Rust语言中文版

极客学院出版

前言

Rust 是 Mozilla 开发的注重安全、性能和并发性的编程语言。Rust 是针对多核体系提出的语言，并且吸收一些其他动态语言的重要特性，比如不需要管理内存，比如不会出现 Null 指针等。本书是官方文档的中文翻译版，内容不仅涉及入门级的基础知识点，也涉及 Rust 语言的高级用法，帮助读者了解这门系统编程语言。

适用人群

Rust 语言吸收了当下流行开发语言的优点，开发的目的是替代 C++ 语言，本书适合使用 C++ 语言的的开发者。

学习前提

学习本书前，你需要对编程思想有一定的了解，如果你掌握 C++ 或其他编程语言，那么本书将会帮助你对未来语言设计发展的趋势有更深入的了解。

翻译地址：https://doc.rust-lang.org/stable/book/README.html

版本信息

文中应用代码版本信息如下：

语言	版本信息
Rust	1.2.0

目录

1

介绍

Rust 编程语言

欢迎学习本教程！本教程将教你如何使用 Rust 编程语言。Rust 是一门强调安全、性能和并发性的系统编程语言。它为了达到这几个目的，甚至没有一个垃圾收集器。这也使 Rust 能够应用到其他语言做不到的地方：嵌入到其他语言，有指定空间和时间需求的程序，写底层代码（如设备驱动程序和操作系统）。针对当前的其他编程语言，Rust 做到了没有运行时(Runtime)，没有数据竞争。 Rust 也致力于实现“零成本抽象”，尽管这些抽象给人的感觉像一个高级的语言。即使是这样，Rust 仍然可以做到像一个低级的语言那样的精确控制。

“Rust 编程语言”分为七个部分。本文的简介是第一个。在这之后:

- 新手入门 – 设置您的电脑来进行 Rust 开发。
- 学习 Rust – 通过小型项目学习 Rust 编程。
- 高效的 Rust – 学习编写优秀 Rust 代码的一些高级概念。
- 语法和语义 – Rust 的每一部分，分解成小块来讲解。
- 每日 Rust – 尚未构建稳定的一些高端特性。
- 术语 – 本教程的相关参考科目。
- 学术研究 – 影响 Rust 的一些著作。

阅读本文之后，你会想了解“学习 Rust”或“语法和语义”，根据你的喜好：如果你想尝试一个项目，可以学习“学习 Rust”章节。丰富的交叉联合使这些部分连接到一起。

贡献

本教程的源文件可以在 Github 上找到： github.com/rust-lang/rust/tree/master/src/doc/trpl

Rust 的一个简单介绍

Rust 是你可能会感兴趣的一门语言么？让我们先来看看一些能展示其一些优势的小代码示例。

让 Rust 变得独特唯一的一个主要的概念是名称为“所有权”的概念。思考下面这个小例子：

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

这个程序有一个变量绑定名称为 x。此绑定的值是一个 `Vec<T>`，是我们通过在标准库中定义的宏创建的一个'向量'。这个宏被称为 Vec，我们利用！调用宏用。这遵循 Rust 的一般原则：做事情要明确。宏可以有效的做一些比函数调用更复杂的东西，所以他们外观上来看是不一样的。这个符号！也有助于解析，使代码更容易编写，这也很重要。

我们使用 mut 使 x 可变：默认情况下，Rust 中的绑定是不可变的。我们将在后面的例子中改变此向量。

另外值得一提的是，在这里我们不需要标注类型：Rust 是静态类型，我们并不需要再明确标注类型。Rust 有类型推断，用以平衡强大的静态类型和冗长标注类型。

Rust 更倾向于堆栈分配而不是堆分配：x 被直接放置在堆栈中。然而，`Vec<T>` 类型是在堆上分配的向量元素的空间。如果你不熟悉它们之间的区别，那么你现在可以先忽略它，或者你可以查看章节“堆栈和堆”，来了解详细了解。作为一个系统编程语言，Rust 赋予了你如何分配内存空间的能力，但是在我们开始的阶段，这是并不是一个大问题。

此前，我们提到的“所有权”是铁锈的关键新概念。生锈的说法，X 被说成“自己”的载体。这意味着，当 x 超出范围，载体的存储器将被解除分配。这是由防锈编译确定性完成，而不是通过一个机制，诸如垃圾收集器。换句话说，在防锈，你不叫喜欢的 malloc 函数和释放自己：编译静态判断，当你需要分配或释放内存，并插入这些调用本身。犯错是做人，但编译器永远不会忘记。

前面我们所提到的,“所有权”是 Rust 中的一个非常重要的新概念。按照 Rust 的说法，x 被称为向量“所有”。这意味着当 x 超出范围,向量的内存将被销毁。这样做是由 Rust 的编译器所决定的,而不是通过一种机制（如垃圾收集器）所决定的。换句话说，在 Rust 中,你不需要自己调用函数，如 `malloc` 和 `free yourself:` 当你需要分配或释放的内存时，编译器会自行静态的决定并插入这些调用函数。犯错是人之常情，但编译器永远不会忘记插入这些调用的函数。

让我们在我们上面的例子中添加另外的一行代码：

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

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

我们在此介绍了另外的一种绑定 ,y 。在这种情况下 ,y 是向量第一个元素的一个“引用”。Rust 的引用与其他语言中的指针类似，不同的是有额外的编译时安全检查。特别指出的是，引用与所有权系统通过“借用”来相互作用,而不是通过拥有它。不同的是,当引用超出范围时，它不会释放底层的内存。如果是那样，我们将会释放两次内存，这是显然是不正确的!

让我们来添加第三行代码。表面上来是没错的，但是它会导致一个编译错误:

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

    let y = &x[0];

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

push 是将一个元素附加到向量组末端的方法。当我们试图编译这个程序时，我们将得到一个错误:

```
error: cannot borrow `x` as mutable because it is also borrowed as immutable
    x.push("foo");
    ^
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() {

}
^
```

Rust 编译器给了很详细的错误，这是其中的一次。如错误解释中所说,虽然我们的绑定是可变的，但我们仍不能调用 push 方法。这是因为我们已经有了一个矢量元素的引用 ,y 。当另外的一个引用存在时，改变某些变量是危险的行为，因为我们可能导致引用的无效。在这个特定的例子中，当创建向量时,我们可能只分配了三个元素的空间。添加第四个元素意味着所有这些元素将会被分配一块新的块内存，同时复制旧值到新内存，更新内部指针到新内存。这些工作都没有什么问题。问题是， y 不会更新,所以我们会有一个“悬空指针”。那就不对了。在这种情况下，任何的使用引用 y，将会导致错误，所以编译器已经帮我们捕捉到了这个错误。

那么，我们如何解决这个问题呢? 有两种我们可以采用的方法。第一种方法，我们利用拷贝而不是一个引用:

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

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

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

在默认情况下，Rust 有移动语义,所有如果我们想要复制一些数据，我们可以调用 clone() 方法。在这个例子中 ,y 不再是存储在 x 中向量的一个引用，而是它的第一个元素“ Hello ”的一个副本。现在没有引用，我们的 push() 方法可以很好地运行。

如果我们真的想用一个引用,我们需要另外一种选择：确保在我们尝试做改变之前，我们的引用跳出其作用域。写法看起来像这样：

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

    {
        let y = &x[0];
    }

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

我们用一组额外的大括号创建了一个内部作用域。在我们调用 push() 之前，y 已经跳出其作用域。所以这样是可行的。

所有权的概念不仅有利于防止悬空指针，而且有利于解决与其相关的所有问题，如迭代器失效，并发性等等。

2

新手入门

此教程的第一部分将会让你初步了解 Rust 及其工具。首先，我们将安装 Rust。然后，学习典型的 “Hello World” 程序。最后，我们将讨论下 Cargo，Rust 的构建系统和包管理器。

安装 Rust

使用 Rust 的第一步当然是安装它，有许多方法来安装 Rust，但其中最简单的方法是使用 rustup 脚本。如果你使用的是 Linux 或 Mac，所有你需要做的就是这些(请注意，你不需要输入 $，它们只显示每个命令的开始)：

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

如果你担心使用 `curl | sh` 所潜在的不安全，请继续往下看我们的免责声明。也可自由的选择使用两步的安装版本-安装和检查的脚本：

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

如果你使用的是 Windows，请下载适合版本的安装程序。

卸载

如果你决定不再想使用 Rust，我们会有点难过，不过没关系。并不是每一个编程语言对每个人来说都是合适的。运行下面的卸载脚本即可：

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

如果你使用的是 Windows 安装程序，重新运行 .msi 文件，它会显示给你一个卸载选项。

当我们告诉你使用 curl | sh 时，有些人会觉得非常不爽，某种程度上来说这是自然而然的。基本上当你这样做时,你是相信维护 Rust 的人是好人，并且他们不会攻击你的电脑或者做坏事的。这是一个很好的本能!如果你是这些人中的一员，请查看文档从源代码构建 Rust ，或官方下载.

我们还应该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

我们在这些平台上对 Rust 进行了大量的测试，像其他平台，如 Android，也做了一些测试，但是推荐利用上述几个平台，因为他们有最多的测试。

最后，关于 Windows 的一个说明。Rust 认为 Windows 是一个一流的发布平台，但我们诚实的讲，Windows 的体验不如 Linux/OS X 体验集成度好。我们正在努力提高它！如果有什么导致 Rust 不能工作，那这就是一个错误。如果这真的发生了，请通知我们。每一个错误的提交都会向任何其他平台一样，会在 Windows 上测试。

如果你已经安装了 Rust，你可以打开一个 shell，输入以下命令：

```
$ rustc --version
```

你将会看到版本号、提交哈希和提交日期。如果你只是安装版本1.0.0，你将会看到：

```
rustc 1.0.0 (a59de37e9 2015-05-13)
```

如果显示如上所示，表明 Rust 已经安装成功!恭喜你!

这个安装程序还会在本地安装文档的一个副本,这样你就可以离线阅读。在 UNIX 系统中，`/usr/local/share/doc/rust` 是安装的位置。在 Windows 上，它在一个 `share/doc` 目录里，无论你安装 Rust 到哪里。

如果不是如上所示,有很多地方你可以得到帮助。最简单的是在 `irc.mozilla.org` 上的 `# Rust IRC` 频道，你可以通过Mibbit访问它。单击该超链接，你就可以与其他 Rustaceans (我们用来称呼自己的萌萌的昵称)聊天了，我们可以帮助你。其它有用的资源包括用户论坛和 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。任何新的语言，按照惯例，让你写的第一个程序就是打印文本“Hello,world!“到屏幕上。从这样一个简单的程序开始的好处是，你可以验证你的编译器不只是安装好了，而且其工作也是正常的。打印信息到屏幕上是一个很常见的事情。

首先，我们需要做的就是创建一个能让我们写入代码的文件。我喜欢在我的主目录里创建一个 projects 目录，并把我所有的项目放在这里。Rust 是不在乎你的代码放在哪里的。

这实际上会导致另一个我们应该提到的问题:本教程假设你对命令行基本熟悉。Rust 本身对你的编辑工具或者代码所在的位置并没有特别要求。如果你更喜欢 IDE 而不是命令行，你可以试试 SolidOak，或者你最喜欢的 IDE 的其他插件。社区中有很多不同质量的扩展开发的组件。Rust 团队还为各种编辑器编写插件。配置编辑器或 IDE 超出了本教程的范围，所以如果需要，请查看你的安装的文档。

按照以上所说,让我们在项目目录中创建一个目录。

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

如果你使用的是 Windows，而不是使用 PowerShell， ~ 可能起作用。详细内容，请参阅您的 shell 文档。

让我们创建一个新的源文件。我们将新文件命名为 main.rs。Rust 文件总是以 .rs 扩展名结束。如果你的文件名使用超过了一个单词，使用下划线分开：是 hello_world.rs，而不是 helloworld.rs.

现在，打开你的文件，输入以下内容：

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

保存文件，然后在终端窗口输入：

```

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

```

成功啦!让我们详细看一遍刚刚发生了什么事。

fnmain() {

```
}
```

在 Rust 中，这些行定义了一个*函数*。这个 main 函数是特殊的：它是每个 Rust 程序的开始。第一行表示说，“我声明了一个函数，名

你还会注意到，函数是用花括号括起来的({和 })。Rust 需要这些花括号包裹起函数体。把左花括号和函数声明放在在同一行，并用空格

接下来是这一行：

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

在我们的小程序中，这一行做了所有的工作。有很多细节是非常重要的。首先，它有四个空格的缩进，没有标签。请配置你的编辑器，这

第二点是 println!() 部分。这里调用了一个 Rust 宏。宏是元程序编程在 Rust 中的实现。如果这里它是一个函数,它应该是这样的： pri

接下来， "Hello, world!"是一个‘字符串’。在系统编程语言里，字符串是一个令人惊奇的复杂主题。这里是一个‘静态分配’的字符串

最后，这一行以分号(；)结束。Rust 是一种面向表达式的语言，这意味着大多数代码是表达式，而不是语句。这个；用于表明这个表达

最终，编译和运行我们的程序。我们可以用我们的编译器编译，通过向命令 rustc 后面传递我们的源文件的名称：

```
$ rustc main.rs
```

如果你有 C 或 C++ 的背景，那么这就类似于 gcc 或者 clang。Rust 将输出一个二进制可执行文件。你可以用命令 ls 来查看它：

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

或者在 Windows 平台上：

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

现在有两个文件：我们带有扩展名 .rs 的源代码文件和可执行文件(在Windows上，main.exe ，其他地方， mian)。

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

这将在我们的终端打印出 Hello,world!.

如果你有动态语言像 Ruby，Python，或者 JavaScript 的背景,你可能不习惯这两个步骤是分开的。Rust 是一个预编译语言,这意味着
.py 或者 .js 文件，他们需要有一个 Ruby/Python/JavaScript 的实现安装，但是你只需要一个命令来编译和运行你的程序。一切都是这

恭喜你！你已经正式书写了一个 Rust 程序。这让你成为了一个 Rust 程序员！欢迎你！

接下来，我想介绍你认识另一个工具，Cargo。它用于编写真实的 Rust 程序。对于简单的代码，只是用 rustc 就很好了，但随着项目的

Hello, Cargo!

Cargo 是 Rustaceans 用来帮助管理他们的 Rust 项目的一个工具。Cargo 目前处在 pre-1.0 状态，所以它仍然是一项正在进行中的

Cargo 管理三个方面的事情：构建代码，下载代码所需要的依赖，构建这些依赖项。前期阶段，你的程序没有任何的依赖，所以我们只

如果你通过官方安装程序安装的 Rust，那么你也将拥有 Cargo。如果你用其他方法安装的 Rust，你可能希望查看 Cargo 的自述，通

转换成 Cargo

让我们将 Hello World 转换成 Cargo。

将我们的项目转换成 Cargo，我们需要做两件事情：创建一个 Cargo.toml 配置文件，把我们的源文件放在正确的地方。让我们先完成

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

注意：因为我们创建的是一个可执行文件，所以我们使用的是 main.rs 文件。而如果我们想创建一个库文件，我们应该使用 lib.rs。自定

Cargo 期望你的源文件放在 src 目录中。这与顶层的其他东西隔离开来，比如 READMEs，许可证信息和其他任何与代码无关的东西

接下来，编辑我们的配置文件：

$ editor Cargo.toml

一定要确保这个名字是正确的：首字母 C 要大写！

把下面的内容输入到文件里面：

[package]

```
name = "hello_world"
version = "0.0.1"
authors = [ "Your name <you@example.com>" ]
```

这个文件是 TOML 格式的。让它向你来做个自我介绍吧：

TOML 旨在成为一个最小的配置文件格式，由于明显的语义的使用，使其容易阅读。 TOML 旨在明确的映射到一个哈希表。在各种各样的语言中，TOML应该易于解析成数据结构。

TOML与 INI 非常类似，但比其有一些额外的优点。

一旦你有了这个文件，我们应该准备构建了！试试这个：

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

我们用 cargo build 构建我们的项目，并且用 ./target/debug/hello_world 运行它。我们可以用 cargo run 一步做到上面两件事：

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

请注意，这一次我们没有重建项目。Cargo 指出我们没有改变源文件，所以它只是运行的二进制文件。如果我们做了修改，我们会看到

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

这没有简单的使用 rustc 来进行整个过程。想想在将来：当我们的项目变得更加复杂时，我们需要做更多的事情让所有的部分正确编译

当你的项目终于准备发布了，你可以使用 cargo build --release 用最优化的方式编译你的项目。

你可能还会注意到，Cargo 已经创建了一个新的文件： Cargo.lock。

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

Cargo 用这个文件跟踪你应用程序中的依赖关系。现阶段，我们没有任何依赖，所以显得有点稀疏。你从不需要编辑该文件，让 Cargo

好了！我们已经成功地利用 Cargo 构建了 hello_world。尽管我们的程序很简单，但是我们开始使用的开拓未来 Rust 事业的真正工具

$ git clone someurl.com/foo $ cd foo $ cargo build

新建一个项目

每次当你想开始一个新的项目时，你不必经历整个过程！Cargo 有能力做一个基本的项目目录，你可以在此基础上马上开始开发。

用 Cargo 开始一个新项目，使用命令 cargo new：

$ cargo new hello_world --bin

我们传递了参数 --bin，因为我们正在创建一个二进制程序：如果我们创建的是一个库文件，就不要这个参数了。

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，你可以从你的发行版的包管理器中获取。它不是必需的，但是它确实有用。

这是开始我们需要的全部。首先，让我们看看 Cargo.toml：

[package]

```
name = "hello_world"
version = "0.0.1"
authors = ["Your Name <you@example.com>"]
```

Cargo 基于你提供的参数作为合理的默认值填充该文件，你可用 git 获取全局配置。你可能会注意到，Cargo 也初始化 hello_world 目

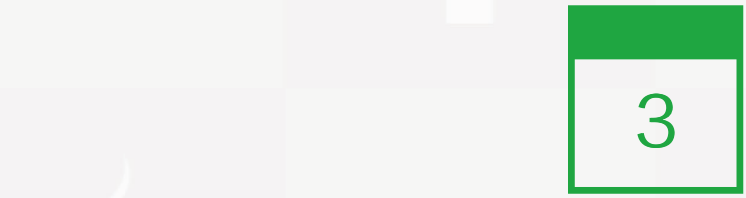
下面是 src/main.rs 里面的内容：

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

Cargo 已经为我们生成了一个 “Hello World！ “，你可以开始编码了！Cargo 有自己的指南，它涵盖 Cargo 许多更有深度的特性。

现在您已经了解了工具，让我们了解更多关于 Rust 语言本身的内容。这些基础知识将使你在剩下的时间里更好的理解 Rust。

你有两个选择：深入项目，进入章节”学习 Rust”，或者从基础开始，学习“语法和语义”章节。更有经验的系统程序员可能会更喜欢“学习 Rust”，而拥有动态语言背景的可能两者皆可。不同的人有不同的学习方式！选择适合你的。



学习 Rust

欢迎学习本节！本节有几个教程，通过构建项目教你 Rust。你将会得到一个高度概览，而我们会掠过细节部分。

如果你更想喜欢一个“从头开始”风格的学习经历，请学习“语法和语义”。

猜谜游戏

我们的第一个项目，将实现一个典型的初学者编程的问题：猜谜游戏。下面介绍下它是如何工作的：我们的程序将生成一个从一到一百的随机整数。然后它会提示我们输入一个猜测值。依据我们的输入的猜测值，它会告诉我们猜测值是否过低或者过高。一旦我们猜正确，它将祝贺我们。听起来不错吧?

设置

进入你的项目目录，让我们建立一个新项目。还记得我们必须为 `hello_world` 创建目录结构和 Cargo.toml 吗? Cargo 有一个命令能为我们做这些事。让我们来试一试:

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

我们向命令 cargo new 传递了我们项目的名称，然后添加了标记 --bin ，这是因为我们正在创建一个二进制文件，而不是一个库文件。

查看生成的 Cargo.toml:

```
[package]

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

Cargo 从你的环境中得到这些信息。如果它是不正确的，进入文件并且改正过来。

最后，Cargo 为我们生成一个“Hello，world!”。查看下 `src/main.rs` :

```
fnmain() {
    println！("Hello， world！")
}
```

让我们尝试编译下 Cargo 给我们的东西:

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

太好了！再次打开你的 `src/main.rs` 。我们将在这个文件中编写所有的代码。

在我们继续之前，让我教你一个 Cargo 的命令： run.cargo run 是有点像 cargo build 的命令，但是它可以在生成的可执行文件后运行此文件。试一试：

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

太棒了！当你需要在一个项目中快速迭代时，这个 run 命令可以派上用场。我们游戏就是这样的一个项目，在下一次迭代之前，我们需要快速测试每个迭代。

处理一个猜测值

我们开始做吧！对于我们的猜谜游戏，我们需要做的第一件事就是允许玩家输入一个猜测值。将下面的内容输入到你的 `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);
}
```

这里面有很多代码啊！让我们一点一点地输入它。

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

我们需要获取用户的输入，然后将打印结果作为输出。因此，我们需要标准库中的 io 库。Rust 利用“序部”只要引入少量的东西到每一个项目，。如果在序部中没有的话，那么你必须直接 use 它。

```
fn main() {
```

正如之前你所见过的， main() 函数是程序的入口点。这个 fn 语法声明了一个新函数， ()表明没有参数， {表示函数的主体开始。因为我们不包含一个返回类型，它假定是 ()，一个空元组。

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

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

我们之前学习到的 println！() 是一个打印字符串到屏幕上的宏。

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

现在越来越有趣了！有很多东西在这个小行上。首先要注意的是，这是一个let 语句，是用来创建变量绑定的。他们以这种形式存在：

```
let foo = bar;
```

这将创建一个名为 foo 的新绑定，并将其绑定到值 bar 上。在许多语言中，这被称为一个“变量”，但是 Rust 的变量绑定有其成熟的方法。

例如，他们在默认情况下是不可变的。这就是我们的例子中使用 mut 的原因：它使一个绑定是可变，而不是不可改变的。 let 的左边不能添加名称，实际上，它可接受一个“模式”。之后，我们将更多的使用模式。现在它很容易使用：

```
let foo = 5; // 不可变的.
let mut bar = 5; // 可变的
```

哦， `//` 之后将开始一个注释，直到行的结束。Rust 编译时，将会忽略所有的注释。

所以现在我们知道了 let mut guess 将引入一个名为 gusss 可变的绑定，但我们必须看 `=` 另一边的绑定： `String::new()` 。

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

`::new()` 语法使用 `::` 因为这是一个特定类型的关联函数。也就是说，它与 String 本身相关，而不是一个特定的实例化的 String。一些语言称之为“静态方法”。

这个函数命名为 new()，因为它创建了一个新的，空的 String。你会在许多类型上发现 new()函数，因为它是创建某类型新值的一个普通的名称。

让我们继续往下看：

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

这又有很多代码！让我们一点一点的看。第一行包含两部分。这是第一部分：

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

还记得我们如何在程序的第一行使用 `use std::io` 吗?我们现在调用一个与之相关的函数。如果我们不用 `use std::io`，我们可以这样写这一行 `std::io::stdin()`。

这个特别的函数返回一个句柄到终端的标准输入。更具体地说，返回一个 `std::io::Stdin`。

下一部分将使用这个句柄获取来自用户的输入：

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

在这里，我们在句柄上调用 `read_line()` 方法。方法就像联合的方法，但其只能在一个特定的实例类型中使用，而不是在类型本身。我们还传递了一个参数到 `read_line(): &mut guess`。

还记得我们上面是怎么绑定 guess 的吗?我们认为它是可变的。然而， `read_line` 不是将一个 String 作为一个参数：它用 `&mut String` 做参数。Rust 有一个称为“引用”的特性，对一个数据你可以有多个引用，这样可以减少复制。引用是一个复杂的特性。如何安全和容易的使用引用是 Rust 的一个主要卖点。现在我们不需要知道很多这些细节来完成我们的项目。现在我们所需要知道的是像 let 绑定，引用是默认是不可变的之类的事情。因此，我们需要写代码 `&mut guess`，而不是 `&guess`。

为什么 `read_line()` 采用一个可变的字符串引用做参数? 它的工作就是获取用户键入到标准输入的内容，并将转换成一个字符串。因此，以该字符串作为参数，因为要添加输入，所以它需要是可变的。

但是我们不确定这一行代码会执行正确。虽然它仅是一行文本，但是它是第一部分的单一逻辑行代码：

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

当你用类似 .foo() 的语法调用一个方法时，你可以用换行和空格来写开启新的一行代码。这能帮助你分开很长的代码行。我们之前的代码是这样写的：

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

但是这样的代码阅读性不好，所以我们将其分隔开。三个方法调用分成三行来写。我们已经讲过 `read_line()` 了，那么 ok() 和 expect() 是什么呢?嗯，我们之前已经提到了， `read_line()` 读取了用户输入到 &mut String 中的内容。而且它也返回了一个值：在上面的例子中，返回值是 `io::Result`。Rust 在其标准库中有很多类型命名为 `Result:` 一个通用的 Result，然后用子库指定具体的版本，比如 `io::Result`。

使用这些 Result 类型的目的是编码错误处理信息。Result 类型的值，类似于任何类型，其有定义的方法。在上面的例子中， `io::Result` 就有一个 ok() 方法。这个方法表明，“我们想假定这个值是一个正确值。如果不是，那么就抛出错误的信息。”那么为什么要抛出它? 对于一个基本的程序，我们只是想打印出一个通用的错误信息。任何这样的基本问题的发生意味着程序不能继续运行了。ok() 方法返回了一个值，在这个值上定义了另外一个方法： expect()。 expect() 方法调用时，会获取一个值，如果这个值不是正确的值，那么会导致一个应

急（ panic！ ）的发生，你要传递一个信息给这个应急。这样的一个应急(panic！)会导致我们的程序崩溃，并且会显示你传递的信息。

如果我们不调用这两个方法，程序能编译成功，但我们会得到一个警告信息：

```
    $ 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，我们没有使用 Result 值。这个警告来自 `io::Result` 所有的一个特殊的注释。Rust 试图告诉你，你没有处理一个有可

第一个例子就剩下这一行代码了：

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

这行代码打印出我们输入的字符串。{} 是一个占位符，所以我们把 guess 作为参数传递给它。如果我们有多个 {} ，那么我们将传递多

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

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

容易吧。

不管怎样，这就是我们的学习过程啦。我们可以用我们所拥有的 cargo run 来运行代码了：

```
$ 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
```

好吧！我们的第一部分已经完成了：我们可以从键盘输入，然后将它打印出来。

生成一个秘密数字

接下来，我们需要生成一个秘密数字。Rust 还未引入有随机数方法的标准库。然而，Rust 的团队提供了一个随机箱 [rand crate](http

使用外部箱是 Cargo 的一个闪光点。在我们使用 rand 写代码之前，我们需要修改 Cargo.toml。打开它，并在文件底部添加这几行：

```
[dependencies]

rand="0.3.0"
```

Cargo.toml 的 [dependencies] 的部分就像 [package] 部分一样：直到下一个部分开始之前，它下面的内容都是它所包含的。Cargo

现在，在不改变我们的任何代码，让我们重新构建我们的项目：

```
$ 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)
```

(你可以看到不同版本)。

有大量的新的内容出现了！现在我们有一个外部依赖，Cargo 从注册表获取所有依赖的最新版本，我们这里的注册表是一个来自 Crate

更新注册表，Cargo 会检查我们的 `[dependencies]` 并下载我们没有的一些依赖。在本例中，尽管我们说我们只想依赖 rand，但是我

如果我们再次运行 cargo build，我们将会得到不同的输出：

```
$ cargo build
```

没有错，这里是没有输出的！Cargo 知道我们的项目已经构建完成，并且项目所有的依赖项也构建完成，因此没有理由将所有这些事情

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

所以，我们告诉 Cargo 我们想要任何 0.3.x 版本的 rand，因此它会获取当时的最新版本，v0.3.8。但是当下周有重要错误修正的版本

这个问题的答案是锁文件 Cargo.lock，现在你可以在你的项目目录中找到它。当你第一次构建你的项目时，Cargo 会找出所有的符合

那么当我们真的想使用 v0.3.9 版本时，怎么办？Cargo 还有另外一个命令， update，它的意思是“忽略锁文件，找出适合我们指定要

这里提到了很多关于 [Cargo](http://doc.crates.io/) 和它的[生态系统](http://doc.crates.io/crates-io.html)的内容，对现在来说，这

让我们实际使用一下 rand。下面是我们要进行的下一个步骤：

```
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);
}
```

我们所做的第一件事就是改变了第一行代码。现在第一行是，extern crate rand。因为我们在我们的 `[dependencies]` 中声明了 ran

接下来，我们添加了另一个 use 行： `use rand::Rng`。一会儿我们要用一种方法，它需要 Rng 在作用域内才运行。基本的思路是这

在中间，有我们添加的另外两行：

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

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

我们使用 `rand::thread_rng()` 函数获取随机数产生器的一个副本，这是我们在执行的一个本地线程。我们上面使用 `use rand::Rng`

第二行打印出了这个生成的秘密数字。这是在我们开发项目时是非常有用的，我们可以很容易地对其进行测试。但是在最终版本里我们

尝试运行几次我们的程序：

```
$ 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
```

太棒了！下一篇：让我们用我们的猜测值和秘密数字比较一下。

比较猜测值

现在，我们已经得到了用户的输入，让我们将我们猜测值与随机值比较下。下面是我们要进行的下一步，它现在还不能运行：

```
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!"),
```

```
    }
}
```

这里的有几个新部分。第一个是另外的一个 use。我们将叫做 `std::cmp::Ordering` 的类型放到了作用域内。然后是底部新的五行代码

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

cmp() 方法可以在任何可比较的地方被调用，但是它需要你想要比较东西的一个引用。它将返回我们之前 use 的 Ordering 类型。我们

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

利用这个定义，任何类型是 Foo 的类型，可以是一个 `Foo::Bar` 或者是一个 `Foo::Baz`。我们使用 `::` 表明一个特定 enum 变量的命

Ordering 枚举有三个可能的变量： Less， Equal 和 Greater。 match 语句需要有一个类型的值输入，并允许你为每一可能的值创建

```
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！，如果是 Greater，打印 Too big！，如果 Equal，打印 You win！。 match 是非常有用的，

我之前提到了，项目现在还不能运行。让我们来试一试：

```
$ 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`.
```

这是一个很严重的错误。它核心的意思是，我们有不匹配的类型。Rust 有一个强壮的、静态的类型系统。然而，它也有类型推导过程。

```
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!"),
    }
}
```

新的三行代码：

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

等一下，我想我们已经有了一个 guess 了吧?对，我们确实有一个，但是 Rust 允许我们用一个新的 guess 来“遮住”前面的那一个。

类似于我们之前写的代码那样，我们将 guess 绑定到一个表达式：

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

紧随其后的是一个 ok().expect() 的调用。这里的 guess 指的是旧的那个 guess，也就是我们输入 String 到其中那一个。String 的 tr

就像 `read_line()`，我们的 parse() 调用可能会导致一个错误。如果我们的字符串包含了 A% 呢？这就没有办法转换成一个数字了。因

让我们来试试我们的新程序！

```
$ 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: 58
Please input your guess.
 76
You guessed: 76
Too big!
```

好了！你可以看到我甚至在猜的数字之前添加了空格，程序仍然发现我猜的数是 76。运行几次程序，再猜一个比较小的数字，验证猜测

现在我们已经做了游戏的大部分工作，但我们只能做一个猜测。让我们通过添加循环来改变这一情况！

循环

loop 关键字为我们提供了一个无限循环。让我们添加如下的代码并尝试一下：

```
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!"),
    }
  }
}
```

先等一下，我们只是添加一个无限循环么？是的，没错。还记得我们关于 parse() 的讲解吗？如果我们给出了一个非数字的回答，它将

```
$ 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: 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!'
```

像其他非数字的输入一样，quit 确实是退出了。嗯，至少可以说这是一个次优的解决方案。第一个方案：当你赢得比赛时，程序退出：

```
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;
            }
        }
    }
}
```

通过在打印 You win！的代码后面添加一行 break，当我们赢了的时候就会退出循环。退出循环也意味着退出了程序，因为它是 main(

```
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;
            }
        }
    }
}
```

这些代码改变了：

```
let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
```

```
    Err(_) => continue,
};
```

通过将使用 ok().expect() 换做使用一个 match 语句，这可以教会你是如何从“崩溃错误”转变到“处理错误”的。parse() 返回的 R

现在，问题应该都解决了！让我们试一试：

```
$ 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!
```

太棒了！经过最后一个小小的调整，我们已经完成了猜谜游戏。你能想到什么？没错，我们不想打印出秘密数字。打印这个数字对于测试

```
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;
            }
        }
    }
}
```

完成了！

此时此刻，你已经成功地构建了猜谜游戏！恭喜你！

这第一个项目向你展示了很多内容： let， match，方法，相关函数，使用外部箱等等。我们的下一个项目将展示更多的内容。

哲学家就餐问题

对于我们的第二个项目，让我们来看一个典型的并发性问题。这就是“哲学家就餐问题”。这最初是由迪杰斯特拉在 1965 年提出的，

>在古代，一个富有的慈善家捐赠了一所学院来安排五个著名的哲学家。每个哲学家都有一个房间，他可以在其中从事他自己专业的思考

这个经典问题展示了一些不同的并发性的元素。原因在于，其实际实现起来比较复杂：一个简单的实现可能导致死锁。举一个例子，让我

1. 一个哲学家拿起了自己左边的叉子。
2. 然后他又拿起了他右边的叉子。
3. 吃意大利面。
4. 放下叉子。

现在，让我们来想象一下这一系列的事件：

1. 哲学家 1 开始此算法，拿起他左边的叉子。
2. 哲学家 2 开始此算法，拿起他左边的叉子。
3. 哲学家 3 开始此算法，拿起他左边的叉子。
4. 哲学家 4 开始此算法，拿起他左边的叉子。
5. 哲学家 5 开始此算法，拿起他左边的叉子。
6. …? 所有的餐叉都被拿起来了，没人能吃面了!

