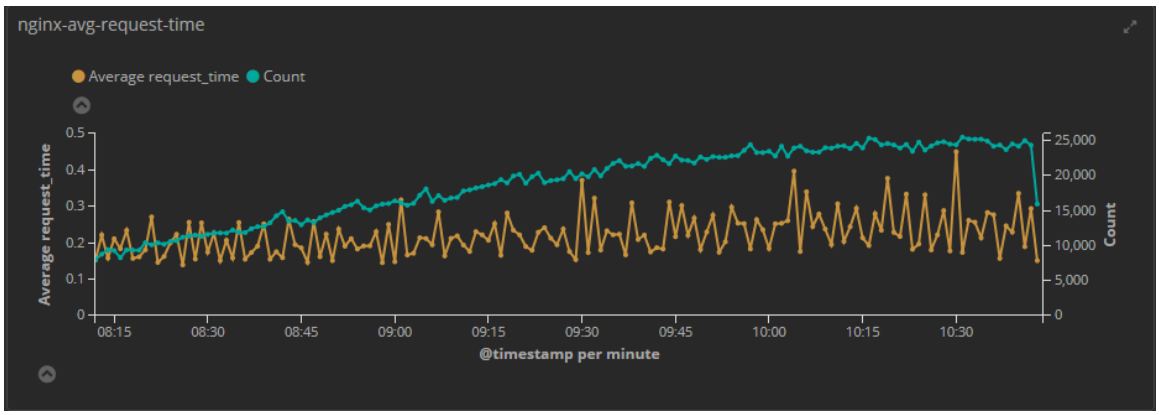
基于 Linux 系统工具对 JavaSpring 架构业务的性能优化

Email: geekidea@gmail.com

2018/12

0. 问题描述

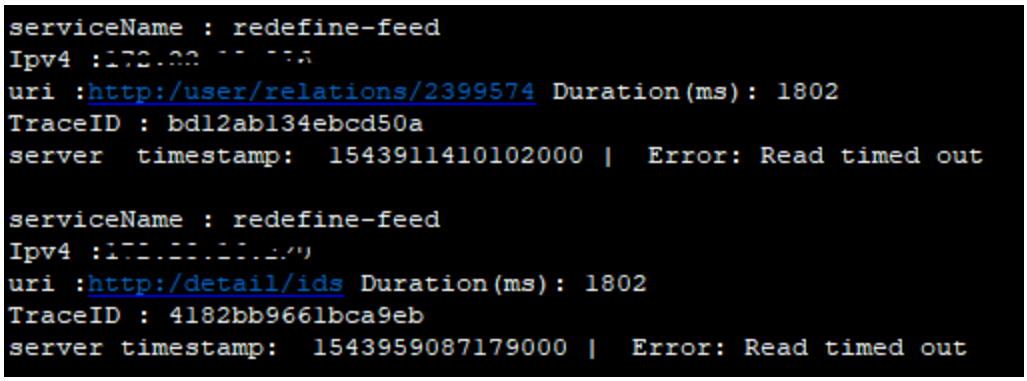
最近线上基于 spring cloud 的 java 业务服务间调用频繁超时报警，通过对外的 nginx 的访问日志 kibana 大盘可以看到总体的平均响应时间以及大多数具体业务的响应时间均出现相关抖动，nginx 总体平均响应时间如下图：



相关主要业务 tp 响应时间分布情况：

filters	50th percentile of request_time	90th percentile of request_time	99th percentile of request_time	99.9th percentile of request_time
request_uri: "/feed"	0.135	0.512	2.353	4.524
request_uri: "/user"	0.013	0.287	1.645	4.038
request_uri: "/im"	0.009	0.023	0.465	2.5

同时应用业务间调用的主要超时报警信息如下：



通过报警信息可以了解到，feed 服务在调用两个内部 URI 业务时，出现了 1.8 秒超时，当用户并发访问加剧时，报警频率也会增加，由于这种内部业务间的个别请求的调用频繁超时，当客户端请求在服务端大量等待堆积时，平均（及 TP）响应时间会随之增大，平均响应时间出现频繁抖动。

用户 app 端会长时间等待服务器响应或频繁重试，影响最终用户体验。

1. 问题分析

对于业务性能瓶颈，可以采取先全局再具体的解决方法：宏观上可以观察整体大盘性能指标，各业务所在服务器（或 docker 容器）的系统资源（CPU/内存/IO/网络）使用情况，梳理服务之间的调用链，然后深入到相关服务内部的调用链，最后聚焦在出现超时的具体业务进程上。

