

## 29 | 案例篇：Redis响应严重延迟，如何解决？

2019-01-25 倪朋飞



朗读：冯永吉

时长15:28 大小14.17M



你好，我是倪朋飞。

上一节，我们一起分析了一个基于 MySQL 的商品搜索案例，先来回顾一下。

在访问商品搜索接口时，我们发现接口的响应特别慢。通过对系统 CPU、内存和磁盘 I/O 等资源使用情况的分析，我们发现这时出现了磁盘的 I/O 瓶颈，并且正是案例应用导致的。

接着，我们借助 pidstat，发现罪魁祸首是 mysqld 进程。我们又通过 strace、lsof，找出了 mysqld 正在读的文件。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，我们猜测这是一个没利用索引导致的慢查询问题。

为了验证猜测，我们到 MySQL 命令行终端，使用数据库分析工具发现，案例应用访问的字段果然没有索引。既然猜测是正确的，那增加索引后，问题就自然解决了。

从这个案例你会发现，MySQL 的 MyISAM 引擎，主要依赖系统缓存加速磁盘 I/O 的访问。可如果系统中还有其他应用同时运行，MyISAM 引擎很难充分利用系统缓存。缓存可能会被其他应用程序占用，甚至被清理掉。

所以，一般我并不建议，把应用程序的性能优化完全建立在系统缓存上。最好能在应用程序的内部分配内存，构建完全自主控制的缓存；或者使用第三方的缓存应用，比如 Memcached、Redis 等。

Redis 是最常用的键值存储系统之一，常用作数据库、高速缓存和消息队列代理等。Redis 基于内存来存储数据，不过，为了保证在服务器异常时数据不丢失，很多情况下，我们要为它配置持久化，而这就可能会引发磁盘 I/O 的性能问题。

今天，我就带你一起来分析一个利用 Redis 作为缓存的案例。这同样是一个基于 Python Flask 的应用程序，它提供了一个查询缓存的接口，但接口的响应时间比较长，并不能满足线上系统的要求。

非常感谢携程系统研发部资深后端工程师董国星，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`

今天的案例由 Python 应用 +Redis 两部分组成。其中，Python 应用是一个基于 Flask 的应用，它会利用 Redis，来管理应用程序的缓存，并对外提供三个 HTTP 接口：

