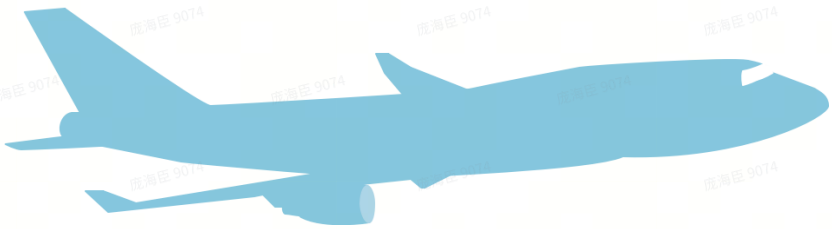
《JAVA线上故障排查全套路》

“文章来自网络，运维搜集整理。希望对大家有所帮助。”

> 一键排查故障脚本：https://github.com/oldratlee/useful-scripts
>
> 推荐Arthas工具（发布平台已集成该工具）：
>
> https://arthas.aliyun.com/doc/arthas-tutorials.html?language=cn&id=arthas-basics【实战教程】
>
> https://arthas.aliyun.com/doc/quick-start.html#dashboard 【快速入门】
>
> ## Arthas实践系列
>
> - Alibaba Arthas实践--获取到Spring Context，然后为所欲为
> - Arthas实践--快速排查Spring Boot应用404/401问题
> - 当Dubbo遇上Arthas：排查问题的实践
> - Arthas实践--抽丝剥茧排查线上应用日志打满问题
> - 深入Spring Boot：利用Arthas排查NoSuchMethodError



线上故障主要会包括cpu、磁盘、内存以及网络问题，而大多数故障可能会包含不止一个层面的问题，所以进行排查时候尽量四个方面依次排查一遍。同时例如jstack、jmap等工具也是不囿于一个方面的问题的，基本上出问题就是df、free、top 三连，然后依次jstack、jmap伺候，具体问题具体分析即可。

CPU

一般来讲我们首先会排查cpu方面的问题。cpu异常往往还是比较好定位的。原因包括业务逻辑问题(死循环)、频繁gc以及上下文切换过多。而最常见的往往是业务逻辑(或者框架逻辑)导致的，可以使用jstack来分析对应的堆栈情况。

使用jstack分析cpu问题

我们先用ps命令找到对应进程的pid(如果你有好几个目标进程，可以先用top看一下哪个占用比较高)。**接着用`top -H -p pid`来找到cpu使用率比较高的**一些线程

```
Threads:  47 total,   0 running,  47 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.5 us,  0.3 sy,  0.0 ni, 99.2 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  4047500 total,   256512 free,  2555208 used,  1235780 buff/cache
KiB Swap:  1044476 total,   700192 free,   344284 used.  1214228 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
   66 root      20   0 4733160 2.002g   9252 S  0.3 51.9   3:46.27 java
    1 root      20   0 4733160 2.002g   9252 S  0.0 51.9   0:00.35 java
   29 root      20   0 4733160 2.002g   9252 S  0.0 51.9   0:37.35 java
   30 root      20   0 4733160 2.002g   9252 S  0.0 51.9  49:38.12 java
   31 root      20   0 4733160 2.002g   9252 S  0.0 51.9  49:32.75 java
   32 root      20   0 4733160 2.002g   9252 S  0.0 51.9   3:11.46 java
   33 root      20   0 4733160 2.002g   9252 S  0.0 51.9  17:55.99 java
```

然后将占用最高的pid转换为16进制`printf '%x\n' pid`得到nid

```
sh-4.2# printf  '%x\n' 66
42
```

接着直接在jstack中找到相应的堆栈信息`jstack pid |grep 'nid' -C5 -color`

```
sh-4.2# jstack 1 | grep '0x42' -C5
        at java.lang.Thread.run(Thread.java:748)

"DestroyJavaVM" #37 prio=5 os_prio=0 tid=0x00007f66d800b000 nid=0x1d waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"Scanner-0" #36 daemon prio=5 os_prio=0 tid=0x00007f66d92d0000 nid=0x42 in Object.wait() [0x00007f6684bc4000]
   java.lang.Thread.State: TIMED_WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        at java.util.TimerThread.mainLoop(Timer.java:552)
        - locked <0x00000000a40172e8> (a java.util.TaskQueue)
        at java.util.TimerThread.run(Timer.java:505)
```

当然更常见的是我们对整个jstack文件进行分析，通常我们会比较关注WAITING和TIMED_WAITING的部分，BLOCKED就不用说了。我们可以使用命令`cat jstack.log | grep "java.lang.Thread.State" | sort -nr | uniq -c`来对jstack的状态有一个整体的把握，如果WAITING之类的特别多，那么多半是有问题啦。

```
sh-4.2# cat jstack.log | grep "java.lang.Thread.State" | sort -nr | uniq -c
      8    java.lang.Thread.State: WAITING (parking)
      2    java.lang.Thread.State: WAITING (on object monitor)
      1    java.lang.Thread.State: TIMED_WAITING (sleeping)
     16    java.lang.Thread.State: TIMED_WAITING (parking)
      3    java.lang.Thread.State: TIMED_WAITING (on object monitor)
     12    java.lang.Thread.State: RUNNABLE
```

频繁gc

当然我们还是会使用jstack来分析问题，但有时候我们可以先确定下gc是不是太频繁，使用`jstat -gc pid 1000`命令来对gc分代变化情况进行观察，1000表示采样间隔(ms)，S0C/S1C、S0U/S1U、EC/EU、OC/OU、MC/MU分别代表两个Survivor区、Eden区、老年代、元数据区的容量和使用量。YGC/YGT、FGC/FGCT、GCT则代表YoungGc、FullGc的耗时和次数以及总耗时。如果看到gc比较频繁，再针对gc方面做进一步分析，具体可以参考一下gc章节的描述。

```
sh-4.2# jstat -gc 1 1000
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC   CCSU   YGC     YGCT    FGC    FGCT     GCT
52416.0 52416.0 22719.3  0.0   419456.0 332459.3 1572864.0   952345.1  56056.0 54794.9 6392.0 6100.9 304524 4098.010  180    30.108 4128.118
52416.0 52416.0 22719.3  0.0   419456.0 332461.3 1572864.0   952345.1  56056.0 54794.9 6392.0 6100.9 304524 4098.010  180    30.108 4128.118
52416.0 52416.0 22719.3  0.0   419456.0 332463.4 1572864.0   952345.1  56056.0 54794.9 6392.0 6100.9 304524 4098.010  180    30.108 4128.118
52416.0 52416.0 22719.3  0.0   419456.0 332467.5 1572864.0   952345.1  56056.0 54794.9 6392.0 6100.9 304524 4098.010  180    30.108 4128.118
52416.0 52416.0 22719.3  0.0   419456.0 332469.5 1572864.0   952345.1  56056.0 54794.9 6392.0 6100.9 304524 4098.010  180    30.108 4128.118
```

