Contents

前言

Rust 是一门为系统级编程设计的语言。

因为很多业务程序员对系统级编程并不是很熟悉，所以这里解释一下，它是我们所做的一切的基础。

当你合上笔记本电脑时，操作系统检测到了这一行为，然后把所有正在运行的程序挂起、关掉屏幕、并把电脑设置为睡眠。之后，当你打开笔记本电脑时：屏幕和其他组件被再次唤醒，并且每个程序可以在它中断的地方继续与逆行。我们对此习以为常。但系统程序员为此编写了很多代码。

系统级编程被用于以下领域：

- 操作系统
- 各种设备的驱动
- 文件系统
- 数据库
- 在非常廉价或需要极高的可靠性的设备上运行的代码
- 密码学
- 多媒体编解码器（用于读写音频、视频、图片文件的软件）
- 多媒体处理（例如，语音识别或图像处理软件）
- 内存管理（例如，实现一个垃圾回收器）
- 文本渲染（把文本和字体转换为像素点的过程）
- 实现更高级的编程语言（例如 JavaScript 和 Python）
- 网络
- 虚拟化和容器
- 科学仿真
- 游戏

简而言之，系统级编程是一种资源受限的编程方式，它是一种每一个字节和每一个 CPU 时钟都需要考虑的编程方式。为了支持一个基本的应用所需要的系统级代码的数量是非常惊人的。

本书并不会教你系统级编程。事实上，这本书包含了很多有关内存管理的细节，如果你没有自己进行过系统级编程，你会感觉这些内容乍一看似乎没有必要。但如果你是一个熟练的系统级程序员，你将会发现 Rust 是一门非常优秀的语言：它是一件可以解决困扰了整个工业界几十年的主要问题的工具。

谁应该阅读这本书

如果你已经是一名系统级程序员并且已经准备好替换C++，那么这本书就是为你而生。如果你是一名有其他任何语言的经验的开发者，不管是C#、Java、Python、JavaScript还是其他语言，这本书同样适用于你。

然而，你不仅需要学习Rust。为了充分利用这门语言，你还需要获取一些系统级编程的经验。我们推荐在阅读这本书的同时用Rust实现一些系统编程的项目，构建一些你以前从来没有构建过的东西、一些充分利用Rust的速度、并发和安全性的东西。本前言开头的列表也许能给你一些启发。

我们为什么要撰写这本书

早在我们开始学习Rust时我们就开始着手编写这本书。我们的目标是首先正面处理Rust中主要的、新的概念，清晰而深入地呈现它们，以最大限度地避免通过试错来学习。

本书概览

本书的前两章介绍了Rust，并提供了一个简单的示例。之后我们转到第3章的基本数据类型。第4章、第5章专注于介绍所有权和引用的核心概念。我们推荐按照顺序阅读前5章。

第6到第10章覆盖了语言的基础部分：表达式（第6章），错误处理（第7章），crate和模块（第8章），结构体（第09章），枚举和模式（第10章）。在这一部分可以跳过一些内容，但请相信我们，不要跳过错误处理的章节。第11章包括trait和泛型，这是最后两个你需要知道的重要概念。trait类似于Java或C#中的接口。它们也是Rust支持把你自己的类型集成到语言中的主要手段。第12章展示了trait怎么支持运算符重载，第13章包括了很多有用的工具trait。

理解了trait和泛型就可以解锁本书的剩余部分了。闭包和迭代器，这两个你绝对不想错过的强大工具，分别在第14章和第15章中介绍。你可以以任意顺序阅读剩下的章节，或者按需阅读。它们包括了剩余的语言部分：集合（第16章），字符串和文本（第17章），输入和输出（第18章），并发（第19章），异步代码（第20章），宏（第21章），unsafe代码（第22章）和调用其它语言中的函数（第23章）。

本书中的约定

本书中用到了以下约定：

斜体

表示新术语、URL、电子邮件地址、文件名和文件拓展名。

`等宽`

用于代码环境和在段落中引用代码中的元素例如变量或函数名、数据库、数据类型、环境变量、语句和关键字等。

`等宽加粗`

表示需要用户逐字输入的命令或其他文本。

等宽斜体

表示需要用户用自己的值或者上下文推断出的值进行替换的文本。

> **NOTE**
>
> 这个标志表示一个注意事项。

使用示例代码

补充材料（示例代码，练习等）可以在https://github.com/ProgrammingRust下载。

本书旨在帮助你完成工作。一般来说，本书中提供的示例代码，你可以在自己的编程和文章中使用。除非你需要再次分发大量本书中的代码，否则你不需要联系我们获取授权。例如，编写一个使用了书中部分代码的程序并不需要授权。售卖或者分发O'Reilly书籍中的示例代码则需要授权。通过引用本书或书中的示例代码回答问题并不需要授权。将本书中的大量代码纳入你自己的产品文档则需要授权。

我们感激，但并不要求署名。如果要署名的话，应该包含标题、作者、出版社和ISBN。例如："Programming Rust, Second Edition by Jim Blandy, Jason Orendorff, and Leonora F.S. Tindall(O'Reilly). Copyright 2021 Jim Blandy, Leonora F.S. Tindall, and Jason Orendorff, 978-1-492-05259-3."

如果你感觉你对示例代码的使用方式不在上述范围内，请放心联系我们permissions@oreilly.com。

O'Reilly在线学习

> **NOTE**
>
> 40多年以来，O'Reilly Media提供技术和业务培训、知识和洞察力，以帮助公司取得成功。

我们独特的专家和创意网络会通过书籍、文章、会议和在线学习平台分享他们的知识。O'Reilly的在线学习平台可以让你按需访问O'Reilly及其他200多家出版社的实时培训课程、深入学习路线、交互式代码环境。更多信息请访问http://oreilly.com。

如何联系我们

请将和本书有关的评论和问题发送给出版社：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

我们有一个本书的web页面，我们在那里列出了勘误表、示例和其他附加信息。你可以通过https://oreil.ly/programming-rust-2e访问该页面。

评论或有关本书的技术问题可以发送到邮箱bookquestions@oreilly.com。

访问http://www.oreilly.com获取更多有关我们的书籍和课程的信息。

我们的Facebook：http://facebook.com/oreilly

我们的Twitter：http://twitter.com/oreillymedia

我们的YouTube：http://youtube.com/oreillymedia

致谢

你手中的这本书从我们的官方技术评审：Brian Anderson、Matt Brubeck、J. David Eisenberg、Ryan Levick、Jack Moffitt、Carol Nichols、Erik Nordin和翻译：Hidemoto Nakada (中田秀基) (Japanese)、Mr. Songfeng Li (Simplified Chinese)、Adam Bochenek和Krzysztof Sawka (Polish)处受益匪浅。

