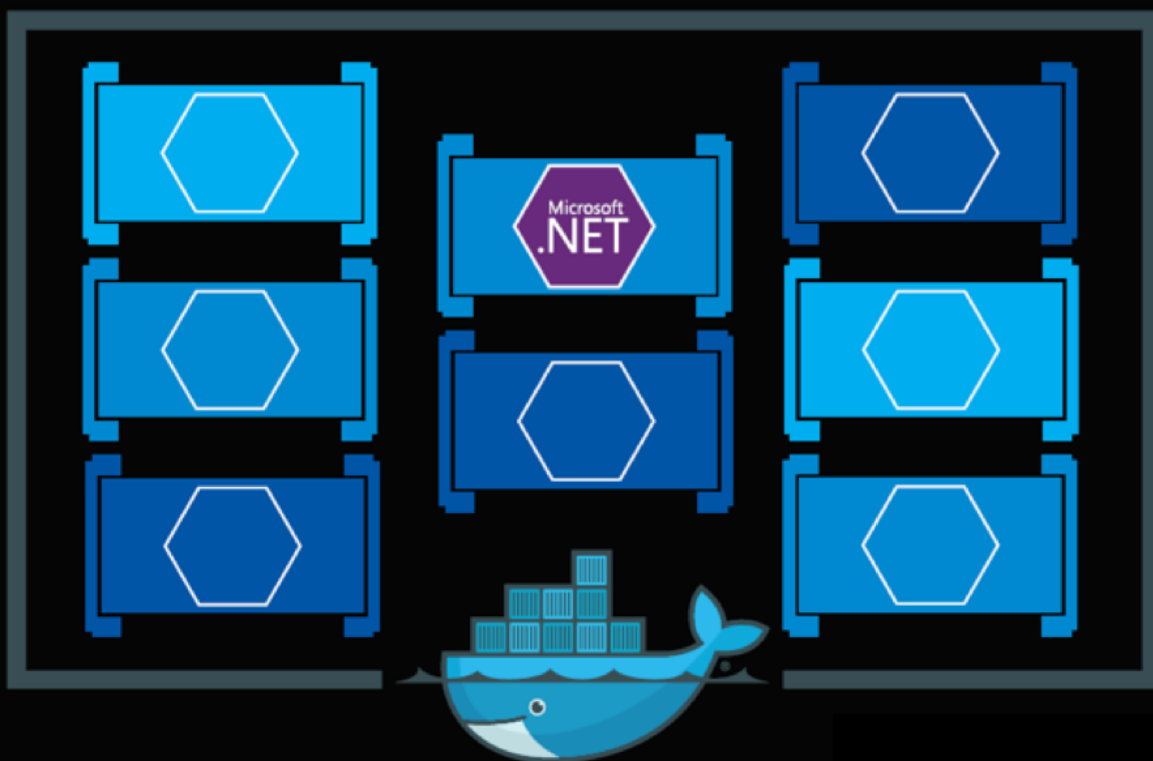


第二版
(支持 .NET Core 2)



.NET 微服务： 容器化 .NET 应用 架构指南



Cesar de la Torre
Bill Wagner
Mike Rousos

Microsoft Corporation

版本 v2.01

电子版下载地址: <https://aka.ms/microservicesebook-zh-cn>

出版商

微软开发部门, .NET 和 Visual Studio 产品组

微软公司下设部门

One Microsoft Way

Redmond, Washington 98052-6399

Copyright © 2017 by Microsoft Corporation

保留所有权利。未经出版商的明确书面许可, 本书的任何部分不得以任何形式或任何方式转载或传播。

本书“依原样”提供, 仅代表作者本人的观点和看法。本书所表达的观点意见以及所提供的信息, 包括 URL 和引用的其他互联网站点, 若有更改恕不另行通知。

本书提供的示例仅限示意和说明, 均为虚构。与现实中的任何实体无关, 也不应做此推断。

Microsoft 以及 <http://www.microsoft.com> 下“商标”网页列出的商标是微软公司的商标。

Mac 和 macOS 是苹果公司 Apple Inc.的商标。

Docker 鲸鱼标志是由 Docker, Inc. 授权使用。

所有其他商标是属于相应所有者的财产。

作者:

Cesar de la Torre, Sr. PM, .NET product team, Microsoft Corp.

Bill Wagner, Sr. Content Developer, C+E, Microsoft Corp.

Mike Rousos, Principal Software Engineer, DevDiv CAT team, Microsoft

编辑:

Mike Pope

Steve Hoag

参与人员和审阅者:

Jeffrey Ritcher, Partner Software Eng, Azure team, Microsoft

Jimmy Bogard, Chief Architect at Headspring

Udi Dahan, Founder & CEO, Particular Software

Jimmy Nilsson, Co-founder and CEO of Factor10

Glenn Condron, Sr. Program Manager, ASP.NET team

Mark Fussell, Principal PM Lead, Azure Service Fabric team, Microsoft

Diego Vega, PM Lead, Entity Framework team, Microsoft

Barry Dorrans, Sr. Security Program Manager

Rowan Miller, Sr. Program Manager, Microsoft

Ankit Asthana, Principal PM Manager, .NET team, Microsoft

Scott Hunter, Partner Director PM, .NET team, Microsoft

Dylan Reisenberger, Architect and Dev Lead at Polly

Steve Smith, Software Craftsman & Trainer at ASPSmith Ltd.

Ian Cooper, Coding Architect at Brighter

Unai Zorrilla, Architect and Dev Lead at Plain Concepts

Eduard Tomas, Dev Lead at Plain Concepts

Ramon Tomas, Developer at Plain Concepts

David Sanz, Developer at Plain Concepts

Javier Valero, Chief Operating Officer at Grupo Solutio

Pierre Millet, Sr. Consultant, Microsoft

Michael Friis, Product Manager, Docker Inc

Charles Lowell, Software Engineer, VS CAT team, Microsoft

目录

引言.....	1
关于本书.....	1
版本.....	1
本书不包含的内容.....	2
谁应该使用本书.....	2
如何使用本书.....	2
微服务和基于容器的示例应用：ESHOPONCONTAINERS.....	2
请向我们发送您的反馈.....	3
容器和 DOCKER 简介.....	4
什么是 DOCKER？.....	5
<i>Docker 容器和虚拟机之间的对比.....</i>	6
DOCKER 术语.....	7
DOCKER 容器，镜像和注册表.....	9
为 DOCKER 容器选择 .NET CORE 还是 .NET FRAMEWORK.....	11
一般建议.....	11
什么情况下选择 .NET CORE 来开发基于 DOCKER 容器的应用程序.....	12
<i>跨平台开发和部署.....</i>	12
<i>基于容器从头开发一个新项目.....</i>	13
<i>在容器上创建和部署微服务.....</i>	13
<i>在可扩展的系统中进行高密度部署.....</i>	13
什么情况下选择 .NET FRAMEWORK 来开发容器化的应用程序.....	14
<i>将现有的应用程序直接迁移到 Windows 容器中.....</i>	14
<i>需要使用不支持 .NET Core 的第三方库和 NuGet 包.....</i>	14
<i>需要使用 .NET Core 不支持的 .NET 技术.....</i>	14
<i>需要使用不支持 .NET Core 的平台或 API.....</i>	15
决策表：什么情况下在 DOCKER 中使用什么样的 .NET 框架.....	16
.NET 容器应该选择哪种操作系统.....	17
官方 .NET DOCKER 镜像.....	18
<i>使用 .NET Core 的 Docker 镜像为开发和生产提供了哪些优化.....</i>	18
基于容器和微服务的应用架构.....	20

愿景.....	20
容器设计原则.....	20
容器化单体应用.....	20
<i>用容器部署单体应用.....</i>	22
<i>发布单容器应用到 Azure 应用服务.....</i>	23
DOCKER 应用的状态和数据.....	23
面向服务的架构.....	26
微服务架构.....	27
<i>微服务的数据自治.....</i>	29
<i>微服务和限界上下文模式的关系.....</i>	30
<i>逻辑架构和物理架构.....</i>	31
<i>分布式数据管理的挑战和解决方案.....</i>	32
<i>识别微服务的领域模型边界.....</i>	37
<i>客户端微服务直连和 API 网关模式.....</i>	40
<i>微服务间的通信.....</i>	45
<i>创建、改进和控制微服务 API 的版本和契约.....</i>	53
<i>微服务的可发现性和服务注册.....</i>	55
<i>创建基于多个微服务组合界面，其中包括由微服务生成的、可视化的 UI 外观和布局.....</i>	56
<i>微服务的适应性和高可用性.....</i>	58
<i>微服务的运行状况管理和诊断.....</i>	58
编排高扩展和高可用的多容器微服务.....	61
<i>在微软 Azure 中使用容器编排引擎.....</i>	63
<i>使用 Azure 容器服务 (Container Service).....</i>	63
<i>使用 Azure Service Fabric.....</i>	65
<i>无状态和有状态的微服务.....</i>	68
基于 DOCKER 的应用程序的开发流程.....	70
愿景.....	70
DOCKER 应用程序的开发环境.....	70
<i>开发工具的选择: IDE 还是编辑器?.....</i>	70
<i>支持 Docker 容器的 .NET 语言和框架.....</i>	71
DOCKER 应用程序的开发流程.....	71
<i>基于 Docker 容器的应用程序的开发流程.....</i>	71
<i>使用 Visual Studio 开发容器的简化流程.....</i>	85
<i>在 Dockerfile 中用 PowerShell 命令创建 Windows 容器.....</i>	86
在 LINUX 或 WINDOWS NANO SERVER 上部署单容器 .NET CORE WEB 应用程序.....	87
愿景.....	87

应用程序概览.....	88
DOCKER 支持	88
疑难解答.....	90
<i>停止 Docker 容器</i>	91
<i>给您的项目添加 Docker 支持</i>	91
将传统的单体 .NET FRAMEWORK 应用程序迁移到 WINDOWS 容器中.....	92
愿景.....	92
容器化一个单体应用程序的好处	93
可能的迁移路径.....	94
应用程序概览.....	94
平移.....	96
从现有的 .NET CORE 目录微服务获取数据	98
开发环境和生产环境.....	98
设计和开发多容器和基于微服务的.NET 应用程序	99
愿景.....	99
设计面向微服务的应用程序	99
<i>应用规范</i>	99
<i>开发团队环境</i>	100
<i>选择一个架构</i>	100
<i>基于微服务的解决方案的优点</i>	103
<i>基于微服务的解决方案的缺点</i>	103
<i>外部与内部架构和设计模式</i>	105
<i>新世界：多种架构模式和多语言微服务</i>	106
创建一个简单的数据驱动的 CRUD 微服务	107
设计一个简单的 CRUD 微服务	107
使用 ASP.NET Core 实现一个简单的 CRUD 微服务	109
从 ASP.NET Core Web API 生成 Swagger 描述元数据	115
使用 DOCKER-COMPOSE.YML 定义多容器应用程序.....	120
一个简单的 Web Service API 容器.....	122
使用 docker-compose 文件面向多种环境.....	123
使用多个 docker-compose 文件来处理多个环境.....	124
构建优化的 ASP.NET Core Docker 镜像.....	130
从构建 (持续集成) 容器构建应用程序.....	131
使用容器运行数据库服务	135
将 SQL Server 作为具有微服务相关数据库的容器运行.....	135
在 Web 应用程序启动时填充种子测试数据	136

EF Core InMemory 数据库与作为容器运行的 SQL Server	137
使用在容器中运行的 Redis 缓存服务.....	138
在微服务（集成事件）之间实现基于事件的通信	140
在生产环境中使用消息代理和服务总线.....	140
集成事件.....	141
事件总线.....	142
删除重复的集成事件消息.....	156
测试 ASP.NET CORE 服务和 WEB 应用	158
实现 ASP.NET Core Web API 的单元测试.....	158
为每个微服务实施集成和功能测试.....	159
在多容器应用程序上实现服务测试.....	160
使用 IHOSTEDSERVICE 和 BACKGROUNDSERVICE 类在微服务中实现后台任务	161
在您的 WebHost 或 Host 中注册托管服务	162
IHostedService 接口.....	163
使用从 BackgroundService 基类派生的自定义托管服务类来实现 IHostedService	163
在微服务中通过领域驱动设计（DDD）和命令查询职责分离（CQRS）应对业务复杂性	168
愿景.....	168
在微服务中运用简化的 CQRS 和 DDD 模式	170
将 CQRS 和 CQS 方式应用到 ESHOPONCONTAINERS 的 DDD 微服务中	171
CQRS 和 DDD 模式不是顶级架构.....	172
在 CQRS 微服务中实现读取/查询.....	173
使用独立于领域模型约束且专门为客户端应用创建的 ViewModel	174
使用 Dapper 作为微型 ORM 实现查询.....	174
动态与静态的 ViewModel.....	175
设计面向 DDD 的微服务.....	179
保持相对较小的微服务上下文边界.....	179
DDD 微服务中的分层.....	179
设计微服务领域模型	184
领域实体模式.....	184
值对象模式.....	186
聚合模式.....	187
聚合根或根实体模式.....	187
使用 .NET CORE 实现微服务领域模型	189
自定义 .NET 标准库中的领域模型结构.....	189
在自定义 .NET 标准库中组织聚合.....	190
以 POCO 类实现领域实体.....	190
在领域实体中封装数据.....	192

Seedwork (可重复用于领域模型的基类和接口)	195
领域模型层的仓储契约 (接口)	198
实现值对象.....	199
使用枚举类代替 C# 语言的 enum 类型.....	207
在领域模型层设计验证.....	209
在领域模型层实现验证.....	209
客户端验证 (表示层验证)	212
领域事件的设计和实现.....	214
什么是领域事件?	214
领域事件和集成事件.....	214
实现领域事件.....	218
引发领域事件.....	218
跨聚合的单个事务与跨聚合最终一致性.....	220
领域事件分派程序: 从事件到事件处理程序的映射.....	222
如何订阅领域事件.....	223
如何处理领域事件.....	223
领域事件的结论.....	225
设计持久层基础架构.....	226
仓储模式.....	226
规格模式.....	229
使用 ENTITY FRAMEWORK CORE 实现持久层基础架构	231
Entity Framework Core 简介.....	231
从 DDD 视角看 Entity Framework Core 的基础架构.....	231
使用 Entity Framework Core 实现自定义仓储.....	234
IoC 容器中的 EF DbContext 和 IUnitOfWork 实例生命周期.....	236
IoC 容器中仓储实例的生命周期.....	237
表映射.....	238
实现规格模式.....	241
使用 NoSQL 数据库作为持久基础架构.....	244
Azure Cosmos DB 和原生的 Cosmos DB API 简介.....	245
针对 MongoDB 和 Azure Cosmos DB 实现 .NET 代码.....	247
设计微服务应用层和 WEB API	254
使用 SOLID 原则和依赖注入.....	254
使用 WEB API 实现微服务应用层	255
使用依赖注入将基础架构对象注入到应用层.....	255
实现命令和命令处理器模式.....	259
命令处理管道: 如何触发命令处理器.....	265

使用中介者模式 (MediatR) 实现命令处理管道.....	268
在使用 MeadiatR 中和 Behaviors 处理命令时应用横切关注点.....	273
实现弹性应用.....	277
愿景.....	277
处理局部故障.....	277
处理局部故障的策略.....	279
使用指数补偿实现重试.....	280
实现断路器模式.....	289
从 eShopOnContainers 使用 ResilientHttpClient 工具类.....	291
在 eShopOnContainers 中测试重试.....	292
在 eShopOnContainers 中测试断路器.....	292
在重试策略中添加抖动策略.....	294
运行状况监测.....	295
在 ASP.NET Core 服务中实现运行状况检查.....	295
使用 Watchdog.....	299
使用编排引擎时的运行状况检查.....	299
高级监视：可视化、分析和警报.....	300
保护 .NET 微服务和 WEB 应用程序.....	301
在 .NET 微服务和 WEB 应用程序中实现验证.....	301
使用 ASP.NET Core Identity 验证.....	302
使用外部提供器验证.....	302
使用 bearer token 验证.....	304
.NET 微服务和 WEB 应用程序的授权.....	308
实现基于角色的授权.....	308
实现基于策略的授权.....	309
在开发中安全存储应用程序密钥.....	311
在环境变量中存储密钥.....	311
使用 ASP.NET Core 密钥管理器存储密钥.....	311
在产品中使用 AZURE KEY VAULT 保护密钥.....	312
关键结论.....	314

引言

企业越来越开始认识到，使用容器可以节约成本、解决部署方面的问题、改进 DevOps 和产品运维。微软通过创建一系列产品比如 Azure 容器服务（Container Service）和 Azure Service Fabric，以及与业界的领导者比如 Docker、Mesosphere 和 Kubernetes 广泛开展合作的方式，发布了一系列基于 Windows 和 Linux 的革命性的容器产品。无论企业选择何种平台和工具，这些产品提供的容器化解决方案，都可以帮助企业在云端迅速且可扩展地构建和部署他们的应用程序。

在容器行业，Docker 已经成为事实上的标准，在 Windows 和 Linux 生态系统中，绝大多数有影响力的提供商都支持 Docker（微软是支持 Docker 的主要云服务提供商之一）。将来，在云端或本地的数据中心中，Docker 可能会变得无处不在。

此外，[微服务架构](#)正在成为设计分布式关键应用程序的一种十分重要的方法。在基于微服务的架构中，应用程序被构建为多个服务的集合，这些服务可以独立地被开发、测试、部署和版本控制。

关于本书

本书介绍了如何开发基于微服务的应用程序，以及如何使用容器来管理它们。本书主要讨论了使用 .NET Core 和 Docker 容器的架构设计和具体的实现方法。为了让大家更快地上手使用容器和微服务，本书主要专注于一款容器化的、基于微服务的应用程序的示例实现上，您可以通过这些示例实现来学习和研究。示例应用可以从 [eShopOnContainers](#) 的 GitHub 仓库下载。

本书为开发和架构设计提供了一个基础指南，在开发环境层面，主要专注于两种技术：Docker 和 .NET Core。我们的目的是，通过阅读本书，当您考虑应用程序设计的时候，可以无需关注您的产品运行环境的基础设施（云端的或本地的）。您可以在您的应用程序需要产品化的时候，再决定使用什么样的基础设施。因此，本书在基础设施方面是中立的，主要以开发环境为中心。

在学习完本书之后，接下来您可以学习微软 Azure 上已经产品化的微服务。

版本

本书做了一些修订，其中涵盖了 .NET Core 2 版本，和许多与技术浪潮（Azure 和其他第三方技术）有关的其他更新，及时和 .NET Core 2 版本保持一致。

本书不包含的内容

本书不包含应用程序生命周期、DevOps、CI/CD 管道和团队协作方面的内容。关于这些内容，可以参考《[基于微软平台和工具的容器化 Docker 应用程序生命周期](#)》这本书。本书也没有提供在 Azure 基础设施上的实现细节，比如关于特定的一些编排引擎的信息。

其他资源

- 基于微软平台和工具的容器化 Docker 应用程序生命周期 (电子书)
<https://aka.ms/dockerlifecylebook>

谁应该使用本书

本书主要为那些刚刚接触基于 Docker 的应用程序开发和基于微服务架构设计的开发人员和架构师而作。如果您想要学习如何使用微软的开发技术（尤其是使用 .NET Core）和 Docker 容器来架构、设计，和进行应用程序原型验证，本书就是为您而作的。

如果您是一个技术决策者（比如企业架构师），在您决定选择何种方式来实现全新的、现代的分布式应用程序之前，您想要概要性地了解一下架构和技术，那您会发现本书也是很有帮助的。

如何使用本书

本书的第一部分介绍了 Docker 容器，讨论了如何在 .NET Core 和 .NET Framework 开发框架之间做出取舍，然后对微服务做了一个概要性的介绍。这部分内容是专门为想有一个概要性的了解，而不需要深入了解代码实现细节的架构师和技术决策者而设计的。

本书的第二部分内容从“基于 Docker 的应用程序开发流程”这一章开始讲起。这部分内容主要专注于开发和微服务模式，以实现基于 .NET Core 和 Docker 的应用程序。这部分内容是专门为想要了解代码、模型和实现细节的开发人员和架构师而设计的。

微服务和基于容器的示例应用：

eShopOnContainers

[eShopOnContainers](#) 是一个基于 .NET Core 和微服务的开源的示例应用程序，它可以直接部署到 Docker 容器中。这个应用程序由多个子系统组成，包括网上商城前端（一个网页应用和一个原生移动应用）。它还包含几个必需的后端容器化的微服务，用于服务端的操作。

请向我们发送您的反馈

本书是为了帮助您理解 .NET 中容器化应用程序和微服务架构而作的。本书和相关的示例应用程序将来还会继续更新，所以我们十分欢迎您的反馈！如果您有关于本书的反馈意见，请直接发送到：

<mailto:dotnet-architecture-ebooks-feedback@service.microsoft.com>

容器和 Docker 简介

容器化是一种软件开发方法。通过使用容器化的开发方法，应用程序或服务以及它们的依赖和配置（被抽象为部署的 manifest 文件）被打包在一起成为一个容器镜像。容器化的应用程序可以作为一个独立的单元进行测试，也可以作为一个容器镜像实例被部署到宿主操作系统（OS）上。

就像在运送集装箱的时候，可以通过船、火车或汽车来运输货物，而无需关心货物的内部情况一样，软件容器充当了软件标准单元的角色，它可以包含不同的代码和依赖。软件容器化这种方式可以让开发人员和 IT 专业人员在很少或不做修改的情况下，跨环境部署软件。

容器还可以让应用程序在一个共享的操作系统上实现彼此隔离。容器化的应用程序运行在容器宿主（container host）之上，而容器宿主运行在 OS（Windows 或 Linux）之上。因此，容器比虚拟机镜像更加轻量。

每个容器可以运行完整的 Web 应用或者服务，如图 2-1 所示。在该示例中，Docker 主机是容器的宿主，App 1、App 2、SVC 1 和 SVC 2 是容器化的应用或服务。

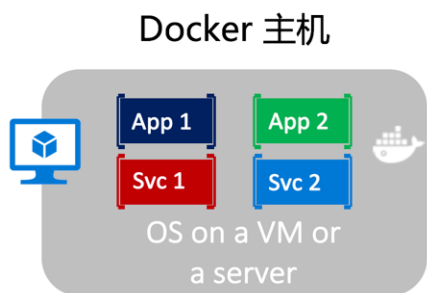


图 2-1. 运行在一个容器主机上的多个容器

容器化的另外一个优势是可扩展性。针对一些短期的任务，您可以通过创建新容器的方法进行快速横向扩展。从应用程序的角度来看，实例化一个镜像（创建一个容器）和实例化一个服务或网页应用这样的进程基本是类似的。就可靠性来说，当您跨多个主机服务器运行同一个镜像的多个实例的时候，您肯定希望每个容器（镜像实例）运行在不同故障域中的不同宿主服务器或虚拟机上。

总之，容器提供了隔离性、可移植性、灵活性、可扩展性和对整个应用程序生命周期可控性等方面的优势。这其中最重要的优势是，对开发和运维提供了隔离。

什么是 Docker ?

[Docker](#) 是一个[开源项目](#)，这个项目旨在通过把应用程序打包为可移植的、自给自足的容器（可以运行在云端或本地）的方式，实现应用程序的自动化部署。Docker 也是一家专门促进和发展这门技术的[公司](#)，它和多家云、Windows、Linux 提供商都有合作，包括微软。

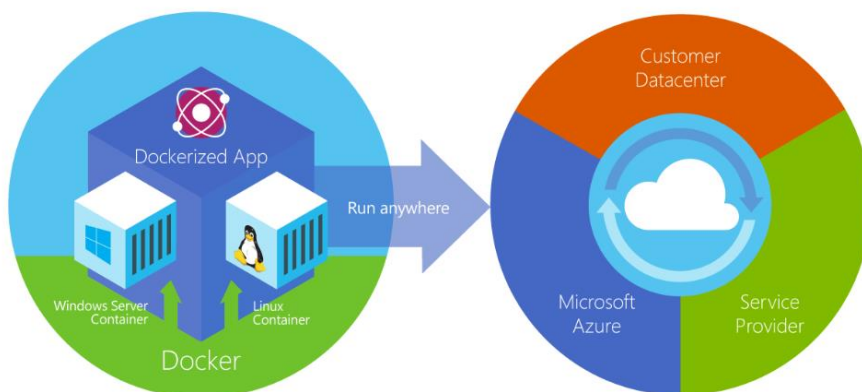


图2-2. Docker 可以把容器部署到混合云的各个板块上

在 Windows 和 Linux 上，Docker 镜像容器可以本地化运行。但是，Windows 镜像只能运行在 Windows 主机上，Linux 镜像只能运行在 Linux 主机上，这意味着镜像需要寄宿到一台服务器或一台虚拟机上。

开发人员可以使用 Windows、Linux 或者 MacOS 上的开发环境进行工作。在开发机上，开发人员运行一个 Docker 主机用于部署 Docker 镜像，它包含了应用程序本身和相关的依赖。工作在 Linux 和 Mac 上的开发人员使用的 Docker 主机是基于 Linux 的，他们只能创建用于 Linux 容器的镜像。（工作在 Mac 上的开发人员可以在 MacOS 上编辑代码或运行 Docker 命令行工具，但是直到写作本书时为止，容器还不能直接运行在 MacOS 上。）工作在 Windows 上的开发人员，他们既可以创建基于 Windows 容器的镜像，也可以创建基于 Linux 容器的镜像。

为了在开发环境中托管容器和提供一些辅助的开发工具，Docker 发布了 Windows 版和 MacOS 版的 [Docker 社区版 \(CE\)](#)。这些产品会安装必需的虚拟机 (Docker 主机) 来托管容器。Docker 还提供了 [Docker 企业版 \(EE\)](#)，这个版本是专门为企业开发而设计的，在生产环境下，需要构建，交付和运行大型关键业务应用程序的 IT 团队可以使用这个版本。

Windows 容器提供了两种不同类型容器（或称运行时）：

- Windows Server 容器：通过进程和命名空间的隔离技术来实现应用程序的隔离。Windows Server 容器会在容器宿主机以及宿主机上运行的所有容器之间共享同一个内核。

- Hyper-V 容器：通过在深度优化的虚拟机上运行每个容器的方式加强了 Windows Server 容器的隔离性。在这种结构中，容器宿主机的内核不会与同一主机运行的 Hyper-V 容器共享，从而提供了更好的隔离性。

这两种容器的镜像以相同的方式创建，并具有相同的功能。不同之处在于如何通过镜像创建容器的方法——运行 Hyper-V 容器需要额外的参数。具体细节可以参考 [Hyper-V 容器](#)。

Docker 容器和虚拟机之间的对比

图 2-3 展示了虚拟机和 Docker 容器之间的对比。

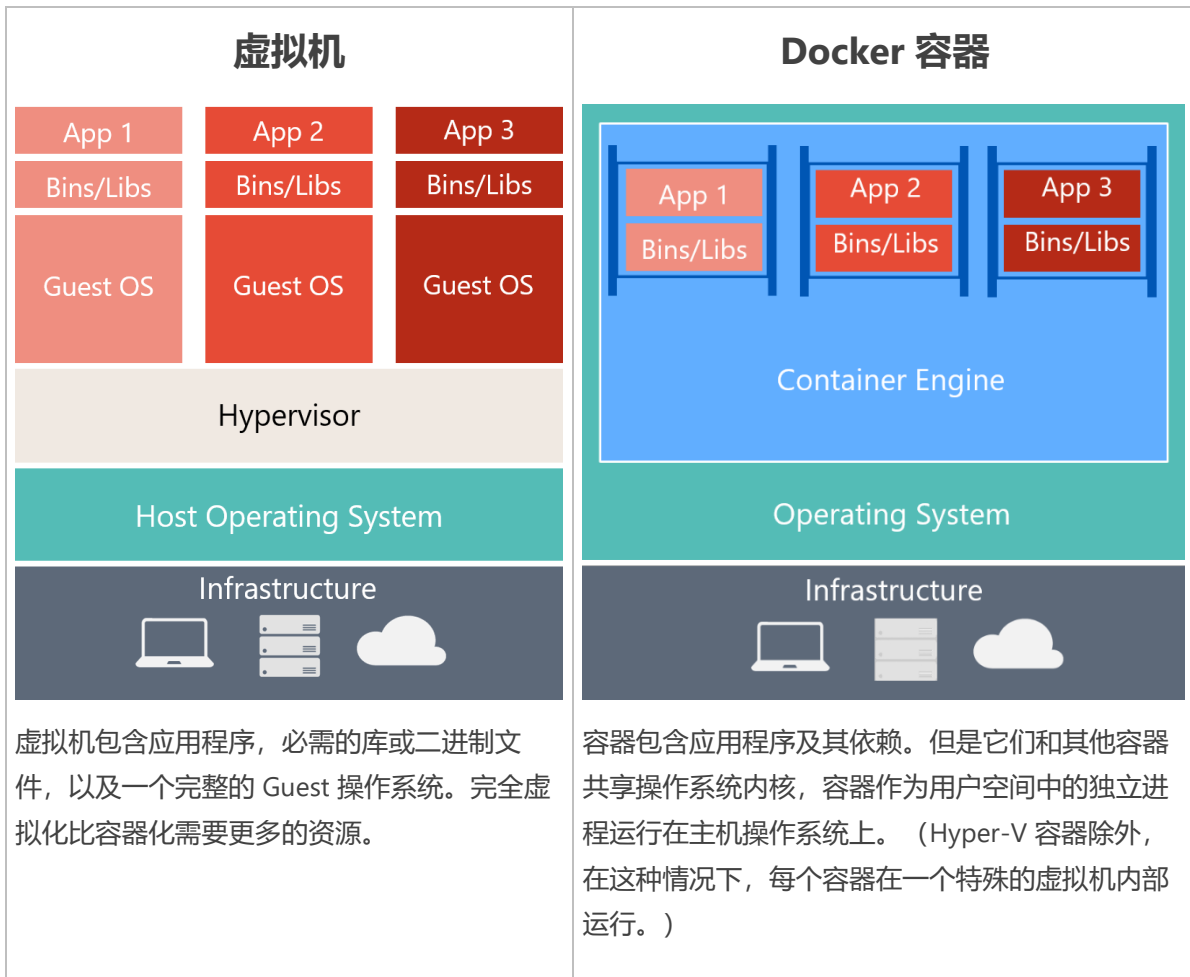


图 2-3. 传统虚拟机和 Docker 容器之间的对比

因为容器需要更少的资源（例如，它们不需要一个完整的操作系统），所以它们更容易部署，和快速启动。这可以让您获得更高的“密度”，也就是说，在同一个硬件单元上，您可以运行更多的服务，从而降低成本。

运行在同一个内核上的副作用是，隔离性要比虚拟机差。

镜像存在的意义在于，它可以多次不同的部署环境（依赖性）保持一致。这意味着您可以在您的机器上调试应用程序，然后确保以相同环境把它部署到另外一台机器上。

容器镜像是一种打包的方法，它把应用程序或服务打包，然后用可靠的、可重复的方式部署它。可以说，Docker 不仅是一门技术，还是一种方法论，一种工作流程。

当您使用 Docker 的时候，您肯定不会听到开发人员们这样说：“在我的机器上运行正常，怎么到生产环境下就不行了呢？”他们会简单的来一句：“在 Docker 上它能正常运行”，这是因为打包的 Docker 应用程序可以在任何支持 Docker 的环境中运行，并且保证在所有部署场景中按预期的方式运行（开发、测试、预发布、生产等等）。

Docker 术语

在更深入了解 Docker 之前，本小节先列出一些您应该熟悉的术语和定义。如果想进一步了解各种定义，可以参考 Docker 提供的完整的[词汇表](https://docs.docker.com/glossary/) (<https://docs.docker.com/glossary/>)。

容器镜像：一个包含了创建容器所需的所有依赖和信息的“包”。一个镜像包含了所有依赖（例如各种框架）和容器运行时需要使用的所有部署和执行配置。通常，一个镜像来自于多个基础镜像，它们一层一层地堆砌成了容器的文件系统。如果一个镜像已经被创建了，那么它是不可变更的。

容器：Docker 镜像的一个实例。一个容器就表示一个正在运行的应用程序，进程或服务。它由 Docker 镜像的内容，一个执行环境和一个标准的指令集组成。当需要扩展一个服务的时候，您可以从同一个镜像创建多个容器实例。或者，您也可以从同一个镜像创建多个实例，并分别给每个实例传递不一样的参数，来完成一个批处理任务。

标签：用于镜像的标记或标签，让不同的镜像或同一个镜像的不同版本（依赖于版本号或目标环境）之间可以被区分开来。

Dockerfile：一个文本文件，它包含如何创建一个镜像的指令。

构建：构建一个容器镜像的操作。基于 Dockerfile 提供的信息和上下文，以及构建这个镜像的文件夹中的其他文件来构建一个容器镜像。您可以使用 Docker 命令（docker build）来构建镜像。

仓库：相关 Docker 镜像的集合，用标签来表示镜像的版本。有些仓库会包含某个特定镜像的多个变体，例如，一个包含了若干 SDK 的镜像（更加重量级），一个只包含了运行时的镜像（更加轻量级）等等。这些变体可以用标签来标记。一个仓库可以包含多个平台变体，例如一个 Linux 镜像和一个 Windows 镜像。

注册表 (Registry)：为仓库提供访问服务。对于大多数公共镜像来说，默认的注册表是 Docker Hub（它隶属于 Docker 公司）。一个注册表通常包含来自于多个团队的仓库。企业通常会创建私有的注册表来存储和管理他们创建的镜像。Azure 容器注册表是另外一个注册表的例子。

Docker Hub: 一个公用的注册表, 用来上传镜像, 让它们协同工作。[Docker Hub](#) 提供了 Docker 镜像的保存服务, 公有或私有的注册表, 构建触发器和 Webhook, 并且它可以和 GitHub 和 Bitbucket 无缝集成。

Azure 容器注册表 (Azure Container Registry): 一种和 Docker 镜像配合使用的公共资源, 它的组件被放在了 Azure 上。它提供了一个与您的 Azure 部署紧密相关的注册表, 让您可以使用 Azure Active Directory 组和权限对其进行访问控制。

Docker 可信注册表 (Docker Trusted Registry): 一种 Docker 注册表服务 (来自 Docker 公司), 可以在本地安装, 所以它一般在企业内部的数据中心和网络中使用。对于需要在企业内部进行管理的私有镜像来说, 这十分方便。Docker 可信注册表包含在 Docker Datacenter 这个产品中。更多信息, 可以参考 [Docker 可信注册表 \(DTR\)](#)。

Docker 社区版 (CE): 针对 Windows 和 MacOS 的一个开发工具集, 用于在本地构建, 运行和测试容器。Docker CE for Windows 提供了 Linux 容器和 Windows 容器的开发环境。Windows 上的 Linux Docker 主机是基于 [Hyper-V](#) 虚拟机的。而 Windows 容器的主机是直接基于 Windows 的。Docker CE for Mac 是基于 Apple 的 Hypervisor 框架和 [xhyve hypervisor](#) 的, 它在 Mac OS X 上提供了一个 Linux Docker 主机的虚拟机。Docker CE for Windows 和 Docker CE for Mac 替代了原来基于 Oracle VirtualBox 的 Docker Toolbox。

Docker 企业版 (EE): 企业级的 Linux 和 Windows 开发的 Docker 工具集。

Compose: 一个命令行工具和用于定义和运行多容器应用程序的 YAML 文件格式的元数据。可以用一个或多个 .yml 文件 (依赖于具体环境, 可以覆盖部分值), 基于多个镜像定义一个应用程序。当您创建了这些定义之后, 您只用一行命令 (`docker-compose up`) 就可以完成整个多容器应用的部署, 这行命令会在 Docker 宿主机上为每一个镜像创建一个容器。

集群: 一组 Docker 宿主机通过一个虚拟的 Docker 宿主机暴露出来, 让应用程序可以扩展为多个服务实例, 分布到集群内的多个宿主机上。Docker 集群可以使用 Docker Swarm、Mesosphere DC/OS、Kubernetes 和 Azure Service Fabric 来创建。(如果您使用 Docker Swarm 来管理集群, 通常使用 swarm 这个术语来代替集群。)

编排引擎: 用来简化集群和 Docker 宿主机管理的一个工具。编排引擎可以让您通过一个命令行工具 (CLI) 或者一个图形化的界面来管理镜像, 容器和主机。您可以管理容器网络, 配置, 负载均衡, 服务发现, 高可用性, Docker 主机配置等等。编排引擎负责在一组节点上运行, 分发, 扩展和修复工作负载。通常来说, 编排引擎产品和提供集群基础设施的产品 (比如 Mesosphere DC/OS、Kubernetes、Docker Swarm 和 Azure Service Fabric) 是相同的。

Docker 容器，镜像和注册表

使用 Docker 的时候，开发人员需要创建一个应用或服务，然后把它和它的依赖打包到一个容器镜像中。镜像应用或服务，以及它的配置和依赖的静态表现形式。

要运行应用和服务，这个应用的镜像会被实例化，来创建一个容器，这个容器将会运行在 Docker 宿主机上。容器最开始的时候是在开发环境或 PC 上进行测试的。

开发人员应该把镜像存储到注册表中，这对于生成环境中的编排引擎也是需要的编排引擎。Docker 维护了一个公用的注册表叫 [Docker Hub](#)，其他提供商也为不同的镜像集合提供了各种注册表。此外，企业可以在本地创建一个私有的注册表，供企业内部的 Docker 镜像来使用。

图 2-4 说明了 Docker 中的镜像和注册表和其他组件之间的关系。图中也展现了来自于其他供应商的多个注册表。

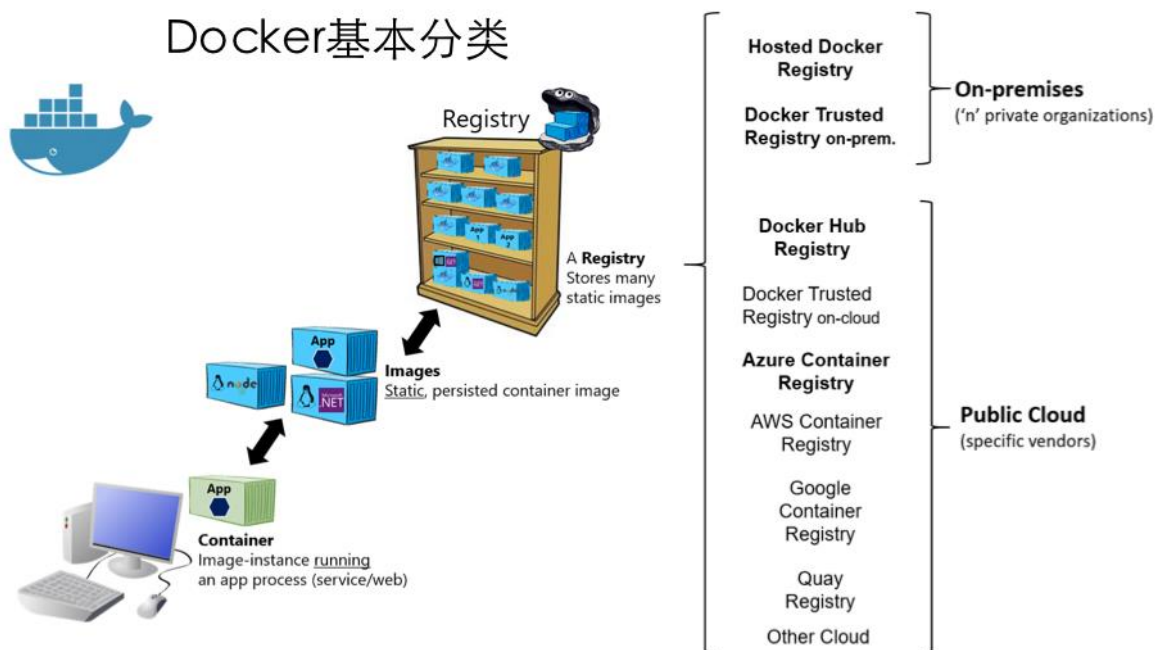


图 2-4. Docker 术语和概念的分类

把镜像放到注册表中，可以让您存储静态的和不可变的应用程序块，其中包含它们在框架级的所有依赖。这些镜像可以被版本化，然后在多个不同的环境下部署，因此镜像提供了一个一致性的部署单元。

以下这些情况，推荐使用私有的镜像注册表（既可以部署到本地，也可以部署到云端）：

- 基于保密性考虑，您的镜像不能被公开共享。

- 您希望拥有在镜像和您选择的部署环境之间的最小的网络延迟。例如，如果您的生产环境是 Azure 云，那么您可能更希望在 Azure 容器注册中存储您的镜像，让网络延迟最小化。如果您的生产环境在本地，您可能更希望在同一个内网中使用本地的 Docker 可信注册表。

为 Docker 容器选择 .NET Core 还是 .NET Framework

使用 .NET 构建服务端容器化的应用程序时，有两个框架可供选择：[.NET Framework](#) 和 [.NET Core](#)。它们之间共享许多 .NET 平台组件，您可以在它们之间共享代码。但是，它们之间有着根本性的不同，选择哪个框架，取决于您想实现什么。本章会针对在何种情况下选择哪个框架给出一些建议。

一般建议

本小节针对选择 .NET Core 还是 .NET Framework 给出一个概要性的介绍。关于如何做出决策，接下来的一些章节，会给出更多的细节。

如下这些情况，您应该使用 .NET Core 和 Linux 或 Windows 容器来开发您的容器化 Docker 服务端应用程序：

- 您需要跨平台。例如，您需要同时使用 Linux 容器和 Windows 容器。
- 您的应用程序架构是基于微服务的。
- 您需要容器快速启动，并且希望容器尽量保持轻量级来获得更高的“密度”，也就是说，为了节约成本，您希望在每个硬件单元上承载更多的容器。

简言之，当您创建一个新的容器化的 .NET 应用程序的时候，您应该把 .NET Core 作为一个默认的选择。.NET Core 有很多优势，并且它和容器的方法论和工作方式完美匹配。

使用 .NET Core 的另外一个优势是，在同一台机器上，您可以同时运行基于不同 .NET 版本的应用程序。对于不使用容器的服务器或虚拟机来说，这一点会更加重要一些，因为容器根据需要的不同 .NET 版本把应用程序进行了隔离。（只要它们和底层的操作系统是兼容的。）

如下这些情况，您应该使用 .NET Framework 来开发您的容器化 Docker 服务端应用程序：

- 您的应用程序目前使用的就是 .NET Framework，并且高度依赖于 Windows。
- 您需要使用 .NET Core 不支持的 Windows API。
- 您需要使用 .NET Core 不支持的第三方的 .NET 库或者 NuGet 包。

使用 Docker 部署 .NET Framework 程序可以改善您的部署体验，简化部署方面的问题。这样的“平移”场景对于容器化传统的使用 .NET Framework 技术来开发的应用程序（如 ASP.NET WebForms, MVC Web App 或 WCF）来说，是十分重要的。

其他资源

- **电子书：借助 Azure 云和 Windows 容器让现有 .NET 应用程序实现现代化革新**
<https://aka.ms/liftandshiftwithcontainersebook>
- **示例应用：使用 Windows 容器让传统 ASP.NET Web 应用实现现代化革新**
<https://aka.ms/eshopmodernizing>

什么情况下选择 .NET Core 来开发基于 Docker 容器的应用程序

由于 .NET Core 具有模块化和轻量级的特性，所以对于容器来说，它是绝佳的“黄金搭档”。当您部署和启动一个容器的时候，使用 .NET Core 的镜像，比使用 .NET Framework 的镜像体积小得多。而且，如果您想在一个容器中使用 .NET Framework，那么您的镜像必须基于 Windows Server Core 基础镜像来构建。如果使用 .NET Core，您的镜像可以基于 Windows Nano Server 镜像或 Linux 镜像来构建，而 Windows Nano Server 系统镜像或 Linux 系统镜像相比 Windows Server Core 系统镜像轻量很多。

此外，.NET Core 是跨平台的，所以您可以使用 Linux 或 Windows 容器镜像来部署您的服务端应用程序。然而如果您使用传统的 .NET Framework，那么您只能部署基于 Windows Server Core 的镜像。

关于选择 .NET Core 的理由，下面是更详细的解释。

跨平台开发和部署

显而易见，如果您的目标是实现一个在多个平台上（Linux 和 Windows）都可以运行的 Docker 应用程序（网页应用或服务），那么 .NET Core 是最合适的选择，因为 .NET Framework 只支持 Windows。

.NET Core 也支持 MacOS，目前只能作为一个开发平台。当您需要把容器部署到一个 Docker 主机上的时候，这个主机必须是基于 Linux 或 Windows 的。例如在开发环境中，您可以在 Mac 上运行一个 Linux 虚拟机来使用。

[Visual Studio](#) 为 Windows 提供了一个集成开发环境（IDE），支持 Docker 开发。

[Visual Studio for Mac](#) 是一个由 Xamarin Studio 演化而来的 IDE，运行在 MacOS 上，它支持基于 Docker 的应用程序开发。对于那些工作在 Mac 机上，想使用一款强大的 IDE 的开发人员来说，这应该是一个更好的选择。

您可以使用 [Visual Studio Code](#)（VS Code），它支持 MacOS、Linux 和 Windows。VS Code 可以完整地支持 .NET Core，并且支持智能提示和调试。VS Code 只是一个轻量级的编辑器，在 Mac 上您可以使用 VS Code，Docker CLI 和 [.NET Core command-line interface \(CLI\)](#) 一起来开发容器化的应用程序。您也可以使用支持 .NET Core 的第三方编辑器，比如 Sublime、Emacs、vi 和开源的 OmniSharp 项目（同样支持智能提示）。

除了 IDE 和编辑器，您还可以使用[.NET Core CLI](#)工具，它支持所有的平台。

基于容器从头开发一个新项目

虽然容器通常和微服务架构一起配合使用，但是也可以使用其他架构模型来开发容器化的网页应用或服务。在 Windows 容器上，您可以使用 .NET Framework，但是，对于容器和微服务架构来说，模块化和轻量级的 .NET Core 才是完美的“黄金搭档”。当您创建和部署一个容器的时候，使用 .NET Core 的镜像体积比使用 .NET Framework 的镜像体积小很多。

在容器上创建和部署微服务

您可以使用传统的 .NET Framework 来开发基于微服务的应用程序（非容器化的），因为 .NET Framework 也支持跨进程安装和共享，而进程是轻量级的可以快速启动。但是，如果您要使用容器，那么对于容器上的微服务来说，使用基于 Windows Server Core 的和传统的 .NET Framework 的镜像还是太重量级了。

相比之下，如果您要基于容器开发原生的微服务系统，.NET Core 才是最佳的候选者，因为 .NET Core 更加轻量级。而且，相关的容器镜像既可以是 Linux 镜像，也可以是 Windows Nano 镜像，它们更加轻量级，这让容器更加轻量级，可以快速启动。

微服务的意思就是尽可能的“小”：运行起来要轻，空间占用要小，边界上下文要小，依赖性要小，启动和停止要快。要满足这些要求，您需要使用像 .NET Core 容器镜像这样的更小的可以快速实例化的容器镜像。

微服务架构可以让您跨越服务的边界混搭各种技术。这可以让您渐进式地迁移到 .NET Core，因为全新的微服务既可以和其他微服务一起工作，也可以和用 Node.js、Python、Java、GoLang 或其他技术开发的服务一起工作。

在可扩展的系统中进行高密度部署

如果您的基于容器的系统需要最优的密度、颗粒度和性能，那么 .NET Core 和 ASP.NET Core 是您绝佳的选择。ASP.NET Core 比传统 .NET Framework 中的 ASP.NET 快 10 倍以上，而且，相对于其他业界比较流行的微服务技术（如 Java Servlets、Go 和 Node.js）来说，它处于业界领先地位。

因为在微服务架构中，您通常需要运行几百个微服务（容器），所以对于微服务来说，这至关重要。在 Linux 或 Windows Nano 上使用 ASP.NET Core 的镜像（基于 .NET Core 运行时），您可以在很少量的服务器或虚拟机上运行您的系统，进而在基础设施和主机方面节约大量的成本。

什么情况下选择 .NET Framework 来开发容器化的应用程序

尽管对于全新的应用程序和应用程序模型来说，.NET Core 的优势十分明显，但是，在很多遗留系统的场景下，.NET Framework 仍然是一个不错的选择。

将现有的应用程序直接迁移到 Windows 容器中

即使您不想创建微服务，您可能也想使用 Docker 容器来简化部署。例如，也许您想用 Docker 改善您的 DevOps 工作流程 — 容器可以提供隔离性更好的测试环境，此外，当您迁移到生产环境的时候，容器还可以帮助您排除一些由于缺少必要的依赖而引起的部署方面的问题。在这些情况下，即使您只需要部署一个独立的应用程序，让您当前的 .NET Framework 应用程序使用 Docker 和 Windows 容器也是比较明智的。

针对这个场景，在绝大多数情况下，您不需要把您现有的应用程序迁移到 .NET Core 上；您可以使用包含传统的 .NET Framework 的 Docker 容器。但是，当您需要扩展一个应用程序的时候，那么使用 .NET Core 才是推荐的方式，比如说，用 ASP.NET Core 写一个新的服务。

需要使用不支持 .NET Core 的第三方库和 NuGet 包

第三方的库正在迅速地拥抱 [.NET 标准](#)，这个标准可以让代码在各种 .NET 框架上实现共享，其中包括 .NET Core。随着 .NET 标准库 2.0 和兼容不同框架的 API 应用的越来越广泛，在 .NET Core 2.0 中，应用程序也可以直接使用各种现有的 .NET Framework 库（可以参考[兼容性说明](#)）。

然而，虽然 .NET Standard 2.0 和 .NET Core 2.0 已经做了一些额外的改进，但是总有这样一些场景：某些 NuGet 包需要 Windows 才能运行，并且它们可能不支持 .NET Core。如果这些包对您的应用程序来说至关重要，那么您必须在 Windows 容器上使用 .NET Framework。

需要使用 .NET Core 不支持的 .NET 技术

有些 .NET Framework 技术在当前版本的 .NET Core（写作本书的时候，当前版本是 2.0）中并不可用。这些不可用的技术中，有些在以后的 .NET Core 版本（.NET Core 2.X）中可用，而有些并不是适用 .NET Core 新应用程序模型的设计目标，它们很可能永远都不可用。

下面列出一些在 .NET Core 2.X 中不可用的技术：

- ASP.NET Web Form。该技术只能在 .NET Framework 上使用。当前，并没有计划在 .NET Core 上支持 ASP.NET Web Form。

- WCF 服务。虽然有一个来自于 .NET Core 的 WCF 客户端库可以用来连接 WCF 服务（基于 2017 年前期的版本），但是 WCF 的服务端实现只支持 .NET Framework。这个场景在将来的 .NET Core 版本中会被考虑的。
- 工作流相关的服务。Windows Workflow Foundation (WF), Workflow Services (WCF + WF), 和 WCF Data Services (过去叫 ADO.NET Data Services) 只支持 .NET Framework。当前没有计划在 .NET Core 上支持它们。

除了在官方的 [.NET Core 路线图](#) 中列出的那些技术，其他特性也有可能被引入到 .NET Core 中。完整的列表可以参考 CoreFX 的 Github 主页上那些被打上 [port-to-core](#) 标签的条目。注意，这个列表并不表示微软承诺会把这些组件引入到 .NET Core 中——这些条目只是简单地捕获记录了来自社区的请求。如果您比较关注上面列出的某些组件，可以考虑参加 Github 上的讨论，以便于让大家听到您的声音。如果您考虑的东西并不存在，可以在 [CoreFX 仓库中新开一个 Issue](#)。

需要使用不支持 .NET Core 的平台或 API

有些微软的，或第三方的平台并不支持 .NET Core。例如，有些 Azure 服务提供了一个 SDK，而这个 SDK 并不支持 .NET Core。这是暂时的，所有 Azure 服务最终都会使用 .NET Core 的。例如，在 2016 年 11 月 16 日，[.NET Core 的 Azure 文档数据库 SDK](#) 已经发布了一个预览版本，现在它已经是稳定的通用版 (GA) 了。

与此同时，如果 Azure 中的某些平台或服务的客户端 API 还不支持 .NET Core，那么您可以使用来自于 Azure 服务相对应的 REST API 或者基于 .NET Framework 的客户端 SDK。

其他资源

- **.NET Core 指南**
<https://docs.microsoft.com/dotnet/articles/core/index>
- **移植 .NET Framework 到 .NET Core**
<https://docs.microsoft.com/dotnet/articles/core/porting/index>
- **Docker 上的 .NET Framework 指南**
<https://docs.microsoft.com/dotnet/articles/framework/docker/>
- **.NET 组件概览**
<https://docs.microsoft.com/dotnet/articles/standard/components>

决策表：什么情况下在 Docker 中使用什么样的 .NET 框架

下面这张决策表概括总结了应该使用 .NET Framework 还是 .NET Core 的决策因素。一定要记住，对于 Linux 容器来说，必须要使用基于 Linux 的 Docker 主机（虚拟机或服务器），而对于 Windows 容器来说，必须要使用基于 Windows Server 的 Docker 主机（虚拟机或服务器）。

架构/ 应用类型	Linux 容器	Windows 容器
基于容器的微服务	.NET Core	.NET Core
单体应用	.NET Core	.NET Framework .NET Core
获得最好的的性能和可扩展性	.NET Core	.NET Core
传统的 Windows Server 应用程序（“遗留的应用程序”） 迁移到容器	--	.NET Framework
全新的基于容器的应用程序（“新开发的应用程序”）	.NET Core	.NET Core
ASP.NET Core	.NET Core	.NET Core (推荐) .NET Framework
ASP.NET 4 (MVC 5, Web API 2, 和 Web Forms)	--	.NET Framework
SignalR 服务	.NET Core 2.1 (当这个版本发布的时候)	.NET Framework .NET Core 2.1 (当这个版本发布的时候)
WCF, WF 和其他传统框架	WCF in .NET Core (只支持 WCF 客户端库)	.NET Framework WCF in .NET Core (只支持 WCF 客户端库)
需要使用 Azure 服务	.NET Core (最终所有 Azure 服务都会提供支持 .NET Core 的客户端 SDK)	.NET Framework .NET Core (最终所有 Azure 服务都会提供支持 .NET Core 的客户端 SDK)

.NET 容器应该选择哪种操作系统

考虑到 Docker 可支持多样化的操作系统，并且 .NET Framework 和 .NET Core 之间存在巨大的差异，因此用户需要结合自己使用的框架确定最适合的操作系统和版本。

对于 Windows，可以使用 Windows Server Core 或 Windows Nano Server。这些 Windows 版本提供了 .NET Framework 或 .NET Core 应用程序可能需要的不同特征（如 IIS，以及 Kestrel 等自托管 Web 服务器）。

对于 Linux，官方提供的 .NET Docker 镜像已可以使用并支持多种发行版（如 Debian）。

如图 3-1 所示，您可以看到依赖于所用的 .NET 框架的可能的操作系统版本。

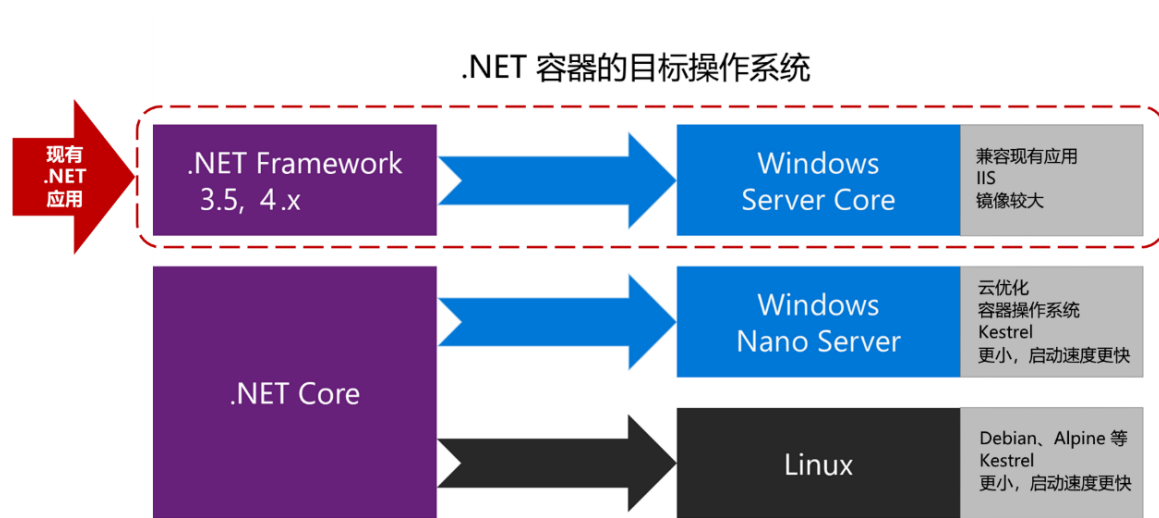


图 3-1. 根据 .NET 框架的版本来选择操作系统

在某些情况下，您可能需要使用其他 Linux 发行版，或者您想使用的镜像版本微软还没有提供，这时您可以创建您自己的 Docker 镜像。例如，您可以用运行在传统的 .NET Framework 和 Windows Server Core 之上的 ASP.NET Core 创建一个镜像，对于 Docker 来说，这并不是特别常见的一个场景。

当您把镜像的名字添加到 Dockerfile 文件的时候，您可以通过标签来指定操作系统和运行时的具体版本，下面是一些具体的实例：

microsoft/dotnet:2.0.0-runtime-jessie	Linux 上的 .NET Core 2.0 运行时
microsoft/dotnet:2.0.0-runtime-nanoserver-1709	Windows Nano Server 上的 .NET Core 2.0 运行时 (Windows Server 2016 秋季创意者更新版本 1709)

microsoft/aspnetcore:2.0	.NET Core 2.0 多架构：支持依赖于具体主机的 Linux 和 Windows Nano Server。aspnetcore 镜像针对 ASP.NET Core 做了一些优化
--------------------------	--

官方 .NET Docker 镜像

官方的 .NET Docker 镜像是由微软创建和优化的 Docker 镜像。它们都公开发布在 [Docker Hub](#) 上的 Microsoft 仓库中。每个仓库可以包含多个镜像，取决于 .NET 的版本，操作系统的类型和版本（Linux Debian, Linux Alpine, Windows Nano Server, Windows Server Core 等等）。

对于 .NET 仓库来说，微软的愿景是尽量构建小而专的仓库。一个仓库代表一种具体的场景，或者一个需要完成的具体工作。例如，[microsoft/aspnetcore](#) 镜像应该在 Docker 上使用 ASP.NET Core 的场景下使用，因为这些镜像做了额外的优化，可以让容器更快地启动。

而 .NET Core 镜像（[microsoft/dotnet](#)）是为基于 .NET Core 的控制台应用程序而设计的。例如，批处理，Azure WebJobs 和其他应该使用 .NET Core 的控制台场景。这些镜像并不包含 ASP.NET Core 栈，所以这些容器镜像的体积会更小一些。

大多数镜像仓库都提供了很多标签，帮助您选择特定的框架版本和操作系统（具体的 Linux 发行版或 Windows 的版本）。

关于微软提供的官方 .NET Docker 镜像的详细信息，可以参考 [.NET Docker 镜像摘要](#)。

使用 .NET Core 的 Docker 镜像为开发和生产提供了哪些优化

为开发人员构建 Docker 镜像的时候，微软主要把注意力放到了以下场景上：

- 用于开发和构建 .NET Core 应用程序的镜像。
- 用于运行 .NET Core 应用程序的镜像。

为什么需要多个镜像？当开发、构建和运行容器化的应用程序时候，您通常会有不同的优先级。通过为这些独立的任务提供不同的镜像，微软可以帮助大家分别优化开发，构建和部署应用的流程。

开发和构建阶段

开发阶段，最重要的事情是，您迭代新功能的速度和调试新功能的能力。镜像的体积并不像代码变更和快速地查看这些变更的能力那么重要。一些工具和“生成代理容器”，在开发和生成的时候用的就是开发版的 ASP.NET Core 镜像（[microsoft/aspnetcore-build](#)），您甚至可以使用这个镜像作为构建容器。当在一个容器内部构建的时候，重要的事情是编译您的应用程序所需的一些组件。这其中包括编译器，其他的 .NET 依赖，以及像 npm、Gulp 和 Bower 这样的网页开发所需的依赖。

这种构建镜像为什么很重要？实际上您并不会把这个镜像部署到生产环境中。这种镜像只是用来构建一些需要放到生产镜像中的东西。这种镜像主要用于持续集成（CI）环境和构建环境。举例来说，与其直接在构建代理主机（例如一台虚拟机）上手动安装所有的应用程序依赖，还不如让构建代理直接实例化一个带有构建应用程序所需的所有依赖的 .NET Core 构建镜像。您的构建代理只需要知道怎样运行这个 Docker 镜像就可以了。这简化了您的 CI 环境，让它变得更加可预测。

在生产环境下

在生产环境下，最重要的事情是，您可以快速地部署和启动基于产品化的 .NET Core 镜像的容器。所以，基于 [microsoft/aspnetcore](#) 的运行时镜像通常都比较小，这可以让它更快速地从您的 Docker 注册表传输到您的 Docker 主机上。

提前准备好需要运行的东西，可以让启动容器到产生处理结果之间的时间间隔缩到最短。在 Docker 模型中，无需从 C# 代码的编译开始，这个过程是在构建容器中，通过运行 `dotnet build` 或 `dotnet publish` 命令来完成。

在这个优化过的镜像中，您只需要把二进制文件和运行这个应用程序所需的其他东西放进去就可以了。例如，通过 `dotnet publish` 命令创建的部署包只包含编译过的 .NET 二进制文件、图片、.js 和 .css 文件。过段时间，您会看到一些包含 JIT 预编译包的镜像。

虽然 .NET Core 和 ASP.NET Core 镜像有很多个版本，但是它们之间共享了一个或多个镜像层，其中包括基础镜像层。所以，存储一个镜像所需的磁盘空间很小。它只包含您自定义的镜像相对于基础镜像的增量。所以，从注册表拉取镜像变得更快了。

当您在 Docker Hub 上浏览 .NET 镜像仓库的时候，您会发现用标签分类或标记的多个版本。下表可以帮助您选择应该使用哪个版本：

<code>microsoft/aspnetcore:2.0</code>	基于 Linux 和 Windows (多架构)，只包含 ASP.NET Core 运行时和 ASP.NET Core 的一些优化
<code>microsoft/aspnetcore-build:2.0</code>	基于 Linux 和 Windows (多架构)，包含 ASP.NET Core 的 SDK

基于容器和微服务的应用架构

愿景

微服务提供了强大优势的同时也带来了巨大的新挑战。微服务架构模式则为创建微服务应用提供了基础支持。

通过本书前面章节的内容，您已经学习了容器和 Docker 的基本知识，这是为了开始使用容器的最基本内容。虽然容器是微服务的促进者和良好搭配，但微服务架构不是必需使用它，本章节的很多架构理论可以不需要容器。然而由于已经介绍过的容器的重要性，本书的关注点是在搭配使用这两种技术。

企业应用通常比较复杂并且由多个服务组成，而不是单一的服务应用。这些情况下，您需要理解额外的架构方法，比如微服务和一定的领域驱动设计（DDD）模式，以及容器编排理论。注意本章节不仅介绍基于容器的微服务，同时也涵盖了其他容器化的应用。

容器设计原则

在容器模型中，一个容器镜像的实例表示一个独立的进程。通过定义一个容器镜像作为进程边界，您可以创建一些初始状态来扩展或批处理进程。

设计容器镜像的时候，在 Dockerfile 里会看到一个[入口点 \(ENTRYPOINT\)](#) 定义。它定义了进程的生命周期，也控制了容器的生命周期。当进程结束，容器的生命周期也结束了。容器能够用来运行长期运行的进程比如 Web 服务器，也能够用来运行短期进程比如批处理任务——在 Azure 上早就已经实现了的[Web 作业 \(Job\)](#)。

如果进程失败，容器也会结束，然后编排引擎开始接管。如果编排引擎被配置成同时运行 5 个实例，然后其中一个失败了，那么编排引擎会创建另外一个容器实例来替换失败的进程。在一个批处理任务中，进程可以带参数启动。当进程结束时，整个任务也就结束了。本书后续会深入讨论编排引擎。

或许您会找到一种方式在一个容器中运行多个进程。由于每个容器只能有一个入口点，在这种方式下您需要在容器中运行一个脚本来启动多个程序。比如您可以使用 [Supervisor](#) 或者类似的工具来管理启动方式。然而即使您能找到每个容器运行多个进程的架构，这样的方式也并不多见。

容器化单体应用

或许您想创建一个单体部署的 Web 应用或服务，然后部署在容器里，应用本身或许不算单体的，它由多种库、组件、甚至很多层（应用层，领域层，数据访问层等）来构成。表面上看起来它就是一个独立的容器——一个独立的进程，一个独立的网页应用或一个独立的服务。

为了管理这种模式，您部署一个单一的容器来容纳应用。要向上扩展时，您只需要在负载均衡后面添加更多的应用拷贝。这样比较简单是因为只需要管理一个单一的容器或虚拟机部署。

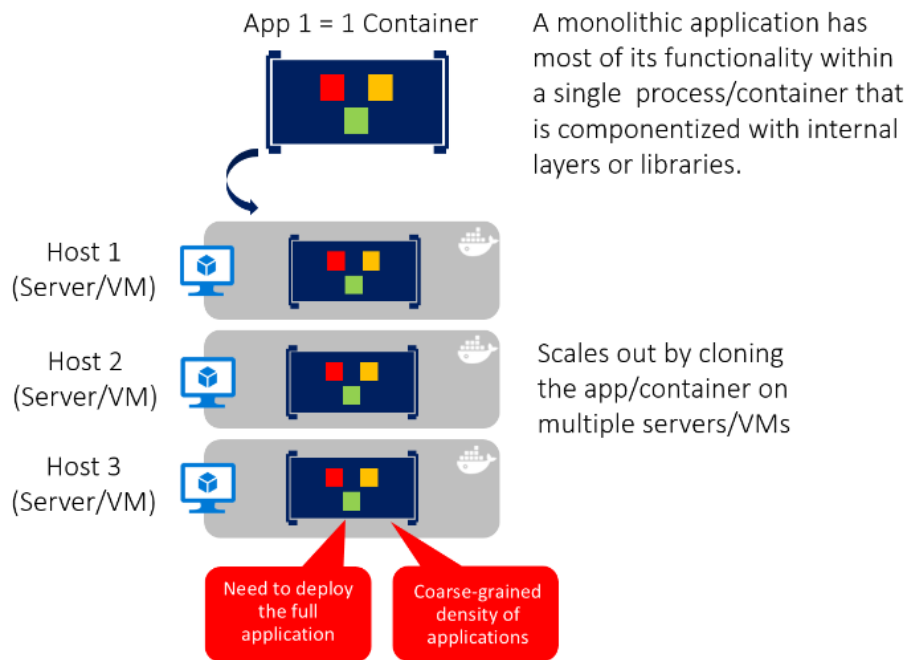


图 4-1. 单体应用容器化架构示例

您可以在每个容器中包含多个组件、库或内部分层，如图 4-1 所示。然而，这种单体式的模式或许跟容器的原则“一个容器做一件事，而且用一个进程来做”互相冲突。但在某些情况下也是可取的。

这种方式的缺陷随着应用的增长和扩展变得更明显。如果整个应用能够扩展，倒没有太大问题。然而在多数情况下，只有应用的少数瓶颈部分需要扩展，而其他组件的使用本身就很少。

比如在一个典型的电子商务应用中，您可能需要扩展产品信息子系统，因为相对于购买产品有更多的用户来浏览产品，有更多的用户使用购物篮而不是支付功能，很少的用户会添加评论或者查看购买历史。您可能只有少量的员工，而且需要管理内容和营销工作。如果您扩展单体式设计，所有这些不同任务的代码都以相同的等级部署了多次。

应用扩展的方式有很多种——水平复制但是除了扩展所有组件带来的问题之外，单个组件的变更还会导致整个应用全部重新测试，所有实例完全重新部署。

然而这种单体式应用很常见，因为比微服务方式的开发更简单，所以很多组织使用这种架构方式来开发应用，有些组织得到了还过得去的结果，但是大部分在这方面遇到了瓶颈。很多组织使用这种模式来设计他们的应用，是因为多年以前的工具和基础架构很难实现面向服务的架构（SOA）。

从基础架构的角度来看，每一台服务器作为相同的主机，都能够运行多个应用，并且具有能够接受的资源利用率。如图 4-2 所示。

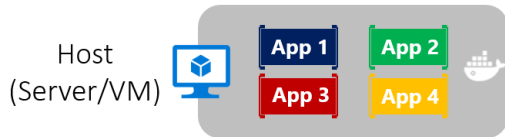


图 4-2. 单体式方法：主机里运行了多个应用，每个应用以一个容器的方式运行

单体应用在微软 Azure 上可以将每个实例部署在一个专用虚拟机 VM 中。另外，使用 [Azure 虚拟机规模集 \(VM Scale Sets\)](#) 能够轻松扩展虚拟机。[Azure 应用服务 \(App Service\)](#) 也可以用来运行单体式应用和方便地扩展实例，而且不需要管理虚拟机。从 2016 年起，Azure 应用服务也能够运行 Docker 容器的单个实例来简化部署。

在测试或受限的生产环境中，您可以部署多个 Docker 主机，然后使用 Azure 负载均衡器来管理它们，如图 4-3 所示。这样能让您以一种粗犷增长的方式来管理扩展，因为整个应用在一个单一的容器中。

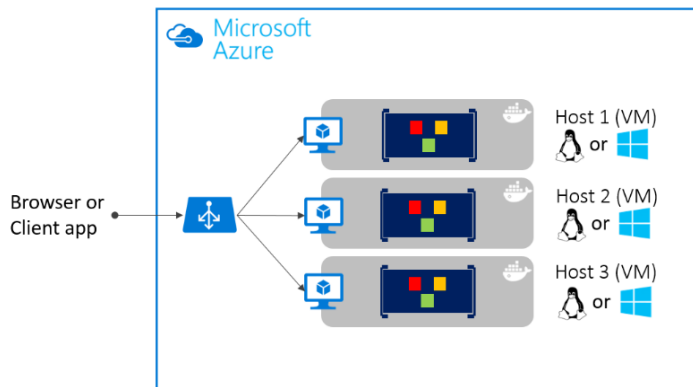


图 4-3. 多主机扩展单一容器应用示例

多主机部署的管理可以使用传统的部署技术。Docker 主机可以使用命令行管理工具比如 `docker run` 或 `docker-compose` 等进行手动管理，也可以使用持续部署 (CD) 管道等自动化方式。

用容器部署单体应用

使用容器来管理单体式应用的部署有很多优势，扩展容器实例远比部署额外的虚拟机更快更容易，即使您使用了虚拟机规模集 (VM Scale Sets)，虚拟机也需要时间来启动。如果以传统的应用实例来部署，应用的配置会作为虚拟机的一部分来管理，这不是很理想。

以 Docker 镜像来部署更新也会更快，并且网络资源消耗的更少，Docker 镜像通常几秒钟就完成启动。要下线一个 Docker 镜像实例也是简单到一个 `docker stop` 命令就可以在几秒钟内完成。

因为容器设计成不可改变的，您永远不需要担心它搞坏虚拟机。而相反，虚拟机的升级脚本可能忘记某些特殊的配置，或者残留一些文件在磁盘上。只从优势角度来看，单体式应用能够从 Docker 获益。容器的额外优势来自于使用容器编排引擎来部署，它管理着多个容器实例以及每个实例的生命周期。把单体式应用拆分成可扩展、可开发和可单独部署的子系统，是进入微服务领域的切入点。

发布单容器应用到 Azure 应用服务

不管您是要验证一个部署在 Azure 上的容器，或者遇到的只是一个简单的单容器应用，Azure 应用服务都提供了很好的方式来扩展单容器的服务。使用 Azure 应用服务很简单，它提供了良好的 Git 整合来获取代码，以及使用 Visual Studio 进行编译，然后直接部署到 Azure 上。

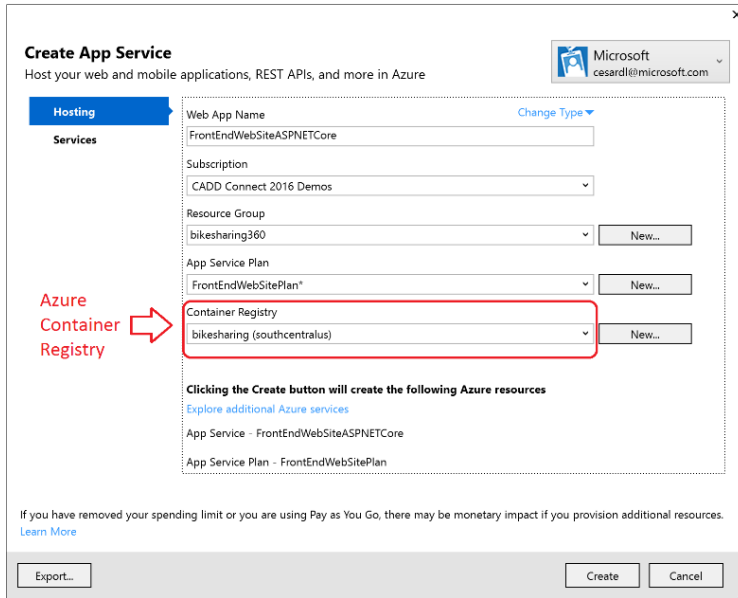


图 4-4. 从 Visual Studio 发布一个单容器应用到 Azure 应用服务

没有 Docker 的话，如果您需要其他的功能、框架或者 Azure 应用服务还不支持的依赖库等，您不得不等待 Azure 团队来更新这些依赖库，或者可以切换到其他服务比如 Azure Service Fabric，Azure 云服务（Cloud Service）甚至虚拟机，它们可以让您有进一步的控制权，您可以为应用安装必要的组件和框架。

Visual Studio 2017 开始支持容器，让您能够在应用环境中引入任何想要的东西，如图 4-4 所示。应用运行在容器里时，想要添加一个依赖到应用中，您可以在 Dockerfile 或者 Docker 镜像里添加它。

图 4-4 中的也展示了发布过程中推送一个镜像到容器注册表，它可以是 Azure Container Registry（Azure 中一个注册表，并使用 Azure Active Directory 的用户组和账户来保证安全性），或者其他的 Docker 注册表，比如 Docker Hub 或一个本地的注册表。

Docker 应用的状态和数据

大多数情况下，您可以认为一个容器就是一个进程的实例。进程不维护持久化的状态，容器能够写入本地存储，而且容器的实例可以一直存在，就像在内存中额一个位置是持久的。容器镜像跟进程类似，可以有多个实例，最终也要被结束掉。通过编排引擎来管理容器，它们可以从一个节点或虚拟机移动到另一个节点和虚拟机。

Docker 提供了一个叫做遮罩 (overlay) 文件系统的功能, 它实现了一种拷贝写入的任务来保存容器底层文件系统的更新信息, 这些信息是容器原始镜像的补充。如果容器从系统中删除, 这些更新也会丢失。因此, 虽然在容器的本地存储中保存状态是可能的, 但基于这种机制的系统设计跟容器设计的前提相冲突, 所以默认情况下是无状态的。

以下方法用来管理和持久化 Docker 应用的数据:

- 加载到宿主中的[数据卷](#)。
- [数据卷容器](#)通过额外的容器提供容器间的共享存储。
- [卷插件](#)通过加载卷为远程服务提供永久存储。
- 提供基于地理位置分布的 [Azure 存储](#), 为容器提供了良好的永久存储方案。
- 远程关系型数据库 (比如 Azure SQL 数据库) 或 NoSQL 数据库 (比如 Azure Cosmos DB) , 或者缓存服务 (比如 [Redis](#)) 。

下面详细解释这些选项。

数据卷是从宿主操作系统中映射到容器里的目录, 当容器里的代码访问这些目录时, 实际上访问宿主操作系统的文件夹。这些目录并没有绑定到容器本身的生命周期中, 它们能被直接运行在宿主操作系统中的代码访问, 或者被其他同样映射了该目录的容器访问。因此, 数据卷是设计成独立于容器生命周期来持久化数据的。如果您从 Docker 宿主里删除一个容器或镜像, 数据卷中的数据并没有被删除, 里面的数据也能够从宿主操作系统中访问。

数据卷容器是常规数据卷的升级, 一个数据卷容器是包含一个或多个数据卷的简单容器, 它为容器访问提供了统一的接入点。除此之外, 它跟普通的数据卷很相似。所以, 数据是独立地持久化在这种专用容器中, 跟应用容器的生命周期区别开来。

如图 4-5 所示, 常规 Docker 卷可以在容器之外但必须在宿主服务器或虚拟机的物理存储之内, 因此 Docker 容器无法从一个宿主服务器或虚拟机访问另外一个宿主机和虚拟机的数据卷。换句话说, 使用这样的卷是不能管理那些运行在不同 Docker 宿主里的容器之间共享的数据。

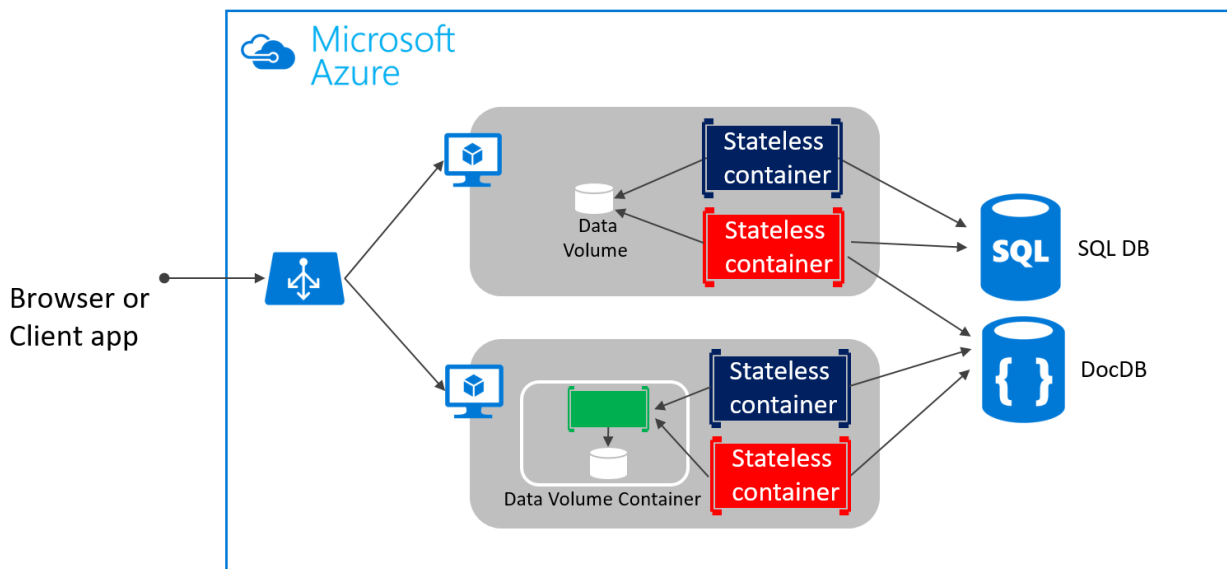


图4-5. 容器化应用的数据卷和外部数据源

另外，当 Docker 容器是由编排引擎来管理，在集群进行优化时，容器可能在宿主之间“移动”。因此不推荐使用数据卷来保存业务数据。但如果是用来保存跟踪文件、临时文件或类似的不影响业务数据一致性的文件时，它又是一个好的机制。

卷插件类似于 [Flocker](#)，为集群里所有机器提供数据访问。不是所有的卷插件都会平等创建，它通常为不可变的容器提供外部的可靠持久化存储。

远程数据源和缓存工具（如 Azure SQL 数据库、Azure Cosmos DB）或者远程缓存（如 Redis）能够在容器化应用中，跟它们不使用容器的开发方式一样。这是一种经过验证的存储业务数据的方式。

Azure 存储，业务数据通常需要保存在外部资源或数据库中，比如 Azure 存储。Azure 存储提供了下列具体的云服务：

- Blob 存储用来保存非结构化对象数据。一个块可以是任意类型的文本或二进制数据，比如文档或媒体文件（图片、音频和视频）。Blob 存储也称为对象存储。
- 文件存储为使用 SMB 协议的传统应用提供了共享存储方式。Azure 虚拟机和云服务能够通过加载共享在应用组件之间共享文件数据。
- 表存储用来保存结构化数据集。表存储是一种 NoSQL 的键值数据存储，允许快速开发和海量数据的高速访问。

关系型数据库和 NoSQL 数据库。外部数据库有很多的选择，从关系型数据库的 SQL Server、PostgreSQL、Oracle，到 NoSQL 数据库的 Azure Cosmos DB、MongoDB 等等。因为它们属于完全不同的主题，本书中不会详细解释这些数据库。

面向服务的架构

面向服务的架构（SOA）是一个被滥用的词汇，不同的人有不同的理解。不过作为一个通用标准，SOA 表示通过拆分成多个服务（通常基于 HTTP）来设计应用，这些服务要能区分成不同的类型比如子系统或分层。

这些服务现在可以作为 Docker 容器来部署，因为所有的依赖都包含在容器镜像里，这就解决了部署的问题。然而当你需要向上扩展 SOA 应用时，如果是部署在单个 Docker 主机里，或许会面临扩展性和可用性的挑战了。这就需要 Docker 集群软件或编排引擎的帮助，我们将在后续章节描述微服务部署方法时介绍它们。

Docker 容器对于传统面向服务架构的应用和更高级的微服务架构都是很有用的（但不是必需的）。

微服务是从 SOA 派生出来的，但 SOA 跟微服务是不同的架构。比如大型中心代理、企业级中心编排引擎和[企业服务总线 \(ESB\)](#) 都是典型的 SOA 功能。在微服务社区中，大多数情况下都是 SOA 的反模式设计。实际上，有的人争论说“微服务架构只是把 SOA 做对了”。

本书重点在微服务架构，因为 SOA 比微服务架构用到的要求和技术要少一些规范。要是你知道如何创建一个微服务应用，你也会知道如何创建一个简单的 SOA 应用。

微服务架构

正如名字所示，微服务架构是一种以小型服务集合来创建服务端应用的方法。每个服务在独立进程中运行，通过通信协议（比如 HTTP/HTTPS、WebSockets 或 AMQP）来彼此通信。每个微服务实现一个特定的端到端的领域或有确定边界的业务逻辑，并且每个微服务必须能够独立地开发和部署。最后，每个微服务应该拥有自己特定的领域数据模型和领域逻辑（自治和去中心化的数据管理），它们是基于不同的数据存储技术（SQL, NoSQL）和不同的编程语言实现的。

一个微服务到底有多小？当开发一个微服务时，大小不应该是重点，重点应该是创建低耦合的服务，以便能独立地开发、部署和扩展每一个服务。当然在判断和设计微服务时，只要它们之间没有太多直接的依赖，就应该使它们尽可能地小。比大小更重要的还有内部必要的一致性和对其他服务的依赖。

为什么需要微服务架构？简而言之，它提供了长期的敏捷性。微服务可以基于多个独立部署的服务来创建应用，这些服务通常是拥有细粒度和独立生命周期，这使得复杂的、大型的和高度扩展的系统拥有更好的可维护性。

微服务另外的优势在于能够进行独立的横向扩展。相对于单体应用必须作为一个整体来扩展，微服务可以只扩展特定的部分，这样就能够只扩展那些需要更多处理资源或网络带宽的功能区域来满足需求，而不是一起将并不需要扩展的其他功能区域也进行扩展。这意味着节约了成本，因为使用了更少的硬件。

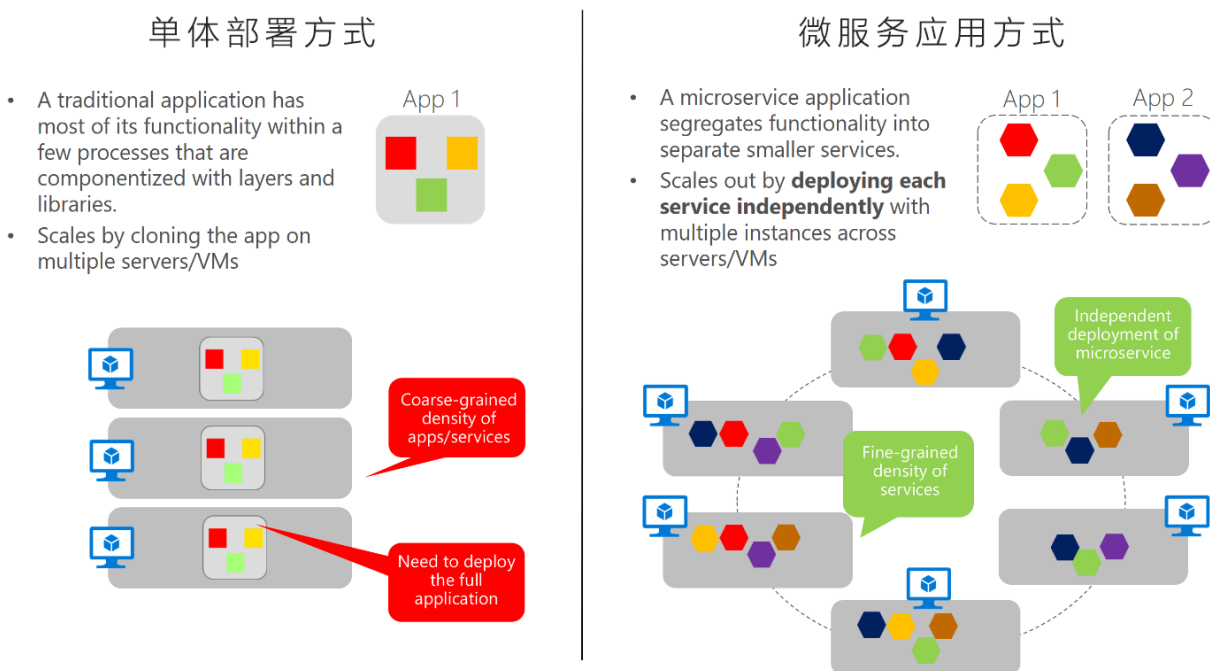


图 4-6. 单体式部署和微服务方式的对比

如图 4-6 所示，您可以只变更复杂的、大型的、可扩展的应用程序的特定的小块区域，所以微服务方式允许对每个微服务进行敏捷变更和快速迭代

架构一个基于微服务的细粒度应用，需要启用持续集成和持续部署，这也可以加速新功能的发布。细粒度构成的应用也使得独立地运行和测试微服务成为可能，并且通过维护清晰的交互协议来自主进化。只要不变更接口和协议，就能够在任何一个微服务里面修改内部实现或添加新功能，而不会破坏其他微服务。

要想成功将微服务系统投入生产环境，以下几点很重要：

- 服务和基础架构的监测和监控状态。
- 为服务扩展基础架构（云和编排引擎）
- 在多个层级设计和实现安全性：认证、授权、密文密码管理和安全通信等等。
- 快速应用发布，通常由不同的团队来实现不同的微服务。
- DevOps 和 CI/CD 的实践和架构。

在这么多方面中，本书只会涵盖和介绍前面三条。后面两条涉及到应用生命周期，在另外一本书 [《使用微软平台和工具的容器化 Docker 应用的生命周期》](#)里做了介绍。

其他资源

- **Mark Russinovich. 微服务：云带来的应用革命**
<https://azure.microsoft.com/blog/microservices-an-application-revolution-powered-by-the-cloud/>
- **Martin Fowler. 微服务**
<http://www.martinfowler.com/articles/microservices.html>
- **Martin Fowler. 微服务的先决条件**
<http://martinfowler.com/bliki/MicroservicePrerequisites.html>
- **Jimmy Nilsson. 分块云计算**
<https://www.infoq.com/articles/CCC-Jimmy-Nilsson>
- **Cesar de la Torre. 使用微软平台和工具的容器化 Docker 应用的生命周期（可下载的电子书）**
<https://aka.ms/dockerlifecyleebook>

微服务的数据自治

微服务架构的一条重要规则是每一个微服务必须拥有领域逻辑和数据，跟一个完整的应用有逻辑和数据类似，微服务的逻辑和数据在一个独立的生命周期中，能够独立的进行开发。

这意味着子系统和微服务的领域概念模型会有差别。考虑一个企业应用，比如客户关系管理（CRM）系统，交易记录子系统和客户支持子系统都会调用客户实体的属性和数据，而它们是隶属于不同的上下文边界的。

这样的原则在[领域驱动设计（DDD）](#)里也是相似的，每一个[限界上下文](#)、自治的子系统或服务必须拥有自己的领域模型（数据 + 逻辑 + 行为）。每一个 DDD 限界上下文关联到一个业务的微服务（一个或多个服务）。（在下一章我们会展开来讲限界上下文的设计。）

另一方面，传统设计（单体式数据）使用单一的中心化数据库，或者有限的几个数据库。通常是一个符合范式的 SQL 数据库，用在整个应用和所有内部子系统中。如图 4-7 所示。

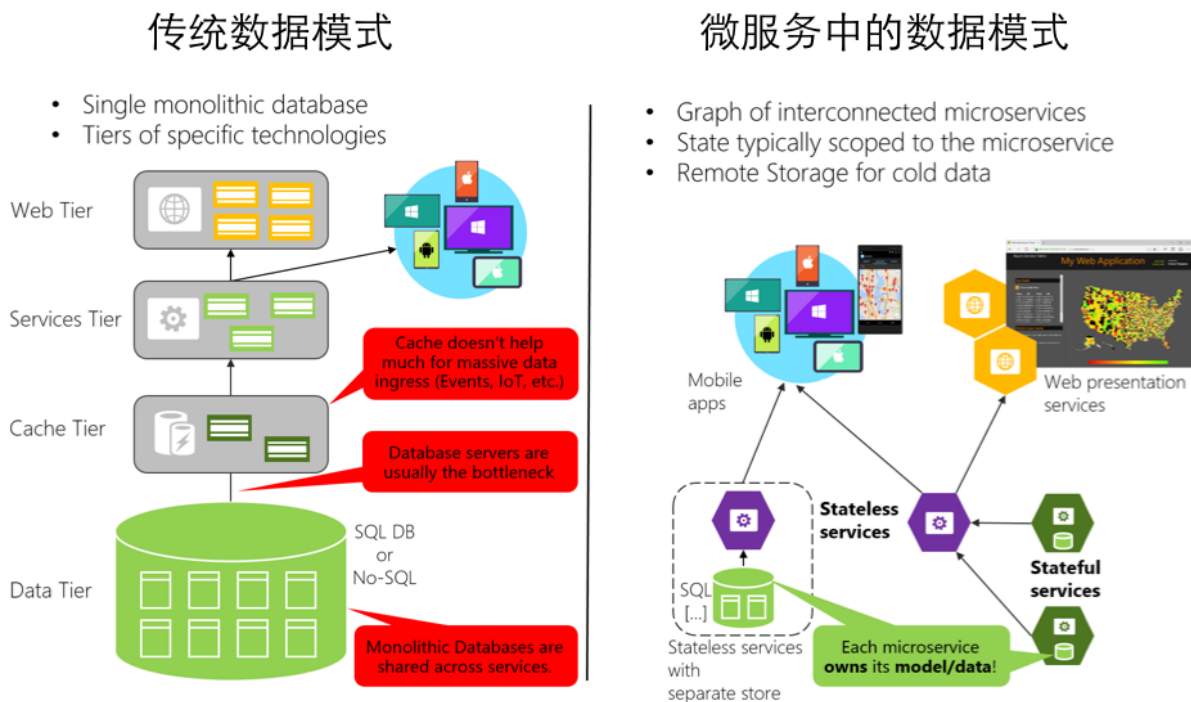


图 4-7. 数据自治对比：单体式数据库和微服务

这种中心化数据库的方式在初期看起来比较简单，好像能够在不同子系统间复用实体来做到所有东西一致。但现实情况是为了满足多个不同的子系统，需要建立非常大的表，包含大部分情况下都不需要的属性和字段。这就像使用同一张地形图来满足徒步旅行，长途汽车旅行和学习地理知识的需求。

具有单一关系型数据库的单体式应用有两个重要的优点：在应用程序的所有数据库表和数据层面，都支持 [ACID 事务](#) 和 SQL 语言。这种方式能简单地就可以写出关联多张表组合数据的查询语句。

然而，微服务架构的数据访问会变得更加复杂。即使在一个微服务或限界上下文里能够或者应该做到 ACID 事务一致，但每个微服务的数据是独立的，所以只能通过微服务的 API 来访问。将数据进行封装确保了微服务是松耦合且能够独立变化的。如果多个服务访问相同的数据，数据的更新就要求协调同步到所有服务，这将会破坏掉微服务生命周期的自治性。这就表示当业务流程跨越多个微服务时，最终一致性是唯一的办法。这比写一个简单的 SQL 连接要复杂得多。类似的，很多其他关系型数据库功能也不支持跨微服务使用。

更进一步，不同的微服务通常使用不同种类的数据库。对于现代的应用存储和对不同类型数据的处理，关系型数据库也不总是最佳选择。一些场景下，使用 NoSQL 数据库（比如 Azure DocumentDB 或 MongoDB），会比 SQL 数据库（比如 SQL Server 或 Azure SQL Database）有着更方便的数据模型和更好的性能以及扩展性。在其他方面，关系型数据库仍然是最佳方式。因此，基于微服务的应用通常混合使用 SQL 和 NoSQL 数据库，有时候被称为[混合数据持久化 \(polyglot persistence\)](#)。

这种隔离的混合数据持久化架构有很多优点，包括服务的松散耦合，更好的性能、扩展性，更低成本和可管理性等。然而也带来了一些分布式数据管理的挑战，在本章后续的“[识别领域模型边界](#)”一节详细介绍。

微服务和限界上下文模式的关系

微服务的理论是从[领域驱动设计 \(DDD\)](#)的[限界上下文 \(BC\) 模式](#)继承而来。DDD 将大型业务划分成多个 BC，并明确它们的边界，每个 BC 必须有自己的模型和数据库。类似的，每个微服务也拥有跟自己相关的数据。另外，每个 BC 通常具有自己的[通用语言](#)用来在开发人员和领域专家之间进行沟通。

在不同限界上下文的表示语言里的信息（主要是领域实体）可以有不同的名字，即使不同的领域实体共用相同的标识（即唯一 ID 用来从存储中读取实体）。比如在一个用户资料的限界上下文中，用户 User 领域实体可能跟订单限界的买家 Buyer 领域实体共享标识。

所以微服务很像限界上下文，但它也表明了是分布式服务，每一个都作为独立的进程被创建，它必须使用先前提到的分布式协议比如 HTTP/HTTPS、WebSockets 或 [AMQP](#)。然而限界上下文模式并没有明确一个 BC 是分布式服务或者只是单体式部署应用里的简单逻辑边界（比如通常的子系统）。

需要强调的是为每个限界上下文定义一个服务是很好的开始，但也没必要局限于此。有时候必须设计一个由多个物理服务组成的限界上下文或业务微服务。最根本的是，限界上下文和微服务这两种模式都极其相似。

DDD 也从分布式的微服务具有良好边界中获益，比如不需要在微服务之间共享模型，则在限界上下文中也是如此。

其他资源

- **Chris Richardson. 模式：服务独立的数据库**
<http://microservices.io/patterns/data/database-per-service.html>

- **Martin Fowler. 限界上下文**
<http://martinfowler.com/bliki/BoundedContext.html>
- **Martin Fowler. 混合持久化**
<http://martinfowler.com/bliki/PolyglotPersistence.html>
- **Alberto Brandolini. 使用上下文映射的策略性领域驱动设计**
<https://www.infoq.com/articles/ddd-contextmapping>

逻辑架构和物理架构

现在是时候讨论逻辑架构和物理架构的区别了，以及如何用在微服务应用的设计中了。

首先，创建微服务并不要求任何特定的技术，比如 Docker 容器就不是必需的。微服务能够作为普通进程运行，微服务是一种逻辑架构。

其次，即使微服务能够以一个独立的服务、进程或容器（为了简单起见，这就是 [eShopOnContainers](#) 初始版本采用的方式）来实现，这种在业务微服务和物理服务或容器的同等性不是必要的，尤其是当您创建一个由成百上千的服务组成的大型复杂应用时。

这就是一个应用的逻辑架构和物理架构存在差别的地方。系统的逻辑架构和逻辑边界跟物理或部署架构没必要一一对应，虽然能够这么做，但是不常见。

您或许已经识别出了业务微服务或限界上下文，但不意味着最佳实现方式就是为每个业务微服务创建单独的服务（比如作为一个 ASP.NET Web API）或单独的 Docker 容器。没有规定说每个业务微服务必须使用单独服务或容器来实现。

因此，业务微服务或限界上下文是一个逻辑架构，或许是（也可以不是）跟物理架构一致。重点在于业务微服务或限界上下文必须自主地进行代码和状态的独立版本控制、部署和扩展。

如图 4-8 所示，目录业务微服务由多个服务或进程组成，它们可以是多个 ASP.NET Web API 或者是其他使用 HTTP 或其他协议的服务。更重要的是这些服务共享相同的数据，因为它们都结合在一个相同的业务领域里。

