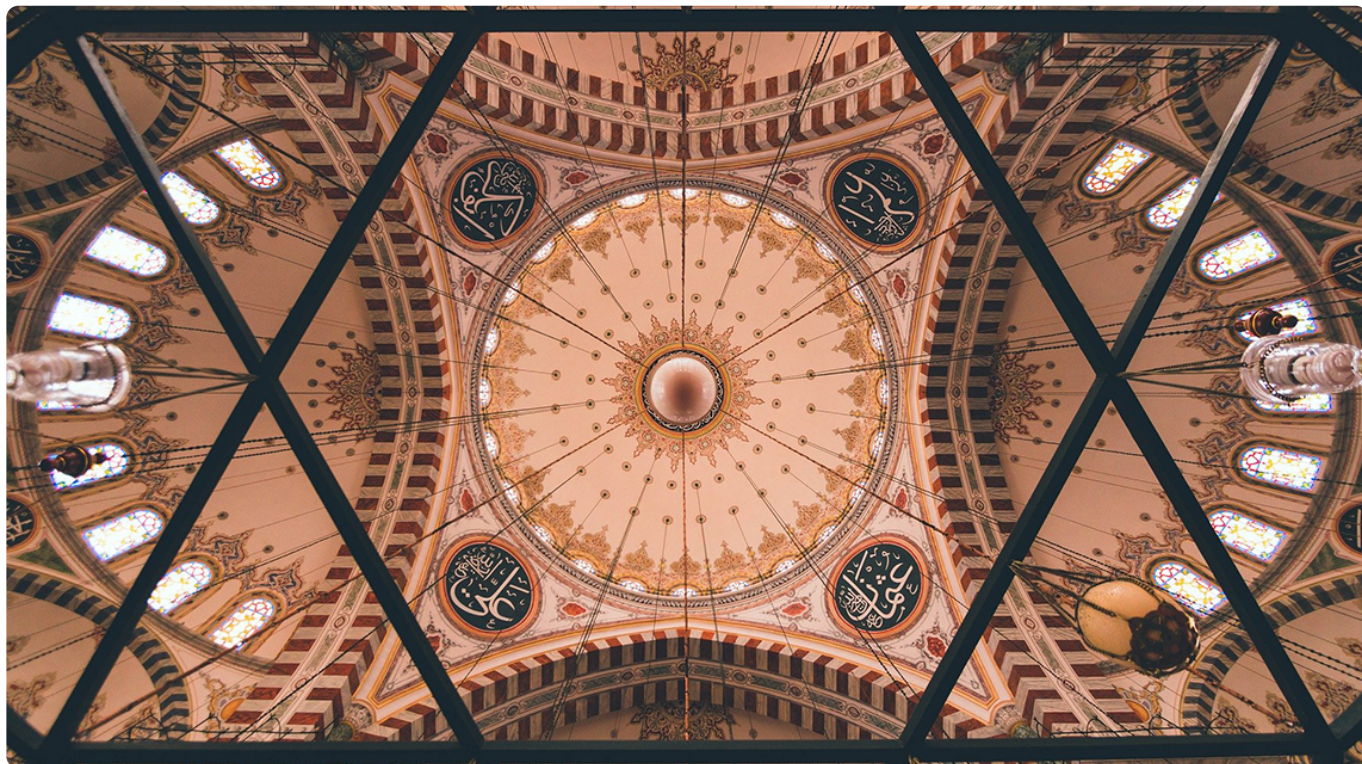


30 | 案例：为什么参数化数据会导致TPS突然下降？

2020-02-28 高楼

性能测试实战30讲

[进入课程 >](#)



讲述：高楼

时长 20:15 大小 16.23M



写这篇文章的时候，我想起一句似乎无关紧要的话：“我离你如此之近，你却对我视而不见。”

在性能测试中，参数化数据是少有的每个性能测试工程师都会用得到，却经常出现问题的技术点之一。从我的角度来说，究其原因，大部分是因为对性能参数化数据的理解不足。导致的结果就是用了参数化，但和真实的用户场景不一致，从而使得整个性能测试场景都失去了意义。

