



下载APP



大咖助场 | 傅健：那些年，影响我们达到性能巅峰的常见绊脚石（下）

2020-07-20 傅健

系统性能调优必知必会

[进入课程 >](#)

讲述：张浩

时长 13:58 大小 12.80M



你好，我是傅健，又见面了。上一期分享我们总结了 3 个场景化的问题以及应对策略，这一期我们就接着“系统性能优化”这个主题继续总结。

场景 1：资源争用

案例

一段时间，我们总是监控到一些性能“掉队”的请求，例如平时我们访问 Cassandra 数据库都在 10ms 以内，但是偶尔能达到 3s，你可以参考下面这个度量数据：



1 {

复制代码

```
2    "stepName": "QueryInformation",
3    "values": {
4        "componentType": "Cassandra",
5        "totalDurationInMS": 3548,
6        "startTime": "2018-05-11T08:20:28.889Z",
7        "success": true
8    }
9
```

持续观察后，我们发现这些掉队的请求都集中在每天 8 点 20 分，话说“百果必有因”，这又是什么情况呢？

解析

这种问题，其实相对好查，因为它们有其发生的规律，这也是我们定位性能问题最基本的手段，即找规律：发生在某一套环境？某一套机器？某个时间点？等等，这些都是非常有用的线索。而这个案例就是固定发生在某个时间点。既然是固定时间点，说明肯定有某件事固定发生在这个点，所以查找问题的方向自然就明了了。

首先，我们上来排除了应用程序及其下游应用程序定时去做任务的情况。那么除了应用程序自身做事情外，还能是什么？可能我们会想到：运行应用程序的机器在定时做事情。果然，我们查询了机器的 CronJob，发现服务器在每天的 8 点 20 分（业务低峰期）都会去归档业务的日志，而这波集中的日志归档操作，又带来了什么影响呢？

日志备份明显是磁盘操作，所以我们要查看的是磁盘的性能影响，如果磁盘都转不动了，可想而知会引发什么。我们一般都会有图形化的界面去监控磁盘性能，但是这里为了更为通用的演示问题，所以我使用了 SAR 的结果来展示当时的大致情况（说是大致，是因为 SAR 的历史记录结果默认是以 10 分钟为间隔，所以只显示 8：20 分的数据，可能 8：21 分才是尖峰，但是却不能准确反映）：

		DEV	tps	rd_sec/s	wr_sec/s	avgrq-sz	avgqu-sz	await	svctm	%util
08:10:01	AM	dev8-0	3.69	0.00	95.72	25.98	0.01	2.38	0.97	0.36
08:20:01	AM	dev8-0	4.95	0.03	627.15	126.75	1.09	219.38	11.49	5.69
08:30:01	AM	dev8-0	3.00	0.00	68.34	22.80	0.01	2.38	1.21	0.36

从上面的结果我们可以看出，平时磁盘 await 只要 2ms，而 8：20 分的磁盘操作达到了几百毫秒。磁盘性能的急剧下降影响了应用程序的性能。

小结

在这个案例中，服务器上的定时日志归档抢占了我们的资源，导致应用程序速度临时下降，即资源争用导致了性能掉队。我们再延伸一点来看，除了这种情况外，还有许多类似的资源争用都有可能引起这类问题，例如我们有时候会在机器上装一些 logstash、collectd 等监控软件，而这些监控软件如果不限它们对资源的使用，同样也可能会争用我们很多的资源。诸如此类，不一一枚举，而针对这种问题，很明显有好几种方法可以供我们去尝试解决，以当前的案例为例：

降低资源争用，例如让备份慢一点；

错峰，错开备份时间，例如不让每个机器都是 8 点 20 分去备份，而是在大致一个时间范围内随机时间进行；

避免资源共享，避免多个机器 / 虚拟机使用同一块磁盘。

上面讨论的争用都发生在一个机器内部，实际上，资源争用也常发生在同一资源（宿主、NFS 磁盘等）的不同虚拟机之间，这也是值得注意的一个点。

场景 2：延时加载

案例


某日，某测试工程师胸有成竹地抱怨：“你这个 API 接口性能不行啊，我们每次执行自动化测试时，总有响应很慢的请求。”于是一个常见的争执场景出现了：当面演示调用某个 API 若干次，结果都是响应极快，测试工程师坚持他真的看到了请求很慢的时候，但是开发又坚定说在我的机器上它就是可以啊。于是互掐不停。



解析

开发者后来苦思冥想很久，和测试工程师一起比较了下到底他们的测试有什么不同，唯一的区别在于测试的时机不同：自动化测试需要做的是回归测试，所以每次都会部署新的包并执行重启，而开发者复现问题用的系统已经运行若干天，那这有什么区别呢？这就引入了我们这部分要讨论的内容——延时加载。

