Web 开发与设计

Rust Web开发

[德] 巴斯蒂安·格鲁伯(Bastian Gruber) 著

赵 永 邹松廷 卢贤泼 译

清華大學出版社

北 京

北京市版权局著作权合同登记号 图字：01-2024-0791

Bastian Gruber
Rust Web Development
EISBN: 9781617299001

图书在版编目(CIP)数据

Rust Web 开发 / (德) 巴斯蒂安 • 格鲁伯(Bastian Gruber) 著；赵永，邹松廷，卢贤泼译. —北京：清华大学出版社，2024.4
(Web 开发与设计)
书名原文：Rust Web Development
ISBN 978-7-302-65823-8

Ⅰ. ①R… Ⅱ. ①巴… ②赵… ③邹… ④卢… Ⅲ. ①程序语言—程序设计 Ⅳ. ①TP312

中国国家版本馆 CIP 数据核字(2024)第 060616 号

责任编辑：王 军 刘远菁
装帧设计：孔祥峰
责任校对：成风进
责任印制：

出版发行：清华大学出版社
网 址：https://www.tup.com.cn，https://www.wqxuetang.com
地 址：北京清华大学学研大厦 A 座 邮 编：100084
社 总 机：010-83470000 邮 购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经 销：全国新华书店
开 本：170mm×240mm 印 张：22.5 字 数：520 千字
版 次：2024 年 4 月第 1 版 印 次：2024 年 4 月第 1 次印刷
定 价：98.00 元

产品编号：099490-01

推 荐 序

《Rust Web 开发》由 Bastian Gruber 撰写，旨在为读者提供从头到尾编写 Web 应用程序的全面指导。作者不仅详细介绍了 Rust 语言的基础知识，还深入探讨了如何使用 Rust 构建高效、安全的 Web 服务。书中涵盖了 Rust 编程的各个方面，包括异步编程、数据库连接、集成第三方 API 等关键技术点，以及如何部署和测试 Rust Web 应用程序。

本书特别适合那些对 Rust 编程有一定的了解，且希望深入探索如何利用 Rust 开发 Web 服务的开发者。通过一系列实用的示例和讲解，读者可以学习到如何利用 Rust 的性能优势和安全特性来构建下一代 Web 应用程序和 API。

经过仔细审阅《Rust Web 开发》一书的试读章节，我发现这本书并非只是一本技术手册，而是一本充满洞见和实用知识的宝典，它为 Rust 编程语言在 Web 开发领域的应用提供了全新的视角。作者通过丰富的示例和实战经验，使得读者能够快速掌握 Rust 在 Web 开发中的实际应用，无论是新手还是有经验的开发者，都能从中获益。

书中的内容安排合理，以 Web 开发为主线，从基本的路由函数开始，到日志调试，再到添加数据库和集成第三方 API，最后将应用程序部署到生产环境中。每一个章节都紧密围绕着实际的项目需求展开，不仅提供了代码示例，还深入讲解了背后的语言特性、设计理念和最佳实践。这种由浅入深的教学方法，旨在帮助读者构建坚实的知识基础，同时激发读者探索更高级话题的兴趣。

特别值得一提的是，这本书虽然看上去用到了一些特定的 Web 框架，但是其内容却超越了框架，读者能够学到更多 Rust 基础知识。

总而言之，《Rust Web 开发》是一本既适合初学者，又能让有经验的开发者深入理解的书籍。它不仅提供了关于如何使用 Rust 进行 Web 开发的详细指导，还展现了 Rust 在现代 Web 开发中的巨大潜力。无论你是 Rust 的初学者，还是希望将 Rust 应用于 Web 开发的资深程序员，这本书都将是你不可多得的资源。强烈建议所有对 Web 开发和 Rust 语言感兴趣的读者阅读本书，相信它将成为你技术书架上的一份珍贵财富。

张汉东
资深独立咨询师
《Rust 编程之道》作者
Rust 中文社区布道者

序　言

我骨子里是一个实用主义者。我对编程的初次接触源于我小镇上的一个邻居对我的启发，他(当时)通过向企业销售网站赚了一大笔钱。我想，如果他能靠这个赚钱，那么我也可以。我在 17 岁时和一个朋友创办了一家公司，致力于为企业建立网站。因为在家里就能为这些公司创造如此巨大的价值，我爱上了这个行业。

然而，编程从来不是我最喜欢的，也不是我想深入研究的东西。它只是达到目的的手段，而我必须这样做才能交付一个应用程序或网站。我起初在主机上编写 PL/I 语言，然后在浏览器应用程序中使用 JavaScript，同时做后端 API。我只是喜欢做互联网开发。这种热情引发了我对 Rust 的兴趣。这是我第一次碰到一种能够支持我的语言和编译器，让我能够专注于最重要的事情：为他人创造价值。

本书就是从这种实用主义的角度来写的，用行话讲就是：利用目前最好的工具创造价值。这本书展示了为什么使用 Rust，即使乍一看不明显，也是未来一代 Web 应用程序和 API 的完美选择。本书不仅关注语法，还提供指导和深入探讨，让你能够自信地开始和完成下一个 Rust 项目。

我想揭开 Rust 的面纱，看看 Rust crate、Rust 语言本身以及选择的 Web 框架的幕后有什么。详细程度始终以实用为目标：你需要了解多少内容才能有所作为，理解解决方案以便在自己的项目中进行调整，并知道如何进一步探索。

正如我的一位前同事所说：“写 Rust 代码就像走捷径一样！”我希望本书能让你认识到 Web 开发的美感，使用一种能够支持你并使你能够比以前更快、更安全地完成工作的语言。我很荣幸能带你踏上这个旅程！

致　谢

首先，我要感谢妻子艾米莉(Emily)，感谢她对我的信任和支持，鼓励我前进，并一直坚信我能完成这本书。写这本书占用了我本来就有限的时间，我将永远感激你的支持。谢谢你始终支持我和家庭。我爱你。

接下来，我要感谢迈克·斯蒂芬斯(Mike Stephens)，是他联系我，使这本书的出版成为可能。最初的通话真正激励了我，让我相信自己真的可以完成一本书。你的智慧和经验影响了这本书和我未来多年的写作。

感谢 Manning 出版社的编辑艾莉莎·海德(Elesha Hyde)：感谢你的耐心、意见、持续的邮件跟进，以及在整个过程中提供的宝贵建议和指导。我总是期待着与你会面，我会非常怀念这些时光。

感谢那些在这个旅程中给我启发的开发人员：马里亚诺(Mariano)，你的智慧和见解不仅帮助我完成这本书，也影响了我作为开发人员的职业生涯；克努特(Knut)和布莱克(Blake)，我们在 smartB 的时光和之后的讨论塑造了我对本书读者的态度；西蒙(Simon)，你教会了我如何成为一名开发人员并认真对待自己的工艺；还有保罗(Paul)，感谢你提供了一个发泄的渠道，让我通过交谈重新充满了能量，并对技术感到兴奋；达达(Dada)，和你一起学习是我能够写这本书的一个重要基石；最后但同样重要的是，塞巴斯蒂安(Sebastian)和费尔南多(Fernando)，和你们在一起的时光塑造了作为开发人员的我，同时影响了我的为人。

感谢所有的审阅者：Alain Couniot、Alan Lenton、Andrea Granata、Becker、Bhagvan Kommadi、Bill LeBorgne、Bruno Couriol、Bruno Sonnino、Carlos Cobo、Casey Burnett、Christoph Baker、Christopher Lindblom、Christopher Villanueva、Dane Balia、Daniel Tomás Lares、Darko Bozhinovski、Gábor László Hajba、Grant Lennon、Ian Lovell、JD McCormack、Jeff Smith、Joel Holmes、John D. Lewis、Jon Riddle、JT Marshall、Julien Castelain、Kanak Kshetri、Kent R. Spillner、Krzysztof Hrynczenko、Manzur Mukhitdinov、Marc Roulleau、Oliver Forral、Paul Whittemore、Philip Dexter、Rani Sharim、Raul Murciano、Renato Sinohara、Rodney Weis、Samuel Bosch、Sergiu Răducu Popa、Timothy Robert James Langford、Walt Stoneburner、William E. Wheeler 和 Xiangbo Mao。你们的建议使这本书变得更好。

关 于 本 书

本书将帮助你从头到尾编写 Web 应用程序(无论是 API、微服务还是单体应用)。你将学习到一切必要的知识，包括如何向外界开放 API，连接数据库以存储数据，以及测试和部署应用程序。

这不是一本参考书，而是一本工作手册。正在构建的应用程序在设计上做出了一些妥协，以便在适当的时候解释概念。需要阅读整本书的内容才能最终将应用程序部署到生产环境中。

哪些人应该阅读本书

本书适合那些已经阅读过 Steve Klabnik 和 Carol Nichols 合著的 *The Rust Programming Language*(No Starch Press，2019)前 6 章，并想知道“可以用它做什么”的读者。它也适合那些之前用其他语言构建过 Web 应用程序的开发人员，他们想知道 Rust 是否适用于他们的下一个项目。最后，对于那些需要使用 Rust 编写和维护 Web 应用程序的新手，这本书也是不错的选择。

本书的编排方式

本书分为三部分，共 11 章和一个附录。

第Ⅰ部分介绍使用 Rust 编写 Web 应用程序的原因和方法。

第 1 章介绍 Rust 适合哪种环境和团队，并解释为团队或下一个项目选择 Rust 的原因。该章将 Rust 与其他语言进行比较，并初步介绍其 Web 生态系统。

第 2 章讲述 Rust 语言基础知识以及完成本书和理解书中代码片段所需的知识，还介绍 Web 生态系统的基础知识，并描述在 Rust 中编写异步应用所需的额外工具。

第Ⅱ部分介绍如何创建应用的业务逻辑。

第 3 章为后续内容打下基础。该章介绍使用的 Web 框架 Warp，以及如何使用 JSON 响应 HTTP GET 请求。

第 4 章涵盖 HTTP POST、PUT 和 DELETE 请求，以及如何从内存中读取假数据。该章还介绍 urlform-encoded 和 JSON 主体之间的区别。

第 5 章讲解如何将代码模块化、执行代码检查和格式化。将大段代码拆分为自己的

模块和文件，使用 Rust 的注释系统对代码库进行注释，添加代码检查规则并进行格式化。

第 6 章介绍如何对运行中的应用程序进行反思。该章解释日志记录和跟踪之间的区别，并展示调试代码的各种方法。

第 7 章不再教你使用内存存储，而是添加一个 PostgreSQL 数据库。你将连接到本地主机上的数据库，创建连接池，并在路由函数之间共享该连接池。

第 8 章教你连接到外部服务，发送数据并处理接收到的响应。该章讨论如何打包异步函数和反序列化 JSON 响应。

第Ⅲ部分确保一切就绪，以便将代码投入生产环境。

第 9 章讨论有状态和无状态认证以及它们在代码中的体现。该章引入用户概念并教你添加令牌验证中间件。

第 10 章对输入变量进行参数化，例如 API 密钥和数据库 URL，并准备将代码库构建在各种架构和 Docker 环境中。

第 11 章以单元测试和集成测试结束本书，并介绍如何在每个测试之后启动和关闭模拟服务器。

附录针对审计和编写安全代码提供指导。

本书可以分章阅读。可以使用代码库来查看各章并为当前阅读的部分进行设置。应用程序是逐章构建的，因此如果你跳过某些章节，可能会错过一些信息。不过，章节可以用作一个软参考指南。

关于代码

本书中的代码示例基于 Rust 2021 edition 编写，并在 Linux 和 macOS 上进行了测试，支持 Intel 和 Apple 芯片。

本书包含许多源代码示例，既有带编号的代码清单，也有与普通文本放在一起的代码。在这两种情况下，源代码都以等宽体进行格式化，以与普通文本区分开来。此外，粗体用于突出显示与章节中先前步骤中的代码不同的代码，例如当新功能添加到现有代码行时。在某些情况下，删除线用于表示正在被替换的代码。

在许多情况下，原始源代码已经重新格式化；添加了换行符并重新调整了缩进，以适应书中可用的页面空间。此外，如果正文已对代码进行描述，源代码中的注释通常会从代码清单中删除。代码注释伴随着许多代码清单，以突出显示重要的概念。

扫描本书封底二维码，即可获取本书示例的完整代码。

关 于 作 者

Bastian Gruber 是 Centrifuge 的一名运行时工程师，全职使用 Rust 进行工作。他曾是 Rust 官方异步工作组的一员，并创办了 Rust and Tell Berlin Meetup 小组。他曾在全球最大的加密货币交易所之一从事 Rust 核心后端开发工作。他也是一位有着 12 年经验的作家，定期为 LogRocket 撰写关于 Rust 的文章，并接受采访和演讲的邀请。通过自己的经验，Bastian 拥有了以简单方式讲授复杂概念的能力，他的文章因易于理解和讲解深刻而受到喜爱。

关于封面插图

本书封面上的插图来自 Jacques Grasset de Saint-Sauveur 于 1788 年出版的作品集中的 *Femme de Stirie*(《来自斯蒂里亚的女人》)。作品集中的每幅插图都是精细绘制、手工上色的。

在那个时代，可通过人们的服装轻松辨识出他们居住的地方以及他们所在的行业和社会地位。

Manning 以几个世纪前丰富多彩的地区文化为基础，通过这样的图片使图书封面栩栩如生，从而颂扬计算机行业的创造性和活力。

目　录

第 I 部分　Rust 介绍

第 1 章　为什么使用 Rust ····· 3
1.1　开箱即用：Rust 提供的工具 ····· 4
1.2　Rust 编译器 ····· 8
1.3　Rust 用于 Web 服务 ····· 10
1.4　Rust 应用程序的可维护性 ····· 15
1.5　本章小结 ····· 15

第 2 章　建立基础 ····· 17
2.1　遵循 Rust 规范 ····· 18
2.1.1　使用结构体对资源进行建模 ····· 19
2.1.2　理解 Option ····· 21
2.1.3　使用文档解决错误 ····· 22
2.1.4　在 Rust 中处理字符串 ····· 27
2.1.5　深入理解移动、借用和所有权 ····· 28
2.1.6　使用和实现 trait ····· 31
2.1.7　处理结果 ····· 39
2.2　创建 Web 服务器 ····· 40
2.2.1　同时处理多个请求 ····· 41
2.2.2　Rust 的异步环境 ····· 42
2.2.3　Rust 处理 async/await ····· 43
2.2.4　使用 Rust Future 类型 ····· 44
2.2.5　选择运行时 ····· 45
2.2.6　选择 Web 框架 ····· 46
2.3　本章小结 ····· 49

第 II 部分　开始

第 3 章　创建第一个路由函数 ····· 53
3.1　认识 Web 框架：Warp ····· 54
3.1.1　Warp 包括哪些内容 ····· 54
3.1.2　Warp 的过滤器系统 ····· 55
3.2　获取第一个 JSON 响应 ····· 56
3.2.1　与你的框架理念保持一致 ····· 57
3.2.2　处理正确的路由 ····· 58
3.2.3　使用 Serde 库 ····· 59
3.2.4　优雅地处理错误 ····· 61
3.3　处理 CORS 头信息 ····· 65
3.3.1　在应用层面返回 CORS 头信息 ····· 66
3.3.2　测试 CORS 响应 ····· 67
3.4　本章小结 ····· 70

第 4 章　实现具象状态传输 API ····· 73
4.1　从内存中获取问题 ····· 74
4.1.1　设置一个模拟数据库 ····· 75
4.1.2　准备一组测试数据 ····· 78
4.1.3　从模拟数据库中读取 ····· 80
4.1.4　解析查询参数 ····· 84
4.1.5　返回自定义错误 ····· 88
4.2　创建、更新和删除问题 ····· 92
4.2.1　在线程安全的情况下更新数据 ····· 92
4.2.2　添加一个问题 ····· 96
4.2.3　更新问题 ····· 98
4.2.4　处理错误的请求 ····· 100
4.2.5　从存储中删除问题 ····· 101
4.3　通过 url 表单创建问题 ····· 103
4.3.1　url 表单和 JSON 的区别 ····· 104
4.3.2　通过 url 表单添加答案 ····· 104

