目录

- 01.DDD设计思想
- 02.DDD四层架构设计
- 03.DDD通用型领域

01.DDD设计思想

- **技术主动理解业务**
  - Controller、Service等概念并不具备实际的业务价值
  - 如果新的设计对已有的架构造成太大的冲突，就只能考虑重构。而重构对于业务专家来说，没有什么业务价值
  - 鼓励形成团队内部的通用语言，业务如何组织，软件就如何构建
  - 让领域专家看到这样的组件，即便不了解实现细节，也能够知道这些组件是干什么的，从而有可能继续深入的参与软件建设过程
- **“刚刚好”解决问题：**
  - 往往项目未来的需求变化并不是技术人员能够把控的。如果未来的业务需求不是按照之前设计的路线演进，这些“提前设计”不但无法起到设计时的作用，反而会增加软件项目的复杂度
  - 同时为软件保持足够的灵活性，比如：如果抽象出了Repository仓库层，不管多复杂的持久化逻辑，都只是一个仓库实现而已，不会对业务造成任何影响。这些需求就可以比较独立的进行改造

02.DDD四层架构设计

一个需求非常确定的项目，不管多复杂，都没有必要非转型成DDD，短平快的设计方式更为快捷。**如果项目中有很多不确定性，以往的设计模式会遇到非常多的变数**，这时DDD就是一个很好的选项了。开发业务时，可以优先开发领域层的业务逻辑，然后再写应用层的服务编排，而具体的外部依赖与具体实现可以最后再完成。整体就形成了一种以**领域优先**的架构形式。

DDD四层架构：

1. **用户接口层**：对外提供服务
2. **领域层**：最底层不再是数据库，而是实体(Entity)，值对象(ValueObject)，领域服务(Domain Service)。这些对象都不依赖任何外部服务和框架。这些对象可以打包成一个领域层(Domain Layer)。**领域层没有任何外部依赖关系。**
   - 使用值对象：具**有业务意义的对象**（比如做参数校验，避免类似逻辑分散出去，污染系统代码）
   - 使用**充血模型**改造实体类，保留改变状态的方法
   - 使用**领域服务封装实体逻辑**（跨领域对象的行为）
3. **应用层**：原有的Service层，经过重新编排后，只依赖于一些抽象出来的防腐层(ACL)和仓库工厂(Repository)。他们的具体实现都是通过依赖注入进来的。他们一起负责整个组件的编排，这样就可以把他们打包成一个应用层(Application Layer)。**应用层依赖领域层，但是不依赖具体实现。**
   - 抽象**数据持久层** - 建立仓库（让业务逻辑不再需要面向数据库编程，而是面向实体编程）
   - ACL **防腐层** - 抽象第三方服务（防止第三方服务状态不可控，入参出参强耦合的问题）
4. **基础设施层**：最后是一些与外部依赖打交道的组件。这些组件的实现通常依赖外部具体的技术实现和框架，可以统称为基础设施层(Infrastructure Layer)。

传统事务脚本式编码的问题：

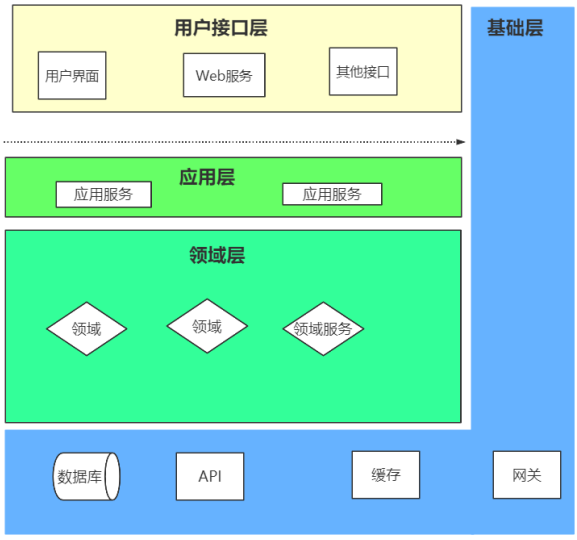
1. 代码层面的软件老化风险（代码不断膨胀）：数据来源被固定、业务逻辑无法复用、业务逻辑与数据存储相互依赖
2. 架构层面的软件老化风险（不敢升级，不敢写新功能）：数据结构变更、JDBC依赖调整、第三方服务变更、MQ中间件调整
3. 随之而来的实施及测试问题：设施搭建困难（依赖的外部组件太多）、测试用例覆盖困难