还有很多非官方的评审阅读了早期的草案并提供了很有价值的反馈。我们想要感谢Eddy Bruel、Nick Fitzgerald、Graydon Hoare、Michael Kelly、Jeffrey Lim、Jakob Olesen、Gian-Carlo Pascutto、Larry Rabinowitz、Jaroslav Šnajdr、Joe Walker、Yoshua Wuyts的认真评论。Jeff Walden和NicolasPierron花费了宝贵的时间来审阅几乎整本书。就像编程一样，一本编程书籍需要高质量的bug报告才能不断成长。感谢你们。

Mozilla对Jim和Jason在此项目中的工作非常宽容，尽管这超出了我们的官方职责范围，并会和他们产生竞争。我们非常感谢Jim和Jason的领导：Dave Camp、Naveed Ihsanullah、Tom Tromey、Joe Walker的支持。他们从长远的角度看待Mozilla，我们希望这些结果证明了他们对我们的信任。

我们还想对O'Reilly里每个帮助过我们的人表达感谢，尤其是非常有耐心的编辑Jeff Bleiel和Brian MacDonald，以及我们的策划编辑Zan McQuade。

最重要的是，我们衷心感谢家人们坚定不移的爱、热情和耐心。

Chapter 1

系统级程序员的福音

在某些场景下——例如 *Rust* 的目标场景——比竞争对手快 *10x* 或者仅仅 *2x* 是足以决定成败的事情。它决定了一个系统在市场中的命运，就像在硬件市场中一样。

——Graydon Hoare

现在所有的计算机都是并行的……并行编程才是编程。

——Michael McCool et al., Structured Parallel Programming

民族国家的攻击者们利用 *TrueType* 解析器的漏洞来进行监视；所有的软件都对安全很敏感。

——Andy Wingo

我们选择用以上三个引言来开始本书是有原因的。但首先让我们以一个谜团开始。下面的 C 程序做了什么？

```
int main(int argc, char **argv) {
    unsigned long a[1];
    a[3] = 0x7ffff7b36cebUL;
    return 0;
}
```

今天早上在 Jim 的笔记本电脑上，上面的程序输出了：

```
undef: Error: .netrc file is readable by others.
undef: Remove password or make file unreadable by others.
```

然后它就崩溃了。如果你在自己的机器上尝试，它的行为可能会不同。这个过程中到底发生了什么呢？

这段代码是有漏洞的。数组 a 的长度为 1，对 a[3] 的访问，根据 C 语言标准，是未定义行为：

> 对于使用不可移植或错误的程序结构或错误的数据的行为，国际标准不做任何要求

未定义行为不仅仅会导致非预期的结果，语言标准甚至允许程序在这种情况下做任何事情。在我们的例子中，把一个特定的值存在特定数组的第 4 个元素处恰巧破坏了函数的调用栈，因此导致 main 函数返回时，并没有正常的退出程序，而是跳转到了 C 标准库里从用户的家目录中的一个文件中读取密码的代码中。这显然是有问题的。

C 和 C++ 有几百条避免未定义行为的规则。它们大多都是一些常识：不要访问不应该访问的内存，不要让算术运算溢出，不要除以零等等。然而编译器并不强制这些规则，它没有义务去检测哪怕明目张胆的违反规则的行为。事实上，上述程序编译时不会有错误和警告。避免未定义行为的责任全部落到你，程序员身上。

从经验上讲，我们程序员并不能很好的识别出未定义行为。然而，一个 Utah 大学的学生，研究员 Peng Li 修改了 C 和 C++ 的编译器来让它们在编译时报告它们正在编译的程序中是否含有会导致未定义行为的模式。他发现，几乎所有的程序都有，包括那些公认的优秀项目。想要在 C 和 C++ 中避免未定义行为就和仅仅知道规则就想赢得国际象棋比赛一样不切实际。

各种偶然的奇怪信息或崩溃可能是质量问题，但自从 1988 年 Morris Worm 使用了以下技术把代码从一台机器迁移到另一台机器后，不经意间的未定义行为就是导致安全漏洞的一大原因。

因此 C 和 C++ 把程序员推到了一个很尴尬的地位：这些语言是系统级编程的工业标准，但它们对程序员的要求却几乎保证崩溃和安全问题源源不断。回答我们的谜团只是抛出了一个更大的问题：我们不能做的更好吗？

1.1 Rust 帮你扛起责任

我们的答案对应着我们开头的三个引言。第三个引言引用自一篇报告，这篇报告中，一个叫做 Stuxnet 的计算机蠕虫在 2010 年被发现侵入了工业界的设备，并获取了受害计算机的控制权。它只是利用了解析 word 文档中嵌入的 TrueType 字体的代码中的未定义行为，没有使用任何其他技术。这段代码的作者显然没有预料到这段代码会被以这种形式利用，这说明不仅仅只有操作系统和服务器需要担心安全问题：任何需要处理来自不受信任来源的数据的软件都可能成为受害者。

Rust 语言做了一个简单的保证：如果你的代码通过了编译器的检查，那么它将不会遇到未

定义行为。悬垂指针，两次释放，空指针解引用都会在编译期被捕捉到。对数组的引用通过编译期和运行期的双重检查保证安全，当索引越界时，Rust不会像不幸的C语言一样出现缓冲区溢出：而是会安全的退出程序并打印出错误消息。

Rust旨在同时实现安全和易于使用。为了对你的程序行为做出更强的保证，Rust对你的代码施加了比C和C++更多的限制，这些限制需要通过一些实践和经验才能习惯。但总体来看这门语言的灵活性和表达力都是很强的。Rust的应用范围之广已经证明了这一点。

根据我们的经验，在相信语言可以帮助我们捕获错误的情况下，我们将有勇气尝试更有挑战性的项目。修改庞大的、复杂的程序将不会有很大风险，因为不需要关注内存管理和指针有效性的问题。调试起来也会简单得多，因为潜在的bug不会出现在不相关的部分。

当然，还有很多Rust也不能检测出的bug。但在实践中，没有未定义行为可以显著改善开发的现状。

1.2 并行编程是安全的

在C和C++中并发是众所周知的难，开发者通常只有在已经证明了单线程代码无法达到所需性能的情况下才会考虑并发。但第二个引言则认为并行非常重要以至于现代计算机将它视为基本的操作。

事实证明，Rust中保证内存安全的限制也可以保证Rust程序中不会出现数据竞争。你可以在线程间安全的共享数据，只要它不是正在被修改。被修改的数据只能通过同步原语来访问。所有传统的工具都可以使用：自旋锁、条件变量、通道、原子量等等。Rust会通过检查确保你正确地使用它们。

这些使Rust能够充分利用现代多核机器的性能。Rust的生态还提供了普通并发原语之外的库来帮助你完成复杂的负载，包括处理器池、无锁同步机制例如Read-Copy-Update等。

1.3 Rust的速度很快

最后，对应我们的第一条引言。Rust遵循了Bjarne Stroustrup在他的文章“Abstraction and the C++ Machine Model”中提到的为C++设计的原则：

