

目錄

1. [介紹](#)
2. [1.介绍](#)
3. [2.基础](#)
 - i. [2.1.安装Rust](#)
 - ii. [2.2.Hello, world!](#)
 - iii. [2.3.Hello, Cargo!](#)
 - iv. [2.4.变量绑定](#)
 - v. [2.5.If语句](#)
 - vi. [2.6.函数](#)
 - vii. [2.7.注释](#)
 - viii. [2.8.复合数据类型](#)
 - ix. [2.9.匹配](#)
 - x. [2.10.循环](#)
 - xi. [2.11.字符串](#)
 - xii. [2.12.数组，向量和片段](#)
 - xiii. [2.13.标准输入](#)
 - xiv. [2.14.猜猜看](#)
4. [3.中级](#)
 - i. [3.1.包装箱和模块](#)
 - ii. [3.2.测试](#)
 - iii. [3.3.指针](#)
 - iv. [3.4.所有权](#)
 - v. [3.5.更多字符串](#)
 - vi. [3.6.模式](#)
 - vii. [3.7.方法语法](#)
 - viii. [3.8.关联类型](#)
 - ix. [3.9.闭包](#)
 - x. [3.10.迭代器](#)
 - xi. [3.11.泛型](#)
 - xii. [3.12.特性](#)
 - xiii. [3.13.静态和动态分发](#)
 - xiv. [3.14.宏](#)
 - xv. [3.15.并发](#)
 - xvi. [3.16.错误处理](#)
 - xvii. [3.17.文档](#)
5. [4.进阶](#)
 - i. [4.1.外部语言接口](#)
 - ii. [4.2.不安全代码](#)
 - iii. [4.3.宏进阶](#)
6. [5.不稳定功能](#)
 - i. [5.1.编译器插件](#)
 - ii. [5.2.内联汇编](#)

- iii. 5.3.不使用标准库
 - iv. 5.4.固有功能
 - v. 5.5.语言项
 - vi. 5.6.链接参数
 - vii. 5.7.基准测试
 - viii. 5.8.装箱语法和模式
- 7. 6.总结
 - 8. 7.词汇表

前言

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

Rust编程语言

欢迎阅读！这本书将教会你使用Rust编程语言。Rust是一个注重安全与速度的现代系统编程语言，通过在没有垃圾回收的情况下保证内存安全来实现它的目标。

本书分为三个章节，简介部分如下：

基础

这一部分顺序介绍了Rust的基本语法和语义。每一小节都介绍了一部分Rust语法，在本章的最后会介绍一个小小的Rust项目：一个猜数字的游戏。

在阅读完“基础”部分之后，你将会有个良好的基础并能够编写非常简单的Rust程序。

中级

这一部分包含一系列独立而完整的章节，分别关注一些特定的主题。你可以按任意顺序阅读这一部分。

在阅读完“中级”部分后，你将会对Rust有一个充分的了解，你将能阅读大部分Rust代码并能编写更加复杂的程序。

进阶

这是一个完全独立并且很有深度的部分，作为“中级”部分的补充，你可以按任意顺序阅读。这一部分的章节关注一些最复杂的功能以及一些只会出现在未来版本Rust中的特性。

阅读完“进阶”之后，你将成为一名Rust专家！

不稳定功能

这也是一个完全独立并且很有深度的部分，作为“中级”部分的补充，你可以按任意顺序阅读。

这一部分包含只在Rust每日构建中出现的功能。（注：可能经常变动）

基础

这一部分顺序介绍了Rust的基本语法和语义。每一小节都介绍了一部分Rust语法，在本章的最后会介绍一个小小的Rust项目：一个猜数字的游戏。

在阅读完“基础”部分之后，你将会有个良好的基础并能够编写非常简单的Rust程序。

安装Rust

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

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

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

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

如果你用Windows，请直接下载[32位](#)或者[64位](#)安装包然后运行即可。

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

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

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

你可以在任何时候重新运行脚本来更新Rust。在现在这个时间，你将会频繁更新Rust，因为Rust还未发布1.0版本，经常更新人们会认为你在使用最新版本的Rust。

不过这带来了另外一个问题（传说中的免责声明？）：一些同学确实有理由对我们让他们运行`curl | sudo sh`感到非常反感。他们理应如此。从根本上说，当你运行上面的脚本时，代表你相信是一些好人在维护Rust，他们不会黑了你的电脑做坏事。对此保持警觉是一个很好的天性。如果你是这些强迫症患者（大雾），请检阅以下文档，[从源码编译Rust](#)或者[官方二进制文件下载](#)。我们保证这将不会一直作为安装Rust的方法：这只是为了方便大家在Alpha时期更新Rust。

当然，我们需要提到官方支持的平台：

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

Rust在以上平台进行了广泛的测试，当然还有一些其他平台，比如Android。不过进行了越多测试的环境，越有可能正常工作。

最后，关于Windows。Rust将Windows作为第一级平台来发布，不过说实话，Windows的集成体验并没有Linux/OS X那么好。我们正在改善它！如果有情况它不能工作了，这是一个bug。如果这种发生了，请让我知道。任何一次提交都在Windows下进行了测试，就像其它平台一样。

如果你已安装Rust，你可以打开一个Shell，然后输入：

```
$ rustc --version
```

你应该看到版本号，提交的hash值，提交时间和构建时间：

```
rustc 1.0.0-nightly (f11f3e7ba 2015-01-04) (built 2015-01-06)
```

如果你做到了，那么Rust已经正确安装！此处应有掌声！

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

Hello, world!

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

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

这里确实有另一个导致我担心的地方：这个教程假定你熟悉基本的命令行操作。Rust并不要求你非常了解命令行，但是直到这个语言到达一个更加完善的地步，IDE支持将是十分差劲的。Rust并不对你使用的编辑工具和代码位置有特定的要求。

既然这么说了，让我们在projects目录下新建一个目录

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

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

接下来让我们新建一个源文件。在本例中我使用 editor filename 这种语法来代表编辑一个文件，不过你应该使用自己的编辑器。我们叫源文件 main.rs：

```
$ editor main.rs
```

Rust源文件总是使用 .rs 后缀。如果你使用了多个词，使用下划线分隔。写成 hello_world.rs 而不是 helloworld.rs。

现在打开你的源文件，键入如下代码：

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

保存文件，然后在终端中输入如下命令：

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

你也可以在play.rust-lang.org中运行这个例子。你可以将鼠标放到代码上方，然后点击出现在右上角的箭头运行代码（明显这里文档已经过时，上面那个网址已经改版，点击evaluate按钮运行脚本，同时还可以

查看汇编和LLVM IR等)。

搞定这些，让我们回过头来看看到底发生了什么

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

这几行定义了一个Rust函数。`main` 函数是特殊的：这是所有Rust程序的开始。第一行表示“我定义了一个叫`main`的函数，没有参数也没有返回值。”如果有参数的话，它们应该出现在括号（(和)）中。因为我们并没有返回任何东西，我们可以完全省略返回类型。我们稍后将谈论这些。

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

接下来是这一行：

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

这一行做了我们这小程序的所有工作。这里有一些细节比较重要。第一，缩进是4个空格，而不是制表符（Tab）。请设置你的编辑器Tab键为插入4个空格。我们提供[多种编辑器的配置例子](#)（已404勿念）。

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

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

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

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

```
$ rustc main.rs
```

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

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

或者在Windows下：

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

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

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

这将在终端上打印出 Hello, world! 。

对于来自像Ruby，Python或者Javascript这样的动态类型语言的同学，可能不太习惯这样将编译和执行分开的行为。Rust是一门预先编译语言 (*ahead-of-time compiled language*)，这意味着你可以编译一个程序，把它给任何人，他们都不需要安装Rust就可运行。如果你给他们一个 .rb , .py 或 .js 文件，他们需要先安装Ruby/Python/Javascript，不过你只需要一句命令就可以编译和执行你的程序。这一切都是语言设计的权衡取舍，而Rust已经做出了它的选择。

祝贺你！你已经正式完成了一个Rust程序。你现在是一名Rust程序猿了！欢迎入坑！

接下来，我将向你介绍另外一个工具，Cargo，它用来编写真实世界的Rust程序。仅仅使用 rustc 可以很好应对简单的程序，但是当你的项目不断增长，你将会一个工具帮助你管理所有可能的选项，帮助你更加轻松的与别人和别的项目分享代码。

Hello, Cargo!

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

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

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

转换到Cargo

让我们将Hello Wold项目转换到Cargo。

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

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

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

接下来，我们的配置文件

```
$ editor Cargo.toml
```

请确保文件名正确：你需要一个大写的C！

在配置文件中添加：

```
[package]

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

[[bin]]

name = "hello_world"
```

配置文件使用[TOML](#)格式。让我们向你解释一下：

TOML旨在作为一个最小化的配置文件格式，明确的语义，易于阅读。TOML被设计成可以无二义地映射到哈希表中。TOML应该能解析成许多类型语言的数据类型。TOML非常像INI文件，不过有一些其它优点。

总之，这个文件中有两张表：`package` 和 `bin`。第一个告诉Cargo你项目的元信息。第二个告诉Cargo你希望构建一个二进制（可执行）文件，不是一个库文件（虽然我们可以两个都是！），就像它的名字一样。

一旦你设置好了配置文件，我们应该可以构建了！试试这些：

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

OK！我们运行 `cargo build` 构建了项目并运行 `./target/hello_world` 执行了它。这并没有比简单的运行 `rustc` 强多少，不过我们可以想象一下：当我们的项目不止一个文件，我们将运行 `rustc` 不止一次，还得传递一堆参数告诉它将所有东西构建到一起。有了Cargo，随着项目的增长，我们可以仅仅运行 `cargo build`，而一切将正常运行。

你还需要注意到Cargo创建了一个新文件：`Cargo.lock`。

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

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

好了！我们成功利用Cargo构建了 `hello_world`。虽然我们的项目很简单，但它用上许多实际的工具，你将会在余下的Rust生涯中一直使用。

一个新项目

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

用Cargo开始一个新项目，运行 `cargo new`：

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

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

让我们看看Cargo为我们生成了什么：

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

1 directory, 2 files
```

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

这就是开始时所需的全部。首先，让我们查看下 `Cargo.toml`：

```
[package]

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

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

下面是 `src/main.rs` 的内容：

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

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

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

变量绑定

你将学习的一个要点是变量绑定。它们看起来像这样：

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

在每个例子中都写上 `fn main()` { 有点冗长，所以之后我们将省略它。如果你是一路看过来的，确保你写了 `main()` 函数，而不是省略不写。否则，你将得到一个错误。

在许多语言中，这叫做变量。不过Rust的变量绑定有自己的玄机。Rust有一个非常强大的功能叫做模式匹配，关于它我们之后再详说，不过 `let` 表达式的左侧是一个完整的模式，而不仅仅是一个变量。这意味着我们可以这样写：

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

在这个表达式被计算后，`x` 将会是1，而 `y` 将会是2.模式非常强大，不过这是我们目前可以做到的全部。所以接下来你只需记住有这个东西就行了。

Rust是一个静态类型语言，这意味着我们需要先确定我们需要的类型。那为什么我们第一例子能编译过呢？嘛，Rust有一个叫做类型推断的功能。如果它能确认这是什么类型，Rust不需要你非得把它写出来。

若你愿意，我们也可以加上类型。类型写在一个冒号 (:) 后面：

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

如果我叫你对着全班同学大声读出这一行，你应该大喊“`x`被绑定为*i32*类型，它的值是5”。

在这个例子中我们选择 `x` 代表一个32位的有符号整数。Rust有许多不同的原生整数类型。以 `i` 开头的代表有符号整数而 `u` 开头的代表无符号整数。可能的整数大小是8，16，32和64位。

在之后的例子中，我们可能会在注释中注明变量类型。例子看起来像这样：

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

注意注释和 `let` 表达式有类似的语法。理想的Rust代码中不应包含这类注释。不过我们偶尔会这么做来帮助你理解Rust推断的是什么类型。

绑定默认是不可变的 (*immutable*)。下面的代码将不能编译：

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

它会给你如下错误：

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

如果你想一个绑定是可变的，使用 `mut`：

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

不止一个理由使得绑定默认不可变的，不过我们可以理解它通过一个Rust的主要目标：安全。如果你没有使用 `mut`，编译器会捕获它，让你知道你改变了一个你可能并不打算让它改变的值。如果绑定默认是可变的，编译器将不可能告诉你这些。如果你确实想变量可变，解决办法也非常简单：加个 `mut`。

尽量避免可变状态有一些其它好处，不过这不在这个教程的讨论范围内。大体上，你总是可以避免显式可变量，并且这也是Rust倾向你做的。即便如此，有时，可变量是你需要的，所以这并不是被禁止的。

让我们回到绑定。Rust变量绑定有另一个不同于其它语言的方面：绑定要求在可以使用它之前必须初始化。

让我们尝试一下。将你的 `src/main.rs` 修改为为如下：

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

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

你可以用 `cargo build` 命令去构建它。它依然会输出“Hello, world!”，不过你会得到一个警告：

```
Compiling hello_world v0.0.1 (file:///home/you/projects/hello_world)
src/main.rs:2:9: 2:10 warning: unused variable: `x`, #[warn(unused_variable)] on by default
src/main.rs:2      let x: i32;
                  ^
```

Rust警告我们从未使用过这个变量绑定，但是因为我们从未用过它，无害不罚。然而，如果你确实想使用 `x`，事情就不一样了。让我们试一下。修改代码如下：

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

```

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

```

然后尝试构建它。你会得到一个错误：

```

$ cargo build
   Compiling hello_world v0.0.1 (file:///home/you/projects/hello_world)
src/main.rs:4:39: 4:40 error: use of possibly uninitialized variable: `x`
src/main.rs:4      println!("The value of x is: {}", x);
                           ^
note: in expansion of format_args!
<std macros>:2:23: 2:77 note: expansion site
<std macros>:1:1: 3:2 note: in expansion of println!
src/main.rs:4:5: 4:42 note: expansion site
error: aborting due to previous error
Could not compile `hello_world`.

```

Rust是不会让我们使用一个没有经过初始化的值的。接下来，让我们讨论一下我们添加到 `println!` 中的内容。

如果你输出的字符串中包含一对大括号（`{}`，一些人称之为。。胡须（moustaches）？），Rust将把它解释为插入值的请求。字符串插值（*String interpolation*）是一个计算机科学术语，代表“在字符串中插入值”。我们加上一个逗号，然后是一个 `x`，来表示我们想插入 `x` 的值。逗号用来分隔我们传递给函数和宏的参数，如果你想传递多个参数的话。

当你只写了大括号的时候，Rust会尝试检查值的类型来显示一个有意义的值。如果你想指定详细的语法，这里有[很多选项可供选择](#)。现在，让我们保持默认，整数并不难打印。

If语句

Rust的If并不是特别复杂，不过你会发现它更像动态类型语言而不是更传统的系统语言。所以让我来说说它，以便你能把握这些细节。

If语句是分支这个更加宽泛的概念的一个特定形式。它的名字来源于树的树枝：一个选择点，根据选择的不同，将会使用不同的路径。

在If语句中，这里有一个选择导致了两个路径：

```
let x = 5;

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

如果在什么别的地方更改了x的值（你确定不加mut可以吗。。。），这一行将不会输出。更具体一点，如果 `if` 后面的表达式的值为 `true`，这个代码块将被执行。为 `false` 则不被执行。

如果你想什么在值为 `false` 是发生，使用 `else`：

```
let x = 5;

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

如果不止一种情况，使用 `else if`：

```
let x = 5;

if x == 5 {
    println!("x is five!");
} else if x == 6 {
    println!("x is six!");
} else {
    println!("x is not five or six :(");
}
```

这些都是非常标准的情况。然而你也可以这么写：

```
let x = 5;

let y = if x == 5 {
    10
} else {
```

```
    15
}; // y: i32
```

你可以也应该这么写：

```
let x = 5;

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

这展示了亮点Rust有趣的地方：他是一门基于表达式的语言，并且这里分号与其它基于“大括号和分号”的语言的分号不同。这两点是相互联系的。

表达式（Expressions） vs 语句（Statements）

Rust是一门主要基于表达式的语言。只有两种语句，然后其它的一切都是表达式。

这又有什么区别呢？表达式返回一个值，而声明不返回。在许多语言中 `if` 语句是语句，也就是说，`let x = if ...` 没有意义。不过在Rust这中，`if` 是一个表达式，也就是说它返回一个值。我们可以用这个值来初始化绑定。

说到底，绑定是两种Rust语句中的一种。正式的名称为声明语句（*declaration statement*）。到目前为止，`let` 是我们见过唯一一种声明语句。让我们接着聊聊这些。

在一些语言中，变量绑定可以被写成表达式，而不只是语句。比如Ruby：

```
x = y = 5
```

然后在Rust中，`let` 声明的绑定不是一个表达式。下面的代码将会产生一个编译时错误：

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

编译器告诉我们它希望看到表达式的开始，而 `let` 只能是一个语句的开头，而不是表达式。

注意为一个绑定过的变量赋值仍是一个表达式，虽然它的返回值不是特别有用。这不像C语言，赋值语句的值是被赋予的值，而Rust中赋值语句的值是一个单元类型（）（这个我们稍后介绍）。

Rust中第二类语句是表达式语句（*expression statement*）。它的目的是将任何表达式变成一个语句。在实际的情况下，Rust语法期望语句后面跟着另一个语句。这意味着你需要用分号分隔每个表达式。这也意味着Rust向大部分其它语言一样在每一行的末尾加一个分号，你也会在大部分Rust代码行尾看到一个分号。

那么除了那些“大部分”代码呢？你已经见过它们了，在如下代码中：

```
let x = 5;

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

注意到我注明了 `y` 的类型，来明确我需要一个整型的 `y`。

这跟下面的代码不一样，下面的代码将不能编译：

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

注意 `10` 和 `15` 后面的分号。Rust给我们如下错误：

```
error: mismatched types: expected `i32`, found `()` (expected i32, found ())
```

我们需要一个整形却得到一个`()`。`()`是一个单元 (*unit*)，这是Rust类型系统中一个特殊的类型。在Rust中，`()`不是一个有效的 `i32` 类型的值。它只能是`()`类型变量的有效值，但这并不是非常有用。还记得我们说语句并不返回值吗？这正是单元的作用。分号将表达式变为语句并且将它的值变为一个单元。

还有一种情况你不会在一行Rust代码的结尾看到分号。这下一章我们要介绍的：函数。

函数

到目前为止你应该见过一个函数，`main` 函数：

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

这可能是最简单的函数声明。就像我们之前提到的，`fn` 表示“这是一个函数”，后面跟着名字，一对括号因为这函数没有参数，然后是一对大括号代表函数体。下面是一个叫 `foo` 的函数：

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

那么有参数是什么样的呢？下面这个函数打印一个数字：

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

下面是一个使用了 `print_number` 函数的完整的程序：

```
fn main() {  
    print_number(5);  
}  
  
fn print_number(x: i32) {  
    println!("x is: {}", x);  
}
```

如你所见，函数参数与 `let` 声明非常相似：参数名加上冒号再加上参数类型。

下面是一个完整的程序，它将两个数相加并打印结果：

```
fn main() {  
    print_sum(5, 6);  
}  
  
fn print_sum(x: i32, y: i32) {  
    println!("sum is: {}", x + y);  
}
```

在调用函数和声明函数时，你需要用逗号分隔多个参数。

与 `let` 不同，你必须为函数参数声明类型。下面代码将不能工作：

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

你会获得如下错误：

```
hello.rs:5:18: 5:19 expected one of `!`, `:`, or `@`, found `)`
hello.rs:5 fn print_number(x, y) {
```

这是一个有意为之的设计决定。即使像Haskell这样的能够全程序推断的语言，也经常建议注明类型是一个最佳实践。我们同意即使允许在函数体中推断也要强制函数声明参数类型是一个全推断与无推断的最佳平衡。

如果我们要一个返回值呢？下面这个函数给一个整数加一：

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

Rust函数确实返回一个值，你需要在一个箭头（`->`）定义返回值类型。

注意这里并没有一个分号。如果你把它加上：

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

你将会得到一个错误：

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

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

还记得我们之前关于分号和`()`的讨论吗？我们声明函数要返回一个`i32`，不过因为分号，它会返回一个`()`。Rust发现这可能不是你想要的，并建议你去掉这个分号。

这非常像我们之前的那个`if`语句：`{}`的结果是表达式的值。像其它面向表达式的语言如Ruby也是如此，不过这在系统编程世界并不常见。当人们第一次学到这的时候，通常会认为这会产生bug。不过因为Rust的类型系统是如此强大，以为单元是一个特殊类型，我们还未发现在返回值时加减分号会导致bug的现象。

不过如果我们想提前返回呢？Rust确实有一个关键字，`return`：

```
fn foo(x: i32) -> i32 {
    if x < 5 { return x; }

    x + 1
}
```

在最后一行使用 `return` 也是可以的，不过被认为是一个拙计的设计：

```
fn foo(x: i32) -> i32 {
    if x < 5 { return x; }

    return x + 1;
}
```

对于之前没有接触过面向表达式语言的人来说那个没有 `return` 的定义可能看起来有点别扭，不过随着时间的推移，他会变得更直观。如果我们在写生产代码，我们不会向上面那样写的，我们会这么写：

```
fn foo(x: i32) -> i32 {
    if x < 5 {
        x
    } else {
        x + 1
    }
}
```

因为 `if` 是这个函数唯一的表达式，它的值就是函数的结果。

发散函数 (Diverging functions)

Rust有些特殊的语法叫“发散函数”，这些函数并不返回：

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

`panic!` 是一个宏，类似我们已经见过的 `println!()`。与 `println!()` 不同的是，`panic!()` 导致当前的执行线程崩溃并返回指定的信息。

因为这个函数会崩溃，所以它不会返回，所以它拥有一个类型 `!`，它代表“发散”。一个发散函数可以是任何类型：

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

我们并没有很好的利用发散函数。因为它结合在别的Rust功能中。不过当你再看到 `-> !` 时，你要知道它是发散函数。

注释

现在我们写了一些函数，所以我们应该学习一下注释。注释是你帮助其他程序猿理解你的代码的备注。编译器基本上会忽略它们。

Rust有两种需要你了解的注释格式：行注释 (*line comments*) 和文档注释 (*doc comments*) 。

```
// Line comments are anything after '//' and extend to the end of the line.

let x = 5; // this is also a line comment.

// If you have a long explanation for something, you can put line comments next
// to each other. Put a space between the // and your comment so that it's
// more readable.
```

另一种注释是文档注释。文档注释使用 `///` 而不是 `//`，并且支持Markdown标记：

```
/// `hello` is a function that prints a greeting that is personalized based on
/// the name given.
///
/// # Arguments
///
/// * `name` - The name of the person you'd like to greet.
///
/// # Example
///
/// ```rust
/// let name = "Steve";
/// hello(name); // prints "Hello, Steve!"
/// ```

fn hello(name: &str) { println!("Hello, {}!", name); } ``
```

当书写文档注释时，加上参数和返回值部分并提供一些用例将是非常有帮助。不要担心那个 `&str`，我们马上会提到它。

你可以使用[rustdoc](#)工具来讲文档注释生成为HTML文档。

复合数据类型

像许多其它语言一样，Rust有很多不同的内建数据类型。我们已经处理过简单的整形和字符串，不过接下来，让我们了解一些复杂的储存数据的方法。

元组 (Tuples)

我们讨论的第一个复合数据类型是元组 (*tuples*)。元组是固定大小的有序列表。如下：

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

这是一个长度为2的元组，有括号和逗号组成。下面也是同样的元组，不过注明了数据类型：

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

如你所见，元组的类型跟元组看起来很像，只不过类型取代的值的位置。细心的读者可能会注意到元组是异质的：这个元组中有一个 `i32` 和一个 `&str`。你不久前应该见过 `&str` 作为一个类型，我们稍后叫讨论字符串的详细内容。在系统编程语言中，字符串要比其它语言中来的复杂。现在，可以认为 `&str` 是一个字符串片段 (*string slice*)，我们马上会讲到它。

你可以通过一个解构`let` (*destructuring let*) 访问元组中的字段。下面是一个例子：

```
let (x, y, z) = (1, 2, 3);
println!("x is {}", x);
```

还记得我曾经说过 `let` 语句的左侧不仅仅是一个绑定吗？这就是证据。我们可以在 `let` 左侧写一个模式，如果它能匹配右侧的话，我们可以一次写多个绑定。这种情况下，`let` “解构”或“拆开”了元组，并分成了三个绑定。

这是一个非常强力的模式，我们后面会经常看到它。

不解构元组，我们也可以做很多事。你一个把一个元组赋值给另一个，如果它们含有相同的类型和数量。当它们有相同数量时，它们也有一样的长度。

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

x = y;
```

你也可以用 `==` 检查它们是否相等。同样，被比较的元组必须有相同的类型代码才能编译。

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

```
let y = (2, 2, 4);

if x == y {
    println!("yes");
} else {
    println!("no");
}
```

这回打印 no , 因为有些值并不相等。

注意检查相等时值的顺序将被考虑，所以下面的例子也会打印 no 。

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

if x == y {
    println!("yes");
} else {
    println!("no");
}
```

元组的另一个作用是你可以让函数返回多个值：

```
fn next_two(x: i32) -> (i32, i32) { (x + 1, x + 2) }

fn main() {
    let (x, y) = next_two(5);
    println!("x, y = {}, {}", x, y);
}
```

即使Rust函数只能返回一个值，元组确实是一个值，它只是碰巧由多个值组成。在这个例子中，你可以看到我们是如何拆开函数返回值的。

元组是一个非常简单的数据结构，而通常也不是你需要的。让我们见识一下它的大哥，结构体。

结构体 (Structs)

就想元组一样，结构是另一种形式的记录类型 (*record type*) 。不过这有个区别：结构体的每个元素都有一个名字，元素叫做字段或成员。看看下面的代码：

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let origin = Point { x: 0, y: 0 }; // origin: Point

    println!("The origin is at ({}, {})", origin.x, origin.y);
}
```

这里有许多细节，让我们分开说。我们使用了 `struct` 关键字后跟名字来定义了一个结构体。根据传统，结构体使用大写字母开头并且使用驼峰命名法：`PointInSpace` 而不要写成 `Point_In_Space`。

想往常一样我们用 `let` 创建了一个结构体的实例，不过我们用 `key: value` 语法设置了每个字段。这里顺序不必和声明的时候一致。

最后，因为每个字段都有名字，我们可以访问字段通过圆点记法：`origin.x`。

结构体中的值默认是不可变的，就像Rust中其它的绑定一样。使用 `mut` 使其可变：

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let mut point = Point { x: 0, y: 0 };

    point.x = 5;

    println!("The point is at ({}, {})", point.x, point.y);
}
```

上面的代码会打印 `The point is at (5, 0)`。

元组结构体和新类型（Tuple Structs and Newtypes）

Rust有一个叫做元组结构体 (*tuple struct*) 的类型，它就像一个元组和结构体的混合体。元组结构体确实有一个名字，不过它的字段没有：

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

这两个是不会相等的，即使它们有一模一样的值：

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

使用结构体几乎总是好于使用元组结构体。我们可以这样重写 `Color` 和 `Point`：

```
struct Color {
    red: i32,
    blue: i32,
    green: i32,
}

struct Point {
```

```
x: i32,
y: i32,
z: i32,
}
```

现在，我们有了名字，而不是位置。好的名字是很重要的，使用结构体，我们就可以设置名字。

不过有种情况元组结构体非常有用，就是当元组结构体只有一个元素时。我们管它叫新类型 (*newtype*)，因为你创建了一个与元素相似的类型：

```
struct Inches(i32);

let length = Inches(10);

let Inches(integer_length) = length;
println!("length is {} inches", integer_length);
```

如你所见，你可以通过一个解构 `let` 来提取内部的整形，就像我们在讲元组时说的那样，`let Inches(integer_length)` 给 `integer_length` 赋值为 `10`。

枚举

最后，Rust有一个“集合类型”，一个枚举。枚举是Rust一个极为有用的功能，他被用到了整个标准库中。枚举是一个类型，它把一系列互不相交的值绑到一起。举个例子来说，下面我们定义了一个 `Character`，它的值只能是 `Digit` 或其它。你可以通过全名来使用它：`Character::Other`（下面介绍 `::`）。

```
enum Character {
    Digit(i32),
    Other,
}
```

枚举的变量可以被定义为大部分正常类型。下面列出了一些可以在 `enum` 中使用的例子。

```
struct Empty;
struct Color(i32, i32, i32);
struct Length(i32);
struct Status { Health: i32, Mana: i32, Attack: i32, Defense: i32 }
struct HeightDatabase(Vec<i32>);
```

根据子数据类型的不同，就像结构体，`enum` 变量，可以或不可以储存数据。就像在 `Character` 中，`Digit` 绑定了一个 `i32`，而 `Other` 只是一个名字。然而，它们两个的区别是非常有用的。

