



下载APP



28 | Web服务业务代码一行不动，性能提升20%，怎么做到的？

2021-07-20 尉刚强

《性能优化高手课》

课程介绍 >



讲述：尉刚强

时长 14:07 大小 12.94M



你好，我是尉刚强。

在软件开发的过程中，为了保持软件系统设计的简单性，一般情况下，我们会把业务操作实现成强一致性的，而且是实时生效的。但是，在设计与实现高性能软件系统的过程中，我们其实还可以通过降低一些非关键业务操作的一致性或实时性，来调整软件设计与实现，从而换取更大的性能收益。就比方说，我们经常使用的部分 Cache 技术，其背后的原理就是通过降低数据的一致性，来提升软件的执行速度。

那么今天这节课，我要分享的也是一个**通过降低业务操作的一致性和实时性，来换取性能提升的案例**。



我会按照“优化前性能分析”“优化解决方案”“优化成果分析”的顺序来进行讲解，并带你剖析在这个过程中我是如何思考问题，以及如何根据具体业务场景和软件实现现状进行权衡的，以此来让你可以更加清楚和明白，如何去分析不同的业务操作的一致性和实时性差异和影响范围，从而设计出更加适合业务场景的 Cache 技术解决方案，来优化提升软件的性能。

下面，我们先来了解下这个案例的背景，一起来分析下这个软件系统优化前的性能。

优化前性能分析

在互联网的 Web 服务中，A/B 测试作为一种数据驱动产品进行优化的科学方法，应用比较广泛。其中 A 代表原有实现方案，B 代表新的实现方案，然后通过显式地控制与调整使用方案 A 和方案 B 的用户占比，并获取观察分析数据，来评估新功能或者实现方案上线后是否有效，以及预期收益是否在合理的范围内。


由于 A/B 测试的机制原理是业务无关的，所以，很多编程语言中的一些第三方库已经实现了 A/B 测试的功能，方便我们使用。今天我介绍的 Web 服务性能优化项目中（基于 Ruby 语言开发的），也正是使用了一个 A/B 测试的库 [Split](#)。

首先，在这个 Web 服务性能优化项目中，会使用监控分析工具 NewRelic，来获取请求内部处理的跟踪信息。而我们通过分析跟踪获取信息后发现，有很多基于 Redis 的请求操作占用了比较多的处理时间，再进一步分析软件的代码实现后发现，这些 Redis 的请求操作主要都来自这个 A/B 测试的第三方库 Split。

因此，为了进一步地分析 A/B 测试的库 Split 对软件性能的影响，接下来我主要做了两件事情：

1. 学习 Github 上 Split 开源库文档中的设计思路，以及这个库的主要流程的源代码，目的是分析是否有 Split 库使用场景不当，导致引入性能问题；
2. 梳理出这个 Web 服务中一次 Split 的调用期间，所有的 Redis 的 API 请求的详细列表，这一步主要是为了寻找是否有冗余的 Redis 读取操作。

其中，针对 Redis 的 API 请求列表梳理结果如下：

 复制代码

```
1 GRedis.instance.exists "new_feature_swich"
2 GRedis.instance.type "new_feature_swich"
3 GRedis.instance.lrange "new_feature_swich",0, -1
4 GRedis.instance.lrange "new_feature_swich:goals", 0, -1
5 GRedis.instance.get "new_feature_swich:metadata",
6 GRedis.instance.hset "experiment_configurations/new_feature_swich", resettabl
7 GRedis.instance.hset "experiment_configurations/new_feature_swich", :algorithm
8 GRedis.instance.hkeys "split:530c9534d48164466d000216"
9
10 GRedis.instance.exists "new_feature_swich"
11 GRedis.instance.hgetall "experiment_configurations/new_feature_swich"
12 GRedis.instance.type "new_feature_swich"
13 GRedis.instance.lrange "new_feature_swich",0, -1
14 GRedis.instance.lrange "new_feature_swich:goals", 0, -1
15 GRedis.instance.get "new_feature_swich:metadata",
16
17 GRedis.instance.hget "experiment_winner", "new_feature_swich"
18 GRedis.instance.hget "experiment_start_times", "new_feature_swich"
19 GRedis.instance.hget "experiment_winner" "new_feature_swich"
20 GRedis.instance.get "new_feature_swich:version"
21 GRedis.instance.hkeys "split:5c6b6a1377fa104b6a427c8e"
22 GRedis.instance.hget "experiment_start_times", "new_feature_swich"
23 GRedis.instance.hget "split:530c9534d48164466d000216", "new_feature_swich:5"
24 GRedis.instance.hset "split:530c9534d48164466d000216", "new_feature_swich:5",
```