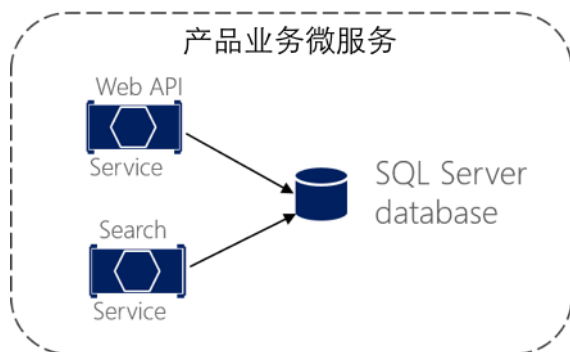


图 4-8. 多个物理服务组成的业务微服务

例子中的 Web API 服务和搜索服务使用同一个数据源，它们共享相同的数据模型。所以在物理实现业务微服务时，把功能做了拆分，以便能够按需向上或向下扩展每个内部的服务。比如 Web API 通常需要更多的运行实例，或者是反过来的。

简而言之，微服务的逻辑架构通常跟物理架构不一样。本书提到的微服务是业务或逻辑上的微服务，可以对应到一个或多个服务，大多数情况下对应着一个服务，但也可以是多个。

分布式数据管理的挑战和解决方案

挑战 1: 如何定义微服务边界

定义微服务的边界很可能是每个人碰到的第一个挑战。每个微服务必须成为应用的一小部分，每个微服务应该是自治的，这带来了优势也带来了挑战。那么要如何定义那些边界呢？

首先，您需要关注应用的逻辑领域模型和相关的数据库。必须尝试识别同一个应用中解耦的数据孤岛和不同的上下文。每个上下文都可以有不同的商业语言（不同的业务术语），上下文应该独立地定义和管理。在这些不同上下文中使用的术语和实体可能听起来相似，但是您可能会发现，它们在特定的上下文中，一个业务概念在另一个上下文中是用于不同的目的，甚至可能具有不同的名称。例如，用户可以被称为身份或成员资格上下文中的用户，CRM 中的客户，订单上下文中的买方等等。

识别多个具有不同领域上下文的应用边界的方法，也能用来识别每个业务微服务和相关领域模型和数据库的边界。您总是尝试微服务之间有最小程度的耦合，本书将在后续的“[识别微服务的领域模型边界](#)”一节里详细介绍。

挑战 2: 如何创建从多个微服务获取数据的查询

第二个挑战是如何实现从多个微服务获取数据的查询，同时避免远程客户端和微服务之间不必要的通信。比如一个移动 App 需要一个页面来展示由购物车、产品目录和用户身份微服务包含的用户信息。再如一个复杂的报表系统涉及到位于多个微服务的多个表。正确的解决方案取决于查询的复杂性。但无论如何都需要一种方式来聚合信息，以便提高系统的通信效率。最流行的解决方案如下。

API 网关：对于多个拥有不同数据库的微服务的简单数据聚合，推荐的方式就是通过一个作为 API 网关的聚合微服务。然而需要小心使用这种模式，因为它可能成为系统的瓶颈，也可能违反微服务自治的原则。为了降低这些可能性，您可以使用多个细粒度的 API 网关，每一个聚焦在系统的一个垂直“切片”即业务领域。API 网关模式将在后续的“[使用 API 网关](#)”一节中做详细介绍。

CQRS 查询/读取表：另一种聚合多个微服务数据的方案是[物化视图模式](#)，这种方案提前（在实际查询发生之前准备非规范的数据）生成包含多个微服务的数据的只读表，这个表具有适用于客户端应用需要的格式。

假设有一个移动 App 的界面，如果有一个单独的数据库，就可以使用一个连接多个表的复杂 SQL 查询来获取界面所需的全部数据。但是当您使用数据库分布在不同微服务的多个数据库时，就不能通过创建

SQL 连接语句来查询数据了。这个复杂的查询变成了一个挑战。这时就可以使用 CQRS 方案来满足要求：在不同的数据库中创建一个只用作查询的非规范表，这个表可以为复杂查询所需的数据专门设计，把应用界面所需的字段和查询表的字段一一对应。查询表也可以用在报表中。

这种方式不仅解决了最初的问题（如何跨微服务查询数据），跟复杂的 SQL 连接语句相比也有相当多的性能提升，因为应用所需的数据已经在查询表里了。当然，使用命令查询职责分离（CQRS）的查询/读取表需要额外的开发工作，以及需要接受数据最终一致性。然而在需要高性能和高扩展性的[协作场景](#)（或者竞争场景，取决于视角）下，应该使用 CQRS 来处理多数据库。

中心数据库的“冷数据”：对于复杂的报表查询或许不需要实时的数据，一种通用的方案是将“热数据”（微服务里的交易数据）导出为“冷数据”存储到报表专用的大型数据库中，中心数据库系统可以是基于大数据的系统比如 Hadoop，可以是数据仓库比如 Azure SQL 数据仓库，甚至是一个单独的报表专用 SQL 数据库（如果容量不是问题的话）。

需要注意的是中心数据库应该只用于不需要实时数据的报表查询，作为事实数据源的原始更新和交易数据必须在微服务中。同步数据的方式可以采用事件驱动通信（下一章会介绍）或者其他数据库底层的导入/导出工具。如果使用事件驱动通信的方式，整合流程就跟前面的使用 CQRS 查询表获取数据的方式很类似。

然而，如果您的应用设计需要不断地从多个微服务里进行复杂查询整合数据，这可能会变成一个糟糕的设计，一个微服务应该尽可能地独立于其他微服务（使用中心冷数据库的报表分析系统除外）。遇到这个问题通常是合并微服务的理由，您需要平衡每个微服务的自治、进化和部署，以及强依赖、高内聚和数据聚合。

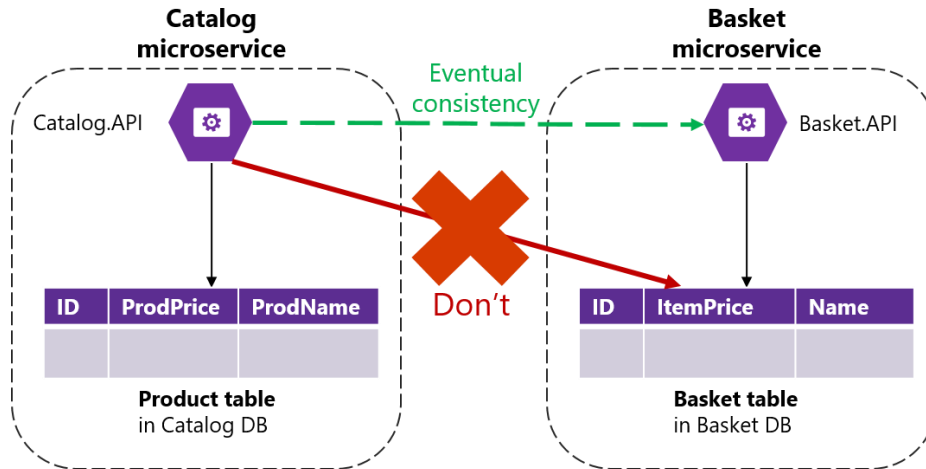
挑战 3: 如何在多个微服务之间实现一致性

如前所述，每个微服务拥有的数据是私有的，只能通过微服务 API 来访问，因此，面临的一个挑战就是如何跨多个微服务保持一致性的同时实现端到端的业务逻辑。

为了分析这个问题，让我们看看[示例应用 eShopOnContainers](#) 中的一个例子。目录微服务维护着所有的产品信息，包括价格。购物篮微服务管理着用户加入购物篮的临时产品数据，包括添加购物篮时的产品价格。当产品服务中一个产品的价格发生了变化，购物篮中相同产品的价格也应该变化，另外，系统应该告诉用户说购物篮里的某个产品的价格变了。

假如这个应用有一个单体式的版本，当产品表的价格发生变化时，产品子系统能够简单地使用 ACID 事务来更新购物篮表里的价格。

然而在一个微服务的应用里，产品表和购物篮表被各自的微服务所占有。任何微服务不应该在自己的事务中包含其他微服务的表或存储，即使是直接查询也是不可以的。如图 4-9 所示。



Databases are private per microservice

图 4-9. 微服务不能直接访问其他微服务的表

目录微服务不能直接更新购物篮表，因为购物篮表被购物篮微服务占有。要更新购物篮微服务，目录微服务应该使用基于异步通信（如集成事件（消息和基于事件的通信）的最终一致性。这就是示例应用 [eShopOnContainers](#) 如何跨微服务解决一致性的。

根据 [CAP 理论](#)，您需要在可用性和强 ACID 一致性之间作出选择。大多数微服务的场景要求高可用性和高扩展性，而不是强一致性。重要应用必须保持随时在线，开发人员可以使用弱一致性或最终一致性的技术来做到强一致性。这也是大多数基于微服务的架构所采取的方法。

此外，ACID 风格或两步提交事务不仅违背微服务原则，大多数 NoSQL 数据库（比如 Azure 文档数据库（Azure DocumentDB），MongoDB 等）也不支持两步提交事务。然而，跨服务和数据库维护数据的一致性非常重要。这个挑战也关系到当某些数据需要做冗余时如何跨微服务来执行变更的问题 - 比如当您更新目录微服务和购物篮微服务中的产品名称或描述时。

因此，作为结论，解决这个问题的好方法就是在微服务之间使用事件驱动通信和发布订阅系统来实现最终一致性。这些主题会在本书后续的“[异步事件驱动通信](#)”一节中介绍。

挑战 4: 如何在多个微服务之间通信

跨微服务边界的通信是一个真正的挑战。在这里，“通信”不是指该用什么协议（HTTP 和 REST，AMQP，消息等等），而是指该是使用什么样的通信风格，尤其是怎样耦合微服务。依赖于耦合等级，在宕机发生时，对系统的影响会变得明显。

像微服务这样的分布式系统有许多在多个服务器或主机之间移动的产物，组件最终会挂掉，部分宕机甚至大面积失效也会发生，面对这样的分布式系统，所以需要在设计微服务和跨服务通信时就考虑这些常见的风险。

一种流行的方法是采用比较简单的基于 HTTP (REST) 的微服务，这种 HTTP 的方式完全可以接受的。问题在于如何使用它，如果微服务使用 HTTP 的请求响应的方式来跟客户端应用或 API 网关交互，这是可以的。但如果在多个微服务间创建了长串的同步 HTTP 调用，在它们之间传递的又是单体式应用的微服务对象，那您的应用最终会碰到问题。

举例来说，假设您的客户端应用发起了一个 HTTP API 请求到一个独立的微服务比如订单微服务，如果订单微服务使用同样的 HTTP 请求响应方式来依次调用额外的微服务，这就创建了一个 HTTP 调用链条。也许听起来是合理的，然而要继续这条路径，有一些重要的观点需要考虑：

- 阻塞和性能低下。由于 HTTP 的同步本质，最初的请求在所有内部 HTTP 请求全部完成之前不会有响应结果。假设这样的请求量在逐步增长，同时某个中间微服务的 HTTP 调用被阻塞了，结果就是性能受到影响，并且整体的扩展性由于额外的 HTTP 请求的增加带来了几何级增长的影响。
- 使用 HTTP 耦合了微服务。业务微服务不应该被其他业务微服务耦合，理想情况下，它们不应该“知道”其他微服务的存在。如果您的应用像例子中一样依靠耦合的微服务，要做到每个微服务自治会变得几乎不可能。
- 任何微服务引起的宕机。如果您实现了使用 HTTP 调用的链式微服务，当任何一个微服务宕机（最终它们都会挂掉）时整个微服务链会挂掉。一个微服务系统应该设计成在部分宕机情况下尽可能地继续正常运行。即使您的客户端逻辑使用了越来越快和灵敏的重试机制，HTTP 调用链越复杂，实现基于 HTTP 的容错策略就越复杂。

实际上，如果您的内部微服务采用上述的链式 HTTP 请求来通信，可能发生争论的地方在于您拥有一个单体式应用，它的进程间是基于 HTTP 的而不是使用内部的通信机制。

因此，为了促进微服务的自治和更具有弹性，应该减少使用跨服务的链式请求/响应的通信。推荐内部微服务通信只使用异步交互，或者使用异步的基于消息或事件的通信，或者使用跟原始 HTTP 请求/响应独立开的 HTTP polling 方式。

本书后续章节“[异步整合方式增强微服务自治](#)”和“[异步消息通信](#)”详细介绍异步通信。

其他资源

- **CAP 定理**
https://en.wikipedia.org/wiki/CAP_theorem
- **最终一致性**
https://en.wikipedia.org/wiki/Eventual_consistency
- **数据一致性入门**
<https://msdn.microsoft.com/library/dn589800.aspx>
- **Martin Fowler. CQRS (命令查询职责分离)**
<http://martinfowler.com/bliki/CQRS.html>

- **物化视图**
<https://docs.microsoft.com/azure/architecture/patterns/materialized-view>
- **Charles Row. ACID 和 BASE: 数据库事务处理的 pH 值变换**
<http://www.dataversity.net/acid-vs-base-the-shifting-ph-of-database-transaction-processing/>
- **补偿事务**
<https://docs.microsoft.com/azure/architecture/patterns/compensating-transaction>
- **Udi Dahan. 面向服务的组合**
<http://udidahan.com/2014/07/30/service-oriented-composition-with-video/>

识别微服务的领域模型边界

虽然微服务应该尽可能地趋向于小型化，识别每个微服务的模型边界不是为了尽可能地拆成细粒度，相反，目标应该是根据领域知识来进行最有意义的划分。重点不在于大小而在于业务需要。另外如果基于大量的依赖关系来看，应用的某个领域有清晰一致的需求，这也就表明应该是一个独立的微服务。这个一致的特点是用来识别如何拆分或者组合微服务的一种方法。本质上，当您获得越多的领域知识，就越能够适配微服务的大小。找到正确的大小不是一步到位的。

[Sam Newman](#) 是公认的微服务布道师和书籍《[微服务设计](#)》的作者，强调应该基于前面提到的限界上下文 (BC) 模式 (领域驱动设计的一部分) 来设计微服务。有时候，一个 BC 能够由多个物理的服务组成，反之则不一定成立。

一个包含特定领域实体的领域模型存在于具体的 BC 或微服务中。一个 BC 界定了领域模型的适用性，给予开发团队成员一个清晰共用的关于哪些必须结合哪些可以独立开发的理解。这对于微服务也是相同的。

另一个会影响设计决策的工具就是[康威定律](#)，它指出一个应用会影响创造这个应用的企业的组织架构。有时候反过来也成立 - 公司的组织架构围绕软件来组成。或许您想不按照康威定律而是建立您想象的公司组织架构，那么学学业务流程咨询吧。

一种能用来识别限界上下文的 DDD 模式是[上下文映射模式](#)。使用上下文映射您可以识别应用中不同的上下文和它们的边界。现实中通常每个子系统有不同的上下文和边界。上下文映射是一种明确并定义那些领域之间边界的方式。一个 BC 是自治的，包含了单独领域的详情比如领域实体，并定义了跟其他 BC 的交互接口。这跟微服务的定义相似：它是自治的，它实现特定的领域功能，以及它必须提供接口。这就是为什么上下文映射和限界上下文模式是用来识别微服务的领域模型边界的好方法。

当设计一个大型应用时，您会看到它的领域模型是如何拆得支离破碎的，比如一个产品领域的专家和一个物流领域的专家会给产品和库存领域的实体进行不同的命名，再比如一个想把客户所有细节都保存下来的 CRM 专家和一个只想要客户部分数据的订单领域专家在处理用户领域实体时会有不同的大小和字段数量。很难去消除一个大型应用里所有领域术语的歧义，但更重要的不是尝试统一术语，而是接纳每个领域的差异和丰富性。如果去尝试为整个应用建立统一的数据库，企图达到统一词汇的目的是很奇怪的，并且对于任何领域专家都是不合适的。因此，BC (以微服务实现) 会帮助澄清哪里需要特定的领域术语和哪里需要把系统拆分成额外的不同领域的 BC。

如果在领域模型之间有很少的强关联，您就知道您得到了每个 BC 和领域模型的正确边界和大小。当进行常规的应用操作时，您通常不必整合多个领域模型的信息。

每个微服务的领域模型到底有多大？这个问题最佳的答案也许是这样的：它应该有一个自治的尽可能独立的 BC，使得您没有不断地在其他上下文 (其他微服务的模型) 之间切换。如图 4-10 所示，您能看到多个微服务 (多个 BC) 自己的领域模型，以及实体是怎样定义的，这取决于应用每一个识别出来的领域的特定需求。

识别每个微服务或限界上下文的领域模型

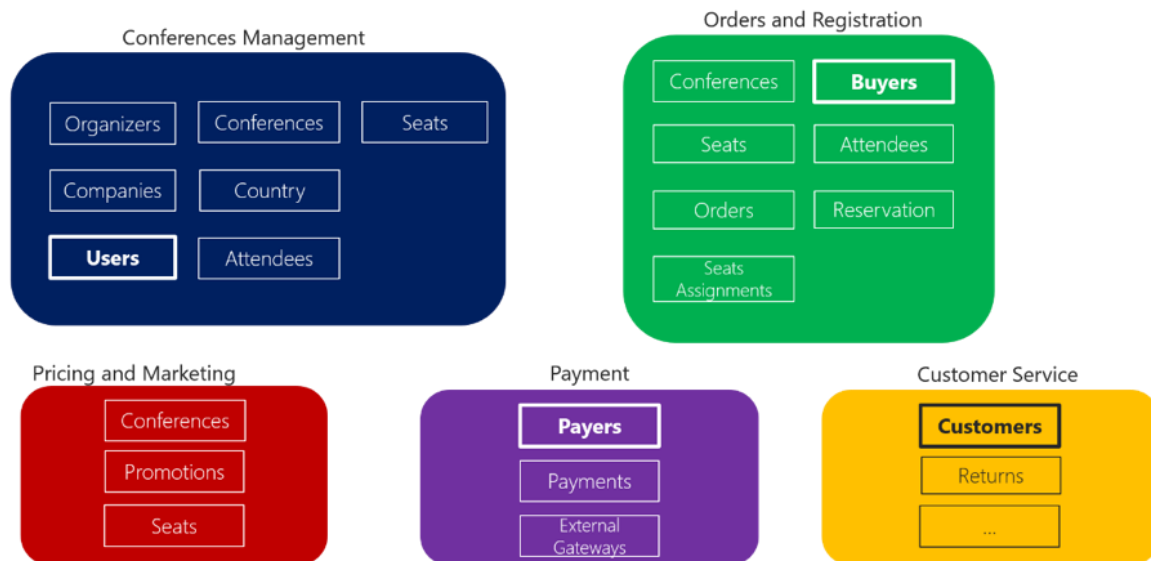


图 4-10. 识别实体和微服务模型的边界

图 4-10 展示了一个在线会议管理系统的场景示例。基于领域专家为您定义的领域信息，您已经识别出多个能够作为微服务来实现的 BC 了。如您所见，有些实体只存在于一个独立的微服务里，比如支付微服务里的支付信息，这些将会很容易实现。

然而，也有一些具有不同外形但却共用相同标识的实体，它们存在于多个微服务的多个领域模型中。比如用户实体在会议管理微服务里，这个有同样标识的同样的用户，在订单微服务里被称为买家，或者在支付微服务里命名为付款方，甚至在客户服务微服务里称作客户。这是因为，基于每个领域专家使用的通用语言，对一个用户有不同的视角，甚至包含不同的属性。在会议管理微服务里的用户实体应该包含最多的个人数据属性，但是跟用户相同的支付微服务里的买家和客户服务里的客户就不需要相同的属性列表。

图 4-11 说明了类似的方法。

将传统数据模型拆分成多个领域模型

(每个微服务或限界上下文一个领域模型)

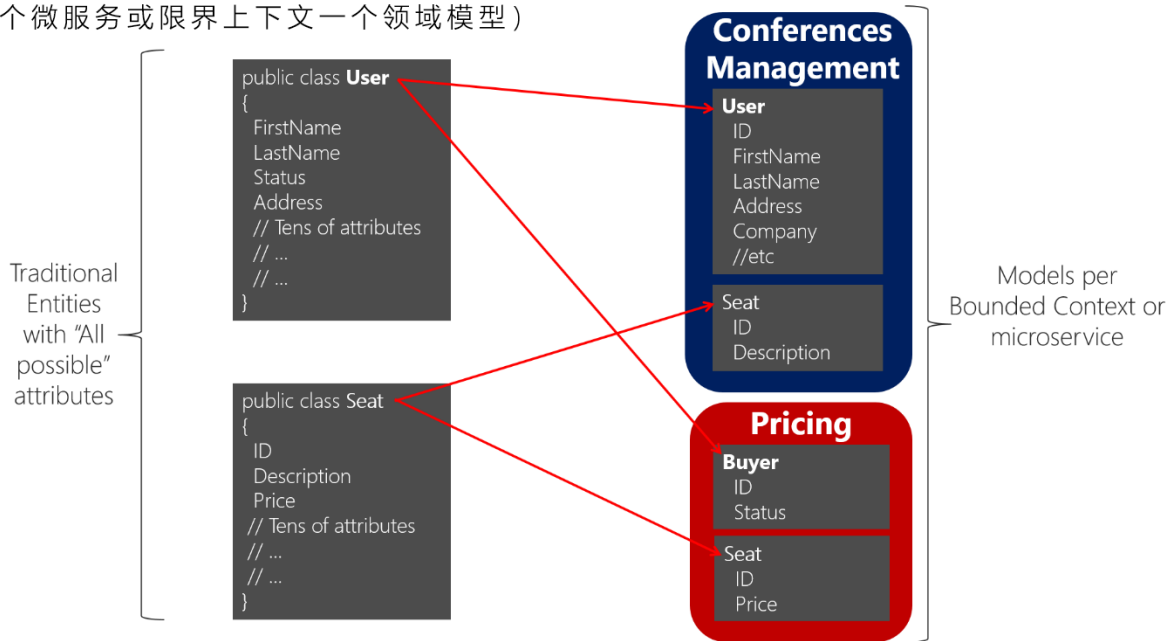


图 4-11. 传统数据模型分解成多个领域模型

您能看到“用户”是如何在会议管理微服务里表示为 User 实体的，以及在定价微服务里以 Buyer 实体来表示买家并且增加了作为买家需要的额外属性和细节。每个微服务或 BC 也许不需要 User 实体的全部数据，根据要解决的问题或上下文，只要一部分即可。比如在定价微服务里就不需要用户的地址，只要 ID（作为标识）和状态就可以在为每个买家定价时决定座位折扣。

座位 Seat 实体在每个领域模型里有相同的名字但是有不同的属性，它们共用相同的 ID 作为标识，这跟 User 和 Buyer 一样。

从根本上来看，存在于多个共用用户标识的服务（或领域）里的用户有一种通用的概念，但是每个领域模型里的用户实体可能有不同的额外细节。因此，需要一种方式来把一个领域（或微服务）的用户实体映射到另外一个领域（或微服务）。

在领域间共用的用户实体不使用相同的属性有许多好处。其中之一是减少了重复，所以微服务模型里就不包含任何不需要的数据。另一个好处是有了专一的微服务，它包含每个实体的特定类型的数据，更新和查询都由这个微服务发起。

客户端微服务直连和 API 网关模式

在微服务架构中，每个微服务暴露了一套（通常情况下）定义良好的接口，这会影响到客户端和微服务之间的通信，本节将作介绍。

客户端微服务直连

一种方法是使用客户端和微服务直接连接通信的架构，客户端应用能够直接向微服务发起请求，如图 4-12 所示。

客户端-微服务直连通信架构

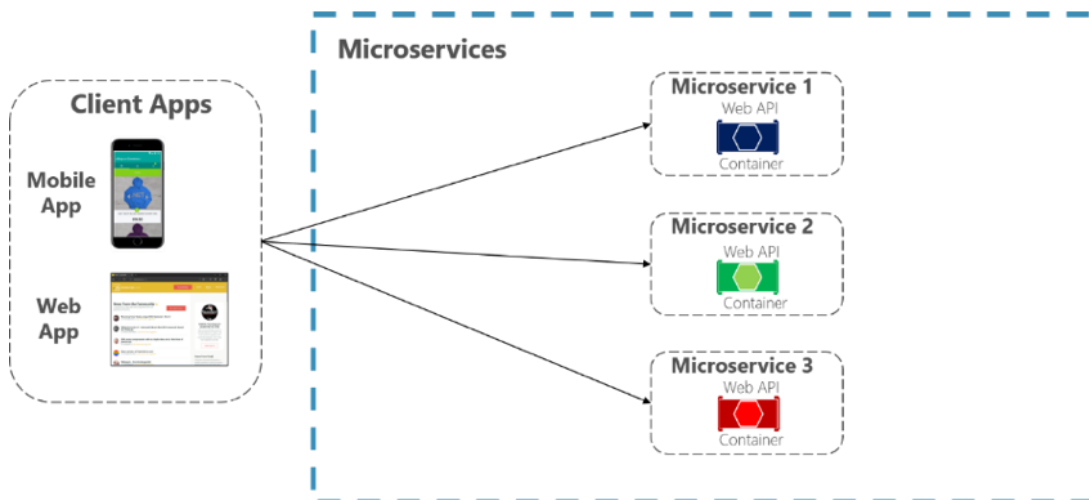


图 4-12. 使用客户端微服务直连的架构

这种方式中，每个微服务有一个公开的端点，有时候每个微服务有不同的 TCP 端口。比如某个特定的服务在 Azure 上的 URL 地址如下：

`http://eshoponcontainers.westus.cloudapp.azure.com:88/`

在一个生产环境集群中，这个 URL 应该映射到集群的负载均衡上，用来把请求分发给微服务。在生产环境的微服务和互联网之间，您应该有一个应用分发控制器（ADC）比如 [Azure 应用网关 \(Application Gateway\)](#)，它作为一个透明层不仅用来均衡负载，也提供了 SSL 来保证服务安全。它通过转移 CPU 密集的 SSL 处理和其他路由任务到 Azure 应用网关（Application Gateway）来提升主机的负载能力。无论如何，从架构视角来看，负载均衡和 ADC 对于逻辑应用是透明的。

客户端微服务直连架构用在小型微服务系统里应该是足够的，尤其是当客户端是服务器上的 Web 应用比如 ASP.NET MVC 时。但在创建大型复杂的微服务应用（比如要处理几十个微服务类型）时，尤其是客户端都是远程的移动应用或 SPA Web 应用时，这种方法就会碰到一些问题。

在开发一个大型微服务应用时，应该考虑如下问题：

- 客户端应用如何降低到后台的请求数量，以及减少跟多个微服务的无效交互？

一个单独的 UI 界面跟多个微服务交互增加了网络交互的数量，也增加了 UI 的延迟和复杂程度。理想情况是在服务端有效地聚合这些响应，这样可以降低延迟。当多块数据并行地返回时，一些 UI 能够在数据到位时立即更新。

- 应该如何处理界限问题比如授权、数据传输和动态的请求派发？

为每个微服务实现安全和界限问题比如安全性和授权需要大量的开发投入。为了限制从外部直接访问，一种可能的途径是把这些服务放进 Docker 主机或内部集群中，再通过一个中间位置比如 API 网关来实现界限问题。

- 客户端如何与使用非互联网友好协议的服务进行交互？

服务端使用的协议（比如 AMQP 或二进制协议）通常在客户端并不支持。因此，请求必须通过像 HTTP/HTTPS 的协议来发起并在后端转换成其他协议。中介者方法能满足这个场景。

- 如何设计外观模式尤其是面向移动应用时？

很多微服务的 API 也许并不能很好满足不同客户端的需要。比如移动端的要求跟网页应用就不一样，面向移动端需要更进一步的优化以便数据响应更有效率，或许您可以通过整合多个微服务的数据来返回单个数据集来实现，有时候要去掉在响应中的移动端不需要的数据，当然也要压缩数据。所以这样的场景下，在移动端和微服务之间的外观模式或 API 就变得很方便了。

使用 API 网关

当设计和创建支持多种客户端的大型或复杂的微服务应用时，一种好办法是考虑使用 API 网关，它是为多个微服务提供单个入口的一种服务，类似于面向对象设计的外观模式，但这里是指分布式系统的一部分。API 网关模式有时候也被称为“面向前端的后端”，因为它是为了满足客户端需要而创建的。

图 4-13 展示了 API 网关是如何融合在微服务架构中的。

在图中需要强调的一点是，您可以使用单一的自定义 API 网关服务来面对多个不同的客户端应用，这会是一个极大的风险，因为 API 网关服务会随着客户端应用更多的需求增长而改进。最终它会因为那些不同的需求而变得臃肿，在效果上就跟一个单体的应用或服务很相似。这就是为什么要极力推荐将 API 网关拆分成多个服务或多个小型的 API 网关，每一个 API 网关都有自己的形式。

使用 API 网关服务

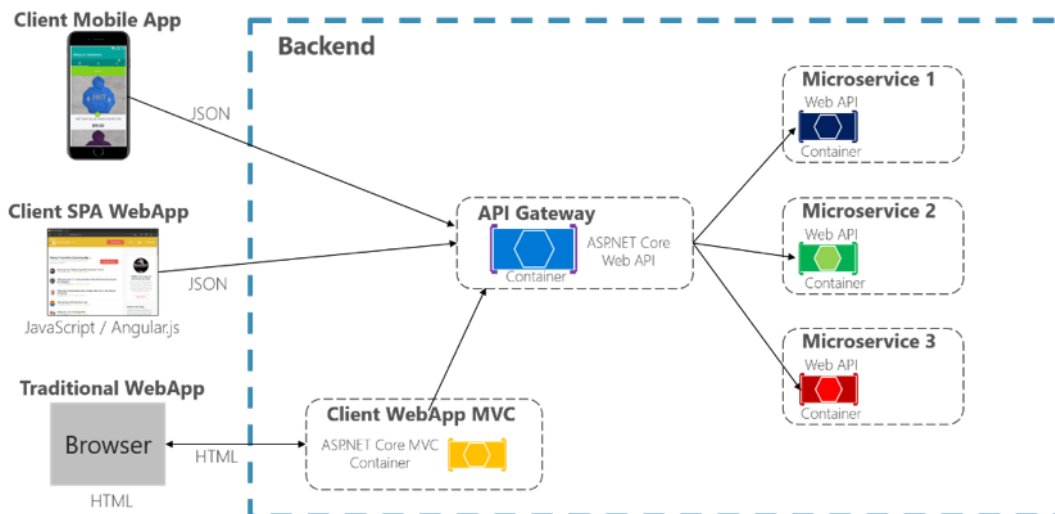


图 4-13. 在微服务架构中使用 API 网关模式

在这个例子里，API 网关使用自定义的 Web API 来实现，并运行在容器中。

如上所述，您应该创建多个 API 网关以便为每种客户端的需求实现一个不同的外观。每个 API 网关能提供为客户端裁剪过的有差异的 API，可能是基于客户端展现形式或设备来实现特定的适配代码，这些代码在底层调用多个内部的微服务。

由于一个自定义 API 网关通常是一个数据聚合器，因此需要小心处理。使用一个单独的聚合应用里所有微服务的 API 网关通常不是一个好办法，如果这么做，它就像是一个单体的聚合器或编排引擎，把所有微服务耦合在一起，违反了微服务自治的原则。因此，API 网关应该基于业务边界来拆分，而不是作为整个应用的一个聚合器。

有时候一个粒度化的 API 网关本身也能够成为一个微服务，甚至有自己的领域或业务名称以及相关的数据。根据业务或领域的描述来确定 API 网关的边界会对设计有帮助。

将 API 网关粒度化分层对基于微服务的、更高级的组合 UI 应用特别有好处，因为设计精良的 API 网关跟 UI 组合服务的概念很类似。我们在后续的“[基于微服务创建组合 UI](#)”一节会讨论。

因此，针对很多中型和大型应用，使用自定义的 API 网关通常是一个好的方案，但不是使用一个单独的单体式聚合器或唯一的中央 API 网关。

另一种方案是使用像 [Azure API 管理 \(API Management\)](#) 这样的产品，如图 4-14 所示。这种方案不仅满足 API 网关的需求，也提供了比如搜集 API 统计信息的功能。如果您在使用 API 管理的方案，API 网关只是整个 API 管理方案的一个组件。

使用 Azure API 管理的 API 网关架构

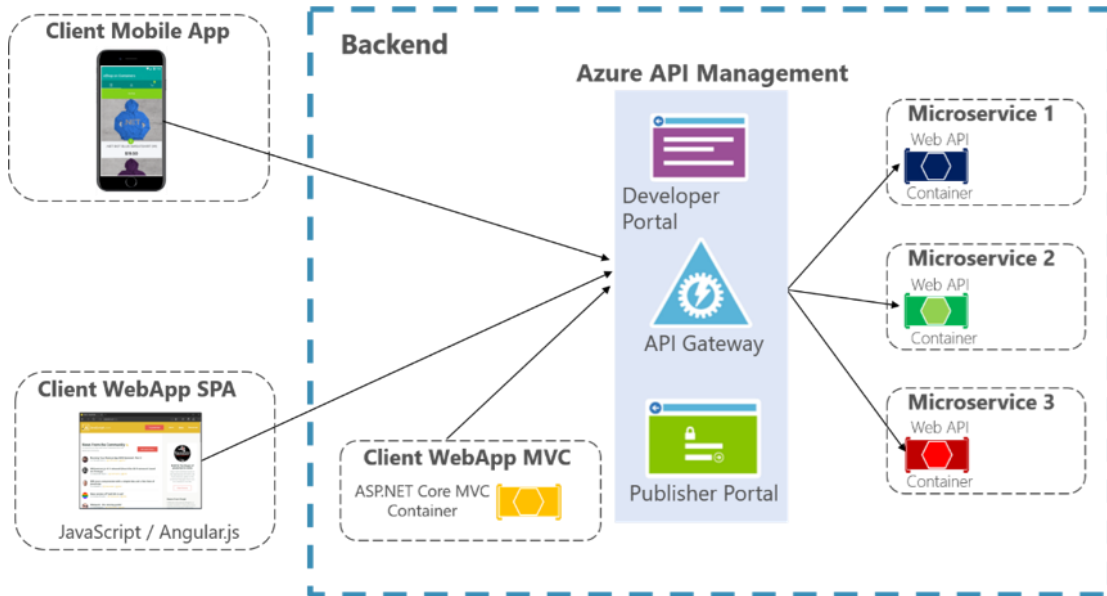


图4-14. 使用 Azure API 管理 (API Management) 实现 API 网关

本方案使用了像 Azure API 管理这样的产品，由于这样的 API 网关更“扁平”，所以用单一的 API 网关也不会有太大的风险，意味着您不必自己写很多朝着单体组件发展的 C# 代码了。

这类产品更像是入口通信的反向代理，在这单独的一层，您也可以过滤内部微服务的 API，或者在已发布的 API 上添加授权。

API 管理系统的统计信息帮助您了解 API 是如何被使用以及性能是怎样的，它允许您查看近乎实时的分析报告以便识别可能影响业务的趋势。另外也提供请求响应的日志供进一步的在线或离线的分析。

使用 Azure API 管理 (API Management)，您能使用密码、密钥和 IP 过滤器来实现安全性，这些功能可以加强稳定性，提供设计精良的限额和速率限制，使用策略来调整 API 的表现和行为，以及使用响应缓存来提升性能。

在本书以及示例应用 eShopOnContainers 中，我们把架构简化成自定义的容器化架构，以便聚焦在不使用像 Azure API 管理 (Azure API Management) 这种 PaaS 类产品的扁平容器上。但对于部署在微软 Azure 上的大型微服务应用，我们鼓励您适配 Azure API 管理 (API Management) 作为 API 网关的基础。

网关模式的不足之处

- 最大的不足是当您实现 API 网关时，相应层级的内部微服务就被耦合起来了。这样的耦合也许会为您的应用带来极大的困难。Azure 服务总线团队的架构师 Clemens Vaster 在 2016 年 GOTO 上的演讲“[消息和微服务](#)”中引用了这个潜在的问题并称作“新 ESB”）。
 - 使用 API 网关增加了额外的故障点。
 - API 网关因额外的网络调用增加了响应时间，但是这些额外的调用通常比不上客户端接口直接调用内部微服务的影响。
 - 如果没有适当扩展，API 网关可能会变成瓶颈。
 - 如果 API 网关包含自定义逻辑和数据集成，就会要求额外的开发成本和未来的维护成本。为了暴露每个微服务的接口，开发人员必须更新 API 网关。甚至内部微服务的实现逻辑变更也会导致 API 网关的代码变更。然而如果 API 网关只是用了安全性、日志管理和版本管理（当使用 Azure API 管理时），这些额外的开发工作不会发生。
- 如果 API 网关由一个团队开发，那么会存在开发瓶颈。这也是提供多个设计精良的、处理不同客户端要求的 API 网关才是一个更好的方式的原因。也可以在内部把 API 网关拆分成多个区域或层级并由不同的内部微服务团队来开发。

其他资源

- **Charles Richardson. 模式：API 网关/前端和后端**
<http://microservices.io/patterns/apigateway.html>
- **Azure API 管理 (API Management)**
<https://azure.microsoft.com/services/api-management/>
- **Udi Dahan. 面向服务的组合**
<http://udidahan.com/2014/07/30/service-oriented-composition-with-video/>
- **Clemens Vasters. GOTO 2016 上的“消息和微服务” (视频)**
<https://www.youtube.com/watch?v=rXi5CLjIQ9k>

微服务间的通信

一个运行在单个进程里的单体应用，它的组件调用使用代码级别的方法和函数。如果您使用代码创建对象（比如 `new ClassName()`）就会造成强耦合。可以使用依赖注入来引用抽象类而不是具体类的实例，能做到解耦的方式调用。不管哪种方式，对象都是在相同的进程里。将一个单体应用改成微服务应用的最大挑战就是改变通信机制，直接把进程内方法调用改成服务间的 RPC 调用会导致在分布式环境中性能低下的、零散的和低效的通信。如何设计合适的分布式系统这个挑战很出名，甚至有个准则被称为[分布式计算的谬误](#)列出了开发人员在迁移单体应用到分布式设计时碰到的问题。

解决方案并不是唯一的，而是有很多种。一种方案是尽可能地使业务微服务独立，之后在内部微服务间使用异步通信，并将通常在进程内的细粒度通信替换成粗粒度通信。可以通过把调用组合起来，然后给客户端返回聚合多个内部调用结果的数据来实现这个目的。

一个微服务应用是运行在多个进程或服务里的分布式系统，通常会跨多个服务器和主机。每个服务实例通常是一个进程。因此根据每个服务的本质，服务之间必须使用进程间的通信协议（比如 HTTP、AMQP）或者二进制协议（比如 TCP）。

微服务社区推崇“[端点智能管道静默](#)”的哲学，这个口号鼓励微服务之间尽可能解耦，一个独立的微服务内部尽可能地协作的设计。如前所述，每个微服务拥有自己的数据和领域逻辑，但是微服务在组成端到端的应用时通常简单地使用 REST 通信而不是复杂协议（比如 WS-*），并且使用稳定的事件驱动通信代替中心化的业务处理编排引擎。

两种常用的协议是用 HTTP 请求/响应来调用资源 API（大部分用在查询），以及轻量的异步消息用来跨多个微服务进行更新。更多细节在接下去的内容里介绍。

通信类型

客户端和服务的通信有很多不同的类型，每一种有不同的目的和场景，这些通信类型可以分为两个维度：

第一个维度是看协议是异步还是同步的：

- 同步协议。HTTP 是一种同步协议。客户端发起一个请求然后等待服务端响应。客户端的代码可以独立地实现同步（线程被阻塞）或异步（线程非阻塞，最终的响应通过回调来处理）的执行方式。这里的重点在于协议（HTTP / HTTPS）是同步的，客户端代码只能在收到 HTTP 服务端的响应后才可以继续先前的任务。
- 异步协议。其他协议比如 AMQP（很多操作系统和云环境支持的一种协议）使用了异步消息。客户端代码或消息发送者通常不需要等待响应，只要把消息发送给 RabbitMQ 队列或其他消息代理。

第二个维度是看有单个接收者还是多个接收者：

- 单个接受者。每个请求必须准确地被一个接收者或服务来处理，一个例子就是[命令模式](#)。
- 多个接收者。每个请求能被 0 个或多个接收者来处理，这类通信必须是异步的，一个例子是[事件驱动架构](#)里的[发布/订阅](#)机制。它基于事件总线接口或消息代理在多个微服务间通过事件来传送数据。通常以使用[主题和订阅](#)的服务总线或类似的产品比如 Azure 服务总线 (Azure Service Bus) 来实现。

一个微服务应用会常常组合使用这些通信风格。当调用一个常规的 Web API HTTP 服务时，最通用的类型是使用单个接受者的同步协议比如 HTTP/HTTPS。微服务也常常使用消息协议来实现微服务间的异步通信。

明白这些维度对于清晰认识可能的通信机制有好处，但是它们不是创建微服务的重点。在集成微服务时的重点在于客户端线程执行的异步特性并不是所选协议的异步特性。重要的是维持微服务独立性的同时能够异步地集成它们，接下来一节会详细解释。

异步整合方式增强微服务自治

如前所述，创建微服务应用的重点在于整合微服务的方式。理想情况下，应该减少内部微服务间的通信，微服务之间的交互越少越好，当然很多场景下不得不进行一些交互。当您着手做的时候，记住核心规则是微服务间的交互需要异步。并不意味着一定要使用某种特定的协议（比如异步消息或同步的 HTTP），只是表明微服务间通过异步传输数据来通信，但请不要依赖于其他内部微服务作为自己 HTTP 请求/响应的一部分。

如果可能，即便只是用于查询，也绝不要依赖微服务间的同步通信（请求/响应）。每个微服务的目标是自治的，对客户端是可用的，即使作为端到端应用一部分的其他服务发生故障或不稳定。如果您认为需要从一个微服务调用其他微服务（比如发起一个 HTTP 请求来查询数据）为客户端应用提供响应结果，那么这样的架构在其他微服务发生故障时就变得不稳定。

此外，在微服务之间如果存在 HTTP 依赖，比如串联 HTTP 请求创建很长的请求/响应周期，如图 4-15 的第一部分所示，这样不仅使您的微服务不能自治，而且一旦这个链条上的某个服务有性能问题，整个服务的性能都受到影响。

微服务间添加的同步依赖（比如查询请求）越多，客户端应用的总响应时间就会越长。

微服务之间的同步和异步通信

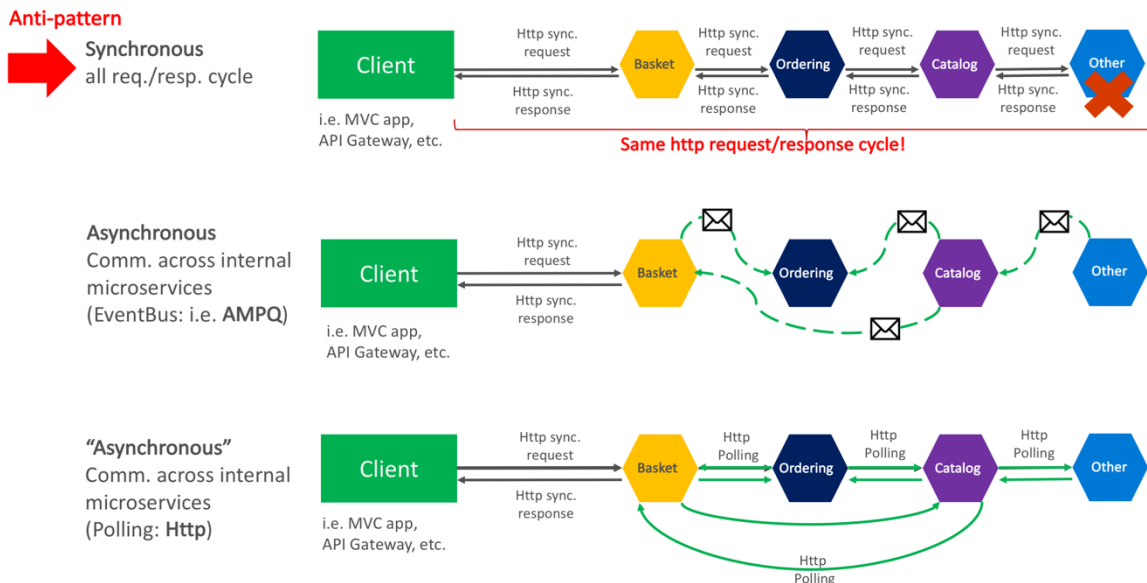


图 4-15. 微服务通信的反模式和模式

如果您的微服务确实需要在其他微服务里引发额外的操作，如果可能，不要用同步方式去执行，也不要作为这个微服务的请求响应操作的一部分，而是用异步方式去处理（使用异步消息或集成事件，队列等）。但是，请尽最大的可能不要作为最初同步请求响应操作的一部分去同步地调用这些额外的操作。

最后（这也是创建微服务时碰到最多的问题），如果最初的微服务需要原本在别的微服务里拥有的数据，不要依靠同步请求来获取数据。而是通过最终一致性（通常通过集成事件，接下去会介绍）方式来复制或传输这些数据到最初的微服务的数据库中。

在前文“[识别微服务的领域模型边界](#)”一节里提到过，跨多个微服务复制数据不是一个良好设计，相反，要做的话可以将数据转换成领域或限界上下文特定的语言或术语。比如在 [eShopOnContainers](#) 应用中有一个叫 `identity.api` 的微服务包含一个叫 `User` 的实体来管理大部分的用户数据。然而，当您需要在订单微服务里保存用户数据的时候，就以名为 `Buyer` 的不同实体来存储。`Buyer` 实体跟 `User` 实体共用相同的标识，但订单领域只需要很少的属性而不是整个用户资料。

为了获得最终一致性，可以使用任何协议在微服务之间异步地通信来获取数据。如前所述，可以使用事件总线或消息代理的集成事件，甚至使用 HTTP 轮询都是可行的。重要的规则是不要在微服务间创建同步依赖。

下一节解释了多种可以考虑用在微服务应用里的通信风格。

通信风格

根据所选择的通信类型，有多种协议和选择能够用来通信。如果使用同步的请求/响应通信机制，最常用的协议是 HTTP 和 REST，尤其是把服务发布在 Docker 主机或微服务集群之外。如果在内部通信（在 Docker 主机或微服务内部）也可以使用二进制通信机制（比如远程 Service Fabric 或使用 TCP 和二进制格式的 WCF）。另外也可以使用异步的、基于消息的通信机制（比如 AMQP）。

可以选择多种消息格式比如 JSON 或 XML，甚至是更有效率的二进制格式。如果选择了非标准的二进制格式，要把服务发布出去就不是一个好方法。可以为内部的微服务选择非标准格式，在 Docker 主机或微服务集群（Docker 编排引擎或 Azure Service Fabric）内的微服务之间通信的时候，或者专用的客户端应用和微服务通信的时候，可以使用这种方式。

使用 HTTP 和 REST 的请求/响应通信

当一个客户端使用请求/响应通信时，它发起一个请求到服务端，然后服务端处理请求并返回一个响应内容。请求/响应通信尤其适合客户端应用的实时数据查询 UI（一个实时的界面）。因此，在一个微服务架构中您很可能使用这种通信机制来处理大部分查询。如图 4-16 所示。

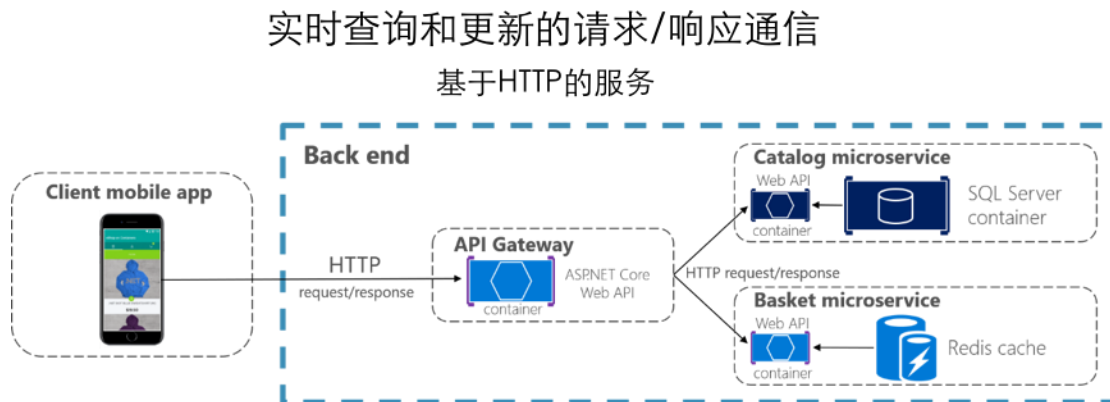


图4-16. 使用 HTTP 请求/响应通信（同步或异步）

当一个客户端使用请求/响应通信，有个假设是响应内容会在很短的时间内返回，通常在 1 秒之内或最多几秒。面对延迟的响应，需要基于消息模式和消息技术来实现异步通信，这是一种不同的方式，会在下一节中讲解。

一种流行的请求/响应通信架构风格是 REST，这种方式是基于、也是紧紧耦合于 HTTP 协议的，它采用 HTTP 谓词（比如 GET、POST 和 PUT）。创建服务时，REST 是最常用的架构通信方式，可以通过开发 ASP.NET Core Web API 服务来实现 REST 服务。

额外的优势是使用 HTTP REST 服务作为接口定义语言。比如使用 Swagger 元数据来描述服务 API，能使用工具来生成客户端直接发现和使用服务。

其他资源

- **Martin Fowler. Richardson 成熟度模型。REST 模型的一种描述**
<http://martinfowler.com/articles/richardsonMaturityModel.html>
- **Swagger. 官方网站**
<http://swagger.io/>

基于 HTTP 的推送和实时通信

另一种选择（通常用于跟 REST 不同的目的）是使用高级框架和协议（比如 [ASP.NET SignalR](#) 和 [WebSockets](#)）的一种实时和一对多的通信机制。

如图 4-17 所示，实时 HTTP 通信意味当数据可用时服务端代码会推送内容到已连接的客户端，而不是服务端等待客户端来请求新数据。

基于HTTP的推送和实时通信 一对多通信

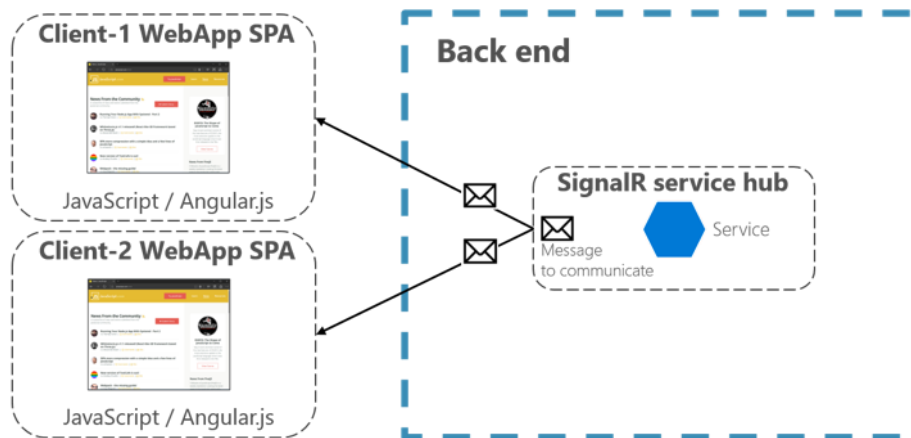


图 4-17. 一对一的实时异步消息通信

因为通信是实时的，客户端应用几乎能立即显示变化，这通常是通过像 WebSockets 这样的协议来处理，并且使用多个 WebSockets 连接（每个客户端一个）。一个经典的例子是服务端同时传输体育运动的得分到多个客户端网页应用。

异步消息通信

当跨越多个微服务及其相关领域模型传送变化时，使用异步消息和事件驱动的通信至关重要。如前所述，模型（用户、客户、产品、帐户等）在不同的微服务或 BC 里可能有不同的含义。这意味着当发生变化时，需要一些方法来协调不同模型中的变化。一种解决方案是基于异步消息传递和事件驱动的最终一致性。

当使用消息时，进程间通过异步交换消息进行通信。客户端通过向服务器发送消息来下达命令或发起请求。如果服务端需要回复，则会向客户端发送另一个不同的消息。由于它是一种基于消息的通信，客户端会认为该回复将不会立即收到，并且可能根本就不会有响应。

消息由标题（诸如标识或安全信息的元数据）和主体组成。消息通常通过异步协议（如 AMQP）来发送。

在微服务社区中，这种类型的通信的首选底层架构是轻量级消息代理，它与 SOA 中使用的大型代理和编排引擎不同。在轻量级消息代理中，底层架构通常是“愚蠢的”，它仅作为消息代理，比如简单的实现有 RabbitMQ，云端的可扩展服务总线实现比如 Azure 服务总线（Service Bus）。在这种场景下，大多数“智能”思维仍然存在于生产和使用消息的端点 - 即微服务中。

需要尽可能遵循的另一个规则是只在内部服务之间使用异步消息传递，只在从客户端应用到前端服务（API 网关加上第一级微服务）使用同步通信（比如 HTTP）。

有两种异步消息通信：单接收者消息通信，多接收者消息通信。在下面的部分我们将提供有关它们的详情。

单接收者消息通信

与单个接收者进行基于消息的异步通信意味着点对点的通信，将消息分发到正在通过信道读取的其中一个消费者，并且该消息仅被处理一次。但是也有特殊情况。例如，在云系统中，在尝试自动从故障中恢复时，一个相同的消息可以发送多次。由于网络或其他故障，客户端必须能够重试发送消息，并且服务器必须实现一个幂等的操作以便特定的消息仅处理一次。

单接收者消息通信特别适用于将异步的命令从一个微服务发送到另一个微服务，如图 4-18 展示了这种方法。

一旦开始发送基于消息的通信（使用命令或事件），您应该避免将基于消息的通信与同步 HTTP 混合。

单接收者消息通信

比如：基于消息的命令

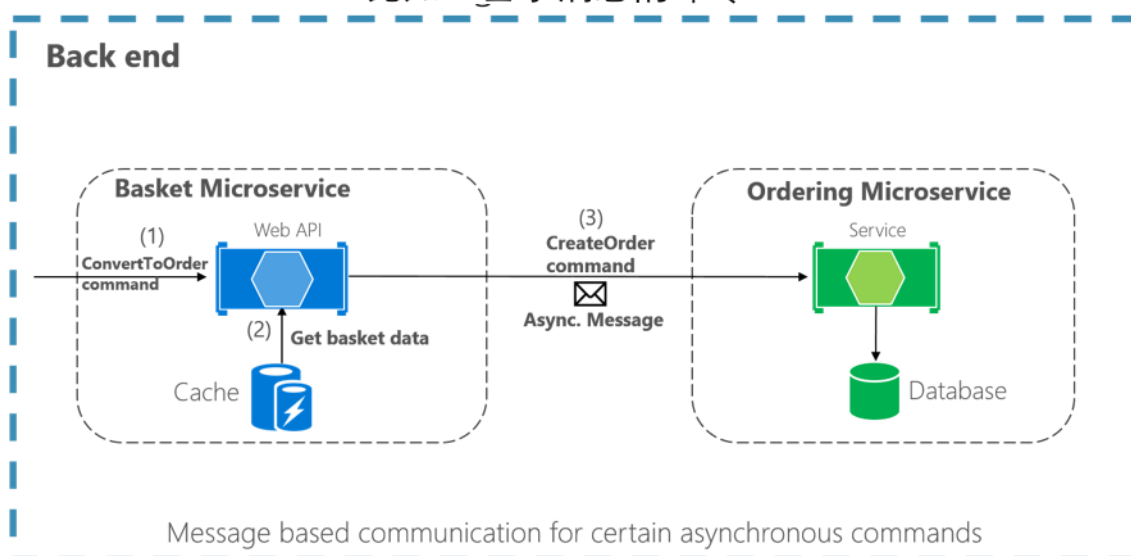


图4-18. 一个接收异步消息的微服务

请注意，当命令来自客户端应用程序时，可以实现为 HTTP 同步命令。当您需要更高级别的可扩展性或者已经在使用基于消息的业务流程时，就应该使用基于消息的命令。

多接受者消息通信

作为一种更灵活的方法，您可能还需要使用发布/订阅机制，以便发布者的通信可用于其他订阅者微服务或外部应用。因此，它可以帮助您在发送服务中遵循[开闭原则](#)。这样在将来可以添加其他订阅者，而无需修改发送服务。

当使用发布/订阅通信时，您可以使用事件总线接口将事件发布到任何一个订阅者。

异步事件驱动通信

当使用异步事件驱动通信时，一个微服务在其领域内某事件发生时发布一条集成事件，另一个微服务则需要关注它，比如目录微服务中的价格变化，其他微服务订阅这个事件，以便可以异步接收它们。当这种情况发生时，接收者可能需要更新自己的领域实体，这可能导致更多的集成事件被发布。这样的发布/订阅系统通常使用事件总线来实现。事件总线可以设计成抽象的或接口的，使用 API 来实现需要订阅或取消的事件和发布事件。事件总线可以有基于任何进程间和消息代理的一个或多个实现，例如支持异步通信的消息队列或服务总线，以及发布/订阅模式。

如果系统使用由集成事件驱动的最终一致性，则建议这种方法对最终用户是完全透明的。系统不应该使用模仿集成事件的方法，如来自客户端的 SignalR 或轮询系统。最终用户和业务方必须明确地接受系统的最终一致性，并意识到在许多情况下，只要它是明确的，这种方法对于业务不会有任何问题。

如前文在“[分布式数据管理的挑战和解决方案](#)”一节所述，您可以使用集成事件来实现跨多个微服务器的业务任务。因此，在这些服务之间将具有最终一致性。最终一致的事务由分布式的操作集合而成。在每个操作中，相关的微服务更新一个领域实体，并发布另一个集成事件，该事件在相同的端到端业务任务中引发下一个操作。

一个重要的地方是您可能要跟多个订阅了相同事件的微服务通信，要实现这样的目标，可以使用基于发布/订阅的消息事件驱动通信，如图 4-19 所示。发布/订阅机制不是微服务架构独有的，它跟 DDD 里的限界上下文的通信和 CQRS 架构模式的从可写数据库到只读数据库产生更新的方式相似。目的是在分布式系统中跨多个数据源达到最终一致性。

异步事件驱动通信 多接收者

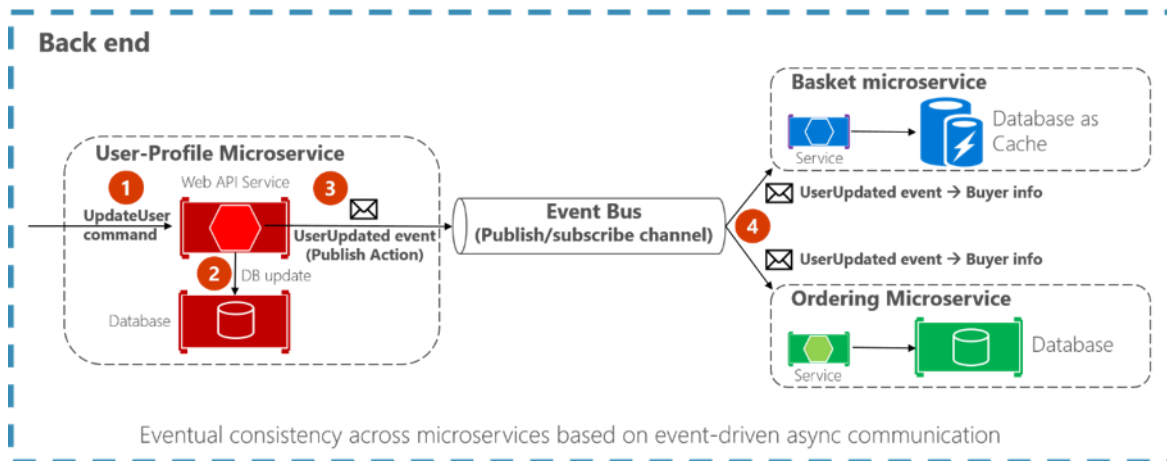


图 4-19. 异步事件驱动消息通信

在实现时要决定为事件驱动、基于消息的通信采用什么协议。[AMQP](#) 能帮助实现可靠的队列通信。

当使用事件总线时，可能需要使用一个抽象层级（比如事件总线接口），它使用相关的类代码来实现，并使用消息代理（比如 RabbitMQ 或服务总线比如 Azure 主题服务总线（Service Bus with Topics））的 API。另外，可能需要使用高层级的服务总线（比如 NServiceBus、MassTransit 或 Brighter）来作为事件总线和发布订阅系统的表示层。

关于生产系统的消息技术

用来实现抽象事件总线的消息技术有不同的层级。比如像 RabbitMQ（一个消息传送代理）和 Azure 服务总线（Service Bus）这些产品属于较低的层级，其他产品（比如 NServiceBus、MassTransit 或 Brighter）能在 RabbitMQ 和 Azure 服务总线（Service Bus）之上运行。可以根据应用需要多少丰富的功能和多少开箱即用的扩展性来作出选择。如果只是在开发环境实现一个验证理论的事件总线，就像我们在 eShopOnContainers 示例中的一样，一个简单的运行在 Docker 容器里的、在 RabbitMQ 之上的实现就足够了。

但对于需要高扩展性的关键生产系统，可以考虑 Azure 服务总线（Service Bus）。如果想要使用高级抽象和功能让开发分布式应用变得更容易，我们推荐考虑其他商业或开源的服务总线比如 NServiceBus, MassTransit 和 Brighter。当然您也能在低层级技术（比如 RabbitMQ 和 Docker）之上创建自己的服务总线，但这些繁重的工作对于企业应用是很大的开销。

弹性发布到事件总线

跨多个微服务实现事件驱动的架构有一个挑战，就是当弹性发布集成事件到事件总线上时如何自动的更新原始微服务的状态，尤其是某些基于事务的方式。以下是一些实现方式，但也有其他的方法。

- 使用（基于 DTC 的）事务队列比如 MSMQ（但这是一个遗留的方法）。
- 使用[事务日志挖掘](#)。
- 使用完整的[事件溯源](#)模式。
- 使用[远见模式](#)：创建和发布事件的组件使用事务数据库的表作为消息队列存储。

使用异步通信要考虑的其他主题有消息的幂等和重复。这些主题在本书“[在微服务间实现事件通信（集成事件）](#)”一节详细介绍。

其他资源

- **事件驱动的消息**
http://soapatterns.org/design_patterns/event_driven_messaging
- **发布/订阅渠道**
<http://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html>
- **Udi Dahan. 认清 CQRS**
<http://udidahan.com/2009/12/09/clarified-cqrs/>
- **命令查询职责分离 (CQRS)**
<https://docs.microsoft.com/azure/architecture/patterns/cqrs>
- **限界上下文间的通信**
<https://msdn.microsoft.com/library/jj591572.aspx>
- **最终一致性**
https://en.wikipedia.org/wiki/Eventual_consistency
- **Jimmy Bogard. 向弹性重构：评估耦合**
<https://jimmybogard.com/refactoring-towards-resilience-evaluating-coupling/>

创建、改进和控制微服务 API 的版本和契约

微服务的 API 是服务端和客户端之间的一份契约。您只能在不破坏 API 契约的前提下独立地改进微服务，所以契约很重要。如果改变了契约，就会影响到客户端应用或 API 网关。

API 的定义依赖于使用的协议。比如使用（像 [AMQP](#) 这类的协议）消息时，API 由消息类型组成。如果使用 HTTP 和 REST 风格的服务，API 由 URL 和请求以及 JSON 格式的响应组成。

然而，即便初始版本的契约已经考虑周全了，随着时间的发展，一个服务的 API 也会需要改变。如果发生了变化，尤其是被多个客户端应用调用的公共 API，通常无法强制所有客户端升级到新的 API 契约。通常需要增量部署服务的新版本，还要让老版本和新版本的服务契约同时运行。因此，服务的版本策略是很重要的。

如果 API 的改动比较小，比如添加了属性或参数，使用旧版本 API 的客户端应该能切换过来，和新版本的服务协同工作。您可以为必需的属性提供默认值，客户端能够忽略任何额外的属性。

但是，有时候需要对服务 API 进行不兼容的大版本更新。因为不能强制客户端应用或服务立刻升级到新版本，服务端必须支持老版本继续运行一段时间。如果使用的是基于 HTTP 的机制（比如 REST），一种方式是把 API 的版本号嵌入到 URL 中或 HTTP 的头部。然后您可以决定是在一个服务里同时实现两个版本的 API，还是部署不同的服务来各自处理一个版本的 API。一个好的方法是采用 [中介者模式](#)（比如 [MediatR 库](#)）来将不同版本的实现用不同的处理器来处理。

最后，如果您使用 REST 架构，[Hypermedia](#) 是用来进行服务版本化和改进的最佳选择。

其他资源

- **Scott Hanselman. 轻松实现 ASP.NET Core RESTful Web API 版本管理**
<http://www.hanselman.com/blog/ASPNETCoreRESTfulWebAPIVersioningMadeEasy.aspx>
- **RESTful Web API 版本管理**
<https://docs.microsoft.com/azure/architecture/best-practices/api-design#versioning-a-restful-web-api>
- **Roy Fielding. 版本、超媒体和 REST**
<https://www.infoq.com/articles/roy-fielding-on-versioning>

微服务的可发现性和服务注册

每个微服务有一个唯一名称（URL）表示它的地址。无论微服务运行在哪里，都必需是可发现的。如果您不得不回忆哪台电脑运行的哪个微服务，事情会很快变得糟糕。跟 DNS 解析一个 URL 到特定的电脑一样，微服务也需要有一个唯一名称来让它的地址是可发现的。微服务需要一个可发现的名称，把它们从运行的基础设施中区分出来，这表明在服务如何部署和服务如何发现之间存在交互，因为需要一个服务注册表。同样的，当一个节点发生了故障，注册表必须能够指出该服务现在在哪里运行。

[服务注册模式](#)是服务发现的关键部分。注册表是一个包含服务实例网络地址的数据库。一个服务注册表需要高度可用并且保持最新。客户端可以将将从服务注册表获取的网络地址缓存下来。但这些信息最终会过时，客户端就找不到服务实例了。因此，服务注册表由采用复制协议来维护可用性的服务器集群组成。

在一些微服务部署环境（称为集群，后续会介绍）中，服务发现是内置的。比如在 Azure 容器服务 (Container Service) 环境里，使用 Marathon 的 Kubernetes 和 DC/OS 能够处理服务实例的注册和撤销。也能够每个集群运行一个代理，用来作为服务端发现的路由。另一个例子是 Azure Service Fabric，它也提供了开箱即用的命名服务来实现服务注册。

注意服务注册和 API 网关模式有些重叠的功能，也可以帮助解决这个问题。比如 [Service Fabric 反向代理](#)实现了基于 Service Fabric 命名服务的 API 网关，能用来解决内部服务的地址查找问题。

其他资源

- **Chris Richardson. 模式：服务注册**
<http://microservices.io/patterns/service-registry.html>
- **Auth0. 服务注册**
<https://auth0.com/blog/an-introduction-to-microservices-part-3-the-service-registry/>
- **Gabriel Schenker. 服务发现**
<https://lostechies.com/gabrielschenker/2016/01/27/service-discovery/>

创建基于多个微服务组合界面，其中包括由微服务生成的、可视化的 UI 外观和布局

微服务架构常常从服务端的数据处理和逻辑开始。但一种更先进的方法是设计基于微服务的应用界面，这意味着您拥有一个由多个微服务生成的组合界面，而不是由一个单体客户端应用来调用服务器上的微服务。采用这种方式的微服务能够完整的包括逻辑和可视化展现。

图 4-20 简单展示了一个单体式客户端应用调用后端微服务的场景。当然您应该用一个 ASP.NET MVC 的服务来生成 HTML 和 JavaScript。该图经过了简化是为了突出一个单一（单体式）的客户端界面来调用多个微服务，这些微服务仅仅关注逻辑和数据，而不关注界面展现（HTML 和 JavaScript）。

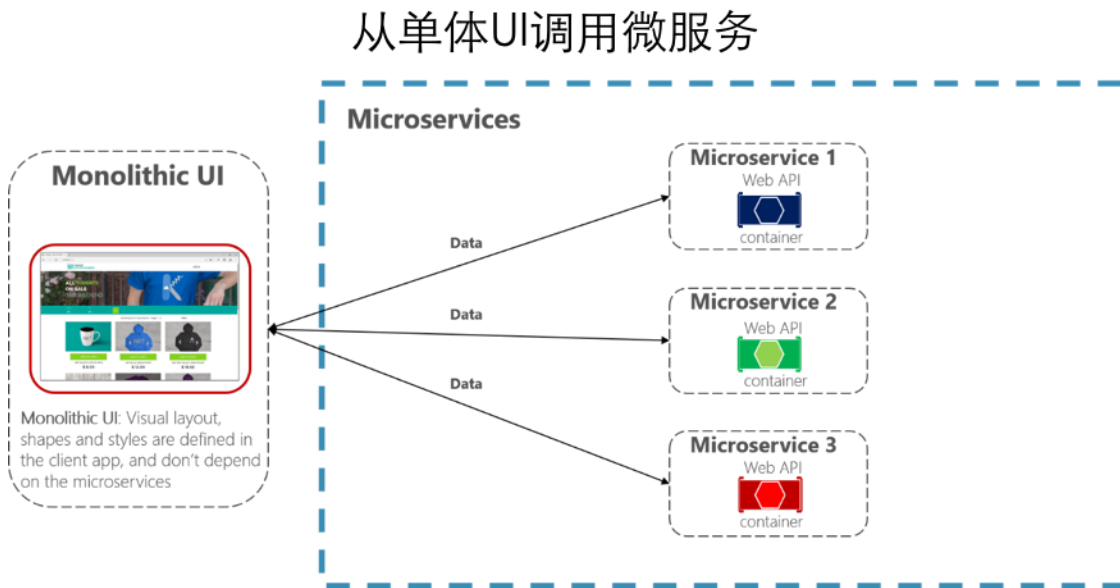


图 4-20. 一个调用多个后台微服务的单体 UI 应用

作为对比，一个组合式界面是由微服务自己生成和组合的。某些微服务控制了界面特定区域的可视化展现。关键的区别在于使用了基于模板的客户端界面组件（比如 TypeScript 类），以及那些来自各个微服务的模板的数据可视界面的 ViewModel。

在客户端应用启动时，每个 UI 组件（比如 TypeScript 类）将自己注册到能为给定场景提供 ViewModel 的底层微服务上。如果这个微服务更新了展现，界面也会发生变化。

图 4-21 展示了一种这样的组合式界面。这也是简化过的，因为您或许还有其他微服务来聚合不同技术的细粒度模块 - 这取决于您是在创建一个传统的 Web 应用（ASP.NET MVC）还是一个 SPA（单页面应用）。

由微服务生成的组合UI

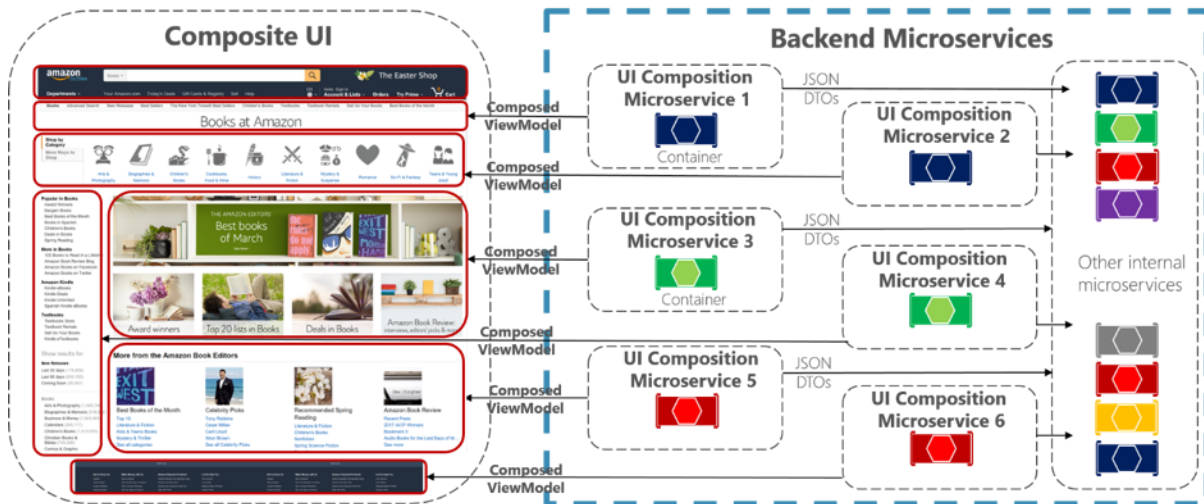


图 4-21. 由后端微服务生成的组合式界面应用示例

每一个界面组合的微服务跟一个小型 API 网关类似，但在这里它只负责一小区域的界面。

根据界面所采用的技术，由微服务生成组合式界面的方式或多或少会带来挑战。举例来说，您将不能使用跟传统 Web 应用相同的技术来创建一个 SPA 或原生的移动应用（尤其是开发 Xamarin 应用时，这种方式会更有难度）。

由于多种原因，示例应用 [eShopOnContainers](#) 采用了单体式界面。首先，它是用来介绍微服务和容器的，组合式界面更先进，但在设计和开发界面时更复杂。其次，eShopOnContainers 提供了基于 Xamarin 的原生移动应用，会使客户端的 C# 代码变得更复杂。

然而，我们鼓励您参考下列资源来了解更多的基于微服务的组合式界面。

其他资源

- **使用 ASP.NET 的组合 UI (Particular 的一个 Workshop)**
<https://github.com/Particular/Workshop.Microservices/tree/master/demos/CompositeUI-MVC>
- **Ruben Oostinga. 微服务架构下的单体前端应用**
<http://blog.xebia.com/the-monolithic-frontend-in-the-microservices-architecture/>
- **Mauro Servienti. 更好组合 UI 的秘诀**
<https://particular.net/blog/secret-of-better-ui-composition>
- **Viktor Farcic. 将前端 Web 组件包含在微服务中**
<https://technologyconversations.com/2015/08/09/including-front-end-web-components-into-microservices/>
- **管理微服务架构中的前端应用**
<http://allegro.tech/2016/03/Managing-Frontend-in-the-microservices-architecture.html>

微服务的适应性和高可用性

处理意外故障是一个很大的难题，尤其是在一个分布式系统中。开发人员的大部分代码包含了异常处理，这也是需要大量时间测试的地方。问题在于处理故障比写代码更困难。微服务发生故障时的服务器到底发生了什么？您不仅仅需要检测到微服务的故障（它本身的问题），也需要一些条件来重启微服务。

一个微服务需要能适应故障，并且能经常在其他机器上重启来获得可用性，这种适应性也对底层保存的微服务状态适用，状态可用于恢复微服务，无论微服务是否成功重启。换句话说，计算能力（进程随时能够重启）和状态数据（没有数据丢失，并且数据要保持一致）都需要适应性。

适应性的问题在一些场景下会加剧，比如在应用升级时发生故障。部署系统的微服务需要决定能否继续升级新版本，或者为了维护一致的状态数据而回滚到以前的版本。类似是否有足够的机器来升级和如何恢复以前的版本这样的问题是需要考虑的。这要求微服务来发布运行状况信息以便总体的应用和编排引擎能够作出决定。

另外，适应性关系到云系统必要的功能。如前所述，一个基于云端的系统必须拥抱故障和尝试自动从故障中恢复。比如，假设网络和容器发生了故障，客户端应用或服务必须有一个策略来重新发送消息或重新发起请求，因为大多数情况下云端的故障都是局部的。在本书的“[实现弹性应用](#)”一章指出了如何处理局部故障，描述了一些技术手段（比如采用越来越快的重试，或者通过在 .NET Core 里使用像 Polly 库的断路器模式），它们都提供了处理这个问题的多种策略。

微服务的运行状况管理和诊断

看起来很明显但常常被忽略的是，一个微服务必须报告它的运行状况和诊断信息，否则，从运营角度上获取的信息会很少。关联跨多个独立的服务的诊断事件和处理机器时钟偏差让事件顺序有意义都是挑战。跟微服务之间的交互需要统一的协议和数据格式一样，也需要一个标准来规定如何记录运行状况和诊断事件，最终保存在事件存储器中供查询查看。在微服务的方式下，关键在于要不同的团队采用统一的日志格式。应用中也需要一个一致的方式来查看诊断事件。

运行状况检查

运行状况跟诊断不同，运行状况是微服务报告当前状态以便执行合适的操作，一个好的例子是配合升级和部署机制来维护可用性，虽然一个服务由于进程崩溃或机器重启而不稳定，但是服务可能仍然是能用的。最后是在升级的时候情况变得更糟。最好的办法是先进行一轮调查，或者给时间让微服务恢复。微服务的运行状况事件帮助我们作出有依据的决定，并且有效帮助我们创建能自愈的服务。

在本书的“[在 ASP.NET Core 服务里实现运行状况检查](#)”一节中，我们解释了如何在微服务中使用新的 ASP.NET HealthChecks 库，以便微服务能向一个监控服务报告状态来执行合适的操作。

使用诊断和日志事件流

日志提供了应用或服务如何运行的信息，包括异常、警告和简单的信息。通常每个日志就是一个事件一行的文本，即使异常提供了多行的栈跟踪信息。

在单体式服务端应用中，您可以简单的将日志写入到磁盘上的文件（一个日志文件）里，然后用任何工具进行分析。由于应用是限定运行在一个固定的服务器或虚拟机上，通常分析事件流不会太复杂。然而，在一个编排引擎或集群里的多个节点上运行多个服务的分布式应用中，要把分布的事件关联起来就成了一个挑战。

一个微服务应用不应该尝试自己存储事件输出流或日志，也不要管理中央存储的事件路由。它应该是透明的，意味着每个进程应该只是将事件流写入到标准输出，所运行的基础执行环境的底层会收集它们。这种事件流转发器的一个例子是 [Microsoft.Diagnostic.EventFlow](#)，它从多个源收集事件流然后发布到输出系统，包括开发环境用到的简单标准输出，或者云上的系统比如 [Application Insights](#), [OMS](#)（本地部署使用）和 [Azure 诊断](#)。也有很好的第三方日志分析平台和工具提供搜索、报警、报表和监控，甚至是实时的，比如 [Splunk](#)。

编排引擎管理运行状况和诊断信息

创建微服务应用时需要处理复杂性，当然，单独的一个微服务容易处理，但是有上百种类型和上千个实例时就是复杂的问题了。如果需要一个稳定的协作系统，不仅仅需要创建微服务架构，也需要高可用性、可寻址能力、适应性、运行状况和诊断等等。



图 4-22. 微服务平台是一个应用的运行状况管理的基础

要自己解决图 4-21 所示的复杂问题非常困难。开发团队应该去解决业务问题和使用微服务方式建立自己的应用程序，而不应该去关注复杂的基础架构问题，如果这么做，任何微服务应用的成本就会很庞大。因此有很多包括编排引擎或微服务集群的面向微服务的平台，它们尝试解决创建和运行服务的问题以及如何有效地使用基础架构的资源。

不同的编排引擎也许看起来相似，但是提供的诊断和运行状况检测等在功能和成熟度方面有差别，有时候这依赖于操作系统，这一点在下一章中将作介绍。

其他资源

- **The Twelve-Factor App. XI. Logs: 将日志作为事件流**
<https://12factor.net/logs>
- **Microsoft Diagnostic EventFlow 库的 GitHub 地址**
<https://github.com/Azure/diagnostics-eventflow>
- **什么是 Azure 诊断**
<https://docs.microsoft.com/azure/azure-diagnostics>
- **将 Windows 计算机连接到 Azure 中的 Log Analytics 服务**
<https://docs.microsoft.com/azure/log-analytics/log-analytics-windows-agents>
- **记录有意义的日志: 使用语义日志应用模块**
[https://msdn.microsoft.com/library/dn440729\(v=pandp.60\).aspx](https://msdn.microsoft.com/library/dn440729(v=pandp.60).aspx)
- **Splunk. 官方网站**
<http://www.splunk.com>
- **EventSource 类. Windows 下的事件跟踪 API (ETW)**
[https://msdn.microsoft.com/library/system.diagnostics.tracing.eventsource\(v=vs.110\).aspx](https://msdn.microsoft.com/library/system.diagnostics.tracing.eventsource(v=vs.110).aspx)

编排高扩展和高可用的多容器微服务

如果您的应用是基于微服务的，或简单的拆分成了多个容器，为准备投产的应用使用编排引擎是非常必要的。如前所述，在微服务的方案中，每个微服务拥有自己的模型和数据，以便在开发和部署上做到自治。但如果是由多个服务（比如 SOA）组成的传统应用，也需要用多个容器或服务来构成一个单体的业务应用，并作为分布式系统来部署。这类系统难以扩展和管理。因此，如果您想要一个生产环境就绪和可扩展的多容器应用，绝对需要编排引擎。

图 4-23 展示了在集群上部署多个微服务（容器）组成的应用。

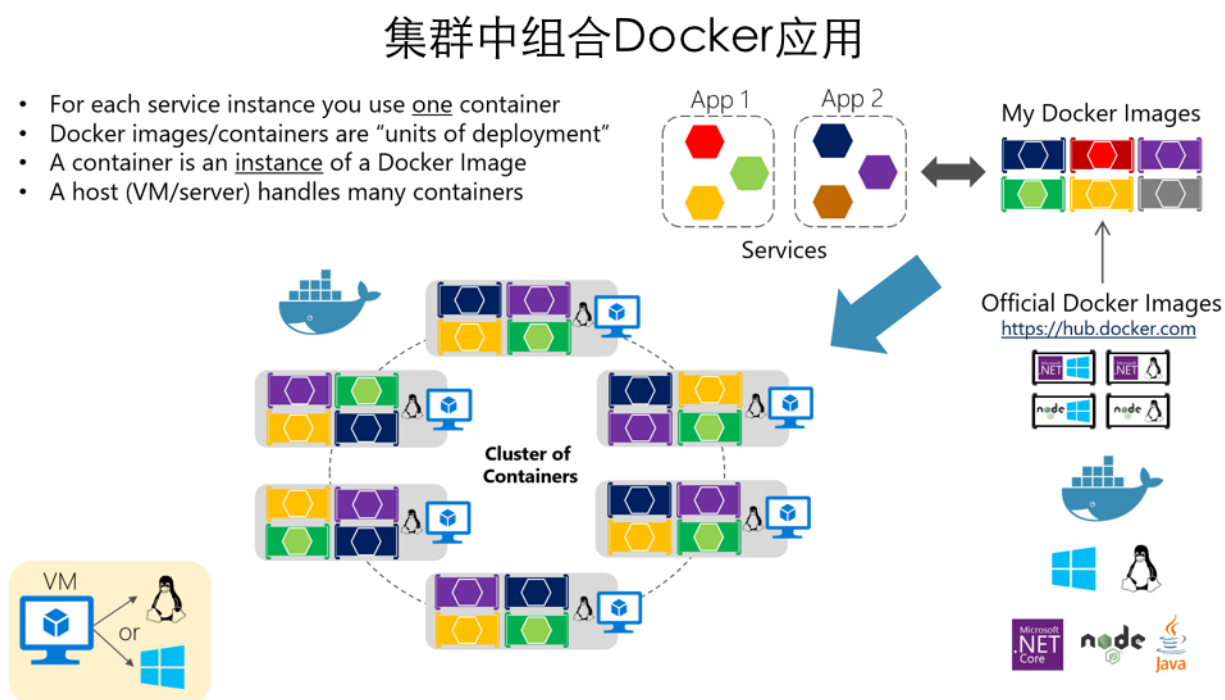


图 4-23. 一个容器集群

它看起来像一个逻辑上的设计，但要如何处理负载均衡、路由和编排这些组合应用？

Docker CLI 满足在一个主机上管理一个容器的需求，但用来管理部署在多个主机的多个容器的复杂分布式应用时就显得力不从心。大部分情况下，需要一个根据需要来自动地启动、挂起或关闭容器的管理平台，理想情况下也要控制资源的访问（比如网络和数据存储）。

要从管理独立的容器或简单的组合应用切换到大型的企业级微服务应用，必须切换到编排和集群平台。从架构和开发的角度来看，如果您在创建大型企业组合式微服务应用，很重要的一点是要理解下列支持高级场景的平台和产品：

集群和编排引擎：当您在多个 Docker 主机上横向扩展应用时，尤其是大型的微服务应用，能够通过抽象底层平台的复杂性，像一个独立的集群一样管理所有主机是很关键的。这是容器集群和编排引擎

能提供的。编排引擎产品有 Azure Service Fabric、Kubernetes、Docker Swarm 和 Mesosphere DC/OS，后面三个开源编排引擎通过 Azure 容器服务在 Azure 上提供。

调度器：调度表示能够让管理员启动集群的容器，同时需要提供一个界面。集群调度器有很多职责：有效地使用集群资源、设置用户提供的约束条件、跨节点或主机有效地均衡容器的负载以及很强的容错以便提供高可用性。

集群和调度器的在概念上紧密相关，所以不同供应商的产品常常提供了这两类功能。下表展示了可为集群和调度器选择的最重要的平台和软件，这些编排引擎通常已经在公有云（比如 Azure）上提供了。

容器集群、编排和调度软件平台	
<p>Kubernetes</p> 	<p>Kubernetes 是一个开源软件，提供的功能涵盖了集群基础架构到容器调度编排。能够跨集群和主机自动化地部署、扩展和运营应用容器。</p> <p>Kubernetes 提供以容器为中心的基础架构，将应用容器组织成逻辑单元来方便管理和发现。</p> <p>Linux 上的 Kubernetes 比 Windows 上的更成熟。</p>
<p>Docker Swarm</p> 	<p>Docker Swarm 允许集群化和调度 Docker 容器。通过使用 Swarm 能够把一群 Docker 主机转变成一个独立的虚拟 Docker 主机。客户端能像跟主机发起请求一样向 Swarm 发起 API 请求，这意味着 Swarm 让应用在多个主机间扩展很容易。</p> <p>Docker Swarm 是 Docker 公司的一个产品。</p> <p>Docker v1.12 或更新的版本能运行在原生和内置的 Swarm 模式下。</p>
<p>Mesosphere DC/OS</p> 	<p>Mesosphere 企业级 DC/OS（基于 Apache Mesos）是一个可用于生产环境的产品，它可以用于运行容器的分布式应用。</p> <p>Linux 上的 DC/OS 比 Windows 上的更成熟。</p> <p>DC/OS 将集群中可用的资源抽象为一个集合，并提供给上层组件。通常把 Marathon 作为调度器跟 DC/OS 配合使用。</p>
<p>Azure Service Fabric</p> 	<p>Service Fabric 是一个微软提供的创建应用程序的微服务平台。它是一个创建主机集群的服务编排引擎。Service Fabric 能以容器或普通进程的方式部署服务，甚至能在同一个应用和集群中混合进程中的服务和容器中的服务。</p>

	<p>Service Fabric 提供附加的和可选的编程模型规范，比如状态服务和 Reliable Actors。</p> <p>Service Fabric 在 Windows 上（经过了多年的改进）比 Linux 上更成熟。</p> <p>从 2017 年开始，Linux 和 Windows 容器都支持 Service Fabric。</p>
--	---

在微软 Azure 中使用容器编排引擎

许多云供应商提供 Docker 容器加上 Docker 集群和编排的支持，包括微软 Azure、Amazon EC2 容器服务和 Google 容器引擎。Microsoft Azure 通过 Azure 容器服务（ACS）来提供 Docker 集群和编排，这部分内容会在下一章介绍。

另一个选择是使用微软 Azure Service Fabric（一个微服务平台），它也支持基于 Linux 和 Windows 容器的 Docker。Service Fabric 可以运行在 Azure 或其他云上，也可以在[本地运行](#)。

使用 Azure 容器服务（Container Service）

一个 Docker 集群组合多个 Docker 主机，然后把它们暴露为单个虚拟的 Docker 主机，这样就能在集群里部署多个容器。集群会处理所有的复杂管理工作（比如可扩展性和运行状况等等）。图 4-23 展示了一个 Docker 集群是如何组合应用程序并映射到 Azure 容器服务（ACS）上的。

ACS 为预先配置为运行容器化应用的虚拟机集群的创建、配置和管理提供了一种简化的方法。通过使用流行的开源调度和业务流程工具的优化配置，ACS 使您能够利用现有技能，或利用大量不断增长的社区专业知识来部署和管理 Microsoft Azure 上的基于容器的应用。

Azure 容器服务（Container Service）专门为 Azure 上流行的 Docker 集群开源工具和技术进行了配置优化，因此您获得了一个开放的解决方案，它同时提供了容器配置和应用配置的可移植性。您只需选择大小、主机数量和编排引擎工具，容器服务会处理其他所有内容。

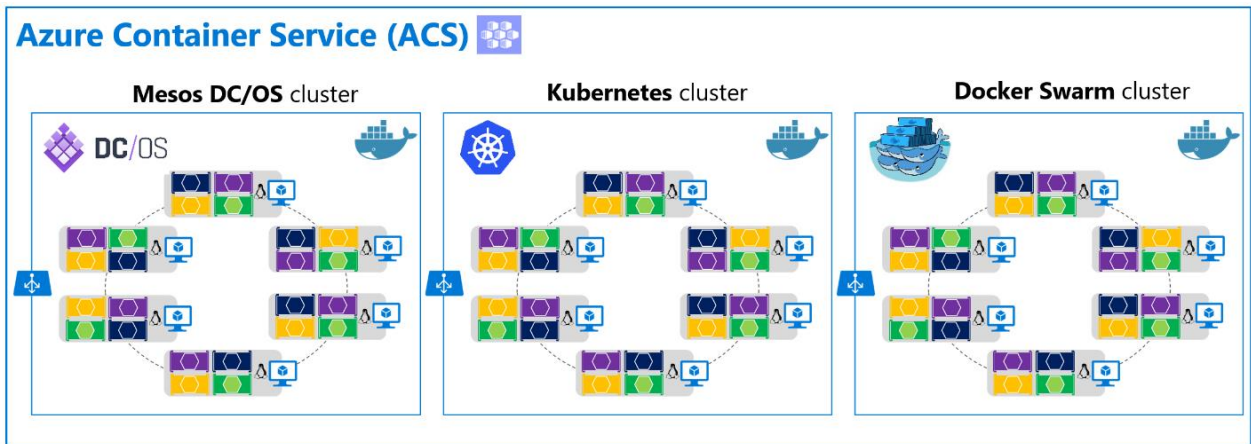


图4-24. Azure 容器服务 (Container Service) 的集群选择

ACS 利用 Docker 镜像来确保应用的容器是完全可移植的。它支持选用开源编排引擎比如 DC/OS（基于 Apache Mesos）、Kubernetes（最初由 Google 创建）和 Docker Swarm 等来确保应用可以扩展到数千甚至数万个容器。

Azure 容器服务 (Container Service) 可以让您在利用 Azure 企业级功能的同时在编排引擎层面保持应用的可移植性。

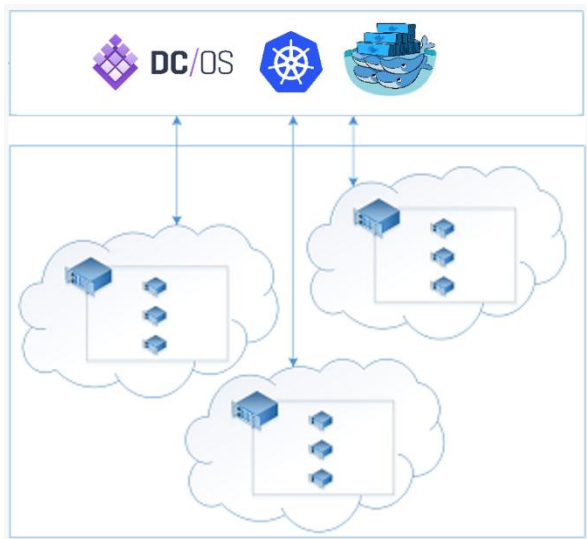


图4-25. ACS 里的编排引擎

如图 4-25 所示，Azure 容器服务 (Container Service) 只是 Azure 提供的用来部署 DC/OS、Kubernetes 或 Docker Swarm 的简单基础架构，ACS 没有实现任何额外的编排引擎。因此 ACS 不是编排引擎而只是利用现有的开源编排引擎的基础架构。

从使用的角度来看，Azure 容器服务 (Container Service) 的目标是通过使用流行的开源工具和技术来提供容器托管环境。为此，它为所选的编排引擎暴露了标准的 API 接口。通过这些 API，您可以使用任

何能与它们通信的软件。例如，对于 Docker Swarm 的 API 会选择 Docker 命令行接口 (CLI)，对于 DC/OS 可以选择使用 DC / OS CLI。

Azure 容器服务入门

要开始使用 Azure 容器服务 (Container Service)，请通过 Azure 门户里的 Azure 资源管理器 (Resource Manager) 模板或 [CLI](#) 来部署 Azure 容器服务 (Container Service) 集群。可用的模板包括 [Docker Swarm](#)、[Kubernetes](#) 和 [DC/OS](#)。可以修改快速入门模板以包含额外的或高级的 Azure 配置。有关部署 Azure 容器服务 (Container Service) 集群的更多信息，请参阅“[部署 Azure 容器服务集群](#)”。

作为 ACS 的一部分默认安装的任何软件都不收费。所有默认选项都是用开源软件实现的。

ACS 目前可用于标准的 A、D、DS、G 和 GS 系列的 Linux 虚拟机。它仅对所选择的计算实例以及其他底层基础资源 (如存储和网络) 进行计费。ACS 本身没有增量费用。

其他资源

- **使用 Azure 容器服务的 Docker 容器主机解决方案介绍**
<https://azure.microsoft.com/documentation/articles/container-service-intro/>
- **Docker Swarm 概览**
<https://docs.docker.com/swarm/overview/>
- **Swarm 模式概览**
<https://docs.docker.com/engine/swarm/>
- **Mesosphere DC/OS 概览**
<https://docs.mesosphere.com/1.7/overview/>
- **Kubernetes. 官方网站**
<http://kubernetes.io/>

使用 Azure Service Fabric

Azure Service Fabric 来自于微软从提供打包产品 (通常是单体式) 到提供服务的过渡阶段创建和运营大规模服务的经验，比如 Azure SQL 数据库 (Azure SQL Database)、Azure 文档数据库 (Azure Document DB)、Azure 服务总线 (Service Bus) 或 Cortana 后端，形成了 Service Fabric。随着这个平台的逐渐演变，越来越多的服务开始使用它。重要的是，Service Fabric 不仅能运行在 Azure 上，还可以在独立部署的 Windows Server 上运行。

Service Fabric 的目的是解决创建和运行服务的难题，以及高效利用基础架构资源的问题，使团队能够使用微服务方法解决业务问题。

Service Fabric 在如下两个领域来帮助您创建使用微服务方案的应用：

- 一个平台，提供一些系统服务来部署、扩展、升级、检测和重启失败服务，以及发现服务位置、管理状态和监控运行状况。这些系统服务有效地实现以前描述的微服务的许多特征。

- 编程 API 或框架，帮助您将应用构建为微服务：[Reliable Actors](#) 和 [Reliable Services](#)。当然，您可以选择其他代码来创建微服务，但这些 API 使工作更加直接了当，并且在更深层次上与平台集成。这样可以获得运行状况和诊断信息，或者可以利用可靠的状态管理。

Service Fabric 与如何构建服务无关，您可以使用任何技术。但是，它提供了内置的编程 API，可以更容易地构建微服务。

如图 4-26 所示，您可以在 Service Fabric 中用简单的进程或 Docker 容器的形式来创建和运行微服务。还可以将基于容器的微服务与基于进程的微服务组合在同一个 Service Fabric 集群中。

Azure Service Fabric - 集群类型

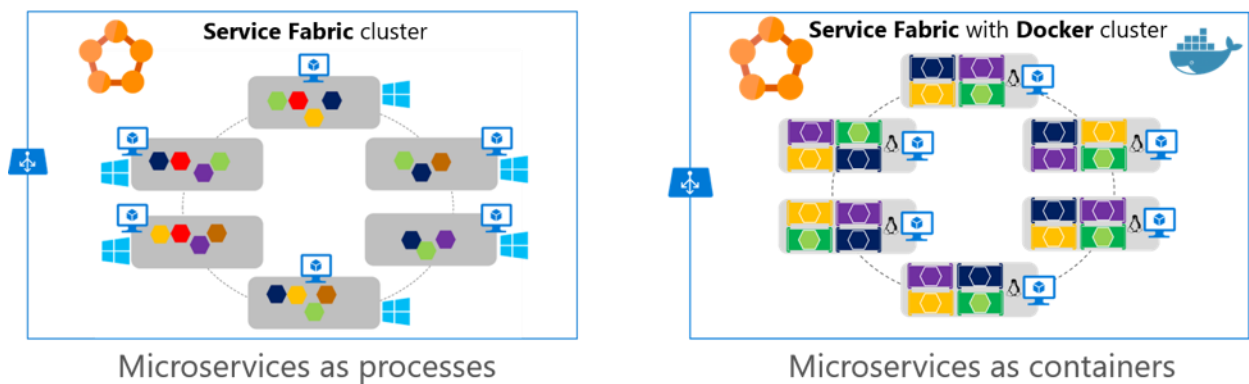


图 4-26. 在 Azure Service Fabric 以进程或容器部署微服务

基于 Linux 和 Windows 主机的 Service Fabric 集群可以运行 Docker Linux 容器和 Windows 容器。

有关 Azure Service Fabric 容器支持的最新信息，请参阅 [Service Fabric 和容器](#)。

Service Fabric 是一个很好的平台例子，在上面可以定义与物理实现不同的逻辑架构（业务微服务或限界上下文），可以参考前文“[逻辑架构和物理架构](#)”一节所介绍的内容。例如，如果在 [Azure Service Fabric](#) 中实现有状态的 [Reliable Services](#)，您将了解到具有多个物理服务的业务微服务的概念。这会在后续的“[无状态与有状态的微服务](#)”一节作介绍。