上下文切换

针对频繁上下文问题，我们可以使用`vmstat`命令来进行查看

```
sh-4.2# vmstat 1
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 3  0 349780 151212 198336 1153836    0    0     2   323    1    1 26 10 64  1  0
 0  0 349780 151084 198336 1153836    0    0     0     0  277  471  0  1 100  0  0
 0  0 349780 151148 198344 1153832    0    0     0    28  326  603  0  0 100  0  0
 0  0 349780 151148 198344 1153840    0    0     0     0  310  539  0  0 100  0  0
 0  0 349780 151148 198344 1153840    0    0     0    32  319  527  1  0 99  0  0
```

cs(context switch)一列则代表了上下文切换的次数。

如果我们希望对特定的pid进行监控那么可以使用`pidstat -w pid`命令，cswch和nvcswch表示自愿及非自愿切换。

```
sh-4.2# pidstat -w 1
Linux 3.19.0-25-generic (backend-test07)        10/31/19

21:29:19      UID       PID   cswch/s nvcswch/s  Command
21:29:20        0       542      1.00      0.00  pidstat

21:29:20      UID       PID   cswch/s nvcswch/s  Command
21:29:21        0       542      1.00     12.00  pidstat
^C

Average:      UID       PID   cswch/s nvcswch/s  Command
Average:        0       542      1.00      6.00  pidstat
```

磁盘

磁盘问题和cpu一f样是属于比较基础的。首先是磁盘空间方面，我们直接使用`df -hl`来查看文件系统状态

```
~ df -hl
Filesystem      Size   Used  Avail Capacity iused      ifree %iused  Mounted on
/dev/disk1s5   233Gi   10Gi   74Gi    12%  481770 2447619550    0%   /
/dev/disk1s1   233Gi  146Gi   74Gi    67% 1045764 2447055556    0%   /System/Volumes/Data
/dev/disk1s4   233Gi  2.0Gi   74Gi     3%       2 2448101318    0%   /private/var/vm
```

更多时候，磁盘问题还是性能上的问题。我们可以通过iostat`iostat -d -k -x`来进行分析

```
root@backend-test07:~# iostat -d -k -x 1
Linux 3.19.0-25-generic (backend-test07)        11/03/2019      _x86_64_        (2 CPU)

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
vda               0.00    75.40    0.69    4.77     3.88   647.00   238.37     0.05    8.64    3.12    9.45   2.89   1.58
dm-0              0.00     0.00    0.54   79.84     3.27   646.47    16.17     0.10    1.26    3.39    1.25   0.20   1.57
dm-1              0.00     0.00    0.15    0.13     0.61     0.53     8.00     0.04  138.40    2.20  295.39   0.69   0.02

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
vda               0.00     0.00    0.00    6.00     0.00    20.00     6.67     0.00    0.67    0.00    0.67   0.67   0.40
dm-0              0.00     0.00    0.00    5.00     0.00    20.00     8.00     0.00    0.80    0.00    0.80   0.80   0.40
dm-1              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00    0.00    0.00    0.00   0.00   0.00
```

最后一列`%util`可以看到每块磁盘写入的程度，而`rrqpm/s`以及`wrqm/s`分别表示读写速度，一般就能帮助定位到具体哪块磁盘出现问题了。

另外我们还需要知道是哪个进程在进行读写，一般来说开发自己心里有数，或者用iotop命令来进行定位文件读写的来源。

```
Total DISK READ :       0.00 B/s | Total DISK WRITE :       3.88 K/s
Actual DISK READ:       0.00 B/s | Actual DISK WRITE:       0.00 B/s
  TID  PRIO  USER     DISK READ  DISK WRITE  SWAPIN     IO>    COMMAND
 1818 be/4 root        0.00 B/s    3.88 K/s  0.00 %  0.00 % java -cp .:/nova/env/Flume/*:/nova/env/Flume/lib/~34545 com.qunhe.logcomplex.logagent.MonitorAgency
    1 be/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % init
    2 be/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [kthreadd]
    3 be/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [ksoftirqd/0]
    5 be/0 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [kworker/0:0H]
    7 be/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [rcu_sched]
    8 be/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [rcu_bh]
    9 be/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [rcuos/0]
   10 be/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [rcuob/0]
   11 rt/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [migration/0]
   12 rt/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [watchdog/0]
   13 rt/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [watchdog/1]
   14 rt/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [migration/1]
   15 be/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [ksoftirqd/1]
10768 be/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % java -Djava.io.tmpdir=/tmp/jetty -Xmx2G -Xms2G -D~ty-deploy.xml /usr/local/jetty/etc/jetty-http.xml
```

不过这边拿到的是tid，我们要转换成pid，可以通过readlink来找到pid`readlink -f /proc/*/task/tid/../..`。

```
root@backend-test07:~# readlink -f /proc/*/task/1818/../..
/proc/1681
```

找到pid之后就可以看这个进程具体的读写情况`cat /proc/pid/io`

```
root@backend-test07:~# cat /proc/1681/io
rchar: 1027941187188
wchar: 8891617768
syscr: 126869002
syscw: 109029325
```

```
syscw: 1090259525
read_bytes: 222339072
write_bytes: 14556504064
cancelled_write_bytes: 1028096
```

