27 | 解决一个互斥问题，系统并发用户数提升了10倍！

2021-07-17 尉刚强

《性能优化高手课》 课程介绍 >

讲述：尉刚强
时长 13:22 大小 12.25M

你好，我是尉刚强。

互斥锁是实现并发场景下业务操作原子性、解决互斥访问问题的有效手段之一，由于它的使用方式相对比较简单和安全，所以不论是在互联网的分布式系统中，还是在嵌入式并发场景下，应用都比较广泛。

但是，**如果互斥锁的选择和使用不当，就很可能成为系统的性能瓶颈之一**，所以合理优化互斥锁，就成为了系统性能优化的一个重要手段。

那么在今天的课程中，我就会给你分享一个互联网场景下使用互斥锁优化的案例，按照优化前的软件实现、性能瓶颈分析、优化解决方案的思路，带你剖析我是通过什么样的方法

优化业务中的互斥锁，以及是如何提升业务 RPS（Requests per second，请求吞吐量）性能指标 10 倍以上的，从而帮助你全面地了解分析与优化互斥锁的详细过程。

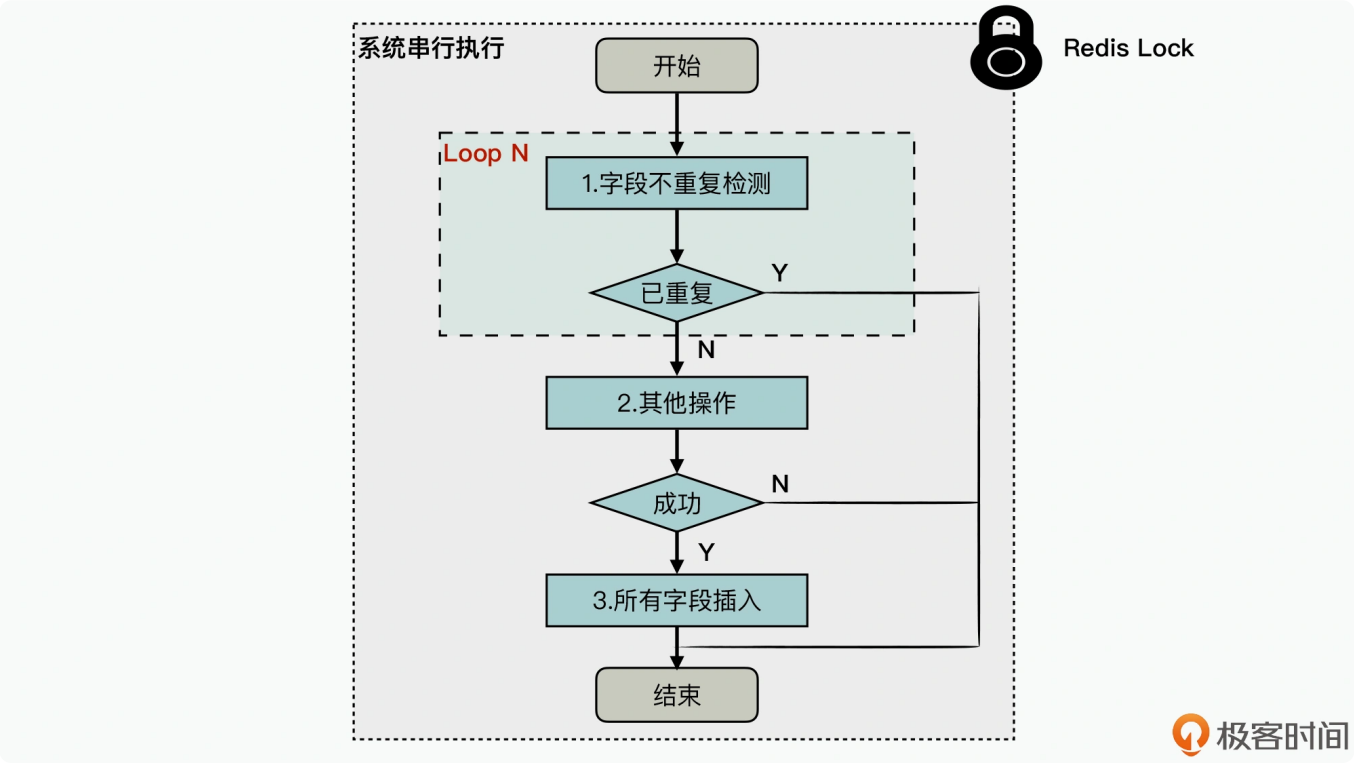
这样，通过学习这个性能优化案例，你在业务中就可以准确识别出哪些场景下的互斥锁可以优化掉，而哪些场景下不可以。并且你还会掌握一种手动实现事务的机制（支持业务操作回滚机制），来替代业务中互斥锁的手段，进一步来帮助优化提升软件的性能。