如图 4-27 所示，从逻辑/业务微服务的角度来看，在实施 Service Fabric 有状态的 [Reliable Service](#) 时，通常要实现两层服务。首先是处理多个分区的后端有状态的 [Reliable Service](#)。其次是前端服务或网关服务，负责跨多个分区或有状态服务实例进行路由和数据聚合。该网关服务还结合 Service Fabric [反向代理](#)来处理使用后端重试循环的客户端通信。

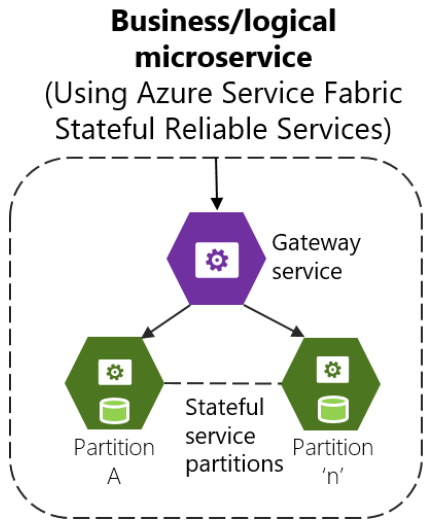


图 4-27. 使用多个有状态服务实例和一个自定义网关前端的业务微服务

任何情况下，当您使用 Service Fabric 有状态的 Reliable Services 时，同时也具有通常由多个物理服务组成的逻辑或业务微服务（限界上下文）。如图 4-27 所示，网关服务和分区服务都可以实现为 ASP.NET Web API 服务。

在 Service Fabric 中，可以将服务分组和部署为 [Service Fabric 应用](#)，它是编排引擎或集群用来打包和部署的基本单位。因此，Service Fabric 应用也可以映射到自治的业务和逻辑微服务的边界或限界上下文中。

Service Fabric 和容器

对于 Service Fabric 中的容器，还可以使用 Service Fabric 集群在容器镜像中部署服务。如图 4-28 所示，大多数情况下每个服务只有一个容器。

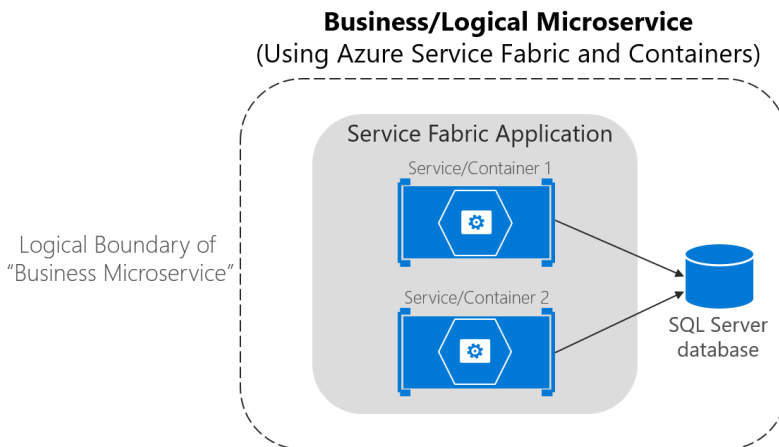


图 4-28. 在 Service Fabric 中使用多个服务（容器）的业务微服务

然而，所谓的“边斗车”容器（两个容器必须作为逻辑服务的一部分一起部署）在 Service Fabric 中也是可能的。重点在于业务微服务就是围绕几个内聚元素的逻辑边界。多数情况下可能是单独的服务具有单个数据模型，但在其他情况下，也可能拥有多个实体服务。

截至 2017 年中旬，通过 Service Fabric 还无法在容器上部署有状态的 Reliable Services，只能部署无状态服务和角色服务。但请注意，您可以在同一个 Service Fabric 应用的容器中混合使用进程中的服务和或容器中的服务，如图 4-29 所示。

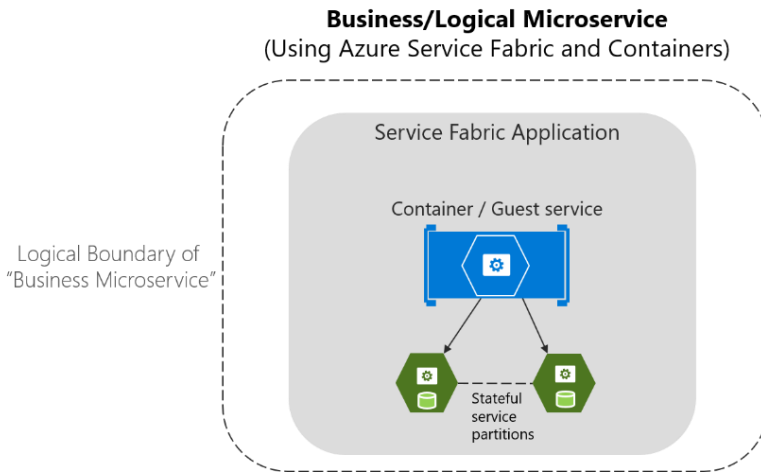


图 4-29. 使用容器和有状态服务映射到一个 Service Fabric 应用的业务微服务

如果想了解更多的 Azure Service Fabric 容器支持情况，请参阅 [Service Fabric 和容器](#)。

无状态和有状态的微服务

如前所述，每个微服务（逻辑限界上下文）必须拥有其领域模型（数据和逻辑）。在无状态微服务的情况下，数据库将是外置的，采用关系型数据库如 SQL Server 或 NoSQL 数据库如 MongoDB 或 Azure 文档数据库（Document DB）。

但服务本身也可以是有状态的，这意味着数据位于微服务内部。这些数据可能不是存在于同一台服务器上，但肯定是在微服务进程内的内存中和持久化在硬盘上，并复制到其他节点。图 4-30 显示了这些不同的方法。

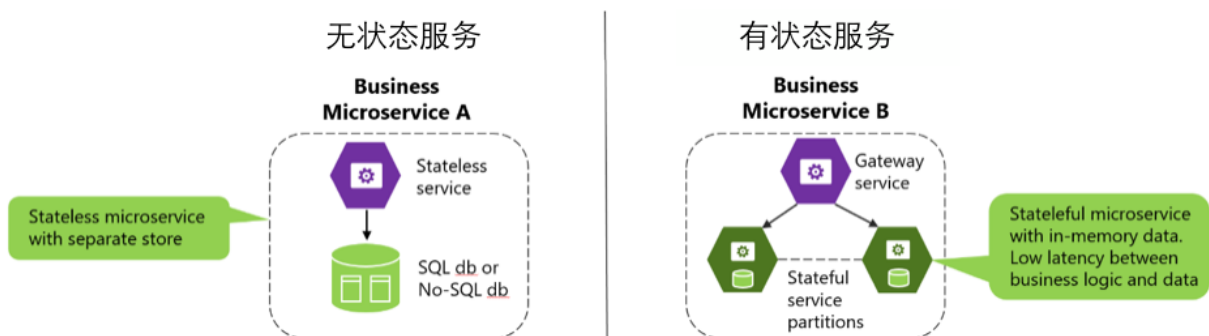


图 4-30. 无状态和有状态的微服务

无状态方法是完全有效的并且比有状态的微服务更容易实现，因为它类似于传统的和公认的模式。但是无状态的微服务在进程和数据源之间施加了延迟。当尝试通过额外的缓存和队列来提高性能时还涉及到更多需要移动的数据块。结果就是以过多层次的复杂架构结束。

相比之下，[有状态的微服务](#)可以在高级场景中胜任，因为领域逻辑和数据之间没有延迟。大型数据处理、游戏后端、数据库即服务和其他低延迟场景都可以从状态服务中受益，因为能更快地访问本地状态。

无状态和有状态的服务是互补的。例如，在图 4-30 的右边是一个能够划为多个分区的有状态服务。要访问这些分区，可能需要一个无状态服务充当网关，该服务知道如何根据分区键对每个分区进行寻址。

有状态的服务确实有缺点。它们在允许扩展上施加了一定程度的复杂性。通常由外部数据库系统实现的功能必须针对诸如跨有状态微服务和数据分区的数据复制等任务进行处理。然而，这是一个编排引擎（比如包含[有状态的 Reliable Services](#) 的 [Azure Service Fabric](#)）能发挥作用的领域之一，它通过使用 [Reliable Services API](#) 和 [Reliable Actors](#) 来简化有状态的微服务的开发过程和生命周期。

其他允许有状态的服务、并且支持 Actor 模式，改善了业务逻辑和数据之间的容错和延迟的微服务框架有来自微软研究院的 Microsoft [Orleans](#) 和 [Akka.NET](#)。这两个框架正在改进对 Docker 的支持。

请注意，Docker 容器本身是无状态的。如果要有状态的服务，则需要先前提到的额外的规范和更高级别的框架。

基于 Docker 的应用程序的开发流程

愿景

为了实现用您喜欢的方式开发容器化的应用程序，在 IDE 方面，微软通过 Visual Studio 和 Visual Studio 的 Docker 工具来提供支持，同时，在 CLI / 编辑器方面，微软通过 Docker CLI 和 Visual Studio Code 来提供支持。

Docker 应用程序的开发环境

开发工具的选择：IDE 还是编辑器？

无论您是喜欢全功能、强大的 IDE，还是钟情于轻量级的、灵活的编辑器，您都可以使用微软提供的工具来开发 Docker 应用程序。

Visual Studio (Windows 版本)。当使用 Visual Studio 开发基于 Docker 的应用程序的时候，推荐使用 Visual Studio 2017 或更高的版本，这些版本已经内置 Docker 工具了。这些 Docker 工具都可以支持您直接在目标 Docker 环境中开发、运行和验证应用程序。您可以按 F5，直接进入 Docker 宿主中运行和调试您的应用程序（单容器应用程序或多容器应用程序），或者，您也可以按 CTRL+F5，在不重新构建容器的前提下编辑和刷新您的应用程序。对于基于 Docker 的应用程序的开发来说，这是最强大的选择。

Visual Studio for Mac。这是一个 IDE，是由 Xamarin Studio 演化而来的，运行在 MacOS 之上，从 2017 年中期开始支持 Docker。对于那些工作在 Mac 机上，想使用一款强大的 IDE 的开发人员来说，这应该是一个更好的选择。

Visual Studio Code 和 Docker CLI。如果您更喜欢可以支持任意开发语言的、轻量级的、跨平台的编辑器，您可以使用 Microsoft Visual Studio Code (VS Code) 和 Docker CLI。这是一个跨平台的开发解决方案，支持 Mac、Linux 和 Windows。

通过安装 [Docker 社区版 \(CE\)](#) 工具，您可以使用一个 Docker CLI 来构建 Windows 应用程序和 Linux 应用程序。

其他资源

- **Visual Studio 的 Docker 工具**
<https://docs.microsoft.com/aspnet/core/publishing/visual-studio-tools-for-docker>
- **Visual Studio Code**. 官方网站
<https://code.visualstudio.com/download>
- **Mac 和 Windows 下的 Docker 社区版 (CE)**
<https://www.docker.com/community-editions>

支持 Docker 容器的 .NET 语言和框架

正如本书前几章所提到的那样，如果要开发 Docker 容器化的 .NET 应用程序，您可以使用 .NET Framework、.NET Core 或者开源的 Mono 项目。如果是基于 Linux 或 Windows 容器的应用程序，根据您的使用的 .NET 框架，您可以使用 C#、F# 或 Visual Basic。关于 .NET 语言的详细信息，可以参考一下这篇博客：[.NET 语言策略](#)。

Docker 应用程序的开发流程

应用程序开发生命周期从每一个开发人员的机器开始，在开发机器上，开发人员使用他们最喜欢的语言来为应用程序编写代码，然后在本机上测试它。不管开发人员选择哪种语言、框架和平台，使用本流程，只需开发和测试 Docker 容器即可，而且是在本机做这些事情。

每个容器（Docker 镜像的一个实例）包含以下组件：

- 一个选定的操作系统（一个 Linux 发行版、Windows Nano Server 或 Windows Server Core）
- 开发人员添加的一些文件（应用程序的二进制文件等等）
- 配置信息（环境设置和依赖）

基于 Docker 容器的应用程序的开发流程

本小节主要说明了基于 Docker 的应用程序的“内环”开发流程。“内环”开发流程的意思是：不考虑更宽泛的 DevOps 流程，只关注开发人员机器上的开发工作本身。此流程并不包含安装环境的初始步骤，这些工作过去已经一次性完成了。

一个应用程序由您自己的服务和附加的库（依赖）组成。下面是构建 Docker 应用程序的一些基本步骤，如图 5-1 所示：

Docker应用的“内环”开发流程

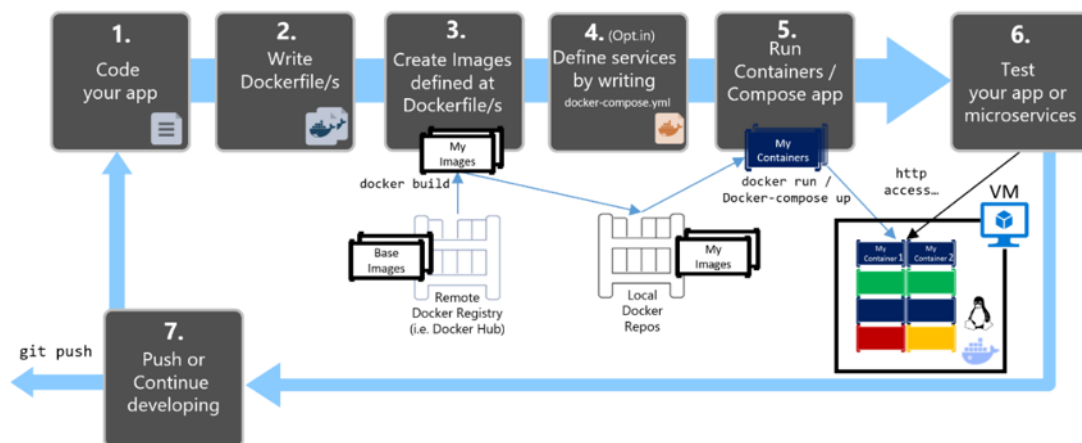


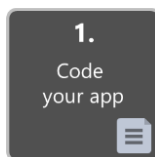
图5-1. 开发 Docker 容器化应用程序的步骤

在本书中，会详细解析整个流程，同时，每个主要的步骤都会基于 Visual Studio 环境，给出详细的说明。

当您使用编辑器/CLI 开发工具（例如：Visual Studio Code + macOS 或 Windows 上的 Docker CLI）的时候，您必须清楚地了解每一个步骤的细节，一般来说，肯定要比使用 Visual Studio 了解的更详细一些。关于在 CLI 环境下开发工作的更多信息，可以参考这本电子书：[《使用微软平台和工具的容器化 Docker 应用的生命周期》](#)。

当您使用 Visual Studio 2015 或 Visual Studio 2017 的时候，这些步骤中的绝大多数已经帮您处理好了，这可以极大地提高生产效率。尤其是使用 Visual Studio 2017 开发多容器应用程序的时候。举例来说，只要点一下鼠标，Visual Studio 就会把包含应用程序配置的 Dockerfile 和 docker-compose.yml 文件添加到您的项目中。当您在 Visual Studio 中运行这个应用程序的时候，它会帮您构建 Docker 镜像，然后直接在 Docker 中运行多容器应用程序。这些特性可以加快您的开发速度。

但是，Visual Studio 自动化了这些步骤，并不意味着您就可以不用了解在底层是如何使用 Docker 的。所以，在接下来的指引中，我们会详细解释每一个步骤。



步骤 1、开始编码，然后创建您的原始应用程序或基础服务

开发一个 Docker 应用程序和开发一个非 Docker 的应用程序的方法基本类似。不同之处在于，基于 Docker 开发的时候，在本地环境（一台由 Docker 安装的 Linux 虚拟机，或者，如果使用 Windows 容器的话，可以直接使用 Windows）中，您需要部署和测试运行在 Docker 容器内部的应用程序或服务。

用 Visual Studio 来搭建您的本地环境

首先，确保您已经安装了 [Windows 上的 Docker 社区版 \(CE\)](#)，下面这个指南中提供了更详细的说明：[Windows 上的 Docker CE 入门](#)。

此外，您还需要安装 Visual Studio 2017，相对于使用带有 Visual Studio Tools for Docker 插件的 Visual Studio 2015 来说，这是一个更好的选择，因为 Visual Studio 2017 为 Docker 提供了更高级别的支持，比如说，调试容器方面的支持。如果在安装的时候，您选择了 .NET Core 和 Docker 安装项，那么 Visual Studio 2017 就已经包含 Docker 工具了。如图 5-2 所示：

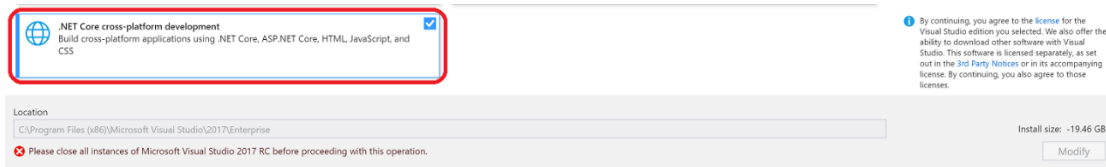


图 5-2. 安装 Visual Studio 2017 的时候选择 .NET Core and Docker 安装项

甚至在您的应用程序启用 Docker 之前，您就可以直接用 .NET（如果您计划要使用容器，那么通常用 .NET Core）开始编码了，然后再在 Docker 中部署和测试这个应用程序。但是，强烈建议您尽早开始在 Docker 上开始您的工作，因为这是一个真实的环境，各种问题会尽早地暴露出来。Visual Studio 可以让 Docker 开发变得如此轻松，以至于您根本感觉不到它的存在——尤其是在 Visual Studio 中调试多容器应用程序的时候，所以强烈建议您尽早开始在 Docker 上开始您的工作。

其他资源

- **Windows 上的 Docker 社区版入门**
<https://docs.docker.com/docker-for-windows/>
- **Visual Studio 2017**
<https://www.visualstudio.com/vs/visual-studio-2017/>



步骤 2、基于现有的 .NET 基础镜像创建一个 Dockerfile

对于您想构建的每一个自定义镜像来说，Dockerfile 都是不可或缺的；无论您是要在 Visual Studio 中进行自动化部署，还是用 Docker CLI（docker run 和 docker-compose 命令）手动进行部署，对于需要部署的每个容器来说，Dockerfile 也是必不可少的。如果您的应用程序只包含一个自定义服务，那么您只

需要一个 Dockerfile。如果您的应用程序包含多个服务（例如在微服务架构中），那么每一个服务都需要一个 Dockerfile。

Dockerfile 应该放在您的应用程序或服务的根目录中。它包含一些命令，这些命令会告诉 Docker 如何在容器中安装和运行您的应用程序或服务。您可以用编码的方式手动创建一个 Dockerfile，然后把它和其他应用程序依赖一起添加到您的项目中。

如果用 Visual Studio 和它的 Docker 工具，这个任务只需轻点几下鼠标即可完成。当您在 Visual Studio 2017 中创建一个新项目的时候，有一个叫做“启用容器（Docker）支持”的选项，如图 5-3 所示。

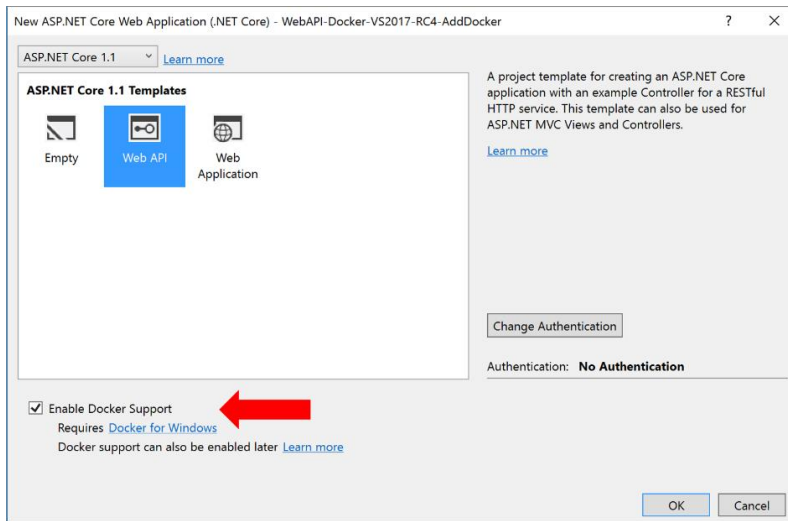


图 5-3. 在 Visual Studio 2017 中创建一个新项目时启用 Docker 支持

在 Visual Studio 中，您也可以通过右击您的项目文件，然后选择“Docker 支持”的方式，在新项目或现有项目上启用 Docker 支持。如图 5-4 所示。

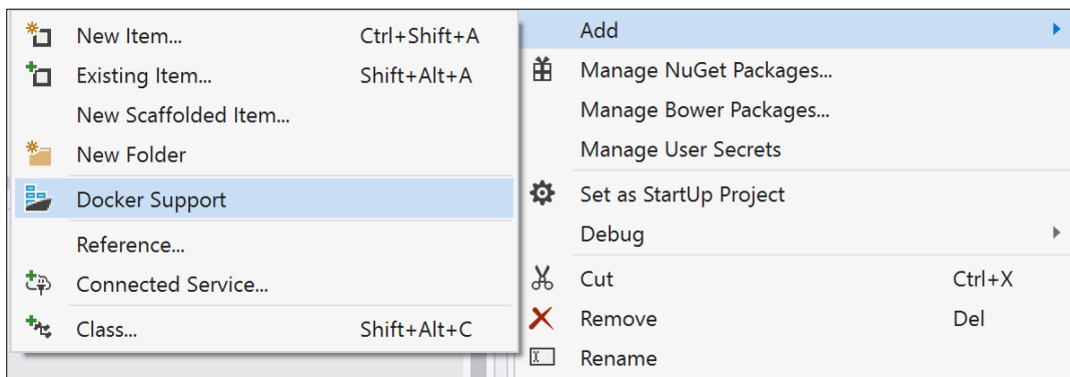


图 5-4. 在现有的 Visual Studio 2017 项目上启用 Docker 支持

在一个项目（例如：ASP.NET 网页应用程序或 Web API 服务）上做这个操作，会添加一个带有所需配置 Dockerfile 到您的项目中。它还会为整个解决方案添加一个 docker-compose.yml 文件。在接下来的

几个小节中，我们会一一说明在每一个文件中都写入了哪些具体信息。Visual Studio 可以帮您完成这个工作，但是弄清楚在 Dockerfile 中写入哪些具体信息对您来说是很有帮助的。

可选步骤 A、用现有的官方 .NET Docker 镜像创建一个项目

您通常会在基础镜像之上，为您的容器构建一个自定义的镜像，您可以在 [Docker Hub](#) 注册表的官方仓库中获取到这些基础镜像。当您在 Visual Studio 中启用 Docker 支持之后，它在底层会一丝不苟地完成各种工作。您的 Dockerfile 将会使用一个现有的 aspnetcore 镜像。

在前面，我们已经详细说明了，根据您的选择的框架和操作系统，您应该使用哪种 Docker 镜像。例如，如果您想使用 ASP.NET Core (Linux 或 Windows 版本)，应该使用的镜像是：

microsoft/aspnetcore:2.0。所以，您必须指定您的 Docker 容器的需要使用哪种基础镜像。通过把 FROM microsoft/aspnetcore:2.0 这行命令添加到您的 Dockerfile 中，您可以实现这个目标。在 Visual Studio 中，这个操作会被自动完成。但是如果您要更新这个版本，您需要自己更新这个值。

使用来自于 Docker Hub 的、带有版本号的、官方的 .NET 镜像仓库，可以确保在所有机器上（包括开发，测试和生产）同样的语言特性都可用。

下面这个例子展示了一个样例 Dockerfile，它是为 ASP.NET Core 容器而设计的。

```
FROM microsoft/aspnetcore:2.0
ARG source
WORKDIR /app
EXPOSE 80
COPY ${source:-obj/Docker/publish} .
ENTRYPOINT ["dotnet", "MySingleContainerWebApp.dll"]
```

在这个例子中，这个容器是基于 2.0 版本的、官方的、基于 Linux 的 ASP.NET Core Docker 镜像（支持 Linux 和 Windows 的多架构）的。这是通过 FROM microsoft/aspnetcore:2.0 这行命令来设置的。（关于这个镜像的详细信息，可以参考 [ASP.NET Core Docker Image](#) 和 [.NET Core Docker Image](#) 这两个网页。）在 Dockerfile 中，您还必须指定 Docker 需要监听的您在运行时使用的 TCP 端口（在这个例子中，是 80 端口，是通过 EXPOSE 来配置的）。

在 Dockerfile 中，根据您的使用的语言和框架，您还可以指定其他的设置。例如，ENTRYPOINT 这行设置为 ["dotnet", "MySingleContainerWebApp.dll"] 会告诉 Docker 需要运行一个 .NET Core 应用程序。如果您使用 SDK 和 .NET CLI (dotnet CLI) 来构建和运行 .NET 应用程序，这个设置会有所不同。最底下一行，就是 ENTRYPOINT 行，根据您的应用程序选择的语言和平台的不同，其他设置也会有所不同。

其他资源

- **为 .NET Core 应用建立 Docker 镜像**
<https://docs.microsoft.com/dotnet/articles/core/docker/building-net-docker-images>

- **Docker 官方文档：创建您自己的镜像**
<https://docs.docker.com/engine/tutorials/dockerimages/>

使用多架构镜像仓库

一个仓库 (repo) 可以包含多个平台，例如 Linux 镜像和 Windows 镜像。这个特性可以让像微软这样的提供商（基础镜像创建者）通过创建一个仓库 (repo) 就可以涵盖多个平台 (Linux 和 Windows)。例如，Docker Hub 注册表上的 [microsoft/aspnetcore](#) 仓库，通过同一个仓库名，同时实现了对 Linux 和 Windows Nano Server 的支持。

如果你指定了一个标签，那么会显式地指定某个具体的平台，如下面的实例所示：

microsoft/aspnetcore:2.0.0-jessie	基于 Linux, 只包含 .NET Core 2.0 运行时
microsoft/dotnet: 2.0.0-nanoserver	基于 Windows Nano Server, 只包含 .NET Core 2.0 运行时

但是，从 2017 年中期以来，又有了新情况，如果你指定了同样的镜像名，甚至同样的标签，那么全新的多架构镜像（例如支持多架构的 aspnetcore 镜像）会根据你部署的 Docker 主机的 OS 来选择使用的 Linux 版本或 Windows 版本，如下面的实例所示：

microsoft/aspnetcore:2.0	根据 Docker 主机的 OS, 选择 Linux 或 Windows Nano Server 上的 .NET Core 2.0 运行时
--------------------------	---

使用这种方式，当您拉取一个来自于 Windows 主机的镜像的时候，它会拉取一个 Windows 的变体，当您拉取具有同一个镜像名的来自于 Linux 主机的镜像的时候，它会拉取一个 Linux 的变体。

可选步骤 B、从零开始创建您的基础镜像

您可以从零开始创建您自己的基础镜像。对于刚开始接触 Docker 的人来说，不推荐这样做，但是如果您想要自定义自己的基础镜像，您也可以这样做。

其他资源

- **多重架构.NET Core 镜像：**
<https://github.com/dotnet/announcements/issues/14>
- **官方文档：创建基础镜像**
<https://docs.docker.com/engine/userguide/eng-image/baseimages/>



步骤 3、创建您的自定义 Docker 镜像，然后在镜像中嵌入您的应用程序或服务

对于您的应用程序中的每一个服务来说，您都需要创建一个对应的镜像。如果您的应用程序是由一个服务或 Web 应用程序组成的，那么您只需要一个镜像。

注意，在 Visual Studio 中，会自动帮您创建好 Docker 镜像。下面的步骤只有编辑器/CLI 才需要，这些步骤只是为了解释清楚底层到底发生了什么。

您作为一个开发人员，在您把一个完整的功能或变更推送到您的源代码控制系统中之前，您都需要一直在本地进行开发和测试。这意味着，您需要创建一个 Docker 镜像，把容器部署到本地的 Docker 主机（Windows 或 Linux 虚拟机）上，然后针对这些本地的容器进行运行、测试和调试。

要在本地环境中，用 Docker CLI 和您的 Dockerfile 创建一个自定义的镜像，您可以使用 `docker build` 命令，如图 5-5 所示：

```
PS C:\dev\netcore-webapi-microservice-docker> docker build -t cesardl/netcore-webapi-microservice-docker:first .
Sending build context to Docker daemon 1.148 MB
Step 1 : FROM microsoft/dotnet:latest
latest: Pulling from microsoft/dotnet
5c90d4a2d1a8: Downloading [=====>] 18.34 MB/51.35 MB
ab30c63719b1: Downloading [=====>] 18.48 MB/18.55 MB
c6072700a242: Downloading [=====>] 18.34 MB/42.53 MB
121d7eef6c20: Waiting
eb57cf4f29ee: Waiting
b2c5ae2d325b: Waiting
```

图 5-5. 创建一个自定义的 Docker 镜像

除了从项目文件夹直接运行 `docker build` 命令，您还可以通过运行 `dotnet publish` 命令先生成一个带有必需的 .NET 库和二进制文件的部署文件夹，然后再使用 `docker build` 命令。

这回创建一个名为 `cesardl/netcore-webapi-microservice-docker:first` 的镜像。在这个例子中，`:first` 是一个标签，它表示一个特定的版本。对于组合应用程序来说，对您需要创建的每一个自定义镜像重复此步骤即可。

当一个应用程序由多个容器组成的时候（也就是说，它是一个多容器应用程序），您也可以使用 `docker-compose up -build` 这一行命令，通过相关的 `docker-compose.yml` 文件暴露出来的元数据，构建出所有相关的镜像。

您可以用 `docker images` 命令找到在您本地的仓库中有哪些镜像，如图 5-6 所示：

```
PS C:\dev\netcore-webapi-microservice-docker> docker images
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
cesardl/netcore-webapi-microservice-docker  first       384c4ac1809b     4 minutes ago   579.8 MB
microsoft/dotnet    latest      49aaaf5daa850    30 hours ago    548.6 MB
ubuntu              latest      cf62323fa025     5 days ago      125 MB
hello-world         latest      c54a2cc56cbb     12 days ago     1.848 kB
```

图 5-6. 使用 `docker images` 命令查看现有的镜像

用 Visual Studio 创建 Docker 镜像

当您用 Visual Studio 创建一个支持 Docker 的项目的时候，您无需显式地创建一个镜像。当您按 F5 运行这个容器化的应用程序或服务的时候，Visual Studio 会帮您创建这个镜像。在 Visual Studio 中，这个步骤是自动的，您并不会看到这个步骤，但是了解底层发生了什么是很重要的。



步骤 4、当构建多容器 Docker 应用程序的时候，在 docker-compose.yml 中定义您的服务

docker-compose.yml 文件可以让您把需要部署一系列相关的服务，定义成一个组合应用程序，然后用部署命令进行部署。

要使用 docker-compose.yml 文件，您需要在您的解决方案的根目录创建这个文件，它的内容和下面这个例子类似：

```
version: '3'
services:
  webmvc:
    image: eshop/web
    environment:
      - CatalogUrl=http://catalog.api
      - OrderingUrl=http://ordering.api
    ports:
      - "80:80"
    depends_on:
      - catalog.api
      - ordering.api
  catalog.api:
    image: eshop/catalog.api
    environment:
      - ConnectionString=Server=sql.data;Port=1433;Database=CatalogDB;...
    ports:
      - "81:80"
    depends_on:
      - postgres.data
  ordering.api:
    image: eshop/ordering.api
    environment:
      - ConnectionString=Server=sql.data;Database=OrderingDb;...
    ports:
      - "82:80"
    extra_hosts:
      - "CESARDLBOOKVHD:10.0.75.1"
    depends_on:
      - sql.data
  sql.data:
```

```
image: mssql-server-linux:latest
environment:
  - SA_PASSWORD=Pass@word
  - ACCEPT_EULA=Y
ports:
  - "5433:1433"
```

注意，这个 docker-compose.yml 文件是一个简化过的、合并的版本。它包含每一个容器的静态配置数据（比如，自定义镜像的名字），这些数据是肯定包含的，此外，还包含一些依赖于部署环境的配置信息，比如，连接字符串等。在后面几个小节中，您会学习到怎么把 docker-compose.yml 中的配置分割到多个 docker-compose 文件中，并且重写依赖于环境和执行类型（debug 或 release）的一些值。

这个 docker-compose.yml 文件的例子中定义了 4 个服务，1 个 Web MVC 服务（一个 Web 应用程序），2 个微服务（ordering.api 和 basket.api），和 1 个数据源容器：基于 SQL Server for Linux 的作为一个容器来运行的 sql.data。每个服务都作为一个容器来部署，所以对于每个服务来说，一个 Docker 镜像是必需的。

这个 docker-compose.yml 文件不仅指定了使用哪种容器，还指定了它们各自的配置是怎样的。例如，在这个 .yml 文件中，webmvc 容器的定义说明如下：

- 使用预构建的 eshop/web: latest 镜像。但是，您也可以配置这个影响在 docker-compose 执行的时候被构建，这可以通过 docker-compose 文件中的 “build:” 这个附加的配置项来实现。
- 初始化了 2 个环境变量（CatalogUrl 和 OrderingUrl）。
- 在这个容器上，把 80 端口暴露给主机上的外部 80 端口。
- 用 depends_on 这个设置把这个 Web 服务链接到购物篮和下单服务上。这个设置可以让本服务在其他指定的服务之后再启动。

在后面一章，当我们讨论如何实现微服务和多容器应用程序的时候，我们还会重新讨论这个 docker-compose.yml 文件的。

Visual Studio 2017 中使用 docker-compose.yml

当您给 Visual Studio 解决方案中的一个服务项目添加 Docker 支持的时候，如图 5-7 所示，Visual Studio 会在您的项目中添加一个 Dockerfile，同时，它还会在您的解决方案中添加一个带有多个 docker-compose.yml 文件的服务节点（项目）。对于组合您的多容器解决方案来说，这是一个比较轻松、快捷的方法。您可以打开这些 docker-compose.yml 文件，更新一下它们的其他特性。

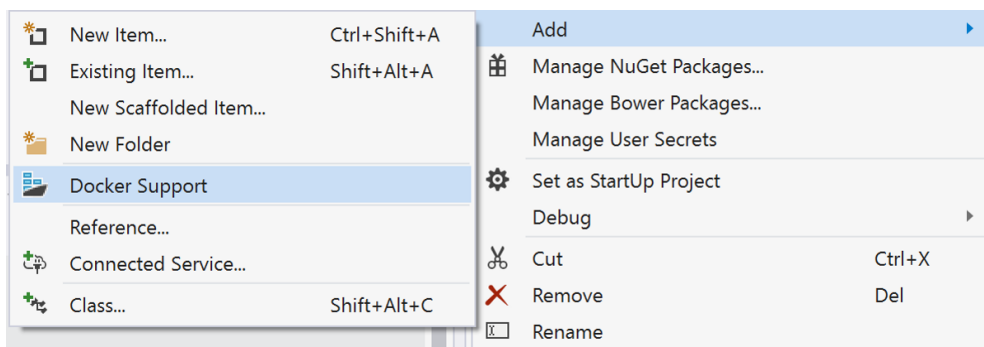


图 5-7. 在 Visual Studio 2017 中，通过右击一个 ASP.NET Core 项目来添加 Docker 支持

在 Visual Studio 中添加 Docker 支持，不仅会在您的项目中添加 Dockerfile，还会在几个全局的 docker-compose.yml 文件中添加一些配置信息，这是一些解决方案级的设置。

当您在 Visual Studio 中给您的解决方案添加了 Docker 支持以后，您会在解决方案资源管理器中看到一个新的节点（在 docker-compose.dcproj 项目中），它包含一些 Visual Studio 添加的 docker-compose.yml 文件，如图 5-8 所示。

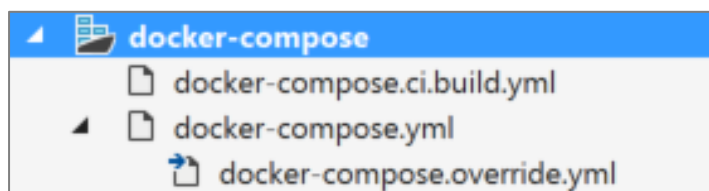


图 5-8. Visual Studio 2017 解决方案资源管理器中 docker-compose 树节点

通过 docker-compose up 命令，您可以用一个 docker-compose.yml 文件部署多容器应用程序。但是，Visual Studio 还是会添加一组 .yml 文件，以便于您可以根据具体环境（开发环境或生产环境）和执行类型（release 或 debug）重写一些值。这个功能在后面的一些章节还会详细解释的。



步骤 5、生成和运行您的 Docker 应用程序

如果您的应用程序只有一个容器，您可以把它部署到您的 Docker 主机（虚拟机或物理机）来运行它。但是，如果您的应用程序包含多个服务，您可以把它作为一个组合应用程序来部署，用一行 CLI 命令（docker-compose up）或者用 Visual Studio（在后台也是使用这个命令）都可以完成这个操作。让我们看一下 2 个不同的可选步骤：

可选步骤 A：用 Docker CLI 运行一个单容器应用程序

您可以用 `docker run` 命令运行一个 Docker 容器，如图 5-9 所示：

```
docker run -t -d -p 80:5000 cesardl/netcore-webapi-microservice-docker:first
```

```
PS C:\dev\netcore-webapi-microservice-docker> docker run -t -d -p 80:5000 cesardl/netcore-webapi-microservice-docker:first
d96975a683b0a9411595816f63be6c135801878b8a85181a4d86dc848ea4ca6f
```

图 5-9. 用 `docker run` 命令运行一个 Docker 容器

在这个例子中，这个命令把容器内部的 5000 端口绑定到了主机的 80 端口上。这意味着，主机会监听 80 端口，然后把消息发送到这个容器的 5000 端口上。

可选步骤 B：运行一个多容器应用程序

在大多数企业场景中，一个 Docker 应用程序通常是由多个服务组成的，这意味着，您需要运行多容器应用程序，如图 5-10 所示：

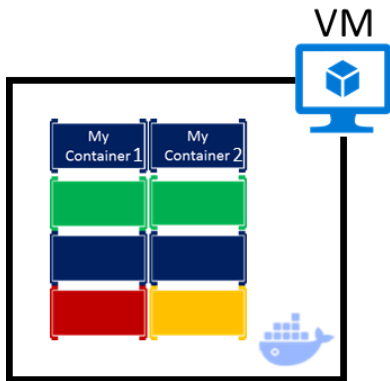


图 5-10. 部署了 Docker 容器的虚拟机

用 Docker CLI 运行一个多容器应用程序

要用 Docker CLI 运行一个多容器应用程序，您可以使用 `docker-compose up` 命令。这个命令使用 `docker-compose.yml` 文件，这个文件位于您要部署的多容器应用程序的解决方案那一级。图 5-11 展示了从您的主项目目录（它包含 `docker-compose.yml` 文件）运行这个命令的结果。

```
PS C:\Dev\WebApplication> docker-compose up
Creating webapplication_webapplication_1
Attaching to webapplication_webapplication_1
webapplication_1 | Hosting environment: Production
webapplication_1 | Content root path: /app
webapplication_1 | Now listening on: http://*:80
webapplication_1 | Application started. Press Ctrl+C to shut down.
```

图 5-11. 运行 `docker-compose up` 命令的结果

运行完 `docker-compose up` 命令以后，应用程序和它相关的容器会被部署到 Docker 主机中，如图 5-10 所示。

用 Visual Studio 运行和调试一个多容器应用程序

用 Visual Studio 2017 运行一个多容器应用程序已经简单的不能再简单了。您不仅可以运行这个多容器应用程序，您还可以设置一些常规的断点直接在 Visual Studio 中调试它包含的所有的容器。

正如前面所提到的那样，每次您在一个解决方案中给一个项目添加 Docker 解决方案支持的时候，都会在全局的（在解决方案级）`docker-compose.yml` 文件中配置这个项目，这可以让您快速地运行和调试整个解决方案。Visual Studio 会为每个添加了 Docker 解决方案支持的项目单独启动一个容器，然后帮您执行所有后台的步骤（`dotnet publish`，`docker build` 等等）。

最重要的一点是，在 Visual Studio 2017 中，有一个附加的 F5 键 Docker 命令，如图 5-12 所示。这个选项会运行解决方案级的 `docker-compose.yml` 文件中定义的所有容器，以此来运行和调试一个多容器应用程序。可以调试多容器解决方案意味着您可以设置几个断点，每个断点在不同的项目（容器）中，在 Visual Studio 中调试的时候，运行在不同容器上的各个项目中的断点都会中断的。

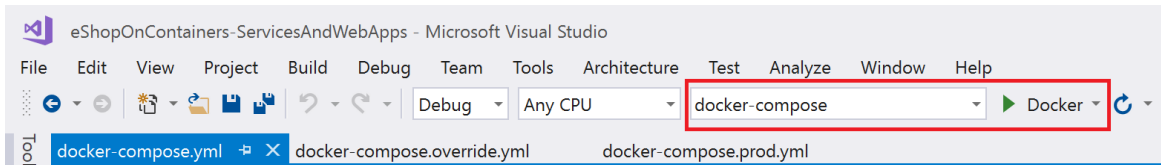


图 5-12. 在 Visual Studio 2017 中运行多容器应用程序

其他资源

- **将 ASP.NET 容器部署到远程 Docker 主机**
<https://azure.microsoft.com/documentation/articles/vs-azure-tools-docker-hosting-web-apps-in-docker/>

用编排引擎测试和部署的注意事项

`docker-compose up` 和 `docker run` 命令（或者在 Visual Studio 中运行和调试容器），对于在开发环境中测试容器来说，勉强够用了。但是，如果您面向的是 Docker 集群和像 Docker Swarm、Mesosphere DC/OS 或 Kubernetes 这样的编排引擎，就不应该使用这个方法了。如果您使用的是像 [Docker Swarm 模型](#) 这样的集群（在 Docker CE for Windows 和 Mac 的 1.12 版后可用），那么您需要用 [docker service create](#) 这样的命令来部署和测试单个的服务。如果您正在部署一个由多个容器组成的应用程序，您可以用 [docker compose bundle](#) 和 [docker deploy myBundleFile](#) 命令，把组合应用程序部署成一个“栈”。详细信息，可以参考一下 Docker 站点上的这篇博客“[Introducing Experimental Distributed Application Bundles in the Docker documentation](#)”。

对于 [DC/OS](#) 和 [Kubernetes](#) 来说，您也需要使用不同的部署命令和脚本。

6.

Test
your app or
microservices

步骤 6、用您本地的 Docker 主机测试您的 Docker 应用程序

根据您的应用程序功能的不同，这一步也会有所不同。在一个简单的 .NET Core Web 应用程序中，它会作为一个容易或服务来部署。在 Docker 主机上打开一个浏览器，然后导航到图 5-13 所示的那个站点，您就可以访问这个服务了。（如果 Dockerfile 中的配置把这个容器映射到了主机的非 80 端口上，还应该在 URL 上添加相应的主机端口号。）

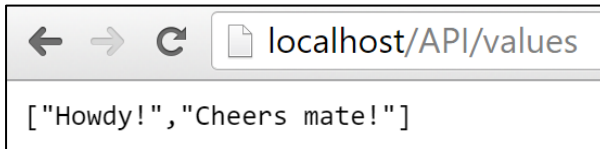


图 5-13. 用 localhost 在本地测试您的 Docker 应用程序

如果 localhost 并没有指向 Docker 主机的 IP（默认情况下，如果使用 Docker CE，应该会指向的），要导航到您的服务，需要使用您的机器的网卡的 IP 地址。

注意，对于这个正在被讨论的特定的容器样例来说，浏览器中的这个 URL 使用的是 80 端口。但是，在内部，这个请求会被重定向到 5000 端口上，因为它是用上一个步骤中解释过的 docker run 命令部署的。

您也可以在终端中，通过 curl 来测试这个应用程序，如图 5-14 所示。如果在 Windows 上安装 Docker，那么除了您机器的实际 IP 地址，还有默认的 Docker 主机 IP，是 10.0.75.1。

```
PS C:\dev\netcore-webapi-microservice-docker> curl http://10.0.75.1/API/values
StatusCode      : 200
StatusDescription : OK
Content         : [\"Howdy!\", \"Cheers mate!\"]
RawContent     : HTTP/1.1 200 OK
                Transfer-Encoding: chunked
                Content-Type: application/json; charset=utf-8
                Date: Thu, 14 Jul 2016 19:48:18 GMT
                Server: Kestrel

                [\"Howdy!\", \"Cheers mate!\"]
Forms          : {}
Headers       : {[Transfer-Encoding, chunked], [Content-Type, application/json;
                charset=utf-8], [Date, Thu, 14 Jul 2016 19:48:18 GMT], [Server, Kestrel]}
Images        : {}
InputFields   : {}
Links         : {}
ParsedHtml    : mshtml.HTMLDocumentClass
RawContentLength : 25
```

图 5-14. 用 curl 在本地测试您的 Docker 应用程序

用 Visual Studio 2017 测试和调试容器

当使用 Visual Studio 2017 运行和调试容器的时候，您可以用过去常规的方法来调试 .NET 应用程序，就像没有使用容器一样。

不使用 Visual Studio 2017 测试和调试

如果您正在使用编辑器/CLI 来开发，那么调试容器会更困难一些，您只能通过生成跟踪的方式来调试。

其他资源

- **调试本地 Docker 容器中的应用**
<https://azure.microsoft.com/en-us/documentation/articles/vs-azure-tools-docker-edit-and-refresh/>
- **Steve Lasker. 创建、调试和部署使用 Docker 的 ASP.NET Core 应用 (视频)**
<https://channel9.msdn.com/Events/Visual-Studio/Visual-Studio-2017-Launch/T115>

使用 Visual Studio 开发容器的简化流程

实际上，使用 Visual Studio 的工作流程，比使用编辑器/CLI 简单的多。和 Dockerfile 和 docker-compose.yml 文件相关的大多数 Docker 所需的步骤都被 Visual Studio 隐藏或简化了，如图 5-15 所示：

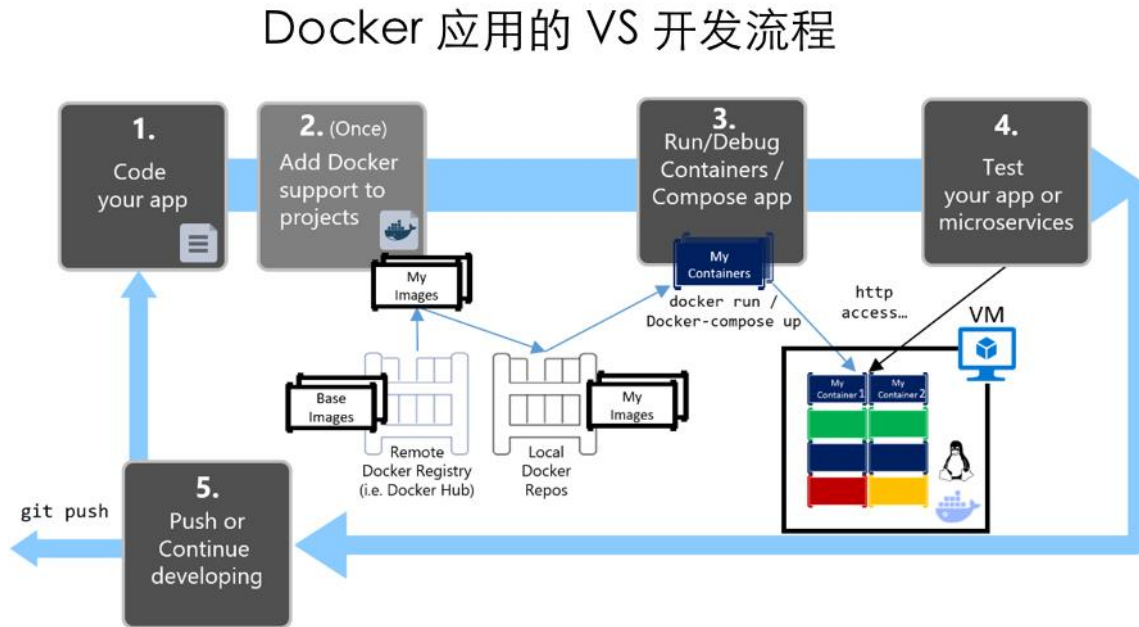


图 5-15. 使用 Visual Studio 开发的简化工作流程

您只需额外执行一次步骤 2（给您的项目添加 Docker 支持）即可。所以，这个工作流程和您平时的开发任务（用 .NET 做其他类型的开发的时候）基本类似。您需要了解在底层发生了什么（镜像构建过程，您正在使用哪种基础镜像，容器的部署等等。）因为有时候，您也需要编辑 Dockerfile 或 docker-compose.yml 文件来自定义一些行为。但是大多数工作已经被 Visual Studio 极大地简化了，这可以显著地提高您的工作效率。

其他资源

- **Steve Lasker. 使用 Visual Studio 2017 进行 .NET Docker 开发**
<https://channel9.msdn.com/Events/Visual-Studio/Visual-Studio-2017-Launch/T111>
- **Jeffrey T. Fritz. 使用 Visual Studio 的新 Docker 工具将一个 .NET Core 应用放入容器中**
<https://blogs.msdn.microsoft.com/webdev/2016/11/16/new-docker-tools-for-visual-studio/>

在 Dockerfile 中用 PowerShell 命令创建 Windows 容器

[Windows 容器](#)可以让您把现有的 Windows 应用程序转换到 Docker 镜像中，然后用 Docker 生态系统中的其他工具来部署它们。要使用 Windows 容器，您需要在 Dockerfile 中运行 PowerShell 命令，如下面这个例子所示：

```
FROM microsoft/windowsservercore
LABEL Description="IIS" Vendor="Microsoft" Version="10"
RUN powershell -Command Add-WindowsFeature Web-Server
CMD [ "ping", "localhost", "-t" ]
```

在这个例子中，我们使用了 Windows Server Core 这个基础镜像（FROM 设置），然后用一个 PowerShell 命令（RUN 设置）安装了 IIS。同理，您也可以用 PowerShell 命令添加其他的组件（例如：ASP.NET 4.x, .NET 4.6 或其他 Windows 软件）。举个例子，在 Dockerfile 中，下面这行命令会安装 ASP.NET 4.5：

```
RUN powershell add-windowsfeature web-asp-net45
```

其他资源

- **aspnet-docker/Dockerfile**. 用来包含 Windows 特性的演示 Powershell 命令的 Dockerfile <https://github.com/Microsoft/aspnet-docker/blob/master/4.6.2/Dockerfile>

在 Linux 或 Windows Nano Server 上部署单容器 .NET Core Web 应用程序

愿景

您可以用 Docker 容器实现简单的 Web 应用程序的单体部署。这可以改进持续集成和持续部署的流程，最终帮助您成功部署到生产环境。“在我的机器上完全正常，在生产环境下为什么就不行了呢？”这样的情况一去不复返了”。

基于微服务的架构有很多好处，但是这些好处同时增加了复杂性方面的成本。在某些情况下，成本比收益更重要，运行在单个容器中或很少的几个容器中的、单体部署的应用程序，可以让您节约大量的成本。

单体部署的应用程序可能很难拆分到彼此完全独立的微服务中。就像您在前面学习到的，应该通过功能来划分微服务：微服务应该是彼此独立的，以此来提供更有弹性的应用程序。如果您不能交付独立的应用程序功能块，那么划分它只会增加复杂性。

一个应用程序也许并不需要可独立扩展的功能块。我们假设，在我们的示例应用程序 eShopOnContainers 生命周期的早期，流量并不足以支撑把各种功能划分到不同的微服务中。流量很小，给一个服务添加资源通常意味着给所有服务添加资源。把这个应用程序划分到离散的服务中，性价比很低。

而且，在一个应用程序开发的早期，对于原生的功能边界到底在哪，也许您并没有一个清晰的想法。当您开发一个最小化可行产品的时候，原生的划分也许还没有形成。

在这些情况中，有些是临时性的。您可以先创建一个单体应用程序，以后再划分成若干功能，开发和部署成微服务。而另一些情况，对于应用程序的问题空间来说，可能是不可避免的，这意味着这样的应用程序可能永远不会被拆分成多个微服务。

把一个应用程序划分成多个离散的进程还会带来开销方面的问题。而把功能划分成多个不同的进程会进一步增加复杂性。通信协议也会变得更加复杂。由于无法直接进行方法调用，您必须在服务之间使用异步通信。当您迁移到微服务架构的时候，您需要添加许多在 eShopOnContainers 应用程序的微服务版本中实现的模块：事件总线处理、弹性消息和重试，最终一致性等等。

eShopOnContainers 的高度简化版本（叫 [eShopWeb](#)，也在同一个 GitHub 仓库中）是作为一个 MVC 单体应用程序来运行的。正如上面所提到的，这种设计选择有很多优势。您可以从 GitHub 上下载这个应用程序的源代码，然后在本地运行。即使是这个单体应用程序也会受益于在容器环境中部署。

首先，容器化的部署意味着这个应用程序的每一个实例都运行在同样的环境中。这其中包括了早期测试和开发所用的开发环境。开发团队可以在与生产环境完全匹配的容器化环境中运行这个应用程序。

其次，容器化应用程序的扩展成本也很低。正如前面所看到的那样，和传统的虚拟机环境相比起来，容器环境可以共享更多的资源。

最后，容器化应用程序可以促进业务逻辑服务器和存储服务器之间的分离。当应用程序扩展的时候，多个容器都依赖于同一个物理存储媒体。这通常是一个运行 SQL Server 数据库的高可用性服务器。

应用程序概览

[eShopWeb](#) 应用程序提取了 eShopOnContainers 应用程序的部分功能，作为一个单体应用程序来运行——一个运行在 .NET Core 之上，基于 ASP.NET Core MVC 的应用程序。它主要提供了我们在前几个章节中提到的目录浏览器功能。

这个应用程序使用 SQL Server 数据库来保存目录。在基于容器的部署中，这个单体应用程序可以像基于微服务的应用程序一样访问同样的数据仓库。这个应用程序被配置成将 SQL Server 运行在这个单体应用程序之外的另外一个容器中。在生产环境中，SQL Server 会运行在一台具有高可用性的机器上，也就是说，运行在 Docker 主机的外部。在开发和测试环境中，为了方便起见，我们推荐在单独的容器中运行 SQL Server。

最开始的功能只有浏览目录，后续的版本会包含容器化应用程序的所有功能。单体 Web 应用程序架构的高级特性可以参考这本电子书：《[ASP.NET Web 应用架构实战](#)》和配套的 [eShopOnWeb 示例应用程序](#)。在这个例子中，应用程序并不是运行在 Docker 上的，因为它的场景主要是用 ASP.NET Core 进行常规的 Web 开发。

而 eShopOnContainers 的简化版本 ([eShopWeb](#)) 是运行在 Docker 容器中的。

Docker 支持

eShopWeb 项目运行在 .NET Core 之上。因此，它既可以在基于 Linux 的容器中运行，也可以在基于 Windows 的容器中运行。Docker 部署的时候需要注意一下，您的应用程序应该和 SQL Server 使用同类型的主机。推荐使用基于 Linux 的容器，它更加轻量级。

Visual Studio 提供了一个项目模板，用来给解决方案添加 Docker 支持。您右键点击一个项目，点击“Add”，然后选择“Docker Support”即可。这个模板会给您的项目添加一个 Dockerfile，同时，还会添加一个新的 docker-compose 项目，它提供了一个初始的 docker-compose.yml 文件。在从 GitHub 下载回来的 eShopWeb 项目中，这一步已经完成了。您会看到这个解决方案已经包含 eShopOnWeb 项目和 docker-compose 项目了，如图 6-1 所示：



图 6-1. 在单容器 Web 应用程序中的 docker-compose 项目

这些文件是标准的 docker-compose.yml 文件，和其他 Docker 项目都是一样的。您可以通过 Visual Studio 和命令行来使用它们。这个应用程序运行在 .NET Core 上，使用 Linux 容器，所以您也可以在 Mac 或 Linux 机器上编码、生成和运行。

docker-compose.yml 文件包含了这样一些信息：要构建哪些镜像，要启动哪些容器。这个模板指定了怎样构建 eshopweb 镜像和怎样启动应用程序的容器。除此之外，您还需要添加对 SQL Server 相关的依赖，包含添加一个 SQL Server 的镜像（例如：mssql-server-linux），以及为 sql.data 镜像添加一个服务，让 Docker 来构建和启动这个容器。下面这个例子展示了这些设置：

```
version: '2'

services:
  eshopweb:
    image: eshop/web
    build:
      context: ./eShopWeb
      dockerfile: Dockerfile
    depends_on:
      - sql.data
  sql.data:
    image: microsoft/mssql-server-linux
```

depends_on 这个指令会告诉 Docker，eShopWeb 镜像依赖于 sql.data 镜像。下面几行指定用 microsoft/mssql-server-linux 镜像来构建一个标签为 sql.data 的镜像。

在 docker-compose.yml 这个主节点的下面，docker-compose 项目还包含了几个其他的 docker-compose 文件，可视化地说明这些文件是相关的。docker-compose.override.yml 文件包含所有服务的设置，例如，连接字符串和其他应用程序设置。

下面这个例子展示了 docker-compose.vs.debug.yml 文件的内容，它包含在 Visual Studio 调试会用到的一些设置。在这个文件中，eshopweb 镜像具有一个附加的 dev 标签。这可以把调试镜像和发布镜像区分开来，防止您不小心把调试信息部署到生产环境中。

```
version: '2'

services:
  eshopweb:
    image: eshop/web:dev
```

```
build:
  args:
    source: ${DOCKER_BUILD_SOURCE}
  environment:
    - DOTNET_USE_POLLING_FILE_WATCHER=1
  volumes:
    - ./eShopWeb:/app
    - ~/.nuget/packages:/root/.nuget/packages:ro
    - ~/clrdbg:/clrdbg:ro
  entrypoint: tail -f /dev/null
  labels:
    - "com.microsoft.visualstudio.targetoperatingsystem=linux"
```

添加的最后一个文件是 `docker-compose.ci.build.yml`。在 CI 服务器中，通过命令行生成这个项目的时候，会用到它。这个 compose 文件会启动一个 Docker 容器，为您的应用程序构建必需的镜像。下面这个例子展示了 `docker-compose.ci.build.yml` 文件的内容：

```
version: '2'

services:
  ci-build:
    image: microsoft/aspnetcore-build:latest
    volumes:
      - ./src
    working_dir: /src
    # The following two lines in the document are one line in the YML file.
    command: /bin/bash -c "dotnet restore ./eShopWeb.sln && dotnet publish
      ./eShopWeb.sln -c Release -o ./obj/Docker/publish"
```

请注意，这个镜像是一个 ASP.NET Core 生成镜像。那个镜像包含了 SDK 和生成工具，用来生成您的应用程序和创建您所需的镜像。运行 `docker-compose` 项目，会用这个文件来启动一个生成容器，然后在这个容器中构建您应用程序的镜像。在 Docker 容器中，您需要把这个 `docker-compose` 文件添加到生成命令中，然后执行它。

在 Visual Studio 中，您可以把 `docker-compose` 项目设置为启动项目，然后按“Ctrl+F5”（F5 用来调试），直接在一个 Docker 容器中运行您的应用程序，这和其他应用程序没有什么不同。当您启动 `docker-compose` 项目的时候，Visual Studio 会用 `docker-compose.yml` 文件，`docker-compose.override.yml` 文件和其中一个 `docker-compose.vs.*` 文件来运行 `docker-compose` 命令。如果应用程序已经启动了，Visual Studio 会直接为您启动浏览器的。

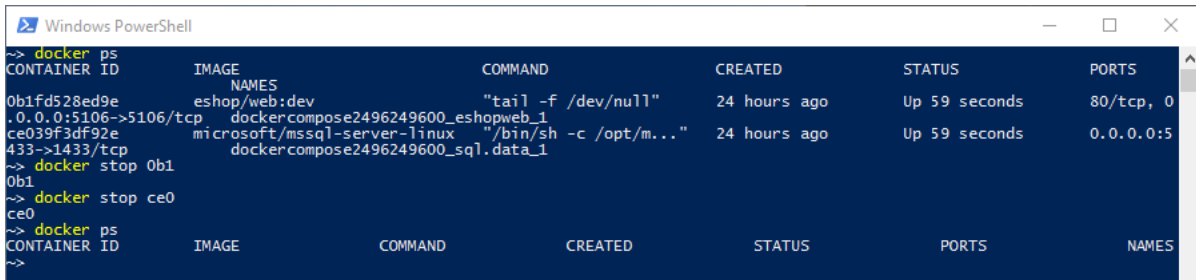
如果您在调试器中启动这个应用程序，Visual Studio 会附加到在 Docker 中运行的应用程序上。

疑难解答

本节列出了几个常见的问题，以及如何修复这些问题，您在本地运行容器的时候，可能会遇到这几个问题。

停止 Docker 容器

当您启动了容器化的应用程序以后，容器会一直运行，即使您已经停止调试了，它也还是会处于运行状态。您可以在命令行中运行 `docker ps` 命令来查看正在运行的容器。`docker stop` 命令可以停止一个正在运行的容器，如图 6-2 所示：



```
Windows PowerShell
~> docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
0b1fd528ed9e       eshop/web:dev      "tail -f /dev/null" 24 hours ago       Up 59 seconds      80/tcp, 0
.0.0.0:5106->5106/tcp dockercompose2496249600_eshopweb_1
ce039f3df92e       microsoft/mssql-server-linux "/bin/sh -c /opt/m.." 24 hours ago       Up 59 seconds      0.0.0.0:5
433->1433/tcp      dockercompose2496249600_sql.data_1
~> docker stop 0b1
0b1
~> docker stop ce0
ce0
~> docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
~>
```

图 6-2. 用 `docker ps` 和 `docker stop` CLI 命令查看和停止容器

当您在不同的配置之间来回切换的时候，您可能需要停止正在运行的进程。否则，正在运行一个 Web 应用程序的容器，会占用您应用程序的端口（在这个例子中，是 5106 端口）。

给您的项目添加 Docker 支持

添加 Docker 支持的向导会和正在运行的 Docker 进程通信。当您启动这个向导的时候，如果 Docker 还没有启动，那么这个向导是无法正常运行的。此外，这个向导还会检查您当前选择的容器，以便于可以添加正确的 Docker 支持。如果您想添加 Windows 容器支持，启动这个向导的时候，您必须有在 Windows 容器模式下运行的 Docker。如果您想添加 Linux 容器支持，启动这个向导的时候，您必须有在 Linux 容器模式下运行的 Docker。

将传统的单体 .NET Framework 应用程序迁移到 Windows 容器中

愿景

Windows 容器 (Containers) 可以用于改进开发和测试环境, 还可以用来部署基于传统的 .NET Framework 技术例如 Web Forms 的应用程序。让传统的应用程序使用容器, 这个场景被称为“平移” (lift and shift)。

本书前面几个章节对微服务架构推崇备至, 在微服务架构中, 业务应用程序会被划分到不同的容器中, 每个容器运行一个小而精的服务。这会带来很多的好处。在全新的开发中, 强烈推荐这种方法。对于企业核心业务应用程序来说, 重新架构和重新实现的投资回报率也很高。

但是, 并不是每一个应用程序这样做的投资回报率都很高。当然, 这并不意味着这些应用程序就不能在容器场景中使用。

在本章中, 我们会深入研究 eShopOnContainers 这个应用程序, 如图 7-1 所示, eShopOnContainers 企业团队的人员会使用这个应用程序来查看和编辑产品信息。

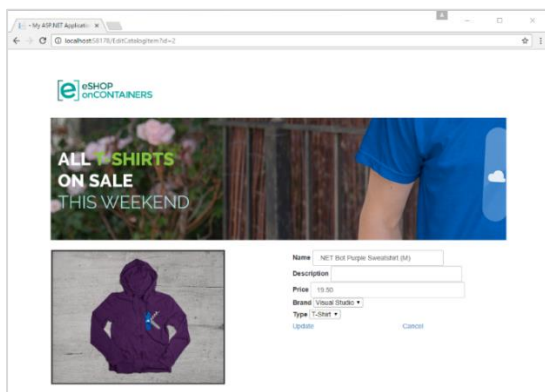


图 7-1. Windows 容器上的 ASP.NET Web Forms 应用程序 (传统的技术)

这是一个 Web Forms 应用程序, 用来浏览和修改产品信息。依赖于 Web Forms 意味着这个应用程序如果不用 ASP.NET Core MVC 代替 Web Forms 来重写, 是无法在 .NET Core 上运行的。接下来, 您会看到这样的应用程序是如何在不做任何修改的情况下直接在容器中运行的。您还会看到如何做很小的修改就能给这些应用程序引入混合模型, 在混合模型中, 有些功能会被迁移到独立的微服务中, 而大多数功能仍然保留在单体应用程序中。

容器化一个单体应用程序的好处

在 eShopOnContainers 的 GitHub 仓库 (<https://github.com/dotnet/eShopOnContainers>) 中，Catalog.WebForms 应用程序已经可以获取了。这个应用程序是一个独立的应用程序，它独立访问一个高可用的数据存储。即便如此，在容器中运行这个应用程序还是很有优势的。您首先为这个应用程序创建一个镜像，以后每一个部署都会运行在同样的环境中。每个容器都使用同样的 OS 版本，安装的依赖的版本也完全一样，它们都使用同样的框架，而且它们都是使用同样的流程生成的。在图 7-2 中，您可以看到在 Visual Studio 2017 中打开的这个应用程序。

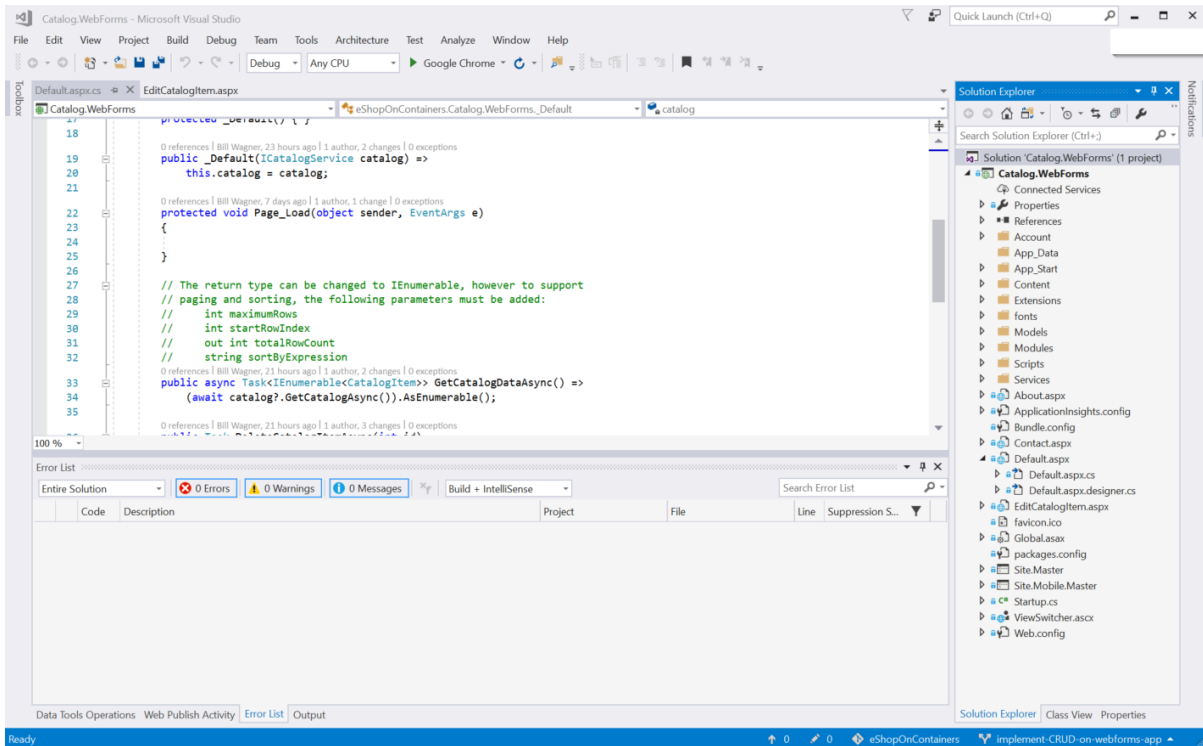


图 7-2. Visual Studio 2017 中的产品管理 Web Forms 应用程序

另外，所有开发人员都可以在完全一致的环境中运行这个应用程序。只在某个版本上才出现的问题，所有开发人员都会马上发现，而不是等到预发布环境或生产环境下才发现。如果应用程序运行在容器中，那么开发团队之间开发环境的差异已经无关紧要了。

最后，容器化应用程序的扩展曲线很平滑。在前面几章，您已经学习过如何在一台虚拟机或物理机上启用更多的容器了。这意味着更高的密度和更少的资源占用。

基于上述这些理由，您可以考虑做一个“平移”操作，让传统的单体应用程序直接在容器中运行。“平移”这个短语描述了这个任务的范围：您从一台物理机或一台虚拟机“提起”整个应用程序，然后把它“移动”到一个容器中。在理想情况下，您无需修改任何应用程序代码，就可以让它直接在容器中运行了。

可能的迁移路径

作为一个单体应用程序，Catalog.Webforms 是一个包含了所有代码的 Web 应用程序，其中包括了一些数据访问库。数据库运行在一台独立的具有高可用性的机器上。在样例代码中，通过一个仿真的产品服务来模拟这个配置：您可以用仿真的数据运行 Catalog.WebForms 应用程序，来模拟单纯的“平移”场景。这描述了一个最简单的迁移路径，在这个迁移路径中，您只需把现有的东西移动到容器中运行即可，无需修改任何代码。此路径适用于已完成且与您要移动到 microservices 的功能仅有最小交互的应用程序。

然而，eShopOnContainers 网站已经为不同场景实现了通过微服务访问数据存储的方式。所以您可以对目录编辑器做一些额外的小修改让它使用目录微服务，而不是直接访问产品的数据存储。

这些修改展示了如何使您的应用程序保持连续性。您有很多种迁移路径可以选择：在不做任何修改的情况下把现有的应用程序迁移到容器中；做很小的修改让现有的应用程序可以访问一些新的微服务；用全新的基于微服务的架构完全重写整个应用程序；哪种迁移路径更适合您，取决于迁移的投资回报率。

应用程序概览

您可以在 Visual Studio 中打开 Catalog.WebForms 解决方案，把这个应用程序作为一个独立的应用程序来运行。在当前配置中，这个应用程序使用一个仿真的服务来返回数据，而不是一个持久化存储的数据库。这个应用程序用 Autofac (<https://autofac.org/>) 来做反转控制 (IoC)，通过依赖注入 (DI)，您可以配置这个应用程序是使用仿真数据还是使用真实的产品数据服务。（关于 DI，我们马上会做进一步的说明。）启动代码会从 web.config 文件中读取 useFake 设置的值，然后根据这个设置的值来配置 Autofac 容器是注入仿真的数据服务还是真实的产品服务。如果您运行的这个应用程序，在 web.config 文件中把 useFake 设置为 false，那么您会看到这个 Web Forms 应用程序显示的是产品数据。

在这个应用程序中使用的绝大多数技术，对于使用 Web Forms 的人来说应该都比较熟悉。但是，对目录微服务引入的两个技术，大家可能就不太熟悉了：依赖注入 (DI)，刚刚也提到了这门技术，在 Web Forms 中它和异步的数据存储一起协同工作。

DI 颠覆了传统的面向对象的策略：在编写类的时候，分配所有必需的资源。取而代之的是，类从一个服务容器请求它们的依赖。DI 的优势是，您可以用仿真（或模拟）的服务代替外部的服务来支持测试或其他的环境。

DI 容器用 web.config 中的 appSettings 配置来控制是使用仿真的产品数据还是使用来自于运行中的服务的真实数据。这个应用程序注册了一个 HttpModule 对象来生成容器，同时还注册了一个预请求处理程序来注入依赖。在 Modules/AutoFacHttpModule.cs 文件中，您可以看到这些代码，如下面的例子所示：

```
private static IContainer CreateContainer()
```

```

{
    // Configure AutoFac:
    // Register Containers:
    var settings = WebConfigurationManager.AppSettings;
    var useFake = settings["usefake"];
    bool fake = useFake == "true";
    var builder = new ContainerBuilder();
    if (fake)
    {
        builder.RegisterType<CatalogMockService>()
            .As<ICatalogService>();
    }
    else
    {
        builder.RegisterType<CatalogService>()
            .As<ICatalogService>();

        builder.RegisterType<RequestProvider>()
            .As<IRequestProvider>();
    }
    var container = builder.Build();
    return container;
}

private void InjectDependencies()
{
    if (HttpContext.Current.CurrentHandler is Page page)
    {
        // Get the code-behind class that we may have written
        var pageType = page.GetType().BaseType;

        // Determine if there is a constructor to inject, and grab it
        var ctor = (from c in pageType.GetConstructors()
                    where c.GetParameters().Length > 0
                    select c).FirstOrDefault();

        if (ctor != null)
        {
            // Resolve the parameters for the constructor
            var args = (from parm in ctor.GetParameters()
                       select Container.Resolve(parm.ParameterType))
                .ToArray();

            // Execute the constructor method with the arguments resolved
            ctor.Invoke(page, args);
        }

        // Use the Autofac method to inject any
        // properties that can be filled by Autofac
        Container.InjectProperties(page);
    }
}
}

```

这个应用程序的页面 (Default.aspx.cs 和 EditPage.aspx.cs) 定义了一些构造器来获取这些依赖。注意，默认的构造器一直是可用的。这个基础构造的代码如下：


```
protected _Default() { }

public _Default(ICatalogService catalog) =>
    this.catalog = catalog;
```

产品 API 都是异步的方法。Web Forms 现在所有的数据控制方法都支持异步。Catalog.WebForms 应用程序的列表页和编辑器使用模型绑定，页面上的控件定义了 SelectMethod、UpdateMethod、InsertMethod 和 DeleteMethod 属性，指定了任务返回时的异步操作。Web Forms 控件知道绑定到控件上的方法是否是异步的。当您使用异步的查询方法时遇到的唯一限制是不支持分页。分页标记需要一个 out 参数，而异步方法没有 out 参数。需要从产品服务获取数据的其他页面也是使用同样的技术。

目录 Web Form 应用程序的默认配置使用 catalog.api 服务的一个模拟实现。这个模拟实现的数据使用一个硬编码的数据集，通过移除在开发环境中与 catalog.api 服务相关的依赖，它简化了一些任务。

平移

Visual Studio 为容器化一个应用程序提供了很好的支持。您只需右击项目节点，然后选择“添加”和“Docker 支持”就可以了。Docker 项目模板会给您的解决方案添加一个叫“docker-compose”的项目。这个项目包含一些 Docker 资源，用于组合（或构建）您所需的镜像，然后开始运行必要的容器，如图 7-3 所示。

在这个最简单的直接迁移场景中，对于这个 Web Forms 应用程序来说就是一个单独的服务。这个模板还会把您的启动项目设置为“docker-compose”项目。现在按“Ctrl+F5”或“F5”会创建一个 Docker 镜像，然后启动一个容器。

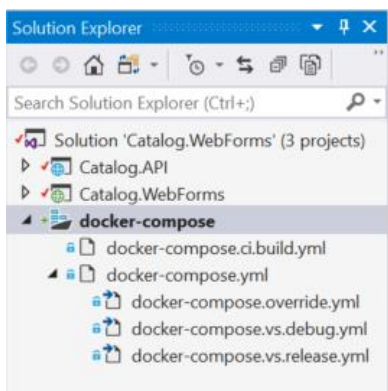


图 7-3. Web Forms 解决方案中“docker-compose”项目

在运行这个解决方案之前，您必须确保您已经把 Docker 配置成使用 Windows 容器了。要实现这个操作，您可以在 Windows 中右击 Docker 任务栏图标，然后选择“切换到 Windows 容器”，如图 7-4 所示：

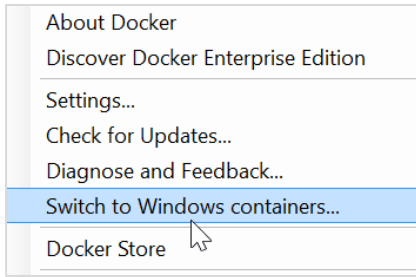


图 7-4. 在 Windows 中, Docker 任务栏图标中的“切换到 Windows 容器”菜单项

如果这个菜单项显示的是“切换到 Linux 容器”,说明 Docker 已经运行在 Windows 容器模式下了。

运行这个解决方案会重启 Docker 主机。当您做生成操作的时候,您会为这个 Web Forms 项目生成的一个应用程序和一个 Docker 镜像。您第一次做这个操作的时候,会花费比较长的时间。这是因为生成过程会拉取基础的 Windows Server 镜像和辅助的 ASP.NET 镜像。以后的生成和运行周期会快得多。

让我们来深入地研究一下 Docker 项目模板添加的文件。它帮您创建了几个文件。Visual Studio 用这些文件来创建 Docker 镜像和启动容器。在 CLI 中手动运行 Docker 命令的时候,您也可以使用这些文件。

下面这个 Dockerfile 的例子,展示了构建一个 Docker 镜像的基础设置,这个镜像是基于 Windows ASP.NET 构建的,在其中承载了一个 ASP.NET 的网站:

```
FROM microsoft/aspnet
ARG source
WORKDIR /inetpub/wwwroot
COPY ${source:-obj/Docker/publish} .
```

这个 Dockerfile 文件初看起来和那些在 Linux 容器中运行 ASP.NET Core 应用程序的镜像基本类似。但是,还是有几个重要的区别。最重要的区别是基础镜像是 microsoft/aspnet,这是最新的 Windows Server 镜像,它包含了 .NET Framework。另外一个区别是从源目录复制到的目标目录不同。

在 docker-compose 项目中的其他文件是一些用来是构建和配置容器必需的 Docker 资源。Visual Studio 会把多个 docker-compose.yml 文件放在一个节点下面来突出他们的作用。基础的 docker-compose 文件包含针对所有配置的通用指令。docker-compose.override.yml 文件包含环境变量和可以重写的开发人员配置。.vs.debug 和 .vs.release 两个变体包含一些环境设置,这些设置可以让 Visual Studio 附加到正在运行的容器中,并且管理这个正在运行的容器。

Visual Studio integration 也是给您的解决方案添加 docker 支持的一部分,与此同时您还可以在命令行中用 docker-compose up 命令生成和运行这个应用程序,正如您在前面几个小节中看到的那样。

从现有的 .NET Core 目录微服务获取数据

您可以把这个 Web Forms 应用程序配置成使用 eShopOnContainers 目录微服务来获取数据，来代替使用仿真数据。要完成这个操作，您可以编辑 web.config 文件，把 useFake 的值设置为“false”。在这种情况下，DI 容器会使用访问真实目录微服务的类，来代替返回硬编码数据的类。其他代码无需做任何修改。

访问真实的产品服务意味着您需要更新 docker-compose 项目，以便构建产品服务镜像和启动产品服务容器。Windows 中的 Docker 社区版支持 Linux 容器和 Windows 容器，但是无法同时运行 Linux 容器和 Windows 容器。要运行这个微服务，您需要构建一个镜像，在基于 Windows 的容器上运行这个目录微服务。比起您在前面几个小节中看到的，这个微服务项目需要一个不同的 Dockerfile，Dockerfile.windows 文件包含用来构建产品 API 容器镜像所需的配置设置，让它可以在 Windows 容器上运行——例如：使用一个 Windows Nano Docker 镜像。

这个产品服务依赖于 SQL Server 数据库。所以，您还需要使用一个基于 Windows 的 SQL Server Docker 镜像。

做了这些变更以后，docker-compose 项目也需要做一些额外的工作才能启动这个应用程序。这个项目现在需要用基于 Windows 的 SQL Server 镜像来启动 SQL Server。它会在一个 Windows 容器中，启动这个目录微服务。并且，它还需要启动基于 Web Forms 的产品编辑器容器，当然，也是一个 Windows 容器。如果有哪个镜像需要构建，会首先创建这个镜像。

开发环境和生产环境

开发环境的配置和生产环境的配置还是有一些区别的。在开发环境中，您会在多个 Windows 容器中分别运行 Web Forms 应用程序，目录微服务和 SQL Server，但是这些容器运行在同一台 Docker 主机上的。在前面几个小节中，我们提到了 SQL Server 镜像和其他基于 .NET Core 的服务都被部署到了同一台基于 Linux 的 Docker 主机上。在同一个 Docker 主机（或集群）上运行多个微服务的优势是：网络通信较少，而且容器之间的通信延迟更低。

在开发环境中，您必须在同一个 OS 上运行所有的容器。Windows 中的 Docker 社区版并不支持同时运行 Windows 容器和 Linux 容器。在生产环境下，您可以自己决定是想在同一台 Docker 主机上的 Windows 容器中运行目录微服务，还是想让 Web Forms 应用程序和运行在不同 Docker 主机上的 Linux 容器中的目录微服务的一个实例之间彼此通信。这取决于您希望怎样来优化网络延迟。在大多数情况下，您肯定希望您的应用程序依赖的微服务运行在同一台 Docker 主机（或 swarm）上，这样可以使开发工作变得更容易一些，网络延迟也更低一些。在这样的配置中，微服务实例与用于持久化数据存储的高可用服务器之间的通信成本是唯一的通信成本。

设计和开发多容器和基于微服务的.NET 应用程序

愿景

开发容器式微服务应用程序意味着您正在构建多容器应用程序。然而，多容器应用程序也可能更简单。例如，三层应用程序可能不使用微服务体系结构构建。

此前我们提出了一个问题：“构建微服务架构时 Docker 是否是必需的？”，答案很明显，并不是。Docker 可以提供显著的优势，但容器和 Docker 并不是微服务的必要条件。例如，当使用 Azure Service Fabric（支持微服务作为简单进程或作为 Docker 容器运行）时，您可以使用或不使用 Docker 创建基于微服务器的应用程序。

但是，如果您知道如何设计和开发基于微服务同时也基于 Docker 容器的应用程序，那么您将能够设计和开发任何其他更简单的应用程序模型。例如您可以设计一个也需要多容器技术的三层应用程序。因此，由于微服务架构是容器世界的重要趋势，本节重点介绍使用 Docker 容器的微服务体系结构实现。

设计面向微服务的应用程序

本节着重介绍开发一个服务器端企业应用示例程序。

应用规范

该应用程序通过执行业务逻辑，访问数据库，然后返回 HTML，JSON 或 XML 响应来处理请求。应用程序支持各种客户端，包括运行单页应用程序（SPA）的桌面浏览器，传统 Web 应用程序、移动 Web 应用程序和原生移动应用程序。该应用程序也可能会暴露一个 API 供第三方使用。它还应该能够异步地集成其微服务或外部应用程序，以便在部分故障的情况下，该方法将有助于微服务器的恢复。

应用程序将由以下类型的组件组成：

- 展示组件。这些负责处理 UI 和调用远程服务。
- 应用程序业务逻辑。
- 数据库访问逻辑。这包括负责访问数据库（SQL 或 NoSQL）的数据访问组件。
- 应用集成逻辑。这包括一个消息通道，主要基于消息代理。

该应用程序将需要高可扩展性，同时允许其纵向的子系统自动扩展，因为某些子系统将需要比其他子系统更多的可扩展性。

应用程序必须能够部署在多种基础设施环境（多种公有云和私有云）中，并且最好应该是跨平台的，能够轻松地从 Linux 迁移到 Windows（或反之亦然）。

开发团队环境

我们还假设应用程序的开发过程如下：

- 您有多个开发团队专注于应用程序的不同业务领域。
- 新团队成员必须快速成长，应用程序必须易于理解和修改。
- 应用程序将具有长期演进和不断变化的业务规则。
- 您需要良好的长期可维护性，这意味着在未来实现新变化时具有敏捷性，同时能够在对其他子系统最小影响下更新多个子系统。
- 您希望实践应用程序的持续集成和持续部署。
- 您在开发应用程序时，希望利用新兴技术（框架，编程语言等）的优势。您不想在迁移到新技术时完全迁移应用程序，因为这将导致高成本并影响应用程序的可预测性和稳定性。

选择一个架构

应用程序部署架构应该是什么？根据应用程序的规范以及开发环境，您应该通过将应用程序分解为独立的子系统，以协作微服务和容器的形式构建应用程序，其中一个微服务是一个容器。

在这种方法中，每个服务（容器）实现一组高内聚的功能。例如，应用程序可能包括目录服务，订单服务，购物篮服务，用户配置文件服务等服务。

微服务使用诸如 HTTP（REST）之类的协议进行通信，但是尽可能地使用异步（例如，使用 AMQP），特别是在利用集成事件传播更新时。

微服务开发和部署为彼此独立的容器。这意味着开发团队可以在不影响其他子系统的情况下开发和部署某个微服务。

每个微服务都有自己的数据库，允许它与其他微服务完全解耦。必要时，使用应用程序级集成事件（通过逻辑事件总线），如命令查询职责分离（CQRS），实现来自不同微服务器的数据库之间的一致性。因此，业务约束必须包含多个微服务器和相关数据库之间最终的一致性。

eShopOnContainers: 使用容器部署的 .NET Core 和微服务的示例应用程序

为了方便，您可以专注于架构和技术，而无需考虑不熟悉的假设的业务领域，我们选择了一个众所周知的业务领域-即一个简化的电子商务（e-shop）应用程序，提供一个产品目录，客户订单，验证库存，以及执行其他业务功能。这个基于容器的应用程序源代码可在 [eShopOnContainers](#) 的 GitHub 仓库中下载。

该应用程序由多个子系统组成，包括几个商店的 UI 前端（一个 Web 应用程序和一个原生移动应用程序），以及后端微服务器和容器，用于所有必需的服务器端操作。示例应用的架构如图 8-1 所示。

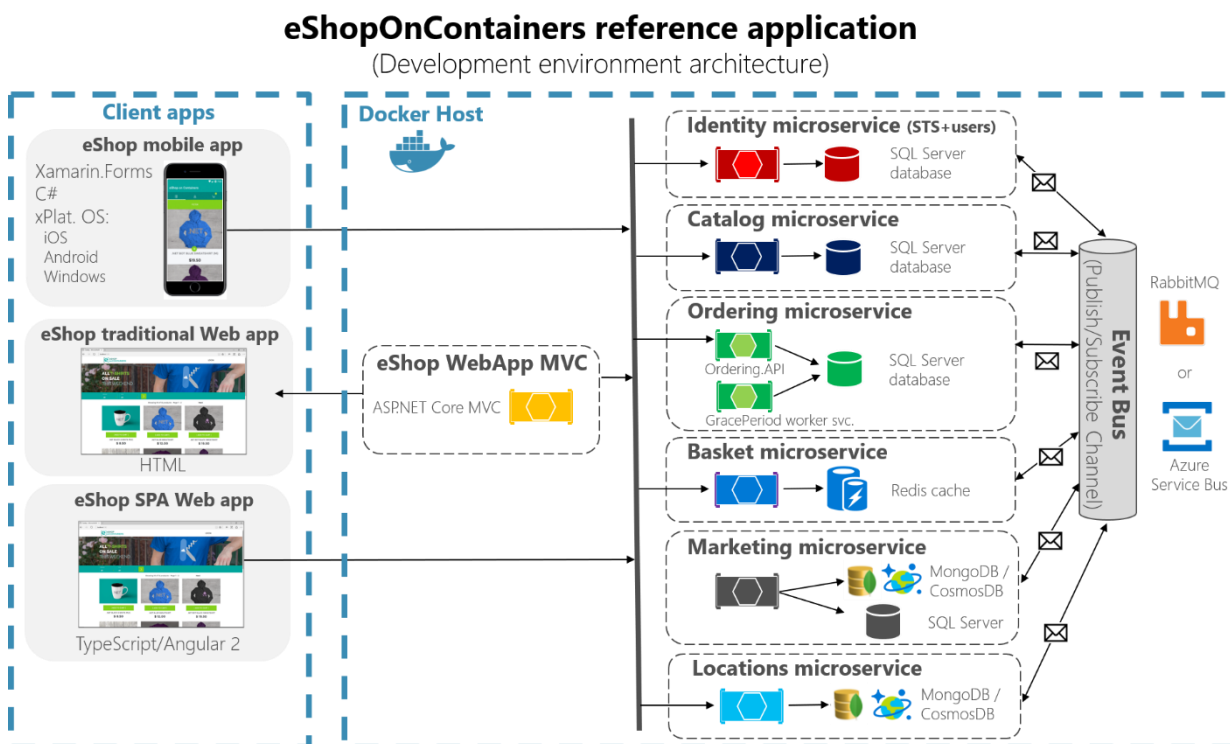


图 8-1. 针对开发环境的 eShopOnContainers 示例应用程序架构

托管环境。 在图 8-1 中，您可以看到在单个 Docker 主机中部署的几个容器。当使用 docker-compose up 命令部署到单个 Docker 主机时就是这种情况。但是，如果您正在使用编排引擎或容器集群，则每个容器可能在不同的主机（节点）中运行，并且任何节点都可以运行任意数量的容器，正如我们前面在架构部分中所述。

通信架构。 eShopOnContainers 应用程序使用两种通信类型，具体取决于功能操作的类型（查询与更新和事务）：

- 直接的客户端与微服务的通信。这用于查询和接受来自客户端应用程序的更新或事务命令。

- 异步事件通信。这通过事件总线来传播来自微服务的更新或与外部应用程序集成。事件总线可以使用任何消息代理架构技术（如 RabbitMQ）来实现，也可以使用诸如 Azure 服务总线（Service Bus）、NServiceBus、MassTransit 或 Brighter 之类的更高级别的服务总线。

该应用程序以容器的形式部署为一组微服务。客户端应用程序可以与这些容器进行通信，也可以在微服务之间进行通信。如前所述，这种初始架构正在使用客户端到微服务的直接通信架构，这意味着客户端应用程序可以直接向每个微服务器发出请求。每个微服务都有一个公共终结点，如 `https://servicename.applicationname.companyname`。如果需要，每个微服务器可以使用不同的 TCP 端口。在生产模式中，该 URL 将映射到微服务器的负载均衡器，该负载均衡器将请求分发到可用的微服务实例。

有关 eShopOnContainers 中 API 网关与直接通信的重要说明：如本指南的架构部分所述，当您构建基于大型和复杂的基于微服务器的应用程序时，直接的客户端到微服务通信架构可能有一些缺点。但是对于一个小型应用程序来说，如 eShopOnContainers 应用程序，其目标重点关注一个更简单的基于容器的入门 Docker 应用程序，我们并不想创建一个能够影响微服务自动部署的单体 API 网关。

但是，如果您要设计一个具有数十个微服务器的大型基于微服务的应用程序，我们强烈建议您考虑 API 网关模式，正如我们在架构部分中所解释的那样。

一旦考虑到为远程客户端准备适合生产环境的应用程序和特制的外观模式，这个架构决策就可以被重构。根据客户端应用程序的外观因素，拥有多个自定义 API 网关可以为每个客户端应用程序提供不同的数据聚合功能，并且可以隐藏针对客户端应用程序的内部微服务或 API，并在该单一层中进行授权。但是，正如前面提到的那样，要小心处理大型的单体 API 网关，有可能会影响微服务的自动部署性不可用。

每个微服务的数据自治

在示例应用程序中，每个微服务拥有自己的数据库或数据源，并将每个数据库或数据源部署为另一个容器。这种设计决策只是为了让开发人员能够轻松获取 GitHub 的代码，克隆它，并在 Visual Studio 或 Visual Studio Code 中打开它。或者，它可以使用 .NET Core CLI 和 Docker CLI 编译定制 Docker 镜像，然后在 Docker 开发环境中部署和运行它们。无论哪种方式，使用数据源容器都可以让开发人员在几分钟内构建和部署，而无需为外部数据库或任何其他基础设施（云端或本地）依赖的其他数据源提供配置。

在真实的生产环境中，为了实现高可用性和可扩展性，数据库应基于云端数据库服务器或本地数据库服务器，而不是容器。

因此，微服务部署（甚至本应用程序中的数据库）的单位是 Docker 容器，示例应用程序是采用微服务原则的多容器应用程序。

其他资源

- **eShopOnContainers GitHub 仓库。** 示例应用程序的源代码
<https://aka.ms/eShopOnContainers/>

基于微服务的解决方案的优点

这样基于微服务的解决方案有很多优点：

每个微服务相对较小 - 易于管理和改进。 特别是：

- 开发人员很容易理解并快速开始，并且生产率很高。
- 容器启动迅速，这使开发人员效率更高。
- 像 Visual Studio 这样的 IDE 可以快速加载较小的项目，使开发人员更有效率。
- 每个微服务都可以独立于其他微服务进行设计，开发和部署，提供了敏捷性，因为部署微服务的新版本更容易。

可以扩展应用程序的各个区域。 例如，目录服务或购物篮服务可能需要扩展，但订购过程不需要。对于使用的资源而言，微服务架构将比单体架构更为有效。

您可以将多个团队之间的开发工作分开。 每个服务都可以由单独的开发团队负责。每个团队都可以独立于其他团队来管理，开发，部署和扩展他们的服务。

问题更加隔离。 如果一个服务中存在问题，则只有该服务会受到影响（除非使用了错误的设计，而且在微服务之间存在直接依赖关系），而其他服务可以继续处理请求。相比之下，单体部署架构中的一个故障组件可能会导致整个系统崩溃，特别是当它涉及资源（如内存泄漏）时。另外，当微服务中的问题得到解决时，您可以仅部署受影响的微服务，而不影响应用程序的其余部分。

您可以使用最新的技术。 您可以独立开发服务并且并行地运行它们（得益于容器和 .NET Core），您可以方便地开始使用最新的技术和框架，而不是停留在整个应用程序的旧技术栈或框架上。

基于微服务的解决方案的缺点

这样基于微服务的解决方案也有一些缺点：

分布式应用。 在开发人员设计和构建服务时，分发应用程序增加了复杂性。例如，开发人员必须使用诸如 HTTP 或 AMQP 之类的协议实现内部服务通信，这增加了测试和异常处理的复杂性。它还增加了系统的延迟。

部署的复杂性。 具有数十个微服务并需要高可扩展性的应用程序（需要能够为每个服务创建许多实例并在多个主机之间平衡这些服务）意味着 IT 操作和管理的高度部署复杂性。如果您不使用面向微服务的基础架构（如编排引擎和调度器），那么额外的复杂性可能需要比业务应用程序本身更多的开发工作。

原子事务。 多个微服务之间的原子事务通常是不可能的。业务需求必须包含多个微服务之间的最终一致性。

增加全局资源需求（所有服务器或主机的总内存，驱动器和网络资源）。在许多情况下，当您使用微服务方法替换单体应用程序时，新的基于微服务的应用程序所需的全局资源数量将大于原始单体应用程序的基础架构需求。这是因为更高的颗粒度和分布式服务需要更多的全局资源。然而，考虑到一般来说资源成本较低，而且在单体应用演进过程中，与长期成本相比，能够将应用程序在特定领域的能力扩大的优势，因此增加资源的使用对大规模、长期的应用程序来说，通常是一个很好的权衡。

客户端到微服务的直连通信问题。 当应用程序很大，有许多微服务时，如果应用程序需要直接的客户端到微服务通信则存在挑战和限制。一个问题是客户端的需求和每个微服务暴露的 API 之间可能不匹配。在某些情况下，客户端应用程序可能需要许多单独的请求来组成用户界面，这在互联网上可能是低效的，在移动网络上更是不切实际的。因此，客户端应用程序对后端系统的请求应该最小化。

直接的客户端到微服务通信的另一个问题是，一些微服务可能使用非标准的 Web 协议。一个服务可能使用二进制协议，而另一个服务可能使用 AMQP 消息。这些协议通常会被防火墙屏蔽，最好在内部使用。通常，应用程序应该使用 HTTP 和 WebSockets 等协议来进行防火墙外的通信。

这种直接的客户端到服务方式的另一个缺点是：难以重构。随着时间的推移，开发人员可能会想修改系统如何划分为服务。例如，他们可能会合并两个服务或将一个服务拆分为两个或更多服务。但是，如果客户端直接与服务进行通信，则执行此类重构可能会破坏与客户端应用程序的兼容性。