我们还可以通过lsof命令来确定具体的文件读写情况`lsof -p pid`

```
root@backend-test07:~# lsof -p 1681
COMMAND  PID USER   FD   TYPE             DEVICE  SIZE/OFF    NODE NAME
java    1681 root  cwd    DIR              252,0      4096  795918 /nova/env/_env/Flume/1.8.9
java    1681 root  rtd    DIR              252,0      4096       2 /
java    1681 root  txt    REG              252,0      6408 2626277 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java
java    1681 root  mem    REG              252,0    252906 1179670 /tmp/monitor_agency/native/libsigar-amd64-linux.so
java    1681 root  mem    REG              252,0    178296 2375234 /usr/lib/jvm/java-8-openjdk-amd64/jre/lib/amd64/libsunec.so
java    1681 root  DEL    REG              252,0           393479 /lib/x86_64-linux-gnu/libresolv-2.19.so
java    1681 root  DEL    REG              252,0           393442 /lib/x86_64-linux-gnu/libnss_dns-2.19.so
java    1681 root  mem    REG              252,0    634744 2375202 /usr/lib/jvm/java-8-openjdk-amd64/jre/lib/jsse.jar
java    1681 root  mem    REG              252,0     72720 2375223 /usr/lib/jvm/java-8-openjdk-amd64/jre/lib/amd64/libnio.so
java    1681 root  mem    REG              252,0     39328 2375215 /usr/lib/jvm/java-8-openjdk-amd64/jre/lib/amd64/libmanagement.so
java    1681 root  mem    REG              252,0     95632 2375194 /usr/lib/jvm/java-8-openjdk-amd64/jre/lib/jce.jar
java    1681 root  mem    REG              252,0     97096 2375226 /usr/lib/jvm/java-8-openjdk-amd64/jre/lib/amd64/libnet.so
java    1681 root  mem    REG              252,0   2575022  795973 /nova/env/_env/Flume/1.8.9/lib/guava-22.0.jar
java    1681 root  mem    REG              252,0   2889817  795972 /nova/env/_env/Flume/1.8.9/lib/sigar-loader-1.6.6-rev002.jar
java    1681 root  mem    REG              252,0    326724  795971 /nova/env/_env/Flume/1.8.9/lib/httpcore-4.4.4.jar
java    1681 root  mem    REG              252,0     15009  795970 /nova/env/_env/Flume/1.8.9/lib/uuid-3.2.jar
java    1681 root  mem    REG              252,0      4968  795969 /nova/env/_env/Flume/1.8.9/lib/metrics-0.0.1.jar
java    1681 root  mem    REG              252,0   1581066  795968 /nova/env/_env/Flume/1.8.9/lib/mockito-all-1.9.5.jar
java    1681 root  mem    REG              252,0      9748  795967 /nova/env/_env/Flume/1.8.9/lib/slf4j-log4j12-1.6.4.jar
java    1681 root  mem    REG              252,0    236880  795966 /nova/env/_env/Flume/1.8.9/lib/lz4-1.3.0.jar
java    1681 root  mem    REG              252,0    263965  795965 /nova/env/_env/Flume/1.8.9/lib/commons-codec-1.9.jar
java    1681 root  mem    REG              252,0     45024  795964 /nova/env/_env/Flume/1.8.9/lib/hamcrest-core-1.3.jar
java    1681 root  mem    REG              252,0     33015  795963 /nova/env/_env/Flume/1.8.9/lib/jsr305-1.3.9.jar
java    1681 root  mem    REG              252,0     57264  795962 /nova/env/_env/Flume/1.8.9/lib/json-20170516.jar
java    1681 root  mem    REG              252,0    412739  795961 /nova/env/_env/Flume/1.8.9/lib/commons-lang3-3.3.2.jar
java    1681 root  mem    REG              252,0      8782  795960 /nova/env/_env/Flume/1.8.9/lib/j2objc-annotations-1.1.jar
java    1681 root  mem    REG              252,0     41071  795959 /nova/env/_env/Flume/1.8.9/lib/slf4j-api-1.7.21.jar
java    1681 root  mem    REG              252,0   1143162  795958 /nova/env/_env/Flume/1.8.9/lib/jackson-databind-2.5.3.jar
java    1681 root  mem    REG              252,0    229650  795957 /nova/env/_env/Flume/1.8.9/lib/gson-2.6.2.jar
java    1681 root  mem    REG              252,0     12078  795956 /nova/env/_env/Flume/1.8.9/lib/error_prone_annotations-2.0.18.jar
java    1681 root  mem    REG              252,0     39815  795955 /nova/env/_env/Flume/1.8.9/lib/jackson-annotations-2.5.0.jar
java    1681 root  mem    REG              252,0    736658  795954 /nova/env/_env/Flume/1.8.9/lib/httpclient-4.5.2.jar
```

内存

内存问题排查起来相对比CPU麻烦一些，场景也比较多。主要包括./。一般来讲，我们会先用`free`命令先来检查一发内存的各种情况。

```
sh-4.2# free
              total        used        free      shared  buff/cache   available
Mem:        4047500     2530092      177572         240     1339836     1242304
Swap:       1044476      345908      698568
```

堆内内存

内存问题大多还都是堆内内存问题。表象上主要分为OOM和StackOverflow。

OOM

JVM中的内存不足，OOM大致可以分为以下几种：

Exception in thread "main" java.lang.OutOfMemoryError: unable to create new native thread

这个意思是没有足够的内存空间给线程分配java栈，基本上还是线程池代码写的有问题，比如说忘记shutdown，所以说应该首先从代码层面来寻找问题，使用jstack或者jmap。如果一切都正常，JVM方面可以通过指定`Xss`来减少单个thread stack的大小。另外也可以在系统层面，可以通过修改`/etc/security/limits.conf` nofile和nproc来增大os对线程的限制

```
* soft nofile 65535
* hard nofile 65535
* soft nproc 65535
* hard nproc 65535
```

Exception in thread "main" java.lang.OutOfMemoryError: Java heap space

这个意思是堆的内存占用已经达到-Xmx设置的最大值，应该是最常见的OOM错误了。解决思路仍然是先应该在代码中找，怀疑存在内存泄漏，通过jstack和jmap去定位问题。如果说一切都正常，才需要通过调整`Xmx`的值来扩大内存。

Caused by: java.lang.OutOfMemoryError: Meta space

这个意思是元数据区的内存占用已经达到`XX:MaxMetaspaceSize`设置的最大值，排查思路和上面的一致，参数方面可以通过`XX:MaxMetaspaceSize`来进行调整(这里就不说1.8以前的永久代了)。