那么，如果仔细观察这些 Redis 的 API 请求列表，你就会发现，这个列表中有好些 API 请求是重复的，比如说：

 复制代码

```
1 GRedis.instance.exists "new_feature_swich"
2 GRedis.instance.type "new_feature_swich"
3 GRedis.instance.lrange "new_feature_swich",0, -1
4 GRedis.instance.get "new_feature_swich:metadata",
5 GRedis.instance.lrange "new_feature_swich:goals", 0, -1
```

这样在整体阅读完代码后，你就会明白，这个 A/B 测试的 Split 库的一次接口调用过程中，Split 内部的不同模块间重复读取了 Redis 中的同一条数据。但其实，这是完全没有必要的，因为这是一个很明显的低效率编码实现问题。

补充：其实我们的软件系统引入的第三方库，因为使用方式和场景不是最佳的，很有可能会导致运行性能比较差，从而也会直接影响到软件系统的性能。

再深入分析业务代码实现，我们发现这个 Web 服务中每个 REST 请求，平均会调用这个 A/B 测试的 Split 库的接口 1.5 次左右，而每个 Split 接口实现中大概会调用 Redis 接口的次数在 10~30 之间。

这里我们根据 Redis 的 API 平均处理时延为 0.5ms 左右，来进行粗略估算，就可以计算出 Web 服务的 REST 请求中，使用 A/B 测试接口占用的处理时间开销大约为： $1.5 \times (10+30) / 20.5 = 15\text{ms}$ 左右。又因为目前这个 Web 服务的 REST 请求平均处理时延在 120ms 左右，所以就可以预估出 A/B 测试占用开销为 $15\text{ms} / 120\text{ms} = 12\%$ 左右。

因此，如果可以优化掉这个执行开销，那么性能收益其实是相当可观的。

好了，现在，我们已经深入了解了 Split 的内部原理和代码实现，以及它对业务的性能影响，那接下来，我们就可以考虑怎么来优化下这里的软件性能了。

性能优化解决方案

不过，在确定优化解决方案之前，我们还需要分析下 A/B 测试功能在业务中的使用场景，这样才能更容易寻找到性能比较好的解决方案。

在这个 Web 服务中，A/B 测试属于常用功能，一般情况下是按照一周的时间维度来手动配置的。所以，在理论上你可以认为 A/B 测试的配置是比较稳定的，不容易频繁变化。但是，在业务实现中的每个 REST 请求中，都会去读取和更新 A/B 测试的配置，因此你可以认为，A/B 测试配置变更可以在 ms 级别后实时生效。

那么这里，我就有一个问题想问：**针对 A/B 测试的配置变更操作，是不是一定要在 ms 级实时生效呢？**

答案是，**不需要**。实际上，通过进一步分析 A/B 测试的业务使用场景，你会发现系统中的各个业务请求间都是相对独立的，如果获取到 A/B 测试的配置数据在短时间内不完全一致时，其实并不会影响到业务功能。

所以基于这个分析判断，我们可以得出一个结论：**适当地降低 A/B 测试配置生效的实时性和一致性，就可以给 Cache 技术留下比较大的空间来优化提升软件的性能。**

补充：可能会有极小的概率会出现，单个用户在很短时间内多个请求，页面的显示风格有些差异，但也是用户可以接受的。

那么，针对这个 A/B 测试的功能，现在的代码实现是**每个请求都会读取配置**。如果可以按照每个服务的进程，使用内存 Cache 手段来保存 A/B 开关配置，就可以优化掉绝大部分的 A/B 测试中，Redis 的 API 占用的时间开销。

而这里的内存 Cache 机制，我们可以采用**基于时间失效**（即 Cache 保存的数据超过一定时间后自动失效）这种比较简单的实现方式，在经过业务分析后，我们得知配置失效时间在 5s~10s 是可以接受的。

那么现在，针对该项目的性能优化思路就已经很清晰了，也就是我们可以考虑**怎么在代码中实现内存 Cache 机制，来保存 A/B 测试配置信息**。这里有两种方式可供选择。