一般情况下，C++的实现遵循0开销原则：你没有用到的部分，将不会有开销。你用到的部分，你将不能找到更好的代码。

系统级编程经常需要关注将机器性能发挥到极限。对于视频游戏，整个机器都需要投入工作来为玩家创造出最好的体验。对于网页浏览，浏览器的性能制约了内容发布者可以做的上限，在机器本身的限制范围内，需要将尽可能多的内存和处理器资源留给内容本身。同样的

原则页也适用于操作系统：内核需要把机器的资源尽可能多的留给用户程序，而不是被它们自身消耗。

但当我们说 Rust 很“快”的时候，到底是什么意思？一个人可以用任何通用语言写出非常慢的代码。更准确地说，如果你已经准备好认真设计你的程序来最大限度的利用底层机器的性能，那么 Rust 可以支撑你的目标。这门语言的效率很高，并且能给予你控制使用多少内存和 CPU 资源的能力。

1.4 Rust 使协作变得更简单

我们在标题中隐藏了第 4 条引言：“系统级程序员的福音”。这是在暗示 Rust 对代码共享和重用的支持。

Rust 的包管理器和构建工具 Cargo，使用户可能很容易的使用其他用户发布在 Rust 的公开仓库 cartes.io 上的库。你只需要简单的在一个文件中加上库的名字和版本号，cargo 将会自动下载库，和这个库的依赖，并把它们连接在一起。你可以将 Rust 的 Cargo 视为 NPM 或者 RubyGems 一类的东西，并强调完善的版本控制和可复制的构建。有很多流行的 Rust 库可以提供从序列化到 HTTP 客户端和服务器再到现代图形 API 等几乎任何功能。

进一步讲，这门语言本身就被设计为支持协作：Rust 的 trait 和泛型让你能创建出拥有灵活接口的库，它们可以在很多不同的上下文中工作。Rust 的标准库也提供了一组核心的基础类型，为常见的情况建立了共享的约定，使不同的库可以更容易的协同使用。

下一章旨在更具体的说明我们在这一章中提出的观点，我们通过几个小的 Rust 程序作为示例来展示这门语言的强大之处。

Chapter 2

Rust 概览

Rust 给像本书一样的书籍的作者提出了一个挑战：赋予这门语言特色的并不是可以在第一页就展示出来的某些惊人的特性，而是如何设计这门语言来让它的各个部分可以无缝的协同工作，最终达到我们在上一章提到的目标：安全、高性能的系统级编程。这门语言的每一部分都在其他所有部分中得到了最好的证明。

因此，相比于一次着眼于一种语言特性，我们选择了几个简单但却完整的程序作为概览，每一个程序都会涉及到一些语言特性：

- 作为热身，我们准备了一个简单的计算命令行参数的程序，以及相应的单元测试。这个程序展示了 Rust 的核心类型，并引入了 *trait*。
- 接下来，我们构建了一个 web 服务器。我们将会使用一个第三方库来处理 HTTP 的细节，并引入字符串处理、闭包、错误处理。
- 我们的第三个程序绘制了一个漂亮的图形，讲计算分布到多个线程来提高速度。这一部分包括一个泛型函数的示例，阐明了怎么处理类似于一个像素的概念，并展示了 Rust 对并发的支持。
- 最后，我们展示了一个使用正则表达式处理文件的健壮的命令行程序。这个程序展示了 Rust 标准库中处理文件的设施，和最常用的第三方正则表达式库。

Rust 保证在对代码的性能影响最小的情况下防止未定义行为，这一保证影响了整个 Rust 中每个部分的设计，从标准的数据结构例如 vector 和 string 到 Rust 程序员使用第三方库的方式都受此影响。这些具体的细节书中都会提到，但是现在，我们想向你展示 Rust 是一门强大且有趣的语言。

当然，首先你要在你的计算机上安装 Rust。

2.1 rustup和Cargo

安装Rust的最佳方式是使用rustup。访问https://rustup.rs并按照说明进行操作。

或者，你可以访问Rust网站来获取预构建好的Linux、macOS、Windows上的包。一些操作系统发行版里也包含Rust。我们推荐rustup，因为它是专用于管理Rust安装的工具，就像Ruby的RVM和Node的NVM一样。例如，当一个新版本的Rust发布时，你只需要输入rustup update就可以完成更新。

在任何情况下，完成了安装之后，你应该可以通过命令行访问以下三条新命令：

```
$ cargo --version
cargo 1.49.0 (d00d64df9 2020-12-05)
$ rustc --version
rustc 1.49.0 (e1884a8e3 2020-12-29)
$ rustdoc --version
rustdoc 1.49.0 (e1884a8e3 2020-12-29)
```

这里，$是命令提示符。在Windows上，可能是C:\>或者别的类似的。这里我们运行了安装的三条命令，查询它们的版本。接下来让我们依次讲解每一个命令：

- cargo是rust的编译管理器、包管理器和通用的工具。你可以使用Cargo来新建项目、构建并运行程序、管理所有代码中依赖的外部库。
- rustc是Rust的编译器。通常我们使用Cargo来调用编译器，但有时也需要直接运行它。
- rustdoc是Rust的文档工具。如果你在源代码中按照文档注释的格式写了文档，那么rustdoc可以通过它们构建出漂亮的HTML文档。和rustc一样，我们通常用Cargo来调用rustdoc。

方便起见，Cargo可以为我们创建新的Rust包，并设置好一些标准元数据：

```
$ cargo new hello
    Created binary (application) `hello` package
```

这个命令创建了一个叫做hello的新的包目录，并准备好构建一个可执行程序。

进入包的顶级目录并查看：

```
$ cd hello
$ ls -la
total 24
drwxrwxr-x.  4 jimb jimb 4096 Sep 22 21:09 .
```

```
drwx------. 62 jimb jimb 4096 Sep 22 21:09 ..
drwxrwxr-x.  6 jimb jimb 4096 Sep 22 21:09 .git
-rw-rw-r--.  1 jimb jimb    7 Sep 22 21:09 .gitignore
-rw-rw-r--.  1 jimb jimb   88 Sep 22 21:09 Cargo.toml
drwxrwxr-x.  2 jimb jimb 4096 Sep 22 21:09 src
```

我们可以看到Cargo创建了一个文件Cargo.toml来保存包的元数据。此时这个文件里还没有太多内容：

```
[package]
name = "hello"
version = "0.1.0"
authors = ["You <you@example.com>"]
edition = "2018"

# See more keys and their definitions at
# https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
```

如果我们的程序中需要依赖的库，我们可以在这个文件中添加它们，Cargo将会负责下载、构建和更新这些库。我们将在第8章中详细讲述Cargo.toml文件。

Cargo已经为我们的包初始化好了git版本控制系统，创建了一个.git元数据目录和一个.gitignore文件。你可以通过向cargo new命令传递--vcs none参数来跳过这一步。

src子目录包含了实际的Rust代码：

```
$ cd src
$ ls -l
total 4
-rw-rw-r--.  1 jimb jimb 45 Sep 22 21:09 main.rs
```

