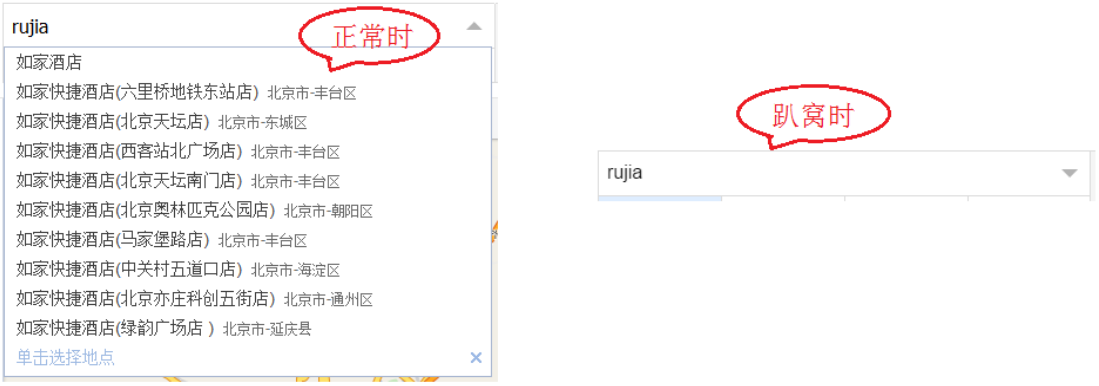
目 录 (下 篇)

联系邮箱：editor@51testing.com

后台性能测试不可不知的二三事

◆腾讯 TMQ：吴 昊

某月黑风高之夜，某打车平台上线了一大波（G+）优惠活动，意在约饭、约酒、约 P 的众人纷纷下单。于是乎，该打车平台使用的智能提示服务扛不住直接趴窝了（如下图）。事后，负责智能提示服务开发和运维的有关部门开会后决定：必须对智能提示服务进行一次全面深入的性能摸底，立刻！现在！马上！



那么一大坨问题就迎面而来：对于智能提示这样的后台服务，性能测试过程中应该关心那些指标？这些指标代表什么含义？这些指标的通过标准是什么？下面将为您一一解答。

一、概述

不同人群关注的性能指标各有侧重。后台服务接口的调用者一般只关心吞吐量、响应时间等外部指标。后台服务的所有者不仅仅关注外部指标，还会关注 CPU、内存、负载等内部指标

拿某打车平台来说，它所关心的是智能提示的外部指标能不能抗住因大波优惠所导致的流量激增。而对于智能提示服务的开发、运维、测试人员，不仅仅关注外部指标，还会关注 CPU、内存、IO 等内部指标，以及部署方式、服务器软硬件配置等运维相关事项。

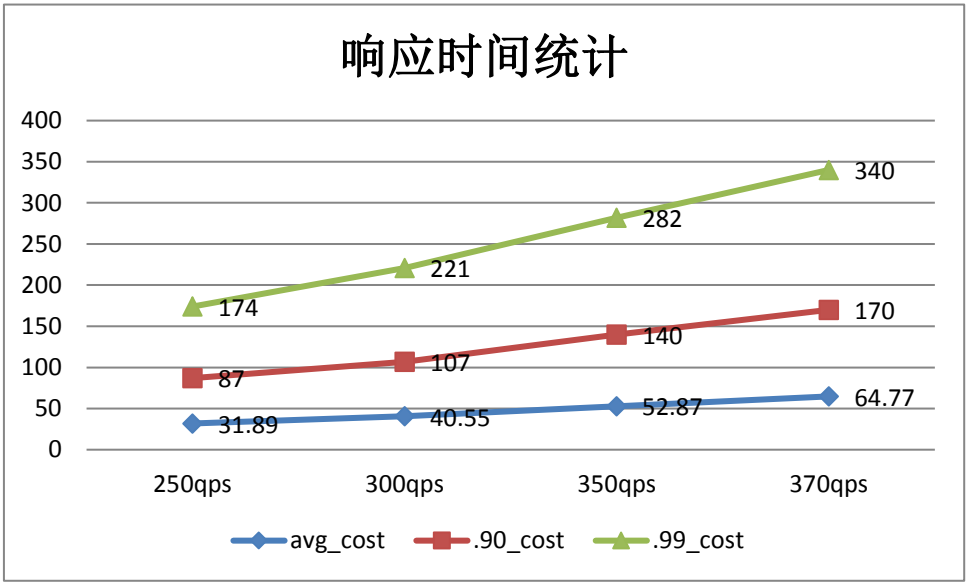
二、外部指标

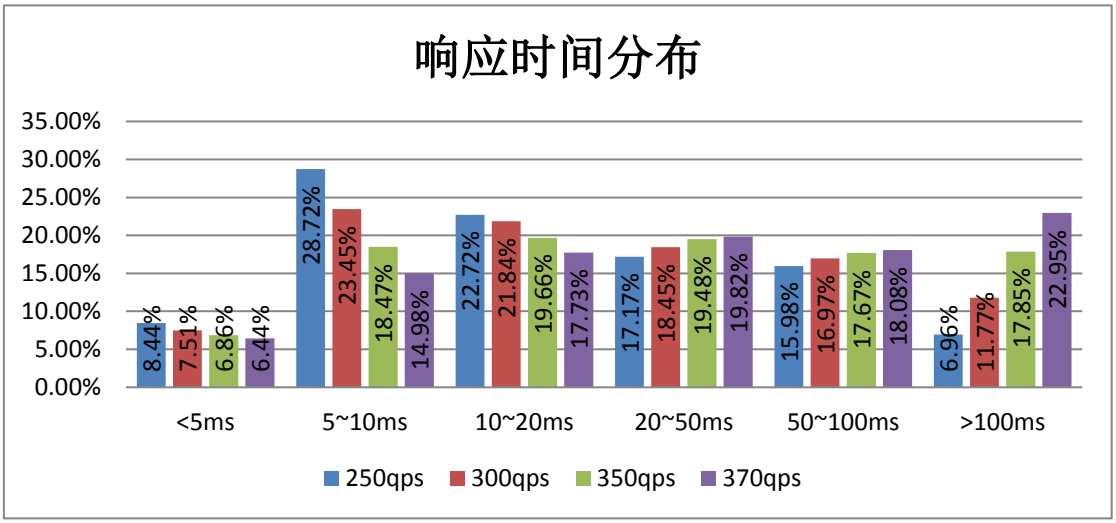
从外部看，性能测试主要关注如下三个指标

- 吞吐量：每秒钟系统能够处理的请求数、任务数。
- 响应时间：服务处理一个请求或一个任务的耗时。
- 错误率：一批请求中结果出错的请求所占比例。

响应时间的指标取决于具体的服务。如智能提示一类的服务，返回的数据有效周期短（用户多输入一个字母就需要重新请求），对实时性要求比较高，响应时间的上限一般在 100ms 以内。而导航一类的服务，由于返回结果的使用周期比较长（整个导航过程中），响应时间的上限一般在 2-5s。

对于响应时间的统计，应从均值、.90、.99、分布等多个角度统计，而不仅仅是给出均值。下图是响应时间统计的一个例子





吞吐量的指标受到响应时间、服务器软硬件配置、网络状态等多方面因素影响。

吞吐量越大，响应时间越长。

服务器硬件配置越高，吞吐量越大。

网络越差，吞吐量越小。

在低吞吐量下的响应时间的均值、分布比较稳定，不会产生太大的波动。

在高吞吐量下，响应时间会随着吞吐量的增长而增长，增长的趋势可能是线性的，也可能接近指数的。当吞吐量接近系统的峰值时，响应时间会出现激增。

错误率和服务的具体实现有关。通常情况下，由于网络超时等外部原因造成的错误比例不应超过 5%%，由于服务本身导致的错误率不应超过 1%

三、内部指标

从服务器的角度看，性能测试主要关注 CPU、内存、服务器负载、网络、磁盘 IO 等

1. CPU

后台服务的所有指令和数据处理都是由 CPU 负责，服务对 CPU 的利用率对服务的性能起着决定性的作用。

Linux 系统的 CPU 主要有如下几个维度的统计数据

- us: 用户态使用的 cpu 时间百分比
- sy: 系统态使用的 cpu 时间百分比
- ni: 用做 nice 加权的进程分配的用户态 cpu 时间百分比
- id: 空闲的 cpu 时间百分比
- wa: cpu 等待 IO 完成时间百分比
- hi: 硬中断消耗时间百分比
- si: 软中断消耗时间百分比

下图是线上开放平台转发服务某台服务器上 top 命令的输出，下面以这个服务为例对 CPU 各项指标进行说明

```
top - 11:20:18 up 1407 days, 18:19,  1 user,  load average: 0.30, 0.28, 0.24
Tasks: 249 total,   1 running, 246 sleeping,   1 stopped,   1 zombie
Cpu0  :  9.2%us,  2.6%sy,  0.0%ni, 73.6%id,  0.0%wa,  0.0%hi, 14.5%si,  0.0%st
Cpu1  :  1.3%us,  1.7%sy,  0.0%ni, 97.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu2  :  0.7%us,  0.3%sy,  0.0%ni, 99.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu3  :  0.3%us,  1.0%sy,  0.0%ni, 98.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  32948468k total, 32665284k used,   283184k free,   844576k buffers
Swap:  2008116k total,  1102200k used,   905916k free, 22748808k cached
```

us & sy: 大部分后台服务使用的 CPU 时间片中 us 和 sy 的占用比例是最高的。同时这两个指标又是互相影响的，us 的比例高了，sy 的比例就低，反之亦然。通常 sy 比例过高意味着被测服务在用户态和系统态之间切换比较频繁，此时系统整体性能会有一定下降。另外，在使用多核 CPU 的服务器上，CPU 0 负责 CPU 各核间的调度，CPU 0 上的使用率过高会导致其他 CPU 核心之间的调度效率变低。因此测试过程中 CPU 0 需要重点关注。

ni: 每个 Linux 进程都有个优先级，优先级高的进程有优先执行的权利，这个叫做 pri。进程除了优先级外，还有个优先级的修正值。这个修正值就叫做进程的 nice 值。一般来说，被测服务和服务器整体的 ni 值不会很高。如果测试过程中 ni 的值比较高，需要从服务器 Linux 系统配置、被测服务运行参数查找原因

id: 线上服务运行过程中，需要保留一定的 id 冗余来应对突发的流量激增。在性能测试过程中，如果 id 一直很低，吞吐量上不去，需要检查被测服务线程/进程配置、服务器系统配置等。

wa: 磁盘、网络等 IO 操作会导致 CPU 的 wa 指标提高。通常情况下，网络 IO 占用的 wa 资源不会很高，而频繁的磁盘读写会导致 wa 激增。如果被测服务不是 IO 密集型的服务，那需要检查被测服务的日志量、数据载入频率等。

hi & si: 硬中断是外设对 CPU 的中断，即外围硬件发给 CPU 或者内存的异步信号就是硬中断信号；软中断由软件本身发给操作系统内核的中断信号。通常是由硬中断处理程序或进程调度程序对操作系统内核的中断，也就是我们常说的系统调用(System Call)。在性能测试过程中，hi 会有一定的 CPU 占用率，但不会太高。对于 IO 密集型的服务，si 的 CPU 占用率会高一些。

2. 内存

性能测试过程中对内存监控的主要目的是检查被测服务所占用内存的波动情况。

在 Linux 系统中有多个命令可以获取指定进程的内存使用情况，最常用的是 top 命令，如下图所示

```
top - 15:21:18 up 1407 days, 22:20,  1 user,  load average: 0.34, 0.26, 0.19
Tasks:   1 total,   0 running,   1 sleeping,   0 stopped,   0 zombie
Cpu(s):  2.2%us,  0.6%sy,  0.0%ni, 93.7%id,  0.0%wa,  0.0%hi,  3.6%si,  0.0%st
Mem:  32948468k total, 32704476k used,   243992k free,   844700k buffers
Swap:  2008116k total,  1102200k used,   905916k free, 22780552k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  SWAP DATA COMMAND
 1015 mqq       15   0  583m  52m 6984 S   12  0.2 777:40.91 530m 563m node
```

其中

VIRT：进程所使用的虚拟内存的总数。它包括所有的代码，数据和共享库，加上已换出的页面，所有已申请的总内存空间

RES：进程正在使用的没有交换的物理内存（栈、堆），申请内存后该内存段已被重新赋值

SHR：进程使用共享内存的总数。该数值只是反映可能与其它进程共享的内存，不代表这段内存当前正被其他进程使用

SWAP：进程使用的虚拟内存中被换出的大小，交换的是已经申请，但没有使用的空间，包括（栈、堆、共享内存）

DATA：进程除可执行代码以外的物理内存总量，即进程栈、堆申请的总空间

从上面的解释可以看出，测试过程中主要监控 RES 和 VIRT，对于使用了共享内存的多进程架构服务，还需要监控 SHR。

3. LOAD（服务器负载）

Linux 的系统负载指运行队列的平均长度，也就是等待 CPU 的平均进程数

从服务器负载的定义可以看出，服务器运行最理想的状态是所有 CPU 核心的运行队列都为 1，即所有活动进程都在运行，没有等待。这种状态下服务器运行在负载阈值下。

通常情况下，按照经验值，服务器的负载应位于阈值的 70%~80%，这样既能利用服务器大部分性能，又留有一定的性能冗余应对流量增长。

Linux 提供了很多查看系统负载的命令，最常用的是 top 和 uptime

top 和 uptime 针对负载的输出内容相同，都是系统最近 1 分钟、5 分钟、15 分钟的负载均值

```
qqmap@10.169.139.58:~$ top
top - 17:07:59 up 32 days, 46 min,  8 users,  load average: 0.11, 0.15, 0.27
Tasks: 506 total,   1 running, 505 sleeping,   0 stopped,   0 zombie
Cpu(s):  8.9%us,  0.7%sy,  0.0%ni, 90.4%id,  0.0%wa,  0.1%hi,  0.0%si,  0.0%st
Mem:  65676092k total, 23739904k used, 41936188k free,  1980284k buffers
Swap:  2097148k total,        0k used,  2097148k free, 11860332k cached
```

```
qqmap@10.169.139.58:~$ uptime
17:07:38 up 32 days, 46 min,  8 users,  load average: 0.16, 0.17, 0.27
```

查看系统负载阈值的命令如下

```
qqmap@10.169.139.58:~$ cat /proc/cpuinfo |grep 'model name' -c
24
```

在性能测试过程中，系统负载是评价整个系统运行状况最重要的指标之一。通常情况下，压力测试时系统负载应接近但不能超过阈值，并发测试时的系统负载最高不能超过阈值的 80%，稳定性测试时，系统负载应在阈值的 50%左右

4. 网络

性能测试中网络监控主要包括网络流量、网络连接状态的监控。

网络流量监控可以使用 nethogs 命令。该命令与 top 类似，是一个实时交互的命令，运行界面如下

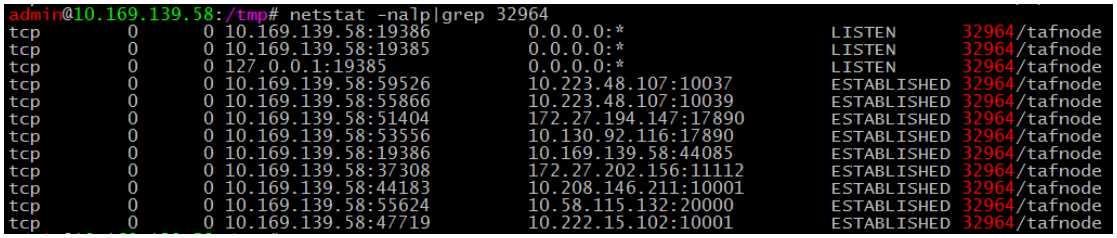
```
NetHogs version 0.8.0

  PID USER     PROGRAM                                          DEV        SENT      RECEIVED
60061 admin    sshd: root@pts/4                                 eth1       0.792       0.457 KB/sec
40637 admin    /usr/local/app/docker/dcnode/bin/dcnode          eth1       0.678       0.241 KB/sec
63445 qqmap    sshd: qqmap@pts/8                                eth1       1.157       0.086 KB/sec
32964 mqq      /usr/local/app/taf/bin/tafnode                   eth1       0.233       0.048 KB/sec
59074 mqq      ..pPoiData.MapRichService/bin/MapRichService     eth1       0.250       0.035 KB/sec
32637 admin    ./fob_idxaccess_dcc3                             eth1       0.044       0.025 KB/sec
32609 admin    ./fob_idxaccess_dcc0                             eth1       0.044       0.025 KB/sec
32618 admin    ./fob_idxaccess_dcc1                             eth1       0.044       0.025 KB/sec
32627 admin    ./fob_idxaccess_dcc2                             eth1       0.044       0.025 KB/sec
32660 admin    ./fob_idxaccess_dcc6                             eth1       0.044       0.025 KB/sec
32646 admin    ./fob_idxaccess_dcc4                             eth1       0.044       0.025 KB/sec
4517  admin    /usr/local/agenttools/agent/agent                eth1       0.000       0.000 KB/sec
32656 admin    ./fob_idxaccess_dcc5                             eth1       0.000       0.000 KB/sec
32773 admin    ./stat_report                                    eth1       0.000       0.000 KB/sec
32726 admin    ./l5_config                                      eth1       0.000       0.000 KB/sec
31025 admin    /usr/local/sa/agent/secu-tcs-agent               eth1       0.000       0.000 KB/sec
?     admin    unknown TCP                                                 0.000       0.000 KB/sec

  TOTAL                                                                    3.371       1.015 KB/sec
```

