DDD 领域驱动设计

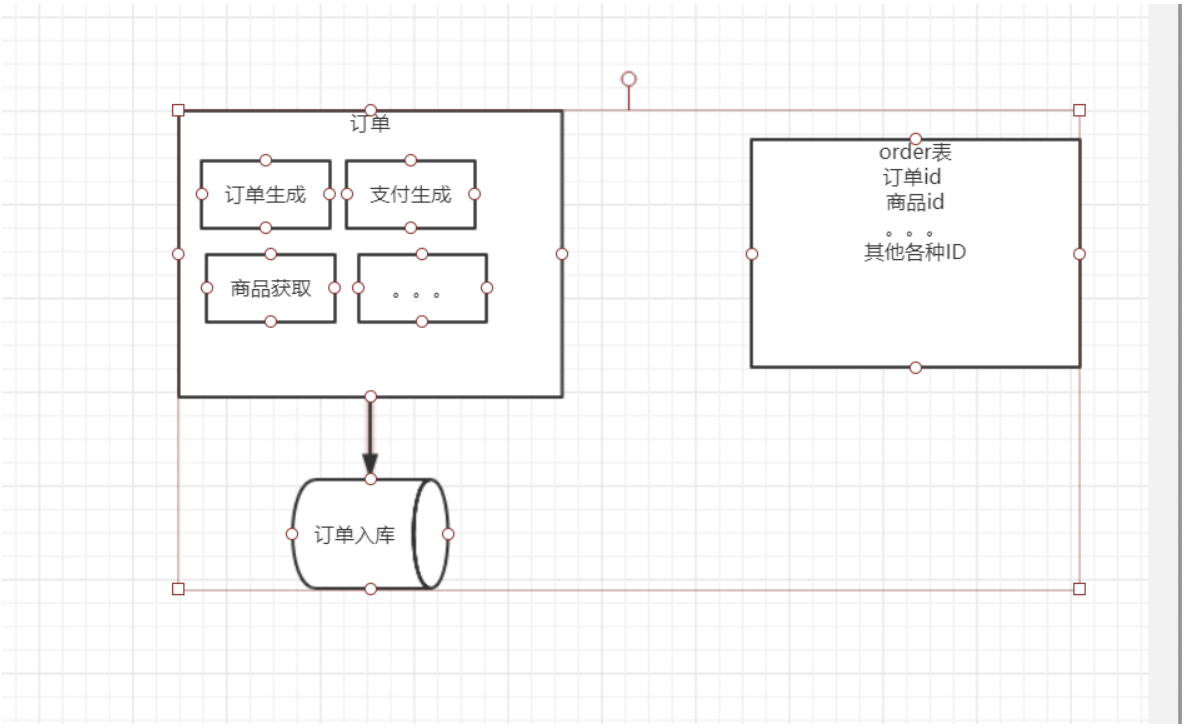
1.当我们要开始一个项目的时候，我们第一步要去怎么做?

领域驱动设计（Domain-Driven Design，简称DDD）

业务初期，我们的功能大都非常简单，普通的CRUD就能满足，此时系统是清晰的。随着迭代的不断演化，业务逻辑变得越来越复杂，我们的系统也越来越冗杂。模块彼此关联，谁都很难说清模块的具体功能意图是啥。修改一个功能时，往往光回溯该功能需要的修改点就需要很长时间，更别提修改带来的不可预知的影响面。

高度耦合的代码：

在一个服务中提供不同领域的接口:



我相信绝大多数同学都会遇到过这个问题，在刚开始开发的时候，我们是这么设计的，随着我们的业务不断发展，我们的项目不断扩大，同时我们的表也是一个订单大表，包含了非常多字段。在我们维护代码时，牵一发而动全身，很可能只是想改下商品的功能，却影响到了创单核心路径。虽然我们可以通过测试保证功能完备性，但当我们在订单领域有大量需求同时并行开发时，改动重叠、恶性循环、疲于奔命修改各种问题。

