustup.sh`：

```
$ curl -s https://static.rust-lang.org/rustup.sh | sudo sh -s -- --channel=nightly
```

如果你担心使用`curl | sudo sh`的潜在不安全性，请继续阅读并查看我们下面的免责声明。并且你也可以随意使用下面这个两步安装脚本以便可以检查我们的安装脚本：

```
$ curl -L https://static.rust-lang.org/rustup.sh -O
$ sudo sh rustup.sh
```

如果你用Windows，请直接下载32位或者64位安装包然后运行即可。

卸载

如果不幸的，你再也不想使用Rust了:(，当然这不要紧。也许Rust不是你的菜（原文：不是所有人都会认为什么语言非常好）。运行下面的卸载脚本：

```
$ sudo /usr/local/lib/rustlib/uninstall.sh
```

如果你使用Windows安装包进行安装的话，重新运行`.exe`文件，它会提供一个卸载选项。

你可以在任何时候重新运行脚本来更新Rust。在现在这个时间，你将会频繁更新Rust，因为Rust还未发布1.0版本，经常更新人们会认为你在使用最新版本的Rust。

不过这带来了另外一个问题（传说中的免责声明？）：一些同学确实有理由对我们让他们运行`curl | sudo sh`感到非常反感。他们理应如此。从根本上说，当你运行上面的脚本时，代表你相信是一些好人在维护Rust，他们不会黑了你的电脑做坏事。对此保持警觉是一个很好的天性。如果你是这些强迫症患者（大雾），请检阅以下文档，从源码编译Rust或者官方二进制文件下载。我们保证这将不会一直作为安装Rust的方法：这只是为了方便大家在Alpha（现在是Beta了）时期更新Rust。

当然，我们需要提到官方支持的平台：

- Windows (7, 8, Server 2008 R2)
- Linux (2.6.18 or later, various distributions), x86 and x86-64
- OSX 10.7 (Lion) or greater, x86 and x86-64

Rust在以上平台进行了广泛的测试，当然还在一些其他平台，比如Android。不过进行了越多测试的环境，越有可能正常工作。

最后，关于Windows。Rust将Windows作为第一级平台来发布，不过说实话，WIndows的集成体验并没有Linux/OS X那么好。我们正在改善它！如果有情况它不能工作了，这是一个bug。如果这种发生了，请让我知道。任何一次提交都在Windows下进行了测试，就像其它平台一样。

如果你已安装Rust，你可以打开一个Shell，然后输入：

```
$ rustc --version
```

你应该看到版本号，提交的hash值，提交时间和构建时间：

```
rustc 1.0.0-nightly (f11f3e7ba 2015-01-04) (built 2015-01-06)
```

如果你做到了，那么Rust已经正确安装！此处应有掌声！

如果你遇到什么错误，这里有几个地方你可以获取帮助。最简单的是通过Mibbit访问Rust IRC频道irc.mozilla.org。点击上面的链接，你就可以与其它Rustaceans（简单理解为Ruster吧）聊天，我们会帮助你。其它的地方有the /r/rust subreddit和Stack Overflow。

编译器插件

介绍

`rustc`可以加载编译器插件，它是由用户提供的库用来扩充编译器的行为，例如新的语法扩展，lint检查等。

一个插件是带有设计好的用来在`rustc`中注册扩展的注册（*registrar*）函数的一个动态库包装箱。其它包装箱可以使用`#![plugin(...)]`属性来装载这个扩展。查看rustc::plugin文档来获取更多关于定义和装载插件的机制。

如果属性存在的话，`#![plugin(foo(... args ...))]`传递的参数并不由`rustc`自身解释。它们被传递给插件的`Registry`args方法。

在绝大多数情况中，一个插件应该只通过`#![plugin]`而不通过`extern crate`来使用。链接一个插件会将`libsyntax`和`librustc`加入到你的包装箱的依赖中。基本上你不会希望如此除非你在构建另一个插件。`plugin_as_library`lint会检查这些原则。

