1. **当前开发模式概要：**
   1. 结构：controller -> service -> repository + framework + view。必须由controller调用service，service调用repository，service和repository可调用framework内容。
   2. 从设计模式来看，类似于SOA，以service为核心，虽然实际开发中大量的业务逻辑泄露到controller和repository中（甚至是view和前端js中）。
   3. 此设计对传统简单的MVC（即框架示例所描述的那样，其中的Model其实是用作ORM和业务逻辑载体的混合体）主要做了如下改动：将Model拆分成负责业务的Service和负责持久化的Repository，舍弃了ORM而采用Active Record。此种设计的一个优点是以下关注点分离：业务逻辑和持久化（基础设施）的分离。
   4. 然而在实际应用中，出现了以下问题：
      1. 业务泄露。本应属于Service的业务逻辑泄露到其他各层中（Controller、Repository、View等），而原本内容丰富的Service反而变成了贫血类。
      2. 职责划分不清。主要表现在以下几点：
         1. 全能Service，主要表现是超多的代码（如vshop的商品和订单的Service代码都在1000行以上）和多方面的功能。例如OrderService，几乎只要是跟订单相关的都在此Service中，而没有进行进一步的精细建模。
         2. 重复功能，表现在（特别是在跨模块时）重复的Service（如MemberService）和重复的方法。这些重复Service大部分地方一样，少数地方有区别。
         3. 一些Service中充斥着各种各样的查询功能（列表和单记录查询），让这个Service看起来很怪异。
         4. 贫血模型。基本都出现在Service层。多出现在查询的地方（列表、单记录）。因为我们默认约定Controller必须通过Service进行读写，而不能直接访问Repository，而实际上很多查询操作只需要Repository直接返回的数据即可，Service不需要做任何操作。
         5. 以技术视角划分模块和系统。这将直接导致4.2.2的问题。例如，所有的作业任务都放在一个系统（如message-center），而作业必然会出现各自业务逻辑（如会员合并），因而同一套业务逻辑必然会出现在多处。
         6. “前后台分离”。此处的分离是指按照前后台站点分离出完全独立的两套系统（如member-center何v-member，虽然商城是在一起，但基本也是两个完全独立的模块，而且v-shop这个名字起得很怪异，估计后本来准备分成两个，后来偷懒了搞成两个模块）。
         7. 经典的“framework问题”。这个问题几乎困扰着全公司人，而且还会继续困扰下去。framework问题的实质是敏捷团队/公司和传统架构思维的阻抗，它实际上正印证着康威定律：组织结构决定着软件架构。该主题将在后面详细讨论。
         8. OO无用论。一般在应聘时，我们都会在简历上写上“良好的OO基础”，但实际上我们骨子里是对OO持有怀疑和抵触的，而且实际开发中也是自觉不自觉地在面向对象框架中进行着面向过程的开发。问题是，为何我们那么抵触OO，那么喜欢过程式开发？另一个问题是，为何我们需要拥抱OO，相比于过程式开发能带来哪些好处？
         9. 二维设计。或者说：“当我拿着MVC这把羊角锤时，全世界都是钉子”。当我们用MVC这一单一设计模式去解决一切问题时，就陷入了二维设计。我们无法立体地看待问题，无论是思考（设计）还是编码都是在平面上进行着（如Controller -> Service -> Repository的形式化调用）。这种思维方式往往导致形式化的流程和约束，而形式化的东西又往往束缚了人们的思想，进而不再去思考。
         10. 耦合与内聚的纠缠。用辩证法的术语说，低耦合和高内聚是矛盾（辩证）统一的，高内聚的内部一定是高耦合的，低耦合的两个东西之间一定是低内聚的。当我们无法辨识出哪些元素之间应当是高内聚，哪些之间又应当是低耦合时，整体的高耦合便产生了（同时整体的低内聚也产生了，如分散在多处的同一套业务逻辑）——“牵一发而动全身”与“牵多发以动一处”。
         11. 对重构的偏见。“推到重来”是每个程序员的梦想，也是很多程序员对重构的理解。之所以出现“重构=推到重来”正是因为以上诸多问题导致的重构的困难性。实际上，重构的精髓在于局部改良。重构应当是改革，而不是革命。纵观千年封建史，一次次的革命换来的不过是一个个的王朝——现实中的重构也往往是这样。（当然改革论是有欠妥的时候，后面再讨论）
2. **相关设计模式和架构概述：**在继续讨论之前，先简要概述下后面会用到的相关设计和架构模式。
   1. **控制反转(IoC)：**或说依赖倒置，属于SOLID五大原则之一(D)。  
      描述：高层模块不应该依赖于低层模块，二者都应该依赖于抽象。抽象不应当依赖于细节，细节应当依赖于抽象。  
      注意这里提到高层和低层，说明该原则主要是用来解决跨层的依赖问题（例如我们的Service和Repository）。一般，高层需要用到低层的东西（Service使用Repository），对低层产生依赖，那么当需要替换低层实现时，就需要改动高层代码。IoC要求，高层不应当依赖于低层实现，低层实现的变更也不应该影响高层。那么如何做到呢？接口，或说抽象。高层和低层遵守相同的抽象，并唯一据此抽象（接口）通信。  
        