我们知道当系统去执行某个操作时，例如访问某个服务，往往都是“按需执行”，这非常符合我们的行为习惯。例如，需要外卖点餐时，我们才打开某 APP，我们一般会在还没有点餐的时候，就把 APP 打开“守株待兔”某次点餐的发生。这种思路写出来的系统，会让我们的系统在上线之时，可以轻装上阵。例如，非常类似下面的 Java “延时初始化” 伪代码：

 复制代码


```
1 public class AppFactory{
2     private static App app;
3     synchronized App getApp() {
4         if (App == null)
5             app= openAppAndCompleteInit();
6         return app;
7     }
8     //省略其它非关键代码
9 }
10
11 App app = AppFactory.getApp();
12 app.order("青椒土豆丝");
```

但这里的问题是什么呢？假设打开 APP 的操作非常慢，那等我们需要点餐的时候，就会额外耗费更长的时间（而一旦打开运行在后台，后面不管怎么点都会很快）。这点和我们的案例其实是非常类似的。

我们现在回到案例，持续观察发现，这个有性能问题的 API 只出现在第一次访问，或者说只出现在系统重启后的第一次访问。当它的 API 被触发时，它会看看本地有没有授权相关的信息，如果没有则远程访问一个授权服务拿到相关的认证信息，然后缓存认证信息，最后再使用这个认证信息去访问其它组件。而问题恰巧就出现在访问授权服务器完成初始化操作耗时比较久，所以呈现出第一次访问较慢，后续由于已缓存消息而变快的现象。

那这个问题怎么解决呢？我们可以写一个“加速器”，说白了，就是把“延时加载”变为“主动加载”。在系统启动之后、正常提供服务之前，就提前访问授权服务器拿到授权

信息，这样等系统提供服务之后，我们的第一个请求就很快了。类似使用 Java 伪代码，对上述的延时加载修改如下：

 复制代码

```
1 public class AppFactory{
2     private static App app = openAppAndCompleteInit();
3     synchronized App getApp() {
4         return app;
5     }
6     //省略其它非关键代码
7 }
```


小结

延时加载固然很好，可以让我们的系统轻装上阵，写出的程序也符合我们的行为习惯，但是它常常带来的问题是，在第一次访问时可能性能不符合预期。当遇到这种问题时，我们也可以根据它的出现特征来分析是不是属于这类问题，即是否是启动完成后的第一次请求。如果是此类问题，我们可以通过变“被动加载”为“主动加载”的方式来加速访问，从而解决问题。

但是这里我不得不补充一点，是否在所有场景下，我们都需要化被动为主动呢？实际上，还得具体情况具体分析，例如我们打开一个网页，里面内嵌了很多小图片链接，但我们是否会为了响应速度，提前将这些图片进行加载呢？一般我们都不会这么做。所以具体问题具体分析永远是真理。针对我们刚刚讨论的案例，这种加速只是一次性的而已，而且资源数量和大小都是可控的，所以这种修改是值得，也是可行的。

场景 3：网络抖动

案例

我们来看一则  新闻：



支付宝



12-5 17:25 来自 7P iPhone 11(黄色)



刚刚，支付宝的机房网络出现了短暂抖动，影响了部分用户的使用体验。一切已经恢复正常，大家的资金和信息安全不会受到影响哈。

类似的新闻还有许多，你可以去搜一搜，然后你就会发现：它们都包含一个关键词——网络抖动。

解析

那什么是网络抖动呢？网络抖动是衡量网络服务质量的一个指标。假设我们的网络最大延迟为 100ms，最小延迟为 10ms，那么网络抖动就是 90ms，即网络延时的最大值与最小值的差值。差值越小，意味着抖动越小，网络越稳定。反之，当网络不稳定时，抖动就会越大，网络延时差距也就越大，反映到上层应用自然是响应速度的“掉队”。

为什么网络抖动如此难以避免？这是因为网络的延迟包括两个方面：传输延时和处理延时。忽略处理延时这个因素，假设我们的一个主机进行一次服务调用，需要跨越千山万水才能到达服务器，我们中间出“岔子”的情况就会越多。我们在 Linux 下面可以使用 traceroute 命令来查看我们跋山涉水的情况，例如从我的 Linux 机器到百度的情况是这样的：