通常的做法是将插件放到它们自己的包装箱中，与任何那些会被库的调用者使用的`macro_rules!`宏或Rust代码分开。

语法扩展

插件可以有多种方法来扩展Rust的语法。一种语法扩展是宏过程。它们与普通宏的调用方法一样，不过扩展是通过执行任意Rust代码在编译时操作语法树进行的。

让我们写一个实现了罗马数字的插件roman_numerals.rs。

```
#![crate_type="dylib"]
#![feature(plugin_registrar, rustc_private)]

extern crate syntax;
extern crate rustc;

use syntax::codemap::Span;
use syntax::parse::token;
use syntax::ast::{TokenTree, TtToken};
use syntax::ext::base::{ExtCtxt, MacResult, DummyResult, MacEager};
use syntax::ext::build::AstBuilder;  // trait for expr_usize
use rustc::plugin::Registry;

fn expand_rn(cx: &mut ExtCtxt, sp: Span, args: &[TokenTree])
        -> Box<MacResult + 'static> {

    static NUMERALS: &'static [(&'static str, u32)] = &[
        ("M", 1000), ("CM", 900), ("D", 500), ("CD", 400),
        ("C",  100), ("XC",  90), ("L",  50), ("XL",  40),
        ("X",   10), ("IX",   9), ("V",   5), ("IV",   4),
        ("I",    1)];
```

```
    let text = match args {
        [TtToken(_, token::Ident(s, _))] => token::get_ident(s).to_string(),
        _ => {
            cx.span_err(sp, "argument should be a single identifier");
            return DummyResult::any(sp);
        }
    };

    let mut text = &*text;
    let mut total = 0;
    while !text.is_empty() {
        match NUMERALS.iter().find(|&&(rn, _)| text.starts_with(rn)) {
            Some(&(rn, val)) => {
                total += val;
                text = &text[rn.len()..];
            }
            None => {
                cx.span_err(sp, "invalid Roman numeral");
                return DummyResult::any(sp);
            }
        }
    }

    MacEager::expr(cx.expr_u32(sp, total))
}

#[plugin_registrar]
pub fn plugin_registrar(reg: &mut Registry) {
    reg.register_macro("rn", expand_rn);
}
```

我们可以像其它宏那样使用`rn!()`：

```
#![feature(plugin)]
#![plugin(roman_numerals)]

fn main() {
    assert_eq!(rn!(MMXV), 2015);
}
```

与一个简单的`fn(&str) -> u32`函数相比的优势有：

- （任意复杂程度的）转换都发生在编译时
- 输入验证也在编译时进行
- 可以扩展并允许在模式中使用，它可以有效的为任何数据类型定义新语法。

除了宏过程，你可以定义新的类derive属性和其它类型的扩展。查看Registry::register_syntax_extension和SyntaxExtension enum。对于更复杂的宏例子，查看regex_macros。

提示与技巧

这里提供一些宏调试的提示。

你可以使用syntax::parse来将记号树转换为像表达式这样的更高级的语法元素：

```
fn expand_foo(cx: &mut ExtCtxt, sp: Span, args: &[TokenTree])
        -> Box<MacResult+'static> {

    let mut parser = cx.new_parser_from_tts(args);

    let expr: P<Expr> = parser.parse_expr();
```

看完libsyntax解析器代码会给你一个解析基础设施如何工作的感觉。

保留你解析所有的Span，以便更好的报告错误。你可以用Spanned包围你的自定数据结构。

调用ExtCtxt::span_fatal将会立即终止编译。相反最好调用ExtCtxt::span_err并返回DummyResult，这样编译器可以继续并找到更多错误。

为了打印用于调试的语法段，你可以同时使用span_note和syntax::print::pprust::*_to_string。

上面的例子使用AstBuilder::expr_usize产生了一个普通整数。作为一个`AstBuilder`特性的额外选择，`libsyntax`提供了一个准引用宏的集合。它们并没有文档并且非常边缘化。然而，这些将会是实现一个作为一个普通插件库的改进准引用的好的出发点。

Lint插件

插件可以扩展Rust Lint基础设施来添加额外的代码风格，安全检查等。你可以查看src/test/auxiliary/lint_plugin_test.rs来了解一个完整的例子，我们在这里重现它的核心部分：

```
declare_lint!(TEST_LINT, Warn,
              "Warn about items named 'lintme'")

struct Pass;

impl LintPass for Pass {
    fn get_lints(&self) -> LintArray {
        lint_array!(TEST_LINT)
    }

    fn check_item(&mut self, cx: &Context, it: &ast::Item) {
        let name = token::get_ident(it.ident);
        if name.get() == "lintme" {
            cx.span_lint(TEST_LINT, it.span, "item is named 'lintme'");
        }
    }
}

#[plugin_registrar]
pub fn plugin_registrar(reg: &mut Registry) {
    reg.register_lint_pass(box Pass as LintPassObject);
}
```