有不同的方法来解决这个问题。在本教程中我们将用我们的方法来解决它。现在，让我们开始从问题的本身开始建模。我们先从哲学家

```
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");
}
```

在这里，我们创建了一个结构体来代表一个哲学家。在结构体里，我们现在所需要的仅仅是一个名字。我们将名字的类型选择成 String

我们继续看下面的部分：

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

这里的 impl 块能让我们为 Philosopher 结构体定义一些内容。在本例中，我们定义了一个名叫 new 的关联函数。它的第一行如下所示

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

这里我们需要输入一个参数 name，类型是 &str。这是另一个字符串的一个引用。此函数会返回一个 Philosopher 结构体的实例。

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

这将会创建一个新的 Philosopher，并将它的 name 值设置成前面 name 的参数值。但这不是参数的自身，因为我们对其调用了 `.to_

那么为什么不直接用 String 类型的参数？这样更容易调用。如果我们采用了 String 类型参数，但是我们的调用者所有的是 &str，那么

最后一件你可能会注意到的事：我们只定义了一个 Philosopher，似乎用它什么事情也没有做。Rust 是一种基于表达式的语言，这意

对于 Rust，这个名字，new()，没有什么特别，但是它是创建新的结构体实例的函数的约定俗成的叫法。在我们讨论为什么是这样之前

```
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");
}
```

在这里，我们创建了五个 Philosopher 的变量绑定。这五个名字都是我最喜欢的，你也可以用任何你想要的名字来替换他们。如果我们

```
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() };
}
```

这看起来就比较麻烦了。当然使用 new 也有其他的一些优势，而在我这个简单的例子中，使用它也使我们的代码简洁了不少。

现在我们在这里已经写好了基本的内容，同时会有很多方法可以解决上述问题。我想从问题的结束端先开始：让我们建立一个每个哲学家

```
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();
    }
}
```

让我们首先来看 main()。我们没有采用有五个哲学家 Philosopher 的变量绑定，相反我们创建了一个哲学家的 `Vec<T>`。`Vec<T>

在循环体内，我们调用的是 `p.eat()`，下面是关于其函数定义：

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

在 Rust 中，方法可以用一个明确的 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() 中没有 self。我们

```
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.
```

够简单吧，程序运行完全没有问题!然而我们还没有实现真正的问题，所以我们还没有完成整个程序!

接下来，我们想想做的是：我们的哲学家不仅吃完，而且实际上他们也是在吃的。下面是程序的下一个版本：

```
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();
    }
}
```

仅有几个变化。让我们一个一个的来讲。

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

use 可以使后面的库加载在作用域内。我们稍后会使用标准库中的 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);
}
```

我们现在打印了两个消息，其中间有一个 `sleep_ms()` 将其分隔开。`sleep_ms()` 将模拟哲学家吃的时间。

如果现在你运行这个程序，你会看到每个哲学家轮流吃：

```
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.
```

太好了!我们又前进了一步。这里还有一个问题：我们实际上并没有以并行的方式操作，并行才是就餐问题的核心部分!

为了能让我们的哲学家并发的吃，我们需要做一个小改变。下面是下一个版本：

```
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"),
    ];

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

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

我们所要做的就是改变 main() 里面的循环，在其中添加上第二个循环体!下面是第一个改变：

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

虽然这里只有五行，但这五行代码确很密集。让我们一个一个的来讲。

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

我们这里引入了一个名叫 handles 新的绑定。我们这样叫它，是因为我们要创建一些新的线程，而这些线程会返回一些能让我们控制此

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

我们的哲学家 philosophers 调用了 into_iter()。这将创建一个迭代器，它将拥有每个哲学家的所有权。我们要这样才能将哲学家传递到

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

这里就是发生并发的地方。`thread::spawn` 函数以一个闭包作为参数，并会在一个新线程里执行这个闭包。关于这个闭包的一些额外的

在线程内部，我们所做的就是用 p 调用 eat()。

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

最后，我们获取了所有的 map 调用的结果并把它们收集起来。`collect()` 会将它们生成某种集合，这就是为什么我们要解释返回类型的

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

main() 函数的结尾部分，我们对 handles 进行循环处理，对其调用 join()，这可以阻塞其它线程的执行，直到本线程执行完成。这确保

如果你运行这个程序，你会发现哲学家可以自由的吃啦!到这儿，我们已经有多线程程序啦!

```
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.
```

但是叉子呢？我们还没有为它们建模。

要做到这一点，让我们写一个新的 struct：

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

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

这桌子 Table 有一个互斥元(Mutex)的向量。互斥元是控制并发性的一种方法：一次只能有一个线程可以访问内容。这正是我们叉子所

让我们使用 Table 来修改这个程序：

```
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();
    }
}
```

又有了许多的变化!这一次我们有了最终可运行的版本。我们看下细节内容:

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

我们将使用 `std::sync` 包中的另一个结构：`Arc<T>`。当我们使用它时，会做进一步的讲解。

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

我们需要为 Philosopher 添加两个字段。每个哲学家都有两个叉子：一个在左，一个在右。我们用 usize 类型来表示它们，因为这类型

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

我们现在需要构造 left 和 right 的值了，所以我们将它们添加到了 new() 中。

```
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);
}
```

我们又添加了两行新代码。我们还添加了一个参数，table。这样我们就可以访问 Table 的 fork 列表，然后可以用 self.left 和 self.righ

lock() 的调用可能会失败，如果是这样，程序会崩溃。在这种情况下，错误的发生可能是因为互斥元“[中毒](https://doc.rust-lang.or

这些行另一个奇怪的现象是：我们将结果命名为 `_left` 和 `_right`。这些下划线有什么用？嗯，因为我们不打算使用锁内的值。我们只

那么如何释放锁呢？嗯，当 `_left` 和 `_right` 超出作用域后，锁会自动释放。

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

接下来，在 main() 中，我们创建了一个新的 Table 并定义成 `Arc<T>` 类型。“arc” 是 “atomic reference count” 的缩写，我

```
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),
];
```

我们要将我们的 left 和 right 值传递到 Philosopher 的构造函数中。这里有一个非常重要的细节。如果你查看他们的样式，开始直到最

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

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

最后，在 `map()/collect()` 循环中，我们调用了 `table.clone()`。`Arc<T>` 调用的 `clone()` 方法会增加引用的计数，但是当它超出作

到这里，我们的程序就可以工作啦!只有两个哲学家可以在同一时间吃，所以你会得到一些如下的输出：

```
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!你已经用 Rust 实现了一个典型的并发性问题。

Rust 嵌入到其他语言

我们的第三个项目，我们要选择展示那些能展示 Rust 最大优点的点：大量运行时的减少。

随着我们组织的发展，其越来越依赖其他的一些编程语言。不同的编程语言有不同的优点和缺点，通晓数种语言的堆栈允许你使用一个

许多程语言一个共同薄弱的地方就是程序的运行时性能。通常情况下，使用了一种运行比较慢的语言，但是如果它同时能提升程序员的

Rust 在两个方面上支持 FFI:它可以容易的调用 C 代码，但至关重要的是，它也可以像容易调用 C 代码那样被调用。当你需要一些额外

在本教程中，我们有一整章来讲述 FFI 和它的细节，但是在本章中，我们将用三个例子来展示 FFI 的特定用例，它们分别是在 Ruby，

问题

这里我们有很多不同的项目可供选择，但我们要选择一个能展示 Rust 比其他许多语言有明显优势的例子：数值计算和线程。

许多语言为了一致性，将数字存放在堆上，而不是在堆栈上。尤其是在专注于面向对象编程和使用垃圾收集的语言上，默认的分配模式

第二问题，许多语言有一个“全局解释器锁”，这在许多情况下限制了并发。这是以安全的名义来进行的，本来这是一个好意，但是它

为了强调这两个方面，我们要创建一个能使用到这两方面的一个小项目。因为示例的重点是将 Rust 嵌入到其他语言，而不是这个问题

开十个线程。每个线程内部实现从一数到五百万。数完后，十个线程结束，并打印出“done!”。

这里我基于我计算机的能力选择了五百万。下面是一个用 Ruby 写的例子的代码:

```
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，并选择一个数字运行几秒钟。基于你的电脑硬件，你可能要增大或减小这个数字。

在我的系统中，运行这个程序需要 2.156 秒。如果我用某种类似 top 的进程监控工具，我可以看到它在我的机器上只使用一个 CPU 核

虽然这确实是一个人工合成的程序，但是你也可以想象许多与现实世界相似的问题。就我们的目的而言，运行一些繁忙线程就代表了某

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
```

这个程序用 Rust 写起来很简单:

```
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!");
```

```
    }
}
```

这个程序中的一些内容与从先前的例子看起来很相似。我们开启了十个线程，将它们收集成一个 handles 向量。在每个线程中，我们都

```
$ 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。如果我们有一个关于此函数的测试，那么这个警告将会消失。但是现在，这个函数从未被调用

第二个警告与 x ，`_x` 有关。因为我们对 x 没有做过任何操作，所以我们得到了一个警告。在我们的例子中，这是完全没有问题的，因

最后，我们连接了每个线程。

然而现在这是一个 Rust 库，它还没有公开任何从 C 中可调用的代码。如果现在我们试图将其链接到另一种语言，那么它是不能使用的

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

我们必须添加一个新的属性，no_mangle。当你创建一个 Rust 库时，在编译输出阶段会改变函数的名称。这个的原因超出了本教程的

另一个变化就是 pub extern。pub 意味着在这个模块以外这个函数应该是可调用的。extern 表示它应该能够从 C 中被调用。就这样了

我们需要做的第二件事就是是改变 Cargo.toml 的设置。添加如下的内容到底部:

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

这会告诉 Rust，我们想将我们的库编译成标准动态库。默认情况下，Rust 会编译成一个 ‘rlib’，这是一个 Rust 独有的格式。

现在让我们来构建项目:

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

我们选择最优化的构建方式 `cargo build --release`。因为我们希望程序能运行的尽可能快！你可以从 `target/release` 中找到库文件

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

这里的 libembed.so 是我们的‘共享对象’库。我们可以像用 C 写的对象库一样使用这个文件!说句题外话，根据平台的不同，这个文

现在我们已经构建了我们的 Rust 库，让我们在 Ruby 使用它吧。

Ruby

在我们的项目中打开 embed.rb 文件，并且按如下所做：

```
require 'ffi'

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

Hello.process

puts "done!”
```

在我们可以运行这个程序之前，我们要先安装 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
```

最终，我们可以试着运行一下：

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

哇，好快！在我的系统上，这花费了 0.086 秒的时间，而不是纯 Ruby 版本花费的两秒钟。让我们详细讲一下这段 Ruby 代码：

```
require 'ffi
```

首先我们需要 ffi gem。这能让我们像连接 C 库一样与 Rust 库连接。

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

ffi gem 的作者推荐使用一个模块来圈定我们将要从共享库中导入的方法的作用域。在模块里面,我们 extend 了必要的 FFI::Library 库

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

`attach_function` 方法是由 FFI gem 提供的。它是用来连接 Rust 与 Ruby 中同名函数 process() 的。因为 process() 不需要任何

```
Hello.process
```

这才是对 Rust 的实际调用。我们的 module 和 `attach_function` 的调用结合才成就了它。它看起来像是一个 Ruby 函数，但实际上

```
    puts "done!"
```

最后，根据我们之前的项目的需求，我们打印出 done!

就是它了!正如我们所看到的，两种语言之间的桥接是真的很容易，并且提升了很多性能。

接下来，让我们来试试 Python 吧!

Python

在目录中创建一个 embed.py 文件，并将如下内容输入：

```
from ctypes import cdll

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

lib.process()
```

```
print("done!")
```

这个更简单！我们用 ctypes 模块中的 cdll。稍后对 LoadLibrary 的一个快速调用后，我们就可以调用 process()了。

在我的系统中，这花费了 0.017 秒。快吧！

Node.js

Node 不是一门语言，但它目前是服务器端 JavaScript 的主要实现。

为了用 Node 实现 FFI，我们首先需要安装库:

```
$ npm install ffi
```

安装完成后，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!");
```

```

与 Python 的例子相比，它看起来更像 Ruby 的例子。我们用 ffi 模块获得 ffi.Library()，它负责加载共享对象。我们要解释下函数的返回类型和参数类型，返回的是‘空’，参数是一个空数组来表示的。从此之后，我们就可以调用它并打印结果。

在我的系统中，程序运行花费了 0.092 秒。

## 结论

正如你可以看到的，这些基本操作是非常简单的。当然，这里我们还有很多可以做的。更多的细节请查看 FFI 章节。
```

4

高效 Rust

现在你已经学会了如何编写一些 Rust 代码。但是在编写出 Rust 代码和编写出良好的 Rust 代码，还是有一定区别的。

这部分由相对独立的教程向您展示如何把 Rust 发展到下一个水平。将介绍常见的模式和标准库特性。你可以按你选择的任何顺序来阅读这些章节。

栈和堆

作为一种系统语言，Rust 运行在较低的层次。如果你只学习过高级语言，有一些系统编程方面的问题，你可能不熟悉。最重要的一个问题是存储器如何工作，例如如何使用堆和栈。如果你对 c 语言如何使用堆栈分配熟悉的话，本章将会是一个复习。如果你不熟悉的话，你将会学习到Rust-y 关注的一些相关基本概念。

内存管理

关于内存管理有两个常用术语。栈和堆是一种抽象概念，帮助您确定何时分配和释放内存。

这里有一个更高层次的比较：

栈是非常快的，也是 Rust 默认的内存分配方式。但是分配存在于本地函数调用，且在大小方面是有限的。另一方面，堆的速度相对比较慢，但是你的程序可以明确地分配堆内存。且它实际上是无限制的，可以在全局范围内访问。

栈

让我们谈谈这个 Rust 程序:

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

这个程序有一个变量绑定 x。这需要分配内存。默认情况下，Rust 进行栈分配，这意味着基值存储到栈内。这是什么意思呢?

当一个函数被调用时，一些内存被分配给它的所有局部变量和其他一些信息。这就是所谓的“栈帧”，本教程的目的，我们将忽略额外的信息，只考虑局部变量的分配。所以在这种情况下，当 main() 运行时，将为我们的栈帧分配一个 32 位整数。如你所见，这是自动处理的，我们不必为此编写任何特殊 Rust 代码或其他任何东西。

函数结束后，它的栈帧被收回。这也是自动实现的，我们不需要做什么特别的事情。

这就是一个简单的程序全部的内容。关键的是要理解栈分配是非常，非常快的。在我们了解所有的局部变量之前，我们有时间可以一次性抓取内存。并且因为我们可以在同一时间丢弃他们，我们也可以迅速摆脱它。

不利的一面是，当我们不只是在一个单一的函数内需要它们时，却不能长存这些值。我们还没有谈到这个名字：“栈”的意思。为此,我们需要一个稍微复杂一点的例子：

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

fn main() {
    let x = 42;

    foo();
}
```

这个程序总共有三个变量：两个在 foo() 中,一个在 main() 中。和之前一样，当 main() 被调用时，单个整数分配它的栈帧。但是在此之前可以显示调用 foo() 时会发生什么，我们需要想象内存中是如何操作的。操作系统提供了一个程序的内存视图，它非常简单：一个巨大的地址列表，从 0 到很大的值，代表你的计算机有多少内存。例如，如果你有一个 G 的内存，你的地址从 0 到 1,073,741,824。这一数字来自于 2 的 30 次方，是一个十亿字节的数字级别。

这个内存就像是一个巨大的数组：地址从 0 开始，到最后的数。这是第一个栈帧的图表：

地址	名字	值
0	x	42

我们可以知道地址为 0 的地方存储了一个名称为 x，值为 42 的东西。

调用 foo()，一个新的栈帧被分配：

地址	名字	值
2	z	100
1	y	5
0	x	42

因为 0 被第一帧占用，1 和 2 用于 foo() 的栈帧。我们调用的函数越多，内存向上增长的越多。

这里，我们必须注意一些重要的事情。数字 0、1 和 2 都仅仅是为了便于说明，和计算机实际使用的情况没有丝毫关系。特别是，是现实中，一系列地址是会被一些单独的自己隔开成单独的每个地址的，这些分割甚至可能会超过存储的值的大小。

foo() 结束后，其帧被收回：

地址	名字	值
0	x	42

然后，main() 结束后，甚至最后的值都会消除。简单吧!

这就是所谓的“栈”，因为它像一个餐盘的栈：你放下去的第一块盘子将是你最后拿回来的盘子。栈有时被称为“后进先出队列”的原因，就是你最后放入栈内的值就是第一个你需要检索的值。

让我们尝试一个三层的例子：

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

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

    bar();
}

fn main() {
    let x = 42;

    foo();
}
```

好吧，首先，我们调用 main()：

地址	名字	值
0	x	42

接下来，main() 调用 foo()：

地址	名字	值
3	c	1
2	b	100
1	a	5
0	x	42

然后 foo() 调用 bar()：

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

现在，我们的栈越来越高了。

bar() 结束后，其栈帧被回收，留下 foo() 和 main()：

地址	名字	值
3	c	1
2	b	100
1	a	5
0	x	42

然后 foo() 结束后，只留下 main()：

地址	名字	值
0	x	42

然后我们就大功告成了。明白了吗？就像堆餐具：一直添加到顶部，然后从顶部开始取走。

堆

这种方法可以很好地工作，但并不是一切事物都可以像这样工作。有时，不同功能之间需要共享内存，或者不只是在单个函数的执行期间保持活动状态。为此，我们可以使用堆。

在 Rust 中，你可以使用 Box 类型在堆上分配内存。这里有一个例子：

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

当 main() 被调用时，内存发生如下变化：

地址	名字	值
1	y	42
0	x	??????

我们在栈上为两个变量分配空间。像以往一样，y 值为 42，但 x 呢？ 恩，x 是一个 Box 类型的值，Boxes 在堆上分配内存。boxes 的实际值是一种数据结构，实际上是指向“堆”的一个指针。当我们开始执行函数，调用 Box::new()，它为堆分配一些内存，将 5 放在内存内。内存现在看起来像这样：

地址	名字	值
2的30的方		5
...	...	...

1	y	42
0	x	2的30的方

在一个假想的电脑里，我们有 1GB 的 RAM。因为我们的栈从零增长，从最简单的地方开始分配内存。所以我们的第一个值是在内存中最高的地方。x 这个结构的值有一个原始指针，指向在堆上分配的位置，所以 x 的值是 2 的 30 次方，即我们请求的内存位置。

关于在这些环境中实际上如何进行分配和释放内存，我们还没有谈论太多。本教程中，我们会讨论更深的细节，但这里必须指出的是，堆不像栈是从一端开始生长。在本书后边的部分，我们会讨论到这样的一个例子，但是由于堆可以以任何顺序分配和释放内存，最终以很多“洞”的存在结束。这是一个现在已经运行了一段时间的程序的内存布局图:

地址	名字	值
2的30的方		5
2的30的方−1		
2的30的方−2		
2的30的方−3		42
...	...	...
3	y	2的30的方−3
2	y	42
1	y	42
0	x	2的30的方

在现在的情况下，我们在堆上分配了四个事物，但释放了两个。目前，在 2 的 30 次方和 2 的 30 次方 −3 之间还有一些空白还没有被使用。这种情况如何以及为什么发生的具体细节取决于你用什么样的策略来管理堆。不同的程序可以使用不同的内存分配器，有函数库来管理。Rust 程序使用 jemalloc 达到这个目的。

无论如何，现在回到我们的例子。因为这是堆上的内存，它的生存周期可以超过函数分配 box 的范围。然而，在这种情况下，在函数结束后，它不能 [moving]。我们需要通过 main().Box 释放栈帧，不过，有一个更巧妙的技巧：Drop。Drop 即释放创建 box 时分配的内存。太棒了!因此，当删除 x 时，它首先释放分配在堆上的内存:

地址	名字	值
1	y	42
0	x	??????

[moving]：我们可以使内存有更长的生存周期，通过转移所有权，有时被称为 “moving out of the box”。稍后介绍更复杂的例子。

然后栈帧消失，释放我们所有的内存。

参数和引用

我们已经了解了一些栈和堆的基本例子，但是关于函数参数和引用呢?这里有一个小的 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	名字	值
1	y	0
0	x	5

x 值为 5，y 是 x 的一个引用。所以它的值是 x 的内存地址，本例中为0。

调用 foo()，将 y 作为参数传递：

地址	名字	值
3	z	42
2	i	0
1	y	0
0	x	5

栈帧不只是本地绑定，它们还可以用作参数。在这种情况下，我们需要有 i 作为参数，z 作为局部变量绑定。i 是参数 y 的一个副本。由于 y 的值为 0，所以 i 的值也是 0。

这就是为什么引用一个变量不占用任何内存：一个引用的值只是一个指向内存位置的指针。但是如果我们不使用潜在的内存，事情就不能很好的工作了。

一个复杂的例子

好吧,让我们一步一步地看下这个复杂的程序：

```
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	名字	值
2的30次方		20
...	...	...
2	j	0
1	i	2的30次方
0	h	3

为 j，i，h 分配内存。i 在堆内，所以它有一个指向那儿的值。

接下来，在 main() 结束时调用 foo()：

地址	名字	值
2的30次方		20
...	...	...
5	z	4

4	y	10
3	x	0
2	j	0
1	i	2的30次方
0	h	3

给 x，y 和 z 分配空间。参数 x 和 j 有相同的值，因为这就是我们在其中传递的。它是一个指向地址 0 的指针，因为 j 指向 h。

接下来，foo() 调用 baz()，传递 z：

地址	名字	值
2的30次方		20
...	...	...
7	g	100
6	f	4
5	z	4
4	y	10
3	x	0
2	j	0
1	i	2的30次方
0	h	3

给 f 和 g.baz() 分配内存，这个内存占用空间不大，所以当它结束时，我们可以释放其栈帧：

地址	名字	值
2的30次方		20
...	...	...
5	z	4
4	y	10
3	x	0
2	j	0
1	i	2的30次方
0	h	3

接下来。foo() 调用 bar()，参数为 x 和 z：

地址	名字	值
2的30次方		20
2的30次方−1		5
...	...	...

10	e	9
9	d	2的30次方-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的30次方
0	h	3

最后，我们堆上分配另一个值，因此我们必须从 2 的 30 次方减去 1。即 1,073,741,823。在任何情况下，我们像往常一样设置变量。

bar() 结束时,它调用 baz():

地址	名字	值
2的30次方		20
2的30次方-1		5
...	...	...
12	g	100
11	f	4
10	e	9
9	d	2的30次方-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的30次方
0	h	3

到这里，我们已经到达了最深点! 看看接下来会怎样?

bar() 结束后，f 和 g 就可以除去了:

地址	名字	值
2的30次方		20

2的30次方−1		5
...	...	...
10	e	9
9	d	2的30次方−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的30次方
0	h	3

接下来，我们从 bar() 返回。在本例中 d 是 Box 类型的，那么它也释放了它所指向的：2 的 30 次方 −1 位置的内存。

地址	名字	值
2的30次方		20
...	...	...
5	z	4
4	y	10
3	x	0
2	j	0
1	i	2的30次方
0	h	3

之后，foo() 返回：

地址	名字	值
2的30次方		20
...	...	...
2	j	0
1	i	2的30次方
0	h	3

最后是 `main()`，它清除了其余的内存。当 `i` 被 `Drop` 时，它也会清理最后的堆。

其他语言怎么做?

大多数语言默认情况下都有一个垃圾收集器 `heap-allocate` 。这意味着，每个值都是会被封装。这样做有许多原因，但是它们不在本教程的范围内，我们在这里就不详细说明了。这有一些可能的优化，但也做不到 100% 时间的优化。，垃圾收集器能够更好地处理堆内存的问题。

使用哪一个?

栈更快，也更容易管理，那么为什么我们还需要堆呢? 一个很大原因是，栈分配意味着你只能根据后进先出的语义回收存储。而堆分配严格说来更一般化，允许以任意顺序从池中取出或返回存储器，但是其更复杂，成本更高。

一般来说，更倾向于使用栈分配，因此，Rust 默认情况下是栈分配。栈后进先出模型比较简单，也更基础。一般存在两大影响因素：运行时效率和语义影响。

运行时的效率。

栈的内存管理是微不足道的：机器只是增加或减少一个单一的值，即所谓的“栈指针”。堆的内存管理就有点不一般了：堆可以在任意点上分配或释放内存，并且堆上分配的内存块可以是任意大小的，内存管理器的日常工作必然更难，从而能够识别内存以便重用。

如果你想更详细地深入这个主题,本文可以给出一个极好的介绍。

语义的影响

栈分配影响 Rust 语言本身，以及开发人员的思维模型。后进先出语义指示 Rust 语言如何自动处理内存管理。如在本章所讨论的那样，甚至一个独特的基于堆的回收箱也可以通过基于栈的后进先出语义驱动。非后进先出语义的灵活性（即表现力）意味着：一般情况下，编译器在编译时无法自动推断应该释放那些内存；它必须依赖于动态协议（可能来自语言本身之外）驱动回收，引用计数器（Rc 和 Arc 所使用的）就是其中的一个例子。

在往更深层次说呢，逐渐增长的堆分配表达力主要来自有效的运行时支持(如垃圾收集器的形式)和有效的程序员工作(显式的内存管理形式,不需要 Rust 编译器提供验证)。

测试

程序测试是一个非常有效的方法，它可以有效的暴漏程序中的缺陷，但对于暴漏缺陷来说，这还是远远不够的。
—— Edsger W. Dijkstra，"卑微的程序员" (1972)

让我们来谈谈如何测试 Rust 代码。我们将谈论不是什么测试 Rust 代码正确的方法。关于正确和错误地编写测试的方式有很多的流派。所有这些方法都使用相同的基本工具，因此，我们将向您展示使用它们的语法。

测试属性

Rust 中一个最简单的测试是一个函数，它使用 test 属性注释。让我们使用 Cargo 做一个叫加法器的新项目：

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

当你做一个新项目时，Cargo 将自动生成一个简单的测试。下面即是 src/lib.rs 的内容：

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

注意 `#[test]` 。该属性表明，这是一个测试函数，目前还没有函数体。我们可以使用 Cargo test 运行这个测试：

```
$ 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 编译和运行我们的测试。这里有两组输出：一个用于我们写的测试，另一个用于文档测试。稍后我们将讨论这一问题。现在，让我们来看看这一行：

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

注意 it_works。这是来自我们的函数的名称:

```
fn it_works() {
```

我们还得到一个总结:

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

那么为什么我们的测试能够通过呢?任何非 panic 的测试都可以通过，任何 panic 的测试都会失败。让我们来看一个失败的测试:

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

assert! 一种 Rust 提供的宏，它需要一个参数：如果参数是true，什么也不会发生。如果参数是 false，它就成为 panic! 的。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/src/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 表明我们的测试失败:

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

反映在结论中就是:

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

还可以得到一个非零的状态代码：

```
$ echo $?
101
```

如果你想将 cargo test 集成到其他工具，这是非常有用的。

我们可以用另一个属性：should_panic 转化我们的测试的失败：

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

如果我们 panic！，这个测试会成功，如果我们完成，则测试会失败。让我们来试一试：

```
$ 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
```

Rust 提供另一个宏 assert_eq!，用来比较两个参数是否相等：

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

这个测试是否可以通过?因为存在 should_panic 属性，它可以通过：

```
$ 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 测试很脆弱。很难保证测试不会因为一个意想不到的原因而失败。为了解决这个问题，可以在 should_panic 属性中添加一个可选的参数：expected。测试工具将确保错误消息包含提供的文本。上面示例的安全版本是：

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

这就是所有的基础让我们来编写一个“真正”的测试：

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

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

这是 assert_eq! 的一个非常常见的用法：使用一些已知的参数调用某些函数并与预期的输出比较。

测试模块

有一种方式，以这种方式我们现有的例子都是不符合惯例的：它缺少测试模块。我们的示例的惯用写作方式，如下所示：

```
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));
}
}
```

这里有一些变化。第一个是引入带有 cfg 属性的 mod tests。模块允许我们对所有的测试进行分组，如果需要也可以定义 helper 函数，这个函数不会成为我们 crate 的一部分。如果目前我们试图运行这些代码，cfg 属性只会编译我们的测试代码。这可以节省编译时间，也保证了我们构建的测试是完全正常的。

第二个变化是 use 声明。因为我们在一个内部模块中，我们需要将我们的测试函数设置范围。如果你有一个大的模块，这可能就会很恼人，所以这是 glob 属性的一种常见的使用方式。让我们改变我们的 `src/lib.rs` 以便能够使用它：

```
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 行的不同使用方式。现在，我们来运行我们的测试：

```
$ 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 tests::it_works ... ok

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

   Doc-tests adder
```

```
running 0 tests

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

看，运转起来了!

当前的惯例是使用测试模块 “unit-style” 测试。任何只测试一个小功能都是有意义的。如果用 “integration-style” 测试替代会怎么样呢?为此，我们引出了测试目录。

测试目录

为了编写集成测试，让我们做一个测试目录，并把一个 `tests/lib.rs` 文件放在里面，这是它的内容:

```
extern crate adder;

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

这类似于我们之前的测试，但略有不同。在代码顶部有一个 extern crate adder。这是因为在测试目录里测试是一个完全独立的箱，所以我们需要导入我们的函数库。这也是为什么 tests 是一个编写集成风格测试的合适的地方：他们使用函数库和其他消费者。

让我们运行它们:

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

running 1 test
test tests::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
```

现在我们有三个部分：我们之前的测试在运行，现在这个新的也在运行。

这就是所有的 tests 目录。这里不需要测试模块，因为整件事都是专注于测试的。

让我们最后检查一下第三部分：文档测试。

文档测试

没有什么是比带有示例的文档更好的了。没有什么是比不能真正工作的例子更糟的了，一直以来文档编写已经改变了代码习惯。为此，Rust 支持自动运行你的文档中的示例。这里有一个完整的 `src/lib.rs` 的例子：

```
//! The `adder` crate provides functions that add numbers to other numbers.
//!
//! # Examples
//!
//! ```
//! assert_eq!(4, adder::add_two(2));
//! ```

/// This function adds two to its argument.
///
/// # Examples
///
/// ```
/// use adder::add_two;
///
/// assert_eq!(4, add_two(2));
/// ```
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：模块级文档使用 `//!`，函数文档使用 `///`。Rust 的文档支持 Markdown 中评论，所以三重斜线标志代码块。包含 # Examples 部分是一种惯例，以下所示。

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 tests::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
```

现在我们运行了所有三种测试！注意这些测试文档的名称：the_0 生成模块测试，add_two_0 生成功能测试。当你添加更多的例子，这些名字会自动增量（例如 add_two_1）。

条件编译

Rust 有一个特殊属性 #[cfg]，它允许你编译基于标志的代码并传递给编译器。它有两种形式：

```
#[cfg(foo)]

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

他们也有一些帮助：

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

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

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

这些可以随意嵌套：

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

至于如何启用或禁用这些开关，如果你使用 Cargo，可以在 Cargo.toml 的 [features] 部分 加以设置：

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

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

当你这样做时，Cargo 传递一个标识给 rustc:

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

这些 cfg 标识的总和将决定哪些得到激活，从而致使哪些代码被编译。让我们看看这段代码：

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

如果我们使用 cargo build --features "foo" 编译代码，它将发送 --cfg feature="foo" 标识给 rustc，且输出中包含 mod foo。如果我们定期地使用 cargo build 编译它，也不传递额外的标识，就不会存在任何 foo 模块。

cfg_attr

你也可以使用 cfg_attr 设置另一个基于 cfg 变量的属性：

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

如果 a 使用 cfg 属性设定，和使用 #[b] 是相同的。

cfg!

cfg! 语法扩展允许你在你代码中的任何位置使用这些类型标记：

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

根据配置设置不同，这些在编译时取真或假。

文档

文档是软件项目的重要组成部分，而且它在 Rust 中是第一位的。让我们看看怎么使用 Rust 给出的工具来记录你的项目。

关于 rustdoc

Rust发行版包含一个工具 rustdoc，它可以生成一个文档。rustdoc通常也被 Cargo 通过 cargo doc 来使用。

文档可以通过两种方式生成：从源代码生成，或者从独立的 Markdown 文件生成。

源代码文档制作

记录一个 Rust 项目的主要方式是通过注释源代码。为此可以使用文档注释：

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

这段代码生成如下所示的文档。在这里只保留了它本来位置的常规的注释，而省去了实现。关于这个注释的第一件需要注意的事：它使用 `///`，而不是 `//`。即三重斜线表示文档的注释。

在 markdown 中书写文档的注释。

Rust 跟踪这些注释，并在生成文档时使用它们。当记录一些诸如枚举时，这是重要的：

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

上面的代码会正常执行，但是下面的代码不会正常执行：

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

这时你会得到一个错误:

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

这个 unfortunate error 是正确的：文档注释适用于紧随其后的内容，但是不能出现在语句的最后。

编写文档注释

不管怎样，让我们先详细了解每个部分的注释:

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

第一行的文档注释应该是功能的一个简短的摘要。一个句子，只需要是最基本的，高水平的。

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

我们最初的例子只是一行注释，但如果我们有更多的事情需要说明，我们可以在一个新的段落添加更多的解释。

特殊部分

```
/// # Examples
```

接下来是特殊部分。这些以一个头 # 表示。有三种常用的头文件。他们没有特殊的语法，现在只是有约定俗成的用法，

```
/// # Panics
```

在Rust中，一个函数出现不可恢复的错误(即编程错误)通常由 panics 表示，它会至少结束整个当前线程。如果你的函数有这样一个重要的约定，它可以被 panics 检测/执行，记录就变得非常重要。

```
/// # Failures
```