在后台服务性能测试中，对于返回文本结果的服务，并不需要太多关注在流量方面。

5. 网络连接状态监控

性能测试中对网络的监控主要是监控网络连接状态的变化和异常。对于使用 TCP 协议的服务，需要监控服务已建立连接的变化情况（即 ESTABLISHED 状态的 TCP 连接）。对于 HTTP 协议的服务，需要监控被测服务对应进程的网络缓冲区的状态、TIME_WAIT 状态的连接数等。Linux 自带的很多命令如 netstat、ss 都支持如上功能。下

图是 netstat 对指定 pid 进程的监控结果

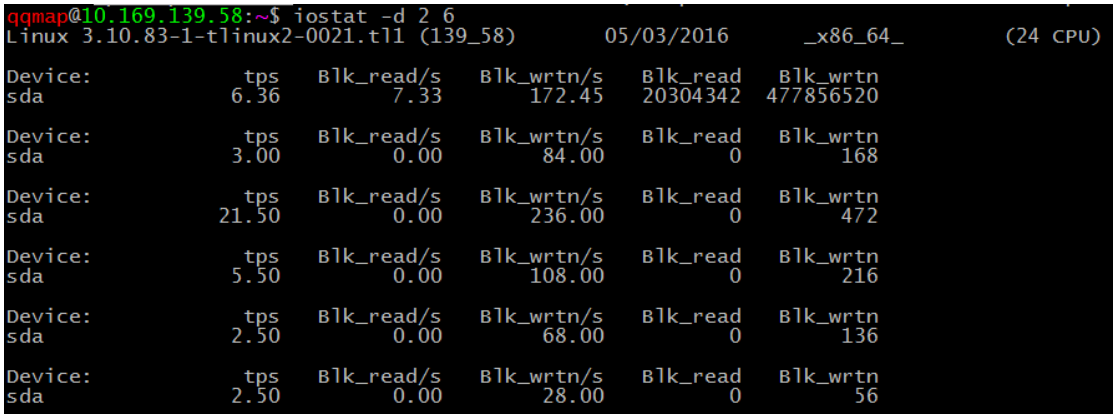


```
admin@10.169.139.58:/tmp# netstat -nalp|grep 32964
tcp        0      0 10.169.139.58:19386         0.0.0.0:*                   LISTEN      32964/tafnode
tcp        0      0 10.169.139.58:19385         0.0.0.0:*                   LISTEN      32964/tafnode
tcp        0      0 127.0.0.1:19385             0.0.0.0:*                   LISTEN      32964/tafnode
tcp        0      0 10.169.139.58:59526         10.223.48.107:10037         ESTABLISHED 32964/tafnode
tcp        0      0 10.169.139.58:55866         10.223.48.107:10039         ESTABLISHED 32964/tafnode
tcp        0      0 10.169.139.58:51404         172.27.194.147:17890        ESTABLISHED 32964/tafnode
tcp        0      0 10.169.139.58:53556         10.130.92.116:17890         ESTABLISHED 32964/tafnode
tcp        0      0 10.169.139.58:19386         10.169.139.58:44085         ESTABLISHED 32964/tafnode
tcp        0      0 10.169.139.58:37308         172.27.202.156:11112        ESTABLISHED 32964/tafnode
tcp        0      0 10.169.139.58:44183         10.208.146.211:10001        ESTABLISHED 32964/tafnode
tcp        0      0 10.169.139.58:55624         10.58.115.132:20000         ESTABLISHED 32964/tafnode
tcp        0      0 10.169.139.58:47719         10.222.15.102:10001         ESTABLISHED 32964/tafnode
```

6. 磁盘 IO

性能测试过程中，如果被测服务对磁盘读写过于频繁，会导致大量请求处于 IO 等待的状态，系统负载升高，响应时间变长，吞吐量下降。

Linux 下可以用 iostat 命令来监控磁盘状态，如下图



```
qqmap@10.169.139.58:~$ iostat -d 2 6
Linux 3.10.83-1-tlinux2-0021.tl1 (139_58)       05/03/2016      _x86_64_        (24 CPU)

Device:            tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
sda               6.36         7.33       172.45   20304342  477856520

Device:            tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
sda               3.00         0.00        84.00          0        168

Device:            tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
sda              21.50         0.00       236.00          0        472

Device:            tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
sda               5.50         0.00       108.00          0        216

Device:            tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
sda               2.50         0.00        68.00          0        136

Device:            tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
sda               2.50         0.00        28.00          0         56
```

tps：该设备每秒的传输次数。“一次传输”意思是“一次 I/O 请求”。多个逻辑请求可能会被合并为“一次 I/O 请求”。“一次传输”请求的大小是未知的。

kB_read/s：每秒从设备（drive expressed）读取的数据量，单位为 Kilobytes

kB_wrtn/s：每秒向设备（drive expressed）写入的数据量，单位为 Kilobytes

kB_read：读取的总数据量，单位为 Kilobytes

kB_wrtn：写入的总数量数据量，单位为 Kilobytes

从 iostat 的输出中，能够获得系统运行最基本的统计数据。但对于性能测试来说，这些数据不能提供更多的信息。需要加上-x 参数

```
qqmap@10.169.139.58:~$ iostat -d 2 6 -x
Linux 3.10.83-1-tlinux2-0021.tl1 (139_58)       05/03/2016      _x86_64_        (24 CPU)

Device:         rrqm/s   wrqm/s     r/s     w/s    rsec/s    wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sda               0.55    15.32    0.15    6.22      7.32    172.42    28.26     0.01    1.09   0.14   0.09

Device:         rrqm/s   wrqm/s     r/s     w/s    rsec/s    wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sda               0.00    21.00    0.00   13.50      0.00    276.00    20.44     0.00    0.00   0.00   0.00

Device:         rrqm/s   wrqm/s     r/s     w/s    rsec/s    wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sda               0.00     4.50    0.00    2.50      0.00     56.00    22.40     0.00    0.00   0.00   0.00

Device:         rrqm/s   wrqm/s     r/s     w/s    rsec/s    wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sda               0.00    17.50    0.00    5.50      0.00    184.00    33.45     0.00    0.00   0.00   0.00

Device:         rrqm/s   wrqm/s     r/s     w/s    rsec/s    wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sda               0.00     0.50    0.00    1.00      0.00     12.00    12.00     0.00    0.00   0.00   0.00

Device:         rrqm/s   wrqm/s     r/s     w/s    rsec/s    wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sda               0.00     9.00    0.00    3.50      0.00    100.00    28.57     0.00    0.00   0.00   0.00
```

rrqm/s：每秒这个设备相关的读取请求有多少被 Merge 了（当系统调用需要读取数据的时候，VFS 将请求发到各个 FS，如果 FS 发现不同的读取请求读取的是相同 Block 的数据，FS 会将这个请求合并 Merge）

wrqm/s：每秒这个设备相关的写入请求有多少被 Merge 了

await：每一个 IO 请求的处理的平均时间（单位是毫秒）

%util：在统计时间内所有处理 IO 时间，除以总共统计时间。例如，如果统计间隔 1 秒，该设备有 0.8 秒在处理 IO，而 0.2 秒闲置，那么该设备的%util = 0.8/1 = 80%，该参数暗示了设备的繁忙程度。

四、常见性能瓶颈

- 吞吐量到上限时系统负载未到阈值：一般是被测服务分配的系统资源过少导致的。测试过程中如果发现此类情况，可以从 ulimit、系统开启的线程数、分配的内存等维度定位问题原因
- CPU 的 us 和 sy 不高，但 wa 很高：如果被测服务是磁盘 IO 密集型型服务，wa 高属于正常现象。但如果不是此类服务，最可能导致 wa 高的原因有两个，一是服务对磁盘读写的业务逻辑有问题，读写频率过高，写入数据量过大，如不合理的数据载入策略、log 过多等，都有可能导致这种问题。二是服务器内存不足，服务在 swap 分区不停的换入换出。
- 同一请求的响应时间忽大忽小：在正常吞吐量下发生此问题，可能的原因有两方面，一是服务对资源的加锁逻辑有问题，导致处理某些请求过程中花了大量的时间等待资源解锁；二是 Linux 本身分配给服务的资源有限，某些请求需要等待其他请求释放资源后才能继续执行。
- 内存持续上涨：在吞吐量固定的前提下，如果内存持续上涨，那么很有可能是

被测服务存在明显的内存泄漏，需要使用 valgrind 等内存检查工具进行定位。

五、举个例子

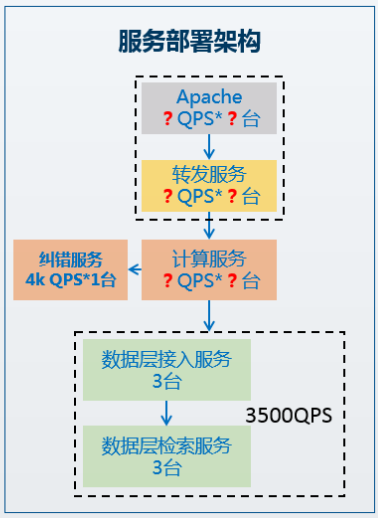
智能提示服务趴窝了以后，必须立刻对其做性能摸底。根据目前的情况，测试结果中需要提供外部指标和内部指标。

智能提示服务的架构和每个模块的功能如下图所示

被测服务各模块功能

Apache	HTTP接入层，将HTTP请求转换为socket请求发送给转发服务
转发服务	根据参数将收到的socket请求转发给不同的业务层
计算服务	业务层，查询底层数据，计算后生成供前端页面渲染的数据
纠错服务	返回查询词的纠错结果
数据层接入服务	底层数据检索接入层
数据层检索服务	检索层，处理数据检索请求

被测服务性能测试的输出
在下游数据服务性能阈值为3500qps的前提下，确定Apache、转发服务、计算服务的**性能阈值**和**部署数量**



从图中我们可以看出，测试前智能提示服务的底层数据服务已经确定了性能上限。因此，本次测试我们的任务是在底层数据服务性能为 3500qps 的前提下，找到智能提示服务上游各个模块的性能上限。

一个完整的后台服务性能测试流程如下图所示。

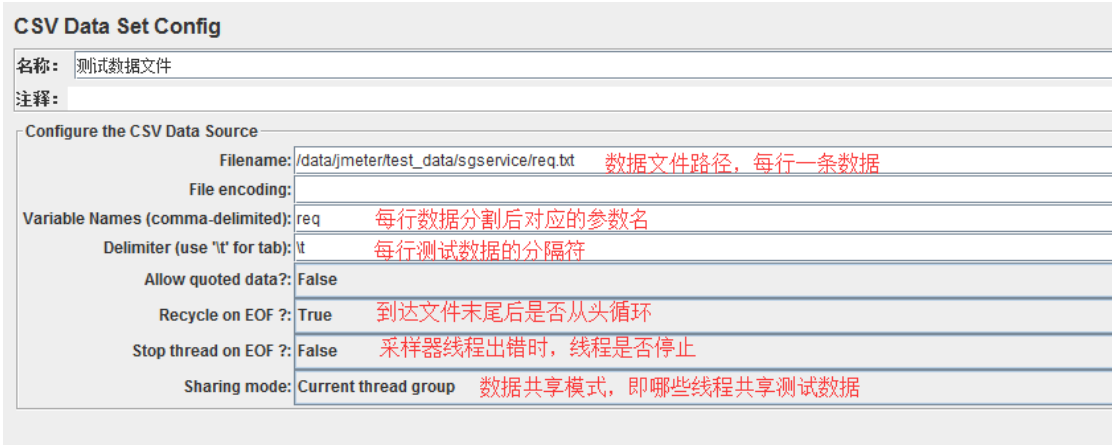
测试前准备:

- 测试数据：由于智能提示已经在线上运行，本次测试使用智能提示趴窝那天的日志作为测试数据
- QPS 预估：本次测试就是为了找这个数
- 服务器配置：使用与线上软硬件配置相同的服务器

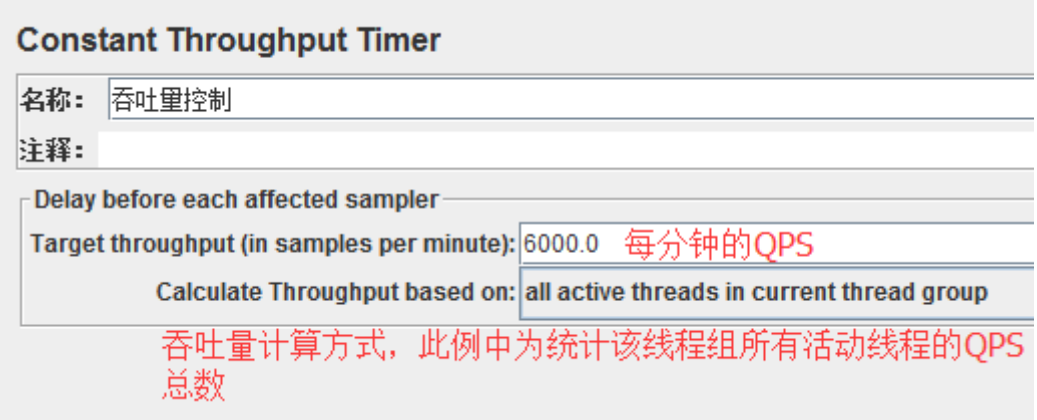
压测过程：

我们使用 Jmeter 发送测试数据来模拟用户请求，Jmeter 测试配置文件使用的原件如下图所示。从图中可以看出，性能测试的配置文件主要由数据文件配置（线程间共享方式、到达末尾时的行为等）、吞吐量控制、HTTP 采样器（域名、端口、HTTP METHOD、请求 body 等）、响应断言（对返回结果的内容进行校验）。

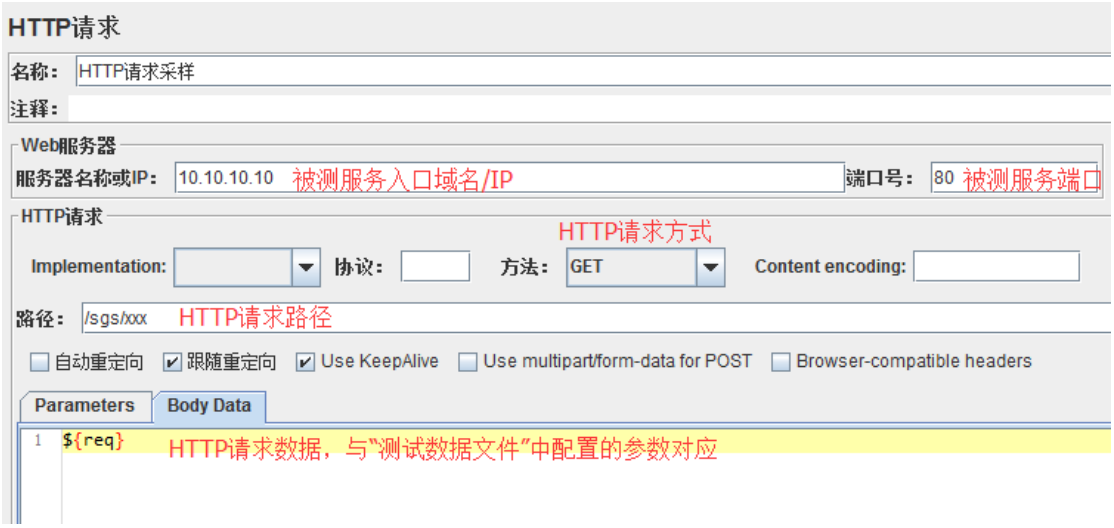
- 数据文件配置



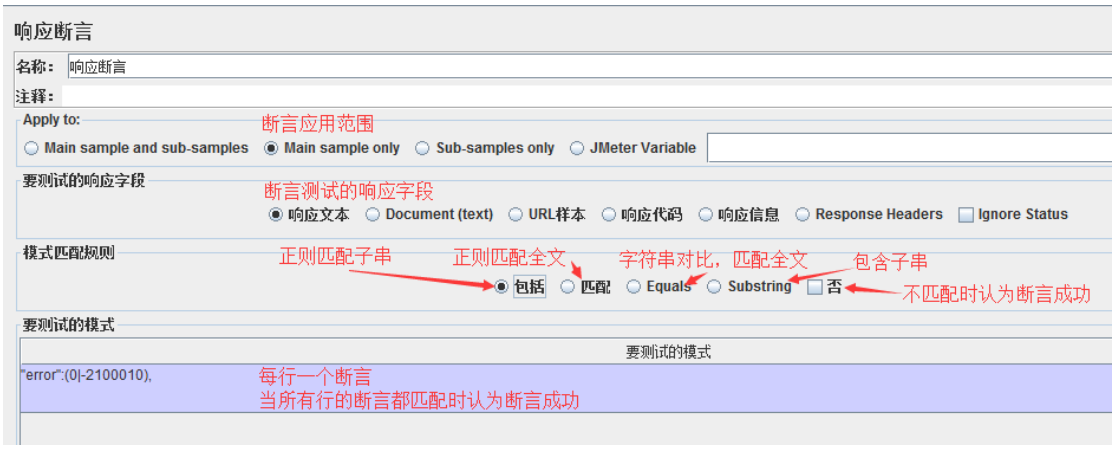
- 吞吐量控制



- HTTP 请求采样

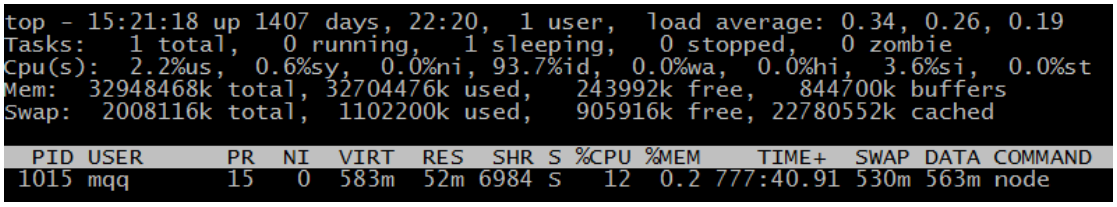


- 响应断言



➢ **CPU**

在 linux 中，sar、top、ps 等命令都可以对 cpu 使用情况进行监控。一般来说，最常用的是 top 命令。top 命令的输出如下:

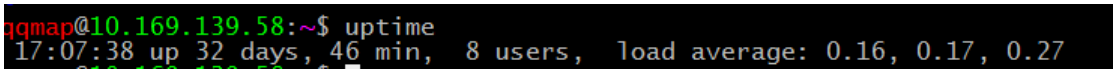


top 命令是一个交互式命令，运行后会一直保持在终端并定时刷新。在性能测试中，可以使用如下参数让 top 命令只运行一次

$top –n 1 –b –p ${pid}

- **服务器负载**

linux 中，服务器负载使用 uptime 命令获取，该命令的输出如下图



每一列的含义如下:

“当前时间、系统运行时长、登录的用户数、最近 1 分钟、5 分钟、15 分钟的平均负载”

- **内存**

在 linux 中，top、ps 命令都可以对指定进程的内存使用状况进行查看。但最准确的信息在/proc/${PID}/status 中，如下图

```
qqmap@10.169.139.58:~$ cat /proc/61284/status
Name:   nginx
State:  S (sleeping)
Tgid:   61284
Pid:    61284
PPid:   61235
TracerPid:      0
Uid:    502     502     502     502
Gid:    502     502     502     502
FDSize: 128
Groups: 502
VmPeak:    49848 kB
VmSize:    49848 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:     27744 kB
VmRSS:     27744 kB
VmData:    27272 kB
VmStk:       136 kB
VmExe:       516 kB
VmLib:      4224 kB
VmPTE:       112 kB
VmSwap:        0 kB
Threads:        1
SigQ:   0/513081
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000040001000
SigCgt: 0000000198016a07
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: 0000001fffffffff
Cpus_allowed:   ffffffff,ffffffff
Cpus_allowed_list:      0-63
Mems_allowed:   00000000,00000000,00000000,00000000,000
00000000,00000000,00000000,00000000,00000000,00000000,0
Mems_allowed_list:      0-1
voluntary_ctxt_switches:        1886808
nonvoluntary_ctxt_switches:     93
```

上面命令的输出中，我们重点关注 VmRSS、VmData、VmSize

- **磁盘 IO**

磁盘监控数据使用 iostat 命令获取

```
qqmap@10.169.139.58:~$ iostat -d 2 6 -x
Linux 3.10.83-1-tlinux2-0021.tl1 (139_58)       05/03/2016      _x86_64_        (24 CPU)

Device:         rrqm/s   wrqm/s     r/s     w/s    rsec/s    wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sda               0.55    15.32    0.15    6.22      7.32    172.42    28.26     0.01    1.09   0.14   0.09

Device:         rrqm/s   wrqm/s     r/s     w/s    rsec/s    wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sda               0.00    21.00    0.00   13.50      0.00    276.00    20.44     0.00    0.00   0.00   0.00

Device:         rrqm/s   wrqm/s     r/s     w/s    rsec/s    wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sda               0.00     4.50    0.00    2.50      0.00     56.00    22.40     0.00    0.00   0.00   0.00

Device:         rrqm/s   wrqm/s     r/s     w/s    rsec/s    wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sda               0.00    17.50    0.00    5.50      0.00    184.00    33.45     0.00    0.00   0.00   0.00

Device:         rrqm/s   wrqm/s     r/s     w/s    rsec/s    wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sda               0.00     0.50    0.00    1.00      0.00     12.00    12.00     0.00    0.00   0.00   0.00

Device:         rrqm/s   wrqm/s     r/s     w/s    rsec/s    wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sda               0.00     9.00    0.00    3.50      0.00    100.00    28.57     0.00    0.00   0.00   0.00
```

测试报告输出

在统计完性能测试过程中收集的监控指标后，就可以输出性能报告了。

通常来说，性能报告中要包含如下内容:

- 测试结论：包括被测服务最大 QPS、响应时间等指标是否达到期望，部署建议等。
- 测试环境描述：包括性能需求、测试用服务器配置、测试数据来源、测试方法等
- 监控指标统计：响应时间统计、QPS、服务器指标统计、进程指标统计。建议最好用图表来表示统计数据。

六、结语

测试完毕后，得出的结论是单台智能提示服务的性能是 300qps，线上整套智能提示服务的性能是 1800qps；而月黑风高那天的流量大概是 5000qps+，难怪智能提示趴窝，确实流量太大，远超线上的吞吐容量。

最后，智能提示服务申请了服务器进行扩容，并对某打车平台的流量进行了限制，既开源又节流，保证今后月黑风高之夜一众约酒、约饭、约 P 之人的打车体验，提高了各种约的成功率，可谓功德无量

APP 测试-安全性测试

◆ 搜狗测试：宋志东

一、前言

在 SDK 最近的项目中上线的包被第三方杀毒软件报出有病毒的问题，后来经过查验发现是 SDK 悬浮窗动画的逻辑被检验出有病毒，最后进行了修改。事情虽然解决了，但是引起该问题的一个原因是在测试中没有安全测试，而安全测试的标准，方法都没有。因此今天将之前工作中参与过的安全测试以及从网上查阅到有关安全测试的资料进行整理。有不足的之处，尽情谅解。

二、软件权限

1）扣费风险：浏览网页，下载，等情况下是否会扣费，一般在游戏 APP，和社交 APP 等需要考虑这些。

2）隐私泄露风险。例如在我们安装 APP 应用时通常会看到“xx 要读取手机通讯录”等提示，这些提示可以提示用户拒绝接受，这些是 APP 测试中的测试点。

3）校验 input 输入。对于 APP 有输入框的要对输入的信息进行校验，比如密码不能显示明文。在测试中红人馆注册时需要对 input 进行测试。

4）限制/允许使用手机功能接入互联网，收发信息，启动应用程序，手机拍照或者录音，读写用户数据。这个在通信行业用的比较多，比如展讯，高通等芯片厂商，他们在出厂芯片时要对手机各个功能进行测试。

三、代码安全性

之所以单独拿出来说，是因为在 SDK 测试过程中 SDK 代码被第三方工具检测出游病毒代码，这样一来就会影响输入法的使用。因此在后续测试中要尝试加入安全性测试。

四、安装与卸载安全性

1）应用程序应能正确安装到设备驱动程序上

2）能够在安装设备驱动程序上找到应用程序的相应图标。在 SDK 测试项目中发现有些设备受权限的问题，无法下发图标创建快链。

3）是否包含数字签名信息。在 SDK 测试项目中基本上没有，但是在输入法打包和主线版本上存在这样的测试。

4）安装路径应能指定

5）没有用户的允许应用程序不能预先设定自动启动

6）卸载是否安全，其安装进去的文件是否全部卸载

7）卸载用户使用过程中产生的文件是否有提示

8）其修改的配置信息是否复原

9）卸载是否影响其他软件的功能

10）卸载应该移除所有的文件

11）安装包的存放。在 SDK 下载安装包的测试中我们经常会看到下载下来的包后面有四个随机的字符串，这个的目的是为了防止第三方工具恶意删除安装包的问题。

在 SDK 测试项目中有专门针对下载安装卸载的用例，对安装的路径和下载的文件夹路径等有相关的测试，测试结果页表明，某些手机（例如华为 mate1）在删除了某个下载路径文件夹之后受权限应用不会自动创建。

五、数据安全性

1) 当将密码或其他的敏感数据输人到应用程序时，其不会被储存在设备中，同时密码也不会被解码

2) 输人的密码将不以明文形式进行显示

3) 密码，信用卡明细，或其他的敏感数据将不被储存在它们预输人的位置上

4) 不同的应用程序的个人身份证或密码长度必需至少在 4 一 8 个数字长度之间

5) 当应用程序处理信用卡明细，或其他的敏感数据时，不以明文形式将数据写到

其它单独的文件或者临时文件中。以防止应用程序异常终止而又没有删除它的临时文件，文件可能遭受人侵者的袭击，然后读取这些数据信息。

6) 当将敏感数据输人到应用程序时，其不会被储存在设备中

7) 备份应该加密，恢复数据应考虑恢复过程的异常通讯中断等，数据恢复后再使用前应该经过校验

8) 应用程序应考虑系统或者虚拟机器产生的用户提示信息或安全警告

9) 应用程序不能忽略系统或者虚拟机器产生的用户提示信息或安全警告，更不能在安全警告显示前，利用显示误导信息欺骗用户，应用程序不应该模拟进行安全警告误导用户

10) 在数据删除之前，应用程序应当通知用户或者应用程序提供一个“取消”命令的操作

11) “取消”命令操作能够按照设计要求实现其功能

12) 应用程序应当能够处理当不允许应用软件连接到个人信息管理的情况

13) 当进行读或写用户信息操作时， 应用程序将会向用户发送一个操作错误的提示信息

14) 在没有用户明确许可的前提下不损坏删除个人信息管理应用程序中的任何内容

15) 应用程序读和写数据正确。

16) 应用程序应当有异常保护。

17) 如果数据库中重要的数据正要被重写，应及时告知用户

18) 能合理地处理出现的错误

19) 意外情况下应提示用户

20) HTTP、HTTPS 覆盖测试。在测试中我们经常会遇到与请求的加密解密测试，以确保产品的安全性。

六、角色管理

角色管理也属于软件权限的一种，之所以单独拿出来是因为角色管理是 web 端测试中一个非常重要的测试点，尤其是对于一些军工产品，对角色管理的要求很严格。在

venus 的设备管理系统上就缺乏这样的角色管理的限制，比如借出的手机任何人都可以归还等细节（之前在测试设备管理系统时已经以 bug 的方式提交），如果有恶意破坏者那么数据将被整乱。

七、通讯安全

1）在运行其软件过程中，如果有来电、SMS、EMS、MMS、蓝牙、红外等通讯或充电时，是否能暂停程序，优先处理通信，并在处理完毕后能正常恢复软件，继续其原来的功能

2）当创立连接时，应用程序能够处理因为网络连接中断，进而告诉用户连接中断的情况

3）应能处理通讯延时或中断

4）应用程序将保持工作到通讯超时，进而发送给用户一个错误信息指示有连接错误

5）应能处理网络异常和及时将异常情况通报用户

6）应用程序关闭或网络连接不再使用时应及时关闭)断开

七、人机接口安全性

在实际场景中，用户的行为操作是不固定的，因此我们要尽可能多的去想一写场景。

1）异常下载。异常下载指的同时下载多个 apk。下载不同安装包程序，比如在之前的 mini 浏览器中，下载 exe 文件会导致程序崩溃；在发送 push 时发送除了 jpg 和 png 以外的图片会导致无法发送通知。

2）异常点击。在测试输入法时，我们会测试同时按下多个键看输入法是否会崩溃。

3）菜单可用。在 APP 中有好多菜单，要保证每个菜单都可用，不出现崩溃。

从吃货的角度谈组合测试

◆腾讯 TMQ：张丽颖

从吃货的角度观察组合

作为一名合格的吃货，小编我每天为了吃的健康着实费了不少心思，每周我都会根据应季蔬果来定制一周的饮食，以下是我这周的定制计划:

蔬菜类：豆角，土豆，莴笋，青椒，西红柿，圆白菜，芹菜

水果类：葡萄，西瓜，苹果，柑橘，菠萝，柚子，香蕉，李子

肉类（蛋白质类）：牛肉，猪肉，鱼，鸡肉，羊肉，豆腐

汤类：菠菜汤，西红柿汤，紫菜汤，五谷粥

在不得不考虑食物相克相生的前提下，这些定制计划中必须要进行适当的搭配才不会把小编自己吃趴下，在上面的食谱中，不能同时食用的食物有:

- 羊肉和西瓜
- 李子和鸡肉
- 鸡肉和芹菜
- 土豆和香蕉
- 豆腐和菠菜

因此在做每日菜品的搭配组合时必须要考虑这些约束，不知道看到这里的你是不是开始头大了？别急别急，小编有秘密武器可以教你简单应对~

一、是什么？

就像上面介绍的故事一样，测试过程中我们也会遇到这样的场景:

有 m 个参数、且每个参数有多个离散但有限的取值 N1、N2...Nm（其中 Ni 可以个数不等，1<=i<=m），为了覆盖参数的全部取值组合，需要 N1*N2*...*Nm 个测试用例。

譬如经过对需求的分析，得出我们需要验证 IE 在不同硬件配置的 PC 上的兼容性测试，且经过数据统计，主要用户占比的 PC 信息如下图所示：

```
PLATFORM:   x86, ia64, amd64
CPUS:       Single, Dual, Quad
RAM:        128MB, 1GB, 4GB, 64GB
HDD:        SCSI, IDE
OS:         NT4, Win2K, WinXP, Win2K3
```

并且需要验证的 IE 版本如下图所示：

```
IE:         4.0, 5.0, 5.5, 6.0
```

在这种场景下，要达到对参数的所有取值组合的覆盖，共需要 3*3*4*2*4*4=1152 条用例，若按 120 条/人日的执行力计算，这个需求的测试执行需要耗时 9.6 人日，这在敏捷迭代节奏的项目中是不太可行的，所以这种情况下我们可以考虑测试成本和错误检测能力上能达到较好平衡的组合测试方法。

像上面测试场景中提到的有多个参数、且每个参数有多个离散但有限的值，致使可能组合的参数值总数非常大，我们称之为组合爆炸。

而组合测试的目的，抽象的说就是为组合爆炸提供一种解决方案，简单地说就是在保证错误检出率的前提下采用较少的测试用例生成方法，它将被测系统或被测系统的模块抽象成一个受到多个因素影响的系统，并提取出每个因素的可能取值，结合组合测试方法，生成最终的测试用例。

根据上面的分析，我们可以了解到组合测试需要解决的最大问题就是：没有足够的测试资源来执行全部的测试用例，因此提出了基于一个数学模型和一个假设的解决方法，如下：

一个数学模型：产品的功能被抽象为函数 f，产品的输入被抽象为函数的变量 x1，x2，…，xm，且 xi（1≤i≤m）的可能取值是有限的，产品的输出被抽象为函数的返回值 y1，y2，…，yn;

一个假设：如果测试覆盖了任意 t 个（2≤t≤m）输入变量的取值组合，那么该测试可以发现函数 f 的大部分错误。

常用的组合测试方法包括：

1、两因素组合测试（也称配对测试、全对偶测试）

生成的测试集可以覆盖任意两个变量的所有取值组合。在理论上，该用例集可以暴露所有由两个变量共同作用而引发的缺陷。

2、多因素（t-way，t>2）组合测试

生成的测试集可以覆盖任意 t 个变量的所有取值组合。在理论上，该测试用例集可以发现所有 t 个因素共同作用引发的缺陷。

3、基于选择的覆盖

要满足基于选择的覆盖，第一步是选出一个基础的组合，且基础组合中包含每个参数的基础值，建议选择最常用的有效值作为基础值。基于基础组合，每次只改变一个参数值，来生成新的组合用例。

关于多因素组合测试在缺陷检出率方面的贡献，IEEE 文章提到早期的一些回顾性研究结果:

	Cumulative % of Defects		
Defect "Trigger"	Minimum	Maximum	Average
Single parameter value	30%	70%	60%
2 parameter values	70%	95%	86%
3 parameter values	88%	99%	91%

从上图可以看出，两因素组合最多能发现 95%的缺陷、平均缺陷检出率也达到了 86%，而对于三因素组合甚至更高因素的组合能发现的缺陷是非常有限的，一般情况下，超过 90%的软件缺陷，都是由 3 个或更少的参数值触发的。因此任何测试设计中都应该至少保证两因素组合的 100%的覆盖测试。有着高可靠性需求的应用，比如医疗设备或者航空电子设备，应该保证至少 3-way 因素组合的 100%的覆盖测试。

由于两因素组合测试在测试用例个数和错误检测能力上达到了较好的平衡，它是目前主流的组合测试方法。

接下来小编带你进入快捷的利用工具进行用例生成的阶段~~

二、怎么做?

