

Software Architecture with C++

Design modern systems using effective architecture concepts, design patterns, and techniques with C++20

Adrian Ostrowski | Piotr Gaczkowski



Software Architecture with C++

Design modern systems using effective architecture concepts, design patterns, and techniques with
C++20

(使用有效的架构概念，设计模式和 C++20 进行系统设计)

作者: Adrian Ostrowski, Piotr Gaczkowski

译者: 陈晓伟

本书概述

通过理解诸如微服务、*DevOps* 和使用现代 C++ 标准和特性的本地云等架构，将业务需求应用到 IT 基础设施，并交付高质量的产品。

软件架构是指复杂应用的高级设计。就像语言一样，它也在不断发展，但在不牺牲可读性和可维护性的情况下，可以了解一些架构概念和模式，以及用高级语言编写高性能的应用。

如果正在使用现代 C++，本书会把相应的知识运用到实际工作中，设计分布式的、大规模的应用。首先，快速了解架构概念，包括已建立的模式和正在兴起的趋势，然后继续理解什么是软件架构，并开始研究其组件。

接下来，在了解如何构建、打包、集成和部署组件之前，需要了解应用程序架构中涉及的设计概念和软件开发中的模式。最后一章，将探讨不同的架构质量，例如：可维护性、可重用性、可测试性、性能、可扩展性和安全性。最后，将概述分布式系统，如面向服务的架构、微服务和本地云，并了解如何在应用开发中应用。

本书的最后，将使用现代 C++ 和相关工具构建分布式服务，从而根据客户的需求交付解决方案。

关键特性

- 用 C++ 设计可扩展的大规模应用程序
- 基于云的持续集成和持续交付 (CI/CD) 中的软件架构解决方案
- 通过设计模式、语言特性和工具来实现架构的目标

内容纲要

- 理解如何应用软件架构的原则
- 应用设计模式和最佳实践来满足架构的目标
- 使用最新的 C++ 特性编写优雅、安全、高效的代码
- 构建易于维护和部署的应用
- 探索不同架构的方法，并根据需求进行应用
- 使用容器简化开发和操作
- 了解软件设计和开发中常见问题的解决方法

作者简介

Adrian Ostrowski 是一个现代 C++ 爱好者，对 C++ 的开发和编写的高质量代码都很感兴趣。他在 IT 行业有超过十年的经验，特别是在 C++ 方面有超过 8 年的经验，总是会与他人分享知识。其操刀的项目包括通过光纤网络进行并行计算，以及在商品交易所的交易系统上工作。目前，他是 Intel 和 Habana 机器学习框架的设计师之一。

在业余时间里，他曾与 Piotr 一起推广乐队，并学会了驾驶滑翔机。目前，他喜欢骑自行车、参加音乐活动和浏览一些网络梗。

敬 Agnieszka，感谢她的爱和支持

敬 Mateusz，我的良师益友

敬我的父母，是你们激发了我的好奇心

敬我的朋友们，感谢你们所做的一切

Piotr Gaczkowski 在编程和实践 DevOps 方面有超过 10 年的经验。喜欢为问题构建简单的解决方案，组织文化活动，教授专业人士。Piotr 热衷于将无聊的活动自动化，并利用自己的经验，通过开设课程、撰写有关个人成长和远程工作的文章来分享知识。

他曾在 IT 行业从事全职工作和自由职业，但他真正热爱的是音乐。当技能不能在工作中发挥作用时，就会去在建立社区。

致 Emilia，她在我写这本书的时候容非常包容我；

我的父母鼓励我编程；为我加油鼓劲的智囊团成员；

Hackerspace Trójmiasto，气氛组重要成员；

IOD，提醒我要保持对写作的热爱；

255，检验工作；

以及所有和我一起走过这段旅程的朋友们。爱你们！

审评者介绍

Andrey Gavrilin 是一名高级软件工程师，就职于提供财务管理云解决方案的国际公司。他拥有工程（工业自动化）硕士学位，曾在会计和人力资源、道路数据库、Web 和 Linux 发行开发以及金融科技等不同领域工作。他的兴趣包括数学、电子、嵌入式系统、全栈网站开发、复古游戏和复古编程。

本书相关

- Github 翻译地址：

<https://github.com/xiaoweiChen/Software-Architecture-with-Cpp>

前言

现代 C++可以在不降低可读性和可维护性的情况下，编写高性能应用程序。软件架构不仅仅是语言，本书将展示如何设计和构建健壮、可扩展，且性能良好的应用程序。

我们将从理解体系结构的重要性开始，逐步了解基本概念、示例和具体问题后，再对实际应用的案例进行研究。

将了解在独立应用程序中使用已有的设计模式，并探索可使应用程序更健壮、更安全、高性能和可维护的方式，再使用面向服务的架构、微服务、容器和无服务器技术等模式，对多个独立应用程序进行连接的高级服务。

阅读完本书，将能使用现代 C++和相关工具构建分布式服务，并以客户推荐的解决方案进行交付。

有兴趣成为一名软件架构师，或者了解更多关于架构的趋势吗？如果答案是“是”，那么这本书应该对你有所帮助！

适读人群

为了帮助使用现代 C++的开发者将能够将知识运用到软件架构的实践指南中，本书采取了实践的方法来实现相关方案，会让读者们感觉干货满满。

本书内容

第 1 章，软件架构的重要性和设计原则。首先，为什么要设计软件。

第 2 章，架构风格。介绍了在架构方面可以使用的方法。

第 3 章，功能和非功能需求。探索客户的需求。

第 4 章，架构与系统设计。创建有效的软件解决方案。

第 5 章，C++特性。

第 6 章，设计模式和 C++。重点关注现代 C++的习惯用法和代码构造。

第 7 章，构建和打包。将代码投入产线。

第 8 章，编写可测试的代码。如何在客户之前找到错误。

第 9 章，持续集成和持续部署。自动化软件发布。

第 10 章，代码和部署的安全性。确保系统不被破坏。

第 11 章，性能。了解性能，看 C++有多快——还能更快？

第 12 章，面向服务的架构。基于服务构建系统。

第 13 章，设计微服务。

第 14 章，容器。为构建、打包和运行应用程序提供了统一的接口。

第 15 章，原生云设计。超越传统基础设施，探索原生云上的设计。

编译环境

本书中的代码示例主要是为 GCC 10 编写的，使用 Clang 或 Microsoft Visual C++编译器也没什么问题，不过在旧编译器中可能缺少 C++20 的某些特性。为了获得一个尽可能接近作者的开发环境，我们建议在一个类似 Linux 的环境中使用 Nix (<https://nixos.org/download.html>) 和

direnv (<https://direnv.net/>)。如果在一个包含示例的目录中运行 `direnv allow`, 这两个工具会为你配置编译器和安装相关的依赖包。

没有 Nix 和 direnv 的话, 就不能保证示例将正确地工作。如果使用 macOS, Nix 应该可以正常工作。如果在 Windows 上, Windows 子系统 Linux 2 中一个 Linux 开发环境, 可以与 Nix 一起使用。

要安装这两个工具, 可以运行以下命令:

```
# Install Nix
curl -L https://nixos.org/nix/install | sh
# Configure Nix in the current shell
. $HOME/.nix-profile/etc/profile.d/nix.sh
# Install direnv
nix-env -i direnv
# Download the code examples
git clone
https://github.com/PacktPublishing/Hands-On-Software-Architecture-with-Cpp.
git
# Change directory to the one with examples
cd Hands-On-Software-Architecture-with-Cpp
# Allow direnv and Nix to manage your development environment
direnv allow
```

执行上述命令后, Nix 应该下载并安装所有依赖项。这可能需要一些时间, 但它有助于确保与作者使用的是完全相同的工具。

如果正在使用这本书的数字版本, 建议自行输入代码或通过 GitHub 库访问代码 (链接在下一节中提供)。这样做将避免与复制和粘贴代码时出现的错误

下载示例

可以从 GitHub 网站 <https://github.com/PacktPublishing/Software-Architecture-with-Cpp> 下载本书的示例代码文件。如果代码有更新, 会在现有的 GitHub 库中更新。

我们还有其他的代码包, 还有丰富的书籍和视频目录, 都在 <https://github.com/PacktPublishing/>。去看看吧!

联系方式

我们欢迎读者的反馈。

反馈: 如果你对这本书的任何方面有疑问, 需要在你的信息的主题中提到书名, 并给发邮件到 customercare@packtpub.com。

勘误: 尽管我们谨慎地确保内容的准确性, 但错误还是会发生。如果在本书中发现了错误, 请向我们报告, 将不胜感激。请访问 www.packtpub.com/support/errata, 选择相应书籍, 点击勘

误表提交表单链接，并输入详细信息。

盗版: 如果在互联网上发现任何形式的非法拷贝，非常感谢提供地址或网站名称。请通过 copyright@packt.com 与我们联系，并提供材料链接。

如果对成为书籍作者感兴趣: 如果你对某主题有专长，又想写一本书或为之撰稿，请访问 authors.packtpub.com。

欢迎评论

请留下评论。当阅读书籍，为什么不在购买网站上留下评论呢？其他读者可以看到您的评论，并根据您的意见来做出购买决定。我们在 Packt 可以了解您对我们产品的看法，作者也可以看到您对他们撰写书籍的反馈。谢谢你！

想要了解 Packt 的更多信息，请访问 packt.com。

目录

第一部分：软件架构的概念和组件	18
第 1 章 软件架构的重要性和设计原则	19
1.1. 相关准备	19
1.2. 了解软件架构	19
1.2.1 从不同的角度来看待架构	19
1.3. 正确架构的重要性	20
1.3.1 软件腐烂	20
1.3.2 随意架构	20
1.4. 优秀架构的基础	20
1.4.1 架构的上下文	21
1.4.2 利益相关方	21
1.4.3 业务和技术环境	21
1.5. 使用敏捷原则开发软件架构	21
1.5.1 领域驱动设计	22
1.6. C++的哲学	23
1.7. SOLID 和 DRY 原则	25
1.7.1 单一职责原则	25
1.7.2 开放封闭原则	25
1.7.3 子可替父原则	26
1.7.4 接口隔离原则	27
1.7.5 依赖倒置原则	28
1.7.6 DRY 规则	31
1.8. 耦合和内聚	31
1.8.1 耦合	31
1.8.2 内聚	33
1.9. 总结	34
1.10. 练习题	35
1.11. 扩展阅读	35
第 2 章 架构风格	36
2.1. 相关准备	36
2.2. 选择有状态方法和无状态方法	36

2.2.1 无状态和有状态服务	38
2.3. 理解大应用——避免使用的原因，以及识别异常	38
2.4. 理解服务和微服务	39
2.4.1 微服务	40
2.5. 探索基于事件的架构	42
2.5.1 基于事件的拓扑	42
2.5.2 事件源	43
2.6. 探索分层架构	44
2.6.1 面向前端的后端	46
2.7. 了解模块化架构	47
2.8. 总结	47
2.9. 练习题	47
2.10. 扩展阅读	48
第3章 功能和非功能需求	49
3.1. 相关准备	49
3.2. 需求类型	49
3.2.1 功能需求	49
3.2.2 非功能需求	50
3.3. 认识架构的重要需求	51
3.3.1 架构指标的意义	51
3.3.2 ASR 的识别及处理方法	52
3.4. 收集需求	52
3.4.1 了解背景	52
3.4.2 了解现有文档	53
3.4.3 了解责任相关方	53
3.4.4 从责任相关方处收集需求	53
3.5. 文档化需求	54
3.5.1 记录背景	54
3.5.2 记录范围	55
3.5.3 记录功能性需求	55
3.5.4 记录非功能性需求	56
3.5.5 管理文档版本的记录	56
3.5.6 记录敏捷项目中的需求	56
3.5.7 其他部分	57
3.6. 文档化架构	57
3.6.1 4+1 模型	57
3.6.2 C4 模型	61
3.6.3 记录敏捷项目中的架构	63
3.7. 选择合适的视图进行记录	63

3.7.1 功能视图	64
3.7.2 信息视图	64
3.7.3 并发视图	64
3.7.4 发展视图	65
3.7.5 部署和操作视图	65
3.8. 生成文档	66
3.8.1 生成需求文档	66
3.8.2 用代码中生成图表	66
3.8.3 用代码生成 (API) 文档	67
3.9. 总结	72
3.10. 练习题	72
3.11. 扩展阅读	72
第二部分：C++软件的设计与开发	73
第 4 章 架构与系统设计	74
4.1. 相关准备	74
4.2. 分布式系统的特性	74
4.2.1 不同的服务模型	75
4.2.2 避免分布式计算的错误	77
4.2.3 CAP 定理和最终一致性	79
4.3. 使系统具有容错性和可用性	80
4.3.1 计算系统的可用性	81
4.3.2 构建容错系统	81
4.3.3 检测故障	83
4.3.4 降低故障的影响	84
4.4. 集成系统	86
4.4.1 管道与过滤器模式	86
4.4.2 使用者竞争	86
4.4.3 从遗留系统的过渡	87
4.5. 分级性能	88
4.5.1 CQRS 和事件源	88
4.5.2 缓冲	90
4.6. 部署系统	91
4.6.1 边车设计模式	91
4.6.2 零宕机部署	97
4.6.3 外部配置存储	98
4.7. 管理 API	99
4.6.1 API 网关	99
4.8. 总结	100
4.9. 练习题	100

4.10. 扩展阅读	100
第 5 章 C++特性	101
5.1. 相关准备	101
5.2. 设计优秀的 API	101
5.2.1 使用 RAII	101
5.2.2 整理 C++中容器的接口	102
5.2.3 接口中使用指针	104
5.2.4 确定前置条件和后置条件	105
5.2.5 使用内联名称空间	105
5.2.6 使用 std::optional	106
5.3. 编写声明性代码	107
5.3.1 展示特色商品	109
5.3.2 标准中的 range	113
5.4. 编译时计算	115
5.4.1 使用 const 来帮助编译器	116
5.5. 安全类型	116
5.5.1 约束模板参数	117
5.6. 编写模块化 C++	119
5.7. 总结	121
5.8. 练习题	122
5.9. 扩展阅读	122
第 6 章 设计模式和 C++	123
6.1. 相关准备	123
6.2. C++的惯用法	123
6.2.1 使用 RAII 保护自动化范围的退出操作	123
6.2.2 管理可复制性和可移动性	124
6.2.3 隐藏友元	125
6.2.4 使用复制和交换习惯性用法提供异常安全	126
6.2.5 编写 nieblloid	127
6.2.6 基于策略的设计风格	129
6.3. 模板递归模式	130
6.3.1 动态多态性和静态多态性	130
6.3.2 实现静态多态性	131
6.3.3 使用类型擦除	133
6.4. 创建对象	134
6.4.1 何为工厂	134
6.4.2 使用构建器	138
6.5. C++中跟踪状态和访问对象	141
6.6. 高效地处理内存	144

6.6.1 使用 SSO/SOO 减少动态分配	144
6.6.2 通过 COW 来节省内存	145
6.6.3 使用多态分配器	145
6.7. 总结	148
6.8. 练习题	148
6.9. 扩展阅读	149
第 7 章 构建和打包	150
7.1. 相关准备	150
7.2. 充分利用编译器	150
7.2.1 多编译器	150
7.2.2 减少构建时间	151
7.2.3 寻找代码的潜在问题	153
7.2.4 以编译器为中心的工具	155
7.3. 抽象构建过程	155
7.3.1 CMake	155
7.3.2 生成器表达式	158
7.4. 使用外部模块	159
7.4.1 获取依赖项	159
7.4.2 find 脚本	160
7.4.3 编写 find 脚本	161
7.4.4 使用 Conan 包管理器	164
7.4.5 添加测试	166
7.5. 重用代码质量	167
7.5.1 安装	168
7.5.2 导出	170
7.5.3 CPack	171
7.6. 使用 Conan 进行打包	173
7.6.1 创建 conanfile.py 脚本	173
7.6.2 测试 Conan 包	175
7.6.3 添加 Conan 打包代码到 CMakeLists	176
7.7. 总结	177
7.8. 练习题	177
7.9. 扩展阅读	178
第三部分：体系架构的质量	179
第 8 章 编写可测试的代码	180
8.1. 相关准备	180
8.2. 为什么要写测试代码？	180
8.2.1 测试金字塔	181
8.2.2 非功能性测试	182

8.2.3 回归测试	182
8.2.4 原因分析	182
8.2.5 改进的基础	183
8.3. 测试框架	184
8.3.1 GTest	184
8.3.2 Catch2	184
8.3.3 CppUnit	185
8.3.4 Doctest	186
8.3.5 编译时测试	187
8.4. 测试中的 mock 和 fake	187
8.4.1 不同的测试替身	187
8.4.3 测试替身的其他用法	188
8.4.3 制作测试替身	188
8.5. 测试驱动的类型设计	191
8.5.1 测试和类设计——冲突	191
8.5.3 防御性编程	192
8.5.3 无聊的重复——先写测试	193
8.6. 在持续集成/持续部署上进行自动化测试	193
8.6.1 测试的基础设施	194
8.6.2 Serverspec 的测试	194
8.6.3 Testinfra 的测试	195
8.6.4 Goss 的测试	195
8.7. 总结	196
8.8. 练习题	196
8.9. 扩展阅读	197
第 9 章 持续集成和持续部署	198
9.1. 相关准备	198
9.2. 了解 CI	198
9.2.1 尽早发布，经常发布	198
9.2.2 CI 的优点	199
9.2.3 阀门机制	199
9.2.4 用 GitLab 实现流水	199
9.3. 检查代码更改	201
9.3.1 自动控制机制	201
9.3.2 代码审查——手动控制机制	202
9.3.3 代码评审的不同方法	202
9.3.4 使用拉请求(合并请求)进行代码评审	203
9.4. 探索测试驱动的自动化	203
9.4.1 行为驱动开发	204

9.4.2 为 CI 编写测试	205
9.4.3 持续测试	206
9.5. 管理部署代码	208
9.5.1 使用 Ansible	208
9.5.2 Ansible 如何适应 CI/CD 流水	208
9.5.3 创建代码部署	208
9.6. 构建部署代码	209
9.7. 建立 CD 流水线	209
9.7.1 持续部署和持续交付	210
9.7.2 构建一个 CD 流水示例	210
9.8. 不可变的基础设施	213
9.8.1 不可变的基础设施	213
9.8.2 不可变的优势	213
9.8.3 使用 Packer 构建实例镜像	214
9.8.4 使用 Terraform 编排基础设施	215
9.9. 总结	218
9.10. 练习题	218
9.11. 扩展阅读	218
第 10 章 代码和部署的安全性	220
10.1. 相关准备	220
10.2. 检查代码安全性	220
10.2.1 安全设计	221
10.2.2 安全编码指南和 GSL	223
10.2.3 防御性编码，验证一切	224
10.2.4 最常见的漏洞	225
10.3. 检查依赖关系是否安全	226
10.2.1 公共缺陷检索	226
10.2.2 自动扫描	226
10.2.3 自动依赖升级管理	227
10.4. 加固代码	227
10.4.1 面向安全性的内存分配器	227
10.4.2 自动检查	228
10.4.3 处理隔离和沙盒	230
10.5. 强化环境	231
10.5.1 静态和动态链接	231
10.5.2 随机地址空间分配	232
10.5.3 DevSecOps	232
10.6. 总结	232
10.7. 练习题	232

10.8. 扩展阅读	233
第 11 章 性能	234
11.1. 相关准备	234
11.2. 测定性能	234
11.2.1 进行准确且有意义的测量	234
11.2.2 利用不同类型的测量工具	235
11.2.3 微基准测试	235
11.2.4 分析	243
11.2.5 跟踪	244
11.3. 让编译器生成高性能代码	245
11.3.1 优化整个项目	245
11.3.2 使用模式进行优化	245
11.3.3 缓存友好的代码	246
11.3.3 设计代码和数据	246
11.4. 并行计算	247
11.4.1 理解线程和进程的区别	248
11.4.2 标准的并行算法	248
11.4.3 使用 OpenMP 和 MPI 并行计算	248
11.5. 协程	249
11.5.1 与 cppcoro 的区别	250
11.5.2 剖析可等待程序和协程	251
11.6. 总结	255
11.7. 练习题	255
11.8. 扩展阅读	255
第四部分：原生云的设计原则	256
第 12 章 面向服务的架构	257
12.1. 相关准备	257
12.2. 了解面向服务的架构	257
12.2.1 实现方法	257
12.2.2 面向服务架构的优点	262
12.2.3 SOA 的挑战	262
12.3. 消息传递原则	263
12.3.1 低开销的消息传递系统	263
12.3.2 代理消息传递系统	264
12.4. 使用 Web 服务	265
12.4.1 调试 Web 服务的工具	265
12.4.2 基于 XML 的 Web 服务	265
12.4.3 基于 JSON 的 Web 服务	267
12.4.4 表达性状态转移 (REST)	269

12.4.5 GraphQL	276
12.5. 托管服务和云提供商	277
12.5.1 云计算是 SOA 的扩展	277
12.5.2 原生云架构	281
12.6. 总结	281
12.7. 练习题	281
12.8. 扩展阅读	282
第 13 章 设计微服务	283
13.1. 相关准备	283
13.2. 深入了解微服务	283
13.2.1 微服务的好处	283
13.2.2 微服务的缺点	284
13.2.3 微服务的设计模式	285
13.3. 构建微服务	287
13.3.1 外包的内存管理	288
13.3.2 外包存储	290
13.3.3 外包计算	290
13.4. 观察微服务	290
13.4.1 记录日志	290
13.4.2 监控	294
13.4.3 跟踪	294
13.4.4 集成的可观测性解决方案	295
13.5. 连接微服务	295
13.5.1 应用程序编程接口	296
13.5.2 远程过程调用	296
13.6. 扩展微服务	298
13.6.1 每台主机部署单个服务	298
13.6.2 每台主机部署多个服务	299
13.7. 总结	299
13.8. 练习题	299
13.8. 扩展阅读	299
第 14 章 容器	300
14.1. 相关准备	300
14.2. 引入容器	300
14.2.1 容器类型	301
14.2.2 微服务的兴起	301
14.2.3 何时使用容器	301
14.3. 构建容器	303
14.3.1 容器镜像	303

14.3.2 使用 Dockerfiles 构建应用程序	303
14.3.3 命名和分发镜像	304
14.3.4 已编译的应用和容器	305
14.3.5 使用清单针对多架构	306
14.3.6 构建应用容器的替代方法	308
14.3.7 使用 CMake 集成容器	310
14.4. 测试和集成容器	311
14.4.1 容器内的运行时库	312
14.4.2 选择容器的运行时	312
14.5. 容器协调器	313
14.5.1 自托管解决方案	313
14.5.2 管理服务	319
14.6. 总结	320
14.7. 练习题	320
14.8. 扩展阅读	320
第 15 章 原生云设计	322
15.1. 相关准备	322
15.2. 了解原生云	322
15.2.1 原生云计算基础	322
15.2.2 操作系统——云	323
15.3. 使用 Kubernetes 协调云原生工作负载	324
15.3.1 Kubernetes 的结构	324
15.3.2 部署 Kubernetes 的方法	325
15.3.3 理解 Kubernetes 的概念	326
15.3.4 Kubernetes 网络	327
15.3.5 何时使用 Kubernetes	328
15.4. 分布式系统中的可观测性	328
15.4.1 跟踪与日志记录有何不同	329
15.4.2 选择跟踪解决方案	329
15.4.3 使用 OpenTracing 测试应用程序	330
15.5. 使用服务网格连接服务	332
15.5.1 引入服务网格	332
15.5.2 网格服务解决方案	332
15.6. 使用 GitOps	334
15.6.1 GitOps 的原则	334
15.6.2 GitOps 的好处	335
15.6.3 GitOps 工具	336
15.7. 总结	337
15.8. 练习题	337

15.9. 扩展阅读	337
附录 A	339
设计数据存储	339
应该使用哪种 NoSQL 技术?	339
无服务器架构	340
文化与交流	340
DevOps	340
练习答案	342
第 1 章	342
第 2 章	342
第 3 章	343
第 4 章	344
第 5 章	344
第 6 章	345
第 7 章	345
第 8 章	346
第 9 章	347
第 10 章	347
第 11 章	348
第 12 章	348
第 13 章	349
第 14 章	350
第 15 章	350

第一部分：软件架构的概念和组件

了解软件架构的基础知识，并演示其设计和文档的有效方法。

包括以下几章：

- 第1章，软件架构的重要性和设计原则
- 第2章，架构风格
- 第3章，功能和非功能需求

第 1 章 软件架构的重要性和设计原则

这一章展示了软件架构在软件开发中扮演的角色，在设计 C++ 解决方案的结构时，软件架构是需要关注的重点。我们将在代码层面讨论如何设计高效的易用功能性接口，还会引入一种通过领域驱动的方法。

本章将讨论以下内容：

- 了解软件架构
- 正确架构的重要性
- 优秀架构的基础
- 使用敏捷原则开发软件架构
- C++ 的哲学
- SOLID 和 DRY 原则
- 领域驱动设计
- 耦合和内聚

1.1. 相关准备

要使用本章的代码，需要提前准备一些东西：

- Git 客户端，用于加载代码库。
- 兼容 C++20 的编译器，用来编译代码。代码大多数使用 C++11/14/17 编写，但是对少数主题会是用到试验性的概念。
- GitHub 链接：<https://github.com/PacktPublishing/Software-Architecture-with-Cpp/tree/master/Chapter01>。
- GSL 的 GitHub 链接：<https://github.com/Microsoft/GSL>

1.2. 了解软件架构

先从定义软件架构开始。在创建应用程序、库或软件组件时，需要考虑所编写的元素的表现方式，以及交互方式。换句话说，在设计它们以及与周围环境的关系时，就需要像建设城市一样，要规划更大的蓝图，要从大局出发，不要陷入杂乱无章的状态。小范围内，每一栋建筑看起来都不错，但它们并不能组成一个更大的整体——只是不太适合，这就是所谓的随意架构，这需要避免。请记住，无论是否将自己的想法放入其中，在编写软件时，都是在构建一个架构。

如果想要谨慎地定义解决方案的架构，需要创建什么呢？软件工程学如是说：

系统的软件架构是系统运行所需的组件，包括软件元素、元素间的关系，以及元素的属性。

这样，为了定义一个架构，需要从不同的角度来考虑，而非直接开始写代码。

1.2.1 从不同的角度来看待架构

这里提供几个思考的角度：

- 企业架构涉及整个公司，甚至是集团公司。其采取整体的方法，只关注企业的战略。考虑企业架构时，应该了解公司中的所有系统的行为和相互合作的模式。其对业务和 IT 之间的一致性相当敏感。
- 解决方案架构不像企业架构那样抽象，它介于企业架构和软件架构之间。通常，解决方案架构与特定的系统，以及与周围环境交互的方式有关。解决方案架构师需要想出一种方法来满足特定的业务需求，通常的方法是通过设计整个软件系统或修改现有的系统。
- 软件架构比解决方案架构更加具体，其集中于一个特定的项目，使用特定的技术，以及关注如何与其他项目进行交互。这里，软件架构师感兴趣的是项目组件的内部结构。
- 基础设施架构，关注的是软件使用的基础设施。其定义了部署环境和策略、应用程序如何扩展、故障转移处理、站点可靠性以及其他面向基础设施的方面。

解决方案架构基于软件和基础设施架构，从而满足业务需求。在后续的内容中，将对这两种架构进行讨论，以便之后为小型和大型的架构设计做准备。在开始之前，先提一个问题：为什么架构很重要？

1.3. 正确架构的重要性

实际上，应该这样问：为什么要关心架构重要性？无论是否有意识地去构建，最终都会得到某种类型的架构。如果在几个月，甚至几年的开发之后，仍希望软件保证质量，需要在早期采取一些措施。如果不考虑架构，那么很可能永远不会呈现期望的品质。

因此，为了让产品满足业务需求和性能、可维护性、可扩展性（等），就需要关注其架构，最好尽早这样做。现在，来聊聊优秀架构师不能接受的两件事。

1.3.1 软件腐烂

完成起始的工作，并在脑中有了特定的架构后，还需要持续地控制架构后续的发展，以及是否符合其用户的需求，因为这些需求在软件的开发和生命周期中也可能发生变化。软件腐烂，有时也称为代码腐烂，发生在实现决策与计划架构不匹配的时候。所有这些差异都应视为技术债。

1.3.2 随意架构

如果不能跟踪开发是否遵循所选择的架构，或者没有计划架构的样子，通常会导致随意架构。不管在其他领域应用最佳实践，例如：测试或具有特定的开发文化，随意架构都会发生。

有几个反模式可以用来验证随意架构。代码就像一个大泥球，这是最明显的。通常，如果软件是紧密耦合的，可能是循环依赖，但在一开始并不是这样，这就是一个重要的信号，表明应该更有意识地关注架构的外在表现。

现在，来了解一下架构师必须完成哪些工作，才能交付一个可行的解决方案。

1.4. 优秀架构的基础

区分架构的好与坏非常重要，但也并不是那么容易。识别反模式显得尤为重要，但是对于好的架构来说，必须从软件中获得相应的交付期望，无论是关于功能需求、解决方案的属性，还是处理不同的限制。其中许多方案都可以很容易地从架构的上下文中派生出来。

1.4.1 架构的上下文

上下文是架构师在设计可靠的解决方案时要考虑的因素，包括需求、假设和限制，可以来自利益相关方，也可以来自业务和技术环境。还会影响利益相关方和环境，例如：公司需要涉猎一个新的细分市场。

1.4.2 利益相关方

利益相关方是所有与产品的人，这些人可以是客户、系统用户或管理人员。沟通是每个架构师的关键技能，适当地统筹各方需求，其关键在于要以各方预期的交付方式进行交付，并满足交付各方的期望。

这对利益相关群体来说很重要，所以尽量收集所有相关群体的意见。

客户可能会关心编写和运行软件的成本、交付的功能、生命周期、上市时间，以及解决方案的质量。

系统的用户可以分为两组：用户和管理员。前者通常关心诸如软件的可用性、用户体验和性能的事情。后者更关心的是用户管理、系统配置、安全性、备份和恢复。

最后，对于从事管理工作的群体来说，保持低成本开发十分重要，并且还要实现业务目标，在按照开发进度进行的同时，还要保持产品的质量。

1.4.3 业务和技术环境

架构可能会受到公司业务的影响，可能的原因有上市时间、时间表、组织结构、劳动力使用情况，以及对现有资产的配比。

所谓技术环境，指的是公司中使用的技术，或者出于需要成为解决方案的技术。需要集成的系统，也是技术环境的重要组成部分。软件工程师的技术专长在这里也很重要，架构师做出的技术决策可以影响项目的人员配置，所以初级开发人员与高级开发人员的比例会对项目管理有一定的影响。因此，好的架构应该把这些都考虑进去。

了解了这些，现在来讨论一个有争议的话题。作为架构师，这个话题在日常工作中很可能会遇到。

1.5. 使用敏捷原则开发软件架构

架构和敏捷开发似乎是对立的，并且围绕着这个主题有许多神话。为了以敏捷的方式开发产品，并且还要关注架构，需要遵循一些简单的原则。

迭代和递增构成了敏捷的本质。敏捷架构中，不需要提前进行整体的设计。相反，应该提出一个小而合理的前期设计。最好是把每个决定的原因都记录下来，如果产品愿景发生了变化，架构也可以随之改变。为了支持频繁的发布，前期应该以增量的方式进行更新。以这种方式发展起来的架构称为演化架构。

管理架构并不意味着要保存大量的文档，文档应该只包含必要的内容，这样更容易更新。并且应该是简单的，只涉及系统的相关内容。

还有一种神话，认为架构师是真理的唯一来源和最终的决策者。不过，在敏捷环境中，是团队在做决定。如上所述，利益相关方对决策过程的贡献至关重要——他们的观点决定了解决方案的表

现形式。

架构师应该是开发团队的一部分，他们具有强大的技术专长和多年的经验。还应该在每次迭代之前，参与评估和设计必要的架构调整。

为了让团队保持敏捷，应该想办法高效地工作，并且只做重要的事情。实现这些目标的一个好方法是领域驱动设计。

1.5.1 领域驱动设计

领域驱动设计，简称 DDD，是 Eric Evans 在其同名著作中提出的术语。该术语关注点在于改进业务和工程之间的沟通，并将开发人员的注意力吸引到领域模型上。基于此模型通常会让设计更容易理解，并随着模型的变化而发展。

DDD 与敏捷有什么关系？这里回顾一下敏捷宣言：

个体和互动高于流程和工具
可工作的软件高于详尽的文档
客户合作高于合同谈判
响应变化高于遵循计划

—— 敏捷软件开发宣言

为了做出正确的设计决策，首先要了解领域。为此，需要与其他人进行频繁的交流，并鼓励开发团队缩小与业务人员之间的认知差距。代码中的概念应该以通用语言的实体命名，基本上是商业专家和技术专家的行话中重叠的那部分。要是在这些组中使用彼此理解不同的术语，就可能会造成误解，从而导致业务逻辑实现中的缺陷和 Bug。谨慎地命名并使用双方一致认可的术语，对项目来说是有利的。业务分析师或其他业务领域专家作为团队的组成部分时，可以提供更多的帮助。

如果正在为一个比较大的系统进行建模，因为每个团队实际上都在不同的环境中运作，所以可能很难使所有术语对不同的团队具有相同的含义。DDD 建议使用边界上下文来处理这个问题。如果正在建模，例如：一个电子商务系统，可能希望仅从购物软件上下文的角度考虑这些术语，在仔细观察后，可能会发现，库存、交付和会计团队实际上都有自己的模型和术语。

它们都是电子商务领域的不同子领域。理想情况下，每个上下文都可以映射到相应领域有界的上下文上——系统的一部分，并具有自己独立的词汇表。将解决方案分解为较小的模块时，设置此类上下文的边界也很重要。每个模块都有明确的职责、独立的数据库模式和自己的代码库。为了帮助大型系统中的团队之间进行沟通，可能需要引入一个上下文映射，需要表明来自不同上下文的术语间，如何进行关联：

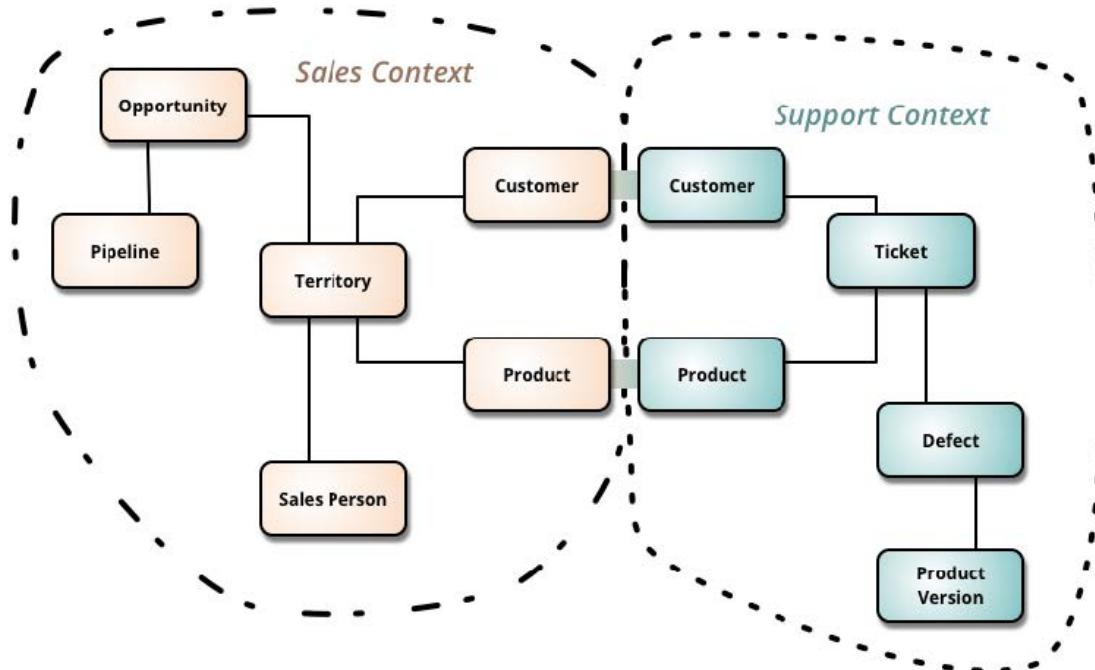


图 1.1 - 两个边界上下文，映射互相匹配的术语 (图片来自 Martin Fowler 关于 DDD 的一篇文章：
<https://martinfowler.com/bliki/BoundedContext.html>)

随着对一些重要的项目管理主题的理解，现在可以转向一些更技术性的主题了。

1.6. C++的哲学

现在，进一步了解在本书中使用最多的编程语言。C++是一种已经存在了几十年的多范式语言。从创立至今，已经发生了很大的变化。当 C++11 问世时，该语言的创造者 Bjarne Stroustrup 说，感觉像是一门全新的语言。C++20 的发布标志着这头巨兽进化的又一个里程碑，也给编码方式带来了一场革命。然而，语言的哲学始终如一，这些年来从未改变。

简而言之，可以总结为三条规则：

- C++中不应该有其他语言（汇编除外）。
- 只为所使的付费。
- 以低成本提供高级抽象（零成本是终极的目标）。

为使用的东西付费意味着，可以在堆栈上创建数据成员（许多语言都在堆上分配对象，但 C++ 却不需要这样做），在堆上分配是有代价的——分配器可能需要锁定互斥量，这在某些类型的应用程序中可能是一个负担。好的方面是，可以方便地分配变量，而不必每次都动态地分配内存。

高级抽象是 C++ 与 C 或汇编等低级语言的区别，其允许在源码中直接表达思想和意图，这对语言的类型安全性非常有利。看下面的代码段：

```

1 struct Duration {
2     int millis_;
3 };
4
5 void example() {

```

```

6 auto d = Duration{};
7 d.millis_ = 100;
8
9 auto timeout = 1; // second
10 d.millis_ = timeout; // ouch, we meant 1000 millis but assigned just 1
11 }

```

更好的做法是利用该语言提供的安全类型:

```

1 #include <chrono>
2
3 using namespace std::literals::chrono_literals;
4
5 struct Duration {
6     std::chrono::milliseconds millis_;
7 };
8
9 void example() {
10     auto d = Duration{};
11     // d.millis_ = 100; // compilation error, as 100 could mean anything
12     d.millis_ = 100ms; // okay
13     auto timeout = 1s; // or std::chrono::seconds(1);
14     d.millis_ =
15         timeout; // okay, converted automatically to milliseconds
16 }

```

前面的抽象可以避免犯错误，并且在这样做的时候不会付出任何代价，生成的汇编与第一个示例相同。这就是 C++ 中为什么有零成本抽象的原因。有时，C++ 允许使用抽象，从而产生比更好的代码。C++20 中的协程就是一个语言特性的例子，在使用时通常会带来这样的好处。

标准库提供的另一组很棒的抽象是算法。认为下列哪一段代码更容易阅读，更容易证明没错？哪个更好地表达意图？

```

1 // Approach #1
2 int count_dots(const char *str, std::size_t len) {
3     int count = 0;
4     for (std::size_t i = 0; i < len; ++i) {
5         if (str[i] == '.') count++;
6     }
7     return count;
8 }
9
10 // Approach #2
11 int count_dots(std::string_view str) {
12     return std::count(std::begin(str), std::end(str), '.');
13 }

```

OK，第二个函数有一个不同的接口，但即使它保持不变，也可以通过指针和长度创建 `std::string_view`。由于它是轻量级的类型，编译器可以对其进行优化。

使用更高层次的抽象可以得到更简单、更易于维护的代码。C++ 从一开始就致力于提供零成本的抽象，所以可以在此基础上进行构建，而不是重新设计轮子。

谈到简单和可维护的代码，下一节将介绍一些在编写此类代码的过程中的规则和方法。

1.7. SOLID 和 DRY 原则

编写代码时要记住许多原则。编写面向对象的代码时，应该熟悉抽象、封装、继承和多态这四个方面。无论是否以面向对象的编程方式编写 C++，都应该牢记两个首字母缩写词：SOLID 和 DRY 背后的原则。

SOLID 是一组可以帮助编写更干净、更少 Bug 软件的实践。它是由五个概念的首字母组成的缩写（译注：中文术语翻译源于 https://blog.csdn.net/louzp_ustc/article/details/83272914）：

- 单一职责原则 (Single responsibility principle)
- 开放封闭原则 (Open-closed principle)
- 子可替父原则 (Liskov substitution principle)
- 接口隔离原则 (Interface Segregation Principle)
- 依赖倒置原则 (Dependency Inversion Principle)

假设读者们已经了解了这些原则与面向对象编程的关系，由于 C++并不总是面向对象的，那么先来看看它是如何应用于不同领域的。

有些例子使用了动态多态，但这同样适用于静态多态。如果正在编写面向性能的代码（如果选择了 C++，很可能就是这样），那么就性能而言，使用动态多态可能不是一个好主意，特别是在热路径上。在其他书中，将会进一步学习如何使用 **奇特重现模板模式 (Curiously Recurring Template Pattern, CRTP)** 编写静态多态类。

1.7.1 单一职责原则

简而言之，单一职责原则 (SRP) 意味着每个代码单元应该只有一个职责。这意味着编写只做一件事的函数，创建负责一件事的类型，以及创建专注一个方面的高级组件。

如果使用类管理某种类型的资源（例如文件句柄），应该只做这一项工作，而将解析工作交给其他类型。

如果看到一个函数的名称中有“And”，那就违反了 SRP，应该进行重构。另一个标志是当函数有注释指出函数的每个部分（原文如此！）是做什么的时候，每个部分拆分为不同的函数可能会更好。

相关主题的信息最小化原则：任何对象都不应该知道其他对象的相关信息，所以它不依赖于其他对象的内部信息。遵循这个原则可以使代码更易于维护，组件之间的依赖更少。

1.7.2 开放封闭原则

开放封闭原则 (OCP) 意味着代码应该对扩展开放，但对修改关闭。开放扩展意味着可以轻松扩展代码支持的类型列表。关闭修改意味着现有代码不应更改，因为这通常会导致系统中其他地方的错误。C++演示这一原理的重要特性是 `operator<<` 的 `ostream`。要扩展它，使它支持自定义类，就需要编写（类似）以下代码：

```
1 std::ostream &operator<<(std::ostream &stream, const MyPair<int, int>
2 &mp) {
```

```
3     stream << mp.firstMember() << ", ";
4     stream << mp.secondMember();
5     return stream;
6 }
```

注意这里 `operator<<` 的实现是一个自由(非成员)函数。可能的话,应该选择自由函数,而不是成员函数,因为这有助于封装。有关这方面的细节,请参阅本章末尾的扩展阅读部分 Scott Meyers 的文章。如果想使用 `ostream` 打印一些非公共访问的成员,可以将相应的 `operator<<` 函数设置为友元函数,像这样:

```
1 class MyPair {
2     // ...
3     friend std::ostream &operator<<(std::ostream &stream,
4         const MyPair &mp);
5 };
6
7 std::ostream &operator<<(std::ostream &stream, const MyPair &mp) {
8     stream << mp.first_ << ", ";
9     stream << mp.second_ << ", ";
10    stream << mp.secretThirdMember_;
11    return stream;
12 }
```

注意,OCP 的这个定义与与多态相关的常见定义略有不同。后者是关于创建基类的,这些基类本身不能修改,但可以继承。

说到多态,就继续下一个原则,因为它是关于正确使用多态的。

1.7.3 子可替父原则

子可替父原则(LSP) 规定,如果函数使用指向基对象的指针或引用,那么必须使用指向其派生对象的指针或引用。这一规则有时会被打破,在源代码中应用的技术,在实际抽象中并不总是有效。

一个著名的例子是正方形和矩形。从数学角度讲,前者是后者的特化,所以两者之间是“is a”的关系。那就可以创建一个 `Square` 类,继承于 `Rectangle` 类。因此,有如下代码:

```
1 class Rectangle {
2 public:
3     virtual ~Rectangle() = default;
4     virtual double area() { return width_ * height_; }
5     virtual void setWidth(double width) { width_ = width; }
6     virtual void setHeight(double height) { height_ = height; }
7 private:
8     double width_;
9     double height_;
10 };
11
12 class Square : public Rectangle {
13 public:
14     double area() override;
```

```
15 void setWidth(double width) override;
16 void setHeight(double height) override;
17 };
```

应该如何实现 `Square` 类的成员？如果想要遵循 LSP，并避免让这些类的使用者感到意外。如果调用 `setWidth`，正方形将不再是正方形。所以，这里只能停止使用正方形（使用前面的代码是无法表达的），或修改高度，从而使正方形看起来不同于矩形。

如果代码违反了 LSP，很可能使用了错误的抽象。例子中，`Square` 不应该继承于 `Rectangle`。更好的方法是让两者分别对 `GeometricFigure` 接口进行实现。

既然要讨论接口的话题，就继续下一个原则吧。

1.7.4 接口隔离原则

接口隔离原则顾名思义。公式如下：

用户需要自主选择想要使用的方法。

这听起来很简单，但内涵并不是简单。首先，比起单一的大界面，应该更倾向于小界面。其次，当需要添加派生类或扩展现有类的功能时，需要在扩展类实现的接口之前进行考虑。

用一个违背这一原则的例子来说明这一点：

```
1 class IFoodProcessor {
2 public:
3     virtual ~IFoodProcessor() = default;
4     virtual void blend() = 0;
5 };
```

可以用一个简单的类来实现它：

```
1 class Blender : public IFoodProcessor {
2 public:
3     void blend() override;
4 };
```

到目前为止还不错。现在，假设要建立另一个更先进的食品处理器模型，所以需要在界面中添加更多方法：

```
1 class IFoodProcessor {
2 public:
3     virtual ~IFoodProcessor() = default;
4     virtual void blend() = 0;
5     virtual void slice() = 0;
6     virtual void dice() = 0;
7 };
8
9 class AnotherFoodProcessor : public IFoodProcessor {
10 public:
11     void blend() override;
12     void slice() override;
13     void dice() override;
14 };
```

现在 `Blender` 类有一个问题，因为它不支持这个新接口——没有合适的方法来实现。可以尝试扩展一个工作区，或者抛出 `std::logic_error`，但更好的解决方案是将界面分成两个，每个都有单独的职责：

```
1 class IBlender {
2 public:
3     virtual ~IBlender() = default;
4     virtual void blend() = 0;
5 };
6
7 class ICutter {
8 public:
9     virtual ~ICutter() = default;
10    virtual void slice() = 0;
11    virtual void dice() = 0;
12};
```

现在 `AnotherFoodProcessor` 可以实现这两个接口，并且不需要更改现有食品处理器的实现。还剩下最后一个 SOLID 原则。

1.7.5 依赖倒置原则

依赖倒置是对解耦有用的原则，说明高级模块不应该依赖于低级模块。相反，两者都应该依赖于抽象。

C++允许两种方法来倒置类之间的依赖关系。第一种是多态，第二种是模板。

假设正在为一个软件开发项目建模，该项目有前端和后端的开发人员。一个简单的写法：

```
1 class FrontEndDeveloper {
2 public:
3     void developFrontEnd();
4 };
5
6 class BackEndDeveloper {
7 public:
8     void developBackEnd();
9 };
10
11 class Project {
12 public:
13     void deliver() {
14         fed_.developFrontEnd();
15         bed_.developBackEnd();
16     }
17 private:
18     FrontEndDeveloper fed_;
19     BackEndDeveloper bed_;
20};
```

每个开发人员都是由 Project 类构造的。但这种方法并不理想，因为现在的高级概念 Project 依赖于较低级的概念——单个开发人员的模块。如何使用多态应用依赖倒置来改变这一点？可以这样定义开发人员，并依赖于一个接口：

```
1 class Developer {
2 public:
3     virtual ~Developer() = default;
4     virtual void develop() = 0;
5 };
6
7 class FrontEndDeveloper : public Developer {
8 public:
9     void develop() override { developFrontEnd(); }
10 private:
11     void developFrontEnd();
12 };
13
14 class BackEndDeveloper : public Developer {
15 public:
16     void develop() override { developBackEnd(); }
17 private:
18     void developBackEnd();
19 };
```

现在，Project 类不再需要知道开发人员的开发过程实现。因此，必须接受它们作为构造函数的参数：

```
1 class Project {
2 public:
3     using Developers = std::vector<std::unique_ptr<Developer>>;
4     explicit Project(Developers developers)
5         : developers_{std::move(developers)} {}
6
7     void deliver() {
8         for (auto &developer : developers_) {
9             developer->develop();
10        }
11    }
12
13 private:
14     Developers developers_;
15 };
```

这种方法中，Project 与具体的实现解耦，而只依赖于名为 Developer 的多态接口，“低级”的具体类也依赖于这个接口。这可以缩短构建时间，并且更容易进行单元测试——现在可以在测试代码中将 mock 作为参数进行传递。

因为我们要处理内存分配，而动态调度本身也有开销，所以在虚拟调度中使用依赖倒置是有代价的。有时 C++ 编译器可以检测到一个给定接口只使用了一个实现，并通过执行反虚拟化来消除开销（通常需要将函数标记为 final）。然而，这里使用了两种实现，因此必须支付动态调度的成本

(通常实现通过虚函数表(或简称 vtables)进行跳转)。

还有一种没有上述缺点的倒置依赖方法。来看看如何使用可变模板(C++14 的泛型 Lambda)和 variant(C++17 或第三方库,如 Abseil 或 Boost)来进行实现。首先是开发者类:

```
1 class FrontEndDeveloper {
2 public:
3     void develop() { developFrontEnd(); }
4 private:
5     void developFrontEnd();
6 };
7
8 class BackEndDeveloper {
9 public:
10    void develop() { developBackEnd(); }
11 private:
12    void developBackEnd();
13 };
```

现在不再依赖于接口,所以不会进行虚拟分派。Project 类仍然接受 Developers 组:

```
1 template <typename... Devs>
2 class Project {
3 public:
4     using Developers = std::vector<std::variant<Devs...>>;
5
6     explicit Project(Developers developers)
7         : developers_{std::move(developers)} {}
8
9     void deliver() {
10         for (auto &developer : developers_) {
11             std::visit([](auto &dev) { dev.develop(); }, developer);
12         }
13     }
14
15 private:
16     Developers developers_;
17 };
```

如果不熟悉 variant,就当它是一个可以保存作为模板参数传递的类即可。因为使用的是可变参数模板,所以可以传递任何类型。要调用存储在变量中的对象的函数,可以使用 std::get 提取,或者使用 std::visit 和可调用对象——在例子中,是 Lambda。它展示了 duck-typing 在实践中的样子。因为所有的开发者类都实现了 develop 函数,所以代码将编译并运行。如果开发者类有不同的方法,可以创建一个函数对象,为不同的类型重载 operator() 就好。

因为 Project 现在是一个模板,必须在每次创建它时指定类型列表,或者提供一个类型别名。可以这样使用:

```
1 using MyProject = Project<FrontEndDeveloper, BackEndDeveloper>;
2 auto alice = FrontEndDeveloper{};
3 auto bob = BackEndDeveloper{};
4 auto new_project = MyProject{{alice, bob}};
```

```
5 new_project.deliver();
```

这种方法不会为每个开发者分配单独的内存或使用虚拟表。然而，在某些情况下，当声明了变量，就不能向其添加其他类型，所以这种方法降低了类的可扩展性。

关于依赖倒置的最后一点是，要注意有一个类似的概念，叫做依赖注入，已经在例子中使用了。它是通过构造函数或 setter 注入依赖关系的，这有利于代码的可测试性（例如，考虑注入模拟对象）。甚至还有在整个应用程序中注入依赖的完整框架，比如 Boost.DI。这两个概念是相关的，经常一起使用。

1.7.6 DRY 规则

DRY 是“不要重复自己”的缩写，应该在可能的情况下避免代码段重复和重用。当代码多次重复类似的操作时，应该提取为函数或函数模板。此外，应该考虑编写一个模板，而不是创建几个类似的类。

同样重要的是，在非必要时不要做重复工作，也就是不要重复别人完成的工作。现在有许多编写良好的成熟库可以帮助我们更快地编写高质量的软件。这里需要特别提到是：

- Boost C++ 库 (<https://www.boost.org/>)
- Facebook 的 Folly (<https://github.com/facebook/folly>)
- Electronic Art 的 EASTL (<https://github.com/electronicarts/EASTL>)
- Bloomberg 的 BDE (<https://github.com/bloomberg/bde>)
- Google 的 Abseil (<https://abseil.io/>)
- 超棒的 Cpp 名单 (<https://github.com/fffaraz/awesome-cpp>) 有几十个之多

有时复制代码也有好处，其中的一个方案就是开发微服务。在单个微服务中遵循 DRY 也是一个好主意，但在多个服务中使用的代码违反 DRY 规则实际上也是值得的。无论讨论的是模型实体还是逻辑，当允许代码重复时，多个服务的维护都会变得更容易。

试想，多个微服务重用相同代码。突然，其中一个需要修改一个字段。所有其他服务现在也必须修改，对于公共代码的依赖关系也是如此。如果因与微服务无关的更改，而要修改几十个或更多的微服务，那么复制代码通常更容易进行维护。

由于讨论的是依赖关系和维护，所以下一节会讨论与此密切相关的话题。

1.8. 緊耦合和內聚

耦合和内聚是软件的两个术语。来看看它们的含义，以及它们之间的关系。

1.8.1 耦合

耦合是软件单元对其他单元依赖的程度。具有高耦合的单元依赖于许多其他单元，所以耦合度越低越好。

如果一个类依赖于另一个类的私有成员，则它们是紧密耦合的。第二个类的变化意味着第一个类也需要改变，这就是为什么高耦合不是一个理想的状态。

为了弱化耦合，可以考虑为成员函数添加参数，而不是直接访问其他类的私有成员。

紧密耦合的另一个例子是依赖倒置中的 Project 和 developer 类的第一个实现。如果添加另一个开发者类型会发生什么：

```
1 class MiddlewareDeveloper {
2 public:
3     void developMiddleware() {}
4 };
5
6 class Project {
7 public:
8     void deliver() {
9         fed_.developFrontEnd();
10    med_.developMiddleware();
11    bed_.developBackEnd();
12 }
13
14 private:
15    FrontEndDeveloper fed_;
16    MiddlewareDeveloper med_;
17    BackEndDeveloper bed_;
18 };
```

看起来不仅是添加了 MiddlewareDeveloper 类，还必须修改 Project 类的公共接口。所以它们是紧密耦合的，并且 Project 类的实现破坏了 OCP。再来看看如何使用依赖倒置将相同的修改应用于实现的：

```
1 class MiddlewareDeveloper {
2 public:
3     void develop() { developMiddleware(); }
4
5 private:
6     void developMiddleware();
7 };
```

不需要更改 Project 类，所以现在这些类是松散耦合的，所需要做的就是添加 MiddlewareDeveloper 类。以这种方式构造代码就能保证更小范围的重构、更快的开发和更容易的测试，所有这些都需要更少的代码，并且更容易维护。使用的新类，只需要修改调用代码：

```
1 using MyProject = Project<FrontEndDeveloper, MiddlewareDeveloper, BackEndDeveloper>;
2 auto alice = FrontEndDeveloper{};
3 auto bob = BackEndDeveloper{};
4 auto charlie = MiddlewareDeveloper{};
5 auto new_project = MyProject{{alice, charlie, bob}};
6 new_project.deliver();
```

这显示了类级别上的耦合。两个服务之间，可以通过引入消息队列等技术来实现低耦合。这样，服务就不会直接相互依赖，并且只依赖于消息格式。如果有一个微服务架构，常见的错误是让多个服务使用相同的数据库。因为无法修改数据库模式，从而不影响使用微服务，进而导致了服务之间的耦合。

1.8.2 内聚

内聚是软件单元之间的关联度。在高度内聚的系统中，同一模块中组件所提供的功能是紧密相关的，这感觉这些组件是一个整体。

在类级别上，一个方法操作的字段越多，它与类的内聚性就越强。最常见的低内聚数据类型，是那些大型单数据类型。当类中有太多内容时，就会缺乏内聚性，也会破坏 SRP。这使得这些类很难维护，并且容易出现 Bug。

较小的类也可能不具有内聚性，考虑下面的示例。这可能看起来没什么，但真实的场景中，通常是数百甚至数千行：

```
1 class CachingProcessor {
2 public:
3     Result process(WorkItem work);
4     Results processBatch(WorkBatch batch);
5     void addListener(const Listener &listener);
6     void removeListener(const Listener &listener);
7
8 private:
9     void addToCache(const WorkItem &work, const Result &result);
10    void findInCache(const WorkItem &work);
11    void limitCacheSize(std::size_t size);
12    void notifyListeners(const Result &result);
13    // ...
14};
```

可以看到处理器实际上执行三种类型的工作：实际工作、缓存结果和管理监听器。在这种情况下，增加内聚的常见方法是提取一个或多个类：

```
1 class WorkResultsCache {
2 public:
3     void addToCache(const WorkItem &work, const Result &result);
4     void findInCache(const WorkItem &work);
5     void limitCacheSize(std::size_t size);
6 private:
7     // ...
8 };
9
10 class ResultNotifier {
11 public:
12     void addListener(const Listener &listener);
13     void removeListener(const Listener &listener);
14     void notify(const Result &result);
15 private:
16     // ...
17 };
18
19 class CachingProcessor {
20 public:
21     explicit CachingProcessor(ResultNotifier &notifier);
```

```

22     Result process(WorkItem work);
23     Results processBatch(WorkBatch batch);
24 private:
25     WorkResultsCache cache_;
26     ResultNotifier notifier_;
27     // ...
28 };

```

现在，每个部分都是由独立的、内聚的实体完成的。重用它们没什么问题，也不会有太多麻烦。甚至将它们变成模板类也不需要太多的工作，测试这样的类也会更容易。

将其置于组件或系统级别上会非常简单——设计的每个组件、服务和系统都应该简洁，专注于做一件事，并且把它做好：

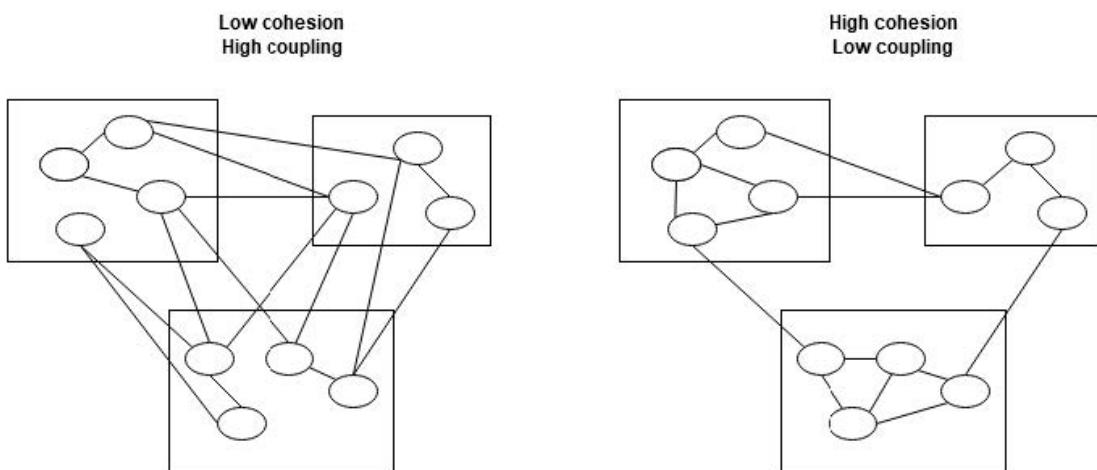


图 1.2 - 耦合与内聚

低内聚和高耦合通常与难以测试、重用、维护甚至理解的软件相应的关联性，因此它缺乏软件中需要的质量属性。

这两个术语经常同时出现，通常一个特征影响另一个特征，不管是功能、类、库、服务，甚至是整个系统。通常情况下，单体服务是高度耦合和低内聚的，而分布式服务则处于另一个极端。

现在让我们总结一下所了解到的知识。

1.9. 总结

本章中，讨论了什么是软件架构，以及为什么值得关注。展示了当架构没有随着需求和实现的变化而更新时会发生什么，以及如何在敏捷环境中看待架构。然后，将注意力转向 C++语言的一些核心原则。

因为 C++不仅编写面向对象的代码，还可以编写非面向对象代码，所以许多来自软件开发的术语在 C++中可以有不同的理解。最后，讨论了耦合和内聚等术语。

现在，应该能够在代码审查中指出许多设计缺陷，并重构的解决方案，以获得更高的可维护性，同时作为开发人员，更不容易出现 Bug。并且，可以设计健壮、自解释和完整的类接口。

下一章中，将了解不同的架构方法或风格，以及何时使用它们来获得更好的结果。

1.10. 练习题

- 为什么要关心软件架构？
- 架构师应该成为敏捷团队的最终决策者吗？
- SRP 与内聚有什么关系？
- 项目生命周期的哪些阶段？架构师可以使项目更好？
- 遵循 SRP 有什么好处？

1.11. 扩展阅读

1. Eric Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software
2. Scott Meyers, How Non-member Functions Improve Encapsulation, <https://www.drdobbs.com/cpp/how-non-member-functions-improve-encapsu/184401197>

第 2 章 架构风格

本章介绍了不同的架构方法或风格。每一节都会讨论软件设计中的不同方法，及其优缺点，并了解如何应用这种方法获利。我们将从比较有状态和无状态架构开始。然后进入系统内部，通过各种类型的面向服务的设计，直到微服务。再通过了解基于事件的系统、分层系统，以及最后的模块化设计，从不同的角度来了解架构风格。

本章将讨论以下内容：

- 选择有状态方法和无状态方法
- 理解大应用——避免使用的原因，以及识别异常
- 理解服务和微服务
- 探索基于事件的架构
- 理解分层架构
- 了解模块化架构

2.1. 相关准备

需要了解什么是软件服务，并且能够阅读 C++11 的代码。

本章的代码可以在以下 GitHub 页面找到：<https://github.com/PacktPublishing/Software-Architecture-with-Cpp/tree/master/Chapter02>。

2.2. 选择有状态方法和无状态方法

有状态和无状态是两种完全相反的软件编写方法，每种方法都有各自的优缺点。

顾名思义，有状态软件的行为取决于其内部状态，这里以 Web 服务为例。如果记住了自己的状态，服务的使用者就可以在每个请求中发送更少的数据。然而，在请求大小和带宽上节省的成本，可能会让 Web 服务端具有隐藏的成本开销。如果用户同时发送许多请求，服务必须进行同步。因为多个请求可能同时改变状态，所以不同步可能会导致数据竞争。

若服务是无状态的，那么每个请求需要包含所需的全部数据。这意味着请求将变得更大，占用更多带宽。但另一方面，将获得更好的性能和服务的扩展性。如果熟悉函数式编程，会发现无状态服务很简洁。处理每个请求可以理解为对纯函数的调用，无状态编程提供的许多优势都源于函数式编程。可变状态是并发的敌人，函数式编程依赖于不可变值，这意味着需要复制对象，而不是修改现有对象。因此，每个线程都可以独立工作，不存在数据竞争。

没有竞争条件，所以不需要锁，这对性能来说是一个巨大的利好，没有锁就不需要处理死锁。纯函数代码也更容易调试，反过来对编译器也有帮助，优化代码也成了一项简单的任务。以函数式编写代码的另一个好处是，编写的源代码往往更简洁和具有表达力，特别是与严重依赖于四人帮（GoF）设计模式的代码相比。

但并不是说带宽没有问题，就应该使用无状态服务。这些决策可以在许多层次上进行，从单个类或函数到整个应用程序。

比如类，如果正在建模，创建一个 *Consultant*，那么将包含诸如顾问的姓名、联系数据、每小时的费率、当前和过去的项目等字段。这是有意义的，它有状态是很自然的事。现在，需要计算其

工作所得，还要创建一个 *PaymentCalculator* 类吗？这时，应该添加一个成员，还是一个自由函数来进行计算？如果使用类，应该传递一个 *Consultant* 作为构造函数参数，还是方法参数？这个类应该有津贴之类的属性吗？

添加一个成员函数来计算薪酬将打破单一职责原则（SRP），因为这个类将有两个职责：计算薪酬和存储顾问的数据（状态）。所以，应该为此引入一个自由函数或单独的类，而不是使用混合类。

首先，在这样的类中必须有这样的状态吗？讨论一下几种不同的 *PaymentCalculator* 类。

一种是公开计算所需的属性：

```
1 class PaymentCalculator;
2 {
3     public:
4         double calculate() const;
5
6         void setHours(double hours);
7         void setHourlyRate(double rate);
8         void setTaxPercentage(double tax);
9
10    private:
11        double hours_;
12        double netHourlyRate_;
13        double taxPercentage_;
14 }
```

这种方法有两个缺点。第一，线程不安全。这样的 *PaymentCalculator* 类的单例不能在没有锁的多线程中使用。第二，当计算复杂时，这个类可能会开始从 *Consultant* 类中复制更多的字段。

为了消除重复，可以重新编写类来存储顾问实例，如下所示：

```
1 class PaymentCalculator {
2 public:
3     double calculate() const;
4
5     void setConsultant(const Consultant &c);
6     void setTaxPercentage(double tax);
7
8     private:
9         gsl::not_null<const Consultant *> consultant_;
10        double taxPercentage_;
11 }
```

注意，因为不能轻易地重新绑定引用，所以使用了来自[指南支持库（GSL）](#)的辅助类在包装器中存储一个可重新绑定的指针，从而确保没有存储空值。

这种方法仍然有线程不安全的缺点。这里能做得更好吗？事实证明，可以通过使类无状态，保证线程安全：

```
1 class PaymentCalculator {
2 public:
3     static double calculate(const Consultant &c, double taxPercentage);
4 }
```

如果没有要管理的状态，那么可以创建自由函数（可能在不同的命名空间中），或作为类的静态函数（就像前面的代码段中的那样），但这都不那么重要。就类而言，区分值（实体）类型和操作类型很有用，因为混合使用可能会违反 SRP。

2.2.1 无状态和有状态服务

在类中讨论的相同原则可以映射到更高级的概念，例如：微服务。

有状态服务是什么样子的？以 FTP 为例，若不是匿名的，则需要用户发送用户名和密码来创建会话。服务器存储这些数据，以识别用户仍然处于连接状态，因此不断地存储状态。每当用户更改工作目录时，状态就会更新。用户所做的每个更改都反映为状态的更改，即使断开连接也是如此。使用有状态服务意味着，根据不同的状态，可以为两个看起来相同的 *GET* 请求返回不同的结果。如果服务器失去状态，请求甚至会停止处理。

有状态服务还可能存在不完整的会话或未完成的事务，以及增加复杂性的问题。会议应持续多久？如何验证客户端是否崩溃或断开连接？应该何时回滚所做的更改？虽然可以找到这些问题的答案，但通常会以动态的、智能的方式与其进行通信。因为服务将维护某种状态，所以让一个服务去维护状态不仅没有必要，而且也是一种浪费。

无状态服务，如本书后面描述的 *REST*，采用了相反的方法。每个请求必须包含所需的全部数据，因此两个相同的幂等请求（例如 *GET*）将有相同的响应。这是在假定存储在服务器上的数据没有更改的情况下，但是数据不一定与状态相同。重要的是每个请求都是自包含的。

现代互联网服务的本质也是无状态。HTTP 协议是无状态的，而许多服务 API（例如 Twitter 的 API）也是无状态的。Twitter 的 API 所依赖的 REST 设计即为功能无状态的。这个缩写词背后的概念为表征状态转移（**Representational State Transfer, REST**），包含了处理请求所需的状态必须传输的概念。如果不这样，就不能拥有 REST 的服务。然而，由于实际需要，这也有一些例外。

如果正在构建一个在线商店，可能希望存储与客户相关的信息，例如：订单历史和送货地址。客户端的客户端可能存储身份验证 cookie，而服务器可能将一些用户数据存储在数据库中。cookie 代替了管理会话的需要，因为这是在有状态服务中完成的。

对于服务来说，将会话保留在服务器端是一种糟糕的方式，原因有以下几点：增加了许多本可以避免的复杂性，使 Bug 更难复现，更为重要的是，无法进行扩展。如果想要将负载分配到另一个服务器，那么可能在复制会话和负载，以及在服务器之间同步会话时遇到困难。所以，会话信息都应该都保存在客户端。

如果希望有一个有状态的架构，就需要这样做的理由。以 FTP 协议为例，必须在客户端和服务器端复制更改。为了进行数据传输，用户只对单个特定的服务器进行身份验证。与 Dropbox 这样的服务相比，数据通常在用户之间共享，文件访问是通过 API 抽象出来的。接下来了解一下，为什么无状态模型更适合这种情况。

2.3. 理解大应用——避免使用的原因，以及识别异常

用于开发的最简单架构风格是整体架构，这就是为什么许多项目开始使用这种风格的原因。相应的应用程序是一个大块，这意味着应用程序中可区分的功能（如处理 I/O、数据处理和用户界面）

是相互交错的，而不是位于架构组件中。这种架构风格的例子是 Linux 内核，内核是整体的并不妨碍它进行模块化。

与部署多组件相比，因为只需要部署一个组件，所以部署单组件可能更容易。并且单个组件更容易测试，因为端到端测试只需要启动单个组件。集成也更容易，在扩展解决方案的同时，可以在负载均衡器后添加更多的实例。有这么多优点，为什么还会有人害怕这种架构风格呢？事实证明，其缺点也很多。

所提供的扩展性从理论上看起来不错，但如果模块有不同的资源需求该怎么办？如果只需要扩展应用程序中的一个模块呢？缺乏模块化（整体系统的固有属性）是此架构许多缺陷的根源。

而且，开发单应用程序的时间越长，维护时遇到的问题就越多。因为在模块之间添加另一个依赖关系非常容易，所以维持这种内部松散的耦合着实是一个挑战。随着应用程序的成长，会变得越来越难以理解。由于增加了复杂性，开发过程很可能会随着时间的推移而变慢。在进行整体开发时，维护设计驱动的开发（**DDD, Design-Driven Development**）的上下文边界也很难定界定。

大应用在部署和执行方面也有缺点，启动应用所需的时间要比启动更多、更小的服务的时间长得多。无论在应用中做了怎样的更改，可能都不想因其重启而重新部署整个应用。现在，假设一个开发人员在应用中引入了内存泄漏。如果漏洞代码反复执行，不仅会影响应用程序的功能，还会影响应用的其他部分。

如果喜欢在项目中使用前沿技术，那么整体化的风格就不那么搭了。由于现在需要一次性迁移整个应用，因此更新库或框架都十分困难。

整体架构只适合于简单和小型的应用程序。如果关心性能，那么与微服务相比，单应用有时可以获得更多的延迟或吞吐量。进程间通信总是会产生一些开销，而整体应用不需要支付这些开销。如果对测试结果感兴趣，请参阅本章扩展阅读部分中的文章。

2.4. 理解服务和微服务

由于整体架构的缺点，应运而生了其他方法。一个常见的想法是将解决方案拆分为多个服务，这些服务相互通信。然后，可以将开发工作分配给不同的团队，每个团队负责单独的服务。每个团队的工作范围都很明确，这与整体架构风格完全不同。

面向服务架构，或简称为 **SOA**，其业务功能是模块化的，并作为单独的服务提供给消费者应用使用。每个服务都应自描述的接口，并隐藏实现细节，比如：内部架构、技术或编程语言。这允许多个团队以其感觉舒服的方式开发服务，所以在内部，每个团队都可以使用合适的服务。假如有两个开发团队，一个精通 C#，另一个精通 C++，他们可以开发两个可以相互通信的服务。

SOA 的倡导者提出了一个宣言，优先考虑以下几点：

- 业务价值 高于 技术策略
- 战略目标 高于 项目效益
- 内在操作性 高于 定制化集成
- 共享服务 高于 目标实现
- 灵活 高于 优化
- 迭代式演进 高于 开始即完美

尽管此宣言没有绑定到技术栈、实现或服务类型，但最常见的两种服务类型是 SOAP 和 REST。

除了这些，还有第三种越来越受欢迎的类型——基于 gRPC 的。可以在关于面向服务架构和微服务的章节中找到更多信息。

2.4.1 微服务

微服务是一种软件开发模式，在这种模式中，应用划分为使用轻量级协议进行通信的松耦合服务集合。微服务模式类似于 UNIX 哲学，即一个程序应该只有一个用途。根据 UNIX 哲学，高级问题可以将这些程序组合到 UNIX 流水线上来解决。类似地，基于微服务的系统由许多微服务和支持服务组成。

先概述一下这种架构风格的优缺点。

微服务的利弊

微服务架构中服务的规模较小，它们的开发、部署和理解速度更快。由于服务是相互独立构建的，因此编译新版本所需的时间不需要很久。由于这一点，在处理这种架构风格时，使用快速原型和开发会更容易。这使得缩短交付时间成为可能，从而可以更快地引入和评估业务需求。

基于微服务的其他一些优点包括：

- 模块化，这是这种架构风格所固有的。
- 良好的可测试性。
- 替换系统部件（例如单个服务、数据库、消息代理或云提供商）时的灵活性。
- 与遗留系统的集成：不需要迁移整个应用，只需迁移需要开发的部分即可。
- 支持分布式开发：独立开发团队可以并行地处理多个微服务。
- 可扩展性：微服务可以独立于其他服务进行扩展。

另一方面，微服务也有缺点：

- 需要成熟的 DevOps 方法和对自动化 CI/CD 的依赖。
- 更难调试，并且需要更好的监视和分布式跟踪。
- 额外的开销（就辅助服务而言）可能会抵消小应用所带来的好处。

现在，了解一下以这种风格编写的服务特征。

微服务的特点

由于微服务风格是最近才出现的，所以对微服务没有明确的定义。根据 Martin Fowler 的说法，微服务有几个基本特征：

- 每个服务都是可替换和可升级的组件。这与服务之间更容易部署和松耦合相关联，而不是将组件作为单应用程序中的库。后一种情况下，当替换库时，常常要重新部署整个应用。
- 每个服务都应该由一个跨职能团队开发，专注于特定的业务功能。听说过康威定律吗？

"设计系统的架构受制于产生这些设计组织的沟通结构。"

—— Melvyn Conway, 1967

如果没有跨职能的团队，最终会陷入软件孤岛。

- 每个服务都应该是一个产品，在其生命周期内由开发团队把控，这与项目思维形成了鲜明对比。项目思维中开发软件，然后把它交给别人维护。
- 服务应该有智能终端，并使用转储管道。这与传统服务不同，传统服务通常依赖于**企业服务总线 (ESB)** 逻辑，通常管理消息的路由，并根据业务规则进行转换。在微服务中，通过将逻辑存储在服务，中并避免与消息传递组件耦合，可以实现内聚。使用“哑”消息队列（如ZeroMQ）有助于实现这一目标。
- 服务应该以分布式方式进行管理，组织中通常使用一种特定的技术栈编写。当划分为微服务时，每个人都可以选择最适合自己的需求的服务。管理并确保每个微服务 24/7 运行是由负责该特定服务的团队完成的，而非中央部门。Amazon、Netflix 和 Facebook 等公司遵循这一方法，让开发人员对其服务在生产过程中的完美执行负责，有助于确保产品的高质量。
- 服务应该以分布式方式管理它们的数据。每个微服务都可以选择适合的数据库，而不是为所有微服务提供统一的数据库。拥有分布式数据可能会给数据更新带来一些挑战，但这样的实现具有更好的扩展性。这就是为什么微服务经常以无事务的方式进行协调，并提供最终一致的结果。
- 服务使用的基础设施应该自动化管理。为了有效的处理几十个微服务，就需要有持续集成和持续交付，否则部署服务将是地狱般的体验。所有测试的自动化运行将节省大量时间。在此基础上实施持续部署将缩短反馈周期，并可以让客户更快地使用新特性。
- 微服务应该为依赖服务的失败做好准备。在具有多部件的分布式部署环境中，有一些部件偶尔会出现中断是很正常的现象。服务应该能够优雅地处理此类故障，例如断路器或隔板（在后面会进行介绍）。为了使架构具有自愈能力，需要能恢复出现故障的服务，甚至提前知道哪些服务即将崩溃。为此，实时监控延迟、吞吐量和资源使用情况就很重要。可以去了解一下 Netflix 的 Simian Army 工具包，它是创建自愈架构特别好的工具。
- 基于微服务的框架应该可持续发展。在设计微服务与架构一起工作时，应该考虑到如何替换单个微服务，甚至是替换一组微服务。恰当地设计服务是一件棘手的事情，特别是因为曾经的代码模块中的复杂性，现在可以作为服务间复杂的通信方案出现，但这就更难管理了——这就是“意大利面集成”（Spaghetti Integration）。这意味着架构师的经验和技能集比传统服务或单体方法更加重要。

在此基础上，以下有许多（但不是所有）微服务的共同特征：

- 通过网络协议，独立进程可以互相通信
- 使用与技术无关的协议（如 HTTP 和 JSON）
- 保持小规模服务，运行时开销低

现在，了解了基于微服务的系统特征，接下来，来看看这种方法与其他架构风格的比较。

微服务和其他架构风格

微服务可以作为架构模式使用。但经常与其他架构组合使用，例如本地云计算、无服务器应用，并且主要与轻量级应用容器组合使用。

面向服务的架构带来了低耦合和高内聚。如果应用得当，微服务也可以做到这一点。然而，这可能有点难度，因为需要良好的直觉，将系统划分为数量众多的微服务。

微服务与其较大的同类服务之间有很多相似之处，可以使用基于 SOAP、REST 或 gRPC 的消

息传递，并使用消息队列等技术来进行事件驱动。也有众所周知的模式来实现所需的质量属性，例如容错（通过隔离故障组件），但是为了拥有一个高效的架构，必须决定处理组件的方法，如 API 网关、服务注册、负载平衡、容错、监视、配置管理，当然还有要使用的技术栈。

扩展微服务

微服务与单应用的扩展性不同。在单应用中，整个功能由单个进程处理。扩展应用程序意味着，会在不同的机器上重复这个过程。这种可扩展性不考虑哪些功能会大量使用，哪些功能不需要额外的资源。

对于微服务，每个功能组件都可以作为单独的服务处理，这就是一个独立的流程。为了扩展基于微服务的应用，只需要将使用更多资源的部分复制到不同的机器上即可，这种方法可以更容易且更好地利用现有资源。

过渡到微服务

大多数公司都有某种程度的整体代码，他们不希望立即使用微服务重写这些代码，但仍然希望过渡到这种架构。这样，可以通过渐进式添加与集成块交互的服务，来逐步适应微服务。可以创建新的功能作为微服务，或只是删除整体的一些部分，然后创建微服务。

更多关于微服务的信息，包括如何从零开始构建自己的微服务，可以在第 13 章中看到。

2.5. 探索基于事件的架构

基于事件的系统，是那些架构围绕着事件处理的系统。有生成事件的组件、传播事件的通道和响应事件的侦听器，也可能触发新的事件。这种风格更加异步和低耦合，使得其成为提高性能和可扩展性的好方法，同时也是一种易部署的解决方案。

除了这些优势，还有一些挑战。其中之一是创建这类系统的复杂性。所有队列必须具有容错功能，以便事件在处理过程中不会丢失。以分布式方式处理事务也是一个挑战。使用关联 ID 模式跟踪进程之间的事件，以及监视技术，可以节省调试时间和思考的时间。

基于事件的系统包括流处理器和数据集成，以及以低延迟或高可扩展性为目标的系统。

现在让我们讨论一下此类系统中常见的拓扑结构。

2.5.1 基于事件的拓扑

事件驱动体系结构有两种主要拓扑：基于代理和基于中介。这些拓扑的不同之处在于事件在系统中的走向。

中介拓扑，适合用于处理需要多个可独立执行的任务或步骤的事件。最初，产生的所有事件都需要放在中介的事件队列中。中介器知道为了处理事件需要做什么，会通过每个处理器的事件通道将事件分派给相应的事件处理器。

敏感的读者可能会突然想到，这是关于“业务流程是如何流动的”话题。当然，可以在业务流程管理（BPM）或业务流程执行语言（BPEL）中实现此拓扑。并且，也可以使用 Apache Camel、Mule ESB 等方式进行实现：

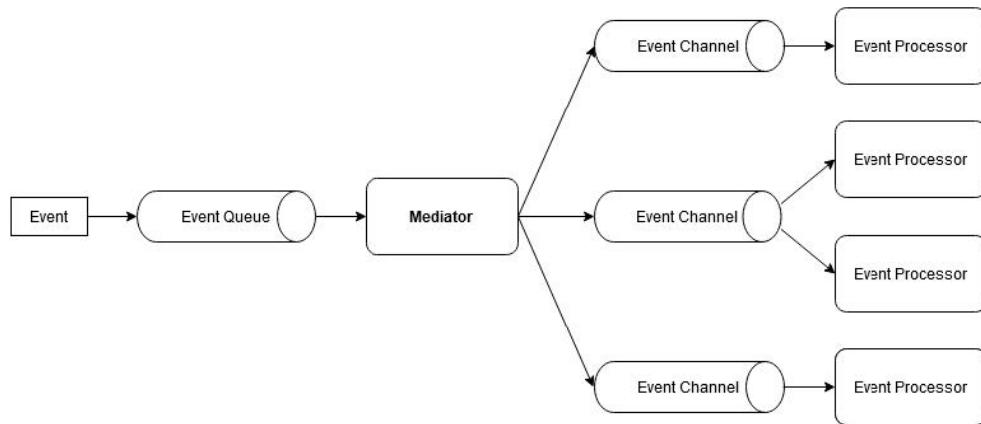


图 2.1 - 中介拓扑

另一方面，代理是轻量级组件，包含所有队列，不协调事件的处理。可以要求收件人订阅特定类型的事件，然后简单地转发他们感兴趣的事件。许多消息队列依赖于代理，例如 ZeroMQ，用 C++ 编写，目标是零浪费和低延迟：

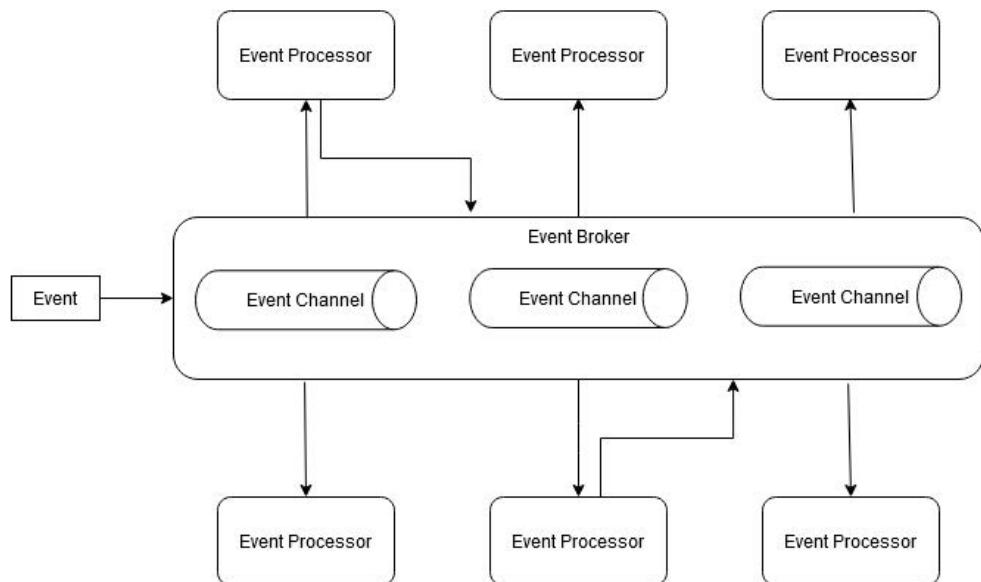


图 2.2 - 代理拓扑

现在，已经了解了基于事件的系统中使用的两种常见拓扑，接下来让看下以事件为核心的架构模式。

2.5.2 事件源

可以将事件视为通知，其中包含要处理的通知服务所需的数据。还可以以另一种方式来看——状态的改变。若能够知道错误发生时的状态，以及需要进行的更改，那么用应用程序逻辑调试就会很容易。这是事件源的一个红利，它通过简单地按照事件发生的顺序记录所有事件，捕获发生在系统上的所有更改。

通常，会发现服务不再需要将其状态存到数据库中，因为将事件存储到系统的某个地方就足够了，也可以异步完成。从事件源中获得的另一个好处是免费的完整审计日志：

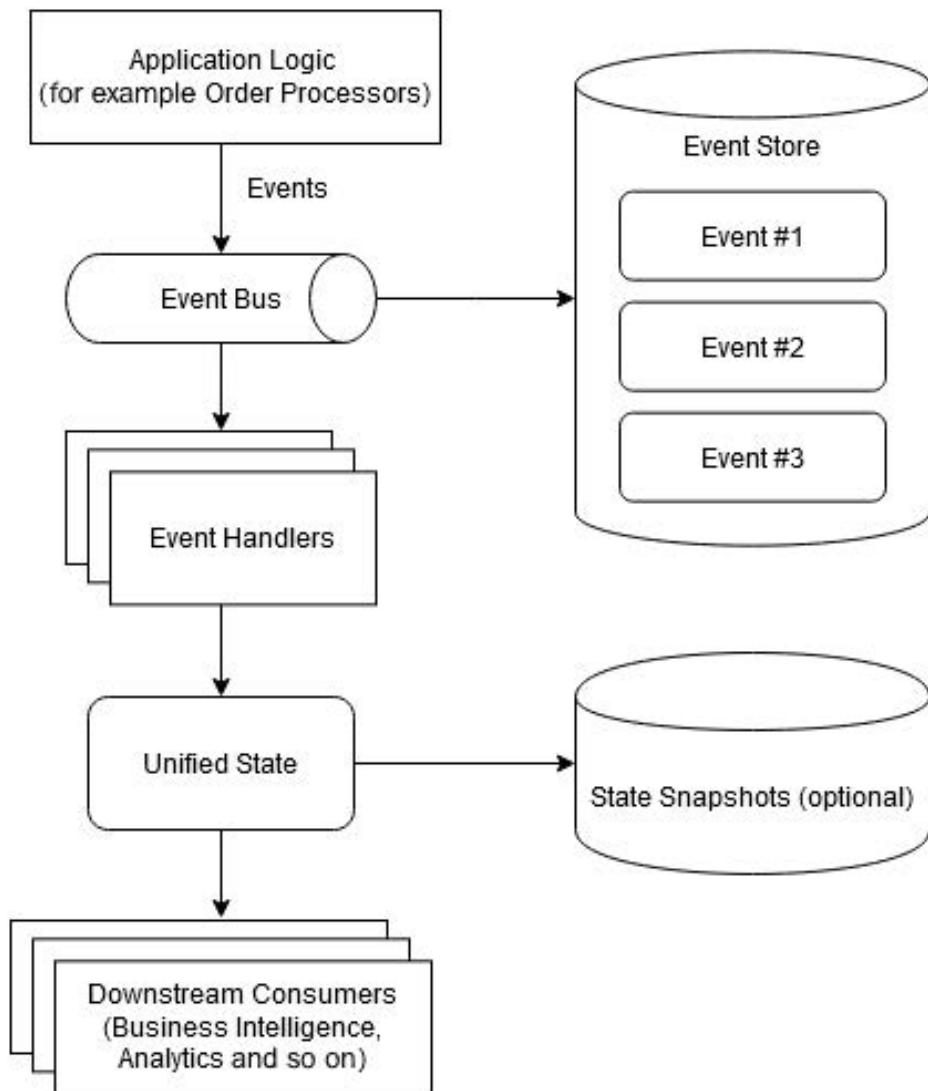


图 2.3 - 事件源架构提供应用程序状态的统一视图，可以使用该视图并创建定时快照以实现快速恢复

由于减少了对数据同步的需求，事件源系统通常提供较低的延迟，所以更适合交易系统和活动跟踪器。

接下来，了解一下另一种流行的架构风格。

2.6. 探索分层架构

如果初始架构开始看起来像意大利面，或者只是想阻止架构成为意大利面，将组件分层结构化可能会有所帮助。还记得“模型-视图-控制器”吗？或者类似的模式，例如“模型-视图-视图模型”或“实体-控制-边界”？这些都是分层架构的示例（如果各层在物理上彼此分离，也称为 N 层架构）。可以在层中构造代码，创建微服务层，或将此模式应用到可以带来利益的其他领域。引入层的主要原因是，其提供了抽象和关注点分离，还可以降低复杂性，同时改进解决方案的模块化、可重用性和可维护性。

一个真实的案例是在自动驾驶中，各个层可以用于分层决策：最底层处理汽车的传感器，然后另一层提取传感器数据的单个特征。在这一层之上，可能还有一层来确保所有特征都是安全行为。当更换另一款汽车的时，只需要更换最底层的传感器即可。

分层架构通常很容易实现，因为大多数开发者对层的概念相当熟悉——只需要开发几个层，然后像下图那样堆叠就好：

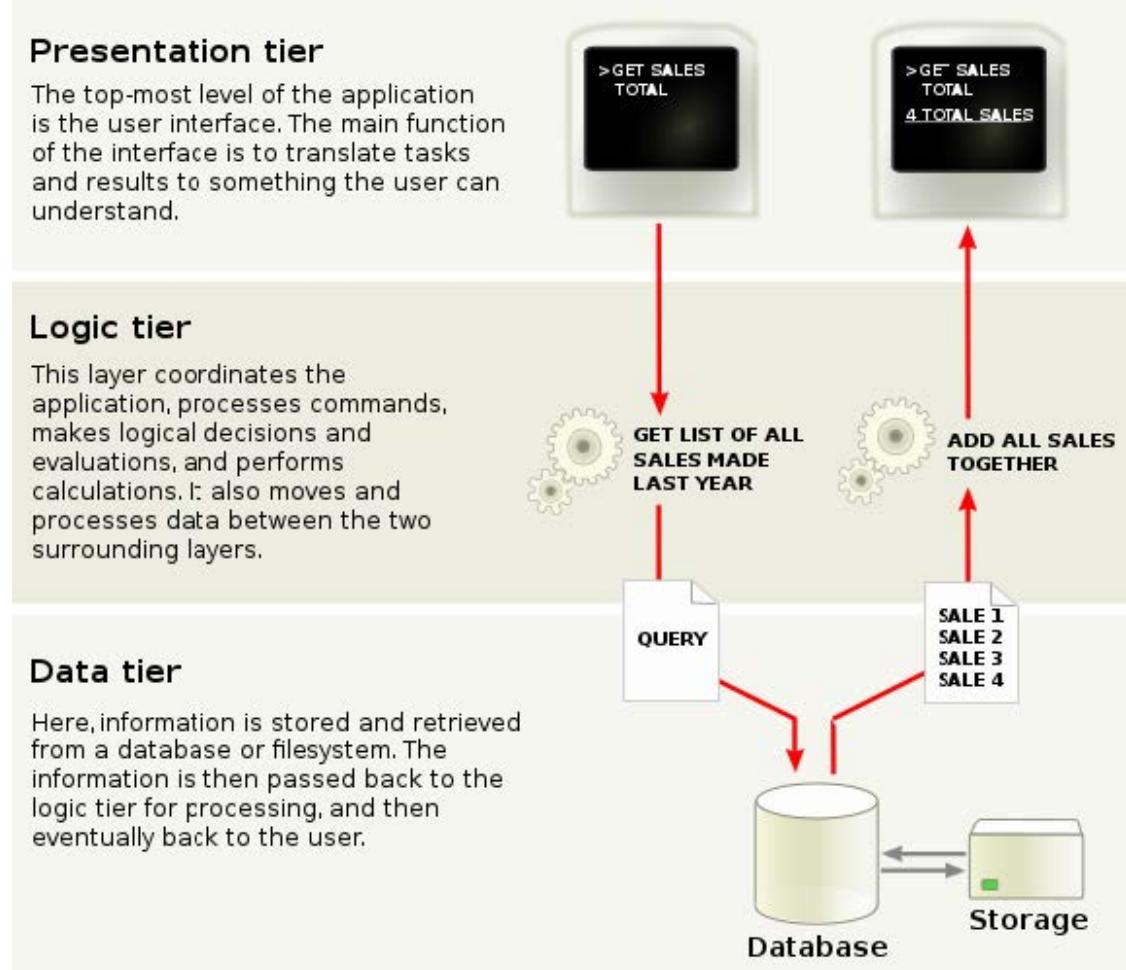


图 2.4 - 表示层中使用文本接口的三层架构示例

创建分层架构的挑战在于层之间稳定和定义良好的接口。通常，可以在一个层之上堆叠多个层。若有一个用于逻辑领域的层，那么可以在表示层之上，也可以是为其他提供 API 服务层之上。

分层不总是一件好事。对于微服务，分层主要出现在两个场景中。第一种是将一组服务与另一组服务分开时，可以使用一个易变化的层与业务合作伙伴进行交互，其中包含经常变化的内容，以及另一个面向业务功能的层。后者无法快速的改变，通常使用的是稳定的技术。还有一种观点认为，不太稳定的组件应该依赖于更稳定的组件，因此很容易看出，这里可以有两个层，其中一个可以面向客户（这都取决于具体业务功能）。

另一种情况是创建层以反映通信结构（又见面了，康威法则）。这可能会减少团队之间的沟通，从而导致创新的减少，因为现在团队不会很好地了解彼此的内部或想法。

接下来，讨论另一个经常与微服务一起使用的分层架构——面向前端的后端。

2.6.1 面向前端的后端

许多前端都依赖于同一个后端，这种情况并不少见。假设有一个移动应用程序和一个 Web 应用程序，都使用相同的后端，开始可能是个不错的设计选择。然而，当这两个应用的需求和使用场景发生分歧时，后端将需要进行频繁地更改，并只服务于其中一个前端。这可能导致后端必须支持相互竞争的需求，比如：更新数据存储的两种不同方法或提供不同场景的数据。同时，前端逐渐需要更多带宽才能与后端正正常通信，这也导致移动应用对电池的使用增加。此时，应该考虑为每个前端引入单独的后端。

这样，就可以将面向用户的应用看作是具有两个层的实体：前端和后端。后端可以依赖于由下游服务的另一层。如图所示：

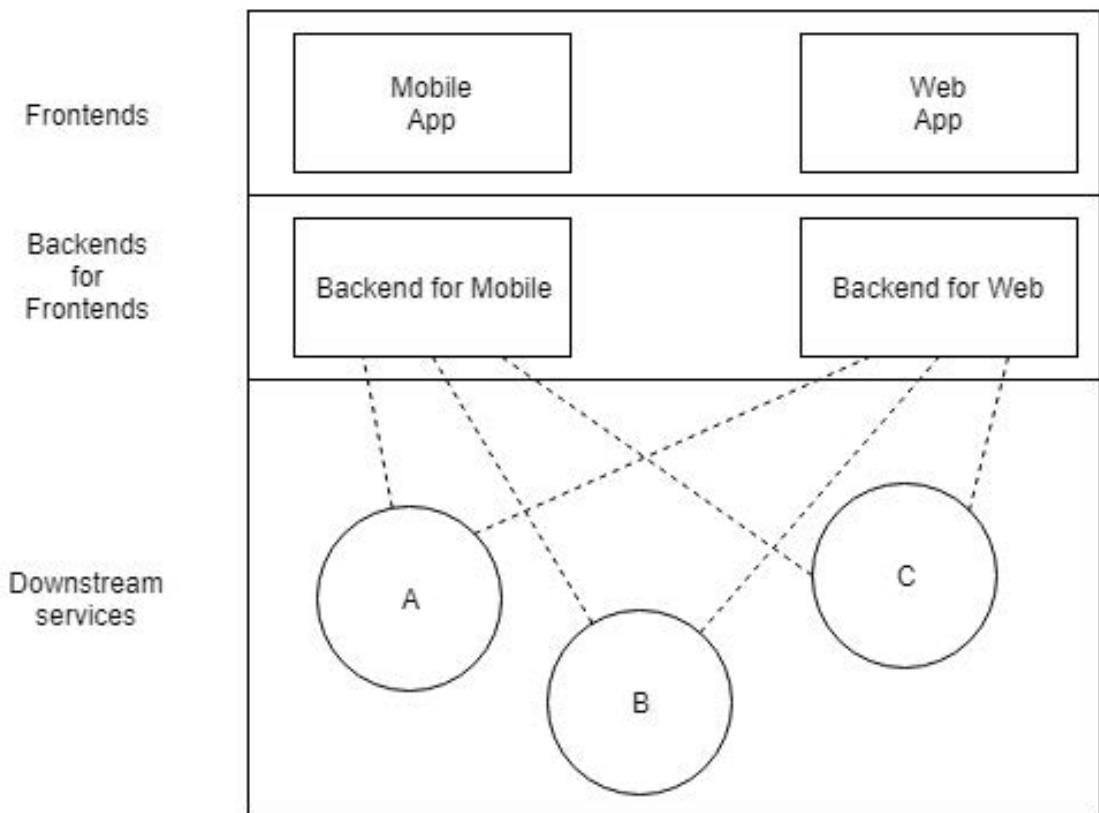


图 2.5 - 面向前端的后端模式

使用**面向前端的后端 (BFF)** 的缺点是有些代码需要复制。只要这能够加速开发，并且从长期来看这只要不是一种负担，就没有问题，但需要密切关注在下游服务中聚合逻辑复制的可能性。有时，引入服务只是为了聚合类似的调用，可以帮助解决重复的问题。如果有多个前端，可以部分共享后端，而不会导致竞争。若正在为 iOS 和 Android 开发移动应用程序，可以考虑使用相同的后端，而为 Web 和/或桌面应用，则需要使用单独的后端。

