DDD实践中的那些坑

微信支付　王立

DDD CHINA

概述

DDD实践过程有很多坑，而且入坑还居然都是“不知不觉”，那才够坑！今天，我们一起用照妖镜来看看有哪些最容易入坑的陷阱吧！

没踩过坑吗？先自测一下.....

DDD的核心是领域模型吗？

应该全面采用DDD战术建模吗？

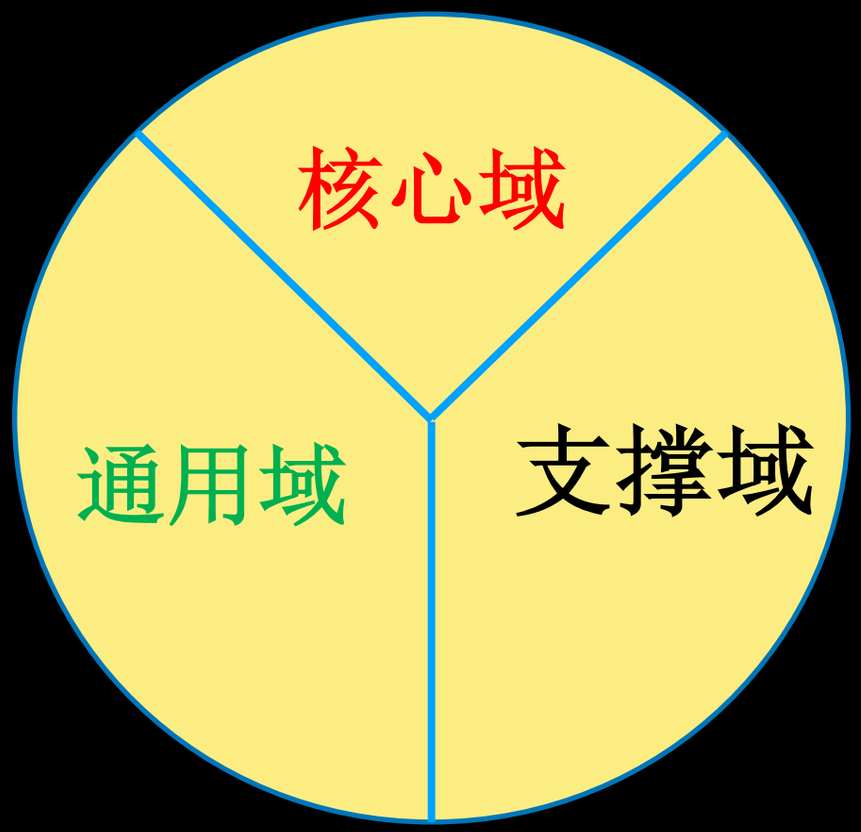
具有生命周期绑定的主从关系对象就是聚合吗？

内容大纲

- 战略的坑
- 分析的坑
- 限界上下文的坑
- 聚合的坑
- 仓储的坑

战略的坑

无差别地全面采用DDD战术建模



- **DDD**战术建模适用于重要或复杂的业务，初期投入成本较高；
- 只有核心域必须采用**DDD**战术建模，其它部分在必要时可以采用表驱动的**CRUD**模式。

