Redis面试项目回答

URL动态加密

如何设计一个分布式锁?

如何设计一个分布式锁?

抢红包

下面是前言：

使用的就是redis的list集合，然后这里有一个意外情况就是：

redis如果当前用户取出数据成功了，但是此时redis宕机了，然后aof同步失败，那么下一次恢复了后，就会拿到这条没有出队的数据，数据就出错了。

但是这个线程他是成功的，所以它可以成功的执行对数据库的操作。

但是redis恢复之后，后面的线程都会出问题。

所以我们必须得保证就是当前线程拿到的这个积分位置是对的。

这也就是一个很正常的秒杀场景。

就算用Lua也没用。

其实只是其中一台宕机，为了保证数据安全，我们可以用红锁，也就是集群中1/2以上的节点数据操作成功。

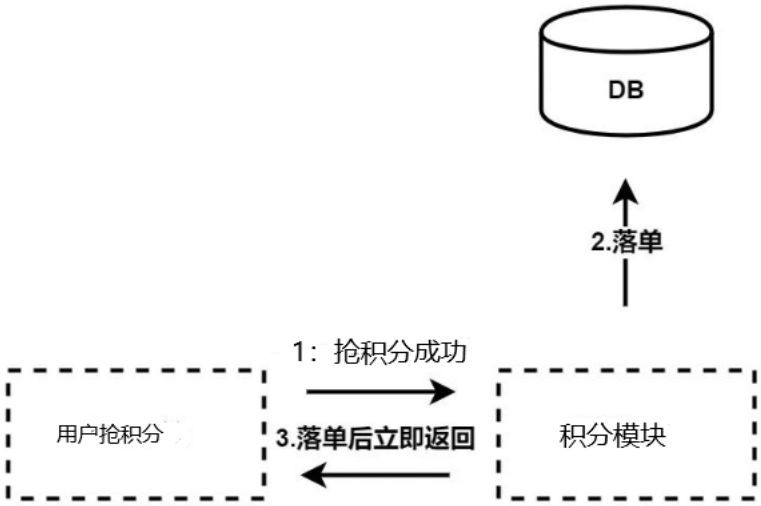
环境以及背景交代：

我们公司的Redis集群的配置为20台实例，每台实例40C256GB内存大小。

在有资格抢红包的用户中，数量大约为100w，所以我们的表中其实就只有100w的数据量，而对于这种数据量，我们的数据库的各种操作其实压力并不大。因为我们的数据库的架构也是高可用的主从。这里我们的数据库的TPS可以达到8000，QPS大概可以达到25000。

那么压力其实就来到了瞬时并发比较大的Redis这边了。所以并发抢积分这一块我们需要保证我们的业务尽可能的高可用。因为用户一般抢到积分之后马上就会去查看自己的积分信息了。

并且考虑到我们的积分发放之后，其实用户并不会马上去使用这些积分，只需要保证用户得到积分后到实际感知到积分增加的时间延迟可控，那么用户体验就不会受到影响，因此我们采用了异步增加积分的模式，积分模块收到上游发放积分请求后，落单后立即返回，通知上游受理成功。



红包存放

对于红包的存放，其实有两种方式，第一种是使用本地内存的方式，第二种是使用Redis的方式。

现在来介绍一下我们当初对这两种方式的思考。

使用本地内存

先说说优点：

1：实现简单，直接把积分信息保存在本地内存中即可，不需要额外操作。

2：没有网络操作开销，速度快。

3：便于管理。

缺点：

1：由于项目是集群部署的，需要考虑能确保用户抢到的是同一个积分红包中的积分，也就是需要额外的方式去确保同一个群中的用户的请求可以路由到同一台机器上。

使用Redis

这里需要考虑到一个个小红包放入到Redis中是成功的，原子性的。

并且为了确保减少不必要的索引次数，我在Cluster集群中使用了前缀key，来确保当前红包的操作都会路由到同一个Redis服务器上。
为了使 Redis 集群流量均匀，不同的积分红包数据被打散到了不同的 Redis 分片上。

这里我们有如下几个方式可以实现这个红包存放：

1. 使用 lpush的方式，一次性放入多个元素

优点

- **简单明了：** 代码逻辑容易理解和实现。
- **容易调试：** 出现问题时，容易定位和解决。
- **批量操作性能低：** 因为每次 **lpush** 都需要与 Redis 服务器进行一次通信，RTT（Round Trip Time）会极大地影响性能。如果使用for循环的方式的方式调用lpush会出问题，但是这里是直接lpush一堆数据进去，因此么有这个问题。
- **原子操作：** 如果在循环过程中出现问题（比如网络问题、服务器崩溃等），可能会导致数据不一致。由于lpush的操作本身是原子性的，所以不用担心。

2. 使用 Pipelining 的方式

优点

- **性能优化：** 通过一次网络调用发送多个命令，降低了网络延迟。
- **减少CPU使用率：** 由于减少了网络请求次数，相应地也减少了CPU的使用。

缺点

- **非原子操作：** 虽然性能有所提升，但 pipelining 本身不保证原子性。
- **代码复杂性：** 相对于简单的 **for** 循环，需要更复杂的编程模型。