2.7. 了解模块化架构

Note

本节中，模块指的是可以在运行时加载和卸载的软件组件。C++20 的模块，请参阅第 5 章。

如果曾经需要在尽可能少的停机时间内运行一个组件，但由于某种原因不能应用常见的容错模式，比如：服务的冗余副本，那么使用基于组件模块的模式可以解决这个问题。或者只是因模块化系统而对其感兴趣，该系统对所有模块可以进行版本控制，并且可以轻松查找所有可用的服务，以及基于模块的系统可能导致的解耦、测试性和增强团队协作。这就是为什么为 Java 创建了**开放式服务平台技术 (Open Service Gateway Initiative, OSGi)** 模块，并在多个框架中移植到 C++。使用模块的架构示例包括 IDE(如 Eclipse)、软件定义网络 (如 OpenDaylight) 项目或家庭自动化软件 (如 OpenHAB)。

OSGi 还允许模块之间的依赖关系进行自动化管理，控制初始化和销毁，以及控制运行。由于它是面向服务的，可以将使用 OSGi 的服务看作在“容器”中拥有微型(?)服务的东西。这就是 C++ 实现命名为 C++ 微服务的原因。要了解其实际操作，请参阅扩展阅读部分中的入门指南。

C++ 微服务框架使用了一个有趣的概念，一种处理单例的新方法。*GetInstance()* 静态函数将不只是传递一个静态实例对象，并且返回一个绑定上下文获取的服务引用。因此，可以配置的服务将取代单例对象。还可以避免静态反初始化的失败，即多个依赖于彼此的单例，必须按照特定的顺序进行卸载。

2.8. 总结

本章中，讨论了各种架构风格都可以应用到软件中。已经讨论了单体架构，讨论了面向服务的架构。然后，转向了微服务，并讨论了提供的外部接口和交互的各种方法。也了解了如何编写 REST 式服务，以及如何创建一个有自愈性，且易于维护的微服务架构。

还展示了如何创建简单的客户端来使用同样简单的服务。之后，讨论了构建架构的各种其他方法：事件驱动的方法、基于运行时模块的方法，并展示了分层的位置和原因。现在，我们已经了解了如何实现事件源，以及识别何时使用 BFF。现在明白了架构风格是如何实现一些质量属性的，以及这会带来怎么样的挑战。

下一章中，将了解如何知道在给定的系统中，哪些属性比较重要。

2.9. 练习题

1. REST 式服务的特点是什么？
2. 可以使用什么工具包来创建自愈分布式体系结构？
3. 微服务应该使用集中存储吗？原因？
4. 什么时候应该编写有状态服务，而不是无状态服务？
5. 代理与中介拓扑有什么不同？
6. N-tier 架构和 N-layer 架构有什么区别？
7. 如何使用基于微服务的架构来取代整体架构？

2.10. 扩展阅读

- Flygare, R., and Holmqvist, A. (2017). Performance characteristics between monolithic and microservice-based systems (Dissertation). Retrieved from <http://urn.kb.se/resolve?urn=urn:nbn:se:bth-14888>
- Engelen, Robert. (2008). A framework for service-oriented computing with C and C++ web service components. ACM Trans. Internet Techn. 8. 10.1145/1361186.1361188
- Fowler, Martin. Microservices – A definition of this new architectural term. Retrieved from <https://martinfowler.com/articles/microservices.html#MicroservicesAndSoa>
- Getting Started – C++ Micro Services documentation. Retrieved from http://docs.cppmicroservices.org/en/stable/doc/src/getting_started.html

第 3 章 功能和非功能需求

作为一名架构师，需要了解哪些需求对架构很重要，以及为什么重要。本章将介绍解决方案的各种需求——功能性和非功能性。功能需求说明解决方案中需要有什么。另一方面，非功能性的说明解决方案应该是怎样的。

本章将讨论以下内容：

- 需求类型
- 认识架构的重要需求
- 收集需求
- 文档化需求
- 文档化架构
- 选择合适的角度进行记录
- 生成文档

本章结束时，将了解如何识别和分类这两种类型的需求，以及如何创建具有清晰描述需求的文档。

3.1. 相关准备

要复制我们从源代码生成文档的步骤，您必须安装 CMake、Doxygen、Sphinx、m2r2 和 Breathe。我们正在使用 ReadTheDocs Sphinx 主题，所以也请安装它。可以使用上述工具的最新版本。

本章的代码可以在以下 GitHub 页面找到: <https://github.com/PacktPublishing/SoftwareArchitecture-with-Cpp/tree/master/Chapter03>。

3.2. 需求类型

创建软件系统时，应该不断地问自己，所做的是否是客户所需要的。很多时候，客户不知道如何才能能满足自己的需求。架构师的角色是发现产品的需求，并确保需求得到满足。这里需要考虑三种不同类型的需求：功能需求、质量属性和约束条件。

3.2.1 功能需求

第一组是功能需求。这定义了系统应该做什么，应该提供什么功能。

TIP

功能并不总是影响体系结构，因此必须注意哪些需求将影响解决方案。

通常，功能需求具有某些必须满足的特性，那么在架构上就很重要。考虑为参加多米尼加博览会 (Dominican Fair) 的商人和游客开发一款应用程序，该博览会是在 Gdańsk 举办的年度活动，内容包括音乐、各种艺术和商店。下面是一些功能需求的例子：

- 作为一个店主，想过滤包含特定产品的订单。

- 单击“订阅”按钮，将客户添加到选定商家的通知列表中。

第一个要求，必须有一个具有搜索功能的跟踪订单和产品的组件。根据 UI 的显示方式和应用的规模，可以只在应用中添加一个简单的页面，或者可能需要 Lucene(一个全文搜索引擎) 或 Elasticsearch(搜索引擎解决方案) 等特性。这意味着**体系结构重要需求 (ASR)**，会影响架构。

第二个例子更直接，我们知道需要订阅和发送通知的服务。这无疑是架构上的重要功能需求。现在来看一些可以是 ASR 的**非功能需求 (NFR)**。

第一个需求实际上是以用户故事的形式给出的。用户故事是以以下格式给出的需求：“作为一个<角色>，我可以/想要<能力>，从而可以<受益>”。这是一种表达需求的常用方法，可以帮助相关负责人和开发人员找到共同点，从而更好地进行沟通。

3.2.2 非功能需求

非功能需求关注的不是系统应有什么功能，而是系统应该如何，以及在何种条件下执行所述功能。这是由两个主要的需求组成：**质量属性 (QA)** 和**约束条件**。

质量属性

质量属性 (QA) 是解决方案的特征，例如：性能、可维护性和用户友好性。软件可以具有几十种(如果不是数百种的话)不同的质量。在选择软件包含的功能时，试着只关注那些重要的功能，而不是所有出现在脑中的功能。质量属性需求的例子如下所示：

- 正常负载下，系统将在 500ms 以内对 99.9% 的请求作出响应(不要忘记定义“正常负载”)。
- 网站不会存储在支付过程中泄漏的客户信用卡数据(保密)。
- 更新系统时，如果更新任何组件失败，系统将回滚到更新之前的状态(生存性)。
- 作为 Windows、macOS 和 Android 的用户，希望能够同时使用这些系统(可移植性。理解需求是否需要支持桌面、移动和/或 Web 等平台)。

虽然在列表中捕捉功能需求非常简单，但不能对质量属性需求进行同样的操作。幸运的是，有其他方法可以做到这一点：

- 有一些可以用**完成的定义**或**验收标准**来表示任务、故事和发布。
- 其他的可以直接表示为用户故事(如前面的最后一个示例所示)。
- 还可以将它们作为设计和代码审查的一部分进行检查，并为其中一些创建自动化测试。

约束条件

约束条件是在交付项目时必须遵循的决策。这些决策可以是设计决策、技术决策，甚至是政治决策(关于人员或组织事务)。另外两个常见的约束是**时间和预算**。约束条件的例子有：

- 团队的规模永远不会超过四名开发人员、一名 QA 工程师和一名系统管理员。
- 由于我们公司在其所有现有产品中都使用了 Oracle DB，所以新产品也必须使用，这样才能充分利用我们的专业知识。

非功能性需求总是会影响架构。重要的是不要过度指定，因为在产品开发过程中，假阳性将是一种持续性负担。同样重要的是，不要对它们要求不足，因为这可能会导致错过销售时机或未能遵

守监管机构的要求。

下一节中，将了解如何在这两个极端之间取得平衡，并只关注在特定情况下那些真正重要的需求。

3.3. 认识架构的重要需求

设计软件系统时，通常要处理几十个或数百个不同的需求。为了理解并想出一个好的设计，需要知道哪些是重要的，哪些是可以实现的。不管设计决策是什么，应该学会如何识别最重要的需求，这样就可以首先关注它们，并在尽可能在短时间内交付具有更大价值的产品。

TIP

应该使用两个指标对需求进行优先级划分：业务价值和对架构的影响。在两个比重排名较高的项目是最重要的，应作优先处理。如果提出了太多这样的需求，应该重新考虑这些需求的优先级。如果无法对优先级进行排序，那么可能是这个系统无法实现。

ASR 对系统架构有影响，其可以是功能性的，也可以是非功能性的。那如何确定哪些是重要的呢？如果将某个特定需求的去掉，就需要创建一个不同的架构，那么这个需求就是 ASR。因为需要重新设计系统的某些部分，所以延迟发现这些需求通常会花费时间和金钱。如果不是整个解决方案的话，只能希望这不会再损失其他资源和/或声誉。

TIP

从架构工作的一开始就将具体的技术应用到架构中，这是一个常见的错误。我们强烈建议首先收集所有的需求，专注于那些对架构重要的需求，然后再决定架构项目所依赖的技术和/或技术栈。

既然识别 ASR 很重要，那就来了解一些有帮助的模式。

3.3.1 架构指标的意义

如果有与外部系统集成的需求，这很可能会影响架构。先来了解需求是 ASR 的一些常见指标：

- **需要创建软件组件进行处理：**包括发送电子邮件、推送通知、与公司的 SAP 服务器交换数据或使用特定的数据存储。
- **对系统影响较大：**核心功能通常会定义系统应该如何。减少关注点，授权、审核或具有事务性行为，也是很好的例子。
- **难以实现：**低延迟是一个很好的例子：除非在开发早期就考虑到这一点，否则实现这可能会是一场鏖战，特别是意识到在热路径上无法真正的进行垃圾收集时。
- **强制对特定架构进行权衡：**如果成本太高，设计决策可能需要对某些需求进行折衷，以支持其他更重要的需求。将这些决策记录在某个地方，并注意到这里处理的是 ASR，这是一个很好的实践。如果需求以某种方式限制了架构师或产品，那么它很可能对架构产生重大影响。如果想要得到最好的架构，就一定要阅读架构权衡分析方法 (ATAM)，可以在扩展阅读中找到它。

应用程序运行的约束和环境也会影响架构。嵌入式应用程序，需要以不同于云中应用的方式进行设计，经验较少的开发者在开发应用的过程中，应该使用简单、安全的框架，而不是使用陡峭的学习曲线或自己开发的框架。

3.3.2 ASR 的识别及处理方法

与直觉相反，许多架构的重要需求不明显，这是由两个因素造成的。它们可能很难定义，即使它们被描述了，也可能是模糊的。客户可能不清楚他们需要什么，但作为架构师应该主动询问，避免任何假设。如果系统要发送通知，必须知道这些通知是实时的，还是每天发送一封就足够了，因为前者可能需要建立发布者-订阅者架构。

大多数情况下，需要做一些假设，因为不是所有事情都已知。如果发现一个需求挑战了之前的假设，那可能是一个 ASR。如果假设只在凌晨 3 点到 4 点之间维持服务，并且意识到来自不同时区的客户仍需要使用它，那么事实将会挑战假设，并可能改变产品的架构。

人们往往倾向于在项目的早期阶段模糊地对待质量属性，特别是缺乏经验或技术的个人。另一方面，这是解决 ASR 的最佳时机，因为现在在系统中实现它们，是成本最低的时刻。

许多人在指定需求时，喜欢在没有仔细考虑的情况下使用模糊的短语。如果正在设计一个类似于 Uber 的服务，一些例子可以是：当收到一个 *DriverSearchRequest*，系统必须快速回复一个 *AvailableDrivers* 消息，或系统必须是可用的 24/7。

提出问题后，会发现每月具有 99.9% 的完美可用性，而快速实际上只需几秒钟。这些短语总是需要确认，了解它们背后的基本原理。也许这只是某人的主观意见，没有任何数据或业务需求的支持。另外，在请求和响应的情况下，质量属性隐藏在另一个需求中，这使得它更难捕获。

最后，即使某些系统服务于类似的目的，对于这个系统架构上重要的需求，对于另一个系统并不一定同样重要。随着时间的推移，当系统开始发展，并与越来越多的其他系统进行通信，其中一些服务会变得更加重要。当产品需求发生变化，原来不重要的需求可能会变得很重要。这就是为什么没有什么诀窍可以明确说明哪些需求是 ASR，哪些不是。

了解了如何区分重要需求，就知道要寻找什么了。接下来，让我们了解一下去哪里了解需求。

3.4. 收集需求

已经知道了需要关注哪些需求，那就聊一下收集这些需求的技巧吧。

3.4.1 了解背景

挖掘需求时，应该考虑背景。必须确定哪些可能出现的问题可能对产品产生负面影响，这些风险通常来自外部。重新审视一下类似 Uber 的服务场景，服务可能面临的风险是政策的变化：应该意识到，一些国家可能试图改变政策，将这个产品从他们的市场中移除。Uber 缓解这种风险的方法，是让当地的合作伙伴来应对地区性限制。

除了未来的风险之外，还必须了解当前的问题，例如公司中缺乏相关领域的专家，或市场上激烈的竞争。可以这样做：

- 关注所有的假设，最好有一个专门的文件来跟踪这些。
- 如果可能的话，通过问题来确认或消除假设。

- 需要考虑项目内部的依赖关系，可能会影响开发进度。另外是要塑造公司日常的业务规则，产品需要遵守这些规则，并且需要强化这些规则。
- 此外，如有足够多的与用户或业务相关的数据，应该尝试挖掘这些数据，以获得深刻的理解，并找到有用的模式，这些模式有助于对产品及其架构的决策。如果已经有一些用户，但无法挖掘数据，那么观察其行为即可。

理想情况下，可以使用部署系统执行日常任务时进行记录。通过这种方式，不仅可以自动化部分工作，还可以使工作流程更高效。但没人会心甘情愿的改变习惯，所以最好是逐步引入改变。

3.4.2 了解现有文档

现有文档是一个很好的信息来源，尽管它们可能存在一些问题。应该预留一些时间，至少熟悉所有与当前工作相关的文档，其可能隐藏了一些需求。另一方面，文档从来不是完美的，很可能会缺少一些重要的信息。同样，也需要为它已经过时做好心理准备。当涉及到架构时，除了阅读文档之外，应该与相关人员进行咨询，阅读文档是为咨询做准备的一种方式。

3.4.3 了解责任相关方

要成为一名成功的架构师，必须学会在需求直接或间接地来自业务人员时与他们进行沟通。无论他们是来自你的公司还是客户，都应该了解他们的业务背景。例如，必须知道以下内容：

- 业务的驱动力是什么？
- 公司的目标是什么？
- 产品将帮助实现哪些目标？

这就需要与来自管理或执行人员等人达成了共识，收集关于软件的需求将会容易得多，例如公司需要保密用户的隐私，可以要求存储尽可能少的用户数据，并使用只存储在用户设备上的密钥对数据进行加密。若这些要求来自于公司文化，那么对一些员工来说就是早已心知肚明的，甚至可以直接进行表达。了解业务背景有助于提出适当的问题，并反过来帮助公司。

话虽如此，责任相关方的需求不一定直接反映在公司目标。对于要提供的功能或软件应该实现的指标，他们也可以有自己的想法，也许会有一个经理承诺给他的员工一个学习新技术或使用特定技术的机会。如果这个项目对他们的职业生涯很重要，则可以成为架构师强有力的盟友，甚至说服别人接受架构师的决定，成为架构师的说客。

另一组重要的责任相关方是负责部署软件的人员。它们可以带有自己的需求子组，称为过渡需求。这些例子包括用户和数据库迁移、基础设施转换或数据转换，所以不要忘记和他们尽早的进行沟通，并收集这些信息。

3.4.4 从责任相关方处收集需求

此时，应该有一个责任相关方列表及其角色和联系信息的清单了吧。现在是时候使用它的时候了，一定要花时间与每个相关方的负责人讨论他们从系统中需要什么，以及他们的想法。可以进行面谈，如面对面的会议或小组会议。当与负责人交谈时，也可以帮助他们做出明智的决定——告知他们对最终产品期望结果的可能性。

相关方的负责人通常会说他们所有的需求都同等重要。试着从业务价值的角度说服他们，从而根据已给定的需求，制定优先级。当然，会有一些关键的任务需求，但最可能的情况是，如果一堆需求都无法支持，项目也不会获批，更不用说需求清单中的东西了。

除了面谈，也可以开会组织研讨，就像头脑风暴一样。当有共识达成，并且每个人都知道为什么要参与这样的项目时，就可以开始向每个人了解尽可能多的使用场景。完成了这些，就可以梳理需求和应用场景并形成故事，确定哪些需要优先考虑。最后，需要对所有的故事进行完善的处理。研讨不仅仅是关于功能需求，每个使用场景也可以分配相应的质量属性。进行提炼之后，所有的质量属性都必须可测量。注意，不需要将所有的相关方的负责人都牵扯到此类事件中，因为根据系统的规模，这些事件有时可能需要很长的时间才能完成。

已经了解了如何使用技术和资源来挖掘需求。接下来，来了解一下如何将现有的发现记录到精心制作的文档中。

3.5. 文档化需求

完成了前面的步骤之后，就可以将收集到的需求放在一个文档中进行细化。文档采用什么形式以及如何管理并不重要，重要的是要有一份文档，将所有相关方放在同一页上，说明产品需要什么，以及每个需求将带来什么价值。

需求是由所有相关方产生的，他们中的大部分人需要阅读文档。架构师需要制作这份文档，以便能为具有各种技术技能的人带来价值，从客户、销售人员和市场人员，到设计师和项目经理，再到软件架构师、开发人员和测试人员。

有时可以准备两个版本的文档，一个是为最接近项目业务方面的人，另一个是为开发团队，更技术性的版本。通常，只要编写一份能看懂的文档就足够了，其中的章节（有时是单个段落）或整章都会有更多技术细节。

现在来看看哪些部分需要写入需求文档。

3.5.1 记录背景

需求文档应该作为进入项目的入口点，需要概述产品的目的、谁使用、如何使用。设计和开发之前，产品团队成员应该阅读文档，从而清楚地知道要做些什么。

背景部分应该提供系统的概述——为什么要构建，试图实现什么业务目标，以及将交付什么关键功能。

可以描述一些典型的用户角色，例如首席技术官 John，或者司机 Ann，使读者可以把系统的用户当作真实的人来思考，并理解对他们的期望。

在了解背景一节中描述的内容也应该有总结的部分，有时需要在文档中设立单独的部分。背景和范围部分应该提供大多数非项目相关方所需的所有信息，这些信息应该简洁明了。

对于想要研究并决定的开放性问题也是如此。对于你所做的每一个决定，最好注意以下几点：

- 决定本身是什么
- 谁来做，何时做
- 这样做的理由是

现在了解了如何记录项目的背景，继续了解如何正确地描述范围。

3.5.2 记录范围

这个部分应该定义什么在项目范围内，什么是在项目范围外。提供一个基本准则，说明为什么要以特定的方式定义范围，特别是对一些不符合要求的内容时。

本节还应该介绍高级功能和非功能性需求，详细信息应在本文档的后续部分中进行描述。如果熟悉敏捷实践，可以在此描述一些传奇和大用户的故事。

如果架构师或相关方对范围有假设，应该在这里提到这些假设。如果由于问题或风险，范围可能会发生变化，那么也应该记录相关的文字，以及需要做出的权衡。

3.5.3 记录功能性需求

每个需求都应该精确和可测试。考虑这个例子：“系统将为司机提供一个排名系统。”如何创建针对它的测试？最好是为排名系统创建一个章节，并在那里定义其需求。

考虑另一个例子：如果有一位免费司机离乘客很近，应该告知他们即将到来的乘车请求。如果有多个合适的司机怎么办？我们能描述的最近距离是多少？

此需求既不精确，又缺乏业务逻辑。我们只能希望没有可用司机的情况是另一个需求。

2009 年，Rolls Royce 开发了简单需求方法语法 (EARS) 来解决这个问题。在 EARS 中，有五种基本类型的需求，它们以不同的方式编写并服务于不同的目的，可以将它们组合起来创建更复杂的需求。这些基本的问题如下：

- **无处不在型需求**：“\$SYSTEM 应是\$REQUIREMENT,”。例如，应用将使用 C++开发。
- **事件驱动型需求**：“当\$trigger 是\$optional_precondition 时，\$SYSTEM 应是\$REQUIREMENT,”。例如：当出现新订单时，网关将生成一个 NewOrderEvent。
- **意外行为型需求**：“若是\$condition，则\$SYSTEM 应是\$REQUIREMENT,”。例如：如果处理请求的时间超过 1 秒，该工具将显示进度条。
- **状态驱动型需求**：“当为\$state 时，\$SYSTEM 应是\$REQUIREMENT,”。例如：在开车时，应用会显示地图，帮助司机导航到目的地。
- **自选特性**：“在具有\$feature 时，\$SYSTEM 应是\$REQUIREMENT,”。例如：在有空调的地方，App 会让用户通过手机设置温度。

更复杂的需求示例：在使用双服务器设置时，如果备份服务器在 5 秒内没有收到主服务器的消息，应该尝试将自己注册为一个新的主服务器。

也可以不使用 EARS，但是若正在与不明确的、模糊的、过于复杂的、不可测试的、省略的或措词糟糕的需求作过斗争，EARS 可以提供帮助。无论选择何种方式或措辞，都要确保使用简洁的模型，该模型基于通用语法并使用预定义关键字。为列出的每一个需求分配标识符也是很好的实践方式，这样就可以使用一种简单的方法来引用它们。

当涉及到更详细的需求格式时，应该有以下字段：

- **ID 或索引**：便于识别特定需求。
- **标题**：可以使用 EARS 模板。
- **详细描述**：可以将相关的信息放在这里，例如：用户故事。
- **拥有者**：这个要求服务于谁，可能是产品所有者、销售团队、法律、IT 等。
- **优先级**：不言自明。

- **交付方式:** 如果关键性日期有所要求, 可以在这里注明。

了解了如何记录功能性需求, 继续看看如何记录非功能性需求。

3.5.4 记录非功能性需求

每个质量属性, 例如性能或可扩展性, 都应该在文档中有记录, 并列出特定的、可测试的需求。大多数都是 QA 可测试的, 所以拥有特定的测试标准可以很好地解决问题。当然, 还可以用单独的部分来说明项目的约束条件。

关于措辞, 可以使用 EARS 模板来记录 NFR(NonFunctional Requirements)。或者, 使用在背景信息中定义的角色, 将其指定为用户故事。

3.5.5 管理文档版本的记录

可以采用以下两种方法的一种: 在文档内创建版本日志或使用外部版本控制工具。这两种方法各有利弊, 但建议采用后一种方法。就像对代码使用版本控制系统一样, 也可以将其用于文档。并不是说必须使用存储在 Git 库中的 Markdown, 只要生成一个适用于业务人群的文档, 无论是网页还是 PDF 文件, 这都是很好的方法。或者, 也可以使用在线工具, 比如 RedmineWikis 或 Confluence, 这些页面允许在每次发布的编辑中添加有意义的评论, 描述修改的内容, 并查看版本之间的差异。

如果决定用修订日志的方法, 表中通常需要包含以下字段:

- **修改:** 数字, 标识变更是在文档的哪个迭代中引入的, 还可以为特殊修订添加标签, 例如初稿。
- **更新:** 谁做的改变。
- **审核:** 谁审阅的修改。
- **修改描述:** 修改的提交信息, 说明了发生了什么变化。

3.5.6 记录敏捷项目中的需求

许多敏捷的支持者声称, 记录所有的需求纯粹是浪费时间, 因为它们都可能会发生变化。然而, 一个好的方法是将它们与待办事项列表中的项目进行类似的处理: 在即将到来的 sprint 中开发的项目, 应该比稍后实现的项目定义得更详细。如果不愿意在必要的时候将大任务分解成故事和任务, 那么可以在确定需要实现任务之前, 对粗粒度需求只进行概要性的描述。

TIP

确定需求的来源, 这样就可以知道哪里会提供输入, 以便在将来对其进行细化。

以多米尼加博览会为例。在下一个 sprint 中, 将构建供访问者查看的商店页面。再下一个 sprint 中, 将添加订阅机制。需求如下所示:

ID	优先级	描述	负责人
DF-42	P1	商店页面必须可以显示商店的库存, 并附有照片和每个商品的价格。	Josh, Rick
DF-43	P2	商店页面必须在地图上标明有线下店铺的位置。	Josh, Candice
DF-44	P2	客户可以通过商店页面订阅商店。	Steven

前两项与接下来要做的特性有关，所以它们描述得更详细。不过，谁知道呢，也许在下一个 sprint 之前，关于订阅的要求会取消，所以考虑其细节信息也没有意义。

有些情况下，仍然需要拥有一个完整的需求列表。如果需要与外部监管机构或内部团队（如审计、法律或合规）打交道，可能需要提供编写良好的文档。有时，只给他们一个包含从 backlog 中提取的工作项文档就可以了。最好与此类负责人沟通清楚：收集其期望，以了解满足需求，以便文档最小化。

编写需求文档的重要之处在于，和提出特定需求的各方之间要互相了解。如何实现这一点？当准备好了草稿，就应该向他们展示文档并收集反馈。这样，就会知道什么是模糊的，不清楚的，或缺失的。即使需要一些迭代，也将帮助与众负责人建立一个共识，从而获得更多的信心，并相信正在构建正确的东西。

3.5.7 其他部分

有一个链接和资源部分是一个好主意，在其中你可以指向诸如问题跟踪板、工件、CI、代码库，以及可以陈现的东西。架构、营销和其他类型的文档也可以在这里列出。

如果需要，还可以包含词汇表。

了解了如何记录需求和相关信息。接下来，就简单介绍一下如何为设计的系统编写文档。

3.6. 文档化架构

正如记录需求一样，也应该记录出现的架构。这当然不仅仅是为了拥有文档，它应该帮助每个参与项目的人提高生产力，使他们更好地理解自己的角色和最终产品需要什么。并不是所有的图表都对每个人都有用，但是可以试着从未来读者的角度来创建它们。

有许多框架可以记录愿景，其中许多框架服务于特定的领域、项目类型或体系结构范围。例如，如果对记录企业架构感兴趣，那么可能会对 TOGAF 感兴趣。这是开放组架构框架的首字母缩写。它依赖于四个领域，即：

- 业务架构（策略、组织、关键流程和治理）
- 数据架构（逻辑和物理数据管理）
- 应用架构（单系统的蓝图）
- 技术架构（硬件、软件和网络基础设施）

如果将软件记录放在整个公司，甚至更广泛的范围内，那么这种分组就很重要。其他类似规模的框架包括由 **英国国防部 (MODAF)** 和 **美国等效机构 (DoDAF)** 开发的框架。

如果没有记录企业架构，若是刚刚开始架构自我开发道路，可能会对其他框架更感兴趣，例如 4+1 和 C4 模型。

3.6.1 4+1 模型

4+1 模型是由 Philippe Kruchten 在 1995 年创建，作者称它旨在“对多个并发视图的使用，描述软件密集型系统的架构”。它的名字来自于它所包含的视图。

这种模型已经在市场上存在了很长时间，也非常有效果，因此广为人知。它非常适合于大项目，当然也可以用于中小型的项目，但对于需求来说也可能过于复杂（特别是用敏捷的方式编写的）。如果正处于这种情况，应该尝试下一节中的 C4 模型。

4+1 模型的缺点是它使用了一组固定的视图，而文档体系结构需要根据项目的具体情况选择视图（稍后将详细介绍）。

优点是可以清晰的了解视图是如何连接在一起，特别是涉及到场景时。同时，每个责任方可以很容易地了解与他们相关的部分。

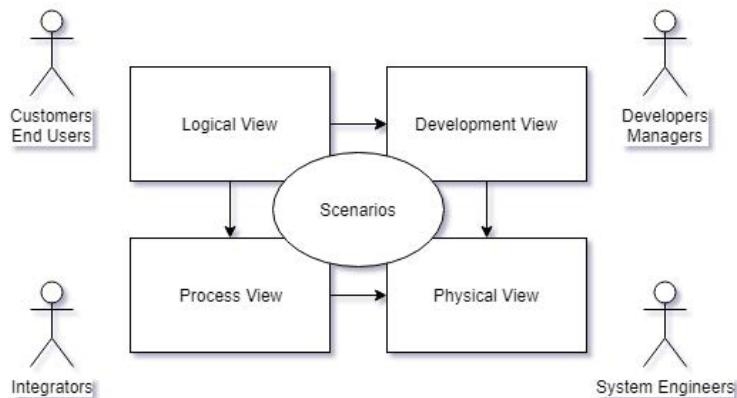


图 3.1 - 4+1 模式概览

上图中的参与者对其对应的视图最感兴趣，所有的视图都可以用不同种类的统一建模语言（UML）图来表示。先了解下每个视图：

- **逻辑视图**展示了如何向用户提供功能，以及系统的组件（对象）如何交互。最常见的是，由类和状态图组成。如果有成千上万的类或者只是想更好地显示交互，应该有通信图或序列图，两者都是下个视图的一部分：

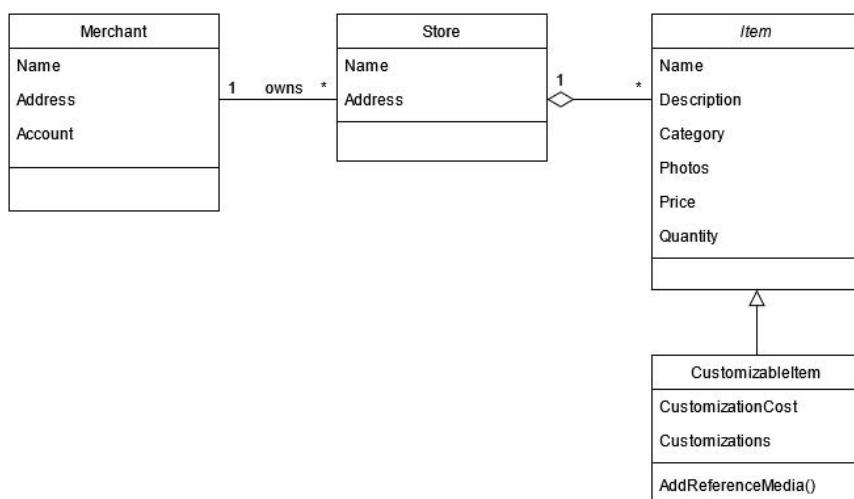


图 3.2 - 类图可以用来显示计划拥有的类型，以及它们的关系

- **进程视图**围绕着系统的运行时行为，显示进程、进程之间的通信以及与外部系统的交互，由活动和交互图表示。此视图处理许多 NFR，包括并发性、性能、可用性和可扩展性：

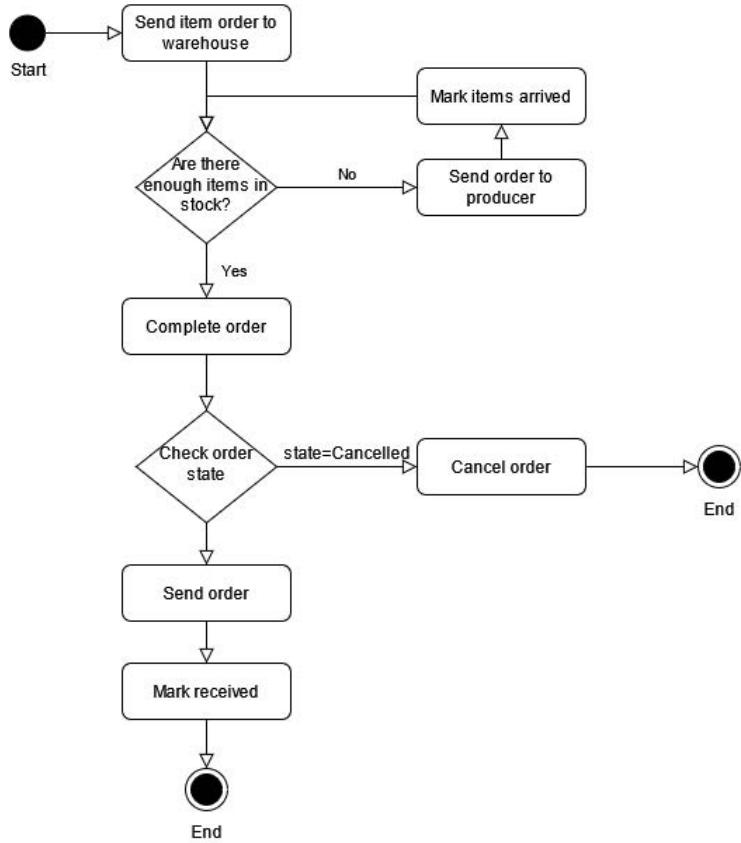


图 3.3 -活动图工作流和过程的图形表示

- **开发视图**为了分解成子系统，并围绕着软件组织。重用、工具约束、分层、模块化、打包、执行环境——这个视图可以通过显示系统的构建块来表示。通过使用组件和包图来实现：

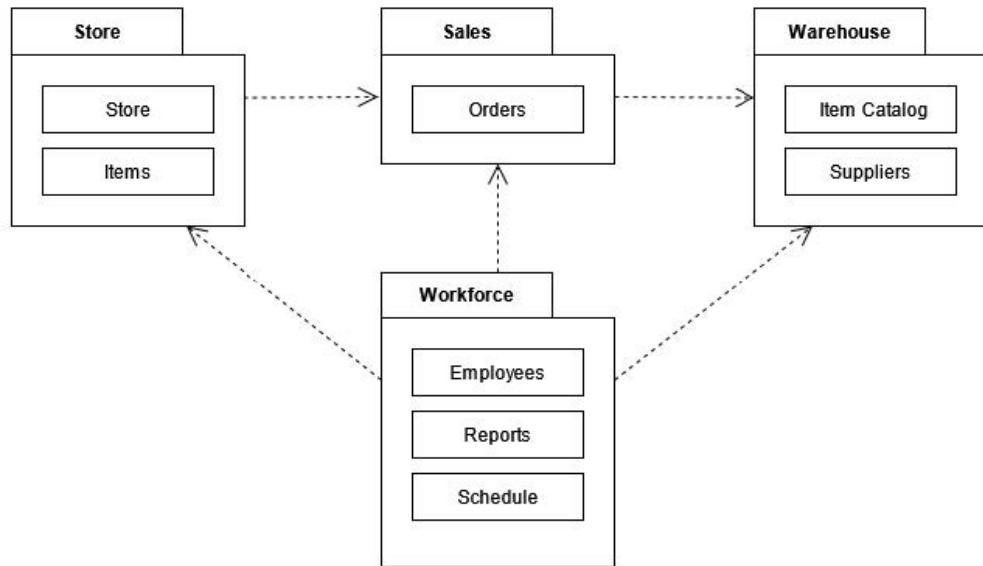


图 3.4 -包图可以从更高的角度显示系统的各个部分，以及特定组件之间的依赖关系

- **物理视图**使用部署图将软件映射到硬件。针对系统工程师，可以涵盖与硬件相关的 NFR 子集，例如通信：

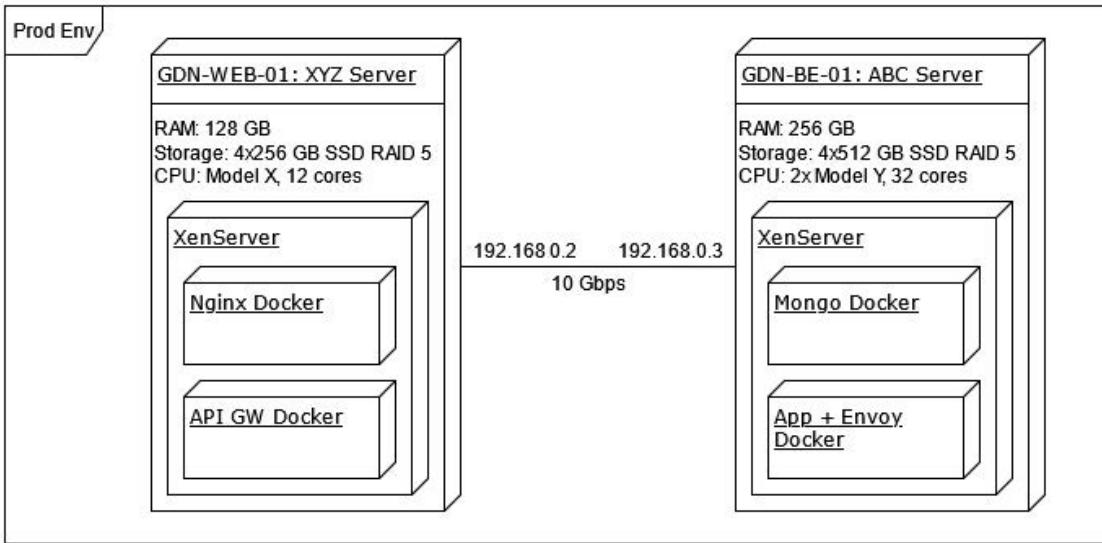


图 3.5 - 部署图展示了运行每个软件组件的硬件，还可以用来传递有关网络的信息

- **场景视图**把所有其他的视图都连接在一起，这些对所有相关方都是有用的。这个视图可以显示系统是否做了它应该做的事情。当其他视图都完成后，场景视图可能是多余的。但是，如果没有使用场景，其他视图都是不可能完成。这个视图从高层次展示了系统，而其他视图则展示了更多的细节：

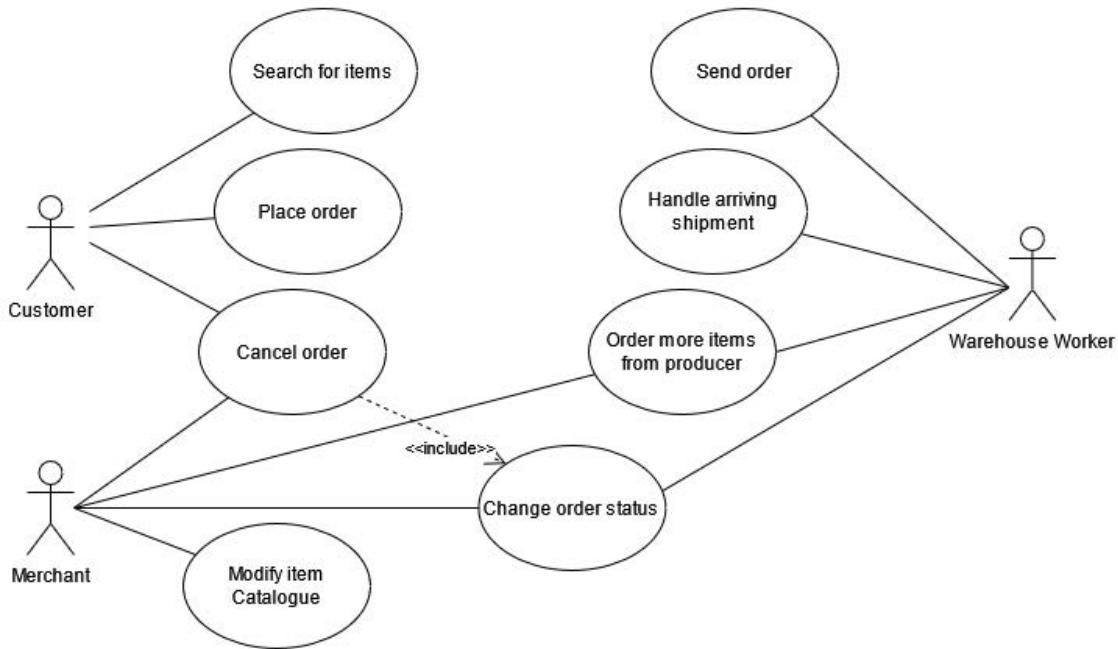


图 3.6 - 用例图显示了特定的参与者如何与系统交互，以及如何交互，彼此关联

这些视图中的每一个都与其他视图相互关联，才能显示出全貌。这里需要考虑下，如何表达并发。它不能只使用逻辑视图来完成，因为将其映射到任务和流程更有表现力，从而需要过程视图。另一方面，进程将映射到物理（通常是分布式的）节点。这意味着需要在三个视图中进行记录，每个视图都与特定的责任方相关。视图之间的其他连接包括：

- 逻辑视图和过程视图都用于分析和设计，以概念化产品。
- 将开发和部署放在一起，描述了软件是如何打包的，以及每个包将在何时部署。

- 逻辑视图和开发视图显示了功能如何反映在源码中。
- 流程和部署视图共同描述 NFR。

熟悉了 4+1 模型，继续了解另一个简单但非常有效的模型：C4 模型。希望它的使用过程充满了爆炸性（双关语 [译注：关联的是著名的 C4 炸弹]）。

3.6.2 C4 模型

C4 模型非常适合中小型项目。因为简单，所以易用，并且不依赖于任何预定义的符号。如果想用它来绘制图表，可以试试 Tobias Shochguertel 的 c4-draw.io 插件 (<https://github.com/tobiasshochguertel/c4-draw.io>) 的免费在线绘图工具 draw.io (<https://www.draw.io/>)。

C4 模型中，主要有四种图类型，即：

- 系统上下文
- 容器
- 组件
- 代码

就像使用地图放大和缩小一样，可以使用这四种类型来显示特定代码区域的细节，或者“缩小”来显示关于特定模块，甚至整个系统的交互和周围环境的更多信息。

系统上下文是查看架构的一个很好的起点，因为它将系统作为一个整体显示出来，周围是与其交互的用户和其他系统。可以在这里看到一个 C4 模型的上下文关系图：

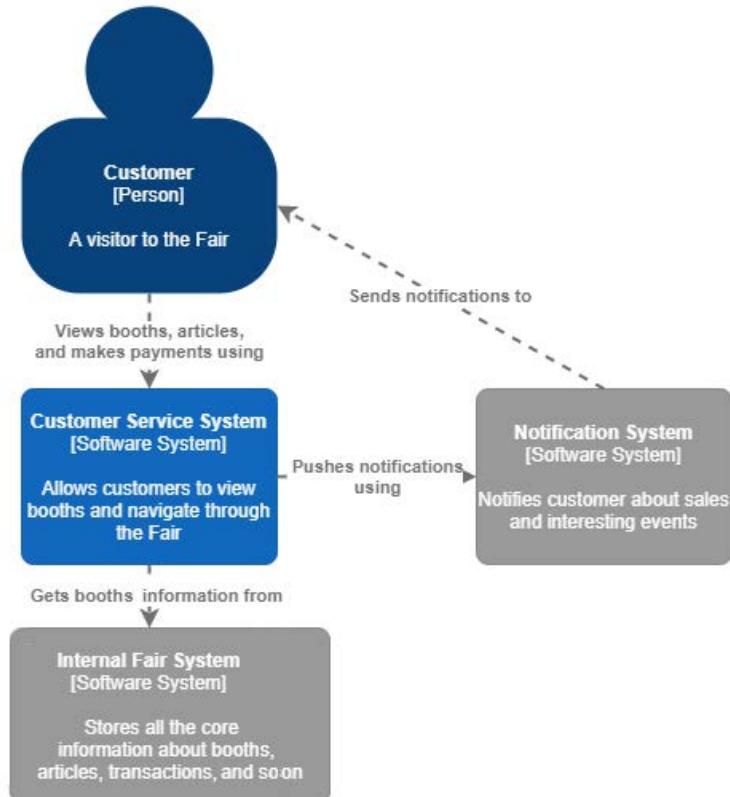


图 3.7 - C4 上下文关系图

它展示了“大局”，所以它不应该专注于特定的技术或协议。相反，可以把它看作是可以向非技术相关方展示的图表。仅通过查看图表，就可以清楚地看到有一个参与者（对客户的人形描述），他与解决方案的一个组件（即客户服务系统）交互。另一方面，这个系统与另外两个系统相互作用，每个相互作用都用箭头表示。

上下文关系图用于提供系统的概览，其中包含一些细节。现在看下其他图表：

- 容器图**：用来显示系统内部的概述。如果系统使用数据库，提供服务，或者只是由某些应用程序组成，则此图将显示它。还可以展示容器的主要技术选择。注意，容器不是指 Docker 容器，尽管每个图都是单独的可运行和部署单元，但此图类型与部署场景无关。容器视图是为技术人员设计的，但并不仅限于开发团队。架构师以及操作和支持人员也是目标受众。
- 组件图**：如果想要关于特定容器的更多细节信息，就需要组件图发挥作用了。它展示了所选容器内的组件如何彼此交互，以及如何与容器外的元素和参与者交互。通过查看这张图，可以了解每个组件的职责，以及使用什么技术构建。组件图的目标受众主要集中在特定的容器上，由开发团队和架构师组成。
- 代码图**：最后来看代码图，将组件放大到一定程度时，代码图就会出现。这个视图主要由 UML 图组成，包括类、实体关系和其他，理想情况下应该由独立的工具和 IDE 从源代码自动创建。不应该为系统中的每个组件制作这样的关系图；相反，把重点放在最重要的内容上，让它们告诉读者架构想要表达的内容。想要这样的图中少一些，就要从代码图中删掉不必要的元素。在许多系统中，特别是在较小的系统中，这类图会直接省略。

细心地读者可能会发现 C4 模型缺少一些特定的视图。例如，想知道如何演示应该如何部署系统，可能会有兴趣了解除了主图之外，还有一些补充图。其中之一是部署图，展示了如何将系统中的容器映射到基础设施中的节点。通常，其是 UML 部署图的简化版本：

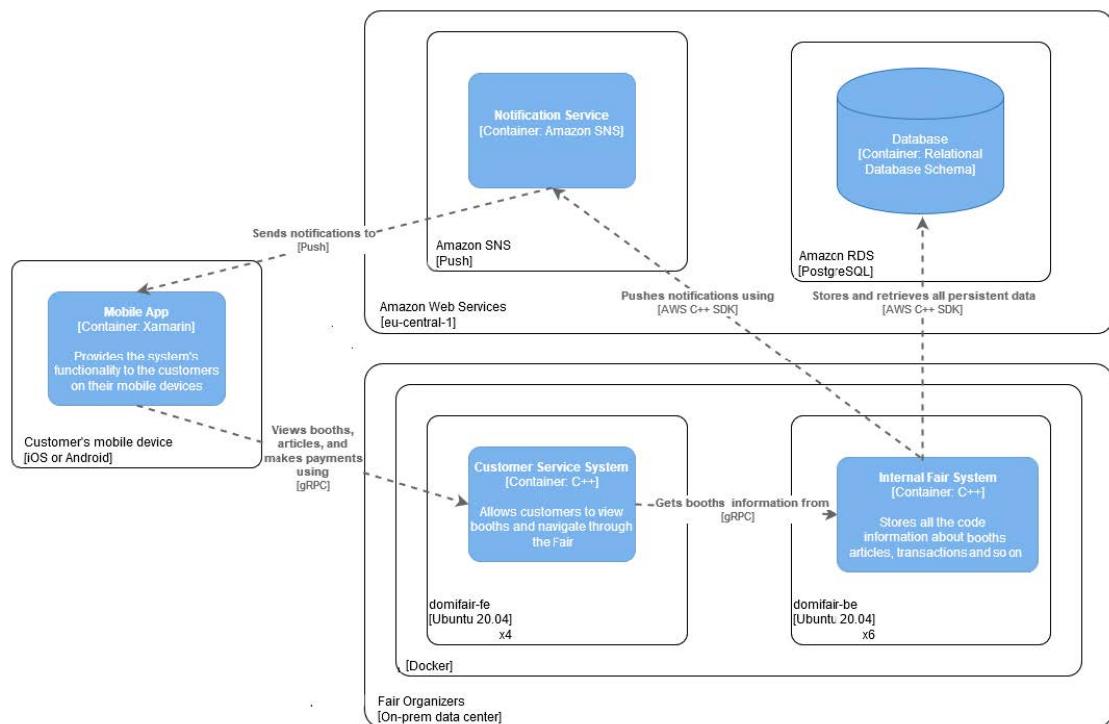


图 3.8 - C4 部署图

说到与 C4 模型相关的 UML 图，可能还要知道为什么它在展示系统的用例上投入的这么少。如果遇到了这种情况，应该考虑使用 UML 的用例图来补充前面的模型，或者可能考虑引入一些序列图。

记录架构时，记录的内容和共享的知识比遵循硬性规则更重要，可以选择最适合的工具。

3.6.3 记录敏捷项目中的架构

敏捷环境中，记录架构的方法应该与记录需求的方法类似。首先，考虑谁会阅读这些材料，确保以正确的方式描述了正确的事情。文档不需要冗长的 Word 文档，当有人描述架构时，可以使用演示文稿、Wiki 页面、单个图表，甚至是会议的录音进行记录。

重要的是收集关于文档化架构的反馈。与文档化的需求一样，需要告知相关方文档的更新，以便大家了解都改进了哪里。尽管这看起来像是在浪费时间，但如果处理得当，会节省交付产品的时间。足够好的文档可以帮助新手快速上手工作，并指导不熟悉框架的责任人。如果只是在一些会议上讨论架构，那么很可能会议之后就没有人会记得出于什么原因才做出的那些决策，以及这些决策在不断变化的敏捷环境中是否仍然有效。

创建文档时，重复阅读很重要，因为很可能会对一两个重要的细节有一些误解。架构师或相关方负责人会了解更多的细节，从而对原有的决定进行更改。在文件认为是成熟和完成之前，至少要把准备好的文档多看几遍。通常，通过即时通讯工具、电话或面对面的交谈可以更快地完成任务，并解决可能出现的后续问题，所以比起电子邮件或其他异步的沟通方式，架构师可能会更喜欢这种方式。

3.7. 选择合适的视图进行记录

架构是一个过于复杂的主题，无法用一个大图表来描述。假设一幢建筑的建筑师，为了设计整个建筑，需要不同视角的独立图表：一个用于设计管道，另一个用于部署电力和其他电缆等。每个图都显示了项目的不同方面。软件架构也是如此：需要从不同的角度，针对不同的受众来展示软件。

此外，如果正在建造一座智能楼宇，很有可能需要规划出想要放置的设备的计划。尽管不是所有的项目都需要，但在当前的项目中需要，所以需要添加。同样的方法也适用于架构：如果发现不同的视图对文档有价值，那么就应该这样做。那么，如何知道哪些视图可能是有价值的呢？可以尝试以下步骤：

1. 从 4+1 模型或 C4 模型的视图开始。
2. 询问相关方责任人，文档化对他们来说是否必要、哪些是必要的，进而考虑修改视图集。
3. 选择可以评估架构是否满足其目标，以及是否满足所有 ASR 视图。阅读下一节中每个视图的第一段，确定是否需要。

如果仍不确定要记录哪些视图，这里有一组提示：

TIP

尝试只选择最重要的视图，因为当它们太多时，架构将变得难以理解。好的视图不仅应该展示架构，还应该向项目暴露技术风险。

选择应该在文档中描述哪些视图时，需要考虑在这里简要地描述它们。如果对这个话题感兴趣，可以在扩展阅读部分中找到 *Rozanski* 和 *Woods* 的书进行阅读。

3.7.1 功能视图

若软件是作为更大的系统的一部分开发，特别是与不进行日常交流的团队一起开发，那么就应该包括一个功能视图（就像在 4+1 模型中那样）。

在为架构编写文档时，接口的定义是最需要描述的事情，也是经常忽视的。无论是两个组件之间的接口，还是外部世界的入口点，都应该花时间进行完备地记录，描述对象和调用的语义，以及使用示例（有时可以作为测试重用）。

文档中包含功能视图的另一个好处是，阐明了系统组件之间的职责。每个开发系统的团队都应该了解边界在哪里，谁负责开发哪些功能。所有需求都应该显式地映射到组件，以消除差距和重复的工作。

TIP

这里需要注意的事情是避免重载函数视图。如果弄得一团糟，就没人想看了。如果开始仅在视图上描述基础设施，那么可以添加一个部署视图。如果在模型中有一个超大对象，试着重新思考设计，把它分成更小的，更内聚的组件。

关于功能视图的最后一个注意事项：尽量将包含的每个图维持在同一个抽象级别上。另一方面，不要选择过于抽象的层次而使图过于模糊，进而确保相关方对每个元素都能正确的定义和理解。

3.7.2 信息视图

如果系统对信息、处理流、管理过程或存储有非直接的需求，那么需要包含这种视图。

选取最重要的、数据丰富的实体，展示如何在系统中流动，谁拥有、谁是生产者、谁是消费者。标记某些数据的有效期，以及何时可以安全地丢弃，到达系统的特定点的预期延迟是什么，或者如果系统工作在分布式环境中，如何处理标识符。若系统管理事务，则相关方责任人也应该清楚这个过程以及任何回滚。转换、发送和持久化数据的技术也很重要。如果从事金融领域的业务或必须处理个人数据，很可能需要遵守一些规定，因此请描述系统计划如何解决这一问题。

数据结构可以用 UML 类模型来表示。要清楚数据的格式，特别是在两个不同的系统之间交互时。NASA 与 Lockheed Martin 公司共同开发的火星气候轨道器，因为他们（在不知情的情况下）使用了不同的计数单位，从而导致 NASA 失去了价值 1.25 亿美元的火星气候轨道器，所以一定要注意系统之间的数据不一致性。

数据的处理流程可以使用 UML 的活动模型，并且可以使用状态图来显示信息的生命周期。

3.7.3 并发视图

如果运行许多并发执行单元对产品很重要，请考虑添加并发视图，可以显示可能遇到的问题和瓶颈（除非太详细）。使用它的其他原因是依赖于进程间通信、具有非直接的任务结构、并发状态管理、同步或任务失败处理逻辑。

可以为这个视图使用想要的表示，只要可以捕获执行单元及其通信。可以为进程和线程分配优先级，然后分析潜在的问题，例如死锁或争用。可以使用状态图来显示可能的状态以及重要执行单

元(等待查询、执行查询、分发结果等)的转换。

如果不确定是否需要将并发性引入系统，好的经验法则是不要做。如果必须的话，尽量设计得简单一些。调试并发性问题从来都不是一件容易的事情，而且时间花费总是很长。因此，可以首先尝试处理现有的问题，而不是抛出更多的线程来处理手头的问题。

查阅图后，若担心资源争用，可以尝试用更多的锁替换大对象上的单锁，这样的粒度会更细，使用轻量级同步(有时原子就足够了)，引入乐观锁，或者减少共享的内容(在线程中创建独立的数据副本，并对其进行处理会比共享访问更快)。

3.7.4 发展视图

如果正在构建一个包含许多模块的大型系统，并且需要结构化代码，拥有系统范围的设计约束，或者如果想在系统的各个部分之间共享，从开发的角度展示解决方案应该会使架构师，软件开发人员和测试人员更了解这个系统。

开发视图的包图可以明确地展示系统中不同的模块位于何处，依赖关系是什么，以及其他相关的模块(例如，驻留在同一个软件层)。不需要是UML图——即使是只有框和线也可以。如果计划一个模块是可替换的，这种图表可以显示出有哪些软件包可能会受到影响。

增加系统重用的策略，例如：为组件创建运行时框架，或者增加系统一致性策略。认证、日志、国际化或其他类型的处理的通用方法，都是开发视图的一部分。可以将系统可见的公共部分记录下来，以确保所有开发人员也能看到。

通用的代码组织、构建和配置管理方法，也应该出现在文档中。如果所有这些听起来都需要记录，那么就专注于最重要的部分，并简要介绍其余部分。

3.7.5 部署和操作视图

如果有一个非标准的或复杂的部署环境，例如关于硬件、第三方软件或网络需求的特定需求，针对系统管理员、开发人员和测试人员，可以考虑将其记录在一个单独的部署部分中。

如有必要，应包括下列内容：

- 所需内存
- CPU线程数(有或没有超线程)
- 关于NUMA节点的亲和性
- 专业网络设备的要求，如交换机标记
- 以黑盒方式度量延迟和吞吐量的包
- 网络拓扑结构
- 估计所需带宽
- 应用程序的存储要求
- 第三方软件

有了需求，就可以将它们映射到特定的硬件，并将它们放入运行时平台模型中。如果想要建模，可以使用带有构造型的UML部署图。这应该显示处理节点和客户机节点、在线和离线存储、网络链接、专门的硬件(如防火墙、FPGA或ASIC设备)，以及功能元素和运行节点之间的映射。

如果有非直接的网络需求，可以添加另一个图来显示网络节点和它们之间的连接。

如果依赖于特定的技术(包括软件的特定版本),最好将它们列出来,以查看所使用的软件是否存在兼容性问题。有时,两个第三方组件需要相同的依赖关系,但版本不同。

如果头脑中有一个特定的安装和升级计划,写几句话来说明可能是个好主意。解决方案所依赖的A/B测试、蓝绿部署或特定容器的使用技巧,相关人员都很应该很清楚。如果需要,还应该包括数据迁移计划,包括迁移需要多长时间,以及何时可以安排迁移。

配置管理、性能监视、操作监视和控制,以及备份策略的计划都值得描述。可能需要创建几个组,确定每个组的依赖关系,并为每个组定义方法。如果能够考虑到可能发生的错误,就制定一个计划来检测,并尝试从错误中恢复。

对支持团队的注意事项也可以进入这个部分:哪个相关方需要什么支持,计划有什么类别的事件,如何升级,以及每个级别的支持将负责什么。

最好尽早与一线人员接触,并为他们专门创建图表,以保持他们的参与度。

已经讨论了如何手动创建关于系统及其需求的文档,接下来切换到以自动化的方式为API编写文档。

3.8. 生成文档

工程师不喜欢体力劳动,某件事可以自动化,就可能会自动化。通过所有这些努力来创建足够好的文档,要是能实现部分工作的自动化实际上也是一种福报。

3.8.1 生成需求文档

如果从零开始创建一个项目,可能很难凭空生成文档。若相应的工具中有需求,那么还可以生成文档。例如,正在使用JIRA,起点应该是从问题导航器视图中导出所有项。可以使用过滤器,并只获相应项目的打印输出。如果不喜欢单一的字段集,或者觉得不是想要的,可以尝试JIRA的需求管理插件,可以导出需求,例如:**R4J(Requirements for Jira)**允许创建需求的整个层次结构,并进行跟踪,管理更改,并在整个项目中进行传播,执行需求更改的影响分析。当然,可以使用用户定义的模板导出。许多这样的工具也可以为需求创建测试套件,但目前没有免费的版本。

3.8.2 用代码中生成图表

如果想在不深入源代码的情况下了解代码结构,可能会对代码生成图表的工具感兴趣。

CppClassDepend就是这样一个工具,能够在不同的源码间创建各种依赖关系图。更重要的是,允许通过各种参数查询和过滤代码。无论只是想了解代码的结构、发现不同软件组件之间的依赖关系,以及它们耦合的紧密程度,还是想快速定位技术负担最重的部分,这个工具都能有所帮助。这个软件需要花钱,但提供试用版本。

有些绘图工具可以从类图创建代码,从代码创建类图。企业版能够获得类图和接口图,并以多种语言生成代码。C++就是其中之一,可以直接从源代码生成UML类图。另一个可以完成这项任务的工具是Visual Paradigm。

3.8.3 用代码生成 (API) 文档

为了帮助其他人浏览现有代码，并使用提供的 API，好的方法是提供从代码中的注释生成的文档。这类文档最好就放在所描述的函数和数据类型旁边，这有助于保持同步。

编写此类文档的标准工具是 Doxygen，优点是速度快（特别是对于大型项目和 HTML 文档生成），生成器有一些内置的正确性检查（例如，对于函数中部分记录的参数—检查文档是否是最新版本，可以有一个很好的标记），并且它允许对类和文件层次结构进行导航。缺点包括不能进行全文搜索，生成的 PDF 不够理想，界面操作可能有点麻烦。

幸运的是，可用性缺陷可以通过使用另一个文档工具来弥补。如果曾经阅读过 Python 文档，那么可能遇到过 Sphinx。它的外观新颖、可用比较好的界面，并使用 reStructuredText 作为标记语言。好消息是，在这两者之间有一个桥梁，因此可以使用 Doxygen 生成的 XML 并在 Sphinx 中使用它。这个桥接软件叫做 Breathe。

现在看看如何在项目中进行设置。假设将源代码保存在 *src* 中，公共头文件保存在 *include* 中，文档保存在 *doc* 中。首先，创建一个 *CMakeLists.txt* 文件：

```
cmake_minimum_required(VERSION 3.10)

project("Breathe Demo" VERSION 0.0.1 LANGUAGES CXX)

list(APPEND CMAKE_MODULE_PATH "${CMAKE_CURRENT_LIST_DIR}/cmake")
add_subdirectory(src)
add_subdirectory(doc)
```

对项目支持的 CMake 版本设置了要求，指定了名称、版本和使用的语言（在例子中，只有 C++），并将 *CMake* 目录添加到 CMake 查找包含文件的路径中。

cmake 子目录中，创建一个文件 *FindSphin.cmake*，这里直接使用它的名字，因为 Sphinx 还没有提供相应的 *cmake* 文件：

```
find_program(
    SPHINX_EXECUTABLE
    NAMES sphinx-build
    DOC "Path to sphinx-build executable")

# handle REQUIRED and QUIET arguments, set SPHINX_FOUND variable
include(FindPackageHandleStandardArgs)
find_package_handle_standard_args(
    Sphinx "Unable to locate sphinx-build executable" SPHINX_EXECUTABLE)
```

现在，CMake 将查找 Sphinx 构建工具。如果找到，将设置适当的 CMake 变量，将 Sphinx 包标记为已找到。接下来，创建源文件来生成文档。创建一个 *include/breathe_demo/demo.h* 文件：

```
1 #pragma once
2
3 // the @file annotation is needed for Doxygen to document the free
```

```

4 // functions in this file
5 /**
6 * @file
7 * @brief The main entry points of our demo
8 */
9
10 /**
11 * A unit of performable work
12 */
13 struct Payload {
14     /**
15      * The actual amount of work to perform
16      */
17     int amount;
18 };
19
20 /**
21  * @brief Performs really important work
22  * @param payload the descriptor of work to be performed
23 */
24 void perform_work(struct Payload payload);

```

注意注释语法。Doxygen 在解析头文件时识别它，以便知道在生成的文档中放入什么。

现在，为头文件添加一个相应的 *src/demo.cpp* 实现：

```

1 #include "breathe_demo/demo.h"
2
3 #include <chrono>
4 #include <thread>
5
6 void perform_work(Payload payload) {
7     std::this_thread::sleep_for(std::chrono::seconds(payload.amount));
8 }

```

这里没有 Doxygen 的注释。更倾向于在头文件中记录类型和函数，因为它们是库接口。源文件只是实现，它们没有向接口添加新内容。

除了前面的文件，还需要在 *src* 中放一个简单的 *CMakeLists.txt*:

```

add_library(BreatheDemo demo.cpp)
target_include_directories(BreatheDemo PUBLIC
    ${PROJECT_SOURCE_DIR}/include)
target_compile_features(BreatheDemo PUBLIC cxx_std_11)

```

这里，指定目标的源文件、包含目标头文件的目录，以及编译所需的 C++ 标准。

现在，移动到 *doc* 文件夹。*CMakeLists.txt* 文件首先检查 Doxygen 是否可用，如果可用则省略生成：

```

find_package(Doxygen)
if (NOT DOXYGEN_FOUND)

```

```
    return()
endif()
```

如果没有安装 Doxygen，将跳过文档生成。还要注意 `return()` 会退出当前的 CMake 文件，这是一个并不广为人知，但有用的技巧。

接下来，假设发现了 Doxygen，需要设置一些变量来控制生成。这里只想要 Breathe 的 XML 输出，所以需要设置以下变量：

```
set(DOXYGEN_GENERATE_HTML NO)
set(DOXYGEN_GENERATE_XML YES)
```

要强制使用相对路径，可以使用 `set(DOXYGEN_STRIP_FROM_PATH ${PROJECT_SOURCE_DIR}/include)`。如果实现细节需要隐藏，可以使用 `set(DOXYGEN_EXCLUDE_PATTERNS "*/detail/*")` 来实现。已经设置好了所有的变量，现在进行生成：

```
# Note: Use doxygen_add_docs(doxxygen-doc ALL ...) if you want your
# documentation to be created by default each time you build. Without the
# keyword you need to explicitly invoke building of the 'doc' target.
doxygen_add_docs(doxxygen-doc ${PROJECT_SOURCE_DIR}/include COMMENT
  "Generating API documentation with Doxygen")
```

这里，调用专门为使用 Doxygen 编写的 CMake 函数。定义了一个目标，`doxygen-doc`，需要显式地调用它，按需生成文档，就像注释所述。

现在创建一个 Breathe 目标来使用 Doxygen。可以使用 `FindSphinx` 模块来完成这个任务：

```
find_package(Sphinx REQUIRED)
configure_file(${CMAKE_CURRENT_SOURCE_DIR}/conf.py.in
  ${CMAKE_CURRENT_BINARY_DIR}/conf.py @ONLY)
add_custom_target(
  sphinx-doc ALL
  COMMAND ${SPHINX_EXECUTABLE} -b html -c ${CMAKE_CURRENT_BINARY_DIR}
  ${CMAKE_CURRENT_SOURCE_DIR} ${CMAKE_CURRENT_BINARY_DIR}
  WORKING_DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}
  COMMENT "Generating API documentation with Sphinx"
  VERBATIM)
```

首先，调用模块。然后，用项目中的变量填充一个 Python 配置文件，以便 Sphinx 使用。再创建了一个 `sphinx-doc` 目标，它将生成 HTML 文件作为输出，并在这样做时在构建输出中打印一行。

最后，强制 CMake 在每次生成 Sphinx 文档时调用 `Doxygen:add_dependencies(sphinx-doc doxygen-doc)`。

如果希望为文档提供更多的目标，引入一些 CMake 函数可能会有用，这些函数将处理与文档相关的目标。

现在来看看 `conf.py.in` 文件中，用来引导的工具。创建它，将其指向 Sphinx：

```
extensions = [ "breathe", "m2r2" ]
breathe_projects = { "BreatheDemo": "@CMAKE_CURRENT_BINARY_DIR@/xml" }
breathe_default_project = "BreatheDemo"

project = "Breathe Demo"
author = "Breathe Demo Authors"
copyright = "2021, Breathe Demo Authors"
version = "@PROJECT_VERSION@"
release = "@PROJECT_VERSION@"

html_theme = 'sphinx_rtd_theme'
```

从前面的清单中可以看到，设置了 Sphinx 要使用的扩展、文档记录项目的名称和其他一些相关变量。注意`@NOTATION@`，由 CMake 使用适当的变量值填充输出文件。最后，告诉 Sphinx 使用 ReadTheDocs 主题 (`Sphinx_rtd_theme`)。

困难的最后一部分是 reStructuredText 文件，它定义了在文档的什么地方包含什么。首先，创建一个 `index.rst` 文件，包含一个目录和一些链接：

```
Breathe Demo
=====
Welcome to the Breathe Demo documentation!
.. toctree::
   :maxdepth: 2
   :caption: Contents:

Introduction <self>
   readme
   api_reference
```

第一个链接指向这个页面，并且可以从其他页面返回到它。显示 `Introduction` 作为标签，其他名称指向带有`.rst` 扩展。因为包含了 M2R2 Sphinx 扩展，所以可以在文档中包含 `README.md` 文件，这样可以避免自述文件内容的重复，其实 `readme.rst` 文件就是`..mdininclude:: ../README.md`。现在进行最后一步：合并 Doxygen 的输出，使用下面的命令在 `api_reference.rst` 中完成：

```
API Reference
=====
.. doxygenindex::
```

因此，目前只是按自己喜欢的方式命名了参考页面，并指定 Doxygen 生成的文档应该列在这

里，就是这样！只要构建 `sphinx-doc` 目标，会得到一个这样的页面：

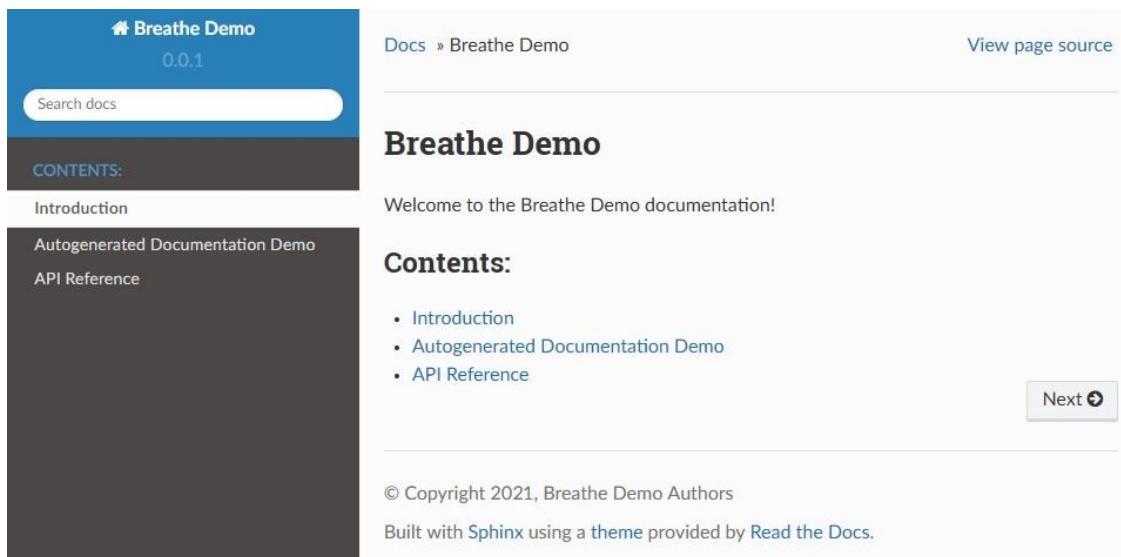


图 3.9 - 文档的主页，合并了生成的和手工编写的部分

查看 API 文档页面时，应该是这样：

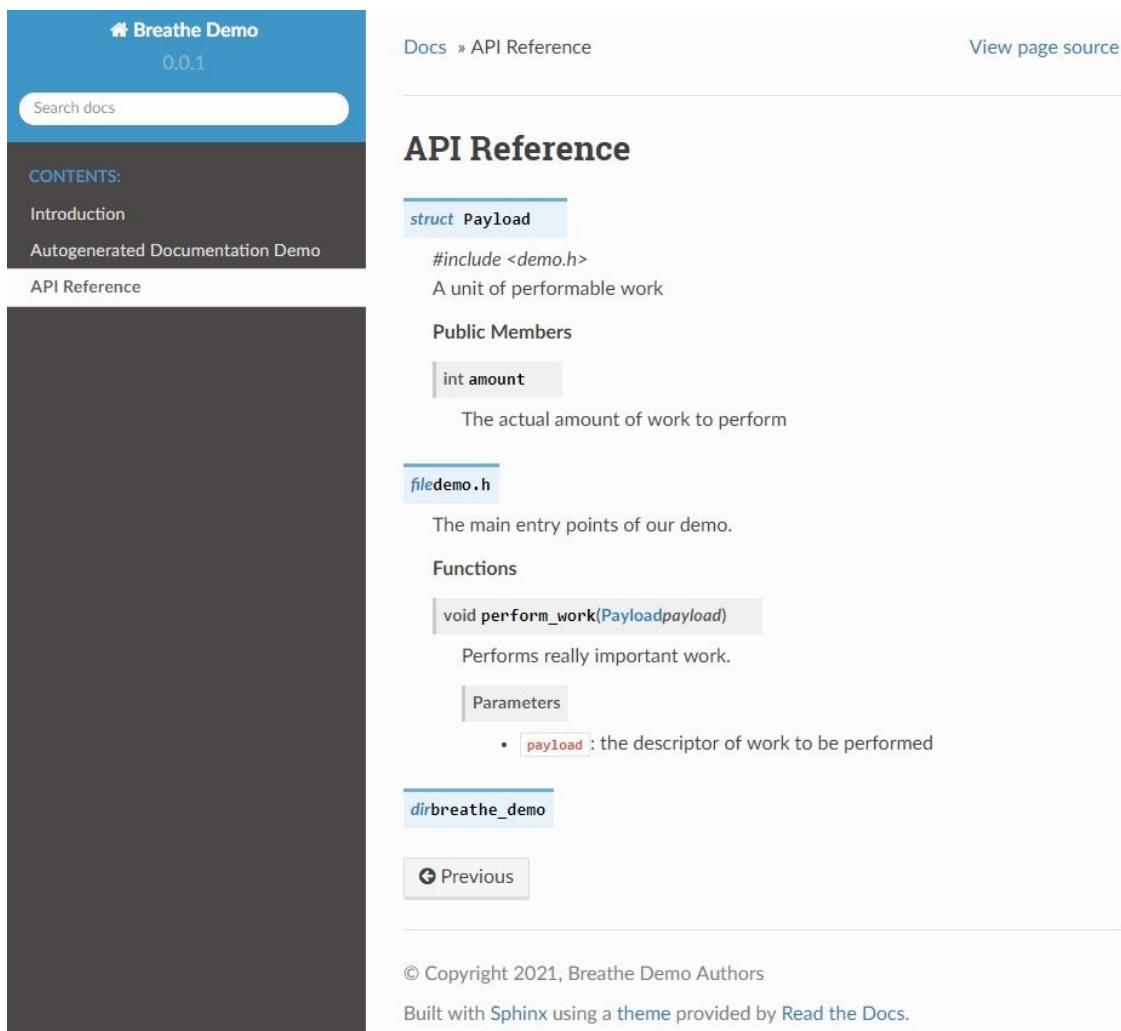


图 3.10 - 自动生成的 API 文档

可以看到，文档是为我们的 *Payload* 类型自动生成的，它的每个成员，以及 *perform_work* 函数，包括它的每个参数，都会根据定义它们的文件进行分组。就是整洁！

3.9. 总结

本章中，了解关于需求和文档的所有要素。学习了如何成功地收集需求，以及如何识别最重要的需求。现在，可以准备精简而有用的文档，以面向视图的方式显示哪些内容是重要的。能够区分不同类型和风格的图表，并使用最适合需求的图表。最后，能够自动生成漂亮的文档。

下一章中，将学习有用的体系结构设计模式，它将帮助您满足系统的需求。我们将讨论各种模式，以及如何应用它们来提供许多重要的质量属性，这两种模式都是在分布式系统的单组件范围内进行。

3.10. 练习题

1. 什么是质量属性？
2. 收集需求时应该使用哪些资源？
3. 如何判断某个需求在架构上是否重要？
4. 应该如何以图形化的方式记录各方对系统的功能性需求？
5. 何时开发视图文档？
6. 如何自动检查代码的 API 文档是否过期？
7. 如何在图中显示，特定的操作由系统的不同组件处理？

3.11. 扩展阅读

- Evaluate the Software Architecture using ATAM, JC Olamendy, blog post:
<https://johnolamendy.wordpress.com/2011/08/12/evaluate-the-software-architecture-using-atam/>
- **EARS:** The Easy Approach to Requirements Syntax, John Terzakis, Intel Corporation, conference talk from the ICCGI conference: https://www.aria.org/conferences2013/filesICCGI13/ICCGI_2013_Tutorial_Terzakis.pdf
- Eoin Woods and Nick Rozanski, *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*

第二部分：C++软件的设计与开发

介绍使用 C++ 创建有效的软件解决方案的技术。演示了在设计、开发和构建 C++ 代码时解决常见挑战和避免陷阱的技术。这些技术来自于 C++ 语言本身，以及设计模式、工具和构建系统。

包括以下几章：

- 第 4 章，架构与系统设计
- 第 5 章，C++ 特性
- 第 6 章，设计模式和 C++
- 第 7 章，构建和打包

第 4 章 架构与系统设计

模式有助于处理复杂的东西。在单组件的级别上，可以使用软件模式，比如著名的[设计模式](#)：可复用面向对象软件的基础所描述的模式。可以向更高的方向发展，了解不同组件之间的架构，了解何时以及如何应用架构模式将大有裨益。

有无数这样的模式，适用于不同的场景。要想了解所有，需要阅读可不止一本书，本书只选择了几个适合于实现架构目标的模式。

这章中，将介绍一些与架构设计相关的概念和谬误。将展示何时使用上述模式，以及如何设计易于部署的高质量组件。

本章将讨论以下内容：

- 不同的服务模型
- 避免分布式计算的错误
- CAP 定理的结果和一致性
- 使系统具有容错性和可用性
- 集成系统
- 分级性能
- 部署系统
- 管理 API

本章结束时，将了解如何设计架构中的几个重要的特性，例如容错性、可扩展性和可部署性。在此之前，首先了解分布式架构的两个方面。

4.1. 相关准备

本章的代码需要以下工具来构建和运行：

- Docker
- Docker Compose

本章的代码可以在以下 GitHub 页面找到：<https://github.com/PacktPublishing/Software-Architecture-with-Cpp/tree/master/Chapter04>。

4.2. 分布式系统的特性

有许多不同类型的软件系统，每一种都适合于不同的场景，为不同的需求构建，并使用不同的假设集。编写和部署一个经典的、独立的桌面应用程序与编写和部署一个需要通过网络与许多其他应用程序通信的微服务完全不同。

这一节中，将介绍可以用来部署软件的各种模型，以及在创建分布式系统时会遇到的常见错误，以及为了创建这样的系统需要做出的一些妥协。

4.2.1 不同的服务模型

从服务模型开始。当设计一个更大的系统时，需要决定管理多少基础设施，以及可以在现有的构建块上构建多少基础设施。有时，可能希望利用现有的软件，而不需要手动部署应用程序或备份数据，例如通过其 API 使用 Google Drive 作为应用程序的存储。也可以依赖现有的云平台，如 Google 的 App Engine 来部署解决方案，而无需担心提供语言运行时或数据库。如果想要以自己的方式部署一切，那么可以利用来自云提供商的基础设施，也可以使用商业的基础设施。

先来了解一下不同的模型，以及它们各自的用处。

本地 (on-premises) 模型

经典的方法，也是在前云时代可用的唯一方法。在一切自己部署的前提下，需要购买所需的所有硬件和软件，并确保能提供足够的容量。如果是一家初创公司工作，这可能是一笔很大的前期成本。随着用户基础的增长，需要购置更多的资源，以便服务能够处理偶尔出现的负载高峰。所有这些都需要预测解决方案的增长，并主动采取行动，因为这时不可能根据当前的负载继续进行扩展。

即使在云时代，本地部署仍然很有用，而且经常在云外使用。有时，由于数据隐私问题或硬性规定，有些数据不能处理，甚至不能带离公司。其他时候，需要尽可能少的延迟，并且需要有自己的数据中心来进行实现。有时，可能需要计算成本，并且需要确定本地部署比云解决方案的开销更低。最后，公司也可能已经有了可以使用的数据中心。

本地部署并不需要完整的系统。通常，公司有自己的私有云部署在内部。可以通过利用可用的基础设施来降低成本。还可以将私有云解决方案与其他服务模型混合使用，这在不时需要额外容量时非常有用，这也称为**混合部署**，所有主流云提供商，以及 OpenStack 的 Omni 项目都会提供这样的服务。

基础设施即服务 (IaaS) 模型

谈到其他模型，最基本的云服务模型称为**基础设施即服务 (IaaS)**。它也是最类似于内部部署的：可以将 IaaS 视为拥有虚拟数据中心的一种方式。顾名思义，云提供商提供他们托管的基础设施，其中包含三种类型的资源：

- 计算。例如虚拟机、容器或裸机（不包括操作系统）
- 网络。除了网络本身，还包括 DNS 服务器、路由和防火墙
- 存储。包括备份和恢复能力

提供所有软件仍然取决于：操作系统、中间件和应用程序。

IaaS 可以用于各种场景，从托管网站（可能比传统的网站托管更便宜），通过存储（例如，Amazon 的 S3 和冰川服务），到高性能计算和大数据分析（需要巨大的计算能力）。一些公司使用它在需要时，可以对开发环境进行快速设置和清除测试。

使用 IaaS（而不是本地基础设施）是测试的一种低开销方案，同时可以节省配置所需的时间。

如果服务观察到使用的高峰期（例如在周末），可能希望利用云的自动扩展能力：在需要时向上扩展，过了高峰期后再向下扩展，从而节省成本。

所有云服务提供商都提供 IaaS 解决方案。

有时认为**容器即服务 (CaaS)** 是 IaaS 的子集。在 CaaS 中，服务不是提供裸机系统和虚拟机，而是提供容器和编排功能，可以构建自己的容器集群。CaaS 产品可以在 Google 云平台和 AWS 中

找到。

平台即服务 (PaaS) 模型

如果基础设施本身不能满足需求，可以使用**平台即服务 (PaaS)** 模型。这个模型中，云服务提供商不仅管理基础设施（就像在 IaaS 中一样），还管理操作系统、任何必需的中间件和运行时（可以部署软件的平台）。

通常，PaaS 解决方案将提供程序版本控制功能、服务监视和发现、数据库管理、商业智能，甚至开发工具。

使用 PaaS 的整个开发过程都会涉及：从构建和测试到部署、更新和管理服务。然而，PaaS 解决方案比 IaaS 成本更高。另一方面，通过提供整个平台，可以减少开发部分软件的成本和时间，并轻松地为分布在全球的开发团队提供相同的设置。

所有主流的云提供商都有自己的产品，例如 Google App Engine 或 Azure App Service。也有一些独立的，如 Heroku。

除了更通用的 PaaS 之外，还有**通信平台即服务 (CPaaS)**，可以使用整个通信后端，包括音频和视频，可以将它们集成到解决方案中。这项技术可以提供视频支持的帮助，或者将实时聊天集成到应用程序中。

软件即服务 (SaaS) 模型

有时，可能不想自己开发软件组件，而只想使用现有的组件。**软件即服务 (SaaS)** 提供了一个托管应用程序。使用 SaaS，不需要担心基础设施或构建在其上的平台，甚至不需要担心软件本身。提供商负责安装、运行、更新和维护整个软件堆栈，以及备份、许可和扩展。

在 SaaS 模型中，可以得到各种各样的软件。示例从 office 365 和 Google Docs 这样的办公套件到 Slack 这样的信息软件，通过**客户关系管理 (CRM)** 系统，甚至延伸到游戏解决方案，如云游戏服务，允许玩托管在云上的视频游戏。

通常要访问这些服务，只需要一个浏览器，所以这可能是为公司员工提供远程工作能力的重要一步。

可以创建自己的 SaaS 应用程序，并通过喜欢的方式或 AWS Marketplace 等方式向用户提供服务。

功能即服务 (FaaS) 模型和无服务器架构

随着原生云的出现，另一种日益流行的模型是**功能即服务 (FaaS)**。如果想要实现一个无服务器的架构，这个模型会很有拥。使用 FaaS，可以获得一个平台（类似于 PaaS），可以在其上运行短期应用程序或功能。

在 PaaS 中，通常需要至少运行一个服务实例。而在 FaaS 中，只能在实际需要时运行它们，函数可以使处理请求的时间更长（以秒为单位，需要启动该函数）。但是，可以缓存其中一些请求，以减少延迟和成本。谈到成本，如果要长时间运行这些功能，FaaS 的成本可能比 PaaS 高得多，因此在设计系统时必须进行计算。

如果使用正确，FaaS 将服务器从开发人员那里抽象出来，因为它可以基于事件，而不是资源，所以可以降低成本，并提供更好的扩展性。该模型通常用于运行预定的或手动触发的任务，处理批

量或数据流，以及处理传入的不那么紧急的请求。一些流行的 FaaS 提供商有 AWS Lambda、Azure Functions 和 Google Cloud Functions。

已经讨论了云中常见的服务模型，现在来了解在设计分布式系统时的错误假设。

4.2.2 避免分布式计算的错误

当刚接触分布式计算的人开始设计这类系统时，往往会忘记或忽略这类系统的一些方面。虽然早在 90 年代就有人注意到了，但现在仍然存在。

这些错误将在下面进行讨论，先了解一下。

可靠的网络

网络设备为长期完美运行而设计，但仍然会发生包丢失的情况，其原因包括由于无线网络信号不良而导致的停电、配置错误、有人被电缆绊倒，甚至是动物咬断电线。例如，Google 必须用 Kevlar 纤维保护他们的水下电缆，因为他们被鲨鱼咬怕了（是的，这是真的）。应该始终假定数据可能在网络的某个地方丢失。即使没有发生这种情况，软件问题仍然可能在网络的另一端发生。

要避免此类问题，请确保有自动重试失败网络请求的策略，以及处理常见网络问题的方法。在重试时尽量不要使另一方超载，也不要多次提交相同的事务。可以使用消息队列存储并重新发送。

像断路器这样的模式，将在本章后面展示，也会有帮助。哦，还要确保不要无限等待，因为每次失败的请求都会占用资源。

延迟为零

在正常情况下，运行的网络和服务也需要一些时间来响应。有时需要花更长的时间，特别是当负载大于平均水平时。有时，请求可能需要几秒钟才能完成。

尝试设计系统，使它不会等待太多细粒度的远程调用，因为每个调用都会增加总处理时间。即使在本地网络中，1 万条记录的 1 万条请求，也要比 1 万条记录的 1 条请求慢得多。为了减少网络延迟，请考虑批量发送和处理请求。还可以通过在等待结果的同时，执行其他处理任务来隐藏调用的成本。

其他处理延迟的方法是引入缓存，在发布者-订阅者模型中推送数据而不是等待请求，或者部署更靠近客户，例如使用**内容分发网络 (CDN)**。

无限带宽

在向架构添加新服务时，请确保注意它将使用多少流量。有时，可能希望通过压缩数据或引入节流策略来降低带宽。

这种错误也与移动设备有关。如果信号较弱，往往网络会成为瓶颈。从而，手机应用使用的数据量通常应该保持在较低水平，使用面向前端的后端中描述的模式（第 2 章）可以帮助节省宝贵的带宽。

如果后端需要在一些组件之间传输大量数据，请确保这些组件是紧密连接在一起的：不要在单独的数据中心运行。对于数据库，这通常可以归结为复制。像 CQRS（本章后面会讨论）这样的模式也很方便。

安全的网络

这是一个危险的假设。一条链的强度取决于它最薄弱的环节，不幸的是，在分布式系统中有许多环节。下面是一些增强链接的方法：

- 一定要始终对使用的每个组件、基础设施、操作系统和其他组件应用安全补丁。
- 培训你的员工，努力保护系统不受人为因素的影响。有时，就是一个流氓员工破坏了系统。
- 如果系统处于在线状态，就会受到攻击，并且有可能在某一点上发生漏洞。一定要有如何应对此类事件的书面计划。
- 可能听说过纵深防御原则。这可以归结为对系统的不同部分（基础设施、应用程序等等）进行不同的检查，以便在发生漏洞时，其范围和相关的损害将得到限制。
- 使用防火墙、证书、加密和适当的身份验证。

关于安全性的更多信息，请参考第 10 章。

不变的拓扑结构

在微服务时代尤其正确。自动扩展和物尽其用基础设施管理方法，意味着拓扑结构将不断变化。这可能会影响延迟和带宽，因此这种假设的一些结果与前面描述的相同。

幸运的是，上述方法还提供了关于如何有效管理服务器群的指导方针。依赖主机名和 DNS（而不是硬编码 IP）是朝着正确方向迈出的一步，本书后面将介绍的服务发现是另一个步骤。第三步（更大的一步）是始终假设实例可能会失败，并自动对此类场景做出反应。Netflix 的 *Chaos Monkey* 工具也可以对相应的情况进行测试。

只有一个管理员

关于分布式系统的知识，由于其性质，通常是分布式的。不同的人负责此类系统，及其基础设施的开发、配置、部署和管理。不同的组件通常由不同的人进行升级，不一定是同步的。还有所谓的总线因素，就是项目关键成员被巴士撞到的风险因素。

该如何应对这一切？答案由几个部分组成，其中之一就是 DevOps 文化。通过促进开发和操作之间的密切协作，可以共享关于系统的知识，从而减少总线因素。引入持续交付，可以升级项目并使其保持正常运行。

尝试对系统进行松散耦合和向后兼容的建模，这样一个组件的升级就不需要带动对其他组件的升级。一种简单的解耦方法是在它们之间引入消息传递，因此可以考虑添加一个或两个队列，将有助于确定在升级期间的停机时间。

最后，尝试监视系统，并在一个集中的地方收集日志。分散系统不必在十几台不同的机器上手动查看日志。**ELK(Elasticsearch, Logstash, Kibana)** 栈是宝贵的。Grafana, Prometheus, Loki 和 Jaeger 也很受欢迎，尤其是 Kubernetes。如果正在寻找比 Logstash 更轻量级的工具，请考虑 Fluentd 和 Filebeat，尤其是处理容器的时候。

传输成本为零

这种假设对于计划项目及其预算非常重要。为分布式系统构建和维护网络需要时间和金钱，无论是在本地部署还是在云中部署——只是何时支付成本的问题。试着估算设备、要传输的数据（云

提供商为此收费) 和所需人力的成本。

如果依赖于压缩, 请注意, 虽然这可以降低网络成本, 但可能会提高计算的成本。通常, 使用二进制 API(比如基于 gRPC 的 API) 将比基于 JSON 的 API 更廉价(也更快), 而且也比 XML 廉价。如果需要发送图像、音频或视频, 必须评估其开销。

同构的网络

即使计划在网络上使用什么硬件和运行什么软件, 也很容易出现一些异构性。某些机器上略有不同的配置、需要集成的遗留系统使用的不同通信协议, 或者向系统发送请求的不同移动电话, 这些只是其中的几个例子。另一个是通过在云中使用额外的人工服务来扩展本地的解决方案。

尽量限制所使用的协议和格式的数量, 使用标准的协议和格式, 并避免供应商锁定, 以确保系统可以在此类异构环境中正常通信。异构性也可能意味着弹性的差异。尝试使用断路器模式, 重新处理此问题。

现在已经讨论了错误的假设, 接下来就来了解分布式架构的另一个重要的方面。

4.2.3 CAP 定理和最终一致性

要设计跨多个节点的系统, 需要了解并使用某些定理, 其中之一就是 **CAP 定理**。其是在设计分布式系统时, 需要做出的重要选择, 其名称来源于分布式系统的三个属性。具体如下:

- **一致性**: 每次读取都将获得最近一次写入(或错误)之后的数据。
- **可用性**: 每个请求都会得到无误的响应(不保证会获得最新的数据)。
- **分区容错性**: 即使两个节点之间出现网络故障, 系统作为一个整体仍可以继续工作。

这个定理说明, 对于一个分布式系统, 最多只能选择这三个属性中的两个。

只要系统正常运行, 就可以满足这三个特性。然而, 从错误假设中知道“网络不可靠”, 因此就需要分区。这种情况下, 分布式系统仍然可以正常运行。这意味着该定理实际上是需要在交付分区容错和一致性(即 CP) 或交付分区容错和可用性(即 AP) 之间做出选择。通常, 后者是更好的选择。如果选择 CA, 则必须完全删除网络, 只剩下单节点系统。

分区下使用交付一致性, 在等待数据一致性时, 要么返回一个错误, 要么会有超时风险。如果选择可用性而不是一致性, 则可能会返回旧数据——最新写入的数据可能无法跨分区传播。

这两种方法适用于不同的需求。如果系统需要原子读写, 例如: 客户可能会亏钱, 那么就使用 CP。如果系统必须继续在分区下运行, 或者可以允许最终的一致性, 那么就使用 AP。

嗯, 那一致性是什么呢? 为了理解这一点, 来了解一下不同层次的一致性。

提供强一致性的系统中, 每个写操作都同步传播。从而所有的读操作将总是看到最新写入的数据, 甚至以更高的延迟或更低的可用性为代价。这是 DBMS 关系模型提供的(基于 ACID 保证)最适合需要事务的系统。

另一方面, 在提供最终一致性的系统中, 只能保证在写操作之后, 读操作最终会看到更改。通常, “最终”指的是几毫秒。这是由于此类系统中的数据复制具有异步特性, 而不是上一段提到的同步传播。这里没有提供 ACID 保证(例如, 使用 RDBMS), 而是使用了 BASE 语义, 通常由 NoSQL 数据库提供。

如果系统是异步的, 并且为最终一致(AP 系统通常是这样的), 就需要有一种方法来解决状态冲突。一种常见的方法是在实例之间交换更新, 并选择第一次或最后一次写入作为可接受的写入。

接下来，来了解两个可以帮助实现一致性的模式。

Saga 机制和补偿事务

当需要执行分布式事务时，Saga 模式非常有用。在微服务时代之前，如果有一台主机和数据库，那么可以依赖数据库引擎来处理事务。在一台主机上有多个数据库，可以使用**两阶段提交 (2PCs)**来完成。对于 2PCs，会有一个协调器，它首先告诉所有数据库准备好，当数据库都报告准备好了，它将告诉所有数据库进行事务提交。

现在，由于每个微服务可能都有自己的数据库（如果想要可扩展性，就应该这样），并且它们遍布于基础设施，因此不能再依赖简单的事务和 2PCs（失去这种能力通常意味着不再需要 RDBMS，因为 NoSQL 数据库可以更快）。

相反，可以使用 saga 模式。用一个例子演示一下。

假设要创建一个在线仓库，跟踪它有多少供应，并允许信用卡支付。要处理订单，在所有其他服务中，需要三个服务：一个用于处理订单，一个用于保留供应，一个用于刷卡。

现在，有两种方式可以实现 Saga 模式：**基于无中心协调器**和**基于中心化控制器**。

基于无中心协调器的 *Saga*

第一种情况下，Saga 的第一部分将是订单处理服务向供应服务发送事件。这个将完成它的部分，并将另一个事件发送给支付服务。然后支付服务将另一个事件发送回订单服务。这将完成交易（Saga），现在可以愉快地发送订单了。

如果订单服务想要跟踪事务的状态，只需要侦听所有这些事件。

当然，有时订单无法完成，需要进行回滚。Saga 的每个步骤都需要分别小心地回滚，因为其他事务可以并行运行，例如：修改供应状态。这种回滚称为**补偿事务**。

这种实现 Saga 模式的方法非常简单，但是如果所涉及的服务之间存在许多依赖关系，使用中心化方法可能会更好。说到这个，现在让我们来看看 Saga 的第二种方法。

基于中心化控制器的 *Saga*

需要消息代理来处理服务之间的通信，还需要协调器来协调整个过程。订单服务将向协调器发送一个请求，然后协调器将向供应服务和支付服务发送命令。然后，其中每一个都将完成自己的工作，并通过代理上可用的应答通道将应答发送回协调器。

此场景中，编排器拥有编排事务所需的所有逻辑，服务本身不需要知道参与该 Saga 的其他服务。

如果向编排器发送一条消息，说明其中一个服务失败了（例如，信用卡已经过期），将需要启动回滚。在示例中，它将再次使用代理向特定的服务发送适当的回滚命令。

好了，关于一致性就聊到这里。现在让我们切换到与可用性相关的话题吧。

4.3. 使系统具有容错性和可用性

可用性和容错是软件质量，至少对每个架构都有一定的的重要性。如果系统无法做到，那创建软件系统还有什么意义呢？本节中，将了解这些术语的确切含义，以及在解决方案中提供的技巧。

4.3.1 计算系统的可用性

可用性是系统正常运行、可用和可访问的时间百分比。导致系统无法响应的崩溃、网络故障或极高的负载(例如，来自DDoS攻击)都会影响其可用性。

通常，尽可能高的可用性是一个不错的指标。可能会偶然发现数9的术语，因为可用性通常指定为99%(两个9)，99.9%(三个)等。每增加一个9就更难获得，所以在做出承诺时要小心。看看下面的表格，看看如果按月计算，能承受多长时间的停机时间：

停机时间/月	正常运行时间
7 小时 18 分钟	99% (“两个9”)
43 分钟 48 秒	99.9% (“三个9”)
4 分钟 22.8 秒	99.99% (“四个9”)
26.28 秒	99.999% (“五个9”)
2.628 秒	99.9999% (“六个9”)
262.8 毫秒	99.99999% (“七个9”)
26.28 毫秒	99.999999% (“八个9”)
2.628 毫秒	99.9999999% (“九个9”)

云应用程序会提供**服务级别协议(SLA)**，它指定每一给定时间段(例如一年)出现停机的时间，云服务的SLA将依赖于构建云服务的SLA。

要计算需要协作的两个服务之间的复合可用性，应该将它们的正常运行时间相乘。这意味着如果有两个可用性为99.99%的服务，复合可用性将为 $99.99\% * 99.99\% = 99.98\%$ 。要计算冗余服务(例如两个独立区域)的可用性，应该将它们的不可用性相乘。例如，如果两个区域有99.99%可用性，它们的总不可用性将是 $(100\% - 99.99\%)*(100\% - 99.99\%) = 0.01\% * 0.01\% = 0.0001\%$ ，因此复合可用性是99.9999%。