那么像这样的代码：

```
#![plugin(lint_plugin_test)]

fn lintme() { }
```

将产生一个编译警告：

```
foo.rs:4:1: 4:16 warning: item is named 'lintme', #[warn(test_lint)] on by default
foo.rs:4 fn lintme() { }
         ^~~~~~~~~~~~~~~
```

Lint插件的组件有：

- 一个或多个`declare_lint!`调用，它定义了Lint结构
- 一个用来存放lint检查所需的所有状态（在我们的例子中，没有）
- 一个定义了如何检查每个语法元素的LintPass实现。一个单独的`LintPass`可能会对多个不同的`Lint`调用`span_lint`，不过它们都需要用`get_lints`方法进行注册。

Lint过程是语法遍历，不过它们运行在编译的晚期，这时类型信息时可用的。`rustc`的内建lint与lint插件使用相同的基础构架，并提供了如何访问类型信息的例子。

由插件定义的语法通常通过属性和插件标识控制，例如，`[#[allow(test_lint)]]`，`-A test-lint`。这些标识符来自于`declare_lint!`的第一个参数，经过合适的大小写和标点转换。

你可以运行`rustc -W help foo.rs`来见检查lint列表是否为`rustc`所知，包括由`foo.rs`加载的插件。

内联汇编

为了极端底层操作和性能要求，你可能希望直接控制CPU。Rust通过 `asm!` 宏来支持使用内联汇编。语法大体上与GCC和Clang相似：

```
asm!(assembly template
   : output operands
   : input operands
   : clobbers
   : options
   );
```

任何 `asm` 的使用需要功能通道（需要在包装箱上加上 `#![feature(asm)]` 来允许使用）并且当然也需要写在 `unsafe` 块中

> 注意：这里的例子使用了x86/x86-64汇编，不过所有平台都受支持。

汇编模板

`assembly template` 是唯一需要的参数并且必须是原始字符串（就是 `""` ）

```
#![feature(asm)]

#[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
fn foo() {
    unsafe {
        asm!("NOP");
    }
}

// other platforms
#[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))]
fn foo() { /* ... */ }

fn main() {
    // ...
    foo();
    // ...
}
```

（ `feature(asm)` 和 `#[cfg]` 从现在开始将被忽略。）

输出操作数，输入操作数，覆盖和选项都是可选的,然而如果你要省略它们的话，你必选加上正确数量的 `:` ：

```
asm!("xor %eax, %eax"
    :
    :
```

```
     : "eax"
    );
```

有空格在中间也没关系：

```
asm!("xor %eax, %eax" ::: "eax");
```

操作数

输入和输出操作数都有相同的格式：`: "constraints1"(expr1), "constraints2"(expr2), ..."`。输出操作数表达式必须是可变的左值，或还未赋值的：

```
fn add(a: i32, b: i32) -> i32 {
    let mut c = 0;
    unsafe {
        asm!("add $2, $0"
             : "=r"(c)
             : "0"(a), "r"(b)
             );
    }
    c
}

fn main() {
    assert_eq!(add(3, 14159), 14162)
}
```

如果你想在这里使用真正的操作数，然而，要求你在你想使用的寄存器上套上大括号`{}`，并且要求你指明操作数的大小。这在非常底层的编程中是很有用的，这时你使用哪个寄存器是很重要的：

```
let result: u8;
asm!("in %dx, %al" : "={al}"(result) : "{dx}"(port));
result
```

覆盖（Clobbers）

一些指令修改的寄存器可能保存有不同的值，所以我们使用覆盖列表来告诉编译器不要假设任何装载在这些寄存器的值是有效的。

```
// Put the value 0x200 in eax
asm!("mov $$0x200, %eax" : /* no outputs */ : /* no inputs */ : "eax");
```

输入和输出寄存器并不需要列出因为这些信息已经通过给出的限制沟通过了。因此，任何其它的被使用的寄存器应该隐式或显式的被列出。

如果汇编修改了代码状态寄存器`cc`则需要在覆盖中被列出，如果汇编修改了内存，`memory`也应被指定。

选项（Options）

最后一部分，`options`是Rust特有的。格式是逗号分隔的基本字符串（也就是说，`:"foo", "bar", "baz"`）。它被用来指定关于内联汇编的额外信息：