3. 使用 Redis 事务的方式

优点

- **原子性：** 通过 **MULTI** 和 **EXEC**，保证了所有命令的原子性。
- **性能相对优化：** 尽管每个命令仍然需要进入队列，但事务的提交是一次性的。

缺点

- **复杂性：** 需要处理事务的开始、提交和可能的错误情况。
- **不支持回滚：** 如果事务中的某个命令失败，已执行的命令不会回滚。

4. 使用 Lua 脚本

优点

- **原子性：** Lua 脚本在 Redis 服务器中运行，确保了一系列命令的原子性。
- **性能最优：** 一次网络调用即可完成多个操作。
- **业务逻辑封装：** 可以在 Lua 脚本中封装更复杂的业务逻辑。

缺点

- **复杂性：** 需要编写和维护 Lua 脚本。
- **调试困难：** 如果 Lua 脚本有问题，可能比较难以调试。

Lua 脚本有几个优点：

1. **原子性**: 如上所述，Lua 脚本在 Redis 中是原子性执行的。
2. **减少网络开销**: 如果你需要执行一系列操作，使用 Lua 脚本会减少客户端和服务器之间的往返次数。
3. **简化客户端逻辑**: 使用 Lua 脚本可以将一些逻辑转移到数据库层，使得客户端代码更简单。
4. **更强的一致性保证**: 由于 Lua 脚本是原子性执行的，因此它们提供了一种有效的方式来处理复杂的、需要多步骤的操作，而不会因并发操作而出现数据不一致的问题。
5. **服务器端计算**: 对于一些计算密集型任务，Lua 脚本允许你在服务器端进行计算，而不是在客户端进行，这样可以减轻客户端的负担。

使用 Lua 脚本的一个潜在缺点是，由于脚本是阻塞性执行的，一个长时间运行的脚本可能会阻塞其他所有操作。因此，务必确保你的 Lua 脚本尽可能地快速和高效。

为了保证红包存放的安全性，由于我已经把当个红包的信息全都路由到了某一台机器上，所以其他机器是没有这个红包的备份信息的，所以我们的节点采用的是Cluster集群模式并且采用的是主从节点。

这样子主机宕机了也会故障切换到从机，就不会出大问题了。

这里我们为了不因为这个积分业务去侵入其他的业务，也就是因为这个业务而修改路由规则，我们选择了使用Redis这种方式。

因此，存放红包的过程可以使用lpush命令直接执行，而不需要使用我们的Lua脚本。

备份到MySQL中

（当然具体实现会更加复杂一点，这里可以简单先看一下，因为你还得考虑同步过程以及备份过程都是存在延迟的，依旧会存在数据不一致，你可以思考一下是否需要考虑这个问题）

我们会将这拆分好的红包，批量的放入到MySQL中，之后，如果某个用户获取到了某个红包之后，我们就会修改这个数据库的信息，比如修改当前这个红包信息的userid为当前用户，这样子就能在Redis出现宕机的时候，通过操作数据库的方式也能得到当前的一个红包的。

之后redis恢复之后可以通过MySQL中的当前消费到的一个红包的状态，来同步到Redis。

而Redis也可以通过配合RocetMQ的方式来同步mysql。

红包获取

上面我们已经保证了红包的存放过程是基本没问题的了，那么接下来的过程就是红包的获取了。

我们知道一个红包肯定只能获取一次，因此当一个用户开始抢红包的时候，并且抢到红包之后，我们可以在redis中为他生成一个token来表示当前用户已经抢过红包，不允许下次再抢了，这个逻辑很简单。但是也有如下的需要考虑的地方：

原子性：生成token和判断用户是否已经抢过红包需要在一个原子操作中完成，以防止同一用户多次抢红包。这里也可以考虑使用Lua脚本。

对于我的抢红包场景，有几种其他方法可以考虑，以提高性能：

1. **预先分配红包**: 在红包被创建时，预先将其分配给各个用户，并在用户实际领取时进行确认。这样，你只需要一个简单的 GET 操作就能完成整个过程。
2. **后处理**: 允许用户“抢”红包，但不立即完成交易。然后在后台进行批处理，确认哪些用户实际上有资格领取红包。
3. **延迟确认**: 在用户抢到红包后，先将其放入一个“待确认”队列。后台服务负责从该队列中取出红包，并进行最终确认和处理。

预分配红包

预先分配红包: 在红包被创建时，预先将其分配给各个用户，并在用户实际领取时进行确认。这样，你只需要一个简单的 GET 操作就能完成整个过程。

预先分配红包通常意味着在活动开始之前，根据一些特定的规则或算法，将红包或红包的"资格"预先分配给一部分预定的用户。这样做的一个主要目的是为了减轻高并发下对后端系统（比如数据库或缓存系统）的压力。

然而，预先分配确实一些问题：如果预分配的用户没有实际参与抢红包，那么这些红包或红包资格如何处理？

1. **超时机制**: 可以设定一个合理的时间窗口，在这个时间内，如果预分配的用户没有抢到红包，

那么这些红包可以重新进入"公共池"，供其他用户抢夺。

2. **动态调整**: 预分配只是一个"优先权"而非"确定权"。也就是说，预分配用户在规定时间内有优先抢红包的权利，但过了这个时间，红包即可供其他用户抢夺。
3. **分阶段处理**: 首先给预分配的用户一个优先抢红包的时间窗口，如果他们在这个时间内没有抢到，那么进入第二阶段，允许所有用户都能抢这些红包。
4. **混合模式**: 一部分红包预先分配，一部分随机分配。这样即使预分配的用户没有参与，也不会影响整体的红包分发。
5. **"懒"预分配**: 仅在用户实际点击抢红包时才进行预分配，这样可以避免预分配但未参与的问题，但这样做可能无法完全避免高并发带来的压力。