没有基于竞争优势定义核心域

自助游网站——旅游攻略

网上商城——交易

网上订餐——点菜

简单理解核心域是主体业务就够了吗？

案例：京东VS天猫

领域类型	问题域
核心域	交易
	商品
	物流
	库存
支撑域	买家
	渠道
	订单
	支付
	营销
通用域	安全

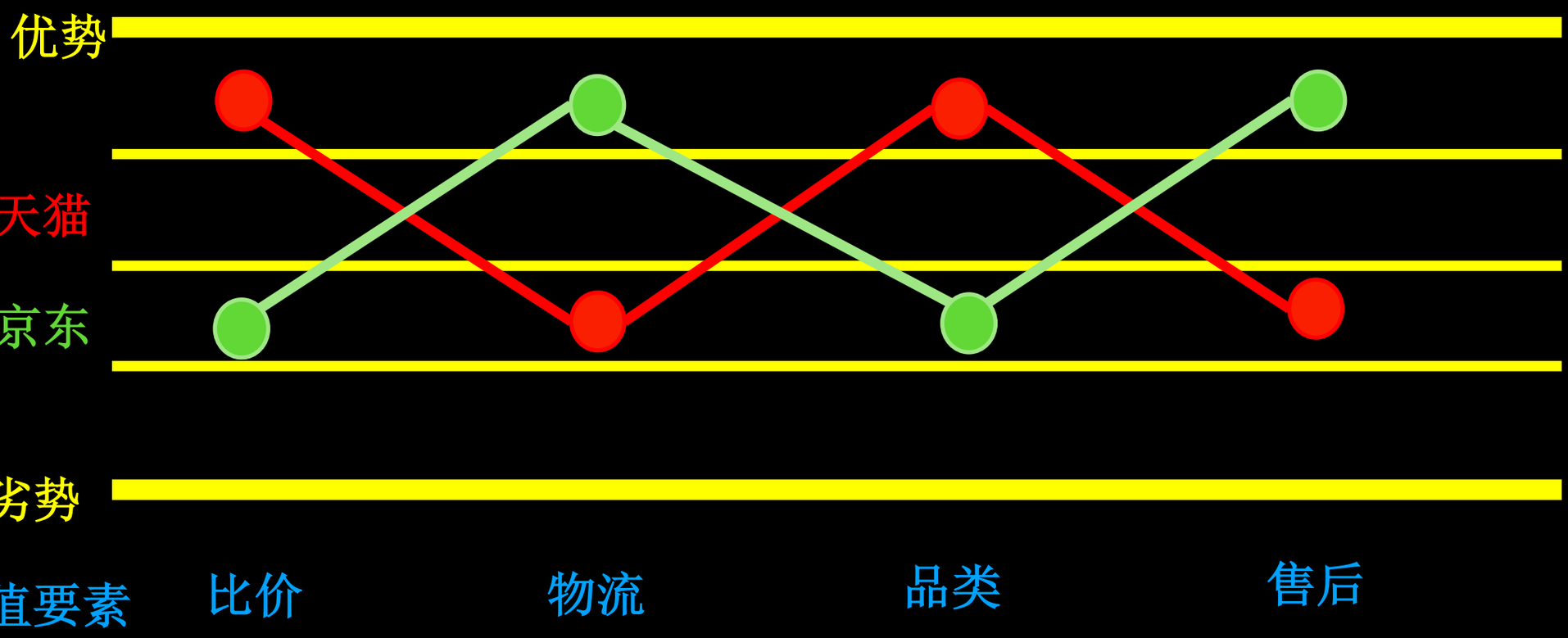
领域类型	问题域
核心域	交易
	商品
	营销
支撑域	买家
	卖家
	订单
	支付
	物流
	库存
通用域	安全

