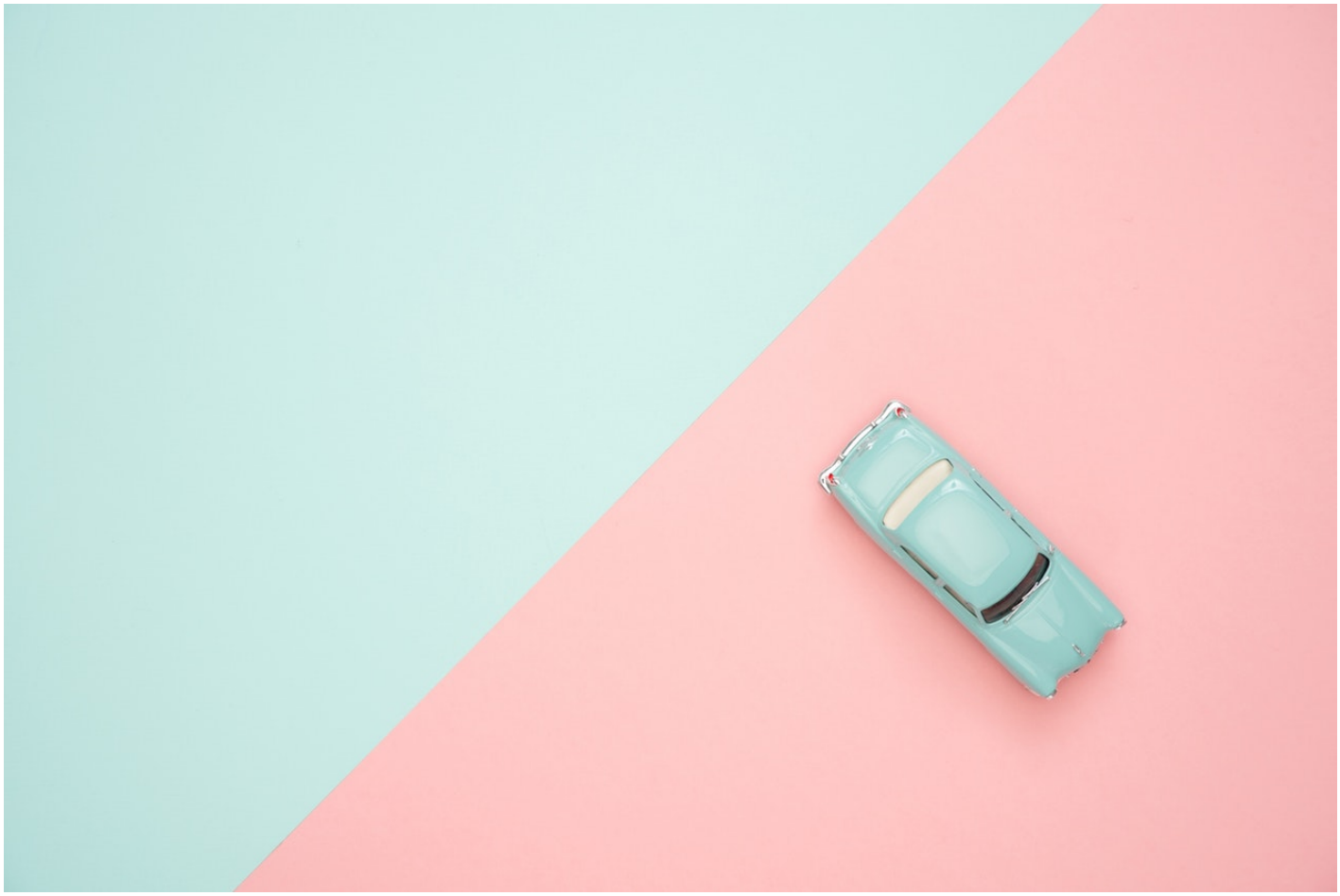
24讲实战一（下）：如何实现一个遵从设计原则的积分兑换系统



上一节课中，我们讲了积分系统的需求分析和系统设计。今天，我们来讲它的代码实现。

上一节课中，我们把积分赚取和消费的渠道和规则的管理维护工作，划分到了上层系统中，所以，积分系统的功能变得非常简单。相应地，代码实现也比较简单。如果你有一定的项目开发经验，那实现这样一个系统，对你来说并不是件难事。