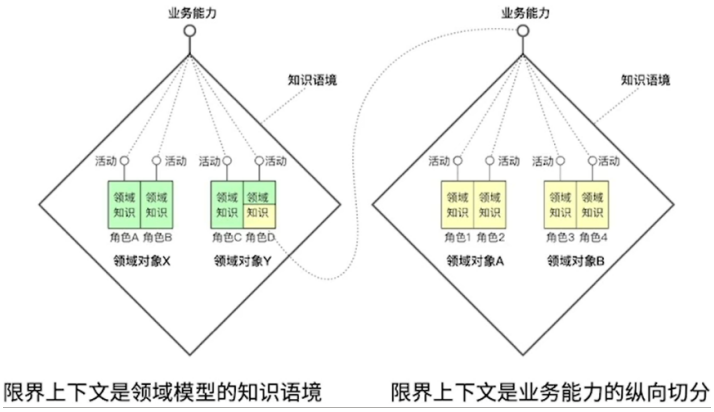
DDD领域划分设计：高内聚，低耦合：单一职责原则、依赖反转原则、开放封闭原则

1. 构建**领域地图**（边界）
   - 在DDD中推荐了**事件风暴会议**这样的具体形式，也强调了统一语言的理论模型。
   - 针对各个核心环节，优先构建单元测试案例，从而形成一些**TDD测试驱动**设计的实践场景。
   - 领域并不是天然存在的，这需要依赖于软件团队对软件需求的合理分析。并且分析的过程，**即不能太过偏向于真实的业务领域，也不能太过偏向于极度抽象的技术领域，需要在整个项目团队内形成共识**，然后通过初步抽象的通用语言将设计结果固化下来。
2. 使用**四层架构**巩固领域基础（包结构）

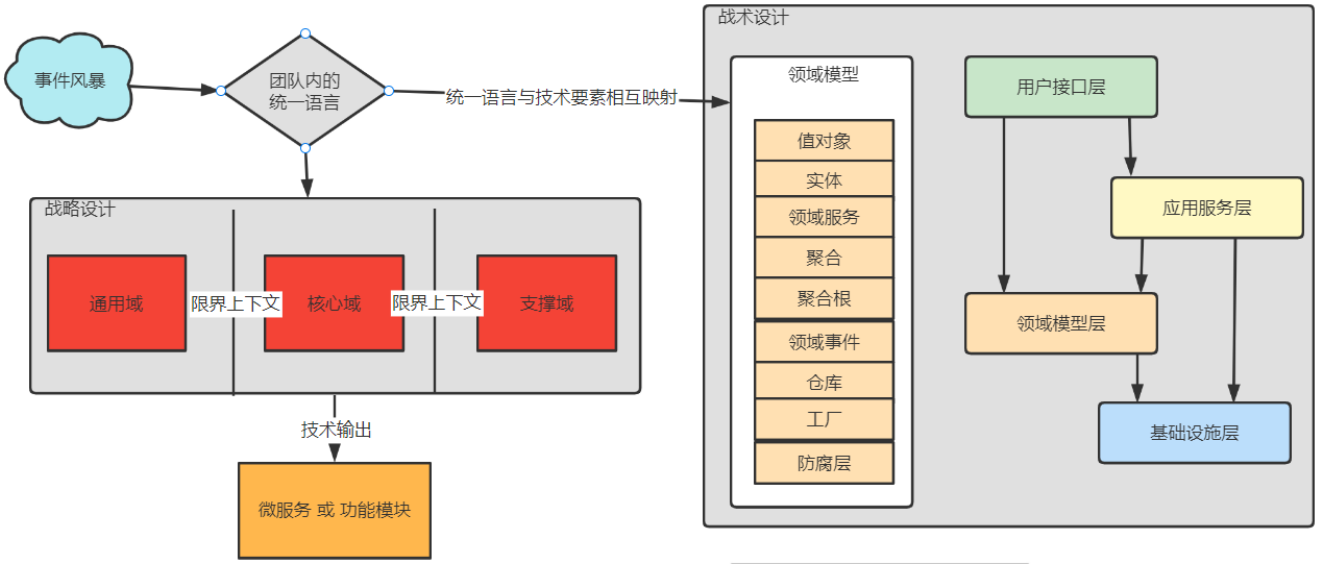
```
application
domain
   xxx
     adaptor
          interface
     entity
     infrastructure
     repository
     service
```