Stack Overflow

栈内存溢出，这个大家见到也比较多。

Exception in thread "main" java.lang.StackOverflowError

- [ ] 表示线程栈需要的内存大于Xss值，同样也是先进行排查，参数方面通过`Xss`来调整，但调整的太大可能又会引起OOM。

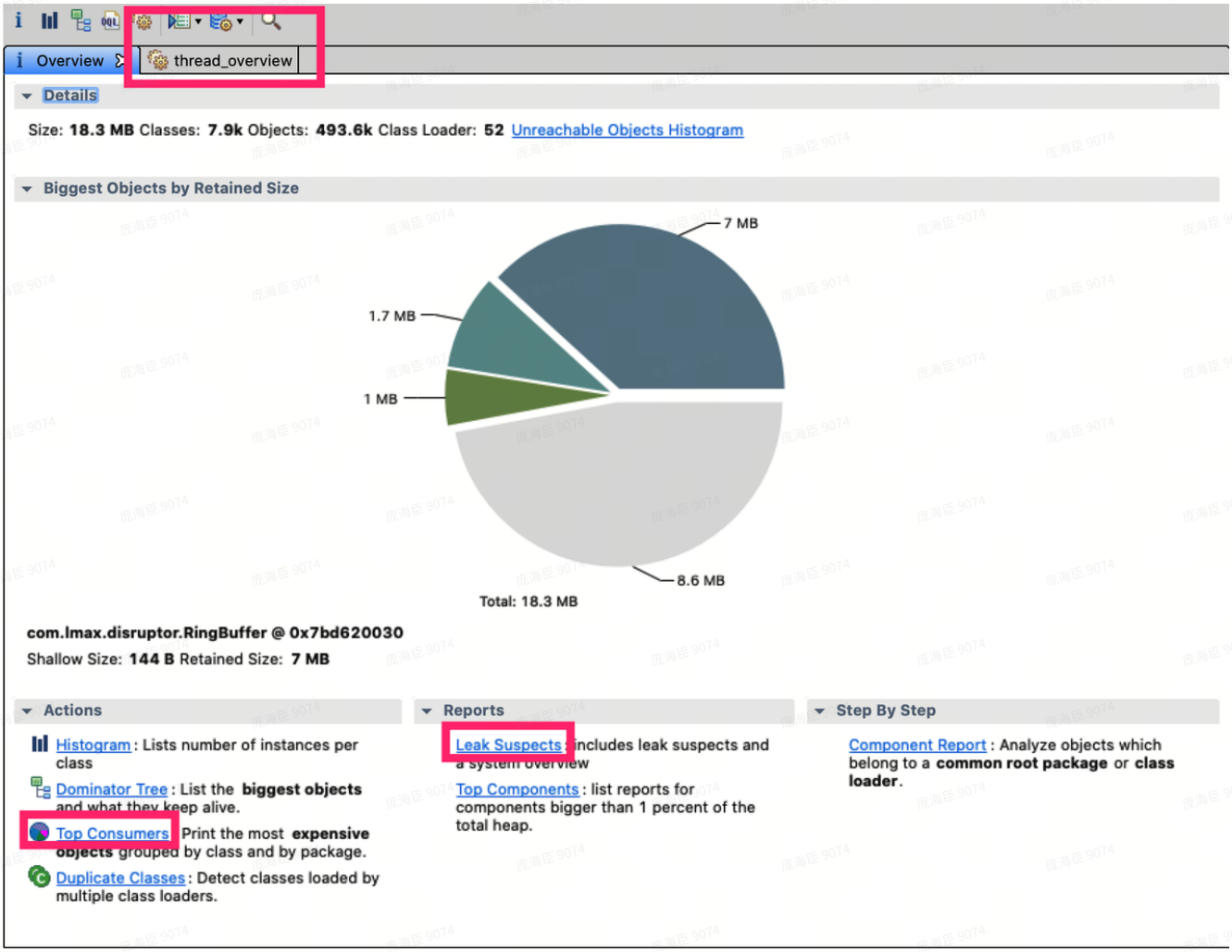
使用JMAP定位代码内存泄漏

上述关于OOM和StackOverflow的代码排查方面，我们一般使用JMAP`jmap -dump:format=b,file=filename pid`来导出dump文件

```
~ jmap -dump:format=b,file=test.hprof 32693
Dumping heap to /Users/fredal/test.hprof ...
```

```
Heap dump file created
```

通过mat(Eclipse Memory Analysis Tools)导入dump文件进行分析，内存泄漏问题一般我们直接选Leak Suspects即可，mat给出了内存泄漏的建议。另外也可以选择Top Consumers来查看最大对象报告。和线程相关的问题可以选择thread overview进行分析。除此之外就是选择Histogram类概览来自己慢慢分析，大家可以搜搜mat的相关教程。



日常开发中，代码产生内存泄漏是比较常见的事，并且比较隐蔽，需要开发者更加关注细节。比如说每次请求都new对象，导致大量重复创建对象；进行文件流操作但未正确关闭；手动不当触发gc；ByteBuffer缓存分配不合理等都会造成代码OOM。

另一方面，我们可以在启动参数中指定`-XX:+HeapDumpOnOutOfMemoryError`来保存OOM时的dump文件。

gc问题和线程

gc问题除了影响cpu也会影响内存，排查思路也是一致的。一般先使用jstat来查看分代变化情况，比如youngGC或者fullGC次数是不是太多呀；EU、OU等指标增长是不是异常呀等。

线程的话太多而且不被及时gc也会引发oom，大部分就是之前说的`unable to create new native thread`。除了jstack细细分析dump文件外，我们一般先会看下总体线程，通过`pstree -p pid |wc -l`。

```
$ pstree -p 10735 | wc -l
47
```

或者直接通过查看`/proc/pid/task`的数量即为线程数量。

```
$ ls -l /proc/10735/task | wc -l
49
```

堆外内存

如果碰到堆外内存溢出，那可真是太不幸了。首先堆外内存溢出表现就是物理常驻内存增长快，报错的话视使用方式都不确定，如果由于使用Netty导致的，那错误日志里可能会出现`OutOfDirectMemoryError`错误，如果直接是DirectByteBuffer，那会报`OutOfMemoryError: Direct buffer memory`。

堆外内存溢出往往是和NIO的使用相关，一般我们先通过pmap来查看下进程占用的内存情况`pmap -x pid | sort -rn -k3 | head -30`，这段意思是查看对应pid倒序前30大的内存段。这边可以再一段时间后再跑一次命令看看内存增长情况，或者和正常机器比较可疑的内存段在哪里。

```
root@backend-test07:~# pmap -x 10735 | sort -rn -k3 |head -30
total kB         4739312 2095936 2059556
0000000080000000 2103544 1885936 1885780 rw---   [ anon ]
00007f66c9000000   38848   37176   35964 rwx--   [ anon ]
00007f6698000000   37136   27036   26980 rw---   [ anon ]
00007f66c07ff000   57348   24588   24588 rw---   [ anon ]
00007f66d8000000   19384   18604   18568 rw---   [ anon ]
00007f669c000000   17868   17624   17596 rw---   [ anon ]
00007f66c802a000    9028    8976    7416 rw---   [ anon ]
00007f66dd924000   13168    6920       0 r-x-- libjvm.so
00007f66c88fc000    7184    6160    6160 rw---   [ anon ]
00007f66dc02c000    5896    5292    5116 rw---   [ anon ]
00007f66861f7000    3064    2972    2300 rw---   [ anon ]
00007f6684bce000    3064    2856    2320 rw---   [ anon ]
00007f66853d0000    3064    2828    2308 rw---   [ anon ]
00007f66b8000000    3404    2536    2536 rw---   [ anon ]
```

```
00007f66b8000000    3404    2536    2536 rw---   [ anon ]
00007f6683f5c000    3064    2408    2304 rw---   [ anon ]
00007f6683b5a000    3064    2132    1016 rw---   [ anon ]
00007f6683758000    3064    2124    1028 rw---   [ anon ]
00007f6687418000    3064    2084     636 rw---   [ anon ]
00007f66868fc000    3064    2060     784 rw---   [ anon ]
00007f6685ef6000    3064    2024     372 rw---   [ anon ]
00007f668425a000    2048    2008     536 rw---   [ anon ]
00007f66850cf000    3064    1996     812 rw---   [ anon ]
00007f668114f000    2048    1992    1588 rw---   [ anon ]
00007f66848c5000    2048    1976     492 rw---   [ anon ]
00007f6686ffa000    2048    1968     432 rw---   [ anon ]
00007f66a43ca000    2048    1964     532 rw---   [ anon ]
00007f6686bfa000    2048    1960     236 rw---   [ anon ]
00007f6687215000    2048    1948     380 rw---   [ anon ]
00007f6685cf3000    2048    1944     572 rw---   [ anon ]
```