局外人非专业分析，但也无妨

案例：京东VS天猫

天猫定位：致力于为中高端用户提供高品质购物体验的B2C综合购物APP。

京东定位：至始至终，专业的综合购物商城APP。



同一个人在各方面的欲望是不同的，要创造性地细分人的欲望，而不是细分人群

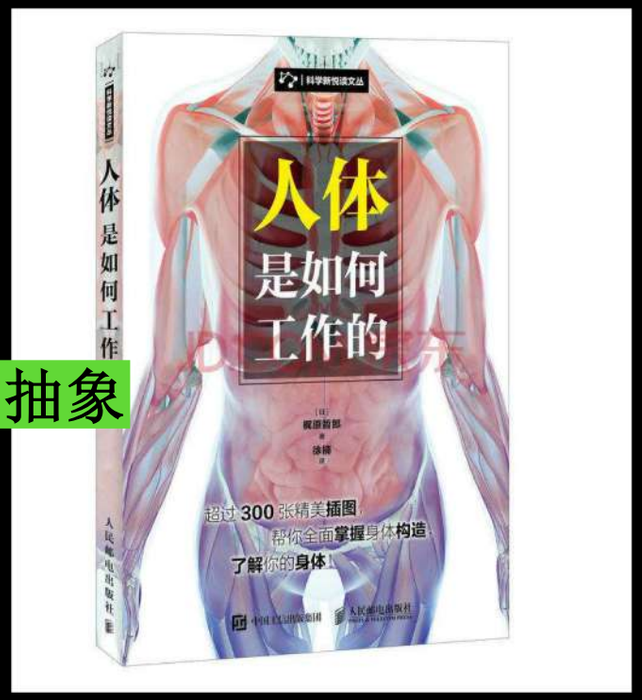
分析的坑

完整映射现实世界到领域模型

人体系统

解决问题

提取有用的抽象



用例模型的操作

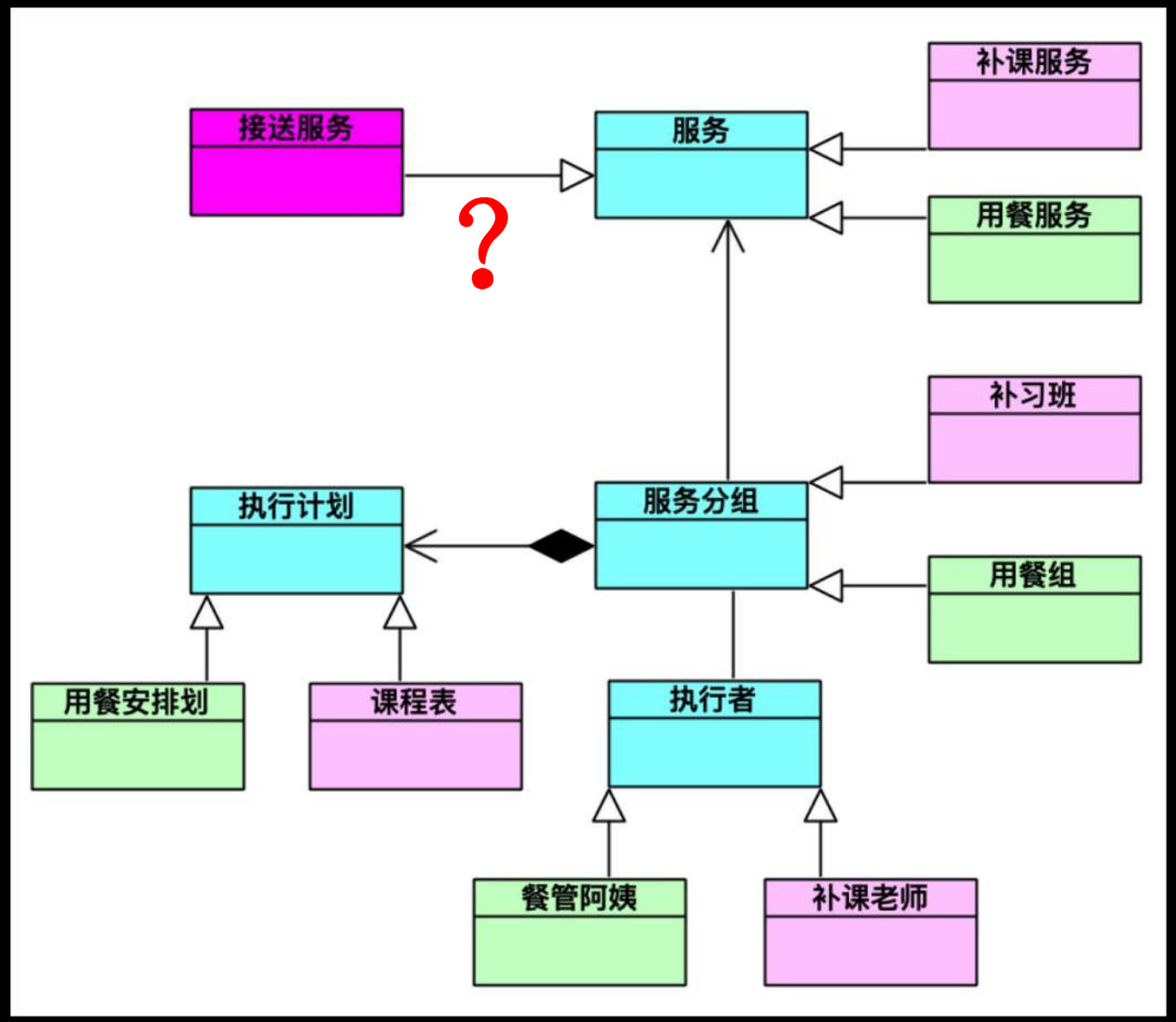
领域模型的结构

完整现实的样子

理由一：会有很多用不上的属性和方法，产生不必要的复杂性