在具体解决方案上，一般优先采取低成本的优化方案，如优化应用程序、基础架构服务或系统配置参数，如：TCP listen backlog，net.core.somaxconn，Tomcat java 连接队列，java 线程池配置等；然后再从程序架构实现寻找优化方案，如：pipeline 访问 redis，mysql 分库分表，将业务数据按业务垂直拆分等。如果基础架构服务（mysql/redis 等）不存在瓶颈，在短期内也可以采用增加应用进程（ECS 或 docker）实例的方法水平扩展。

超时问题出现在 feed 业务调用两个内部业务上，首先可以梳理下 feed 服务间的上下游依赖（包括数据库，redis 等基础组件）的调用链，然后仔细分析出现具体超时的业务。通过 kibana 大盘可以看到其他相关服务也会出现平均响应时间抖动现象，初步可以判断为这些业务的某些共同依赖的基础业务服务（或基础架构服务）出现了瓶颈。

超时错误剖析（SIGPIPE/EPIPE/Broken pipe Exception）

当业务 A 基于 TCP 连接方式（例如：http restful/grpc）调用业务 B 时，如果业务 A 设置了读写（或连接）超时时间，在该文档所描述的问题中 feed 业务设置的读写超时是 1.8 秒，当读写超时时间达到后业务 A 会主动关闭 TCP 连接，此时如果业务 B 通过该连接发送应答时，由于该 TCP 连接已被 A 提前关闭，此时 B 会触发 **SIGPIPE** 的系统信号，对于健壮的服务器程序，为避免该信号产生时导致服务器进程退出（core dump），在应用代码层面会忽略该 SIGPIPE 信号，取而代之会向应用代码返回 **EPIPE** 系统错误码（errno），对于

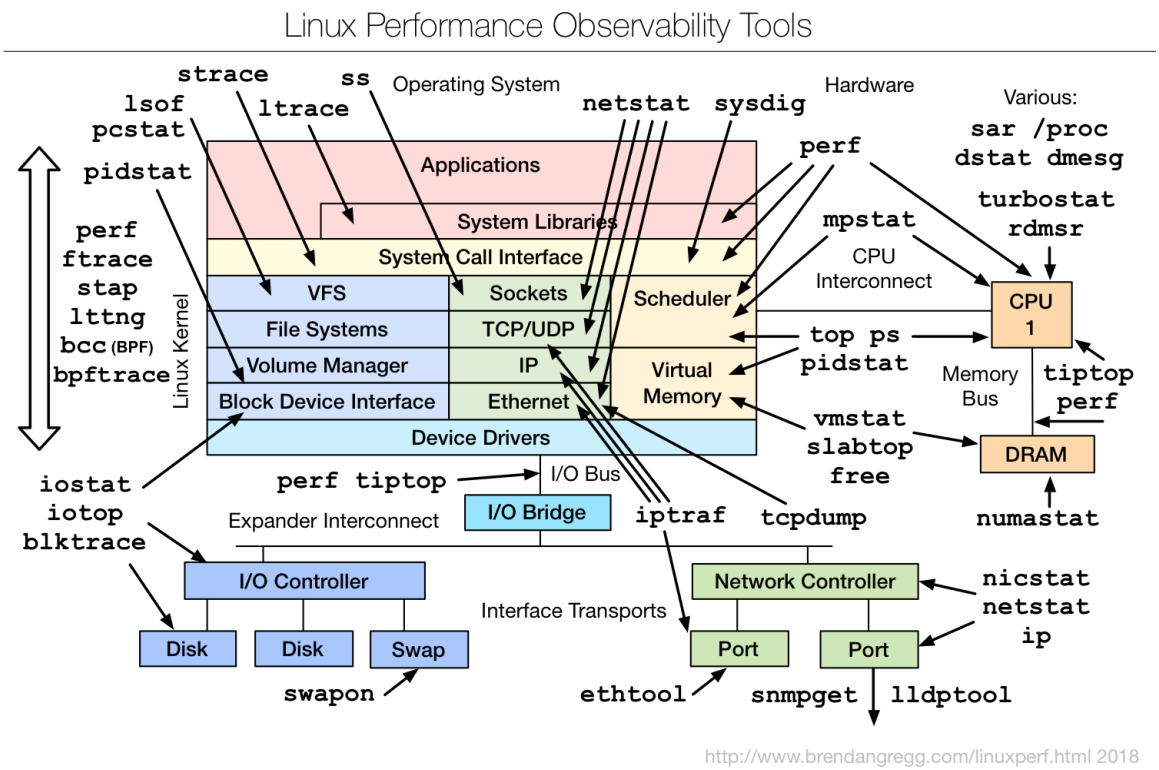
java 应用会返回“**java.io.IOException: Broken pipe**”异常，当 java 业务日志中频繁出现该异常信息时，需要引起注意。