就像结构体一样，枚举默认不能使用像比较（`==` 和 `!=`），二进制操作（`? * +`）和大小（`<` 和 `>=`）运算符。也就是说对于之前的 `Character` 类型，下面的代码是无效的：

```
// These assignments both succeed
let ten = Character::Digit(10);
let four = Character::Digit(4);

// Error: `*` is not implemented for type `Character`
let forty = ten * four;

// Error: `<=` is not implemented for type `Character`
let four_is_smaller = four <= ten;

// Error: `==` is not implemented for type `Character`
let four_equals_ten = four == ten;
```

这看起来可能更像限制，不过这是一个我们可以克服的限制。我们有两种方法：实现一个我们自己的比较方法，或者使用[match](#)关键字。我们还并不知道如何在Rust中实现比较方法，不过我们可以使用标准库中的[ordering](#)枚举，它是：

```
enum Ordering {
    Less,
    Equal,
    Greater,
}
```

因为我们没有定义[ordering](#)，我们必须使用[use](#)关键字（从标准库中）导入它。下面演示是如何使用[Ordering](#)的：

```
use std::cmp::Ordering;

fn cmp(a: i32, b: i32) -> Ordering {
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
    else { Ordering::Equal }
}

fn main() {
    let x = 5;
    let y = 10;

    let ordering = cmp(x, y); // ordering: Ordering

    if ordering == Ordering::Less {
        println!("less");
    } else if ordering == Ordering::Greater {
        println!("greater");
    } else if ordering == Ordering::Equal {
        println!("equal");
    }
}
```

:: 被用来表示命名空间。在这个例子中，[Ordering](#) 存在于 [std](#) 模块的 [cmp](#) 子模块中。我们会在本书的后面讨论模块。现在，你只需知道你可以 [use](#) 标准库中你想要的东西。

好，让我们看看例子中实际的代码。`cmp` 是一个函数，它比较两个数并返回一个 `Ordering`。我们能够返回 `Ordering::Less`，`Ordering::Greater` 或者 `Ordering::Equal`，根据这比较这两个值是小于，大于，或相等。注意 `enum` 的每一个变量都位于 `enum` 自己的命名空间中：是 `ordering::Greater` 而不是 `Greater`。

`ordering` 变量是 `Ordering` 类型的，所以它是上面三个值中的一个。接着我们可以用一系列的 `if/else` 去检查每个值。

`Ordering::Greater` 这个标记显得太长了。让我们用 `use` 导入 `enum` 的值。这样可以避免写完整的命名空间：

```
use std::cmp::Ordering::{self, Equal, Less, Greater};

fn cmp(a: i32, b: i32) -> Ordering {
    if a < b { Less }
    else if a > b { Greater }
    else { Equal }
}

fn main() {
    let x = 5;
    let y = 10;

    let ordering = cmp(x, y); // ordering: Ordering

    if ordering == Less { println!("less"); }
    else if ordering == Greater { println!("greater"); }
    else if ordering == Equal { println!("equal"); }
}
```

导入值是方便简洁的，不过这也可能导致命名冲突，所以使用时要慎重。因为这个原因，尽量少导入值被认为是一个好的风格。

正如你看到的，`enum` 是一个非常强力的数据表示工具，当它是泛型时则更加有用。当然，在我们讲泛型之前，让我们聊聊如何在模式匹配中，一个比一堆 `if/else` 更加高明的可以让我们解构集合类型（枚举的类型术语名称）的工具，使用枚举。

匹配

一个简单的 `if/else` 往往是不够的，因为你可能有两个或更多个选项。这样 `else` 也会变得异常复杂，所以我们该如何解决？

Rust有一个 `match` 关键字，它可以让你有效的取代复杂的 `if/else` 组。看看下面的代码：

```
let x = 5;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    4 => println!("four"),
    5 => println!("five"),
    _ => println!("something else"),
}
```

`match` 使用一个表达式然后基于它的值分支。每个分支都是 `val => expression` 这种形式。当匹配到一个分支，它的表达式将被执行。`match` 属于“模式匹配”的范畴，`match` 是它的一个实现。

那么这有什么巨大的优势呢？这确实有优势。第一，`match` 强制穷尽性检查（*exhaustiveness checking*）。你看到了最后那个下划线开头的分支了吗？如果去掉它，Rust将会给我们一个错误：

```
error: non-exhaustive patterns: `_` not covered
```

换句话说，Rust试图告诉我们，我们忘记了一个值。因为 `x` 是一个整形，Rust知道它有很多不同的值，比如，`6`。如果没有 `_` 分支，那么这就没有分支可以匹配了，Rust就会拒绝编译。`_` 就像一个匹配所有的分支。如果其它的分支都没有匹配上，就会选择 `_` 分支，就是因为有了之歌匹配所有的分支，我们现在就有了一个可以畜类 `x` 所有可能的值的分支了，这样我们的程序就能顺利编译了。

`match` 语句也会解构枚举。还记得这段来自枚举那一章的代码吗？

```
use std::cmp::Ordering;

fn cmp(a: i32, b: i32) -> Ordering {
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
    else { Ordering::Equal }
}

fn main() {
    let x = 5;
    let y = 10;

    let ordering = cmp(x, y);

    if ordering == Ordering::Less {
```

```

        println!("less");
    } else if ordering == Ordering::Greater {
        println!("greater");
    } else if ordering == Ordering::Equal {
        println!("equal");
    }
}

```

我们可以用 `match` 重写它：

```

use std::cmp::Ordering;

fn cmp(a: i32, b: i32) -> Ordering {
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
    else { Ordering::Equal }
}

fn main() {
    let x = 5;
    let y = 10;

    match cmp(x, y) {
        Ordering::Less => println!("less"),
        Ordering::Greater => println!("greater"),
        Ordering::Equal => println!("equal"),
    }
}

```

这个版本显得不那么杂乱，并且检查了穷尽性确保我们覆盖了 `Ordering` 所有可能的值。而在我们的 `if/else`，如果我们忘记写 `Greater` 分支，我们的程序也会快乐的编译通过。如果我们在 `match` 中忘了它，则不能编译通过。Rust帮助我们确保考虑到了所有情况。

`match` 表达式还允许我们获取包含在 `enum` 中的值（也叫解构），如下：

```

enum OptionalInt {
    Value(i32),
    Missing,
}

fn main() {
    let x = OptionalInt::Value(5);
    let y = OptionalInt::Missing;

    match x {
        OptionalInt::Value(n) => println!("x is {}", n),
        OptionalInt::Missing => println!("x is missing!"),
    }

    match y {
        OptionalInt::Value(n) => println!("y is {}", n),
        OptionalInt::Missing => println!("y is missing!"),
    }
}

```

这就是我们如何使用包含在 enum 中的值的。它也允许我们处理错误和异常计算；举例来说，对于一个不能保证计算出一个结果（比如一个 `i32`）的函数，我们可以返回一个 `OptionalInt`，这样我们就可以用 `match` 来处理它的值。如你所见，`enum` 和 `match` 的组合是灰常NB的！

`match` 也是一个表达式，也就是说它可以用在 `let` 绑定的右侧或者其它用到表达式的地方。我们也可以这样实现上面那个例子：

```
use std::cmp::Ordering;

fn cmp(a: i32, b: i32) -> Ordering {
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
    else { Ordering::Equal }
}

fn main() {
    let x = 5;
    let y = 10;

    println!("{}", match cmp(x, y) {
        Ordering::Less => "less",
        Ordering::Greater => "greater",
        Ordering::Equal => "equal",
    });
}
```

有时，这是一个好的模式。

循环

循环是Rust中最后一个你还没有学到的基础结构。Rust有两种主要的循环结构：`for` 和 `while`。

for循环

`for` 用来循环一个特定的次数。然而，Rust的 `for` 循环与其它系统语言有些许不同。Rust的 `for` 循环看起来并不像这个“C语言样式”的 `for` 循环：

```
for (x = 0; x < 10; x++) {
    printf( "%d\n", x );
}
```

相反，它看起来像这个样子：

```
for x in 0..10 {
    println!("{}", x); // x: i32
}
```

更抽象的形式：

```
for var in expression {
    code
}
```

这个表达式是一个迭代器，这个我们会在后面更仔细的讨论到。迭代器返回一系列的元素。每个元素是循环中的一次重复。然后它的值与 `var` 绑定，它在循环体中有效。每当循环体执行完后，我们从迭代器中取出下一个值，然后我们再重复一遍。当迭代器中不再有值时，`for` 循环结束。

在我们的例子中，`0..10` 表达式取一个开始和结束的位置，然后给出一个含有这之间值得迭代器。当然它不包括上限值，所以我们的循环会打印 `0` 到 `9`，而不是到 `10`。

Rust没有使用“C语言风格”的 `for` 循环是有意为之的。手动控制要循环的每个元素是复杂且易于出错的，甚至对于有经验的C语言选手。

我们会在后面讲解迭代器时继续了解 `for` 循环。

while循环

Rust另一个循环结构是 `while` 循环。它看起来像：

```
let mut x = 5; // mut x: u32
let mut done = false; // mut done: bool
```

```
while !done {
    x += x - 3;
    println!("{}", x);
    if x % 5 == 0 { done = true; }
}
```

`while` 循环是当你不确定应该循环多少次时正确的选择。

如果你需要一个无限循环，你可能想要这么写：

```
while true {
```

然而，Rust有一个专用的关键字 `loop` 来处理这个情况：

```
loop {
```

Rust的控制流分析会区别对待这个与 `while true`，因为我们知道它会一直循环。现阶段理解这些细节意味着什么并不是非常重要，基本上，你给编译器越多的信息，越能确保安全和生成更好的代码，所以当你打算无限循环的时候应该总是倾向于使用 `loop`。

提早结束迭代

让我们再看一眼之前的 `while` 循环：

```
let mut x = 5;
let mut done = false;

while !done {
    x += x - 3;
    println!("{}", x);
    if x % 5 == 0 { done = true; }
}
```

我们必须使用一个 `mut` 布尔型变量绑定，`done`，来确定何时我们应该推出循环。Rust有两个关键字帮助我们来修改迭代：`break` 和 `continue`。

这样，我们可以用 `break` 来写一个更好的循环：

```
let mut x = 5;

loop {
    x += x - 3;
    println!("{}", x);
    if x % 5 == 0 { break; }
}
```

现在我们用 `loop` 来无限循环，然后用 `break` 来提前退出循环。

`continue` 比较类似，不过不是退出循环，它直接进行下一次迭代。下面的例子只会打印奇数：

```
for x in 0u32..10 {  
    if x % 2 == 0 { continue; }  
  
    println!("{}", x);  
}
```

`break` 和 `continue` 在所有循环中都有效。

字符串

对于每一个程序，字符串都是需要掌握的重要内容。由于Rust注重系统，所以它的字符串处理系统与其它语言有些许区别。每当你碰到一个可变大小的数据结构时，情况都会变得很微妙，而字符串正是可变大小的数据结构。这也就是说，Rust的字符串与一些像C这样的系统编程语言也不相同。

让我们深入细节。一个字符串是一串UTF-8字节编码的Unicode量级值的序列。所有的字符串都确保是有效编码的UTF-8序列。另外，字符串并不以null结尾并且可以包含null字节。

Rust有两种主要的字符串类型：`&str` 和 `String`。

第一种是 `&str`。这叫做字符串片段 (*string slices*)。下面这个字面意思的`String`是 `&str` 类型的：

```
let string = "Hello there." // string: &str
```

这个字符串是静态分配的，也就是说它储存在我们编译好的程序中，并且整个程序的运行过程中一直存在。这个 `String` 绑定了一个静态分配的字符串的引用。字符串片段是固定大小的并且不能改变。

一个 `String`，相反，是一个在堆上分配的字符串。这个字符串可以增长，并且也保证是UTF-8编码的。`String` 通常通过一个字符串片段调用 `to_string` 方法转换而来。

```
let mut s = "Hello".to_string(); // mut s: String
println!("{}", s);

s.push_str(", world.");
println!("{}", s);
```

`String` 可以通过一个 `&` 强转为 `&str`：

```
fn takes_slice(slice: &str) {
    println!("Got: {}", slice);
}

fn main() {
    let s = "Hello".to_string();
    takes_slice(&s);
}
```

把 `String` 当作 `&str` 是廉价的，不过从 `&str` 转换到 `String` 涉及到分配内存。除非必要，没有理由这样做！

这就是Rust字符串的基础！如果你来自一个脚本语言，这可能比你熟悉的字符串要复杂一些，不过当底层细节很重要时，这也就显得十分重要了。你只需记住 `String` 分配内存并控制自己的数据，而 `&str` 是另一个字符串的引用就行了。

数组，向量和片段

像很多编程语言一样，Rust有用来表示数据序列的列表类型。最基本的是数组，一个定长相同类型的元素列表。数组默认是不可变的。

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

这里有一个可以将数组中每一个元素初始化为相同值的简写。在这个例子中，`a` 的每个元素都被初始化为 `0`：

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

数组的类型是 `[T; N]`。我们会在讲解泛型的时候讨论这个 `T` 标记。

你可以用 `a.len()` 来获取 `a` 中元素的数量，用 `a.iter()` 在循环中迭代所有元素。下面的代码会按顺序打印每一个元素：

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

println!("a has {} elements", a.len());
for e in a.iter() {
    println!("{}", e);
}
```

你可以用下标 (*subscript notation*) 来访问特定的元素：

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

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

就跟大部分编程语言一个样，下标从0开始，所以第一个元素是 `names[0]`，第二个是 `names[1]`。上面的例子会打印出 `The second name is: Brian`。如果你尝试使用一个不在数组中的下标，你会得到一个错误：数组访问会在运行时进行边界检查。这种不适当的访问时其它系统编程语言中很多bug的根源。

向量是一个动态或“可增长”的数组，被实现为标准库类型 `Vec`（我们会在后面讨论 `<T>` 是什么意思）。向量一般在堆上分配数据。向量与片段就像 `String` 与 `&str` 一样。你可以使用 `vec!` 宏来创建它：

```
let v = vec![1, 2, 3]; // v: Vec<i32>
```

(与我们之前使用 `println!` 宏时不一样，我们可以在 `vec!` 中使用中括号 `[]`。为了方便，Rust允许你使用上述情况。)

你可以获取向量的长度，迭代和使用下标，就像数组一样。另外，（可变）的向量会自动增长：

```
let mut nums = vec![1, 2, 3]; // mut nums: Vec<i32>
nums.push(4);
println!("The length of nums is now {}", nums.len()); // Prints 4
```

向量有很多有用的方法。

一个片段 (*slice*) 是一个数组的引用（或者“视图”）。它有利于安全，有效的访问数组的一部分而不用进行拷贝。距离来说，你可能只想要引用读入到内存的文件中的一行。原理上，片段并不是直接创建的，而是引用一个已经存在的变量。片段有长度，可以是可变也可以是不可变的，并且表现起来像一个数组。

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

for e in middle.iter() {
    println!("{}", e); // Prints 1, 2, 3
}
```

你也可以取向量的片段，例如 `String` 或者 `&str`，因为向量的底层使用了数组。片段的类型是 `&[T]`，我们会在讲泛型时讨论它。

我们现在已经学会了大部分Rust的基础知识。我们已经准备好创建你个猜猜看的游戏了，我们还需要知道最后一个内容：如何从键盘获取输入。你不能在你都不能猜的情况下实现一个猜猜看的游戏！

标准输入

注意：此章节在原书**Beta**版中被删除，故内容可能过时了，保留仅作为参考！！！

从键盘获取输入是很简单的，不过需要用到一些我们还没见过的东西。下面是一个简单例子，读取一些输入，然后把它打印出来：

```
fn main() {
    println!("Type something!");

    let input = std::old_io::stdin().read_line().ok().expect("Failed to read line");

    println!("{}", input);
}
```

让我们一个接一个的看看这块代码：

```
std::old_io::stdin();
```

这里调用了一个在 `std::old_io` 模块中的 `stdin()` 函数。你可以想象的到，`std` 标准库是由Rust提供的。我们会在后面讨论模块系统。

因为每次都全名是很烦人的，我们可以用 `use` 语句来导入它的命名空间：

```
use std::old_io::stdin;

stdin();
```

然而，实践中最好不要导入单个的函数，而是导入模块，并且只保留一级命名空间：

```
use std::old_io;

old_io::stdin();
```

让我们用上述风格更新下我们的例子：

```
use std::old_io;

fn main() {
    println!("Type something!");

    let input = old_io::stdin().read_line().ok().expect("Failed to read line");

    println!("{}", input);
}
```

接下来：

```
.read_line()
```

`read_line()` 可以从 `stdin()` 的结果中获取一整行的输入。简单明了。

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

还记得这些代码吗？

```
enum OptionalInt {
    Value(i32),
    Missing,
}

fn main() {
    let x = OptionalInt::Value(5);
    let y = OptionalInt::Missing;

    match x {
        OptionalInt::Value(n) => println!("x is {}", n),
        OptionalInt::Missing => println!("x is missing!"),
    }

    match y {
        OptionalInt::Value(n) => println!("y is {}", n),
        OptionalInt::Missing => println!("y is missing!"),
    }
}
```

我们的匹配必须每次都检查它是否有值。在这个例子中，当然，我们知道 `x` 有 `value` 值，不过 `match` 强制我们处理 `missing` 分支。99% 的情况这就是我们要的，不过有时，我们比编译器知道得更多。

一样，`read_line()` 并不返回一行输入。它可能返回一行输入，它也有可能没能成功获取到输入。这有可能发生在我们的程序并不运行在终端上，而是在一个定时任务中，或者有时运行在没有标准输入的环境中。因此，`read_line` 返回一个类似于 `OptionalInt` 的类型：`IoResult<T>`（应该已经改为 `Result<T>` 了，坐等本书原版更新）。我们还未讲到 `IoResult<T>` 因为它是 `OptionalInt` 的泛型形式。目前为止，你可以认为它们是类似的，不过值不是 `i32` 类型的。

Rust 在 `IoResult<T>` 提供了一个 `ok()` 方法，它做了与我们的 `match` 语句一样的工作，不过假设我们有一个有效的值。我们接着在结果上调用 `expect()` 方法，它会在我们没有有效结果时终止程序。我们可以接受这种情况，因为如果我们没有获取到输入，我们的程序也不能工作。大部分情况下，我们需要显式地出来错误情况。`expect()` 允许我们在程序崩溃时打印错误信息。

我们会在后面介绍这写代码工作的具体细节的。现在，这里提供了一个基本的理解。

回到我们刚刚的代码！下面复习一下：

```
use std::old_io;

fn main() {
    println!("Type something!");

    let input = old_io::stdin().read_line().ok().expect("Failed to read line");

    println!("{}", input);
}
```

对于像这样的长语句，Rust给了我们一些使用空格的灵活性。我们可以这样写：

```
use std::old_io;

fn main() {
    println!("Type something!");

    // here, we'll show the types at each step

    let input = old_io::stdin() // std::old_io::stdio::StdinReader
        .read_line() // IoResult<String>
        .ok() // Option<String>
        .expect("Failed to read line"); // String

    println!("{}", input);
}
```

这样写有时可读性更好，有时则更差。一切交由你来判断。

这是你关于从标准输入获取基本输入所需的一切！这并不复杂，不过确实有很多小的细节。

猜猜看

注意：此章节在原书**Beta**版中被删除，故内容可能过时了，保留仅作为参考！！！

好的！我们搞定了Rust的基础。让我们写点逼格er的程序。

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

准备

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

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

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

查看生成的 `Cargo.toml` 文件：

```
[package]

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

Cargo从环境变量中获取这些信息。如果这不对，赶紧修改它。

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

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

让我们编译Cargo为我们生成的项目：

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

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

目。

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

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

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

处理一次猜测

下面，我们要生成一个隐藏的数字。为了实现这个目的，我们要使用我们还未讲到的Rust随机数生成器。Rust的标准库中有一大堆有意思的函数。如果你需要一些代码，可能它们已经被写好了！在这里，我们确实知道Rust有随机数生成器，只不过我们不知道怎么用。

查看文档。Rust有一个页面专门介绍标准库。你可以在这里查看这个[页面](#)。这里有很多信息，不过最亮的部分是搜索栏。在页面的最上面，这有一个文本框你可以输入搜索内容。搜索现在还很原始，不过正在一直变得更好。如果你输入“random”，它将会出现[这个页面](#)。第一个结果是一个链接[std::rand::random](#)。如果我们点击它，我们会看到关于它的文档。

这个页面告诉了我们一些信息：函数的生命，一些解释性文字然后是一个用例。让我们修改我们的代码来加入 `random` 函数并看看会发生什么：

```
use std::old_io;
use std::rand;

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

    let secret_number = (rand::random() % 100) + 1; // secret_number: i32

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

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

    let input = old_io::stdin().read_line()
        .ok()
        .expect("Failed to read line");

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

我们修改的第一点是 `use std::rand`，就像文档里说的。我们接着添加了一个 `let` 表达式来创建一个叫 `secret_number` 的变量绑定，然后我们把它打印出来。

另外，你可能会好奇我们对 `rand::random()` 的结果使用了 `%` 运算符。这个运算符叫做取模，它返回除法的余数。通过对 `rand::random()` 的结果取余数，我们可以限制结果在0到99之间。接着我们对结果加一，使其变成1到100之间。通过取模操作，我们可以生成一个概率非常非常小的结果，不过对于我们的例子来说这并不重要。

让我们用 `cargo build` 来编译项目：

```
$ cargo build
Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
src/main.rs:7:26: 7:34 error: the type of this value must be known in this context
src/main.rs:7      let secret_number = (rand::random() % 100) + 1;
                                         ^
error: aborting due to previous error
```

它不能工作！Rust说“在此上下文中我们必须知道这个值得类型”。这发生了什么？好吧，事实证明 `rand::random()` 可以生成很多类型的值，不仅仅是整形。在这个例子中，Rust不知道 `random()` 应该生成什么类型的随机数。所以我们必须帮助它。对于整数值，我们可以在后面加上一个 `i32` 来告诉Rust它们是整形的，不过这个对函数不起作用。它有一个不同的语法，它看起来像：

```
rand::random::<i32>();
```

它说“请给我一个 `i32` 的随机数”。我们修改代码来加上这个提示：

```
use std::old_io;
use std::rand;

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

    let secret_number = (rand::random::<i32>() % 100) + 1;

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

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

    let input = old_io::stdin().read_line()
        .ok()
        .expect("Failed to read line");

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

试试运行我们的新程序几次：

```
$ cargo run
Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
Running `target/guessing_game`
Guess the number!
The secret number is: 7
```

```

Please input your guess.
4
You guessed: 4
$ ./target/guessing_game
Guess the number!
The secret number is: 83
Please input your guess.
5
You guessed: 5
$ ./target/guessing_game
Guess the number!
The secret number is: -29
Please input your guess.
42
You guessed: 42

```

等等，-29？我们需要一个1到100之间的数字！这里我们有两个选项：我们要么要求 `random()` 生成一个无符号数，它只能是正值；要么我们可以使用 `abs()` 函数。让我们使用无符号数的方法。如果我们需要一个正数，那么我们应该要求它生成一个正数。现在我们的代码看起来这样：

```

use std::old_io;
use std::rand;

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

    let secret_number = (rand::random::<u32>() % 100) + 1;

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

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

    let input = old_io::stdin().read_line()
        .ok()
        .expect("Failed to read line");

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

```

让后试试它：

```

$ cargo run
Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
Running `target/guessing_game`
Guess the number!
The secret number is: 57
Please input your guess.
3
You guessed: 3

```

很好，接下来：让我们比较我们猜的数和隐藏的值。

比较猜测的大小

如果你还记得，在本教程的前面，我们写了一个 `cmp` 函数来比较两个数。让我们把它加进来，再用上一个 `match` 语句来比较我们猜的数和隐藏数：

```
use std::old_io;
use std::rand;
use std::cmp::Ordering;

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

    let secret_number = (rand::random::<u32>() % 100) + 1;

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

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

    let input = old_io::stdin().read_line()
        .ok()
        .expect("Failed to read line");

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

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

fn cmp(a: i32, b: i32) -> Ordering {
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
    else { Ordering::Equal }
}
```

如果我们尝试编译，我们会获得一些错误：

```
$ cargo build
Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
src/main.rs:20:15: 20:20 error: mismatched types: expected `i32` but found
`collections::string::String` (expected i32 but found struct collections::string::String)
src/main.rs:20      match cmp(input, secret_number) {
                    ^
src/main.rs:20:22: 20:35 error: mismatched types: expected `i32` but found `u32` (expected :
src/main.rs:20      match cmp(input, secret_number) {
                    ^
error: aborting due to 2 previous errors
```

这在Rust编程中经常出现，同时这也被认为是Rust的一个最强大的能力。你尝试了一些代码，看看它能否

编译，然后Rust会告诉你你出错了。在这里，我们的 `cmp` 函数需要整形，而我们给了它一个无符号数。这里很好修改，因为 `cmp` 函数是我们写的！让我们把参数改为 `u32` 的：

```
use std::old_io;
use std::rand;
use std::cmp::Ordering;

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

    let secret_number = (rand::random::<u32>() % 100) + 1;

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

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

    let input = old_io::stdin().read_line()
        .ok()
        .expect("Failed to read line");

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

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

fn cmp(a: u32, b: u32) -> Ordering {
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
    else { Ordering::Equal }
}
```

再次编译：

```
$ cargo build
Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
src/main.rs:20:15: 20:20 error: mismatched types: expected `u32` but found
`collections::string::String` (expected u32 but found struct collections::string::String)
src/main.rs:20      match cmp(input, secret_number) {
                           ^
error: aborting due to previous error
```

这个错误与之前的很相似：我们期望一个 `u32`，不过却得到了一个 `String`！这是因为 `input` 变量是来自标准输入的，而我们不光可以猜一个数。试试这个：

```
$ ./target/guessing_game
Guess the number!
The secret number is: 73
Please input your guess.
```

```
hello
You guessed: hello
```

噢！同时你需要注意到我们刚运行了程序，甚至在我们没有编译通过的情况下。这是因为之前顺利编译的版本还保留在那里。一定要小心！

不管如何，我们有了一个 `String`，不过我们需要 `u32`。该怎么办？好吧，这有个函数来处理这个问题：

```
let input = old_io::stdin().read_line()
    .ok()
    .expect("Failed to read line");
let input_num: Result<u32, _> = input.parse();
```

`parse` 函数把一个 `&str` 转换为其它类型。我们可以通过类型提示告诉它需要什么类型。还记得我们给 `random()` 类型提示吗？它像这样：

```
rand::random::<u32>();
```

这还有一种方式提供一个提示，这就是使用 `let` 来声明类型：

```
let x: u32 = rand::random();
```

在这个例子中，我们显式的说明 `x` 是 `u32` 类型的，所以 Rust 能够准确的告诉 `random()` 应该生成什么。下面两种情况都能正确，作为一个相似的方法：

```
let input_num_option = "5".parse::<u32>().ok(); // input_num: Option<u32>
let input_num_result: Result<u32, _> = "5".parse(); // input_num: Result<u32, <u32 as FromStr>::Err>
```

上面，我们也能通过 `ok()` 方法把 `parse` 的结果 `Result` 转换成一个 `option`。不管怎么说，我们把输入转换成了一个数，我们的代码看起来像这样：

```
use std::old_io;
use std::rand;
use std::cmp::Ordering;

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

    let secret_number = (rand::random::<u32>() % 100) + 1;

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

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

    let input = old_io::stdin().read_line()
        .ok()
```

```

        .expect("Failed to read line");
let input_num: Result<u32, _> = input.parse();

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

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

fn cmp(a: u32, b: u32) -> Ordering {
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
    else { Ordering::Equal }
}

```

让我们试试：

```

$ cargo build
Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
src/main.rs:21:15: 21:24 error: mismatched types: expected `u32`, found `core::result::Result<core::num::ParseIntError>` (expected u32, found enum `core::result::Result`)
src/main.rs:21     match cmp(input_num, secret_number) {
                           ^
error: aborting due to previous error

```

好的，我们的 `input_num` 是 `Result<u32, <some error>>` 类型的，而不是 `u32`。我们需要展开这个结果。如果你还记得前面所讲的，`match` 是一个好的方法。试试下面的代码：

```

use std::old_io;
use std::rand;
use std::cmp::Ordering;

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

    let secret_number = (rand::random::<u32>() % 100) + 1;

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

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

    let input = old_io::stdin().read_line()
        .ok()
        .expect("Failed to read line");
    let input_num: Result<u32, _> = input.parse();

    let num = match input_num {
        Ok(n) => n,
        Err(_) => {
            println!("Please input a number!");
            return;
        }
    }
}
```

```

        }
    };

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

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

fn cmp(a: u32, b: u32) -> Ordering {
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
    else { Ordering::Equal }
}

```

我们使用一个 `match` 要么获取 `Result` 中的 `u32` 值，要么打印一个错误信息并退出。让我尝试一下：