如架构部分所述，在基于微服务设计和构建复杂应用程序时，您可能会考虑使用多个细粒度的 API 网关，而不是使用更简单的直接客户端到微服务的通信方法。

分割微服务。 最后，无论您为微服务架构采取哪种方法，另一个挑战是决定如何将端到端应用程序分割成多个微服务。如本指南的架构部分所述，您可以使用几种技术和方法。基本上，您需要确定应用程序与其他区域解耦并具有低数量硬依赖的区域。在许多情况下，这与用例中的分区服务一致。例如，在我们的电子商店应用中，我们有一个订单服务，负责与订单流程相关的所有业务逻辑。我们还拥有实现其他功能的目录服务和购物篮服务。理想情况下，每项服务应该只有一小部分职责。这与应用于类的单一职责原则（SRP）类似，它指出一个类要更改时，只应该有一个原因。但在这种情况下，它是关于微服务的，所以范围将大于单个类。最重要的是，微服务必须是完全自主的，端到端，包括对自己数据源的职责。

外部与内部架构和设计模式

外部架构是由多个服务组成的微服务架构，遵循本指南基础架构部分所述的原则。然而，根据每个微服务的性质，并且独立于您选择的高级微服务体系结构，对于不同的微服务，具有不同的内部架构（每种都基于不同的模式）是常见的，有时候是更合适的。微服务甚至可以使用不同的技术和编程语言。

图 8-2 说明了这种多样性。

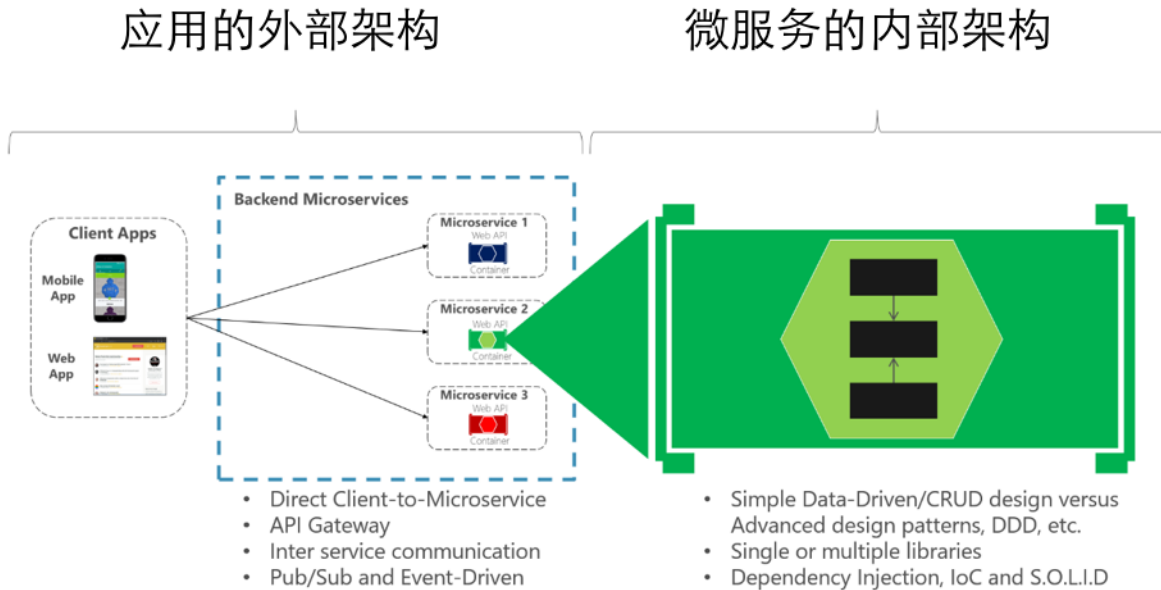


图 8-2. 外部与内部架构和设计

例如，在我们的 eShopOnContainers 示例中，目录，购物篮和用户配置文件的微服务很简单（基本上是 CRUD 子系统）。因此，他们的内部架构和设计很简单。但是，您可能还有其他微服务器，如订单微服务，它更为复杂，代表了不断变化的业务规则，具有高度的领域复杂性。在这样的情况下，您可能希望在特定的微服务中实现更高级的模式，例如使用领域驱动设计（DDD）定义的模式，正如我们在 eShopOnContainers 订单微服务中所做的那样。（我们将在稍后的部分中回顾这些 DDD 模式，以解释 eShopOnContainers 订单微服务的实现。）

每个微服务采用不同技术的另一个原因可能是每个微服务的性质。例如，如果您针对 AI 和机器学习领域，使用像 F# 这样的函数式编程语言，或者甚至像 R 这样的语言可能会更好一些，而不是像 C# 这样更为面向对象的编程语言。

每个微服务可以基于不同的设计模式具有不同的内部架构。并不是所有的微服务都应该使用先进的 DDD 模式来实现，因为这会导致过度设计。类似地，具有不断变化的业务逻辑的复杂微服务不应该被实现为 CRUD 组件，否则你会得到低质量的代码。

新世界：多种架构模式和多语言微服务

有许多软件架构师和开发人员使用的架构模式。以下是一些模式（混合架构风格和架构模式）：

- 简单的 CRUD，单层。
- [传统 N 层](#)。
- [领域驱动设计 N 层](#)。
- [干净架构](#)（eShopOnWeb 使用的）
- [命令和查询职责分离](#)（CQRS）。
- [事件驱动架构](#)（EDA）。

您还可以使用许多技术和语言构建微服务，例如 ASP.NET Core Web API、NancyFx、ASP.NET Core SignalR（可用于 .NET Core 2）、F#、Node.js、Python、Java、C++、GoLang 或更多。

重要的一点是，没有特定的架构模式或风格，也没有任何特定的技术，能够适用于所有情况。图 8-3 显示了可用于不同微服务的一些方法和技术（尽管不是特定的顺序）。

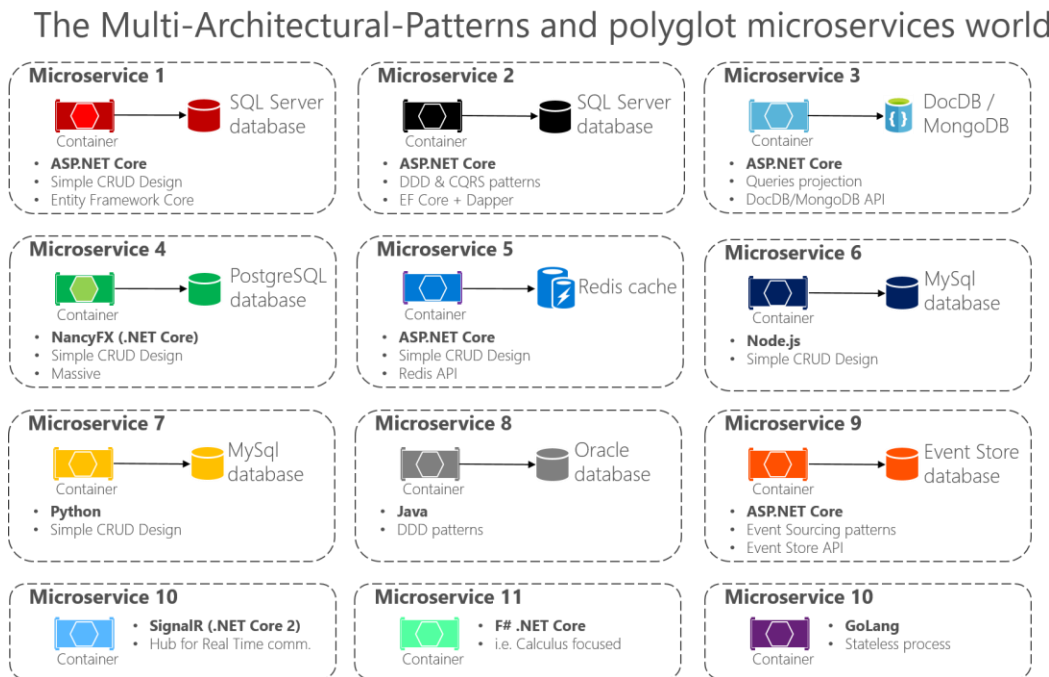


图 8-3. 多种架构模式和多语言的微服务世界

如图 8-3 所示，在由许多微服务（领域驱动设计术语中的边界上下文，或作为自主微服务的简单“子系统”）组成的应用程序中，您可以以不同的方式实现每个微服务。每个可能具有不同的架构模式，并

根据应用程序的性质，业务需求和优先级使用不同的语言和数据库。在某些情况下，微服务可能是相似的。但是通常情况并非如此，因为每个子系统的上下文边界和要求通常是不同的。

例如，对于简单的 CRUD 维护应用程序，设计和实现 DDD 模式可能没有什么意义。但是，对于您的核心领域或核心业务，您可能需要应用更先进的模式来应对业务规则不断变化的业务复杂性。

特别是当您处理由多个子系统组成的大型应用程序时，您不应该基于单个架构模式应用单个顶级架构。例如，CQRS 不能作为整个应用程序的顶级架构应用，但可能对特定的服务集有用。

对每个给定的情况来说，并不存在银弹或正确的架构模式。您不可能用“一种架构模式来解决所有问题”。根据每个微服务器的优先级，您必须为每个微服务选择不同的方法，如以下部分所述。

创建一个简单的数据驱动的 CRUD 微服务

本节概述了如何创建一个简单微服务，在数据源上执行创建，读取，更新和删除（CRUD）操作。

设计一个简单的 CRUD 微服务

从设计的角度来看，这种容器化的微服务非常简单。也许要解决的问题很简单，或者说该实现方法只是为了证明一个概念。

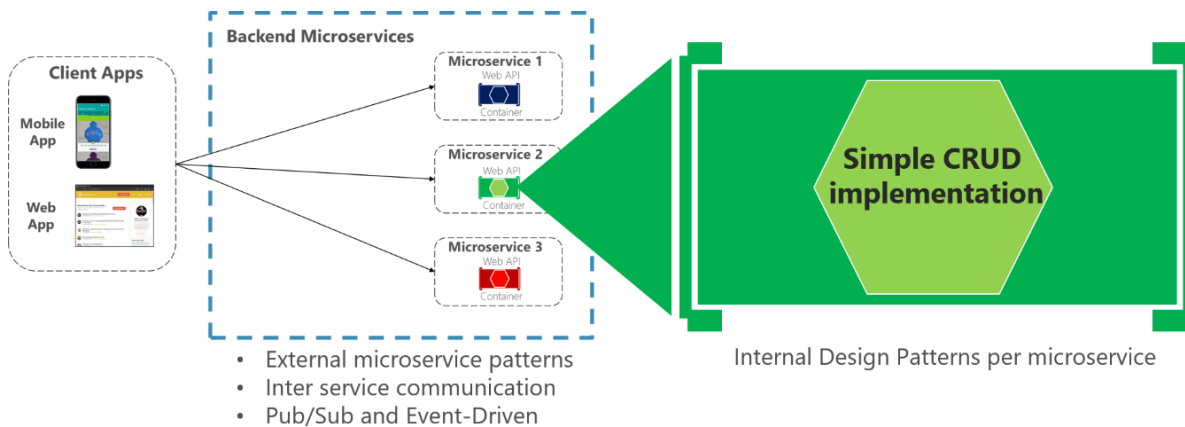


图 8-4. 简单的 CRUD 微服务的内部设计

这种简单数据驱动服务的一个例子是 eShopOnContainers 示例应用程序的目录微服务。这种类型的服务在单个 ASP.NET Core Web API 项目中实现其所有功能，该项目包括其数据模型类，业务逻辑类及其数据访问代码类。它还将相关数据存储在与运行在 SQL Server（作为另一个用于开发/测试目的的容器）的数据库中，但也可以是任何常规的 SQL Server 主机，如图 8-5 所示。

数据驱动/CRUD微服务容器

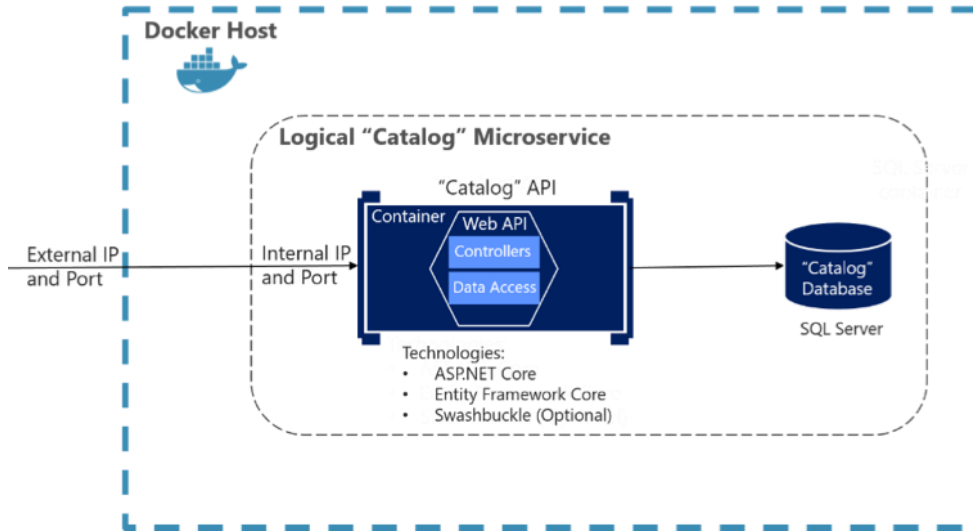


图 8-5. 简单的数据驱动/CRUD 微服务设计

当您开发此类服务时，您只需要 [ASP.NET Core](#) 和数据访问 API 或 ORM 如 [Entity Framework Core](#)。您还可以通过 [Swashbuckle](#) 自动生成 [Swagger](#) 元数据，为您的服务提供说明，如下一节所述。

请注意，在 Docker 容器内运行数据库服务器（如 SQL Server）对于开发环境是非常好的，因为您可以使所有的依赖项启动并运行，而无需在云或本地部署数据库。这在运行集成测试时非常方便。但是，对于生产环境，不建议在容器中运行数据库服务器，因为这种方法通常不具有高可用性。对于 Azure 中的生产环境，建议您使用 Azure SQL DB 或可提供高可用性和高可扩展性的任何其他数据库技术。例如，对于 NoSQL 路线来说，您可以选择 DocumentDB。

最后，通过编辑 Dockerfile 和 docker-compose.yml 元数据文件，您可以配置如何创建该容器的镜像-将要使用的基本镜像以及设计设置，如内部和外部名称以及 TCP 端口等。

使用 ASP.NET Core 实现一个简单的 CRUD 微服务

要使用 .NET Core 和 Visual Studio 实现一个简单的 CRUD 微服务，您首先要创建一个简单的 ASP.NET Core Web API 项目（在 .NET Core 上运行，以便可以在 Linux Docker 主机上运行），如图 8-6。

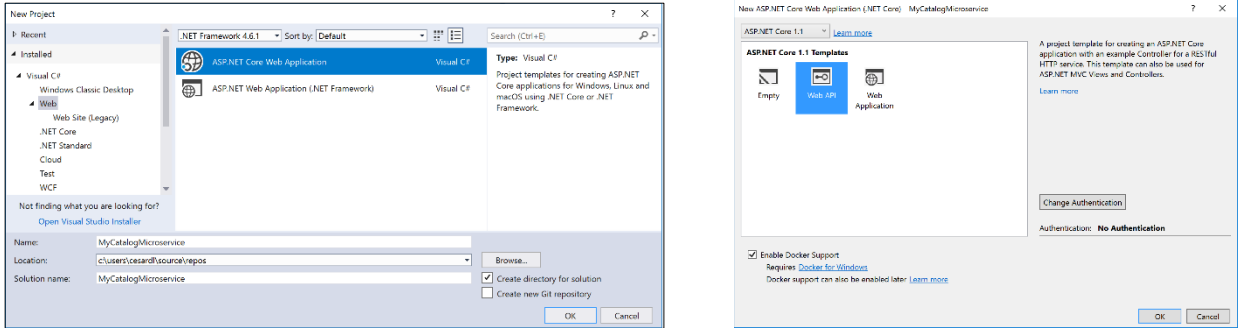


图 8-6. 在 Visual Studio 中创建一个 ASP.NET Core Web API 项目

创建项目后，可以像任何其他 Web API 项目一样实现 MVC 控制器，并使用 Entity Framework API 或其他 API。在新的 Web API 项目中，你可以看到这个微服务中唯一的依赖是 ASP.NET Core 本身。在内部的 Microsoft.AspNetCore.All 依赖中，它引用了 Entity Framework 和许多其他 .NET Core Nuget 包，如图 8-7 所示。

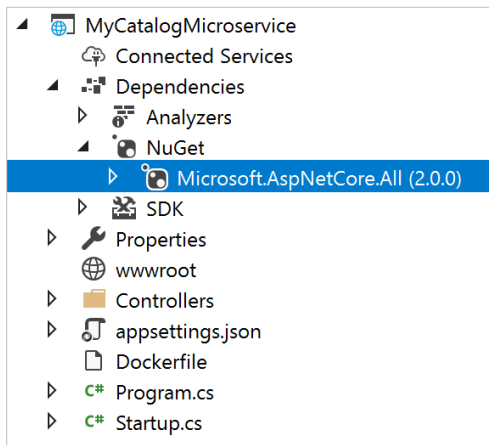


图 8-7. 一个简单的 CRUD Web API 微服务中的依赖关系

使用 Entity Framework Core 实现 CRUD Web API 服务

Entity Framework (EF) Core 是流行的 Entity Framework 数据访问技术的轻量级，可扩展和跨平台版本。EF Core 是一个对象关系映射器 (ORM)，它使 .NET 开发人员可以使用 .NET 对象处理数据库。

目录微服务使用 EF 和 SQL Server 提供，因为它的数据库运行在具有 SQL Server for Linux Docker 镜像的容器中。但是，数据库可以部署到任何 SQL Server 中，例如 Windows 本地部署或 Azure SQL DB。您唯一需要更改的是 ASP.NET Web API 微服务中的连接字符串。

数据模型

通过 EF Core，使用模型执行数据访问。模型由实体类和派生上下文组成，表示与数据库的会话，允许您查询和保存数据。您可以从现有数据库生成模型，手动编写模型以匹配数据库，或使用 EF 迁移从模型创建数据库（随着模型和时间而变化）。对于目录微服务，我们使用后面的方法。您可以在以下代码示例中看到 CatalogItem 实体类的示例，它是一个简单的 Plain Old CLR Object (POCO) 实体类。

```
public class CatalogItem
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public decimal Price { get; set; }
    public string PictureFileName { get; set; }
    public string PictureUri { get; set; }
    public int CatalogTypeId { get; set; }
    public CatalogType CatalogType { get; set; }
    public int CatalogBrandId { get; set; }
    public CatalogBrand CatalogBrand { get; set; }
    public int AvailableStock { get; set; }
    public int RestockThreshold { get; set; }
    public int MaxStockThreshold { get; set; }

    public bool OnReorder { get; set; }
    public CatalogItem() { }

    // Additional code ...
}
```

您还需要一个表示数据库会话的 DbContext。对于目录微服务，CatalogContext 类派生自 DbContext 基类，如以下示例所示：

```
public class CatalogContext : DbContext
{
    public CatalogContext(DbContextOptions<CatalogContext> options) : base(options)
    {
    }
    public DbSet<CatalogItem> CatalogItems { get; set; }
    public DbSet<CatalogBrand> CatalogBrands { get; set; }
    public DbSet<CatalogType> CatalogTypes { get; set; }
    // Additional code ...
}
```

您可以拥有其他的 DbContext 实现。例如，在示例 Catalog.API 微服务中，有另一个名为 CatalogContextSeed 的 DbContext，它在第一次尝试访问数据库时自动填充示例数据。此方法对演示数据很有用。

在 DbContext 中，您还可以使用 OnModelCreating 方法来自定义对象/数据库实体映射和其他 [EF 扩展性特点](#)。

从 Web API 控制器查询数据

实体类的实例通常使用 Language Integrated Query (LINQ) 从数据库中检索，如以下示例所示：

```
[Route("api/v1/[controller]")]
public class CatalogController : ControllerBase
{
    private readonly CatalogContext _catalogContext;
    private readonly CatalogSettings _settings;
    private readonly ICatalogIntegrationEventService
        _catalogIntegrationEventService;
    public CatalogController(CatalogContext context,
        IOptionsSnapshot<CatalogSettings> settings,
        ICatalogIntegrationEventService
            catalogIntegrationEventService)
    {
        _catalogContext = context ?? throw new
            ArgumentNullException(nameof(context));
        _catalogIntegrationEventService = catalogIntegrationEventService ??
            throw new ArgumentNullException(nameof(catalogIntegrationEventService));
        _settings = settings.Value;
        ((DbContext)context).ChangeTracker.QueryTrackingBehavior =
            QueryTrackingBehavior.NoTracking;
    }
    // GET api/v1/[controller]/items[?pageSize=3&pageIndex=10]
    [HttpGet]
    [Route("[action]")]
    [ProducesResponseType(typeof(PaginatedItemsViewModel<CatalogItem>)),
        (int)HttpStatusCode.OK]
    public async Task<IActionResult> Items([FromQuery]int pageSize = 10,
        [FromQuery]int pageIndex = 0)
    {
        var totalItems = await _catalogContext.CatalogItems
            .LongCountAsync();

        var itemsOnPage = await _catalogContext.CatalogItems
            .OrderBy(c => c.Name)
            .Skip(pageSize * pageIndex)
            .Take(pageSize)
            .ToListAsync();

        itemsOnPage = ChangeUriPlaceholder(itemsOnPage);
    }
}
```



```

        var model = new PaginatedItemsViewModel<CatalogItem>(
            pageIndex, pageSize, totalItems, itemsOnPage);

        return Ok(model);
    }
    //...
}

```

保存数据

使用实体类的实例在数据库中创建，删除和修改数据。您可以在 Web API 控制器中添加以下硬编码示例（在这种情况下为 Mock 数据）。

```

var catalogItem = new CatalogItem() {CatalogTypeId=2, CatalogBrandId=2,
                                     Name="Roslyn T-Shirt", Price = 12};

_context.Catalog.Add(catalogItem);
_context.SaveChanges();

```

ASP.NET Core 和 Web API 控制器中的依赖注入

在 ASP.NET Core 中，您可以使用开箱即用的依赖注入 (DI)。您不需要设置第三方控制反转 (IoC) 容器，但如果需要，可以将首选的 IoC 容器插入到 ASP.NET Core 基础架构中。在这种情况下，这意味着您可以通过控制器的构造函数直接注入所需的 EF DbContext 或其他仓储库。

在 CatalogController 类上面的例子中，我们通过 CatalogController()构造函数注入了一个 CatalogContext 类型的对象和其他对象。

在 Web API 项目中一个需要设置的重要配置是将 DbContext 类注册到服务的 IoC 容器中。您通常在 Startup 类中通过在 ConfigureServices 方法中调用 services.AddDbContext<DbContext>() 方法执行此操作，如以下示例所示：

```

public void ConfigureServices(IServiceCollection services)
{
    // Additional code...

    services.AddDbContext<CatalogContext>(options =>
    {
        options.UseSqlServer(Configuration["ConnectionString"],
            sqlServerOptionsAction: sqlOptions =>
            {
                sqlOptions.
                    MigrationsAssembly(
                        typeof(Startup).
                            GetTypeInfo().
                                Assembly.
                                    GetName().Name);
            }
        );
    });
    //Configuring Connection Resiliency:
}

```

```
        sqlOptions.
            EnableRetryOnFailure(maxRetryCount: 5,
                                maxRetryDelay: TimeSpan.FromSeconds(30),
                                errorNumbersToAdd: null);
    });
    // Changing default behavior when client evaluation occurs to throw.
    // Default in EFCore would be to log warning when client evaluation is done.
    options.ConfigureWarnings(warnings => warnings.Throw(
        RelationalEventId.QueryClientEvaluationWarning));
});
//...
}
```

其他资源

- **查询数据**
<https://docs.microsoft.com/ef/core/querying/index>
- **保存数据**
<https://docs.microsoft.com/ef/core/saving/index>

Docker 容器使用的 DB 连接字符串和环境变量

您可以使用 ASP.NET Core 设置，并将 ConnectionString 属性添加到 settings.json 文件中，如以下示例所示：

```
{
  "ConnectionString": "Server=tcp:127.0.0.1,5433;Initial Catalog=
    Microsoft.eShopOnContainers.Services.CatalogDb;User Id=sa;Password=Pass@word",
  "ExternalCatalogBaseUrl": "http://localhost:5101",
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

settings.json 文件可以具有 ConnectionString 属性或任何其他属性的默认值。但是，当使用 Docker 时，这些属性将被您在 docker-compose.override.yml 文件中指定的环境变量的值覆盖。

您可以从 docker-compose.yml 或 docker-compose.override.yml 文件初始化这些环境变量，以便 Docker 将其设置为操作系统环境变量，如下面的 docker-compose.override.yml 文件所示（连接字符串和其他行在这个例子中换行了，但在您自己的文件中不会换行）。

```
# docker-compose.override.yml
catalog.api:
  environment:
```

```
- ConnectionString=Server=sql.data;
    Database=Microsoft.eShopOnContainers.Services.CatalogDb;
    User Id=sa;Password=Pass@word
# Additional environment variables for this service
ports:
- "5101:80"
```

解决方案级别的 docker-compose.yml 文件不仅比项目或微服务级别的配置文件更加灵活，而且更加安全。请考虑您为每个微服务构建的 Docker 镜像不包含 docker-compose.yml 文件，仅包含每个微服务的二进制文件和配置文件，包括 Dockerfile。但是 docker-compose.yml 和 docker-compose.override.yml 文件并不包含在 Docker 镜像中，不会随应用程序一起部署；它仅在部署时使用。

因此，在这些 docker-compose.yml 文件中放置环境变量值（即使不加密值）比将这些值放置在与代码一起部署的常规.NET 配置文件中更为安全。

最后，您可以通过使用 Configuration["ConnectionString"]，从代码中获取该值，如刚才的代码示例中的 ConfigureServices 方法所示。

但是，对于生产环境，您可能需要探索如何存储连接字符串之类机密信息的其他方法。通常这将由您选择的编排引擎管理，您可以参考 [Docker Swarm 加密管理](#) 的内容。

在 ASP.NET Web API 中实现版本控制

随着业务需求的变化，可能会增加新的资源集合，资源之间的关系可能会发生变化，资源结构可能会被修改。更新 Web API 以处理新的需求是一个相对简单的过程，但是您必须考虑这种更改对调用 Web API 的客户端应用程序产生的影响。虽然设计和实现 Web API 的开发人员可以完全控制该 API，但开发人员对可能由第三方组织远程构建的客户端应用程序没有相同程度的控制。

版本控制使 Web API 能够指示其暴露的功能和资源。然后，客户端应用程序可以向特定版本的功能或资源提交请求。实现版本控制有几种方法：

- URI 版本控制
- 查询字符串版本控制
- Header 版本

查询字符串和 URI 版本控制是最简单的实现。Header 版本控制是一个很好的方法。但是，Header 版本不像 URI 版本控制那样明显和直接。因为 URL 版本是最简单和最明确的，所以 eShopOnContainers 示例应用程序使用 URI 版本控制。

使用 URI 版本控制，如 eShopOnContainers 示例应用程序中每次修改 Web API 或更改资源模式时，都可以向每个资源的 URI 添加一个版本号。现有 URI 应该像以前一样继续运行，返回符合与请求的版本相匹配的模式的资源。

如下面的代码示例所示，可以通过使用 Web API 中的 Route 属性来设置版本，这使 URI 中的版本很明显（在这种情况下为 v1）。

```
[Route("api/v1/[controller]")]
public class CatalogController : ControllerBase
{
    // Implementation ...
}
```

这种版本控制机制很简单，取决于服务器将请求路由到适当的服务。然而，对于更复杂的版本控制和使用 REST 的最佳方法来说，您应该使用超媒体并实现 [HATEOAS \(Hypertext as the Engine of Application State\)](#)。

其他资源

- **Scott Hanselman. 简易 ASP.NET Core RESTful Web API 版本控制**
<http://www.hanselman.com/blog/ASPNETCoreRESTfulWebAPIVersioningMadeEasy.aspx>
- **对 RESTful Web API 进行版本控制**
<https://docs.microsoft.com/azure/architecture/best-practices/api-design#versioning-a-restful-web-api>
- **Roy Fielding. 版本控制，超媒体和 REST**
<https://www.infoq.com/articles/roy-fielding-on-versioning>

从 ASP.NET Core Web API 生成 Swagger 描述元数据

[Swagger](#) 是一个常用的开源框架，由大量工具生态系统支持，可帮助您设计，构建，记录和调用您的 RESTful API。它正在成为 API 描述元数据域的标准。您应该使用 Swagger 描述任何类型的微服务元数据，包括数据驱动的微服务或更高级的领域驱动的微服务（如下一节所述）。

Swagger 的核心是 Swagger 规范，它是 JSON 或 YAML 文件中的 API 描述元数据。该规范为您的 API 创建 RESTful 契约，以人机可读的格式详细描述其所有资源和操作，以便于开发，发现和集成。

该规范是 OpenAPI 规范（OAS）的基础，并在开放，透明和协作的社区中开发，以将 RESTful 接口定义的方式标准化。

该规范定义了如何发现服务以及如何理解其功能的结构。有关更多信息，包括网页编辑器和来自 Spotify、Uber、Slack 和 Microsoft 等公司的 Swagger 规范的示例，请参阅 Swagger 站点（<http://swagger.io>）。

为什么使用 Swagger？

为您的 API 生成 Swagger 元数据的主要原因如下。

其他产品能够自动调用和集成您的 API。 数十种产品和[商业工具](#)以及许多[库和框架](#)支持 Swagger。Microsoft 拥有可以自动使用基于 Swagger 的 API 的高级产品和工具，如下所示：

- [AutoRest](#)。您可以自动生成调用 Swagger 的 .NET 客户端类。该工具可以从 CLI 中使用，还可与 Visual Studio 集成，以便通过 GUI 轻松使用。
- [Microsoft Flow](#)。您可以自动[使用 API 并集成](#)到高级 Microsoft Flow 工作流程中，无需任何编程技能。
- [Microsoft PowerApps](#)。您可以使用 [PowerApps Studio](#) 构建的 [PowerApps 移动应用程序](#)自动使用 API，而无需编程技能。
- [Azure App Service Logic Apps](#)。您可以自动[使用并将您的 API 集成到 Azure 应用服务逻辑应用 \(App Service Logic App\)](#) 中，而不需要编程技能。

能够自动生成 API 文档。 当您创建大型 RESTful API（例如复杂的基于微服务的应用程序）时，需要处理许多使用不同数据模型的终结点，这些模型在请求和响应负载中使用。拥有适当的文档并拥有一个稳定的 API 资源管理器，就像您使用 Swagger 一样，是您的 API 成功和易于开发人员使用的关键。

Swagger 的元数据是 Microsoft Flow、PowerApps 和 Azure Logic Apps 用于理解如何使用 API 并连接到它们的元数据。

如何使用 Swashbuckle NuGet 软件包自动生成 API Swagger 元数据

手动生成 Swagger 元数据（在 JSON 或 YAML 文件中）可能非常繁琐。但是，您可以通过使用 [Swashbuckle NuGet 软件包](#)来动态生成 Swagger API 元数据，使 ASP.NET Web API 服务的 API 发现自动化。

Swashbuckle 会为您的 ASP.NET Web API 项目自动生成 Swagger 元数据。它支持 ASP.NET Core Web API 项目和传统的 ASP.NET Web API 以及任何其他风格，如 Azure API App、Azure Mobile App、基于 ASP.NET 的 Azure Service Fabric 微服务。它还支持部署在容器上的普通 Web API，如示例应用程序。

Swashbuckle 结合了 API Explorer 和 Swagger 或 [swagger-ui](#)，为您的 API 消费者提供丰富的发现和文档体验。除了 Swagger 元数据生成器引擎外，Swashbuckle 还包含一个 swagger-ui 的嵌入式版本，一旦安装 Swashbuckle，它将自动运行。

这意味着您可以使用一个具有良好发现性的 UI 来补充您的 API，以帮助开发人员使用您的 API。它需要非常少量的代码和维护，因为它是自动生成的，允许您专注于构建您的 API。API Explorer 的结果如图 8-8 所示。

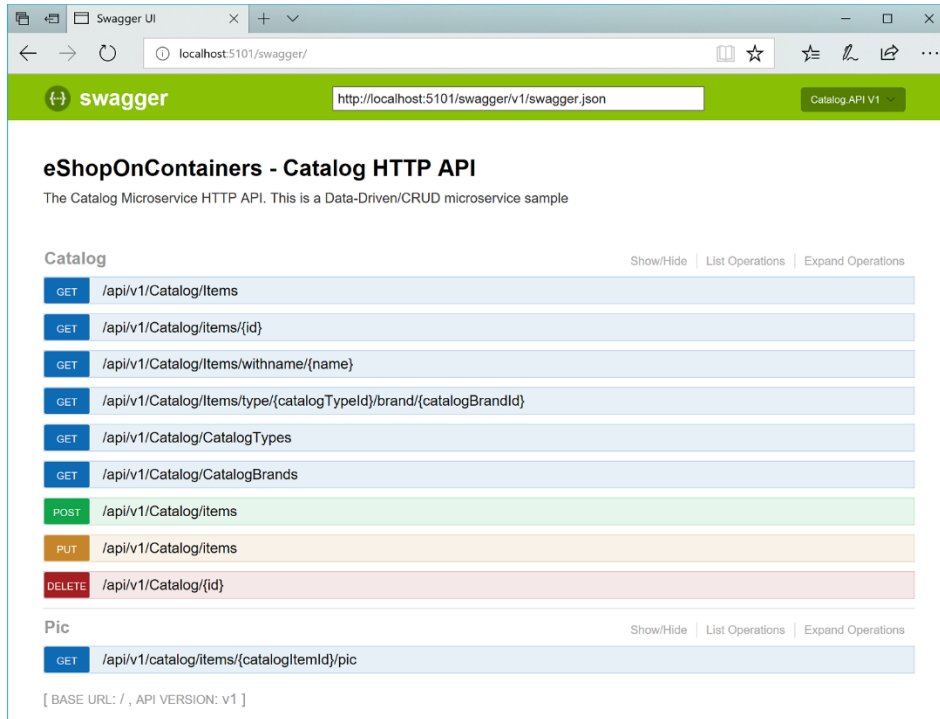


图 8-8. Swashbuckle API Explorer, 基于 Swagger 元数据的 eShopOnContainers 目录微服务

API 资源管理器不是此处最重要的事情。一旦您拥有可以在 Swagger 元数据中描述自己的 Web API, 就可以从基于 Swagger 的工具无缝地使用您的 API, 包括针对许多平台的客户端代理类代码生成器。例如, 如上所述, [AutoRest](#) 自动生成 .NET 客户端类。但是还可以使用其他工具, 如 [swagger-codegen](#), 它们可以自动生成 API 客户端库, 服务器存根和文档。

目前, 对 ASP.NET Core 应用程序来说, Swashbuckle 包含两个内部 NuGet 软件包, 位于高级元软件包 Swashbuckle.AspNetCore.SwaggerGen version 1.0.0 或更高版本下。

在 Web API 项目中将这些依赖关系添加到这些 NuGet 包之后, 您还需要在 Startup 类中配置 Swagger, 如下面的代码所示:

```
public class Startup
{
    public IConfigurationRoot Configuration { get; }

    // Other startup code...

    public void ConfigureServices(IServiceCollection services)
    {
        // Other ConfigureServices() code...

        // Add framework services.
        services.AddSwaggerGen(options =>
        {
            options.DescribeAllEnumsAsStrings();
        });
    }
}
```

```

options.SwaggerDoc("v1", new Swashbuckle.AspNetCore.Swagger.Info
{
    Title = "eShopOnContainers - Catalog HTTP API",
    Version = "v1",
    Description = "The Catalog Microservice HTTP API",
    TermsOfService = "Terms of Service"
});
});
// Other ConfigureServices() code...
}

public void Configure(IApplicationBuilder app,
                    IHostingEnvironment env,
                    ILoggerFactory loggerFactory)
{
    // Other Configure() code...
    // ...

    app.UseSwagger()
        .UseSwaggerUI(c =>
        {
            c.SwaggerEndpoint("/swagger/v1/swagger.json", "Catalog.API V1");
        });
}
}

```

完成此操作后，您可以启动您的应用程序，并使用这些 URL 来浏览以下 Swagger JSON 和 UI 终结点：

```

http://<your-root-url>/swagger/v1/swagger.json
http://<your-root-url>/swagger/

```

您之前看到的是由 Swashbuckle 为 `http://<your-root-url>/swagger` 这样的 URL 创建的 UI。在图 8-9 中，您还可以看到如何测试所有 API 方法。

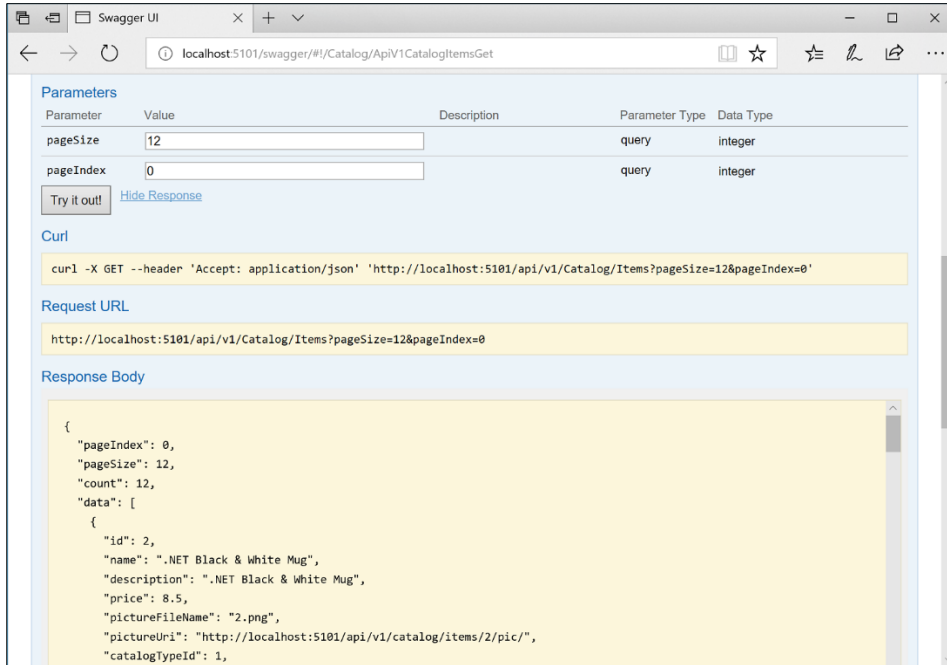


图 8-9. Swashbuckle UI 测试 Catalog/Items API 方法

图 8-10 显示了当您使用 Postman 请求 <your-rooturl> /swagger/v1/swagger.json 时，eShopOnContainers 微服务生成的 Swagger JSON 元数据（这是由工具在底层使用的）。

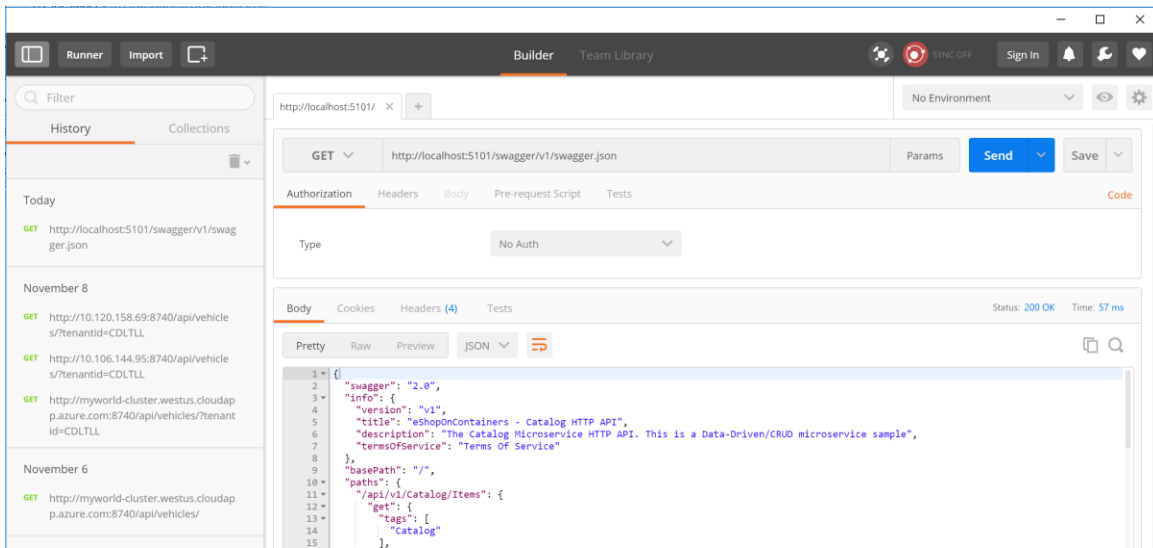


图 8-10. Swagger JSON 元数据

这很简单。因为它是自动生成的，所以当您向 API 添加更多功能时，Swagger 元数据将会增长。

其他资源

- 使用 Swagger 的 ASP.NET Web API 帮助页面
<https://docs.microsoft.com/aspnet/core/tutorials/web-api-help-pages-using-swagger>

使用 docker-compose.yml 定义多容器应用程序

在本书中，[docker-compose.yml](#) 文件在[步骤 4、当构建多容器 Docker 应用程序的时候](#)，在[docker-compose.yml 中定义您的服务](#)中作了介绍。但是，使用 docker-compose 文件还有其他值得探索细节的方法。

例如，您可以显式描述如何在 docker-compose.yml 文件中部署多容器应用程序。或者，您还可以描述如何构建自定义 Docker 镜像。（自定义 Docker 图像也可以使用 Docker CLI 构建。）

基本上，您需要为每个容器部署定义要附加的某些特性。一旦拥有了多容器部署描述文件，就可以在由 [docker-compose up](#) CLI 命令编排的单个操作中部署整个解决方案，也可以从 Visual Studio 透明部署。否则，您需要使用 Docker CLI，通过使用命令行中的 docker run 命令在多个步骤中部署容器。因此，在 docker-compose.yml 中定义的每个服务必须指定一个镜像或构建。其他的选项是可选的，类似于它们对应 docker run 命令行的一些项。

以下 YAML 代码是 eShopOnContainers 示例使用的 docker-compose.yml 文件的定义，可以在全局使用，但此处作为单独使用。这不是 eShopOnContainers 中实际的 docker-compose 文件。相反，它是单个文件的简化和合并版本，这不是使用 docker-compose 文件的最佳方法，稍后将做出说明。

```

version: '2'
services:
  webmvc:
    image: eshop/webmvc
    environment:
      - CatalogUrl=http://catalog.api
      - OrderingUrl=http://ordering.api
      - BasketUrl=http://basket.api
    ports:
      - "5100:80"
    depends_on:
      - catalog.api
      - ordering.api
      - basket.api
  catalog.api:
    image: eshop/catalog.api
    environment:
      - ConnectionString=Server=sql.data;Initial Catalog=CatalogData;
        User Id=sa;Password=your@password
    expose:
      - "80"
    ports:
      - "5101:80"
    #extra hosts can be used for standalone SQL Server or services at the dev PC
    extra_hosts:
      - "CESARDLSURFBOOK:10.0.75.1"
    depends_on:
      - sql.data
  ordering.api:
    image: eshop/ordering.api
    environment:
      - ConnectionString=Server=sql.data;Database=Services.OrderingDb;
        User Id=sa;Password=your@password
    ports:
      - "5102:80"
    #extra hosts can be used for standalone SQL Server or services at the dev PC
    extra_hosts:
      - "CESARDLSURFBOOK:10.0.75.1"
    depends_on:
      - sql.data
  basket.api:
    image: eshop/basket.api
    environment:
      - ConnectionString=sql.data
    ports:
      - "5103:80"
    depends_on:
      - sql.data
  sql.data:
    environment:
      - SA_PASSWORD=your@password
      - ACCEPT_EULA=Y
    ports:
      - "5434:1433"
  basket.data:
    image: redis

```

此文件中的根键是 `services`。在该键下，您可以定义在执行 `docker-compose up` 命令时，或使用此 `docker-compose.yml` 文件从 Visual Studio 部署时，要部署和运行的服务。在这种情况下，`docker-compose.yml` 文件具有定义的多个服务，如下表所述。

docker-compose.yml 中的服务名称	描述
webmvc	包括从服务器端 C#调用微服务的 ASP.NET Core MVC 应用程序的容器
catalog.api	包括产品 ASP.NET Core Web API 微服务的容器
ordering.api	包括订单 ASP.NET Core Web API 微服务的容器
sql.data	运行 SQL Server for Linux，支持微服务数据库的容器
basket.api	购物篮 ASP.NET Core Web API 微服务容器
basket.data	运行 Redis 缓存服务的容器，将购物篮数据库作为 Redis 缓存

一个简单的 Web Service API 容器

针对单个容器，`catalog.api` 容器微服务具有一个很简单的定义：

```
catalog.api:
  image: eshop/catalog.api
  environment:
    - ConnectionString=Server=sql.data;Initial Catalog=CatalogData;
      User Id=sa;Password=your@password
  expose:
    - "80"
  ports:
    - "5101:80"

  #extra hosts can be used for standalone SQL Server or services at the dev PC
  extra_hosts:
    - "CESARDLSURFBOOK:10.0.75.1"

  depends_on:
    - sql.data
```

此容器化服务具有以下基本配置：

- 它基于自定义的 `eshop/catalog.api` 镜像。为了简单起见，文件中没有 `build: key` 设置。这意味着镜像必须已经构建（使用 `docker build`）或已从 Docker 注册表下载（使用 `docker pull` 命令）。
- 它使用连接字符串定义一个名为 `ConnectionString` 的环境变量，由 Entity Framework 用于访问包含目录数据模型的 SQL Server 实例。在这种情况下，同一 SQL Server 容器保存了多个数据

库。因此，您的开发机器中只需要较少的内存。但是，您也可以为每个微服务数据库部署一个 SQL Server 容器。

- SQL Server 的名称是 sql.data，它与 Linux 上运行 SQL Server 实例的容器的名称相同。这很方便，使用此名称解析（Docker 主机内部）能够解析网络地址，因此您不需要知道您正在从其他容器访问的容器的内部 IP。
- 因为连接字符串是由环境变量定义的，所以您可以通过不同的机制在不同的时间来设置该变量。例如，您可以在最终主机中部署生产环境时设置不同的连接字符串，也可以在 VSTS 或您更喜欢的其他 DevOps 系统中的 CI/CD 管道中进行设置。
- 它暴露端口 80，以便内部访问 Docker 主机中的 catalog.api 服务。主机目前是 Linux VM，因为它基于使用 Linux 的 Docker 镜像，但是您可以将容器配置为在 Windows 镜像上运行。
- 它将容器暴露的端口 80 转发到 Docker 主机（Linux VM）上的端口 5101。
- 它将 Web 服务链接到 sql.data 服务（在容器中运行的 Linux 数据库的 SQL Server 实例）。指定此依赖项时，只有 sql.data 容器已经启动，才能启动 catalog.api 容器。这很重要，因为 catalog.api 需要首先启动并运行 SQL Server 数据库。但是，这种容器依赖关系在许多情况下是不够的，因为 Docker 仅在容器级别进行检查。有时，服务（在这种情况下为 SQL Server）可能还没有准备好，因此建议您在客户端微服务器中实现基于指数退避算法的重试逻辑。这样一来，如果一个依赖项容器在短时间内没有准备就绪，那么应用程序仍可迅速恢复。
- 它被配置为允许访问外部服务器：extra_hosts 设置允许您访问外部服务器或 Docker 主机之外的机器（即默认的作为开发 Docker 主机的 Linux VM 之外），例如在您的开发 PC 上的本地 SQL Server 实例。

还有其他更高级的 docker-compose.yml 设置，我们将在以下部分中讨论。

使用 docker-compose 文件面向多种环境

docker-compose.yml 是定义文件，可以被理解该格式的多个基础架构使用。最简单的工具是 docker-compose 命令，但其他工具如编排引擎（例如 Docker Swarm）也能够理解该文件。

因此，通过使用 docker-compose 命令可以针对以下主要场景。

开发环境

开发应用程序时，能够在独立的开发环境中运行应用程序很重要。您可以使用 docker-compose CLI 命令来创建该环境，或通过 Visual Studio 来使用 docker-compose。

docker-compose.yml 文件允许您配置和记录应用程序的所有服务依赖关系（其他服务，缓存，数据库，队列等）。使用 docker-compose CLI 命令，您可以使用一个命令（docker-compose up）为每个依赖关系创建和启动一个或多个容器。

docker-compose.yml 文件是由 Docker 引擎解释的配置文件，也可以作为多容器应用程序组合的文档文件。

测试环境

单元测试和集成测试是任何持续部署（CD）或持续集成（CI）流程的重要组成部分。这些自动化测试需要一个独立的环境，这样它们不会受到应用程序数据中用户或任何其他变化的影响。

使用 Docker Compose，您可以使用命令提示符或脚本中的几个命令轻松创建和销毁该独立环境，如下命令：

```
docker-compose up -d
./run_unit_tests
docker-compose down
```

生产环境

您也可以使用 Compose 部署到远程 Docker 引擎。典型的情况是部署到单个 Docker 主机实例（如使用 [Docker Machine](#) 配置的生产 VM 或服务器）。但它也可能是一个完整的 [Docker Swarm](#) 集群，因为集群也与 dockercompose.yml 文件兼容。

如果您正在使用任何其他编排引擎（Azure Service Fabric，Mesos DC / OS，Kubernetes 等），则可能需要像其他 docker-compose.yml 中一样添加安装和元数据配置设置，但要按其他编排引擎需要的格式。

无论如何，docker-compose 是用于开发，测试和生产工作流程的便捷工具和元数据格式，尽管您正在使用的编排引擎的生产工作流程可能会有所不同。

使用多个 docker-compose 文件来处理多个环境

面向不同的环境时，您应该使用多个 compose 文件。这允许您根据环境创建多个配置。

重写基本 docker-compose 文件

您可以使用单个 docker-compose.yml 文件，如前几节所示的简化示例。但是，对大多数应用程序来说不推荐使用这种方式。

默认情况下，Compose 读取两个文件，一个 `docker-compose.yml` 和一个可选的 `docker-compose.override.yml` 文件。如图 8-11 所示，当您使用 Visual Studio 并启用 Docker 支持时，Visual Studio 还会创建一个额外的 `docker-compose.ci.build.yml` 文件供您 CI/CD 管道中使用，就像在 VSTS 中一样。

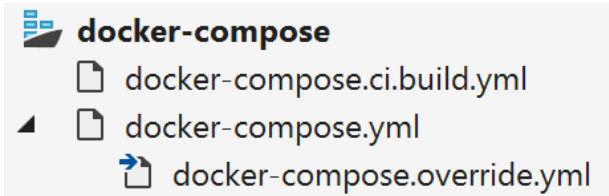


图 8-11. Visual Studio 2017 中的 docker-compose 文件

您可以使用任何编辑器（如 Visual Studio Code 或 Sublime）编辑 docker-compose 文件，并使用 `docker-compose up` 命令运行应用程序。

按照惯例，`docker-compose.yml` 文件包含您的基本配置和其他静态设置。这意味着服务配置不应该根据您面向的部署环境而改变。

`docker-compose.override.yml` 文件，如其名称所示，包含重写基本配置的配置设置，例如依赖于部署环境的配置。您也可以使用不同名称的多个重写文件。重写文件通常包含应用程序所需的附加信息，但特定于环境或部署。

针对多个环境

一个典型的用例是定义多个 compose 文件，以针对多个环境，如生产，预发布，持续集成或开发。为了支持这些差异，您可以将 Compose 配置拆分成多个文件，如图 8-12 所示。

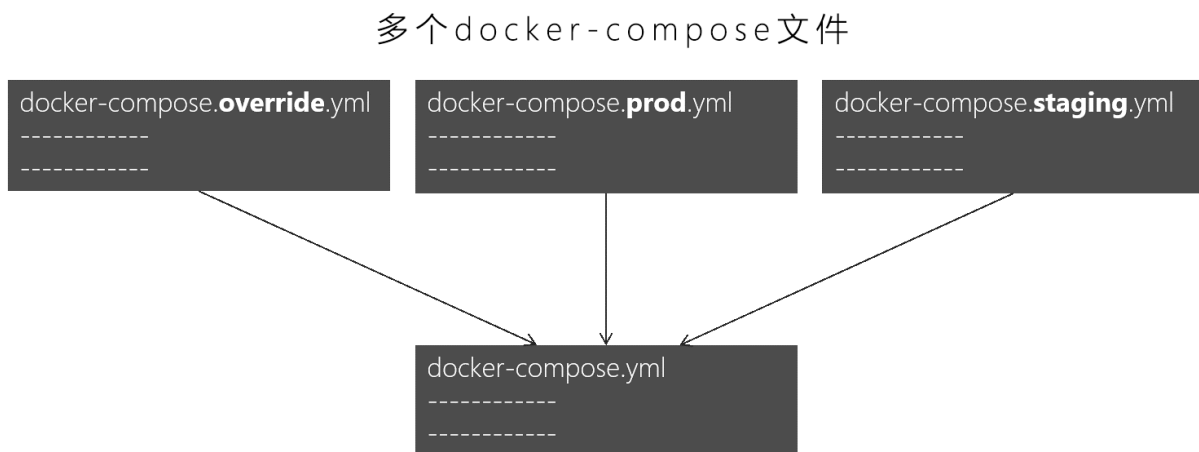


图 8-12. 多个 docker-compose 文件重写基本 docker-compose.yml 文件中的值

从基础 docker-compose.yml 文件开始，此基础文件必须包含依赖环境而不会更改的基本或静态配置设置。例如，eShopOnContainers 具有以下 docker-compose.yml 文件（已经简化为较少的服务）作为基础文件。

```
#docker-compose.yml (Base)
version: '3'
services:
  basket.api:
    image: eshop/basket.api:${TAG:-latest}
    build:
      context: ./src/Services/Basket/Basket.API
      dockerfile: Dockerfile
    depends_on:
      - basket.data
      - identity.api
      - rabbitmq

  catalog.api:
    image: eshop/catalog.api:${TAG:-latest}
    build:
      context: ./src/Services/Catalog/Catalog.API
      dockerfile: Dockerfile
    depends_on:
      - sql.data
      - rabbitmq

  marketing.api:
    image: eshop/marketing.api:${TAG:-latest}
    build:
      context: ./src/Services/Marketing/Marketing.API
      dockerfile: Dockerfile
    depends_on:
      - sql.data
      - nosql.data
      - identity.api
      - rabbitmq

  webmvc:
    image: eshop/webmvc:${TAG:-latest}
    build:
      context: ./src/Web/WebMVC
      dockerfile: Dockerfile
    depends_on:
      - catalog.api
      - ordering.api
      - identity.api
      - basket.api
      - marketing.api

  sql.data:
    image: microsoft/mssql-server-linux:2017-latest

  nosql.data:
    image: mongo

  basket.data:
    image: redis

  rabbitmq:
```

```
image: rabbitmq:3-management
```

基础 docker-compose.yml 文件中的值不应该因为针对不同的部署环境而更改。

例如，如果您关注 webmvc 服务定义，无论您可能面向哪个环境，都可以看到很多类似的信息。如下信息：

- 服务名称：webmvc。
- 容器的自定义镜像：eshop/webmvc。
- 构建自定义 Docker 镜像的命令，指示要使用哪个 Dockerfile。
- 依赖的其他服务，因此在其他依赖项容器启动之后，此容器才会启动。

您可以进行其他配置，但重要的一点是在基础 dockercompose.yml 文件中，您只需要设置不同环境中的一般信息。然后在 docker-compose.override.yml 或用于生产环境或预发布环境的类似文件中，再放置特定于每个环境的配置。

通常，docker-compose.override.yml 用于您的开发环境，如 eShopOnContainers 中以下示例所示：

```
#docker-compose.override.yml (Extended config for DEVELOPMENT env.)
version: '3'

services:
# Simplified number of services here:

  basket.api:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=http://0.0.0.0:80
      - ConnectionString=${ESHOP_AZURE_REDIS_BASKET_DB:-basket.data}
      - identityUrl=http://identity.api
      - IdentityUrlExternal=http://${ESHOP_EXTERNAL_DNS_NAME_OR_IP}:5105
      - EventBusConnection=${ESHOP_AZURE_SERVICE_BUS:-rabbitmq}
      - EventBusUserName=${ESHOP_SERVICE_BUS_USERNAME}
      - EventBusPassword=${ESHOP_SERVICE_BUS_PASSWORD}
      - AzureServiceBusEnabled=False
      - ApplicationInsights__InstrumentationKey=${INSTRUMENTATION_KEY}
      - OrchestratorType=${ORCHESTRATOR_TYPE}
      - UseLoadTest=${USE_LOADTEST:-False}

    ports:
      - "5103:80"

  catalog.api:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=http://0.0.0.0:80
```



```

- ConnectionString=${ESHOP_AZURE_CATALOG_DB:-
Server=sql.data;Database=Microsoft.eShopOnContainers.Services.CatalogDb;User
Id=sa;Password=Pass@word}
- PicBaseUrl=${ESHOP_AZURE_STORAGE_CATALOG_URL:-
http://localhost:5101/api/v1/catalog/items/[0]/pic/}
- EventBusConnection=${ESHOP_AZURE_SERVICE_BUS:-rabbitmq}
- EventBusUserName=${ESHOP_SERVICE_BUS_USERNAME}
- EventBusPassword=${ESHOP_SERVICE_BUS_PASSWORD}
- AzureStorageAccountName=${ESHOP_AZURE_STORAGE_CATALOG_NAME}
- AzureStorageAccountKey=${ESHOP_AZURE_STORAGE_CATALOG_KEY}
- UseCustomizationData=True
- AzureServiceBusEnabled=False
- AzureStorageEnabled=False
- ApplicationInsights__InstrumentationKey=${INSTRUMENTATION_KEY}
- OrchestratorType=${ORCHESTRATOR_TYPE}
ports:
- "5101:80"

marketing.api:
environment:
- ASPNETCORE_ENVIRONMENT=Development
- ASPNETCORE_URLS=http://0.0.0.0:80
- ConnectionString=${ESHOP_AZURE_MARKETING_DB:-
Server=sql.data;Database=Microsoft.eShopOnContainers.Services.MarketingDb;User
Id=sa;Password=Pass@word}
- MongoConnectionString=${ESHOP_AZURE_COSMOSDB:-mongodb://nosql.data}
- MongoDatabase=MarketingDb
- EventBusConnection=${ESHOP_AZURE_SERVICE_BUS:-rabbitmq}
- EventBusUserName=${ESHOP_SERVICE_BUS_USERNAME}
- EventBusPassword=${ESHOP_SERVICE_BUS_PASSWORD}
- identityUrl=http://identity.api
- IdentityUrlExternal=http://${ESHOP_EXTERNAL_DNS_NAME_OR_IP}:5105
- CampaignDetailFunctionUri=${ESHOP_AZUREFUNC_CAMPAIGN_DETAILS_URI}
- PicBaseUrl=${ESHOP_AZURE_STORAGE_MARKETING_URL:-
http://localhost:5110/api/v1/campaigns/[0]/pic/}
- AzureStorageAccountName=${ESHOP_AZURE_STORAGE_MARKETING_NAME}
- AzureStorageAccountKey=${ESHOP_AZURE_STORAGE_MARKETING_KEY}
- AzureServiceBusEnabled=False
- AzureStorageEnabled=False
- ApplicationInsights__InstrumentationKey=${INSTRUMENTATION_KEY}
- OrchestratorType=${ORCHESTRATOR_TYPE}
- UseLoadTest=${USE_LOADTEST:-False}
ports:
- "5110:80"

webmvc:
environment:
- ASPNETCORE_ENVIRONMENT=Development
- ASPNETCORE_URLS=http://0.0.0.0:80
- CatalogUrl=http://catalog.api
- OrderingUrl=http://ordering.api
- BasketUrl=http://basket.api
- LocationsUrl=http://locations.api
- IdentityUrl=http://10.0.75.1:5105
- MarketingUrl=http://marketing.api
- CatalogUrlHC=http://catalog.api/hc
- OrderingUrlHC=http://ordering.api/hc
- IdentityUrlHC=http://identity.api/hc

```

```

- BasketUrlHC=http://basket.api/hc
- MarketingUrlHC=http://marketing.api/hc
- PaymentUrlHC=http://payment.api/hc
- UseCustomizationData=True
- ApplicationInsights_InstrumentationKey=${INSTRUMENTATION_KEY}
- OrchestratorType=${ORCHESTRATOR_TYPE}
- UseLoadTest=${USE_LOADTEST:-False}
ports:
- "5100:80"
sql.data:
environment:
- MSSQL_SA_PASSWORD=Pass@word
- ACCEPT_EULA=Y
- MSSQL_PID=Developer
ports:
- "5433:1433"
nosql.data:
ports:
- "27017:27017"
basket.data:
ports:
- "6379:6379"
rabbitmq:
ports:
- "15672:15672"
- "5672:5672"

```

在这个例子中，开发重写配置将一些端口暴露给主机，使用重定向 URL 定义环境变量，并为开发环境指定连接字符串。所有这些设置只适用于开发环境。

当您运行 `docker-compose up`（或从 Visual Studio 启动）时，该命令将自动读取重写，就像这两个文件被合并一样。

假设您需要另一个 Compose 文件用于生产环境，但具有不同的配置值，端口或连接字符串。您可以创建另一个覆盖文件，如具有不同设置和环境变量的名为 `docker-compose.prod.yml` 的文件。该文件可能存储在不同的 Git 仓库中，或者由不同的团队管理和保管。

如何使用特定的重写文件进行部署

要使用多个重写文件或使用不同名称的重写文件，可以使用 `docker-compose` 命令的 `-f` 选项来指定文件。Compose 按照命令行中指定的顺序合并文件。以下示例显示如何使用重写文件进行部署。

```
docker-compose -f docker-compose.yml -f docker-compose.prod.yml up -d
```

在 docker-compose 文件中使用环境变量

能够从环境变量获取配置信息是非常方便的，特别是在生产环境中，如前面的例子所示。您可以使用语法\${MY_VAR}引用 docker-compose 文件中的环境变量。dockercompose.prod.yml 文件中的以下行显示了如何引用环境变量的值。

```
IdentityUrl=http://${ESHOP_PROD_EXTERNAL_DNS_NAME_OR_IP}:5105
```

根据您的主机环境（Linux、Windows、云集群等），以不同的方式创建和初始化环境变量。但是，方便的方法是使用.env 文件。docker-compose 文件支持在.env 文件中声明默认环境变量。这些环境变量的值是默认值。但是，您可能用每个环境中定义的值（主机操作系统或集群中的环境变量）重写它们。将.env 文件放在执行 docker-compose 命令的文件夹中。

以下示例显示.env 文件，如 eShopOnContainers 应用程序的.env 文件。

```
# .env file
ESHOP_EXTERNAL_DNS_NAME_OR_IP=localhost
ESHOP_PROD_EXTERNAL_DNS_NAME_OR_IP=10.121.122.92
```

Docker-compose 要求.env 文件中的每行格式为<variable>=<value>。

请注意，运行时环境中设置的值始终重写.env 文件中定义的值。类似的，通过命令行命令参数传递的值也将重写.env 文件中设置的默认值。

其他资源

- **Docker Compose 概述**
<https://docs.docker.com/compose/overview/>
- **多个 Compose 文件**
<https://docs.docker.com/compose/extends/#multiple-compose-files>

构建优化的 ASP.NET Core Docker 镜像

如果您在互联网上查询 Docker 和 .NET Core 的资料，您会发现 Dockerfiles 通过将源代码复制到容器中来展示构建 Docker 镜像的简单性。这些示例表明，通过使用简单的配置，您便可以使用包含应用程序环境的 Docker 镜像。以下示例显示了一个简单的 Dockerfile。

```
FROM microsoft/dotnet
WORKDIR /app
ENV ASPNETCORE_URLS http://+:80
EXPOSE 80
COPY . .
RUN dotnet restore
ENTRYPOINT ["dotnet", "run"]
```

这样的 Docker 文件可以正常工作。但是，您可以大幅优化镜像，特别是生产镜像。

在容器和微型服务模型中，您将不断启动容器。典型的使用容器的方式不会重新启动一个休眠的容器，因为容器是一次性的。编排引擎（如 Docker Swarm、Kubernetes、DC/OS 或 Azure Service Fabric）只简单地创建新的镜像实例。这意味着您需要通过在构建应用程序时进行预编译来优化，以便实例化过程更快。当容器启动时，它应准备好运行了。您不应该在运行时从 dotnet CLI 使用 dotnet restore 和 dotnet build 命令进行恢复和编译，如您在许多有关 .NET Core 和 Docker 的博客文章中看到的那样。

.NET 团队一直在做重要的工作，使 .NET Core 和 ASP.NET Core 成为对容器优化的框架。不仅是 .NET Core 是一个内存占用空间小的轻量级框架；该团队还专注于启动性能，并生成了一些优化的 Docker 镜像，如 Docker Hub 提供的 [microsoft/aspnetcore](https://hub.docker.com/r/microsoft/aspnetcore) 镜像，与常规的 [microsoft/dotnet](https://hub.docker.com/r/microsoft/dotnet) 或 [microsoft/nanoserver](https://hub.docker.com/r/microsoft/nanoserver) 镜像相比具有更好的性能。[microsoft / aspnetcore](https://github.com/microsoft/aspnetcore) 镜像提供 aspnetcore_urls 到端口 80 的自动设置和程序集的 pre-ngend 缓存；这两个设置都会提升启动速度。

其他资源

- **使用 ASP.NET Core 构建优化的 Docker 镜像**
<https://blogs.msdn.microsoft.com/stevelasker/2016/09/29/building-optimized-docker-images-with-asp-net-core/>

从构建（持续集成）容器构建应用程序

Docker 的另一个优点是您可以从预配置的容器构建应用程序，如图 8-13 所示，因此您不需要创建构建机器或虚拟机来构建应用程序。您可以通过在开发机器上运行它来使用或测试该构建容器。但更有趣的是，您可以使用与您的 CI（持续集成）管道中相同的构建容器。

用容器构建应用程序

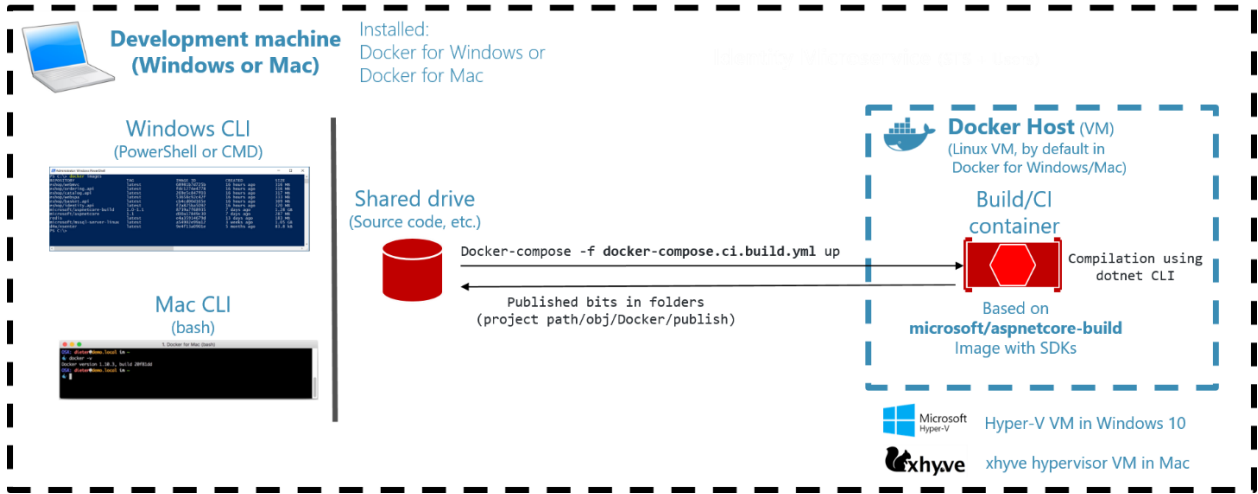


图 8-13. Docker 构建容器编译您的.NET 二进制组件

对于这种情况，我们提供了 [microsoft/aspnetcore-build](#) 镜像，您可以使用它来编译和构建您的 ASP.NET Core 应用程序。输出被放置在基于 [microsoft/aspnetcore](#) 镜像的镜像中，这是一个优化的运行时镜像，如前所述。

该 aspnetcore-build 镜像包含编译 ASP.NET Core 应用程序所需的一切内容，包括 .NET Core、ASP.NET SDK、npm、Bower、Gulp 等。

我们在构建时需要这些依赖关系。但是我们不想在运行时与应用程序一起附带这些依赖，因为它会使镜像更大，这是不必要的。在 eShopOnContainers 应用程序中，您可以通过运行以下 docker-compose 命令从容器中构建应用程序。

```
docker-compose -f docker-compose.ci.build.yml up
```

图 8-14 显示了在命令行运行的命令。

```
Administrator: Windows PowerShell
PS D:\GITREPOS\eshoponContainers> Docker-compose -f docker-compose.ci.build.yml up
Creating network "eshoponcontainers_default" with the default driver
Creating eshoponcontainers_ci-build_1
Attaching to eshoponcontainers_ci-build_1
ci-build_1 | Restoring packages for /src/src/Services/Catalog/Catalog.API/Catalog.API.csproj...
ci-build_1 | Restoring packages for /src/src/Services/Basket/Basket.API/Basket.API.csproj...
ci-build_1 | Restoring packages for /src/src/Services/Identity/Identity.API/Identity.API.csproj...
ci-build_1 | Installing Microsoft.AspNetCore.DataProtection.Abstractions 1.1.0.
ci-build_1 | Installing Microsoft.AspNetCore.Cryptography.Internal 1.1.0.
ci-build_1 | Installing Microsoft.DotNet.PlatformAbstractions 1.1.0.
```

图 8-14. 从容器构建您的.NET 应用程序

正如您所看到的，正在运行的容器是 ci-build_1 容器。这基于 aspnetcore-build 镜像，因此它可以从该容器内而不是从 PC 中编译和构建整个应用程序。这就是为什么在现实中它在 Linux 中构建和编译 .NET Core 项目，因为容器运行在默认的 Docker Linux 主机上。

该镜像的 [docker-compose.ci.build.yml](#) 文件（eShopOnContainers 的一部分）包含以下代码。您可以看到它使用 [microsoft/aspnetcore-build](#) 镜像启动一个构建容器。

```
version: '3'

services:

  ci-build:

    image: microsoft/aspnetcore-build:2.0

    volumes:
      - ./src

    working_dir: /src

    command: /bin/bash -c "pushd ./src/Web/WebSPA && npm rebuild node-sass && popd
    && dotnet restore ./eShopOnContainers-ServicesAndWebApps.sln && dotnet publish
    ./eShopOnContainers-ServicesAndWebApps.sln -c Release -o ./obj/Docker/publish"
```

一旦构建容器启动并运行，它将针对解决方案中的所有项目运行 .NET SDK dotnet restore 和 dotnet publish 命令，以编译 .NET 字节码。在这个示例中，由于 eShopOnContainers 还有一个基于 TypeScript 和 Angular 的客户端 SPA 代码，它还需要使用 npm 检查 JavaScript 依赖关系，但该操作与 .NET 字节码无关。

dotnet publish 命令构建并将每个项目文件夹中的编译输出发布到 ./obj/Docker/publish 文件夹，如图 8-15 所示。

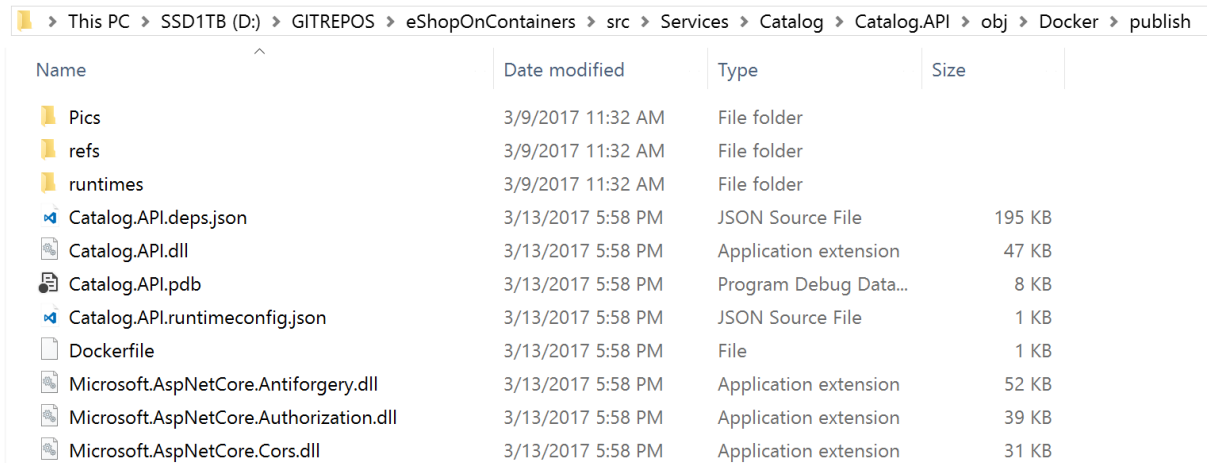


图 8-15. 由 dotnet publish 命令生成的字节码文件

从 CLI 创建 Docker 图像

一旦将应用程序输出发布到相关文件夹（每个项目中），下一步就是实际构建 Docker 镜像。为此，您可以使用 `docker-compose build` 和 `dockercompose up` 命令，如图 8-16 所示。

构建 Docker 映像并运行容器

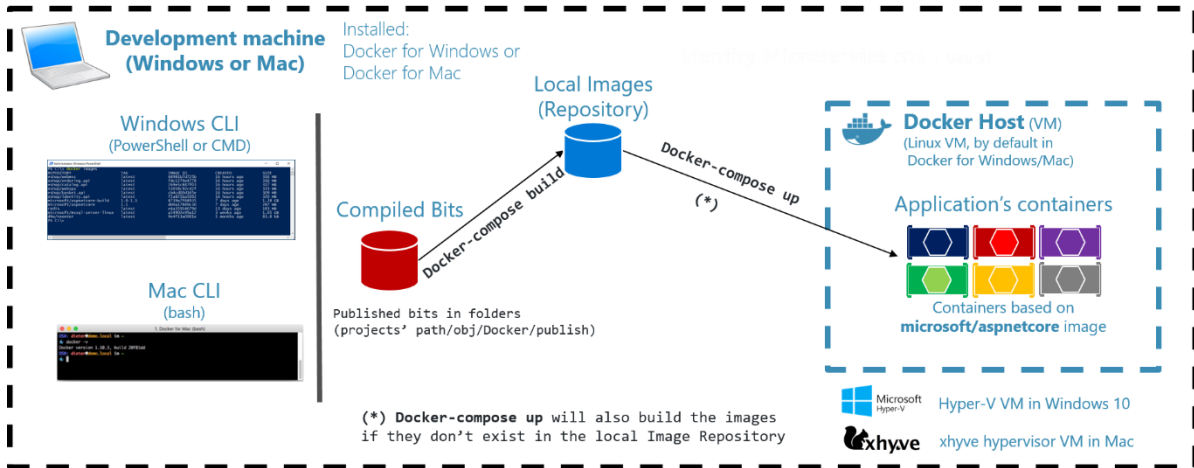


图 8-16. 构建 Docker 镜像并运行容器

在图 8-17 中，您可以看到 `docker-compose build` 命令的运行方式。

```
Administrator: Windows PowerShell
PS D:\GITREPOS\eshopOnContainers> docker-compose build
basket.data uses an image, skipping
sql.data uses an image, skipping
Building identity.api
Step 1/6 : FROM microsoft/aspnetcore:1.1
--> d00a17849c30
Step 2/6 : ARG source
--> Running in b149b22c88c9
--> 16295b92d4ee
Removing intermediate container b149b22c88c9
Step 3/6 : WORKDIR /app
--> ac31d193b6dc
Removing intermediate container 3296c6a7c16f
Step 4/6 : EXPOSE 80
```

图 8-17. 使用 `docker-compose build` 命令构建 Docker 镜像

`docker-compose build` 和 `docker-compose up` 命令之间的区别在于 `docker-compose up` 包括构建镜像和启动镜像两个操作。

当您使用 Visual Studio 时，所有这些步骤都在后台执行。Visual Studio 编译您的 .NET 应用程序，创建 Docker 镜像，并将容器部署到 Docker 主机中。Visual Studio 还提供了其他功能，例如直接从 Visual Studio 调试在 Docker 中运行的容器。

总体来说，您能以与您的 CI/CD 管道使用相同的方式构建应用程序 - 从容器而不是本地机器。创建镜像后，您只需要使用 `docker-compose up` 命令运行 Docker 镜像。

其他资源

- **从容器构建字节码：在 Windows CLI 环境中设置 eShopOnContainers 解决方案 (dotnet CLI、Docker CLI 和 VS Code)**
[https://github.com/dotnet/eShopOnContainers/wiki/03.-Setting-the-eShopOnContainers-solution-up-in-a-Windows-CLI-environment-\(dotnet-CLI,-Docker-CLI-and-VS-Code\)](https://github.com/dotnet/eShopOnContainers/wiki/03.-Setting-the-eShopOnContainers-solution-up-in-a-Windows-CLI-environment-(dotnet-CLI,-Docker-CLI-and-VS-Code))

使用容器运行数据库服务

您可以将数据库 (SQL Server、PostgreSQL、MySQL 等) 安装在常规独立服务器上, 或本地部署的集群中, 或在云端的 PaaS 服务中, 如 Azure SQL DB。但是, 对于开发和测试环境, 将数据库作为容器运行是方便的, 因为您没有任何外部依赖关系, 只需运行 docker-compose 命令即可启动整个应用程序。将这些数据库作为容器也非常适合集成测试, 因为数据库在容器中启动, 并且始终填充相同的样本数据, 因此测试是更可预测的。

将 SQL Server 作为具有微服务相关数据库的容器运行

在 eShopOnContainers 的 [docker-compose.yml](#) 文件中定义了一个名为 sql.data 的容器, 该容器运行 SQL Server for Linux, 其中包含微服务所需的所有 SQL Server 数据库。(您还可以为每个数据库分配一个 SQL Server 容器, 但这需要给 Docker 分配更多的内存。) 微服务的重点在于每个微服务都拥有与自身相关的数据, 在这种情况下即拥有相关的 SQL 数据库。但数据库可以部署在任何地方。

示例应用程序中的 SQL Server 容器由 docker-compose.yml 文件中的以下 YAML 代码进行配置, 该文件在运行 docker-compose up 时执行。请注意, YAML 代码已经从通用 docker-compose.yml 文件和 docker-compose.override.yml 文件中整合了配置信息。(通常您会将环境设置与 SQL Server 镜像相关的基础或静态信息分开。)

```
sql.data:  
  image: microsoft/mssql-server-linux  
  environment:  
    - MSSQL_SA_PASSWORD=Pass@word  
    - ACCEPT_EULA=Y  
    - MSSQL_PID=Developer  
  ports:  
    - "5434:1433"
```

以类似的方式, 使用以下 docker run 命令 (而不是使用 docker-compose) 可以运行该容器:

```
docker run -e 'ACCEPT_EULA=Y' -e 'SA_PASSWORD= your@password' -p 1433:1433 -d  
microsoft/mssql-server-linux
```

但是, 如果要部署多容器应用程序 (如 eShopOnContainers), 则使用 docker-compose up 命令更方便, 这样会为应用程序部署所有必需的容器。

首次启动此 SQL Server 容器时，容器将使用您提供的密码初始化 SQL Server。一旦 SQL Server 作为容器运行，您可以通过任何常规 SQL 连接（如 SQL Server Management Studio，Visual Studio 或 C# 代码）进行连接并更新数据库。

eShopOnContainers 应用程序在启动时通过填充种子数据，初始化每个微服务数据库中的示例数据，如下节所述。

将 SQL Server 作为容器运行不仅仅适用于您可能无法访问 SQL Server 实例的演示。如上所述，它也适用于开发和测试环境，因此您可以从干净的 SQL Server 镜像和填充示例种子数据后的已知数据开始，轻松运行集成测试。

其他资源

- **在 Linux、Mac 或 Windows 上运行 SQL Server Docker 镜像**
<https://docs.microsoft.com/sql/linux/sql-server-linux-setup-docker>
- **使用 sqlcmd 在 Linux 上连接和查询 SQL Server**
<https://docs.microsoft.com/sql/linux/sql-server-linux-connect-and-query-sqlcmd>

在 Web 应用程序启动时填充种子测试数据

要在应用程序启动时向数据库添加数据，您可以在 Web API 项目的 Startup 类中的 Configure 方法中添加如下代码：

```
public class Startup
{
    // Other Startup code...

    public void Configure(IApplicationBuilder app,
                        IHostingEnvironment env,
                        ILoggerFactory loggerFactory)
    {
        // Other Configure code...

        // Seed data through our custom class
        CatalogContextSeed.SeedAsync(app)
            .Wait();

        // Other Configure code...
    }
}
```

自定义的 CatalogContextSeed 类中的以下代码将会填充数据。

```
public class CatalogContextSeed
{
    public static async Task SeedAsync(IApplicationBuilder applicationBuilder)
    {
        var context = (CatalogContext)applicationBuilder
            .ApplicationServices.GetService(typeof(CatalogContext));
    }
}
```

```

using (context)
{
    context.Database.Migrate();

    if (!context.CatalogBrands.Any())
    {
        context.CatalogBrands.AddRange(
            GetPreconfiguredCatalogBrands());

        await context.SaveChangesAsync();
    }
    if (!context.CatalogTypes.Any())
    {
        context.CatalogTypes.AddRange(
            GetPreconfiguredCatalogTypes());

        await context.SaveChangesAsync();
    }
}
}
static IEnumerable<CatalogBrand> GetPreconfiguredCatalogBrands()
{
    return new List<CatalogBrand>()
    {
        new CatalogBrand() { Brand = "Azure"},
        new CatalogBrand() { Brand = ".NET" },
        new CatalogBrand() { Brand = "Visual Studio" },
        new CatalogBrand() { Brand = "SQL Server" }
    };
}

static IEnumerable<CatalogType> GetPreconfiguredCatalogTypes()
{
    return new List<CatalogType>()
    {
        new CatalogType() { Type = "Mug"},
        new CatalogType() { Type = "T-Shirt" },
        new CatalogType() { Type = "Backpack" },
        new CatalogType() { Type = "USB Memory Stick" }
    };
}
}
}

```

当您运行集成测试时，生成与您的集成测试一致的数据是很有用的。能够从头开始创建所有内容，包括在容器上运行的 SQL Server 实例，对于测试环境来说非常好。

EF Core InMemory 数据库与作为容器运行的 SQL Server

运行测试时的另一个好的选择是使用 Entity Framework InMemory 数据库提供程序。您可以在 Web API 项目中的 Startup 类的 ConfigureServices 方法中指定该配置：

```

public class Startup
{
    // Other Startup code ...

```

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IConfiguration>(Configuration);

    // DbContext using an InMemory database provider
    services.AddDbContext<CatalogContext>(opt => opt.UseInMemoryDatabase());

    //(Alternative: DbContext using a SQL Server provider
    //services.AddDbContext<CatalogContext>(c =>
    //{
    //    c.UseSqlServer(Configuration["ConnectionString"]);
    //});
}
// Other Startup code ...
}

```

但是有一个重要的问题。内存数据库不支持特定数据库的许多约束。例如，您也许要在 EF Core 模型的列上添加唯一索引，并对内存数据库进行测试，以检查它是否不允许您添加重复值。但是当您使用内存数据库时，您无法处理列上的唯一索引。因此，内存数据库的行为与真正的 SQL Server 数据库的行为并不完全一样 - 它不会模拟特定于数据库的约束。

即使如此，内存数据库仍然可用于测试和原型设计。但是，如果要创建考虑到特定数据库实现行为的精确集成测试，则需要使用像 SQL Server 这样的真实数据库。为此，在容器中运行 SQL Server 是一个更好的选择，并且比 EF Core InMemory 数据库更准确。

使用在容器中运行的 Redis 缓存服务

您可以在容器上运行 Redis，特别是用于开发和测试以及概念验证方案。这种情况很方便，因为您可以将所有依赖项运行在容器上，不仅适用于本地开发机器，还适用于您的 CI/CD 管道中的测试环境。

然而，当您在生产环境中运行 Redis 时，最好寻找像 Redis Microsoft Azure 这样的高可用性解决方案，它作为 PaaS（平台即服务）运行。在代码中，您只需要更改连接字符串即可。

Redis 提供了支持 Redis 的 Docker 镜像。该镜像可从 Docker Hub 在此 URL 处获得：

https://hub.docker.com/_/redis/

您可以通过在命令提示符下执行以下 Docker CLI 命令来直接运行 Docker Redis 容器：

```
docker run --name some-redis -d redis
```

Redis 镜像包括 expose:6379（Redis 使用的端口），因此标准的容器链接将对要链接的容器自动可用。

在 eShopOnContainers 中，basket.api 微服务使用作为容器运行的 Redis 缓存。该 basket.data 容器被定义为多容器 docker-compose.yml 文件的一部分，如下示例所示：

```
//docker-compose.yml file
```

```
//...
basket.data:
  image: redis
  expose:
    - "6379"
```

docker-compose.yml 中的此代码基于 redis 镜像定义了一个名为 basket.data 的容器，并在内部发布端口 6379，这意味着它只能从 Docker 主机中运行的其他容器访问。

最后，在 docker-compose.override.yml 文件中，eShopOnContainers 示例的 basket.api 微服务定义了用于该 Redis 容器的连接字符串：

```
basket.api:
  environment:
    // Other data ...
    - ConnectionString=basket.data
    - EventBusConnection=rabbitmq
```

在微服务（集成事件）之间实现基于事件的通信

如前所述，当您使用基于事件的通信时，微服务器会在事件发生时发布事件，如更新业务实体时。其他微服务订阅这些事件。当微服务器收到事件时，它可以更新自己的业务实体，这可能导致更多的事件被发布。该发布/订阅系统通常通过使用事件总线的实现来执行。事件总线可以设计为一个接口，包括订阅和取消订阅事件和发布事件所需的 API。它还可以具有基于任何进程间或消息通信的一个或多个实现，例如消息队列或支持异步通信以及发布/订阅模型的服务总线。

您可以使用事件来实现跨多个服务的业务事务，这使得这些服务之间保持最终一致性。最终一致性的事务由一系列分布式操作组成。在每个操作中，微服务更新业务实体并发布一个事件来触发下一个操作。

使用事件总线实现异步 事件驱动通信

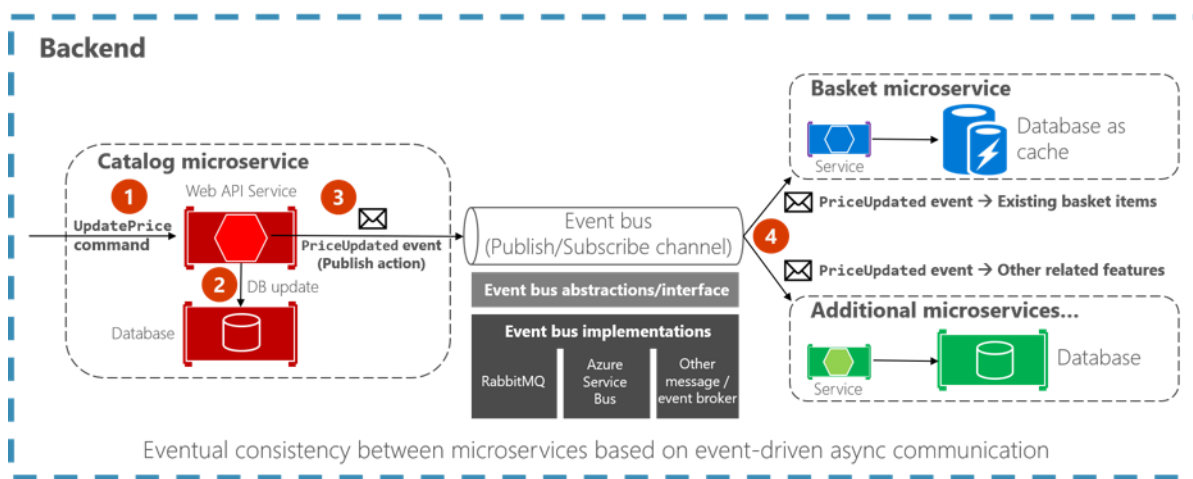


图 8-18. 基于事件总线的事件驱动通信

本节介绍如何使用通用事件总线接口在.NET 平台实现此类的通信，如图 8-18 所示。对此有多个可能的实现，分别不同的技术或基础架构，如 RabbitMQ，Azure Service Bus 或任何其他第三方开源或商业服务总线。

在生产环境中使用消息代理和服务总线

如架构部分所述，您可以从多种消息传递技术中选择实现抽象事件总线，但这些技术层次不同。例如 RabbitMQ 是一种消息传输代理，比商业产品如 Azure Service Bus、NServiceBus、MassTransit 或 Brighter 更底层一些。大多数这些产品可以在 RabbitMQ 或 Azure Service Bus 之上工作。您的产品选择取决于您的应用程序需要多少功能和多少开箱即用的可扩展性。

针对您的开发环境，为了实现概念验证性的事件总线，如 eShopOnContainers 示例中，将 RabbitMQ 之上的简单实现作为容器运行，可能已经足够了。但是对于需要高可扩展性的任务关键型和生产系统，您可能需要评估和使用 Azure 服务总线。

如果您需要高级抽象和更丰富的功能，如 [Sagas](#)，具有长期运行的进程使分布式开发更容易，其他商业和开源服务总线，如 NServiceBus、MassTransit 和 Brighter 也是值得评估的。在这种情况下，使用的抽象和 API 通常直接由那些高级服务总线提供，而不是你自己的抽象（比如 [eShopOnContainers 中提供的简单事件总线抽象](#)）。对于这个问题，您可以研究[使用 NServiceBus 的 eShopOnContainers 分支](#)（特定软件实现的其他派生示例）。

当然，您可以随时在诸如 RabbitMQ 和 Docker 等底层技术的基础上构建自己的服务总线功能，但是重新设计轮子所需的工作对于需要定制的企业应用来说可能成本太高了。

重申：eShopOnContainers 示例中展示的示例事件总线抽象和实现仅用作概念证明。一旦您决定要进行异步和事件驱动的通信，如本节所述，您应该选择最符合您生产环境需求的服务总线产品。

集成事件

集成事件用于跨多个微服务或外部系统使域状态同步。这是通过在微服务之外发布集成事件来实现的。当事件发布到多个接收微服务（与订阅集成事件相同的微服务）时，每个接收微服务中的相应事件处理程序将处理该事件。

集成事件基本上是一个数据保存类，如下例所示：

```
public class ProductPriceChangedIntegrationEvent : IntegrationEvent
{
    public int ProductId { get; private set; }
    public decimal NewPrice { get; private set; }
    public decimal OldPrice { get; private set; }

    public ProductPriceChangedIntegrationEvent(int productId, decimal newPrice,
                                                decimal oldPrice)
    {
        ProductId = productId;
        NewPrice = newPrice;
        OldPrice = oldPrice;
    }
}
```

集成事件可以在每个微服务的应用程序级进行定义，因此它们是与其它微服务解耦的，方式与服务器和客户端中的 ViewModel 的定义相当。不建议在多个微服务之间共享一个通用的集成事件库；这样做会将这些微服务与单个事件定义数据库相结合。不这样做的原因与您不想在多个微服务中共享通用的域模型相同：微服务必须完全自主。

只有几种类型的库，您应该在微服务中共享。一个是最终的应用程序块，像 [Event Bus 客户端 API](#) 一样，如 eShopOnContainers。另一个是可以作为 NuGet 组件共享的工具库，如 JSON 序列化程序。

事件总线

事件总线允许微服务之间的发布/订阅式通信，而不需要组件明确地了解彼此，如图 8-19 所示。

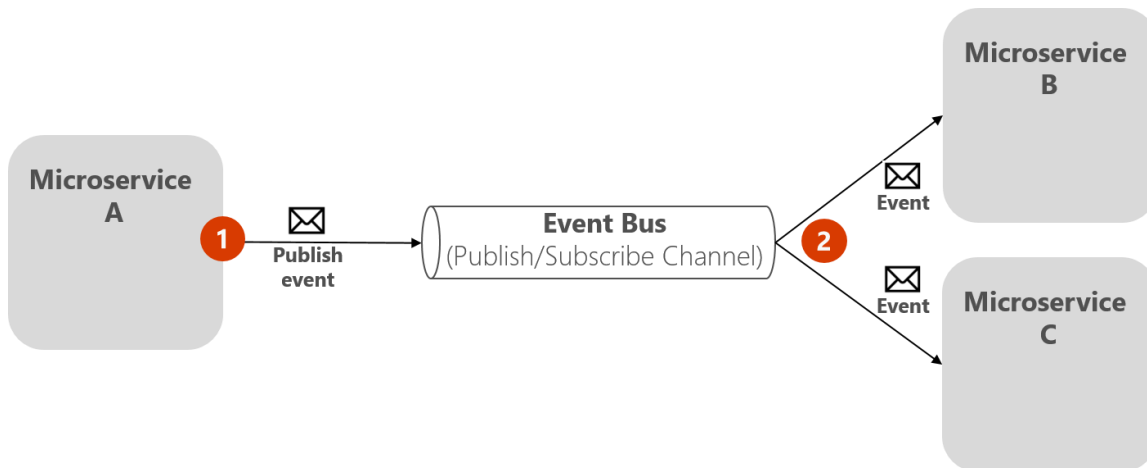


图 8-19. 使用事件总线发布/订阅的基础形式

事件总线与观察者模式和发布订阅模式相关。

观察者模式

在[观察者模式](#)中，您的主要对象（称为可观察的）可以用相关信息（事件）通知其他有兴趣的对象（称为观察者）。

发布-订阅 (Pub/Sub) 模式

[发布/订阅模式](#)的目的与观察者模式相同：您希望在发生某些事件时通知其他服务。但是观察者和发布/订阅模式之间存在重要的语义差异。在发布/订阅模式中，重点是广播消息。另外，在观察者模式中，Observable 并不知道事件将发送给谁，只是发布它们。换句话说，Observable（发布者）不知道观察者（订阅者）是谁。

中间人或事件总线

如何实现发布者和订阅者之间的匿名性？一个简单的方法是让一个中间人负责所有沟通。事件总线就是这样一个中间人。

事件总线通常由两部分组成：

- 抽象或接口。
- 一个或多个实现。

在图 8-19 中，您可以看到，从应用程序的角度来看，事件总线只不过是发布/订阅通道。实现此异步通信的方式可能有很多种。它可以有多个实现，以便您可以根据环境要求（例如，生产与开发环境）在它们之间进行切换。

在图 8-20 中，您可以看到一个事件总线的抽象，基于诸如 RabbitMQ、Azure Service Bus 或其他事件/消息代理，并具有多个实现。

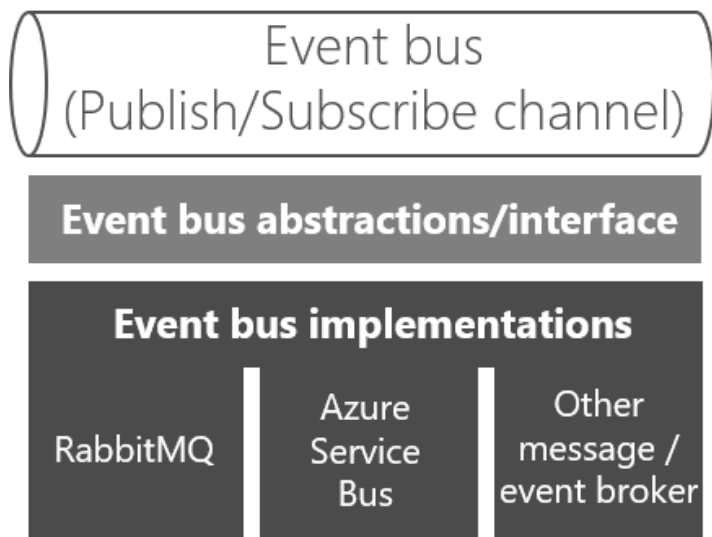


图 8-20. 事件总线的多种实现

然而，如前所述，只有当您需要被抽象支持的基本事件总线功能时，才能使用自己的抽象（事件总线接口）。如果您需要更丰富的服务总线功能，则应该使用由您首选的商用服务总线提供的 API 和抽象，而不是您自己的抽象。

定义事件总线接口

我们先从事件总线接口的一些实现代码开始进行探索。接口应该是通用和简单的，如下面的接口。

```
public interface IEventBus
{
    void Publish(IntegrationEvent @event);

    void Subscribe<T, TH>()
        where T : IntegrationEvent
        where TH : IIntegrationEventHandler<T>;

    void SubscribeDynamic<TH>(string eventName)
        where TH : IDynamicIntegrationEventHandler;

    void UnsubscribeDynamic<TH>(string eventName)
        where TH : IDynamicIntegrationEventHandler;

    void Unsubscribe<T, TH>()
        where TH : IIntegrationEventHandler<T>
```



```
    where T : IntegrationEvent;
}
```