这里，不可能提供100%的可用性。故障确实会不时地发生，所以系统需要容错。

4.3.2 构建容错系统

容错是系统检测故障，并优雅地处理故障的能力。因为由于云的性质，有许多可能会突然出现各种各样的问题，所以基于云的服务必须具有容错性。良好的容错能力有助于提高服务的可用性。

不同类型的问题需要不同的处理方法：从预防到检测，再到最小化影响。先从避免单点故障的方法开始。

替代装置

最基本预防方式是引入**替代装置**。类似于为汽车准备一个备用轮胎，这样就有了一份服务的副本，当主服务器出现故障时，可以由副本接管。这种方式也称为**失效备援**。

备用服务器如何了解何时介入？实现的一种方法是使用检测故障中的心跳机制。

为了使切换更快，可以将所有进入主服务器的消息也发送到备用服务器，这称为**热备份**。好的方式是保留最后一条消息，如果这条消息有毒，并杀死了主服务器，备份服务器可以直接拒绝它。

上述机制称为主动(或主从)故障转移，因为备份服务器不处理传入的流量。如果是这样，将有一个双控(或双主)故障转移。有关双控架构的更多信息，请参考扩展阅读中的最后一个链接。

确保在发生故障转移时不会丢失数据，使用带有备份存储的消息队列可能有助于解决这一问题。

领袖选举

对于两个服务器来说，了解两个服务器各自的任务也很重要——如果两个服务器开始都作为主实例运行，就可能会遇到麻烦。选择主服务器称为领袖选举模式。有几种方法可以做到这一点，例如：引入第三方仲裁者，通过竞争获得共享资源的独占所有权，通过选择级别最低的实例，或通过使用恶霸选举或令牌环选举等算法。

领袖选举也是下一个相关概念的重要组成部分：达成共识。

达成共识

如果希望系统即使在网络分区发生，或某些服务实例出现故障时也能运行，需要一种方法使实例之间达成共识。必须同意承担什么样的风险，以及以相应的顺序。一种简单的方法是允许每个实例对正确的状态进行投票。然而，在某些情况下，这不足以正确地或根本无法达成一致。另一种方法是选举一个领袖，让他宣传自己的能力。因为手工实现这样的算法不容易，所以建议使用行业验证的主流共识协议，如 Paxos 和 Raft。后者会更受欢迎，因为它更简单，更容易理解。

现在让我们讨论另一种防止系统出错的方法。

重复

这种方法在数据库中尤为流行，并且有助于扩展数据库。重复意味着将运行服务的几个实例与重复的数据并行，其也会处理所有的传入流。

Note

不要混淆复制和分片。后者不需要冗余数据，可以在带来良好的性能。如果正在使用 Postgres，那可以试试 Citus(<https://www.citusdata.com>)。

对于数据库，有两种方式进行重复。

单主架构

这个场景中，所有服务器都能够执行只读操作，但是只有一个主服务器也可以写。数据从主节点通过从节点进行复制，可以采用一对多拓扑，也可以采用树状拓扑。如果主服务器发生故障，系统仍然可以以只读模式运行，直到故障解决。

多主架构

还可以使用一个具有多个主服务器的系统。如果有两个服务器，可以进行双主重复的方案。如果其中一个服务器挂了，其他服务器仍然可以正常运行。但现在要么同步写操作，要么提供更宽松的一致性保证。另外，还需要一个负载均衡器。

这类重复的例子还包括微软的 Active Directory、OpenLDAP、Apache 的 CouchDB 或

Postgres-XL。

现在了解一下两种防止过高负载(会导致故障)的方法。

基于队列的负载均衡

此策略旨在减少系统负载突然激增的影响。请求泛滥会导致性能问题、可靠性问题，甚至丢失有效的请求。还有，队列是对该场景进行缓和。

要实现此模式，只需要引入队列，以便异步添加传入请求。可以使用 Amazon 的 SQS、Azure 的服务总线、Apache Kafka、ZeroMQ 或其他队列来实现。

现在，不是在传入请求中出现峰值，而是平均负载。服务可以从上述队列中获取请求，并在不知道负载增加的情况下处理它们，就这么简单。

如果是高性能队列，并且任务可以并行化，则此模式的另一个好处是有更好的扩展性。

此外，如果服务不可用，当服务恢复时，请求仍然会添加到该服务的队列中进行处理，因此这可能是一种有助于提高可用性的方法。

如果请求不经常出现，考虑将服务实现为仅在队列中有项目时运行的函数，以节省成本。

使用此模式时，总延迟将随着队列的增加而增加。Apache Kafka 和 ZeroMQ 可以提供低延迟队列，但如果这是一个问题，需要另一种方法来处理负载。

反压力

如果负载仍然很高，很可能是有更多的任务超出了服务器处理能力。这可能会导致缓存丢失和交换异常(如果请求不再适合内存)，以及删除请求和其他麻烦的事情。如果预计会有沉重的负担，反压力可能是一个更好的解决办法。

反向压力意味着，不会在每个传入请求时对服务施加更大的压力，而是将其推回调用方，以便调用方需要处理这种情况。有几种不同的方法可以做到这一点。

例如，可以阻塞接收网络数据包的线程。调用者将发现无法将请求推到服务——相反，其实是我们将压力推回到上游。

另一种方法是，识别更大的负载并简单地返回错误代码，例如：503。可以为架构建模，以便由另一个服务来完成请求。Envoy 代理(Envoy Proxy, <https://envoyproxy.io>)就是这样一个服务，许多其他场合都会用到。

Envoy 代理可以应用基于预定义配额的反压力，所以服务实际上永远不会超载。它还可以测量处理请求和仅在超过某个阈值时，施加反压力所需的时间；还有许多其他情况会返回各种错误代码。希望使用者有提前的计划，从而明确如果压力返回来应该怎么办。

既然已经知道了如何防止故障，再了解一下如何在故障发生时检测故障。

4.3.3 检测故障

正确而快速的故障检测可以省去很多麻烦，而且往往还能节约成本。有许多方法可以根据不同的需要检测故障。让我们来过一下。

边车设计模式

Envoy 代理可以说这是**边车设计模式**的一个例子。该模式在很多情况下都很有用，而不仅仅是错误预防和检测。

通常，边车允许向服务添加许多功能，而不需要编写额外的代码。类似地，物理边车可以连接到摩托车上，软件边车则可以连接到服务上——这两种情况下都可以对功能进行扩展。

边车如何帮助检测故障？首先，通过提供健康检查功能。对于被动运行状况检查，特使代理可以检测服务集群中的实例是否已经开始行为异常，这叫做**离群检测**。Envoy 代理可以查找连续的 5XX 错误代码、网关故障等。除了检测这些错误之外，还可以弹出它们，以便整个集群保持健康。

Envoy 代理还提供了主动的运行状况检查，这意味着它可以探测服务本身，而不仅仅是观察它对传入流的反应。

本章中，展示了一般的边车模式的其他用法，特别是特使代理。现在来讨论另一种故障检测机制。

心跳机制

最常见的故障检测方法是通过**心跳机制**。一个**心跳**是在两个服务之间定期（通常是几秒钟）发送的信号或消息。

如果几个连续的心跳没有了，接收服务可以认为发送服务挂了。前面几节中提到的主备份服务的情况下，这可能导致故障转移。

实现心跳机制时，要确保可靠性。误报可能会造成麻烦，因为服务可能会混淆，例如：哪个服务应该成为新的主服务。一个好方法是仅为心跳提供一个单独的端点，这样就不会那么受到常规端点流的影响。

漏桶计数器

另一种检测错误的方法是添加**漏桶计数器**。对于每个错误，计数器将增加，并且在达到某个阈值（桶已满）后，将发出错误信号并处理。在固定的时间间隔内，计数器将减少（因此，漏桶了）。这样，只有在短时间内出现很多错误时，才会认为是故障。

如果有时出现错误是很正常的（例如：在处理网络时），这种模式会很有用。

既然已经知道了如何检测故障，就继续了解在故障发生时应该做什么。

4.3.4 降低故障的影响

检测正在发生的故障需要花费时间，而解决故障则需要更多宝贵的资源。这就是为什么应该努力将错误的影响降到最低。

重试调用

当应用程序调用另一个服务时，有时调用会失败。对于这种情况，最简单的补救方法就是重试。如果错误是暂时的，而没有重试，那么该错误可能会在系统中传播，造成更大的损失。实现自动重试此类调用可以省去很多麻烦。

还记得 Envoy 代理吗？它可以代表管理者执行自动重试，从而无需对源代码进行更改。

例如，在 Envoy 代理中可以添加一个重试策略的配置：

```
retry_policy:  
  retry_on: "5xx"  
  num_retries: 3  
  per_try_timeout: 2s
```

如果 Envoy 代理返回错误，比如：映射到 5XX 代码的 503 HTTP 代码或 gRPC 错误，则将使 Envoy 代理重试调用。将有三次重试，如果没有在 2 秒内完成，每次认为失败。

避免级联故障

如果没有重试，错误将会传播，在整个系统中造成一连串的失败。现在来了解更多防止这种情况发生的方法。

断路器

断路器的模式是一个非常有用的工具。它允许快速地注意到某个服务无法处理请求，因此对它的调用可能会短路。这可以发生在调用方附近（Envoy 代理提供了这样的功能），也可以发生在调用方（减少了调用时间）。在使用 Envoy 代理的情况下，可以添加以下配置：

```
circuit_breakers:  
  thresholds:  
    - priority: DEFAULT  
      max_connections: 1000  
      max_requests: 1000  
      max_pending_requests: 1000
```

这两种情况下，对服务的调用引起的负载可能会下降，这在某些情况下可以帮助服务恢复正常操作。

如何在调用方实现一个断路器？在进行了一些调用之后，例如：漏桶溢出了，可以在指定的一段时间内停止进行新的调用（例如，直到桶不再溢出）。

隔板

另一种限制断层扩散的方法是直接从隔板中获取。当建造船只时，通常不希望船体上有一个洞，让船充满水。为了减少这些洞的破坏，可以把船体分成几个舱壁，每一个舱壁都很容易分开。在这种情况下，只有损坏的舱壁才会充满水。

同样的原则也适用于限制软件架构中的故障影响。可以将实例划分到组中，也可以将其使用的资源分配到组中。设置配额也可以视为该模式的一个示例。

可以为不同的用户组创建单独的舱壁，如果需要对它们进行优先级划分，或者为关键用户提供不同级别的服务，这会很有用。

多地部署

最后一种方法称为**多地部署**，这个名字来自地理节点。当服务部署在多个地区时，可以使用它。

当一个区域发生故障时，可以将流重定向到其他不受影响的区域。当然，与调用同一数据中心中的其他节点相比，这将导致更高的延迟，但通常将不太重要的用户重定向到远程区域要比完全失败调用要好得多。

既然已经知道了如何通过系统体系结构提供可用性和容错，接下来就来了解一下如何将其组件集成在一起。

4.4. 集成系统

分布式系统不仅可以运行的孤立实例，也相互通讯，两种方式可以进行适当地整合。

关于集成的主题已经讨论了很多，因此在本节中，将展示一些模式，用于集成全新的系统，以及如何使系统的新组件与现有组件（通常是遗留部分）共存。

为了不使本章的体量成书，这里会对现有的相关书籍进行推荐。如果对集成模式感兴趣，特别是消息传递，那么 Gregor Hohpe 和 Bobby Woolf 的《企业集成模式》是必读的。

来简要地看一下本书介绍的两个模式。

4.4.1 管道与过滤器模式

第一个集成模式叫做**管道与过滤器**，其目的是将一个大的处理任务分解为一系列较小的、独立的任务（称为过滤器），然后可以将这些任务连接在一起（使用管道，如消息队列）。这种方法有较好的扩展性、性能和重用性。

假设需要接收和处理传入的订单。当然，可以在一个大模块中完成，所以不需要额外的通信，但这样一个模块很难测试其不同的功能，也很难对其进行扩展。

不过，可以将订单处理分成多个单独的步骤，每个步骤由一个不同的组件处理：一个用于解码，一个用于验证，另一个用于实际处理订单，然后还有一个用于存储订单。使用这种模式，就可以单独执行这些步骤中的每一个，在需要时可以轻松地替换或禁用，并重用来处理不同类型的输入消息。

如果希望同时处理多个订单，还可以流水线处理：一个线程验证消息，另一个线程解码下一个消息，以此类推。

缺点是需要使用同步队列作为管道，这会带来一些开销。

要扩展处理的步骤，可能需要将此模式与管道中的下一个步骤一起使用。

4.4.2 使用者竞争

使用者竞争的思想很简单：有一个输入队列（或消息传递通道）和几个从队列中，并发获取和处理项的使用者实例。每个使用者都可以处理消息，因此他们相互竞争成为接收者。

通过这种方式，可以获得可扩展性、自由负载平衡和弹性。添加了队列之后，现在还拥有了基于队列的负载均衡模式。

如果要减少请求的延迟，或者只是希望以更紧急的方式执行提交到队列的特定任务，则此模式可以轻松地与优先级队列集成。

Note

如果顺序很重要，那么使用这种模式会很棘手。用户接收和完成消息处理的顺序可能不同，因此要确保这不会影响系统，或者可以找到一种方法在以后重新排列结果。如果需要按顺序处理消息，则可能无法使用此模式。

现在，为了帮助集成现有的系统，来继续了解更多的模式。

4.4.3 从遗留系统的过渡

从头开始开发一个系统是幸福的。开发不是维护，有了使用前沿技术的可能性——为什么不喜欢的呢？不幸的是，当与现有的遗留系统开始集成时，这种幸福就结束了。幸运的是，有一些方法可以缓解这种痛苦。

防腐层

引入防腐层可以让解决方案与具有不同语义的遗留系统，进行无痛的集成。这层负责双方之间的通信。

这样的组件可以让解决方案有更大的灵活性——而不需要损害技术栈或架构决策。要实现这一点，只需要在遗留系统中进行最小的更改（如果遗留系统不需要调用新系统，则不需要进行更改）。

例如解决方案是基于微服务的，那么遗留系统可以只与防腐层通信，而不是直接定位和到达每个微服务。转换（例如，由过时的协议版本）也在这层中完成。

请记住，添加这样的层可能会引入延迟，并且必须满足解决方案的质量属性，例如可扩展性。

扼杀者模式

扼杀器模式允许从遗留系统逐步迁移到新系统。刚才看到的防腐层对于两个系统之间的通信很有用，而扼杀者模式则用于从两个系统向外部提供服务。

迁移过程的早期，扼杀器将把大多数请求放到遗留系统中。在迁移过程中，越来越多的调用可以转发到新的调用中，同时越来越多地限制遗留系统，限制它所提供的功能。作为迁移的最后一步，扼杀器逼迫遗留系统退役——新系统将提供所有的功能：

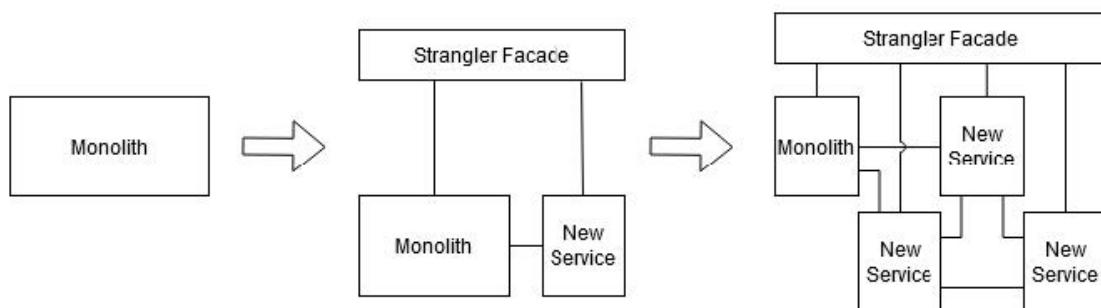


图 4.1 - 扼杀者模式。迁移之后，扼杀器仍然可以用作遗留请求的入口点或适配器

对于小型系统，这种模式可能有些大材小用。如果共享数据存储，或者是用于事件源系统，那这种模式会变得棘手。在将其添加到解决方案时，请提前为实现适当的性能和扩展性做好计划。

接下来，了解一些有助于实现这两个属性的方式。

4.5. 分级性能

设计 C++ 程序时，性能通常是一个关键因素。虽然在单个应用范围内使用该语言可能会有很长的路要走，但适当的高级设计对于实现最佳延迟和吞吐量很重要。先来讨论几个关键的模式。

4.5.1 CQRS 和事件源

有许多方法可以扩展计算，但扩展数据访问可能很棘手。然而，当用户基础增长时，这通常是有必要的。**命令查询责任分离 (CQRS)** 模式可以在这里提供帮助。

命令-查询式责任分离

传统的 CRUD 系统中，读和写都使用相同的数据模型和数据流，并以相同的方式执行。名义上的隔离意味着可以以两种不同的方式处理查询 (读取) 和命令 (写入)。

许多应用程序的读和写的比例偏差很大——通常的应用程序中，从数据库中读取的数据比更新数据库的数据要多得多。这意味着需要尽可能快地读取数据，从而有更好的性能：读和写现在可以分别进行优化和扩展。更重要的是，如果许多写操作相互竞争，或者需要维护所有写操作的轨迹，或者一组 API 用户应该具有只读访问权限，那么引入 CQRS 可能会有所帮助。

拥有独立的读和写模型可以让不同的团队在分开工作。从事读取方面工作的开发人员不需要对领域有深入的了解，这是正确执行更新所必需的。当他们发出请求时，只需要一个简单的调用就可以读取层得到一个**数据传输对象 (DTO)**，而非通过域模型。

如果不知道 DTO 是什么，可以考虑从数据库返回项目数据。如果调用者要求一个项目列表，可以提供一个只包含项目名称和缩略图的 *ItemOverview* 对象。另一方面，如果想要特定商店的商品，也可以提供一个包含名称、更多图片、描述和价格的 *StoreItem* 对象。*ItemOverview* 和 *StoreItem* 都是 DTO，从数据库中相同的 *Item* 对象中抓取数据。

读取层可以位于用于写的数据存储的顶部，也可以是通过事件更新的不同的数据存储，如图 4.2 所示。

使用图 4.2 中所示的方法，可以创建多个不同的命令，每个命令都有自己的处理程序。命令是异步的，不会向调用者返回。每个处理程序都使用域对象，并会等待更改完成。之后，将发布事件，事件处理程序可用于更新读取操作使用的存储。继续上一个示例，项目数据查询将从由诸如 *ItemAdded* 或 *ItemPriceChanged* 等事件更新的数据库中获取信息，这些事件可以由 *AddItem* 或 *ModifyItem* 等命令触发。

使用 CQRS 可以为读写操作使用不同的数据模型。例如，可以创建存储过程和物化视图来加速读取。对于读存储和域存储，使用不同类型的存储 (SQL 和 NoSQL) 也有好处：持久化数据的一种方法是使用 Apache Cassandra 集群，而使用 Elasticsearch 是快速搜索存储数据的一种不错的方法。

除了前面的优点，CQRS 也有它的缺点。由于其引入的复杂性，通常不适合小型或要求较少的架构。还有，在域存储之后更新读存储，数据有一致性，但不是强一致性。

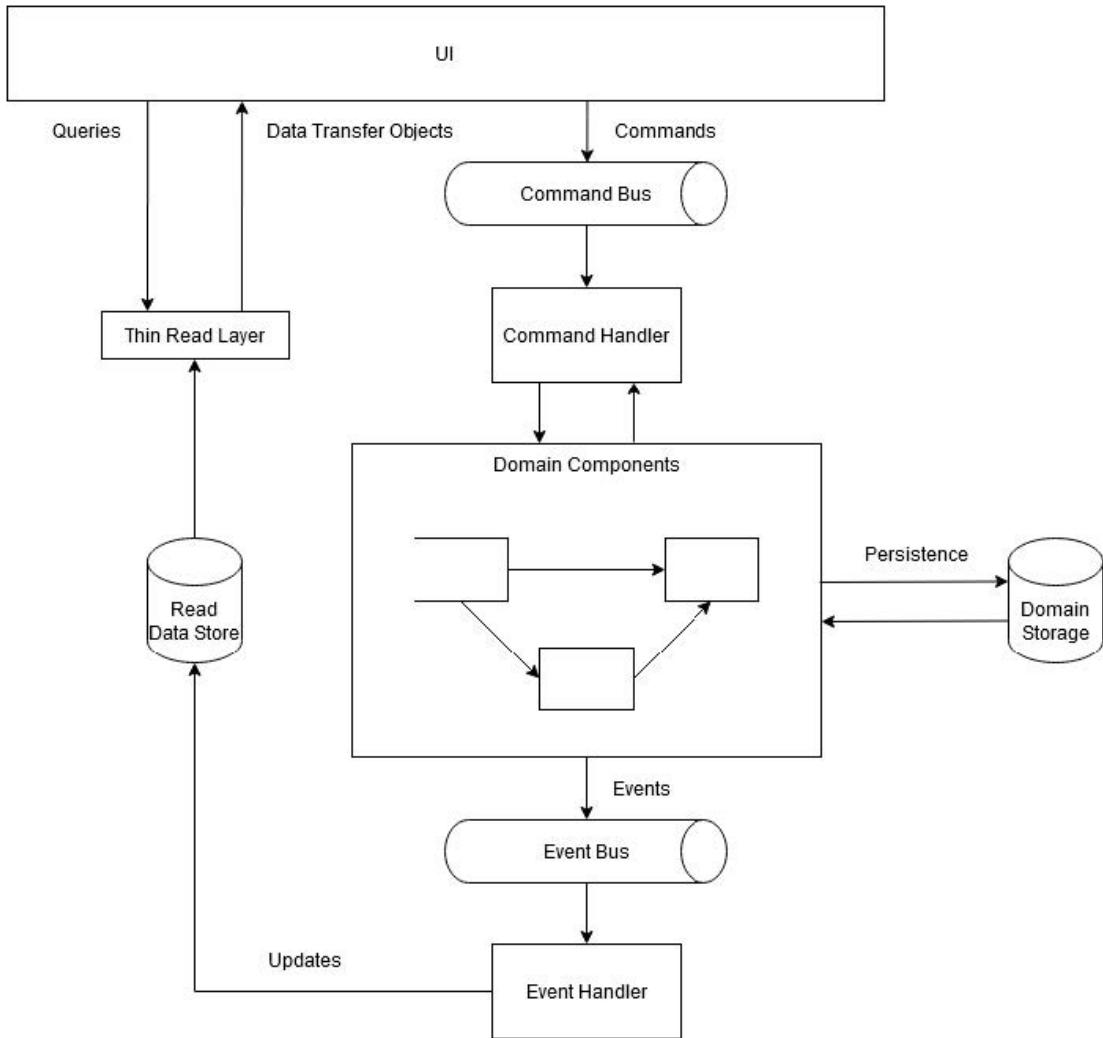


图 4.2 -带有事件源的 CQRS

命令-查询式分离

CQRS 实际上基于很久以前在 Eiffel 编程语言中引入的一个更简单的思想 (与引入契约的思想相同)。**命令查询分离 (CQS)** 是一个原则，设计分离 API 调用。命令和查询-就像在 CQRS，但不考虑规模，在目标编程和命令式编程中都发挥得很好。

如果函数名以 *has*, *is*, *can* 或类似的单词开头，那应该只是一个查询，而不会对底层状态进行修改或有其他副作用。这带来了两个好处：

- **代码更易读:** 这样的函数在语义上只是 *read*, 而不是 *write*。这使得在调试时查找状态变化更加容易。
- **减少海森堡 bug:** 如果需要调试在发布版本中出现的错误，但在调试版本中却没有 (或者相反)，就已经处理了一个海森堡 bug。这很少是令人愉快的事情。许多这样的错误可能是由修改状态的断言引起的。遵循 CQS 可以消除这样的 bug。

与断言类似，如果想要使用契约 (前置条件和后置条件)，那么只在其中使用查询非常重要。否则，禁用一些契约检查也可能导致海森堡 bug，更不用说这将是多么的违反直觉。

现在，来了解一下事件源。

事件源

如在第 2 章所述，事件源可以只存储发生在应用程序状态上的更改，而不是总是存储应用程序的整个状态（可能在更新期间处理冲突）。使用事件源可以通过消除并发更新和允许所有感兴趣的方对其状态执行，逐步更改来提高应用程序的性能。保存所做操作的历史记录（例如，市场交易）可以更容易地进行调试（通过稍后重播）和审核。这也带来了更多的灵活性和扩展性。引入事件源时，一些域模型可以变得更简单。

事件源的成本是最终保持一致。另一个问题是降低应用程序的启动速度——除非对状态进行定期快照，或者可以使用 CQRS 中的只读存储，上一节已经讨论过。

好了，关于 CQRS 和相关模式已经了解了。现在转向另一个有关性能的话题：缓存。

4.5.2 缓冲

正确地使用缓存可以产生更好的性能、更低的延迟、减少服务器负载（在云中运行的成本），并有助于解决可扩展性问题（所需的服务器更少）。

Note

如果在寻找 CPU 缓存的技巧，可以在第 11 章了解到相关信息。

缓存是一个大主题，在这里只讨论几个方面。

缓存的工作原理是，将最常读取的数据存储在访问时间较短的非持久存储中。有许多不同类型的缓存：

- **客户端缓存**: 用于专门为给定客户存储数据，通常放置在客户机或浏览器上。
- **Web 服务器缓存**: 为了加速网页的阅读，可以通过 Varnish 这样的 HTTP 加速器来缓存 Web 服务器的响应。
- **数据库缓存**: 许多数据库引擎都有内置的、可调优的缓存。
- **应用程序缓存**: 为了加速应用程序，可以从缓存读取数据，而不是从数据库读取数据。
- **CDN 也可以视为缓存**: 从靠近用户的位置提供内容，以减少延迟。

可以在集群中复制或部署某些类型的缓存，以提供大规模性能。另一种选择是进行分片：类似于对数据库进行分片，可以为数据的不同部分使用缓存的不同实例。

现在了解一下更新缓存中的数据的不同方法。毕竟，没有人喜欢旧数据。

更新缓存

有几种方法可以保持缓存数据的新鲜度。无论决定更新缓存项的人还是其他公司，对其进行了解都是很有意义的。本节中，将讨论它们的优缺点。

直写模式

如果需要强一致性，同步更新数据库和缓存是有效的方法。这种方法可以防止数据丢失：如果数据对用户可见，就已经写入数据库。直写缓存的缺点是，执行更新的延迟比其他方法更大。

滞后写

另一种方法，也称为回写，是只向用户提供对缓存的访问。当用户执行更新时，缓存将对传入的更新进行排队，然后将异步执行该更新，从而更新数据库。缺点是，如果出现错误，就永远不能写入数据。它也不像其他方法那样容易实现。但是，好处是用户看到的延迟最低。

缓存延迟

最后一种方法，也称为**延迟加载**，是关于按需填充缓存的。这种情况下，数据访问如下所示：

1. 调用缓存来检查值是否已经存在。如果是，就返回。
2. 提供值的主数据存储或服务。
3. 将值存储在缓存中，并返回给用户。

这种类型的缓存通常使用 Memcached 或 Redis，可以非常快速和高效—缓存只包含请求的数据。

但是，如果经常请求不在缓存中的数据，则前三个调用会显著增加延迟。为了减轻缓存重启的影响，可以用持久存储中选定的数据初始化(初始化)缓存。

缓存中的项也可能过时，因此最好为每个条目设置生存时间。如果要更新数据，可以采用直写的方式，即从缓存中删除记录并在数据库中更新它。当只使用基于时间的更新策略(例如，在 DNS 缓存中)使用多级缓存时要小心。这可能导致长时间使用旧数据。

已经讨论了缓存的类型和更新它们的策略，所以关于缓存的内容就到此为止。接下来，讨论扩展架构的另一个方面。

4.6. 部署系统

尽管部署服务听起来很简单，如果仔细观察，会发现还有很多事情需要考虑。本节将描述如何执行有效的部署、在安装服务后配置服务、在部署后检查服务是否保持正常运行，以及如何在最小化宕机时间内的同时完成这些工作。

4.6.1 边车设计模式

还记得本章前面提到的 Envoy 吗？对于高效的应用程序开发来说，它是一个非常有用的工具。可以将 Envoy 代理与应用程序一起部署，就像在摩托车旁边部署挎车一样，而不是将日志、监控或网络等基础设施服务嵌入到应用程序中，加在一起会比没有 sidekick(这种模式的另一种叫法) 的应用做得更多。

使用边车可以加速开发，因为它带来的许多功能需要由微服务独立开发。因为边车独立于应用程序，所以可以使用适合的编程语言进行开发。边车及其提供的所有功能，都可以由独立的开发团队进行维护，并独立于主服务进行更新。

因为边车就在需要增强的应用程序旁边，可以使用本地的进程进行通信。通常，与另一个主机通信相比，这就已经足够快，而且要快得多。但请记住，这有时可能是一个巨大的负担。

即使部署了第三方服务，将所选的边车部署到旁边仍然有用：可以监视资源使用情况，以及主机和服务的状况，还可以在整个分布式系统中跟踪请求。有时也可以根据服务的情况，通过编辑配置文件或 Web 界面动态地配置服务。

使用 Envoy 部署带有跟踪和反向代理的服务

现在让 Envoy 作为部署的代理。首先创建 Envoy 的配置文件，例子中名为 *envoy-front_proxy.yaml*，代理地址为：

```
static_resources:  
  listeners:  
    - address:  
        socket_address:  
          address: 0.0.0.0  
          port_value: 8080  
    traffic_direction: INBOUND
```

这里已经指定 Envoy 将侦听端口 8080 上的传入信息。在配置后，将把信息转到服务中。现在，让指定使用服务的实例集来处理 HTTP 请求，并在上面添加一些跟踪功能。首先，添加一个 HTTP 端点：

```
filter_chains:  
  - filters:  
    - name: envoy.filters.network.http_connection_manager  
      typed_config:  
        "@type":  
          type.googleapis.com/envoy.extensions.filters.network.  
          http_connection_manager.v3.HttpConnectionManager
```

现在，指定请求分配 ID，并由分布式跟踪系统 Jaeger 跟踪：

```
generate_request_id: true
tracing:
  provider:
    name: envoy.tracers.dynamic_ot
    typed_config:
      "@type":
        type.googleapis.com/envoy.config.trace.v3.DynamicOtConfig
      library: /usr/local/lib/libjaegertracing_plugin.so
      config:
        service_name: front_proxy
        sampler:
          type: const
          param: 1
        reporter:
          localAgentHostPort: jaeger:6831
        headers:
          jaegerDebugHeader: jaeger-debug-id
          jaegerBaggageHeader: jaeger-baggage
          traceBaggageHeaderPrefix: uberctx-
        baggage_restrictions:
          denyBaggageOnInitializationFailure: false
        hostPort: ""
```

将为请求创建 ID，并在本地 Jaeger 插件中使用 OpenTracing 标准 (*DynamicOtConfig*)。该插件将生成一个 Jaeger 实例在指定地址下运行，并添加指定的头文件。

还需要指定来自所有域名的所有信息 (参见匹配部分) 都应转到我们的服务集群：

```
codec_type: auto
stat_prefix: ingress_http
route_config:
  name: example_route
  virtual_hosts:
    - name: front_proxy
      domains:
        - "*"
```

```
routes:
  - match:
    prefix: "/"
  route:
    cluster: example_service
  decorator:
    operation: example_operation
```

这里定义了 *example_service* 集群。注意，到达集群的每个请求都将由预定义的操作修饰符标记。还需要指定路由器地址：

```
http_filters:
  - name: envoy.filters.http.router
    typed_config: {}
use_remote_address: true
```

现在知道了如何处理和跟踪请求，剩下的就是定义使用的集群。先从我们的服务集群开始：

```
clusters:
  - name: example_service
    connect_timeout: 0.250s
    type: strict_dns
    lb_policy: round_robin
    load_assignment:
      cluster_name: example_service
      endpoints:
        - lb_endpoints:
        - endpoint:
          address:
            socket_address:
              address: example_service
              port_value: 5678
```

每个集群可以有多个实例（端点）。这里，若决定添加更多端点，则将使用循环策略对传入请求进行负载平衡。

再添加一个管理界面：

```
admin:  
  access_log_path: /tmp/admin_access.log  
  address:  
    socket_address:  
      address: 0.0.0.0  
      port_value: 9901
```

现在将配置放在一个容器中，使用 Dockerfile 运行 Envoy，将其命名为 *Dockerfile-front_proxy*:

```
FROM envoyproxy/envoy:v1.17-latest  
  
RUN apt-get update && \  
apt-get install -y curl && \  
rm -rf /var/lib/apt/lists/*  
RUN curl -Lo -  
https://github.com/tetratelabs/getenvoy-package/files/3518103/getenvoy-cent  
os-jaegertracing-plugin.tar.gz | tar -xz && mv libjaegertracing.so.0.4.2  
/usr/local/lib/libjaegertracing_plugin.so  
  
COPY envoy-front_proxy.yaml /etc/envoy/envoy.yaml
```

还下载了在 Envoy 配置中使用的 Jaeger 本地插件。

现在指定如何使用 Docker Compose 在几个容器中运行代码。从前端代理服务定义开始，创建一个 *docker-compose.yaml* 文件:

```
version: "3.7"  
  
services:  
  front_proxy:  
    build:  
      context: .  
      dockerfile: Dockerfile-front_proxy
```

```
networks:  
  - example_network  
ports:  
  - 12345:12345  
  - 9901:9901
```

这里使用 Dockerfile，这是一个简单的网络，在主机上公开了容器的两个端口：服务和管理接口。现在让添加代理要指向的服务：

```
example_service:  
  image: hashicorp/http-echo  
  networks:  
    - example_network  
  command: -text "It works!"
```

在例子中，服务只会在一个简单的 Web 服务器中显示一个预定义的字符串。

现在，在另一个容器中运行 Jaeger，将其端口暴露给外部：

```
jaeger:  
  image: jaegertracing/all-in-one  
  environment:  
    - COLLECTOR_ZIPKIN_HTTP_PORT=9411  
  networks:  
    - example_network  
  ports:  
    - 16686:16686
```

最后一步是定义网络：

```
networks:  
  example_network: {}
```

完成了。现在可以使用 `docker-compose up --build` 运行这个服务，并将浏览器指向特定的端口。

使用边车代理还有一个好处：即使服务死机，边车通常是活着的，可以在主服务关闭时响应外部请求。当服务重新部署（例如，由于更新）时，这同样适用。说到这里，继续了解如何最小化相关的宕机时间。

4.5.2 零宕机部署

有两种常见的方法来最小化部署期间的宕机风险: 蓝绿部署和金丝雀发布。两者都可以使用 Envoy 边车模式。

蓝绿部署

蓝绿部署可以最小化宕机时间和与部署应用程序的相关风险。为此, 需要两个相同的生产环境, 称为蓝色和绿色。绿色服务于客户, 可以执行蓝色的更新。一旦进行了更新, 服务进行了测试, 所有看起来都很稳定, 可以切换流量, 让其流到更新的(蓝色)环境。

如果在切换后蓝色环境中发现了问题, 绿色环境仍然存在——可以将切换回来。用户甚至可能不会注意到, 而且由于两个环境都已启动并运行, 在切换期间应该不会停机。只要确保在切换期间不会丢失任何数据(例如, 在新环境中进行的事务)即可。

金丝雀发布

要避免所有服务实例在更新后都失败, 最简单的方法是, 不要一次更新所有服务实例。这就是蓝绿部署的增量变体背后的关键思想, 也称为金丝雀发布。

在 Envoy 中, 可以在配置的 *routes* 部分中输入以下内容:

```
- match:  
  prefix: "/"  
  route:  
    weighted_clusters:  
      clusters:  
        - name: new_version  
          weight: 5  
        - name: old_version  
          weight: 95
```

还应该记得在前面的代码片段中定义两个集群, 第一个是旧版服务:

```
clusters:  
  - name: old_version  
    connect_timeout: 0.250s  
    type: strict_dns  
    lb_policy: round_robin
```

```
load_assignment:  
    cluster_name: old_version  
    endpoints:  
        - lb_endpoints:  
            - endpoint:  
                address:  
                    socket_address:  
                        address: old_version  
                        port_value: 5678
```

第二个集群将运行新版本:

```
- name: new_version  
  connect_timeout: 0.250s  
  type: strict_dns  
  lb_policy: round_robin  
  load_assignment:  
    cluster_name: new_version  
    endpoints:  
        - lb_endpoints:  
            - endpoint:  
                address:  
                    socket_address:  
                        address: new_version  
                        port_value: 5678
```

当一个部署更新时，服务的新版本将只会让一小部分(这里:5%)的用户看到和使用。如果更新的实例保持稳定，并且没有检查和验证失败，那么可以分几个步骤逐步更新越来越多的主机，直到所有主机都切换到新版本。可以通过手动更新配置文件或使用管理端点来实现。

现在，来了解一下最后一个部署模式。

4.5.3 外部配置存储

若部署的是一个简单的应用程序，可以只部署其配置。然而，当希望有一个包含许多应用程序实例的更复杂的部署时，为了重新配置而重新部署新版本的应用程序很快就会成为一种负担，从而手动配置更改是不可取的。引入外部配置存储是解决问题的好方法。

从本质上说，应用程序可以从上述商店获取配置，而不是仅仅依赖于本地配置文件。这可以为多个实例提供公共设置，并为其中一些实例调优参数，同时拥有一种简单而集中的方式来监视所有配置。如果想让仲裁程序决定哪些节点将作为主节点，哪些节点将作为备份节点，外部配置存储可

以为实例提供此类信息。实现配置更新过程也很有用，这样可以在操作期间轻松地重新配置实例。可以使用现成的解决方案，如 Firebase Remote Config，利用基于 Java 的 Netflix Archaius，或者制作一个存储配置，利用云存储和更改通知。

了解了一些有用的部署模式，接下来转向另一个高级主题:API。

4.7. 管理 API

适当的 API 对于开发团队和产品的成功至关重要。可以将这个主题分成两个小主题: 系统级 API 和组件级 API。本节中，将讨论系统级别上处理 API，而下一章将提供关于组件级别的技巧。

除了管理对象之外，还需要管理整个 API。如果想要引入有关 API 使用的策略，控制对所述 API 的访问，收集性能指标和其他分析数据，或者只是根据客户使用的接口向他们收费，**API 管理 (APIM)** 就是问题的解决方案。

通常一组 APIM 工具包含以下组件:

- **API 网关**: API 的所有用户的单一入口。下一节将对此进行更多介绍。
- **报告和分析**: 监控 API、资源消耗或数据发送的性能和延迟。可以利用这些工具来检测使用趋势，了解 API 的哪些部分和背后的哪些组件是性能瓶颈，或者可以提供哪些 SLA，以及如何改进。
- **开发者门户**: 帮助用户快速了解 API，并使用 API。
- **管理员门户**: 管理策略、用户，并将 API 打包成可销售的产品。
- **货币化**: 根据客户使用 API 的方式向他们收费，并帮助相关的业务流程。

APIM 工具由云提供商和独立方提供，例如 NGINX 的 Controller 或 Tyk。

在为给定的云设计 API 时，云提供商通常会提供的良好实践。例如，可以在扩展阅读部分找到 Google 云平台的通用设计模式，其许多实践都围绕着 Protobuf 进行。

选择正确的使用 API 方式可以走的更远，向服务器提交请求的最简单方法是直接连接到服务。虽然对于小型应用程序来说很容易设置，但可能会导致性能问题。API 使用者可能需要调用几个不同的服务，从而导致高延迟。使用这种方法也不可能实现适当的扩展性。

更好的方法是使用 API 网关。此类网关通常是 APIM 解决方案的重要组成，也可以单独使用。

4.6.1 API 网关

API 网关是希望使用 API 的客户端的入口点，可以将传入的请求转到特定的实例或服务集群。因为不再需要知道所有后端节点，或者如何相互协作，所以这可以简化客户端的代码。客户端只需要知道 API 网关的地址——网关将处理其余的事情。由于对客户机隐藏了后端架构，因此可以在不涉及客户机代码的情况下对其进行重构。

网关可以将系统 API 的多个部分聚合，然后使用 **7 层路由** (例如，基于 URL) 到系统的部分。7 层路由是由两个云提供商提供的，还有 Envoy 等工具。

与本章中描述的许多模式一样，始终要考虑是否需要引入另一个模式来增加架构的复杂性。考虑添加它将如何影响可用性、容错和性能 (如果很重要的话)。毕竟，网关通常只是单个节点，所以不要让它成为瓶颈或故障点。

前面几章提到的后端前端模式可以认为是 API 网关模式的一个变体。在后端用于前端的情况下，每个前端都可以连接到自己的网关。

了解了系统设计与 API 设计之间的关系，就来总结一下在本章中讨论的内容吧。

4.8. 总结

这一章中，了解了何时应用哪种服务模型，以及如何避免设计分布式系统的常见错误。了解了 CAP 定理，以及对分布式架构有哪些实际结果。现在可以在这样的系统中成功地运行事务，减少它们的停机时间，避免问题，并从容地从错误中恢复，处理异常高的负载不再是黑魔法。还可以将部分系统（甚至是遗留的部分）与新设计的部分进行集成。还掌握了一些技巧来提高系统的性能和扩展性。系统的部署和负载平衡也不再神秘，因此可以高效地执行。最后，了解了服务以及设计和管理它们的 API。

下一章中，将了解如何使用 C++ 特性，以一种更愉快、更有效的方式踏上通往卓越架构的道路。

4.9. 练习题

1. 什么是事件源？
2. CAP 定理的实际结果是什么？
3. Netflix 的 Chaos Monkey 能做什么？
4. 缓存可以应用在哪里？
5. 当整个数据中心宕机时，如何防止应用程序宕机？
6. 为什么使用 API 网关？
7. Envoy 如何实现各种架构目标？

4.10. 扩展阅读

- Microsoft Azure cloud design patterns: <https://docs.microsoft.com/en-us/azure/architecture/patterns/>
- Common design patterns for cloud APIs by Google: https://cloud.google.com/apis/design/design_patterns
- Microsoft REST API guidelines: <https://github.com/microsoft/apiguidelines/blob/vNext/Guidelines.md>
- Envoy Proxy's Getting Started page: <https://www.envoyproxy.io/docs/envoy/latest/start/start>
- Active-active application architectures with MongoDB: <https://developer.mongodb.com/article/active-active-application-architectures>

第 5 章 C++ 特性

C++ 是一种独特的语言，用于多种情况，从创建固件和操作系统、桌面和移动应用程序，到服务器软件、框架和服务。C++ 代码可以运行在各种硬件上，大量部署在计算云上，甚至可以在外太空找到。如果没有这种多范式语言所具有的特性，就不可能取得这样的成功。

本章描述了如何使用 C++ 语言提供的功能，来实现安全和高性能的解决方案。我们将演示类型安全、避免内存问题，以及以同样有效的方式创建高效代码的最佳行业实践。还将了解在设计 API 时如何使用某些语言特性。

本章将讨论以下内容：

- 将计算从运行时移到编译时
- 利用安全类型的力量
- 创建易于阅读和性能的代码
- 将代码划分为模块

在这段旅程中，将了解各种 C++ 标准（从 C++98 到 C++20）中可用的特性和技术。这将包括声明式编程、RAII、*constexpr*、模板、概念和模块。废话少说，赶快开始吧。

5.1. 相关准备

需要以下工具来构建本章中的代码：

- 支持 C++20 的编译器（推荐 GCC 11+）
- CMake 3.15+

本章的代码可以在以下 GitHub 页面找到：<https://github.com/PacktPublishing/Software-Architecture-with-Cpp/tree/master/Chapter05>。

5.2. 设计优秀的 API

尽管 C++ 允许使用面向对象 API（如果使用基于 coffee 的语言编写代码，可能会熟悉这些 API），但它还有其他一些诀窍，将在本节中提到。

5.2.1 使用 RAII

C API 和 C++ API 的主要区别是什么？这与多态或类本身无关，而是与 RAII 用法有关。

RAII 表示资源获取即初始化，实际上更多地是关于释放资源，而不是获取资源。来看一下用 C 和 C++ 编写的类似 API，来展示这个特性的实际应用情况：

```
1 struct Resource;
2
3 // C API
4 Resource* acquireResource();
5 void releaseResource(Resource *resource);
6
7 // C++ API
```

```
8 using ResourceRaii = std::unique_ptr<Resource, decltype(&releaseResource)>;
9 ResourceRaii acquireResourceRaii();
```

C++ API 是基于 C API 的，但并不总是如此。在 C++ API 中，不需要单独的函数来释放资源。由于 RAII 用法，当 ResourceRaii 对象超出作用域，就会自动释放。这样省去了手工管理资源的负担，而且不需要额外的成本。

更重要的是，不需要编写类——可以重用标准库的 *unique_ptr*，这是一个轻量级指针。确保所管理的对象总可以释放，并且只释放一次。

因为要管理一些特殊类型的资源，所以必须使用自定义删除类型。*acquireResourceRaii* 函数需要将实际的指针传递给 *releaseResource* 函数。如果只想在 C++ 中使用 C API，则不需要向用户公开。

这里需要注意的事情是，RAII 不仅仅用于管理内存：可以使用它处理任何资源的所有权，例如锁、文件句柄、数据库连接，以及在其 RAII 包装器超出作用域时应该释放的所有东西。

5.2.2 整理 C++ 中容器的接口

标准库实现是惯用的、性能好的 C++ 代码。如果想阅读一些有趣的模板代码，应该去 *std::chrono* 看一看，其演示了一些有用的技术，并用新的方法来进行实现。在扩展阅读部分可以找到有关 libstdc++ 实现的链接。

当涉及到库的其他位置时，快速浏览一下它的容器，会发现其接口与其他编程语言中的有所不同。为了说明这一点，来看一下来自标准库的一个简单的类，*std::array*，并对其进行分析：

```
1 template <class T, size_t N>
2 struct array {
3     // types:
4     typedef T& reference;
5     typedef const T& const_reference;
6     typedef /*implementation-defined*/ iterator;
7     typedef /*implementation-defined*/ const_iterator;
8     typedef size_t size_type;
9     typedef ptrdiff_t difference_type;
10    typedef T value_type;
11    typedef T* pointer;
12    typedef const T* const_pointer;
13    typedef reverse_iterator<iterator> reverse_iterator;
14    typedef reverse_iterator<const_iterator> const_reverse_iterator;
```

开始阅读类定义时，首先看到的是它为某些类型创建了别名。这在标准容器中很常见，并且在许多容器中别名的名称是相同的。这种情况的出现有几个原因，其中之一是——这样做可以节省开发人员花在挠头和试图理解创建者的意思，以及特定别名是如何命名的时间。另一个原因是类和库编写器的用户在编写自己的代码时，常常依赖于此类类型特征。如果容器不提供这样的别名，将使它与一些标准实用程序或类型特征一起使用变得更加困难，因此使用 API 的用户需要解决这个问题，甚至使用一个完全不同的类。

即使没有在模板中使用这些类型别名，也可以使用它们。函数参数和类成员字段依赖于这些类型并不罕见，如果正在编写一个其他人可以使用的类，请始终记住提供这些类型。例如，正在编写

一个分配器，许多使用者将依赖于特定类型别名的存在。

来看看 array 类会带来什么：

```
1 // no explicit construct/copy/destroy for aggregate type
```

std::array 没有构造函数的定义，包括复制/移动构造函数; 赋值运算符; 或析构函数。通常，在不必要的时候添加这样的成员实际上会损害性能。使用非默认构造函数 (和 $T() \{\}$) 是非默认的，而不是 $T() = default;$ ，类不再简单，也不再有简单的构造函数，这将阻止编译器对其进行优化。

看看类还有哪些声明：

```
1 constexpr void fill(const T& u);
2 constexpr void swap(array<T, N>&) noexcept(is_nothrow_swappable_v<T>);
```

可以看到两个成员函数，包括一个成员交换。通常，不依赖于 *std::swap* 的默认行为，提供我们自己的行为是有益的，例如在 *std::vector* 中，底层存储作为一个整体进行交换，而不是交换每个元素。当写一个成员 *swap* 函数时，一定要引入名为 *swap* 的自由函数，这样它就可以通过**参数依赖查找 (ADL)** 检测到，从而可以调用成员函数 *swap*。

关于 *swap* 函数，是有条件的 *noexcept*。如果存储的类型可以在不引发异常的情况下进行交换，则数组的交换也将是 *noexcept*。使用非异常抛出交换可以将类型作为成员存储的类的复制操作中呈现强异常安全的保证。

如下面的代码块所示，现在出现了一组大的函数，展示了许多类的另一个重要组件——迭代器：

```
1 // iterators:
2 constexpr iterator begin() noexcept;
3 constexpr const_iterator begin() const noexcept;
4 constexpr iterator end() noexcept;
5 constexpr const_iterator end() const noexcept;
6
7 constexpr reverse_iterator rbegin() noexcept;
8 constexpr const_reverse_iterator rbegin() const noexcept;
9 constexpr reverse_iterator rend() noexcept;
10 constexpr const_reverse_iterator rend() const noexcept;
11
12 constexpr const_iterator cbegin() const noexcept;
13 constexpr const_iterator cend() const noexcept;
14 constexpr const_reverse_iterator crbegin() const noexcept;
15 constexpr const_reverse_iterator crend() const noexcept;
```

迭代器对于每个容器都至关重要。如果不为类提供迭代器访问，就不能在基于范围的 for 循环中使用，而且也不能与标准库中所有有用的算法兼容。这并不意味着需要编写自己的迭代器类型——如果存储是连续的，只需使用简单的指针即可。提供 *const* 迭代器可以以不可变的方式使用类，而提供反向迭代器可以为容器提供更多用例。

继续往下看：

```
1 // capacity:
2 constexpr size_type size() const noexcept;
3 constexpr size_type max_size() const noexcept;
4 constexpr bool empty() const noexcept;
```

```

5
6 // element access:
7 constexpr reference operator[](size_type n);
8 constexpr const_reference operator[](size_type n) const;
9 constexpr const_reference at(size_type n) const;
10 constexpr reference at(size_type n);
11 constexpr reference front();
12 constexpr const_reference front() const;
13 constexpr reference back();
14 constexpr const_reference back() const;
15 constexpr T * data() noexcept;
16 constexpr const T * data() const noexcept;
17 private:
18 // the actual storage, like T elements[N];
19 };

```

迭代器之后，有几种方法来检查和修改容器的数据。`array` 中，它们是 `constexpr`。这意味着，如果要编写编译时代码，可以使用数组类。我们将在本章后面的编译时计算部分更多的进行了解。

最后，完成了 `array` 的定义。然而，从 C++17 开始，在类型定义之后，会发现如下代码：

```

1 template<class T, class... U>
2 array(T, U...) -> array<T, 1 + sizeof...(U)>;

```

这样的语句被称为演绎指南，是称为类模板参数演绎 (CTAD) 特性的一部分，这个特性在 C++17 中引入，允许在声明变量时忽略模板参数。其有利于 `array`，现在可以只写以下内容：

```

1 auto ints = std::array{1, 2, 3};

```

然而，对于更复杂的类型，例如 `map`，可能使用起来更方便：

```

1 auto legCount = std::unordered_map{ std::pair{"cat", 4}, {"human", 2}, {"mushroom", 1} };

```

然而，这里有一个 catch：当传递第一个参数时，需要指定传递的是键值对（注意，还使用了一个推导指引）。

既然正在讨论接口的主题，就再看看接口的其他方面。

5.2.3 接口中使用指针

接口中使用的类型非常重要。即使有文档，好的 API 也应该是一目了然的。看看向函数传递资源参数的不同方法，如何向 API 使用者提出不同的建议。

考虑下面的函数声明：

```

1 void A(Resource*);
2 void B(Resource&);
3 void C(std::unique_ptr<Resource>);
4 void D(std::unique_ptr<Resource>&);
5 void E(std::shared_ptr<Resource>);
6 void F(std::shared_ptr<Resource>&);

```

应该什么时候使用这些函数中的哪一个？

由于智能指针现在是处理资源的标准方式，*A* 和 *B* 应该留给简单的参数传递，如果不处理传递对象的所有权，就不应该使用。*A* 应该只用于单个资源，想传递多个实例，可以使用一个容器，例如 *std::span*。如果确定想传递的对象不为空，最好使用引用来传递它，比如 *const* 引用。如果对象不是太大，也可以考虑按值传递。

关于函数 *C* 到 *F* 的一个很好的经验法则是，如果想操作指针本身，应该只将智能指针作为参数传递；例如，用于转移所有权。

C 函数根据值接受 *unique_ptr*，这意味着它是一个资源池。换句话说，它先消耗资源，然后释放资源。请注意，仅仅通过选择特定的类型，接口就可以清楚地表达其意图。

只有想要传入包含资源的 *unique_ptr*，并接收另一个包含相同 *unique_ptr* 的资源作为输出参数时，才应该使用 *D* 函数。这并不是一个好主意，因为它要求调用者将其存储在一个 *unique_ptr* 中。换句话说，如果想传递一个 *const unique_ptr<Resource>*，换成传递一个 *Resource**（或 *Resource&*）会更好。

E 函数用于与被调用方共享资源所有权。通过值传递 *shared_ptr* 的开销相对比较大，因为需要增加它的引用计数器。然而，通过值传递 *shared_ptr* 是可以的，如果被调用方真的想成为共享的所有者，就必须在某处创建一个副本。

F 函数类似于 *D*，应该只在想要操作 *shared_ptr* 实例并通过这个 *in/out* 参数传播更改时使用。如果不确定函数是否应该拥有所有权，可以考虑传递 *const shared_ptr&*。

5.2.4 确定前置条件和后置条件

函数对参数有一些要求很正常，每一项要求都应作为先决条件。如果函数可以保证结果具有某些属性——例如，是非负的——该函数也应该明确这一点。一些开发人员使用注释来告知其他人，但并没有真正执行需求。使用 *if* 语句更好，但隐藏了检查的原因。目前，C++ 标准仍然没有提供一种方法来处理这个问题（契约第一次加入是在 C++20 标准，只是后面删除了）。幸运的是，像 Microsoft 的指南支持库（GSL）这样的库提供了自己实现的检查。

假设，出于某种原因，正在编写自己的队列。*push* 成员函数看起来像这样：

```
1 template<typename T>
2 T& Queue::push(T&& val) {
3     gsl::Expects(!this->full());
4     // push the element
5     gsl::Ensures(!this->empty());
6 }
```

注意，用户不需要访问实现来确保某些检查已经就绪。代码也是自解释的，因为函数需要什么，结果会是什么都很清楚。

5.2.5 使用内联名称空间

系统编程中，通常并不总是针对 API 编写代码，还需要关心 ABI 兼容性。一个著名的 ABI 崩溃发生在 GCC 发布第 5 个版本的时候，其中一个主要的变化是改变了 *std::string* 的布局。这意味着与旧的 GCC 版本一起工作的库（或者仍然在新版本中使用新的 ABI，这在最近的 GCC 版本中仍然是一种现象）不能与使用新版本 ABI 编写的代码一起工作。在 ABI 不同的情况下，如果收到一

个链接器错误，可以认为自己很幸运。某些情况下，例如混合 *NDEBUG* 代码和 *DEBUG* 代码。若类只有一个这样的配置中可用的成员，为了更好地调试而添加的成员，可能会出现内存崩溃。

使用 C++11 的内联名称空间时，一些通常很难调试的内存损坏很容易转化为链接器错误。考虑下面的代码：

```
1 #ifdef NDEBUG
2 inline namespace release {
3 #else
4 inline namespace debug {
5 #endif
6
7 struct EasilyDebuggable {
8 // ...
9 #ifndef NDEBUG
10 // fields helping with debugging
11 #endif
12 };
13
14 } // end namespace
```

由于前面的代码使用内联名称空间，当声明这个类的对象时，用户不会看到这两种构建类型之间的区别：来自内联名称空间的所有声明在周围的作用域中都是可见的。但是，链接器将以不同的符号名结束，这将导致链接器在试图链接不兼容的库时失败，从而提供了我们正在寻找的 ABI 安全性，并提供了一个很好的错误消息，其中提到了内联名称空间。

有关提供安全优雅 ABI 的更多信息，请参见 Arvid Norberg 的《C++ Now 2019》ABI 挑战演讲，链接在扩展阅读部分。

5.2.6 使用 `std::optional`

从 ABI 回到 API，再提一种在本书前面设计优秀 API 时忽略的类型。当涉及到函数的可选参数时，本节的主角可以挽救一切，因为它可以帮助类型拥有可能有值也可能没有值的组件，它还可以用于设计干净的接口或替代指针。它就是 `std::optional`，在 C++17 中标准化了。如果不能使用 C++17，可以在 Abseil(`absl::optional`) 中找到它，或者从 Boost(`Boost::optional`) 中找到非常类似的类型。使用该类的一个很大的优点是，非常清楚地表达了意图，这有助于编写干净和自文档化的接口。来看看实际情况。

可选的函数参数

首先将参数传递给可以（但可能不能）保存值的函数。曾经发现过类似于以下的函数签名吗？

```
1 void calculate(int param); // If param equals -1 it means "no value"
2 void calculate(int param = -1);
```

有时，如果 `param` 是在代码的其他地方计算的，那么很容易错误地传递-1——可能它是一个有效值。下面的签名呢？

```
1 void calculate(std::optional<int> param);
```

这一次，如果不想传递一个值，该做什么就更清楚了：只传递一个空的可选值。目的很明确，-1仍然可以用作有效值，而不必以一种类型不安全的方式赋予它特殊的含义。

这只是 optional 模板的一种用法。

optional 函数返回值

就像接受特殊值表示形参空值一样，函数有时也会空值返回。看看更喜欢下面哪一种？

```
1 int try_parse(std::string_view maybe_number);
2 bool try_parse(std::string_view maybe_number, int &parsed_number);
3 int *try_parse(std::string_view maybe_number);
4 std::optional<int> try_parse(std::string_view maybe_number);
```

如何知道第一个函数在出现错误时将返回什么值？或者会抛出一个异常，而不是返回一个魔值吗？继续看第二个签名，如果有错误，看起来会返回 `false`，但很容易忘记检查，并直接读取 `parsed_number`。第三种情况下，虽然假设错误时返回 `nullptr`，成功时返回一个整数相对安全，但现在还不清楚是否应该释放返回的整数。

对于最后一个签名，只要查看它，就可以清楚地看到在发生错误时将返回一个空值，并且不需要做任何操作。就是这么简单、易懂、优雅！

optional 的返回值也可以用来标记没有返回值，而不一定表示发生了错误。话虽如此，来看看 optional 的最后一个用例。

optional 类的成员

类状态中实现一致性并不总是一件容易的任务，有时希望有一两个成员不能简单地设置。可以使用可选的类成员，而不是为这种情况创建另一个类（这会增加代码的复杂性）或保留一个特殊的值（很容易不被注意地传递）。考虑以下类型：

```
1 struct UserProfile {
2     std::string nickname;
3     std::optional<std::string> full_name;
4     std::optional<std::string> address;
5     std::optional<PhoneNumber> phone;
6 };
```

可以看到哪些字段是必要的，哪些字段不需要填充。同样的数据可以用空字符串存储，但仅从结构体的定义来看并不明显。另一种选择是使用 `std::unique_ptr`，但是这样就会丢失局部性数据，而这对于性能来说是必不可少的。对于这种情况，`std::optional` 就非常合适。想要设计干净直观的 API 时，它绝对应该是开发者工具箱的一部分。

这些知识可以提供高质量和直观的 API。还可以继续改进它们，这也会使编写的代码出现更少的错误。我们将在下一节对此进行讨论。

5.3. 编写声明性代码

熟悉命令式编码风格和声明式编码风格吗？前者是代码告诉机器如何一步步实现想要的结果。后者是告诉机器想要达到的目标。某些编程语言支持其中一种。例如，C 语言是命令式的，而 SQL

是声明式的，就像许多函数式语言一样。有些语言可以混合——比如 C# 中的 LINQ。

C++ 是一种灵活的语言，可以以两种方式对代码进行编写。当编写声明性代码时，通常会保留更高层次的抽象，这会让错误出现的更少，以及更容易发现错误。那么，如何以声明的方式编写 C++ 呢？主要有两种方式。

第一种是编写函数式风格的 C++ 代码，如果更喜欢纯函数式风格（没有函数的副作用），应该尝试使用标准库算法，而不是手工编写循环。比如下面的代码：

```
1 auto temperatures = std::vector<double>{ -3., 2., 0., 8., -10., -7. };
2 // ...
3 for (std::size_t i = 0; i < temperatures.size() - 1; ++i) {
4     for (std::size_t j = i + 1; j < temperatures.size(); ++j) {
5         if (std::abs(temperatures[i] - temperatures[j]) > 5)
6             return std::optional{i};
7     }
8 }
9 return std::nullopt;
```

现在，比较上下两段代码，它们的功能相同：

```
1 auto it = std::ranges::adjacent_find(temperatures,
2                                     [] (double first, double second) {
3             return std::abs(first - second) > 5;
4 });
5 if (it != std::end(temperatures))
6     return std::optional{std::distance(std::begin(temperatures), it)};
7 return std::nullopt;
```

两个代码段都返回温度相对稳定的最后一天，更愿意喜欢哪个版本？哪一个更容易理解？即使现在对 C++ 算法不是很熟悉，但在代码中多次看到这些算法后，肯定会觉得它们比手工编写的循环更简单、更安全、更干净。

用 C++ 编写声明性代码的第二种方式在前面的代码段中已经有所体现，通常会更偏向使用声明性 API，比如 ranges 库中的 API。尽管在代码段中没有使用范围视图，也可以产生很大的不同。看看下面的代码段：

```
1 using namespace std::ranges;
2 auto is_even = [] (auto x) { return x % 2 == 0; };
3 auto to_string = [] (auto x) { return std::to_string(x); };
4 auto my_range = views::iota(1)
5     | views::filter(is_even)
6     | views::take(2)
7     | views::reverse
8     | views::transform(to_string);
9 std::cout << std::accumulate(begin(my_range), end(my_range), ""s) << '\n';
```

这是一个很好的声明性编码示例：只指定应该发生什么，而不是如何发生。前面的代码取前两个偶数，将它们的顺序颠倒，并将它们打印为字符串，从而打印出生命、宇宙和一切的答案：42。所有这些都是以一种直观且容易修改的方式完成。

5.3.1 展示特色商品

不过，简单例子到此为止。还记得在第 3 章的多米尼加集市应用程序吗？现在编写一个组件，选择并显示客户保存为其收藏夹的商店中的一些特色商品。例如，当编写移动应用程序时，这种功能非常方便。

先从一个 C++17 的实现开始，在本章中将其更新到 C++20。这包括添加对 range 的支持。

首先，从获取当前用户信息的代码开始：

```
1 using CustomerId = int;
2
3 CustomerId get_current_customer_id() { return 42; }
```

现在，让添加商店所有者：

```
1 struct Merchant {
2     int id;
3 };
```

商店还需要有商品：

```
1 struct Item {
2     std::string name;
3     std::optional<std::string> photo_url;
4     std::string description;
5     std::optional<float> price;
6     time_point<system_clock> date_added{};
7     bool featured{};
8 };
```

有些商品可能没有照片或价格，这就是为什么对这些字段使用 `std::optional` 的原因。

接下来，让添加一些代码来描述商品：

```
1 std::ostream &operator<<(std::ostream &os, const Item &item) {
2     auto stringify_optional = [](&const auto &optional) {
3         using optional_value_type =
4             typename std::remove_cvref_t<decltype(optional)>::value_type;
5         if constexpr (std::is_same_v<optional_value_type, std::string>) {
6             return optional ? *optional : "missing";
7         } else {
8             return optional ? std::to_string(*optional) : "missing";
9         }
10    };
11
12    auto time_added = system_clock::to_time_t(item.date_added);
13
14    os << "name: " << item.name
15    << ", photo_url: " << stringify_optional(item.photo_url)
16    << ", description: " << item.description
17    << ", price: " << std::setprecision(2)
18    << stringify_optional(item.price)
19    << ", date_added: "
```

```
20    << std::put_time(std::localtime(&time_added), "%c %Z")
21    << ", featured: " << item.featured;
22
23 }
```

首先，创建了一个辅助 lambda 将可选项转换为字符串。因为只想在<<操作符中使用它，所以在<<操作符中进行了定义。

注意如何使用 C++14 的泛型 lambdas(auto 参数)，以及 C++17 的 `constexpr` 和 `is_same_v` 类型特征，以便在处理 `std::optional<string>`和其他情况时，我们有不同的实现。实现同样的功能，使用 C++17 前需要编写带有重载的模板，从而出现更复杂的代码：

```
1 enum class Category {
2     Food,
3     Antiques,
4     Books,
5     Music,
6     Photography,
7     Handicraft,
8     Artist,
9 };
```

最后，定义存储：

```
1 struct Store {
2     gsl::not_null<const Merchant *> owner;
3     std::vector<Item> items;
4     std::vector<Category> categories;
5 };
```

这里值得注意的是使用 `gsl::not_null` 模板，表示所有者总是设置。为什么不使用一个普通的引用呢？这是因为希望商店是可移动和可复制的，使用引用可能会有麻烦。

现在，有了这些构建模块，来定义如何获得客户最喜欢的商店。简单起见，假设处理的是硬编码的商店和商家，而不是创建代码来处理外部数据存储。

首先，定义存储的类型别名，并开始函数定义：

```
1 using Stores = std::vector<gsl::not_null<const Store *>>;
2 Stores get_favorite_stores_for(const CustomerId &customer_id) {
```

接下来，对一些商家进行硬编码：

```
1 static const auto merchants = std::vector<Merchant>{{17}, {29}};
```

现在，让添加一个带有商品的商店：

```
1 static const auto stores = std::vector<Store>{
2     {.owner = &merchants[0],
3      .items =
4      {
5          {.name = "Honey",
6           .photo_url = {},
7           .description = "Straight outta Compton's apiary",
```

```

8     .price = 9.99f,
9     .date_added = system_clock::now(),
10    .featured = false},
11    {.name = "Oscypek",
12     .photo_url = {},
13     .description = "Tasty smoked cheese from the Tatra
14     mountains",
15     .price = 1.23f,
16     .date_added = system_clock::now() - 1h,
17     .featured = true},
18 },
19 .categories = {Category::Food},
20 // more stores can be found in the complete code on GitHub
21 };

```

这里用到了的第一个 C++20 的特性。可能不熟悉 `.field = value;` 语法，除非用过 C99 或更高标准的代码。从 C++20 开始，可以使用这种表示法（正式称为指定初始化器）来初始化聚合类型。它比 C99 的限制更加严格，因为顺序很重要。如果没有这些初始化器，就很难理解哪个值初始化哪个字段。使用时，代码会更冗长，但更容易理解，即使对不熟悉编程的人也是如此。

当定义了商店，就可以完成函数的最后一部分，进行查找：

```

1 static auto favorite_stores_by_customer =
2 std::unordered_map<CustomerId, Stores>{{42, {&stores[0],
3     &stores[1]}}};
4
5 return favorite_stores_by_customer[customer_id];
}

```

现在有了商店，可以编写一些代码来获取这些商店的特色商品：

```

1 using Items = std::vector<gsl::not_null<const Item *>>;
2 Items get_featured_items_for_store(const Store &store) {
3
4     auto featured = Items{};
5     const auto &items = store.items;
6     for (const auto &item : items) {
7
8         if (item.featured) {
9             featured.emplace_back(&item);
10        }
11    }
12
13    return featured;
14 }

```

前面的代码用于从一个存储中获取商品。再编写一个函数，从所有给定的商店获取商品：

```

1 Items get_all_featured_items(const Stores &stores) {
2
3     auto all_featured = Items{};
4     for (const auto &store : stores) {
5
6         const auto featured_in_store = get_featured_items_for_store(*store);
7         all_featured.reserve(all_featured.size() + featured_in_store.size());
8         std::copy(std::begin(featured_in_store), std::end(featured_in_store),
9             std::back_inserter(all_featured));
10    }
}

```

```
9     return all_featured;
10 }
```

上面的代码使用 `std::copy` 将元素插入到 `vector` 中，内存由 `reserve` 预先分配。

现在，有了一种获得特殊物品的方法，先按“新鲜度”对它们进行排序，这样最近添加的物品就会首先出现：

```
1 void order_items_by_date_added(Items &items) {
2     auto date_comparator = [] (const auto &left, const auto &right) {
3         return left->date_added > right->date_added;
4     };
5     std::sort(std::begin(items), std::end(items), date_comparator);
6 }
```

使用自定义比较器来使用 `std::sort`，还可以强制左对齐和右对齐使用相同的类型。为了以通用方式做到这一点，这里使用另一个 C++20 特性：模板 lambda。这里，把它们应用到前面的代码中：

```
1 void order_items_by_date_added(Items &items) {
2     auto date_comparator = []<typename T>(const T &left, const T &right) {
3         return left->date_added > right->date_added;
4     };
5     std::sort(std::begin(items), std::end(items), date_comparator);
6 }
```

`T` 的类型可以推导出来，就像其他模板一样。最后缺少的两个部分是实际的呈现代码和将它们粘合在一起的主要功能。例子中，呈现部分将像打印到 `ostream` 一样简单：

```
1 void render_item_gallery(const Items &items) {
2     std::copy(
3         std::begin(items), std::end(items),
4         std::ostream_iterator<gsl::not_null<const Item *>>(std::cout, "\n"));
5 }
```

本例中，将每个元素复制到标准输出中，并在元素之间插入一个换行符。使用 `copy` 和 `ostream_iterator` 处理元素的分隔符。这在某些情况下是很方便的；例如，如果不希望最后一个元素后面有逗号（或换行符，在例子中）。

最后，`main` 函数就像这样：

```
1 int main() {
2     auto fav_stores = get_favorite_stores_for(get_current_customer_id());
3
4     auto selected_items = get_all_featured_items(fav_stores);
5
6     order_items_by_date_added(selected_items);
7
8     render_item_gallery(selected_items);
9 }
```

运行代码，看看打印出的特色商品：

```

name: Handmade painted ceramic bowls, photo_url:
http://example.com/beautiful_bowl.png, description: Hand-crafted and hand-
decorated bowls made of fired clay, price: missing, date_added: Sun Jan 3
12:54:38 2021 CET, featured: 1
name: Oscypek, photo_url: missing, description: Tasty smoked cheese from
the Tatra mountains, price: 1.230000, date_added: Sun Jan 3 12:06:38 2021
CET, featured: 1

```

现在已经完成了基本实现，继续了解如何使用 C++20 的新语言特性来改进它。

5.3.2 标准中的 range

首先添加的是 `ranges` 库，它可以实现优雅、简单和声明性的代码。简洁起见，首先引入 `ranges` 命名空间：

```

1 #include <ranges>
2
3 using namespace std::ranges;

```

将保留定义商家、商品和商店的代码，使用 `get_featured_items_for_store` 函数进行修改：

```

1 Items get_featured_items_for_store(const Store &store) {
2     auto items = store.items | views::filter(&Item::featured) |
3         views::transform([](const auto &item) {
4             return gsl::not_null<const Item *>(&item);
5         });
6     return Items(std::begin(items), std::end(items));
7 }

```

在容器中创建 range 很简单：只需将其传递给管道操作符。可以使用 `views::filter` 表达式，将成员指针作为谓词传递给它。由于 `std::invoke` 的魔力，它将正确地过滤掉所有布尔数据成员设置为 `false` 的项。

接下来，需要将每个项目转换为一个 `gsl::not_null` 指针，这样就可以避免不必要的复制。最后，与基本代码一样，返回此类指针的 `vector`。

现在，来看看如何使用前面的函数从所有商店获得所有特色商品：

```

1 Items get_all_featured_items(const Stores &stores) {
2     auto all_featured = stores | views::transform([](auto elem) {
3         return get_featured_items_for_store(*elem);
4     });
5     auto ret = Items{};
6     for_each(all_featured, [&](auto elem) {
7         ret.reserve(ret.size() + elem.size());
8         copy(elem, std::back_inserter(ret));
9     });
10    return ret;

```

```
11 }
```

从所有存储中创建了一个 range，并使用在前面步骤中创建的函数对它们进行了转换。因为需要对每个元素解引用，所以使用了辅助 lambda。视图进行惰性求值，因此每个转换只有在即将使用时才会执行。这有时可以为节省大量的时间和计算：假设只需要前 N 个项目，可以跳过对 `get_featured_items_for_store` 不必要的调用。

有了惰性视图后，类似于现有的基实现，可以在 `vector` 中保留空间，并从 `all_featured` 视图中的每个嵌套 `vector` 中复制项。如果使用容器，range 算法使用起来会更简洁。看看 `copy` 是如何不要求使用 `std::begin(elem)` 和 `std::end(elem)`。

现在有了商品，简化排序代码，使用 range 来处理它们：

```
1 void order_items_by_date_added(Items &items) {
2     sort(items, greater{}, &Item::date_added);
3 }
```

同样，可以看到使用 range 如何编写更简洁的代码。前面的复制和排序都是与视图相反的 range 算法。这里使用投影可能会更合适。在例子中，只是传递了 `item` 类的另一个成员，以便在排序时使用它进行比较。实际上，每个项目将投影为它的 `date_added`，然后使用 `greater{}` 对其进行比较。

等等——项目实际上是指向项目的 `gsl::not_null` 指针，这是如何运作的呢？因为 `std::invoke` 很聪明，所以投影将首先解引用 `gsl::not_null` 指针。完美！

可以做的最后一个改变是在“呈现”代码中：

```
1 void render_item_gallery([[maybe_unused]] const Items &items) {
2     copy(items,
3         std::ostream_iterator<gsl::not_null<const Item *>>(std::cout,
4         "\n"));
5 }
```

这里，range 只删除了一些样板代码。当运行更新版本的代码时，应该得到与原始版本相同的输出。

如果期望 range 不仅仅是简洁的代码，那么有一个好消息：在示例中，可以更有效地使用。

使用 range 减少内存开销，增加性能

已经知道在 `std::ranges::views` 中使用惰性求值，可以通过消除不必要的计算来提高性能。在示例中，还可以使用 range 来减少内存开销。回顾一下从商店获取特色商品的代码，可以缩短为：

```
1 auto get_featured_items_for_store(const Store &store) {
2     return store.items | views::filter(&Item::featured) |
3         views::transform(
4             [](const auto &item) { return gsl::not_null(&item); });
5 }
```

注意，函数不再返回商品，而是依赖于 C++14 的自动返回类型推断。例子中，代码将返回一个惰性视图，而不是 `vector`。

如何为所有商店使用这段代码：

```
1 Items get_all_featured_items(const Stores &stores) {
```

```

2 auto all_featured = stores | views::transform([](auto elem) {
3     return get_featured_items_for_store(*elem);
4 }) |
5 views::join;
6 auto as_items = Items{};
7 as_items.reserve(distance(all_featured));
8 copy(all_featured, std::back_inserter(as_items));
9 return as_items;
10 }

```

因为前面的函数返回的是一个视图而不是 `vector`, 在调用 `transform` 之后, 最终会得到了一个视图的视图。这意味着可以使用另一个 `join` 的标准视图, 将嵌套视图连接到一个统一的视图中。

接下来, 使用 `std::ranges::distance` 来预分配目标 `vector` 中的空间, 然后再进行复制。有些 `range` 有限制, 在这种情况下, 可以调用 `std::ranges::size` 来代替。最后的代码只有一个对 `reserve` 的调用, 这应该会带来很好的性能提升。

这就结束了对代码 `range` 的介绍。接下来, 再讨论一个对 C++ 编程很重要的另一个话题。

5.4. 编译时计算

从 21 世纪初现代 C++ 的出现开始, C++ 编程更多地是在编译期间进行计算, 而不是将它们推迟到运行时, 在编译期间检测错误比较容易调试。类似地, 在程序启动之前就准备好结果, 要比之后再计算快。

起初有模板元编程, 但随着 C++11 的发展, 每一个新标准都为编译时计算带来了额外的特性: 类型特征, 诸如 `std::enable_if` 或 `std::void_t` 之类的构造, 或者 C++20 的构造, 都是只在编译时进行计算。

多年来改进的一个特性是 `constexpr` 关键字及其相关代码。C++20 确实改进和扩展了 `constexpr`。现在, 不仅可以编写常规的 `constexpr` 函数 (这是对 C++11 的表达式函数的极大改进), 而且还可以在其中使用动态分配和异常, 更不用说 `std::vector` 和 `std::string` 了!

甚至虚函数现在也可以是 `constexpr`: 重载解析照常进行, 但如果给定的函数是 `constexpr`, 则可以在编译时调用。

对标准算法进行的另一项改进是, 非并行版本可以在编译时代码中使用。下面的例子中, 可以用来检查给定的商人信息是否存在与容器中:

```

1 #include <algorithm>
2 #include <array>
3
4 struct Merchant { int id; };
5
6 bool has_merchant(const Merchant &selected) {
7     auto merchants = std::array{Merchant{1}, Merchant{2}, Merchant{3},
8         Merchant{4}, Merchant{5}};
9     return std::binary_search(merchants.begin(), merchants.end(), selected,
10        [] (auto a, auto b) { return a.id < b.id; });
11 }

```

代码正在对一组商家进行二分搜索，根据他们的 ID 进行排序。

为了深入了解代码及其性能，建议查看一下代码生成的汇编码。随着编译时计算和追逐性能的出现，有一个开发工具是 <https://godbolt.org>。可用于快速处理代码，以查看不同的架构、编译器、标志、库版本和实现如何影响生成的汇编码。

使用`-O3` 和`--std=c++2a` 编译标志，再使用 GCC trunk(在 GCC 11 正式发布之前) 测试了上述代码。在例子中，使用以下代码检查生成的汇编码：

```
1 int main() { return has_merchant({4}); }
```

在这里可以看到相应的几十行汇编: <https://godbolt.org/z/PYMTYx>.

请等一下——可以说在汇编码中有一个函数调用，也许可以内联它，这样它可以更好地优化？这是一个不错的想法。但现在，这里只是将汇编码进行内联 (参见:<https://godbolt.org/z/hPadxd>)。

现在，试着将签名改为以下形式：

```
1 constexpr bool has_merchant(const Merchant &selected)
```

`constexpr` 函数是隐式内联的，因此删除了该关键字。如果研究一下汇编码，就会发现发生了一些神奇的事情：搜索被优化掉了！可以在 <https://godbolt.org/z/v3hj3E> 看到，所有剩下的程序集如下所示：

```
main:  
    mov eax, 1  
    ret
```

编译器优化了代码，因此只剩下返回预先计算的结果。很真的很令人吃惊，对吧？

5.4.1 使用 `const` 来帮助编译器

编译器可以很好地优化，即使没有给 `inline` 或 `constexpr` 关键字。为了让实现性能良好，可以将变量和函数标记为 `const`。更重要的是，还可以避免代码中的错误。许多语言在默认情况下都有常量，这可以减少错误，更容易推理的代码，并获得更好的多线程性能。

尽管 C++ 默认使用可变变量，并且需要显式地输入 `const` 类型，但还是鼓励这样做。因为，这样可以及时停止在修改不应该修改的变量时，出现输入错误。

使用 `const`(或 `constexpr`) 代码是类型安全原理的一部分。接下来，就来简单了解一下。

5.5. 安全类型

C++ 严重依赖于类型安全代码机制，语言构造（如显式构造函数和转换操作符）已经融入语言中很长时间了，越来越多的安全类型在引入标准库。有 `optional` 避免引用空值，`string_view` 避免超出范围，和作为类型的安全 `package`，这里仅举几个例子。

通常，使用 C 风格的结构会导致类型不安全。例子就是 C 的强制转换，可以对应为 `const_cast`、`static_cast`、`reinterpret_cast`，或者两者之一与 `const_cast` 结合使用。若不小

心对 `const` 对象使用 `const_cast`, 则会出现未定义行为。从 `reinterpret_cast<T>` 中读取返回的内存地址也是会如此, 如果 `T` 不是对象的原始类型 (C++20 添加了 `std::bit_cast`)。如果使用 C++ 的类型转换, 这两种情况都更容易避免。

C 在类型方面太宽松了。幸运的是, C++ 为有问题的 C 结构体引入了许多类型安全的替代方案。用 `streams` 和 `std::format` 来代替 `printf` 等, 还有 `std::copy` 和其他算法来代替不安全的 `memcpy`。最后, 用模板代替函数使用 `void *` (需要性能方面付出代价)。在 C++ 中, 模板通过概念的特性获得了更多的安全性。

5.5.1 约束模板参数

概念可以改进代码的第一种方式是使其更通用。还记得需要更改容器类型, 从而导致在其他地方也发生了更改的情况吗? 如果没有将容器改为具有完全不同语义的容器, 并且必须以不同的方式使用, 则代码可能不够通用。

另外, 是否曾经编写过模板或在代码中加入 `auto`, 然后想知道如果改变了底层类型, 代码是否会崩溃?

概念是关于在操作类型上放置正确级别的约束, 限制模板可以匹配的类型, 并在编译时进行检查。假设写了以下内容:

```
1 template<typename T>
2 void foo(T& t) {...}
```

现在, 可以这样写:

```
1 void foo(std::swappable auto& t) {...}
```

这里, `foo()` 必须传递一个支持 `std::swap` 的类型才能工作。还记得一些匹配太多类型的模板吗? 以前, 可以使用 `std::enable_if`、`std::void_t` 或 `if constexpr` 来约束它们。然而, 编写 `enable_if` 语句有些麻烦, 会增加编译时间。这里, 由于概念的简洁性, 以及可以清晰地表达其意图, 概念再次发挥了作用。

C++20 中有几十个标准概念, 大多数存在于头文件`<concepts>`中, 可以分为四类:

- 核心语言概念, 如 `derived_from`、`integral`、`swappable` 和 `move_constructible`
- 比较概念, 比如布尔可测试、`equality_comparable_with` 和 `totally_ordered`
- 对象概念, 如可移动、可复制、半正则和正则
- 可调用概念, 如函数操作符、谓词和 `strict_weak_order`

其他的定义在`<iterator>`头文件中。这些可以分为以下几类:

- 间接可调用的概念, 例如 `indirect_binary_predicate` 和 `indirectly_unary_invocable`
- 常见的算法要求, 例如 `indirectly_swappable`, `permutable`, `mergeable` 和 `sortable`

最后, 可以在`<ranges>`头文件中找到一堆相关特性。包括 `range(duh)`、`continuous_range` 和 `view`。

如果这还不能满足需要, 可以声明自己的概念, 类似于标准定义中的方式。例如, `movable` 的概念是这样实现的:

```
1 template <class T>
2 concept movable = std::is_object_v<T> && std::move_constructible<T> &&
3 std::assignable_from<T&, T> && std::swappable<T>;
```

此外，如果查看 `std::swappable` 的实现，会看到以下内容：

```
1 template<class T>
2 concept swappable = requires(T& a, T& b) { ranges::swap(a, b); };
```

如果 `range::swap(a, b)` 编译了该类型的两个引用，类型 `T` 将是可交换的。

TIP

定义自己的概念时，请确保涵盖了语义需求。定义接口时指定和使用一个概念是对该接口的使用者作出的承诺。

简单起见，可以在声明中使用简写符号：

```
1 void sink(std::movable auto& resource);
```

为了可读性和类型安全，建议将 `auto` 与概念一起使用，以约束类型，并让使用者知道正在处理的对象的类型，以这种方式保留 `auto` 通用性的优点。当然，也可以在常规函数和 `lambda` 中使用它。

使用概念的好处是更短的错误信息。将编译错误的几十行代码，缩减为几行是常规操作。另一个好处是，可以尽可能多地使用概念。

现在，回到多米尼加博览会的例子。这一次，添加一些概念，看看它们如何改进原来的实现。

首先，`get_all_featured_items` 返回一个商品 `range`。这里，可以将概念添加到返回类型来实现：

```
1 range auto get_all_featured_items(const Stores &stores);
```

一切顺利！现在，让向这个类型添加另一个需求，当调用 `order_items_by_date_added` 时将强制执行，所以 `range` 必须是可排序的。

`std::sortable` 已经为 `range` 迭代器定义过了，为了方便起见，需要定义一个新的概念 `sortable_range`：

```
1 template <typename Range, typename Comp, typename Proj>
2 concept sortable_range =
3     random_access_range<Range> && std::sortable<iterator_t<Range>, Comp,
4 Proj>;
```

与标准库类似，可以接受比较器和投影（通过 `range` 引入）。满足 `random_access_range` 概念的类型要求（将由满足 `random_access_range` 概念的类型匹配），以及满足上述可排序概念的迭代器。就是这么简单！

定义概念时，还可以使用 `requires` 子句指定其他约束。如果想要 `range` 只存储一个 `date_added` 成员的元素，可以这样写：

```
1 template <typename Range, typename Comp>
2 concept sortable_indirectly_dated_range =
```

```
3 random_access_range<Range> &&std::sortable<iterator_t<Range>, Comp> &&
4 requires(range_value_t<Range> v) { { v->date_added } }; }
```

例子中，不需要对类型进行太多的约束，因为在使用概念和定义时，应该保留一定的灵活性，以便重用。

可以使用 `requires` 子句指定当类型满足某个概念的要求时，应该调用哪些代码，可以为每个子表达式返回的类型指定约束。例如，要定义一个可递增的数时，可以使用以下语句：

```
1 requires(I i) {
2     { i++ } -> std::same_as<I>;
3 }
```

现在有了自己的概念，可以用来重新定义 `order_items_by_date_added` 函数：

```
1 void order_items_by_date_added(
2 sortable_range<greater, decltype(&Item::date_added)> auto &items) {
3     sort(items, greater{}, &Item::date_added);
4 }
```

现在，编译器将检查传递给它的 `range` 是否可排序，并包含一个可以使用 `std::ranges::greater{}` 进行排序的 `date_added` 成员。

若在使用更受约束的概念，函数看起来会像这样：

```
1 void order_items_by_date_added(
2 sortable_indirectly_dated_range<greater> auto &items) {
3     sort(items, greater{}, &Item::date_added);
4 }
```

最后，来重写函数：

```
1 template <input_range Container>
2 requires std::is_same_v<typename Container::value_type,
3     gsl::not_null<const Item *>> void
4 render_item_gallery(const Container &items) {
5     copy(items,
6         std::ostream_iterator<typename Container::value_type>(std::cout,
7         "\n"));
8 }
```

这里，可以看到在模板声明中可以使用概念名来代替 `typename` 关键字。再下一行，可以看到 `requires` 关键字还可以用于根据特征进一步约束类型。如果不指定新概念，也很方便。

这就是概念！接下来编写一些模块化的 C++ 代码。

5.6. 编写模块化 C++

本章讨论的 C++ 的最后一个特性是模块。它们是对 C++20 的又一个补充，对代码的构建和划分有很大的影响。

C++ 使用 `#include` 已经有很长时间了。然而，这种文本形式的依赖包含有其缺陷，如下所示：

- 由于需要处理大量文本（即使是预处理后的 Hello World 也需要大约 50 万行代码），所以速度很慢。这将违反定义规则（ODR）。
- 顺序很重要，但不应该这样。这个问题是前一个问题的两倍，这还会导致循环依赖。
- 最后，很难封装只需要放在头文件中的内容。即使你把一些东西放在一个详细的命名空间中，也会有人使用它，正如 Hyrum 定律所预测的那样。

这正是模块入场的时候。它应该解决上述的缺陷，极大地加快构建速度，并在构建时提供更好的 C++ 可扩展性。使用模块，可以导出想要导出的内容，这将带来良好的封装。因为导入的顺序并不重要，所以包含依赖项的特定顺序也不是问题。

Note

不幸的是，在编写本文时，编译器对模块的支持仍然只是部分完成。这就是为什么只展示了 GCC 11 中已有的功能。遗憾的是，这意味着像模块分区这样的东西将不会在这里出现。

每个模块编译后，不仅会编译成目标文件，还会编译成模块接口文件。编译器可以快速知道一个给定模块包含什么类型和函数，而不是解析一个文件的所有依赖项。开发者所需要做的就是输入以下内容：

```
1 import my_module;
```

当 `my_module` 编译并可用，就可以使用了。模块本身应该定义在 `.cppm` 文件中，但 CMake 仍然不支持这些文件，所以最好暂时将它们的后缀改成 `.cpp`。

言归正传，回到多米尼加博览会示例，并展示如何在实践中使用它们。

首先，为客户代码创建第一个模块，从下面的指令开始：

```
1 module;
```

这个语句标志着，从此以后，这个模块中的所有东西都是私有的。这是放置包含和其他不会导出的内容的好地方。

接下来，必须指定导出的模块名称：

```
1 export module customer;
```

这是稍后用于导入模块的名称，并且这一行必须在导出的内容之前。现在，让指定模块导出实际的内容，并在定义前加上 `export` 关键字：

```
1 export using CustomerId = int;
2
3 export CustomerId get_current_customer_id() { return 42; }
```

完成了！我们的第一个模块可以使用了。让为商人创建另一个实例：

```
1 module;
2 export module merchant;
3 export struct Merchant {
4     int id;
5 };
```

与第一个模块非常相似，这里指定了要导出的名称和类型（与第一个模块的类型别名和函数相反）。也可以导出其他定义，比如模板。但是，使用宏时需要一些技巧，因为需要导入`<header_file>`才能看到。

顺便说一下，模块的一个优点是它们不允许宏传播到导入的模块中。这意味着当你写下面的代码时，模块不会定义 MY_MACRO：

```
1 #define MY_MACRO  
2 import my_module;
```

它有助于在模块中有决定论，因为它可以保护不破坏其他模块中的代码。

现在，为商店和商品定义第三个模块。这里不会讨论导出其他函数、枚举和其他类型，因为它与前两个模块没有区别。有趣的是模块文件的启动方式。首先，在私有模块部分包含需要的内容：

```
1 module;  
2  
3 #include <chrono>  
4 #include <iomanip>  
5 #include <optional>  
6 #include <string>  
7 #include <vector>
```

在 C++20 标准中，标准库头文件还不是模块，但这在不久的将来可能会改变。

现在，看看接下来会发生什么：

```
1 export module store;  
2 export import merchant;
```

这是有趣的部分。商店模块导入前面定义的商人模块，然后将其作为商店接口的一部分重新导出。如果模块是其他模块的外观，在不久的将来（也是 C++20 的一部分）的模块分区中，这将非常方便，可以将模块拆分为多个文件。其中一个可以包含以下内容：

```
1 export module my_module:foo;  
2 export template<typename T> foo() {}
```

正如前面所讨论的，然后模块的主文件导出：

```
1 export module my_module;  
2 export import :foo;
```

这就完成了本章所规划的模块和主要的 C++ 特性。现在，总结一下所了解的知识。

5.7. 总结

本章中，了解了许多 C++ 特性，以及对编写简洁、有表现力和性能的影响，还了解了如何提供正确的 C++ 组件接口。现在，可以应用诸如 RAI 之类的原则来编写不受资源泄漏影响的优雅代码。还知道如何利用 `std::optional` 之类的类型在接口中，更好地表达意图。

接下来，演示了如何使用泛型和模板 lambda 等特性，以及如何使用 `if constexpr` 来编写可用于多种类型的（更少）代码。现在，还能够使用指定的初始化器以清晰的方式定义对象。之后，了

解了如何使用标准范围以声明式风格编写简单的代码，如何使用 `constexpr` 编写可在编译时和运行时执行的代码，以及如何使用概念编写受约束的模板代码。

最后，演示了如何用 C++ 模块编写模块化代码。下一章中，将讨论如何设计 C++ 代码，以便在可用的习惯用法和模式的基础上进行构建。

5.8. 练习题

1. 如何确保打开的每个代码文件在不再使用时都是关闭的？
2. 什么时候应该在 C++ 代码中使用“裸”指针？
3. 什么是推演指引？
4. 什么时候应该使用 `std::optional` 和 `gsl::not_null`？
5. 范围算法与视图有何不同？
6. 定义函数时，除了指定概念名之外，如何约束类型？
7. `import X` 与 `import <X>` 有何不同？

5.9. 扩展阅读

- C++ Core Guidelines, the section on Concepts: <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-concepts>
- libstdc++’s implementation of `std::chrono`: <https://code.woboq.org/gcc/libstdc++-v3/include/std/chrono.html>

第 6 章 设计模式和 C++

C++不仅是一种面向对象的语言，也不仅提供动态多态性，所以 C++的设计不只有“四人帮”介绍的那些模式。本章中，将了解常用的 C++习惯用法和设计模式，以及应该在哪里使用。

本章将讨论以下内容：

- 编写风格一致的 C++
- 模板递归模式
- 创建对象
- C++中跟踪状态和访问对象
- 高效地处理内存

这是一个相当长的列表！别浪费时间了，马上开始吧。

6.1. 相关准备

本章的代码需要以下工具来构建和运行：

- 支持 C++20 的编译器
- CMake 3.15+

本章的代码可以在以下 GitHub 页面找到：<https://github.com/PacktPublishing/Software-Architecture-with-Cpp/tree/master/Chapter06>。

6.2. C++的惯用法

如果熟悉面向对象编程语言，那么一定听说过“四人帮”的设计模式。虽然可以用 C++实现（通常也是这样），但这种多范例语言通常采用不同的方法来实现相同的目标。如果想要击败所谓的基于 coffee 的语言（如 Java 或 C#）的性能，有时虚拟调度的代价太大了。很多情况下，需要在一开始就知道要处理什么类型的问题。如果是这样，可以使用该语言和标准库中提供的工具编写更多性能更好的代码。在众多习语中，先从了解语言习语开始我们的旅程吧！

根据定义，习语是在给定语言中反复出现的结构，是特定于该语言的表达式。以 C++为母语的人应该凭直觉知道它的习语。之前已经提到了智能指针，这是最常见的一种。现在来讨论一个类似的例子。

6.2.1 使用 RAII 保护自动化范围的退出操作

C++中最强大的表达式之一是关闭作用域的大括号。这是调用析构函数和产生 RAII 魔法的地方。要驯服这个咒语，不需要使用智能指针。所需要的是一个 RAII 守卫——一个对象，构建，对内存进行操作，销毁。这样，无论作用域是正常存在还是异常存在，工作都将自动完成。

最好的部分——甚至不需要从头编写一个 RAII 保护程序，各种库中已经存在经过良好测试的实现。如果正在使用 GSL，可以使用 `GSL::finally()`。考虑下面的例子：

```
1 using namespace std::chrono;
2 void self_measuring_function() {
```

```

3 auto timestamp_begin = high_resolution_clock::now();
4
5 auto cleanup = gsl::finally([timestamp_begin] {
6     auto timestamp_end = high_resolution_clock::now();
7     std::cout << "Execution took: " <<
8     duration_cast<microseconds>(timestamp_end - timestamp_begin).count() << "
9 us";
10 });
11
12 // perform work
13 // throw std::runtime_error{"Unexpected fault"};
14 }

```

这里，函数的开始处使用一个时间戳，在函数的结束处使用另一个时间戳。试着运行这个例子，看看取消注释 `throw` 语句会如何影响执行。这两种情况下，RAII 守卫将正确地打印执行时间（假设异常在某个地方捕获）。

现在让我们讨论一些更流行的 C++ 习惯性用法。

6.2.2 管理可复制性和可移动性

C++ 中设计新类型时，确定是否可复制和可移动很重要，更重要的是正确实现类的语义。

实现不可复制的类型

某些情况下，类的复制代价非常昂贵，另一种可能是由于切片而导致的错误。过去，防止此类对象复制的常见方法是使用 `noncopyable` 用法：

```

1 struct Noncopyable {
2     Noncopyable() = default;
3     Noncopyable(const Noncopyable&) = delete;
4     Noncopyable& operator=(const Noncopyable&) = delete;
5 };
6
7 class MyType : NonCopyable {};

```

但注意，这样的类也是不可移动的，尽管在阅读类定义时很容易忽略。更好的方法是显式地添加两个缺失的成员（移动构造函数和移动赋值操作符）。作为经验法则，当声明这些特殊成员函数时，总是声明所有的函数。这意味着从 C++11 开始，首选的方式将是编写以下代码：

```

1 struct MyTypeV2 {
2     MyTypeV2() = default;
3     MyTypeV2(const MyTypeV2 &) = delete;
4     MyTypeV2 & operator=(const MyTypeV2 &) = delete;
5     MyTypeV2(MyTypeV2 &&) = delete;
6     MyTypeV2 & operator=(MyTypeV2 &&) = delete;
7 };

```

这次，成员是直接在目标类型中定义的，没有使用帮助性质的 `NonCopyable` 类型。

遵守三五规则

讨论特殊成员函数时，还有一件事需要提到：如果不删除，而是提供自己的实现，很可能需要定义所有的成员函数，包括析构函数。这在 C++98 中称为三规（因为需要定义三个函数：复制构造函数、复制赋值操作符和析构函数），而由于 C++11 的移动操作，现在被五规（另外两个是移动构造函数和移动赋值操作符）所取代。应用这些规则可以减少资源管理问题。

坚持零原则

只需要使用所有特殊成员函数的默认实现，就不要声明它们。这是一个明确的信号，表明希望使用默认行为，也是最不容易混淆的。考虑以下类型：

```
1 class PotentiallyMisleading {
2 public:
3     PotentiallyMisleading() = default;
4     PotentiallyMisleading(const PotentiallyMisleading &) = default;
5     PotentiallyMisleading &operator=(const PotentiallyMisleading &) = default;
6     PotentiallyMisleading(PotentiallyMisleading &&) = default;
7     PotentiallyMisleading &operator=(PotentiallyMisleading &&) = default;
8     ~PotentiallyMisleading() = default;
9
10 private:
11     std::unique_ptr<int> int_;
12 };
```