```

$ cargo run
Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
Running `target/guessing_game`
Guess the number!
The secret number is: 17
Please input your guess.
5
Please input a number!

```

呃。。。神马？不管怎么说，我们做到了。

。。实际上，我们没有做到。你看，当我们从 `stdin()` 中获取一行输入时，你得到了所有的输入。包括你敲回车产生的 `\n` 字符。因此，`parse()` 认为它是 `5\n` 然后告诉我们“不，这不是一个数字，这里有非数字字符！”幸运的是，`&str` 定义了一个简单的方法可以用来处理它：`trim()`。经过简单的修改，我们的代码看起来像这样：

```

use std::old_io;
use std::rand;
use std::cmp::Ordering;

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

    let secret_number = (rand::random::<u32>() % 100) + 1;

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

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

    let input = old_io::stdin().read_line()
        .ok()
        .expect("Failed to read line");
    let input_num: Result<u32, _> = input.trim().parse();
}

```

```

let num = match input_num {
    Ok(num) => num,
    Err(_) => {
        println!("Please input a number!");
        return;
    }
};

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

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

fn cmp(a: u32, b: u32) -> Ordering {
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
    else { Ordering::Equal }
}

```

让我们试试：

```

$ cargo run
Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
Running `target/guessing_game`
Guess the number!
The secret number is: 58
Please input your guess.
76
You guessed: 76
Too big!

```

好的！你可以看到我甚至在我的猜测之前加了空格，它还是能判断出我猜了76。运行我们的程序几次，包括猜猜比较小的数字，验证我们的猜猜看游戏能正常工作。

Rust在这里提供了很多帮助！这个技巧叫做“依靠编译器”，并且这在写代码时很有帮助。让错误信息帮助我们确定正确的类型。

现在，我们的游戏已经大体上能够工作了，不过我们只能猜一次。让我们加上循环。

循环

如同我们讨论过的，`loop` 关键字代表应给无限循环。让我们加上它：

```

use std::old_io;
use std::rand;
use std::cmp::Ordering;

```

```

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

    let secret_number = (rand::random::<u32>() % 100) + 1;

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

    loop {

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

        let input = old_io::stdin().read_line()
            .ok()
            .expect("Failed to read line");
        let input_num: Result<u32, _> = input.trim().parse();

        let num = match input_num {
            Ok(num) => num,
            Err(_) => {
                println!("Please input a number!");
                return;
            }
        };

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

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

fn cmp(a: u32, b: u32) -> Ordering {
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
    else { Ordering::Equal }
}

```

然后来试试。不过先等等，我们刚刚不是添加了一个无限循环吗？是的，还记得那个 `return` 吗？如果我们猜了一个非数字，我们会 `return` 然后退出。看看：

```

$ cargo run
Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
Running `target/guessing_game`
Guess the number!
The secret number is: 59
Please input your guess.
45
You guessed: 45
Too small!
Please input your guess.
60
You guessed: 60

```

```

Too big!
Please input your guess.
59
You guessed: 59
You win!
Please input your guess.
quit
Please input a number!

```

哈哈！`quit` 确实退出了。就像其它别的非数字输入一样。好吧，这并不是最理想的结果。首先，让我们在猜对的时候才真正退出程序：

```

use std::old_io;
use std::rand;
use std::cmp::Ordering;

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

    let secret_number = (rand::random::<u32>() % 100) + 1;

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

    loop {

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

        let input = old_io::stdin().read_line()
            .ok()
            .expect("Failed to read line");
        let input_num: Result<u32, _> = input.trim().parse();

        let num = match input_num {
            Ok(num) => num,
            Err(_) => {
                println!("Please input a number!");
                return;
            }
        };

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

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

fn cmp(a: u32, b: u32) -> Ordering {
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
}

```

```

    else { Ordering::Equal }
}

```

通过在 You win! 后添加 return , 我们会在猜对的时候退出程序。我们还需做一个小的修改：当我们输入非数字时，我们不希望退出，而是忽略它。把那个 return 改为 continue :

```

use std::old_io;
use std::rand;
use std::cmp::Ordering;

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

    let secret_number = (rand::random::<u32>() % 100) + 1;

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

    loop {

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

        let input = old_io::stdin().read_line()
            .ok()
            .expect("Failed to read line");
        let input_num: Result<u32, _> = input.trim().parse();

        let num = match input_num {
            Ok(num) => num,
            Err(_) => {
                println!("Please input a number!");
                continue;
            }
        };

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

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

fn cmp(a: u32, b: u32) -> Ordering {
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
    else { Ordering::Equal }
}

```

现在好了！让我们试一下：

```
$ cargo run
Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
Running `target/guessing_game`
Guess the number!
The secret number is: 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 a number!
Please input your guess.
61
You guessed: 61
You win!
```

好的！再还有最后小的修改，我们就完成了猜猜看的游戏。你能想到这是什么吗？对了，那就是我们可不想打印出隐藏数字。它方便了测试，不过会毁了这个游戏。下面是我们的最终代码：

```
use std::old_io;
use std::rand;
use std::cmp::Ordering;

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

    let secret_number = (rand::random::<u32>() % 100) + 1;

    loop {

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

        let input = old_io::stdin().read_line()
            .ok()
            .expect("Failed to read line");
        let input_num: Result<u32, _> = input.trim().parse();

        let num = match input_num {
            Ok(num) => num,
            Err(_) => {
                println!("Please input a number!");
                continue;
            }
        };

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

        match cmp(num, secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
        }
    }
}
```

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

fn cmp(a: u32, b: u32) -> Ordering {
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
    else { Ordering::Equal }
}
```

完成

到目前为止，你已经成功的完成了这个猜猜看游戏！祝贺你！

你现在学会了Rust的基本语法。它们大都跟你之前使用过的其它各种编程语言比较相近。这些句法和语义基础将会作为你接下来学习Rust的基础。

现在你是一个基础专家了。是时候去学习下一些Rust更加独特的特性了。

中级

这一部分包含一系列独立而完整的章节，分别关注一些特定的主题。你可以按任意顺序阅读这一部分。

在阅读完“中级”部分后，你将会对Rust有一个充分的了解，你将能阅读大部分Rust代码并能编写更加复杂的程序。

包装箱和模块

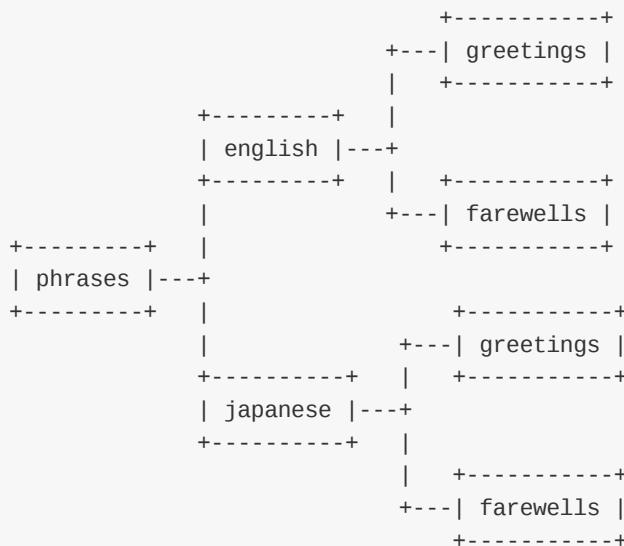
当一个项目开始变得更大，把它分为一堆更小的部分然后再把它们装配到一起被认为是一个好的软件工程实践。另外定义良好的接口，这样有些函数是私有的而有些是私有的，也非常重要。Rust有一个模块系统来帮助我们处理这些工作。

基础术语：包装箱和模块

Rust有两个不同的术语与模块系统有关：包装箱（crate）和模块（module）。包装箱是其它语言中库或包的同义词。因此“Cargo”则是Rust包管理工具的名字：你通过Cargo发送你的包装箱给别人。包装箱可以根据项目的不同生成可执行文件或库文件。

每个包装箱有一个隐含的根模块（root module）包含模块的代码。你可以在根模块下定义一个子模块树。模块允许你为自己模块的代码分区。

作为一个例子，让我们来创建一个短语（phrases）包装箱，它会给我们一些不同语言的短语。为了使事情变得简单，我们仅限于“你好”和“再见”这两个短语，并使用英语和日语的短语。我们采用如下模块布局：



在这个例子中，`phrases` 是我们包装箱的名字。剩下的所有都是模块。你可以看到它们组成了一个树，它们以包装箱为根，这同时也是树的根：`phrases`。

现在我们有了一个计划，让我们在代码中定义这些模块。让我们以用Cargo创建一个新包装箱作为开始：

```
$ cargo new phrases
$ cd phrases
```

如果你还记得的，这会为我们生成一个简单的项目：

```
$ tree .
```

```
.
├── Cargo.toml
└── src
    └── lib.rs

1 directory, 2 files
```

`src/lib.rs` 是我们包装箱的根，与上面图表中的 `phrases` 对应。

定义模块

我们用 `mod` 关键字来定义我们的每一个模块。让我们把 `src/lib.rs` 写成这样：

```
// in src/lib.rs

mod english {
    mod greetings {
        }

        mod farewells {
            }

    }

mod japanese {
    mod greetings {
        }

        mod farewells {
            }

    }
}
```

在 `mod` 关键字之后是模块的名字。模块的命名采用Rust其它标识符的命名惯例：`lower_snake_case`。在大括号中（`{}`）是模块的内容。

在 `mod` 中，你可以定义子 `mod`。我们可以用双冒号（`::`）标记访问子模块。我们的4个嵌套模块是 `english::greetings`，`english::farewells`，`japanese::greetings` 和 `japanese::farewells`。因为子模块位于父模块的命名空间中，所以这些不会冲突：`english::greetings` 和 `japanese::greetings` 是不同的，即便它们的名字都是 `greetings`。

因为这个包装箱的根文件叫做 `lib.rs`，且没有一个 `main()` 函数。Cargo会把这个包装箱构建为一个库：

```
$ cargo build
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
$ ls target
deps  libphrases-a7448e02a0468eaa.rlib  native
```

`libphrase-hash.rlib` 是构建好的包装箱。在我们了解如何使用这个包装箱之前，先让我们把它拆分为多个文件。

多文件包装箱

如果每个包装箱只能有一个文件，这些文件将会变得非常庞大。把包装箱分散到多个文件也非常简单，Rust支持两种方法。

除了这样定义一个模块外：

```
mod english {
    // contents of our module go here
}
```

我们还可以这样定义：

```
mod english;
```

如果我们这么做的话，Rust会期望能找到一个包含我们模块内容的 `english.rs` 文件，或者 `english/mod.rs` 文件：

```
// contents of our module go here
```

注意在这些文件中，你不需要重新定义这些文件：它们已经由最开始的 `mod` 定义。

使用这两个技巧，我们可以将我们的包装箱拆分为两个目录和七个文件：

```
$ tree .
.
├── Cargo.lock
├── Cargo.toml
└── src
    ├── english
    │   ├── farewells.rs
    │   ├── greetings.rs
    │   └── mod.rs
    ├── japanese
    │   ├── farewells.rs
    │   ├── greetings.rs
    │   └── mod.rs
    └── lib.rs
└── target
    ├── deps
    └── libphrases-a7448e02a0468eaa.rlib
        └── native
```

`src/lib.rs` 是我们包装箱的根，它看起来像这样：

```
// in src/lib.rs

mod english;

mod japanese;
```

这两个定义告诉Rust去寻找 `src/english.rs` 和 `src/japanese.rs` , 或者 `src/english/mod.rs` 和 `src/japanese/mod.rs` , 具体根据你的偏好。在我们的例子中，因为我们的模块含有子模块，所以我们选择第二种方式。`src/english/mod.rs` 和 `src/japanese/mod.rs` 都看起来像这样：

```
// both src/english/mod.rs and src/japanese/mod.rs

mod greetings;

mod farewells;
```

再一次，这些定义告诉Rust去寻找 `src/english/greetings.rs` 和 `src/japanese/greetings.rs` , 或者 `src/english/farewells/mod.rs` 和 `src/japanese/farewells/mod.rs` 。因为这些子模块没有自己的子模块，我们选择 `src/english/greetings.rs` 和 `src/japanese/farewells.rs` 。

现在 `src/english/greetings.rs` 和 `src/japanese/farewells.rs` 都是空的。让我们添加一些函数。

在 `src/english/greetings.rs` 添加如下：

```
// in src/english/greetings.rs

fn hello() -> String {
    "Hello!".to_string()
}
```

在 `src/english/farewells.rs` 添加如下：

```
// in src/english/farewells.rs

fn goodbye() -> String {
    "Goodbye.".to_string()
}
```

在 `src/japanese/greetings.rs` 添加如下：

```
// in src/japanese/greetings.rs

fn hello() -> String {
    "こんにちは".to_string()
}
```

当然，你可以从本文复制粘贴这些内容，或者写点别的东西。事实上你写进去“konnichiwa”对我们学习模块系统并不重要。在 `src/japanese/farewells.rs` 添加如下：

```
// in src/japanese/farewells.rs

fn goodbye() -> String {
    "さようなら".to_string()
}
```

(这是“Sayōnara”，如果你很好奇的话。)

现在我们在包装箱中添加了一些函数，让我们尝试在别的包装箱中使用它。

导入外部的包装箱

我们有了一个库包装箱。让我们创建一个可执行的包装箱来导入和使用我们的库。

创建一个 `src/main.rs` 文件然后写入如下：(现在它还不能编译)

```
// in src/main.rs

extern crate phrases;

fn main() {
    println!("Hello in English: {}", phrases::english::greetings::hello());
    println!("Goodbye in English: {}", phrases::english::farewells::goodbye());

    println!("Hello in Japanese: {}", phrases::japanese::greetings::hello());
    println!("Goodbye in Japanese: {}", phrases::japanese::farewells::goodbye());
}
```

`extern crate` 声明高速Rust我们需要编译和链接 `phrases` 包装箱。然后我们就可以在这里使用 `phrases` 的模块了。就想我们之前提到的，你可以用双冒号引用子模块和之中的函数。

另外，Cargo假设 `src/main.rs` 是二进制包装箱的根，而不是库包装箱的。现在我们的包中有两个包装箱：`src/lib.rs` 和 `src/main.rs`。这种模式在可执行包装箱中非常常见：大部分功能都在库包装箱中，而可执行包装箱使用这个库。这样，其它程序可以只使用我们的库，另外这也是各司其职的良好分离。

现在它还不能很好的工作。我们会得到4个错误，它们看起来像：

```
$ cargo build
Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
/home/you/projects/phrases/src/main.rs:4:38: 4:72 error: function `hello` is private
/home/you/projects/phrases/src/main.rs:4      println!("Hello in English: {}", phr
ases::english::greetings::hello());
^~~~~~
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!
/home/you/projects/phrases/src/main.rs:4:5: 4:76 note: expansion site
```

Rust的一切默认都是私有的。让我们深入了解一下这个。

导出公用接口

Rust允许你严格的控制你的接口哪部分是公有的，所以它们默认都是私有的。你需要使用 `pub` 关键字，来公开它。让我们先关注 `english` 模块，所以让我们像这样减少 `src/main.rs` 的内容：

```
// in src/main.rs

extern crate phrases;

fn main() {
    println!("Hello in English: {}", phrases::english::greetings::hello());
    println!("Goodbye in English: {}", phrases::english::farewells::goodbye());
}
```

在我们的 `src/lib.rs`，让我们给 `english` 模块声明添加一个 `pub`：

```
// in src/lib.rs

pub mod english;

mod japanese;
```

然后在我们的 `src/english/mod.rs` 中，加上两个 `pub`：

```
// in src/english/mod.rs

pub mod greetings;

pub mod farewells;
```

在我们的 `src/english/greetings.rs` 中，让我们在 `fn` 声明中加上 `pub`：

```
// in src/english/greetings.rs

pub fn hello() -> String {
    "Hello!".to_string()
}
```

然后在 `src/english/farewells.rs` 中：

```
// in src/english/farewells.rs

pub fn goodbye() -> String {
    "Goodbye.".to_string()
```

```
}
```

这样，我们的包装箱就可以编译了，虽然会有警告说我们没有使用 `japanese` 的方法：

```
$ cargo run
Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
/home/you/projects/phrases/src/japanese/greetings.rs:1:1: 3:2 warning: code is never used:
#[warn(dead_code)] on by default
/home/you/projects/phrases/src/japanese/greetings.rs:1 fn hello() -> String {
/home/you/projects/phrases/src/japanese/greetings.rs:2     "こんにちは".to_string()
/home/you/projects/phrases/src/japanese/greetings.rs:3 }
/home/you/projects/phrases/src/japanese/farewells.rs:1:1: 3:2 warning: code is never used:
#[warn(dead_code)] on by default
/home/you/projects/phrases/src/japanese/farewells.rs:1 fn goodbye() -> String {
/home/you/projects/phrases/src/japanese/farewells.rs:2     "さようなら".to_string()
/home/you/projects/phrases/src/japanese/farewells.rs:3 }

    Running `target/phrases`
Hello in English: Hello!
Goodbye in English: Goodbye.
```

现在我们的函数是公有的了，我们可以使用它们。好的！然

而，`phrases::english::greetings::hello()` 非常长并且重复。Rust 有另一个关键字用来导入名字到当前空间中，这样我们就可以用更短的名字来引用它们。让我们聊聊 `use`。

用 `use` 导入模块

Rust 有一个 `use` 关键字，它允许我们导入名字到我们本地的作用域中。让我们把 `src/main.rs` 改成这样：

```
// in src/main.rs

extern crate phrases;

use phrases::english::greetings;
use phrases::english::farewells;

fn main() {
    println!("Hello in English: {}", greetings::hello());
    println!("Goodbye in English: {}", farewells::goodbye());
}
```

这两行 `use` 导入了两个模块到我们本地作用域中，这样我们就可以用一个短得多的名字来引用函数。作为一个传统，当导入函数时，导入模块而不是直接导入函数被认为是一个最佳实践。也就是说，你可以这么做：

```
extern crate phrases;

use phrases::english::greetings::hello;
use phrases::english::farewells::goodbye;
```

```
fn main() {
    println!("Hello in English: {}", hello());
    println!("Goodbye in English: {}", goodbye());
}
```

不过这并不理想。这意味着更加容易导致命名冲突。在我们的小程序中，这没什么大不了的，不过随着我们的程序增长，它将会成为一个问题。如果我们有命名冲突，Rust会给我们一个编译错误。举例来说，如果我们将 `japanese` 的函数设为公有，然后这样尝试：

```
extern crate phrases;

use phrases::english::greetings::hello;
use phrases::japanese::greetings::hello;

fn main() {
    println!("Hello in English: {}", hello());
    println!("Hello in Japanese: {}", hello());
}
```

Rust会给我们一个编译时错误：

```
Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
/home/you/projects/phrases/src/main.rs:4:5: 4:40 error: a value named `hello`
has already been imported in this module
/home/you/projects/phrases/src/main.rs:4 use phrases::japanese::greetings::hello;
                                         ^
error: aborting due to previous error
Could not compile `phrases`.
```

如果你从同样的模块中导入多个名字，我们不必写多遍。Rust有一个简便的语法：

```
use phrases::english::greetings;
use phrases::english::farewells;
```

你可以使用大括号：

```
use phrases::english::{greetings, farewells};
```

这两中声明是等同的，不过第二种少打更多字。

使用 `pub use` 重导出

你不仅可以用 `use` 来简化标识符。你也可以在包装箱内用它重导出函数到另一个模块中。这意味着你可以展示一个外部接口可能并不直接映射到内部代码结构。

让我们看个例子。修改 `src/main.rs` 让它看起来像这样：

```
// in src/main.rs

extern crate phrases;

use phrases::english::{greetings,farewells};
use phrases::japanese;

fn main() {
    println!("Hello in English: {}", greetings::hello());
    println!("Goodbye in English: {}", farewells::goodbye());

    println!("Hello in Japanese: {}", japanese::hello());
    println!("Goodbye in Japanese: {}", japanese::goodbye());
}
```

然后修改 `src/lib.rs` 公开 `japanese` 模块：

```
// in src/lib.rs

pub mod english;

pub mod japanese;
```

接下来，把这两个函数声明为公有，先是 `src/japanese/greetings.rs`：

```
// in src/japanese/greetings.rs

pub fn hello() -> String {
    "こんにちは".to_string()
}
```

然后是 `src/japanese/farewells.rs`：

```
// in src/japanese/farewells.rs

pub fn goodbye() -> String {
    "さようなら".to_string()
}
```

最后，修改你的 `src/japanese/mod.rs` 为这样：

```
// in src/japanese/mod.rs

pub use self::greetings::hello;
pub use self::farewells::goodbye;

mod greetings;

mod farewells;
```

`pub use` 声明将这些函数导入到了我们模块结构空间中。因为我们在 `japanese` 模块内使用了 `pub use`，我们现在有了 `phrases::japanese::hello()` 和 `phrases::japanese::goodbye()` 函数，即使它们的代码在 `phrases::japanese::greetings::hello()` 和 `phrases::japanese::farewells::goodbye()` 函数中。内部结构并不反映外部接口。

这里我们对每个我们想导入到 `japanese` 空间的函数使用了 `pub use`。我们也可以使用通配符来导入 `greetings` 的一切到当前空间中：`pub use self::greetings::*;`。

另外，注意 `pub use` 出现在 `mod` 定义之前。Rust 要求 `use` 位于最开始。

构建然后运行：

```
$ cargo run
Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
  Running `target/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
```

测试

Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.

Edsger W. Dijkstra, "The Humble Programmer" (1972)

软件测试是证明bug存在的有效方法，而证明它们不存在时则显得令人绝望的不足。

Edsger W. Dijkstra , 【谦卑的程序员】 (1972)

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

测试属性 (The test attribute)

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

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

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

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

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

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

running 1 test
test it_works ... ok

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

Doc-tests adder

running 0 tests

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

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

再讨论这些。现在，看看这行：

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

注意那个 `it_works`。这是我们函数的名字：

```
fn it_works() {
```

然后我们有一个总结行：

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

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

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

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

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

running 1 test
test it_works ... FAILED

failures:

---- it_works stdout ----
    thread 'it_works' panicked at 'assertion failed: false', /home/steve/tmp/adder/src/./it_works:1
    note: run with `RUST_BACKTRACE=1` for a backtrace

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:24'
```

Rust指出我们的测试失败了：

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

这反映在了总结行上：

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

我们也得到了一个非0的状态值：

```
$ echo $?
101
```

这在你想把 cargo test 集成进其它工具是非常有用。

我们可以使用另一个属性反转我们的失败的测试：should_fail：

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

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

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

running 1 test
test it_works ... ok

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

Doc-tests adder

running 0 tests

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

Rust提供了另一个宏，assert_eq! 用来比较两个参数：

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

那个测试通过了吗？因为那个 `should_fail` 属性，它通过了：

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

running 1 test
test it_works ... ok

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

Doc-tests adder

running 0 tests

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

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

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

这就是全部的基础内容！让我们写一个“真实”的测试：

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

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

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

test 模块

这有一个问题令我们当前的项目并不理想：它缺少测试模块。我们例子的理想写法如下：

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

#[cfg(test)]
```

```
mod tests {
    use super::add_two;

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

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

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

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

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

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

注意 `use` 行的变化。现在运行我们的测试：

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

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

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

Doc-tests adder

running 0 tests

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

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

么“集成测试”怎么办呢？我们有 `tests` 目录来处理这些。

tests 目录

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

```
extern crate adder;

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

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

让我们运行一下：

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

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

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

Running target/lib-c18e7d3494509e74

running 1 test
test it_works ... ok

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

Doc-tests adder

running 0 tests

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

现在我们有了三个部分：我们之前的两个测试，然后还有我们新添加的。

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

让我们最后看看第三部分：文档测试。

文档测试

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

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

```
///! The `adder` crate provides functions that add numbers to other numbers.
///!
///! # Examples
///!
///!
```

```
///! 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));
}
```

```
}
```

注意模億态的文档以`///`开剧然后函数态的文档以`///`开剧。Rust文档在注中支持Markdown语法，所以它支持3个

让我 再次运行傾濾：

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

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

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

Running target/lib-c18e7d3494509e74

running 1 test
test it_works ... ok

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

Doc-tests adder

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

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



## 指针

Rust的指针是它最独特最引人入胜的特性之一。指针也是对于Rust新人来说最令人迷惑的主题之一。对于来自像C++这样的支持指针的语言的同学来说它也会是令人迷惑的。这章教程会帮助你理解这个重要的主题。

对于非引用指针持怀疑态度：为了特定的目的使用它，而不是仅仅用它来调戏编译器。我们会解释每个指针类型何时适合使用它们。默认使用引用除非你处于任何一种特定的情况下。

你可能对这个[速查表](#)有兴趣，它提供了一个关于不同指针类型，名称和使用目的的快速预览。

## 简介

如果你对指针的概念还不熟悉，这是一个简单的介绍。指针是系统编程语言中一个非常根本的概念，所以理解它们非常重要。

## 指针基础

当你创建一个新的变量绑定时，你给了一个储存在栈上特定位置的值一个名字（如果你对堆和栈（\*heap vs. stack）的区别还不熟悉，请看看[这个Stack Overflow上的问题](#)，因为下面的介绍中假定你知道这些区别。）例如：

```
let x = 5;
let y = 8;
```

地址	值
0xd3e030	5
0xd3e028	8

我们在这里补充了内存地址，它们都仅仅是虚构的值。不管如何，重点是 `x`，我们给变量的名字。对应着内存地址 `0xd3e030`，然后这个地址的值是 5。当我们引用 `x` 时，我们获得对应的值。也就是说，`x` 是 5。

让我们来介绍指针。在一些语言中，这只有唯一一种类型的“指针”，不过在Rust中，我们有好几种指针。在这个例子中，我们用了一个 Rust 引用 (reference)，它是一种最简单的指针。

```
let x = 5;
let y = 8;
let z = &y;
```

地址	值
0xd3e030	5
0xd3e028	8

0xd3e020

0xd3e028

看出了区别了吗？并不是包含一个值，指针的值是内存中的地址。在我们的例子中，指的是 `y` 的地址。`x` 和 `y` 是 `i32` 类型的，不过 `z` 是 `&i32` 类型的。我们可以用 `{:p}` 格式化字符串打印出这个地址：

```
let x = 5;
let y = 8;
let z = &y;

println!("{:p}", z);
```

这会打印出 `0xd3e028`，那个我们虚构的内存地址。

因为 `i32` 和 `&i32` 是不同的类型。我们不能，比如，把它们相加：

```
let x = 5;
let y = 8;
let z = &y;

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

这会给我们一个错误：

```
hello.rs:6:24: 6:25 error: mismatched types: expected `_`, found `&_` (expected
integral variable, found &-ptr)
hello.rs:6 println!("{}", x + z);
 ^
```

我们可以用 `*` 运算符来逆向引用 (*dereference*) 指针。逆向引用是指访问储存在指针中地址的值。下面的代码可以工作：

```
let x = 5;
let y = 8;
let z = &y;

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

它会打印 `13`。

好的！这就是指针的全部：它们指向一些内存地址。不是其它什么东西。现在我们讨论的什么是指针，让我们讨论一下为什么要用指针。

## 指针运用

Rust的指针非常有用，不过用起来跟其它系统语言有所不同。我们会在后面讲到Rust指针的最佳实践，不过在这里我们说说指针在其它语言中的用法：

在C语言中，字符串是一个 `char` 列表的指针，它以 `null` 字符结束。学会使用字符串的唯一方法唯有非常熟悉指针。

当需要指向不在栈上的内存地址时指针非常有用。举例来说，我们的例子用了两个栈变量，所以我们可以给它们命名。不过如果我们分配了一些堆内存，我们并没有可用的名字。在C语言中，`malloc` 用来分配堆内存，并返回一个指针。

作为一个上面两点更宽泛的变体，任何时候如果你需要一个可变大小的结构，你就需要一个指针。你不能在编译时确定你需要分配多少内存，所以你需要一个指针指向将要分配内存的位置，然后在运行时处理它。

指针在值传递（pass-by-value）的语言中比在引用传递（pass-by-reference）的语言中有用。基本上，计算机语言可以做出两种选择（这是一个虚构的语法，不是Rust）：

```
func foo(x) {
 x = 5
}

func main() {
 i = 1
 foo(i)
 // what is the value of i here?
}
```

在一个值传递的语言中，`foo` 是 `i` 的一个拷贝，然后原始的 `f` 并不会改变。此时，`i` 仍是 `1`。在一个引用传递的语言中，`foo` 会是 `i` 的一个引用，并且因此，可以改变它的值。此时，`i` 将是 `5`。

那么指针用来干嘛的呢？好吧，因为指针可以指向一个内存地址。。。

```
func foo(&i32 x) {
 *x = 5
}

func main() {
 i = 1
 foo(&i)
 // what is the value of i here?
}
```

即使在一个值传递的语言中，现在 `i` 也会是 `5`。你可以看到，因为参数 `x` 是一个指针，我们确实传递了 `foo` 的拷贝，不过因为它指向一个内存地址，当我们后面赋值时，原始的值仍会被改变。这个模式叫做通过值传递引用（pass-reference-by-value）。非常微妙！

## 常见指针问题

我们现在讲述和赞颂了指针。那么缺点是什么？好吧，Rust尝试去减少这种问题，不过在其它语言中仍有这些问题：

未初始化的指针会造成麻烦。举例来说，下面的程序会怎么做？

```
&int x;
*x = 5; // whoops!
```

谁知道呢？我们刚刚声明了一个指针，不过还未指向任何东西，然后它指向的内存地址的值为 5。不过这是什么地址呢？没人知道。这可能是无害的也有可能是灾难的。

如果你结合指针和函数，这就很容易使指针指向的内存无效化。举例：

```
func make_pointer(): &int {
 x = 5;

 return &x;
}

func main() {
 &int i = make_pointer();
 *i = 5; // uh oh!
}
```

`x` 是 `make_pointer` 函数的局部变量，因此，`make_pointer` 一返回它就无效了。不过我们返回了一个指向它内存地址的指针，那么回到 `main` 函数中，我们尝试使用这个指针，这与我们第一个情况非常相似。设置无效的内存地址是不好的。

最后一个关于使用指针大问题的例子，别名 (*aliasing*) 也会是一个问题。两个指针如果指向相同的内存地址就是别名。如下：

```
func mutate(&int i, int j) {
 *i = j;
}

func main() {
 x = 5;
 y = &x;
 z = &x; // y and z are aliased

 run_in_new_thread(mutate, y, 1);
 run_in_new_thread(mutate, z, 100);

 // what is the value of x here?
}
```

在这个虚构的例子中，`run_in_new_thread` 启动一个新线程。因为我们有两个线程，并且它们都操作 `x` 的别名，我们不能确定它们俩哪个会先执行完，因此 `x` 的值事实上是不确定的。更糟糕的是，如果它们之中有一个使他们指向的内存地址无效化了呢？当我们设置了一个无效地址后，我们就跟之前有了同样的问题。

## 结论

这就是一个常见指针概念的基本预览。就像我们之前提及的，Rust有不止一种类型的指针，并且也减轻了我们上面提到的所有问题。这也确实意味着Rust的指针要比其它语言更复杂，不过与有问题的简单指针相比更值得。

## 引用

Rust最基本的指针叫引用。Rust引用看起来像这样：

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

println!("{}", *y);
println!("{:p}", y);
println!("{}", y);
```

我们说“y是x的一个引用”。第一个 `println!` 打印出 y 引用的值，通过逆向引用运算符 `*`。第二个打印出 y 指向的内存地址，通过指针格式化字符串。第三个也会打印出 y 引用的值，因为 `println!` 会帮我们自动逆向引用指针。

这有一个函数取一个引用作为参数：

```
fn succ(x: &i32) -> i32 { *x + 1 }
```

你也可以用 `&` 来创建一个引用。所以我们可以用两种不同的方法调用这个函数：

```
fn succ(x: &i32) -> i32 { *x + 1 }

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

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

上面两行 `println!` 都会打印出6.