当出现 EPIPE 系统错误或相关异常时，我们的应用程序该做什么？

由于连接已被对端（上游业务）关闭，此时下游业务中任何基于该连接的后续网络 I/O 均会返回错误，此时下游应用程序也关闭该 socket 连接即可。具体处理细节也可以参考 redis/memcached/haproxy/nginx 等开源代码。

Linux 系统性能诊断工具

Linux 中存在很多系统性能诊断工具，用于通过不同的角度来观察操作系统和应用系统的瓶颈，常见 Linux 性能工具集图谱如下（图片取自 BrendanGregg 个人主页）：



对于应用程序性能瓶颈分析，我最常用的工具有 strace 和 lsof，常见 Linux（centos/RHEL）操作系统一般默认已经安装，或者通过 sudoyum install -y stracelsof 安装。

lsof 可以观察系统当前所有文件描述符（含 socket 文件描述符）打开情况：

sudolsof

观察某一进程的当前文件及 socket 打开情况，-p 指定进程 ID，例如：

sudolsof -nNPp12345

查看某一端口被某些进程占用的情况：

```
sudo lsof -nNPi:3306
sudo lsof -nNPi@10.0.100.3:6379
```

查看某一文件（或 unix domain 文件）被某些进程占用的情况：

```
sudo lsof -nNP /var/log/nginx/error.log
sudo lsof -nNP /var/run/docker.sock
```

特别地 lsof 工具对于判断某一进程是否存在文件（socket 连接）句柄泄漏 bug，非常有帮助：当某一进程的 lsof 输出中存在大量处于 **CLOSE_WAIT** 状态的 socket 句柄，或者存在大量的 can't identify protocol 文件描述符时，一般可以判断该程序处于繁忙状态，以至于无暇关闭客户端（已经被客户端提前关闭的）TCP 连接，或者程序本身存在 bug。

strace 可以观察某进程所有的系统调用情况，如网络及文件 I/O，用户态锁（futex）等，常用命令：

```
export jspid=12345
strace -f -s 1024 -ttT -o ${jspid}.st -p ${jspid}
```

通过-p 参数指定进程 ID，-o 将系统调用输出到指定文本文件，用于后续分析。

特别注意，使用 strace 对指定进程诊断分析时，该进程性能一般会损失 15%左右，因此在生产环境下，一般对某进程进行诊断分析时仅抓取一小段时间（问题复现的时间周期内）即可；在测试及开发环境等对性能指标不特别关心的场景下，可以用 strace 抓取相对长一些时间，便于辅助我们详细观察程序的行为，上下游依赖数据，剖析业务代码的实现架构及细节，如：是否基于 master/worker 多线程、异步线程调用，是否正确处理了 EPIPE 系统错误等。

feed 超时问题分析

基于通过 lsof 和 strace 工具对 feed 进程调用链做一个宏观梳理，在生产环境（或测试环境）找到 feed 进程，假定进程 ID 为 12345，然后执行如下命令：

```
export jspid=12345
sudo lsof -nNPp${jspid} > ${jspid}.lsof; sudo strace -f -s 1024 -ttT -p ${jspid} -o ${jspid}.st
```

根据实际情况，抓取 30 秒左右 ctrl+c 退出 strace 抓取，其中文件描述符状态快照（lsof 输出）保存在了 12345.lsof 文件中，该时间段进程 12345 产生系统调用保存在了 12345.st 中。

