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

基本类型

Chapter 3

所有权和 move

Chapter 4

引用

Chapter 5

表达式

Chapter 6

错误处理

Chapter 7

crate 和模块

Chapter 8

结构体

Chapter 9

枚举和模式

Chapter 10

trait 和泛型

Chapter 11

运算符重载

Chapter 12

工具 trait

Chapter 13

闭包

Chapter 14

迭代器

Chapter 15

集合

Chapter 16

字符串和文本

Chapter 17

输入输出

Chapter 18

并发

Chapter 19

异步编程

Chapter 20

宏

Chapter 21

unsafe 代码

Chapter 22

外部函数