所以，我们今天讲解的重点，并不是教你如何来实现积分系统的每个功能、每个接口，更不是教你如何编写SQL语句来增删改查数据，而是给你展示一些更普适的开发思想。比如，为什么要分MVC三层来开发？为什么要针对每层定义不同的数据对象？最后，我还会总结这其中都蕴含哪些设计原则和思想，让你知其然知其所以然，做到真正地透彻理解。

话不多说，让我们正式开始今天的学习吧！

业务开发包括哪些工作?

实际上，我们平时做业务系统的设计与开发，无外乎有这样三方面的工作要做：接口设计、数据库设计和业务模型设计（也就是业务逻辑）。

数据库和接口的设计非常重要，一旦设计好并投入使用之后，这两部分都不能轻易改动。改动数据库表结构，需要涉及数据的迁移和适配；改动接口，需要推动接口的使用者作相应的代码修改。这两种情况，即便是微小的改动，执行起来都会非常麻烦。因此，我们在设计接口和数据库的时候，一定要多花点心思和时间，切不可过于随意。相反，业务逻辑代码侧重内部实现，不涉及被外部依赖的接口，也不包含持久化的数据，所以对改动的容忍性更大。

针对积分系统，我们先来看，如何设计数据库。

数据库的设计比较简单。实际上，我们只需要一张记录积分流水明细的表就可以了。表中记录积分的赚取和消费流水。用户积

分的各种统计数据，比如总积分、总可用积分等，都可以通过这张表来计算得到。

积分明细表（credit_transaction）	
id	明细ID
user_id	用户ID
channel_id	赚取或消费渠道ID
event_id	相关事件ID，比如订单ID、评论ID、优惠券换购交易ID
credit	积分（赚取为正值、消费为负值）
create_time	积分赚取或消费时间
expired_time	积分过期时间

接下来，我们再来看，如何设计积分系统的接口。

接口设计要符合单一职责原则，粒度越小通用性就越好。但是，接口粒度太小也会带来一些问题。比如，一个功能的实现要调用多个小接口，一方面如果接口调用走网络（特别是公网），多次远程接口调用会影响性能；另一方面，本该在一个接口中完成的原子操作，现在分拆成多个小接口来完成，就可能会涉及分布式事务的数据一致性问题（一个接口执行成功了，但另一个接口执行失败了）。所以，为了兼顾易用性和性能，我们可以借鉴facade（外观）设计模式，在职责单一的细粒度接口之上，再封装一层粗粒度的接口给外部使用。

对于积分系统来说，我们需要设计如下这样几个接口。

接口	参数	返回
赚取积分	userId, channelId, eventId, credit, expiredTime	积分明细ID
消费积分	userId, channelId, eventId, credit, expiredTime	积分明细ID
查询积分	userId	总可用积分
查询总积分明细	userId+分页参数	id, userId, channelId, eventId, credit, createTime, expiredTime
查询赚取积分明细	userId+分页参数	id, userId, channelId, eventId, credit, createTime, expiredTime
查询消费积分明细	userId+分页参数	id, userId, channelId, eventId, credit, createTime, expiredTime

最后，我们来看业务模型的设计。

前面我们讲到，从代码实现角度来说，大部分业务系统的开发都可以分为Controller、Service、Repository三层。Controller层负责接口暴露，Repository层负责数据读写，Service层负责核心业务逻辑，也就是这里说的业务模型。

除此之外，前面我们还提到两种开发模式，基于贫血模型的传统开发模式和基于充血模型的DDD开发模式。前者是一种面向过程的编程风格，后者是一种面向对象的编程风格。不管是DDD还是OOP，高级开发模式的存在一般都是为了应对复杂系统，应对系统的复杂性。对于我们要开发的积分系统来说，因为业务相对比较简单，所以，选择简单的基于贫血模型的传统开发模式就足够了。

