DDD——架构

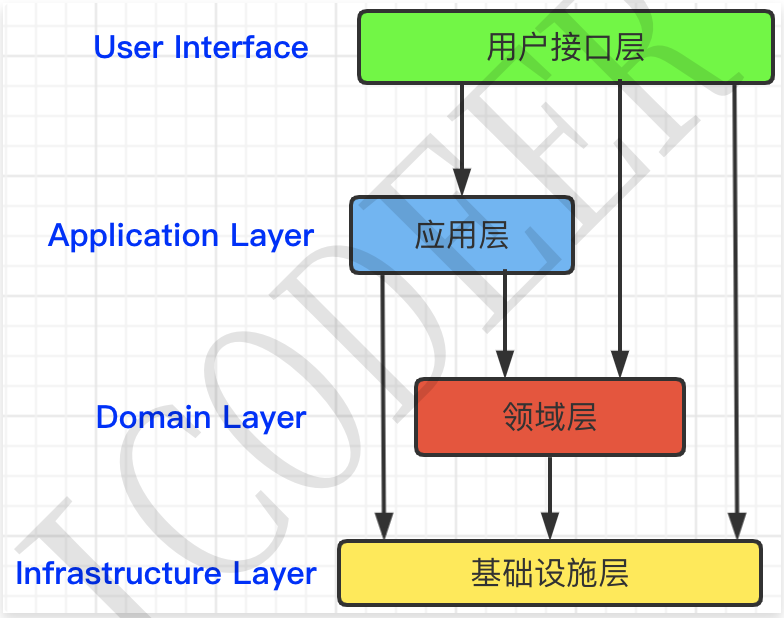
〇、本章学习路线

- 听听SaaSOvation的CIO是如何做项目回顾的。
- 学习使用依赖注入原则（DIP）和六边形（Hexagonal）架构来改进分层架构（Layers Architecture）
- 学习六边形架构对SOA和REST的支持。
- 学习数据网织（DataFabric）或基于网格的分布式缓存（Grid-Based Distributed Cache）和事件驱动风格。
- 学习DDD世界的新架构模式——CQRS。
- 学习SaaSOvation所采用的架构。

一、分层架构

1.1> 概述

- 一提到分层架构，大家应该都不会陌生。因为当我们开始从事软件开发这一行业的时候，接触到的企业项目基本都是采用分层架构的。它产生的时间比较早，可以说，**分层架构模式被认为是所有架构的始祖**。
- 分层架构的一个重要的原则就是——**每层只能与位于其下方的层发生耦合**。那么，以下图为例，我们一般在项目开发中，会将整个项目分为：**用户接口层**、**应用层**、**领域层**和**基础设施层**。



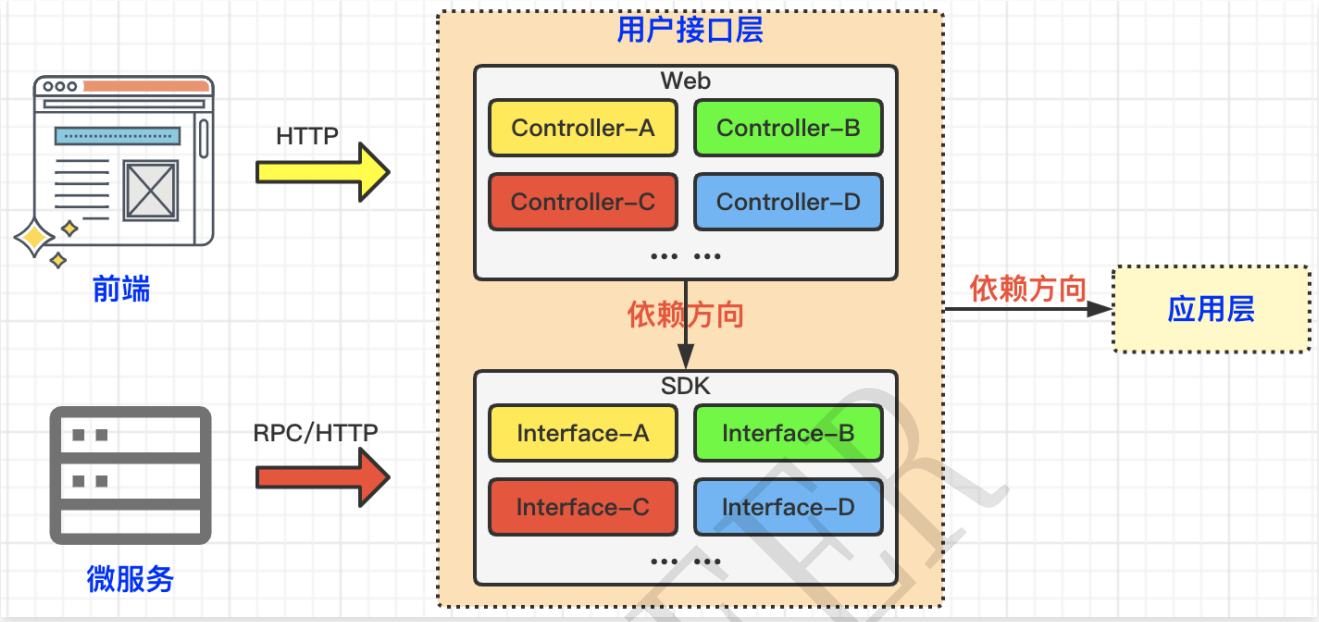
- 针对分层架构分为：**严格分层架构**和**松散分层架构**。由于用户界面层和应用服务通常需要与基础设施打交道，许多系统都是基于松散分层架构的。

> **严格分层架构**（Strict Layers Architecture）：某层只能与直接位于其下方的层发生耦合；
>
> **松散分层架构**（Relaxed Layers Architecture）：允许任意上方层与任意下方层发生耦合。

1.2> 用户接口层

- 一般负责承载对外暴露接口或者服务的职责，那么也是与前端沟通紧密的一层。用户界面只用于对数据进行展示以及收集请求数据，而不应该包含领域业务或业务逻辑，但是可以包含请求参数的校验和数据封装的逻辑。