看起来Cargo好像已经替我们写好了程序。main.rs中包含以下文本：

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

在 Rust 中，你甚至不需要编写自己的“Hello, World!”程序，这是新的 Rust 程序的模板：两个文件，总共 13 行。

我们可以从包中的任何目录调用 `cargo run` 命令来构建并运行我们的程序：

```
$ cargo run
    Compiling hello v0.1.0 (/home/jimb/rust/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.28s
    Running `/home/jimb/rust/hello/target/debug/hello`
Hello, world!
```

这里，Cargo 调用了 Rust 的编译器 `rustc`，然后运行了它生成的可执行文件。Cargo 把可执行文件放在了顶层目录的 `target` 子目录下：

```
$ ls -l ../target/debug
total 580
drwxrwxr-x. 2 jimb jimb   4096 Sep 22 21:37 build
drwxrwxr-x. 2 jimb jimb   4096 Sep 22 21:37 deps
drwxrwxr-x. 2 jimb jimb   4096 Sep 22 21:37 examples
-rwxrwxr-x. 1 jimb jimb 576632 Sep 22 21:37 hello
-rw-rw-r--. 1 jimb jimb    198 Sep 22 21:37 hello.d
drwxrwxr-x. 2 jimb jimb     68 Sep 22 21:37 incremental
$ ../target/debug/hello
Hello, world!
```

如果需要的话，Cargo 可以为我们清理生成的文件：

```
$ cargo clean
$ ../target/debug/hello
bash: ../target/debug/hello: No such file or directory
```

2.2 Rust 函数

Rust 的语法借鉴自其他语言。如果你熟悉 C、C++、Java 或者 JavaScript，你可以很快找到自己的方式来理解 Rust 的程序结构。这里有一个使用欧几里得算法计算两个整数的最大公约数的函数。你可以把它添加到 `src/main.rs` 的最后：

```
fn gcd(mut n: u64, mut m: u64) -> u64 {
    assert!(n != 0 && m != 0);
```

```
        while m != 0 {
            if m < n {
                let t = m;
                m = n;
                n = t;
            }
            m = m % n;
        }
        n
    }
```

fn 关键字（读作“fun”）创建了一个函数。这里，我们定义了一个叫 gcd 的函数，它有两个参数 m 和 n，类型都是 u64，也就是 64 位无符号整数。-> 词元指明了返回值类型：我们的函数返回一个 u64 类型的值。四个空格缩进是 Rust 的标准风格。

Rust 的整数类型的名字代表了它们的大小和符号性：i32 是有符号 32 位整数；u8 是无符号 8 位整数（用于“字节”值）等等。isize 和 usize 类型分别代表可以存下一个指针的有符号和无符号整数，在 32 位平台上它们就是 32 位，在 64 位平台上就是 64 位。Rust 还有两种浮点数类型：f32 和 f64，分别是 IEEE 标准的单精度和双精度浮点数类型，类似于 C 和 C++ 中的 float 和 double。

默认情况下，当变量初始化后，它的值就不能再被改变，但通过在参数 m 和 n 前加上 mut 关键字（读作“mute”，*mutable* 的缩写）就可以在函数体中对它们进行赋值。在实践中，大多数变量都不会被重新赋值，在阅读代码时 mut 关键字将是一个有用的提示。

函数体中首先调用了 assert! 宏，确保两个参数都不是 0。! 字符标志着这是宏调用，而不是函数调用。类似于 C 和 C++ 中的 assert 宏，Rust 中的 assert! 宏也会检查参数是否为真，如果不为真则中断程序，并输出一条有用的信息，其中包括断言失败的源码位置。这种终止的方式被称为 *panic*。和 C 和 C++ 中断言可以被跳过不同，Rust 总是检查断言，不管程序怎么编译。还有一个 debug_assert! 宏，当程序被编译为 release 模式时会被跳过。

我们函数的主体是一个包含一条 if 语句和一条赋值语句的 while 循环。和 C 和 C++ 不同，Rust 的条件表达式不需要括号，但紧随其后的控制流语句需要花括号。

let 语句声明了一个局部变量，比如函数中的 t。我们不需要写出 t 的类型，因为 Rust 可以通过使用这个值的方式来推断它的类型。在我们的函数中，t 只有和 m、n 相匹配，是 u64 类型时才可以正常运行。Rust 只在函数体内推断类型：你必须写出函数参数和返回值的类型，就像我们所做的那样。如果你想指明 t 的类型，你可以写：

```
    let t: u64 = m;
```

Rsut有return语句，但是gcd函数并不需要。如果一个函数体以一个没有分号结尾的表达式结尾，那么这个表达式的值就是函数的返回值。事实上，任何一个花括号包围的语法块都可以作为一个表达式。例如，这里有一个表达式打印出一条消息，然后返回x.cos()作为它的值：

```
    {
        println!("evaluating cos x");
        x.cos()
    }
```

在Rust中当控制流到达函数底部时利用这种形式返回值是一种很典型的做法，只有当在函数的中途显式地返回时才会使用return语句。

2.3 编写并运行单元测试

Rust语言内建有对测试的支持。为了测试我们的gcd函数，我们可以在src/main.rs的最后添加下列代码：

```
    #[test]
    fn test_gcd() {
        assert_eq!(gcd(14, 15), 1);

        assert_eq!(gcd(2 * 3 * 5 * 11 * 17,
                       3 * 7 * 11 * 13 * 19),
                   3 * 11);
    }
```

这里我们定义了一个叫test_gcd的函数，它调用了gcd函数并检查返回值是否正确。喊函数上方的#[test]标记test_gcd是一个测试函数，这种函数在正常编译时会被跳过，但在使用cargo test命令时会被编译并自动调用。我们可以在整个源码树的任何位置定义测试函数，cargo test会自动收集它们并运行。

#[test]标记是属性的是一个示例。属性是一种为函数和其他声明标记额外信息的开放式系统，类似于C++和C#中的属性，或者Java中的注解。它们被用来控制编译器警告和代码风格检查、条件编译（类似于C和C++中的#ifdef）、告诉Rust怎么和其它语言编写的代码交互等。随着继续深入我们将会看到更多使用属性的例子。

把 gcd 和 test_gcd 函数的定义添加到 hello 包里之后，我们可以在包内的某个目录下按照如下方式运行测试：

```
$ cargo test
    Compiling hello v0.1.0 (/home/jimb/rust/hello)
     Finished test [unoptimized + debuginfo] target(s) in 0.35s
      Running /home/jimb/rust/hello/target/debug/deps/hello-2375a82d9e9673d7

running 1 test
test test_gcd ... ok

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

2.4 处理命令行参数

为了让我们的函数能获取一些作为命令行参数传入的数字并打印出他们的最大公约数，我们可以把 src/main.rs 中 main 函数的代码替换为如下内容：