目前有效的选项有：

1. *volatile* - 相当于gcc/clang中的`__asm__ __volatile__ (...)`
2. *alignstack* - 特定的指令需要栈按特定方式对齐（比如，SSE）并且指定这个告诉编译器插入通常的栈对齐代码
3. *intel* - 使用intel语法而不是默认的AT&T语法

```
let result: i32;
unsafe {
   asm!("mov eax, 2" : "={eax}"(result) : : : "intel")
}
println!("eax is currently {}", result);
```

不使用标准库

`std`默认被链接到每个Rust包装箱中。在一些情况下，这是不合适的，并且可以通过在包装箱上加入`#![no_std]`属性来避免这一点。

```
// a minimal library
#![crate_type="lib"]
#![feature(no_std)]
#![no_std]
```

很明显不光库可以使用这一点：你可以在可执行文件上使用`#[no_std]`，控制程序入口点有两种可能的方式：`#[start]`属性，或者用你自己的去替换C语言默认的`main`函数。

被标记为`#[start]`的函数传递的参数格式与C一致：

```
#![feature(lang_items, start, no_std)]
#![no_std]

// Pull in the system libc library for what crt0.o likely requires
extern crate libc;

// Entry point for this program
#[start]
fn start(_argc: isize, _argv: *const *const u8) -> isize {
    0
}

// These functions and traits are used by the compiler, but not
// for a bare-bones hello world. These are normally
// provided by libstd.
#[lang = "stack_exhausted"] extern fn stack_exhausted() {}
#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "panic_fmt"] fn panic_fmt() -> ! { loop {} }
```

要覆盖编译器插入的`main`函数，你必须使用`#![no_main]`并通过正确的ABI和正确的名字来创建合适的函数，这也需要需要覆盖编译器的命名改编：

```
#![feature(no_std)]
#![no_std]
#![no_main]
#![feature(lang_items, start)]

extern crate libc;

#[no_mangle] // ensure that this symbol is called `main` in the output
pub extern fn main(argc: i32, argv: *const *const u8) -> i32 {
    0
}

#[lang = "stack_exhausted"] extern fn stack_exhausted() {}
```

```
#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "panic_fmt"] fn panic_fmt() -> ! { loop {} }
```

目前编译器对能够被可执行文件调用的符号做了一些假设。正常情况下，这些函数是由标准库提供的，不过没有它你就必须定义你自己的了。

这三个函数中的第一个`stack_exhausted`，当检测到栈溢出时被调用。这个函数对于如何被调用和应该干什么有一些限制，不顾如果栈限制寄存器没有被维护则一个线程可以有"无限的栈"，这种情况下这个函数不应该被触发。

第二个函数，`eh_personality`，被编译器的错误机制使用。它通常映射到GCC的特性函数上（查看libstd实现来获取更多信息），不过对于不会触发恐慌的包装箱可以确定这个函数不会被调用。最后一个函数，`panic_fmt`，也被编译器的错误机制使用。

使用libcore

> 注意：核心库的结构是不稳定的，建议在任何可能的情况下使用标准库。

通过上面的计数，我们构造了一个少见的运行Rust代码的可执行程序。标准库提供了很多功能，然而，这是Rust的生产力所需要的。如果标准库是不足的话，那么可以使用被设计为标准库替代的libcore。

核心库只有很少的依赖并且比标准库可移植性更强。另外，核心库包含编写符合习惯和高效Rust代码的大部分功能。

例如，下面是一个计算由C提供的两个向量的数量积的函数，使用常见的Rust实现。

```
#![feature(lang_items, start, no_std, core, libc)]
#![no_std]

extern crate core;

use core::prelude::*;

use core::mem;

#[no_mangle]
pub extern fn dot_product(a: *const u32, a_len: u32,
                          b: *const u32, b_len: u32) -> u32 {
    use core::raw::Slice;

    // Convert the provided arrays into Rust slices.
    // The core::raw module guarantees that the Slice
    // structure has the same memory layout as a &[T]
    // slice.
    //
    // This is an unsafe operation because the compiler
    // cannot tell the pointers are valid.
    let (a_slice, b_slice): (&[u32], &[u32]) = unsafe {
        mem::transmute((
            Slice { data: a, len: a_len as usize },
            Slice { data: b, len: b_len as usize },
        ))
```

```
    };

    // Iterate over the slices, collecting the result
    let mut ret = 0;
    for (i, j) in a_slice.iter().zip(b_slice.iter()) {
        ret += (*i) * (*j);
    }
    return ret;
}

#[lang = "panic_fmt"]
extern fn panic_fmt(args: &core::fmt::Arguments,
                    file: &str,
                    line: u32) -> ! {
    loop {}
}

#[lang = "stack_exhausted"] extern fn stack_exhausted() {}
#[lang = "eh_personality"] extern fn eh_personality() {}
```

