

## 28 | 案例篇：一个SQL查询要15秒，这是怎么回事？

2019-01-23 倪朋飞

Linux性能优化实战

[进入课程 >](#)



讲述：冯永吉

时长 18:17 大小 16.75M



你好，我是倪朋飞。

上一节，我们分析了一个单词热度应用响应过慢的案例。当用 top、iostat 分析了系统的 CPU 和磁盘 I/O 使用情况后，我们发现系统出现了磁盘的 I/O 瓶颈，而且正是案例应用导致的。

接着，在使用 strace 却没有任何发现后，我又给你介绍了两个新的工具 filetop 和 opensnoop，分析它们对系统调用 write() 和 open() 的追踪结果。

我们发现，案例应用正在读写大量的临时文件，因此产生了性能瓶颈。找出瓶颈后，我们又用把文件数据都放在内存的方法，解决了磁盘 I/O 的性能问题。

当然，你可能会说，在实际应用中，大量数据肯定是要存入数据库的，而不会直接用文本文件的方式存储。不过，数据库也不是万能的。当数据库出现性能问题时，又该如何分析和定位它的瓶颈呢？

今天我们就来一起分析一个数据库的案例。这是一个基于 Python Flask 的商品搜索应用，商品信息存在 MySQL 中。这个应用可以通过 MySQL 接口，根据客户端提供的商品名称，去数据库表中查询商品信息。

非常感谢唯品会资深运维工程师阳祥义，帮助提供了今天的案例。

## 案例准备

本次案例还是基于 Ubuntu 18.04，同样适用于其他的 Linux 系统。我使用的案例环境如下所示：

机器配置：2 CPU，8GB 内存

预先安装 docker、sysstat、git、make 等工具，如 `apt install docker.io sysstat make git`

其中，docker 和 sysstat 已经用过很多次，这里不再赘述；git 用来拉取本次案例所需脚本，这些脚本存储在 Github 代码仓库中；最后的 make 则是一个常用构建工具，这里用来运行今天的案例。