下面是一段预分配红包的代码

```
@Service
public class RedPacketService {

    @Autowired
    private RedisTemplate<String, Object> redisTemplate;

    // 生成红包并放入Redis
    public void generateRedPackets(int totalAmount, int numPackets, String redPacketId) {
        List<Integer> redPacketList = splitRedPackets(totalAmount, numPackets);
        redisTemplate.opsForList().leftPushAll("red_packet:" + redPacketId, redPacketList);
    }

    // 将红包预分配给指定用户
    public void preallocateRedPacket(String redPacketId, String userId) {
        redisTemplate.opsForValue().set("preallocated_red_packet:" + userId, redPacketId, 10, TimeUnit.MINUTES);
    }

    // 从Redis中抢红包
    public Integer grabRedPacket(String redPacketId, String userId) {
        // 检查用户是否有预分配的红包
        String preallocatedId = (String) redisTemplate.opsForValue().get("preallocated_red_packet:" + userId);
        if (preallocatedId != null && preallocatedId.equals(redPacketId)) {
            // 这里假设抢到红包的逻辑是从Redis List的左侧弹出一个红包
            Integer amount = (Integer) redisTemplate.opsForList().leftPop("red_packet:" + redPacketId);
            return amount;  // 返回抢到的红包金额
        }
        // 没有预分配或预分配的红包不匹配
        return null;
    }

    // 其他方法，例如splitRedPackets()用于拆分红包金额
}

```

之后我们就是要考虑如何对预分配的用户进行预分配操作了，这里我是用的是Redission的读写锁。

我可以让没有预分配的用户抢红包的时候只拿读锁，此时不会阻塞，如果出现了当前用户是预分配的用户，那么我给他加一个写锁，那么此时其他用户就会阻塞，只有当前预分配用户抢完红包之后才可以结束阻塞。

1. **对于预分配的用户**：在他们准备抢红包时，获取写锁。这将会阻塞所有其他没有写锁的用户（无论是否预分配）。只有当前预分配的用户完成了红包的抢夺逻辑后，才会释放写锁。

```
javaCopy code
// Java pseudocode
RReadWriteLock rwlock = redisson.getReadWriteLock("redPacketLock");
RLock writeLock = rwlock.writeLock();
try {
    writeLock.lock();
    // 抢红包逻辑
} finally {
    writeLock.unlock();
}
```

2. **对于非预分配的用户**：在他们准备抢红包时，获取读锁。如果没有任何写锁（即没有预分配的用户在抢红包），这将不会被阻塞。

```
javaCopy code
// Java pseudocode
RReadWriteLock rwlock = redisson.getReadWriteLock("redPacketLock");
RLock readLock = rwlock.readLock();
try {
    readLock.lock();
    // 抢红包逻辑
} finally {
    readLock.unlock();
}
```

这种方式确实可以实现你所描述的并发控制。需要注意的是，使用读写锁会增加系统的复杂性和潜在的性能开销，特别是在高并发场景下。因此，在使用这种方案之前，最好先进行充分的性能和压力测试。

评估了一下预分配红包的优点，如下：

预分配红包也有一系列潜在的优点：

1. **用户体验提升**：对于被预分配红包的用户，因为他们的红包已经预先保留，所以可以提供更流畅和

快速的体验。

2. **降低服务器压力**：通过预分配，部分工作可以提前完成，从而在红包开抢的高峰期减轻服务器压力。
3. **简化并发控制**：因为一部分红包已经预先分配给特定用户，所以在实际的抢红包操作中，系统可能需要处理更少的并发请求。
4. **提高成功率**：对于预分配用户，由于红包已经为他们保留，因此抢红包的成功率会相对提高。
5. **营销和用户粘性**：通过预分配，可以为某些特定群体或者忠实用户提供特权，这有助于提高用户粘性和进行更精细化的营销。
6. **灵活的业务策略**：预分配机制允许业务方根据用户行为、历史信息或者其他业务逻辑来动态调整红包分配，增加了业务操作的灵活性。
7. **减少冲突和竞争**：预分配可以减少用户间抢同一个红包的冲突和竞争，可能会减少因并发导致的错误或问题。
8. **提前验证用户身份**：在预分配阶段，还有机会进行额外的身份验证或者安全检查，这可以作为一个额外的安全层。
9. **可用于数据分析**：通过观察哪些用户更可能参与预分配，或者预分配成功后的用户行为，可以收集到有用的数据用于进一步的分析和决策。
10. **有利于资源规划**：了解预分配的数量和特点有助于更准确地进行资源规划，如缓存、数据库和服务器等。