方式 1：将 A/B 测试的配置的 Cache 机制直接实现在这个 A/B 测试库 split 的源码中。

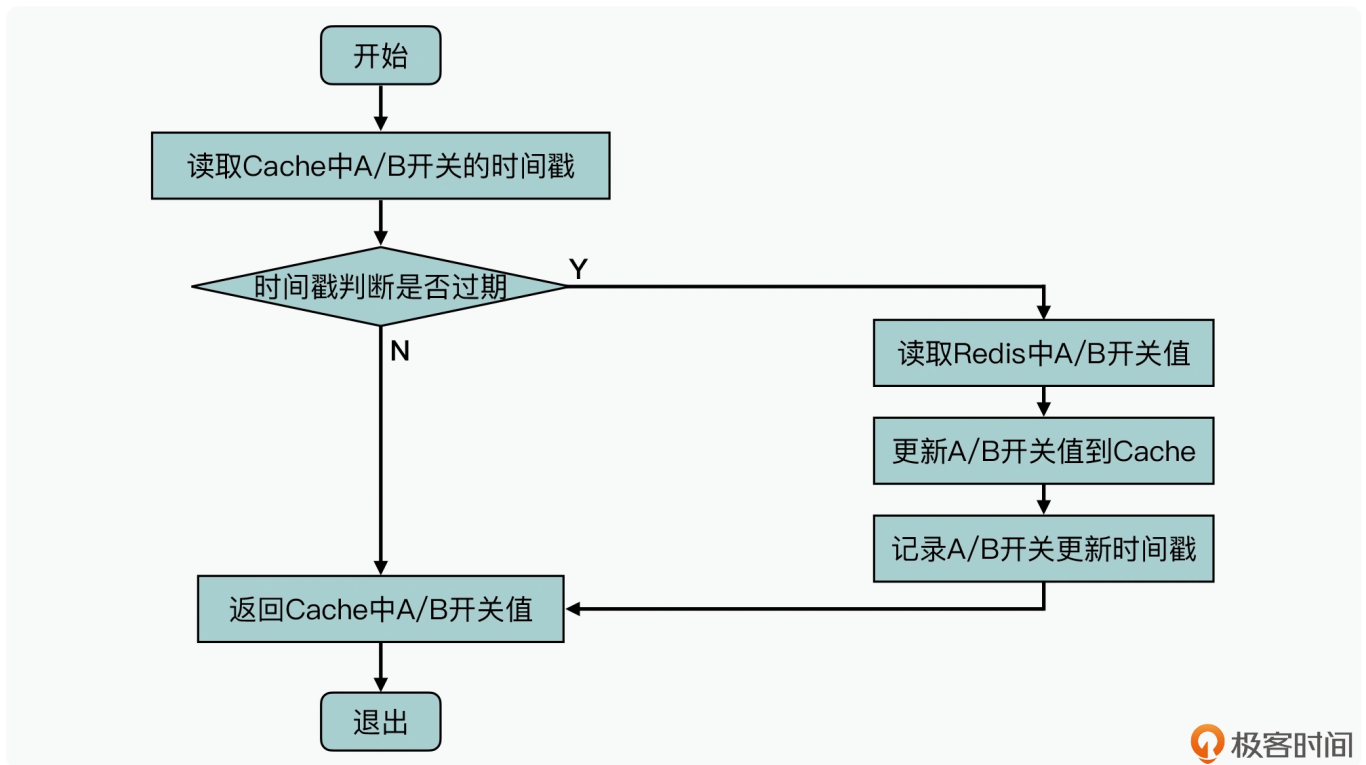
方式 2：在 A/B 测试库的外部再封装一层接口，来实现 Cache 机制，然后业务代码修改使用新的接口。

这两种实现方式我们具体该怎么选择呢？

使用方式 1 有一个好处是：所有的代码修改对 Web 服务中业务代码都完全不感知，如果出现异常时，直接更新依赖的 Split 库的版本即可。如果使用方式 2，业务中所有调用 A/B 测试库的接口处都需要进行修改，由于目前业务代码中使用 A/B 测试接口的地方特别多，所以综合分析之后，我选择采用方式 1。

这里你可能会想：在 A/B 测试库 split 的源码中直接实现内存 Cache 机制，不是会很复杂吗？因为它还需要引入一个内存 Cache 的库才行。

但其实，对于很多软件实现来说，可能你只需要加入一个 HashMap 数据结构，就可以实现 Cache 的机制了，所以，修改 A/B 测试功能的 Split 库的源代码，增加内存 Cache 机制的原理是比较简单的，具体原理如下图所示：