      再从另一个角度理解这个问题，就是将对象的使用和创建分离，使用者（调用者，依赖方）只是使用对象，而不负责创建它，创建工作由外部负责，这样当需要更换被依赖方的实现时，无需修改依赖方代码。比如说，我们的Service需要使用Repository，传统做法是在Service里面new一个Repository，现在要将原Repository替换成其他的（如原来是Db仓储，需换成Redis或nosql仓储），则需要修改所有使用了该Repository的地方。当使用IoC时，由于该Repository是由外部创建的，只需要调整外部。

这种由原来的内部new变成由外部注入的实现方式，称为依赖注入（DI）。Yii框架里面的构造函数注入和服务容器（\Yii::$app->container）是DI的两种实现方式。

IoC是设计原则，DI是该原则的实现方式。

虽然该原则主要用来解决跨层依赖问题，但他同样适用于同层之间的解耦，如果这些依赖之间不是高内聚的话。（但是，每当出现同层的低内聚类之间的依赖时（例如聚合之间的调用），首先需要考察是否需要一个更高层次的协调者，例如一个Service）。

另外需要注意的是，不要滥用IoC。虽然IoC是用来实现松耦合的很好方式，但软件设计中除了“松耦合”原则还有“高内聚”原则，高内聚的类之间是可以强耦合的，否则，很容易出现过度设计的问题。

参见：<http://www.tuicool.com/articles/JBRBzqm> 《从百草园到三味书屋》，laravel作者著，里面有很好的示例诠释了依赖反转。

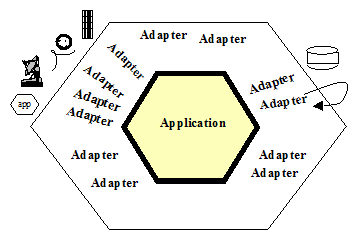
* 1. **六边形架构**（端口-适配器模式）：  
     工作中，我们经常遇到以下难题：
     1. 业务逻辑泄露到各层中，以及业务代码对输入输出的强依赖（请求对象、数据库存储对象等），很难干净的进行单元测试；
     2. 开发时依赖于数据库、缓存系统的正常运行，一旦这些挂了，就无法开发了；
     3. 当需要切换一个底层的技术实现时，需要改动相关业务层代码。

六边形架构的目的：让应用程序能够以一致的方式被用户、程序、自动化测试、批处理脚本所驱动；并且，可以在与实际运行的设备和数据库相隔离的情况下开发和测试。

例如，我们有会员合并的需求，而该需求的出现场景很多：web应用、api、后台批处理、消息队列异步处理等，目前，我们是在各场景分别写了一套合并逻辑，很难维护，而且合并逻辑本身和存储层是强耦合的，我们无法在忽略存储层的情况下进行单元测试（实际上，目前的代码根本无法进行单元测试）。按照六边形架构，web应用、api、后台处理都属于不同的输入源，这些需要和应用程序本身（会员合并业务）解耦，这些不同的输入源的不同输入数据需要通过各自的适配器适配成应用程序能理解的数据格式。适配器和应用程序接口遵守共同的契约（inteface），唯一通过该契约通信。输出端也是一样，应用程序本身不依赖于特定的输出端实现（web、api、单测、数据库），实现中也是通过各自的适配器根据各自的输出端技术将应用程序的输出转换适配成具体输出端需要的数据，如html、xml、具体数据库所需要的。