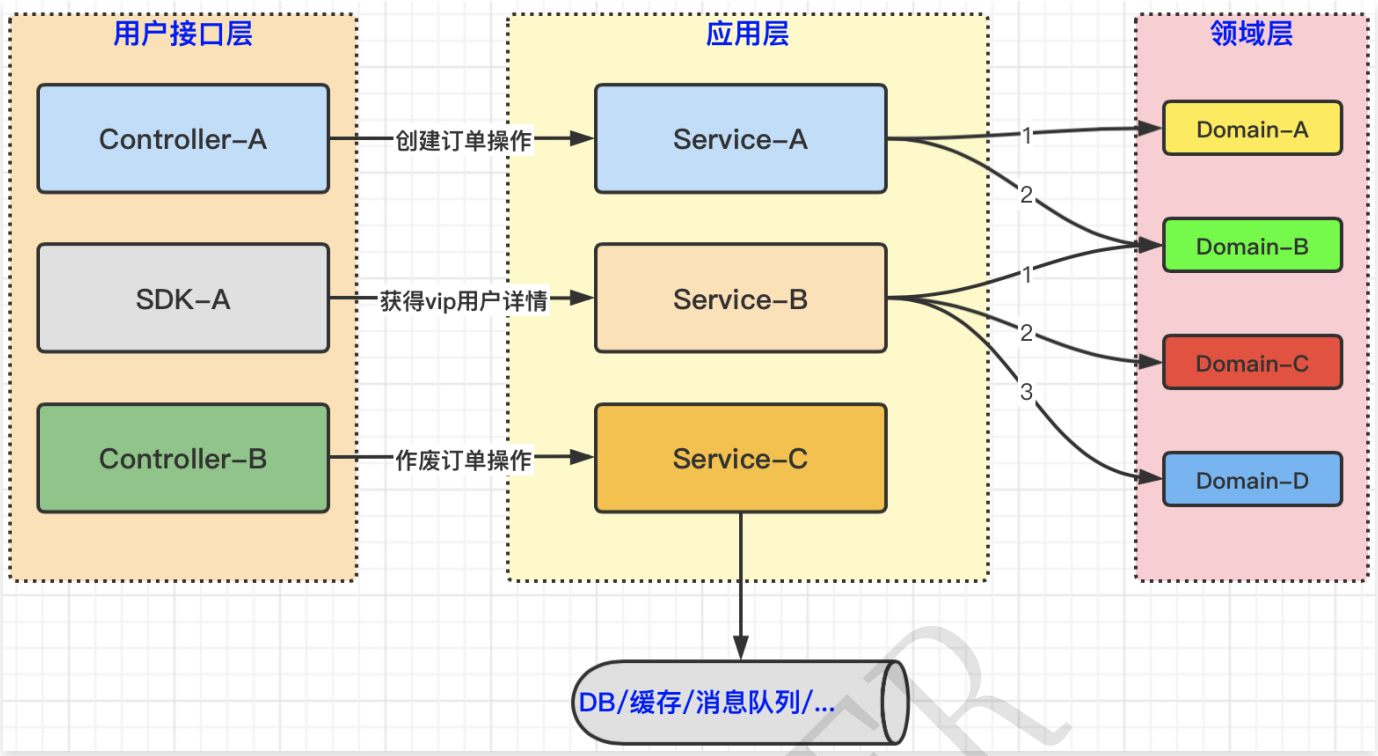
- 该层即包含与前端交互的接收**Http请求**的`Web模块`，也包含着服务间**RPC请求**调用所需要的`SDK模块`。在Web模块中，主要存放的是对外Controller接口集合；在SDK模块中，由于是需要调用方服务端进行maven依赖的，所以只需要包含最基本的interface接口类和Entity实体类即可，不相干的类不要放入这层，以免客户方引入一大堆无用的类。具体如下图所示：



- 用户界面层是应用层的直接客户。

1.3> 应用层

- 应用服务存在于应用层，它负责针对某一业务的逻辑实现和拼装，比如：一个业务操作需要涉及多个领域服务的支持，那么相关业务逻辑的聚合就是在应用层中。所以，**应用层中是不应该出现领域逻辑的**，它本身并不处理业务逻辑，而是作为领域模型的客户，交由领域层进行处理。应用服务可以用于控制持久化事务、安全认证、发送消息通知等，同时也是表达用例和用户故事的主要手段。应用服务应该是**很轻量的**。如果我们发现应用服务变得很复杂了，这通常说明领域逻辑已经渗透到了应用服务中了。最佳实践通常是——**应用服务调用领域服务来完成和领域相关的任务操作，但此时的操作应该是无状态的**。
- 一般来说，用户层的请求会发送到应用层，这里面即包括前端发过来的请求，也包含后端服务间的请求。在应用层中，是针对业务逻辑来调用和整合一个或多个领域层的服务，当然，也并不是说应用层一定要调用领域层，也可以通过调用基础设施层来直接操作数据库或中间件等。具体如下图所示：



- 应用层是领域层的直接客户。

1.4> 领域层

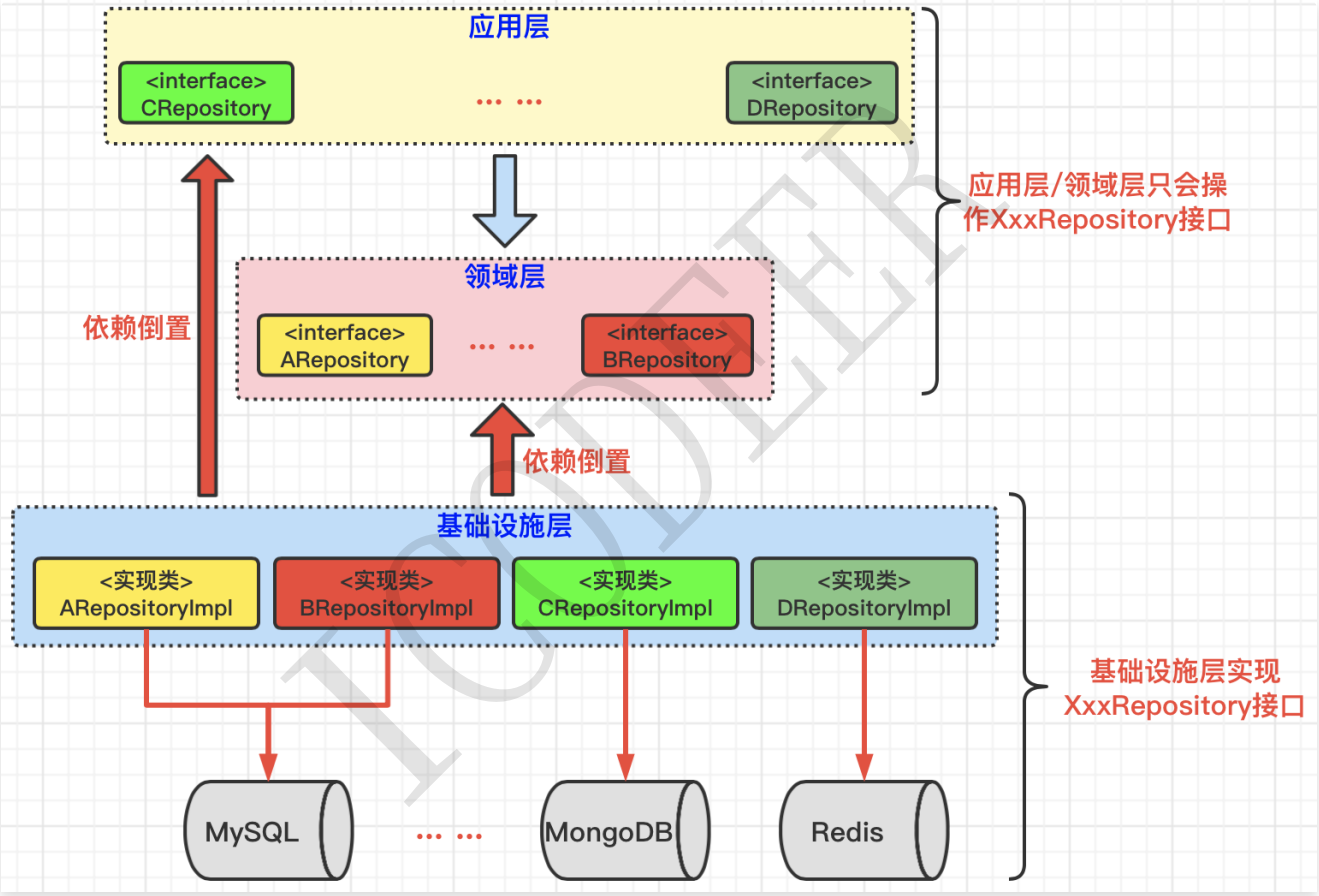
- 包含了某一领域内的领域逻辑，该层只与自己的领域有关，对于其他领域的逻辑调用，都不会在这一层内处理，该层要具有领域的隔离性。
- 领域层是整个系统的核心部分，领域相关的所有核心逻辑都放在这一层，我们开发的重心也是在这一层。此处我们先不对其展开讲，后续我们掌握了更多领域驱动知识了之后， 就会对其有更深的认知了。

1.5> 基础设施层

- 基础设施层包含的内容比较宽泛，包含：关系型数据库，NoSQL，文件存储，缓存，第三方代理接口等等。这一切都是对于整个项目的最基础的设施支持。
- 针对我们上面在**1.1> 概述**章节中画的各层依赖关系图中，我们可以看到， 图中的应用层和领域层都依赖了基础设施层，那么基础设施的相关接口和实现类，就都会放在基础设置层中。这种在模块间的调用上没有太大的问题。但是，对于基础设置层中所需要的接口和方法，其实是应用层或领域层来决定的，比如，针对tb_user表的操作接口——UserRepository，由与业务密切相关的**应用层/领域层**决定相关操作方法，例如：需要添加用户：`saveUser(...)`，删除用户：`deleteUser(...)`，通过用户id查询用户—— `findUserById(...)` 等等。那么，针对技术设施层的