从开发的角度来说，我们可以把积分系统作为一个独立的项目，来独立开发，也可以跟其他业务代码（比如营销系统）放到同一个项目中进行开发。从运维的角度来说，我们可以将它跟其他业务一块部署，也可以作为一个微服务独立部署。具体选择哪种开发和部署方式，我们可以参考公司当前的技术架构来决定。

实际上，积分系统业务比较简单，代码量也不多，我更倾向于将它跟营销系统放到一个项目中开发部署。只要我们做好代码的模块化和解耦，让积分相关的业务代码跟其他业务代码之间边界清晰，没有太多耦合，后期如果需要将它拆分成独立的项目来开发部署，那也并不困难。

相信这样一个简单的业务功能的开发，对你来说并没有太大难度。所以，具体的代码实现我就不在专栏中给出了。感兴趣的话，你可以自己实现一下。接下来的内容，才是我们这一节的重点。

为什么要分MVC三层开发?

我们刚刚提到，大部分业务系统的开发都可以分为三层：Contoller层、Service层、Repository层。对于这种分层方式，我相信大部分人都很认同，甚至成为了一种开发习惯，但你有没有想过，为什么我们要分层开发？很多业务都比较简单，一层代码搞定所有的数据读取、业务逻辑、接口暴露不好吗？你可以把它作为一道面试题，试着自己思考下，然后再看我下面的讲解。

对于这个问题，我总结了以下几点原因。

1.分层能起到代码复用的作用

同一个Repository可能会被多个Service来调用，同一个Service可能会被多个Controller调用。比如，UserService中的

getUserById()接口封装了通过ID获取用户信息的逻辑，这部分逻辑可能会被UserController和AdminController等多个Controller使用。如果没有Service层，每个Controller都要重复实现这部分逻辑，显然会违反DRY原则。

2.分层能起到隔离变化的作用

分层体现了一种抽象和封装的设计思想。比如，Repository层封装了对数据库访问的操作，提供了抽象的数据访问接口。基于接口而非实现编程的设计思想，Service层使用Repository层提供的接口，并不关心其底层依赖的是哪种具体的数据库。当我们需要替换数据库的时候，比如从MySQL到Oracle，从Oracle到Redis，只需要改动Repository层的代码，Service层的代码完全不需要修改。

除此之外，Controller、Service、Repository三层代码的稳定程度不同、引起变化的原因不同，所以分成三层来组织代码，能有效地隔离变化。比如，Repository层基于数据库表，而数据库表改动的可能性很小，所以Repository层的代码最稳定，而Controller层提供适配给外部使用的接口，代码经常会变动。分层之后，Controller层中代码的频繁改动并不会影响到稳定的Repository层。

3.分层能起到隔离关注点的作用

Repository层只关注数据的读写。Service层只关注业务逻辑，不关注数据的来源。Controller层只关注与外界打交道，数据校验、封装、格式转换，并不关心业务逻辑。三层之间的关注点不同，分层之后，职责分明，更加符合单一职责原则，代码的内聚性更好。

4.分层能提高代码的可测试性

后面讲单元测试的时候，我们会讲到，单元测试不依赖不可控的外部组件，比如数据库。分层之后，Repsitory层的代码通过依赖注入的方式供Service层使用，当要测试包含核心业务逻辑的Service层代码的时候，我们可以用mock的数据源替代真实的数据库，注入到Service层代码中。代码的可测试性和单元测试我们后面会讲到，这里你稍微了解即可。

5.分层能应对系统的复杂性

所有的代码都放到一个类中，那这个类的代码就会因为需求的迭代而无限膨胀。我们知道，当一个类或一个函数的代码过多之后，可读性、可维护性就会变差。那我们就要想办法拆分。拆分有垂直和水平两个方向。水平方向基于业务来做拆分，就是模块化；垂直方向基于流程来做拆分，就是这里说的分层。

还是那句话，不管是分层、模块化，还是OOP、DDD，以及各种设计模式、原则和思想，都是为了应对复杂系统，应对系统的复杂性。对于简单系统来说，其实是发挥不了作用的，就是俗话说的“杀鸡焉用牛刀”。

BO、VO、Entity存在的意义是什么?