当我们将应用程序中对外部的依赖从具体实现改成对接口的依赖，并且将泄露到外围（控制器、仓储层等）的业务逻辑封装进应用程序内部后，就可以很容易进行单测，例如很容易用mocker代替实际的web输入、存储层、缓存组件等。

下面是六边形架构的图解：



这里有两个层：业务逻辑层（领域 + 应用程序，目前可简单理解为领域层或业务逻辑层），也称之为应用程序（内层、内圆）；外围设施层（web、单测、数据库、缓存服务器等，外层、外圆）。内层不依赖于外层的存在而存在（如业务逻辑层在数据库不可用时应当仍然可通过其他方式代替数据库来运行）。内层通过暴露端口（api或函数调用）为外层提供功能（服务）或从外层接收数据——这很像操作系统的端口。端口的表现形式是契约interface（api或函数的入参以及返回值）。外层并不直接和内层打交道，而是通过各自的适配器来实现通信（控制器就是典型的适配器）。适配器将内层输出转换适配成其为之服务的外层设备所需的数据，或将外层设备的输入数据适配成内层所需要的数据。适配器和内层唯一通过契约（interface）通信。

举个例子：电脑需要接收各种外设的输入进行处理，这里的电脑就是内层应用程序，各种外设（u盘、手机、网络等）属于外层。为了对外通信，主机上有各种插槽（端口），每个插槽遵循不同的规格。外设通过适配器（如各种数据线和数据转换设备）和电脑进行数据交换，而适配器需要遵循两端协议：一端是电脑插口、一端是具体的外设插口。

IoC是实现六边形架构的有效手段：内层不应当依赖于外层实现，双方都应当依赖于接口定义——这也正是IoC的描述。但六边形架构还有一点：内层的业务逻辑不应当泄露到外层，因为一旦业务逻辑泄漏到外层，那么内层就不再是通用的、与外层实现无关的了，也就无法进行多边适配了。

另外需要注意的是，这里用的是“内层”和“外层”，并没有用上层和下层的说法，这里强调的是多边适配，而不是类似网络模型中的七层架构。

我们发现，在实际应用中，我们或多或少用到了六边形架构的东西（如IoC），但为啥代码还是难以测试难以维护呢？应用了六边形的某些东西不代表整个应用遵循六边形架构，比如我们用DI，但并不是特别清楚为啥要用DI，以及哪些地方要用哪些地方不需要用，更重要的，我们的业务逻辑并没有进行很好的封装与解耦，自然难以维护。