`/`：返回 hello redis；

`/init/`：插入指定数量的缓存数据，如果不指定数量，默认的是 5000 条；

缓存的键格式为 uuid；

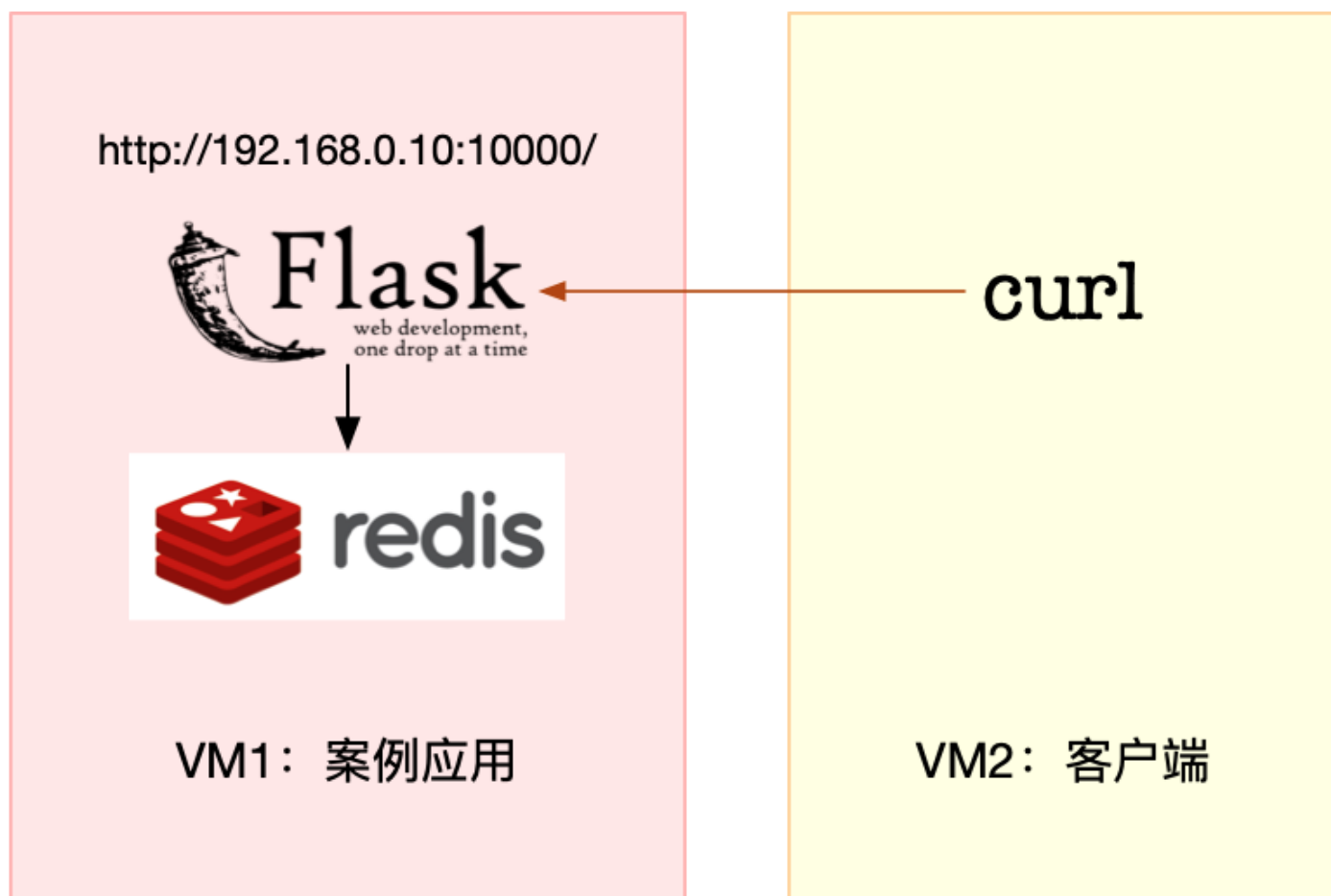
缓存的值为 good、bad 或 normal 三者之一

`/get_cache/<type_name>`：查询指定值的缓存数据，并返回处理时间。其中，`type_name` 参数只支持 good, bad 和 normal（也就是找出具有相同 value 的 key 列

表)。

由于应用比较多，为了方便你运行，我把它们打包成了两个 Docker 镜像，并推送到了 [Github](#) 上。这样你就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，其中一台用作案例分析的目标机器，运行 Flask 应用，它的 IP 地址是 192.168.0.10；而另一台作为客户端，请求缓存查询接口。我画了一张图来表示它们的关系。




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，执行下面的命令，运行本次案例要分析的目标应用。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
2 $ docker run --name=redis -itd -p 10000:80 feisky/redis-server
3 ec41cb9e4dd5cb7079e1d9f72b7cee7de67278dbd3bd0956b4c0846bffa211803
4 $ docker run --name=app --network=container:redis -itd feisky/redis-app
5 2c54eb252d0552448320d9155a2618b799a1e71d7289ec7277a61e72a9de5fd0
```


然后，再运行 `docker ps` 命令，确认两个容器都处于运行（Up）状态：

 复制代码

```
1 $ docker ps
2 CONTAINER ID          IMAGE                COMMAND              CREATED              STATUS
3 2c54eb252d05          feisky/redis-app    "python /app.py"     48 seconds ago      Up
4 ec41cb9e4dd5          feisky/redis-server "docker-entrypoint.s..." 49 seconds ago      Up
5
```


今天的应用在 10000 端口监听，所以你可以通过 <http://192.168.0.10:10000>，来访问前面提到的三个接口。

比如，我们切换到第二个终端，使用 `curl` 工具，访问应用首页。如果你看到 `hello redis` 的输出，说明应用正常启动：

 复制代码


```
1 $ curl http://192.168.0.10:10000/
2 hello redis
```

接下来，继续在终端二中，执行下面的 `curl` 命令，来调用应用的 `/init` 接口，初始化 Redis 缓存，并且插入 5000 条缓存信息。这个过程比较慢，比如我的机器就花了十几分钟时间。耐心等待一会儿后，你会看到下面这行输出：

 复制代码