理由二：聚合的大小等同于原子不变性约束的边界大小，这与现实世界概念模型的边界也是不一致的

作茧自缚的抽象



分析模型虽然存在跨业务的共性，但如果行为无关，则不应该以继承的方式被共享使用，分析模式可以作为思考的起点的，但不能作为抽象接口把无关的业务耦合起来。

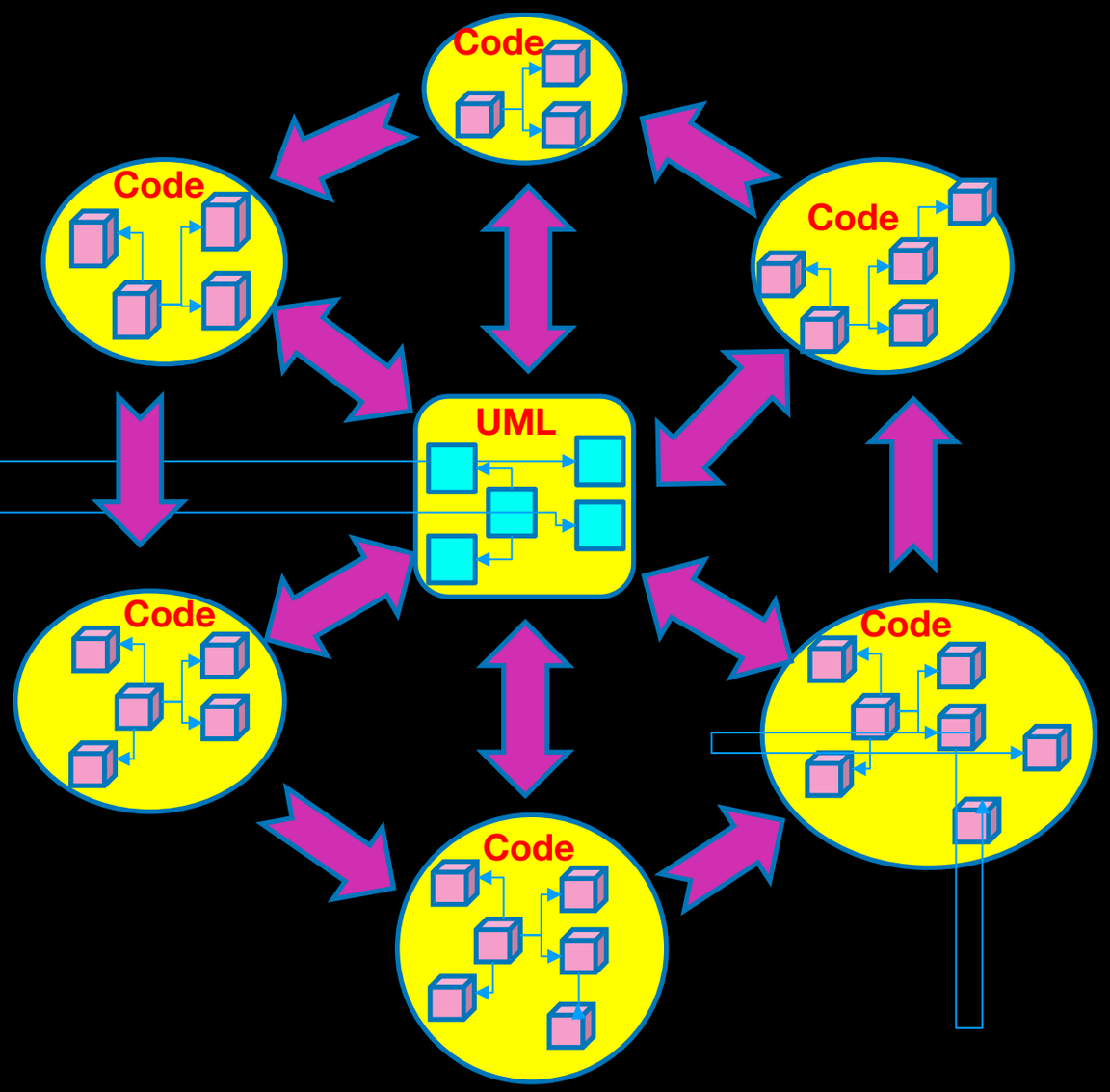
托管班管理系统

没有以发展的眼光看待模型

对待模型的错误观念

- 认为成熟的模型可以不经过版本迭代得到；
- 不愿意为模型的演化提供长期的投入，满足于早期识别的模型；

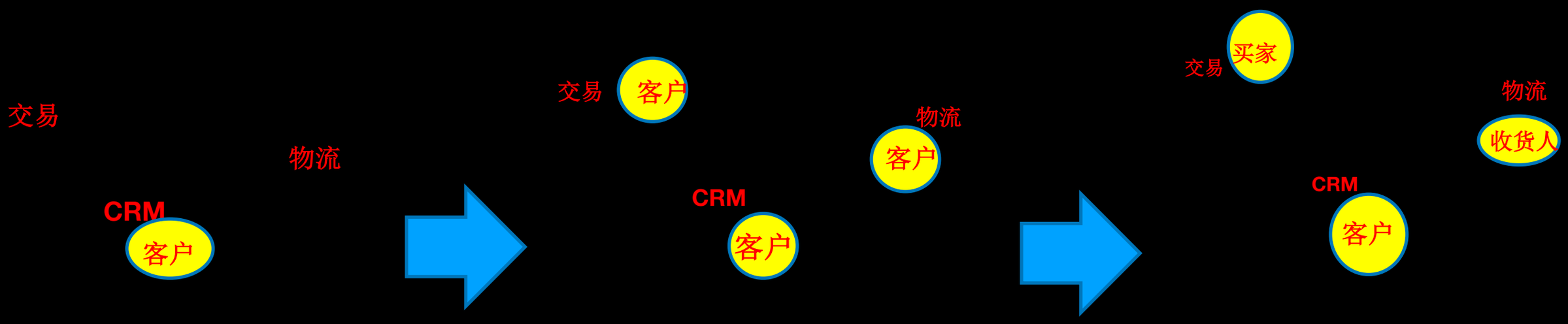
成熟的模型至少需要抛弃两个以上的早期版本



模型需要迭代演化