注意这里有一个额外的`lang`项与之前的例子不同，`panic_fmt`。它必须由libcore的调用者定义因为核心库声明了恐慌，但没有定义它。`panic_fmt`项是这个包装箱的恐慌定义，并且它必须确保不会返回。

正如你在例子中所看到的，核心库尝试在所有情况下提供Rust的功能，不管平台的要求如何。另外一些库，例如`liballoc`，为libcore增加了进行其它平台相关假设的功能，不过这依旧比标准库更有可移植性。

固有功能

> 注意：固有功能将会永远是一个不稳定的接口，推荐使用稳定的libcore接口而不是直接使用编译器自带的功能。

可以像FFI函数那样导入它们，使用特殊的`rust-intrinsic` ABI。例如，如果在一个独立的上下文，但是想要能在类型间`transmute`，并想进行高效的指针计算，你可以声明函数：

```
extern "rust-intrinsic" {
    fn transmute<T, U>(x: T) -> U;

    fn offset<T>(dst: *const T, offset: isize) -> *const T;
}
```

跟其它FFI函数一样，它们总是`unsafe`的。

语言项

> 注意：语言项通常由Rust发行版的包装箱提供，并且它自身有一个不稳定的接口。建议使用官方发布的包装箱而不是定义自己的版本。

`rustc`编译器有一些可插入的操作，也就是说，功能不是硬编码进语言的，而是在库中实现的，通过一个特殊的标记告诉编译器它存在。这个标记是`#[lang="..."]`属性并且有不同的值`...`,也就是不同的“语言项”。

例如，`Box`指针需要两个语言项，一个用于分配，一个用于释放。下面是一个独立的程序使用`Box`语法糖进行动态分配，通过`malloc`和`free`：

```
#![feature(lang_items, box_syntax, start, no_std, libc)]
#![no_std]

extern crate libc;

extern {
    fn abort() -> !;
}

#[lang = "owned_box"]
pub struct Box<T>(*mut T);

#[lang="exchange_malloc"]
unsafe fn allocate(size: usize, _align: usize) -> *mut u8 {
    let p = libc::malloc(size as libc::size_t) as *mut u8;

    // malloc failed
    if p as usize == 0 {
        abort();
    }

    p
}
#[lang="exchange_free"]
unsafe fn deallocate(ptr: *mut u8, _size: usize, _align: usize) {
    libc::free(ptr as *mut libc::c_void)
}

#[start]
fn main(argc: isize, argv: *const *const u8) -> isize {
    let x = box 1;

    0
}

#[lang = "stack_exhausted"] extern fn stack_exhausted() {}
#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "panic_fmt"] fn panic_fmt() -> ! { loop {} }
```

注意`abort`的使用：`exchange_malloc`语言项假设返回一个有效的指针，所以需要在内部进行检查。

其它语言项提供的功能包括：

- 通过特性重载运算符：`==`，`<`，解引用（`*`）和`+`等运算符对应的特性都有语言项标记；上面4个分别为`eq`，`ord`，`deref`和`add`
- 栈展开和一般故障：`eh_personality`，`fail`和`fail_bounds_checks`语言项
- `std::marker`中用来标明不同类型的特性：`send`，`sync`和`copy`。
- `std::marker`中的标记类型和变化指示器：`covariant_type`和`contravariant_lifetime`等

语言项由编译器延时加载；例如，如果你从未用过`Box`则就没有必要定义`exchange_malloc`和`exchange_free`的函数。`rustc`在一个项被需要而无法在当前包装箱或任何依赖中找到时生成一个错误。

链接参数

这里还有一个方法来告诉rustc如何自定义链接，这就是通过`link_args`属性。这个属性作用于`extern`块并指定当产生构件时需要传递给连接器的原始标记。一个用例将是：

```
#![feature(link_args)]

#[link_args = "-foo -bar -baz"]
extern {}
```