```
use std::str::FromStr;
use std::env;

fn main() {
    let mut numbers = Vec::new();

    for arg in env::args().skip(1) {
        numbers.push(u64::from_str(&arg)
                     .expect("error parsing argument"));
    }

    if numbers.len() == 0 {
        eprintln!("Usage: gcd NUMBER ...");
        std::process::exit(1);
    }

    let mut d = numbers[0];
```

```
        for m in &numbers[1..] {
            d = gcd(d, *m);
        }

        println!("The greatest common divisor of {:?} is {}",   numbers, d);
    }
```

这是一个很大的代码块，让我们一步步来理解它：

```
use std::str::FromStr;
use std::env;
```

第一个 use 声明引入了标准库中的 FromStr *trait*。一个 trait 就是一些可以被实现的方法的集合。任何实现了 FromStr trait 的类型都有一个 from_str 方法，这个方法尝试把一个字符串解析为该类型。u64 类型实现了 FromStr，因此我们将调用 u64::from_str 来解析命令行参数。尽管我们在程序中并没有使用到 FromStr 这个名字，但为了使用这个 trait 的方法，必须将它引入作用域。我们将会在第 11 章讲解 trait。

第二个 use 声明引入了 std::env 模块，它提供了一些和运行环境进行交互的函数和类型，包括 args 函数，它可以让我们获取到程序的命令行参数。

接下来移步到程序的 main 函数：

```
fn main() {
```

我们的 main 函数不返回值，因此我们可以省略 -> 和返回类型。

```
let mut numbers = Vec::new();
```

我们声明了一个可变的局部变量 numbers，并将它初始化为一个空的向量。Vec 是 Rust 的可变长的向量类型，类似于 C++ 的 std::vector、Python 的 list、或者 JavaScript 的 array。即使 vector 被设计为动态伸缩，我们仍然需要将变量标记为 mut，这样 Rust 才允许我们向它的末尾添加元素。

numbers 的类型是 Vec<u64>，一个 u64 的 vector，但和之前一样，我们不需要写出类型。Rust 将会为我们推断出它的类型，在这里我们把 u64 类型的值添加到了 vector 里，而且我们之后还把这个 vector 的元素传给了 gcd 函数，gcd 函数的参数只能是 u64。

```
for arg in env::args().skip(1) {
```

这里我们使用了一个 for 循环来处理命令行参数，将每个参数命名为 arg 变量，然后执行循环体。

std::env 模块的 args 函数返回一个迭代器，迭代器可以惰性产生每一个值，并指示我们何时迭代结束。迭代器在 Rust 中无处不在，标准库中还包含其他的迭代器例如产生 vector 中的每个元素、产生文件的每一行、产生通道收到的每一条消息、以及其他几乎所有可以循环处理的东西。Rust 的迭代器非常高效：编译器通常能将它们翻译为和手写循环一样的代码。我们将会在第 15 章介绍该怎么使用它并给出一些示例。

除了和 for 循环一起使用之外，迭代器还有很多可以直接使用的方法。例如，args 方法返回的迭代器的第一个值总是正在运行的程序名。我们想要跳过它，因此我们调用了迭代器的 skip 方法来生成一个省略了第一个值的新迭代器。

```
numbers.push(u64.from_str(&arg)
             .expect("error parsing argument"));
```

这里我们调用了 u64::from_str 来尝试将命令行参数解析为 64 位整数。和通过 u64 类型的值调用的方法不同，u64::from_str 是一个和 u64 类型的值关联的方法，类似于 C++ 和 Java 中的静态方法。from_str 函数不直接返回一个 u64 类型的值，而是返回一个 Result 类型的值来表示解析是否成功。一个 Rusult 类型的值有两种可能：

- 一个写作 Ok(v)，表示解析成功，v 就是解析出的值。
- 另一个写作 Err(e)，表示解析失败，e 是解释失败的原因。

任何可能会失败的函数，例如输入输出或其他和操作系统交互的函数，都会返回 Result 值，Ok 时会携带成功的结果——读写的字节数、打开的文件等等——Err 时会携带错误码来指示错误的原因。和大多数现代编程语言不同，Rust 没有异常：所有的错误都通过 Rust 或者 panic 来处理，这会在第 7 章中介绍。

我们使用 Rust 的 expect 方法来检查解析是否成功。如果结果是 Err(e)，expect 会打印出包含 e 的描述错误的消息。如果结果是 Ok(v)，expect 会简单的返回 v，之后我们才能把它添加到 vector 的尾部。

```
if numbers.len() == 0 {
    eprintln!("Usage: gcd NUMBER ...");
    std::process::exit(1);
}
```

空的数字集合没有最大公约数，因此我们检查 vector 是否不为空，如果为空就退出程序。我们使用 eprintln! 宏来把错误信息写入到标准错误输出流。

```
let mut d = numbers[0];
for m in &numbers[1..] {
```

```
        d = gcd(d, *m);
    }
```

这一个循环使用了变量 d，将它更新为目前的最大公约数。和之前一样，我们将 d 标记为可变的，因此我们在循环里给它赋值。

这个 for 循环有两个特别的地方。一个是 `for m in &numbers[1..];`，操作符 & 是什么意思？另一个是 `gcd(d, *m);`，`*m` 里的 * 又是什么意思？事实上这两个细节是互补的。

到目前为止，我们的代码只操作过像整数这类固定内存大小的值。但是现在我们要迭代一个 vector，它的值可能是任意大小——有可能非常大。在处理这类值时 Rust 是很谨慎的：它想让程序员自己控制对内存的消耗、明确每个值的生命周期，同时确保当内存不再被需要时立即释放内存。

因此当我们在迭代时，我们想告诉 Rust 这个 vector 的所有权仍然属于 numbers，我们只是借用它的值来进行循环。`&numbers[1..]` 中的 & 运算符借用了 vector 中从第二个元素开始到最后一个元素的引用。for 循环迭代引用的那些元素，每次迭代中用 m 借用每一个元素。`*m` 中的 * 运算符解引用了 m，返回了它所指向的值，也就是我们传递给 gcd 的第二个值。最后，因为 numbers 拥有 vector 的所有权，当 numbers 离开 main 的作用域时 Rust 会自动释放它的内存。

Rust 对所有权和引用的规则时 Rust 的内存管理和安全并发的关键。我们将在第 4 章讨论他们，并在第 5 章讨论他们的伙伴。

要想舒服的使用 Rust，你必须习惯这些规则，但在这篇概览中，你只需要知道 `&x` 是借用 x 的引用，`*r` 返回引用 r 指向的值。

继续我们的程序：

```
    println!("The greatest common divisor of {:?} is {}", numbers, d);
```

迭代完 numbers 的元素之后，程序把结果打印到标准输出流。println! 宏接收一个模板字符串，用剩余参数替换掉模板字符串里的 `{...}`，并把结果写入到标准输出流。

与 C 和 C++ 中的 main 函数成功执行结束时要返回 0、执行失败时返回非 0 不同，Rust 假设不管 main 返回什么值都代表成功执行结束。只有当显式调用 expect 或 `std::process::exit` 之类的才会导致程序以错误的状态码终止。

cargo run 命令允许我们向程序传递参数，因此我们可以测试我们的命令行程序：