```
1 # 案例插入 5000 条数据，在实践时可以根据磁盘的类型适当调整，比如使用 SSD 时可以调大，而 HDD 可以调小
2 $ curl http://192.168.0.10:10000/init/5000
3 {"elapsed_seconds":30.26814079284668,"keys_initialized":5000}
```

继续执行下一个命令，访问应用的缓存查询接口。如果一切正常，你会看到如下输出：


 复制代码

```
1 $ curl http://192.168.0.10:10000/get_cache
2 {"count":1677,"data":["d97662fa-06ac-11e9-92c7-0242ac110002",...],"elapsed_seconds":10.545
```

我们看到，这个接口调用居然要花 10 秒！这么长的响应时间，显然不能满足实际的应用需求。

到底出了什么问题呢？我们还是要用前面学过的性能工具和原理，来找到这个瓶颈。

不过别急，同样为了避免分析过程中客户端的请求结束，在进行性能分析前，我们先要把 curl 命令放到一个循环里来执行。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/get_cache; done
```

接下来，再重新回到终端一，查找接口响应慢的“病因”。

最近几个案例的现象都是响应很慢，这种情况下，我们自然先会怀疑，是不是系统资源出现了瓶颈。所以，先观察 CPU、内存和磁盘 I/O 等的使用情况肯定不会错。

我们先在终端一中执行 top 命令，分析系统的 CPU 使用情况：

 复制代码


```
1 $ top
2 top - 12:46:18 up 11 days, 8:49, 1 user, load average: 1.36, 1.36, 1.04
3 Tasks: 137 total, 1 running, 79 sleeping, 0 stopped, 0 zombie
4 %Cpu0 : 6.0 us, 2.7 sy, 0.0 ni, 5.7 id, 84.7 wa, 0.0 hi, 1.0 si, 0.0 st
5 %Cpu1 : 1.0 us, 3.0 sy, 0.0 ni, 94.7 id, 0.0 wa, 0.0 hi, 1.3 si, 0.0 st
6 KiB Mem : 8169300 total, 7342244 free, 432912 used, 394144 buff/cache
7 KiB Swap: 0 total, 0 free, 0 used. 7478748 avail Mem
8
9  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM    TIME+  COMMAND
10  9181 root        20   0  193004  27304  8716 S   8.6   0.3   0:07.15 python
11  9085 systemd+  20   0   28352   9760  1860 D   5.0   0.1   0:04.34 redis-server

12  368 root        20   0      0      0      0 D   1.0   0.0   0:33.88 jbd2/sda1-8
13  149 root         0 -20      0      0      0 I   0.3   0.0   0:10.63 kworker/0:1H
```

观察 top 的输出可以发现，CPU0 的 iowait 比较高，已经达到了 84%；而各个进程的 CPU 使用率都不太高，最高的 python 和 redis-server，也分别只有 8% 和 5%。再看内存，总内存 8GB，剩余内存还有 7GB 多，显然内存也没啥问题。

综合 top 的信息，最有嫌疑的就是 iowait。所以，接下来还是要继续分析，是不是 I/O 问题。

还在第一个终端中，先按下 Ctrl+C，停止 top 命令；然后，执行下面的 iostat 命令，查看有没有 I/O 性能问题：

 复制代码

```
1 $ iostat -d -x 1
2 Device            r/s      w/s      rkB/s      wkB/s      rrqm/s      wrqm/s      %rrqm      %wrqm      r_await
3 ...
4 sda                0.00    492.00        0.00    2672.00        0.00     176.00        0.00    26.35        0.00
```

观察 iostat 的输出，我们发现，磁盘 sda 每秒的写数据（wkB/s）为 2.5MB，I/O 使用率（%util）是 0。看来，虽然有些 I/O 操作，但并没导致磁盘的 I/O 瓶颈。

排查一圈儿下来，CPU 和内存使用没问题，I/O 也没有瓶颈，接下来好像就没啥分析方向了？

碰到这种情况，还是那句话，反思一下，是不是又漏掉什么有用线索了。你可以先自己思考一下，从分析对象（案例应用）、系统原理和性能工具这三个方向下功夫，回忆它们的特性，查找现象的异常，再继续往下走。