Publish 方法很简单。事件总线将集成事件广播给订阅了该事件的任何微服务，甚至是外部应用程序。该方法由发布事件的微服务使用。

Subscribe 方法（取决于不同的参数，您可以使用多个实现）由想要接收事件的微服务使用。这个方法有两个参数。第一个是要订阅的集成事件（IntegrationEvent）。第二个参数是集成事件处理程序（或回调方法），名为 IIntegrationEventHandler<T>，当接收方微服务获取该集成事件消息时执行。

为开发或测试环境使用 RabbitMQ 实现事件总线

首先我们应该说，如果你创建一个基于运行在容器中的 RabbitMQ 的自定义事件总线，就像 eShopOnContainers 应用一样，它只能用于你的开发和测试环境。除非您将其作为可随时投产的服务总线的一部分进行构建，否则不应将其用于生产环境。简单的自定义事件总线可能缺少商业服务总线具有的许多可用于生产环境的关键功能。

eShopOnContainers 中一个事件总线的自定义实现基本上是一个使用 RabbitMQ API 的库（还有另一个基于 Azure Service Bus 的实现）。

该事件总线的实现使微服务订阅事件，发布事件和接收事件，如图 8-21 所示。

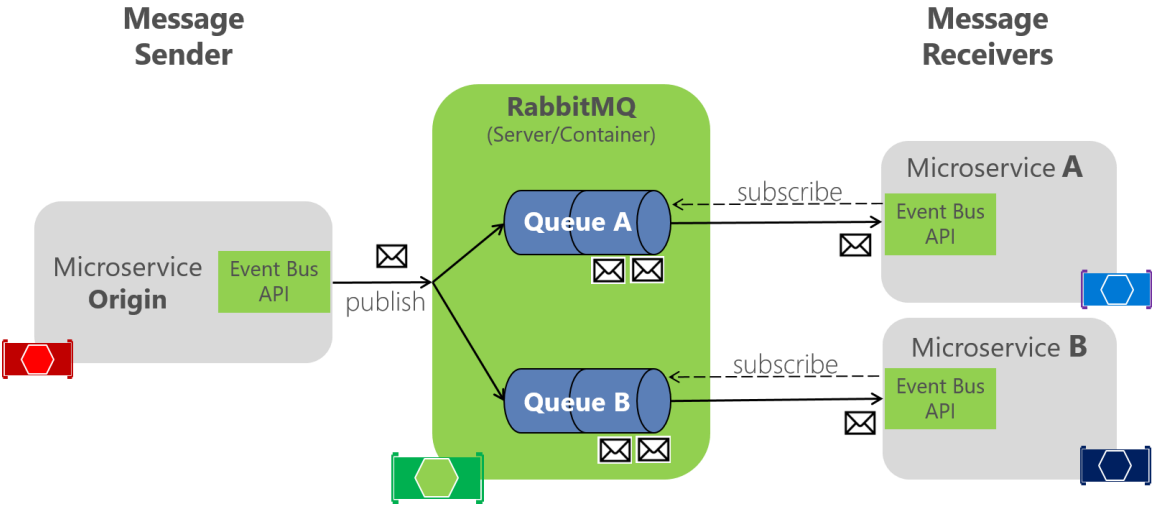


图 8-21. 使用 RabbitMQ 实现一个事件总线

在代码中，EventBusRabbitMQ 类实现了通用的 IEventBus 接口。这是基于依赖注入的，以便您可以从此开发/测试版本切换到生产版本。

```
public class EventBusRabbitMQ : IEventBus, IDisposable
{
    // Implementation using RabbitMQ API
    //...
```

示例开发/测试事件总线的 RabbitMQ 实现可以作为范例代码。它必须处理与 RabbitMQ 服务器的连接，并提供将消息事件发布到队列的代码。它还必须为每个事件类型实现一个字典，作为集成事件处理程序的集合；这些事件类型可以为每个接收者微服务，拥有不同的实例化和不同的订阅，如图 8-21 所示。

使用 RabbitMQ 实现简单的发布方法

以下代码是针对 RabbitMQ 的简化事件总线实现的一部分，在 eShopOnContainers 的[实际代码](#)中进行了改进。除非您需要改进，通常不需要修改这部分代码。代码先获取 RabbitMQ 的连接和通道，创建一个消息，然后将消息发布到队列中。

```
public class EventBusRabbitMQ : IEventBus, IDisposable
{
    // Member objects and other methods ...
    // ...
    public void Publish(IntegrationEvent @event)
    {
        var eventName = @event.GetType().Name;
        var factory = new ConnectionFactory() { HostName = _connectionString };
        using (var connection = factory.CreateConnection())
        using (var channel = connection.CreateModel())
        {
            channel.ExchangeDeclare(exchange: _brokerName,
                                   type: "direct");

            string message = JsonConvert.SerializeObject(@event);
            var body = Encoding.UTF8.GetBytes(message);

            channel.BasicPublish(exchange: _brokerName,
                                routingKey: eventName,
                                basicProperties: null,
                                body: body);
        }
    }
}
```

通过使用 [Polly](#) 重试策略，可以改善 eShopOnContainers 应用程序中 Publish 方法的[实际代码](#)，如果 RabbitMQ 容器未准备就绪，该策略会将任务重复一定次数。当使用 docker-compose 启动容器时，可能会发生这种情况，例如 RabbitMQ 容器可能比其他容器启动得慢。

如前所述，RabbitMQ 中有许多种可能的配置，因此该代码只能用于开发/测试环境。

使用 RabbitMQ API 实现订阅代码

与发布代码一样，以下代码简化了 RabbitMQ 事件总线实现的一部分。再次强调，您通常不需要改动这段代码，除非您需要改进。

```

public class EventBusRabbitMQ : IEventBus, IDisposable
{
    // Member objects and other methods ...
    // ...

    public void Subscribe<T, TH>()
        where T : IntegrationEvent
        where TH : IIntegrationEventHandler<T>
    {
        var eventName = _subsManager.GetEventKey<T>();

        var containsKey = _subsManager.HasSubscriptionsForEvent(eventName);
        if (!containsKey)
        {
            if (!_persistentConnection.IsConnected)
            {
                _persistentConnection.TryConnect();
            }

            using (var channel = _persistentConnection.CreateModel())
            {
                channel.QueueBind(queue: _queueName,
                                   exchange: BROKER_NAME,
                                   routingKey: eventName);
            }
        }

        _subsManager.AddSubscription<T, TH>();
    }
}

```

每个事件类型都有一个相关的通道来从 RabbitMQ 获取事件。然后，您可以根据需要为每个通道和事件类型设置尽可能多的事件处理程序。

Subscribe 方法接受一个 IIntegrationEventHandler 对象，类似当前微服务中的一个回调方法，以及它相关的 IntegrationEvent 对象。然后，该代码将该事件处理程序添加到一个事件处理程序列表，列表中包含每个客户端微服务中每个集成事件类型的事件处理程序。如果客户端代码尚未订阅该事件，则该代码将为该事件类型创建一个通道，以便当该事件从任何其他服务发布时，可以从 RabbitMQ 接收推送的事件。

订阅事件

使用事件总线的第一步是使微服务订阅他们想要接收的事件。这应该在接收者微服务中完成。

以下简单的代码显示了启动服务时（即在 Startup 类中）需要实现的每个接收者微服务，以便它订阅所需的事件。在这种情况下，basket.api 微服务需要订阅 ProductPriceChangedIntegrationEvent 和 OrderStartedIntegrationEvent 消息。

例如，当订阅 ProductPriceChangedIntegrationEvent 事件时，购物篮微服务会关注对产品价格的任何更改，并且如果该产品在用户的购物篮中，则可以向用户通知该更改。

```
var eventBus = app.ApplicationServices.GetRequiredService<IEventBus>();

eventBus.Subscribe<ProductPriceChangedIntegrationEvent,
    ProductPriceChangedIntegrationEventHandler>();

eventBus.Subscribe<OrderStartedIntegrationEvent,
    OrderStartedIntegrationEventHandler>();
```

该代码运行后，订阅者微服务将通过 RabbitMQ 通道进行监听。当任何类型为 ProductPriceChangedIntegrationEvent 的消息到达时，该代码将调用传递给它的事件处理程序并处理该事件。

通过事件总线发布事件

最后，消息发送方（原始微服务）使用与以下示例类似的代码发布集成事件。（这是一个不考虑原子性的简化示例。）每当事件必须跨多个微服务传播时，通常在从原始微服务提交数据或事务之后，您将实现类似的代码。

首先，事件总线实现对象（基于 RabbitMQ 或基于服务总线）将被注入控制器构造函数，如下面的代码：

```
[Route("api/v1/[controller]")]
public class CatalogController : ControllerBase
{
    private readonly CatalogContext _context;
    private readonly IOptionsSnapshot<Settings> _settings;
    private readonly IEventBus _eventBus;

    public CatalogController(CatalogContext context,
        IOptionsSnapshot<Settings> settings,
        IEventBus eventBus)
    {
        _context = context;
        _settings = settings;
        _eventBus = eventBus;
        // ...
    }
}
```

然后，您可以从控制器中的方法中使用它，如 UpdateProduct 方法：

```

[Route("update")]
[HttpPost]
public async Task<IActionResult> UpdateProduct([FromBody]CatalogItem product)
{
    var item = await _context.CatalogItems.SingleOrDefaultAsync(
                                                i => i.Id == product.Id);

    // ...
    if (item.Price != product.Price)
    {
        var oldPrice = item.Price;
        item.Price = product.Price;
        _context.CatalogItems.Update(item);

        var @event = new ProductPriceChangedIntegrationEvent(item.Id,
                                                                item.Price,
                                                                oldPrice);

        // Commit changes in original transaction
        await _context.SaveChangesAsync();

        // Publish integration event to the event bus
        // (RabbitMQ or a service bus underneath)
        _eventBus.Publish(@event);
        // ...
    }
}

```

在这种情况下，由于原始微服务是简单的 CRUD 微服务，因此该代码被直接放置在 Web API 控制器中。在更高级的微服务中，比如使用 CQRS 方法时，它可以在提交原始数据之后，在 CommandHandler 类中实现。

在发布到事件总线时设计原子性和弹性

当您通过分布式消息系统（如事件总线）发布集成事件时，您会遇到原子更新原始数据库和发布事件的问题。例如，在前面所示的简化示例中，当产品价格更改时，代码将数据提交给数据库，然后发布 ProductPriceChangedIntegrationEvent 消息。首先，这两个操作被原子性地执行可能是至关重要的。但是，如果您正在使用涉及数据库和消息代理的分布式事务，就像在[微软消息队列 \(MSMQ\)](#) 等旧系统中所做的那样，这并不是推荐的做法，[CAP 定理](#)描述了这一原因。

基本上，您使用微服务来构建可扩展和高可用性的系统。简单来说，CAP 定理认为，您无法构建持续可用，强一致和能够容忍任何分区的数据库（或拥有其模型的微服务器）。您必须选择这三个属性中的两个。

在基于微服务的体系结构中，您应该选择可用性和容忍度，不再强调强一致性。因此，在大多数现代的基于微服务的应用程序中，您通常不想在消息传递中使用[分布式事务](#)，就像在基于使用 [MSMQ](#) 的 Windows 分布式事务处理协调器 (DTC) 实现分布式事务时一样。

我们回到最初的问题及其例子。如果服务在数据库更新后崩溃（在这种情况下，紧跟在 `_context.SaveChangesAsync()` 代码行之后），但在集成事件发布之前，整个系统可能会变得不一致。这可能是关键的业务，取决于您正在处理的具体业务操作。

如前面在架构部分所述，您可以使用多种方法来处理此问题：

- 使用完整的[事件溯源模式](#)
- 使用[事务日志挖掘](#)
- 使用[发件箱模式](#)。这是一个用于存储集成事件（扩展本地事务）的事务表。

对于这种情况，使用完整的事件溯源（ES）模式如果不是最好的方法，也是最好的方法之一。但是，在许多应用场景中，您可能无法实现完整的 ES 系统。ES 意味着只将领域事件存储在事务数据库中，而不是存储当前的状态数据。仅存储领域事件可以带来很大的好处，例如，您将拥有可用的系统历史记录，并且能够随时确定系统在过去任意时刻的状态。然而，实施完整的 ES 系统需要重新构建系统的大部分，并引入许多其他的复杂性和要求。例如，您将要使用专门用于事件溯源的数据库，例如 [Event Store](#) 或文档数据库，如 Azure Document DB、MongoDB、Cassandra、CouchDB 或 RavenDB。ES 是解决这个问题一个很好的方法，但不是最简单的解决方案，除非你已经熟悉事件溯源模式。

使用事务日志挖掘的方案最初看起来是很透明的。但是，要使用这种方法，必须将微服务耦合到您的 RDBMS 事务日志，如 SQL Server 事务日志。这可能是不合适的。另一个缺点是记录在事务日志中的低级别更新可能与您的高级别集成事件并不在相同的级别上。如果是这样，那么对这些事务日志进行逆向工程分析的过程可能很困难。

一个均衡的方法是将事务数据库表和简化的 ES 模式组合使用。您可以使用诸如 “ready to publish the event” 的状态，当您将其提交到集成事件表时，它将在原始事件中设置。然后，尝试将事件发布到事件总线。如果发布事件动作成功，您将在原始服务中启动另一个事务，并将状态从 “ready to publish the event” 更新到 “event already published” 。

如果事件总线中的发布事件动作失败，数据在原始微服务中不会不一致 - 它仍被标记为 “ready to publish the event” ，而对于其余的服务，它将最终保持一致。您可以随时使用后台作业来检查事务或集成事件的状态。如果作业在 “ready to publish the event” 状态中发现事件，则可以尝试将该事件重新发布到事件总线。

请注意，通过这种方法，您只会保留每个原始微服务的集成事件，以及要与其他微服务或外部系统通信的事件。相反，在完整的 ES 系统中，您可以存储所有的领域事件。

因此，这种均衡的方法是一个简化的 ES 系统。您需要使用其当前状态（ “ready to publish” 与 “published” ）的集成事件列表。但是，您只需要为集成事件实现这些状态。在这种方法中，您不需要像完整的 ES 系统一样，将所有领域数据作为事件存储在事务数据库中。

如果您已经在使用关系数据库，则可以使用事务表来存储集成事件。要在应用程序中实现原子性，您可以使用基于本地事务的两步过程。基本上，您会在存储领域实体的数据库中有一个 IntegrationEvent 表。该表用作实现原子性的保险，以便将持久集成事件包括在提交领域数据的相同事务中。

循序渐进的步骤如下：应用程序开始本地数据库事务。然后更新领域实体的状态，并将事件插入到集成事件表中。最后提交事务，即实现所需的原子性。

在实施发布事件的步骤时，您有以下选择：

- 在提交事务后立即发布集成事件，并使用另一个本地事务将表中的事件标记为已发布。然后，在远程微服务发生问题的情况下，使用该表作为工件来跟踪集成事件，并根据存储的集成事件执行补救措施。
- 将表用作一种队列。使用单独的应用程序线程或进程查询集成事件表，将事件发布到事件总线，然后使用本地事务将事件标记为已发布。

图 8-22 显示了第一种方法的架构。

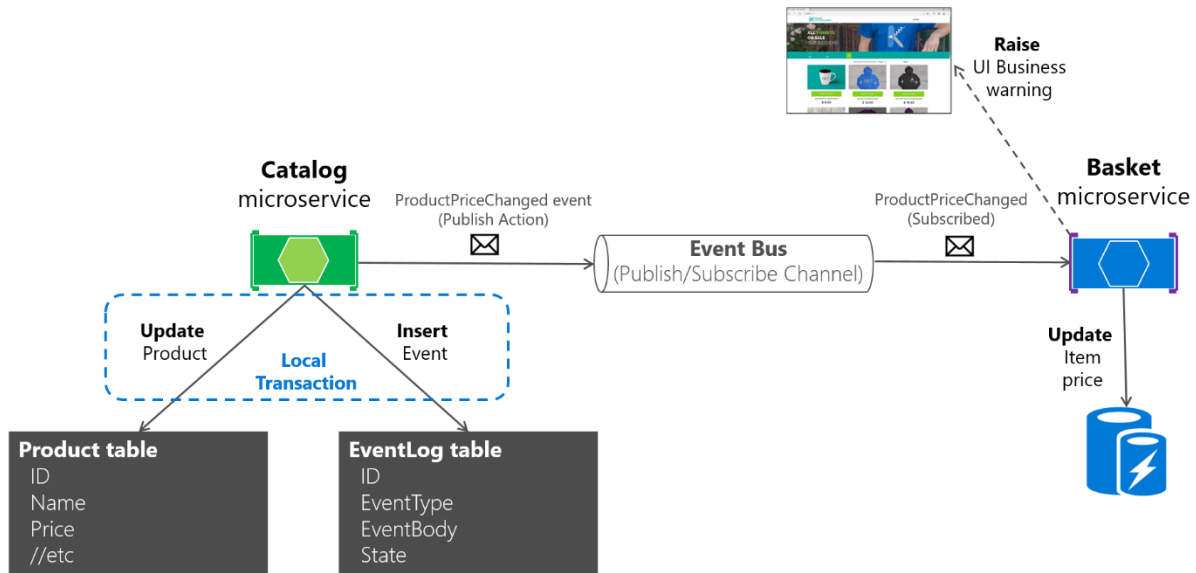


图 8-22. 将事件发布到事件总线时的原子性

图 8-22 所示的方法缺少一个附加的 Worker 微服务，负责检查和确认已发布的集成事件是否成功。如果发生故障，该附加的检查工作微服务可以从表中读取事件并重新发布。

关于第二种方法：将 EventLog 表用作队列，并始终使用一个 Worker 微服务来发布消息。在这种情况下，过程如图 8-23 所示。该图显示了一个附加的微服务，表是发布事件时的单一来源。

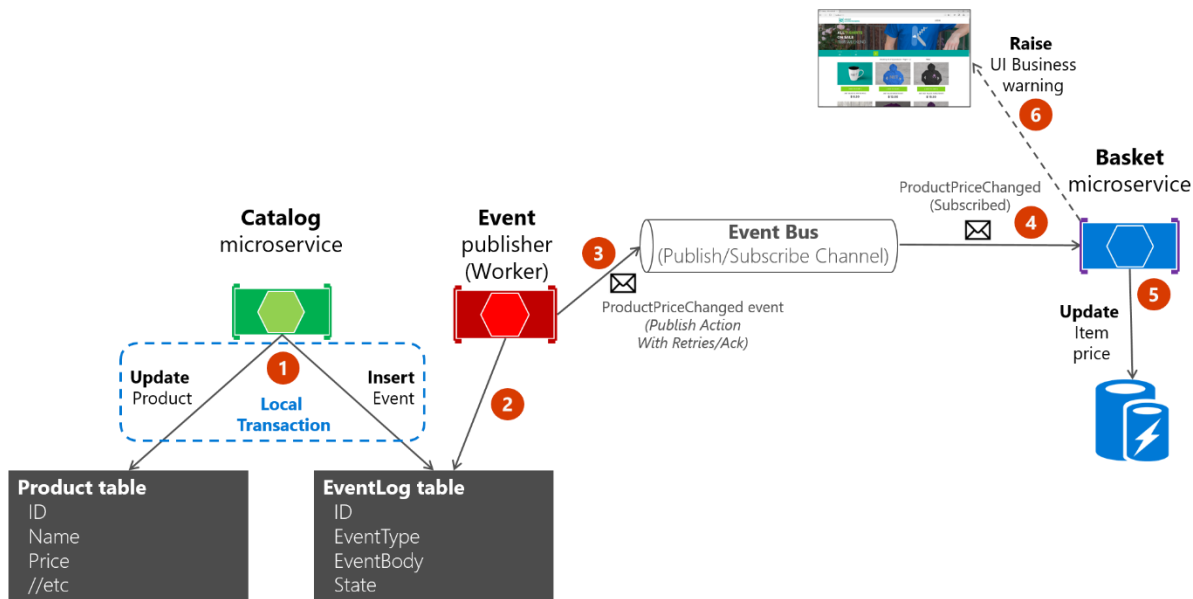


图 8-23. 使用一个 Worker 微服务将事件发布到事件总线时的原子性

为了简单起见，eShopOnContainers 示例使用第一种方法（没有其他进程或检查者微服务）以及事件总线。但是，eShopOnContainers 没有处理所有可能的故障情况。在部署到云端的实际应用中，您必须接受一个事实，即最终问题总会出现，您必须实现该检查和重发逻辑。如果将表作为单一事件来源，通过事件总线发布时，将表用作队列可以比第一种方法更有效。

通过事件总线发布集成事件时实现原子性

以下代码显示了如何创建涉及多个 DbContext 对象的单个事务 – 一个上下文与要更新的原始数据相关，第二个上下文与 IntegrationEventLog 表相关。

请注意，如果与数据库的连接在代码运行时有任何问题，下面示例代码中的事务将无法迅速恢复。这在基于云的系统可能会发生，如 Azure SQL DB，这可能会跨服务器移动数据库。要在多个上下文中实现弹性事务，请参阅本书后面的“[实现弹性 Entity Framework Core SQL 连接](#)”部分。

为了清楚起见，以下示例在一段代码中展示了整个过程。但是，eShopOnContainers 的实现实际上经过了重构，并将此逻辑分为多个类，以便更容易维护。


```

// Update Product from the Catalog microservice
//
public async Task<IActionResult> UpdateProduct([FromBody]CatalogItem
                                                productToUpdate)
{
    var catalogItem =
        await _catalogContext.CatalogItems.SingleOrDefaultAsync(i => i.Id ==
                                                                productToUpdate.Id);

    if (catalogItem == null) return NotFound();

    bool raiseProductPriceChangedEvent = false;
    IntegrationEvent priceChangedEvent = null;
    if (catalogItem.Price != productToUpdate.Price)
        raiseProductPriceChangedEvent = true;

    if (raiseProductPriceChangedEvent) // Create event if price has changed
    {
        var oldPrice = catalogItem.Price;
        priceChangedEvent = new ProductPriceChangedIntegrationEvent(catalogItem.Id,
                                                                    productToUpdate.Price,
                                                                    oldPrice);
    }
    // Update current product
    catalogItem = productToUpdate;

    // Just save the updated product if the Product's Price hasn't changed.
    if !(raiseProductPriceChangedEvent)
    {
        await _catalogContext.SaveChangesAsync();
    }
    else // Publish to event bus only if product price changed
    {
        // Achieving atomicity between original DB and the IntegrationEventLog
        // with a local transaction
        using (var transaction = _catalogContext.Database.BeginTransaction())
        {
            _catalogContext.CatalogItems.Update(catalogItem);
            await _catalogContext.SaveChangesAsync();

            // Save to EventLog only if product price changed
            if(raiseProductPriceChangedEvent)
                await
                _integrationEventLogService.SaveEventAsync(priceChangedEvent);

            transaction.Commit();
        }

        // Publish the intergation event through the event bus
        _eventBus.Publish(priceChangedEvent);

        integrationEventLogService.MarkEventAsPublishedAsync(
            priceChangedEvent);
    }

    return Ok();
}

```

在创建 ProductPriceChangedIntegrationEvent 集成事件之后，存储原始领域操作的事务（更新目录项）还包括事件在 EventLog 表中的持久性。这使它成为一个单一事务，您将始终能够检查事件消息是否已发送。

事件日志表与原始数据库操作进行原子更新，对同一数据库使用本地事务。如果有任何操作失败，将抛出异常，并且事务回滚任何已完成的操作，从而保持领域操作与发送的事件消息之间的一致性。

从订阅接收消息：接收者微服务中的事件处理程序

除了事件订阅逻辑之外，还需要实现集成事件处理程序的内部代码（如回调方法）。事件处理程序是指定某个类型的事件消息将被接收和处理的地方。

事件处理程序首先从事件总线接收事件实例。然后它定位与该集成事件相关的要处理的组件，传播该事件并持久化为接收者微服务中状态的变化。例如，如果 ProductPriceChanged 事件来源于目录微服务，则它将在购物篮微服务中处理，并更改此接收者购物篮微服务中的状态，如下面的代码所示。

```
//
//Integration-Event handler
//

Namespace Microsoft.eShopOnContainers.Services.Basket.
    API.IntegrationEvents.EventHandling
{
    public class ProductPriceChangedIntegrationEventHandler :
        IIntegrationEventHandler<ProductPriceChangedIntegrationEvent>
    {
        private readonly IBasketRepository _repository;

        public ProductPriceChangedIntegrationEventHandler(
            IBasketRepository repository)
        {
            _repository = repository ??
                throw new ArgumentNullException(nameof(repository));
        }

        public async Task Handle(ProductPriceChangedIntegrationEvent @event)
        {
            var userIds = await _repository.GetUsers();
            foreach (var id in userIds)
            {
                var basket = await _repository.GetBasket(id);
                await UpdatePriceInBasketItems(@event.ProductId, @event.NewPrice,
                    basket);
            }
        }

        private async Task UpdatePriceInBasketItems(int productId, decimal newPrice,
            CustomerBasket basket)
        {
            var itemsToUpdate = basket?.Items?.

```

```
        Where(x => int.Parse(x.ProductId) == productId).ToList();

    if (itemsToUpdate != null)
    {
        foreach (var item in itemsToUpdate)
        {
            if(item.UnitPrice != newPrice)
            {
                var originalPrice = item.UnitPrice;
                item.UnitPrice = newPrice;
                item.OldUnitPrice = originalPrice;
            }
        }

        await _repository.UpdateBasket(basket);
    }
}
}
```

事件处理程序需要验证产品是否存在于任何购物篮实例中。它还会更新每个相关购物篮订单项的价格。最后，它创建一个警告，向用户显示价格变化，如图 8-24 所示。

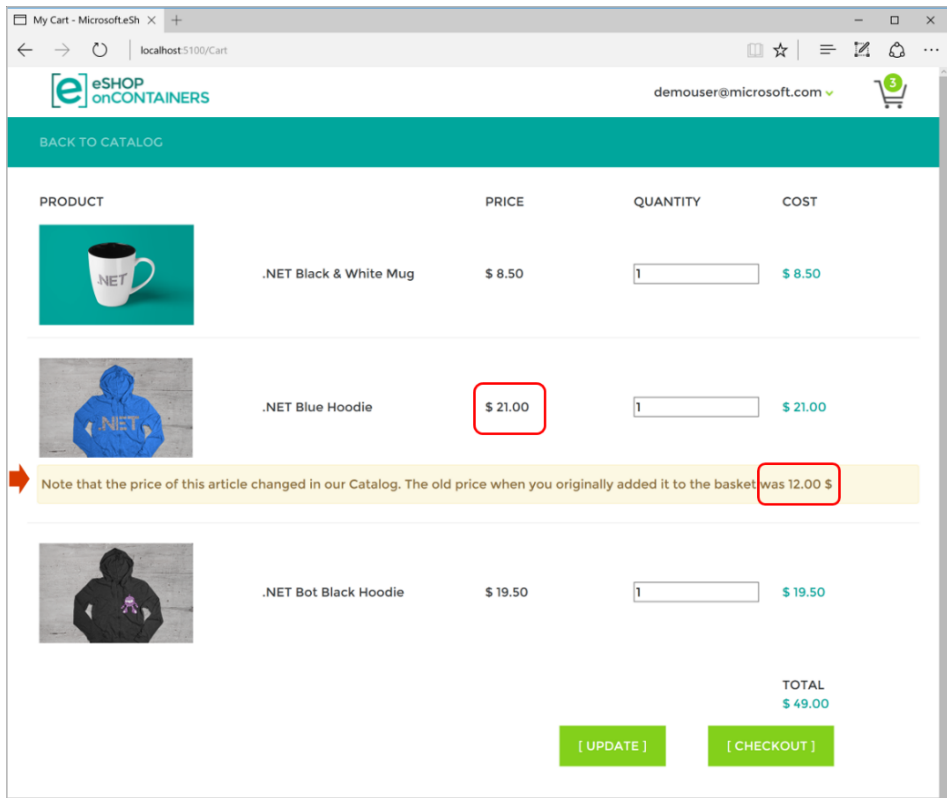


图 8-24. 通过集成事件通信来显示购物篮中的项目价格变动

更新消息事件中的幂等性

更新消息事件的一个重要方面是通信中的任何故障都应导致重新发送消息。否则后台任务可能会尝试发布已经发布的事件，导致一个竞争状态。您需要确保更新是幂等的，或者它们能提供足够的信息，以确保您可以检测到重复，丢弃它，并仅发送一个响应。

如前所述，幂等性意味着可以多次执行操作而不改变结果。在消息传递环境中，当传播事件时，如果可以多次传送事件而不改变接收者微服务的结果，则事件是幂等的。因为事件本身的性质，或者由于系统处理事件的方式，这可能是必需的。不仅仅在实现事件总线模式的应用程序中，在任何使用消息传递的应用程序中，消息幂等性都是很重要的。

幂等操作的一个例子是一个如下的 SQL 语句，该语句只有当表中不存在该数据时才将该数据插入到表中。运行该插入 SQL 语句的次数无关紧要；结果将是相同的 - 表将包含该数据。如果消息可能被发送，并因此被处理多次，那么在处理消息时也可能需要这样的幂等性。例如，如果重试逻辑导致发送者多次发送完全相同的消息，您需要确保它是幂等的。

可以设计幂等消息。例如，您可以创建一个表示“将产品价格设置为\$25”而不是“将产品价格增加\$5”的事件。您可以安全地处理第一条消息，无论多少次，结果都将一样。第二个消息不是这样。但即使在第一种情况下，您可能也不想处理第一个事件，因为系统也可能发送了较新的价格变动事件，而您将覆盖新的价格。

另一个例子可能是发送到多个订阅者的订单完成事件。重要的是，即使同一订单完成事件有重复的消息事件，也只需在其他系统中更新订单信息一次。

每个事件具有某种类型的标识是很方便的，您可以创建强制每个接收者对每个事件只处理一次的逻辑。

一些消息处理过程本质上是幂等的。例如，如果系统生成图像缩略图，不管处理生成缩略图的消息多少次，都可能不重要；结果就是生成缩略图，并且每次都相同。另一方面，诸如调用支付网关对信用卡收费的操作绝对不是等幂的。在这些情况下，您需要确保多次处理消息具有您期望的效果。

其他资源

- **尊重消息幂等性 (本页的小标题)**
<https://msdn.microsoft.com/library/jj591565.aspx>

删除重复的集成事件消息

您可以确保在不同层面上，每个订阅者仅发送和处理消息事件一次。一种方法是使用您正在使用的消息架构提供的重复数据删除功能。另一个是在您的目的地微服务中实现自定义逻辑。在传输层面和应用层面进行验证是最好的选择。

在 *EventHandler* 级别删除重复消息事件

要确保任何接收者只处理一次事件，一种方法是在事件处理程序中处理消息事件时实现某些逻辑。例如，这是 eShopOnContainers 应用程序中使用的方法，如 OrdersController 类收到 CreateOrderCommand 命令时的方法，您可以在 [OrdersController 类的源代码](#) 中看到。（在这种情况下，我们使用 HTTP 请求命令，而不是基于消息的命令，但是您需要创建基于消息的命令幂等性的逻辑是类似的。）

当使用 *RabbitMQ* 时删除重复消息

当发生间歇性网络故障时，消息有可能重复，消息接收者必须准备好处理这些重复的消息。如果可能，接收方应以幂等的方式处理消息，这比用删除重复数据的方式直接处理消息更好。

根据 [RabbitMQ 文档](#)，如果消息传递给消费者，然后重新排队（例如因为在消费者连接掉线之前未被确认），那么 RabbitMQ 会在再次传送时将其设置为 “redelivered” 标志（不管是相同的消费者或不同的消费者）。

如果设置了 “redelivered” 标志，接收方必须考虑到这一点，因为该消息可能已被处理。但这无法保证：消息可能在离开消息代理之后没有到达接收者，也许是因为网络问题。另一方面，如果没有设置 “redelivered” 标志，则该消息未被发送多次是可以保证的。因此，只有在消息中设置了 “redelivered” 标志时，接收方才需要以幂等方式对消息进行重复数据删除或处理消息。

其他资源

- **使用 NServiceBus (特定软件) 的 eShopOnContainers 分支**
<http://go.particular.net/eShopOnContainers>
- **事件驱动消息**
http://soapatterns.org/design_patterns/event_driven_messaging
- **Jimmy Bogard. 重构弹性：评估耦合**
<https://jimmybogard.com/refactoring-towards-resilience-evaluating-coupling/>
- **发布-订阅通道**
<http://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html>
- **限界上下文之间的通信**
<https://msdn.microsoft.com/library/jj591572.aspx>
- **最终一致性**
https://en.wikipedia.org/wiki/Eventual_consistency
- **Philip Brown. 集成限界上下文的策略**
<http://culttt.com/2014/11/26/strategies-integrating-bounded-contexts/>

- **Chris Richardson. 使用聚合，事件溯源和 CQRS 开发事务性微服务 - 第 2 部分**
<https://www.infoq.com/articles/microservices-aggregates-events-cqrs-part-2-richardson>
- **Chris Richardson. 事件溯源模式**
<http://microservices.io/patterns/data/event-sourcing.html>
- **事件溯源介绍**
<https://msdn.microsoft.com/library/jj591559.aspx>
- **Event Store 数据库。官方网站**
<https://geteventstore.com/>
- **Patrick Nommensen. 针对微服务的事件驱动数据管理**
<https://dzone.com/articles/event-driven-data-management-for-microservices-1>
- **CAP 定理**
https://en.wikipedia.org/wiki/CAP_theorem
- **什么是 CAP 定理?**
<https://www.quora.com/What-Is-CAP-Theorem-1>
- **数据一致性入门**
<https://msdn.microsoft.com/library/dn589800.aspx>
- **Rick Saling. CAP 定理：为什么在云和互联网中“一切都不一样”**
<https://blogs.msdn.microsoft.com/rickatmicrosoft/2013/01/03/the-cap-theorem-why-everything-is-different-with-the-cloud-and-internet/>
- **Eric Brewer. CAP 十二年后：“规则”如何改变**
<https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed>
- **参与外部 (DTC) 事务 (MSMQ)**
https://msdn.microsoft.com/library/ms978430.aspx#bdadotnetasync2_topic3c
- **Azure 服务总线。代理消息：重复检测**
<https://code.msdn.microsoft.com/Brokered-Messaging-c0acea25>
- **可靠性指南 (RabbitMQ 文档)**
<https://www.rabbitmq.com/reliability.html#consumer>

测试 ASP.NET Core 服务和 Web 应用

控制器是任何 ASP.NET Core API 服务和 ASP.NET MVC Web 应用程序的核心部分。因此，您应该相信控制器会按您的应用程序的预期正常工作。自动测试可以为您提供这种保证，并可以在部署到生产环境前检测错误。

您需要根据有效或无效的输入来测试控制器的运行情况，并根据其执行的业务操作的结果来测试控制器的响应。但是，对您微服务的测试应该有以下几种类型：

- 单元测试。这些测试确保应用程序的各个组件按预期工作。使用断言测试组件 API。
- 集成测试。这些测试确保组件交互按预期方式与诸如数据库的外部工件进行工作。使用断言可以测试组件 API、UI 或数据库 I/O，日志记录等操作的副作用。
- 为每个微服务进行功能测试。这些测试确保应用程序按照用户的预期运行。
- 服务测试。这些测试确保了端到端服务使用情况，包括同时测试多个服务的测试。对于这种类型的测试，您需要首先准备环境。在这种情况下，这意味着要启动服务（例如，通过使用 docker-compose up）。

实现 ASP.NET Core Web API 的单元测试

单元测试涉及测试应用程序的一部分，与其基础架构和依赖关系相隔离。单元测试用于测试控制器的逻辑时，只测试单个 Action 或方法的内容，而不是其依赖项或框架本身的行为。单元测试不会检测组件交互中的问题，那是集成测试的目的。

当您对您的控制器的 Action 进行单元测试时，请确保您只关注它们的行为。控制器单元测试可避免过滤器，路由或模型绑定之类的问题。因为他们只专注测试这一件事情，单元测试通常很容易开发，运行也很快。一套精心编写的单元测试可以经常运行，而不需要太多的开销。

单元测试是基于诸如 xUnit.net、MSTest、Moq 或 NUnit 之类的测试框架实现的。对于 eShopOnContainers 示例应用程序，我们正在使用 XUnit。

当您为 Web API 控制器编写单元测试时，您可以使用 C# 中的 new 关键字直接实例化控制器类，以便测试尽可能快地运行。以下示例显示了如何使用 [XUnit](#) 作为测试框架执行此操作。

```

[Fact]
public void Add_new_Order_raises_new_event()
{
    // Arrange
    var street = " FakeStreet ";
    var city = "FakeCity";
    // Other variables omitted for brevity ...

    // Act
    var fakeOrder = new Order(new Address(street, city, state, country, zipcode),
                                cardTypeId, cardNumber,
                                cardSecurityNumber, cardHolderName,
                                cardExpiration);

    // Assert
    Assert.Equal(fakeOrder.DomainEvents.Count, expectedResult);
}

```

为每个微服务实施集成和功能测试

如上所述，集成测试和功能测试具有不同的意义和目的。然而，在测试 ASP.NET Core 控制器时实现的方式是类似的，所以在本节中我们将重点关注集成测试。

集成测试确保应用程序的组件在集成时能够正常运行。ASP.NET Core 支持使用单元测试框架的集成测试和内置的测试 Web 主机，可用于处理请求而无需网络开销。

与单元测试不同，集成测试经常涉及应用程序基础架构问题，如数据库，文件系统，网络资源或 Web 请求和响应。单元测试使用虚拟或模拟对象代替这些关注点。但集成测试的目的是确认系统按照预期的方式与这些系统一起工作，因此对于集成测试，您不要使用虚拟的或模拟的对象。相反，您应包括基础设施，如数据库访问或其他服务的服务调用。

因为集成测试比单元测试执行更大的代码段，并且因为集成测试依赖于基础架构，所以它们往往比单元测试慢一个数量级。因此，对您编写和运行的集成测试加以限制是一个好的做法。

ASP.NET Core 包含一个内置的测试 Web 主机，可以用于处理 HTTP 请求而无需网络开销，这意味着您可以比使用真实的 Web 主机更快地运行这些测试。测试 Web 主机是名为 Microsoft.AspNetCore.TestHost 的 NuGet 组件。它可以添加到集成测试项目中，并用于托管 ASP.NET Core 应用程序。

如下面的代码所示，当您为 ASP.NET Core 控制器创建集成测试时，可以通过测试主机实例化控制器。这与 HTTP 请求相当，但运行速度更快。


```

public class PrimeWebDefaultRequestShould
{
    private readonly TestServer _server;
    private readonly HttpClient _client;
    public PrimeWebDefaultRequestShould()
    {
        // Arrange
        _server = new TestServer(new WebHostBuilder()
            .UseStartup<Startup>());
        _client = _server.CreateClient();
    }
    [Fact]
    public async Task ReturnHelloWorld()
    {
        // Act
        var response = await _client.GetAsync("/");
        response.EnsureSuccessStatusCode();

        var responseString = await response.Content.ReadAsStringAsync();

        // Assert
        Assert.Equal("Hello World!",
            responseString);
    }
}

```

其他资源

- **Steve Smith. 控制器测试 (ASP.NET Core)**
<https://docs.microsoft.com/aspnet/core/mvc/controllers/testing>
- **Steve Smith. 集成测试 (ASP.NET Core)**
<https://docs.microsoft.com/aspnet/core/testing/integration-testing>
- **在 .NET Core 中使用 dotnet test 的单元测试**
<https://docs.microsoft.com/dotnet/articles/core/testing/unit-testing-with-dotnet-test>

在多容器应用程序上实现服务测试

如前所述，当您测试多容器应用程序时，所有的微服务都需要在 Docker 主机或容器集群中运行。端到端的服务测试，包括多个微服务的多个操作，要求您通过运行 docker-compose 命令（或者其他类似机制，如果您正在使用协调器）在 Docker 主机中部署和启动整个应用程序。一旦整个应用程序及其所有的服务正在运行，您就可以执行端到端的集成和功能测试。

有几种方法可以使用。在用于部署应用程序（或类似 Docker-compose.ci.build.yml）的 docker-compose.yml 文件中，可以在解决方案级别扩展入口点以使用 [dotnet test](#)。您还可以使用另一个 compose 文件，以在您的目标镜像中运行测试。通过使用另一个 compose 文件进行集成测试，包括容器上的微服务和数据库，您可以确保在运行测试之前，相关数据始终重置为原始状态。

一旦组合应用程序启动并运行后，如果运行 Visual Studio，就可以使用断点和异常。或者您可以在 Visual Studio Team Services 或支持 Docker 容器的任何其他 CI/CD 系统的 CI 管道中自动运行集成测试。

使用 IHostedService 和 BackgroundService 类在微服务中实现后台任务

实际上，您可能需要在基于微服务的应用程序或任何类型的应用程序中实现后台任务和计划任务。使用微服务架构的区别在于，您可以实现一个单独的微服务进程/容器来托管这些后台任务，以便您可以根据需要进行缩放，或者，您甚至可以确保它只运行该微服务进程/容器的单个实例。

从一般的角度来看，在 .NET Core 中，我们将这些类型的任务称为 *托管服务*，因为它们是您在主机/应用程序/微服务中托管的服务/逻辑。请注意，在这种情况下，托管服务仅简单表示具有后台任务逻辑的类。

自 .NET Core 2.0 以来，框架提供了一个名为 `IHostedService` 的新接口，帮助您轻松实现托管服务。基本的想法是，您可以注册多个后台任务（托管服务），在您的 Web 主机或主机运行时，它们在后台运行，如图 8-25 所示。

使用 .NET Core 的 `IHostedService` 实现后台任务



图 8-25. 在 `WebHost` 与主机中使用 `IHostedService`

注意 `WebHost` 和 `Host` 之间的区别。

ASP.NET Core 2.0 中的 `WebHost`（实现 `IWebHost` 的基类）是用于为您的进程提供 Http 服务器功能的基础设施工件，例如，如果您正在实现 MVC Web 应用程序或 Web API 服务。它提供了 ASP.NET Core

中所有新的基础架构优势，使您能够使用依赖注入，在 Http 管道中插入中间件等，并精确地将这些 IHostedServices 用于后台任务。

但是，一个 Host（实现 IHost 的基类）是 .NET Core 2.1 中的新东西（截至撰写本文时，尚未发布）。基本上，Host 允许您有一个类似于 WebHost（依赖注入，托管服务等）的基础设施，但是在这种情况下，您只是希望有一个简单和轻量级的进程作为 Host，没有任何与 MVC，Web API 或 Http 服务器相关的功能。

因此，您可以选择并创建一个专门的主机进程来处理托管服务，而不是其他的什么，这样的微服务仅用于托管 IHostedServices，或者您可以换一种方式扩展现有的 ASP.NET Core WebHost，例如现有的 ASP.NET Core Web API 或 MVC 应用程序。每种方法都有优点和缺点，具体取决于您的业务和可扩展性需求。

在您的 WebHost 或 Host 中注册托管服务

让我们继续深入 IHostedService 接口，因为它在 WebHost 或 Host 中的使用非常相似。

SignalR 是使用托管服务工件的一个示例，但是您也可以使用它来执行更简单的操作，例如：

- 一个轮询数据库查找更改的后台任务
- 一个定期更新某个缓存的计划任务
- 一个 QueueBackgroundWorkItem 的实现，允许任务在后台线程上执行。
- 在 Web 应用程序的后台处理消息队列中的消息，同时共享 ILogger 等常用服务。

您基本上可以将这些操作转移到基于 IHostedService 的后台任务。

将一个或多个 IHostedServices 添加到 WebHost 或 Host 的方式是通过 ASP.NET Core WebHost（或 .NET Core 2.1 中的 Host）中的标准 DI（依赖注入）注册它们。基本上，您必须在 Startup 类中的 ConfigureServices() 方法中注册托管服务，如下面一个典型 ASP.NET WebHost 中的代码所示。

```
public IServiceCollection ConfigureServices(IServiceCollection services)
{
    //Other DI registrations;

    // Register Hosted Services
    services.AddSingleton<IHostedService, GracePeriodManagerService>();
    services.AddSingleton<IHostedService, MyHostedServiceB>();
    services.AddSingleton<IHostedService, MyHostedServiceC>();
    //...
}
```

在该代码中，GracePeriodManagerService 托管服务是来自 eShopOnContainers 中的订单业务微服务的实际代码，其他两个是另外两个样本。

IHostedService 后台任务的执行与应用程序的生命周期（主机或微服务，就此而言）是一致的。您在应用程序启动时注册任务，并且在应用程序关闭时有机会做一些优雅的操作或清理工作。

在不使用 IHostedService 的情况下，您总是可以启动后台线程来运行任何任务。不同之处在于应用程序关闭时，该线程会直接被终止，而没有机会运行正常的清理操作。

IHostedService 接口

当您注册 IHostedService 时，.NET Core 将分别在应用程序启动和停止期间调用您的 IHostedService 类型的 StartAsync() 和 StopAsync() 方法。具体而言，在服务器启动并启动 IApplicationLifetime.ApplicationStarted 之后调用 start。

在.NET Core 中定义的 IHostedService 如下所示。

```
namespace Microsoft.Extensions.Hosting
{
    //
    // Summary:
    //     Defines methods for objects that are managed by the host.
    public interface IHostedService
    {
        //
        // Summary:
        //     Triggered when the application host is ready to start the service.
        Task StartAsync(CancellationToken cancellationToken);
        //
        // Summary:
        //     Triggered when the application host is performing a graceful shutdown.
        Task StopAsync(CancellationToken cancellationToken);
    }
}
```

如您所想的那样，您可以创建 IHostedService 的多个实现，并在 ConfigureServices() 方法中将它们注册到 DI 容器中，如前所示。所有这些托管服务将随应用程序/微服务一起启动和停止。

作为开发人员，当主机触发 StopAsync() 方法时，您需要负责处理停止操作或服务。

使用从 BackgroundService 基类派生的自定义托管服务类来实现 IHostedService

您可以继续从头开始创建自定义托管服务类，并实现 IHostedService，在使用.NET Core 2.0 时您需要这样做。

但是，由于大多数后台任务在取消令牌管理和其他典型操作方便，都具有非常相似的需求，因此 .NET Core 2.1 将提供一个非常方便的抽象基类，名为 BackgroundService，您可以从该类派生自己的实现类。

。

该类提供了设置后台任务所需的主要工作。请注意，这个类来自 .NET Core 2.1 库，所以你不需编写它。

不过，截至本文撰写之时，.NET Core 2.1 尚未发布。因此，在目前使用 .NET Core 2.0 的 eShopOnContainers 中，我们只是从开源仓库中“窃取”了这个类，因为它与 .NET Core 2.0 中的当前 IHostedService 接口是兼容的。当 .NET Core 2.1 发布时，你只需要指向正确的 NuGet 包。

下面的代码是在 .NET Core 2.1 中实现的抽象 BackgroundService 基类。

```
// Copyright (c) .NET Foundation. Licensed under the Apache License, Version 2.0.
/// <summary>
/// Base class for implementing a long running <see cref="IHostedService"/>.
/// </summary>
public abstract class BackgroundService : IHostedService, IDisposable
{
    private Task _executingTask;
    private readonly CancellationTokenSource _stoppingCts =
        new CancellationTokenSource();

    protected abstract Task ExecuteAsync(CancellationToken stoppingToken);

    public virtual Task StartAsync(CancellationToken cancellationToken)
    {
        // Store the task we're executing
        _executingTask = ExecuteAsync(_stoppingCts.Token);

        // If the task is completed then return it,
        // this will bubble cancellation and failure to the caller
        if (_executingTask.IsCompleted)
        {
            return _executingTask;
        }

        // Otherwise it's running
        return Task.CompletedTask;
    }

    public virtual async Task StopAsync(CancellationToken cancellationToken)
    {
        // Stop called without start
        if (_executingTask == null)
        {
            return;
        }

        try
        {
            // Signal cancellation to the executing method
            _stoppingCts.Cancel();
        }
        finally
        {
            // Wait until the task completes or the stop token triggers
            await Task.WhenAny(_executingTask, Task.Delay(Timeout.Infinite,
                cancellationToken));
        }
    }
}
```

```

    }

    }

    public virtual void Dispose()
    {
        _stoppingCts.Cancel();
    }
}

```

当从前面的抽象基类派生的时候，得益于这个内置的实现，你只需要在你自己定制的托管服务类中实现 `ExecuteAsync()` 方法，就像下面的 `eShopOnContainers` 中的简单代码，在需要时轮询数据库并将集成事件发布到事件总线。

```

public class GracePeriodManagerService
    : BackgroundService
{
    private readonly ILogger<GracePeriodManagerService> _logger;
    private readonly OrderingBackgroundSettings _settings;

    private readonly IEventBus _eventBus;

    public GracePeriodManagerService(IOptions<OrderingBackgroundSettings> settings,
        IEventBus eventBus,
        ILogger<GracePeriodManagerService> logger)
    {
        //Constructor's parameters validations...
    }

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        _logger.LogDebug($"GracePeriodManagerService is starting.");

        stoppingToken.Register(() =>
            _logger.LogDebug($" GracePeriod background task is stopping."));

        while (!stoppingToken.IsCancellationRequested)
        {
            _logger.LogDebug($"GracePeriod task doing background work.");

            // This eShopOnContainers method is quering a database table
            // and publishing events into the Event Bus (RabbitMS / ServiceBus)
            CheckConfirmedGracePeriodOrders();

            await Task.Delay(_settings.CheckUpdateTime, stoppingToken);
        }

        _logger.LogDebug($"GracePeriod background task is stopping.");

        await Task.CompletedTask;
    }
}

```

```

protected override async Task StopAsync (Cancellation token stoppingToken)
{
    // Run your graceful clean-up actions
}
}

```

在 eShopOnContainers 这个特定情况下，它正在执行一个应用程序方法，来查询数据库表以查找具有特定状态的订单，并且在应用更改时，它通过事件总线（可以使用 RabbitMQ 或 Azure 服务总线）发布集成事件。

当然，您也可以运行其他任何业务后台任务。

默认情况下，取消令牌会设置为 5 秒超时，但也可以使用 IWebHostBuilder 的 UseShutdownTimeout 扩展在构建 WebHost 时更改该值。这意味着我们的服务应该在 5 秒内完成，否则会更有可能被直接关闭。

下面的代码将这个时间改变到 10 秒。

```

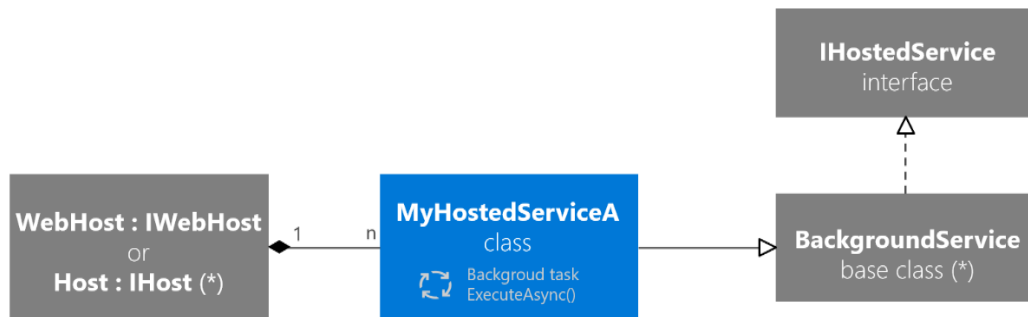
WebHost.CreateDefaultBuilder(args)
    .UseShutdownTimeout(TimeSpan.FromSeconds(10))
    ...

```

类图摘要

以下图 8-26 显示了实现 IHostedServices 时所涉及的类和接口的可视摘要。

自定义 IHostedService 和相关类和接口的类图



(**) IHost and IHostedService and BackgroundService are implemented in Microsoft.Extensions.Hosting only since .NET Core 2.1

图 8-26. 显示与 IHostedService 相关的多个类和接口的类图

结论

IHostedService 接口为 ASP.NET Core Web 应用程序（在 .NET Core 2.0 中）或任何进程/主机（从支持 IHost 的 .NET Core 2.1 开始）启动后台任务提供了一种便捷的方法。

它的主要好处是，当主机本身关闭时，您可以利用优雅的取消令牌来清理后台任务代码。**其他资源**

- **在 ASP.NET Core/Standard 2.0 中构建计划任务**
<https://blog.maartenballiauw.be/post/2017/08/01/building-a-scheduled-cache-updater-in-aspnet-core-2.html>
- **在 ASP.NET Core 2.0 中实现 IHostedService**
<https://www.stevejgordon.co.uk/asp-net-core-2-ihostedservice>
- **ASP.NET Core 2.1 Hosting 示例**
<https://github.com/aspnet/Hosting/tree/dev/samples/GenericHostSample>

在微服务中通过领域驱动设计 (DDD) 和命令查询职责分离 (CQRS) 应对业 务复杂性

愿景

为每个微服务或限界上下文设计领域模型反映了对业务领域的理解。

本章主要讨论更高级的微服务，比如需要处理复杂子系统的微服务，或者继承自领域专家不断变更的商业规则的微服务。本章讨论的架构模式是基于领域驱动设计 (DDD) 和命令查询职责分离 (CQRS) 方法，如图 9-1 所示。

应用程序外部架构 微服务内部架构

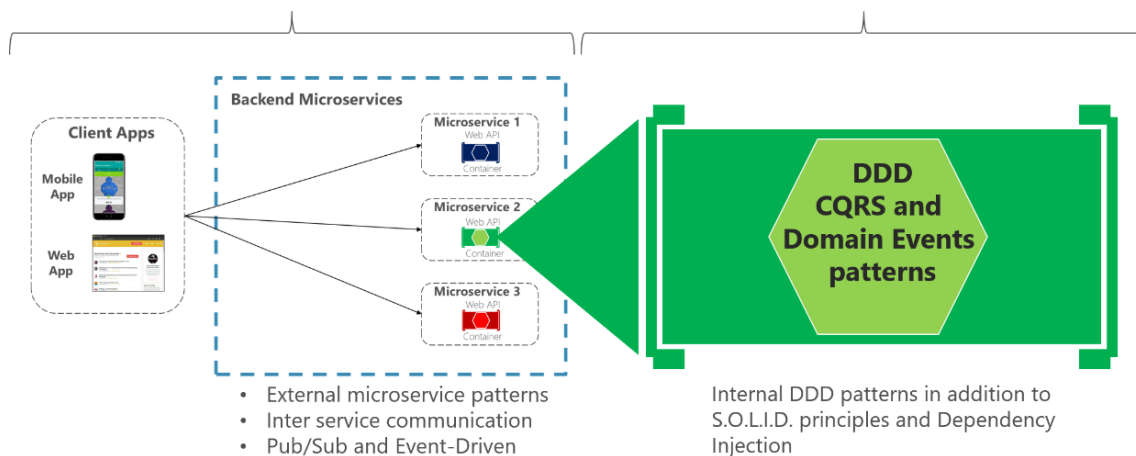


图 9-1. 应用外部微服务架构与微服务内部架构模式

然而，大部分数据驱动微服务所用的技术，比如如何实现 ASP.NET Core Web API 服务或如何用 Swashbuckle 暴露 Swagger 元数据等也同样适用于使用 DDD 模式作为内部实现的更高级的微服务。本章是前一章的扩展，因为大部分之前描述的实践也同样适用于本章，或者任何类型的微服务。

本章首先提供示例应用 eShopOnContainers 里的简化 CQRS 模式的详细情况。然后概要介绍 DDD 技术以便您可以找到通用的模式并能够用在自己的应用程序中。

DDD 是一个庞大的主题，拥有丰富的学习资源，您可以通过 Eric Evans 的书《[领域驱动设计](#)》以及来自 Vaughn Vernon, Jimmy Nilsson, Greg Young, Udi Dahan, Jimmy Bogard 以及很多其他 DDD/CQRS 专家的资料开始您的 DDD 学习旅程。但首先您需要尝试着学习如何从对话、白板、业务领域专家的领域建模会议中运用 DDD 技术。

其他资源

DDD (领域驱动设计)

- **Eric Evans. 领域语言**
<http://domainlanguage.com/>
- **Martin Fowler. 领域驱动设计**
<http://martinfowler.com/tags/domain%20driven%20design.html>
- **Jimmy Bogard. 强化您的领域：入门**
<https://lostechies.com/jimmybogard/2010/02/04/strengthening-your-domain-a-primer/>

DDD 书籍

- **Eric Evans. 领域驱动设计，软件核心复杂性应对之道**
<https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/>
- **Eric Evans. 领域驱动设计参考：定义和模式摘要**
<https://www.amazon.com/Domain-Driven-Design-Reference-Definitions-2014-09-22/dp/B01N8YB4ZO/>
- **Vaughn Vernon. 实现领域驱动设计**
<https://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon/dp/0321834577/>
- **Vaughn Vernon. 领域驱动设计精华**
<https://www.amazon.com/Domain-Driven-Design-Distilled-Vaughn-Vernon/dp/0134434420/>
- **Jimmy Nilsson. 应用领域驱动设计和模式**
<https://www.amazon.com/Applying-Domain-Driven-Design-Patterns-Examples/dp/0321268202/>
- **Cesar de la Torre. .NET 多层面面向领域的架构指南**
<https://www.amazon.com/N-Layered-Domain-Oriented-Architecture-Guide-NET/dp/8493903612/>
- **Abel Avram 和 Floyd Marinescu. 领域驱动设计快速教程**
<https://www.amazon.com/Domain-Driven-Design-Quickly-Abel-Avram/dp/1411609255/>
- **Scott Millett, Nick Tune - 领域驱动设计的模式、原则和实践**
<http://www.wrox.com/WileyCDA/WroxTitle/Patterns-Principles-and-Practices-of-Domain-Driven-Design.productCd-1118714709.html>

DDD 培训

- **Julie Lerman 和 Steve Smith. 领域驱动设计基础**
<http://bit.ly/PS-DDD>

在微服务中运用简化的 CQRS 和 DDD 模式

CQRS 是一种用来分离读写数据模型的架构模式。相关术语[命令查询分离 \(CQS\)](#) 原本是 Bertrand Meyer 在他的书籍《面向对象的软件结构》中定义的，其基本思想是将系统的操作划分成两个独立的类别：

- 查询：返回结果并且不改变系统的状态，即查询是没有副作用的。
- 命令：改变系统的状态。

CQS 是一个简单的概念 —— 同一个对象中作为查询或命令的方法，每个方法要么返回状态要么改变状态，但不能两者都有。甚至一个简单的仓储模式对象也可以遵循 CQS。CQS 可以看作是 CQRS 的基本原则。

[命令查询职责分离 \(CQRS\)](#) 由 Greg Young 提出，并被 Udi Dahan 等人强烈推动，它基于 CQS 原则但更加详细，可以看作是基于命令、事件以及加上可选的异步消息的模式。多数情况下，CQRS 和更高级的场景有关，比如有一个不同的物理数据库用来读取（查询）和另一个用来写入（更新）。此外，一个更先进的 CQRS 系统应该为更新数据库实现[事件溯源 \(ES\)](#)，这样您只需要在领域模型中存储事件，而不是存储当前状态的数据。但是在本书中不会用到这种方式，我们使用最简单的 CQRS 模式，即仅仅将查询和命令分离。

CQRS 的分离是将查询操作组合在一层，而命令在另一层。每一层拥有自己的数据模型（注：这里的模型没必要是另一个数据库），并且通过各自的模式和技术组合来构建。更重要的是，这两层可以包含在相同的物理层级或微服务里，如本书的示例应用里（Ordering 微服务）的用法。这两层也可以在不同的微服务或进程中实现，以便可以单独地优化和独立地横向扩展，避免彼此干扰。

CQRS 意味着有两个对象分别用于读/写操作，而其他方式只需要一个。有理由使用非规范化的只读数据库，这可以从更高级的 CQRS 文化中了解到，但是我们并没有在这里使用这种方法，其目标是在查询中拥有更多的灵活性，而不是像聚合一样限制来自 DDD 模式的约束。

这种服务的一个例子是示例应用 eShopOnContainers 的订单微服务。该服务基于简化的 CQRS 方法来实现，使用单个数据源或数据库，但是对于事务领域使用了两个逻辑模型和 DDD 模式，如图 9-2 所示。

简化的CQRS和DDD微服务 高层设计

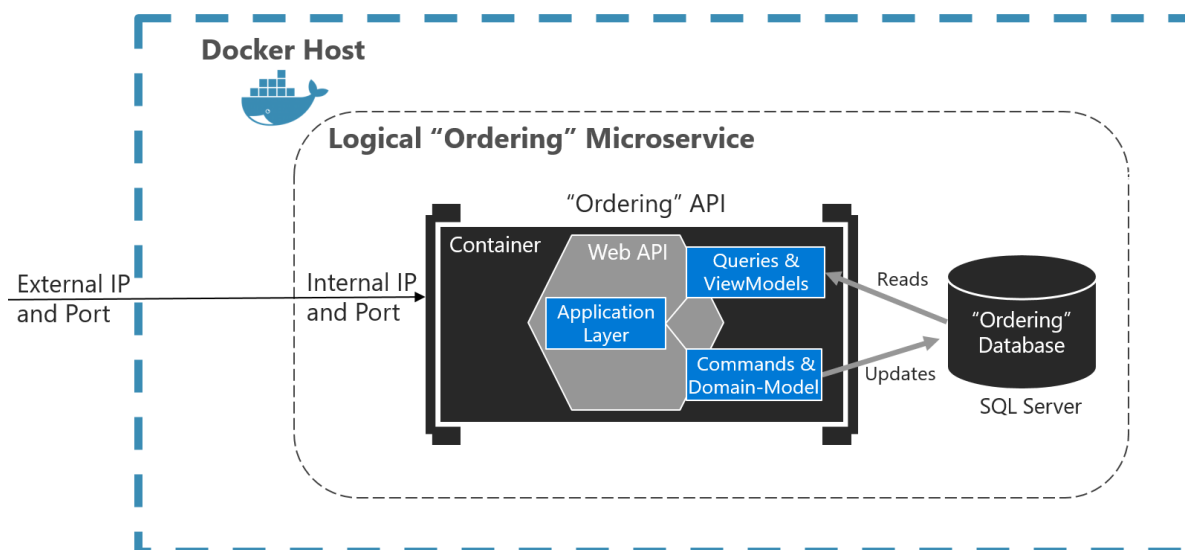


图9-2. 基于简化的CQRS和DDD的微服务

应用层可以是 Web API 本身，这里重要的设计角度是该微服务已经按照 CQRS 模式将查询和 ViewModel（专门为客户端应用创建的数据模型）跟命令、领域模型和事务拆分开了。这种方法使查询独立于来自 DDD 模式的限制和约束，这些模式只对事务和更新有意义，在后面的部分会讲述。

将 CQRS 和 CQS 方式应用到 eShopOnContainers 的 DDD 微服务中

示例应用 eShopOnContainers 中的订单微服务的设计是基于 CQRS 原则的，但使用了最简单的方式，即只是将查询和命令分离并且使用相同的数据库。

这些模式的本质和重点在于查询是幂等的：无论查询一个系统多少次，系统的状态都不会改变。即使订单微服务使用相同的数据库，也可以使用不同的“读取”数据模型而不是事务逻辑里“写入”领域模型。因此这是一个简化的 CQRS 方法。

另一方面，触发事务和数据更新的命令会改变系统的状态。使用命令需要小心处理系统复杂性和不断变化的业务规则。这就是您为了得到一个建模更好的系统而运用 DDD 技术的地方。

本书介绍的 DDD 模式不是普遍适用的，它们在您的设计中引入约束，这些约束提供了诸如随着时间的推移质量更高的益处，特别是在改变系统状态的命令和其他代码中。但是，这些约束增加了读取和查询数据的复杂性，并没有太多益处。

这样的模式之一是聚合模式，我们将在后面的章节中进一步讨论。简而言之，在聚合模式中，您将多个领域对象视为单个单位，作为其在领域中的关系的结果。对于查询，您可能不会总是从这种模式中获益，它增加了查询逻辑的复杂性。对于只读查询，您不会从将多个对象视为单个聚合中得到任何益处，只会得到复杂性。

如图 9-2 所示，本书建议仅仅在微服务的事务/更新区域（即由命令触发的）使用 DDD 模式。查询可以遵循更简单的方法，并且应该按照 CQRS 方法与命令分离。

为了实现“查询层”，您有多种方式可以选择，全面的 ORM 比如 EF Core、AutoMapper 映射、存储过程、视图、物化视图或者一个微型 ORM。

在本书和 eShopOnContainers（特别是订单微服务）中，我们选择使用像 [Dapper](#) 这样的微型 ORM 来实现直接查询。这使您可以实现任何基于 SQL 语句的查询以获得最佳性能，这要归功于这样一个开销很小的轻量级框架。

请注意，使用此方法时，任何影响实体如何持久化到 SQL 数据库的模型更新都需要单独更新 Dapper 用到的 SQL 查询或其他单独（非 EF）的查询。

CQRS 和 DDD 模式不是顶级架构

理解 CQRS 和大多数 DDD 模式（如 DDD 分层或带有聚合的领域模型）都不是架构风格而只是架构模式，这一点很重要。微服务、SOA 和事件驱动架构（EDA）是架构风格的例子，它们描述了由很多组件构成的系统，比如多个微服务。CQRS 和 DDD 模式则描述了单个系统或组件内的某些东西，在这里是指微服务内部的东西。

不同的限界上下文（BC）将采用不同的模式，它们有不同的职责，因此有不同的解决方案。值得强调的是，每种情况强行使用相同的模式将导致失败，不要在任何地方都使用 CQRS 和 DDD。很多子系统、限界上下文或微服务都比较简单，很容易用简单的 CRUD 服务或其他方式来实现。

只存在一种应用架构：您正在设计的系统或端到端应用程序（例如微服务架构）的架构。然而，该应用程序中的每个限界上下文或微服务的设计反映了其在架构模式级别下的自身权衡和内部的设计决策。不要尝试在任何地方运用相同的架构模式比如 CQRS 或 DDD。

其他资源

- **Martin Fowler. CQRS**
<https://martinfowler.com/bliki/CQRS.html>
- **Greg Young. CQS vs. CQRS**
<http://codebetter.com/gregyoung/2009/08/13/command-query-separation/>
- **Greg Young. CQRS 文档**
https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf
- **Greg Young. CQRS、基于任务的 UI 和事件溯源**
<http://codebetter.com/gregyoung/2010/02/16/cqrs-task-based-uis-event-sourcing-agh/>

- **Udi Dahan. 认清 CQRS**
<http://udidahan.com/2009/12/09/clarified-cqrs/>
- **CQRS**
<http://udidahan.com/2009/12/09/clarified-cqrs/>
- **事件溯源(ES)**
<http://codebetter.com/gregyoung/2010/02/20/why-use-event-sourcing/>

在 CQRS 微服务中实现读取/查询

对于读取/查询，示例应用 eShopOnContainers 的订单微服务的查询独立于 DDD 模型和事务来实现的。这主要是因为查询和事务的需求大不相同。写入执行的事务必须符合领域逻辑，另一方面，查询则是幂等的，可以从领域规则中分离出来。

方法很简单，如图 9-3 所示。API 接口由 Web API 控制器使用任何的基础架构（例如 Dapper 等微型 ORM）来实现，并根据 UI 应用的需要返回动态 ViewModel。

简化CQRS中的“查询侧”高层视图

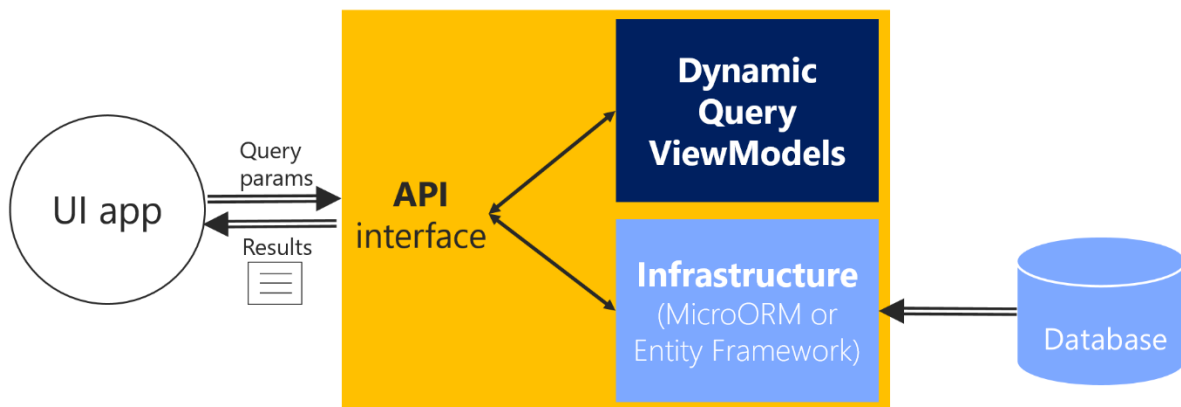


图9-3. 一个CQRS 微服务中最简单的查询

这是查询最简单的方法。查询的定义是查询数据库并返回为每个查询动态构建的 ViewModel。由于查询是幂等的，所以无论您运行多少次查询，它们都不会更改数据。因此，您不需要受到任何事务层面使用的 DDD 模式的限制，比如聚合和其他模式，这就是查询与事务分离的原因。您只需简单地查询数据库以获取 UI 所需的数据，然后返回一个动态 ViewModel 而不需要在任何位置明确定义（即没有为 ViewModel 定义类），除了在 SQL 语句中。

由于这是一个简单的方法，查询层所需的代码（例如使用 [Dapper](#) 等微型 ORM 的代码）可以在[同一个 Web API 项目中](#)实现。图 9-4 显示了这一点。查询是在 eShopOnContainers 解决方案中的 Ordering.API 微服务项目中定义的。

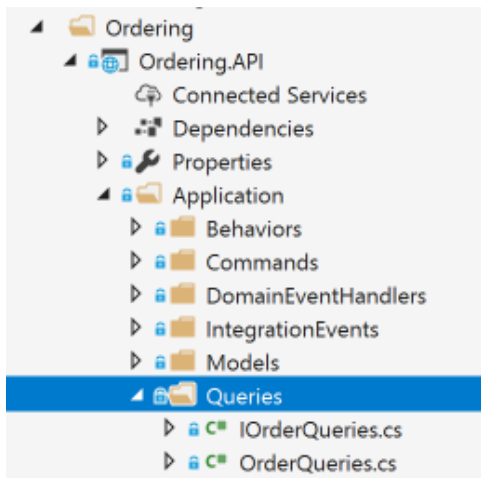


图9-4. eShopOnContainers 的订单微服务中的查询

使用独立于领域模型约束且专门为客户端应用创建的 ViewModel

由于查询是用来获取客户端应用所需的数据，因此可以根据查询返回的数据为客户端专门创建返回类型。这些模型或数据传输对象（DTO）称为 ViewModel。

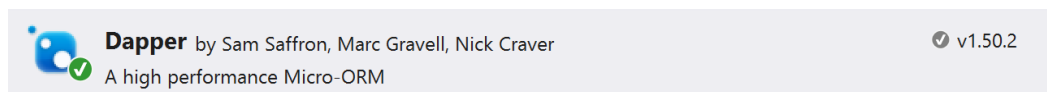
返回的数据（ViewModel）可以是连接数据库中多个实体或表的数据的结果，也可以是事务的领域模型中定义的一个或多个聚合的结果。在这种情况下，因为您正在创建独立于领域模型的查询，所以聚合的边界和约束完全被忽略，您可以自由地查询需要的任何表和任何列。这种方法为开发人员创建或更新查询提供了极大的灵活性和生产力。

ViewModel 可以在类中定义的静态类型，或者可以根据执行的查询来动态创建它们（就像在订单微服务中实现的一样），这对开发人员来说非常的灵活。

使用 Dapper 作为微型 ORM 实现查询

您可以使用任何微型 ORM、Entity Framework Core，甚至纯 ADO.NET 来进行查询。在示例应用中，我们选择了 Dapper 作为 eShopOnContainers 中的订单微服务，作为流行微型 ORM 的一个很好的例子。它能以很好的性能来运行纯 SQL 查询，因为它是一个非常轻量的框架。使用 Dapper 可以编写一个访问和连接多个表的 SQL 查询。

Dapper 是一个开源项目（最初由 Sam Saffron 创建），并且是 [Stack Overflow](#) 使用的构建块的一部分。要使用 Dapper，只需要通过 [Dapper 的 NuGet 包](#) 安装它，如下图所示。



还需要添加一个 using 语句，以便您的代码可以访问 Dapper 的扩展方法。

当您在代码中使用 Dapper 时，直接使用 System.Data.SqlClient 命名空间中的 SqlClient 类，通过 QueryAsync 方法和 SqlClient 类的其他扩展方法，可以简单直观地运行查询。

动态与静态的 ViewModel

将 ViewModel 从服务器端返回到客户端应用时，您可以将这些 ViewModel 视为 DTO（数据传输对象），它们可能与实体模型的内部领域实体不同，因为 ViewModel 按客户端应用需要的方式保存数据。因此，在多数情况下，您可以汇总来自多个领域实体的数据，并根据客户端应用需要的数据精确地组合 ViewModel。

那些 ViewModel 或 DTO 可以被显式地定义（像数据持久化类），就像后面代码片段中的 OrderSummary 类一样，或者您可以简单地根据查询返回的属性返回动态的 ViewModel 或 DTO。

动态类型作为 ViewModel

如下列代码所示，ViewModel 可以通过查询直接返回，查询内部基于获取的属性简单地生成一个动态类型。这意味着要返回的属性子集是基于查询本身的。如果您向查询或表连接新增一列，则该数据将动态添加到返回的 ViewModel 中。

优点：这种方式减少了只要更新查询的 SQL 语句时就要修改静态 ViewModel 类的需要，使得这种设计方式在编码时非常敏捷、直截了当和快速响应未来的变化。

```
using Dapper;
using Microsoft.Extensions.Configuration;
using System.Data.SqlClient;
using System.Threading.Tasks;
using System.Dynamic;
using System.Collections.Generic;
public class OrderQueries : IOrderQueries
{
    public async Task<IEnumerable<dynamic>> GetOrdersAsync()
    {
        using (var connection = new SqlConnection(_connectionString))
        {
            connection.Open();
            return await connection.QueryAsync<dynamic>(@"SELECT o.[Id] as ordernumber,
                o.[OrderDate] as [date],os.[Name] as [status],
                SUM(oi.units*oi.unitprice) as total
                FROM [ordering].[Orders] o
                LEFT JOIN[ordering].[orderitems] oi ON o.Id = oi.orderid
                LEFT JOIN[ordering].[orderstatus] os on o.OrderStatusId = os.Id
                GROUP BY o.[Id], o.[OrderDate], os.[Name]");
        }
    }
}
```

重点在于通过使用动态类型，返回的数据集合将被动态地组合为 ViewModel。

缺点：从长远来看，动态类型会降低清晰度甚至影响服务和客户端应用的兼容性。另外，如果使用动态类型，像 Swagger 等中间件软件就无法基于返回类型提供相同层次的文档。

预先定义的 DTO 类作为 ViewModel

优点：使用静态的预先确定的 ViewModel 类，就像基于明确 DTO 类的“契约”，对于公开的 API 和长期的微服务明显是更好的，即使它们用在同一个应用中。

如果您要为 Swagger 指定响应的类型，就需要使用明确的 DTO 类作为返回类型。因此，预先确定的 DTO 类允许您从 Swagger 里提供更丰富的信息。这对于 API 文档和调用 API 时的兼容性都有好处。

缺点：如前所述，当更新代码时，需要更多的步骤来更新 DTO 类。

我们的经验之谈：在 eShopOnContainers 的订单微服务里实现的查询中，我们起初使用动态的 ViewModel 来开发，因为非常简单和敏捷。然而，一旦开发稳定了，基于前面描述的原因，我们选择重构这个主题并为 ViewModel 使用静态的或预先确定的 DTO 类。

在下面的代码中，您能看到该场景下查询是如何通过使用明确的 ViewModel DTO 类来返回数据的，即 OrderSummary 类：

```
using Dapper;
using Microsoft.Extensions.Configuration;
using System.Data.SqlClient;
using System.Threading.Tasks;
using System.Dynamic;
using System.Collections.Generic;

public class OrderQueries : IOrderQueries
{
    public async Task<IEnumerable<OrderSummary>> GetOrdersAsync()
    {
        using (var connection = new SqlConnection(_connectionString))
        {
            connection.Open();
            var result = await connection.QueryAsync<dynamic>(
                @"SELECT o.[Id] as ordernumber,
                o.[OrderDate] as [date],os.[Name] as [status],
                SUM(oi.units*oi.unitprice) as total
                FROM [ordering].[Orders] o
                LEFT JOIN[ordering].[orderitems] oi ON o.Id = oi.orderid
                LEFT JOIN[ordering].[orderstatus] os on o.OrderStatusId = os.Id
                GROUP BY o.[Id], o.[OrderDate], os.[Name]
                ORDER BY o.[Id]");
        }
    }
}
```

描述 Web API 的响应类型

开发人员调用 Web API 和微服务最关心的是返回结果是什么——尤其是响应类型和错误码（如果不是标准的）。这些是通过 XML 注释和数据标注来处理的。

如果在 Swagger UI 里没有合适的文档，调用者就缺少返回什么类型或返回什么 HTTP 编码的信息。这个问题通过添加 `ProducesResponseType` 特性（定义在 `Microsoft.AspNetCore.Mvc` 中）来解决，因此 Swagger 会生成返回类型和值的丰富信息，如下列代码所示：

```
namespace Microsoft.eShopOnContainers.Services.Ordering.API.Controllers
{
    [Route("api/v1/[controller]")]
    [Authorize]
    public class OrdersController : Controller
    {
        //Additional code...
        [Route("")]
        [HttpGet]
        [ProducesResponseType(typeof(IEnumerable<OrderSummary>),
            (int)HttpStatusCode.OK)]
        public async Task<IActionResult> GetOrders()
        {
            var orderTask = _orderQueries.GetOrdersAsync();
            var orders = await orderTask;
            return Ok(orders);
        }
    }
}
```

然而，`ProducesResponseType` 特性无法使用 `dynamic` 作为类型，必须使用明确的类型比如 `OrderSummary` 这样的 ViewModel DTO，如下列代码片段所示：

```
public class OrderSummary
{
    public int ordernumber { get; set; }
    public DateTime date { get; set; }
    public string status { get; set; }
    public double total { get; set; }
}
```

长期来看，这是为什么明确的返回类型比动态类型要好的另一个原因。

当使用 `ProducesResponseType` 特性时，您也可以根据可能的 HTTP 错误或代码指定期望的输出，比如 200, 400 等。

如下图所示，您能看到 Swagger UI 是如何展示响应类型信息的。

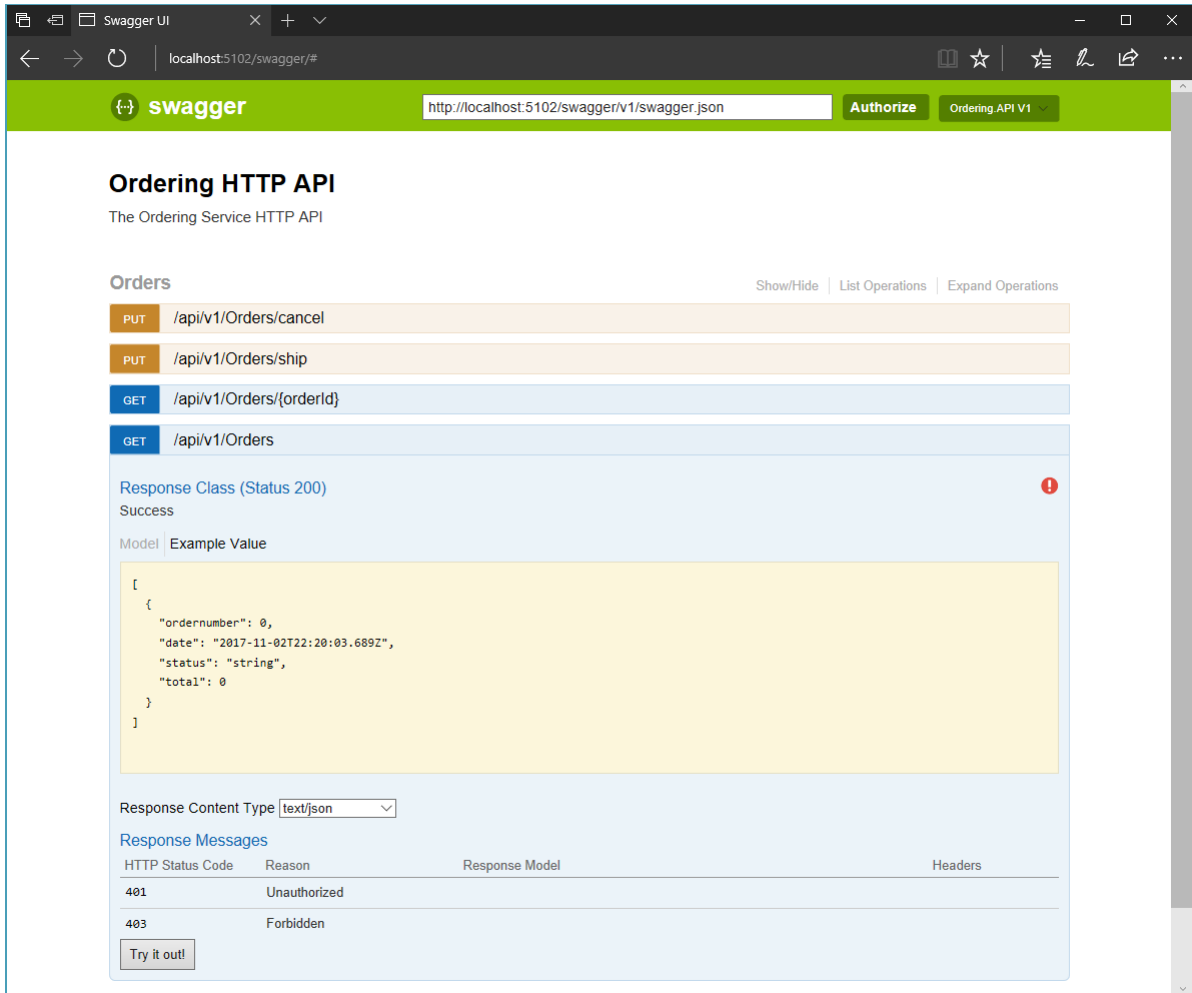


图9-5. Swagger UI 展示一个 Web API 的响应类型和可能的 HTTP 状态码

在上图中您还可以看到一些示例值，它们基于 ViewModel 类型和可能返回的 HTTP 状态码。

其他资源

- **Dapper**
<https://github.com/StackExchange/dapper-dot-net>
- **Julie Lerman. Data Points - Dapper, Entity Framework 和混合应用程序 (MSDN 杂志文章)**
<https://msdn.microsoft.com/magazine/mt703432.aspx>
- **使用 Swagger 生成 ASP.NET Core Web API 的帮助页面**
<https://docs.microsoft.com/en-us/aspnet/core/tutorials/web-api-help-pages-using-swagger?tabs=visual-studio>

设计面向 DDD 的微服务

领域驱动设计 (DDD) 主张根据和用例相关的实际业务来建模。在构建应用程序的上下文中, DDD 将问题作为领域, 它把独立的问题域看作限界上下文 (每一个限界上下文跟一个微服务有关), 并突出一种说明这些问题的通用语言。它也提出多种技术概念和模式来支持内部的实现, 例如丰富模型领域实体 (非贫血领域模型)、值对象、聚合和聚合根 (或根实体) 规则。本节介绍这些内部模式的设计和实现。

有时候, 这些 DDD 技术规则和模式被看作是障碍, 因为在实现 DDD 方法时有陡峭的学习曲线。但重点不在于模式本身, 而是组织代码以便划清业务问题并使用相同的业务术语 (统一语言)。此外, DDD 方式应该只用于实现拥有重要业务规则的复杂微服务。比较简单的一些任务例如 CRUD 服务可以使用更简单的方法进行管理。

如何确定边界是设计和定义微服务的关键任务。DDD 模式帮助您理解领域中的复杂性。对于每个限界上下文中的领域模型, 识别和定义用来模型化该领域的实体、值对象和聚合。构建和提炼出在一个定义上下文的边界里的领域模型。很明显这是微服务的形式。那些边界内的组件最终成为微服务, 即使在某些情况下, 一个 BC 或业务微服务可以由多个物理的服务组成。DDD 和微服务都是关于边界的。

保持相对较小的微服务上下文边界

决定在限界上下文之间划分边界要平衡两个相互竞争的目标。首先, 最开始需要创建尽可能小的微服务, 尽管不应该是主要的意图, 但也应该为需要内聚的事物建立一个边界。其次, 需要避免微服务间无谓的通信。这两个目标互相制约, 您需要权衡它们, 尽可能地将系统拆分成多个小型微服务, 直到尝试拆分一个新的限界上下文而出现快速增长的边界通信。内聚是一个独立的限界上下文的关键。

这跟实现类时的不当亲密关系的代码习惯很类似。如果两个微服务需要很多的彼此协作, 它们应该成为同一个微服务。

另一种看待这种情况的方法是自主权。如果一个微服务必须依靠其他服务来直接处理请求, 那么它就不是真正的自治。

DDD 微服务中的分层

大多数具有重要业务和技术复杂性的企业应用都定义了多个层级。层级是一种逻辑体, 跟服务的部署无关, 它们的存在是为了帮助开发人员管理代码中的复杂性。不同的层 (比如领域模型层和表现层等) 可能有不同的类型, 因此在这些类型之间需要转义。

举个例子, 一个实体可能从数据库中加载, 然后实体的部分信息或包含其他实体的额外数据的一个信息聚合, 能通过一个 REST Web API 发送到客户端 UI。这里的要点是领域实体包含在领域模型层内部, 并且不应该传送到不属于它的其他区域, 比如表现层。

此外，您需要由聚合根（根实体）控制的始终有效的实体（详见“[设计领域模型层的验证](#)”一节），因此实体不应该跟客户端视图绑定，因为在 UI 层级的某些数据可能未经过验证。这就是 ViewModel 的用武之地，ViewModel 是一个仅仅为满足表现层需要的数据模型。领域实体不是直接属于 ViewModel，相反，您需要在 ViewModel 和领域实体之间相互转换。

当应对复杂性时，重点是要有一个由聚合根（稍后会详细介绍）控制的领域模型，以确保与该组实体（聚合）相关的所有不变量和规则都通过一个单独的入口点或门来进行，即聚合根。

图 9-5 显示了 eShopOnContainers 应用中是如何实现分层设计的。

领域驱动设计微服务里的分层

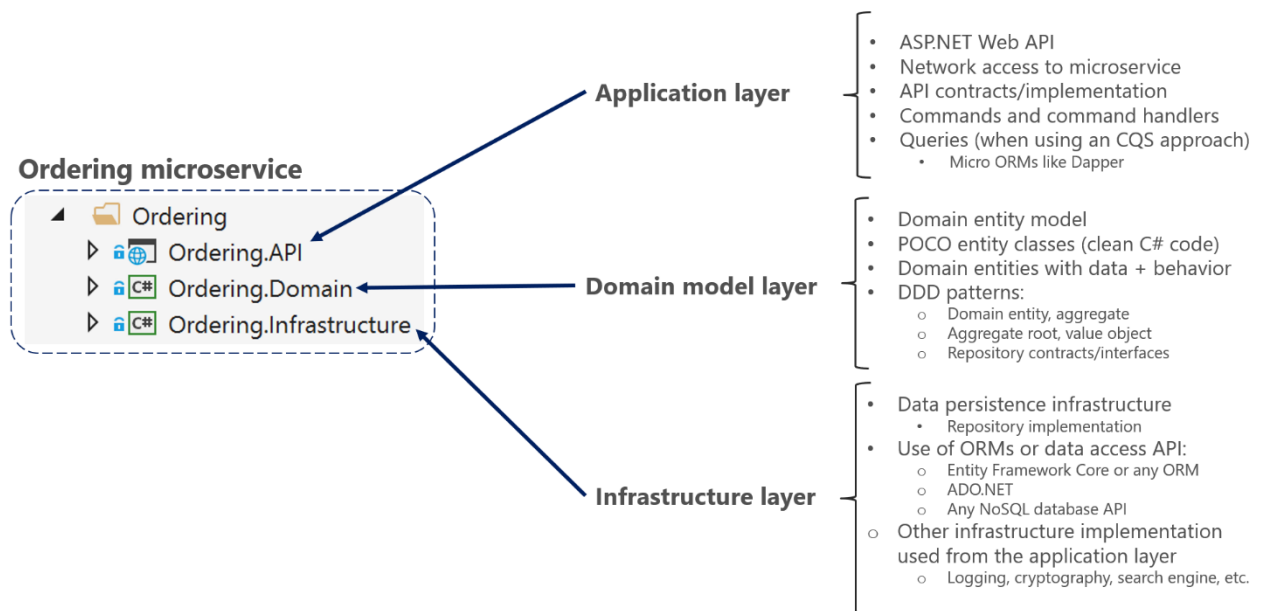


图 9-5. eShopOnContainers 里订单微服务中的 DDD 分层

您想设计系统以便每一层只跟特定的其他层进行通信，强制将这些层作为不同的类库来实现会更容易实现这个目标，因为能够很清晰的识别类库之间有哪些依赖。例如，领域模型层不应该依赖于其他任何层（领域模型类应该是普通的 CLR 对象，或称为 [POCO](#) 的类）。如图 9-6 所示，Ordering.Domain 类库仅仅依赖 .NET Core 的库或 NuGet 包，而没有依赖任何其他自定义库（数据层类库，持久化库等等）。

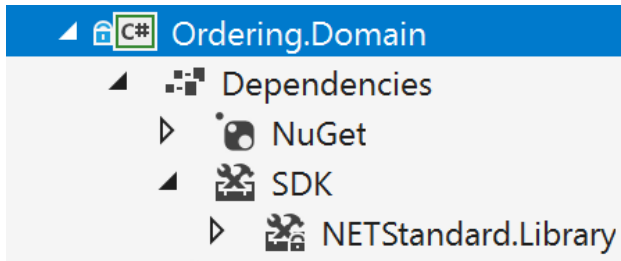


图9-6. 作为类库实现的层可以更好地控制层与层之间的依赖

领域模型层

Eric Evans 的名著《[领域驱动设计](#)》的下列段落描述了领域模型层和应用层。

领域模型层：负责表达业务概念，业务状态信息以及业务规则。尽管保存业务状态的技术细节是由基础设施层实现的，但是反映业务情况的状态是由本层控制并且使用的，领域模型层是业务软件的核心。

领域模型层是表达业务的地方。当您在 .NET 中使用微服务来实现领域模型层时，该层使用包含获取数据和行为（具有业务逻辑的方法）的领域实体的类库来实现。

根据[持久化透明](#)和[基础架构透明](#)的原则，该层必须完全忽视数据持久化的细节，这些持久化任务应该在基础架构层完成。因此，该层不应该直接依赖于基础架构层，这意味着一个重要规则即您的领域模型实体类应该是 [POCO](#)。

领域实体不应该对任何数据访问基础框架比如 Entity Framework 或 NHibernate 有任何直接的依赖。理想情况下，领域实体不应该继承自或实现任何基础框架里的任何类型。

大多数现代的 ORM 框架比如 Entity Framework Core 允许使用这种方式，因此您的领域模型类不会跟基础架构耦合。然而，在使用某些 NoSQL 数据库和框架时，使用 POCO 实体并不都是可能的，比如 Azure Service Fabric 里的 Actors 和 Reliable 集合。

即使为领域模型遵守持久化透明原则很重要，也不应该忽略持久化的问题。理解数据的物理模型和如何映射到实体对象模型依然很重要，否则您的设计就无法实现。

另外，这也不是意味着您可以将一个为关系型数据库设计的模型直接移动到 NoSQL 或面向文档的数据库中。对于某些实体模型可能可以这么做，但通常是不匹配的。也存在一些实体模型必须遵守同时来自于存储技术和 ORM 技术的约束条件。

应用层

转到应用层，我们可以再次引用 Eric Evans 的《[领域驱动设计](#)》：

应用层：定义软件要完成的任务，并且指挥表达领域概念的对象来解决问题。这一层所负责的工作对业务来说意义重大，也是与其他系统的应用层进行交互的必要渠道。应用层要尽量简单，不包含业务规则或知识，而只为下一层中的领域对象协调任务，分配工作，使它们互相协作。它没有反映业务情况的状态，但是却可以具有另外一种状态，即为用户或程序显示某个任务的进度。

.NET 中的微服务应用层通常使用 ASP.NET Core Web API 项目来实现，包括微服务的交互、远程网络访问和从界面或客户端应用发起的外部 Web API 调用。它还包含使用 CQRS 方法时的查询、微服务接受的命令以及微服务之间的事件驱动通信（集成事件）。表示应用层的 ASP.NET Core Web API 必须不包含业务规则或领域知识（尤其是事务或更新的领域规则），它们应该由领域模型类库所拥有。应用层必须仅仅协调任务而不能保留或定义任何领域状态（领域模型）。应用层将业务规则的执行委托给领域模型类本身（聚合根或领域实体），最终在那些领域实体中更新数据。

基本上，应用逻辑是实现基于给定前端的所有用例的地方，比如跟一个 Web API 服务相关的实现。

其目的是在领域模型层中的领域逻辑，包括不变量、数据模型和相关的业务规则必须完全独立于表现层和应用层。最重要的是领域模型层必须不能直接依赖任何基础架构框架。

基础架构层

基础架构层是实现如何将最初保留在（内存中）领域实体中的数据持久化到数据库或其他持久存储中。一个例子是使用 Entity Framework Core 实现的使用 DbContext 的仓储模式类将数据持久化到关系型数据库中。

根据前文提到的[持久化透明](#)和[基础架构透明](#)原则，基础架构层必须不能“污染”领域模型层。您必须通过不发生框架的强依赖来保持领域模型实体类和用来持久化数据（EF 或任何其他框架）的基础架构无关。领域模型层类库应该只有领域代码，即仅有实现软件核心的 [POCO](#) 实体类，并且跟基础架构技术完全解耦。

因此，您的层或类库以及项目应该最终依赖于领域模型层（库），反过来则不行。如图 9-7 所示。

领域驱动设计服务里各层之间的依赖关系

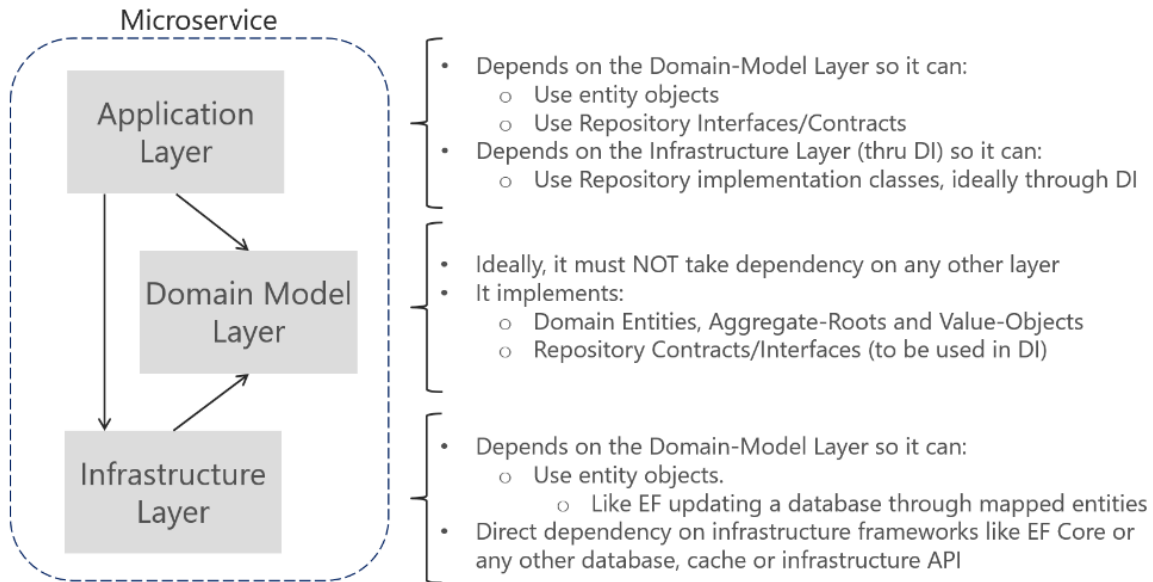


图9-7. DDD 中各层之间的依赖关系

每个微服务的基础架构层设计应该是独立的。如前所述，您可以按照 DDD 模式实现最复杂的微服务，而用更简单的方法实现更简单的数据驱动微服务（单层中的简单 CRUD）。

其他资源

- **DevIQ. 持久化透明原则**
<http://deviq.com/persistence-ignorance/>
- **Oren Eini. 基础架构透明原则**
<https://ayende.com/blog/3137/infrastructure-ignorance>
- **Angel Lopez. 领域驱动开发的分层架构**
<https://ajlopez.wordpress.com/2008/09/12/layered-architecture-in-domain-driven-design/>