那接下来我们就先看看，在该案例中，业务优化前的实现是怎么样的，以及它都存在什么性能问题。

优化前的业务实现为什么会有性能问题？

这个性能优化案例的业务场景是这样的：用户给在线表单提交一条记录，在这条记录中会包含很多个字段内容，其中有些字段在插入时有一个规则要求，即不能与已有的字段值重复。

为了便于理解，这里我用一个图来描述下，原业务系统中实现字段插入值不重复规则的实现逻辑，具体如下所示：



可见，在该业务中使用了一个 Redis 锁来实现互斥访问，从而实现了被加锁的业务逻辑执行的原子性，所以这部分计算逻辑在系统中是串行执行的。而被加锁的业务逻辑主要有三

个关键操作，分别是：

1. **字段不重复检测**：检查插入的字段值在数据库中是否有重复的情况出现，如果出现重复的值，插入失败直接退出，否则执行下一步的操作。这里系统会遍历要求值不能重复的所有字段项，如果其中任何一个字段项出现值重复，就都会退出。
2. **其他操作**：即用户提交记录的过程中一些关键业务操作，其特点是不能被拆分执行，也不能被回滚。如果操作成功的话，就会执行下一步的操作，否则也会直接退出。
3. **所有字段插入**：由于这三个操作，通过加锁保证了原子性执行，所以前面检测的“字段值不重复”的条件仍然是有效的，这一步会将所有的字段进行插入。

除此之外，在优化前的代码实现中，需要进行重复性校验的字段都会记录在 Redis 中。所以，图中的操作 1、操作 3 都是基于 Redis 来实现的。

那么在看完这个业务实现逻辑图之后，你可能会比较好奇：**这种字段唯一性检测机制，为什么不使用关系数据库中的字段唯一性检测机制来实现呢？**这是一个好问题，我在刚看到这个业务逻辑实现的时候也很好奇，而到后来深入分析了业务之后，才理解为什么会这样实现。

其实这是因为，在这个业务系统中有大约 1000 万张表单，其中每张表单的字段唯一性规则可能都不一样，而且用户还可以随意修改这个规则。所以，这个系统在设计实现的时候，就将所有表单中的所有字段，都放到了一张很大的数据库表中，因此这样我们就没有办法使用数据库表上的字段唯一性规则，来处理这个问题了。

原来的这种 **Redis 加锁的实现方式**比较简单，而且是按照单个表单来进行加锁的，所以**在单个表单并发提交请求吞吐量不是很大的情况下，并不会对系统性能产生太大的影响**。

可问题就是，随着系统的业务规模逐渐增大，会出现少量表单的并发请求吞吐量暴增的情况，而这个时候，当单个表单提交请求超过了并发请求吞吐量的上限值后，就会引起两个比较严重的性能问题：

1. 针对超过并发请求吞吐量性能上限值的那个表单，用户在提交表单的页面会出现卡死，导致提交数据失败；

2. 由于后端服务系统是基于进程模型的，而进程资源的数目是有限的，所以一旦个别表单提交数据请求的处理进程被阻塞，占用了大量的进程资源，就会导致整个系统无法正常处理所有的业务请求。

因此，**提升单个表单提交请求吞吐量的性能指标，就成为了这个软件系统性能优化的关键问题**。那么接下来，我们就要先搞明白，这个互斥锁是如何影响这个表单的请求吞吐量性能的。

> 补充：在这个软件系统实现中，由于单个表单的吞吐量性能瓶颈，而导致的整个系统业务处理受阻，实际上是另外一个软件设计问题，所以不在我们今天的课程讲解范围内。

互斥锁是如何影响最大请求吞吐量的？

在第 22 讲中，我们已经学习过加锁的计算逻辑属于串行资源，这是系统中潜在的性能瓶颈之一，会影响系统的请求吞吐量的性能指标。

接下来，我就使用一个公式来描述下在这个案例中，使用了 Redis 互斥锁以后，来计算 Max RPS（最大请求吞吐量）的计算方法，具体公式如下所示：

$$\text{Max RPS} = \frac{1 \text{ second}}{\text{lock time} + \text{resource competition} + \text{Unlock time}}$$

```
if (redis.call("exists", KEYS[1]) == 0
and ARGV[3] == "yes") or
redis.call("get", KEYS[1]) == ARGV[1]
then
  return redis.call("set", KEYS[1],
ARGV[1], "PX", ARGV[2])
end
```

```
if redis.call("get",KEYS[1]) == ARGV[1] then
  return redis.call("del",KEYS[1])
else
  return 0
end
```