接口类，建议放到领域层/应用层中。由这两层去定义xxxRepository接口，然后由基础设施层去依赖应用层/领域层，去实现相关接口。

- 那么这种方式，虽然貌似破坏了分层架构的约束（即：**每层只能与位于其下方的层发生耦合**），但是，我们通过**依赖倒置**的方式，使得**应用层/领域层只关注基础设施的接口方法，而并不关系其具体实现**。比如，在UserRepository接口中的saveUser(...)方法，其实现类UserRepositoryImpl是以MyBatis作为持久层框架来操作MySQL数据库，如果某一天领导要求将MySQL更换为MongoDB，那么，我们只需要改变UserRepositoryImpl内部实现即可，而对于应用层/领域层是没有任何影响的，因为在应用层/领域层中，它们只会操作UserRepository接口。具体如下图所示：

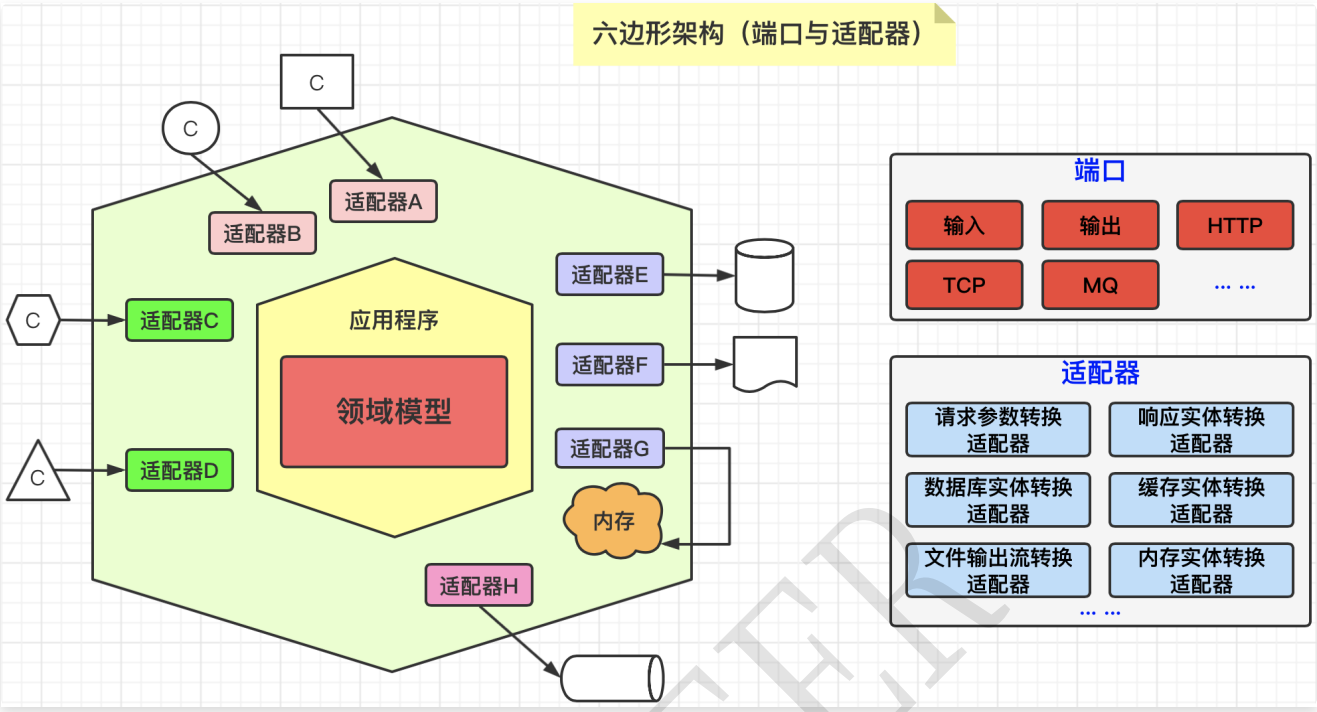


- 由于应用层是领域层的直接客户，它将依赖于领域层接口，并且间接地访问资源库和由基础设施层提供的实现类。

二、六边形架构

- 六边形架构又称“端口与适配器”，六边形每条**不同的边代表了不同种类的端口**，端口要么处理输入，要么处理输出。如下图所示：适配器A和B是一个边，适配器C和D是另一个边，这里有可能

就是 适配器A 和 适配器B 接收到的是输入端发来的HTTP请求， 适配器C 和 适配器D 是输入端发来的TCP请求。



- 不过针对六边形架构中的端口，并没有明确的定义，它是一个**非常灵活的概念**。无论采用哪种方式对端口进行划分，当客户请求到达时，都应该有相应的适配器对输入进行转化，然后端口将调用应用程序的某个操作或者向应用程序发送一个事件，控制权由此交给内部区域。
- 以下就是请求到达HTTP的输入端口时，相应的适配器将对请求的处理委派给应用服务——OrderService。

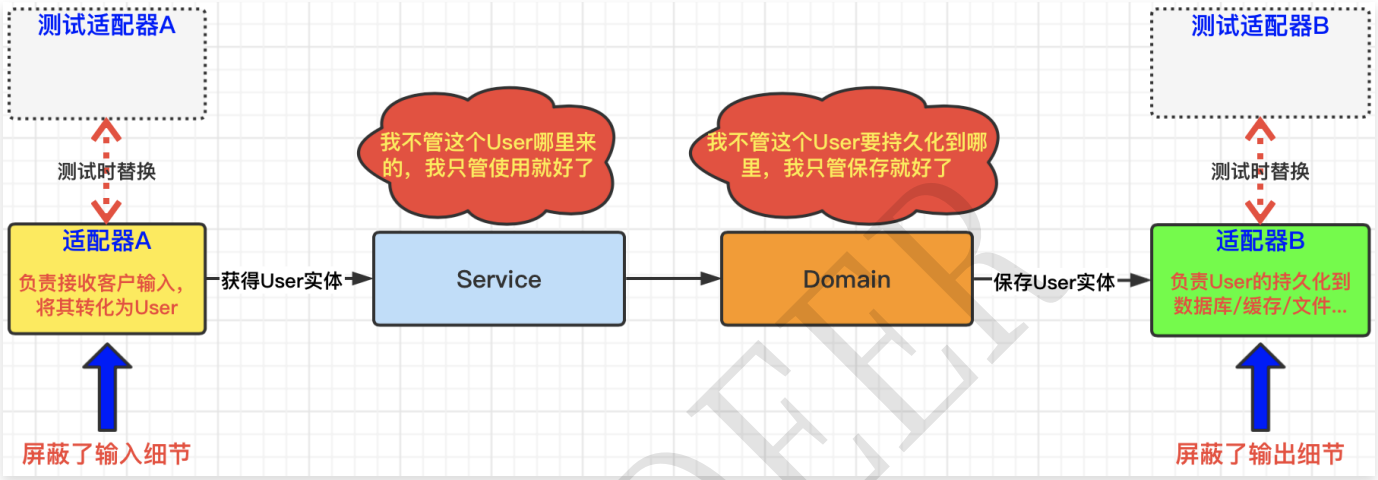
```java
@Slf4j
@RestController
public class OrderController {
    2 usages
    @Resource
    private OrderService orderService;

    /**
     * 下单操作
     */
    @PostMapping(value = Constant.HTTP_PREFIX + "/order/add")
    public BaseResponse addOrder(@RequestBody AddOrderDTO addOrderDTO) {
        BaseResponse response = BaseResponse.getSuccessResponse();
        AddOrderResultDTO addOrderResultDTO = orderService.addOrder(addOrderDTO);
        response.setData(addOrderResultDTO);
        return response;
    }
```

