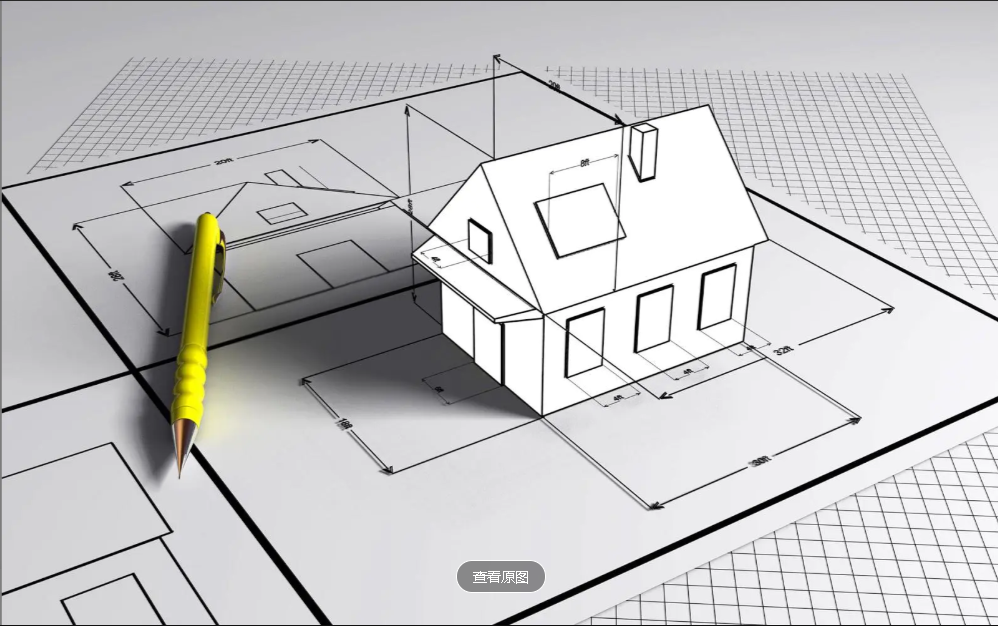
# **后端开发实践系列——Spring Boot项目模板**

在我的工作中，我从零开始搭建了不少软件项目，其中包含了基础代码框架和持续集成基础设施等，这些内容在敏捷开发中通常被称为“第0个迭代”要做的事情。但是，当项目运行了一段时间之后再来反观，我总会发现一些不足的地方，要么测试分类没有分好，要么基本的编码架子没有考虑周全。

另外，我在工作中也会接触到很多既有项目，公司内部和外部的都有，多数项目的编码实践我都是不满意的。比如，我曾经新加入一个项目的时候，前前后后请教了3位同事才把该项目在本地运行起来；又比如在另一项目中，我发现前端请求对应的Java类命名规范不统一，有被后缀为Request的，也有被后缀为Command的。

再者，工作了这么多年之后，我越来越发现基础知识以及系统性学习的重要性。诚然，技术框架的发展使得我们可以快速地实现业务功能，但是当软件出了问题之后有时却需要将各方面的知识融会贯通并在大脑里综合反应才能找到解决思路。

基于以上，我希望整理出一套公共性的项目模板出来，旨在尽量多地包含日常开发之所需，减少开发者的重复性工作以及提供一些最佳实践。对于后端开发而言，我选择了当前被行业大量使用的Spring Boot，基于此整理出了一套公共的、基础性的实践方式，在结合了自己的经验以及其他项目的优秀实践之后，总结出本文以飨开发者。



本文以一个简单的电商订单系统为例，源代码请访问：

git clone https://github.com/e-commerce-sample/order-backend  
git checkout a443dace

所使用的技术栈主要包括：Spring Boot、Gradle、MySQL、Junit 5、Rest Assured、Docker等。

## 第一步：从写好README开始

一份好的README可以给人以项目全景概览，可以使新人快速上手项目，可以降低沟通成本。同时，README应该简明扼要，条理清晰，建议包含以下方面：

* 项目简介：用一两句话简单描述该项目所实现的业务功能；
* 技术选型：列出项目的技术栈，包括语言、框架和中间件等；
* 本地构建：列出本地开发过程中所用到的工具命令；
* 领域模型：核心的领域概念，比如对于示例电商系统来说有Order、Product等；
* 测试策略：自动化测试如何分类，哪些必须写测试，哪些没有必要写测试；
* 技术架构：技术架构图；
* 部署架构：部署架构图；
* 外部依赖：项目运行时所依赖的外部集成方，比如订单系统会依赖于会员系统；
* 环境信息：各个环境的访问方式，数据库连接等；
* 编码实践：统一的编码实践，比如异常处理原则、分页封装等；
* FAQ：开发过程中常见问题的解答。

需要注意的是，README中的信息可能随着项目的演进而改变（比如引入了新的技术栈或者加入了新的领域模型），因此也是需要持续更新的。虽然我们知道，软件文档的一个痛点便是无法与项目实际进展保持同步，但是就README这点信息来讲，还是建议开发者们不要吝啬那一点点敲键盘的时间。

此外，除了保持README的持续更新，一些重要的架构决定可以通过示例代码的形式记录在代码库中，新开发者可以通过直接阅读这些示例代码快速了解项目的通用实践方式以及架构选择，请参考ThoughtWorks的[技术雷达](https://links.jianshu.com/go?to=https://www.thoughtworks.com/radar/techniques/lightweight-architecture-decision-records" \t "_blank)。

## 一键式本地构建

为了避免诸如前文中所提到的“请教了3位同事才本地构建成功”的尴尬，为了减少“懒惰”的程序员们的手动操作，也为了为所有开发者提供一种一致的开发体验，我们希望用一个命令就可以完成所有的事情。这里，对于不同的场景我总结出了以下命令：

* 生成IDE工程：idea.sh，生成IntelliJ工程文件并自动打开IntelliJ
* 本地运行：run.sh，本地启动项目，自动启动本地数据库，监听调试端口5005
* 本地构建：local-build.sh，只有本地构建成功才能提交代码

以上3个命令基本上可以完成日常开发之所需，此时，对于新人的开发流程大致为：

1. 拉取代码；
2. 运行idea.sh，自动打开IntelliJ；
3. 编写代码，包含业务代码和自动化测试；
4. 运行run.sh，进行本地调试或必要的手动测试(本步骤不是必需)；
5. 运行local-build.sh，完成本地构建；
6. 再次拉取代码，保证local-build.sh成功，提交代码。

事实上，这些命令脚本的内容非常简单，比如run.sh文件内容为：

#!/usr/bin/env bash

./gradlew clean bootRun

然而，这种显式化的命令却可以减少新人的恐惧感，因为他们只需要知道运行这3个命令就可以搞开发了。另外，一个小小的细节：本地构建的local-build.sh命令本来可以重命名为更简单的build.sh，但是当我们在命令行中使用Tab键自动补全的时候，会发现自动补全到了build目录，而不是build.sh命令，并不方便，因此命名为了local-build.sh。细节虽小，但是却体现了一个宗旨，即我们希望给开发者一种极简的开发体验，我把这些看似微不足道的东西称作是对程序员的“人文关怀”。

## 目录结构

Maven所提倡的[目录结构](https://links.jianshu.com/go?to=https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html" \t "_blank)当前已经成为事实上的行业标准，Gradle在默认情况下也采用了Maven的目录结构，这对于多数项目来说已经足够了。此外，除了Java代码，项目中还存在其他类型的文件，比如Gradle插件的配置、工具脚本和部署配置等。无论如何，项目目录结构的原则是简单而有条理，不要随意地增加多余的文件夹，并且也需要及时重构。

在示例项目中，顶层只有2个文件夹，一个是用于放置Java源代码和项目配置的src文件夹，另一个是用于放置所有Gradle配置的gradle文件夹，此外，为了方便开发人员使用，将上文提到的3个常用脚本直接放到根目录下：

└── order-backend

├── gradle // 文件夹，用于放置所有Gradle配置

├── src // 文件夹，Java源代码

├── idea.sh //生成IntelliJ工程

├── local-build.sh // 提交之前的本地构建

└── run.sh // 本地运行

对于gradle而言，我们刻意地将Gradle插件脚本与插件配置放到了一起，比如Checkstyle：

├── gradle

│ ├── checkstyle

│ │ ├── checkstyle.gradle

│ │ └── checkstyle.xml

事实上，在默认情况下Checkstyle插件会从项目根目录下的config目录查找checkstyle.xml配置文件，但是这一方面增加了多余的文件夹，另一方面与该插件相关的设施分散在了不同的地方，违背了广义上的内聚原则。

## 基于业务分包

早年的Java分包方式通常是基于技术的，比如与domain包平级的有controller包、service包和infrastructure包等。这种方式当前并不被行业所推崇，而是应该首先基于业务分包。比如，在订单示例项目中，有两个重要的领域对象Order和Product（在[DDD](https://links.jianshu.com/go?to=%5Bhttps://en.wikipedia.org/wiki/Domain-driven_design%5D(https://en.wikipedia.org/wiki/Domain-driven_design)" \t "_blank)中称为[聚合根](https://links.jianshu.com/go?to=%5Bhttps://martinfowler.com/bliki/DDD_Aggregate.html%5D(https://martinfowler.com/bliki/DDD_Aggregate.html)" \t "_blank)），所有的业务都围绕它们展开，因此分别创建order包和product包，再分别在包下创建与之相关的各个子包。此时的order包如下：

├── order

│ ├── OrderApplicationService.java

│ ├── OrderController.java

│ ├── OrderNotFoundException.java

│ ├── OrderRepository.java

│ ├── OrderService.java

│ └── model

│ ├── Order.java

│ ├── OrderFactory.java

│ ├── OrderId.java

│ ├── OrderItem.java

│ └── OrderStatus.java

可以看到，在order包下我们直接放置了OrderController和OrderRepository等类，而没有必要再为这些类划分单独的子包。而对于领域模型Order来讲，由于包含了多个对象，因此基于内聚性原则将它们归到model包中。但是这并不是一个必须，如果业务足够简单，我们甚至可以将所有类直接放到业务包下，product包便是如此：

└── product

├── Product.java

├── ProductApplicationService.java

├── ProductController.java

├── ProductId.java

└── ProductRepository.java

在编码实践中，我们总是基于一个业务用例来实现代码，在技术分包场景下，我们需要在分散的各包中来回切换，增加了代码导航的成本；另外，代码提交的变更内容也是散落的，在查看代码提交历史时，无法直观的看出该次提交是关于什么业务功能的。在业务分包下，我们只需要在单个统一的包下修改代码，减少了代码导航成本；另外一个好处是，如果哪天我们需要将某个业务迁移到另外的项目（比如识别出了独立的微服务），那么直接整体移动业务包即可。

当然，基于业务分包并不意味着所有的代码都必须囿于业务包下，这里的逻辑是：优先进行业务分包，然后对于一些不隶属于任何业务的代码可以单独分包，比如一些util类、公共配置等。比如我们依然可以创建一个common包，下面放置了Spring公共配置、异常处理框架和日志等子包：

└── common

├── configuration

├── exception

├── loggin

└── utils

## 自动化测试分类

在当前的微服务和前后端分离的开发模式下，后端项目仅提供纯粹的业务API，而不包含UI逻辑，因此后端项目不会再包含诸如WebDriver的重量级端到端测试。同时，后端项目作为向外提供业务功能的独立运行单元，在API级别也应该有相应的测试。

此外，程序中有些框架性代码，要么是诸如Controller之类的技术性框架代码，要么是基于某种架构风格的代码（比如DDD实践中的[ApplicationService](https://links.jianshu.com/go?to=%5Bhttps://blog.arkency.com/application-service-ruby-rails-ddd/%5D(https://blog.arkency.com/application-service-ruby-rails-ddd/)" \t "_blank)），这些代码一方面并不包含业务逻辑，一方面是很薄的一个抽象层（即实现相对简单），用单元测试来覆盖显得没有必要，因此笔者的观点是可以不为此编写单独的单元测试。再者，程序中有些重要的组件性代码，比如访问数据库的Repository或者分布式锁，使用单元测试实际上“测不到点上”，而使用API测试又显得在分类逻辑上不合理，为此我们可以专门创建一种测试类型谓之组件测试。

基于以上，我们可以对自动化测试做个分类：

* 单元测试：核心的领域模型，包括领域对象(比如Order类)，Factory类，领域服务类等；
* 组件测试：不适合写单元测试但是又必须测试的类，比如Repository类，在有些项目中，这种类型测试也被称为集成测试；
* API测试：模拟客户端测试各个API接口，需要启动程序。

Gradle在默认情况下只提供src/test/java目录用于测试，对于以上3种类型的测试，我们需要将它们分开以便于管理（也是职责分离的体现）。为此，可以通过Gradle提供的SourceSets对测试代码进行分类：

sourceSets {

componentTest {

compileClasspath += sourceSets.main.output + sourceSets.test.output

runtimeClasspath += sourceSets.main.output + sourceSets.test.output

}

apiTest {

compileClasspath += sourceSets.main.output + sourceSets.test.output

runtimeClasspath += sourceSets.main.output + sourceSets.test.output

}

}

到此，3种类型的测试可以分别编写在以下目录：

* 单元测试：src/test/java
* 组件测试：src/componentTest/java
* API测试：src/apiTest/java

需要注意的是，这里的API测试更多强调的是对业务功能的测试，有些项目中可能还会存在契约测试和安全测试等，虽然从技术上讲都是对API的访问，但是这些测试都是单独的关注点，因此建议分开对待。

值得一提的是，由于组件测试和API测试需要启动程序，也即需要准备好本地数据库，我们采用了Gradle的[docker-compose](https://links.jianshu.com/go?to=https://plugins.gradle.org/plugin/com.avast.gradle.docker-compose" \t "_blank)插件(或者[jib插件](https://links.jianshu.com/go?to=https://github.com/GoogleContainerTools/jib" \t "_blank))，该插件会在运行测试之前自动运行Docker容器（比如MySQL）：

apply plugin: 'docker-compose'

dockerCompose {

useComposeFiles = ['docker/mysql/docker-compose.yml']

}

bootRun.dependsOn composeUp

componentTest.dependsOn composeUp

apiTest.dependsOn composeUp

更多的测试分类配置细节，比如[JaCoCo](https://links.jianshu.com/go?to=https://docs.gradle.org/current/userguide/jacoco_plugin.html" \t "_blank)测试覆盖率配置等，请参考本文的示例项目代码。对Gradle不熟悉的读者可以参考笔者的[Gradle学习系列](https://links.jianshu.com/go?to=https://www.cnblogs.com/CloudTeng/p/3417762.html" \t "_blank)文章。

## 日志处理

在日志处理中，除了完成基本配置外，还有2个需要考虑的点：

1. 在日志中加入请求标识，便于链路追踪。在处理一个请求的过程中有时会输出多条日志，如果每条日志都共享统一的请求ID，那么在日志追踪时会更加方便。此时，可以使用Logback原生提供的MDC(Mapped Diagnostic Context)功能，创建一个RequestIdMdcFilter：

protected void doFilterInternal(HttpServletRequest request,

HttpServletResponse response,

FilterChain filterChain)

throws ServletException, IOException {

//request id in header may come from Gateway, eg. Nginx

String headerRequestId = request.getHeader(HEADER\_X\_REQUEST\_ID);

MDC.put(REQUEST\_ID, isNullOrEmpty(headerRequestId) ? newUuid() : headerRequestId);

try {

filterChain.doFilter(request, response);

} finally {

clearMdc();

}

}

1. 集中式日志管理，在多节点部署的场景下，各个节点的日志是分散的，为此可以引入诸如ELK之类的工具将日志统一输出到ElasticSearch中。本文的示例项目使用了RedisAppender将日志输出到Logstash：

<appender name="REDIS" class="com.cwbase.logback.RedisAppender">

<tags>ecommerce-order-backend-${ACTIVE\_PROFILE}</tags>

<host>elk.yourdomain.com</host>

<port>6379</port>

<password>whatever</password>

<key>ecommerce-ordder-log</key>

<mdc>true</mdc>

<type>redis</type></appender>

当然，统一日志的方案还有很多，比如Splunk和Graylog等。

## 异常处理

在设计异常处理的框架时，需要考虑以下几点：

* 向客户端提供格式统一的异常返回
* 异常信息中应该包含足够多的上下文信息，最好是结构化的数据以便于客户端解析
* 不同类型的异常应该包含唯一标识，以便客户端精确识别

异常处理通常有两种形式，一种是层级式的，即每种具体的异常都对应了一个异常类，这些类最终继承自某个父异常；另一种是单一式的，即整个程序中只有一个异常类，再以一个字段来区分不同的异常场景。层级式异常的好处是能够显式化异常含义，但是如果层级设计不好可能导致整个程序中充斥着大量的异常类；单一式的好处是简单，而其缺点在于表意性不够。

本文的示例项目使用了层级式异常，所有异常都继承自一个AppException：

public abstract class AppException extends RuntimeException {

private final ErrorCode code;

private final Map<String, Object> data = newHashMap();}

这里，ErrorCode枚举中包含了异常的唯一标识、HTTP状态码以及错误信息；而data字段表示各个异常的上下文信息。

在示例系统中，在没有找到订单时抛出异常：

public class OrderNotFoundException extends AppException {

public OrderNotFoundException(OrderId orderId) {

super(ErrorCode.ORDER\_NOT\_FOUND, ImmutableMap.of("orderId", orderId.toString()));

}}

在返回异常给客户端时，通过一个ErrorDetail类来统一异常格式：

public final class ErrorDetail {

private final ErrorCode code;

private final int status;

private final String message;

private final String path;

private final Instant timestamp;

private final Map<String, Object> data = newHashMap();}

最终返回客户端的数据为：

{

requestId: "d008ef46bb4f4cf19c9081ad50df33bd",

error: {

code: "ORDER\_NOT\_FOUND",

status: 404,

message: "没有找到订单",

path: "/order",

timestamp: 1555031270087,

data: {

orderId: "123456789"

}

}}

可以看到，ORDER\_NOT\_FOUND与data中的数据结构是一一对应的，也即对于客户端来讲，如果发现了ORDER\_NOT\_FOUND，那么便可确定data中一定存在orderId字段，进而完成精确的结构化解析。

## 后台任务与分布式锁

除了即时完成客户端的请求外，系统中通常会有一些定时性的例行任务，比如定期地向用户发送邮件或者运行数据报表等；另外，有时从设计上我们会对请求进行异步化处理。此时，我们需要搭建后台任务相关基础设施。Spring原生提供了任务处理(TaskExecutor)和任务计划(TaskSchedulor)机制；而在分布式场景下，还需要引入[分布式锁](https://links.jianshu.com/go?to=https://en.wikipedia.org/wiki/Distributed_lock_manager" \t "_blank)来解决并发冲突，为此我们引入一个轻量级的分布式锁框架[ShedLock](https://links.jianshu.com/go?to=https://github.com/lukas-krecan/ShedLock" \t "_blank)。

启用Spring任务配置如下：

@Configuration@EnableAsync@EnableSchedulingpublic class SchedulingConfiguration implements SchedulingConfigurer {

@Override

public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {

taskRegistrar.setScheduler(newScheduledThreadPool(10));

}

@Bean(destroyMethod = "shutdown")

@Primary

public TaskExecutor taskExecutor() {

ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();

executor.setCorePoolSize(2);

executor.setMaxPoolSize(5);

executor.setQueueCapacity(10);

executor.setTaskDecorator(new LogbackMdcTaskDecorator());

executor.initialize();

return executor;

}

}

然后配置Shedlock：

@Configuration@EnableSchedulerLock(defaultLockAtMostFor = "PT30S")public class DistributedLockConfiguration {

@Bean

public LockProvider lockProvider(DataSource dataSource) {

return new JdbcTemplateLockProvider(dataSource);

}

@Bean

public DistributedLockExecutor distributedLockExecutor(LockProvider lockProvider) {

return new DistributedLockExecutor(lockProvider);

}

}

实现后台任务处理：

@Scheduled(cron = "0 0/1 \* \* \* ?")