在这个公式中，因为 Lock 和 Unlock 是采用 Redis 的互斥锁来实现的，它们所使用的 Redis 的 script 脚本实现如图中所示，通过在真实系统中进行测量，其中 Lock time + Unlock time 的操作时间之和在 3ms 左右。

然后你就可以通过上面的公式计算出，如果中间加锁的计算逻辑（resource competition）执行开销为 30ms 左右，那么对应的 Max RPS=1s/(3ms+30ms)，也就是大约在 30RPS 左右。

也就是说，如果把加锁的计算逻辑降低极限值为 0 时，对应的 Max RPS 才可以到达 300RPS 左右。

那么这里你需要注意的是，因为业务中的互斥锁是全局控制的，所以当系统达到最大 RPS 时，即使通过弹性扩展机制部署再多的后端服务实例进程，也不能再提升这个性能指标了。

因此到这里，在这个性能优化案例中，我们经过测量加锁的计算逻辑执行时间为 30RPS，然后根据上面的公式，我们计算出的最大 RPS 值也为 30RPS 左右，这与真实的性能测试获取的性能指标值是完全一致的。

好的，现在问题就已经比较清楚了，那么有没有办法可以优化提升这个系统的性能呢？下面我们来看一下。

性能优化解决方案

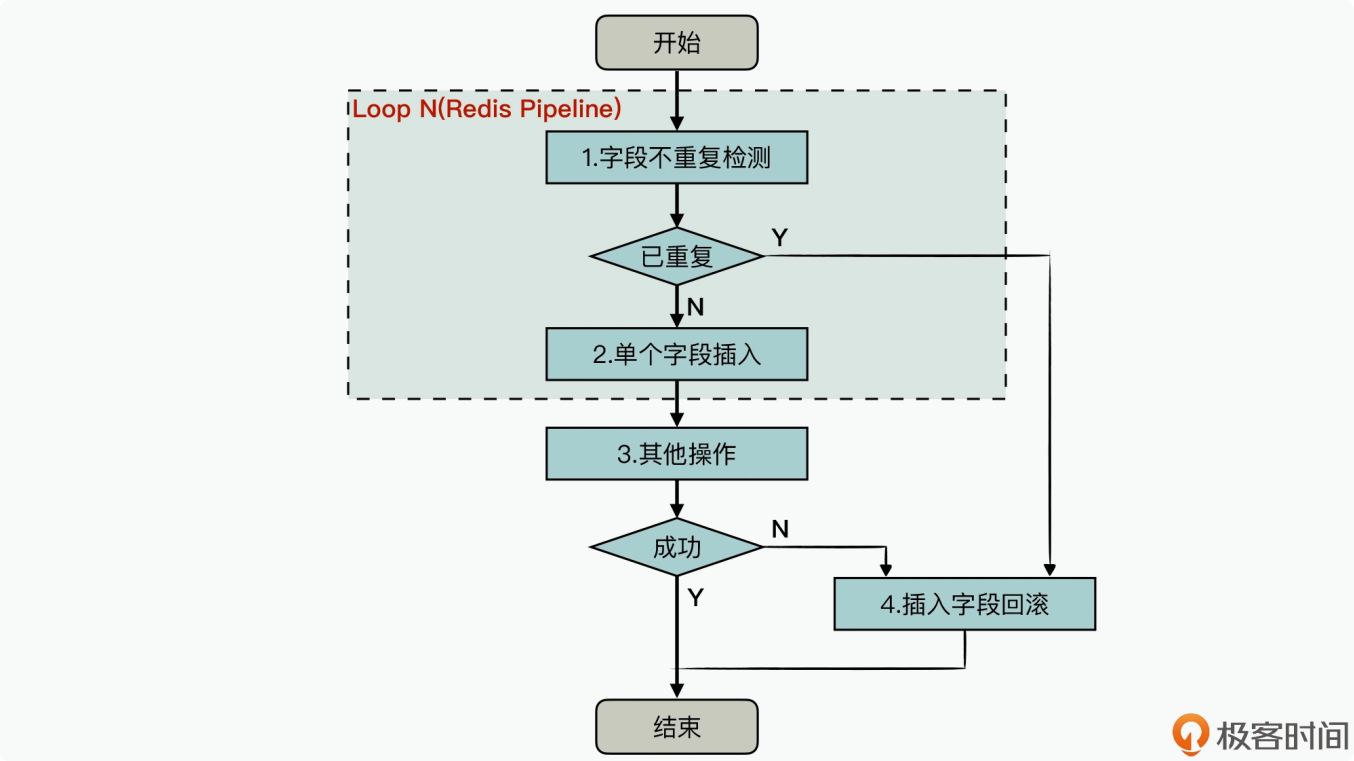
针对这个业务案例，你应该也能发现一个明显的特点：绝大多数情况下，并不会出现并发提交相同字段值的情况。也就是说，如果这个业务逻辑没有增加互斥锁，在 99.9% 的情况下业务逻辑也是正确的。所以，针对这种场景，我们就可以采用手动实现事务机制，来优化掉业务代码中的互斥锁，来提升请求吞吐量的性能。

