# 高并发下幂等性的七大解决方案（图文总结）

原创 ArchitePlus 架构与思维 2025年04月07日 08:00 福建

01

架构 — 思维

## 什么是幂等性问题？

**幂等（idempotent）是一个数学与计算机学概念，常见于抽象代数中。**
在我们的开发过程中，保证幂等性就是保证你的程序的无论执行多少次，影响均与第一次执行的影响是一致的，产生的结果也是一样的。

而幂等函数(幂等方法)，是指使用相同的参数结构重复执行，产生相同的结果的函数，重复执行幂等函数不会影响系统的状态或者造成改变。
例如，"getUserName(String uCode)" 和 "delUser(String uCode)" 函数就是典型的幂等函数，而更复杂的幂等保证是类似 高并发场景下的订单号（流水号）或者 秒杀场景下的唯一有效数据 等。
所以， **幂等指的是一个操作，不论执行多少次，产生的效果和返回的结果都是一样的。**

### 1.1 常见幂等性问题

**典型的违反幂等原则导致的问题，如：**

#### 1. 订单处理场景

**订单创建**：用户提交订单时，可能因网络问题导致请求重复发送，系统需确保同一订单号只创建一次，避免生成多个相同订单。
**订单状态变更**：订单状态从“待支付”变更为“已支付”时，若客户端重复发送状态变更请求，系统需保证状态只变更一次，避免状态不一致。

#### 2. 库存扣减场景

**高并发抢购**：在秒杀、抢购活动中， 大量用户同时发起库存扣减请求，系统需保证同一商品库存只被扣减一次，避免超卖。
**分布式库存扣减**：在分布式系统中， 多个服务节点可能同时处理同一商品的库存扣减请求，需通过幂等性设计保证数据一致性。

#### 3. 支付与退款场景

**支付重复提交**：用户在网络波动或前端响应延迟时，可能多次点击支付按钮，导致 同一笔订单被多次扣款。
**退款操作**：在退款接口中，若客户端因超时未收到响应而重试，或恶意用户利用漏洞重复提交退款请求，导致商家资金损失。

#### 4. 消息队列场景

**消息重复投递**：在消息队列（如Kafka、RabbitMQ）中，消息可能因网络问题或消费者处理失败而被重复投递。例如， 库存扣减消息被重复消费，导致库存实际数据不一致。
**消息重复消费**：消费者在处理消息后，未正确发送确认信号，导致消息被重新投递，系统需确保消息只被处理一次。 我们经常会收到多条短信或者IM消息，就是这种情况。

### 1.2 幂等性目标

- 同一请求多次执行 → 系统状态始终一致

- 不同请求 → 系统正常处理差异

02

架构 — 思维

## 常见的七种解决方案

### 🔑 方案1：Token令牌机制（防重复提交）

**原理**：Redis创建Token，客户端先获取唯一Token，后续请求携带Token，服务端校验是否已被使用。
**适用场景**：订单创建、表单提交等**插入类操作**
**代码示例（Java + Redis）**：

```
// Token生成接口
@GetMapping("/token")
public String getToken() {
    String token = UUID.randomUUID().toString();
    redisTemplate.opsForValue().set(token, "1", 30, TimeUnit.SECONDS);
    return token;
}

// 业务接口
@PostMapping("/order")
public String createOrder(@RequestHeader("X-Token") String token) {
    // 通过删除操作检查Token是否已被使用，避免重复处理。
    if (redisTemplate.delete(token) == 0) {
        return "重复请求，请忽略";
    }
    // 执行下单逻辑...
    return "下单成功";
}

```

**Token机制流程图**：

## 🎨 方案2：唯一ID + 去重表（防重复处理）

**原理**：为每次请求生成唯一ID（如UUID、雪花算法生成的ID），请求前查询去重表，若存在则直接返回结果。
**适用场景**：支付回调、订单状态更新等**更新类操作**
**实现要点**：

- 使用雪花算法生成分布式唯一ID
- 数据库建立唯一索引保证原子性
  **缺点**：
- 需要额外的数据库查询操作，可能影响性能。
- 防重表可能成为性能瓶颈，需合理设计索引和分片。

**SQL示例**：

```
CREATE TABLE idempotence_record (
    biz_id VARCHAR(64) PRIMARY KEY COMMENT '业务唯一ID',
    status TINYINT COMMENT '处理状态',
    create_time DATETIME
);
```