如图中所示，首先读取内存 Cache 中 A/B 开关值的时间戳，然后基于时间戳判断，这个 Cache 中数据是否已经过期（这里配置过期时间暂定 5s）。如果没有过期的情况下，系统可以直接返回 Cache 中记录的 A/B 开关值；否则的话，就需要重新读取 Redis 中的 A/B 开关配置信息，更新到 Cache 中并记录更新时间戳，然后再返回新读取到的 A/B 开关值。

完成这一步的 Cache 机制的代码修改时，其实还并没有修改我们在前面的分析中所发现的低性能问题：A/B 测试库 Split 实现中会重复读取 Redis 的值。那么，这个性能问题需要进行修改吗？

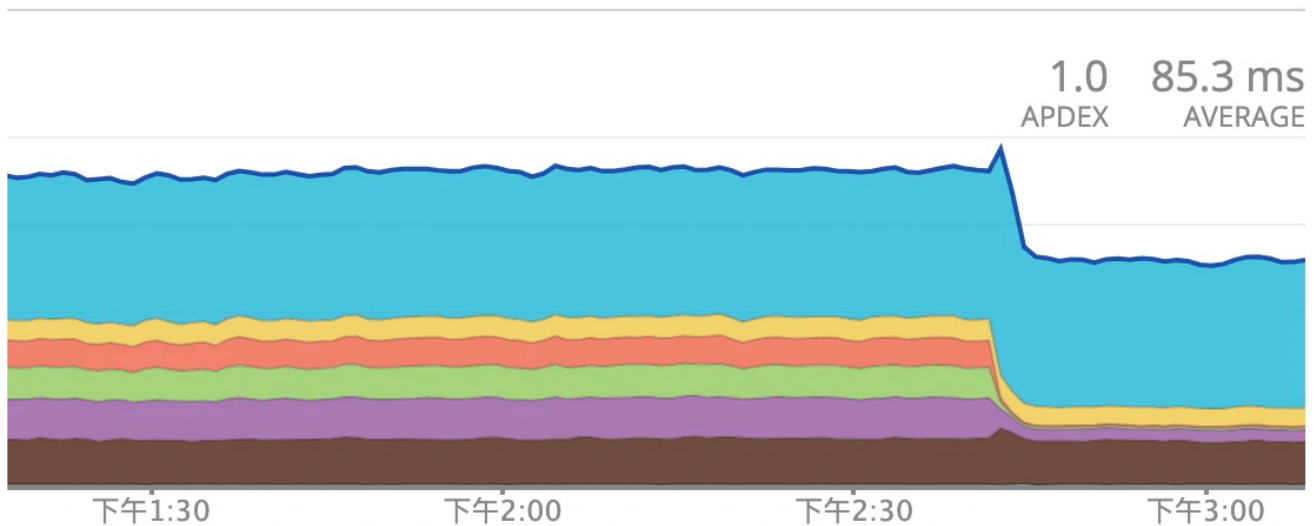
其实引入 Cache 之后，由于大量的 A/B 开关值都是从 Cache 中读取的，原来在 Split 库中重复读取 Redis 值请求的问题，对 Web 服务的性能影响会比较小，因此理论上你可以不用修改。

补充：不过我在阅读代码的过程中发现，其实只需要修改几行代码就可以解决这个问题，所以就顺带也修改了，但实际上我并不推荐这么做，因为这里修改性能收益是比较有限的。

那么到这里，我们就已经完成了这个性能优化方案的所有代码修改工作。不过现在还需要确认一个问题，就是在 A/B 测试的 Split 库中增加了内存级 Cache 优化修改后，这个 Web 服务的性能提升效果和预期是一致的吗？