当然这个方案我当初评估了很久，缺点如下：

1. **复杂性增加**：预分配逻辑会增加整体系统的复杂性，包括但不限于锁管理、超时处理和一致性维护等。
2. **资源占用**：预分配意味着你需要额外的数据结构和存储来追踪哪些红包已经被预分配。这可能会增加数据库或缓存的负载。
3. **响应延迟**：如果使用了锁机制（如读写锁）来保证预分配用户有优先权，这可能导致其他用户在高并发场景下遭遇一定的延时。
4. **预分配的无效性**：如果预分配的用户没有按时来领取红包，那么预分配就可能变得无效，需要额外的逻辑来处理这种情况。
5. **不公平性**：预分配策略可能引发一些公平性问题。如果预分配的用户最终没有参与，而其他非预分配用户由于某种原因（例如锁）不能立即访问，这可能会被认为是不公平的。
6. **调试和监控困难**：由于预分配引入了额外的逻辑和状态，这可能会使得系统更难以调试和监控。
7. **优先级管理**：如果预分配用户很多，还需要考虑他们之间的优先级，这也是一个复杂性增加的点。

8. **潜在的性能问题**：使用锁或其他并发控制机制可能会导致性能下降，特别是在高并发的环境中。
9. **预分配和实际行为不符**：用户的行为是不可预测的，即使预分配了红包，也不能保证用户一定会来领。
10. **滥用风险**：如果用户知道了预分配的逻辑，可能会尝试滥用这一特权。

后处理

允许用户“抢”红包，但不立即完成交易。然后在后台进行批处理，确认哪些用户实际上有资格领取红包。这种方案其实类似于抽奖了，也并不是说不行。

后处理（post-processing）策略的核心思想是：先让所有用户的请求通过，即先“接单”，然后在后台异步地处理这些请求。这种做法与实时处理用户请求相反，实时处理需要立即完成所有的业务逻辑，包括数据验证、业务规则应用、数据写入等。

后处理主要有以下几个特点：

1. **高并发接受能力**：由于前端只是接收请求而不进行处理，因此系统可以在短时间内接收大量的请求。
2. **异步处理**：具体的业务逻辑会在后台异步执行，这通常会用批处理、队列、定时任务等方式来实现。
3. **复杂性隐藏**：用户不需要等待所有的业务逻辑都执行完成，这些复杂的逻辑被移到了后台。
4. **容错能力**：即使部分后台处理任务失败，也不会直接影响用户的请求。这给了系统更多的机会进行重试或者人工干预。

在红包抢夺的场景下，后处理可以这样工作：

1. 当用户尝试抢红包时，系统仅记录下用户的请求，通常是将用户ID和红包ID放入一个待处理队列。
2. 一个后台任务（可以是定时任务或者是一个持续运行的服务）负责从队列中取出待处理的抢红包请求，并进行实际的业务处理。比如，判断红包是否还有剩余，是否已经被这个用户抢过等。
3. 最终的抢红包结果（成功或失败）会被记录下来，用户可以通过某种方式（比如查询API或者是推送通知）来获取自己是否成功抢到红包。

这种方式允许系统在短时间内接收大量的抢红包请求，而具体的业务逻辑处理则可以在后台慢慢进行。这样既提高了系统的吞吐量，也简化了前端逻辑。缺点是增加了系统复杂性，并且如果后台处理不及时，可能会影响用户体验。

延迟确认

在用户抢到红包后，先将其放入一个“待确认”队列。后台服务负责从该队列中取出红包，并进行最终确认和处理。这个方案和后处理差不多。

延迟确认是一种用于应对高并发场景的策略，它可以有效地降低实时处理的压力。在这种方案中，用户在前端“抢”到红包后，系统先不立即进行最终的确认和处理，而是将相关信息放入一个“待确认”队列中。后台服务会在稍后从这个队列中取出信息进行处理。

这样做有几个好处：

1. **解耦**: 将红包的“抢取”和“确认”逻辑解耦，可以让前端和后端服务各自独立地进行优化。
2. **缓解压力**: 在高峰期，后端服务可以根据实际情况控制从“待确认”队列中取出信息的速度，从而避免因高并发导致的系统压力。
3. **容错和重试**: 如果后台服务在处理某个红包时出现了问题（比如数据库暂时不可用等），可以将这个红包重新放入队列，稍后再试。

然而，这种方案也有一些缺点：

1. **用户体验**: 用户在“抢”到红包后需要等待后台服务的最终确认，这可能会影响用户体验。
2. **数据一致性**: 因为红包的最终确认是异步进行的，如果系统在这个过程中出现问题（比如后台服务宕机等），可能会导致数据不一致。

上面说了这么多，其实很明显，每个方案都有优缺点，权衡性能，实现难度等，我们否了所有上述方案，就直接用lpop这种方式取出红包哈哈哈。

第一种预分配虽然好，但是实现起来复杂，还会耦合我们的其他的业务，所以我们没选。

那么回到开头说的，如果出现了lpop命令执行成功，但是写入日志aof的时候失败了怎么办?