在前面的章节中，我们提到，针对Controller、Service、Repository三层，每层都会定义相应的数据对象，它们分别是VO（View Object）、BO（Business Object）、Entity，例如UserVo、UserBo、UserEntity。在实际的开发中，VO、BO、Entity可能存在大量的重复字段，甚至三者包含的字段完全一样。在开发的过程中，我们经常需要重复定义三个几乎一样的类，显然是一种重复劳动。

相对于每层定义各自的数据对象来说，是不是定义一个公共的数据对象更好些呢?

实际上，我更加推荐每层都定义各自的数据对象这种设计思路，主要有以下3个方面的原因。

- VO、BO、Entity并非完全一样。比如，我们可以在UserEntity、UserBo中定义Password字段，但显然不能在UserVo中定义Password字段，否则就会将用户的密码暴露出去。
- VO、BO、Entity三个类虽然代码重复，但功能语义不重复，从职责上讲是不一样的。所以，也并不能算违背DRY原则。在前面讲到DRY原则的时候，针对这种情况，如果合并为同一个类，那也会存在后期因为需求的变化而需要再拆分的问题。
- 为了尽量减少每层之间的耦合，把职责边界划分明确，每层都会维护自己的数据对象，层与层之间通过接口交互。数据从

下一层传递到上一层的时候，将下一层的数据对象转化成上一层的数据对象，再继续处理。虽然这样的设计稍微有些繁琐，每层都需要定义各自的数据对象，需要做数据对象之间的转化，但是分层清晰。对于非常大的项目来说，结构清晰是第一位的！

既然VO、BO、Entity不能合并，那如何解决代码重复的问题呢？

从设计的角度来说，VO、BO、Entity的设计思路并不违反DRY原则，为了分层清晰、减少耦合，多维护几个类的成本也并不是不能接受的。但是，如果你真的有代码洁癖，对于代码重复的问题，我们也有一些办法来解决。

我们前面讲到，继承可以解决代码重复问题。我们可以将公共的字段定义在父类中，让VO、BO、Entity都继承这个父类，各自只定义特有的字段。因为这里的继承层次很浅，也不复杂，所以使用继承并不会影响代码的可读性和可维护性。后期如果因为业务的需要，有些字段需要从父类移动到子类，或者从子类提取到父类，代码改起来也并不复杂。

前面在讲“多用组合，少用继承”设计思想的时候，我们提到，组合也可以解决代码重复的问题，所以，这里我们还可以将公共的字段抽取到公共的类中，VO、BO、Entity通过组合关系来复用这个类的代码。

代码重复问题解决了，那不同分层之间的数据对象该如何互相转化呢？

当下一层的数据通过接口调用传递到上一层之后，我们需要将它转化成上一层对应的数据对象类型。比如，Service层从Repository层获取的Entity之后，将其转化成BO，再继续业务逻辑的处理。所以，整个开发的过程会涉及“Entity到BO”和“BO到VO”这两种转化。

最简单的转化方式是手动复制。自己写代码在两个对象之间，一个字段一个字段的赋值。但这样的做法显然是没有技术含量的低级劳动。Java中提供了多种数据对象转化工具，比如BeanUtils、Dozer等，可以大大简化繁琐的对象转化工作。如果你是用其他编程语言来做开发，也可以借鉴Java这些工具类的设计思路，自己在项目中实现对象转化工具类。

VO、BO、Entity都是基于贫血模型的，而且为了兼容框架或开发库（比如MyBatis、Dozer、BeanUtils），我们还需要定义每个字段的set方法。这些都违背OOP的封装特性，会导致数据被随意修改。那到底该怎么办好呢？

前面我们也提到过，Entity和VO的生命周期是有限的，都仅限在本层范围内。而对应的Repository层和Controller层也都不包含太多业务逻辑，所以也不会有太多代码随意修改数据，即便设计成贫血、定义每个字段的set方法，相对来说也是安全的。

不过，Service层包含比较多的业务逻辑代码，所以BO就存在被任意修改的风险了。但是，设计的问题本身就没有最优解，只有权衡。为了使用方便，我们只能做一些妥协，放弃BO的封装特性，由程序员自己来负责这些数据对象的不被错误使用。