4.4 本章小结……107

第 5 章 清理代码库……109
5.1 将代码模块化……109
5.1.1 使用 Rust 的内置模块系统……110
5.1.2 针对不同用例的文件夹结构……116
5.1.3 创建库和 sub-crate……120
5.2 为代码创建文件……124
5.2.1 使用文档注释和私有注释……124
5.2.2 在注释中添加代码……126
5.3 检测和格式化代码库……128
5.3.1 安装和使用 Clippy……128
5.3.2 使用 Rustfmt 格式化代码……131
5.4 本章小结……132

第 6 章 记录、追踪和调试……133
6.1 在 Rust 应用中记录日志……134
6.1.1 在 Web 服务中实现日志记录……136
6.1.2 记录 HTTP 请求日志……142
6.1.3 创建结构化的日志……145
6.2 异步应用中的追踪……152
6.2.1 引入 Tracing crate……153
6.2.2 集成 Tracing 到应用……154
6.3 调试 Rust 应用……158
6.3.1 在命令行上使用 GDB……159
6.3.2 使用 LLDB 调试 Web 服务……160
6.3.3 使用 Visual Studio 和 LLDB……162
6.4 本章小结……165

第 7 章 为应用添加数据库……167
7.1 设置示例数据库……168
7.2 创建第一个表……168
7.3 使用数据库 crate……171
7.3.1 将 SQLx 添加到项目中……173
7.3.2 将 Store 连接到数据库……174
7.4 重新实现路由函数……177
7.4.1 在 get_questions 中添加数据库……178
7.4.2 重新实现 add_question 路由函数……185
7.4.3 问题处理函数的更新和删除……187
7.4.4 更新 add_answer 路由……190
7.5 处理错误和追踪数据库交互……192
7.6 集成 SQL 迁移……198
7.7 案例研究：切换数据库管理系统……201
7.8 本章小结……204

第 8 章 集成第三方 API……205
8.1 准备代码库……207
8.1.1 选择一个 API……208
8.1.2 了解 HTTP 库……209
8.1.3 添加一个使用 Reqwest 的 HTTP 调用示例……211
8.1.4 处理外部 API 请求的错误……213
8.2 将 JSON 响应反序列化为结构体……219
8.2.1 收集 API 响应信息……220
8.2.2 为 API 响应创建类型……221
8.3 向 API 发送问题和答案……226
8.3.1 重构 add_question 路由函数……226
8.3.2 进行敏感词检查以更新问题……229
8.3.3 更新 add_answer 路由函数……230
8.4 处理超时和同时发生的多个请求……231
8.4.1 实现外部 HTTP 调用的重试机制……232
8.4.2 并发或并行执行 future……236
8.5 本章小结……238

第Ⅲ部分 投入生产

第 9 章 添加认证和授权……241
9.1 为 Web 服务添加认证……243
9.1.1 创建用户概念……243
9.1.2 迁移数据库……245

9.1.3 添加注册端点 ……247
9.1.4 对密码进行哈希处理……250
9.1.5 处理重复账户错误 ……252
9.1.6 有状态认证与无状态认证 ……258
9.1.7 添加登录端点 ……259
9.1.8 为令牌添加有效期 ……263
9.2 添加授权中间件 ……265
9.2.1 迁移数据库表 ……265
9.2.2 创建令牌验证中间件……266
9.2.3 扩展现有路由以处理账户 ID……270
9.3 未涵盖的内容 ……275
9.4 本章小结 ……276

第 10 章 部署应用程序 ……277
10.1 通过环境变量设置应用程序 ……277
10.1.1 设置配置文件 ……279
10.1.2 在程序中接收命令行参数……281
10.1.3 在 Web 服务中读取和解析环境变量 ……283
10.2 根据不同环境编译 Web 服务……288
10.2.1 构建二进制文件时的 development 和 release 标志…289
10.2.2 针对不同环境交叉编译二进制文件……290
10.3 在构建流程中使用 build.rs ……291
10.4 创建正确的 Web 服务 Docker 镜像 ……294
10.4.1 创建静态链接的 Docker 镜像 ……295
10.4.2 使用 docker-compose 建立本地 Docker 环境 ……296
10.4.3 将 Web 服务器的配置提取到一个新模块中……299
10.5 本章小结 ……303

第 11 章 测试 Rust 应用程序 ……305
11.1 业务逻辑的单元测试……306
11.1.1 测试分页逻辑和处理自定义错误 ……307
11.1.2 使用环境变量测试配置模块 ……310
11.1.3 使用新创建的模拟服务器测试 profanity 模块 ……314
11.2 测试 Warp 过滤器 ……321
11.3 创建集成测试配置……325
11.3.1 将代码库拆分为 lib.rs 和二进制文件 ……327
11.3.2 创建集成测试 crate 和单发服务器实现 ……330
11.3.3 添加注册测试……332
11.3.4 发生错误时进行堆栈展开 ……336
11.3.5 测试登录和发布问题……337
11.4 本章小结 ……339

附录 关于安全的思考 ……341

第 I 部分

Rust 介 绍

本书第 I 部分为你学习 Rust 语言奠定基础。在使用 Rust 进行 Web 应用开发之前，需要了解使用 Rust 语言和编写异步服务器应用程序所需的工具。第 I 部分将涵盖这两个主题。

第 1 章关注为什么要使用 Rust。该章展示 Rust 如何在比其他语言有更高性能的同时，让你能够轻松且安全地使用它创建应用程序；介绍如何在本地配置 Rust、工具链的形态，以及 Rust 中的异步和 Web 应用生态系统的形态(重要内容)。

第 2 章进一步讨论本书学习过程中所需的所有基础知识，让你不仅能理解书中的代码片段，而且能足够自信地开始新的 Rust 项目。

第1章

为什么使用 Rust

本章内容

- Rust 安装程序中包含的工具
- 初步了解 Rust 编译器及其独特之处
- 使用 Rust 编写 Web 服务所需的工具
- 支持 Rust 应用程序可维护性的特性

Rust 是一门系统编程语言。与 JavaScript 或 Ruby 这样的解释型语言不同，Rust 拥有编译器，类似于 Go、C 或 Swift。它结合了无运行时开销(如 Go 中的主动垃圾回收，Java 中的虚拟机)，同时提供了易于阅读的语法，类似于 Python 和 Ruby。因此，Rust 的性能和 C 语言相近。这一切都是因为 Rust 编译器可以在运行应用程序之前使所有类型的错误都得到纠正并确保消除许多经典的运行时错误，如释放后仍使用(use-after-free)。

Rust 提供了性能(无运行时和垃圾回收机制)、安全性(编译器确保内存安全，即使在异步环境中)和生产力(其内置的测试、文档和包管理工具使其构建和维护变得轻而易举)。

你可能听说过 Rust，但在跟着教程学习后，发现这门语言似乎过于复杂，以至于你放弃了对它的学习。然而，Rust 在 Stack Overflow 的年度调查中被评为最受欢迎的编程语言，并在 Facebook、Google、Apple 和 Microsoft 等公司中拥有大量粉丝。本书将帮助你解决学习 Rust 过程中的困难，并向你介绍如何熟悉 Rust 的基础知识，以及如何使用它构建和部署可靠的 Web 服务。

注意：

本书假设你已经写过一些小型的 Rust 应用程序，并且熟悉 Web 服务的一般概念。本书将介绍所有基本的 Rust 语言特性及用法，不过，这更像一种复习，而非深入的学习体验。例如，如果你已经读完了 Steve Klabnik 和 Carol Nichols 所写的 *The Rust Programming Language*(No Starch Press，2019)的前 6 章，那么你应该能够轻松地跟上本书中的练习。本书涵盖了 Rust 2021，并兼容 Rust 2018 版本。

Rust 为开发人员提供了一个独特的拓宽视野的机会。你可能是一名希望涉足后端开发的前端开发人员，或者是一名想要学习新语言的 Java 开发人员。Rust 的多功能性使你

可以通过学习它来扩展你能够使用的系统类型。你可以在任何可以使用 C++或 C 的地方使用 Rust，也可以在使用 Node.js、Java 或 Ruby 的情况下使用它。Rust 甚至开始在机器学习生态系统中找到立足点，而 Python 多年来一直在这个领域占据主导地位。此外，Rust 非常适合编译为 WebAssembly(https://webassembly.org)，而且许多现代区块链实现(如 Cosmos、Polkadot)都是用 Rust 编写的。

编写代码的时间越长，学习的编程语言越多，你就越能意识到，最重要的是掌握概念及最适合用来解决问题的编程语言。因此，本书不仅探讨如何使用 Rust 代码生成 HTTP 请求，还介绍 Web 服务的一般工作原理，以及异步 Rust 的底层概念，以便你选择最合适的传输控制协议(TCP)抽象。

1.1 开箱即用：Rust 提供的工具

Rust 提供了适量的工具，使应用程序的启动、维护和构建变得简单。图 1.1 列出了开始编写 Rust 应用程序时所需的最重要工具。

工具链/版本管理	Rust编译器	代码格式化程序	代码检查	包管理器	包仓库
Rustup	Rustc	Rustfmt	Clippy	Cargo	crates.io

图 1.1 编写和发布 Rust 应用程序时需要使用的全部工具

可通过在终端上执行代码清单 1.1 中所示的命令来下载 Rustup 并安装 Rust。这适用于 macOS(通过 brew install rustup-init)和 Linux。关于在 Windows 上安装 Rust 的最新方法，可参考 Rust 网站(www.rust-lang.org/tools/install)上的说明。

代码清单 1.1 安装 Rust

```
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

命令行工具 curl 用于通过 URL 传输数据。你可以获取远程文件并将其下载到计算机上。选项--proto 表示可以使用的协议，如安全超文本传输协议(HTTPS)。通过参数--tlsv1.2，使用 1.2 版本的传输层安全协议(http://mng.bz/o5QM)。接下来是 URL，如果你通过浏览器打开它，它会提供一个用于下载的 shell 脚本。此 shell 脚本通过管道(|)传输给 sh 命令行工具，然后执行该脚本。

shell 脚本还将安装 Rustup 工具，让你可以更新 Rust 并安装辅助组件。若要更新 Rust，只需要运行 rustup update：

```
$ rustup update
info: syncing channel updates for 'stable-aarch64-apple-darwin'
info: syncing channel updates for 'beta-aarch64-apple-darwin'
```

```
info: latest update on 2022-04-26,
      rust version 1.61.0-beta.4 (69a6d12e9 2022-04-25)

…

  stable-aarch64-apple-darwin unchanged - rustc 1.60.0
(7737e0b5c 2022-04-04)
     beta-aarch64-apple-darwin updated -
     rustc 1.61.0-beta.4 (69a6d12e9 2022-04-25)
     (from rustc 1.61.0-beta.3 (2431a974c 2022-04-17))
nightly-aarch64-apple-darwin updated -
  rustc 1.62.0-nightly (e85edd9a8 2022-04-28)
(from rustc 1.62.0-nightly (311e2683e 2022-04-18))

info: cleaning up downloads & tmp directories
```

如果想安装更多组件，比如图 1.1 中提到的代码格式化工具，也可使用 Rustup。若要安装代码格式化工具(Rustfmt)，可以运行代码清单 1.2 中的命令。

代码清单 1.2　安装 Rustfmt

```
$ rustup component add rustfmt
```

通过运行 cargo fmt，格式化工具会根据风格指南检查并格式化代码。你需要指定要格式化的文件夹或文件。例如，你可以导航到项目的根目录，然后运行 cargo fmt.(带有一个点)，对所有的目录和文件进行格式化。

执行代码清单 1.1 中的 curl 命令后，你不仅安装了 Rust 库，还安装了包管理器 Cargo。这将使你能够创建和运行 Rust 项目。鉴于这点，让我们创建并执行第一个 Rust 程序。代码清单 1.3 展示了如何运行一个 Rust 应用程序。命令 cargo run 将执行 rustc，编译代码，并运行生成的二进制文件。

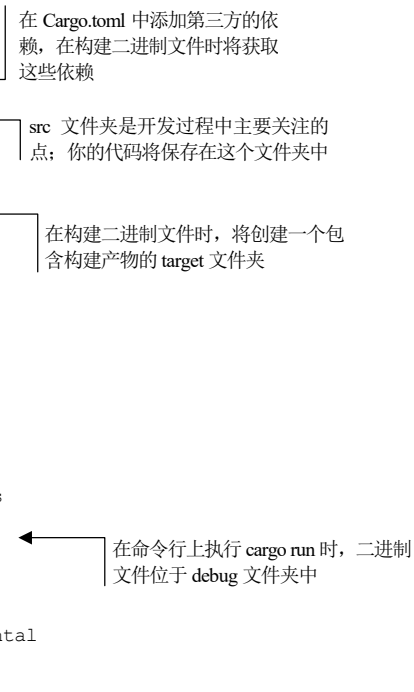
代码清单 1.3　运行第一个 Rust 程序

```
$ cargo new hello
$ cd hello
$ cargo run

   Compiling hello v0.1.0 (/private/tmp/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 1.54s
     Running `target/debug/hello`
Hello, world!
```

新程序在控制台打印出“Hello, world!”，稍后将介绍它的原理。可以在项目文件夹 hello 中看到代码清单 1.4 中列出的文件和文件夹。cargo new 命令用指定的名称创建了一个新的文件夹，并为其初始化了一个新的 Git 结构。

代码清单 1.4 Rust 新项目的文件夹内容

```
  tree .
.
├── Cargo.lock
├── Cargo.toml
├── src
│   └── main.rs
└── target
    ├── CACHEDIR.TAG
    └── debug
        ├── build
        ├── deps
        │   ├── …
        ├── examples
        ├── hello
        ├── hello.d
        └── incremental
            └── …

9 directories, 28 files
```



cargo run 命令将构建应用程序并执行位于./target/debug 文件夹内的二进制文件。源代码位于 src 文件夹中。根据构建的应用程序的类型，src 文件夹中包含一个带有代码清单 1.5 所示内容的 main.rs 或 lib.rs 文件。

代码清单 1.5 自动生成的 main.rs 文件

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

第 5 章将介绍 lib.rs 和 main.rs 文件的区别，以及 Cargo 何时创建这两个文件。target 文件夹中包含 debug 文件夹，debug 文件夹中包含由 cargo run 命令生成的编译后的代码。简单的 cargo build 命令也会产生相同的效果，但只会构建程序，不会执行程序。

在构建 Rust 程序时，Rust 编译器(Rustc)会创建 Rust 字节码，并将其传递给另一个名为 LLVM(https://llvm.org)的编译器，以创建机器代码(LLVM 也被 Swift 和 Scala 之类的语言所使用，并将语言编译器产生的字节码转换为操作系统运行的机器代码)。这意味着 Rust 可以在 LLVM 支持的任何操作系统上进行编译。整个技术栈如图 1.2 所示。

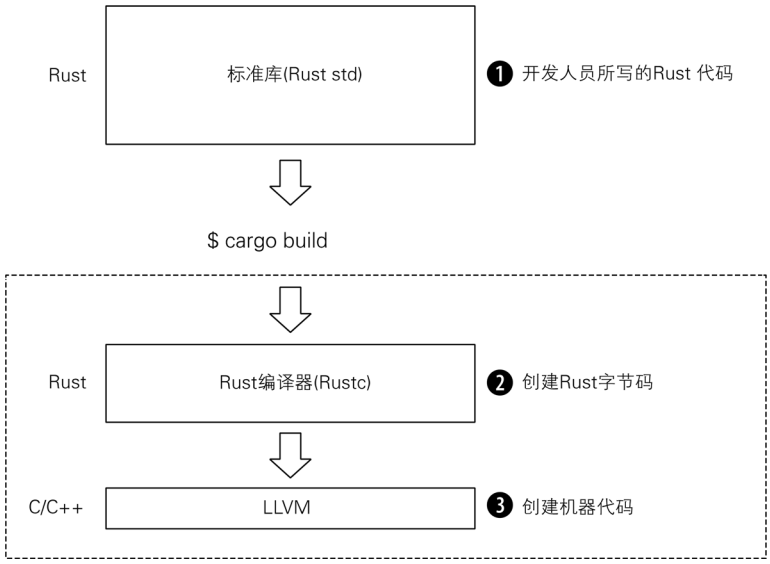


图 1.2　在安装了 Rustup 后，你的设备将会包含 Rust 标准库，其中包括 Rust 编译器

另一个重要的文件是 Cargo.toml，如代码清单 1.6 所示，它包含了项目的整体信息并在必要时指定第三方依赖项。

注意：

在开发库时，不应将 Cargo.lock 文件提交到版本控制系统(如 Git)中。但是在创建应用程序(二进制文件)时，应该将该文件添加到版本控制系统中。应用程序(二进制文件)通常依赖于特定版本的外部库，因此与你合作的其他开发人员需要知道哪些版本是可以被安全安装或需要更新的。另一方面，对于库，应该保证其在所使用库的最新版本上可用。

代码清单 1.6　Cargo.toml 文件内容