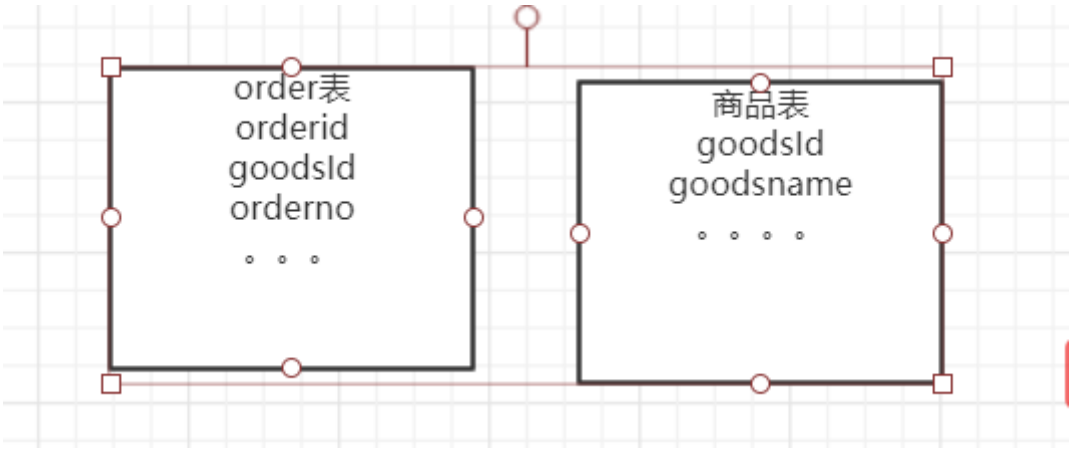
上述问题，归根到底在于系统架构不清晰，划分出来的模块内聚度低、高耦合。

事实上，30年以前，一些软件设计人员就已经意识到领域建模和设计的重要性，并形成一种思潮，Eric Evans将其定义为领域驱动设计（Domain-Driven Design，简称DDD）

2.DDD 是什么

领域驱动模型,什么叫做领域驱动？我们可以举一个例子。

比如我们上面的订单商品模块,假设我们数据库是如此设计：



如果我们跟随数据库去设计我们会这么干：

```
class goods{
    String id;//主键
    String skuId;//唯一识别号
    String goodsName;
    Bigdecimal price;
    Category category;//分类
    List<Specification> specifications;//规格
    ...
}

class Order{
    String id;//主键
    String orderNo;//订单号
    List<OrderItem> orderItems;//订单明细
    BigDecimal orderAmount;//总金额
    ...
}

class OrderItem{
    String id;
    Goods goods;//关联商品
    BigDecimal snapshotPrice;//下单时的价格
}
```

看似好像没问题，考虑到了订单要保存下单时候的价格（当然，这是常识）但这么设计却存在诸多的问题。在分布式系统中，商品和订单这两个模块必然不在同一个模块，也就意味着不在同一个网段中。上述的类设计中直接将Product的列表存储到了Order中，也就是一对多的外键关联。这会导致，每次访问订单的商品列表，都需要发起n次远程调用。

反思我们的设计，其实我们发现，订单BC的Product和商品BC的Product其实并不是同一个entity，在商品模块中，我们更关注商品的规格，种类，实时价格，这最直接地反映了我们想要买什么的欲望。而当生成订单后，我们只关心这个商品买的时候价格是多少，不会关心这个商品之后的价格变动，还有他的名称，仅仅是方便我们在订单的商品列表中定位这个商品。

> 重点是，我们的领域设计思路需要去脱离数据库的桎梏，最高的预期是根据我们界限去完成数据库设计，最次。。不需要数据库来绑架我们的系统设计。业务才是王道，一个架构师的核心价值不仅仅体现在框架的应用上，最关键在于能够将我们的系统设计安排得明明白白。

如何改造?

```
class OrderItem{
    String id;
    String productId;//只记录一个id用于必要的时候发起command操作
    String skuId;
    String productName;
    ...
    BigDecimal snapshotPrice;//下单时的价格
}
```

是的，我们做了一定的冗余，这使得即使商品模块的商品，名称发生了微调，也不会被订单模块知晓。这么做也有它的业务含义，用户会声称：我买的时候他的确就叫这个名字。记录productId和skuId的用意不是为了查询操作，而是方便申请售后一类的命令操作（command）。

在这个例子中，Order 和 goods都是entity，而OrderItem则是value object（想想之前的定义，OrderItem作为一个类，的确是描述了Order这个entity的一个属性集合）。关于标识，我的理解是有两层含义，第一个是作为数据本身存储于数据库，主键id是一个标识，第二是作为领域对象本身，orderNo是一个标识，对于人而言，身份证是一个标识。而OrderItem中的productId，id不能称之为标识，因为整个OrderItem对象是依托于Order存在的，Order不存在，则OrderItem没有意义。