如果你的函数或方法返回一个Result<T, E>，这个结果描述了现在的状况，如果返回 Err(E)，表示现在状况还不是很坏。这和 Panics 相比就显得稍微不那么重要了，因为错误已经被编码，但它仍然不是一件特别坏的事。

```
/// # Safety
```

如果你的功能是 unsafe 的，你应该向调用者解释哪些不变量是负责维护的。

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

第三，Examples。包括一个或多个使用你的函数或方法的例子，你的用户会因为这些函数或方法而爱你。这些例子都包含了代码块注释，我们将讨论这部分：

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

```
/// assert_eq!(v, vec!["abc", "def", "ghi"]);
/// ```
```

让我们讨论一下这些代码块的细节。

代码块说明

使用三重斜线对 Rust 代码进行注释：

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

如果你想要某些代码不是 Rust 代码，您可以添加一个注解：

```
/// ```c
/// printf("Hello, world\n");
/// ```
```

根据你选择的任何语言突出显示这些内容。如果你只想显示纯文本，只需要选择 text 就可以了。

选择正确的注释是很重要的，因为 rustdoc 以一个有趣的方式使用注释：实际上它可以用来测试你的例子，所以，他们离不开日期。如果你有一些C代码，但 rustdoc 认为它是 Rust，因为你没有使用注释，当试图生成文档 rustdoc 会产生异常。

文档测试

让我们讨论一下文档样例示例：

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

你会注意到，你不需要一个 fn main() 或任何东西。rustdoc 会在你的代码中的正确位置自动添加一个 main()函数。例如：

```
/// ```
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ```
```

这是最终的测试：

```
fn main() {
    use std::rc::Rc;
```

```
    let five = Rc::new(5);
}
```

这是 rustdoc 使用后处理例子的完整算法：

- 任何主要 #!(foo) 属性是完好的 crate 属性。
- 一些常见的 allow 属性被插入，包括 unused_variables ，unused_assignments，unused_mut，unused_attributes，dead_code。小例子经常会引发这些麻烦。
- 如果样例不包含 extern crate，那么 extern crate 就会被插入。
- 最后，如果样例不包含 fn main，文本的其余部分会包装在fn main() { your_code }。

然而，有时这还是不够的。例如，所有这些代码示例中，我们用 `///` 来标注我们在说什么，原始文本：

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

输出看起来有些不同：

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

是的，这是正确的：你可以以 # 开始添加注释，输出时这些注释将被隐藏，编译代码时这些注释将会被使用。你可以利用这个优势。在这种情况下，文档注释需要适用于某种功能，所以如果我想给一个文档注释，我就需要添加一个小函数来定义它。同时，它只是为了满足编译器，所以隐藏它会使样例更加清晰。您可以使用这种方法来解释更详细的例子，同时仍然保留文档的可测试性。例如，这段代码：

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

下面呈现一个解释：

首先，我们将 x 设置为 5：

```
let x = 5;
```

接下来，我们将 y 设置为 6：

```
let y = 6;
```

最后，我们打印 x 和 y 的总和：

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

这是在原始文本中的相同的解释：

```
First, we set x to five:

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

Next, we set y to six:

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

Finally, we print the sum of x and y:

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

通过重复例子的所有部分，您可以确保您的例子仍然可以编译，而只显示跟你的解释相关的那部分。

记录宏

这里有一个宏的例子:

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

你需要注意三件事：添加自己的 extern crate 行，这样我们就可以添加 #[macro_use] 属性。第二，我们需要添加自己的 main() 函数。最后，明智地使用 # 注释掉这两个东西，确保他们不会显示在输出中。

运行文档测试

运行测试

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

cargo test 也可以对嵌套文档进行测试。然而，cargo test只能测试函数库而不能测试二进制工具。这是由于 rustdoc 的工作方式：链接库进行测试，但如果是二进制的，就没有什么可链接的。

当测试你的代码时，这还有几个有用的注释，可以帮助 rustdoc 做正确的事情：

```
/// ```ignore
/// fn foo() {
/// ```
```

ignore 指令告诉 Rust 忽略代码。这是最通用的。相反，如果它不是代码，可以考虑使用 text 注释，或者使用 #s 得到一个工作示例，这个工作示例只显示你关心的部分。

```
/// ```should_panic
/// assert!(false);
/// ```
```

should_panic 告诉 rustdoc 应该正确地编译代码，但测试实例实际上并没有通过。

```
/// ```no_run
/// loop {
/// println!("Hello, world");
/// }
/// ```
```

no_run 属性将编译代码，但不运行它。这是很重要的例子，例如 “Here's how to start up a network service,” 这表示您想要确保编译，但可能以无限循环的模式运行!

记录模块

Rust有另外一种文档评论 `//!` 。这个评论不记录下一个项目，而是记录封闭的项目。换句话说：

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

    // ...
}
```

`//!` 最常见的用途：用于模块文档。如果你在 foo.rs 之内有一个模块，你经会经常打开它的代码，并且看到这个：

```
//! 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 505href="https://github.com/rust-lang/rfcs/blob/master/text/0505-api-comment-conventions.md")，得到完整规范的风格和格式文档。

其他文档

所有这一切行为都工作于 non-Rust 源文件。因为评论都写在 Markdown 中，他们经常都是 .md 文件。

当你在 Markdown 文件中编写文档时，不需要使用评论给文档加前缀。例如：

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

只需要

```
# Examples

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

let five = Rc::new(5);
```
```

这些代码存在于 Markdown 文件中。不过有一个问题：Markdown 文件需要一个这样的标题：

```
% The title

This is the example documentation.
```

注意：加 % 的行需要是文件的第一行。

文档属性

在更深的层面，文档注释是文档属性的装饰：

```
/// this

#[doc="this"]
```

如下是相同的：

```
//! this

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

你不会经常看到这个属性用于编写文档，但当改变一些选项，或者当编写一个宏是，它可能是有用的。

双出口

rustdoc 将显示一个文档为了在两个位置得到一个公共再出口：

```
extern crate foo;

pub use foo::bar;
```

因为 bar 且在文档内部因为 crate foo 而创建文档。它将在两个地方使用相同的文档。

这种行为可以被 no_inline 禁止：

```
extern crate foo;

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

控制 HTML

你可以通过 #!(doc) 的属性版本控制 rustdoc 产生 HTML 的若干部分：

```
#![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/")];
```

这是一个不同选项的集合，包括标志，标识，和一个根 URL。

生成选项

rustdoc 还包含一些其他命令行选项，为了进一步用户化：

- html-in-header FILE：在 <head>…</head> 的结尾部分，包括文件的内容。
- html-before-content FILE：在 <body> 之后，呈现的内容之前(包括搜索栏)，包括文件的内容。
- html-after-content FILE：在所有呈现内容之后，包括文件的内容。

安全事项

Markdown 中放置的文档注释，没有被处理成最终的网页。小心文字的 HTML：

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

迭代器

下面我们来探讨一下循环问题。

还记得 Rust 的 for 循环吗？下面有一个例子：

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

现在你已经知道了更多的 Rust，我们可以详细谈谈它是如何工作的。范围 (0 . . 10) 是一个迭代器。我们可以使用 .next() 方法反复调用迭代器，它给出了事情的一个序列。

如下所示：

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

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

我们针对范围给出一个可变的绑定，这就是迭代器。然后用一个内在的 match 进行 loop 。用这个 match 操作 range.next() 的结果，这就给出了到迭代器的下一个值的一个引用。next 返回一个 Option，在这种情况下，一旦循环运行完毕，我们会得到一个值： Some(i32) 或者 None。如果我们得到 Some(i32)，就打印出来，如果我们得到 None，就跳出循环。

这个代码示例和我们的 for 循环版本基本上是一样的。for 循环仅仅是编写loop/match/break 构造的一个方便的方式。

然而，for 循环不是唯一使用迭代器的情况。编写自己的迭代器包括实现**迭代器**的特征。虽然这种操作不是在本指南的范围之内，Rust 提供了许多有用的迭代器来完成各种任务。在我们谈论这些之前，我们应该谈论一下 Rust 反模式。这就是范围的使用方式。

是的，我们刚刚谈到范围很有用。但范围也很原始。例如，如果你需要遍历一个 vector 的内容，你可能会这样写：

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

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

这比使用一个真正的迭代器严格的多。你可以直接对 vector 进行迭代，像下面写的这样：

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

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

这样做有两个原因。首先，这可以更直接的表达我们的意思。我们遍历整个 vector，而不是遍历索引，然后通过索引访问 vector。第二，这个版本是更高效的：第一个版本将有额外的边界检查，因为它使用了索引、num

s[i]。但是因为我们使用迭代器依次针对 vector 的每个元素产生一个引用，在第二个示例中不涉及边界检查。这对于迭代器是很常见的：我们可以忽略不必要的检查范围，但仍知道我们是安全的。

关于 println! 怎样工作，这里还有一个细节不是 100% 的清楚。num 实际上是 &i32 类型的数字。也就是说，它是对 i32 的一个引用，而不是本身就是 i32，println! 为我们处理非关联化的事物，所以我们不能看到它的细节。下面这段代码同样工作正常：

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

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

现在我们来明确非关联化 num。为什么 &num 可以给我们引用？首先，因为我们明确地使用 & 调用。其次，如果给我们数据本身，我们必须是它的拥有者，它将生成一个数据的副本，并将副本给我们。与引用相比，我们只是借用了引用数据，所以它只是传递一个引用,而不需要做移动。

所以，既然我们已经认定范围往往不是你想要的，让我们来谈谈你想要的东西。

这里主要有3个类，它们是彼此相关的事物：迭代器（iterators），迭代器适配器（iterator adapters），和消费者（consumers）。下面给出一些定义：

- 迭代器给出序列值。
- 迭代器适配器使用一个迭代器，产生一个新的迭代器，它拥有不同的输出序列。
- 消费者使用一个迭代器，产生一些值最后的设置。

首先让我们来谈谈消费者，因为你已经看到一个迭代器。

消费者

consumer 操作一个迭代器，返回某些类型的值。最常见的 consumer 是 collect()。下面的代码并没有完全编译，但它已经显示了意图：

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

正如你所看到的，我们可以在我们的迭代器上调用 collect()。collect() 尽可能多地接收迭代器给它的值，并返回结果的集合。为什么这不能编译呢？Rust 不能确定你想收集什么类型的值，所以你需要让它知道。下边是编译的版本：

```
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，所以我们可以告诉它，我们需要一个整数向量。尽管你并不总是需要使用整个类型。使用一个 _ 可以允许你提供部分提示：

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

即：“请收集 Vec，但为我推断出T是什么。”因为这个原因 _ 有时被称为一种“占位符”。

collect() 是最常见的 consumer，但也存在其他的消费者。find()就是其中之一：

```
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 消耗一个闭包，针对迭代器的每个元素的引用操作。如果元素是我们要找的元素，这个闭包返回 true，否则返回 false。因为我们可能找不到一个匹配的元素，find 会返回一个 Option 而不是元素本身。

另一个重要的消费者是 fold。下面就是 fold 的示例：

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

fold() 是一个消费者，语法：fold(base, |accumulator, element| ...)。它需要两个参数：第一个是一个称为 基 的元素 。第二个是一个闭包，本身有两个参数：第一个被称为累加器，第二个是一个元素。在每次迭代中，调用闭包，结果是在下一次迭代中累加器的值。在第一次迭代中，base 是累加器的值。

好吧，这有点令人困惑。让我们看看在这个迭代器中所有事物的值：

基	累加器	元素	闭包结果
0	0	1	1
0	1	2	3
0	3	3	6

我们使用这些参数来调用 fold() 函数:

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

所以，0 是基，sum 是累加器，x 是我们的元素。在第一次迭代中，我们将 sum 设置为 0，x 是 nums 的第一个元素 1。然后将 sum 和 x 相加，即 0 + 1 = 1。第二次迭代中，和值成为我们的累加器 sum，元素是数组的第二个元素 2，相加，即 1 + 2 = 3 ，这样就得到了最后一次迭代的累加器的值。在这次迭代中，x 是最后一个元素 3 ，相加，即 3 + 3 = 6，这就是求和最后的结果。1 + 2 + 3 = 6，这就是我们最后得到的结果。

对于 fold 如果你刚开始接触它，可能会觉得这种语法有点奇怪，但一旦开始使用它，您会发现它的使用范围很广，几乎到处都可以使用。任何时候，如果你有一系列的事物，而你想要一个单一的结果，fold 都是最适当的。

由于迭代器存在另一个我们还没有谈到属性：懒惰，消费者就变得尤为重要。让我们多谈论一些关于迭代器的问题，你就会明白消费者为什么如此重要。

迭代器

正如之前说过的，我们可以使用 .next() 方法反复调用一个迭代器，它给出了一个事情的序列。因为你需要调用这个方法，这意味着迭代器可以偷懒，而不是预先生成的所有值。例如，在这段代码中，实际上并没有生成 1–100 的数字，它仅仅代表了一个序列，而不是产生一个值：

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

因为我们没有针对范围做任何事情，它不会生成序列。下面让我们加入消费者：

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

消费者 collect() 要求范围给它一些数字，这样它才会做生成序列的工作。

您将看到范围是两个基本的迭代器之一。另一种是 iter()。iter() 可以把一个 vector 变成一个简单的迭代器，反过来这个迭代器给出每个元素：

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

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

这两种基本迭代器应该能很好地为你服务。有一些更高级的迭代器，包括那些不限范围的。

这里谈论到的迭代器已经足够你使用。迭代器适配器是我们需要谈论的关于迭代器最后的概念。

迭代器适配器

迭代器适配器获得一个迭代器，以某种方式对其进行修改，产生一个新的迭代器。最简单的一个叫做 map：

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

map 被另一个迭代器调用，产生一个新的迭代器，在新的迭代器中，每个元素引用迭代器给出的关闭作为调用它的参数。这将打印出 2–100 的数字。如果你编译这个示例，您会得到一个警告：

```
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);
 ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

迭代器的懒惰又起作用了！这个闭包永远不会执行。这个例子也不会打印任何数字:

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

如果你只是为了测试其副作用，而在一个迭代器上执行一个闭包，那只需要使用 for 就好了。

有很多有趣的迭代器适配器。take(n) 将在原迭代器的 n 个元素的基础上返回一个新的迭代器。注意，这个对于原迭代器没有副作用。让我们使用一下之前提到的无限迭代器：

```
for i in (1..).step_by(5).take(5) {
    println!("{}", i);
}
```

这将打印出

```
1
6
11
16
21
```

filter() 是一个以一个闭包作为参数的适配器。这个闭包返回 true 或者 false。新的迭代器 filter() 产生唯一的元素，闭包返回true：

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

这将打印 1 到 100 之间的所有的偶数。(注意：因为 filter 不消耗将遍历的元素，它只是传递每一个元素的引用，从而可以使用 &x 模式过滤谓词来提取整数本身。)

现在你可以将三件事放在一起考虑：首先是一个迭代器，经过几次调整，然后消耗这个结果。检查一下:

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

这将给你一个包含 6、12、18、24 和 30 的向量。

这只是一个关于迭代器，迭代器适配器，和消费者的小的尝试。有很多非常有用的迭代器，您也可以编写您自己的迭代器。迭代器提供一个安全、有效的方式来操作各种列表。起初你会觉得它们有点不同寻常，但一旦你开始使用它们，你就会迷上它们。关于迭代器和消费者的不同点的完整列表，你可以查阅迭代器模块文档。

并发性

并发性和并行性在计算机科学中是非常重要的主题，即使在当今工业中也是个热门的话题。电脑得到了越来越多的核心，然而，很多程序并没有能力来利用它们。

Rust 内存安全特性同样采用了并发的方式。甚至 Rust 程序内存必须是安全，没有数据之间的竞争。Rust 的类型系统的任务就是给你强大的方式让程序能够在编译时并发执行。

之前我们谈论过 Rust 的并发特性，需要理解的重要的是: Rust 足够低级别的,所有这些都是由标准库提供的，而不是语言本身。这意味着，如果你不喜欢 Rust 某些方面处理并发性的方式，你可以自己实现另一种做事的方式。mio 是一个在现实中践行这一原则的例子。

后台：Send 和 Sync

并发性是很难说清楚的。在 Rust 中，我们有一个强大的、静态类型系统来帮助使我们的代码可理解。因此，Rust 提供给了我们两个特性来帮助我们理解可以并发执行的代码。

Send

我们将谈论第一个特性是 Send。当类型 T 实现 Send 时，它告诉编译器，这种类型的线程拥有在线程之间安全转移的所有权。

强行添加一些限制是很重要的。例如，如果我们有一个通道连接两个线程，我们将希望能够向通道中发送一些数据，接着将这些数据传送给另外一个线程。因此，我们要保证要被发送的类型实现了 Send。

相反地，如果我们利用 FFI 封装一个库，然而它不是线程安全的，此时我们不想实现 Send，所以编译器会帮助我们强制它不能离开当前线程。

Sync

第二个特性被称为 Sync。当一个类型 T 实现 Sync 时，它告诉编译器，但这种类型被使用在多个线程并发时不可能引起内存不安全的状态。

例如,与一个原子索引计数器共享不可变的数据是线程安全的。Rust 提供了一种类似 Arc< T > 类型，并且它实现了 Sync，所以它在线程之间共享是安全的。

这两个特性将会让你在并发的情况下，你所使用类型系统的对代码的属性做出强有力的保证。在说明为什么能够这样保障之前，我们需要首先学习如何创建一个并发的 Rust 程序。

线程

Rust 的标准库为线程提供了一个库，它允许你以并行的方式运行 Rust 代码。这里是使用 std::Thread 的一个简单例子:

```
use std::thread;

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

thread:spawn() 方法接受一个封闭参数，这个参数会在一个新线程中执行。它返回该线程的句柄，它可以用来在等待子线程完成之后提取其结果:

```
use std::thread;

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

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

许多语言有能力执行线程,但普遍的是非常不安全的。可以写整本书讲关于如何防止在共享可变状态的情况下发生错误。Rust 通过他的类型系统在编译期阻止数据的竞争来解决这个问题。接下来让我们看看如何在线程之间共享的数据。

安全共享可变状态

由 Rust 的类型系统,我们可以有一个概念,尽管听起来不切实际:“安全共享可变状态。“许多程序员都同意共享可变状态是非常，非常糟糕的。

有人曾经说过：共享可变状态是一切罪恶的根源。大多数语言尝试解决这个问题通过“可变”部分,但 Rust 处理是通过解决“共享”的部分来解决这个问题。

同一所有权系统有助于防止错误的使用指针，同时也有助于排除数据竞争，这个是并发执行时最糟糕的一种 bug。

如下是一个 Rust 程序，它会在许多语言中存在数据竞赛。它将不会编译通过:

```
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);
}
```

会输出如下的错误:

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

在这种情况下，我们知道我们的代码应该是安全的，但是 Rust 不确定。其实实际上它并不安全:如果我们在每个线程中都引用 data，那么每个线程都会有一个自己索引数据的权限，那么同一个数据有三个所有者！那是不好的。我们可以通过使用 `Arc<T>` 类型来解决这个问题，它是一个原子引用计数器数指针。“原子”意味着它跨线程共享是安全的。

`Arc< T >` 假定一个或更多关于其内容的属性，以确保它在跨线程共享是安全的：它假定其内容拥有 Sync 属性。但在我们的例子中，我们希望能够修改变量的值。我们需要一种类型能够确保一次只能有一个用户能够修改变量值。为此，我们可以使用 `Mutex< T >` 类型。这是第二个版本的代码。它仍然不能正常工作,但是由于其他的原因:

```
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);
}
```

这里存在如下的错误：

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

你可以发现，Mutex 有一个 lock 方法，它的函数声明如下：

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

因为 Send 没有实现 `MutexGuard < T >`，我们不能越过线程范围进转换，这就是错误的原因。

我们可以用 `Arc< T >` 来解决这个问题。如下是可以工作的版本:

```
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);
}
```

现在我们可以在 Arc 中调用 clone() 方法，它会增加内部计数。这个句柄接着就会跳转到一个新的线程中进行执行。让我们更仔细地查看线程的主体代码：

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

```
    data[i] += 1;
});
```

首先，我们调用 lock() 函数获得互斥锁的锁。它将返回 `Result< T， E >`，而且这只是一个例子，由于这个函数可能会失败，因此我们使用 unwrap() 函数来得到引用的数据。真正的代码在这里会有更健壮的错误处理代码。当我们得到锁之后，我们就可以随意的修改变量了。

最后，在线程运行时，我们等待了一会。但这不是理想：我们可以选择一个合理的时间等待，但更有可能我们会是等待的时间比必要的时间长或者还要短，这取决于线程运行时执行计算花费的实际时间。

更精确的计时器的替代品是使用 Rust 标准库中提供的线程同步方法中的一个机制。接下来让我们谈谈其中一个的机制：通道。

通道

如下是使用 channel 进行线程同步的一个版本，而不是等待一个特定的时间:

```
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() 方法来构造一个新的通道。我们只是发送一个简单的 () 到通道中，然后等待十次执行后返回。

虽然这通道只是发送一个通用的信号，但我们可以向通道中发送任何 Send 类型的数据！

```
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 是 Send 类型，因此我们可以复制。所以我们创建一个线程，让它来计算答案，然后使用 send() 通过通道向我们发送答案。

异常

panic！ 将当前执行的线程中断。您可以用一个简单的隔离机制执行 Rust 的线程:

```
use std::thread;

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

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

线程返回给我们一个结果，这让我们能够检查线程是否发生了异常。

错误处理

> 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

有时候,事情会出乎意料的发生错误。重要的是要提前想好应对错误的方法。Rust 有丰富的支持错误处理方法来应对可能(老实说:将会)发生在您的程序中的错误。

主要有两种类型的错误可能发生在你的程序中:故障和异常。让我们谈谈两者之间的区别，然后讨论如何处理它们。接着，将讨论如何将故障升级为异常。

故障 VS 异常

Rust 使用两个术语来区分两种形式的错误:故障和异常。故障是可以用某种方式中恢复的错误。异常是一种不能恢复的错误。

我们说的 ” 恢复 “ 是什么意思？嗯，在大多数情况下，指的是预计一个错误的可能性。例如，考虑 parse 函数:

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

这个方法将一个字符串转换成另一种类型。但因为它是一个字符串,你不能确保转换工作正常执行。例如，执行如下的转换会得到什么？

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

这是行不通的。所以我们知道，这个函数只会对一些特定的输入才能正常工作。这是预期行为。我们称这种错误为故障。

另一方面，有时，有意想不到的错误，或者我们不能恢复它。一个典型的例子是一个断言：

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

我们使用 assert! 说明参数是正确的。如果这不是正确的，那么这个断言就是错误的。错误的话，我们不就能继续在当前状态往下执行了。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));
}
```

它将会输出如下的错误：

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

尽管我们已经涵盖所有我们知道的可能情况情况，但是 Rust 不清楚。Rust 不知道概率是 0.0 和 1.0 之间。所以我们添加另一个例子：

```
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));
}
```

我们不应该得到 _ 情况，所以我们使用 unreachable!() 宏来说明这个。unreachable!() 比结果给出了不同于 Result 类型的错误。Rust 称这些类型的错误为异常。

利用 Option 和 Result 处理错误

表明一个函数可能会失败的最简单方法是使用 option< T >类型。例如，字符串 find 方法试图找到字符串的一个模式串，并返回 Option：

```
let s = "foo";

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

对这些简单的情况下是可以的，但是在故障的情况下并不会给我们提供很多的信息。如果我们想知道为什么函数发生了故障，怎么办?为此，我们可以使用 Result<T， E>类型。它看起来像这样：

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

这个枚举类型由 Rust 本身提供，所以你不需要在你的代码中定义就可以使用它。Ok(T) 变量代表着成功执行，Err(E) 变量代表着执行失败。推荐在大多数情况下返回一个 Result而不是一个 Option 变量。

如下是一个使用 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);
    }
}
```

这个函数使用枚举类型变量 ParseError 列举各种可能发生的错误。

调试特点就是让我们使用 {:?} 格式来打印该枚举变量的值。

遇到 panic! 类型的不可恢复错误

遇到为意料的和不可恢复的的错误时，宏 panic! 会引起异常。这将崩溃当前线程，并给出一个错误:

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

当你运行时会输出:

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

因为这些类型的情况相对较少见，很少使用恐慌。

升级故障为异常

在某些情况下，即使一个函数可能发生故障，我们仍想要把它当作一个异常对待。例如，`io::stdin().read_line(&mut buff)` 函数在读取某一行时出现错误会返回 `Result< usize >` 变量。这使我们能够处理它并可能从错误中恢复。

如果我们不想处理这个错误，而宁愿只是中止程序，那么我们可以使用 unwrap() 方法:

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

如果 Result 变量值是 Err，unwrap() 方法将会产生调用 panic！，输出异常。这基本上是说“给我变量的值，如果出现错误，就让程序崩溃。“这相对于匹配错误并试图恢复的方式可靠性较低，但也大大缩短执行时间。有时,只是崩溃程序是合理的。

有另一种方式比 unwrap() 方法好一点:

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

ok() 方法将 Result 转换成一个 Option，并 expect() 和 unwrap() 方法做的事是一样的，不同点在于它需要一个参数，用来输出提示信息。这个消息被传递到底层的 panic！，如果代码出现错误,它提供一个较好的错误消息展示方式。

使用 try!

当编写的代码调用很多的返回 Result 类型的函数时，错误处理就变得比较冗长。try! 宏利用堆栈对产生的错误进行引用从而隐藏具体的细节。

将如下的代码:

```
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 = 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(());
}
```

替换成：

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

用 try！ 封装一个表达式,在成功执行时给 Result 赋值为 Ok，否则赋值为 Err，在这种情况下，在函数未执行完成之前就会返回 Err 值。

值得注意的是，你只能在返回 Result 的函数中使用 try！，这意味着您不能在 main() 中使用 try！，因为 main() 不返回任何值。

try! 利用 From 来确定在错误的情况下返回的值。

外部函数接口

引言

本指南将使用 snappy 压缩/解压库作为引言来介绍编写绑定外部代码。Rust 目前无法直接调用 c++ 库，但是 snappy 包括 C 的接口(记录在 snappy-c.h)。

下面是调用外部函数的一个例子，如果你的机器安装了 snappy 它将能够编译通过:

```
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 语句块中包含的是外部库中函数签名列表，在这个例子中调用的是平台的 C ABI。#[link(...)] 属性用于指示链接器对 snappy 库进行连接，从而保证库中的符号能够被解析。

外部函数被假定为不安全的，所以当调用他们时，需要利用 unsafe{ } 进行封装，进而告诉编译器被调用的函数中包含的代码是安全的。C 库经常暴露不是线程安全的接口给外部调用，而且几乎任何携带指针参数的函数对于所有的输入都不是有效的，因为这些指可能悬空，并且未经处理的指针可能指向 Rust 内存安全模型之外的区域。

当声明外部函数的参数类型时，Rust 编译器不会检查声明是正确的，所以正确地指定它是在运行时能够正确的绑定的一部分。

extern 块可以扩展到覆盖整个 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;
}
```

创建一个安全接口

原始 C API 需要经过封装之后提供内存安全性，并且才可以使用更高级的概念类似向量。库可以选择只暴露安全、高级接口而隐藏不安全的内部细节。

封装那些使用 slice::raw 模块来访问缓冲区的函数，从而将其当作指针来操作 Rust 的向量。Rust 的向量是内存中的一块连续的区域。向量的长度指的是其中包含元素个数的长度，向量的容量指的是在分配的内存中元素的总大小。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
    }
}
```

上述 validate_compressed_buffer 封装器使用了不安全的语句块，但它保证对于所有的输入在离开那个 unsafe 函数签名的时候是安全的。

snappy_compress 和 snappy_uncompress 函数更复杂，因为必须要分配一个缓冲区来保存输出数据。

snappy_max_compressed_length 函数可以通过指定最大所需容量用来分配向量空间，接着用该向量来保存输出。向量接着可以被传递到 snappy_compress 函数作为输出参数。输出参数也会被传递，这样通过设置长度之后被压缩的数据的实际长度也可以得到。

```
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 保存未压缩的大小作为压缩格式的一部分，snappy_uncompressed_length 能够返回所需的确切缓冲区大小。

```
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
        }
    }
}
```

供参考，这里使用的例子也可以在 GitHub 库 里面查看。

析构函数

外部库经常更换被调用代码资源的所有权。当这种情况发生时，我们必须使用 Rust 提供的析构函数来提供安全保证的释放这些资源(特别是在恐慌的情况下)。

想要了解更多的析构函数，请查看 Drop trait

Rust 函数调用 C 代码进行回调

一些外部库需要使用回调函数来报告给调用者他们的当前状态或中间数据。可以通过 Rust 中定义的传递函数与外部库进行通信。当调用 C 代码时，要求回调函数必须使用 extern 标记。

回调函数可以通过注册器发送给调用的 C 库，之后就可以被调用。

一个基本的例子如下：

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
}
```

在这个例子中 Rust 的 main() 函数将调用 C 语言中的 trigger_callback() 函数，接着会在 C 语言中反过来调用 Rust 中的 callback() 函数。

针对 Rust 对象的回调

前面的例子显示了如何在 C 代码中如何回调一个全局函数。然而通常这个回调是针对于 Rust 中某个特定的对象。这个对象可能相应的由 C 对象封装之后的对象。

这个可以通过利用传递一个不安全的指针给 C 库来实现。接着 C 库能够在通知中包含 Rust 对象的指针。此时，允许不安全的访问 Rust 索引对象。

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

在前面例子中给出的是直接调用外部 C 库中提供的函数进行回调。当前线程的控制会从 Rust 转向 C 接着转向 Rust，接着执行回调，最后，触发回调的被调用的函数会在同一线程中执行。

当外部库生成自己的线程，并调用回调时情况就变得更加的复杂。在这些情况下，在回调函数内使用 Rust 中的数据结构是特别不安全的，而且必须使用适当的同步机制。除了经典的同步机制，例如互斥，Rust 中提供了一种可行的方式是使用管道(std::comm)，它会将数据从调用回调的 C 线程中转发到 Rust 中的线程。

如果异步回调的目标是 Rust 地址空间中的一个特殊对象，那么在对象的 Rust 对象被销毁之前在 C 库中肯定不会有更多回调会执行。这个可以通过在对象的析构函数中解除回调关系，并且设计该库确保在正确执行完解除注册之前不会有回调执行。

链接

extern 块中的 link 属性提供给 rustc 基本构建块，告诉它如何链接到本地库。有两种可接受的 link 编写形式：

- `#[link(name = "foo")]`
- `#[link(name = "foo", kind = "bar")]`

在这两种情况下，foo 是它要连接到本地库的名称，而在第二种情况中 bar 是编译期连接到本地库的类型。目前有三个已知的本地库类型：

- 动态 – `#[link(name = "readline")]`

- 静态 – `#[link(name = "my_build_dependency", kind = "static")]`
- 框架 – `#[link(name = "CoreFoundation", kind = "framework")]`

注意，框架类型仅仅对 OSX 目标平台可用。

不同的 kind 值是为了区分本地库如何不同的进行连接。从连接的角度来看，Rust 编译器创建两种构件：部分(rlib/staticlib)和最终(dylib/binary)。本地动态库和框架属于最终构件范围，而静态库不属于。

如下是几个例子关于如何使用这个模型的：

- 本地构建依赖。有时候在编写 Rust 代码是需要使用一些 C/C++ 代码，但是分布的 C / C++ 代码库格式只是一个负担。在这种情况下,代码将被归档到libfoo。然后锈箱将声明一个依赖通过 `#[link(name = "foo", kind = "static")]` 。

不管输出箱的味道,本机静态库将被包含在输出中,这意味着分配本机静态库是不必要的。

- 一个正常的动态依赖关系。常见的系统库(比如readline)都可以在大量的系统,而且经常无法找到这些库的静态副本。当这种依赖是包含在铁锈箱,部分目标(如rlibs)不会链接到库,但当rlib包含在最终的目标(如二进制),本地库将联系在一起。

在 OSX,框架的行为作为一个动态库相同的语义。

不安去的语句块

一些操作，比如引用不安全指针或调用已经被标明为不安全的函数时只允许在不安全的区域内进行。不安全的区域隔离危险，并且向编译期保证不会溢出不安全区域。

不安全的函数，另一方面，必须要显式的表明出。不安全的函数如下所示：

```
unsafe fn kaboom(ptr: *const i32) -> i32 { *ptr }
```

这个函数只能从一个不安全的区域中调用或被其他不安全的函数调用。

访问外部全局变量

外部 API 经常导出全局变量，这样可以做一些类似于跟踪全局状态的事情。为了访问这些变量，你在 extern 语句块中声明他们时要使用关键字 static：

```
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 i32);
}
```

或者，您可能需要使用外部接口来改变全局状态。为了做到这一点，在声明他们时使用 mut，这样就可以修改他们了。

```
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，所有与 static mut 类型的变量交互是不安全的，包括读和写。为了处理全局可变状态你需要多花点心思。

外部调用约定

大多数外部代码暴露了 C ABI，并且 Rust 默认情况下调用外部函数时使用的是 C 平台调用约束。一些外部函数，尤其是 Windows API，使用的是其他调用约定。Rust 提供了一种方法来告诉编译器它使用的是哪个约定：

```
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;
}
```

下面的适用于整个 extern 块。Rust 中支持的ABI 约束列表如下：

- stdcall
- aapcs
- cdecl
- fastcall
- Rust
- rust-intrinsic
- system
- C
- win64

上面列表中的大部分 abis 是不需要解释的，但 system 这个 abi 可能看起来有点奇怪。这个约束的意思是选择与任何与目标库合适的 ABI 进行交互。例如，在 win32 x86 体系结构中，这意味着 abi 将会选择 stdcall。然而在 `x86_64` 中，windows 使用 C 调用协定，因此将会使用 C 的标准。也就是说，在前面的例子中，我们可以在 extern 中使用 “system”{...} 来定义 所有 windows 系统中的块，而不仅仅是 x86 的。

与外部代码的交互

只要 `#[repr(C)]` 这个属性应用在代码中，Rust 保证的 struct 的结构与平台的表示形式是兼容的。`#[repr(C、包装)]` 可以用来布局 sturct 的成员没而不需要有填充元素。`#[repr(C)]` 也适用于枚举类型。

Rust 中的 `boxes(Box<T>)` 使用非空指针作为句柄指向其中所包含的对象。然而，他们不应该手动创建的，因为它们是由内部分配器管理。引用可以安全地假定指针非空指向该类型。然而，破坏 borrow 的检查或易改变的改规则不能保证是安全的，所以如果有必要请使用原始指针(*)，因为编译器不能对他们呢进行过多的假设。

向量和字符串共享相同的基本的内存布局，并且可以通过 vec 和 str 模块与 C APIs 进行交流。然而，字符串不是以 `\0` 作为它的结束符。如果你想要使用一个空终结符字符串与 C 语言的交互，此时你应该使用 std::ffi 模块中的 CString 类型。

标准库包括类型别名和函数的定义，对于 C 标准库位于 libc 模块中，而且 Rust 默认情况下已经链接了 libc 和 libm 库。

可空指针优化

某些类型的定义不为空。这包括引用类型（&T、&mut T），boxes(Box<T>)，和函数指针（extern "abi" fn()）。当与 C 交互时，经常使用的指针可能为空。特殊的情况下，泛型枚举中仅仅包含两个亮亮，其中一个不包含数据，另一个包含单个字段，这个能够进行空指针优化。当这个枚举类型被一个非空类型初始化时，它就表示一个指针，并且那个没有数值的变量就成为空指针。因此，Option<extern "C" fn(c_int) -> c_int> 展示了一个表示空函数指针是如何使用 C ABI。

C 语言中调用 Rust 代码

你可能想要在 C 中调用 Rust 代码，并且编译。这也好似相当容易，但是需要几件事：

```
\#[no_mangle]
pub extern fn hello_rust() -> *const u8 {
    "Hello, world!\0".as_ptr()
}
```

extern 让这个函数符合 C 调用函数的约束，就如上面说的“外部函数调用约束”。no_mangle 属性关闭了 Rust 的名称纠正，因此这里是很容易的进行连接的。

Borrow 和 AsRef

Borrow 和 AsRef 特性是非常相似的，但是也有些区别。这里有一个简单回顾一下这两个特质是什么意思。

Borrow

Borrow 特性是当你写一个数据结构时，并且你想要使用一个 owned 或 borrowed 类型作为用于某些目的的同义词。

例如，HashMap 的 get 方法就使用了 Borrow:

```
fn get<Q: ?Size>(&self, k:&Q) -> Option<&V>
  where K:Borrow<Q>,
      Q:Hash +Eq
```

这个签名是相当复杂的。在这里我们感兴趣的是 K 参数。它指的是 HashMap 本身的一个参数:

```
struct HashMap<K, V, S = RandomState> {
```

K 参数是 HashMap 中使用的 key 类型。所以，再次看 get() 函数签名，当 key 实现了 Borrow“ 时我们可以使用 get() 函数。这样，我们可以使用 String 类型作为 HashMap 的 key，而使用 &strs 进行搜索:”

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

let mut map = HashMap::new();
map.insert("Foo".to_string(), 42);

assert_eq!(map.get("Foo"), Some(&42));
```

这是因为标准库中已经为 String 类型实现了 Borrow 接口。

对于大多数类型，当你想要一个使用 owned 或者 borrowed 类型，使用 &T 就足够了。但在不止一种 borrowed 值时，Borrow 处理起来是很高效的。Slice 指的是一块区域：既可以是 &[T]也可以是 &mut[t] 类型。如果我们想使该区域同时存在着两种类型，就可以使用 Borrow:

```
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);
```

上面的代码会输出 a is borrowed: 5 两次。

AsRef

AsRef 是转换特征。它被用在通用的代码中将一些值转换成索引类型。比如：

```
let s = "Hello".to_string();

fn foo<T: AsRef<str>>(s: T) {
    let slice = s.as_ref();
}
```

应该用哪一个?