复制代码

```
1 [root@linux~]# traceroute -n -T www.baidu.com
2 traceroute to www.baidu.com (119.63.197.139), 30 hops max, 60 byte packets
3 1  10.224.2.1  0.452 ms  0.864 ms  0.914 ms
4 2  1.1.1.1  0.733 ms  0.774 ms  0.817 ms
5 3  10.224.16.193  0.361 ms  0.369 ms  0.362 ms
6 4  10.224.32.9  0.355 ms  0.414 ms  0.478 ms
7 5  10.140.199.77  0.400 ms  0.396 ms  0.544 ms
8 6  10.124.104.244  12.937 ms  12.655 ms  12.706 ms
9 7  10.124.104.195  12.736 ms  12.628 ms  12.851 ms
10 8  10.124.104.217  13.093 ms  12.857 ms  12.954 ms
11 9  10.112.4.65  12.586 ms  12.510 ms  12.609 ms
12 10  10.112.8.222  44.250 ms  44.183 ms  44.174 ms
```

```

13 11 10.112.0.122 44.926 ms 44.360 ms 44.479 ms
14 12 10.112.0.78 44.433 ms 44.320 ms 44.508 ms
15 13 10.75.216.50 44.295 ms 44.194 ms 44.386 ms
16 14 10.75.224.202 46.191 ms 46.135 ms 46.042 ms
17

```

通过上面的命令结果我们可以看出，我的机器到百度需要很多“路”。当然大多数人并不喜欢使用 traceroute 来评估这段路的艰辛程度，而是直接使用 ping 来简单看看“路”的远近。例如通过以下结果，我们就可以看出，我们的网络延时达到了 40ms，这时网络延时就可能是一个问题了。

[复制代码](#)

```

1 [root@linux~]# ping www.baidu.com
2 PING www.wshifen.com (103.235.46.39) 56(84) bytes of data.
3 64 bytes from 103.235.46.39: icmp_seq=1 ttl=41 time=46.2 ms
4 64 bytes from 103.235.46.39: icmp_seq=2 ttl=41 time=46.3 ms

```

其实上面这两个工具的使用只是直观反映网络延时，它们都默认了一个潜规则：网络延时越大，网络越抖动。且不说这个规则是否完全正确，至少从结果来看，评估网络抖动并不够直观。

所以我们可以再寻求一些其它的工具。例如可以使用 MTR 工具，它集合了 tractroute 和 ping。我们可以看下执行结果：下图中的 best 和 wrst 字段，即为最好的情况与最坏的情况，两者的差值也能在一定程度上反映出抖动情况，其中不同的 host 相当于 traceroute 经过的“路”。


My traceroute [v0.75]									
Keys: Help Display mode Restart statistics Order of fields quit									
Wed Jul 8 09:08:33 2020									
Host	Packets			Pings					
	Loss%	Snt	Last	Avg	Best	Wrst	StDev		
1. 10.224.2.1	0.0%	26	0.6	0.6	0.5	1.0	0.1		
2. 1.1.1.1	0.0%	26	0.5	0.6	0.5	1.3	0.2		
3. 10.224.16.193	0.0%	26	0.3	1.3	0.3	23.8	4.6		
4. 10.224.32.9	0.0%	26	0.4	0.4	0.3	0.5	0.0		
5. 10.140.199.77	0.0%	26	0.3	0.4	0.3	1.5	0.2		

小结

对于网络延时造成的抖动，特别是传输延迟造成的抖动，我们一般都是“有心无力”的。只能说，我们需要做好网络延时的抖动监测，从而能真正定位到这个问题，避免直接无证据就“甩锅”于网络。

另外，在做设计和部署时，我们除了容灾和真正的业务需求，应尽量避免或者说减少“太远”的访问，尽量将服务就近到一个机房甚至一个机柜，这样就能大幅度降低网络延迟，抖动情况也会降低许多，这也是为什么 CDN 等技术兴起的原因。

那同一网络的网络延时可以有多好呢？我们可以自己测试下。例如，我的 Linux 机器如果 ping 同一个网段的机器，是连 1ms 都不到的：

 复制代码

```
1 [root@linux~]# ping 10.224.2.146
2 PING 10.224.2.146 (10.224.2.146) 56(84) bytes of data.
3 64 bytes from 10.224.2.146: icmp_seq=1 ttl=64 time=0.212 ms
4 64 bytes from 10.224.2.146: icmp_seq=2 ttl=64 time=0.219 ms
5 64 bytes from 10.224.2.146: icmp_seq=3 ttl=64 time=0.154 ms
```


场景 4：缓存失效

案例

在产线上，我们会经常观察到一些 API 接口调用周期性（固定时间间隔）地出现“长延时”，而另外一些接口调用也会偶尔如此，只是时间不固定。继续持续观察，你会发现，虽然后者时间不够规律，但是它们的出现时间间隔都是大于一定时间间隔的。那这种情况是什么原因导致的呢？