```
[package]
name = "check"
version = "0.1.0"
edition = "2021"

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

通过在[dependencies]部分下添加依赖项名称并运行 cargo run 或 cargo build 来安装第三方库。这将从 crates.io(Rust 包注册表)获取库(在 Rust 社区中称为 crates)。已安装包的实际获取版本将显示在 Cargo.lock 的新文件中。如果该文件位于项目的根目录中，Cargo 将获取 Cargo.lock 文件中指定的确切的包版本。这将有助于在不同机器上使用相同代码库的开发人员复制完全相同的状态。

TOML 文件

TOML 文件格式与 JavaScript 对象表示法(JavaScript Object Notation，JSON)或 YAML Ain't Markup Language(YAML)一样，是一种配置文件格式。它表示 Tom's Obvious Minimal Language，顾名思义，旨在使配置易于阅读和解析。包管理器 Cargo 使用此文件来安装依赖项并填充有关项目的信息。

正如 Rust 核心成员之一所说: “这是不太糟糕的选择。”(http://mng.bz/aP9J)这并不意味着 TOML 很糟糕，而是说在处理配置文件时总是存在一些取舍。

工具箱中的最后一个工具是官方代码检查器 Clippy。现在，在安装 Rust 时，默认会安装此工具。如果使用的是较旧的 Rust 版本，也可以手动安装它，如代码清单 1.7 所示。

代码清单 1.7 安装 Clippy

```
$ rustup component add clippy
```

第 5 章将详细介绍如何使用 Clippy 以及如何配置它。

1.2 Rust 编译器

Rust 相比于其他语言的优势在于其编译器。Rust 编译为二进制代码，运行时不进行垃圾回收。这使得它具有类似 C 语言的速度。然而，与 C 语言不同的是，Rust 编译器在编译时强制保证内存安全。图 1.3 展示了用于服务器端编程的流行编程语言与 C 语言之间的差异。

每种语言都有取舍。Go 是图 1.3 所示的语言中最新的一种，它的速度最接近于 C 语言。它使用运行时进行垃圾回收，因此比 Rust 需要更多的开销。Go 的编译器比 Rustc 快。Go 的目标是简化代码，并为此牺牲了一些运行时性能。

Rust 没有运行时开销，并且由于编译器的原因，你在编写代码时会发现 Rust 比 Go 或 JavaScript 提供了更高的舒适度和安全性。例如，Java 和 JavaScript 需要某种虚拟机来运行代码，这会导致严重的性能损失。

在用 Rust 编写程序时，你需要改用 Rust 编译器构建应用程序。如果你是从脚本语言转过来的，这将是一种巨大的心态转变。相比于在几秒钟内启动一个应用程序并对其进行调试，直到它失败，Rust 编译器会在启动之前确保一切正常。例如，请考虑代码清单 1.8 中的代码片段(摘自本书后续内容，用于举例)。

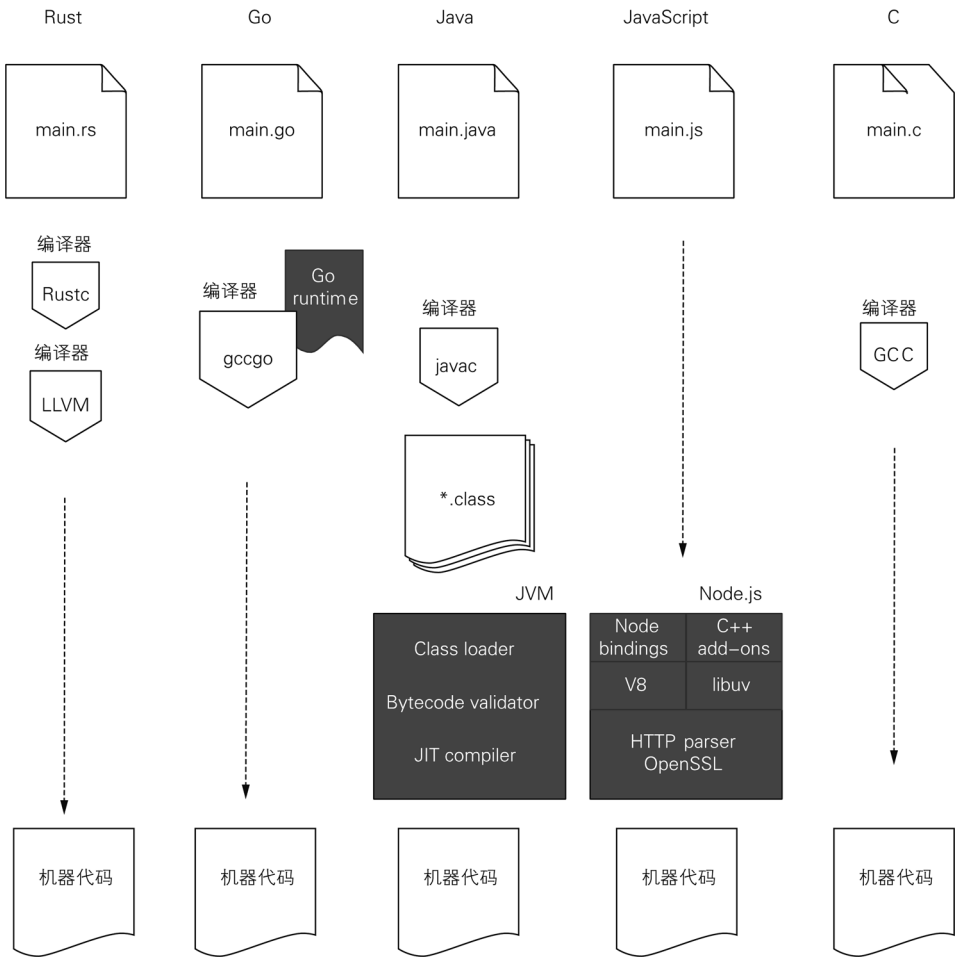


图 1.3　对比 Rust 和其他语言(将源代码编译为机器代码)

代码清单 1.8　校验空 ID

```
match id.is_empty() {
    false => Ok(QuestionId(id.to_string())),
    true => Err(Error::new(ErrorKind::InvalidInput, "No id provided")),
}
```

如果你还不能阅读代码清单 1.8 中的代码片段，不用担心，很快就会了。这是一个 match 块，编译器确保覆盖了每一种用例(无论 id 是否为空)。如果删除 true =>的那一行，并尝试编译代码，将得到代码清单 1.9 所示的错误。

代码清单 1.9　缺少模式匹配的编译器错误

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

```
  --> src/main.rs:31:15
   |
31 |         match id.is_empty() {
   |               ^^^^^^^^^^^^^ pattern `true` not covered
   |
   = help: ensure that all possible cases are being handled,
      possibly by adding wildcards or more match arms
   = note: the matched value is of type `bool`
```

编译器会突出显示错误的行及其在语句中的确切位置，并提供一个解决当前问题的建议。编译器旨在生成易于理解和阅读的错误信息，而不是仅仅揭露内部解析器的错误。

对于小型应用程序，事先确保程序在所有用例中都能正确运行的做法可能看起来很烦琐，但是一旦你需要维护更大的系统并添加或删除功能，你很快就会发现 Rust 有时会让你觉得自己在走捷径，因为过去你必须考虑的许多问题现在都会被编译器解决。

因此，新编写的 Rust 代码多数不能立即运行。编译器将成为你日常工作的一部分，帮助你了解在哪里改进代码和你可能忘记考虑的内容。

你不能像掌握 JavaScript 或 Go 那样快速上手 Rust。你需要先熟悉一组基本概念。此外，你还必须学习 Rust 的许多方面，才能成为一名熟练的 Rust 开发人员。即便如此，你也不需要在了解了所有内容之后才开始；你可以在编译器的帮助下边学边做。可见，Rust 编译器是你选择使用 Rust 的最有力的理由之一。

一旦熟练掌握了 Rust，就可以将其用于多个领域，如游戏开发、后端服务器、机器学习，不久之后甚至能将其用于 Linux 内核开发(目前正在讨论和试验阶段：https://github.com/Rust-for-Linux)。

如果在一个较大的团队中开发应用程序，需要认识到，刚开始接触 Rust 的程序员必须先熟练使用编译器，然后才能为代码库做出贡献。这将覆盖大量的代码审查，并保证代码质量的基准。

1.3 Rust 用于 Web 服务

前面的小节已经介绍了开发人员选择 Rust(而不是其他编程语言)的主要原因。接下来将介绍如何用 Rust 来编写 Web 服务。令人意外的是，当涉及 HTTP 时，Rust 并没有像 Go 或 Node.js 那样覆盖广泛的范围。由于 Rust 是一种系统编程语言，Rust 社区决定将实现 HTTP 和其他功能的工作留给社区。

图 1.4 展示了一个典型的 Web 服务技术栈，以及 Rust 在多大程度上提供支持。底部的两层(TCP/IP)由 Rust 栈覆盖。Rust 标准库实现了 TCP，可以打开一个 TCP(或用户数据报协议，UDP)套接字并监听传入的消息。

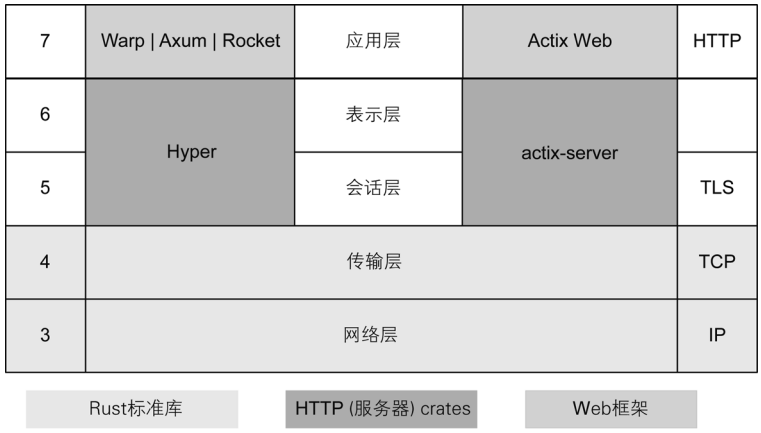


图 1.4　Rust 标准库和第三方库的覆盖范围(基于 OSI 模型)

然而，由于没有 HTTP 实现，因此，如果想编写一个纯 HTTP 服务器，你必须从头开始实现它，或者使用第三方库(如 Hyper，作为底层被 curl 使用)。

若使用的是 Web 框架，那么 HTTP 的实现是已经确定的。例如，Web 框架 Actix Web 使用它自己的 HTTP 服务器(actix-server)实现。当你使用 Warp、Axum 或 Rocket 时，它们都使用 Hyper 作为 Web 服务器(打开套接字，等待并解析 HTTP 消息)。

如图 1.4 所示，TCP 包含在 Rust 标准库中，但是其上的一切都是由社区支持的。这在代码中是什么样子的呢？下面以 Go 为例。代码清单 1.10 展示了用 Go 编写的 HTTP 服务器。

代码清单 1.10　使用 Go 编写的 HTTP 服务器的简单示例

```
package main

import (
     "fmt"
     "net/http"
)

func hello(w http.ResponseWriter, req *http.Request) {
   fmt.Fprintf(w, "hello\n")
}

func main() {

     http.HandleFunc("/hello", hello)

     http.ListenAndServe(":8090", nil)
}
```

可以看到，Go 提供了一个 HTTP 包。Rust 则缺少 HTTP 部分，它仅实现了 TCP 部分。可以用 Rust 创建一个 TCP 服务器，参考代码清单 1.11，但不能立即使用它来响应常规的 HTTP 消息。

代码清单 1.11 使用 Rust 编写的 TCP 服务器示例

```
use std::net::{TcpListener, TcpStream};

fn handle_client(stream: TcpStream) {
     // 做点事情
}

fn main() -> std::io::Result<()> {
   let listener = TcpListener::bind("127.0.0.1:80")?;

   for stream in listener.incoming() {
      handle_client(stream?);
   }
   Ok(())
}
```

因此，HTTP 的实现取决于社区。幸运的是，已经有了很多实现。当你在后面的章节中选择 Web 框架时，不需要担心 HTTP 实现的部分。

编写 Web 服务的另一大基石是异步编程。这使你有能力同时处理多个请求，而且它减少了等待服务器响应的时间。

当一个 Web 服务器收到一个请求时，需要完成一些任务(访问数据库，写入文件，等等)。如果 Web 服务器在第一个请求处理完成之前收到第二个请求，那么第二个请求将不得不等待第一个请求处理完成。想象一下几乎同时收到数百万个请求的情况。

因此，需要一种方法将任务放在后台，并继续在服务器上接收请求。这就是异步编程的意义所在。其他框架和语言(如 Node.js 和 Go)在一定程度上会自动在后台处理这个问题。而在使用 Rust 时，需要更细致地了解异步编程的组件，这样才能正确选择框架。

为了创建可以异步处理工作的应用程序，编程语言(或其周围的生态系统)需要提供以下概念：

- 语法——标记一段代码为异步代码。
- 类型——一种更复杂的类型，可以保存异步任务的状态。
- 线程调度器(运行时)——处理线程或其他将工作放在后台并进行处理的方法。
- 内核抽象——在后台使用异步的内核方法。

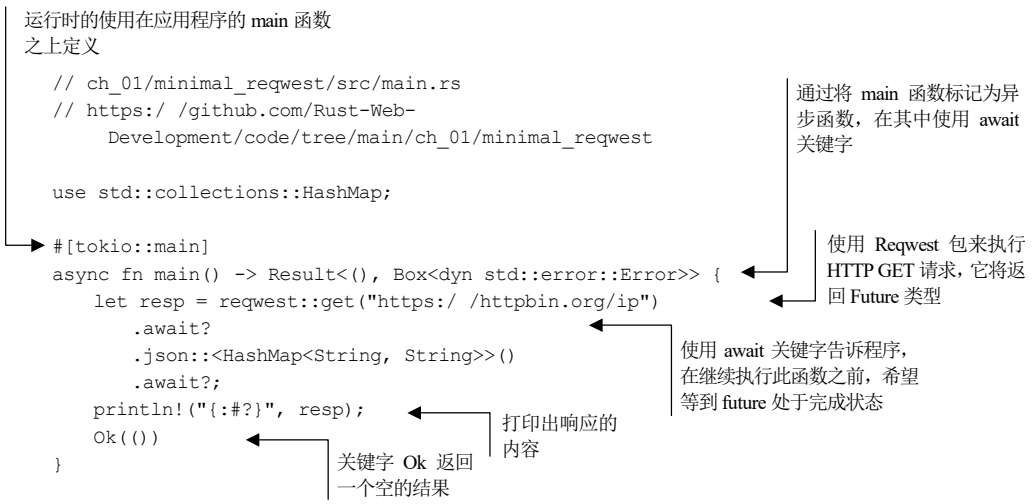
Rust 中的异步运行时

这里谈及的运行时，与 Java 运行时或 Go 的垃圾回收不一样。在编译过程中，运行时将被编译成静态代码。每个支持某种形式异步代码的库或框架都会选择一种运行时来构建。运行时的工作是选择自己的方式来处理线程和管理后台的工作(任务)。

因此，最终有可能出现多个运行时。例如，如果选择了一个基于 Tokio 运行时的 Web 框架，并且有一个建立在另一个运行时之上的工具库来执行异步的 HTTP 请求，那么在二进制代码中至少会编译两个运行时。至于是否会产生副作用，取决于应用程序的设计。