我们可以看到他们是如何的相似：他们都处理 owned 和 borrowed 版本的数据类型。然而，他们还是有点不同。

当你想要抽象出各种 borrowing 数据中的不同，或者你创建的数据结构用同样的方式对待 owned 和 borrowed 值，比如哈希操作和比较操作，此时你应该选择 Borrow。

当你想要直接转换某些类型为引用类型，并且你在编写泛型时，你应该选择 AsRef。

发布通道

Rust 项目使用一个叫做“发布渠道”的概念来管理版本的发布。理解这个过程从而决定你的 Rust 程序应该选择哪个版本是很重要的。

综述

Rust 中的发布有三种：

- Nightly
- Beta
- Stable

Nightly 版本指的是每天更新一次。每隔六周，Nightly 版本会晋升为 “Beta” 版。在这一点上，它只会收到补丁修复严重错误。六周后，Beta 版提升为 “Stable” 版，并成为下一个 1. x 版本。

这个过程是平行地发生的。所以每隔六周的同一天，Nightly 变成 Beta，Beta 变成 Stable。当 1.x 版本发布时，与此同时，1.(x + 1)−beta 也被发布，Nightly 版本变成第一个 1.(x + 2)−nightly 版。

选择一个版本

一般来说，除非你有一个特别的原因，你应该使用stable 发行版。这些版本是针对大众的。

然而，Rust 中那取决于你的兴趣，你可以选择使用nightly 版本。基本的权衡是这样的：在 nightly 通道中，您可以使用不稳定，新的 Rust 特性。然而，不稳定的特性易于改变，所以新的 nightly 版本发布时可能会破坏你的代码。如果你使用 stable 版本，你不能使用实验特性，但 Rust 的下一版本重大的变化不会造成严重的问题。

通过 CI 帮助构建生态系统

beta 版怎么样？我们鼓励所有使用 stable 版通道的 Rust 用户也去在他们集成的系统中尝试下 beta 版通道。这将有助于提醒团队，以防有意外情况发生。

另外，测试 nightly 版本能够更快的捕获复位情况，并且，如果你不介意第三个构建，我们推荐你测试下所有通道。

5

语法和语义

这一章节将 Rust 的知识划分成小块，每一块描述一个概念。

如果你想从下至上的学习 Rust，按照顺序阅读本教程是个比较好的方法。

这些部分也为每个概念生成了索引，所以如果你阅读另一个教程，发现让你困惑的，你也可以在这里找到关于它的解释。

变量绑定

几乎所有非 “Hello World” Rust 程序使用变量绑定。如下的形式：

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

在每个例子中都写 fn main(){ 显得有点冗余，所以在以后我们会省略掉它。如果你一直跟着教程学习，确保修改你的 mian() 函数，而不是丢掉了它。否则，你的程序编译会得到一个错误。

在许多语言中，这被称为一个变量，但 Rust 中的变量绑定有一些小窍门需要注意。例如让表达式的左边是一个 “pattern”，而不仅仅是一个变量名。这意味着我们可以这样做:

```
let (x, y) = (1, 2);
```

这个表达式求值后，x 将会被赋值为 1，y 将会被赋值为 2。Pattern 是非常的强大，在本教程中有专门的章节讲解。我们暂时不需要这些功能，所以我们就把暂时不关注这个，继续学习本节的知识。

Rust 是一种静态类型语言，这意味着我们要提前指定类型，并且在编译时期被检查。那么，为什么我们的第一个例子能够被编译？Rust 有个称为 “类型推断” 的机制。如果它能推断出变量是什么类型，Rust 就不需要指定实际类型。

如果想要我们可以添加它的类型。在冒号(:)之后输入类型：

```
let x: i32 = 5;
```

如果我要求你将上面的大声读给班上的其他同学，你会说 “x 被绑定为 i32 类型，并且它的值为5。”

在这种情况下，我们选择将 x 表示为一个 32 位带符号整数。Rust 有许多不同的基本的整数类型。以 i 开头的表示有符号整型，u 开头的表示无符号整型。可能的整数尺寸是 8、16、32、64 位。

在以后的例子中，我们可能会在注释中说明其类型。以后的例子会看起来像这样:

```
fn main() {
  let x = 5; // x: i32
}
```

注意使用 let 的语法和注释表示的是相似的。和让您所使用的语法。Rust 中包含上面的注释方式，但它不是通用的方式，因此我们会偶尔的使用上面的注释方式来帮助你理解变量表示 Rust 中的实际类型。

默认情况下，绑定是不可变的。这段代码将不会编译通过:

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

它将会输出如下的错误：

```
error: re-assignment of immutable variable `x`
    x = 10;
    ^~~~~~~
```

如果你想绑定是可变的，您可以使用 mut 关键字:

```
let mut x = 5; // mut x:i32
x = 10;
```

没有原因说明默认绑定是不可变的，但我们可以通过考虑 Rust 关注的一个要点：安全性。如果你忘了什么 mut，编译器会捕捉到它，让你知道你修改了某些值但是你可能没有声明该值为 mut。如果绑定默认情况下是可变的，那么编译器将无法告诉你这一点。如果你的确想修改变量值，那么解决方案很简单:添加 mut。

还有其他理由避免出现可变状态，但他们超出本指南的范围。一般来说，通常可以避免显式变化，那也是 Rust 所推荐的。也就是说，有时，修改变量是你需要的，但是它不是禁止的。

让我们回到绑定。Rust 变量绑定与其他语言有一个方面是不同：在你能够使用绑定变量之前你要初始化该变量。

让我们来自己动手试一试。修改 src/main.rc 文件，使其中的内容像下面的一样：

```
fn main() {
    let x: i32;

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

你可以在命令行中使用 cargo build 命令对该文件进行编译 。尽管你会得到一个警告，但是仍然会打印 ”Hello, world！ “：

```
 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 default
src/main.rs:2     let x: i32;
                      ^
```

Rust 警告我们从来没有使用这个变量绑定，但是因为我们从来没有使用它，没有破坏性，没有恶意。然后，如果我们试图使用变量 x 事情就会发生变化。让我们来试一下。改变你的程序为如下的样子:

```
fn main() {
   let x: i32;
```

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

尝试编译上面的代码。将会看到输出一个错误：

```
$ 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 不会让我们使用未初始化的变量。接下来，让我们谈谈添加到 println！ 中的东西。

如果在你打印的字符串语句中包含花括号({})，Rust 将这个视为请求插入某种值。字符串插入值是一个计算机科学名词，它的意思是“粘在字符串的中间。“我们添加一个逗号，然后是 x，表明我们希望插入的是 x 的值。如果传递给函数和宏的参数不止一个，我们用逗号对参数进行分隔。

当你仅仅只是使用花括号，Rust 通过检查它的类型将尝试以一种有意义的方式显示它的值。如果你想用更详细的方式指定打印格式，请查看多个可选参数可用。此时，我们只关注默认值：整数打印起来并不是很复杂。

函数

每个 Rust 程序至少包含一个函数，也就是 main 函数：

```
fn main() {
}
```

这可能是最简单的函数声明。正如我们前面所提到的，fn 表明“这是一个函数”，紧随其后的是函数名字，一些括号，因为这个函数没有参数，然后一些花括号来表示函数体。这里有一个函数名称为 foo 的函数：

```
fn foo() {
}
```

那么，函数的带参数是什么样的呢？如下是打印一个数字的函数：

```
fn print_number(x: i32) {
    println!("x is:{}", x);
}
```

如下是使用 print_number 函数的完整程序：

```
fn main() {
   print_number(5);
}

fn print_number(x: i32) {
    println!("x is: {}", x);
}
```

正如你所看到的，函数参数的编写方式和 let 声明很相似：您可以在冒号之后添加参数的类型。

如下是一个将两个数加起来然后打印的完整程序：

```
fn main() {
    print_sum(5, 6);
}

fn print_sum(x: i32, y: i32) {
    println!("sum is: {}", x + y);
}
```

当你调用和声明函数的时候，都是利用逗号分隔参数。

不像 let，你必须声明函数参数的类型。如下的代码并不会正常工作：

```
fn print_sum(x, y) {
  println!("sum is: {}", x + y);
}
```

将会输出如下错误：

```
 expected one of `!`, `:`, or `@`, found `)`
fn print_number(x, y) {
```

这是一个深思熟虑的设计决策。尽管可以通过推理判断出数据类型，比如 Haskell 语言就拥有这种特性，但通常推荐显式声明是一种最佳实践。我们赞成强制函数声明类型推这种方式，尽管能够在函数内部能够推断出变量的类型，这种方式对于指出全文推断和支持推断来说都是是个不错方式。

返回值是什么样的呢？如下这个函数给整型参数进行加 1 操作：

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

Rust 函数仅仅只能返回一个值，并且在一个“箭头”符号之后声明返回值类型，该“箭头”由是一个破折号(-)和大于符号(>)组成。函数的最后一行决定它的返回值是什么。您会注意到这里函数最后一行缺少分号。如果我们给它加上分号：

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

我们会得到一个错误：

```
error: not all control paths return a value
fn add_one(x: i32) -> i32 {
   x + 1;
}

help: consider removing this semicolon:
   x + 1;
       ^
```

这揭示了 Rust 两个有趣的事情：它是一门基于表达式的语言，并且分号与其他基于花括号和分号的语言中的分号是不同的。这两件事是相关的。

表达式 VS 语句

Rust 是一门主要基于表达式的语言。它只有两种类型的语句，其他的都是表达式。

那么有什么区别？表达式返回一个值，和但语句并不会。这也就是为什么我们以“并不是所有的控制路径返回一个值”结尾，在这里语句 x + 1；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”。其他的都是一个表达式。让我们先谈谈声明语句。

在一些语言中，变量绑定可以写成表达式的形式，而不仅仅是语句。像 Ruby：

```
x = y = 5
```

然而在 Rust 中使用 let 引起的绑定不是一个表达式。以下将会产生编译时错误：

```
let x = (let y = 5); // expected identifier, found keyword 'let'
```

编译器告诉我们在这里它希望看到一个表达式的开始，而 let 只能声明一个语句，而不是一个表达式。

注意，给一个已经绑定变量(如 y = 5)赋值仍然是一个表达式，虽然它的值不是特别有用。不像其他语言，赋值语句的值为被赋值的值(例如前面例子中的 5)，在 Rust 中赋值语句的值是一个空元组 ()：

```
let mut y = 5;
let x = (y = 6); //x has the value '()', not '6'
```

Rust 中的第二种语句是表达式语句。它的目的是把任何表达式变成一个语句。实际上，Rust 的语法希望在语句之后还是语句。这意味着您可以使用分号来将表达式分隔开。这意味着 Rust 看起来很像大多数其他语言，需要你在每一行的末尾使用分号，而且你看到的所有 Rust 代码几乎都是以分号作为行结束符。

是什么例外让我们说“几乎”？其实你已经看到它，如下这段代码中：

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

函数声明返回一个 i32 类型的值，但如果语句末尾是分号，它将返回 ()。Rust 意识到这可能不是我们想要的，于是正如在前面我们看到错误中建议删除分号。

提前返回

如果出现提前返回呢？Rust 中的确有一个 return 关键字：

```
fn foo(x: i32) -> i32 {
    return x;

    // we never run this code!
    x + 1
}
```

利用 return 关键字仍然会将函数的最后一行返回，但是这被认为是糟糕的编程风格：

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

如果你以往没有使用过基于表达式的语言，那么前面没有使用 return 的定义可能看起来有些奇怪，但是使用的时间长了就会变得很自然了。

发散函数

Rust 对于 ‘发散函数’有一些特殊的语法，该函数不返回任何值：

```
fn diverges() -> ! {
    panic!("This function never returns!");
}
```

panic! 是一个宏，和 println!() 类似，我们已经看见过。与 println!() 不同的是 panic!() 会导致当前执行线程的崩溃并给出特定的消息。

因为这个函数会导致崩溃，它永远不会返回，所以它有 “!” 类型，读“发散”。发散函数可以作为任何类型：

```
let x: i32 = diverges();
let x: String = diverges();
```

基本类型

Rust 的语言有很多被认为是基本的数据类型。这意味着他们是语言内置的。Rust 是结构化的语言，并且标准库在这些类型至上提供了一些有用的其他的类型，但是这些是最基础的类型。

Booleans

Rust 拥有内置的 boolean 类型，名称为 bool。这种类型的变量能够被赋值为 true 或者 false：

```
let x = true;

let y: bool = false;
```

booleans 比较类型常用的方式是在 if 条件中。

在标准库文档说明中查看更多的关于 bool 的说明信息。

char

char 类型代表的是一个 Unicode 标量值。你可以使用单引号来创建 char 类型变量：

```
let x = 'x';
let two_hearts = '?';
```

不像其他语言，这意味着在 Rust 中，char 类型不是单个字节而是由四个字节表示。

同样，你可以在标准库文档中查看更多关于 char 的说明。

数值类型

Rust 中有各种数值类型，可以分为这几类：有符号数和无符号数、固定长度和可变长度、浮点数和整数。

这些类型包括两个部分：类别和大小。例如，u16是一个 16 位的无符号类型。更多的比特位能够表示更大的数字。

如果一个数字不能从它字面值推断出他的类型，那么就会默认如下：

```
let x = 42; // x has type i32
let y = 1.0; // y has type f64
```

下面列出了不同的数值类型，同时连接到标准库中的文档说明:

- i8
- i16
- i32
- i64
- u8
- u16
- u32
- u64
- isize
- usize
- f32
- f64

接下来让我们去看看他们的类别:

有符号和无符号数

整数类型有两种：有符号和无符号。为了理解他们的差异，首先让我们看一个四位长度的数字。对于有符号数，四位数字能够表示的数据范围是 −8 ~ +7。有符号数用二进制补码表示。对于无符号的四位数字，因为它不需要存储负值，所以可以存储从 0 到 +15。

无符号类型使用 u 表示数据的类别，而有符号数使用 i 表示。i 指的是“整数”。所以 u8 表示的是无符号的 8 位整数，i8 表示的是有符号的 8 位整数。

固定大小类型

固定大小的类型由特定数量的比特位数表示。有效位数是 8、16、32、64 位。u32 表示的是无符号 32 位整型数，而 i64 表示的是有符号的 64 位数。

可变大小类型

Rust 还提供了大小取决于底层机器指针的大小的类型。这些类型根据大小分为不同的类别，同样有有符号和无符号的类型。比如这两种类型：isize 和 usize。

浮点类型

Rust 也有两种浮点类型：f32 和 f64。这些符合于 IEEE-754 单、双精度浮点数的标准。

数组

像许多编程语言一样，Rust 有许多类型能够表示顺序对象。最基本的是数组，相同类型的元素和固定大小的顺序对象。默认情况下，数组元素是不可变的。

```
let a = [1, 2, 3]; // a: [i32; 3]
let mut m = [1, 2, 3]; // m: [i32; 3]
```

数组有 [T；N] 类型。我们将在泛型部分谈论这个 T 符号。N 是一个编译时常量，表示数组的长度。

有一个快捷的方式用相同的值初始化数组的每个元素。在这个例子中，每个元素将被初始化为0：

```
let a = [0; 20]; // a: [i32; 20]
```

你可以利用 a.len() 得到数组中元素的个数：

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

println!("a has {} elements", a.len());
```

你可以使用数组下标访问数组中某个特定的元素：

```
let names = ["Graydon", "Brian", "Niko"]; // names: [&str; 3]

println!("The second name is: {}", names[1]);
```

下标从 0 开始，就像在大多数编程语言中，第一个元素是 names[0]，第二个元素是 names[1]。上面的示例输出 the second name is: Brian。如果您尝试使用一个不在数组范围内的下标，你会得到一个错误：在运行期访问数组下标越界。这样的访问错误是其他许多编程语言程序中 bug 的来源。

你可以在标准库文档中找到更多关于数组的说明。

切片

“切片”指的是对另一个数据结构的索引(或“视图”)。他们是用于允许安全，高效的访问数组的一部分而不需复制数组的内容。例如，您可能只是想索引文件中某一行并将其读入内存中。从本质上说，切片不是直接创建的，而是来自于现有的变量。切片拥有长度，是可变的也可以设置不可变，并且在许多方面像数组是相似的：

```
let a = [0, 1, 2, 3, 4];
let middle = &a[1..4]; // A slice of a: just the elements 1, 2, and 3
let complete = &a[..]; // A slice containing all of the elements in a
```

切片有 &[T] 类型。我们将会在泛型部分讲解 T。

你可以在标准库文档中找到更多关于切片的内容。

str

Rust 的 str 类型是最基本的字符串类型。作为一种无固定大小类型，它本身不是很有用，但是当将其使用在引用符号之后它会会变得作用更大，类似于 &str。这里我们不会进一步讲解这个。

你可以在标准库文档中找到更多关于 str 的说明。

元组

元组是有固定大小的有序列表。类似于：

```
let x = (1, "hello");
```

括号和逗号构成了长度为 2 的元组。如下是相同的代码，但是有类型说明：

```
let x: (i32, &str) = (1, "hello");
```

正如你所看到的，元组的类型看上去和元组一样，但每个位置都有一个类型名称而不是元素值。细心的读者会注意到，元组是异构：在该元组中有一个 i32 类型和一个 &str 类型的元素。在系统编程语言中，字符串比其他语言稍微更复杂点。当下，我们把 &str 当作字符串切片，之后我们会了解更多的。

如果两个元组有相同的元素类型和相同的参数数量，那么你可以将一个元组赋值给另外一个。,当元组有相同的长度时，他们具有相同的参数数量。

```
let mut x = (1, 2); // x: (i32, i32)
let y = (2, 3); // y: (i32, i32)

x = y;
```

你可以使用解析符来访问元组中的字段。如下：

```
let (x, y, z) = (1, 2, 3);

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

记得以前我们曾说过左边操作符 let 语句比赋值绑定有强的能力吗？现在我们就可以解释了，我们可以在左边的 let 之后编写一个 pattern，如果它和右边的语句相匹配，那么我们就可以同时进行多个元素的赋值绑定。在这种情况下，let “析构”或者 “分解” 元组，接着进行三个元素的赋值。

这种模式是非常强大的，我们将会在后面的章节再次看到。

你可以在括号中使用逗号来消除是单个元素还是不是单元素的二义性。

```
(0,); // single-element tuple
(0); // zero in parentheses
```

元组索引

你还可以使用索引的方式访问元组的字段：

```
let tuple = (1, 2, 3);

let x = tuple.0;
let y = tuple.1;
let z = tuple.2;

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

类似于数组索引，下标从零开始，但与数组索引不同的是它使用一个 . 而不是 [] 进行访问。

你可以在标准库文档中找到更多的关于元组的说明。

函数

函数同样也是一个类型！如下所示：

```
fn foo(x: i32) -> i32 { x }

let x: fn(i32) -> i32 = foo;
```

在这种情况下，x 是带一个 i32 类型的参数和返回值为 i32 类型的函数指针。

注释

既然我们对函数有了一定了解之后，那么学习下如何写注释是不错的。注释的作用在于它能够帮助其他的程序员更好的理解你的代码。而编译期通常会忽视他们。

Rust 中有两种你应该学习的注释方式：行注释和文档注释。

```
// 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 //，并且在该注释部分中支持 markdown 注解文法：

```
/// Adds one to the number given.
///
/// # Examples
///
/// ```
/// let five = 5;
///
/// assert_eq!(6, add_one(5));
/// ```
fn add_one(x: i32) -> i32 {
    x + 1
}
```

在编写文档注释时提供一些程序使用的例子是非常有用的。你会注意到在这里我们使用了一个新的宏： assert_eq!。它将两个值进行比较，如果两个值不相等就会调用 panic!。这是非常有用的文档说明。还有另一个宏，assert!，如果它的参数值是 false，那么同样会触发 panic！ 的执行。

您可以使用 rustdoc 工具从这些文档注释生成HTML文档，并运行测试代码来尝试一下!

if

Rust 提供的 if 语法不是特别复杂，但它比起传统的系统语言更像动态类型语言中提供的 if。接下来让我们谈论它，以确保你掌握 Rust 中 if 的微妙之处。

if 是“分支”特定形式的更一般的概念。这个名字来自于树中的一个分支：在一个决策点有多条路径可以选择，当我们依赖于某个条件可以选择其中一个分支。

if 通常情况是一个选择会有两条路径:

```
let x = 5;

if x == 5 {
    println!("x is five!");
}
```

如果我们改变了 x 的值，上面那一行不会打印。更具体地说，如果 if 后面的表达式的求值结果为 true，那么它后面的语句块会被执行。如果是 false，那么该语句块不会被执行。

如果你想要在 if 后面表达式是 false 时发生什么，你可以使用一个 else 语句：

```
let x = 5;

if x == 5 {
    println!("x is five!");
} else {
    println!("x is not five :(");
}
```

如果有一个以上的情况下，可以使用 else if 语句：

```
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 :(");
}
```

这都是非常标准的语法结构。然而，你也可以这样做：

```
let x = 5;

let y = if x == 5 {
    10
} else {
    15
}; // y: i32
```

我们可以(而且应该)这样写：

```
let x = 5;

let y = if x == 5 { 10 } else { 15 }; // y: i32
```

上面的代码会正常执行，因为 if 是个表达式。表达式的值是被选择的那个分支中最后一个表达式的值。并且如果 if 语句后面没有 else 语句，通常会将 () 作为语句执行的结果。

for 循环

for 循环被用来循环执行代码特定次数。然而 Rust 的 for 循环与其他系统语言稍微有些区别。Rust 的for循环看起来不像如下 “C” 风格的 for 循环：

```
for (x = 0; x < 10; x++) {
    printf( "%d\n", x );
}
```

相反，它看起来像这样：

```
for x in 0..10 {
    println!("{}", x); // x: i32
}
```

在更抽象的术语中，

```
for var in expression {
    code
}
```

上面的表达式是一个迭代器。迭代器提供一系列的元素。每个元素是迭代的一次循环。接着迭代器的值被绑定到变量 var，它是循环体控制循环的主体。一旦循环体执行结束，就从迭代器中获取下一个，接着执行下一次循环。当迭代器中没有更多的值可以获取的时候，for 循环结束。

在我们的示例中，0..10 是一个表达式，说明了开始和结束的位置，并给出一个迭代器遍历在这些值。上界是不包含在内的，所以我们的循环将打印 0 到9，而不是10。

Rust 故意没有按照 “C 风格” 设计 for 循环。因为手动的控制循环中每个元素对于熟练使用 C 语言的开发者来说不仅是是复杂的而且容易出错。

While 循环

Rust 中也有 while 循环。如下代码所示：

```
let mut x = 5; // mut x: i32
let mut done = false; // mut done: bool

while !done {
x += x - 3;

println!("{}", x);

if x % 5 == 0 {
done = true;
}
}
```

当你不能确定你需要循环多少次的时候 while 循环是一个不错的选择。

如果你需要一个无限循环，你可能尝试如下这样写：

```
while true {
```

然而，Rust 有一个专用的关键字， loop ，来解决这个问题：

```
loop {
```

Rust 的控制流分析相对于 while true，在处理此构建上有所不同，因为我们知道它将永远循环。一般情况下，我们给编译器的信息越多，编译器越能够更好的处理安全和代码生成问题，所以在你打算实现无限循环时，你应该首选 loop。

提早结束循环

让我们看看早些时候我们有的 while 循环：

```
let mut x = 5;
let mut done = false;

while !done {
x += x - 3;

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

```
if x % 5 == 0 {
done = true;
}
}
```

我们不得不使用专用的 mut 布尔变量绑定， done，用来让我们知道什么时候应该退出循环。Rust 有两个关键字来帮助我们修改循环： break 和 continue。

在这种情况下，我们可以通过使用 break ，来用一种更好的方式书写循环：

```
let mut x = 5;

loop {
x += x - 3;

println!("{}", x);

if x % 5 == 0 { break; }
}
```

现在我们使用 loop 来实现无限循环，同时使用 break 来提早退出循环。

continue 是相似的，但是它不是结束循环，而是转到下一次循环。以下是只打印奇数的代码：

```
for x in 0..10 {
if x % 2 == 0 { continue; }

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

continue 和 break 在 while 循环和 for循环中都有效。

所有权

这篇指南是 Rust 已经存在的三个所有权制度之一。这是 Rust 最独特和最令人信服的一个特点，其中 Rust 开发人员应该相当熟悉。所有权即 Rust 如何实现其最大目标和内存安全。这里有几个不同的概念，每一个概念都有它自己的章节：

- 所有权，即正在读的这篇文章。
- 借用，和与它们相关的功能‘引用’
- 生存期，借用的先进理念

这三篇文章相关且有序。如果你想完全的理解所有权制度，你将需要这三篇文章。

元

在我们了解细节之前，这里有关于所有权制度的两个重要说明需要知道。

Rust 注重安全和速度。它通过许多‘零成本抽象’来完成这些目标，这意味着在 Rust 中，用尽可能少的抽象成本来保证它们正常工作。所有权制度是一个零成本抽象概念的一个主要例子。我们将在这篇指南中提到的所有分析都是在*编译时完成*的。你不用为了任何功能花费任何运行成本。

然而，这一制度确实需要一定的成本：学习曲线。许多新用户使用我们喜欢称之为‘与借检查器的人斗争’，即 Rust 编译器拒绝编译那些作者认为有效的程序的 Rust 经验。这往往因为程序员对于所有权的怎样工作与 Rust 实现的规则不相符的心理模型而经常出现。你可能在第一次时会经历类似的事情。然而有个好消息：更有经验的 Rust 开发者报告称，一旦他们遵从所有权制度的规则工作一段时间后，他们会越来越少的与借检查器的行为斗争。

学习了这些后，让我们来了解所有权。

所有权

变量绑定在 Rust 中有一个属性：它们有它们绑定到的变量的‘所有权’。这意味着当一个绑定超出范围时，将释放它们绑定到资源。例如：

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

当 v 进入范围时，将新创建新的 Vec<T>。在这种情况下，向量也在堆上为三个元素分配空间。当 v 超出 foo() 函数的作用域时，Rust 将清除一切与向量有关的东西，也包括为堆分配的内存。在该作用域结束时，这种情况就一定会发生。

移动语义

尽管这里也有很多微妙的东西：Rust 确保*任何给定的资源都有一个确定的绑定*。例如，如果我们有一个向量，我们可以将它赋值给另一个绑定。

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

let v2 = v;
```

但是，如果我们在之后尝试使用 v 时，我们将发现一个错误：

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

let v2 = v;

println!("v[0] is: {}", v[0]);
```

它会报出如下错误：

```
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，然后在我们已经将它作为参数传递后，然后使用时，类似的情况将会发生：

```
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]);
```

同样的错误：‘移动值使用’。当我们将所有权转移给其他东西时，我们可以说我们已经‘移动’了我们提到的东西。这里你不需要某种特殊注释，它是 Rust 默认做的事情。

详细信息

当我们移动一个绑定后，我们不可以使用这个绑定的原因是微妙的，但是很重要。当我们编写如下代码时：

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

let v2 = v;
```

第一行为向量对象 v 和它包含的内容分配内存。向量对象存放在栈中，同时包含一个指向存放在堆中的内容 ([1, 2, 3]) 的指针。当我们将 v 赋值给 v2 时，它将为 v2 创建一个这个指针的副本。这意味着将会有两个指针指向堆中的向量内容。它将引进数据竞争，这违反了 Rust 的安全保障。因此，Rust 禁止在我们移动后使用 v。

我们需同样注意某种情况下优化可能删除在栈中字节的真正副本。所以它并不像最初看起来的那样毫无效率。

复制类型

在我们将所有权转移到另一个绑定时，我们已经建立了，你不可以使用原来的绑定。然而，这里有一个特性可以改变这种行为，它被称为 Copy 。我们还没有讨论过这个特性，但是现在，你可以把它们看作是增加额外行为的一个特殊类型的一个注释。例如：

```
let v = 1;

let v2 = v;

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

在这种情况下，v 是一个 i32，这实现了 Copy 的特性。这意味着，就像一个移动，当我们将 v 赋值给 v2 时，我们就完成了数据的一个副本。但是，与移动不同，我们在之后仍然可以使用 v。这是因为 i32 在别处没有指向数据的指针，这是一个完整的副本。

我们将在特性章节讨论怎样完成你自己类型的复制。

不仅仅是所有权

当然，如果我们不得不将每个函数的所有权交回，我们可以写如下代码：

```
fn foo(v: Vec<i32>) -> Vec<i32> {
// do stuff with v
```

```
// hand back ownership
v
}
```

这将会特别繁琐。当我们想要取得所有权的东西它将会越糟糕：

```
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);
```

返回值类型，返回的行，以及调用的函数获取方式变的更加复杂。

幸运的是，Rust 提供了一种功能，借用，它能帮助我们解决这个问题。它是下一章节的话题！

引用与借用

这篇指南是 Rust 已经存在的三个所有权制度之一。这是 Rust 最独特和最令人信服的一个特点，其中 Rust 开发人员应该相当熟悉。所有权即 Rust 如何实现其最大目标和内存安全。这里有几个不同的概念，每一个概念都有它自己的章节：

- 所有权，即正在读的这篇文章。
- 借用，和与它们相关的功能‘引用’
- 生存期，借用的先进理念

这三篇文章相关且有序。如果你想完全的理解所有权制度，你将需要这三篇文章。

元

在我们了解细节之前，这里有关于所有权制度的两个重要说明需要知道。

Rust 注重安全和速度。它通过许多‘零成本抽象’来完成这些目标，这意味着在 Rust 中，用尽可能少的抽象成本来保证它们正常工作。所有权制度是一个零成本抽象概念的一个主要例子。我们将在这篇指南中提到的所有分析都是在*编译时完成*的。你不用为了任何功能花费任何运行成本。

然而，这一制度确实需要一定的成本：学习曲线。许多新用户使用我们喜欢称之为‘与借检查器的人斗争’，即 Rust 编译器拒绝编译那些作者认为有效的程序的 Rust 经验。这往往因为程序员对于所有权的怎样工作与 Rust 实现的规则不相符的心理模型而经常出现。你可能在第一次时会经历类似的事情。然而有个好消息：更有经验的 Rust 开发者报告称，一旦他们遵从所有权制度的规则工作一段时间后，他们会越来越少的与借检查器的行为斗争。

学习了这些后，让我们来了解借用。

借用

在所有权章节的末尾部分，我们有一个令人讨厌的功能，如下所示：