领域模型

在Martin Fowler理论中，有四种领域模型：

在Martin Fowler理论中，有四种领域模型：

1. **失血模型**
2. **贫血模型**
3. **充血模型**
4. **胀血模型**

我们以修改商品为例来举例模型的概念

```
class goods{
    String id;
    String skuId;//唯一识别号
    String goodsName;
}
```

失血模型：略过，可以理解为所有的操作都是直接操作数据库。

贫血模型：

```
class GoodsDao {
    @Autowired
    JdbcTemplate jdbcTemplate;

    public void updateName(String name,String id){
```

```
        jdbcTemplate.excute("update goods u set u.goods_name = ? where
id=?",name,id);
    }
}

class UserService{

    @Autowired
    UserDao userDao;

    void updateName(String name,String id){
        userDao.updateName(goodsName,id);
    }
}
```

贫血模型中，dao是一类sql的集合，在项目中的表现就是写了一堆sql脚本，与之对应的service层，则是作为Transaction Script的入口。观察仔细的话，会发现整个过程中user对象都没出现过。

充血模型

```
interface UserRepository extends JpaRepository<Goods,String>{
    //springdata-jpa自动扩展出save findOne findAll方法
}

class UserService{
    @Autowoird
    UserRepository userRepository;

    void updateName(String name,String id){
        Goods goods = goodsRepository.findOne(id);
        goods.setName(name);
        goodsRepository.save(user);
    }
}
```

充血模型中，整个修改操作是“隐性”的，对内存中goods对象的修改直接影响到了数据库最终的结果，不需要关心数据库操作，只需要关注领域对象goods本身。Repository模式就是在于此，屏蔽了数据库的实现。与贫血模型中goods对象恰恰相反，整个流程没有出现sql语句。

涨血模型：

没有具体的实现，可以这么理解：

```
void updateName(String name,String id){
    Goods goods = new Goods(id);
    goods.setName(name);
    goods.save();
}
```

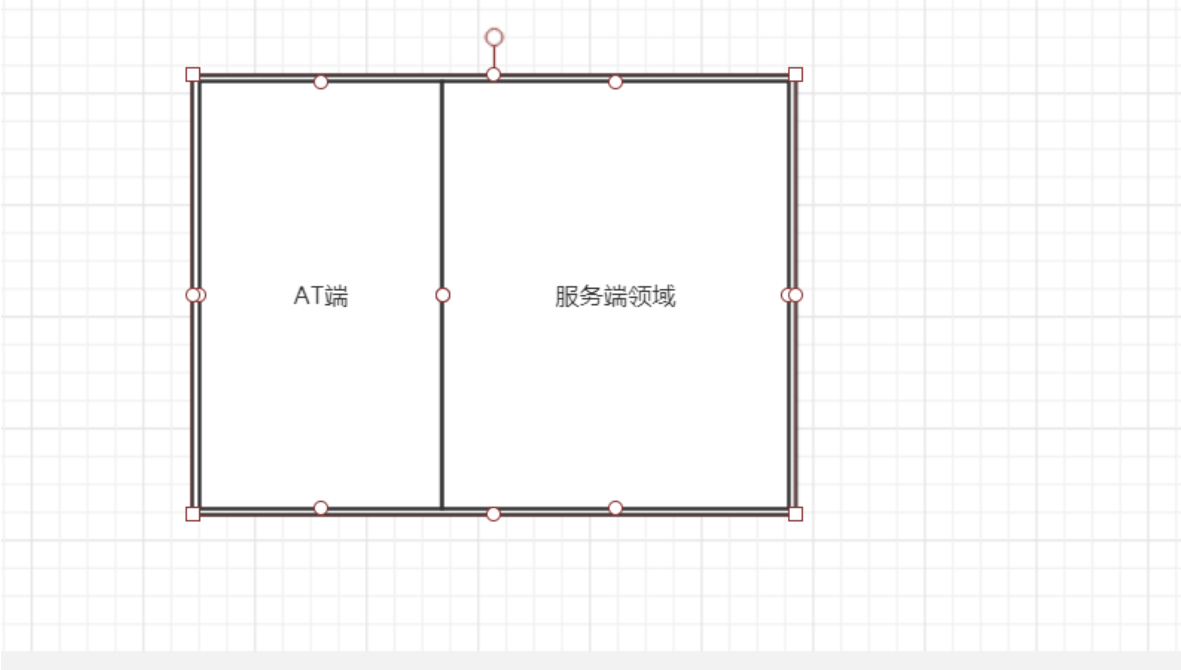
我们在Repository模式中重点关注充血模型。

3.微服务架构中的DDD应用

在微服务架构中，我们提倡的是低耦合，高内聚，那么需要达到低耦合高内聚这个目标，我们需要去如何应用DDD领域的概念去完成呢?