在 Rust 中，异步获取网站的示例代码如代码清单 1.12 所示。关于代码的细节，此处不会详细介绍，将留给第 2 章及以后的章节，但你应该能初步了解 Rust 中的异步代码。为了让代码清单 1.12 中的代码片段能正常工作，需要将外部 crate Reqwest(这不是拼写错误，这里并非指单词 request)添加到项目中(通过 Cargo.toml 文件)。

代码清单 1.12　在 Rust 中异步地发起 HTTP GET 请求



使用#[tokio::main]注解来标记 main 函数。这里使用的异步运行时(或线程调度器)是 Tokio。Tokio 本身通过另一个名为 Mio 的 crate 使用了之前提到的操作系统的异步内核应用程序编程接口(API)。

可以使用标准的Rust语法将函数标记为异步(async)函数，并在future(http://mng.bz/5mv1)类型上使用 await。在 Rust 中，future 是一个可以实现到类型上的 trait。这个 trait 规定实现必须有一个类型 Output(表示当计算完成时 future 返回的内容)，以及一个名为 poll 的函数(运行时可以调用它来处理 future)。当使用 Web 框架时，你可能永远不会接触到 future 的实际实现，但仍须了解底层概念，这样编译器消息和框架实现对你来说才有意义。

Rust 与其他语言不同，在 Rust 中，future 的工作只有在交给运行时并主动启动时才开始。异步函数返回一个 Future 类型，函数的调用者负责将这个 future 传递给运行时，以便对其进行处理。

对于开发人员来说，这意味着在调用函数时添加.await，表示运行时将执行它。此外，还有其他启动 future 的方法(如 Tokio 的 join!宏：http://mng.bz/694D)，你将在第 8 章中使用这些方法。

对运行时的选择基本上由后续选择的 Web 框架负责。Web 框架将决定它所依赖的运行时。

图 1.5 展示了之前列出的组件，以及代码清单 1.12 中使用的组件。第 2 章将更深入地讨论相关内容。Rust 在其标准库中提供了语法和类型，并将运行时和内核抽象留给社区来实现。

图 1.5 Rust 为异步编程提供了语法和类型，但运行时和内核抽象在核心语言之外实现

代码清单 1.13 展示了一个使用 Web 框架 Warp(本书后续的选择)的最小可用 Web 应用。Web 框架 Warp 建立在 Tokio 运行时之上，这意味着也必须将 Tokio 添加到项目中(通过 Cargo.toml 文件，如代码清单 1.14 所示)。

代码清单 1.13 使用 Warp 的最小可用 Rust HTTP 服务器

```
// ch_01/minimal_warp/src/main.rs
// https:/ /github.com/Rust-Web-Development/code/tree/main/ch_01/minimal-warp

use warp::Filter;

#[tokio::main]
async fn main() {
    let hello = warp::get()
         .map(|| format!("Hello, World!"));

    warp::serve(hello)
        .run(([127, 0, 0, 1], 1337))
        .await;
}
```

代码清单 1.14 最简单的 Warp 示例的 Cargo.toml 文件

```
[package]
name = "minimal-warp"
version = "0.1.0"
edition = "2021"
[dependencies]
tokio = { version = "1.2", features = ["full"] }
warp = "0.3"
```

以上语法可能看起来很陌生，但你可以看到 Tokio 运行时的内容都被抽象到 Warp 框架中，Tokio 运行时的唯一标志是 main 函数上方的那一行。第 2 章将更详细地介绍 Tokio 框架，以及为什么选择它。

你可能会疑惑，既然 Rust 没有标准的运行时来处理异步代码，Rust 标准库中也不包含 HTTP，为什么用它来写 Web 服务呢？这归根结底是因为语言特性和社区。

你可以认为，正因为没有标准的 HTTP 实现和运行时，所以 Rust 更加面向未来，因为社区可以随时介入并改进某些方面，或为不同的问题提供不同的解决方案。在需要处

理应用程序中的异步工作和大流量的环境中，语言的类型安全性、速度和正确性发挥了至关重要的作用。从长期来看，快速且安全的语言将使你获益。

1.4　Rust 应用程序的可维护性

Rust 编译器帮助你编写健壮的软件，而 Rust 的其他语言特性也使 Rust 更易于维护。例如，Rust 内置了文档功能。第 5 章将更深入地介绍如何使用内置工具正确地为代码编写文档。包管理器 Cargo 提供了一个命令，利用此命令，可以通过代码注释生成文档。这些文档可以在本地浏览，并在将库导出到 crates.io 时默认构建。嵌入代码文档中的代码不仅会出现在预生成的 HTML 文档中，而且会通过测试，因此可以确保永远不会有过时的代码示例。

除了文档化，模块化代码库还有助于将部分代码组合在一起，或将可重用的代码提取到它自己的 crate 中。Rust 通过在 Cargo.toml 文件中使用依赖项(dependency)来将本地库(从官方 crates.io 注册表或你希望的任何其他位置)纳入其中，从而使模块化变得非常简单。

Rust 还默认支持测试。不需要额外的 crate 或其他辅助工具来创建和运行测试。所有这些内置的和标准化的功能消除了团队中的许多分歧，你可以专注于编写和实现代码，而不是总忙着寻找新工具来编写文档或测试。

如果以后需要帮助，但团队中没有专业人士，可以在诸多 Discord 频道、Reddit 论坛和 Stack Overflow 标签中寻求指导。例如，关于运行时 Tokio 和 Web 框架 Warp 的帮助，可以在 Tokio Discord 服务器上找到，该服务器为每个工具提供了一个频道。这是一个很好的方法，可以用来寻求帮助或阅读其他人的评论，以了解更多关于所用工具的信息。

1.5　本章小结

- Rust 是一种系统编程语言，可以生成二进制文件。
- Rust 配备了一个严格的编译器，提供有用的错误信息，因此你可以轻松发现错误并改进。
- 与 Rust 相关的工具随安装包一起提供，或者有官方推荐，不需要你不断地探索、讨论和学习新工具，从而为你节省时间。
- 编写异步代码(第 2 章将介绍具体方法)时，需要选择一个运行时，因为 Rust 不像 Go 或 Node.js 那样包含一个运行时。
- Web 框架是基于运行时构建的，因此运行时的选择将由后续选择的框架决定。
- Rust 的速度、安全性和正确性在维护大大小小的 Web 服务和代码库时将带来巨大帮助。
- 文档和测试内置于语言本身，这使代码的维护变得更加容易。

第2章

建立基础

本章内容

- 介绍 Rust 类型
- 理解 Rust 的所有权系统
- 在自定义类型中实现自定义行为
- 理解异步生态系统中的组件
- 选择用于构建 Web 服务的第三方库
- 使用 Rust 构建基本可用的 Web 服务

第 1 章介绍了 Rust 自带的特性和创建 Web 服务所需的工具。本章将详细讨论这些要点。本章分为两部分：第一部分详细介绍如何使用 Rust 语言创建自定义类型和函数；第二部分将构建一个 Web 服务器，并为用户提供响应。

正如前面提到的，阅读 *The Rust Programmig Language*(https://doc.rust-lang.org/book/)的第 1~6 章将对你有很大帮助。本章将介绍阅读本书所需的概念，所以即使你没有任何基础的知识，仅阅读本章可能也足够了。然而，再次建议你至少简要地浏览一下 *The Rust Programmig Language* 的前 6 章，这样你才能对语言本身有一定的基础。

在本书中，你将创建一个示例问答 Web 服务，用户可以在其中提问和回答问题。你将构建一个具象状态传输(REST)API，并在本书结束时部署和测试一个运行中的服务。你还将对问题进行存储、更新和删除，并发布答案。在本书后续部分，你将弄清楚如何对这个 Web 服务进行认证，以及如何进行适当的测试。

本书将重点关注 Web 服务中和 Rust 相关的内容。无论最终你在自己的项目中选择了哪个 Web 框架，本书的内容都应该在某种程度上对你有所帮助。本书的目标是传授和展示一种实现方式。

需要注意，Rust 具有两面性。一方面，你不需要了解操作系统的底层细节；另一方面，应适当了解操作系统如何分配内存和执行函数，这对你来说是有益的。这也是本书的目的。你不仅要学习另一种语法，还要增强对操作系统和 Web 服务的整体理解。

本章的目的在于建立基础。图 2.1 展示了后续各小节的内容。了解了 Rust 中会经常接触的领域之后，便可以花时间有针对性地深入学习。对于 Web 服务，也是如此。如果

曾经遇到性能问题或对框架的选择不满意，你便会知道如何在 Rust 生态系统中选择更符合需求的 crate。

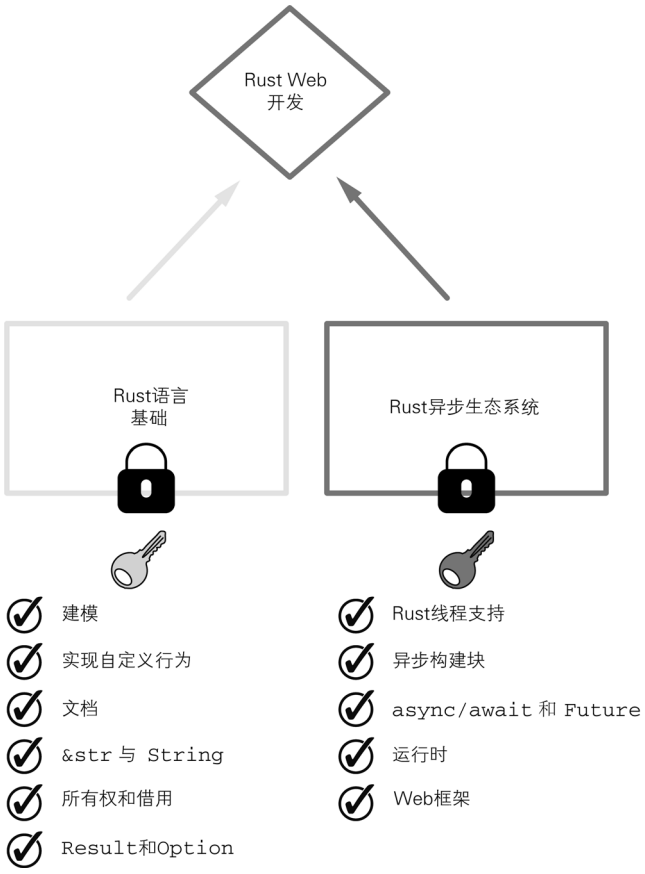


图 2.1 本章路线图(解锁成为 Rust Web 开发人员所需的能力)

为了使阅读本书成为一件有价值的事，你应该现在学习这些基础知识，这样，在许多年以后，你仍然能从中受益。这是本书中最后一个更注重基本原理(而非代码)的章节，从下一章开始将加快节奏。

你将从结构体(struct)开始实现 Web 服务，并在本章末尾运行一个基本的 Web 服务器，但本章的目标是在过程中解释相关概念。接下来的章节将假设你对 Rust 语言和生态系统已经有了基本了解。

2.1 遵循 Rust 规范

Rust 是一门复杂的语言，但在开始时或在进行较大的项目时，你不必了解所有细节。编译器和其他工具(如 Clippy，将在第 5 章中深入研究)将在很大程度上帮助你以整洁的方

式完成代码。因此，本书虽然不会涵盖语言的每个方面，但当你遇到问题时，你可以信心满满地搜索到相关话题。

为了自信地使用 Rust 工作，需要掌握以下技能：

- 通过官方 Rust 文档 docs.rs 查询类型和行为。
- 快速迭代错误或问题。
- 理解 Rust 所有权系统原理。
- 识别和使用宏。
- 通过结构体(struct)创建自定义的类型，并通过 impl 实现行为。
- 在现有类型上实现 trait 和宏(macro)。
- 使用 Result 和 Option 编写函数式 Rust。

本章将介绍以上基础知识，在后续章节中，你将进行练习和更深入的探索。需要了解的一点是，即使在探索过程中遇到复杂的问题或挑战，也可通过之前学习的技能来解决，你只需要积累经验和正确的思维方式来克服它们。

由于 Rust 是一种静态类型语言，你需要在程序开始时投入更多精力。如果不熟悉现有类型和如何处理未知值，那么你可能需要更长的时间，才能快速从另一个端点获取 JSON 文件或建立一个简单的程序。

2.1.1　使用结构体对资源进行建模

创建一个具象状态传输 API，意味着将提供创建、读取、更新和删除(CRUD)资源的路由。因此，第一步是思考在 Web 服务中需要处理哪些模型或类型。

一个明智的选择是从最小可用应用程序开始规划。这包括想要实现的自定义数据类型及其行为(方法)。对于当前的应用程序，需要考虑以下内容：

- 用户(User)
- 问题(Question)
- 答案(Answer)

用户可以注册并登录系统，然后发布和查看问题以及这些问题的答案。本书后面的章节将在讨论应用程序的认证和授权时重点关注用户。下面将实现不需要检查密码和用户 ID 的路由。

在 Rust 中创建和实现自己的类型时所需的东西如图 2.2 所示。你将从实现 Question 类型开始，并在探索过程中了解所遇到的所有问题和需要的类型；可通过使用 struct 关键字并向其添加字段来创建自己的类型，然后使用 impl 块以函数形式添加行为。

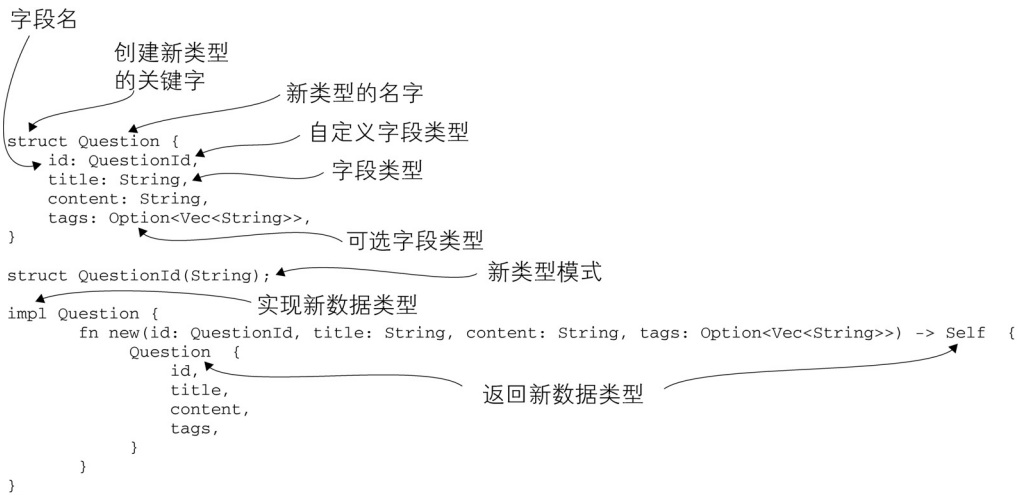


图 2.2 自定义类型可以用 struct 来创建，通过 impl 代码块来添加自定义方法

下面从创建问题(Question)和答案(Answer)开始，如代码清单 2.1 所示，回顾一下在 Rust 中创建自定义类型的基本过程。

代码清单 2.1 创建并实现 Question 类型

```
struct Question {
    id: QuestionId,
    title: String,
    content: String,
    tags: Option<Vec<String>>,
}
struct QuestionId(String);

impl Question {
   fn new(
       id: QuestionId,
       title: String,
       content: String,
       tags: Option<Vec<String>>
   ) -> Self {
       Question {
            id,
            title,
            content,
            tags,
        }
    }
}
```

在创建 Question 类型时，使用 ID 来区分不同的 Question(当前手动创建 ID，在本书后面的部分中，将使用自动生成的 ID)。每个 Question 都有一个标题，实际内容在 content 中。我们还使用 tags 将某些问题组合在一起。2.1.2 节将解释 Option 的含义。Rust 中的函

