积分系统的设计

需求分析

- 积分赚取和兑换规则
- 积分消费和兑换规则
- 积分及其明细查询

系统设计

- 三种模块划分方法：
- 第一种划分方式是：积分赚取渠道及兑换规则、消费渠道及兑换规则的管理和维护（增删改查），不划分到积分系统中，而是放到更上层的营销系统中。这样积分系统就会变得非常简单，只需要负责增加积分、减少积分、查询积分、查询积分明细等这几个工作。
- 比如，用户通过下订单赚取积分。订单系统通过异步发送消息或者同步调用接口的方式，告知营销系统订单交易成功。营销系统根据拿到的订单信息，查询订单对应的积分兑换规则（兑换比例、有效期等），计算得到订单可兑换的积分数量，然后调用积分系统的接口给用户增加积分。

- 第二种划分方式是：积分赚取渠道及兑换规则、消费渠道及兑换规则的管理和维护，分散在各个相关业务系统中，比如订单系统、评论系统、签到系统、换购商城、优惠券系统等。
- 还是刚刚那个下订单赚取积分的例子，在这种情况下，用户下订单成功之后，订单系统根据商品对应的积分兑换比例，计算所能兑换的积分数量，然后直接调用积分系统给用户增加积分。

- 第三种划分方式是：所有的功能都划分到积分系统中，包括积分赚取渠道及兑换规则、消费渠道及兑换规则的管理和维护。
- 还是同样的例子，用户下订单成功之后，订单系统直接告知积分系统订单交易成功，积分系统根据订单信息查询积分兑换规则，给用户增加积分。

设计模块与模块之间的交互关系

- 常见的系统之间的交互方式有两种：一种是同步接口调用，另一种是利用消息中间件异步调用。
- 第一种方式简单直接，第二种方式的解耦效果更好。
- 上下层系统之间的调用倾向于通过同步接口，同层之间的调用倾向于异步消息调用。比如，营销系统和积分系统是上下层关系，它们之间就比较推荐使用同步接口调用。

业务系统的设计与开发

- 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）。

数据库设计

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

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。
  - 多次远程接口调用会影响性能
  - 多个小接口可能会涉及分布式事务的数据一致性问题
- 可以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。

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

分层

- 大部分业务系统的开发都可以分为三层：Contoller 层、Service 层、Repository 层。好处是：
- 1. 分层能起到代码复用的作用
- 2. 分层能起到隔离变化的作用
- 3. 分层能起到隔离关注点的作用
- 4. 分层能提高代码的可测试性
- 5. 分层能应对系统的复杂性

BO、VO、Entity

- 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。
- VO、BO、Entity 三个类虽然代码重复，但功能语义不重复。
- 可以使用继承或组合，解决代码重复问题。

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。