@SchedulerLock(name = "scheduledTask", lockAtMostFor = THIRTY\_MIN, lockAtLeastFor = ONE\_MIN)

public void run() {

logger.info("Run scheduled task.");

}

为了支持代码直接调用分布式锁，基于Shedlock的LockProvider创建DistributedLockExecutor：

public class DistributedLockExecutor {

private final LockProvider lockProvider;

public DistributedLockExecutor(LockProvider lockProvider) {

this.lockProvider = lockProvider;

}

public <T> T executeWithLock(Supplier<T> supplier, LockConfiguration configuration) {

Optional<SimpleLock> lock = lockProvider.lock(configuration);

if (!lock.isPresent()) {

throw new LockAlreadyOccupiedException(configuration.getName());

}

try {

return supplier.get();

} finally {

lock.get().unlock();

}

}

}

使用时在代码中直接调用：

public String doBusiness() {

return distributedLockExecutor.executeWithLock(() -> "Hello World.",

new LockConfiguration("key", Instant.now().plusSeconds(60)));

}

本文的示例项目使用了基于JDBC的分布式锁，事实上任何提供原子操作的机制都可用于分布式锁，Shedlock还提供基于Redis、ZooKeeper和Hazelcast等的分布式锁实现机制。

## 统一代码风格

除了Checkstyle统一代码格式之外，项目中有些通用的公共的编码实践方式也需要在整个开发团队中进行统一，包括但不限于以下方面：

* 客户端的请求数据类统一使用相同后缀，比如Command
* 返回给客户端的数据统一使用相同后缀，比如Represetation
* 统一对请求处理的流程框架，比如采用传统的3层架构或者DDD战术模式
* 提供一致的异常返回（请参考“异常处理”小节）
* 提供统一的分页结构类
* 明确测试分类以及统一的测试基础类（请参考“自动化测试分类”小节）

## 静态代码检查

静态代码检查主要包含以下Gradle插件，具体配置请参考本文示例代码：

* [Checkstyle](https://links.jianshu.com/go?to=https://docs.gradle.org/current/userguide/checkstyle_plugin.html" \t "_blank)：用于检查代码格式，规范编码风格
* [Spotbugs](https://links.jianshu.com/go?to=https://spotbugs.github.io/" \t "_blank)：Findbugs的继承者
* [Dependency check](https://links.jianshu.com/go?to=https://plugins.gradle.org/plugin/org.owasp.dependencycheck" \t "_blank)：[OWASP](https://links.jianshu.com/go?to=https://www.owasp.org/index.php/OWASP_Dependency_Check" \t "_blank)提供的Java类库安全性检查
* [Sonar](https://links.jianshu.com/go?to=https://en.wikipedia.org/wiki/Sonar" \t "_blank)：用于代码持续改进的跟踪

## 健康检查

健康检查主要用于以下场景：

* 我们希望初步检查程序是否运行正常
* 有些负载均衡软件会通过一个健康检查URL判断节点的可达性

此时，可以实现一个简单的API接口，该接口不受权限管控，可以公开访问。如果该接口返回HTTP的200状态码，便可初步认为程序运行正常。此外，我们还可以在该API中加入一些额外的信息，比如提交版本号、构建时间、部署时间等。

启动本文的示例项目：

./run.sh

然后访问健康检查API：[http://localhost:8080/about](https://links.jianshu.com/go?to=http://localhost:8080/about" \t "_blank)，结果如下：

{

requestId: "698c8d29add54e24a3d435e2c749ea00",

buildNumber: "unknown",

buildTime: "unknown",

deployTime: "2019-04-11T13:05:46.901+08:00[Asia/Shanghai]",

gitRevision: "unknown",

gitBranch: "unknown",

environment: "[local]"}

以上接口在示例项目中用了一个简单的Controller实现，事实上Spring Boot[的Acuator](https://links.jianshu.com/go?to=https://github.com/spring-projects/spring-boot/tree/master/spring-boot-project/spring-boot-actuator" \t "_blank)框架也能够提供相似的功能。

## API文档

软件文档的难点不在于写，而在于维护。多少次，当我对照着项目文档一步一步往下走时，总得不到正确的结果，问了同事之后得到回复“哦，那个已经过时了”。本文示例项目所采用的[Swagger](https://links.jianshu.com/go?to=%5Bhttp://springfox.github.io/springfox/%5D(http://springfox.github.io/springfox/)" \t "_blank)在一定程度上降低了API维护的成本，因为Swagger能自动识别代码中的方法参数、返回对象和URL等信息，然后自动地实时地创建出API文档。

配置Swagger如下：

@Configuration@EnableSwagger2@Profile(value = {"local", "dev"})public class SwaggerConfiguration {

@Bean

public Docket api() {

return new Docket(SWAGGER\_2)

.select()

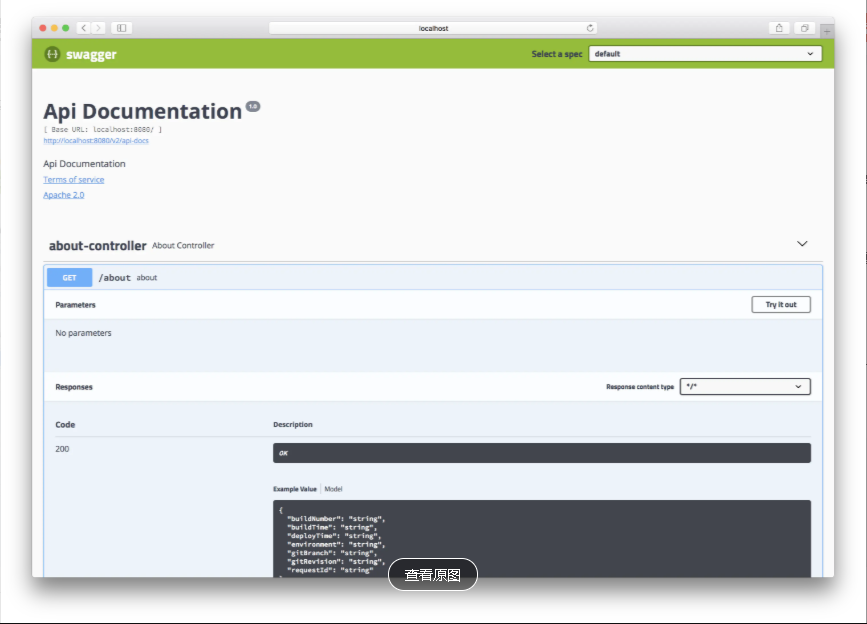
.apis(basePackage("com.ecommerce.order"))

.paths(any())

.build();

}}

启动本地项目，访问[http://localhost:8080/swagger-ui.html](https://links.jianshu.com/go?to=http://localhost:8080/swagger-ui.html" \t "_blank)：



Swagger API文档

## 数据库迁移

在传统的开发模式中，数据库由专门的运维团队或者DBA来维护，要对数据库进行修改需要向DBA申请，告之迁移内容，最后由DBA负责数据库变更实施。在[持续交付](https://links.jianshu.com/go?to=%5Bhttps://en.wikipedia.org/wiki/Continuous_delivery%5D(https://en.wikipedia.org/wiki/Continuous_delivery)" \t "_blank)和[DevOps](https://links.jianshu.com/go?to=%5Bhttps://en.wikipedia.org/wiki/DevOps%5D(https://en.wikipedia.org/wiki/DevOps)" \t "_blank)运动中，这些工作逐步提前到开发过程，当然并不是说不需要DBA了，而是这些工作可以由开发者和运维人员一同完成。另外，在微服务场景下，数据库被包含在单个服务的边界之内，因此基于内聚性原则（咦，这好像是本文第三次提到内聚原则了，可见其在软件开发中的重要性），数据库的变更最好也与项目代码一道维护在代码库中。

本文的示例项目采用了Flyway作为数据库迁移工具，加入了[Flyway](https://links.jianshu.com/go?to=https://flywaydb.org/" \t "_blank)依赖后，在src/main/sources/db/migration目录下创建迁移脚本文件即可：

resources/

├── db

│ └── migration

│ ├── V1\_\_init.sql

│ └── V2\_\_create\_product\_table.sql

迁移脚本的命名需要遵循一定的[规则](https://links.jianshu.com/go?to=https://flywaydb.org/documentation/migrations%23syntax" \t "_blank)以保证脚本执行顺序，另外迁移文件生效之后不要任意修改，因为Flyway会检查文件的checksum，如果checksum不一致将导致迁移失败。

## 多环境构建

在软件的开发流程中，我们需要将软件部署到多个环境，经过多轮验证后才能最终上线。在不同的阶段中，软件的运行态可能是不一样的，比如本地开发时可能将所依赖的第三方系统stub掉；持续集成构建时可能使用的是测试用的内存数据库等等。为此，本文的示例项目推荐采用以下环境：

* local：用于开发者本地开发
* ci：用于持续集成
* dev：用于前端开发联调
* qa：用于测试人员
* uat：类生产环境，用于功能验收(有时也称为staging环境)
* prod：正式的生产环境

## CORS

在前后端分离的系统中，前端单独部署，有时连域名都和后端不同，此时需要进行跨域处理。传统的做法可以通过JSONP，但这是一种比较“trick”的做法，当前更通用的实践是采用[CORS](https://links.jianshu.com/go?to=https://en.wikipedia.org/wiki/Cross-origin_resource_sharing" \t "_blank)机制，在Spring Boot项目中，启用CORS配置如下：

@Configurationpublic class CorsConfiguration {

@Bean

public WebMvcConfigurer corsConfigurer() {

return new WebMvcConfigurer() {

@Override

public void addCorsMappings(CorsRegistry registry) {

registry.addMapping("/\*\*");

}

};

}}

对于使用Spring Security的项目，需要保证CORS工作于Spring Security的过滤器之前，为此Spring Security专门提供了相应配置：

@EnableWebSecuritypublic class WebSecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http

// by default uses a Bean by the name of corsConfigurationSource

.cors().and()

...

}

@Bean

CorsConfigurationSource corsConfigurationSource() {

CorsConfiguration configuration = new CorsConfiguration();

configuration.setAllowedOrigins(Arrays.asList("https://example.com"));

configuration.setAllowedMethods(Arrays.asList("GET","POST"));

UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();

source.registerCorsConfiguration("/\*\*", configuration);

return source;

}}

## 常用第三方类库

这里列出一些比较常见的第三方库，开发者们可以根据项目所需引入：

* [Guava](https://links.jianshu.com/go?to=https://en.wikipedia.org/wiki/Guava" \t "_blank)：来自Google的常用类库
* [Apache Commons](https://links.jianshu.com/go?to=https://commons.apache.org/" \t "_blank)：来自Apache的常用类库
* [Mockito](https://links.jianshu.com/go?to=https://site.mockito.org/" \t "_blank)：主要用于单元测试的mock
* [DBUnit](https://links.jianshu.com/go?to=http://dbunit.sourceforge.net/" \t "_blank)：测试中管理数据库测试数据
* [Rest Assured](https://links.jianshu.com/go?to=http://rest-assured.io/" \t "_blank)：用于Rest API测试
* [Jackson 2](https://links.jianshu.com/go?to=https://www.mkyong.com/java/jackson-2-convert-java-object-to-from-json/" \t "_blank)：Json数据的序列化和反序列化
* [jjwt](https://links.jianshu.com/go?to=https://jwt.io/" \t "_blank)：Jwt token认证
* [Lombok](https://links.jianshu.com/go?to=https://projectlombok.org/" \t "_blank)：自动生成常见Java代码，比如equals()方法，getter和setter等；
* [Feign](https://links.jianshu.com/go?to=https://github.com/OpenFeign/feign" \t "_blank)：声明式Rest客户端
* [Tika](https://links.jianshu.com/go?to=https://tika.apache.org/" \t "_blank)：用于准确检测文件类型
* [itext](https://links.jianshu.com/go?to=https://itextpdf.com/en" \t "_blank)：生成Pdf文件等
* [zxing](https://links.jianshu.com/go?to=https://github.com/zxing/zxing" \t "_blank)：生成二维码
* [Xstream](https://links.jianshu.com/go?to=https://x-stream.github.io/" \t "_blank)：比Jaxb更轻量级的XML处理库

## 总结

本文通过一个示例项目谈及到了项目之初开发者搭建后端工程的诸多方面，其中的绝大多数实践均在笔者的项目中真实落地。读完本文之后你可能会发现，文中的很多内容都是很基础很简单的。没错，的确没有什么难的东西，但是要系统性地搭建好后端项目的基础框架却不见得是每个开发团队都已经做到的事情，而这恰恰是本文的目的。最后，需要提醒的是，本文提到的实践方式只是一个参考，一方面依然存在考虑不周的地方，另一方面示例项目中用到的技术工具还存在其他替代方案，请根据自己项目的实际情况进行取舍。

作者：无知者云  
链接：https://www.jianshu.com/p/bd8136129dfb  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

# **后端开发实践系列之二——领域驱动设计(DDD)编码实践**

Martin Fowler在《[企业应用架构模式](https://links.jianshu.com/go?to=%5Bhttps://www.martinfowler.com/books/eaa.html%5D(https://www.martinfowler.com/books/eaa.html)" \t "_blank)》一书中写道：

I found this(business logic) a curious term because there are few things that are less logical than business logic.

初略翻译过来可以理解为：业务逻辑是很没有逻辑的逻辑。

的确，很多时候软件的业务逻辑是无法通过推理而得到的，有时甚至是被臆想出来的。这样的结果使得原本已经很复杂的业务变得更加复杂而难以理解。而在具体编码实现时，除了应付业务上的复杂性，技术上的复杂性也不能忽略，比如我们要讲究技术上的分层，要遵循软件开发的基本原则，又比如要考虑到性能和安全等等。

在很多项目中，技术复杂度与业务复杂度相互交错纠缠不清，这种火上浇油的做法成为不少软件项目无法继续往下演进的原因。然而，在合理的设计下，技术和业务是可以分离开来或者至少它们之间的耦合度是可以降低的。在不同的软件建模方法中，[领域驱动设计](https://links.jianshu.com/go?to=https://en.wikipedia.org/wiki/Domain-driven_design" \t "_blank)(Domain Driven Design,DDD)尝试通过其自有的原则与套路来解决软件的复杂性问题，它将研发者的目光首先聚焦在业务本身上，使技术架构和代码实现成为软件建模过程中的“副产品”。

## DDD总览

DDD分为战略设计和战术设计。在战略设计中，我们讲求的是子域和[限界上下文(Bounded Context,BC)](https://links.jianshu.com/go?to=https://martinfowler.com/bliki/BoundedContext.html" \t "_blank)的划分，以及各个限界上下文之间的上下游关系。当前如此火热的“在微服务中使用DDD”这个命题，究其最初的逻辑无外乎是“DDD中的限界上下文可以用于指导微服务中的服务划分”。事实上，限界上下文依然是软件模块化的一种体现，与我们一直以来追求的模块化原则的驱动力是相同的，即通过一定的手段使软件系统在人的大脑中更加有条理地呈现，让作为“目的”的人能够更简单地了解进而掌控软件系统。

如果说战略设计更偏向于软件架构，那么战术设计便更偏向于编码实现。DDD战术设计的目的是使得业务能够从技术中分离并突显出来，让代码直接表达业务的本身，其中包含了聚合根、应用服务、资源库、工厂等概念。虽然DDD不一定通过面向对象(OO)来实现，但是通常情况下在实践DDD时我们采用的是OO编程范式，行业中甚至有种说法是“DDD是OO进阶”，意思是面向对象中的基本原则(比如[SOLID](https://links.jianshu.com/go?to=https://en.wikipedia.org/wiki/SOLID" \t "_blank))在DDD中依然成立。本文主要讲解DDD的战术设计。

本文以一个简单的电商订单系统为例，通过以下方式可以获取源代码：

git clone https://github.com/e-commerce-sample/order-backend  
git checkout a443dace

## 实现业务的3种常见方式

在讲解DDD之前，让我们先来看一下实现业务代码的几种常见方式，在示例项目中有个“修改Order中Product的数量”的业务需求如下：

可以修改Order中Product的数量，但前提是Order处于未支付状态，Product数量变更后Order的总价(totalPrice)应该随之更新。

#### 1. 基于“Service + 贫血模型”的实现

这种方式当前被很多软件项目所采用，主要的特点是：存在一个贫血的“领域对象”，业务逻辑通过一个Service类实现，然后通过setter方法更新领域对象，最后通过[DAO](https://links.jianshu.com/go?to=https://en.wikipedia.org/wiki/Data_access_object" \t "_blank)(多数情况下可能使用诸如Hibernate之类的ORM框架)保存到数据库中。实现一个OrderService类如下：

@Transactional

public void changeProductCount(String id, ChangeProductCountCommand command) {

Order order = DAO.findById(id);

if (order.getStatus() == PAID) {

throw new OrderCannotBeModifiedException(id);

}

OrderItem orderItem = order.getOrderItem(command.getProductId());

orderItem.setCount(command.getCount());

order.setTotalPrice(calculateTotalPrice(order));

DAO.saveOrUpdate(order);

}

这种方式依然是一种面向过程的编程范式，违背了最基本的OO原则。另外的问题在于职责划分模糊不清，使本应该内聚在Order中的业务逻辑泄露到了其他地方(OrderService)，导致Order成为一个只是充当数据容器的[贫血模型(Anemic Model)](https://links.jianshu.com/go?to=https://en.wikipedia.org/wiki/Anemic_domain_model" \t "_blank)，而非真正意义上的领域模型。在项目持续演进的过程中，这些业务逻辑会分散在不同的Service类中，最终的结果是代码变得越来越难以理解进而逐渐丧失扩展能力。

#### 2. 基于事务脚本的实现

在上一种实现方式中，我们会发现领域对象(Order)存在的唯一目的其实是为了让ORM这样的工具能够一次性地持久化，在不使用ORM的情况下，领域对象甚至都没有必要存在。于是，此时的代码实现便退化成了[事务脚本(Transaction Script)](https://links.jianshu.com/go?to=https://martinfowler.com/eaaCatalog/transactionScript.html" \t "_blank)，也就是直接将Service类中计算出的结果直接保存到数据库(或者有时都没有Service类，直接通过SQL实现业务逻辑):

@Transactional

public void changeProductCount(String id, ChangeProductCountCommand command) {

OrderStatus orderStatus = DAO.getOrderStatus(id);

if (orderStatus == PAID) {

throw new OrderCannotBeModifiedException(id);

}

DAO.updateProductCount(id, command.getProductId(), command.getCount());

DAO.updateTotalPrice(id);

}

可以看到，DAO中多出了很多方法，此时的DAO不再只是对持久化的封装，而是也会包含业务逻辑。另外，DAO.updateTotalPrice(id)方法的实现中将直接调用SQL来实现Order总价的更新。与“Service+贫血模型”方式相似，事务脚本也存在业务逻辑分散的问题。

事实上，事务脚本并不是一种全然的反模式，在系统足够简单的情况下完全可以采用。但是：一方面“简单”这个度其实并不容易把握；另一方面软件系统通常会在不断的演进中加入更多的功能，使得原本简单的代码逐渐变得复杂。因此，事务脚本在实际的应用中使用得并不多。

#### 3. 基于领域对象的实现

在这种方式中，核心的业务逻辑被内聚在行为饱满的领域对象(Order)中，实现Order类如下：

public void changeProductCount(ProductId productId, int count) {

if (this.status == PAID) {

throw new OrderCannotBeModifiedException(this.id);

}

OrderItem orderItem = retrieveItem(productId);

orderItem.updateCount(count);

}

然后在Controller或者Service中，调用Order.changeProductCount()：

@PostMapping("/order/{id}/products")

public void changeProductCount(@PathVariable(name = "id") String id, @RequestBody @Valid ChangeProductCountCommand command) {

Order order = DAO.byId(orderId(id));

order.changeProductCount(ProductId.productId(command.getProductId()), command.getCount());

order.updateTotalPrice();

DAO.saveOrUpdate(order);

}

可以看到，所有业务（“检查Order状态”、“修改Product数量”以及“更新Order总价”）都被包含在了Order对象中，这些正是Order应该具有的职责。（不过示例代码中有个地方明显违背了内聚性原则，下文会讲到，作为悬念读者可以先行尝试着找一找）

事实上，这种方式与本文要讲的DDD战术模式已经很相近了，只是DDD抽象出了更多的概念与原则。

## 基于业务的分包

在本系列的[上一篇:Spring Boot项目模板](https://links.jianshu.com/go?to=todo" \t "_blank)文章中，其实我已经讲到了基于业务的分包，结合DDD的场景，这里再简要讨论一下。所谓基于业务分包即通过软件所实现的业务功能进行模块化划分，而不是从技术的角度划分(比如首先划分出service和infrastruture等包)。在DDD的战略设计中，我们关注于从一个宏观的视角俯视整个软件系统，然后通过一定的原则对系统进行子域和限界上下文的划分。在战术实践中，我们也通过类似的提纲挈领的方法进行整体的代码结构的规划，所采用的原则依然逃离不了“内聚性”和“职责分离”等基本原则。此时，首先映入眼帘的便是软件的分包。

在DDD中，聚合根(下文会讲到)是主要业务逻辑的承载体，也是“内聚性”原则的典型代表，因此通常的做法便是基于聚合根进行顶层包的划分。在示例电商项目中，有两个聚合根对象Order和Product，分别创建order包和product包，然后在各自的顶层包下再根据代码结构的复杂程度划分子包，比如对于product包：

└── product

├── CreateProductCommand.java

├── Product.java

├── ProductApplicationService.java

├── ProductController.java

├── ProductId.java

├── ProductNotFoundException.java

├── ProductRepository.java

└── representation

├── ProductRepresentationService.java

└── ProductSummaryRepresentation.java

可以看到，ProductRepository和ProductController等多数类都直接放在了product包下，而没有单独分包；但是展现类ProductSummaryRepresentation却做了单独分包。这里的原则是：在所有类已经被内聚在了product包下的情况下，如果代码结构足够的简单，那么没有必要再次进行子包的划分，ProductRepository和ProductController便是这种情况；而如果多个类需要做再次的内聚，那么需要另行分包，比如通过REST API接口返回Product数据时，代码中涉及到了两个对象ProductRepresentationService和ProductSummaryRepresentation，这两个对象是紧密关联的，因此将他们放在representation子包下。而对于更加复杂的Order，分包如下：