查看该进程监听的端口：

cat 12345.lsof | grep LISTEN

查看该进程当时保持的 TCP 连接：

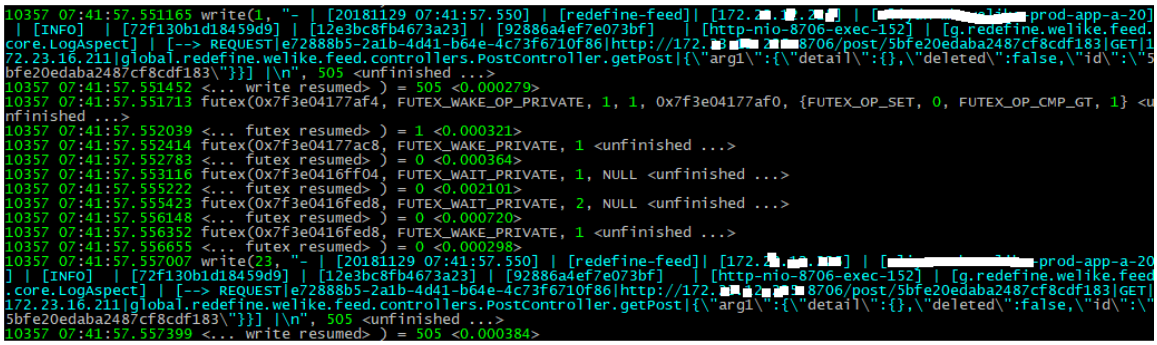
cat 12345.lsof | grep ESTAB

可以看到 feed 进程监听的端口为 8706 端口，feed 跟其他服务保持 TCP 主要连接有：

```
eureka (172.*.*.*:8090)
redis (172.*.*.*:6379)
rabbitmq (172.*.*.*:5672)
kafka(172.*.*.*:9092)
下游业务A (172.*.*.*:8715)
下游业务B (172.*.*.*:8720)
```

优先解决程序自身的耗时

通过 strace 日志（12345.st）观察到，程序会输出两份同样的业务日志，其中一份输出到标准输出（屏幕），另一份会打印到业务日志文件，如下图：



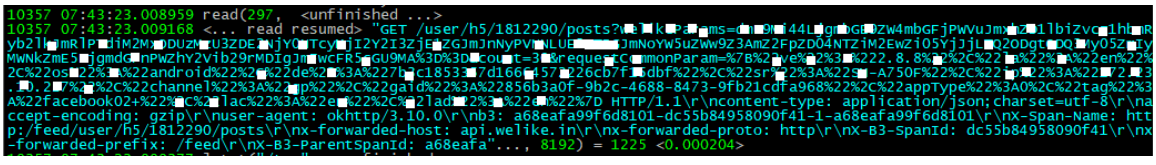
如上截图中可见同一份日志信息写入到了文件句柄 1，句柄 23 中，其中文件句柄 23 在 12345.lsof 文件中可以查看到与业务日志文件关联（/data/**/**/all_**-feed.log）；通过 12345.lsof 可见文件句柄 1 已被重定向到/dev/null 中：

由于标准输出及标准出错都被定向到了/dev/null，因此可以在应用程序中禁掉向标准输出（stdout）、及标准出错（stderr）打印日志信息。

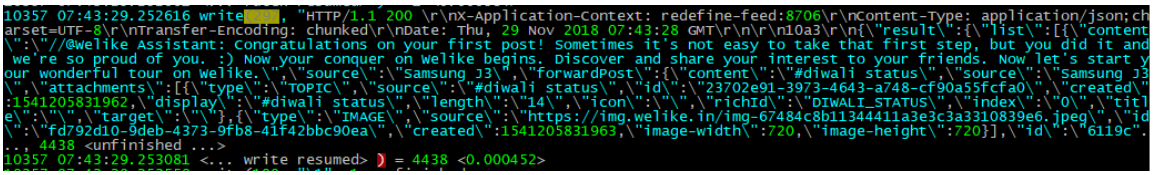
strace 日志文件第一列为相关的线程 ID，对于单进程单线程程序，该线程 ID 等同于进程 ID；对于多线程应用来说，该 ID 关联了具体的工作线程 ID，为了便于分析某一工作线程的运行逻辑，可以对该 strace 日志文件按照第 1 列进行分组，例如只分析线程 ID 为 10357 的执行逻辑，可以通过如下命令查看：

grep ‘^10357 ’ 12345.st > t10357.st

在该文件中通过请求接收和应答发送时间戳，可以计算该请求的具体耗时，接受请求如下图：

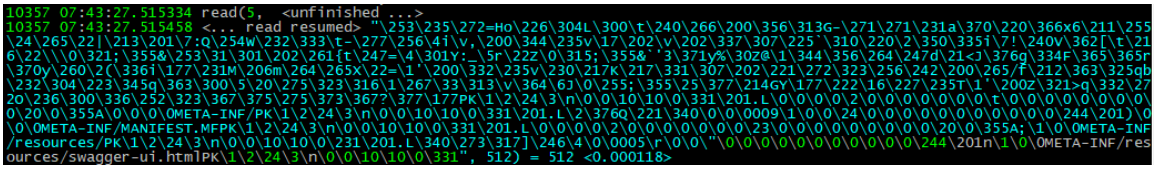


应答发送如下图：

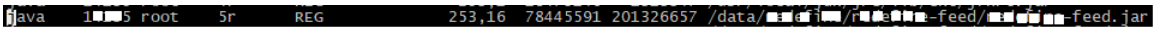


其中文件句柄 297 标识了一个具体的客户端 socket TCP 连接，如果是通过 strace 抓取后的新进连接，可以通过过滤”accept”系统调用观察该句柄 297 标识的来源客户端 IP 及端口。