因为我们线上的redis肯定都是选择RDB和AOF两种备份方式一起使用的，RDB在这里明显不可靠。

所以我们得依靠实时性更好的AOF，但是即使你选择牺牲性能的always方案来保证每次提交操作之后都会马上写入日志，也有可能出现数据丢失。那么接下来我们就重点解决一下这个问题吧。

对于数据存储的可靠性，依赖单一存储引擎（如Redis）往往是有风险的，尤其是在高并发、高可用的分布式系统中。AOF（Append Only File）是Redis用于持久化的一种方式，其虽然提供了多种fsync选项（如**always**、**everysec**、**no**）用于权衡性能与数据安全，但还是有数据丢失的风险。

最终方案

其实对于上面的问题，我们可以想到的最基本的解决方案如下，我们先列出来：

1. **双写策略**：除了写入Redis外，也同时写入一个更为可靠的存储（比如关系数据库）。这样即使Redis的持久化出现问题，也可以从这个可靠的存储中恢复。
2. **消息队列**：在业务逻辑和数据持久化之间引入消息队列。业务逻辑只负责将操作写入消息队列，由单独的服务负责读取队列并更新Redis和其他存储。这样即使某个环节失败，也可以通过消息队列进行重试。
3. **数据校验和重试机制**：在写入AOF后，可以有一个异步的校验过程确保数据已经持久化。如果校验失败，则进行重试。
4. **高可用架构**：使用主从复制和哨兵模式来提高Redis的可用性。这样即使主节点出现问题，从节点还可以继续提供服务。
5. **事务性操作与备份**：利用Redis的事务特性（虽然不是ACID的全功能事务）和定期备份来降低数据丢失的风险。
6. **应用层补偿机制**：应用层记录关键操作，如果确认数据丢失，利用这些记录进行补偿。
7. **分布式事务或两阶段提交**：在涉及多个存储或服务时，使用更为复杂的事务机制来确保数据一致性。
8. **监控与报警**：通过实时监控Redis和AOF的状态，一旦发现异常立即触发报警。

其实上面这些东西我们基本都能保证，但是其实考虑的就是最最最坏的情况，就是他真的宕机了怎么办?

所以我就考虑到了redis-check-aof 。这是我们最最最后出现了宕机这种情况的时候的数据恢复方法。

因为如果真的出现了由于宕机导致出现的数据记录失败问题，此时RocketMQ消息也发送到MySQL那边成功了，其实我们可以考虑使用MySQL来在崩溃的时候进行数据同步。

redis-check-aof 是一个专门用于检查和修复 AOF（Append Only File）文件的工具，属于 Redis 发行版的一部分。AOF 文件用于 Redis 的持久化，它包含一个完整的事务日志，可用于在系统崩溃后重建数据库状态。然而，在某些情况下，由于突然的宕机或其他意外情况，AOF 文件可能会损坏。

这里是 **redis-check-aof** 可以为你做的一些具体事项：

1. 文件检查

该工具会扫描 AOF 文件，识别任何不一致或损坏的命令序列。这是一种预防性措施，用于检测是否存在潜在问题。

2. 文件修复

如果 **redis-check-aof** 发现了问题，它会尝试修复这些问题。具体来说，它会生成一个新的、修复后的 AOF 文件，其中删除了损坏或不一致的命令。

3. 数据恢复

修复后的 AOF 文件可以用于启动一个新的 Redis 实例，从而尽可能地恢复到故障发生前的状态。请注意，这可能意味着一些最近的事务数据将丢失，但大部分数据应该是安全的。

使用场景

- **预防性维护**: 你可以定期运行 **redis-check-aof** 来检查 AOF 文件的完整性，特别是在做重要更改或更新之前。
- **紧急恢复**: 如果 Redis 实例因为 AOF 文件损坏而无法启动，该工具可用作紧急恢复手段。

在高可用和高冗余的设置中，**redis-check-aof** 更多地是一个“最后一道防线”。即使你的架构非常健壮，有时也难以避免硬件故障或其他不可预见的问题。在这种情况下，拥有一个能快速恢复数据的工具是非常有价值的。

需要注意的是，使用 **redis-check-aof** 是一个有损过程，可能会导致一部分数据丢失。因此，它通常应作为最后的恢复选项，而不是首选解决方案。对于极端重要的数据，最好还是有多重备份和恢复策略。

消息发送

完成了红包的获取，那么接下来就是发送RocketMQ消息到消费者那边去进行处理来保存到数据库中了。

在这个过程中，我们需要保证的就是这个消息的可靠性。

对于方案我们有如下几种：

为了确保这个过程中RocketMQ的可靠性，我思考了如下几个点：

1. **同步发送（Synchronous Send）**：最基础的方式就是采用同步发送，等待服务器返回确认。这种方式最直观，但可能会影响到服务的吞吐量。
2. **异步发送（Asynchronous Send）**：为了提高性能，可以采用异步发送。但这种方式需要一个回调机制来处理消息确认或者失败。
3. **发送重试（Retry Mechanism）**：为了应对网络抖动或是短暂的服务不可用，RocketMQ 客户端通常会内置重试机制，你可以设置重试次数和间隔。
4. **事务消息（Transactional Messages）**：如果消息发送和其他资源（比如数据库更新）需要在一个事务中，RocketMQ 支持事务消息。这个已经在前面的回答中详细描述了。

5. **消息持久化（Message Durability）**：确保 RocketMQ 服务器配置了持久化存储，即使宕机也不会丢失消息。
6. **生产者流控（Producer Flow Control）**：通过合理设置生产者端的流控，防止因为生产消息速度太快而导致消息发送失败。
7. **集群容错（Cluster Fault Tolerance）**：在 RocketMQ 集群中，多个 Broker 可以共同工作以提供更高的可用性。
8. **消息确认（Ack）**：在高可靠性要求的场景中，可以等待 Broker 的确认消息，确认消息已经写入磁盘或者已经复制到 Slave。
9. **DLQ（死信队列）**：对于无法成功处理的消息，将其转移到一个专门的死信队列，以便后续分析和处理。
10. **消费者幂等性**: 确保消费者处理消息是幂等的，即重复处理同一消息不会产生不同效果。这通常需要业务侧的支持。