```
$ cargo run 42 56
   Compiling hello v0.1.0 (/home/jimb/rust/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.22s
     Running `/home/jimb/rust/hello/target/debug/hello 42 56`
```

```
The greatest common divisor of [42, 56] is 14
$ cargo run 799459 28823 27347
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `/home/jimb/rust/hello/target/debug/hello 799459 28823 27347`
The greatest common divisor of [799459, 28823, 27347] is 41
$ cargo run 83
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `/home/jimb/rust/hello/target/debug/hello 83`
The greatest common divisor of [83] is 83
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `/home/jimb/rust/hello/target/debug/hello`
Usage: gcd NUMBER ...
```

我们在这一节中用到了Rust标准库中的一小部分特性。如果你对其他的部分很好奇，我们强烈建议你去尝试Rust的在线文档。它的搜索功能很有用，甚至还包括到源代码的链接。当你安装Rust时`rustup`命令会自动在你的计算机上安装一份文档的拷贝。你可以在Rust的网站上查阅标准库的文档，或者通过如下命令在你的浏览器中查阅：

```
$ rustup doc --std
```

2.5 提供web页面

Rust的一个强项就是发布在网站crates.io上的可以自由使用的包。`cargo`命令让你可以方便的使用crates.io的包：它将会按需下载正确的版本，构建并在需要时更新它。一个Rust包，无论是库还是可执行文件，都被称为一个*crate*。Cargo和crates.io都是因为这个术语而得名。

为了展示它的工作方式，我们将会使用`actix-web`网络框架crate，`serde`序列化crate和其它它们依赖的crate来构建一个简单的web服务器。如图2-1所示，我们的网站将会提示用户输入两个数字，然后计算它们的最大公约数。

首先，我们需要使用Cargo创建一个新的包，名称为`actix-gcd`：

```
$ cargo new actix-gcd
    Created binary (application) `actix-gcd' package
$ cd actix-gcd
```

然后，我们将编辑项目中的`Cargo.toml`文件来列举出我们需要的包，它的内容如下所示：

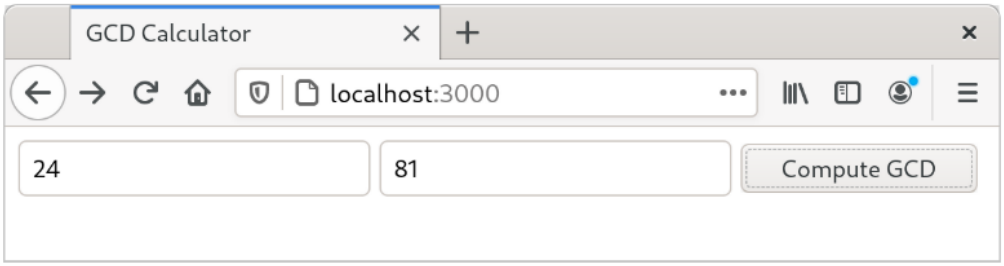

Figure 2.1: 提供计算最大公约数功能的 web 页面

```
[package]
name = "actix-gcd"
version = "0.1.0"
authors = ["You <you@example.com>"]
edition = "2018"

# See more keys and their definitions at
# https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
actix-web = "1.0.8"
serde = { version = "1.0", features = ["derive"] }
```

`Cargo.toml` 中 `dependencies` 节的每一行都有一个 cartes.io 上的 crate 的名字和需要使用的版本。在这个例子中，我们需要 `actix-web` crate 的 `1.0.8` 版本和 `serde` crate 的 `1.0` 版本。crates.io 上这两个 crate 可能还有更新的版本，但是通过指定我们测试成功的特定版本，可以保证即使这两个 crate 发布了新的版本，代码仍然可以正常工作。我们将会在第 8 章中详细的讨论版本管理。

crate 有一些可选的特性：这些特性是有些用户可能用不到、但仍然需要包含在 crate 中的部分接口或实现。`serde` 提供了一个非常简洁的方式来处理 web 表单中的数据，但是根据 `serde` 的文档，只有当我们选择了这个 crate 的 `derive` 特性，才可以使用这种方式，因此我们在 `Cargo.toml` 文件中制定了这个特性。

注意我们只需要指明那些我们直接使用的 crate，`cargo` 会自动下载它们依赖的其他 crate。

在我们的第一个版本中，我们将保持 web 服务器的简洁：它将只提供一个页面，提示用户输入要计算的数字。将 `actix-gcd/src/main.rs` 中的内容替换为如下：

```
use actix_web::{web, App, HttpResponse, HttpServer};

fn main() {
    let server = HttpServer::new( || {
        App::new()
            .route("/", web::get().to(get_index))
    });

    println!("Servering on http://localhost:3000...");
    server
        .bind("127.0.0.1:3000").expect("error binding server to     address")
        .run().expect("error running server");

}

fn get_index() -> HttpResponse {
    HttpResponse::Ok()
        .content_type("text/html")
        .body(
            r#"
                <title>GCD Calculator</title>
                <form action="/gcd" method="post">
                <input type="text" name="n"/>
                <input type="text" name="m"/>
                <button type="submit">Compute GCD</button>
                </form>
            "#
        )
}
```

我们以一条 use 声明开始，引入一些 actix-web 的定义。当我们写下 use actix_web::{...} 时，花括号里的每一个名称都被导入到作用域中，这样我们就可以直接使用 HttpResponse，而不用每次都写出全名 actix_web::HttpResponse。（我们稍后才会使用 serde crate）

我们的main函数很简单：它首先调用HttpServer::new来创建一个服务器，这个服务器会响应对"/"路径的get方法；然后打印出一条消息；最后让服务器监听本地机器上的3000 TCP端口。

我们传递给HttpServer::new的参数是Rust的闭包表达式|| { App::new() ... }。闭包是一种可以被当作函数来调用的值。这里定义的闭包没有参数，但如果需要参数的话，要写在||之间。{ ... }是闭包的函数体。当我们启动服务器时，Actix会启动一个线程池来处理到达的请求。每一个线程都会调用我们的闭包来获取一个App的拷贝，App的值将告诉它如何路由和处理请求。

闭包会调用App::new来创建一个新的、空的App并调用它的route方法来添加一个路径"/"的路由。用web::get().to(get_index)为这个路由添加处理函数，作用是遇到HTTP GET请求时调用函数get_index。route方法会返回调用它的App自身，并增加新的路由。因为闭包的结尾处没有分号，因此App就是闭包的返回值，HttpServer线程将会使用它。

get_index函数构建了一个HttpResponse类型的值作为HTTP GET /请求的响应。HttpResponse::Ok()表HTTP 200 OK状态，表示请求被成功处理。我们还调用了它的content_type和body方法来填充相应的细节；每一个调用都会返回调用它们的HttpResponse。最后，body方法的返回值作为get_index的返回值。

因为相应的文本中包含很多双引号，所以我们使用了Rust的raw string语法：字母r、0个或多个井号（#字符）、一个双引号，string的内容、最后以另一个双引号加上和开头处相同数量的井号结尾。raw string中出现的任何字符都不会被转义，包括双引号；事实上，没有转移的序列例如\"也可以被识别。我们可以通过增多井号的数量来保证终止的标记不会出现在字符串内容中。

编写完main.rs之后，我们可以使用carto run命令来运行它：它会自动获取所需的crate、编译它们、编译我们自己的程序、并链接在一起、然后启动它：