即使默认了所有成员，该类仍然是不可复制的。因为它有一个不可复制的 `unique_ptr` 成员。幸运的是，Clang 会就此进行警告，但是 GCC 在默认情况下不会警告。更好的方法是应用零规则：

```
1 class RuleOfZero {
2     std::unique_ptr<int> int_;
3 };
```

现在有更少的样板代码，通过查看成员，很容易注意到它不支持复制。

说到复制，还有一个更重要的习语需要了解。在此之前，将讨论另一个习惯性用法，可以（也应该）用于实现第一个习惯性用法。

6.2.3 隐藏友元

隐藏友元是在声明为友元的类型体中定义的非成员函数。这使得除了使用依赖于参数的查找（ADL）之外，无法以其他方式调用这类函数，从而有效地进行隐藏。因为减少了编译器考虑的重载，所以可以加快编译速度。好处是，提供的错误消息比其他选项更短。有个有趣的属性是，如果先进行隐式转换，则不能调用，这可以避免意外转换。

虽然 C++ 中的友元方式通常不推荐使用，但对于隐藏好友，情况会有所不同。如果上一段的优点不足够，那么它们应该是实现特化点的首选方式。现在，可能想知道特化点是什么。简单地说，是库代码使用的可调用对象，用户可以特化它们的类型。标准库为这些对象保留了相当多的名称，例如 `begin`、`end`、反向变量和 `const` 变量、`swap`、`(s)size`、`(c)data` 和许多操作符等。如果决定为这些特化点中的一个提供自己的实现，那么其行为最好符合标准库的预期。

好了，理论到此为止。来看看如何在实践中使用隐藏友元来提供特化点特化。例如，创建一个简化的类来管理类型数组：

```
1 template <typename T> class Array {
2 public:
3     Array(T *array, int size) : array_{array}, size_{size} {}
4
5     ~Array() { delete[] array_; }
6
7     T &operator[](int index) { return array_[index]; }
8     int size() const { return size_; }
9
10    friend void swap(Array &left, Array &right) noexcept {
11        using std::swap;
12        swap(left.array_, right.array_);
13        swap(left.size_, right.size_);
14    }
15
16 private:
17     T *array_;
18     int size_;
19 };
```

这里定义了析构函数，这意味着还应该提供其他特殊的成员函数。这将在下一节使用隐藏友元交换实现。请注意，尽管在 `Array` 类的主体中声明了 `swap` 函数，但它仍然是一个非成员函数。它接受两个 `Array` 实例，但没有访问权限。

`std::swap` 使编译器首先在交换成员的名称空间中查找交换函数，如果没有找到，将返回到 `std::swap`。这称为两步 ADL 和回退习惯性用法，简称为两步，因为首先使 `std::swap` 可见，然后调用 `swap`。`noexcept` 关键字告诉编译器 `swap` 函数不会抛出异常，在某些情况下可以更快地生成代码。除了 `swap` 之外，也要用 `noexcept` 关键字标记默认构造函数和移动构造函数。

现在有了一个交换函数，可以对 `Array` 类应用另一个习惯性用法。

6.2.4 使用复制和交换习惯性用法提供异常安全

正如上一节提到的，因为 `Array` 类定义了析构函数，根据五规，它还应该定义其他特殊的成员函数。本节中，将了解一个习语，该习语可以在没有样板文件的情况下完成此任务，同时还添加了强大的异常安全性。

如果不熟悉异常安全级别，这里有一个函数和类型异常安全级别的简单介绍：

- 不保证：这是最基本的水平。在使用对象时抛出异常后，不能保证对象的状态。
- 基本异常安全：可能会有副作用，但对象不会泄漏任何资源，将处于有效状态，并将包含有效数据（不一定与操作之前相同）。自定义类型至少应该提供这个级别的异常安全。
- 强异常安全性：无副作用。对象的状态将与操作之前相同。
- 无抛出保证：操作总是成功的。如果在操作期间抛出异常，将在内部捕获并处理它，这样操作就不会在外部抛出异常。这些操作可以标记为 `noexcept`。

怎样才能一石二鸟，写出无模板的特殊成员，同时又提供强大的异常安全性呢？很简单，当有 swap 函数后，就可以用它来实现赋值操作符：

```
1 Array &operator=(Array other) noexcept {
2     swap(*this, other);
3     return *this;
4 }
```

在例子中，一个操作符就可以同时完成复制和移动赋值。在复制的情况下，通过值来获取参数，因此这是进行临时复制的地方。然后，要做的就是交换成员。这里不仅实现了强异常安全性，而且还能够不抛出赋值操作符的主体。但是，当复制发生时，仍然可以在调用函数之前抛出异常。在移动赋值的情况下，没有复制，因为按值获取将只获取移动的对象。

现在，定义复制构造函数：

```
1 Array(const Array &other) : array_{new T[other.size_]},
2 size_{other.size_} {
3     std::copy_n(other.array_, size_, array_);
4 }
```

因为它分配了内存，可以根据 T 来抛出。再来定义移动构造函数：

```
1 Array(Array &&other) noexcept
2     : array_{std::exchange(other.array_, nullptr)},
3 size_{std::exchange(other.size_, 0)} {}
```

这里，使用 std::exchange 以便初始化成员，清除其他成员，所有这些都在初始化列表中。构造函数没有声明，除非出于性能原因。例如，只有当元素不可移动构造时， std::vector 才能在增长时移动，否则将复制。

这样！这里已经创建了一个数组类，提供了强大的异常安全性，而且只需要很少的工作，也没有代码重复。

现在来处理另一个 C++ 习惯性用法，可以在标准库中的一些地方看到。

6.2.5 编写 niebroid

Niebloids 以 Eric Niebler 的名字命名，是一种函数对象类型，从 C++17 开始，标准将其用于定制点。随着第 5 章中标准范围的引入，就开始流行起来了。它们是 Niebler 在 2014 年首次提出的，目的是在不需要 ADL 的地方禁用 ADL，这样编译器就不会考虑来自其他名称空间的重载。还记得前几节中的两步习语吗？因为不方便且容易忘记，所以引入了自定义点对象的概念。本质上，这些是执行两步操作的函数对象。

如果库需要提供特化点，那么使用 niebloids 实现可能是个好主意。C++17 及以后引入的标准库中的所有特化点，都以这种方式实现是有原因的。即使只需要创建一个函数对象，也可以考虑使用 niebloids。它们提供了 ADL 的所有优点，同时减少了缺点。允许特化，并且与概念一起提供了一种自定义可调用对象的重载集的方法。还允许定制算法，这一切的代价是编写更冗长的代码。

本节中，将创建一个简单的范围算法，并将其实现为 niebroid。我们称它为 contains，因为它只返回一个布尔值，表示给定的元素是否在范围内。首先，让创建函数对象本身，首先声明其基于迭代器的调用操作符：

```

1 namespace detail {
2     struct contains_fn final {
3         template <std::input_iterator It, std::sentinel_for<It> Sent, typename T,
4             typename Proj = std::identity>
5             requires std::indirect_binary_predicate<
6                 std::ranges::equal_to, std::projected<It, Proj>, const T *> constexpr
7             bool
8             operator()(It first, Sent last, const T &value, Proj projection = {})
9         const {

```

看起来很冗长，但所有这些代码都是有目的的。使用 `final` 结构来帮助编译器生成更高效的代码。如果查看模板形参，会看到一个迭代器和一个哨兵——每个标准范围的基本构建块。哨兵通常是一个迭代器，可以与迭代器进行比较的半正则类型（半正则类型是可复制和可默认初始化的）。接下来，`T` 是要搜索的元素类型，而 `Proj` 表示投影——在比较之前应用于每个范围元素的操作（`std::identity` 的默认值只是将其输入作为输出传递）。

模板参数之后，还有对它们的要求，运算符可以比较投影值和搜索值是否相等。这些约束之后，只需指定函数参数即可。

来看看实现：

```

1 while (first != last && std::invoke(projection, *first) != value)
2     ++first;
3 return first != last;
4 }
```

即使没有使用标准范围，函数也可以工作；还需要一个重载的范围。其声明可如下：

```

1 template <std::ranges::input_range Range, typename T,
2     typename Proj = std::identity>
3 requires std::indirect_binary_predicate<
4     std::ranges::equal_to,
5     std::projected<std::ranges::iterator_t<Range>, Proj>,
6     const T *> constexpr bool
7 operator()(Range &&range, const T &value, Proj projection = {}) const {
```

这一次，使用满足 `input_range` 的概念、元素值和投影类型的类型作为模板参数。要求调用投影后的范围迭代器可以与 `T` 类型的对象比较是否相等，类似于前面的操作。最后，使用范围、值和投影作为重载的参数。

这个操作符的主体也非常简单：

```

1     return (*this)(std::ranges::begin(range), std::ranges::end(range),
2                     value,
3                     std::move(projection));
4     }
5 };
6 } // namespace detail
```

只需使用给定范围内的迭代器和哨兵调用前面的重载，同时传递值和未更改的投影。现在，对于最后一部分，不仅是 `contains_fn` 可调用对象，还需要提供一个包含 nieblloid：

```
1 inline constexpr detail::contains_fn contains{};
```

通过声明类型为 `contains` 的内联变量 `contains_fn`, 允许使用变量名调用 niebloid。现在, 调用它, 看看是否有效:

```
1 int main() {
2     auto ints = std::ranges::views::iota(0) | std::ranges::views::take(5);
3
4     return contains(ints, 42);
5 }
```

就是这样! ADL 约束函数会按预期的方式工作。

如果认为所有这些都有点过于冗长, 那么可能会对 `tag_invoke` 感兴趣, 它可能在将来的某个时候成为标准的一部分。请参阅扩展阅读部分, 以获得关于这个主题的论文和一个 YouTube 视频, 很好地解释了 ADL、niebloids、隐藏友元和 `tag_invoke`。

现在, 来了解一下另一个 C++的习惯性用法。

6.2.6 基于策略的设计风格

基于策略的设计是由 Andrei Alexandrescu 在他的著作《Modern C++ design》中首次提出的。虽然这本书出版于 2001 年, 但其中的许多观点至今仍在使用, 可以在本章末尾的扩展阅读部分找到链接。策略习语基本上是一个编译时等效“四人帮”的策略模式。如果需要编写具有可定制行为的类, 可以将其作为模板参数, 并将适当的策略作为模板参数。一个实际的例子可能是标准的分配器, 作为最后一个模板参数作为策略传递给许多 C++容器。

返回 `Array` 类, 并添加一个用于调试打印的策略:

```
1 template <typename T, typename DebugPrintingPolicy = NullPrintingPolicy>
2 class Array {
```

可以使用不打印任何内容的默认策略, `NullPrintingPolicy` 可以进行如下实现:

```
1 struct NullPrintingPolicy {
2     template <typename... Args> void operator()(Args...) {}  
3 };
```

不管给出什么参数, 都不会做任何事情。编译器将完全优化它, 所以当调试打印功能没有使用时, 不会支付任何开销。

如果希望类更详细一些, 可以使用不同的策略:

```
1 struct CoutPrintingPolicy {
2     void operator()(std::string_view text) { std::cout << text << std::endl;
3 }
4 };
```

这次, 将简单地将传递给策略的文本打印到 `cout`。还需要修改类来使用策略:

```
1 Array(T *array, int size) : array_{array}, size_{size} {
2     DebugPrintingPolicy{}("constructor");
3 }
```

```
4
5 Array(const Array &other) : array_{new T[other.size_]}, 
6 size_{other.size_} {
7     DebugPrintingPolicy{}("copy constructor");
8     std::copy_n(other.array_, size_, array_);
9 }
10 // ... other members ...
```

只需调用策略的 `operator()`，传递要打印的文本。因为策略是无状态的，所以可以在每次需要使用时实例化，并且不需要额外的成本。另一种方法是调用静态函数。

现在，需要做的就是实例化 `Array` 类，使用所需的策略：

```
1 Array<T, CoutPrintingPolicy>(new T[size], size);
```

使用编译时间策略的缺点是，使用不同策略的模板实例化具有不同的类型。这意味着需要更多的工作，例如：从常规的 `Array` 类分配给一个具有 `CoutPrintingPolicy` 的类。为此，需要将赋值操作符实现为模板函数，并将策略作为模板参数。

有时，使用策略的另一种选择是使用特征。例如，以 `std::iterator_traits` 为例，在编写使用迭代器的算法时，可以用它们来定义迭代器的各种信息。一个例子是 `std::iterator_traits<T>::value_type`，既可以用于定义 `value_type` 成员的自定义迭代器，也可以用于简单的指针（这种情况下，`value_type` 将指向指向的类型）。

基于策略的设计的内容已经够多了。列表中的下一个，是可以应用于多种场景的强大习惯性用法。

6.3. 模板递归模式

尽管名称中有模式，但模板递归模式 (CRTP) 是 C++ 中的一种习惯性用法。可以用来实现其他的习惯性用法和设计模式，以及使用静态多态性。

6.3.1 动态多态性和静态多态性

提到多态性，许多程序员会想到动态多态性，即在运行时收集执行函数调用所需的信息，而静态多态性是关于在编译时确定调用。前者的优点是可以在运行时修改类型列表，允许使用插件和库扩展类的层结构。第二种方法的最大优点是，若事先知道类型，可以获得非常好的性能。当然，第一种情况下，有时可以期望编译器对调用进行反虚拟化，但不能总期望会这样做。在第二种情况下，编译时间可能会更长。

不可能在所有情况下达到双赢，为类型选择正确的多态性类型仍有很长的路要走。如果性能受到威胁，强烈建议考虑静态多态性。CRTP 是一个使用静态多态性的习惯性用法。

许多设计模式都可以以各种方式实现。由于动态多态性的代价并不总合适，四人帮设计模式通常不是 C++ 中的最佳解决方案。如果类型层结构需要在运行时进行扩展，或者编译时间是一个比性能重要的问题（并且不打算很快使用模块），那么四人帮模式的经典实现可能是一个很好的选择。否则，可以尝试使用静态多态或以更简单的 C++ 为中心的解决方案来实现，其中一些将在本章中描述。关键在于，选择最适合的工具完成这项工作。

6.3.2 实现静态多态性

实现静态多态类层结构。需要一个基模板类：

```
1 template <typename ConcreteItem> class GlamorousItem {
2     public:
3         void appear_in_full_glory() {
4             static_cast<ConcreteItem *>(this)->appear_in_full_glory();
5         }
6     };
7 }
```

基类的模板形参是派生类。这看起来可能很奇怪，但可以静态地将接口函数转换为正确的类型。本例中，名为 `appear_in_full_glory`。然后在派生类中调用该函数的实现，派生类可以这样实现：

```
1 class PinkHeels : public GlamorousItem<PinkHeels> {
2     public:
3         void appear_in_full_glory() {
4             std::cout << "Pink high heels suddenly appeared in all their beauty\n";
5         }
6     };
7
8 class GoldenWatch : public GlamorousItem<GoldenWatch> {
9     public:
10        void appear_in_full_glory() {
11            std::cout << "Everyone wanted to watch this watch\n";
12        }
13    };
14 }
```

这些类都使用自身作为模板参数派生自基类 `GlamorousItem`，还实现了所需的功能。

请注意，与动态多态性不同，CRTP 中的基类是一个模板，因此每个派生类将获得不同的基类型。从而，很难创建 `GlamorousItem` 基类的容器。不过，还可以做以下几件事：

- 存储在元组中。
- 创建派生类的 `std::variant`。
- 添加一个公共类来包装 Base 的所有实例化，可以为这个版本使用 `variant`。

第一种情况下，可以像下面这样使用这个类。首先，创建基类的实例元组：

```
1 template <typename... Args>
2 using PreciousItems = std::tuple<GlamorousItem<Args>...>;
3
4 auto glamorous_items = PreciousItems<PinkHeels, GoldenWatch>{};
```

类型别名元组将能够存储 `GlamorousItem`。现在，需要做的就是调用这个函数：

```
1 std::apply(
2     []<typename... T>(GlamorousItem<T>... items) {
3         (items.appear_in_full_glory(), ...); },
4     glamorous_items);
```

这里试图迭代一个元组，所以最简单的方法是使用 `std::apply`，在给定元组的元素上调用给定的可调用对象。例子中，可调用对象是一个只接受 `GlamorousItem` 基类的 lambda。这里使用了 C++17 中引入的折叠表达式，以确保所有元素都能调用这个函数。

如果需要使用一个变量来代替元组，可以使用 `std::visit`，就像这样：

```
1 using GlamorousVariant = std::variant<PinkHeels, GoldenWatch>;
2 auto glamorous_items = std::array{GlamorousVariant{PinkHeels{}}, 
3     GlamorousVariant{GoldenWatch{}}};
4 for (auto& elem : glamorous_items) {
5     std::visit([]<typename T>(GlamorousItem<T> item){
6         item.appear_in_full_glory(); }, elem);
7 }
```

`std::visit` 函数基本上接受 `std::variant`，并在其中存储的对象上调用传入的 lambda。这里，创建了一个 `GlamorousVariant` 数组，可以像对其他容器一样对其进行迭代，使用适当的 lambda 访问每个变量。

如果从用户接口的角度编写不是很直观，考虑下一个方法，将 `variant` 包装到另一个类中，在例子中称为 `CommonGlamorousItem`：

```
1 class CommonGlamorousItem {
2 public:
3     template <typename T> requires std::is_base_of_v<GlamorousItem<T>, T>
4     explicit CommonGlamorousItem(T &&item)
5         : item_{std::forward<T>(item)} {}
6 private:
7     GlamorousVariant item_;
8 };
```

为了构造包装器，使用转发构造函数（模板化的 `T&&` 是它的参数）。然后向前移动，而不是移动创建 `item_ wrapped` 的 `variant`，因为这样只移动右值输入。这里还要约束模板参数，因此一方面需要包装 `GlamorousItem` 基类，另一方面模板不可用作移动或复制构造函数。

还需要包装成员函数：

```
1 void appear_in_full_glory() {
2     std::visit(
3         []<typename T>(GlamorousItem<T> item) {
4             item.appear_in_full_glory(); },
5             item_);
6 }
```

这次，是使用 `std::visit` 的一个技巧。用户可以按以下方式使用这个包装器类：

```
1 auto glamorous_items = std::array{CommonGlamorousItem{PinkHeels{}},
2     CommonGlamorousItem{GoldenWatch{}}};
3 for (auto& elem : glamorous_items) {
4     elem.appear_in_full_glory();
5 }
```

这种方法可以让类的用户编写易于理解的代码，并依旧保持静态多态性的性能。

为了提供类似的用户体验，尽管性能较差，还可以使用 **类型擦除**。

6.3.3 使用类型擦除

尽管类型擦除与 CRTP 无关，但与当前的示例非常适配，也是在这里展示它的原因。

类型擦除习惯性用法是关于在多态接口下隐藏具体类型。Sean Parent 在 GoingNative 2013 大会上的演讲《Inheritance Is The Base Class of Evil》就是一个很好的例子。强烈推荐在业余时间观看，可以在扩展阅读部分找到链接。在标准库中，可以在 `std::function`, `std::shared_ptr` 的删除器或 `std::any` 中找到它。

方便性和灵活性需要付出代价——需要使用指针和虚拟调度，这使得标准库中提到的实用程序在面向性能的用例中不适合使用。

为了在示例中引入类型擦除，不再需要 CRTP，`GlamorousItem` 类将把动态多态对象包装在一个智能指针中：

```
1 class GlamorousItem {
2 public:
3     template <typename T>
4     explicit GlamorousItem(T t)
5     : item_{std::make_unique<TypeErasedItem<T>>(std::move(t))} {}
6
7     void appear_in_full_glory() { item_->appear_in_full_glory_impl(); }
8
9 private:
10    std::unique_ptr<TypeErasedItemBase> item_;
11};
```

这次，存储一个指向基类 (`TypeErasedItemBase`) 的指针，指向派生包装 (`TypeErasedItem<T>`)。基类可以有如下定义：

```
1 struct TypeErasedItemBase {
2     virtual ~TypeErasedItemBase() = default;
3     virtual void appear_in_full_glory_impl() = 0;
4 };
```

每个派生包装器也需要实现这个接口：

```
1 template <typename T> class TypeErasedItem final : public
2 TypeErasedItemBase {
3 public:
4     explicit TypeErasedItem(T t) : t_{std::move(t)} {}
5     void appear_in_full_glory_impl() override { t_.appear_in_full_glory(); }
6
7 private:
8     T t_;
9 };
10};
```

基类的接口通过从包装对象调用函数来实现。注意，这种方式称为“类型擦除”，因为 `GlamorousItem` 类并不知道实际包装的是什么 `T`。信息类型在项构造时删除，但它仍然可以工作，因为 `T` 实现了所需的方法。

具体项目可以以更简单的方式实现，如下所示：

```

1 class PinkHeels {
2     public:
3     void appear_in_full_glory() {
4         std::cout << "Pink high heels suddenly appeared in all their beauty\n";
5     }
6 };
7
8 class GoldenWatch {
9     public:
10    void appear_in_full_glory() {
11        std::cout << "Everyone wanted to watch this watch\n";
12    }
13 };

```

不需要从任何基类继承，只需要鸭子类型——如果它叫得像鸭子，那很可能就是鸭子。

类型擦除 API 可以这样用：

```

1 auto glamorous_items =
2     std::array{GlamorousItem{PinkHeels{}}, GlamorousItem{GoldenWatch{}}};
3 for (auto &item : glamorous_items) {
4     item.appear_in_full_glory();
5 }

```

只创建一个包装器数组并对其进行迭代，所有这些都使用基于值的语义。因为多态性作为实现细节对调用者是隐藏的，所以使用它没什么难度。

然而，这种方法的一个大缺点是性能差。类型擦除是有代价的，所以应该谨慎使用，绝对不要在热路径中使用。

既然已经描述了如何包装和删除类型，继续来讨论如何创建类型。

6.4. 创建对象

本节中，将讨论与对象创建相关的问题和常见解决方案。将讨论各种类型的对象工厂，浏览构建器，并接触组合和原型。然而，在描述解决方案时，将采用与四人帮不同的方法。提出了复杂的、动态多态的类层结构作为模式的实现。C++许多模式都可以解决实际问题，而不需要引入太多的类和动态调度的开销。这就是为什么在例子中，实现是不同的，并且在许多情况下更简单或性能更好（四人帮会更特化和更不“通用”）。让我们开始吧！

6.4.1 何为工厂

第一种创造模式是工厂。当对象构造可以在单个步骤中完成时（如果不能在工厂之后立即覆盖该模式，则该模式很有用），但当构造函数不够方便时，这种方式就很有用。工厂有三种类型——工厂方法、工厂函数和工厂类。

工厂方法

工厂方法，也称为命名构造函数习惯性用法，基本上是调用私有构造函数的成员函数。什么时候使用？以下是一些场景：

- 当有许多不同的方法来构造一个对象时，这很可能会导致错误。例如为给定的像素构造一个存储不同颜色通道的类，每个通道由一个字节值表示。只使用一个构造函数将很容易传递错误的通道顺序，或意味着完全不同的调色板的值。此外，切换像素的内部颜色表示也会变得棘手。有读者可能会争辩说，应该用不同的类型来表示这些不同格式的颜色，但使用工厂方法也是一种很有效的方法。
- 当希望强制在堆上或在另一个特定内存区域中创建对象时。如果对象占用了堆栈上的大量空间，并且担心会耗尽堆栈内存，那么使用工厂方法是一个解决方案。如果要求在设备的某个内存区域中创建所有实例，情况也是如此。
- 构造对象时可能会失败，但不能抛出异常。应该使用异常而不是其他错误处理方法。如果使用得当，可以生成更清晰、性能更好的代码。然而，一些项目或环境要求禁用异常。这种情况下，使用工厂方法可以报告在构造期间发生的错误。

描述的第一种情况的工厂方法如下：

```
1 class Pixel {  
2     public:  
3         static Pixel fromRgba(char r, char b, char g, char a) {  
4             return Pixel{r, g, b, a};  
5         }  
6         static Pixel fromBgra(char b, char g, char r, char a) {  
7             return Pixel{r, g, b, a};  
8         }  
9         // other members  
10  
11     private:  
12         Pixel(char r, char g, char b, char a) : r_(r), g_(g), b_(b), a_(a) {}  
13         char r_, g_, b_, a_;  
14 }
```

这个类有两个工厂方法（实际上，C++标准不识别术语方法，而是将其称为成员函数）：`fromRgba` 和 `fromBgra`。现在就很难以错误的顺序初始化通道了。

注意，拥有私有构造函数可以有效地阻止其他类进行继承，因为如果不访问其构造函数，就不能创建实例。然而，如果这是目标，而不是副作用，应该更倾向于把类型标记为 `final`。

工厂函数

与使用工厂成员函数相反，也可以使用非成员函数来实现。这样，就可以提供更好的封装，正如 Scott Meyers 的文章中所述，文章链接在扩展阅读部分可以找到。

在 `Pixel` 类中，还可以创建一个自由函数来创建实例：

```
1 struct Pixel {  
2     char r, g, b, a;  
3 };  
4  
5 Pixel makePixelFromRgba(char r, char b, char g, char a) {
```

```

6   return Pixel{r, g, b, a};
7 }
8
9 Pixel makePixelFromBgra(char b, char g, char r, char a) {
10  return Pixel{r, g, b, a};
11 }

```

这种方法会使设计符合第 1 章中描述的开闭原则。很容易为其他调色板添加更多的工厂功能，而不需要修改 Pixel 本身。

Pixel 的这个实现允许用户手工初始化，而不是使用提供的函数，可以通过更改类声明来抑制这种行为。以下是修复后的处理方法：

```

1 struct Pixel {
2     char r, g, b, a;
3
4 private:
5     Pixel(char r, char g, char b, char a) : r(r), g(g), b(b), a(a) {}
6     friend Pixel makePixelFromRgba(char r, char g, char b, char a);
7     friend Pixel makePixelFromBgra(char b, char g, char r, char a);
8 };

```

这次，工厂函数是类的友元函数。但该类型不再是聚合类型，因此不能使用聚合初始化(Pixel{})，包括指定的初始化方式，所以也放弃了开闭原理。这两种方法提供了不同的权衡，所以要明智地选择。

选择工厂的返回类型

实现对象工厂时，应该选择的另一件事是应该返回的实际类型。

Pixel 是一个值类型，而不是一个多态类型，最简单的方法效果最好——简单地按值返回。如果生成了一个多态类型，通过一个智能指针返回它(永远不要使用裸指针，可能在某些时候产生内存泄漏)。如果调用者拥有创建的对象，通常将其以 unique_ptr 形式返回给基类是最好的方法。在不太常见的情况下，工厂和调用方都必须拥有对象，使用 shared_ptr 或其他引用计数替代。有时候，工厂跟踪对象而不存储。这种情况下，在工厂内部存储 weak_ptr，并在工厂外部返回 shared_ptr。

一些 C++ 程序员可能会争辩说，应该使用 out 参数返回特定的类型，但在大多数情况下，这不是最好的方法。性能方面，按值返回通常是最好的选择，因为编译器不会生成对象的副本。如果问题是类型不可复制，C++17 标准指定在哪些地方必须省略复制，因此按值返回此类类型通常不是问题。如果函数返回多个对象，请使用 pair、tuple、结构体或容器。

如果在构建过程中出现问题，这里有几个选择：

- 不需要向调用者提供错误消息，则返回类型为 std::optional。
- 构造过程中很少出现错误并且应该传播错误，则抛出异常。
- 在构造过程中经常出现错误，则返回类型为 absl::StatusOr(请参阅 Abseil 关于该模板的文档，详见扩展阅读部分)。

既然知道了该返回什么类型，接着讨论最后一个——工厂类。

工厂类

工厂类是可以制造对象的类型，可以将多态对象类型与其调用者解耦。允许使用对象池（其中保存了可重用对象，不需要经常分配和释放）或其他分配方案。仔细看看另一个种方式，假设需要根据输入参数创建不同的多态类型。某些情况下，一个多态工厂函数是不够的：

```
1 std::unique_ptr<IDocument> open(std::string_view path) {
2     if (path.ends_with(".pdf")) return std::make_unique<PdfDocument>();
3     if (name == ".html") return std::make_unique<HtmlDocument>();
4
5     return nullptr;
6 }
```

如果想打开其他类型的文档，比如 OpenDocument 文本文件，怎么办？发现前面的开放式工厂并不是开放用于扩展，这非常具有讽刺意味。如果拥有代码库，这可能不是一个大问题，但如果库的使用者需要注册自己的类型，这可能就有问题了。为了解决这个问题，可以使用工厂类，它允许注册函数打开不同类型的文档：

```
1 class DocumentOpener {
2 public:
3     using DocumentType = std::unique_ptr<IDocument>;
4     using ConcreteOpener = DocumentType (*) (std::string_view);
5
6 private:
7     std::unordered_map<std::string_view, ConcreteOpener> openerByExtension;
8 };
```

这个类还需要完成很多事情，但它有一个扩展到函数的 map，这些函数可以用来打开给定类型的文件。现在添加两个公共成员函数，第一个将注册新的文件类型：

```
1 void Register(std::string_view extension, ConcreteOpener opener) {
2     openerByExtension.emplace(extension, opener);
3 }
```

现在有了填充 map 的方法。第二个新的公共功能将使用适当的 opener 打开文件：

```
1 DocumentType open(std::string_view path) {
2     if (auto last_dot = path.find_last_of('.'));
3     last_dot != std::string_view::npos) {
4         auto extension = path.substr(last_dot + 1);
5         return openerByExtension.at(extension)(path);
6     } else {
7         throw std::invalid_argument{"Trying to open a file with no
8             extension"};
9     }
10 }
```

从文件路径中提取扩展名，如果为空，则抛出异常。如果不是，则在 map 中查找一个 opener。如果找到，则使用它打开给定的文件，如果没有找到，则 map 将抛出另一个异常。

现在可以实例化工厂，并注册自定义文件类型，例如 OpenDocument 文本格式：

```

1 auto document_opener = DocumentOpener{};
2
3 document_opener.Register(
4     "odt", [](auto path) -> DocumentOpener::DocumentType {
5         return std::make_unique<OdtDocument>(path);
6     });

```

注意，这里注册了一个 lambda，因为它可以转换为 `ConcreteOpener` 类型，这是一个函数指针。如果有状态，就不是这样了。这种情况下，需要一些包装。其中之一就是 `std::function`，但是这样做的缺点是每次运行该函数时吗，都需要类型擦除（性能降低）。在打开文件的情况下，这可能是可以的。但是，如果需要更好的性能，可以考虑使用 `function_ref` 这样的类型。

在 Sy Brand 的 GitHub repo 上可以找到一个针对 C++ 标准（尚未接受）的实现示例，参见扩展阅读部分。

现在在工厂中注册了 opener，用它来打开一个文件并从中提取一些文本：

```

1 auto document = document_opener.open("file.odt");
2 std::cout << document->extract_text().front();

```

若希望为库的消费者提供一种注册自己类型的方式，那么必须在运行时访问 map。可以提供一个 API 来进行访问，或者使工厂成为静态的，并允许在任何地方进行注册。

对于工厂和创建对象来说，这一步就完成了。然后，讨论一下若工厂不适合的话，可以使用的另一种方式。

6.4.2 使用构建器

构建器类似于工厂，这是来自“四人帮”的一种设计模式。与工厂不同，它们可以构建更复杂的对象：那些不能一步完成的对象，比如由许多独立部件组装而成的类型，还提供了一种自定义对象构造的方法。本例中，将跳过设计复杂构建器的层结构。从而，展示构建器如何提供帮助。这里会把层结构的实现留给读者们作为练习。

当无法在单个步骤中生成对象时，就需要构建器。若单步没法完成，那么使用连贯的接口可以让它们变得简单。让我们演示如何使用 CRTP 创建流畅的构建器层次结构。

在示例中，我们将创建一个 CRTP，使用 `GenericItemBuilder` 作为基本构建器，以及使用 `FetchingItemBuilder` 作为一个专用的构建器，可以使用远程地址获取数据（如果远程地址是受支持的特性）。这种特化甚至可以存在于不同的库中，例如：使用不同的 API，这些 API 在构建时可能可用，也可能不可用。

为了演示目的，将构建第 5 章中 `Item` 结构体的实例，利用 C++ 语言特性：

```

1 struct Item {
2     std::string name;
3     std::optional<std::string> photo_url;
4     std::string description;
5     std::optional<float> price;
6     time_point<system_clock> date_added{};
7     bool featured{};
8 };

```

可以通过将默认构造函数设为私有，并将构造函数设为友元的方式，来强制使用构造函数来构建 Item 实例：

```
1 template <typename ConcreteBuilder> friend class GenericItemBuilder;
```

构建器的实现如下所示：

```
1 template <typename ConcreteBuilder> class GenericItemBuilder {
2 public:
3     explicit GenericItemBuilder(std::string name)
4         : item_{name = std::move(name)} {}
5 protected:
6     Item item_;
```

虽然不建议创建受保护的成员，但希望后代构建器能够访问父类。另一种方法是在派生类中只使用基类构建器的公共方法。

在构造器的构造函数中接受名称，因为它是来自用户的一个输入，需要在创建项目时设置。这样，可以确保名称会进行设置。另一种选择是在构建的最后阶段，即对象释放给用户时，检查它是否合适。在例子中，构建步骤的实现如下：

```
1 Item build() && {
2     item_.date_added = system_clock::now();
3     return std::move(item_);
4 }
```

当调用该方法时，强制构造器被“消耗”，必须是右值。这意味着既可以在一行中使用构建器，也可以在最后一步中移动它，以标记工作的结束。然后，设置项目的创建时间，并将其移动到构建器之外。

构建器 API 可以提供如下功能：

```
1 ConcreteBuilder &&with_description(std::string description) {
2     item_.description = std::move(description);
3     return static_cast<ConcreteBuilder &&>(*this);
4 }
5
6 ConcreteBuilder &&marked_as_featured() {
7     item_.featured = true;
8     return static_cast<ConcreteBuilder &&>(*this);
9 }
```

它们将具体（派生）构建器对象作为右值引用返回。也许与直觉相反，这次应该首选这样的返回类型，而不是按值返回。这是为了避免在构建时对项目进行不必要的复制。另一方面，通过左值引用返回可能导致悬空引用，并使调用 build() 更加困难，因为返回的左值引用与预期的右值引用不匹配。

最终的构建器类型如下所示：

```
1 class ItemBuilder final : public GenericItemBuilder<ItemBuilder> {
2     using GenericItemBuilder<ItemBuilder>::GenericItemBuilder;
3 }
```

只是一个重用通用构造器中的构造函数的类。使用方式如下所示：

```
1 auto directly_loaded_item = ItemBuilder{"Pot"}  
2     .with_description("A decent one")  
3     .with_price(100)  
4     .build();
```

可以使用函数链调用最终的接口，方法名使整个调用可读。

如果不直接加载每一项，而是使用可以从远程端点加载部分数据的更专用构建器，会怎么样呢？可以这样定义：

```
1 class FetchingItemBuilder final  
2 : public GenericItemBuilder<FetchingItemBuilder> {  
3 public:  
4     explicit FetchingItemBuilder(std::string name)  
5         : GenericItemBuilder(std::move(name)) {}  
6  
7     FetchingItemBuilder&& using_data_from(std::string_view url) && {  
8         item_ = fetch_item(url);  
9         return std::move(*this);  
10    }  
11};
```

还使用 CRTP 来继承通用构建器，并强制给一个名称。这一次，使用函数扩展了基本构建器，以获取内容并将其放入正在构建的项中。多亏了 CRTP，当从基本构建器调用函数时，将返回派生函数，这使接口更容易使用。可以通过以下方式调用：

```
1 auto fetched_item =  
2     FetchingItemBuilder{"Linen blouse"}  
3     .using_data_from("https://example.com/items/linen_blouse")  
4     .marked_as_featured()  
5     .build();
```

一切都很好！

如果需要创建不可变对象，构建器也可以派上用场。因为构造器可以访问类的私有成员，所以可以修改它们（即使类没有为它们提供 setter）。

使用组合模式和原型模式

需要使用构建器的情况是在创建组合时。组合是一种设计模式，一组对象视为一个对象，所有对象共享相同的接口（或相同的基类型）。例如，图（可以由子图组成）或文档（可以嵌套其他文档）。当对这样的对象调用 `print()` 时，所有子对象都将调用它们的 `print()`，以便打印整个组合。构建器模式可以用于创建每个子对象并将它们组合在一起。

原型是另一种可用于对象构造的模式。如果类型创建成本很高，或者只想在其基础上构建一个基对象，可能需要使用此模式。它可以归结为提供一种克隆对象的方法，可以单独使用该对象，也可以对其进行修改，使其成为它应该的样子。在多态层结构的情况下，可以这样添加 `clone()`：

```
1 class Map {  
2     public:
```

```

3   virtual std::unique_ptr<Map> clone() const;
4   // ... other members ...
5 };
6
7 class MapWithPointsOfInterests {
8 public:
9   std::unique_ptr<Map> clone() override const;
10  // ... other members ...
11 private:
12   std::vector<PointOfInterest> pois_;
13 };

```

`MapWithPointsOfInterests` 对象也可以克隆，所以不需要手动重新添加。通过这种方式，可以在最终用户创建自己的 map 时向他们提供一些默认值。在某些情况下，使用简单的复制构造函数就足够了，而不是使用原型。

现在已经介绍了对象创建。在这个过程中接触了变体，所以为什么不重新访问它们（双关语），看看它们还能做些什么？

6.5. C++中跟踪状态和访问对象

状态是一种设计模式，用于在对象的内部状态发生变化时改变对象的行为。不同状态的行为应该相互独立，这样添加新状态就不会影响当前状态。在有状态对象中实现所有行为的简单方法无法扩展，也不允许扩展。使用状态模式，可以通过引入新的状态类，并定义它们之间的转换来添加新的行为。本节中，将展示一种实现状态的方法，以及使用 `std::variant` 和静态多态双重分发模式。换句话说，将以 C++ 的方式连接状态模式和访问者模式，从而构建一个有限的状态机。

首先，定义状态。示例中，为商店中的产品进行状态建模。如下所示：

```

1 namespace state {
2
3   struct Depleted {};
4
5   struct Available {
6     int count;
7   };
8
9   struct Discontinued {};
10 } // namespace state

```

状态可以拥有自己的属性，例如：剩余物品的数量。而且，与动态多态不同，不需要从公共基类继承。可以存储在一个变量中，如下所示：

```

1 using State = std::variant<state::Depleted, state::Available,
2 state::Discontinued>;

```

除了状态之外，还需要事件来进行状态转换：

```

1 namespace event {
2

```

```

3 struct DeliveryArrived {
4     int count;
5 };
6
7 struct Purchased {
8     int count;
9 };
10
11 struct Discontinued {};
12
13 } // namespace event

```

事件也可以具有属性，而不是继承自公共基类。现在，需要实现状态之间的转换。可以这样做：

```

1 State on_event(state::Available available, event::DeliveryArrived
2 delivered) {
3     available.count += delivered.count;
4     return available;
5 }
6
7 State on_event(state::Available available, event::Purchased purchased) {
8     available.count -= purchased.count;
9     if (available.count > 0)
10        return available;
11    return state::Depleted{};
12 }

```

如果进行了“购买”，状态可以改变，但也可以保持不变。可以使用模板一次处理多个状态：

```

1 template <typename S> State on_event(S, event::Discontinued) {
2     return state::Discontinued{};
3 }

```

如果商品停产了，处于什么状态并不重要。现在，来实现最后一个转换：

```

1 State on_event(state::Depleted depleted, event::DeliveryArrived delivered)
2 {
3     return state::Available{delivered.count};
4 }

```

需要解决的下一个难题是如何在对象中定义多个调用操作符，以便调用最佳匹配的重载。稍后使用它来调用刚刚定义的转换。助手类可以这样写：

```

1 template<class... Ts> struct overload : Ts... { using Ts::operator()...; };
2 template<class... Ts> overload(Ts...) -> overload<Ts...>;

```

创建重载结构，该结构将使用变量模板、折叠表达式和类模板参数推导指南提供在构造过程中，传递给所有调用操作符。要了解更深入的解释，以及实现访问的另一种方法，请参阅 Bartłomiej Filipek 在扩展阅读部分的博客文章。

现在可以开始实现状态机本身了：

```

1 class ItemStateMachine {
2 public:
3     template <typename Event> void process_event(Event &&event) {
4         state_ = std::visit(overload{
5             [&](const auto &state) requires std::is_same_v<
6                 decltype(on_event(state, std::forward<Event>(event))), State> {
7                 return on_event(state, std::forward<Event>(event));
8             },
9             [](const auto &unsupported_state) -> State {
10                 throw std::logic_error{"Unsupported state transition"};
11             }
12         },
13         state_);
14     }
15
16 private:
17     State state_;
18 };

```

`process_event` 函数将接受任何事件，将使用当前状态和传递的事件调用适当的 `on_event` 函数，并切换到新的状态。如果给定的状态和事件可以找到 `on_event` 重载，将调用第一个 lambda。否则，约束将不满足，第二个更通用的重载将调用。这意味着如果存在不受支持的状态转换，则会抛出异常。

现在，提供一种方法来报告当前状态：

```

1 std::string report_current_state() {
2     return std::visit(
3         overload{[] (const state::Available &state) -> std::string {
4             return std::to_string(state.count) +
5                 " items available";
6         },
7         [] (const state::Depleted) -> std::string {
8             return "Item is temporarily out of stock";
9         },
10        [] (const state::Discontinued) -> std::string {
11            return "Item has been discontinued";
12        },
13        state_);
14    }

```

这里，使用重载来传递三个 lambda，每个 lambda 都返回一个通过访问状态对象生成的字符串报告。

应用解决方案：

```

1 auto fsm = ItemStateMachine{};
2 std::cout << fsm.report_current_state() << '\n';
3 fsm.process_event(event::DeliveryArrived{3});
4 std::cout << fsm.report_current_state() << '\n';
5 fsm.process_event(event::Purchased{2});

```

```
6 std::cout << fsm.report_current_state() << '\n';
7 fsm.process_event(event::DeliveryArrived{2});
8 std::cout << fsm.report_current_state() << '\n';
9 fsm.process_event(event::Purchased{3});
10 std::cout << fsm.report_current_state() << '\n';
11 fsm.process_event(event::Discontinued{});
12 std::cout << fsm.report_current_state() << '\n';
13 // fsm.process_event(event::DeliveryArrived{1});
```

运行时，将产生如下输出：

```
Item is temporarily out of stock
3 items available
1 items available
3 items available
Item is temporarily out of stock
Item has been discontinued
```

除非用不支持的转换取消最后一行的注释，否则将在最后抛出异常。

尽管支持的状态和事件列表仅限于编译时提供的状态和事件列表，但与基于动态多态的解决方案相比，解决方案性能要好得多。更多关于状态、变量和各种访问方式的信息，请参见 Mateusz Pusz 在 2018 年 CppCon 上的讲话，该讲话在扩展阅读部分可以找到。

结束这一章之前，最后了解一下如何处理内存。

6.6. 高效地处理内存

即使机器的内存资源很丰富，了解如何使用它的也是个好主意。通常，内存吞吐量是现代系统的性能瓶颈，因此充分利用它很重要。执行太多的动态分配可能会降低程序的速度，并导致内存碎片。让我们来了解一些缓解这些问题的方法。

6.6.1 使用 SSO/SOO 减少动态分配

动态分配有时会带来其他麻烦，而不仅仅是在没有足够内存的情况下构造对象时抛出问题。它们通常会占用 CPU 周期，并可能导致内存碎片。幸运的是，有一种方法可以避免这种情况。若使用过 `std::string`(GCC 5.0 之后)，可以使用小字符串优化 (Small string optimization, SSO) 的策略。这是一个名为小对象优化 (Small Object optimization, SSO) 的更通用的优化示例，可以在 Abseil 的 `InlinedVector` 类型中看到。主要思想很简单：如果动态分配的对象足够小，就应该存储在拥有它的类中，而不是动态分配。在 `std::string` 中，通常有容量、长度和存储的实际字符串。如果字符串足够短 (对于 GCC 来说，在 64 位平台上是 15 字节)，将存储在其中的一些成员中。

就地存储对象，而不是进行分配，只存储指针还有一个好处：更少的指针跟踪。每次需要访问存储在指针后面的数据时，都会增加 CPU 缓存的压力，并面临从主存获取数据的风险。如果这是

一个常见的模式，会影响应用程序的整体性能，特别是指向的地址没有被 CPU 的预取器猜到。使用 SSO 和 SOO 等技术对于减少这些问题十分有用。

6.6.2 通过 COW 来节省内存

若在 GCC 5.0 之前使用了 GCC 的 `std::string`，可以使用 Copy-On-Write(COW) 优化。当使用相同的底层字符数组创建多个实例时，COW 字符串实现实际上共享相同的内存地址。将字符串写入时，将复制存储的数据——因此有了名称。

这种技术有助于节省内存并保持缓存热度，通常在单个线程上提供稳定的性能。但要注意不要在多线程上下文中使用它，锁是真正的性能杀手。与任何与性能相关的主题一样，最好只确定它是否匹配目前的工作。

现在来讨论 C++17 的一个特性，可以通过动态分配实现良好的性能。

6.6.3 使用多态分配器

这里讨论的特性是多态分配器。具体来说，是分配程序用来分配内存的 `std::pmr::polymorphic_allocator` 和多态的 `std::pmr::memory_resource` 类。

本质上，可以轻松地链接内存资源，以最大限度地利用内存。链的实现可以很简单，比如将资源保留一个大块并分发，然后回到另一个资源，如果耗尽了，则使用 `new` 和 `delete`。实现也可以更复杂：可以构建一个处理不同大小的池的长内存资源链，只在需要时提供线程安全，绕过堆并直接获取系统内存，返回最后释放的内存块以提供缓存热度，以及做其他有趣的事情。所有这些功能都是由标准的多态内存资源提供的，因为它们的设计很容易进行扩展。

存储区

存储区，可以在有限时间内存在的大块内存。可以用分配在存储区生命周期中使用的较小的对象。存储区中的对象也可以释放，也可以在清空内存的过程中删除。

与通常的分配和释放方式相比，存储区有几大优势——因为限制了需要获取上游资源的内存分配，可以提高性能。因为可能发生的碎片化都将发生在区域内部，还减少了内存的碎片化。当区域内存释放后，也就不存在碎片化了。一个好的方式是为每个线程创建独立的存储区，若只有一个线程使用存储区，那么不需要使用锁或其他线程安全的机制，从而可以减少线程争用，并在性能上带来重大的提升。

如果程序是单线程的，提高其性能的低成本解决方案可以这样：

```
1 auto single_threaded_pool = std::pmr::unsynchronized_pool_resource();
2 std::pmr::set_default_resource(&single_threaded_pool);
```

如果不显式设置任何资源，默认资源将是 `new_delete_resource`，会像常规的 `std::allocator` 一样每次调用 `new` 和 `delete`，并保证线程的安全性（和成本）。

如果使用前面的代码，那么使用 pmr 分配器完成的所有分配都不需要锁，但仍需要使用 pmr 类型。想要在标准容器中这样使用，就需要简单地传递 `std::pmr::polymorphic_allocator<T>` 作为 `allocator` 模板参数。许多标准容器都有启用 pmr 的类型别名。下面创建的两个变量是相同类型的，都使用默认的内存资源：

```
1 auto ints = std::vector<int>;
2 std::pmr::polymorphic_allocator<int>>(std::pmr::get_default_resource());
3 auto also_ints = std::pmr::vector<int>{};
```

但第一个函数会显式传递资源。现在了解一下 pmr 中可用的资源。

单调内存

第一个是 `std::pmr::monotonic_buffer_resource`, 一个只分配内存, 不做关于回收的事情, 只会在内存销毁或显式调用 `release()` 时释放内存。由于不保证线程安全, 使得这种类型具有极高的性能。如果应用程序偶尔需要在给定线程上执行大量分配资源的任务, 然后释放之后一次使用的所有对象, 使用单调的内存会有性能上的很大收益。这也是构建资源链的重要基础模块。

资源池

两种资源的常见组合是在单调内存上使用资源池。标准资源池可以创建不同大小的块的池。在 `std::pmr` 中有两种类型, `unsynchronized_pool_resource` 只在一个线程上分配和释放时使用, `synchronized_pool_resource` 适用于多线程使用。两者都能提供比全局分配器更好的性能, 特别是在使用单调内存区作为其上游资源时。如果想知道如何锁住它们, 可以参考以下代码:

```
1 auto buffer = std::array<std::byte, 1 * 1024 * 1024>{};
2 auto monotonic_resource =
3 std::pmr::monotonic_buffer_resource{buffer.data(), buffer.size()};
4 auto pool_options = std::pmr::pool_options{.max_blocks_per_chunk = 0,
5 .largest_required_pool_block = 512};
6 auto arena =
7 std::pmr::unsynchronized_pool_resource{pool_options,
8 &monotonic_resource};
```

创建了一个 1MB 的缓冲区供区域重用。这里将其传递给一个单调内存, 然后将其传递给一个不同步的资源池, 并创建一个简单而有效的分配器链, 在所有初始内存用完之前不会调用 `new`。

可以传递 `std::pmr::pool_options` 对象给这两种池类型, 以限制给定块的最大数量 (`max_blocks_per_chunk`) 或最大块的大小 (`max_required_pool_block`)。传递 0 将使用实现的默认值。在 GCC 的情况下, 每个块的实际块取决于块的大小。如果超过了最大的内存大小, 资源池将直接从其上游资源分配。如果初始内存耗尽, 也会转到上游资源。在这种情况下, 会分配以几何倍数增长的内存块。

编写自己的内存资源

如果标准内存资源不能满足需求, 可以简单地创建一个自定义内存资源, 并不是所有的标准库实现都会提供很好的优化, 即跟踪最后释放的大小给定的内存块, 并在给定大小的下一个分配中进行返回。最近使用的缓存可以增加数据缓存的热度, 这有助于应用的性能。可以将它看作一组块的后进先出队列。

有时, 可能还想调试分配和释放。下面的代码段中, 我写了一个简单的资源器, 可以完成这个任务:

```
1 class verbose_resource : public std::pmr::memory_resource {
2     std::pmr::memory_resource *upstream_resource_;
3 public:
4     explicit verbose_resource(std::pmr::memory_resource *upstream_resource)
5         : upstream_resource_(upstream_resource) {}
```

verbose_resource 继承 memory_resource，可以接受上游资源，用于实际的分配。其必须实现三个私有函数——一个用于分配，一个用于释放，一个用于比较资源本身的实例。先是第一个：

```
1 private:
2 void *do_allocate(size_t bytes, size_t alignment) override {
3     std::cout << "Allocating " << bytes << " bytes\n";
4     return upstream_resource_->allocate(bytes, alignment);
5 }
```

确实在标准输出上打印分配大小，然后使用上游资源来分配内存。下一个类似：

```
1 void do_deallocate(void *p, size_t bytes, size_t alignment) override {
2     std::cout << "Deallocating " << bytes << " bytes\n";
3     upstream_resource_->deallocate(p, bytes, alignment);
4 }
```

记录释放和使用上游执行任务的内存数量。下面是最后一个必需的函数：

```
1 [[nodiscard]] bool
2 do_is_equal(const memory_resource &other) const noexcept override {
3     return this == &other;
4 }
```

只要比较实例的地址就可以知道是否相等。[[nodiscard]] 属性可以确保调用者实际上使用了返回的值，这可以避免函数的滥用。

就是这样。对于像 pmr 分配器这样强大的特性，API 现在还不怎么复杂，不是吗？

除了跟踪分配之外，还可以使用 pmr 来避免在不应该分配时进行分配。

确保没有意外的分配

std::pmr::null_memory_resource() 将在试图使用它分配内存时抛出异常。通过将 pmr 设置为默认资源，从而避免使用 pmr 进行任何分配，如下所示：

```
1 std::pmr::set_default_resource(null_memory_resource());
```

还可以用来限制不应该发生的上游分配。检查以下代码：

```
1 auto buffer = std::array<std::byte, 640 * 1024>{}; // 640K ought to be
2 enough for anybody
3 auto resource = std::pmr::monotonic_buffer_resource{
4     buffer.data(), buffer.size(), std::pmr::null_memory_resource()};
```

若试图分配超过设置的缓冲区大小，将抛出 std::bad_alloc 异常。

接着讨论本章的最后一项内容。

闪烁清理内存

有时，不必释放内存（如单调内存）仍然不足以提高性能。一种叫做“闪烁清理”的技术可以解决这个问题。其意味着不会逐个释放，而且构造函数也不会调用。对象只是蒸发，节省了在内存区中为每个对象及其成员（及其成员…）调用析构函数的时间。

Note

这是一个高级话题。使用这个技巧时要小心，只有在可能有性能收益的时候使用。

这种技术可以节省宝贵的 CPU 周期，但并不总可以使用。如果对象处理的是内存以外的资源，则需要避免闪烁清理内存。否则，就会出现资源泄漏。如果依赖于对象的析构函数有副作用，情况也是如此。

现在让看看“闪烁”的动作：

```
1 auto verbose = verbose_resource(std::pmr::get_default_resource());
2 auto monotonic = std::pmr::monotonic_buffer_resource(&verbose);
3 std::pmr::set_default_resource(&monotonic);
4
5 auto alloc = std::pmr::polymorphic_allocator{};
6 auto *vector = alloc.new_object<std::pmr::vector<std::pmr::string>>();
7 vector->push_back("first one");
8 vector->emplace_back("long second one that must allocate");
```

这里，手工创建了一个多态分配器，将使用默认资源——单调内存资源，每次它到达上游时都会在日志中记录。为了创建对象，这里使用一个在 C++20 的 pmr 中新加入的函数，即 `new_object` 函数。这里创建了一个字符串向量，可以使用 `push_back` 传递第一个字符串，因为它足够小，可以放入 SSO 提供的小字符串缓冲区中。第二个字符串需要使用默认资源分配一个字符串，只有在使用 `push_back` 时才将它传递给 `vector`。绑定会导致字符串在 `vector` 函数中构造（而不是在调用之前），所以这里会使用 `vector` 的分配器。最后，不会在其他地方调用已分配对象的析构函数，而是在退出作用域时立即释放所有对象。这将带来非常可观的性能。

6.7 总结

本章中，介绍了 C++ 中使用的各种习惯性用法和模式，揭开了如何执行自动清理内存的神秘面纱，编写更安全的类型来正确地移动、复制和交换。并了解了如何在编译时间和编写定制点方面充分利用 ADL。讨论了如何在静态多态性和动态多态性之间进行选择。还了解到如何为类型引入策略，何时使用和不使用类型擦除。

更重要的是，讨论了如何使用工厂和流利的生成器创建对象。此外，使用内存区也不再是神秘魔法。使用变量等工具编写状态机也是如此。

还谈到了一些其他有趣的话题。呼！旅程的下一站将是关于构建软件，并将其打包。

6.8. 练习题

1. 三、五、零的规则是什么？
2. 什么时候使用 niebloids，而不是隐藏友元？
3. 如何改进阵列接口，使其更适合生产？

4. 什么是折叠表达式?
5. 什么时候不使用静态多态?
6. 可以在闪烁清理的示例中再省一个分配吗?

6.9. 扩展阅读

1. tag_invoke: A general pattern for supporting customisable functions, Lewis Baker, Eric Niebler, Kirk Shoop, ISO C++ proposal, <https://wg21.link/p1895>
2. tag_invoke::niebloids evolved, Gašper Ažman for the Core C++ Conference, YouTube video, <https://www.youtube.com/watch?v=oQ26YL0J6DU>
3. Inheritance Is The Base Class of Evil, Sean Parent for the GoingNative 2013 Conference, Channel9 video, <https://channel9.msdn.com/Events/GoingNative/2013/Inheritance-Is-The-Base-Class-of-Evil>
4. Modern C++ Design, Andrei Alexandrescu, Addison-Wesley, 2001
5. How Non-Member Functions Improve Encapsulation, Scott Meyers, Dr. Dobbs article, <http://www.drdobbs.com/cpp/how-non-member-functions-improve-encapsu/184401197>
6. Returning a Status or a Value, Status User Guide, Abseil documentation, <https://abseil.io/docs/cpp/guides/status#returning-a-status-or-a-value>
7. function_ref, GitHub repository, https://github.com/TartanLlama/function_ref
8. How To Use std::visit With Multiple Variants, Bartłomiej Filipek, post on Bartek's coding blog, <https://www.bfilipek.com/2018/09/visit-variants.html>
9. CppCon 2018: Mateusz Pusz, Effective replacement of dynamic polymorphism with std::variant, YouTube video, <https://www.youtube.com/watch?v=gKb0RJtnVu8>

第 7 章 构建和打包

作为架构师，需要了解构成构建过程的所有元素。本章将解释构建过程的所有元素。从编译器标志到自动化脚本和其他内容，将带领读者了解每个模块、服务和工件，并将其进行了版本控制，并存储在一个中心位置，以备部署。本章将主要关注于 CMake。

本章将讨论以下内容：

- 考虑使用哪些编译器标志
- 如何创建基于现代 CMake 的构建系统
- 如何构建可重用组件
- 如何使用 CMake 干净的外部代码
- 如何使用 CPack 创建 DEB 和 RPM 包，以及 NSIS 安装程序
- 如何使用 Conan 包管理器来安装依赖和创建包

了解本章之后，将知道如何编写最先进的代码来构建和打包项目。

7.1. 相关准备

要运行本章的例子，需要安装 GCC 和 Clang 的最新版本，CMake 3.15 或更高版本，Conan 和 Boost 1.69。

本章的代码可以在以下 GitHub 页面找到：<https://github.com/PacktPublishing/Software-Architecture-with-Cpp/tree/master/Chapter07>。

7.2. 充分利用编译器

编译器是每个开发者工作中最重要的工具之一。本节中，将介绍一些有用的技巧。这只是冰山一角，因为关于这些工具及其各种可用标志、优化、功能和其他细节汇总起来就是可以成书的量级，GCC 还有一个专门的 Wiki 页面，里面有关于编译器的书籍列表！可以在本章末尾的扩展阅读中找到它。

7.2.1 多编译器

构建过程中，需要考虑使用多个编译器（而非一个），原因是它们可以检测代码中的不同问题。例如，MSVC 默认启用签名检查。使用多个编译器可以提前发现可能遇到的可移植性问题，特别是决定在不同的操作系统上编译代码时，例如：从 Linux 迁移到 Windows 或其他操作系统。要无痛完成这些工作，需要编写可移植的、符合 ISO C++ 的代码。Clang 是一个比较好的选择，其力求比 GCC 更符合 C++ 标准。如果正在使用 MSVC，可以试着添加 /permissive- 选项（在 Visual Studio 17 加入，对于使用 15.5+ 版本创建的项目默认启用）。对于 GCC，在为代码选择 C++ 标准时尽量不要使用 GNU 扩展（例如，使用 `-std=c++17` 而不是 `-std=gnu++17`）。如果目标是性能，那么使用各种编译器来构建也可以为特定的用例，选择运行最快的二进制文件。

TIP

无论为发布版本选择哪种编译器，都可以考虑使用 Clang 进行开发。它可以在 macOS、Linux 和 Windows 上运行，支持与 GCC 相同的标志集，旨在提供最快的构建时间和简明的编译错误。

如果正在使用 CMake，有两种常见的方法来添加另一个编译器。一种是在调用 CMake 时传递相应的编译器，像这样：

```
mkdir build-release-gcc  
cd build-release-gcc  
cmake .. -DCMAKE_BUILD_TYPE=Release -DCMAKE_C_COMPILER=/usr/bin/gcc \  
-DCMAKE_CXX_COMPILER=/usr/bin/g++
```

也可以在调用 CMake 之前设置 CC 和 CXX，但这些环境变量在不同的平台（如 macOS）上会有所不同。

另一种方法是使用工具链文件。如果只使用不同的编译器，这可能没必要，但若想要交叉编译时，这是首选的解决方案。要使用一个工具链文件，应该作为一个参数传递给 CMake:`-DCMAKE_TOOLCHAIN_FILE=toolchain.cmake`。

7.2.2 减少构建时间

程序员每年都要花费无数个小时等待他们的构建完成。减少构建时间是提高整个团队生产力的一种简单方法，所以这里讨论一些实现它的方法。

快速编译器

提高构建速度的最简单方法之一就是升级编译器。例如，通过将 Clang 升级到 7.0.0，可以使用预编译头 (PCH) 文件节省多达 30% 的构建时间。从 Clang 9 开始，就有了`-ftime-trace` 选项，该选项可以提供有关它处理的所有文件的编译时间的信息。其他编译器也有类似的开关：查看 GCC 的`-ftime` 报告或 MSVC 的/Bt 和/d2cgsummary。通常可以通过切换编译器来获得更快的编译速度，这在开发机器上特别有用，例如：Clang 编译代码的速度通常比 GCC 快。

当有了一个快速的编译器后，再来看看需要编译什么。

反思模板

编译过程的不同部分需要不的时间来完成，这对于编译时构造尤其重要。Odin Holmes 的实习生之一 Chiel Douwes 在对各种模板操作的编译时成本进行基准测试的基础上创建了 Chiel 规则。其他基于类型的模板元编程技巧，可以在 Odin Holmes 的《基于类型的模板元编程并未消亡》讲座中看到。从最快到最慢，排列如下：

- 查找已实例化的类型（例如，模板实例化）
- 将参数传递给别名调用
- 向类型添加参数

- 调用一个别名
- 实例化一个类型
- 实例化一个函数模板
- 使用 SFINAE(替换失败不是错误)

为了演示该规则，看一下以下的代码：

```

1 template<bool>
2 struct conditional {
3     template<typename T, typename F>
4     using type = F;
5 };
6
7 template<>
8 struct conditional<true> {
9     template<typename T, typename F>
10    using type = T;
11 };
12
13 template<bool B, typename T, typename F>
14 using conditional_t = conditional<B>::template type<T, F>;

```

代码定义条件模板别名，该别名存储一个类型，如果条件 B 为真，则解析为 T，否则解析为 F。编写这样一个实用程序的传统方法如下：

```

1 template<bool B, class T, class F>
2 struct conditional {
3     using type = T;
4 };
5
6 template<class T, class F>
7 struct conditional<false, T, F> {
8     using type = F;
9 };
10
11 template<bool B, class T, class F>
12 using conditional_t = conditional<B,T,F>::type;

```

第二种方法的编译速度比第一种方法慢，因为它依赖于创建模板实例，而不是类型别名。

现在，来看看可以使用哪些工具及其特性，可以用来减少编译的时间。

利用工具

使用单个编译单元构建，或统一构建，可以使构建速度更快。它不会加快每个项目的速度，但若头文件中有大量的代码，可以试一试。统一构建是通过在一个转译单元中包含所有的.cpp 文件来构建的，另一个类似的想法是使用预编译头文件。诸如 CMake 的 Cotire 插件可以使用这两种技术。CMake 3.16 还增加了对统一构建的原生支持，可以为目标启用，`set_target_properties(<target> PROPERTIES UNITY_BUILD ON)`，或者通过设置 `CMAKE_UNITY_BUILD` 为 `true` 来启用全局统一构建。如果只是想要 PCH(预编译头文件)，只需要

去看看 CMake 3.16 的 `target_precompile_headers` 就好。

TIP

若在 C++ 文件中包含了太多的内容，可以考虑使用一个名为 `include-what-you-use` 的工具进行整理。可以将声明类型和函数放到头文件中，并且可以减少编译时间。

如果项目需要很长时间才能链接，也有一些方法可以解决这个问题。使用不同的链接器，如 LLVM 的 LLD 或 GNU 的 Gold，可以有很大的帮助，因为可以使用多线程链接。如果无法使用不同的链接器，可以尝试使用 `-fvisibility-hidden` 或 `-fvisibility-inlinesehidden` 这样的标志，并在源代码中使用适当的注释，只标记希望在动态库中可见的函数。这样，链接器要做的工作就更少了。如果正在对链接时间进行优化，可以尝试只对性能关键的构建进行优化：计划分析的构建和那些用于生产的构建。

如果正在使用 CMake，并且没有绑定到特定的生成器（例如，CLion 需要使用 Code::Blocks 生成器），可以用更快的生成器替换默认的 Make 生成器。Ninja 是一个很好的开始，它专为减少构建时间而创建的，只需在调用 CMake 时传递 `-G Ninja` 就可以使用。

还有两个很棒的工具，其中一个就是 Ccache。它是一个执行 C 和 C++ 编译输出缓存的工具。如果试图构建相同的内容两次，将从缓存中获得结果，而不是运行编译。它保留统计数据，例如：缓存命中和未命中，可以记住编译特定文件时应该发出的警告，并且有许多配置选项可以存储在 `/.ccache/ccache.conf` 文件中。要获取它的统计信息，只需运行 `ccache --show-stats` 即可。

第二个工具是 IceCC（或 Icecream）。它是 distcc 的一个分支，本质上是一个跨主机分发构建的工具。有了 IceCC，使用自定义工具链就更容易了。它在每个主机上运行 iceccd 守护进程，并运行一个 icecc-scheduler 服务来管理整个集群。与 distcc 不同的是，调度程序确保只使用每台机器上的空闲周期，因此不会导致其他人的工作机超载。

要使用 IceCC 和 Ccache 为您的 CMake 构建，只需在 CMake 调用中添加：

```
-DCMAKE_C_COMPILER_LAUNCHER="ccache;icecc"  
-DCMAKE_CXX_COMPILER_LAUNCHER="ccache;icecc"
```

如果在 Windows 上编译，想使用其他类似的工具，可以选择使用 clcache 和 Incredibuild 或其他替代工具。

7.2.3 寻找代码的潜在问题

如果代码有 bug，即使是最快的构建也没有多大价值。有许多标志可以用来警告代码中可能存在的问题。本节将展示应该考虑启用哪些选项。

首先，让从一个稍微不同的问题开始：如何不收到来自其他库的代码问题的警告。对于无法解决的问题上的警告没有任何价值。幸运的是，有一些编译器开关可以禁用此类警告。例如，在 GCC 中有两种类型的包含文件：常规文件（使用 `-I` 传递）和系统文件（使用 `-system` 传递）。如果使用后者指定目录，则不会从相应的头文件中获得警告。MSVC 有一个 `-system` 的等价选项 `:/external:I`。此外，它还有其他标志来处理外部包含，比如 `/external:anglebrackets`，它告诉编译器将使用尖括号包含的所有文件视为外部文件，从而禁用对它们的警告。可以为外部文件

指定警告级别，也可以让警告来自于使用`/external`进行模板实例化的代码：如果正在寻找一种可移植的方式来将包含路径标记为系统/外部路径，并且正在使用 CMake，可以将`SYSTEM`关键字添加到`target_include_directories`指令的参数中。

说到可移植性，如果想要符合 C++ 标准，考虑在 GCC 或 Clang 的编译选项中加入`-pedantic`，或者在 MSVC 中加入`/permissive-`选项。这样，就会了解每个可能使用非标准扩展的模块。如果正在使用 CMake，为每个目标添加设置相应的属性，`set_target_properties(<target> PROPERTIES CXX_EXTENSIONS OFF)` 以禁用特定于编译器的扩展。

如果正在使用 MSVC，可以尝试使用`/W4` 来编译代码，因为它启用了大多数重要的警告。对于 GCC 和 Clang，尝试使用`-Wall -Wextra -Wconversion -Wsign-conversion`。第一个，只启用一些常见的警告。然而，第二个又增加了一堆警告。第三个是基于 Scott Meyers 的著作《Effective C++》中的提示（这是一组很好的警告，但是要确保是否符合需求）。后两个是关于类型转换和签名转换的。所有这些标志创建了一个完整的安全网，当然可以启用更多的标志。Clang 有一个`-Weverything` 标志，尝试使用它运行构建，可以发现新的警告。虽然启用一些警告标志可能没这么麻烦，但是开发者可能会对使用此标志获得的消息数量感到惊讶。MSVC 的一个替代方案命名为`/Wall`。看看下面的表格，了解一下其他没有启用的有趣选项：

GCC/Clang:

标志	含义
<code>-Wduplicated-cond</code>	当在 if 和 else-if 块中使用相同的条件时发出警告
<code>-Wduplicated-branches</code>	如果两个分支包含相同的源代码，则发出警告。
<code>-Wlogical-op</code>	当逻辑操作中的操作数相同且应该使用位操作符时发出警告。
<code>-Wnon-virtual-dtor</code>	当类有虚函数，但没有虚析构函数时发出警告。
<code>-Wnull-dereference</code>	警告空解引用。在未优化的构建中，此检查可能是不开启的。
<code>-Wuseless-cast</code>	转换为相同类型时发出警告。
<code>-Wshadow</code>	所有关于覆盖了之前声明的警告。

MSVC:

Flag	Meaning
<code>/w44640</code>	对非线程安全的静态成员初始化发出警告。

最后一个问题：何时使用`-Werror`（或在 MSVC 是`/WX`）或不使用`-Werror`？这实际上取决于个人偏好，因为发出错误而不是警告有其优缺点。从积极的方面来说，不会让警告溜走。CI 构建将失败，代码将无法编译。在运行多线程构建时，不会在快速传递的编译消息中丢失警告。然而，也有一些缺点。如果编译器启用了新的警告或只是检测到更多问题，那么就无法升级编译器。依赖关系也是如此，依赖关系会使一些函数不受欢迎。如果代码中的内容在项目的其他部分使用，就不能弃用它。幸运的是，可以使用混合解决方案：使用`-Werror` 进行编译，但在需要时禁用它。这就需要代码的编写很有纪律性，如果有新的警告出现，可能很难消除。

7.2.4 以编译器为中心的工具

与几年前相比，编译器可以做更多的事情，这是由于 LLVM 和 Clang 的引入。通过提供 API 和重用的模块化架构，使消毒工具、自动重构或代码完成引擎等工具蓬勃发展，并可以利用这个编译器基础设施提供的优势。使用 clang-format 来确保代码库中的所有代码都符合给定的标准。考虑添加预提交钩子，使用预提交工具在提交之前重新格式化新代码。也可以添加 Python 和 CMake 格式器。使用 clang-tidy 静态分析代码——这个工具能够理解代码，而不仅仅是对代码进行推演。这个工具可以执行不同的检查，所以一定要根据特定需求定制列表和选项。还可以在启用杀毒软件的情况下，每晚或每周运行软件测试。通过这种方式，可以检测线程问题、未定义的行为、内存访问、管理问题等等。如果发布版本禁用了断言，那么使用调试版本运行测试也可以。

如果认为还可以做得更多，可以考虑使用 Clang 的基础架构编写自己的代码重构。如果想了解如何创建自己的基于 llvm 的工具，那么可以参考 clang-rename 工具。对 clang-tidy 进行额外的检查和修复也不是那么难，这个工具可以节省几个小时的体力劳动。

可以将许多工具集成到构建过程中。现在就讨论这个过程的核心：构建系统。

7.3. 抽象构建过程

本节中，将深入研究 CMake 脚本，它是全世界范围内用于 C++ 项目标准构建系统生成器。

7.3.1 CMake

CMake 是一个构建系统生成器，而不是构建系统本身是什么意思呢？简单地说，CMake 可以用来生成各种类型的构建系统。可以使用它来生成 Visual Studio 项目、Makefile 项目、基于 Ninja 的项目、Sublime、Eclipse 和其他一些项目。

CMake 附带了一组其他工具，如用于执行测试的 CTest 和用于打包和创建安装程序的 CPack。CMake 本身也允许导出和安装目标。

CMake 的生成器可以是单配置生成器，比如 Make 或 NMAKE，也可以是多配置生成器，比如 Visual Studio。对于单配置生成器，第一次在一个文件夹中运行生成时，应该传递 CMAKE_BUILD_TYPE 标志。例如，要配置一个调试版本，可以运行 `cmake <project_directory> -DCMAKE_BUILD_TYPE=Debug`。其他预定义的配置包括 Release、RelWithDebInfo(带有调试符号的版本) 和 MinSizeRel(为最小二进制优化版本)。为了保持源目录的整洁，总是创建一个单独的构建文件夹，并从那里运行 CMake 生成。

尽管可以添加自己的构建类型，但尽量不要这样做，因为这会使某些 IDE 使用起来会变得更加困难，而且无法扩展。更好的选择是使用选项。

CMake 文件可以用两种风格编写：过时的基于变量的风格，以及基于目标的现代 CMake 风格。这里我们只关注后者。尽量避免通过全局变量进行设置，在重用目标时，这会导致问题。

创建 CMake 工程

每个 CMake 项目在顶部 CMakeLists.txt 文件中应该有以下几行：

```
cmake_minimum_required(VERSION 3.15...3.19)
```

```
project(  
    Customer  
    VERSION 0.0.1  
    LANGUAGES CXX)
```

设置支持的最小和最大版本很重要，因为它会通过设置策略影响 CMake 的行为。如果需要，也可以手动设置。

项目的定义指定了它的名称、版本 (将用于填充一些变量) 和 CMake 将用于构建项目 (填充更多变量并找到所需的工具) 的编程语言。

通常，一个 C++ 项目有以下目录：

- cmake: CMake 脚本
- include: 公共头文件，通常带有以项目名称命名的子文件夹
- src: 源文件和私有头文件
- test: 测试

可以使用 CMake 目录来存储定制的 CMake 模块。为了方便地从这个目录访问脚本，可以添加到 CMake 的 `include()` 搜索路径中，如下所示：

```
list(APPEND CMAKE_MODULE_PATH "${CMAKE_CURRENT_LIST_DIR}/cmake"
```

当包含 CMake 模块时，可以省略`.cmake` 后缀。这样，`include(CommonCompileFlags.cmake)` 就等价于 `include(CommonCompileFlags)`。

CMake 内置目录变量

在 CMake 中目录有一个常见的陷阱，并不是每个人都知道。当编写 CMake 脚本时，试着区分以下内置变量：

- `PROJECT_SOURCE_DIR`: 项目命令 CMake 脚本调用的目录。
- `PROJECT_BINARY_DIR`: 与前面的示例类似，但用于构建目录。
- `CMAKE_SOURCE_DIR`: 顶层源目录 (这可能在另一个项目中，只是将当前项目作为依赖项/子目录添加)。
- `CMAKE_BINARY_DIR`: 类似于 `CMAKE_SOURCE_DIR`，但用于构建目录。
- `CMAKE_CURRENT_SOURCE_DIR`: 对应于当前处理的 `CMakeLists.txt` 文件的源目录。
- `CMAKE_CURRENT_BINARY_DIR`: 与 `CMAKE_CURRENT_SOURCE_DIR` 匹配的二进制 (构建) 目录。
- `CMAKE_CURRENT_LIST_DIR`: `CMAKE_CURRENT_LIST_FILE` 目录。如果当前的 CMake 脚本包含在另一个目录中 (对于包含的 CMake 模块来说很常见)，可以不同于当前的源目录。

在顶层的 `CMakeLists.txt` 文件中，可能需要调用 `add_subdirectory(src)` 以便 CMake 处理该目录。

指定 CMake 目标

`src` 目录中，应该有另一个 `CMakeLists.txt` 文件，这次可能定义了一个或两个目标。现在为前面提到的多米尼加博览会系统，添加一个客户微服务的可执行文件：

```
add_executable(customer main.cpp)
```

源文件可以在前面的代码行中指定，或者使用 `target_sources` 添加。

常见的 CMake 反模式是使用 `glob` 指定源文件，这种方式的缺点是 CMake 在重新运行生成步骤之前，不会知道文件是否已经添加。这样做的后果是，如果从存储库中提取更改并简单地构建，可能会错过编译和运行新的单元测试或其他代码。即使使用带有 `CONFIGURE_DEPENDS` 的 `glob`，构建时间也会变长，因为 `glob` 必须作为每个构建的一部分进行检查。此外，该标志可能并不适用于所有生成器。甚至 CMake 的作者也不喜欢使用它，而是喜欢显式地声明源文件。

好的，定义了代码来源。现在指定目标需要编译器支持 C++17：

```
target_compile_features(customer PRIVATE cxx_std_17)
```

`PRIVATE` 关键字说明这是一个内部需求，也就是只对这个特定目标可见，但对依赖它的目标不可见。如果正在编写一个为用户提供 C++17 API 的库，那么可以使用 `INTERFACE` 关键字。要指定接口和内部需求，可以使用 `PUBLIC` 关键字。当用户链接到目标时，CMake 也会自动要求支持 C++17。如果正在编写一个未构建的目标（即仅包含头文件的库或导入的目标），使用 `INTERFACE` 关键字通常就足够了。

指定目标使用 C++17 特性，并不会强制执行 C++ 标准，也不允许对目标进行编译器扩展。为此，应该这样使用：

```
set_target_properties(customer PROPERTIES
  CXX_STANDARD 17
  CXX_STANDARD_REQUIRED YES
  CXX_EXTENSIONS NO
)
```

若想要有一组编译器标志来传递给每个目标，可以将它们存储在一个变量中。若想创建一个目标，将这些标志设置为 `INTERFACE`，并且没有任何源文件，并在 `target_link_libraries` 中使用这个目标：

```
target_compile_options(customer PRIVATE ${BASE_COMPILE_FLAGS})
```

除了添加链接器标志外，该命令还自动传播包括目录、选项、宏和其他属性。说到链接，先创建一个要链接的库：

```
add_library(libcustomer lib.cpp)
add_library(domifair::libcustomer ALIAS libcustomer)
set_target_properties(libcustomer PROPERTIES OUTPUT_NAME customer)
# ...
target_link_libraries(customer PRIVATE libcustomer)
```

`add_library` 可用于创建静态、动态、对象和接口（仅考虑头文件）库，以及定义导入库。

它的 `ALIAS` 版本创建了一个命名空间目标，这有助于调试许多 CMake 问题，是现代 CMake 推荐的实践方式。

因已经给目标一个 lib 前缀，所以输出名称为 ibcustomer.a，而不是 libibcustomer.a。

最后，将可执行文件与添加的库链接起来。尽量为 `target_link_libraries` 命令指定 PUBLIC、PRIVATE 或 INTERFACE 关键字，因为这对于 CMake 管理目标依赖项的可传递性至关重要。

指定输出目录

当使用 `cmake --build .` 等命令构建代码，可能想知道在哪里可以找到构建好的组件。默认情况下，CMake 将在与定义它们的源目录相匹配的目录中创建它们。例如，如果有一个 `src/CMakeLists.txt` 文件，其中有一个 `add_executable` 指令，那么这个二进制文件将默认放在构建目录的 `src` 子目录中。也可以使用以下代码进行覆盖：

```
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${PROJECT_BINARY_DIR}/bin)
set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY ${PROJECT_BINARY_DIR}/lib)
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${PROJECT_BINARY_DIR}/lib)
```

通过这种方式，二进制文件和 DLL 文件将放在项目的构建目录的 `bin` 子目录中，而静态和动态 Linux 库将放在 `lib` 子目录中。

7.3.2 生成器表达式

在设置编译标志方面，同时支持单配置和多配置生成器可能会很棘手，因为 CMake 在配置时执行 `if` 语句和许多其他构造，而不是在构建/安装时执行。

这意味着以下是 CMake 的反模式：

```
if(CMAKE_BUILD_TYPE STREQUAL Release)
  target_compile_definitions(libcustomer PRIVATE RUN_FAST)
endif()
```

相反，生成器表达式是实现相同目标的正确方法，因为它们将在稍后处理。若想为 `Release` 配置添加一个预处理器定义，可以这样写：

```
target_compile_definitions(libcustomer PRIVATE
  "$<${CONFIG:Release}>:RUN_FAST")
```

这将解析为在特定的配置下，对 `RUN_FAST` 进行构建。对于其他配置，将解析为空值。它适用于单配置和多配置生成器。不过，这并不是生成器表达式的唯一用例。

当项目在构建期间使用目标，而其他项目在安装目标时，使用目标的方式可能会有所不同。比如 `include` 目录，在 CMake 中处理这个的常见方法如下：

```
target_include_directories(
  libcustomer PUBLIC $<INSTALL_INTERFACE:include>
  $<BUILD_INTERFACE:${PROJECT_SOURCE_DIR}/include>)
```

本例中，有两个生成器表达式。第一个在安装时可以在 `include` 目录中找到包含文件，相对于安装前缀（安装的根目录）。如果不安装，这个表达式将变成一个空表达式。这就是为什么有另一个表达“构建”的短语，这个将解析为找到最后一个使用的 `project()` 目录的 `include` 子目录。

Note

不要使用带有模块外部路径的 `target_include_directories`。如果这样做，就是在窃取别人的头文件，而不是显式声明库/目标依赖。这是 CMake 的一个反模式。

CMake 定义了许多生成器表达式，可以使用它们来查询编译器和平台，以及目标（例如全名、目标文件列表、任何属性值等）。除此之外，还有运行布尔运算、`if` 语句、字符串比较等的表达式。

现在，对于更复杂的例子，若想要在目标中使用一组编译标志，并且依赖于使用的编译器，可以这样定义：

```
list(
    APPEND
    BASE_COMPILE_FLAGS
    "$$<$OR:$<CXX_COMPILER_ID:Clang>,
     $<CXX_COMPILER_ID:AppleClang>,$<CXX_COMPILER_ID:GNU>>:-Wall;-Wextra;-pedantic;-Werror>"
    "$<$<CXX_COMPILER_ID:MSVC>:/W4;/WX>")
```

如果编译器是 Clang、AppleClang 或 GCC，这将附加一组标志，如果使用 MSVC 则附加另一组标志。注意，这里用分号分隔标志，因为这是 CMake 分隔列表元素的方式。

接下来，来看看如何添加项目，并使用的外部代码。

7.4. 使用外部模块

有几种方法可以获取所依赖的外部项目。例如，可以将它们添加为一个 Conan 依赖，使用 CMake 的 `find_package` 来查找操作系统提供的或以其他方式安装的版本，或者自己获取和编译依赖。

这一节的关键是：如果可以，使用 Conan。这样，就可以使用与项目及其依赖项需求相匹配的依赖项版本。

如果目标是支持多个平台，甚至同一发行版本的多个版本，使用 Conan 或自行编译都是可行的方法。这样，无论您在哪个操作系统上编译，都可以使用相同的依赖版本。

现在来讨论一下获取 CMake 本身提供的依赖项的几种方法，然后跳转到使用 Conan 的多平台包管理器。

7.4.1 获取依赖项

从源代码中准备依赖项的一种方法是使用 CMake 的内置 `FetchContent` 模块。它可以下载依赖项，然后作为常规目标对其进行构建。

新特性出现在 CMake 3.11 中。它是 `ExternalProject` 模块的替代品，`ExternalProject` 模块有很多缺陷。其中之一是，构建期间克隆外部存储库，因此 CMake 无法推断外部项目定义的目标，及其依赖关系。这使得许多项目需要手动定义此类外部目标的 `include` 目录和 `lib` 路径，从而完全忽略所需接口编译标志和依赖关系。而 `FetchContent` 就没有这样的问题，所以推荐使用。

Note

展示如何使用之前，必须知道 `FetchContent` 和 `ExternalProject`(以及使用 Git 子模块和类似的方法) 都有一个重要的缺陷。如果有许多依赖项使用相同的第三方库本身，那么可能最终会拥有同一个项目的多个版本，例如 Boost 的几个版本。使用类似 Conan 的包管理器则可以避免此类问题的发生。

作为示例，演示如何使用 `FetchContent` 特性将 GTest 集成到项目中。首先，创建一个 `FetchGTest.cmake` 文件，并把它放在源码树中的 `cmake` 目录中。`FetchGTest` 脚本将有如下定义：

```
include(FetchContent)

FetchContent_Declare(
    googletest
    GIT_REPOSITORY https://github.com/google/googletest.git
    GIT_TAG        dcc92d0ab6c4ce022162a23566d44f673251eee4)

FetchContent_GetProperties(googletest)
if(NOT googletest_POPULATED)
    FetchContent_Populate(googletest)
    add_subdirectory(${googletest_SOURCE_DIR} ${googletest_BINARY_DIR}
        EXCLUDE_FROM_ALL)
endif()

message(STATUS "GTest binaries are present at ${googletest_BINARY_DIR}")
```

首先，包含内置的 `FetchContent` 模块。加载模块后，使用 `FetchContent_Declare` 声明该依赖，命名依赖项，并指定 CMake 克隆的存储库，以及相应的提交版本。

现在，读取外部库的属性并填充（如果还没有完成的话）。当有了源码，就可以使用 `add_subdirectory` 进行处理。在运行 `make all` 等命令时，如果其他目标不需要这些目标，那么 `EXCLUDE_FROM_ALL` 选项会告诉 CMake 不要构建这些目标。在成功处理目录之后，脚本将打印一条消息，指出构建后 GTests 库将在其中放置的目录。

若不喜欢将依赖关系与项目一起构建，也许下一种集成依赖关系的方法将更适合。

7.4.2 `find` 脚本

假设依赖项在本地上的某个地方可用，只需使用 `find_package` 尝试搜索它即可。如果依赖项提供了配置或目标文件（稍后会详细介绍），那么只需要编写这个简单的命令就可以了。当然，前提是依赖项在本地可用。如果不可用，使用者有责任在项目运行 CMake 之前安装它们。

要创建上述文件，依赖项将需要使用 CMake，但并非总是如此。如何处理那些不使用 CMake 的库？如果该库很受欢迎，那么很可能已经有人创建了 `find` 脚本供他人使用。1.70 以上版本的 Boost 库就是这种方法的一个常见示例。CMake 附带了一个 `FindBoost` 模块，可以通过 `find_package(Boost)` 来执行。

要使用前面的模块找到 Boost，首先需要在系统上安装它。之后，在 CMake 列表中，应该设置合理的选项。例如使用动态和多线程 Boost 库，而不是静态链接到 C++ 运行时，参考以下代码：

```
set(Boost_USE_STATIC_LIBS OFF)
set(Boost_USE_MULTITHREADED ON)
set(Boost_USE_STATIC_RUNTIME OFF)
```

然后，搜索库，如下所示：

```
find_package(Boost 1.69 EXACT REQUIRED COMPONENTS Beast)
```

这里只使用 Beast，它是 Boost 的一部分，是一个很棒的网络库。当找到该库，可以链接到目标，代码如下所示：

```
target_link_libraries(MyTarget PUBLIC Boost::Beast)
```

现在已经知道了如何正确地使用 find 脚本，接下来了解如何编写 find 脚本。

7.4.3 编写 find 脚本

如果依赖项既没有提供配置文件和目标文件，也没有人为它编写 find 模块，那可以自己编写这样的模块。

这应该不会是经常要做的事情，所以本小节如果不需要可以略过。要获得更多的描述，也应该阅读 CMake 官方文档中的指南（链接在扩展阅读部分）或只看一些安装了 CMake 的模块（通常在 Unix 系统上的目录如`/usr/share/cmake-3.17/Modules`）。简单起见，假设只需要找到一个依赖项的配置，但是可以分别找到 Release 和 Debug 二进制文件。这将使设置不同的目标，关联不同的变量。

脚本名决定了你将传递给 `find_package`，如果希望以 `find_package(Foo)` 结束，脚本应该命名为 `FindFoo.cmake`。

一个好的方式是用 reStructuredText 部分开始脚本，描述脚本实际将做什么，将设置哪些变量等。这种描述的例子如下所示：

```
#.rst:
# FindMyDep
# -----
#
# Find my favourite external dependency (MyDep).
#
# Imported targets
# ~~~~~
#
# This module defines the following :prop_tgt:`^IMPORTED` target:
#
# ``MyDep::MyDep``
# The MyDep library, if found.
#
```

通常，还需要描述脚本将设置的变量：

```
# Result variables
# ~~~~~
#
# This module will set the following variables in your project:
#
# ``MyDep_FOUND``
# whether MyDep was found or not
# ``MyDep_VERSION_STRING``
# the found version of MyDep
```

如果 MyDep 本身有依赖项，是时候找到它们了：

```
find_package(Boost REQUIRED)
```

现在可以开始搜索库了。常见的方法是使用 `pkg-config`:

```
find_package(PkgConfig)
pkg_check_modules(PC_MyDep QUIET MyDep)
```

若 `pkg-config` 有关于依赖的信息，就会设置一些变量，可以用这种方式来找到依赖。一个好主意可能是让用户设置一个变量来指向库的位置。按照 CMake 约定，应该命名为 `MyDep_ROOT_DIR`。

为 CMake 提供这个变量，用户可以使用 `-DMyDep_ROOT_DIR=some/path` 调用 CMake，在他们的构建目录中修改 `CMakeCache.txt` 中的变量，或使用 `ccmake` 或 `cmake-gui`。

现在，可以使用前面提到的路径来搜索依赖的头文件和库文件：

```

find_path(MyDep_INCLUDE_DIR
    NAMES MyDep.h
    PATHS "${MyDep_ROOT_DIR}/include" "${PC_MyDep_INCLUDE_DIRS}"
    PATH_SUFFIXES MyDep
)

find_library(MyDep_LIBRARY
    NAMES mydep
    PATHS "${MyDep_ROOT_DIR}/lib" "${PC_MyDep_LIBRARY_DIRS}"
)

```

然后，还需要设置找到的版本，就像在脚本头部中承诺的那样。要使用从 pkg-config 中找到的，可以这样写：

```
set(MyDep_VERSION ${PC_MyDep_VERSION})
```

或者，可以从头文件的内容、库路径的组件或使用其他方法提取版本。当这些做完后，可以使用 CMake 的内置脚本来决定是否成功找到库，同时处理 `find_package` 所有可能的参数：

```

include(FindPackageHandleStandardArgs)
find_package_handle_standard_args(MyDep
    FOUND_VAR MyDep_FOUND
    REQUIRED_VARS
    MyDep_LIBRARY
    MyDep_INCLUDE_DIR
    VERSION_VAR MyDep_VERSION
)

```

当决定提供一个目标而不仅仅是一堆变量时，是时候定义了：

```

if(MyDep_FOUND AND NOT TARGET MyDep::MyDep)
    add_library(MyDep::MyDep UNKNOWN IMPORTED)
    set_target_properties(MyDep::MyDep PROPERTIES
        IMPORTED_LOCATION "${MyDep_LIBRARY}"
        INTERFACE_COMPILE_OPTIONS "${PC_MyDep_CFLAGS_OTHER}"
        INTERFACE_INCLUDE_DIRECTORIES "${MyDep_INCLUDE_DIR}"
        INTERFACE_LINK_LIBRARIES Boost::boost
    )
endif()

```

最后，将内部使用的变量进行隐藏，不将其暴露给不想处理它们的用户：

```

mark_as_advanced(
    MyDep_INCLUDE_DIR
)
```

```
MyDep_LIBRARY  
)
```

现在，有了一个完整的 find 模块，可以按以下方式使用：

```
find_package(MyDep REQUIRED)  
target_link_libraries(MyTarget PRIVATE MyDep::MyDep)
```

这就是自己编写 find 模块的方式。

Note

不要为拥有的包创建 Find*.cmake 模块。这些是为不支持 CMake 的包准备的。不过，可以创建一个 Config*.cmake 模块（本章后面内容）。

现在，来展示如何使用合适的包管理器，而不是手动进行下载依赖。

7.4.4 使用 Conan 包管理器

Conan 是一个开源的、分散的原生包管理器。它支持多种平台和编译器，可以与多个构建系统集成。

如果有包还没有为环境所构建，Conan 将在机器上构建它，而不是下载已经构建的版本。构建完成时，可以将它上传到公共存储库、私有的 `conan_server` 实例或 Artifactory 服务器。

准备使用 Conan

如果第一次运行 Conan，它将根据环境创建一个默认配置文件。可以通过创建一个新的配置文件，或更新默认配置文件来修改它的一些设置。假设正在使用 Linux，并且希望使用 GCC 9.x 编译所有东西。则可以使用如下代码：

```
conan profile new hosacpp  
conan profile update settings.compiler=gcc hosacpp  
conan profile update settings.compiler.libcxx=libstdc++11 hosacpp  
conan profile update settings.compiler.version=10 hosacpp  
conan profile update settings.arch=x86_64 hosacpp  
conan profile update settings.os=Linux hosacpp
```

如果依赖来自于其他存储库而不是默认存储库，可以使用 `conan remote add <repo> <repo_url>` 添加它们。例如，可以使用它来配置公司的一个库。

现在已经设置了 Conan，接下来展示如何使用 Conan 抓取依赖，并将所有依赖集成到 CMake 脚本中。

指定 Conan 的依赖

项目依赖于 C++ REST SDK。为了告诉 Conan 这一点，需要创建一个名为 `conanfile.txt` 的文件。在例子中，它将包含以下内容：

```
[requires]
cpprestsdk/2.10.18
```

```
[generators]
CMakeDeps
```

可以在这里指定任意数量的依赖项。它们中的每一个都可以有一个固定版本、固定版本的范围或一个标签，比如 latest。在@符号之后，可以找到拥有包的公司和可以选择包的特定版本（通常是稳定和测试）的通道。

生成器部分是指定想要使用的构建系统的地方。对于 CMake 项目，应该使用 CMakeDeps。还可以生成许多其他的工具，包括用于生成编译器参数、CMake 工具链文件、Python 虚拟环境等的工具。

在例子中，没有指定其他的选项，但可以很容易地进行添加，并为包及其依赖项配置变量。例如，要将依赖项编译为静态库，可以这样写：

```
[options]
cpprestsdk:shared=False
```

完成了 conanfile.txt 后，就可以让 Conan 使用它了。

安装 Conan 的依赖

要在 CMake 代码中使用 Conan 包，必须先安装。Conan 中，这意味着下载源代码并构建它们或下载预构建的二进制文件，以及创建将在 CMake 中使用的配置文件。要让 Conan 在创建构建目录后处理这个，应该跳转到其目录下，并运行以下命令：

```
conan install path/to/directory/containing/conanfile.txt --build=missing -s
build_type=Release -pr=hosacpp
```

默认情况下，Conan 希望将所有依赖项下载为预构建的二进制文件。如果服务器没有预先构建，Conan 将进行构建，而不是在传递--build=missing 时退出。这会告诉 Conan 获取使用与概要文件中相同的编译器和环境构建的发布版本。通过简单地调用 build_type 设置为其他 CMake 构建类型的另一个命令，这里可以为多个构建类型安装包。如果需要的话，这还可以在多个版本间快速的切换。如果使用默认配置文件（Conan 可以自动检测的配置文件），就不要使用-pr。

若计划使用的 CMake 生成器没有在 conanfile.txt 中指定，可以将它添加到前面的命令中。例如，要使用 compiler_args 生成器，应该添加--generator compiler_args。之后，可以使用它传递@conanbuildinfo.args 生成的内容到编译器端。

CMake 中使用 Conan

当 Conan 完成下载、构建和配置依赖项，则需要告诉 CMake 对它们进行使用。

如果使用 Conan 与 CMakeDeps 生成器，一定要为 `CMAKE_BUILD_TYPE` 指定一个值。其他情况下，CMake 将无法使用由 Conan 配置的包。调用（来自运行 Conan 的同一个目录）方法可以如下所示：

```
cmake path/to/directory/containing/CMakeLists.txt -DCMAKE_BUILD_TYPE=Release
```

这样，就可以在发布模式中构建项目，从而必须使用 Conan 安装的一个类型。要找到依赖项，可以使用 CMake 的 `find_package`：

```
list(APPEND CMAKE_PREFIX_PATH "${CMAKE_BINARY_DIR}")
find_package(cpprestsdk CONFIG REQUIRED)
```

首先，将根构建目录添加到 CMake 找到包配置文件的路径。然后，找到 Conan 生成的包配置文件。

要将 Conan 定义的目标作为目标的依赖项，最好使用有命名空间的目标名：

```
target_link_libraries(libcustomer PUBLIC cpprestsdk::cpprest)
```

在 CMake 的配置过程中没有找到包时，会得到一个错误。如果没有别名，将在尝试链接时得到一个错误。

现在已经按照想要的方式编译并链接了目标，是时候对进行测试了。

7.4.5 添加测试

CMake 有自己的测试驱动程序 CTest。从 CMakeLists 添加新的测试套件很容易，可以使用自己写的，也可以使用测试框架提供的。本书的后面部分，将深入讨论测试，这里先展示如何基于 GoogleTest 或 GTest 测试框架快速且干净地添加单元测试。

通常，要在 CMake 中已写好的测试，需要添加以下内容：

```
if(CMAKE_PROJECT_NAME STREQUAL PROJECT_NAME)
    include(CTest)
    if(BUILD_TESTING)
        add_subdirectory(test)
    endif()
endif()
```

前面的代码段将首先检查是否正在构建的主要项目。通常，只想为项目运行测试，甚至省略对使用的第三方组件构建测试。这就是检查项目名称的原因。

如果要运行测试，则需要包含 CTest 模块。这将加载 CTest 提供的整个测试基础设施，定义它的附加目标，并调用一个名为 `enable_testing` 的 CMake 函数，它将在其他事情中启用 `BUILD_TESTING` 标志。这个标志可以缓存，所以可以在构建项目时禁用所有的测试，只要在生成构建系统时传递 `-DBUILD_TESTING=OFF` 即可。