事务消息

这里我们一开始考虑的是，既然后面的消息是否成功被处理，都得依靠RocketMQ的消息是否成功发出，所以一开始我考虑的是使用RocketMQ的事务消息。

在 **pom.xml** 文件中：

Plain Text

```
xmlCopy code
<dependencies>
    <!-- Spring Boot starter -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
    <!-- RocketMQ Spring Boot starter -->
    <dependency>
        <groupId>org.apache.rocketmq</groupId>
        <artifactId>rocketmq-spring-boot-starter</artifactId>
        <version>2.1.0</version>
    </dependency>
    <!-- 其他依赖，比如 JPA，数据库等 -->
</dependencies>
```

然后在你的 Spring Boot 应用中：

1. **定义一个消息发送服务**：

Plain Text

```
javaCopy code
import org.apache.rocketmq.spring.core.RocketMQTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
public class RedPacketService {

    @Autowired
    private RocketMQTemplate rocketMQTemplate;

    @Transactional
    public void grabRedPacket(String userId, String redPacketId) {
        // 抢红包逻辑
        // ...

        // 发送事务消息
        rocketMQTemplate.sendMessageInTransaction(
            "redPacketTransactionGroup",
            "RedPacketTopic",
            "RedPacket grabbed by " + userId,
            redPacketId
        );
    }
}
```

1. 创建一个 RocketMQ 事务监听器：

Plain Text

```
javaCopy code
import org.apache.rocketmq.client.producer.LocalTransactionState;
import org.apache.rocketmq.spring.annotation.RocketMQTransactionListener;
import org.apache.rocketmq.spring.core.RocketMQLocalTransactionListener;
import org.apache.rocketmq.spring.core.RocketMQLocalTransactionState;
import org.apache.rocketmq.common.message.MessageExt;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.transaction.annotation.Transactional;

@RocketMQTransactionListener(txProducerGroup = "redPacketTransactionGroup")
public class RedPacketTransactionListener implements RocketMQLocalTransactionListener {

    @Autowired
    private YourDatabaseService yourDatabaseService;

    @Override
    @Transactional
    public RocketMQLocalTransactionState executeLocalTransaction(Message msg, Object arg) {
        String redPacketId = (String) arg;

        try {
            // 本地事务逻辑，比如存储消息至数据库
            yourDatabaseService.saveRedPacketTransaction(redPacketId, msg);
            return RocketMQLocalTransactionState.COMMIT;
        } catch (Exception e) {
            return RocketMQLocalTransactionState.ROLLBACK;
        }
    }

    @Override
    public RocketMQLocalTransactionState checkLocalTransaction(MessageExt msg) {
        // 事务状态检查逻辑
        String redPacketId = msg.getKeys();
        boolean isRedPacketTransactionExist = yourDatabaseService.isRedPacketTransactionExist(redPacketId);

        if (isRedPacketTransactionExist) {
            return RocketMQLocalTransactionState.COMMIT;
        } else {
            return RocketMQLocalTransactionState.UNKNOWN;
```

```
            }
        }
    }
```

在这个示例中，**YourDatabaseService** 是一个用于执行数据库操作的服务。当发送事务消息时，RocketMQ 会首先调用 **executeLocalTransaction** 方法以尝试执行本地事务。之后，它会根据该方法的返回值来决定是否提交或回滚消息。

但是，事务消息有一个很大的问题在这个场景中！

在高并发场景下，直接在消息发送过程中进行数据库操作确实可能会引入性能瓶颈或其他问题。RocketMQ 的事务消息功能通常更适用于需要确保消息和本地事务同时成功或失败的场景，而不一定适用于高并发场景。

所以这里我的消息发送使用的就是一个同步消息，然后使用RocketMQ集群以及消息持久化这些机制保证消息的可靠性。

绿色通道-提升用户体验

异步发送积分的假设是用户从抢积分动作发生，到实际感知到积分增加的存在，中间是有一段时间缓冲的，但是用户有可能领到积分后直接进入积分钱包查看，如果此时异步发放积分还未完成，有可能会造成客诉。针对这种情况，我们与上游用户服务主端做了约定，当用户抢积分后短时间内进入用户积分钱包查看积分变动时，上游会再次调用积分查询接口去查询Redis，并增加绿色通道的标识，我们收到这个标识后会将异步积分变更为同步，优先为当前用户增加积分，保证用户体验。

这里的做法其实是由于我们在Redis中以及保存了用户抢到的积分信息以及用户的ID信息了，所以用户如果查询自己的积分的时候，我们的上游用户服务端会马上先去查询Redis的这个用户的积分信息，因为用户肯定是抢到了才回去看自己的积分信息，此时我们可以保证Redis中以及存储了这个用户抢到的积分信息了，之后上游用户服务会发送一个消息到我们的当前积分服务中，然后我们就会马上同步的去更新数据库信息，而非异步更新了。