当然，如果我们在写实际代码是，我们根本犯不着用引用，直接这么写：

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

引用默认是不可改变的：

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

```
*y = 5; // error: cannot assign to immutable borrowed content `*y`
```

我们可以使用 `mut` 来使其可变，不过只在它引用的值也是可变的时候才行。这是可以的：

```
let mut x = 5;
let y = &mut x;
```

这是不行的：

```
let x = 5;
let y = &mut x; // error: cannot borrow immutable local variable `x` as mutable
```

不可变指针允许别名：

```
let x = 5;
let y = &x;
let z = &x;
```

然而可变指针则不允许：

```
let mut x = 5;
let y = &mut x;
let z = &mut x; // error: cannot borrow `x` as mutable more than once at a time
```

尽管它完全安全，在运行时一个引用的表现与C程序中的正常指针式一样的。它是零开销的。编译器在编译时做了所有安全检查。这个理论允许一个原来叫域指针 (*region pointers*) 的概念。域指针涉及到我们今天叫做生命周期 (*lifetimes*) 的概念。

这有一个简单的解释：你认为这段代码能编译吗：

```
fn main() {
 println!("{}", x);
 let x = 5;
}
```

估计不行。因为你知道 `x` 从它声明的地方开始到它离开作用域是有效的。在这里，它一直到 `main` 函数的结尾。所以你知道这个代码会产生一个错误。我们把这个区域称为声明周期。让我们试试一个更复杂的例子：

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

 if x < 10 {
 let y = &x;
```

```

 println!("Oh no: {}", y);
 return;
}

x -= 1;

println!("Oh no: {}", x);
}

```

这里，我们在 `if` 里借用了一个 `x` 的指针。然而，编译器能够识别出如果 `x` 不是可变的话我们的指针超出作用域，因此，让我们通过编译。下面的不能工作：

```

fn main() {
 let mut x = 5;

 if x < 10 {
 let y = &x;

 x -= 1;

 println!("Oh no: {}", y);
 return;
 }

 x -= 1;

 println!("Oh no: {}", x);
}

```

它会给出如下错误：

```

test.rs:7:9: 7:15 error: cannot assign to `x` because it is borrowed
test.rs:7 x -= 1;
 ^
test.rs:5:18: 5:19 note: borrow of `x` occurs here
test.rs:5 let y = &x;
 ^

```

如你所想，这种类型的分析对人来说是复杂的，对电脑来说也是负责的。这里有一个全面的[专门关于引用、所有权和生命周期的介绍](#)，它详细的讨论了这个主题，如果你想深入了解，看看它。

## 最佳实践

总的来说，比起堆分配应倾向于使用栈分配。任何时候只要可能的话都应选择使用栈分配信息的引用。因此，除非你特定的理由使用其它指针，引用是你默认应该使用的指针类型。我们将在其它指针的最佳实践部分介绍何时应该使用它们。

当你需要一个指针但不需要它的所有权时使用引用。引用只是借用所有权，这在你不需要所有权是显得更合理。也就是说，倾向于：

```
fn succ(x: &i32) -> i32 { *x + 1 }
```

而不是：

```
fn succ(x: Box<i32>) -> i32 { *x + 1 }
```

根据上面的规则推断，引用允许你接受大部分其它指针，这非常实用，你可以不用针对每种指针写一个函数了。也就是说，倾向于：

```
fn succ(x: &i32) -> i32 { *x + 1 }
```

而不是：

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

fn box_succ(x: Box<i32>) -> i32 { *x + 1 }

fn rc_succ(x: Rc<i32>) -> i32 { *x + 1 }
```

注意这样的话调用函数必须稍微修改一下调用它的方式：

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

fn succ(x: &i32) -> i32 { *x + 1 }

let ref_x = &5;
let box_x = Box::new(5);
let rc_x = Rc::new(5);

succ(ref_x);
succ(&*box_x);
succ(&*rc_x);
```

开始先 \* 逆向引用指针，在 & 取它内容的引用。

## 装箱指针 (Boxes)

`Box<T>` 是Rust的装箱指针 (*boxed pointer*) 类型。装箱指针是Rust最简单的堆分配形式。像这样创建一个装箱指针：

```
let x = Box::new(5);
```

装箱在堆上分配，并且在它离开作用域时Rust会自动释放它：

```
{
 let x = Box::new(5);

 // stuff happens

} // x is destructed and its memory is free'd here
```

然而，装箱并不使用引用计数或垃圾回收。装箱是所谓的仿射类型 (*affine type*)。这意味着Rust编译器，在编译时确定装箱是否在作用域中，然后插入合适的调用。更准确的，装箱是一个叫做区域 (*region*) 的特殊的仿射类型。你可以在这篇关于Cyclone编程语言的文章中了解区域的概念。

你并不需要完全意会仿射类型和域才能理解装箱。作为一个粗略的估计，你可以认为如下Rust代码：

```
{
 let x = Box::new(5);

 // stuff happens
}
```

如下C语言类似：

```
{
 int *x;
 x = (int *)malloc(sizeof(int));
 *x = 5;

 // stuff happens

 free(x);
}
```

当然这是一个不准确的类比。比如这忽略了析构函数。不过大体的概念是正确的：你使用了 `malloc/free` 的语义，并且带有一些改进：

1. 不可能分配不正确数量的内存，因为Rust根据类型判断它的大小
2. 你不可能“忘记”释放你分配的内存，因为Rust为你做了这些。
3. Rust确保释放发生在合适的时间，当它真正不再被使用的时候。释放后使用是不可能的。
4. Ruse强制没用其它可写指针是这个堆内存的别名，也就是说写一个无效的指针是不可能的。

看看关于引用的部分或者[所有权指南](#)详细了解生命周期是如何工作的。

同时使用装箱和引用时非常常见的。例如：

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

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

```

 println!("{}", add_one(&x));
}

```

在这个例子中，Rust知道 `x` 被 `add_one()` 函数借用了，然后因为这是对它值的唯一使用，允许它。

我们可以借用 `x` 多次，只要它们不是同时发生的。

```

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

fn main() {
 let x = Box::new(5);

 println!("{}", add_one(&x));
 println!("{}", add_one(&x));
 println!("{}", add_one(&x));
}

```

只要这不是一个可变借用，下面会出错

```

fn add_one(x: &mut i32) -> i32 {
 *x + 1
}

fn main() {
 let x = Box::new(5);

 println!("{}", add_one(&x)); // error: cannot borrow immutable dereference
 // of `&`-pointer as mutable
}

```

注意我们修改了 `add_one()` 来接收一个可变的引用。

## 最佳实践

装箱适合在下面两种情况下使用：递归数据结构，偶尔作为返回数据。

### 递归数据结构

有时你需要一个递归数据结构。最简单的比如 `cons` 列表 (`cons list`)：

```

#[derive(Debug)]
enum List<T> {
 Cons(T, Box<List<T>>),
 Nil,
}

fn main() {
 let list: List<i32> = List::Cons(1, Box::new(List::Cons(2,

```

```
Box::new(List::Cons(3, Box::new(List::Nil))));
 println!("{:?}", list);
}
```

这会打印：

```
Cons(1, Box(Cons(2, Box(Cons(3, Box(Nil))))))
```

如果想引用 cons 枚举中的另一个 List 必须使用装箱，因为我们并不知道列表的长度。因为我们不知道长度，我们也就无法知道大小，因此，我们需要在堆上分配我们的列表。

处理未知大小的递归数据结构是装箱的主要用法。

## 返回数据

这足够重要来单独形成一个部分。TL;DR 是：基本上你是不会想要返回指针的，甚至在你用 C 或 C++ 的时候。

浏览[返回指针](#)获取更多信息。

## Rc 和 Arc

即将到来

## 最佳实践

即将到来

## 裸指针 (Raw Pointers)

即将到来

## 最佳实践

即将到来

## 创建你自己的指针

即将到来

## 最佳实践

即将到来

## 模式和 ref

如果你想要匹配储存在指针里的内容，可能直接匹配并不是最好的选择。让我们看看应该如何正确处理这些：

```
fn possibly_print(x: &Option<String>) {
 match *x {
 // BAD: cannot move out of a `&`
 Some(s) => println!("{}", s)

 // GOOD: instead take a reference into the memory of the `Option`
 Some(ref s) => println!("{}", *s),
 None => {}
 }
}
```

这里的 `ref s` 是指 `s` 会是 `&String` 类型，而不是 `String` 类型。

这在你尝试访问一个带有构造函数的类型而你又不想移动它的时候很重要，你只是需要一个引用而已。

## 速查表

这是一个Rust指针类型快速描述：

类型	名称	概要
<code>&amp;T</code>	引用	允许一个或多个引用读写 <code>T</code>
<code>&amp;mut T</code>	可变引用	允许唯一一个可读写的 <code>T</code> 的引用
<code>Box&lt;T&gt;</code>	装箱	堆分配 <code>T</code> 的唯一可读写的拥有者
<code>Rc&lt;T&gt;</code>	“arr cee”指针	允许多个读取堆分配的 <code>T</code>
<code>Arc&lt;T&gt;</code>	Arc指针	同上，不过是线程安全的
<code>*const T</code>	裸指针	可以不安全读 <code>T</code>
<code>*mut T</code>	可变裸指针	可以不安全读写 <code>T</code>

## 相关资源

- [装箱API文档](#)
- [所有权教程](#)
- [Cyclone语言的域](#)，这启发了Rust的声明周期系统

# 所有权

---

本教程展示Rust的所有权系统。这也是Rust最独特最引人入胜的特性之一，也是作为Rust开发者应该知晓的。所有权是Rust取得其最大的目标内存安全的关键。所有权系统有一些不同的主题：所有权，借用和生命周期。让我们来依此讨论它们。

## 原则 (Meta)

---

在我们开始详细讲解之前，这有两点关于所有权系统重要的注意事项。

Rust注重安全和速度。它通过很多零开销抽象 (*zero-cost abstractions*) 来实现这些目标，也就是说在Rust中，实现抽象的开销尽可能的小。所有权系统是一个主要的零开销抽象的例子。本文提到所有的分析都是在编译时完成的。你不需要在运行时为这些功能付出任何开销。

然而，这个系统确实有一个开销：学习曲线。很多Rust初学者会经历我们所谓的“与借用检查器作斗争”的过程，也就是指Rust编译器拒绝编译一个作者认为合理的程序。这经常发生因为程序猿关于所有权系统如何工作的心智模型与Rust实现的现实规则不匹配。你最开始可能会有相似的经历。然而这是一个好消息：更有经验的Rust开发者反应一旦他们适应所有权系统一段时间之后，与借用检查器的冲突会越来越少。

记住这些之后，让我们学习所有权。

## 所有权 (Ownership)

---

作为它的核心，所有权是关于资源 (*resources*) 的。作为本文大部分内容的目的，我们讨论一个特定的资源：内存。这个原则可以泛化到任何类型的资源，比如文件句柄，不过为了使它更加具体，我们将关注内存。

当你的程序分配了一些内存后。它需要一些释放它们的方法。想象一个函数 `foo` 分配4字节的内存，但之后从不释放内存。我们称之为内存泄露，因为我们每次调用 `foo`，我们会分配另外4个字节。最终，调用足够多次 `foo` 后，我们会内存溢出。这是不好的。我们需要让 `foo` 释放那4个字节。另外，不要多次释放内存也很重要。在不涉及底层细节时，尝试释放同一内存多次可能导致问题。换句话说，每当一些内存被分配后，我们需要确保我们释放它一次且仅有一次。过度不好，不足也不行。数量必须正好。

关于分配内存还有一个重要的细节。每当我们获取一些内存的时候，我们会得到一个指向那片内存的句柄。这个句柄（通常叫做指针，当我们引用内存时）用来与被分配的内存交互。只要我们有这个句柄，我们就可以处理这篇内存。当我们不再保有句柄时，我们也就不再保有那片内存，因为没有句柄的话我们无法做任何事。

历史上，系统编程语言曾要求你自己关注分配，释放和句柄。例如，在像C这样的语言中如果你想要一些堆内存，我们这么做：

```
int *x = malloc(sizeof(int));
// we can now do stuff with our handle x
*x = 5;
```

```
 free(x);
}
```

`malloc` 调用分配一些内存。 `free` 释放内存。它们也会记录分配内存的正确数量。

Rust将关于内存分配（或其它资源）的两方面综合为一个叫做所有权（*ownership*）的概念。每当你获取一些内存后，我们接收到的句柄叫做所有句柄（*owning handle*）。当这个句柄离开作用域后，Rust知道你将无法在使用这块内存了，所以就会帮你释放这片内存。下面是Rust一个等价的例子：

```
{
 let x = Box::new(5);
}
```

`Box::new` 函数通过在堆上分配足够放下一个 `i32` 的内存片段来创建了一个 `Box<T>` （在这里特指 `Box<i32>`）。不过释放内存的代码在哪呢？我们之前说过我们必须在每次分配后释放。Rust为你处理了这些。它知道我们的句柄，`x`，拥有装箱的引用。Rust知道 `x` 会在代码块的结尾离开作用域，所以在块的末尾插入一个释放内存的调用。因为编译器为我们做了这些，所以不可能会忘记。我们总是准确的有一个分配对应一个释放。

这是非常直观的，不过如果我们想把我们的装箱传递给一个函数会发生什么呢？让我们看看一些代码：

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

 add_one(x);
}

fn add_one(mut num: Box<i32>) {
 *num += 1;
}
```

这些代码能工作，不过并不理想。例如，让我们再加上一行代码，我们会打印出 `x` 的值：

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

 add_one(x);

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

fn add_one(mut num: Box<i32>) {
 *num += 1;
}
```

这不能编译，并会给我们一个错误：

```
error: use of moved value: `x`
```

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

还记得我们需要一个分配对应一个释放吗？当我们尝试传递我们的装箱给 `add_one` 时，我们会有内存的两个句柄：`main` 的 `x` 和 `add_one` 的 `num`。如果我们在每个句柄离开作用域时都释放内存，我们会有两次释放和一次分配，这是不对的。所以当我们调用 `add_one` 时，Rust 定义 `num` 为句柄的所有者。然后，现在我们把所有权交给了 `num`。`x` 的值从 `x` “移动”到了 `num`。因此会有如下错误：使用被移动的值 `x`。

为了修改这个问题，我们可以在 `add_one` 处理完装箱后收回所有权：

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

 let y = add_one(x);

 println!("{}", y);
}

fn add_one(mut num: Box<i32>) -> Box<i32> {
 *num += 1;

 num
}
```

这些代码会编译并正常运行。现在，我们返回了一个 `box`，并且它的所有权也被转换到了 `main` 函数。在函数归还它之前我们才拥有所有权。这个模式非常常见，并且 Rust 引入了一个概念来描述一个临时指向另一个句柄所有权的句柄。这叫做借用 (*borrowing*)，and it's done with references, designated by the & symbol.

## 借用

这是目前 `add_one` 的状态：

```
fn add_one(mut num: Box<i32>) -> Box<i32> {
 *num += 1;

 num
}
```

这个函数获取所有权，因为它获取了一个拥有所有权的装箱。不过之后又交还了所有权。

在现实生活中，你可以把你一部分的财产交给某人一小段时间。你仍然拥有你的财产，你只是让别人使用它一小会儿。我们称这为你借出 (*lending*) 某物给某人，然后某人从你那借用 (*borrowing*) 了某物。

Rust 的所有权系统也允许你借出所有权一段时间。这也叫做借用。这有一个版本的 `add_one` 借用而不是获取它参数的所有权：

```
fn add_one(num: &mut i32) {
 *num += 1;
```

```
}
```

这个函数从调用者那里借用了一个 `i32`，然后增加了它。当函数执行完毕，`num` 离开作用域，借用也随之结束。

我么也必须修改一下 `main`：

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

 add_one(&mut x);

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

fn add_one(num: &mut i32) {
 *num += 1;
}
```

我们不再需要给 `add_one()` 的结果赋值，因为它并不返回什么东西。这是因为我们没有归还所有权，因为我们现在只是借用，没有获取。

## 生命周期

---

然而，借出一个其它人所有资源的引用是很复杂的。例如，想象一下下列操作：

1. 我获取了一个某种资源的句柄
2. 我借给你了一个引用
3. 我决定不再需要这个资源了，然后释放了它，这时你仍然持有它的引用
4. 你决定使用这个资源

噢！你的引用指向一个无效的资源。这叫做不定指针 (*dangling pointer*) 或者“释放后使用”，如果这个资源是内存的话。

要修正这个问题的话，我们必须确保第四步永不在第三步之后发生。Rust 所有权系统通过一个叫声明周期 (*lifetime*) 的概念来做到这一点，它定义了一个引用有效的作用域。

还记得那个借用了一个 `i32` 的函数吗？让我们再看看它：

```
fn add_one(num: &mut i32) {
 *num += 1;
}
```

Rust 有一个功能叫做生命周期省略 (*lifetime elision*)，它允许你在特定情况下不写生命周期标记。我们会在后面讲到它。在不省略生命周期的情况下，`add_one` 看起来像这样：

```
fn add_one<'a>(num: &'a mut i32) {
 *num += 1;
```

```
}
```

'a 叫做一个生命周期。大部分生命周期都使用像 'a , 'b 和 'c 这样的短而清楚的名字，不过通常一个描述性的名字也很有用。让我们更深入地了解一下语法：

```
fn add_one<'a>(...)
```

这部分声明了我们的生命周期。它说 add\_one 有一个生命周期 'a 。如果我们有两个生命周期，它看起来像这样：

```
fn add_two<'a, 'b>(...)
```

之后，在参数列表里，我们使用刚刚命名的生命周期：

```
...(num: &'a mut i32)
```

如果你对比一下 &mut i32 和 &'a mut i32 ，他们是一样的，只是后者在 & 和 mut i32 之间夹了一个 'a 生命周期。 &mut i32 读作“一个i32的可变引用”，而 &'a mut i32 读作“一个带有生命周期'a的i32的可变引用”。

为什么生命周期重要呢？好吧，举例来说，下面是代码：

```
struct Foo<'a> {
 x: &'a i32,
}

fn main() {
 let y = &5; // this is the same as `let _y = 5; let y = &_y;`
 let f = Foo { x: y };

 println!("{}", f.x);
}
```

如你所见，结构体也可以拥有生命周期。跟函数类似的写法：

```
struct Foo<'a> {
```

声明一个生命周期：

```
x: &'a i32,
```

然后使用它。那么为什么我们这需要一个生命周期呢？我们需要确保任何 Foo 的引用不能比它对 i32 的引用活的更久。

## 理解作用域 (Thinking in scopes)

理解生命周期的一个办法是想象一个引用有效的作用域。例如：

```
fn main() {
 let y = &5; // -+ y goes into scope
 // |
 // stuff // |
 // | // |
} // -+ y goes out of scope
```

加入我们的 `Foo` :

```
struct Foo<'a> {
 x: &'a i32,
}

fn main() {
 let y = &5; // -+ y goes into scope
 let f = Foo { x: y }; // -+ f goes into scope
 // stuff // |
 // | // |
} // -+ f and y go out of scope
```

我们的 `f` 生存在 `y` 的作用域之中，所以一切正常。那么如果不是呢？下面的代码不能工作：

```
struct Foo<'a> {
 x: &'a i32,
}

fn main() {
 let x; // -+ x goes into scope
 // |
 {
 // |
 let y = &5; // ---+ y goes into scope
 let f = Foo { x: y }; // ---+ f goes into scope
 x = &f.x; // | | error here
 } // ---+ f and y go out of scope
 // |
 println!("{}", x); // |
 // -+ x goes out of scope
}
```

(口哨) 就像你看到的一样，`f` 和 `y` 的作用域小于 `x` 的作用域。不过当我们尝试 `x = &f.x` 时，我们让 `x` 引用将要离开作用域的变量。

命名作用域用来赋予作用域一个名字。有了名字是我们谈论它的第一步。

### 'static

叫做 `static` 的作用域是特殊的。它代表其具有一个整个程序的作用域。大部分Rust程序员当他们处理字符

串时第一次遇到 `'static` :

```
let x: &'static str = "Hello, world.;"
```

基本字符串是 `&'static str` 类型的因为它的引用一直有效：它们被写入了最终库文件的数据段。另一个例子是全局量：

```
static FOO: i32 = 5;
let x: &'static i32 = &FOO;
```

它在二进制文件的数据段中保存了一个 `i32`，而 `x` 是它的一个引用。

## 共享所有权

在之前的所有例子中，我们假设每个句柄只有一个所有者。不过有的时候并不如此。考虑一辆车，车有4个轮子。我们想通过一个轮子来判断它装在哪个车上。不过这不行：

```
struct Car {
 name: String,
}

struct Wheel {
 size: i32,
 owner: Car,
}

fn main() {
 let car = Car { name: "DeLorean".to_string() };

 for _ in 0..4 {
 Wheel { size: 360, owner: car };
 }
}
```

我们尝试创建4个轮子，每一个都有一个代表它装在那辆车上的字段。不过编译器在循环的第二遍发现了这个问题：

```
error: use of moved value: `car`
 Wheel { size: 360, owner: car };
 ^~~
note: `car` moved here because it has type `Car`, which is non-copyable
 Wheel { size: 360, owner: car };
 ^~~
```

我们需要 `car` 被多个 `Wheel` 引用。我们用 `Box<T>` 无法做到这些，因为它只有一个所有者。我们可以用 `Rc<T>` 来代替：

```

use std::rc::Rc;

struct Car {
 name: String,
}

struct Wheel {
 size: i32,
 owner: Rc<Car>,
}

fn main() {
 let car = Car { name: "DeLorean".to_string() };

 let car_owner = Rc::new(car);

 for _ in 0..4 {
 Wheel { size: 360, owner: car_owner.clone() };
 }
}

```

我们把 `Car` 包装到一个 `Rc<T>` 中，得到一个 `Rc<Car>`，然后使用 `clone()` 方法生成新的引用。我们也修改了 `Wheel` 使用 `Rc<Car>` 而不仅仅是 `Car`。

这可能是最简单的多所有权的例子。例如，这里还有 `Arc<T>`，它是一个使用更昂贵的原子操作来确保线程安全的 `Rc<T>` 的替代品。

## 生命周期省略 (Lifetime Elision)

之前，我们提到生命周期省略，Rust的一个允许你在特定情况下不写生命周期标记的功能。所有引用都有一个生命周期，然而如果你省略了一个生命周期（比如 `&T` 而不是 `&'a T`），Rust会做3件事来确定它的生命周期应该是什么样的。

当我们讨论生命周期省略的时候，我们使用输入生命周期和输出生命周期 (*input lifetime and output lifetime.*)。输入生命周期是关于函数参数的，而输出生命周期是关于函数返回值的。例如，这个函数有一个输入生命周期：

```
fn foo<'a>(bar: &'a str)
```

这个有一个输出生命周期：

```
fn foo<'a>() -> &'a str
```

这个两者皆有：

```
fn foo<'a>(bar: &'a str) -> &'a str
```

这里有3条规则：

- 每一个被省略的函数参数成为一个不同的生命周期参数。
- 如果确实有一个输入生命周期，不管是否省略，这个生命周期被赋予所有函数返回值中被省略的生命周期。
- 如果这里有多个输入生命周期，不过它们当中有一个是 `&self` 或者 `&mut self`，`self` 的生命周期被赋予所有省略的输出生命周期。

否则，省略一个输出生命周期将是一个错误。

## 例子

这里有一些省略了生命周期的函数，还有对应的非省略版本：

```
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
```

## 相关资料

---

即将到来

## 更多字符串

字符串是一个在任何编程语言中都需要掌握的重要内容。如果你来自托管语言环境，你可能会对系统编程语言对字符串处理的复杂性感到惊讶。为一个动态大小的结构有效的分配和访问内存涉及到很多细节。幸运的是Rust有很多工具来帮助我们。

一个字符串是一串UTF-8字节编码的Unicode量级值的序列。所有的字符串都确保是有效编码的UTF-8序列。另外，字符串并不以null结尾并且可以包含null字节。

Rust有两种主要的字符串类型：`&str` 和 `String`。

### `&str`

第一种类型是 `&str`。它念做“字符串片段”。字符串字面上指 `&str`：

```
let string = "Hello there.;"
```

就像任何Rust引用一样，字符串片段有一个相应的生命周期。一个字面上的字符串是一个 `&'static str` 类型。很多情况下字符串片段不需要一个显式的生命周期，例如作为函数参数。这种情况下它的生命周期可以被推断出来：

```
fn takes_slice(slice: &str) {
 println!("Got: {}", slice);
}
```

就像向量片段一样，字符串片段就是简单的指针加长度。这意味着它是一个已经被分配了的字符串的僕，例如匿名字符串（string literal）或 `String`。

### `String`

`String` 是一个在堆上分配的字符串。这个字符串可以增长，并且也保证是UTF-8编码的。`String` 通常通过一个字符串片段调用 `to_string` 方法转换而来。

```
let mut s = "Hello".to_string();
println!("{}", s);

s.push_str(", world.");
println!("{}", s);
```

一个 `String` 的引用会自动强转为一个字符串片段：

```
fn takes_slice(slice: &str) {
 println!("Got: {}", slice);
```

```

 }

fn main() {
 let s = "Hello".to_string();
 takes_slice(&s);
}

```

你也从一个基于栈的字节数组取得 `&str` :

```

use std::str;

let x: &[u8] = &[b'a', b'b'];
let stack_str: &str = str::from_utf8(x).unwrap();

```

## 最佳实践

---

### String vs. &str

一般来说，你应该在你需要所有权的时候倾向于使用 `String`，而在只想借用一个字符串时使用 `&str`。这大体上与 `Vec<T>` 之于 `&[T]` 和 `T` 之于 `&T` 相似。

这意味着你应该这么开始：

```
fn foo(s: &str) {
```

而这么写的：

```
fn foo(s: String) {
```

如果你有非常好的理由。保留你不需要的所有权是不优雅的，并且这回令你的生命周期变得更加复杂。

## 泛型函数

用 `&str` 来写一个字符串泛型的函数。