我们如果确定有可疑的内存端，需要通过gdb来分析`gdb --batch --pid {pid} -ex "dump memory filename.dump {内存起始地址} {内存起始地址+内存块大小}"`

```
gdb --batch --pid 10735 -ex "dump memory test.dump 0x7f66dc02c000 0x7f66dc02c000+5292"
```

获取dump文件后可用heaxdump进行查看`hexdump -C filename | less`，不过大多数看到的都是二进制乱码。

NMT是Java7U40引入的HotSpot新特性，配合jcmd命令我们就可以看到具体内存组成了。需要在启动参数中加入`-XX:NativeMemoryTracking=summary`或者`-XX:NativeMemoryTracking=detail`，会有略微性能损耗。

一般对于堆外内存缓慢增长直到爆炸的情况来说，可以先设一个基线`jcmd pid VM.native_memory baseline`。

```
~ jcmd 42950 VM.native_memory baseline
42950:
Baseline succeeded
```

然后等放一段时间后再去看看内存增长的情况，通过`jcmd pid VM.native_memory detail.diff(summary.diff)`做一下summary或者detail级别的diff。

```
jcmd 42950 VM.native_memory detail.diff
```

```
Total: reserved=2548758KB +25890KB, committed=1263918KB +25962KB

-                 Java Heap (reserved=1048576KB, committed=1048576KB)
                            (mmap: reserved=1048576KB, committed=1048576KB)

-                     Class (reserved=1102068KB +6419KB, committed=58700KB +6163KB)
                            (classes #8916 +1113)
                            (malloc=12532KB +275KB #11797 +2033)
                            (mmap: reserved=1089536KB +6144KB, committed=46168KB +5888KB)

-                    Thread (reserved=67933KB +17546KB, committed=67933KB +17546KB)
                            (thread #67 +17)
                            (stack: reserved=67584KB +17408KB, committed=67584KB +17408KB)
                            (malloc=207KB +54KB #342 +85)
                            (arena=142KB +84 #129 +34)

-                      Code (reserved=251320KB +66KB, committed=9848KB +394KB)
                            (malloc=1720KB +66KB #4613 +318)
                            (mmap: reserved=249600KB, committed=8128KB +328KB)

-                        GC (reserved=51009KB, committed=51009KB)
                            (malloc=12693KB #227 +2)
                            (mmap: reserved=38316KB, committed=38316KB)

-                  Compiler (reserved=215KB +71KB, committed=215KB +71KB)
                            (malloc=84KB +71KB #444 +102)
                            (arena=131KB #7)

-                  Internal (reserved=13364KB +494KB, committed=13364KB +494KB)
                            (malloc=13332KB +494KB #12887 +1890)
                            (mmap: reserved=32KB, committed=32KB)

-                    Symbol (reserved=11980KB +1009KB, committed=11980KB +1009KB)
                            (malloc=9862KB +1009KB #94142 +11653)
                            (arena=2118KB #1)

-    Native Memory Tracking (reserved=2118KB +288KB, committed=2118KB +288KB)
                            (malloc=140KB +30KB #1948 +404)
                            (tracking overhead=1978KB +258KB)

-               Arena Chunk (reserved=175KB -3KB, committed=175KB -3KB)
                            (malloc=175KB -3KB)
```

可以看到jcmd分析出来的内存十分详细，包括堆内、线程以及gc(所以上述其他内存异常其实都可以用nmt来分析)，这边堆外内存我们重点关注Internal的内存增长，如果增长十分明显的话那就是有问题了。

detail级别的话还会有具体内存段的增长情况，如下图。

```
[0x0000000108bff259] Unsafe_AllocateMemory+0x78
[0x000000010ba18667]
                              (malloc=248KB type=Internal +176KB #40 +28)
```

此外在系统层面，我们还可以使用strace命令来监控内存分配 `strace -f -e "brk,mmap,munmap" -p pid`

这边内存分配信息主要包括了pid和内存地址。

```
root@backend-test07:~# strace -f -e "brk,mmap,munmap" -p 10735
Process 10735 attached with 48 threads
[pid  7227] --- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_ACCERR, si_addr=0x7f66df53ad80} ---
[pid  7226] --- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_ACCERR, si_addr=0x7f66df53b000} ---
[pid 10942] --- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_ACCERR, si_addr=0x7f66df53b000} ---
[pid 10942] --- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_ACCERR, si_addr=0x7f66df53b000} ---
[pid 10942] --- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_ACCERR, si_addr=0x7f66df53b000} ---
[pid 10942] --- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_ACCERR, si_addr=0x7f66df53b000} ---
[pid  7226] --- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_ACCERR, si_addr=0x7f66df53ad00} ---
[pid  7226] --- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_ACCERR, si_addr=0x7f66df53ad00} ---
[pid 10942] --- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_ACCERR, si_addr=0x7f66df53b000} ---
[pid 10942] --- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_ACCERR, si_addr=0x7f66df53a180} ---
```

不过其实上面那些操作也很难定位到具体的问题点，关键还是要看错误日志栈，找到可疑的对象，搞清楚它的回收机制，然后去分析对应的对象。比如DirectByteBuffer分配内存的话，是需要full GC或者手动system.gc来进行回收的(所以最好不要使用 `-XX:+DisableExplicitGC`)。那么其实我们可以跟踪一下DirectByteBuffer对象的内存情况，通过 `jmap -histo:live pid` 手动触发fullGC来看看堆外内存有没有被回收。如果被回收了，那么大概率是堆外内存本身分配的太小了，通过 `-XX:MaxDirectMemorySize` 进行调整。如果没有什么变化，那就要使用jmap去分析那些不能被gc的对象，以及和DirectByteBuffer之间的引用关系了。

GC问题

堆内内存泄漏总是和GC异常相伴。不过GC问题不只是和内存问题相关，还有可能引起CPU负载、网络问题等系列并发症，只是相对来说和内存联系紧密些，所以我们在此单独总结一下GC相关问题。