在利用组合测试方法生成测试用例的过程中，小编推荐使用 PICT 工具，PICT 工具是一个从 2000 年开始在微软被使用的测试用例生成工具，它实现了 t 组合测试策略，可

以有效地按照两两测试的原理，进行测试用例设计。在使用 PICT 时，需输入与测试用例相关的所有参数，以达到全面覆盖的效果。

PICT 的使用相对简单，PICT 是一个命令行工具，接受纯文本模型文件作为输入，并输入一系列测试用例。PICT 的可选参数如下:

```
Usage: pict model [options]

Options:
 /o:N    - Order of combinations (default: 2)
 /d:C    - Separator for values  (default: ,)
 /a:C    - Separator for aliases (default: |)
 /n:C    - Negative value prefix (default: ~)
 /e:file - File with seeding rows
 /r[:N]  - Randomize generation, N - seed
 /c      - Case-sensitive model evaluation
 /s      - Show model statistics
```

更详细的说明详见 PICT 安装目录下的 PICTHelp.htm 文件

PICT 安装时会自动加入环境变量 path 中，因此你可以在任何目录下执行。

参考我们在“是什么”部分中提到的案例，接下来小编将使用 PICT 工具边介绍边实践。

PICT 接受一个输入文本模型文件，你可以使用 Windows 的记事本来保存（假设保存为 ModelFile.txt），如下图:

```
PLATFORM:  x86, ia64, amd64
CPUS:      Single, Dual, Quad
RAM:       128MB, 1GB, 4GB, 64GB
HDD:       SCSI, IDE
OS:        NT4, Win2K, WinXP, Win2K3
IE:        4.0, 5.0, 5.5, 6.0
```

其中每个参数占一行，参数和参数值之间以英文冒号分隔、参数值之间以英文逗号分隔。

通过“>”符号可以将输出重定向到 xls 文件中（假设输出文件为 OutputFile.xls），使用如下命令运行 PICT:

```
C:\YourFolder> pict ModelFile.txt > OutputFile.xls
```

生成的结果如下:

1	PLATFORM	CPUS	RAM	HDD	OS	IE
2	amd64	Single	4GB	SCSI	Win2K	4
3	amd64	Dual	128MB	IDE	Win2K3	6
4	x86	Quad	64GB	SCSI	Win2K3	5
5	ia64	Quad	128MB	IDE	Win2K	5.5
6	ia64	Dual	1GB	SCSI	Win2K	6
7	x86	Single	128MB	IDE	NT4	4
8	amd64	Single	1GB	SCSI	WinXP	5.5
9	ia64	Dual	64GB	SCSI	NT4	6
10	x86	Quad	4GB	IDE	WinXP	6
11	ia64	Quad	1GB	IDE	Win2K3	4
12	x86	Dual	64GB	IDE	NT4	5.5
13	ia64	Single	1GB	IDE	NT4	5
14	ia64	Dual	64GB	IDE	WinXP	4
15	amd64	Dual	4GB	SCSI	NT4	5
16	ia64	Single	4GB	SCSI	Win2K3	5.5
17	amd64	Quad	64GB	SCSI	Win2K	5
18	x86	Single	128MB	SCSI	WinXP	5
19	x86	Single	1GB	IDE	Win2K	6
20	x86	Quad	64GB	SCSI	NT4	6
21	amd64	Single	64GB	SCSI	NT4	4

每一行即为一条测试用例，通过此方式生成了共计 20 条测试用例，若按 120 条/人日的执行力计算，仅需 0.17 人日，相对于“是什么”部分的 9.6 人日的测试耗时，测试成本大大降低，组合测试的优势不言自明。

事实上，如果这六个参数中的某两个参数值的任意不同的组合会触发一个 bug 的话，那表格上的那组测试用例也可以发现该 bug。当三个特殊的值组合在一起触发的某个 bug，那表格上的那组测试用例不一定能发现该 bug，但是至少我们覆盖了所有的两因素组合。相对于所有组合情况来说，两因素组合的测试覆盖率要容易很多。例如，如果你想测试 10 个参数且都有 26 个值的功能，所有组合情况将生成 141，167，095，653，376 个测试用例。而两因素组合就只要测试 1094 个测试用例就可以。

三、精彩在这里！

看到这里，你是不是已经迫不及待要找一份需求来练手了呢？别急，精彩还在继续！

1、定义因素之间的约束关系

上文的例子中参数之间是互相独立的，但大多数被测试应用的因素之间存在约束关

系。如果不考虑约束关系，组合测试用例集将包含大量的无效测试用例。这些无效的测试用例，包含一些无效的取值组合，也有可能包含一些有效的取值组合。仅仅删除无效测试用例，会导致最终的测试用例集不能实现两因素或多因素组合覆盖。面对因素之间存在约束关系的被测试应用，应该明确定义约束关系，让组合测试工具根据约束来生成有效的测试用例集。

因此我们需要在设计 PICT 输入模型时，加入约束限制，好在 PICT 工具很强大并在设计之初就考虑到了这点，以本文最开始的故事为例，除了多参数、多值以外，还存在着参数值之间的约束关系，修改我们在上文中的输入文本文件如下:

```
蔬菜类: 豆角, 土豆, 莴笋, 青椒, 西红柿, 圆白菜, 芹菜
水果类: 葡萄, 西瓜, 苹果, 柑橘, 菠萝, 柚子, 香蕉, 李子
肉类:   牛肉, 猪肉, 鱼, 鸡肉, 羊肉, 豆腐
汤类:   菠菜汤, 西红柿汤, 紫菜汤, 五谷粥

IF [肉类] = "羊肉" THEN [水果类] <> "西瓜";
IF [肉类] = "鸡肉" THEN [水果类] <> "李子";
IF [肉类] = "鸡肉" THEN [蔬菜类] <> "芹菜";
IF [蔬菜类] = "土豆" THEN [水果类] <> "香蕉";
IF [肉类] = "豆腐" THEN [汤类] <> "菠菜汤";
```

即如果肉类是羊肉，则水果类不能选择西瓜，其他以此类推。当 PICT 读取模型文件时，它会解析约束规则，并将其应用于测试用例生成过程。生成的测试用例集既满足对有效取值组合的覆盖，又不包含无效取值组合。

执行 PICT 命令行，生成的食谱合理搭配如下:

蔬菜类	水果类	肉类	汤类
青椒	柚子	羊肉	西红柿汤
豆角	菠萝	猪肉	紫菜汤
莴笋	香蕉	羊肉	五谷粥
西红柿	李子	牛肉	五谷粥
莴笋	菠萝	牛肉	西红柿汤
西红柿	西瓜	鸡肉	菠菜汤
西红柿	柚子	鱼	紫菜汤
…	…	…	…

共计 58 种组合选择，完全满足一周的营养健康搭配需要了（yeah!!!）

2、引入随机种子

看到这里，资深吃货一定会问：上面的 58 种组合最多只能吃一阵子，然后就不得不重复食谱了-_-|||，针对这个问题，PICT 也提供了随机种子来应对。PICT 采用的是伪

随机算法，输入不变的情况下，输出的测试用例是相同的，因此引入随机种子，在测试成本相同的情况下，改变组合次序，测试执行可以执行更多的路径，覆盖更广大的状态空间，发现隐藏缺陷的概率也会提高。

在 PICT 中，参数"/r[:N]"可以为测试用例生成引入随机种子（N 是作为随机种子的整数），以生成不同的测试用例。譬如我们分别尝试不带种子、和带种子 100 的食谱搭配结果:

```
C:\Users\Administrator\Desktop>pict ModelFile.txt > OutputFile.xls

C:\Users\Administrator\Desktop>pict ModelFile.txt /r:100 > OutputFile.xls
Used seed: 100
```

结果如下:

蔬菜类	水果类	肉类	汤类
豆角	菠萝	猪肉	紫菜汤
豆角	苹果	鸡肉	西红柿汤
豆角	西瓜	鱼	五谷粥
豆角	香蕉	羊肉	紫菜汤
豆角	葡萄	牛肉	菠菜汤
豆角	柑橘	鸡肉	菠菜汤
豆角	柚子	牛肉	紫菜汤
豆角	李子	豆腐	西红柿汤
…	…	…	…

不带种子值

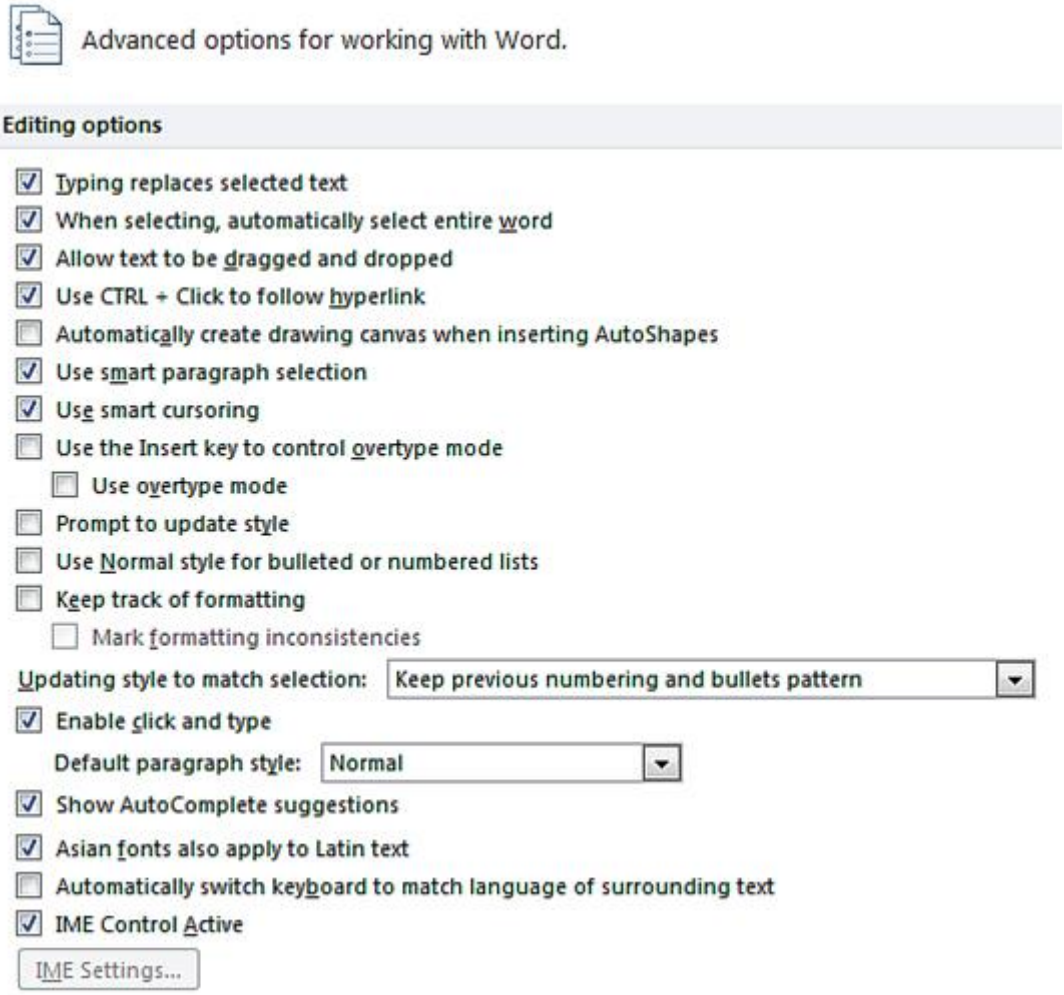
蔬菜类	水果类	肉类	汤类
豆角	柚子	鱼	五谷粥
豆角	香蕉	猪肉	紫菜汤
豆角	李子	羊肉	西红柿汤
豆角	苹果	鱼	菠菜汤
豆角	葡萄	豆腐	五谷粥
豆角	西瓜	鸡肉	西红柿汤
豆角	菠萝	牛肉	西红柿汤
豆角	柑橘	羊肉	菠菜汤
…	…	…	…

种子值为100

从上图可以看出，在引入随机种子后，组合次序发生了变化，但是组合个数一样。这样大家可以在不提高测试成本的前提下，提高覆盖率。

3、覆盖最最重要的组合，即基于选择的覆盖

追求完美的吃货可能还会发现，上面那么多食谱组合，竟然没有组合到他最爱的搭配，例如：豆角、香蕉、猪肉、五谷粥，即组合测试可能会错过最重要的取值组合。



上图是 Word 2010“高级”设置的一部分，每个复选框都是一个因素，每个因素都有“勾选、未勾选”两个选择，为了测试 Word 在不同设置下的行为，对所有因素生成组合测试用例集。但是该测试用例集很可能没有覆盖 Word 的默认设置。事实上，大多数用户几乎不修改默认配置，测试用例集没有覆盖最常用、也是最重要的取值组合，所以建议使用“基于选择的覆盖”方法。这揭示了组合测试的一个潜在风险：如果测试人员不仔细分析被测试对象，只依赖组合测试工具，他可能错过用户最常见的测试用例。

4、警惕卫哨语句

为了设计更合理的测试用例，大家不光要从需求分析输入、更要从代码层次分析输入，因为许多软件会利用卫哨语句来“过滤”无效的输入。例如，在如下代码中，if 语句会“过滤”掉所有 A<=0 的输入。

```
int func(int A, int B, int C)
{
    if (A <= 0) return ERROR;

    ...
}
```

如果不了解代码中的卫哨语句，可能会将输入定义为:

```
A: -1, 0, 1
B: -1, 0, 1
C: -1, 0, 1
```

生成的输出中

A	B	C
0	1	-1
1	-1	1
-1	0	-1
1	1	0
-1	-1	0
0	0	1
1	-1	-1
-1	1	1
0	-1	0
1	0	0

在这 10 条测试用例中，因为 A<=0，有 6 条测试用例会被 if 语句过滤掉。所以如果忽视了卫哨语句对执行流的中断，组合测试用例集将不能达成两因素或多因素覆盖的目标。

面对此类问题，测试人员要仔细阅读规格说明或源代码，发现会导致执行流中断的“负面”取值。在 PICT 的模型中，支持用特殊符号"~"标记出非法值，例如在上述模型中将 A 的取值 0 标记为非法值。

```
A: ~0, 1, 10
B: -1, 0, 1
C: -1, 0, 1
```

PICT 会保证所有有效值的取值组合都会被覆盖，此外任意非法值与有效值的组合也会被覆盖。以上模型将生成如下测试用例集。

A	B	C
1	1	-1
1	0	1
10	-1	-1
1	-1	0
10	0	-1
10	-1	1
10	0	0
10	1	0
1	1	1
~0	0	-1
~0	-1	0
~0	1	1

5、多因素组合

如果你觉得两因素组合不能满足需求，可以利用多因素组合和子模型来定义不同的强度组合。

PICT 通过参数/o:N 支持多因素组合，譬如上图中的案例，未定义/o 参数则默认采用的是两两组合，一共生成了 12 条用例；改成三因素组合的话：

```
C:\Users\Administrator\Desktop>pict input.txt /o:3 > res.xls
```

会生成 27 条测试用例：

A	B	C
1	0	-1
10	-1	-1
1	0	0
1	1	0
10	1	0
10	0	0
10	0	-1
1	-1	-1
1	1	-1
1	-1	1
10	0	1
10	-1	0
10	-1	1
1	-1	0
1	0	1
10	1	-1
10	1	1
1	1	1
~0	1	1
~0	1	0
~0	-1	-1
~0	1	-1
~0	-1	0
~0	0	-1
~0	0	0
~0	-1	1
~0	0	1

在提高覆盖率的同时，测试成本同时也增加了 100%以上，因此大家在考虑提高覆盖率的同时必须将测试成本加入考量。

或者大家若可以确定在 M 个参数中有 N 个参数的组合是非常重要的，那么这种情况下，小编推荐使用子模型，着重对 N 参数的组合加大覆盖率:

如下输入中有 ABCDE 共 5 个参数，每个参数的值都是 0，1，小编定义了”(B，C，D) @ 3”子模型并设置为 3 因素覆盖，

```
A: 0, 1
B: 0, 1
C: 0, 1
D: 0, 1
E: 0, 1

(B, C, D) @ 3
```

生成的组合展示如下:

AB	AC	AD	AE	BCD	BE	CE	DE
00	00	00	00	000	00	00	00
01	01	01	01	001	01	01	01
10	10	10	10	010	10	10	10
11	11	11	11	011	11	11	11
				100			
				101			
				110			
				111			

该模型除了所有因素的两两覆盖外，还着重对 BCD 三个因素做更高阶的覆盖，而且测试成本也相对没有提高太高，具有可执行性。

四、写在最后

工具的使用说了这么说，其实重点还在于大家对需求的透彻理解和分析，总结起来就是:

1、一定要注意分析元素间的约束条件，避免生成多条无效测试用例，还增加了测试成本。

2、确定建模的输入时，需要重点分析各个参数的值，尤其是一些有代表性的值，着重分析各类参数值对结果的影响。

3、PICT 还有更多的属性可以应用，请参考 PICT 安装目录下的 PICTHelp.htm 文件。

客户端与服务器交互的功能，如何进行测试？

◆ 搜狗测试：王爱艳

测试客户端与服务器交互的功能，如何进行测试，需要考虑哪些内容呢？下面我们分阶段来说明一下~

一、测试沟通阶段

- 需要跟客户端和服务器端开发沟通，确定客户端发送请求的样式，需要包含哪些参数值，参数值具体有什么样的作用。
- 跟服务器端确认是否需要添加特定的 user-agent（添加 user-agent 的目的：确保服务器安全）。
- 确认客户端和服务器端交互时是否需要对文件进行加密操作。
- 跟开发确认请求时是否需要增加重试和具体的超时机制，有无下载的断点续传。
- 确认服务器的具体类型，是 apache 的还是 Ngnix 的。
- 对于需要客户端识别的参数，确认服务器端返回该参数是可能存在的返回值。
- 对于异常情况，跟服务器和客户端的开发确定相应的容错处理。
- 需要注意询问开发，与服务器之间的交互是用什么做的，标准的 http 协议还是自写的协议。
- 还要注意，交互时的 http 连接，是用 get 还是 post。
- 需要和开发事先沟通清楚，是否需要特殊工具。