```
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);
```

这不是惯用的 Rust，然而，由于它不能利用借出，以下是第一步：

```
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!
```

我们使用了一个引用：&Vec<i32>，而不是将 Vec<i32> 作为我们的参数。同时，我们不是直接传递 V1 和 V2，我们传递 &V1 和 &V2。我们称 &T 为一个‘引用’，它借用了所有权，而不是拥有了资源。借用东西的绑定当它超出作用域时，它不解除分配给它的资源。这意味着在调用 foo() 函数后，我们原始的绑定可以再次被使用。

引用与绑定类似，它们都是不可变的。这意味着，在 foo() 函数中，向量根本不能改变：

```
fn foo(v: &Vec<i32>) {
 v.push(5);
}

let v = vec![];

foo(&v);
```

错误：

```
error: cannot borrow immutable borrowed content `*v` as mutable
v.push(5);
^
```

推入一个值改变了这个变量，所以我们不允许这样做。

&mut引用

这里有第二种引用：&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);
```

以上代码将输出 6。我们将 y 标记为 x 的一个可变引用，然后将 y 指向的内容加 1。你将会注意到 x 也不得不被标记为 mut，如果不被标记，我们不能将一个可变值借用给一个不可变值。

其他方面，&mut 引用与其他引用一样。尽管它们之间以及它们如何相互作用有一个很大的区别。你可以在上面的例子中看到一些可疑的东西，因为我们需要那个用 { 和 } 包围的额外空间。如果我们删除它们，我们将会得到一个错误：

```
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() {

}
^
```

事实证明，这里是有规则的。

规则

这里有关于借用在 Rust 中的规则：

首先，任何借用必须持续比所有者的作用域小。其次，你可能有一个或者两种其他的借用，但是两种不能在同一时间同时使用：

- 一个资源的从 0 到 N 的引用（&T ）。

- 一个可变的引用（ &mut T ）

你可能会注意到这与数据竞争的定义非常相似，尽管并不是完全相同:

```
There is a ‘data race’ when two or more pointers access the same memory location at the same time, where at least o
```

使用引用，由于它们都不编写，所以你喜欢用多少用多少。如果你想要编写，你需要两个或者更多的指针指向同一内存，你每次仅仅可以使用一个 &mut。这就是 Rust 如何在编译时避免数据竞争：如果我们违反了规则，我们将会得到错误。

学习了以上内容后，让我们再次重新考虑我们的例子。

在作用域中的思考

以下为相关代码:

```
let mut x = 5;
let y = &mut x;

*y += 1;

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

这些代码给出我们如下错误:

```
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 &mut T 指向 x ，所以我们不允许创建任何 &T。反之亦然。以下注释暗示我们如何思考这个问题:

```
note: previous borrow ends here
fn main() {

}
^
```

换句话说，这个可变的借用贯穿了我们剩余的示例。我们想要的是在我们尝试调用 printLn! *之前*，可变借用结束，然后使用不变的借用。在 Rust 中，借用被绑定在对于借用有效的作用域内。我们的作用域如下所示:

```
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
```

作用域冲突：我们不能让 &x 和 y 在一个作用域内。

因此我们添加大括号：

```
  let mut x = 5;

{
let y = &mut x; // -+ &mut borrow starts here
*y += 1;// |
}   // -+ ... and ends here

println!("{}", x);  // <- try to borrow x here
```

此时没有问题。我们可变的借用在我们创建不变的借用之前，超出了作用域。但是作用域是我们可以观察到一个借用可以持续多久的关键。

借用问题预防

为什么会有这些预防性的规则？正如我们所指出的，这些规则预防了数据竞争现象。什么样的问题是引起数据竞争现象的原因呢？以下有几个原因。

迭代器失效

其中一个例子是当你尝试改变你正在循环的集合时将会出现的‘迭代器失效’。Rust 的借用检查器防止这种情况的发生：

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

for i in &v {
println!("{}", i);
}
```

以上代码将打印出从 1 到 3 的数字。当我们循环访问这些向量时，我们仅仅给出了这些元素的引用。V 本身作为不可变的借用，这意味着，在我们遍历的过程中我们不能改变它：

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

```
for i in &v {
println!("{}", i);
v.push(34);
}
```

以下是出现的错误：

```
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，因为它在循环中被借用。

释放内存后再使用

引用必须与它们引用到的资源存活时间一样长。 Rust 会检查你的引用的作用域来保证它为真。

如果 Rust 不检查这个属性，我们可能会无意间使用一个无效的引用。例如：

```
let y: &i32;
{
let x = 5;
y = &x;
}

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

我们将会得到以下错误：

```
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 存在的作用域内有效。一旦 x 消失了，它将会变成一个 x 的无效引用。因此，上面代码中的错误中说借用‘活的时间不够长’，因为它在有效的矢量的时间内是无效的。

当在引用到的变量之*前*声明引用时，会发生同样的问题：

```
let y: &i32;
let x = 5;
y = &x;

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

我们会得到以下错误：

```
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);
}
```

生存期

这篇指南是 Rust 已经存在的三个所有权制度之一。这是 Rust 最独特和最令人信服的一个特点，Rust 开发人员应该相当熟悉。所有权即 Rust 如何实现其最大目标和内存安全。这里有几个不同的概念，每一个概念都有它自己的章节：

- 所有权，即正在读的这篇文章。
- 借用，和与它们相关的功能‘引用’
- 生存期，借用的先进理念

这三篇文章相关且有序。如果你想完全的理解所有权制度，你将需要这三篇文章。

元

在我们了解细节之前，这里有关于所有权制度的两个重要说明需要知道。

Rust 注重安全和速度。它通过许多‘零成本抽象’来完成这些目标，这意味着在 Rust 中，用尽可能少的抽象成本来保证它们正常工作。所有权制度是一个零成本抽象概念的一个主要例子。我们将在这篇指南中提到的所有分析都是在*编译时完成*的。你不用为了任何功能花费任何运行成本。

然而，这一制度确实需要一定的成本：学习曲线。许多新用户使用我们喜欢称之为‘与借检查器的人斗争’，即 Rust 编译器拒绝编译那些作者认为有效的程序的 Rust 经验。这往往因为程序员对于所有权的怎样工作与 Rust 实现的规则不相符的心理模型而经常出现。你可能在第一次时会经历类似的事情。然而有个好消息：更有经验的 Rust 开发者报告称，一旦他们遵从所有权制度的规则工作一段时间后，他们会越来越少的与借检查器的行为斗争。

学习了这些后，让我们来了解生存期。

生存期

借出一个对于其他人已经拥有的资源的引用会很复杂。例如，假设这一系列的操作：

- 我获得某种资源的一个句柄。
- 我借给你对于这个资源的引用。

- 我决定我使用完了这个资源，然后释放它，然而你仍然拥有这个引用。
- 你决定使用资源。

你的引用指向一个无用的资源。当资源是内存时，这被称为悬挂指针或者‘释放后再利用’。

要解决此类问题，我们必须确保在第三步后一定不会发生第四步。Rust 的所有权制度通过被称为生存期的一章来实现，生存期用来介绍一个引用的有效的作用域。

当我们有一个函数将一个引用作为参数时，我们可以用隐式和显式两种方式来表示引用的生存期：

```
// implicit
fn foo(x: &i32) {
}

// explicit
fn bar<'a>(x: &'a i32) {
}
```

'a 读‘生存期为 a ’。从技术上讲，每个引用都有一些与之关联的生存期，但是编译器允许你在一般情况下忽略它们。但是在此之前，以下是我们分解显式例子的代码：

```
fn bar<'a>(...)
```

这一部分声明了我们的生存期。这说明 bar 有一个生存期 'a。如果我们有两个引用的参数，以下是相关代码：

```
fn bar<'a, 'b>(...)
```

然后在我们的参数列表中，我们使用我们已经命名的生存期。

```
...(x: &'a i32)
```

如果我们想要一个 &mut 引用，我们可以书写以下代码：

```
...(x: &'a mut i32)
```

如果你将 &mut i32 和 &'a mut i32 比较，它们是相同的，只是在 & 和 mut i32 之间多了一个 'a。我们将 &mut i32 读作‘对于 i32 的一个可变引用’，将 &'a mut i32读作‘对于 i32 的一个生存期为'a 的一个可变引用’。

当你操作结构体时，也需要显式的生存期：

```
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);
}
```

正如你所看到的的，**结构体**也可以有生存期。与函数相似的方式，

```
struct Foo<'a> {
```

声明一个生存期，和以下代码

```
x: &'a i32,
```

使用它。所以，为什么我们在这里需要一个生存期？我们需要确保对 Foo 的任何引用都不能比对它包含的 i32 的引用的寿命长。

作用域的考虑

考虑生存期的一种方式是将一个引用的有效作用域可视化。例如：

```
fn main() {
let y = &5; // -+ y goes into scope
// |
// stuff// |
// |
}  // -+ y goes out of scope
```

在我们的 Foo 中添加:

```
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 在 y 的作用域内存活，所以一切正常。如果它不是呢？这个代码不会有效工作：

```
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
```

正如你所看到的，f 和 y 的作用域比 x 的作用域要小。但是，当我们运行 x = &f.x 时，我们给了 x 一个可以超出其作用域范围的引用。

命名生存期是给这些作用域命名的一种方式。给东西命名是能不能讨论它的第一步。

'static

命名为 ‘static’ 的生存期是一个特殊的生存期。这标志着这种东西具有整个程序的生存期。很多的 Rust 程序员在处理字符串时会第一次遇到 'static：

```
let x: &'static str = "Hello, world.";
```

字符串具有 &'static str 这种类型，是因为这种引用始终存在：它们融入到最终二进制的数据段中。另一个例子是全局变量：

```
static FOO: i32 = 5;
let x: &'static i32 = &FOO;
```

以上代码是将 i32 加入到二进制文件的数据段中，其中 x 是它的一个引用。

生存期省略

Rust 在函数体中支持强大的局部类型推断，但是它在项目签名中禁止允许仅仅基于单独的项目签名中的类型推断。然而，对于人体工学推理来说被称为“生存期省略”的一个非常受限的二级推理算法，对于函数签名非常适用。它能推断仅仅基于签名组件本身而不是基于函数体，仅推断生存期参数，并且通过仅仅三个容易记住和明确的规则来实现，这使得生存期省略成为一个项目签名的缩写，而不是引用它之后隐藏包含完整的本地推理的实际类型。

当我们谈到生存期省略时，我们使用*生存期输入*和*生存期输出* 这两个术语。*生存期输入*是与一个函数的一个参数结合的一个生存期，同时一个*生存期输出*是一个与函数的返回值相结合的一个生存期。例如，以下函数有一个生存期输入：

```
fn foo<'a>(bar: &'a str)
```

以下函数有一个生存期输出:

```
fn foo<'a>() -> &'a str
```

以下函数在两个位置都有一个生存期：

```
fn foo<'a>(bar: &'a str) -> &'a str
```

这里有三个规定：

- 在函数参数中每个省略的生存期都成为一个独特的生存期参数。
- 如果仅仅有一个输入生存期，省略或者不省略，在这个函数的返回值中，这个生存期被分配给所有的省略的生存期。
- 如果这里有多个输入生存期，但是其中之一是 &self 或者 &mut self，这个生存期的 self 被分配给所有省略的生存期输出。

另外，省略一个生存期输出是错误的。

例子

以下列举了生存期省略的函数的一些例子。我们已经将每个生存期省略的例子和它的扩展形式进行了配对。

```
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，可以改变东西的能力，与其他语言相比它在 Rust 中有点不同。可变性的第一个方面是它的非默认状态：

```
let x = 5;
x = 6; // error!
```

我们可以应用 mut 关键字来介绍可变性：

```
let mut x = 5;

x = 6; // no problem!
```

这是一个可变的变量绑定。当一个绑定是可变时，这意味着你可以更改绑定的指向。所以在上面的例子中，不是 x 的值发生变化，而是这个绑定从 i32 更改为其他。

如果你想要更改绑定的指向，你将需要一个可变的引用：

```
let mut x = 5;
let y = &mut x;
```

y 是一个绑定到一个可变引用的不可变值，这意味着你不可以将 y 绑定到其他 (y = &mut z)，但是你可以改变绑定到 y上的东西 (*y=5).细微的差别。

当然，如果你两个都需要，可以如下书写：

```
let mut x = 5;
let mut y = &mut x;
```

现在 y 可以绑定到另一个值，并且这个引用的值可以改变。

注意到 mut 是<模式的一部分是非常重要的，你可以这样编写代码：

```
let (mut x, y) = (5, 6);

fn foo(mut x: i32) {
```

内部和外部可变性

然而，在 Rust 中当我们说有些东西是‘不变’的，这并不意味着它不可以更改：我们说的是它有‘外部可变性’。考虑如下例子，Arc<T>：

```
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。然而我们在这里还没有使用 mut，x 是一个不可变绑定，同时我们没有使用 &mut 5 或者任何东西。所以谁给?

要理解这一点，我们需要回到 Rust 的指导哲学的核心，内存安全，和 Rust 保证的机制，所有权系统，和更具体的，借用：

```
You may have one or the other of these two kinds of borrows, but not both at the same time:

one or more references (&T) to a resource.
exactly one mutable reference (&mut T)
```

所以，这就是‘不变’的真正定义：有两个指针指向内容是否安全? 在 Arc<T> 的例子中，是的：变化完全包含在内部结构本身中。它不是面向用户的。为此，将 &T 传递给 clone()。但是，如果将 &mut T 传递给 clone()，将会成为一个问题：

其它类型，诸如 std::cell 模块中，具有相反的：内部可变性。例如：

```
use std::cell::RefCell;

let x = RefCell::new(42);

let y = x.borrow_mut();
```

RefCell 用 borrow_mut() 函数来将 &mut 引用传递到它包含的东西中。那样不危险吗? 如果那样做会怎样：

```
use std::cell::RefCell;

let x = RefCell::new(42);

let y = x.borrow_mut();
let z = x.borrow_mut();
```

事实上，这将在运行时引起恐慌。这是 RefCell 做的事情：它在运行时保证 Rust 的借用规则，同时如果它们违背了规则时的 panic!。这允许我们能够绕过 Rust 的不变规则的另一方面。让我们先讲讲吧。

字段级可变性

可变性是借用（&mut）或者绑定（let mut）的一个属性。这意味着，例如，你不能有一个结构体既有一些字段可变还有一些不可变：

```
struct Point {
x: i32,
mut y: i32, // nope
}
```

一个结构体在绑定方面的可变性：

```
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>，你可以模仿字段级可变性：

```
use std::cell::Cell;

struct Point {
x: i32,
y: Cell<i32>,
}

let point = Point { x: 5, y: Cell::new(6) };

point.y.set(7);

println!("y: {:?}", point.y);
```

以上代码将打印 y: Cell { value: 7 }。我们已经成功更新 y。

结构体

结构体是创建更复杂的数据类型的一种方式。例如，如果我们做涉及在二维空间中的坐标计算时，我们可能既需要 x 的值，也需要 y 的值：

```
let origin_x = 0;
let origin_y = 0;
```

一个结构体让我们将二者结合成为一个单一的，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);
}
```

这里有许多东西需要讲，所以先在这里打断一下。我们用一个 struct 关键字来声明一个**结构体**，同时 struct 后跟着此结构体的名字。按照惯例，**结构体**以大写字母开头，同时都为驼峰式书写：PointInSpace，而不是Point_In_Space。

我们可以通过 let 来创建一个我们的结构体的实例，像往常一样，但是我们使用一个**关键字：值**的语法风格来设置每个字段。不要求顺序与原始声明的顺序相同。

最后，因为字段需要名称，我们可以通过点符号来访问字段：origin.x。

在默认情况下，结构体中的值像 Rust 中的其他绑定一样都是不可变的。我们使用 mut 关键字来使它们为可变：

```
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);
}
```

以上代码将打印 The point is at (5, 0)。

Rust 在语言层面不支持字段可变性，所以你不能像以下这样书写代码：

```
struct Point {
mut x: i32,
y: i32,
}
```

可变性是绑定的一个属性，而不是结构体本身。如果你习惯于字段级别可变性，第一次用起来会感觉很奇怪，但是它极大的简化了一些东西。它甚至允许您使某东西为可变，但是仅仅适用于短时间内：

```
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
}
```

语法更新

一个**结构体**可以包含 .. 来表明你想要使用一些其他结构体的副本用于某些值。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 };
```

以上代码中给了 point 结构体一个新的 y，但是仍然保持 x 和 z 的原来的值。它也并不一定是相同的**结构体**，当你想要添加新的变量时可以使用这个语法，同时它会复制你没有指定的值:

```
let origin = Point3d { x: 0, y: 0, z: 0 };
let point = Point3d { z: 1, x: 2, .. origin };
```

数组结构体

Rust 有另外一个数据类型就像一个数组与一个结构体的混合，称为“数组结构体”。数组结构体有一个名字，但是它们的字段没有名字:

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

以上代码中的两个结构体不相等，即使它们有相同的值:

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

与数组结构体相比，使用结构体几乎总是比较好。我们可以像如下代码一样书写 Color 和 Point:

```
struct Color {
red: i32,
blue: i32,
green: i32,
}

struct Point {
x: i32,
y: i32,
z: i32,
}
```

现在，我们有真正的名字，而不是位置。好名字固然重要，同时，结构体我们需要有真正的名字。

在一*种*情况下，数组结构体非常有用，尽管，那样一个数组结构体仅包含一个元素。我们称之为‘新型’模式，因为它允许我们创建一个新的类型，不同于其包含的值，并表示它自己的语义:

```
struct Inches(i32);

let length = Inches(10);
```

```
let Inches(integer_length) = length;
println!("length is {} inches", integer_length);
```

正如你所看到的，你可以通过一个非结构化的 let 关键字来提取内在的整数类型，正如通常的数组一样。在这种情况下，let Inches(integer_length) 将 10 赋值给 integer_length。

Unit-like 结构体

你可以定义一个根本没有任何成员的结构体：

```
struct Electron;
```

这样的结构体被称为 ‘unit-like’ ，因为它类似于空数组，()，有时被称为 ‘unit’ 。与数组结构体一样，它定义了一个新的类型。

这种结构体通常对自己没有什么用处（虽然它有时可以作为一个标记类型使用），但是结合其他的功能，它可以变的有用。例如，一个库可能想要要求你创建一个可以实现某些特定特征的结构体来处理事件。如果你没有需要在结构体中存储的数据，你可以只创建一个 unit-like 结构体。

枚举

在 Rust 中**枚举**是一个类型，它表示几个可能变量中的一个数据:

```
enum Message {
Quit,
ChangeColor(i32, i32, i32),
Move { x: i32, y: i32 },
Write(String),
}
```

每个变量可以选择性的有与之关联的数据。定义变量的语法与之前定义结构体的语法类似：你可以有不包含数据的变量（如 unit-like 结构体），已经命名数据的变量，和未命名数据的变量（如数据结构体）。然而，不同于单独的结构体定义，**枚举**是一种类型。枚举的值可以与任何变量匹配。为此，一个枚举有时被称为‘总和类型’：这个枚举的可能值的集合是每个变量的可能值的集合的总和。

我们使用 :: 语法来使用每个变量的名称：它们由**枚举**本身的名字来划分作用域。这允许以下的代码实现:

```
let x: Message = Message::Move { x: 3, y: 4 };

enum BoardGameTurn {
Move { squares: i32 },
Pass,
}

let y: BoardGameTurn = BoardGameTurn::Move { squares: 1 };
```

这两个变量都被命名为 Move，但是由于它们局限于枚举的名称，它们可以无冲突使用。

一个枚举类型的值包含它是哪个变量的信息，不包括与该变量相关联的任何数据。它有时被称为‘标签结合’，因为数据中包括一个‘标签’用来指示数据的类型。编译器使用这些信息来保证你访问枚举中的数据的安全性。例如，你不能简单的就好像它是可能变量之一一样来尝试解构一个值:

```
fn process_color_change(msg: Message) {
let Message::ChangeColor(r, g, b) = msg; // compile-time error
}
```

不支持这些操作可能会看起来相当受限制，但是这是一个我们可以克服的限制。这里有两种方式：通过我们自己实现相等，或者通过模式匹配带有[模式href="http://doc.rust-lang.org/stable/book/match.html)表达式的变量，你将在下一节中学习这些内容。我们对于 Rust 怎样实现平等了解还不够，但是我们会在特性章节学习到。

匹配

通常情况下，一个简单的 if/else 是不足够的，因为你可能有两个以上的选择。此外，条件可能变得相当复杂。Rust 有关键字 match，允许你用更强大的 match 关键字，取代复杂的 if/else 集合。如下所示：

```
let x = 5;

match x {
1 => println!("one"),
2 => println!("two"),
3 => println!("three"),
4 => println!("four"),
5 => println!("five"),
_ => println!("something else"),
}
```

match 采用表达式的形式，然后根据它的值来分支。分支的每个‘臂’都是 val=>expression 的形式。当值匹配时，这个臂的表达式将被执行实现。之所以称之为 match 是因为‘模式匹配’的术语，而这种正是 match 实现的形式。这里有一整章关于模式，包含了这里可能的所有模式。

那么大的优势是什么呢？这里有几个优势。首先，匹配保障‘穷尽性检查’。你是否看到了最后一个下划线 (_)？如果我们删除该下划线，Rust 将会给出我们如下错误：

```
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 是一个整数，Rust 知道它可以有许多不同的值 – 例如，6。然而，如果没有 _，将没有可匹配项，所以 Rust 拒绝编译这段代码。_ 就像一个‘全匹配通配符’。如果其他的匹配项都不能匹配，将会匹配 _ 的分支，由于我们有了全匹配通配符，我们现在对于 x 的每个可能值都有一个匹配项，所以我们的程序将会被成功编译。

match 也是一个表达式，这意味着我们可以直接在一个 let 绑定的右边使用它，或者直接作为表达式使用：

```
let x = 5;

let number = match x {
1 => "one",
2 => "two",
3 => "three",
4 => "four",
5 => "five",
_ => "something else",
};
```

有时它是将某种东西从一种类型转换为另一种类型的好方式。

枚举匹配

match 关键字的另一个重要的用途是处理一个枚举的可能变量：

```
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),
};
}
```

再次，Rust 编译器检查内容详尽，所以它要求你对于枚举的每个变量都有一个匹配的臂。如果你遗漏了一个，它将会给你一个编译时错误，除非你使用 _。

与之前使用的 match 不同，你不可以使用正常的 if 语句来做到这一点。你可以使用可以被看做 match 的一种缩写形式的if let 语句。

模式

模式在 Rust 中颇为常见。我们在变量绑定，match语句和其他地方也使用它们。让我们继续旋风般的学习模式可以做的所有事情!

快速学习：你可以直接匹配文字，同时 _ 充当一个‘任何’的事件：

```
let x = 1;

match x {
1 => println!("one"),
2 => println!("two"),
3 => println!("three"),
_ => println!("anything"),
}
```

以上代码将打印 one。

多个模式

你可以使用 | 来匹配多个模式：

```
let x = 1;

match x {
1 | 2 => println!("one or two"),
3 => println!("three"),
_ => println!("anything"),
}
```

以上代码将打印 one or two。

范围

你可以使用 ... 来匹配值的范围：

```
let x = 1;

match x {
1 ... 5 => println!("one through five"),
```

```
_ => println!("anything"),
}
```

以上代码将打印 one through five。

范围通常在整数和字符的情况下使用：

```
let x = '?';

match x {
'a' ... 'j' => println!("early letter"),
'k' ... 'z' => println!("late letter"),
_ => println!("something else"),
}
```

以上代码将打印 something else。

绑定

你可以使用 @ 将值绑定到名称：

```
let x = 1;

match x {
e @ 1 ... 5 => println!("got a range element {}", e),
_ => println!("anything"),
}
```

以上代码将打印 got a range element 1。当你想要操作数据结构中的一部分的一个复杂匹配时，这将非常有用：

```
#[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),
_ => {}
}
```

以上代码将打印 Some("Steve")：我们已经把内部名称绑定到 a。

如果你使用 @ 和 | 时，你需要确保在模式的每个部分都已经绑定好名称。

```
let x = 5;

match x {
e @ 1 ... 5 | e @ 8 ... 10 => println!("got a range element {}", e),
_ => println!("anything"),
}
```

忽略变量

如果你要匹配一个包含变量的枚举，你可以使用 .. 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."),
}
```

以上代码将输出 Got an int! 。

守卫

你可以通过 if 语句来介绍‘守卫匹配’：

```
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."),
}
```

以上代码将输出 Got an int! 。

ref 和 ref mut

如果你想要获得一个引用，可以使用 ref 关键字：

```
let x = 5;

match x {
ref r => println!("Got a reference to {}", r),
}
```

以上代码将打印出 Got a reference to 5。

在这里，在 match 中的 r 的类型为 &i32。换句话说，在模式中，使用 ref 关键字*创建*一个引用以供使用。如果你需要一个可变引用，ref mut 将会以相同的方式工作：

```
let mut x = 5;

match x {
ref mut mr => println!("Got a mutable reference to {}", mr),
}
```

重构

如果你有一个复合数据类型，诸如结构体，你可以在一个模式中重构它：

```
struct Point {
x: i32,
y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
Point { x: x, y: y } => println!("({},{})", x, y),
}
```

如果我们只关心一部分值，我们不需要给出它们所有的名字：

```
struct Point {
x: i32,
y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
Point { x: x, .. } => println!("x is {}", x),
}
```

以上代码将打印出 x is 0。

你可以在任何成员中做这种匹配，而不仅仅是第一个成员：

```
struct Point {
x: i32,
y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
Point { y: y, .. } => println!("y is {}", y),
}
```

以上代码将打印出 y is 0。

这种‘重构’行为对于任何复合数据类型都有效，比如数组或者枚举。

混合与匹配

这里有很多种不同的方法来匹配东西，它们又可以被混合和匹配，完全取决于你做什么任务：

```
match x {
Foo { x: Some(ref name), y: None } => ...
}
```

模式非常强大。充分利用它。

方法语法

函数很好，但是如果你想要在一些数据上调用很多函数，那是非常不合适的。请思考以下代码：

```
baz(bar(foo()));
```

我们从左往右读这些代码，就会看到 ‘baz bar foo’。但是这并不是我们由内-外调用函数的顺序：‘foo bar baz’。如果我们这样写，会不会更好?

```
foo.bar().baz();
```

幸运的是，你可能已经猜到了，关于上面问题的答案，可以! Rust 提供了一种可以通过 impl 关键字来使用‘方法调用语法’的能力。

方法调用

以下代码展示了它是怎样工作的：

```
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());
}
```

以上代码将打印 12.566371。

我们已经建了一个表示一个圆的结构体。然后我们写一个 impl 块，同时我们在块中定义一个方法，area。

方法使用一个特殊的第一参数，其中有三个变量：self，&self 和 &mut self。你可以将这个第一参数想象成 foo.bar() 中的 foo。这三种变量对应于 foo 可能成为的三种东西：如果它只是堆栈中的一个值使用 self，如果它是

一个引用使用 &self，如果它是一个可变引用使用 &mut self。因为我们使用了 &self作为 area 的参数，我们可以像其他参数一样使用它。由于我们知道它是一个 Circle，我们可以像访问其它结构体一样，访问 radius。

我们应默认使用 &self，同时相对于取得所有权，你应该更倾向于借用，以及使用不可变的引用来顶替可变的引用。以下是关于所有三个变量的一个例子：

```
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!");
}
}
```

链接方法调用

所以，至此我们知道了怎样去调用一个方法，诸如 foo.bar()。但是我们原来的例子 foo.bar().baz() 的例子怎么办？这就是所谓的‘方法链接’，我们可以通过返回 self 来完成它。

```
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, increment: f64) -> Circle {
```

```
Circle { x: self.x, y: self.y, radius: self.radius + increment }
}
}

fn main() {
let c = Circle { x: 0.0, y: 0.0, radius: 2.0 };
println!("{}", c.area());

let d = c.grow(2.0).area();
println!("{}", d);
}
```

检查返回类型：

```
fn grow(&self) -> Circle {
```

我们只是说我们返回了一个 Circle。通过这个方法，我们可以将一个新的圆圈增加到任意大小。

关联函数

你还可以定义一个不使用 self 作为参数的关联函数。以下代码是在 Rust 代码中非常常见的一种模式：

```
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’为我们生成一个新的 Circle。请注意，关联函数被 Struct::function() 语法调用，而不是 ref.method() 语法。其他一些语言称关联函数为‘静态方法’。

生成器模式

假如我们想要我们的用户能够创建 Circles，但是我们只允许他们设置他们关心的属性。否则，x 和 y 的属性将会是 0.0，同时 radius 为 1.0。Rust 没有方法重载，参数命名或者变量参数。我们使用生成器模式来代替它。如以下代码所示：

```
struct Circle {
x: f64,
y: f64,
radius: f64,
}

impl Circle {
fn area(&self) -> f64 {
std::f64::consts::PI * (self.radius * self.radius)
}
}

struct CircleBuilder {
x: f64,
y: f64,
radius: f64,
}

impl CircleBuilder {
fn new() -> CircleBuilder {
CircleBuilder { x: 0.0, y: 0.0, radius: 1.0, }
}

fn x(&mut self, coordinate: f64) -> &mut CircleBuilder {
self.x = coordinate;
self
}

fn y(&mut self, coordinate: f64) -> &mut CircleBuilder {
self.y = coordinate;
self
}

fn radius(&mut self, radius: f64) -> &mut CircleBuilder {
self.radius = radius;
self
}
```

```
fn finalize(&self) -> Circle {
Circle { x: self.x, y: self.y, radius: self.radius }
}
}

fn main() {
let c = CircleBuilder::new()
.x(1.0)
.y(2.0)
.radius(2.0)
.finalize();

println!("area: {}", c.area());
println!("x: {}", c.x);
println!("y: {}", c.y);
}
```

我们这里所做的就是建立了另一个结构体，CircleBuilder。我们已经对它定义了我们的生成器方法。我们也已经在 Circle 上定义了我们的 area() 方法。我们又在 CircleBuilder 上定义了另一方法：finalize()。这个方法从生成器中创建了我们的最后的 Circle。现在，我们已经使用类型系统来强化我们关心的事情：我们可以使用在 CircleBuilder 上的方法来限制以我们选择的任何方式制作 Circle。

向量

一个‘向量’是一个动态的或者‘可增长的’数组，作为标准库类型 Vec<T>来实现。T 表示我们可以有任何类型的向量（更多信息，请参照泛型的章节）。向量总是在堆上分配它们的数据。你可以使用 vec! 宏来创建它们：

```
let v = vec![1, 2, 3, 4, 5]; // v: Vec<i32>
```

（注意，与在之前我们使用的 println! 宏不同，对于 vec! 宏我们使用方括号 []。Rust 允许您在两种情况下使用，这只是个约定。）

对于重复一个初始值，这里有 vec! 的另一种形式：

```
let v = vec![0; 10]; // ten zeroes
```

访问元素

若要获取在向量中的特定索引处的值，我们使用 []：

```
let v = vec![1, 2, 3, 4, 5];

println!("The third element of v is {}", v[2]);
```

由于指数从 0 开始，所以第三个元素是 v[2]。

循环访问

一旦你有了一个向量，你可以通过 for 来遍历它的元素。这里有三个版本：

```
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);
}
```

向量有许多更有用的方法，你可以在它们的 API 文档中读到。

字符串

对于任何程序员来说，字符串是一个重要的且必须掌握的概念。由于其系统专注的点不同，Rust 的字符串处理系统有点不同于其他计算机语言,。无论何时，当你有一个可变大小的数据结构,事情可能会变得棘手，还有，字符串是一种能重设大小的数据结构。也就是说，Rust 的字符串的工作方式也不同于其他的系统语言，如 C 语言。

让我们深入细节。“字符串”是 Unicode 标量值的序列编码为 utf－8 字节的流。所有字符串都必须保证为有效的 utf-8 编码序列。此外，不像一些系统语言，这里的字符串是非空终止字符串，并且，可以包含空字节。

Rust 有两个主要类型的字符串： str 和字符串。首先让我们来谈谈 str。它一般被称为“串片”。字符串一般都是&'static str 类型的：

```
let string = "Hello there."; // string: &'static str
```

这个字符串是静态分配的，这意味着它保存在我们的编译程序里面，并且存在于程序运行的整个时间。字符串通过绑定来引用这个静态分配的字符串。字符串片有固定大小,不能突变。

一个字符串，另一方面来说，是一个基于堆的字符串。这个字符串是可生长的,并且保证为 utf-8。通常使用 to_string 方法将一个字符串转换为另一个字符串。

```
let mut s = "Hello".to_string(); // mut s: String
println!("{}", s);

s.push_str(", world.");
println!("{}", s);
通过&将字符串放入&str：
fn takes_slice(slice: &str) {
println!("Got: {}", slice);
}

fn main() {
let s = "Hello".to_string();
takes_slice(&s);
}
```

将一个字符串转换为 &str 很容易，但将 &str 转换为一个字符串包含分配内存的操作。除非有必要，否则不要这样做!