数与其他编程语言中的函数非常相似。图 2.3 展示了 Rust 中的函数签名，以及每个元素的含义。

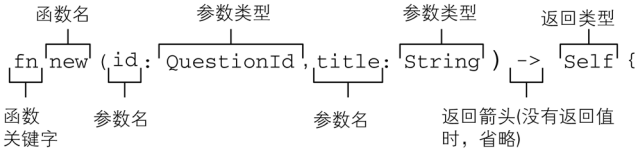


图 2.3　Rust 返回值时的函数签名细节

在 Rust 中，你需要经过以下步骤来创建自定义类型数据：

(1) 通过 struct Question{...}创建新的结构体。

(2) 将字段及其类型添加到结构体中。

(3) Rust 没有默认的构造函数名称，最佳实践是使用 new 作为方法名。

(4) 使用 impl 代码块将行为添加到自定义类型中。

(5) 返回 Self 或 Question 以实例化该类型的新对象。

我们还使用了 New Type 模式(http://mng.bz/o5Zr)，在这种模式下，并非简单地为 Question ID 使用字符串，而是将字符串封装到一个名为 QuestionId 的结构体中。每次传递参数或尝试创建一个新 Question 时，需要创建一个新的 QuestionId，而不是简单地传递一个字符串。通过自定义类型，可以传达特定的目的——编译器会确保使用的是正确类型。

就目前而言，这似乎是不必要的，但在更大的应用程序中，这会使参数更有意义。你可以将自定义类型视为 ID。应用程序中可以存在同时处理 Question ID(如上面提到的)和 Answer ID 的函数。将原始类型封装在结构体中以赋予它们意义，并使其在实例化时保持灵活性。

2.1.2　理解 Option

Option 在 Rust 中很重要。Option 能够确保在应该有值的地方不会出现空值(null)。通过 Option 枚举，可以随时检查所提供的值是否存在，并处理不存在的情况。而且，当使用 Option 枚举时，编译器会确保始终覆盖每一种情况(Some 或 None)。这还允许声明非必需的字段，因此在创建新的 Question 时，标签列表并不是必需的，你可以根据自己的需求和偏好决定是否提供。

另外，在与外部 API 交互并且需要接收数据时，也可将某些字段标记为可选。因为 Rust 是严格类型化的，如果类型上存在没有被设置的字段，编译器就会抛出一个错误。此外，在默认情况下，所有的结构体字段在定义时都是必需的。因此，必须确保那些非必需的字段都被标记为 Option<Type>。

Rust Playground

Rust 学习过程中的一个重要部分是使用一些可用的工具来快速验证想法。Rust Playground 网站(https://play.rust-lang.org/)提供了 Rust 编译器和最常用的包，以便快速迭代

较小的程序。因此，你不必每次都创建本地 Rust 项目来尝试某个主题。

检查 Option 是否具有值的常见方法是使用 match 关键字。对于代码清单 2.2，你可以复制、粘贴其中的代码并在 Rust Playground 中运行或通过单击 http://mng.bz/neZg 来运行，该代码清单展示了如何在可选值上使用 match 块。这个示例是凭空创建的，旨在演示 match 块的用例。

Rust 中的模式匹配

初学者往往将 match 视为 switch 的替代关键字。然而，Rust 中的模式匹配功能要强大得多。*The Rust Programmig Language* 的第 18 章详细介绍了这一点(http://mng.bz/AVYp)。

例如，match 模式还允许解构结构体(http://mng.bz/49na)、枚举(http://mng.bz/Qnww)等。这种强大的机制使代码更具可读性，并利用 Rust 强大的类型系统在代码库中表达更多含义，同时编译器可以保证机制生效。

代码清单 2.2 使用 match 处理 Option 值

```
fn main() {
   struct Book {
      title: String,
      isbn: Option<String>,
   }

   let book = Book {
      title: "Great book".to_string(),
      isbn: Some(String::from("1-123-456"))
   };

   match book.isbn {
      Some(i) => println!(
           "The ISBN of the book: {} is: {}",
           book.title,
           i
     ),
     None => println!("We don't know the ISBN of the book"),
  }
}
```

标准库还提供了大量可用于 Option 值的方法和特性(http://mng.bz/Xa7G)。例如，book.isbn.is_some()返回 true 或 false，表示它是否有值。

2.1.3 使用文档解决错误

通过简单的程序，可以接触到许多基本的 Rust 行为和功能，因此，下面将尝试在程序中使用之前在 Question 结构上实现的构造函数创建一个新的 question(见代码清单 2.3)。如果要在 Rust Playground 中调试，可以使用 http://mng.bz/yaNG。注意，代码清单 2.3 中

的代码无法被编译，而且会出现一些错误(见代码清单 2.4)，稍后一起解决这些错误。

代码清单 2.3　创建示例 question 并打印

```
// ch02/src/main.rs

struct Question {
     id: QuestionId,
     title: String,
     content: String,
     tags: Option<Vec<String>>,
}

struct QuestionId(String);

impl Question {
   fn new(
        id: QuestionId,
        title: String,
        content: String,
        tags: Option<Vec<String>>
   ) -> Self {
       Question {
           id,
           title,
           content,
           tags,
       }
   }
}

fn main() {
     let question = Question::new(
          "1",
          "First Question",
          "Content of question",
          ["faq"]
     );
     println!("{}", question);
}
```

在 main.rs 文件的末尾添加以上片段并运行。注意，此处使用了双冒号(::)来调用 Question 上的 new 方法。Rust 有两种在类型上实现函数的形式：

- 关联函数(associated function)
- 方法(method)

关联函数不接收&self 参数，此类函数通过两个双冒号(::)来调用，大致等同于其他编程语言中的静态函数。尽管被称为关联函数，但它们并不与一个特定的实例相关联。在另一种方式中，方法接收&self 参数，并且可以简单地通过点(.)来调用。图 2.4 展示了以上两种实现和调用的区别。

你可以在终端上使用 cargo run 命令来启动应用程序。然而，图 2.4 中的代码将返回

一长串错误。即使你已编写 Rust 多年，也有可能遇到这种情形。编译器很严格，你需要熟悉它的错误红海。

```
impl Question {
   fn new(id: QuestionId, title: String, …) -> Self {
       Question {
           id,
           title,
           content,
           tags,
       }
   }

   fn update_title(&self, new_title: String) -> Self {
       Question::new(self.id, new_title, self.content, self.tags)
   }
}
```

```
let q = Question::new(QuestionId("1".to_string()), "title".to_string(), …);
```

```
q.update_title("better_title".to_string());
```

图 2.4 关联函数(new，上面的函数)不需要&self 参数，通过双冒号(::)调用；方法(update_title，下面的函数)需要 &self 参数，通过点符号(.)调用。若要从一个 impl 代码块中调用函数，可通过该块的名称(本例中为 Question::new(...))来实现

Rust 希望生成安全且正确的代码，因此对允许编译的内容很挑剔。这样的好处是，它可以指导编码，并给出友好的错误信息，以便你快速看出错误的位置。

代码清单 2.4 给出了尝试编译当前代码后的错误信息。

代码清单 2.4 尝试编译当前代码后的错误信息

```
error[E0308]: arguments to this function are incorrect
   --> src/main.rs:27:20
    |
27  |     let question = Question::new(
    |                    ^^^^^^^^^^^^^
28  |         "1",
    |         --- expected struct `QuestionId`, found `&str`
29  |         "First Question",
    |          ---------------- expected struct `String`, found `&str`
30  |         "Content of question",
    |         --------------------- expected struct `String`, found `&str`
31  |          ["faq"],
    |         ------- expected enum `Option`, found array `[&str; 1]`
    |
= note: expected enum `Option<Vec<String>>`
              found array `[&str; 1]`
note: associated function defined here
  --> src/main.rs:11:8
    |
11  |     fn new(
    |        ^^^
12  |         id: QuestionId,
    |          --------------
13  |         title: String,
    |          -------------
14  |         content: String,
    |         ---------------
15  |         tags: Option<Vec<String>>,
    |         -------------------------
```

Rust 编译器标识了问题的确切位置和内容

双引号之间的文本不是 String，而是 &str

编译器期望的 tags 不是一个数组，而是一个枚举 Option

```
help: try using a conversion method
   |
29 |            "First Question".to_string(),
   |                            ++++++++++++
help: try using a conversion method
   |
30 |            "Content of question".to_string(),
   |                                  ++++++++++++
error[E0277]: `Question` doesn't implement `std::fmt::Display`
--> src/main.rs:33:20
   |
33 |     println!("{}", question);
   |                    ^^^^^^^^ `Question` cannot
       be formatted with the default formatter
   |
   = help: the trait `std::fmt::Display` is not implemented for `Question`
   = note: in format strings you may be able to use `{:?}` (or {:#?}
           for pretty-print) instead
   = note: this error originates in the macro `$crate::format_args_nl`
           (in Nightly builds, run with -Z macro-backtrace for more info)

Some errors have detailed explanations: E0277, E0308.
For more information about an error, try `rustc --explain E0277`.
error: aborting due to 5 previous errors; 1 warning emitted
```

无法在控制台打印 question

你可以利用以上错误信息来学习更多关于 Rust 语言及其特性的知识，以便为未来构建稳固的 Web 应用程序做好准备。你会发现有些错误会导致两次报错，还有一些错误是重复的，通过解决前一个错误就可以消除这些错误。

最佳实践是始终从第一个错误开始纠正，因为这可能是后面出现错误的原因。因此，下面回顾一下第一个问题(见代码清单 2.5)，看看如何解决它。

代码清单 2.5　第一个编译错误

```
 --> src/main.rs:27:20
   |
27 |       let question = Question::new(
   |                      ^^^^^^^^^^^^^
28 |           "1",
   |           --- expected struct `QuestionId`, found `&str`
```

以上错误描述了两个问题：首先，需要传递自定义的 QuestionId 类型，而不是&str；其次，根据代码清单 2.3 中的结构体定义，需要封装 String，而不是&str。

不妨趁机查看一下&str 的文档(https://doc.rust-lang.org/std/primitive.str.html)，看看可以如何解决这个问题。第一次打开 Rust 的文档时可能会心生畏惧，但不要担心，因为你只需要时间来适应，看一下图 2.5。

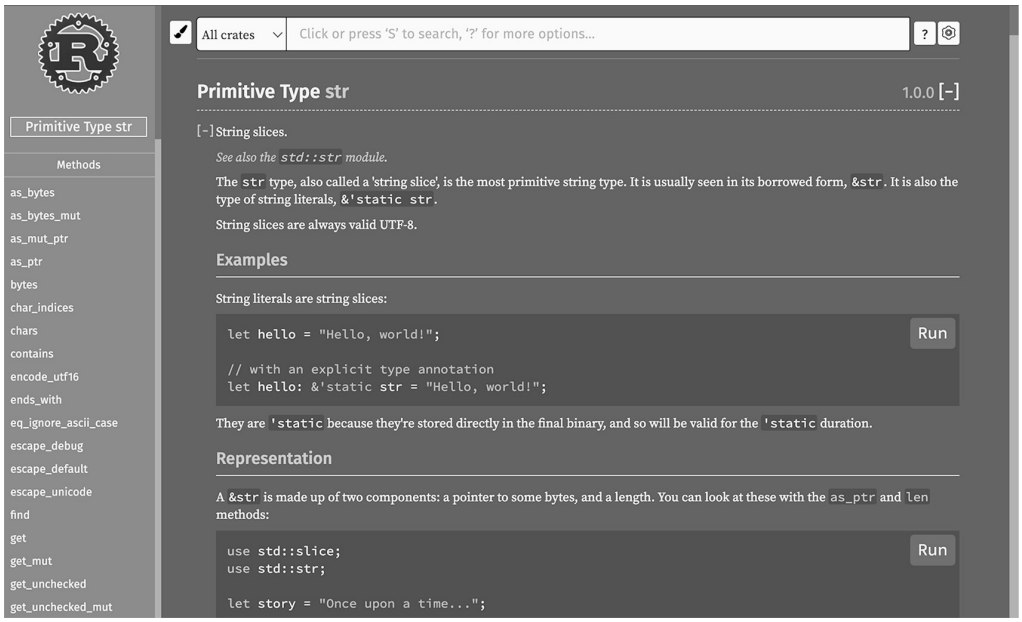


图 2.5 Rust 的文档虽然复杂，但它允许快速浏览，并提供大量的信息

文档包含一个主窗口和左侧边栏。主窗口通常展示你正在查看的类型，侧边栏则提供实现细节以及在这个类型上实现的方法和特质。务必了解以下内容：

- 方法(method)
- 特质实现(trait implementation)
- 自动特质实现(auto trait implementation)
- 全局特质实现(blanket implementation)

本地和离线浏览文档

如果在火车上、飞机上，或者只是想在本地拥有 Rust 的文档，可通过 rustup 安装文档(doc)组件：

```
$ rustup component add rust-docs
```

之后，你可以在默认浏览器中打开标准库中的文档：

```
$ rustup doc -std
```

也可从代码库中生成文档，其中包含所有 Cargo 依赖关系：

```
$ cargo doc -open
```

这将包含你定义的结构体和函数，即使你没有明确为它们创建文档，也是如此。

在继续探讨并选择一种方法将&str 转化为 String 之前，必须了解这两者之间的区别，以及为什么 Rust 对它们的处理方式不同。

2.1.4　在 Rust 中处理字符串

在 Rust 中，String(http://mng.bz/M0w7)和&str(https://doc.rust-lang.org/std/primitive.str.html)的主要区别在于，String 是可调整大小的。String 是一个字节的集合，它是作为向量实现的。你可以在源代码中查看 String 的定义，如代码清单 2.6 所示。

代码清单 2.6　标准库中 String 的定义

```
// Source: https://doc.rust-lang.org/src/alloc/string.rs.html#294-296

pub struct String {
    vec: Vec<u8>,
}
```

通过 String::from("popcorn")，可以创建字符串，并且可以在创建后修改它们。可以看到，String 底层是一个向量，这意味着可以根据需求在这个向量中移除或插入 u8 值。

&str(字符串字面量)是 u8 值(文本)的表示，不允许修改。你可以将其视为指向底层字符串的固定大小窗口。2.1.5 节将解释 Rust 中的所有权概念，但现在重要的是理解，如果拥有一个 String，则“拥有”这块内存并可以修改它。

在处理&str 时，处理的是指向内存空间的指针，可以读取但不能修改。这使得使用&str 的内存效率更高。一个经验法则：如果创建的函数只需要读取字符串，应使用&str 作为参数类型；如果想拥有并修改它，则使用 String。

如图 2.6 所示，字符串字面量和字符串都存在于堆(heap)中，但在栈(stack)中分配了不同的指针。你不需要详细了解堆和栈的概念，但为了将来更好地理解编译器的错误，不妨熟悉一下这个概念。下面的“栈与堆”部分补充说明了主要概念。

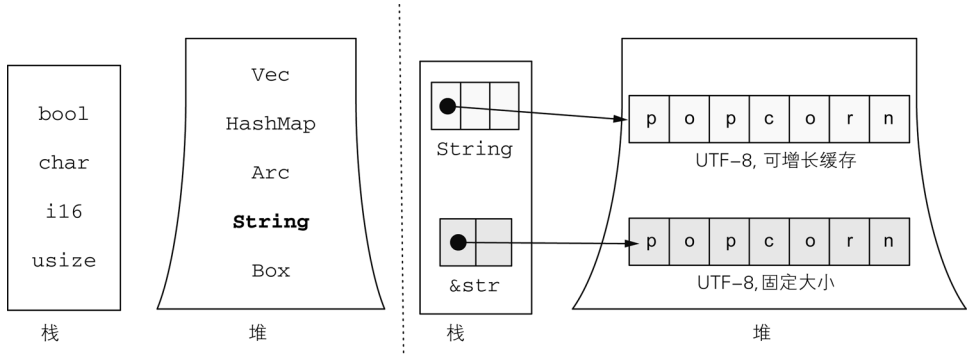


图 2.6　在 Rust 中，原始类型被存储在栈中，而更复杂的类型被存储在堆中。String 和&str 指向更复杂的数据类型(UTF-8 值的集合)。&str 有一个显示堆上位置的胖指针(内存地址和长度字段)，而 String 指针不仅有地址和长度字段，还有一个容量字段