案例总共由三个容器组成，包括一个 MySQL 数据库应用、一个商品搜索应用以及一个数据处理的应用。其中，商品搜索应用以 HTTP 的形式提供了一个接口：

`/`：返回 Index Page；

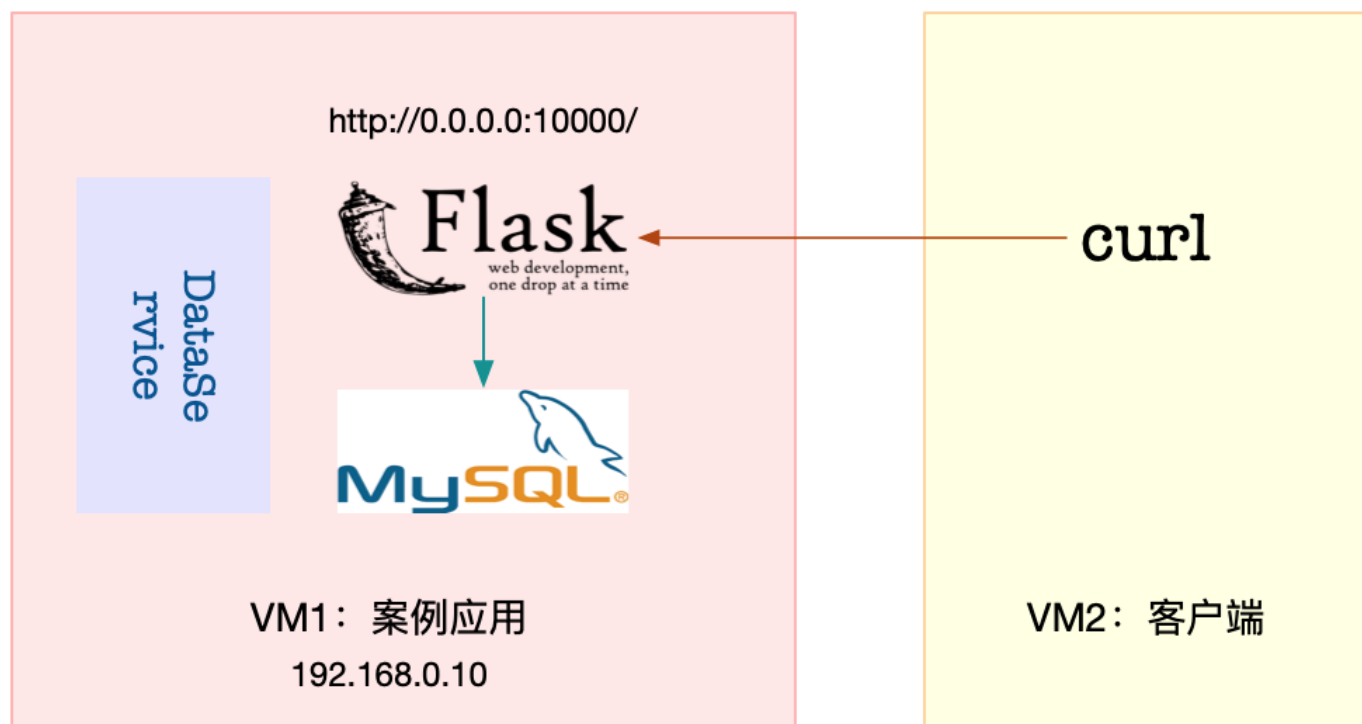
`/db/insert/products/`：插入指定数量的商品信息；

`/products/`：查询指定商品的信息，并返回处理时间。

由于应用比较多，为了方便你运行它们，我把它们同样打包成了几个 Docker 镜像，并推送到了 Github 上。这样，你只需要运行几条命令，就可以启动了。

今天的案例需要两台虚拟机，其中一台作为案例分析的目标机器，运行 Flask 应用，它的 IP 地址是 192.168.0.10；另一台则是作为客户端，请求单词的热度。我画了一张图表示它

们的关系。



接下来，打开两个终端，分别 SSH 登录到这两台虚拟机中，并在第一台虚拟机中安装上述工具。

跟以前一样，案例中所有命令都默认以 root 用户运行，如果你是用普通用户身份登陆系统，请运行 `sudo su root` 命令切换到 root 用户。

到这里，准备工作就完成了。接下来，我们正式进入操作环节。

## 案例分析

首先，我们在第一个终端中执行下面命令，拉取本次案例所需脚本：

复制代码


```
1 $ git clone https://github.com/feiskyer/linux-perf-examples
2 $ cd linux-perf-examples/mysql-slow
```

接着，执行下面的命令，运行本次的目标应用。正常情况下，你应该可以看到下面的输出：

复制代码

```
1 # 注意下面的随机字符串是容器 ID，每次运行均会不同，并且你不需要关注它，因为我们只会用到名字
2 $ make run
3 docker run --name=mysql -itd -p 10000:80 -m 800m feisky/mysql:5.6
4 WARNING: Your kernel does not support swap limit capabilities or the cgroup is not mount
5 4156780da5be0b9026bcf27a3fa56abc15b8408e358fa327f472bcc5add4453f
6 docker run --name=dataservice -itd --privileged feisky/mysql-dataservice
7 f724d0816d7e47c0b2b1ff701e9a39239cb9b5ce70f597764c793b68131122bb
8 docker run --name=app --network=container:mysql -itd feisky/mysql-slow
9 81d3392ba25bb8436f6151662a13ff6182b6bc6f2a559fc2e9d873cd07224ab6
```


然后，再运行 `docker ps` 命令，确认三个容器都处在运行（Up）状态：

 复制代码

```
1 $ docker ps
2 CONTAINER ID          IMAGE                COMMAND              CREATED
3 9a4e3c580963          feisky/mysql-slow   "python /app.py"     42 seconds ago
4 2a47aab18082          feisky/mysql-dataservice "python /dataservice..." 46 seconds ago
5 4c3ff7b24748          feisky/mysql:5.6    "docker-entrypoint.s..." 47 seconds ago
```


MySQL 数据库的启动过程，需要做一些初始化工作，这通常需要花费几分钟时间。你可以运行 `docker logs` 命令，查看它的启动过程。

当你看到下面这个输出时，说明 MySQL 初始化完成，可以接收外部请求了：

 复制代码

```
1 $ docker logs -f mysql
2 ...
3 ... [Note] mysqld: ready for connections.
4 Version: '5.6.42-log' socket: '/var/run/mysqld/mysqld.sock' port: 3306 MySQL Communi
```

而商品搜索应用则是在 10000 端口监听。你可以按 `Ctrl+C`，停止 `docker logs` 命令；然后，执行下面的命令，确认它也已经正常运行。如果一切正常，你会看到 Index Page 的输出：

 复制代码

```
1 $ curl http://127.0.0.1:10000/
```

接下来，运行 `make init` 命令，初始化数据库，并插入 10000 条商品信息。这个过程比较慢，比如在我的机器中，就花了十几分钟时间。耐心等待一段时间后，你会看到如下的输出：

[📄 复制代码](#)

```
1 $ make init
2 docker exec -i mysql mysql -uroot -P3306 < tables.sql
3 curl http://127.0.0.1:10000/db/insert/products/10000
4 insert 10000 lines
```