- 如果客户端与服务器端交互的配置文件为密文时，需要询问开发有没有需要特别关注测试的地方。
- 需要和开发对一下 url 中的参数，是否可以满足需求
- 需求了解时，要注意跟产品/开发确认好测试范围;
- 跟开发确认好，是否需要进行压力测试;
- 跟开发确认都哪些地方需要加防盗链;
- 如果涉及到网页相关，要确认好测试环境：xp、win7、Win8、Win10 及 IE 版本等
- 跟开发确认好待测试的常规网络错误，以及测试方法;
- 对于服务器端策略，需要和该服务的运营负责人沟通清楚，是否需要测试关注

功能划分

如果一个功能同时涉及服务端和客户端的修改，首先从功能上就要分别对客户端和服务端分别进行测试。

二、测试准备阶段

- 尽量使用线上的服务器；如果需要搭建服务器则尽量保持跟线上或未来线上的服务器类型一致。
- 搭建测试服务器环境时，测试服务器的返回策略，尽量跟线上或未来线上的服务器端策略一致。

用例设计

- 写 case 的时候，详细了解客户端和服务端的逻辑后，要确认一下，之前定好的测试范围是否合理，是否有部分不需测试的逻辑可能会存在问题，如果有这种不确认的地方，需要再次确认测试范围;
- 要考虑不联网时，待测模块是否仍需支持某些功能;

三、执行阶段

3.1、客户端关注

- 尝试连接时，不联网，要有超时
- 对于本地无连接的测试，一定要区分断网和禁用网卡，这两种情况不同。
- 下载过程中，网络情况不佳或者断网，要有超时，最好也有三次重连机制，重连的时间不宜过短建议 20ms
- 文件过大，建议需要有断点续传逻辑
- 要验证各种网络错误环境，最起码要包括 200、302、403、404、417、500、502 等错误和服务器超时、本地超时
- 测试时需要关注，每条 url 请求是否支持 302 跳转
- 需要模仿 502 跳转，确保调整后，客户端能够正常运行
- 服务器返回文件类型需要关注：文件格式错误、Html 格式文件、空文件、0 字节的文件。
- 服务器返回文件时，文件的储存路径，空间，路径的读写权限、储存路径是否已经存在文件，存在文件的数据（0 字节，其他类型的文件、损坏的文件、下载的文件不完整）
- 服务器端返回的 url 值的类型、长度的容错，服务器返回文件的内容：是否加密，参数值为中英文、简繁体、特殊符号、数字、空、缺省、零、小数、负值、超长、乱码等，参数缺省，规定下载的文件个数与下载的文件实际个数不匹配。
- 发送的 url 内容，参数值中的特殊符号、中文是否已经转义;
- 需要测试时关注，交互时配置文件的编码问题，例如要覆盖到 ASCII、unicode、ANSI 等编码。
- 还要注意测试时，交互时配置文件中的换行与回车换行的问题，要保证这两种都可以测试通过
- 要注意在低权限进程中，该功能能够正常进行
- 该交互是否会被安全软件拦截；如果要打开浏览器访问，还要检查是否会被浏览器拦截

- 客户端发送的请求中是否带具有特性的 User-Agent（具体根据与开发的沟通结果来验证）
- 基本功能测试完毕后，需要进行跟服务器端的联调。
- 客户端与服务器联调时，要注意确认联调 case，多跟开发沟通;

3.2、服务器端关注

- 对于服务器来说，要进行压力测试
- 客户端、服务器端分别测试后，上线前要有联调，除了走主功能外、还要结合开发与运营的意见设计联调 case
- 服务端需要考虑是否要做安全校验，以免被攻击
- 服务端和客户端均通过测试后，在上线前，需要客户端和服务端进行联调测试，确认服务端和客户端均 ok
- 客户端发送请求的内容：是否加密，参数值为中英文、简繁体、特殊符号、数字、空、缺省、超长。

四、测试完成后

- 服务端上线后，如果有需要，需要验证服务端的服务正常上线，通过外网 IP 能够获取服务端的服务
- 公示客户端与服务器端交互时需要注意的相关事项及存在的风险，确保服务器端的策略能够与客户端正确匹配。

压力测试遭遇大量 TIME_WAIT 之后

◆腾讯 TMQ：崔 杰

HTTP 协议是互联网中最常使用的应用层协议，它的绝大多数实现是基于 TCP 协议的。

一、问题描述:

某天，在对一个提供 http 接口的后台服务进行压力测试过程中，我们设定了几百 qps（每秒请求数）开始测试几分钟后，请求一端（我们后续简称为：客户端）的压力结果统计日志中开始连续出现大量的报错信息:

```
Get http://[redacted]/?idx=107&lv=1&bn=2716: dial tcp 10.170.9.30:10008: cannot assign requested address
Get http://[redacted]/?idx=99&lv=1&bn=854: dial tcp 10.170.9.30:10008: cannot assign requested address
Get http://[redacted]/?idx=98&lv=0&bn=69930: dial tcp 10.170.9.30:10008: cannot assign requested address
Get http://[redacted]/?idx=107&lv=1&bn=729: dial tcp 10.170.9.30:10008: cannot assign requested address
Get http://[redacted]/?idx=107&lv=2&bn=113: dial tcp 10.170.9.30:10008: cannot assign requested address
Get http://[redacted]/?idx=98&lv=2&bn=6: dial tcp 10.170.9.30:10008: cannot assign requested address
Get http://[redacted]/?idx=98&lv=1&bn=9541: dial tcp 10.170.9.30:10008: cannot assign requested address
Get http://[redacted]/?idx=98&lv=0&bn=435324: dial tcp 10.170.9.30:10008: cannot assign requested address
Get http://[redacted]/?idx=95&lv=1&bn=299: dial tcp 10.170.9.30:10008: cannot assign requested address
Get http://[redacted]/?idx=105&lv=0&bn=2450: dial tcp 10.170.9.30:10008: cannot assign requested address
Get http://[redacted]/?idx=96&lv=0&bn=812: dial tcp 10.170.9.30:10008: cannot assign requested address
Get http://[redacted]/?idx=107&lv=2&bn=116: dial tcp 10.170.9.30:10008: cannot assign requested address
```

在压力测试前，根据之前的经验，同类服务的单机性能一般能够达到几千 QPS，然而此时测试设定的压力值还不足 200qps，这与预期存在 1 个数量级以上的性能差距，难道是被测服务存在问题么？

二、问题跟踪:

为了确认被测服务的状态，我们首先登录了服务所在的机器，检查了服务资源的占用情况，结果是：CPU、内存、硬盘、I/O、网卡、fd、socket 等各项资源都不存在较大负载。看来服务本身还远没有达到它的负载瓶颈。

在排除服务端问题后，我们重新分析了统计日志中的错误--"can not assign requested address"，这是一个常见的 socket 的 error，报错信息说明无法为 socket 创建新的连接，很可能是：tcp 层的连接端口已经耗尽，无法为新的 http 请求分配端口建立连接。通过 netstat 命令，我们检查客户端，发现确实存在大量请求连接处于 TIME_WAIT 状态下:

```
root@10.169.139.58:/home/qqmap/cj/go/scheduler/bin# netstat -alp|grep 'TIME_WAIT'|grep '[redacted]'|wc -l
27890
```

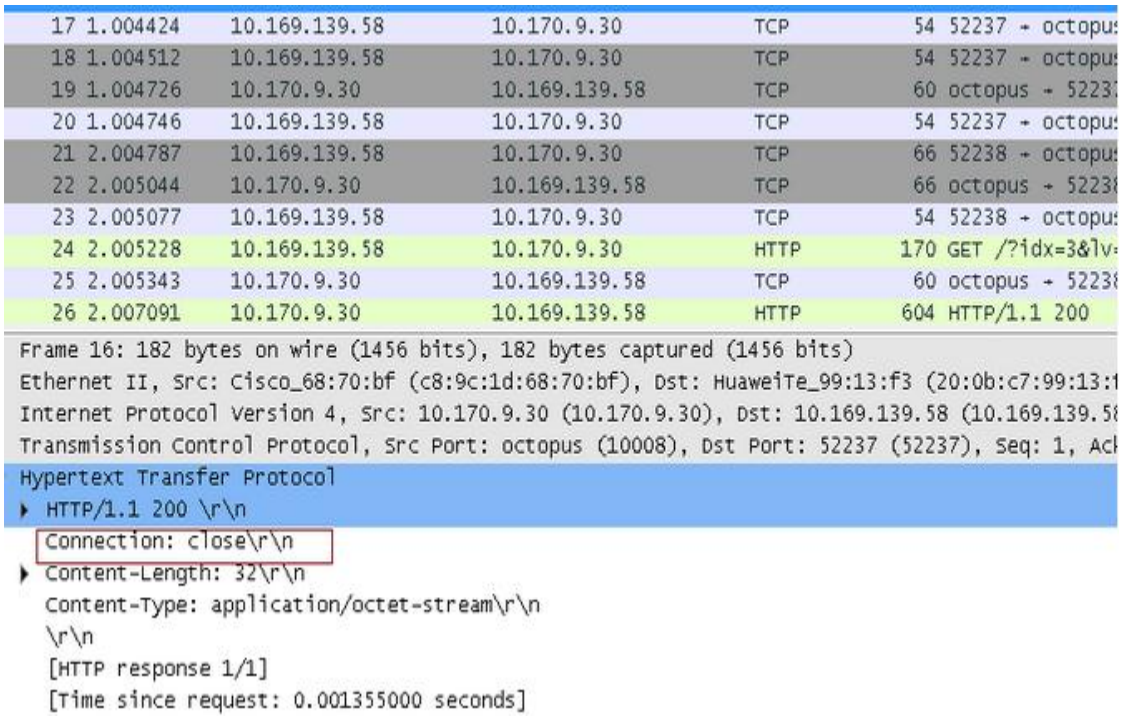
这里要说明一下，虽然理论上 tcp 连接可用端口号为 0~65535--大约 65536 个，但是实际在不指定端口情况下连接服务时可用端口默认为 32768~61000--大约只有 28000 多个，在 linux 系统中这个限制可以通过/proc/sys/net/ipv4/ip_local_port_range 文件进行修改。

我们知道 http 协议主要是基于 tcp 协议之上的，为了解决 tcp 层连接通道复用的问题，在 http 协议中通过 header 中的 Connection 字段定义了对于 tcp 长连接的支持:

（1）在 HTTP/1.0 版本中，默认情况下在 HTTP1.0 中所有连接不被保持，如果客户端浏览器支持 Keep-Alive，那么就在 HTTP 请求头中添加一个字段 Connection: Keep-Alive，当服务器收到附带有 Connection: Keep-Alive 的请求时，它也会在响应头中添加一个同样的字段来使用 Keep-Alive。这样一来，客户端和服务器之间的 HTTP 连接就会被保持，当客户端发送另外一个请求时，就使用这条已经建立的连接通道。

（2）在 HTTP/1.1 版本中，默认情况下在 HTTP1.1 中所有连接都会被保持，除非在请求头或响应头中指明要关闭：Connection: Close，这也就是为什么 Connection: Keep-Alive 字段再没有意义的原因。

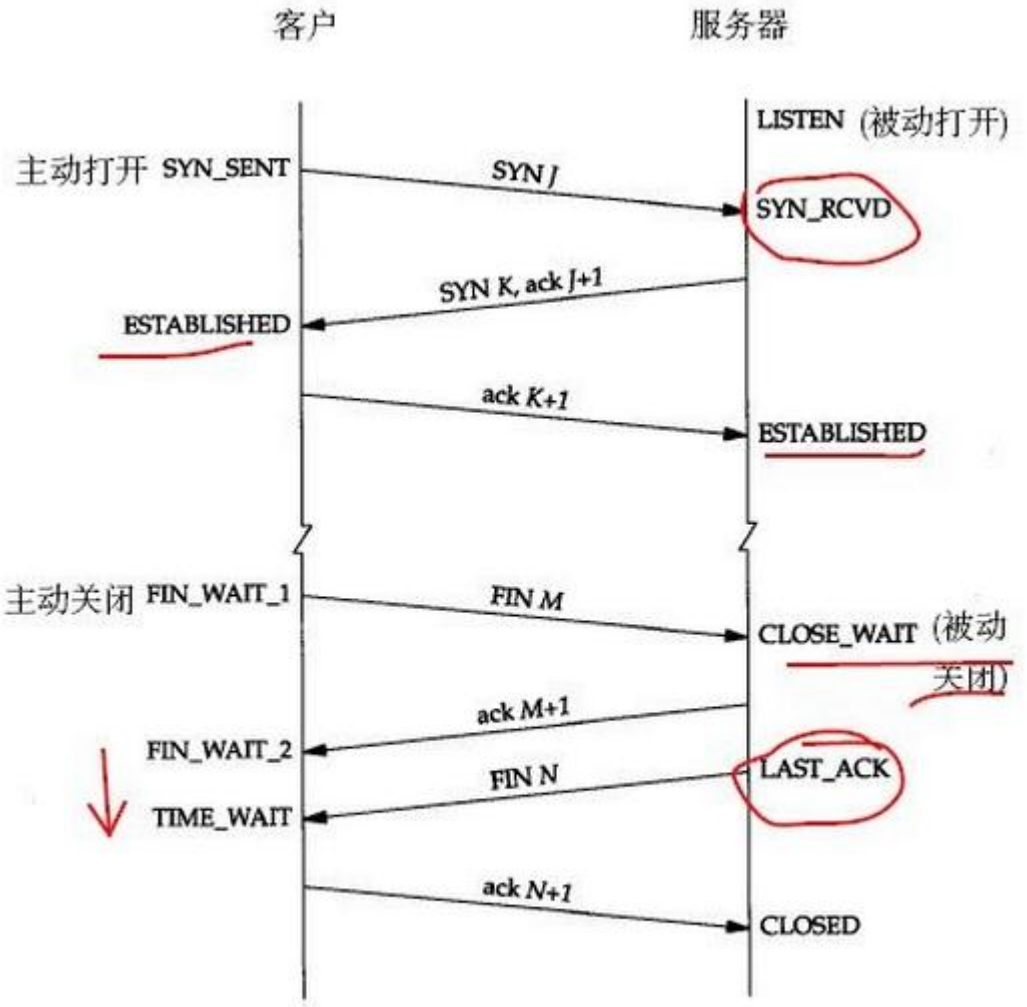
在压力测试过程中，我们模拟发送 http 请求的代码中使用的是 http/1.1 协议，应该会默认使用长连接，看来很可能是服务端不支持长连接，才会引起客户端频繁的创建 TCP 连接。通过 tcpdump 抓包，我们对此进行了证实:



三、跟进分析:

到这，我们发现服务端确实返回不支持长连接的信息（header 中 connection:close），导致客户端每次发起请求都会重新创建 tcp 通道。但是根据以往测试经验来看，比较常见的是在服务端出现大量 time_wait 状态的，**那么为什么大量的 time_wait 状态会在客户端出现呢?**

了解这个问题我们之前，可以先来看一下 TCP 正常连接建立和关闭连接时的状态变化图:



TCP正常连接建立和终止所对应的状态

上图是 TCP"三次握手"和"四次挥手"的过程，相信读者都会比较了解，下面我们来说说为什么要存在 TIME_WAIT 状态:

- **可靠地实现 TCP 全双工连接的终止**

TCP 协议在关闭连接的四次挥手中，在主动关闭方发送的最后一个 ack(fin) ，有可

能丢失，这时被动方会重新发 fin，如果这时主动方处于 CLOSED 状态，就会响应 rst 而不是 ack。所以主动方要处于 TIME_WAIT 状态，而不能是 CLOSED 。

- **允许老的报文段在网络中消失**

TCP 报文段可能由于路由器异常而“迷路”，在迷途期间，TCP 发送端可能因确认超时而重发这个报文，迷途的报文在路由器修复后也会被送到最终目的地，这个原来的迷途报文就称为 lost duplicate。在关闭一个 TCP 连接后，马上又重新建立起一个相同的 IP 地址和端口之间的 TCP 连接，后一个连接被称为前一个连接的化身（incarnation），那么有可能出现这种情况，前一个连接的迷途重复报文在前一个连接终止后出现，从而被误解成从属于新的化身。为了避免这个情 况，TCP 不允许处于 TIME_WAIT 状态的连接启动一个新的化身，因为 TIME_WAIT 状态持续 2MSL，就可以保证当成功建立一个 TCP 连接的时候，来自连接先前化身的重复报文已经在网络中消逝。

明白了 time_wait 的存在原因和出现时机，可以看到 TIME_WAIT 状态总是出现的主动关闭连接的一方，也就是说在我们压力测试过程中每次都是客户端主动关闭 tcp 连接的。从实际的抓包结果来看，确实如此：

10.169.139.58	10.170.9.30	TCP	66 52165 → octopus [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PER
10.170.9.30	10.169.139.58	TCP	66 octopus → 52165 [SYN, ACK] Seq=0 Ack=1 Win=14600 Len=0 MSS=14
10.169.139.58	10.170.9.30	TCP	54 52165 → octopus [ACK] Seq=1 Ack=1 Win=14720 Len=0
10.169.139.58	10.170.9.30	HTTP	170 GET /?idx=2&lv=2&bn=84 HTTP/1.1
10.170.9.30	10.169.139.58	TCP	60 octopus → 52165 [ACK] Seq=1 Ack=117 Win=14720 Len=0
10.170.9.30	10.169.139.58	HTTP	1318 HTTP/1.1 200 (application/octet-stream)
10.169.139.58	10.170.9.30	TCP	54 52165 → octopus [ACK] Seq=117 Ack=1265 Win=17152 Len=0
10.169.139.58	10.170.9.30	TCP	54 52165 → octopus [FIN, ACK] Seq=117 Ack=1265 Win=17152 Len=0
10.170.9.30	10.169.139.58	TCP	60 octopus → 52165 [FIN, ACK] Seq=1265 Ack=118 Win=14720 Len=0
10.169.139.58	10.170.9.30	TCP	54 52165 → octopus [ACK] Seq=118 Ack=1266 Win=17152 Len=0

客户端优先发起FIN请求

但是我们实际遇到的多是 time_wait 出现在服务一端出现的，那么在 http 协议规定中，服务端返回 connection:close 的信息后，到底是应该由客户端还是服务端来主动关闭连接呢？

Connection: close 是一个 general-header（ RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1 ）即：既可以作为 request header 也可以作为 response header。Connection: close 的作用在于"协商(signal)"。在 RFC2616 14.10 中：

HTTP/1.1 defines the "close" connection option for the sender to signal that the connection will be closed after completion of the response.