```
$ cargo run
    Updating crates.io index
 Downloading crates ...
  Downloaded serde v1.0.100
  Downloaded actix-web v1.0.8
  Downloaded serde_derive v1.0.100
...
  Compiling serde_json v1.0.40
  Compiling actix-router v0.1.5
```

```
   Compiling actix-http v0.2.10
   Compiling awc v0.2.7
   Compiling actix-web v1.0.8
   Compiling gcd v0.1.0 (/home/jimb/rust/actix-gcd)
    Finished dev [unoptimized + debuginfo] target(s) in 1m 24s
     Running `/home/jimb/rust/actix-gcd/target/debug/actix-gcd`
Serving on http://localhost:3000...
```

到目前为止，我们可以可以访问给定的URL并看到如图2-1所示的页面。

不幸的是，点击GCD并不会做任何事，而且会把我们的浏览器导航到一个空页面。接下来让我们来修复它，通过向我们的App添加另一个路由来处理我们表单的POST请求的响应。

终于到了使用我们在Cargo.toml中列出的serde crate的时候了：它提供了一种帮助我们处理表单数据的简单方法。首先，我们需要在src/main.rs中添加如下use声明：

```
use serde::Deserialie;
```

Rust程序员通常会把use声明集中在文件开始处，但这并不是必须的：Rust允许use声明以任何顺序出现，只要它们出现在正确的嵌套层级。

接下来让我们定义一个Rust结构体类型来表示我们希望从表单中接收到的数据：

```
#[derive(Deserialize)]
struct GcdParameters {
    n: u64,
    m: u64,
}
```

这里定义了一个新的类型叫做GcdParameters，它有两个字段n和m，都是u64类型，和gcd函数的参数类型保持一致。

struct定义上方的注解是一个属性，类似于我们之前用来标记为测试函数时使用的#[test]属性。在一个类型定义前加上#[derive(Deserialize)]属性可以告诉serde crate在程序编译时自动为该类型生成代码来把HTML POST请求的表单数据转为该类型的值。事实上，这个属性可以让你从几乎所有结构化的数据：JSON、YAML、TOML或其他文本或二进制格式中解析出一个GcdParameters类型的值。serde crate还提供一个Serialize属性来做相反的事，即把Rust值写成结构化的格式。

有了上面的定义，我们可以简单的写出我们的处理函数：

```
fn post_gcd(form: web::Form<GcdParameters>) -> HttpResponse {
    if form.n == 0 || form.m == 0 {
        return HttpResponse::BadRequest()
            .content_type("text/html")
            .body("Computing the GCD with zero is boring.");
    }

    let response =
        format!("The greatest common divisor of the numbers {} and {} \
                 is <b>{}</b>\n",
                form.n, form.m, gcd(form.n, form.m));

    HttpResponse::Ok()
        .content_type("text/html")
        .body(response)
}
```

作为 Actix 请求的处理函数，它的参数的类型必须是 Actix 知道怎么从中提取出 HTTP 请求的类型。我们的 `post_gcd` 函数有一个参数 `form`，它的类型是 `web::Form<GcdParameters>`。当且仅当 `T` 可以从 HTML 的 `POST` 表单数据反序列化出来时，Actix 才知道怎么 `web::Form<T>` 类型中提取出值。因为我们在 `GcdParameters` 类型的定义前加上了 `#[derive(Deserialize)]` 属性，所以 Actix 可以从表单数据中反序列化出它，因此请求的处理函数可以使用 `web::Form<GcdParameters>` 作为参数。这些类型和函数之间的关系都是在编译期处理的，如果你用了 Actix 不知道该如何处理的类型作为参数的类型，Rust 编译器将会立刻告诉你这个错误。

再看 `post_gcd` 的实现：如果有参数的值为 0，这个函数会返回一个 HTTP `401 BAD REQUEST` 错误，因为这种情况下我们的 `gcd` 函数会 panic。然后，它使用 `format!` 宏创建了一个响应。`format!` 宏类似于 `println!` 宏，只不过它不会把字符串写入到标准输出，而是会返回字符串。当响应的文本就绪后，`post_gcd` 用一个 HTTP `200 OK` 响应来包装它，设置好 content type 之后，就把它返回到请求方。

我们还要把 `post_gcd` 注册为表单的处理函数。我们要把 `main` 函数替换为如下内容：

```
fn main() {
    let server = HttpServer::new(|| {
        App::new()
```

```
            .route("/", web::get().to(get_index))
            .route("/gcd", web::post().to(post_gcd))
    });

    println!("Servering on http://localhost:3000...");
    server
        .bind("127.0.0.1:3000").expect("error binding server to address")
        .run().expect("error running server");
}
```

唯一的变化在于多了一个 `route` 的调用，把 `web::post().to(post_gcd)` 注册为路径 `"/gcd"` 的 handler。

最后剩余的部分是我们之前编写的 `gcd` 函数，要把它添加到 `actix-gcd/src/main.rs` 文件中。完成这些之后，你可以停止之前运行的服务器并重新启动程序：