├── order

│ ├── OrderApplicationService.java

│ ├── OrderController.java

│ ├── OrderPaymentProxy.java

│ ├── OrderPaymentService.java

│ ├── OrderRepository.java

│ ├── command

│ │ ├── ChangeAddressDetailCommand.java

│ │ ├── CreateOrderCommand.java

│ │ ├── OrderItemCommand.java

│ │ ├── PayOrderCommand.java

│ │ └── UpdateProductCountCommand.java

│ ├── exception

│ │ ├── OrderCannotBeModifiedException.java

│ │ ├── OrderNotFoundException.java

│ │ ├── PaidPriceNotSameWithOrderPriceException.java

│ │ └── ProductNotInOrderException.java

│ ├── model

│ │ ├── Order.java

│ │ ├── OrderFactory.java

│ │ ├── OrderId.java

│ │ ├── OrderIdGenerator.java

│ │ ├── OrderItem.java

│ │ └── OrderStatus.java

│ └── representation

│ ├── OrderItemRepresentation.java

│ ├── OrderRepresentation.java

│ └── OrderRepresentationService.java

可以看到，我们专门创建了一个model包用于放置所有与Order聚合根相关的领域对象；另外，基于同类型相聚原则，创建command包和exception包分别用于放置请求类和异常类。

## 领域模型的门面——应用服务

UML中有用例(Use Case)的概念，表示的是软件向外提供业务功能的基本逻辑单元。在DDD中，由于业务被提到了第一优先级，那么自然地我们希望对业务的处理能够显现出来，为了达到这样的目的，DDD专门提供了一个名为应用服务(ApplicationService)的抽象层。ApplicationService采用了[门面模式](https://links.jianshu.com/go?to=https://en.wikipedia.org/wiki/Facade_pattern" \t "_blank)，作为领域模型向外提供业务功能的总出入口，就像酒店的前台处理客户的不同需求一样。



在编码实现业务功能时，通常用2种工作流程：

* 自底向上：先设计数据模型，比如关系型数据库的表结构，再实现业务逻辑。我在与不同的程序员结对编程的时候，总会是听到这么一句话：“让我先把数据库表的字段设计出来吧”。这种方式将关注点优先放在了技术性的数据模型上，而不是代表业务的领域模型，是DDD之反。
* 自顶向下：拿到一个业务需求，先与客户方确定好请求数据格式，再实现Controller和ApplicationService，然后实现领域模型(此时的领域模型通常已经被识别出来)，最后实现持久化。

在DDD实践中，自然应该采用自顶向下的实现方式。ApplicationService的实现遵循一个很简单的原则，即一个业务用例对应ApplicationService上的一个业务方法。比如，对于上文提到的“修改Order中Product的数量”业务需求实现如下：

实现OrderApplicationService：

@Transactional

public void changeProductCount(String id, ChangeProductCountCommand command) {

Order order = orderRepository.byId(orderId(id));

order.changeProductCount(ProductId.productId(command.getProductId()), command.getCount());

orderRepository.save(order);

}

OrderController调用OrderApplicationService：

@PostMapping("/{id}/products")

public void changeProductCount(@PathVariable(name = "id") String id, @RequestBody @Valid ChangeProductCountCommand command) {

orderApplicationService.changeProductCount(id, command);

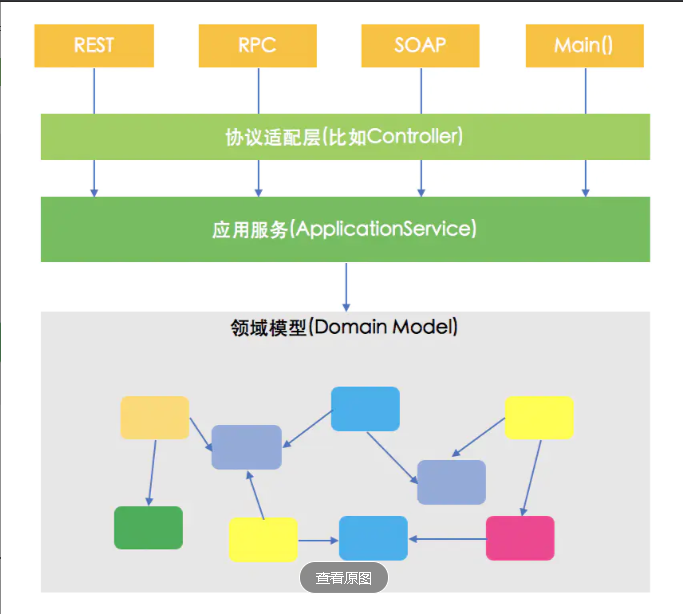
}

此时，order.changeProductCount()和orderRepository.save()都没有必要实现，但是由OrderController和OrderApplicationService所构成的业务处理的架子已经搭建好了。

可以看到，“修改Order中Product的数量”用例中的OrderApplicationService.changeProductCount()方法实现中只有不多的3行代码，然而，如此简单的ApplicationService却存在很多讲究。

ApplicationService需要遵循以下原则：

* 业务方法与业务用例一一对应：前面已经讲到，不再赘述。
* 业务方法与事务一一对应：也即每一个业务方法均构成了独立的事务边界，在本例中，OrderApplicationService.changeProductCount()方法标记有Spring的@Transactional注解，表示整个方法被封装到了一个事务中。
* 本身不应该包含业务逻辑：业务逻辑应该放在领域模型中实现，更准确的说是放在聚合根中实现，在本例中，order.changeProductCount()方法才是真正实现业务逻辑的地方，而ApplicationService只是作为代理调用order.changeProductCount()方法，因此，ApplicationService应该是很薄的一层。
* 与UI或通信协议无关：ApplicationService的定位并不是整个软件系统的门面，而是领域模型的门面，这意味着ApplicationService不应该处理诸如UI交互或者通信协议之类的技术细节。在本例中，Controller作为ApplicationService的调用者负责处理通信协议(HTTP)以及与客户端的直接交互。这种处理方式使得ApplicationService具有普适性，也即无论最终的调用方是HTTP的客户端，还是RPC的客户端，甚至一个Main函数，最终都统一通过ApplicationService才能访问到领域模型。
* 接受原始数据类型：ApplicationService作为领域模型的调用方，领域模型的实现细节对其来说应该是个黑盒子，因此ApplicationService不应该引用领域模型中的对象。此外，ApplicationService接受的请求对象中的数据仅仅用于描述本次业务请求本身，在能够满足业务需求的条件下应该尽量的简单。因此，ApplicationService通常处理一些比较原始的数据类型。在本例中，OrderApplicationService所接受的Order ID是Java原始的String类型，在调用领域模型中的Repository时，才被封装为OrderId对象。



应用服务(ApplicationService)是领域模型的门面

## 业务的载体——聚合根

接地气一点地讲，聚合根(Aggreate Root, AR)就是软件模型中那些最重要的以名词形式存在的领域对象，比如本文示例项目中的Order和Product。又比如，对于一个会员管理系统，会员(Member)便是一个聚合根；对于报销系统，报销单(Expense)便是一个聚合根；对于保险系统，保单(Policy)便是一个聚合根。聚合根是主要的业务逻辑载体，DDD中所有的战术实现都围绕着聚合根展开。

然而，并不是说领域模型中的所有名词都可以建模为聚合根。所谓“聚合”，顾名思义，即需要将领域中高度内聚的概念放到一起组成一个整体。至于哪些概念才能聚到一起，需要我们对业务本身有很深刻的认识，这也是为什么DDD强调开发团队需要和领域专家一起工作的原因。近年来流行起来的[事件风暴](https://links.jianshu.com/go?to=https://en.wikipedia.org/wiki/Event_storming" \t "_blank)建模活动，究其本意也是通过罗列出领域中发生的所有事件可以让我们全面的了解领域中的业务，进而识别出聚合根。

对于“更新Order中Product数量”用例，聚合根Order的实现如下：

public void changeProductCount(ProductId productId, int count) {

if (this.status == PAID) {

throw new OrderCannotBeModifiedException(this.id);

}

OrderItem orderItem = retrieveItem(productId);

orderItem.updateCount(count);

this.totalPrice = calculateTotalPrice();

}

private BigDecimal calculateTotalPrice() {

return items.stream()

.map(OrderItem::totalPrice)

.reduce(ZERO, BigDecimal::add);

}

private OrderItem retrieveItem(ProductId productId) {

return items.stream()

.filter(item -> item.getProductId().equals(productId))

.findFirst()

.orElseThrow(() -> new ProductNotInOrderException(productId, id));

}

在本例中，Order中的品项(orderItems)和总价(totalPrice)是密切相关的，orderItems的变化会直接导致totalPrice的变化，因此，这二者自然应该内聚在Order下。此外，totalPrice的变化是orderItems变化的必然结果，这种因果关系是业务驱动出来的，为了保证这种“必然”，我们需要在Order.changeProductCount()方法中同时实现“因”和“果”，也即聚合根应该保证业务上的一致性。在DDD中，业务上的一致性被称为[不变条件(Invariants)](https://links.jianshu.com/go?to=http://www.informit.com/articles/article.aspx?p=2020371" \t "_blank)。

还记得上文中提到的“违背内聚性的悬念”吗？当时调用Order上的业务方式如下：

.....

order.changeProductCount(ProductId.productId(command.getProductId()), command.getCount());

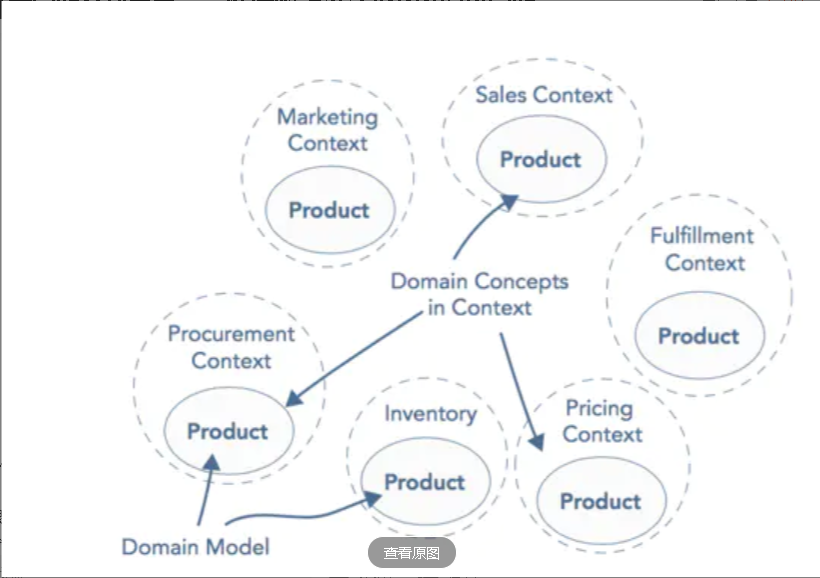
order.updateTotalPrice();

.....

为了实现“更新Order中Product数量”业务功能，这里先后调用了Order上的两个public方法changeProductCount()和updateTotalPrice()。虽然这种做法也能正确地实现业务逻辑，但是它将保证业务一致性的职责交给了Order的调用方(上文中的Controller)而不是Order自身，此时调用方需要确保在调用了changeProductCount()之后必须调用updateTotalPrice()方法，这一方面是Order中业务逻辑的泄露，另一方面调用方并不承担这样的职责，而Order才最应该承担这样的职责。

对内聚性的追求会自然地延伸出聚合根的边界。在DDD的战略设计中，我们已经通过限界上下文的划分将一个大的软件系统拆分为了不同的“模块”，在这样的前提下，再在某个限界上下文中来讨论内聚性将比在[大泥球](https://links.jianshu.com/go?to=https://en.wikipedia.org/wiki/Big_ball_of_mud" \t "_blank)系统中讨论变得简单得多。

对聚合根的设计需要提防[上帝对象(God Object)](https://links.jianshu.com/go?to=https://en.wikipedia.org/wiki/God_object" \t "_blank)，也即用一个大而全的领域对象来实现所有的业务功能。上帝对象的背后存在着一种表面上看似合理的逻辑：既然要内聚，那么让我们把所有相关的东西都聚到一起吧，比如用一个Product类来应付所有的业务场景，包括订单、物流、发票等等。这种机械的方式看似内聚，实则恰恰是内聚性的反面。要解决这样的问题依然需要求助于限界上下文，不同限界上下文使用各自的[通用语言(Ubiquitous Language)](https://links.jianshu.com/go?to=https://martinfowler.com/bliki/UbiquitousLanguage.html" \t "_blank)，通用语言要求一个业务概念不应该有二义性，在这样的原则下，不同的限界上下文可能都有自己的Product类，虽然名字相同，却体现着不同的业务。



不同的限界上下文中都有各自的Product，有些Product是聚合根，有些不是

除了内聚性和一致性，聚合根还有以下特征：

聚合根的实现应该与框架无关：既然DDD讲求业务复杂度和技术复杂度的分离，那么作为业务主要载体的聚合根应该尽量少地引用技术框架级别的设施，最好是[POJO](https://links.jianshu.com/go?to=https://en.wikipedia.org/wiki/Plain_old_Java_object" \t "_blank)。试想一下，如果你的项目哪天需要从Spring迁移到[Play](https://links.jianshu.com/go?to=https://www.playframework.com/" \t "_blank)，而你可以自信地给老板说，直接将核心Java代码拷贝过去即可，这将是一种多么美妙的体验。又或者说，很多时候技术框架会有“大步”的升级，这种升级会导致框架中API的变化并且不再支持向后兼容，此时如果我们的领域模与框架无关，那么便可做到在框架升级的过程中幸免于难。

聚合根之间的引用通过ID完成：在聚合根边界设计合理的情况下，一次业务用例只会更新一个聚合根，此时你在该聚合根中去引用另外聚合根的整体有什么好处呢？在本文示例中，一个Order下的OrderItem引用了ProductId，而不是整个Product。

聚合根内部的所有变更都必须通过聚合根完成：为了保证聚合根的一致性，同时避免聚合根内部逻辑向外泄露，客户方只能将整个聚合根作为统一调用入口。

如果一个事务需要更新多个聚合根，首先思考一下自己的聚合根边界处理是否出了问题，因为在设计合理的情况下通常不会出现一个事务更新多个聚合根的场景。如果这种情况的确是业务所需，那么考虑引入[消息机制](https://links.jianshu.com/go?to=https://en.wikipedia.org/wiki/Message_queue" \t "_blank)和[事件驱动架构](https://links.jianshu.com/go?to=https://en.wikipedia.org/wiki/Event-driven_architecture" \t "_blank)，保证一个事务只更新一个聚合根，然后通过消息机制异步更新其他聚合根。

聚合根不应该引用基础设施。

外界不应该持有聚合根内部的数据结构。

尽量使用小聚合。

## 实体 vs 值对象

软件模型中存在实体对象(Entity)和值对象(Value Object)之说，这种划分方式事实上并不是DDD的专属，但是在DDD中我们非常强调这两者之间的区别。

实体对象表示的是具有一定生命周期并且拥有全局唯一标识(ID)的对象，比如本文中的Order和Product，而值对象表示用于起描述性作用的，没有唯一标识的对象，比如Address对象。

聚合根一定是实体对象，但是并不是所有实体对象都是聚合根，同时聚合根还可以拥有其他子实体对象。聚合根的ID在整个软件系统中全局唯一，而其下的子实体对象的ID只需在单个聚合根下唯一即可。 在本文示例项目中，OrderItem是聚合根Order下的子实体对象：

public class OrderItem {

private ProductId productId;

private int count;

private BigDecimal itemPrice;}

可以看到，虽然OrderItem使用了ProductID作为ID，但是此时我们并没有享受ProductID的全局唯一性，事实上多个Order可以包含相同ProductID的OrderItem，也即多个订单可以包含相同的产品。

区分实体和值对象的一个很重要的原则便是根据相等性来判断，实体对象的相等性是通过ID来完成的，对于两个实体，如果他们的所有属性均相同，但是ID不同，那么他们依然两个不同的实体，就像一对长得一模一样的双胞胎，他们依然是两个不同的自然人。对于值对象来说，相等性的判断是通过属性字段来完成的。比如，订单下的送货地址Address对象便是一个典型的值对象：

public class Address {

private String province;

private String city;

private String detail;

@Override

public boolean equals(Object o) {

if (this == o) {

return true;

}

if (o == null || getClass() != o.getClass()) {

return false;

}

Address address = (Address) o;

return province.equals(address.province) &&

city.equals(address.city) &&

detail.equals(address.detail);

}

@Override

public int hashCode() {

return Objects.hash(province, city, detail);

}

}

在Address的equals()方法中，通过判断Address所包含的所有属性(province，city，detail)来决定两个Address的相等性。

值对象还有一个特点是[不变的(Immutable)](https://links.jianshu.com/go?to=https://en.wikipedia.org/wiki/Immutable_object" \t "_blank)，也就说一个值对象一旦被创建出来了便不能对其进行变更，如果要变更，必须重新创建一个新的值对象整体替换原有的。比如，示例项目有一个业务需求：

在订单未支付的情况下，可以修改订单送货地址的详细地址(detail)

由于Address是Order聚合根中的一个对象，对Address的更改只能通过Order完成，在Order中实现changeAddressDetail()方法：

public void changeAddressDetail(String detail) {

if (this.status == PAID) {

throw new OrderCannotBeModifiedException(this.id);

}

this.address = this.address.changeDetailTo(detail);

}

可以看到，通过调用address.changeDetailTo()方法，我们获取到了一个全新的Address对象，然后将新的Address对象整体赋值给address属性。此时Address.changeDetailTo()的实现如下：

public Address changeDetailTo(String detail) {

return new Address(this.province, this.city, detail);

}

这里的changeDetailTo()方法使用了新的详细地址detail和未发生变更的province、city重新创建出了一个Address对象。

值对象的不变性使得程序的逻辑变得更加简单，你不用去维护复杂的状态信息，需要的时候创建，不要的时候直接扔掉即可，使得值对象就像程序中的过客一样。在DDD建模中，一种受推崇的做法便是将业务概念尽量建模为值对象。

对于OrderItem来说，由于我们的业务需要对OrderItem的数量进行修改，也即拥有生命周期的意味，因此本文将OrderItem建模为了实体对象。但是，如果没有这样的业务需求，那么将OrderItem建模为值对象应该更合适一些。

另外，需要指明的是，实体和值对象的划分并不是一成不变的，而应该根据所处的限界上下文来界定，相同一个业务名词，在一个限界上下文中可能是实体，在另外的限界上下文中可能是值对象。比如，订单Order在采购上下文中应该建模为一个实体，但是在物流上下文中便可建模为一个值对象。

## 聚合根的家——资源库

通俗点讲，资源库(Repository)就是用来持久化聚合根的。从技术上讲，Repository和DAO所扮演的角色相似，不过DAO的设计初衷只是对数据库的一层很薄的封装，而Repository是更偏向于领域模型。另外，在所有的领域对象中，只有聚合根才“配得上”拥有Repository，而DAO没有这种约束。

实现Order的资源库OrderRepository如下：

public void save(Order order) {

String sql = "INSERT INTO ORDERS (ID, JSON\_CONTENT) VALUES (:id, :json) " +

"ON DUPLICATE KEY UPDATE JSON\_CONTENT=:json;";

Map<String, String> paramMap = of("id", order.getId().toString(), "json", objectMapper.writeValueAsString(order));

jdbcTemplate.update(sql, paramMap);

}

public Order byId(OrderId id) {

try {

String sql = "SELECT JSON\_CONTENT FROM ORDERS WHERE ID=:id;";

return jdbcTemplate.queryForObject(sql, of("id", id.toString()), mapper());

} catch (EmptyResultDataAccessException e) {

throw new OrderNotFoundException(id);

}

}

在OrderRepository中，我们只定义了save()和byId()方法，分别用于保存/更新聚合根和通过ID获取聚合根。这两个方法是Repository中最常见的方法，有的DDD实践者甚至认为一个纯粹的Repository只应该包含这两个方法。

读到这里，你可能会有些疑问：为什么OrderRepository中没有更新和查询等方法？事实上，Repository所扮演的角色只是向领域模型提供聚合根而已，就像一个聚合根的“容器”一样，这个“容器”本身并不关心客户端对聚合根的操作到底是新增还是更新，你给一个聚合根对象，Repository只是负责将其状态从计算机的内存同步到持久化机制中，从这个角度讲，Repository只需要一个类似save()的方法便可完成同步操作。当然，这个是从概念的出发点得出的设计结果，在技术层面，新增和更新还是需要区别对待，比如SQL语句有insert和update之分，只是我们将这样的技术细节隐藏在了save()方法中，客户方并无需知道这些细节。在本例中，我们通过MySQL的[ON DUPLICATE KEY UPDATE](https://links.jianshu.com/go?to=https://dev.mysql.com/doc/refman/8.0/en/insert-on-duplicate.html" \t "_blank)特性同时处理对数据库的新增和更新操作。当然，我们也可以通过编程判断聚合根在数据库中是否已经存在，如果存在则update，否则insert。另外，诸如Hibernate这样的持久化框架自动提供saveOrUpate()方法可以直接用于对聚合根的持久化。