接着，我们切换到第二个终端，访问一下商品搜索的接口，看看能不能找到想要的商品。执行如下的 `curl` 命令：

[📄 复制代码](#)

```
1 $ curl http://192.168.0.10:10000/products/geektime
2 Got data: () in 15.364538192749023 sec
```

稍等一会儿，你会发现，这个接口返回的是空数据，而且处理时间超过 15 秒。这么慢的响应速度让人无法忍受，到底出了什么问题呢？

既然今天用了 MySQL，你估计会猜到是慢查询的问题。

不过别急，在具体分析前，为了避免在分析过程中客户端的请求结束，我们把 `curl` 命令放到一个循环里执行。同时，为了避免给系统过大压力，我们设置在每次查询后，都先等待 5 秒，然后再开始新的请求。

所以，你可以在终端二中，继续执行下面的命令：


[📄 复制代码](#)

```
1 $ while true; do curl http://192.168.0.10:10000/products/geektime; sleep 5; done
```

接下来，重新回到终端一中，分析接口响应速度慢的原因。不过，重回终端一后，你会发现系统响应也明显变慢了，随便执行一个命令，都得停顿一会儿才能看到输出。

这跟上一节的现象很类似，看来，我们还是得观察一下系统的资源使用情况，比如 CPU、内存和磁盘 I/O 等的情况。


首先，我们在终端一执行 top 命令，分析系统的 CPU 使用情况：

 复制代码

```
1 $ top
2 top - 12:02:15 up 6 days, 8:05, 1 user, load average: 0.66, 0.72, 0.59
3 Tasks: 137 total, 1 running, 81 sleeping, 0 stopped, 0 zombie
4 %Cpu0 : 0.7 us, 1.3 sy, 0.0 ni, 35.9 id, 62.1 wa, 0.0 hi, 0.0 si, 0.0 st
5 %Cpu1 : 0.3 us, 0.7 sy, 0.0 ni, 84.7 id, 14.3 wa, 0.0 hi, 0.0 si, 0.0 st
6 KiB Mem : 8169300 total, 7238472 free, 546132 used, 384696 buff/cache
7 KiB Swap: 0 total, 0 free, 0 used. 7316952 avail Mem
8
9  PID USER      PR  NI   VIRT   RES    SHR S  %CPU %MEM    TIME+  COMMAND
10 27458 999      20   0 833852 57968 13176 S   1.7  0.7   0:12.40 mysqld
11 27617 root     20   0 24348  9216  4692 S   1.0  0.1   0:04.40 python
12 1549  root     20   0 236716 24568 9864 S   0.3  0.3 51:46.57 python3
13 22421 root     20   0      0      0      0 I   0.3  0.0   0:01.16 kworker/u
```

观察 top 的输出，我们发现，两个 CPU 的 iowait 都比较高，特别是 CPU0，iowait 已经超过 60%。而具体到各个进程，CPU 使用率并不高，最高的也只有 1.7%。


既然 CPU 的嫌疑不大，那问题应该还是出在了 I/O 上。我们仍然在第一个终端，按下 Ctrl+C，停止 top 命令；然后，执行下面的 iostat 命令，看看有没有 I/O 性能问题：

 复制代码

```
1 $ iostat -d -x 1
2 Device            r/s      w/s      kB/s      kB/s      rrqm/s      wrqm/s      %rrqm      %wrqm  r_await
3 ...
4 sda                273.00    0.00  32568.00    0.00    0.00      0.00    0.00    0.00    7.9
```

iostat 的输出你应该非常熟悉。观察这个界面，我们发现，磁盘 sda 每秒的读数据为 32 MB，而 I/O 使用率高达 97%，接近饱和，这说明，磁盘 sda 的读取确实碰到了性能瓶颈。

那要怎么知道，这些 I/O 请求到底是哪些进程导致的呢？当然可以找我们的老朋友，pidstat。接下来，在终端一中，按下 Ctrl+C 停止 iostat 命令，然后运行下面的 pidstat 命令，观察进程的 I/O 情况：

 复制代码

```
1 # -d 选项表示展示进程的 I/O 情况
2 $ pidstat -d 1
3 12:04:11      UID      PID   kB_rd/s   kB_wr/s kB_ccwr/s iodelay  Command
4 12:04:12      999    27458  32640.00     0.00     0.00      0  mysqld
5 12:04:12       0    27617    4.00     4.00     0.00      3  python
6 12:04:12       0    27864    0.00     4.00     0.00      0  systemd-journal
```


从 pidstat 的输出可以看到，PID 为 27458 的 mysqld 进程正在进行大量的读，而且读取速度是 32 MB/s，跟刚才 iostat 的发现一致。两个结果一对比，我们自然就找到了磁盘 I/O 瓶颈的根源，即 mysqld 进程。