设计微服务领域模型

为每个业务微服务或限界上下文定义一个富领域模型

您的目标是为每个业务微服务或限界上下文（BC）创建一个单一的内聚的领域模型。但是请记住，BC 或业务微服务有时候可能由多个共享一个单一的领域模型的物理服务组成。领域模型必须捕获它所表达的单一限界上下文或业务微服务的规则、行为、业务语言和约束条件。

领域实体模式

实体表示领域对象，主要由其标识、连续性和随时间变化的持久性来定义，而不仅仅是构成它们的那些属性。正如 Eric Evans 所说“一个主要由其标识来定义的对象被称为实体”。由于实体是模型的基础，因此它们在领域模型中非常重要。因此，您应该仔细识别和设计它们。

一个实体的标识可以跨多个微服务或限界上下文。

相同的标识（尽管不是相同的实体）可以跨多个限界上下文或微服务进行建模。然而并不意味着具有相同属性和逻辑的相同实体应该在多个限界上下文中实现。相反，每个限界上下文中的实体将它们的属性和行为限定在领域必需的范围內。

例如买家实体可能拥有大部分个人属性，这些属性定义在个人资料或标识微服务中的用户实体里，包括它的标识。但是订单微服务中的买家实体具有更少的属性，因为只有部分的买家数据跟订单流程相关。每个微服务或限界上下文的环境影响它们的领域模型。

领域实体除了实现数据属性之外，还必须实现行为。

DDD 中的领域实体必须实现跟实体数据（内存中访问的对象）相关的领域逻辑或行为。比如，作为订单实体类的一部分，必须将业务逻辑和操作以任务的方法来实现，例如添加一个订单项、数据验证和计算总和。实体的方法处理实体的不变量和规则，而不是将它们扩散到应用层。

图 9-8 显示的领域实体不仅实现了数据属性，而且实现了相关领域逻辑的操作或方法。

领域实体模式

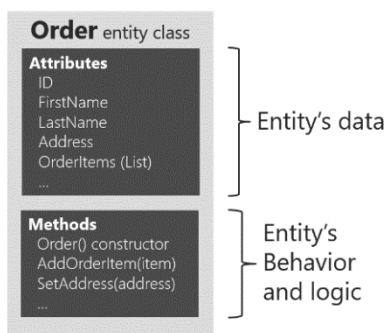


图9-8. 实现数据和行为的领域实体设计示例

当然，有时候可以拥有不实现任何逻辑的实体类，这会发生在一个聚合的子实体中，如果子实体没有任何特殊的逻辑，因为大部分逻辑已经在聚合根里定义了。如果您有一个复杂的微服务，在服务类中实现了大量的逻辑，而不是在领域实体中实现它们的话，您可能陷入了贫血领域模型中，在下一节将会解释。

富领域模型和贫血领域模型

Martin Fowler 在他的帖子[贫血领域模型](#)中以这种方式进行了描述：

贫血领域模型一个明显的特征是它仅仅是看上去和领域模型一样，都拥有对象、属性、对象间通过关系关联。但是当你观察模型所持有的业务逻辑时，你会发现，贫血模型中除了一些getter 和 setter 方法，几乎没有其他业务逻辑。

当然，在您使用一个贫血领域模型时，那些数据模型会从一系列服务对象（传统上称为业务层）中使用，这些服务对象捕获所有领域或业务逻辑。业务层位于数据模型层之上并仅仅将数据模型作为数据使用。

贫血领域模型只是一种过程化的风格设计。因为缺乏行为（方法），贫血实体对象不是真正的对象。它们只有数据属性因此也不是面向对象的设计。通过将所有的行为放到服务对象（业务层）中，最终得到的是[意大利面代码](#)或[事务性脚本](#)，因此失去了领域模型提供的优势。

无论如何，如果您的微服务或限界上下文非常简单（如 CRUD 服务），使用表现为只有数据属性的实体对象的贫血领域模型或许足够了，可能不值得实现更复杂的 DDD 模式。这种情况下，它将是一个简单的持久化模型，因为已经有意地为 CRUD 这个目标创建的只有数据的实体。

这就是为什么微服务架构完美适用于依赖每个限界上下文的多重架构方法。例如在 eShopOonContainers 中，订单微服务实现了 DDD 模式，但只是简单的 CRUD 服务的目录微服务就没有用到。

有人说贫血领域模型是一个反模式。这其实取决于您在实现什么。如果您正在创建的微服务足够简单（比如一个 CRUD 服务），遵循贫血领域模型就不是一个反模式。但是，如果您需要应对微服务的领域有很多不断变化的业务规则的复杂性时，贫血领域模型对于这样的微服务或限界上下文可能就是一个反模式了。在这种情况下，将其设计为包含数据和行为以及实现额外的 DDD 模式（聚合，值对象等等）的富模型或许对于这样的微服务的长期的成功有很大好处。

其他资源

- **DevIQ. 领域实体**
<http://deviq.com/entity/>
- **Martin Fowler. 领域模型**
<https://martinfowler.com/eaaCatalog/domainModel.html>
- **Martin Fowler. 贫血领域模型**
<https://martinfowler.com/bliki/AnemicDomainModel.html>

值对象模式

正如 Eric Evans 所说“许多对象没有概念上的标识。这些对象描述了一件事情的某些特征。”

实体需要标识，但系统中很多对象没有标识，比如值对象模式。一个值对象是描述一个领域方面的没有概念上的标识的对象。这些对象是实例化用来临时表示设计元素的。您只关心它们是什么，而不关心它们是谁。例子包括数字和字符串，但也有高级别的概念比如一组属性。

一个微服务中的实体可能不是另一个微服务的实体，因为在后面一种情况下，限界上下文可能有不同的含义。例如，一个电子商务应用中的一个地址可能根本没有标识，因为它可能只是表示一个人或公司的客户资料的一组属性。这种情况下的地址应该归类为值对象。然而在一个电力实业公司的应用中，客户地址应该对于业务领域很重要。因此，地址必须有标识，以便计费系统能够直接关联到地址上。这种情况下的地址应该归类为领域实体。

带有名和姓的人通常是一个实体，因为人有标识，即使名和姓跟另一组值相同，例如另外的名指向一个不同的人。

值对象在关系型数据库和类似 EF 的 ORM 中很难管理，而在文档型数据库里更容易实现和使用。

其他资源

- **Martin Fowler. 值对象模式**
<https://martinfowler.com/bliki/ValueObject.html>
- **值对象**
<http://deviq.com/value-object/>
- **测试驱动开发中的值对象**
<https://leanpub.com/tdd-ebook/read#leanpub-auto-value-objects>
- **Eric Evans. 领域驱动设计，软件核心复杂性应对之道（图书，包括关于值对象的讨论）**
<https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/>

聚合模式

领域模型包含不同数据实体的聚集和能控制特定功能区域的流程，比如订单履行或库存。一种更细粒度的 DDD 单元就是聚合，描述了可以被视为一个内聚体的一组实体和行为。

通常基于需要的事务来定义聚合。一个典型的例子是订单也包含了一组订单项。一个订单项通常是一个实体，但它在订单聚合里会是一个子实体，它也会包含作为它的根实体的订单实体，通常被称作聚合根。

识别聚合会很困难。一个聚合是一组必须一致的对象，但不能仅仅选择一组对象并将它们标记为聚合。您必须从领域概念开始并考虑与概念相关的用在最通用的事务中的实体。那些需要事务性一致的实体就构成一个聚合。考虑事务操作很可能是识别聚合的最佳方法。

聚合根或根实体模式

一个聚合至少包含一个实体：聚合根。也叫做根实体或者主标识。此外，它可以有多个子实体和值对象。所有实体对象共同协作去实现所需的行为和事务。

聚合根的用途是保证聚合的一致性。它应该是聚合根类中通过方法或者操作更新聚合的唯一入口点。您应该只通过聚合根去修改聚合里的实体。因此它是聚合一致性的保护者，负责聚合中所有的所需要遵守的不变性和一致性规则。当您独立更改一个子实体或者值对象，聚合根无法保证聚合处于有效的状态，就像腿散开一地的桌子。维护一致性是聚合根的主要用途。

在图 9-9 中，您可以看到简单的买家聚合，它包含一个单一的实体（聚合根 Buyer），订单聚合包含多个实体和一个值对象。

聚合模式

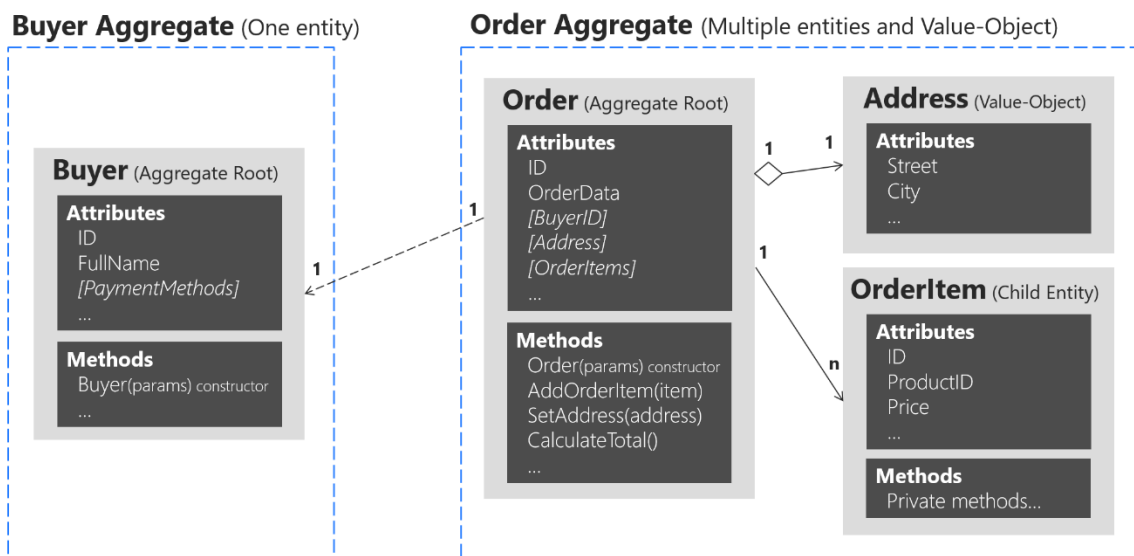


图9-9. 跟单一或多个实体聚合的例子

注意，买家聚合可以包含额外的实体，这取决于您的领域，就像在 eShopOnContainers 示例应用程序的订单微服务中的那样。图 9-9 只是用买家演示作为一个聚合只包含一个聚合根对象，所以才只包含一个实体。

为了维护聚合的独立以及保持它们之间清晰的边界。作为一种好的实践方式，在 DDD 领域模型中禁止直接在聚合间导航并且只包含外键（FK）字段。如同 eShopOnContainers 在[订单微服务的领域模型](#)中的实现。Order 实体只包含买家的 FK 字段，而不是 EF Core 导航属性。如下所示的代码：

```
public class Order : Entity, IAggregateRoot
{
    private DateTime _orderDate;
    public Address Address { get; private set; }
    private int? _buyerId; //FK pointing to a different aggregate root
    public OrderStatus OrderStatus { get; private set; }
    private readonly List<OrderItem> _orderItems;
    public IReadOnlyCollection<OrderItem> OrderItems => _orderItems;

    //... Additional code
}
```

标识和使用聚合需要研究和经验，更多的信息参考下面的资源列表：

其他资源

- **Vaughn Vernon. 高效聚合设计 - 卷 1: 单个聚合建模**
https://vaughnvernon.co/wordpress/wp-content/uploads/2014/10/DDD_COMMUNITY_ESSAY_AGGREGATES_PART_1.pdf
- **Vaughn Vernon. 高效聚合设计 - 卷 2: 组合聚合**
https://vaughnvernon.co/wordpress/wp-content/uploads/2014/10/DDD_COMMUNITY_ESSAY_AGGREGATES_PART_2.pdf
- **Vaughn Vernon. 高效聚合设计 - 卷 3: 通过发现获得洞察力**
https://vaughnvernon.co/wordpress/wp-content/uploads/2014/10/DDD_COMMUNITY_ESSAY_AGGREGATES_PART_3.pdf
- **Sergey Grybniak. DDD 战术设计模式**
<https://www.codeproject.com/Articles/1164363/Domain-Driven-Design-Tactical-Design-Patterns-Part>
- **Chris Richardson. 使用聚合开发事务性微服务**
<https://www.infoq.com/articles/microservices-aggregates-events-cqrs-part-1-richardson>
- **DevIQ. 聚合模式**
<http://deviq.com/aggregate-pattern/>

使用.NET Core 实现微服务领域模型

在前面一章中已经阐述了设计领域模型的基本的设计原则和模式。现在是探索使用.NET Core(纯 C#代码)和 EF Core 实现领域模型的各种可能的途径的时候了。注意，您的领域模型将只包含您编写的简单的 C#代码。它将只需要 EF Core 模型类，但是不真的需要依赖 EF，您不需要在您的领域模型中硬性引用或者依赖于 EF Core 或者其它 ORM。

自定义.NET 标准库中的领域模型结构

eShopOnContainers 示例应用程序中的文件目录组织演示了应用程序的 DDD 模型。您可以发现用不同的文件夹组织会使得表达您的应用程序的设计方式更清晰。如同您在图 9-10 所见到的，订单微服务模型有两个聚合，分别是 Order 聚合和 Buyer 聚合，每个聚合是一组领域实体和值对象，虽然您也可以使用单个领域实体（聚合根或者根实体）来构成一个聚合。

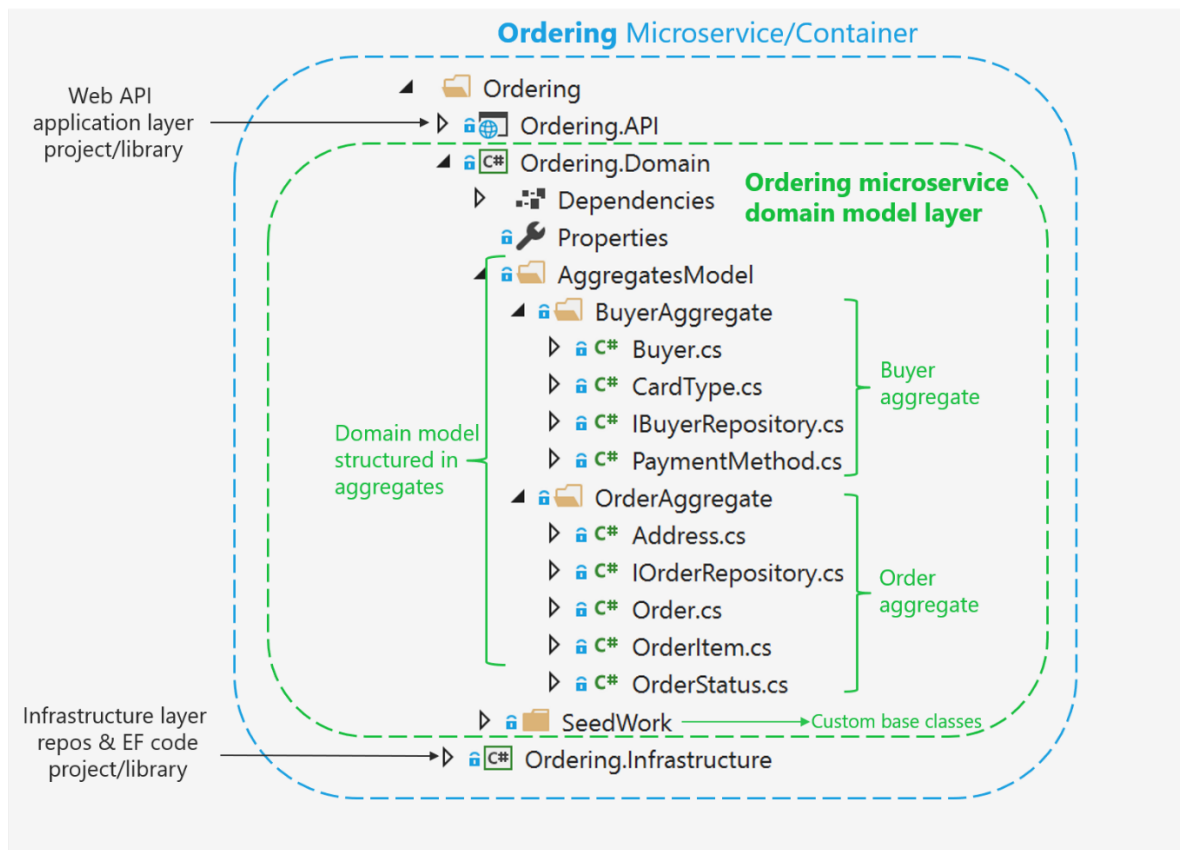


图 9-10. eShopOnContainers 中订单微服务的领域模型结构

另外，领域模型层包括了领域模型的基础设施的需求的仓储契约（接口）。换句话说，这些接口表达了基础设施层需要拥有什么仓储以及如何实现。这对于实现放在领域模型层之外的仓储非常重要。在基础

设施层库中，领域模型层不是被 API 或者基础设施技术所“污染”（也就是领域模型和所用的基础设施技术强绑定在一起，译者注），比如 Entity Framework。

您还可以看到一个叫做 [SeedWork](#) 的目录，包含自定义的基类，您可以用以作为领域实体和值对象的基础。这样，就不会在每个领域对象中出现冗余的代码。

在自定义.NET 标准库中组织聚合

一个聚合指的是按照匹配事务性一致性分组的一批领域对象。这些对象可以是实体（根聚合或者根实体之一）的实例。

事务性一致性意味着一个聚合被保证是一致的，并且在业务活动结束后保持最新。比如，eShopOnContainers 的订单微服务领域模型的订单聚合由图 9-11 所示构成。

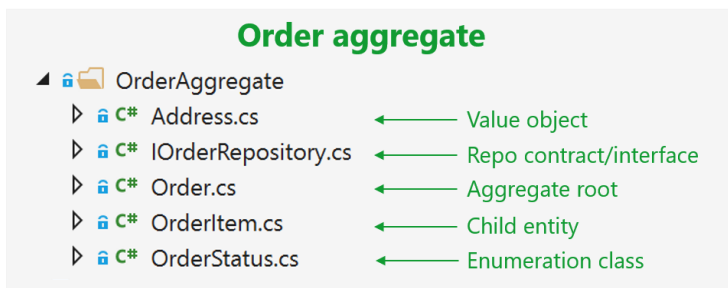


图9-11. Visual Studio 解决方案中的订单聚合

如果您打开任意一个聚合的文件夹，您可以看到它是如何被标记为自定义基类，或者接口的，比如 [SeedWork](#) 文件夹中实现的实体或值对象。

以 POCO 类实现领域实体

您通过 POCO 类作为您的领域实体在.NET 里实现一个领域模型。在下面的例子里，Order 类被定义为一个实体同时也是一个聚合根。因为 Order 从 Entity 基类继承，它可以重用实体相关的公共代码。请记住，基类和接口是您在领域模型项目中定义的，所以这是您的代码，而不是像 EF 那样的 ORM 的基础设施代码。

```
// COMPATIBLE WITH ENTITY FRAMEWORK CORE 2.0
// Entity is a custom base class with the ID
public class Order : Entity, IAggregateRoot
{
    private DateTime _orderDate;
    public Address Address { get; private set; }
    private int? _buyerId;

    public OrderStatus OrderStatus { get; private set; }
    private int _orderStatusId;
}
```

```

private string _description;
private int? _paymentMethodId;

private readonly List<OrderItem> _orderItems;
public IReadOnlyCollection<OrderItem> OrderItems => _orderItems;

public Order(string userId, Address address, int cardTypeId,
             string cardNumber, string cardSecurityNumber,
             string cardHolderName, DateTime cardExpiration,
             int? buyerId = null, int? paymentMethodId = null)
{
    _orderItems = new List<OrderItem>();
    _buyerId = buyerId;
    _paymentMethodId = paymentMethodId;
    _orderStatusId = OrderStatus.Submitted.Id;
    _orderDate = DateTime.UtcNow;
    Address = address;

    // ...Additional code ...
}

public void AddOrderItem(int productId, string productName,
                        decimal unitPrice, decimal discount,
                        string pictureUrl, int units = 1)
{
    //...
    // Domain rules/logic for adding the OrderItem to the order
    // ...

    var orderItem = new OrderItem(productId, productName, unitPrice,
discount, pictureUrl, units);

    _orderItems.Add(orderItem);

}
// ...
// Additional methods with domain rules/logic related to the Order
aggregate
// ...
}

```

注意这是一个 POCO 类作为领域实体，这很重要。它不包含任何直接的对于 Entity Framework Core 或者其它基础设施框架的依赖。这个实现像在 DDD 里的那样，只是用 C#代码实现的领域模型。

还有，这个类被一个叫做 IAggregateRoot 的接口所修饰，这个接口是一个空接口，有时被称为标记接口。它用于指示这个实体类同时也是一个聚合根。

标记接口有时候会被人认为是反模式，然而这同时是一种标记一个类的清楚的方式。特别是当这个接口可能会不断演化。特性 (Attribute) 是作为标记的另一个选择。但是相比在类前面加上 [Aggregate] 特性，在基类 (entity) 后面跟着 IAggregate 接口更快更容易被看到。无论如何，这两个办法只是个人喜好问题。

拥有聚合根意味着和一致性、业务规则有关的聚合实体的大多数代码应该通过 Order 聚合根类的方法实现 (比如, AddOrderItem, 用于添加 OrderItem 到聚合)。您不应该独立或者直接地创建和更新 OrderItem 对象。AggregateRoot 类必须保持对子实体的任何更新操作的控制和保持一致性。

在领域实体中封装数据

在实体模型中的一个常见问题是，它们以公有可访问的列表类型暴露了集合导航属性。这使得任何别的协作的程序员可以操纵这些集合类型的内容。这将绕过相关的集合的重要业务规则。可能导致对象拥有无效的状态。解决方案是以只读访问的方式暴露这些相关的集合，同时显式提供让调用者操纵它们的方法。

在之前的代码中，请注意很多特性是只读的或者私有并且只能由类的方法更新的。因此，任何更新都无例外地由领域模型和类的方法所指定的逻辑负责。

比如，下面的 DDD，您不应该在任何命令处理程序方法或者应用层的类中做下面的事情：

```
// WRONG ACCORDING TO DDD PATTERNS - CODE AT THE APPLICATION LAYER OR
// COMMAND HANDLERS

// Code in command handler methods or Web API controllers

//... (WRONG) Some code with business logic out of the domain classes ...
OrderItem myNewOrderItem = new OrderItem(orderId, productId, productName,
                                           pictureUrl, unitPrice, discount, units);

//... (WRONG) Accessing the OrderItems collection directly from the application layer
// or command handlers

myOrder.OrderItems.Add(myNewOrderItem);

//...
```

在这个例子里，Add 方法是一个纯粹添加数据的方法，它直接访问了 OrderItems 集合，因此，和子实体相关的大部分领域逻辑、规则或者验证被扩散到应用层 (命令处理程序，或者 web api 控制器)。

如果您遍历聚合根，聚合根不能保证它的不变性、有效性以及一致性。最终将会产生形如意大利面条或者事务性脚本代码 (指的是一次性编写的不良代码，译者注)。

要遵守 DDD 模式，实体的任何实体属性都必须不含有任何公有的 set 属性访问器。在实体中的更改应该以如何在实体中执行更改的通用语言明显说明的方法显式驱动。

进一步地，带有实体的集合（比如 orderitems）应该是只读属性（AsReadOnly 方法，之后会提到），您应该只能从聚合根类方法或者子实体方法中更新。

如您所见的 Order 聚合根的代码，所有 set 属性访问器应该是私有的或者起码是对外部只读的。因此任何对实体的数据或者它的子实体的操作都通过实体类的方法进行。这将以受控和面向对象的方式维护一致性，以取代用事务性脚本的实现。

下面的代码片段展示了添加 OrderItem 对象到 Order 聚合的任务的正确编码方式。

```
// RIGHT ACCORDING TO DDD--CODE AT THE APPLICATION LAYER OR COMMAND HANDLERS

// The code in command handlers or WebAPI controllers, related only to application stuff
// There is NO code here related to OrderItem object's business logic

myOrder.AddOrderItem(productId, productName, pictureUrl, unitPrice, discount, units);

// The code related to OrderItem params validations or domain rules should
// be WITHIN the AddOrderItem method.
//...
```

在本代码片段中，大部分的验证和创建 OrderItem 对象的逻辑将受到 Order 聚合根的控制——在 AddOrder 方法，特别是验证以及和其他元素相关的逻辑中。比如您多次调用 AddOrderItem，结果是得到相同的产品项。在此方法中，您可以验证产品项以及相同的产品归并为一个 OrderItem，并且拥有多个单位数量。额外地，如果对于相同产品 ID，有不同的折扣比例，您希望应用折扣比例高的，这个原则将运用到 OrderItem 对象的其它领域逻辑中。

另外，在 Order 聚合根里的 new OrderItem(params) 操作也将被 AddOrderItem 方法控制和执行。因此，多数的逻辑或者相关操作（特别是影响到子实体一致性的）的验证将放在聚合根这唯一的位置里。这就是聚合根模式的最终目的。

当您使用的是 Entity Framework Core 1.1, 2.0 或者更新版本，DDD 实体可以被更好地用 EF Core 1.1 或 2.0 的一个新特性描述。这个特性允许在额外的属性中[映射到字段](#)。这对于您保护子实体和值对象的集合很有用。使用这一增强功能，您可以使用私有字段代替属性，您还可以在公有方法中实现对字段集合的任何更新以及通过 AsReadOnly 方法提供只读访问。

在 DDD 中，您希望更新只通过实体中的方法（或者构造方法）更新实体，以便控制数据的所有不变性和一致性。所以只定义 get 属性访问器。属性拥有私有的后备字段私有成员只能通过类内部访问，然而，有一个例外，EF Core 也需要设置这些字段。

映射只有 get 访问器的字段到数据库表

映射属性到数据库表的列不是领域而是基础设施和持久层的职责。我们在这里提出这一点，只是让您意识到 EF Core 1.1, 2.0 或者更新版的新特性关系到您如何对实体建模。更多关于这个细节的内容在基础设施和持久化的章节中有解释。

当您使用带有 DbContext 类的 EF Core 1.0，您需要将只定义了 get 访问器的属性映射到实际数据库表的字段。这通过 PropertyBuilder 的 HasField 方法实现。

映射没有属性的字段

使用 EF Core 1.1 和 2.0 的将列映射到字段的特性。还可以不使用属性，而是只将字段映射到表的列。对此一个常见的使用场景是，内部状态的私有字段不需要被实体外部访问。

比如，在前面的 OrderAggregate 代码例子中，有若干私有字段，比如 _paymentMethodId，没有与之相关的 get 或 set 属性访问器。那个字段可以在 order 的业务逻辑中被计算，并被 order 类的方法使用。但是它也需要被放入数据库持久化。所以，在 EF Core(从 1.1 开始)，提供了一个不需要相关属性就能映射到数据库表的列的方式。这一点，也在本书的[基础架构层](#)小节中解释了。

其他资源

- **Vaughn Vernon. 使用 DDD 和 EF 建模聚合。注意，不是 EF Core。**
<https://vaughnvernon.co/?p=879>
- **Julie Lerman. 领域驱动设计的编码：关注数据的开发技巧**
<https://msdn.microsoft.com/magazine/dn342868.aspx>
- **Udi Dahan. 如何创建完全封装的领域模型**
<http://udidahan.com/2008/02/29/how-to-create-fully-encapsulated-domain-models/>

Seedwork (可重复用于领域模型的基类和接口)

如之前提到，在解决方案文件夹，您还可以看到 SeedWork 文件夹，这个文件夹包含了可以用于领域实体和值对象的自定义基类，因而避免在每个领域对象中包含冗余的代码。包含这种类型的类的文件夹叫做 SeekWork，而不是 Framework。这是因为这个文件夹只包含了很少可重用类的集合，因此还不能称得上是真正的框架库。Seedwork 这个术语是 Michael Feathers [提出](#)的，并且被 [Martin Fowler](#) 推广，但是您也可以将这个目录命名为 Common, ShareKernel, 或者类似的名字。

图 9-12 显示了订单微服务中的领域模型的 seedwork 的类，它拥有几个自定义基类，比如 Entity, ValueObject 以及 Enumeration，加上一些接口。这些接口 (IRepository 和 IUnitOfWork) 告诉基础设施层需要实现什么，这些接口还被应用层的依赖注入使用。

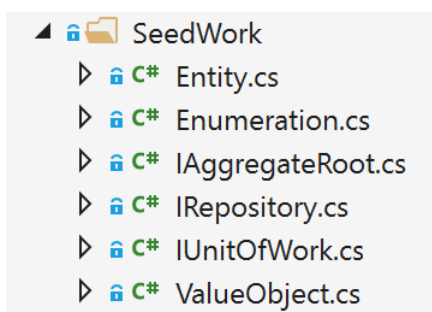


图 9-12. 领域模型中的“seedwork”基类和接口的一些例子

这是一种复制粘贴式的重用，很多开发人员用它来在不同项目中共享代码。它不是一个正式的框架。seekwork 可以放在任何层或者库中，然而如果这个类和接口的集合变得足够大，您会考虑创建一个单独类库。

自定义 Entity 基类

下面是 Entity 基类的一个例子，别的领域实体可以使用相同的方式放入代码。比如实体 ID, [相等判断](#)操作，每个实体的领域的事件列表，等等。

```
// COMPATIBLE WITH ENTITY FRAMEWORK CORE (1.1, 2.0 or later)
public abstract class Entity
{
    int? _requestedHashCode;
    int _Id;
    private List<INotification> _domainEvents;
    public virtual int Id
    {
        get
        {
            return _Id;
        }
        protected set
    }
}
```

```

    {
        _Id = value;
    }
}

public List<INotification> DomainEvents => _domainEvents;
public void AddDomainEvent(INotification eventItem)
{
    _domainEvents = _domainEvents ?? new List<INotification>();
    _domainEvents.Add(eventItem);
}
public void RemoveDomainEvent(INotification eventItem)
{
    if (_domainEvents is null) return;
    _domainEvents.Remove(eventItem);
}

public bool IsTransient()
{
    return this.Id == default(Int32);
}

public override bool Equals(object obj)
{
    if (obj == null || !(obj is Entity))
        return false;
    if (Object.ReferenceEquals(this, obj))
        return true;
    if (this.GetType() != obj.GetType())
        return false;
    Entity item = (Entity)obj;
    if (item.IsTransient() || this.IsTransient())
        return false;
    else
        return item.Id == this.Id;
}

public override int GetHashCode()
{
    if (!IsTransient())
    {
        if (!_requestedHashCode.HasValue)
            _requestedHashCode = this.Id.GetHashCode() ^ 31;
        // XOR for random distribution. See:
        //
    }
}

```

```
// http://blogs.msdn.com/b/ericlippert/archive/2011/02/28/guidelines-and-rules-for-gethashcode.aspx
```

```
        return _requestedHashCode.Value;
    }
    else
        return base.GetHashCode();
}
public static bool operator ==(Entity left, Entity right)
{
    if (Object.Equals(left, null))
        return (Object.Equals(right, null)) ? true : false;
    else
        return left.Equals(right);
}
public static bool operator !=(Entity left, Entity right)
{
    return !(left == right);
}
}
```

上述代码使用了每个实体的领域事件列表，将在下面关注领域事件的小节中介绍。

领域模型层的仓储契约（接口）

仓储契约是简单的.NET 接口，表达了每个聚合所需要的仓储的契约需求。

仓储本身以及所带的 EF Core 代码或者其它基础设施和代码依赖（Linq, SQL 等等）不能在领域模型中被实现。仓储应该只实现您定义的接口。

和此相关的一个模式（在领域模型层放置仓储接口）是分离接口模式，Martin Fowler [解释道](#) “使用分离接口模式在一个包中定义一个接口，但是在另一个包中实现它，使用这个方式，调用者所需要的接口依赖可以对它的实现完全透明”。

遵循分离接口模式使得应用程序层（本案例中的微服务的 Web API 项目）拥有对定义在领域模型的需求的依赖关系，但是不直接依赖基础设置/持久化层。另外，您可以使用依赖注入来隔离在基础设施/持久化层中使用仓储的代码实现。

比如，下面的例子，使用 IOrderRepository 接口定义在基础设施层里 OrderRepository 类需要实现什么操作，在应用程序的当前实现中，代码仅仅需要添加和更新订单到数据库，而查询按照 CQRS 模式被切分。

```
// Defined at IOrderRepository.cs
public interface IOrderRepository : IRepository<Order>
{
    Order Add(Order order);

    void Update(Order order);

    Task<Order> GetAsync(int orderId);
}

// Defined at IRepository.cs (Part of the Domain Seedwork)
public interface IRepository<T> where T : IAggregateRoot
{
    IUnitOfWork UnitOfWork { get; }
}
```

其他资源

- **Martin Fowler. 分离的接口**
<http://www.martinfowler.com/eaCatalog/separatedInterface.html>

实现值对象

如之前章节对实体和聚合的讨论，标识是实体的基础。然而，系统中的很多对象和数据项不需要标识和标识跟踪，比如值对象。

值对象可以引用别的实体，例如，应用程序生成了一个描述一个点到另一个点的路径，那么这个路径对象就是一个值对象，它是对于一个特定路径上的点的快照，但是路径被认为没有标识，尽管对外部它参照了城市、街道等实体。

图 9-13 显示了 Order 聚合里的 Address 值对象。

聚合中的值对象

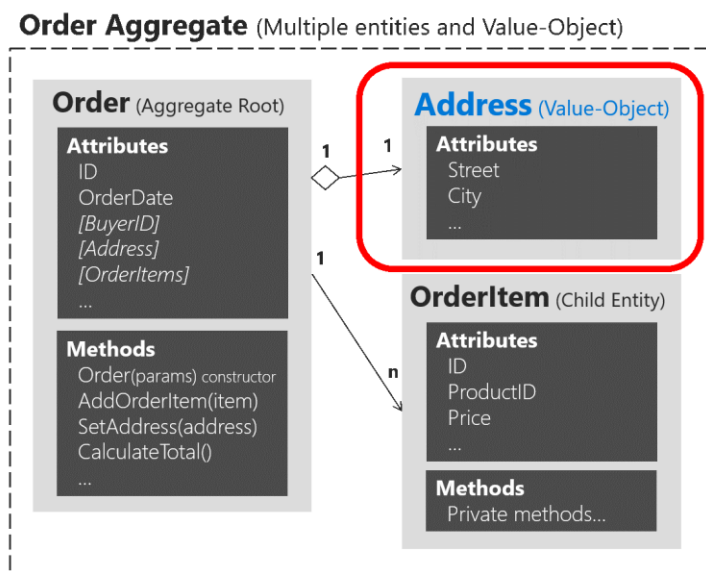


图9-13. Order 聚合中的 Address 值对象

如图 9-13 所示，一个实体往往由多个属性组成。比如，Order 实体可以被建模成一个带有 ID 以及带有 OrderId, OrderDate, OrderItems 等内在属性的实体。而地址，是由国家、城市、街道等组成的复合值，在这个域中，没有 ID，必须被建模和当作值对象来处理。

值对象的重要特征

值对象有两个重要特征：

- 他们没有 ID。
- 它们是不可变的。

第一个特征已经讨论过了，不变性是非常重要的需求。当对象创建后，对象的值必须不再可更改。因而在对象构造的时候，必须提供所需的值，而在对象的生命周期里不允许再改变它们。

值对象得益于不变性的特征，它允许您使用特定的技巧来优化性能。当系统中可能存在数以千计的值对象的实例，而它们中的很多拥有相同的值的时候，这一点尤为突出。因为它们是不变的，所以可以重用。它们是可以互相替换的对象，因为它们没有 ID，而且值相同。这种优化有时可以让运行缓慢的系统拥有不同的性能表现。当然，所有这些案例取决于应用程序的环境和部署上下文。

值对象在 C# 中的实现

关于实现，您可以先创建一个值对象的基类，包含基本的辅助方法，比如比较全部属性判断是否相等（因此值对象不能基于 ID），以及其它的基础的特性。下面的示例是来自 eShopOnContainers 的订单微服务的值对象的基类。

```
public abstract class ValueObject
{
    protected static bool EqualOperator(ValueObject left, ValueObject right)
    {
        if (ReferenceEquals(left, null) ^ ReferenceEquals(right, null))
        {
            return false;
        }
        return ReferenceEquals(left, null) || left.Equals(right);
    }

    protected static bool NotEqualOperator(ValueObject left, ValueObject right)
    {
        return !(EqualOperator(left, right));
    }

    protected abstract IEnumerable<object> GetAtomicValues();

    public override bool Equals(object obj)
    {
        if (obj == null || obj.GetType() != GetType())
        {
            return false;
        }
        ValueObject other = (ValueObject)obj;
        IEnumerator<object> thisValues = GetAtomicValues().GetEnumerator();
        IEnumerator<object> otherValues = other.GetAtomicValues().GetEnumerator();
        while (thisValues.MoveNext() && otherValues.MoveNext())
        {
            if (ReferenceEquals(thisValues.Current, null) ^
                ReferenceEquals(otherValues.Current, null))
            {
                return false;
            }
            if (thisValues.Current != null &&
                !thisValues.Current.Equals(otherValues.Current))
            {
                return false;
            }
        }
    }
}
```

```

    }
}
return !thisValues.MoveNext() && !otherValues.MoveNext();
}
// Other utility methods
}

```

当您实现您实际的值对象的时候，可以使用这一代码，以及下面所示的 Address 值对象。

```

public class Address : ValueObject
{
    public String Street { get; private set; }
    public String City { get; private set; }
    public String State { get; private set; }
    public String Country { get; private set; }
    public String ZipCode { get; private set; }
    public Address(string street, string city, string state,
                  string country, string zipcode)
    {
        Street = street;
        City = city;
        State = state;
        Country = country;
        ZipCode = zipcode;
    }

    protected override IEnumerable<object> GetAtomicValues()
    {
        // Using a yield return statement to return each element one at a time
        yield return Street;
        yield return City;
        yield return State;
        yield return Country;
        yield return ZipCode;
    }
}

```

您可以看到这个 Address 类是如何实现的，它没有 ID，因此没有 ID 字段，无论是 Address 类还是 ValueObject 类。

Entity Framework 在 EF Core 2.0 以前不能使用没有 ID 字段的类，EF Core 2.0 对于更好实现没有 ID 的值对象提供了强大的支持，这将在下面小节精确地解释。

如何使用 EF Core 2.0 将值对象持久化到数据库中

您已经看到如何在您的领域模型中定义值对象。但是，如何通过 Entity Framework Core 将其存入通常目标是带有 ID 的实体的数据库中呢？

背景以及 EF Core 1.1 中的旧方法

作为背景或者历史，当使用 EF Core 1.0/1.1 的时候，您会遇到一个限制，您不能使用传统 .NET 框架的 EF 6.x 那样的[复合类型](#)。在 EF Core 1.0/1.1 中，您需要将您的值对象作为带有 ID 字段的实体来存储。为了让它看起来更像没有 ID 的值对象，您可能需要隐藏它的 ID，以便明确在领域模型中 ID 对于值对象是不重要的。您可以将 ID 作为[影子属性](#)。因此在模型中隐藏 ID 的配置是在 EF 基础设施层进行的。某种程度上，对于您的领域模型是透明的。

在最初版本的 eShopOnContainers (.NET Core 1.1) 中，隐藏 ID 需要通过 EF Core 基础设施中用下列方式实现，在基础设施项目中在 DbContext 层面上使用 Fluent API。然后，ID 从领域模型的视角看被隐藏了，在基础设施层仍然保留。

```
// Old approach with EF Core 1.1
// Fluent API within the OrderingContext:DbContext in the Infrastructure project
void ConfigureAddress(EntityTypeBuilder<Address> addressConfiguration)
{
    addressConfiguration.ToTable("address", DEFAULT_SCHEMA);

    addressConfiguration.Property<int>("Id") // Id is a shadow property
        .IsRequired();
    addressConfiguration.HasKey("Id"); // Id is a shadow property
}
```

然而，将值对象持久化到数据库中的操作，像在不同的表中放入常规的实体。

使用 EF Core 2.0。我们有更新和更好的办法来保存值对象。

在 EF Core 2.0 中以被持有实体类型持久化值对象

虽然正规的 DDD 中的值对象模式和 EF Core 的被持有的实体类型仍然有差距（看本节末尾的限制部分），但是它是 EF Core 2.0 保存值对象的最佳方式。

被持有的实体类型特性从 2.0 版开始加入到 EF Core 中。

被持有的的实体类型允许您在您的实体中映射没有在领域模型显式定义其 ID 并被用作属性的类型（比如值对象）。被持有的实体类型和其它实体类型共享 CLR 类型。实体中包含对其拥有者实体的导航的定义。当查询其拥有者，会默认包含其被持有的类型。

只要观察领域模型就会看到，被持有的类型看上去没有任何 ID，但是，在表面之下，被持有的类型有 ID，只是所有者的导航属性是此 ID 的一部分。

被持有类型的 ID 的实例不是它们完全拥有的，其包含三个组件，持有者的 ID，指向它们的导航属性，以及就被持有类型的集合来说（EF Core 2.0 尚未实现），一个独立的组件。

比如，在 eShopOnContainers 的订购领域模型中，作为 Order 实体的一部分，Address 值对象被实现为所有者实体 Order 实体内的被持有的实体类型。地址是在领域模型里定义的没有 ID 属性的类型。在 Order 类型中被作为属性用来指定某个订单的邮寄地址。

作为约定，所有者类型的影子主键将会被创建，并将会使用表切分同所有者一道映射到相同的表里。这将使得使用被持有的类型和在传统.NET 框架的 Entity Framework 6 使用的复合类型很相似。

请重点注意，按照约定，被持有的类型在 EF Core 中不会被发现。所以您得显式定义它们。

在 eShopOnContainers 的 OrderingContext.cs 中的 OnModelCreating()方法中有多个被应用的基础设施的配置项。其中一个和 Order 实体相关。

```
// Part of the OrderingContext.cs class at the Ordering.Infrastructure project
//
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.ApplyConfiguration(new ClientRequestEntityTypeConfiguration());
    modelBuilder.ApplyConfiguration(new PaymentMethodEntityTypeConfiguration());
    modelBuilder.ApplyConfiguration(new OrderEntityTypeConfiguration());
    modelBuilder.ApplyConfiguration(new OrderItemEntityTypeConfiguration());
    //...Additional type configurations
}
```

下述代码是持久化基础设施在 Order 实体中定义的位置。

```
// Part of the OrderEntityTypeConfiguration.cs class
//
public void Configure(EntityTypeBuilder<Order> orderConfiguration)
{
    orderConfiguration.ToTable("orders", OrderingContext.DEFAULT_SCHEMA);
    orderConfiguration.HasKey(o => o.Id);
    orderConfiguration.Ignore(b => b.DomainEvents);
    orderConfiguration.Property(o => o.Id)
        .ForSqlServerUseSequenceHiLo("orderseq", OrderingContext.DEFAULT_SCHEMA);

    //Address value object persisted as owned entity in EF Core 2.0
    orderConfiguration.OwnsOne(o => o.Address);

    orderConfiguration.Property<DateTime>("OrderDate").IsRequired();

    //...Additional validations, constraints and code...
    //...
}
```

上面的代码中 orderConfiguration.OwnsOne(o => o.Address); 方法指定 Address 属性为 Order 类的被持有实体。

EF Core 约定将为被持有的实体类型的属性的数据库列命名为 EntityProperty_OwnedEntityProperty。然后, Address 的内部属性将以 Address_Street, Address_City (州、国家和邮政编码类似) 的名字出现在 Orders 表中。

您可以追加 Property().HasColumnName() 这一 Fluent API 方法来重命名这些列, 当 Address 是公有属性的情况下, 映射如下所示:

```
orderConfiguration.OwnsOne(p => p.Address)
    .Property(p=>p.Street).HasColumnName("ShippingStreet");

orderConfiguration.OwnsOne(p => p.Address)
    .Property(p=>p.City).HasColumnName("ShippingCity");
```

可以在 Fluent API 的映射中链式调用 OwnsOne 方法。在下面的假想的例子里, OrderDetails 拥有 BillingAddress 和 ShippingAddress, 都是 Address 类型。Order 类型拥有 OrderDetails。

```
orderConfiguration.OwnsOne(p => p.OrderDetails, cb =>
    {
        cb.OwnsOne(c => c.BillingAddress);
        cb.OwnsOne(c => c.ShippingAddress);
    });

//...
//...
public class Order
{
    public int Id { get; set; }
    public OrderDetails OrderDetails { get; set; }
}

public class OrderDetails
{
    public StreetAddress BillingAddress { get; set; }
    public StreetAddress ShippingAddress { get; set; }
}

public class Address
{
    public string Street { get; set; }
    public string City { get; set; }
}
```

被持有的实体类型的更多细节:

- 被持有的类型的定义方式是使用 OwnsOne fluent API 配置导航属性到特定的类型。
- 在我们的元数据模型中的被持有的类型的定义由以下组成: 被持有类型, 导航属性以及被持有类型的 CLR 类型。
- 在我们的堆栈中被持有类型的 ID(键)的实例由被持有类型的 ID 和被持有类型的定义构成。

被持有实体的兼容性

- 被持有类型可以引用其它实体，无论是被持有（嵌套的被持有类型）还是非被持有（常规引用导航属性到其它实体）。
- 通过分离的导航属性作为相同所有者实体的不同的被持有类型，您可以映射相同的 CLR 类型。
- 表切分是由约定设置的，但是您可以通过 ToTable 映射被持有类型到另一个不同的表来实现自定义。
- 贪婪加载自动在被持有类型上执行，例如，不需要在查询中调用 Include()。

被持有实体的限制

- 不能创建被持有类型的 DbSet<T>（通过设计）。
- 不能在被持有类型上调用 modelBuilder.Entity<T>()（目前通过设计）
- 目前没有被持有类型的集合（但是在 EF Core 2.0 以后将被支持）
- 不支持通过特性来配置它们
- 不支持在相同表（比如使用表切分）中映射给持有者的被持有类型的可选（比如 nullable），这是因为我们没有 null 的单独的标识。
- 被持有类型不支持继承，但是您可以映射相同继承层次结构下的两个叶子类型作为不同的被持有类型。EF Core 将不会对它们是相同层次的事实进行推断。

和 EF6.x 的复合类型的主要不同

- 表切分是可选的，比如它们可以可选地被映射到分离的表中并且仍然是被持有的类型。它们可以引用其它实体（比如，它们可以作为其它非被持有类型的关系的依赖方）。

其他资源

- **Martin Fowler. 值对象模式**
<https://martinfowler.com/bliki/ValueObject.html>
- **Eric Evans. 领域驱动设计，软件核心复杂性应对之道**
<https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/>
- **Vaughn Vernon. 实现领域驱动设计**
<https://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon/dp/0321834577/>
- **影子属性**
<https://docs.microsoft.com/ef/core/modeling/shadow-properties>
- **复杂类型和/或值对象。在 EF Core GitHub 中的讨论 (Issues 标签)**
<https://github.com/aspnet/EntityFramework/issues/246>
- **eShopOnContainers 中的值对象基类，ValueObject.cs。**
<https://github.com/dotnet/eShopOnContainers/blob/masterdev/src/Services/Ordering/Ordering.Domain/SeedWork/ValueObject.cs>

- **eShopOnContainers 中的简单值对象类, Address 类。**
<https://github.com/dotnet/eShopOnContainers/blob/masterdev/src/Services/Ordering/Ordering.Domain/AggregatesModel/OrderAggregate/Address.cs>

使用枚举类代替 C#语言的 enum 类型

枚举（简称为 enum）是对单一完整类型的薄的语言包装器。您可能希望在存储封闭的一组值时限制为其中一个。比如基于性别的分类（比如男、女、未知）或者尺码（S、M、L、XL）。使用枚举来控制流程或者更鲁棒的抽象会变成一种**代码味道**。这种用法将产生很多控制流语句来检查枚举的值，导致代码变得脆弱。

替代方法是，您可以创建 Enumeration 类来利用面向对象语言的丰富的特性。

然而这不是一个重要的主题，并且在很多情况下，为了简化，如果您有这样的偏好，您仍然可以使用传统的枚举类型。

实现枚举类

eShopOnContainers 里的订单微服务提供了一个简化的 Enumeration 基类的实现，如下例所示：

```
public abstract class Enumeration : IComparable
{
    public string Name { get; private set; }

    public int Id { get; private set; }

    protected Enumeration()
    {
    }
    protected Enumeration(int id, string name)
    {
        Id = id;
        Name = name;
    }
    public override string ToString()
    {
        return Name;
    }
    public static IEnumerable<T> GetAll<T>() where T : Enumeration, new()
    {
        var type = typeof(T);
        var fields = type.GetTypeInfo().GetFields(BindingFlags.Public |
                                                    BindingFlags.Static |
                                                    BindingFlags.DeclaredOnly);

        foreach (var info in fields)
        {
            var instance = new T();
            var locatedValue = info.GetValue(instance) as T;

            if (locatedValue != null)
            {
                yield return locatedValue;
            }
        }
    }
}
```



```

public override bool Equals(object obj)
{
    var otherValue = obj as Enumeration;

    if (otherValue == null)
    {
        return false;
    }

    var typeMatches = GetType().Equals(obj.GetType());
    var valueMatches = Id.Equals(otherValue.Id);

    return typeMatches && valueMatches;
}

public int CompareTo(object other)
{
    return Id.CompareTo(((Enumeration)other).Id);
}

// Other utility methods ...
}

```

您可以在任何实体或者值对象中将这个类作为一个类型使用，如下面的 CardType 的枚举类：

```

public class CardType : Enumeration
{
    public static CardType Amex = new CardType(1, "Amex");
    public static CardType Visa = new CardType(2, "Visa");
    public static CardType MasterCard = new CardType(3, "MasterCard");

    protected CardType() { }
    public CardType(int id, string name)
        : base(id, name)
    {
    }

    public static IEnumerable<CardType> List()
    {
        return new[] { Amex, Visa, MasterCard };
    }
    // Other util methods
}

```

其他资源

- **Enum 是邪恶的——更新**
<http://www.planetgeek.ch/2009/07/01/enums-are-evil/>
- **Daniel Hardman. enum 是如何变成顽疾以及应对之策**
<https://codecraft.co/2012/10/29/how-enums-spread-disease-and-how-to-cure-it/>
- **Jimmy Bogard. 枚举类**
<https://losttechies.com/jimmybogard/2008/08/12/enumeration-classes/>
- **Steve Smith. C#的 enum 的替代方法**
<http://ardalis.com/enum-alternatives-in-c>

- **eShopOnContainers 的枚举基类, Enumeration.cs**
<https://github.com/dotnet/eShopOnContainers/blob/master/src/Services/Ordering/Ordering.Domain/SeedWork/Enumeration.cs>
- **eShopOnContainers 的枚举类的例子, CardType.cs**
<https://github.com/dotnet/eShopOnContainers/blob/master/src/Services/Ordering/Ordering.Domain/AggregatesModel/BuyerAggregate/CardType.cs>

在领域模型层设计验证

在 DDD 中，验证规则可以视作是不变的。聚合的主要职责是在其内的所有实体的跨状态的更改都强制要求不变性。

领域实体应该永远是有效的实体。对于一个对象，有特定数量的不变量总是为真。比如，对于订单项目对象。是订销量——永远是一个正整数，加上文章名字和价格。因此，对不变性的强制措施是领域实体（特别是聚合根）的职责，并且实体对象不允许存在无效的状态。不变性规则简单表述为契约以及异常或者当它们被违反的时候产生通知。

这么做背后的原因是，很多的 bug 都是因为对象出现它们所不该出现的状态而引起。Greg Young 在他的一次[在线讨论](#)中对此有很好的解释：

让我们假设现在有一个 SendUserCreationEmailService 服务，传一个 UserProfile，如何在那个服务中的 Name 字段为 null 时合理化？我们需要再次检查么？或者更可能地，您不为检查烦恼而“希望更好”——您希望别人在发送给您前为校验而烦恼。当然，使用 TDD 首先需要我们写的测试之一是当我们发送一个带有 null 的 name 的客户信息它会产生一个错误。但是当我们开始一遍一遍地写这种测试的时候我们意识到等等，“如果我们的名字永远不允许为空，那么这些测试也就用不到了”。

在领域模型层实现验证

验证通常在领域模型的构造方法或者更新实体的方法中实现。有很多办法实现验证。比如校验数据，当校验失败产生一个异常。还有更高级的模式，比如使用规格模式（Specification 模式）验证，以及通知模式，当每次验证出现失败的时候，用返回错误的集合代替返回一个异常。

校验条件并抛出异常

下面的代码例子显示了最简单的校验领域模型并抛出异常的方式。在本节末尾的参考中，您可以看到基于我们之前讨论过的模式的更高级的实现的链接。

```
public void SetAddress(Address address)
{
    _shippingAddress = address ?? throw new ArgumentNullException(nameof(address));
}
```

一个更好的例子，它将演示需要保证要么内部状态没有改变，要么在方法里所有的变化都改变。比如下面的实现将导致对象进入无效的状态：

```
Public void SetAddress(string line1, string line2,
                      string city, string state, int zip)
{
    _shippingAddress.line1 = line1 ?? throw new ...
    _shippingAddress.line2 = line2;
    _shippingAddress.city = city ?? throw new ...
    _shippingAddress.state = (IsValid(state) ? state : throw new ...);
}
```

如果状态的值是无效的，第一行的 Address 和 City 已经被改变了，这将导致地址无效。

一个简单的做法是使用实体的构造方法，通过抛出异常保证实体一旦创建就是有效的。

在模型中使用基于数据标记的验证特性

另一个方式是使用基于数据标记的验证特性。验证特性提供了配置模型验证的一种方式。和数据库表的字段的验证概念上类似。它包含了如数据类型和非空字段的约束。其它类型的验证包括按照业务规则强制应用的格式，比如信用卡卡号、电话号码或者电子邮件地址。使用验证特性可以简化强制实施这些需求。

然而，如下代码所示，这个方式对于 DDD 模型来说太具有侵入性，因为它从 Microsoft.AspNetCore.Mvc.ModelState 带来了对 ModelState.IsValid 的依赖，所以必须在 MVC 控制器里调用。模型验证在每个控制器动作被调用之前优先调用，控制器方法的职责是通过 ModelState.IsValid 的调用得到结果，并且做出适当处理。是否决定采用取决于您希望模型和基础设施层是否紧密耦合。

```
using System.ComponentModel.DataAnnotations;
// Other using statements ...
// Entity is a custom base class which has the ID

public class Product : Entity
{
    [Required]
    [StringLength(100)]
    public string Title { get; private set; }

    [Required]
    [Range(0, 999.99)]
    public decimal Price { get; private set; }

    [Required]
    [VintageProduct(1970)]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; private set; }

    [Required]
    [StringLength(1000)]
```

```
public string Description { get; private set; }

// Constructor...

// Additional methods for entity logic and constructor...
}
```

然而，从 DDD 的观点看，领域模型倾向实体行为方法中的异常的使用，或者实现用规格和通知模式来强制实施规则。像 ASP.NET Core 的数据标记或其它像 FluentValidation 一类的验证框架会产生对应用框架的调用。比如您调用数据标记中的 ModelState.IsValid，您需要调用 ASP.NET 控制器。

在应用层的 ViewModel 类（代替领域模型）中使用数据标记接受输入并在 UI 层数据验证是明智的。然而领域模型中的专有的验证不应该这么做。

通过实现规格模式和通知模式实现对实体的验证

最后，在领域模型中实现验证的一个更详尽的方法是使用规格模式加上通知模式实现。在之后的附加资源列表里有解释。

值得提起的是，您也可以只使用它们中的一个模式——比如，通过控制语句手动验证，但是使用通知模式去堆叠和返回验证错误的列表。

在领域中使用延迟验证

有很多种方式用来在领域中进行延迟验证，在 Vaughn Vernon 的书《[领域驱动设计的实现](#)》里，他在验证一节有讨论它们。

两步验证

同时考虑两步验证，在命令数据传输对象（DTO）上使用字段级别的验证以及实体内的领域级别的验证。您可以通过返回一个结果对象代替异常，以使得处理验证错误更简单。

使用数据标记的字段验证，比如，您不需要重复验证的定义。在 DTOs(比如命令和 ViewModel)里，可以同时客户端和服务端执行。

其他资源

- **Rachel Appel. ASP.NET Core MVC 模型验证入门**
<https://docs.microsoft.com/aspnet/core/mvc/models/validation>
- **Rick Anderson. 添加验证**
<https://docs.microsoft.com/aspnet/core/tutorials/first-mvc-app/validation>
- **Martin Fowler. 用验证中的通知代替抛出异常**
<https://martinfowler.com/articles/replaceThrowWithNotification.html>
- **规范通知模式**
<https://www.codeproject.com/Tips/790758/Specification-and-Notification-Patterns>
- **Lev Gorodinski. DDD 中的验证**
<http://gorodinski.com/blog/2012/05/19/validation-in-domain-driven-design-ddd/>

- **Colin Jack. 领域模型验证**
<http://colinjack.blogspot.com/2008/03/domain-model-validation.html>
- **Jimmy Bogard. 在 DDD 中的验证**
<https://lostechies.com/jimmybogard/2009/02/15/validation-in-a-ddd-world/>

客户端验证（表示层验证）

尽管其真正的来源是领域模型，最终您必须要在领域模型层验证，验证仍然需要同时在领域模型层（服务器端）和客户端处理。

客户端验证对于用户是非常便利的。它节约了用户的时间，否则用户将需要等待数据通过服务器绕一大圈来返回验证错误。在商业环境下，哪怕是几秒钟的耽搁，乘以每天几百次，都会累积成很长的时间、花费、挫折。直接和立即的验证使得用户更高效地工作，并且提高输入输出的质量。

就像视图模型和领域模型是不同的，视图模型验证和领域模型验证可能类似，但是服务于不同的目的。如果您关注到 DRY 原则（Don't Repeat Yourself），在这种情况下，代码重用也意味着耦合，在企业应用中，比起遵循 DRY 原则来说，不要耦合服务器端更重要。

尽管使用了客户端验证，您应该始终在服务器端验证您的命令或者输入的 DTOs，因为服务器 API 是一个可能的受攻击方向，通常，两边都做是您最好的赌注，因为您的应用程序，从 UX 的观点看，最好它是积极的，不允许用户输入无效的信息。

因此，在客户端代码中您通常验证视图模型。您也可以在发送 DTOs 或者命令到服务之前验证客户端的输出。

客户端验证的实现取决于您构建的客户端应用的类型。对于在 MVC 中编写大部分代码的 web 应用程序，大部分验证代码用 JavaScript 或者 TypeScript 编写的 SPA（单页面应用），以及使用 Xamarin 和 C# 编写的移动应用，它们是不同的。

其他资源

Xamarin 移动应用中的验证

- **验证文本输入并显示错误**
https://developer.xamarin.com/recipes/ios/standard_controls/text_field/validate_input/
- **验证回调**
<https://developer.xamarin.com/samples/xamarin-forms/XAML/ValidationCallback/>

ASP.NET Core 应用中的验证

- **Rick Anderson. 添加验证**
<https://docs.microsoft.com/aspnet/core/tutorials/first-mvc-app/validation>

单页 Web 应用（SPA）（Angular 2, TypeScript, JavaScript）中的验证

- **Ado Kukic. Angular 2 表单验证**
<https://scotch.io/tutorials/angular-2-form-validation>
- **表单验证**
<https://angular.io/docs/ts/latest/cookbook/form-validation.html>
- **验证.** Breeze 文档.
<http://breeze.github.io/doc-js/validation.html>

作为总结，以下是验证方面最重要的概念：

- 实体和聚合应该强制保持其一致性，并做到始终有效。聚合根在相同的聚合内负责多实体的一致性。
- 如果您考虑一个实体需要进入无效的状态，应考虑使用不同的对象模式，例如使用临时 DTO，直到创建最终的领域实体。
- 如果您需要创建几个相关的对象，例如聚合，并且只有在创建完所有对象后才有效，请考虑使用工厂模式。
- 验证框架最好在特定的层中使用，例如表示层或应用程序/服务层，但通常不在领域模型层，因为您需要其强依赖于基础设施框架。
- 在大多数情况下，在客户端进行冗余验证是好的，因为应用程序应该是积极的。

领域事件的设计和实现

使用领域事件来明确地实现领域内更改的副作用。换句使用 DDD 术语的话来说就是，使用领域事件来显式实现跨多个聚合的副作用。作为可选方案，为了更好的可扩展性和对数据库锁更小的影响，可以使用在同一领域的聚合之间的最终一致性。

什么是领域事件？

事件是指过去发生的事情。领域事件在逻辑上是特定领域中发生的事情，并且您希望同一领域（进程内）的其他部分知道并可能对其作出反应的事情。

领域事件的一个重要优点是事件发生后的副作用可以很明确而不是隐含的。这些副作用必须一致，要么所有与业务相关的操作都会发生，要么一个都不发生。此外，领域事件可以更好地分离同一领域的类的关注点。

例如，如果您只是使用 Entity Framework 和实体甚至是聚合，如果有一个用例必须引发副作用，则在发生事件后，它们将被隐含在耦合的代码中。如果只是看到这段代码，可能不知道该代码（副作用）是主要操作的一部分还是真的是副作用。另一方面，可以使用领域事件使概念明确，并且作为通用语言的一部分。例如，在 eShopOnContainers 应用中，创建订单不仅仅是关于订单，它会根据原始用户更新或创建买家聚合，因为用户在下订单之前还不是买家。如果您使用领域事件，您可以根据领域专家提供的通用语言明确表达该领域规则。

领域事件有点类似于消息风格的事件，但有一个重要的区别。使用实时消息、消息队列、消息代理或使用 AMQP 的服务总线时，消息总是以异步方式发送，并跨越进程和计算机进行通信。这对于集成多个限界上下文、微服务或者甚至不同的应用来说是有用的。但是，对于领域事件，您可能希望从当前正在运行的领域操作来引发事件，并且希望任何副作用都发生在同一个领域中。

领域事件及其副作用（由事件处理程序管理的后续触发的操作）几乎会立即发生，通常在进程内并在同一个领域中。因此，领域事件可以是同步或异步的。但是集成事件应该始终是异步的。

领域事件和集成事件

从语义上来看，领域和集成事件是相同的：关于刚刚发生的事情的通知。但是，它们的实现必须是不同的。领域事件只是推送到领域事件调度器的消息，它可以作为基于 IoC 容器或任何其他方法的内存调解器来实现。

另一方面，集成事件的目的是将提交的事务和更新传播到附加的子系统，无论它们是其他的微服务、限界上下文还是外部应用。因此，只有当实体成功持久化时才会出现这种情况，因为在多数情况下，如果失败则整个操作就不会发生。

此外，如上所述，集成事件必须基于多个微服务（其他限界上下文）甚至是外部系统或应用之间的异步通信。因此，事件总线接口需要一些可以允许潜在的远程服务之间进行进程内和分布式通信的基础架构。它可以基于商业的服务总线、队列、用作邮箱的共享数据库，或任何其他分布式的、理想的基于推送的消息系统。

领域事件作为在同一领域内多个聚合触发副作用的首选方式

如果执行与一个聚合实例相关的命令需要在一个或多个其他聚合上运行其他领域规则，则应该设计并实现由领域事件触发那些副作用。如图 9-14 所示，作为最重要的用例之一，在同一领域模型中应该使用领域事件在多个聚合之间传播状态更改。

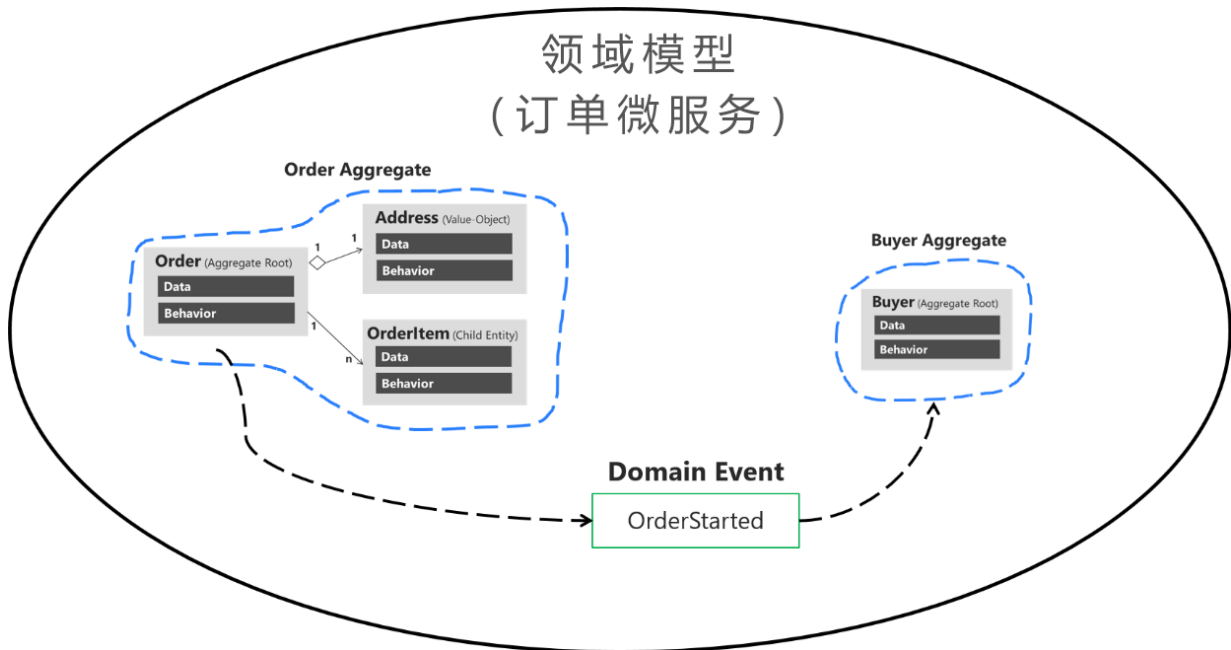


图9-14. 用领域事件来强制同一领域内多个聚合之间的一致性

在图中，当用户创建订单时，OrderStarted 领域事件将触发在订单微服务中根据来自身份微服务的原始用户信息（使用 CreateOrder 命令中提供的信息）创建 Buyer 对象。领域事件是由首先创建的订单聚合生成的。

另外，可以订阅聚合根来获取成员（子实体）提供的事件。例如，每个 OrderItem 子实体可以在单价高于特定数值时产生事件，或者当产品数量过多时引发事件。聚合根可以接收这些事件并执行全局计算或聚合。

重要的是要理解这种基于事件的通信不是直接在聚合内实现的，而是需要实现领域事件处理程序。处理领域事件是应用程序关注的问题。领域模型层应该只关注领域逻辑 - 领域专家理解的东西，而不是应用程序基础架构比如处理程序和使用存储库的副作用持久化操作。因此，当领域事件被引发，应该在应用程序层面具有能触发操作的领域事件处理程序。

领域事件也可用于触发任何数量的应用程序操作，更重要的是，必须是开放的，在未来可以以解耦的方式增加数量。例如，当下订单时，可能需要发布领域事件以将该信息传播到其他聚合，甚至可以发起诸如通知之类的应用程序操作。

关键点是当领域事件发生时要执行的操作的开放数量。最终领域和应用中的操作和规则将会增多。当事件发生时，副作用的复杂性或数量会增加，但是如果代码是“胶合”（即仅使用 C# 的 `new` 关键字实例化对象）的，则每次需要添加新操作时都需要修改原始代码。这可能会导致新的 bug，因为每个新的需求都需要修改原始代码。这也违反了 SOLID 的 [开闭原则](#)。不仅如此，编排操作的原始类将会越来越多，这违反了 [单一职责原则 \(SRP\)](#)。

另一方面，如果您使用领域事件，则可以通过使用下列方法分离职责来创建细粒度的解耦的实现：

1. 发送命令（例如 `CreateOrder`）。
2. 在命令处理程序中接收命令。
 - 执行单个聚合的事务。
 - （可选）发起副作用的领域事件（例如 `OrderStartedDomainEvent`）。
3. 处理将在多个聚合或应用程序操作中执行开放数量的副作用的领域事件（在当前进程中）。例如：
 - 验证或创建买家和付款方式。
 - 创建并发送相关的集成事件到事件总线，以跨微服务传播状态或触发外部操作，例如向买家发送电子邮件。
 - 处理其他副作用。

如图 9-15 所示，从同一个领域事件开始可以有多个处理操作，它们可以是和领域中的其他聚合相关的操作，或者，也可以是您在跨微服务连接集成事件和事件总线是需要的额外的应用程序操作。

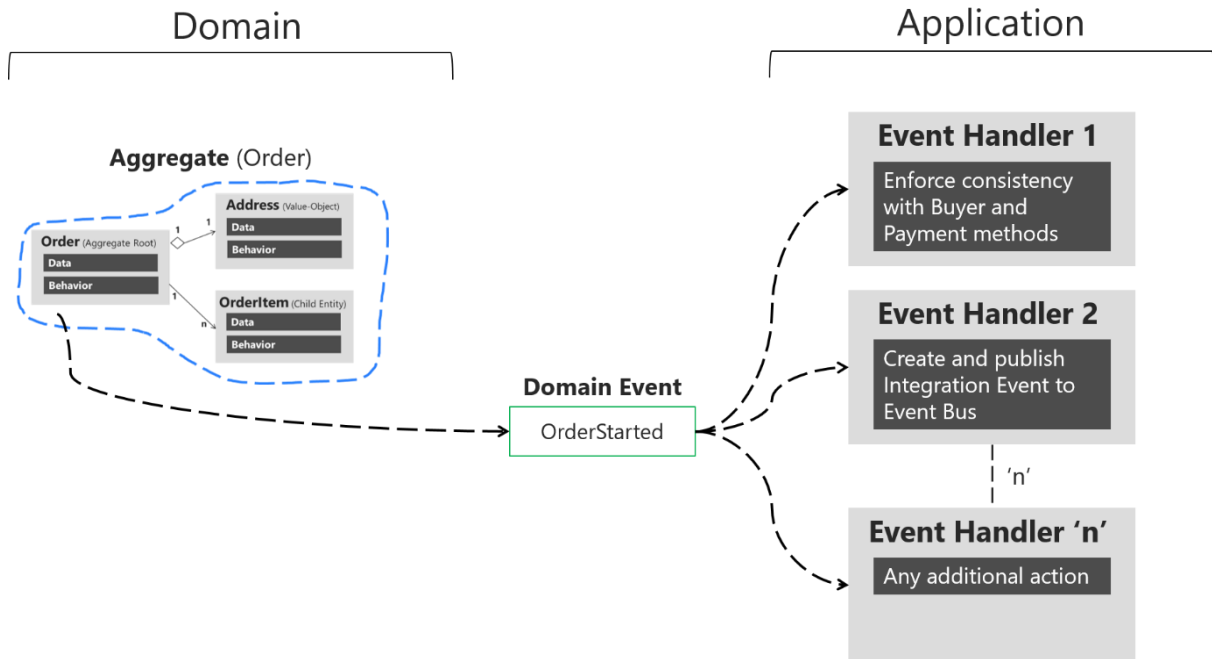


图9-15. 一个领域中处理多个操作

事件处理程序通常在应用程序层，因为您需要使用基础架构的对象（如存储库或应用 API）来实现微服务的行为。在这个场景下，事件处理程序类似于命令处理程序，所以这两者都是应用程序层的一部分。重要的区别在于一个命令应该只被处理一次，领域事件可以被处理 0 或 n 次，因为可以由具有不同目的的多个事件处理程序或多个接收者接收。

每个领域事件的处理程序开放数量的可能性允许添加更多领域规则而不影响当前的代码。例如，实现如下在一个事件后立即发生的业务规则，或许简单的添加一些事件处理程序（甚至只需要一个）：

当客户在商店中购买的所有订单总额超过 6,000 美元时，每个新订单都可以享受 10% 的优惠并发送电子邮件通知客户。

实现领域事件

在 C# 中，领域事件只是一个类似 DTO 的数据存储结构或类，其中包含与领域中刚刚发生的事件相关的所有信息，如下例所示：

```
public class OrderStartedDomainEvent : INotification
{
    public string UserId { get; private set; }
    public int CardTypeId { get; private set; }
    public string CardNumber { get; private set; }
    public string CardSecurityNumber { get; private set; }
    public string CardHolderName { get; private set; }
    public DateTime CardExpiration { get; private set; }
    public Order Order { get; private set; }

    public OrderStartedDomainEvent(Order order,
                                    int cardTypeId, string cardNumber,
                                    string cardSecurityNumber, string cardHolderName,
                                    DateTime cardExpiration)
    {
        Order = order;
        CardTypeId = cardTypeId;
        CardNumber = cardNumber;
        CardSecurityNumber = cardSecurityNumber;
        CardHolderName = cardHolderName;
        CardExpiration = cardExpiration;
    }
}
```

这本质上是一个包含与 OrderStarted 事件相关的素所有数据的类。

就领域的通用语言来讲，由于事件是过去发生的事情，事件的类名应该表示为动词的过去式，如 OrderStartedDomainEvent 或 OrderShippedDomainEvent。这就是 eShopOnContainers 中的订单微服务是如何实现领域事件的。

正如已经指明的，事件的一个重要特征是它表示过去发生的事情，所以不应该改变。因此它必须是一个不可变的类。在上述代码中可以看到属性对外部都是只读的。更新对象的唯一方法是在创建事件对象时通过构造方法。

引发领域事件

下一个问题是如何引发领域事件，以及到达相关的事件处理程序。可以使用很多种方法。

Udi Dahan 最初提出（例如一些相关的文章比如 [“Domain Events – Take 2”](#)）使用静态类来管理和引发事件。可能是一个名为 DomainEvents 的静态类，它会在调用比如 DomainEvents.Raise(Event myEvent) 这样的方法时立即引发领域事件。Jimmy Bogard 写了一篇博客 [“Strengthening your domain: Domain Events”](#)，也建议采用类似的方法。

但是，当领域事件类为静态时，它也会立即发送到处理程序。这使得测试和调试更加困难，因为在事件发生后立即执行了具有副作用逻辑的事件处理程序。当进行测试和调试时，只需要关注当前聚合类中发生的情况，而不想突然被重定向到其他事件处理程序，这些事件处理程序与其他聚合或应用逻辑是相关的，具有副作用。

这就是为什么发展出了其他的方法，如下一节所述。

延迟引发和分派事件的方式

相对于立即派送到领域事件处理程序，更好的方法是将领域事件添加到一个集合中，然后在提交事务之前或之后发送这些领域事件（与 EF 中的 `SaveChanges` 一样）。（这个方法由 Jimmy Bogard 在博文 [“A better domain events pattern”](#) 中描述。）

决定是否在提交事务之前或之后发送领域事件很重要，因为它将确定是否将副作用作为一部分包含在同一个事务中或者在不同的事务中。在后一种情况下，您需要处理多个聚合中的最终一致性。本主题将在下一章讨论。

延迟方式是 `eShopOnContainers` 使用的方法。首先，将实体中发生的事件添加到每个实体的一个集合或事件列表中。这个列表应该是实体对象的一部分，或者最好的是实体类的基类的一部分，如下示例的基类 `Entity` 所示：

```
public abstract class Entity
{
    //...
    private List<INotification> _domainEvents;
    public List<INotification> DomainEvents => _domainEvents;

    public void AddDomainEvent(INotification eventItem)
    {
        _domainEvents = _domainEvents ?? new List<INotification>();
        _domainEvents.Add(eventItem);
    }

    public void RemoveDomainEvent(INotification eventItem)
    {
        if (_domainEvents is null) return;
        _domainEvents.Remove(eventItem);
    }
    //... Additional code
}
```

当您想引发一个事件时，只需要在聚合根实体的任何方法代码中将它添加到事件集合里，如下面的代码所示，这是 `eShopOnContainers` 的 `Order` 聚合根的一部分：

```
var orderStartedDomainEvent = new OrderStartedDomainEvent(this, //Order object
    cardTypeId, cardNumber,
    cardSecurityNumber,
    cardHolderName,
```

```
cardExpiration);  
this.AddDomainEvent(orderStartedDomainEvent);
```

请注意，AddDomainEvent 方法唯一做的事情是将事件添加到列表中，不会分发事件，也不会调用事件处理程序。

当您提交事务到数据库时，实际上希望稍后分派事件。如果正在使用 Entity Framework Core，这会在 EF DbContext 的 SaveChanges 方法中发生，如以下代码所示：

```
// EF Core DbContext  
public class OrderingContext : DbContext, IUnitOfWork  
{  
    // ...  
    public async Task<bool> SaveEntitiesAsync(CancellationToken cancellationToken =  
                                             default(CancellationToken))  
    {  
        // Dispatch Domain Events collection.  
        // Choices:  
        // A) Right BEFORE committing data (EF SaveChanges) into the DB. This makes  
        // a single transaction including side effects from the domain event  
        // handlers that are using the same DbContext with Scope lifetime  
        // B) Right AFTER committing data (EF SaveChanges) into the DB. This makes  
        // multiple transactions. You will need to handle eventual consistency and  
        // compensatory actions in case of failures.  
        await _mediator.DispatchDomainEventsAsync(this);  
  
        // After this line runs, all the changes (from the Command Handler and Domain  
        // event handlers) performed through the DbContext will be committed  
        var result = await base.SaveChangesAsync();  
    }  
    // ...  
}
```

使用这段代码，将实体事件分派到各自的事件处理程序。总体的结果是将引发领域事件（一个简单的添加到内存列表中）和分派到事件处理程序解耦了。另外，根据使用的分派程序，您可以同步或异步地分派事件。

请注意，事务边界在这里变得很明显。如果您的工作单位和交易单位可以跨越多个聚合（比如使用 EF Core 和关系数据库），这可以很好地工作。但如果事务不能跨聚合，例如当您使用 NoSQL 数据库（比如 Azure 文档数据库（Azure DocumentDB））时，必须执行额外的步骤来实现一致性。这是持久性忽略没有那么普遍的另一个原因，这也取决于使用的存储系统。

跨聚合的单个事务与跨聚合最终一致性

T 是跨聚合执行单个事务，还是依赖于跨聚合的最终一致性，这是一个有争议的问题。许多 DDD 作者比如 Eric Evans 和 Vaughn Vernon 倡导一个事务 = 一个聚合的规则，因此对跨聚合的最终一致性一直有争议。例如，Eric Evans 在他的《领域驱动设计》一书中表示：

不要期望任何跨越聚合的规则在任何时候都是最新的。通过事件处理、批处理或其他更新机制，其他的依赖可以在特定时间内解决。（第 128 页）

Vaughn Vernon 在文章 [有效聚合设计，第二部分：使聚合协同工作](#) 中说到如下观点：

因此，如果在一个聚合实例上执行命令需要在一个或多个聚合上执行额外的业务规则，就使用最终一致性[...] 有一种实用的方式来支持 DDD 模型中的最终一致性。一个聚合方法应该发布一个及时传递给一个或多个异步订阅者的领域事件。

这个原理是基于细粒度的事务，而不是跨越多个聚合或实体的事务。这个想法来自于第二种情况下，数据库锁的数量在具有高可扩展性需求的大规模应用程序中将是巨大的。拥抱高可扩展应用不需要在多个聚合之间具有即时事务一致性的事实有助于接受最终一致性的概念。业务往往不需要原子性的更改，无论如何，领域专家有责任说明特定操作是否需要原子事务。如果操作总是需要在多个聚合之间进行原子事务处理，那么您可能会询问聚合是否应该更大或者是没有正确地设计。

然而，其他开发人员和架构师（像 Jimmy Bogard 等）也会通过跨多个聚合来执行单个事务，但只有当这些额外的聚合与相同的原始命令的副作用相关时才可以。例如，Bogard 在文章 “一种更好的领域事件模式” 中说道：

通常，我希望领域事件的副作用发生在相同的逻辑事务中，但不一定在引发领域事件的相同范围 [...] 在我们提交事务之前，我们将事件分派到相应的处理程序。

如果在提交原始事务之前分派领域事件，那是希望您希望将这些事件的副作用包含在同一个事务中。例如，如果 EF DbContext 的 SaveChanges 方法失败，则事务将回滚所有更改，包括由相关的领域事件处理程序实现的任何副作用操作的结果。这是因为 DbContext 的生命作用范围默认定义为 “范围的”。因此，DbContext 对象在同一范围或对象树中被实例化的多个存储库对象共享。这与开发 Web API 或 MVC 应用时的 HttpRequest 的范围是一致的。

实际上，这两种方法（单原子事务和最终一致性）都是正确的。这真的取决于您的领域或业务需求以及领域专家告诉的内容。它还取决于需要怎样扩展服务（更细粒度的事务对数据库锁的影响较小）。也取决于您愿意在代码中做出多少投入，因为最终一致性需要更复杂的代码，以便检测跨聚合的可能不一致以及需要实现的补偿措施。考虑到如果对原始聚合进行更改之后，当事件已经被分派后出现问题并且事件处理程序无法提交其副作用，则聚合之间将存在不一致。

一种允许补偿措施的方法是将领域事件存储在额外的数据库表中，以便它们可以作为原始事务的一部分。之后，您可以通过批处理将事件列表与聚合的当前状态进行比较来检测不一致并运行补偿措施。补偿措施是一个复杂主题的一部分，需要您进行深入的分析，其中包括与业务用户和领域专家进行讨论。

无论如何，您可以选择自己需要的方法。但是最初的延迟方法 - 即使用单个事务在提交之前引发事件 - 是使用 EF Core 和关系数据库时最简单的方法。在许多业务场景中它很容易实现并且是可用的。它也是 eShopOnContainers 中订单微服务使用的方法。

但是，实际上是如何将这些事件分派到各自的事件处理程序的？您在上一个示例中看到的 `_mediator` 对象是什么？这与您可以用来映射事件和事件处理程序的技术和工件有关。

领域事件分派程序：从事件到事件处理程序的映射

一旦能够分派或发布事件，您需要一些发布事件的工件以便每个相关的处理程序都可以获取并处理基于该事件的副作用。

一种方法是真正的消息系统或者事件总线，可能是基于服务总线而不是内存中的事件。然而对于第一种情况，真正的消息系统对处理领域事件是过度的，因为只需要在相同进程（即同一领域和应用程序层）内处理这些事件。

将事件映射到多个事件处理程序的另一种方法是在 IoC 容器中使用类型注册，以便可以动态地推断将事件分派到哪里。换句话说，您需要知道什么事件处理程序需要获取特定的事件。图 9-16 展示了一个简化的方法。

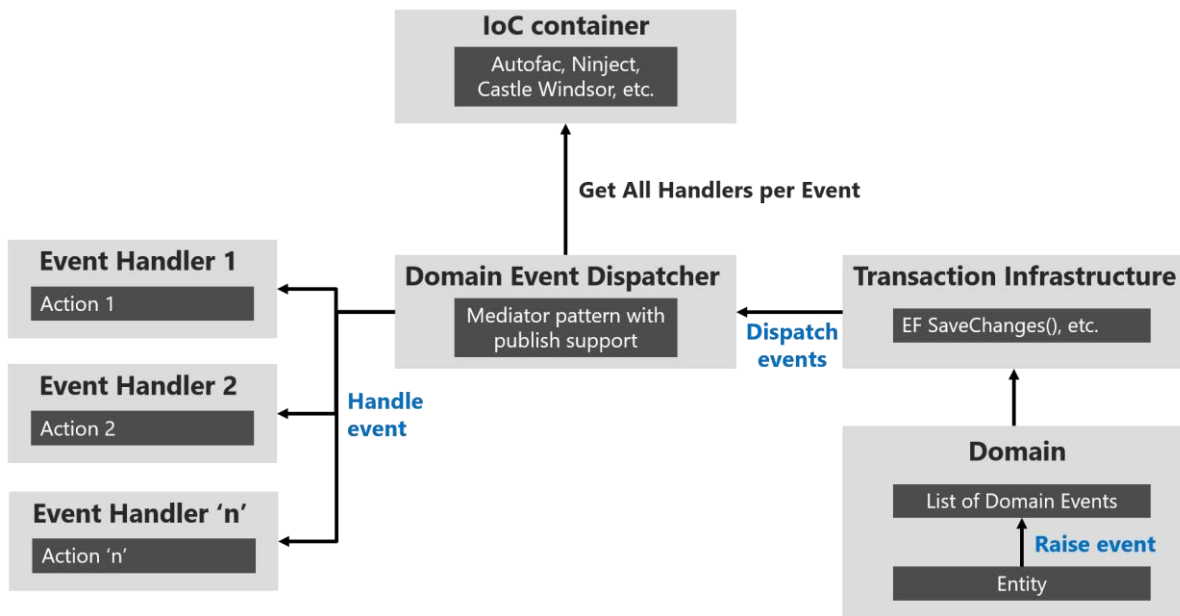


图9-16. 使用 IoC 的领域事件分派

您可以创建所有的管道和工件来自己实现这种方法。但是，您也可以使用（如 [MediatR](#) 等）可用的库，其中包含您需要使用的 IoC 容器。因此可以直接使用预定义的接口和调解器对象的发布/分派方法。

在代码中，首先需要在 IoC 容器中注册事件处理程序的类型，如下 [eShopOnContainers 的订单微服务](#) 的代码所示：

```
public class MediatorModule : Autofac.Module
{
```

```

protected override void Load(ContainerBuilder builder)
{
    // Other registrations ...
    // Register the DomainEventHandler classes (they implement
    // IAsyncNotificationHandler<>) in assembly holding the Domain Events

    builder.RegisterAssemblyTypes(
        typeof(ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler)
            .GetTypeInfo().Assembly)
        .AsClosedTypesOf(typeof(IAsyncNotificationHandler<>));

    // Other registrations ...
    //...
}
}

```

代码中首先通过定位包含任何处理程序的程序集（使用 `typeof(ValidateOrAddBuyerAggregateWhenXxxx)`，但您可以选择任何其他事件处理程序来定位程序集）来标识包含领域事件处理程序的程序集。由于所有事件处理程序都实现了 `IAsyncNotificationHandler` 接口，所以代码只需要搜索这些类型然后注册所有的事件处理程序。

如何订阅领域事件

当您使用 MediatR 时，每个事件处理程序必须使用 `IAsyncNotificationHandler` 接口的通用参数提供的事件类型，如下面的代码所示：

```

public class ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler
    : IAsyncNotificationHandler<OrderStartedDomainEvent>

```

根据事件和事件处理程序之间的关系，可以将其视为订阅，MediatR 工件可以发现每个事件的所有事件处理程序并触发每个事件处理程序。

如何处理领域事件

最后，事件处理程序通常实现应用程序层的代码，这些代码使用基础架构存储库来获取所需的额外的聚合，并执行具有副作用的领域逻辑。下列 [eShopOnContainers 的领域事件处理代码](#) 展示了一种实现方式。

```

public class ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler
    : INotificationHandler<OrderStartedDomainEvent>
{
    private readonly ILoggerFactory _logger;
    private readonly IBuyerRepository<Buyer> _buyerRepository;
    private readonly IIdentityService _identityService;

```



```

public ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler(
    ILoggerFactory logger,
    IBuyerRepository<Buyer> buyerRepository,
    IIdentityService identityService)
{
    // ...Parameter validations...
}

public async Task Handle(OrderStartedDomainEvent orderStartedEvent)
{
    var cardTypeId = (orderStartedEvent.CardTypeId != 0) ?
        orderStartedEvent.CardTypeId : 1;

    var userGuid = _identityService.GetUserIdentity();

    var buyer = await _buyerRepository.FindAsync(userGuid);
    bool buyerOriginallyExisted = (buyer == null) ? false : true;

    if (!buyerOriginallyExisted)
    {
        buyer = new Buyer(userGuid);
    }

    buyer.VerifyOrAddPaymentMethod(cardTypeId,
        $"Payment Method on {DateTime.UtcNow}",
        orderStartedEvent.CardNumber,
        orderStartedEvent.CardSecurityNumber,
        orderStartedEvent.CardHolderName,
        orderStartedEvent.CardExpiration,
        orderStartedEvent.Order.Id);

    var buyerUpdated = buyerOriginallyExisted ? _buyerRepository.Update(buyer) :
        _buyerRepository.Add(buyer);

    await _buyerRepository.UnitOfWork.SaveEntitiesAsync();

    // Logging code using buyerUpdated info, etc.
}
}

```

上面的事件处理程序代码被认为是应用程序层代码，因为它使用了基础架构存储库，如在下一章介绍的基础架构持久层。事件处理程序也可以使用其他基础架构组件。

领域事件能生成发布到微服务边界之外的集成事件

最后提醒您可能有时想要跨多个微服务传播事件也是很重要的。这被认为是一个集成事件，它可以通过事件总线从任何特定的领域事件处理程序发布。

领域事件的结论

如上所述，使用领域事件可以明确地实现领域内更改的副作用。使用 DDD 术语来说明的话，就是使用领域事件来显式实现一个或多个聚合的副作用。另外，为了更好的可扩展性和对数据库锁较小的影响，请使用同一领域内聚合之间的最终一致性。

其他资源

- **Greg Young. 什么是领域事件**
<http://codebetter.com/gregyoung/2010/04/11/what-is-a-domain-event/>
- **Jan Stenberg. 领域事件和最终一致性**
<https://www.infoq.com/news/2015/09/domain-events-consistency>
- **Jimmy Bogard. 一种更好的领域事件模式**
<https://lostechies.com/jimmybogard/2014/05/13/a-better-domain-events-pattern/>
- **Vaughn Vernon. 有效聚合设计，第二部分：使聚合协同工作**
http://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_2.pdf
- **Jimmy Bogard. 强化您的领域：领域事件**
<https://lostechies.com/jimmybogard/2010/04/08/strengthening-your-domain-domain-events/>
- **Tony Truong. 领域事件模式示例**
<http://www.tonytruong.net/domain-events-pattern-example/>
- **Udi Dahan. 如何创建完全封装的领域模型**
<http://udidahan.com/2008/02/29/how-to-create-fully-encapsulated-domain-models/>
- **Udi Dahan. 领域事件 — 2**
<http://udidahan.com/2008/08/25/domain-events-take-2/>
- **Udi Dahan. 领域事件 – 救赎**
<http://udidahan.com/2009/06/14/domain-events-salvation/>
- **Jan Kronquist. 不要发布领域事件，返回它们**
<https://blog.jayway.com/2013/06/20/dont-publish-domain-events-return-them/>
- **Cesar de la Torre. DDD 和微服务架构中的领域事件与集成事件**
<https://blogs.msdn.microsoft.com/cesardelatorre/2017/02/07/domain-events-vs-integration-events-in-domain-driven-design-and-microservices-architectures/>

设计持久层基础架构

数据持久层组件提供对于微服务内承载的数据（即微服务的数据库）的数据访问。其包含如仓储和[工作单元](#)的实际的组件实现，例如定制的 EF DbContext。

仓储模式

仓储是封装了访问数据源所需逻辑的类或者组件。其中心化公共的数据访问功能，提供最佳的维护性，并将领域模型层从用于访问数据库的基础架构或技术中进行了解耦。如果您使用过一种 ORM 技术，比如 Entity Framework，由于 LINQ 和强类型，实现的代码被简化了。这使得您可以专注于数据持久性逻辑而不是数据访问管道。

仓储模式是文档完备的使用数据源的方式。在《[企业应用架构模式](#)》一书中，Martin Fowler 如下描述了仓储：

仓储协调领域模型和数据映射层，起到类似内存中领域对象集合的作用。客户对象以声明的方式构建查询说明，并把他们提交给仓储以获取满足条件的对象。从概念上说，仓储封装了保存在数据存储中的对象集以及在这些对象上执行的操作，提供一个更加符合面向对象观点的持久层实现。仓储也支持在领域和数据分配或映射层之间实现彻底分离和单向依赖关系的目标。

为每个聚合定义一个仓储

对应于每个聚合或聚合根，您应当创建一个仓储类。在基于 DDD 模式的微服务中，更新数据库的唯一渠道应当是仓储。这是因为它们与聚合根是一一对应的关系。其控制聚合的不变性和事务一致性。也可以通过其它途径查询数据库（如 CQRS 方式），因为查询不改变数据库的状态。但是，事务领域 - 更新，必须始终由仓储和聚合根控制。

基本上，仓储允许您以领域实体的形式把来自数据库的数据填充到内存中。一旦实体处于内存中，它们就可以通过事务被修改并更新回数据库中。

如前所述，如果您使用 CQS/CQRS 架构模式，初始的查询将由领域模型中的查询侧执行，执行使用简短的 SQL 语句。这种方式比仓储灵活的多，因为您可以查询和连接任何所需的表，并且这些查询不受聚合中的规则所限制。这些数据将被用于表示层或者客户端应用。

如果用户对数据做出更新，被更新的数据将从客户端应用或者表示层来到应用层（比如 Web API 服务），当您在命令处理器中收到带有数据的命令，使用仓储从数据库中获取待更新的数据。您使用命令传递的信息在内存中更新数据，然后通过事务在数据库中添加或者更新数据（领域实体）。

我们必须再次强调，对于每个聚合根只应定义一个仓储，如图 9-17 所示。为了达到聚合根在聚合中的所有对象之间维护事务一致性的目的，您应当永远不要对数据库中的每张表创建一个仓储。

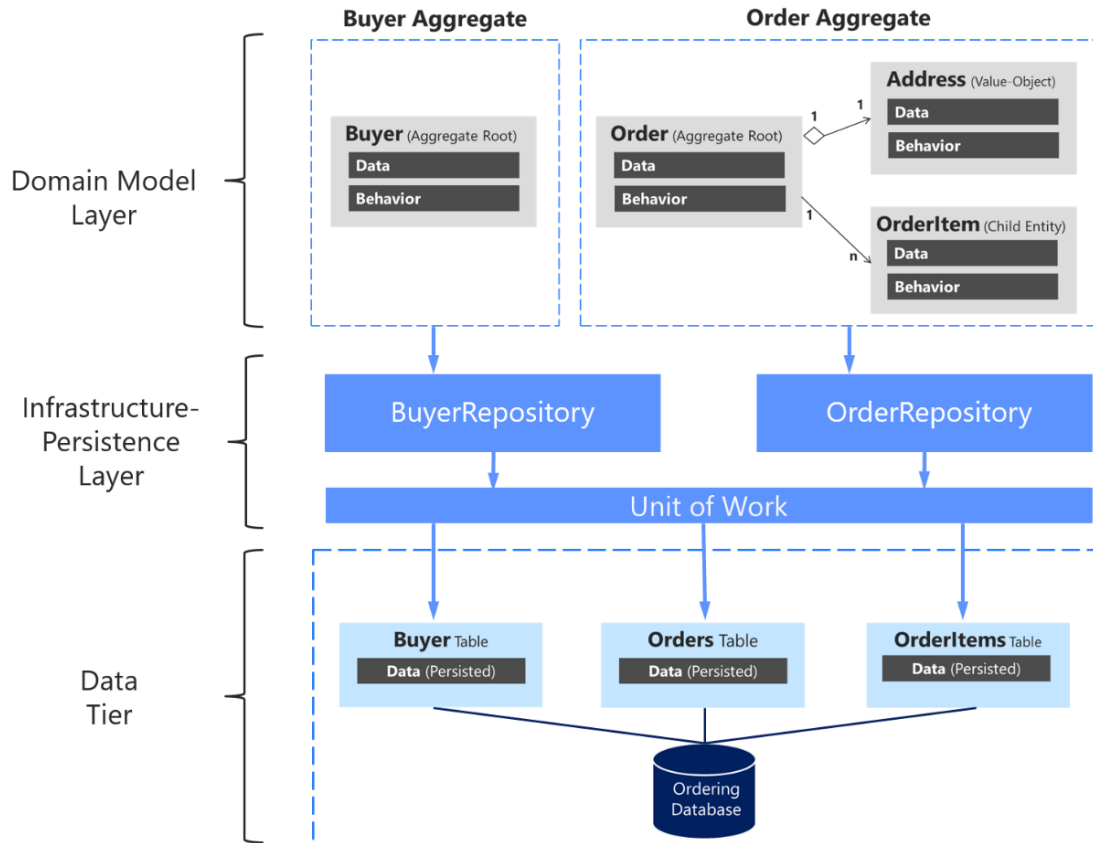


图9-17. 仓储、聚合与数据库之间的关系

强制每个仓储有一个聚合根

以这样的方式实现您的仓储设计是有价值的，强制只有聚合根拥有仓储。您可以创建一个泛型或者仓储基类，以约束它所使用的实体类型，确保它们具有 `IAggregateRoot` 标记接口。

然后，在基础架构层每个仓储类实现其自身的约定或者接口，代码如下所示：

```
namespace Microsoft.eShopOnContainers.Services.Ordering.Infrastructure.Repositories
{
    public class OrderRepository : IOrderRepository
    {
    }
}
```

每个特定的仓储接口实现泛型的 `IRepository` 接口：

```
public interface IOrderRepository : IRepository<Order>
{
    Order Add(Order order);
    // ...
}
```