解析

当我们遇到上述现象，很有可能是因为“遭遇”了缓存失效。缓存的定义与作用这里不多赘述，你应该非常熟悉了。同时我们也都知道，缓存是以空间换时间的方式来提高性能，讲究均衡。在待缓存内容很多的情况下，不管我们使用本地缓存，还是 Redis、Memcached 等缓存方案，我们经常都会受限于“空间”或缓存条目本身的时效性，而给缓存设置一个失效时间。而当失效时间到来时，我们就需要访问数据的源头从而增加延时。这里以下面的缓存构建代码作为示例：

 复制代码

```
1 CacheBuilder.newBuilder().maximumSize(10_000).expireAfterWrite(10, TimeUnit.MI
```


我们可以看到：一旦缓存后，10 分钟就会失效，但是失效后，我们不见得刚好有请求过来。如果这个接口是频繁调用的，则长延时请求的出现频率会呈现出周期性的 10 分钟间隔规律，即案例描述的前者情况。而如果这个接口调用很不频繁，则我们只能保证在 10 分钟内的延时是平滑的。当然，这里讨论的场景都是在缓存够用的情况下，如果缓存的条目超过了设定的最大允许值，这可能会提前淘汰一些缓存内容，这个时候长延时请求出现的规律就无章可循了。

小结

缓存失效是导致偶发性延时增加的常见情况，也是相对来说比较好定位的问题，因为接口的执行路径肯定是不同的，只要有充足的日志，即可找出问题所在。另外，针对这种情况，一方面我们可以增加缓存时间来尽力减少长延时请求，但这个时候要求的空间也会增大，同时也可能违反了缓存内容的时效性要求；另一方面，在一些情况下（比如缓存条目较少、缓存的内容可靠性要求不高），我们可以取消缓存的 TTL，更新数据库时实时更新缓存，这样读取数据就可以一直通过缓存进行。总之，应情况调整才是王道。

总结

结合我上一期的分享，我们一共总结了 7 种常见“绊脚石”及其应对策略，那通过了解它们，我们是否就有十足的信心能达到性能巅峰了呢？

其实在实践中，我们往往还是很难的，特别是当前微服务大行其道，决定我们系统性能的因素往往是下游微服务，而下游微服务可能来源于不同的团队或组织。这时，已经不再单纯是技术本身的问题了，而是沟通、协调甚至是制度等问题。但是好在对于下游微服务，我们依然可以使用上面的分析思路来找出问题所在，不过通过上面的各种分析你也可以知道，让性能做到极致还是很难的，总有一些情况超出我们的预期，例如我们使用的磁盘发生损害，彻底崩溃前也会引起性能下降。

另外一个值得思考的问题是，是否有划算的成本收益比去做无穷无尽的优化，当然对于技术极客来说，能不能、让不让解决问题也许不是最重要的，剥丝抽茧、了解真相才是最有成就感的事儿。

感谢阅读，希望今天的分享能让你有所收获！如果你发现除了上述我介绍的那些“绊脚石”外，还有其它一些典型情况存在，也欢迎你在留言区中分享出来作为补充。

提建议

更多课程推荐

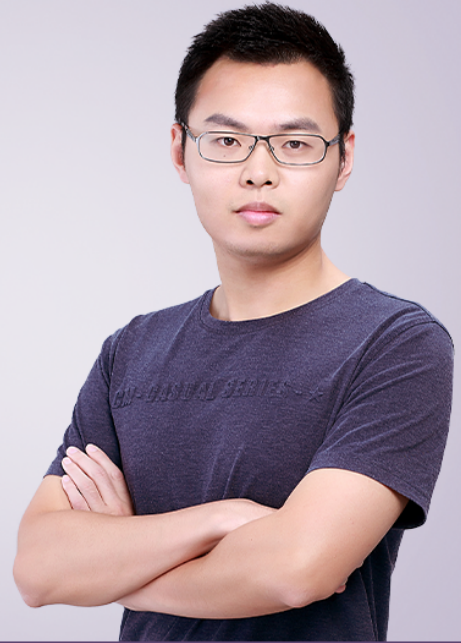
设计模式之美

前 Google 工程师手把手教你写高质量代码

王争

前 Google 工程师

《数据结构与算法之美》专栏作者



涨价倒计时 🕒

限时秒杀 **¥149**，7月31日涨价至 **¥299**

© 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 大咖助场 | 傅健：那些年，影响我们达到性能巅峰的常见绊脚石（上）

下一篇 28 | MapReduce：如何通过集群实现离线计算？

精选留言 (1)

写留言



Jeff.Smile

2020-07-20

这个专栏感觉挺值得的，陶辉老师的思路很清晰，傅健老师的实践经验，两个结合，所向披靡！

2