参考文章：<http://blog.csdn.net/zhongjinggz/article/details/43889821>

* 1. **GRASP九大设计模式：**GRASP系列设计模式主要是用来解决OOD中模块划分、职责分配问题，此处我们重点看下信息专家模式、创建者模式和低耦合模式。  
     **信息专家模式**：将职责分配给拥有履行一个职责所必需信息的类，即信息专家。  
     信息专家模式解决的是由谁来承担该职责问题。首先考察该职责（方法）需要用到的数据（信息）从何而来，一般是将职责分配给主数据源类。  
     例如，论坛系统有文章Article类和作者Author类，有发布文章的职责（publish方法），那么该职责由哪个类负责呢？先假设由Author类负责（从需求描述“张三发布一篇文章”来看，貌似属于Author的职责），像这样调用：Author::publish(Article)。我们发现，publish内部使用到的数据基本都是从Article来的（除了作者信息），当Article的数据结构有所变化时，同时需要修改Author类，并且在publish内部还需要调用Article类的方法进行业务规则校验。两者之间产生了很强的依赖关系，同时违反了SOLID原则中的单一职责、开放关闭等原则。如果将publish方法放到Article类中，那么所需要的数据都是自足的，校验规则也是内在的，不需要对外公开，也就保证了修改规则时不影响其他类。  
     信息专家模式需要结合高内聚和低耦合模式一起使用。比如实体对象的持久化问题(save())，持久化所用到的数据显然是实体对象的，按照信息专家模式，save()方法应该在实体类中，但从职责上来说，持久化属于低层技术实现，不属于业务逻辑，不应该由实体承担——我们用单独的仓储来负责实体的持久化工作。  
     **创建者模式：**谁应该负责产生类的实例？  
     该模式解决的是类的创建职责问题。B包含或聚合A，或直接使用A，则由B来创建A。  
     比如文章Article和作者Author，Article实例拥有Author实例的引用，那么由谁来创建这个Author对象呢？一种可能是通过Article构造函数从外界传入，由外界创建Author对象，但这样就将Article的内部细节暴露给了外界。更好的做法是由Article内部自己创建Author对象，隐藏实现细节。  
     创建者模式的使用同样需要结合高内聚低耦合模式。  
     再看另一个例子：每当文章发布后，需要给相关订阅者发送通知。ArticleService调用Article::publish()后，需获取订阅者列表并调用Email::sendMessage()给他们发送邮件。这里ArticleService使用了Email实例，按照创建者模式，是否应该由ArticleService内部创建Email实例呢？假如是的话，考虑下当以后需要替换Email实现时会发生什么？此时就需要挨个去找哪里使用了该Email实例，然后一个一个替换。显然此种情况需要用控制反转原则，由外界注入Email实例。  
     这两个例子有何区别？  
     前一个例子中，Article和Author属于聚合关系，是较强的关系，他们共同组成了业务整体，因而可以采用创建者模式，而且也应当使用该模式以隐藏内部细节。后一个例子中，ArticleService和Email纯粹是使用关系，是很弱的关系，而且两者是跨边界调用（ArticleService属于领域层，Email属于基础设施层），在六边形架构中，ArticleService在内圆，而Email在外圆，内圆不应当依赖于外圆的实现，因而这里不能采用创建者模式，而应当采用IoC以保持低耦合。  
     在创建者模式的条件列表中，“使用”列在最后，是最弱的关系，实际使用中，如果两者仅仅是“使用”关系，则要慎用创建者模式。
  2. **SOLID原则：**  
     SOLID原则是面向对象设计和编程中最基本也最重要的五大经典原则(该单词是该五原则的首字母缩写)。下面一一讲解。  
     **S.单一职责原则：**  
     有且只有一个(一类)原因（理由）去改变一个类。  
     文章Article有publish()用来发表文章，也有save()用来保存文章到数据库中。  
     现在来考察下save()：将文章对象持久化到数据库。某一天，当持久化策略变了（用mongodb代替mysql），我们需要替换持久化引擎，此时就需要去修改这个save()方法了。“改变持久化策略”显然和Article没有直接关系，但却影响到了Article类，这就违反了单一职责原则。  
     **O.开放封闭原则：**  
     代码对扩展开放，对修改封闭。  
     文章发布后，需要给订阅者发消息。前面的做法是在ArticleService::publish()方法中获取订阅者列表，并调用Email::sendMessage()给其发消息。  
     现在，有这样的需求：只给最近三个月看过该作者文章的订阅者发消息。此时我们需要修改OrderService，在发送之前对每个订阅者做检查。再过几天，又有需求：只给关注了相关栏目的订阅者发消息......你会发现，随着需求的每次改动，ArticleService会被没完没了地改来改去（实际中我们正在做这样的事）。这里对修改是开放的。  
     因给订阅者发消息规则的变动而需要修改ArticleService，这本身违反了单一职责原则（SOLID原则都是互通的，违反其中一个往往也违反其他的）。可以在ArticleService::publish()中发布一个article-published事件，外部订阅该事件，这样可以在事件订阅端做任何业务扩展（对扩展开放）而不影响这里的类（对修改封闭）。  
     **L.里氏替换原则:**  
     一个抽象的任意一个实现，可以用在任何需要该抽象的地方。  
     有个IAnimal抽象定义了run()和sound()方法（发声），下面有实现类：Bird、Earthworm(蚯蚓)、Person，Bird和Person都对sound()做了各自的实现（发鸟声和人声），但Earthworm::sound()却throw了个异常。现在有个AnimalTrainer::train(IAnimal)这样的调用，想想会发生什么？当我们传入Earthworm对象时，其运行结果是未知的，有可能抛异常（如果AnimalTrainer调用了sound的话）。我们通过train(IAnimal)的声明无法知道它如何使用这个IAnimal，而根据里氏替换原则，Earthworm自然应该被AnimalTrainer正确使用（而不是抛异常），因而这里Earthworm的实现违背了里氏替换原则。如果实现类既要实现一个抽象，又不想去实现该抽象的某些契约（通过抛异常来抗议），说明你的抽象设计有问题。  
     **I.接口隔离原则：**  
     在实现接口时，不能强迫去实现没有用处的方法。  
     还是上面的例子，训练师去训练蚯蚓发声是枉然的。Earthworm::sound()是完全没有用处的，要么放着空函数什么都不做，要么抛异常，这里的设计就违背了接口隔离原则。  
     **D.依赖反转原则：**  
     该原则在前面已经单独讨论过（因为对六边形架构太重要了），此处不再赘述。  
       