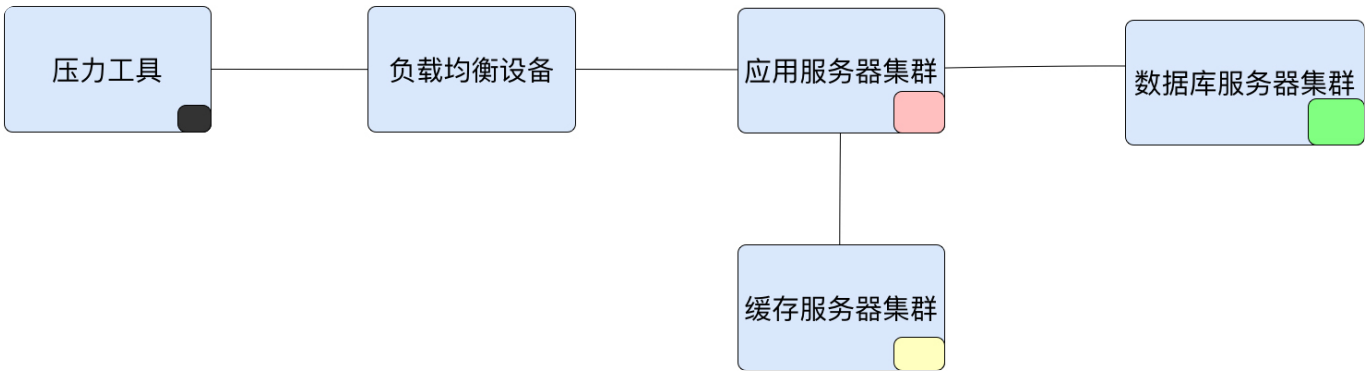


这样的例子不在少数。

一个项目开始之初，由于没有历史沉淀的数据，所以我们需要造一些数据来做性能测试。造多少呢？并不是按未来生产的容量来造，而是按性能场景中需要的数据量级来造。这种错误的做法是很多项目中真实出现的事情。

这并不止是性能测试工程师之过，还有很多其他的复杂原因，比如时间不够；经验不足，只能造重复的数据等等。

那么性能测试参数化数据的获取逻辑到底是什么呢？我们来看一个图吧。



在这个图中，我用不同的颜色表示不同组件中的数据。压力工具中的参数化数据有两种，这一点，我们前面有提到过，参数化数据有两大类型：

- 1. 用户输入的数据同时在后台数据库中已存在。
- 2. 用户输入的数据同时在后台数据库中不存在。

当我们使用数据库中已存在的数据时，就必须考虑到这个数据是否符合真实用户场景中的数据分布。当我们使用数据库中不存在的数据时，就必须考虑输入是否符合真实用户的输入。

在本篇要说的案例中，我们来看一下参数化数据如果做错了，对性能结果会产生什么样的影响。

案例问题描述

在一次压力测试的过程中，出现了如下所示的 TPS 数据（本篇文章中一些截图会有些模糊，因为来自于之前项目中的具体案例，在当时截图时，也并没有考虑清晰度，不过我们只要看趋势就好）。