所有这些缓存变量都存储在构建目录中名为 CMakeCache.txt 的文本文件中。随意修改变量，改变 CMake 所做的。其不会覆盖文本中的设置，除非删除文件。可以使用 ccmake、cmake-gui 或手工完成。

如果 BUILD_TESTING 为 true，只需处理测试目录中的 CMakeLists.txt 文件即可。可以是这样的：

```
include(FetchGTest)
include(GoogleTest)

add_subdirectory(customer)
```

第一个包含调用前面描述的 GTest 的脚本。获取 GTest 后，当前的 CMakeLists.txt 通过 include(GoogleTest)，加载 GoogleTest 模块中定义的一些帮助函数。这能够更容易地将测试集成到 CTest 中。最后，通过 add_subdirectory(customer) 来告诉 CMake 包含一些测试目录。

test/customer/CMakeLists.txt 文件将添加一个带有测试的可执行文件，该可执行文件是用预定义的标志集和测试模块和 GTest 的链接编译的。然后，调用 CTest 助手函数来发现定义的测试。所有这些只是四行 CMake 代码：

```
add_executable(unittests unit.cpp)
target_compile_options(unittests PRIVATE ${BASE_COMPILE_FLAGS})
target_link_libraries(unittests PRIVATE domifair::libcustomer gtest_main)
gtest_discover_tests(unittests)
```

瞧！

现在，可以通过进入构建目录并使用以下命令来构建和执行测试：

```
cmake --build . --target unittests
ctest # or cmake --build . --target test
```

可以为 CTest 传递-j 标志。其工作原理就像 Make 或 Ninja 一样——并行执行测试。如果希望使用更短的构建命令，只需运行构建系统，即直接调用 make 就好。

Note

脚本中，最好使用较长的命令形式。这将使脚本独立于所使用的构建系统。

当测试通过后，就可以考虑将它们提供给更广泛的受众了。

7.5. 重用代码质量

CMake 有内置程序，在分发构建的结果时，这些程序可以大有作为。本节将描述安装和导出实用程序，以及它们之间的区别。后面的小节将展示如何使用 CPack 打包代码，以及如何使用 Conan 进行打包。

安装和导出对于微服务本身并不是那么重要，但交付库供他人使用，就就有用了。

7.5.1 安装

如果编写或使用过 Makefile，那么应该用过 `make install`，并看到过项目的可交付成果是如何安装在 OS 目录或选择的另一个目录中的。如果正在使用 make 与 CMake，使用本节的步骤将可以以相同的方式安装可交付成果。如果没有，仍然可以使用安装目标。除此之外，将有一种简单的方法来使用 CPack 根据安装命令来创建安装包。

如果在 Linux 系统上，通过调用下面的命令根据操作系统的惯例预先设置一些安装目录，可能是个不错的方式：

```
include(GNUInstallDirs)
```

这将使安装程序使用由 bin、lib 和其他类似目录组成的目录结构。这样的目录也可以手动设置一些 CMake 变量。

创建安装目标还包括几个步骤。首先，也是最重要的是定义想要安装的目标，在例子中如下所示：

```
install(
    TARGETS libcustomer customer
    EXPORT CustomerTargets
    LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
    ARCHIVE DESTINATION ${CMAKE_INSTALL_LIBDIR}
    RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR})
```

这告诉 CMake 将本章前面定义的库和可执行文件公开为 `CustomerTargets`，使用前面设置的目录。

如果计划将库的不同配置安装到不同的文件夹中，可以这样：

```
install(TARGETS libcustomer customer
    CONFIGURATIONS Debug
    # destinations for other components go here...
    RUNTIME DESTINATION Debug/bin)

install(TARGETS libcustomer customer
    CONFIGURATIONS Release
    # destinations for other components go here...
    RUNTIME DESTINATION Release/bin)
```

可以注意到，这里为可执行文件和库指定了目录，但没有为包含文件指定目录。这需要在另一个命令中提供：

```
install(DIRECTORY ${PROJECT_SOURCE_DIR}/include/
    DESTINATION include)
```

这意味着顶层 `include` 目录的内容将安装在安装根目录下的 `include` 目录中。第一个路径后的斜杠修复了一些路径问题，所以需要留意一下它。

现在有一系列的目标，需要生成一个文件，让另一个 CMake 项目可以了解其意图。这可以通过以下方式实现：

```
install(
    EXPORT CustomerTargets
    FILE CustomerTargets.cmake
    NAMESPACE domifair::
    DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/Customer)
```

这个命令接受目标集，并创建 CustomerTargets.cmake 文件，将包含关于目标和要求的信息。每个目标都有一个名称空间前缀，例如：customer 将变成 domifair::customer。生成的文件将安装在安装树中的库文件夹子目录中。

为了让依赖项目使用 CMake 的 `find_package` 找到相应的目标，需要提供一个 CustomerConfig.cmake 文件。如果目标没有依赖项，可以直接将前面的目标导出到该文件，而不是目标文件。否则，应该编写包含上述目标文件的配置文件。

例子中，想要重用一些 CMake 变量，所以需要创建一个模板，并使用 `configure_file` 对其进行填充：

```
configure_file(${PROJECT_SOURCE_DIR}/cmake/CustomerConfig.cmake.in
CustomerConfig.cmake @ONLY)
```

CustomerConfig.cmake.in 文件将从处理依赖关系开始：

```
include(CMakeFindDependencyMacro)
find_dependency(cpprestsdk 2.10.18 REQUIRED)
```

`find_dependency` 宏是用于配置文件中的 `find_package` 的包装器。虽然使用在 `conanfile.txt` 中定义 C++ REST SDK 2.10.18 的方式让 Conan 提供了相应的依赖，但这里需要再次指定这些依赖。生成的包可以在另一台机器上使用，因此也需要在那里安装依赖。如果想在目标机上使用 Conan，可以按照如下步骤安装 C++ REST SDK：

```
conan install cpprestsdk/2.10.18
```

处理了依赖关系后，配置文件模板将包括之前创建的目标文件：

```
if(NOT TARGET domifair::@PROJECT_NAME@)
    include("${CMAKE_CURRENT_LIST_DIR}/@PROJECT_NAME@Targets.cmake")
endif()
```

`configure_file` 执行时，将用项目中定义的与 \${VARIABLES} 匹配的内容替换所有的 @VARIABLES@。这样，基于 CustomerConfig.cmake.in 文件模板，将创建一个 CustomerConfig.cmake 文件。

使用 `find_package` 查找依赖项时，通常需要指定要查找的包的版本。为了在包中支持版本信息，必须创建 CustomerConfigVersion.cmake 文件。CMake 提供了一个帮助函数，可以创建这个文件。可以这样使用：

```
include(CMakePackageConfigHelpers)
write_basic_package_version_file(
CustomerConfigVersion.cmake
VERSION ${PACKAGE_VERSION}
COMPATIBILITY AnyNewerVersion)
```

PACKAGE_VERSION 将根据在顶层 CMakeLists.txt 文件顶部调用 project 时，传递的 VERSION 参数来对其进行填充。

若是新的或相同的版本请求，AnyNewerVersion COMPATIBILITY 表示包可以兼容。其他选项包括 SameMajorVersion, SameMinorVersion 和 ExactVersion。

在创建了配置文件和配置版本文件后，就可以告诉 CMake 它们应该与二进制文件和目标文件一起进行安装：

```
install(FILES ${CMAKE_CURRENT_BINARY_DIR}/CustomerConfig.cmake
${CMAKE_CURRENT_BINARY_DIR}/CustomerConfigVersion.cmake
DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/Customer)
```

最后一件事，这里应该为项目安装许可证。可以利用 CMake 的命令来安装文件，把它们放在文档目录中：

```
install(
FILES ${PROJECT_SOURCE_DIR}/LICENSE
DESTINATION ${CMAKE_INSTALL_DOCDIR})
```

要成功地在操作系统的根目录中创建安装目标，只需要知道这些。这里可能想了解如何将包安装到另一个目录，例如仅为当前用户安装。为此，需要在生成构建系统时，设置 CMAKE_INSTALL_PREFIX。

注意，若不想安装到 Unix 树的根目录中，将需要为依赖项目提供一个安装目录的路径，可以通过 CMAKE_PREFIX_PATH 设置。

现在来了解一下重用构建内容的另一种方法。

7.5.2 导出

导出是一种将关于本地构建包的信息添加到 CMake 包注册表的技术。当目标在构建目录中直接可见时，即使没有安装这也很有用的。导出的一个常见的场景是在本地进行构建，但在开发机器上使用它们。

从 CMakeLists.txt 文件中添加对这种机制的支持非常容易：

```
export(
TARGETS libcustomer customer
NAMESPACE domifair::
FILE CustomerTargets.cmake)
```

```
set(CMAKE_EXPORT_PACKAGE_REGISTRY ON)
export(PACKAGE domifair)
```

CMake 将创建类似于安装部分的目标文件，在提供的命名空间中定义库和可执行目标。自 CMake 3.15 起，包注册表默认禁用，所以需要通过设置适当的前置变量来启用。然后，可以通过导出包将关于目标的信息直接放入注册表中。

注意，现在有了一个目标文件，但没有匹配的配置文件。若目标依赖于外部库，必须在找到包之前找到它们。在我们的例子中，调用必须按以下方式排序：

```
find_package(cpprestsdk 2.10.18)
find_package(domifair)
```

首先，找到 C++ REST SDK，然后寻找依赖于它的包。这就是开始输出目标文件所需要知道的全部信息。这比安装它们容易多了，不是吗？

现在，来了解第三种将目标暴露在外部的方法。

7.5.3 CPack

本节中，将描述如何使用 CMake 自带的打包工具 CPack。CPack 可以轻松地创建各种格式的包，从 ZIP 和 TGZ 压缩包到 DEB 和 RPM 包，甚至包括 NSIS 或一些 OS X-specific 的安装向导。当安装逻辑就位，集成该工具就不难了。现在就来展示如何使用 CPack 来打包项目。

首先，指定 CPack 在创建包时使用的变量：

```
set(CPACK_PACKAGE_VENDOR "Authors")
set(CPACK_PACKAGE_CONTACT "author@example.com")
set(CPACK_PACKAGE_DESCRIPTION_SUMMARY
    "Library and app for the Customer microservice")
```

需要手工提供一些信息，但是可以根据定义项目时指定的项目版本填充变量。还有更多的 CPack 变量，可以在本章末尾的扩展阅读部分的 CPack 链接中进行了解。其中一些是所有包生成器通用的，而一些是特定于相应生成器的。例如，若计划使用安装程序，可以设置以下两个变量：

```
set(CPACK_RESOURCE_FILE_LICENSE "${PROJECT_SOURCE_DIR}/LICENSE")
set(CPACK_RESOURCE_FILE_README "${PROJECT_SOURCE_DIR}/README.md")
```

设置好的变量后，就可以选择 CPack 使用的生成器了。先在 CPACK_GENERATOR 中放入一些基本的代码，CPACK 依赖于一个变量：

```
list(APPEND CPACK_GENERATOR TGZ ZIP)
```

这将使 CPack 根据本章前面定义的安装步骤，生成两种类型的压缩文件。

可以根据许多东西选择不同的包生成器，比如正在运行的机器上可用的工具。在 Windows 上构建时创建 Windows 安装程序，在 Linux 上构建时安装了适当的工具，则创建 DEB 或 RPM 包。如果是在 Linux 上运行，可以检查 dpkg 是否安装，如果安装了，则创建 DEB 包：

```

if(UNIX)
    find_program(DPKG_PROGRAM dpkg)
    if(DPKG_PROGRAM)
        list(APPEND CPACK_GENERATOR DEB)
        set(CPACK_DEBIAN_PACKAGE_DEPENDS "${CPACK_DEBIAN_PACKAGE_DEPENDS}
libcpprest2.10 (>= 2.10.2-6)")
        set(CPACK_DEBIAN_PACKAGE_SHLIBDEPS ON)
    else()
        message(STATUS "dpkg not found - won't be able to create DEB
packages")
    endif()

```

使用 CPACK_DEBIAN_PACKAGE_DEPENDS 变量使 DEB 包要求首先安装 C++ REST SDK。

对于 RPM 包，可以手动检查 rpmbuild:

```

find_program(RPMBUILD_PROGRAM rpmbuild)
if(RPMBUILD_PROGRAM)
    list(APPEND CPACK_GENERATOR RPM)
    set(CPACK_RPM_PACKAGE_REQUIRES "${CPACK_RPM_PACKAGE_REQUIRES}
cpprest >= 2.10.2-6")
else()
    message(STATUS "rpmbuild not found -
    won't be able to create RPM packages")
endif()
endif()

```

很优雅，对吧？

这些生成器提供了很多变量，如果需要比这里描述的这些基本需求更多的东西，可以查阅 CMake 文档。

最后，当涉及变量时——还可以使用它们来避免打包不需要的文件。可以通过以下方式实现:

```
set(CPACK_SOURCE_IGNORE_FILES /.git /dist /.*build.* /\*\*.DS_Store)
```

当所有的工具都准备好，就可以在 CMakeLists 中包括 CPack 了:

```
include(CPack)
```

记住，在最后一步总是这样做，因为 CMake 不会传播后续使用的变量到 CPack 中。

可以直接调用 cpack 或更长的形式运行，其会检查是否需要重新构建:cmake --build . --target package。如果只需要使用-G 标志重新构建一种类型的包，则可以覆盖生成器。例如，-G DEB 只构建 DEB 包，-G WIX -C Release 打包一个发行版 MSI 可执行文件，或-G DragNDrop 可以得到一个 DMG 安装程序。

接下来，让我们讨论一种更野蛮的方法（进行包的构建）。

7.6. 使用 Conan 进行打包

已经展示了如何使用 Conan 安装依赖项。现在，尝试创建自己的 Conan 包。

先在项目中创建一个新的顶层目录，命名为 conan，将在其中存储使用此工具打包所需的文件：用于构建包的脚本和测试环境。

7.6.1 创建 conanfile.py 脚本

所有 Conan 包需要的文件是 conanfile.py。在本例中，希望使用 CMake 变量填充它的一些细节，因此来创建一个 conanfile.py.in 文件。使用它来创建文件，并添加以下文件到 CMakeLists.txt 文件中：

```
configure_file(${PROJECT_SOURCE_DIR}/conan/conanfile.py.in  
${CMAKE_CURRENT_BINARY_DIR}/conan/conanfile.py @ONLY)
```

文件将从 Python 的 import 开始，比如 Conan 在 CMake 项目中需要的那些：

```
import os  
from conans import ConanFile, CMake
```

需要创建一个定义包的类：

```
class CustomerConan(ConanFile):  
    name = "customer"  
    version = "@PROJECT_VERSION@"  
    license = "MIT"  
    author = "Authors"  
    description = "Library and app for the Customer microservice"  
    topics = ("Customer", "domifair")
```

首先，从一堆通用变量开始，从 CMake 代码中获取项目版本。通常，描述将是一个多行字符串。这些主题对于在 JFrog 的 Artifactory 等网站上找到库很有用，并且可以告诉读者我们的包是什么的。现在来看看其他变量：

```
homepage = "https://example.com"  
url =  
"https://github.com/PacktPublishing/Hands-On-Software-Architecture-with-Cpp  
/"
```

homepage 应该指向项目的主页：文档、教程、FAQ 和类似的东西都可以放在那里。另一方面，url 是放置包存储库的位置。许多开放源码库在一个库中包含代码，而在另一个库中包含打包代码。还有种情况是，包是由中央的 Conan 包服务器构建的，这样 url 应该指向 <https://github.com/conan-io/conan-center-index>。

接下来，可以指定如何构建我们的包：

```
settings = "os", "compiler", "build_type", "arch"  
options = {"shared": [True, False], "fPIC": [True, False]}  
default_options = {"shared": False, "fPIC": True}  
generators = "CMakeDeps"  
keep_imports = True # useful for repackaging, e.g. of licenses
```

`settings` 将确定是否需要构建包，或者可以下载已构建的版本。

`options` 和 `default_options` 可以是任何值。`shared` 和 `fPIC` 是大多数包提供的两个功能，所以可以遵循这个约定。

现在已经定义了变量，就开始编写 Conan 用来打包软件的方法。首先，指定包的使用者应该链接的库：

```
def package_info(self):
    self.cpp_info.libs = ["customer"]
```

`self.cpp_info` 对象允许进行更多的设置，但这是最小值。可以查看 Conan 文档中的其他属性。

接下来，指定其他需要的包：

```
def requirements(self):
    self.requires.add('cpprestsdk/2.10.18')
```

这次，直接从 Conan 获取 C++ REST SDK，而不是指定操作系统的包管理器应该依赖哪些包。现在，来指定 CMake 应该如何（以及在哪里）生成构建系统：

```
def _configure_cmake(self):
    cmake = CMake(self)
    cmake.configure(source_folder="@CMAKE_SOURCE_DIR@")
    return cmake
```

本例中，可以简单地将其指向源目录。当构建系统配置好后，就需要构建项目了：

```
def build(self):
    cmake = self._configure_cmake()
    cmake.build()
```

Conan 还支持非 CMake 的构建系统。构建完包之后，就可以打包了，这需要提供另一种方法：

```
def package(self):
    cmake = self._configure_cmake()
    cmake.install()
    self.copy("license*", ignore_case=True, keep_path=True)
```

注意我们是如何使用相同的 `_configure_cmake()` 函数来构建和打包项目的。除了安装二进制文件之外，还需要指定在哪里存放部署许可证。最后，告诉 Conan 在安装包时应该复制哪些东西：

```
def imports(self):
    self.copy("license*", dst="licenses", folder=True,
             ignore_case=True)

    # Use the following for the cmake_multi generator on Windows
    # and/or Mac OS to copy libs to the right directory.

    # Invoke Conan like so:
    # conan install . -e CONAN_IMPORT_PATH=Release -g cmake_multi
    dest = os.getenv("CONAN_IMPORT_PATH", "bin")
    self.copy("*.dll", dst=dest, src="bin")
    self.copy("*.dylib*", dst=dest, src="lib")
```

上面的代码指定在安装库时，在何处解压缩许可证文件、库和可执行文件。

现在，了解了如何构建一个 Conan 包，接下来来测试安装包。

7.6.2 测试 Conan 包

当 Conan 构建了程序包，就应该测试这个包否构建正确。首先在 conan 目录中创建一个 test_package 子目录。

还需要包含一个 conanfile.py 脚本，但这次是一个更小的脚本。应该是这样开始的：

```
import os
from conans import ConanFile, CMake, tools

class CustomerTestConan(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    generators = "CMakeDeps"
```

这里没什么特别的，提供构建测试包的逻辑：

```
def build(self):
    cmake = CMake(self)
    # Current dir is "test_package/build/<build_id>" and
    # CMakeLists.txt is in "test_package"
    cmake.configure()
    cmake.build()
```

马上就到写 CMakeLists.txt 文件的环节了。在写之前，还需要再完成两件事：导入方法和测试方法。导入方法可以这样写：

```
def imports(self):
    self.copy("*.dll", dst="bin", src="bin")
    self.copy("*.dylib*", dst="bin", src="lib")
    self.copy('*.so*', dst='bin', src='lib')
```

然后，就有了包测试逻辑的核心——测试方法：

```
def test(self):
    if not tools.cross_building(self.settings):
        self.run("./sexample" % os.sep)
```

我们只希望在为本地架构构建时才运行它。否则，可能无法运行已编译的可执行文件。

现在，就可以来写 CMakeLists.txt 文件了：

```
cmake_minimum_required(VERSION 3.12)

project(PackageTest CXX)

list(APPEND CMAKE_PREFIX_PATH "${CMAKE_BINARY_DIR}")

find_package(customer CONFIG REQUIRED)
```

```

add_executable(example example.cpp)
target_link_libraries(example customer::customer)

# CTest tests can be added here

```

就这么简单，链接到所有提供的 Conan 库（在例子中，只有 Customer 库）。

最后，检查包是否成功创建了 example.cpp 文件：

```

1 #include <customer/customer.h>
2 int main() { responder{}.prepare_response("Conan"); }

```

运行这些前，需要在 CMake 的主构建树中做一些小的更改。现在来了解一下，如何正确地从 CMake 文件中导出 Conan 目标。

7.6.3 添加 Conan 打包代码到 CMakeLists

还记得在重用质量代码的章节中编写的安装逻辑吗？如果依赖于 Conan 进行打包，可能不需要运行裸 CMake 导出和安装逻辑。假设只需要在不使用 Conan 的情况下导出和安装，就需要修改 CMakeLists：

```

if(NOT CONAN_EXPORTED)

    install(
        EXPORT CustomerTargets
        FILE CustomerTargets.cmake
        NAMESPACE domifair::
        DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/Customer)

    configure_file(${PROJECT_SOURCE_DIR}/cmake/CustomerConfig.cmake.in
        CustomerConfig.cmake @ONLY)

    include(CMakePackageConfigHelpers)
    write_basic_package_version_file(
        CustomerConfigVersion.cmake
        VERSION ${PACKAGE_VERSION}
        COMPATIBILITY AnyNewerVersion)

    install(FILES ${CMAKE_CURRENT_BINARY_DIR}/CustomerConfig.cmake
        ${CMAKE_CURRENT_BINARY_DIR}/CustomerConfigVersion.cmake
        DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/Customer)
endif()

install(

```

```
FILE ${PROJECT_SOURCE_DIR}/LICENSE
DESTINATION
${IF:$<BOOL:$CONAN_EXPORTED>,licenses,${CMAKE_INSTALL_DOCDIR}})
```

添加 if 语句和一个生成器表达式对于拥有干净的包来说是合理的，这就是所要做的事情了。

最后一件事情——可以构建一个目标来创建 Conan 包：

```
add_custom_target(
    conan
    COMMAND
        ${CMAKE_COMMAND} -E copy_directory
        ${PROJECT_SOURCE_DIR}/conan/test_package/
        ${CMAKE_CURRENT_BINARY_DIR}/conan/test_package
    COMMAND conan create . customer/testing -s build_type=$<CONFIG>
    WORKING_DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}/conan
    VERBATIM)
```

当运行 `cmake --build .--target conan`(或 `ninja conan` 若已使用生成器并想命令行简单一些)，CMake 将复制 `test_package` 目录到构建文件夹，构建 conan 包，并使用复制的文件对其进行测试。

全部完成！

Note

涉及到创建 Conan 包的内容远不止在这里描述到的，更多信息请参考 Conan 的文档。可以在扩展阅读找到相应的链接。

7.7. 总结

本章中，了解了很多关于构建和打包的知识。现在能够编写更快的构建模板代码，知道如何选择工具来更快地编译代码(将在下一章了解更多关于工具的知识)，并且知道何时使用直接声明，而不是`#include`。

除此之外，可以使用现代 CMake 来定义构建目标和测试套件，使用 `find` 模块和 `FetchContent` 来管理外部依赖，创建各种格式的包和安装程序，再使用 Conan 来安装依赖和创建自己的工件。

下一章中，将了解如何编写易于测试的代码。持续集成和持续部署在拥有良好的测试覆盖率时才有用。没有全面测试的持续部署，更容易向生产环境引入新的 bug，但这并不是设计软件架构时的目标。

7.8. 练习题

1. CMake 中安装和导出目标有什么区别？
2. 如何使模板代码编译得更快？
3. 如何在 Conan 中使用多个编译器？

4. 如果想用 C++11 前的 GCC ABI 来编译 Conan 依赖，该怎么做？
5. 如何使用 CMake 强制指定 C++ 标准？
6. 如何在 CMake 中构建文档，并随 RPM 包一起发布？

7.9. 扩展阅读

- List of compiler books on GCC's wiki: <https://gcc.gnu.org/wiki/ListOfCompilerBooks>
- Type Based Template Metaprogramming is Not Dead, a lecture by Odin Holmes, C++Now 2017: <https://www.youtube.com/watch?v=EtU4RDCCsiU>
- The Modern CMake online book: <https://cliutils.gitlab.io/modern-cmake>
- Conan documentation: <https://docs.conan.io/en/latest/>
- CMake documentation on creating find scripts: <https://cmake.org/cmake/help/v3.17/manual/cmake-developer.7.html?highlight=find#a-sample-findmodule>

第三部分：体系架构的质量

关注使软件项目成功的高级概念，展示有助于保持软件高质量的工具。

包括以下几章：

- 第 8 章，编写可测试的代码
- 第 9 章，持续集成和持续部署
- 第 10 章，代码和部署的安全性
- 第 11 章，性能

第 8 章 编写可测试的代码

测试代码的能力是软件产品最重要的品质。如果没有适当的测试，重构代码或改进代码的其他部分（如安全性、可扩展性或性能）的代价将会非常昂贵。本章中，将了解如何设计和管理自动化测试，以及在必要的时候正确地使用 fake 和 mock。

本章将讨论以下内容：

- 为什么要写测试代码？
- 测试框架
- 测试中的 mock 和 fake
- 测试驱动的类型设计
- 在持续集成/持续部署上进行自动化测试

8.1. 相关准备

本章的代码可以在以下 GitHub 页面找到：<https://github.com/PacktPublishing/Software-Architecture-with-Cpp/tree/master/Chapter08>。

本章的例子中，将使用的软件如下所示：

- GTest 1.10+
- Catch2 2.10+
- CppUnit 1.14+
- Doctest 2.3+
- Serverspec 2.41+
- Testinfra 3.2+
- Goss 0.3+
- CMake 3.15+
- Autoconf
- Automake
- Libtool

8.2. 为什么要写测试代码？

软件工程和软件架构放在一起会非常复杂，处理不确定性的方法是确保不受潜在风险的影响。我们在人寿保险、健康保险和汽车保险上一直都是这么做的。然而，当涉及到软件开发时，往往会忘记所有的安全预防措施，而只是期望一个乐观的结果。

不仅可能而且会出错，但难以置信的是，测试软件的话题仍然是一个有争议的话题。无论是由于缺乏技能还是缺乏预算，仍然有一些项目缺乏最基本的测试。当客户决定改变需求时，一个简单的修改可能会导致无休止的返工和战争。

当第一次返工发生时，由于没有执行适当的测试而节省的时间就浪费了。如果认为返工不会很快发生，很可能是大错特错的。如今所处的敏捷环境中，返工是日常生活的一部分。我们对世界和客户变更的了解意味着需求的变更，以及对代码的修改。

因此，测试的主要目的是在项目后期节约时间。当需要执行各种测试而不是只关注功能时，这是一项早期投资，但这是一项不会后悔的投资。就像保险政策一样，当事情按照计划进行时，测试需要从预算中拿出一小部分，但当事情变得糟糕时，将获得一笔丰厚的回报。

8.2.1 测试金字塔

设计或实现软件系统时，可能会遇到不同类型的测试。每个类的用途略有不同，可以分为以下几类：

- 单元测试：代码
- 集成测试：设计
- 系统测试：需求
- 验收测试（端到端或 E2E）：客户的需求

这种区分比较随意，可能经常会看到不同的金字塔，如下所示：

- 单元测试
- 服务测试
- UI 测试（端到端或 E2E）

这里，单元测试指的是与前面示例相同的层。服务测试是集成测试和系统测试的结合。另一方面，UI 测试是指验收测试。

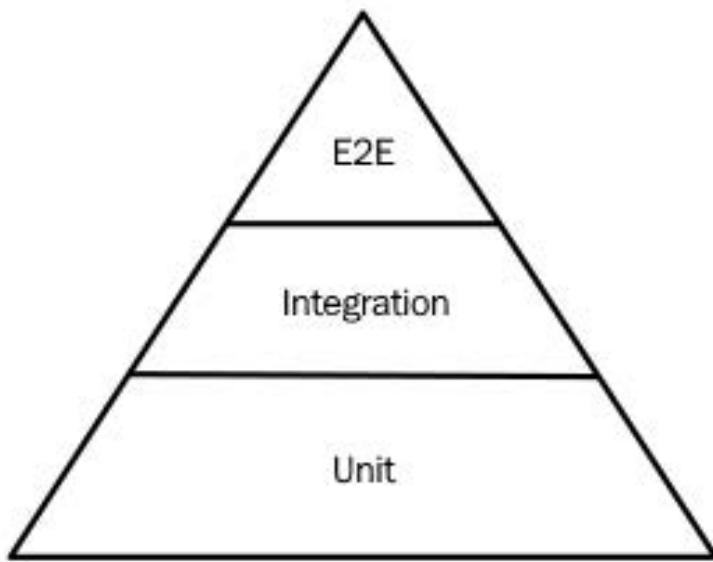


图 8.1 - 测试金字塔

值得注意的是，单元测试不仅构建成本最低，而且执行速度也非常快，通常可以并行。这就可以创造一个持续集成控制机制。不仅如此，还经常提供关于系统健康状况的最佳反馈。更高级别的测试不仅难以正确编写，而且还可能不够健壮。这可能导致测试结果不稳定，每几次测试运行就有一个会失败。如果高级测试中的失败与单元测试级别上的失败不相关，那么问题很可能是测试本身的问题。

不想说高级测试完全没用，不过应该只专注于编写单元测试，但事实并非如此。看金字塔的形状，因为单元测试应该覆盖基础框架。然而，在此基础上，也应该以适当的比例安排高级测试。不

很难想象这样一个系统：所有的单元测试都通过了，但系统本身却没有为客户提供价值。一个极端的例子是完美工作的后端，但没有用户界面（无论是图形界面还是以 API 的形式）。当然，它通过了所有的单元测试，但这都不是无价值的借口！

可以想象，与测试金字塔相反的是冰锥，这是一个反模式。违反测试金字塔通常会导致脆弱的代码和难以跟踪的 bug。这使得调试成本的激增，并且在测试开发中也没有节省成本。

8.2.2 非功能性测试

已经介绍了所谓的功能测试，目的是检查测试系统是否满足功能需求。但是除了功能性需求之外，还有其他类型的需求是想要控制的。其中一些如下所示：

- 性能：应用程序可能会根据功能方面的需求运行，但由于性能较差，最终用户仍然无法使用。我们将在第 11 章中更多地关注如何提高性能。
- 持久性：即使系统确实性能很好，但这并不意味着可以承受持续的高工作负载。当这样做时，它能经受住一些部件的故障吗？当接受了每个软件都是脆弱的，并且可能在任何时刻发生故障的想法时，就要开始设计能够抗故障的系统了。这是 Erlang 生态系统所采用的一个概念，但这个概念本身并不仅限于 Erlang。第 13 章和第 15 章中，将更多地提到设计容错系统和混沌工程的作用。
- 安全性：没有必要重复安全性的重要性吧！但由于没有得到严肃对待，我们将多次提醒，甚至让开发者感到厌烦。每个连接到网络的系统都可能——而且很可能会——被破坏。在开发过程中应尽早执行安全性测试与其他类型的测试相同：可以在问题修复成本过高之前捕获它们。
- 可用性：虽然较差的性能可能会让用户不选择您的产品，但较差的可用性甚至可能让用户都不理您的产品。虽然性能过载可能会导致可用性问题，但也有其他原因导致的可用性问题。
- 完整性：客户数据不应该只受到外部攻击者的保护。它也应该是安全的，不会因为软件故障而发生任何更改或损失。防止位衰减、快照和备份是防止完整性丢失的方法。通过将当前版本与以前记录的快照进行比较，可以确定差异是由所采取的操作造成的，还是由错误造成的。
- 可用性：即使产品符合上述所有条件，如果有一个笨拙的界面和不直观的交互，用户可能仍然不满意。可用性测试大多是手动执行的。每当 UI 或系统的工作流发生变化时，执行可用性评估就很重要。

8.2.3 回归测试

回归测试通常是端到端测试，可以避免在同一地方跌倒两次。当（QA 团队或客户）在生产系统中发现一个 bug 时，仅仅应用一个热修复程序，并将其遗忘是不行的。

这里需要做的事就是编写回归测试，以防止相同的错误再次进入系统。好的回归测试甚至可以防止同类错误进入生产环境中。毕竟，当知道自己做错了什么，就可以想象其他把事情搞砸的方式。然后，可以做的另一件事是进行原因分析。

8.2.4 原因分析

原因分析是用来发现问题的来源的过程，而不仅仅是一个形式。最常见的方法是使用丰田公司著名的 5 个为什么的方法。这种方法包括剥去问题表象的所有表层，找出隐藏在下面的根本原因。在每一层都问“为什么”，直到找到根本原因。

来看一个实际使用该方法的示例。

问题: 我们的一些交易没有收到付款:

1. 为什么? 系统没有向客户发送合适的邮件。
2. 为什么? 邮件发送系统不支持客户姓名中的特殊字符。
3. 为什么? 邮件发送系统测试不正常。
4. 为什么? 由于需要开发新特性, 所以没有时间进行适当的测试。
5. 为什么? 对功能的时间估计不正确。

这个例子中, 时间估计的问题可能是在生产系统中发现的错误的根本原因, 但这也可能是另一层需要剥离的东西。该框架提供了一种启发式的方法, 在大多数情况下是有效的, 但如果不能完全确定所得到的就是根本原因, 可以继续剥离额外的层, 直到找到造成所有问题的原因为止。

考虑到许多错误都是由完全相同, 且通常是可重复的根本原因导致的, 找到根本原因很有用, 这样就可以保护自己在未来的几个不同级别上避免犯下相同的错误。这是应用于软件测试和解决问题时的深度防御原则。

8.2.5 改进的基础

测试代码可以避免意外的错误, 但也开启了不同的可能性。当代码使用测试用例覆盖时, 不必担心重构。重构是将完成相应工作的代码转换为功能相似代码的过程, 只是它有更好的内部组织。这里可能想知道为什么需要更改代码的组织, 这可能有几个原因。

首先, 代码可能不具有可读性, 这意味着每次修改都需要花费太多的时间。其次, 修复的 bug 会导致其他一些特性的行为不正确, 因为随着时间的推移, 代码收集了太多的变通方法和特殊情况。这两个原因都可以归结为生产力的提高。从长远来看, 会使维护成本更低。

但除了提高工作效率, 可能还想提高工作表现。这意味着运行时性能(应用程序在生产环境中的行为)或编译时性能(这是另一种形式的生产率提高)。

通过将当前的次优算法替换为更高效的算法, 或者通过正在重构的模块更改所使用的数据结构, 可以重构运行时性能。

为了提高编译时性能而进行的重构, 通常包括将部分代码移动到不同的编译单元、重新组织头文件或减少依赖关系。

无论最终目标是什么, 重构通常都是一件有风险的事情。选择了一些基本可以正常工作的内容, 但最终可能会得到更好的版本或更差的版本。怎么知道哪个箱子是对的? 这里, 测试起到了决定性的作用。

如果当前的特性集完全覆盖, 并且想要修复最近发现的 bug, 那么需要做的就是添加一个将在那时失败的测试用例。当整个测试套件开始通过时, 意味着重构工作的成功。

最坏的情况是, 若不能在指定的时间范围内满足所有测试用例, 必须中止重构。如果想提高性能, 可以采用类似的过程, 但不是单元测试(或端到端测试), 而是将重点放在性能测试上。

随着最近帮助重构和代码维护的自动化工具的兴起(如 ReSharper C++: <https://www.jetbrains.com/resharper-cpp/features/ReSharperC++>), 现在甚至可以将一部分代码单独外包给外部软件服务。诸如 Renovate 等服务(<https://renovatebot.com/>), Dependabot(<https://dependabot.com>) 和 Greenkeeper (<https://greenkeeper.io/>) 可能很快就会支持 C++。拥有可靠的测试覆盖, 可以不用担心在依赖更新期间破坏现有的应用程序。

应该考虑让依赖项在安全漏洞方面保持最新，这样的服务可以减轻负担。因此，测试不仅可以防止犯错，还可以减少引入新特性所需的工作量。还可以改进代码库，并保持其稳定和安全！

理解了测试的必要性，可以开始编写自己的测试。先在没有外部依赖的情况下编写测试，目前只关注测试逻辑。暂时对管理测试结果和报告的细节不感兴趣。因此，将选择一个测试框架来处理这项乏味的工作。下一节中，将介绍一些主流的测试框架。

8.3. 测试框架

至于框架，目前的标准是 Google 的 GTest，与对应的 GMock 组成了一个小型工具套件，可以遵循 C++ 测试的最佳实践。

GTest/GMock 组合的其他替代方案是 Catch2、CppUnit 和 Doctest。CppUnit 已经存在很长一段时间了，但缺少更新，所以不推荐使用。Catch2 和 Doctest 都支持现代 C++ 标准——特别是 C++14、C++17 和 C++20。

为了比较这些测试框架，将使用想要测试的代码库。以它为基础，在每个框架中实现测试。

8.3.1 GTest

下面是用 GTest 编写的一个测试示例：

```
1 #include "customer/customer.h"
2
3 #include <gtest/gtest.h>
4
5 TEST(basic_responses,
6 given_name_when_prepare_responses_then_greets_friendly) {
7     auto name = "Bob";
8     auto code_and_string = responder{}.prepare_response(name);
9     ASSERT_EQ(code_and_string.first, web::http::status_codes::OK);
10    ASSERT_EQ(code_and_string.second, web::json::value("Hello, Bob!"));
11 }
```

测试期间大部分任务已经抽象，这里主要关注于提供测试的动作 (`prepare_response`) 和所需的行为 (有 `ASSERT_EQ` 的行)。

8.3.2 Catch2

下面是用 Catch2 编写的一个测试示例：

```
1 #include "customer/customer.h"
2
3 #define CATCH_CONFIG_MAIN // This tells Catch to provide a main() - only do
4 // this in one cpp file
5 #include "catch2/catch.hpp"
6
7 TEST_CASE("Basic responses",
8 "Given Name When Prepare Responses Then Greets Friendly") {
9     auto name = "Bob";
```

```

10 auto code_and_string = responder{}.prepare_response(name);
11 REQUIRE(code_and_string.first == web::http::status_codes::OK);
12 REQUIRE(code_and_string.second == web::json::value("Hello, Bob!"));
13 }

```

看起来和前一个例子很像。有些关键字不同 (TEST 和 TEST_CASE)，检查结果的方式也不同 (使用 REQUIRE(a == b) 而不是 ASSERT_EQ(a, b))。

8.3.3 CppUnit

下面是用 CppUnit 编写的测试示例，将把它分成几个片段。

代码为使用 CppUnit 库中的构造做了准备工作：

```

1 #include <cppunit/BriefTestProgressListener.h>
2 #include <cppunit/CompilerOutputter.h>
3 #include <cppunit/TestCase.h>
4 #include <cppunit/TestFixture.h>
5 #include <cppunit/TestResult.h>
6 #include <cppunit/TestResultCollector.h>
7 #include <cppunit/TestRunner.h>
8 #include <cppunit/XmlOutputter.h>
9 #include <cppunit/extensions/HelperMacros.h>
10 #include <cppunit/extensions/TestFactoryRegistry.h>
11 #include <cppunit/ui/text/TextTestRunner.h>
12
13 #include "customer/customer.h"
14
15 using namespace CppUnit;
16 using namespace std;

```

接下来，必须定义测试类并实现执行测试用例的方法。之后，注册这个类，以便在测试运行器中使用：

```

1 class TestBasicResponses : public CppUnit::TestFixture {
2     CPPUNIT_SUITE(TestBasicResponses);
3     CPPUNIT_TEST(testBob);
4     CPPUNIT_SUITE_END();
5
6 protected:
7     void testBob();
8 };
9
10 void TestBasicResponses::testBob() {
11     auto name = "Bob";
12     auto code_and_string = responder{}.prepare_response(name);
13     CPPUNIT_ASSERT(code_and_string.first == web::http::status_codes::OK);
14     CPPUNIT_ASSERT(code_and_string.second == web::json::value("Hello,
15         Bob!"));
16 }
17

```

```
18 CPPUNIT_TEST_SUITE_REGISTRATION(TestBasicResponses);
```

最后，必须提供测试运行器的行为：

```
1 int main() {
2     CPPUNIT_NS::TestResult testresult;
3
4     CPPUNIT_NS::TestResultCollector collectedresults;
5     testresult.addListener(&collectedresults);
6     CPPUNIT_NS::BriefTestProgressListener progress;
7     testresult.addListener(&progress);
8     CPPUNIT_NS::TestRunner testrunner;
9     testrunner.addTest(CPPUNIT_NS::TestFactoryRegistry::getRegistry().makeTest(
10    ));
11    testrunner.run(testresult);
12
13    CPPUNIT_NS::CompilerOutputter compileroutputter(&collectedresults,
14        std::cerr);
15    compileroutputter.write();
16
17    ofstream xmlFileOut("cppTestBasicResponsesResults.xml");
18    XmlOutputter xmlOut(&collectedresults, xmlFileOut);
19    xmlOut.write();
20
21    return collectedresults.wasSuccessful() ? 0 : 1;
22 }
```

与前面的两个示例相比，这里有很多样板文件，而测试看起来与前面的示例非常相似。

8.3.4 Doctest

下面是用 Doctest 编写的一个测试示例：

```
1 #include "customer/customer.h"
2
3 #define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
4 #include <doctest/doctest.h>
5
6 TEST_CASE("Basic responses") {
7     auto name = "Bob";
8     auto code_and_string = responder{}.prepare_response(name);
9     REQUIRE(code_and_string.first == web::http::status_codes::OK);
10    REQUIRE(code_and_string.second == web::json::value("Hello, Bob!"));
11 }
```

代码很清晰，也很容易理解。Doctest 的主要卖点是，与其他具有类似特性的替代产品相比，在编译时和运行时都最快。

8.3.5 编译时测试

模板元编程允许编写在编译时执行的 C++ 代码。C++11 中添加的 `constexpr` 关键字允许开发者使用更多的编译时代码，而 C++20 中的 `consteval` 关键字旨在对代码的计算方式有更大的控制权。

编译时编程的问题是没有简单的方法来进行测试。虽然用于执行时代码的单元测试框架非常丰富（正如刚才看到的），但关于编译时编程的资源并不多。部分原因可能是编译时编程仍然很复杂，而且只针对专家。

尽管如此，不容易并不意味着不可能。就像执行时间测试依赖于在运行时检查断言一样，可以使用 `static_assert` 检查编译时代码的行为，`static_assert` 在 C++11 中与 `constexpr` 一起引入。

下面是使用 `static_assert` 的简单示例：

```
1 #include <string_view>
2 constexpr int generate_lucky_number(std::string_view name) {
3     if (name == "Bob") {
4         number = number * 7 + static_cast<int>(letter);
5     }
6     return number;
7 }
8
9 static_assert(generate_lucky_number("Bob") == 808);
```

因为可以在编译时计算这里测试的每个值，所以可以使用编译器作为测试框架。

8.4. 测试中的 mock 和 fake

只要测试函数不与外部有太多交互，事情就会很简单很多。当正在测试的单元与第三方组件（如数据库、HTTP 连接和特定文件）连接时，问题就出现了。

一方面，希望查看代码在各种环境下的行为。另一方面，不希望等待数据库启动，而且肯定不希望多个数据库包含不同版本的数据，以便检查所有必要条件。

要如何处理这种情况呢？我们不想执行触发所有这些副作用的实际代码，而是使用测试替身。测试替身是代码中模拟实际 API 的结构，只是它们不执行模拟函数或对象的操作。

最常见的测试替身是 `mock`、`fake` 和 `stub`。虽然它们不相同，但很相似，许多人会认错。

8.4.1 不同的测试替身

`mock` 是一种测试替身，注册所有接收到的调用，但仅此而已。不返回任何值，也不会以任何方式改变状态。当有第三方框架来调用代码时，这就很有用。通过使用模拟，可以观察所有调用，从而能够验证框架的行为是否符合预期。

`stub` 的实现稍微复杂一些。具有返回值，这些值已经定义。`StubRandom.randomInteger()` 方法总是返回相同的值（例如，3），这似乎令人惊讶，但当测试返回值的类型或返回值时，可能是正确的 `stub` 实现。具体值是什么可能就不那么重要了。

最后，fake 是具有工作实现的对象，其行为与实际的生产实现基本相同。区别是，fake 数据库可能采取各种快捷方式，避免调用生产数据库或文件系统。

实现命令查询分离 (Command Query Separation, CQS) 设计模式时，可以使用 stub 进行双重查询，使用 mock 进行命令查询。

8.4.3 测试替身的其他用法

在检测之外，fake 也可以在有限的范围内使用。在内存中处理数据而不需要数据库访问，这对于创建原型或遇到性能瓶颈时也很有用。

8.4.3 制作测试替身

为了制作测试替身，通常使用外部库，就像使用单元测试一样。以下是一些主流的解决方案：

- GoogleMock(也被称为 gMock)，现在是 GoogleTest 库的一部分：<https://github.com/google/googletest>。
- Trompeloeil 专注于 C++14，很好地集成了许多测试库，如 Catch2、doctest 和 GTest：<https://github.com/rollbear/trompeloeil>。

下面的代码将向展示如何使用 GoogleMock 和 Trompeloeil。

GoogleMock

由于 GoogleMock 是 GoogleTest 的一部分，这里将一起使用：

```
1 #include "merchants/reviews.h"
2
3 #include <gmock/gmock.h>
4
5 #include <merchants/visited_merchant_history.h>
6
7 #include "fake_customer_review_store.h"
8
9 namespace {
10
11 class mock_visited_merchant : public i_visited_merchant {
12 public:
13     explicit mock_visited_merchant(fake_customer_review_store &store,
14                                     merchant_id_t id)
15     : review_store_{store},
16       review_{store.get_review_for_merchant(id).value()} {
17         ON_CALL(*this, post_rating).WillByDefault([this](stars s) {
18             review_.rating = s;
19             review_store_.post_review(review_);
20         });
21         ON_CALL(*this, get_rating).WillByDefault([this] { return
22             review_.rating; });
23     }
24 }
```

```

25 MOCK_METHOD(stars, get_rating, (), (override));
26 MOCK_METHOD(void, post_rating, (stars s), (override));
27
28 private:
29     fake_customer_review_store &review_store_;
30     review review_;
31 };
32
33 } // namespace
34
35 class history_with_one_rated_merchant : public ::testing::Test {
36 public:
37     static constexpr std::size_t CUSTOMER_ID = 7777;
38     static constexpr std::size_t MERCHANT_ID = 1234;
39     static constexpr const char *REVIEW_TEXT = "Very nice!";
40     static constexpr stars RATING = stars{5.f};
41
42 protected:
43     void SetUp() final {
44         fake_review_store_.post_review(
45             {CUSTOMER_ID, MERCHANT_ID, REVIEW_TEXT, RATING});
46
47         // nice mock will not warn on "uninteresting" call to get_rating
48         auto mocked_merchant =
49             std::make_unique<::testing::NiceMock<mock_visited_merchant>>(
50                 fake_review_store_, MERCHANT_ID);
51
52         merchant_index_ = history_.add(std::move(mocked_merchant));
53     }
54
55     fake_customer_review_store fake_review_store_{CUSTOMER_ID};
56     history_of_visited_merchants history_{};
57     std::size_t merchant_index_{};
58 };
59
60 TEST_F(history_with_one_rated_merchant,
61     when_user_changes_rating_then_the_review_is_updated_in_store) {
62     const auto &mocked_merchant = dynamic_cast<const mock_visited_merchant
63     &>(
64         history_.get_merchant(merchant_index_));
65     EXPECT_CALL(mocked_merchant, post_rating);
66
67     constexpr auto new_rating = stars{4};
68     static_assert(RATING != new_rating);
69     history_.rate(merchant_index_, stars{new_rating});
70 }
71
72 TEST_F(history_with_one_rated_merchant,
73     when_user_selects_same_rating_then_the_review_is_not_updated_in_store) {

```

```

74     const auto &mocked_merchant = dynamic_cast<const mock_visited_merchant
75     &>(
76         history_.get_merchant(merchant_index_));
77     EXPECT_CALL(mocked_merchant, post_rating()).Times(0);
78
79     history_.rate(merchant_index_, stars{RATING});
80 }
```

GTest 是本书编写时最流行的 C++ 测试框架，与 GMock 的集成，意味着 GMock 在项目中可能已经可用。这种组合使用起来很直观，而且功能齐全，所以如果已经使用了 GTest，就没有理由选择其他框架。

Trompeloeil

为了将这个示例与前一个示例进行对比，这次使用 Trompeloeil 进行测试，并使用 Catch2 作为测试框架：

```

1 #include "merchants/reviews.h"
2
3 #include "fake_customer_review_store.h"
4
5 // order is important
6 #define CATCH_CONFIG_MAIN
7 #include <catch2/catch.hpp>
8 #include <catch2/trompeloeil.hpp>
9
10 #include <memory>
11
12 #include <merchants/visited_merchant_history.h>
13
14 using trompeloeil::_;
15
16 class mock_visited_merchant : public i_visited_merchant {
17 public:
18     MAKE MOCK0(get_rating, stars(), override);
19     MAKE MOCK1(post_rating, void(stars s), override);
20 };
21
22 SCENARIO("merchant history keeps store up to date", "[mobile app]") {
23     GIVEN("a history with one rated merchant") {
24         static constexpr std::size_t CUSTOMER_ID = 7777;
25         static constexpr std::size_t MERCHANT_ID = 1234;
26         static constexpr const char *REVIEW_TEXT = "Very nice!";
27         static constexpr stars RATING = stars{5.f};
28
29         auto fake_review_store_ = fake_customer_review_store{CUSTOMER_ID};
30         fake_review_store_.post_review(
31             {CUSTOMER_ID, MERCHANT_ID, REVIEW_TEXT, RATING});
32
33         auto history_ = history_of_visited_merchants{};
34     }
35
36     WHEN("a merchant is rated")
37     {
38         AND("the merchant has not been rated before")
39         THEN("the merchant's rating is updated")
40     }
41
42     WHEN("a merchant is rated again")
43     {
44         AND("the merchant has already been rated")
45         THEN("the merchant's rating is updated again")
46     }
47 }
```

```

34     const auto merchant_index_ =
35         history_.add(std::make_unique<mock_visited_merchant>());
36
37     auto &mocked_merchant = const_cast<mock_visited_merchant &>(
38         dynamic_cast<const mock_visited_merchant &>(
39             history_.get_merchant(merchant_index_)));
40
41     auto review_ = review{CUSTOMER_ID, MERCHANT_ID, REVIEW_TEXT, RATING};
42     ALLOW_CALL(mocked_merchant, post_rating(_))
43         .LR_SIDE_EFFECT(review_.rating = _1;
44         fake_review_store_.post_review(review_));
45
46     ALLOW_CALL(mocked_merchant, get_rating()).LR_RETURN(review_.rating);
47
48     WHEN("a user changes rating") {
49         constexpr auto new_rating = stars{4};
50         static_assert(RATING != new_rating);
51
52         THEN("the review is updated in store") {
53             REQUIRE_CALL(mocked_merchant, post_rating(_));
54             history_.rate(merchant_index_, stars{new_rating});
55         }
56     }
57
58     WHEN("a user selects same rating") {
59         THEN("the review is not updated in store") {
60             FORBID_CALL(mocked_merchant, post_rating(_));
61             history_.rate(merchant_index_, stars{RATING});
62         }
63     }
64 }
65 }
```

Catch2 的一个重要特性是，它可以很容易地编写行为驱动开发风格的测试，比如上面这个测试。如果比较喜欢这种风格，那么带有 Trompeloeil 的 Catch2 将是一个很好的选择。

8.5. 测试驱动的类型设计

区分不同类型的测试和了解特定的测试框架（或几种）是不够的。开始测试时，就会注意到，并不是所有的类都可以轻松地测试。有时，可能会觉得需要访问私有属性或方法。如果想保持良好的架构原则，请抵制这种冲动！相反，可以考虑测试通过类型的公共 API 可用的业务需求，或者重构类型，以便有另一个代码单元可以测试。

8.5.1 测试和类设计——冲突

可能面临的问题不是测试框架不充分，而是遇到的是设计不当的类。即使类的行为和外观可能是正确的，除非允许测试，否则其设计不正确。

然而，这是个好消息。说明可以在不方便这样做之前修复问题。类设计可能会在稍后，基于它构建类层次结构时再次困扰开发者。在测试执行过程中修改设计可以减少可能的技术债。

8.5.3 防御性编程

防御性编程并不是一种安全特性，其名字来自于保护类和函数的使用不违背它们的初衷。它与测试没有直接关系，但它是一种很好的设计模式，因为它提高了代码的质量，使项目具有前瞻性。

防御性编程从静态类型开始。若创建一个函数来处理自定义类型作为参数，必须确保没有人会用一些意外的值来使用它。用户必须有意识地检查函数的期望并准备相应的输入。

C++中可以在编写模板代码时利用类型安全特性。当为客户的评论创建一个容器时，可以接受任何类型的列表，并从中复制。为了得到更详细的错误信息和精心设计的检查，可以这样做：

```
1 class CustomerReviewStore : public i_customer_review_store {
2 public:
3     CustomerReviewStore() = default;
4     explicit CustomerReviewStore(const std::ranges::range auto
5         &initial_reviews) {
6         static_assert(is_range_of_reviews_v<decltype(initial_reviews)>,
7             "Must pass in a collection of reviews");
8         std::ranges::copy(begin(initial_reviews), end(initial_reviews),
9             begin(reviews_));
10    }
11 // ...
12 private:
13     std::vector<review> reviews_;
14 };
```

`explicit` 关键字保护不需要隐式类型转换。通过指定输入参数满足范围概念，可以确保只使用有效的容器进行编译。由于使用了概念，可以让防御代码输出更清晰的错误消息。在代码中使用 `static_assert` 也是一个很好的防御措施，因为它允许在需要的时候提供错误消息。`is_range_of_reviews` 的检查可以这样实现：

```
1 template <typename T>
2 constexpr bool is_range_of_reviews_v =
3     std::is_same_v<std::ranges::range_value_t<T>, review>;
```

通过这种方式，可以确保获得的范围包含合适类型的“评论”。

静态类型不会在运行时阻止无效的值传递给函数，这就是为什么防御型编程的下一种形式是检查的前提条件。这样，若有出现问题的迹象，代码就会失败，这比返回一个传播到系统其他部分的无效值要好得多。C++中有契约之前，可以使用前面章节提到的 GSL 库来检查代码的前置和后置条件：

```
1 void post_review(review review) final {
2     Requires(review.merchant);
3     Requires(review.customer);
4     Ensures(!reviews_.empty());
5
6     reviews_.push_back(std::move(review));
```

这里，通过使用 `Expect`s 宏，检查传入的评论是否确实具有商家和评论人的 ID 集。除了不会发生的情况外，还可以防止在使用“确保后置条件宏”时，向存储添加复查失败的情况。

当谈到运行时检查时，首先想到的是检查一个或多个属性是否不是 `nullptr`。避免这个问题的最好方法是区分可空资源（那些可以接受空指针的资源）和非空资源。C++17 的标准库中，`std::optional` 是一个很棒的工具。如果可以，请在设计 API 时使用它。

8.5.3 无聊的重复——先写测试

这句话已经重复过很多次了，但是很多人会“忘记”。当实际编写测试时，必须做的第一件事是减少创建难以测试的类的风险。从 API 使用开始，需要调整实现以最佳地服务于 API。这样，就会得到更易于使用和测试的 API。当在实现测试驱动开发 (TDD) 或在编写代码之前编写测试时，还可以将最终实现的依赖进行注入，这意味着可以创建更松散地耦合类。

用另一种方法（先编写类，然后再向类添加单元测试）会让编写的代码更容易，但更难以测试。当测试变得困难时，就会有跳过它的冲动。

8.6. 在持续集成/持续部署上进行自动化测试

下一章中，将关注持续集成和持续部署 (CI/CD)。要使 CI/CD 流水正常工作，需要进行一组测试，以便在 bug 进入生产环境之前捕获它们。开发者和其团队需要确保所有业务需求都正确地在测试进行表达。

测试在几个层次上都很有用。对于行为驱动开发，业务需求构成了自动化测试的基础。但在构建的系统并不仅仅由业务需求组成，希望能够确保所有第三方集成都按预期工作，并且希望所有子组件（如微服务）可以相互接口。最后，希望正在构建的函数和类没有任何错误。

可以自动化的每个测试都是 CI/CD 流水的候选，它们中的每一个都在这个流水的某个地方有自己的位置。例如，端到端测试在作为验收测试部署之后最有意义。另一方面，单元测试在编译后直接执行时最有意义。毕竟，流水的目标是为了发现可能与规格不符的地方时，就立即中断。

每次运行 CI/CD 流水时，不必运行已经自动化的所有测试。如果每个流水的运行时间相对较短就更好了。理想情况下，应该在提交后几分钟内完成。如果想保持最小的运行时，如何确保一切都得到了适当的测试？

一种方法是为不同的目的准备不同的测试套件。例如，提交到特性分支的测试可以很少。由于每天都有很多人提交特性分支，这意味着它们只会简单地测试，而且结果会很快得到。将特性分支合并到共享开发分支需要大一些的测试用例集。这样，就可以确保没有破坏其他团队成员的任何东西。最后，将运行一组更广泛的案例，用于合并到生产分支。毕竟，即使测试需要相当长的时间，也希望对生产分支进行彻底的测试。

另一种是为 CI/CD 目的使用精简的测试用例集，并有一个额外的连续测试过程。此过程定期运行，并对特定环境的当前状态执行深入检查。这些测试可以包括安全性测试和性能测试，因此可以评估待提升环境的合格性。

当选择一个环境，并承认这个环境具备成为一个更成熟环境的所有特质时，就会出现提升，例如开发环境可以成为下一个阶段环境，或者阶段环境可以成为下一个生产环境。如果这种升级是自

动发生的，那么提供自动回滚也是一个好做法，以防细微的差异（比如，域名或流量方面的差异）使新升级的环境不再通过测试。

这也提出了另一个重要的实践：始终在生产环境上运行测试。当然，这样的测试必须最小化干扰，但是它们应该告知，系统在给定的时间内都在正确地执行。

8.6.1 测试的基础设施

如果想将配置管理、基础设施作为代码或不可变部署的概念合并到应用程序的软件架构中，还应该考虑测试基础设施本身。可以使用工具来实现这一点，包括 Serverspec、Testinfra、Goss 和 Terratest，这些都是比较主流的工具。

这些工具的适用范围略有不同，如下所述：

- Serverspec 和 Testinfra 更专注于测试通过配置管理（如 Salt、Ansible、Puppet 和 Chef）配置的服务器的实际状态，分别用 Ruby 和 Python 编写，插入到语言的测试引擎中。这意味着用于 Serverspec 的 RSpec 和用于 Testinfra 的 Pytest。
- Goss 在范围和形式上都有点不同。除了测试服务器之外，还可以使用 Goss 来使用 dgoss 包装器测试项目中使用的容器。至于形式，没有使用在 Serverspec 或 Testinfra 中看到的命令式代码。相反，与 Ansible 或 Salt 类似，使用一个 YAML 文件来描述想要检查的状态。如果已经在使用声明式方法进行配置管理（例如，前面提到的 Ansible 或 Salt），那么 Goss 可能更直观，因此更适合测试。
- 最后，Terratest 是一个工具，允许测试基础设施的输出作为代码工具，如 Packer 和 Terraform（因此得名）。就像 Serverspec 和 Testinfra 使用语言测试引擎为服务器编写测试一样，Terratest 利用 Go 的测试包来编写适当的测试用例。

接下来，来了解一下如何使用这些工具来验证部署是否按照计划进行（至少从基础结构的角度来看）。

8.6.2 Serverspec 的测试

下面是一个测试 Serverspec 的例子，检查 Git 在特定版本中的可用性和加密配置文件：

```
# We want to have git 1:2.1.4 installed if we're running Debian
describe package('git'), :if => os[:family] == 'debian' do
  it { should be_installed.with_version('1:2.1.4') }
end

# We want the file /etc/letsencrypt/config/example.com.conf to:
describe file('/etc/letsencrypt/config/example.com.conf') do
  it { should be_file } # be a regular file
```

```

it { should be_owned_by 'letsencrypt' } # owned by the letsencrypt user
it { should be_mode 600 } # access mode 0600
it { should contain('example.com') } # contain the text example.com
                                # in the content
end

```

Ruby DSL 语法应该是可读的，即使对那些不日常使用 Ruby 的人也是如此。

8.6.3 Testinfra 的测试

下面是 Testinfra 的一个测试示例，检查 Git 在特定版本中的可用性和加密配置文件：

```

# We want Git installed on our host
def test_git_is_installed(host):
    git = host.package("git")
    # we test if the package is installed
    assert git.is_installed
    # and if it matches version 1:2.1.4 (using Debian versioning)
    assert git.version.startswith("1:2.1.4")

# We want the file /etc/letsencrypt/config/example.com.conf to:
def test_letsencrypt_file(host):
    le = host.file("/etc/letsencrypt/config/example.com.conf")
    assert le.user == "letsencrypt" # be owned by the letsencrypt user
    assert le.mode == 0o600 # access mode 0600
    assert le.contains("example.com") # contain the text example.com in the contents

```

Testinfra 使用纯 Python 语法。可读性不错，但就像 Serverspec 一样，可能需要一段时间的了解，才能自信地在其中编写测试。

8.6.4 Goss 的测试

下面是 Goss 的 YAML 文件的一个例子，检查 Git 在特定版本的可用性和加密配置文件：

```

# We want Git installed on our host
package:
  git:
    installed: true # we test if the package is installed
    versions:
      - 1:2.1.4 # and if it matches version 1:2.1.4 (using Debian versioning)

```

```

file:
  # We want the file /etc/letsencrypt/config/example.com.conf to:
  /etc/letsencrypt/config/example.com.conf:
    exists: true
    filetype: file # be a regular file
    owner: letsencrypt # be owned by the letsencrypt user
    mode: "0600" # access mode 0600
    contains:
      - "example.com" # contain the text example.com in the contents

```

读和写 YAML 的语法可能需要最少的准备。但是，如果项目已经使用了 Ruby 或 Python，那么在编写更复杂的测试时，可以坚持使用 Serverspec 或 Testinfra。

8.7. 总结

本章集中在测试软件不同部分的体系结构和技术方面。我们了解了测试金字塔，从而了解不同类型的测试，如何对软件项目的整体健康和稳定做出贡献。由于测试可以同时是功能性和非功能性的，我们看到了这两种类型的一些示例。

从本章中最重要的是，测试不是结束。我们需要它们，不是因为它们能带来即时的价值，而是当重构时，或者当我们改变系统现有部分的行为时，可以使用它们来检查已知的回归。当想要进行原因分析时，测试也可以快速验证不同的假设。

建立了理论需求之后，展示了可以用来编写测试替身效果的不同测试框架和库的示例。先编写测试，然后再实现它们需要一些实践，但这种方式是有好处的。这个好处是可以更好对类进行设计。

最后，为了强调现代架构不仅仅是软件代码，还研究了一些用于测试基础设施和部署的工具。下一章中，将看到持续集成和持续部署如何为要架构的应用程序，带来更好的服务质量、健壮性。

8.8. 练习题

1. 测试金字塔的基础层是什么？
2. 有哪些类型的非功能测试？
3. 著名的原因分析方法叫什么？
4. 是否有可能在 C++ 中测试编译时代码？
5. 当为具有外部依赖的代码编写单元测试时，应该使用什么进行测试？
6. 单元测试在持续集成/持续部署中扮演什么角色？
7. 哪些工具可以使用测试基础架构代码？
8. 单元测试中访问类的私有属性和方法是一个好主意吗？

8.9. 扩展阅读

Testing C++ Code: <https://www.packtpub.com/application-development/modern-cprogramming-cookbook>

Test Doubles: <https://martinfowler.com/articles/mocksArentStubs.html>

Continuous Integration/Continuous Deployment: <https://www.packtpub.com/virtualization-and-cloud/hands-continuous-integration-and-delivery> and <https://www.packtpub.com/virtualization-and-cloud/cloud-native-continuous-integrationand-delivery>

第 9 章 持续集成和持续部署

前面关于构建和打包的章节中，了解了应用程序可以使用不同构建系统和不同打包系统。持续集成 (CI) 和持续部署 (CD) 允许使用构建和打包，来改进服务质量的应用程序的健壮性。

CI 和 CD 都依赖于良好的测试覆盖率。CI 主要使用单元测试和集成测试，而 CD 更依赖于冒烟测试和端到端测试。在第 8 章中了解了更多关于测试的不同方面。有了这些知识，就可以构建 CI/CD 流水线了。

本章将讨论以下内容：

- 了解 CI
- 检查代码更改
- 探索测试驱动的自动化
- 管理部署代码
- 构建部署代码
- 建立 CD 流水线
- 不可变的基础设施

9.1. 相关准备

本章的代码可以在以下 GitHub 页面找到：<https://github.com/PacktPublishing/Software-Architecture-with-Cpp/tree/master/Chapter09>。

为了理解本章中解释的概念，需要安装以下软件：

- 一个免费的 GitLab 账号
- Ansible 2.8 以上版本
- Terraform 0.12 以上版本
- Packer 1.4 以上版本

9.2. 了解 CI

CI 是可以缩短集成周期的过程。传统软件中，许多不同的特性可以单独开发，并且只在发布之前集成，而在使用 CI 开发的项目中，集成可能一天发生几次。通常，开发人员所做的每个更改在提交到中央存储库的同时都要进行测试和集成。

因为测试发生在开发之后，所以反馈循环会更快，这让开发人员更容易地修复 bug(通常记得修改了什么)。与传统的在发布之前进行测试的方法相比，CI 节省了大量的工作，并提高了软件的质量。

9.2.1 尽早发布，经常发布

听说过“早发布，多发布”吗？这是一种强调短发布周期重要性的软件开发哲学。反过来，短的发布周期在计划、开发和验证之间提供了更短的反馈循环。当某物损坏时，应该尽早发现，这样修复问题的成本就相对较小。

这是由 Eric S. Raymond(也被称为 ESR)在他 1997 年的文章《大教堂和集市》中提出的。还有一本同名的书包含了作者的这篇文章和其他文章。考虑到 ESR 在开源运动中的活动，“尽早发布，经常发布”的口号成为了开源项目如何运作的代名词。

几年后，同样的原则已经超越了开源项目。随着人们对敏捷方法(如 Scrum)的兴趣日益浓厚，“尽早发布，经常发布”的口头语变成了以产品增量结束的开发冲刺的同义词。当然，这个增量是一个软件版本，但通常在冲刺期间还会有许多其他版本。

如何实现如此短的发布周期？答案是尽可能地自动化。理想情况下，每次提交到代码存储库都应该以发布作为结束，这个版本最终是否面向客户则是另一回事。重要的是，每一次代码更改都可以产生一个可用的产品。

当然，构建并对外发布每一个提交对开发人员来说都是一项乏味的工作。即使一切都是脚本化的，这也会给日常工作增加不必要的开销。这就是为什么要建立一个 CI 系统进行自动化发布。

9.2.2 CI 的优点

CI 会集成多个开发人员的工作，至少每天是这样，有时它意味着一天几次。每个进入存储库的提交都是集成的，并分别验证。生成系统检查是否可以在不出错的情况下生成代码。打包系统可以创建一个包，这个包可以作为工件保存，甚至可以在 CD 时部署。最后，可以用自动化测试检查与变更相关的已知回归是否发生。下面来详细了解一下它的优点：

- CI 可以快速解决问题。如果某个开发人员忘记了行尾的分号，那么 CI 系统上的编译器将在其他开发人员收到错误代码之前立即捕获该错误，从而阻塞他们的工作。当然，开发人员应该总是构建更改并在提交代码之前进行测试，但在开发人员的机器上，轻微的键入错误可能会忽略，并进入共享的代码库。
- 使用 CI 的另一个好处是可以避免常见的“在我的机器上工作”的借口。如果开发人员忘记提交必要的文件，CI 系统将无法构建更改，从而再次防止它们进一步传播并对整个团队造成损害。开发人员环境的特殊配置也不再是问题。如果一个更改构建在两台机器上，开发人员的计算机和 CI 系统，那么可以放心地假设也可以在其他机器上构建。

9.2.3 阀门机制

如果想让 CI 带来价值，而不仅仅是简单地构建包，则需要一个限制机制。这种门控机制可以区分好和坏的代码更改，从而使应用程序免受可能使其无用的修改的影响。为此，需要一套全面的测试。这样的套件允许在变更有问题时自动识别，并且能够快速地进行识别。

对于单个组件，单元测试扮演着一个门控机制的角色。CI 系统可以丢弃任何没有通过单元测试的更改，或者没有达到特定代码覆盖率阈值的更改。在构建单个组件时，CI 系统还可以使用集成测试来进一步确保更改是稳定的，不仅是更改本身，而且是在一起时可以正确地运行。

9.2.4 用 GitLab 实现流水

本章中，将使用流行的开源工具来构建一个完整的 CI/CD 流水，包括门控机制、自动化部署，并展示基础设施自动化的概念。

第一个这样的工具是 GitLab。可能听说过它是一个 Git 托管解决方案，但实际上，它的作用远不止于此。GitLab 有以下几种发行版：

- 开源的解决方案，可以自行托管
- 自托管的付费版本提供了比开源社区版更多的特性
- 最后是软件即服务 (SaaS) 管理的产品 <https://gitlab.com>

对于本书的要求，每个发行版都有所有必要的特性。因此，将重点关注 SaaS 版本，这不需要太多的准备工作。

<https://gitlab.com> 主要针对开源项目，如果不想与全世界分享自己的工作，也可以创建私人项目和存储库。可以在 GitLab 中创建一个新的私有项目，并使用在第 7 章中演示过的代码填充它。

很多现代的 CI/CD 工具都可以代替 GitLab 的 CI/CD。包括 GitHub Actions, Travis CI, CircleCI 和 Jenkins。这里选择了 GitLab，因为既可以以 SaaS 的形式使用，也可以在本地使用，因此应该可以适应不同的用例。

然后，使用之前的构建系统，在 GitLab 中创建一个简单的 CI 流水。这些流水在 YAML 文件中描述为一系列步骤和元数据。构建所有需求的流水示例，以及第 7 章中的示例项目，如下所示：

```
# We want to cache the conan data and CMake build directory
cache:
  key: all
  paths:
    - .conan
    - build

# We're using conanio/gcc10 as the base image for all the subsequent commands
default:
  image: conanio/gcc10

stages:
  - prerequisites
  - build

before_script:
  - export CONAN_USER_HOME="$CI_PROJECT_DIR"

# Configure conan
prerequisites:
  stage: prerequisites
```

```

script:
- pip install conan==1.34.1
- conan profile new default || true
- conan profile update settings.compiler=gcc default
- conan profile update settings.compiler.libcxx=libstdc++11 default
- conan profile update settings.compiler.version=10 default
- conan profile update settings.arch=x86_64 default
- conan profile update settings.build_type=Release default
- conan profile update settings.os=Linux default
- conan remote add trompeloeil
https://api.bintray.com/conan/trompeloeil/trompeloeil || true

# Build the project
build:
stage: build
script:
- sudo apt-get update && sudo apt-get install -y docker.io
- mkdir -p build
- cd build
- conan install ../ch08 --build=missing
- cmake -DBUILD_TESTING=1 -DCMAKE_BUILD_TYPE=Release ../ch08/customer
- cmake --build .

```

将前面的文件保存为`.gitlab-ci.yml`。Git 库根目录下的会自动在 GitLab 中启用 CI，并在每次提交时运行流水。

9.3. 检查代码更改

代码审查既可以用于 CI 系统，也可以不用于。其主要目的是再次检查代码中引入的每一个更改，以确保它是正确的，符合应用程序的体系结构，并遵循项目的指导方针和最佳实践。

在没有 CI 系统的情况下使用时，通常是评审人员的任务来手动测试更改，并验证是否按预期工作。CI 减少了这种负担，使软件开发人员可以专注于代码的逻辑结构。

9.3.1 自动控制机制

自动化测试只是门控机制的一个例子。当质量足够高时，可以保证代码按照设计工作。但是，正确运行的代码和良好的代码之间仍存在差异。目前为止，如果代码满足几个条件，就可以认为是好的，功能正确只是其中之一。

其他工具可以实现所需的代码库标准。其中一些已经在前面的章节中介绍过，所以这里就不详

细讨论了。记住，在 CI/CD 流水中使用检查器、代码格式化器和静态分析是一个很好的实践。虽然静态分析可以作为一种门控机制，但可以对进入中央存储库的每个提交应用检测和格式化，使其与其他代码库保持一致。可以在附录中找到更多关于静态代码分析器和格式化工具的信息。

理想情况下，这种机制只需要检查代码是否已经格式化，因为格式化步骤应该由开发人员在将代码放入存储库之前完成。当使用 Git 作为版本控制系统时，Git hook 的机制可以防止在没有运行必要工具的情况下提交代码。

但自动化分析只能帮到这里。接下来，需要检查代码在功能上是否完整，是否没有已知的错误和漏洞，是否符合编码标准，这就需要人工检查了。

9.3.2 代码审查——手动控制机制

代码修改的手动检查通常称为代码审查，其目的是识别问题，包括特定子系统的实现和应用程序的整体架构。自动化性能测试可能会发现，也可能不会发现给定功能的潜在问题。另一方面，人眼通常可以发现问题的次优方案。无论是错误的数据结构，还是计算复杂度过高的算法，优秀的架构师都应该能够查明问题所在。

但执行代码检查的不仅仅是架构师的角色。同行评审，即由作者的同行执行的代码评审，在开发过程中也有自己的位置。这样的审查之所以有价值，不仅仅是因为允许同事们在彼此的代码中发现错误。更重要的方面是，许多同事也可以了解到其他人在做什么。这样，当团队中出现缺勤时（无论是因为长时间的会议、假期，还是工作轮换），另一个团队成员可以代替缺勤的那个人。即使他们不是这个方向的专家，每个其他成员至少知道相关的代码位于哪里，每个人都应该能够记住对代码的最后更改。这既指它们发生的时间，也指这些变化的范围和内容。

随着越来越多的人意识到应用程序内部是如何显示的，也更有可能找出一个组件中最近的更改与新发现的 bug 之间的关联。尽管团队中的每个人可能都有不同的经验，但当每个人都非常了解代码时，就可以共享资源。

因此，代码审查可以检查更改是否符合所需的体系结构，以及实现是否正确。可以将这样的代码评审称为架构评审，或者专家评审。

另一种类型的代码审查，即同行审查，不仅有助于发现错误，而且还提高了团队中其他成员正在做什么的意识。如果有必要，在处理与外部服务集成的更改时，还可以执行不同类型的专家审查。

由于每个接口都可能成为问题来源，接近接口级别的更改应该视为特别危险。这里建议在通常的同行评议中加入来自另一方的专家。例如，如果正在编写生产者的代码，请使用者进行审查。通过这种方式，可以确保不会错过一些重要的用例，这些用例在某个方面可能认为不太可能出现，但另一方面可能经常使用。

9.3.3 代码评审的不同方法

代码审查通常会异步进行，这意味着审查变更的作者和审查者之间的交流不是实时发生的。相反，每个参与者随时都可以发布他们的评论和建议。当没有更多的评论，作者需要重新修改，并再次将修改置于审查中。这个过程可能需要很多轮，直到每个人对这次修正没有意见。

当一个修改特别有争议，并且异步代码审查花费了太多时间时，同步地进行代码审查是有益的。这意味着一场会议（面对面或远程），以解决前进道路上的各种意见。当由于在实施变更时获得

的新知识而使变更与最初的决定相矛盾时，这种情况就会发生。

有一些专门的工具只针对代码检查。更常见的是，希望使用内置在存储库服务器中的工具，包括如下服务：

- GitHub
- Bitbucket
- GitLab
- Gerrit

上述所有服务都提供 Git 托管和代码审查。有些甚至更进一步，提供完整的 CI/CD 管道、问题管理、wiki 等等。

当使用代码托管和代码审查的组合包时，默认的工作流程是将更改作为一个单独的分支推送，然后要求项目所有者在一个称为拉请求（或合并请求）的过程中合并更改。尽管名称很花哨，但 pull 请求或 merge 请求会告诉项目所有者你有代码想要与主分支合并。这意味着审查人员应该检查更改，以确保一切正常。

9.3.4 使用拉请求（合并请求）进行代码评审

像 GitLab 这样的系统创建拉请求或合并请求非常容易。首先，从命令行将一个新分支推送到中央存储库时，可以观察到以下消息：

```
remote:  
remote: To create a merge request for fix-ci-cd, visit:  
remote:  
https://gitlab.com/hosacpp/continuous-integration/merge\_requests/new?merge\_request%5Bsource\_branch%5D=fix-ci-cd  
remote:
```

如果以前启用了 CI（通过添加 `.gitlab-ci.yml` 文件），还会看到新推送的分支已经置于 CI 进程中。这甚至发生在创建合并请求之前，从而这意味着可以推迟发送给同事，直到从 CI 获得每个自动检查都已通过的信息。

打开合并请求的两种主要方式如下：

- 通过链接中提到的推送信息
- 通过在 GitLab UI 中导航到合并请求，并选择创建合并请求按钮或新建合并请求按钮

当提交合并请求时，并完成了所有相关字段，将看到 CI 流水的状态也是可见的。若流水失败，就不可能进行合并。

9.4. 探索测试驱动的自动化

CI 主要关注集成部分，构建不同子系统的代码并确保它们协同工作。虽然测试并不是实现此目的的严格要求，但运行 CI 而不使用似乎是一种浪费。没有自动化测试的 CI 更容易在代码中引入

bug，同时给人一种错误的安全感。

这就是 CI 经常与持续测试密切相关的原因，将在下一节中讨论这个问题。

9.4.1 行为驱动开发

目前为止，已经成功地建立了一个流水，可以称之为持续构建。对代码所做的每一个更改最终都会编译，但不再进一步测试。现在是时候介绍持续测试的实践了。在一个低级别上的测试，也将作为一个控制机制来自动拒绝所有不满足需求的修改。

如何检查给定的更改是否满足需求？最好通过基于这些需求编写测试来实现。做到这一点的方法是遵循行为驱动开发 (BDD)。BDD 的概念是鼓励敏捷项目中不同参与者之间进行更深层次的合作。

与传统的由开发人员或 QA 团队编写测试的方法不同。使用 BDD，测试是由以下人员协作创建的：

- 开发人员
- QA 工程师
- 业务代表

为 BDD 指定测试的最常见方法是使用 Cucumber 框架，该框架使用简单的英语短语来描述系统的期望行为。这些句子遵循特定的模式，然后可以转换为工作代码，与所选的测试框架集成。

Cucumber 框架中有对 C++ 的官方支持，基于 CMake、Boost、GTest 和 GMock。在以 cucumber 格式（它使用一种称为 Gherkin 的领域特定语言）指定所需的行为之后，还需要提供所谓的步骤定义（步骤定义是与 cucumber 规范中描述的操作相对应的实际代码）。例如，考虑在 Gherkin 中的以下行为：

```
# language: en
Feature: Summing
In order to see how much we earn,
Sum must be able to add two numbers together

Scenario: Regular numbers
Given I have entered 3 and 2 as parameters
When I add them
Then the result should be 5
```

可以把它保存为 `sum.feature` 文件。为了通过测试生成有效的 C++ 代码，这里将使用适当的步骤对其进行定义：

```
1 #include <gtest/gtest.h>
2 #include <cucumber-cpp/autodetect.hpp>
3
4 #include <Sum.h>
5
```

```

6 using cucumber::ScenarioScope;
7
8 struct SumCtx {
9     Sum sum;
10    int a;
11    int b;
12    int result;
13};
14
15 GIVEN("^I have entered (\d+) and (\d+) as parameters$", (const int a,
16 const int b)) {
17     ScenarioScope<SumCtx> context;
18     context->a = a;
19     context->b = b;
20 }
21
22 WHEN("^I add them") {
23     ScenarioScope<SumCtx> context;
24     context->result = context->sum.sum(context->a, context->b);
25 }
26
27 THEN("^the result should be (.*)$", (const int expected)) {
28     ScenarioScope<SumCtx> context;
29     EXPECT_EQ(expected, context->result);
30 }

```

从零开始构建应用程序时，最好遵循 BDD 模式。本书旨在展示在这样一个新项目中使用的最佳实践，但这并不能在现有的项目中尝试这里的示例。CI 和 CD 可以在项目生命周期的任何给定时间添加。因为频繁地运行测试总是一个好主意，所以仅为了持续测试的目的而使用 CI 系统总是第一个好主意。

如果没有行为测试，也不需要担心。可以稍后再添加它们，目前只关注已经拥有的测试。无论是单元测试还是端到端测试，能够帮助评估应用程序状态的内容都可以作为门控机制的候选。

9.4.2 为 CI 编写测试

对于 CI，最好关注单元测试和集成测试。他们在尽可能低的层级上工作，这意味着通常能够快速执行，并且拥有最小的需求。理想情况下，所有单元测试都应该是自包含的（没有像工作数据库那样的外部依赖），并且能够并行运行。这样，当问题出现在单元测试能够捕捉到的级别时，错误代码将在几秒钟内标记出来。

有些人说，单元测试只在解释语言或具有动态类型的语言中有意义。其论点是，C++已经通过类型系统和编译器检查错误代码的方式内置了测试。虽然类型检查确实可以捕获一些需要在动态类型语言中进行单独测试的 bug，但这不能成为不编写单元测试的借口。毕竟，单元测试的目的不是验证代码是否可以毫无问题地执行。编写单元测试以确保代码不仅能够执行，而且能够满足所有的业务需求。

作为一个极端的例子，看一下下面两个函数。它们在语法上都是正确的，并且使用了正确的输入。然而，仅仅通过观察，可能就能猜出哪个是正确的，哪个是错误的。单元测试有助于捕捉这种

错误行为:

```
1 int sum (int a, int b) {  
2     return a+b;  
3 }
```

前面的函数返回提供的两个参数的和。下面的代码只返回第一个参数的值:

```
1 int sum (int a, int b) {  
2     return a;  
3 }
```

即使类型匹配，编译器也不会报错，这段代码也不会执行。为了区分有用的代码和错误的代码，这里需要使用测试和断言。

9.4.3 持续测试

已经建立了一个简单的 CI 流水，很容易通过测试对其进行扩展。因为使用 CMake 和 CTest 来构建和测试过程，所以需要做的就是在流水中添加另一个步骤来执行测试:

```
# Run the unit tests with ctest  
  
test:  
    stage: test  
    script:  
        - cd build  
        - ctest .
```

因此，整个流水如下:

```
cache:  
key: all  
paths:  
    - .conan  
    - build  
  
default:  
image: conanio/gcc9  
  
stages:  
    - prerequisites  
    - build  
    - test # We add another stage that runs the tests
```

```

before_script:
  - export CONAN_USER_HOME="$CI_PROJECT_DIR"

prerequisites:
  stage: prerequisites
  script:
    - pip install conan==1.34.1
    - conan profile new default || true
    - conan profile update settings.compiler=gcc default
    - conan profile update settings.compiler.libcxx=libstdc++11 default
    - conan profile update settings.compiler.version=10 default
    - conan profile update settings.arch=x86_64 default
    - conan profile update settings.build_type=Release default
    - conan profile update settings.os=Linux default
    - conan remote add trompeloeil
    https://api.bintray.com/conan/trompeloeil/trompeloeil || true

build:
  stage: build
  script:
    - sudo apt-get update && sudo apt-get install -y docker.io
    - mkdir -p build
    - cd build
    - conan install ../ch08 --build=missing
    - cmake -DBUILD_TESTING=1 -DCMAKE_BUILD_TYPE=Release ../ch08/customer
    - cmake --build .

# Run the unit tests with ctest
test:
  stage: test
  script:
    - cd build
    - ctest .