而这里我们已经知道，在这个案例中，使用互斥锁解决的核心问题是**判断字段不重复**和**字段插入操作的原子性问题**。所以说，我们其实可以考虑采用一些优化机制，来单独实现这两个操作组合的原子性。

但是要注意，如果在互斥锁的使用场景中，被加锁的业务操作还有更复杂的一致性要求，比方说存在数据库写冲突的问题等，那么这种互斥锁实现就不能被简单地优化掉了。

那么对于这个案例中的互斥锁而言，我们应该怎样优化呢？

我来说说我想到的优化思路。这里呢，为了更清晰地描述该解决方案，我用了一个流程图来给你详细地介绍下性能优化后的具体实现过程，如下图所示。



也就是说，我们可以把“字段不重复检测”“单个字段插入”“其他操作”三个操作绑定到一起，然后实现一种事务机制的能力，支持在后面操作失败的情况下，可以回滚到前面的操作中。

实际上，原来的 Redis 互斥锁主要是为了实现“字段不重复检测”和“字段的插入操作”的原子性，而在手动实现事务机制之后，我们就可以把这两步操作放到开始处执行，然后**使用 Redis 的 Pipeline 机制**保证这两步操作组合的原子性，从而不会被其他 Redis 操作干扰到。

这样针对接下来的其他操作（也就是用户提交数据过程中的一些不可拆分的关键业务操作），如果操作成功，就提交任务成功结束；如果操作失败，则需要回滚之前的字段插入操作。另外为了实现事务的机制和能力，我们还需要在前面字段插入时，同时记录插入前的状态和插入后的变更状态，从而就可以实现失败后的回滚机制。

其实，这里我还考虑过另外两种实现方案，分别是基于 Redis 的事务机制和基于 MongoDB 上的事务机制。

但是，我最后在实现时并没有采纳，这背后其实有很多的原因。比方说，使用 MongoDB 的事务需要进行数据迁移，而且需要升级系统的 MongoDB 集群的数据库版本等，以及使用 Redis 事务机制的代码实现并不友好，等等。

不过这里有一个**最重要的原因**就是，不管是使用 Redis 事务还是 MongoDB 上的事务，它们都把对字段插入操作的冲突时间，拉长到了步骤 3“其他操作”结束之后，而这样就显著增大了事务冲突失败的概率。

所以最后，我们采用前面这种优化后的实现机制，因为去除了互斥锁，所以用户间的提交记录可以更大程度地并行。而且优化后的实现方式，只有 Pipeline 操作会排队处理，而由于单个 Pipeline 的执行时长在 1ms~3ms 之间，所以最后优化后的表单最大请求吞吐量，就从原来的 30RPS，提升到了 300RPS 左右，这样就**实现了性能提升超过 10 倍的目标**。

小结

同步互斥是高性能分布式系统设计实现的关键技术之一，而在软件开发的过程中，我们很容易因为设计或实现不高效，导致整个软件系统的性能不够理想。所以在今天的课堂上，我通过一个真实的性能优化案例，给你讲解了业务中使用互斥锁存在的性能问题，包括它对请求吞吐量的性能影响，以及优化互斥锁来提升系统性能的方法和过程。

但是，每个业务系统中的同步互斥问题都存在特有的复杂性，所以你应该学习收获到的是，如何分析同步互斥对性能的影响，以及如何优化同步互斥的实现来提升性能的方法。在今天所讲的案例中，采用的是先默认成功来执行操作，后面出现异常时再回滚的机制，你也可以在自己项目优化中去使用。

思考题

基于 Redis 上有 CAS 命令，有 Pipeline 机制，还支持 lua.script 和事务能力，那么在业务中你会怎么选择来优化软件的性能呢？欢迎给我留言，分享你的思考和看法。如果觉得有收获，也欢迎你把今天的内容分享给更多的朋友。

上一篇 26 | 一个嵌入式实时系统都要从哪些方面做好性能优化？

下一篇 28 | Web服务业务代码一行不动，性能提升20%，怎么做到的？

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。