回想一下，今天的案例问题是从 Redis 缓存中查询数据慢。对查询来说，对应的 I/O 应该是磁盘的读操作，但刚才我们用 iostat 看到的却是写操作。虽说 I/O 本身并没有性能瓶颈，但这里的磁盘写也是比较奇怪的。为什么会有磁盘写呢？那我们就得知道，到底是哪个进程在写磁盘。

要知道 I/O 请求来自哪些进程，还是要靠我们的老朋友 pidstat。在终端一中运行下面的 pidstat 命令，观察进程的 I/O 情况：



```

1 $ pidstat -d 1
2 12:49:35      UID      PID    kB_rd/s    kB_wr/s kB_ccwr/s iodelay  Command
3 12:49:36        0      368        0.00      16.00      0.00      86  jbd2/sda1-8
4 12:49:36     100     9085        0.00     636.00      0.00       1  redis-server

```

从 pidstat 的输出，我们看到，I/O 最多的进程是 PID 为 9085 的 redis-server，并且它刚好是在写磁盘。这说明，确实是 redis-server 在进行磁盘写。

当然，光找到读写磁盘的进程还不够，我们还要再用 strace+lsf 组合，看看 redis-server 到底在写什么。

接下来，还是在终端一中，执行 strace 命令，并且指定 redis-server 的进程号 9085：

```

1 # -f 表示跟踪子进程和子线程，-T 表示显示系统调用的时长，-tt 表示显示跟踪时间
2 $ strace -f -T -tt -p 9085
3 [pid 9085] 14:20:16.826131 epoll_pwait(5, [{EPOLLIN, {u32=8, u64=8}}], 10128, 65, NULL, 8
4 [pid 9085] 14:20:16.826301 read(8, "*2\r\n$3\r\nGET\r\n$41\r\nuuid:5b2e76cc-"..., 16384)
5 [pid 9085] 14:20:16.826477 read(3, 0x7fff366a5747, 1) = -1 EAGAIN (Resource temporarily u
6 [pid 9085] 14:20:16.826645 write(8, "$3\r\nbad\r\n", 9) = 9 <0.000173>
7 [pid 9085] 14:20:16.826907 epoll_pwait(5, [{EPOLLIN, {u32=8, u64=8}}], 10128, 65, NULL, 8
8 [pid 9085] 14:20:16.827030 read(8, "*2\r\n$3\r\nGET\r\n$41\r\nuuid:55862ada-"..., 16384)
9 [pid 9085] 14:20:16.827149 read(3, 0x7fff366a5747, 1) = -1 EAGAIN (Resource temporarily u
10 [pid 9085] 14:20:16.827285 write(8, "$3\r\nbad\r\n", 9) = 9 <0.000141>
11 [pid 9085] 14:20:16.827514 epoll_pwait(5, [{EPOLLIN, {u32=8, u64=8}}], 10128, 64, NULL, 8
12 [pid 9085] 14:20:16.827641 read(8, "*2\r\n$3\r\nGET\r\n$41\r\nuuid:53522908-"..., 16384)
13 [pid 9085] 14:20:16.827784 read(3, 0x7fff366a5747, 1) = -1 EAGAIN (Resource temporarily u
14 [pid 9085] 14:20:16.827945 write(8, "$4\r\ngood\r\n", 10) = 10 <0.000288>
15 [pid 9085] 14:20:16.828339 epoll_pwait(5, [{EPOLLIN, {u32=8, u64=8}}], 10128, 63, NULL, 8
16 [pid 9085] 14:20:16.828486 read(8, "*3\r\n$4\r\nSADD\r\n$4\r\n$36\r\n535"..., 163
17 [pid 9085] 14:20:16.828623 read(3, 0x7fff366a5747, 1) = -1 EAGAIN (Resource temporarily u
18 [pid 9085] 14:20:16.828760 write(7, "*3\r\n$4\r\nSADD\r\n$4\r\n$36\r\n535"..., 67
19 [pid 9085] 14:20:16.828970 fdatsync(7) = 0 <0.005415>
20 [pid 9085] 14:20:16.834493 write(8, ":1\r\n", 4) = 4 <0.000250>