3. 划分**限界上下文**，巩固领域划分（DDD落地实践的关键）
   - 传统MVC设计方式中，强调的是技术隔离，而并没有从业务上的限界上下文边界，所以，逻辑边界是混乱的。
   - 领域应该是足够**内敛**的，每个领域内的业务能力是与当前领域的知识语境紧密相连的。一个领域想要调用另外一个领域的业务能力，只能通过对方暴露出来的业务能力进行协作，而不能去干预对方领域的实现细节。所谓的单体架构、微服务架构、甚至包括MQ事件驱动架构，在DDD的视野中，都是领域之间的不同协作方式而已，对领域内部都是没有影响的。
4. **单体架构优先**（SPI）
   - 越是复杂的项目，就越应该重视单体架构的快速验证和快速试错，这对于提高复杂项目的运行效率是非常重要的，然后将单体架构中的关键领域拆分成微服务（比如SPI切换为Nacos，由于有防腐层，所以只需要切换抽象接口的实现类即可）





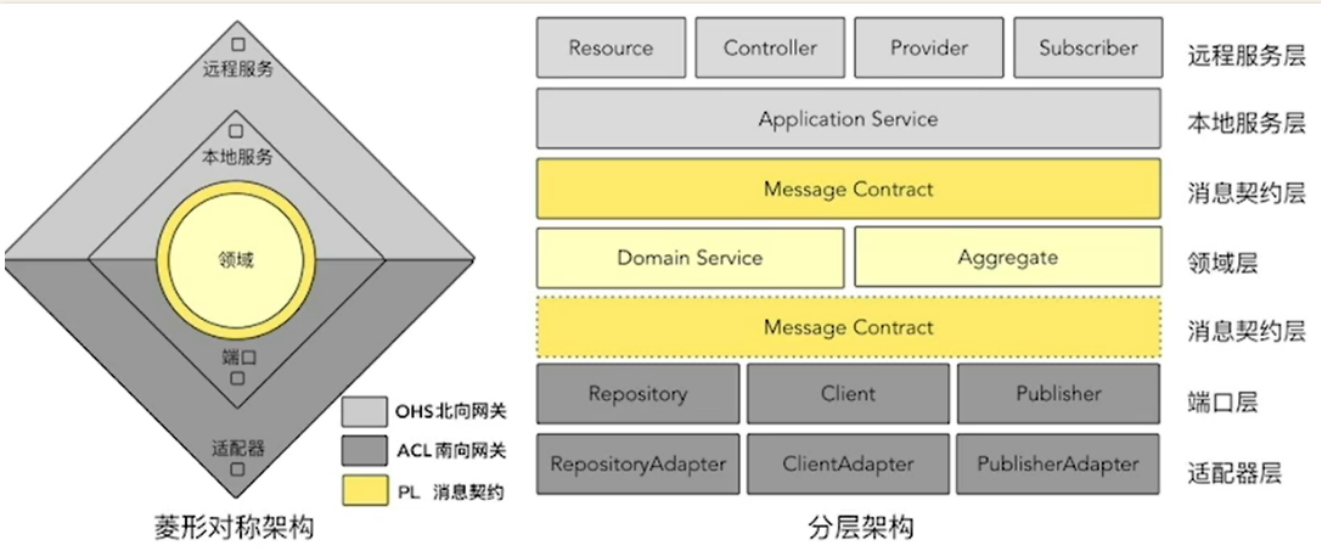
限界上下文是领域模型的知识语境　　限界上下文是业务能力的纵向切分



03.DDD通用型领域

可以构建领域仓库，实现领域复用：也可以用于优化中台战略，并且取缔大中台（中台拆分），因为**一个中台组不可能同时满足所有其他项目组的需求**

DDD通用领域模型



通过消息契约层，不光保证逻辑边界清晰，同时也保证数据边界也很清晰