索引

因为字符串都是有效的 UTF-8，字符串不支持索引

```
let s = "hello";

println!("The first letter of s is {}", s[0]); // ERROR!!!
```

通常。用 [] 访问向量是非常方便的。但是,因为 utf-8编码的字符串里面的每个字符可以多个字节，你必须检索字符串才可以找到字符串的第 n 个字母。这是一个太过复杂的操作，我们不想被误导。此外，“字母”不是 Unicode 中定义的东西，没错。我们可以把字符串看做一个单独的字节，或 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!("");
```

这将输出：

```
229,191,160,229,191,172,227,131,143,227,131,129,229,133,172,

忠犬,ハ,チ、公
```

如您所见，这的字节比字符更多。

你可以得到类似于这样的一个索引：

```
let dog = hachiko.chars().nth(1); // kinda like hachiko[1]
```

这强调，我们必须跨越整个字符列表。

连接

如果你有一个字符串，可以在它的结尾连接一个 str 作为结束：

```
let hello = "Hello ".to_string();
let world = "world!";

let hello_world = hello + world;
```

但是如果你有两个字符串，您需要一个 &：

```
let hello = "Hello ".to_string();
let world = "world!".to_string();

let hello_world = hello + &world;
```

这是因为 &String 可以自动连接一个 str。这个特点也叫作叫做 ‘Deref coercions’ 。

泛型

有时候，写一个函数或数据类型时，我们可能希望它可以作为多种类型的工作参数。幸运的是，Rust 给我们提供了一种更好地解决办法：泛型。在类型理论中，泛型类型被称为“参数多态性”，这意味着在给定参数(变量)的情况下，他们是具有多种形式的类型或函数 (“聚”是多个“变形”形式)。

不管怎样，类型理论已经足够充分,让我们看看一些通用的代码。Rust 的标准库提供了一个类型， `Option<T>` ,这是通用的：

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

`< T >` 部分。正如你之前已经见到过的，表明这是一个通用的数据类型。在我们枚举的声明里面，无论我们在哪里看到 T，我们用泛型中相同的数据类型来作为那个数据类型的替代品。这里有一个使用 `Option<T>` 的例子，与一些额外的类型注解：

```
let x: Option<i32> = Some(5);
```

在类型声明里面，我们用 `Option<i32>` 。请注意它与 `Option<T>` 看起来非常类似。所以,在这个特殊的选项 T 中，T 的值为 i32。在等号的右边,有一个 some(T)，T 的值是 5。因为这是一个 i32，所以双方能够匹配。此时 Rust 运行没有错误。如果他们不匹配,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)
```

这并不意味着我们不能让 `Option<T>s` 的值为 f64 !他们必须匹配才可以：

```
let x: Option<i32> = Some(5);
let y: Option<f64> = Some(5.0f64);
```

这样很好。因为一个定义，可以用于多个用途。

泛型不必只是通用的一种类型。考虑 Rust 的标准库里面的另一种相似的类型， `Result<T, E>`

```
enum Result<A, Z> {
Ok(A),
Err(Z),
}
```

这个类型对这两种类型都是通用的：T 和 E,顺便说一下，大写字母可以是任何你喜欢的字母。我们可以定义 `Result<T, E>` 如下：

```
enum Result<A, Z> {
Ok(A),
Err(Z),
}
```

如果我们想要这样。公约说，第一个通用的参数应该是 T，它对应于“类型”，我们使用 E 对应于 error。然而，Rust 并不关心这些。

`Result<T, E>` 类型的作用是用于返回计算的结果，并能够在发生差错的时候返回一个错误。

通用函数

我们可以用相似的语法编写带有泛型的函数：

```
fn takes_anything<T>(x: T) {
// do something with x
}
```

语法分为两个部分：`< T >` 表示“这个函数对类型 T” 是通用，和 x:T 表示“x 的类型为 T .”

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) {
// ...
}
```

我们可以写一个拥有多种类型的版本：

```
fn takes_two_things<T, U>(x: T, y: U) {
// ...
}
```

对于特征边界来说通用函数是最有用的，我们将在特征部分讨论它。

通用结构

你可以存储一个泛型类型的结构：

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

let int_origin = Point { x: 0, y: 0 };
let float_originstruct Point<T> {
x: T,
y: T,
}

let int_origin = Point { x: 0, y: 0 };
let float_origin = Point { x: 0.0, y: 0.0 };
```

类似于函数，`< T >` 是我们声明泛型参数的地方，然后，我们在类型声明里使用 x:T。

特征

你还记得 impl 关键字吗，它用于调用一个函数的 methodsyntax

```
struct Circle {
x: f64,
y: f64,
radius: f64,
}

impl Circle {
fn area(&self) -> f64 {
std::f64::consts::PI * (self.radius * self.radius)
}
}
```

特征几乎都是相似的，除了我们用方法签名去定义一个特征,然后实现该结构的特征。如下面所示：

```
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)
}
}
```

如您所见，这个特征块和 impl 块非常相似，但我们不定义一个主体，只是定义一个类型签名。当我们 impl 一个特征时，我们使用 impl 特征项，而不仅仅是 impl 项。

我们可以使用特征约束泛型。思考下面这个函数，它没有编译，只给我们一个类似的错误：

```
fn print_area<T>(shape: T) {
println!("This shape has an area of {}", shape.area());
}
```

Rust 可能会抱怨道：

```
error: type `T` does not implement any method in scope named `area`
```

因为 T 可以是任何类型，我们不能确保它实现了 area 的方法。但我们可以添加一个特征约束的泛型 T，确保它已经实现：

```
fn print_area<T: HasArea>(shape: T) {
println!("This shape has an area of {}", shape.area());
}
```

语法 `< T:HasArea >` 意味着实现 HasArea 特征的任何类型。因为特征定义函数的类型签名，我们可以肯定，任何实现 HasArea 的类型都会有.area()方法。

这里有一个扩展的例子展示它是如何工作的：

```
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);
}
```

这个程序输出：

```
This shape has an area of 3.141593
This shape has an area of 1
```

如您所见，print_area 现在是通用的，同时也确保我们通过了正确的类型。如果我们通过一个错误的类型：

```
print_area(5);
```

我们会得到一个编译错误：

```
error: failed to find an implementation of trait main::HasArea for int
```

到目前为止，我们只添加特征实现结构，但您可以实现任何类型的特征。所以从技术上讲，我们可以为 i32 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();
```

我们一般认为用这种基本类型来实现方法是一种不够好的风格，即便这种实现方法是可行的。

这可能看起来比较粗糙，但是有两个其他的限制来控制特征的实现,防止失控。首先，如果特征不是在你的范围中定义的,那么不适用。下面是一个例子：标准库提供了编写特征，这个特征给文件的输入输出增加了额外的功能。默认情况下，文件并不会有自己的方法：

```
let mut f = std::fs::File::open("foo.txt").ok().expect("Couldn’t open foo.txt");
let result = f.write("whatever".as_bytes());
```

这里有一个错误：

```
error: type `std::fs::File` does not implement any method in scope named `write`

let result = f.write(b"whatever");
```

我们需要先使用编写特征：

```
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());
```

此时编译没有出现错误。

这意味着，即使有人做了坏事比如向 int 添加方法，也不会影响你，除非你使用这个特征。

实现特征还有其他限制。你为特征或类型所写的 impl 必须由你来定义。所以,我们可以实现 HasArea 类型等，因为 HasArea 是我们的代码。但是如果我们试图实现浮点型,它是 Rust 为 i32 所提供的特征，是不可能的，因为无论特征还是类型都不在我们的代码里面。

最后关于特征：通用函数特征的捆绑使用 “monomorphization”(mono:一个，morph:形式),所以他们都是静态调用。那意味着什么呢?到特征对象那一章查看更多细节。

多个特征边界

如您所看到的，您可以绑定特征与一个泛型的类型参数：

```
fn foo<T: Clone>(x: T) {
x.clone();
}
```

如果你需要多个绑定，您可以使用+：

```
use std::fmt::Debug;

fn foo<T: Clone + Debug>(x: T) {
x.clone();
println!("{:?}", x);
}
```

T 现在需要复制以及调试。

Where 子句

只用少数泛型和少量的特征界限来写函数并不是太糟糕，但随着数量的增加，语法变得越来越糟糕：

```
use std::fmt::Debug;

fn foo<T: Clone, K: Clone + Debug>(x: T, y: K) {
x.clone();
y.clone();
println!("{:?}", y);
}
println !（“{:?}”,y);
}
```

函数的名称是在最左边，参数列表在最右边。边界就以这种方式存在。

Rust 已经有了解决方法，这就是所谓的“where 子句”：

```
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()使用我们之前讲解的语法，bar()使用一个 where 子句。所有你需要做的就是定义类型参数时不要定义边界，然后在参数列表之后添加 where 语句。对于更长的列表，可以添加空格：

```
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清晰。

Where 语句也比简单的语法更加强大。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()
}
```

这个例子展了示 where 子句的附加特性：它们允许范围内的左边可以是一个任意的类型(在这里是 i32)，而不只是一个普通的类型参数(如 T)。

默认的方法

我们的最后一个特征的特性应包括：默认的方法。举个简单的例子更容易说明：

```
trait Foo {
fn bar(&self);

fn baz(&self) { println!("We called baz."); }
}
```

Foo 特征的实现者需要实现 bar()方法，但是他们不需要实现 baz()。他们会得到这种默认行为。如果他们这么选择的话就可以覆盖默认的情形：

```
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 baz."

let over = OverrideDefault;
over.baz(); // prints "Override baz!"
↖
```

继承

有时，想要实现某个特征需要先实现另一个特征：

```
trait Foo {
fn foo(&self);
}
```

```
trait FooBar : Foo {
fn foobar(&self);
}
 FooBar的实现者也要实现Foo,像这样:
struct Baz;

impl Foo for Baz {
fn foo(&self) { println!("foo"); }
}

impl FooBar for Baz {
fn foobar(&self) { println!("foobar"); }
}
```

如果我们忘记实现 Foo，Rust 会告诉我们：

```
error: the trait `main::Foo` is not implemented for the type `main::Baz` [E027]
```

降

既然我们已经讨论了特征，让我们谈谈 Rust 标准库提供的特定特征，降降特性提供了一种方法，这种方法可以保证即便某个值超出范围时也能运行代码。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 x 在 main()函数结尾超出范围时，降的代码将运行。降有一个方法,该方法也被称为 drop()。它将 self 作为可变参考。

这正是我们想要的!降的机制是非常简单的，但是有一些微妙之处。例如，值以与它的声明相反的顺序下降。这里还有一个例子：

```
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 };
}
```

这将输出：

```
BOOM times 100!!!

BOOM times 1!!!
```

TNT 在爆竹之前爆炸，因为它是之后被声明的。后进先出。

那降有什么优点呢？一般来说,降是用来清理任何与结构相关联的资源。例如，`ARC<T> TYPE` 类型是一个引用计数的类型。调用降时，它将减量引用计数，如果引用的总数为零，它将清理潜在值。

If let

If let 允许你把 if 和 let 结合到一起，来减少某些类型的模式匹配所需的开销。

例如，有某种 `Option<T>` 。如果它是 `Some<T>` ，我们希望在它上面调用一个函数，如果不是，则什么也不做。就像下面这样：

```
match option {
Some(x) => { foo(x) },
None => {},
}
```

在这里我们不一定非要使用匹配,例如,我们可以使用 if

```
if option.is_some() {
let x = option.unwrap();
foo(x);
}
```

这些选项都不是特别有吸引力。我们可以用 if let 语句以更好的方式做同样的事情：

```
if let Some(x) = option {
foo(x);
}
```

如果一个模式匹配成功，它将给模式的标识符绑定任意合适的值，然后评估表达式。如果模式不匹配，则什么也不去做。

当模式不匹配时，如果你希望去做别的事情,您可以使用 else：

```
if let Some(x) = option {
foo(x);
} else {
bar();
}
while let
```

以类似的方式，当一个值匹配某种模式时，你可以用 while let 来进行条件循环。代码如下面所示：

```
loop {
match option {
Some(x) => println!("{}", x),
_ => break,
```

```
}
}
```

转换成下面这样的代码:

```
while let Some(x) = option {
println!("{}", x);
}
```

特征的对象

当代码涉及多态，需要有一种机制来确定实际上是哪些特定版本在运行。这就是所谓的“调度”。调度的主要形式有两种：静态调度和动态调度。Rust 不仅支持静态调度，它还通过“特征对象”机制支持动态调度。

背景

在本章的其余部分，我们将需要一个特征和一些实现。让我们先从一个简单的开始，Foo。它有一个方法，该方法将返回一个字符串。

```
trait Foo {
fn method(&self) -> String;
}
```

我们将在 u8 和 String 里面实现这个特征：

```
impl Foo for u8 {
fn method(&self) -> String { format!("u8: {}", *self) }
}

impl Foo for String {
fn method(&self) -> String { format!("string: {}", *self) }
}
```

静态调度

我们可以利用这一特性的特征边界来执行静态调度：

```
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 使用 “monomorphization” 来执行静态调度。这意味着 Rust 将为 u8 和 String 创建一个特殊版本的 do_something()函数，然后调用这些具体的函数来取代调用 sites 。换句话说，Rust 生成是这样的：

```
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);
}
```

这样会有很大的好处:静态调度允许内联函数调用，因为被调用的函数在编译时是已知的，并且内联是进行良好的优化的关键。静态调度虽然快，但它是折衷之后的结果：代码膨胀，由于许多相同的函数都有二进制副本，并且每种类型的函数都对应一个二进制副本。

另外，编译器并不完美，可能“优化”反而使代码变得更慢。例如，内联函数过于急切地将膨胀指令缓存（缓存规定我们周围的一切）。这是 #[inline] 和 #[inline(always)] 应该被谨慎使用的部分原因，另一个使用动态调度的原因是它有时效率更高。

然而，常见的情况是使用静态调度更高效，并且你总是可以用一个简练的 statically-dispatched 包装器函数进行动态调度，但是用动态调度却不是这样，这意味着静态调用更灵活。可能因为这个原因，标准库更多的使用静态调度。

动态调度

Rust 通过一个称为特征对象的特性提供动态调度。特征对象，如 `&Foo or Box<Foo>` 可以是正常值，这个值存储一个任意类型的值，而该类型实现了给定的特征，至于究竟是哪种类型在运行时才能知道。

特征对象可以从指针获得，这个指针指向一个具体的类型，而该类型通过 cast(例如 &x as &Foo)或 coerce(例如，使用 &x 作为函数的参数，函数里面带有 &foo) 实现了特征。

特征对象 coercions 和 casts 也服务于指针像 `&mut T to &mut Foo` 与 `Box<T> to Box<Foo>` 但都只是在此刻有效。而 Coercions 和 casts 是相同的。

这个操作可以被视为“擦除”编译器对特定类型的指针的认知，因此特征对象有时也被称为类型擦除。

回到上面的例子，我们可以使用相同的特征来执行动态调度与特征的对象的 cast：

```
fn do_something(x: &Foo) {
x.method();
}

fn main() {
let x = 5u8;
do_something(&x as &Foo);
}
或者通过coerce：：
fn do_something(x: &Foo) {
x.method();
}

fn main() {
let x = "Hello".to_string();
do_something(&x);
}
```

每种实现 Foo 的类型并不专门指定一个带有个特征对象的函数：只有当一个副本生成时，才会(但不总是如此)导致更少的代码膨胀。然而，这是需要以更慢的虚拟函数调用作为代价的，并且极大地抑制了任何内联的机会和相关优化的发生。

为选择指针呢?

在默认情况下，Rust 不给指针赋予初值，不像许多其他管理语言那样，所以类型可以有不同的大小。在编译时知道值的大小是很重要的，比如把它作为参数传递给一个函数，又或者把它放进堆栈并且在存储它的堆里面分配（或取消分配）空间给它。

对于 Foo，我们需要有一个值，这个值至少要是一个字符串(24 字节)或 u8(1 个字节)，或者是任意其他依赖于箱来实现他们的Foo的类型(任意的字节数)。如果没有存储的值没有指针来指向，就没有办法保证这最后一点行得通，因为其他那些类型可以是任意大小的。

让指针指向一个值意味着，当我们抛出一个特征对象时，值的大小是无关紧要的，我们只关心指针本身的大小。

表示

特征的方法可以通过特征对象来调用。特征对象是通过函数指针的一个特殊的记录(由编译器创建和管理)来调用方法的，该记录传统上也被称为“虚表”。

特征对象既简单的又复杂：其核心表示和布局其实很简单，但也有一些复杂的错误消息和令人意外的结果。

让我们以简单的特征对象的运行时间的表示来开始。`std::raw` 模块包含与内置类型一样复杂的布局结构，包括特征对象:

```
pub struct TraitObject {
pub data: *mut (),
pub vtable: *mut (),
}
```

即特征对象 &Foo 包含数据指针和一个虚表指针。

数据指针指向特征对象所存储的数据(一些未知类型T)，而虚表指针指向对应的虚表(“虚拟方法表”)对应于 T 的 Foo 实现。

vtable 本质上是一个函数指针结构体，指向每个实现方法的具体的机器代码。调用方法 `trait_object.method()` 将检索虚表外部的正确的指针，然后做一个动态调用。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,
};
```

每个 vtable 的析构函数的字段都指向一个函数，这个函数将清理 vtable 类型的任何资源，对 u8 来说这几乎是没有什么用处的，但对字符串来说却可以释放内存。如果想要拥有 `Box<Foo>` 那样的特征对象，这个操作就是必不可少的，当超出范围时，就需要清理 Box 分配以及内部类型。size 和 align 字段存储着被删除类型的大小，及其一致性要求；由于信息被嵌入到了析构函数，目前这些实际上都未使用，但将来会得到应用，因为特征对象逐渐变得更加灵活。

假设我们有一些实现 Foo 的值，然后你会发现 Foo 的显式形式的构建和 Foo 特征对象的使用看起来有点类似(忽略类型不匹配：无论如何，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 拥有特征对象（`Box < Foo >`）当他们超出范围时，就会有一个(`b.vtable.destructor`)(b.data) (分别 y)调用。

闭包

Rust 不仅命名函数，还命名匿名函数。匿名函数有一个关联的环境被称为“闭包”，因为他们关闭了一个环境。如我们将会看到的，Rust 对它们有完美的实现。

语法

闭包看起来是这样的：

```
let plus_one = |x: i32| x + 1;

assert_eq!(2, plus_one(1));
```

我们创建一个绑定，plus_one 并将其分配给一个闭包。关闭管道之间的参数(|)，并且主体是一个表达式，在本例中是 x + 1。记住{ }也是一个表达式，所以我们可以多行的闭包：

```
let plus_two = |x| {
let mut result: i32 = x;

result += 1;
result += 1;

result
};

assert_eq!(4, plus_two(2));
```

你会注意到闭包与用 fn 定义的普通函数略有不同。第一个不同点是，我们不需要对参数的类型和返回值进行注释。我们可以：

```
let plus_one = |x: i32| -> i32 { x + 1 };

assert_eq!(2, plus_one(1));
```

但我们并不需要这样做。这是为什么呢?根本上来说，这是出于人体工程学的考虑。对于文档和类型推断而言，指定已命名函数的完整类型命名是有用的，然而闭包的类型很少被记录因为他们是匿名的，而且他们不会引 error-at-a-distance 错误，这种错误能够推断命名函数的类型。

第二，语法是相似的，但有点不同。我在这里添加空格让他们看起来更接近：

```
ffn  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  ;
```

差异很小，但它们类似。

闭包和他们的环境

闭包之所以被称为闭包，是因为他们封闭了他们的环境。它看起来像这样：

```
let num = 5;
let plus_num = |x: i32| x + num;

assert_eq!(10, plus_num(5));
```

plus_num，这个闭包指的是一个 let 绑定在它的范围 num 内。更具体地说，它借用了绑定。如果我们做与绑定相冲突的事情，就会得到一个错误。像下面这样：

```
let mut num = 5;
let plus_num = |x: i32| x + num;

let y = &mut num;
```

错误如下：

```
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;
}
^
```

这里的错误信息虽然冗长却是有用的!比如说，我们不能对 num 进行一个可变的 borrow，因为闭包已经 borrow 过它了。如果我们让闭包超出范围，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，Rust 将获取所有权并且移动环境：

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

let takes_nums = || nums;

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

这告诉我们：

```
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 >` 对其内容拥有所有权，因此，当我们在闭包的操作涉及到它时，我们将不得不声称对 num 的所有权。同样的，如果我们将 num 传递给一个函数，则这个函数对其拥有所有权。

移动闭包

我们可以强制我们的闭包用 move 关键字获取环境移所有权：

```
let num = 5;

let owns_num = move |x: i32| x + num;
```

现在，即使关键字是 move，变量仍然遵循正常 move语义。在这种情况下，5 实现复制，所以 owns_num 持有 num 的副本。然而区别在哪里呢？

```
let mut num = 5;

{
let mut add_num = |x: i32| num += x;

add_num(5);
}
```

```
assert_eq!(10, num);
```

所以在这种情况下，我们的闭包获得了一个可变 num，我们称为 add_num， 如我们所期望的，它改变了 num 的潜在值。我们还需要将 add_nu m声明为 mut，因为我们正在改变其环境。

如果我们换成一个 move 闭包，就会出现不同：

```
let mut num = 5;

{
let mut add_num = move |x: i32| num += x;

add_num(5);
}

assert_eq!(5, num);
```

我们只得到 5。而不是从 num 得到可变的 borrow 我们对副本拥有所有权。

另一种方式思考 mov e闭包:他们分配给闭包一个自己的堆栈帧。没有 move 一个闭包可能与创建它的堆栈帧联系到一起，而且闭包是自包含的。这意味着你不能从函数返回一个 non-move 闭包。

但在我们讨论使用和返回闭包并之前，我们应该更多的讨论闭包的实现方式。作为一种系统语言、Rust 让你能够控制代码所做的事情，而闭包是没有什么不同的。

闭包实现

Rust 的闭包实现有点不同于其他语言。对特征来说他们是非常高效的语言。你要确保阅读这一章之前已经阅读了特征这一章，以及特征对象这一章。

都明白了吗？很棒。

闭包工作的关键点有些奇怪:使用()调用一个函数，就像 foo() 是一种可重载操作符。由此，在其他的任何地方单击鼠标都能进入空间。在 Rust 语言里面，我们使用特征系统重载操作符。调用函数也不例外。我们有三个独立的过载与特征：

```
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;
}
```

你会注意到这些特征之间的差异，但很大的一个点是 self:Fn 和 &self,FnMut 和 &mut self,FnOnce 和 self。这通过常用的方法调用语法涵盖了所有三种 self。但是我们把他们分成三个特征，而不是一个。这给了我们足够的控制权去决定我们可以采用什么样的闭包。

对闭包的| | { }语法对这三个特征来说是糖衣语法。Rus t将为环境生成一个 struct， impl 适当的特征,然后使用它。

使用闭包作为参数

现在我们知道，闭包特征，我们已经知道如何接受和返回闭包:就像任何其他特征那样!

这也意味着我们可以选择静态与动态调度。首先，让我们写一个可以调用其他函数的函数，调用它，并返回结果:

```
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);
```

我们通过闭包，`| | x + 2` 去调用 call_with_one。它只是做它所表明的事情：它调用闭包，1 作为参数。

让我们更深入地检查 call_with_one 的签名:

```
fn call_with_one < F >(some_closure:F)- >i32
```

我们需要一个参数，它的类型为 F .我们也返回一个 i32。这部分不是很有趣。下一个部分是:

```
where F : Fn(i32) -> i32 {
```

因为 Fn 是一个特征，我们可以将我们的泛型和它绑定到一起。在这种情况下，我们的闭包需要把 i32 作为参数，并返回一个 i32 所以我们使用的泛型边界是 `Fn(i32) -> i32` 。

这里还有一个关键问题:因为我们吧泛型和特征绑定到了一起，这将导致单形态，因此，我们将在闭包里面做静态调度。那是就清晰多了。在许多语言中，闭包本身就是堆分配，总是涉及到动态调度。在 Rust 语言中，我们可以用堆栈分配我们的闭包环境，和静态调度 call 语句。这种情况通常发生于迭代器和适配器，通常采用闭包作为参数。

当然，如果我们想要动态调度，我们也可以那样。特征对象处理这种情况时，像往常一样：

```
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);
```

现在我们把一个特征对象, `&Fn` 。当我们传递这个特征到 call_with_one 时我们必须参考我们的闭包,所以我们使用 `&||` 。

返回闭包

在各种情况下，返回闭包是很常见的函数形式。如果你想返回一个闭包，您可能会遇到一个错误。起初，这看起来可能有些奇怪，但是我们会找到答案。你可以试着从下面的函数返回一个闭包：

```
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);
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 known at compile-time
fa ctory() -> (Fn(i32) -> Vec<i32>) {
  ^~~~~~~~~~~~~~~~~~~~~
```

为了从一个函数返回一些东西，Rust 需要知道返回类型的大小。但由于 Fn 是一个特征，它可能具有是各种不同大小的 size： 许多不同的类型都可以实现 Fn。来给一些事物赋予 siz e的一种简单的方法是参考已知的 siz e。所以如下面我们写的这样：

```
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);
```

但我们得到了另一个错误：

```
error: missing lifetime specifier [E0106]
fn factory() -> &(Fn(i32) -> i32) {
^~~~~~~~~~~~~~~~~
```

正确的。因为我们有一个参考，我们需要给它指定生命周期。但是我们的 factory() 函数不带参数，所以此处省略不写。我们可以选择什么样的生命周期呢?静态：

```
fn factory() -> &'static (Fn(i32) -> i32) {
let num = 5;

|x| x + num
}

let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

但我们得到另一个错误：

```
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
 ^~~~~~~~~~
```

这个错误是让我们知道，我们没有 `&'static Fn(i32) -> i32` 但是我们有一个 `[closure <anon>:7:9: 7:20]` 。等等，这是什么?

因为每个闭包生成自己的环境结构和实现 Fn 特征以及 friends 这些都是匿名的。它们的存在只是因为这个闭包。Rust 把他们显示为 `closure <anon>` 而不是一些自动生成的名字。

但是为什么我们的闭包不实现 `&'static Fn` ?正如我们之前讨论的，闭包 borrow 了他们的环境。在这种情况下，我们的环境是基于 5 栈分配以及num 变量绑定。因此，borrow 的生命周期为的堆栈帧长度。所以如果我们返回这个闭包，函数调用将结束，堆栈框架将消失，并且我们的闭包将对垃圾内存的环境进行捕获!

那么该怎么办?这就是工作原理:

```
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);
```

我们使用一个特征对象,通过建立 Fn。只有最后一个问题:

```
error: `num` does not live long enough
Box::new(|x| x + num)
 ^~~~~~~~~~
```

我们仍然有一个对父堆栈帧的引用。通过最后一次修复，我们这样做可以做可以行得通:

```
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);
```

通过内部 move Fn，我们可以为我们的闭包创建一个新的堆栈帧。我们给它一个已知大小，并对它进行填充，而且允许他脱离我们的堆栈帧。

通用函数调用语法

有时，函数可以有相同的名字。看看下面这段代码：

```
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;
```

如果我们试图调用 b.f()，我们就会得到一个错误：

```
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"); }
```

我们需要一种消除歧义的方法。这种方法称为“通用函数调用语法”，它这种语法看起来是下面这样的：

```
Foo::f(&b);
Bar::f(&b);
我们让它停止一下。
```

```
Foo::
Bar::
```

这些部分调用的类型有两个特征:Foo 和 Bar。这最终实际上是做了两者之间的消歧：Rust 从两者中调用你所使用的特征名称。

```
f(&b)
```

当我们用 method syntax（ ）调用一个方法比如 b.f()，如果 f() 含有 &self，Rust 会自动 borrow b。在本例这种情况下，Rust 不会，所以我们需要传递一个具体的 &b。

尖括号形式

现在我们谈论一下 UFCS 形式：

```
Trait::method(args);
```

这只是一种速记收手法。下面是在某些情况下需要使用的扩展形式：

```
<Type as Trait>::method(args);
```

`<>::` 语法是一种提供类型提示的方法。类型写在 `<>s` 里面。在本例中，类型是 Type as Trait 表明我们希望特征的方法在这里被调用。在不出现歧义的情况下，特征部分是可选的。这同样适用于用尖括号括，因此要用较短的形式。

这是使用更长形式的一个例子：

```
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);
}
}
```

这将调用 Clone 特征的 Clone () 方法，而不是 Foo 特征的方法。

箱和模块

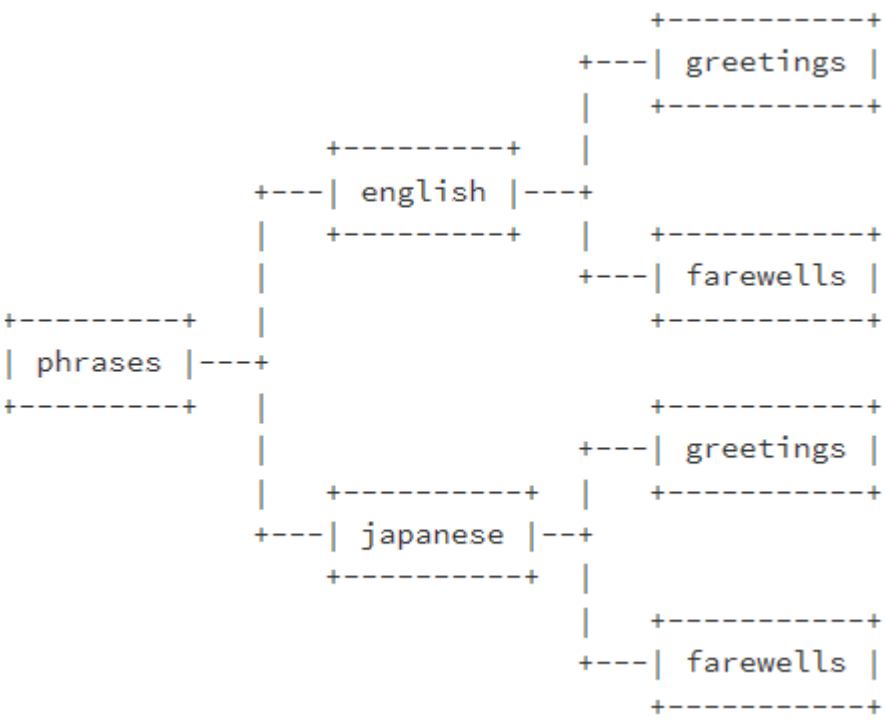
当一个项目开始变大时，我们通常认为的良好的软件工程实践是把它分割成小块，然后把它们组合在一起。同样重要的是有一个定义良好的接口，这样你的一些功能可以是私人的，另外一些可以是公开的。为了促进这些事情，Rust 使用了模块系统。

基本术语：箱和模块

关与模块系统，Rust 有两个不同的术语:“箱” 和“模块”。在其他语言里箱的代名词是 “库” 或 “包”。因此 “Cargo” 就是 Rust语言的的包管理工具:你可以用Cargo 装载你的箱转运给其他程序。根据不同的项目，箱可以产生一个可执行文件或库。

每个箱有一个包含箱代码的隐式根模块。然后，您可以定义一个根模块下的子树模块。 模块允许你分区箱内箱外代码。

作为一个例子，让我们做一个短语箱，它将在不同的语言中给我们不同的词语。为简单起见，我们将使用 “问候” 和 “告别” 两种类型的短语，并使用英语和日语 (日本语) 两种语言。我们将使用下面这个模块布局:



图片 5.1 image

在这个例子中，短语是我们箱的名字。其余都是模块。你可以看到，他们形成一个树，分支从箱根发出，根指的是树的根：短语本身。

现在我们有一个计划，让我们来在代码中定义这些模块。首先，用 Cargo 生成一个新的箱：

```
$ cargo new phrases
$ cd phrases
```

如果你记得以前所讲的，这将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` 是我们箱根，对应于我们在上图中的短语。

定义模块

我们使用 mod 关键字来定义我们的每个模块。让我们使我们的 `src/lib.rs`，看起来就像这样：

```
mod english {
mod greetings {
}

mod farewells {
}
}

mod japanese {
mod greetings {
}

mod farewells {
}
}
```

在 mod 关键字后，我们给出模块的名称。模块名称遵守 Rus t规定的标识符命名规则:lower_snake_case。每个模块的内容在花括号 ({ }) 里面。

在一个给定的模式下，您可以声明 sub-mods。我们可以用双冒号 (::) 符号引用子模块：我们的四个嵌套模块是 english::greetings， english::farewells， japanese::greetings， 还有 japanese::farewells。因为这些子模块是在他们父模块命名空间命名的，名字不冲突： english::greetings 和 japanese::greetings 是不同的，尽管他们的名字都是问候。

因为这个箱子没有 main() 函数，并且被称为 lib.rs， Cargo 将把这个箱建成一个库：

```
$ cargo build
  Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
$ ls target/debug
build  deps  examples  libphrases-a7448e02a0468eaa.rlib  native
```

libphrase-hash.rlib 是编译后的箱。在我们知道如何在另一个箱里面使用这个箱之前，让我们把它分成多个文件。

多个文件箱

如果每个箱只是一个文件，那么这些文件会很大。我们常常很容易将箱分成多个文件，并且 Rust 从两个方面来支持这样做。

不是像下面这样声明一个模块：

```
mod english {
// contents of our module go here
}
```

相反我们可以这样声明我们的模块：

```
mod english;
```

如果我们这样做，Rust 将期望找到一个 english.rs 文件，或者是含有我们的模块的内容的 `english/mod.rs` 文件。

注意，在这些文件中，您不需要 re-declare 模块：这些已经由最初的模块的声明了。

使用这两种技巧，我们可以把箱子拆分成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
```

图片 5.2 image

src/lib.rs 是我们箱根，看起来像这样：

```
mod english;
mod japanese;
```

这两个声明告诉 Rust 根据我们的偏好去寻找 `src/english.rs` 和 `src/japanese.rs`， 或者 `src/english/mod.rs` 和 `src/japanese/mod.rs` 。在这种情况下，由于我们的模块有子模块，我们就选择第二个。`src/english/mod.rs` 和 `src/japanese/mod.rs` 看起来都像这样:

```
mod greetings;
mod farewells;
```

再一次，这些声明告诉 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，我们选择让他们 `src/english/greetings.rs` 和 `src/japanese/farewells.rs` 。

`src/english/greetings.rs` 和 `src/japanese/farewells.rs` 的内容在此时都是空的。让我们添加一些函数。

把下面这些放到 `src/english/greetings.rs` 里面：

```
fn hello() -> String {
"Hello!".to_string()
}
把下面这些放到src/english/farewells.rs:
fn goodbye() -> String {
"Goodbye.".to_string()
}
src/japanese/greetings.rs:
fn hello() -> String {
"こんにちは".to_string()
}
```

当然，你可以从这个网页复制和粘贴这些或者自己敲一些其他的东西。你用 “konnichiwa” 还是其他的什么学习模块系统实际上并不重要。

把下面这些放到 `src /日本/ farewells.rs` :

```
fn goodbye() -> String {
"さようなら".to_string()
}
```

(如果你好奇的话，可以告诉你这是 “Say ō nara” 。)

现在，我们的箱具有一些功能，让我们试着从另一个箱使用这些功能。

导入外部箱

我们有一个库箱。让我们做一个可执行的箱，这个箱导入和并且使用我们的库。

生成一个 `src/main.rs` 并且把下面这些代码输进去(此时还不会完全编译):

```
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());
}
```

外面的箱声明告诉，我们需要编译和链接短语箱。我们可以使用短语“模块。如前所述，您可以使用双冒号来引用子模块的内部功能。

另外， Cargo 假设 src/main.rs 是一个二进制箱的根箱，而不是一个箱库。我们的包现在有两个箱： src/lib.rs 以及 src/main.rs 。对可执行文件箱来说，这种模式是很常见的：大多数功能都是在库箱里面，并且可执行箱将使用这个库。在这种方式下，其他程序也可以使用库箱，这也是一个不错的关注点分离方法。

然而这并不管用。我们得到了类似下面四个的错误：

```
$ 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
```

默认情况下，在 Rus t语言里面一切都是非公开的。让我们从更深的层次来谈一下。

导出一个公共接口

在默认情况下，Rust 可以精确地控制你的接口的哪些方面是公开的，哪些方面是非公开的。要把某些事物公开，你需要使用使用 pub 关键字。让我们首先关注 english 模块，然后让我们减小我们的 src/main.rs 到下面这样：

```
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/english/mod.rs 里面，让我们添加 pub 到英语模块声明里面：

```
pub mod english;
mod japanese;
```

并且在我们的 src/english/mod.rs 里面，我们写两个 pub 语句：

```
pub mod greetings;
pub mod farewells;
```

在我们的 src/english/greetings.rs 里面，我们添加 pub 到 fn 的声明里面：

```
pub fn hello() -> String {
"Hello!".to_string()
}
```

也在 src/english/farewells.rs 里面这样做：

```
pub fn goodbye() -> String {
"Goodbye.".to_string()
}
```

现在，虽然有警告告诉我们不能使用带有日语的函数，我们的箱依然进行编译：

```
$ 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(dead_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，我们的函数是公开的，我们可以使用它们。太棒了！然而，输入 phrases::english::greetings::hello() 太长而且重复。Rust 还有另一个关键字可以导入名称到当前的范围，这样你可以用更短的名字来引用他们。让我们谈谈 use。

用 use 导入模块

Rust 有一个 use 关键字，它允许我们将名称导入本地范围。让我们改变我们的 src/main.rs 成下面这样：

```
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());
}
```

两个 use 行将每个模块导入到本地范围，所以我们可以用更短的名称来调用函数。按照惯例，在导入功能时，通常认为最好的做法是导入模块而不是直接导入函数。换句话说，你可以这样做：

```
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());
}
```

但它不是惯用的方法。这是更有可能引入命名冲突。在我们的短程序里面，这不是一个大问题，但是当它在大型程序里面就变成一个问题了。如果我们有相互矛盾的名字，Rust 会给出一个编译错误。例如，如果我们用 public 修饰日语函数，并试图做到这一点：

```
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 将给我们一个编译时的错误：

```
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 module [E0252]
src/main.rs:4 use phrases::japanese::greetings::hello;

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

如果我们从相同的模块导入多个名称，我们不需要输入两次。不必像下面这样：

```
use phrases::english::greetings;
use phrases::english::farewells;
```

我们可以使用这个快捷键：

```
use phrases::english::{greetings, farewells};
```

用 pub use 重新导出

你不要只是使用 use 关键字来缩短标识符。您还可以在你的箱里使用它去再次导入一个在另一个模块里的函数。这允许您呈现一个外部接口，并且这个接口可以并不直接映射到您的内部代码组织。

让我们来看一个例子。修改您的 `src/main.rs` 像下面这样：

```
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，修改您的 `src/lib.rs` 使日语 mod 公开：

```
pub mod english;
pub mod japanese;
```

接下来，公开这两个函数，首先在 `src/japanese/greetings.rs` 里面：

```
pub fn hello() -> String {
"こんにちは".to_string()
}
```

然后在 `src/japanese/farewells.rs` 里面：

```
pub fn goodbye() -> String {
"さようなら".to_string()
}
```

最后，修改您的 src/japanese/mod.rs 成下面这样:

```
pub use self::greetings::hello;
pub use self::farewells::goodbye;
```

```
mod greetings;
mod farewells;
```

在我们模块的层次结构的这一部分，pub use 将在这个范围内声明函数。因为我们在日语模块里面已经这样做了，我们现在有一个 phrases::japanese::hello() 函数 和 一个 phrases::japanese::goodbye() 函数，虽然他们的代码存在于 phrases::japanese::greetings::hello() 和 phrases::japanese::farewells::goodbye()。我们内部组织不能定义我们的外部接口。

在这里，每个我们想纳入日语范围的函数都有一个 pub use。我们也可以使用通配符语法去吧 greating 里面的所有东西列入到当前范围: pub use self::greetings::*。

那 sel f呢?默认情况下，从你的箱根开始，use 的声明都是绝对路径。相反，self 使这条路是一条相对于当前的层次结构的相对路径。还有一个 use 的特殊形式：您可以使用 super:: 达到你所在的树的当前层次的上一层。从许多 shell 的当前目录和父目录所显示的来看，有些人经常认为 self是.而 super是. .。

在 Use 的外部，路径是相对的: 相对于我们所处的位置，foo::bar() 指向的是一个 foo内部的函数。如果那是一个 :: 前缀，就像 ::foo:bar()里面的，那么它指的是一个不同的 foo，是一条从你的箱根开始的绝对路径。

同时，注意，我们在声明 mod 之前就 pub use 了。Rust 语言要求首先进行 use 的声明。

这将构建并运行下面的代码：

```
$ 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
```

“常量”和“静态”

Rust 有用 const 关键字定义常量的方法：

```
const N: i32 = 5;
```

不像 let 绑定那样，您必须标注常量的类型。

常量活在程序的整个生命周期。更具体地说，在 Rust 语言里面常量没有固定内存地址。这是因为他们会被有效的内联到每个使用地方。引用相同的常数并不一定保证引用同一个内存地址也是因为这个原因。

静态

Rust 在静态条目中提供一种“全局变量”。它们类似于常量，但静态条目不内联使用。这意味着每个值只对应一个实例， 并且在内存中只有一个固定的地址。

这里有一个例子：

```
static N: i32 = 5;
```

不像 let 绑定那样，您必须标注一个静态的类型。

静态类型活在程序的整个生命周期，因此任何存储在常量中的引用都有“静态生命周期：

```
static NAME: &'static str = "Steve";
```

可变性

你可以用 mut 关键字介绍可变性:

```
static mut N: i32 = 5;
```

因为这是可变的，一个线程正在更新 N，而此时另一个正在读入它，这样会导致内存不安全。这样访问和改变一个静态 mut 是不安全的，所以必须在一个unsafe 的块里面完成：

```
unsafe {
N += 1;

println!("N: {}", N);
}
```

此外，任何存储在 static 的类型都必须是 Sync。

初始化

常量和静态常量都要求给他们一个值。并且他们可能只被赋予一个值，这个值是一个常数表达式。换句话说，在程序运行时，您不能使用函数调用的结果或任何其他类似的复杂操作。

我应该使用哪个构造?

几乎总是如此，如果你能在两者之间选择，那么就选择常量。人们几乎不怎么希望内存地址与你的常量关联到一起，并且允许对常量进行优化，就像常量的使用范围不仅在本程序块也在下游程序块一样。

常量可以被认为是在 C 语 言中用 # define 定义的:这里会有元数据开销，但没有运行开销。“在 C 语言中，我应该使用 # define 还是 static，“很大程度与这个问题是相同的，“在 Rus t语言中，我是应该使用 const 还是 st atic”。

属性

在 Rust 语言中，声明可以用 ‘attributes’ 来注释。它们看起来像下面这样：

```
#[test]
```

或者是像这样：

```
#![test]
```

两者的区别是!，！ 改变了属性所能够适用的事物：

```
#[foo]
struct Foo;

mod bar {
#![bar]
}
```

#(foo) 属性应用到下一个项目，而这个项目就是结构体声明。 #![bar] 属性适用于包含它的项目，这种属性是一个 mod 声明。否则，它们是相同的。这样都在某种程度上改变项了目的意义。

例如，考虑这样一个函数：

```
#[test]
fn check() {
assert_eq!(2, 1 + 1);
}
```

这是用 #[test] 来标志的。这意味着它是特殊的:当您运行测试时，该函数将执行。当你和往常一样编译时， #[test] 甚至不会被包括在编译的范围之内。这个函数是现在一个测试函数。

属性也可能含有额外的数据：

```
#[inline(always)]
fn super_fast_fn() {
或者甚至是关键字和值:
#[cfg(target_os = "macos")]
mod macos_only {
```

Rust 属性被用于许多不同的事情。这有一个属性的完整引用列表。目前，不允许用户创建自己的属性，只能由 Rust 编译器来定义它们。

type 别名

你可以使用 type 关键字声明另一类型的别名：

```
type Name = String;
```

然后，你可以就像使用一个真正的类型一样使用这种类型：

```
type Name = String;

let x: Name = "Hello".to_string();
```

但是请注意，这是一个别名，不完全是一个新类型。换句话说，因为 Rust 是强类型的，所以你不能比较两个不同类型：

```
let x: i32 = 5;
let y: i64 = 5;

if x == y {
   // ...
}
```

这会产生这样的结果：

```
error: mismatched types:
 expected `i32`,
found `i64`
(expected i32,
found i64) [E0308]
 if x == y {
 ^
```

但是，如果我们有一个别名：

```
type Num = i32;

let x: i32 = 5;
let y: Num = 5;

if x == y {
   // ...
}
```

这个编译没有错误。无论如何，Num 类型的值和 i32 类型的值是相同的。

你还可以使用泛型类型别名：

```
use std::result;

enum ConcreteError {
Foo,
Bar,
}

type Result<T> = result::Result<T, ConcreteError>;
```

这将创建一个 Result 类型的专门的版本，它总是有一个针对 `Result< T E >` 的 E 部分的 ConcreteError 。这常被用在标准库来为每一部分创建自定义错误。例如，`io::Result` 。

类型转换

Rust 以其安全性为重点，为不同的类型之间的转换提供了不同的方法。首先，as 用于数据类型安全转换 。相反，transmute 允许类型之间的任意转换，是 Rust 的最危险的特征！

as

as 关键字可以做基本的转换：

```
let x: i32 = 5;

let y = x as i64;
```

然而，它只允许某些类型的转换：

```
let a = [0u8, 0u8, 0u8, 0u8];

let b = a as u32; // four eights makes 32
```

这个错误如下：

```
error: non-scalar cast: `[u8; 4]` as `u32`
let b = a as u32; // four eights makes 32
```

这是一个“非标量转换”，因为我们这里有多个值：数组的四个元素。这些类型的转换是非常危险的，因为他们对多个底层结构的实现方式做了假设。为此，我们需要一些更危险的东西。

transmute

transmute 函数是由内在编译器提供的，它做的非常简单，但非常可怕。它告诉 Rust 把一种类型的值当作它是另一种类型的值。它这样做不管类型检查系统，完全信任你。

在我们前面的例子中，我们知道，一个数组的四个元素 u8 正好表示一个 u32，所以我们做这样的转换。使用 transmute 而不是 as，Rust 代码如下：

```
use std::mem;

unsafe {
let a = [0u8, 0u8, 0u8, 0u8];
```

```
let b = mem::transmute::<[u8; 4], u32>(a);
}
```

为了成功编译我们必须在一个 unsafe 块中封装操作。从技术上讲，只有 mem::transmute 调用自己本身的时候需要在在代码块中。但是在把所有相关的一切封装在内的情况下是可以调用自己的，所以你知道在哪里看。在这种情况下，a 的细节也很重要，它们确实在代码块中。你会看到其它风格的代码，有时上下文是太远，将所有代码封装在 unsafe 不是一个好主意。

虽然 transmute 确实很少检查，它至少能确保类型是相同大小的。看下面这段代码：

```
use std::mem;

unsafe {
let a = [0u8, 0u8, 0u8, 0u8];

let b = mem::transmute::<[u8; 4], u64>(a);
}
```

这个错误是：

```
error: transmute called on types with different sizes: [u8; 4] (32 bits) to u64
(64 bits)
```

除此之外，你需要自己学习相关知识！

关联类型

关联类型是 Rust 的类型系统一个强大的部分。它们与一个“类型家族”的概念有关，换句话说，就是将多种类型组合在一起。这样描述有点抽象，所以让我们深入理解一个例子。如果你想写一个特征，名字为 Graph ，你有2种类型是通用的：节点类型和边类型。所以你可以写一个特征， `Graph<N, E>` ，看起来像这样：

```
trait Graph<N, E> {
fn has_edge(&self, &N, &N) -> bool;
fn edges(&self, &N) -> Vec<E>;
// etc
}
```

但是这类代码结束的不太合适。例如，任何想要以 Graph 为参数的函数，现在也需要在节点和边类型上是通用的：

```
fn distance<N, E, G: Graph<N, E>>(graph: &G, start: &N, end: &N) -> u32 { ... }
```

我们的距离计算与我们的类型 Edge 无关，所以填充在声明中的 E 只是一个无关变量。

我们真正想说的是,边 Edge 和 Node 类型一起构成 Graph 类。我们可以称它们为关联类型：

```
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，我们的客户端可以抽象出一个给定的 Graph：

```
fn distance<G: Graph>(graph: &G, start: &G::N, end: &G::N) -> uint { ... }
```

这里不需要处理 Edge 类型!

让我们更详细地去学习这些知识。

定义关联类型

让我们构建特征 Graph。这里是它的定义：

```
trait Graph {
type N;
type E;

fn has_edge(&self, &Self::N, &Self::N) -> bool;
fn edges(&self, &Self::N) -> Vec<Self::E>;
}
```

这很简单。关联类型要在特征函数体内使用 type 关键字。

这些 type 声明可以有和函数一样的功能。例如，如果我们希望我们的 N 类型实现 Display，所以我们可以打印节点，我们可以这样做：

```
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>;
}
```

关联类型的实现

就像任何特征，使用关联类型的特征要使用 impl 关键字提供实现。这是一个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()
```

```
}
}
```

这个简单的实现总是返回 true 和一个空的 `Vec <Edge>`，但它给了你一个如何实现这种功能的办法。我们首先需要三个struct，一个图，一个节点，一个边。如果使用不同的类型，那也行，我们只是要用所有这三个变量的 struct。

接下来是 impl，它就像实现任何其他特征一样。

在这里,我们使用 = 定义我们的关联类型。特征使用的名字放在 = 的左侧，我们实现的具体类型放在 = 右边。最后，我们使用函数中声明的具体类型。

关联类型的特征对象

还有一个语法我们应该谈论一下：特征对象。如果你想创建一个关联类型的特征对象，如下：

```
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>;
```

我们不能像这样创建一个特征对象，因为我们不知道关联类型。相反，我们可以写：

```
let graph = MyGraph;
let obj = Box::new(graph) as Box<Graph<N=Node, E=Edge>>;
```

N=Node 语法允许我们为 N 类型参数提供一个具体的类型，Node。E=Edge 也一样。如果我们没有提供这个约束，我们无法确定哪一个 impl 与这个特征对象相匹配。

全类型

大多数类型都有一个特定的大小，以字节为单位，是在编译时可知。例如，一个 32 位大的 i32 或四个字节。然而，也有一些用于表达的类型，但没有指定的大小。这些被称为全类型或动态大小类型。有一个例子是[T]。这个类型代表了序列中一定数量的 T。但是我们不知道有多少，所以不知道大小。

Rust 使用许多这样的类型，但是他们有一些限制。有如下三种：

1. 我们只能通过指针操作一个全类型的实例。一个 &[T] 就可以了，但[T]不可以。
2. 变量和参数没有动态大小类型。
3. 只有 struct 的最后一个字段可能有一个动态大小类型；其他字段没有。动态大小类型。

所以，为什么要找麻烦呢？好吧，因为 [T] 只能在指针后使用如果我们没有语言支持全类型，就不可能写出下面代码：

```
impl Foo for str {
```

或

```
impl<TFoo for [T] {
```

相反，你必须这样写：

```
impl Foo for &str {
```

这意味着，只能实现引用,而不是指针的其他类型。impl str，所有指针，包括（在某种程度上,首先有一些bug要修复）用户定义的定制智能指针，可以使用这个 impl。

?Sized

如果你想写一个接受一个动态大小类型的函数，你可以使用特殊的约束，?Sized：

```
struct Foo<T: ?Sized> {
f: T,
}
```

这个？，读作“T可能的大小”，意味着这个绑定很特别：它让我们匹配更多的种类,而不是更少。就像每一个 T 隐式包含 `T: Sized`，并且？可以撤销默认。

操作符和重载

Rust 允许有限形式的操作符重载。有一些操作符能够被重载。为了支持一个类型之间特定的操作符，有一个你可以实现的特定的特征，然后重载操作符。

例如，可以用 Add 特征重载 + 操作符。

```
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 函数中，你可以在两个 Point之间使用 + 操作符，因为我们可以使用 Point 的方法 `Add<Output=Point>` 。

有许多操作符可以以这种方式被重载，所有的关联特征都在 `std::ops` 模块中。看看完整列表的文档。

这些特征的实现遵循一个模式。让我们看看 Add 的更多细节：

```
pub trait Add<RHS = Self> {
type Output;
```

```
fn add(self, rhs: RHS) -> Self::Output;
}
```

这里总共三种类型包括：impl Add for的类型，默认为 Self 的 RSH，还有 Output。一个表达式 `let z = x + y`，x 是 Self 类型，y 是 RSH 类型，还有 z 是 Self::Output 类型。

这可以让你这样做：

```
let p: Point = // ...
let x: f64 = p + 2i32;
```

'Deref'强制转换

标准库提供了一个特殊的特征，Deref。它通常用于重载 * ，取消引用运算符：

```
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);
}
```

这用于编写自定义指针类型。然而，有一个与 Deref 相关的语言特征：‘deref 强制转换’。规则是这样的：如果你有一个类型 U，它实现 `Deref<Target=T>`，&U 的值自动强制转换为 &T。这里有一个例子：

```
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);
```

在一个值前使用 & 需要一个引用。所以 owned 是一个 String，&owned 是一个 &String，并且由于 `impl Deref<Target=str> for String`，&String 参考传入函数 foo() 的 &str。

就这样。这条规则是 Rust 为你自动转换的少有的几处之一，但它增加了很大的灵活性。例如，类型 `Rc<T>` 实现 `Deref<Target=T>`，所以它的工作原理如下：

```
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);
```

所有我们所做的就是把我们的 String 封装到 `Rc<T>`。但是我们现在可以把 `Rc<String>` 传到任何有 String 的地方。foo 的声明并没有改变，但能实现与其它类型一样的功能。这个例子有两个转换：`Rc<String>` 转换为 String，然后 String 转换为 &str。Rust 会这样做尽可能多的次数直到类型匹配。

标准库提供的另一个很常见的实现是：

```
fn foo(s: &[i32]) {
// borrow a slice for a second
}

// Vec<T> implements Deref<Target=[T]>
let owned = vec![1, 2, 3];

foo(&owned);
```

向量可以取消对程序片的引用。

Deref 和方法调用

Deref 调用方法时也起作用。换句话说，Rust 有相同的两件事：

```
struct Foo;

impl Foo {
fn foo(&self) { println!("Foo"); }
}

let f = Foo;

f.foo();
```

尽管 f 不是引用，但是函数 foo 中传入 &self 就会起作用。这是因为这些东西是相同的：

```
f.foo();
(&f).foo();
(&&f).foo();
(&&&&&&&&f).foo();
```

`&&&&&&&&&&&&&&&&Foo` 类型的值仍然可以有定义在 Foo 上的方法，因为编译器会插入许多 * 操作只要程序正确运行。因为它的插入* s，就要使用 Deref。

宏命令

现在你已经了解了很多 Rust 为抽象和重用代码提供的工具。这些代码重用的单元有丰富的语义结构。例如，函数有一个类型声明，有特征约束的类型参数，重载的函数必须属于一个特定的特征。

这种结构意味着 Rust 的核心抽象有强大的编译时正确性检查。但这是以灵活性的减少为代价。如果你从表面上识别重复代码的模式，你可能会发现像一个泛型函数，特征，或者 Rust 语义中其它任何东西一样表达模式是很困难的或者是很繁琐的。

宏定义允许我们实现语法水平上的抽象。宏调用的简单来说就是“扩大”语法形式。这种扩张发生在编译早期，在任何静态检查之前。因此，宏可以捕获许多代码重用模式，这些是 Rust 的核心抽象做不到的。

缺点是基于宏的代码比较难以理解,因为更少的内置规则可以使用。像一个普通的函数，可以使用一个功能良好的宏而无需理解它的实现。然而，很难设计一个功能良好的宏！此外，在宏代码中的编译错误难以解释，因为他们用扩展代码来描述问题，而不是开发人员使用的源代码级别的形式。

这些缺点是宏的重要的“特性”。这并不是说宏不好，它是有时是 Rust 的一部分，因为他们真正需要简洁、抽象的代码。我们要记住这个折衷。

定义一个宏

你可能看到过宏 vec!，用于初始化包含任意数量元素的向量。

```
let x: Vec<u32> = vec![1, 2, 3];
```

这不可能是一个普通的函数，因为它有任意数量的参数。但我们可以把它想象成下面语法的简称

```
let x: Vec<u32> = {
let mut temp_vec = Vec::new();
temp_vec.push(1);
temp_vec.push(2);
temp_vec.push(3);
temp_vec
};
```

我们可以使用一个宏实现这个函数：

```
macro_rules! vec {
( $( $x:expr ),* ) => {
{
```

```
let mut temp_vec = Vec::new();
$(
temp_vec.push($x);
)*
temp_vec
}
};
}
```

哇，这是一个新的语法！让我们分解一下。

```
macro_rules! vec { ... }
```

这表示我们定义一个名为 vec 的宏，正如 fn vec 将定义一个名为 vec 的函数。换句话说，我们用一个感叹号非正式地编写一个宏的名字，例如 vec!。感叹号是调用语法的一部分，用来区分一个宏和一个普通的函数。

匹配

宏是通过一系列的规则来定义的，这些规则是用来模式匹配的。上面，我们有

```
( $( $x:expr ),* ) => { ... };
```

这就像一个匹配表达式的处理器，但编译时匹配发生 Rust 语法树。在最后的实例后面分号是可选的。= > 左边”的“模式”被称为“匹配器”。这些在语言里都有自己的小语法。

匹配器 $x:expr 通过将语法树绑定到元变量 $x 来匹配任何 Rust 表达式。标识符 expr 是一个片段说明符，完整的标识符是在本章后面的枚举。`$(...),*` 周围的匹配器将匹配零个或多个表达式，这些表达式由逗号分隔开。

除了特殊的匹配器语法，任何出现在一个匹配器中的 Rust 指令必须完全匹配。例如，

```
macro_rules! foo {
(x => $e:expr) => (println!("mode X: {}", $e));
(y => $e:expr) => (println!("mode Y: {}", $e));
}

fn main() {
foo!(y => 3);
}
```

会打印出

```
mode Y: 3
```

通过函数

```
foo!(z => 3);
```

我们会得到以下编译错误

```
error: no rules expected the token `z`
```

扩展

在大多数情况下，宏观规则的右边是普通的 Rust 语法。但我们可以拼接一些被匹配器捕获的语法。下面是一个典型的例子：

```
$(
temp_vec.push($x);
)*
```

每个匹配表达式 $x 在宏扩展中产生一个单独的 push 语句。扩展的副本与匹配器中的副本是同步的。

因为 $x 已经声明为一个表达式匹配，我们不要在右侧重复 :expr。同样，我们不能把逗号作为重复操作符的一部分。相反，我们在重复的块内有一个终止分号。

另一个细节：vec! 宏在右侧有两个双括号。他们通常组合如下：

```
macro_rules! foo {
() => {{
...
}}
}
```

外层的括号是语法 macro_rules! 的一部分。实际上，你也可以使用 () 或 []。他们只是将右侧划分为一个整体。

内层括号是扩展语法的一部分。记住，vec! 宏被用在一个表达式上下文。为了写一个包含多个语句的表达式，包括 let-bindings，我们需要使用一个块。如果你的宏扩展到一个单个表达式，你就不需要这些额外的括号。

注意，我们从未声明宏产生一个表达式。事实上，这是不确定的，直到我们作为一个表达式使用宏。小心，你可以编写一个宏，它的扩展可以在多个上下文中起作用。例如，数据类型的简写作为一个表达式或模式是有效的。

重复

重复操作符遵循以下两个主要规则：

1. `$(...)*` 为它包含的所有 $name 同步处理一个重复“层”，并且
2. 每个 $name 必须至少在它能匹配的尽可能多的 `$(...)*` 下。如果它在更多的重复操作符下，它会适当的复制。

这个结构复杂的宏说明了变量从外层重复层的复制。

```
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]);
}
```

上面包含了大部分的匹配器语法。这些例子使用 `$(...)*` ，这是一种“零个或多个”匹配。或者你可以写 `$(...)+` 进行“一个或多个”匹配。两种形式都可选地包括一个分隔符，它可以是任何除 `+` 或 `*` 的符号。

该系统是基于“Macro-by-Example”的。

卫生

一些语言通过使用简单的文本替换来实现宏，从而导致各种各样的问题。例如，这个 C 程序打印 13，而不是预期的 25。

```
#define FIVE_TIMES(x) 5 * x

int main() {
```

```
printf("%d\n", FIVE_TIMES(2 + 3));
return 0;
}
```

扩展后，我们有 5 * 2 + 3，乘法有比加法更高的优先级。如果你使用很多 C 宏，你可能知道以避免这个问题的通用方法，还有其它五六种方法。在 Rust 里，我们不必担心这些。

```
macro_rules! five_times {
($x:expr) => (5 * $x);
}

fn main() {
assert_eq!(25, five_times!(2 + 3));
}
```

元变量 $x 被解析为一个表达式节点，并保持它在语法树上的位置，即使在替换以后。

宏观系统的另一个常见的问题是“变量捕获”。这里有一个 C 宏，使用 a GNU C extension模拟 Rust 的表达式块。

```
#define LOG(msg) ({ \
int state = get_log_state(); \
if (state > 0) { \
printf("log(%d): %s\n", state, msg); \
} \
})
```

这是一个简单发生严重故障的用例：

```
const char *state = "reticulating splines";
LOG(state)
```

这可以扩展为

```
const char *state = "reticulating splines";
int state = get_log_state();
if (state > 0) {
printf("log(%d): %s\n", state, state);
}
```

命名为 state 第二个变量覆盖了第一个变量。这是一个问题,因为 print 语句应该参考这两个变量。

下面 Rust 宏可以达到预期结果。

```
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);
}
```

这个能起作用是因为 Rust 有一个卫生宏系统。每个宏扩展发生在一个独特的“语法语境”，每个变量被产生它的语法语境所标记。好像 main 里面的变量 state 在宏里被涂上不同的颜色，因此他们互相不冲突。

这也限制了宏在调用点引入新的绑定的能力。以下代码就不会起作用：

```
macro_rules! foo {
() => (let x = 3);
}

fn main() {
foo!();
println!("{}", x);
}
```

相反，你需要通过变量名调用，所以它被正确的语法语境所标记。

```
macro_rules! foo {
($v:ident) => (let $v = 3);
}

fn main() {
foo!(x);
println!("{}", x);
}
```

这适用于 let 绑定和循环标签，而不适用于 items。那么下面的代码可以通过编译：

```
macro_rules! foo {
() => (fn x() { });
}

fn main() {
foo!();
```

```
x();
}
```

递归宏

一个宏的扩展可以包括更多的宏调用，包括调用正在扩展的同一宏。这些递归宏用于处理树形结构输入，正如下面(简单的) HTML 速记所示：

```
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

为了看到宏扩展的结果，运行 rustc --pretty expanded。输出代表一个整体运行结果，所以你也可以把结果存入 rustc，它有时候会比原始的编译产生更好的错误信息。注意，如果多个同名的变量(但在不同的语法语境内)在

相同的范围内起作用，--pretty expanded 的输出可能有不同的意义。在这种情况下，--pretty expanded，hygiene 会告诉你关于语法语境的情况。

rustc 提供了两种语法扩展以帮助宏调试。现在,他们是不稳定的。

1. log_syntax!(...) 将其参数打印到标准输出，在编译时，没有“扩展”。
2. trace_macros!(true) 在每次宏扩展时产生编译器信息。在扩展结束时使用 trace_macros!(false) 。

语法要求

即使当 Rust 代码包含 un-expanded 宏时，它可以解析为一个完整的语法树。这个属性对编辑器和其他处理代码的工具是非常有用的。它也对 Rust 宏系统的设计产生一些后果。

一个后果是，解析一个宏调用时 Rust 必须确定宏是否代表

- 零个或多个项目，
- 零个或多个方法，
- 一个表达式，
- 一个语句，或
- 一个模式。

在代码块中的宏调用可以代表一些项目，或者一个表达式或语句。Rust 使用一个简单的规则来消除这个不确定性。宏调用的代表项目必须是

- 由花括号分隔开，例如 foo! { ... } ，或者
- 由一个分号终止，例如 foo!(...);

扩展前解析另一个后果是宏调用必须由有效 Rust 符号组成。此外，圆括号，方括号，花括号必须在一个宏调用中是平衡的。例如，foo!(]是禁止的。这允许 Rust 知道宏调用在哪里结束。

更正式地，宏调用体必须是一个“标记树”序列。一个标记树递归地定义为

- 一系列匹配的由 () ，[] ，或 {} 包围的标记树，或者
- 任何其它单个标记

在一个匹配器中，每个元变量都有一个“片段说明符”，来识别匹配哪些语法形式。

- ident：标识符。例如： `x; foo` 。
- path：一个合格的名字。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：单个语句。例如： `let x = 3` 。
- block：一个括号分隔的语句序列。例如： `{ log(error, "hi"); return 12; }` 。
- item：一个项目。例如： `fn foo() { }; struct Bar;` 。
- meta：一个 "元项目"， 在属性中建立的。 例如： `cfg(target_os = "windows")` 。
- tt：一个单个标记树。

还有其他关于元变量后下一个标记的附加规则：

- 变量 expr 后必须加下面中的一个： => , ;
- 变量 ty 和 path 后必须加下面中的一个： => , : = > as
- 变量 pat 后必须加下面中的一个：=> , =
- 其它变量后可能要加其它符号。

这些规则提供一些在不破坏现有宏的情况下，Rust 语法发展的灵活性。

宏系统不处理解析的不明确性。例如，语法 `$($t:ty)* $e:expr` 总是无法解析，因为解析器将被迫选择解析 $t 和解析 $e。将调用语法改为在前面加一个独特的符号可以解决这个问题。在这种情况下，你可以编写 `$(T $t:ty)* E $e:exp` 。

范围和宏导入/导出

宏在编译的早期阶段名称解析前被扩展。一个缺点是，相对于其他结构的语言，范围工作原理不同。

宏的定义和扩展都发生在一个 crate 资源的深度优先，词序遍历。所以一个定义在模块范围内的宏对在同一个模块内任何后续代码是可见的，其中包括任何后续的孩子 mod 项目的主体。

一个定义在单个 fn 的体内的宏，或其它不在模块范围的任何地方，只有在这个项目内是可见的。

如果一个模块有 macro_use 属性，其宏在它的孩子模块的 mod 项目后的父模块中是可见的。如果父模块也有 macro_use 属性，那么宏在父模块的 mod 项目后的祖父模块中也是可见的，等等。

macro_use属性也可以出现在 extern crate。在这种情况下它控制从 extern crate 加载哪些宏，如

```
#[macro_use(foo, bar)]
extern crate baz;
```

如果属性被简单定义如 #[macro_use]，所有宏被加载。如果没有 #[macro_use] 那么 宏就不能被加载。只有定义 #[macro_export] 属性的宏可能被加载。

为了加载没有连接到输出的 crate 的宏，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
```

当用 #[macro_use] extern crate 加载这个库时，只有 m2 将被导入。

Rust 参考有一个宏相关属性的列表。

变量 $crate

进一步困难发生在当一个宏在多个 crates 被使用。也就是 mylib 定义如下

```
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 只在 mylib 起作用，同时 inc_b 只能在库外起作用。此外，如果用户在另一个名字下引入 mylib ，inc_b 将失去作用 。

Rust 没有针对 crate 参考的卫生系统，但它确实提供了一个解决这个问题的简单方法。在一个从一个名为foo的 crate 引入的宏，特殊宏变量 $crate 将扩展到 ::foo。相反，当一个宏被定义，然后在同一 crate 中被使用，$crate 就不会扩展。这意味着我们可以这样写

```
#[macro_export]
macro_rules! inc {
($x:expr) => ( $crate::increment($x) )
}
```

来定义一个在库内库外的宏。函数名也扩展到 ::increment 或者 ::mylib::increment。

为了保持这个系统简单而正确，#[macro_use] extern crate ... 可能只出现在 crate 的根部，而不是 mod 内部。这将确保 $crate 是一个标识符。