限界上下文的坑

可能导致上下文耦合的共享模型

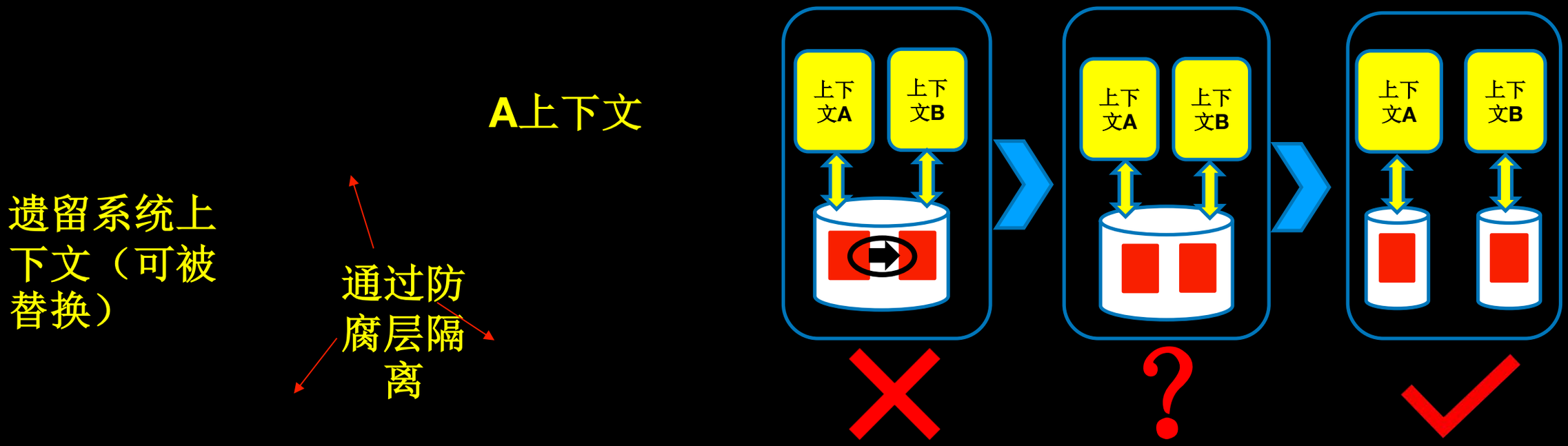


耦合了不同行为的客户　　分解客户在不同BC的行为　　应用四色原型分解客户行为

上下文中模型概念一致还不够，还要保持内涵（行为和职责）一致

藕断丝连的限界上下文边界

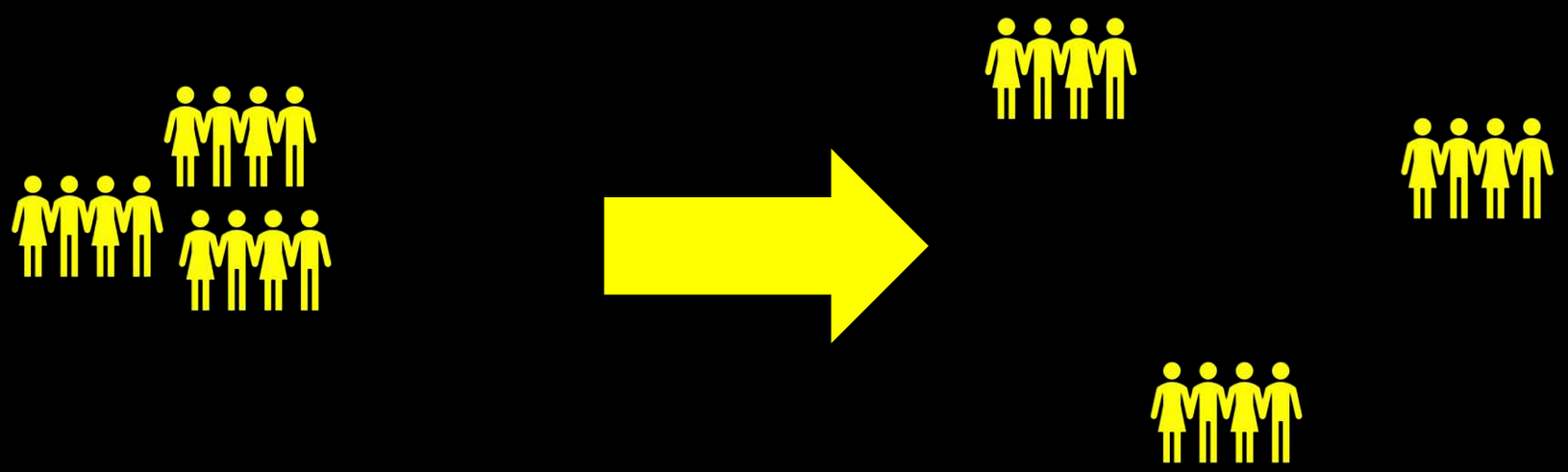
存储上的藕断丝连



BC划分物理边界的意义在于Inter-Operate，而不是Integrate； Inter-Operate是指定义系统边界和接口，并为整个团队提供完整的堆栈，实现完全的自制。如此就能降低系统间的依赖性，减少通信成本；