```

fn some_string_length(x: &str) -> uint {
 x.len()
}

fn main() {
 let s = "Hello, world";

 println!("{}", some_string_length(s));

 let s = "Hello, world".to_string();

 println!("{}", some_string_length(&s));
}

```

```
}
```

这两行都会打印 `12`。

## 字符串索引 (Indexing strings)

你可能尝试访问 `String` 中的一个特定字符，像这样：

```
let s = "hello".to_string();
println!("{}" , s[0]);
```

这不能编译。这是有意为之的。在UTF-8编码世界中，直接索引基本上不会是你想要的。原因是每个字符可以是不定长字节组成的。这意味着如果你必须遍历字符，这将是一个O(n)操作。

这有3个级别的unicode (和它的编码)：

- 代码单元，用来储存一切的底层数据类型
- 代码点/unicode标量值
- 字母 (grapheme) (可见字符)

Rust提供这3种情况下的迭代器：

- `.bytes()` 会遍历底层字节
- `.chars()` 会遍历代码点
- `.graphemes()` 会遍历字母

通常，对 `&str` 使用 `graphemes()` 将是你想要的：

```
let s = "unicode";
for l in s.graphemes(true) {
 println!("{}" , l);
}
```

这会打印：

```
u
n
i
c
o
d
e
```

注意 `l` 在这里是 `&str` 类型的，因为每个字母可以含有多个代码点，所以 `char` 是不合适的。

这回按顺序打印出每个可见的字符，正如你期望的：先是 `u`，再是 `n`...如果你需要每个字母的代码点可以你可使用 `.chars()`：

```
let s = "unicode";
for l in s.chars() {
 println!("{}", l);
}
```

这会打印：

u

n

i

c

o

d

e

你可以看到它们有些是组合字符，所以输出看起来有些奇怪。

如果你需要每个代码点的单独字节表示，你可以使用 `.bytes()`：

```
let s = "unicode";
for l in s.bytes() {
 println!("{}", l);
}
```

这会打印：

```
205
148
110
204
142
205
136
204
176
105
204
153
204
174
205
154
204
166
99
204
137
205
154
111
205
151
204
188
204
169
204
176
100
204
134
205
131
205
165
205
148
101
204
129
```

比字母多了去了！

## 其它文档

---

- [&str API 文档](#)
- [String API 文档](#)

## 模式

我们在教程使用过几次模式：第一个是 `let` 绑定，接着是 `match` 语句。让我们开始一个快速的关于模式可以干什么的教程！

一个快速回顾：你可以直接匹配基本字符串，并且 `_` 作为“任意”类型：

```
let x = 1;

match x {
 1 => println!("one"),
 2 => println!("two"),
 3 => println!("three"),
 _ => println!("anything"),
}
```

你可以使用 `|` 匹配多个模式：

```
let x = 1;

match x {
 1 | 2 => println!("one or two"),
 3 => println!("three"),
 _ => println!("anything"),
}
```

你可以用 `...` 匹配一个范围的值：

```
let x = 1;

match x {
 1 ... 5 => println!("one through five"),
 _ => println!("anything"),
}
```

范围经常用在整数和单个字符上。

如果你匹配多个值，通过 `|` 或 `...`，你可以用 `@` 给这个值绑定一个名字：

```
let x = 1;

match x {
 e @ 1 ... 5 => println!("got a range element {}", e),
 _ => println!("anything"),
}
```

如果你匹配一个带有变体的枚举，你可以用 `..` 来省略变体的值和类型：

```

enum OptionalInt {
 Value(i32),
 Missing,
}

let x = OptionalInt::Value(5);

match x {
 OptionalInt::Value(..) => println!("Got an int!"),
 OptionalInt::Missing => println!("No such luck."),
}

```

你可以用 `if` 来引入匹配守卫 (*match guards*) :

```

enum OptionalInt {
 Value(i32),
 Missing,
}

let x = OptionalInt::Value(5);

match x {
 OptionalInt::Value(i) if i > 5 => println!("Got an int bigger than five!"),
 OptionalInt::Value(..) => println!("Got an int!"),
 OptionalInt::Missing => println!("No such luck."),
}

```

如果你匹配一个指针，你可以跟声明它一样的语法。首先，`&` :

```

let x = &5;

match x {
 &val => println!("Got a value: {}", val),
}

```

这里，`match` 中的 `val` 是 `i32` 类型的。换句话说，模式的左边解构它的值。如果我们有 `&5`，那么在 `&val` 中，`val` 是 `5`。

如果你想要一个引用，使用 `ref` 关键字：

```

let x = 5;

match x {
 ref r => println!("Got a reference to {}", r),
}

```

这里，`match` 中的 `r` 是 `&i32` 类型的。换句话说，`ref` 关键字创建了一个在模式中使用的引用。如果你需要一个可变引用，`ref mut` 同样可以做到：

```
let mut x = 5;

match x {
 ref mut mr => println!("Got a mutable reference to {}", mr),
}
```

如果你有一个结构体，你可以在模式中解构它：

```
struct Point {
 x: i32,
 y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
 Point { x: x, y: y } => println!("({}, {})", x, y),
}
```

如果你只关心部分值，我们不需要给它们都命名：

```
struct Point {
 x: i32,
 y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
 Point { x: x, .. } => println!("x is {}", x),
}
```

你可以对任何成员进行这样的匹配，不仅仅是第一个：

```
struct Point {
 x: i32,
 y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
 Point { y: y, .. } => println!("y is {}", y),
}
```

如果你想匹配一个片段或数组，你可以用 &：

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

```
match &v[..] {
 ["match_this", second] => println!("The second element is {}", second),
 _ => {},
}
```

(口哨)！这里有很多种匹配的方法，它们都能进行组合匹配，根据你想干什么：

```
match x {
 Foo { x: Some(ref name), y: None } => ...
}
```

模式灰常强大，好好使用它们。

## 方法语法

函数是伟大的，不过如果你在一些数据上调用了一堆函数，这将是令人尴尬的。考虑下面代码：

```
baz(bar(foo(x)));
```

我们可以从左向右阅读，我们会看到“baz bar foo”。不过这不是函数被调用的顺序，调用应该是从内向外的：“foo bar baz”。如果能这么做不是更好吗？

```
x.foo().bar().baz();
```

幸运的是，正如对上面那个问题的猜测，你可以！Rust通过 `impl` 关键字提供了使用方法调用语法 (*method call syntax*)。

## 方法调用

这是它如何工作的：

```
struct Circle {
 x: f64,
 y: f64,
 radius: f64,
}

impl Circle {
 fn area(&self) -> f64 {
 std::f64::consts::PI * (self.radius * self.radius)
 }
}

fn main() {
 let c = Circle { x: 0.0, y: 0.0, radius: 2.0 };
 println!("{}", c.area());
}
```

这会打印 `12.566371`。

我们创建了一个代表圆的结构体。我们写了一个 `impl` 块，并且在里面定义了一个方法，`area`。方法的第一参数比较特殊，`&self`。它有3种变体：`self`，`&self` 和 `&mut self`。你可以认为这第一个参数就是 `x.foo()` 中的 `x`。这3种变体对应 `x` 可能的3种类型：`self` 如果只是栈上的一个值，`&self` 如果是一个引用，然后 `&mut self` 如果是一个可变引用。我们应该默认使用 `&self`，因为它最常见。这是一个三种变体的例子：

```
struct Circle {
 x: f64,
 y: f64,
```

```

 radius: f64,
 }

impl Circle {
 fn reference(&self) {
 println!("taking self by reference!");
 }

 fn mutable_reference(&mut self) {
 println!("taking self by mutable reference!");
 }

 fn takes_ownership(self) {
 println!("taking ownership of self!");
 }
}

```

最后，你可能还记得，一个圆的面积是  $\pi \cdot r^2$ 。因为我们向 `area` 传递了 `&self` 参数，我们可以像任何其它参数那样使用它。因为我们知道它是一个 `Circle`，我们可以像处理其它结构体一样访问 `radius`。导入 `π` 再进行一些乘法，我们就有了面积。

## 链式方法调用 (Chaining method calls)

现在我们知道如何调用方法了，例如 `foo.bar()`。那么我们最开始的那个例子呢，`foo.bar().baz()`？我们称这个为“方法链”，我们可以通过返回 `self` 来做到这点。

```

struct Circle {
 x: f64,
 y: f64,
 radius: f64,
}

impl Circle {
 fn area(&self) -> f64 {
 std::f64::consts::PI * (self.radius * self.radius)
 }

 fn grow(&self) -> Circle {
 Circle { x: self.x, y: self.y, radius: (self.radius * 10.0) }
 }
}

fn main() {
 let c = Circle { x: 0.0, y: 0.0, radius: 2.0 };
 println!("{}", c.area());

 let d = c.grow().area();
 println!("{}", d);
}

```

注意返回值：

```
fn grow(&self) -> Circle {
```

我们看到我们返回了一个 `Circle`。通过这个函数，我们可以增长一个圆的面积100倍。

## 静态方法

我们也可以定义一个不带 `self` 参数的方法。这是一个Rust代码中非常常见的模式：

```
struct Circle {
 x: f64,
 y: f64,
 radius: f64,
}

impl Circle {
 fn new(x: f64, y: f64, radius: f64) -> Circle {
 Circle {
 x: x,
 y: y,
 radius: radius,
 }
 }
}

fn main() {
 let c = Circle::new(0.0, 0.0, 2.0);
}
```

这个静态函数 (*static method*) 为我们构建了一个新的 `Circle`。注意静态函数是通过 `Struct::method()` 语法调用的，而不是 `ref.method()` 语法。

## 创建者模式 (Builder Pattern)

我们说我们需要我们的用户可以创建圆，不过我们只允许他们设置他们关心的属性。否则，`x` 和 `y` 将是 `0.0`，并且 `radius` 将是 `1.0`。Rust并没有方法重载，命名参数或者可变参数。我们利用创建者模式来代替。它看起像这样：

```
struct Circle {
 x: f64,
 y: f64,
 radius: f64,
}

impl Circle {
 fn area(&self) -> f64 {
 std::f64::consts::PI * (self.radius * self.radius)
 }
}

struct CircleBuilder {
```

```

 coordinate: f64,
 radius: f64,
 }

impl CircleBuilder {
 fn new() -> CircleBuilder {
 CircleBuilder { coordinate: 0.0, radius: 0.0, }
 }

 fn coordinate(&mut self, coordinate: f64) -> &mut CircleBuilder {
 self.coordinate = coordinate;
 self
 }

 fn radius(&mut self, radius: f64) -> &mut CircleBuilder {
 self.radius = radius;
 self
 }

 fn finalize(&self) -> Circle {
 Circle { x: self.coordinate, y: self.coordinate, radius: self.radius }
 }
}

fn main() {
 let c = CircleBuilder::new()
 .coordinate(10.0)
 .radius(5.0)
 .finalize();

 println!("area: {}", c.area());
}

```

我们在这里又声明了一个结构体，`CircleBuilder`。我们给它定义了一个创建者函数。我们也在`Circle`中定义了`area()`方法。我们还定义了另一个方法`circleBuilder: finalize()`。这个方法从构造器中创建了我们最后的`Circle`。现在我们使用类型系统来强化我们的考虑：我们可以用`circleBuilder`来强制生成我们需要的`Circle`。

## 关联类型

关联类型是Rust类型系统中十分强力的一部分。它涉及到'类型族'的概念，换句话说，就是把多种类型归于一类。这个描述可能比较抽象，所以让我们深入研究一个例子。如果你想编写一个 `Graph` 特性，你需要泛型化两个类型：点类型和边类型。所以你可能会像这样写一个特性，`Graph<N, E>`：

```
trait Graph<N, E> {
 fn has_edge(&self, &N, &N) -> bool;
 fn edges(&self, &N) -> Vec<E>;
 // etc
}
```

虽然这可以工作，不过显得很尴尬，例如，任何需要一个 `Graph` 作为参数的函数都需要泛型化的 `N'ode` 和 `E'dge` 类型：

```
fn distance<N, E, G: Graph<N, E>>(graph: &G, start: &N, end: &N) -> u32 { ... }
```

我们的距离计算并不需要 `Edge` 类型，所以函数签名中 `E` 只是写着玩的。

我们需要的是对于每一种 `Graph` 类型，都使用一个特定的的 `N'ode` 和 `E'dge` 类型。我们可以用关联类型来做到这一点：

```
trait Graph {
 type N;
 type E;

 fn has_edge(&self, &Self::N, &Self::N) -> bool;
 fn edges(&self, &Self::N) -> Vec<Self::E>;
 // etc
}
```

现在，我们使用一个抽象的 `Graph` 了：

```
fn distance<G: Graph>(graph: &G, start: &G::N, end: &G::N) -> uint { ... }
```

这里不再需要处理`E'dge`类型了。

让我们更详细的回顾一下。

## 定义关联类型

让我们构建一个 `Graph` 特性。这里是定义：

```
trait Graph {
```

```

type N;
type E;

fn has_edge(&self, &Self::N, &Self::N) -> bool;
fn edges(&self, &Self::N) -> Vec<Self::E>;
}

```

十分简单。关联类型使用 `type` 关键字，并出现在特性体和函数中。

这些 `type` 声明跟函数定义一样。例如，如果我们想 `N` 类型实现 `Display`，这样我们就可以打印出点类型，我们可以这样写：

```

use std::fmt;

trait Graph {
 type N: fmt::Display;
 type E;

 fn has_edge(&self, &Self::N, &Self::N) -> bool;
 fn edges(&self, &Self::N) -> Vec<Self::E>;
}

```

## 实现关联类型

就像任何特性，使用关联类型的特性用 `impl` 关键字来提供实现。下面是一个 `Graph` 的简单实现：

```

struct Node;

struct Edge;

struct MyGraph;

impl Graph for MyGraph {
 type N = Node;
 type E = Edge;

 fn has_edge(&self, n1: &Node, n2: &Node) -> bool {
 true
 }

 fn edges(&self, n: &Node) -> Vec<Edge> {
 Vec::new()
 }
}

```

这个可笑的实现总是返回 `true` 和一个空的 `Vec<Edge>`，不过它提供了如何实现这类特性的思路。首先我们需要3个 `struct`，一个代表图，一个代表点，还有一个代表边。如果使用别的类型更合理，也可以那样做，我们只是准备使用 `struct` 来代表这3个类型。

接下来是 `impl` 行，它就像其它任何特性的实现。

在这里，我们使用 `=` 来定义我们的关联类型。特性使用的名字出现在 `=` 的左边，而我们 `impl` 的具体类型出现在右边。最后，我们在函数声明中使用具体类型。

## 特性对象和关联类型

这里还有另外一个我们需要讨论的语法：特性对象。如果你创建一个关联类型的特性对象，像这样：

```
let graph = MyGraph;
let obj = Box::new(graph) as Box<Graph>;
```

你会得到两个错误：

```
error: the value of the associated type `E` (from the trait `main::Graph`) must
be specified [E0191]
let obj = Box::new(graph) as Box<Graph>;
^-----
24:44 error: the value of the associated type `N` (from the trait
`main::Graph`) must be specified [E0191]
let obj = Box::new(graph) as Box<Graph>;
^-----
```

我们不能这样创建一个特性对象，因为我们并不知道关联的类型。相反，我们可以这样写：

```
let graph = MyGraph;
let obj = Box::new(graph) as Box<Graph<N=Node, E=Edge>>;
```

`N=Node` 语法允许我们提供一个具体类型，`Node`，作为 `N` 类型参数。`E=Edge` 也是一样。如果我们不提供这个限制，我们不能确定应该 `impl` 那个来匹配特性对象。

## 闭包

到目前为止，我们已经写了不少Rust函数了，不过它们都有名字。Rust也允许我们创建匿名函数。Rust的匿名函数叫做闭包（closures）。闭包自身并不很有意思，不过当它们与用闭包作为参数的函数结合时，就可能非常强大了。

让我们写一个闭包：

```
let add_one = |x| { 1 + x };
println!("The sum of 5 plus 1 is {}", add_one(5));
```

我们用`|...| { ... }`语法来创建一个闭包，然后我们创建一个绑定方便我们后面使用它。注意我们调用闭包时使用绑定的名字和括号，就像我么调用命名函数一样。

让我们比较一下语法。这两个非常相似：

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

你可能注意到了，闭包会推断它参数和返回值的类型，所以你不需要声明它们。这与命名函数有所不同，它们默认返回单元`(())`。

闭包和命名函数有一个巨大的区别，就在于1它们的名字：闭包“闭合了它的环境”。这是什么意思呢？它意味着：

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

 let printer = || { println!("x is: {}", x); };
 printer(); // prints "x is: 5"
}
```

`||`语法代表这个匿名闭包不带参数。没有它，我们就一块`{}`中的代码了。

换句话说，闭包可以访问定义它的作用域内的变量。闭包借用任何它使用的变量，所以下面是错误的：

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

 let printer = || { println!("x is: {}", x); };

 x = 6; // error: cannot assign to `x` because it is borrowed
}
```

## 移动闭包 (Moving closures)

Rust第二种类型的闭包，叫做移动闭包 (*moving closure*)，移动闭包用 `move` 关键字来标明。移动闭包与正常闭包的区别是移动闭包总是获取它使用变量的所有权。正常闭包，相反，只是在自己所在的栈上创建一个引用。移动闭包在Rust的并发功能中最有用，所以我们现在先不讲它。我们会在“线程”部分更详细的介绍它。

### 闭包作为参数

闭包作为另一个函数的参数时最有用。这是个例子：

```
fn twice<F: Fn(i32) -> i32>(x: i32, f: F) -> i32 {
 f(x) + f(x)
}

fn main() {
 let square = |x: i32| { x * x };

 twice(5, square); // evaluates to 50
}
```

让我们把例子分开来看，从 `main` 开始：

```
let square = |x: i32| { x * x };
```

我们之前见过这个。我们创建了一个闭包获取一个整形，然后返回它的平方。

```
twice(5, square); // evaluates to 50
```

这一行更有意思。这里，我们调用函数，`twice`，我们传递了两个参数：一个整形，`5`，然后是我们的闭包，`square`。这跟向函数传递两个变量绑定并无区别，不过如果你之前并未接触过闭包，这看起来可能有点复杂。只需要想“我们传递了两个参数：一个是 `i32`，一个是函数”就行。

下面，让我们看看 `twice` 是如何定义的：

```
fn twice<F: Fn(i32) -> i32>(x: i32, f: F) -> i32 {
```

`twice` 有两个参数，`x` 和 `f`。这是它为什么我们调用它时传递两个参数。`x` 是一个 `i32`，我们已经很熟悉了。`f` 是一个函数，不过它需要一个 `i32` 并返回一个 `i32`。这是 `Fn(i32) -> i32` 类型的 `F` 的需要。现在 `F` 代表任何需要一个 `i32` 和返回一个 `i32` 的函数。

这可以是我们见过最复杂的函数标识了！多看几次直到你知道怎么用它了为止。这需要一些小的实践，然后就简单了。好消息是这样传递闭包时是非常有效率的。在编译时拥有所有这些类型信息的话编译器可以做非常神奇的事。

最后，`twice` 也返回一个 `i32`。

让我们看看 `twice` 函数体：

```
fn twice<F: Fn(i32) -> i32>(x: i32, f: F) -> i32 {
 f(x) + f(x)
}
```

因为我们的闭包叫做 `f`，我们可以像之前调用闭包那样调用它们，然后我们给它们每个传递一个 `x`，因为我们的函数叫做 `twice`。

如果你计算一下，`(5 * 5) + (5 * 5) == 50`，这就是我们得到的输出。

多要要这些内容直到你熟悉它们。Rust的标准库在合适的地方大量使用了闭包，所以你也应该多使用这个技术。

如果我们不想给 `square` 一个名字，我们也可以内联的定义它。这个例子跟上面是一样的：

```
fn twice<F: Fn(i32) -> i32>(x: i32, f: F) -> i32 {
 f(x) + f(x)
}

fn main() {
 twice(5, |x: i32| { x * x }); // evaluates to 50
}
```

你可以在任何用闭包的地方使用一个命名函数。上面例子的另一种写法：

```
fn twice<F: Fn(i32) -> i32>(x: i32, f: F) -> i32 {
 f(x) + f(x)
}

fn square(x: i32) -> i32 { x * x }

fn main() {
 twice(5, square); // evaluates to 50
}
```

这种写法并不是特别常见，不过偶尔也是有用的。

在我们继续之前，让我们看看接收两个闭包的函数。

```
fn compose<F, G>(x: i32, f: F, g: G) -> i32
 where F: Fn(i32) -> i32, G: Fn(i32) -> i32 {
 g(f(x))
}

fn main() {
 compose(5,
 |n: i32| { n + 42 },

```

```
|n: i32| { n * 2 }); // evaluates to 94
}
```

你可用会问了：为什么我们要引入两个类型变量 `F` 和 `G`？很明显，`f` 和 `g` 有相同的标记：`Fn(i32) -> i32`。

这是因为在Rust中每个闭包都有自己独特的类型。所以，不仅不同标记的闭包有不同的类型，相同标记的不同闭包也有不同的类型。

你可以这样理解：闭包的行为是它类型的一部分。因此，对两个闭包使用一个类型标记将会导致只接受第一个闭包，拒绝第二个。因为不同类型的第二个闭包不允许它表现为与第一个类型参数一样的类型。我们承认这个问题，所以使用了 `F` 和 `G` 两个类型参数。

这也引入了 `where` 分句，它允许我们更灵活的描述类型参数。

这就是你了解闭包所需要的一切！闭包最开始有点奇怪，不过一旦你熟悉它了，你会在其它语言中想念它的。向其它函数传递函数异常强大，如你将在接下来关于迭代器这一章将看到的。

## 迭代器

让我们讨论一下循环。

还记得Rust的 `for` 循环吗？这是一个例子：

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

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

就像这样：

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

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

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

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

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

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

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

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

这严格的说比使用现成的迭代器还要糟。你可以直接在向量上迭代。所以这么写：

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

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

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

这里还有一个细节不是100%清楚的就是 `println!` 是如何工作的。`num` 是 `&i32` 类型。也就是说，它是一个 `i32` 的引用，并不是 `i32` 本身。`println!` 为我们处理了非关联化，所以我们看不到。下面的代码也能工作：

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

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

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

那么，现在我们已经明确了范围经产不是我们需要的，让我们来讨论下你需要什么。

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

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

让我们先看看消费者，因为我们已经见过范围这个迭代器了。

## 消费者

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

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

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

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

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

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

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

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

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

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

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

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

另一个重要的消费者是`fold`。他看起来像这样：

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

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

好吧，这有点混乱。让我们检查一下这个迭代器中所有这些值：

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

我们可以使用这些参数调用`fold()`：

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

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

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

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

## 迭代器

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

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

因为我们没有用范围做任何事，它并生成序列。让我们加上消费者：

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

现在，`collect()` 会要求范围生成一些值，接着它会开始产生序列。

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

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

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

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

```
std::iter::count(1, 5);
```

这个迭代器从1开始计数，每次加5.它每次都会给你一个新值，直到永远（好吧，从技术上讲直到它循环到 `i32` 所能代表的最大值）。不过因为它是懒惰的，这没有问题！你可能不会想在它之上使用 `collect()`。

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

## 迭代适配器 (Iterator adapters)

迭代适配器 (*Iterator adapters*) 获取一个迭代器然后按某种方法修改它，并产生一个新的迭代器。最简单的是一个 `map`：

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

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

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

又是惰性！那个闭包永远也不会执行。这个例子也不会打印任何数字：

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

如果你尝试在一个迭代器上执行带有副作用的闭包，不如直接使用 `for`。

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

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

这会打印：

```
1
6
11
16
21
```

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

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

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

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

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

这会给你一个包含 6 , 12 , 18 , 24 和 30 的向量。

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

## 泛型

有时，当你编写函数或数据类型时，我们可能会希望它能处理多种类型的参数。例如，还记得我们的 `OptionalInt` 类型吗？

```
enum OptionalInt {
 Value(i32),
 Missing,
}
```

如果你也想要一个 `OptionalFloat64`，我们需要一个新的枚举：

```
enum OptionalFloat64 {
 Valuef64(f64),
 Missingf64,
}
```

这真是非常可惜。幸运的是，Rust有一个能给我们更好选择的功能：泛型。泛型在类型理论中叫做参数多态 (*parametric polymorphism*)，它意味着他们是对给定参数 (*parametric*) 能够有多种形式 (*poly* 是多，*morph* 是形态) 的函数或类型。

不管怎么说，类型理论声明已经足够了，让我们看看泛型版本的 `OptionalInt`。事实上Rust自身已经提供了它，而且它看起来像这样：

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

`<T>` 部分，你可能见过几次了，代表它是一个泛型数据类型。在我们枚举声明中，每当我们看到 `T`，我们用这个类型代替我们泛型中使用的类型。下面是一个使用 `Option<T>` 的例子，它带有额外的类型标注：

```
let x: Option<i32> = Some(5);
```

在类型声明中，我们看到 `Option<i32>`。注意它与 `option<T>` 的相似之处。所以在这个特定的 `option` 中，`T` 是 `i32`。在绑定的右侧，我们用了 `Some(T)`，`T` 是 `5`。因为那是个 `i32`，两边类型相符，所以皆大欢喜。如果不相符，我们会得到一个错误：

```
let x: Option<f64> = Some(5);
// error: mismatched types: expected `core::option::Option<f64>`,
// found `core::option::Option<_>` (expected f64 but found integral variable)
```

这并不是意味着我们不能写一个 `f64` 的 `Option<T>`！只是类型必须相符：

```
let x: Option<i32> = Some(5);
let y: Option<f64> = Some(5.0f64);
```

这样就好了。一个定义，到处使用。

不一定只有一个类型是泛型的。考虑下Rust内建的 `Result<T, E>` 类型：

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

这里有两个泛型类型：`T` 和 `E`。另外，大写字母可以是任何你喜欢的（大写）字母。我们可以定义 `Result<T, E>` 为：

```
enum Result<A, Z> {
 Ok(A),
 Err(Z),
}
```

如果你想这么做的话。惯例告诉我们第一个泛型参数应该是 `T`，代表 `type`，然后我们用 `E` 来代表 `error`。然而，Rust并不管这些。

`Result<T, E>` 意图作为计算的返回值，并为了能够在不能工作时返回一个错误。这是一个例子：

```
let x: Result<f64, String> = Ok(2.3f64);
let y: Result<f64, String> = Err("There was an error.".to_string());
```

这个特定的 `Result` 在我们成功时返回一个 `f64`，失败时返回一个 `String`。让我们写一个使用 `Result<T, E>` 的函数：

```
fn inverse(x: f64) -> Result<f64, String> {
 if x == 0.0f64 { return Err("x cannot be zero!".to_string()); }

 Ok(1.0f64 / x)
}
```

我们不想要0的倒数，所以我们进行了检查以确保我们不会传递一个0.如果我用了0，那么我们返回一个带有信息的 `Err`。如果不是，返回一个结果。

为什么这重要呢？好吧，还记得 `match` 如何进行穷尽性匹配的吗？这里是那个函数如何使用的：

```
let x = inverse(25.0f64);

match x {
 Ok(x) => println!("The inverse of 25 is {}", x),
```

```
 Err(msg) => println!("Error: {}", msg),
}
```

`match` 强制我们处理 `Err` 分支。另外，因为结果包含在 `Ok` 中，我们不能在没有 `match` 的情况下使用它的结果：

```
let x = inverse(25.0f64);
println!("{}", x + 2.0f64); // error: binary operation `+` cannot be applied
// to type `core::result::Result<f64, collections::string::String>`
```

这个函数很棒，不过还有另一个问题：它只能作用于64位浮点数。如果我们也想处理32位浮点数呢？好吧，我们得这么写：

```
fn inverse32(x: f32) -> Result<f32, String> {
 if x == 0.0f32 { return Err("x cannot be zero!".to_string()); }

 Ok(1.0f32 / x)
}
```

真倒霉。我们需要的是一个泛型函数。幸运的是，我们可以写一个！然而现在它还不能很好的工作。在我们开始之前，让我们看看语法。一个泛型版本的 `inverse` 看起来像这样：

```
fn inverse<T>(x: T) -> Result<T, String> {
 if x == 0.0 { return Err("x cannot be zero!".to_string()); }

 Ok(1.0 / x)
}
```

就像我们的 `option<T>` 一样，我们在 `inverse<T>` 中使用了相似的语法。我们现在可以在剩下的标记中使用 `T` 了：`x` 是 `T` 型的，然后 `Result` 一半是 `T`` 型的。然而，如果我们尝试编译这个例子，我们会得到一个错误：

```
error: binary operation `==` cannot be applied to type `T`
```

因为 `T` 可以是任何类型，它可能是一个并没有实现 `==` 的类型，因此，第一行可能出错。我们应该怎么做？

为了修改这个问题，我们需要学习Rust的另一个功能：特性。

## 特性

你还记得 `impl` 关键字吗，曾用方法语法调用方法的那个？