```

观察一会儿，有没有发现什么有趣的现象呢？

事实上，从系统调用来看，epoll\_pwait、read、write、fdatsync 这些系统调用都比较频繁。那么，刚才观察到的写磁盘，应该就是 write 或者 fdatsync 导致的了。

接着再来运行 lsf 命令，找出这些系统调用的操作对象：

```

1 $ lsof -p 9085
2 redis-ser 9085 systemd-network 3r FIFO 0,12 0t0 15447970 pipe
3 redis-ser 9085 systemd-network 4w FIFO 0,12 0t0 15447970 pipe
4 redis-ser 9085 systemd-network 5u a_inode 0,13 0 10179 [eventpoll]
5 redis-ser 9085 systemd-network 6u sock 0,9 0t0 15447972 protocol: TCP
6 redis-ser 9085 systemd-network 7w REG 8,1 8830146 2838532 /data/appendonly.ac
7 redis-ser 9085 systemd-network 8u sock 0,9 0t0 15448709 protocol: TCP

```

现在你会发现，描述符编号为 3 的是一个 pipe 管道，5 号是 eventpoll，7 号是一个普通文件，而 8 号是一个 TCP socket。

结合磁盘写的现象，我们知道，只有 7 号普通文件才会产生磁盘写，而它操作的文件路径是 /data/appendonly.aof，相应的系统调用包括 write 和 fdatasync。

如果你对 Redis 的持久化配置比较熟，看到这个文件路径以及 fdatasync 的系统调用，你应该能想到，这对应着正是 Redis 持久化配置中的 appendonly 和 appendfsync 选项。很可能是因为它们配置不合理，导致磁盘写比较多。

接下来就验证一下这个猜测，我们可以通过 Redis 的命令行工具，查询这两个选项的配置。

继续在终端一中，运行下面的命令，查询 appendonly 和 appendfsync 的配置：

```

1 $ docker exec -it redis redis-cli config get 'append*'
2 1) "appendfsync"
3 2) "always"
4 3) "appendonly"
5 4) "yes"

```

从这个结果你可以发现，appendfsync 配置的是 always，而 appendonly 配置的是 yes。这两个选项的详细含义，你可以从 [Redis Persistence](#) 的文档中查到，这里我做一下简单介绍。

Redis 提供了两种数据持久化的方式，分别是快照和追加文件。

**快照方式**，会按照指定的时间间隔，生成数据的快照，并且保存到磁盘文件中。为了避免阻塞主进程，Redis 还会 fork 出一个子进程，来负责快照的保存。这种方式的性能好，无论是



备份还是恢复，都比追加文件好很多。

不过，它的缺点也很明显。在数据量大时，fork 子进程需要用到比较大的内存，保存数据也很耗时。所以，你需要设置一个比较长的时间间隔来应对，比如至少 5 分钟。这样，如果发生故障，你丢失的就是几分钟的数据。

**追加文件**，则是用在文件末尾追加记录的方式，对 Redis 写入的数据，依次进行持久化，所以它的持久化也更安全。

此外，它还提供了一个用 appendfsync 选项设置 fsync 的策略，确保写入的数据都落到磁盘中，具体选项包括 always、everysec、no 等。

always 表示，每个操作都会执行一次 fsync，是最为安全的方式；

everysec 表示，每秒钟调用一次 fsync，这样可以保证即使是最坏情况下，也只丢失 1 秒的数据；

而 no 表示交给操作系统来处理。

回忆一下我们刚刚看到的配置，appendfsync 配置的是 always，意味着每次写数据时，都会调用一次 fsync，从而造成比较大的磁盘 I/O 压力。

当然，你还可以用 strace，观察这个系统调用的执行情况。比如通过 -e 选项指定 fdatsync 后，你就会得到下面的结果：


 复制代码

```
1 $ strace -f -p 9085 -T -tt -e fdatsync
2 strace: Process 9085 attached with 4 threads
3 [pid 9085] 14:22:52.013547 fdatsync(7) = 0 <0.007112>
4 [pid 9085] 14:22:52.022467 fdatsync(7) = 0 <0.008572>
5 [pid 9085] 14:22:52.032223 fdatsync(7) = 0 <0.006769>
6 ...
7 [pid 9085] 14:22:52.139629 fdatsync(7) = 0 <0.008183>
```