栈与堆

操作系统为其处理的变量和函数分配内存。操作系统必须保存函数并调用它们，以

处理并重用数据。为此，操作系统使用了两个概念：栈和堆。

栈通常由程序控制，每个线程都有自己的栈。它存储地址、寄存器值和程序值(变量、参数和返回值)。基本上，所有具有固定大小(或填充到正确的模数)的内容都可以存储在栈上。

堆的特征更明显(尽管可以有多个堆)，堆上的操作更昂贵。数据没有固定的大小，可以被分割成多个块，因此读取操作可能需要更多时间。

str 表示什么

去掉&符号的&str 就是 str，这是实际处理的数据类型。但是，str 是一个不可变的、没有固定长度的 UTF-8 字节序列。由于其长度未知，你需要通过指针来处理它(可以参考 Stack Overflow 上的一个很好的解释：http://mng.bz/aP9z)。

也可引用 Rust 文档的说法："str 类型，也称为'字符串切片'，是最原始的字符串类型，通常以借用形式&str 出现。" 2.1.5 节将详细介绍 Rust 中的借用。

简短总结：

- 如果需要拥有和修改文本，则创建一个 String 类型。
- 在只需要查看文本时，使用&str。
- 通过结构体创建新的数据类型时，通常创建 String 类型的字段。
- 当向函数传递字符串/文本时，通常使用&str。

2.1.5 深入理解移动、借用和所有权

如果对 String 和&str 进行进一步的比较，将涉及 Rust 的一个重要概念：所有权。简单而言，Rust 希望在不使用垃圾回收器或不需要开发人员小心翼翼地管理内存的情况下实现安全的内存管理。

每个计算机程序都要处理内存，所以要么垃圾回收器负责清理并确保没有变量指向空值，要么开发人员必须考虑这个过程。Rust 选择了第三种方案：引入一个不同的概念。

代码清单 2.7 至代码清单 2.9 最好在 Rust Playground 中运行：http://mng.bz/gRml。你可以尝试各种组合，看看是否可以自己修复错误。

代码清单 2.7 给&str 赋值

```
fn main() {
    let x = "hello";
    let y = x;

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

当运行此程序时，会在控制台上看到“hello”被打印出来。现在尝试使用 String。

代码清单 2.8　给 String 赋值

```
fn main() {
    let x = String::from("hello");
    let y = x;

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

会得到以下错误信息：

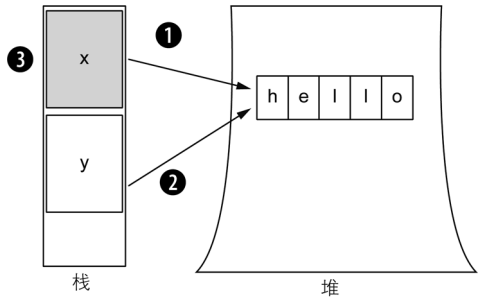
```
error[E0382]: borrow of moved value: `x`
 --> src/main.rs:5:20
  |
2 |     let x = String::from("hello");
  |         - move occurs because `x` has type `String`,
  |             which does not implement the `Copy` trait
3 |     let y = x;
  |             - value moved here
4 |
5 |     println!("{}", x);
  |                ^ value borrowed here after move
```

为什么会遇到这个错误？代码清单 2.7 创建了一个类型为&str 的新变量：一个指向字符串切片(str)的引用(&)，值为 hello。如果将这个变量赋值给一个新变量(y = x)，就创建了一个指向内存中相同地址的新指针。现在有两个指针指向同一个底层值。

如图 2.6 所示，在创建字符串切片后不能更改它，因此，它是不可变的。你可以打印两个变量，它们都是有效的，并且都将指向保存单词 hello 的底层内存。

当处理实际字符串(而不是引用)时，情况会发生变化。代码清单 2.8 创建了一个复杂类型——String。Rust 编译器当前强制实行单一所有权原则。当像之前那样使用 y = x 重新分配 String 时，将所有权从变量 x 转移到 y。

由于所有权从 x 转移到 y，因此 x 退出作用域，并且 Rust 在内部将其标记为未初始化(uninit)状态(https://doc.rust-lang.org/nomicon/drop-flags.html)。图 2.7 阐释了这个概念。如果尝试打印变量 x，你会发现由于所有权已经转移到了 y，现在 x 已经不存在了且没有值。



❶ let x = String::from("hello");

❷ let y = x;

❸ //mark x as uninitialized

图 2.7 当将复杂类型重新分配给新变量时，Rust 复制指针信息并将所有权转移到新变量。旧变量不再被需要并退出作用域

下面探讨另一个需要理解 Rust 所有权原则的领域：函数处理。将变量传递给函数时会将底层数据的所有权移交给函数。在 Rust 中，有不同的处理方式：

- 将所有权移交给函数并从函数返回一个新变量。
- 传递变量的引用以保留所有权。

代码清单 2.9 展示了通过函数修改 String 对象的例子。将一个 String 的可变引用(这样就可以修改它)传递给一个函数。现在，该函数可以访问底层数据并进行修改。一旦该函数完成任务，将在 main 函数内部重新获得所有权，因此可以打印 address。

可通过此 Playground 链接(http://mng.bz/epBz)尝试此示例中的各种选项。

代码清单 2.9 将所有权移交给函数

```
fn main() {
    let address = String::from("Street 1");

    let a = add_postal_code(address);

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

fn add_postal_code(mut address: String) -> String {
    address.push_str(", 1234 Kingston");
    address
}
```

声明一个变量并将其赋值为一个字符串(String)

将 address 传递给函数，并将返回值分配给名为 a 的变量

打印更新后的 address

函数参数也必须声明为可变的(mut address: String)，后续才能修改它

push_str 方法直接修改了字符串

返回修改后的字符串(address)

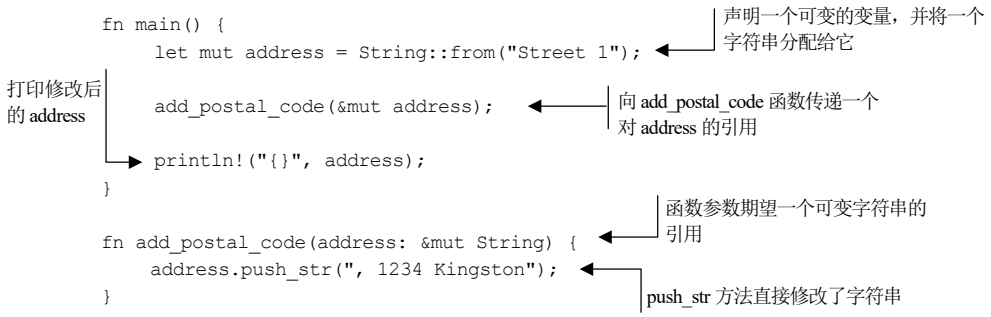
下面详细看看上面的例子。首先，默认情况下，变量是只读的，如果想要改变(mutate)它们，必须在创建新变量时在 let 关键字后添加 mut。然后，调用 add_postal_code 函数，它将把邮政编码添加到刚刚创建的 String 对象中。

通过将 address 传递给 add_postal_code 函数，将所有权移交到了这个函数。当尝试在这行代码之后打印 address 时，将会像代码清单 2.8 一样出现错误。add_postal_code 函数期望一个可变的 String 对象(通过参数中的 mut 关键字)并通过.push_str 函数向其添加新字符。然后，它返回更新后的字符串，并将其重新分配给变量 a。

可以使用与之前完全相同的名称(address)，而不是为这个新变量找个新名字。这是 Rust 的一个特性，被称为变量遮蔽(variable shadowing，http://mng.bz/p6ZG)，因此不必不断地为要修改的变量寻找新的名称。

代码清单 2.10 展示了一种略有不同的方式，该方式在 Rust 代码库中可能更常见。与其传递 address 的值并失去所有权，不如传递一个引用。因此，我们保留了所有权，并在函数需要的时候将所有权借给它。

代码清单 2.10　传递引用



add_postal_code 函数在函数体的执行期间借用所有权。因此，在你尝试打印它之前，address 变量不会超出范围(如前所述)。

以上内容总结了关于 String 与&str 以及 Rust 中所有权原则的探讨。一个简单的错误揭示了这门语言的许多内部动态。现在你能够修复代码清单 2.5 中的第一个错误(期望一个 QuestionId 类型而不是&str 类型)。

2.1.6　使用和实现 trait

编译器告诉你，它期望的是一个 QuestionId，而不是一个&str 类型。打开&str 的文档(https://doc.rust-lang.org/std/primitive.str.html)，看看如何将其转化为一个 String 类型的值。向下滚动时，你会看到一个名为 ToString 的 trait 实现。必须单击 ToString 旁边的[+]来获得更多的细节(见图 2.8)。单击 Read more 以浏览 to_string 函数定义，如果一个类型实现了 ToString trait(&str 也实现了)，便可以使用。

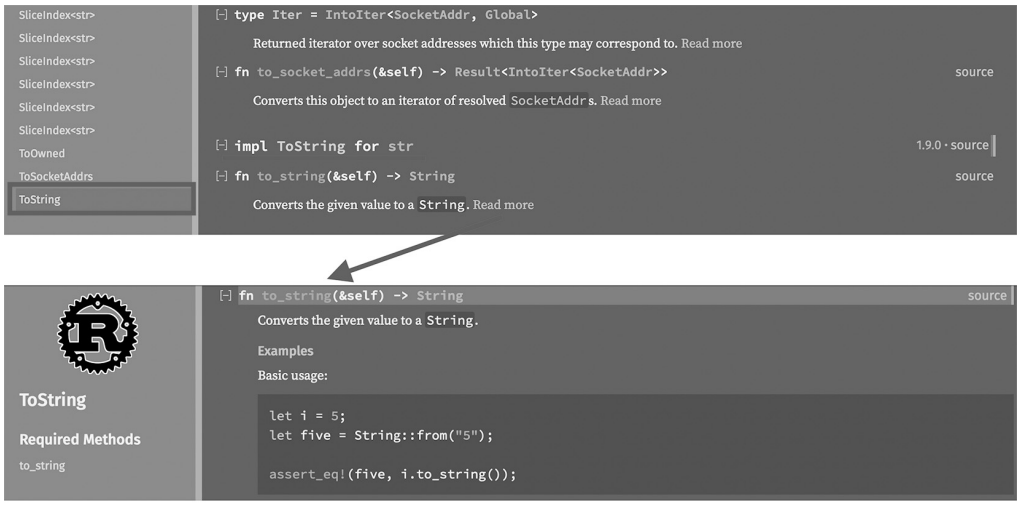


图 2.8 侧边栏提供了某一类型可用的所有方法，有时需要进一步探索才能找到实现细节

trait

在 Rust 中，若要实现共享行为，可以使用 trait。这大致相当于其他语言中的接口。然而，在 Rust 中，你可以在你没有定义的类型上实现 trait。

你可以使用 trait 为应用程序中的多个类型创建所需的行为，还可以使用 trait 标准化行为。例如，在将一种类型转换为另一种类型时，你可以使用 trait(如标准库中的 ToString trait)。

trait 的另一个优点是，它们使你能够在不同的上下文中使用类型。Rust 程序可以是泛型的，接收所有表现出某种行为方式的类型。想象一个餐厅，它接收所有能在桌子下喝水的动物。你的 Rust 程序中的函数可以有类似的行为。例如，它们可以返回具有某种特征的类型。只要你的类型实现了这些特征，这些类型就可以被返回。

当你想要在控制台上输出自定义的结构体时，你会很快在 Rust 中实现 trait，例如，可以使用 derive 宏(在编译时为你编写所有自定义的 trait 实现)。

你可以使用 to_string 将&str 转换为 String。该方法接收&self，这表明可以在所有定义的&str 上使用点符号调用它，并返回 String。这有助于解决许多错误。此外，尝试将 ID 封装在 QuestionId 中，因为这是在结构体中定义的方式，如代码清单 2.11 所示。

代码清单 2.11 将&str 转化为 String

```
// ch_02/src/main.rs

struct Question {
    id: QuestionId,
    title: String,
    content: String,
    tags: Option<Vec<String>>,
}
```

```
struct QuestionId(String);

impl Question {
     fn new(
          id: QuestionId,
          title: String,
          content: String,
          tags: Option<Vec<String>>
     ) -> Self {
        Question {
            id,
            title,
            content,
            tags,
        }
     }
}

fn main() {
     let question = Question::new(
          QuestionId("1".to_string()),
          "First Question".to_string(),
          "Content of question".to_string(),
          ["faq".to_string()],
     );
     println!("{}", question);
}
```

只需要在命令行上执行 cargo run，你便可以看到取得的一些进展。现在只剩下两个错误，不妨再次查看第一个错误(见代码清单 2.12)。

代码清单 2.12　错误地返回了数组而不是向量

```
error[E0308]: mismatched types
  --> src/main.rs:25:9
    |
25 |          ["faq".to_string()],
   |           ^^^^^^^^^^^^^^^^^^
               expected enum `Option`,
               found array of 1 element
|
= note: expected enum `Option<Vec<String>>`
          found array `[String; 1]`
```

报错信息似乎是由两个错误组成的。编译器期望接收一个包含 Vec 的 Option 枚举，却传递了一个数组。下面再次使用 docs.rs 文档查找 Option(https://doc.rust-lang.org/std/option/index.html)。

根据提供的示例可知，需要将标签(tag)封装在 Some 中。Question 结构体需要字符串向量，而不是数组。在 Rust 中，向量和数组并不相同。如果想要的东西类似于其他语言中的数组，你可能需要在 Rust 中使用 Vec。

文档也提供了相应的帮助，它介绍了在 Rust 中创建向量的两种方式。可以使用

Vec::new，然后使用.push 插入元素，或者使用 vec!宏。不妨借助代码清单 2.13 中的示例来相应地更新代码。

代码清单 2.13 封装及创建向量

```
fn main() {
     let question = Question::new(
          QuestionId("1".to_string()),
          "First Question".to_string(),
          "Content of question".to_string(),
          Some(vec!("faq".to_string())),
     );
     println!("{}", question);
}
```

再次运行该程序，可以看到其中只剩下一个错误，如代码清单 2.14 所示。

代码清单 2.14 面临的最后一个错误：缺少 trait 实现

```
error[E0277]: `Question` doesn't implement `std::fmt::Display`
  --> src/main.rs:27:20
    |
27 |      println!("{}", question);
    |                     ^^^^^^^^ `Question` cannot
           be formatted with the default formatter
    |
    = help: the trait `std::fmt::Display` is not implemented for `Question`
    = note: in format strings you may be able to use `{:?}` (or {:#?}
           for pretty-print) instead
    = note: required by `std::fmt::Display::fmt`
    = note: this error originates in a macro
         (in Nightly builds, run with -Z macro-backtrace for more info)

error: aborting due to previous error
```

最后一个错误似乎是缺少一个名为 std::Display 的 trait 实现。在 Rust 中，可通过 println!宏来打印变量，并为想要打印的每个变量添加大括号({})：

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

这将调用 Display trait 实现中的 fmt 方法：http://mng.bz/O6wn。上面的错误信息还建议使用{:?}而不是常见的大括号({})。下面的“Display 与 Debug”补充说明了两者之间的区别。

Display 与 Debug

Rust 中的所有原始类型都实现了 Display trait(http://mng.bz/YKZN)。Display trait 规定提供的实现应以人类可读的形式显示数据。对于数字和字符串，这很容易，但是对于向量，应该如何做？任何东西都可以放在向量里面，因为 Vec<T>是一个数据类型的通用容器。对于这些用例(像向量这样的复杂数据结构)，Rust 标准库使用了 Debug trait。

对开发人员来说，这两种方式的区别如下：当处理字符串和数字时，打印是通过大括号({})完成的，如 println!("{}", 3)。当处理更复杂的数据结构(如结构体或 JSON 值)时，也可以使用{:?}，它可以调用 Debug trait，如 println! ("{:?}", question)。

Debug trait 可通过在结构体上方放置 derive 宏来实现。因此，不一定必须自己实现 Debug trait，也可以打印出相应的数据结构。

```
#[derive(Debug)]
struct Question {
   title: String,
   ...
}
```