```

通过这种方式，每个提交不仅要进行构建，还要经受测试。如果其中一个步骤失败，开发者都将收到通知，并且可以在指示板中看到哪些步骤是成功的，哪些是失败的。

9.5. 管理部署代码

修改经过测试和批准后，现在是时候将其部署到操作环境了。

有许多工具可以帮助进行部署。这里提供 Ansible 的例子，因为不需要在目标机器上进行任何设置，除了需要安装 Python(这是大多数 UNIX 系统已经拥有的)。为什么是 Ansible？它在配置管理领域非常受欢迎，并且得到了值得信赖的开源公司 (Red Hat) 的支持。

9.5.1 使用 Ansible

为什么不使用一些已经可用的东西，如 Bourne shell 脚本或 PowerShell？对于简单的部署，shell 脚本可能是更好的方法。但随着部署过程变得更加复杂，使用 shell 的条件语句处理每个可能的初始状态变得更加困难。

与传统的 shell 脚本不同，处理初始状态之间的差异实际上是 Ansible 擅长的，使用命令式形式（移动这个文件，编辑那个文件，运行一个特定的命令），Ansible playbooks 仅使用声明形式（确保文件在这个路径中可用，确保文件包含指定的行，确保程序正在运行，确保程序成功完成）。

这种声明式方法还有助于实现幂等性。幂等性是函数的一个特征，意味着多次应用该函数将得到与单个应用完全相同的结果。如果 Ansible playbook 的第一次运行对配置进行了一些更改，那么后续的每次运行都将以所需的状态启动。这将阻止 Ansible 执行任何更改。

换句话说，当调用 Ansible 时，它会首先评估配置中所有机器的当前状态：

- 如果其中任何一个需要更改，Ansible 只会运行所需的任务，以达到所需的状态。
- 如果没有必要修改某一特定内容，Ansible 是不会去碰它的。只有当期望状态和实际状态不同时，才会看到 Ansible 采取行动，将实际状态收敛到 playbook 内容所描述的期望状态。

9.5.2 Ansible 如何适应 CI/CD 流水

Ansible 的幂等性使它成为 CI/CD 流水中使用的一个目标。毕竟，多次使用相同的 Ansible 策略没有风险，即使两次运行之间没有变化。如果使用 Ansible 来编写部署代码，那么创建 CD 就是为了准备合适的验收测试（比如冒烟测试或端到端测试）。

声明式方法可能需要改变对部署的看法，除了运行 playbook，还可以使用 Ansible 在远程机器上执行一次性命令，但这里不会讨论这个用例，因为它对部署没有实质性帮助。

任何能用 shell 做的事情，都可以用 Ansible 的 shell 模块完成。在 playbook 中，可以编写任务，指定使用哪些模块以及它们各自的参数。前面提到的 shell 模块就是这样一个模块，它只是在远程机器上的 shell 中执行提供的参数。但是使 Ansible 不仅方便而且跨平台（至少在不同的 UNIX 发行版中是这样）的原因是它提供了一些模块来操作常见的概念，比如用户管理、包管理和类似的实例。

9.5.3 创建代码部署

除了标准库中提供的常规模块外，还有第三方组件允许代码重用。可以单独测试这些组件，这也使部署的代码更加健壮。这样的组件称为角色，包含一组任务，使机器适合承担特定的角色，如 Web 服务器、数据库或 Docker。虽然有些角色让机器准备提供特定服务，但其他角色可能更抽象，例如流行的 ansible-hardening 角色。这是由 OpenStack 团队创建的，过使用这个角色保护会让入侵机器变得更加困难。

当开始理解 Ansible 使用的语言时，所有的 playbook 都不再只是脚本。反过来，它们将成为部署过程的文档。可以通过运行 Ansible 逐个使用，或者可以在离线的机器上阅读描述的任务并手动执行所有操作。

团队中使用 Ansible 进行部署有一个风险，开始就必须确保团队中的每个人都能够使用它并修改相关的任务。DevOps 是整个团队必须遵循的实践，它不能只是部分地实现。当应用程序的代码发生重大更改时，需要在部署端进行适当的修改，负责应用程序更改的人员还应该在部署代码中提供修改。当然，这是测试可以验证的，因此门控机制可以拒绝不完整的修改。

Ansible 值得注意的方面是，可以同时运行 push 和 pull 模型：

- push 模型是在自己的机器或 CI 系统中运行 Ansible 的时候。Ansible 连接到目标计算机（例如，通过 SSH 连接），并在目标计算机上执行必要的步骤。
- pull 模型中，整个过程由目标机发起。Ansible 的组件 ansible-pull 直接运行在目标机器上，检查代码库以确定特定分支是否有任何更新。在刷新本地 playbook 后，Ansible 照常执行所有步骤。这一次，控制组件和实际执行都发生在同一台机器上。大多数时候，可以周期性地运行 ansiblepull，例如从一个 cron(定时执行) 作业中。

9.6. 构建部署代码

最简单的形式中，使用 Ansible 的部署可能包括复制一个二进制文件到目标机器上，然后运行这个二进制文件。可以通过下面的 Ansible 代码实现：

```
tasks:  
  # Each Ansible task is written as a YAML object  
  # This uses a copy module  
  - name: Copy the binaries to the target machine  
    copy:  
      src: our_application  
      dest: /opt/app/bin/our_application  
  # This tasks invokes the shell module. The text after the `shell:` key  
  # will run in a shell on target machine  
  - name: start our application in detached mode  
    shell: cd /opt/app/bin; nohup ./our_application </dev/null >/dev/null  
  2>&1 &
```

每个任务都以连字符开头。对于每个任务，需要指定它使用的模块（例如 copy 模块或 shell 模块），以及它的参数（如果适用）。任务也可以有一个 name 参数，这使得单独引用任务更加容易。

9.7. 建立 CD 流水线

已经到了可以使用本章所学的工具安全地构建 CD 流水的时候了，已经知道 CI 如何运作，以及它如何帮助拒绝不适合发布的修改。关于测试自动化的部分展示了拒绝过程中更加健壮的不同

方法。通过冒烟测试或端到端测试，可以超越 CI，检查整个部署的服务是否满足需求。使用部署代码，不仅可以自动化部署过程，还可以在测试开始失败时准备回滚。

9.7.1 持续部署和持续交付

缩写 CD 可以表示两种不同的意思，持续交付和持续部署的概念非常相似，但它们有一些细微的区别。本书中关注的是持续部署的概念。这是一种自动化流程，产生于开发者将修改推入中央存储库，并在所有测试都通过的情况下成功地将更改部署到生产环境。因此，可以说这是一个端到端的过程，因为开发人员的工作在没有人工干预的情况下一路传递给客户（当然，要遵循代码审查）。可能听说过与这种方法相关的术语 GitOps。由于所有操作都是自动化的，因此推送到 Git 中指定的分支会触发部署脚本。

持续交付不会走那么远。与 CD 一样，特点是能够发布最终产品并对其进行测试，但最终产品不会自动交付给客户。可以首先交付给 QA，也可以交付给企业内部使用。理想情况下，只要内部客户端接受，交付的工件就可以部署到生产环境中。

9.7.2 构建一个 CD 流水示例

再次以 GitLab CI 为例，将所有这些技能放在一起构建流水。在测试步骤之后，将增加两个步骤，一个创建包，另一个使用 Ansible 部署这个包。

所需要的包装步骤如下：

```
# Package the application and publish the artifact
package:
  stage: package
  # Use cpack for packaging
  script:
    - cd build
    - cpack .
  # Save the deb package artifact
artifacts:
  paths:
    - build/Customer*.deb
```

当添加包含工件定义的包步骤时，能够从仪表板下载它们。

这样，就可以在部署步骤中调用 Ansible 了：

```
# Deploy using Ansible
deploy:
  stage: deploy
  script:
    - cd build
    - ansible-playbook -i localhost, ansible.yml
```

最后的流水如下所示:

```
cache:
  key: all
  paths:
    - .conan
    - build

default:
  image: conanio/gcc9

stages:
  - prerequisites
  - build
  - test
  - package
  - deploy

before_script:
  - export CONAN_USER_HOME="$CI_PROJECT_DIR"

prerequisites:
  stage: prerequisites
  script:
    - pip install conan==1.34.1
    - conan profile new default || true
    - conan profile update settings.compiler=gcc default
    - conan profile update settings.compiler.libcxx=libstdc++11 default
```

```

- conan profile update settings.compiler.version=10 default
- conan profile update settings.arch=x86_64 default
- conan profile update settings.build_type=Release default
- conan profile update settings.os=Linux default
- conan remote add trompeloeil
https://api.bintray.com/conan/trompeloeil/trompeloeil || true

build:
  stage: build
  script:
    - sudo apt-get update && sudo apt-get install -y docker.io
    - mkdir -p build
    - cd build
    - conan install ../ch08 --build=missing
    - cmake -DBUILD_TESTING=1 -DCMAKE_BUILD_TYPE=Release ../ch08/customer
    - cmake --build .

test:
  stage: test
  script:
    - cd build
    - ctest .

# Package the application and publish the artifact
package:
  stage: package
  # Use cpack for packaging
  script:
    - cd build
    - cpack .

# Save the deb package artifact
artifacts:
  paths:
    - build/Customer*.deb

# Deploy using Ansible
deploy:
  stage: deploy
  script:
    - cd build
    - ansible-playbook -i localhost, ansible.yml

```

要查看整个示例，可以在本章的代码库中看到。

9.8. 不可变的基础设施

如果对自己的 CI/CD 流水有足够的信心，则可以进行更进一步的了解。可以部署系统的构件，而不是部署应用程序的构件。有什么区别呢？在接下来的章节中来了解这一点。

9.8.1 不可变的基础设施

之前，重点讨论了如何使应用程序的代码在目标基础设施上的可部署性。CI 系统创建软件包（如容器），然后由 CD 进程部署这些软件包。每次流水运行时，基础设施保持不变，但软件不同。

如果正在使用云计算，可以像对待任何其他工件一样对待基础设施。不需要部署容器，可以部署整个虚拟机（VM），例如 AWS EC2 实例。可以预先构建这样一个 VM 镜像，作为 CI 过程的另一个元素。这样，版本化的 VM 镜像以及部署所需的代码将成为工件，而不是容器本身。

有两个工具（都是由 HashiCorp 编写的）可以精确地处理这种场景。Packer 帮助以一种可重复的方式创建 VM 镜像，将所有指令存储为代码（通常以 JSON 文件的形式）。Terraform 是一个基础设施作为代码的工具，用来提供所有必要的基础设施资源。这里将使用 Packer 的输出作为 Terraform 的输入。通过这种方式，Terraform 将创建一个包含以下内容的完整系统：

- 实例组
- 负载平衡器
- VPC
- 包含代码的 VM 时，使用的其他云元素

这一节的标题可能会让你感到困惑。什么是“不可变的基础设施”？而内容中却主张在每次提交后都要改变整个基础设施？如果了解过函数式语言，可能会更清楚不变性的概念。

可变对象是可以改变其状态的对象。基础架构中，这很容易理解：可以登录到 VM，并下载最新版本的代码。现在的状态已经和介入之前不一样了。

不可变对象是指无法改变其状态的对象。这意味着无法登录到机器上并更改内容。当从映像部署 VM，它就会一直保持原样，直到销毁。这听起来可能非常麻烦，但解决了软件维护的问题。

9.8.2 不可变的优势

首先，不可变的基础设施使配置出错的概念过时。没有配置管理，所以也不会有出错的机会。升级也更安全，因为不能以半生不熟的状态结束。这种状态既不是上一个版本也不是下一个版本，而是介于两者之间。部署过程提供二进制信息：机器是否已创建并运行。

为了让不可变基础设施在不影响正常运行时间的情况下工作，还需要以下几点：

- 负载平衡
- （一定程度的）冗余

毕竟，升级过程包括关闭整个实例。不能依赖这台机器的地址或特定于这台机器的东西。相反，需要至少有两个机器来处理工作负载，同时用最新的版本替换另一个。当完成对一台机器的升

级后，可以对另一台机器重复相同的过程。这样，将拥有两个升级的实例，而不会丢失服务，这种策略称为滚动升级。

可以从流程中认识到，在处理无状态服务时，不可变基础结构的工作效果最好。当服务具有某种形式太久时，就很难正确地实现。这时，必须将持久性级别拆分为一个独立对象，例如包含所有应用程序数据的 NFS 卷。这样的卷可以在实例组中的所有计算机之间共享，每台新机都可以访问前面运行的应用程序留下的公共状态。

9.8.3 使用 Packer 构建实例镜像

考虑到示例应用程序已经是无状态的，可以继续在其上构建不可变的基础设施。由于 Packer 生成的工件是 VM 镜像，必须决定想要使用的格式和构建器。这里将示例的重点放在 Amazon Web Services 上，类似的方法也可以用于其他受支持的提供商。简单的 Packer 模板可能是这样的：

```
{  
  "variables": {  
    "aws_access_key": "",  
    "aws_secret_key": ""  
  },  
  "builders": [ {  
    "type": "amazon-ebs",  
    "access_key": "{{user `aws_access_key`}}",  
    "secret_key": "{{user `aws_secret_key`}}",  
    "region": "eu-central-1",  
    "source_ami": "ami-0f1026b68319bad6c",  
    "instance_type": "t2.micro",  
    "ssh_username": "admin",  
    "ami_name": "Project's Base Image {{timestamp}}"  
  ],  
  "provisioners": [ {  
    "type": "shell",  
    "inline": [  
      "sudo apt-get update",  
      "sudo apt-get install -y nginx"  
    ]  
  ]  
}
```

前面的代码将使用 EBS 构建器为 Amazon Web 服务构建一个镜像。该镜像将驻留在 eu-central-1 区域，并将基于 ami-5900cc36 镜像，这是一个 Debian Jessie 镜像。这里想让构建器是 t2.micro 实例（在 AWS 中相当于一个虚拟机大小）。为了准备镜像，需要运行两个 apt-get 命令。

也可以重用之前定义的 Ansible 代码，而不是使用 Packer 来提供我们的应用程序，可以将 Ansible 替换为提供程序。代码如下所示：

```
{  
  "variables": {  
    "aws_access_key": "",  
    "aws_secret_key": ""  
  },  
  "builders": [{  
    "type": "amazon-ebs",  
    "access_key": "{{user `aws_access_key`}}",  
    "secret_key": "{{user `aws_secret_key`}}",  
    "region": "eu-central-1",  
    "source_ami": "ami-0f1026b68319bad6c",  
    "instance_type": "t2.micro",  
    "ssh_username": "admin",  
    "ami_name": "Project's Base Image {{timestamp}}"  
  ]},  
  "provisioners": [{  
    "type": "ansible",  
    "playbook_file": "./provision.yml",  
    "user": "admin",  
    "host_alias": "baseimage"  
  ]},  
  "post-processors": [{  
    "type": "manifest",  
    "output": "manifest.json",  
    "strip_path": true  
  }]  
}
```

更改在供应程序块中，还添加了一个新的块，即后处理器。这一次，不使用 shell，而是一个不同的供应程序来运行 Ansible。后处理器在这里以机器可读的格式产生构建结果。当 Packer 完成构建所需的工件，就会返回它的 ID 并将其保存在 manifest.json 中。对于 AWS 来说，这就是一个 AMI ID，可以提供给 Terraform。

9.8.4 使用 Terraform 编排基础设施

使用 Packer 创建图像是第一步。之后，希望部署镜像来使用它。可以使用 Terraform 基于 Packer 模板中的镜像构建 AWS EC2 实例。

Terraform 代码示例如下所示：

```
# Configure the AWS provider
provider "aws" {
    region = var.region
    version = "~> 2.7"
}

# Input variable pointing to an SSH key we want to associate with the
# newly created machine
variable "public_key_path" {
    description = <<DESCRIPTION
Path to the SSH public key to be used for authentication.
Ensure this keypair is added to your local SSH agent so
provisioners can connect.
Example: ~/.ssh/terraform.pub
DESCRIPTION

    default = "~/.ssh/id_rsa.pub"
}

# Input variable with a name to attach to the SSH key
variable "aws_key_name" {
    description = "Desired name of AWS key pair"
    default = "terraformer"
}

# An ID from our previous Packer run that points to the custom base image
variable "packer_ami" {

variable "env" {
    default = "development"
}
```

```

variable "region" {
}

# Create a new AWS key pair containing the public key set as the input
# variable
resource "aws_key_pair" "deployer" {
    key_name = var.aws_key_name
    public_key = file(var.public_key_path)
}

# Create a VM instance from the custom base image that uses the previously
# created key
# The VM size is t2.xlarge, it uses a persistent storage volume of 60GiB,
# and is tagged for easier filtering
resource "aws_instance" "project" {
    ami = var.packer_ami

    instance_type = "t2.xlarge"

    key_name = aws_key_pair.deployer.key_name

    root_block_device {
        volume_type = "gp2"
        volume_size = 60
    }

    tags = {
        Provider = "terraform"
        Env = var.env
        Name = "main-instance"
    }
}

```

这将创建一个密钥对和一个使用该密钥对的 EC2 实例，其基于作为变量提供的 AMI。在调用 Terraform 时，将该变量设置为指向 Packer 生成的镜像。

9.9. 总结

目前为止，了解了在项目开始时实现 CI 如何从长远来看节省时间。特别是当与 CD 搭配使用时，还可以减少正在进行的工作。本章中，已经介绍了一些有用的工具，可以实现这两个过程。

上面已经展示了 GitLab CI 如何在 YAML 文件中编写流水。已经讨论了代码检查的重要性，并解释了不同形式的代码检查之间的区别。也介绍了 Ansible，可以帮助配置管理和创建部署代码。最后，尝试了 Packer 和 Terraform 将注意力从创建应用转移到创建系统上。

本章中的知识并不是 C++独有的，可以在使用任何语言和任何技术编写的项目中使用。所有应用程序都需要测试，编译器或静态分析器不足以验证软件。作为架构师，不仅需要考虑当前项目（应用程序本身），还需要考虑产品（应用程序将在其中工作的系统），只交付工作代码是不够的。理解基础设施和部署过程是至关重要的，因为它们是现代系统的新构建块。

下一章将重点介绍软件的安全性。将讨论源代码本身、操作系统级别以及与外部服务和最终用户的交互。

9.10. 练习题

1. CI 在开发过程中如何节省时间？
2. 是否需要单独的工具来实现 CI 和 CD？
3. 什么时候在会议上进行代码审查才有意义？
4. 在 CI 期间，可以使用什么工具来评估代码的质量？
5. 谁参与指定 BDD 的场景？
6. 什么时候会考虑使用不可变的基础设施？什么时候不用？
7. Ansible、Packer 和 Terraform 之间有什么区别？

9.11. 扩展阅读

- Continuous integration/continuous deployment/continuous delivery:
<https://www.packtpub.com/virtualization-and-cloud/hands-continuous-integration-and-delivery>
<https://www.packtpub.com/virtualization-and-cloud/cloud-nativecontinuous-integration-and-delivery>
- Ansible:
<https://www.packtpub.com/virtualization-and-cloud/masteringansible-third-edition>
<https://www.packtpub.com/application-development/handsinfrastructure-automation-ansible-video>
- Terraform:
<https://www.packtpub.com/networking-and-servers/getting-startedterraform-second-edition>
<https://www.packtpub.com/big-data-and-business-intelligence/handsinfrastructure-automation-terraform-aws-video>

- Cucumber:

<https://www.packtpub.com/web-development/cucumber-cookbook>

- GitLab:

<https://www.packtpub.com/virtualization-and-cloud/gitlab-quickstart-guide>

<https://www.packtpub.com/application-development/hands-autodevops-gitlab-ci-video>

第 10 章 代码和部署的安全性

创建了适当的测试之后，有必要执行安全审计，以确保应用程序不会恶意使用。本章描述如何评估代码库的安全性，包括内部开发的软件和第三方模块的安全性，还将展示如何在代码级别和操作系统级别改进现有的软件。

将了解如何设计应用程序，重点关注每个级别的安全性，从代码开始，到依赖关系、体系结构和部署。

本章将讨论以下内容：

- 检查代码安全性
- 检查依赖关系是否安全
- 加固代码
- 强化环境

10.1. 相关准备

本章中使用的例子要求编译器的最低版本：

- GCC 10+
- Clang 3.1+

本章的代码可以在以下 GitHub 页面找到：<https://github.com/PacktPublishing/Software-Architecture-with-Cpp/tree/master/Chapter10>。

10.2. 检查代码安全性

本章中，提供了如何检查代码、依赖和环境的潜在威胁的信息，但遵循本章中列出的每一步并不一定能避免所有问题。我们的目的是展示一些可能的危险，以及应对的方法。因此，应该时刻注意系统的安全性，并将审计作为常规事件。

在互联网普及之前，软件作者并不太关心设计的安全性。毕竟，如果用户提供了错误的数据，最多只会让自己的机器崩溃。为了利用软件漏洞访问受保护的数据，攻击者必须获得对保存数据机器的物理访问权。

即使在设计用于网络内部的软件中，安全问题也常常事后才考虑。以 HTTP 协议为例，尽管它允许对某些资产进行密码保护，但所有数据都以明文传输，这意味着在同一网络上的每个人都可能窃听正在传输的数据。

现在，应该从设计的第一阶段就拥抱安全性，并在软件开发、操作和维护的每一个阶段都要牢记安全性。我们每天生产的大多数软件都是为了以某种方式与其他现有系统连接。

由于忽略了安全措施，不仅让自己，也让合作伙伴面临潜在的攻击、数据泄露，并最终导致对簿公堂。请记住，未能保护个人数据可能会导致数百万美元的罚款。

10.1.1 安全设计

何设计安全架构？最好的方法是像攻击者一样思考。有很多方法可以打破盒子，但会从不同角度寻找元素之间的缝隙。（对于盒子来说，这可能是在盖子和盒子底部之间。）

软件体系结构中，元素之间的连接称为接口。因为主要任务是与外部互动，所以是整个系统中最脆弱的部分。确保接口是受保护的、直观的和健壮的，这将解决软件容易破坏的问题。

使界面易于使用，不易误用

要以一种既易于使用，又不易误用的方式设计接口，请考虑以下练习。假设有一个界面的客户，想实现一个使用支付网关的电子商务商店，或者想实现一个 VR 应用程序，连接到本书示例系统的 Customer API。

作为界面设计的一般规则，应避免以下设计：

- 传递给函数/方法的参数太多
- 参数名称不明确
- 使用输出参数
- 参数依赖于其他参数

为什么这些是有问题的呢？

- 第一个，不仅很难记住，而且很难记住参数的顺序。这可能会导致使用上的错误，而错误又可能导致崩溃和安全问题。
- 第二个，与第一个有着相似的结果。通过减少使用界面的直觉性，用户更容易犯错误。
- 第三个，是第二个的变体。用户不仅记住哪些参数是输入的，哪些是输出的，而且还需要记住应该如何处理输出。谁管理资源的创建和删除？这是如何实现的？它背后的内存管理模型是什么？

现代 C++ 中，返回包含所有必要数据的值，比以往要容易。对于 pair、tuple 和 vector，没有理由使用输出参数。除此之外，返回值还有助于接受不修改对象状态的做法，这反过来可以减少与并发相关的问题。

- 最后一个引入了不必要的认知负担，就像前面的例子一样，这会导致错误和最终的失败。这样的代码也更难测试和维护，因为引入的每个更改都必须考虑到所有可能的组合。不能正确处理的组合，都是对系统的威胁。

上述规则适用于对外接口分，还应该有对内部接口应用类似的方式。方法是验证输入，确保值是正确和合理的，并防止不必要地使用接口提供的服务。

启用资源自动管理

内存泄漏、数据竞争和死锁也会导致系统不稳定，这些症状都是资源管理不善的表现。尽管资源管理是一个令人头痛的话题，但有一种机制可以减少这些问题的数量。其中一种机制是自动资源管理。

资源可以通过操作系统访问，必须确保正确地使用。使用动态分配的内存、打开文件、套接字、进程或线程，所有这些都需要在获取和释放时采取特定的行动，其中一些还需要在其生命周期中进行特定的操作。如果不能在正确的时间释放这些资源，就会导致泄漏。由于资源是有限的，从

长远来看，当无法创建新资源时，泄漏将成为意外行为。

资源管理在 C++ 中是为什么如此重要？因为与许多其他高级语言不同，C++ 中没有垃圾收集机制，软件开发人员负责资源的生命周期。了解这个生命周期有助于创建安全和稳定的系统。

资源管理最常见的习惯性用法是资源获取即初始化 (RAII)。它起源于 C++，也用于其他语言，如 Vala 和 Rust。这种习惯性用法使用对象的构造函数和析构函数分别分配和释放资源。通过这种方式，可以保证当保存资源的对象超出作用域时，使用的资源将释放。

标准库中这种用法的例子是 `std::unique_ptr` 和 `std::shared_ptr` 智能指针类型。其他的例子包括互斥对象——`std::lock_guard`, `std::unique_lock` 和 `std::shared_lock`, 或者文件——`std::ifstream` 和 `std::ofstream`。

稍后将详细讨论的指南支持库 (GSL)，其也实现了一个用于自动化资源管理的指南。通过在代码中使用 `gsl::finally()`，创建了一个 `gsl::final_action` 对象，并添加了一些在调用对象的析构函数时执行的代码。这意味着该代码将在函数成功返回时，以及在异常期间堆栈展开时执行。

这种方法不应该经常使用，因为在设计类时考虑到 RAII 通常是一个更好的主意。但是，如果正在与第三方模块进行对接，并且希望确保包装器的安全性，`finally()` 可以实现这一点。

假设有一个支付运营商，只允许每个帐户同时登录一次。如果不阻止用户进行未来的支付，应该在完成交易处理后立即注销。当一切都按照设计进行时，不会有任何不愉快。但在发生异常的情况下，也希望安全并释放资源。下面是使用 `gsl::finally()` 实现的方法：

```
1 TransactionStatus processTransaction(AccountName account, ServiceToken
2 token,
3 Amount amount)
4 {
5     payment::login(account, token);
6     auto _ = gsl::finally([] { payment::logout(); });
7     payment::process(amount); // We assume this can lead to exception
8
9     return TransactionStatus::TransactionSuccessful;
10 }
```

不管在调用 `payment::process()` 期间发生了什么，至少可以保证，当超出了 `processTransaction()` 的作用域，用户就会注销。

简而言之，使用 RAII 会让类设计阶段更多地考虑资源管理，同时可以完全控制代码。而当意图不清晰时，可能会较少地考虑何时使用接口。

并发性的缺点

虽然并发性提高了性能和资源利用率，但它也使代码更难设计和调试。因为与单线程流不同，操作的时间不能提前确定。单线程代码中，可以向资源写入或从资源读取，但总是知道操作的顺序，因此可以预测对象的状态。

使用并发性，多个线程或进程可以同时从一个对象中读取或修改。如果修改不是原子的，就会遇到问题。

```
1 TransactionStatus chargeTheAccount(AccountNumber accountNumber, Amount
2 amount)
3 {
```

```

4   Amount accountBalance = getAccountBalance(accountNumber);
5   if (accountBalance > amount)
6   {
7       setAccountBalance(accountNumber, accountBalance - amount);
8       return TransactionStatus::TransactionSuccessful;
9   }
10  return TransactionStatus::InsufficientFunds;
11 }

```

当从非并发调用 `chargeTheAccount` 函数时，一切都会结束。如果可能的话，程序会检查账户余额并收取费用。然而，并发执行可能导致负余额。这是因为两个线程可以一个接一个地调用 `getAccountBalance()`，并返回相同的数据，比如 20。执行该后，两个线程都检查当前余额是否高于可用金额。检查后，再修改帐户余额。假设两个事务的金额都是 10，每个线程将设置余额为 $20 - 10 = 10$ 。在两次操作后，该帐户的余额为 10，但它会是 0！

为了缓解类似的问题，可以使用互斥锁和临界区、CPU 提供的原子操作或并发安全的数据结构等解决方案。

互斥锁、临界区和其他类似的并发设计模式防止多个线程修改（或读取）数据。尽管设计并发应用程序时很有用，但与其相关的是一种权衡。因为互斥锁保护的代码只允许单个线程执行，所以可以使部分代码成为单线程的。其他所有的线程都必须等待，直到释放互斥锁。由于引入了等待，会降低代码的性能，但使用并发最初的目标是提高性能。

原子操作意味着使用单个 CPU 指令来获得所需的效果，这个术语可以指任何转换为单个 CPU 指令的高级操作。当单个指令实现的功能超过正常情况下可能实现的功能时，就特别有趣。例如，`compare-and-swap(CAS)` 是一条指令，它将内存位置与给定值进行比较，只有当比较成功时才将该位置的内容修改为新的值。自 C++11 起，有 `<std::atomic>` 头文件可用，其包含几个原子数据类型和操作。例如，CAS 可以实现为一个 `compare_and_exchange_*` 函数集。

最后，并发安全的数据结构（也称为并发数据结构）为需要某种同步的数据结构提供了安全的抽象，例如 Boost。Lockfree(https://www.boost.org/doc/libs/1_66_0/doc/html/lockfree.html) 库提供了用于多个生产者和多个消费者的并发队列和栈。libcds(<https://github.com/khizmax/libcds>) 还提供了有序 list、set 和 map，但在本书写作时，它已经几年没有更新了。

在设计并发处理时要记住一些有用的规则：

- 考虑是否需要并发。
- 通过值传递数据，而不是通过指针或引用。这可以防止在其他线程读取该值时修改该值。
- 如果数据的大小无法按值共享，请使用 `shared_ptr`。这样，可以避免资源泄漏。

10.1.2 安全编码指南和 GSL

标准 C++ 基金会发布了一套指导方针，以记录构建 C++ 系统的最佳实践，是在 <https://github.com/isocpp/CppCoreGuidelines> 下发布的 Markdown 文档。它是一个没有发布时间表的演进文档（不像 C++ 标准本身）。这些准则针对的是现代 C++，这意味着至少要基于 C++11 特性进行实现。

指南中提出的许多规则涵盖了在本章中提出的主题，有一些与接口设计、资源管理和并发性相关的规则。指导方针的编辑是 Bjarne Stroustrup 和 Herb Sutter，都是 C++ 社区中的元老级成员。

这里不会详细描述这些指导原则，但建议读者们自行阅读。本书的灵感来自于其提出的许多规则，本书的例子也遵循这些规则。

为了方便在各种代码库中使用这些规则，Microsoft 将指南支持库 (GSL) 作为一个开源项目发布在 <https://github.com/microsoft/GSL> 上。它是一个头文件库，可以包含在项目中以使用已定义的类型。可以包含整个 GSL，也可以有选择地只使用或计划使用的一些类型。

这个库的另一个有趣之处在于，它使用 CMake 进行构建，使用 Travis 进行持续集成，使用 Catch 进行单元测试。因此，它是的一个很好的例子，应用了在第 7 章、第 8 章和第 9 章中所了解的知识。

10.1.3 防御性编码，验证一切

在关于可测试性的章节中，提到了防御性编程的方法。尽管这个方法严格来说不是一个安全特性，但有助于创建一个健壮的接口。这样的接口反过来可以增加了系统的总体安全性。

作为一种良好的启发式方法，可以将所有外部数据视为不安全数据。这里所说的外部数据，是指通过某些接口（编程接口或用户接口）进入系统的每一个输入。为了明确表达这一点，可以在适当的类型前加上不安全的前缀：

```
1 RegistrationResult registerUser(UnsafeUsername username, PasswordHash
2 passwordHash)
3 {
4     SafeUsername safeUsername = username.sanitize();
5     try
6     {
7         std::unique_ptr<User> user = std::make_unique<User>(safeUsername,
8             passwordHash);
9         CommitResult result = user->commit();
10        if (result == CommitResult::CommitSuccessful)
11        {
12            return RegistrationResult::RegistrationSuccessful;
13        }
14        else
15        {
16            return RegistrationResult::RegistrationUnsuccessful;
17        }
18    }
19    catch (UserExistsException _)
20    {
21        return RegistrationResult::UserExists;
22    }
23}
```

如果已经阅读了指南，就会知道通常应该避免直接使用 C API。C API 中的一些函数可能会以不安全的方式使用，需要注意以防御的方式使用。相反，最好使用 C++中的概念，以确保更好的类型安全性和保护（例如，防止缓冲区溢出）。

防御性编程的另一个方面是对现有代码的重用。每次当尝试某些技术时，确保没有其他人在你之前执行过。在学习一门新的编程语言时，自己编写排序算法可能是一项有趣的挑战，但对于生产

代码来说，使用标准库中提供的排序算法要好得多，哈希密码也是如此。毫无疑问，可以找到一些聪明的方法来计算哈希密码，并将它们存储在数据库中，但更明智的做法是使用经过验证的（不要忘记同行评审！）方式，或是 bcrypt（跨平台文件加密工具）。请记住，代码重用假定检查和审计第三方解决方案，需要与人工检查和审计自己的代码时一样仔细。这个话题将在下一节“依赖关系安全吗？”中继续探讨。

防御性编程不应该变成偏执型编程。检查用户输入是一件合理的事情，而在初始化之后断言初始化的变量是否仍然等于初始值就过分了。需要控制数据和算法的完整性，以及第三方解决方案的完整性。没谁想通过包含语言特性来验证编译器的正确性吧。

简而言之，从安全性和可读性的角度来看，使用 C++ Core Guidelines 中的 `expect()` 和 `ensure()` 并通过输入和转换区分不安全数据和安全数据是一种不错的方式。

10.1.4 最常见的漏洞

要检查代码对于最常见的漏洞是否安全，首先要了解常见漏洞。只有当知道进攻是什么样子，才可能进行防守。开放 Web 应用程序安全项目（OWASP）已经编目了最常见的漏洞，并在 https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project 上发布了它们。撰写本书时，这些漏洞有以下几种：

- **注入:** 也称为 *SQL* 注入，但不仅限于 SQL。当不受信任的数据直接传递给解释器（如 SQL 数据库、NoSQL 数据库、shell 或 eval 函数）时，就会出现此漏洞。攻击者可以通过这种方式访问系统中应该受到保护的部分。
- **失效的身份验证:** 如果身份验证实现不正确，攻击者可能会利用漏洞泄露秘密数据或冒充其他用户。
- **敏感数据暴露:** 由于缺乏加密和适当的访问权限，可能导致敏感数据被公开。
- **XML 外部实体 (XXE):** 一些 XML 处理器可能会公开服务器文件系统的内容或允许远程代码执行。
- **失效的访问控制:** 当访问控制没有正确执行时，攻击者可能会获得对应该限制的文件或数据的访问。
- **安全错误配置:** 使用不安全的默认值和配置不当是最常见的漏洞来源。
- **跨站脚本 (XSS):** 包括和执行不受信任的外部数据，特别是使用 JavaScript，这允许控制用户的 Web 浏览器。
- **不安全的反序列化:** 有缺陷的解析器可能成为拒绝服务攻击或远程代码执行的牺牲品。
- **使用有已知漏洞的组件:** 现代应用程序中的很多代码都作为第三方组件出现。应该定期审计和更新这些组件，因为单个依赖项中的已知安全缺陷可能会导致整个应用程序和数据受损。幸运的是，有一些工具可以自动化实现这个过程。
- **日志记录和（或）监控不足:** 如果系统受到攻击，而日志记录和监控不是很彻底，攻击者可能会获得更深入的访问，但仍然不被注意。

这里不会详细讨论每个提到的漏洞。但在要强调的是，可以通过前面提到的防御性编程技术来防止注入、XML 外部实体和不安全的反序列化。通过将所有外部数据视为不安全数据，可以在开始实际处理之前先删除所有不安全的内容，从而对其进行消杀。

当谈到日志和监控不足时，将在第 15 章中详细介绍。届时，将了解一些可能的可观察性方法，包括日志、监视和分布式跟踪。

10.3. 检查依赖关系是否安全

在计算机系统的早期，所有的程序都是没有外部依赖的庞然大物。自从操作系统出现以来，重要的软件都很难摆脱依赖。这些依赖可以有两种形式：外部依赖和内部依赖：

- 外部依赖项应该存在于运行应用程序的环境中。示例可以包括前面提到的操作系统、动态链接的库和其他应用程序（例如数据库）。
- 内部依赖是想要重用的模块，所以这通常意味着静态库或只包含头文件的库。

这两种依赖关系都提供了潜在的安全风险。随着每一行代码增加漏洞的风险，拥有的组件越多，系统受到攻击的可能性就越高。在下面的部分中，将看到如何检查软件是否容易受到已知漏洞的影响。

10.2.1 公共缺陷检索

要检查软件中已知的安全问题，首先要查看 <https://cve.mitre.org/> 上的常见漏洞和披露（Common vulnerability and exposed, CVE）列表。该名单由几个名为 CVE 编号机构（CNAs）的机构不断更新。这些机构包括供应商和项目、漏洞研究人员、国家和行业 CERT，以及漏洞奖励计划。

该网站还提供了一个搜索引擎，可以使用几种方法来了解漏洞：

- 可以输入漏洞编号。它们的前缀是 CVE，示例包括臭名昭著的 ShellShock CVE-2014-6271，或 CVE-2017-5715，也被称为 Spectre。
- 可以输入漏洞的通用名称，如前面提到的 ShellShock 或 Spectre。
- 可以输入想要检查软件的名称，例如 Bash 或 Boost。

对于每个搜索结果，可以看到描述以及其他 bug 跟踪器和相关资源的引用列表。描述通常会列出受漏洞影响的版本，这样就可以检查计划使用的依赖项是否已经打了补丁。

10.2.2 自动扫描

有一些工具可以帮助您检查依赖项列表。OWASP Dependency-Check(https://www.owasp.org/index.php/OWASP_Dependency_Check) 就是这样一个工具。虽然官方支持 Java 和 .NET，但它对 Python、Ruby、Node.js 和 C++（当与 CMake 或 autoconf 一起使用时）有实验性的支持。除了作为一个独立的工具，它还集成了持续集成/持续部署（CI/CD）软件，如 Jenkins、SonarQube 和 CircleCI。

另一个允许检查已知漏洞的依赖关系的工具是 Snyk，这是一个具有多个级别支持的商业产品。它还做了比 OWASP 依赖检查更多的事情，因为 Snyk 还可以检查容器映像和许可证遵从性问题，还提供了更多与第三方解决方案的集成。

10.2.3 自动依赖升级管理

监视依赖项是否存在漏洞只是确保项目安全的第一步。在此之后，需要采取行动，手动更新失效的依赖项。正如期望的那样，也有一些自动化的解决方案。其中一个是 Dependabot，它会扫描你的源代码库，并在有安全相关更新可用的时候发出一个 pull 请求。在写这本书的时候，Dependabot 还不支持 C++。但是，可以与您的应用程序可能使用的其他语言一起使用。除此之外，还可以扫描 Docker 容器在基本映像中发现的漏洞。

自动化的依赖项管理需要成熟的测试支持，没有测试的情况下切换依赖版本可能会导致不稳定和 bug，防止依赖关系升级相关问题的一种保护措施是使用包装器与第三方代码进行接口。这样的包装器有自己的测试套件，可以立即告知在升级期间接口何时失效。

10.4. 加固代码

可以通过使用现代的 C++ 结构，来减少代码中常见安全漏洞的数量，而不是使用较老的 C 代码。然而，总是有更安全的抽象也证明是脆弱的。仅仅选择更安全的实现，并认为已经尽了最大努力是不够的。大多数情况下，有一些方法可以加强代码。

什么是加强代码？根据定义，是减少系统表面脆弱性的过程。通常，这意味着关闭不使用的功能，并着眼于更简单的系统而不是复杂的系统。这还可能使用工具来增加现有功能的健壮性。

当应用于操作系统级别时，这些工具可能意味着内核补丁、防火墙和入侵检测系统 (IDS)。在应用程序级别，这可能意味着各种缓冲区溢出和下溢保护机制，使用容器和虚拟机 (VM) 进行特权隔离和进程隔离，或强制加密通信和存储。

本节中，将关注应用程序级别的示例，而下一节将关注操作系统级别。

10.4.1 面向安全性的内存分配器

如果非常重视保护应用程序免受堆相关的攻击，例如：堆溢出、释放后使用或双重释放，可以考虑用面向安全的版本替换标准内存分配器。以下是两个可能会引起兴趣的项目：

- FreeGuard，可在 <https://github.com/UTSASRG/FreeGuard> 获得，并在 <https://arxiv.org/abs/1709.02746> 的一篇论文中描述
- GrapheneOS 项目的 hardened_malloc，可在 https://github.com/GrapheneOS/hardened_malloc 处获得

FreeGuard 于 2017 年发布，自那以后除了零星的 bug 修复外，并没有太大的变化。另一方面，hardened_malloc 在积极地开发。这两个分配器都设计为作为标准 malloc() 的替代品。可以在不修改应用程序的情况下使用它们，只需设置 LD_PRELOAD 环境变量或将库添加到 /etc/preload.so 配置文件中。FreeGuard 的目标是 Linux 和 64 位 x86 系统上的 Clang 编译器，而 hardened_malloc 的目标是更广泛的兼容性，尽管目前主要支持 Android 的 Bionic，musl 和 glibc。hardened_malloc 也是基于 OpenBSD 的 alloc，OpenBSD 本身就是以安全为中心的项目。

可以将所使用的集合替换为更安全的等价物，而不是替换内存分配器。SaferCPlusPlus(<https://duneroadrunner.github.io/SaferCPlusPlus/>) 项目提供了 std::vector<>、std::array<> 和 std::string 的替代品，这些可以作为现有代码中的临时替换。该项目还包括防止未初始化使用或符号不匹配的基本类型的替代品、并发数据类型，以及指针和引用的替代品。

10.4.2 自动检查

一些工具可以有助于确保正在构建的系统的安全性。

编译器警告

虽然编译器警告本身不是一种工具，但可以使用并调整编译器警告，以获得每个 C++ 开发人员都将使用的工具——C++ 编译器——更好的输出。

因为编译器可以做一些比标准要求的更深入的检查，所以建议利用这种可能性。当使用 GCC 或 Clang 等编译器时，推荐的设置包括-Wall -Wextra 标志。这将产生更多的诊断信息，并在代码没有遵循诊断时产生警告。如果希望更严格一些，还可以启用-Werror，它会把所有警告转换为错误，并防止编译没有通过增强诊断的代码。如果想严格遵守标准，有-pedantic 和-pedanticerrors 标志，将检查代码是否符合标准。

当使用 CMake 进行构建时，可以使用以下函数在编译期间启用这些标志：

```
add_library(customer ${SOURCES_GO_HERE})
target_include_directories(customer PUBLIC include)
target_compile_options(customer PRIVATE -Werror -Wall -Wextra)
```

这样，除非修复编译器报告的所有警告（已转换的错误），否则编译将失败。

也可以在 OWASP(https://www.owasp.org/index.php/C-Based_Toolchain_Hardening) 和 Red Hat(<https://developers.redhat.com/blog/2018/03/21/compiler-and-linker-flags-gcc/>) 的这些文章中找到工具链的建议设置。

静态分析

有一类工具可以提高代码的安全性，这就是所谓的静态应用程序安全测试 (SAST) 工具，只关注安全方面的静态分析工具。

SAST 工具可以很好地集成到 CI/CD 流水中，它们只是读取源代码。输出通常也适用于 CI/CD，因为它突出了源代码中特定位置发现的问题。另一方面，静态分析可能会忽略许多类型的问题，这些问题无法自动找到，或者仅通过静态分析无法找到。这些工具也不关心与配置相关的问题，因为配置文件没有在源代码本身中表示。

C++ SAST 工具的例子包括以下开源解决方案：

- Cppcheck(<http://cppcheck.sourceforge.net/>)，是一个通用的静态分析工具，专注于消除误报
- Flawfinder(<https://d Wheeler.com/flawfinder/>)，似乎没有积极地维护
- LGTM(<https://lgtm.com/help/lgtm/about-lgtm>)，支持几种不同的语言，具有自动分析拉请求的功能
- SonarQube(<https://www.sonarqube.org/>)，可以很好的 CI/CD 集成和语言覆盖，并提供了一个商业版本
- Checkmarx CxSAST(<https://www.checkmarx.com/products/staticapplication-security-testing/>)，可以零配置和语言覆盖

- CodeSonar(<https://www.grammatech.com/products/codesonar>)，专注于深度分析和发现缺陷
- Klocwork(<https://www.perforce.com/products/klocwork>)，专注于准确性
- Micro Focus Fortify(<https://www.microfocus.com/en-us/products/staticcode-analysis-sast/overview>)，具有广泛的语言支持和集成的其他工具(由同一制造商)
- Parasoft C/C++ 测试 (<https://www.parasoft.com/products/ctest>)，这是一个静态和动态分析、单元测试、跟踪等的(集成)解决方案
- Polyspace Bug Finder 来自 MathWorks(<https://www.mathworks.com/products/polyspace-bug-finder.html>)，可以与 Simulink 模型的集成
- Veracode 静态分析 (<https://www.veracode.com/products/binarystatic-analysis-sast>)，是一个用于静态分析的 SaaS 解决方案
- 白帽哨兵源 (<https://www.whitehatsec.com/platform/staticapplication-security-testing/>)，也专注于消除误报

动态分析

就像对源代码执行静态分析一样，对生成的二进制文件执行动态分析，名称中的“动态”是指观察代码在动作中处理的实际数据。当关注安全性时，这类工具也可以称为动态应用程序安全测试(DAST)。

与 SAST 相比，主要优势是可以找到许多从源代码分析角度看不到的流。当然，这也有缺点，必须运行应用程序才能执行分析，这可能既耗时又耗内存。

DAST 工具通常专注于 Web 相关的漏洞，如 XSS、SQL(和其他)注入，或公开的敏感信息。下一小节中，将更多地关注更通用的动态分析工具 Valgrind。

Valgrind 和应用验证器

Valgrind 通常称为内存泄漏调试工具。事实上，它是一个帮助构建动态分析工具的工具框架，而不一定与内存问题相关。除了内存错误检测器之外，该工具套件目前还包括一个线程错误检测器、一个缓存和分支预测分析器以及一个堆分析器。它支持类 unix 操作系统(包括 Android)上的各种平台。

本质上 Valgrind 作为 VM，将二进制转换为一种更简单的形式，称为中间表示。它不是在一个实际的处理器上运行程序，而是在这个 VM 下执行，这样就可以分析和验证每个调用。

若在 Windows 上开发，可以使用应用程序验证器(AppVerifier)，而不是 Valgrind。AppVerifier 可以检测稳定性和安全性问题，可以监视正在运行的应用程序和用户模式驱动程序，以查找内存问题，如泄漏和堆损坏、线程和锁问题、无效句柄等。

消杀工具

消杀程序是基于代码编译时插装的动态测试工具，可以提高系统的整体稳定性和安全性，以及避免未定义行为。在 <https://github.com/google/sanitizers>，可以找到 LLVM(Clang 是基于 LLVM) 和 GCC 的实现。它们解决了内存访问、内存泄漏、数据竞争和死锁、未初始化的内存使用和未定义行为。

AddressSanitizer(ASan) 保护代码免受与内存寻址相关的问题，例如全局缓冲区溢出、释放后使用或返回后堆栈使用。尽管这是同类中最快的解决方案之一，但仍然会将这个过程减慢两倍左右。最好在运行测试和进行开发时使用，但在生产版本中关闭。其可以通过`-fsanitize=address` 标志开启。

AddressSanitizerLeakSanitizer(LSan) 集成了 ASan 来查找内存泄漏，默认在 x86_64 Linux 和 x86_64 macOS 上启用。需要设置一个环境变量，`ASAN_OPTIONS=detect_leaks=1`，这样 LSan 在过程结束时就可以执行泄漏检测。LSan 也可以作为一个独立的库使用，而不需要 AddressSanitizer，但是这种模式的测试要少得多。

Threadsantizer(TSan) 可以检测并发性问题，如数据竞争和死锁。可以使用`-fsanitize=thread` 标志来启用。

MemorySanitizer(MSan) 专注于与访问未初始化内存相关的 bug，实现了在前一小节中介绍的 Valgrind 的一些特性。MSan 支持 64 位 x86、ARM、PowerPC 和 MIPS 平台。可以使用 Clang 的`-fsanitize=memory -fPIE -pie` 标志来启用（还会打开独立于位置的可执行文件，这个稍后会讨论）。

硬件辅助地址消杀程序 (HWASAN) 类似于常规的 ASan。主要区别是在可能的情况下使用硬件辅助，该特性目前仅在 64 位 ARM 架构上可用。

UndefinedBehaviorSanitizer(UBSan) 查找未定义行为的其他可能原因，如整数溢出、被零除或不当的位移位操作。可以使用 Clang 的`-fsanitize=undefined` 标志来启用它。

尽管消杀程序可以发现许多潜在的问题，但它们的性能仅与运行它们的测试一样。在使用消杀程序时，请保持测试的代码高覆盖率。否则，可能会对安全性产生错觉。

模糊测试

模糊测试是 DAST 工具的子类，其检查应用程序在遇到无效、意外、随机或恶意格式的数据时的行为。当对跨信任边界的接口（如最终用户文件上传表单或输入）使用这种检查时，会特别有用。

这个类别中一些有趣的工具包括：

- Peach Fuzzer: <https://www.peach.tech/products/peach-fuzzer/>
- PortSwigger Burp: <https://portswigger.net/burp>
- OWASP Zed 攻击代理项目: https://www.owasp.org/index.php/OWAS_Zed_Attack_Proxy_Project
- Google 的 ClusterFuzz: <https://github.com/google/clusterfuzz> (和 OSSFuzz: <https://github.com/google/oss-fuzz>)

10.4.3 处理隔离和沙盒

如果想在自己的环境中运行未经验证的软件，可能希望将其与系统的其余部分隔离开来。一些使用沙箱的方法是通过虚拟机、容器或微虚拟机，如 AWS Lambda 使用的是 Firecracker(<https://firecracker-microvm.github.io/>)。

这样，应用程序的崩溃、泄漏和安全问题就不会传播到整个系统，从而使它变得无用或受损。由于每个进程都有自己的沙箱，最坏的情况是只丢失这一个服务。

对于 C 和 C++ 代码，也有沙盒 API(SAPI: <https://github.com/google/sandboxed-api>) 是一个由 Google 领导的开源项目，允许为库而不是整个过程构建沙盒。它在 Google 的 Chrome 和 Chromium 网络浏览器中就有使用。

尽管 VM 和容器可以是过程隔离策略的一部分，但不要将它们与微服务混淆，后者通常使用类似的构建块。微服务是一种架构设计模式，它们并不自动等于更好的安全性。

10.5. 强化环境

即使采取了必要的预防措施来确保依赖项和代码不受已知漏洞的影响，仍然存在一个可能危及安全策略的区域。所有应用程序都需要执行环境，这可能意味着容器、VM 或操作系统。有时，这也意味着底层基础设施。

当运行应用程序的操作系统具有开放访问权限时，将应用程序加固到最大限度是不够的。这样，攻击者就可以直接从系统或基础设施级别获得对数据的未授权访问，而不是针对应用程序。

本节将重点介绍一些可以在最底层执行级别的应用的加固技术。

10.5.1 静态和动态链接

链接是在编译后将所编写的代码及其各种依赖项（如标准库）组合在一起时发生的过程。链接可以发生在构建时、加载时（操作系统执行二进制文件时）或运行时，就像插件和其他动态依赖项一样。最后两个用例只适用于动态链接。

动态链接和静态链接有什么区别呢？使用静态链接，所有依赖项的内容都被复制到生成的二进制文件中。当程序加载时，操作系统将这个二进制文件放入内存中并执行它。静态链接由称为链接器的程序执行，作为构建过程的最后一步。

因为每个可执行文件都必须包含所有依赖项，所以静态链接的程序体积往往比较大。这也有好处，由于执行问题所需的所有东西都已经可用，因此执行速度会更快，并且将程序加载到内存中所花费的时间总是相同的。对依赖关系的任何更改都需要重新编译和链接，如果不改变产生的二进制文件，就无法升级依赖项。

动态链接中，生成的二进制文件包含编写的代码，但没有依赖项的内容，只有对实际库的引用，需要单独加载。在加载期间，动态加载器的任务是找到适当的库，并将它们与二进制文件一起加载到内存中。当多个应用程序同时运行，并且每个应用程序都使用类似的依赖项（例如 JSON 解析库或 JPEG 处理库）时，动态方式可以减少二进制文件内存的使用，这是因为只能将给定库的副本加载到内存中。相比之下，对于静态链接的二进制文件，相同的库将作为生成的二进制文件的一部分反复加载。当需要升级某个依赖项时，可以这样做，而不需要接触系统的其他组件。下一次将应用程序加载到内存中时，将自动引用新升级的组件。

静态和动态链接也有安全问题，对动态链接的应用程序更容易获得未经授权的访问。这可以通过替换受损的动态库来代替常规的动态库，或者通过在每个新执行的进程中预加载某些库来实现。

当将静态链接与容器结合使用时（在后面的章节中会详细解释），将得到一个小型、安全的沙盒执行环境。甚至可以对基于微内核的 VM 使用这样的容器，从而减少攻击面。

10.5.2 随机地址空间分配

随机地址空间分配 (ASLR) 是一种用于防止基于内存的攻击的技术，工作原理是将程序和数据的标准内存布局替换为随机的布局。这意味着攻击者无法可靠地跳转到在没有 ASLR 的系统上存在的特定函数。

当与无执行 (NX) 位支持结合使用时，这种技术会更加有效。NX 位将内存中的某些页面（如堆和堆栈）标记为只包含无法执行的数据。NX 位支持已经在大多数主流操作系统中实现，只要硬件支持就可以使用。

10.5.3 DevSecOps

要在可预测的基础上交付软件增量，最好采用 DevOps 的理念。简而言之，DevOps 意味着打破传统模式，鼓励商业、软件开发、软件运营、质量保证和客户之间的沟通。DevSecOps 是 DevOps 的一种形式，强调在设计过程的每一步都要考虑到安全性。

这意味着正在构建的应用程序从一开始就具有可观察性，利用 CI/CD 流水，并定期扫描漏洞。DevSecOps 让开发人员在底层基础设施的设计上有了发言权，也让操作专家在组成应用程序的软件包设计上有了发言权。由于每一个增量都代表一个完整的工作系统（尽管不是完全功能），因此定期执行安全审计，因此比正常情况下花费的时间更少。这将导致更快、更安全的发布，并允许对安全事件做出更快的反应。

10.6. 总结

本章中，讨论了安全系统的不同方面。由于安全性是一个复杂的主题，不能只从应用程序的角度来处理。现在所有的应用程序都在某些环境中运行，重要的是控制这个环境并根据需求塑造，或者通过沙箱和隔离代码来保护自己不受环境的影响。

阅读完本章后，现在可以在依赖项和自己的代码中搜索漏洞了。知道如何设计提高安全性的系统，以及使用什么工具来发现可能的缺陷。维护安全性是一个持续的过程，良好的设计可以减少后续的维护工作。

下一章将讨论可扩展性，以及在发展系统时可能面临的各种挑战。

10.7. 练习题

1. 为什么安全性在现代系统中很重要？
2. 并发的挑战是什么？
3. 什么是 C++ 核心指南？
4. 安全编码和防御编码的区别是什么？
5. 如何检查软件是否包含已知的漏洞？
6. 静态分析和动态分析的区别是什么？
7. 静态链接和动态链接有什么区别？
8. 如何使用编译器修复安全问题？
9. 如何在 CI 流水中体现相应的安全意识？

10.8. 扩展阅读

网络安全概述:

- <https://www.packtpub.com/eu/networking-and-servers/handscybersecurity-architects>
- <https://www.packtpub.com/eu/networking-and-servers/informationsecurity-handbook>
- https://www_OWASP.org/index.php/Main_Page
- <https://www.packtpub.com/eu/networking-and-servers/practical-securityautomation-and-testing>

并发性:

- <https://www.packtpub.com/eu/application-development/concurrentpatterns-and-best-practices>
- <https://www.packtpub.com/eu/application-development/mastering-cmultithreading>

操作系统安全强化:

- <https://www.packtpub.com/eu/networking-and-servers/mastering-linuxsecurity-and-hardening>

第 11 章 性能

选择 C++ 作为项目的关键编程语言的最常见原因是性能需求。C++ 在性能方面的竞争中有明显优势，但要获得最佳结果需要理解相关问题。本章主要讨论如何提高 C++ 软件的性能。我们将首先展示用于测量性能的工具，再展示提高单线程计算速度的技术。然后将讨论如何使用并行计算。最后，展示如何使用 C++20 的协程实现非抢占式多任务处理。

本章将讨论以下内容：

- 测定性能
- 让编译器生成高性能代码
- 并行计算
- 使用协程

首先，指定运行本章中的示例需要哪些东西。

11.1. 相关准备

要运行本章中的示例，应该安装以下组件：

- CMake 3.15+
- 一个支持 C++20 的 range 和协程的编译器，例如 GCC 10+

本章的代码可以在以下 GitHub 页面找到：<https://github.com/PacktPublishing/Software-Architecture-with-Cpp/tree/master/Chapter11>。

11.2. 测定性能

为了有效地改进代码的性能，必须从测量其性能开始。如果不知道真正的瓶颈在哪里，将最终优化错误的地方，损失时间，并对努力工作几乎没有收获感到惊讶和沮丧。本节中，将展示如何正确地使用基准测试来度量性能，如何成功地分析代码，以及深入了解分布式系统中的性能。

11.2.1 进行准确且有意义的测量

为了进行精确和可重复的测量，可能还希望将机器设置为性能模式，而不是通常的默认省电模式。如果需要较低的系统延迟，可能需要在进行基准测试的机器和生产环境中永久禁用节能功能。很多时候，这可能意味着进入 BIOS 并正确配置服务器。注意，如果使用公共云提供商，这样的操作是不可能的。如果有 root/admin 权限，操作系统通常也可以控制一些设置。例如，在 Linux 系统上，可以通过执行以下命令强制 CPU 以其最高频率运行：

```
sudo cpupower frequency-set --governor performance
```

此外，为了获得有意义的结果，可能希望在与生产环境尽可能接近的系统上执行测量。除了配置之外，RAM 的不同速度、CPU 缓存的数量和 CPU 的微架构等方面也会影响结果，导致错误的结论。对于硬盘设置，甚至网络拓扑结构和使用的硬件也是如此。构建的软件也扮演着至关重要的

角色: 从使用的固件, 通过操作系统和内核, 一路向上的软件堆会找到依赖。最好有第二个与生产环境相同的环境, 并使用相同的工具和脚本进行治理。

现在有了测量的环境, 看看能测试到什么。

11.2.2 利用不同类型的测量工具

有几种方法可以测量性能, 每种方法关注不同的范围。

基准测试可以用于在制作的测试中计时系统的速度。通常, 会测试完成时间或另一个性能指标(如每秒处理的订单)。有几种基准:

- 微基准测试, 可以测量小代码片段的执行。将在下一节中介绍。
- 用人工数据模拟在更大范围内进行的合成测试。如果无法访问目标数据或目标硬件, 这种方式就可能很有用。例如, 当计划检查正在工作的硬件的性能, 但它还不存在, 或者当计划处理传入的通信流, 但只能假设通信流的外观。
- 回放, 这是在现实工作负载下测量性能的一种非常准确的方法。其思想是记录进入生产系统的所有请求或工作负载, 通常带有时间戳。然后可以在基准测试系统中“重播”这些转储, 并考虑到它们之间的时间差异, 检查它的执行情况。这样的基准测试可以很好地了解代码或环境的变化对系统延迟和吞吐量的影响。
- 行业标准, 这是一个很好的方式来了解自己的产品如何与其竞争对手的相比。这类基准测试包括用于 CPU 的 SuperPi, 用于显卡的 3D Mark, 以及用于人工智能处理器的 ResNet-50。

除了基准测试之外, 另一种用于测量性能的工具是分析器。分析器不只提供总体性能指标, 并允许检查代码正在做什么并寻找瓶颈。对于捕获使系统变慢的意外事件非常有用。

掌握系统性能的最后一种方法是跟踪, 其本质是记录执行期间系统行为的一种方法。通过监视请求完成各种处理步骤所需的时间(例如不同类型的微服务处理), 可以了解系统的哪些部分需要提高性能, 或者系统处理不同类型请求的情况: 要么是不同的类型, 要么是接受或拒绝的类型。

现在来看看微基准测试。

11.2.3 微基准测试

微基准测试用于测量“微”代码片段的执行速度。如果想知道如何实现给定的功能, 或者不同的第三方库处理相同任务的速度有多快, 它们是完成这项工作的最佳工具。虽然它们不能代表现实环境, 但非常适合进行这样的小实验。

接下来, 展示如何使用 C++ 中最常用的创建微基准测试的框架: Google 基准测试来执行这样的实验。

设置 Google 基准测试

通过使用 Conan 将这个库引入代码中, 并在 conanfile.txt 中添加以下内容:

```
[requires]
benchmark/1.5.2
```

```
[generators]
CMakeDeps
```

将使用 CMakeDeps 生成器，因为它是 Conan2.0 中推荐的 CMake 生成器。它依赖于 CMake 的 `find_package` 特性来使用野蛮依赖管理器安装包。要在发布版本中安装依赖关系，请执行以下命令：

```
cd <build_directory>
conan install <source_directory> --build=missing -s build_type=Release
```

如果使用的是自定义的 Conan 配置，记得在这里也添加。

以 `CMakeLists.txt` 的方式使用也非常简单：

```
list(APPEND CMAKE_PREFIX_PATH "${CMAKE_BINARY_DIR}")
find_package(benchmark REQUIRED)
```

首先，将构建目录添加到 `CMAKE_PREFIX_PATH` 中，这样 CMake 就可以找到 Conan 生成的配置和/或目标文件。接下来，只需使用它们来找到依赖即可。

当要创建几个微基准测试时，可以使用 CMake 函数进行定义：

```
function(add_benchmark NAME SOURCE)
    add_executable(${NAME} ${SOURCE})
    target_compile_features(${NAME} PRIVATE cxx_std_20)
    target_link_libraries(${NAME} PRIVATE benchmark::benchmark)
endfunction()
```

该函数将能够创建单转译单元微基准测试，每一个都使用 C++20 并链接到 Google 基准测试库。现在使用它来创建第一个微基准可执行文件：

```
add_benchmark(microbenchmark_1 microbenchmarking/main_1.cpp)
```

现在准备在源文件中放入一些代码。

第一个微基准测试

我们将测试在排序后的数组中使用二分法进行查找的速度比线性查找快多少。先从创建排序数组的代码开始：

```
1 using namespace std::ranges;
2
3 template <typename T>
```

```

4 auto make_sorted_vector(std::size_t size) {
5     auto sorted = std::vector<T>{};
6     sorted.reserve(size);
7
8     auto sorted_view = views::iota(T{0}) | views::take(size);
9     std::ranges::copy(sorted_view, std::back_inserter(sorted));
10    return sorted;
11 }

```

数组将包含 size 个元素，所有的数字从 0 到 size - 1 按升序排列。现在指定要查找的元素和容器大小：

```

1 constexpr auto MAX_HAYSTACK_SIZE = std::size_t{10'000'000};
2 constexpr auto NEEDLE = 2137;

```

我们会测量查找需要多长时间。简单的线性搜索可以这样实现：

```

1 void linear_search_in_sorted_vector(benchmark::State &state) {
2     auto haystack = make_sorted_vector<int>(MAX_HAYSTACK_SIZE);
3     for (auto _ : state) {
4         benchmark::DoNotOptimize(find(haystack, NEEDLE));
5     }
6 }

```

这里，可以看到 Google 基准测试的首次使用。每个微基准测试都应该接受 State 作为参数。这种特殊类型的功能如下：

- 包含有关执行的迭代和测试在计算上花费时间的信息
- 如果需要，计算处理的字节数
- 返回其他状态信息，比如需要进一步运行（通过 `keeprunning()` 成员函数）
- 可用于暂停和恢复迭代的计时（分别通过 `PauseTiming()` 和 `ResumeTiming()` 成员函数）

测量循环中的代码，根据运行这个特定基准测试的总允许时间进行所需的多次迭代。我们的数组在循环之外创建，创建耗时不会计入测量。

循环内部，有一个接收器助手，名为 `DoNotOptimize`，目的是确保编译器不会优化掉计算。例子中，将把 `std::find` 的结果标记为必要的，因此查找指针的实际代码没有优化掉。使用 objdump 等工具或 Godbolt 和 QuickBench 等网站可以查看想运行的代码是否优化掉。QuickBench 还有一个优势，就是在云端运行基准测试，并在线分享测试结果。

回到手头上的任务，现在有一个线性搜索的微基准，现在让我们在另一个微基准中计算二分搜索的时间：

```

1 void binary_search_in_sorted_vector(benchmark::State &state) {
2     auto haystack = make_sorted_vector<int>(MAX_HAYSTACK_SIZE);
3     for (auto _ : state) {
4         benchmark::DoNotOptimize(lower_bound(haystack, NEEDLE));
5     }
6 }

```

新基准非常类似，只在使用的函数上有所不同:`lower_bound` 将执行二进制搜索。注意，与基础示例类似，不检查迭代器返回的是 `vector` 中的有效元素，还是 `vector` 的末尾。在 `lower_bound` 的情况下，可以检查迭代器指向的元素是否就是要找的元素。

现在有了微基准函数，可以通过添加以下代码来创建实际的基准：

```
1 BENCHMARK(binary_search_in_sorted_vector);
2 BENCHMARK(linear_search_in_sorted_vector);
```

如果接受默认基准设置，那么只需通过这些设置即可。作为最后一步，添加 `main()` 函数：

```
1 BENCHMARK_MAIN();
```

就这么简单！或者，也可以使用 `benchmark_main` 来链接应用。使用 Google Benchmark 的 `main()` 函数的好处是提供了一些默认选项。如果编译基准测试，并将`--help` 作为参数运行它，将看到以下内容：

```
benchmark [--benchmark_list_tests={true|false}]
           [--benchmark_filter=<regex>]
           [--benchmark_min_time=<min_time>]
           [--benchmark_repetitions=<num_repetitions>]
           [--benchmark_report_aggregates_only={true|false}]
           [--benchmark_display_aggregates_only={true|false}]
           [--benchmark_format=<console|json|csv>]
           [--benchmark_out=<filename>]
           [--benchmark_out_format=<json|console|csv>]
           [--benchmark_color={auto|true|false}]
           [--benchmark_counters_tabular={true|false}]
           [--v=<verbosity>]
```

这是一组很好的特性。例如，在设计实验时，可以使用 `benchmark_format` 开关来获得 CSV 输出，以便在图表上绘图。

现在通过运行编译后的不带命令行参数的可执行文件，来看看基准测试的实际效果。运行`./microbenchmark_1` 可能的输出如下：

```
2021-02-28T16:19:28+01:00
Running ./microbenchmark_1
Run on (8 X 2601 MHz CPU s)
Load Average: 0.52, 0.58, 0.59
-----
Benchmark Time CPU Iterations
-----
linear_search_in_sorted_vector 984 ns 984 ns 746667
binary_search_in_sorted_vector 18.9 ns 18.6 ns 34461538
```

从一些关于运行环境的数据 (基准测试的时间、可执行名称、服务器的 CPU 和当前负载) 开始，得到定义的每个基准测试的结果。对于每个基准测试，可以得到每个迭代的平均挂钟时间、每个迭代的平均 CPU 时间，以及基准测试运行的迭代次数。默认情况下，单个迭代越长，所经过的迭代就越少。运行更多的迭代可以获得更稳定的结果。

向微基准测试传递参数

如果要测试更多处理手头问题的方法，可以寻找重用基准测试代码的方法，只要将其传递给用于执行查找的函数，就可以使用 Google 基准测试的一个特性。通过将参数作为附加参数添加到函数签名中，该框架实际上可以将任何想要的参数传递给基准测试。

有了这个特性，基准测试的统一签名会是什么样子：

```
1 void search_in_sorted_vector(benchmark::State &state, auto finder) {
2     auto haystack = make_sorted_vector<int>(MAX_HAYSTACK_SIZE);
3     for (auto _ : state) {
4         benchmark::DoNotOptimize(finder(haystack, NEEDLE));
5     }
6 }
```

可以注意到函数的新 `finder` 参数，它用于之前调用 `find` 或 `lower_bound` 的位置。现在，可以使用与上次不同的宏来进行两个微基准测试：

```
1 BENCHMARK_CAPTURE(search_in_sorted_vector, binary, lower_bound);
2 BENCHMARK_CAPTURE(search_in_sorted_vector, linear, find);
```

`BENCHMARK_CAPTURE` 宏接受函数、名称后缀和任意数量的参数。如果想要更多，可以把它们传到这里。基准函数可以是常规函数或模板——两者都支持。现在来看看当运行代码时得到了什么：

```
Benchmark Time CPU Iterations  
  
search_in_sorted_vector/binary 19.0 ns 18.5 ns 28000000  
search_in_sorted_vector/linear 959 ns 952 ns 640000
```

传递给函数的参数不是函数名的一部分，而是函数名和后缀的一部分。

现在，看看如何进一步定制基准测试。

向微基准传递数值参数

设计像这样的实验时，常见的需求是在不同大小的参数上进行检验。在 Google 基准测试中，可以通过多种方式解决此类需求。最简单的方法是在 BENCHMARK 宏返回的对象上添加对 Args() 的调用。这样，就可以传递一组值给相应的微基准中使用。要使用传递的值，需要对基准函数进行如下修改：

```
1 void search_in_sorted_vector(benchmark::State &state, auto finder) {  
2     const auto haystack = make_sorted_vector<int>(state.range(0));  
3     const auto needle = 2137;  
4     for (auto _ : state) {  
5         benchmark::DoNotOptimize(finder(haystack, needle));  
6     }  
7 }
```

对 state.range(0) 的调用将读取传递的第 0 个参数，其支持任意数量参数的传递。例子中，用于参数化子数组的大小。如果想传递一个范围的值集呢？这样，就可以更容易地看到更改大小如何影响性能。可以在基准测试中调用 Range，而非 Args：

```
1 constexpr auto MIN_HAYSTACK_SIZE = std::size_t{1'000};  
2 constexpr auto MAX_HAYSTACK_SIZE = std::size_t{10'000'000};  
3 BENCHMARK_CAPTURE(search_in_sorted_vector, binary, lower_bound)  
    ->RangeMultiplier(10)  
    ->Range(MIN_HAYSTACK_SIZE, MAX_HAYSTACK_SIZE);  
4 BENCHMARK_CAPTURE(search_in_sorted_vector, linear, find)  
    ->RangeMultiplier(10)  
    ->Range(MIN_HAYSTACK_SIZE, MAX_HAYSTACK_SIZE);
```

使用预定义的最小值和最大值来指定范围边界。然后，告诉基准测试工具通过乘以 10 而不是默认值来创建范围。当运行这样的基准时，可以得到如下的结果：

```
Benchmark Time CPU Iterations

search_in_sorted_vector/binary/1000 0.2 ns 19.9 ns 34461538
search_in_sorted_vector/binary/10000 24.8 ns 24.9 ns 26352941
search_in_sorted_vector/binary/100000 26.1 ns 26.1 ns 26352941
search_in_sorted_vector/binary/1000000 29.6 ns 29.5 ns 24888889
search_in_sorted_vector/binary/10000000 25.9 ns 25.7 ns 24888889
search_in_sorted_vector/linear/1000 482 ns 474 ns 1120000
search_in_sorted_vector/linear/10000 997 ns 1001 ns 640000
search_in_sorted_vector/linear/100000 1005 ns 1001 ns 640000
search_in_sorted_vector/linear/1000000 1013 ns 1004 ns 746667
search_in_sorted_vector/linear/10000000 990 ns 1004 ns 746667
```

分析这些结果时，可能想知道为什么线性搜索没有显示出线性增长。那是因为我们寻找的是一个定值，可以在固定的位置发现。如果数组中包含了指针，那么需要相同数量的操作来找到它，而不管数组的大小，所以执行时间停止增长（仍然可能受到小波动的影响）。

为什么不试试尝试换换查找的位置呢？

以编程方式生成传递参数

在简单的函数中生成数组大小和对应值的位置可能很容易。Google 基准可以进行这样的操作，所以来展示下在实践这种方式是如何工作的。

首先重写基准函数，在每次迭代中使用两个参数：

```
1 void search_in_sorted_vector(benchmark::State &state, auto finder) {
2     const auto needle = state.range(0);
3     const auto haystack = make_sorted_vector<int>(state.range(1));
4     for (auto _ : state) {
5         benchmark::DoNotOptimize(finder(haystack, needle));
6     }
7 }
```

`state.range(0)` 将标记寻找值的位置，而 `state.range(1)` 将是数组的大小，这意味着每次需要传递两个值。创建一个生成函数：

```
1 void generate_sizes(benchmark::internal::Benchmark *b) {
2     for (long haystack = MIN_HAYSTACK_SIZE; haystack <= MAX_HAYSTACK_SIZE;
3          haystack *= 100) {
4         for (auto needle :
5              {haystack / 8, haystack / 2, haystack - 1, haystack + 1}) {
6             b->Args({needle, haystack});
7         }
8     }
}
```

没有使用 Range 和 rangemultiplier，而是写了一个循环来生成数组，这次每次增加 100。说到查找值，在数组的相应位置使用三个位置，还有一个在外面。在每次循环迭代中调用 Args，传递两个生成的值。

现在，将生成器函数应用于基准测试：

```

1 BENCHMARK_CAPTURE(search_in_sorted_vector, binary,
2 lower_bound)->Apply(generate_sizes);
3 BENCHMARK_CAPTURE(search_in_sorted_vector, linear,
4 find)->Apply(generate_sizes);

```

使用这样的函数，可以很容易地将相同的生成器传递给许多基准测试。这些基准的结果如下：

```

-----
Benchmark Time CPU Iterations
-----
search_in_sorted_vector/binary/125/1000 20.0 ns 20.1 ns 37333333
search_in_sorted_vector/binary/500/1000 19.3 ns 19.0 ns 34461538
search_in_sorted_vector/binary/999/1000 20.1 ns 19.9 ns 34461538
search_in_sorted_vector/binary/1001/1000 18.1 ns 18.0 ns 40727273
search_in_sorted_vector/binary/12500/100000 35.0 ns 34.5 ns 20363636
search_in_sorted_vector/binary/50000/100000 28.9 ns 28.9 ns 24888889
search_in_sorted_vector/binary/99999/100000 31.0 ns 31.1 ns 23578947
search_in_sorted_vector/binary/100001/100000 29.1 ns 29.2 ns 23578947
// et cetera

```

现在有一个非常明确的实验来执行搜索。作为练习，请在自己的机器上运行这个实验，以查看完整的结果，并尝试从结果中得出一些结论。

选择微基准测试和优化

进行这样的实验可能具有意义，甚至会让人上瘾。但请记住，微基准测试不应该是项目中唯一的性能测试类型。Donald Knuth 有句名言：

*We should forget about small efficiencies, say about 97% of the time: premature optimization is
the root of all evil*
(不成熟的优化是罪恶之源)

应该只对重要的代码进行微基准测试，特别是热路径上的代码。可以使用较大的基准测试，以及跟踪和分析来查看何时何地进行优化，而不是过早地猜测和优化。首先，了解软件是如何执行的。

Note

关于上面的引用，还有一点想说明，但不要过早的悲观。糟糕的数据结构或算法选择，甚至分散的所有低效代码，有时会影响系统的整体性能。例如，执行不必要的动态分配，虽然一开始看起来没有那么糟糕，但随着时间的推移，可能会导致堆碎片，如果应用程序要运行很长一段时间，会带来严重的麻烦。过度使用基于节点的容器也会导致更多的缓存失败。长话短说，如果编写高效的代码更容易，那就去写吧。

现在，若项目有一些地方需要随着时间的推移保持良好的性能，该做些什么呢？

创建基准性能测试

与使用单元测试进行精确测试和使用功能测试进行大规模的代码正确性测试类似，可以使用微基准测试和大型基准测试来测试代码的性能。

如果对某些代码路径的执行时间有严格的限制，进行测试以确保满足限制是非常有用的。即使没有这样特定的约束，也可能对监视代码变更时性能的变化感兴趣。如果在更改之后，代码运行速度比以前慢了一个数量级，那么测试可能标记为失败。

尽管这也是一种有用的工具，但请记住，这样的测试很容易产生沸腾青蛙效应：随着时间的推移缓慢降低性能可能不会注意到，所以一定要偶尔监控执行时间。向 CI 引入性能测试时，请确保在相同的环境中运行，以获得稳定的结果。

现在，讨论下一种性能工具。

11.2.4 分析

虽然基准测试和跟踪可以提供一个概述和给定范围的特定数字，但分析器可以分析这些数字的来源。如果需要观察自己的表现并改进它，那它们是必不可少的工具。

选择分析器

有两种类型的分析器：仪表分析器和采样分析器。其中一个较为知名的仪器分析器是 Valgrind 中的 Callgrind。采样分析器的开销很大，需要插装代码，以查看调用了哪些函数，以及每个函数占用了多少开销。这样，生成的结果甚至包含最小的函数，但是执行时间可能会因这种开销而扭曲。它还有一个缺点是不能总是捕捉输入/输出 (I/O) 低速和抖动。它们会降低执行速度，因此虽然可以告知调用某个函数的频率，但不会说明，慢是因为要等待磁盘读取完成。

由于仪表分析器的缺陷，通常更好的方法是使用采样分析器。值得一提的是 Linux 系统上的开源性能分析和 Intel 的专有工具 VTune(开源项目免费使用)。尽管由于抽样的性质，有时会错过关键事件，但通常可以更好地了解代码花费时间的地方。

如果决定使用 perf，应该知道可以通过 `perf stat` 来使用它，它可以提供像 CPU 缓存使用情况这样的统计信息的快速概览，或者通过调用 `perf record -g` 和 `perf report -g` 来捕获和分析分析结果。

如果想对 perf 有一个全面的了解，请观看 Chandler Carruth 的视频，视频中展示了该工具的各种可能性以及如何使用它，或者看一看它的教程。两者在扩展阅读部分都有链接。

分析器和处理结果

分析分析结果时，可能经常想要执行一些准备、清理和处理。若代码大部分时间都在空转，那么可能需要将其过滤掉。启动分析器之前，一定要编译或保留尽可能多的调试符号，包括代码、依赖项、甚至 OS 库和内核的调试符号。另外，禁用帧指针优化也是必要的。在 GCC 和 Clang 上，可以通过`-fno-omit-frame-pointer` 标志来实现这一点。它不会对性能造成太大的影响，但会提供更多关于代码执行的数据。当涉及到结果的后期处理时，使用 perf 时，根据结果创建火焰图通常是一个好主意。Brendan Gregg 在扩展阅读部分的工具就不错。火焰图是一种简单而有效的工具，可以查看执行在哪里花费了太多的时间，因为图上每个项目的宽度对应于资源使用情况。可以有 CPU 使用情况的火焰图，以及内存使用情况、分配和页面故障等资源的火焰图，或者代码不执行时所花费的时间，例如在系统调用期间保持阻塞、互斥锁、I/O 操作等。也有方法在生成的火焰图上执行的差异。

分析结果

记住，并不是所有的性能问题都会显示在这样的图上，也不是所有的问题都可以使用分析器找到。虽然有一定经验的开发者可以看到，可以通过设置线程的亲和性或更改在特定 NUMA 节点上执行的线程而获益，但是忘记禁用省电特性或启用或禁用超线程可能并不总是那么明显，关于正在运行的硬件的信息也很有用。有时可能会看到正在使用 CPU 的 SIMD 寄存器，但代码仍然不能全速运行：可能使用 SSE 指令，而不是 AVX 指令，使用 AVX 而不是 AVX2，或使用 AVX2 而不是 AVX512。在分析分析结果时，了解 CPU 能够运行哪些特定指令也非常重要。

解决性能问题也需要一些经验。另一方面，有时经验会产生错误的假设。例如，使用动态多态性会损害性能。某些情况下，它不会降低代码的速度。在得出结论之前，有必要对代码进行分析，了解编译器优化代码的各种方法以及这些技术的局限性。具体地说到虚拟化，当不想让其他类型分别继承和覆盖虚成员函数的类时，标记为 `final` 通常是有益的。这在很多情况下对编译器都有帮助。

如果编译器可以“看到”对象的类型，就可以更好地优化：如果在作用域中创建一个类型并调用它的虚成员函数，编译器应该能够推断应该调用哪个函数。GCC 倾向于比其他编译器更好地进行反虚拟化。关于这方面的更多信息，可以参考 Arthur O’ dwyer 在扩展阅读部分的博客文章。

与本节介绍的其他类型的工具一样，不要只依赖于分析器。分析结果的改进并不能保证系统变得更快。再好看的测试报告，也不能说明事情的全部。而且一个部件的性能提高，并不意味着整个系统的性能提高。

接下来，是最后一种工具。

11.2.5 跟踪

本节中我们将讨论的最后一种技术适用于分布式系统。当查看整个系统（通常部署在云中）时，在一台机器上分析软件并不能说明全部情况。在这样的范围内，最好的方法是跟踪流经系统的请求和响应。

跟踪是记录代码执行情况的一种方法。当请求（有时是它的响应）必须流经系统的许多部分时，通常使用它。消息是沿着路线跟踪的，可以在感兴趣的执行点添加时间戳。

相关 ID

对时间戳的一个常见添加是相关 ID。基本上，是分配给每个跟踪消息的惟一标识符。目的是关联系统中不同组件（如不同的微服务）在处理相同的传入请求期间产生的日志，有时也关联它导致的事件。这样的 ID 应该随消息传递到任何地方，例如通过附加到其 HTTP 头。即使原始请求已经消失，也可以将其关联 ID 添加到生成的每个响应中。

通过使用相关 ID，可以跟踪给定请求的消息如何在系统中传播，以及系统的不同部分处理它所花费的时间。通常，需要在整个过程中收集额外的数据，比如用于执行计算的线程、为给定请求产生的响应的类型和数量，或者所经过的机器的名称。

像 Jaeger 和 Zipkin（或其他 OpenTracing 替代工具）这样的工具可以快速地向系统添加跟踪支持。

下面，让我们讨论另一个话题，从而对生成代码进行讨论。

11.3. 让编译器生成高性能代码

有很多东西可以帮助编译器生成高效的代码。有些方法可以归结为正确地操纵它，有些则需要以编译器友好的方式编写代码。

要知道在关键路径上需要做什么，并有效地设计它。例如，尽量避免在那里进行虚拟分派（除非您能证明它正在去虚拟化），并尽量不要在其上分配新的内存。通常，避免锁定（或至少使用无锁算法）的代码设计很有用。一般来说，所有可能使性能恶化的内容都应该排除在热路径之外。让指令和数据缓存都处于热状态会付出巨大的代价。甚至像 `[[likely]]` 和 `[[unlikely]]` 这样的属性提示编译器应该执行哪个分支，有时也会产生很大的变化。

11.3.1 优化整个项目

提高 C++ 项目性能的一种的方法是启用链接时间优化 (LTO)。在编译期间，编译器不知道代码将如何与其他目标文件或库链接。许多优化的机会只出现在链接时，工具可以看到程序各部分如何相互交互的宏观场景。启用 LTO，有时可以以很少的成本获得性能上的显著改进。在 CMake 项目中，可以通过设置全局 `CMAKE_INTERPROCEDURAL_OPTIMIZATION` 标志或在目标上设置 `INTERPROCEDURAL_OPTIMIZATION` 属性来启用 LTO。

使用 LTO 的一个缺点是，它使构建过程更长。有时会特别久。为了降低开发人员的成本，可以只对经过性能测试或打算发布的构建版本启用这种优化。

11.3.2 使用模式进行优化

另一种优化代码的方法是使用 Profile-Guided Optimization(PGO)，这个优化有两个步骤。第一步，需要使用标志来编译代码，这些标志会导致可执行文件在运行时收集特殊的分析信息，然后在预期的生产负载下执行。完成后，可以使用收集到的数据第二次编译可执行文件，这次传递一个不同的标志，指示编译器使用收集到的数据来生成更适合您的配置文件的代码。这样，将得到一个准备好，并针对特定工作负载进行调优的二进制文件。

11.3.3 缓存友好的代码

这两种类型的优化都很有用，但在处理高性能系统时，还有一件更重要的事情需要记住：缓存友好性。使用连续数据结构，而不是基于节点的数据结构，在运行时需要执行更少的指针跟踪，这有助于提高性能。使用内存中连续的数据，无论是向前读还是向后读，都意味着 CPU 的内存预取器可以在使用前加载，这通常会产生巨大的差异。基于节点的数据结构和前面提到的指针追逐会导致随机的内存访问模式，这可能会把预取器搞懵，使其无法预取正确的数据。

如果想查看一些性能结果，请参阅扩展阅读部分链接的 C++ 容器基准测试。比较了 `std::vector`、`std::list`、`std::deque` 和 `plf::colony` 的各种使用场景。有些读者可能不了解最后一个，它是一个有趣的“包”式容器，可以快速地插入和删除大量数据。

选择关联容器时，通常希望使用“连续”实现，而不是基于节点的实现。所以，除了使用 `std::unordered_map` 和 `std::unordered_set`，可以尝试 `tsl::hopscotch_map` 或 Abseil 的 `flat_hash_map` 和 `flat_hash_set`。

将较冷的指令（如异常处理代码）放入非内联函数中的方式，可以提高指令缓存的热度。这样，用于处理罕见情况的冗长代码就不会加载到指令缓存中，从而为应该存在的更多代码留出空间，这也提高性能。

11.3.3 设计代码和数据

如果想对缓存友好，有一种技术可以使用，就是面向数据的设计。通常，将经常使用的成员存储在内存中相邻的位置是个好主意。较冷的数据通常可以放在另一个结构中，通过 ID 或指针与较热的数据连接。

有时，使用数组的对象可以获得更好的性能，而不是更常见的对象数组。不是以面向对象的方式编写代码，而是将对象的数据成员拆分为几个数组，每个数组包含多个对象的数据。以下的代码为例：

```
1 struct Widget {
2     Foo foo;
3     Bar bar;
4     Baz baz;
5 };
6
7 auto widgets = std::vector<Widget>{};
```

考虑将其替换为以下内容：

```
1 struct Widgets {
2     std::vector<Foo> foos;
3     std::vector<Bar> bars;
4     std::vector<Baz> bazs;
5 };
```

这样，当针对某些对象处理一组特定的数据点时，缓存热度会提高，性能也会提高。如果不知道这是否会产生更多的性能收益，可以测试一下。

有时，即使对类型的成员进行重新排序也可以获得更好的性能。可以考虑数据成员类型的对齐。如果性能很重要，通常最好对它们进行排序，这样编译器就不需要在成员之间插入太多的填

充。由于这一点，数据类型的大小可以更小，因此可以将许多这样的对象放入一条缓存线中。考虑下面的例子（假设为 x86_64 架构编译）：

```
1 struct TwoSizesAndTwoChars {
2     std::size_t first_size;
3     char first_char;
4     std::size_t second_size;
5     char second_char;
6 };
7 static_assert(sizeof(TwoSizesAndTwoChars) == 32);
```

尽管每个大小是 8 个字节，每个字符只有 1 个字节，最终得到了 32 个字节！这是因为 `second_size` 必须从一个 8 字节对齐的地址开始，所以在 `first_char` 之后，得到 7 个字节的填充。对于 `second_char` 也是如此，因为类型需要与最大的数据类型成员对齐。

能做得更好吗？试着交换成员的顺序：

```
1 struct TwoSizesAndTwoChars {
2     std::size_t first_size;
3     std::size_t second_size;
4     char first_char;
5     char second_char;
6 };
7 static_assert(sizeof(TwoSizesAndTwoChars) == 24);
```

通过简单地将最大的成员放在前面，能够将结构的大小减少 8 个字节，这是其大小的 25%。对于这样一个微不足道的变化来说还不错，若目标是将许多这样的结构打包在一个连续的内存块中并遍历，那么可以看到该代码段的性能大大提高。

现在来看看另一种提高性能的方法。

11.4. 并行计算

本节中，将讨论几种并行计算的不同方法。将从比较线程和进程开始，之后将向展示 C++ 标准中可用的工具，最后，将简单介绍 OpenMP 和 MPI 框架。

开始之前，简单介绍一下如何估计代码并行化可能带来的最大收益，这里有两条规律可以给予我们帮助。首先是阿姆达尔定律，如果想通过添加更多的核心来加快程序的速度，那么代码中必须保持顺序（不能并行化）的部分将限制可扩展性。例如，如果代码有 90% 是可并行的，那么即使核数无限，仍然只能获得 10 倍的加速。即使将 90% 的执行时间减少到零，10% 的代码仍然会留在那里。

第二定律是古斯塔夫森定律，每个足够大的任务都可以并行化。通过增加问题的规模，可以获得更好的并行化（假设有免费的计算资源可以使用）。换句话说，有时添加更多的功能以在同一时间段内运行，而不是试图减少现有代码的执行时间，这是更好的做法。如果可以通过加倍核心来减少一半的任务时间，在某些情况下，一次次地加倍会让性能回报递减，所以处理能力可以花在其他地方。

11.4.1 理解线程和进程的区别

为了有效地并行化计算，还需要了解何时使用进程来执行计算，以及何时线程是这项工作的更好工具。长话短说，如果目标是实际并行工作，那么最好从添加额外的线程开始。此时，在网络中的其他机器上添加更多进程，每个机器也有多个线程。

这是为什么呢？因为进程比线程更重量级。生成一个进程并在它们之间切换所花费的时间，比创建线程和在线程之间切换要长。每个进程需要自己的内存空间，而同一个进程中的线程可以共享内存。而且，进程间通信比线程之间传递变量要慢。使用线程比使用进程更容易，开发速度也会更快。

然而，进程在单个应用程序的范围内也有其用途。非常适合隔离那些可以独立运行和崩溃的组件，而不会破坏整个应用程序。拥有独立的内存还意味着一个进程不能窥探另一个进程的内存，当运行可能是恶意的第三方代码时，这是非常好。这两个原因就是为什么它们用于网络浏览器和其他应用程序。除此之外，还可以运行具有不同操作系统权限或特权的不同进程，这是多线程无法实现的。

现在讨论在单个机器范围内并行工作的一种简单方法。

11.4.2 标准的并行算法

如果执行的计算可以并行化，有两种方法可以使用。一种是用可并行的算法替换对标准库算法的常规调用。如果不熟悉并行算法，它们是在 C++17 中添加，本质上是相同的算法，但可以为传递一个执行策略。有三种执行策略：

- `std::execution::seq`: 以非并行方式执行算法的顺序策略。
- `std::execution::par`: 一种并行策略，表明执行可能并行化，通常在后台使用线程池。
- `std::execution::par_unseq`: 一种并行策略，表示执行可以并行化和向量化。
- `std::execution::unseq`: C++20 的新成员。该策略表示可以向量化执行，但不能并行化执行。

如果上述策略都还不够，那么标准库实现可能会提供其他策略。可能的未来添加可能包括 CUDA，SyCL，OpenCL，甚至人工智能处理器的支持。

现在来看看并行算法的实际应用。要对一个数组进行并行排序，可以这样写：

```
1 std::sort(std::execution::par, v.begin(), v.end());
```

简单且容易！尽管在许多情况下，这将产生更好的性能，但在某些情况下，以传统的方式执行算法可能会更好。为什么呢？因为在更多的线程上调度工作需要额外的工作和同步。此外，根据应用程序的架构，可能会影响其他已经存在的线程的性能，并刷新核心数据缓存。这里同样也需要先进行测试。

11.4.3 使用 OpenMP 和 MPI 并行计算

使用标准并行算法的另一种选择是利用 OpenMP 的实用程序，是一种简单的方法，只需添加几行代码，就可以并行处理许多类型的计算。如果想在集群中分发代码，可能想了解 MPI 能做些什么。这两者也可以连接在一起。

使用 OpenMP，可以使用各种 pragmas 来轻松地并行化代码。例如，可以在 for 循环之前添加`#pragma omp parallel for`，以便使用并行线程执行它。这个库可以做更多的事情，比如在 GPU 和其他加速器上执行计算。

将 MPI 集成到项目中比添加适当的 pragma 更困难。这里，需要在代码库中使用 MPI API 来发送或接收进程之间的数据（使用如 MPI_Send 和 MPI_Recv 的调用），或执行各种收集和减少操作（调用 MPI_Bcast 和 MPI_Reduce）。通信可以使用通信器对象进行点对点，或对所有集群的通讯。

根据算法实现的不同，MPI 节点可以全部执行相同的代码，也可以根据需要不同。节点将根据排名知道应该如何行为：计算开始时分配的唯一数字。说到这里，使用 MPI 启动一个进程，应该通过一个包装器运行它，像这样：

```
$ mpirun --hostfile my_hostfile -np 4 my_command --with some ./args
```

这将从上述文件逐一读取主机，连接到每个主机，并在每个主机上运行 `my_command` 的四个实例，并传递参数。

MPI 有很多实现。其中最值得注意的是 OpenMPI（不要将其与 OpenMP 混淆），一些特性提供了容错功能。毕竟，节点宕机的情况并不少见。

本节中，最后一个工具是 GNU Parallel，如果想通过生成并行进程来轻松地跨越执行工作的进程，就会发现它很有用。它既可以在一台机器上使用，也可以跨计算集群使用。

说到执行代码的不同方式，那就要说一下 C++20 中的另一个大主题：协程。

11.5. 协程

协程是可以暂停执行并在稍后继续执行的函数，可以以与编写同步代码非常相似的方式编写异步代码。与使用 `std::async` 编写异步代码相比，这允许编写更清晰、更容易理解和维护的代码。不需要再写回调函数，也不需要用 `promise` 和 `future` 来处理 `std::async`。

除此之外，还可以为提供更好的性能。基于 `std::async` 的代码通常在切换线程和等待方面有更多的开销。与调用函数的开销相比，协程可以非常快捷地恢复和挂起，这意味着可以产生更好的延迟和吞吐量。此外，其设计目标是具有高度的可扩展性，甚至可以扩展到数十亿个并发协程。

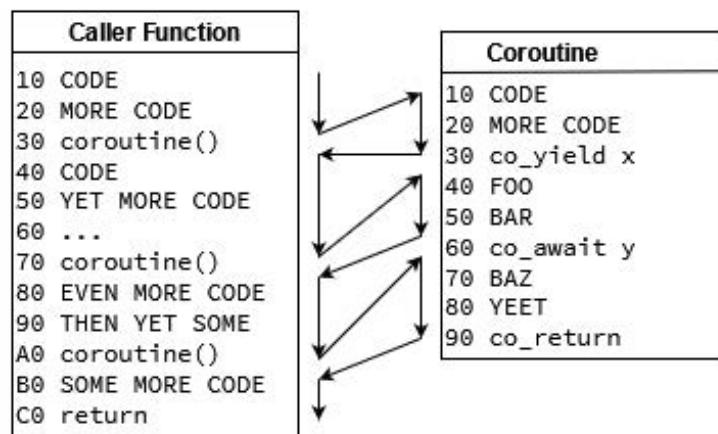


图 11.1 - 调用和执行协程与使用常规函数不同，因为它们可以挂起和恢复

C++协程无堆栈，它们的状态不存储在调用线程的堆栈上。这给了它们一个有趣的属性：几个不同的线程可以执行一个协程。换句话说，尽管协程函数体看起来是按顺序执行的，但某些部分可以在不同的线程中执行。这使得将函数的一部分留给专用线程来执行成为可能。例如，I/O 操作可以在专用的 I/O 线程中完成。

要检查函数是否为 C++ 协程，需要在函数体中查找以下关键字：

- `co_await`, 暂停协程。
- `co_yield` 用于向调用方返回值，并暂停协程。
- 类似于 Python 在生成器中使用的 `yield` 关键字，允许惰性生成值。
- `co_return`, 返回一个值并完成协程的执行，与 `return` 关键字等价。

只要函数体中有其中一个关键字，函数就会自动变成协程。虽然这是一个实现细节，但还有一个提示：协程返回类型必须满足某些要求（后面再说）。

协程是 C++ 世界中的一等公民，可以获取它们的地址，用作函数参数，从函数返回它们，并将它们存储在对象中。

在 C++ 中，可以在 C++20 之前编写协程。这要感谢 Boost 等库。Coroutine2，也就是彭博的 Quantum。后者用来实现 CoroKafka——使用协程高效处理 Kafka 流的库。随着标准 C++ 协程的出现，新的库开始出现。现在，将向展示其中一个。

11.5.1 与 `cppcoro` 的区别

从头编写基于协程的代码很困难。C++20 只提供了编写协程的基本实用程序，所以在编写自己的协程时需要一组原语。Lewis Baker 创建的 `cppcoro` 库是 C++ 中最常用的协程框架之一。本节中，将展示这个库，并演示如何在编写基于协程的代码时使用它。

先来概述一下库提供的协程类型：

- `task<>`: 为了调度工作稍后执行——当 `co_await` 开始执行。
- `shared_task<>`: 多个协程可以等待一个任务。可以复制，以便多个协程引用相同的结果。本身没有提供任何线程安全性。
- `generator`: 缓慢而同步地产生一个 T 序列。它实际上是 `std::range`，它有一个返回迭代器的 `begin()` 和一个返回哨兵的 `end()`。
- `recursive_generator`: 类似于 `generator<T>`，但可以生成 T 或 `recursive_generator<T>`。这里会有一些额外的开销。
- `async_generator`: 类似于 `generator<T>`，但是可以异步生成值。这意味着，与生成器相反，异步
- 生成器可以在它们的函数体内使用 `co_await`。

应该使用这些类型作为协程的返回类型。通常，在生成器中（协程返回前面的生成器类型），可以使用 `co_yield` 的返回值（类似于 Python 生成器）。然而，在任务中，通常需要使用 `co_await` 来安排工作。

与前面的协程类型相比，这个库实际上提供了更多的编程抽象。还提供以下类型：

- `co_await` 的可等待类型，比如协程事件和同步原语：互斥锁、锁存、栅栏等。
- 可取消，可以取消协程的执行。

- 调度器——可以调度工作的对象，例如 `static_thread_pool`，或在特定线程上调度工作。
- I/O 和网络工具，允许读写文件和 IP 套接字。
- 元函数和概念，比如 `awaitable_traits`、`awaitable` 和 `Awaiter`。

除了前面的实用程序之外，`cppcoro` 还提供了一些函数——用于使用其他类和指导执行的程序，例如：

- `sync_wait`: 阻塞式，直到传递的可等待对象完成工作。
- `when_all`, `when_all_ready`: 返回一个可等待对象，该对象在传递的所有可等待对象完成时完成。这两者之间的区别在于处理子可等待对象的失败。即使在失败的情况下，`when_all_ready` 将完成，调用者可以检查每个结果，而 `when_all` 将重新抛出一个异常，如果任何子可等待对象抛出一个异常（虽然不知道是哪个），其还会取消所有未完成的任务。
- `fmap`: 与函数式编程类似，将函数应用于可等待对象。可以将其视为将一种类型的任务转换为另一种类型的任务。例如，可以通过调用 `fmap(serialize, my_coroutine())` 来序列化协程返回的类型。
- `resume_on`: 指示协程在完成某些工作后，使用哪个调度器继续执行。这能够在特定的执行上下文中执行特定的工作，例如在专用的 I/O 线程上运行与 I/O 相关的任务。注意，这意味着一个 C++ 函数（协程）可以在单独的线程上执行它的各个部分。可以使用类似于 `std::ranges` 的计算“通道”。
- `schedule_on`: 指示协程使用哪个调度器来启动某些工作。通常用法 `auto foo = co_await schedule_on(scheduler, do_work());`。

在开始使用这些实用程序之前，再了解一下可等待程序。

11.5.2 剖析可等待程序和协程

除了 `cppcoro` 之外，标准库还提供了两个可等待对象：`suspend_never` 和 `suspend_always`。通过了解它们，可以看到如何在需要时实现自己的可等待对象：

```

1 struct suspend_never {
2     constexpr bool await_ready() const noexcept { return true; }
3     constexpr void await_suspend(coroutine_handle<>) const noexcept {}
4     constexpr void await_resume() const noexcept {}
5 };
6
7 struct suspend_always {
8     constexpr bool await_ready() const noexcept { return false; }
9     constexpr void await_suspend(coroutine_handle<>) const noexcept {}
10    constexpr void await_resume() const noexcept {}
11};

```

当输入 `co_await` 时，告诉编译器首先调用等待者的 `await_ready()`。如果返回 `true` 表示已经准备好了，`await_resume()` 将调用。`await_resume()` 的返回类型应该是等待者实际生成的类型。如果没有准备好，程序将执行 `await_suspend()`。完成后，有三种情况：

- `await_suspend` 返回 `void`: 执行之后总是挂起。
- `await_suspend` 返回 `bool`: 执行是否挂起取决于返回值。

- `await_suspend` 返回 `std::coroutine_handle<PromiseType>`: 另一个协程将恢复运行。

协程的背后还有更多的东西。即使协程不使用 `return` 关键字，编译器也会在后台生成代码，使它们能够编译和工作。当使用诸如 `co_yield` 这样的关键字时，将把它们重写为对协助类型的适当成员函数的调用。例如，对 `co_yield x` 的调用等价于 `co_await promise.yield_value(x)`。如果想进一步了解协程更多信息，并编写自己的协程类型，请参阅扩展阅读部分的《第一个协程》。

好了，让我们用这些知识来写自己的协程吧。我们将创建一个简单的应用程序来模拟有意义的工作，这里现来使用一个线程池来用一些数字填充向量。

CMake 目标如下：

```
add_executable(coroutines_1 coroutines/main_1.cpp)
target_link_libraries(coroutines_1 PRIVATE cppcoro fmt::fmt
Threads::Threads)
target_compile_features(coroutines_1 PRIVATE cxx_std_20)
```

将链接到 `cppcoro` 库。在例子中，使用的是 Andreas Buhr 的 `cppcoro` 分支，因为它是 Lewis Baker 仓库的一个维护良好的分支，并且支持 CMake。

还将链接到用于文本格式化的优秀{fmt}库。如果标准库提供了 C++20 的字符串格式，可以使用标准库的格式化方法。

最后，需要一个线程库——希望在池中使用多个线程。

以一些常量和一个 `main` 函数开始实现：

```
1 inline constexpr auto WORK_ITEMS = 5;
2
3 int main() {
4     auto thread_pool = cppcoro::static_thread_pool{3};
```

希望使用三个池中的线程生成五个元素，`cppcoro` 的线程池以一种简单的方法对工作进行安排。默认情况下，创建的线程数量与机器的硬件线程数量相同。接下来，需要具体说明相应的工作：

```
1 fmt::print("Thread {}: preparing work\n", std::this_thread::get_id());
2 auto work = do_routine_work(thread_pool);
3
4 fmt::print("Thread {}: starting work\n", std::this_thread::get_id());
5 const auto ints = cppcoro::sync_wait(work);
```

在代码中散布日志消息，这样就可以更好地看到在哪个线程中发生了什么。可以通过调用名为 `do_routine_work` 的协程来创建工作。返回协程，使用 `sync_wait` 阻塞函数运行协程，协程只有在实际待时才会执行。这样，实际工作的将在这个函数调用中开始。

当运行有了结果，就可以把结果记录下来：

```
1 fmt::print("Thread {}: work done. Produced ints are: ",
2           std::this_thread::get_id());
3 for (auto i : ints) {
4     fmt::print("{} ", i);
5 }
6 fmt::print("\n");
```

定义我们的 `do_routine_work` 协程:

```
1 #include <coroutine>
2 #include <vector>
3 #include <mutex>
4 #include <chrono>
5
6 #include <fmt/format.h>
```

它返回一个任务，该任务生成一些整数。因为使用线程池，所以让使用 `cppcoro` 的 `async_mutex` 来同步线程。现在来使用池:

```
1 #include <coroutine>
2 #include <vector>
3 #include <mutex>
4 #include <chrono>
5
6 #include <fmt/format.h>
```

`schedule()` 调用没有传入可调用对象来执行，在协程的情况下，实际上当前线程暂停协程。并开始执行它的调用程序。现在将等待协程完成（在 `sync_wait` 调用的地方）。

与此同时，池中的一个线程将恢复协程——继续执行它的任务。以下是为此所做的准备:

```
1 #include <coroutine>
2 #include <vector>
3 #include <mutex>
4 #include <chrono>
5
6 #include <fmt/format.h>
```

我们创建一个要执行的任务集，每个任务在互斥锁下填充一个整数。`schedule_on` 调用池中的另一个线程运行填充协程。最后，等待所有的结果。此时，任务开始执行。最后，由于协程是一个任务，这里使用 `co_return`。

Note

不要忘记 `co_return` 生成的值。如果移除 `co_return int;`，将简单地返回一个默认构造的数组。程序将运行，打印空数组，并以代码 0 退出。

最后一步是实现产生数字的协程:

```
1 #include <coroutine>
2 #include <vector>
3 #include <mutex>
4 #include <chrono>
5
6 #include <fmt/format.h>
```

这个任务不返回任何值。相反，会把它加到数组上。繁重工作实际上是通过休眠几毫秒来完成的。睡醒后，协作程序将继续进行更有成效的工作:

```
1 {
2     auto lock = co_await mutex.scoped_lock_async();
```

```
3     ints.emplace_back(i);
4 }
```

它将锁定互斥对象。其实，只是等待而已。当互斥锁锁定时，会给 `vector` 添加一个数字——与调用时相同的数字。

Note

还记得 `co_await` 么？如果记不清了，可以不使用可等待对象（可能因为不使用每个可等待对象也是可以的），那么就会跳过一些必要的计算。在例子中，这可能代表着不锁定互斥对象。

现在来完成协程的实现：

```
1 fmt::print("Thread {}: produced {}\n", std::this_thread::get_id(), i);
2 co_return;
```

一个简单的状态打印和一个 `co_return` 来标记协程完成。返回时，协程框架就可以销毁了，释放其占用的内存。

这是所有了！现在运行代码，看看会发生什么：

```
Thread 140471890347840: preparing work
Thread 140471890347840: starting work
Thread 140471890347840: passing execution to the pool
Thread 140471890282240: running first pooled job
Thread 140471890282240: producing 4
Thread 140471881828096: producing 1
Thread 140471873373952: producing 0
Thread 140471890282240: produced 4
Thread 140471890282240: producing 3
Thread 140471890282240: produced 3
Thread 140471890282240: producing 2
Thread 140471881828096: produced 1
Thread 140471873373952: produced 0
Thread 140471890282240: produced 2
Thread 140471890347840: work done. Produced ints are: 4, 3, 1, 0, 2,
```

主线程用于启动池上的工作，然后等待结果。然后，池中的三个线程产生数字。最后安排的任务实际上是第一个运行的任务，产生数字 4。这是因为它一直在执行 `do_routine_work`：首先，调度池中的其他任务，然后在调用 `when_all_ready` 时开始执行第一个任务。稍后，执行继续进行，第一个空闲线程接受池上调度的下一个任务，直到整个数组填满。最后，执行返回到主线程。

这个例子到此结束。通过它，我们就结束了本章的最后一部分。现在来回顾一下本章了解到的内容。

11.6. 总结

本章中，了解了哪些类型的工具可以帮助代码获得更好的性能。了解了如何进行实验、编写性能测试以及寻找性能瓶颈，使用 Google Benchmark 编写微基准测试。此外，还讨论了如何分析代码，以及如何（以及为什么）实现系统的分布式跟踪。还讨论了使用标准库实用程序和外部解决方案并行计算。最后，介绍了协程。现在知道了 C++20 为协程给 C++ 带来了什么，以及可以在 `cppcoro` 库中找到什么。还了解了如何编写自己的协程。

本章最重要的是：当涉及到性能时，首先测试，然后优化。这将有助于最大限度地发挥工作的影响。

这就是性能——在书中讨论的最后一个质量属性。下一章中，将开始进入服务和云的世界。我们将从讨论面向服务的架构开始。

11.7. 练习题

1. 可以从本章的微基准测试的性能结果中学到什么？
2. 如何遍历多维数组对性能重要吗？为什么重要，或为什么不重要？
3. 在的协程例子中，为什么不能在 `do_routine_work` 函数中创建自己的线程池？
4. 如何重新编写协程示例，使其使用生成器，而不仅仅使用任务？

11.8. 扩展阅读

- When can the C++ compiler devirtualize a call?, blog post, Arthur O'Dwyer, <https://quuxplusone.github.io/blog/2021/02/15/devirtualization/>
- CppCon 2015: Chandler Carruth "Tuning C++: Benchmarks, and CPUs, and Compilers! Oh My!", YouTube video, <https://www.youtube.com/watch?v=nXaxk27zwIk>
- Tutorial, Perf Wiki, <https://perf.wiki.kernel.org/index.php/Tutorial>
- CPU Flame Graphs, Brendan Gregg, <http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html>
- C++ Containers Benchmark, blog post, Baptiste Wicht, <https://baptiste-wicht.com/posts/2017/05/cpp-containers-benchmark-vector-list-deque-plfcolony.html>
- Your First Coroutine, blog post, Dawid Pilarski, <https://blog.panicsoftware.com/your-first-coroutine>

第四部分：原生云的设计原则

介绍起源于分布式系统和云环境中的现代架构风格，展示了诸如面向服务的架构、微服务（包括容器）和各种消息传递系统等概念。

包括以下几章：

- 第 12 章，面向服务的架构
- 第 13 章，设计微服务
- 第 14 章，容器
- 第 15 章，原生云设计

第 12 章 面向服务的架构

分布式系统的一个常见架构是面向服务的架构 (SOA)。这并不是什么新发明，因为这种架构风格几乎和计算机网络一样古老。SOA 有很多方面，从企业服务总线 (ESB) 到云本地微服务。

如果应用程序包含 Web、移动或物联网 (IoT) 接口，本章将有助于理解如何以模块化和可维护性为重点构建它们。由于当前的大多数系统都以客户机-服务器 (或其他网络拓扑结构) 的方式工作，因此了解 SOA 原则将有助于设计和改良此类系统。

本章将讨论以下内容：

- 了解 SOA
- 消息传递原则
- 使用 Web 服务
- 托管服务和云提供商

12.1. 相关准备

本章给出的大多数示例都不需要任何特定的软件。对于 AWS API 的例子，需要 AWS C++ SDK，可以在 <https://aws.amazon.com/sdk-for-cpp/> 上找到。

本章的代码可以在以下 GitHub 页面找到：<https://github.com/PacktPublishing/Software-Architecture-with-Cpp/tree/master/Chapter12>。

12.2. 了解面向服务的架构

面向服务的架构是一个软件设计的例子，特点是组件松散耦合，这些组件相互提供服务。这些组件通常通过网络使用共享通信协议，服务意味着可以在原始组件外部访问的功能单元。组件的一个例子可以是映射服务，该服务根据地理坐标提供该地区的地图。

根据定义，服务有四个属性：

- 具有已定义结果的业务活动的表示。
- 自包含的。
- 对用户黑盒。
- 由其他服务组成。

12.2.1 实现方法

面向服务的架构没有规定如何实现面向服务，这个术语可以应用于许多不同的实现。对于某些方法是否应该视为面向服务的架构，存在一些讨论。我们不想参与这些讨论，只是想强调一些常称为 SOA 的方法。