在下图中，我们可以看到，在压力测试过程中，出现了 TPS 陡减到底的情况。这显然是不合理的。



线程数：300 - 100 - 50 - 10 - 1

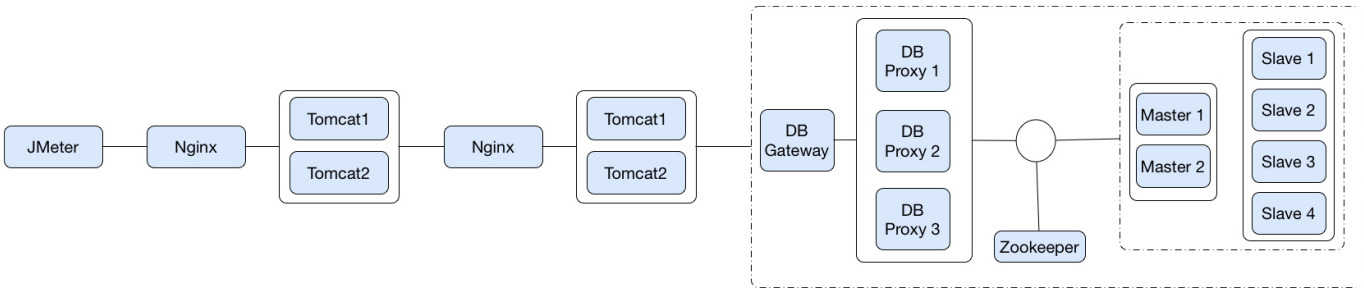
这个曲线的趋势把性能瓶颈呈现得非常明显。在出现这个问题之后，当时我们也尝试过把线程数降低，观察 TPS 的趋势，结果从 300 到 100 到 50 到 10，最后到 1，发现都会出现这样的 TPS 陡减到底的情况，只是时间长度不同而已。

这非常像某个资源因为处理业务量的累积达到了某个临界点而产生的情形。

但不管怎样，我们还是要按正常分析的思路来分析它。

分析过程

首先，仍然是画一个架构图。



在这个图中，我们可以看到，JMeter 是连接到第一层服务（这里是有两个 Tomcat 实例），再到第二层服务（这里也是有两个 Tomcat 实例），然后再连到 DB 中。这个 DB 是一个互联网金融 DB（通过 MySQL 改造来的）。

了解了架构图之后，现在就开始查看下性能数据吧。

查操作系统

先看一下操作系统的性能数据：

top - 22:51:42 up 39 days, 6:22, 2 users, load average: 0.22, 0.14, 0.14
Tasks: 121 total, 1 running, 119 sleeping, 0 stopped, 1 zombie
%Cpu0 : 8.6 us, 2.8 sy, 0.0 ni, 88.3 id, 0.0 wa, 0.0 hi, 0.3 si, 0.0 st
%Cpu1 : 8.5 us, 2.7 sy, 0.0 ni, 88.4 id, 0.0 wa, 0.0 hi, 0.0 si, 0.3 st
%Cpu2 : 9.3 us, 3.1 sy, 0.0 ni, 87.6 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu3 : 8.2 us, 3.7 sy, 0.0 ni, 87.4 id, 0.0 wa, 0.0 hi, 0.3 si, 0.3 st
KiB Mem : 8010952 total, 2009128 free, 911736 used, 5090088 buff/cache
KiB Swap: 0 total, 0 free, 0 used. 6813832 avail Mem

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
18860	app	20	0	7954696	716464	10680	S	51.8	8.9	84:03.01	java
13	root	20	0	0	0	0	S	0.3	0.0	21:35.07	rcu_sched
3724	app	20	0	144052	2020	1368	S	0.3	0.0	0:22.64	top
3725	root	20	0	144052	2024	1372	R	0.3	0.0	0:22.73	top
1	root	20	0	78672	41108	2288	S	0.0	0.5	7:31.07	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.45	kthreadd
3	root	20	0	0	0	0	S	0.0	0.0	0:32.56	ksoftirqd/0
5	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	kworker/0:0H
7	root	rt	0	0	0	0	S	0.0	0.0	0:18.44	migration/0
8	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcu_bh
9	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcuob/0
10	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcuob/1
11	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcuob/2
12	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcuob/3
14	root	20	0	0	0	0	S	0.0	0.0	8:54.99	rcuos/0

从 top 中，我们可以看到这个应用服务器没啥压力，在这样的状态中，你可能都不用再去查其他的操作系统信息了，因为目前的压力对这个系统来说确实是小了点。

查应用

再看下应用的状态，这里用的工具仍然是前文中提到过多次的 JvisualVM（请你在用性能监控工具的时候，不要纠结，只要工具好使，用到吐都行，不用跟风）。



从这个图中可以看到的是，这个应用使用到的 CPU 确实很低，并且堆也没用多少。其实在这一步，我查了四个 Tomcat 的状态，只是截了一个图而已。

在这里还是要啰嗦一下了，对这样的曲线，我们一定要一眼就能看出问题在哪里。出现上图这样的情况是因为以下两个原因：

1. 应用 CPU 使用率（橙色 CPU 线）确实是太低了，才 15% 左右。这和前面的 top 也是能对得上的。Java 的 GC 几乎没占 CPU（蓝色 CPU 线），也就是说 Tomcat 在这里没压力。
2. 从堆曲线的趋势上来看，1G 的堆才到了 400M 多一点，并且回收一直都非常正常。怎么判断这个“正常”呢？首先，年轻代、年老代回收很有规律，并且没消耗什么 CPU；其次，每次 FullGC 都能回到 150M 左右，非常平稳。可见这个内存使用没啥问题。