通过上面截图第 2 列的请求接收时间戳为 07:43:23.009168，应答发送的时间戳为 07:43:29.252616，可以计算出该请求耗时约 6.25 秒，进一步地观察到处理该请求时出现了大量的 lseek/read 系统调用：



在 12345.lsof 文件中记录了文件句柄 5 关联的具体文件



lseek/read 一般用于通过指定文件偏移量顺序读取文件内容，通过以上截图可以判断进程在加载业务 Jar 包文件某个 java 类时，耗时约 4.5 秒。由于该 jar 包除新版本上线之外并不会变化，因此可以初步判断 spring cloud 中的 tomcat 容器开启了 reloadable 属性，用于检测 jar 包中类库动态加载。对于该问题基本优化思路可以：

1. 生产环境中禁止 spring cloudtomcat 容器的 reloadable 属性。

2. 如果该类库无关紧要，将该类库从 jar 包中去掉。

3. 调整 jar 包加载读取缓冲区，由 strace 看到每次顺序读取 512 字节，可以调大读取缓冲区，或者使用 mmap 等方式一次将 Jar 包加载至内存。

4. 将 jar 包文件保存在内存文件系统，如：tmpfs，减少实际文件 I/O 开销。

实际优化中，由于 tomcat 使用了 spring cloud 内嵌默认的版本，在业务代码中使用 context.setReloadable(false);并未达到预期效果，方法 3 显然需要 hack 修改 tomcat 源码或其他相关配置，复杂度较高。方法 4 需要将 jar 包文件发布在 tmpfs 路径下，对现在发布流程影响较大。

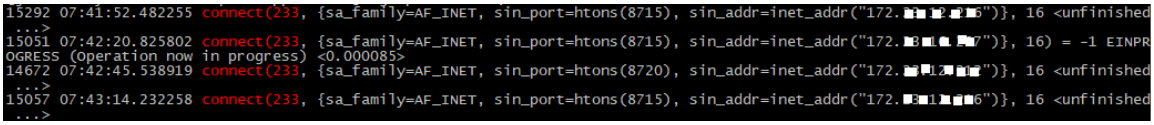
由于频繁加载的该类库属于 swagger restful 函数说明帮助页面，因此最终采用了方法 2，将该类库从生产环境 jar 包中剔除，对正常业务本身无影响。

确定超时的下游业务 IP 及端口

由应用报警截图可见，当 feed 应用发送 **relations** 及 **detail/ids** 请求时出现了超时报警，可以在 strace 日志文件中过滤'/user/relations'查看发送该请求的情况（句柄、时间戳、频度、请求耗时等）：grep '/user/relations' 12345.st | grep sendto，结果如下图

可以通过截图 HTTP Header 中的 Host 找到对应下游机器 IP 及端口，如果 Header 中不存在 Host 信息，另外结合上下文（lsof 日志文件，及 connect 系统调用）找到该下游业务对应的 ip 及端口。

以某一条发送给（系统调用为 sendto***）下游的 relations 请求为例，假设句柄号为 233，在 strace 日志文件向上查找该句柄，vim 中向上查找：'(233,'直到找到 connect(233, 位置结束，在 connect(***中可以看到对应的下游服务 IP 端口，例如：



找到与发送请求（sendto）对应时间戳向前匹配最接近的那个连接，然后可以确定对该连接对应下游服务 IP 及端口。

如果没有找到任何该句柄的连接动作，可以推断出：

1. 该连接在 strace 抓取前已经建立好，可以尝试到 lsof 日志文件中查看该句柄，或者

2. 再使用上述方法尝试查看其它 sendto 对应的句柄连接。

下游业务 relations 诊断

当确定了下游业务 relations 所在 IP 后，我们可以重复使用 lsof/strace 工具在下游业务所在机器上对定位瓶颈，首先找到 8715 端口对应的业务进程，执行命令：sudolsof -nNPi:8715 | grep LISTEN，输出如下：

```
java    14985 root   42u  IPv4 426991408      0t0  TCP *:8715 (LISTEN)
```