通过 RFC 可以发现：请求和响应的双方都可以主动关闭 TCP 连接。

但是大多数的 web Service 实现是返回 connection:close 内容之后服务端会主动关闭连接。至于这样设计的原因，网上找到 2 个比较靠谱的解释：

（1）server 主动关闭连接是历史原因：HTTP/0.9 协议中 response 是没有 header 的，所以 client 根本无从知道什么时候这个东西结束。

（2）在 server 主动关闭连接的情况下，只要调用一次 close() 就可以释放连接，剩下的工作由内核 TCP 栈直接进行了处理，整个过程只有一次 syscall；如果是要求 client 关闭，则 server 在写完最后一个 response 之后需要把这个 socket 放入 readable 队列，调用 select / epoll 去等待事件；然后调用一次 read() 才能知道连接已经被关闭，这其中是两次 syscall，多一次用户态程序被激活执行，而且 socket 保持时间也会更长；

也许会有读者担心：如果客户端也不主动关闭 TCP 连接，服务端的 socket 资源会不会很快用完呢。这里留给读者们一个问题进行思考：在某个服务的服务端理论上能支持的最大 TCP 连接数是多少呢？

四、解决方法：

根据分析，我们知道了客户端请求报错的原因在于：服务端拒绝了客户端的长连接请求，同时服务端没有主动关闭 tcp 连接，而是由客户端主动关闭网络连接，导致在客户端出现大量 time_wait，在压测进行到一段时候后由于没有新的 socket 端口可用而开始报错。

了解了原因后，解决方法就比较简单了，需要我们修改客户端所在 linux 环境下的 tcp 相关参数，编辑/etc/sysctl.conf 文件，增加三行：

```
net.ipv4.tcp_syncookies = 1  #开启SYN Cookies,当出现SYN等待队列溢出时，启用cookies
处理
                             #默认为0，表示关闭
net.ipv4.tcp_tw_reuse = 1    #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_recycle = 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
```

再执行以下命令，让修改结果立即生效即可:

```
/sbin/sysctl -p       #从配置文件“/etc/sysctl.conf”加载内核参数设置
```

然后，我们的压力测试的客户端不会再受 time_wait 的困扰了。

如何提升工作效率？

◆ 搜狗测试：赵素丽

> 请你告诉我，我该走哪条路？"爱丽丝说。
>
> "那要看你想去哪里？"猫说。
>
> "去哪儿无所谓。"爱丽丝说。
>
> "那么走哪条路也就无所谓了。"猫说。

引入一个经典的例子，是想告诉大家也是小编自勉，工作首先需要有目标，如果自己都不知道干什么，别人是无法帮助你的。在工作中，如果没有目标，往往是被动的完成一些工作上的事情，可能存在的问题是，工作 2~3 年，依旧是原地踏步。有了目标也就有了工作道路上的灯塔，知道了该选择哪条路了，即使道路崎岖和遇阻，也不会轻言放弃，因为在灯塔的指引下，有了向前冲的激情和动力。

如何制定有效的目标呢？引入目标的 SMART 原则，结合大熊、小明的故事给大家讲解。

S--Specific，明确的，制定的目标需要是明确。

大熊：“小明，你近期在忙什么？看你每天都挺忙的”

小明：“我最近在做服务端相关的测试，调研服务端测试相关的知识。”

大熊：“当前有什么成果吗？”

小明：“还没有，服务端测试的类型很多，我最近了解了压力测试相关的知识、服务端缓存机制还有接口测试，每一类都了解到表面，感觉每天都很忙，但是没有什么成就感。”

大熊：“你想解决什么问题？服务端测试的类型很多，你当前优先解决的问题是什么？”

小明：“想提升我们的测试专业度，优先解决的问题…”，小明陷入沉思…

大熊：“建议你回去翻下你的问题列表，思考清晰你当前优先解决的问题是什么？”

小明回去翻开了问题列表，仔细合计了一晚上，第二天找到大熊。

小明：“我查阅了我的问题列表，做了分析和调研，当前开发提测了很多服务端接口的测试，手动测试效率较低，遭到了开发的投诉。我最优先解决的是将接口测试自动化，提升我们的专业度”

大熊：“这样的思路是很好的，服务端测试的范畴很广，你不可能把所有的范畴在第一时间内解决，你需要找到你问题的根源，具体化你的目标。”

旁白：在我们工作过程中，需要时刻维护一份问题列表，时刻的归纳翻阅问题列表，而我们目标的制定，需要从问题列表中抽取问题，明确核心问题，思考问题的策略和方案，而目标是来源于问题的，这样的目标才是明确具体的。

M--Measurable，可衡量的，制定的目标是能够衡量的。

大熊：“你前阵子代码调研的结果怎么样了？”

小明：“嗯，已经完成。”

大熊：“调研完的产出是什么呢？”

小明想了想回答：“也没什么，对我们的代码熟悉了，方便今后测试范围的确认。”

大熊：“有补充/删减用例吗？调研完后，我们有挖掘到好的测试方法和手段吗？”

小明：“这个也需要吗？我还没有考虑这方面。”

大熊：“你再重新补充一下吧。”

几天过去了，小明再次找到大熊，汇报了代码调研的产出，及量化了增/删了多少的case，通过代码调研的分析，优化了测试方法，增加了测试的手段。

大熊点头称赞，做的不错！

旁白：目标的制定，需要量化，可衡量。如果制定的目标没有量化，我们就不知道什么情况下是一个可以完成的状态，那么我们的目标就没有落地，执行力就会下降，我们或许永远不能完成目标。可能在代码调研后，就认为已经完成了，但是对于后续更大

的价值，我们却忽略了。

A--Achievable，可实现的，制定的目标是可达成的。

季度初，小明找到大熊定目标。

小明：“因当前项目中，二轮测试过程中，时间较长，我的目标是计划实现移动端的自动化测试，让所有二轮的模块均实现自动化。”

大熊：“这个进行过调研吗？是否可行？”

小明：“我感觉这个应该可行，PC 端可已经在用自动化了，我们也可以。”

大熊：“你计划用什么框架呢？所有二轮的模块你这边都调研了吗？”

小明：“暂时没有调研，要不我调研下，再给出结论。”

小明又仔细思考了一晚上，第二天找到大熊。

小明：“这个我已经调研了，具体细节发给你邮件了，总体的结论是，当前的一些自动化框架，不适合咱们项目组，且不同的模块的处理方式不一样，所以暂时不能实现。”

大熊：“哎呦，不错哦。”

旁白：在我们制定一份目标或方案时，首先需要考虑是否可行，必要时，需要给出可行性方案。如果一项任务没有经过评估就开始投入，很大的概率会失败，且前期已经投入了很多，个人的自信心、执行力都会下降。如果频繁发生个人目标不能完成的情况，个人的口碑也会出现不好的评价”。

R--Relevance，相适的，制定的目标是必要的，能够为自己带来有意义的收获。

季度初，小明找到大熊定目标。小明：“我了解到咱们团队中，团队创新想法比较少，我计划开发出一套平台，记录大家的创新想法，提高大家的创新能力。”

大熊：“通过这套平台，就可以提高大家的创新能力吗？”

小明：“应该是可以的，大家可以在上边备忘各自的新想法。”

大熊：“通过开发一套平台就可以解决吗？创新想法少的原因是什么？”

小明想了一会回答：“开发一套平台，解决的是创新想法及时备忘的问题，不能解决创新想法少的问题。创新想法少的原因是，可能大家不知道哪些属于新想法的范畴，

可能大家没有意识到新想法的意义，这个具体的原因还回去再调研。”

小明听到大熊的话后，回去进行了收集、分析，再次找到大熊。

小明说，“这个问题的根本原因，经过调研是大家没有意识到新想法产生的意义，我的目标是，整理出新想法意义的报告，同步给大家，引导大家理解新想法的意义。”

大熊露出了满意的微笑连生夸赞：“很不错”

旁白：我们在制定目标时，需要明确，目标的实现是否真能为自己带来有意义的收获。所以我们在制定目标时，收集到问题后，需要多思考、多收集资料、多做调查，根据收到的素材和以往的经验，分析问题的最根本原因，找到问题的解决方案，这样制定的目标才是相适的。”

T--Time，时间，制定的目标是有完成的期限。

季度末的某一天，小明沮丧的找到大熊谈话。

小明：“这个 Q，我的稳定性评测工具估计不能完成了，提前跟你打个招呼。”

大熊：“我们做过可行性评估的，是 ok 的，是遇到了什么问题吗？”

小明：“实现是没有问题的，发现越做东西越多，估计要 delay 了。”

大熊：“稳定性评测工具你有具体的计划方案吗？当初给出完成时间点了吗？”

小明摇摇头：“稳定性评测工具实现起来不难，想着这个 Q 完成就好，没有规定具体的完成时间点。”

大熊：“你回去后，将你的目标细化，按照细化的目标进行排期，给出最终完成时间点。”

小明回去以后，把目标进行拆分，细化到每一项事情及完成时间点，整理成表格，信心满满的找到大熊，”我已经规划好这项目标的具体计划，以及具体完成时间点，此目标目前看来问题不大。”

大熊：“好的，就像蛋糕一样，一下子是没办法全部消化的，你可以进行切分，逐步完成。你看现在做的很好嘛。”

旁白：优秀的目标制定者通常都会将自己的目标进行拆解、分析，每项目标设定一个完成的时间点，按照计划，逐步完成自己所设定的目标。同时，给自己提出了时间上

的要求，不但保证了执行的效率，也给了自己适当的压力，鞭策自己在规定的时间内完成，兑现自己的承诺。”

引用一句话:”一心向着自己目标前进的人，整个世界都为他让路”。小编通过一些实例，为大家介绍了 SMART（聪明）目标原则的使用方法。在大家的目标制定中，可参考此原则制定自己的目标，如有问题，请回复。后续小编为持续为大家介绍如何提升工作效率的文章，请关注。

从 Android 测试体系之内存篇

◆搜狗测试：王政达

一、引言

相信很多小伙伴在做 Android App 测试的时候都遇到过这些情况：App Crash 了、APP 由于内存暴涨导致 OOM 了、APP 运行卡顿不流畅了、APP 流量电量耗费过快了等。我们如何在测试阶段及时的发现并定位问题的原因呢？正所谓工欲善其事必先利其器，要解决这些问题，我们必须要利其器，而且要全方位的利：功能测试、性能测试、自动化测试、兼容性测试、安全测试、覆盖率测试，一个都不能少。今天我们就先从性能测试的内存谈起，以后也会逐步完善这一系列的其他部分。

二、基本概念

在开始我们的内存测试之前，先不要着急，让我们从最开始的地方出发，一步一步的弄清楚我们要测什么、怎么测、以及如何分析。首先了解一些相关的基本概念吧。既然我们是在 Android 系统上做内存测试，那么就先从 Android 系统的分层架构谈起吧。

1. Android 分层架构

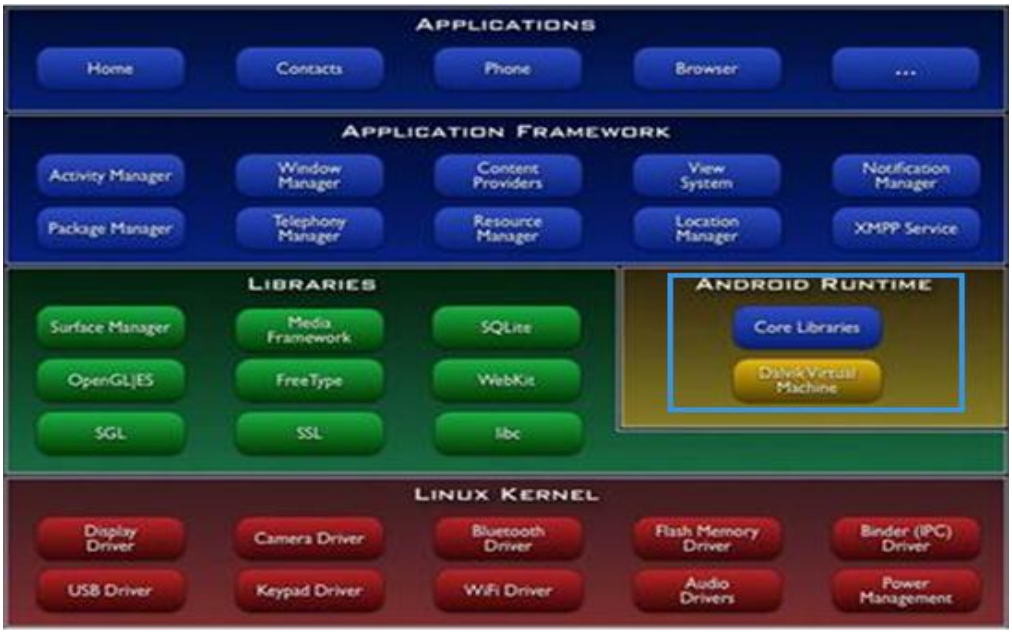
Android 的系统从上层到底层共包括四层，分别是应用程序层、应用框架层、系统库和 Android 运行时和 Linux 内核层。

1) 应用层主要是提供核心应用程序包，如发邮件，日历等，由 java 语言实现;

2) 应用框架，包括窗口管理器，内容提供者，视图系统等，开发可以访问提问的 api 进行程序设计，由 java 语言实现;

3) 系统库和 Android 运行时也叫 native 层，主要是本地服务和链接库，包含了 java 语言所需要调用的函数和虚拟机等，c 或者 c++实现效率高;

4) linux 内核，提供相应的协议和驱动。



那这跟我们的内存测试有什么关系呢？请大家注意上图中的 dalvik virtual matchine，这是个什么东西呢，通常我们简写为 DVM，现在大家是不是觉得熟悉多了，就是 Android 虚拟机，是 Android 系统中 java 虚拟机，我们的程序就运行在这里面。看来我们还有必要了解一下 JVM 的内存模型。

2. JVM 内存模型

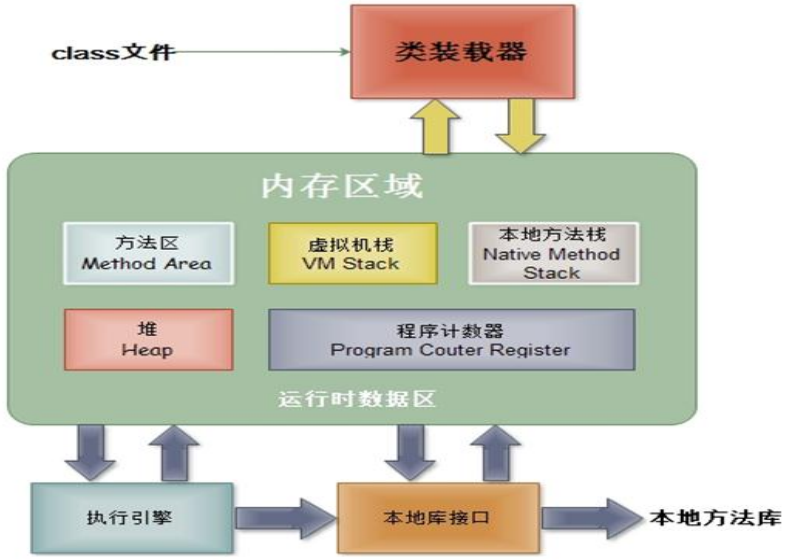
我们一起来看一下 java 虚拟机的内存区域都有哪几部分:

1) 方法区：用于存储已被虚拟机加载的类信息、常量、静态常量、即时编译器编译后的代码等数据。

2) 程序计数器：当前线程所执行的字节码的行号指示器。

3) stack：当方法被执行时，存放函数地址、参数、局部变量、对象的引用，在方法执行结束时会自动释放，由操作系统负责

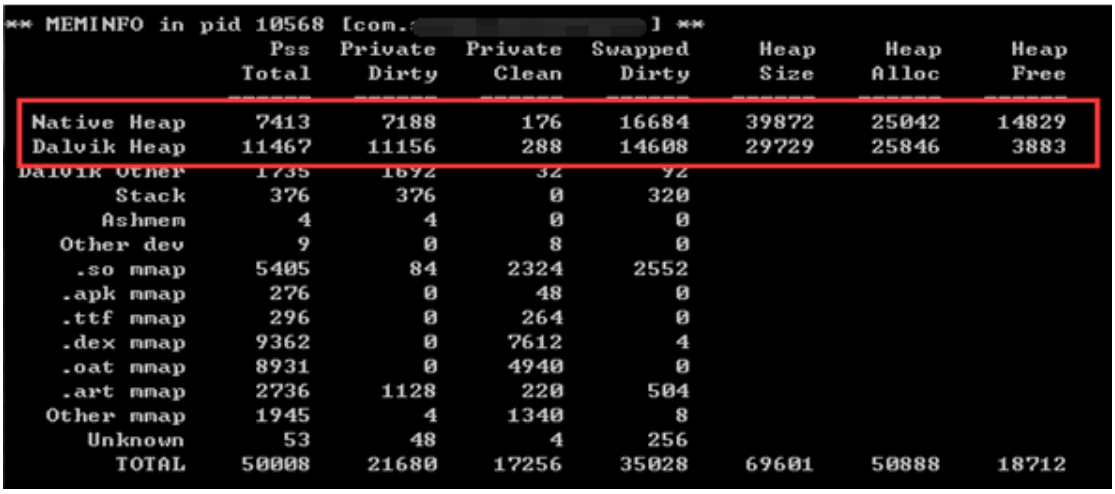
4) heap：程序运行时直接 new 出来的内存（对象实例）。这部分内存在不使用时将会由 Java 垃圾回收器来负责回收。



了解完以上的概念之后，相信大家不难得出结论，我们做 Android 内存测试，就是测 dalvik 虚拟机里的 heap 嘛。在这里我必须承认大家真的是很机智，因为我们几乎已经要找到答案了，但是现实往往都很残酷，我们越是接近真相的时候越要加倍的小心，才能不会前功尽弃。为什么这么说呢，首先有一点是很正确的，我们在做内存测试的时候，确实几乎可以等同于在做 heap 的大小测试，但是并不仅仅只有 java 虚拟机里面的 heap。从上面的 Android 分层架构模型中，我们知道 dalvik 虚拟机只是其中的一部分，并不是孙猴子一样跳出三界外，不在五行中呀，怎么可能不跟其他部分打交道呢，所以除了虚拟机里面的 heap，还有 native heap。

3. Dalvik heap 和 native heap

到这里是不是有小伙伴已经在心里默默呐喊：证据呢、证据呢！不信看下图：



** MEMINFO in pid 10568 [com.[redacted]] **

	Pss Total	Private Dirty	Private Clean	Swapped Dirty	Heap Size	Heap Alloc	Heap Free
Native Heap	7413	7188	176	16684	39872	25042	14829
Dalvik Heap	11467	11156	288	14608	29729	25846	3883
Dalvik Other	1735	1692	32	92			
Stack	376	376	0	320			
Ashmem	4	4	0	0			
Other dev	9	0	8	0			
.so mmap	5405	84	2324	2552			
.apk mmap	276	0	48	0			
.ttf mmap	296	0	264	0			
.dex mmap	9362	0	7612	4			
.oat mmap	8931	0	4940	0			
.art mmap	2736	1128	220	504			
Other mmap	1945	4	1340	8			
Unknown	53	48	4	256			
TOTAL	50008	21680	17256	35028	69601	50888	18712

我们先不要管笔者是如何得到的这张图（相信很多机智的小伙伴都知道这张图的来

历，当然不知道也没关系，先卖个关子，让丑媳妇先躲一会），从图中可以清晰的看到 heap 的组成：dalvik heap 和 native heap。Dalvik heap 是 dalvik 虚拟机中 java 代码占用的 heap，那 native heap 呢？既然 dalvik 虚拟机只是整个 Android 系统中的一部分，那么必然会涉及到很多的本地库、第三方库调用，当调用了 C 程序时，占用的 heap 就是 native heap 了。

聊到这里很多机智的小伙伴肯定已经了然于心中：所以咯，Android 内存测试主要就是测 dalvik heap 和 native heap 咯（请原谅我的港台腔）。你真的很机智！

4. Heap 阈值

我们知道了要测 dalvik heap 和 native heap，那么问题来了，heap 的大小有没有限制呢，答案当然是肯定的：

1) Native：通过 JNI 等方式申请的内存，理论上只要手机还有内存就可以继续申请，但内存越大，切到后台越容易被 kill

2) Dalvik：java 代码中申请的内存大小，大小受限，超过限制会 oom

5. 内存快照

那如果发现了内存泄露，要如何定位问题呢？我们知道 java 依靠垃圾回收机制来管理内存，创建的对象到垃圾回收 root 节点还有引用路径，GC 就不会回收这个对象，所以我们可以通过分析内存快照的方式，找出可疑对象的最短引用路径，找到引起内存泄露的根源，这部分我们会在后面详细讲解，这里我们先大致了解一些即可。

三、实战之基础测试

讲到这里很多小伙伴心里应该在嘀咕几件事情了：

1）我们要测 Native heap，而且 native heap 只要手机还有内存就可以继续申请，那么手机还有多少内存可以申请呢

2）我们要测 Dalvik heap，而且 dalvik heap 大小受限制，超过这个限制会 oom，那么这个限制是多少呢

3）如果内存泄露了，那么 heap 值会越来越大，那我们怎么获得这个值呢

下面我们一步一步来，逐个的解决这些问题。前面我们已经多次提到过 Android 系统是基于 linux 内核实现的（有些小伙伴表示没有看到，心机 boy 要告诉你，我要向老

师举报你，打你一百个小板子），那么很多 linux 命令就可以用了，如果你是一个刚刚接触 Android 测试的人，笔者可以告诉你，在 Android 系统中使用 linux 命令的方式是：手机连接 PC 后，在 cmd 命令行窗口中敲入 adb shell，前提是你需要有 Android 的开发环境。

1. native heap 阈值测试

下面需要大家打起一百分的精神了，因为我们要开始讨论我们的第一个非常严肃的问题了，如何获得手机还有多少内存可以申请

```
shell@cancro:/ $ cat /proc/meminfo
MemTotal:        1894836 kB
MemFree:           86964 kB
Buffers:          116516 kB
Cached:           671328 kB
```

答案是：adb shell cat /proc/meminfo

得到的结果中，我们应当关注图中的几项

1) Memtotal：设备 ram 总大小

2) Memfree：设备 free 内存大小

3) Cached：设备 cached 内存大小

Native heap 最大值=memfree+cached（当然这不是一个非常准确的值，但是基本就是这么大了，相信你的应用也不会真的要一口吃掉手机的所有内存）

2. dalvik heap 阈值测试

首先，肯定有很多小伙伴要问，为什么 dalvik heap 会有大小限制呢，最简单的解释：你一个人要吃掉家里所有的果冻，这样合适吗？我们都知道每一个 Android 应用都是由 zygote fork 出来的（如果你不知道的话，现在知道了吧，可以把 zygote 看做是母亲，可以生很多儿子，每个儿子都可以继承母亲的遗传基因，同时也可以通过母亲拿到的家里资源），那么 zygote fork 程序之前，有些事情是需要定下来的，比如可以吃几个果冻。所以 dalvik 内存的大小是有限制的。

废话撒了这么多，那 dalvik heap 大小限制到底是多少呢？我们越接近真相，越是要更加的谨慎，同样通过 adb shell getprop|grep dalvik 可以得到我们要的信息

```
shell@cancro:/ $ getprop|grep dalvik
[dalvik.vm.heapgrowthlimit]: [128m]
[dalvik.vm.heapmaxfree]: [32m]
[dalvik.vm.heapminfree]: [8m]
[dalvik.vm.heapsize]: [512m]
[dalvik.vm.heapstartsize]: [8m]
[dalvik.vm.heaptargetutilization]: [0.75]
[dalvik.vm.stack-trace-file]: [/data/anr/traces.txt]
[persist.sys.dalvik.vm.lib]: [libdvm.so]
```

是的，系统关于 dalvik heap 大小限制的信息竟然这么多，所以笔者总是提醒大家要加倍的谨慎。

1）Heapstartsize：进程的起始 heap 值

2）Heapgrowthlimit：进程的最大 heap 值

3）Heapsize：程序使用了 largeheap 时，heap 最大值

dalvik heap 限制=最大值=heapgrowthlimit（使用 largeheap 时为 heapseize）

这几个概念如何理解呢：母亲决定自己生的每个儿子都应该至少吃 1 个果冻，这就是 heapstartsize；但是你就是想多吃几个果冻呀，所以母亲又告诉你，最多只能吃 8 个果冻，这就是 heapgrowthlimit；但是总有一些绝世高手，万中无一，他们还没出生时就告诉妈妈，我要吃 20 个，不然我就不出生了，妈妈当然也没办法，这就是 heapsize（需要在代码中声明了 largeheap，特别需要内存，所以 DVM 初始化时，heap 限制就比正常的要大）

3. Heap 值测试

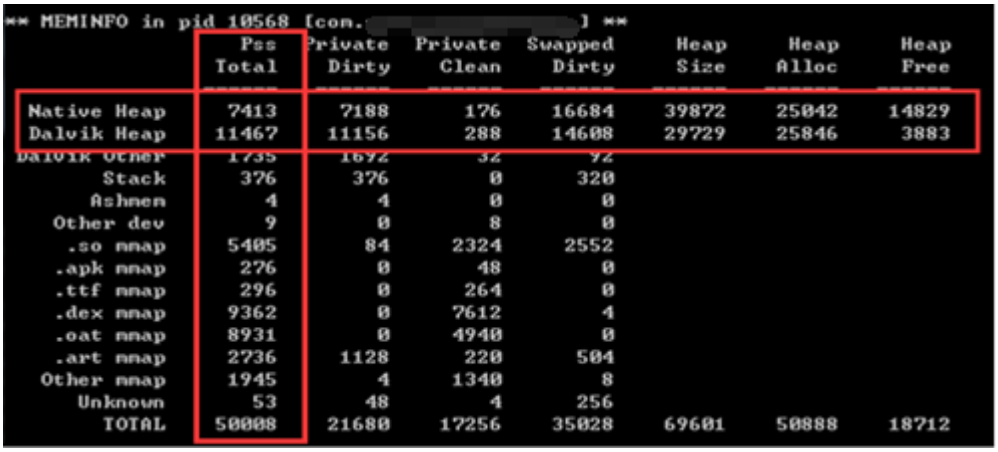
讲到 heap 大小测试，机智的小伙伴一定在心里嘀咕了：前面不是有一张图吗？里面似乎已经有了 dalvik heap、nativie heap 的很多信息，在这里我不得不夸赞你们真的是太聪明了，简直要上天啊。Android 给我们提供了一系列的定制的 shell 命令，其中就有我们的丑媳妇 dumpsys。Dumpsys 的功能有多强大呢，看下图吧。

adb shell dumpsys

Also you can apply filters to running services:

1 SurfaceFlinger
2 accessibility
3 account
4 activity
5 alarm
6 appwidget
7 audio
8 backup
9 battery
10 batteryinfo
11 bluetooth
12 bluetooth_a2dp
13 clipboard
14 connectivity
15 content
16 **cpuinfo**
17 device_policy
18 devicestoragemonitor
19 diskstats
20 dropbox
21 entropy
22 ethernet
23 hardware
24 input_method
25 iphonesubinfo
26 isms
27 keybar
28 location
29 media.audio_flinger
30 media.audio_policy
31 media.camera
32 media.player
33 meminfo
34 mount
35 netstat
36 network_management
37 notification
38 package
39 permission
40 phone
41 power
42 search
43 sensorservice
44 simphonebook
45 statusbar
46 telephony.registry
47 throttle
48 uimode
49 usagestats
50 vibrator
51 wallpaper
52 **wifi**
53 window

我们获取 heap 大小的是 dumpsys meminfo

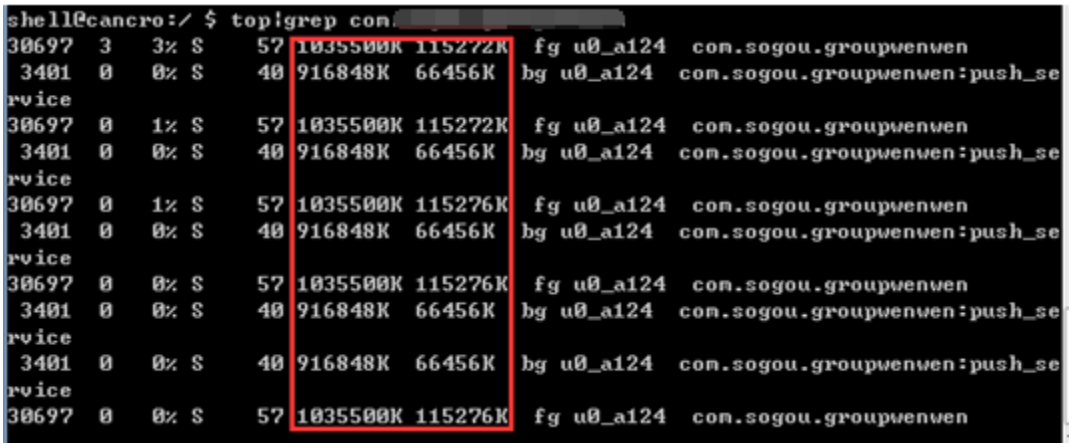


** MEMINFO in pid 10568 [com.[redacted]] **

	Pss Total	Private Dirty	Private Clean	Swapped Dirty	Heap Size	Heap Alloc	Heap Free
Native Heap	7413	7188	176	16684	39872	25042	14829
Dalvik Heap	11467	11156	288	14608	29729	25846	3883
Dalvik Other	1735	1672	32	92			
Stack	376	376	0	320			
Ashmen	4	4	0	0			
Other dev	9	0	8	0			
.so mmap	5405	84	2324	2552			
.apk mmap	276	0	48	0			
.ttf mmap	296	0	264	0			
.dex mmap	9362	0	7612	4			
.oat mmap	8931	0	4940	0			
.art mmap	2736	1128	220	504			
Other mmap	1945	4	1340	8			
Unknown	53	48	4	256			
TOTAL	50008	21680	17256	35028	69601	50888	18712

使用 dumpsys meminfo <packagename/pid>可以获取 pss、native heap、dalvik heap

当然机智的小伙伴也一定想到了，既然是 linux 内核，那么 top 也可以查看到进程的内存，没错，非常的聪明



使用 top | grep <packagename>可以获取 VSS、RSS

Pss、vss、rss 的区别

机智的小伙伴又要发问了，为什么有 pss、vss、rss 呢，这里我们稍作解释

- VSS: Virtual Set Size 虚拟耗用内存（包含共享库占用的内存）
- RSS: Resident Set Size 实际使用物理内存（包含共享库占用的内存）
- PSS: 除去共享的实存后进程独自占用的物理内存大小。

所以我们可以测试 pss 中 native heap 和 dalvike heap 的值（就是图中两个红框的交叉部分）来做 heap 的大小统计，这样除去了很多干扰。

4. 结果分析

其实讲到这里，该说的大部分都说完了（当然你要知道笔者肯定不只有这么几把刷子，我可是粉刷匠，有很多刷子呢，后面还有更有趣的），我们回顾一下前面讲到的内容:

1） Cat /proc/meminfo，测 native heap 阈值

2） Getprop|grep dalvik，测 dalvik heap 阈值

3） Dumpsys meminfo <packagename/pid>，测 dalvik heap、native heap、pss 大小

4） Top|grep <packagename/pid>，测进程 vss、rss 大小

有了这些测试结果后，如何分析就很明了了:

1） Rss、pss 持续增长，内存整体异常

2）Dalvik heap 持续增长，java 层内存异常

3）Native heap 持续增长，native 层内存异常

4）Dalvik heap 值趋于平稳，但是接近阈值，java 层内存过大

四、实战之快速测试

上面是 Android 内存测试的一些基础方法，相信小伙伴们已经发现了一些问题的命门所在

1）需要配合稳定性测试，测试时间耗时过长

2）问题定位困难

有没有别的方法呢，能更快的测试，更好的定位问题。既然你们诚心的发问了，我就大发慈悲的告诉你，我是来自搜狗的粉刷匠（原谅心机 boy 无耻的引用宠物小精灵的台词）

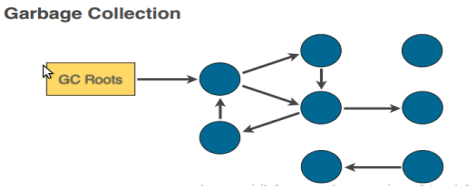
1）分析内存快照

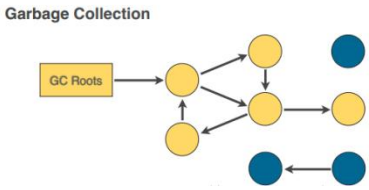
2）知道此刻内存里应该有什么，不应该有什么

3）从 activity/fragment 入手

1. 垃圾回收

前两点不难理解，那为什么要从 activity/fragment 入手呢，前面我们讲到了，java 是依靠垃圾回收机智进行内存管理的，那么垃圾回收是如何确定哪些对象可以被回收呢？