不过，这事儿还没完。我们自然要怀疑一下，为什么 mysqld 会去读取大量的磁盘数据呢？按照前面猜测，我们提到过，这有可能是个慢查询问题。

可是，回想一下，慢查询的现象大多是 CPU 使用率高（比如 100%），但这里看到的却是 I/O 问题。看来，这并不是一个单纯的慢查询问题，我们有必要分析一下 MySQL 读取的数据。

要分析进程的数据读取，当然还要靠上一节用到过的 strace+ lsof 组合。

接下来，还是在终端一中，执行 strace 命令，并且指定 mysqld 的进程号 27458。我们知道，MySQL 是一个多线程的数据库应用，为了不漏掉这些线程的数据读取情况，你要记得在执行 strace 命令时，加上 -f 参数：


 复制代码

```
1 $ strace -f -p 27458
2 [pid 28014] read(38, "934EiwT363aak7VtqF1mHGa4LL4Dhbks"... , 131072) = 131072
```



```
3 [pid 28014] read(38, "hSs7KBDepBqA6m4ce6i6iUfFTeG90t9z"... , 20480) = 20480
4 [pid 28014] read(38, "NRhRjCSsLLBjTfdqiBRLvN9K6FRfqqlm"... , 131072) = 131072
5 [pid 28014] read(38, "AKgsik4BilLb7y60kwQUjjqGeCTQTaRl"... , 24576) = 24576
6 [pid 28014] read(38, "hFMHx7FzUSqfFI22fQxwCpSnDmRjamaW"... , 131072) = 131072
7 [pid 28014] read(38, "ajUzLmKqivcDJSkiw7QWf2ETLgvQIpfc"... , 20480) = 20480
```

观察一会，你会发现，线程 28014 正在读取大量数据，且读取文件的描述符编号为 38。这儿的 38 又对应着哪个文件呢？我们可以执行下面的 lsof 命令，并且指定线程号 28014，具体查看这个可疑线程和可疑文件：

 复制代码

```
1 $ lsof -p 28014
```

奇怪的是，lsof 并没有给出任何输出。实际上，如果你查看 lsof 命令的返回值，就会发现，这个命令的执行失败了。

我们知道，在 SHELL 中，特殊标量 \$? 表示上一条命令退出时的返回值。查看这个特殊标量，你会发现它的返回值是 1。可是别忘了，在 Linux 中，返回值为 0，才表示命令执行成功。返回值为 1，显然表明执行失败。

 复制代码

```
1 $ echo $?
2 1
```

为什么 lsof 命令执行失败了呢？这里希望你暂停往下，自己先思考一下原因。记住我的那句话，遇到现象解释不了，先去查查工具文档。

事实上，通过查询 lsof 的文档，你会发现，-p 参数需要指定进程号，而我们刚才传入的是线程号，所以 lsof 失败了。你看，任何一个细节都可能成为性能分析的“拦路虎”。

回过头我们看，mysqld 的进程号是 27458，而 28014 只是它的一个线程。而且，如果你观察一下 mysqld 进程的线程，你会发现，mysqld 其实还有很多正在运行的其他线程：



```

1 # -t 表示显示线程，-a 表示显示命令行参数
2 $ pstree -t -a -p 27458
3 mysqld,27458 --log_bin=on --sync_binlog=1
4 ...
5   └─{mysqld},27922
6   └─{mysqld},27923
7   └─{mysqld},28014

```

找到了原因，lsof 的问题就容易解决了。把线程号换成进程号，继续执行 lsof 命令：

```

1 $ lsof -p 27458
2 COMMAND  PID USER  FD   TYPE DEVICE SIZE/OFF NODE NAME
3 ...
4 mysqld   27458      999   38u   REG    8,1 512440000 2601895 /var/lib/mysql/test/products

```

这次我们得到了 lsof 的输出。从输出中可以看到，mysqld 进程确实打开了大量文件，而根据文件描述符（FD）的编号，我们知道，描述符为 38 的是一个路径为 /var/lib/mysql/test/products.MYD 的文件。这里注意，38 后面的 u 表示，mysqld 以读写的方式访问文件。

看到这个文件，熟悉 MySQL 的你可能笑了：

MYD 文件，是 MyISAM 引擎用来存储表数据的文件；

文件名就是数据表的名字；

而这个文件的父目录，也就是数据库的名字。

换句话说，这个文件告诉我们，mysqld 在读取数据库 test 中的 products 表。

实际上，你可以执行下面的命令，查看 mysqld 在管理数据库 test 时的存储文件。不过要注意，由于 MySQL 运行在容器中，你需要通过 docker exec 到容器中查看：

```

1 $ docker exec -it mysql ls /var/lib/mysql/test/