其实，更好的方式是让代码强制执行约定，即每个仓储应当仅与单个聚合相关。实现一个泛型的仓储类型，以明确表示正在使用的仓储针对特定的聚合。通过在 `IRepository` 接口中使用泛型，可以很容易完成此操作，代码如下所示：

```
public interface IRepository<T> where T : IAggregateRoot
```

仓储模式使得测试应用程序逻辑变得简单

仓储模式使得更容易使用单元测试来测试您的应用程序。记住，单元测试仅仅测试代码，而不是基础架构，所以仓储概念的抽象使得易于达到这个目标。

如前所述，建议在领域模型层定义和构建仓储接口，以便应用层（例如，您的 Web API 微服务）不直接依赖于实际实现仓储类的基础架构层。通过这种方式和在 Web API 应用程序中的控制器使用依赖注入，您可以实现返回模拟数据而不是来自数据库数据的模拟的仓储。这种解耦的方式，使得您可以不需要连接到数据库，就可以创建并运行仅仅测试应用逻辑的单元测试。

与数据库的连接可能会失败，更重要的是，基于数据库运行成百的测试是错误的，原因有二。首先，大量的测试将会花费大量的时间。其次，数据库记录可能会发生变化从而影响测试结果，以致其不一致。基于数据库的测试是集成测试而不是单元测试。您应当有大量快速运行的单元测试，很少基于数据库的集成测试。

从关注点分离的角度来看单元测试，您的应用逻辑在内存中操作领域实体。其假定仓储类负责交付它们。一旦您的逻辑修改了领域实体，其假定仓储类将会正确保存它们。这里重要的是基于领域模型和其领域逻辑创建单元测试。在 DDD 中聚合根是主要的一致性边界。

仓储模式与旧式数据访问类 (DAL 类) 模式的区别

数据访问对象直接执行数据访问和基于存储的持久化操作。仓储在内存中将数据标记为工作单元对象上（在 EF 中使用 `DbContext`）执行的操作，但这些操作不会立即执行。

工作单元是执行多个插入、更新或者删除操作的单个事务单位。简单地说，其意味着一个特定的用户操作（例如，在站点注册），所有的插入、更新和删除事务在单个事务中处理。这比松散地处理多个数据库事务更为有效。

当来自应用层的代码下达命令的时候，多个持久化操作随后执行。将内存中的修改应用到实际数据库存储中的决策通常基于[工作单元模式](#)。在 EF 中，工作单元模式由 `DbContext` 实现。

在多数场景下，对存储运用这种操作模式或者方式可以增强应用程序的性能并降低不一致的可能性。此外，它还减少了在数据库表上的事务阻塞，因为所有的意向操作被作为单个事务的一部分提交。与对数据库执行许多独立的操作相比，这种方式更为有效。因此，所选中的 ORM 能够通过分组同一事务中的多个更新操作，而不是多个小的和单独的事务操作，来优化数据库的执行。

仓储不是强制的

基于前面的原因，创建自定义仓储是有用的，这也是在 eShopOnContainers 中订单微服务的方式。但是在 DDD 设计中，甚至在 .NET 的常规开发中，它并不是必要的模式。

例如，Jimmy Bogard 在为本书提供直接反馈时说：

这可能是我最重大的反馈。我真的不喜欢仓储，主要因为它们隐藏了底层持久化机制的重要细节。这也是为什么我通过 Dediator 使用 Command 的原因。我可以使用持久层全部的能力，并将所有的领域行为推入聚合根。我通常不会 mock 我的仓储 - 我仍然需要与实际的东西进行集成测试。CQRS 意味着我们不再需要仓储了。

我们发现仓储很有用，但是我们承认其对您的 DDD 设计并不重要，就像聚合模式和富领域模型一样。因此，请按您认为合适的方式使用仓储模式。

其他资源

仓储模式

- **Edward Hieatt 和 Rob Mee. 仓储模式**
<http://martinfowler.com/eaCatalog/repository.html>
- **仓储模式**
<https://msdn.microsoft.com/library/ff649690.aspx>
- **仓储模式：一种数据持久化抽象**
<http://deviq.com/repository-pattern/>
- **Eric Evans. 领域驱动设计：软件核心复杂性应对之道（书籍：包括仓储模式的讨论）**
<https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/>

工作单元模式

- **Martin Fowler. 工作单元模式**
<http://martinfowler.com/eaCatalog/unitOfWork.html>
- **在 ASP.NET MVC 应用中实现仓储和工作单元模式**
<https://www.asp.net/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application>

规格模式

规格模式，全称查询规格 (Query-specification) 模式，是一个领域驱动设计模式，被设计为用来表示对带有可选的排序和分页逻辑定义的查询。

规格模式通过一个对象来定义查询。例如，为了封装搜索某些产品的分页查询，您可以创建一个带有必要输入参数（pageNumber, pageSize, filter 等）的 PagedProduct 规格。然后，在任何 Repository 方法（通常是一个 List () 重载）中，接受一个 ISpecification 对象，并根据该规格运行预期的查询。

这种方式有几个好处。

- 规格有一个您可以讨论的名称（而不仅仅是一堆 LINQ 表达式）。
- 规格可以单独进行单元测试以确保它是正确的。如果您需要类似的行为（例如，在 MVC 视图操作和 Web API 操作以及各种服务中），可以轻松地重用它。
- 也可以使用规格来描述要返回的数据的形状，以便查询可以仅返回所需的数据。这消除了 Web 应用程序中进行延迟加载的需要（这通常不是一个好主意），并有助于使存储库实现免受这些细节的干扰。

下面通用规格的示例代码来自 [eShopOnweb](#) ：

```
// https://github.com/dotnet-architecture/eShopOnWeb
public interface ISpecification<T>
{
    Expression<Func<T, bool>> Criteria { get; }
    List<Expression<Func<T, object>>> Includes { get; }
    List<string> IncludeStrings { get; }
}
```

在随后的章节中，将会说明如何使用 Entity Framework Core 2.0 实现规格模式，以及如何从任何 Repository 类中使用它。

重要提示：规格模式是一种悠久的模式，可以使用不同的方式实现，如下面其它资源中所示。作为一种模式/思想，了解以前的处理方式很好，但是要留意那些没有利用现代语言功能（如 Linq 和表达式）的老版本。

其他资源

- **规格模式**
<http://deviq.com/specification-pattern/>
- **Evans, Eric (2004). 领域驱动设计 Addison-Wesley. 第 224 页.**
- **Martin Fowler: 规格**
<https://www.martinfowler.com/apSUPP/spec.pdf>

使用 Entity Framework Core 实现持久层基础架构

在使用关系数据库，比如：SQL Server、Oracle 或者 PostgreSQL 时，建议的方式是基于 Entity Framework (EF) 实现持久层。EF 支持 LINQ 并对您的模型提供强类型对象，同样还有简化的持久性数据库操作。

作为 .NET Framework 的一部分，Entity Framework 有很长的历史。在使用 .NET Core 时，也应该使用 Entity Framework Core，像 .NET Core 一样，它可以运行在 Windows 或者 Linux 上。EF Core 对 Entity Framework 进行了完全的重写，实现了更小的体积和重要的性能改进。

Entity Framework Core 简介

Entity Framework (EF) Core 是一种轻量级、可扩展和跨平台的流行实体框架数据访问技术。在 2016 年中被引入 .NET Core。

由于在 Microsoft 文档中已经提供了对 Entity Framework Core 的介绍，这里我们只提供针对这些信息的链接。

其他资源

- **Entity Framework Core**
<https://docs.microsoft.com/ef/core/>
- **使用 Visual Studio 的 ASP.NET Core 和 Entity Framework Core 入门**
<https://docs.microsoft.com/aspnet/core/data/ef-mvc/>
- **DbContext 类**
<https://docs.microsoft.com/ef/core/api/microsoft.entityframeworkcore.dbcontext>
- **EF Core 和 EF6.x 的对比**
<https://docs.microsoft.com/ef/efcore-and-ef6/index>

从 DDD 视角看 Entity Framework Core 的基础架构

从 DDD 的观点来看，EF 的一个重要能力是能够使用 POCO 风格的领域实体，在 EF 术语中称为 POCO 代码优先实体。如果您使用 POCO 领域实体，您的领域模型类就是持久化透明的，遵循[持久化透明](#)和[基础架构透明](#)的原则。

根据 DDD 模式，您应当在实体类中封装领域行为和规则，以便在访问任何集合的时候可以控制不变量、验证和规则。因此，在 DDD 中，允许公开访问子实体集合或者值对象不是好的做法。相反，您期望暴露一些方法来控制如何和何时可以更新字段和属性集合，以及在发生这些情况时，应该发生的行为和操作。

从 EF Core 1.1 开始，为了满足这些 DDD 需求，您可以在实体中使用纯字段，而不是公共的属性。如果您不希望实体的字段可以在外部访问，您可以只创建特性或者字段而不是属性。您也可以使用私有的属性设置器。

类似地，您现在可以通过使用类型为 `ICollection<T>` 的公共属性，对集合进行只读访问，集合中的私有字段（如 `List<>`）支持 EF 持久化。以前版本的 Entity Framework 版本需要集合属性支持 `ICollection<T>`，这意味着任何开发人员可以使用父实体类从其属性集合来添加或删除项目。这种可能性将违背 DDD 的推荐模式。

在公开只读 `ICollection` 对象时可以使用私有的集合，如下代码所示：

```
public class Order : Entity
{
    // Using private fields, allowed since EF Core 1.1
    private DateTime _orderDate;
    // Other fields ...

    private readonly List<OrderItem> _orderItems;
    public ICollection<OrderItem> OrderItems => _orderItems;

    protected Order() { }
    public Order(int buyerId, int paymentMethodId, Address address)
    {
        // Initializations ...
    }

    public void AddOrderItem(int productId, string productName,
                             decimal unitPrice, decimal discount,
                             string pictureUrl, int units = 1)
    {
        // Validation logic...

        var orderItem = new OrderItem(productId, productName,
                                         unitPrice, discount,
                                         pictureUrl, units);

        _orderItems.Add(orderItem);
    }
}
```

注意 `OrderItems` 属性只能使用 `ICollection<OrderItem>` 被只读访问。该类型是只读的，因此可以防止常见的外部更新。

EF Core 提供了一种将领域模型映射到物理数据库的方法，而不会污染领域模型。它是纯 .NET POCO 代码，因为映射操作实现在持久层。在该映射操作中，您需要配置字段到数据库的映射。在随后的 `OnModelCreating` 方法示例中，高亮代码告诉 EF Core 通过它的字段访问 `OrderItems` 属性。

```

// At OrderingContext.cs from eShopOnContainers
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    // ...
    modelBuilder.ApplyConfiguration(new OrderEntityTypeConfiguration());
    // Other entities' configuration ...
    // ...
}

// At OrderEntityTypeConfiguration.cs from eShopOnContainers
class OrderEntityTypeConfiguration : IEntityTypeConfiguration<Order>
{
    public void Configure(EntityTypeBuilder<Order> orderConfiguration)
    {
        orderConfiguration.ToTable("orders", OrderingContext.DEFAULT_SCHEMA);
        // Other configuration
        // ...

        var navigation =
            orderConfiguration.Metadata.FindNavigation(nameof(Order.OrderItems));

        //EF access the OrderItem collection property through its backing field
        navigation.SetPropertyAccessMode(PropertyAccessMode.Field);

        // Other configuration
        // ...
    }
}

```

当您使用字段而不是属性时，OrderItem 实体将保持不变，如同它具有一个 List<OrderItem> 属性一样。但是，它暴露了单个访问器 AddOrderItem() 方法，用于向订单添加新项。因此，行为和数据捆绑在一起，并且在所有使用该领域模型的应用代码中都保持一致。

使用 Entity Framework Core 实现自定义仓储

在实现层面，仓储只是一个在执行更新时，由工作单元（EF Core 中的 DbContext）协调的数据持久性代码的类。如下所示。

```
// using statements...
namespace Microsoft.eShopOnContainers.Services.Ordering.Infrastructure.Repositories
{
    public class BuyerRepository : IBuyerRepository
    {
        private readonly OrderingContext _context;
        public IUnitOfWork UnitOfWork
        {
            get
            {
                return _context;
            }
        }

        public BuyerRepository(OrderingContext context)
        {
            _context = context ?? throw new ArgumentNullException(nameof(context));
        }

        public Buyer Add(Buyer buyer)
        {
            return _context.Buyers
                .Add(buyer)
                .Entity;
        }

        public async Task<Buyer> FindAsync(string BuyerIdentityGuid)
        {
            var buyer = await _context.Buyers
                .Include(b => b.Payments)
                .Where(b => b.FullName == BuyerIdentityGuid)
                .SingleOrDefaultAsync();

            return buyer;
        }
    }
}
```

注意，作为规约，IBuyerRepository 接口来自领域模型层。但是，仓储实现是在持久性和基础架构层完成的。

EF DbContext 通过依赖注入提供给构造函数。它在同一 HTTP 请求范围内被多个仓储所共享，这得益于 IoC 容器（可以使用 services.AddDbContext<> 显示设置）中的默认生命周期（ServiceLifetime.Scoped）。

在仓储中实现的方法 (更新、事务与查询)

在每个仓储类中，都应当置入用来更新其相关聚合中所包含的实体状态的持久性方法。请记住在聚合和其相关的仓储之间存在一对一关系。考虑到聚合根实体对象可能可能在其 EF 关系图内有嵌入的子实体。例如，买方可能有作为相关子实体的多个付款方式。

由于 eShopOnContainers 中订单微服务的处理方式也基于 CQS/CQRS，大多数的查询都不是在自定义的仓储中实现的。开发人员可以自由地创建表示层所需要的查询和联接，而不受限于聚合、每个聚合的自定义仓储和 DDD。本指南建议的大多数自定义仓储都有一些更新或者事务方法，但只是查询用来更新的数据的方法。例如，BuyerRepository 仓储实现了一个 FindAsync 方法，因为应用需要在创建关联到新买家的订单之前知道特定的买家是否存在。

但是，如前所述，在基于灵活查询的 CQRS 查询中，使用基于 Dapper 的实现，真正实现了将获取的数据提供给展示层或客户端应用的查询方法。

对比自定义仓储与直接使用 EF DbContext

Entity Framework DbContext 类基于工作单元和仓储模式，可以直接被您的代码所使用，比如从 ASP.NET Core MVC 的控制器。这是您可以创建最简代码的方式，比如在 eShopOnContainers 中的 CRUD 目录微服务中。在需要最简代码的情况下，您可能希望直接使用 DbContext 类，就像许多开发人员所做的那样。

但是，在实现更为复杂的微服务或应用时，实现自定义的仓储提供了一些优势。工作单元和仓储模式用于封装基础架构持久层，以便与应用程序和领域模型层解耦。实现这些模式有助于模拟对数据库的访问。

在图 9-18 中，您可以看到不使用仓储（直接使用 EF DbContext）与使用仓储的区别，这使得模拟这些仓储更为容易。

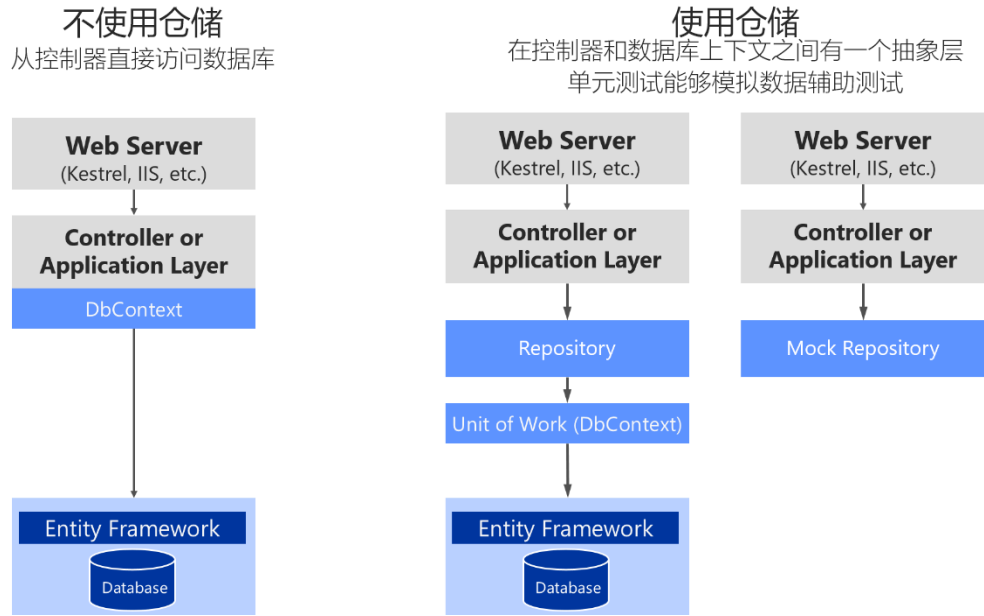


图9-18. 使用自定义仓储与使用纯 DbContext

模拟时有多种选择。您可以仅模拟仓储或者您可以模拟整个工作单元。通常仅模拟仓储就足够了，抽象并模拟整个工作单元通常是不必要的。

稍后，当我们关注应用层的时候，您将看到依赖注入是如何在 ASP.NET Core 中工作的，以及当使用仓储的时候，它是如何实现的。

简而言之，自定义的仓储使得在不受数据层状态影响的情况下，通过单元测试测试代码更为方便。如果您运行的测试也通过 Entity Framework 访问实际数据库，则它们不是单元测试，而是集成测试，这要慢很多。

如果直接使用 DbContext，则唯一的选择是使用具有预知数据的内存中的 SQL Server 来运行单元测试。您将无法以相同的方式在仓储级别控制模拟对象和仿真数据。当然，您可以随时测试 MVC 控制器。

IoC 容器中的 EF DbContext 和 IUnitOfWork 实例生命周期

DbContext 对象（以 IUnitOfWork 对象暴露）可能需要在同一 HTTP 请求范围内的多个仓储之间共享。例如，当正在执行的操作必须处理多个聚合就是这样，或者仅仅是因为您使用了多个仓储实例。必须指出的是接口 IUnitOfWork 是领域层的一部分，而不是 EF Core 类型。

为了做到这一点，DbContext 对象实例必须将其服务的生命周期设置为 ServiceLifetime.Scoped。在您的 ASP.NET Core Web API 项目中，从 Startup.cs 文件的 ConfigureServices 方法中使用 services.AddDbContext 在 IoC 容器中注册 DbContext 的时候，这是默认的生命周期。下面的代码说明了这一点。

```

public IServiceCollection ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc(options =>
    {
        options.Filters.Add(typeof(HttpGlobalExceptionHandler));
    }).AddControllersAsServices();
    services.AddEntityFrameworkSqlServer()
        .AddDbContext<OrderingContext>(options =>
        {
            options.UseSqlServer(Configuration["ConnectionString"],
                sqlOptions => sqlOptions.MigrationsAssembly(typeof(Startup).
                    GetTypeInfo().
                        Assembly.GetName().Name));
        },
        ServiceLifetime.Scoped // Note that Scoped is the default choice
                               // in AddDbContext. It is shown here only for
                               // pedagogic purposes.
    );
}

```

不应将 DbContext 实例化模式配置为 ServiceLifetime.Transient 或 ServiceLifetime.Singleton。

IoC 容器中仓储实例的生命周期

类似地，仓储的生命周期通常设置为作用域（Autofac 中的 InstancePerLifetimeScope）。也可能是瞬态的（Autofac 中的 InstancePerDependency），但是当使用作用域生命周期时，您的服务在内存方面更为有效。

```

// Registering a Repository in Autofac IoC container
builder.RegisterType<OrderRepository>()
    .As<IOrderRepository>()
    .InstancePerLifetimeScope();

```

请注意，如果将 DbContext 设置为作用域（InstancePerLifetimeScope）生命周期（DbContext 的默认生命周期），同时对仓储使用单例生命周期，可能会导致严重的并发问题。

其他资源

- **在 ASP.NET MVC 应用中实现仓储和工作单元模式**
<https://www.asp.net/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application>
- **Jonathan Allen. 使用 Entity Framework, Dapper 和 Chain 的仓储模式实现策略**
<https://www.infoq.com/articles/repository-implementation-strategies>
- **Cesar de la Torre. 对比 ASP.NET Core IoC 容器服务的生命周期和 Autofac IoC 容器实例范围**
<https://blogs.msdn.microsoft.com/cesardelatorre/2017/01/26/comparing-asp-net-core-ioc-service-lifetimes-and-autofac-ioc-instance-scopes/>

表映射

表映射标识查询得到的表数据和保存到数据库的表数据。前面您已经看到领域实体（例如，产品或订单领域）如何用于生成相关的数据库架构。EF 是围绕 *约定* 这个概念设计的。约定解决了类似“表名是什么？”或“主键是哪个属性？”的问题。约定通常基于约定俗成的命名 - 例如，作为主键的属性典型地以 `Id` 结尾。

按照约定，每个实体都将设置成映射到作为 `DbSet<TEntity>` 属性相同的表，其暴露了在派生上下文中的实体。如果没有为给定实体提供 `DbSet<TEntity>` 值，则使用类名。

数据标记与 Fluent API

有许多额外的 EF Core 约定，其中大多数可以通过使用 `OnModelCreating` 方法中的数据标记或 Fluent API 进行更改。

数据标记必须在实体模型类本身使用，从 DDD 角度这是更具侵入性的方式。这是因为您使用与基础架构数据库相关的数据标记来污染您的模型。另一方面，Fluent API 是修改数据持久化基础架构层中大多数约定和映射的一种方便方法，因为实体模型将是干净的，并且与持久化基础架构解耦。

Fluent API 与 OnModelCreating 方法

如上所述，为了更改约定和映射，您可以在 `DbContext` 类中使用 `OnModelCreating` 方法。

在需要的时候，`eShopOnContainers` 中的 `ordering` 微服务实现了显式的映射和配置，如下代码所示。

```

// At OrderingContext.cs from eShopOnContainers
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    // Other entities' configuration ...
    modelBuilder.ApplyConfiguration(new OrderEntityTypeConfiguration());
}

// At OrderEntityTypeConfiguration.cs from eShopOnContainers
class OrderEntityTypeConfiguration : IEntityTypeConfiguration<Order>
{
    public void Configure(EntityTypeBuilder<Order> orderConfiguration)
    {
        orderConfiguration.ToTable("orders", OrderingContext.DEFAULT_SCHEMA);
        orderConfiguration.HasKey(o => o.Id);
        orderConfiguration.Ignore(b => b.DomainEvents);

        orderConfiguration.Property(o => o.Id)
            .ForSqlServerUseSequenceHiLo("orderseq",
                OrderingContext.DEFAULT_SCHEMA);
        //Address Value Object persisted as owned entity supported since EF Core 2.0
        orderConfiguration.OwnsOne(o => o.Address);
        orderConfiguration.Property<DateTime>("OrderDate").IsRequired();
        orderConfiguration.Property<int?>("BuyerId").IsRequired(false);
        orderConfiguration.Property<int>("OrderStatusId").IsRequired();
        orderConfiguration.Property<int?>("PaymentMethodId").IsRequired(false);
        orderConfiguration.Property<string>("Description").IsRequired(false);

        var navigation =
            orderConfiguration.Metadata.FindNavigation(nameof(Order.OrderItems));
        navigation.SetPropertyAccessMode(PropertyAccessMode.Field);

        orderConfiguration.HasOne<PaymentMethod>()
            .WithMany()
            .HasForeignKey("PaymentMethodId")
            .IsRequired(false)
            .OnDelete(DeleteBehavior.Restrict);
        orderConfiguration.HasOne<Buyer>()
            .WithMany()
            .IsRequired(false)
            .HasForeignKey("BuyerId");
        orderConfiguration.HasOne(o => o.OrderStatus)
            .WithMany()
            .HasForeignKey("OrderStatusId");
    }
}

```


您可以在同一个 `OnModelCreating` 方法中设置所有的 Fluent API 映射，但是最好对代码进行分区，并且有多个配置类，每个实体一个，如示例所示。尤其是对于特别大的模型，可以有单独的配置类来配置不同的实体类型。

示例中的代码展示了一些显式定义和映射。但是，EF Core 约定能自动处理许多映射，因此在您的案例中，实际需要您编写的实际代码要更少。

EF Core 中的 Hi/Lo 算法

前面代码中有趣的方面是，它使用了 [Hi/Lo 算法](#) 作为关键字生成策略。

Hi/Lo 算法在需要唯一键时很有用。简而言之，Hi/Lo 算法可以为表中的行分配唯一标识符，而不需要基于立即将行存储到数据库中。这使得您可以立即开始使用标识符，就像常规的数据库序列 ID 一样。

Hi/Lo 算法描述了在客户端而不是在数据库中生成安全 ID 的机制。在这里安全意味着没有碰撞。由于以下原因，此算法很有趣：

- 它不破坏工作单元模式
- 它不需要其它 DBMS 中序列生成器方式的往返
- 与使用 GUID 技术不同，它生成一个人可读的标识符

EF Core 使用 `ForSqlServerUseSequenceHiLo` 方法支持 [Hi/Lo](#) 算法，如前面的例子所示。

映射字段而不是属性

使用从 EF Core 1.1 开始的的特性，您可以直接将数据库列映射到字段，可以不使用实体类中的任何属性，而只是将表中的列映射到字段。对于不需要从实体外部访问的任何内部状态，通常使用私有字段。

您可以对单个字段或者集合，例如 `List<>` 字段执行此操作。当前面我们讨论对领域模型类建模的时候提到这一点，但是在这里，您可以看到如何使用 `PropertyAccessMode.Field` 配置前面高亮的代码。

在 EF Core 中使用影子属性，在基础架构层面隐藏 ID

在 EF Core 中，影子属性是在您的实体类模型中不存在的属性。这些属性的值或状态仅在基础架构级别的 [ChangeTracker](#) 类中维护。

实现规格模式

正如在前面设计部分介绍的那样，规格模式，全称是 (Query-specification pattern)，是一种领域驱动设计模式，被设计为用来表示对带有可选的排序和分页逻辑定义的查询。

规格模式通过一个对象来定义查询。例如，为了封装搜索某些产品的分页查询，您可以创建一个带有必要输入参数 (pageNumber, pageSize, filter 等) 的 PagedProduct 规格。然后，在任何 Repository 方法 (通常是一个 List()重载) 中，接受一个 ISpecification 对象，并根据该规格运行预期的查询。

下面通用规格的示例代码来自 [eShopOnweb](#):

```
// GENERIC SPECIFICATION INTERFACE
// https://github.com/dotnet-architecture/eShopOnWeb

public interface ISpecification<T>
{
    Expression<Func<T, bool>> Criteria { get; }
    List<Expression<Func<T, object>>> Includes { get; }
    List<string> IncludeStrings { get; }
}
```

然后，通用的规格基类如下实现。

```
// GENERIC SPECIFICATION IMPLEMENTATION (BASE CLASS)
// https://github.com/dotnet-architecture/eShopOnWeb

public abstract class BaseSpecification<T> : ISpecification<T>
{
    public BaseSpecification(Expression<Func<T, bool>> criteria)
    {
        Criteria = criteria;
    }
    public Expression<Func<T, bool>> Criteria { get; }

    public List<Expression<Func<T, object>>> Includes { get; } =
        new List<Expression<Func<T, object>>>();

    public List<string> IncludeStrings { get; } = new List<string>();

    protected virtual void AddInclude(Expression<Func<T, object>> includeExpression)
    {
        Includes.Add(includeExpression);
    }

    // string-based includes allow for including children of children
    // e.g. Basket.Items.Product
    protected virtual void AddInclude(string includeString)
    {
        IncludeStrings.Add(includeString);
    }
}
```

下面的规格或者使用 basket 的 ID，或者使用 basket 所属的买家的 ID 加载单个的 basket 实体。它将提前加载 basket 的项目集合。

```
// SAMPLE QUERY SPECIFICATION IMPLEMENTATION

public class BasketWithItemsSpecification : BaseSpecification<Basket>
{
    public BasketWithItemsSpecification(int basketId)
        : base(b => b.Id == basketId)
    {
        AddInclude(b => b.Items);
    }
    public BasketWithItemsSpecification(string buyerId)
        : base(b => b.BuyerId == buyerId)
    {
        AddInclude(b => b.Items);
    }
}
```

最后，您可以看到泛型的 EF Repository 如何使用这些规格对给定的实体类型 T 进行过滤和提前加载相关的数据。

```
// GENERIC EF REPOSITORY WITH SPECIFICATION
// https://github.com/dotnet-architecture/eShopOnWeb

public IEnumerable<T> List(ISpecification<T> spec)
{
    // fetch a Queryable that includes all expression-based includes
    var queryableResultWithIncludes = spec.Includes
        .Aggregate(_dbContext.Set<T>().AsQueryable(),
            (current, include) => current.Include(include));

    // modify the IQueryable to include any string-based include statements
    var secondaryResult = spec.IncludeStrings
        .Aggregate(queryableResultWithIncludes,
            (current, include) => current.Include(include));

    // return the result of the query using the specification's criteria expression
    return secondaryResult
        .Where(spec.Criteria)
        .AsEnumerable();
}
```

除了封装过滤逻辑，规格还可以指定返回的数据的形状，包括要填充哪些属性。

尽管不建议从仓储库中返回 IQueryable，但在仓储库中使用它们来构建一组结果是完全正确的。您可以在上面的 List 方法中看到这种方式，该方法中使用中间的 IQueryable 表达式在最后一行执行具有规格条件的查询之前，构建查询的包含列表。

其他资源

- **表映射**
<https://docs.microsoft.com/ef/core/modeling/relational/tables>
- **使用 HiLo 和 Entity Framework Core 生成键**
<http://www.talkingdotnet.com/use-hilo-to-generate-keys-with-entity-framework-core/>
- **后台字段**
<https://docs.microsoft.com/ef/core/modeling/backing-field>
- **Steve Smith. 在 Entity Framework Core 中封装集合**
<http://ardalis.com/encapsulated-collections-in-entity-framework-core>
- **影子属性**
<https://docs.microsoft.com/ef/core/modeling/shadow-properties>
- **规格模式**
<http://deviq.com/specification-pattern/>

使用 NoSQL 数据库作为持久基础架构

当您使用 NoSQL 数据库作为数据层基础架构，显然您不会使用类似 Entity Framework Core 的 ORM。而是使用由 NoSQL 引擎提供的 API，比如 Azure Cosmos DB、MongoDB、Cassandra、RavenDB、CouchDB 或者 Azure Storage Tables。

但是，当您使用 NoSQL 数据库时，尤其是文档导向的数据库，比如 Azure Cosmos DB、MongoDB、Cassandra 或 RavenDB 时，使用 DDD 聚合方式设计您的模型与使用 EF Core 所做的部分类似，识别聚合根、子实体类和值对象类。但是，最终，数据库的选择将影响您的设计。

在使用文档导向的数据库时，可以将聚合实现为单个文档，序列化为 JSON 或者其它格式。但是，从领域模型代码的角度来看，数据库的使用是透明的。当使用 NoSQL 数据库时，您仍然使用实体类和聚合根类，但是比使用 EF Core 更为灵活，因为持久化是非关系的。

区别在于您如何持久化模型。如果您基于 POCO 实体类实现您的领域模型，基础架构持久化对此透明，那么您可能觉得可以转移到不同的持久性基础架构，甚至是从关系型到 NoSQL。然而，这并不应该是您的目标，不同的数据库总有限制将您挡回来，因此您不能对关系型数据库或 NoSQL 数据库拥有相同的模型。改变持久模型不会很简单，因为事务和持久化操作将非常不同。

例如，在文档导向的数据库中，聚合根拥有多个子集合属性是可以的。在关系数据库中，查询多个子集合属性则非常糟糕，因为您将从 EF 得到使用 UNION ALL 这样的 SQL 语句。对关系型数据库和 NoSQL 数据库拥有同样的领域模型并不简单，您不要试图尝试它。您确实需要基于不同的数据库是如何使用数据的理解来设计您的领域模型。

使用 NoSQL 数据库的一个好处是实体更为非标准化，所以您不必设置表映射。领域模型比使用关系数据库更为灵活。

在基于聚合设计领域模型时，转移到使用 NoSQL 和文档导向数据库可能比使用关系数据库更容易，因为您设计的聚合根更类似在文档导向数据库中序列化的文档。然后，您可以包含这些“包”中包含您可能需要的所有信息。

例如，在使用文档导向的数据库时，下面的 JSON 代码是订单聚合的示例实现。它类似于我们在 eShopOnContainers 示例中实现的订单集，但其中没有使用 EF Core。

```

{
  "id": "2017001",
  "orderDate": "2/25/2017",
  "buyerId": "1234567",
  "address": [
    {
      "street": "100 One Microsoft Way",
      "city": "Redmond",
      "state": "WA",
      "zip": "98052",
      "country": "U.S."
    }
  ],
  "orderItems": [
    {"id": 20170011, "productId": "123456", "productName": ".NET T-Shirt",
      "unitPrice": 25, "units": 2, "discount": 0},
    {"id": 20170012, "productId": "123457", "productName": ".NET Mug",
      "unitPrice": 15, "units": 1, "discount": 0}
  ]
}

```

Azure Cosmos DB 和原生的 Cosmos DB API 简介

[Azure Cosmos DB](#) 是 Microsoft 针对关键任务型应用程序的全球分布式多模式数据库服务。 Azure Cosmos DB 提供遍布全球[开箱即用的部署](#)，全球范围[吞吐量和存储的弹性扩展](#)，第 99 百分位的单位数毫秒的等待时间，[5 个明确定义的一致性水平](#)以及保证的高可用性，所有这些都由[行业领先的 SLA](#) 提供支持。 Azure Cosmos 数据库自动索引数据，无需处理模式和索引管理。它是多模型，支持文档、键值、图形和列式数据模型。

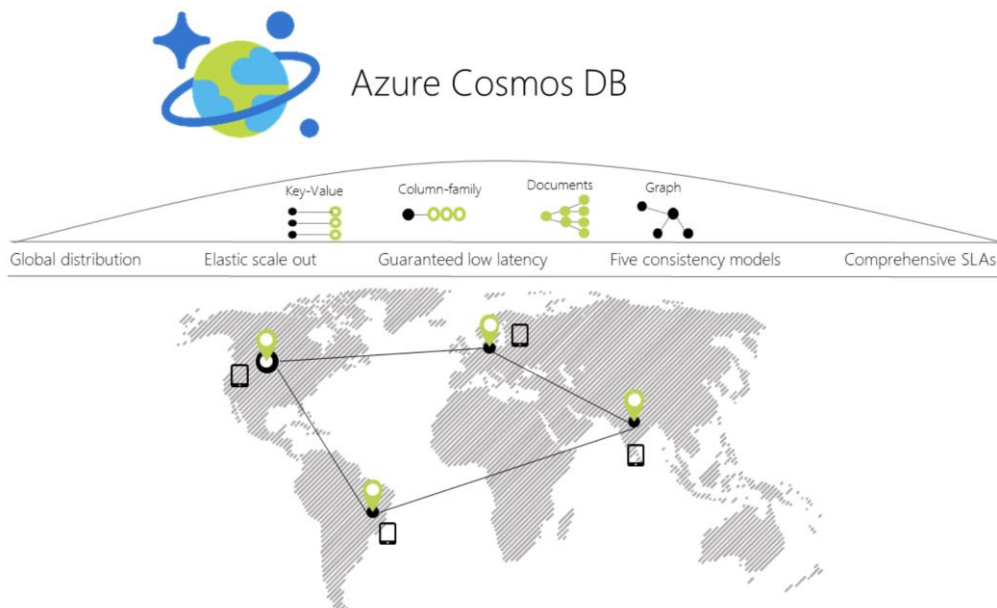


图9-19. Azure Cosmos DB 全球分布

在您使用 C# 模型实现该聚合以用于类似 Azure Cosmos DB API 时，该聚合类似于 EF Core 中使用的 C# POCO 类。区别在于从应用程序和基础架构层中使用它们的方式，如下代码所示：

```
// C# EXAMPLE OF AN ORDER AGGREGATE BEING PERSISTED WITH AZURE COSMOS DB API
// *** Domain Model Code ***
// Aggregate: Create an Order object with its child entities and/or value objects.
// Then, use AggregateRoot's methods to add the nested objects & apply invariants.
Order orderAggregate = new Order
{
    Id = "2017001",
    OrderDate = new DateTime(2005, 7, 1),
    BuyerId = "1234567",
    PurchaseOrderNumber = "PO18009186470"
}
Address address = new Address
{
    Street = "100 One Microsoft Way",
    City = "Redmond",
    State = "WA",
    Zip = "98052",
    Country = "U.S."
}

orderAggregate.UpdateAddress(address);

OrderItem orderItem1 = new OrderItem
{
    Id = 20170011,
    ProductId = "123456",
    ProductName = ".NET T-Shirt",
    UnitPrice = 25,
    Units = 2,
    Discount = 0;
};

//Using methods with domain logic within the entity. No anemic-domain model
orderAggregate.AddOrderItem(orderItem1);
// *** End of Domain Model Code ***

// *** Infrastructure Code using Cosmos DB Client API ***
Uri collectionUri = UriFactory.CreateDocumentCollectionUri(databaseName,
                                                         collectionName);
await client.CreateDocumentAsync(collectionUri, order);

// As your app evolves, let's say your object has a new schema. You can insert
// OrderV2 objects without any changes to the database tier.
Order2 newOrder = GetOrderV2Sample("IdForSalesOrder2");
await client.CreateDocumentAsync(collectionUri, newOrder);
```

您可以看到，您使用领域模型的方式，与基础架构为 EF 时在领域模型层中使用领域模型的方式是类似的。您仍然使用同样的聚合根方法来确保聚合中的一致性、不变量和验证。

但是，当您将模型持久化到 NoSQL 数据库中时，与 EF Core 或与关系数据库相关的任何其它代码相比，代码和 API 都发生了显著变化。

针对 MongoDB 和 Azure Cosmos DB 实现 .NET 代码

在 .NET 容器中使用 Azure Cosmos DB

您可以从运行在容器中的 .NET 代码访问 Azure Cosmos DB 数据库，就像从任何其他 .NET 应用程序一样。例如，eShopOnContainers 中的 Locations.API 和 Marketing.API 微服务实现，所以它们可以使用 Azure Cosmos DB 数据库。

但是，从 Docker 开发环境的角度来看，Azure Cosmos DB 存在一些限制，即使在本地的 [Azure Cosmos DB 仿真器](#) 能够在本地开发机器（如 PC）中运行的情况下。截至 2017 年底，它只支持 Windows，而不支持 Linux。

在 Docker 上运行这个模拟器也是可能的，但是在 Windows 容器上运行的却不是 Linux 容器。如果您的应用程序部署为 Linux 容器，那么这对开发环境来说是一个初期的障碍，因为目前您不能同时在 Docker for Windows 上部署 Linux 和 Windows 容器。所有正在部署的容器都必须用于 Linux 或 Windows。

开发/测试解决方案的理想和更直接的部署是能够将数据库系统作为容器与自定义容器一起部署，因此您的开发/测试环境始终保持一致。

为本地开发/测试 Linux/Windows 容器和 Azure Cosmos DB 使用 MongoDB API

Cosmos DB 数据库支持 MongoDB API for .NET 以及本地 MongoDB 有线协议。这意味着通过使用现有的驱动程序，为 MongoDB 编写的应用程序现在可以与 Cosmos DB 进行通信，并使用 Cosmos DB 数据库而不是 MongoDB 数据库，如图 9-20 所示。

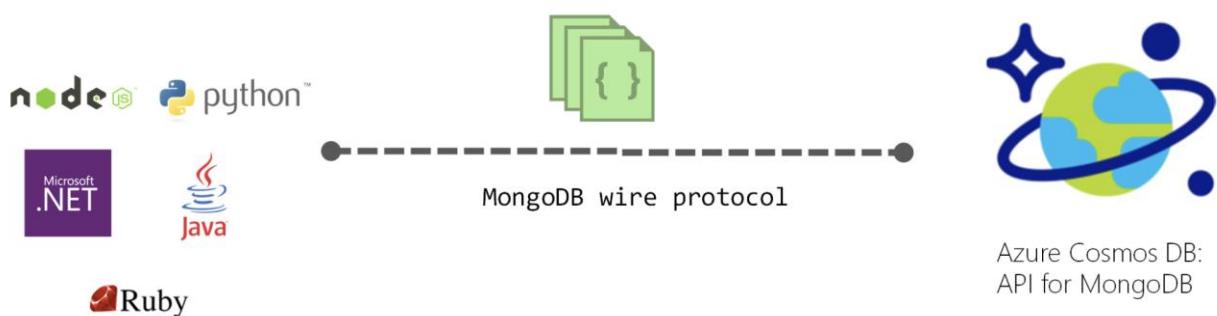


图9-20. 使用 MongoDB API 和协议访问 Azure Cosmos DB

这是一个非常方便的方法来验证 Docker 环境下的 Linux 容器的概念，因为 [MongoDB Docker 镜像](#) 是支持 Docker Linux 容器和 Docker Windows 容器的多拱形镜像。

如图 9-21 所示，通过使用 MongoDB API，eShopOnContainers 支持用于本地开发环境的 MongoDB Linux 和 Windows 容器，但您可以通过简单地[更改 MongoDB 连接字符串指向 Azure Cosmos DB](#) 来转移到可扩展 PaaS 云解决方案如 Azure Cosmos DB。

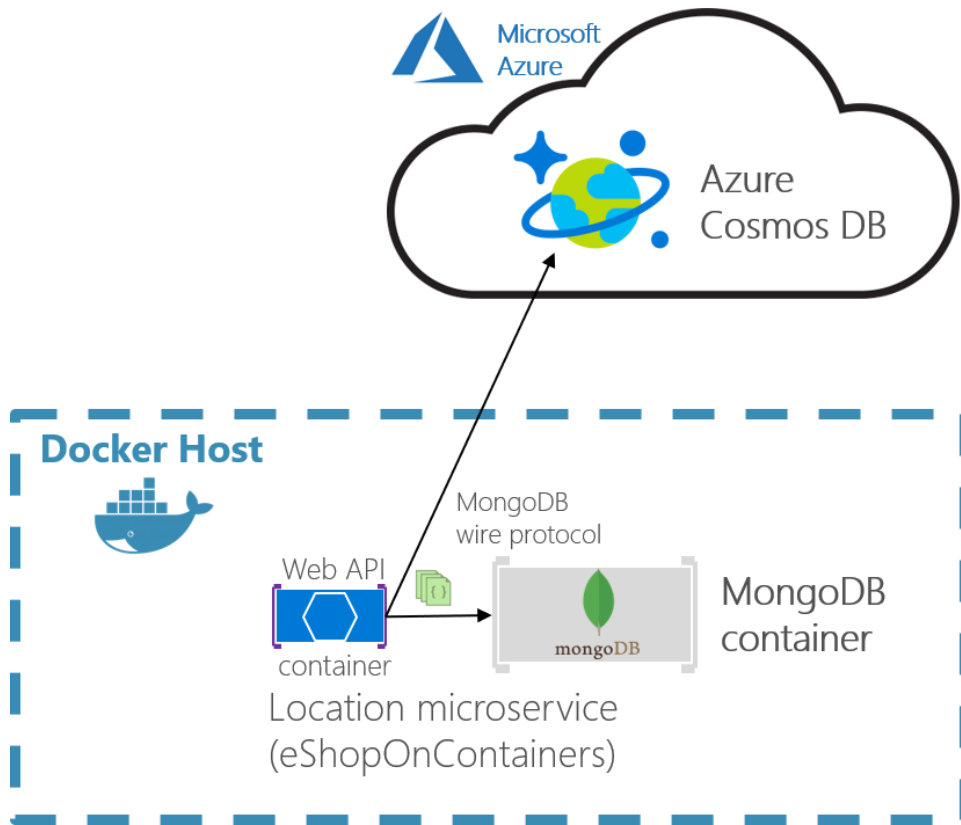


图9-21. eShopOnContainers 使用 MongoDB 容器用于开发环境或者产品的 Azure Cosmos DB 环境

作为 PaaS 和可扩展服务，生产环境的 Azure Cosmos DB 加载运行在 Azure 云中。

您的自定义 .NET Core 容器可以在本地开发 Docker 主机（即在 Windows 10 计算机上使用 Docker for Windows）上运行，也可以部署到生产环境中，如 Azure AKS 或 Azure Service Fabric 中的 Kubernetes。在第二个环境中，您将仅部署 .NET Core 定制容器，但不部署 MongoDB 容器，因为您将使用云中的 Azure Cosmos DB 来处理生产中的数据。

如果使用 MongoDB API，一个明显的好处是你的解决方案可以在 MongoDB 或 Azure Cosmos DB 这两个数据库引擎上运行，所以迁移到不同的环境应该很容易。但是，有时候为了充分利用特定数据库引擎的功能，使用本地 API（即本地的 Cosmos DB API）是值得的。

要进一步比较在云中简单使用 MongoDB 与 Cosmos DB，请参阅此页面：[使用 Azure Cosmos DB 的优势](#)。

分析生产应用程序的方式: MongoDB API 与 Cosmos DB API

在 eShopOnContainers 中, 我们使用的是 MongoDB API, 因为我们的优先级基本上是使用 NoSQL 数据库的一致的发展/测试环境, 这个数据库也可以用于 Azure Cosmos 数据库。

但是, 如果您计划使用 MongoDB API 访问 Azure 中的 Azure Cosmos DB 用于生产应用程序, 则应首先分析使用 MongoDB API 与使用本机 Azure Cosmos DB API 访问 Azure Cosmos DB 数据库时的功能和性能差异。根据具体情况可能会有所不同, 所以您可以继续使用 MongoDB API, 并且同时支持两个 NoSQL 数据库引擎, 或者, 如果 Azure Cosmos DB 中有任何特殊的功能, 与使用 MongoDB API 时执行的方式不同, 则可能会有所不同。

相反, 您也可以使用 MongoDB 集群作为 Azure 云中的生产数据库, 也可以使用 [MongoDB Azure 服务](#)。但这不是 Microsoft 提供的 PaaS 服务。在这种情况下, Azure 只是托管来自 MongoDB 的解决方案。

基本上, 这只是一个免责声明, 说明您不应该总是使用 MongoDB API 来针对 Azure Cosmos DB, 就像我们在 eShopOnContainers 中使用的那样, 因为这是 Linux 容器的一个方便的选择。决定应该基于您需要为您的生产应用程序执行的特定需求和测试。

代码: 在 .NET Core 应用中使用 MongoDB API

MongoDB API for .NET 基于 NuGet 包, 您需要添加到您的项目中, 如来自 eShopOnContainers 中 Locations.API 的图 9-22 所示。

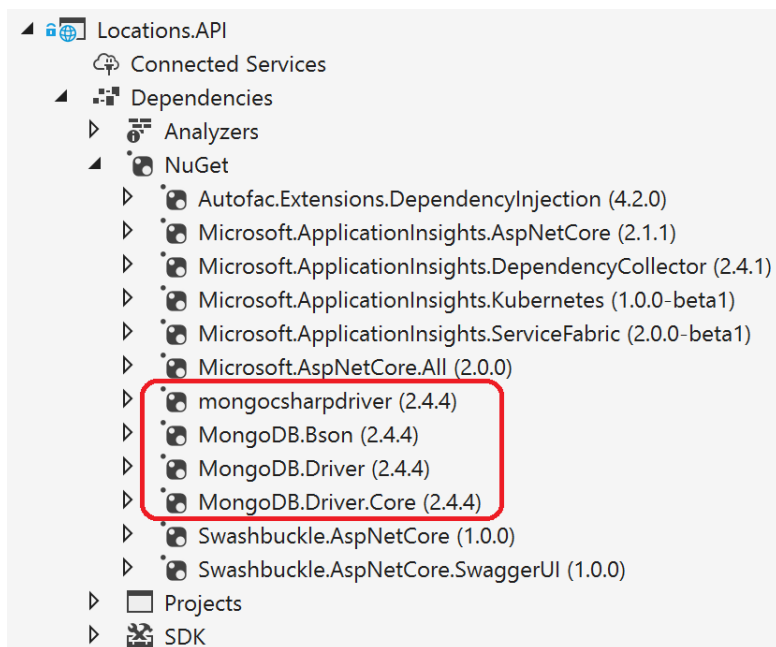


图 9-22. 在 .NET Core 项目中的 MongoDB API NuGet 包引用

关于代码, 有几个地方需要考查, 如下面的部分所述。

MongoDB API 使用的模型

首先，您需要定义一个模型，用于在应用程序的内存空间中保存来自数据库的数据。以下是 eShopOnContainers 中用于位置的模型示例。

```
using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;
using MongoDB.Driver.GeoJsonObjectModel;
using System.Collections.Generic;

public class Locations
{
    [BsonId]
    [BsonRepresentation(BsonType.ObjectId)]
    public string Id { get; set; }
    public int LocationId { get; set; }
    public string Code { get; set; }
    [BsonRepresentation(BsonType.ObjectId)]
    public string Parent_Id { get; set; }
    public string Description { get; set; }
    public double Latitude { get; set; }
    public double Longitude { get; set; }
    public GeoJsonPoint<GeoJson2DGeographicCoordinates> Location
        { get; private set; }
    public GeoJsonPolygon<GeoJson2DGeographicCoordinates> Polygon
        { get; private set; }
    public void SetLocation(double lon, double lat) => SetPosition(lon, lat);
    public void SetArea(List<GeoJson2DGeographicCoordinates> coordinatesList)
        => SetPolygon(coordinatesList);

    private void SetPosition(double lon, double lat)
    {
        Latitude = lat;
        Longitude = lon;
        Location = new GeoJsonPoint<GeoJson2DGeographicCoordinates>(
            new GeoJson2DGeographicCoordinates(lon, lat));
    }

    private void SetPolygon(List<GeoJson2DGeographicCoordinates> coordinatesList)
    {
        Polygon = new GeoJsonPolygon<GeoJson2DGeographicCoordinates>(
            new GeoJsonPolygonCoordinates<GeoJson2DGeographicCoordinates>(
                new GeoJsonLinearRingCoordinates<GeoJson2DGeographicCoordinates>(
                    coordinatesList)));
    }
}
```

你可以看到来自 MongoDB NuGet 包的一些属性和类型。

NoSQL 数据库通常非常适合处理非关系但是分层的数据。在这个例子中，我们使用特别为地理位置而设计的 MongoDB 类型，比如 **GeoJson2DGeographicCoordinates**。

检索数据库和集合

在 eShopOnContainers 中，我们创建了一个自定义的数据库上下文，在这里我们实现代码来检索数据库和 MongoCollections，如下面的代码所示。

```
public class LocationsContext
{
    private readonly IMongoDatabase _database = null;

    public LocationsContext(IOptions<LocationSettings> settings)
    {
        var client = new MongoClient(settings.Value.ConnectionString);
        if (client != null)
            _database = client.GetDatabase(settings.Value.Database);
    }
    public IMongoCollection<Locations> Locations
    {
        get
        {
            return _database.GetCollection<Locations>("Locations");
        }
    }
}
```

检索数据

然后，从任何 C# 代码（如 Web API 控制器或自定义仓储库）中，通过 MongoDB API 进行查询时，可以编写类似的代码，_context 对象是以前 LocationsContext 类的一个实例。

```
public async Task<Locations> GetAsync(int locationId)
{
    var filter = Builders<Locations>.Filter.Eq("LocationId", locationId);
    return await _context.Locations
        .Find(filter)
        .FirstOrDefaultAsync();
}
```

在 docker-compose.override.yml 文件中使用环境变量作为 MongoDB 连接串

在创建 MongoClient 对象时，所需的参数恰好是 ConnectionString 指向 eShopOnContainers 的情况，既可以指向本地 MongoDB Docker 容器，也可以指向“生产” Azure Cosmos DB 数据库。该连接字符串来自 docker-compose 或 Visual Studio 部署时在 docker-compose.override.yml 文件中定义的 environment 变量，如下面的 yml 代码所示。

```
# docker-compose.override.yml
version: '3'
services:
  # Other services
  locations.api:
```

```
environment:
  # Other settings
  - ConnectionString=${ESHOP_AZURE_COSMOSDB:-mongodb://nosql.data}
```

ConnectionString 环境变量是可定义的，因此如果在带有 Azure Cosmos DB 连接字符串的 .env 文件中定义了 ESHOP_AZURE_COSMOSDB 全局变量，它将访问云中的 Azure Cosmos DB 数据库。

在这里，您可以看到带有 Azure Cosmos DB 连接字符串 global 环境变量的 .env 文件，在 eShopOnContainers 中用作便捷的方法。

```
# .env file, in eShopOnContainers root folder
# Other Docker environment variables
ESHOP_EXTERNAL_DNS_NAME_OR_IP=localhost
ESHOP_PROD_EXTERNAL_DNS_NAME_OR_IP=<YourDockerHostIP>

#ESHOP_AZURE_COSMOSDB=<YourAzureCosmosDBConnData>

#Other environment variables for additional Azure infrastructure assets
#ESHOP_AZURE_REDIS_BASKET_DB=<YourAzureRedisBasketInfo>
#ESHOP_AZURE_SERVICE_BUS=<YourAzureServiceBusInfo>
```

您应取消注释 ESHOP_AZURE_COSMOSDB 行，并使用从 Azure portal 获取的真实 Azure Cosmos DB 连接字符串进行更新，如该教程所述：[将 MongoDB 应用程序连接到 Azure Cosmos DB](#)。

如果 ESHOP_AZURE_COSMOSDB 全局变量为空，即在 .env 文件中注释它，那么它将使用默认的 MongoDB 连接字符串，该连接字符串指向在 eShopOnContainers 中部署的本地 MongoDB 容器，该容器名为 nosql.data，在 docker-compose 文件中，如下面的 .yml 代码所示。

```
# docker-compose.yml
version: '3'
services:
  # ...Other services...
  nosql.data:
    image: mongo
```

其他资源

- **为 NoSQL 数据库的文档数据建模**
<https://docs.microsoft.com/en-us/azure/cosmos-db/modeling-data>
- **Vaughn Vernon. 理想的领域驱动设计聚合存储?**
<https://vaughnvernon.co/?p=942>
- **Azure Cosmos DB 介绍: 面向 MongoDB 的 API**
<https://docs.microsoft.com/en-us/azure/cosmos-db/mongodb-introduction>
- **Azure Cosmos DB: 使用 .NET 和 Azure 门户构建 MongoDB API 的 Web 应用**
<https://docs.microsoft.com/en-us/azure/cosmos-db/create-mongodb-dotnet>
- **在本地开发和测试时使用 Azure Cosmos DB 仿真器**
<https://docs.microsoft.com/en-us/azure/cosmos-db/local-emulator>
- **将 MongoDB 应用连接到 Azure Cosmos DB**
<https://docs.microsoft.com/en-us/azure/cosmos-db/connect-mongodb-account>

- **Cosmos DB 模拟器的 Docker 影像 (Windows 容器)**
<https://hub.docker.com/r/microsoft/azure-cosmosdb-emulator/>
- **MongoDB Docker 镜像 (Linux 和 Windows 容器)**
https://hub.docker.com/r/_/mongo/
- **用 MongoChef (Studio 3T) 访问 Azure Cosmos DB: MongoDB 账户 API**
<https://docs.microsoft.com/en-us/azure/cosmos-db/mongodb-mongochef>

设计微服务应用层和 Web API

使用 SOLID 原则和依赖注入

SOLID 原则是用于任何现代和关键任务应用程序的关键技术。如开发使用 DDD 模式的微服务。SOLID 是五个基础原则的缩写：

- 单一职责原则 (Single Responsibility principle)
- 开闭原则 (Open/closed principle)
- 李氏替换原则 (Liskov substitution principle)
- 接口分离原则 (Inversion Segregation principle)
- 依赖倒置原则 (Dependency Inversion principle)

SOLID 更多地是关于如何设计应用程序或微服务的内部层以及解耦它们之间的依赖关系。它与领域无关，而是与应用程序的技术设计有关。最后的依赖倒置原则，允许您将基础架构层与其它层解耦，这样可以更好地解耦 DDD 中各层的实现。

DI 是实现依赖倒置原则的一种方式。它是实现在对象和其依赖之间松耦合，而不是直接实例化它们，或者使用静态引用的技术。类所需要的用于执行其操作的对象被提供（或“注入”）给类。通常，类通过构造函数声明其依赖，从而类遵循显式依赖原则。DI 通常基于特定的控制反转（IoC）容器。

ASP.NET Core 提供了一个简单的内置 IoC 容器，但是您可以使用您偏爱的 IoC 容器，如 Autofac 或 Ninject。

通过遵循 SOLID 原则，您的类自然会变得更小、分解良好和易于测试。但是，如何才能知道是否有太多的依赖项被注入到您的类中呢？如果通过构造函数使用 DI，那么只需要查看构造函数的参数个数就可以很容易地检测这一点。如果存在太多的依赖，这通常是一个信号（[代码味道](#)），您的类试图做得太多了，还可能违背单一职责原则。

可能需要另一本指南来涵盖 SOLID 原则的细节，因此，本指南只要求您对这些主题有最基本的了解。

其他资源

- **SOLID: 基础 OOP 原则**
<http://deviq.com/solid/>
- **控制反转容器和依赖注入模式**
<https://martinfowler.com/articles/injection.html>
- **Steve Smith. New 是一种胶水**
<http://ardalis.com/new-is-glue>

使用 Web API 实现微服务应用层

使用依赖注入将基础架构对象注入到应用层

如前所述，应用层可以作为正在构建的产品的一部分来实现。如在 Web API 项目或 MVC Web 应用项目中。在使用 ASP.NET Core 构建的微服务中，应用层通常是您的 Web API 库。如果您希望将来自 ASP.NET Core（其基础架构和控制器）的部分从您的自定义应用层代码中分离出来，还可以将应用层放在单独的类库中，但这是可选的。

例如，订单微服务的应用层代码直接实现为 Ordering.API 项目的一部分（ASP.NET Core Web API 项目），如图 9-23 所示。

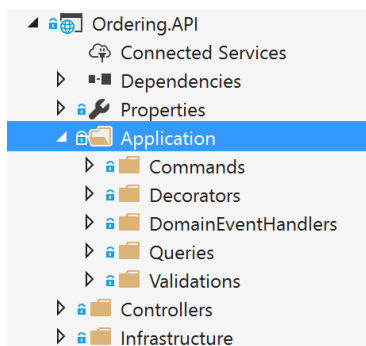


图9-23. *Ordering.API ASP.NET Core Web API 项目的应用层*

ASP.NET Core 包括一个简单的[内置 IoC 容器](#)（通过 `IServiceProvider` 接口表示），默认支持构造函数注入。ASP.NET 通过 DI 提供某些服务。ASP.NET Core 使用术语 *服务* 表示您注册的将被通过 DI 注入的类型。在应用程序的 `Startup` 类的 `ConfigureServices` 方法中配置内置容器的服务。您的依赖项实现为服务所需的类型。

通常，您希望注入实现基础架构对象的依赖项。典型的依赖注入是仓储。但是您可以注入任何您拥有的其它基础架构依赖项。对于更简单的实现，您可以直接注入工作单元模式对象（EF Context 对象），因为 `DbContext` 同样实现为基础架构持久对象。

在下面的示例中，您可以看到 .NET Core 如何通过构造函数注入仓储对象。该类是一个命令处理器，我们将在下一节中进行介绍。


```

// Sample command handler
public class CreateOrderCommandHandler
    : IAsyncRequestHandler<CreateOrderCommand, bool>
{
    private readonly IOrderRepository _orderRepository;
    private readonly IIdentityService _identityService;
    private readonly IMediator _mediator;

    // Using DI to inject infrastructure persistence Repositories
    public CreateOrderCommandHandler(IMediator mediator,
                                     IOrderRepository orderRepository,
                                     IIdentityService identityService)
    {
        _orderRepository = orderRepository ??
            throw new ArgumentNullException(nameof(orderRepository));
        _identityService = identityService ??
            throw new ArgumentNullException(nameof(identityService));
        _mediator = mediator ??
            throw new ArgumentNullException(nameof(mediator));
    }

    public async Task<bool> Handle(CreateOrderCommand message)
    {
        // Create the Order AggregateRoot
        // Add child entities and value objects through the Order aggregate root
        // methods and constructor so validations, invariants, and business logic
        // make sure that consistency is preserved across the whole aggregate
        var address = new Address(message.Street, message.City, message.State,
                                  message.Country, message.ZipCode);
        var order = new Order(message.UserId, address, message.CardTypeId,
                              message.CardNumber, message.CardSecurityNumber,
                              message.CardHolderName, message.CardExpiration);

        foreach (var item in message.OrderItems)
        {
            order.AddOrderItem(item.ProductId, item.ProductName, item.UnitPrice,
                               item.Discount, item.PictureUrl, item.Units);
        }

        _orderRepository.Add(order);

        return await _orderRepository.UnitOfWork
            .SaveEntitiesAsync();
    }
}

```

该类使用注入的仓储来执行事务并持久化状态变更。该类是否是命令处理器，ASP.NET Core 的 Web API 控制器方法，还是 [DDD 应用服务](#) 方法都无关紧要。终归是一个简单的类，使用类似于命令处理器的仓储、领域实体和其它应用程序配合方法。依赖注入对所有提到的类使用相同的工作方式，如在使用基于构造函数的 DI 示例中所示。

注册依赖实现类型，接口或抽象

在使用通过构造函数注入的对象之前，您需要知道在何处注册接口和生产通过 DI 注入到应用程序类中的对象的类（如前所述，DI 基于构造函数）。

使用 ASP.NET Core 内置的 IoC 容器

当您使用由 ASP.NET Core 提供的内置 IoC 容器时，在 Startup.cs 文件中的 ConfigureServices 方法中注册希望注入的类型，如下代码所示。

```
// Registration of types into ASP.NET Core built-in container
public void ConfigureServices(IServiceCollection services)
{
    // Register out-of-the-box framework services.
    services.AddDbContext<CatalogContext>(c =>
    {
        c.UseSqlServer(Configuration["ConnectionString"]);
    },
    ServiceLifetime.Scoped
    );
    services.AddMvc();
    // Register custom application dependencies.
    services.AddScoped<IMyCustomRepository, MyCustomSQLRepository>();
}
```

在注册类型到 IoC 容器时，最常见的模式是注册一对类型 - 接口和相关的实现类。然后在您通过构造函数从 IoC 容器请求对象的时候，您会请求特定接口类型的对象。例如，在上述示例中，最后一行代码指出，任何依赖 IMyCustomRepository（接口或者抽象）的构造函数，IoC 容器将会注入一个 MyCustomSQLRepository 实现类的实例。

使用 Scrutor 库实现自动类型注册

当在 .NET Core 中使用 DI 时，您可能希望能够扫描程序集并通过约定自动注册其类型。此功能目前在 ASP.NET Core 中不可用。但是，您可以使用 [Scrutor](#) 库来实现。当您有需要需要在 IoC 容器中注册的类型时，此方法非常方便。

其他资源

- **Matthew King. 使用 Scrutor 注册服务**
<https://mking.io/blog/registering-services-with-scrutor>
- **Kristian Hellang. Scrutor 的 GitHub 仓库**
<https://github.com/khellang/Scrutor>

使用 Autofac 作为 IoC 容器

您还可以使用其它的 IoC 容器并将其插入到 ASP.NET Core 处理管道中。在 eShopOnContainers 的订单微服务中，使用了 [Autofac](#)。在使用 Autofac 时，您通常通过模块注册类型，这使您可以根据类型的位置拆分多个文件之间的注册类型。就像您可以将应用程序类型分布在多个类库中一样。

例如，以下代码是 [Ordering.API Web API](#) 项目中 [Autofac 应用模块](#) 中，您希望注入的类型。

```
public class ApplicationModule : Autofac.Module
{
    public string QueriesConnectionString { get; }

    public ApplicationModule(string qconstr)
    {
        QueriesConnectionString = qconstr;
    }
    protected override void Load(ContainerBuilder builder)
    {
        builder.Register(c => new OrderQueries(QueriesConnectionString))
            .As<IOrderQueries>()
            .InstancePerLifetimeScope();

        builder.RegisterType<BuyerRepository>()
            .As<IBuyerRepository>()
            .InstancePerLifetimeScope();

        builder.RegisterType<OrderRepository>()
            .As<IOrderRepository>()
            .InstancePerLifetimeScope();

        builder.RegisterType<RequestManager>()
            .As<IRequestManager>()
            .InstancePerLifetimeScope();
    }
}
```

注册的过程和概念非常类似于您使用内置的 ASP.NET Core IoC 容器注册类型的方式，只是在使用 Autofac 时语法有点不同。

在示例代码中，抽象 `IOrderRepository` 与其实现类 `OrderRepository` 一起注册，这意味着，每当构造函数通过 `IOrderRepository` 声明依赖项时，IoC 容器将注入 `OrderRepository` 类的实例。

实例作用域类型决定实例如何在同一服务或依赖项的请求之间共享。当请求依赖项时，IoC 容器可以返回如下结果：

- 每个生命周期内单例（在 ASP.NET Core 的 IoC 容器中称为 *scoped*）
- 每个依赖一个新实例（在 ASP.NET Core 的 IoC 容器中称为 *transient*）

- 在使用 IoC 容器的所有对象之间共享的单个实例（在 ASP.NET Core 的 IoC 容器中称为 *singleton*）

其他资源

- **ASP.NET Core 中的依赖注入简介**
<https://docs.microsoft.com/aspnet/core/fundamentals/dependency-injection>
- **Autofac. 官方文档**
<http://docs.autofac.org/en/latest/>
- **对比 ASP.NET Core IoC 容器服务生命周期和 Autofac IoC 容器示例范围 - Cesar de la Torre**
<https://blogs.msdn.microsoft.com/cesardelatorre/2017/01/26/comparing-asp-net-core-ioc-service-life-times-and-autofac-ioc-instance-scopes/>

实现命令和命令处理器模式

在上一节展示的通过构造函数的 DI 示例中，IoC 容器通过类中的构造函数注入了仓储。但到底是在哪里注入的？在简单的 Web API（例如，eShopOnContainers 中的目录微服务）中，您可以在 MVC 控制器的构造函数中注入，但是，在本节的初始代码（eShopOnContainers 中来自 Ordering.API 服务的 [CreateOrderCommandHandler](#) 类），依赖的注入通过特定的命令处理器来完成的。让我们解释一下命令处理器是什么，以及为什么要使用它。

命令模式本质上与本指南前面介绍的 CQRS 模式相关。CQRS 模式由两方面组成。第一方面是查询，通过 [Dapper](#) 微 ORM 使用简洁的查询，我们前面已经介绍过。第二方面是命令，它是事务的起点，以及来自服务外部的输入通道。

如图 9-24 所示，该模式基于从客户端接收命令，基于领域模型规则处理它们，最终使用事务持久化状态。

CQRS 中的“写入侧”高层视图

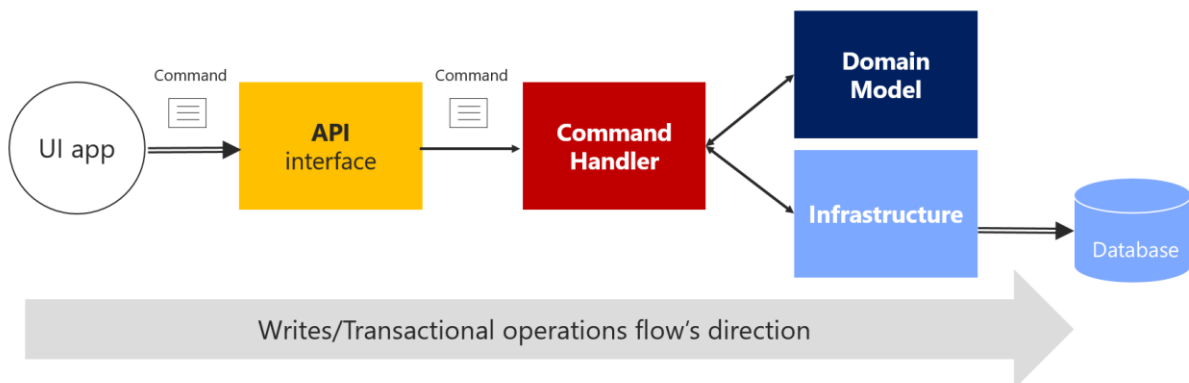


图 9-24. CQRS 模式中命令或“事务层面”的高级视图

命令类

命令是请求系统执行一个改变系统状态的操作的请求。命令是强制的，应该只执行一次。

由于命令是强制的，所以它们通常是使用命令语气中的谓词（例如，“创建”或“更新”）来命名的，并且可能包括聚合类型，如 `CreateOrderCommand`。与事件不同，命令不是过去的事实，它仅仅是一个请求，因此可能被拒绝。

命令可以源自来自 UI 的用户的初始请求，或者源自进程管理器，在进程管理器指示聚合执行一个操作的时候。

命令的一个重要特征是它应该由一个接收器仅仅处理一次。这一因为命令是您希望在应用程序中执行的单个操作或事务。例如，不应多次处理同一个创建订单命令。这是命令和事件之间的一个重要区别。事件可能被多次处理，因为许多系统或微服务可能都对事件感兴趣。

此外，在命令不是幂等的情况下，命令只处理一次是很重要的。如果命令可以执行多次而不改变结果，不管是因为命令的本质，还是因为系统处理命令的方式，则命令是幂等的。

最佳实践是在您的领域业务规则和不变量有意义的情况下，使您的命令和更新成为幂等。例如，使用同样的示例，如果由于任何原因（重试逻辑，黑客等等），相同的 `CreateOrder` 命令多次到达您的系统，您应当能够区分并确保不会多次创建订单。为此，您需要在操作中附加某种标识，并确定是否已处理该命令或更新。

将命令发送到单个接收方，您不应当发布命令。发布是用于说明事实的集成事件 - 即发生了某些事情，可能对事件接收器来说感兴趣。在事件的状况下，发布者不关心接收者哪些接收者得到事件，或它们做什么。集成事件是前几节中已经介绍的一个不同的故事。

命令实现为包含所有执行命令所需信息的数据字段或者集合的类。命令是一种特殊的数据传输对象 (DTO)，是专门用于请求更新或者事务的一种对象。该命令本身基于处理命令所需的信息，仅此而已。

如下代码展示了简化的 `CreateOrderCommand` 类。这是 `eShopOnContainers` 中用于订单服务的不变命令。

```
// DDD and CQRS patterns comment
// Note that it is recommended that you implement immutable commands
// In this case, immutability is achieved by having all the setters as private
// plus being able to update the data just once, when creating the object
// through the constructor.

// References on immutable commands:
// http://cQRS.nu/Faq
// https://docs.spine3.org/motivation/immutability.html
// http://blog.gauffin.org/2012/06/griffin-container-introducing-command-support/
// https://msdn.microsoft.com/library/bb383979.aspx
```

```

[DataContract]
public class CreateOrderCommand : IRequest<bool>
{
    [DataMember]
    private readonly List<OrderItemDTO> _orderItems;
    [DataMember]
    public string City { get; private set; }
    [DataMember]
    public string Street { get; private set; }
    [DataMember]
    public string State { get; private set; }
    [DataMember]
    public string Country { get; private set; }
    [DataMember]
    public string ZipCode { get; private set; }
    [DataMember]
    public string CardNumber { get; private set; }
    [DataMember]
    public string CardHolderName { get; private set; }
    [DataMember]
    public DateTime CardExpiration { get; private set; }
    [DataMember]
    public string CardSecurityNumber { get; private set; }
    [DataMember]
    public int CardTypeId { get; private set; }
    [DataMember]
    public IEnumerable<OrderItemDTO> OrderItems => _orderItems;

    public CreateOrderCommand()
    {
        _orderItems = new List<OrderItemDTO>();
    }

    public CreateOrderCommand(List<BasketItem> basketItems, string city,
        string street,
        string state, string country, string zipcode,
        string cardNumber, string cardHolderName, DateTime cardExpiration,
        string cardSecurityNumber, int cardTypeId) : this()
    {
        _orderItems = MapToOrderItems(basketItems);
        City = city;
        Street = street;
        State = state;
        Country = country;
        ZipCode = zipcode;
        CardNumber = cardNumber;
        CardHolderName = cardHolderName;
        CardSecurityNumber = cardSecurityNumber;
        CardTypeId = cardTypeId;
        CardExpiration = cardExpiration;
    }
}

public class OrderItemDTO
{
    public int ProductId { get; set; }
    public string ProductName { get; set; }
    public decimal UnitPrice { get; set; }
    public decimal Discount { get; set; }
    public int Units { get; set; }
}

```

```
        public string PictureUrl { get; set; }
    }
}
```

基本上，该命令类包含使用领域模型对象执行业务事务所需的所有数据。因此，命令是包含只读数据和没有行为的简单数据结构。命令的名称指示其用途。在如 C# 这样的许多语言中，命令被表示为类，但它们不是真正的面向对象意义上的类。

作为一个附加特征，命令是不变的，因为预期的用法是它们由领域模型直接处理。在其生命周期中它们不需要改变。在 C# 类中，不变性可以通过没有任何 setter 或其它改变内部状态的方法来实现。

例如，用于创建订单的命令类在数据方面可能与要创建的订单数据类似，但是您可能不需要同样的特性。例如，CreateOrderCommand 没有订单 ID，因为订单尚未创建。

许多命令类可以很简单，只需要一些关于需要改变状态的字段。如果您只是需要将订单的状态从“处理中”更新为“已付款”或“已发货”，则使用类似下面的命令即可：

```
[DataContract]
public class UpdateOrderStatusCommand
    : IAsyncRequest<bool>
{
    [DataMember]
    public string Status { get; private set; }
    [DataMember]
    public string OrderId { get; private set; }
    [DataMember]
    public string BuyerIdentityGuid { get; private set; }
}
```

有些开发人员将他们的 UI 请求对象与他们的命令 DTO 对象分开，但这只是一个偏好问题。这是一种乏味的拆分，没有多少附加值，对象几乎完全类似。举例来说，在 eShopOnContainers 中，有些命令直接来自客户端。

命令处理器类

应当为每个命令实现专用的命令处理器类。这是该模式的工作方式，它是使用命令对象、领域对象和基础架构仓储对象的地方。从 CQRS 和 DDD 的角度来说，命令处理器是应用层的核心。但是，所有的领域逻辑应当包含于领域类中 - 位于聚合根（根实体）、子实体、或者[领域服务](#)中，但在命令处理器中，它是来自应用层的类。

命令处理器接收命令，并从其所使用的聚合根获得结果。结果或者是命令成功执行，否则是一个异常。在异常情况下，系统状态应当不变。

命令处理器通常采取如下步骤：

- 接收命令对象，如 DTO（从[中介](#)或其它基础架构对象）

- 验证命令有效（如果中介没有验证）
- 实例化当前命令作为当前命令目标的聚合根对象实例
- 执行聚合根对象实例的方法，从命令获取必要的数据库
- 将聚合的新状态持久化到相关的数据库。最后的操作是实际的事务

通常，命令处理器处理由其聚合根（根实体）所驱动的一个聚合。如果多个聚合被所接收的单个命令所影响，则可以使用领域事件在多个聚合之间传播状态或操作。

这里重要的一点是，当一个命令被处理时，所有的领域逻辑应当在领域模型（聚合）中，完全封装且单元测试就绪。命令处理器只是作为从数据库中获取领域模型的一种方式，作为最后一步，告知基础架构层（仓储）在模型被修改之后持久化更新。这种方式的好处是您可以在隔离的、完全封装的、丰富的、行为领域模型上重构领域逻辑，而无需更新应用层或基础架构层，即管道级别的代码（命令处理器、Web API、仓储等等）。

当命令处理器变得复杂，逻辑太多时，意味着代码可能变味了。检查这些代码，如果找到领域逻辑，请重构代码以将该领域行为转移到领域对象（聚合根和子实体）的方法中。

作为命令处理器的示例，以下代码展示了在本章开头所看到的同样的 `CreateOrderCommandHandler` 类。这里我们希望突出了使用领域对象/聚合的处理方法和操作。


```

public class CreateOrderCommandHandler
    : IAsyncRequestHandler<CreateOrderCommand, bool>
{
    private readonly IOrderRepository _orderRepository;
    private readonly IIdentityService _identityService;
    private readonly IMediator _mediator;

    // Using DI to inject infrastructure persistence Repositories
    public CreateOrderCommandHandler(IMediator mediator,
        IOrderRepository orderRepository,
        IIdentityService identityService)
    {
        _orderRepository = orderRepository ??
            throw new ArgumentNullException(nameof(orderRepository));
        _identityService = identityService ??
            throw new ArgumentNullException(nameof(identityService));
        _mediator = mediator ??
            throw new ArgumentNullException(nameof(mediator));
    }

    public async Task<bool> Handle(CreateOrderCommand message)
    {
        // Create the Order AggregateRoot
        // Add child entities and value objects through the Order aggregate root
        // methods and constructor so validations, invariants, and business logic
        // make sure that consistency is preserved across the whole aggregate
        var address = new Address(message.Street, message.City, message.State,
            message.Country, message.ZipCode);
        var order = new Order(message.UserId, address, message.CardTypeId,
            message.CardNumber, message.CardSecurityNumber,
            message.CardHolderName, message.CardExpiration);

        foreach (var item in message.OrderItems)
        {
            order.AddOrderItem(item.ProductId, item.ProductName, item.UnitPrice,
                item.Discount, item.PictureUrl, item.Units);
        }

        _orderRepository.Add(order);

        return await _orderRepository.UnitOfWork
            .SaveEntitiesAsync();
    }
}

```

命令处理器应执行的其它步骤:

- 使用命令的数据操作聚合根的方法和行为
- 在领域对象内部, 在执行事务时引发领域事件, 但是从命令处理器的角度这是透明的。
- 如果聚合的操作成功完成且事务完成之后, 引发集成事件命令处理程序。(这些也可能由诸如仓储的基础架构类所引发)

其他资源

- **Mark Seemann. 在边界，应用程序不是面向对象的**
<http://blog.ploeh.dk/2011/05/31/AttheBoundaries,ApplicationsareNotObject-Oriented/>
- **命令和事件**
<http://cqrs.nu/Faq/commands-and-events>
- **命令处理器是做什么的？**
<http://cqrs.nu/Faq/command-handlers>
- **Jimmy Bogard. 领域命令模式——处理器**
<https://jimmybogard.com/domain-command-patterns-handlers/>
- **Jimmy Bogard. 领域命令模式——验证**
<https://jimmybogard.com/domain-command-patterns-validation/>

命令处理管道：如何触发命令处理器

下一个问题是如何调用命令处理器。您可以从每个相关的 ASP.NET Core 控制器中手工调用它。然而，这种方式过于耦合，并不理想。

其它两个主要选择（推荐选择）是：

- 通过内存中的中介者模式组件。
- 在控制器和命令处理器之间使用异步消息队列。

在命令管道中使用中介者模式（内存中）

如图 9-25 所示，在 CQRS 方式中，您使用智能中介器。类似于内存中的总线，根据所接收到的命令或 DTO 的类型重定向到正确的命令处理器。组件之间的黑色箭头表示对象之间的依赖关系（多数情况下，通过 DI 注入）及其相关交互。

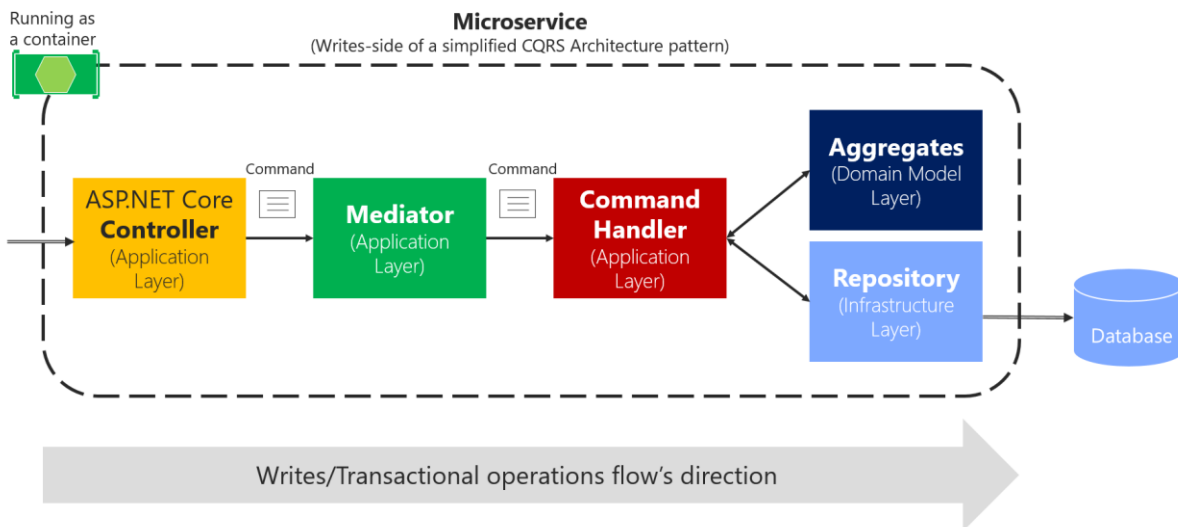


图9-25. 在单个 CQRS 微服务进程中使用中介者模式

使用中介者模式的意义在于，在企业应用中，处理请求可能会变得复杂。您希望能够加入开放的横切关注点（cross-cutting concerns），如日志记录、验证、审计和安全。在这些情况下，您可以依赖中介者管道（参见[中介者模式](#)）为这些额外的行为或横切关注点提供支持。

中介者是封装该处理“如何做”的对象，它根据状态、命令处理器被调用的方式、或提供给处理器的有效载荷来协调执行。使用中介者组件，您可以通过使用装饰器（或来自 MediatR v3 的[管道 behaviors](#)）以中心化和透明化的方式应用横切关注点。（更多信息，请参阅[装饰模式](#)）

装饰器和 behaviors 类似于[面向切面编程 \(AOP\)](#)，仅适用于由中介组件管理的特定处理管道。在 AOP 中，实现横切关注点的方面 (Aspects) 基于编译时注入的方面编织器 (aspect weaver) 或基于对象调用拦截实现。这两种典型的 AOP 方式有时被认为“像魔术一样”，因为不容易看出 APO 是如何工作的。当处理严重问题或者 bug 的时候，AOP 方式难以调试。反过来说，这些装饰器/behaviors 是显式的，只在中介者的上下文中应用，因此调试更为容易预测和简单。

例如，在 eShopOnContainers 中的订单微服务中，我们实现了两个示例 behaviors，一个 [LogBehavior](#) 类和一个 [ValidatorBehavior](#) 类。通过展示 eShopOnContainers 如何使用 [MediatR 3](#)，下一节将介绍 behaviors 的实现。

在命令管道中使用消息队列（进程外）

另一个选择是使用基于 brokers 或者消息队列的异步消息，如图 9-26 所示。该选择也可以在命令处理器之前与中介组件组合在一起。

使用消息的CQRS架构模式的写入侧

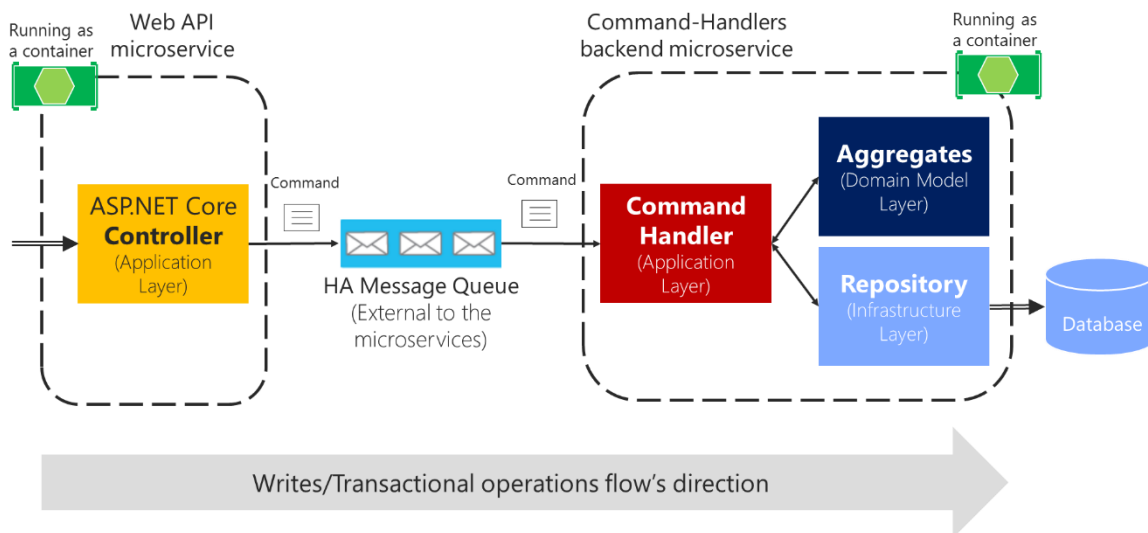


图 9-26. 使用 CQRS 命令的消息队列 (进程外通信和进程间通信)

使用消息队列接收命令可能使得命令管道更为复杂，因为您可能需要将管道拆分为两个通过外部消息队列连接的进程。但是，如果您需要基于异步消息传递改进可伸缩性和性能，则应该使用它。考虑到在图 9-26 中，MVC 控制器仅仅需要将命令消息投递到管道中并返回。然后命令处理器按照自己的速度处理消息。这是队列巨大的优势 - 当需要超可伸缩性时，队列可以充当缓冲区，例如，对于具有大量入口数据的股票或任何其它场景。

但是，由于消息队列的异步特性，您需要找出如何与客户端应用通信，以了解命令处理的成功或失败。作为规则，您永远不应使用“发射后不管”命令。任何商业应用需要直到被处理的命令是否被成功，或者至少有效和被接受。

因此，在验证提交给异步队列的命令消息之后，能够对客户端做出响应增加了系统的复杂性，与在运行事务之后返回操作结果的进程内命令进程相比，使用消息队列，可能需要通过其它操作结果消息来返回命令处理结果，这需要在您的系统中额外的组件和自定义通讯。

此外，异步命令是单向命令，在许多场景下可能并不需要，正如 Burtsev Alexey 和 Greg Yound 的一次[网聊](#)中，所做的有趣交流一样。

[Burtsev Alexey] 我发现人们大量使用异步命令处理或者单向命令消息而没有任何理由这样做（他们没有做长时间的操作，他们也没有执行外部异步代码，甚至他们没有跨应用程序边界就使用消息总线）。为什么他们要引入这种不必要的复杂性？实际上，到目前为止，我都还没有看到一个 CQRS 的示例代码使用阻塞命令处理器，尽管在大多数情况下它都会正常工作。

[Greg Young] 异步命令并不存在，它实际上是另一种事件。如果我必须接收您发送给我的东西，如果我不同意的话，就抛出事件，这就不是您告诉我去做什么。这是您告诉我已经发生了什么。这与一开始似乎有点不同，但它有很多含义。

异步命令极大增加了系统的复杂性，因为没有简单的方式来指示失败。因此，除了需要伸缩性要求或特性情况下通过消息在内部微服务之间通信时，不建议使用异步消息命令。这些场景下，您必须为故障设计一个单独的报告和恢复系统。

在初始版本的 eShopOnContainers 中，我们决定使用同步的命令处理，从 HTTP 请求开始，由中介者模式驱动。这使您很容易可以返回处理的成功或者失败，如在 [CreateOrderCommandHandler](#) 中的实现。

在任何情况下，这应该基于您的应用程序或微服务的业务需求来决定。

使用中介者模式 (MediatR) 实现命令处理管道

作为示例实现，本指南建议基于中介者模式的进程内管道来驱动命令接收，并将它们（在内存中）路由到正确的命令处理器。本指南还建议应用 [behaviors](#) 以分离横切关注点。

对于 .NET Core 中的实现，有多个实现中介者模式的开源库可用。本指南中使用的是 [MediatR](#) 开源库（由 Jimmy Bogard 创建），但您也可以使用其它方式。MediatR 是一个小且简单的库，它允许您使用装饰器或 behaviors 处理内存中的消息，如命令。

使用中介者模式可以帮助您减少耦合，并隔离所请求的工作关注点，同时自动连接到执行该工作的处理器 - 这里，就是命令处理器。

在审阅本指南时，Jimmy Bogard 给出了另外一个使用中介者模式的好的理由。

我认为最值得一提的是测试 - 它提供了一个很好的您的应用程序行为的一致窗口，请求进，响应出。我们发现在构建持续的行为测试方面相当有价值。

首先，让我们看一个示例 WebAPI 中您实际使用中介者对象的控制器代码。如果您没有使用中介者对象，则需要为该控制器注入所有的依赖项，例如日志器对象或其它对象。因此，构造函数将非常复杂。另一方面，如果您使用中介者对象，您的控制器的构造函数则简单得多，仅有几个依赖项而不是许多依赖项，如果每个横切操作都有一个依赖项的话。如下示例所示：

```
public class MyMicroserviceController : Controller
{
    public MyMicroserviceController(IMediator mediator,
                                    IMyMicroserviceQueries microserviceQueries)
        // ...
}
```

可以看到，中介者提供了一个干净和精干的 Web API 控制器构造函数。此外，在控制器方法中，向中介者对象发送命令的代码几乎就是一行。

```
[Route("new")]
[HttpPost]
public async Task<IActionResult> ExecuteBusinessOperation([FromBody]RunOpCommand
                                                         runOperationCommand)
{
    var commandResult = await _mediator.SendAsync(runOperationCommand);

    return commandResult ? (IActionResult)Ok() : (IActionResult)BadRequest();
}
```

实现幂等命令

在 eShopOnContainers 中，比上面示例更为高级的示例是从 Ordering 微服务提交 CreateOrderCommand 对象的例子。但是，由于 Ordering 业务流程稍微复杂一点，在我们的例子中，它实际上是从购物篮微服务开始的，提交 CreateOrderCommand 对象的这个操作是从一个名为 [UserCheckoutAcceptedIntegrationEvent.cs](#) 的集成事件处理程序执行的，而不是如前面较简单的例子中，一个简单的客户端应用程序的 WebAPI 控制器调用。

不过，将命令提交给 MediatR 的操作非常相似，如下面的代码所示。

```
var createOrderCommand = new CreateOrderCommand(eventMsg.Basket.Items,
                                                  eventMsg.UserId, eventMsg.City,
                                                  eventMsg.Street, eventMsg.State,
                                                  eventMsg.Country, eventMsg.ZipCode,
                                                  eventMsg.CardNumber,
                                                  eventMsg.CardHolderName,
                                                  eventMsg.CardExpiration,
                                                  eventMsg.CardSecurityNumber,
                                                  eventMsg.CardTypeId);

var requestCreateOrder = new IdentifiedCommand<CreateOrderCommand, bool>(
    createOrderCommand,
    eventMsg.RequestId);

result = await _mediator.Send(requestCreateOrder);
```

不过，这种情况也有点高级，因为我们也在实施幂等指令。CreateOrderCommand 过程应该是幂等的，所以如果同一个消息通过网络重复（因为任何原因，如重试），相同的业务订单将被处理。这是通过封装业务命令（在这种情况下为 CreateOrderCommand）并将其嵌入到一个通用的 IdentifiedCommand 中来实现的，后者由通过网络传递的每个消息的 ID 进行跟踪，该 ID 必须是幂等的。

在下面的代码中，您可以看到 IdentifiedCommand 只不过是一个带有 ID 的 DTO 以及包装的业务命令对象。

```

public class IdentifiedCommand<T, R> : IRequest<R>
    where T : IRequest<R>
{
    public T Command { get; }
    public Guid Id { get; }

    public IdentifiedCommand(T command, Guid id)
    {
        Command = command;
        Id = id;
    }
}

```

然后，IdentifiedCommand 的 CommandHandler（名为 [IdentifiedCommandHandler.cs](#)）将主要检查作为消息一部分的 ID 是否已经存在于表中。如果它已经存在，那么这个命令将不会被再次处理，所以它的行为就是一个幂等指令。该基础结构代码由以下 `_requestManager.ExistsAsync()` 方法调用执行。

```

// IdentifiedCommandHandler.cs

public class IdentifiedCommandHandler<T, R> :
    IAsyncRequestHandler<IdentifiedCommand<T, R>, R>
    where T : IRequest<R>
{
    private readonly IMediator _mediator;
    private readonly IRequestManager _requestManager;

    public IdentifiedCommandHandler(IMediator mediator,
        IRequestManager requestManager)
    {
        _mediator = mediator;
        _requestManager = requestManager;
    }

    protected virtual R CreateResultForDuplicateRequest()
    {
        return default(R);
    }

    public async Task<R> Handle(IdentifiedCommand<T, R> message)
    {
        var alreadyExists = await _requestManager.ExistsAsync(message.Id);
        if (alreadyExists)
        {
            return CreateResultForDuplicateRequest();
        }
        else
        {
            await _requestManager.CreateRequestForCommandAsync<T>(message.Id);

            // Send the embeded business command to mediator
            // so it runs its related CommandHandler
            var result = await _mediator.Send(message.Command);

            return result;
        }
    }
}

```

```
}  
}
```

由于 IdentifiedCommand 的行为类似于业务命令的卷宗，当业务命令需要被处理时，因为它不是一个重复的 Id，那么它将采用该内部业务命令并将其重新提交给 Mediator，如如上所示代码的最后部分，从 [IdentifiedCommandHandler.cs](#) 运行 _mediator.Send(message.Command)。

这样做的话，它会链接并运行业务命令处理程序，在这种情况下，它是按照 Ordering 数据库运行事务的 [CreateOrderCommandHandler](#)，如以下代码所示：

```
// CreateOrderCommandHandler.cs  
public class CreateOrderCommandHandler  
    : IAsyncRequestHandler<CreateOrderCommand, bool>  
{  
    private readonly IOrderRepository _orderRepository;  
    private readonly IIdentityService _identityService;  
    private readonly IMediator _mediator;  
  
    // Using DI to inject infrastructure persistence Repositories  
    public CreateOrderCommandHandler(IMediator mediator,  
        IOrderRepository orderRepository,  
        IIdentityService identityService)  
    {  
        _orderRepository = orderRepository ??  
            throw new ArgumentNullException(nameof(orderRepository));  
        _identityService = identityService ??  
            throw new ArgumentNullException(nameof(identityService));  
        _mediator = mediator ??  
            throw new ArgumentNullException(nameof(mediator));  
    }  
  
    public async Task<bool> Handle(CreateOrderCommand message)  
    {  
        // Add/Update the Buyer AggregateRoot  
        var address = new Address(message.Street, message.City, message.State,  
            message.Country, message.ZipCode);  
        var order = new Order(message.UserId, address, message.CardTypeId,  
            message.CardNumber, message.CardSecurityNumber,  
            message.CardHolderName, message.CardExpiration);  
  
        foreach (var item in message.OrderItems)  
        {  
            order.AddOrderItem(item.ProductId, item.ProductName, item.UnitPrice,  
                item.Discount, item.PictureUrl, item.Units);  
        }  
  
        _orderRepository.Add(order);  
  
        return await _orderRepository.UnitOfWork  
            .SaveEntitiesAsync();  
    }  
}
```


注册 MediatR 使用的类型

为了使 MediatR 知道您的命令处理程序类，您需要在 IoC 容器中注册中介类和命令处理程序类。默认情况下，MediatR 使用 Autofac 作为 IoC 容器，但是您也可以使用内置的 ASP.NET Core IoC 容器或 MediatR 支持的任何其他容器。

下面的代码展示了使用 Autofac 模块时如何注册 Mediator 类型和命令。

```
public class MediatorModule : Autofac.Module
{
    protected override void Load(ContainerBuilder builder)
    {
        builder.RegisterAssemblyTypes(typeof(IMediator).GetTypeInfo().Assembly)
            .AsImplementedInterfaces();

        // Register all the Command classes (they implement IAsyncRequestHandler)
        // in assembly holding the Commands
        builder.RegisterAssemblyTypes(
            typeof(CreateOrderCommand).GetTypeInfo().Assembly)
            .AsClosedTypesOf(typeof(IAsyncRequestHandler<, >));

        // Other types registration
        //...
    }
}
```

这是 MediatR 的“魔术发生”的地方。

由于每个命令处理程序都使用通用的 `IAsyncRequestHandler<T>` 实现接口，因此，在注册程序集时，由于 `CommandHandler` 类中声明的关系，代码将 `RegisterAssemblyTypes` 中的所有类型都作为 `RequestHandler` 注册，同时将 `CommandHandlers` 与其命令相关联，在下面的例子中：

```
public class CreateOrderCommandHandler
    : IAsyncRequestHandler<CreateOrderCommand, bool>
{
```

这是将命令与命令处理程序相关联的代码。该处理器只是一个简单的类，但它继承自 `RequestHandler<T>`，并且 MediatR 确保它被正确的载荷所调用。

在使用 MediatR 中和 Behaviors 处理命令时应用横切关注点

还有一件事：能够应用横切关注点到 mediator 管道中。您还可以在 Autofac 注册模块的代码的末尾，看到它是如何注册一个 behavior 类型的，具体来说是一个自定义的 LoggingBehavior 类和一个 ValidatorBehavior 类。但是您可以添加其他可以考虑的自定义行为。

```
public class MediatorModule : Autofac.Module
{
    protected override void Load(ContainerBuilder builder)
    {
        builder.RegisterAssemblyTypes(typeof(IMediator).GetTypeInfo().Assembly)
            .AsImplementedInterfaces();

        // Register all the Command classes (they implement IAsyncRequestHandler)
        // in assembly holding the Commands
        builder.RegisterAssemblyTypes(
            typeof(CreateOrderCommand).GetTypeInfo().Assembly)
            .AsClosedTypesOf(typeof(IAsyncRequestHandler<, >));

        // Other types registration
        //...

        builder.RegisterGeneric(typeof(LoggingBehavior<, >))
            .As(typeof(IPipelineBehavior<, >));

        builder.RegisterGeneric(typeof(ValidatorBehavior<, >))
            .As(typeof(IPipelineBehavior<, >));
    }
}
```