     参见：<http://www.tuicool.com/articles/JBRBzqm> 《从百草园到三味书屋》，laravel作者著，里面对SOLID原则有很好的示例讲解。  
       
     下面举个综合例子：  
     客户购买商品，下单时，系统需要进行各项校验。  
     假设在OrderProccessor::confirm()中进行订单校验（该类维持一个对Order实例的引用）。  
     最开始只需要校验相关商品是否有足够库存，我们创建Order::validate()执行这些校验。由于validate所使用的数据大部分都可从Order对象得到，这符合信息专家模式。Ok。  
     某天，接到一个添加校验规则的需求：校验下单者是否符合下单规则（只有业主才能下单）。我们需要改Order::validate()方法。然后我们发现订单校验规则的变化会导致Order的修改，这违反了单一职责原则和开放封闭原则。于是我们决定将“订单校验”业务逻辑抽离出来形成OrderValidator类，由OrderValidator::validate(Order)实现校验,这样就隔离了校验规则改动对订单类的影响。  
     但是我们后来发现，一个validate()方法搞定所有校验，导致validate()这个方法过于臃肿，于是我们对校验规则分类后抽离出validateGoodsStock()、validateBuyer()等独立的方法，然后在validate()中调用这些方法。  
     过几天，又要加个校验规则：订单价格是否合法。于是我们又加个validatePrice()方法。虽然说OrderValidator隔离了校验对Order类的影响，但每加个规则就去改下该类，这违反了开放封闭原则。  
     有没有什么办法能够隔离校验业务的变动对OrderValidator的影响呢？  
     答案是抽象。  
     我们抽象出IOrderValidator接口，定义一个validate(Order)契约，然后创建GoodsStockValidator、BuyerValidator、PriceValidator等实现类实现该接口，在其validate(Order)中实现上述种种校验。然后给OrderProccesor注入一个包含IOrderValidator的集合，在confirm()中顺序调用每个验证器的validate方法。  
     现在，需要增加校验规则，没问题，创建一个新的IOrderValidator实现类并放到校验集合中即可——该设计对扩展开放（通过创建新的校验器类），对修改封闭（不需要修改其他的类）。