当然到了这里，我当时也是查了网络的，只是也没什么压力，所以没做具体的记录（从这点可以看出，如果你在做性能测试的时候，要想记录性能瓶颈的分析过程，一定要记得把数据

记全了，不然以后你可能都想不起来当时做了什么事情）。

查 DB

既然上面都没啥问题，DB 又是一个 MySQL，所以这里，我先手动执行了几个常规的查询语句。在 DB 中查看如下信息。

查processlist、innodb_trx、innodb_locks、innodb_lock_waits。在没有监控工具时，这几个是我经常在 MySQL 数据库检查的表，因为数据库如果慢的话，基本上会在这几个表中留些蛛丝马迹。

processlist 是看当然数据库中的 session 的，并且也会把正在执行的 SQL 列出来，快速刷新几次，就可以看到是不是有 SQL 一直卡在那里。

innodb_trx是正在执行的 SQL 事务表，这个表很重要。

innodb_locks和innodb_lock_waits是为了看有没有锁等待。

拿一条业务 SQL 执行一下，看看在压力之中会不会慢。这是在没有数据库监控时，快速判断业务的方法。因为这个业务很单一，用的 SQL 也单一，所以我在这里可以这样做。执行了之后，并没有发现业务 SQL 慢。

由此基本判断 DB 没什么问题。

注意，判断到了这里，其实已经出现了证据不完整产生的方向偏离！

陷入困局之后的手段

更悲催的是这个业务系统的日志记录的非常“简洁”，连时间消耗都没有记录下来。想来想去，在这么简单的一个架构中，没什么可查的东西了吧，除非网络中有设备导致了这个问题的出现？

在没有其它监控工具的情况下，当时我们上了最傻最二最基础又最有效的时间拆分手段：抓包！

抓包其实是个挺需要技巧的活，不止是说你都能把包抓出来，还要能分析出来时间消耗在谁那里。这时我提醒一下，当你学会抓包工具的使用时，不要在每个场景下都想露一手你的抓包能力，通过抓的包分析响应时间的消耗点。

在我的工作中，只有万般无奈时才会祭出“抓包”这样的手段，并不是因为我对网络不够了解。恰恰是因为了解得足够多的，我才建议不要随便抓包。因为但凡在应用层有工具可以分析响应时间，都会比抓网络层的包来得更加简单直观。