```
struct Circle {
 x: f64,
 y: f64,
 radius: f64,
}

impl Circle {
 fn area(&self) -> f64 {
 std::f64::consts::PI * (self.radius * self.radius)
 }
}
```

特性也很类似，除了我们用函数标记来定义一个特性，然后为结构体实现特性。例如：

```
struct Circle {
 x: f64,
 y: f64,
 radius: f64,
}

trait HasArea {
 fn area(&self) -> f64;
}

impl HasArea for Circle {
 fn area(&self) -> f64 {
 std::f64::consts::PI * (self.radius * self.radius)
 }
}
```

如你所见，`trait` 块与 `impl` 看起来很像，不过我们没有定义一个函数体，只是函数标记。当我们 `impl` 一个特性时，我们使用 `impl Trait for Item`，而不是仅仅 `impl Item`。

那么只有什么重要的呢？还记得我们使用泛型 `inverse` 函数得到的错误吗？

```
error: binary operation `==` cannot be applied to type `T`
```

我们可以用特性来约束我们的泛型。考虑下这个函数，它不能编译并给出一个类似的错误：

```
fn print_area<T>(shape: T) {
 println!("This shape has an area of {}", shape.area());
}
```

Rust抱怨说：

```
fn print_area<T>(shape: T) {
 println!("This shape has an area of {}", shape.area());
}
```

因为 `T` 可以是任何类型，我们不能确定它实现了 `area` 方法。不过我们可以在泛型 `T` 添加一个特性约束 (*trait constraint*)，来确保它实现了对应方法：

```
fn print_area<T: HasArea>(shape: T) {
 println!("This shape has an area of {}", shape.area());
}
```

`<T: HasArea>` 是指 any type that implements the `HasArea` trait (任何实现了 `HasArea` 特性的类型)。因为特性定义了函数类型标记，我们可以确定任何实现 `HasArea` 将会拥有一个 `.area()` 方法。

这是一个扩展的例子演示它如何工作：

```
trait HasArea {
 fn area(&self) -> f64;
}

struct Circle {
 x: f64,
 y: f64,
 radius: f64,
}

impl HasArea for Circle {
 fn area(&self) -> f64 {
 std::f64::consts::PI * (self.radius * self.radius)
 }
}

struct Square {
 x: f64,
 y: f64,
 side: f64,
}

impl HasArea for Square {
 fn area(&self) -> f64 {
 self.side * self.side
 }
}

fn print_area<T: HasArea>(shape: T) {
 println!("This shape has an area of {}", shape.area());
}

fn main() {
 let c = Circle {
 x: 0.0f64,
```

```

 y: 0.0f64,
 radius: 1.0f64,
 };

 let s = Square {
 x: 0.0f64,
 y: 0.0f64,
 side: 1.0f64,
 };

 print_area(c);
 print_area(s);
}

```

这个程序会输出：

```

This shape has an area of 3.141593
This shape has an area of 1

```

如你所见，`print_area` 现在是泛型的了，并且确保我们传递了正确的类型。如果我们传递了错误的类型：

```
print_area(5);
```

我们会得到一个编译时错误：

```
error: failed to find an implementation of trait main::HasArea for int
```

目前为止，我们只在结构体上添加特性实现，不过你为任何类型实现一个特性。所以技术上讲，你可以在 `i32` 上实现 `HasArea`：

```

trait HasArea {
 fn area(&self) -> f64;
}

impl HasArea for i32 {
 fn area(&self) -> f64 {
 println!("this is silly");

 *self as f64
 }
}

5.area();

```

在基本类型上实现方法被认为是不好的设计，即便这是可以的。

这看起来有点像狂野西部，不过这还有两个限制来避免情况失去控制。第一，特性必须在你想要使用它的作用域中被 `use`。所以例如，下面的代码不能工作：

```

mod shapes {
 use std::f64::consts;

 trait HasArea {
 fn area(&self) -> f64;
 }

 struct Circle {
 x: f64,
 y: f64,
 radius: f64,
 }

 impl HasArea for Circle {
 fn area(&self) -> f64 {
 consts::PI * (self.radius * self.radius)
 }
 }
}

fn main() {
 let c = shapes::Circle {
 x: 0.0f64,
 y: 0.0f64,
 radius: 1.0f64,
 };

 println!("{}", c.area());
}

```

现在我们把结构体和特性都放到我们自己的模块中了，我们得到一个错误

```
error: type `shapes::Circle` does not implement any method in scope named `area`
```

如果你在 `main` 函数之前使用一个 `use` 行并把相应的结构公有化，则一切正常：

```

use shapes::HasArea;

mod shapes {
 use std::f64::consts;

 pub trait HasArea {
 fn area(&self) -> f64;
 }

 pub struct Circle {
 pub x: f64,
 pub y: f64,
 pub radius: f64,
 }

 impl HasArea for Circle {
 fn area(&self) -> f64 {
 consts::PI * (self.radius * self.radius)
 }
 }
}

```

```

 }
 }
}

fn main() {
 let c = shapes::Circle {
 x: 0.0f64,
 y: 0.0f64,
 radius: 1.0f64,
 };

 println!("{}" , c.area());
}

```

这意味着即使有人做了像给 `int` 增加函数这种坏事，它也不会影响你，除非你 `use` 了那个特性。

这还有一个实现特性的限制。不管是特性还是你写的 `impl` 都只能在你自己的包装箱内生效。所以，我们可以为 `i32` 实现 `HasArea` 特性，因为 `HasArea` 在我们的包装箱中。不过如果我们想为 `i32` 实现 `Float` 特性，它是由Rust提供的，则无法做到，因为这个特性和类型都不在我们的包装箱中。

关于特性的最后一点：带有特性限制的泛型函数是单态（*monomorphization*）（mono：单一，morph：形式）的，所以它是静态分发（*statically dispatched*）的。这是神马意思？查看[静态和动态分发](#)来了解更多细节。

## where从句（Where clause）

编写只有少量泛型和特性的函数并不算太糟，不过当它们的数量增加，这个语法就看起来比较诡异了：

```

use std::fmt::Debug;

fn foo<T: Clone, K: Clone + Debug>(x: T, y: K) {
 x.clone();
 y.clone();
 println!("{:?}" , y);
}

```

函数的名字在最左边，而参数列表在最右边。限制写在中间。

Rust有一个解决方案，它叫“where从句”：

```

use std::fmt::Debug;

fn foo<T: Clone, K: Clone + Debug>(x: T, y: K) {
 x.clone();
 y.clone();
 println!("{:?}" , y);
}

fn bar<T, K>(x: T, y: K) where T: Clone, K: Clone + Debug {
 x.clone();
 y.clone();
}

```

```

 println!("{:?}", y);
}

fn main() {
 foo("Hello", "world");
 bar("Hello", "world");
}

```

`foo()` 使用我们刚才的语法，而 `bar()` 使用 `where` 从句。所有你所需要做的就是在定义参数时省略限制，然后在参数列表后加上一个 `where`。对于很长的列表，你也可以加上空格：

```

use std::fmt::Debug;

fn bar<T, K>(x: T, y: K)
 where T: Clone,
 K: Clone + Debug {
 x.clone();
 y.clone();
 println!("{:?}", y);
}

```

这种灵活性可以是复杂情况变得简洁。

`where` 也比基本语法更强大。例如：

```

trait ConvertTo<Output> {
 fn convert(&self) -> Output;
}

impl ConvertTo<i64> for i32 {
 fn convert(&self) -> i64 { *self as i64 }
}

// can be called with T == i32
fn normal<T: ConvertTo<i64>>(x: &T) -> i64 {
 x.convert()
}

// can be called with T == i64
fn inverse<T>() -> T
 // this is using ConvertTo as if it were "ConvertFrom<i32>"
 where i32: ConvertTo<T> {
 i32.convert()
}

```

这突显出了 `where` 从句的额外特性：它允许限制的左侧可以是任意类型（在这里是 `i32`），而不仅仅是一个类型参数（比如 `T`）。

## 我们的`inverse`例子

回到泛型，我们曾尝试编写这样的代码：

```
fn inverse<T>(x: T) -> Result<T, String> {
 if x == 0.0 { return Err("x cannot be zero!".to_string()); }

 Ok(1.0 / x)
}
```

如果我们尝试编译它，会得到这个错误：

```
error: binary operation `==` cannot be applied to type `T`
```

这是因为 `T` 太宽泛了：我们不知道一个随机的 `T` 是否能够比较。为此，我们可以使用特性约束。它还不能很好的工作，不过试试这个：

```
fn inverse<T: PartialEq>(x: T) -> Result<T, String> {
 if x == 0.0 { return Err("x cannot be zero!".to_string()); }

 Ok(1.0 / x)
}
```

你应该会得到这个错误：

```
error: mismatched types:
expected `T`,
 found `_
(expected type parameter,
 found floating-point variable)
```

所以它不能工作。因为 `T` 是 `PartialEq` 的，我们期望有另一个 `T`，然而相反，我们找到一个浮点变量。我们需要一个不同的限制。`Float` 就可以：

```
use std::num::Float;

fn inverse<T: Float>(x: T) -> Result<T, String> {
 if x == Float::zero() { return Err("x cannot be zero!".to_string()); }

 let one: T = Float::one();
 Ok(one / x)
}
```

我们必须把我们泛型的 `0.0` 和 `1.0` 替换为 `Float` 特性中合适的方法。`f32` 和 `f64` 都实现了 `Float`，所以我们的程序可以正常工作：

```
println!("the inverse of {} is {:?}", 2.0f32, inverse(2.0f32));
println!("the inverse of {} is {:?}", 2.0f64, inverse(2.0f64));
```

```
println!("the inverse of {} is {:?}", 0.0f32, inverse(0.0f32));
println!("the inverse of {} is {:?}", 0.0f64, inverse(0.0f64));
```

## 默认方法 (Default methods)

关于特性还有最后一个我们需要讲到的特性。它简单到我们只需展示一个例子：

```
trait Foo {
 fn bar(&self);

 fn baz(&self) { println!("We called baz."); }
}
```

Foo 特性需要实现 `bar()`，不过并不需要实现 `baz()`。它会使用默认的行为。如果你想的话也能覆盖默认行为：

```
struct UseDefault;

impl Foo for UseDefault {
 fn bar(&self) { println!("We called bar."); }
}

struct OverrideDefault;

impl Foo for OverrideDefault {
 fn bar(&self) { println!("We called bar."); }

 fn baz(&self) { println!("Override baz!"); }
}

let default = UseDefault;
default.baz(); // prints "We called bar."

let over = OverrideDefault;
over.baz(); // prints "Override baz!"
```

## 静态和动态分发

当涉及到多态的代码时，我们需要一个机制来决定哪个具体的版本应该得到执行。这叫做“分发”。大体上有两种形式的分发：静态分发和动态分发。虽然Rust喜欢静态分发，不过它也提供了一个叫做“特性对象”的机制来支持动态分发。

## 背景

在本章接下来的内容中，我们需要一个特性和一些实现。让我们来创建一个简单的 `Foo`。它有一个返回一个 `String` 的方法。

```
trait Foo {
 fn method(&self) -> String;
}
```

我们也在 `u8` 和 `String` 上实现了这个特性：

```
impl Foo for u8 {
 fn method(&self) -> String { format!("u8: {}", *self) }
}

impl Foo for String {
 fn method(&self) -> String { format!("string: {}", *self) }
}
```

## 静态分发

我们可以使用特性限制来进行静态分发：

```
fn do_something<T: Foo>(x: T) {
 x.method();
}

fn main() {
 let x = 5u8;
 let y = "Hello".to_string();

 do_something(x);
 do_something(y);
}
```

在这里Rust用“单态”来进行静态分发。这意味着Rust会为 `u8` 和 `String` 分别创建一个特殊版本的 `do_something()`，然后将调用替换为这些特殊函数。也就是说，Rust生成了一些像这样的函数：

```
fn do_something_u8(x: u8) {
```

```

 x.method();
}

fn do_something_string(x: String) {
 x.method();
}

fn main() {
 let x = 5u8;
 let y = "Hello".to_string();

 do_something_u8(x);
 do_something_string(y);
}

```

这有一个很牛的好处：静态分发允许函数被内联调用，因为调用者在编译时就知道它，并且内联是好的优化的关键。静态分发是很快的，不过也有它的权衡之处：“代码膨胀”，因为对于每个类型都会有多份同样函数的拷贝存在。

此外，编译器也不是完美的并且“优化”后的代码可能更慢。例如，过度的函数内联会是指令缓存膨胀（缓存控制着我们周围的一切）。这也是为什么要谨慎使用 `#[inline]` 和 `#[inline(always)]` 的部分原因，另外一个使用动态分发的原因是有时它更有效率。

然而，常规情况下静态分发更有效率，并且我们总是可以写一个静态分发的封装函数来进行动态分发，不过反过来不行，这就是说静态调用更加灵活。因为这个原因标准库尽可能的使用了静态分发。

## 动态分发

Rust通过一个叫做“特性对象”的功能提供动态分发。特性对象，就像 `&Foo` 或 `Box<Foo>`，是一些储存了实现了给定特性的任意类型的一个值的对象，它的具体类型只能在运行时才能确定。特性的方法可以通过一个特殊的函数指针的记录（由编译器创建和管理）在特性对象上调用。

特性对象可以从一个通过转换（例如，`&x as &Foo`）或者强制多态（例如，从函数的参数 `&Foo` 取得 `&x`）实现了该特性的指针中取得。

这些特性对象的强制多态和转换也适用于像从 `&mut T` 到 `&mut Foo` 和从 `Box<T>` 到 `Box<Foo>` 这样的指针，不过这也是目前的全部情况。强制多态和转换是一样的。

这个操作可以被看作“清除”编译器关于特定类型指针的信息，因此特性对象有时被称为“类型清除”。

回到上面的例子，我们可以使用相同的特性进行动态分发转换特性对象：

```

fn do_something(x: &Foo) {
 x.method();
}

fn main() {
 let x = 5u8;
 do_something(&x as &Foo);
}

```

或者通过强制多态：

```
fn do_something(x: &Foo) {
 x.method();
}

fn main() {
 let x = "Hello".to_string();
 do_something(&x);
}
```

一个使用特性对象的函数并没有为每个实现了 `Foo` 的类型专门生成函数：它生成了一份拷贝，一般（但不总是）会减少代码膨胀。然而，这回带来使用更慢的虚函数调用的开销，也会有效的阻止任何内联和相关的优化的进行。

## 为什么用指针？

Rust默认不用指针来存放数据，不想很多托管语言，所以类型可以有不同的大小。在编译时知道值的大小对像作为参数传递给函数，在栈上移动和在堆上分配（或释放）并储存等情况是很重要的。

对于 `Foo`，我们需要一个值至少是一个 `String`（24字节）或一个 `u8`（1字节），或者其它任何相关包装箱中可能实现了 `Foo`（任意字节）的类型。我们无法保证最后这一点如果值没有使用指针储存，因为其它类型可以是任意大的。

用指针来储存值意味着当我们使用特性对象时值的大小是无关的，只关系到指针自己的大小。

## 表现（Representation）

可以在一个特性对象上通过一个特殊的函数指针的记录调用的特性函数通常叫做“虚函数表”（通过编译器创建和管理）。

特性对象既简单又复杂：它的核心表现和设计是十分直观的，不过这有一些难懂的错误信息和诡异行为有待发掘。

让我们从一个简单的，带有特性对象的运行时表现开始。`std::raw` 模块包含与复杂的内建类型有相同结构的结构体，[包括特性对象](#)：

```
pub struct TraitObject {
 pub data: *mut(),
 pub vtable: *mut(),
}
```

这就是了，一个特性对象就像包含一个“数据”指针和“虚函数表”指针的 `&Foo`。

数据指针指向特性对象保存的数据（某个未知的类型 `T`），和一个虚表指针指向对应 `T` 的 `Foo` 实现的虚函数表。

一个虚表本质上是一个函数指针的结构体，指向每个函数实现的具体机器码。一个

像 `trait_object.method()` 的函数调用会从虚表中取出正确的指针然后进行一个动态调用。例如：

```

struct FooVtable {
 destructor: fn(*mut ()),
 size: usize,
 align: usize,
 method: fn(*const ()) -> String,
}

// u8:

fn call_method_on_u8(x: *const ()) -> String {
 // the compiler guarantees that this function is only called
 // with `x` pointing to a u8
 let byte: &u8 = unsafe { &*(x as *const u8) };

 byte.method()
}

static Foo_for_u8_vtable: FooVtable = FooVtable {
 destructor: /* compiler magic */,
 size: 1,
 align: 1,

 // cast to a function pointer
 method: call_method_on_u8 as fn(*const ()) -> String,
};

// String:

fn call_method_on_String(x: *const ()) -> String {
 // the compiler guarantees that this function is only called
 // with `x` pointing to a String
 let string: &String = unsafe { &*(x as *const String) };

 string.method()
}

static Foo_for_String_vtable: FooVtable = FooVtable {
 destructor: /* compiler magic */,
 // values for a 64-bit computer, halve them for 32-bit ones
 size: 24,
 align: 8,

 method: call_method_on_String as fn(*const ()) -> String,
};

```

在每个虚表中的 `destructor` 字段指向一个会清理虚表类型的任何资源的函数，对于 `u8` 是普通的，不过对于 `String` 它会释放内存。这对于像 `Box<Foo>` 这类有所有权的特性对象来说是必要的，它需要在离开作用域后清理 `Box` 分配和它内部的类型。`size` 和 `align` 字段储存需要清除类型的大小和它的对齐情况；它们原理上是无用的因为这些信息已经嵌入了析构函数中，不过在将来会被使用到，因为特性对象正日益变得更灵活。

假设我们有一些实现了 `Foo` 的值，那么显式的创建和使用 `Foo` 特性对象可能看起来有点像这个（忽略不匹

配的类型，它们只是指针而已) :

```
let a: String = "foo".to_string();
let x: u8 = 1;

// let b: &Foo = &a;
let b = TraitObject {
 // store the data
 data: &a,
 // store the methods
 vtable: &Foo_for_String_vtable
};

// let y: &Foo = x;
let y = TraitObject {
 // store the data
 data: &x,
 // store the methods
 vtable: &Foo_for_u8_vtable
};

// b.method();
(b.vtable.method)(b.data);

// y.method();
(y.vtable.method)(y.data);
```

如果 b 或 y 拥有特性对象 ( `Box<Foo>` ) , 在它们离开作用域后会有一个 `(b.vtable.destructor)` (`b.data`) (相应的还有 y 的) 调用。

# 宏

到目前为止你已经学到了不少Rust提供的抽象和重用代码的工具了。这些代码重用单元有丰富的语义结构。例如，函数有类型标记，类型参数有特性限制并且能重载的函数必须属于一个特定的特性。

这些结构意味着Rust核心抽象拥有强大的编译时正确性检查。不过作为代价的是灵活性的减少。如果你识别出一个重复代码的模式，你会发现把它们解释为泛型函数，特性或者任何Rust语义中的其它结构很难或者很麻烦。

宏允许我们在句法水平上进行抽象。宏是一个“可扩展”句法形式的速记。这个扩展发生在编译的早期，在任何静态检查之前。因此，宏可以实现很多Rust核心抽象不能做到的代码重用模式。

缺点是基于宏的代码更难懂，因为它很少利用Rust的内建规则。就像一个常规函数，一个通用的宏可以在不知道其实现的情况下使用。然而，设计一个通用的宏困难的！另外，在宏中的编译错误更难解释，因为它在扩展代码上描述问题，恶如不是在开发者使用的代码级别。

这些缺点让宏成了所谓“最后求助于的功能”。这并不是说宏的坏话；只是因为它是Rust中需要真正简明，良好抽象的代码的部分。切记权衡取舍。

## 定义一个宏

你可能见过 `vec!` 宏。用来初始化一个任意数量元素的[向量](#)。

```
let x: Vec<u32> = vec![1, 2, 3];
```

这不可能是一个常规函数，因为它可以接受任何数量的参数。不过我们可以想象的到它是这些代码的句法简写：

```
let x: Vec<u32> = {
 let mut temp_vec = Vec::new();
 temp_vec.push(1);
 temp_vec.push(2);
 temp_vec.push(3);
 temp_vec
};
```

我们可以使用宏来实现这么一个简写：

```
macro_rules! vec {
 ($($x:expr),*) => {
 {
 let mut temp_vec = Vec::new();
 $($(
 temp_vec.push($x));
)*
 temp_vec
 }
 }
}
```

```
 };
}
```

哇哦，这里有好多新语法！让我们分开来看。

```
macro_rules! vec { ... }
```

这里我们定义了一个叫做 `vec` 的宏，跟用 `fn vec` 定义一个 `vec` 函数很相似。再罗嗦一句，我们通常写宏的名字时带上一个感叹号，例如 `vec!`。感叹号是调用语法的一部分用来区别宏和常规函数。

## 匹配

宏通过一系列规则定义，它们是模式匹配的分支。上面我们有：

```
($($x:expr),*) => { ... };
```

这就像一个 `match` 表达式分支，不过匹配发生在编译时Rust的语法树中。最后一个分支（这里只有一个分支）的分号是可选的。`=>` 左侧的“模式”叫匹配器（*matcher*）。它有自己的语法。

`$x:expr` 匹配器将会匹配任何Rust表达式，把它的语法树绑定到元变量 `$x` 上。`expr` 标识符是一个片段分类符（*fragment specifier*）。在[宏进阶章节](#)中列举了所有可能的分类符。匹配器写在 `$(...)` 中，`*` 会匹配0个或多个表达式，表达式之间用逗号分隔。

除了特殊的匹配器语法，任何出现在匹配器中的Rust标记必须完全相符。例如：

```
macro_rules! foo {
 (x => $e:expr) => (println!("mode X: {}", $e));
 (y => $e:expr) => (println!("mode Y: {}", $e));
}

fn main() {
 foo!(y => 3);
}
```

将会打印：

```
mode Y: 3
```

而这个：

```
foo!(z => 3);
```

我们会得到编译错误：

```
error: no rules expected the token `z`
```

## 扩展

宏规则的右边是正常的Rust语法，大部分是。不过我们可以拼接一些匹配器中的语法。例如最开始的例子：

```
$(
 temp_vec.push($x);
)*
```

每个匹配的 \$x 表达式都会在宏扩展中产生一个单独 push 语句。扩展中的重复与匹配器中的重复“同步”进行（稍后介绍更多）。

因为 \$x 已经在表达式匹配中声明了，我们并不在右侧重复 :expr。另外，我们并不将用来分隔的逗号作为重复操作的一部分。相反，我们在重复块中使用一个结束用的分号。

另一个细节：vec! 宏的右侧有两对大括号。它们经常像这样结合起来：

```
macro_rules! foo {
 () => {{
 ...
 }}
}
```

外层的大括号是 macro\_rules! 语法的一部分。事实上，你也可以 () 或者 []。它们只是用来界定整个右侧结构的。

内层大括号是扩展语法的一部分。记住，vec! 在表达式上下文中使用。要写一个包含多个语句，包括 let 绑定，的表达式，我们需要使用块。如果你的宏只扩展一个单独的表达式，你不需要内层的大括号。

注意我们从未声明宏产生一个表达式。事实上，直到宏被展开之前我们都无法知道。足够小心的话，你可以编写一个能在多个上下文中扩展的宏。例如，一个数据类型的简写可以作为一个表达式或一个模式。

## 重复

重复运算符遵循两个原则：

1. \$(...) 对它包含的所有 \$name 都执行“一层”重复
2. 每个 \$name 必须有至少这么多的 \$(...) 与其相对。如果多了，它将是多余的。

这个巴洛克宏展示了外层重复中多余的变量。

```
macro_rules! o_0 {
 (
```

```

$(

 $x:expr; [$($y:expr),*]

);*

) => {

 &[$($($x + $y),*),*]

}

}

fn main() {

 let a: &[i32]

 = o_0!(10; [1, 2, 3];

 20; [4, 5, 6]);

 assert_eq!(a, [11, 12, 13, 24, 25, 26]);

}

```

这就是匹配器的大部分语法。这些例子使用了 `$(...)*`，它指“0次或多次”匹配。另外你可以用 `$(...)+` 代表“1次或多次”匹配。每种形式都可以包括一个分隔符，分隔符可以使用任何除了 `+` 和 `*` 的符号。

这个系统基于[Macro-by-Example](#) (PDF链接)。

## 卫生 (Hygiene)

一些语言使用简单的文本替换来实现宏，它导致了很多问题。例如，这个C程序打印 `13` 而不是期望的 `25`。

```

#define FIVE_TIMES(x) 5 * x

int main() {
 printf("%d\n", FIVE_TIMES(2 + 3));
 return 0;
}

```

扩展之后我们得到 `5 * 2 + 3`，并且乘法比加法有更高的优先级。如果你经常使用C的宏，你可能知道标准的习惯来避免这个问题，或更多其它的问题。在Rust中，你不需要担心这个问题。

```

macro_rules! five_times {
 ($x:expr) => (5 * $x);
}

fn main() {
 assert_eq!(25, five_times!(2 + 3));
}

```

元变量 `$x` 被解析成一个单独的表达式节点，并且在替换后依旧在语法树中保持原值。

宏系统中另一个常见的问题是变量捕捉 (*variable capture*)。这里有一个C的宏，使用了[GNU C 扩展](#)来模拟Rust表达式块。

```
#define LOG(msg) ({ \
```

```

int state = get_log_state(); \
if (state > 0) { \
 printf("log(%d): %s\n", state, msg); \
} \
})

```

这是一个非常糟糕的用例：

```

const char *state = "reticulating splines";
LOG(state)

```

它扩展为：

```

const char *state = "reticulating splines";
int state = get_log_state();
if (state > 0) {
 printf("log(%d): %s\n", state, state);
}

```

第二个叫做 `state` 的参数参数被替换为了第一个。当打印语句需要用到这两个参数时会出现问题。

等价的Rust宏则会有理想的表现：

```

macro_rules! log {
 ($msg:expr) => {{
 let state: i32 = get_log_state();
 if state > 0 {
 println!("log({}): {}", state, $msg);
 }
 }};
}

fn main() {
 let state: &str = "reticulating splines";
 log!(state);
}

```

这之所以能工作时因为Rust有一个[卫生宏系统](#)。每个宏扩展都在一个不同的语法上下文 (*syntax context*) 中，并且每个变量在引入的时候都在语法上下文中打了标记。这就好像是 `main` 中的 `state` 和宏中的 `state` 被画成了不同的“颜色”，所以它们不会冲突。

这也限制了宏在被执行时引入新绑定的能力。像这样的代码是不能工作的：

```

macro_rules! foo {
 () => (let x = 3);
}

fn main() {
 foo!();
 println!("{}", x);
}

```

```
}
```

相反你需要在执行时传递变量的名字，这样它会在语法上下文中被正确标记。

```
macro_rules! foo {
 ($v:ident) => (let $v = 3);
}

fn main() {
 foo!(x);
 println!("{}", x);
}
```

这对 `let` 绑定和 `loop` 标记有效，对 `items` 无效。所以下面的代码可以编译：

```
macro_rules! foo {
 () => (fn x() { });
}

fn main() {
 foo!();
 x();
}
```

## 递归宏

一个宏扩展中可以包含更多的宏，包括被扩展的宏自身。这种宏对处理树形结构输入时很有用的，正如这个（简化了的）HTML简写所展示的那样：

```
macro_rules! write_html {
 ($w:expr,) => ((()));

 ($w:expr, $e:tt) => ($write!($w, "{}", $e));

 ($w:expr, $tag:ident [$($inner:tt)*] $($rest:tt)*) => {{
 write!($w, "<{}>", stringify!($tag));
 write_html!($w, $($inner)*);
 write!($w, "</{}>", stringify!($tag));
 write_html!($w, $($rest)*);
 }};
}

fn main() {
 use std::fmt::Write;
 let mut out = String::new();

 write_html!(&mut out,
 html[
 head[title["Macros guide"]]
 body[h1["Macros are the best!"]]
]);
}
```