上面的几个设计模式都是非常基础非常通用的，是实施OO必须掌握的，它们共同的基础原则都是“高内聚低耦合”，进行OOD时必须时刻进行这些原则反思。

其他设计/架构模式：

1. **DDD（领域驱动设计）：**  
   领域驱动设计指出传统的需求分析和模型设计、代码编写是相互割裂的，传统有需求分析师和系统设计师两个独立的职位，这种割裂导致相互之间的不匹配，比如系统设计不能完全反映出需求分析，而代码又和系统设计割裂，各自有一套自己的私有语言，相互之间很难沟通。  
   DDD强调需求分析、建模和编码的内在统一性，三者（以及执行三者的人）之间使用一致的领域**通用语言**沟通，因而业务专家、设计师、程序员之间能够很容易达成共识。  
   现实情况是，程序员和业务专家（以产品经理为代表）之间的沟通要么存在严重的鸿沟，要么使用非业务（往往是技术性的）语言沟通，背离真正的业务领域概念。程序员很喜欢用技术性语言（甚至直接拿数据库说事）和别人（哪怕是非技术人员如客服、销售）沟通，导致各执一词。一般敏捷团队往往只有一个产品经理，而有好几个技术人员，往往会出现以技术性语言主导沟通的场面（如果产品经理本身不注重对团队的业务语言引导的话）。  
   程序员为何那么喜欢用技术性语言和别人沟通？一方面，程序员的沟通对象常常也是程序员，技术性语言沟通成本最低；另一方面，他们往往在沟通的同时就在想着实现方案（或者说沟通本身就是对实现方案的描述）。  
   然而，技术语言沟通对业务模型的建立有着很严重的损害。技术本身和业务是两个领域的东西，技术语言在现实中最典型的代表就是“数据库语言”，比如“某个时候将某表的某字段标记为1”，这于业务本身无任何意义。这种思维导向会让我们脑海中越过建模而直达存储实现低层。另一方面，这种技术与业务语言的混杂会让业务逻辑本身耦合进存储层的设计中。例如，单从存储设计（技术实现）上来说，“登录状态”应当由单独的字段来标记，而在业务领域中，“登录”与“退出登录”操作会导致另外的状态变化（存储设计上表现为另一个字段），当我们在进行存储层设计时过多的代入业务逻辑本身（或者毋宁说在业务逻辑描述时过多地代入存储层的技术实现），我们可能会用另一个字段（存储着其他的状态）来代表登录状态（并且很自豪地认为这样能节约存储空间——典型的技术主导一切的思维）。这里的问题是：登录状态的存储实现依赖于业务逻辑，由于业务逻辑是不稳定的（相对于存储层），因而这里作为最底层的存储层设计也是不稳定的。（实际上会员的登录状态存储就存在这样的问题）  
   DDD强调：
   * 1. 从需求分析到代码实现到测试的整个过程各人员之间的沟通需要使用一致的无歧义的领域内通用语言。
     2. 该通用语言必须能够准确反映业务需求和领域知识（而不是反映技术实现）。

对于程序员来说，DDD的这种思想可概括为：代码即模型，编码即设计。我们写出来的代码，类与类之间的调用关系，方法、变量的命名都要反映领域通用语言本身。DDD非常强调命名，对于DDD来说，编程本身就是语言活动（不是机器语言）。DDD强调语言的重要性，这语言是人类的语言，而且是某特定领域下的人类语言。  
现实情况是，我们的代码中充斥着大量的面向数据库的语言。例如update()、delete()充斥在各种业务代码中。“将字段A的值更新为b”是技术（数据库）语言，不是业务领域语言。更有甚，在控制器层写个UpdateFields()搞定一切更新操作。  
然而，并不是所有的技术实现都要以当前业务领域语言来诠释，实际上这也是做不到的。有些技术实现并非属于业务领域之内，例如持久化存储、事件系统、消息队列等，这些不属于当前业务领域的技术实现当然也就不需要遵守该业务领域通用语言（它们需要遵守的是各自的领域语言规约）。这种业务领域边界在DDD中叫“限界上下文”，上下文内的东西属于六边形架构中的内圆部分，而外部的东西属于外圆，内外圆通过契约适配通信（适配器）。  
DDD并非泛泛的理论阐述，它有一套详细的实现体系（方法论），如实体、值对象、聚合、聚合根、领域服务、仓储、各种设计模式等，此处不做详细阐述（那得写成一本厚厚的书，而且DDD本身是很注重实践的，一百个人就有一百种实现方式，重在掌握其核心思想）。  
这里提出DDD，重点在于对比我们现在使用的开发/设计模式：面向服务设计以及面向数据库设计。在DDD中，实体是核心，服务只是辅助，而数据库则是领域外的基础设施。