- 我们再看上图六边形架构中的 适配器E、F、G ，我们可以通过不同的方式实现资源库，比如：关

系型数据库、基于**文档的存储**、基于**分布式缓存**和**内存存储**等。如果应用程序向外界发送**领域事件**消息，我们将使用 `适配器H` 进行处理。由于适配器H是处理消息输出的，我们可以将其使用不同的端口。

- 由于六边形架构采用了**输入/输出适配器**，所以，**可以很轻易的开发用于测试的输入适配器和输出适配器**。那么，在整个应用程序和领域模型就可以在没有客户和存储机制的条件下进行设计和开发。这样，在开发过程中，我们就可以在核心领域上进行持续开发，而不需要考虑那些支撑性的技术组件。

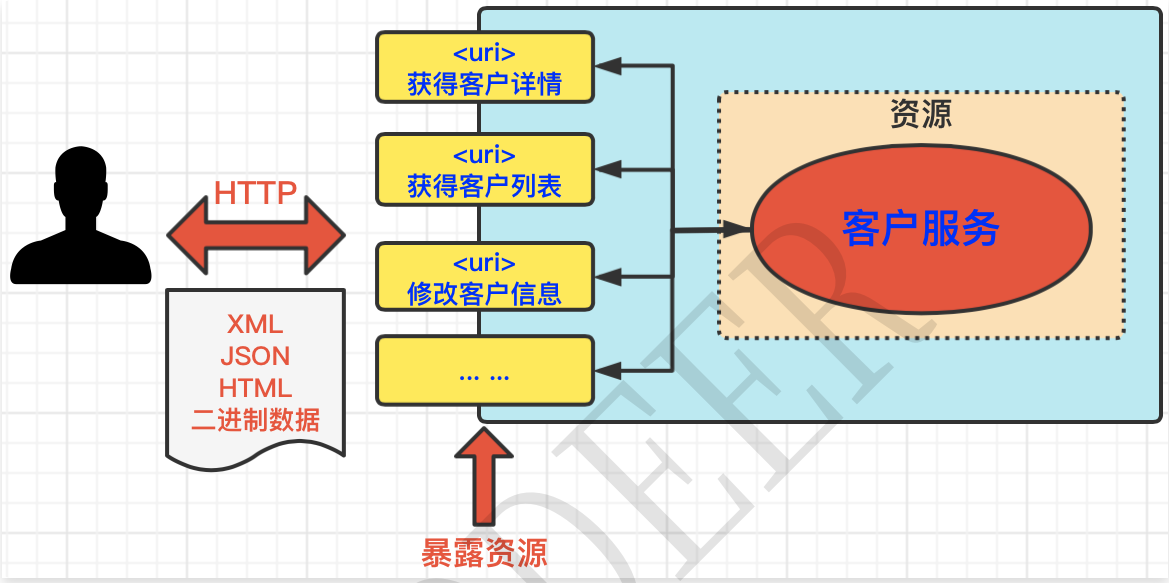


- 如果你采用的是严格分层架构，那么你应该考虑推平这种架构，然后开始采用端口与适配器。通过合理的适配器设计，我们可以保障**内部六边形**（应用程序&领域模型）是不会泄漏到外部区域的，这样也有助于形成一种清晰的应用程序边界。
- 六边形架构可以支持系统中的其他架构，如：SOA、REST、事件驱动、CQRS、数据网织、基于网格的分布式缓存、Map-Reduce……六边形架构为这些架构提供了坚实的支撑基础。

三、REST

- 对于REST来说，它其实是一种**基于Web架构的架构风格**。这时候会有同学说，我使用HTTP对服务请求的时候，也没有采用什么所谓的REST架构风格，在项目使用中也没出现什么大问题啊？那为什么需要REST呢？其实，我相信这也绝对不是少数人会有疑问，其实我们将REST称之为“基于Web架构的架构风格”，本质是提供一种使用Web协议的**更合理**的方式。这就类似于当我们在MySQL中建表的时候，我们可以遵循数据库三范式的方式去创建业务表，添加主键、外键、索引、复合索引、非空约束、视图、触发器……，也可以像使用NoSQL一样，只创建两个列，一个列作为 `Key` ，用于存储业务数据的唯一标识；另一个列作为 `Value` ，用于存储序列化后的对象信息。这两种方式我们其实都是在使用MySQL数据库，区别就在于是否合理、是否可以使用到数据库给我们提供的各种功能。

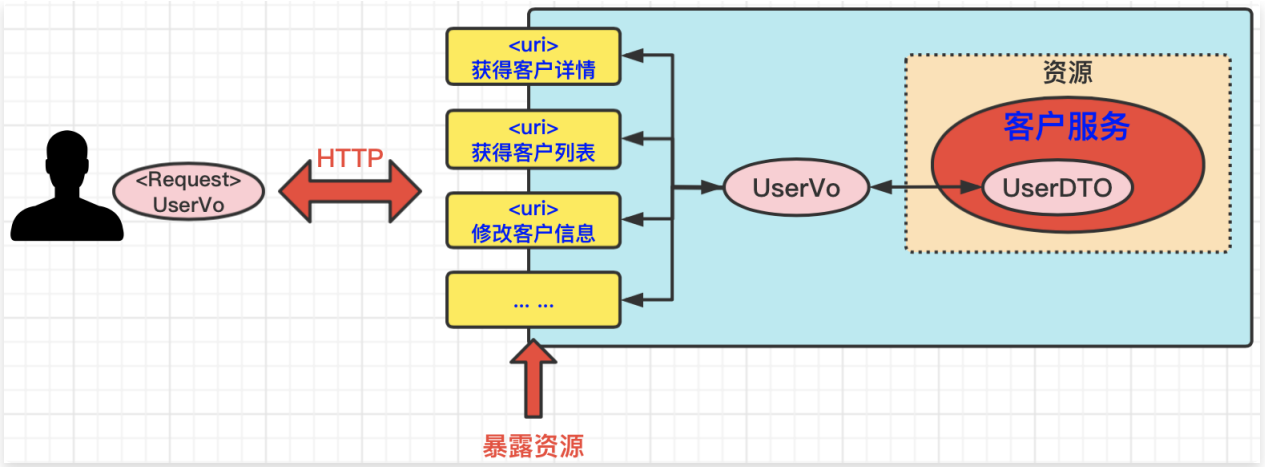
- 同样的道理，当我们使用HTTP对服务的进行请求的时候，如果遵循了REST风格的架构风格，便可以获得由于使用了REST风格的HTTP所带来的好处。那么具体来说，使用还是不使用这种架构风格，还是与项目实际情况来确定的。例如，我只是希望通过HTTP的方式触发一个补偿机制，那么，即使不采用REST，也无所谓。
- HTTP基于服务端而言，是一种**可以将服务资源对外暴露的重要方式**之一，比如：我们想要获取客户的详细信息，那么客户服务负责对客户资源的管理，所以，由客户服务提供一个URI，将客户信息以 `XML` 、 `JSON` 、 `HTML` 或者 `二进制数据` 返回给客户端。



- 那么，既然我们可以通过HTTP的方式去获取和操作资源，那么如果我们将资源也看做是一种对象，那么也会有对资源的增删改查等操作。所以，当我们引入RESTful时，就可以通过HTTP中请求method的一些动词—— `GET` 、 `PUT` 、 `POST` 、 `DELETE` ，来对资源进行不同行为的操作。Rest风格支持（使用HTTP请求方式动词来表示对资源的操作）

