Rust 程序设计
第二版

Jim Blandy, Jason Orendorff, and Leonora F.S. Tindall

目 录

0. 前言

Rust 是一种用于系统编程的语言。 现在这有一些解释，因为系统编程对大多数工作的程序员来说都不熟悉。 然而，它是我们所做一切的基础。 你合上笔记本电脑。 操作系统检测到这一点，暂停所有正在运行的程序，关闭屏幕，并让计算机进入睡眠状态。 稍后，您打开笔记本电脑：屏幕和其他组件再次启动，每个程序都可以从中断处继续运行。 我们认为这是理所当然的。 但是系统程序员编写了大量代码来实现这一点。 系统编程是为了：

- 操作系统
- 各种设备驱动程序
- 文件系统
- 数据库
- 在非常便宜的设备或必须极其可靠的设备上运行的代码
- 密码学
- 媒体编解码器（用于读取和写入音频、视频和图像文件的软件）
- 媒体处理（例如，语音识别或照片编辑软件）
- 内存管理（例如，实现垃圾收集器）
- 文本渲染（将文本和字体转换为像素）
- 实现高级编程语言（如 JavaScript 和 Python）
- 联网
- 虚拟化和软件容器
- 科学仿真
- 游戏

简而言之，系统编程是资源受限的编程。 当每个字节和每个 CPU 周期都很重要时，使用系统编程。

支持基本应用程序所涉及的系统代码量是惊人的。

本书不会教您系统编程。 事实上，本书涵盖了内存管理的许多细节，如果您还没有自己进行过一些系统编程，那么这些细节乍一看可能显得不必要的深奥。 但是，如果您是一位经验丰富的系统程序员，您会发现 Rust 与众不同：一种新工具可以消除困扰整个行业数十年的重大、广为人知的问题。

0.1. 谁应该读这本书

如果您已经是一名系统程序员并且准备好使用 C++ 的替代品，那么本书适合您。 如果您是任何编程语言的经验丰富的开发人员，无论是 C#、Java、Python、JavaScript 还是其他语言，本书也适合您。

但是，您不仅需要学习 Rust。 要充分利用该语言，您还需要获得一些系统编程经验。 我们建议在阅读本书的同时使用 Rust 实现一些系统编程方面的项目。 构建您以前从未构建过的东西，利用 Rust 的速度、并发性和安全性。 本序言开头的主题列表应该会给您一些想法。

0.2. 我们为什么写这本书

当我们开始学习 Rust 时，我们开始着手编写我们希望拥有的书。 我们的目标是预先和正面处理 Rust 中的大的、新的概念，清晰而深入地呈现它们，以最大限度地减少通过反复试验学习。

0.3. 本书各章节内容摘要

本书的前两章介绍了 Rust，并在我们继续第 3 章的基本数据类型之前提供了一个简短的浏览。第 4 章和第 5 章讨论了所有权和引用的核心概念。 我们建议您按顺序通读前五章。

第 6 章到第 10 章涵盖了该语言的基础知识：表达式（第 6 章）、错误处理（第 7 章）、包和模块（第 8 章）、结构体（第 9 章）以及枚举和模式（第 10 章）。 在这里略读一下是可以的，但是不要跳过关于错误处理的章节。

第 11 章涵盖特征和泛型，这是您需要了解的最后两个重要概念。 特征就像 Java 或 C# 中的接口。 它们也是 Rust 支持将类型集成到语言本身的主要方式。 第 12 章展示了特征如何支持运算符重载，第 13 章涵盖了更多实用特征。

了解特征和泛型可以打开本书的其余部分。 闭包和迭代器是你不想错过的两个重要的强大工具，分别在第 14 章和第 15 章中介绍。 您可以按任何顺序阅读其余章节，或根据需要深入阅读。 它们涵盖了语言的其余部分：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）、不安全代码（第 22 章），以及用其他语言调用函数（第 23 章）。