注意现在这个功能隐藏在`feature(link_args)`通道之后因为它并不是一个被认可的执行链接的方法。目前rustc从shell调用系统的连接器，所以使用额外的命令行参数是可行的，不过这并一定永远可行。将来rustc可能使用LLVM直接链接原生库这样一来`link_args`就毫无意义了。

强烈建议你不要使用这个属性，而是使用一个更正式的`[link(...)]`属性作用于`extern`块。

基准测试

Rust也支持基准测试，它可以测试代码的性能。让我们把`src/lib.rs`修改成这样（省略注释）：

```
#![feature(test)]

extern crate test;

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

#[cfg(test)]
mod test {
    use super::*;
    use test::Bencher;

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

    #[bench]
    fn bench_add_two(b: &mut Bencher) {
        b.iter(|| add_two(2));
    }
}
```

注意`test`功能通道，它启用了这个不稳定功能。

我们导入了`test`包装箱，它包含了对基准测试的支持。我们也定义了一个新函数，带有`bench`属性。与一般的不带参数的测试不同，基准测试有一个`&mut Bencher`参数。`Bencher`提供了一个`iter`方法，它接收一个闭包。这个闭包包含我们想要测试的代码。

我们可以用`cargo bench`来运行基准测试：

```
$ cargo bench
   Compiling adder v0.0.1 (file:///home/steve/tmp/adder)
     Running target/release/adder-91b3e234d4ed382a

running 2 tests
test tests::it_works ... ignored
test tests::bench_add_two ... bench:         1 ns/iter (+/- 0)

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

我们的非基准测试将被忽略。你也许会发现`cargo bench`比`cargo test`花费的时间更长。这是因为Rust会多次运行我们的基准测试，然后取得平均值。因为我们的函数只做了非常少的操作，我们耗费了`1 ns/iter (+/- 0)`，不过运行时间更长的测试就会有出现偏差。

编写基准测试的建议：

- 把初始代码放于`iter`循环之外，只把你需要测试的部分放入它
- 确保每次循环都做了“同样的事情”，不要累加或者改变状态
- 确保`iter`循环内简短而快速，这样基准测试会运行的很快同时校准器可以在合适的分辨率上调整运转周期
- 确保`iter`循环执行简单的工作，这样可以帮助我们准确的定位性能优化（或不足）

Gocha：优化

写基准测试有另一些比较微妙的地方：开启了优化编译的基准测试可能被优化器戏剧性的修改导致它不再是我们期望的基准测试了。举例来说，编译器可能认为一些计算并无外部影响并且整个移除它们。

```
#![feature(test)]

extern crate test;
use test::Bencher;

#[bench]
fn bench_xor_1000_ints(b: &mut Bencher) {
    b.iter(|| {
        (0..1000).fold(0, |old, new| old ^ new);
    });
}
```

得到如下结果：

```
running 1 test
test bench_xor_1000_ints ... bench:         0 ns/iter (+/- 0)

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

基准测试运行器提供两种方法来避免这个问题：要么传递给`iter`的闭包可以返回一个随机的值这样强制优化器认为结果有用并确保它不会移除整个计算部分。这可以通过修改上面例子中的`b.iter`调用：

```
b.iter(|| {
    // note lack of `;` (could also use an explicit `return`).
    (0..1000).fold(0, |old, new| old ^ new)
});
```

要么，另一个选择是调用通用的`test::black_box`函数，它会传递给优化器一个不透明的“黑盒”这样强制它考虑任何它接收到的参数。

```
#![feature(test)]

extern crate test;
```

```
b.iter(|| {
    let n = test::black_box(1000);

    (0..n).fold(0, |a, b| a ^ b)
})
```

上述两种方法均未读取或修改值，并且对于小的值来说非常廉价。对于大的只可以通过间接传递来减小额外开销（例如：`black_box(&huge_struct)`）。

执行上面任何一种修改可以获得如下基准测试结果：

```
running 1 test
test bench_xor_1000_ints ... bench:       131 ns/iter (+/- 3)

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

然而，即使使用了上述方法优化器还是可能在不合适的情况下修改测试用例。

装箱语法和模式

目前唯一稳定的创建 `Box` 的方法是通过 `Box::new` 方法。并且不可能在一个模式匹配中稳定的析构一个 `Box` 。不稳定的 `box` 关键字可以用来创建和析构 `Box` 。下面是一个用例：