[LoggingBehavior](#) 类可以实现为如下代码，它记录了正在执行的命令处理程序的信息，以及它是否成功。

```
public class LoggingBehavior<TRequest, TResponse>
    : IPipelineBehavior<TRequest, TResponse>
{
    private readonly ILogger<LoggingBehavior<TRequest, TResponse>> _logger;
    public LoggingBehavior(ILogger<LoggingBehavior<TRequest, TResponse>> logger) =>
        _logger = logger;

    public async Task<TResponse> Handle(TRequest request,
        RequestHandlerDelegate<TResponse> next)
    {
        _logger.LogInformation($"Handling {typeof(TRequest).Name}");
        var response = await next();
        _logger.LogInformation($"Handled {typeof(TResponse).Name}");
        return response;
    }
}
```

只要实现这个 behavior 类，并注册 behavior 类型，所有通过 MediatR 处理的命令将被记录关于执行的信息。

eShopOnContainers 中 ordering 微服务还为基本验证应用了第二种行为，即依赖于 [FluentValidation](#) 库的 [ValidatorBehavior](#) 类，如以下代码所示：

```
public class ValidatorBehavior<TRequest, TResponse>
    : IPipelineBehavior<TRequest, TResponse>
{
    private readonly IValidator<TRequest>[] _validators;
    public ValidatorBehavior(IValidator<TRequest>[] validators) =>
        _validators = validators;

    public async Task<TResponse> Handle(TRequest request,
        RequestHandlerDelegate<TResponse> next)
    {
        var failures = _validators
            .Select(v => v.Validate(request))
            .SelectMany(result => result.Errors)
            .Where(error => error != null)
            .ToList();

        if (failures.Any())
        {
            throw new OrderingDomainException(
                $"Command Validation Errors for type {typeof(TRequest).Name}",
                new ValidationException("Validation exception", failures));
        }

        var response = await next();
        return response;
    }
}
```

然后，基于 [FluentValidation](#) 库，我们创建了对使用 CreateOrderCommand 传递的数据的验证，如下面的代码所示：

```

public class CreateOrderCommandValidator : AbstractValidator<CreateOrderCommand>
{
    public CreateOrderCommandValidator()
    {
        RuleFor(command => command.City).NotEmpty();
        RuleFor(command => command.Street).NotEmpty();
        RuleFor(command => command.State).NotEmpty();
        RuleFor(command => command.Country).NotEmpty();
        RuleFor(command => command.ZipCode).NotEmpty();
        RuleFor(command => command.CardNumber).NotEmpty().Length(12, 19);
        RuleFor(command => command.CardHolderName).NotEmpty();
        RuleFor(command =>
command.CardExpiration).NotEmpty().Must(BeValidExpirationDate).WithMessage("Please
specify a valid card expiration date");
        RuleFor(command => command.CardSecurityNumber).NotEmpty().Length(3);
        RuleFor(command => command.CardTypeId).NotEmpty();
        RuleFor(command =>
command.OrderItems).Must(ContainOrderItems).WithMessage("No order items found");
    }

    private bool BeValidExpirationDate(DateTime dateTime)
    {
        return dateTime >= DateTime.UtcNow;
    }

    private bool ContainOrderItems(IEnumerable<OrderItemDTO> orderItems)
    {
        return orderItems.Any();
    }
}

```

您可以创建其它的验证。这是执行您的命令验证的非常干净和优雅的方式。

以类似的方式，您可以实现其他 behaviors，以处理您在处理命令时应用于命令的其他方面或横切关注点。

其他资源

中介者模式

- **中介者模式**
https://en.wikipedia.org/wiki/Mediator_pattern

装饰器模式

- **装饰器模式**
https://en.wikipedia.org/wiki/Decorator_pattern

MediatR (Jimmy Bogard)

- **MediatR**. GitHub 仓库.
<https://github.com/jbogard/MediatR>
- **使用 MediatR 和 AutoMapper 的 CQRS**
<https://lostechies.com/jimmybogard/2015/05/05/cqrs-with-mediatr-and-automapper/>
- **将您的控制器加入饮食：POST 和命令**
<https://lostechies.com/jimmybogard/2013/12/19/put-your-controllers-on-a-diet-posts-and-commands/>
- **使用中介者管道应对横切关注点**
<https://lostechies.com/jimmybogard/2014/09/09/tackling-cross-cutting-concerns-with-a-mediator-pipeline/>
- **CQRS 和 REST：最佳搭档**
<https://lostechies.com/jimmybogard/2016/06/01/cqrs-and-rest-the-perfect-match/>
- **MediatR 管道示例**
<https://lostechies.com/jimmybogard/2016/10/13/mediatr-pipeline-examples/>
- **针对 MediatR 和 ASP.NET Core 的垂直切片测试**
<https://lostechies.com/jimmybogard/2016/10/24/vertical-slice-test-fixtures-for-mediatr-and-asp-net-core/>
- **微软依赖注入的 MediatR 扩展现已发布**
<https://lostechies.com/jimmybogard/2016/07/19/mediatr-extensions-for-microsoft-dependency-injection-released/>

Fluent 验证

- **Jeremy Skinner. FluentValidation**. GitHub 仓库.
<https://github.com/JeremySkinner/FluentValidation>

实现弹性应用

愿景

您的微服务和基于云的应用程序一定会包含终将发生的局部故障。您需要设计您的应用程序，以便其能够从局部故障中恢复。

弹性是从故障中恢复并继续工作的能力。它并不是避免故障，而是接受故障将会发生的事实，以避免停机或丢失数据的方式对其作出响应。弹性的目标是在故障之后将应用程序恢复到完全正常运行的状态。

设计和部署基于微服务的应用程序很有挑战性。但是，您还需要在某种必然发生故障的环境下保持应用程序运行。因此，应用程序应该具有弹性。其应被设计为可应对局部故障，例如网络故障、节点或者在云中的虚拟机崩溃。甚至集群中的微服务(容器中)被转移到其它节点，导致应用程序中出现间歇性的短暂故障。

您的应用程序中的许多独立组件还应包含运行状态监视功能。通过遵循本章中的准则，即使在复杂和基于云的部署中出现短暂停机或常规故障的情况下，您创建的应用程序也可以顺利地工作。

处理局部故障

在分布式系统，例如基于微服务的应用程序中，经常存在部分故障的风险。例如，一个微服务/容器可能失败，或者在短时间内无法响应。或者一个 VM 或服务器可能会崩溃。由于客户端和服务端是独立的进程，因此服务可能无法及时响应客户端的请求。服务可能过载导致对请求的响应速度非常慢，或者由于网络问题，服务短时间不可用。

例如，细想来自 eShopOnContainers 示例应用中的“订单详情页”。如果当用户试图提交订单时，订单微服务没有响应，一个客户端进程的错误实现（该 MVC 应用程序）— 例如，如果客户端代码使用没有超时的同步 RPCs 调用，将导致阻塞线程无限地等待响应。除了不好的用户体验之外，每个无响应的等待都会消耗或者阻塞线程，而线程在高度可伸缩的应用程序中是极其珍贵的资源。如果有许多阻塞的线程，最终应用程序的运行时将耗尽线程。在这种情况下，应用程序可能变的全面没有响应而不是局部没有响应，如图 10-1 所示。

局部故障

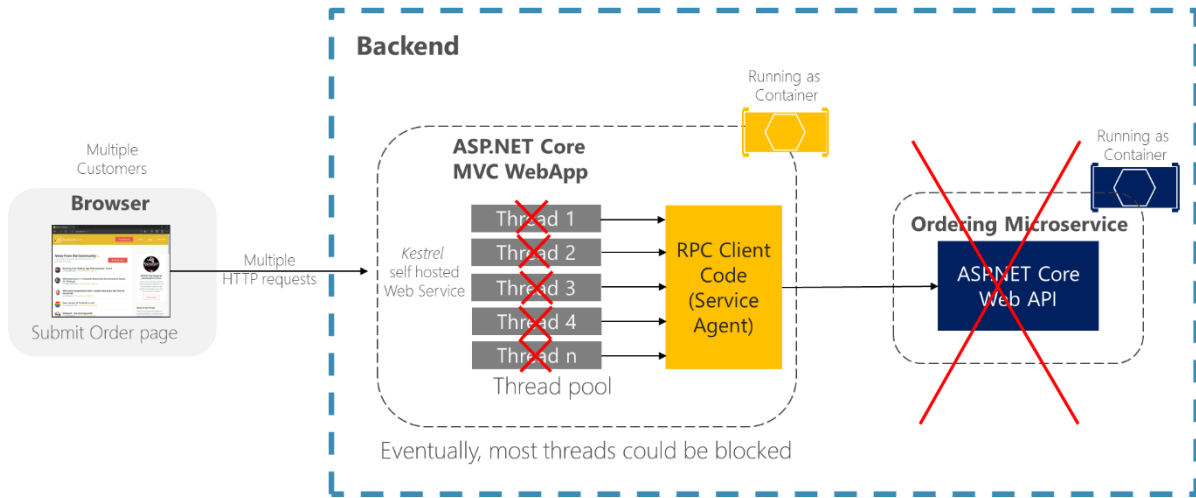


图 10-1. 由于影响服务线程可用性的依赖关系导致部分故障

在大型基于微服务的应用程序中，任何局部故障可能会被放大，特别是如果多数内部的微服务基于同步 HTTP 调用协作时（这被认为是反模式）。考虑一下每天接收数以百万计传入调用的系统。如果您的系统有一个基于长链的同步 HTTP 调用的错误设计，这些传入的调用可能会导致数百万的传出调用（假设比率为 1: 4）到数十个作为同步依赖的内部微服务。这种情况如图 10-2 所示，尤其是依赖 #3。

多个分布式依赖

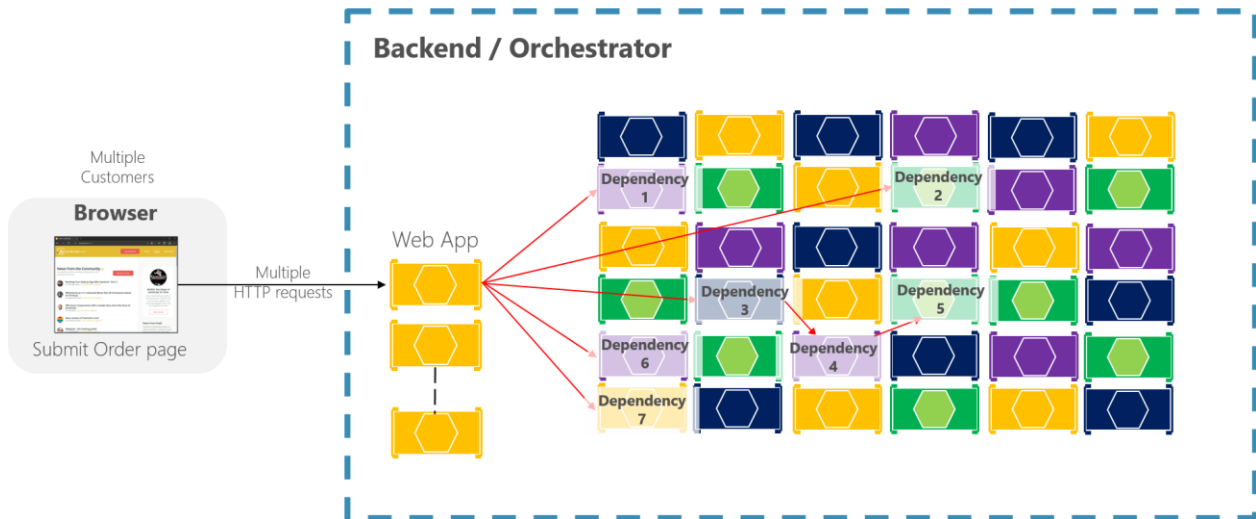


图 10-2. 具有长链 HTTP 请求的错误设计的影响

在分布式和基于云的系统，即使每个依赖项本身具有很好的可用性，间歇性故障也肯定会发生。这应该是您需要考虑的事实。

如果您没有设计和实现用以确保容错的技术，即使是小的停机也可能被放大。例如，由于连锁反应，每个月有 99.99% 可用性的 50 个依赖项都会导致数小时的停机时间。当一个微服务依赖项在处理大量请求时失败时，该故障会很快导致每个服务中的所有可用请求线程饱和，使整个应用程序崩溃。

微服务中的局部故障被放大

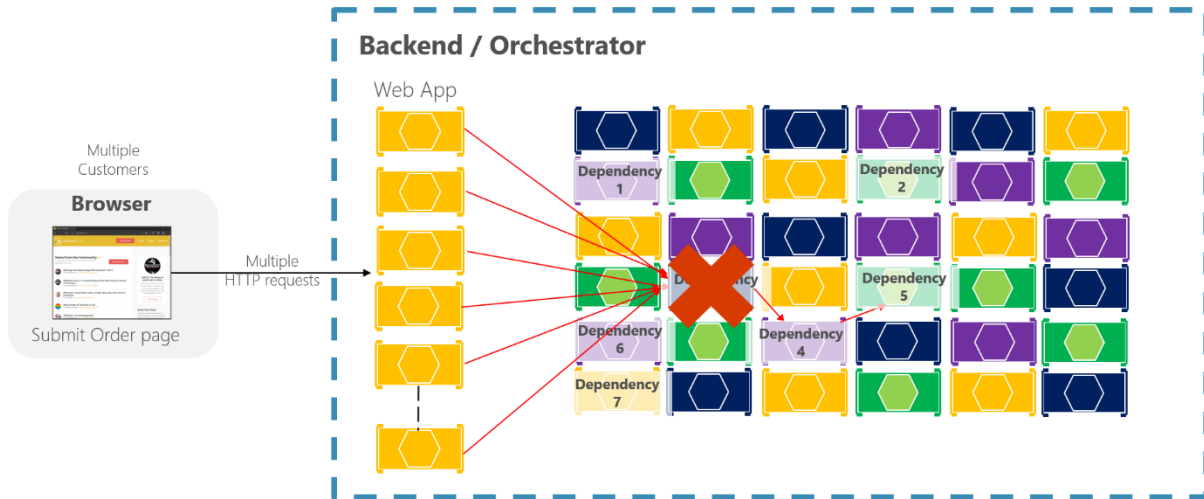


图 10-3. 具有长链的同步 HTTP 调用的微服务放大局部故障

为了最大限度减少此类问题，在“异步微服务集成强制微服务自治”一节中（在体系结构章节中），我们鼓励您在跨内部微服务调用中使用异步通讯。我们将在下一节简要说明。

此外，设计微服务和客户端以处理局部故障是很有必要的，即构建有弹性的微服务和客户端应用程序。

处理局部故障的策略

处理局部故障的策略包括如下内容。

在内部微服务之间使用异步通讯（例如，基于消息的通讯）。最好不要在内部微服务之间创建同步 HTTP 调用的长链，因为错误的设计终将成为导致故障停机的主要原因。相反，除了客户端应用程序和第一级微服务或者细粒度的 API 网关之间的前端通信之外，在初始的请求/响应周期后，建议仅使用异步（基于消息）的通信。最终一致性和事件驱动架构将有助于最大限度地减少涟漪效应。这些方法强制执行较高级别的微服务自治，防止此处提到的问题。

使用指数补偿重试。当服务仅在短期内不可用时，通过一定次数的重试调用，此技术有助于避免短时间和间歇性故障。这可能是在间歇性网络故障的情况下，或者在微服务/容器被迁移到集群中的其它节点时发生。但是，如果这些重试不是用断路器模式正确设计，它会加剧连锁反应，最终导致[拒绝服务 \(DoS\)](#)。

解决网络超时。通常，客户端应设计为非无限期阻塞，并在等待响应时始终使用超时。使用超时可确保资源不会无限期地被占用。

使用断路器模式。在这种方式中，客户端进程跟踪失败请求的数量。如果错误率超过配置的限制，则触发“断路器”跳闸，从而使进一步的尝试立即失败。（如果大量请求失败，则表明该服务不可用，继续发送请求毫无意义）在超时之后，客户端应再次尝试，如果新请求成功，则关闭断路器。

提供回退。在这种方式中，客户端进程在请求失败时执行回退逻辑，例如返回缓存的数据或者默认值。这是适合查询的方式，对于更新或者命令则更为复杂。

限制排队请求的数量。客户端应对微服务客户端可以发送到特定服务的未完成请求数量施加上限。如果已经达到上限，继续发出请求可能毫无意义，这些尝试应该立即失败。在实施方面，Polly [隔板隔离策略](#)可以用来满足这一需求。该方法实质上是一个并行的节流 [SemaphoreSlim](#) 实现。它也允许在隔板外面有一个“队列”。即使在执行之前，您可以预先将超出的负荷入库（例如，当认为容量已满时）。这使得它对某些故障场景的响应比断路器速度更快，因为断路器等待故障发生。Polly 隔板策略对象暴露隔板和队列“满”的程度，并提供了溢出事件，所以也可以用于自动化的水平缩放。

其他资源

- **弹性模式**
<https://docs.microsoft.com/azure/architecture/patterns/category/resiliency>
- **添加弹性和优化性能**
<https://msdn.microsoft.com/library/jj591574.aspx>
- **Bulkhead.** GitHub 仓库。使用 Polly 策略实现
<https://github.com/App-vNext/Polly/wiki/Bulkhead>
- **为 Azure 设计弹性应用**
<https://docs.microsoft.com/azure/architecture/resiliency/>
- **瞬态故障处理**
<https://docs.microsoft.com/azure/architecture/best-practices/transient-faults>

使用指数补偿实现重试

使用[指数补偿重试](#)是一种尝试重试操作的技术，使用指数递增的等待时间，直到达到最大重试次数（[指数补偿](#)）。该技术包含了这样一个事实：由于任何原因，云资源可能间歇性地不可用几秒钟。例如，编排引擎可能将容器转移到集群中的另一个节点以进行负载均衡。在这段时间内，有些请求可能会失败。另一个示例可能是类似 SQL Azure 这样的数据库，其中数据库可以转移到另一台服务器进行负载均衡，导致数据库在数秒内不可用。

有多种方式来实现具有指数补偿的重试逻辑。

实现弹性 Entity Framework Core SQL 连接

对于 Azure SQL DB, Entity Framework Core 库已经提供了内部数据连接的弹性和重试逻辑。如果您希望拥有[弹性 EF Core 连接](#), 您需要对每个 DbContext 连接启用 Entity Framework 执行策略。

例如, 下列代码在 EF Core 连接层启用了如果连接失败进行重试的弹性 SQL 连接。

```
// Startup.cs from any ASP.NET Core Web API
public class Startup
{
    // Other code ...
    public IServiceCollection ConfigureServices(IServiceCollection services)
    {
        // ...
        services.AddDbContext<OrderingContext>(options =>
        {
            options.UseSqlServer(Configuration["ConnectionString"],
                sqlServerOptionsAction: sqlOptions =>
                {
                    sqlOptions.EnableRetryOnFailure(
                        maxRetryCount: 5,
                        maxRetryDelay: TimeSpan.FromSeconds(30),
                        errorNumbersToAdd: null);
                });
        });
    }
    //...
}
```

执行策略和使用 BeginTransaction 和多 DbContexts 的显式事务

当在 EF Core 连接中启用重试时, 您使用 EF Core 执行的每个操作都是独立的、可重试的操作。如果发生瞬时故障, 则每个查询和每个对 SaveChanges 的调用都将作为一个单元进行重试。

但是, 如果您的代码使用 BeginTransaction 启动一个事务, 则您要定义您自己的一组需要视为一个单元的操作, 如果发生故障, 事务内所有的操作都将被回滚。如果在使用 EF 执行策略 (重试策略) 时尝试执行该事务, 并且在事务中包含多个 DbContexts 的 SaveChanges 调用, 则会看到类似下面的异常。

```
System.InvalidOperationException: The configured execution strategy
'SqlServerRetryingExecutionStrategy' does not support user initiated
transactions. Use the execution strategy returned by
'DbContext.Database.CreateExecutionStrategy()' to execute all the operations in
the transaction as a retrievable unit.
```

解决方案是用表示所有需要执行的内容的委托来手动调用 EF 执行策略。如果发生瞬态故障, 执行策略将重新调用该委托。例如, 下面的代码演示了在更新产品, 和然后需要使用不同的 DbContext 来保存 ProductPriceChangedIntegrationEvent 对象的情况下, 如何在 eShopOnContainers 中实现使用两个 DbContext (_catalogContext 和 IntegrationEventLogContext)。

```

public async Task<IActionResult> UpdateProduct([FromBody]CatalogItem
                                             productToUpdate)
{
    // Other code ...

    // Update current product
    catalogItem = productToUpdate;

    // Use of an EF Core resiliency strategy when using multiple DbContexts
    // within an explicit transaction
    // See:
    // https://docs.microsoft.com/ef/core/miscellaneous/connection-resiliency
    var strategy = _catalogContext.Database.CreateExecutionStrategy();

    await strategy.ExecuteAsync(async () =>
    {
        // Achieving atomicity between original Catalog database operation and the
        // IntegrationEventLog thanks to a local transaction
        using (var transaction = _catalogContext.Database.BeginTransaction())
        {
            _catalogContext.CatalogItems.Update(catalogItem);
            await _catalogContext.SaveChangesAsync();

            // Save to EventLog only if product price changed
            if (raiseProductPriceChangedEvent)
                await _integrationEventLogService.SaveEventAsync(priceChangedEvent);

            transaction.Commit();
        }
    });
}

```

第一个 DbContext 是 `_catalogContext`，第二个 DbContext 是在 `_integrationEventLogService` 对象中。Commit 使用了 EF 执行策略执行跨多个 DbContext 的操作。

其他资源

- **Entity Framework 中的弹性连接和命令捕获**
<https://docs.microsoft.com/azure/architecture/patterns/category/resiliency>
- **Cesar de la Torre. 使用弹性的 EF Core SQL 连接和事务**
<https://blogs.msdn.microsoft.com/cesardelatorre/2017/03/26/using-resilient-entity-framework-core-sql-connections-and-transactions-retries-with-exponential-backoff/>

使用自定义的带有指数补偿的 HTTP 调用重试

为了创建弹性微服务，您需要处理可能出现的 HTTP 故障场景。为此，您可以使用指数补偿来创建自己的重试实现。

除了处理暂时的资源不可用，指数补偿还需要考虑云提供者可能会限制资源的可用性，以防止资源使用过量。例如，非常快速地创建过多的连接请求可能被云提供者视为拒绝服务 (DoS) 攻击。因此，在遇到容量阈值时，您需要提供一种机制来相应缩减连接请求。

作为初步探索，您可以使用指数补偿工具类实现您自己的代码，比如在 [RetryWithExponentialBackoff.cs](#) 中加上如下代码（代码也在 [GitHub 仓库](#)中）。

```
public sealed class RetryWithExponentialBackoff
{
    private readonly int maxRetries, delayMilliseconds, maxDelayMilliseconds;

    public RetryWithExponentialBackoff(int maxRetries = 50,
                                       int delayMilliseconds = 200,
                                       int maxDelayMilliseconds = 2000)
    {
        this.maxRetries = maxRetries;
        this.delayMilliseconds = delayMilliseconds;
        this.maxDelayMilliseconds = maxDelayMilliseconds;
    }

    public async Task RunAsync(Func<Task> func)
    {
        ExponentialBackoff backoff = new ExponentialBackoff(this.maxRetries,
                                                           this.delayMilliseconds,
                                                           this.maxDelayMilliseconds);

        retry:
        try
        {
            await func();
        }
        catch (Exception ex) when (ex is TimeoutException ||
                                   ex is System.Net.Http.HttpRequestException)
        {
            Debug.WriteLine("Exception raised is: " +
                           ex.GetType().ToString() +
                           " -Message: " + ex.Message +
                           " -- Inner Message: " +
                           ex.InnerException.Message);
            await backoff.Delay();
            goto retry;
        }
    }
}

public struct ExponentialBackoff
{
    private readonly int m_maxRetries, m_delayMilliseconds, m_maxDelayMilliseconds;
    private int m_retries, m_pow;

    public ExponentialBackoff(int maxRetries, int delayMilliseconds,
                              int maxDelayMilliseconds)
```

```

{
    m_maxRetries = maxRetries;
    m_delayMilliseconds = delayMilliseconds;
    m_maxDelayMilliseconds = maxDelayMilliseconds;
    m_retries = 0;
    m_pow = 1;
}

public Task Delay()
{
    if (m_retries == m_maxRetries)
    {
        throw new TimeoutException("Max retry attempts exceeded.");
    }
    ++m_retries;
    if (m_retries < 31)
    {
        m_pow = m_pow << 1; // m_pow = Pow(2, m_retries - 1)
    }
    int delay = Math.Min(m_delayMilliseconds * (m_pow - 1) / 2,
        m_maxDelayMilliseconds);
    return Task.Delay(delay);
}
}

```

在 C# 客户端应用程序中（其它 Web API 客户微服务，一个 ASP.NET MVC 应用，或者甚至为 C# Xamarin 应用程序），使用这些代码都很简单。下面的示例演示了如何使用 HttpClient 类。

```

public async Task<Catalog> GetCatalogItems(int page,int take, int? brand, int? type)
{
    _apiClient = new HttpClient();
    var itemsQs = $"items?pageIndex={page}&pageSize={take}";
    var filterQs = "";

    var catalogUrl =
        $"{_remoteServiceBaseUrl}items{filterQs}?pageIndex={page}&pageSize={take}";
    var dataString = "";
    //
    // Using HttpClient with Retry and Exponential Backoff
    //
    var retry = new RetryWithExponentialBackoff();
    await retry.RunAsync(async () =>
    {
        // work with HttpClient call
        dataString = await _apiClient.GetStringAsync(catalogUrl);
    });

    return JsonConvert.DeserializeObject<Catalog>(dataString);
}

```

但是，此代码仅适用于作为概念验证。下一节将介绍如何使用更为复杂和经过验证的库。

使用基于 Polly 的指数补偿实现 HTTP 调用重试

推荐的使用指数补偿重试的方式是利用更高级的 .NET 库，例如开源的 [Polly](#) 库。

Polly 是一个提供弹性和瞬时故障处理能力的 .NET 库。您可以通过应用 Polly 策略，比如重试、断路器、隔舱隔离、超时和回退，轻松实现这些功能。Polly 支持 .NET 4.x 和 .NET Standard 库 1.0（支持 .NET Core）。

Polly 的重试策略是在 eShopOnContainers 中实现 HTTP 重试中使用的方法。您可以实现一个接口，这样您可以选择注入标准的 HttpClient 或者使用 Polly 的 HttpClient 的弹性版本，具体取决于您想使用的重试策略配置。

下面的示例演示在 eShopOnContainers 中的接口实现。

```
public interface IHttpApiClient
{
    Task<string> GetStringAsync(string uri, string authorizationToken = null,
                              string authorizationMethod = "Bearer");

    Task<HttpResponseMessage> PostAsync<T>(string uri, T item,
                                           string authorizationToken = null, string requestId = null,
                                           string authorizationMethod = "Bearer");

    Task<HttpResponseMessage> DeleteAsync(string uri,
                                          string authorizationToken = null, string requestId = null,
                                          string authorizationMethod = "Bearer");

    // Other methods ...
}
```

如果您不想使用弹性机制，就像在开发或测试更简单的方法时一样，您可以使用标准实现。下面的代码展示了在允许可选地带有身份验证 token 的请求的情况标准下的 HttpClient 实现。

```

public class StandardHttpClient : IHttpClient
{
    private HttpClient _client;
    private ILogger<StandardHttpClient> _logger;

    public StandardHttpClient(ILogger<StandardHttpClient> logger)
    {
        _client = new HttpClient();
        _logger = logger;
    }
    public async Task<string> GetStringAsync(string uri,
                                             string authorizationToken = null,
                                             string authorizationMethod = "Bearer")
    {
        var requestMessage = new HttpRequestMessage(HttpMethod.Get, uri);
        if (authorizationToken != null)
        {
            requestMessage.Headers.Authorization =
                new AuthenticationHeaderValue(authorizationMethod, authorizationToken);
        }

        var response = await _client.SendAsync(requestMessage);
        return await response.Content.ReadAsStringAsync();
    }

    public async Task<HttpResponseMessage> PostAsync<T>(string uri, T item,
                                                         string authorizationToken = null, string requestId = null,
                                                         string authorizationMethod = "Bearer")
    {
        // Rest of the code and other Http methods ...
    }
}

```

有趣的实现是编码另外一个类似的类，但使用 Polly 实现您希望使用的弹性机制，在下面的示例中，是带有指数补偿的重试。

```

public class ResilientHttpClient : IHttpClient
{
    private HttpClient _client;
    private PolicyWrap _policyWrapper;
    private ILogger<ResilientHttpClient> _logger;

    public ResilientHttpClient(Policy[] policies,
                               ILogger<ResilientHttpClient> logger)
    {
        _client = new HttpClient();
        _logger = logger;

        // Add Policies to be applied
        _policyWrapper = Policy.WrapAsync(policies);
    }

    private Task<T> HttpInvoker<T>(Func<Task<T>> action)
    {
        // Executes the action applying all
        // the policies defined in the wrapper
        return _policyWrapper.ExecuteAsync(() => action());
    }

    public Task<string> GetStringAsync(string uri,
                                       string authorizationToken = null,
                                       string authorizationMethod = "Bearer")
    {
        return HttpInvoker(async () =>
        {
            var requestMessage = new HttpRequestMessage(HttpMethod.Get, uri);

            // The Token's related code eliminated for clarity in code snippet

            var response = await _client.SendAsync(requestMessage);
            return await response.Content.ReadAsStringAsync();
        });
    }
    // Other Http methods executed through HttpInvoker so it applies Polly policies
    // ...
}

```

使用 Polly，您可以使用重试次数、指数补偿配置、以及在出现 HTTP 异常（例如记录错误）时要执行的操作来定义重试策略。在此例中，策略配置为在 IoC 容器中注册类型时重试指定的次数。由于指数补偿配置，当代码检测到 HttpRequest 异常时，基于策略是如何配置的，它会在等待一段呈指数增长的时间后重试 Http 请求。

重要的是 HttpInvoker 方法，它在这个工具类中负责调用 HTTP 请求。该方法在内部通过调用 _policyWrapper.ExecuteAsync 执行 HTTP 请求，同时考虑重试策略。在 eShopOnContainers 中，您在 IoC 容器中注册类型时指定 Polly 策略，如下代码来自 [MVC Web 应用的 startup.cs](#) 类。


```

// Startup.cs class
if (Configuration.GetValue<string>("UseResilientHttp") == bool.TrueString)
{
    services.AddTransient<IResilientHttpClientFactory,
        ResilientHttpClientFactory>();

    services.AddSingleton<IHttpClient,
        ResilientHttpClient>(sp =>
            sp.GetService<IResilientHttpClientFactory>().
                CreateResilientHttpClient());
}
else
{
    services.AddSingleton<IHttpClient, StandardHttpClient>();
}

```

注意 `IHttpClient` 对象作为单例而不是瞬态被实例化，这样服务就可以有效使用 TCP 连接，并且不会发生套接字的问题。

但是关于弹性的重要一点是，您在 `CreateResilientHttpClient` 方法中的 `ResilientHttpClientFactory` 中应用了 Polly 的 `WaitAndRetryAsync` 策略，如下述代码所示：

```

public ResilientHttpClient CreateResilientHttpClient()
    => new ResilientHttpClient(CreatePolicies(), _logger);
// Other code

private Policy[] CreatePolicies()
    => new Policy[]
    {
        Policy.Handle<HttpRequestException>()
            .WaitAndRetryAsync(
                // number of retries
                6,
                // exponential backoff
                retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt)),
                // on retry
                (exception, timeSpan, retryCount, context) =>
                {
                    var msg = $"Retry {retryCount} implemented with Pollys
                        RetryPolicy " +
                        $"of {context.PolicyKey} " +
                        $"at {context.ExecutionKey}, " +
                        $"due to: {exception}.";
                    _logger.LogWarning(msg);
                    _logger.LogDebug(msg);
                }
            ),
    }

```

实现断路器模式

如前所述，当您尝试连接到远程服务或资源发生故障时，您可能需要处理需要一段时间才能恢复的错误。处理这种类型的故障可以提高应用程序的稳定性和弹性。

在分布式环境下，对远程资源和服务的调用可能会由于瞬态故障，例如网络连接速度慢和超时，或资源缓慢或暂时不可用而失败。这些故障通常会在很短时间内修复自己，而健壮的云应用程序应该准备好使用类似重试模式这样的策略来处理它们。

但是，也可能是由于无法预料的事件造成的故障，可能需要更长时间才能修复。这些故障可能在严重程度从部分连接丢失到服务完全失败不等。在这些情况下，应用程序不断重试不太可能成功，操作可能毫无意义。相反，应该将应用程序编码为接受操作失败并相应地处理故障。

断路器模式的用途与重试模式不同。重试模式使应用程序能够在期望操作最终成功时执行重试操作。断路器模式可以防止应用程序执行可能失败的操作。应用程序可以组合这两种模式，使用重试模式并通过断路器调用操作。但是，重试逻辑应对断路器所返回的任何异常敏感，如果断开则表明故障不是瞬态的，应放弃重试尝试。

使用 Polly 实现断路器模式

在实现重试时，建议的断路器方式是利用经过验证的 .NET 库，例如 Polly。

eShopOnContainers 应用程序在实现 HTTP 重试时使用了 Polly 的断路器策略。实际上，应用程序将这两种策略应用于 ResilientHttpClient 使用工具类。无论何时使用 ResilientHttpClient 的对象进行 HTTP 请求时（来自 eShopOnContainers），都将同时应用这些策略，但也可以添加其它策略。

在这里，相对于 HTTP 调用重试的代码，唯一新增的是将断路器策略添加到要使用的策略列表中的代码，如下面的代码末端所示。

```

public ResilientHttpClient CreateResilientHttpClient()
    => new ResilientHttpClient(CreatePolicies(), _logger);

private Policy[] CreatePolicies()
    => new Policy[]
    {
        Policy.Handle<HttpRequestException>()
            .WaitAndRetryAsync(
                // number of retries
                6,
                // exponential backoff
                retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt)),
                // on retry
                (exception, timeSpan, retryCount, context) =>
                {
                    var msg = $"Retry {retryCount} implemented with Polly
                        RetryPolicy " +
                        $"of {context.PolicyKey} " +
                        $"at {context.ExecutionKey}, " +
                        $"due to: {exception}.";
                    _logger.LogWarning(msg);
                    _logger.LogDebug(msg);
                }
            ),
        Policy.Handle<HttpRequestException>()
            .CircuitBreakerAsync(
                // number of exceptions before breaking circuit
                5,
                // time circuit opened before retry
                TimeSpan.FromMinutes(1),
                (exception, duration) =>
                {
                    // on circuit opened
                    _logger.LogTrace("Circuit breaker opened");
                }
            ),
        () =>
        {
            // on circuit closed
            _logger.LogTrace("Circuit breaker reset");
        }
    });
}

```

该代码将策略添加到 HTTP 包装器中。该策略定义了当代码检测到指定数量的连续异常（行中的异常）时打开断路器，如在 `exceptionsAllowedBeforeBreaking` 参数中传递的那样（本例中为 5）。当断路器打开时，HTTP 请求不会工作，但异常会抛出。

如果在与执行 HTTP 调用的客户端应用程序或服务不同的环境中部署的特定资源可能存在问题，则还应使用断路器将请求重定向到后备基础架构。这样，如果数据中心的中断仅仅影响到您的后端微服务，而不是客户端应用程序，则客户端应用程序可以被重定向到后备服务。Polly 正在计划一个新的策略来自动化该[故障转移](#)策略。

当然，所有这些特性都是针对在 .NET 代码中管理故障转移的情况，而不是使用带有位置透明性的 Azure 自动为您管理。

从 eShopOnContainers 使用 ResilientHttpClient 工具类

使用 ResilientHttpClient 实用工具类的方法与使用 .NET HttpClient 类的方式类似。在下面的示例中, 从 eShopOnContainers 的 MVC web 应用程序 (OrderController 使用的 OrderingService 代理类) 中, 通过构造函数的 httpClient 参数注入 ResilientHttpClient 对象。然后该对象用于执行 HTTP 请求。

```
public class OrderingService : IOrderingService
{
    private IHttpClient _apiClient;
    private readonly string _remoteServiceBaseUrl;
    private readonly IOptionsSnapshot<AppSettings> _settings;
    private readonly IHttpContextAccessor _httpContextAccessor;

    public OrderingService(IOptionsSnapshot<AppSettings> settings,
        IHttpContextAccessor httpContextAccessor,
        IHttpClient httpClient)
    {
        _remoteServiceBaseUrl = $"{settings.Value.OrderingUrl}/api/v1/orders";
        _settings = settings;
        _httpContextAccessor = httpContextAccessor;
        _apiClient = httpClient;
    }

    async public Task<List<Order>> GetMyOrders(ApplicationUser user)
    {
        var context = _httpContextAccessor.HttpContext;
        var token = await context.Authentication.GetTokenAsync("access_token");

        _apiClient.Inst.DefaultRequestHeaders.Authorization = new
            System.Net.Http.Headers.AuthenticationHeaderValue("Bearer", token);
        var ordersUrl = _remoteServiceBaseUrl;
        var dataString = await _apiClient.GetStringAsync(ordersUrl);
        var response = JsonConvert.DeserializeObject<List<Order>>(dataString);

        return response;
    }
    // Other methods ...

    async public Task CreateOrder(Order order)
    {
        var context = _httpContextAccessor.HttpContext;
        var token = await context.Authentication.GetTokenAsync("access_token");
        _apiClient.Inst.DefaultRequestHeaders.Authorization = new
            System.Net.Http.Headers.AuthenticationHeaderValue("Bearer", token);
        _apiClient.Inst.DefaultRequestHeaders.Add("x-requestid",
            order.RequestId.ToString());

        var ordersUrl = $"{_remoteServiceBaseUrl}/new";
        order.CardTypeId = 1;
        order.CardExpirationApiFormat();
        SetFakeIdToProducts(order);
        var response = await _apiClient.PostAsync(ordersUrl, order);
        response.EnsureSuccessStatusCode();
    }
}}
```

每当使用 `_appClient` 成员对象时，其内部会使用 Polly 策略的包装类 - 重试策略、断路器策略和任何其它可能需要的应用从 Polly 收集的策略集。

在 eShopOnContainers 中测试重试

当您在 Docker 宿主中启动 eShopOnContainers 解决方案的时候，都需要启动多个容器。一些容器的启动和初始化比较慢，比如 SQL Server 容器。特别是您第一次在 Docker 中部署 eShopOnContainers 应用的时候尤其如此，因为需要设置镜像和数据库。某些容器比其它容器启动慢的事实造成其它服务开始抛出 HTTP 异常，甚至在您在 Docker 编排级别设置了容器之间的依赖之后，如前几节所述。在容器之间的 Docker 编排依赖仅仅处于进程级别。容器的入口点进程可能已经启动，但是 SQL Server 可能并没有准备好接受查询。结果可能是一连串的错误，应用程序在尝试使用特定容器时得到异常。

当应用程序部署到云的时候，您可能还会在启动时看到这种类型的错误。在这种情况下，在平衡群集节点上的容器数量时，编排引擎可能会将容器从一个节点或者虚拟机转移到另一个（即启动新实例）。

eShopOnContainers 解决这个问题的方式是使用我们刚刚说明的重试模式。这也是为什么在启动应用程序的时候，您可能得到如下的日志跟踪或警告：

```
"Retry 1 implemented with Polly's RetryPolicy, due to:
System.Net.Http.HttpRequestException: An error occurred while sending the
request. ---> System.Net.Http.CurlException: Couldn't connect to server\n at
System.Net.Http.CurlHandler.ThrowIfCURLError(CURLcode error)\n at [...].
```

在 eShopOnContainers 中测试断路器

有几种方式，您可以打开断路，在 eShopOnContainers 中测试它。

一种方式是在断路器策略中将最低允许的重试次数设置为 1，重新在 Docker 中部署整个解决方案。使用单次重试，很有可能在部署期间 HTTP 请求就将失败，断路器被打开，您将收到错误。

另一种方式是在 购物篮微服务中实现自定义的中间件。启动此中间件后，其将捕获所有 HTTP 请求并返回状态码 500。您可以通过对故障 URI 发出一个 GET 请求来启用中间件，如下所示：

- GET /failing
该请求返回中间件的当前状态。如果中间件启用，则请求返回状态码 500。如果中间件被禁用，这里没有响应。
- GET /failing?enable
该请求启用中间件
- GET /failing?disable
该请求禁用中间件

例如，一旦应用程序运行，您可以通过在任何浏览器中使用如下 URI 来发出请求以启用中间件。注意订单微服务使用的端口是 5103。

http://localhost:5103/failing?enable

您可以使用 URI <http://localhost:5103/failing> 来检查状态，如图 10-4 所示。

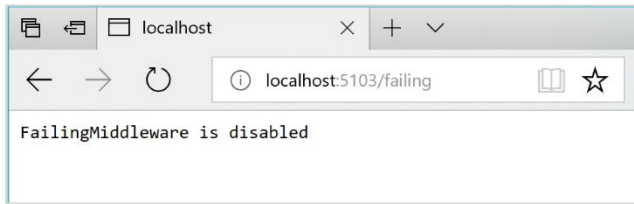


图 10-4. 检查“故障”ASP.NET 中间件的状态，在这个场景下被禁用了。

此时，无论何时通过调用访问它，购物篮微服务都会返回响应码 500。

一旦中间件运行，您可以尝试从 MVC web 应用程序中创建订单。因为请求失败，断路器被打开。

在下面的示例中，您可以看到 MVC web 应用程序在下订单的逻辑中存在一个 catch 块。如果代码捕获到断路器打开的异常，它将向用户显示一条友好消息，告诉它们等待。

```
public class CartController : Controller
{
    //...
    public async Task<IActionResult> Index()
    {
        try
        {
            //... Other code
        }
        catch (BrokenCircuitException)
        {
            // Catch error when Basket.api is in circuit-opened mode
            HandleBrokenCircuitException();
        }
        return View();
    }

    private void HandleBrokenCircuitException()
    {
        TempData["BasketInoperativeMsg"] = "Basket Service is inoperative, please
try later on. (Business Msg Due to Circuit-Breaker)";
    }
}
```

作为总结，重试策略尝试多次以发出 HTTP 请求并获取 HTTP 错误。当尝试次数达到为断路器设置的最大值（本例中为 5）时，应用程序抛出一个 BrokenCircuitException 异常。结果是一条友好的消息，如图 10-5 所示。

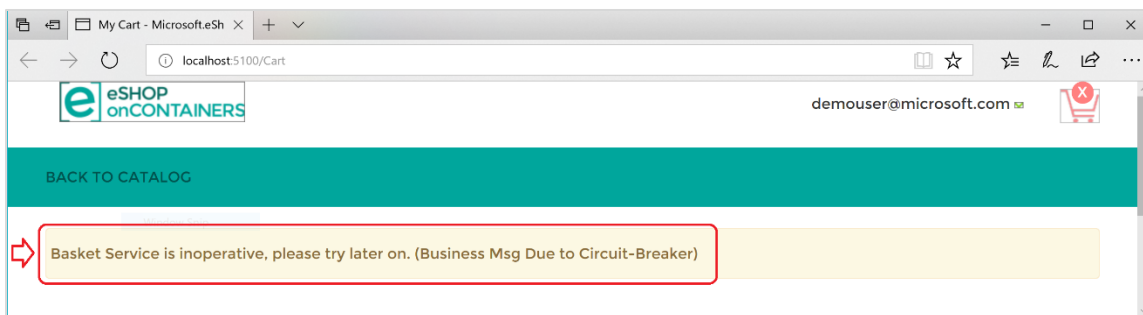


图 10-5. 断路器向 UI 返回错误

您可以实现不同逻辑来决定何时打开断路器。或者在存在后备数据中心或者冗余后端系统的时候，您可以对另外的后端微服务尝试 HTTP 请求。

最后，CircuitBreakerPolicy 的另一种可能是使用 Isolate（强制打开并保持打开断路器）和 Reset（再次关闭）。这可用于构建一个实用工具 HTTP 端点，它直接调用 Isolate 和 Reset 策略。这样的 HTTP 端点也可以在生产中用于临时隔离下游系统，比如当您升级它时。或者它可以手工触发，以保护您怀疑有故障的下游系统。

在重试策略中添加抖动策略

常规的重试策略可以在高并发、可伸缩和高争用的情况下影响系统。为克服在部分停机时来自大量客户端的类似重试的峰值，一个很好的方案是在重试算法/策略中添加抖动策略。这可以通过在指数补偿中增加随机性来改善端对端系统的整体性能。当问题发生时，这可以分散峰值。使用 Polly 时，实现抖动的代码可能类似如下示例。

```
Random jitterer = new Random();
Policy
    .Handle<HttpResponseException>() // etc
    .WaitAndRetry(5, // exponential back-off plus some jitter
        retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt))
            + TimeSpan.FromMilliseconds(jitterer.Next(0, 100))
    );
```

其他资源

- **重试模式**
<https://docs.microsoft.com/azure/architecture/patterns/retry>
弹性连接 (Entity Framework Core)
<https://docs.microsoft.com/ef/core/miscellaneous/connection-resiliency>
- **Polly** (.NET 弹性和瞬态故障处理库)
<https://github.com/App-vNext/Polly>
- **断路器模式**
<https://docs.microsoft.com/azure/architecture/patterns/circuit-breaker>

- **Marc Brooker. Jitter: 随机性让事情变得更好**
<https://brooker.co.za/blog/2015/03/21/backoff.html>

运行状况监测

运行状况监测可以提供关于您的应用程序和微服务的近乎实时的状态信息。运行状况监测对于运营微服务的多个方面至关重要，并且在协调分阶段应用程序升级时尤为重要，如后所述。

基于微服务的应用程序经常使用心跳或运行状况检查来启用其性能监视器、调度器和编排引擎，以跟踪众多的服务。如果服务不能按需或按计划发送某类“我还活着”的信号，则在部署升级时，您的应用程序可能会面临风险，或者只是太晚检测到故障，无法阻止可能导致主要宕机的级联故障。

在典型的模式中，服务发送关于其状态的报告，并将信息聚合起来，以提供应用程序运行状况的整体视图。如果您使用编排引擎，您可以为您的编排者群集提供运行状况信息以便群集可以采取相应行动。如果您为您的应用程序定制高质量的运行状况报告，您就可以更为轻松地检测并修复运行中的应用程序问题。

在 ASP.NET Core 服务中实现运行状况检查

在开发 ASP.NET Core 微服务或 web 应用程序的时候，您可以使用名为 HealthChecks 的来自 ASP.NET 团队的库（早期发布版在 GitHub 可用）。

该库易于使用，并提供一些可以让您验证应用程序所需的任何特定外部资源（如 SQL Server 数据库或远程 API）是否正常工作的特性。当您使用该库时，您还可以定义什么意味着资源是正常的，如后所述。

为使用此库，您首先需要在微服务中使用该库。其次，您需要一个前端应用来查询运行状况报告。前端应用可以是一个定制的报告应用，或者是编排引擎本身，它可以对运行状况做出相应的反应。

在后端 ASP.NET 微服务中使用 HealthChecks 库

您可以在 eShopOnContainers 示例应用程序中看到 HealthChecks 库是如何使用的。首先，您需要定义什么构成了每个微服务的运行状况。在示例应用程序中，如果微服务的 API 可以通过 HTTP 访问，并且其相关的 SQL Server 数据库也可用，则微服务是正常的。

未来，您将能够将通过 NuGet 包来安装 HealthChecks 库。但直到本书创作时，您还是需要下载并编译代码作为应用程序的一部分。克隆 <https://github.com/dotnet-architecture/HealthChecks> 中的代码，复制如下文件夹到您的解决方案中。

```
src/common
src/Microsoft.AspNetCore.HealthChecks
src/Microsoft.Extensions.HealthChecks
src/Microsoft.Extensions.HealthChecks.SqlServer
```


您还可以使用类似 Azure (Microsoft.Extensions.HealthChecks.AzureStorage) 的其它检查，但由于此版本的 eShopOnContainers 对 Azure 没有任何依赖，因此您不需要它。您不需要 ASP.NET 的运行状况检查，因为 eShopOnContainers 基于 ASP.NET Core。

图 10-6 展示了在 Visual Studio 中的 healthChecks 库，作为编译块可以被任何微服务所使用。

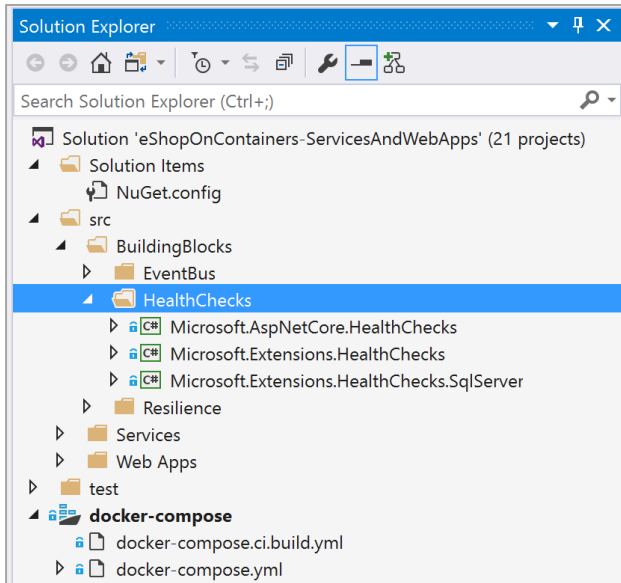


图 10-6. Visual Studio 解决方案中的 ASP.NET Core HealthChecks 库源码

如前所述，在每个微服务项目中要做的第一件事是添加对三个 HealthChecks 库的引用。然后，添加要在该微服务中执行的运行状况检查操作。这些操作基本上依赖于其它微服务 (HttpRequestCheck) 或数据库 (当前 SqlCheck* 用于 SQL Server 数据库)。在每个 ASP.NET 微服务或 ASP.NET web 应用程序的启动类中添加操作。

每个服务或 web 应用程序都应通过将其所有的 HTTP 或数据库依赖作为一个方法 AddHealthCheck 进行配置。例如，来自 eShopOnContainers 的 MVC web 应用程序依赖于许多服务，因此，有多个 AddCheck 方法添加到运行状况检查中。

例如，在下面的代码中，您可以看到 catalog 微服务添加了一个对 SQL Server 数据库的依赖项。

```
// Startup.cs from Catalog.api microservice
//
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // Add framework services
        services.AddHealthChecks(checks =>
        {
            checks.AddSqlCheck("CatalogDb", Configuration["ConnectionString"]);
        });
        // Other services
    }
}
```

```
}  
}
```

但是，eShopOnContainers 的 MVC web 应用程序对其它的微服务有多个依赖项。因此，它为每个微服务调用一个 AddUrlCheck 方法，如下示例所示。

```
// Startup.cs from the MVC web app  
public class Startup  
{  
    public void ConfigureServices(IServiceCollection services)  
    {  
        services.AddMvc();  
        services.Configure<AppSettings>(Configuration);  
  
        services.AddHealthChecks(checks =>  
        {  
            checks.AddUrlCheck(Configuration["CatalogUrl"]);  
            checks.AddUrlCheck(Configuration["OrderingUrl"]);  
            checks.AddUrlCheck(Configuration["BasketUrl"]);  
            checks.AddUrlCheck(Configuration["IdentityUrl"]);  
        });  
    }  
}
```

因此，直到微服务所有的检查都是正常的，微服务才会提供一个“正常”的状态。

如果微服务不依赖于其它服务或者 SQL Server，您可以仅仅添加一个 Healthy(“OK”) 检查。下面的代码来自 eShopOnContainers 的 **basket.api** 微服务。（购物篮微服务使用 Redis 缓存，但是该库不包含 Redis 运行状况检查提供器。）

```
services.AddHealthChecks(checks =>  
{  
    checks.AddValueTaskCheck("HTTP Endpoint", () => new  
        ValueTask<IHealthCheckResult>(HealthCheckResult.Healthy("Ok")));  
});
```

要使服务或 web 应用程序公开运行状况检查端点，必须启用 UserHealthChecks ([url_for_health_checks]) 扩展方法。此方法位于 ASP.NET Core 或 web 应用程序的 Main 方法中的 WebHostBuilder 级别，如下面代码所示紧随 UseKestrel。

```
namespace Microsoft.eShopOnContainers.WebMVC  
{  
    public class Program  
    {  
        public static void Main(string[] args)  
        {  
            var host = new WebHostBuilder()  
                .UseKestrel()  
                .UseHealthChecks("/hc")
```

```
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseIISIntegration()
        .UseStartup<Startup>()
        .Build();
    host.Run();
}
}
```

该过程的工作方式如下：每个微服务公开端点 /hc。该端点由 HealthChecks 库 ASP.NET Core 中间件创建。当调用该端点时，它运行在 Startup 类中的 AddHealthChecks 方法中配置的所有运行状况检查。

UseHealthChecks 方法需要一个端口或路径。该端口或路径是用于检查服务运行状态的端点。例如，目录微服务使用路径为 /hc。

缓存运行状况检查响应

由于您不希望在服务中导致拒绝服务（DoS），或者只是不希望由于频繁检查资源而影响服务性能，可以缓存检查结果并为每个运行状况配置缓存的持续时间。

默认情况下，缓存的持续时间内部设置为 5 分钟，但您可以更改每个运行状况检查的缓存时间，如下代码所示。

```
checks.AddUrlCheck(Configuration["CatalogUrl"], 1); // 1 min as cache duration
```

查询微服务以报告其运行状况

如果您已经按此处所述配置了运行状况检查，则一旦微服务运行在 Docker 中，您可以直接从浏览器中检查其是否正常。（这需要您将容器的端口映射到宿主机，以便您可以直接通过 localhost 或者外部 Docker Host IP 访问）。图 10-7 展示了浏览器中的请求和相应的响应。

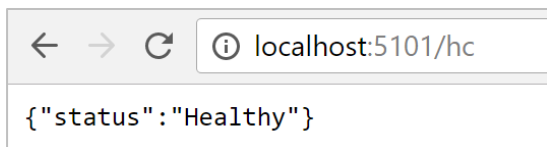


图 10-7. 通过浏览器检查单个服务的运行状况

在这个测试中，您可以看到 catalog.api 微服务（运行在端口 5101）是正常的。返回 HTTP 状态码 200 以及 JSON 格式的状态信息。它还意味着在内部还检查了其 SQL Server 数据库依赖关系的运行状况，并报告其同样正常。

使用 Watchdog

Watchdog 是一种独立的服务，可以监视服务的运行状况和负载，并通过查询前面介绍的 HealthChecks 库来报告微服务的运行状况。这可以帮助防止基于单个服务视图而无法检测到的错误。在 Watchdog 中也适合在没有用户交互的情况下，放置对已知条件进行补救操作的代码。

eShopOnContainers 示例中包含一个 Web 页面显示运行状况监测报告。如图 10-8 所示。这是您可以使用的最简单的 Watchdog，因为它所做的事情只是显示了 eShopOnContainers 中微服务和 web 应用程序的状态。通常，Watchdog 在检测到不正常状态时也会采取行动。

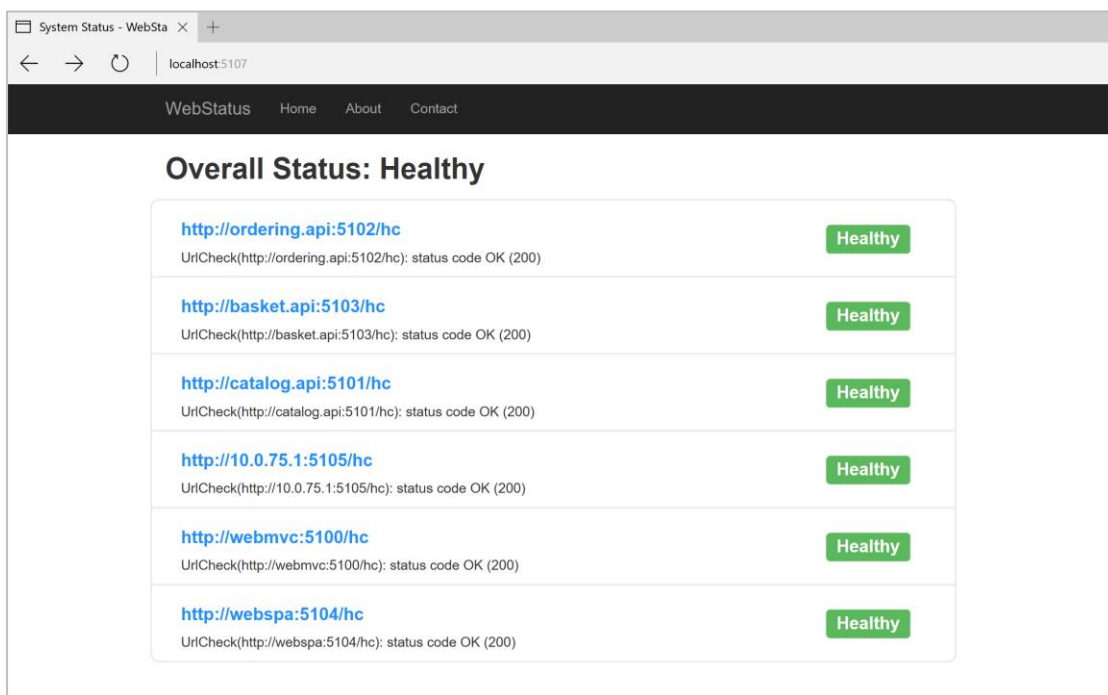


图 10-8. 在 eShopOnContainers 中的示例运行状况监测报告

总之，ASP.NET Core 的中间件 HealthChecks 库为每个微服务提供了一个单一的运行状况检查端点。其将执行定义的所有运行状况检查，并根据所有这些检查返回一个整体运行状况状态。

HealthChecks 库可通过对未来外部资源的新的运行状况检查进行扩展。例如，我们预计，将来该库将对 Redis 缓存和其它数据库进行运行状况检查。该库允许由多个服务或应用程序依赖项进行运行状况报告，然后您可以根据这些运行状况检查执行操作。

使用编排引擎时的运行状况检查

要监视您的微服务的可用性，编排引擎，类似 Docker Swarm、Kubernetes 和 Service Fabric 定期通过发送请求来测试微服务执行运行状况检查。当编排引擎检测到某个服务/容器不再正常时，它将停止对该实例的路由请求。通常还会创建该容器的一个新实例。

例如，多数编排引擎可以使用运行状况检查来管理零停机部署。只有当服务/容器的状态变为正常时，编排引擎才会开始将通信路由到服务/容器实例。

当编排引擎执行应用程序升级时，运行状况监视尤为重要。有些编排引擎（如 Azure Service Fabric）分阶段更新服务，例如，每个应用程序升级可能更新 1/5 的群集。同时升级的节点集合称为升级域。升级每个升级域并可供用户使用之后，升级域必须在部署转移到下一个升级域之前通过运行状况检查。

服务运行状况的另一个方面是源自服务的报告指标。这是某些编排引擎运行状况模型的高级功能，比如 Service Fabric。在使用编排引擎时，度量指标非常重要，因为它们用于平衡资源的使用。指标也可以是系统运行状况的指示器。例如，您可能有一个具有许多微服务的应用程序，每个实例都报告每秒请求（RPS）量。如果一个服务正在使用比其它服务更多的资源（内存、处理器等），则编排引擎可以在群集中移动服务实例，以便尝试保持均衡的资源利用率。

请注意，如果您使用的是 Azure Service Fabric，它将提供自己的[运行状况监视模型](#)，它比简单的运行状况检查更为高级。

高级监视：可视化、分析和警报

监视的最后一部分是可视化事件流，报告服务性能，并在检测到问题时发出警报。您可以对监视的这个方面使用不同的解决方案。

您可以使用简单的定制应用程序来展示服务的状态。就像我们在解释 [ASP.NET Core HealthChecks](#) 时显示的自定义页面一样。或者，您可以使用更为高级的工具 - 如 Azure Application Insight 和 Operations Management Suite，来根据事件流发出警报。

最后，如果您存储了所有事件流，则可以使用 Microsoft Power BI 或者第三方解决方案（如 Kibana 或者 Splunk）来可视化数据。

其他资源

- **ASP.NET Core HealthChecks**（早期发布版本）
<https://github.com/aspnet/HealthChecks/>
- **Service Fabric 运行状况监视简介**
<https://docs.microsoft.com/azure/service-fabric/service-fabric-health-introduction>
- **Azure Application Insights**
<https://azure.microsoft.com/services/application-insights/>
- **Microsoft Operations Management Suite**
<https://www.microsoft.com/cloud-platform/operations-management-suite>

保护 .NET 微服务和 Web 应用程序

在 .NET 微服务和 Web 应用程序中实现验证

通常，服务所公开的资源 and API 必须仅限于特定受信任的用户和客户端。进行此类 API 级别决策的第一步是身份验证。验证是可靠地确定用户身份的过程。

在微服务场景中，身份验证通常统一处理。如果您使用 API 网关，则网关是进行验证的好位置，如图 11-1 所示。如果使用这种方式，请确保无法直接（不通过 API 网关）访问单个的微服务，除非有其它的安全机制来验证消息是否来自网关。

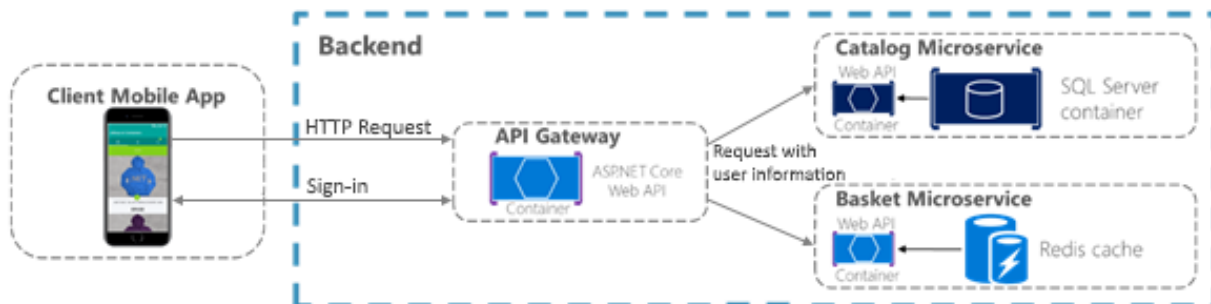


图 11-1. 使用 API 网关中心化验证

如果服务可以被直接访问，类似 Azure 活动目录的验证服务或者类似安全令牌服务（STS）的专用验证微服务可以用来对用户进行身份验证。信任决策是在具有安全令牌或 Cookie 的服务之间共享的。（如果需要，可以在应用程序之间共享带有[数据保护服务](#)的 ASP.NET Core）。此模式如图 11-2 所示。

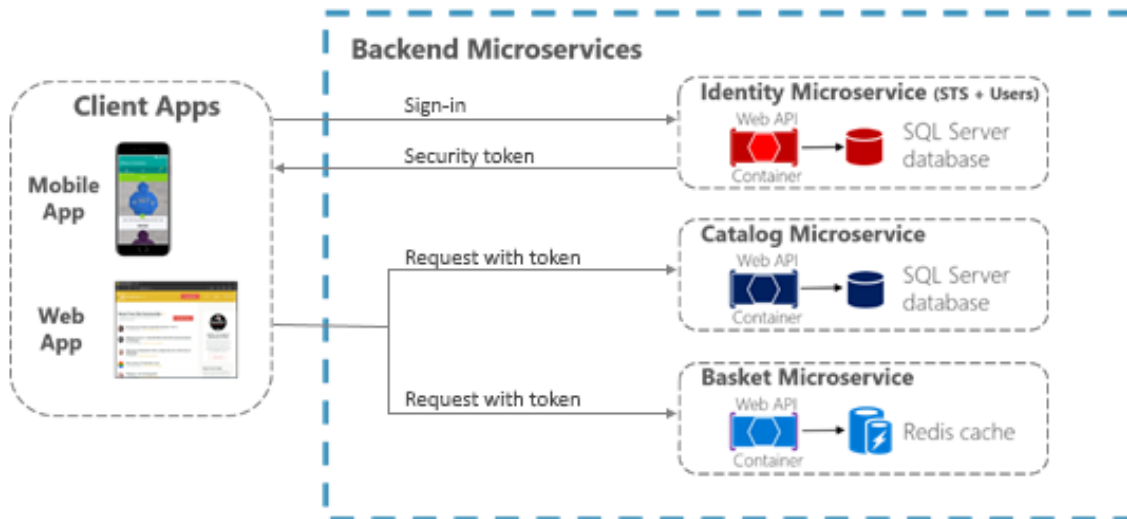


图 11-2. 使用共享授权令牌的身份验证微服务验证

使用 ASP.NET Core Identity 验证

ASP.NET Core 中用于验证用户的主要机制是 ASP.NET Core Identity 成员系统。[ASP.NET Core Identity](#) 在开发人员配置的数据存储区中存储用户信息（包括登录信息、角色和声明）。通常，ASP.NET Core Identity 数据存储由 Microsoft.AspNetCore.Identity.EntityFrameworkCore 包提供的 Entity Framework 存储。但是，可以使用自定义存储或其它的第三方包在 Azure 表存储、DocumentDB 或其它位置存储标识信息。

下面的代码取自使用选择了独立用户账户身份验证的 ASP.NET Core Web 应用程序项目模板。它显示了如何在启动配置方法中使用 Entity Framework Core 配置 ASP.NET Core 的方法。

```
services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();
```

一旦配置了 ASP.NET Core Identity，您就可以在服务的 Startup.Configure 方法中通过调用 app.UserIdentity 来启用它。

使用 ASP.NET Core Identity 可以实现以下几种情况：

- 使用 UserManager 类型创建新用户信息（UserManager.CreateAsync）
- 使用 SignInManager 类型来验证用户。您可以使用 signInManager.SignInAsync 直接登录，或者，signInManager.PasswordSignInAsync 来确认用户的口令是正确的，然后登录。
- 基于存储在 Cookie 中的信息（由 ASP.NET Core 中间件读取）识别用户，以便在浏览器的后继请求中包含登录用户的身份和声明。

ASP.NET Identity Core 还支持[双因子验证](#)。

对于使用本地用户数据存储，并在请求之间使用 Cookie 保持身份的身份验证场景（如典型的 MVC Web 应用程序），ASP.NET Core Identity 是一个推荐的解决方案。

使用外部提供器验证

ASP.NET Core 还支持使用[外部验证提供器](#)通过 [OAuth 2.0](#) 流进行登录。这意味着用户可以使用来自 Microsoft、Google、Facebook 或 Twitter 等供应商的现有身份验证过程登录，并将这些身份与 ASP.NET Core Identity 应用程序中的身份关联起来。

若要使用外部身份验证，请在应用程序的 HTTP 请求处理管道中包括适当的身份验证中间件。该中间件负责处理从身份验证提供器返回的 URI 路由请求，捕获身份信息并通过 SignInManager.GetExternalLoginInfo 方法使其可用。

下面列出了流行的外部身份验证提供者及其相关的 NuGet 包：

提供者	包
Microsoft	Microsoft.AspNetCore.Authentication.MicrosoftAccount
Google	Microsoft.AspNetCore.Authentication.Google
Facebook	Microsoft.AspNetCore.Authentication.Facebook
Twitter	Microsoft.AspNetCore.Authentication.Twitter

在所有情况下，中间件都是通过调用类似于 `app.Use{ExternalProvider}Authentication` 的注册方法注册的。这些注册方法接收一个提供者所需的选项对象，其中包含应用程序的 ID 和机密信息（例如口令）。外部身份验证提供者要求注册应用程序（如 [ASP.NET Core 文档所述](#)），这样就可以通知用户请求访问其身份的应用程序。

一旦中间件在 `Startup.Configure` 中注册，您就可以提示用户从任何控制器的 Action 中登录。为此，您需要创建一个 `AuthenticationProperties` 对象，其中包括身份验证提供者的名称和重定向 URL。然后返回一个通过 `AuthenticationProperties` 对象质询响应。下面展示了示例代码：

```
var properties = _signInManager.ConfigureExternalAuthenticationProperties(provider,
                                                                    returnUrl);
return Challenge(properties, provider);
```

参数 `redirectUrl` 包括一旦用户被验证之后外部提供者应重定向的 URL 地址。该 URL 应表示将根据外部标识信息对用户进行签名的 Action，如下简化的示例所示：

```
// Sign in the user with this external login provider if the user
// already has a login.
var result = await _signInManager.ExternalLoginSignInAsync(info.LoginProvider,
info.ProviderKey, isPersistent: false);
if (result.Succeeded)
{
    return RedirectToLocal(returnUrl);
}
else
{
    ApplicationUser newUser = new ApplicationUser
    {
        // The user object can be constructed with claims from the
        // external authentication provider, combined with information
        // supplied by the user after they have authenticated with
        // the external provider.
        UserName = info.Principal.FindFirstValue(ClaimTypes.Name),
        Email = info.Principal.FindFirstValue(ClaimTypes.Email)
    };

    var identityResult = await _userManager.CreateAsync(newUser);
    if (identityResult.Succeeded)
```



```
{
    identityResult = await _userManager.AddLoginAsync(newUser, info);
    if (identityResult.Succeeded)
    {
        await _signInManager.SignInAsync(newUser, isPersistent: false);
    }

    return RedirectToLocal(returnUrl);
}
}
```

如果您在 Visual Studio 中创建 ASP.NET Core web 应用程序项目的时候，选择在**独立用户账户**验证选项，在项目中所有使用外部提供器所需的代码都已经准备好了，如图 11-3 所示。

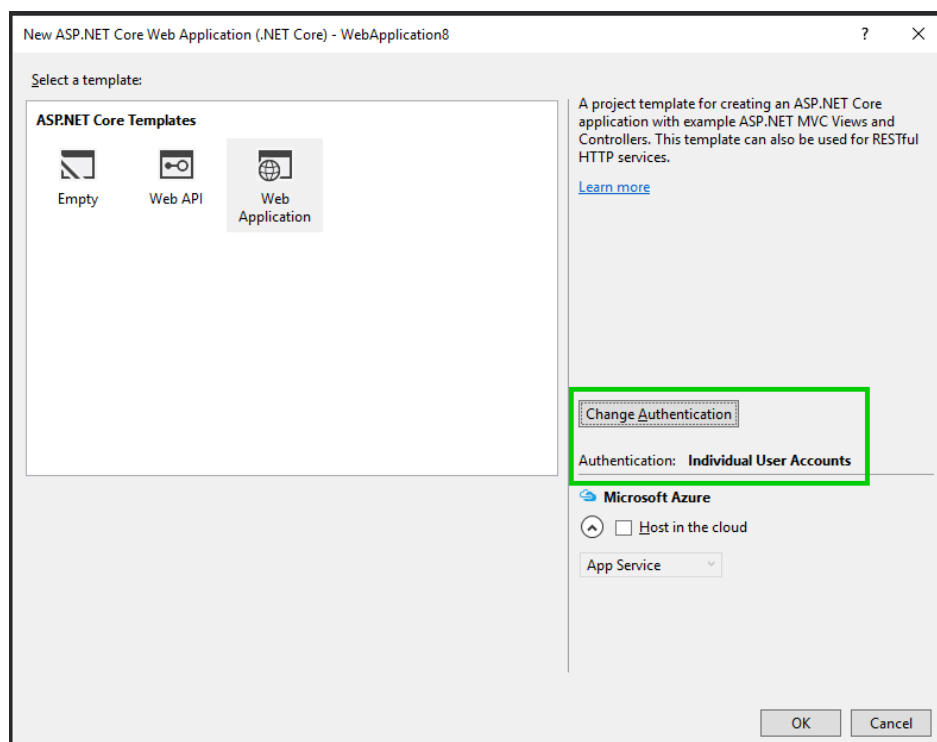


图 11-3. 在创建 Web 应用程序时选择使用外部验证选项

除了前面列出的外部验证提供器之外，还有第三方的软件包为使用更多的外部身份验证提供中间件。有关列表，请参阅 GitHub 上的 [AspNet.Security.OAuth.Providers](https://github.com/aspnet/Security.OAuth.Providers) 仓库。

当然，也可以创建您自己的外部验证中间件。

使用 bearer token 验证

使用 ASP.NET Core Identity 加上外部验证提供器适用于许多 Web 应用程序场景，这些场景中 Cookie 中存储用户信息是适当的。但在其它场景下，Cookie 就不是保持和传输数据的自然方式。

例如，在 ASP.NET Core Web API 中，公开了可能由单页应用程序（SPA）、本地客户端，甚至是其它 Web API 访问的 RESTful 风格的端点，因此通常使用 bearer token 验证。此类应用程序不再使用 Cookie，但是很容易获取 Bearer Token，并将其包含在后继请求的 authorization 头中。为了启用令牌验证，ASP.NET Core 支持使用 [OAuth 2.0](#) 和 [OpenID Connect](#) 的几个选项。

使用 OpenID Connect 或 OAuth 2.0 Identity 提供器验证

如果用户信息存储在 Azure 活动目录中或者支持 OpenID Connect 或 OAuth 2.0 的其它 Identity 方案中，您可以使用 Microsoft.AspNetCore.Authentication.OpenIdConnect 包的 OpenID Connect 流进行验证。例如，基于 [Azure 活动目录验证](#)，ASP.NET Core web 应用程序可以使用该包中的中间件，如下例所示：

```
// Configure the OWIN pipeline to use OpenID Connect auth
app.UseOpenIdConnectAuthentication(new OpenIdConnectOptions
{
    ClientId = Configuration["AzureAD:ClientId"],
    Authority = String.Format(Configuration["AzureAd:AadInstance"],
                             Configuration["AzureAd:Tenant"]),
    ResponseType = OpenIdConnectResponseType.IdToken,
    PostLogoutRedirectUri = Configuration["AzureAd:PostLogoutRedirectUri"]
});
```

配置值来自应用程序注册为 [Azure 活动目录客户端](#) 时创建的 Azure 活动目录值。如果多个微服务都需要通过 Azure 活动目录验证用户，单个的客户端 ID 可以在应用程序的多个微服务之间共享。

请注意，当您使用此 workflow 时，不再需要 ASP.NET Core Identity 中间件，因为所有的用户信息存储和验证都由 Azure 活动目录管理。

从 ASP.NET Core 服务中颁发安全令牌

如果您希望为本地的 ASP.NET Core Identity 用户颁发安全令牌，而不是使用外部 Identity 提供器，您可以利用一些好的第三方库。

[IdentityServer4](#) 和 [OpenIddict](#) 是 OpenID Connect 提供器，可以轻松与 ASP.NET Core Identity 集成，使您从 ASP.NET Core 服务颁发安全令牌。[IdentityServer4 文档](#) 有深入的使用说明。但是，使用 [IdentityServer4](#) 来颁发令牌的基本步骤如下：

1. 在 Startup.Configure 方法中调用 app.UseIdentityServer 来添加 IdentityServer4 到应用程序的 HTTP 请求管理管道中。这使得库处理对类似 /connect/token 的 OpenID Connect 和 OAuth2 端点请求。
2. 通过调用 services.AddIdentityServer 方法在 Startup.ConfigureServices 方法中配置 IdentityServer4。
3. 提供以下数据配置 Identity 服务器

- 用于签名的 [credentials](#)
- 用户可能请求访问的 [Identity 和 API 资源](#)
 - API 资源指的是用户可以使用访问令牌访问的受到保护的数据或功能。API 资源的一个示例是需要授权的 Web API（或 API 集）
 - Identity 资源指的是向客户端提供的用于标识用户的信息（声明）。这些声明可能包括用户名、电子邮件地址等。
- 为了请求令牌将要连接的[客户端](#)。
- 用户信息的存储机制，如 [ASP.NET Core Identity](#) 或者替代方法。

当您指定要使用 IdentityServer4 的客户端和资源时，可以将 IEnumerable<T> 的适当类型集合传递给使用内存中或资源存储的方法。或者，对于更为复杂的场景，您可以通过依赖注入提供客户端或资源提供类型。

IdentityServer4 使用的内存中资源，和由自定义 IClientStore 类型提供的客户端配置，可能类似于如下示例：

```
// Add IdentityServer services
services.AddSingleton<IClientStore, CustomClientStore>();

services.AddIdentityServer()
    .AddSigningCredential("CN=sts")
    .AddInMemoryApiResources(MyApiResourceProvider.GetAllResources())
    .AddAspNetIdentity<ApplicationUser>();
```

使用安全令牌

基于 OpenID Connect 端点或颁发您自己的安全令牌验证覆盖了一些场景。但是，如果服务仅仅需要限制那些拥有由不同服务提供的有效安全令牌的用户，又该如何处理呢？

对于此类场景，Microsoft.AspNetCore.Authentication.JwtBearer 包提供了处理 JWT 令牌的验证中间件。JWT 代表“[JSON Web Token](#)”，是一种常用的安全令牌格式（由 RFC 7519 定义），用于通信安全声明。关于如何通过中间件使用这些令牌的简单示例，可能类似如下示例。代码必须在调用 ASP.NET Core MVC 中间件（app.UseMvc）之前调用。

```
app.UseJwtBearerAuthentication(new JwtBearerOptions()
{
    Audience = "http://localhost:5001/",
    Authority = "http://localhost:5000/",
    AutomaticAuthenticate = true
});
```

参数如下所示：

- Audience 表示传入令牌的接收者或令牌访问授权的资源。如果参数中指定的值不匹配令牌中的参数，令牌将被拒绝。
- Authority 为颁发令牌的验证服务器地址。JWT Bearer 验证中间件使用这个 URI 来获取可用于校验令牌签名的公钥。中间件还要确认令牌中的参数匹配此 URI。
- AutomaticAuthenticate 是一个标识令牌定义的用户是否自动登录的布尔值。

另一个参数 RequireHttpsMetadata，在此示例中未被使用。其用于测试目的，将此参数设置为 false，可以在没有证书的环境中进行测试。在实际部署中，JWT bearer token 应当仅通过 HTTPS 传输。

中间件就绪之后，JWT 令牌被自动从 Authorization 请求头中提取。然后反序列化，校验（使用在 Audience 和 Authority 参数中的值），并存储为稍后由 MVC 的 Action 或者授权过滤器引用的用户信息。

JWT Bearer 验证中间件还支持更为高级的场景，比如在 Authority 不可用时，使用本地的证书来验证令牌。对于这种场景，您可以在 JwtBearerOptions 对象上指定一个 TokenValidationParameters 对象。

其他资源

- **在应用程序之间共享 Cookie**
<https://docs.microsoft.com/aspnet/core/security/data-protection/compatibility/cookie-sharing#sharing-authentication-cookies-between-applications>
- **Identity 简介**
<https://docs.microsoft.com/aspnet/core/security/authentication/identity>
- **Rick Anderson. 使用短信的双因子验证**
<https://docs.microsoft.com/aspnet/core/security/authentication/2fa>
- **使用 Facebook, Google 和其他外部提供者实现验证**
<https://docs.microsoft.com/aspnet/core/security/authentication/social/>
- **Michell Anicas. OAuth 2 简介**
<https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2>
- **AspNet.Security.OAuth.Providers**（ASP.NET OAuth 提供器的 GitHub 仓库）
<https://github.com/aspnet-contrib/AspNet.Security.OAuth.Providers/tree/dev/src>
- **Danny Strockis. 将 Azure AD 整合到 ASP.NET Core Web 应用**
<https://azure.microsoft.com/resources/samples/active-directory-dotnet-webapp-openidconnect-aspnetcore/>
- **IdentityServer4. 官方文档**
<https://identityserver4.readthedocs.io/en/release/>

.NET 微服务和 Web 应用程序的授权

在验证之后，ASP.NET Core Web API 需要授权访问。此过程允许服务使得 API 对某些验证用户可用，但并不是所有验证用户。[授权](#)可以基于用户的角色或基于自定义的策略，这可能包括检查声明或其它方式。

限制对 ASP.NET Core 路由的访问与将 Authorize 特性应用到 Action 方法同样简单（或者控制器，如果所有控制器中的 Action 都要求授权），如下代码所示：

```
public class AccountController : Controller
{
    public ActionResult Login()
    {
    }

    [Authorize]
    public ActionResult Logout()
    {
    }
}
```

默认情况下，添加没有参数的 Authorize 特性将限制对控制器或 Action 限制为已验证用户。如果进一步限制仅对特定的用户 API 可用，可以扩展该特性以指定必需的角色或策略。

实现基于角色的授权

ASP.NET Core Identity 具有内置的角色概念。除了用户之外，ASP.NET Core Identity 存储用于应用程序的不同角色的信息，并跟踪哪些用户被分配了哪些角色。这些分配可以通过类型 RoleManager（可以更新持续存储中的角色）和 UserManager（可以将用户从角色中分配或取消分配）以编程方式进行更新。

如果您使用 JWT Bearer Token 验证，ASP.NET Core 的 JWT Bearer 验证中间件将基于从令牌中发现的角色声明来填充用户的角色。若要限制对 MVC 中 Action 或者控制器的访问仅为特定角色中的用户，可以在 Authorize 特性中包含 Roles 参数，如下所示。

```
[Authorize(Roles = "Administrator, PowerUser")]
public class ControlPanelController : Controller
{
    public ActionResult SetTime()
    {
    }

    [Authorize(Roles = "Administrator")]
    public ActionResult ShutDown()
    {
    }
}
```

在示例中，仅仅 Administrator 和 PowerUser 用户可以访问 ControlPanelController 中的 API（比如 SetTime 这个 Action）。ShutDown API 则进一步限制为只有 Administrator 角色中的用户方可访问。

对于必需拥有多个不同角色的用户，您可以使用多个 Authorize 特性，如下示例所示：

```
[Authorize(Roles = "Administrator, PowerUser")]
[Authorize(Roles = "RemoteEmployee ")]
[Authorize(Policy = "CustomPolicy")]
public ActionResult API1 ()
{
}
```

在这个示例中，为调用这个 API1，用户必须

- 在角色 Administrator 或 PowerUser 中，并且
- 在角色 RemoteEmployee 中，并且
- 满足 CustomPolicy 授权的自定义处理程序。

实现基于策略的授权

还可以使用[授权策略](#)编写自定义授权规则。在本节中，我们提供一个概览。更多详细信息可以通过在线的[ASP.NET 授权研讨会](#)发现。

自定义的授权策略通过在 Startup.ConfigureServices 方法使用 service.AddAuthorization 方法注册。该方法采用一个参数为 AuthorizationOptions 的委托。

```
services.AddAuthorization(options =>
{
    options.AddPolicy("AdministratorsOnly", policy =>
        policy.RequireRole("Administrator"));
    options.AddPolicy("EmployeesOnly", policy =>
        policy.RequireClaim("EmployeeNumber"));
    options.AddPolicy("Over21", policy =>
        policy.Requirements.Add(new MinimumAgeRequirement(21)));
});
```

如示例所示，策略可以与不同类型的需求相关联。注册策略后，可以将策略名称作为 Authorize 特性的 Policy 参数应用于 Action 或者 Controller（例如，[Authorize(Policy="EmployeesOnly")]），策略可以拥有多个必需条件，而不是一个（如示例所示）。

在前面的示例中，第一个 AddPolicy 调用只是通过角色授权的一种方式。如果 [Authorize(Policy="AdministratorsOnly")] 应用于某个 API，将只有被赋予 Administrator 角色的用户可以访问。

第二个 AddPolicy 调用演示了一种简单的方式，要求用户提供特定的声明。RequireClaim 方法还可选地为该声明获取预期的值。如果指定了值，只有既提供了正确类型的声明，又拥有指定的值的用户，才能满足要求。如果您使用 JWT Bearer 验证中间件，所有的 JWT 属性将作为用户声明可用。

最有趣的策略是第三个 AddPolicy 方法，因为它使用了自定义的验证 Requirement。通过使用自定义授权 Requirement，您可以对如何执行授权有很大的控制权。要执行此操作，您必须实现如下类型：

- 实现 IAuthorizationRequirement 的 Requirements 类型，包含特定 Requirement 的详细信息的字段。在本示例中，这里有一个 Age 字段用于 MinimumAgeRequirement 类型。
- 实现 AuthorizationHandler<T> 的处理器，其中 T 是处理程序可以满足的 IAuthorizationRequirement 类型。该处理程序必须实现 HandleRequirementAsync 方法。其检查包含用户信息的上下文是否满足要求。
- 如果用户满足要求，则调用 context.Succeed 来表示用户已被授权。如果用户有多种方式可以满足授权要求，则可以创建多个处理程序。

除了使用 AddPolicy 注册自定义的策略要求外，还需要通过依赖注入注册自定义 Requirement 处理程序 (services.AddTransient<IAuthorizationHandler, MinimumAgeHandler>())。

在 ASP.NET Core [授权文档](#)中，提供了关于检查用户年龄（基于 DateOfBirth）的自定义授权 Requirement 和处理器的示例。

其他资源

- **ASP.NET Core 认证**
<https://docs.microsoft.com/aspnet/core/security/authentication/identity>
- **ASP.NET Core 授权**
<https://docs.microsoft.com/aspnet/core/security/authorization/introduction>
- **基于角色的授权**
<https://docs.microsoft.com/aspnet/core/security/authorization/roles>
- **自定义基于策略的授权**
<https://docs.microsoft.com/aspnet/core/security/authorization/policies>

在开发中安全存储应用程序密钥

要连接受保护的资源或其它服务，ASP.NET Core 应用程序通常需要使用连接串、口令或者包含敏感信息的其它凭据。这些敏感信息被称为密钥。最佳实践是不要在源代码中包含密钥，当然也不要将密钥存储在源代码管理器中。相反，您应当使用 ASP.NET Core 配置模型从更安全的位置读取密钥。

您应该将用于开发的密钥和预发布资源的密钥从用于访问生产资源的密钥分离出来，因为不同的人需要访问不同的密钥集。为了存储在开发过程中使用的密钥，常见的方式是在环境变量中存储密钥，或者使用 ASP.NET Core 密钥管理工具。为了在生产环境中实现更安全的存储，微服务可以在 Azure Key Vault 中存储。

在环境变量中存储密钥

源代码中不存储密钥的一种方式是将开发人员将使用的密钥设置为开发机上的[环境变量](#)。当您使用环境变量存储具有分层结构的名称（嵌套在配置节中）的密钥时，请为包含该密钥名称的完整分层结构的环境变量创建一个名称，并使用冒号（:）分隔。

例如，将环境变量 Logging:LogLevel:Default 设置为 Debug，将等效于下面的 JSON 文件中的配置值：

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug"
    }
  }
}
```

要从环境变量中访问这些值，应用程序只需在构造 IConfigurationRoot 对象时，在 ConfigurationBuilder 上调用 AddEnvironmentVariables 方法。

注意，环境变量通常存储为纯文本，因此，如果存储环境变量的计算机或进程被攻破，环境变量将被发现。

使用 ASP.NET Core 密钥管理器存储密钥

ASP.NET Core [密钥管理](#)工具提供了另一种从源代码中不保存密钥的方式。要使用“密钥管理器”工具，请在项目文件中包含工具引用（DotNetCliToolReference）到 Microsoft.Extensions.SecretManager 包。一旦该依赖项存在并被恢复，dotnet 用户密钥命令就可以从命令行设置密钥值。这些密钥将以 JSON 格式文件存储在用户的配置文件目录中（具体因操作系统而不同），从而远离源代码。

密钥管理工具设置的密钥由使用密钥项目的 UserSecretsId 属性组织。因此，必须确保项目文件中设置 UserSecretsId 属性（如下所示）。作为 ID 使用的实际字符串并不重要，只要在项目中是唯一的。


```
<PropertyGroup>
  <UserSecretsId>UniqueIdentifyingString</UserSecretsId>
</PropertyGroup>
```

在应用程序中通过在 ConfigurationBuilder 实例上调用 AddUserSecrets<T> 来使用存储在密钥管理器中密钥。泛型参数 T 应当是使用 UserSecretId 的程序集类型。通常使用 AddUserSecrets<Startup> 就可以。

在产品中使用 Azure Key Vault 保护密钥

将密钥作为环境变量存储，或使用密钥管理工具存储，仍然在本地计算机上未机密存储。存储密钥更安全的方式是 [Azure Key Vault](#)，它为存储密钥和机密提供了一个安全的中心位置。

包 Microsoft.Extensions.Configuration.AzureKeyVault 允许 ASP.NET Core 应用程序从 Azure Key Vault 中读取配置信息。要使用 Azure Key Vault 中存储的密钥，请按照如下步骤操作：

1. 将应用程序注册为 Azure AD 应用程序。（对密钥库的访问由 Azure AD 管理）。这可以通过 Azure 管理门户完成。

或者，如果希望应用程序使用证书（而不是密码或者客户端密钥），则可以使用 [New-AzureRmADApplication](#) 这个 PowerShell 命令。您在 Azure Key Vault 中注册的证书只需要您的公钥。（您的应用程序将使用私钥。）

2. 通过创建新的服务主题，授予注册的应用程序访问密钥库的权限。可以使用如下的 PowerShell 命令执行此操作。

```
$sp = New-AzureRmADServicePrincipal -ApplicationId "<Application ID guid>"
Set-AzureRmKeyVaultAccessPolicy -VaultName "<VaultName>" -ServicePrincipalName
$sp.ServicePrincipalNames[0] -PermissionsToSecrets all -ResourceGroupName
"<KeyVault Resource Group>"
```

3. 在创建 IConfigurationRoot 实例时，通过调用 IConfigurationBuilder.AddAzureKeyVault 扩展方法，将密钥库作为配置源包含在应用程序中。请注意，调用 AddAzureKeyVault 将需要注册的应用程序 ID，并在前面的步骤中授予对密钥存储库的访问权限。

目前，.NET 标准库和 .NET Core 支持使用客户端 ID 和客户端密钥从 Azure Key Vault 获取配置信息。.NET Framework 应用程序可以使用 IConfigurationBuilder.AddAzureKeyVault 的一个重载，该 AddAzureKeyVault 需要一个证书来代替客户端密钥。在编写本书时，对于 .NET Standard 和 .NET Core 的重载版本 [正在开发中](#)。在接受证书的 AddAzureKeyVault 重载可用之前，ASP.NET Core 应用程序可以通过显式创建一个 KeyVaultClient 对象来访问具有基于证书身份验证的 **Azure Key Vault**，如下所示：

```

// Configure Key Vault client
var kvClient = new KeyVaultClient(new KeyVaultClient.AuthenticationCallback(async
    (authority, resource, scope) =>
    {
        var cert = // Get certificate from local store/file/key vault etc. as needed
        // From the Microsoft.IdentityModel.Clients.ActiveDirectory package
        var authContext = new AuthenticationContext(authority,
            TokenCache.DefaultShared);
        var result = await authContext.AcquireTokenAsync(resource,
            // From the Microsoft.Rest.ClientRuntime.Azure.Authentication package
            new ClientAssertionCertificate("{Application ID}", cert));
        return result.AccessToken;
    }));

// Get configuration values from Key Vault
var builder = new ConfigurationBuilder()
    .SetBasePath(env.ContentRootPath)
    // Other configuration providers go here.
    .AddAzureKeyVault("{KeyValueUri}", kvClient,
        new DefaultKeyVaultSecretManager());

```

在此示例中，对 `AddAzureKeyVault` 的调用在配置提供程序注册结束时出现。最佳实践是将 Azure Key Vault 注册为最后一个配置提供程序，以便它有机会重写前面提供程序的配置值，并使其它源中的配置值不会覆盖密钥存储库中的这些参数。

其他资源

- **使用 Azure Key Vault 保护应用秘密**
<https://docs.microsoft.com/azure/guidance/guidance-multitenant-identity-keyvault>
- **在开发阶段安全存储应用密钥**
<https://docs.microsoft.com/aspnet/core/security/app-secrets>
- **配置数据保护**
<https://docs.microsoft.com/aspnet/core/security/data-protection/configuration/overview>
- **密钥管理和生命周期**
<https://docs.microsoft.com/aspnet/core/security/data-protection/configuration/default-settings#data-protection-default-settings>
- **Microsoft.Extensions.Configuration.DockerSecrets**. GitHub 仓库
<https://github.com/aspnet/Configuration/tree/dev/src/Microsoft.Extensions.Configuration.DockerSecrets>

关键结论

作为总结和要点，以下是本书最重要的结论。

使用容器的优势。基于容器的方案提供了节省成本的重要优势，因为容器是解决由于在生产环境下缺失依赖关系而导致的部署问题的解决方案。容器显著改善 DevOps 和生产作业流程。

容器无处不在。在 Windows 和 Linux 生态系统中最重要供应商的支持下，基于 Docker 的容器正在成为容器行业的事实标准。这些供应商包括微软、亚马逊 AWS、谷歌和 IBM。在不久的将来，Docker 可能在云和内部数据中心都无处不在。

容器作为部署单位。对于任何基于服务器的应用程序或服务，Docker 容器成为部署的基本单位。

微服务。微服务架构正在成为分布式、大型的或复杂的关键任务型应用程序的首选方式，它基于多个独立的子系统，以自主服务的形式运行。在基于微服务的架构中，应用程序可以被构建为可独立开发、测试、版本化、部署和扩展的服务集合，这可以包括任何相关的自治的数据库。

领域驱动设计和 SOA。微服务架构模式源于面向服务的体系结构 (SOA) 和领域驱动设计 (DDD)。当您为不断发展的业务规则形成的特定领域的领域设计和开发微服务时，必须考虑到 DDD 方法和模式。

微服务的挑战。微服务提供了许多强大的功能，如独立部署，强子系统边界和技术多样性。然而，它们也引发了与分布式应用程序开发相关的众多新挑战，例如分散和独立的数据模型，通信，最终一致性，以及来自多个微服务的聚合日志记录和监视信息导致的操作复杂性。这些方面引入了比传统的单体应用程序更高的复杂性。因此，只有特定的场景适用基于微服务的应用程序。其中包括具有多个演进子系统的大型复杂应用程序。在这些情况下，值得投资于更复杂的软件架构，因为它将提供更好的长期敏捷性和更好的应用程序维护性。

适用于任何应用程序的容器。容器适用于微服务，但并不排斥其它类型的应用。容器还可以与单体应用程序一起使用，像基于传统的 .NET Framework 的遗留应用程序，可以通过 Windows 容器进行现代化。Docker 的优势适用于许多不同类型的应用程序，例如解决许多部署到生产系统的问题，并提供最先进的开发和测试环境

CLI 与 IDE。利用微软的工具，您可以使用您偏爱的方式开发容器化的 .NET 应用程序。通过使用 Docker CLI 和 Visual Studio Code，您可以使用 CLI 和基于编辑器的环境开发。或者利用 Visual Studio 和其对于 Docker 的特有特性的支持，使用以 IDE 为中心的方式开发，例如能够调试多容器的应用程序，

弹性云应用程序。在基于云的系统 and 分布式系统中，总是存在部分故障的风险。由于客户端和服务是分离的进程（容器），服务可能无法及时响应客户端的请求。例如，由于部分故障或维护，服务可能会关闭；或者该服务可能过载，响应速度非常慢；或者由于网络问题，可能在短时间内无法访问该服务。因此，基于云的应用程序必须接受这些故障，并制订应对这些故障的策略。这些策略可以包括重试策略（重新发送消息或者重试请求），并实现熔断器模式以避免重复请求导致的指数负载。基本上，基于云的应用程序必须具有弹性机制，无论是定制的还是基于云基础设施的机制，例如来自编排引擎或者服务总线的高级框架。

安全。我们现代的容器和微服务可能会暴露新的漏洞。基础应用安全是基于验证和授权的，存在多种实现方式。但是，容器安全性包含了额外的关键组件，这些组件可以产生更安全的应用程序。构建更为安全的应用程序的一个关键因素是有一个与其它应用程序和系统进行通讯的安全方式，通常需要凭据、令牌、密码和其它类型的机密信息 - 通常称为应用程序机密。任何安全解决方案必须遵循安全最佳实践，例如在传输过程中加密；静态加密；以及在最终应用程序消费数据时，防止无意中泄密。这些机密需要在某个地方存储并保持安全。为提高安全，您可以利用您所选择的编排引擎的基础架构，或者利用 Azure 基础架构，例如 Azure 的密钥保管库（Key Vault）键值以及其提供给应用程序代码使用的方式。

编排引擎。基于容器的编排引擎，像 Azure 容器服务（Kubernetes、Mesos DC/OS 和 Docker Swarm），和 Azure Service Fabric 中提供的，对于任何生产就绪的基于微服务的应用程序，和具有显著复杂性的、需扩展及持续演进的多容器应用程序来说，都是必不可少的。本书介绍了编排引擎及其在基于微服务和容器的解决方案中的作用。如果您的应用需要转移到复杂的容器应用，您将发现寻找更多资源去深入了解编排引擎是有帮助的。