我们在cpu章介绍了使用jstat来获取当前GC分代变化信息。而更多时候，我们是通过GC日志来排查问题的，在启动参数中加上 `-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintGCTimeStamps` 来开启GC日志。

常见的Young GC、Full GC日志含义在此就不做赘述了。

针对gc日志，我们就能大致推断出youngGC与fullGC是否过于频繁或者耗时过长，从而对症下药。我们下面将对G1垃圾收集器来做分析，这边也建议大家使用G1 `-XX:+UseG1GC`。

youngGC过频繁

- youngGC频繁一般是短周期小对象较多，先考虑是不是Eden区/新生代设置的太小了，看能否通过调整-Xmn、-XX:SurvivorRatio等参数设置来解决问题。如果参数正常，但是young gc频率还是太高，就需要使用Jmap和MAT对dump文件进行进一步排查了。

youngGC耗时过长

耗时过长问题就要看GC日志里耗时耗在哪一块了。以G1日志为例，可以关注Root Scanning、Object Copy、Ref Proc等阶段。Ref Proc耗时长，就要注意引用相关的对象。Root Scanning耗时长，就要注意线程数、跨代引用。Object Copy则需要关注对象生存周期。而且耗时分析它需要横向比较，就是和其他项目或者正常时间段的耗时比较。比如说图中的Root Scanning和正常时间段比增长较多，那就是起的线程太多了。

```
2019-11-01T09:03:58.306+0800: 49672.811: [GC pause (G1 Evacuation Pause) (young), 0.0196271 secs]
  [Parallel Time: 16.7 ms, GC Workers: 4]
    [GC Worker Start (ms): Min: 49672811.7, Avg: 49672811.9, Max: 49672812.3, Diff: 0.5]
    [Ext Root Scanning (ms): Min: 3.5, Avg: 4.2, Max: 5.6, Diff: 2.1, Sum: 17.0]
    [Update RS (ms): Min: 7.9, Avg: 8.1, Max: 8.6, Diff: 0.7, Sum: 32.3]
      [Processed Buffers: Min: 29, Avg: 64.0, Max: 89, Diff: 60, Sum: 256]
    [Scan RS (ms): Min: 0.1, Avg: 0.4, Max: 0.5, Diff: 0.4, Sum: 1.5]
    [Code Root Scanning (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.1]
    [Object Copy (ms): Min: 2.4, Avg: 3.7, Max: 4.4, Diff: 2.1, Sum: 14.9]
    [Termination (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
      [Termination Attempts: Min: 1, Avg: 2.5, Max: 4, Diff: 3, Sum: 10]
    [GC Worker Other (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.3]
    [GC Worker Total (ms): Min: 16.1, Avg: 16.5, Max: 16.7, Diff: 0.6, Sum: 66.1]
    [GC Worker End (ms): Min: 49672828.4, Avg: 49672828.4, Max: 49672828.4, Diff: 0.0]
  [Code Root Fixup: 0.0 ms]
  [Code Root Purge: 0.0 ms]
  [Clear CT: 0.6 ms]
  [Other: 2.3 ms]
    [Choose CSet: 0.0 ms]
    [Ref Proc: 0.7 ms]
    [Ref Enq: 0.0 ms]
    [Redirty Cards: 0.1 ms]
    [Humongous Register: 0.1 ms]
    [Humongous Reclaim: 0.1 ms]
    [Free CSet: 1.0 ms]
  [Eden: 2446.0M(2446.0M)->0.0B(2446.0M) Survivors: 10.0M->10.0M Heap: 2927.3M(4096.0M)->456.5M(4096.0M)]
[Times: user=0.06 sys=0.00, real=0.02 secs]
```

触发fullGC

G1中更多的还是mixedGC，但mixedGC可以和youngGC思路一样去排查。触发fullGC了一般都会有问题，G1会退化使用Serial收集器来完成垃圾的清理工作，暂停时长达到秒级别，可以说是半跪

了。

fullGC的原因可能包括以下这些，以及参数调整方面的一些思路：

- 并发阶段失败：在并发标记阶段，MixGC之前老年代就被填满了，那么这时候G1就会放弃标记周期。这种情况，可能就需要增加堆大小，或者调整并发标记线程数`-XX:ConcGCThreads`。
- 晋升失败：在GC的时候没有足够的内存供存活/晋升对象使用，所以触发了Full GC。这时候可以通过`-XX:G1ReservePercent`来增加预留内存百分比，减少`-XX:InitiatingHeapOccupancyPercent`来提前启动标记，`-XX:ConcGCThreads`来增加标记线程数也是可以的。
- 大对象分配失败：大对象找不到合适的region空间进行分配，就会进行fullGC，这种情况下可以增大内存或者增大`-XX:G1HeapRegionSize`。
- 程序主动执行System.gc()：不要随便写就对了。

另外，我们可以在启动参数中配置`-XX:HeapDumpPath=/xxx/dump.hprof`来dump fullGC相关的文件，并通过jinfo来进行gc前后的dump

```
jinfo -flag +HeapDumpBeforeFullGC pid
jinfo -flag +HeapDumpAfterFullGC pid
```

这样得到2份dump文件，对比后主要关注被gc掉的问题对象来定位问题。

网络

涉及到网络层面的问题一般都比较复杂，场景多，定位难，成为了大多数开发的噩梦，应该是最复杂的了。这里会举一些例子，并从tcp层、应用层以及工具的使用等方面进行阐述。