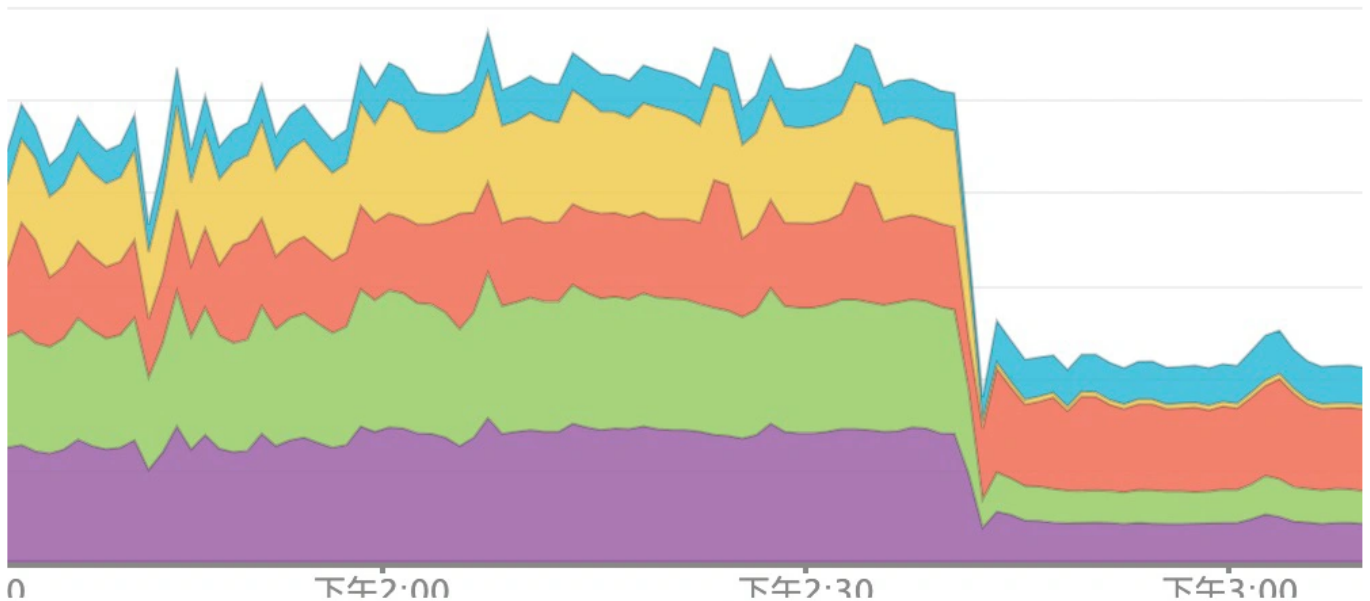
优化成果分析

为了更好地支撑优化成果分析，在优化版本上线过程中，我们又使用监控工具来获取了系统的平均响应时延变化情况，具体如下图所示：



从图上你可以看到，系统的平均响应时延值降低了接近 20%，远超过了性能优化分析的预估值 12%。其实，这种现象是比较正常的，这是因为**减少业务中大量 IO 请求操作的同时，也会减少进程 / 线程的切换发生次数**，而这部分的性能优化提升效果是比较难进行估算分析的。

同时我们还会发现，整个软件服务器集群对 Redis 的 API 请求操作次数也降低了接近一倍，具体如下图所示：



可以发现，当系统对 Redis 的 API 操作请求次数减少之后，也就意味着，现在系统中的 Redis 数据库就可以服务于更大的软件服务规模。所以很多时候，**一个点上的性能优化会改善产品中多个方面的性能特征。**

小结

今天我分享的这个性能优化案例，并没有去修改业务中的代码，只是修改了第三方库中少量的源代码，但是性能优化提升的效果却是非常明显的，所以在剖析完这个优化案例之后，你可以重点关注以下几个要点，来帮助支撑你的性能优化工作。

首先，我们业务软件的性能不仅会受制于系统内的代码实现，还会受到外部依赖库的执行性能影响；

其次，在性能优化的过程中，最具挑战的事情可能并不在于最后的代码修改，而在于前面的解决方案设计过程中，如何深入业务场景和软件设计实现，在每一个环节都进行分析、评估、取舍，最后才确定出更优的解决方案；

最后，就是其实在软件系统中有不少的业务操作，对实时性和一致性要求并不是非常高，你就可以采用 Cache 机制来优化提升软件性能。

思考题

在今天分享的性能优化案例中，Cache 机制中的 A/B 测试开关的失效时间配置为 5s，那么如果这个配置得再长一些，你认为会对软件有什么影响呢？

欢迎在留言区分享你的看法。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。

分享给需要的人，Ta订阅后你可得 **20元** 现金奖励

👍 赞 0

💡 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 27 | 解决一个互斥问题，系统并发用户数提升了10倍！

更多学习推荐

Java 面试必考 300 题

最新汇总

限时免费领取



精选留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

💬 写留言