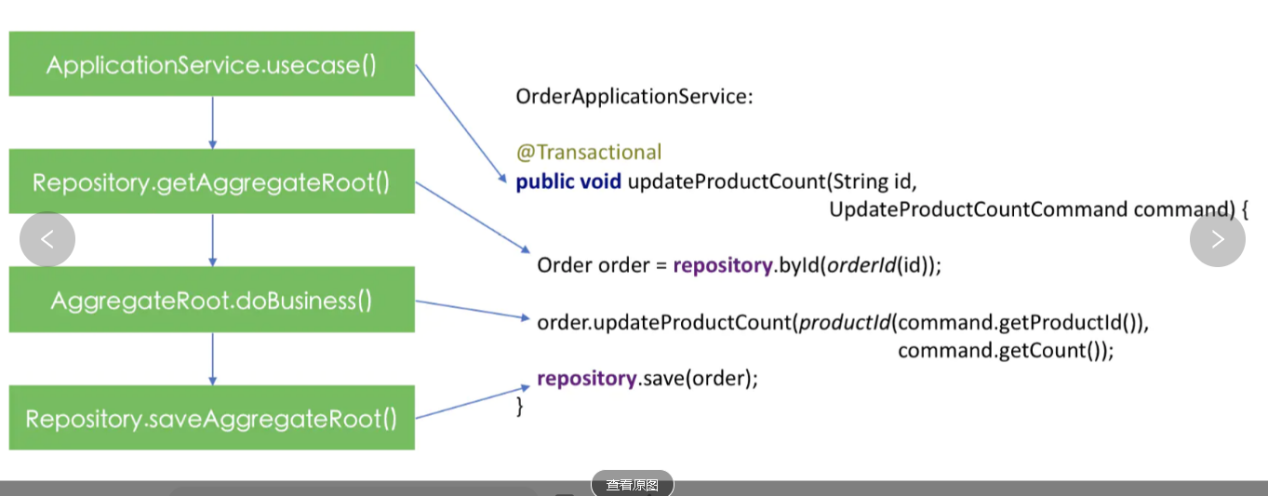
对于查询功能来说，在Repository中实现查询本无不合理之处，然而项目的演进可能导致Repository中充斥着大量的查询代码“喧宾夺主”似的掩盖了Repository原本的目的。事实上，DDD中读操作和写操作是两种很不一样的过程，笔者的建议是尽量将此二者分开实现，由此查询功能将从Repository中分离出去，在下文中我将详细讲到。

在本例中，我们在技术实现上使用到了Spring的JdbcTemplate和JSON格式持久化Order聚合根，其实Repository并不与某种持久化机制绑定，一个被抽象出来的Repository向外暴露的功能“接口”始终是向领域模型提供聚合根对象，就像“聚合根的家”一样。

好了，至此让我们来做个回顾，上文中我们以“更新Order中的Product数量”业务需求为例，讲到了应用服务、聚合根和资源库，对该业务需求的处理流程体现了DDD处理业务需求的最常见最典型的形式：

**应用服务作为总体协调者，先通过资源库获取到聚合根，然后调用聚合根中的业务方法，最后再次调用资源库保存聚合根。**

流程示意图如下：



DDD处理业务流程的典型流程

## 创生之柱——工厂

稍微提炼一下，我们便知道软件里面的写操作要么是修改既有数据，要么是新建数据。对于前者，DDD给出的答案已经在上文中讲到，接下来我们讲讲在DDD中如何新建聚合根。

创建聚合根通常通过设计模式中的[工厂(Factory)模式](https://links.jianshu.com/go?to=https://en.wikipedia.org/wiki/Factory_method_pattern" \t "_blank)完成，这一方面可以享受到工厂模式本身的好处，另一方面，DDD中的Factory还具有将“聚合根的创建逻辑”显现出来的效果。



创生之柱——恒星诞生的地方，距地球约6500光年，由哈勃太空望远镜于1995年拍摄

聚合根的创建过程可简单可复杂，有时可能直接调用构造函数即可，而有时却存在一个复杂的构造流程，比如需要调用其他系统获取数据等。通常来讲，Factory有两种实现方式：

* 直接在聚合根中实现Factory方法，常用于简单的创建过程
* 独立的Factory类，用于有一定复杂度的创建过程，或者创建逻辑不适合放在聚合根上

让我们先演示一下简单的Factory方法，在示例订单系统中，有个业务用例是“创建Product”：

创建Product，属性包括名称(name)，描述(description)和单价(price)，ProductId为UUID

在Product类中实现工厂方法create()：

public static Product create(String name, String description, BigDecimal price) {

return new Product(name, description, price);

}

private Product(String name, String description, BigDecimal price) {

this.id = ProductId.newProductId();

this.name = name;

this.description = description;

this.price = price;

this.createdAt = Instant.now();

}

这里，Product中的create()方法并不包含创建逻辑，而是将创建过程直接代理给了Product的构造函数。你可能觉得这个create()方法有些多此一举，然而这种做法的初衷依然是：我们希望将聚合根的创建逻辑突显出来。构造函数本身是一个非常技术的东西，任何地方只要涉及到在计算机内存中新建对象都需要使用构造函数，无论创建的初始原因是业务需要，还是从数据库加载，亦或是从JSON数据反序列化。因此程序中往往存在多个构造函数用于不同的场景，而为了将业务上的创建与技术上的创建区别开来，我们引入了create()方法用于表示业务上的创建过程。

“创建Product”所设计到的Factory的确简单，让我们再来看看另外一个例子：“创建Order”：

创建Order，包含用户选择的Product及其数量，OrderId必须调用第三方的OrderIdGenerator获取

这里的OrderIdGenerator是具有服务性质的对象(即下文中的领域服务)，在DDD中，聚合根通常不会引用其他服务类。另外，调用OrderIdGenerator生成ID应该是一个业务细节，如前文所讲，这种细节不应该放在ApplicationService中。此时，可以通过Factory类来完成Order的创建：

@Componentpublic class OrderFactory {

private final OrderIdGenerator idGenerator;

public OrderFactory(OrderIdGenerator idGenerator) {

this.idGenerator = idGenerator;

}

public Order create(List<OrderItem> items, Address address) {

OrderId orderId = idGenerator.generate();

return Order.create(orderId, items, address);

}}

## 必要的妥协——领域服务

前面我们提到，聚合根是业务逻辑的主要载体，也就是说业务逻辑的实现代码应该尽量地放在聚合根或者聚合根的边界之内。但有时，有些业务逻辑并不适合于放在聚合根上，比如前文的OrderIdGenerator便是如此，在这种“迫不得已”的情况下，我们引入领域服务(Domain Service)。还是先来看一个列子，对于Order的支付有以下业务用例：

通过支付网关OrderPaymentService完成Order的支付。

在OrderApplicationService中，直接调用领域服务OrderPaymentService：

@Transactional

public void pay(String id, PayOrderCommand command) {

Order order = orderRepository.byId(orderId(id));

orderPaymentService.pay(order, command.getPaidPrice());

orderRepository.save(order);

}

然后实现OrderPaymentService：

public void pay(Order order, BigDecimal paidPrice) {

order.pay(paidPrice);

paymentProxy.pay(order.getId(), paidPrice);

}

这里的PaymentProxy与OrderIdGenerator相似，并不适合于放在Order中。可以看到，在OrderApplicationService中，我们并没有直接调用Order中的业务方法，而是先调用OrderPaymentService.pay()，然后在OrderPaymentService.pay()中完成调用支付网关PaymentProxy.pay()这样的业务细节。

到此，再来反观在通常的实践中我们编写的Service类，事实上这些Servcie类将DDD中的ApplicationService和DomainService糅合在了一起，比如在”基于Service + 贫血模型”的实现“小节中的OrderService便是如此。在DDD中，ApplicationService和DomainService是两个很不一样的概念，前者是必须有的DDD组件，而后者只是一种妥协的结果，因此程序中的DomainService应该越少越好。

## Command对象

通常来说，DDD中的写操作并不需要向客户端返回数据，在某些情况下(比如新建聚合根)可以返回一个聚合根的ID，这意味着ApplicationService或者聚合根中的写操作方法通常返回void即可。比如，对于OrderApplicationService，各个方法签名如下：

public OrderId createOrder(CreateOrderCommand command) ;

public void changeProductCount(String id, ChangeProductCountCommand command) ;

public void pay(String id, PayOrderCommand command) ;

public void changeAddressDetail(String id, String detail) ;

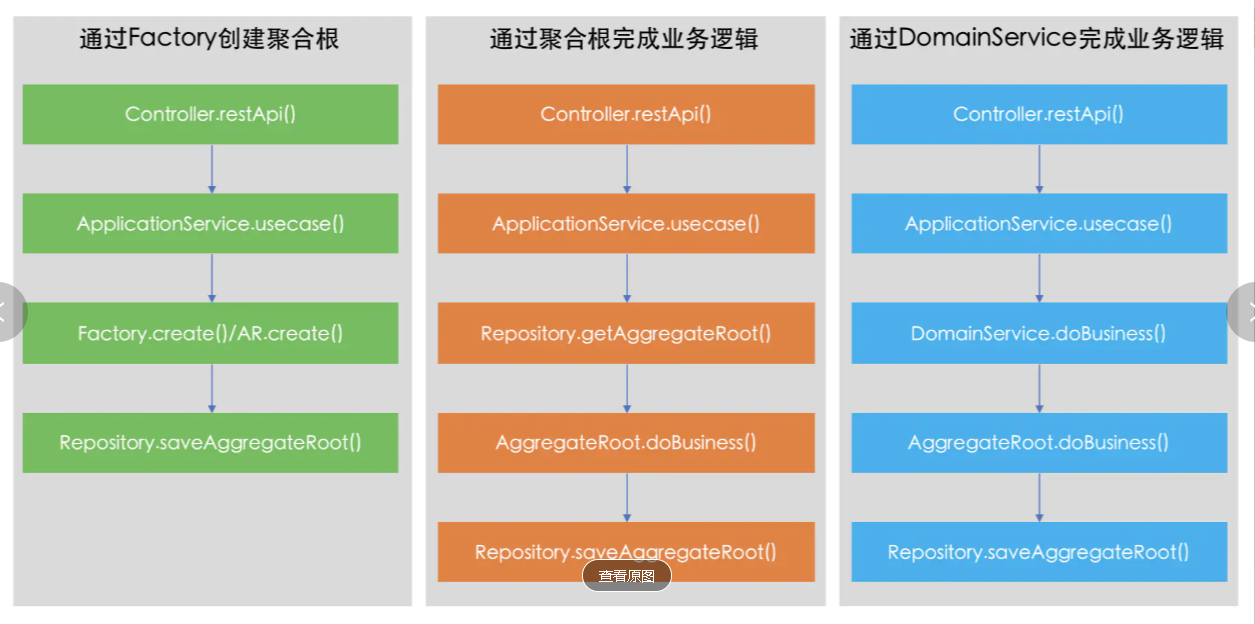
可以看到，在多数情况下我们使用了后缀为Command的对象传给ApplicationService，比如CreateOrderCommand和ChangeProductCountCommand。Command即命令的意思，也即写操作表示的是外部向领域模型发起的一次命令操作。事实上，从技术上讲，Command对象只是一种类型的DTO对象，它封装了客户端发过来的请求数据。在Controller中所接收的所有写操作都需要通过Command进行包装，在Command比较简单(比如只有1-2个字段)的情况下Controller可以将Command解开之后，将其中的数据直接传递给ApplicationService，比如changeAddressDetail()便是如此；而在Command中数据字段比较多时，可以直接将Command对象传递给ApplicationService。当然，这并不是DDD中需要严格遵循的一个原则，比如无论Command的简繁程度，统一将所有Command从Controller传递给ApplicationService，也不存在太大的问题，更多的只是一个编码习惯上的选择。不过有一点需要强调，即前文提到的“ApplicationService需要接受原始数据类型而不是领域模型中的对象”，在这里意味着Command对象中也应该包含原始的数据类型。

统一使用Command对象还有个好处是，我们通过查找所有后缀为Command的对象，便可以概览性地了解软件系统向外提供的业务功能。

阶段性小结一下，以上我们主要围绕着软件的“写操作”在DDD中的实现进行讨论，并且讲到了3种场景，分别是：

* 通过聚合根完成业务请求
* 通过Factory完成聚合根的创建
* 通过DomainService完成业务请求

以上3种场景大致上涵盖了DDD完成业务写操作的基本方面，总结下来3句话：创建聚合根通过Factory完成；业务逻辑优先在聚合根边界内完成；聚合根中不合适放置的业务逻辑才考虑放到DomainService中。



DDD实现软件"写操作"的3种场景

## DDD中的读操作

软件中的读模型和写模型是很不一样的，我们通常所讲的业务逻辑更多的时候是在写操作过程中需要关注的东西，而读操作更多关注的是如何向客户方返回恰当的数据展现。

在DDD的写操作中，我们需要严格地按照“应用服务 -> 聚合根 -> 资源库”的结构进行编码，而在读操作中，采用与写操作相同的结构有时不但得不到好处，反而使整个过程变得冗繁。这里介绍3种读操作的方式：

* 基于领域模型的读操作
* 基于数据模型的读操作
* CQRS

首先，无论哪种读操作方式，都需要遵循一个原则：领域模型中的对象不能直接返回给客户端，因为这样领域模型的内部便暴露给了外界，而对领域模型的修改将直接影响到客户端。因此，在DDD中我们通常为读操作专门创建相应的模型用于数据展现。在写操作中，我们通过Command后缀进行请求数据的统一，在读操作中，我们通过Representation后缀进行展现数据的统一，这里的Representation也即REST中的“R”。

#### 基于领域模型的读操作

这种方式将读模型和写模型糅合到一起，先通过资源库获取到领域模型，然后将其转换为Representation对象，这也是当前被大量使用的方式，比如对于“获取Order详情的接口”，OrderApplicationService实现如下：

@Transactional(readOnly = true)

public OrderRepresentation byId(String id) {

Order order = orderRepository.byId(orderId(id));

return orderRepresentationService.toRepresentation(order);

}

我们先通过orderRepository.byId()获取到Order聚合根对象，然后调用orderRepresentationService.toRepresentation()将Order转换为展现对象OrderRepresentation，OrderRepresentationService.toRepresentation()实现如下：

public OrderRepresentation toRepresentation(Order order) {

List<OrderItemRepresentation> itemRepresentations = order.getItems().stream()

.map(orderItem -> new OrderItemRepresentation(orderItem.getProductId().toString(),

orderItem.getCount(),

orderItem.getItemPrice()))

.collect(Collectors.toList());

return new OrderRepresentation(order.getId().toString(),

itemRepresentations,

order.getTotalPrice(),

order.getStatus(),

order.getCreatedAt());

}

这种方式的优点是非常直接明了，也不用创建新的数据读取机制，直接使用Repository读取数据即可。然而缺点也很明显：一是读操作完全束缚于聚合根的边界划分，比如，如果客户端需要同时获取Order及其所包含的Product，那么我们需要同时将Order聚合根和Product聚合根加载到内存再做转换操作，这种方式既繁琐又低效；二是在读操作中，通常需要基于不同的查询条件返回数据，比如通过Order的日期进行查询或者通过Product的名称进行查询等，这样导致的结果是Repository上处理了太多的查询逻辑，变得越来越复杂，也逐渐偏离了Repository本应该承担的职责。

#### 基于数据模型的读操作

这种方式绕开了资源库和聚合，直接从数据库中读取客户端所需要的数据，此时写操作和读操作共享的只是数据库。比如，对于“获取Product列表”接口，通过一个专门的ProductRepresentationService直接从数据库中读取数据：

@Transactional(readOnly = true)

public PagedResource<ProductSummaryRepresentation> listProducts(int pageIndex, int pageSize) {

MapSqlParameterSource parameters = new MapSqlParameterSource();

parameters.addValue("limit", pageSize);

parameters.addValue("offset", (pageIndex - 1) \* pageSize);

List<ProductSummaryRepresentation> products = jdbcTemplate.query(SELECT\_SQL, parameters,

(rs, rowNum) -> new ProductSummaryRepresentation(rs.getString("ID"),

rs.getString("NAME"),

rs.getBigDecimal("PRICE")));

int total = jdbcTemplate.queryForObject(COUNT\_SQL, newHashMap(), Integer.class);

return PagedResource.of(total, pageIndex, products);

}

然后在Controller中直接返回：

@GetMapping

public PagedResource<ProductSummaryRepresentation> pagedProducts(@RequestParam(required = false, defaultValue = "1") int pageIndex,

@RequestParam(required = false, defaultValue = "10") int pageSize) {

return productRepresentationService.listProducts(pageIndex, pageSize);

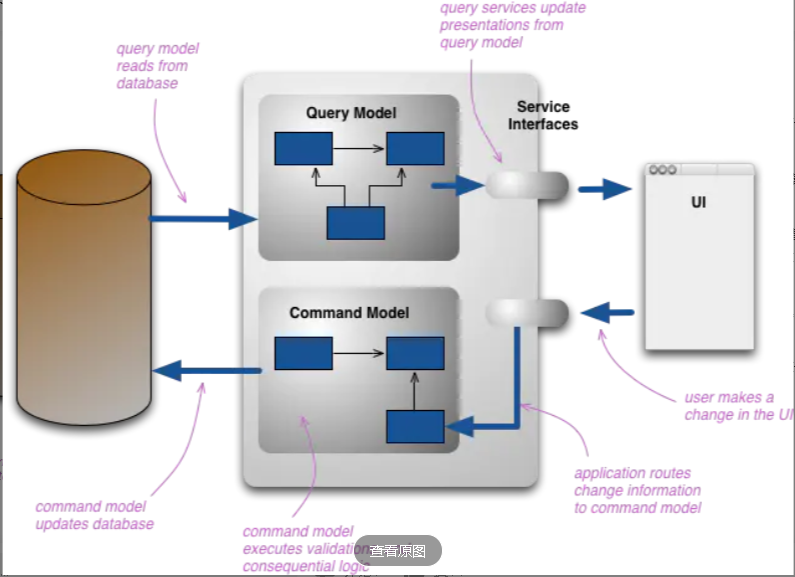
}

可以看到，真个过程并没有使用到ProductRepository和Product，而是将SQL获取到的数据直接新建为ProductSummaryRepresentation对象。

这种方式的优点是读操作的过程不用囿于领域模型，而是基于读操作本身的需求直接获取需要的数据即可，一方面简化了整个流程，另一方面大大提升了性能。但是，由于读操作和写操作共享了数据库，而此时的数据库主要是对应于聚合根的结构创建的，因此读操作依然会受到写操作的数据模型的牵制。不过这种方式是一种很好的折中，微软也提倡过这种方式，更多细节请参考[微软官网](https://links.jianshu.com/go?to=https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/microservice-ddd-cqrs-patterns/apply-simplified-microservice-cqrs-ddd-patterns" \t "_blank)。

#### CQRS

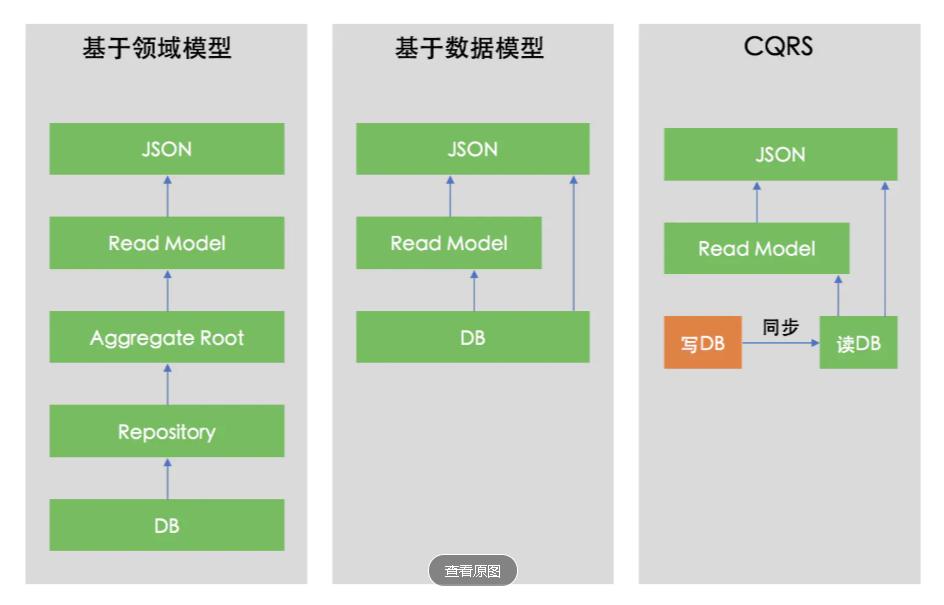
[CQRS](https://links.jianshu.com/go?to=https://martinfowler.com/bliki/CQRS.html" \t "_blank)(Command Query Responsibility Segregation)，即命令查询职责分离，这里的命令可以理解为写操作，而查询可以理解为读操作。与“基于数据模型的读操作”不同的是，在CQRS中写操作和读操作使用了不同的数据库，数据从写模型数据库同步到读模型数据库，通常通过[领域事件](https://links.jianshu.com/go?to=https://martinfowler.com/eaaDev/DomainEvent.html" \t "_blank)的形式同步变更信息。