0.4. 本书中使用的约定

本书使用以下排版约定：

Italic

表示新术语、URL、电子邮件地址、文件名和文件扩展名。

`Constant width`

用于程序列表，以及在段落中引用程序元素，例如变量或函数名称、数据库、数据类型、环境变量、语句和关键字。

`Constant width bold`

显示应由用户逐字输入的命令或其他文本。

`Constant width italic`

显示应替换为用户提供的值或由上下文确定的值的文本。

> NOTE
>
> 此图标表示一般注释。

0.5. 代码示例

补充材料（代码示例、练习等）可在 https://github.com/ProgrammingRust 下载。 本书旨在帮助您完成工作。 一般来说，如果本书提供了示例代码，您可以在您的程序和文档中使用它。 除非您要复制代码的重要部分，否则无需联系我们获得许可。 例如，编写一个使用本书中几段代码的程序不需要许可。 销售或分发 O’Reilly 图书中的示例需要获得许可。 通过引用本书和引用示例代码来回答问题不需要许可。 将本书中的大量示例代码合并到您的产品文档中确实需要获得许可。

我们感谢但不要求署名。 署名通常包括书名、作者、出版商和 ISBN。 例如：“Programming Rust, Second Edition by Jim Blandy, Jason Orendorff, and Leonora F.S. Tindall (O’Reilly). Copyright 20 21 Jim Blandy, Leonora F.S. Tindall, and Jason Orendorff, 978-1-492-05259-3.”