企业服务总线

当有人说到面向服务的架构时，ESB 通常是第一个关联。它是实现 SOA 的最古老的方法之一。

ESB 借鉴了计算机硬件体系结构，硬件结构采用 PCI 等计算机总线实现模块化。这样，只要每个人都遵守总线要求的标准，第三方提供商就能够独立于主板制造商实现模块（如图形卡、声卡或 I/O 接口）。

与 PCI 非常相似，ESB 体系结构旨在构建一种标准的、通用的方式，以允许松散耦合的服务进行交互。这些服务预计将独立开发和部署，还可以组合为异构服务。

与 SOA 本身一样，ESB 不是由全局标准定义的。要实现 ESB，需要在系统中建立一个组件，这个组件就是总线本身。ESB 上的通信是事件驱动的，通常通过面向消息的中间件和消息队列来实现。

企业服务总线组件服务于以下角色：

- 控制服务的部署和版本控制
- 维护服务的冗余
- 服务之间的消息路由
- 监视和控制消息交换
- 解决组件之间的争用
- 提供公共服务，如事件处理、加密或消息入队
- 加强服务素质 (QOS)

实现企业服务总线功能的产品既有专有的商业产品，也有开源的产品。以下是一些最受欢迎的开源产品：

- Apache Camel
- Apache ServiceMix
- Apache Synapse
- JBoss ESB
- OpenESB
- Red Hat Fuse(基于 Apache Camel)
- Spring Integration

最受欢迎的商业产品有：

- IBM Integration Bus (已经被 IBM WebSphere ESB 替代)
- Microsoft Azure Service Bus
- Microsoft BizTalk Server
- Oracle Enterprise Service Bus
- SAP Process Integration

与本书中介绍的所有模式和产品一样，在决定使用特定的架构之前，必须考虑其优点和缺点。引入企业服务总线的一些好处如下：

- 更好的服务扩展性
- 分布式负载
- 可以专注于配置，而不是实现服务中的自定义集成
- 设计松散耦合服务的简单方法
- 服务可替换

- 内置的冗余功能

另一方面，缺点主要有以下几个方面：

- 单点故障——ESB 组件的故障意味着整个系统的中断。
- 配置比较复杂，影响维护。
- 消息队列、消息转换和 ESB 提供的其他服务可能会降低性能，甚至成为瓶颈。

Web 服务

Web 服务是面向服务架构的另一种流行实现。根据 Web 服务的定义，Web 服务是一台机器向另一台机器（或操作员）提供的服务，其中通过万维网协议进行通信。尽管万维网管理机构 W3C 允许使用其他协议，如 FTP 或 SMTP，但 Web 服务通常使用 HTTP 作为传输协议。

虽然可以使用专有的解决方案来实现 Web 服务，但大多数实现都基于开放协议和标准。尽管许多方法通常称为 Web 服务，但彼此之间有着本质上的不同。本章的后面，将详细描述各种方法。现在，让我们关注它们的共同特性。

Web 服务的优缺点

Web 服务的好处如下：

- 使用流行的 Web 标准
- 很多工具
- 可扩展性

下面是缺点：

- 开销很大
- 有些实现复杂（例如，SOAP/WSDL/UDDI 规范）

消息和流

介绍企业服务总线体系结构时，已经提到了消息队列和消息代理。除了作为 ESB 实现的一部分外，消息传递系统还可以是独立的架构元素。

消息队列

消息队列是用于进程间通信（IPC）的组件。顾名思义，它们使用队列数据结构在不同进程之间传递消息。通常，消息队列是面向消息中间件（MOM）设计的一部分。

在最低级别，消息队列在 UNIX 规范中可用，在 System V 和 POSIX 中都有。虽然在单台机器上实现 IPC 很有趣，但我们更希望关注适合于分布式计算的消息队列。

目前，开源软件中与消息队列相关的标准有三种：

1. 高级消息队列协议（AMQP），一种运行在 7 层 OSI 模型的应用层上的二进制协议。流行的实现包括以下几点：

- Apache Qpid
- Apache ActiveMQ

- RabbitMQ
 - Azure Event Hubs
 - Azure Service Bus
2. 流式面向文本的消息传递协议 (STOMP)，类似于 HTTP 的基于文本的协议 (使用诸如 CONNECT、SEND、SUBSCRIBE 等动词)。流行的实现包括以下几点：
- Apache ActiveMQ
 - RabbitMQ
 - syslog-ng
3. MQTT，一种针对嵌入式设备的轻量级协议。流行的实现包括家庭自动化解决方案，如以下：
- OpenHAB
 - Adafruit IO
 - IoT Guru
 - Node-RED
 - Home Assistant
 - Pimatic
 - AWS IoT
 - Azure IoT Hub

消息代理

消息代理处理消息系统中消息的转译、验证和路由。与消息队列一样，也是 MOM 的一部分。

通过使用消息代理，可以最大限度地减少应用程序对系统其他部分的感知，这会让系统的设计变得松散耦合，因为消息代理承担了与消息上的通用操作相关的所有负担，从而称为 PublishSubscribe (PubSub) 设计模式。

代理通常为接收者管理消息队列，也能够执行其他功能，例如：

- 将消息从一种表示转换为另一种表示
- 验证消息发送方、接收方或内容
- 将消息路由到一个或多个目的地
- 聚合、分解和重组传输中的消息
- 从外部服务检索数据
- 通过与外部服务的交互来增强和丰富消息
- 处理和响应错误和其他事件
- 提供不同的路由模式，如 PubSub

消息代理的主流实现如下：

- Apache ActiveMQ
- Apache Kafka
- Apache Qpid
- Eclipse Mosquitto MQTT Broker

- NATS
- RabbitMQ
- Redis
- AWS ActiveMQ
- AWS Kinesis
- Azure Service Bus

云计算

云计算是一个宽泛的术语，有很多不同的含义。最初，术语云指的是架构不应该太担心的抽象层。例如，这可能意味着由专门的运营团队管理服务器和网络基础设施。后来，服务提供商开始将术语云计算应用到他们自己的产品中，这些产品抽象了底层基础设施的复杂性。不必单独配置每个基础设施，而是可以使用简单的应用程序编程接口 (Application Programming Interface, API) 来设置所有必要的资源。

如今，云计算已经发展到包括许多应用程序架构的新方法，可能包括以下内容：

- 托管服务，如数据库、缓存层和消息队列
- 可扩展的工作负载编排
- 容器部署和编排平台
- 无服务的计算平台

考虑使用云时，要记住的一点是，在云中托管应用程序需要专门为云设计的架构。通常，还意味着专门为给定的云提供商设计的架构。

这意味着选择云提供商，不仅仅是在给定的时间点上决定一个选择是否比另一个更好，更换供应商的未来成本可能太大，不值得这样做。提供商之间的迁移需要更改架构，而对于一个没问题的应用程序，这些更改可能会超过迁移所节省的成本。

云架构设计的另一个后果。对于遗留应用程序，这意味着为了利用云的优势，应用程序首先必须重新构建和编写。迁移到云不仅仅是将二进制文件和配置文件从本地托管复制到云提供商管理的虚拟机。这种方法只会浪费金钱，因为只有当应用程序具有可扩展性和云感知能力时，云计算才具有成本效益。

云计算并不一定意味着使用外部服务和从第三方提供商租赁机器，还有一些解决方案，比如在本地运行的 OpenStack，可以使用已经拥有的服务器来获得云计算的红利。

我们将在本章后面讨论托管服务。容器、云本地设计和无服务器架构将在本书后面的章节中具体描述。

微服务

关于微服务是否是 SOA 的一部分存在一些争论。大多数时候，术语 SOA 几乎等同于 ESB 设计。微服务在许多方面与 ESB 相反。这导致人们认为微服务是不同于 SOA 的一种模式，是软件架构发展的下一步。

我们相信它们是一种现代的 SOA 方法，旨在消除 ESB 中出现的一些问题。毕竟，微服务非常符合面向服务架构的定义。

12.2.2 面向服务架构的优点

将系统的功能拆分到多个服务上有几个好处。首先，每个服务都可以单独维护和部署。这有助于团队专注于给定的任务，而不需要理解系统内的每一个可能的交互，其还支持敏捷开发，因为测试只需要覆盖特定的服务，而不是整个系统。

第二个好处是服务的模块化有助于创建分布式系统。使用网络（通常基于 Internet 协议）作为通信手段，服务可以在不同的机器之间分割，以提供扩展性、冗余和更好的资源使用。

当每个服务都有许多生产者和消费者时，实现新特性和维护现有软件是一项困难的任务。这就是 SOA 鼓励使用文档化和版本化 API 的原因。

另一种使服务生产者和消费者更容易交互的方法是，使用描述如何在不同服务之间传递数据和元数据的既定协议。这些协议可能包括 SOAP、REST 或 gRPC。

API 和标准协议的使用使得创建新服务变得很容易，这些新服务提供了比现有服务更高的价值。考虑到我们有一个服务 A，它返回地理位置，另一个服务 B，它提供给定位置的当前温度，可以调用 A 并在向 B 的请求中使用它的响应。这样，就可以获得当前位置的当前温度，而不用自己实现整个逻辑。

这两个服务的所有复杂性和实现细节对外都是未知的，我们将它们视为黑盒。这两个服务的维护者也可能引入新的功能并发布服务的新版本，但不需要对外通知。

使用面向服务的架构进行测试和试验也比使用单块应用程序更容易，小更改不需要重新编译整个代码库。通常可以使用客户端工具以特定的方式调用服务。

回到天气和地理位置服务的示例。如果两个服务都公开一个 REST API，就能够构建一个原型，只使用 cURL 客户机手动发送适当的请求。当确认响应是令人满意的时，可以开始编写代码，使整个操作自动化，并可能将结果作为另一个服务公开。

Note

要获得 SOA 的好处，需要记住所有服务都必须是松散耦合的。如果服务依赖于彼此的实现，这意味着它们不再是松散耦合的，而是紧密耦合的。理想情况下，给定的服务都可以被不同的类似服务替代，而不会影响整个系统的运行。

在天气和位置的例子中，这意味着用另一种语言重新实现位置服务（比如从 Go 切换到 C++）应该不会影响到该服务的下游使用者，只要他们使用已建立的 API 即可。

通过发布新的 API 版本，仍然有可能在 API 中引入破坏性的变化。连接到 1.0 版本的客户机将观察遗留行为，而连接到 2.0 版本的客户机将受益于 bug 修复、更好的性能和其他以兼容性为代价的改进。

对于依赖于 HTTP 的服务，API 版本控制通常发生在 URI 级别。因此，当调用 `URLhttps://service.local/v1/customer` 时可以访问版本 1.0、1.1 或 1.2 的 API，而版本 2.0 的 API 位于 `https://service.local/v2/customer`。然后，API 网关、HTTP 代理或负载均衡器能够将请求切换到适当的服务。

12.2.3 SOA 的挑战

引入抽象层总是要付出代价的。同样的规则也适用于面向服务的架构。在查看企业服务总线、Web 服务或消息队列和代理时，很容易看到抽象成本。可能不那么明显的是，微服务也是有代价

的。其成本与它们使用的远程过程调用 (Remote Procedure Call, RPC) 框架有关，以及与服务冗余和功能重复相关的资源消耗。

与 SOA 相关的另一个批评是缺乏统一的测试框架，开发应用程序服务的单个团队可能会使用其他团队不知道的工具。与测试相关的其他问题是，组件的异构性质和互换性意味着需要测试大量的组合。一些组合可能会引入通常观察不到的边界情况。

由于关于特定服务的知识主要集中在单个团队中，因此很难理解整个应用程序如何工作。

应用程序的生命周期中开发 SOA 平台时，可能需要所有服务更新其版本，以针对最近的平台开发。在平台发生变化后，开发人员将专注于确保应用程序功能正确，而不是引入新特性。极端的情况下，维护成本可能会急剧上升，因为那些服务不知道出现了新版本，并不断打补丁以满足平台的需求。

面向服务的体系结构遵循 Conway 的法则，在第 2 章中有介绍过。

12.3. 消息传递原则

消息传递有许多不同的用例，从物联网和传感器网络到运行在云中基于微服务的分布式应用程序。

消息传递的好处之一是，连接使用不同技术实现服务的中立。在开发 SOA 时，每个服务通常由专门的团队开发和维护。团队可以选择他们觉得舒服的工具，这适用于编程语言、第三方库和构建系统。

维护统一的工具集可能会适得其反，因为不同的服务可能有不同的需求。kiosk 应用程序可能需要图形用户界面 (GUI) 库，例如 Qt。作为同一应用程序一部分的硬件控制器可能有其他需求，可能链接到硬件制造商的第三方组件。这些依赖可能会施加一些不能同时满足两个组件的限制（例如，GUI 应用程序可能需要一个最新的编译器，而硬件对应的可能固定在一个旧的编译器上），使用消息传递系统来解耦这些组件使它们具有独立的生命周期。

消息传递系统的一些用例包括：

- 金融业务
- 车辆监控
- 物流捕获
- 处理传感器
- 执行数据订单
- 任务队列

以下部分将重点介绍为低开销设计的消息传递系统，以及用于分布式的代理的消息系统。

12.3.1 低开销的消息传递系统

低开销的消息传递系统通常用于占用空间小或延迟低的环境中，这些通常是传感器网络、嵌入式解决方案和物联网设备。它们在基于云的分布式服务中不太常见，但也可以在此类解决方案中使用它们。

MQTT

MQTT 表示消息队列遥测传输。它是 OASIS 和 ISO 下的开放标准。MQTT 通常在 TCP/IP 上使用 PubSub 模型，也可以与其他传输协议一起工作。

顾名思义，MQTT 的设计目标是低代码占用和在低带宽位置运行的可能性。有一个独立的规范称为 MQTT-SN，代表传感器网络的 MQTT，专注于没有 TCP/IP 栈的电池供电的嵌入式设备。

MQTT 使用消息代理接收来自客户机的所有消息，并将这些消息路由到目的地。服务质素分为三个级别：

- 最多发一次（不保证交付）
- 至少发一次（确认交付）
- 一次发送（保证交付）

MQTT 在各种物联网应用程序中特别受欢迎，这并不奇怪。它由 OpenHAB、Node-RED、Pimatic、Microsoft Azure IoT Hub 和 Amazon IoT 支持。在即时通讯中也很流行，在 ejabberd 和 Facebook messenger 中使用。其他用例包括汽车共享平台、物流和运输。

支持该标准的两个最流行的 C++ 库是基于 C++ 14 和 Boost.Asio 的 Eclipse Paho 和 mqtt_cpp。对于 Qt 应用程序，还有 qmqtt。

ZeroMQ

ZeroMQ 是一个无代理消息队列。它支持常见的消息传递模式，如 PubSub、客户机/服务器和其他几种模式。它独立于特定的传输，可以与 TCP、WebSockets 或 IPC 一起使用。

其主要思想包含在名称中，即 ZeroMQ 不需要任何代理和管理。还提倡提供零延迟，这意味着代理的存在不会增加延迟。

底层库是用 C 编写的，它有各种流行编程语言的实现，包括 C++。主流 C++ 实现是 cppzmq，是一个使用 C++11 的头文件库。

12.3.2 代理消息传递系统

主流的两种不关注低开销的消息系统是基于 AMQP 的 RabbitMQ 和 Apache Kafka。两者都是成熟的解决方案，在许多不同的设计中都非常受欢迎。很多文章关注 RabbitMQ 或 Apache Kafka 在某一特定领域的优势。

这是一个稍微不正确的观点，因为两个消息传递系统都基于不同的范例。Apache Kafka 专注于将大量数据流化，并将流存储在持久内存中，以允许未来重用。另一方面，RabbitMQ 经常用作不同微服务之间的消息代理或处理后台任务的任务队列。因为这个原因，RabbitMQ 中的路由比 Apache Kafka 中的路由要先进得多。Kafka 的主要用来进行数据分析和实时处理。

而 RabbitMQ 使用 AMQP 协议（也支持其他协议，如 MQTT 和 STOMP），Kafka 使用自己的基于 TCP/IP 的协议。RabbitMQ 可以与基于这些受支持协议的其他现有解决方案互操作。如果写了一个使用 AMQP 与 RabbitMQ 交互的应用程序，应该可以将其迁移到 Apache Qpid, Apache ActiveMQ，或者 AWS 或 Microsoft Azure 的托管解决方案。

扩展方面的考虑也可能促使选择一个消息代理而不是另一个。Apache Kafka 的架构允许水平扩展，可以向现有的工作池添加更多的机器。另一方面，RabbitMQ 在设计时考虑了垂直扩展，这意味着可以向现有机器添加更多的资源，而不是添加更多机器。

12.4. 使用 Web 服务

Web 服务的共同特征是基于标准的 Web 技术。大多数时候，这将意味着超文本传输协议 (HTTP)，这是本节重点关注的技术。尽管可以基于不同的协议实现 Web 服务，但这样的服务非常罕见，超出了本书的范围。

12.4.1 调试 Web 服务的工具

使用 HTTP 作为传输的主要好处之一是工具的广泛可用性。通常，测试和调试 Web 服务只需要使用 Web 浏览器即可。除此之外，还有很多其他的程序可能对自动化很有帮助。其中包括：

- 标准的 Unix 文件下载程序 wget
- 现代 HTTP 客户端 curl
- 流行的开源库，如 libcurl、curlpp、C++ REST SDK、cpr(C++ HTTP 请求库) 和 NFHTTP
- 测试框架，如 Selenium 或 Robot Framework
- 浏览器扩展，如 Boomerang
- 独立的解决方案，如 Postman 和 Postwoman
- 专用测试软件包括 SoapUI 和 Katalon Studio

基于 HTTP 的 Web 服务通过向使用适当 HTTP 动词 (如 GET、POST 和 PUT) 的 HTTP 请求返回 HTTP 响应来工作。请求和响应，以及传递数据的语义在不同的实现中也是不同的。

大多数实现分为两类：基于 XML 的 Web 服务和基于 JSON 的 Web 服务。目前，基于 JSON 的 Web 服务正在取代基于 XML 的 Web 服务，但使用 XML 格式的服务仍然很常见。

对于处理用 JSON 或 XML 编码的数据，可能需要额外的工具，如 xmllint、xmlstarlet、jq 和 libxml2。

12.4.2 基于 XML 的 Web 服务

第一个获得关注的 Web 服务主要基于 XML 的。XML 或可扩展标记语言在当时是分布式计算和网络环境中交换格式的选择。使用 XML 有效负载设计服务有几种不同的方法。

可能希望与组织内部或外部开发的基于 XML 的现有 Web 服务进行交互。但建议使用更轻量级的方法来实现新的 Web 服务，比如基于 JSON 的 Web 服务、REST 的 Web 服务或 gRPC。

XML-RPC

出现的第一个标准被称为 XML-RPC。该项目背后的想法是提供一种 RPC 技术，可以与当时流行的公共对象模型 (COM) 和 CORBA 竞争。其目的是将 HTTP 作为一种传输协议，并使这种格式既可由人类读/写，也可由机器解析。为此，选择 XML 作为数据编码格式。

当使用 XML-RPC 时，希望执行远程过程调用的客户机向服务器发送 HTTP 请求。请求可以有多个参数，服务器用一个响应响应。XML-RPC 协议为参数和结果定义了几种数据类型。

尽管 SOAP 具有类似的数据类型，但它使用 XML 模式定义，这使得消息的可读性比 XML-RPC 低得多。

与 SOAP 的关系

由于 XML-RPC 不再积极地维护，因此标准没有现代 C++ 实现。如果想从现代代码中与 XML-RPC Web 服务交互，最好的方法可能是使用支持 XML-RPC 和其他基于 XML Web 服务标准的 gSOAP 工具包。

对 XML-RPC 的批评是，与发送普通的 XML 请求和响应相比，它没有提供太多的价值，同时使消息明显变多了。

随着标准的发展，它变成了 SOAP。作为 SOAP，它形成了 W3C Web 服务协议栈的基础。

SOAP

SOAP 最初的缩写是简单对象访问协议 (Simple Object Access Protocol)。该缩写在标准的 1.2 版中删除了，它是 XML-RPC 标准的扩展。

SOAP 由三部分组成：

- SOAP 包络，定义消息的结构和处理规则
- SOAP 头规则定义特定于应用程序的数据类型 (可选)
- SOAP 主体，它携带远程过程调用和响应

下面是一个使用 HTTP 作为传输的 SOAP 消息示例：

```
POST /FindMerchants HTTP/1.1
Host: www.domifair.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 345
SOAPAction: "http://www.w3.org/2003/05/soap-envelope"

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
  </soap:Header>
  <soap:Body xmlns:m="https://www.domifair.org">
    <m:FindMerchants>
      <m:Lat>54.350989</m:Lat>
      <m:Long>18.6548168</m:Long>
      <m:Distance>200</m:Distance>
    </m:FindMerchants>
  </soap:Body>
</soap:Envelope>
```

该示例使用标准 HTTP 头和 POST 方法调用远程过程，SOAP 唯一的一个报头是 SOAPAction。它指向一个标识操作意图的 URI，由客户机决定如何解释这个 URI。

`soap:Header` 是可选的，所以为空。它与 `soap:Body` 一起包含在 `soap:Envelope` 中，主要调用发生在 `soap:Body` 中。我们引入了自己的 XML 名称空间，这是多米尼加博览会应用程序特有的。

命名空间指向域的根，调用 FindMerchants，并提供三个参数：纬度、纬度和距离。

由于 SOAP 设计为可扩展、传输中立和独立于编程模型，导致了其他伴随标准的创建。从而在使用 SOAP 之前，通常有必要了解所有相关的标准和协议。

如果应用程序大量使用 XML，并且开发团队熟悉所有术语和规范，这自然不是问题。然而，如果想要的只是为第三方公开 API，更简单的方法是构建 REST API，因为它对开发者和使用者来说学习成本更低。

WSDL

Web 服务描述语言 (Web Services Description Language, WSDL) 提供了关于如何调用服务以及应该如何形成消息的机器可读描述。与其他 W3C Web 服务标准一样，它是用 XML 编码的。

经常与 SOAP 一起使用来定义 Web 服务提供的接口。

当用 WSDL 定义了 API，就可以（而且应该！）使用自动化工具在此基础上创建代码。对于 C++ 来说，gSOAP 就是带有此类工具的框架。它附带了一个名为 wsdl2h 的工具，该工具将根据定义生成一个头文件。然后，可以使用另一个工具 soapcpp2 生成从接口定义到实现的绑定。

不幸的是，由于消息的冗长，SOAP 服务的大小和带宽需求通常非常大。如果这不是问题，那么 SOAP 可以使用。同时允许同步和异步调用，以及有状态和无状态操作。如果需要严格的、正式的通信方式，SOAP 也可以提供，只要确保使用协议的 1.2 版本，因为它引入了许多改进（其中之一就是增强了服务的安全性）。另一个优点是改进了服务本身的定义，这有助于互操作性，或者能够正式定义传输方式（允许使用消息队列），等等。

UDDI

记录 Web 服务接口之后的下一步是发现服务，它允许应用程序查找并连接到由其他方实现的服务。

统一描述、发现和集成 (UDDI) 是可以手动或自动搜索的 WSDL 文件的注册中心。与本节讨论的其他技术一样，UDDI 使用 XML 格式。

可以用 SOAP 消息查询 UDDI 注册中心，以便自动发现服务。尽管 UDDI 提供了 WSDL 的逻辑扩展，但公开版本仍然令人失望。仍然有可能找到公司内部使用的 UDDI 系统。

SOAP 库

两个主流的 SOAP 库是 Apache Axis 和 gSOAP。

Apache Axis 适合实现 SOAP（包括 WSDL）和 REST Web 服务。值得注意的是，该库已经有十多年没有发布过新版本了。

gSOAP 是一个工具包，它允许创建基于 XML 的 Web 服务并与之交互，重点关注于 SOAP。它处理数据绑定、SOAP 和 WSDL 支持、JSON 和 RSS 解析、UDDI API 以及其他一些相关的 Web 服务标准。虽然它没有使用现代的 C++，但是它仍然积极地维护着。

12.4.3 基于 JSON 的 Web 服务

JSON 代表 JavaScript 对象表示法。与它的名字相反，它并不局限于 JavaScript（其是独立于语言的）。JSON 的解析器和序列化器存在于大多数编程语言中，JSON 比 XML 紧凑得多。

其语法来源于 JavaScript，因为基于 JavaScript 子集。

JSON 支持的数据类型如下：

- 数字: 具体的格式可能在不同的实现中有所不同; 默认为 JavaScript 中的双精度浮点。
- 字符串:unicode 编码的。
- 布尔值: 使用 true 和 false 值。
- 数组: 可能为空。
- 对象: 具有键-值对的映射。
- null: 表示空值。

第 9 章中介绍的 Packer 配置是 JSON 文档的一个例子:

```
{  
  "variables": {  
    "aws_access_key": "",  
    "aws_secret_key": ""  
  },  
  "builders": [ {  
    "type": "amazon-ebs",  
    "access_key": "{{user `aws_access_key`}}",  
    "secret_key": "{{user `aws_secret_key`}}",  
    "region": "eu-central-1",  
    "source_ami": "ami-5900cc36",  
    "instance_type": "t2.micro",  
    "ssh_username": "admin",  
    "ami_name": "Project's Base Image {{timestamp}}"  
  ] ,  
  "provisioners": [ {  
    "type": "ansible",  
    "playbook_file": "./provision.yml",  
    "user": "admin",  
    "host_alias": "baseimage"  
  ] ,
```

```
"post-processors": [{}  
    "type": "manifest",  
    "output": "manifest.json",  
    "strip_path": true  
}]  
}
```

使用 JSON 作为格式的标准之一是 JSON-RPC 协议。

JSON-RPC

JSON-RPC 是一种 JSON 编码的远程过程调用协议，类似于 XML-RPC 和 SOAP。与 XML 不同，它需要很少的开销。在维护 XML-RPC 的人类可读性的同时，它也是非常简单的。

这就是前面用 JSON-RPC 2.0 在 SOAP 调用中表示的示例：

```
{  
    "jsonrpc": "2.0",  
    "method": "FindMerchants",  
    "params": {  
        "lat": "54.350989",  
        "long": "18.6548168",  
        "distance": 200  
    },  
    "id": 1  
}
```

这个 JSON 文档仍然需要适当的 HTTP 报头，但即使有了报头，仍然比对应的 XML 文档小得多。唯一存在的元数据是带有 JSON-RPC 版本和请求 ID 的文件，方法和参数字段几乎是自解释的，不过 SOAP 并不总是如此。

尽管该协议是轻量级的，易于实现和使用，但与 SOAP 和 REST Web 服务相比，还没有广泛采用。它发布的时间比 SOAP 晚得多，与 REST 服务开始流行的时间差不多。REST 很快就获得了成功（可能是由于它的灵活性），而 JSON-RPC 却未能获得类似的成功。

两个有用的 C++ 实现是 libjson-rpc-cpp 和 json-rpc-cxx，json-rpc-cxx 是以前库的现代重置版。

12.4.4 表达性状态转移 (REST)

Web 服务的另一种方法是表示状态传输 (representational State Transfer, REST)。符合这种架构风格的服务通常称为 REST 式服务。REST 与 SOAP 或 JSON-RPC 之间的主要区别在于 REST 几乎完全基于 HTTP 和 URI 语义。

REST 是在实现 Web 服务时定义一组约束的架构样式，符合这种风格的服务称为 RESTful。其限制如下：

- 必须使用客户机-服务器模型。
- 无状态 (客户机和服务器都不需要存储与其通信相关的信息)。
- 可缓存性 (响应应该定义为可缓存或不可缓存，以受益于标准的 Web 缓存来提高可扩展性和性能)。
- 分层系统 (代理和负载均衡器不应该影响客户端和服务器之间的通信)。

REST 使用 HTTP 作为传输协议，RUI 代表资源和操作资源或调用操作的 HTTP 动词。没有关于每个 HTTP 方法应该如何行为的标准，但最常达成一致的语义如下：

- POST – 创建一个新的资源。
- GET – 检索现有资源。
- PATCH – 更新现有资源。
- DELETE – 删除已有资源。
- PUT – 替换已有资源。

由于对 Web 标准的依赖，RESTful Web 服务可以重用现有的组件，如代理、负载均衡器和缓存。由于开销低，这些服务的性能和效率也非常高。

描述语言

就像基于 XML 的 Web 服务一样，可以用机器和人类可读的方式描述 RESTful 服务。目前有一些相互竞争的标准，OpenAPI 是最受欢迎的。

OpenAPI

OpenAPI 是由 OpenAPI 计划 (Linux 基金会的一部分) 监督的规范。它曾经被称为 Swagger 规范，因为它曾经是 Swagger 框架的一部分。

该规范与语言无关。它使用 JSON 或 YAML 输入来生成方法、参数和模型的文档。通过这种方式，使用 OpenAPI 有助于保持文档和源代码的更新。

有很多与 OpenAPI 兼容的工具可供选择，比如代码生成器、编辑器、用户界面和模拟服务器。OpenAPI 生成器可以使用 cpp-restsdk 或 Qt 5 为客户端实现生成 C++ 代码。还可以使用 Pistache，Restbed 或 Qt 5 QHTTPEngine 生成服务器代码。在线上还有一个方便的 OpenAPI 编辑器：<https://editor.swagger.io/>。

使用 OpenAPI 文档化的 API 会像下面这样：

```
{
  "openapi": "3.0.0",
  "info": {
    "title": "Items API overview",
    "version": "2.0.0"
  },
  "paths": {
    "/item/{itemId)": {
      "get": {
        "operationId": "getItem",
        "summary": "get item details",
        "parameters": [
          {
            "name": "itemId",
            "description": "Item ID",
            "required": true,
            "schema": {
              "type": "string"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "content": {
              "application/json": {
                "example": {
                  "itemId": 8,
                  "name": "Kürtőskalács",
                  "locationId": 5
                }
              }
            }
          }
        }
      }
    }
  }
}
```

前两个字段 (`openapi` 和 `info`) 是描述文档的元数据。`paths` 字段包含与 REST 接口的资源和方法对应的所有可能路径。前面的示例中，只记录了单个路径 (/项目) 和单个方法 (GET)。此方法将 `itemId` 作为必需参数。提供了一个可能的响应码，即 200。200 响应包含一个 JSON 文档本身的主体。与示例键相关联的值是成功响应的示例负载。

RAML

另一个与之竞争的规范是 RAML，它代表 RESTful API 建模语言。它使用 YAML 进行描述，并支持发现、代码重用和模式共享。

建立 RAML 的基本原理是，虽然 OpenAPI 是一个记录现有 API 的伟大工具，但在当时，它并不是设计新 API 的最佳方式。目前，该规范正考虑成为 OpenAPI 计划的一部分。

可以将 RAML 文档转换为 OpenAPI 来使用可用的工具。

下面是一个使用 RAML 编写的 API 示例：

```
#%RAML 1.0

title: Items API overview
version: 2.0.0

annotationTypes:
  oas-summary:
    type: string
    allowedTargets: Method

/item:
  get:
    displayName: getItem
    queryParameters:
      itemId:
        type: string
    responses:
      '200':
        body:
          application/json:
            example: |
              {
                "itemId": 8,
                "name": "Kürtőskalács",
```

```
        "locationId": 5
    }
  description: 200 response
  (oas-summary): get item details
```

这个例子描述了与 OpenAPI 相同的接口。在 YAML 中序列化时，OpenAPI 3.0 和 RAML 2.0 看起来非常相似。主要的区别是 OpenAPI 3.0 要求使用 JSON 模式来记录结构。使用 RAML 2.0，可以重用现有的 XML 模式定义 (XSD)，这使得从基于 XML 的 Web 服务迁移或包含外部资源变得更容易。

API 蓝图

API 蓝图提供了与前面两个规范不同的方法，不依赖 JSON 或 YAML，而是使用 Markdown 来记录数据结构和端点。

方法类似于测试驱动开发方法，因为它鼓励在实现特性之前设计契约。通过这种方式，测试实现是否真正履行了契约就更容易了。

就像使用 RAML 一样，将 API 蓝图规范转换为 OpenAPI 也是可能的，反之亦然。还有一个命令行接口和一个用于解析 API 蓝图的 C++ 库，称为 Drafter，可以在代码中使用它。

一个用 API 蓝图记录的简单 API 示例：

```
FORMAT: 1A

# Items API overview

# /item/{itemId}

## GET

+ Response 200 (application/json)

{[
  "itemId": 8,
  "name": "Kürtőskalács",
  "locationId": 5
}]
```

前面，看到指向 /item 端点的 GET 方法应该会导致响应代码 200，下面是对应于我们的服务通常返回的 JSON 消息。

API 蓝图允许更多自然的文档，其主要缺点是它是目前所描述的格式中最不受欢迎的，所以其

文档和工具的质量都远不及 OpenAPI。

RSDL

与 WSDL 类似，RSDL(或 RESTful 服务描述语言) 是 Web 服务的 XML 描述。它是独立于语言的，并且设计成人人可读的。

API 蓝图或 RAML 相比，比之前提供的替代方案要少得多，也更难阅读。

超媒体作为应用状态引擎

尽管提供二进制接口 (比如基于 gRPC 的接口) 可以提供很好的性能，但在许多情况下，仍然希望拥有 RESTful 接口的简单性。如果想要直观的基于 REST 的 API，那么超媒体作为应用程序状态引擎 (HATEOAS) 是一个有用的原则。

就像打开一个网页并基于显示的超媒体进行导航一样，可以用 HATEOAS 编写服务来实现同样的事情。这促进了服务器和客户端代码的分离，并允许客户端快速知道发送哪些请求是有效的，而这通常不是二进制 API 的情况。该发现是动态的，基于所提供的超媒体。

如果采用典型的 RESTful 服务，在执行操作时，会得到包含对象状态等数据的 JSON。除此之外，使用 HATEOAS 将获得一个链接 (URL) 列表，显示可以在该对象上运行的有效操作，链接 (超媒体) 是应用程序的引擎。换句话说，可用的操作是由资源的状态决定的。虽然超媒体这个术语在这里听起来很奇怪，但它基本上是指链接资源，包括文本、图像和视频。

例如，如果有一个 REST 方法可以通过使用 PUT 方法添加项，可以添加一个返回参数，该参数链接到以这种方式创建的资源。如果使用 JSON 进行序列化，可能会出现以下形式：

```
{  
  "itemId": 8,  
  "name": "Kürtőskalács",  
  "locationId": 5,  
  "links": [  
    {  
      "href": "item/8",  
      "rel": "items",  
      "type" : "GET"  
    }  
  ]  
}
```

目前还没有普遍接受的序列化 HATEOAS 超媒体的方法。一方面，不管服务器实现是什么，都更容易实现。另一方面，客户端需要知道如何解析响应，以找到相关的遍历数据。

HATEOAS 的一个好处是，使在服务器端实现 API 更改成为可能，而不必破坏客户端代码。当其中一个端点重命名时，新的端点将在后续响应中引用，因此客户机将被告知将进一步请求定向到哪里。

同样的机制可以提供分页等特性，或者使发现给定对象可用的方法变得容易。回到我们的示例，下面是在发出 GET 请求后可能收到的响应：

```
{  
  "itemId": 8,  
  "name": "Kürtőskalács",  
  "locationId": 5,  
  "stock": 8,  
  "links": [  
    {  
      "href": "item/8",  
      "rel": "items",  
      "type" : "GET"  
    },  
    {  
      "href": "item/8",  
      "rel": "items",  
      "type" : "POST"  
    },  
    {  
      "href": "item/8/increaseStock",  
      "rel": "increaseStock",  
      "type" : "POST"  
    },  
    {  
      "href": "item/8/decreaseStock",  
      "rel": "decreaseStock",  
      "type" : "POST"  
    }  
  ]  
}
```

这里，获得了两个负责修改股票的方法的链接。如果股票不再可用，响应将是这样的（注意其中一个方法不再发布）：

```
{  
    "itemId": 8,  
    "name": "Kürtőskalács",  
    "locationId": 5,  
    "stock": 0,  
    "links": [  
        {  
            "href": "items/8",  
            "rel": "items",  
            "type": "GET"  
        },  
        {  
            "href": "items/8",  
            "rel": "items",  
            "type": "POST"  
        },  
        {  
            "href": "items/8/increaseStock",  
            "rel": "increaseStock",  
            "type": "POST"  
        }  
    ]  
}
```

与 HATEOAS 相关的一个重要问题是，这两个设计原则似乎彼此不一致。如果总是以相同的格式呈现，添加可遍历的超媒体将更容易使用。这里的表达自由使得编写不知道服务器实现的客户机变得更加困难。

并不是所有 RESTful API 都能从引入这一原则中受益——通过引入 HATEOAS，将承诺以特定的方式编写客户机，从而使它们能够从这种 API 风格中受益。

C++ 实现的 REST

Microsoft 的 C++ REST SDK 是目前在 C++ 应用程序中实现 RESTful API 的最佳方法之一。它也称为 `cpp-restsdk`，在本书中使用它来演示各种示例。

12.4.5 GraphQL

REST Web 服务的最新替代方案是 GraphQL，名称中的 QL 代表查询语言。在 GraphQL 中，客户机不依赖服务器序列化和呈现必要的数据，而是直接查询和操作数据。除了职责颠倒之外，GraphQL 还具有使数据处理更容易的机制。类型、静态验证、内省和模式都是规范的一部分。

GraphQL 的服务器实现可用于很多语言，包括 C++。其中一个主流的实现是来自 Microsoft 的 `cppgraphqlgen`。还有许多工具可以帮助开发和调试。有趣的是，由于 Hasura 或 PostGraphile 等产品在 Postgres 数据库上添加了 GraphQL API，就可以直接使用 GraphQL 查询数据库。

12.5. 托管服务和云提供商

面向服务的架构可以扩展到当前的云计算趋势。虽然企业服务总线提供的服务通常是内部开发的，但通过云计算，可以使用一个或多个云提供商提供的服务。

为云计算设计应用程序架构时，在实现替代方案之前，应该始终考虑提供商提供的托管服务。例如，在决定用选定的插件托管自己的 PostgreSQL 数据库之前，请确保了解与提供商提供的托管数据库相比的利弊和成本。

当前的云环境提供了很多服务，旨在处理如下流行的用例：

- 存储
- 关系数据库
- 文档 (NoSQL) 数据库
- 内存缓存
- 电子邮件
- 消息队列
- 容器编排器
- 计算机视觉
- 自然语言处理
- 语音合成和语音识别
- 监视、日志记录和跟踪
- 大数据
- 网络加速器
- 数据分析
- 任务管理与调度
- 身份管理
- 密钥和机密管理

由于有大量可用的第三方服务可供选择，云计算非常适合面向服务的架构。

12.5.1 云计算是 SOA 的扩展

云计算是虚拟机托管的扩展。云计算提供商与传统 VPS 提供商的区别在于两点：

- 云计算可以通过 API 使用，这使它本身成为一种服务。
- 除了虚拟机实例之外，云计算还提供其他服务，如存储、托管数据库、可编程网络等。所有这些都可以通过 API 获得。

有几种方法可以使用云提供商的 API 在应用程序中提供特性，现在将介绍这些方法。

直接使用 API

如果云提供商提供了以选择的语言访问的 API，就可以直接在应用程序中与云资源交互。

例如：有一个允许用户上传自己照片的应用程序。这个应用程序使用云 API 为每个新注册的用户创建一个存储桶：

```
1 #include <aws/core/Aws.h>
2 #include <aws/s3/S3Client.h>
3 #include <aws/s3/model/CreateBucketRequest.h>
4
5 #include <spdlog/spdlog.h>
6
7 const Aws::S3::Model::BucketLocationConstraint region =
8     Aws::S3::Model::BucketLocationConstraint::eu_central_1;
9 bool create_user_bucket(const std::string &username) {
10     Aws::S3::Model::CreateBucketRequest request;
11
12     Aws::String bucket_name("userbucket_" + username);
13     request.SetBucket(bucket_name);
14
15     Aws::S3::Model::CreateBucketConfiguration bucket_config;
16     bucket_config.SetLocationConstraint(region);
17     request.SetCreateBucketConfiguration(bucket_config);
18
19     Aws::S3::S3Client s3_client;
20     auto outcome = s3_client.CreateBucket(request);
21
22     if (!outcome.IsSuccess()) {
23         auto err = outcome.GetError();
24         spdlog::error("ERROR: CreateBucket: {}:{}",
25             err.GetExceptionName(),
26             err.GetMessage());
27         return false;
28     }
29     return true;
30 }
```

本例中，有一个 C++ 函数，创建了一个 AWS S3 bucket，该 bucket 以参数中提供的用户名命名。此桶配置为保存在给定区域中。如果操作失败，希望获得错误消息，并使用 spdlog 记录。

通过 CLI 工具使用 API

操作不必在应用程序运行时执行。通常在部署期间运行，因此可以在 shell 脚本中自动运行。使用 CLI 工具创建 VPC 的场景如下：

```
gcloud compute networks create database --description "A VPC to access the data
```

使用来自 Google 云平台的 gcloud CLI 工具创建一个名为数据库的网络，将用于处理从私有实例到数据库的通信。

与云 API 交互的第三方工具

看一个运行 HashiCorp Packer 来构建一个虚拟机实例镜像的例子，它已经预先配置好了应用程序：

```
{  
  variables : {  
    do_api_token : {{env `DIGITALOCEAN_ACCESS_TOKEN`}} ,  
    region : fra1 ,  
    packages : "customer"  
    version : 1.0.3  
  },  
  builders : [  
    {  
      type : digitalocean ,  
      api_token : {{user `do_api_token`}} ,  
      image : ubuntu-20-04-x64 ,  
      region : {{user `region`}} ,  
      size : 512mb ,  
      ssh_username : root  
    }  
  ],  
  provisioners: [  
    {  
      type : file ,  
      source : ./{{user `package`}}-{{user `version`}}.deb ,  
      destination : /home/ubuntu/  
    },  
    {  
      type : shell ,  
      inline : [  
        dpkg -i /home/ubuntu/{{user `package`}}-{{user `version`}}.deb  
      ]  
    }  
  ]  
}
```

前面的代码中，提供了所需的凭证和区域，并雇佣了一个构建器来从 Ubuntu 映像为我们准备一个实例。我们感兴趣的实例需要有 512MB RAM。然后，通过发送一个.deb 包给它来提供实例，

然后通过执行 shell 命令来安装这个包。

访问云 API

通过 API 访问云计算资源是区别于传统托管的最重要特性之一。使用 API 意味着可以随意创建和删除实例，而不需要操作符的干预。这样，就很容易实现一些特性，比如基于负载的自动扩展、高级部署 (Canary 版本或 Blue-Green 版本) 以及应用程序的自动化开发和测试环境。

云提供商通常将其 API 公开为 RESTful 服务。除此之外，还经常为几种编程语言提供客户端库。虽然三家最流行的提供商都支持 C++ 作为客户端库，但来自较小供应商的支持可能有所不同。

如果正在考虑将 C++ 应用程序部署到云上并计划使用云 API，请确保提供商已经发布了一个 C++ 软件开发工具包 (SDK)。在没有官方 SDK 的情况下仍然可以使用 Cloud API，例如使用 CPP REST SDK 库。但请注意，后续将需要更多的工作。

要访问 Cloud SDK，还需要访问控制。通常，有两种方法可以验证你的应用程序使用云 API：

- 通过提供 API 令牌

API 令牌应该是保密的，永远不要作为版本控制系统的一部分或在编译后的二进制文件中存储。为了防止被盗，也应该加密。

将 API 令牌安全地传递给应用程序的方法之一是通过 HashiCorp Vault 这样的安全框架，是可编程的秘密存储内置租赁时间管理和密钥旋转。

- 通过托管在具有适当访问权限的实例上

许多云提供商允许为特定的虚拟机实例提供访问权限。这样，保存在此类实例上的应用程序就不必使用单独的令牌进行身份验证。然后，访问控制将基于产生云 API 请求的实例。

这种方法更容易实现，因为它不需要考虑秘密管理的需要。缺点是，当实例受到破坏时，访问权限将对运行在那里的所有应用程序可用，而不仅仅是部署的应用程序。

使用云命令行

人工操作人员通常使用 Cloud CLI 与 Cloud API 进行交互。另外，也可以用于编写脚本，或者使用带有官方不支持的语言的 Cloud API。

例如，Bourne Shell 脚本在 Microsoft Azure 云上创建一个资源组，然后创建一个属于该资源组的虚拟机：

```
#!/bin/sh
RESOURCE_GROUP=dominiccanfair
VM_NAME=dominic
REGION=germanynorth

az group create --name $RESOURCE_GROUP --location $REGION

az vm create --resource-group $RESOURCE_GROUP --name $VM_NAME --image
UbuntuLTS --ssh-key-values dominic_key.pub
```

查找关于如何管理云资源的文档时，将遇到许多使用 Cloud CLI 的示例。即使通常不会使用 CLI，而是更喜欢像 Terraform 这样的解决方案，手边有一个 Cloud CLI 也可以调试环境问题。

与云 API 交互的工具

已经了解了使用云提供商的产品时锁定供应商的危险。通常，每个云提供商将为所有其他云提供商提供不同的 API 和不同的 CLI。某些情况下，较小的提供商提供抽象层，允许通过类似于知名提供商的 API 访问它们的产品。这种方法旨在帮助将应用程序从一个平台迁移到另一个平台。

不过，这种情况很少发生，而且通常用于与来自一个提供者的服务交互的工具与来自另一个提供者的服务不兼容，这不仅仅是在考虑从一个平台迁移到另一个平台时才会出现的问题。如果希望将应用程序托管在各种平台上，这也可能会有问题。

为此，出现了一组新的工具，统称为基础架构代码 (IaC) 工具，在不同的平台之上提供了一个抽象层。这些工具也不一定仅限于云提供商。它们是通用的，有助于将应用程序自动化架构在不同层上。

在第 9 章时，简要介绍了其中的一些。

12.5.2 原生云架构

新工具允许架构师和开发人员进一步抽象环境，并首先在构建时考虑到云。Kubernetes 和 OpenShift 等流行的解决方案正在推动这一趋势，但这个领域也包含许多体量较小的参与者。本书的最后一章专门介绍了云原生设计，并描述了这种构建应用程序的现代方法。

12.6. 总结

本章中，了解了实现面向服务架构的不同方法。由于服务可能以不同的方式与其环境交互，因此有许多架构模式可供选择。我们已经了解了最受欢迎的方法的优点和缺点。

关注了一些广泛流行的方法的架构和实现方面：消息队列、包括 REST 的 Web 服务，以及使用托管服务和云平台。将在后续独立的章节中更深入地探讨其他方法，比如微服务和容器。

下一章中，来了解一下微服务。

12.7. 练习题

1. 面向服务的架构中，服务的属性是什么？
2. Web 服务有哪些好处？
3. 什么时候微服务不是一个好的选择？
4. 消息队列用来做什么？
5. 选择 JSON 而不是 XML 有哪些好处？
6. REST 是如何建立在 Web 标准之上的？
7. 云平台与传统托管有何不同？

12.8. 扩展阅读

- SOA Made Simple: <https://www.packtpub.com/product/soa-made-simple/9781849684163>
- SOA Cookbook: <https://www.packtpub.com/product/soa-cookbook/9781847195487>

第 13 章 设计微服务

随着微服务的日益普及，我们想用本书的整整一章来献给它。讨论架构时，可能会听到：“我们应该为此使用微服务吗？”本章将展示如何将现有的应用程序迁移到微服务架构，以及如何构建一个利用微服务的新应用程序。

本章将讨论以下内容：

- 深入了解微服务
- 构建微服务
- 观察微服务
- 连接微服务
- 扩展微服务

13.1. 相关准备

本章给出的大多数示例都不需要任何特定的软件。对于 redis-cpp 库，可以在这里看到相关信息 <https://github.com/tdv/redis-cpp>。

本章的代码可以在以下 GitHub 页面找到：<https://github.com/PacktPublishing/Software-Architecture-with-Cpp/tree/master/Chapter13>。

13.2. 深入了解微服务

虽然微服务不依赖于特定的编程语言或技术，但实现微服务的一个常见选择是 Go。这并不意味着其他语言不适合微服务的开发，C++ 的低计算和内存开销使其成为微服务的理想选择。

但首先，将详细介绍微服务的一些优点和缺点。之后，将重点关注与微服务相关的设计模式（与第 4 章中涉及的一般设计模式相反）。

13.2.1 微服务的好处

可能经常听到“微服务”的高级说法，其确实能带来一些好处，以下是其中一些。

模块化

由于整个应用程序可分割成许多相对较小的模块，所以更容易理解每个微服务的功能。这种理解的自然结果是，测试单个微服务也更容易。每个微服务通常都有一个有限的范围，这一事实也有助于测试。毕竟，只测试日历应用程序要比测试整个 Personal Information Management(PIM) 套件容易得多。

然而，这种模块化是有代价的。开发团队可能对单个微服务有更好的理解，但与此同时，会发现很难理解整个应用程序是如何组成的。虽然不需要了解构成应用程序的微服务的内部细节，但组件之间的关系数量之多对认知构成了挑战。在使用这种架构方法时，应用微服务契约是一个良好的实践。

扩展性

扩展范围有限的应用程序更容易，这样做的一个原因是瓶颈更少。

扩展更小的工作流也更划算。想象一个负责管理展会的应用程序，当系统开始出现性能问题，实现规模化的唯一方法就是引入更大的机器来运行整个系统，这叫做垂直扩展。

使用微服务的第一个优势是可以横向扩展，可以引入更多的机器，而不是更大的机器（通常更便宜）。第二个优势来自于这样一个事实：只需要扩展应用程序中存在性能问题的那些部分。这也有助于节省基础设施的资金。

灵活性

如果设计得当，微服务不太容易受到供应商的限制。当决定要切换一个第三方组件时，不必一次完成整个痛苦的迁移。微服务的设计考虑到需要使用接口，因此唯一需要修改的部分是微服务和第三方组件之间的接口。

组件也可能一个接一个地迁移，有些仍然使用来自旧供应商的软件。这样，可以将一次在多个地方引入破坏性更改的风险分离开来。更重要的是，可以将此与金丝雀部署模式结合起来，以更细的粒度管理风险。

这种灵活性不仅仅与单个服务相关。还可能意味着不同的数据库、不同的队列和消息传递解决方案，甚至完全不同的云平台。虽然不同的云平台通常提供不同的服务和 API 来使用，但通过微服务架构，可以逐个迁移工作负载，并在一个新的平台上独立测试。

由于性能问题、扩展性或可用的依赖关系而需要重写时，重写微服务要比重写整体服务快得多。

与旧系统集成

微服务不一定是全有或全无的方法。如果应用程序经过了良好的测试，并且迁移到微服务可能会产生很多风险，就没有压力去废除可用的解决方案。最好只拆分需要进一步开发的部分，并将它们作为原始服务使用的微服务引入。

通过这种方法，将获得与微服务相关的敏捷发布周期的好处，同时避免从零开始创建新的架构和基本上重新构建整个应用程序。如果某些东西已经运行得很好了，那最好是专注于如何在不破坏好的部分的情况下添加新功能，而不是从头开始。

分布式开发

开发团队规模小、位置集中的时代已经过去了。即使在传统的基于办公室的公司中，远程工作和分布式开发也是一个事实。像 IBM、Microsoft 和 Intel 这样的巨头会让来自不同地方的人在一个项目上工作。

微服务支持更小、更敏捷的团队，这使得分布式开发更加容易。当不再需要促进 20 人或 20 人以上的团队之间的沟通时，构建需要较少外部管理的自组织团队也更容易。

13.2.2 微服务的缺点

即使认为可能需要微服务，请记住它们的一些缺点。简而言之，它们不适合每个人。大公司通常可以抵消这些缺点，但小公司通常不能有这种奢求。

依赖成熟的 DevOps

构建和测试微服务应该比在大型单应用程序上执行类似操作要快得多。但是为了实现敏捷开发，这种构建和测试将需要更频繁地执行。

当处理一个庞然大物时，手动部署应用程序可能是明智的，但是如果将同样的方法应用于微服务，则会导致许多问题。

为了在开发中拥抱微服务，必须确保团队有 DevOps 的思维，理解构建和运行微服务的需求。仅仅把代码交给别人，然后忘记是不够的。

DevOps 的思维模式将帮助团队尽可能地实现自动化，在没有持续集成/持续交付流水的情况下开发微服务可能是软件架构中最糟糕的想法，这种方法将带来微服务的所有其他缺点。

难以调试

微服务需要引入可观察性。当出现了问题，永远不知道从哪里开始寻找根本原因。可观察性是一种推断应用程序状态的方法，无需运行调试器或将日志记录到运行工作负载的机器上。

日志聚合、应用程序指标、监视和分布式跟踪的组合是管理基于微服务架构的先决条件。当考虑到自动扩展和自修复可能阻止访问单个服务（开始崩溃）时，这一点尤为正确。

额外的开销

微服务应该是精简和敏捷的，这通常是正确的。然而，基于微服务的架构通常需要额外的开销，第一层开销与用于微服务通信的附加接口有关。RPC 库、API 提供者和使用者不仅要乘以微服务的数量，还要乘以副本的数量。然后还有辅助服务，如数据库、消息队列等。这些服务还包括可观察性设施，通常包括存储设施和收集数据的个人收集器。

以更好的扩展性进行优化的成本可能会超过运行整个服务所需的成本，这些服务不能立即带来业务价值。更重要的是，可能很难向相关方证明这些成本（在基础设施和开发开销方面）的合理性。

13.2.3 微服务的设计模式

很多通用的设计模式也适用于微服务，还有一些通常与微服务相关的设计模式。这里提供的模式对于新建项目和从单一应用程序迁移都很有用。

分解模式

这些模式与分解微服务的方式有关。我们希望确保架构是稳定的，服务是松散耦合的。还希望确保服务具有内聚性和可测试性。最后，希望团队完全拥有一个或多个服务。

按业务能力分解

其中一种分解模式需要根据业务能力进行分解，业务能力与企业为了产生价值所做的事情有关。业务功能的例子有商人管理和客户管理，通常按层次结构组织。

应用此模式时的主要挑战是正确识别业务功能。这需要理解业务本身，并可能需要与业务分析师合作。

分解子域名

另一种分解模式与领域驱动设计 (DDD) 方法有关。为了定义服务，需要识别 DDD 子域。就像业务能力一样，标识子域需要业务方面的知识。

这两种方法的主要区别在于，按业务能力分解时，重点更多地放在业务组织 (其结构) 上。而按子域分解时，重点放在业务试图解决的问题上。

数据库/服务模式

存储和处理数据在每个软件架构中都是一个复杂的问题。错误的选择可能会影响扩展性、性能或维护成本。对于微服务，由于希望微服务是松散耦合的，这就增加了复杂性。

这导致了这样一种设计模式：每个微服务都连接到自己的数据库，因此不受其他服务引入的影响。虽然这种模式会增加一些开销，但好处是可以为每个微服务分别优化模式和索引。

由于数据库往往是相当庞大的基础设施，所以这种方法可能不可行，在微服务之间共享数据库是一种可以理解的权衡方式。

部署策略

对于在多个主机上运行的微服务，可能想知道哪种分配资源的方式更好。让我们比较一下这两种方法。

单主机单服务

使用这个模式，允许每个主机只服务特定类型的微服务。好处是可以调整机器以更好地适应所需的工作负载，并且很好地隔离了服务。当提供超大内存或快速存储时，可以确保其只用于需要的微服务。服务也无法消耗超过所提供的资源。

这种方法的缺点是有些主机可能没有得到充分利用。一种可能的解决方法是使用尽可能小的机器来满足微服务需求，并在必要时扩展。但这种解决方法不能解决主机本身额外开销的问题。

单主机多服务

另一种相反的方法是在每个主机上托管多个服务。这有助于优化机器的利用率，但也有一些缺点。首先，不同的微服务可能需要不同的优化，因此将它们托管在单个主机上仍然不可能。此外，这种方法将失去对主机分配的控制，因此一个微服务中的问题可能会导致一个托管微服务的中断，即使后者在其他方面不会受到影响。

另一个问题是微服务之间的依赖冲突。当微服务没有彼此隔离时，部署必须考虑不同的依赖关系。这个模型也不太安全。

可观测性模式

前一节中，提到微服务是有代价的。这个代价包括引入可观察性的需求，或者冒着失去调试应用程序能力的风险。这里有一些与可观察性相关的模式。

日志收集

微服务就像单应用程序一样使用日志。日志不是存储在本地，而是聚合并转发到一个中央设施。这样，即使服务本身宕机，日志也可用。以集中的方式存储日志还有助于关联来自不同微服务

的数据。

应用指标

要根据数据做出决策，首先需要一些数据来采取行动。收集应用程序度量有助于理解实际用户使用的应用程序行为，而不是在合成测试中使用的行为。收集这些指标的方法是推（应用程序在其中主动调用性能监视服务）和拉（性能监视服务定期检查配置的端点）。

分布式追踪

分布式追踪不仅有助于研究性能问题，还有助于更好地了解真实流量下的应用程序行为。与从单个点收集信息片段的日志记录不同，跟踪关注单个事务的整个生命周期，从源自用户操作的点开始。

健康检查 API

由于微服务通常是自动化的目标，因此需要能够传达内部状态。即使进程存在于系统中，也不意味着应用程序是可操作的。这同样适用于开放的网络端口，应用程序可能正在监听，但还不能响应。运行状况检查 API 为外部服务提供了一种确定应用程序是否已准备好处理工作负载的方法。自修复和自动扩展使用运行状况检查来确定何时需要干预。基本前提是，当应用程序的行为符合预期时，给定端点（例如/health）返回 HTTP 代码 200，如果发现问题，则返回不同的代码（或者根本不返回）。

了解了所有的优点、缺点和模式，接下来将展示如何将单个应用程序分割并将其部分地转换为微服务。所提出的方法不仅限于微服务；它们在其他情况下也可能有用，包括单应用程序。

13.3. 构建微服务

关于单应用有很多观点。一些架构师认为，单应用就是邪恶的，因为它们不能很好地扩展，是紧密耦合的，而且很难维护。还有一些人声称，来自单应用的性能优势抵消了缺点。事实上，紧密耦合的组件在网络、处理能力和内存方面所需的开销比松散耦合的组件要小得多。

当涉及到相关方时，每个应用程序都有独特的业务需求，并在独特的环境中操作，因此没有关于哪种方法更适合的通用规则。更令人困惑的是，在从单服务到微观服务的最初迁移之后，一些公司开始将微观服务整合为宏观服务。这是因为维护数千个独立软件实例的负担太大，有时可能无法处理。

架构的选择应该总是来自于业务需求和对不同备选方案的仔细分析，将意识形态置于实用主义之前通常会导致组织内耗。当一个团队不考虑不同的解决方案或不同的外部意见，而试图不惜一切代价坚持给定的方法时，这个团队就不再履行为正确的工作交付正确的工具的义务。

如果正在开发或维护一个单应用，可以考虑提高扩展性。本节介绍的技术旨在解决这个问题，同时也使应用程序更容易迁移到微服务（如果决定这样做的话）。

瓶颈的主要原因有三：

- 内存
- 存储
- 计算

这里将展示如何使用它们来开发基于微服务的可扩展解决方案。

13.3.1 外包的内存管理

帮助微服务规模扩大的方法是外包他们的任务。内存管理和缓存数据可能会阻碍扩展工作。

对于单应用，将缓存的数据直接存储在进程内存中没有问题，因为进程将是唯一访问缓存的进程。但是由于一个过程有多个副本，这种方法会出现一些问题。

如果一个副本已经计算了工作负载的一部分，并将其存储在本地缓存中，该怎么办？另一个副本不知道这一事实，必须再次计算。这样，应用程序既浪费了计算时间（因为必须多次执行相同的任务），又浪费了内存（因为结果也分别存储在每个副本中）。

为了避免这种情况，可以考虑切换到外部内存存储，而不是在应用程序内部管理缓存。使用外部解决方案的另一个好处是，缓存的生命周期不再与应用程序的生命周期绑定。可以重新启动并部署新版本的应用程序，并保留已经存储在缓存中的值。

这也可能导致更短的启动时间，因为应用程序不再需要在启动期间执行计算。两种主流的内存缓存解决方案是 Memcached 和 Redis。

Memcached

2003 年发布的 Memcached 是两者中较老的产品。它是一个通用的、分布式的键值存储。该项目最初的目标是通过将缓存的值存储在内存中来卸载 Web 应用程序中使用的数据库。Memcached 是按设计分发的，从 1.5.18 版本开始，可以在不丢失缓存内容的情况下重新启动 Memcached 服务器。这可以通过使用 RAM 磁盘作为临时存储空间来实现。

它使用了一个简单的 API，可以通过 telnet 或 netcat 进行操作，也可以使用许多流行编程语言的绑定。没有针对 C++ 的绑定，但可以使用 C/C++ 的 libmemcached 库。

Redis

Redis 是一个比 Memcached 更新的项目，最初版本发布于 2009 年。从那以后，Redis 在很多情况下都取代了 Memcached。就像 Memcached 一样，它是一个分布式的、通用的、内存中的键值存储。

与 Memcached 不同的是，Redis 还有可选的数据持久性。虽然 Memcached 操作的键和值是简单的字符串，Redis 也支持其他数据类型，例如：

- 字符串列表
- 字符串集合
- 已排序的字符串集合
- 其中键和值为字符串的哈希表
- 地理空间数据（从 Redis 3.2 开始）
- HyperLog 日志

Redis 的设计使它成为缓存会话数据、缓存网页和实现排行榜的绝佳选择。除此之外，还可以用于消息队列。Python 中流行的分布式任务队列库 Celery，使用 Redis 作为可能的代理之一，还有 RabbitMQ 和 Apache SQS。

Microsoft，Amazon，Google 和 Alibaba 都将基于 redis 的管理服务作为其云平台的一部分。

Redis 客户端有很多 C++ 的实现。两个有趣的是使用 C++17 编写的 redis-cpp 库 (<https://github.com/tdv/redis-cpp>) 和使用 Qt 工具包编写的 QRedisClient (<https://github.com/uglide/qredisclient>)。

以下使用 redis-cpp 的示例取自官方文档，演示了如何来设置和获取存储中的一些数据：

```
1 #include <cstdlib>
2 #include <iostream>
3 #include <redis-cpp/execute.h>
4 #include <redis-cpp/stream.h>
5 int main() {
6     try {
7         auto stream = rediscpp::make_stream("localhost", "6379");
8
9         auto const key = "my_key";
10
11        auto response = rediscpp::execute(*stream, "set", key,
12                                         "Some value for 'my_key'", "ex",
13                                         "60");
14
15        std::cout << "Set key '" << key << "' : "
16                  << response.as<std::string>()
17                  << std::endl;
18
19        response = rediscpp::execute(*stream, "get", key);
20        std::cout << "Get key '" << key << "' : "
21                  << response.as<std::string>()
22                  << std::endl;
23    } catch (std::exception const &e) {
24        std::cerr << "Error: " << e.what() << std::endl;
25        return EXIT_FAILURE;
26    }
27    return EXIT_SUCCESS;
28 }
```

该库处理不同的数据类型，该示例将该值设置为字符串列表。

内存中的哪个缓存更好？

对于大多数应用程序来说，Redis 是一个更好的选择。它有更好的用户社区，许多不同的实现，并且得到很好的支持。除此之外，还具有快照、复制、事务和发布/订阅模型。在 Redis 中嵌入 Lua 脚本是可能的，对地理空间数据的支持使它成为地理支持的 Web 和移动应用程序的一个很好的选择。

但若主要目标是在 Web 应用程序中缓存数据库查询的结果，Memcached 是一个更简单的解决方案，开销要小得多。这意味着其可以更好地使用资源，因为它不必存储类型元数据或执行不同类型之间的转换。

13.3.2 外包存储

引入和扩展微服务时，另一个可能的限制是存储。通常，本地块设备用于存储不属于数据库的对象（如静态 PDF 文件、文档或图像）。即使在今天，块存储仍然非常受本地块设备和网络文件系统（如 NFS 或 CIFS）的欢迎。

虽然 NFS 和 CIFS 是网络附加存储（NAS）的领域，但还有一些协议与在不同级别上运行的一个概念相关：存储区域网络（SAN）。一些流行的是 iSCSI，网络块设备（NBD），以太网 ATA，光纤通道协议和以太网光纤通道。

不同的方法为分布式计算设计了集群文件系统：GlusterFS、CephFS 或 Lustre。然而，所有这些都作为块设备向用户公开相同的 POSIX 文件 API。

Amazon 网络服务提出了一个关于存储的新观点。S3（Amazon Simple Storage Service）是对象存储服务，API 提供了对存储在 bucket 中的对象的访问。这与传统的文件系统不同，因为文件、目录或索引节点之间没有区别。有指向对象的桶和键，对象是服务存储的二进制数据。

13.3.3 外包计算

微服务的原则之一，是流程应该只负责完成工作流的单个部分。从单服务迁移到微服务的一个自然步骤是定义可能的长时间运行的任务，并将它们拆分为单独的进程。

这是任务队列背后的概念，任务队列处理管理任务的整个生命周期。使用任务队列，不需要自己实现线程化或多处理，而是委托要执行的任务，然后由任务队列异步处理。该任务可以在与原始进程相同的机器上执行，但也可以在具有特殊需求的机器上运行。

任务及其结果是异步的，因此在主进程中不存在阻塞。Web 开发中流行的任务队列例子有 Python 的 Celery，Ruby 的 Sidekiq，Node.js 的 Kue 和 Go 的 Machinery。所有这些都可以用 Redis 作为代理。不幸的是，对于 C++ 还没有任何成熟的解决方案。

如果正在认真考虑采用这种方式，可能的方法是直接在 Redis 中实现一个任务队列。Redis 和它的 API 提供了必要的原语来支持这样的行为。另一种可能的方法是使用现有的任务队列之一（如 Celery），并通过直接调用 Redis 来进行调用。但不建议这样做，因为取决于任务队列的实现细节，而不是文档化的公共 API。另一种方法是使用 SWIG 或类似方法提供的绑定来对接任务队列。

13.4. 观察微服务

构建的每个微服务都需要遵循一般的架构设计模式。微服务和传统应用程序的主要区别在于前者需要实现可观察性。

本节主要讨论一些可观察性的方法。在这里描述了几种开源解决方案，可能会发现它们在设计系统时很有用。

13.4.1 记录日志

即使从未设计过微服务，也应该熟悉日志记录这个主题。日志（或日志文件）保存了系统中发生的事件的信息。系统可能意味着应用程序、应用程序运行的操作系统或用于部署的云平台。这些组件中的每一个都可以提供日志。

日志作为单独的文件存储，因为提供了发生的所有事件的永久记录。当系统变得无响应时，可以查询日志并找出故障的原因。

这意味着日志还提供了审计跟踪。因为事件是按时间顺序记录的，所以能够通过检查记录的历史状态来了解系统的状态。

为了协助调试，日志通常需要提供好的可读性。日志有二进制格式，但在使用文件存储日志时，这种格式非常罕见。

微服务中的日志记录

这种日志记录方法本身与传统方法没有太大区别。微服务通常将日志打印到标准输出，而不是使用文本文件在本地存储日志。然后使用统一的日志记录层检索和处理日志。要实现日志记录，需要一个可以配置的日志库。

使用 spdlog 在 C++ 中进行日志记录

spdlog 是 C++ 中一种流行且快速的日志库。用 C++11 构建，既可以作为头文件库使用，也可以作为静态库使用（这样可以减少编译时间）。

spdlog 的一些有趣特性包括：

- 格式化
- 多种接收方式：
 - 文件
 - 控制台
 - 系统记录
 - 自定义（作为单个函数实现）
- 多线程和单线程版本
- 可选的异步模式

spdlog 可能缺少的一个特性是对 Logstash 或 Fluentd 的直接支持。如果希望使用这些聚合器，仍然可以使用文件接收输出配置 spdlog，并使用 Filebeat 或 Fluent Bit 将文件内容转发到适当的聚合器。

统一的日志记录层

大多数时候，无法控制使用的所有微服务。其中一些将使用同一个日志库，而其他将使用不同的日志库。最重要的是，格式将完全不同，轮换政策也将完全不同。更糟糕的是，仍然希望将操作系统事件与应用程序事件关联起来。这就是统一日志记录层发挥作用的地方。

统一日志记录层的目的是收集来自不同来源的日志。这种统一的日志记录层工具提供许多集成，并理解不同的日志记录格式和传输（如文件、HTTP 和 TCP）。

统一日志记录层还可以对日志进行过滤。可能需要过滤以满足合规性，匿名化客户个人详细信息，或保护服务的实施细节。

为了便于以后查询日志，统一日志层还可以执行格式之间的转换。即使使用的服务以 JSON、CSV 和 Apache 格式存储日志，统一的日志层解决方案也能够将它们全部转换为 JSON。

统一日志记录层的最后一项任务是将日志转发到下一个目的地。根据系统的复杂性，下一个目的地可能是存储设施或另一个过滤、转换和转发设施。

以下是一些有趣的组件，允许构建统一的日志记录层。

Logstash

Logstash 是最受欢迎的统一日志层解决方案之一。目前，它属于 Elasticsearch 背后的公司 Elastic。若听说过 ELK 堆栈（现在称为弹性堆栈），Logstash 是首字母缩写中的“L”。

Logstash 是用 Ruby 编写的，然后移植到 JRuby 上。不幸的是，这意味着它是相当资源密集型的。由于这个原因，不建议在每台机器上运行 Logstash。相反，它主要用作日志转发，在每台机器上部署轻量级的 Filebeat，只进行收集。

Filebeat

Filebeat 是 Beats 系列产品的一部分，目标是为 Logstash 提供一个轻量级的替代方案，可以直接受与应用程序一起使用。

通过这种方式，Beats 提供了低开销和良好的扩展性，而集中的 Logstash 安装执行所有繁重的工作，包括转译、过滤和转发。

除了 Filebeat，Beats 家族的其他产品如下：

- Metricbeat 用于性能
- Packetbeat 用于网络数据
- Auditbeat 用于审计数据
- Heartbeat 用于运行时监控

Fluentd

Fluentd 是 Logstash 的主要竞争对手，也是一些云提供商选择的工具。

由于使用插件的模块化方法，可以找到用于数据源（如 Ruby 应用程序、Docker 容器、SNMP 或 MQTT 协议）、数据输出（如 Elastic Stack、SQL Database、Sentry、Datadog 或 Slack）以及其他几种过滤器和中间件的插件。

Fluentd 应该比 Logstash 更少的资源，但不是大规模运行的完美解决方案。与使用 Fluentd 的 Filebeat 对应的称为 Fluent Bit。

Fluent Bit

Fluent Bit 用 C 编写，提供了一种插入 Fluentd 的更快、更轻的解决方案。作为一个日志处理器和转发器，还具有许多输入和输出的集成功能。

除了日志收集，Fluent Bit 还可以监控 Linux 系统上的 CPU 和内存指标。可以与 Fluentd 一起使用，也可以直接转发给 Elasticsearch 或 InfluxDB。

Vector

虽然 Logstash 和 Fluentd 是稳定、成熟和经过尝试的解决方案，但在统一日志记录层领域也有新的主张。

其中一个是 Vector，目标是在一个工具中处理所有的可观测数据。为了区别于竞争，其关注性能和正确性。这也体现在技术的选择上。Vector 使用 Rust 作为引擎，使用 Lua 编写脚本（与 Logstash 和 Fluentd 使用的定制领域特定语言相反）。

在撰写本书的时候，它还没有达到稳定的 1.0 版本，所以不应该将其应用于生产。

日志聚合

日志聚合解决了由于数据过多而产生的另一个问题：如何存储和访问日志。虽然统一的日志记录层使日志即使在机器停机的情况下也可用，但日志聚合的任务是快速找到正在寻找的信息。

可以存储、索引和查询大量数据的两种产品是 Elasticsearch 和 Loki。

Elasticsearch

Elasticsearch 是自托管日志聚合的最流行的解决方案。这是（前）ELK 堆栈中的“E”。它有一个基于 Apache Lucene 的强大搜索引擎。

作为其领域内的事实上的标准，Elasticsearch 有很多集成，并且有来自社区和商业服务的强大支持。一些云提供商将 Elasticsearch 作为托管服务提供，这使得在应用程序中引入 Elasticsearch 更加容易。除此之外，制造 Elasticsearch 的公司 Elastic 提供了一个托管解决方案，不绑定任何特定的云提供商。

Loki

Loki 的目标是解决 Elasticsearch 的一些缺点。Loki 关注的领域是水平可扩展性和高可用性。它是作为云本机解决方案从头构建的。

Loki 的设计灵感来自 Prometheus 和 Grafana。这并不奇怪，因为它是由负责 Grafana 的团队开发的。

虽然 Loki 应该是一个稳定的解决方案，但它没有 Elasticsearch 那么受欢迎，这意味着可能会缺少一些集成，文档和支持也不会像 Elasticsearch 一样，但 Fluentd 和 Vector 都有支持 Loki 日志聚合的插件。

日志可视化

想要考虑的日志堆栈的最后一部分是日志可视化。便于对日志进行查询和分析。可以以一种可访问的方式显示数据，以便所有相关方（如运营商、开发人员、QA 或业务人员）都可以检查数据。

日志可视化工具允许创建仪表板，更容易读取感兴趣的数据。就能够探索事件，搜索相关性，并从一个简单的用户界面中找到离群的数据。

有两个主要的产品专门用于日志可视化。

Kibana

Kibana 是 ELK Stack 的最后一个元素，在 Elasticsearch 之上提供了一种更简单的查询语言。尽管可以使用 Kibana 查询和可视化不同类型的数据，但它主要集中在日志上。

与 ELK Stack 的其他部分一样，它目前是可视化日志的实际标准。

Grafana

Grafana 是另一个数据可视化工具。最近，它还主要关注性能指标的时间序列数据。然而，随着 Loki 的引入，也可以用于日志。

它的优点是，它在构建时考虑到了可插拔后端，因此很容易切换存储以满足需求。

13.4.2 监控

监视是从系统中收集与性能相关的指标的过程。当与警报结合使用时，监视可以了解系统何时按预期运行，以及何时发生事件。

最感兴趣的三种参数类型如下：

- 可用性，知道哪些资源已经启动并运行，以及哪些资源已经崩溃或失去响应。
- 资源利用率，了解工作负载如何适应系统。
- 性能，告诉我们在哪里以及如何提高服务质量。

监测的两种模型是推和拉。前者，每个监视对象（一台机器、一个应用程序和一个网络设备）周期性地将数据推送到中心点。在后一种情况下，对象在配置的端点上显示数据，监视代理定期抓取数据。

拉模型使它更容易扩展，多个对象就不会阻塞监视代理连接。相反，多个代理可以随时收集数据，从而更好地利用可用资源。

提供 C++ 客户端库特性的两种监控解决方案是 Prometheus 和 InfluxDB。Prometheus 是一个基于拉模型的例子，专注于收集和存储时间序列数据。默认情况下，fluxdb 使用推送模型。除了监控，物联网、传感器网络和家庭自动化也很受欢迎。

Prometheus 和 InfluxDB 通常与 Grafana 一起用于可视化数据和管理仪表板。两者都有内置警报，也可以通过 Grafana 与外部警报系统集成。

13.4.3 跟踪

跟踪提供的信息通常比事件日志的信息级别低。另一个重要的区别是，跟踪存储每个事务的 ID，因此很容易可视化整个工作流。这个 ID 通常称为跟踪 ID、事务 ID 或关联 ID。

与事件日志不同，跟踪并不意味着良好的可读性，其由示踪器处理。在实现跟踪时，需要使用与系统的所有可能元素集成的跟踪解决方案：前端应用程序、后端应用程序和数据库。通过这种方式，跟踪有助于查明性能滞后的确切原因。

OpenTracing

分布式跟踪的标准之一是 OpenTracing。这个标准是由开源追踪器之一的 Jaeger 的作者提出的。

除了 Jaeger 之外，OpenTracing 还支持许多不同的追踪器，并且支持许多不同的编程语言。最重要的包括以下几点：

- Go
- C++
- C#

- Java
- JavaScript
- Objective-C
- PHP
- Python
- Ruby

OpenTracing 最重要的特性是它是厂商中立的。在测试应用程序时，将不需要修改整个代码库来切换到不同的跟踪程序，这就可以防止厂商锁定。

Jaeger

Jaeger 是一个跟踪器，可以用于各种后端，包括 Elasticsearch，Cassandra 和 Kafka。

它与 OpenTracing 天生兼容，由于它是一个本地云计算基础毕业的项目，因此它拥有强大的社区支持，这也转化为与其他服务和框架的良好集成。

OpenZipkin

OpenZipkin 是 Jaeger 的主要竞争对手，它在市场上已经很长时间了。虽然这意味着它是一个更成熟的解决方案，但与 Jaeger 相比，它的受欢迎程度正在下降。特别是由于 OpenZipkin 中的 C++没有积极地维护，这可能会导致未来的维护问题。

13.4.4 集成的可观测性解决方案

如果不想自己构建可观察层，可以考虑一些流行的商业解决方案。它们都以软件即服务的模式运作，不会在这里进行详细的比较，因为在本书写完之后，相应的东西可能会发生巨大的变化。

这些服务如下：

- Datadog
- Splunk
- Honeycomb

本节中，了解到了在微服务中实现可观察性。接下来，将继续了解如何连接微服务。

13.5. 连接微服务

微服务之所以如此有用，是因为它们可以通过许多不同的方式与其他服务连接起来，从而创造新的价值。然而，由于微服务没有标准，所以没有单一的方式进行连接。

大多数情况下，想要使用一个特定的微服务时，必须学会如何与它交互。好消息是，尽管在微服务中实现任何通信方法都是可能的，但有一些流行的方法是大多数微服务所遵循的。

如何连接微服务只是围绕设计架构时的相关问题。另一个是连接什么，以及连接在哪里，就是服务发现发挥作用的地方。通过服务发现，让微服务使用自动的方法来发现和连接应用程序中的其他服务。

这三个问题，如何做，做什么，何处做，将是下一个主题。我们将介绍现代微服务中使用的一些主流的通信和发现方法。

13.5.1 应用程序编程接口

就像软件库一样，微服务经常公开 API。这些 API 使得与微服务通信成为可能。由于典型的通信方式利用计算机网络，最流行的 API 形式是 Web API。

前一章中，已经介绍了一些使用 Web 服务的可能方法。如今，微服务通常使用基于具象状态传输 (REpresentational State Transfer, REST) 的 Web 服务。

13.5.2 远程过程调用

虽然 REST 等 Web API 允许轻松调试和良好的互操作性，但与数据转换和使用 HTTP 进行传输相关的开销很大。

对于一些微服务来说，这种开销可能太大了，这就是轻量级远程过程调用 (Remote Procedure call, RPC) 的原因。

Apache Thrift

Apache Thrift 是一个接口描述语言和二进制通信协议。它用作 RPC 方法，允许创建以各种语言构建的分布式和可扩展的服务。

支持多种二进制协议和传输方法。每种编程语言都使用本机数据类型，因此即使在现有的代码库中也很容易引入本机的数据。

gRPC

如果真的关心性能，通常会发现基于文本的解决方案并不适合。REST 无论多么优雅和容易理解，对于需求来说可能太慢了。如果是这种情况，应该尝试围绕二进制协议构建 API，其中一种是 gRPC。

gRPC 是最初由 Google 开发的 RPC 系统。使用 HTTP/2 进行传输，并使用协议缓冲区作为接口描述语言 (IDL) 来实现多种编程语言之间的互操作性和数据序列化。这可以使用替代技术，例如 FlatBuffers。gRPC 可以以同步和异步的方式使用，并允许创建简单的服务和流服务。

假设决定使用 protobufs，我们的 Greeter 服务定义如下：

```
1 service Greeter {
2     rpc Greet(GreetRequest) returns (GreetResponse);
3 }
4 message GreetRequest {
5     string name = 1;
6 }
7 message GreetResponse {
8     string reply = 1;
9 }
```

使用协议编译器，从这个定义创建数据访问代码。假设想为 Greeter 创建一个同步服务器，可以按以下方式创建服务：

```
1 class Greeter : public Greeter::Service {
2     Status sendRequest(ServerContext *context, const GreetRequest
3                         *request,
```

```
4         GreetReply *reply) override {
5     auto name = request->name();
6     if (name.empty()) return Status::INVALID_ARGUMENT;
7     reply->set_result("Hello " + name);
8     return Status::OK;
9 }
10};
```

然后，必须为它构建并运行服务器：

```
1 int main() {
2     Greeter service;
3     ServerBuilder builder;
4     builder.AddListeningPort("localhost", grpc::InsecureServerCredentials());
5     builder.RegisterService(&service);
6
7     auto server(builder.BuildAndStart());
8     server->Wait();
9 }
```

就这么简单。现在看看使用这个服务的客户端：

```
1 #include <grpcpp/grpcpp.h>
2
3 #include <string>
4
5 #include "grpc/service.grpc.pb.h"
6
7 using grpc::ClientContext;
8 using grpc::Status;
9
10 int main() {
11     std::string address("localhost:50000");
12     auto channel =
13         grpc::CreateChannel(address, grpc::InsecureChannelCredentials());
14     auto stub = Greeter::NewStub(channel);
15
16     GreetRequest request;
17     request.set_name("World");
18
19     GreetResponse reply;
20     ClientContext context;
21     Status status = stub->Greet(&context, request, &reply);
22
23     if (status.ok()) {
24         std::cout << reply.reply() << '\n';
25     } else {
26         std::cerr << "Error: " << status.error_code() << '\n';
27     }
28 }
```

这是一个简单的同步示例。为了让它异步工作，需要添加标签和 CompletionQueue，如 gRPC 网站上描述的那样。

gRPC 的一个有趣特性是，可用于 Android 和 iOS 上的移动应用程序。如果在内部使用 gRPC，不需要提供额外的服务器来转换来自移动应用程序的流量。

本节中，了解了微服务使用的最流行的通信和发现方法。接下来，将了解到微服务是如何扩展的。

13.6. 扩展微服务

微服务的好处是，比单应用更扩展起来更容易。考虑到相同的硬件基础设施，理论上可以从微服务中获得比单体服务更多的性能。

实践中，好处的获得可没那么简单。微服务和相关的助手还提供了开销，对于较小规模的应用程序来说，性能可能不如最优的单应用。

记住，即使某件事“纸面上”看起来很好，这并不意味着它会飞起来。如果希望基于可扩展性或性能来制定架构决策，那么最好准备计算和实验。这样，相应的行为就会基于数据，而不仅仅是情绪。

13.6.1 每台主机部署单个服务

对于每个主机部署一个服务，扩展微服务需要添加或删除托管该微服务的其他机器。如果应用程序运行在云架构（公共的或私有的）上，许多提供商提供了一种称为自动扩展组的概念。

自动扩展组定义了一个基本虚拟机镜像，在所有分组实例上运行。每当达到临界阈值时（例如，CPU 使用率为 80%），会创建一个新实例并将其添加到组中。由于自动扩展组在负载均衡器后面运行，因此增加的流量会在现有和新实例之间分配，从而降低每个实例的平均负载。当流量峰值减弱时，扩展控制器会关闭多余的机器，以保持低成本运行。

不同的指标可以作为扩展事件的触发器。CPU 负载是最容易使用的负载之一，但它可能不是最准确的负载。其他指标（如队列中的消息数量）可能更适合相应的应用程序。

以下是扩展策略的 Terraform 配置的一个摘录：

```
1 autoscaling_policy {
2   max_replicas = 5
3   min_replicas = 3
4   cooldown_period = 60
5   cpu_utilization {
6     target = 0.8
7   }
8 }
```

在任何给定的时间，至少有三个实例在运行，最多有五个实例。当所有组实例的平均 CPU 负载达到至少 80% 时，扩展器将触发。当这种情况发生时，将启动一个新实例。只有在新机器运行至少 60 秒（冷却期）后，才会收集它的指标。

13.6.2 每台主机部署多个服务

这种扩展模式也适用于每台主机部署多个服务的情况。但这不是最有效的方法，仅根据单个服务的吞吐量降低来扩展整个服务集类似于扩展整体。

如果正在使用这种模式，那么扩展微服务的更好方法是使用协调器。如果不使用容器，Nomad 是一个很好的选择，它可以与许多不同的执行驱动程序一起工作。对于集装箱化的工作负载，Docker Swarm 或 Kubernetes 都是不错的助手。协调器是接下来的两章中的主题。

13.7. 总结

微服务是软件架构的一个新趋势。如果确定了解危险并做好准备，它们可能是一个很好的选择。本章解释了有助于引入微服务的常见设计和迁移模式。我们还讨论了一些高级主题，比如在建立基于微服务的体系结构时至关重要的可观察性和连接性。

目前为止，应该能够将应用程序设计并分解为单个微服务。然后，每个微服务都能够处理单个工作负载。

虽然微服务本身是有效的，但它们与容器结合使用时尤其受欢迎。容器是下一章的主题。

13.8. 练习题

1. 为什么微服务可以更好地利用系统资源？
2. 微服务和单应用如何共存（在一个不断发展的系统中）？
3. 哪种类型的团队从微服务中受益最大？
4. 为什么在引入微服务时需要一个成熟的 DevOps 方法？
5. 什么是统一日志记录层？
6. 日志记录和跟踪有什么不同？
7. 为什么 REST 不是连接微服务的最佳选择？
8. 微服务的部署策略是什么？各自的好处是什么？

13.8. 扩展阅读

- Mastering Distributed Tracing: <https://www.packtpub.com/product/masteringdistributed-tracing/9781788628464>
- Hands-On Microservices with Kubernetes: <https://www.packtpub.com/product/hands-on-microservices-with-kubernetes/9781789805468>
- Microservices by Martin Fowler: <https://martinfowler.com/articles/microservices.html>
- Microservice architecture: <https://microservices.io/>

第 14 章 容器

从开发到生产一直是一个痛苦的过程，涉及大量文档、交接、安装和配置。由于每种编程语言产生的软件的行为都略有不同，因此异构应用程序的部署总是很困难。

容器缓解了其中一些问题。对于容器，安装和配置基本上是标准化的。有几种方法来处理分发，但这个问题也有一些标准要遵循。这使得容器成为增加开发和操作之间合作的选择。

本章将讨论以下内容：

- 构建容器
- 测试和集成容器
- 容器协调器

14.1. 相关准备

本章的例子需要提前安装如下软件：

- Docker 20.10
- manifest-tool (<https://github.com/estesp/manifest-tool>)
- Buildah 1.16
- Ansible 2.10
- ansible-bender
- CMake 3.15

本章的代码可以在以下 GitHub 页面找到：<https://github.com/PacktPublishing/Software-Architecture-with-Cpp/tree/master/Chapter14>。

14.2. 引入容器

容器最近引起了很多关注。有人可能会认为它们是一种全新的技术，以前没有。然而，在 Docker 和 Kubernetes 崛起之前，已经有像 LXC 这样的解决方案，它提供了很多类似的功能。

可以通过自 1979 年以来在 UNIX 系统中使用的 chroot 机制，来追溯将一个执行环境与另一个执行环境分离的起源。类似的概念也在 FreeBSD jails 和 Solaris 专区中使用。

容器的主要任务是隔离一个执行环境与另一个执行环境。这个隔离的环境可以拥有自己的配置、不同的应用程序，甚至与主机环境不同的用户帐户。

尽管容器与主机是隔离的，但通常共享相同的操作系统内核。这是与虚拟化环境的主要区别。虚拟机有专用的虚拟资源，这意味着在硬件级别的分离。容器在流程级别上是分开的，这意味着运行它们的开销更少。

能够打包并运行已经为运行应用程序进行了优化和配置的另一个操作系统，这是容器的一个强大优势。如果没有容器，构建和部署过程通常包括以下几个步骤：

1. 构建应用程序。
2. 提供配置文件示例。
3. 准备已安装脚本和相关文档。

4. 该应用程序是针对目标操作系统 (如 Debian 或 Red Hat) 打包的。
5. 这些包部署到目标平台。
6. 安装脚本为应用程序的运行准备了基础。
7. 必须调整配置以适应现有的系统。

当切换到容器时，不太需要健壮的安装脚本。应用程序将只针对一个已知的操作系统——容器中存在的操作系统。配置也是如此：与准备许多可配置选项不同，应用程序是为目标操作系统预先配置的，并与之一起分发。部署流程只包括解包容器映像并在其中运行应用程序流程。

虽然集成和微型服务通常是一回事，但它们并非如此。此外，容器可能意味着应用程序容器或操作系统容器，只有应用程序容器适合微服务。将描述可能遇到的不同容器类型，展示与微服务的关系，并解释何时使用（以及何时不使用）。

14.2.1 容器类型

目前所描述的容器中，操作系统容器与目前由 Docker、Kubernetes 和 LXD 所引领的容器趋势不同。应用程序容器关注的是在容器中运行单个进程——仅是应用程序，而不是使用 syslog 和 cron 等服务重新创建整个操作系统。

专有解决方案取代了所有常见的操作系统级服务，这些解决方案提供了统一的方法来管理容器内的应用程序。例如，不使用 syslog 来处理日志，而将 PID 为 1 的进程的标准输出视为应用程序日志。而不是使用 init.d 等机制或 systemd，应用程序容器的生命周期由运行时应用程序处理。

由于 Docker 目前是应用程序容器的主要解决方案，将在本书中主要使用它作为例子。为了使情况更完整，将提供可行的替代方案，因为它们可能更适合某些需要。由于项目和规范是开源的，这些替代方案与 Docker 兼容，可以作为替代品使用。

本章的后面，将解释如何使用 Docker 来构建、部署、运行和管理应用程序容器。

14.2.2 微服务的兴起

Docker 的成功恰逢微服务的兴起，因为微服务和应用程序容器可以很自然地结合在一起。

如果没有应用程序容器，就没有简单而统一的方式来打包、部署和维护微服务。尽管个别公司开发了一些解决方案来解决这些问题，但没有一个可以成为行业标准。

如果没有微服务，应用程序容器就非常有限。软件架构的重点是构建针对运行在那里的给定服务集显式配置的整个系统。用另一个服务替换一个服务需要改变架构。

当应用程序容器组合在一起时，它为微服务的分发提供了一种标准方式。每个微服务器都自带嵌入式配置，因此自动扩展或自修复等操作不再需要了解底层应用。

仍然可以在没有应用程序容器的情况下使用微服务，也可以在没有托管微服务的情况下使用应用程序容器。例如，尽管 PostgreSQL 数据库和 Nginx Web 服务器都没有设计成微服务，但它们通常会用在应用程序容器中。

14.2.3 何时使用容器

容器方法有几个好处，操作系统容器和应用程序容器也有它们的优势所在。

容器的好处

与虚拟机（隔离环境的另一种流行方式）相比，容器在运行时需要的开销更少。与虚拟机不同，不需要运行操作系统内核的单独版本，也不需要使用硬件或软件虚拟化技术。应用程序容器也不运行虚拟机中常见的其他操作系统服务，如 syslog、cron 或 init。此外，应用程序容器提供较小的映像，因为它们通常不需要携带整个操作系统副本。在极端的例子中，应用程序容器可以由单个静态链接的二进制文件组成。

如果容器中只有一个二进制文件，为什么还要费心处理容器呢？有一种统一和标准化的方式来构建和运行容器有一个特别的好处。由于容器必须遵循特定的约定，因此它比常规的二进制文件更容易进行编排，后者在日志记录、配置、开放端口等方面可能有不同的期望。

另外，容器提供了一种内置的隔离方法。每个容器都有自己的进程名称空间和用户帐户名称空间。这意味着来自一个容器的进程（或多个进程）没有主机上或其他容器中的进程的概念。沙箱甚至可以更进一步，可以用相同的标准用户界面（无论是 Docker、Kubernetes 还是其他东西）为容器分配内存和 CPU 配额。

标准化的运行还意味着更高的可移植性。构建容器之后，通常可以在不同的操作系统上运行，而无需进行修改。在操作中运行的内容，与在开发中运行的内容非常接近或相同。问题复现更加轻松，调试也是。

容器的缺点

由于现在将工作负载迁移到容器的压力很大，作为架构师，希望了解与这种迁移相关的所有风险，并到处都在吹捧它的好处。

采用容器的主要问题是，并不是所有的应用程序都可以轻松地迁移到容器。对于在设计时考虑到微服务的应用程序容器尤其如此。如果应用程序不是基于微服务体系结构的，那么将其放入容器可能会带来比它能解决的更多的问题。

如果应用程序已经可以很好地扩展，使用基于 TCP/IP 的 IPC，并且大多数情况下是无状态的，那么迁移到容器应该不会很困难。否则，这些方面中的每一个都将带来挑战，并需要对现有设计进行重新思考。

与容器相关的另一个问题是持久存储。理想情况下，容器应该没有自己的持久存储。这使得快速创建、易于扩展和灵活调度成为可能。问题在于，如果没有持久存储，提供业务价值的应用程序就不可能存在。

通过使大多数容器无状态，并依赖外部非容器化组件来存储数据和状态，这一缺陷通常可以得到缓解。这样的外部组件可以是传统的自托管数据库，也可以是来自云提供商的托管数据库。无论选择哪个方向，都需要重新考虑架构并进行相应地修改。

由于应用程序容器遵循特定的约定，因此必须修改应用程序以遵循这些约定。对于某些应用程序来说，这将是一项简单的任务。对于其他组件，例如使用内存内进程间通信（IPC）的多进程组件，就会很复杂。

经常忽略的是，只要应用程序容器中的应用程序是本机 Linux 应用程序，应用程序容器就可以工作得很好。虽然支持 Windows 容器，但它们既不方便，也不像 Linux 容器那样受支持，还需要作为主机运行授权的 Windows 机器。

如果从头构建一个新的应用程序，并且可以基于这种技术进行设计，就更容易享受应用程序容器的好处。将现有的应用程序移动到应用程序容器，特别是在复杂的情况下，将需要大量的工作，

而且可能还需要修改整个架构。这种情况下，建议考虑所有的好处和坏处。做出错误的决定可能会影响产品的交付时间、可用性和预算。

14.3. 构建容器

应用容器是本节的重点。虽然操作系统容器大多遵循系统编程原则，但容器带来了新的挑战和模式。此外，它们还提供专门的构建工具来应对这些挑战。我们的主要工具是 Docker，它是当前构建和运行应用程序容器的标准。我们还会介绍一些构建应用程序容器的替代方法。

本节中，将重点介绍使用 Docker 构建和部署容器的不同方法。

14.3.1 容器镜像

描述容器镜像以及如何构建之前，理解容器和容器镜像之间的区别至关重要。这两个术语经常会有混淆，尤其是在非正式对话中。

容器和容器镜像之间的区别，就像正在运行的进程和可执行文件之间的区别一样。

容器镜像是静态的，是特定文件系统和相关元数据的快照。元数据描述在运行时设置哪些环境变量，或者在从镜像创建容器时运行哪些程序。

容器是动态的，运行容器镜像中包含的进程。可以从容器镜像创建容器，也可以通过对运行中的容器进行快照来创建容器镜像。实际上，容器镜像构建过程包括创建多个容器，在容器中执行命令，并在命令完成后对它们进行快照。

为了区分容器镜像引入的数据和运行时生成的数据，Docker 使用联合挂载文件系统来创建不同的文件系统层。

这些层也出现在容器镜像中。通常，容器镜像的每个构建步骤对应于生成的容器镜像中的一个新层。

14.3.2 使用 Dockerfiles 构建应用程序

使用 Docker 构建应用程序容器镜像的最常见方法是使用 Dockerfile。Dockerfile 是一种命令式语言，描述生成结果图像所需的操作。一些操作创建新的文件系统层，其他则对元数据进行操作。

我们不讨论 Dockerfiles 相关的细节。相反，将展示载入 C++ 应用程序的不同方法。为此，需要引入一些与 Dockerfiles 相关的语法和概念。

下面是一个非常简单的 Dockerfile：

```
FROM ubuntu:bionic
RUN apt-get update && apt-get -y install build-essentials gcc
CMD /usr/bin/gcc
```

通常，可以把 Dockerfile 分成三个部分：

- 导入基础镜像 (FROM 指令)
- 在容器内执行将产生容器镜像的操作 (RUN 指令)
- 运行时使用的元数据 (CMD 命令)

后两部分可以相互交织，每一部分可以包含一个或多个指令。因为只有基本镜像是必需的，所以可以省略后面的部分。这并不意味着不能从一个空文件系统开始。有一个名为 scratch 的特殊基本镜像正是为此创建，将一个静态链接的二进制文件添加到一个空的文件系统：

```
FROM scratch
COPY customer /bin/customer
CMD /bin/customer
```

第一个 Dockerfile 中，采取如下步骤：

1. 导入 Ubuntu Bionic 基础镜像。
2. 在容器内运行命令。该命令的结果将在目标镜像中创建一个新的文件系统层。这意味着 apt-get 安装的包将在基于此镜像的所有容器中可用。
3. 设置运行时元数据。基于此镜像创建容器时，希望运行 GCC 作为默认进程。

要从 Dockerfile 构建镜像，将使用 `docker build` 命令。它有一个必需的参数，即包含构建上下文的目录，Dockerfile 本身以及想要复制到容器中的其他文件。要从当前目录构建 Dockerfile，请使用 `docker build|`。

这将构建一个匿名镜像，大多数时候，都希望使用已命名的镜像。在命名容器镜像时需要遵循一个约定，将在下一节中进行讨论。

14.3.3 命名和分发镜像

Docker 中的每个容器镜像都有一个独特的名称，由三个元素组成：注册表名称、图像名称、标签。容器注册表是保存容器镜像的对象存储库，Docker 的默认注册表是 Docker.io。当从这个注册表中提取镜像时，可以省略注册表名称。

前面 `ubuntu:bionic` 的例子的全名是 `docker.io/ubuntu:bionic`。这个例子中，`ubuntu` 是镜像的名字，而 `bionic` 是代表镜像特定版本的标签。

基于容器构建应用程序时，可能会对存储所有注册表镜像感兴趣。可以托管私有注册中心，并将镜像保存在那里，或者使用托管解决方案。流行的托管解决方案包括：

1. Docker Hub
2. quay.io
3. GitHub
4. 云提供商（比如 AWS、GCP 或 Azure）

Docker Hub 仍然是最受欢迎的，不过一些公共镜像正在迁移到 quay.io。两者都是通用的，并允许存储公共和私有镜像。如果已经在使用一个平台，并希望让自己的镜像接近 CI 流水或部署目标，那么 GitHub 或云提供商的服务则最有吸引力。如果想减少使用的单个服务的数量，这也会很有帮助。

如果这些解决方案都不合适，托管本地注册表也非常简单，并且需要运行单个容器。

要构建命名镜像，需要将 `-t` 参数传递给 `docker` 构建命令。例如，要构建一个名为 `dominicfair/merchant:v2.0.3` 的镜像，可以使用 `docker build -t`

`dominicanfair/merchant:v2.0.3 ..` 命令进行构建。

14.3.4 已编译的应用和容器

当使用解释语言 (如 Python 或 JavaScript) 构建应用程序的容器映像时，方法基本相同：

1. 安装依赖。
2. 将源文件复制到容器镜像内。
3. 复制必要的配置。
4. 设置运行时命令。

但是，对于编译的应用程序，还有一个额外的步骤，即首先编译应用程序。实现这一步骤有几种可能的方法，每种方法都有其优缺点。

最明显的方法是首先安装所有依赖项，复制源文件，然后将应用程序编译为容器构建步骤之一。主要的好处是可以精确地控制工具链的内容和配置，因此有一种可移植的方式来构建应用程序。但是，缺点太大：生成的容器镜像包含许多不必要的文件。毕竟，在运行时，既不需要源代码，也不需要工具链。由于覆盖文件系统的工作方式，不可能在前一层引入文件后删除文件。更重要的是，如果攻击者设法闯入容器，容器中的源代码可能是一个安全风险。

可以这样进行构建：

```
FROM ubuntu:bionic
RUN apt-get update && apt-get -y install build-essentials gcc cmake
ADD . /usr/src
WORKDIR /usr/src
RUN mkdir build && \
    cd build && \
    cmake .. -DCMAKE_BUILD_TYPE=Release && \
    cmake --build . && \
    cmake --install .
CMD /usr/local/bin/customer
```

另一种方法，是在主机上构建应用程序，并且只在容器映像中复制生成的二进制文件。当一个构建过程已经建立时，这需要对当前构建过程进行更少的更改。主要缺点是必须在构建机器上匹配与容器中相同的库集。例如，如果运行的是 Ubuntu 20.04 作为主机操作系统，容器也必须基于 Ubuntu 20.04。否则，将面临不兼容的风险。使用这种方法，还需要独立于容器配置工具链。

就像这样：

```
FROM scratch
COPY customer /bin/customer
CMD /bin/customer
```

一种稍微复杂一点的方法是采用多阶段构建。对于多阶段构建，一个阶段可能专门用于设置工具链并编译项目，而另一个阶段将生成的二进制文件复制到目标容器映像。这比以前的解决方案有几个优点。首先，Dockerfile 现在同时控制了工具链和运行时环境，因此构建的每一步都有完整的文档记录。其次，可以将镜像与工具链一起使用，以确保开发与持续集成/持续部署 (CI/CD) 流水之间的兼容性。这种方式也使得向工具链本身分发升级和修复更容易。主要的缺点是，容器化工具链使用起来可能不像本地工具链那样舒适。另外，构建工具并不是特别适合应用程序容器，因为应用程序容器要求每个容器都有一个进程在运行。当某些进程崩溃或强制停止时，这可能会导致意外行为。

前面例子的多阶段版本是这样的：

```
FROM ubuntu:bionic AS builder
RUN apt-get update && apt-get -y install build-essentials gcc cmake
ADD . /usr/src
WORKDIR /usr/src
RUN mkdir build && \
    cd build && \
    cmake .. -DCMAKE_BUILD_TYPE=Release && \
    cmake --build .
FROM ubuntu:bionic
COPY --from=builder /usr/src/build/bin/customer /bin/customer
CMD /bin/customer
```

第一阶段，从第一个 FROM 命令开始，设置构建器，添加源文件，并构建二进制文件。然后，第二个阶段（从第二个 FROM 命令开始）从上一个阶段复制生成的二进制文件，而不复制工具链或源代码。

14.3.5 使用清单针对多架构

带有 Docker 的应用程序容器通常用于 x86_64（也称为 AMD64）机器。如果只针对这个平台，那没什么好担心的。然而，如果正在开发物联网、嵌入式或边缘应用程序，可能会对多架构镜像感兴趣。

由于 Docker 可以在许多不同的 CPU 架构上使用，所以有几种方法可以在多个平台上进行镜像管理。

处理针对不同目标构建的镜像的一种方法是，使用镜像标签来描述特定的平台。可以使用 `merchant:v2.0.3-aarch64`，而不是 `merchant:v2.0.3`。尽管这种方法看起来是最容易实现的，但实际上有点问题。

不仅必须更改构建过程，以便在标记过程中包含架构。当拉出镜像来运行时，必须手动将预期的后缀添加到所有地方。如果使用编配器，则无法以直接的方式在不同平台之间共享清单。

一种不需要修改部署步骤的方法是使用清单工具 (<https://github.com/estesp/manifest-tool>)，构建过程最初看起来与前面建议的类似。镜像在所有受支持的体系结构上分别构建，并将其

推送到注册中心，其标签中带有平台后缀。在推送所有镜像之后，manifest-tool 将这些映像合并为一个单一的多架构镜像。这样，每个受支持的平台都能够使用完全相同的标记。

清单工具的配置示例如下：

```
image: hosacpp/merchant:v2.0.3
manifests:
- image: hosacpp/merchant:v2.0.3-amd64
  platform:
    architecture: amd64
    os: linux
- image: hosacpp/merchant:v2.0.3-arm32
  platform:
    architecture: arm
    os: linux
- image: hosacpp/merchant:v2.0.3-arm64
  platform:
    architecture: arm64
    os: linux
```

这里，有三个受支持的平台，每个平台都有各自的后缀 (`hosacpp/merchant:v2.0.3-amd64`、`hosacpp/merchant:v2.0.3-arm32` 和 `hosacpp/merchant:v2.0.3-arm64`)。Manifest-tool 结合了为每个平台构建的镜像，并生成了一个 `hosacpp/merchant:v2.0.3` 镜像，可以在任何地方使用。

另一种可能是使用 Docker 的内置功能 Buildx。使用 Buildx，可以附加几个构建器实例，每个实例针对所需的架构。有趣的是，不需要本地机器来运行构建，还可以在多阶段构建中使用 QEMU 模拟或交叉编译。尽管 Buildx 比之前的方法强大得多，但它也相当复杂。在编写本文时，它需要 Docker 实验模式和 Linux 内核 4.8 或更高版本。它需要设置和管理构建器，并不是所有东西都以直观的方式运行。有可能在不久的将来会有所改善，变得更加稳定。

准备构建环境并构建多平台镜像的示例代码如下所示：

```

# create two build contexts running on different machines
docker context create \
  --docker host=ssh://docker-user@host1.domifair.org \
  --description="Remote engine amd64" \
  node-amd64

docker context create \
  --docker host=ssh://docker-user@host2.domifair.org \
  --description="Remote engine arm64" \
  node-arm64

# use the contexts
docker buildx create --use --name mybuild node-amd64
docker buildx create --append --name mybuild node-arm64

# build an image
docker buildx build --platform linux/amd64,linux/arm64 .