超时

超时错误大部分处在应用层面，所以这块着重理解概念。超时大体可以分为连接超时和读写超时，某些使用连接池的客户端框架还会存在获取连接超时和空闲连接清理超时。

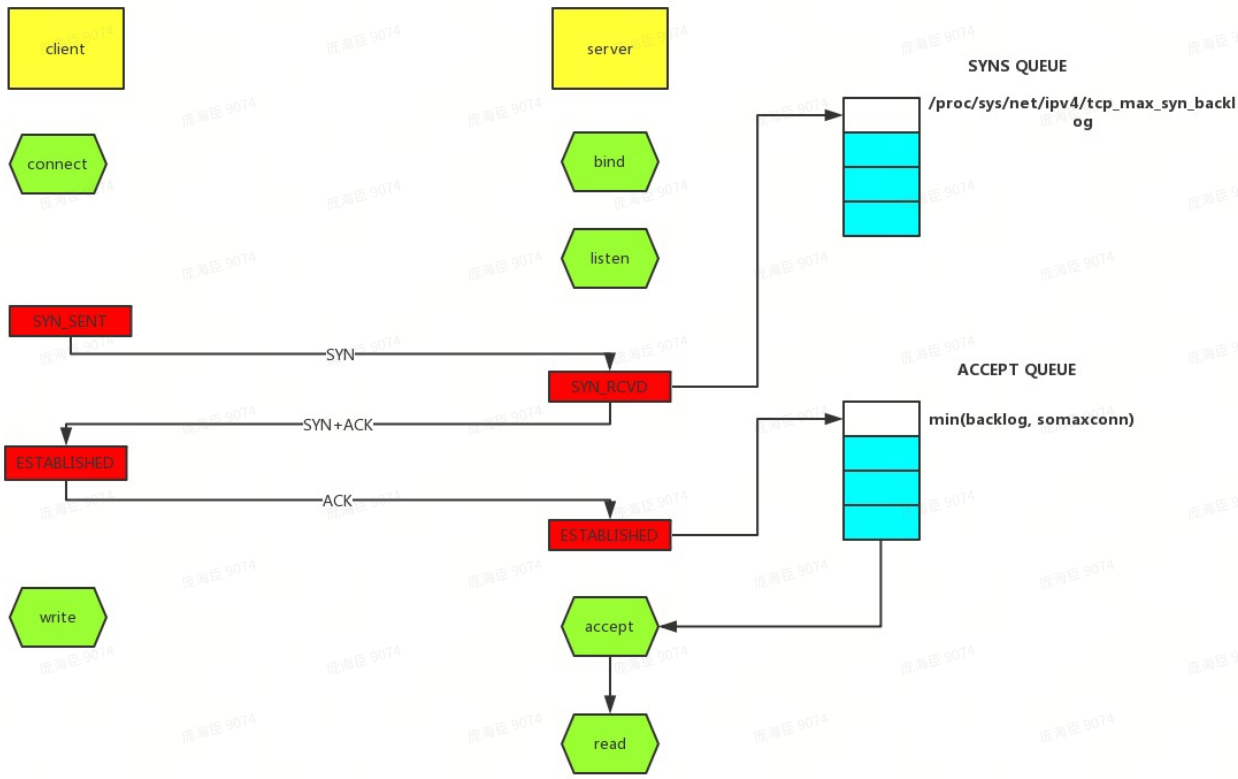
- 读写超时。readTimeout/writeTimeout，有些框架叫做so_timeout或者socketTimeout，均指的是数据读写超时。注意这边的超时大部分是指逻辑上的超时。soa的超时指的也是读超时。读写超时一般都只针对客户端设置。
- 连接超时。connectionTimeout，客户端通常指与服务端建立连接的最大时间。服务端这边connectionTimeout就有些五花八门了，jetty中表示空闲连接清理时间，tomcat则表示连接维持的最大时间。
- 其他。包括连接获取超时connectionAcquireTimeout和空闲连接清理超时idleConnectionTimeout。多用于使用连接池或队列的客户端或服务端框架。
-

我们在设置各种超时时间中，需要确认的是尽量保持客户端的超时小于服务端的超时，以保证连接正常结束。

在实际开发中，我们关心最多的应该是接口的读写超时了。如何设置合理的接口超时是一个问题。如果接口超时设置的过长，那么有可能会过多地占用服务端的tcp连接。而如果接口设置的过短，那么接口超时就会非常频繁。服务端接口明明rt降低，但客户端仍然一直超时又是另一个问题。这个问题其实很简单，客户端到服务端的链路包括网络传输、排队以及服务处理等，每一个环节都可能是耗时的原因。

TCP队列溢出

tcp队列溢出是个相对底层的错误，它可能会造成超时、rst等更表层的错误。因此错误也更隐蔽，所以我们单独说一说。



如上图所示，这里有两个队列：syns queue(半连接队列）、accept queue（全连接队列）。三次握手，在server收到client的syn后，把消息放到syns queue，回复syn+ack给client，server收到client的ack，如果这时accept queue没满，那就从syns queue拿出暂存的信息放入accept queue中，否则按tcp_abort_on_overflow指示的执行。

tcp_abort_on_overflow 0表示如果三次握手第三步的时候accept queue满了那么server扔掉client发过来的ack。tcp_abort_on_overflow 1则表示第三步的时候如果全连接队列满了，server发送一个rst包给client，表示废掉这个握手过程和这个连接，意味着日志里可能会有很多`connection reset / connection reset by peer`。

那么在实际开发中，我们怎么能快速定位到tcp队列溢出呢？

netstat命令，执行netstat -s | egrep "listen|LISTEN"

```
[yuntian@site-brandcenter-prod01 ~]$ netstat -s | egrep "listen|LISTEN"
    99 times the listen queue of a socket overflowed
    99 SYNs to LISTEN sockets dropped
```

如上图所示，overflowed表示全连接队列溢出的次数，sockets dropped表示半连接队列溢出的次数。

ss命令，执行ss -lnt

```
State       Recv-Q Send-Q Local Address:Port               Peer Address:Port

LISTEN      0      5            *:12071                    *:*
```

上面看到Send-Q 表示第三列的listen端口上的全连接队列最大为5，第一列Recv-Q为全连接队列当前使用了多少。

接着我们看看怎么设置全连接、半连接队列大小吧：

全连接队列的大小取决于min(backlog, somaxconn)。backlog是在socket创建的时候传入的，somaxconn是一个os级别的系统参数。而半连接队列的大小取决于max(64, /proc/sys/net/ipv4/tcp_max_syn_backlog)。

在日常开发中，我们往往使用servlet容器作为服务端，所以我们有时候也需要关注容器的连接队列大小。在tomcat中backlog叫做`acceptCount`，在jetty里面则是`acceptQueueSize`。

RST异常

RST包表示连接重置，用于关闭一些无用的连接，通常表示异常关闭，区别于四次挥手。

在实际开发中，我们往往会看到`connection reset / connection reset by peer`错误，这种情况就是RST包导致的。

端口不存在

如果像不存在的端口发出建立连接SYN请求，那么服务端发现自己并没有这个端口则会直接返回一个RST报文，用于中断连接。

主动代替FIN终止连接

一般来说，正常的连接关闭都是需要通过FIN报文实现，然而我们也可以用RST报文来代替FIN，表示直接终止连接。实际开发中，可设置SO_LINGER数值来控制，这种往往是故意的，来跳过TIMED_WAIT，提供交互效率，不闲就慎用。

客户端或服务端有一边发生了异常，该方向对端发送RST以告知关闭连接

我们上面讲的tcp队列溢出发送RST包其实也是属于这一种。这种往往是由于某些原因，一方无法再能正常处理请求连接了(比如程序崩了，队列满了)，从而告知另一方关闭连接。

接收到的TCP报文不在已知的TCP连接内

比如，一方机器由于网络实在太差TCP报文失踪了，另一方关闭了该连接，然后过了许久收到了之前失踪的TCP报文，但由于对应的TCP连接已不存在，那么会直接发一个RST包以便开启新的连接。