这里你可能会认为同步提前更新数据库，那不是和我RocketMQ中发送的消息冲突了，其实是不用担心的，因为我们更新数据库的积分的时候，是存在有时间戳的，用户抢到积分的时间戳肯定小于用户查看积分的时间戳，所以此时我们的绿色通道修改的时间戳一定会大于RocetMQ消息中的时间戳，我们只需要判断这是一个旧消息就可以阻止重复更新。也就保证了幂等性。

红包雨

与传统的抢红包不同，抢红包出现的情况是红包可能马上就被抢完了，但是红包雨则不一样，红包雨要求只要红包雨还在持续，那么后来的抢红包的用户也要能得到红包。

所以红包雨的实现场景则多了一个“配额”的概念。

红包是通过很多场定时“活动”来发放红包的。每场活动里面能发放多少现金，能发放多少虚拟物品，发放的比例如何，这些都是配额数据。
更进一步，我们要做到精确控制现金和虚拟物品的发放速度，使得无论何时用户来参加活动，都有机会获得红包，而不是所有红包在前几分钟就被用户横扫一空
配额信息由配额管理工具负责检查和修改，每次修改都会生成新的 SeqNo。一旦配额 agent 发现 SeqNo 发生变化，则会更新本地共享内存。由于 agent 采用双 buffer 设计，所以更新完成前不会影响当前业务进程。

这里的双buffer设计指的是，新生成的配额信息会保存在另一个buffer中，如果第一个buffer中的东西还没有消费完毕，可以继续先消费第一个中的，当然也可以消费第二个的。
所以并不会导致说由于需要对原有的buffer配额池子进行修改，导致出现并发读写问题。提升了性能。

双缓存策略

对于读多写少的情况，比如地理位置信息的获取，这种情况，基本都是读，那么我们只需要优化读操作即可。

但是如果出现了少量的写，那么此时就会为了这个少量的写必须先并发控制，从而出现一段时间的性能障碍，那么为了解决这个缓存读的情况，我们可以使用双缓存。也就是全量备份两个缓存。一般情况下都使用第一个，当出现了写的情况的时候，先把数据写到第二个缓存中，直到第二个缓存写完了，才进行缓存切换，这样子就能做到无感。

百万级别数据量抽奖系统

首先从架构上起手，如果是数百级别的数据量，那么单体架构完全可以应付的过来，但是百万级别，很明显单体架构肯定是不行的，因此我们需要使用集群，将大流量分发到大量的集群节点上去，来减少对某一台机器的流量。

负载均衡

1：这里我们就可以引入nginx来做负载均衡，使请求代理到某一台机器上，比如使用轮询的策略，从而减少对单一机器的压力。

但是，数百万级别的用户可能造成数百万数千万的抽奖请求，那么使用nginx去代理到这些机器上，依旧会对机器造成巨大的压力，我们也不可能说请求数量大了就马上扩容集群，一是来不及，二是机器成本高，公司肯定都是希望节省开支的。

限流

2：可以考虑在gateway网关中整合sentinel，或者在nginx中记录ip拦截请求，都是可以的。

防止用户重复抽奖

重复抽奖和恶意脚本可以归在一起，同时几十万的用户可能发出几百万的请求。

所以我们得对请求进行限流过滤，首先是过滤，对于我们的抽奖系统，可能会出现恶意刷流量或者出现单一用户多次点击抽奖的情况，那么我们首先就需要先对这些请求进行过滤。比如使用ip记录的方式将某个重复请求的ip拉入黑名单，而对于重复抽奖的用户，我们可以把已经抽到奖的用户的信息放入到Redis中，之后访问Redis就可以拒绝当前用户的请求。

总之，对于这种情况，解决方法就是限流，减少瞬间高并发对于系统的危害。

拦截无效流量

无论是抽奖还是秒杀，奖品和商品都是有限的，所以后面涌入的大量请求其实都是无用的。

举个例子，假设100万人抽奖，就准备了100个红包，那么100万请求瞬间涌入，其实前1000个请求就把红包抢完了，后续的几十万请求就没必要让他再执行业务逻辑，直接暴力拦截返回抽奖结束就可以了。

同时前端在按钮置灰上也可以做一些文章。

那么思考一下如何才能知道奖品抽完了呢，也就是库存和订单之前的数据同步问题。

服务降级和服务熔断

当然，有了以上的方法依旧有可能会出现错误，所以我们还需要使用熔断和降级两种方式来保障服务安全。

首先是熔断，假设有100w个请求，但是我们整个服务也就负载50w，那么剩下50w的请求就没办法处理了，那怎么办？很明显啊，这些额外的服务只能进行等待对吧，但是我们一般的请求都有超

时时间，超过一段时间还没有返回就会自动超时返回，当系统中的超时请求达到一定数量，也就是失败的请求达到一定数量，就会触发熔断器，熔断器触发之后，我们可以选择提供降级服务，比如输出一条“请稍候再来”。