```
assert_eq!(out,
 "<html><head><title>Macros guide</title></head>\n <body><h1>Macros are the best!</h1></body></html>");
}
```

## 调试宏代码

运行 `rustc --pretty expanded` 来查看宏扩展后的结果。输出表现为一个完整的包装箱，所以你可以把它反馈给 `rustc`，它会有时会比原版产生更好的错误信息。注意如果在同一作用域中有多个相同名字（不过在不同的语法上下文中）的变量的话 `--pretty expanded` 的输出可能会有不同的意义。这种情况下 `--pretty expanded, hygiene` 将会告诉你有关语法上下文的信息。

`rustc` 提供两种语法扩展来帮助调试宏。目前为止，它们是不稳定的并且需要功能入口（feature gates）。

- `log_syntax!(...)` 会打印它的参数到标准输出，在编译时，并且不“扩展”任何东西。
- `trace_macros!(true)` 每当一个宏被扩展时会启用一个编译器信息。在扩展后使用 `trace_macros!(false)` 来关闭它。

## 进一步阅读

[进阶宏章节](#)介绍了更详细的宏语法。它也介绍了如何在不同模块和包装箱中共享宏。

1. 在 `libcollections` 中的 `vec!` 的实际定义与我们在这展示的有所不同，出于效率和可重用性的考虑。这些内容在[进阶宏章节](#)中有涉及。

# 并发

---

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

Rust的内存安全功能也适用于并发环境。甚至并发的Rust程序也会是内存安全的，并且没有数据竞争。Rust的类型系统也能胜任，并给你强大的能力在编译时推论出并发代码。

在我们讨论Rust提供的并发功能之前，理解一些问题是重要的：Rust非常底层以至于所有这些都是由标准库，而不是由语言提供的。这意味着如果你不喜欢一些Rust处理并发的方面，你可以自己实现一个。[mio](#)是现实生活中这个原则的实践。

## 背景：`Send` 和 `Sync`

---

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

### `Send`

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

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

想反，如果我们通过FFI封装了一个不是线程安全的库，我们并不想实现`Send`，并且因此编译器会帮助我们强制它不会离开当前线程。

### `Sync`

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

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

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

## 线程

---

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

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

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

`Thread::scoped()` 方法接受一个闭包，它将会在一个新线程中执行。它叫做 `scoped` 因为这个线程返回一个联合守护（join guard）：

```
use std::thread;

fn main() {
 let guard = thread::scoped(|| {
 println!("Hello from a thread!");
 });

 // guard goes out of scope here
}
```

当 `guard` 离开作用域，它会阻塞执行直到线程结束。如果我们不想要这个行为，我们可以使用 `thread::spawn()`：

```
use std::thread;
use std::old_io::timer;
use std::time::Duration;

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

 timer::sleep(Duration::milliseconds(50));
}
```

我们需要在这里 `sleep` 是因为当 `main()` 结束，它会杀死所有正在执行的线程。

`scoped` 有一个有意思的类型标记：

```
fn scoped<'a, T, F>(<self, f: F>) -> JoinGuard<'a, T>
where T: Send + 'a,
 F: FnOnce() -> T,
 F: Send + 'a
```

特别的，`F`，那个我们传递到新线程执行的闭包。它有两个限制：它必须是一个从 `()` 到 `T` 的 `FnOnce`。`FnOnce` 允许这个闭包从父线程获取任何它提到的数据。另一个限制是 `F` 必须实现 `Send`。我们不能传递这个所有权除非这个类型认为这是可以的。

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

共享数据。

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

根据Rust的类型系统，我们有个听起来很假的概念叫做：“安全的共享可变状态”。很多程序员都同意共享可变状态是灰常，灰常坏的。

有人曾说道：

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

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

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

```
use std::thread;
use std::old_io::timer;
use std::time::Duration;

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

 for i in 0..2 {
 thread::spawn(move || {
 data[i] += 1;
 });
 }

 timer::sleep(Duration::milliseconds(50));
}
```

这会给我们一个错误：

```
12:17 error: capture of moved value: `data`
 data[i] += 1;
 ^~~~
```

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

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

```
use std::thread;
use std::old_io::timer;
```

```

use std::time::Duration;
use std::sync::Mutex;

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

 for i in 0..2 {
 let data = data.lock().unwrap();
 thread::spawn(move || {
 data[i] += 1;
 });
 }

 timer::sleep(Duration::milliseconds(50));
}

```

下面是错误：

```

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

```

你看，[Mutex](#)有一个有如下标记的[lock](#)方法：

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

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

我们可以用 `Arc<T>` 修复这个问题。下面是一个可以工作的版本：

```

use std::sync::{Arc, Mutex};
use std::thread;
use std::old_io::timer;
use std::time::Duration;

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

 for i in 0..2 {
 let data = data.clone();
 thread::spawn(move || {
 let mut data = data.lock().unwrap();
 data[i] += 1;
 });
 }

 timer::sleep(Duration::milliseconds(50));
}

```

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

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

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

然而，定时器部分显得有点奇怪。我们选择等待了一个合理的时间，不过这也完全有可能我们等了太久，我们可以等更少的时间。也有可能我们等了太短，这样我们并没有实际完成计算。

Rust的标准库提供了更多同步两个线程的机制。让我们看看其中一个：通道。

## 通道

下面是我们代码使用通道同步的版本，而不是等待特定时间：

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

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

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

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

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

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

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

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

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

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

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

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

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

 tx.send(answer);
 });
 }

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

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

## 恐慌 (Panics)

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

```
use std::thread;

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

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

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

## 错误处理

The best-laid plans of mice and men Often go awry

"Tae a Moose", Robert Burns

不管是人是鼠,即使最如意的安排设计,结局也往往会出现不意

《致老鼠》,罗伯特·彭斯

有时,杯具就是发生了。有一个计划来应对不可避免会发生的问题是很重要的。Rust提供了丰富的处理你软件中可能(让我们现实点:将会)出现的错误的支持。

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

### 失败 vs. 恐慌

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

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

```
from_str("5");
```

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

```
from_str("hello5world");
```

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

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

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

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

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

```

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

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

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

```

这回给我们一个错误：

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

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

```

use Event::NewRelease;

enum Event {
 NewRelease,
}

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

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

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

```

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

## 使用Option和Result来处理错误

最简单的表明函数会失败的方法是使用 `Option<T>` 类型。还记得我们的 `from_str()` 例子吗？这是它的函数标记：

```
pub fn from_str<A: FromStr>(s: &str) -> Option<A>
```

`from_str()` 返回一个 `Option<A>`。如果转换成功了，会返回 `Some(value)`，如果失败了，返回 `None`。

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

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

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

这是一个使用 `Result` 的例子：

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

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

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

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

```
}
```

这个例子使用了个枚举，`ParseError`，来列举各种可能出现的错误。

## panic!和不可恢复错误

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

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

给出：

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

当你运行它的时候。

因为这种情况相对稀少，保守的使用恐慌。

## 升级失败为恐慌

在特定的情况下，即使一个函数可能失败，我们也想把它当成恐慌。例如，`io::stdin().read_line()` 返回一个 `IoResult<String>`，一种形式的 `Result`（目前只有 `Result` 了，坐等文档更新），当读取行出现错误时。这允许我们处理和尽可能从错误中恢复。

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

```
io::stdin().read_line().unwrap();
```

如果 `Option` 是 `None` 的话 `unwrap()` 会 `panic!`。这基本上就是说“给我一个值，然后如果出错了的话，直接崩溃。”这与匹配错误并尝试恢复相比更不稳定，不过它的处理明显更短小。有时，直接崩溃就行。

这是另一个比 `unwrap()` 稍微聪明点的做法：

```
let input = io::stdin().read_line()
 .ok()
 .expect("Failed to read line");
```

`ok()` 将 `Result` 转换为 `Option`，然后 `expect()` 做了和 `unwrap()` 同样的事，不过带有一个信息。这个信息会传递给底层的 `panic!`，当错误是这样可以提供更好的错误信息。

## 使用try!

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

它可以代替这些：

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

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

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

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

 return Ok(());
}
```

为下面这些代码：

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

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

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

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

 return Ok(());
}
```

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

当前函数提早返回。

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

`try!` 使用[FromError](#)特性来确定错误时应该返回什么。

# 文档

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

## 关于rustdoc

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

文档可以使用两种方法生成：从源代码，或者从单独的Markdown文件。

## 文档化源代码

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

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

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

焰段代捣責生像[焰](<http://doc.rust-lang.org/nightly/std/rc/struct.Rc.html#method.new>)的文档。

文档注用Markdown语法写。

Rust会禁焰些注，并在生成文档 使用它。焰在文档化像数杂焰的机构很重要：

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

上面的代码可以工作，但这个不行：

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

你会得到一个错误：

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

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

编写文档注释

不管怎样，让我们来详细了解一下注释的每一部分：

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

文档注释的第一行应该是他功能的一个简要总结。一句话。只包括基础。高层次。

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

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

特殊部分

```
/// # Examples
```

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

```
/// # Panics
```

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

```
/// # Failures
```

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

```
/// # Safety
```

如果你的函是 `unsafe` 的，你应该解释调用者应该支持哪种不可变量。

```
/// # Examples
///
///
```

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

第三个，`Examples`。包含一个或多个使用你函数的例子，焰 你的用垢会隔此感 (ai) 漏 (shang) 你的。焰些例子²

```
```rust
/// # 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 聊聊焰些代捣億的权柄。

#### 代捣億注  
在注 中 写Rust代捣，使用三重音符：  
```rust
///

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

如果你想要一些不是Rust的代捣，你可以加上一个注：

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

焰回根据你 的宿言高亮代捣。如果你只是想展示普通文本，`text`。

正确的注 是很重要的，因隔`rustdoc`用一种有意思的方法是用它：它可以用来厕璉傾滤你的代捣，焰 你的注

## 文档作隔傾滤  
让我 看看我的例子文档的 例：  
```rust
///

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

```
你会注意到你并不需要`fn main()`或者瓣的什么函数。`rustdoc`会自 一个`main()`包装你的代捣，并且在正确的
```rust
///
```

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

焰回作隔噴濾：

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

这里是完整的算法：

1. 给定的代码块，如果它不包含 `fn main()`，它将会被包装成 `fn main() { your_code }`
2. 给定的代码块，如果它不包含 `extern crate` 指示不过仍包含了被测试包装箱的名字，那么会在开头插入 `extern crate <name>`
3. 在开头会插入一些常见的属性

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

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

与它的输出看起来有些不同：

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

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

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

这是表现出来的解释：

首先，我们把 `x` 设置为 5：

```
let x = 5;
```

接着，我们把 `y` 设置为 6：

```
let y = 6;
```

最后，我们打印 `x` 和 `y` 的和：

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

这是同样的解释的原始文本：

首先，我 把``x`` 置隔``5``：

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

接着，我 把``y`` 置隔``6``：

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

最后，我 打印``x``和``y``的和：

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

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

要运行测试，那么

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

对了，`cargo test` 也会测试嵌入的文档。

这还有一些注释有利于帮助 `rustdoc` 在测试你的代码时正常工作：

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

````ignore` 指令告诉 Rust 忽略你的代码。它几乎不会是你想要的，因为它是不受支持的。相反，考虑注释 `text` 如果你希望在生成的文档中包含某些内容。  
````rust`  
````should_panic`

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

`should\_panic` 告诉 `rustdoc` 热段代捣 正确 满，但是作隔一个帧滤幕不能通。  
`no\_run` 属性会 满你的代捣，但是不运行它。热友像如“如何开始一个网 眼中”热 的例子很重要，你会希望确保它能

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

文档化模億

Rust有另一种文档注 `//!`。热种注 并不文档化接下来的内容，而是包 它的内容。塘句 滂：

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

 // ...
}
```

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

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

## 文档注释风格

查看[RFC 505](#)以了解文档风格和格式的惯例。

## 其它文档

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

当你在Markdown文件中写文档时，你并不需要加上注释前缀。例如：

```
/// # Examples
///
///
```

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

```
就是
>
```

## Examples

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

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

```
% The title
This is the example documentation.
```

% 行需要放在文件的第一行。

## doc属性

在更底层，文档注释是文档属性的语法糖：

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

跟下面这个是相同的：

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

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

## 再输出 (Re-exports)

rustdoc 会将公有部分的文档再输出：

```
extern crate foo;
pub use foo::bar;
```

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

这种行文可以通过 `no_inline` 来阻止：

```
extern crate foo;

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

## 控制HTML

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

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

这里设置了一些不同的选项，带有一个logo，一个收藏夹，和一个根URL。

## 通用选项

`rustdoc` 也提供了一些其他命令行选项，以便进一步自定义：

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

## 进阶

---

这是一个完全独立并且很有深度的部分，作为“中级”部分的补充，你可以按任意顺序阅读。这一部分的章节关注一些最复杂的功能以及一些只会出现在未来版本Rust中的特性。

阅读完“进阶”之后，你将成为一名Rust专家！

# 外部语言接口

## 介绍

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

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

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

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

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

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

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

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

`extern` 块可以扩展以包括整个snappy API：

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

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

```

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

```

## 创建安全接口

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

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

```

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

```

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

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

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

```

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

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

 snappy_compress(psrc, src_len, pdst, &mut dst_len);
 dst.set_len(dst_len as usize);
 dst
 }
}

```

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

```

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

```

```

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

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

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

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

```

作为一个参考，我们在这里使用的例子可以在[GitHub](#)的这个库中找到。

## 析构函数

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

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

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

接着回调函数可以通过一个C库的注册调用传递并在后面被执行。

一个基础的例子：

Rust代码：

```

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

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

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

```

```
}
```

C代码：

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

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

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

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

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

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

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

Rust代码：

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

extern "C" fn callback(target: *mut RustObject, a: i32) {
 println!("I'm called from C with value {}", 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 });

 // Register the callback
 let cb = register_callback;
 register_callback(&rust_object, cb);

 // Trigger the callback
 trigger_callback();
}
```

```

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

```

C代码：

```

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

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

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

```

## 异步回调

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

当外部库生成了自己的线程并触发回调时情况就变得复杂了。在这种情况下回调中对Rust数据结构的访问时特别不安全的并必须有合适的同步机制。除了想互斥量这种经典同步机制外，另一种可能就是使用通道（在 `std::comm` 中）来从触发回调的C线程转发数据到Rust线程。如果一个异步回调指定了一个在Rust地址空间的特殊Rust对象，那么在确保在对应Rust对象被销毁后不会再有回调被C库触发就格外重要了。这一点可以通过在对象的析构函数中注销回调和设计库使其确保在回调被注销后不会再被触发来取得。

## 链接

在 `extern` 上的 `link` 属性提供了基本的构建块来指示 `rustc` 如何连接到原生库。现在有两种被接受的链接属性形式：

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

在这两种形式中，`foo` 是我们链接的原生库的名字，而在第二个形式中 `bar` 是编译器要链接的原生库的类型。目前有3种已知的原生库类型：

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

注意框架只支持OSX平台。

不同 kind 的值意味着链接过程中不同原生库的参与方式。从链接的角度看，rust编译器创建了两种组件：部分的（rlib/staticlib）和最终的（dylib/binary）。原生动态库和框架会从扩展到最终组件部分，而静态库则完全不会扩展。

一些关于这些模型如何使用的例子：

- 一个原生构建依赖。有时编写部分Rust代码时需要一些C/C++代码，另外使用发行为库格式的C/C++代码只是一个负担。在这种情况下，代码会被归档为 `libfoo.a` 然后rust包装箱可以通过 `#[link(name = "foo", kind = "static")]` 声明一个依赖。

不管包装箱输出为何种形式，原生静态库将会包含在输出中，这意味着分配一个原生静态库是没有必要的。

- 一个正常动态库依赖。通用系统库（像 `readline`）在大量系统上可用，通常你找不到这类库的静态拷贝。当这种依赖被添加到包装箱里时，部分目标（比如`rlibs`）将不会链接这些库，但是当`rlib`被包含进最终目标（比如二进制文件）时，原生库将被链接。

在OSX上，框架与动态库有相同的语义。

## 不安全块

一些操作，像解引用不安全的指针或者被标记为不安全的函数只允许在unsafe块中使用。unsafe块隔离的不安全性并向编译器保证不安全代码不会泄露到块之外。

不安全函数，另一方面，将它公布于众。一个不安全的函数这样写：

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

这个函数只能被从 `unsafe` 块中或者 `unsafe` 函数调用。

## 访问外部全局变变量

外部API经常导出一个全局变量来进行像记录全局状态这样的工作。为了访问这些变量，你可以在 `extern` 块中用 `static` 关键字声明它们：

```
extern crate libc;

#[link(name = "readline")]
extern {
 static rl_readline_version: libc::c_int;
}

fn main() {
 println!("You have readline version {} installed.",
 rl_readline_version as int);
}
```

另外，你可能想修改外部接口提供的全局状态。为了做到这一点，声明为 `mut` 这样我们就可以改变它了。

```
extern crate libc;

use std::ffi::CString;
use std::ptr;

#[link(name = "readline")]
extern {
 static mut rl_prompt: *const libc::c_char;
}

fn main() {
 let prompt = CString::new("[my-awesome-shell] $").unwrap();
 unsafe {
 rl_prompt = prompt.as_ptr();

 println!("{}:{}", prompt);
 rl_prompt = ptr::null();
 }
}
```

注意与 `static mut` 变量的所有交互都是不安全的，包括读或写。与全局可变量打交道需要足够的注意。

## 外部调用约定

大部分外部代码导出为一个C的ABI，并且Rust默认使用平台C的调用约定来调用外部函数。一些外部函数，尤其是大部分Windows API，使用其它的调用约定。Rust提供了一个告诉编译器应该用哪种调用约定的方法：

```
extern crate libc;

#[cfg(all(target_os = "win32", target_arch = "x86"))]
#[link(name = "kernel32")]
#[allow(non_snake_case)]
extern "stdcall" {
 fn SetEnvironmentVariableA(n: *const u8, v: *const u8) -> libc::c_int;
}
```

这适用于整个 `extern` 块。被支持的ABI约束的列表为：

- `stdcall`
- `aapcs`
- `cdecl`
- `fastcall`
- `Rust`
- `rust-intrinsic`

- system
- C
- win64

列表中大部分ABI都是自解释的，不过 system ABI可能看起来有点奇怪。这个约束会选择任何能和目标库正确交互的ABI。例如，在x86构架的win32，这意味着会使用 stdcall ABI。在x86\_64上，然而，windows使用 c 调用约定，所以 c 会被使用。这意味在我们之前的例子中，我们可以使用 `extern "system" { ... }` 定义一个适用于所有windows系统的块，而不仅仅是x86系统。

## 外部代码交互性 (Interoperability with foreign code)

只有当 `#[repr(C)]` 属性被用于结构体时Rust能确保 `struct` 的布局兼容平台的C的表现。`#[repr(C, packed)]` 可以用来不对齐的排列结构体成员。`#[repr(C)]` 也可以被用于一个枚举。

Rust拥有的装箱（`Box<T>`）使用非空指针作为指向他包含的对象的句柄。然而，它们不应该手动创建因为它们由内部分配器托管。引用可以被安全的假设为直接指向数据的非空指针。然而，打破借用检查和可变性规则并不能保证安全，所以倾向于只在需要时使用裸指针（`*`）因为编译器不能为它们做更多假设。

向量和字符串共享同样基础的内存布局，`vec` 和 `str` 模块中可用的功能可以操作C API。然而，字符串不是 `\0` 结尾的。如果你需要一个NUL结尾的字符串来与C交互，你需要使用 `std::ffi` 模块中的 `cstring` 类型。

标准库中的 `libc` 模块包含类型别名和C标准库中的函数定义，Rust默认链接 `libc` 和 `libm`。

## “可空指针优化”

特定类型被定义为不为 `null`。这包括引用（`&T`，`&mut T`），装箱（`Box<T>`），和函数指针（`extern "abi" fn()`）。当使用C接口时，可能为空的指针经常被使用。作为一个特殊的例子，一个泛化的 `enum` 包含两个变体，其中一个没有数据，而另一个包含一个单独的字段，非常适合“可空指针优化”。当这么一个枚举被用一个非空指针类型实例化时，它表现为一个指针，而无数据的变体表现为一个空指针。那么 `Option<extern "C" fn(c_int) -> c_int>` 可以用来表现一个C ABI中的可空函数指针。

## 在C中调用Rust代码

你可能会希望这么编译Rust代码以便可以在C中调用。这是很简单的，不过需要一些东西：

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

`extern` 使这个函数遵循C调用约定，就像之前讨论[外部调用约定](#)时一样。`no_mangle` 属性关闭Rust的命名改编，这样它更容易链接。

# 不安全和底层代码

## 介绍

Rust计划在CPU和操作系统的底层细节之上提供安全的抽象，不过有时我们需要编写底层代码。本教程旨在提供一个Rust不安全子集力量与危险的概括。

Rust提供了一个 `unsafe { ... }` 块这种形式的安全出口，它允许程序猿绕开编译器检查并进行大范围的操作，例如：

- 解引用裸指针
- 通过FFI调用函数（由FFI教程介绍）
- 二进制的转换类型（`transmute`，也就是指“强制类型转换”）
- 内联汇编

注意在 `unsafe` 并没有放松对 `&` 声明周期的监管和对借用数据的锁定。

`unsafe` 的使用代表着程序员“比编译器知道的更多”，并且因此，程序员必须非常确信他们确实比这段代码为何是有效的知道的更多。总体来说，他们应该在代码库中尽可能减少不安全代码的数量；倾向于使用最少的 `unsafe` 块来构建安全的接口。

注意：Rust语言的底层细节仍在不断改变中，并且没有稳定性和后向兼容性的保证。特别的，可能会有不会造成编译错误，不过会导致语义改变（例如使用未定义的行为）的修改。因此，使用时需要格外小心。

## 指针

### 引用

Rust最大的功能之一是内存安全。这部分依赖于[所有权系统](#)，它使得编译器可以确保每一个 `&` 引用总是有效的，并且，例如，从不指向被释放的内存。

对 `&` 的限制有巨大的优势。然而，这也限制了我们对它的使用。例如，`&` 与C指针并不相同，因此并不能在外部语言接口（FFI）中作为指针使用。另外，不可变和可变引用有一些混淆和冻结的保证，作为内存安全的需要。

特别的，如果你有一个 `&T` 引用，那么 `T` 肯定不能通过这个引用或其它引用被修改。一些像 `Cell` 和 `RefCell` 的标准库类型类型，它们通过用运行时动态检查代替编译时保障来提供内部的可变性。

`&mut` 有一个不同的约束：如果一个对象有一个 `&mut T` 指向它，那么这 `&mut` 引用必须是整个程序中唯一一个指向该对象的可用路径。也就是说，`&mut` 不能是任何其它引用的别名。

使用 `unsafe` 代码来不正确的绕过和违反这些限制会导致没有定义的行为。例如，下面的包装箱创建了两个混淆的 `&mut` 指针，而这时无效的。

```

use std::mem;
let mut x: u8 = 1;

let ref_1: &mut u8 = &mut x;
let ref_2: &mut u8 = unsafe { mem::transmute(&mut *ref_1) };

// oops, ref_1 and ref_2 point to the same piece of data (x) and are
// both usable
*ref_1 = 10;
*ref_2 = 20;

```

## 裸指针

Rust提供另外两种指针类型（裸指针），写作`*const T`和`*mut T`。它们分别与C的`const T*`和`T*`类似；实际上，它们最常用的作用之一就是FFI，与外部C库交互。

与Rust语言和库提供的其它指针类型相比裸指针拥有更少的保障。例如，它

- 不能保证指向有效的内存，甚至不能保证是非空的（不像`Box`和`&`）；
- 没有任何自动清除，不像`Box`，所以需要手动管理资源；
- 是普通旧式类型，也就是说，它不移动所有权，这也不像`Box`，因此Rust编译器不能保证不出像释放后使用这种bug；
- 被认为是可发送的（如果它的内容是可发送的），因此编译器不能提供帮助确保它的使用是线程安全的；例如，你可以从两个线程中并发的访问`*mut i32`而不用同步。
- 缺少任何形式的生命周期，不像`&`，因此编译器不能判断出悬垂指针；
- 缺少关于别名的保障，除非使用`*const T`直接不允许改变，也缺少可变性的保障。

幸运的是，它有一个补偿功能：更弱的保障意味着更弱的限制。缺失的限制使得裸指针适合编写实现库中像智能指针和向量的构建模块。例如，`*`指针允许别名，使得他可以用来编写像引用计数和垃圾回收的指针这样的共享所有权类型，甚至是线程安全的共享内存类型（`Rc`和`Arc`类型都是完全用Rust实现的）。

关于裸指针有两点需要注意（也就是说需要`unsafe { ... }`块）：

- 解引用：它可以拥有任何值：所以可能的结果包括崩溃，读取未初始化内存，释放后使用，或者正常的数据读取。
- 通过`offset Intrinsic`（或者`.offset`方法）的指针算术：这个intrinsic使用所谓的“界内”算术，这就是说，如果结果是在原始指针指向的对象之内（或者是结尾之后的一个字节）的这是唯一定义了的行为。

后一个假设使得编译器可以更有效的优化。就像你看到的，实际“创建”一个裸指针是不安全的，即便是把它转换为整形。

## 引用和裸指针

在运行时，指向同样一片数据的裸指针和引用有相同的表现。事实上，在安全代码中`&T`引用会隐式转换为一个`*const T`裸指针而`mut`变体也有相似的情况（这两个转换都可以显式进行，分别生成`value as *const T`和`value as *mut T`）。

反过来，把 `*const` 转换为 `&` 是不安全的。`&T` 总是有效的，因此，最少裸指针 `*const T` 必须得指向一个有效的 `T` 的实例。此外，得到的指针必须满足引用的别名和可变性规则。编译器假设这些属性使用与任何引用，不过它是如何创建的，所以任何裸指针的转换都断言它们满足这些条件，程序猿必须保证这些。

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` 解引用风格倾向于使用 `transmute`。后一种方法远比需要的强力，并且限制更严格的操作更难出错；例如，它要求 `x` 是一个指针（不像 `transmute`）

## 使不安全变（更）安全

这里有多种方法在有不安全代码时导出安全的接口：

- 私有的保存指针（也就是说，不在公有结构体的公有字段中），这样你可以在一个地方看见和控制对指针的所有读写。
- 多使用 `assert!()`：因为你不能依靠编译器或类型系统保证你的 `unsafe` 代码在编译时是正确的，使用 `assert!()` 来验证它们会在运行时正常工作。
- 为资源实现 `Drop` 以便通过析构函数清理，并遵循RAII（资源获取就是初始化）原则。这减少了任何手动内存管理的需要，并且确保清理总会自动运行，即便是线程恐慌的情况下。
- 确保任何储存在一个裸指针中的数据都在合适的时间被销毁。

# 宏进阶

---

这一章讲解[宏介绍章节](#)遗留下来的问题。

## 句法要求

---

即使Rust代码中含有未扩展的宏，它也可以被解析为一个完整的语法树。这个属性对于编辑器或其它处理代码的工具来说十分有用。这里也有一些关于Rust宏系统设计的推论。

一个推论是Rust必须确定，当它解析一个宏扩展时，宏是否代替了

- 0个或多个项
- 0个或多个方法
- 一个表达式
- 一个语句
- 一个模式

一个块中的宏扩展代表一些项，或者一个表达式/语句。Rust使用一个简单的规则来解决这些二义性。一个代表项的宏扩展必须是

- 用大括号界定的，例如 `foo! { ... }`
- 分号结尾的，例如 `foo!(...);`

另一个展开前解析的推论是宏扩展必须包含有效的Rust记号。更进一步，括号，中括号，大括号在宏扩展中必须是封闭的。例如，`foo!([ )`是不允许的。这让Rust知道宏何时结束。

更正式一点，宏扩展体必须是一个记号树 (*token trees*) 的序列。一个记号树是一系列递归的

- 一个由 `()`，`[]` 或 `{}` 包围的记号树序列
- 任何其它单个记号

在一个匹配器中，每一个元变量都有一个片段分类符 (*fragment specifier*)，确定它匹配的哪种句法。

- `ident`：一个标识符。例如：`x`，`foo`
- `path`：一个合适的名字。例如：`T::SpecialA`
- `expr`：一个表达式。例如：`2 + 2`；`if true then { 1 } else { 2 }`；`f(42)`
- `ty`：一个类型。例如：`i32`；`Vec<(char, String)>`；`&T`
- `pat`：一个模式。例如：`Some(t)`；`(17, 'a')`；`_`
- `stmt`：一个单独语句。例如：`let x = 3`
- `block`：一个大括号界定的语句序列。例如：`{ log(error, "hi"); return 12; }`
- `item`：一个项。例如：`fn foo() { }`，`struct Bar`
- `meta`：一个“元项”，可以在属性中找到。例如：`cfg(target_os = "windows")`
- `tt`：一个单独的记号树

对于一个元变量后面的一个记号有一些额外的规则：

- `expr` 变量必须后跟一个 `=> , , , ;`
- `ty` 和 `path` 变量必须后跟一个 `=> , , , : , = , > , as`
- `pat` 变量必须后跟一个 `=> , , , =`
- 其它变量可以后跟任何记号

这些规则为Rust语法提供了一些灵活性以便将来的扩展不会破坏现有的宏。

宏系统完全不理解解析模糊。例如，`$(t:ty)* $e:expr` 语法总是会解析失败，因为解析器会被强制在解析 `$t` 和解析 `$e` 之间做出选择。改变扩展在它们之前分别加上一个记号可以解决这个问题。在这个例子中，你可以写成 `$(T $t:ty)* E $e:expr`。

## 范围和宏导入/导出

宏在编译的早期阶段被展开，在命名解析之前。这有一个缺点是与语言中其它结构相比，范围对宏的作用不一样。

定义和扩展都发生在同一个深度优先，字典顺序的包装箱的代码遍历中。那么在模块范围内定义的宏对同模块的接下来的代码是可见的，这包括任何接下来的子 `mod` 项。

一个定义在 `fn` 函数体内的宏，或者任何其它不在模块范围内的地方，只在它的范围内可见。

如果一个模块有 `subsequent` 属性，它的宏在子 `mod` 项之后的父模块也是可见的。如果它的父模块也有 `macro_use` 属性那么在父 `mod` 项之后的祖父模块中也是可见的，以此类推。

`macro_use` 属性也可以出现在 `extern crate`。在这个上下文中它控制那些宏从外部包装箱中装载，例如