经过一段段的分析之后，在数据库的一个主机上看到了如下信息：

tcp_stream eq 2007						
Time	Source	Destination	Protocol	Length	Info	
4303126 13:29:50.403752	10.21.12.39	10.21.12.21	TCP	60	8080 → 52041 [ACK] Seq=63636 Ack=15499 Win=73216 Len=0	
4303430 13:29:52.433451	10.21.12.39	10.21.12.21	TCP	1514	[TCP segment of a reassembled PDU]	
4303431 13:29:52.433455	10.21.12.39	10.21.12.21	TCP	1514	[TCP segment of a reassembled PDU]	
4303432 13:29:52.433470	10.21.12.39	10.21.12.21	HTTP	915	HTTP/1.1 200 (application/json)	
4303433 13:29:52.433534	10.21.12.21	10.21.12.39	TCP	54	52041 → 8080 [ACK] Seq=15499 Ack=67417 Win=131328 Len=0	
4303434 13:29:52.434025	10.21.12.21	10.21.12.39	TCP	303	[TCP segment of a reassembled PDU]	
4303435 13:29:52.434069	10.21.12.21	10.21.12.39	HTTP	666	POST /bankchannel/wms/query/transrequestquery.json HTTP/1.1 (application/json)	
4303436 13:29:52.434375	10.21.12.39	10.21.12.21	TCP	60	8080 → 52041 [ACK] Seq=67417 Ack=15748 Win=74368 Len=0	
4303437 13:29:52.434378	10.21.12.39	10.21.12.21	TCP	60	8080 → 52041 [ACK] Seq=67417 Ack=16360 Win=75648 Len=0	
4303756 13:29:54.445736	10.21.12.39	10.21.12.21	TCP	1514	[TCP segment of a reassembled PDU]	
4303757 13:29:54.445741	10.21.12.39	10.21.12.21	TCP	1514	[TCP segment of a reassembled PDU]	
4303758 13:29:54.445754	10.21.12.39	10.21.12.21	HTTP	808	HTTP/1.1 200 (application/json)	
4303759 13:29:54.445803	10.21.12.21	10.21.12.39	TCP	54	52041 → 8080 [ACK] Seq=16360 Ack=71091 Win=131328 Len=0	
4303760 13:29:54.446545	10.21.12.21	10.21.12.39	TCP	303	[TCP segment of a reassembled PDU]	
4303761 13:29:54.446591	10.21.12.21	10.21.12.39	HTTP	666	POST /bankchannel/wms/query/transrequestquery.json HTTP/1.1 (application/json)	
4303762 13:29:54.446800	10.21.12.39	10.21.12.21	TCP	60	8080 → 52041 [ACK] Seq=71091 Ack=16609 Win=76800 Len=0	
4303763 13:29:54.446826	10.21.12.39	10.21.12.21	TCP	60	8080 → 52041 [ACK] Seq=71091 Ack=17221 Win=78080 Len=0	
4304067 13:29:56.448636	10.21.12.39	10.21.12.21	TCP	1514	[TCP segment of a reassembled PDU]	
4304068 13:29:56.448640	10.21.12.39	10.21.12.21	TCP	1514	[TCP segment of a reassembled PDU]	
4304069 13:29:56.448656	10.21.12.39	10.21.12.21	HTTP	808	HTTP/1.1 200 (application/json)	
4304070 13:29:56.448745	10.21.12.21	10.21.12.39	TCP	54	52041 → 8080 [ACK] Seq=17221 Ack=74765 Win=131328 Len=0	
4304071 13:29:56.449230	10.21.12.21	10.21.12.39	TCP	303	[TCP segment of a reassembled PDU]	
4304072 13:29:56.449272	10.21.12.21	10.21.12.39	HTTP	666	POST /bankchannel/wms/query/transrequestquery.json HTTP/1.1 (application/json)	
4304073 13:29:56.449566	10.21.12.39	10.21.12.21	TCP	60	8080 → 52041 [ACK] Seq=74765 Ack=17470 Win=79232 Len=0	
4304074 13:29:56.449571	10.21.12.39	10.21.12.21	TCP	60	8080 → 52041 [ACK] Seq=74765 Ack=18082 Win=80512 Len=0	
4304405 13:29:58.443957	10.21.12.39	10.21.12.21	TCP	1514	[TCP segment of a reassembled PDU]	
4304406 13:29:58.443961	10.21.12.39	10.21.12.21	TCP	1514	[TCP segment of a reassembled PDU]	
4304407 13:29:58.443976	10.21.12.39	10.21.12.21	HTTP	808	HTTP/1.1 200 (application/json)	
4304408 13:29:58.444022	10.21.12.21	10.21.12.39	TCP	54	52041 → 8080 [ACK] Seq=18082 Ack=78439 Win=131328 Len=0	
4304409 13:29:58.444534	10.21.12.21	10.21.12.39	TCP	303	[TCP segment of a reassembled PDU]	

看到这里的 TCP segment of a reassembled PDU 没有？它之上是 ACK。放大一下，看看这里的时间：

4303763	13:29:54.446826	10.21.12.39	10.21.12.21	TCP	60	8080 → 52041	[ACK] Seq=71091 Ack=16609 Win=76800 Len=0
4304067	13:29:56.448636	10.21.12.39	10.21.12.21	TCP	1514		[TCP segment of a reassembled PDU]