总结用到的设计原则和思想

前面我们提到，很多人做业务开发，总感觉就是CRUD，翻译代码，根本用不到设计原则、思想和模式。实际上，只是你没有发现而已。现在，我就给你罗列一下，今天讲解的内容中，都用到了哪些设计原则、思想和模式。

总结这两节用到的设计原则和思想	
高内聚、松耦合	上节课中，我们将不同的功能划分到不同的模块，遵从的划分原则就是尽量让模块本身高内聚，让模块之间松耦合。
单一职责原则	上节课中，我们讲到，模块的设计要尽量职责单一，符合单一职责原则。这节课中，分层的一个目的也是为了更加符合单一职责原则。
依赖注入	在MVC三层结构的代码实现中，下一层的类通过依赖注入的方式注入到上一层代码中。
依赖反转原则	在业务系统开发中，如果我们通过类似Spring IOC这样的容器来管理对象的创建、生命周期，那就用到了依赖反转原则。
基于接口而非实现编程	在MVC三层结构的代码实现中，Service层使用Repository层提供的接口，并不关心其底层是依赖的哪种具体的数据库，遵从基于接口而非实现编程的设计思想。
封装、抽象	分层体现了抽象和封装的设计思想，能够隔离变化，隔离关注点。
DRY与继承和组合	尽管VO、BO、Entity存在代码重复，但功能语义不同，并不违反DRY原则。为了解决三者之间的代码重复问题，我们还用到了继承或组合。
面向对象设计	系统设计的过程可以参照面向对象设计的步骤来做。面向对象设计本质是将合适的代码放到合适的类中。系统设计是将合适的功能放到合适的模块中。

实际上，这两节课中还蕴含了很多其他的设计思想、原则、模式，你可以像我一样试着去总结一下，放在留言区说一说。

重点回顾

今天的内容到此就讲完了。我们一块来总结回顾一下，你需要掌握的重点内容。

1.为什么要分MVC三层开发?

对于这个问题，我总结了以下5点原因。

- 分层能起到代码复用的作用
- 分层能起到隔离变化的作用
- 分层能起到隔离关注点的作用
- 分层能提高代码的可测试性
- 分层能应对系统的复杂性

2.BO、VO、Entity存在的意义是什么?

从设计的角度来说，VO、BO、Entity的设计思路并不违反DRY原则，为了分层清晰、减少耦合，多维护几个类的成本也并不是不能接受的。但是，如果你真的有代码洁癖，对于代码重复的问题，我们可以通过继承或者组合来解决。

如何进行数据对象之间的转化？最简单的方式就是手动复制。当然，你也可以使用Java中提供了数据对象转化工具，比如BeanUtils、Dozer等，可以大大简化繁琐的对象转化工作。

尽管VO、BO、Entity的设计违背OOP的封装特性，有被随意修改的风险。但Entity和VO的生命周期是有限的，都仅限在本层范围内，相对来说是安全的。Service层包含比较多的业务逻辑代码，所以BO就存在被任意修改的风险了。为了使用方便，我们只能做一些妥协，放弃BO的封装特性，由程序员自己来负责这些数据对象的不被错误使用。

3.总结用到的设计原则和思想

从表面上看，做业务开发可能并不是特别有技术挑战，但是实际上，如果你要做到知其然知其所以然，做到透彻理解、真的懂，并不是件容易的事情。深挖一下，你会发现这其中还是蕴含了很多设计原则、思想和模式的。

课堂讨论

1. 上节课中，我们讲到，下层系统不要包含太多上层系统的业务信息。但在今天的数据库设计中，积分明细表中credit_transaction中包含event_id，channel_id这些跟上层业务相关的字段，那这样的设计是否合理呢?
2. 我们经常说，修改和查询不要耦合在一个接口中，要分成两个接口来做。赚取积分和消费积分接口返回积分明细ID，这样的接口设计是否违背单一职责原则呢？是不是返回void或者boolean类型更合理呢?