呃！很可能万恶之源是组织上的藕断丝连

康威定律：设计系统的组织，其产生的设计等价于组织间的沟通结构



大泥球团队

限界上下文团队

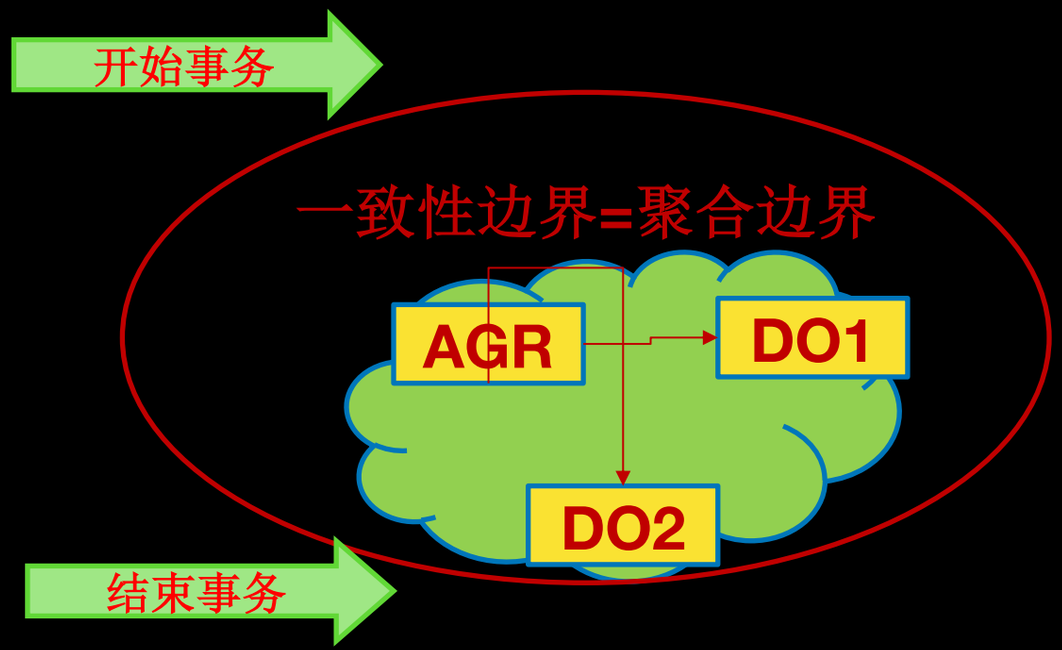
注：业务无关的基础设施维护团队除外

过大的限界上下文团队

超过“Two Pizza”，你的UL规模会很难hold住

聚合的坑

过大的聚合



如果：

AGR存在属性X

DO1存在属性Y

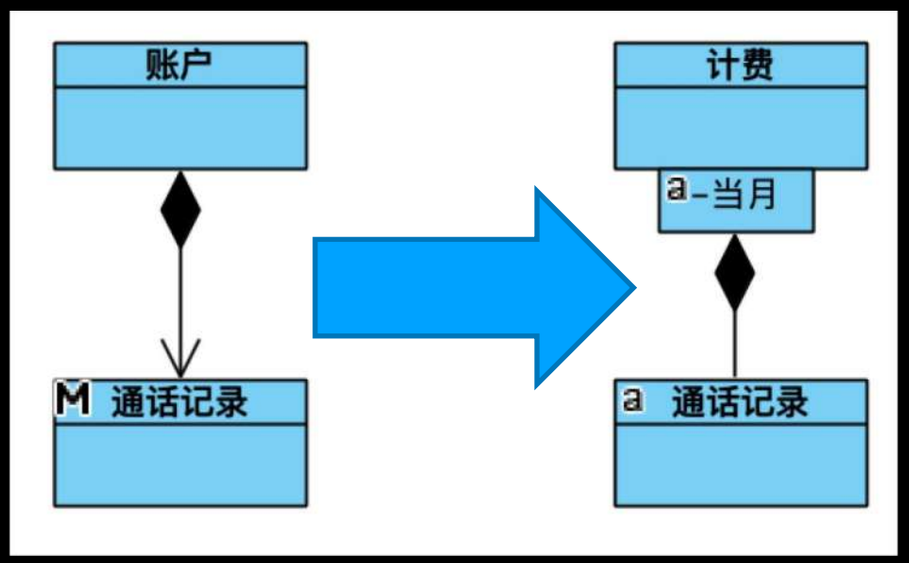
DO2存在属性Z

那么：

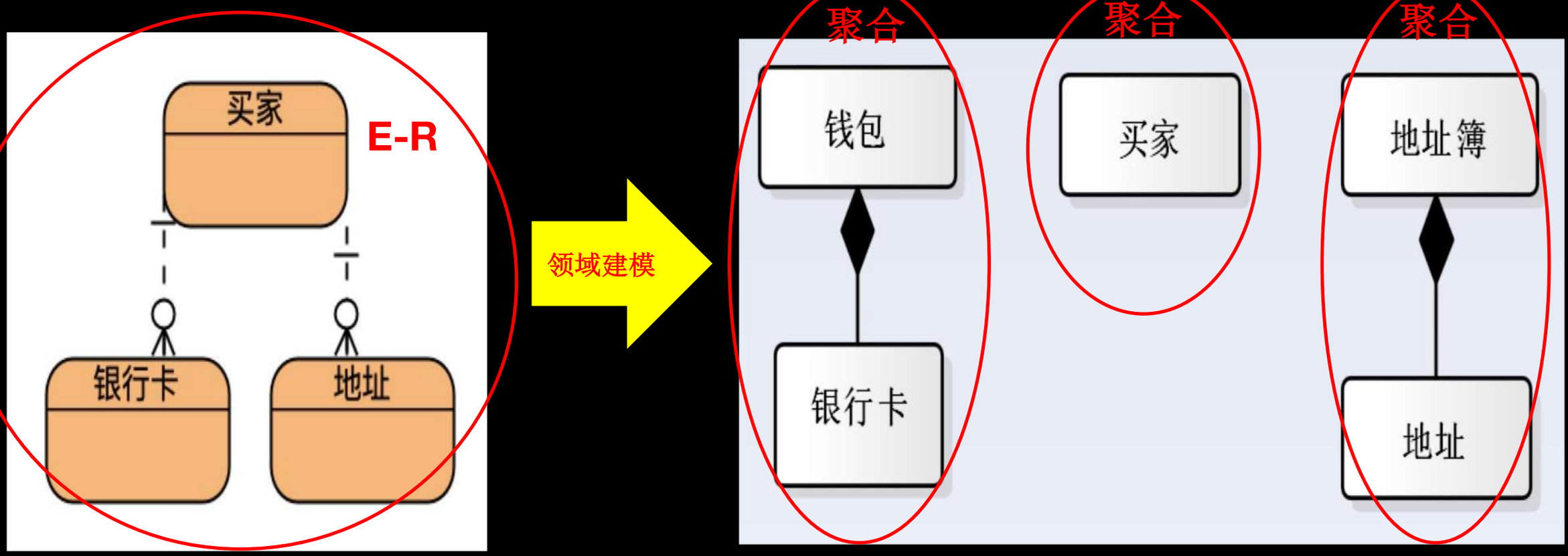
不变性：X=F(Y，Z)