从这里你可以看到，每隔 10ms 左右，就会有一次 fdatsync 调用，并且每次调用本身也要消耗 7~8ms。

不管哪种方式，都可以验证我们的猜想，配置确实不合理。这样，我们就找出了 Redis 正在进行写入的文件，也知道了产生大量 I/O 的原因。

不过，回到最初的疑问，为什么查询时会有磁盘写呢？按理来说不应该只有数据的读取吗？这就需要我们再来审查一下 `strace -f -T -tt -p 9085` 的结果。

 复制代码

```
1 read(8, "*2\r\n$3\r\nGET\r\n$41\r\nuuid:53522908-"..., 16384)
2 write(8, "$4\r\ngood\r\n", 10)
3 read(8, "*3\r\n$4\r\nSADD\r\n$4\r\n$36\r\n535"..., 16384)
4 write(7, "*3\r\n$4\r\nSADD\r\n$4\r\n$36\r\n535"..., 67)
5 write(8, ":1\r\n", 4)
```

细心的你应该记得，根据 `lsuf` 的分析，文件描述符编号为 7 的是一个普通文件 `/data/appendonly.aof`，而编号为 8 的是 TCP socket。而观察上面的内容，8 号对应的 TCP 读写，是一个标准的“请求 – 响应”格式，即：

从 socket 读取 GET uuid:53522908–... 后，响应 good；

再从 socket 读取 SADD good 535... 后，响应 1。

对 Redis 来说，SADD 是一个写操作，所以 Redis 还会把它保存到用于持久化的 `appendonly.aof` 文件中。

观察更多的 `strace` 结果，你会发现，每当 GET 返回 good 时，随后都会有一个 SADD 操作，这也就导致了，明明是查询接口，Redis 却有大量的磁盘写。

到这里，我们就找出了 Redis 写磁盘的原因。不过，在下最终结论前，我们还是要确认一下，8 号 TCP socket 对应的 Redis 客户端，到底是不是我们的案例应用。

我们可以给 `lsuf` 命令加上 `-i` 选项，找出 TCP socket 对应的 TCP 连接信息。不过，由于 Redis 和 Python 应用都在容器中运行，我们需要进入容器的网络命名空间内部，才能看到完整的 TCP 连接。

注意：下面的命令用到的 [nsenter](#) 工具，可以进入容器命名空间。如果你的系统没有安装，请运行下面命令安装 `nsenter`：

```
docker run --rm -v /usr/local/bin:/target jpetazzo/nsenter
```

还是在终端一中，运行下面的命令：

[复制代码](#)

```
1 # 由于这两个容器共享同一个网络命名空间，所以我们只需要进入 app 的网络命名空间即可
2 $ PID=$(docker inspect --format {{.State.Pid}} app)
3 # -i 表示显示网络套接字信息
4 $ nsenter --target $PID --net -- lsof -i
5 COMMAND      PID      USER      FD  TYPE  DEVICE  SIZE/OFF  NODE NAME
6 redis-ser 9085 systemd-network 6u IPv4 15447972      0t0  TCP localhost:6379 (LISTEN)
7 redis-ser 9085 systemd-network 8u IPv4 15448709      0t0  TCP localhost:6379->localhos
8 python 9181 root      3u IPv4 15448677      0t0  TCP *:http (LISTEN)
9 python 9181 root      5u IPv4 15449632      0t0  TCP localhost:32996->localhc
10
```

这次我们可以看到，redis-server 的 8 号文件描述符，对应 TCP 连接 localhost:6379->localhost:32996。其中，localhost:6379 是 redis-server 自己的监听端口，自然 localhost:32996 就是 redis 的客户端。再观察最后一行，localhost:32996 对应的，正是我们的 Python 应用程序（进程号为 9181）。

历经各种波折，我们总算找出了 Redis 响应延迟的潜在原因。总结一下，我们找到两个问题。

第一个问题，Redis 配置的 appendfsync 是 always，这就导致 Redis 每次的写操作，都会触发 fdatasync 系统调用。今天的案例，没必要用这么高频的同步写，使用默认的 1s 时间间隔，就足够了。