```
#![feature(box_syntax, box_patterns)]

fn main() {
    let b = Some(box 5);
    match b {
        Some(box n) if n < 0 => {
            println!("Box contains negative number {}", n);
        },
        Some(box n) if n >= 0 => {
            println!("Box contains non-negative number {}", n);
        },
        None => {
            println!("No box");
        },
        _ => unreachable!()
    }
}
```

注意这些功能目前隐藏在 `box_syntax` （装箱创建）和 `box_patterns` （析构和模式匹配）通道之中因为它的语法在未来可能会改变。

返回指针

在很多有指针的语言中，你的函数可以返回一个指针来避免拷贝大的数据结构。例如：

```
struct BigStruct {
    one: i32,
    two: i32,
    // etc
    one_hundred: i32,
}

fn foo(x: Box<BigStruct>) -> Box<BigStruct> {
    Box::new(*x)
}

fn main() {
    let x = Box::new(BigStruct {
        one: 1,
        two: 2,
        one_hundred: 100,
    });

    let y = foo(x);
}
```

要点是通过传递一个装箱，你只需拷贝了一个指针，而不是那构成了`BigStruct`的一百个`int`值。

上面是Rust中的一个反模式。相反，这样写：

```
#![feature(box_syntax)]

struct BigStruct {
    one: i32,
    two: i32,
    // etc
    one_hundred: i32,
}

fn foo(x: Box<BigStruct>) -> BigStruct {
    *x
}

fn main() {
    let x = Box::new(BigStruct {
        one: 1,
        two: 2,
        one_hundred: 100,
    });

    let y: Box<BigStruct> = box foo(x);
}
```

这在不牺牲性能的前提下获得了灵活性。

你可能会认为这会给我们带来很差的性能：返回一个值然后马上把它装箱？难道这在哪里不都是最糟的吗？Rust显得更聪明。这里并没有拷贝。`main`为装箱分配了足够的空间，向`foo`传递一个指向他内存的`x`，然后`foo`直接向`Box<T>`中写入数据。

因为这很重要所以要说两遍：返回指针会阻止编译器优化你的代码。允许调用函数选择它们需要如何使用你的输出。

切片模式

如果你想在一个切片或数组上匹配，你可以通过`slice_patterns`功能使用`&`：

```
#![feature(slice_patterns)]

fn main() {
    let v = vec!["match_this", "1"];

    match &v[..] {
        ["match_this", second] => println!("The second element is {}", second),
        _ => {},
    }
}
```

`advanced_slice_patterns`通道让你使用`..`表明在一个切片的模式匹配中任意数量的元素。这个通配符对一个给定的数组只能只用一次。如果在`..`之前有一个标识符，结果会被绑定到那个名字上。例如：

```
#![feature(advanced_slice_patterns, slice_patterns)]

fn is_symmetric(list: &[u32]) -> bool {
    match list {
        [] | [_] => true,
        [x, inside.., y] if x == y => is_symmetric(inside),
        _ => false
    }
}

fn main() {
    let sym = &[0, 1, 4, 2, 4, 1, 0];
    assert!(is_symmetric(sym));

    let not_sym = &[0, 1, 7, 2, 4, 1, 0];
    assert!(!is_symmetric(not_sym));
}
```

关联常量

通过 `associated_consts` 功能，你像这样可以定义常量：

```
#![feature(associated_consts)]

trait Foo {
    const ID: i32;
}

impl Foo for i32 {
    const ID: i32 = 1;
}

fn main() {
    assert_eq!(1, i32::ID);
}
```

任何 `Foo` 的定义都必须定义 `ID`，不定义的话：

```
#![feature(associated_consts)]

trait Foo {
    const ID: i32;
}

impl Foo for i32 {
}
```

会给出

```
error: not all trait items implemented, missing: `ID` [E0046]
     impl Foo for i32 {
     }
```

实现也可以定义一个默认值：

```
#![feature(associated_consts)]

trait Foo {
    const ID: i32 = 1;
}

impl Foo for i32 {
}

impl Foo for i64 {
    const ID: i32 = 5;
}
```

```
fn main() {
    assert_eq!(1, i32::ID);
    assert_eq!(5, i64::ID);
}
```

如你所见，当实现`Foo`时，你可以不实现它（关联常量），当作为`i32`时。接着它将会使用默认值。不过，作为`i64`时，我们可以增加我们自己的定义。

关联常量并不一定要关联在一个特性上。一个`struct`的`impl`也行：