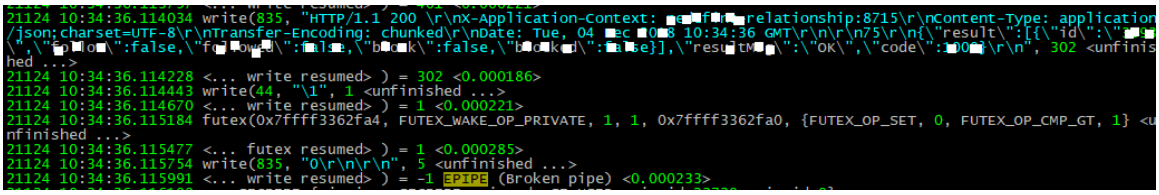
找到对应进程 PID（如：14985）后，再利用 lsof/strace 抓取一段时间该进程的系统调用情况，执行命令：

```
export jspid=14985
sudo lsof -nNPp${jspid} > ${jspid}.lsof; sudo strace -f -s 1024 -ttT -p ${jspid} -o ${jspid}.st
```

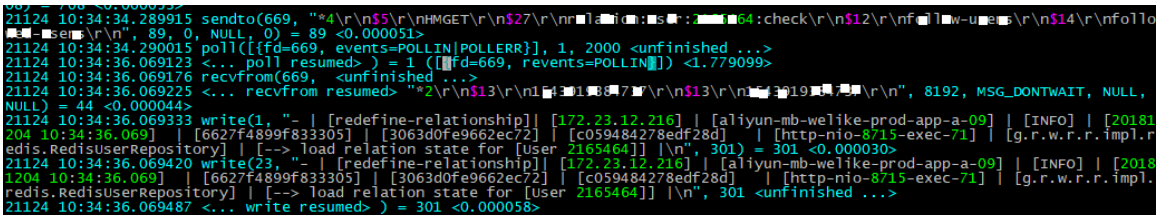
一段时间后，当 feed 业务出现调用超时报警时，退出 strace 抓取，前面提到由于 feed 业务超时关闭连接后，下游服务处理完成发送应答时，会出现 EPIPE 异常，可以通过下面脚本（ana_epipe.sh）命令过滤出 EPIPE 错误关联的具体线程处理逻辑：

```
#!/bin/sh

pid=""
if [ $# == 0 ]; then
  pid=${jspid}
elif [ $# -ge 1 ]; then
  pid=$1
fi

if [ "x$pid" == "x" ]; then
  echo "Error: No process to be analysed, pls specified ENV 'jspid' or pass pid param"
  exit 1;
fi

echo "Analysing EPIPE Error process: $pid ..."
mkdir -p ep
grep EPIPE ${pid}.st > ep/epipe.st
cat ep/epipe.st | awk '{print $1}' | sort| uniq > ep/tid

cat ep/tid | while read line
do
grep "^$line\ " ${pid}.st > ep/t${line}.st
done

echo "Nav the dir 'ep' for more details"
```

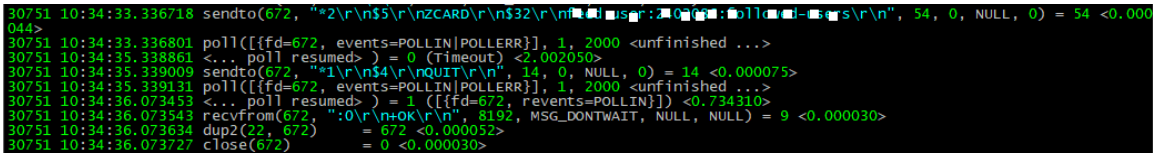
上述脚本会过滤 strace 抓取日志中的 EPIPE 错误，将出现 EPIPE 的工作线程执行的系统调用以线程名字存储在 ep 子目录下的文件中，打开其中一个文件（如：t21124.st），查找 EPIPE 错误，如下图：



然后向上反查找到耗时最大的响应，



可见，该业务在执行某个 redis 查询（HMGET）时候出现了 1.7 秒左右耗时，通过该方法查看其它出现 EPIPE 错误的其他工作线程，基本都是在执行 redis 查询时候出现了比较长的耗时，又如：

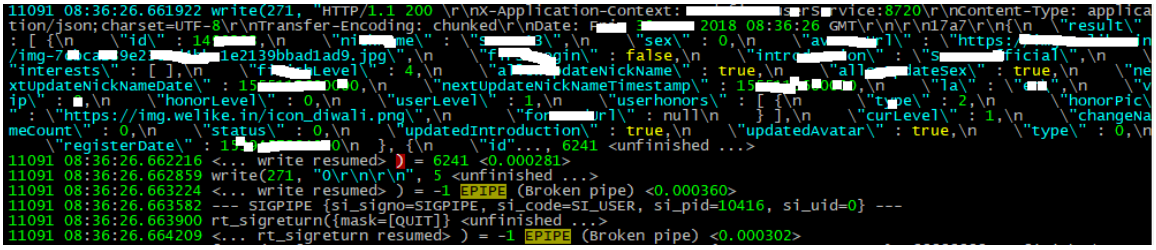


上面截图看到，该线程（ID 30751）发送 ZCARD 命令后，等待 redis 服务器应答时出现了约 2 秒的超时，然后该线程通过发送 QUIT 命令及后续的 close 系统调用关闭了该 redis 连接。