CQRS架构

这样一来，读操作便可以根据自身所需独立设计数据结构，而不用受写模型数据结构的牵制。CQRS本身是一个很大的话题，已经超出了本文的范围，读者可以自行研究。

到此，DDD中的读操作可以大致分为3种实现方式：



DDD读操作的3种实现方式

## 总结

本文主要介绍了DDD中的应用服务、聚合、资源库和工厂等概念以及与它们相关的编码实践，然后着重讲到了软件的读写操作在DDD中的实现方式，其中写操作的3种场景为：

* 通过聚合根完成业务请求，这是DDD完成业务请求的典型方式
* 通过Factory完成聚合根的创建，用于创建聚合根
* 通过DomainService完成业务请求，当业务放在聚合根中不合适时才考虑放在DomainService中

对于读操作，同样给出了3种方式：

* 基于领域模型的读操作(读写操作糅合在了一起，不推荐)
* 基于数据模型的读操作(绕过聚合根和资源库，直接返回数据，推荐)
* CQRS(读写操作分别使用不同的数据库)

以上“3读3写”基本上涵盖了程序员完成业务功能的日常开发之所需，原来DDD就这么简单，不是吗？

作者：无知者云  
链接：https://www.jianshu.com/p/84f1d922a6d4  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

# **后端开发实践系列之三——事件驱动架构(EDA)编码实践**

在本系列的前两篇文章中，笔者分别讲到了[后端项目的代码模板](https://links.jianshu.com/go?to=https://insights.thoughtworks.cn/backend-development-iteration0/" \t "_blank)和[DDD编码实践](https://links.jianshu.com/go?to=https://insights.thoughtworks.cn/backend-development-ddd/" \t "_blank)，在本文中，我将继续以编码实践的方式分享如何落地事件驱动架构。



单纯地讲事件驱动架构(Event Driven Architecture, EDA)，那是几十年前就出现了的话题；单纯地讲领域事件，那也是这些年被大量[提及并讨论](https://links.jianshu.com/go?to=https://insights.thoughtworks.cn/category/ddd/" \t "_blank)得快熟透了的软件用语。然而，就笔者的观察看，事件驱动架构远没有想象中那样普遍地被开发团队所接受。即便搞微服务的人都知道除了同步的HTTP还有异步的消息机制，即便搞DDD的人都知道领域事件是其中的一等公民，事件驱动架构所带来的优点并没有相应地转化为软件从业者的青睐。

我尝试着去思考其中的原因，总结出了两点：第一是事件驱动可能是客观世界的运作方式，但不是人的自然思考问题的方式；第二是事件驱动架构在给软件带来好处的同时，又会增加额外的复杂性，比如调试的困难性，又比如并不直观的最终一致性。

当然，事实上有不少软件项目都使用了消息队列，但是这里需要明确的是，对消息队列的使用并不意味着你的项目就一定是事件驱动架构，很多项目只是由于技术方面的驱动，小范围地采用了某些消息队列（比如RabbitMQ和Kafka等）的产品而已。偌大一个系统，如果你的消息队列只是用作邮件发送的通知，那么这样系统自然谈不上采用了事件驱动架构。

放到当下，微服务兴起，DDD重现，在采用事件驱动架构时， 我们需要考虑业务的建模、领域事件的设计、DDD的约束、限界上下文的边界以及更多技术方面的因素，这一个系统工程应该如何从头到尾的落地，是需要经过思考和推敲的。还是那句话，有讲究的编程并不是一件易事。

诚然，用好事件驱动架构存在实践上的难处，然而它的[优点](https://links.jianshu.com/go?to=https://wso2.com/whitepapers/event-driven-architecture-the-path-to-increased-agility-and-high-expandability/" \t "_blank)也委实诱人，本文希望形成一定的“条理”和“套路”，让事件驱动架构能够更简单的落地。

本文主要分为两大部分，第一部分独立于具体的消息队列实现来讲解通用的对领域事件的建模，第二部分以一个真实的微服务系统为例，采用RabbitMQ作为消息队列，并以此分享完整的事件驱动架构落地实践。

本文以DDD为基础进行编码，其中会涉及到DDD中的不少概念，比如聚合根、资源库和应用服务等，对DDD不熟悉的读者可以参考笔者的[DDD编码实践文章](https://links.jianshu.com/go?to=https://insights.thoughtworks.cn/backend-development-ddd/" \t "_blank)。

本文的示例代码请参考github上的[e-commerce-sample](https://links.jianshu.com/go?to=https://github.com/e-commerce-sample" \t "_blank)项目。

## 第一部分：领域事件的建模

领域事件是DDD中的一个概念，表示的是在一个领域中所发生的一次对业务有价值的事情，落到技术层面就是在一个业务实体对象(通常来说是聚合根)的状态发生了变化之后需要发出一个领域事件。虽然事件驱动架构中的“事件”不一定指“领域事件”，但本文由于密切结合DDD，因此当提到事件时，我们特指“领域事件”。

#### 创建领域事件

关于领域事件的基础知识，请参考笔者的[在微服务中使用领域事件](https://links.jianshu.com/go?to=https://insights.thoughtworks.cn/use-domain-events-in-microservices/" \t "_blank)文章，本文直接进入编码实践环节。

在建模领域事件时，首先需要记录事件的一些通用信息，比如唯一标识ID和创建时间等，为此创建事件基类DomainEvent：

public abstract class DomainEvent {

private final String \_id;

private final DomainEventType \_type;

private final Instant \_createdAt;}

在DDD场景下，领域事件一般随着聚合根状态的更新而产生，另外，在事件的消费方，有时我们希望监听发生在某个聚合根下的所有事件，为此笔者建议为每一个聚合根对象创建相应的事件基类，其中包含聚合根的ID，比如对于订单(Order)类，创建OrderEvent：

public abstract class OrderEvent extends DomainEvent {

private final String orderId;}

然后对于实际的Order事件，统一继承自OrderEvent，比如对于创建订单的OrderCreatedEvent事件：

public class OrderCreatedEvent extends OrderEvent {

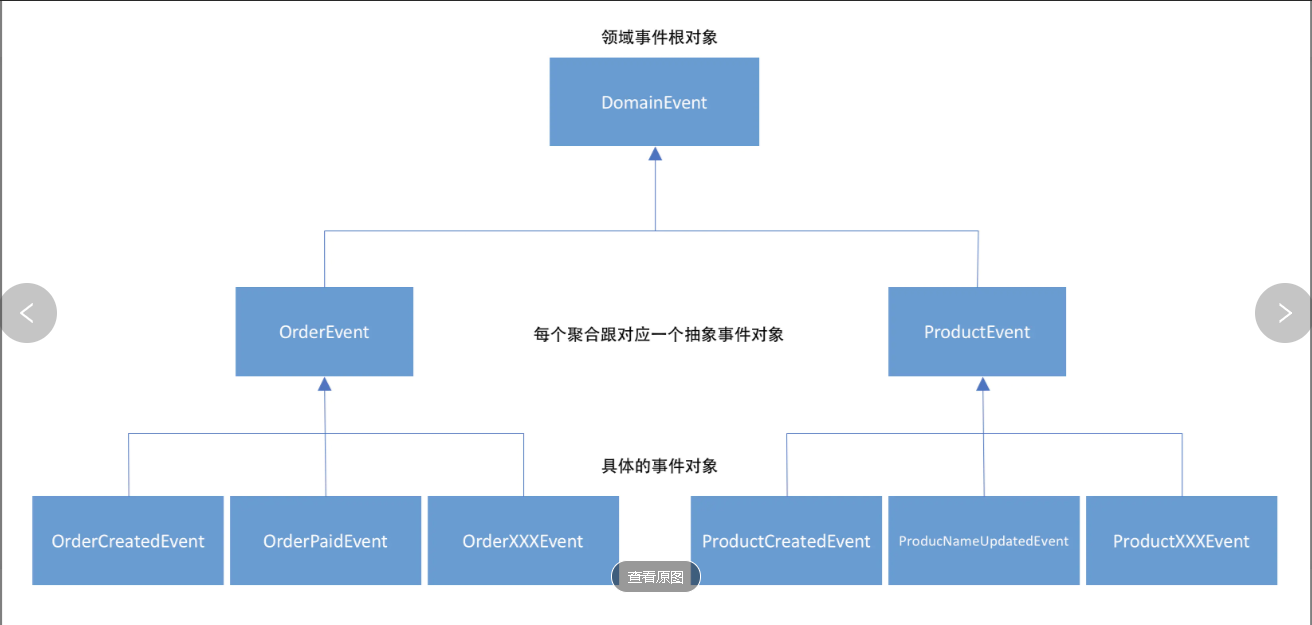
private final BigDecimal price;

private final Address address;

private final List<OrderItem> items;

private final Instant createdAt;}

领域事件的继承链如下：



领域事件继承链

在创建领域事件时，需要注意2点：

* 领域事件本身应该是不变的(Immutable)；
* 领域事件应该携带与事件发生时相关的上下文数据信息，但是并不是整个聚合根的状态数据，例如，在创建订单时可以携带订单的基本信息，而对于产品(Product)名称更新的ProductNameUpdatedEvent事件，则应该同时包含更新前后的产品名称：

public class ProductNameUpdatedEvent extends ProductEvent {

private String oldName; //更新前的名称

private String newName; // 更新后的名称}

#### 发布领域事件

发布领域事件有多种方式，比如可以在应用服务(ApplicationService)中发布，也可以在资源库(Repository)中发布，还可以引入事件表的方式，这3种发布方式的详细比较可以参考笔者的[在微服务中使用领域事件](https://links.jianshu.com/go?to=https://insights.thoughtworks.cn/use-domain-events-in-microservices/" \t "_blank)文章。笔者建议采用事件表方式，这里展开讨论一下。

通常的业务处理过程都会更新数据库然后发布领域事件，这里一个比较重要的点是：我们需要保证数据库更新和事件发布之间的原子性，也即要么二者都成功，要么都失败。在传统的实践方式中，[全局事务(Global Transaction/XA Transaction)](https://links.jianshu.com/go?to=https://en.wikipedia.org/wiki/X/Open_XA" \t "_blank)通常用于解决此类问题。然而，全局事务本身的效率是很低的，另外，一些技术框架并不提供对全局事务的支持。当前，一种比较受推崇的方式是引入事件表，其流程大致如下：

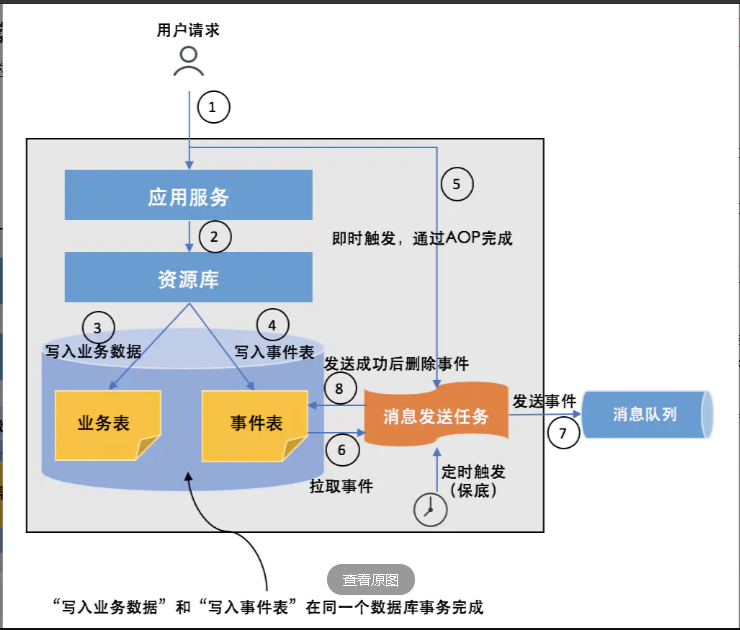
在更新业务表的同时，将领域事件一并保存到数据库的事件表中，此时业务表和事件表在同一个本地事务中，即保证了原子性，又保证了效率。

在后台开启一个任务，将事件表中的事件发布到消息队列中，发送成功之后删除掉事件。

但是，这里又有一个问题：在第2步中，我们如何保证发布事件和删除事件之间的原子性呢？答案是：我们不用保证它们的原子性，我们需要保证的是“至少一次投递”，并且保证消费方幂等。此时的大致场景如下：

* 代码中先发布事件，成功后再从事件表中删除事件；
* 发布消息成功，事件删除也成功，皆大欢喜；
* 如果消息发布不成功，那么代码中不会执行事件删除逻辑，就像事情没有发生一样，一致性得到保证；
* 如果消息发布成功，但是事件删除失败，那么在第二次任务执行时，会重新发布消息，导致消息的重复发送。然而，由于我们要求了消费方的幂等性，也即消费方多次消费同一条消息是ok的，整个过程的一致性也得到了保证。

发布领域事件的整个流程如下：



领域事件的发布

1. 接受用户请求；
2. 处理用户请求；
3. 写入业务表；
4. 写入事件表，事件表和业务表的更新在同一个本地数据库事务中；
5. 事务完成后，即时触发事件的发送（比如可以通过Spring AOP的方式完成，也可以定时扫描事件表，还可以借助诸如MySQL的binlog之类的机制）；
6. 后台任务读取事件表；
7. 后台任务发送事件到消息队列；
8. 发送成功后删除事件。

更多有关事件表的介绍，请参考[Chris Richardson](https://links.jianshu.com/go?to=https://microservices.io/about.html" \t "_blank)的["Transaction Outbox模式"](https://links.jianshu.com/go?to=https://microservices.io/patterns/data/transactional-outbox.html" \t "_blank)和[Udi Dahan](https://links.jianshu.com/go?to=http://udidahan.com/about/" \t "_blank)的["在不使用分布式事务条件下如何处理消息可靠性"](https://links.jianshu.com/go?to=https://vimeo.com/channels/946231/111998645" \t "_blank)的视频。

在事件表场景下，一种常见的做法是将领域事件保存到聚合根中，然后在Repository保存聚合根的时候，将事件保存到事件表中。这种方式对于所有的Repository/聚合根都采用的方式处理，因此可以创建对应的抽象基类。

创建所有聚合根的基类DomainEventAwareAggregate如下：

public abstract class DomainEventAwareAggregate {

@JsonIgnore

private final List<DomainEvent> events = newArrayList();

protected void raiseEvent(DomainEvent event) {

this.events.add(event);

}

void clearEvents() {

this.events.clear();

}

List<DomainEvent> getEvents() {

return Collections.unmodifiableList(events);

}}

这里的raiseEvent()方法用于在具体的聚合根对象中产生领域事件，然后在Repository中获取到事件，与聚合根对象一起完成持久化，创建DomainEventAwareRepository基类如下：

public abstract class DomainEventAwareRepository<AR extends DomainEventAwareAggregate> {

@Autowired

private DomainEventDao eventDao;

public void save(AR aggregate) {

eventDao.insert(aggregate.getEvents());

aggregate.clearEvents();

doSave(aggregate);

}

protected abstract void doSave(AR aggregate);}

具体的聚合根在实现业务逻辑之后调用raiseEvent()方法生成事件，以“更改Order收货地址”业务过程为例：

public class Order extends DomainEventAwareAggregate {

//......

public void changeAddressDetail(String detail) {

if (this.status == PAID) {

throw new OrderCannotBeModifiedException(this.id);

}

this.address = this.address.changeDetailTo(detail);

raiseEvent(new OrderAddressChangedEvent(getId().toString(), detail, address.getDetail()));

}

//......}

在保存Order的时候，只需要处理Order自身的持久化即可，事件的持久化已经在DomainEventAwareRepository基类中完成：

@Component

public class OrderRepository extends DomainEventAwareRepository<Order> {

//......

@Override

protected void doSave(Order order) {

String sql = "INSERT INTO ORDERS (ID, JSON\_CONTENT) VALUES (:id, :json) " +

"ON DUPLICATE KEY UPDATE JSON\_CONTENT=:json;";

Map<String, String> paramMap = of("id", order.getId().toString(), "json", objectMapper.writeValueAsString(order));

jdbcTemplate.update(sql, paramMap);

}

//......

}

当业务操作的事务完成之后，需要通知消息发送设施即时发布事件到消息队列。发布过程最好做成异步的后台操作，这样不会影响业务处理的正常返回，也不会影响业务处理的效率。在Spring Boot项目中，可以考虑采用AOP的方式，在HTTP的POST/PUT/PATCH/DELETE方法完成之后统一发布事件：

@Aspect@Componentpublic class DomainEventPublishAspect {

//......

@After("@annotation(org.springframework.web.bind.annotation.PostMapping) || " +

"@annotation(org.springframework.web.bind.annotation.PutMapping) || " +

"@annotation(org.springframework.web.bind.annotation.PatchMapping) || " +

"@annotation(org.springframework.web.bind.annotation.DeleteMapping) ||")

public void publishEvents(JoinPoint joinPoint) {

logger.info("Trigger domain event publish process.");

taskExecutor.execute(() -> publisher.publish());

}

//......}

以上，我们使用了TaskExecutor在后台开启新的线程完成事件发布，实际的发布由RabbitDomainEventPublisher完成：

@Componentpublic class DomainEventPublisher {

// ......

public void publish() {

Instant now = Instant.now();

LockConfiguration configuration = new LockConfiguration("domain-event-publisher", now.plusSeconds(10), now.plusSeconds(1));

distributedLockExecutor.execute(this::doPublish, configuration);

}

//......}

这里，我们使用了分发布锁来处理并发发送的情况，doPublish()方法将调用实际的消息队列(比如RabbitMQ/Kafka等)API完成消息发送。更多的代码细节，请参考本文的[示例代码](https://links.jianshu.com/go?to=https://github.com/e-commerce-sample/common" \t "_blank)。

#### 消费领域事件

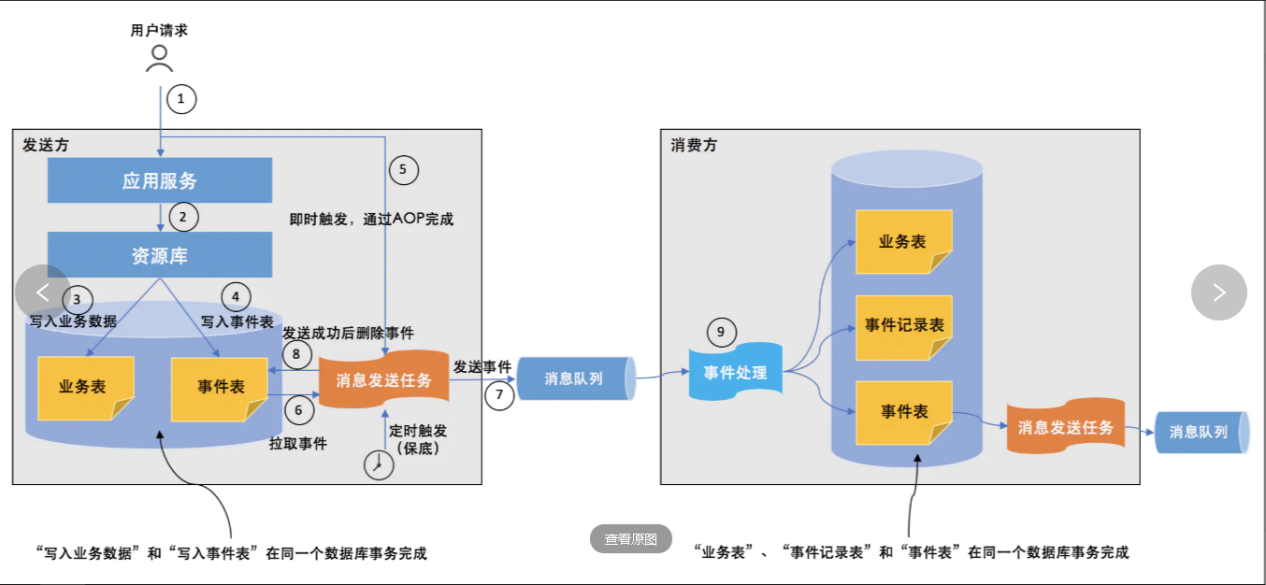
在事件消费时，除了完成基本的消费逻辑外，我们需要重点关注以下两点：

1. 消费方的幂等性
2. 消费方有可能进一步产生事件

对于“消费方的幂等性”，在上文中我们讲到事件的发送机制保证的是“至少一次投递”，为了能够正确地处理重复消息，要求消费方是幂等的，即多次消费事件与单次消费该事件的效果相同。为此，在消费方创建一个事件记录表，用于记录已经消费过的事件，在处理事件时，首先检查该事件是否已经被消费过，如果是则不做任何消费处理。

对于第2点，我们依然沿用前文讲到的事件表的方式。事实上，无论是处理HTTP请求，还是作为消息的消费方，对于聚合根来讲都是无感知的，领域事件由聚合根产生进而由Repository持久化，这些过程都与具体的业务操作源头无关。

综上，在消费领域事件的过程中，程序需要更新业务表、事件记录表以及事件发送表，这3个操作过程属于同一个本地事务，此时整个事件的发布和消费过程如下：



事件的发布和消费全流程

在编码实践时，可以考虑与事件发布过程相同的AOP方式完成对事件的记录，以Spring和RabbitMQ为例，可以将@RabbitListener通过AOP代理起来：

@Aspect@Componentpublic class DomainEventRecordingConsumerAspect {

//......