看到没有，这里有两秒的时间才发数据，那它是在干吗呢？

这里就要说明一下TCP segment of a reassembled PDU了，PDU 就是Protocol Data Unit。

以下高能烧脑，不喜可跳过！

它是指在 TCP 层接收到应用层发的非常大的数据之后，需要将数据大刀阔斧地砍成几段之后再发出去。就是这个砍数据的过程消耗了 2 秒的时间。

可是为什么 TCP 层要干这个事呢？上层应用给了你一大块数据包，你直接往外扔不就行了吗？还要自己 reassemble（重新装配），费老大劲。

这其实 TCP 的一个参数来决定的，它就是 MSS（Maximum Segment Size）。在 TCP 一开始打招呼的时候（就是握手的过程），已经通过 MSS 这个可选项告诉对方自己能接收的最大报文是多少了，这是不加任何信息的大小，纯的。而在以太网上，这个值是设置为 1460 字节的，为啥是 1460 呢？因为加上 TCP 头的 20 个字节和 IP 头的 20 个字节，刚好不大不小 1500 字节。

当你看到 1500 字节的时候，是不是有一种似曾相识的感觉？它就是现在普遍设置的 MTU（Maximum Transmission Unit）的大小呀。

这时你可能会说了，那我可以把 MTU 设置大嘛。可是你自己设置不行呀，别人（各主机和网络设备）都得跟着你设置才行，要不然到了 MTU 不大的地方，还得分包，还是要费时间。

而接收端呢？接数据时接到这些包的 ACK 序号都是一样的，但 Sequence Number 不同，并且后一个 Sequence Number 是前一个 Sequence Number+ 报文大小的值，那接收端就可以判断这是一个 TCP Segment 了。

好了，解释完这些之后，回到前面的问题。数据库自己耗时了两秒来做 reassemble PDU。至于吗？不就是过来查个数据吗？考虑了一下业务特征，这就是根据客户 ID 查一个帐户的一个月或三个月的记录信息，通常是 100 条左右，最多也就 200 条，也不至于有这么大。但是不管怎么样，还是数据库的问题！


这就是我前面说的查 DB 的时候，由于证据不全导致了分析思路的偏差。因为我手动执行了这个语句的时候并不慢，只要 10 几毫秒，所以，那时候我觉得数据库不是问题点。

但是经过了抓包之后，发现问题还是出在 DB 上。有时候真不能那么自信呀，容易给自己挖坑，要是早把活干得细致一点，也不至于要抓包了。

接着分析 DB

那我们肯定要接着看 DB 上的信息了，既然数据量大，SQL 执行得慢，那就先捞出慢日志看看。

查看如下负载信息：

 复制代码

```
1 # Profile
2
3
4 # Rank Query ID          Response time      Calls R/Call  V/M    Item
5 # ==== =====
6 #      1 0xB5DEC0207094BA2F 117365.8906 44.9% 14120   8.3120   8.46 SELECT
7 #      2 0xFF8A1413823E401F  62050.0287 23.7% 12078   5.1374   2.78 SELECT
8 #      3 0xC861142E667B5663  36004.3209 13.8% 21687   1.6602   0.13 SELECT
9 #      4 0xFB7DBC1F41799DDD  32413.9030 12.4% 19615   1.6525   0.09 SELECT
10 #     5 0xC065900AEAC5717F  11056.5444  4.2%  9304   1.1884   0.02 SELECT
11 #     9 0x6422DFBA813FC194    202.4342  0.1%    54   3.7488   1.83 INSERT
12 #    11 0x197C9DCF5DB927C8    137.4273  0.1%    36   3.8174   1.14 INSERT
13 #    13 0x1A9D64E72B53D706     97.9536  0.0%    31   3.1598   2.65 UPDATE
14 #    36 0x3B44178A8B9CE1C3     20.1134  0.0%    16   1.2571   0.04 INSERT
15 #    39 0x370753250D9FB9EF     14.5224  0.0%    11   1.3202   0.04 INSERT
16 # MISC 0xMISC          2152.2442  0.8%   151 14.2533   0.0 <72 ITEMS>
```