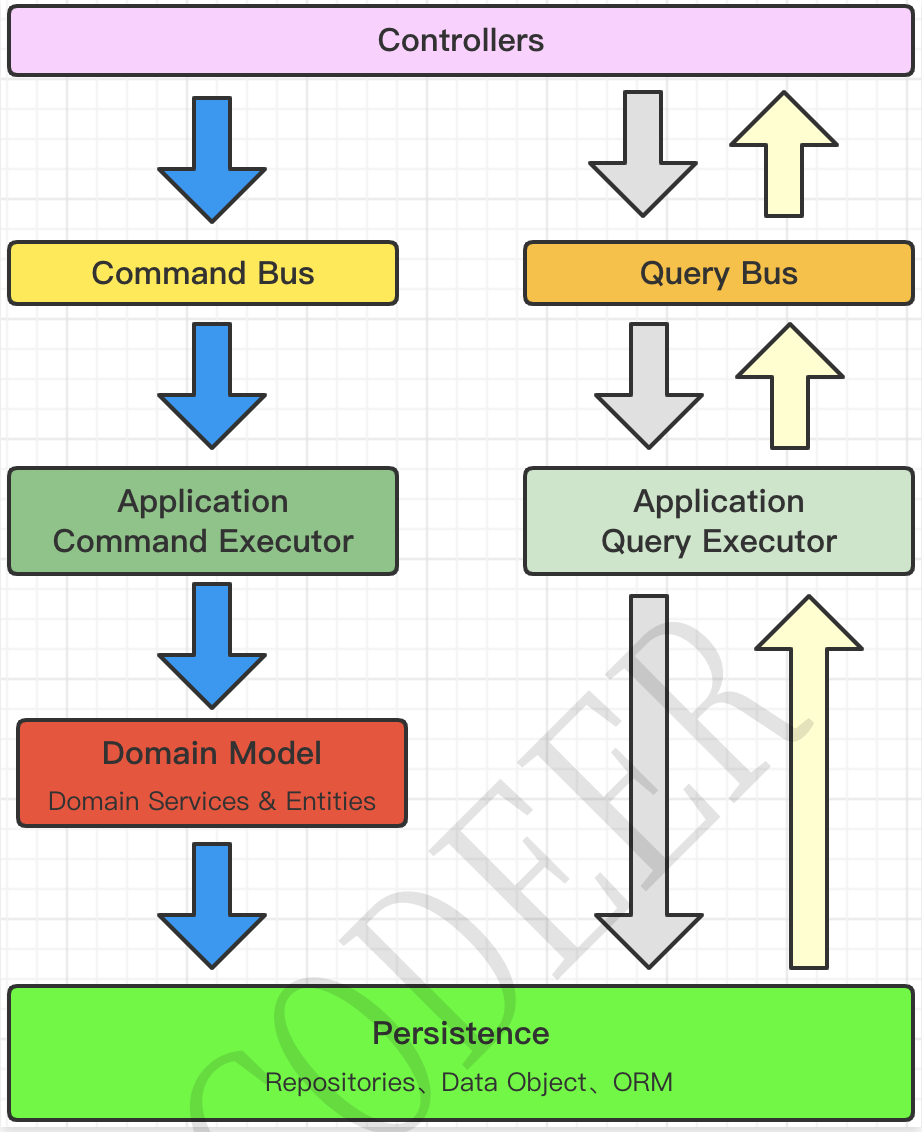
是否使用REST	获取用户信息	删除用户信息	更新用户信息	保存用户信息
否	/getUser	/deleteUser	/updateUser	/saveUser
是	/user method=GET	/user method=DELETE	/user method=PUT	/user method=POST

- 虽然刚刚我们将资源类比为了一种对象，但是，究其本质资源并不表示任何可以持久化的实体，它更像是**封装了某种行为**，当我们将HTTP动词应用在这些资源上时，我们实际上是在调用这些行为——处理某些业务逻辑、对其他系统发起领域事件、缓存某些数据，获取业务数据……
- 这里我们需要注意的是，当我们暴露资源的时候，并不是要将领域模型直接暴露给外界，因为这样当我们修改领域模型时，就会影响到暴露出来的接口。所以，**我们需要将客户请求和响应对象与领域模型隔离开**，例如：客户请求对象我们采用XxxVo、XxxQry和XxxCmd来进行命名，领域模型内的对象我们采用XxxDTO、XxxEntity来命名。通过使用不同的对象来起到表现层与应用层/领域层的隔离。



四、CQRS

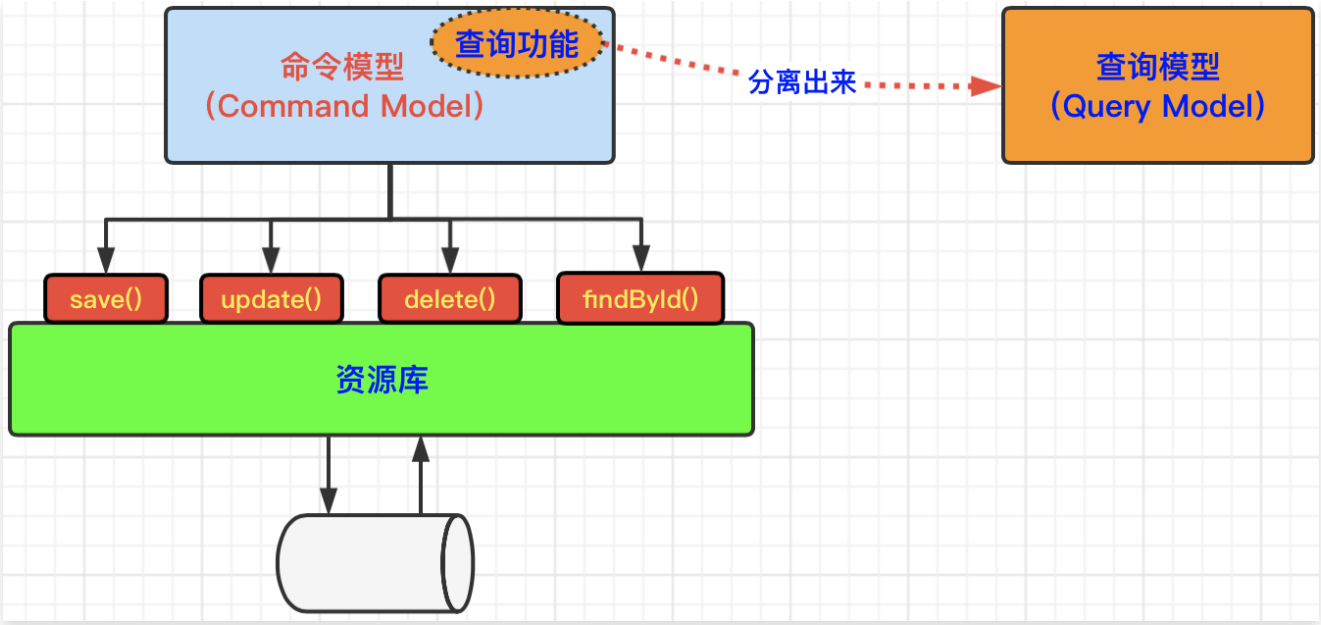
- CQRS（Cammand-Query Responsibility Segregation）：**将查询操作与命令操作进行分离**。其架构图如下图所示：