欢迎在留言区写下你的答案，和同学一起交流和分享。如果有收获，也欢迎你把这篇文章分享给你的朋友。

精选留言

桂城老托尼

感谢争哥分享。

个人觉得积分系统只有流水记录不太够，设想下消费积分场景，要完成扣积分的动作，没有积分余额表容易造成多扣，特别是针对羊毛党。当然上层系统，比如活动系统也可以做好幂等。

尝试回答下课后讨论

1. 保留上层应用id和channel完全符合设计原理，冗余业务信息方便日后做数据统计，沉淀数据资产反推营销策略迭代; 一次消费或赚取积分行为可能存在多次调用情况，方便幂等，不至于多次记账;方便业务系统查询某次赚取或消费的积分明细;暂时想到这么多。

2. 尽量不要返回void或boolean，有些业务需要反向关键积分流水id做单笔流水查询。

其实这两个讨论都类似于现实生活，去便利店买东西会给你小票，支付宝扫码付款会返回交易流水。都是为了方便真是场景解决“纠纷”(查询)用的。

2019-12-27 08:40

Jxin

1.不符合一对上下游系统的设计要求，但适合当下业务场景的需求。 下游积分核心系统设计上不该持有事件和渠道字段，因为它不该去关心上游业务，事件或渠道与对应积分明细的关联应该由上游系统来维护，或则在上游系统和积分系统之间再加一层积分系统的业务层，用于维护这层关系（关于易复用性的中台思想）。当下的业务场景，积分的管理系统是有必要维护一份 事件渠道的值对象的。因为带有这个值对象，积分系统管理员才能不需要再多个系统中寻找积分增减关系，进而可以独立满足管理积分这件事的整个业务域。（事件和渠道只能作为值对象冗余在积分系统）。

2.不符合单一职责的限制，但满足当前业务场景的诉求。该接口做了增减积分和返回积分id两件事，且语义上并没有返回积分id的相关字眼，所以方法名定义也不明确。但是上游系统在变更积分后，需要获取积分id以作为上游系统变更事件与积分记录的关联key。而这个key只有在当前变更操作获得，所以就只能写这种语义不明且违反单一职责的方法。（让我来设计，我会把积分id的生成作为积分系统的一个外放接口，上游业务调用该接口获取id，记录关联关系，然后走mq推积分系统实现最终扣减。这样就可以规避上述这种无奈的场景）。

2019-12-27 13:12

李朝辉

课堂讨论1：

业务驱动的系统还是应该从业务的角度出发去做设计，这两个字段在积分明细查询中是不可或缺的，所以我认为是合理的。既然是不可或缺的，如果不记录在这张表中，就要记录在其他表中，或者查询不便，或者破坏内聚。

2.根据个人经验，insert操作的都是返回记录id，原因的个人观点是为调用方提供便利。还请老师解答

2019-12-27 08:24

作者回复

说的挺对的

2020-01-02 15:59

杨杰

个人感觉：VO、BO、Entity 通过组合关系来复用这个类的代码不是特别好，尤其是VO。因为用组合的方式会增加返回数据的层次，这对前端来说是不是不不太友好?

2019-12-27 16:16

作者回复

是的 主要是对象转json的格式问题

2020-01-02 16:04

辣么大

反馈一个文章朗读的小问题：音频3:00左右 facade设计模式 应该读做/fə'sɑ:d/ ，冯老师读的不是很准确。 ”吹毛求疵“，希望专栏做的更专业。

2019-12-27 16:13

作者回复

应该吹毛求疵，我们fix去

2019-12-27 19:27

辣么大

针对争哥的第一个问题，从设计角度来说不应该记录渠道和事件。从业务来说，必须记录交易的渠道和事件。基于这种妥协可以设计一张表。那是否也可以设计两张表?

积分交易表和明细表：

1、credit_transaction

trans_id

user_id

channel_id

event_id

create_time

2、积分明细表credit_detail，只记录积分加减

trans_id

credit (积分加减)