```

从这里你可以发现，`/var/lib/mysql/test/` 目录中有四个文件，每个文件的作用分别是：

MYD 文件用来存储表的数据；


MYI 文件用来存储表的索引；

frm 文件用来存储表的元信息（比如表结构）；

opt 文件则用来存储数据库的元信息（比如字符集、字符校验规则等）。

当然，看到这些，你可能还有一个疑问，那就是，这些文件到底是不是 `mysqld` 正在使用的数据库文件呢？有没有可能是不再使用的旧数据呢？其实，这个很容易确认，查一下 `mysqld` 配置的数据路径即可。

你可以在终端一中，继续执行下面的命令：


 复制代码

```
1 $ docker exec -i -t mysql mysql -e 'show global variables like "%datadir%";'
2 +-----+-----+
3 | Variable_name | Value          |
4 +-----+-----+
5 | datadir       | /var/lib/mysql/ |
6 +-----+-----+
```

这里可以看到，`/var/lib/mysql/` 确实是 `mysqld` 正在使用的数据存储目录。刚才分析得出的数据库 `test` 和数据表 `products`，都是正在使用。

注：其实 `Isof` 的结果已经可以确认，它们都是 `mysqld` 正在访问的文件。再查询 `datadir`，只是想换一个思路，进一步确认一下。


既然已经找出了数据库和表，接下来要做的，就是弄清楚数据库中正在执行什么样的 SQL 了。我们继续在终端一中，运行下面的 `docker exec` 命令，进入 MySQL 的命令行界面：

 复制代码

```
1 $ docker exec -i -t mysql mysql
2 ...
3
4 Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
5
6 mysql>
```

下一步你应该可以想到，那就是在 MySQL 命令行界面中，执行 `show processlist` 命令，来查看当前正在执行的 SQL 语句。

不过，为了保证 SQL 语句不截断，这里我们可以执行 `show full processlist` 命令。如果一切正常，你应该可以看到如下输出：

 复制代码

```
1 mysql> show full processlist;
2 +-----+-----+-----+-----+-----+-----+-----+-----+
3 | Id | User | Host                | db   | Command | Time | State          | Info
4 +-----+-----+-----+-----+-----+-----+-----+-----+
5 | 27 | root | localhost           | test | Query   | 0    | init           | show full proces
6 | 28 | root | 127.0.0.1:42262     | test | Query   | 1    | Sending data   | select * from pro
7 +-----+-----+-----+-----+-----+-----+-----+-----+
8 2 rows in set (0.00 sec)
```

这个输出中，

`db` 表示数据库的名字；

`Command` 表示 SQL 类型；

`Time` 表示执行时间；

`State` 表示状态；

而 `Info` 则包含了完整的 SQL 语句。


多执行几次 `show full processlist` 命令，你可看到 `select * from products where productName= 'geektime'` 这条 SQL 语句的执行时间比较长。

再回忆一下，案例开始时，我们在终端二查询的产品名称

<http://192.168.0.10:10000/products/geektime>，其中的 geektime 也符合这条查询语句的条件。

我们知道，MySQL 的慢查询问题，很可能是没有利用好索引导致的，那这条查询语句是不是这样呢？我们又该怎么确认，查询语句是否利用了索引呢？

其实，MySQL 内置的 explain 命令，就可以帮你解决这个问题。继续在 MySQL 终端中，运行下面的 explain 命令：

 复制代码

```
1 # 切换到 test 库
2 mysql> use test;
3 # 执行 explain 命令
4 mysql> explain select * from products where productName='geektime';
5 +-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
6 | id | select_type | table      | type | possible_keys | key  | key_len | ref  | rows |
7 +-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
8 | 1  | SIMPLE      | products  | ALL  | NULL          | NULL | NULL    | NULL | 10000 |
9 +-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
10 1 row in set (0.00 sec)
```

观察这次的输出。这个界面中，有几个比较重要的字段需要你注意，我就以这个输出为例，分别解释一下：

select\_type 表示查询类型，而这里的 SIMPLE 表示此查询不包括 UNION 查询或者子查询；

table 表示数据表的名字，这里是 products；

type 表示查询类型，这里的 ALL 表示全表查询，但索引查询应该是 index 类型才对；

possible\_keys 表示可能选用的索引，这里是 NULL；


key 表示确切会使用的索引，这里也是 NULL；

rows 表示查询扫描的行数，这里是 10000。

根据这些信息，我们可以确定，这条查询语句压根儿没有使用索引，所以查询时，会扫描全表，并且扫描行数高达 10000 行。响应速度那么慢也就难怪了。


走到这一步，你应该很容易想到优化方法，没有索引那我们就自己建立，给 productName 建立索引就可以了。不过，增加索引前，你需要先弄清楚，这个表结构到底长什么样儿。