在DDD领域中，提供了我们一个非常有意思的东西，叫做界限上下文.界限上下文是怎么来的，我们肯定需要知道，我们要理解一个领域的概念。

举例，我们有一个业务，即我们需要售货机为AT端，需要java服务来作为服务支撑，那么我们就可以将一整个项目做到这样的一个结果：



以服务端而言，我们需要来界定领域，这时候我们需要来对需求文档进行分析：

1. 根据需求划分出初步的领域和限界上下文，以及上下文之间的关系；
2. 进一步分析每个上下文内部，识别出哪些是实体，哪些是值对象；
3. 对实体、值对象进行关联和聚合，划分出聚合的范畴和聚合根；
4. 为聚合根设计仓储，并思考实体或值对象的创建方式；
5. 在工程中实践领域模型，并在实践中检验模型的合理性，倒推模型中不足的地方并重构。

领域

现实世界中，领域包含了问题域和解系统。一般认为软件是对现实世界的部分模拟。在DDD中，解系统可以映射为一个个限界上下文，限界上下文就是软件对于问题域的一个特定的、有限的解决方案。

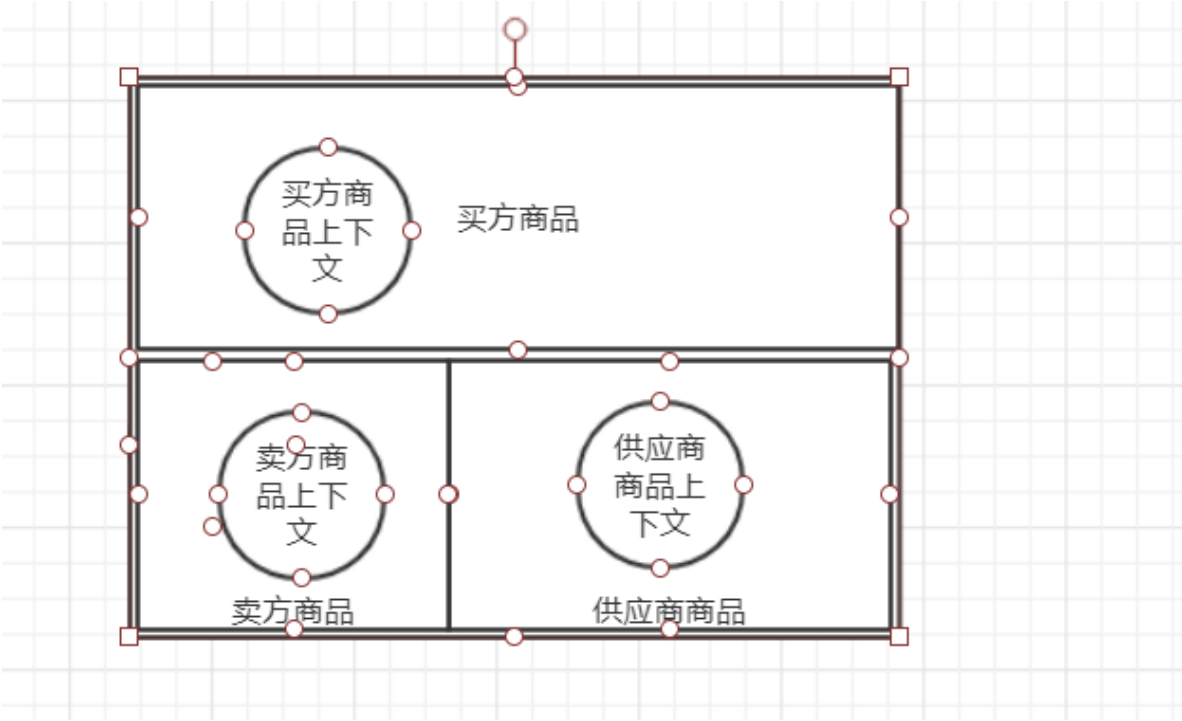
那么我们的服务端假设只有两个领域--一个是订单，一个是商品，那么我们可以把商品领域进行进一步的细分：