深端

介绍性章节曾今提到过递归宏，但它没有给出完整的描述。递归宏是有用的另一个原因：每个递归调用给你匹配宏参数的另一个机会。

作为一个极端的例子，尽管不明智，在 Rust 的宏系统实现位循环标记自动机是可能的。

```
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：使用宏来减少上述 bct! 宏的定义中的复制。

常见的宏

下面是一些你会在 Rust 代码中常见的宏。

panic!

这个宏会导致当前线程的叛逆。你可以给它一个消息产生叛逆：

```
panic!("oh no!");
```

vec!

vec !宏在整本书被使用，所以你可能已经见过了。它毫不费力地创建 Vec：

```
let v = vec![1, 2, 3, 4, 5];
```

它还允许你用重复的值构建向量。例如，一百个 0：

```
let v = vec![0; 100];
```

assert! 和 assert_eq!

这两个宏用于测试。assert! 传入一个布尔值，assert_eq! 传入两个值并且比较它们。像下面这样：

```
// A-ok!

assert!(true);
assert_eq!(5, 3 + 2);

// nope :(

assert!(5 < 3);
assert_eq!(5, 3);
```

try!

try! 用于错误处理。它传入可以返回 `Result<T, E>` 的参数，并给出 T 如果它是 Ok，并且返回 Err(E) 。像这样：

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

fn foo() -> std::io::Result<()> {
let f = try!(File::create("foo.txt"));

Ok(())
}
```

这比这样做干净：

```
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!

当你认为一些代码不应该执行是使用这个宏：

```
if false {
unreachable!();
}
```

有时，编译器可能会让你有一个不同的分支，你知道它永远不会运行。在这些情况下，使用这个宏，这样如果你错了，你会得到一个关于它的 panic! 。

```
let x: Option<i32> = None;

match x {
Some(_) => unreachable!(),
None => println!("I know x is None!"),
}
```

unimplemented!

当你想让你的函数检查类型，不想担心写出函数的主体时 unimplemented! 宏可以被使用。这种情况的一个例子是你想要一次同时用多个所需的方法实现一个特征。把其它的定义为 unimplemented! 直到你准备写他们。

程序宏

如果 Rust 宏系统做不到你需要的，你可能想要编写一个编译器插件。相比 macro_rules !宏，这需要更多的工作，不太稳定的接口，bugs 可能更难追踪。作为交换你获得了在编译器内运行任意 Rust 代码的灵活性。这就是语法扩展插件有时被称为“程序宏”的原因。

在效率和可重用性方面， vec! 在 libcollections 的实际定义不同于这里介绍的。

原始指针

Rust 在标准库有许多不同的智能指针类型，但是有两种特别的类型。Rust 的安全来自于编译时检查，但原始指针没有这样的保证，使用起来不安全。

*const T 和 *mut T 在 Rust 中被称为“原始指针”。有时，当写库的某些类型时，出于某种原因你需要绕过 Rust 的安全保证。在这种情况下，你可以使用原始指针来实现你的库，同时为给用户一个安全接口。例如，允许 * 指针起别名，允许他们被用来写共享类型，甚至线程安全共享内存类型（ Rc和 Arc 类型在 Rust 中都被完全实现）。

这里要记住原始指针与其他指针类型是不同的地方。他们：

- 不能保证指有效内存，甚至不保证非空（不像 Box 和 & ）；
- 不像 Box ，没有任何自动清理功能，所以需要手动管理资源；
- 是原始旧数据，也就是说，他们不转移指向，不像 Box ，因此 Rust 编译器不能防止像 use-after-free 一样的漏洞；
- 与 & 不用，缺乏任何形式的生命周期，所以编译器无法推断悬空指针；
- 不能保证别名使用或者易变性，不同于直接通过 *const T 不能产生变化。

Basics

创建一个原始指针是绝对安全的：

```
let x = 5;
let raw = &x as *const i32;

let mut y = 10;
let raw_mut = &mut y as *mut i32;
```

然而，非关联化的指针是不安全的。这是行不通的：

```
let x = 5;
let raw = &x as *const i32;

println!("raw points at {}", *raw);
```

它会给出这样的错误：

```
error: dereference of unsafe pointer requires unsafe function or block [E0133]
 println!("raw points at{}", *raw);
 ^~~~
```

原始指针废弃时，你承担这样的后果，那就是它不指向那个正确的地方。因此，你需要 unsafe：

```
let x = 5;
let raw = &x as *const i32;

let points_at = unsafe { *raw };

println!("raw points at {}", points_at);
```

要了解更多关于原指针的操作，请看 API 文档。

FFI

对 FFI 来说原指针是很有用的：Rust 的 *const T 和 *mut T 与 C 的 const T* 和 T* 是相似的。更多关于它的使用，请看 FFI 章节。

引用和原始指针

在运行时，一个原始指针 * 和一个指向同一块数据的引用具有相同的表示。事实上，一个 &T 引用将隐式强制转换为安全代码中的 *const T 原始指针并且与常量 mut 相似（两种强制转换可以显式地被执行，值分别是 *const T 和 *mut T）。

相反，从 *const 指向引用 & 是不安全的。 &T 总是有效的，因此至少，原始指针 *const T 必须指向一个 T 类型的有效实例。此外，由此产生的指针必须满足引用的别名使用和可变性规则。编译器假定这些属性对任何引用都是真的，不管他们是如何创建的，因此任何从原始指针的转换都认为它们有这些属性。程序员必须保证这一点。

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;
}
```

&*x 解除引用要使用 transmute。transmute 是非常强大的，更受限制的操作也能正确使用；例如，它要求 x 是一个指针（而不像 transmute）。

unsafe

Rust 的主要缺点是其对行为的强大的静态担保。但安全检查是保守的：有些程序实际上是安全的，但是编译器无法证实这是真的。为了写这种程序，我们需要告诉编译器放宽限制。为此, Rust 有一个关键词，unsafe。代码使用 unsafe 比正常的代码有更少的限制。

让我们复习语法，然后我们将讨论语义。unsafe 在两种情况下被使用。第一个情况是标记一个函数为不安全：

```
unsafe fn danger_will_robinson() {
// scary stuff
}
```

例如，所有函数调用 FFI 时必须标记为 unsafe。第二个使用 unsafe 情况是不安全块：

```
unsafe {
// scary stuff
}
```

能够明确划定可能有漏洞的代码是非常重要的，并且这些漏洞能造成大问题。如果 Rust 程序段错误，你能确定这部分中哪里标记为 unsafe。

“safe”是什么意思?

在 Rust 的上下文中，safe 的意思是“不做任何不安全的事。” 这很简单!

好吧，让我们再试一次：什么是不安全的? 这里有一个列表：

- 数据竞争
- 非关联化空的或悬空的原始指针
- 读取 undef（未初始化的）内存
- 打破原始指针的指针别名规则。
- &mut T 和 &T 遵循LLVM的作用域 noalias 模式，除非 &T 包含一个 UnsafeCell<U>。不安全的代码必须不违反这些别名担保。
- 在没有 UnsafeCell<U> 的情况下，改变一个不可变的值/引用
- 通过这些编译器特性调用未定义的行为：
    - 有 std::ptr::offset 的对象的越界索引，除了一个字节结束过去这是允许的。

    - 在重叠的缓冲区时使用 std::ptr::copy_nonoverlapping_memory (memcpy32/memcpy64 特性)
- 原始类型的无效值，即使在私有作用域/局部:
    - 空/悬空的引用或盒子
    - 在一个 bool 中除了 false (0) 或 true (1) 的值
    - 在一个不包括类型定义的 enum 的一个判别式
    - 在一个大于或等于 char::MAX 的 char 里的一个值
    - 在一个 str 里的 Non-UTF-8 类型序列
    - 从外部代码展开到 Rust 或 从 Rust 展开到到外部代码。

这是很多东西。注意到各种各样的不好的但没有标记为 unsafe 的行为是很重要的。

- 死锁
- 从私有作用域读取数据
- 由于引用计数周期引起的泄漏
- 没有调用析构函数的情况下退出
- 发送信号
- 访问/修改文件系统
- 整数溢出

Rust 不能防止各种各样的软件问题。bug 代码可以并将写在 Rust。这些行为并不好，但他们不符合 unsafe。

Usafe 超级能力

在不安全的函数和不安全的代码块内，Rust 通常会让你做三件通常不会做的事。以下就是这三件事:

- 访问或更新一个静态可变的变量。
- 解除引用原始指针。
- 调用不安全的函数。这是最强大的能力。

就这样。重要的是，例如， unsafe 不会“关掉借用查器”。将 unsafe 添加到一些随机 Rust 代码并没有改变其语义，它不会开始接受任何东西。

但是它会让你写一些打破一些规则的东西。让我们学习这三种能力。

访问或更新 static mut

Rust 有一个称为‘static mut’的特性，它允许可变的全局状态。这样做会导致数据竞赛，因此本身是不安全的。有关详细信息，请参本书的静态部分。

解除引用一个原始指针

原始指针让你做任意指针的运算，会导致许多不同的内存安全问题。在某种意义上，一个任意指针的解除引用的能力是你可以做的最危险的事情。更多关于原始指针，请看本书相关部分。

调用 unsafe 函数

这最后的能力关于 unsafe 的两个方面：您只能调用一个 unsafe 块内标记 unsafe 的函数。

这种能力是强大的。Rust 为 unsafe 函数提供一些编译器特性，绕过安全检查和一些安全功能，换来安全速度。

我将再次重复：即使你可以在 unsafe 块和函数中做任意事情，并不意味着你应该这样做。虽然你坚持不变量编译器仍然将起作用，所以要小心！

6

Nightly Rust

Rust 提供了三个版本渠道：nightly，beta，还有stable。不稳定特性只在 Nightly Rust 有效。这个过程的更多细节，请参见 “Stability as a deliverable”。

安装 nightly Rust，你可以使用 rustup.sh：

```
$ curl -s https://static.rust-lang.org/rustup.sh | sh -s -- --channel=nightly
```

如果你担心使用 curl | sh 有潜在的不安全性，请阅读下面我们的免责声明。免费使用两个版本的安装和检查我们的安装脚本：

```
$ curl -f -L https://static.rust-lang.org/rustup.sh -O
$ sh rustup.sh --channel=nightly
```

如果你在使用 Windows 系统，请下载32位的安装程序或64位安装程序并运行它。

卸载

如果你决定你不在使用 Rust 了，我们会有点难过，但没关系。并不是每一个编程语言对每个人来说都是很好用的。只需运行卸载脚本：

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

如果你使用 Windows 安装程序，只要重新运行 .msi，它会给你一个卸载选项。

理所当然，当我们告诉你 curl | sh，有些人变得非常沮丧。基本上，当你这样做时，你相信维护 Rust 的人不会攻击你的电脑和做其它坏事。这是一个很好的本能！如果你是这些人中的一员，请查看从源代码构建 Rust 的文档，或官方二进制下载。

哦，我们还应该提到官方支持平台：

- Windows (7， 8， Server 2008 R2)
- Linux (2.6.18 或更新版本， 或不同版本)， x86 和 x86-64
- OSX 10.7 (Lion) 或更高版本， x86 和 x86-64

我们在这些平台上广泛测试 Rust，还有其他平台，像 Android。但这些都是最有可能的运行平台，因为他们都是最经常测试的平台。

最后，看一下 Windows 。Rust 认为 Windows 是发布的一个一流平台，但如实说来，Windows 体验不像 Linux / OS X 体验一样集成。我们正在努力做到这一点! 如果有什么不起作用，这就是一个漏洞。请让我们看看这是否真的发生。测试对 Windows 的每个提交就像测试任何其他平台一样。

如果你已经安装 Rust ，你可以打开一个 shell 和类型：

```
$ rustc --version
```

你应该看到版本号、提交哈希表，提交日期和构建日期：

```
rustc 1.0.0-nightly (f11f3e7ba 2015-01-04) (built 2015-01-06)
```

如果你这么做了，Rust 已经安装成功！恭喜！

这个安装程序也在本地安装一个文档的副本，这样你就可以离线阅读。在 UNIX 系统中， `/usr/local/share/doc/rust` 就是安装地址。在 Windows 中，安装在一个 `share/doc` 目录中，无论你将 Rust 安装在何处。

如果没有安装，有很多地方你可以得到帮助。最简单的就是 the #rust IRC channel on irc.mozilla.org ，你可以通过 Mibbit 访问到。单击该链接，你可以与其他 Rustaceans 交流，我们可以帮助你。还有其它的资源，包括用户的论坛和堆栈溢出。

编译器插件

简介

rustc 可以加载编译器插件，它是用户提供的库,这个库使用新语法扩展编译器的行为，lint 检查等。

插件是一个有指定的 registrar 函数的动态库，注册 rustc 扩展。其它库可以使用属性 `#![plugin(...)]` 加载这些扩展。想了解更多关于定义机制和加载插件，查看 rustc::plugin 文档。

如果存在，像 `#![plugin(foo(... args ...))]` 传递的参数不被 rustc 本身编译。他们通过注册表参数的方法提供给插件。

在绝大多数情况下，一个插件只能通过 #![plugin] 而不是通过一个 extern crate 项目来被使用。连接一个插件将会把所有 libsyntax 和 librustc 作为库的依赖关系。这通常是不必要的，除非你正在创建另一个插件。`plugin_as_library lint` 检查这些准则。

通常的做法是将编译器插件放入自己的库中，独立于库中被客户端使用的任何 macro_rules ! 宏或普通 Rust 代码。

语法扩展

插件可以用不同的方法扩展 Rust 的语法。一种语法扩展是程序宏。调用程序宏就像调用普通的宏一样，但扩展是由任意在运行时操纵语法树的 Rust 代码执行。

让我们写一个插件 `roman_numerals.rs` 实现罗马数字整数常量。

```
#![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);
}
```

然后我们可以像使用其他宏一样使用 rn !()：

```
#![feature(plugin)]
#![plugin(roman_numerals)]

fn main() {
```

```
assert_eq!(rn!(MMXV), 2015);
}
```

一个简单的 `fn(&str) -> u32` 的优势是：

- (任意复杂的)转换是在编译时完成的。
- 输入验证在编译时执行。
- 它可以扩展到模式中允许使用，它有效地给出了一个为数据类型定义新的文字语法的方法。

除了程序宏，你可以定义新的 derive-like 属性和其他类型的扩展。请看Registry::register_syntax_extension 和 SyntaxExtension enum。更多调用宏的例子，请见 regex_macros。

提示和技巧

有一些宏的调试技巧是适用的。

您可以使用 syntax::parse 将标记树转化为更高级的语法元素如表达式：

```
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();
```

通过这些 libsyntax 解析代码我们可以知道解析基础结构是如何工作的。

为得到更准确的错误报告，保持所有你解析的代码的 span 。你可以将 spaned 封装到自定义数据结构中。

调用 ExtCtxt:span_fatal 会立即中止编译。最好不要调用 ExtCtxt:span_err 并返回 DummyResult，编译器可以继续并找到更多的错误。

为了打印语法片段进行调试，可以使用 `span\_note` 加上 `syntax::print::pprust::*_to_string` 。

上面的例子使用 AstBuilder::expr_usize 产生一个整数。除了 AstBuilder 特征，libsyntax 提供了一组 quasiquote 宏。他们没有正式文件并且非常粗糙的。然而，它的实现可能是一个改进的一个普通的插件库 quasiquote 的好的起点 。

Lint 插件

插件可以通过对额外的代码类型、安全等等的检查来扩展 Rust 的 lint 基础结构。在 `src/test/auxiliary/lint_plugin\_test.rs` 中你可以看到一个完整的例子。这个例子的核心如下：

```
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);
}
```

然后代码

```
#![plugin(lint_plugin_test)]

fn lintme() { }
```

将会产生一个编译器警告：

```
foo.rs:4:1: 4:16 warning: item is named 'lintme', #[warn(test_lint)] on by default
foo.rs:4 fn lintme() { }
```

lint插件的组件如下：

- 一个或多个 定义静态的 Lint 结构的 declare_lint ! 调用；
- 一个控制 int pass (here, none) 所需的任何 state 的 struct；

- 一个定义如何检查每个语法元素的 LintPass 实现。一个 LintPass 可能为几个不同的 Lint 调用span_lint，但应该通过 get_lints 方法注册它们。

Lint 通过遍历语法，但他们在编译的后期运行，在哪里可以获得类型信息。rustc 内置的 lint 大多使用相同的基础结构作为 lint 插件，并提供了一些说明如何访问类型信息的例子。

插件定义的 lint 是由通常的属性和编译器标志所控制，例如 `#[allow(test_lint)]` 或 `-A test-lint` 。通过合适的案例和标点符号的转换，这些标识符由第一个参数传递到 declare_lint ! 。

您可以运行 rustc -W help foo.rs 来看 rustc 知道的 lint 的列表，包括那些由foo.rs 加载的插件提供的列表。

内联汇编

由于低水平的操作和性能，人们希望直接控制 CPU。Rust 通过 asm！宏来支持使用内联汇编。语法大致匹配 GCC & Clang：

```
asm!(assembly template
   : output operands
   : input operands
   : clobbers
   : options
   );
```

任何 asm 的使用是特征封闭的（需要库的 #![feature(asm)] 允许），当然需要一个 unsafe 块。

```
注意:这里给出了 x86 和 x86-64 支持的例子，但所有平台都支持
```

汇编模板

assembly template 是唯一所需的参数，它必须是一个文字字符串（例如，""）

```
#![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();
// ...
}
```

（从现在开始省略 feature(asm) 和 #[cfg]。）

输出操作数，输入操作数，clobber 和选择项都是可选的，但是你必须添加正确的数量的：如果你跳过它们：

```
asm!("xor %eax, %eax"
:
:
: "{eax}"
   );
```

空格也没有关系：

```
asm!("xor %eax, %eax" ::: "{eax}");
```

操作数

输入和输出操作数遵循相同的格式："constraints1"(expr1), "constraints2"(expr2), ..."。输出操作数表达式必须是可变左值，或者没有分配内存：

```
fn add(a: i32, b: i32) -> i32 {
let c: i32;
unsafe {
asm!("add $2, $0"
 : "=r"(c)
 : "0"(a), "r"(b)
 );
}
c
}

fn main() {
assert_eq!(add(3, 14159), 14162)
}
```

如果你想在这个位置上使用真正的操作数，然而，你需要把花括号 { } 放在你想要的的寄存器两边，你需要加具体操作数的大小。对于低水平的编程这是非常有用的，在程序中使用哪个寄存器很重要：

```
let result: u8;
asm!("in %dx, %al" : "={al}"(result) : "{dx}"(port));
result
```

Clobbers

一些指令修改有可能持有不同值的寄存器，所以我们使用超时列表来指示编译器不承担加载到寄存器将保持有效的任何值。

```
// Put the value 0x200 in eax
asm!("mov $$0x200, %eax" : /* no outputs */ : /* no inputs */ : "{eax}");
```

输入和输出寄存器不需要被列出来，因为信息已经被给定约束传达。否则，任何其他被隐式或显式地使用的寄存器应该列出。

如果内联会更改代码寄存器， cc 应该指定为一个 clobber。同样，如果内联会修改内存，memory 还应该被指定。

选择项

最后一部分，options 是 Rust 特有的。形式是逗号分隔字符串（例如：:"foo", "bar", "baz"）。这是用于指定内联汇编的一些额外的信息：

当前有效的选项是：

1. *volatile* –这类似于在 gcc/clang 中指定_ _asm__ __volatile__(...)。
2. *alignstack*–某些指定堆的对齐某种方式（例如，SSE）的指令并说明这个指示编译器插入其通常堆栈对齐的代码的指令。
3. *intel* –使用 intel 语法而不是默认的 AT&T。

```
let result: i32;
unsafe {
   asm!("mov eax, 2" : "=`{eax}"(result) : : : "intel")
}  println!("eax is currently {}", result);
```

不依赖 stdlib

默认情况下， `std` 会链接到每一个 Rust 的封装对象。在一些场景下，这并不是很理想的，在需要的情境下使用 `#![no_std]` 属性使得程序可以独立于 `stdlib` 。

```
// a minimal library
#![crate_type="lib"]
#![feature(no_std)]
#![no_std]
```

很显然，这种方式比仅仅使用库有更长的声明周期：一个是使用 `#[no_std]` 来开始程序的执行，控制程序开始指针可以有两种方式：使用 `#[start]` 属性或覆盖 C `main` 函数。

以 `#[start]` 标记的函数可以以C语言的方式来传递命令行参数:

```
#![feature(lang_items, start, no_std, libc)]
#![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 {} }
```

为了覆盖插入式编辑器的 `main` 函数。一种方法是使用 `#![no_main]` 来关闭，然后使用正确 ABI 和名字来创建合适的标示，这些也需要覆盖编译器的名字。

```
`#![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 {} }`
```

编译器目前已经制造了一些假定的符号，它们可以在可执行文件来调用。通常，这些函数是由标准库提供的，如果没有它的话，用户就必须定义自己的。

这三个函数中的第一个，`stack_exhausted`，被调用何时堆栈溢出。此函数关于如何被调用和它必须做什么存在一些约束和限制。但是，如果堆栈限制寄存器没有被维护，然后一个线程经常有个无限的堆栈且这个函数不会被触发。

这三个函数中的第三个函数，`eh_personality`，用于编译器的错误机制。这通常会被映射到 GCC 的个人函数（参照 libstd 的实现来获取更多的信息），不会触发错误的对象就被认为是该函数没有被调用。最后一个函数，`panic_fmt`，也是用于编译器的错误机制的。

使用 libcore

注意：核心库的结构是不稳定的，这里特别建议尽量在能使用标准库的情况下就使用标准库。

在上述技术的基础上，我们就可以运行一些简单的 Rust 代码。标准库可以提供一些更理想的函数，但是，必须在 Rust 代码中主动去建立。在某些情况下，标准库也不是很有效，那么，可以使用 libcore.

核心库具有较少的依赖性，并且比标准库还精简。此外，核心库为编写惯用且高效的 Rust 代码提供了很多必要的函数。

举个例子，下面是关于 C 中的两个向量进行点乘的程序，这里使用了惯用的 Rust 实践。

```
`#![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，这里有个与上例中不同的附加lang 项目， `panic_fmt` .它必须由 libcore 的使用者来ID来定义，因为，核心库声明了这个警告，但是并没有定义它。这里的 `panic_fmt` lang 项目 就是警告的对象定义。它必须满足应用不会退出的条件。

正如上例中看到的那样，核心库就是用于在多种独立于平台的应用场景下支持 Rust.未来的库,如liballoc、会在libcore的基础上添加更多的功能，使的可以应用于其他特定的平台,但不变的是仍然会比标准库更精简便携。

内敛函数

注意：内敛函数永远不具有稳定接口，所以最好直接使用libcore的稳定接口，而不是直接使用内敛函数。

使用 `rust-intrinsic` ABI 可以使得内敛函数类似于外部函数接口方法一样去引入。比如，如果一段程序希望不依赖于上下文，同时又希望能够在不同类型见切换，然后高效的执行指针运算，那么这段代码可以通过如下的声明来引入这些方法。

```
extern "rust-intrinsic" {
fn transmute<T, U>(x: T) -> U;

fn offset<T>(dst: *const T, offset: isize) -> *const T;
}
```

调用其他任何外部函数接口方法都是不安全的。

Lang 项目

注意：lang 项目一般是由 Rust 发布的封装提供的，lang 项目本身具有不稳定的接口。最好使用官方发布的封装，不要使用自己编写的 lang 项目。

`rustc` 编译器具有可插拔的操作功能，它并不是硬编码成语言的，而是被以库的形式实现，并且以特殊标记来告知编译器它的存在。这个标记就是 `#[lang = "..."]`，并且有多种多样的值，多种多样的 lang 项目。

例如，`Box` 指针需要两个 lang 项目，一个是分配，一个是回收。一个使用 `Box` 的独立程序可以通过 `malloc` 和 `free` 来动态的管理内存分配。

```
#![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` lang 项目可以被认为是返回一个有效的指针，且需要内部的检查。

其他的由 lang 项目提供的功能包括：

- 基于特征的可重载操作符：操作符 ==, <, * 和 + 都被 lang 项目标记起来。这些特殊的操作符对应于 `eq` , `ord` , `deref` 和 `add` .
- 展开堆栈和一般性错误。 `eh_personality` , `fail` 和 `fail_bounds_checks` lang项目。
- `std::marker` 中的特点用于表示不同类型，lang 项目 `send` , `sync` 和 `copy` 。
- `std::marker` 中的标记类型和变量指示符，lang 项目有 covariant_type, contravariant_lifetime

lang 项目会被编译器以消极的方式加载。如果用户不使用 Box ，那么就不需要为 `exchange_malloc` 和 `exchange_free` 定义函数。当以个lang 项目在当前代码中存在，那么 `rustc` 就会发出错误提示。

链接参数

有一种可以定制化 rust 程序的方法，那就是 `link_args` 属性。这个属性可以附加代码块然后定制化需要通过链接器的行标志。

```
#![feature(link_args)]

#[link_args = "-foo -bar -baz"]
extern {}
```

需要注意的是实现的功能在 `feature(link_args)` 之后，因为这并不是一种被认可的执行链接的方法。目前，rust 不会再使用系统链接器，所以提供附加的命令行参数将是非常有意义的，但是往往会事与愿违。未来，rust 会直接使用 LLVM 来链接库，那时 `link_args` 将毫无意义。

最好还是不要使用这个属性，在附加代码块中使用更正规的 `#[link(...)]` 将是一个不错的选择。

基准测试

Rust支持基准测试来测试用户代码的性能。我们来看一下 `src/lib.rs` 的性能如何。

```
`#![feature(test)]

extern crate test;

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

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

注意上面代码中的 `test` 的功能声明，这表示这并不是一个稳定的功能。

用户需要引入 `test` 的封装，来使得基准测试得以支持。通过 `bench` 属性的使用，我们可以实现一个新的函数。与不能有参数的常规测试不同，这里的基准测试使用 `&mut Bencher` 来改善这个情况。这里的 `Bencher` 提供了一个闭包的 iter 方法。这个闭包就包含了需要进行基准测试的代码。

用户通过 `cargo bench` 指令来执行基准测试：

```
$ 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
```

我们的非基准测试可以被忽略掉。用户也许会注意到 `cargo bench` 比 `cargo test` 长一个比特。这是因为 Rust 会多次执行基准测试，然后会取一个平均值。因为在这个例子中，只是实现了一些简单的功能，所以我们有一个 `ns/iter (+/- 0)`，但是这会展示出结果的方差。

如下是编写基准测试的建议：

- 将安装代码移到 `iter` 循环之外，只是将用户希望进行测试的代码放到 `iter` 循环内。
- 将实现同一功能的代码放到迭代体内，不要将结果进行累计，也不要改变状态。
- 将外部函数进行幂等化，基准测试程序会将它测试多次。
- 尽量使 `iter` 循环体内容的代码更精简，然后使得基准测试运行起来更快捷，使得校准器可以以更高精度的来校准。
- 使 `iter` 循环体内的代码更简单，以帮助查明性能改进的地方（或回归）。

疑难杂症：优化

在基准测试程序的编写方面还存在一些棘手的问题：优化编译后的基准测试代码可能会被优化器篡改，这样使得基准测试可能就不再是用户希望的测试对象了。比如，编译器可能会重新组织一些代码，因为这些代码并没有什么作用，甚至会将它全部删除。

```
#![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);
});
}
```

上述代码会出现下述结果：

```
running 1 test
test bench_xor_1000_ints ... bench: 0 ns/iter (+/- 0)

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

为了避免上述问题，基准测试的执行者会采用两个方法。`iter` 方法可以获得一个闭包，它可以返回一个任意值来迫使优化器考虑这个结果，来保证优化器不会更改基准测试代码。下面的例子就是来展示如何校准 `b.iter`.

```
`b.iter(|| {
// note lack of `;` (could also use an explicit `return`).
(0..1000).fold(0, |old, new| old ^ new)
});`
```

另一种方法是通用的 `test::black_box` 函数，它对优化器就是一个“黑盒”，可以迫使优化器考虑更多的参数。

```
#![feature(test)]

extern crate test;

b.iter(|| {
let n = test::black_box(1000);

(0..n).fold(0, |a, b| a ^ b)
})
```

这些并不会读取和更改这个值。较大的值可以直接降低开销。

```
`black_box(&huge_struct)`
```

执行上述任何一种变化提供了以下基准测试结果。

```
running 1 test
test bench_xor_1000_ints ... bench:   131 ns/iter (+/- 3)

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

即使使用上述方式，优化器仍然会以不可想象的方式来修改测试用例。

盒语法和模式

目前，唯一稳定可靠地方法就是通过 `Box::new` 方法来创建 `Box` 。当然，它不可能在稳定的 Rust 来析构匹配模式下的 `Box` 。

不稳定的 `box` 关键字可以用来创建和析构 `Box` 。相关的例子如下：

```
#![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!()
}
}
```

注意这个功能目前隐藏在 `box_syntax` (盒创建方法) 和 `box_patterns` （析构和匹配模型）方法，因为这个语法在未来仍可能会被更改。

返回指针

在很多计算机语言中都有指针，用户可以通过返回一个指针来避免返回较大数据结构的拷贝。比如：

```
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);
}
```

这里面的想法就是通过返回一个盒，用户可以仅仅拷贝一个指针，从而避免 拷贝 `BigStruct` 中的上百个 `int` 数。

如下为 Rust 的反模式，相反，可以编写成下面的方式：

```
#![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);
}
```

这个方法是一种不牺牲性能的前提下提供了灵活性。

用户可能会认为这会表现出较差的性能：返回一个值，然后立即用盒装起来？这是最糟糕的模式么？Rust 远远比这些更智能。这并不是将代码进行拷贝。 `main` 为 `box` 分配足够的空间，然后传递一个指针 x 来指向 `fo o` 。然后， `foo` 将值写回 `Box<T>` 。

下面这一点很重要：指针不仅可以优化代码块中返回的值。也允许调用者来选择他们希望他们如何使用他们的输出。

切片模式

如果用户希望匹配一个切片或数组，用户可以使用 `&` 来修饰 `slice_patterns` 功能。

```
#![feature(slice_patterns)]

fn main() {
let v = vec!["match_this", "1"];

match &v[..] {
["match_this", second] => println!("The second element is {}", second),
_ => {},
}
}
```

`advanced_slice_pattern` 使用户可以使用 `..` 来表示切片匹配模式内部的元素任何数目。此通配符仅能为给定的数组使用一次。如果在 `..` 前有个标识符，切片的结果将会绑定到这个名字。比如:

```
`#![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));
} `
```

相关常量

使用 associated_consts 功能，用户可以采用如下方式来定义常量：

```
#![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 的实现都必须定义 ID . 如果没有定义：

```
#![feature(associated_consts)]

trait Foo {
const ID: i32;
}

impl Foo for i32 {
}
```

上述代码出现下面的提示：

error: not all trait items implemented, missing: ID [E0046] impl Foo for i32 { }

默认的值可以采用如下实现：

```
#![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);
}
```

正如用户看到那样，当实现 `Foo` 的时候，用户可以不实现 `i32`，它就会使用默认值。至于 `i64`，我们也可以增加我们自己的定义。

相关常量并不是必须与一个特性相关。`struct` 的 `impl` 块也可以很好的工作，如下：

```
#![feature(associated_consts)]

struct Foo;

impl Foo {
pub const FOO: u32 = 3;
}
```

7

词汇表

并不是每个 Rust 中的概念都有系统编程或者计算机科学中的相关背景， 所以我们增加一些可能是陌生术语的解释。

元数

元数是指函数或操作需要的参数个数。

```
let x = (2, 3); let y = (4, 6); let z = (8, 2, 6);
```

在上述例子中，`x` 和 `y` 的元数为2。 `z` 的元数是3。

抽象语法树

当编译器编译程序的时候，它会一下事情。其中一个就是将程序中的代码段组成抽象语法树，简称“AST”。这个抽象语法树就表示了程序的结构。比如，`2+3` 可以表示为如下形式：

```
  +
 /\
 2  3
```

此外，2+（3*4）可以表示为如下：

```
 +
/\
2  *
 /\
 3  4
```

8

相关学术研究

如下是部分对 Rust 有影响的学术论文。通过阅读它们可以更好的理解 Rust 的背景，甚至迸发出更精彩的想法。

类型系统

- Region based memory management in Cyclone
- Safe manual memory management in Cyclone
- Typeclasses: making ad-hoc polymorphism less ad hoc
- Macros that work together
- Traits: composable units of behavior
- Alias burying 我们试过类似的并且会将之摒弃。
- External uniqueness is unique enough
- Uniqueness and Reference Immutability for Safe Parallelism
- Region Based Memory Management

并发性

- Singularity: rethinking the software stack
- Language support for fast and reliable message passing in singularity OS
- Scheduling multithreaded computations by work stealing
- Thread scheduling for multiprogramming multiprocessors
- The data locality of work stealing
- Dynamic circular work stealing deque 循环双向队列
- Work-first and help-first scheduling policies for async-finish task parallelism 比非严格 work-steal 策略更通用。
- A Java fork/join calamity 针对 Java 中的 fork/join 库的批判, 尤其是 JAVA 中的 work-steal 策略到非严格计算的应用。
- Scheduling techniques for concurrent systems
- Contention aware scheduling
- Balanced work stealing for time-sharing multicores
- Three layer cake
- Non-blocking steal-half work queues
- Reagents: expressing and composing fine-grained concurrency
- Algorithms for scalable synchronization of shared-memory multiprocessors

其他

Crash-only software

Composing High-Performance Memory Allocators

Reconsidering Custom Memory Allocation

关于 Rust 的论文

GPU programming in Rust

Parallel closures: a new twist on an old idea – 提到的不完全是 rust，但是是由 nmatsakis 提出的。