如果您觉得您对代码示例的使用不属于合理使用或上述许可范围，请随时通过 permissions@oreilly.com 与我们联系。

1. 系统程序员可以拥有美好的事物

在某些情况下——例如 Rust 的目标——比竞争对手快 10 倍甚至 2 倍是一个成败攸关的事情。 它决定了一个系统在市场上的命运，就像在硬件市场上一样。

—Graydon Hoare

现在所有的计算机都是并行的...
并行编程就是编程。

—Michael McCool et al., Structured Parallel Programming

民族国家攻击者通过 TrueType 解析器缺陷实现监视目的； 所有软件都是安全敏感的。

—Andy Wingo

我们选择用上面的三个引用打开我们的书是有原因的。 但让我们从一个谜开始。 下面的 C 程序是做什么的？

```
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，这个程序打印出：

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

然后它崩溃了。 如果您在您的机器上尝试它，它可能会做其他事情。 这里发生了什么？

该程序存在缺陷。 数组 `a` 只有一个元素长，所以根据 C 编程语言标准，使用 `a[3]` 是未定义的行为：

“使用不可移植或错误的程序构造或错误数据时的行为，本国际标准对此没有强加要求”

未定义的行为不仅会产生不可预测的结果：标准明确允许程序做任何事情。 在我们的例子中，将这个特定值存储在这个特定数组的第四个元素中恰好会破坏函数调用堆栈，这样从 main 函数返回，而不是像它应该的那样优雅地退出程序，跳转到标准代码的中间用于从用户主目录中的文件中检索密码的 C 库。 它并不顺利。

C 和 C++ 有数百条规则来避免未定义的行为。 它们大多是常识：不要访问不该访问的内存，不要让算术运算溢出，不要除以零，等等。 但是编译器并不强制执行这些规则； 它没有义务发现哪怕是公然的违规行为。 事实上，前面的程序编译时没有错误或警告。 避免未定义行为的责任完全落在你身上，程序员。

从经验上讲，我们程序员在这方面并没有很好的记录。 在犹他大学学习期间，研究人员 Peng Li 修改了 C 和 C++ 编译器，使他们翻译的程序能够报告它们是否执行了某些形式的未定义行为。 他发现几乎所有的程序都这样做，包括那些将代码保持在高标准的备受推崇的项目。 假设您可以避免 C 和 C++ 中的未定义行为就像假设您可以仅仅因为知道规则就可以赢得一场象棋比赛。

偶尔出现的奇怪消息或崩溃可能是质量问题，但自从 1988 年的莫里斯蠕虫病毒在早期 Internet 上从一台计算机传播到另一台计算机以来，无意的未定义行为也一直是安全漏洞的主要原因。

因此 C 和 C++ 将程序员置于一个尴尬的境地：这些语言是系统编程的行业标准，但它们对程序员的要求几乎保证了源源不断的崩溃和安全问题。 解开我们的谜团只会提出一个更大的问题：我们不能做得更好吗？

1.1. Rust 为您肩负重担

我们的答案由我们的三个开场白引述。 第三条引述是关于 2010 年发现侵入工业控制设备的计算机蠕虫病毒 Stuxnet 的报道，它使用许多其他技术控制了受害者的计算机，其中包括解析嵌入在文字处理文档中的 TrueType 字体的代码中的未定义行为。 可以肯定的是，该代码的作者不希望以这种方式使用它，这表明需要担心安全性的不仅仅是操作系统和服务器：任何可能处理来自不受信任来源的数据的软件都可能成为攻击的目标。

Rust 语言给你一个简单的承诺：如果你的程序通过了编译器的检查，它就没有未定义的行为。 悬挂指针、二次释放和空指针解引用都在编译时被捕获。 数组引用通过编译时和运行时检查的混合来保护，因此没有缓冲区溢出：我们不幸的 C 程序的 Rust 等价代码可以安全退出并显示错误消息。

此外，Rust 的目标是既安全又易于使用。 为了对你的程序的行为做出更有力的保证，Rust 对你的代码施加了比 C 和 C++ 更多的限制，这些限制需要实践和经验才能习惯。 但整体语言是灵活和富有表现力的。 用 Rust 编写的代码的广度及其应用的应用领域范围证明了这一点。

根据我们的经验，能够相信该语言能够发现更多错误会鼓励我们尝试更有雄心的项目。 当您知道内存管理和指针有效性问题已得到处理时，修改大型复杂程序的风险就会降低。 当错误的潜在后果不包括破坏程序的不相关部分时，调试就会简单得多。

当然，还有很多 Rust 无法检测到的错误。 但在实践中，将未定义的行为从桌面上移除会大大改变开发的特征，使其变得更好。

1.2. 并行编程被驯服

众所周知，并发很难在 C 和 C++ 中正确使用。 开发人员通常只有在证明单线程代码无法达到他们需要的性能时才转向并发。 但是第二个开场白指出，并行性对于现代机器来说太重要了，不能将其视为最后的手段。

事实证明，在 Rust 中确保内存安全的相同限制也确保 Rust 程序没有数据竞争。 您可以在线程之间自由共享数据，只要它不发生变化。 确实发生变化的数据只能使用同步原语访问。 所有传统的并发工具都可用：互斥锁、条件变量、通道、原子等。 Rust 只是检查你是否正确地使用了它们。

这使得 Rust 成为开发现代多核机器能力的优秀语言。 Rust 生态系统提供的库超越了通常的并发原语，可帮助您在处理器池之间均匀分布复杂负载，使用无锁同步机制（如读取-复制-更新等）。

1.3. 然而 Rust 仍然很快

最后，这是我们的第一个开场白。 Rust 分享了 Bjarne Stroustrup 在他的论文“抽象和 C++ 机器模型”中阐明的 C++ 的雄心：

“通常，C++ 实现遵循零开销原则：不使用的东西，不需要付费。 更进一步：你所使用的，你无法编写更好的代码。”

系统编程通常涉及将机器推向极限。 对于电子游戏来说，整机应该致力于为玩家创造最好的体验。 对于网络浏览器，浏览器的效率限制了内容作者的能力。 在机器的固有限制内，必须将尽可能多的内存和处理器注意力留给内容本身。 同样的原则也适用于操作系统：内核应该让机器的资源可供用户程序使用，而不是自己消耗它们。

但是当我们说 Rust 是“快”的时候，那到底是什么意思呢？ 可以用任何通用语言编写慢速代码。 更准确地说，如果您准备投资设计您的程序以充分利用底层机器的功能，Rust 会支持您的努力。 该语言的设计具有高效的默认值，使您能够控制内存的使用方式以及处理器注意力的分配方式。

1.4. Rust 使协作更容易

我们在本章的标题中隐藏了第四句话：“系统程序员可以拥有美好的事物。” 这是指 Rust 对代码共享和重用的支持。

Rust 的包管理器和构建工具 Cargo 可以轻松使用其他人在 Rust 的公共包存储库 crates.io 网站上发布的库。 您只需将库的名称和所需的版本号添加到一个文件中，Cargo 就会负责下载该库以及它依次使用的任何其他库，并将所有内容链接在一起。 您可以将 Cargo 视为 Rust 对 NPM 或 RubyGems 的回应，强调健全的版本管理和可重现的构建。 流行的 Rust 库提供了从现成的序列化到 HTTP 客户端和服务器以及现代图形 API 的一切。

更进一步，该语言本身也旨在支持协作：Rust 的特性和泛型让您可以创建具有灵活接口的库，以便它们可以在许多不同的上下文中使用。 Rust 的标准库提供了一组核心的基本类型，这些类型为常见情况建立了共享约定，使不同的库更容易一起使用。

下一章旨在具体化我们在本章中提出的广泛主张，并浏览几个展示该语言优势的小型 Rust 程序。

2. Rust 之旅