```
#[macro_use(foo, bar)]
extern crate baz;
```

如果属性只是简单的写成 `#[macro_use]`，所有的宏都会被装载。如果没有 `#[macro_use]` 属性那么没有宏被装载。只有被定义为 `#[macro_export]` 的宏可能被装载。

装载一个包装箱的宏而不链接到输出，使用 `#[no_link]`。

一个例子：

```
macro_rules! m1 { () => (()) }

// visible here: m1

mod foo {
 // visible here: m1

 #[macro_export]
 macro_rules! m2 { () => (()) }

 // visible here: m1, m2
}
```

```
// visible here: m1

macro_rules! m3 { () => (()) }

// visible here: m1, m3

#[macro_use]
mod bar {
 // visible here: m1, m3

 macro_rules! m4 { () => (()) }

 // visible here: m1, m3, m4
}

// visible here: m1, m3, m4
```

当这个库被用 `#[macro_use] extern crate` 装载时，只有 `m2` 会被导入。

Rust参考中有一个[宏相关的属性列表](#)。

## \$crate变量

当一个宏在多个包装箱中使用时会产生另一个困难。让我们说 `mylib` 定义了

```
pub fn increment(x: u32) -> u32 {
 x + 1
}

#[macro_export]
macro_rules! inc_a {
 ($x:expr) => (::increment($x))
}

#[macro_export]
macro_rules! inc_b {
 ($x:expr) => (::mylib::increment($x))
}
```

`inc_a` 只能在 `mylib` 内工作，同时 `inc_b` 只能在库外工作。进一步说，如果用户有另一个名字导入 `mylib` 时 `inc_b` 将不能工作。

Rust（目前）还没有针对包装箱引用的卫生系统，不过它确实提供了一个解决这个问题的变通方法。当从一个叫 `foo` 的包装箱总导入宏时，特殊宏变量 `$crate` 会展开为 `::foo`。相反，当这个宏在同一包装箱内定义和使用时，`$crate` 将展开为空。这意味着我们可以写

```
#[macro_export]
macro_rules! inc {
 ($x:expr) => ($crate::increment($x))
}
```

来定义一个可以在库内外都能用的宏。这个函数名字会展开为 `::increment` 或 `::mylib::increment`。

为了保证这个系统简单和正确，`#[macro_use] extern crate ...` 应只出现在你包装箱的根中，而不是在 `mod` 中。这保证了 `$crate` 扩展为一个单独的标识符。

## 深入 (The deep end)

---

之前的介绍章节提到了递归宏，但并没有给出完整的介绍。还有一个原因令递归宏是有用的：每一次递归都给你匹配宏参数的机会。

作为一个极端的例子，可以，但极端不推荐，用Rust宏系统来实现一个位循环标记自动机。

```
#![feature(trace_macros)]

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 ...
 ($0, $p:tt, $($ps:tt),* ; $($ds:tt),*)
 => (bct!($($ps),*, 0, $p ; $($ds),*));

 // halt on empty data string
 ($($ps:tt),* ;)
 => (());
}

fn main() {
 trace_macros!(true);
 bct!(0, 0, 1, 1, 1 ; 1, 0, 1);
}
```

## 宏程序 (Procedural macros)

---

如果Rust宏系统不能做你想要的，你可能想要写一个编译器插件。与 `macro_rules!` 宏相比，它能做更多的事，接口也更不稳定，并且bug将更难以追踪。相反你得到了可以在编译器中运行任意Rust代码的灵活性。为此语法扩展插件有时被称为宏程序 (*procedural macros*)。

## 不稳定功能

---

这也是一个完全独立并且很有深度的部分，作为“中级”部分的补充，你可以按任意顺序阅读。

这一部分包含只在Rust每日构建中出现的功能。 (注：可能经常变动)

## 编译器插件

### 介绍

`rustc` 可以加载编译器插件，它是由用户提供的库用来扩充编译器的行为，例如新的语法扩展，lint检查等。

一个插件是带有设计好的用来在 `rustc` 中注册扩展的注册 (*registrar*) 函数的一个动态库包装箱。其它包装箱可以使用 `#[plugin(...)]` 属性来装载这个扩展。查看[rustc::plugin](#)文档来获取更多关于定义和装载插件的机制。

如果属性存在的话，`#[plugin(foo(... args ...))]` 传递的参数并不由 `rustc` 自身解释。它们被传递给插件的 Registry [args方法](#)。

在绝大多数情况下，一个插件应该只通过 `#[plugin]` 而不通过 `extern crate` 来使用。链接一个插件会将 `libsyntax` 和 `librustc` 加入到你的包装箱的依赖中。基本上你不会希望如此除非你在构建另一个插件。`plugin_as_library` lint会检查这些原则。

通常的做法是将插件放到它们自己的包装箱中，与任何那些会被库的调用者使用的 `macro_rules!` 宏或Rust代码分开。

### 语法扩展

插件可以有多种方法来扩展Rust的语法。一种语法扩展是宏过程。它们与[普通宏](#)的调用方法一样，不过扩展是通过执行任意Rust代码在编译时操作[语法树](#)进行的。

让我们写一个实现了罗马数字的插件[roman\\_numerals.rs](#)。

```
#![crate_type="dylib"]
#![feature(plugin_registrar, rustc_private)]

extern crate syntax;
extern crate rustc;

use syntax::codemap::Span;
use syntax::parse::token;
use syntax::ast::{TokenTree, TtToken};
use syntax::ext::base::{ExtCtxt, MacResult, DummyResult, MacEager};
use syntax::ext::build::AstBuilder; // trait for expr_usize
use rustc::plugin::Registry;

fn expand_rn(cx: &mut ExtCtxt, sp: Span, args: &[TokenTree])
 -> Box<MacResult + 'static> {

 static NUMERALS: &'static [(&'static str, u32)] = &[
 ("M", 1000), ("CM", 900), ("D", 500), ("CD", 400),
 ("C", 100), ("XC", 90), ("L", 50), ("XL", 40),
 ("X", 10), ("IX", 9), ("V", 5), ("IV", 4),
 ("I", 1)];
}
```

```

let text = match args {
 [TtToken(_, token::Ident(s, _))] => token::get_ident(s).to_string(),
 _ => {
 cx.span_err(sp, "argument should be a single identifier");
 return DummyResult::any(sp);
 }
};

let mut text = &*text;
let mut total = 0;
while !text.is_empty() {
 match NUMERALS.iter().find(|&&(rn, _)| text.starts_with(rn)) {
 Some(&(rn, val)) => {
 total += val;
 text = &text[rn.len()..];
 }
 None => {
 cx.span_err(sp, "invalid Roman numeral");
 return DummyResult::any(sp);
 }
 }
}

MacEager::expr(cx.expr_u32(sp, total))
}

#[plugin_registrar]
pub fn plugin_registrar(reg: &mut Registry) {
 reg.register_macro("rn", expand_rn);
}

```

我们可以像其它宏那样使用 `rn!()` :

```

#![feature(plugin)]
#![plugin(roman_numerals)]

fn main() {
 assert_eq!(rn!(MMXV), 2015);
}

```

与一个简单的 `fn(&str) -> u32` 函数相比的优势有 :

- (任意复杂程度的) 转换都发生在编译时
- 输入验证也在编译时进行
- 可以扩展并允许在模式中使用，它可以有效的为任何数据类型定义新语法。

除了宏过程，你可以定义新的类[derive](#)属性和其它类型的扩展。查看[Registry::register\\_syntax\\_extension](#)和[SyntaxExtension enum](#)。对于更复杂的宏例子，查看[regex\\_macros](#)。

## 提示与技巧

这里提供一些[宏调试的提示](#)。

你可以使用[syntax::parse](#)来将记号树转换为像表达式这样的更高级的语法元素：

```
fn expand_foo(cx: &mut ExtCtxt, sp: Span, args: &[TokenTree])
 -> Box<MacResult+'static> {
 let mut parser = cx.new_parser_from_tts(args);
 let expr: P<Expr> = parser.parse_expr();
```

看完[libsyntax解析器代码](#)会给你一个解析基础设施如何工作的感觉。

保留你解析所有的[Span](#)，以便更好的报告错误。你可以用[Spanned](#)包围你的自定数据结构。

调用[ExtCtxt::span\\_fatal](#)将会立即终止编译。相反最好调用[ExtCtxt::span\\_err](#)并返回[DummyResult](#)，这样编译器可以继续并找到更多错误。

为了打印用于调试的语法段，你可以同时使用[span\\_note](#)和[syntax::print::pprust::\\*::to\\_string](#)。

上面的例子使用[AstBuilder::expr\\_usize](#)产生了一个普通整数。作为一个 `AstBuilder` 特性的额外选择，`libsyntax` 提供了一个[准引用宏](#)的集合。它们并没有文档并且非常边缘化。然而，这些将会是实现一个作为一个普通插件库的改进准引用的好出发点。

## Lint插件

插件可以扩展[Rust Lint基础设施](#)来添加额外的代码风格，安全检查等。你可以查看[src/test/auxiliary/lint\\_plugin\\_test.rs](#)来了解一个完整的例子，我们在这里重现它的核心部分：

```
declare_lint!(TEST_LINT, Warn,
 "Warn about items named 'lintme'")

struct Pass;

impl LintPass for Pass {
 fn get_lints(&self) -> LintArray {
 lint_array!(TEST_LINT)
 }

 fn check_item(&mut self, cx: &Context, it: &ast::Item) {
 let name = token::get_ident(it.ident);
 if name.get() == "lintme" {
 cx.span_lint(TEST_LINT, it.span, "item is named 'lintme'");
 }
 }
}

#[plugin_registrar]
pub fn plugin_registrar(reg: &mut Registry) {
 reg.register_lint_pass(box Pass as LintPassObject);
}
```

那么像这样的代码：

```
#![plugin(lint_plugin_test)]

fn lintme() { }
```

将产生一个编译警告：

```
foo.rs:4:1: 4:16 warning: item is named 'lintme', #[warn(test_lint)] on by default
foo.rs:4 fn lintme() { }
^-----
```

Lint插件的组件有：

- 一个或多个 `declare_lint!` 调用，它定义了 `Lint` 结构
- 一个用来存放lint检查所需的所有状态（在我们的例子中，没有）
- 一个定义了如何检查每个语法元素的 `LintPass` 实现。一个单独的 `LintPass` 可能会对多个不同的 `Lint` 调用 `span_lint`，不过它们都需要用 `get_lints` 方法进行注册。

Lint过程是语法遍历，不过它们运行在编译的晚期，这时类型信息时可用的。`rustc` 的内建lint与lint插件使用相同的基础构架，并提供了如何访问类型信息的例子。

由插件定义的语法通常通过属性和插件标识控制，例如，`[#[allow(test_lint)]]`，`-A test-lint`。这些标识符来自于 `declare_lint!` 的第一个参数，经过合适的大小写和标点转换。

你可以运行 `rustc -W help foo.rs` 来见检查lint列表是否为 `rustc` 所知，包括由 `foo.rs` 加载的插件。

## 内联汇编

为了极端底层操作和性能要求，你可能希望直接控制CPU。Rust通过 `asm!` 宏来支持使用内联汇编。语法大体上与GCC和Clang相似：

```
asm!(assembly template
 : output operands
 : input operands
 : clobbers
 : options
);
```

任何 `asm` 的使用需要功能通道（需要在包装箱上加上 `#![feature(asm)]` 来允许使用）并且当然也需要写在 `unsafe` 块中

注意：这里的例子使用了x86/x86-64汇编，不过所有平台都受支持。

## 汇编模板

`assembly template` 是唯一需要的参数并且必须是原始字符串（就是 `" "`）

```
#![feature(asm)]

#[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
fn foo() {
 unsafe {
 asm!("NOP");
 }
}

// other platforms
#[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))]
fn foo() { /* ... */ }

fn main() {
 // ...
 foo();
 // ...
}
```

（`feature(asm)` 和 `#[cfg]` 从现在开始将被忽略。）

输出操作数，输入操作数，覆盖和选项都是可选的不过你必选加上正确数量的：`:` 如果你要省略它们的话：

```
asm!("xor %eax, %eax"
 :
 :
 : "eax"
);
```

空格是无所谓的：

```
asm!("xor %eax, %eax" :::"eax");
```

## 操作数

输入和输出操作数都有相同的格式：`: "constraints1"(expr1), "constraints2"(expr2), ...`。输出操作数表达式必须是可变的左值：

```
fn add(a: i32, b: i32) -> i32 {
 let mut c = 0;
 unsafe {
 asm!("add $2, $0"
 : "=r"(c)
 : "0"(a), "r"(b)
);
 }
 c
}

fn main() {
 assert_eq!(add(3, 14159), 14162)
}
```

## 覆盖

一些指令修改可能保存有不同值寄存器所以我们使用覆盖列表来告诉编译器不要假设任何装载在这些寄存器的值是有效的。

```
// Put the value 0x200 in eax
asm!("mov $$0x200, %eax" : /* no outputs */ : /* no inputs */ : "eax");
```

输入和输出寄存器并不需要列出因为这些信息已经通过给出的限制沟通过了。因此，任何其它的被使用的寄存器应该隐式或显式的被列出。

如果汇编修改了代码状态寄存器 `cc` 则需要在覆盖中被列出，如果汇编修改了内存，`memory` 也应被指定。

## 选项

最后一部分，`options` 是Rust特有的。格式是逗号分隔的基本字符串（也就是说，`:"foo", "bar", "baz"`）。它被用来指定关于内联汇编的额外信息：

目前有效的选项有：

1. `volatile` - 相当于gcc/clang中的`__asm__ __volatile__ (...)`

2. *alignstack* - 特定的指令需要栈按特定方式对齐（比如，SSE）并且指定这个告诉编译器插入通常的栈对齐代码
3. *intel* - 使用intel语法而不是默认的AT&T语法

## 不使用标准库

`std` 默认被链接到每个Rust包装箱中。在一些情况下，这是不合适的，并且可以通过在包装箱上加入`#![no_std]`属性来避免这一点。

```
// a minimal library
#![crate_type="lib"]
#![feature(no_std)]
#![no_std]
```

很明显不光库可以使用这一点：你可以在可执行文件上使用`#![no_std]`，控制程序入口点有两种可能的方式：`#[start]`属性，或者用你自己的去替换C语言默认的`main`函数。

被标记为`#[start]`的函数传递的参数格式与C一致：

```
#![feature(lang_items, start, no_std)]
#![no_std]

// Pull in the system libc library for what crt0.o likely requires
extern crate libc;

// Entry point for this program
#[start]
fn start(_argc: isize, _argv: *const *const u8) -> isize {
 0
}

// These functions and traits are used by the compiler, but not
// for a bare-bones hello world. These are normally
// provided by libstd.
#[lang = "stack_exhausted"] extern fn stack_exhausted() {}
#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "panic_fmt"] fn panic_fmt() -> ! { loop {} }
```

要覆盖编译器插入的`main`函数，你必须使用`#![no_main]`并通过正确的ABI和正确的名字来创建合适的函数，这也需要覆盖编译器的命名改编：

```
#![feature(no_std)]
#![no_std]
#![no_main]
#![feature(lang_items, start)]

extern crate libc;

#[no_mangle] // ensure that this symbol is called `main` in the output
pub extern fn main(argc: i32, argv: *const *const u8) -> i32 {
 0
}

#[lang = "stack_exhausted"] extern fn stack_exhausted() {}
```

```
#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "panic_fmt"] fn panic_fmt() -> ! { loop {} }
```

目前编译器对能够被可执行文件调用的符号做了一些假设。正常情况下，这些函数是由标准库提供的，不过没有它你就必须定义你自己的了。

这三个函数中的第一个 `stack_exhausted`，当检测到栈溢出时被调用。这个函数对于如何被调用和应该干什么有一些限制，不顾如果栈限制寄存器没有被维护则一个线程可以有“无限的栈”，这种情况下这个函数不应该被触发。

第二个函数，`eh_personality`，被编译器的错误机制使用。它通常映射到GCC的特性函数上（查看[libstd实现来获取更多信息](#)），不过对于不会触发恐慌的包装箱可以确定这个函数不会被调用。最后一个函数，`panic_fmt`，也被编译器的错误机制使用。

## 使用libcore

**注意**：核心库的结构是不稳定的，建议在任何可能的情况下使用标准库。

通过上面的计数，我们构造了一个少见的运行Rust代码的可执行程序。标准库提供了很多功能，然而，这是Rust的生产力所需要的。如果标准库是不足的话，那么可以使用被设计为标准库替代的[libcore](#)。

核心库只有很少的依赖并且比标准库可移植性更强。另外，核心库包含编写符合习惯和高效Rust代码的大部分功能。

例如，下面是一个计算由C提供的两个向量的数量积的函数，使用常见的Rust实现。

```
#![feature(lang_items, start, no_std)]
#![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 },
))
 }

 let mut sum: u32 = 0;
 for i in 0..a_len {
 sum += a_slice[i] * b_slice[i];
 }
 sum
}
```

```

};

// Iterate over the slices, collecting the result
let mut ret = 0;
for (i, j) in a_slice.iter().zip(b_slice.iter()) {
 ret += (*i) * (*j);
}
return ret;
}

#[lang = "panic_fmt"]
extern fn panic_fmt(args: &core::fmt::Arguments,
 file: &str,
 line: u32) -> ! {
 loop {}
}

#[lang = "stack_exhausted"] extern fn stack_exhausted() {}
#[lang = "eh_personality"] extern fn eh_personality() {}

```

注意这里有一个额外的 `lang` 项与之前的例子不同，`panic_fmt`。它必须由 `libcore` 的调用者定义因为核心库声明了恐慌，但没有定义它。`panic_fmt` 项是这个包装箱的恐慌定义，并且它必须确保不会返回。

正如你在例子中所看到的，核心库尝试在所有情况下提供 Rust 的功能，不管平台的要求如何。另外一些库，例如 `liballoc`，为 `libcore` 增加了进行其它平台相关假设的功能，不过这依旧比标准库更有可移植性。

## 固有功能

注意：固有功能将会永远是一个不稳定的接口，推荐使用稳定的libcore接口而不是直接使用编译器自带的功能。

可以像FFI函数那样导入它们，使用特殊的 `rust-intrinsic` ABI。例如，如果在一个独立的上下文，但是想要能在类型间 `transmute`，并想进行高效的指针计算，你可以声明函数：

```
extern "rust-intrinsic" {
 fn transmute<T, U>(x: T) -> U;

 fn offset<T>(dst: *const T, offset: isize) -> *const T;
}
```

跟其它FFI函数一样，它们总是 `unsafe` 的。

## 语言项

注意：语言项通常由Rust发行版的包装箱提供，并且它自身有一个不稳定的接口。建议使用官方发布的包装箱而不是定义自己的版本。

rustc 编译器有一些可插入的操作，也就是说，功能不是硬编码进语言的，而是在库中实现的，通过一个特殊的标记告诉编译器它存在。这个标记是 `#[lang="..."]` 属性并且有不同的值 ... ,也就是不同的“语言项”。

例如，`Box` 指针需要两个语言项，一个用于分配，一个用于释放。下面是一个独立的程序使用 `Box` 语法糖进行动态分配，通过 `malloc` 和 `free`：

```
#![feature(lang_items, box_syntax, start, no_std)]
#![no_std]

extern crate libc;

extern {
 fn abort() -> !;
}

#[lang = "owned_box"]
pub struct Box<T>(*mut T);

#[lang="exchange_malloc"]
unsafe fn allocate(size: usize, _align: usize) -> *mut u8 {
 let p = libc::malloc(size as libc::size_t) as *mut u8;

 // malloc failed
 if p as usize == 0 {
 abort();
 }

 p
}
#[lang="exchange_free"]
unsafe fn deallocate(ptr: *mut u8, _size: usize, _align: usize) {
 libc::free(ptr as *mut libc::c_void)
}

#[start]
fn main(argc: isize, argv: *const *const u8) -> isize {
 let x = box 1;

 0
}

#[lang = "stack_exhausted"] extern fn stack_exhausted() {}
#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "panic_fmt"] fn panic_fmt() -> ! { loop {} }
```

注意 `abort` 的使用：`exchange_malloc` 语言项假设返回一个有效的指针，所以需要在内部进行检查。

其它语言项提供的功能包括：

- 通过特性重载运算符：`==`，`<`，解引用（`*`）和`+`等运算符对应的特性都有语言项标记；上面4个分别为`eq`，`ord`，`deref`和`add`
- 栈展开和一般故障：`eh_personality`，`fail`和`fail_bounds_checks`语言项
- `std::marker`中用来标明不同类型的特性：`send`，`sync`和`copy`。
- `std::marker`标记类型和变化指示器：`covariant_type`和`contravariant_lifetime`等

语言项由编译器延时加载；例如，如果你从未用过`Box`则就没有必要定义`exchange_malloc`和`exchange_free`的函数。`rustc`在一个项被需要而无法在当前包装箱或任何依赖中找到时生成一个错误。

## 链接参数

这里还有一个方法来告诉rustc如何自定义链接，这就是通过 `link_args` 属性。这个属性作用于 `extern` 块并指定当产生构件时需要传递给连接器的原始标记。一个用例将是：

```
#![feature(link_args)]
#[link_args = "-foo -bar -baz"]
extern {}
```

注意现在这个功能隐藏在 `feature(link_args)` 通道之后因为它并不是一个被认可的执行链接的方法。目前 rustc 从 shell 调用系统的连接器，所以使用额外的命令行参数是可行的，不过这并不一定永远可行。将来 rustc 可能使用 LLVM 直接链接原生库这样以来 `link_args` 就毫无意义了。

强烈建议你不要使用这个属性，而是使用一个更正式的 `[link(...)]` 属性作用于 `extern` 块。

## 基准测试

Rust也支持基准测试，它可以测试代码的性能。让我们把 `src/lib.rs` 修改成这样（省略注释）：

```
extern crate test;

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));
 }

 #[bench]
 fn bench_add_two(b: &mut Bencher) {
 b.iter(|| add_two(2));
 }
}
```

我们导入了 `test` 包装箱，它包含了对基准测试的支持。我们也定义了一个新函数，带有 `bench` 属性。与一般的不带参数的测试不同，基准测试有一个 `&mut Bencher` 参数。`Bencher` 提供了一个 `iter` 方法，它接收一个闭包。这个闭包包含我们想要测试的代码。

我们可以用 `cargo bench` 来运行基准测试：

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

running 2 tests
test tests::it_works ... ignored
test tests::bench_add_two ... bench: 1 ns/iter (+/- 0)

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

我们的非基准测试将被忽略。你也许会发现 `cargo bench` 比 `cargo test` 花费的时间更长。这是因为Rust会多次运行我们的基准测试，然后取得平均值。因为我们的函数只做了非常少的操作，我们耗费了 `1 ns/iter (+/- 0)`，不过运行时间更长的测试就会有出现偏差。

编写基准测试的建议：

- 把初始代码放于 `iter` 循环之外，只把你需要测试的部分放入它
- 确保每次循环都做了“同样的事情”，不要累加或者改变状态

- 确保 `iter` 循环内简短而快速，这样基准测试会运行的很快同时校准器可以在合适的分辨率上调整运转周期
- 确保 `iter` 循环执行简单的工作，这样可以帮助我们准确的定位性能优化（或不足）

## Gocha：优化

写基准测试有另一些比较微妙的地方：开启了优化编译的基准测试可能被优化器戏剧性的修改导致它不再是期望的基准测试了。举例来说，编译器可能认为一些计算并无外部影响并且整个移除它们。

```
extern crate test;
use test::Bencher;

#[bench]
fn bench_xor_1000_ints(b: &mut Bencher) {
 b.iter(|| {
 (0..1000).fold(0, |old, new| old ^ new);
 });
}
```

得到如下结果：

```
running 1 test
test bench_xor_1000_ints ... bench: 0 ns/iter (+/- 0)

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

基准测试运行器提供两种方法来避免这个问题：要么传递给 `iter` 的闭包可以返回一个随机的值这样强制优化器认为结果有用并确保它不会移除整个计算部分。这可以通过修改上面例子中的 `b.iter` 调用：

```
b.iter(|| {
 // note lack of `;` (could also use an explicit `return`).
 (0..1000).fold(0, |old, new| old ^ new)
});
```

要么，另一个选择是调用通用的 `test::black_box` 函数，它会传递给优化器一个不透明的“黑盒”这样强制它考虑任何它接收到的参数。

```
extern crate test;

b.iter(|| {
 let n = test::black_box(1000);

 (0..n).fold(0, |a, b| a ^ b)
})
```

上述两种方法均未读取或修改值，并且对于小的值来说非常廉价。对于大的只可以通过间接传递来减小额外开销（例如：`black_box(&huge_struct)`）。

执行上面任何一种修改可以获得如下基准测试结果：

```
running 1 test
test bench_xor_1000_ints ... bench: 131 ns/iter (+/- 3)

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

然而，即使使用了上述方法优化器还是可能在不合适的情况下修改测试用例。

## 装箱语法和模式

目前唯一稳定的创建 Box 的方法是通过 `Box::new` 方法。并且不可能在一个模式匹配中稳定的析构一个 Box。不稳定的 `box` 关键字可以用来创建和析构 Box。下面是一个用例：

```
#![feature(box_syntax, box_patterns)]

fn main() {
 let b = Some(box 5);
 match b {
 Some(box n) if n < 0 => {
 println!("Box contains negative number {}", n);
 },
 Some(box n) if n >= 0 => {
 println!("Box contains non-negative number {}", n);
 },
 None => {
 println!("No box");
 },
 _ => unreachable!()
 }
}
```

注意这些功能目前隐藏在 `box_syntax` (装箱创建) 和 `box_patterns` (析构和模式匹配) 通道之中因为它的话语在未来可能会改变。

## 返回指针

在很多有指针的语言中，你的函数可以返回一个指针来避免拷贝大的数据结构。例如：

```
struct BigStruct {
 one: i32,
 two: i32,
 // etc
 one_hundred: i32,
}

fn foo(x: Box<BigStruct>) -> Box<BigStruct> {
 Box::new(*x)
}

fn main() {
 let x = Box::new(BigStruct {
 one: 1,
 two: 2,
 one_hundred: 100,
 });

 let y = foo(x);
}
```

要点是通过传递一个装箱，你只需拷贝了一个指针，而不是那构成了 `BigStruct` 的一百个 `int` 值。

上面是Rust中的一个反模式。相反，这样写：

```
#![feature(box_syntax)]

struct BigStruct {
 one: i32,
 two: i32,
 // etc
 one_hundred: i32,
}

fn foo(x: Box<BigStruct>) -> BigStruct {
 *x
}

fn main() {
 let x = Box::new(BigStruct {
 one: 1,
 two: 2,
 one_hundred: 100,
 });
 let y: Box<BigStruct> = box foo(x);
}
```

注意我们使用了 `box_syntax` 特性gate，因此这个语法将来可能会改变。

这在不牺牲性能的前提下获得了灵活性。

你可能会认为这会给我们带来很差的性能：返回一个值然后马上把它装箱？难道这在哪里不都是最糟的吗？Rust显得更聪明。这里并没有拷贝。`main` 为装箱分配了足够的空间，向 `foo` 传递一个指向他内存的 `x`，然后 `foo` 直接向 `Box<T>` 中写入数据。

因为这很重要所以要说两遍：返回指针会阻止编译器优化你的代码。允许调用函数选择它们需要如何使用你的输出。

## 总结

---

我们已经讲了很多内容了。当你掌握了本教程的所有内容后，你将会对Rust开发有一个坚实的理解。当然，还有很多内容我们没有涉及到，我吗仅仅是理解了表面。这里还有成吨的主题值得挖掘，比如[标准库](#)的API文档，在[Rust by Example](#)中寻找通用问题的解决方法，或者在[crates.io](#)上浏览由社区编写的包装箱。

快乐编程！←\_←

## 词汇表

不是每位Rustacean都是系统编程或计算机科学背景的，所以我们加上了可能难以理解的词汇解释。

### 数量 (Arity)

Arity代表函数或操作所需的参数数量。

```
let x = (2, 3);
let y = (4, 6);
let z = (8, 2, 6);
```

在上面的例子中 `x` 和 `y` 的Arity是 2，`z` 的Arity是 3。

### 抽象语法树 (Abstract Syntax Tree)

当一个编译器编译你程序的时候，它做了很多不同的事。其中之一就是将你程序中的文本转换为一个‘抽象语法树’，或者‘AST’。这个树是你程序结构的表现。例如，`2 + 3` 可以转换为一个树：

```
+
/\
2 3
```

而 `2 + (3 * 4)` 看起来像这样：

```
+
/ \
2 *
 / \
 3 4
```