一、什么是幂等

1、错误的认知

1）每次相同请求返回相同结果

不只是对上游的结果相同，对自身造成的影响也是相同的。

2）幂等就是防重

防重是实现幂等的一种方式；防重是指提交多次相同的请求到后台，系统必须能够去重，防止重复执行。

2、定义

幂等（idempotence）一词原为数学上的概念，用一个最直观的数学式子表达为：f(f(x)) = f(x)

对应到开发领域，是指相同请求（identical request）执行多次，返回结果都相同，以及所带来的副作用（side-effects）或者影响都是一样的。

此定义来自HTTP幂等性定义，HTTP方法的幂等性是指一次和多次请求某一个资源应该具有同样的副作用。副作用或影响是指后续多次重复调用对系统的数据一致性不造成破坏，即不会写入重复数据，不会执行重复动作（比如重复支付）。简单点说，就是接口可重复调用。（包括时间和空间上两个维度）。

二、为什么做幂等

1、幂等重要性

针对一个分布式服务（微服务）架构，如果不支持幂等操作，那将会出现以下情况：

1）电商超卖

商品库存系统，并发情况下（秒杀），存在多人并发库存只减1。

2）重复支付

订单系统或结算系统生成重复数据，导致重复付款、扣款等。

3）虚拟资产损失

重复增加虚拟账户余额、积分或优惠券等虚拟资产。

2、CURD分析

读取（Read）和删除（Delete）是天然幂等的，这里说的读取是纯查询，不包含附加的写入操作，比如计数器逻辑。

创建（Create）和更新（Update）通常是不幂等的，这里说的更新是计算式Update，即：`UPDATE goods SET number=number-1 WHERE id=1`；非计算式Update，具备幂等性，即：`UPDATE goods SET number=newNumber WHERE id=1`

3、重复请求场景

分布式服务中，无论是用户重复操作角度，还是提高服务高可用性角度，都少不了重试，重试就会重复请求。

1）前端重复提交

前端页面，用户快速重复点击多次，造成提交多个重复的表单。

2）接口失败重试

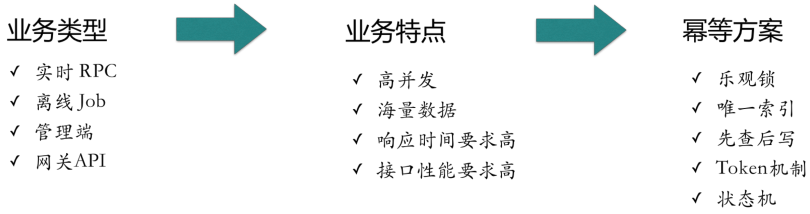
分布式服务中，调用其他系统的接口，为了防止网络抖动、服务超时等原因造成请求丢失，一般会设计超时重试。

3）消息重复消费

MQ消息中间件，消息重复消费。

三、怎么做幂等

思考



不同场景、不同业务 对幂等的要求不同，使用的幂等实现方式也不同

前端怎么做?

重点介绍后端的实现方式，前端简单介绍；防止表单重复提交，按钮置灰、隐藏、不可点击等方式。

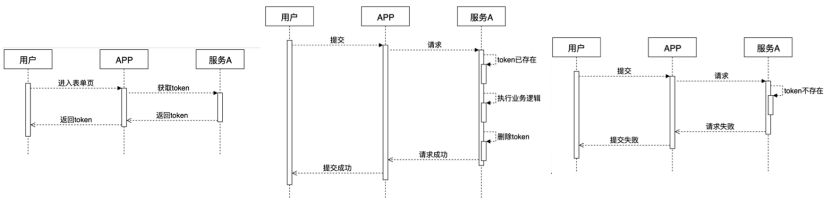
后端怎么做?