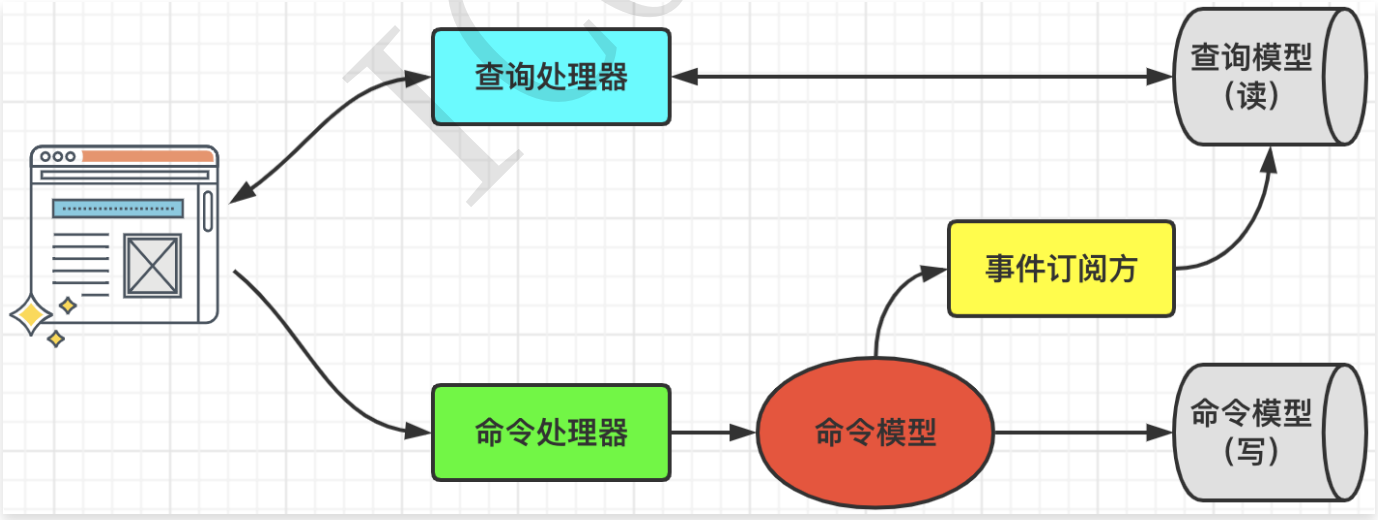
- 在CQRS模式中，一个方法要么是执行某种动作的**命令（Cammand）**，要么是返回数据的**查询（Query）**，而不能两者皆是。

  - 如果一个方法修改了对象的状态，该方法便是一个命令（Command），它不应该返回数据。
  - 如果一个方法返回了数据，该方法便是一个查询（Query），此时它不应该通过直接的或间接的手段修改对象的状态。

- 在以往我们涉及到的开发模型中，同时包含着命令和查询的聚合。那么，在CQRS中，我们会考虑将那些纯粹的查询功能从命令功能中分离出来。聚合将不再有查询方法，而是只有命令方法。资源库只提供**新增**（`save()` / `add()`）、**更新**（`edit()` / `modify()` / `update()`）、**删除**（`delete()` / `remove()`）方法。针对于**查询**方法，只提供根据唯一标识来进行查询的方法（`findUserById()` / `findUserByUserId()`）。



- 有的同学会有疑问，这么把命令和查询拆分开来，分别的构建，不是为系统增加了复杂度嘛？但无论如何，不要急于否定这种架构。其实，我们需要记住一点，就是**CQRS旨在解决数据显示复杂性问题**。只有当有这方面业务需求的时候，我们才会选择这种架构，而并非所有架构都要按照CQRS的方式去构建。
- 由于在上面的介绍中，我们已经将查询功能拆分出来了。那么下面我们就将原有的领域模型一分为二，即：**命令模型** & **查询模型**。那么，对于命令操作，可以通过单独的路径抵达命令模型。而查询操作，则请求到查询处理器中，并且可以采用不同的数据源，并且便于对查询数据进行优化而不会影响到命令模型。



4.1> 查询模型

- 对于查询模型返回给客户端的结果，一般来说有两种处理方式，无论采用哪种方式，没有绝对的好坏，根据具体情况而定。

> **方式一**：直接返回查询后的结果集或者基本的序列化数据（JSON/XML）。
>
> **方式二**：返回封装好的DTO或者VO对象。

- 针对于查询模型，它并不反映领域行为，只是**用于数据显示**或**生成数据报告**。
- 在查询模型中，如果采用的是关系型数据库，那么**视图**就代表着数据库中的一张表。为了满足不同的查询需求，我们可以针对一个或多个视图进行组合拼装、数据过滤。

4.2> 命令处理器

- 客户端提交的命令将被命令处理器接收。一般来说，我们会采用如下两种风格去实现：

> **分类风格**：多个命令处理器位于同一个应用服务中。我们可以根据不同的命令类型来寻找对应的命令处理器。优点：简单，便于维护。
>
> **专属风格**：每种命令处理器对应一个处理类，这个类只提供一个用于处理某个指令的方法。优点：每个处理类职责单一，处理器之间互相独立。

- 在调用命令处理器的方式上，也可以分为两种：

> **同步调用**：提升整个流程的处理时间。可以在同一个事务下保证数据的一致性。
>
> **异步调用**：可以实现与命令处理器的解耦，但是，只有在有伸缩性需求的情况下才考虑采取异步方式。

- 但是，无论采取哪种风格以及哪种调用方式，**一个处理器不能依赖于另一个处理器**。这样可以保证对于任何处理器的重新部署都不会影响到其他处理器。
- 命令处理器通常只完成有限的功能。例如，我们要通过某个命令处理器执行某种命令，那么，命令处理器将从资源库中获取聚合实例，然后再调用该聚合实例的某个行为方法。如下所示：

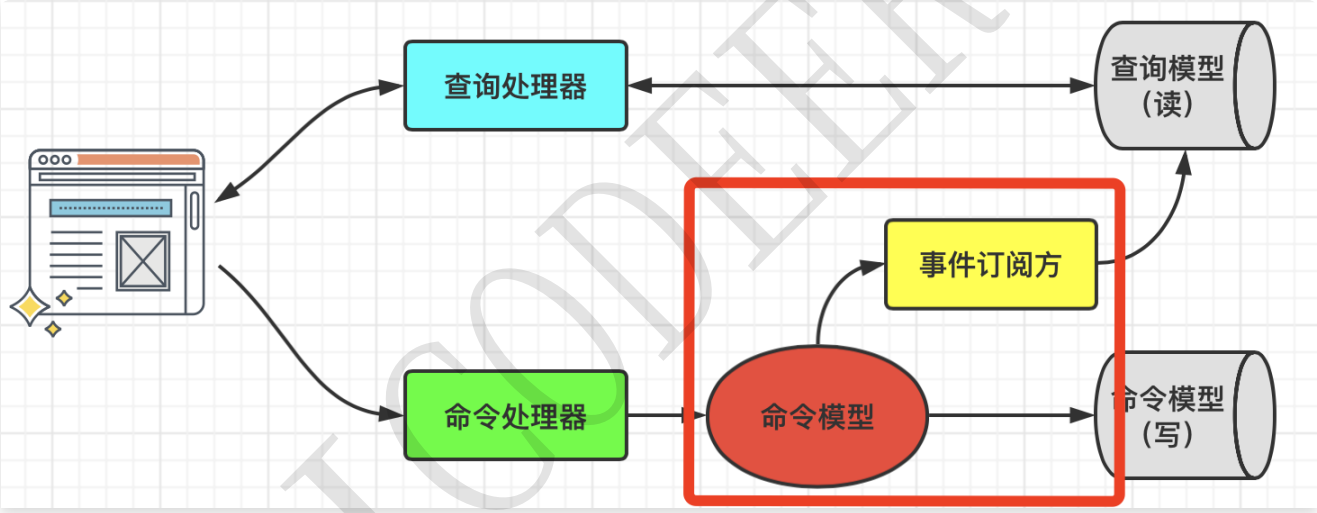
```java
@Transactional
public void orderToPay(String orderId, String paymentId) {
    Order order = orderRepository.orderOfId(orderId);
    Payment payment = paymentRepository.paymentOfId(paymentId);
    order.pay(payment);
}
```

4.3> 命令模型执行业务行为

- 命令模型上每个方法在执行完成时都将发布领域事件。下面我们以 `Order.pay(...)` 为例：

```java
public class Order extends ConcurrencySafeEntity {
    ...
    public void pay(Payment payment) {
        ...
        // 发布领域事件
        DomainEventPublisher.instance().publish(new OrderPaid(this.orderId,
 payment.paymentId));
    }
    ...
}
```

- 当我们对命令模型执行更新操作后，需要通过发布领域事件，来通知查询模型也执行相应的更新操作。该领域事件的发布，是基于请求合法的情况下，并且针对查询模型接收领域事件，需要添加幂等的能力，否则因为网络抖动或者服务异常会导致多次相同事件触发通知。请见下图红框所示：