```
$ cargo run
Compiling actix-gcd v0.1.0 (/home/jimb/rust/actix-gcd)
 Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
  Running `target/debug/actix-gcd`
Serving on http://localhost:3000...
```

这一次，通过访问 *http://localhost:3000*，输入一些数字，然后点击 Compute GCD 按钮，你应该能看到如下的结果（图 2-2）。

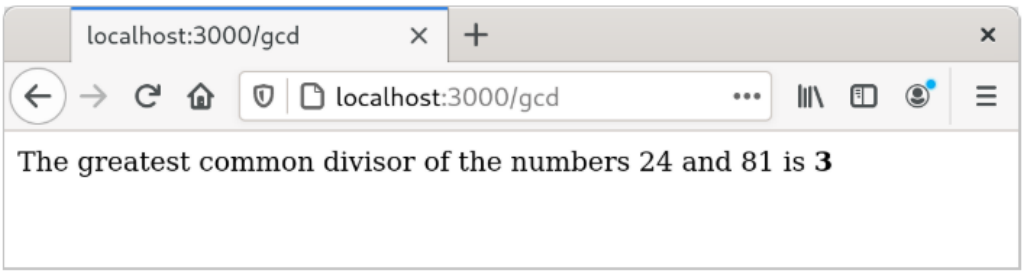

Figure 2.2: 显示计算 GCD 结果的 web 页面

2.6 并发

Rust 的另一项长处是对并发编程的支持。Rust 中用于避免内存安全问题的规则同样可以保证线程之间在没有数据竞争的情况下共享数据。例如：

- 如果你使用自旋锁来同步需要修改共享数据结构的线程，Rust 保证只有当你持有锁的情况下才可以访问数据，并且当你处理完数据后自动释放锁。在 C 和 C++ 中，自旋锁和要保护的数据之间的关系一般都写在注释里。
- 如果你想在几个线程中共享只读数据，Rust 保证你不可能意外修改数据。在 C 和 C++ 中，类型系统也可以帮我们做到这一点，但很容易出错。
- 如果你把一个数据结构的所有权从一个线程转移到另一个线程，Rust 保证你确实已经失去了对它的访问权。在 C 和 C++ 中，需要由你自己来保证发送线程不会再次访问该数据。

如果你没有正确做到这些，那么结果将会却决于处理器的缓存和你最近写入了多少内存。

在这一节中，我们将带领你编写你的第二个多线程程序。

你已经编写过第一个多线程程序了：你用来实现最大公约数服务器的 Actix web 框架使用了线程池来运行请求的处理函数。如果服务器同时收到很多请求，它会立刻在若干个线程里运行 `get_form` 和 `post_gcd` 函数。这可能让我们有些震惊，因为当我们编写那些函数时我们完全没有并发的概念。

不过 Rust 保证这么做是安全的，不管你的服务器变得多复杂：只要你的程序能够编译，它就能免于数据竞争。所有的 Rust 函数都是线程安全的。

这一节的程序将绘制曼德勃罗集，它是一种通过迭代一个简单的复数函数得到的分形。绘制曼德勃罗集经常被称为 *embarrassingly parallel* 算法，因为线程之间的通信太过简单；我们将会在第 19 章中讲述更为复杂的模式，但这个例子已经可以展示出一些核心的部分。

开始之前，我们要创建一个新的 Rust 项目：

```
$ cargo new mandelbrot
     Created binary (application) `mandelbrot` package
$ cd mandelbrot
```

所有的代码都会添加到 `mandelbrot/src/main.rs` 里，我们将会把一些依赖添加到 `mandelbrot/Cargo.t`

在开始实现并行的曼德勃罗集之前，我们需要先描述一下我们准备实现的计算过程。

2.6.1 曼德勃罗集到底是什么

了解这一点可以帮我们在阅读代码时更加清楚它要做什么，因此首先让我们先探讨一些纯数学只是。我们将以一个简单的例子开始，并逐渐添加复杂的细节，直到我们讲到曼德勃罗集的核心。

首先这里有一个无限循环，用 Rust 的语法来写的话就是一个 `loop` 语句：

```
fn square_loop(mut x: f64) {
    loop {
```

```
            x = x * x;
        }
    }
```

在现实中，Rust 可能会看出 x 的值从来没有被使用过，因此并不执行计算。但一开始，首先让我们假设代码按照我们所写的运行。x 的值将会发生什么变化？任何小于 1 的数平方都会变得更小，因此它会接近于 0；1 的平方还是 1；大于 1 的数平方会变得更大，因此它会接近无限大；负数的平方将会使它变为整数，然后它的变化就和前面说的一样（图 2-3）。

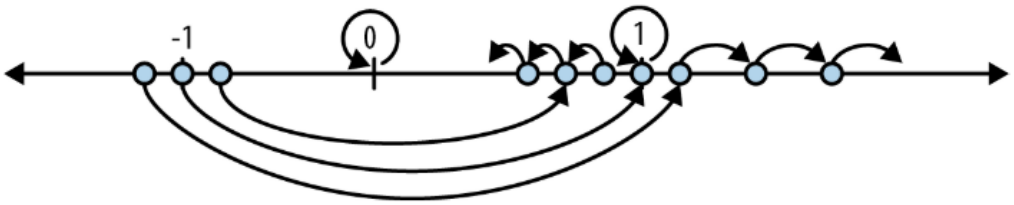

Figure 2.3: 重复平方一个数的结果

因此根据传递给 square_loop 的值不同，x 可能会保持 0 或者 1 不变，也可能接近 0 或者接近无限大。

现在让我们考虑一个稍有不同的循环：

```
fn square_add_loop(c: f64) {
    let mut x = 0.;
    loop {
        x = x * x + c;
    }
}
```

这一次，x 从 0 开始，并且每次迭代时平方之后再加上 c。这导致我们更难看出 x 会怎么变化，但一些实验表明如果 c 大于 0.25 或者小于-2.0，那么 x 将会变得接近无穷大；否则它会保持在接近 0 的某个区间。

更进一步，如果不用 f64 类型的值，考虑使用复数来执行相同的循环。crates.io 上的 num crate 提供了一个我们可以使用的复数类型，因此我们必须在我们程序的 Cargo.toml 文件的 [dependencies] 节中添加一行对 num 的引用。这是到目前为止这个文件里的全部内容（稍后我们还会添加一些内容）：

```
[package]
name = "mandelbrot"
```

```
version = "0.1.0"
authors = ["You <you@example>"]
edition = "2018"

# See more keys and their definitions at
# https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
num = "0.4"
```

现在我们可以写出循环的倒数第二个版本：

```
use num::Complex;

fn complex_square_add_loop(c: Complex<f64>) {
    let mut z = Complex { re: 0.0, im: 0.0};
    loop {
        z = z * z + c;
    }
}
```

用 z 表示复数是一种习惯，因此我们重命名了变量。表达式 `Complex { re: 0.0, im: 0.0 }` 创建了一个 num crate 的 `Complex` 类型的复数 0 值。`Complex` 时一个 Rust 的结构体类型，定义类似于如下：

```
struct Complex<T> {
    /// Real portion of the complex number
    re: T,

    /// Imaginary portion of the complex number
    im: T,
}
```

上面的代码定义了一个叫 `Complex` 的结构体，有两个字段：`re` 和 `im`。`Complex` 是一个泛型结构体：你可以将类型名后的 `<T>` 理解为“任意类型 `T`”。例如，`Complex<f64>` 是一个 `re` 和 `im` 字段都是 `f64` 类型的复数，`Complex<f32>` 则是 32 位浮点数，等等。有了这个定义之后，类似

于 `Complex { re: 0.24, im: 0.3 }` 这样的表达式将会产生一个 `re` 字段初始化为 0.24、`im` 字段初始化为 0.3 的 `Complex` 类型的值。

Chapter 3

基本类型

Chapter 4

所有权和 move

Chapter 5

引用

Chapter 6

表达式

Chapter 7

错误处理

Chapter 8

crate 和模块

Chapter 9

结构体

Chapter 10

枚举和模式

Chapter 11

trait 和泛型

Chapter 12

运算符重载

Chapter 13

工具 trait

Chapter 14

闭包

Chapter 15

迭代器

Chapter 16

集合

Chapter 17

字符串和文本

Chapter 18

输入输出

Chapter 19

并发

Chapter 20

异步编程

Chapter 21

宏

Chapter 22

unsafe 代码

Chapter 23

外部函数