1、Token机制

适用场景：写入/更新场景，跟前端（页面、app）直接交互的接口，比如网关api等，有用户无意或恶意的重复提交。

描述：即使前端做了防止重复请求，后端也要设计重复提交幂等校验。比如：用户购物提交订单，提交订单的接口就可以通过Token 的机制实现防止重复提交。

使用方式：



获取Token　　带Token请求　　重复请求

1）当用户进入到表单页面的时候，前端会从服务端申请到一个token，并保存在前端。

2）当用户第一次点击提交的时候，会将该token和表单数据一并提交到服务端（一般token放在请求头中），服务端判断该token是否存在，如果存在则执行业务逻辑。（分布式系统一般token存在redis中）

3）当用户第二次点击提交的时候，会将该token和表单数据一并提交到服务端，服务端判断该token是否存在，如果不存在则返回错误提示，前端显示重复提交。

2、唯一索引

适用场景：写入场景，分布式后端服务调用，避免不了重试的场景。

实现方式：

两种实现方式，一种是设计防重表，一种是在业务表上建唯一索引；防重表优势是可以做成通用幂等模块，相关业务子域共用，与业务解耦，职能清晰。

1）业务表上建唯一索引

利用数据库的唯一索引特性，保证唯一的逻辑，可以是一个字段建索引，也可以是多个字段建组合索引。

注意：如果分库分表，唯一索引需要跟分表键保持一致。

2）防重表

设计一张防重表，不含任何业务属性，在所有接口入口处统一做防重处理。也适用于老系统不想或者不能新增唯一索引字段的情况。

使用方式：

```
    public ClearDetail save(ClearDetail clearDetail) {
      try {
         clearDetailMapper.insert(clearDetail);
      } catch (DuplicateKeyException e) {
         clearDetail = clearDetailMapper.selectByUniqueKey(clearDetail.getUserId(),clearDetail.getVoucherN
      return clearDetail;
    }
```

3、状态机

适用场景：更新场景，有状态流转的业务，比如订单、审核流等。

描述：对于很多业务是有一个业务流转状态的，每个状态都有前置状态和后置状态，以及最后的结束状态。例如流程的待审批，审批中，驳回，审批通过，审批拒绝。订单的待提交，待支付，已支付，取消。

使用方式：

```
//以订单为例，已支付的状态的前置状态只能是待支付，通过这种状态机的流转我们就可以控制请求的幂等。
enum OrderStatusEnum {
    ORDER_CREATET(1, "订单创建"),
    PAY_NEED(2, "待支付"),
    PAY_DONE(3, "已支付"),
    ORDER_FINISHED(4, "订单完成"),
    ORDER_CANCEL(5, "订单取消")
    ;
}
```

4、乐观锁

适用场景：更新场景，高并发场景下的更新，对接口有性能要求的。

描述：用于在数据库并发访问时的情况。当数据更新时需要去判断版本号是否一致，如果不一致，则无法对数据进行更新。接口性能优于悲观锁、同步锁。

使用方式：

```
update table set deposit = deposit-#{amount}, version = version + 1 where id = #{id} and version = #{versi
```

5、先查后写

适用场景：写入/更新场景，并发不高的后台系统，或者一些任务JOB；并发场景用要配合分布式锁使用，查询要走主库，否则主从延迟会导致幂等失效（除非并发锁失效时间完全能cover住主从延迟时间）。

先查后写（select insert）或先查后更新（select update），高并发场景，接口对性能要求比较高的不建议这么写，每次请求都多一次DB 网络IO，对数据库qbs造成双倍压力。

6、分布式锁

适用场景：写入/更新场景，写入场景需要配合先查后写（更新）的操作，分布式锁解决这个非原子操作的并发问题；

描述：与去重表思路相同，只是将对数据库的的访问转移到了对缓存（如redis）的访问，提高了效率。解决高并发业务场景下的并发问题。业务入口，用唯一单号去redis加锁（底层setnx），finally释放锁（compareAndDelete）

使用方式：

```
try {
    lockValue = lockService.acquireLock(LockType.VOUCHER_LOCK, ExpireEnum.EXPIRE_MINUTE, t.getOutNo();
    if (null == lockValue || LockResultCode.SUCCESS != lockValue.getLockResult()) {
       LOGGER.info("XXX接口，并发锁抢占失败,out_no:{}",  t.getOutNo());
       throw new BizException(ResponseEnum.OTHER_ERR_LOCK);
    }
    //核心业务处理
} finally {
    lockService.releaseLock(lockValue, LockType.VOUCHER_LOCK, t.getOutNo();
}
```

7、悲观锁

适用场景：查询场景，适合并发不高的后台系统或Job任务，即对性能效率没有太多要求的。

描述：查询的时候显示上锁，防止查到的数据不一样。id字段一定是主键或者唯一索引，不然是锁表

使用方式：

```
select * from table where id=#{id} for update;
```

防重校验机制

SQL代码：

```
SELECT *
  from table
 where valid=1
   and time ='20200101'
 group by out_no
having count(1)>1
```