b) **CQRS(命令与查询职责分离)**

命令：会导致实体状态变化的操作（反映在数据库上的更新、删除、插入等）；

查询：不会导致实体状态变化的操作。

CQRS原则：命令中不要有查询，查询中不要有命令。例如常做的在修改方法中同时查询并返回某记录，这就是违反CQRS的。

CQRS的目的是为了应对这样一个事实：命令模型与查询模型往往存在很大的不同，想想我们的数据库设计就很好理解。我们进行数据库设计时，一般是按照命令模型进行设计的，这和我们脑海中的业务模型比较匹配，而报表则是典型的查询模型（分析模型），一般情况下，按照命令模型设计的数据表结构是满足不了查询模型的报表分析的，因而，为了出报表，要么需要写很复杂的sql，要么进行数据加工清洗以得出符合条件的查询模型。显然，在命令模型上执行分析查询性能是非常低下的。  
可以在不同层面上使用CQRS:

1. 传统意义的读写分离。命令和查询使用不同的库，但两个库中的表结构相同。

2.代码层面分离，存储层不分离。（有可能采用读写分离，但表结构是一样的）

3.代码和存储层都分离。这也是严格意义上的CQRS。这里写表是基于命令模型设计的，读表是基于查 询模型设计的，读和写是通过事件来同步的（命令端执行完毕后发布相应事件，查询端订阅事件并更新 查询模型）。在代码和存储层进行命令与查询分离，在两端各自采用最适合的实现方式，以达到最优设 计和最好的性能。  
严格意义上的CQRS实现起来很复杂，要求基础支撑够健壮才行。  
我们这里提出CQRS，一方面是为了指出以上事实，另一方面，在实践中我们可以尝试第二层面的CQRS，以获得代码层面带来的益处，如缓存管理、两端可采用不同的设计模式（如命令端采用DDD，查询端采用传统的MVC）。