可通过添加#来进行格式打印，如 println!("{:#?}", question)。数据结构将以多行形式进行展示，而不是展示一个长字符串。

和前面的 ToString trait 一样，Display 也是 Rust 标准库中所有基本类型都会实现的一个 trait。这使得编译器知道如何显示这些数据类型(将它们转换为人类可读的输出信息)。自定义类型不是标准库的一部分，因此没有实现这个特性。

该如何搜索并实现 Display trait 呢？答案还是 Rust 文档。可以搜索 Display (http://mng.bz/G1wq)，然后单击[src]以找到对应的实现。注释部分展示了一个实现的例子(见代码清单 2.15)。

代码清单 2.15　Display trait 的实现示例

```
// https://doc.rust-lang.org/src/core/fmt/mod.rs.html#743-767

/// # Examples
///
/// ```
/// use std::fmt;
///
/// struct Position {
///   longitude: f32,
///   latitude: f32,
/// }
///
/// impl fmt::Display for Position {
///    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
///        write!(f, "({}, {})", self.longitude, self.latitude)
///    }
/// }
///
/// assert_eq!("(1.987, 2.983)",
///            format!(
///             "{}",
///             Position {
///               longitude: 1.987, latitude: 2.983,
///             }
///           )
```

```
/// );
/// ```
```

通过 Rust 文档中的代码示例(见代码清单 2.15)，你可以逐步学习如何使用文档在自己的类型上实现 trait。先将以上基本示例复制到代码库中，并尝试用自己的结构体(Question)对示例中的结构体(Position)进行替换。以下代码片段在 Question 上实现了 Display trait，粗体字表示进行的修改：

```
impl std::fmt::Display for Question {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> Result<(), std::fmt::Error>
    {
      write!(
          f,
          "{}, title: {}, content: {}, tags: {:?}",
          self.id, self.title, self.content, self.tags
      )
    }
}
```

当尝试通过 println!宏来打印一个 question 时，会调用在 Question 上实现的 fmt 函数。函数会调用 write!宏，将文本打印到控制台，你可通过传递参数来定义具体的内容。

不过，需要注意两点：其一，Question 结构体有一个名为 QuestionId 的自定义结构体，默认情况下不实现 Display trait；其二，对于 tags，我们使用的是一个更复杂的向量结构。不能在向量这种更复杂的结构上使用 Display trait，而必须使用 Debug trait。代码清单 2.16 展示了实现 Display trait 和 Debug trait 的 main.rs 文件。

代码清单 2.16 将 Display trait 的实现添加到 Question 上

```
…

impl std::fmt::Display for Question {
    fn fmt(&self, f: &mut std::fmt::Formatter)
           -> Result<(), std::fmt::Error> {
         write!(
           f,
           "{}, title: {}, content: {}, tags: {:?}",
           self.id, self.title, self.content, self.tags
       )
    }
}

impl std::fmt::Display for QuestionId {
   fn fmt(&self, f: &mut std::fmt::Formatter)
        -> Result<(), std::fmt::Error> {
      write!(f, "id: {}", self.0)
   }
}

impl std::fmt::Debug for Question {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>)
            -> Result<(), std::fmt::Error> {
```

```
            write!(f, "{:?}", self.tags)
        }
    }

    …
```

Debug 的实现看上去与 Display trait 很相似。在 write!宏中使用{:?}替代{}。如果每次都要通过上面的代码来实现和展示一个自定义类型，该过程会非常烦琐。

Rust 标准库为此提供了一个名为 derive 的过程宏(procedural macro)，你可以将其(通过#[derive])放在结构体定义之上。Display trait 的文档还介绍了一个重要信息：“Display 与 Debug 类似，但 Display 用于面向用户的输出，所以不能被派生。”

声明宏(declarative macro)

在 Rust 中，可通过名字后面的感叹号来识别宏。这表示声明宏或类似函数的过程宏(另一种宏类型)。在 Rust 中一开始就会用到的最常见的声明宏是 println!，它可以将文本打印到控制台。

宏会将封装的代码转换为标准的 Rust 代码。这会在编译器将所有的 Rust 代码组合成二进制文件之前完成。

你还可以创建自己的宏，尽管该过程中几乎没有什么规则与限制。但应该在熟练掌握 Rust 标准之后，当想让自己的生活变得更轻松时，再去创建自己的宏。

因此，继续为类型派生 Debug trait。然后，需要在 println!宏中通过{:?}(而不是{})来使用 Debug。更新后的代码如代码清单 2.17 所示。运行程序后，你将在控制台上看到 question 的内容(暂时忽略警告)。

代码清单 2.17　使用 derive 宏来实现 Debug trait

```
#[derive(Debug)]
struct Question {
    id: QuestionId,
    title: String,
    content: String,
    tags: Option<Vec<String>>,
}

#[derive(Debug)]
struct QuestionId(String);

impl Question {
  fn new(
      id: QuestionId,
      title: String,
      content: String,
      tags: Option<Vec<String>>
  ) -> Self {
     Question {
         id,
         title,
         content,
```

```
            tags,
        }
    }
}

fn main() {
    let question = Question::new(
        QuestionId("1".to_string()),
        "First Question".to_string(),
        "Content of question".to_string(),
        Some(vec!("faq".to_string())),
    );
    println!("{:?}", question);
}
```

以上代码仍有改进空间。我们已经将 question 的 ID 抽象为 QuestionID 结构体，但仍然要记住 QuestionID 需要一个 String 作为输入。可以隐藏这个实现细节，让用户能更方便地生成 question 的 ID。

Rust 提供了一些用于常见功能的 trait。其中之一是 FromStr trait，它类似于之前讨论过的 ToString trait。可以按照以下方式使用 FromStr：

```
let id = QuestionId::from_str("1").unwrap(); // from_str() can fail
```

以上代码简单地表达了："从类型&str 创建类型 *X*。" Rust 没有隐式类型转换，只有显式类型转换。因此，你总是需要指明是否要将一个类型转换为另一个类型，参见代码清单 2.18。

代码清单 2.18 将 FromStr trait 实现添加到 QuestionId

```
use std::io::{Error, ErrorKind};
use std::str::FromStr;

…

impl FromStr for QuestionId {
    type Err = std::io::Error;

    fn from_str(id: &str) -> Result<Self, Self::Err> {
        match id.is_empty() {
            false => Ok(QuestionId(id.to_string())),
            true => Err(
              Error::new(ErrorKind::InvalidInput, "No id provided")
            ),
        }
    }
}

…
```

trait 的签名允许接收一个&str 类型，并返回自定义的类型(QuestionId)或在 ID 为空的情况下返回一个错误。将参数命名为 id，类型为&str(因为这是将接收的类型)。参数的名

称(本例中是 id)可以是任何内容。然后通过匹配判断 id 是否不为空，同时返回一个包含一个字段的 QuestionId 类型，并将其转换为 String，正如在结构体中指定的那样。

接下来我们就可以改变在 main 函数中创建 question ID 的方式。不再使用.to_string，而是在 QuestionId 上调用::from_str。可以在 trait 的实现(见代码清单 2.19)中看到，from_str 不接收&self 类型，因此它不是可以通过点(.)来调用的方法，而是需要通过双冒号(::)来调用的关联函数。

代码清单 2.19　使用 FromStr trait 通过&str 创建 QuestionId

```
…

fn main() {
       let question = Question::new(
            QuestionId::from_str("1").expect("No id provided"),
            "First Question".to_string(),
            "Content of question".to_string(),
            Some(vec!("faq".to_string())),
       );
       println!("{:?}", question);
}
```

2.1.7　处理结果

为什么需要在函数后添加.expect？仔细观察，可以看到 FromStr trait 的实现返回了一个 Result。与 Option 类型相似，Result 类型有两种变体：成功或错误。在成功的情况下，通过 Ok(value) 封装值。在错误的情况下，通过 Err(error) 封装错误。Result 类型也是通过枚举类型实现的，它的结构如代码清单 2.20 所示。

代码清单 2.20　Rust 标准库中 Result 的定义

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

就像 Option 一样，Result 也实现了许多方法和 trait。其中一个方法是 expect，根据文档，它返回包含的 Ok 值。可以看到，该方法实际上是将 QuestionId 包装在 Ok 中返回的，这意味着 expect 返回其中的值；如果发生错误，它会使用指定的错误信息来触发 panic。

适当的错误处理需要一个 match 语句，从 from_str 函数中接收一个错误，并以某种方式处理它。不过，在这个简单的例子中，目前的处理方式已经足够了。另一个常见的方法是 unwrap，它更具可读性，但在没有指定错误信息的情况下会触发 panic。

在生产环境中，不要使用 unwrap 或 expect，因为它们会导致 panic 并使应用程序崩溃。始终使用 match 来处理错误情况，或者确保自己能捕获错误并优雅地返回错误。

你也可以轻松地在自己的函数中使用 Result 枚举。返回的签名类似于-> Result<T, E>，

其中，T 是你想要返回的数据，E 是错误(可以是自定义的，也可以是来自标准库的)。

Result 与 Option 类似，主要区别在于 Error 变体。当数据可以存在但不一定存在(缺少的数据不会引起问题)时，使用 Option。当确实期望数据存在并且必须主动管理不存在的情况时，使用 Result。

之前的代码就是一个简单的例子。在 Question 结构体中将 tags 标记为可选的，因为即使没有它们，你也可以创建 question。然而，ID 是必需的，如果 QuestionId 结构体无法从&str 创建，则会创建失败，且必须向此方法的调用者返回错误。在完成了基本结构和类型实现之后，让我们为应用程序添加一个 Web 服务器，以便为用户提供第一个虚拟数据。

2.2 创建 Web 服务器

在讨论 Web 服务器的构建时，我们已经大致了解了 Rust 提供的特性和它不具备的特性。下面回顾以下要点：

- Rust 不附带可以处理异步后台工作的运行时。
- Rust 提供了表示异步代码块的语法。
- Rust 包含具有状态和返回类型的 Future 类型。
- Rust 实现了 TCP(和 UDP)，但没有实现 HTTP。
- 选择的 Web 框架附带了实现的 HTTP 和其他所有功能。
- 运行时由选择的 Web 框架决定。

Web 框架在幕后决定了很多东西：运行时、HTTP(及 HTTP 服务器实现)的抽象，以及如何将请求传递给路由函数。因此，你需要确保选择的 Web 框架的设计理念及其选择的运行时是和自己的需求匹配的。

为了让你对将要讨论的语法和主题有所了解，代码清单 2.21 简单呈现了本章结束时将完成的示例。阅读本章的最后一部分后，你将了解这段代码的功能。当前只是想突出部分内容。

代码清单 2.21 使用 Warp 的最小可用 Rust HTTP 服务器

```
use warp::Filter;

#[tokio::main]
async fn main() {
  let hello = warp::get()
     .map(|| format!("Hello, World!"));

  warp::serve(hello)
       .run(([127, 0, 0, 1], 1337))
       .await;
}
```

代码的第 3 行，即#[tokio::main]，表示正在使用的运行时。2.2.1 节将讨论运行时。

为了让你有个直观的印象，上面展示了它在 Rust 源代码中的样子。接下来，第 4 行是 main 函数，我们将其标记为异步函数。这样做是为了同时处理多个请求(由于有了运行时)，并且在异步函数内部，可以使用.await 关键字来表示该函数在本质上是异步的，不会立即产生结果。这些便是 Rust 异步编程中的 3 个(共 4 个)关键组件。

2.2.1　同时处理多个请求

当编写服务器应用程序时，你通常需要同时为多个客户端提供服务。即使并非所有连接都在同一毫秒到达，你也需要时间从数据库中读取数据或在硬盘上打开文件。

与其等待每个线程完全结束，让另外数百个(甚至数千个)请求等待并堆积，不如选择触发一个线程(比如数据库查询)，并确保你在它完成时得到通知。与此同时，可以为其他客户端提供服务。

绿色线程(green thread)

在谈论异步编程时，总会涉及线程。线程是由进程创建(生成)的，并在其内部运行。线程通常由操作系统(在内核内部)处理，因此从用户的角度看，管理线程的成本较高(因为需要不断中断内核)。

因此，绿色线程(http://mng.bz/nemK)的概念应运而生。绿色线程完全位于用户空间，并由运行时管理。

通过编写异步应用程序，你可以同时处理多个请求。回顾一下第 1 章中的 minimal-tcp 代码(http://mng.bz/z52a)，你会发现它以阻塞的方式处理一个接一个的流(stream)。处理完一个流之后再去处理下一个流。

在多线程环境中，可以把每个流放在自己的线程上，让它们在后台完成计算，然后将它们放回前台，并把答案发回给请求的客户端。另一个设计方案是使用单线程，在可能的情况下不断地执行任务。关键是，一个耗时较长的方法可将控制权交还给运行时，并发出信号，表示它需要更长的时间来完成。运行时可以执行其他计算，并检查这个耗时较长的方法是否已经完成计算。

要异步处理传入的 HTTP 请求，需要一种理解异步概念的编程语言，而且该语言应提供相应的类型和语法，以便我们标记应该异步执行的代码。我们还需要一个运行时，它可以接收代码并知道如何以非阻塞方式执行它。图 2.9 显示了所需的成分。

图 2.9　异步编程环境需要 4 个要素(语法、类型、运行时和内核抽象)才能工作

异步编程环境的 4 个要素总结如下：

- 通过 epoll/select/poll 使用内核的异步读写 API(详细信息见 http://mng.bz/09zx)。
- 能在用户空间中关闭长耗时任务，并在任务完成时发出通知，以便继续工作。这意味着运行时能创建和管理绿色线程。
- 编程语言的语法允许在代码中标记异步块，使得编译器能够理解如何处理它们。
- 标准库中的特定类型允许互斥访问和修改。与存储特定值的类型(如 number 类型)不同，异步编程类型需要在存储值的同时存储长耗时任务的当前状态。

Rust 最初使用了绿色线程，但后来因运行时占用空间较大而放弃了它们。因此，Rust 没有运行时和对异步内核 API 的抽象。这与 Node.js 和 Go 不同，它们都自带原生运行时和对内核 API 的抽象。

Rust 提供了语法和类型。Rust 本身支持异步概念，并提供了足够的组件来构建运行时并对内核 API 进行抽象。

2.2.2 Rust 的异步环境

为了使 Rust 占用更小的空间，其开发人员决定使其不包含任何运行时以及对内核异步 API 的抽象。这使得程序员有机会选择符合项目需求的运行时。这也使 Rust 在未来的运行时发生巨大进步时变得更具前瞻性。

图 2.8 展示了异步编程的主要要素。Rust 提供了语法和类型。经过充分测试的运行时(如 Tokio 和 async-std)也是可用的，你还可使用 Mio 对异步内核 API 进行抽象。图 2.10 展示了 Rust 生态系统的组件。

图 2.10 Rust 异步编程生态系统的组件

对于语法，Rust 提供了 async 和 await 的组合作为关键字。可以将函数标记为 async，以便在其中使用 await。使用了 await 的函数会返回 Future 类型，它具有在成功执行时返回的值的类型，以及一个名为 poll 的方法，该方法执行长耗时的线程并返回 Pending 或 Ready。Ready 状态可以具有 Error 或成功的返回值。

通常情况下，你不需要了解 Future 类型的细节。它有助于你进一步理解底层系统，以便你在需要时创建一个，但在开始时，你只需要了解 Future 为什么存在，以及它如何与生态中的其他部分一起使用。

在 Rust 中，运行时的选择对每个异步应用程序来说都很重要。运行时将包含对内核 API 的抽象(在大多数情况下，这是一个名为 Mio 的库)。但是，让我们先看一下 Rust 自带的语法和类型。

2.2.3　Rust 处理 async/await

Rust 在运行时之上集成了两个组件。第一个是 async/await 语法。下面回顾一下第 1 章中执行异步 HTTP 调用的一个代码片段(见代码清单 2.22)。

代码清单 2.22　async HTTP 调用示例

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

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let resp = reqwest::get("https:/ /httpbin.org/ip")
        .await?
        .json::<HashMap<String, String>>()
        .await?;
    println!("{:#?}", resp);
    Ok(())
}
```

注意，为了让以上代码片段正常工作，必须在 Cargo.toml 文件中添加 Tokio 和 Reqwest 包。许多 Rust 包将其逻辑分割成不同的功能，这使得应用程序可以只包含较小的功能子集，而不必包含不需要的代码。