假设商品需求如下（事实上这个和用户角色进行了挂钩，我们先不用去太在意这个，先来理解下领域）：

买方商品--可见购买商品，购买者可以看到所有商品（进行价格排序）。

卖方商品--可以去上架商品，上架成功之后购买者就能够看到商品。

供应商商品--可以给销售者提供商品，销售者的商品需要在销售列表中可被选择。



在每一个边界就形成了界限上下文。

在进行上下文划分之后，我们还需要进一步梳理上下文之间的关系。

> **康威（梅尔·康威）定律**
> 任何组织在设计一套系统（广义概念上的系统）时，所交付的设计方案在结构上都与该组织的沟通结构保持一致。

康威定律告诉我们，系统结构应尽量的与组织结构保持一致。这里，我们认为团队结构（无论是内部组织还是团队间组织）就是组织结构，限界上下文就是系统的业务结构。因此，团队结构应该和限界上下文保持一致。

拓展：墨菲定律--每当你觉得可能会发生的时候，这件事一定会发生。

通过我们界限上下文的划分，我们可以开始对商品服务内部进行处理：

```
import com.dn.goods.bussiness.buyer.*;//买方上下文
import com.dn.goods.bussiness.seller.*;//卖方上下文
import com.dn.goods.bussiness.supplier.*;//供应商上下文
```

将我们的界限全部按照这种方式进行划分（粗糙的示例，期望同学们根据这个思路衍生出更合理更精细的想法）。

> 我们可以在每一个上下文中去进行进一步的划分。划分的思路可以通过各种关系来进行划分处理。通常来说，我们就利用MVC思想为主体，然后再对我们上下文内部的各种业务进行划分。
>
> 上下文内部，我们可以根据一些划分好的模块进行进一步的设计，但是我们会忽然发现，到了这一步之后会出现一些问题--如：我们的各种上下文会开始出现耦合现象--比如，当供应商提交了一个商品之后，我们的卖方如何去获取？--直接通过访问资源的方式去加载吗？

解耦--以事件为驱动

我们在之前学习过了事件机制，对于我们的服务内部的设计以及其他的方式，我们都可以采用事件的机制来进行上下文处理：当我们供应商进行了货物补充，我们可以向我们的卖方提供一个事件，一旦卖方接收到事件之后再去进行相应的处理--这样无论我们针对于供应商进行各种修改，也绝不会影响到卖方整体逻辑。

在本文中，我们采用了分治的思想，从抽象到具体阐述了DDD在互联网真实业务系统中的实践。通过领域驱动设计这个强大的武器，我们将系统解构的更加合理。

但值得注意的是，如果你面临的系统很简单或者做一些SmartUI之类，那么你不一定需要DDD。尽管本文对贫血模型、演进式设计提出了些许看法，但它们在特定范围和具体场景下会更高效。读者需要针对自己的实际情况，做一定取舍，适合自己的才是最好的。

本篇通过DDD来讲述软件设计的术与器，本质是为了高内聚低耦合，紧靠本质，按自己的理解和团队情况来实践DDD即可。

另外，关于DDD在迭代过程中模型腐化的相关问题，本文中没有提及，将在后续的文章中论述，敬请期待。

备注：

> **限界上下文之间的映射关系**

- 合作关系（Partnership）：两个上下文紧密合作的关系，一荣俱荣，一损俱损。
- 共享内核（Shared Kernel）：两个上下文依赖部分共享的模型。
- 客户方-供应方开发（Customer-Supplier Development）：上下文之间有组织的上下游依赖。
- 遵奉者（Conformist）：下游上下文只能盲目依赖上游上下文。
- 防腐层（Anticorruption Layer）：一个上下文通过一些适配和转换与另一个上下文交互。
- 开放主机服务（Open Host Service）：定义一种协议来让其他上下文来对本上下文进行访问。
- 发布语言（Published Language）：通常与OHS一起使用，用于定义开放主机的协议。
- 大泥球（Big Ball of Mud）：混杂在一起的上下文关系，边界不清晰。
- 另谋他路（SeparateWay）：两个完全没有任何联系的上下文。