```

如果习惯于常规的 docker 构建命令，可能会有些困惑。

14.3.6 构建应用容器的替代方法

使用 Docker 构建容器映像需要 Docker 守护进程运行。Docker 守护进程需要根权限，这可能在某些设置中造成安全问题。即使执行构建的 Docker 客户端可能由非特权用户运行，但在构建环境中安装 Docker 守护进程并不总是可行的。

Buildah

Buildah 是构建容器镜像的替代工具，可以配置为在没有根访问权限的情况下运行。Buildah 可以与常规的 dockerfile 一起工作。它还提供了自己的命令行界面，可以在 shell 脚本或其他更直观的自动化操作中使用该界面。使用 buildah 接口将之前的一个 dockerfile 重写为 shell 脚本，如下所示：

```

#!/bin/sh
ctr=$(buildah from ubuntu:bionic)
buildah run $ctr -- /bin/sh -c 'apt-get update && \
  apt-get install -y buildessential gcc'
buildah config --cmd '/usr/bin/gcc' "$ctr"
buildah commit "$ctr" hosacpp-gcc
buildah rm "$ctr"

```

Buildah 的一个有趣特性是，它允许将容器镜像文件系统挂载到主机文件系统中。这样，就可以使用主机的命令与镜像的内容进行交互。如果有不希望（或由于许可限制不能）放在容器中的软件，在使用 Buildah 时仍然可以在容器外部调用。

Ansible-bender

Ansible-bender 使用 Ansible playbook 和 Buildah 来构建容器镜像。所有的配置，包括基本镜像和元数据，都作为 playbook 中的一个变量传递。下面是之前转换成 Ansible 语法的例子：

```
---
```

```
- name: Container image with ansible-bender
  hosts: all
  vars:
    ansible_bender:
      base_image: python:3-buster

      target_image:
        name: hosacpp-gcc
        cmd: /usr/bin/gcc

  tasks:
    - name: Install Apt packages
      apt:
        pkg:
          - build-essential
          - gcc
```

`ansible_bender` 变量负责所有特定于容器的配置，下面的任务是在基于 `base_image` 的容器内执行的。

Ansible 需要在基础镜像中提供一个 Python 解释器，这就是必须把之前例子中的 `ubuntu:bionic` 改为 `python:3-buster` 的原因。`bionic` 是一个没有预装 Python 解释器的 `ubuntu` 镜像。

其他

还有其他方法可以构建容器镜像。例如，可以使用 Nix 创建一个文件系统镜像，然后使用 Dockerfile 的 `COPY` 指令将其放入镜像中。更进一步，使用其他方法准备文件系统镜像，然后使用 `docker import` 将其作为基本容器镜像导入。

选择适合特定需求的解决方案。请记住，使用 `docker build` 构建 Dockerfile 是最流行的方法，因此它是文档记录最好、支持最好的方法。使用 Buildah 更灵活，允许更好地将创建容器映像融入到构建过程中。最后，如果已经在 Ansible 上投入了大量成本，并且想重用已经在用的模块，`ansiblebender` 可能是一个很好的解决方案。

14.3.7 使用 CMake 集成容器

本节中，将演示如何使用 CMake 创建 Docker 镜像。

使用 CMake 配置 Dockerfile

首先，需要一个 Dockerfile。使用另一个 CMake 输入文件：

```
configure_file(${CMAKE_CURRENT_SOURCE_DIR}/Dockerfile.in  
    ${PROJECT_BINARY_DIR}/Dockerfile @ONLY)
```

注意，如果项目是一个更大项目的一部分，则使用 PROJECT_BINARY_DIR 来不会覆盖源树中其他项目创建的 dockerfile。

Dockerfile.in 文件如下所示：

```
FROM ubuntu:latest  
ADD Customer-@PROJECT_VERSION@-Linux.deb .  
RUN apt-get update && \  
    apt-get -y --no-install-recommends install ./Customer-  
@PROJECT_VERSION@-Linux.deb && \  
    apt-get autoremove -y && \  
    apt-get clean && \  
    rm -r /var/lib/apt/lists/* Customer-@PROJECT_VERSION@-Linux.deb  
ENTRYPOINT ["/usr/bin/customer"]  
EXPOSE 8080
```

首先，指定使用最新的 Ubuntu 映像，在上面安装 DEB 包及其依赖项，然后进行整理。更新包管理器缓存的步骤和安装包的步骤是一样的，这很重要，以避免由于 Docker 中的层的工作方式而导致缓存过期的问题。清理也作为同一 RUN 命令的一部分（在同一层中）执行，以便层体积更小。安装包之后，让镜像在启动时运行客户微服务。最后，告诉 Docker 公开它将要监听的端口。

现在，回到 CMakeLists.txt 文件。

使用 CMake 集成容器

对于基于 CMake 的项目，可以包含负责构建容器的构建步骤。为此，需要告诉 CMake 找到 Docker 可执行文件，如果没有就退出。可以这样做：

```
find_program(Docker_EXECUTABLE docker)  
if(NOT Docker_EXECUTABLE)  
    message(FATAL_ERROR "Docker not found")  
endif()
```

让我们重温第 7 章中的一个例子，为客户应用程序构建了一个二进制文件和一个 Conan 包。现在，希望将该应用程序打包为 Debian 存档，并使用客户应用程序的预安装包构建一个 Debian 容器镜像。

为了创建 DEB 包，需要一个帮助器目标。可以使用 CMake 的 `add_custom_target` 功能：

```
add_custom_target(
    customer-deb
    COMMENT "Creating Customer DEB package"
    COMMAND ${CMAKE_CPACK_COMMAND} -G DEB
    WORKING_DIRECTORY ${PROJECT_BINARY_DIR}
    VERBATIM)
add_dependencies(customer-deb libcustomer)
```

目标调用 CPack 来创建一个包，而忽略了其余的包。为了方便起见，希望将包创建在与 Dockerfile 相同的目录下。建议使用 `VERBATIM` 关键字，有了它 CMake 将会转义有问题的字符。如果没有指定，脚本的行为可能会在不同的平台上有所不同。

`add_dependencies` 调用将确保在 CMake 构建 `customer-deb` 目标之前，`libcustomer` 已经构建好。现在我们有了帮助器目标，可以在创建容器镜像时使用：

```
add_custom_target(
    docker
    COMMENT "Preparing Docker image"
    COMMAND ${Docker_EXECUTABLE} build ${PROJECT_BINARY_DIR}
        -t dominicanfair/customer:${PROJECT_VERSION} -t
        dominicanfair/customer:latest
    VERBATIM)
add_dependencies(docker customer-deb)
```

调用 Docker 可执行文件来创建一个镜像，之前在包含 Dockerfile 和 DEB 包的目录中找到了 Docker 可执行文件。还告诉 Docker 将镜像标记为 `dominicanfair/customer:latest`。最后，确定在调用 Docker 目标时构建 DEB 包。

如果使用 `make` 作为生成器，构建镜像就和 `make docker` 一样简单。如果喜欢使用完整的 CMake 命令（例如，创建与生成器无关的脚本），可以用 `cmake --build . --target docker`。

14.4. 测试和集成容器

容器非常适合 CI/CD 流水。因为除了容器运行时本身之外，大多不需要其他依赖项，所以很容易测试。不需要配置工作机器来满足测试需求，因此添加节点要容易得多。更重要的是，它们都是通用的，因此可以充当构建器、测试运行器，甚至部署执行器，而不需要事先进行配置。

在 CI/CD 中使用容器的另一个好处是彼此隔离，在同一台机器上运行的多个副本不应互有干扰。除非测试需要来自主机操作系统的一些资源，例如端口转发或卷挂载。因此，最好设计测试，使这些资源不是必需的（或至少它们不冲突）。例如，端口随机化是避免冲突的一种技术。

14.4.1 容器内的运行时库

容器的选择可能会影响工具链的选择，从而影响应用程序可用的 C++ 语言特性。因为容器通常是基于 Linux 的，所以可用的系统编译器通常是 GNU GCC，并将 glibc 作为标准库。然而，一些流行于容器的 Linux 发行版，如 Alpine Linux，是基于不同的标准库 musl 的。

如果目标是这样一个发行版，请确保使用的代码（无论是内部开发的还是来自第三方提供商的）与 musl 兼容。musl 和 Alpine Linux 的主要优点是它可以产生更小的容器映像。例如，为 Debian Buster 构建的 Python 镜像大约是 330MB，精简版 Debian 大约是 40MB，而 Alpine 版只有 16MB。更小的镜像意味着更少的带宽浪费（用于上传和下载）和更快的更新。

Alpine 还可能引入一些不必要的特性，比如更长的构建时间、晦涩的 bug 或性能下降。如果想使用它来减小体积，请运行适当的测试以确保应用程序的行为没有问题。

为了进一步缩小镜像大小，可以考虑放弃底层操作系统。这里的操作系统指的是通常出现在容器中的所有用户域工具，例如 shell、包管理器和动态库。毕竟，如果应用程序是唯一要运行的东西，那么其他一切都是不必要的。

Go 或 Rust 应用程序通常可以提供自给自足的静态构建，并可以形成一个容器镜像。虽然这在 C++ 中可能不那么简单，但它值得考虑。

减小镜像大小也有一些缺点。首先，如果决定使用 Alpine Linux，请记住它不像 Ubuntu、Debian 或 CentOS 那样流行。尽管它通常是容器开发人员的首选平台，但很少有其他用途。

这意味着可能会出现新的兼容性问题，主要是因为它不是基于事实上的标准 glibc 实现。如果依赖于第三方组件，提供商可能不提供此平台的支持。

如果决定在容器镜像路由中使用单个静态链接的二进制文件，那么还需要考虑一些挑战。首先，不建议静态链接 glibc，因为在内部使用 dlopen 来处理名称服务开关 (NSS) 和 iconv。如果软件依赖于 DNS 解析或字符集转换，那么必须提供 glibc 和相关库的副本。

需要考虑的是，shell 和包管理器通常用于调试行为不正常的容器。当某个容器运行异常时，可以启动容器内的另一个进程，并通过使用标准的 UNIX 工具（如 ps、ls 或 cat）来了解容器内发生了什么。要在容器内运行这样的应用程序，必须首先出现在容器镜像中。一些变通方法允许操作符在运行的容器中注入调试二进制文件，但目前都没有得到很好的支持。

14.4.2 选择容器的运行时

Docker 是最流行的构建和运行容器的方式，但由于容器标准是开放的，可以使用其他的运行时。提供类似用户体验的 Docker 的主要替代品是 Podman。和上一节描述的 Buildah 一起，是旨在完全取代 Docker 的工具。

好处是它们不需要像 Docker 那样在主机上运行额外的守护进程。还支持（尽管还不成熟）无根操作，这使它们更适合安全性关键的操作。Podman 接受所有希望 Docker CLI 接受的命令，所以可以简单地用它作为别名。

另一种旨在提供更好安全性的容器方法是 Kata 容器倡议。Kata 容器使用轻量级虚拟机来利用容器和主机操作系统之间额外隔离级别所需的硬件虚拟化。

Cri-O 和 containerd 也是 Kubernetes 使用的主流运行时。

14.5. 容器协调器

只有在使用容器协调器进行管理时，容器的一些好处才会显现出来。协调器跟踪将运行工作负载的所有节点，它还监视分布在这些节点上的容器的健康状况和状态。

更高级的特性，例如高可用性，需要对协调器进行适当的设置，这通常意味着至少为控制平面专用三台机器，另外三台机器用于工作节点。除了容器的自动扩展外，节点的自动扩展还需要协调器拥有能够控制底层基础设施的驱动程序（例如，通过使用云提供商的 API）。

这里，将介绍一些最主流的协调器，可以选择它们作为系统的基础。第 15 章中可以看到更多关于 Kubernetes 的实用信息。在此，将概述一些可能的选择。

所提供的协调器操作类似的对象（服务、容器、批处理作业），尽管每个对象的行为可能不同，可用特性和工作原理各不相同。共同点是，通常编写一个配置文件，以声明的方式描述所需的资源，然后使用专用的 CLI 工具应用此配置。为了说明这两种工具之间的区别，提供了一个配置示例，指定一个之前介绍过的 Web 应用程序（merchant 服务）和一个 Web 服务器 Nginx 作为代理。

14.5.1 自托管解决方案

无论是在本地运行应用程序、私有云中，还是公共云中运行应用程序，可能希望对所选择的协调器进行严格的控制。以下是该领域的自托管解决方案集合，它们中的大多数也可以作为托管服务使用。然而，使用自托管可以防止供应商锁定，这可能是开发者和其组织所希望的。

Kubernetes

Kubernetes 可能是在这里提到的最著名的协调器。它很流行，如果决定使用它，将会有大量的文档和社区支持。

尽管 Kubernetes 使用了与 Docker 相同的应用程序容器格式，但这基本上是所有相似之处的终结。使用标准的 Docker 工具直接与 Kubernetes 集群和资源交互是不可能的。在使用 Kubernetes 时，有一组新的工具和概念需要学习。

而在 Docker 中，容器是操作的主要对象，而在 Kubernetes 中，运行时最小的部分称为 Pod。一个 Pod 可以由一个或多个共享挂载点和网络资源的容器组成。Pod 本身很少引起人们的兴趣，因为 Kubernetes 也有高阶的概念，如复制控制器、部署控制器或守护进程集的作用是跟踪 Pod，并确保在节点上运行所需数量的副本。

Kubernetes 的网络模型也与 Docker 有很大的不同。使用 Docker，可以从一个容器转发端口，以便从不同的机器访问它。使用 Kubernetes，如果想访问 Pod，通常会创建一个 Service 资源，它可以充当负载均衡器来处理到 Pod（形成服务的后端）的流量。服务可能用于 Pod 对 Pod 的通信，但也可能暴露在互联网上。在内部，Kubernetes 资源使用 DNS 名称执行发现服务。

Kubernetes 是说明性的，最终是一致的。这意味着不是直接创建和分配资源，而是只需要提供所需的最终状态的描述，Kubernetes 将完成将集群带到所需状态所需的工作。资源通常使用 YAML 进行描述。

由于 Kubernetes 是高度可扩展的，因此有很多相关的项目是在云本地计算基金会（CNCF）下开发的，这使得 Kubernetes 成为一个提供不可知的云开发平台。我们将在下一章，第 15 章中更详细地介绍 Kubernetes。

下面是资源定义如何使用 YAML（merchant.yaml）查找 Kubernetes：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: dominican-front
    name: dominican-front
spec:
  selector:
    matchLabels:
      app: dominican-front
  template:
    metadata:
      labels:
        app: dominican-front
    spec:
      containers:
        - name: webserver
          imagePullPolicy: Always
          image: nginx
          ports:
            - name: http
              containerPort: 80
              protocol: TCP
      restartPolicy: Always
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: dominican-front
    name: dominican-front
spec:
  ports:
    - port: 80
      protocol: TCP
      targetPort: 80
  selector:
    app: dominican-front
  type: ClusterIP
---
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: dominican-merchant
    name: merchant
spec:
  selector:
    matchLabels:
      app: dominican-merchant
  replicas: 3
  template:
    metadata:
      labels:
        app: dominican-merchant
  spec:
    containers:
      - name: merchant
        imagePullPolicy: Always
        image: hosacpp/merchant:v2.0.3
        ports:
          - name: http
            containerPort: 8000
            protocol: TCP
    restartPolicy: Always
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: dominican-merchant
    name: merchant
spec:
  ports:
    - port: 80
      protocol: TCP
      targetPort: 8000
  selector:
    app: dominican-merchant
  type: ClusterIP
```

要应用此配置并协调容器，请使用 `kubectl apply -f merchant.yaml`。

Docker Swarm

Docker Engine 也需要构建和运行 Docker 容器，自带预安装的协调器。这个协调器就是 Docker Swarm，它的主要特点是通过使用 Docker API 与现有的 Docker 工具高度兼容。

Docker Swarm 使用服务的概念来管理健康检查和自动扩展，支持本地服务的滚动升级。服务可以发布端口，然后由 Swarm 的负载均衡器提供服务。支持将配置存储为用于运行时定制的对象，并内置了基本的机密管理。

Docker Swarm 比 Kubernetes 简单得多，扩展性也差得多。如果不了解 Kubernetes 的所有细节，这可能是一个优势。然而，主要的缺点是缺乏知名度，这意味着很难找到有关 Docker Swarm 的相关资料。

使用 Docker Swarm 的一个好处是不需要学习新的命令。如果已经习惯使用 Docker 和 Docker Compose，那么 Swarm 将使用相同的资源。它允许特定的选项来扩展 Docker 来处理部署。

与 Swarm 协调的两个服务是这样的 (`docker-compose.yml`):

```
version: "3.8"
services:
  web:
    image: nginx
    ports:
      - "80:80"
    depends_on:
      - merchant
  merchant:
    image: hosacpp/merchant:v2.0.3
    deploy:
      replicas: 3
    ports:
      - "8000"
```

要应用这个配置，可以运行 `docker stack deploy -- composition -file dockercompose.yml dominican`。

Nomad

Nomad 不同于前两种解决方案，因为它不单单关注容器。它是一个通用的协调器，支持 Docker、Podman、Qemu 虚拟机、隔离的 fork/exec 和其他一些任务驱动程序。如果希望在不将应用程序迁移到容器的情况下获得容器协调的一些优势，那么 Nomad 是一个值得去了解的解决方案。

相对容易设置，并与其他 HashiCorp 产品集成得很好，比如用于服务发现的 Consul 和用于机

密管理的 Vault。和 Docker 或 Kubernetes 一样，Nomad 客户端可以在本地运行，并连接到负责管理集群的服务器。

Nomad 有三种工作类型：

- 服务：一个长生命周期的任务，在没有人工干预的情况下不能退出（例如，Web 服务器或数据库）。
- 批处理：短时间的任务，可以在几分钟内完成。如果批处理作业返回指示错误的退出码，则根据配置重新启动或重新调度它。
- 系统：需要在集群中的每个节点上运行的任务（例如，日志代理）。

与其他协调器相比，Nomad 相对容易安装和维护。当涉及到任务驱动程序或设备插件（用于访问专用硬件，如 GPU 或 FPGA）时，也是可扩展的。与 Kubernetes 相比，缺乏社区支持和第三方集成。Nomad 不需要重新设计应用程序的架构来访问所提供的好处，而 Kubernetes 就是这样。

要用 Nomad 配置这两个服务，需要两个配置文件。第一个是 nginx.nomad：

```
job "web" {
  datacenters = ["dc1"]
  type = "service"
  group "nginx" {
    task "nginx" {
      driver = "docker"
      config {
        image = "nginx"
        port_map {
          http = 80
        }
      }
      resources {
        network {
          port "http" {
            static = 80
          }
        }
      }
      service {
        name = "nginx"
        tags = [ "dominican-front", "web", "nginx" ]
        port = "http"
        check {
          type = "tcp"
        }
      }
    }
  }
}
```

```
    interval = "10s"
    timeout = "2s"
  }
}
}
}
}
```

第二个描述的是商业应用程序，所以称为 merchant.nomad:

```
job "merchant" {
  datacenters = ["dc1"]
  type = "service"
  group "merchant" {
    count = 3
    task "merchant" {
      driver = "docker"
      config {
        image = "hosacpp/merchant:v2.0.3"
        port_map {
          http = 8000
        }
      }
      resources {
        network {
          port "http" {
            static = 8000
          }
        }
      }
    }
    service {
      name = "merchant"
      tags = [ "dominican-front", "merchant" ]
      port = "http"
      check {
        type = "tcp"
        interval = "10s"
      }
    }
  }
}
```

```
    timeout = "2s"
}
}
}
}
}
```

要应用该配置，可以运行 `nomad job run merchant.Nomad && nomad job run nginx.nomad`。

OpenShift

OpenShift 是红帽公司基于 Kubernetes 开发的商业容器平台，包括许多在 Kubernetes 集群的日常操作中很有用的附加组件。其中包括获得一个容器注册表、一个类似 Jenkins 的构建工具、用于监视的 Prometheus、用于服务网格的 Istio 和用于跟踪的 Jaeger。它不能与 Kubernetes 完全兼容，所以不能视为一个临时替代品。

它是建立在现有的红帽技术，如 CoreOS 和红帽企业 Linux 之上。可以在 Red Hat 云内部使用它，也可以在受支持的公共云提供商（包括 AWS、GCP、IBM 和 Microsoft Azure）上使用它，或者作为混合云使用。

还有一个开源社区支持的项目 OKD，它是红帽 OpenShift 的基础。如果不需要 OpenShift 的商业支持和其他好处，可以在 Kubernetes 工作流中使用 OKD。

14.5.2 管理服务

如前所述，其中一些协调器也可以作为托管服务使用。例如，Kubernetes 是多个公共云提供商提供的托管解决方案。本节将向展示容器协调的一些不同方法，不基于上面提到的解决方案。

AWS ECS

在 Kubernetes 发布 1.0 版本之前，Amazon Web 服务就提出了自己的容器协调技术，称为弹性容器服务 (ECS)。ECS 提供了一个协调器，可以在需要时监视、扩展和重新启动服务。

要在 ECS 中运行容器，需要提供工作负载将在其上运行的 EC2 实例。不需要为协调器的使用付费，但是需要为常使用的所有 AWS 服务（例如，底层 EC2 实例或 RDS 数据库）付费。

ECS 的好处是与 AWS 生态系统的其他部分完美集成。如果熟悉 AWS 服务，并投资于该平台，那么理解和管理 ECS 将会更容易。

如果不使用 Kubernetes 的许多高级特性及其扩展，ECS 可能是一个更好的选择，因为它更直接，更容易学习。

AWS Fargate

AWS 提供的另一个托管协调器是 Fargate。与 ECS 不同的是，不要求准备底层 EC2 实例并为其付费。关注的唯一组件是容器、与它们相连的网络接口和 IAM 权限。

与其他解决方案相比，Fargate 需要最少的维护，而且最容易学习。由于该领域现有的 AWS 产品，自动扩展和负载平衡可以开箱即用。

其主要缺点是，与 ECS 相比，需要为托管服务而支付的额外费用。直接比较是不可能的，因为 ECS 需要为 EC2 实例付费，而 Fargate 需要单独为内存和 CPU 使用付费。当服务开始自动扩展，这种对集群直接控制的缺乏，很容易导致非常高的成本。

Azure 服务结构

上述所有解决方案的问题在于，主要针对 Docker 容器，而 Docker 容器首先是以 Linux 为中心的。另一方面，Azure 服务结构是 Microsoft 支持 windows 的优先产品。它支持不需要修改就可以运行遗留的 Windows 应用程序，如果应用程序依赖于这些服务，这可能会有助于迁移应用程序。

与 Kubernetes 一样，Azure 服务结构本身并不是一个容器协调器，而是一个可以在其上构建应用程序的平台。其中一个构建块恰好是容器，因此可以作为协调器使用。

随着 Azure Kubernetes 服务 (Azure 云中的托管 Kubernetes 平台) 的引入，使用 Service Fabric 的需求逐渐变少了。

14.6. 总结

对于现代软件的架构师来说，必须考虑到现代技术。考虑这些因素并不意味着盲目跟风，需要能客观地评估一个特定的命题，在现在所处的情况下是否有意义。

前几章介绍的微服务和本章介绍的容器都值得考虑和理解。它们也值得应用吗？很大程度上取决于所设计的产品类型。如果读到这里，估计已经准备出了自己的决定。

下一章将专注于云原生设计。这是一个非常有趣但也很复杂的主题，涉及到面向服务的体系结构、CI/CD、微服务、容器和云服务。事实证明，对于一些云原生构建块来说，C++出色的性能是一个受欢迎的特性。

14.7. 练习题

1. 应用程序容器与操作系统容器有何不同？
2. 在早期 UNIX 系统中，有哪些沙箱环境的例子？
3. 为什么容器很适合微服务？
4. 容器和虚拟机之间的主要区别是什么？
5. 什么时候应用程序容器是糟糕的选择？
6. 构建多平台容器映像的工具有哪些？
7. 除了 Docker，还有哪些其他的容器运行时？
8. 流行的容器协调器有哪些？

14.8. 扩展阅读

- Learning Docker - Second Edition: <https://www.packtpub.com/product/learningdocker-second-edition/9781786462923>

- Learn OpenShift: <https://www.packtpub.com/product/learnOpenshift/9781788992329>
- Docker for Developers: <https://www.packtpub.com/product/docker-for-developers/9781789536058>

第 15 章 原生云设计

云原生设计描述的是应用程序的架构，首先是在云中运行。它不是由单一的技术或语言定义的，而是利用了现代云平台提供的所有优势。

这可能意味着在必要时结合使用平台即服务 (PaaS)、多云部署、边缘计算、功能即服务 (FaaS)、静态文件托管、微服务和托管服务，超越了传统操作系统的界限。云本地开发人员使用 boto3、Pulumi 或 Kubernetes 等库和框架构建更高级的概念，而不是针对 POSIX API 和类 UNIX 操作系统。

本章将讨论以下内容：

- 了解原生云
- 使用 Kubernetes 协调云原生工作负载
- 使用服务网格连接服务
- 分布式系统中的可观测性
- 使用 GitOps

本章结束时，将很好地理解如何在应用程序中使用现代软件架构，及趋势。

15.1. 相关准备

本章中的一些例子需要 Kubernetes 1.18。

本章的代码可以在以下 GitHub 页面找到：<https://github.com/PacktPublishing/Software-Architecture-with-Cpp/tree/master/Chapter15>。

15.2. 了解原生云

虽然可以将现有的应用程序迁移到云中运行，但这种迁移不会使应用程序成为云本地应用程序。它将在云中运行，但架构选择仍基于内部部署模型。

简而言之，云原生应用程序本质上是分布式的、松散耦合的、可扩展的。它们不与任何特定的物理基础设施绑定，甚至不需要开发人员考虑特定的基础设施。这类应用程序通常以网络为中心。

本章中，将介绍一些云-本地构建块的例子，并描述一些云-本地模式。

15.2.1 原生云计算基础

云原生设计的一个支持者是云原生计算基金会 (CNCF)，它主持了 Kubernetes 项目。CNCF 是各种技术的发源地，使得构建独立于云供应商的云本地应用程序变得更加容易。这类技术的例子包括：

- Fluentd，统一的日志记录层
- Jaeger，用于分布式跟踪
- Prometheus，用于监控
- CoreDNS，用于发现服务

云本地应用程序通常是用应用程序容器构建的，运行在 Kubernetes 平台上。然而，这并不是必需的，在 Kubernetes 和容器之外使用许多 CNCF 框架完全可能。

15.2.2 操作系统——云

云原生设计的主要特点是将各种云资源视为应用程序的构建块。在 cloudnative 设计中很少使用单独的虚拟机 (VM)。使用本地云方法，不是针对运行在某些实例上的给定操作系统，而是直接针对云 API(例如，使用 FaaS) 或一些中间解决方案 (如 Kubernetes)。从这个意义上说，云可以成为操作系统，因为 POSIX API 无法进行限制。

由于容器改变了构建和发布软件的方法，现在可以从底层硬件基础设施的思考中解放出来。软件不是独立工作的，因此仍然需要连接不同的服务、监视、控制其生命周期、存储数据或传递机密。这是 Kubernetes 提供的，也是它变得如此受欢迎的原因之一。

可以想象，云原生应用程序是 Web 和移动优先的。桌面应用程序也可以从一些云本地组件中受益，但这是一种不常见的用例。

仍然可以在云本地应用程序中使用硬件和其他底层访问。如果工作负载需要使用 GPU，这不应该成为转向云本地的阻碍。更重要的是，如果想访问别处不可用的定制硬件，可以在本地构建云本地应用程序。这个术语并不局限于公有云，而是指使用不同资源的方式。

负载均衡和服务发现

负载平衡是分布式应用程序的重要组成部分，不仅在服务集群中传播传入请求 (这对于扩展至关重要)，还可以帮助提高应用程序的响应能力和可用性。智能负载均衡器可以收集指标来响应传入流量中的模式，监视其集群中服务器的状态，并将请求转发到负载较低且响应更快的节点——避免使用当前不健康的节点。

负载平衡带来更多的吞吐量和更少的停机时间。通过将请求转发到多个服务器，可以消除单点故障，特别是在使用多个负载均衡器的情况下，例如，在主动-被动方案中。

负载均衡器可以在架构的任何地方使用：可以平衡来自网络的请求，由网络服务器完成的请求到其他服务，缓存或数据库服务器的请求，以及任何适合的需求。

TIP

引入负载平衡时，有几件事需要记住。其中之一是会话持久性——确保来自同一客户的请求都发送到同一服务器，这样精心挑选的粉色细高跟鞋就不会从电子商务网站的购物篮中消失。会话在负载平衡方面可能会变得很棘手：要格外注意不要混合使用会话，这样客户就不会突然开始登录到彼此的配置文件中——之前有无数公司遇到过这个错误，特别是在将缓存添加到混合使用时。将两者结合起来是个好主意，只要确保使用正确的方法即可。

反向代理

即使只想部署服务器的一个实例，在它前面添加另一个服务 (而不是负载平衡器) 也是一个好主意——反向代理。代理通常代表发送请求的客户端进行操作，而反向代理则代表处理这些请求的服务器进行操作。

为什么要用它？使用这种代理的原因和用途如下：

- 安全性: 服务器地址现在是隐藏的, 服务器可以通过代理的 DDoS 防御功能来保护。
- 灵活性和扩展性: 可以在需要的时候修改隐藏在代理背后的环境。
- 缓存: 如果已经知道服务器会给出什么答案, 为什么还要麻烦它呢?
- 压缩: 压缩数据将减少所需的带宽, 这可能对连接不良的移动用户特别有用。它还可以降低网络成本(但可能会消耗计算能力)。
- SSL 终端: 通过承担后端服务器加密和解密网络流量的负担来减少其负载。

一个反向代理的例子是 NGINX, 还提供负载平衡功能、A/B 测试等等。它的其他功能之一是发现服务。

发现服务

发现服务 (SD) 允许自动检测计算机网络中特定服务的实例。调用者必须只指向服务注册中心, 而不是硬编码应该承载服务的域名或 IP。使用这种方法, 架构将变得更加灵活, 因为现在可以很容易地找到使用的所有服务。如果设计了一个基于微服务的架构, 引入 SD 确实需要很长一段时间。

实现可持续发展有几种方法。在客户端发现中, 调用者直接联系 SD 实例。每个服务实例都有一个注册客户端, 用于注册和注销实例、处理心跳等。虽然非常简单, 但在这种方法中, 每个客户机都必须实现服务发现逻辑。Netflix 的 Eureka 就是这种方法中服务注册中心的一个例子。

另一种方法是使用服务器端发现。这里还提供了一个服务注册中心, 以及每个服务实例中的注册中心客户机。然而, 打电话的人不会直接联系它。相反, 它们连接到负载均衡器, 例如 AWS 弹性负载均衡器, 在将客户端调用分派到特定实例之前调用服务注册表或使用其内置的服务注册表。除了 AWS ELB, NGINX 和 Consul 可以用来提供服务器端 SD 功能。

现在知道了如何有效地查找和使用服务, 以及如何良好地部署它们。

15.3. 使用 Kubernetes 协调云原生工作负载

Kubernetes 是一个可扩展的开源平台, 用于自动化和管理容器应用程序。它有时称为 k8s, 因为它以 “k” 开头, 以 “s” 结尾, 中间有八个字母。

设计基于 Google 内部使用的 Borg 系统。Kubernetes 的一些特征如下:

- 自动扩展的应用程序
- 可配置的网络
- 批处理执行
- 应用统一升级
- 具有运行高可用性应用程序的能力
- 声明式配置

在团队中有不同的方式来管理 Kubernetes, 选择其中一种需要分析与之相关的成本和收益。

15.3.1 Kubernetes 的结构

虽然可以在一台机器上运行 Kubernetes(例如, 使用 minikube、k3s 或 k3d), 但不建议在生产环境中这样做。单机集群的功能有限, 而且没有故障转移机制。Kubernetes 集群通常的大小是六

台或更多机器。然后，三个机器组成控制台。其他三个是工作节点。

3 台机器的最低要求来自于这样一个事实，这是提供高可用性的最低数量。控制台节点也可以作为工作节点使用，但不推荐这样做。

控制台

在 Kubernetes 中，很少与单个工作节点交互。相反，所有 API 请求都进入控制台。然后控制台根据请求决定采取的操作，然后它与工作节点通信。

与控制台的交互可以采取的几种形式：

- 通过 kubectl 命令行
- 使用 Web 仪表盘
- 在 kubectl 之外的应用程序中使用 Kubernetes API

控制台节点通常运行 API 服务器、调度器、配置存储 (etcd)，可能还会运行一些进程来处理特定的需求。例如，Kubernetes 集群部署在公有云 (如 Google 云平台) 上，在控制平面节点上运行云控制器。云控制器与云提供商的 API 交互，以替换故障机器、提供负载平衡器或分配外部 IP 地址。

工作节点

构成控制台和工作池的节点是工作负载将在其上运行的实际机器，可能是本地托管的物理服务器、私有托管的虚拟机或来自云提供商的虚拟机。

集群中的每个节点至少运行下面三个程序：

- 容器运行时 (例如，Docker 引擎或 cri-o)，允许机器处理应用程序容器
- kubelet，负责接收来自控制台的请求，并基于这些请求管理各个容器
- kube-proxy，负责节点级的网络和负载平衡

15.3.2 部署 Kubernetes 的方法

正如上一节中了解到的，部署 Kubernetes 有多种不同的方法。

其中之一是将其部署到本地托管的裸服务器上。这样的好处是，对于大型应用程序来说，这可能比云提供商提供的服务更便宜。这种方法有一个缺点——需要在必要时提供额外的节点。

为了缓解这个问题，可以在裸服务器上运行一个虚拟化设备。这使得使用 Kubernetes 内置的云控制器自动提供必要的资源成为可能。仍可以对成本进行控制，但手工工作减少了。虚拟化会增加一些开销，但在大多数情况下，这是一种权衡。

如果对自托管服务器不感兴趣，可以将 Kubernetes 部署在云提供商提供的虚拟机上运行。通过选择这种方式，可以使用一些现有模板进行最佳设置。有 terraform 和 Ansible 模块可以在流行的云平台上构建集群。

最后，还有来自主要云计算参与者的托管服务。只需要支付其中一些工作节点的费用，而控制台是免费的。

当在公共云中运行时，为什么要选择自托管的 Kubernetes，而不是托管的服务呢？其中一个原因是需要特定版本的 Kubernetes。云提供商在引入更新时通常会有延迟。

15.3.3 理解 Kubernetes 的概念

Kubernetes 介绍了一些概念，如果第一次听到它们，可能会感到陌生或困惑。当了解他们的目的时，应该更容易理解 Kubernetes 的特别之处。以下是 Kubernetes 最常见的一些概念：

- 容器，特别是应用程序容器，是分发和运行单个应用程序的一种方法。可以运行未修改的应用程序所需的代码和配置。
- Pod 是 Kubernetes 的基本构件，是原子的，由一个或多个容器组成。Pod 中的所有容器共享相同的网络接口、卷（如持久存储或机密）和资源（CPU 和内存）。
- 部署是描述工作负载，及其生命周期特性的高级对象。通常管理一组 Pod 副本，允许滚动升级，并在失败时管理回滚。这使得扩展和管理 Kubernetes 应用程序的生命周期变得很容易。
- DaemonSet 是一个类似于部署的控制器，它管理 Pod 分布的位置。虽然部署关注的是保持给定数量的副本，但 daemonset 将 Pod 分散到所有工作节点。主要用在每个节点上运行系统级服务，例如监视或日志记录代理。
- 作业是为一次性任务而设计的，部署中的 Pod 在容器终止时自动重启。它们适用于所有在网络端口上监听请求的 always-on 服务。但是，部署不适合批处理作业，例如缩略图生成，通常只希望在需要时运行这些作业。作业创造一个或多个 Pod，并观察他们，直到他们完成指定的任务。当成功终止特定数量的 Pod 时，作业认定为完成。
- CronJobs，是指在集群中周期性运行的作业。
- 服务表示在集群中执行的特定功能，有一个与它们相关联的网络端点（通常是负载均衡的）。服务可以由一个或多个 Pod 执行，服务的生命周期独立于许多 Pod 的生命周期。由于 Pod 是短暂的，可能在任何时候创建和销毁。服务将各个 Pod 抽象出来，以实现高可用性。为了方便使用，服务有自己的 IP 地址和 DNS 名称。

声明式方法

已经在第 9 章中讨论了声明式方法和命令式方法之间的区别。Kubernetes 采用声明式方法。提供描述集群所需状态的资源，而不是给出关于需要采取的步骤的指示。由控制平面来分配内部资源，以满足需求。

可以直接使用命令行添加资源。这可以快速进行测试，可能希望在大多数情况下都能跟踪创建的资源。因此，大多数人使用清单文件，提供所需资源的编码描述。清单通常是 YAML 文件，也可以使用 JSON。

下面是一个带有单个 Pod 的 YAML 清单示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-server
  labels:
    app: dominican-front
```

```
spec:  
  containers:  
    - name: webserver  
      image: nginx  
      ports:  
        - name: http  
          containerPort: 80  
          protocol: TCP
```

第一行是强制性的，在清单中将使用哪个 API 版本。有些资源仅在扩展中可用，因此这是解析器关于如何操作的信息。

第二行描述创建的资源。接下来是元数据和资源规范。

名称在元数据中是必需的，因为这是区分一个资源与另一个资源的方法。如果想创建另一个具有相同名称的 Pod，将得到一个错误，说明这样的资源已经存在。标签是可选的，在编写选择器时很有用。例如，想创建一个允许连接到 Pod 的服务，将使用一个值等于 dominican-front 的标签匹配应用的选择器。

规范也是必需的部分，因为它描述了资源的实际内容。示例中，列出了在 Pod 中运行的所有容器。准确地说，一个名为 webserver 的容器使用了一个来自 Docker Hub 的 nginx 镜像。因为想从外部连接到 Nginx web 服务器，所以也公开了服务器正在监听的容器端口 80。端口描述中的名称为可选参数。

15.3.4 Kubernetes 网络

Kubernetes 支持可插拔网络架构，有几种驱动程序可以根据需求使用。无论选择哪种驱动程序，有些概念是通用的。以下是常规的组网场景。

容器对容器的通信

一个 Pod 可以容纳几个不同的容器。由于网络接口绑定到 Pod，而不是容器，因此每个容器都在相同的网络名称空间中操作。这意味着各种容器可以使用本地主机网络彼此寻址。

Pod 对 Pod 的通信

每个 Pod 都分配一个内部集群本地 IP 地址。当 Pod 删除了，地址就不存在了。当一个 Pod 知道另一个的地址时，因为它们共享相同的网络，所以可以连接到另一个暴露的端口。对于这个通信模型，可以将 Pod 看作是托管容器的 VM。这种方法很少使用，因为首选的方法是 Pod 到服务的通信。

Pod 对服务的通信

Pod 对服务通信是集群内最流行的通信用例。每个服务都有一个单独的 IP 地址和分配给它的 DNS。当一个 Pod 连接到服务时，该连接被代理到由服务选择的组中的一个 Pod。代理是前面描

述的 kube-proxy 工具的任务。

外部对内部的通信

外部流量通常通过负载均衡器到达集群。它们要么绑定到特定的服务或入口控制器，要么使用特定的服务或入口控制器处理。当外部公开的服务处理通信时，行为类似于 Pod 到服务的通信。有了入口控制器，就有了其他可用的特性，允许路由、可观察性或高级负载平衡。

15.3.5 何时使用 Kubernetes

在组织中引入 Kubernetes 需要一些成本。Kubernetes 提供了许多好处，例如自动可扩展性、自动化或部署场景。然而，这些好处可能不足以证明必要的成本是合理的。

这项投资成本涉及几个领域：

- 基础设施成本：与运行控制台和工作节点相关的成本可能相对较高。此外，若希望使用各种 Kubernetes 扩展，例如 GitOps 或服务网格（稍后将介绍），那么成本可能会增加。还需要额外的资源来运行，并在应用程序的常规服务之上提供更多的开销。除了节点本身，还应该考虑其他成本。Kubernetes 的一些特性在部署到受支持的云提供商时工作得最好。这意味着为了从这些功能中受益，必须遵循以下方式：
 - a 将工作负载转移到特定支持的云上
 - b 为选择的云提供商，实现自己的驱动程序
 - c 将基础设施迁移到一个虚拟化的支持 API 的环境，如 VMware vSphere 或 OpenStack。
- 运营成本：Kubernetes 集群和相关服务需要维护。尽管应用程序的维护减少了，但保持集群运行的成本略微抵消了这一好处。
- 学习成本：整个产品团队必须学习新的概念。即使有一个专业的平台团队，为开发人员提供易于使用的工具，开发人员仍然需要了解所做的工作如何影响整个系统，以及应该使用哪些 API。

引入 Kubernetes 之前，首先考虑是否能够负担它所需的成本。

15.4. 分布式系统中的可观测性

分布式系统（如云本机架构）带来了一些独特的挑战。在给定时间工作的不同服务的数量之多，使得研究组件的性能十分不便。

在单系统中，日志记录和性能监控通常就足够了。对于分布式系统，甚至日志记录也需要一个设计选择。不同的组件产生不同的日志格式。那些日志总得有个地方存放。将它们与交付它们的服务结合在一起，将很难在宕机情况下了解全局状态。此外，由于微服务的生命周期可能很短，可能希望将日志的生命周期与提供日志的服务或承载该服务的机器的生命周期分离开来。

在第 13 章中，描述了统一的日志记录层如何帮助管理日志。但是日志只显示系统中给定点发生的事情。要从单个事务的角度查看问题，需要使用不同的方法。

这时就需要进行跟踪了。

15.3.1 跟踪与日志记录有何不同

跟踪是一种特殊形式的日志记录，提供的信息级别比日志低。包括所有函数调用、参数、大小和执行时间。还包含正在处理的事务的 ID。通过这些细节，可以重新组合，并查看给定事务通过系统时的生命周期。

跟踪中显示的性能信息，可以发现系统中的瓶颈和次优组件。

日志通常由操作人员和开发人员读取，所以它们是人类可读的，对跟踪没有这样的要求。要查看轨迹，需要使用专用的可视化程序。这意味着即使跟踪更详细，也可能比日志占用更少的空间。

下面的图表是单个跟踪的概述：

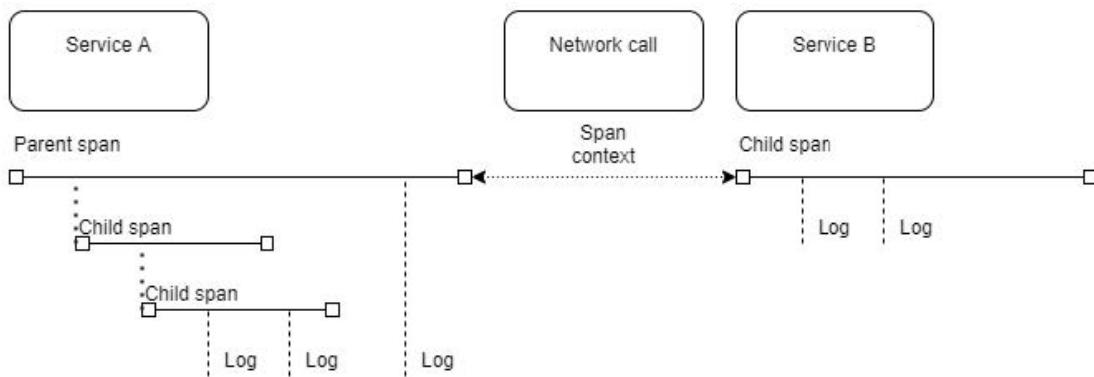


图 15.1 - 单个跟踪

两个服务在一个网络上通信。在服务 A 中，有一个父跨度，它包含一个子跨度和一个日志。子跨度通常对应于更深层的函数调用，日志表示最小的一条信息。每一个都是计时的，并且可能包含额外的信息。

对服务 B 的网络调用保存了时间的上下文。即使服务 B 是在另一台机器上的不同进程中执行的，因为事务 ID 保留，所有信息都可以在稍后重新加载。

从重新加载跟踪中获得的一个额外信息是，分布式系统中服务之间的依赖关系图。由于跟踪包含整个调用链，因此可以将此信息可视化并检查意外依赖项。

15.2.2 选择跟踪解决方案

实现跟踪时，有几种可能的解决方案可供选择，可以使用自托管和托管工具来测试应用程序。我们将简要介绍托管类，并重点介绍自托管类。

Jaeger 和 OpenTracing

分布式跟踪的标准之一是 Jaeger 的作者提出的 OpenTracing。Jaeger 是为云本地应用程序构建的跟踪程序，解决了监视分布式事务和传播跟踪上下文的问题。它的用途如下：

- 性能或延迟优化
- 执行原因分析
- 分析服务间的依赖关系

OpenTracing 是一个开放标准，提供了一个独立于所使用的跟踪程序的 API。这意味着当应用程序使用 OpenTracing 测试时，可以避免锁定在一个特定的供应商。如果在某些时候，可以从

Jaeger 切换到 Zipkin、DataDog 或任何其他兼容的跟踪程序，从而不必修改整个检测代码。

有许多与 OpenTracing 兼容的客户端库。还可以找到许多资源，包括解释如何根据需要实现 API 的教程和文章。OpenTracing 官方支持以下语言：

- Go
- JavaScript
- Java
- Python
- Ruby
- PHP
- Objective-C
- C++
- C#

还有一些非官方的库可用，特定的应用程序也可以导出 OpenTracing 数据。这包括 Nginx 和 Envoy，这两种流行的 Web 代理。

Jaeger 也接受 Zipkin 格式的样本，将在下一节介绍 Zipkin。如果（或依赖项）已经使用 Zipkin，则无需将测试从一种格式改写为另一种格式。对于所有新应用程序，建议采用 OpenTracing 方法。

Jaeger 的扩展性也不错。如果想对其进行计算，可以将其作为单个二进制文件或单个应用程序容器运行。可以为生产环境配置 Jaeger，使用它自己的后端或支持的外部后端，如 Elasticsearch，Cassandra 或 Kafka。

Jaeger 是一个 CNCF 的毕业项目，已经达到了与 Kubernetes、Prometheus 或 Fluentd 相似的成熟水平。正因为如此，我们希望它在其他 CNCF 应用中得到更多的支持。

Zipkin

Jaeger 的主要竞争对手是 Zipkin。这是一个较老的项目，这也意味着它更成熟。通常，更高级的项目也会得到更好的支持，但 CNCF 的支持对 Jaeger 有利。

Zipkin 使用其专有协议来处理跟踪，提供了 OpenTracing 支持，但是它的成熟度和支持级别可能比不上原生的 Jaeger 协议。正如前面提到的，还可以配置 Jaeger 以 Zipkin 格式收集跟踪。说明这两者至少在某种程度上是可以互换的。

该项目由 Apache 基金会托管，但不认定为 CNCF 项目。在开发云原生应用程序时，Jaeger 是一个更好的选择。如果正在寻找一种通用的跟踪解决方案，Zipkin 也是值得尝试。

Zipkin 的一个缺点是没有 C++ 实现。有非官方的库，但它们似乎没有得到很好的支持。使用 C++ 的 OpenTracing 库是测试 C++ 代码的首选方法。

15.2.3 使用 OpenTracing 测试应用程序

本节将演示如何将 Jaeger 和 OpenTracing 插装添加到现有的应用程序中，这里将使用 opentrace-cpp 和 jaeger-client-cpp 库。

首先，想要设置跟踪器：

```
1 #include <jaegertracing/Tracer.h>
2
```

```

3 void setUpTracer()
4 {
5     // We want to read the sampling server configuration from the
6     // environment variables
7     auto config = jaegertracing::Config;
8     config.fromEnv();
9     // Jaeger provides us with ConsoleLogger and NullLogger
10    auto tracer = jaegertracing::Tracer::make(
11        "customer", config, jaegertracing::logging::consoleLogger());
12    opentracing::Tracer::InitGlobal(
13        std::static_pointer_cast<opentracing::Tracer>(tracer));
14 }

```

配置采样服务器的两种首选方法是使用环境变量，或者使用 YAML 配置文件。在使用环境变量时，必须在运行应用程序之前设置它们。需要注意以下几点：

- JAEGER_AGENT_HOST: Jaeger 代理所在的主机名
- JAEGER_AGENT_PORT: Jaeger 代理正在监听的端口
- JAEGER_SERVICE_NAME: 应用程序的名称

接下来，配置跟踪程序并提供日志记录实现。如果可用的 ConsoleLogger 不够用，则可以实现自定义日志记录解决方案。对于具有统一日志记录层的基于容器的应用程序，ConsoleLogger 应该足够了。

当设置好跟踪程序后，希望将跨度添加到想要检测的函数中。下面的代码就是这样做的：

```

1 auto responder::respond(const http_request &request, status_code status,
2 const json::value &response) -> void {
3     auto span = opentracing::Tracer::Global()->StartSpan("respond");
4     // ...
5 }

```

稍后可以使用此跨度在给定函数中创建子跨度，也可以作为参数传播到更深层的函数调用：

```

1 auto responder::prepare_response(const std::string &name, const
2 std::unique_ptr<opentracing::Span>& parentSpan)
3 -> std::pair<status_code, json::value> {
4     auto span = opentracing::Tracer::Global()->StartSpan(
5         "prepare_response", { opentracing::ChildOf(&parentSpan->context())
6     });
7     return {status_codes::OK,
8         json::value::string(string_t("Hello, ") + name + "!"});
9 }
10
11 auto responder::respond(const http_request &request, status_code status)
12 -> void {
13     auto span = opentracing::Tracer::Global()->StartSpan("respond");
14     // ...
15     auto response = this->prepare_response("Dominic", span);
16     // ...
17 }

```

当调用 `opentracing::ChildOf` 函数时，就会发生上下文传播。还可以使用 `inject()` 和 `extract()` 调用通过网络调用传递上下文。

15.5. 使用服务网格连接服务

微服务和原生云设计也有自己的问题。即使服务数量有限，不同服务之间的通信、可观察性、调试、速率限制、认证、访问控制和 A/B 测试也可能具有挑战性。当服务的数量增加时，上述需求的复杂性也会增加。

这就是服务网格进入的地方。简而言之，服务网格权衡了一些资源（运行控制平面和 sidecars 所必需的），以实现对上述挑战的自动化和集中控制的解决方案。

15.5.1 引入服务网格

本章的介绍中，提到的所有需求过去都是在应用程序本身中编码的。事实证明，许多可能可以抽象，因为它们在许多不同的应用程序之间共享。当应用程序包含许多服务时，为所有服务添加新特性的成本将会很高。使用服务网格，可以控制这些特性。

因为容器化的工作流已经对一些运行时和网络进行了抽象，服务网格将这种抽象提升到了另一个层次。这样，容器中的应用程序只知道 OSI 网络模型的应用程序级别发生了什么，而服务网格处理的任务较为底层。

设置服务网格可以以一种新的方式控制所有的网络流量，并更好地了解这些流量，依赖关系变得可见。

不仅是由服务网格处理的流量。其他流行的模式，如电路断路、速率限制或重试，不必由每个应用程序实现并单独配置，这也是一个可以外包给服务网格的特性。类似地，A/B 测试或金丝雀部署是服务网格能够实现的用例。

如前所述，服务网格的好处之一是更好的控制。架构通常由外部流量的可管理边缘代理和内部代理组成，这些代理通常部署在每个微服务上。通过这种方式，可以将网络策略编写为代码，并与所有其他配置一起存储在一个位置。只需在服务网格配置中启用该功能一次，而不必为想要连接的两个服务开启相互 TLS 加密。

15.5.2 网格服务解决方案

这里描述的所有解决方案都是自托管的。

Istio

Istio 是一个强大的服务网格工具集合，可以通过将 Envoy 代理部署为 sidecar 容器来连接微服务。因为 Envoy 是可编程的，所以 Istio 控制平面的配置更改会与所有代理通信，然后这些代理会相应地重新配置自己。

Envoy 代理还负责提供加密和身份验证。使用 Istio，在服务之间启用相互 TLS 在大多数时间需要在配置中添加一个交换机。如果不希望在所有服务之间使用 mTLS，也可以选择那些需要这种额外保护的服务，同时允许在其他所有服务之间使用未加密的通信。

Istio 还可以提供可观察性。首先，Envoy 代理导出与 Prometheus 兼容的代理级指标，Istio 导出了服务级别指标和控制台指标。接下来，有描述网格内流量的分布式轨迹。Istio 可以为不同的后端提供跟踪:Zipkin，Jaeger，Lightstep 和 Datadog。最后，还有 Envoy 访问日志，类似于 Nginx 的格式显示每个调用。

这是可能的可视化网格使用 Kiali，一个交互式的网络界面。通过这种方式，可以看到服务的图表，包括加密是否启用、不同服务之间的流大小或每个服务的健康检查状态等信息。

Istio 的作者声称这个服务网格应该与不同的技术兼容。在撰写本文时，具有最好的文档、最好的集成和最好的测试是与 Kubernetes 的集成。其他受支持的环境有 on-premises、通用云、Mesos 和 Nomad with Consul。

如果在与合规相关的行业（如金融机构）工作，那么 Istio 可以在这些方面提供帮助。

Envoy

Envoy 本身并不是一个服务网格，但由于它在 Istio 中使用，所以在本节进行介绍。

Envoy 是一个服务代理，作用很像 Nginx 或 HAProxy，主要的区别在于可以动态地重新配置。这是通过 API 以编程方式实现的，不需要更改配置文件，也不需要重新加载守护进程。

关于特使有趣的事是它的性能和知名度。根据 SolarWinds 的测试，Envoy 在作为服务代理的性能方面优于其他竞争对手，这包括 HAProxy，Nginx，Traefik 和 AWS 应用负载均衡器。Envoy 比 Nginx、HAProxy、Apache 和 Microsoft IIS 等已经建立的领导者要年轻得多，但根据 Netcraft 的数据，这并不妨碍 Envoy 进入最常用 Web 服务器的前 10 名。

Linkerd

在 Istio 成为服务网格的代名词之前，这个领域的代表是 Linkerd。其在命名方面有一些混淆，因为最初的 Linkerd 项目设计为平台无关的，并且以 Java VM 为目标。这意味着它的重资源，而且经常行动迟缓，新版本 Linkerd2 已经重写以解决这些问题。与最初的 Linkerd 不同，Linkerd2 只关注 Kubernetes。

Linkerd 和 Linkerd2 都使用自己的代理解决方案，而不是依赖于像 Envoy 这样的现有项目。这样做的理由是专用代理（与通用 Envoy 相比）提供更好的安全性和性能。Linkerd2 的一个有趣特性是，开发它的公司还提供付费支持版本。

领事服务网格

服务网格空间最近增加了 Consul 服务网格。这是 HashiCorp 的产品，HashiCorp 是一家知名的云计算公司，以 terraform、Vault、Packer、Nomad 和 Consul 等工具而闻名。

就像其他解决方案一样，具有 mTLS 和流量管理功能。宣传为多云、多数据中心和多区域网格。集成了不同的平台、数据平面产品和可观察性提供商。在撰写本文时，实际情况要好一些，因为主要支持的平台是 Nomad 和 Kubernetes，而支持的代理要么是内置代理，要么是 Envoy。

如果考虑在应用程序中使用 Nomad，那么 Consul 服务网格可能是一个很好的选择，因为它们都是 HashiCorp 的产品。

15.6. 使用 GitOps

本章中要讨论的最后一个主题是 GitOps。尽管这个词听起来新颖时髦，但它背后的理念并不完全新颖。它是著名的持续集成/持续部署 (CI/CD) 模式的扩展。或者可能扩展不是一个好的描述。

CI/CD 系统的目标通常是非常灵活，而 GitOps 则寻求尽可能减少可能的集成数量。两个主要常量是 Git 和 Kubernetes。Git 主要用于版本控制、发布管理和环境分离。Kubernetes 用作标准化和可编程的部署平台。

这样，CI/CD 流水就变得几乎透明了，它与命令式代码处理构建的所有阶段的方法相反。为了允许这样的抽象级别，通常需要了解以下几点：

- 将基础设施作为代码，以允许所有必要的自动化部署
- Git 工作流，带有特性分支和拉请求或合并请求
- 声明式工作流配置，Kubernetes 中已经有了

15.6.1 GitOps 的原则

由于 GitOps 是已建立的 CI/CD 模式的扩展，因此无法很清楚地区分两者。以下是将这种方法与通用 CI/CD 区别开来的一些 GitOps 原则。

声明性描述

经典 CI/CD 系统与 GitOps 的主要区别在于操作方式。大多数 CI/CD 系统都是必要的：它们包含一系列步骤，以便流水成功。

甚至连流水的概念都是必要的，因为它是一个具有条目、一组连接和一个接收器的对象。有些步骤可以并行执行，但是只要存在依赖项，流程就必须停止并等待依赖项步骤完成。

在 GitOps 中，配置是声明性的。这指的是系统的整个状态——应用程序、配置、监视和指示盘。其会视为代码，具有与常规应用程序代码相同的特性。

使用 Git 版本控制系统的状态

由于系统的状态是用代码编写的，因此可以从中获得一些好处。更容易的审核、代码审查和版本控制等特性现在不仅适用于应用程序代码。结果是若出现错误，恢复到工作状态需要使用 `git revert`。

可以使用 Git 的签名提交以及 SSH 和 GPG 密钥的强大功能来控制不同的环境。通过添加限制机制来确保，只有符合所需标准的提交才能推送到存储库，还消除了许多由于使用 ssh 或 kubectl 手动运行命令而导致的意外错误。

可审计

存储在版本控制系统中的所有内容都变得可审计。在引入新代码之前，要进行代码评审。当注意到一个 bug 时，可以还原导致 bug 的更改，或者返回到上一个可工作版本。存储库成为关于整个系统的关键。

当应用于应用程序代码时，可以将功能扩展到审计配置、助手服务、指标、仪表板，甚至部署

策略，使其更加强大。不再需要问自己：“好吧，为什么这种配置最终会用于生产呢？”所要做的就是检查 Git 日志。

与已存在的组件进行集成

大多数 CI/CD 工具引入专有的配置语法。Jenkins 使用 Jenkins DSL。每个流行的 SaaS 解决方案都使用 YAML，但是 YAML 文件彼此不兼容。在必须重写流水的情况下从 Travis 切换到 CircleCI，或从 CircleCI 切换到 GitLab CI。

这有两个缺点。其一是明显的供应商锁定。另一个是需要学习使用给定工具的配置语法。即使大多数流水已经在其他地方定义（shell 脚本、Dockerfiles 或 Kubernetes 清单），仍然需要编写一些粘合代码来指示 CI/CD 工具使用它。

这与 GitOps 不同，不需要编写显式指令或使用专有语法。相反，可以重用其他通用标准，如 Helm 或 Kustomize。需要学习的东西更少，迁移过程也更舒适。此外，GitOps 工具通常与 CNCF 生态系统中的其他组件集成得很好，因此可以将部署指标存储在 Prometheus 中，并使用 Grafana 进行审计。

防漂配置

当给定系统的当前状态与存储库中描述的期望状态不同时，就会发生配置漂移。多种原因导致了配置漂移。

例如一个配置管理工具，它具有基于 VM 的工作负载，所有虚拟机以相同的状态启动。当 CM 第一次运行时，将机器运行至所需的状态。但是，如果自动更新代理在默认情况下运行在这些机器上，那么该代理可能会自己更新一些包，而不考虑 CM 期望的状态。此外，由于网络连接可能很脆弱，一些机器可能会更新到一个软件包的新版本，而另一些则不会。

极端情况下，某个更新的包可能与应用程序所需的固定包不兼容。这种情况将破坏整个 CM 工作流，使机器处于不可用的状态。

使用 GitOps，始终在系统中运行代理，跟踪系统的当前状态和期望状态。如果当前状态与期望的状态突然不同，代理可以修复它或发出有关配置漂移的警报。

防止配置漂移为系统增加了另一层自修复功能。如果在运行 Kubernetes，就已经有了自愈功能。每当一个 Pod 失效时，另一个 Pod 就会在其位置上重新创建。如果在底层使用可编程的基础设施（例如云提供商或 OpenStack 本地部署），那么节点还具有自修复功能。通过 GitOps，可以自愈工作负载及其配置。

15.6.2 GitOps 的好处

GitOps 的上述特性提供了几个好处，以下是其中一些。

提高生产率

CI/CD 流水已经自动化了许多常规任务，通过帮助获得更多部署来缩短交付时间。GitOps 添加了一个反馈循环，防止配置漂移，并允许自修复。开发者和其团队可以更快地交付，因为它们很容易恢复，不用担心引入问题。反过来，开发吞吐量增加，则可以更快地、更有信心引入新特性。

更好的开发体验

使用 GitOps，开发人员不必担心构建容器或使用 kubectl 来控制集群。部署新特性只需要使用 Git，在大多数环境中已经是一个熟悉的工具。

新员工的上手速度更快，因为不需要为了提高效率而学习很多新工具。GitOps 使用标准和一致的组件，因此在操作端引入更改不会影响开发人员。

更高的稳定性和可靠性

使用 Git 存储系统状态意味着可以访问审计日志，该日志包含所有引入的更改的描述。如果任务跟踪系统与 Git 集成（这是一个很好的实践），通常可以分辨出哪个业务特性与系统的更改相关。

使用 GitOps，不太需要允许手动访问节点或整个集群，这减少了由于运行无效命令而产生意外错误的机会。通过使用 Git 强大的恢复特性，可以很容易地修复进入系统的那些随机错误。

从严重的灾难（如失去整个控制台）中恢复也容易得多，所需要的是建立一个新的干净的集群，在那里安装一个 GitOps 操作符，并将其指向带有配置的存储库。过一会儿，就会得到了之前生产系统的副本，这一切都不需要人工干预。

安全性提高

减少了对集群和节点的访问，这意味着提高了安全性。就秘钥丢失或被盗而言，不用太担心了。为了避免了这样一种情况：即使某人不再在团队（或公司）工作，仍然保留对生产环境的访问权。

当涉及到对系统的访问时，单点是由 Git 存储库处理的。即使恶意的参与者决定在系统中引入后门，所需要的更改也将经过代码审查。当存储库使用带有强验证的 GPG 签名提交时，模仿其他开发人员的难度更大。

目前为止，主要讨论了开发和操作的好处，GitOps 也给企业带来了好处。它提供了系统中的业务可观察性，以前这是很难实现的。

跟踪给定版本中的特性很容易，因为都存储在 Git 中。由于 Git 提交了任务跟踪器的链接，业务人员可以获得预览链接，以查看应用程序在不同开发阶段的表现。

还提供了回答以下常见问题的答案：

- 生产中运行的是什么？
- 上一个版本解决了哪些问题？
- 哪些更改可能导致服务降级？

所有这些答案的问题可以在一个友好的仪表盘中显示。当然，仪表盘本身也可以存储在 Git 中。

15.6.3 GitOps 工具

GitOps 是一个新兴且不断增长的领域，有些工具可以认为是稳定和成熟的。以下是一些最受欢迎的工具。

FluxCD

FluxCD 是 Kubernetes 的 GitOps 运营商，选定的集成提供核心功能。它使用 Helm chart 和 Kustomize 来描述资源。

它与 Prometheus 的集成成为部署过程增加了可观察性。为了帮助维护，FluxCD 提供了一个 CLI。

ArgoCD

与 FluxCD 不同的是，提供了更广泛的工具选择。如果已经在配置中使用 Jsonnet 或 Ksonnet，这可能会有用。与 FluxCD 一样，它集成了 Prometheus，并具有 CLI 功能。

撰写本书时，ArgoCD 是比 FluxCD 更受欢迎的解决方案。

Jenkins X

与名字所暗示的相反，Jenkins X 与著名的 Jenkins CI 系统并没有太多的共同之处。由同一家公司支持，但 Jenkins 和 Jenkins X 的整个概念完全不同。

其他两个工具都是小而独立的，而 Jenkins X 是一个复杂的解决方案，有很多集成和更广泛的范围。支持自定义构建任务的触发，使它看起来像一个经典的 CI/CD 系统和 GitOps 之间的桥梁。

15.7. 总结

恭喜你读完了这一章！使用现代 C++ 并不局限于理解最近添加的语言特性。应用程序将在生产环境中运行。作为架构师，确保运行时环境匹配需求也是工作的一项内容。在前几章中，描述了分布式应用程序的一些流行趋势。希望这些知识将帮助读者们决定，哪一个是最适合自己的产品。

使用云本地会带来很多好处，并且可以自动化工作流程。将定制工具转换为行业标准会使软件更有弹性、更容易更新。本章中，已经讨论了流行的云本地解决方案的优点、缺点和用例。

有些，比如使用 Jaeger 的分布式跟踪，为大多数项目带来了直接的好处。其他的，如 Istio 或 Kubernetes，在大规模“作战”中表现最好。阅读本章之后，应该有足够的知识来决定在自己的应用程序中引入原生云设计是否可行。

15.8. 练习题

1. 云中运行应用程序和让它们成为云本地应用程序之间有什么区别？
2. 如何在本地运行云本地应用程序？
3. Kubernetes 的最小高可用集群大小是多少？
4. 哪个 Kubernetes 对象代表一个允许网络连接的微服务？
5. 为什么日志记录在分布式系统中是不够的？
6. 服务网格是如何帮助构建安全系统的？
7. GitOps 是如何提高生产力的？
8. CNCF 监控的标准项目是什么？

15.9. 扩展阅读

- Mastering Kubernetes: <https://www.packtpub.com/product/masteringkubernetes-third-edition/9781839211256>

- Mastering Distributed Tracing: <https://www.packtpub.com/product/masteringdistributed-tracing/9781788628464>
- Mastering Service Mesh: <https://www.packtpub.com/product/masteringservice-mesh/9781789615791>

附录 A

感谢在软件架构的旅程中走了这么远！我们写这本书的目的是帮助您对您的应用程序和系统的设计做出明智的决定。到目前为止，在决定是选择 IaaS、PaaS、SaaS 还是 FaaS 时，您应该更有信心了。

有很多东西我们在这本书中都没有涉及，因为它们超出了这本书的范围。或是我们对给定的主题缺乏经验，要么我们认为它太小众了。还有一些我们觉得非常重要的地方，但我们在章节中找不到合适的位置。各位读者可以在这个附录中找到它们。

设计数据存储

现在让我们讨论应用程序的存储。首先来决定是使用 SQL、NoSQL 还是其他什么。

一个经验法则是，根据数据库的大小来决定技术。对于小型数据库，例如那些大小永远不会增长到 TB 的数据库，使用 SQL 是一种有效的方法。如果有一个非常小的数据库或想要创建一个内存缓存，可以尝试 SQLite。如果计划使用 1 个 TB，并且保证不会超过这个值，那么最好的选择是使用 NoSQL。某些情况下，仍然可以使用 SQL 数据库，但是由于硬件成本的原因，很快就会变得昂贵，因为需要为主节点使用一个庞大的服务器。即使这不是问题，也应该衡量性能是否足以满足需求，并为长期维护做好准备。某些情况下，使用 Citus(本质上是一个分片的 PostgreSQL) 等技术来运行 SQL 机器集群也比较适合。但在这种情况下，通常使用 NoSQL 更便宜、更简单。如果数据库的大小超过 10TB，或者需要实时获取数据，请考虑使用数据仓库而不是 NoSQL。

应该使用哪种 NoSQL 技术？

这个问题的答案取决于几个因素。这里列出了一些：

- 如果想要存储时间序列（以较小的、定期的间隔保存增量），那么最好的选择便是使用 InfluxDB 或 victoria metrics。
- 如果需要类似 SQL 但不需要连接，或者换句话说，如果计划将数据存储在列中，可以尝试使用 Apache Cassandra、AWS DynamoDB 或 Google 的 BigTable。
- 如果不是这样，那么应该考虑数据是否是没有模式的文档，比如 JSON 或某种应用程序日志。如果是这种情况，可以使用 Elasticsearch，它非常适合灵活的数据，并提供 RESTful API。也可以试试 MongoDB，它以二进制 JSON (BSON) 格式存储数据，并允许使用 MapReduce。

好！但是如果不想存储文档呢？可以选择对象存储，尤其是在数据很大的情况下。通常，在这种情况下，与云提供商合作是不错的选择，这意味着使用 Amazon 的 S3 或 Google 的云存储，或 Microsoft 的 Blob 存储应该比较合适。如果想使用本地的工具，可以使用 OpenStack 的 Swift 或者部署 Ceph。

如果文件存储也不是想要的，那么可能只是存储简单的键值数据，使用这种存储方式的好处是速度快。这就是为什么很多分布式缓存都使用它来构建的原因。值得注意的技术包括 Riak、Redis 和 Memcached(最后一个不适合持久化数据)。

除了前面提到的选项之外，还可以考虑使用基于树的数据库，如 BerkeleyDB。这些数据库基本上是特殊的键值存储，具有类似路径的访问。如果树对目前的需求限制太多，可能会对面向图形的数据库感兴趣，如 Neo4j 或 OrientDB。

无服务器架构

虽然与云原生设计有关，但无服务器架构本身是一个热门话题。自从引入了 AWS Lambda、AWS Fargate、Google Cloud Run 和 Azure Functions 等 FaaS 或 CaaS 产品后，就变得非常受欢迎。

无服务主要是像 Heroku 这样的 PaaS 产品的发展。它抽象了底层基础设施，这样开发人员就可以专注于应用程序，而不是基础设施的选择。

与旧的 PaaS 解决方案相比，无服务器的另一个好处是不必为不使用的东西付费。通常不是为给定的服务水平付费，而是为部署的无服务器工作负载的实际执行时间付费。如果只想每天运行一段特定的代码，则不需要每月为底层服务器支付费用。

虽然没有详细介绍关于无服务器的内容，但它很少与 C++一起使用。谈到 FaaS，目前只有 AWS Lambda 可能支持 C++。由于容器是语言无关的，所以可以使用 C++应用程序和函数与 CaaS 产品（如 AWS Fargate、Azure 容器实例或 Google Cloud Run）一起使用。

若想运行与 C++应用程序一起使用的非 C++的辅助代码，那么无服务器函数可能是一个不错的选择。维护任务和计划作业非常适合无服务器环境，它们通常不依赖 C++二进制的性能或效率。

文化与交流

本书的重点是软件架构，为什么要一本关于软件的书中提到文化和交流呢？仔细想想，所有的软件都为人编写的。人性是要考虑，但我们常常不承认这一点。

作为一名架构师，您的角色不是找出解决给定问题的最佳方法。您还必须与您的团队成员交流您提出的解决方案。通常情况下，所做的选择将源于之前的交流。

所以，沟通和团队文化在软件架构中也发挥作用。

前面的章节中，提到了康威定律。这条法则指出软件系统的架构反映了从事该工作的组织。这意味着，打造伟大的产品需要建立伟大的团队和理解心理学。

如果想成为一名伟大的架构师，学习人际交往技能可能与学习技术技能一样重要。

DevOps

本书中多次使用了 DevOps(和 DevSecOps) 这个术语。在我们看来，这个话题值得一说。DevOps 是一种构建软件产品的方法，打破了传统的基于塔式的开发。

瀑布模型中，团队相互独立地操作单个工作方面。开发团队会编写代码，QA 会测试并验证代码，然后是安全性和合规性。最终，运营团队将负责维护工作。团队很少沟通。即使这样，也是一个非常正规的过程。

关于特定领域的专业知识，只对负责工作流中给定部分的团队有效。开发者对 QA 知之甚少，对运营更是一无所知。虽然这种设置非常方便，但现代开发方式需要的灵活性，要比瀑布模型高得多。

这就是为什么提出了一种新的工作模式，它鼓励在软件产品的不同涉众之间进行更多的协作、更好的交流和大量的知识共享。虽然 DevOps 指的是把开发人员和运营人员聚集在一起，但其本意是让每个人都更了解他人的工作。

开发人员甚至在编写第一行代码之前，就开始从事 QA 和安全性工作。操作工程师更熟悉代码库。企业可以轻松地跟踪票据的进度。某些情况下，甚至可以以自助方式进行部署和预览。

DevOps 已经成为使用 Terraform 或 Kubernetes 等特定工具的代名词。但 DevOps 绝不等同于使用特定的工具。你的组织可以在不使用 Terraform 或 Kubernetes 的情况下遵循 DevOps 原则，也可以在不使用 DevOps 的情况下使用 Terraform 和 Kubernetes。

DevOps 的原则之一是，鼓励改进产品利益相关者之间的信息流。有了这些，就有可能实现另一个原则：减少对最终产品没有价值的浪费活动。

当构建现代系统时，使用现代方法是值得的。将现有的组织迁移到 DevOps 可能需要大量的思维转变，所以这并不总可行。当开始一个可以控制的新项目时，这时是可以的。

练习答案

第 1 章

1. 为什么要关心软件架构?

- 架构可以保证实现和维护软件的必要质量。注意和关心它可以防止项目拥有随意架构，从而保证质量，防止软件衰退。

2. 架构师应该成为敏捷团队的最终决策者吗?

- 不。敏捷就是授权给整个团队。架构师将他们的经验和知识带到桌面上，但如果决定必须被整个团队接受，应该对其进行决策，而不是架构师一言堂。利益相关者的需求，在这里也非常重要。

3. SRP 与内聚有什么关系?

- 遵循 SRP 会带来更好的内聚。如果一个组件开始具有多个职责，通常会变得不那么具有内聚性。这种情况下，最好将其重构为多个组件，每个组件具有单一的职责。通过这种方式，增加了内聚性，因此代码变得更容易理解、开发和维护。

4. 项目生命周期的哪些阶段? 架构师可以使项目更好?

- 从项目开始到项目维护，架构师可以为项目带来价值。项目开发的早期阶段可以实现最大的价值，这是决定项目外观的关键。然而，这并不意味着架构师在开发过程中没有价值，他们可以使项目保持在正确的轨道上。通过帮助决策和监督项目，确保代码不会以随意架构，也不会受到软件退化的影响。

5. 遵循 SRP 有什么好处?

- 遵循 SRP 的代码更容易理解和维护，这也意味着 bug 更少。

第 2 章

1. REST 式服务的特点是什么?

- 显然，有 REST API 的使用。
- 无状态——每个请求包含其处理所需的所有数据。请记住，这并不意味着 REST 式服务不能使用数据库。
- 使用 cookie，而不是保持会话

2. 可以使用什么工具包来创建自愈分布式体系结构?

- Netflix 的猿猴军团

3. 微服务应该使用集中存储吗? 原因?

- 微服务应该使用去中心化存储。每个微服务都应该选择最适合自己的存储类型，因为这样可以提高效率和可扩展性。

4. 什么时候应该编写有状态服务，而不是无状态服务？

- 无状态不合理，且不需要扩展的时候。例如，当客户端和服务必须保持它们的状态同步时，或者当要发送的状态非常大时。

5. 代理与中介拓扑有什么不同？

- 中介者在服务之间进行“调解”，因此需要知道如何处理每个请求。代理只知道将每个请求发送到哪里，因此它是一个轻量级组件，可用于创建发布-订阅（发布-订阅）架构。

6. N-tier 架构和 N-layer 架构有什么区别？

- 英文里面有两个不同的概念 N-Tier 和 N-Layer，N-Tier 指不同系统（一般为不同物理系统）互相协作的架构。而 N-Layer 指一个系统内部不同模块的结构。N-Tier 为物理分层概念，而 N-Layer 为逻辑分层概念。
- 层（Layer）是逻辑的，并指定如何组织代码。层（Tier）是物理的，它指定如何运行代码。每一层都必须由其他层隔开，要么在不同的进程中运行，要么甚至在不同的机器上运行。

7. 如何使用基于微服务的架构来取代单体架构？

- 增量，在大公司中开辟小型微服务。可以使用在第 4 章中描述的扼杀器模式。

第 3 章

1. 什么是质量属性？

- 系统可能具有的特征或品质通常称为“能力”，因为它们中的许多名字中都有这个后缀，例如：可移植性。

2. 收集需求时应该使用哪些资源？

- 系统的上下文、现有的文档和系统的相关方。

3. 如何判断某个需求在架构上是否重要？

- 架构上的重要需求（ASR）通常需要一个单独的软件组件，影响系统的大部分，很难实现，并且/或迫使开发者做出权衡。

4. 应该如何以图形化的方式记录各方对系统的功能性需求？

- 准备一个用例图。

5. 何时开发视图文档？

- 如果正在开发一个带有许多模块的大型系统，并且需要与所有软件团队沟通全局约束和通用设计选择。

6. 如何自动检查代码的 API 文档是否过期？

- Doxygen 有内置的检查，比如在注释中警告函数签名和参数不匹配

7. 如何在图中显示，特定的操作由系统的不同组件处理？

- 为此目的使用 UML 交互图之一。序列图是一个很好的选择，通信图在某些场景中也可以很好工作。

第 4 章

1. 什么是事件源?
 - 这是一种架构模式，依赖于跟踪改变系统状态的事件，而不是跟踪状态本身。好处包括较低的延迟、免费的审计日志和可调试性。
2. CAP 定理的实际结果是什么?
 - 随着网络分区的发生，如果想要一个分布式系统，需要在一致性和可用性之间做出选择。对于分区，可以返回旧的数据、错误或冒超时的风险。
3. Netflix 的 Chaos Monkey 能做什么?
 - 可以为服务意外停机做好准备。
4. 缓存可以应用在哪里?
 - 在客户端，在 Web 服务器、数据库或应用程序的前面，或者在潜在客户附近的主机上，这取决于需求。
5. 当整个数据中心宕机时，如何防止应用程序宕机?
 - 使用 geodes。
6. 为什么使用 API 网关?
 - 为了简化客户机代码，不需要硬编码服务实例的地址。
7. Envoy 如何实现各种架构目标?
 - 通过提供反压、断路、自动重试和异常值来检测系统的容错。
 - 通过允许金丝雀版本和蓝绿色部署来帮助可部署性。
 - 提供负载平衡、跟踪、监视和指标。

第 5 章

1. 如何确保打开的每个代码文件在不再使用时都是关闭的?
 - 通过使用 RAII 习语；例如，通过使用 `std::unique_ptr`，将在析构函数中关闭。
2. 什么时候应该在 C++ 代码中使用“裸”指针?
 - 只传递 `optional`(可空的) 引用
3. 什么是推演指引?
 - 一种告诉编译器应该为模板推导哪些参数的方法。可以是隐式的，也可以是用户定义的。
4. 什么时候应该使用 `std::optional` 和 `gsl::not_null`?

- 前者用于传递所包含的值的情况，后者只是将指针传递给它。另外，前者可以是空的，而后者将始终指向一个对象。
5. 范围算法与视图有何不同?
 - 算法是直接的，而视图是惰性的。算法也允许使用投影。
 6. 定义函数时，除了指定概念名之外，如何约束类型?
 - 通过使用 `require` 子句。
 7. `import X` 与 `import <X>` 有何不同?
 - 后者允许从导入的 X 头文件中看到宏。

第 6 章

1. 三、五、零的规则是什么?
 - 是编写具有普通语义和更少 bug 的类型的最佳实践。
2. 什么时候使用 niebloids，而不是隐藏友元?
 - niebloids “禁用” ADL，而隐藏友元依靠它被发现。因此，前者可以加快编译速度（需要考虑的重载更少），而后者可以实现定制化。
3. 如何改进阵列接口，使其更适合生产?
 - 应该添加 `begin`、`end` 以及它们的常量和反向等价函数，以便将其用作合适的容器。诸如 `value_type`、指针和迭代器等特征对于在泛型代码中重用它很有用。向成员标记 `constexpr` 和 `noexcept` 有助于安全和性能。操作符 `[]` 的 `const` 重载也没有。
4. 什么是折叠表达式?
 - 折叠或缩减参数的表达式在二进制函子上打包。换句话说，这些语句将给定的操作应用于所有传递的可变参数模板参数，从而生成单个值（或 `void`）。
5. 什么时候不使用静态多态?
 - 当需要为代码的使用者提供一种在运行时添加更多类型的方法时。
6. 可以在闪烁清理的示例中再省一个分配吗?
 - 通过避免在添加元素时调整数组的大小。

第 7 章

1. CMake 中安装和导出目标有什么区别?
 - 导出意味着目标将对试图找到我们的包的其他项目可用，即使代码没有安装，CMake 的包注册表可以用来存储关于导出目标的位置的数据。二进制文件永远不会离开构建目

录。安装需要将目标复制到某个地方，如果不是系统目录，则需要设置到配置文件或目标本身的路径。

2. 如何使模板代码编译得更快？

- 遵循 Chiel 规则。

3. 如何在 Conan 中使用多个编译器？

- 使用 Conan 配置文件。

4. 如果想用 C++11 前的 GCC ABI 来编译 Conan 依赖，该怎么做？

- 设置 compiler.libcxx，将 libstdc++11 替换为 libstdc++。

5. 如何使用 CMake 强制指定 C++ 标准？

```
set_target_properties(our_target PROPERTIES
    CXX_STANDARD our_required_cxx_standard
    CXX_STANDARD_REQUIRED YES CXX_EXTENSIONS NO).
```

6. 如何在 CMake 中构建文档，并随 RPM 包一起发布？

- 创建一个目标来生成第 3 章中描述的文档，将其安装到 CMAKE_INSTALL_DOCDIR，然后确保路径没有在 CPACK_RPM_EXCLUDE_FROM_AUTO_FILELIST 变量中指定。

第 8 章

1. 测试金字塔的基础层是什么？

- 单元测试。

2. 有哪些类型的非功能测试？

- 性能、持久性、安全性、可用性、完整性和可用性。

3. 著名的原因分析方法叫什么？

- 5 whys

4. 是否有可能在 C++ 中测试编译时代码？

- 是的，例如使用 static_assert。

5. 当为具有外部依赖的代码编写单元测试时，应该使用什么进行测试？

- 测试替身，例如 mock 和 fake。

6. 单元测试在持续集成/持续部署中扮演什么角色？

- 它们是门控机制的基础，并作为一种早期预警特征。

7. 哪些工具可以使用测试基础架构代码？

- Serverspec, Testinfra, Goss.

8. 单元测试中访问类的私有属性和方法是一个好主意吗?

- 在设计类时，应该避免直接访问类的私有属性。

第 9 章

1. CI 在开发过程中如何节省时间?

- 更早地捕获 bug，并在它们进入生产环境之前修复它们。

2. 是否需要单独的工具来实现 CI 和 CD?

- 流水通常使用单一工具编写，实际的测试和部署使用了多种工具。

3. 什么时候在会议上进行代码审查才有意义?

- 当异步代码评审花费太长时间时。

4. 在 CI 期间，可以使用什么工具来评估代码的质量?

- 测试，静态分析。

5. 谁参与指定 BDD 的场景?

- 开发人员，QA 人员和业务人员。

6. 什么时候会考虑使用不可变的基础设施？什么时候不用？

- 最适合用于无状态服务，或可以使用数据库或网络存储外包存储的服务。它不适合有状态的服务。

7. Ansible、Packer 和 Terraform 之间有什么区别?

- Ansible 用于现有虚拟机的配置管理，Packer 用于构建云虚拟机镜像，terraform 用于构建云基础设施(如网络、虚拟机和负载均衡器)。

第 10 章

1. 为什么安全性在现代系统中很重要?

- 现代系统通常连接到网络，因此可能容易受到外部攻击。

2. 并发的挑战是什么?

- 代码更难设计和调试，可能会出现更新问题。

3. 什么是 C++ 核心指南?

- 构建 C++ 系统的最佳实践。

4. 安全编码和防御编码的区别是什么?

- 安全编码为最终用户提供健壮性，而防御编码为接口提供健壮性。

5. 如何检查软件是否包含已知的漏洞?

- 通过使用 CVE 数据库或自动扫描器，如 OWASP 依赖检查或 Snyk。

6. 静态分析和动态分析的区别是什么？

- 静态分析是在源代码上执行的，不需要执行它。动态分析需要执行。

7. 静态链接和动态链接有什么区别？

- 使用静态链接，可执行文件包含运行应用程序所需的所有代码。使用动态链接，代码的某些部分（动态库）在不同的可执行文件之间共享。

8. 如何使用编译器修复安全问题？

- 现代编译器包括检查某些缺陷的消杀程序。

9. 如何在 CI 流水中体现相应的安全意识？

- 通过使用自动工具扫描漏洞，并执行各种静态和动态分析。

第 11 章

1. 可以从本章的微基准测试的性能结果中学到什么？

- 二分搜索比线性搜索要快得多，即使要检查的元素数量没有那么多。这意味着计算复杂度（又名大 O）很重要。可能在你的机器上，即使在最大的数据集上进行最长的二分搜索，仍然比线性搜索的最短搜索要快！
- 根据缓存大小的不同，可能还会注意到，当数据不再适合特定的缓存级别时，增加所需的内存会导致速度变慢。

2. 如何遍历多维数组对性能重要吗？为什么重要，或为什么不重要？

- 这是至关重要的，因为可能会在内存中线性访问数据，这是 CPU 预取器希望并奖励我们更好的性能，或跳过内存，从而阻碍性能。

3. 在协程的例子中，为什么不能在 `do_routine_work` 函数中创建自己的线程池？

- 因为生命周期的问题。

4. 如何重新编写协程示例，使其使用生成器，而不仅仅使用任务？

- 生成器的主体将需要 `co_yield`。此外，池中的线程也需要同步，可能需要使用原子线程。

第 12 章

1. 面向服务的架构中，服务的属性是什么？

- 具有已定义结果的业务活动的表示。
- 自包含的。
- 对用户不透明。
- 可能由其他服务组成。

2. Web 服务有哪些好处?

- 很容易使用常用工具进行调试，可以很好地与防火墙一起工作，并且可以利用现有的基础设施，如负载平衡、缓存和 CDN。

3. 什么时候微服务不是一个好的选择?

- 当 RPC 和冗余的成本超过收益时。

4. 消息队列用来做什么?

- IPC、交易服务、物联网。

5. 选择 JSON 而不是 XML 有哪些好处?

- JSON 需要更低的开销，比 XML 更受欢迎，而且更容易阅读。

6. REST 是如何建立在 Web 标准之上的?

- 使用 HTTP 谓词和 RUL 构建块。

7. 云平台与传统托管有何不同?

- 云平台提供易于使用的 API，这意味着可以对资源进行编程。

第 13 章

1. 为什么微服务可以更好地利用系统资源?

- 只扩展缺少的资源，比扩展整个系统要容易得多。

2. 微服务和单应用如何共存(在一个不断发展的系统中)?

- 新功能可能会作为微服务开发，而一些功能可能会从整体中分离出来并外包出去。

3. 哪种类型的团队从微服务中受益最大?

- 遵循 DevOps 原则的跨职能自治团队。

4. 为什么在引入微服务时需要一个成熟的 DevOps 方法?

- 测试和部署大量的微服务几乎不可能由单独的团队手工完成。

5. 什么是统一日志记录层?

- 是一种可配置的工具，用于收集、处理和存储日志。

6. 日志记录和跟踪有什么不同?

- 日志记录通常是人类可读的，关注于操作；而跟踪通常是机器可读的，关注于调试。

7. 为什么 REST 不是连接微服务的最佳选择

- 例如与 gRPC 相比，可能提供更大的开销。

8. 微服务的部署策略是什么？各自的好处是什么？

- 每个主机提供单个服务——更容易根据工作负载调整机器。

- 每个主机提供多个服务——更好地利用资源。

第 14 章

1. 应用程序容器与操作系统容器有何不同?
 - 应用程序容器设计用来托管单个进程，而操作系统容器通常运行 Unix 系统中通常可用的所有进程。
2. 在早期 UNIX 系统中，有哪些沙箱环境的例子?
 - chroot, BSD Jails, Solaris Zones。
3. 为什么容器很适合微服务?
 - 提供了统一的接口来运行应用程序，而不管底层技术是什么。
4. 容器和虚拟机之间的主要区别是什么?
 - 容器更轻量级，因为不需要管理程序、操作系统内核的副本或辅助进程（如初始化系统或 syslog）。
5. 什么时候应用程序容器是糟糕的选择?
 - 当想要将多进程应用程序放在单个容器中时。
6. 构建多平台容器映像的工具有哪些?
 - manifest-tool, docker buildx。
7. 除了 Docker，还有哪些其他的容器运行时?
 - Podman, containerd, CRI-O。
8. 流行的容器协调器有哪些?
 - Kubernetes, Docker Swarm, Nomad。

第 15 章

1. 云中运行应用程序和让它们成为云本地应用程序之间有什么区别?
 - 云本地设计包含了现代技术，如容器和无服务器，这些技术打破了对虚拟机的依赖。
2. 如何在本地运行云本地应用程序?
 - 是的，这是可能的解决方案，例如 OpenStack。
3. Kubernetes 的最小高可用集群大小是多少?
 - HA 最小集群需要 3 个控制台节点和 3 个工作节点。
4. 哪个 Kubernetes 对象代表一个允许网络连接的微服务?
 - 服务。

5. 为什么日志记录在分布式系统中是不够的?

- 分布式系统中收集日志，并查找之间的相关性是有问题的。分布式跟踪更适合于某些用例。

6. 服务网格是如何帮助构建安全系统的?

- 服务网格抽象了不同系统之间的连接，从而允许应用加密和审计。

7. GitOps 是如何提高生产力的?

- 使用熟悉的工具 Git 来处理 CI/CD，而不需要编写专用的流水。

8. CNCF 监控的标准项目是什么?

- Prometheus。