@Around("@annotation(org.springframework.amqp.rabbit.annotation.RabbitHandler) || " +

"@annotation(org.springframework.amqp.rabbit.annotation.RabbitListener)")

public Object recordEvents(ProceedingJoinPoint joinPoint) throws Throwable {

return domainEventRecordingConsumer.recordAndConsume(joinPoint);

}

//......}

然后在代理过程中通过DomainEventRecordingConsumer完成事件的记录：

@Componentpublic class DomainEventRecordingConsumer {

//......

@Transactional

public Object recordAndConsume(ProceedingJoinPoint joinPoint) throws Throwable {

Object[] args = joinPoint.getArgs();

Optional<Object> optionalEvent = Arrays.stream(args)

.filter(o -> o instanceof DomainEvent)

.findFirst();

if (optionalEvent.isPresent()) {

DomainEvent event = (DomainEvent) optionalEvent.get();

try {

dao.recordEvent(event);

} catch (DuplicateKeyException dke) {

logger.warn("Duplicated {} skipped.", event);

return null;

}

return joinPoint.proceed();

}

return joinPoint.proceed();

}

//......}

这里的DomainEventRecordingConsumer通过直接向事件记录表中插入事件的方式来判断消息是否重复，如果发生重复主键异常DuplicateKeyException，即表示该事件已经在记录表中存在了，因此直接return null;而不再执行业务过程。

需要特别注意的一点是，这里的封装方法recordAndConsume()需要打上@Transactional注解，这样才能保证对事件的记录和业务处理在同一个事务中完成。

此外，由于消费完毕后也需要即时发送事件，因此需要在发布事件的AOP配置DomainEventPublishAspect中加入@RabbitListener：

@Aspect@Componentpublic class DomainEventPublishAspect {

//......

@After("@annotation(org.springframework.web.bind.annotation.PostMapping) || " +

"@annotation(org.springframework.web.bind.annotation.PutMapping) || " +

"@annotation(org.springframework.web.bind.annotation.PatchMapping) || " +

"@annotation(org.springframework.web.bind.annotation.DeleteMapping) ||" +

"@annotation(org.springframework.amqp.rabbit.annotation.RabbitListener) ||")

public void publishEvents(JoinPoint joinPoint) {

logger.info("Trigger domain event publish process.");

taskExecutor.execute(() -> publisher.publish());

}

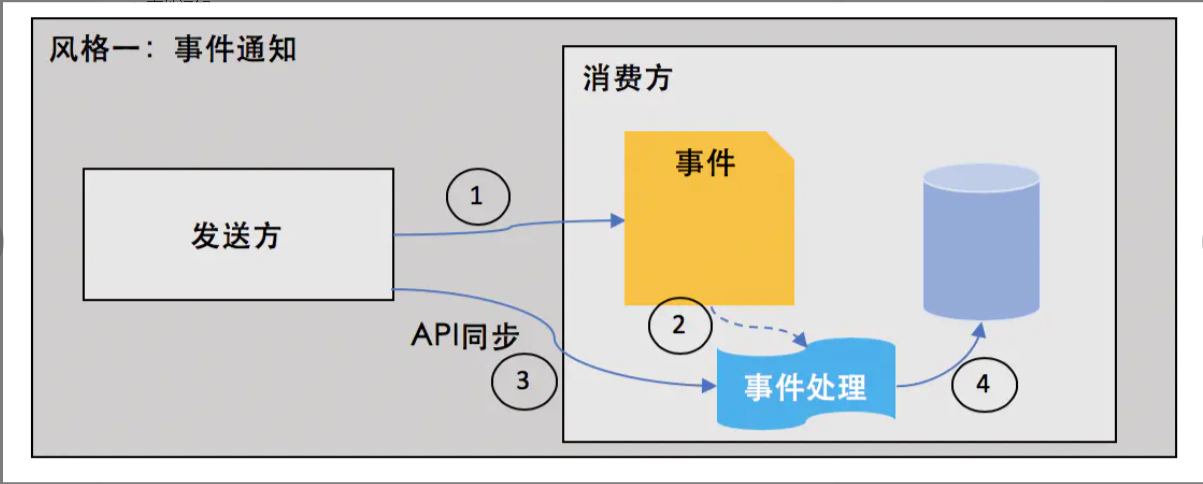
//......}

#### 事件驱动架构的2种风格

事件驱动架构存在多种风格，本文就其中的2种主要风格展开讨论，它们是：

1. 事件通知
2. 事件携带状态转移(Event-Carried State Transfer)

在“事件通知”风格中，事件只是作为一种信号传递到消费方，消费方需要的数据需要额外API请求从源事件系统获取，如图：



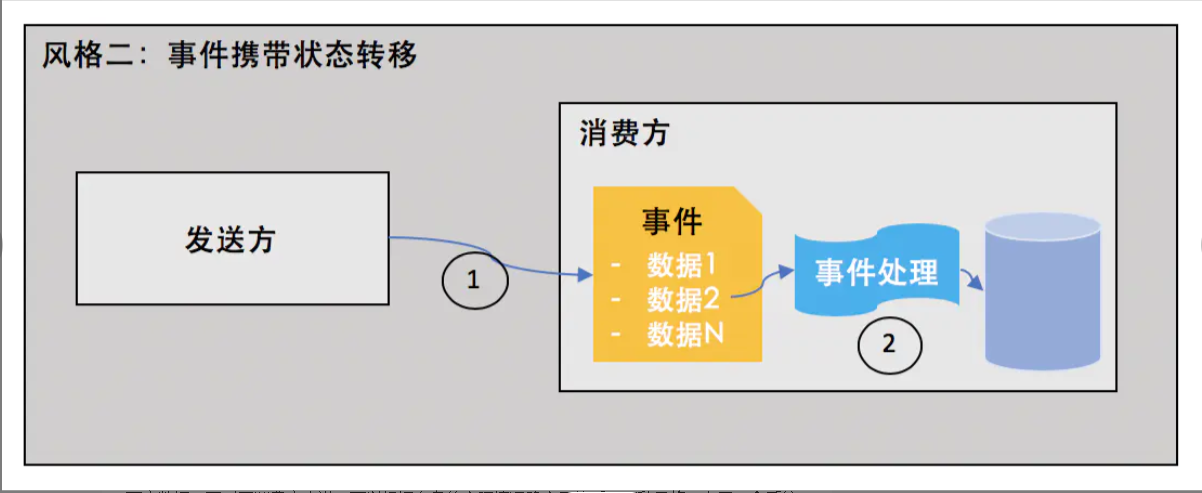
事件通知

在上图的事件通知风格中，对事件的处理流程如下：

1. 发布方发布事件
2. 消费方接收事件并处理
3. 消费方调用发布方的API以获取事件相关数据
4. 消费方更新自身状态

这种风格的好处是，事件可以设计得非常简单，通常只需要携带聚合根的ID即可，由此进一步降低了事件驱动系统中的耦合度。然而，消费方需要的数据依然需要额外的API调用从发布方获取，这又从另一个角度增加了系统之间的耦合性。此外，如果源系统宕机，消费方也无法完成后续操作，因此可用性会受到影响。

在“事件携带状态转移”中，消费方所需要的数据直接从事件中获取，因此不需要额外的API请求：



事件携带状态转移

这种风格的好处在于，即便发布方系统不可用，消费方依然可以完成对事件的处理。

笔者的建议是，对于发布方来说，作为一种数据提供者的“自我修养”，事件应该包含足够多的上下文数据，而对于消费方来讲，可以根据自身的实际情况确定具体采用哪种风格。在同一个系统中，同时采用2种风格是可以接受的。比如，对于基于事件的CQRS而言，可以采用“事件通知”，此时的事件只是一个“触发器”，一个聚合下的所有事件所触发的结果是一样的，即都是告知消费方需要从源系统中同步数据，因此此时的消费方可以对聚合下的所有事件一并处理，而不用为每一种事件单独开发处理逻辑。

事实上，事件驱动还存在第3种风格，即事件溯源，本文不对此展开讨论。更多有关事件驱动架构不同风格的介绍，请参考[Martin Fowler的“事件风格”文章](https://links.jianshu.com/go?to=https://martinfowler.com/articles/201701-event-driven.html" \t "_blank)。

## 第二部分：基于RabbitMQ的示例项目

在本部分中，我将以一个简单的电商平台微服务系统为例，采用RabbitMQ作为消息机制讲解事件驱动架构落地的全过程。

该电商系统包含3个微服务，分别是：

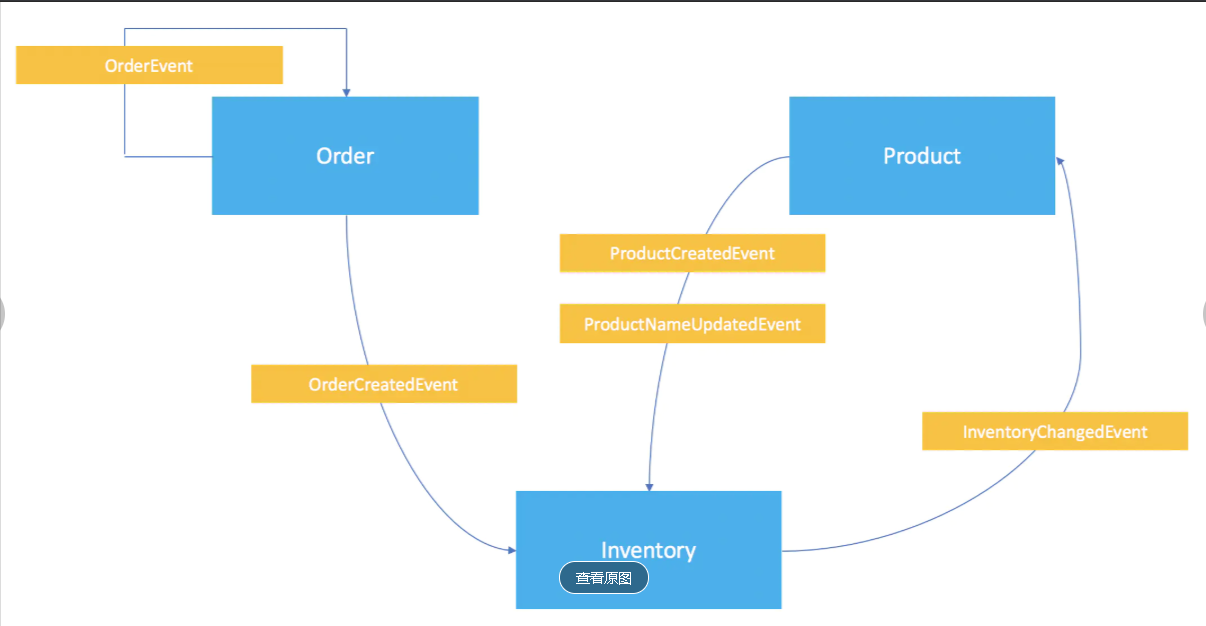
* 订单(Order)服务：用于用户下单
* 产品(Product)服务：用于管理/展示产品信息
* 库存(Inventory)服务：用于管理产品对应的库存

[整个系统](https://links.jianshu.com/go?to=https://github.com/e-commerce-sample/" \t "_blank)包含以下代码库：

| **代码库** | **用途** | **地址** |
| --- | --- | --- |
| order-backend | Order服务 | [https://github.com/e-commerce-sample/order-backend](https://links.jianshu.com/go?to=https://github.com/e-commerce-sample/order-backend" \t "_blank) |
| product-backend | Product服务 | [https://github.com/e-commerce-sample/product-backend](https://links.jianshu.com/go?to=https://github.com/e-commerce-sample/product-backend" \t "_blank) |
| inventory-backend | Inventory服务 | [https://github.com/e-commerce-sample/inventory-backend](https://links.jianshu.com/go?to=https://github.com/e-commerce-sample/inventory-backend" \t "_blank) |
| common | 共享依赖包 | [https://github.com/e-commerce-sample/common](https://links.jianshu.com/go?to=https://github.com/e-commerce-sample/common" \t "_blank) |
| devops | 基础设施 | [https://github.com/e-commerce-sample/devops](https://links.jianshu.com/go?to=https://github.com/e-commerce-sample/devops" \t "_blank) |

其中，common代码库包含了所有服务所共享的代码和配置，包括所有服务中的所有事件（请注意，这种做法只是笔者为了编码上的便利，并不是一种好的实践，一种更好的实践是各个服务各自管理自身产生的事件），以及RabbitMQ的通用配置（即每个服务都采用相同的方式配置RabbitMQ设施），同时也包含了异常处理和分布式锁等配置。devops库中包含了RabbitMQ的Docker镜像，用于在本地测试。

整个系统中涉及到的领域事件如下：



示例电商系统中的领域事件

其中：

* Order服务自己消费了自己产生的所有OrderEvent用于CQRS同步读写模型；
* Inventory服务消费了Order服务的OrderCreatedEvent事件，用于在下单之后即时扣减库存；
* Inventory服务消费了Product服务的ProductCreatedEvent和ProductNameChangedEvent事件，用于同步产品信息；
* Product服务消费了Inventory服务的InventoryChangedEvent用于更新产品库存。

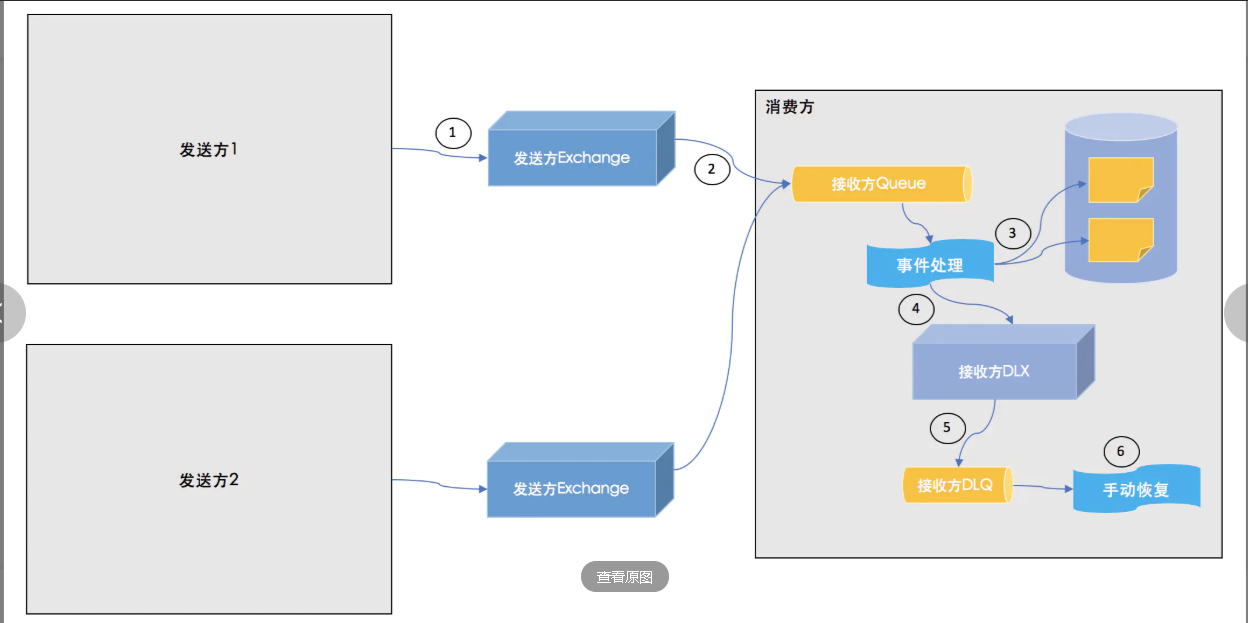
#### 配置RabbitMQ

阅读本小节需要熟悉RabbitMQ中的基本概念，建议不熟悉RabbitMQ的读者事先参考[RabbitMQ入门文章](https://links.jianshu.com/go?to=https://www.cloudamqp.com/blog/2015-05-18-part1-rabbitmq-for-beginners-what-is-rabbitmq.html" \t "_blank)。

这里介绍2种RabbitMQ的配置方式，一种简单的，一种稍微复杂的。两种配置过程中会反复使用到以下概念，读者可以先行熟悉：

| **概念** | **类型** | **解释** | **命名** | **示例** |
| --- | --- | --- | --- | --- |
| 发送方Exchange | Exchange | 用于接收某个微服务中所有消息的Exchange，一个服务只有一个发送方Exchange | xxx-publish-x | order-publish-x |
| 发送方DLX | Exchange | 用于接收发送方无法路由的消息 | xxx-publish-dlx | order-publish-dlx |
| 发送方DLQ | Queue | 用于存放发送方DLX的消息 | xxx-publish-dlq | order-publish-dlq |
| 接收方Queue | Queue | 用于接收发送方Exchange的消息，一个服务只有一个接收方Queue用于接收所有外部消息 | xxx-receive-q | product-receive-q |
| 接收方DLX | Exchange | 死信Exchange，用于接收消费失败的消息 | xxx-receive-dlx | product-receive-dlx |
| 接收方DLQ | Queue | 死信队列，用于存放接收方DLX的消息 | xxx-receive-dlq | product-receive-dlq |
| 接收方恢复Exchange | Exchange | 用于接收从接收方DLQ中手动恢复的消息，接收方Queue应该绑定到接收方恢复Exchange | xxx-receive-recover-x | product-receive-recover-x |

在简单配置方式下，消息流向图如下：

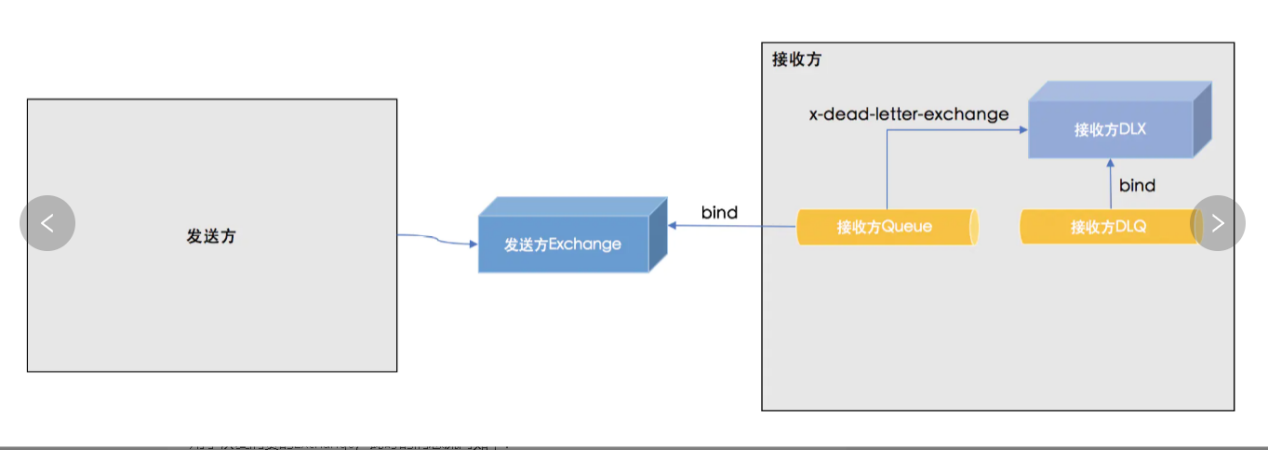


简单方式的RabbitMQ配置

1. 发送方发布事件到发送方Exchange
2. 消息到达消费方的接收方Queue
3. 消费成功处理消息，更新本地数据库
4. 如果消息处理失败，消息被放入接收方DLX
5. 消息到达死信队列接收方DLQ
6. 对死信消息做手工处理（比如作日志记录等）