第二个问题，Python 应用在查询接口中会调用 Redis 的 SADD 命令，这很可能是不合理使用缓存导致的。

对于第一个配置问题，我们可以执行下面的命令，把 appendfsync 改成 everysec：

[复制代码](#)


```
1 $ docker exec -it redis redis-cli config set appendfsync everysec
2 OK
```

改完后，切换到终端二中查看，你会发现，现在的请求时间，已经缩短到了 0.9s：

[复制代码](#)

```
1 {..., "elapsed_seconds":0.9368953704833984,"type":"good"}
```

而第二个问题，就要查看应用的源码了。点击 [Github](#)，你就可以查看案例应用的源代码：


 复制代码

```
1 def get_cache(type_name):
2     '''handler for /get_cache'''
3     for key in redis_client.scan_iter("uuid:*"):
4         value = redis_client.get(key)
5         if value == type_name:
6             redis_client.sadd(type_name, key[5:])
7     data = list(redis_client.smembers(type_name))
8     redis_client.delete(type_name)
9     return jsonify({"type": type_name, 'count': len(data), 'data': data})
```

果然，Python 应用把 Redis 当成临时空间，用来存储查询过程中找到的数据。不过我们知道，这些数据放内存中就可以了，完全没必要再通过网络调用存储到 Redis 中。

基于这个思路，我把修改后的代码也推送到了相同的源码文件中，你可以通过 [http://192.168.0.10:10000/get\\_cache\\_data](http://192.168.0.10:10000/get_cache_data) 这个接口来访问它。


我们切换到终端二，按 Ctrl+C 停止之前的 curl 命令；然后执行下面的 curl 命令，调用 [http://192.168.0.10:10000/get\\_cache\\_data](http://192.168.0.10:10000/get_cache_data) 新接口：

 复制代码

```
1 $ while true; do curl http://192.168.0.10:10000/get_cache_data; done
2 {...,"elapsed_seconds":0.16034674644470215,"type":"good"}
```

你可以发现，解决第二个问题后，新接口的性能又有了进一步的提升，从刚才的 0.9s，再次缩短成了不到 0.2s。

当然，案例最后，不要忘记清理案例应用。你可以切换到终端一中，执行下面的命令进行清理：

 复制代码

```
1 $ docker rm -f app redis
```

## 小结

今天我带你一起分析了一个 Redis 缓存的案例。

我们先用 top、iostat，分析了系统的 CPU、内存和磁盘使用情况，不过却发现，系统资源并没有出现瓶颈。这个时候想要进一步分析的话，该从哪个方向着手呢？

通过今天的案例你会发现，为了进一步分析，就需要你对系统和应用程序的工作原理有一定的了解。

比如，今天的案例中，虽然磁盘 I/O 并没有出现瓶颈，但从 Redis 的原理来说，查询缓存时不应该出现大量的磁盘 I/O 写操作。

顺着这个思路，我们继续借助 pidstat、strace、lsof、nsenter 等一系列的工具，找出了两个潜在问题，一个是 Redis 的不合理配置，另一个是 Python 应用对 Redis 的滥用。找到瓶颈后，相应的优化工作自然就比较轻松了。

## 思考

最后给你留一个思考题。从上一节 MySQL 到今天 Redis 的案例分析，你有没有发现 I/O 性能问题的分析规律呢？如果你有任何想法或心得，都可以记录下来。

当然，这两个案例这并不能涵盖所有的 I/O 性能问题。你在实际工作中，还碰到过哪些 I/O 性能问题吗？你又是怎么分析的呢？

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。

# 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，未经许可不得转载

上一篇 28 | 案例篇：一个SQL查询要15秒，这是怎么回事？

下一篇 30 | 套路篇：如何迅速分析出系统I/O的瓶颈在哪里？

## 精选留言 (12)

写留言



ninuxer

2019-01-25

6

打卡day30

io问题一般都是先top发现iowait比较高，然后iostat看是哪个进程比较高，然后再通过strace，lsof找出进程在读写的具体文件，然后对应的分析



李博

2019-01-25

4

老师，有个问题咨询下，为什么top显示 iowait比较高，但是使用iostat却发现io的使用率并不高那？

作者回复: iowait不代表磁盘I/O存在瓶颈，只是代表CPU上I/O操作的时间占用的百分比。假如这时候没有其他进程在运行，那么很小的I/O就会导致iowait升高