## ⏳ 方案3：唯一索引或唯一组合索引

**原理**：唯一索引或唯一组合索引来防止新增数据出现脏数据（当表存在唯一索引，并发执行时，先进入的执行成功，后进入的会执行失败，说明该数据已经存在了，返回结果即可）。
**适用场景**：订单创建、注册、会议室抢订等**插入类操作**，避免插入同样信息的脏数据。
**典型案例**：
比如：中秋节到了，淘宝上线某款限量版的月饼，每个用户都只能购买一盒月饼，如何防止用户被创建多条月饼订单数据，可以给月饼销售表中的用户ID加唯一索引（不允许被索引的数据列包含重复的值），
保证一个用户只能创建成功一条月饼订单记录。

**SQL示例**：

```
CREATE UNIQUE INDEX uni_user_userid ON t_user(userid);
```

**订单信息表防重机制**：

## 🧩 方案4：乐观锁（读多写少场景）

**原理**：通过版本号控制更新，仅当数据未被修改时执行操作。
**适用场景**：账户余额更新、库存扣减
**代码示例（Java CAS操作）**：

```
@Update("UPDATE account SET balance = balance + #{amount}, version = ve
int updateBalance(@Param("id") Long id, @Param("amount") BigDecimal amo
```

**版本号执行过程图例**：

## 🚦 方案5：状态机（有状态流转）

**原理**：定义状态转移规则，确保操作只能向合法状态跳转。
**适用场景**：订单生命周期管理（如：已支付 → 已发货）

**典型案例**：
在设计单据相关的业务，或者是任务相关的业务，肯定会涉及到状态机(状态变更图)，就是业务单据上面有个状态，状态在不同的情况下会发生变更，一般情况下存在有限状态机，

这时候，如果状态机已经处于下一个状态，这时候来了一个上一个状态的变更，理论上是不能够变更的，这样的话，保证了有限状态机的幂等。

注意：订单等单据类业务，存在很长的状态流转，一定要深刻理解状态机，对业务系统设计能力提高有很大帮助

**代码逻辑**：

```
if (currentStatus == Status.PAID && targetStatus == Status.SHIPPED) {
    // 允许状态转移
} else {
    throw new IllegalStateException("非法状态跳转");
}
```

**电商购物全流程的状态流转**：

## 🔒 方案6：分布式锁（强一致性要求）

**原理**：通过Redis或Zookeeper实现互斥锁，确保同一时间仅一个请求处理。

**适用场景**：超卖防护、秒杀库存扣减

**Java+Redis实现**：

```
Boolean locked = redisTemplate.opsForValue().setIfAbsent("lock_key", "1
if (locked != null && locked) {
    try {
        // 执行库存扣减
    } finally {
        redisTemplate.delete("lock_key");
    }
```

```
8 }
```

### 📜 方案7：前后端双重校验

**原理**：前端防抖（如按钮置灰或者浮起旋转框）+ 后端唯一ID校验
**适用场景**：低并发场景快速实现（如用户注册）

## 03
架构 — 思维
## 方案对比和选型参考

| 方案 | 性能 | 复杂度 | 适用场景 |
| --- | --- | --- | --- |
| Token机制 | 高 | 低 | 插入类操作（订单创建） |
| 唯一ID + 去重表 | 中 | 中 | 更新类操作（状态变更） |
| 唯一索引/唯一组合索引 | 高 | 低 | 插入类操作（订单创建、注册、会议室抢订） |
| 乐观锁 | 高 | 低 | 读多写少（余额更新） |
| 状态机 | 中 | 中 | 有状态流转（订单生命周期） |
| 分布式锁 | 低 | 高 | 强一致性（秒杀库存） |

## 04
架构 — 思维
## 真实业务案例

**场景**：
双十一秒杀活动，10万人同时抢购100台手机

**方案组合**：
**1. Token机制**：用户点击“抢购”时先获取Token，防止重复提交

**2. 分布式锁**：以商品ID为锁键，保证库存扣减原子性

**3. 状态机**：订单状态只能从“待支付”→“已支付”转移

**效果**：

- QPS提升30%
- 零超卖记录
- 可用性提升到5个9以上，用户投诉率下降85%

## 05
架构 — 思维

## 最佳实践

**1. 优先Token机制**：简单高效，适合80%的写入场景

**2. 组合拳更稳**：高价值业务（如金融支付）建议“唯一ID + 乐观锁”双重保障

**3. 监控是关键**：记录幂等拦截日志，定期分析重复请求类型，逐一治理



END

**推荐阅读**

Redis系列：完整的Redis进阶之路

MySQL系列：MySQL核心技术点剖析

MQ系列：超高并发下的流控神器

微服务系列：互联网大规模分布式服务的技术进阶

架构设计：培养良好思维，设计优质架构，大厂架构决策分享

幂等性 1 高可用 15 分布式锁 5 秒杀 4 电商 4