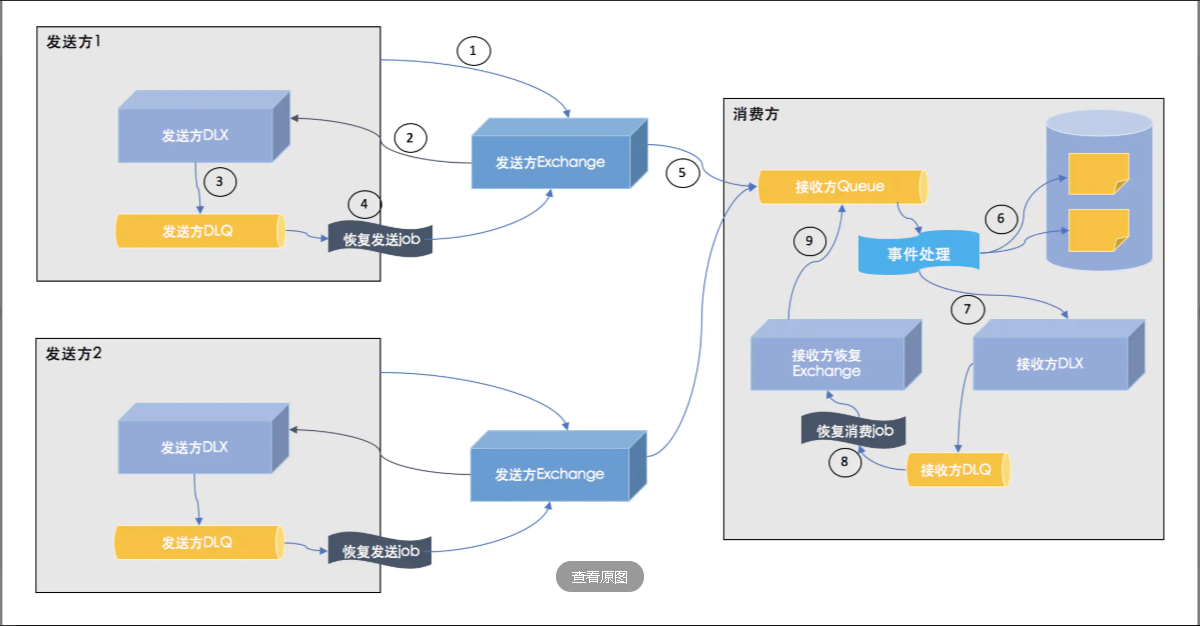
对于发送方而言，事件驱动架构提倡的是“发送后不管”机制，即发送方只需要保证事件成功发送即可，而不用关心是谁消费了该事件。因此在配置发送方的RabbitMQ时，可以简单到只配置一个发送方Exchange即可，该Exchange用于接收某个微服务中所有类型的事件。在消费方，首先配置一个接收方Queue用于接收来自所有发送方Exchange的所有类型的事件，除此之外对于消费失败的事件，需要发送到接收方DLX，进而发送到接收方DLQ中，对于接收方DLQ的事件，采用手动处理的形式恢复消费。

在简单方式下的RabbitMQ配置如下：



简单方式的RabbitMQ配置

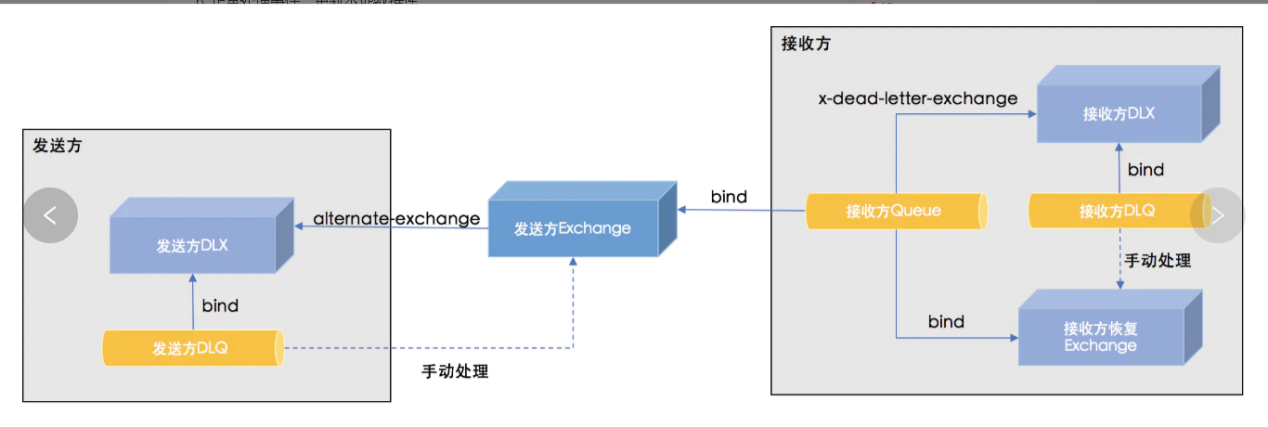
在第2种配置方式稍微复杂一点，其建立在第1种基础之上，增加了发送方的死信机制以及消费方用于恢复消费的Exchange，此时的消息流向如下：



配置发送方DLQ和接收方恢复Exchange

1. 发送方发布事件
2. 事件发布失败时被放入死信Exchange发送方DLX
3. 消息到达死信队列发送方DLQ
4. 对于发送方DLQ中的消息进行人工处理，重新发送
5. 如果事件发布正常，则会到达接收方Queue
6. 正常处理事件，更新本地数据库
7. 事件处理失败时，发到接收方DLX，进而路由到接收方DLQ
8. 手工处理死信消息，将其发到接收方恢复Exchange，进而重新发到接收方Queue

此时的RabbitMQ配置如下：



在以上2种方式中，我们都启用了RabbitMQ的“发送方确认”和“消费方确认”，另外，发送方确认也可以通过RabbitMQ的事务(不是分布式事务)替代，不过效率更低。更多关于RabbitMQ的知识，可以参考笔者的[Spring AMQP学习笔记](https://www.jianshu.com/p/69388fb6c8ec" \t "_blank)和[RabbitMQ最佳实践](https://www.jianshu.com/p/5f6afe31898c" \t "_blank)。

### 系统演示

* 启动RabbitMQ，切换到ecommerce-sample/devops/local/rabbitmq目录，运行：

./start-rabbitmq.sh

* 启动Order服务：切换到ecommerce-sample/order-backend项目，运行：

./run.sh //监听8080端口，调试5005端口

* 启动Product服务：切换到ecommerce-sample/product-backend项目，运行：

./run.sh //监听8082端口，调试5006端口

* 启动Inventory服务：切换到ecommerce-sample/inventory-backend项目，运行：

./run.sh //监听8083端口，调试5007端口

* 创建Product：

curl -X POST \

http://localhost:8082/products \

-H 'Content-Type: application/json' \

-H 'cache-control: no-cache' \

-d '{

"name":"好吃的苹果",

"description":"原生态的苹果",

"price": 10.0}'

此时返回Product ID：

{"id":"3c11b3f6217f478fbdb486998b9b2fee"}

* 查看Product：

curl -X GET \

http://localhost:8082/products/3c11b3f6217f478fbdb486998b9b2fee \

-H 'cache-control: no-cache'

返回如下：

{

"id": {

"id": "3c11b3f6217f478fbdb486998b9b2fee"

},

"name": "好吃的苹果",

"price": 10,

"createdAt": 1564361781956,

"inventory": 0,

"description": "原生态的苹果"}

可以看到，新创建的Product的库存(inventory)默认为0。

* 创建Product时，会创建ProductCreatedEvent，Inventory服务接收到该事件后会自动创建对应的Inventory，日志如下：

2019-07-29 08:56:22.276 -- INFO [taskExecutor-1] c.e.i.i.InventoryEventHandler : Created inventory[5e3298520019442b8a6d97724ab57d53] for product[3c11b3f6217f478fbdb486998b9b2fee].

* 增加Inventory为10：

curl -X POST \

http://localhost:8083/inventories/5e3298520019442b8a6d97724ab57d53/increase \

-H 'Content-Type: application/json' \

-H 'cache-control: no-cache' \

-d '{

"increaseNumber":10}'

* 增加Inventory之后，会发送InventoryChangedEvent，Product服务接收到该事件后会自动同步自己的库存，再次查看Product：

curl -X GET \

http://localhost:8082/products/3c11b3f6217f478fbdb486998b9b2fee \

-H 'cache-control: no-cache'

返回如下：

{

"id": {

"id": "3c11b3f6217f478fbdb486998b9b2fee"

},

"name": "好吃的苹果",

"price": 10,

"createdAt": 1564361781956,

"inventory": 10,

"description": "原生态的苹果"}

可以看到，Product的库存已经更新为10。

* 至此，Product和Inventory都准备好了，让我们下单吧：

curl -X POST \

http://localhost:8080/orders \

-H 'Content-Type: application/json' \

-H 'cache-control: no-cache' \

-d '{

"items": [

{

"productId": "3c11b3f6217f478fbdb486998b9b2fee",

"count": 2,

"itemPrice": 10

}

],

"address": {

"province": "四川",

"city": "成都",

"detail": "天府软件园1号"

}}'

返回Order ID：

{

"id": "d764407855d74ff0b5bb75250483229f"}

* 创建订单之后，会发送OrderCreatedEvent，Inventory服务接收到该事件会自动扣减相应库存：

2019-07-29 09:11:31.202 -- INFO [taskExecutor-1] c.e.i.i.InventoryEventHandler : Inventory[5e3298520019442b8a6d97724ab57d53] decreased to 8 due to order[d764407855d74ff0b5bb75250483229f] creation.

同时，Inventory将发送InventoryChangedEvent，Product服务接收到该事件会自动更新Product的库存，再次查看Product：

curl -X GET \

http://localhost:8082/products/3c11b3f6217f478fbdb486998b9b2fee \

-H 'cache-control: no-cache'

返回如下：

{

"id": {

"id": "3c11b3f6217f478fbdb486998b9b2fee"

},

"name": "好吃的苹果",

"price": 10,

"createdAt": 1564361781956,

"inventory": 8,

"description": "原生态的苹果"}

可以看到，Product的库存从10减少到了8，因为先前下单时我们选了2个Product。

## 总结

本文首先独立于消息队列的技术实现，讲到了事件驱动架构在落地过程中的诸多方面以及问题，包括领域事件的建模、通过聚合根暂存事件然后由Repository完成存储，再由后台任务读取事件表完成事件的实际发布。在消费方，通过幂等性解决在“至少一次投递”的情况下所带来的重复消费问题。另外，还讲到了事件驱动架构的2种常见风格，即事件通知和事件携带状态转移，以及他们之间的优劣势。在第二部分，以RabbitMQ为例，分享了如何在一个微服务化的系统中落地事件驱动架构。

作者：无知者云  
链接：https://www.jianshu.com/p/fbd3951259eb  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

# **后端开发实践系列之四——简单可用的CQRS编码实践**

本文只讲了一件事情：软件模型中存在读模型和写模型之分，CQRS便为此而生。

20多年前，[Bertrand Meyer](https://links.jianshu.com/go?to=https://en.wikipedia.org/wiki/Bertrand_Meyer" \t "_blank)在他的《[Object-Oriented Software Construction](https://links.jianshu.com/go?to=https://en.wikipedia.org/wiki/Object-Oriented_Software_Construction" \t "_blank)》一书中提出了CQS（Command Query Seperation，命令查询分离）的概念，指出：

Every method should either be a command that performs an action, or a query that returns data to the caller, but never both. (一个方法要么作为一个“命令”执行一个操作，要么作为一次“查询”向调用方返回数据，但两者不能共存。）

这里的“命令”可以理解为更新软件状态的写操作，Martin Fowler将此称为“[Modifier](https://links.jianshu.com/go?to=https://martinfowler.com/bliki/CommandQuerySeparation.html" \t "_blank)”；而“查询”即为读操作，是无副作用的。这种分离的好处在于使程序变得更容易推理与维护，由于查询操作不会更新软件状态，在编码时我们将更加有信心。试想，如果程序中出了一个bug，如果这个bug出现在查询过程中，那么我们至少可以消除这个bug可能给软件带来脏数据的恐惧。

后来，[Greg Young](https://links.jianshu.com/go?to=https://twitter.com/gregyoung?lang=en" \t "_blank)在此基础上提出了CQRS（Command Query Resposibility Segregation，命令查询职责分离），将CQS的概念从方法层面提升到了模型层面，即“命令”和“查询”分别使用不同的对象模型来表示。

采用CQRS的驱动力除了从CQS那里继承来的好处之外，还旨在解决软件中日益复杂的查询问题，比如有时我们希望从不同的维度查询数据，或者需要将各种数据进行组合后返回给调用方。此时，将查询逻辑与业务逻辑糅合在一起会使软件迅速腐化，诸如逻辑混乱、可读性变差以及可扩展性降低等等一些列问题。

## 一个例子

设想电商系统中的订单（Order）对象，一开始其对应的OrderRepository类可以简单到只包含2个方法：

public interface OrderRepository {

void save(Order order);

Order byId(String id);}

在项目的演进中，你可能需要依次实现以下需求：

1. 查询某个Order详情，详情中不用包含Order的某些字段；
2. 查询Order列表，列表中所展示的数据比Order详情更少；
3. 根据时间、类别和金额等多种筛选条件查询Order列表；
4. 展示Order中的产品（Product）概要信息，而Product属于另一个业务实体；
5. 展示Order下单人的昵称，下单人信息属于另一个单独的账户系统，用户修改昵称之后，Order下单人昵称也需要相应更新；
6. ......

当这些需求实现完后，你可能会发现OrderRepository和领域模型已经被各种“查询”功能淹没了。什么？OrderRepository不是给领域模型提供Order聚合根对象的吗，为什么却充斥着如此多的查询逻辑？

CQRS通过单独的读模型解决上述问题，其大致的架构图如下：



CQRS架构，摘自https://martinfowler.com/bliki/CQRS.html

对于Command侧，主要的讲究是将业务用例建模成对应的Command对象，然后在对Command的处理流程中应用核心的业务逻辑，其中最重要的是领域模型的建模，关于此的内容请参考笔者的《[领域驱动设计(DDD)编码实践](https://www.jianshu.com/p/84f1d922a6d4" \t "_blank)》文章，本文着重介绍Query侧的编码实践。

在本文中，查询模型（Query Model）也被表达为读模型（Read Model）；命令模型（Command Model）也被表达为写模型（Write Model）。

## CQRS实现模式概览

#### 常见误解

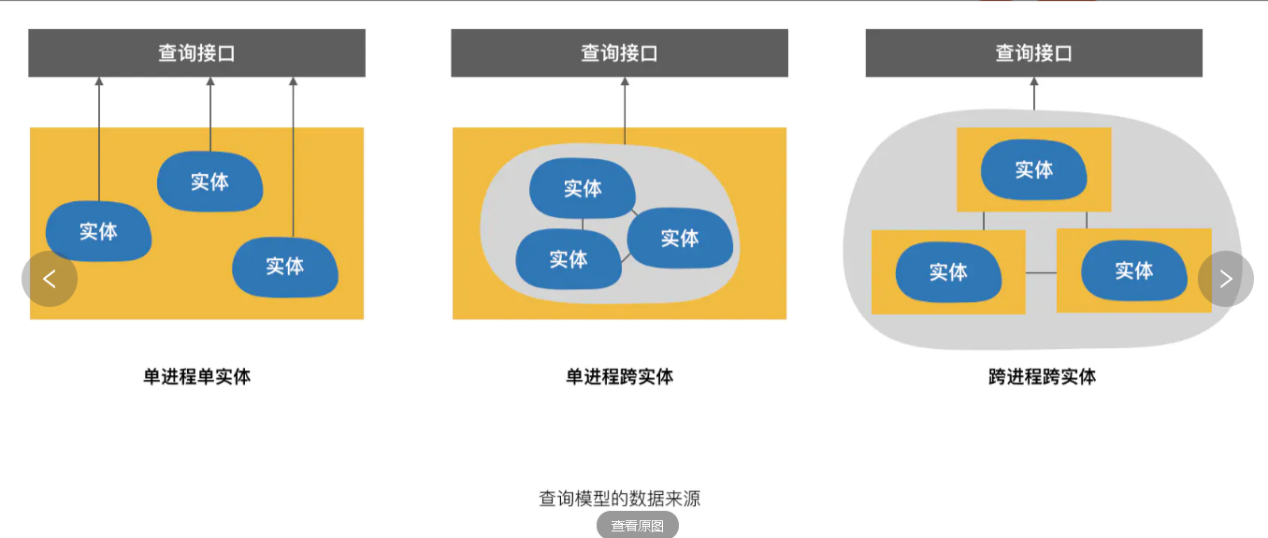
在网上搜索一番，你会发现很多关于CQRS的文章都将CQRS与[Event Sourcing](https://links.jianshu.com/go?to=https://martinfowler.com/eaaDev/EventSourcing.html" \t "_blank)（事件溯源）结合起来使用，这容易让人觉得采用CQRS就一定需要同时使用Event Sourcing，事实上这是一种误解。CQRS究其本意只是要求“读写模型的分离”，并未要求使用Event Sourcing；再者，Event Sourcing会极大地增加软件的复杂度，而本文追求的是“简单可用的CQRS”，因此本文将不会涉及Event Sourcing相关内容。更多内容，请参考[简化版CQRS](https://links.jianshu.com/go?to=https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/apply-simplified-microservice-cqrs-ddd-patterns" \t "_blank)的文章。

另外需要指出的是，读写模型的分离并不一定意味着数据存储的分离，不过在实际应用中，数据存储分离是一种常见的CQRS实践模式，在这种模式中，写模型的数据会同步到读模型数据存储中，同步过程通常通过消息机制完成，在DDD场景下，消息通常承载的是领域事件（Domain Event）。

#### 查询模型的数据来源

无论是单体还是微服务，所读数据的唯一正确来源（Single Source of Truth）最终都来自于业务实体（Entity）对象（比如DDD中的聚合根），基于此，所读数据的来源形式大致分为以下几种：

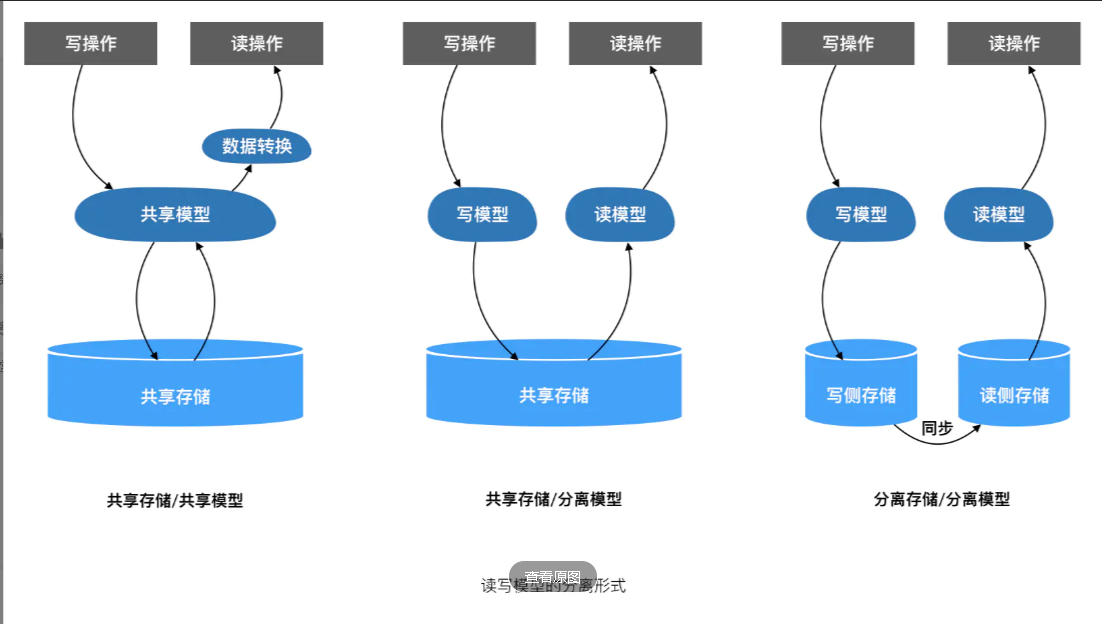
* 所读数据来源于同一个进程空间的单个实体（后文简称“**单进程单实体**”），这里的进程空间指某个单体应用或者单个微服务；
* 所读数据来源于同一个进程空间中的多个实体（后文简称“**单进程跨实体**”）；
* 所读数据来源于不同进程空间中的多个实体（后文简称“**跨进程跨实体**”）。



#### 读写模型的分离形式

CQRS中的读写分离存在2个层次，一层是代码中的模型是否需要分离，另一层是数据存储是否需要分离，总结下来有以下几种：

* **共享存储/共享模型**：读写模型共享数据存储（即同一个数据库），同时也共享代码模型，数查询据通过模型转换（Projection）后返回给调用方，事实上这不能算CQRS，但是对于很多中小型项目而言已经足够；
* **共享存储/分离模型**：共享数据存储，代码中分别建立写模型和读模型，读模型通过最适合于查询的方式进行建模；
* **分离存储/分离模型**：数据存储和代码模型都是分离的，这种方式通常用于需要聚合查询多个子系统的情况，比如微服务系统。



将以上“查询模型的数据来源”与“读写模型的分离形式”相组合，我们可以得到以下不同的CQRS模式及其适用范围：

| **数据来源形式** | **模型分离形式** | **适用范围** |
| --- | --- | --- |
| 单进程单实体 | 共享存储/共享模型 | 其实算不上CQRS，但对于很多中小型项目已经足够 |
| 单进程单实体 | 共享存储/分离模型 | 适用于单实体查询比较复杂或者对查询效率要求较高的场景 |
| 单进程单实体 | 不同存储/分离模型 | 适用于对单个实体的查询非常复杂的场景 |
| 单进程跨实体 | 共享存储/共享模型 | 不适用 |
| 单进程跨实体 | 共享存储/分离模型 | 适用于查询比较复杂的场景，比如需要做多表join操作 |
| 单进程跨实体 | 分离存储/分离模型 | 适用于复杂查询或者对查询效率要求较高的情况 |
| 跨进程跨实体 | 共享存储/共享模型 | 不适用 |
| 跨进程跨实体 | 共享存储/分离模型 | 不适用 |
| 跨进程跨实体 | 分离存储/分离模型 | 主要用于微服务中需要对多个服务进行聚合查询的场景 |