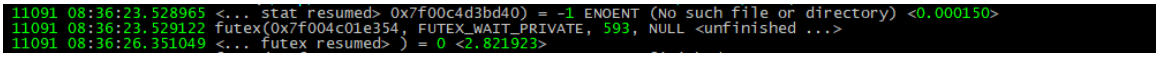
因此对于该下游业务 relations 来说，该问题的瓶颈出现在 redis 服务器上。

下游业务 details/ids 诊断

对于 detail/ids 的下游业务，采用同样的上述方法，在所在 IP 机器上执行 lsof/strace 抓取一段时间的系统调用执行情况（分别保存在 10416.lsof 文件、10416.st 文件汇总），然后执行 ana_epipe.sh 脚本继续解析出 EPIPE 错误的关联线程，打开其中一个文件（如：t11091.st），如下图：



向上反查耗时最大的系统调用：



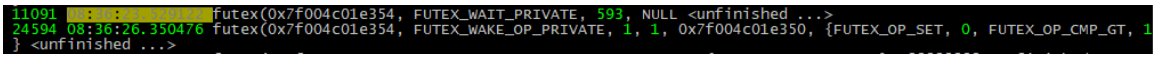
该耗时阻塞在了 futex 系统调用，第一个参数为锁 ID，**FUTEX_WAIT_PRIVATE** 参数为等待获取锁，futex 一般用于实现上层用户态的锁，例如 c/c++常用的 libpthread 中的 pthread_mutex，java 的锁机制等，通过该锁机制可以实现不同线程的基本通信。

对于 spring cloud tomcat 的应用，通过 strace 日志可以观察到，端口监听线程和工作线程（组）是不同的线程，当端口监听线程接受到（使用 accept 系统调用）新连接到来时，会将该连接投递到指定工作线程（所在的队列中），然后向该工作线程发送管道通知。每

个工作线程都有各自不同的锁（因为 futex 锁 ID 是不同的），监听线程与工作线程的协作底层均是通过该 futex 锁来协调。

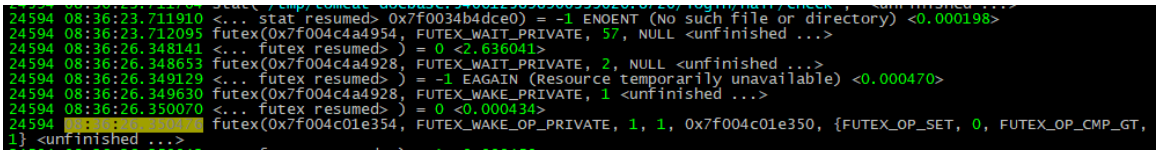
截图中主要耗时的 futex 锁 ID 为 0x7f004c01e354，可以初步判断，该业务请求在另一个线程中做了进一步的处理，当请求在另一个线程处理完毕后，通过系统调用 futex，参数为 **FUTEX_WAKE_OP_PRIVATE** 解锁，这样原线程可以执行后续处理，如：发送应答给上游服务。

了解到了 futex 的基本机制，我们可以按照锁 ID 维度来查找另一个请求处理线程，执行命令：grep 0x7f004c01e354 ../10416.st > 0x7f004c01e354.st，得到 futex 锁 0x7f004c01e354 等待阶段的另一个处理线程 ID，如下图：



可以看到该处理线程 ID 为 24594 在 08:36:26.350476 时刻处理完毕后，释放了这个锁，这样之前第一个工作线程才能执行后续操作（发送应答）。

以该线程 ID 24594 过滤查找相关的操作：grep '^24594 ' ../10416.st > t24594.st，最终找到该处理线程的耗时操作：



可以看到该处理线程同样陷入了 futex 锁等待，该 futex 锁 ID 为 0x7f004c4a4954，利用上述同样方法，找到这个时间段内释放锁 0x7f004c4a4954 的另一个线程，如果该线程同样陷入了不同的 futex 锁等待，继续循环上述方法，最终找到实际耗时的瓶颈。

换个角度，在 strace 日志中当某一个较短的时间段内（同一秒内），如果出现大量的 futexFUTEX_WAIT_PRIVATE 锁等待情况，此时或许该业务进程内大量的工作线程都集中陷入到了一个下游服务或系统调用里面，比如：等待 mysql、redis 服务器返回数据，因此会出现大量的请求堆积等待被处理，需要引起注意。