你可以看到确实有四个 SQL 消耗了更多的时间，并且时间还不短。这是明显的性能问题，但是我把这 SQL 拿出来执行过呀，并不慢。

怎么回事呢？

我让做数据库运维的人把 DB proxy 层的所有 SQL 日志拿出来分析一遍。为什么我要 DB proxy 层的数据呢？因为这一段会把所有执行的 SQL 都记录下来，而慢日志记录的是 1s 以上的（取决于 DB 中的配置）。首先是把 time cost 大于 200ms 的 SQL 都拉出来，结果发现，真的在 TPS 下降的那个时间段，出现了 SQL 执行超长的情况，并且和我执行的，还是同样的业务 SQL。

怎么办？既然到这个层面了，这些执行的 SQL 只有一点区别，那就是查询条件。慢的 SQL 的查询条件，我拿回来试了，果然是慢，查出来的数据也是完全不一样的，居然能查出几万条数据来。前面说了，这个语句是根据客户 ID 查出记录数的，那么就根据客户 ID，做一次 group by，看下数据量为啥有这么大差别。

于是得到了如下的结果：

 复制代码

```
1  客户ID, 数量
2  '这一列只是客户id, 无它', '91307'
3  '这一列只是客户id, 无它', '69865'
4  '这一列只是客户id, 无它', '55075'
5  '这一列只是客户id, 无它', '54990'
6  '这一列只是客户id, 无它', '54975'
7  '这一列只是客户id, 无它', '54962'
8  '这一列只是客户id, 无它', '54899'
9  '这一列只是客户id, 无它', '54898'
10 '这一列只是客户id, 无它', '54874'
11 '这一列只是客户id, 无它', '54862'
12 .....
13 '这一列只是客户id, 无它', '161'
14 '这一列只是客户id, 无它', '161'
15 '这一列只是客户id, 无它', '161'
16 '这一列只是客户id, 无它', '161'
17 '这一列只是客户id, 无它', '161'
18 '这一列只是客户id, 无它', '161'
19 '这一列只是客户id, 无它', '160'
20 '这一列只是客户id, 无它', '160'
```

从这个结果可以看到，不同客户 ID 的记录条数差别太大了。这是谁干的好事？！我们一开始就强调数据需要造均衡，要符合生产真实用户的数据分布。

到这里，问题基本上就明确了，查一下参数化的数据，里面有 10 万条数据，而取到记录数在五六万左右的客户 ID 的时候，才出现了响应时间长的问题。

而我之前的执行的 SQL，恰好试了多次都是数据量少的。

下面怎么办呢？先做个最规矩的实验，把 5 万条往后的数据全都删掉！场景再执行一遍。

于是就得到了如下的结果：

所以我现在都会诚心地告诫一些性能测试从业人员：一定要全局监控、定向监控一层层数据查，不要觉得查了某个点就判断这个组件没问题了。像我这样的老鸟也照样得从全局查起，不然也是掉坑里。而这个“全局 - 定向”的思路，也照样适用一些新手，可以形成排查手册。

在我带过的项目中，我经常讲这样的思路，制作排查手册（因为每个项目用的东西都会有些区别），而这些思路和排查手册，现在就变成了你一篇篇看过的文章。

所以我希望看专栏的人都能知道真正的分析性能瓶颈的过程是什么样子。不要在意自己现在会什么，要多在意以后会什么。

问题

讲完了今天的内容，你能说一下为什么通过抓包可以判断出响应时间的拆分吗？以及，数据分布不均衡还会带来哪些性能问题？

欢迎你在评论区写下你的思考，也欢迎把这篇文章分享给你的朋友或者同事，一起交流一下。

课程学习计划

关注极客时间服务号 每日学习签到

月领 25+ 极客币

【点击】保存图片，打开【微信】扫码>>>



© 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](#) [30 | 案例：当磁盘参数导致I/O高的时候，应该怎么办？](#)

[下一篇](#) [31 | 案例：当磁盘参数导致I/O高的时候，应该怎么办？](#)

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。