**三、现状分析：**

1. 关于Service（Service的含义以及面向服务编程有何问题）：  
   当我们说“服务”时，我们强调的是其**为外在它方提供功能。**服务本身具有动词性，核心是其功能性，该功能当然由服务提供者提供，但我们并不关心服务提供者本身。服务具有**外向性**，即服务存在的价值取决于其对他方（而非自身）的价值。因而，当我们以服务为核心时，我们就必然会以外在**旁观者的角度**审视其外在价值。当我们说“消息队列服务”，我们看重的是消息队列给我们业务系统带来异步解耦的好处，而不是消息队列本身的内部机制。  
   这种看待问题的角度对建模是无益的。当提及服务时，总是有个“我”存在（就是那个旁观者），而建模强调的是达到“无我”的境界，需要消除这个旁观者，化身为模型本身，从事物内在角度去思考问题。模型是内在自满足的，它本身具有特定的行为，而不是靠各种服务“提供”给它。  
   举个现实的例子。软件外包公司对外提供软件外包服务。当我们作为甲方（也就是旁观者）接受其提供的服务时，只需要告诉外包公司我们有什么样的需求，双方达成共识，签订合同，最终我们从外包公司那里接收符合合同预期的软件成品，我们并不关心外包公司内部具体怎样运作，哪个团队负责ui设计，哪个负责编程等。但是，对于外包公司本身来说，这种视角是行不通的，它必须设计一套详细的、健全的公司运作体系以保证其可以提供该服务，也就是说必须对“外包公司”这个概念进行内在建模，以形成一个自运作的实体。  
   面向服务编程有什么问题呢？不是一样可以对服务内部建模吗？  
   问题是当我们面向服务，以服务为核心载体时，往往做不到对其内部很好地建模。根本原因在于面向服务的外向型思维和面向实体的内向型思维是冲突的。我们建模时，只会有一种主导思维，其它只起辅助作用。当我们以服务为主导时，首先想的是**功能**，是做什么的问题；当我们以实体为主导时，首先想的是谁来做，是**主体**问题。一种是由功能推导出功能提供者，此处提供者是辅体，功能是主体；一种是由主体推导出主体行为，行为是主体的自在要素。  
   由于在面向服务编程中，提供者只是辅体，往往容易被忽略，我们并不是太在意谁提供功能，结果就是往往一个类提供了n多个功能，比如一个OrderService提供和订单相关的一切功能。久而久之，Service（特别是业务中的核心Service如商城的OrderService和GoodsService）会变得异常臃肿，难以维护。混乱的职责分配还会导致业务逻辑泄露，不同的开发者可能会选择由不同的提供者提供相同或相似的功能，导致一份业务逻辑在多处出现。
2. 面向数据库编程：  
   当我们接收到业务需求，大致分析完毕后，首先想要做的事情是什么？  
   表结构设计。  
   是的，这是我们大部分人一贯的行事风格，甚至一个程序员技术水平高低全看他表结构设计的好坏。  
   但是我们冷静地想想，数据库对于业务到底意味着什么？它不过是数据的一种持久化方式而已，属于基础支撑层的东西。  
   面向数据库编程的问题是，我们从一开始就钻到最底层的细枝末节上，而不是以居高临下的角度设计和审视业务系统内在的、本质的逻辑规则。这种设计方式，由于没能深入地设计建模，很容易就事论事，往往迷失在细节森林中。另外，由于数据库设计和业务建模是同时进行的，表设计中往往会带进过多的当前业务逻辑。由于业务规则会随着不同时期需求变化而变化，因而这种表结构是不稳定的。典型的体现是一个字段表示多方面的状态值，这些状态之间遵守（目前的）固定的业务规则。数据库设计不可能完全脱离具体的业务，但一定要尽量识别并减少不稳定业务规则的摄入。  
   面向数据库编程思维是自始至终的，而不仅仅在初期表设计阶段。编程过程中，我们习惯将增删改查字段这种数据库术语带进业务代码中，操作也是直接面向数据库进行的。  
   这种编程方式中，往往没有实体，就算有，也是仅仅作为数据传输对象（DTO）使用，这些对象往往没有方法，其属性基本和数据库表一一对应。模型(Model)也沦为数据库表的对象化表述，自带CRUD。  
   面向数据库编程之所以那么流行是因为简单，会敲代码就能做，完全不用考虑对象建模，而且其弊端在简单的业务复杂度面前并不会暴露。然而当业务复杂度达到一定规模后，其弊端是致命的。大量的面条式代码牵一发而动全身，业务逻辑零散各地。需求迭代越频繁，这些弊端就越早暴露，前期的大步流星扯得后面蛋疼不已。
3. 关于Repository（仓储层职责以及事务该由谁负责）:  
   从分层上来说，仓储处于业务边界处，连接业务层和存储层（基础设施层）。仓储是懂领域语言的（但不代表要在里面实现具体的业务逻辑），另一方面它也懂持久化相关工作。在《实现领域驱动设计》一书中建议将仓储当做集合使用，并将其定义（接口）放在领域层，将实现（实现类）放在基础设施层。  
   我们先看看集合。集合里面装着一组同质（同类）元素（对象），拥有添加（add）、移除（remove）。注意，集合并没有modify操作，因为我们获取的是集合中对象的引用，当对象状态发生变化时，集合中的那个对象是同步变化的。
4. 关于Controller（传输层职责以及不该做什么）:
5. 关于Entity（为何实体被忽略）:
6. 关于framework/sdk（康威定律如何体现在framework的内部矛盾中）:
7. 关于单元测试（为何单测总是不了了之）：
8. 关于内部系统间调用（内部就安全了？）：
9. 关于与外部系统的通信（内外接口的区别与联系以及如何统筹设计接口层）：  
   9.关于登录（推还是拉？）：
10. 关于消息队列（健壮性到底有多重要）：
11. 关于数据同步（双边业务不匹配问题）：
12. 关于Module：
13. 关于缓存：
14. 关于数据库设计：
15. 关于建模：