一方长期未收到另一方的确认报文，在一定时间或重传次数后发出RST报文

这种大多也和网络环境相关了，网络环境差可能会导致更多的RST报文。

之前说过RST报文多会导致程序报错，在一个已关闭的连接上读操作会报`connection reset`，而在一个已关闭的连接上写操作则会报`connection reset by peer`。通常我们可能还会看到`broken pipe`错误，这是管道层面的错误，表示对已关闭的管道进行读写，往往是在收到RST，报出`connection reset`错后继续读写数据报的错，这个在glibc源码注释中也有介绍。

我们在排查故障时候怎么确定有RST包的存在呢？当然是使用tcpdump命令进行抓包，并使用wireshark进行简单分析了。`tcpdump -i en0 tcp -w xxx.cap`，en0表示监听的网卡。

```
sh-3.2# tcpdump -i en0 tcp -w soatest.cap
tcpdump: listening on en0, link-type EN10MB (Ethernet), capture size 262144 bytes
```

接下来我们通过wireshark打开抓到的包，可能就能看到如下图所示，红色的就表示RST包了。

```
80 → 53196 [ACK] Seq=1 Ack=479 Win=30336 Len=0
53196 → 80 [RST, ACK] Seq=581 Ack=1 Win=0 Len=0
[TCP Dup ACK 2407#1] 80 → 53196 [ACK] Seq=1 Ack=479 Win=30336 Len=0
53200 → 80 [SYN] Seq=0 Win=8192 Len=0 MSS=1428 WS=256 SACK_PERM=1
80 → 53200 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1 WS=128
53196 → 80 [RST] Seq=479 Win=0 Len=0
```

TIME_WAIT和CLOSE_WAIT

TIME_WAIT和CLOSE_WAIT是啥意思相信大家都知道。

在线上时，我们可以直接用命令`netstat -n | awk '/^tcp/ {++S[$NF]} END {for(a in S) print a, S[a]}'`来查看time-wait和close_wait的数量

用ss命令会更快`ss -ant | awk '{++S[$1]} END {for(a in S) print a, S[a]}'`

```
~ netstat -n | awk '/^tcp/ {++S[$NF]} END {for(a in S) print a, S[a]}'
CLOSE_WAIT 1
```

```
TIME_WAIT 2
ESTABLISHED 53
```

TIME_WAIT

time_wait的存在一是为了丢失的数据包被后面连接复用，二是为了在2MSL的时间范围内正常关闭连接。它的存在其实会大大减少RST包的出现。

过多的time_wait在短连接频繁的场景比较容易出现。这种情况可以在服务端做一些内核参数调优:

```
#表示开启重用。允许将TIME-WAIT sockets重新用于新的TCP连接，默认为0，表示关闭
net.ipv4.tcp_tw_reuse = 1
#表示开启TCP连接中TIME-WAIT sockets的快速回收，默认为0，表示关闭
net.ipv4.tcp_tw_recycle = 1
```

当然我们不要忘记在NAT环境下因为时间戳错乱导致数据包被拒绝的坑了，另外的办法就是改小`tcp_max_tw_buckets`，超过这个数的time_wait都会被干掉，不过这也会导致报`time wait bucket table overflow`的错。

CLOSE_WAIT

close_wait往往都是因为应用程序写的有问题，没有在ACK后再次发起FIN报文。close_wait出现的概率甚至比time_wait要更高，后果也更严重。往往是由于某个地方阻塞住了，没有正常关闭连接，从而渐渐地消耗完所有的线程。

想要定位这类问题，最好是通过jstack来分析线程堆栈来排查问题，具体可参考上述章节。这里仅举一个例子。

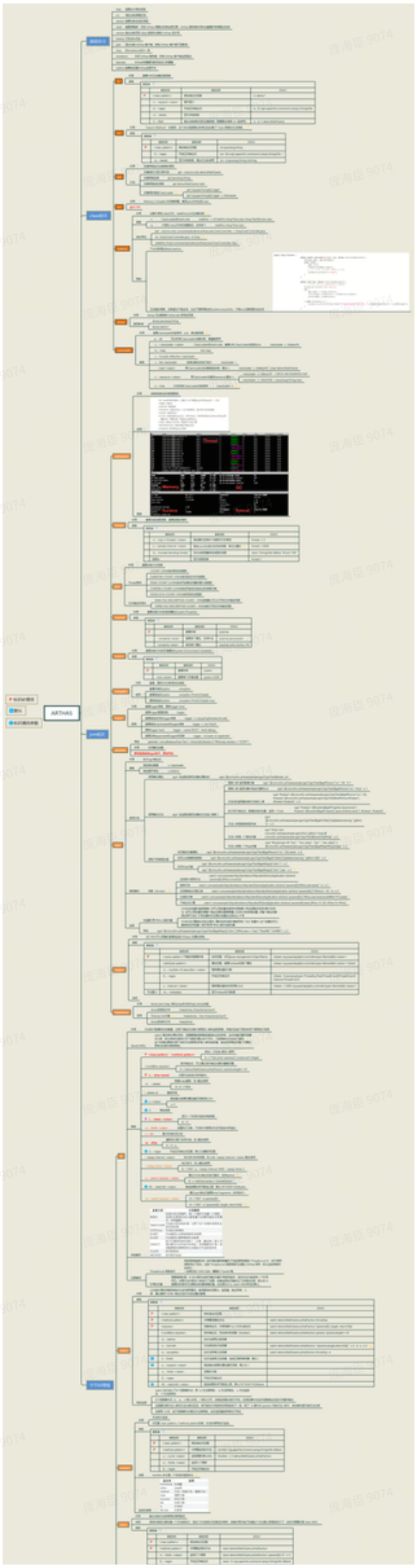
开发同学说应用上线后CLOSE_WAIT就一直增多，直到挂掉为止，jstack后找到比较可疑的堆栈是大部分线程都卡在了`countdownlatch.await`方法，找开发同学了解后得知使用了多线程但是确没有catch异常，修改后发现异常仅仅是最简单的升级sdk后常出现的`class not found`。

以上内容由运维独家搜集整理。

> 标题：JAVA线上故障排查全套路
>
> 作者：fredalxin

> 地址：https://fredal.xin/java-error-check

Arthas 工具命令



Java具体故障分析

1、Java druid连接耗时过长

问题：查看应用调用链监控发现某个接口在druid/connecton上花费了较长时间，尤其在跨机房访问明显（蓝组的应用访问绿组的数据库）

分析：初步查看以为是蓝组跨网络导致花费，无计可施，后经过与其他应用druid/connecton上花费对比发现，有问题的应用在绿组上的花费也还是要比正常的应用少多些，因此判断该问题不是跨机房导致，是跨机房将问题放大了。最后经开发对比确认发现其druid驱动版本过高，将其降低，应用恢复正常。