```
[dependencies]
reqwest = { version = "0.11", features = ["json"] }
tokio = { version = "1", features = ["full"] }
```

特性标记

在 Cargo.toml 文件中添加 Tokio 到依赖项时，需要添加特性标志。特性标志允许开发人员仅添加一个包的子集，从而节省编译项目的时间并减小项目占用的空间。

并非所有的包都支持特性标志，但有些确实支持。注意，如果你添加一个包并希望使用某些特性，编译器不会通知你该特性未包含在 Cargo.toml 文件中。对新手来说，最安全的做法是添加一个包的所有特性，完成后，看看你是否可通过仅使用某些特性来减少你拉取的代码量。

特性标志的命名没有标准化，包的所有者可以为其特性命名。对于 Tokio，以下代码使用特性标志 full:

```
tokio = { version = "1", features = ["full"] }
```

函数调用 reqwest::get("https://httpbin.org/ip")返回一个 future，它包装了返回类型。此调用以对象的形式返回当前的 IP 地址，该对象具有一个键和一个值。在 Rust 中，这可通过哈希映射来表示：HashMap<String, String>。Reqwest 包默认返回 Future(https://docs.rs/reqwest/latest/reqwest/#making-a-get-request)。如果想以阻塞的方式发出 HTTP 请求，可以使用 reqwest::blocking 客户端：

```
https://docs.rs/reqwest/latest/reqwest/blocking/index.html
```

我们期望一个包装在 future 内部的哈希映射作为响应：Future<Output=HashMap<String, String>>。然后，我们可以在 future 上调用 await，这样运行时就会接管它，并尝试执行其中的功能。假设这将花费较长的时间，因此运行时将在后台处理任务，并在读取文件时填充内容变量。

你通常不会遇到需要自己定义 future 的情况，至少在开始使用 Rust 和 Web 服务时不会。重要的一点是，在使用包或其他人的代码时，如果函数被标记为 async，则必须使用 await。

这种语法的目的是使异步 Rust 代码对程序员来说看起来像同步的、阻塞的代码。在后台，Rust 将这段代码转换为一个状态机，其中每个不同的 await 代表一个状态。一旦所有状态都完成，函数将继续执行到最后一行并返回结果。

由于语法看起来像阻塞性的、并发的过程，你可能很难理解异步编程的本质和陷阱。在实现第一个应用程序时，我们将更深入地了解这个问题。现在，了解这些成分就够了。下面看一下 future 的内部，了解正在处理的内容，或者所用的运行时正在处理的内容。

2.2.4 使用 Rust Future 类型

在代码清单 2.22 中，可以看到 await 函数返回了一些内容，并将其保存在变量 resp 中。在这里，Future 类型开始发挥作用。如前所述，Future 是一个更复杂的类型，具有以下签名(见代码清单 2.23)。你不必完全了解它的功能，不过代码片段中包含两个链接，以便你深入了解。代码清单 2.23 之后的内容对关键功能进行了解释。

代码清单 2.23 Rust 中的 Future trait

```
// Docs: https://doc.rust-lang.org/std/future/trait.Future.html
// Source code:
// https:/ /doc.rust-lang.org/src/core/future/future.rs.html#36-104

pub trait Future {
     type Output;
     fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<Self::Output>;
}
```

Future 有一个名为 Output 的关联类型，例如，它可以是一个文件或字符串，并且有一个名为 poll 的方法。此方法被频繁调用以确认 future 是否到达 Ready 状态。poll 方法返回一个类型 Poll，它可以是 Pending 或 Ready 状态。一旦到达 Ready 状态，poll 将返回第 2 行中指定的类型或错误。一旦 future 到达 Ready 状态，它将返回一个结果，然后将其赋值给变量。

你可以看到 Future 是 trait。这提供了一个优势——你可以对程序中的任何类型实现这个 trait。

Rust 的不同之处在于 future 不会主动启动。在其他语言(如 Go 或 JavaScript)中，当将

变量赋值给 promise 或创建 go routine 时，它们的每个运行时都将立即开始执行。在 Rust 中，必须主动对 future 使用 poll 方法，这是运行时的工作。

2.2.5　选择运行时

运行时是异步 Web 服务的核心。其设计和性能在很大程度上决定了应用程序的基本性能和安全性。在 Node.js 中，由 Google V8 引擎来处理这个任务。Go 也有自己的运行时，它同样是由 Google 开发的。

你不需要详细了解运行时的原理以及如何执行异步代码。不过，可以尝试了解术语和它关联的概念，这是有益的。你可能会在后面的代码中遇到问题，了解你选择的运行时的工作原理，可以帮助你解决问题或重写代码。

许多人批评 Rust 没有附带运行时，因为运行时在每个 Web 服务中都是一个核心部分。不过从另一个角度看，根据自己的需求选择特定的运行时，其优点是可以根据性能和平台的要求来调整应用程序。

Tokio 是最受欢迎的运行时之一，被广泛应用于整个行业。因此，它是应用程序的首要安全保障。我们将在示例中选择 Tokio，稍后将详细介绍如何根据需要选择运行时。

运行时负责创建线程，轮询各个 future，并驱动它们完成。它还负责将任务传递给内核，并确保使用异步内核 API 以避免出现瓶颈。Tokio 使用一个名为 Mio(https://github.com/tokio-rs/mio)的库来与操作系统内核进行异步通信。作为开发人员，你可能永远不会接触到 Mio 中的代码，但不妨了解一下，在使用 Rust 开发 Web 服务器时，你将哪些类型的库和抽象层加入项目中。

如图 2.11 所示，运行时在 Web 服务中起着相当重要的作用。将代码标记为 async 时，编译器将代码交给运行时。然后，实现决定了执行此任务的速度、准确性和无错误性。

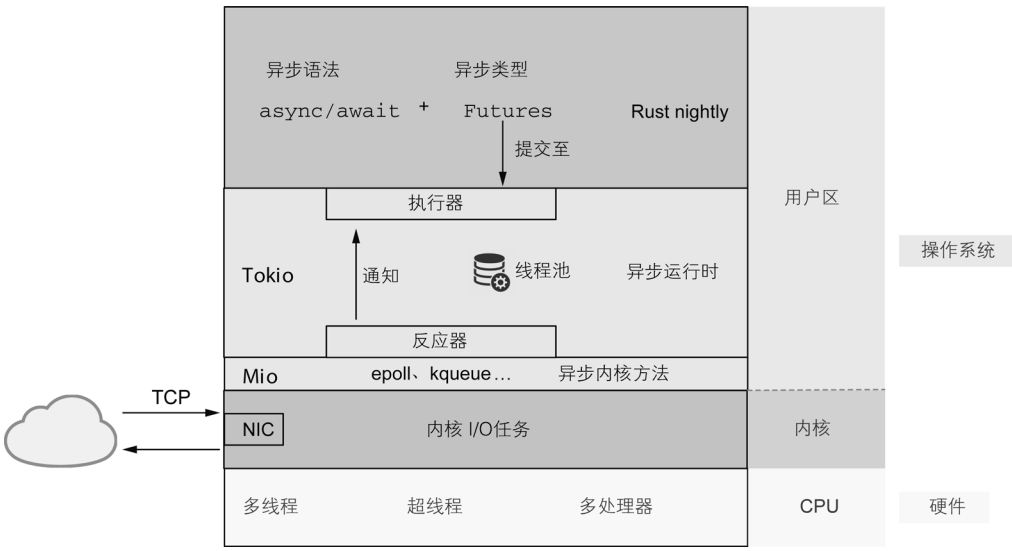


图 2.11　完整的 Rust 异步环境

下面通过一个示例任务来看看幕后发生了什么，见图 2.12。

❶ 在 Rust 代码中，把一个函数标记为 async。当等待函数的返回值时，在编译期间告诉运行时这是一个返回 Future 类型的函数。

❷ 运行时将这段代码交给执行器。执行器负责调用 Future 上的 poll 方法。

❸ 如果是网络请求，运行时将其交给 Mio，Mio 在内核中创建异步套接字，并请求一些 CPU 时间来完成任务。

❹ 一旦内核完成工作(例如，发送请求并获取响应)，它会通知套接字上的等待进程。响应器负责唤醒执行器，执行器继续使用从内核返回的结果进行计算。

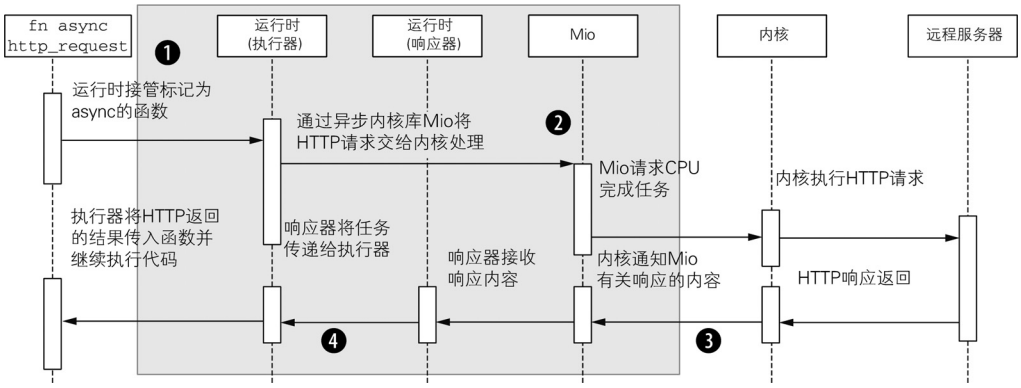


图 2.12 在后台执行异步 HTTP 请求

2.2.6 选择 Web 框架

由于 Rust 在 Web 服务方面仍然是一个新领域，你可能需要开发团队和社区更积极的帮助来解决你在探索过程中可能遇到的问题。

以下是 Rust 提供的四大 Web 框架：

- Actix Web 是最完整、使用最频繁的 Web 框架，具有很多功能；有时可能会有点“偏执”。
- Rocket 使用宏来注解路由函数，并内置了 JSON 解析。它是一个完整的框架，包含了编写稳定的 Web 服务器所需的所有功能。
- Warp 是 Rust 最初的 Web 框架之一。它与 Tokio 社区密切合作，提供了很大的自由发挥空间。它是最基础的框架，将许多设计决策留给了开发人员。
- Axum 是最新的框架之一，试图尽可能构建在已有的 Tokio 生态系统的 crate 之上，并借鉴了 Warp 和其他框架的设计经验。

Actix Web 带有自己的运行时(但你也可以选择使用 Tokio)。Rocket、Warp 和 Axum 框架使用 Tokio。

本书选择了 Warp。它足够小，不碍事，而且被广泛使用，有一个非常活跃的 Discord 频道。你自己的公司或项目中的情况可能会有所不同。重要的是了解框架的来龙去脉，哪些是纯 Rust 代码，以及你的代码在哪里会受到你选择的框架的影响。

书中的大部分内容和代码都与框架无关。一旦配置好服务器并添加了路由函数，你将再次进入纯 Rust 领域，之后就不会看到太多涉及框架的内容了。书中将明确突出这些部分，以便你知道在哪里添加自己选择的框架。

如图 2.13 所示，传入的 TCP 请求必须交给运行时 Tokio，它直接与内核通信。Hyper 库将启动 HTTP 服务器并接收这些传入的 TCP 流。在此基础上，Warp 将包装框架功能，如将 HTTP 请求传递给正确的路由函数。代码清单 2.24 展示了所有这些操作。

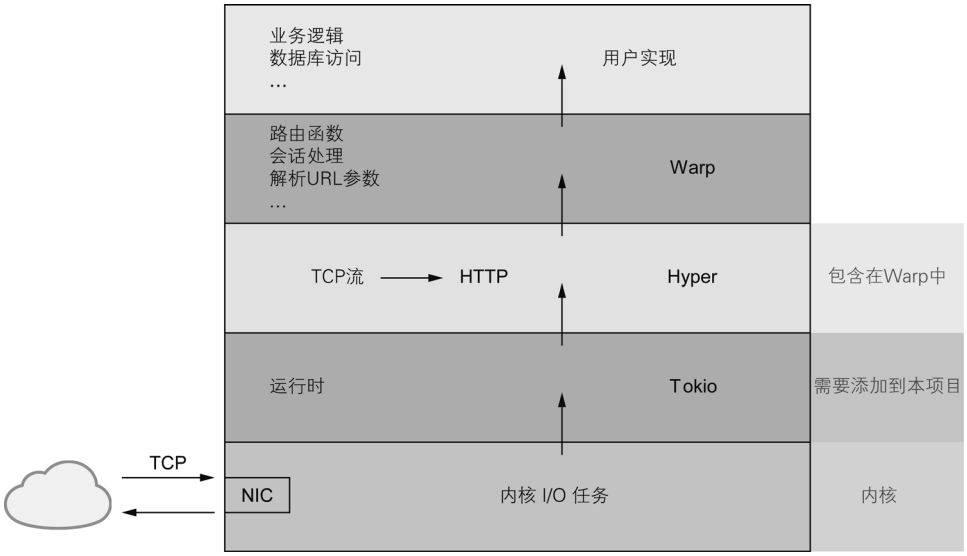


图 2.13　当使用 Warp 时，你将继承运行时 Tokio 和 Hyper 作为底层的 HTTP 抽象和服务器

代码清单 2.24　使用 Warp 的最小可用 Rust HTTP 服务器

```
use warp::Filter;

#[tokio::main]
async fn main() {
    let hello = warp::path("hello")
        .and(warp::path::param())
        .map(|name: String| format!("Hello, {}!", name));

    warp::serve(hello)
        .run(([127, 0, 0, 1], 1337))
        .await;
}
```

.map 函数是一个 Warp 过滤器，它从前面的函数中获取可能的参数并对它们进行转换

(.map(||...))的签名使用了一个闭包(||)，它捕获环境中的变量并使它们在函数(map)内部允许被访问。在代码清单 2.24 中，map 函数内部没有使用任何变量。然而，如果 HTTP GET 请求有任何参数，可通过闭包(||)在 map 内部捕获和处理这些参数。*The Rust Programming Language* 对闭包有更详细的介绍(https://doc.rust-lang.org/book/ch13-01-closures.html)。

为了以一个可用的 Web 服务器结束本章，我们将把这个示例片段放入代码库中。到

目前为止，main 函数如代码清单 2.25 所示。

代码清单 2.25 目前为止的 main 函数

```
fn main() {
     let question = Question::new(
          QuestionId::from_str("1").expect("No id provided"),
          "First Question".to_string(),
          "Content of question".to_string(),
          Some(vec!("faq".to_string())),
     );
     println!("{:?}", question);
}
```

我们移除了创建新 question 的代码(第 3 章将介绍如何返回 JSON)，在项目中添加了运行时和 Warp 服务器，并在 main 函数中启动了服务器；需要将两个依赖项添加到项目中。Hyper crate 包含在 Warp 中，而 Tokio 必须手动添加到项目中。代码清单 2.26 展示了更新后的 Cargo.toml 文件。

代码清单 2.26 更新后的 Cargo.toml 文件(已将 Tokio 和 Warp 添加到项目中)

```
[package]
name = "ch_02"
version = "0.1.0"
edition = "2021"

[dependencies]
tokio = { version = "1.2", features = ["full"] }
warp = "0.3"
```

添加 Tokio 依赖项后，可以在 main 函数上通过注解使用 Tokio 运行时，并在其中编写 Warp 所需的异步代码。代码清单 2.27 展示了 ch_02 文件夹中更新后的 main.rs 文件。

代码清单 2.27 在 main.rs 文件中启动 Warp 服务器

```
use std::str::FromStr;
use std::io::{Error, ErrorKind};

use warp::Filter;

…

#[tokio::main]
async fn main() {
    let question = Question::new(
         QuestionId::from_str("1").expect("No id provided"),
         "First Question".to_string(),
         "Content of question".to_string(),
         Some(vec!("faq".to_string())),
    );
  println!("{:?}", question);
```