最终该业务瓶颈仍旧在 redis 服务器上。

优化 redis

瓶颈确定后，redis 一般可以采取几个优化策略：

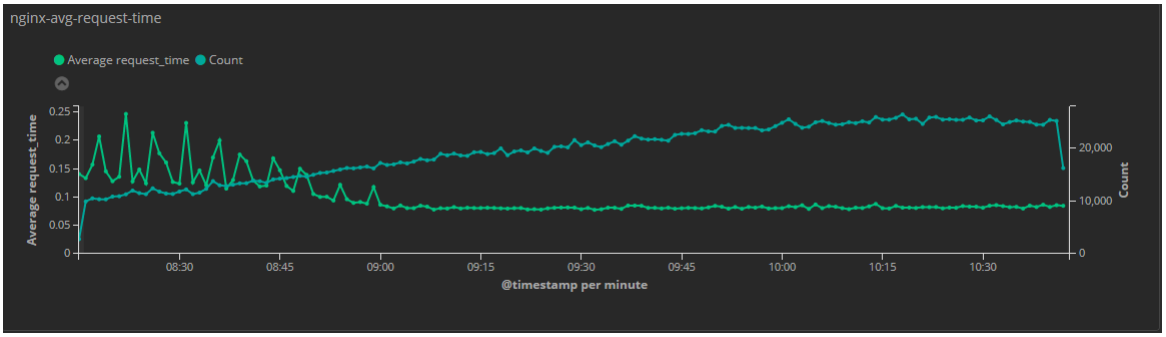
1. 按业务垂直拆分，将延迟敏感的业务，业务相对独立的数据单独 redis 实例存储。
2. 如果压力在主库频繁读写时，根据业务需要关闭主库的持久化策略，如果主库宕机时可能会丢失部分数据，或者利用 redis 中间件将读写分离。
3. 修改配置参数，如 listen backlog 增大 TCP 连接并发吞吐，调小定时刷新过期 key 频率（hz）。
4. 优化 redis 业务数据类型设计。

最终采用的优化策略为：将 relations 业务数据拆分到新的 redis 实例，优化后 relations 响应超时大幅降低。

对于 details/ids 服务，调整定时刷新 key 频率（hz 由 80 调整至 20），优化后 feed 调用该服务的超时大幅降低，同时整体响应时间也大幅缩小，相关报警也基本消失。

优化结果

最终优化后，出口 nginx 的平均响应时间如下：



相关 tp 响应时间分布情况如下：

filters	50th percentile of request_time	90th percentile of request_time	99th percentile of request_time	99.9th percentile of request_time
request_uri: "/feed"	0.078	0.245	0.762	2.235
request_uri: "/user"	0.009	0.105	0.571	1.976
request_uri: "/im"	0.008	0.018	0.131	0.832

优化前后对比，平均响应时间比之前下降了约 **1/2**，各 tp 分布的响应时间均下降了约 **1/2**，各个具体子业务的平均响应时间抖动区间相比之前更为平滑，app 端的用户体验更好。

2. 后续思考

以上描述了通过 Linux 相关系统工具 lsof 及 strace，对 Java spring cloud 业务的优化分析过程，由于是从系统角度分析业务的瓶颈，该方法并不局限于特定的编程语言，对于 c/c++，go，nodejs，php，python 等业务应用的瓶颈分析优化，仍可以使用该方法，由于 strace 会导致待监控的业务进程性能下降产生略微损失，随着 Linux 内核的完善，更轻量级诊断工具如 **sysdig** 等，对业务本身产生的额外开销更小，会逐渐流行起来。

对于 spring cloud 业务应用，随着我们的全链路监控逐步完善，可以通过继续完善定制 zipkin，SDK 调用埋点方式，将各个业务的每个请求的调用链及耗时开销完整展示出来，便于开发和业务运维同学快速定位问题。

如你所见，目前我们的业务系统依旧存在以下问题：

1. nginx 响应时间仍旧会出现某一时间段的小范围抖动。

2. tp 响应时间仍需优化，个别业务的 tp99.9 的响应时间超过了 1 秒

3. 在 lsof 截图可见线上业务依旧使用 root 启动。

因此还需要对具体业务的完整调用链路及运行环境（权限），进一步持续优化。

3. 参考资料

《TCP/IP 详解》卷一

《UNIX 环境高级编程》

lsofstrace Linux man 手册

Brendan Gregg 主页：http://www.brendangregg.com/