大聚合带来的问题：

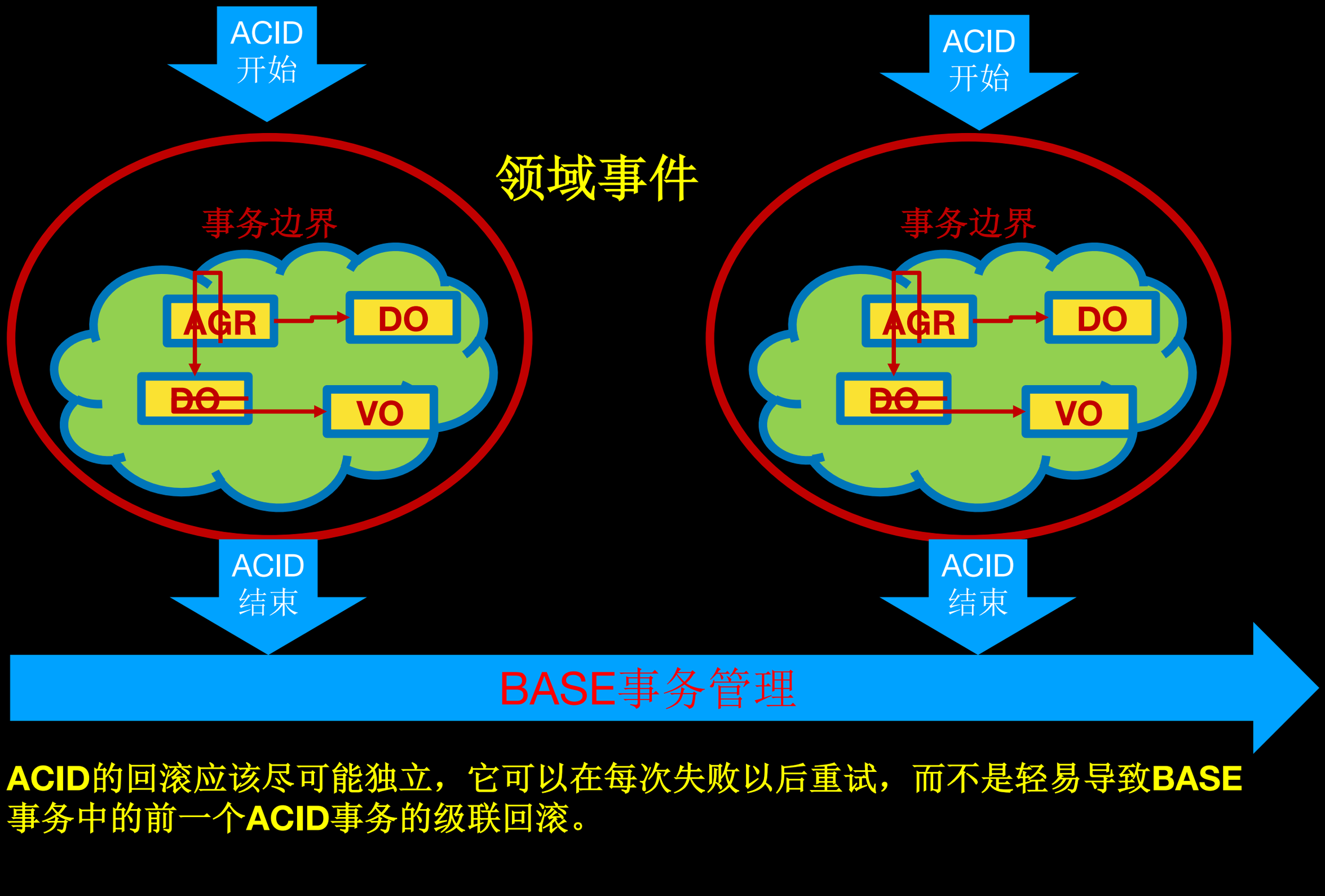
- 数据难以分离和扩容
- 性能问题
- 并发冲突问题



再看一个错误理解聚合的例子



过小的聚合



ACID的回滚应该尽可能独立，它可以在每次失败以后重试，而不是轻易导致BASE事务中的前一个ACID事务的级联回滚。

仓储的坑

延迟加载聚合成员

理由：聚合太大，所以需要延迟加载

后果：可能在循环中进行加载，导致n+1次select问题

解决：　在聚合执行前完成聚合的填充。

```
public class OrderAggregate{
    private void computePrice(){
        for(OrderItem product:orderItems) {
            BigDecimal price=product.getPrice();//也许实时查询
```

上图product.getPrice()延迟加载，产生n+1次select问题，优化的做法是product在聚合构造时就已经全部加载数据，在聚合中依赖注入仓储就是一种典型的延迟加载