利俊杰

2019-01-26

👍 2

`nsenter --target $PID -- lsof -i`

执行失败，提示：`loadlocale.c:130: _nl_intern_locale_data: Assertion `cnt < (sizeof (_nl_value_type_LC_COLLATE) / sizeof (_nl_value_type_LC_COLLATE[0]))' failed`

可以参考下<https://stackoverflow.com/questions/37121895/yocto-build-loadlocale-c-130...>

展开 ▼

作者回复: 谢谢分享



往事随风, 顺...

2019-01-25

👍 2

`git clone https://github.com/feiskyer/linux-perf-examples/tree/master/redis-slow`

Initialized empty Git repository in `/root/redis-slow/.git/`

error: The requested URL returned error: 403 Forbidden while accessing

`https://github.com/feiskyer/linux-perf-examples/tree/master/redis-slow/info/refs`

...

展开 ▼

作者回复: clone要指定代码仓库的路径，而不是子目录:

`git clone https://github.com/feiskyer/linux-perf-examples`



Cranliu

2019-01-25

👍 2

top、iostat、pidstat、strace，以及对应用程序的了解，MySQL、Redis本质上也是一款应用程序。



开始懂了

2019-01-25

👍 1

`curl http://10.39.25.7:10000/init/get_cache`

`<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">`

`<title>500 Internal Server Error</title>`



<h1>Internal Server Error</h1>

<p>The server encountered an internal error and was unable to complete your reque...

展开 ▾

作者回复: /init/ 后面需要一个数字



Christmas

2019-01-25

👍 1

进程iowait高，磁盘iowait不高，说明是单个进程使用了一些blocking的磁盘打开方式，比如每次都fsync



\_\_\_的我

2019-02-11

👍

前段时间刚找到一个由于内存数据被交换到swap文件中导致内存数据遍历效率变低的问题 问题定位过程是使用pidstat命令发现进程cpu使用率变低 mpstat命令观察到系统iowait升高 由此怀疑跟io有什么关系 perf命令观察发现内存数据遍历过程中swap相关调用时间占比有点异常 然后使用pidstat命令+r参数 也观察到进程在那段时间主缺页中断升高 由此确定问题

...

展开 ▾



陈云卿

2019-02-06

👍

打卡day30

IO性能问题首先可以通过top 查看机器的整体负载情况，一般会出现CPU 的iowait 较高的现象

然后使用 pidstat -d 1 找到读写磁盘较高的进程

然后通过 strace -f -TT 进行跟踪，查看系统读写调用的频率和时间...

展开 ▾

作者回复: 🍊



Geek\_a9ec17

2019-01-29

👍

Device: rrqm/s wrqm/s r/s w/s kB/s kB/s avgrq-sz avgqu-sz await r\_await w\_await svctm %util

vda 0.00 392.00 0.00 1103.00 0.00 5984.00 10.85 1.00 0.58 0.00 0.58 0.91 100.00

我这边 %util到了100%，说明磁盘有瓶颈了吗？（请求参数一致）

作者回复: 一般来说是的。不过如果磁盘支持并发写的话，实际上这时候磁盘还是可以继续接受其他的写请求，所以这不是绝对的。最好是做个基准测试，这样以后再观察的时候就有了对比的基准。



我来也

2019-01-26



[D29打卡]

感觉每次分析的套路都差不多.

- 1.用top查看指标,发现 [系统] 有i/o瓶颈 或者 cpu瓶颈.
- 2.使用iostat辅助看下磁盘i/o读写速度和大小等指标.
- 3.用pidstat判断是哪个 [进程] 导致的. 既可以看进程各线程的cpu中断数,也可以看磁盘io数...

展开 ▼

作者回复: 如果能一个套路查遍所有问题就好了😂 我相信很多人都期待这样



夹心面包

2019-01-25



想请问下老师,通过top观察下的iowait到达百分之多少才算磁盘瓶颈,有没有业界的统一标准,磁盘的util值肯定是100%,还是得考虑IOPS,只通过iowait判断行不行

作者回复: 只用iowait不能说明磁盘瓶颈，需要用 iostat