总结下来，有以下几种常见做法：

* 单进程单实体 + 共享存储/共享模型
* 单进程单实体 + 共享存储/分离模型
* 单进程跨实体 + 共享存储/分离模型
* 单进程跨实体 + 分离存储/分离模型
* 跨进程跨实体 + 分离存储/分离模型

接下来，针对以上几种常见做法，本文将依次给出编码示例。

## CQRS编码实践

本文的示例是一个简单的电商系统，其中包含以下微服务：

| **服务** | **用途** | **所含实体** | **Git地址** |
| --- | --- | --- | --- |
| 订单服务 | 用于用户下单 | Order | [ecommerce-order-service](https://links.jianshu.com/go?to=https://github.com/e-commerce-sample/ecommerce-order-service" \t "_blank) |
| 订单查询服务 | 用于订单的CQRS查询操作 | 无 | [ecommerce-order-query-service](https://links.jianshu.com/go?to=https://github.com/e-commerce-sample/ecommerce-order-query-service" \t "_blank) |
| 产品服务 | 用于管理/展示产品信息 | Product Category(产品目录) | [ecommerce-product-service](https://links.jianshu.com/go?to=https://github.com/e-commerce-sample/ecommerce-product-service" \t "_blank) |
| 库存服务 | 用于管理产品对应的库存 | Inventory | [ecommerce-inventory-service](https://links.jianshu.com/go?to=https://github.com/e-commerce-sample/ecommerce-inventory-service" \t "_blank) |

示例代码请参考：

[https://github.com/e-commerce-sample](https://links.jianshu.com/go?to=https://github.com/e-commerce-sample" \t "_blank)

请注意，本文的示例电商项目只是一个虚构出来的简单项目，仅仅用于演示CQRS的各种编码模式，并不具备实际参考价值。

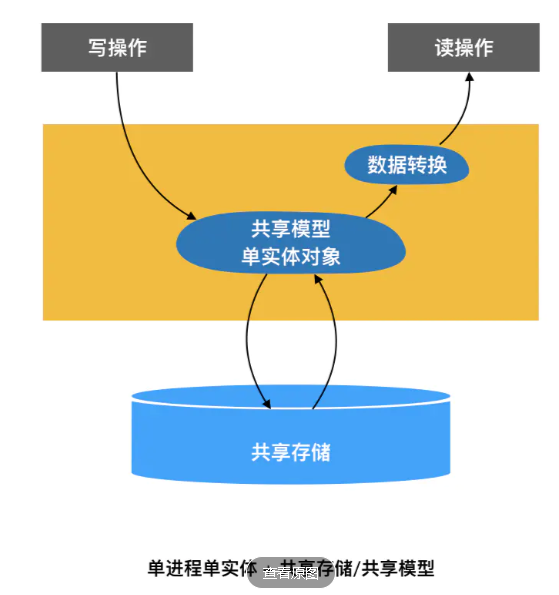
针对以上各种CQRS模式组合，本文将使用电商系统中的以下业务用例进行演示：

| **CQRS模式** | **业务查询用例** | **所属服务** |
| --- | --- | --- |
| 单进程单实体 + 共享存储/共享模型 | Inventory详情查询 | 库存服务 |
| 单进程单实体 + 共享存储/分离模型 | Product摘要查询 | 产品服务 |
| 单进程跨实体 + 共享存储/分离模型 | Product详情查询（包含Category信息） | 产品服务 |
| 单进程跨实体 + 分离存储/分离模型 | Product详情查询（包含Category信息） | 产品服务 |
| 跨进程跨实体 + 分离存储/分离模型 | Order详情查询（包含Product信息） | 订单查询服务 |

#### 1. ****单进程单实体**** + ****共享存储/共享模型****

对于简单的单体或者微服务应用，这种方式是最自然最直接的方式，事实上我们并不需要太多设计上的思考便能想到这种方式。在这种方式中，存在单个领域实体模型同时用于读写操作，在向调用方返回查询数据时，需要针对性地对领域模型进行转换，转换的目的在于：

* 调用方所需的数据模型与领域模型可能不一致；
* 有些敏感信息是不能返回给调用方的，需要屏蔽；
* 从设计上讲，领域模型不能直接返回给调用方，否则会产生领域模型的泄露
* 将领域模型直接返回给调用方会在领域模型与对外接口间产生强耦合，不利于领域模型自身的演进。



这里，我们以“库存(Inventory)详情查询”为例进行演示，Inventory领域模型定义如下：

public class Inventory{

private String id;

private String productId;

private String productName;

private int remains;

private Instant createdAt;}

在获取Inventory详情时，我们并不需要返回领域模型中的productId和createdAt字段，于是在Inventory中创建相应的转换方法如下：

public InventoryRepresentation toRepresentation() {

return new InventoryRepresentation(this.id,

this.productName,

this.remains);

}

这里的InventoryRepresentation即表示读模型，后缀Representation取自REST中的“R”，表示读模型是一种数据展现，下文将沿用这种命名形式。在InventoryApplicationService服务中返回InventoryRepresentation:

public InventoryRepresentation byId(String inventoryId) {

return repository

.byId(inventoryId)

.toRepresentation();

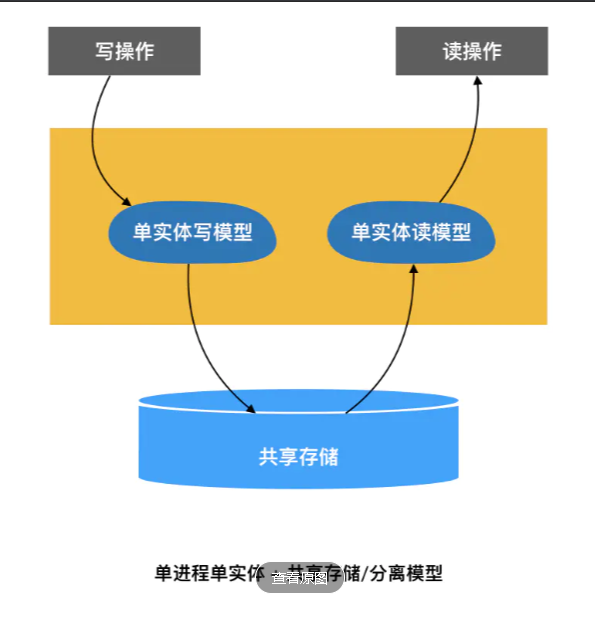
}

值得一提的是，在查询Inventory时，我们使用了应用服务（ApplicationService）-InventoryApplicationService，此时的InventoryApplicationService同时承担了读操作和写操作的业务入口，在实践中也可以将此二者分离开来，即让InventoryApplicationService只负责写操作，而另行创建InventoryRepresentationService专门用于读操作。

另外，抛开CQRS，为了保证每一个聚合根实体自身的完备性，即便在没有调用方查询的情况下，笔者也建议为每一个聚合根提供一个Representation 并对外暴露查询接口。因此每一个聚合根中都会有一个toRepresentation()方法，该方法仅仅返回当前聚合根的状态，而不会关联其他实体对象（比如下文提到的“单进程跨实体”）。

#### 2. 单进程单实体 + 共享存储/分离模型

有时，即便是对于单个实体，其查询也会变得复杂，为了维护读写过程彼此的清晰性，我们可以对读模型和写模型分别建模，事实上这也是CQRS的本意。



在Product服务中，需要返回Product的摘要信息，并对返回列表进行分页处理，为此独立于ApplicationService创建ProductRepresentationService，直接从数据库读取数据构建ProductSummaryRepresentation。

@Transactional(readOnly = true)

public PagedResource<ProductSummaryRepresentation> listProducts(int pageIndex, int pageSize) {

MapSqlParameterSource parameters = new MapSqlParameterSource();

parameters.addValue("limit", pageSize);

parameters.addValue("offset", (pageIndex - 1) \* pageSize);

List<ProductSummaryRepresentation> products = jdbcTemplate.query(SELECT\_SQL, parameters,

(rs, rowNum) -> new ProductSummaryRepresentation(rs.getString("ID"),

rs.getString("NAME"),

rs.getBigDecimal("PRICE")));

int total = jdbcTemplate.queryForObject(COUNT\_SQL, newHashMap(), Integer.class);

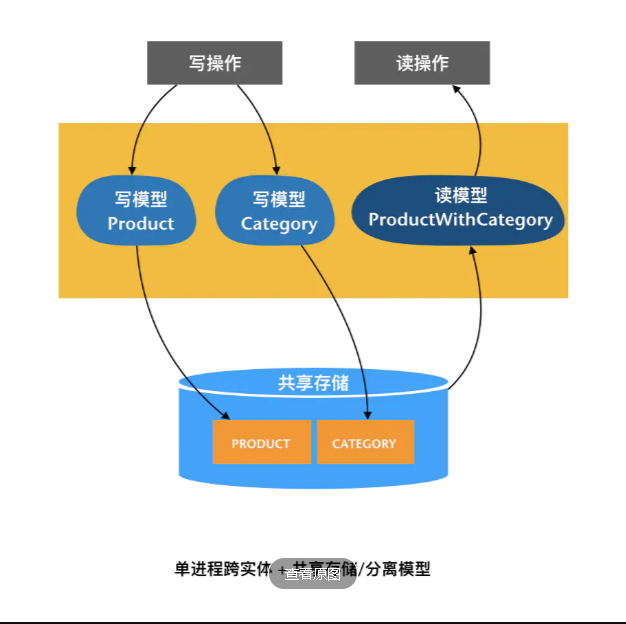
return PagedResource.of(total, pageIndex, products);

}

这里，我们绕过了领域模型Product，也绕过了其对应的ProductRepository，以最快速的方式从数据库中直接获取数据。

#### 3. 单进程跨实体 + 共享存储/分离模型

既然单个实体都有必要使用分离模型，那么在同一个进程空间中的跨实体查询更有理由使用分离模型的形式。对于简单形式跨实体查询，还用不着使用分离的存储，只需要做一些join联合查询即可。



在Product服务中，存在Product和Category两个聚合根对象， 在查询Product时，我们希望一并带上Category的信息，为此创建ProductWithCategoryRepresentation如下：

@Valuepublic class ProductWithCategoryRepresentation {

private String id;

private String name;

private String categoryId;

private String categoryName;}

在ProductRepresentationService中，直接从数据库获取Product和Category数据，此时需要对PRODUCT和CATEGORY两张表做join操作：

@Transactional(readOnly = true)

public ProductWithCategoryRepresentation productWithCategory(String id) {

String sql = "SELECT PRODUCT.ID, PRODUCT.NAME, CATEGORY.ID AS CATEGORY\_ID, CATEGORY.NAME AS CATEGORY\_NAME FROM PRODUCT JOIN CATEGORY ON PRODUCT.CATEGORY\_ID=CATEGORY.ID WHERE PRODUCT.ID=:productId;";

return jdbcTemplate.queryForObject(sql, of("productId", id),

(rs, rowNum) -> new ProductWithCategoryRepresentation(rs.getString("ID"),

rs.getString("NAME"),

rs.getString("CATEGORY\_ID"),

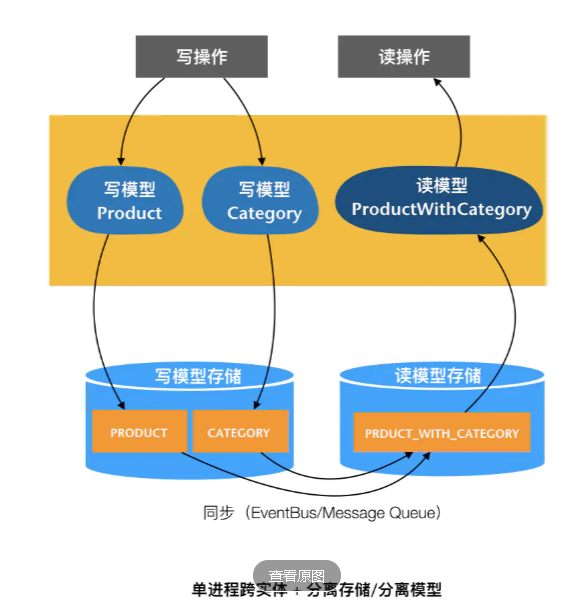
rs.getString("CATEGORY\_NAME")));

}

需要注意的是，如果join的级联太多，那么会大大影响查询的效率，并且使程序变得更加复杂。一般来讲，如果join次数达到了3次及其以上，建议考虑采用分离存储的形式。

#### 4. 单进程跨实体 + 分离存储/分离模型

依然以返回ProductWithCategoryRepresentation为例，假设我们认为先前的join操作太复杂或者太低效了，需要采用专门的数据库来简化查询提升效率。



为此创建单独的读模型数据库表PRODUCT\_WITH\_CATEGORY：

CREATE TABLE PRODUCT\_WITH\_CATEGORY(

PRODUCT\_ID VARCHAR(32) NOT NULL,

PRODUCT\_NAME VARCHAR(100) NOT NULL,

CATEGORY\_ID VARCHAR(32) NOT NULL,

CATEGORY\_NAME VARCHAR(100) NOT NULL,

PRIMARY KEY (PRODUCT\_ID)) CHARACTER SET utf8mb4

COLLATE utf8mb4\_unicode\_ci;

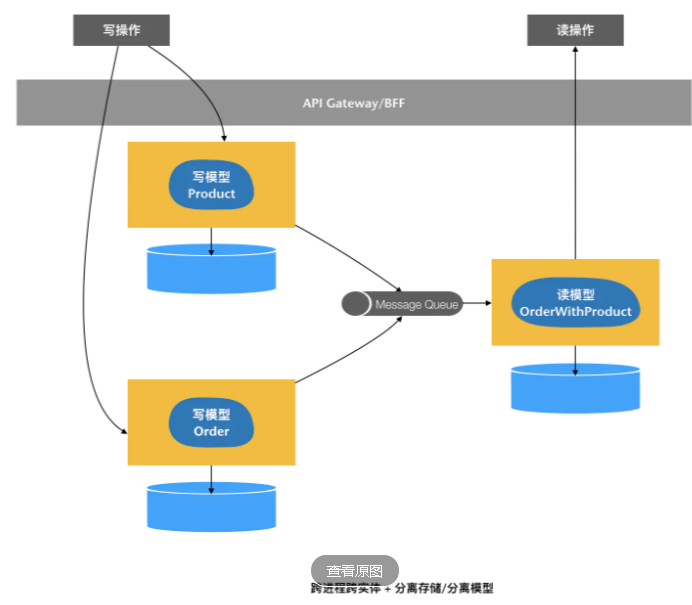
读写同步通常通过领域事件的形式完成，由于是在同一个进程空间中，因此读写同步相比于跨进程的同步来说，可以有更多的选择：

* 使用进程内事件机制（比如Guava的EventBus），在与写操作相同的事务中同步，这种方式的好处是可以保证写操作与同步操作的原子性进而确保读写间的数据一致性，缺点是在写操作过程中存在额外的数据库同步开销进而增加了写操作的延迟时间；
* 使用进程内事件机制，独立事务同步（比如Guava的AsyncEventBus），这种方式的好处是写操作和同步操作彼此独立互不影响，缺点是无法保证二者的原子性进而可能使系统产生脏数据；
* 使用独立的消息机制（比如RabbitMQ/Kafka等），独立事务同步，可以将查询功能分离为单独的子系统，事实上这种方式已经与“跨进程跨实体 + 分离存储/分离模型”相似，因此请参考“5. 跨进程跨实体 + 分离存储/分离模型”小节。

#### 5. 跨进程跨实体 + 分离存储/分离模型

这种方式在微服务中最常见，因为微服务系统首先是多进程的，每个服务都内聚性地管理自身的聚合根对象，另外，微服务的数据存储通常也是独占式的，意味着在微服务系统中数据存储一定是分离的，在这种场景下，跨微服务之间的查询通常采用“[API Compositon](https://links.jianshu.com/go?to=https://microservices.io/patterns/data/api-composition.html" \t "_blank)”模式或者本文的CQRS模式。

在"跨进程跨实体 + 分离存储/分离模型"中，存在一个单独的查询服务用于CQRS的读操作，查询所需数据通常通过事件机制从不同的其他业务服务中同步而来，读操作所返回的数据通过API Gateway或者BFF向外暴露，示意图如下：



在本文的示例电商项目中，需要在查询Order的时候同时带上Product的信息，但是由于Order和Product分别属于不同的服务，为此创建ecommerce-order-query-service查询服务，该服务负责接收Order和Product服务发布的领域事件以同步其自身的读模型OrderWithProductRepresentation。

在ecommerce-order-query-service服务中，在接收到OrderEvent事件后，OrderQueryRepresentationService负责分别调用Order和Product的接口完成数据同步：

public void cqrsSync(OrderEvent event) {

String orderUrl = "http://localhost:8080/orders/{id}";

String productUrl = "http://localhost:8082/products/{id}";

OrderRepresentation orderRepresentation = restTemplate.getForObject(orderUrl, OrderRepresentation.class, event.getOrderId());

List<Product> products = orderRepresentation.getItems().stream().map(orderItem -> {

ProductRepresentation productRepresentation = restTemplate.getForObject(productUrl,

ProductRepresentation.class,

orderItem.getProductId());

return new Product(productRepresentation.getId(),

productRepresentation.getName(),

productRepresentation.getDescription());

}).collect(Collectors.toList());

OrderWithProductRepresentation order = new OrderWithProductRepresentation(

orderRepresentation.getId(),

orderRepresentation.getTotalPrice(),

orderRepresentation.getStatus(),

orderRepresentation.getCreatedAt(),

orderRepresentation.getAddress(),

products

);

dao.save(order);

log.info("CQRS synced order {}.",orderId);

}

在本例中，ecommerce-order-query-service查询服务使用了关系型数据库，但在实际应用中应该根据项目所需选择适当的数据存储机制。例如，对于海量数据的查询，可以选择诸如MongoDB或者Cassandra之类的NoSQL数据库；而对于需要进行全文搜索的场景，可以采用Elasticsearch等。

事实上，在接收并处理事件时，存在2中风格，一种是本例中的仅将事件作为消息通知，然后调用其他服务的API接口完成同步，另一种是直接使用事件所携带的数据进行同步，更多关于这2种风格的比较，请参考笔者的《[事件驱动架构(EDA)编码实践](https://www.jianshu.com/p/fbd3951259eb" \t "_blank)》文章。

事件驱动架构总是意味着异步，它将给软件带来以下方面的影响：

读模型和写模型之间不再是强事务一致性，而是[最终一致性](https://links.jianshu.com/go?to=https://en.wikipedia.org/wiki/Eventual_consistency" \t "_blank)。

从用户体验上讲，用户发起操作之后将不再立即返回结果数据，此时要么需要调用方（比如前端）进行轮询查询，要么需要在用户体验上[做些权衡】([http://danielwhittaker.me/2014/10/27/4-ways-handle-eventual-consistency-ui/](https://links.jianshu.com/go?to=http://danielwhittaker.me/2014/10/27/4-ways-handle-eventual-consistency-ui/" \t "_blank))，比如使用确认页面延迟用户对查询数据的获取。

## 关于Representation对象的命名

命名总是一件令开发者头疼的事情，特别对于需要返回多种数据形式的查询接口来说。为此，笔者自己采用以下方式命名不同的Representation对象，以Order为例：

* OrderRepresentation：仅仅包含聚合根实体自身状态详情，一种常见的形式是通过Order.toRepresentation()方法获得
* OrderSummaryRepresentation：用于返回聚合根的列表，仅仅包含Order本身的状态
* OrderWithProductRepresentation：用于返回带有Product数据的Order详情
* OrderWithProductSummaryRepresentation：用于返回带有Product数据的Order列表

当然，命名是一件见仁见智的事情，以上也绝非最佳方式，不过总的原则是要一致、清晰、可读。

## 什么时候该采用CQRS

事实上，不管是[Martin Fowler](https://links.jianshu.com/go?to=https://martinfowler.com/" \t "_blank)、[Udi Dahan](https://links.jianshu.com/go?to=http://udidahan.com/" \t "_blank)还是[Chris Richardson](https://links.jianshu.com/go?to=https://microservices.io/about.html" \t "_blank)，都提醒到需要慎用CQRS，因为它会带来额外的复杂性；而另有人（比如[Gabriel Schenker](https://links.jianshu.com/go?to=https://lostechies.com/gabrielschenker/2015/04/07/cqrs-revisited/" \t "_blank)）却提到，当前很多软件逻辑复杂性能低下恰恰是因为没有选择CQRS造成的。

的确，不管在架构层面还是编码层面，采用CQRS的都会增加程序的复杂度和代码量，不过，这种复杂性可以在很大程度上被其所带来的“条理性”所抵消，“有条理的多”恰恰是为了简单。因此，当你的项目正在承受本文一开始的“一个例子”小节中所提到的“痛楚”时，不妨试一试本文提到的几种简化版的CQRS实践。

## 总结

本文本着“简单可用的CQRS”的目的讲到了不同的CQRS实现模式，其中包含如何在单体和微服务架构中进行不同的CQRS落地实践。可以看出，CQRS并不像人们想象中的那么难，通过适当的设计与选择，CQRS可以在很大程度上将程序架构变得更加的有条理，进而使软件项目在CQRS上的付出变成一件值得做的事情。

作者：无知者云  
链接：https://www.jianshu.com/p/d5e344ccf62a  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。