等到系统的错误率减小到一个值之后，熔断器可能会恢复，之后就可以选择关闭降级服务，从而继续提供正常的服务。

当然，也并不一定先熔断后降级，也可以先降级后熔断。

1. **先熔断再降级**：
    - 在某些情况下，当系统检测到错误率上升或服务超时等问题时，熔断器可能首先被触发，以防止进一步的错误和系统过载。
    - 随后，系统可能会执行服务降级，为用户提供降级的服务（例如显示“系统繁忙”或提供基本功能），以保持系统的基本可用性。
2. **先降级再熔断**：
    - 在其他情况下，系统可能首先执行服务降级，以应对外部依赖服务的问题或响应延迟。
    - 如果问题持续存在并且错误率继续上升，那么服务熔断可能随后被触发，以防止系统过载和进一步的错误。

数据同步

上面我们也提到了，再进行限流等操作之后，其实依旧是很快的系统中的红包就已经被抽完了，那么剩下的请求其实都基本无效了，但是系统中的红包的数量肯定是在数据库中的，我们不可能直接让这么大的请求去操作数据库，这可能操作性能更好的缓存Redis了，所以我们可以先把库存信息同步到Redis，然后先操作Redis的同时，异步的使用MQ去修改MySQL，当然这里还需要很多的缓存与数据库数据一致性策略来保证数据一致性，但是大致思想是没错的，就是当Redis中已经没有库存后，直接返回请求即可。
当然库存这一块的操作，如果多个线程同时读到了一个数据，那么这多个线程都认为还有库存，就会出现超卖问题了，所以我们得考虑这多个线程扣减库存的原子性。

所以我们可以考虑CAS+Lua脚本的方式。

因为Lua脚本可以保证原子性，我们可以把判断库存和扣减库存放在一起，原子性操作之后，只要一个线程扣减了库存，其他的线程都会返回失败，那么就可以让这些返回失败的线程在通过CAS的方式重新的去请求一次库存，这样子就能解决超卖问题了，只要超过了库存，就直接返回即可。

线程优化

对于线上环境，工作线程数量是一个至关重要的参数，需要根据自己的情况调节。

众所周知，对于进入Tomcat的每个请求，其实都会交给一个独立的工作线程来进行处理，那么Tomcat有多少线程，就决定了并发请求处理的能力。

但是这个线程数量是需要经过压测来进行判断的，因为每个线程都会处理一个请求，这个请求又需要访问数据库之类的外部系统，所以不是每个系统的参数都可以一样的，需要自己对系统进行压测。

但是给一个经验值的话，Tomcat的线程数量不宜过多。因为线程过多，普通服务器的CPU是扛不住的，反而会导致机器CPU负载过高，最终崩溃。

同时，Tomcat的线程数量也不宜太少，因为如果就100个线程，那么会导致无法充分利用Tomcat的线程资源和机器的CPU资源。

所以一般来说，Tomcat线程数量在200~500之间都是可以的，但是具体多少需要自己压测一下，不断的调节参数，看具体的CPU负载以及线程执行请求的一个效率。

在CPU负载尚可，以及请求执行性能正常的情况下，尽可能提高一些线程数量。

但是如果到一个临界值，发现机器负载过高，而且线程处理请求的速度开始下降，说明这台机扛不住这么多线程并发执行处理请求了，此时就不能继续上调线程数量了。

业务逻辑

现在该研究一下如何进行抽奖逻辑的设定了，即使我们的100w的请求已经拦截到只剩5w，这些请求去请求数据库，数据库依旧有可能顶不住，但是如果使用Redis，那么Redis的单机并发量都已经可以接受这些请求了，更何况生产环境是Redis集群，所以这里我们考虑使用Redis来做红包的具体抽奖逻辑。

并且Redis也提供了非常非常适合抽奖逻辑的Set结构，我们可以把红包拆分好之后放入到Set结构中，然后Set结构提供了一个随机的出队操作，我们就可以用这个操作来模拟随机抽奖了。

流量削峰

由上至下，还剩中奖通知部分没有优化。

思考这个问题：假设抽奖服务在5万请求中有1万请求抽中了奖品，那么势必会造成抽奖服务对礼品服务调用1万次。

那也要和抽奖服务同样处理吗?

其实并不用，因为发送通知不要求及时性，完全可以让一万个请求慢慢发送，这时就要用到消息中间件，进行限流削峰。

也就是说，抽奖服务把中奖信息发送到MQ，然后通知服务慢慢的从MQ中消费中奖消息，最终完成完礼品的发放，这也是我们会延迟一些收到中奖信息或者物流信息的原因。

假设两个通知服务实例每秒可以完成100个通知的发送，那么1万条消息也就是延迟100秒发放完毕罢了。

同样对MySQL的压力也会降低，那么数据库层面也是可以抗住的。

其他优化

CDN

CDN全称内容分发网络，是建立并覆盖在承载网之上，由分布在不同区域的边缘节点服务器群组成的分布式网络。通俗的讲，就是把经常访问又费时的资源放在你附近的服务器上。

淘宝的图片访问，有98%的流量都走了CDN缓存。只有2%会回源到源站，节省了大量的服务器资源。

但是，如果在用户访问高峰期，图片内容大批量发生变化，大量用户的访问就会穿透cdn，对源站造成巨大的压力。

所以，对于图片这种静态资源，尽可能都放入CDN。

URL动态加密

这说的是防止恶意访问，有些爬虫或者刷量脚本会造成大量的请求访问你的接口，你更加不知道他会传什么参数给你，所以我们定义接口时一定要多加验证，因为不止是你的朋友调你的接口，敌人也有可能。