- 对查询模型的更新应该是同步的呢，还是异步的？这取决于系统的负荷，也有可能取决于查询模型数据库的存储位置。数据的一致性约束和性能需求等因素对此也有很大的影响作用。如果要同步更新查询模型，**查询模型和命令模型通常需要共享一个数据库**，这时我们会在同一个事务过程中处理更新。这种方式可以保证两种模型的数据达到完全一致性。
- 如果命令模型和查询模型采取异步更新，那么最终一致性问题就摆在了我们的面前。会出现命令已经执行成功，但是用户查询时，发现查询模型中还是“旧”的数据。针对这个问题，我们可以采取**先将更新数据放入缓存中**，用户读取数据的时候，先查询缓存，如果不存在，再去查询模型的数据库中获取。对于缓存数据，我们设定一个合理的过期时间。但是这种方式，也没法真正的解决这个问题，并且随着引入缓存中间件，也对系统的稳定性产生了影响。其次，我们可以采取**业务数据+创建日期的方式**，即：在展示数据后面，增加当前所展示的数据的创建时间。这样，用户

可以根据数据创建时间，来知道这个数据是新数据还是旧数据。当然，还有其他多种的处理方式，具体选择哪种方式，我们还是需要根据具体的业务场景来决定。

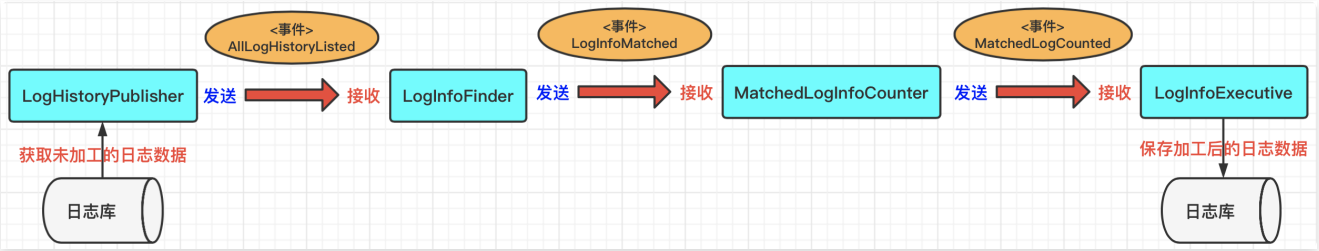
五、事件驱动架构

5.1> 概述

- 事件驱动架构（Event-Driven Architecture，**EDA**）是一种用于处理事件的**生成**、**发现**和**处理**等任务的软件架构。
- 一个系统的**输出端口**所发出的领域事件将被发送到另一个系统的**输入端口**，此后输入端口的事件订阅方将对事件进行处理。往往这种领域事件都是基于MQ的方式实现的。它除了在功能上实现了一步的事件传输之外，也可以实现类似Linux中**管道**和**过滤器**的方式，即：`cat log_history.log | grep orderId=123456 | wc -l`

```
muse@muse:/Users/muse/Desktop> cat log_history.log
orderId=123456
orderId=123456
orderId=123456
orderId=121111
orderId=123456
orderId=123456
orderId=121111
muse@muse:/Users/muse/Desktop> cat log_history.log | grep orderId=123456
orderId=123456
orderId=123456
orderId=123456
orderId=123456
orderId=123456
muse@muse:/Users/muse/Desktop> cat log_history.log | grep orderId=123456 | wc -l
       5
```

- 利用领域事件，我们可以采用如下方式实现：



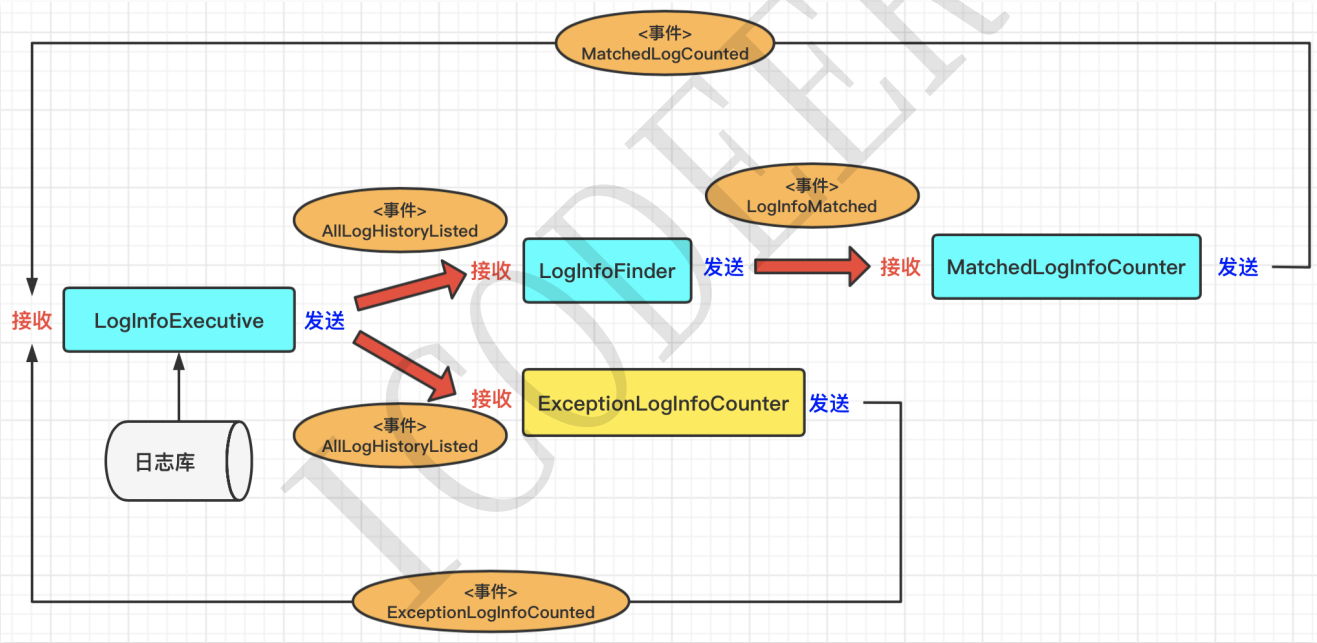
- 上面的例子，只是使用领域事件来类比Linux中的管道概念，在真实的企业应用里，我们将通过这种模式**将一个大问题分解成若干个较小的步骤来完成**，这使得分布式处理更容易理解和管

理。

- 在DDD应用场景中，领域事件的名字将反映业务操作。

5.2> 长时处理过程——Saga

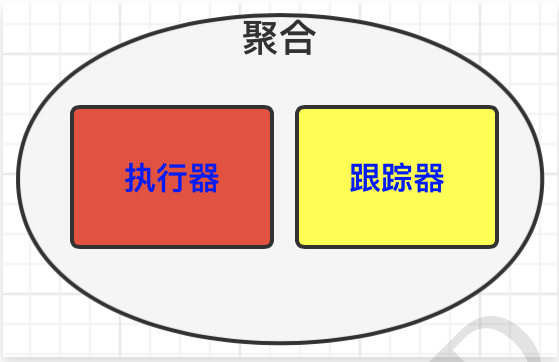
- 长时处理过程（Long-Running Process）也称为**Saga**，它是一种事件驱动的、分布式的**并行**处理模式。
- 我们对上面介绍的领域事件例子进行改造，由 `LogInfoExecutive` 负责启动，并且添加了新的过滤器 `ExceptionLogInfoCounter` ，用于统计所有发生了Exception异常的日志数，大家注意，此时它与 `LogInfoFinder` 是平行处理的，那么整个长时处理是否完成，就取决于**统计指定查询日志信息的日志数**和**统计所有发生了Exception异常的日志数**是否全部都完成，那么这就需要 `LogInfoExecutive` 负责对多个并行处理任务是否完成进行判断了。