数据抓取模式影响了你的聚合？你的聚合可能是基于视图而不是业务规则构建的。

从聚合组装用于展示的查询结果

- 聚合的数据只和不变性有关，和展示无关；
- 如果聚合要兼顾展示，而展示只需要一两个属性，那么也要构造整个聚合；
- 如果聚合要兼顾展示，而或者展示需要来自更多领域对象的属性，那么聚合执行和展示无关的行为时，也要加载这些无关的部分。
- 展示的需求复杂多变，SQL从表中组装数据的能力，比我们从多个聚合手工组装数据的能力强太多，建议采用 CQRS，使得查询不经过聚合；

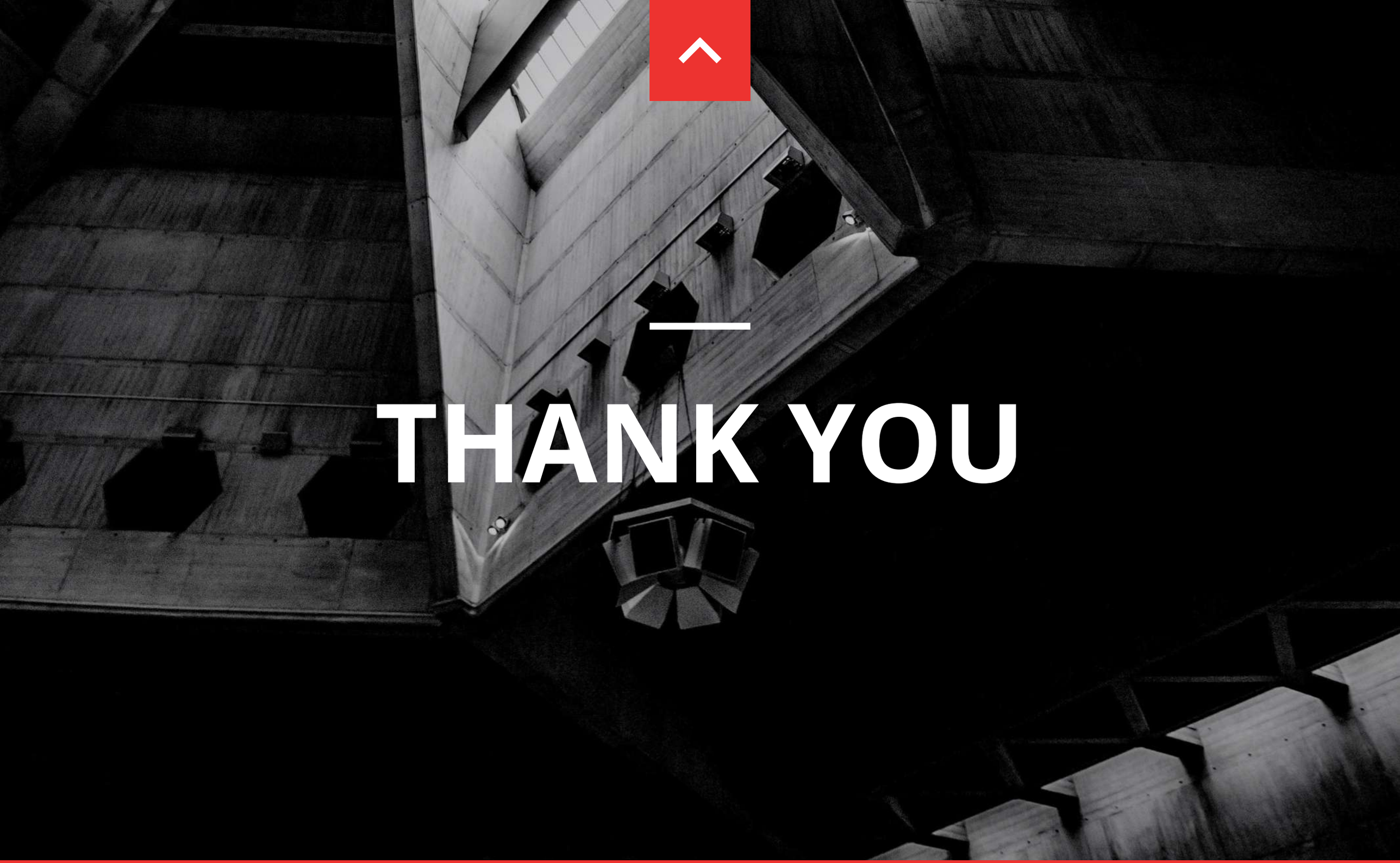
仓储层为领域层提供通用查询方法

```
public interface OrderRepository {

    public List<OrderAggregate> findBy(Query<OrderAggregate> query);
```

- 仓储层的意义在于把领域对象和底层技术完全隔离，使得领域对象可以独立演化；
- 传入的通用查询条件**Query**的组装逻辑依赖于仓储内部的数据结构，这使得仓储的内部数据结构被泄漏到了领域层。破坏了仓储层的隔离作用。
- 通用查询方法无法为不同意图的查询进行优化。

结束了吗？不，还有很多……



THANK YOU

DDD CHINA