create_time

expire_time

2019-12-27 08:41

黄林晴

打卡✔

难道是我膨胀了，实战的内容没有理论看的起劲

2019-12-27 08:27

作者回复

期望太高了 也不可能篇篇让你高潮

2019-12-30 08:49

aya

1增加上层相关字段有利于出现问题时的排查工作，并且ods等系统在抽数据时也可以提供完整数据。

2不违背单一指责原则，赚取积分和消费积分的业务逻辑必然会伴随积分余额查询，业务上应属同一逻辑，拆分反而shi程序复杂。

2019-12-27 08:02

杨陆伟

VO、BO、Entity频繁的克隆、拷贝会不会引入性能问题，这点是怎么考虑的？谢谢

2020-01-02 09:06

作者回复

虽然有性能损耗 但可以忽略 影响不大

2020-01-02 10:25

Jeff.Smile

实际中没见过同时定义vo bo entity的

2019-12-27 09:25

下雨天

说说第一个问题，系统设计跟数据库设计没必然联系，这样设计合理！

数据库设计中有点像服务器请求设计

1.数据库拆分过细，各表字段间关联变麻烦了，提高数据不一致性风险！

2.数据库中表多意味着访问可能性变大，数据库的连接，建立都是需要时间和消耗性能的！

2019-12-27 09:16

林

为什么积分明细表没有user_id字段，如何查询用户的可用积分

2020-01-02 11:55

作者回复

是个问题 我改下

2020-01-02 16:57

斐波那契

老师 beanutils会不会性能不好 毕竟大量用到了反射 有没有代码不那么繁琐性能也比较好的方法

2019-12-31 14:14

作者回复

对于大部分业务系统来说 数据库是最耗时的 对象转化那点性能损失可以忽略

2020-01-02 14:22

兴

第一次评论说的不对请指正

第一个问题分两种情况：

1. 如果是积分和营销业务分离的话，下层业务就不应该关注上层业务，积分系统只记录自己的流水，event_id和channel_id则应该由上层记录

2. 如果是积分和业务系统放在一起的话（就像争哥推荐的那样），这种方案就符合当前业务情景。

第二个问题：

这种虽不符合单一职责，但如果每次插入后业务需要ID的做其他事情的话（或者以后也需要），这种方案的成本最小。

2019-12-29 00:13

胖子

我有一个疑问：接口设计、数据库设计和业务模型设计如何对应或过渡到MVC分层?

2019-12-27 12:57

作者回复

接口 controller层

数据库 repository

业务 service

2019-12-30 08:45

京京beaver

第一个问题：合理。设计积分和金钱类的修改需要对账，不引入渠道id和事件id，无法与高层系统完成对账，是设计隐患，

第二个问题：不违反。积分明细 ID是本次事务执行成功的回执，是后续对账，发起重试，处理幂等的依据。属于正常返回值。如果只返回void或者boolean没有任何业务含义，存在安全隐患。

2020-01-08 16:50

传说中的成大大

1. 是合理的 因为虽然数据库设计包含了上层的数据，但是绝大多数都是用来进行存数据库，存记录方便以后的维护和查询，比

如像游戏开发 在数据库模块中总会设计到玩家角色id啊 名字啊等等 所以我觉得是合理的

2. 并没有违背单一职责原则，因为修改或者查询的接口里面做的事情是很单一的，返回一个id，还便于去查询验证

2020-01-02 14:08

作者回复

2020-01-03 08:09

zk_207

争哥，

Vo和Bo应该分别放到Controller层、service层呢，还是统一放到domain层呢?

2019-12-30 10:14

作者回复

vo放到controller层 bo放到service层

2020-01-02 15:06

陈迎春

各位大佬好，我主要是做嵌入式软件开发的，平常设计模式用的不多，所以相对差一些，最近的两偏实战，我听了好几遍，但还是云里雾里，可否给出一些实际的代码？结合代码理解，可能会更好些

2019-12-29 23:12

作者回复

估计你要自己下点功夫 毕竟讲的跟你现在做的有比较大距离

2019-12-30 08:30

知行合一

争哥给出的是一个基础的积分系统案例，主要是从设计的角度来分析我们用到哪些原则和模式。真实场景的话肯定需要考虑高并发和大数据量的问题，那样的话可能需要单独出来积分账户表，存储积分余额，明细表需要分库分表。

针对问题一，我觉的很有必要，毕竟业务需要查询积分的来源和做幂等性检验。

问题二，这个主要看业务系统是否需要，大部分情况下我觉得是不需要的，业务系统增加积分或者消费积分就成了。需要的话也是为了对账，我哪个动作增加的哪笔积分，对账意义大于业务意义。

2019-12-27 12:17