执行下面的 MySQL 命令，查询 products 表的结构，你会看到，它只有一个 id 主键，并不包括 productName 的索引：

 复制代码

```
1 mysql> show create table products;
2 ...
3 | products | CREATE TABLE `products` (
4   `id` int(11) NOT NULL,
5   `productCode` text NOT NULL COMMENT '产品代码',
6   `productName` text NOT NULL COMMENT '产品名称',
7   ...
8   PRIMARY KEY (`id`)
9 ) ENGINE=MyISAM DEFAULT CHARSET=utf8 ROW_FORMAT=DYNAMIC |
10 ...
```

接下来，我们就可以给 productName 建立索引了，也就是执行下面的 CREATE INDEX 命令：

 复制代码

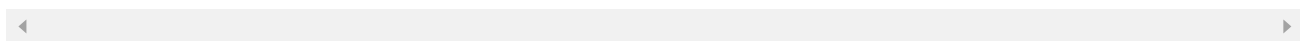
```
1 mysql> CREATE INDEX products_index ON products (productName);
2 ERROR 1170 (42000): BLOB/TEXT column 'productName' used in key specification without a l
```

不过，醒目的 ERROR 告诉我们，这条命令运行失败了。根据错误信息，productName 是一个 BLOB/TEXT 类型，需要设置一个长度。所以，想要创建索引，就必须为 productName 指定一个前缀长度。

那前缀长度设置为多大比较合适呢？这里其实有专门的算法，即通过计算前缀长度的选择性，来确定索引的长度。不过，我们可以稍微简化一下，直接使用一个固定数值（比如 64），执行下面的命令创建索引：


 复制代码

```
1 mysql> CREATE INDEX products_index ON products (productName(64));
2 Query OK, 10000 rows affected (14.45 sec)
```

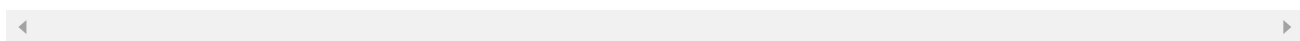


现在可以看到，索引已经建好了。能做的都做完了，最后就该检查一下，性能问题是否已经解决了。

我们切换到终端二中，查看还在执行的 curl 命令的结果：

 复制代码

```
1 Got data: ()in 15.383180141448975 sec
2 Got data: ()in 15.384996891021729 sec
3 Got data: ()in 0.0021054744720458984 sec
4 Got data: ()in 0.003951072692871094 sec
```



显然，查询时间已经从 15 秒缩短到了 3 毫秒。看来，没有索引果然就是这次性能问题的罪魁祸首，解决了索引，就解决了查询慢的问题。

## 案例思考


到这里，商品搜索应用查询慢的问题已经完美解决了。但是，对于这个案例，我还有一点想说明一下。

不知道你还记不记得，案例开始时，我们启动的几个容器应用。除了 MySQL 和商品搜索应用外，还有一个 DataService 应用。为什么这个案例开始时，要运行一个看起来毫不相关的应用呢？

实际上，DataService 是一个严重影响 MySQL 性能的干扰应用。抛开上述索引优化方法不说，这个案例还有一种优化方法，也就是停止 DataService 应用。


接下来，我们就删除数据库索引，回到原来的状态；然后停止 DataService 应用，看看优化效果如何。

首先，我们在终端二中停止 curl 命令，然后回到终端一中，执行下面的命令删除索引：

 复制代码

```
1 # 删除索引
2 $ docker exec -i -t mysql mysql
3
4 mysql> use test;
5 mysql> DROP INDEX products_index ON products;
```

接着，在终端二中重新运行 curl 命令。当然，这次你会发现，处理时间又变慢了：

 复制代码


```
1 $ while true; do curl http://192.168.0.10:10000/products/geektime; sleep 5; done
2 Got data: ()in 16.884345054626465 sec
```

接下来，再次回到终端一中，执行下面的命令，停止 DataService 应用：

 复制代码

```
1 # 停止 DataService 应用
2 $ docker rm -f dataservice
```

最后，我们回到终端二中，观察 curl 的结果：

 复制代码


```
1 Got data: ()in 16.884345054626465 sec
2 Got data: ()in 15.238174200057983 sec
3 Got data: ()in 0.12604427337646484 sec
4 Got data: ()in 0.1101069450378418 sec
5 Got data: ()in 0.11235237121582031 sec
```