- 设计长时处理过程有三种方法：

> - **方法1**：将处理过程设计成**一个组合任务**，使用一个执行组件对任务进行跟踪，并对各个步骤和任务完成情况进行持久化。
> - **方法2**：将处理过程设计成**一组聚合**，这些聚合在一系列的活动中相互协作。一个或多个聚合实例充当执行组件并维护整个处理过程的状态。
> - **方法3**：设计**一个无状态的处理过程**，其中每一个消息处理组件都将对所接收到的消息进行扩充——即：向其中加入额外的数据信息。然后，再将消息发送到下一个处理组件。在这种方法种，整个处理过程的状态包含在每条消息中。

- 当LogInfoExecutive接收到MatchedLogCounted或ExceptionLoginfoCounted事件后，我们需要在领域事件中的每个任务中加入独特的**唯一标识**（例如：`UUID`），才能判断到底是哪个任务的哪一步执行完毕了。
- 对于最简单的方式，我们可以将**执行器**和**跟踪器**都放到一个聚合中，这样通过调用聚合的命令方法，来触发执行器和跟踪器。这样我们就不需要单独的开发一个跟踪器来作为状态机。



- 针对长时处理过程的执行器将创建一个新的类似聚合的**状态对象**，用来跟踪事件的完成情况。它与相关的领域事件共享同一个唯一标识，用于标识它是用来维护某个长时处理的状态。在这个聚合状态对象中，除了包含子任务的完成状态之外，还包含了对整体任务的是否完成状态（`isCompleted()`）和是否超时状态（`hasTimeOut()`）。每当子任务完成后，都需要更新对应的状态对象。那么，如何去更新整体的任务状态呢？一般来说，有如下两种处理方式：

  - **被动更新**：由执行器在每次`子任务`完成事件到达时执行completed/timeout。
    - 缺点：如果由于某些原因导致执行器始终接收不到完成领域事件，那么即便处理过程已经超时，执行器还是会认为处理过程正处于活跃状态
  - **主动更新**：创建一个`独立的定时器`，由它对任务的状态进行管理。
    - 缺点：它需要更多的系统资源，这可能加重系统的运行负担。同时，定时器和完成事件之间的竞态条件有可能会造成系统失败。

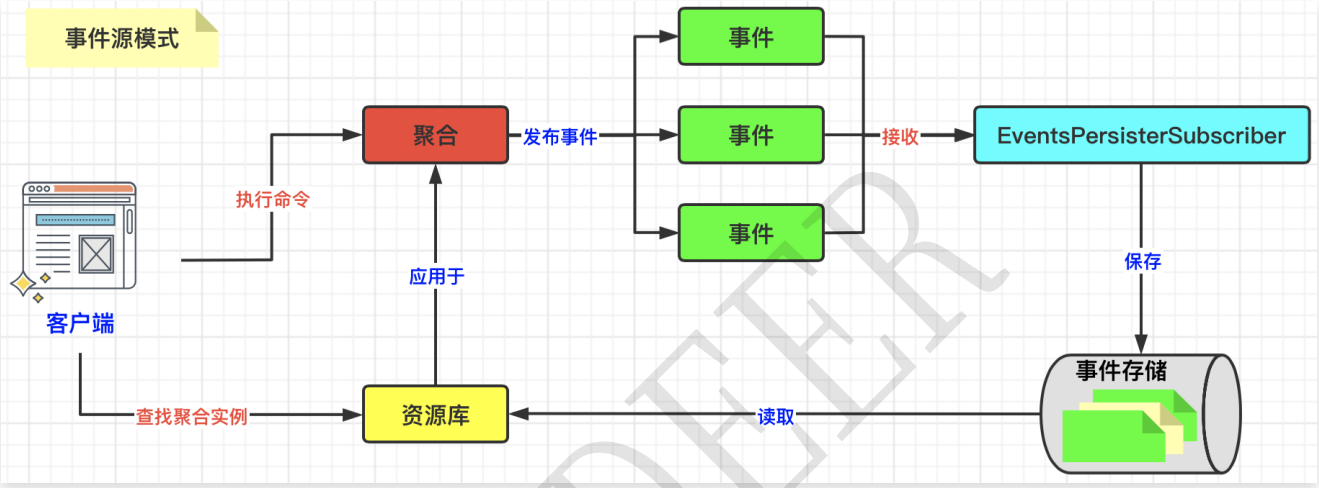
- 由于长时处理本身的特性，它追求的是**最终一致性**，那么如果这个处理过程中，由于基础设施问题或处理过程本身的问题导致失败的时候，我们是需要添加重试的方式进行适当的“**自我修复**”。那么，这就需要执行器在接收到结果通知的时候，要具有幂等的能力。
- 长时处理的优势就是伸缩性非常好，并且非常适合那种业务本身就需要较大时间延迟的情况，但是，针对最终一致性的保证，以及重试后也无法成功的异常情况回滚或数据修复，对我们来说，都是一种较大的挑战。

5.3> 事件源

- 有时，我们的业务可能需要对发生在领域对象上的修改进行跟踪。简单的跟踪是，关注于业务数据

的创建时间（create_time）、修改时间（modify_time）和删除时间（delete_time），以及相关的操作人。对于这种跟踪不敏感的业务场景，只用多列维护即可；对于相对敏感的业务场景，每次新增、修改、删除（逻辑删除），我们都会针对其操作时间和操作人记录一条详细的操作记录，这样方便后续对业务数据修改的回溯与跟踪。

- 那么，还有一种更敏感的场景，就是需要记录对数据的**改变前**和**改变后**的状态，通过操作记录，可以实现数据的重放或回滚。这种与我们常用的代码库工具Git、SVN等非常相似，可以跟踪到历史每次数据的变化。那么我们将这种概念应用在单个实体或聚合上，这种变化跟踪便是事件源（Event Sourcing）的核心。事件源模式，如下图所示：



- 如上图所示，事件源是由聚合发布多个事件，这些事件被保存，同时被用于跟踪模型的状态变化。资源库从事件存储中读取事件，并将这些事件应用于对聚合状态的**重建**。
- 事件源是对于某个聚合上的每次命令操作，都有至少一个领域事件发布出去，该领域事件描述了操作的执行结果。每一个领域事件都将被保存到**事件存储**（Event Store）中。每次从资源库中获取某个聚合时，我们将根据发生在该聚合上的历史事件来重建该聚合实例，事件的作用顺序应该与它们的产生顺序相同。这种也类似于针对聚合状态的**快照**（Snapshot），但是对于请求量级比较大的情况，频繁的去创建快照也是非常消耗资源的，所以，我们可以自定义一个阈值（例如：事件数超过50个），当超过这个阈值的时候，我们在创建这个聚合状态的快照，从而获得最优的聚合创建与获取效果。
- 事件源为我们提供了**设计领域模型的新思路**。从最基本的层面来看，事件历史可以用来消除系统中的bug，对调试也有很大的益处。事件源有助于获得高吞吐量的领域模型，从而极大地提高事务处理效率。比如：向单张数据库表中追加事件是非常快的。另外，事件源还有助于提高CQRS查询模型的伸缩性，因为此时查询模型的数据源可以在事件存储更新之后得到静默更新。这样做的另外一个好处是，我们可以复制多个查询模型的数据源实例以满足更多的新增客户。

吾尝跂而望矣，不如登高之博见也。

登高而招，臂非加长也，而见者远；

顺风而呼，声非加疾也，而闻者彰。

假舆马者，非利足也，而致千里；

假舟楫者，非能水也，而绝江河。

君子生非异也，善假于物也。

------ 摘自《劝学》

愿本文可以成为大家的“山”、“风”、“马”、“舟”，助大家在技术之路上乘风破浪，大展宏图~~

同时，也欢迎大家关注我的公众号“爪哇缪斯” ~\(^o^)/~ 「干货分享，每天更新」