```
#![feature(associated_consts)]

struct Foo;

impl Foo {
    pub const FOO: u32 = 3;
}
```

词汇表

不是每位Rustacean都是系统编程或计算机科学背景的，所以我们加上了可能难以理解的词汇解释。

数量（**Arity**）

Arity代表函数或操作所需的参数数量。

```
let x = (2, 3);
let y = (4, 6);
let z = (8, 2, 6);
```

在上面的例子中`x`和`y`的Arity是`2`，`z`的Arity是`3`。

抽象语法树（**Abstract Syntax Tree**）

当一个编译器编译你程序的时候，它做了很多不同的事。其中之一就是将你程序中的文本转换为一个‘抽象语法树’，或者‘AST’。这个树是你程序结构的表现。例如，`2 + 3`可以转换为一个树：

```
  +
 / \
2   3
```

而`2 + (3 * 4)`看起来像这样：

```
  +
 / \
2   *
   / \
  3   4
```

学院派研究

一个曾影响过Rust的论文的不完整列表。

推荐进一步了解Rust背景和激发灵感阅读。

> （注：以下翻译属个人理解，勿作为参考）

类型系统

- Cyclone语言中基于区域的内存管理（Region based memory management in Cyclone）
- Cyclone语言中的手动安全内存管理（Safe manual memory management in Cyclone）
- 类型类：使临时多态不再临时（Typeclasses: making ad-hoc polymorphism less ad hoc）
- 宏综述（Macros that work together）
- 特性：组合类型的行为（Traits: composable units of behavior）
- 消除别名（Alias burying） - 我们尝试了一些相似的内容并放弃了它
- 外部唯一性是足够的（External uniqueness is unique enough）
- 用于安全并行的唯一性和引用不可变性（Uniqueness and Reference Immutability for Safe Parallelism）
- 基于区域的内存管理（Region Based Memory Management）

并发

- Singularity：软件栈的重新思考（Singularity: rethinking the software stack）
- Singularity操作系统中支持快速和可靠的消息传递的语言（Language support for fast and reliable message passing in singularity OS）
- 通过work stealing来安排多线程计算（Scheduling multithreaded computations by work stealing）
- 多道程序多处理器的线程调度（Thread scheduling for multiprogramming multiprocessors）
- work stealing中的数据局部性（The data locality of work stealing）
- 动态环形work stealing双端队列（Dynamic circular work stealing deque） - Chase/Lev双端队列
- 异步-完成并行的work优先和help优先的调度策略（Work-first and help-first scheduling policies for async-finish task parallelism） - 比严格的work stealing更宽泛
- 一个Java的fork/join灾难（A Java fork/join calamity） - 对Java fork/join库的批判，特别是其在非严格计算时的work stealing实现
- 并发系统的调度技巧（Scheduling techniques for concurrent systems）
- 竞争启发调度（Contention aware scheduling）
- 时间共享多核系统的平衡work stealing（Balanced work stealing for time-sharing multicores）
- 三层蛋糕？（Three layer cake）
- 非阻塞半work stealing队列（Non-blocking steal-half work queues）
- Reagents：表现和编写细粒度的并发（Reagents: expressing and composing fine-grained concurrency）
- 用于共享内存多处理器的可扩展同步性的算法（Algorithms for scalable synchronization of shared-memory multiprocessors）

其它

- 只能崩溃的软件（Crash-only software）
- 编写高性能内存分配器（Composing High-Performance Memory Allocators）
- 对手动内存分配的思考（Reconsidering Custom Memory Allocation）

关于**Rust**的论文

- Rust中的GPU编程（GPU programming in Rust）
- 并行闭包：一个基于老观点的新做法（Parallel closures: a new twist on an old idea） - 并不完全关于Rust，不过是Nicholas D. Matsakis写的

勘误

> 这里收集了一些比较有争议的名词翻译，并在此统一说明。如果你在阅读时遇到无法理解地方，请提issue。

向量（**vector**）

鉴于将`vector`翻译为`向量`容易引起误解，故决定不再对其进行翻译，如果你在本书中看到“向量”一词，这一定是还未修改过来，请自行脑补为`vector`

片段（**slice**）

`slice`译为`切片`，而不是`片段`

特性（**trait**）

`trait`不做翻译

特性对象（**trait object**）

`trait object`译为`trait对象`