果然，停止 DataService 后，处理时间从 15 秒缩短到了 0.1 秒，虽然比不上增加索引后的 3 毫秒，但相对于 15 秒来说，优化效果还是非常明显的。

那么，这种情况下，还有没有 I/O 瓶颈了呢？



我们切换到终端一中，运行下面的 `vmstat` 命令（注意不是 `iostat`，稍后解释原因），观察 I/O 的变化情况：

 复制代码

```
1 $ vmstat 1
2 procs -----memory----- ---swap-- -----io----- -system-- -----cpu-----
3  r  b   swpd   free   buff  cache   si   so    bi   bo    in   cs us sy id wa st
4  0  1       0 6809304   1368 856744    0    0 32640    0   52  478  1  0 50 49  0
5  0  1       0 6776620   1368 889456    0    0 32640    0   33  490  0  0 50 49  0
6  0  0       0 6747540   1368 918576    0    0 29056    0   42  568  0  0 56 44  0
7  0  0       0 6747540   1368 918576    0    0    0    0   40  141  1  0 100  0  0
8  0  0       0 6747160   1368 918576    0    0    0    0   40  148  0  1 99  0  0
```

你可以看到，磁盘读（`bi`）和 `iowait`（`wa`）刚开始还是挺大的，但没过多久，就都变成了 0。换句话说，I/O 瓶颈消失了。

这是为什么呢？原因先留个悬念，作为今天的思考题。

回过头来解释一下刚刚的操作，在查看 I/O 情况时，我并没用 `iostat` 命令，而是用了 `vmstat`。其实，相对于 `iostat` 来说，`vmstat` 可以同时提供 CPU、内存和 I/O 的使用情况。

在性能分析过程中，能够综合多个指标，并结合系统的工作原理进行分析，对解释性能现象通常会有意想不到的帮助。

## 小结

今天我们分析了一个商品搜索的应用程序。我们先是通过 `top`、`iostat` 分析了系统的 CPU 和磁盘使用情况，发现了磁盘的 I/O 瓶颈。

接着，我们借助 `pidstat`，发现瓶颈是 `mysqld` 导致的。紧接着，我们又通过 `strace`、`lsof`，找出了 `mysqld` 正在读的文件。同时，根据文件的名称和路径，我们找出了 `mysqld` 正在操作的数据库和数据表。综合这些信息，我们判断，这是一个没有利用索引导致的慢查询问题。

于是，我们登录到 MySQL 命令行终端，用数据库分析工具进行验证，发现 MySQL 查询语句访问的字段，果然没有索引。所以，增加索引，就可以解决案例的性能问题了。

## 思考

最后，给你留一个思考题，也是我在案例最后部分提到过的，停止 DataService 后，商品搜索应用的处理时间，从 15 秒缩短到了 0.1 秒。这是为什么呢？

我给个小小的提示。你可以先查看 [dataservice.py](#) 的[源码](#)，你会发现，DataService 实际上是在读写一个仅包括 “data” 字符串的小文件。不过在读取文件前，它会先把 `/proc/sys/vm/drop_caches` 改成 1。

还记得这个操作有什么作用吗？如果不记得，可以用 `man` 查询 `proc` 文件系统的文档。

欢迎在留言区和我讨论，也欢迎把这篇文章分享给你的同事、朋友。我们一起在实战中演练，在交流中进步。

 极客时间

# Linux 性能优化实战

---

## 10 分钟帮你找到系统瓶颈

---



倪朋飞 微软资深工程师  
Kubernetes 项目维护者

新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 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 | 案例篇：为什么我的磁盘I/O延迟很高？

下一篇 28 | 案例篇：Docker 容器严重卡顿，如何解决？

## 精选留言 (23)

写留言



ninuxer

2019-01-23

22

打卡day29

echo 1>/proc/sys/vm/drop\_caches表示释放pagecache，也就是文件缓存，而mysql读书的数据就是文件缓存，dataservice不停的释放文件缓存，就导致MySQL都无法利用磁盘缓存，也就慢了~

展开

作者回复: 赞，正解



某、人

2019-01-24

6

非常赞，这篇案例很好的展示了怎么从操作系统层面去排查慢查询的问题



rm -rf ...

2019-04-04

1

有个疑问，老师之前不是说过数据库有自己一套缓存机制吗？为何删除cached也会影响到mysql的读写呢

作者回复: 数据库的缓存还要看选择了存储引擎，MyISAM不会缓存数据



大坏狐狸

2019-02-11

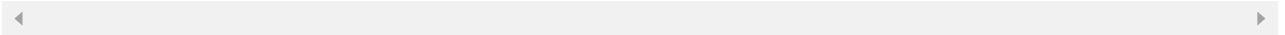
1

这个地方因为之前了解过，所以来打个卡，这个算法在高性能mysql里有介绍，叫最优前缀。

有时候需要索引很长的字符列，这会让索引变得大且慢。通常可以索引开始的部分字符，这样可以大大节约索引空间，从而提高索引效率。但这样也会降低索引的选择性。索引的选择性是指不重复的索引值（也称为基数，cardinality）和数据表的记录总数的比值，范...

展开 ▾

作者回复: 赞, 谢谢分享详细的前缀选择算法



**guoew**

2019-01-25

👍 1

以前遇到数据库慢的问题都是直接进数据库show processlist , 没有一个循序渐进的过程。感谢老师



**李逍遥**

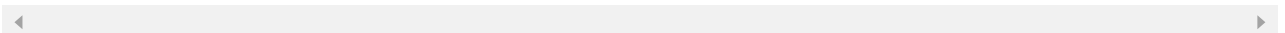
2019-01-23

👍 1

mysql打开慢查询日志, 排查SQL性能问题更方便

展开 ▾

作者回复: 嗯嗯, 慢查询推荐总是开启



**LYy**

2019-05-18

👍

硬核 赞👍

展开 ▾



**无名老卒**

2019-04-25

👍

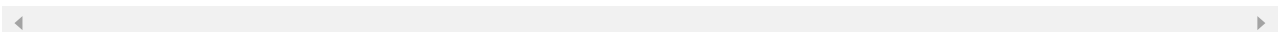
实例插入了3W条数据, 这样子才使得get data的时间变大了一些, 后面增加了索引之后, 就变正常了。

```

```
root@fdm:~#while ;;do curl http://192.168.254.131:10000/products/geektime;done
Got data: () in 2.335655689239502 sec...
```

展开 ▾

作者回复: 插入更多的数据试试?





王伟

2019-04-23



打卡

展开 ▾



大飞

2019-04-03



打卡

展开 ▾



如果

2019-03-13



DAY28，打卡

展开 ▾



陈云卿

2019-02-06



打卡day29，之前在工作中遇到过 释放缓存的情况，但是也会发现echo 1在某些情况下没有办法有效地释放缓存，这会是什么原因引起的呢？

展开 ▾

作者回复: 试试 echo 3



我来也

2019-01-31



[D28打卡]

居然漏打卡了，回来补上。

平常偶尔手动释放缓存还有可能，但把这个操作放在需要频繁执行的代码中就有点不科学了。

作者回复: 嗯嗯，要看场景的。比如，对一个定期评估系统性能的工具，为了获得准确的指标，就有可能这么做（不过这种应用跟其他业务一般都是分开部署）



唯美

2019-01-27



打卡Day27

展开 ▾



小老鼠

2019-01-26



即停止DataService，又增加索3引是不是性能会更好？

展开 ▾

作者回复: 是的，利用缓存后性能会更好



安小依

2019-01-24



老师好，上次留言说 Java 无法 strace 打日志的问题解决了，原来 strace 必须跟上 -f 选项才可以，最终定位到当疯狂 System.out.println 的时候，原来是写到 pipe 管道里边去了：

```
java 2794 zk 1w FIFO 0,12 0t0 3626492 pipe...
```

展开 ▾

作者回复: pipe 是一个管道，而不是文件，可以用 lsof 找一下 pipe 的对端进程



划时代

2019-01-24



打卡

展开 ▾



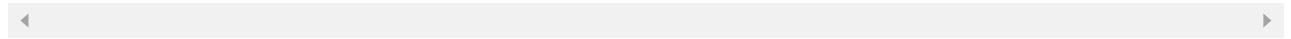
往事随风，...

2019-01-23



centos下执行这个案例，不会出现查询慢的情况，很正常的结束

作者回复: 增大初始化的数据量（比如增大10倍）再试试呢？



往事随风, ...

2019-01-23

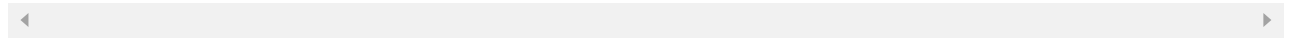


curl http://192.168.193.132:10000/products/geektime

Got data: () in 0.4080049991607666 sec

时间很快并没有出现长时间查询的情况

作者回复: 看来你的系统配置比较高，增加下初始化时的数据量（比如增加10倍）再试试？



往事随风, ...

2019-01-23



make run报错如下：

flag provided but not defined: --network

mysql-slow这个没法运行起来

展开 ∨

作者回复: mysql 没有运行起来，运行 docker logs mysql 看看报了什么错误？