垃圾回收依靠引用路径来确定对象是否可以被回收，如上图所示，图中的黄球到 CG root 有引用路径，所以不能被回收，而篮球可以被回收。

2. Activity 生命周期

了解了垃圾回收机制，我们来看下 activity 和 fragment。为什么快速内存测试时，我们可以分析内存快照中的 activity/fragment

1）Activity 是 Android 四大组件之一，是用户的视觉承载，也是绝大多数其他组件的承载。

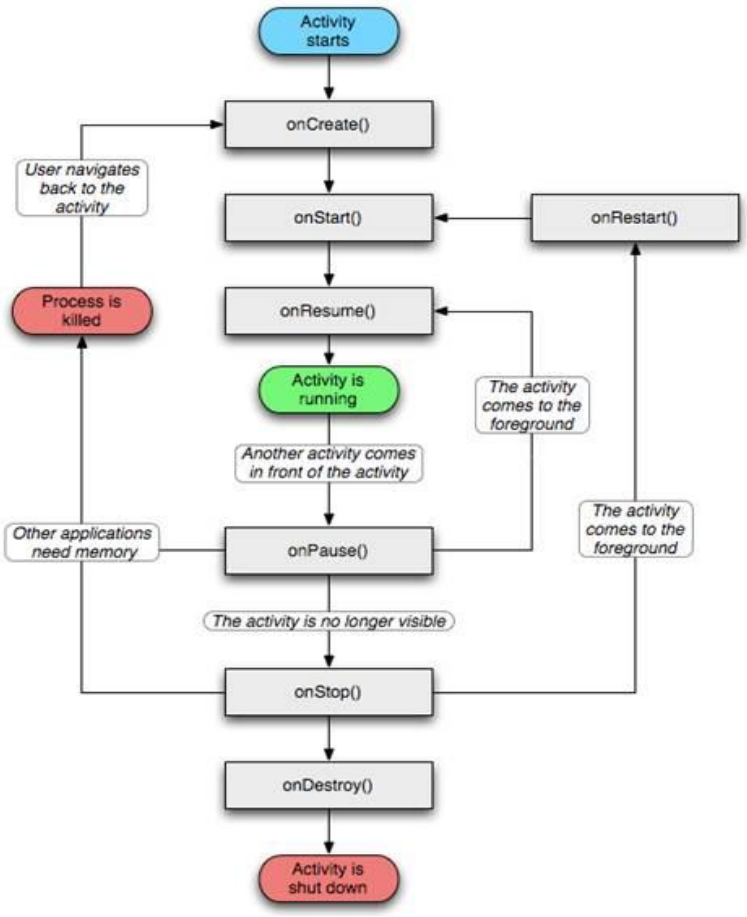
2）Activity 是 Android 四大组件中使用最多的。

3）其他三个组件在很大程度上依赖于 activity。

4）Activity 的界面可以由不同的 Fragment 组成，fragment 拥有自己的生命周期，并且依附于 activity 的生命周期

5）Activity/fragment 泄露会导致里面所有的资源文件都无法释放，泄露较大。

那么问题来了，activity 和 fragment 何时应该被释放呢？我们要先从他们的生命周期谈起。下图是 activity 的生命周期图，笔者结合 activity 的生命周期做了一些列操作：从 A 界面跳转到 B 界面，然后退出 B 界面返回 A 界面，同时在回调函数中打印了 log



大家可能看着有点乱，不过我已经非常机智的帮大家整理了一番

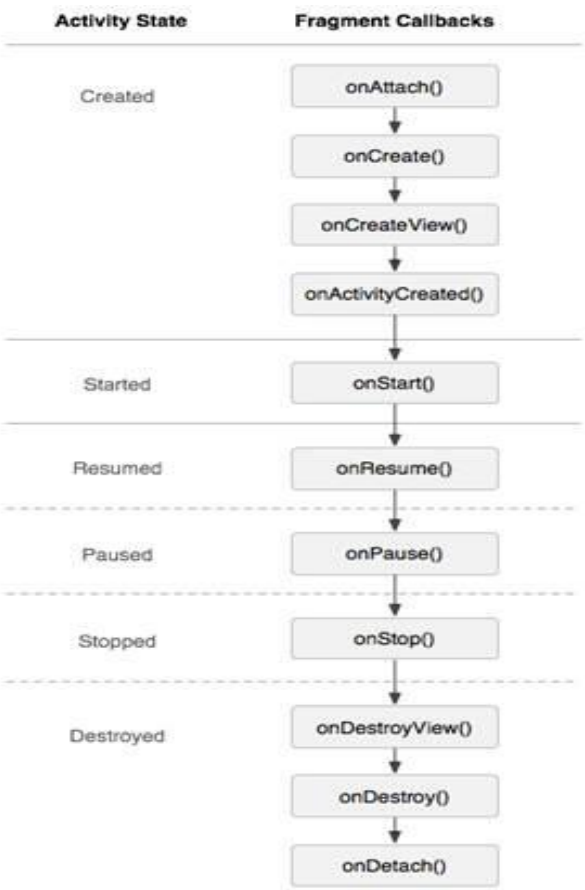
A 执行的回调函数有：onpause—》onstop—》onrestart—》onstart—》onresume

B 执行的回调函数有：oncreate--》onstart—》onresume—》onpause—》onstop—》ondestory

结论就是：从 A->B->A 的操作过程中，内存中不应该有 B 对应的 activity

3. Fragment 生命周期

前面我们说了 fragment 有独立的生命周期，并且依附于 activity 的生命周期，下图是这部分的对应关系。



从上图我们可以看出:

1） Fragement 依附于 activity，且随 activity 的生命周期变化而变化

2） Activity 生命周期结束、fragement 的生命周期也相应结束

结论：从 A->B->A 的操作过程中，内存中不应该有 B activity 包含的 fragement

4. 如何快速测试

到这里，我要说的也基本说完了，我们一起来回顾一下刚才说过的内容

1） 我们需要获取内存快照

2） 我们需要知道内存快照里面不应该有什么

3） 我们可以先关注内存快照里面的 activity 和 fragement

4） A->B->A，这个操作，内存里面不应该有 B activity 和 fragement

那么相应的测试方法已经浮出水面了:

1） app 功能测试结束，回到 app 主界面，导出内存快照

2）分析内存快照内容，查找其中有 GC root 引用路径的 activity、fragement

3）非主界面对应的 activity、fragement，为内存泄露对象

五、测试工具

讲到这里，肯定有小伙伴要问了，除了 adb shell，还有别的测试工具吗，哈哈，机智的你想必已经知道了答案是肯定的，下面我们罗列一下常用的测试工具和工具的试用场景、优劣对比

工具	优势	劣势	适用场景
Adb shell	使用成本较低，只需安装 Android 开发环境 易于上手，跟 linux 命令基本一致 无需源码	数据较粗糙，不够细致，不利于定位问题 数据需要二次处理	测试 heap 大小及增长趋势
GT	使用成本低，只需安装 GT 安装包 易于上手，有可视化界面 可以直接导出统计数据	数据粗糙，不利于定位问题	测试 pss 大小及增长趋势
DDMS	集成了很多工具，可以统计很多方面	工具比较重，需要一定 Android 知识 统计结果繁多，分析困难	测试 heap 大小、组成、增长趋势
Leakcanary	使用简单、图形化界面、可以屏蔽部分 SDK 引发的泄露	需要修改工程源码、对 Android 版本兼容不佳	测试 activity/fragment 泄露
MAT	功能强大、支持筛选、查找、对比	使用麻烦、对 Android 知识要求高	分析内存快照
Android studio	使用较方便、支持自动分析	没有 MAT 功能强大	分析内存快照

六、结尾

我们的故情到了这里也要接近尾声了，好在这只是我们这个系列的开始，而且我此刻心里深深的感受到了小伙伴们的怨气，为什么通篇没有给我们详细的讲解一下工具使用啊，但是你们要明白笔者的良苦用心，我并不想剥夺你们自己去挖掘这些的权利，最重要的原因是，最重要的原因是，最重要的原因是（重要的事情说三遍）：欲知后事如何，且听下回分解。

用 FSM 写 Case，玩过没？

◆ 腾讯 TMQ：黎懋靓

一、引言

腾讯测试工程师小新一是一名安卓客户端测试工程师，对于安卓客户端的功能测试、自动化测试和性能测试方面都有着非常丰富的经验。最近小新一被通知负责某二手交易 APP 的功能测试，在初步了解了该 APP 后，小新一皱起了眉头。

该 APP 虽然看起来功能简单，只是提供了一个买家和卖家的交易沟通平台，但是其中涉及到了多个实体的状态变迁，如果只是对于需求进行测试用例设计的话，很难保证所有的功能路径都被覆盖了，而且测试用例对于路径的覆盖无法区分优先级。这样的测试用例是远远不能保证到产品的质量的。

针对这个情况，小新一和测试分析小组负责人锅仔进行了一次深入的沟通，在听完小新一对于测试任务的描述后，锅仔提出了使用基于状态的测试方法来完成对于该 APP 的测试。

那么什么是状态机呢？什么又是基于状态的测试呢？怎么使用基于状态的测试呢？基于状态的测试适用于什么情况呢？在使用状态机的时候需要注意哪些事项呢？如果你对这些问题还存有疑问，那么请看官继续往下看，和小新一一起，学习基于状态的测试方法。

二、基于状态的测试

2.1 定义

基于状态的测试是一种基于模型的测试方法，作为黑盒测试设计技术中的一种，常被用于事件驱动的系统中。基于状态的测试核心思路是通过遍历系统所有的状态转换迁移，来保证整个系统功能的正常。

2.2 状态机

顾名思义，基于状态机的测试，其核心模型就是状态机，也叫状态图。状态机的组成其实比较简单，要素大致有三个：输入，输出，还有状态。输入和输出比较容易理解，那么什么叫做状态呢？状态就是对象生命期中的条件或情况，在这种状态中，对象满足某种条件，执行某种活动，或者等待某种事件。

在基于状态的测试中，状态机的准确度直接决定了测试效果，所以状态机的绘制是非常重要的一环，我们可以通过以下三步来分析如何绘制状态机：

步骤一：列出研究对象拥有的各种状态

通过启发式的探索来发现系统的状态：

1）通过三个简单问题发现状态：有没有什么事情是我现在可以做但之前不可以做的？有没有什么事情是我现在不可以做但之前可以做的？我现在所采取的行动是否产生了和之前不同的结果？

2）留意用于描述正在发生事情的言辞，如“当……的时候”(While)、“当系统正在导入数据的时候……”、“当账户被冻结的时候……”

3）每个状态都由事件所触发，认出状态可回过头找出触发事件，反之亦然

步骤二：列出状态之间的转换，确定引起各个转换的事件

在步骤一的基础上，考虑状态之间的事件。从测试的视角来看，引起状态转换的事件可以分为三种类型：

1）外部产生事件：来自于软件之外的任何事件，如用户操作

2）系统产生事件：软件自己产生的任何事件，如系统完成了某些后台活动而产生的结果

3）时间流逝：超时、计时事件（如 After 3 sec）

步骤三：分析各个转换过程中发生的事情

转换代表了从一种状态到另一种状态的改变，当然也可以是自身到自身的。每个状态都可以指定三种可选的信息：

1）触发器：触发器对应事件

2）守卫：守卫是一个布尔表达示，事件发生时，守卫必须为真，转换才会执行

3）效果：效果是在转换过程中执行的行为（活动或交互）

步骤四：状态机 Review

在上面三步做完之后，就形成了一个状态机。但是，状态机的正确性，完整性是否可以得到保证呢？状态机的绘制是否符合规范呢？这些都需要通过人工或者工具的方式进行 review。

以腾讯地图 APP 的收藏夹模块为例，我们尝试对其进行状态机建模。

1、列举研究对象的各种状态。收藏夹功能模块包含的对象比较简单，就是收藏夹页，这个页面包含了以下六个状态：

1）未登录/无数据态

2）未登录/有数据态

3）微信登录/同步态

4）微信登录/未同步态

5）QQ 登录/同步态

6）QQ 登录/未同步态

2、列举各个状态之间的转换，确定各个转换的事件。

从收藏夹需求中，我们不难得出收藏夹六个状态之间的转换关系如下：

1）在状态 1 添加数据，进入状态 2

2）在状态 2 修改数据，保持状态 2;

3）在状态 2 将数据全部删除，进入状态 1

4）在状态 1 进行微信登录，进入状态 3

5）在状态 1 进行 QQ 登录，进入状态 5

6）在状态 2 进行微信登录，进入状态 4

7）在状态 2 进行 QQ 登录，进入状态 6

3、在列举完所有状态转换事件之后，对各个事件以触发器/守卫/效果三个维度进行分析。

对于事件 1，触发器为添加了收藏点或者常用地址，守卫为网络畅通，效果为在收藏夹页面添加了相应的收藏夹数据。

在上面三个步骤执行玩之后，我们可以得到收藏夹模块的状态图，如下所示：

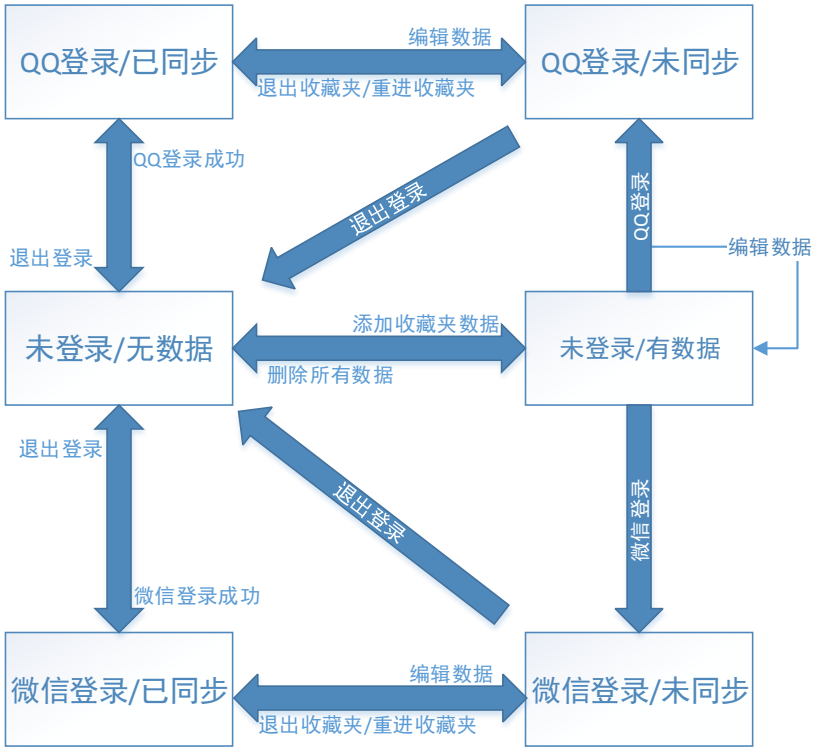


图 2.1 腾讯地图 APP 收藏夹功能状态机

2.3 适用范围及注意事项

基于状态的测试适用于下面的情况:

需求或者设计中使用了状态机

功能中包含了实体的增删查改

功能属于事件驱动型，并且系统状态容易识别

在需要对关键模块/业务进入深入的测试的时候使用

状态机绘制的注意事项:

保证每个状态的单一性，但是也要谨慎选择状态集合，避免状态空间爆炸

选择合适的分析实体，使用层次化方法以管理复杂性

要关注可观察的行为而不是实现细节

借助工具思考（NMODEL，基于整数规划的覆盖方法，后面会讲到）

2.4 状态机绘制实例

2.4.1 功能需求描述

闲贝是一款提供二手物品交易的 APP，交易模块是该 APP 的重点功能。交易过程中，卖家可以发布自己的闲置物品，买家可以根据分类来搜索自己想要购买的物品，当看到自己心仪的物品后可以拍下，当然也可以直接和买家联系，当买家拍下后，卖家可以收到系统发来的 IM 消息通知，从而处理发货等流程，当然在买卖过程中如果发生退款退货等情况的时候，双方可以自行处理，如果一方不满意的话，可以发起申请，平台介入后会根据双方提交的证据在进行仲裁，仲裁的结束后，会将处理结果告知双方。

2.4.2 模块选择过程

一开始，我们认为在买卖过程中，我们只需要以“买家”和“卖家”作为两个元素来进行建模，就可以覆盖到所有的状态，因此我便画出了如下的状态转化图:

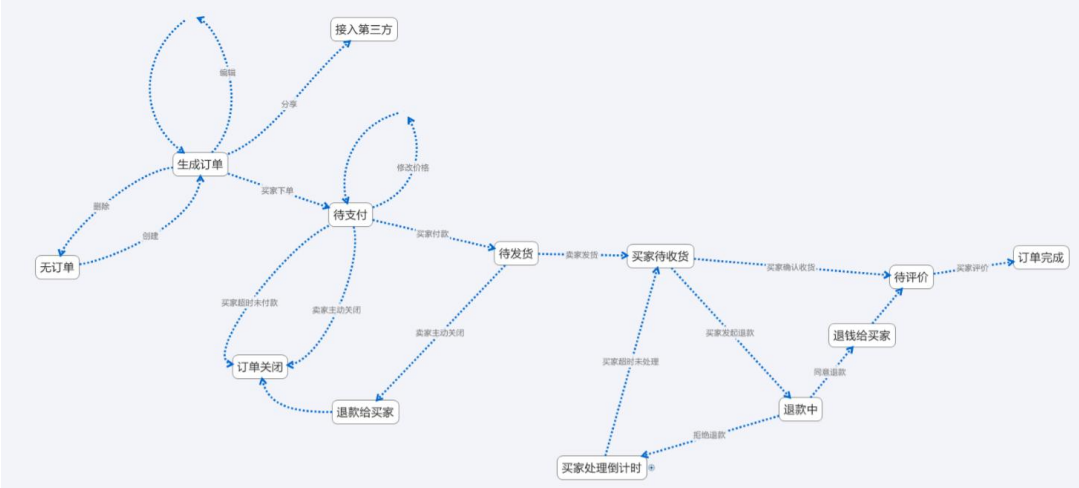


图 2.2 卖家的订单状态图

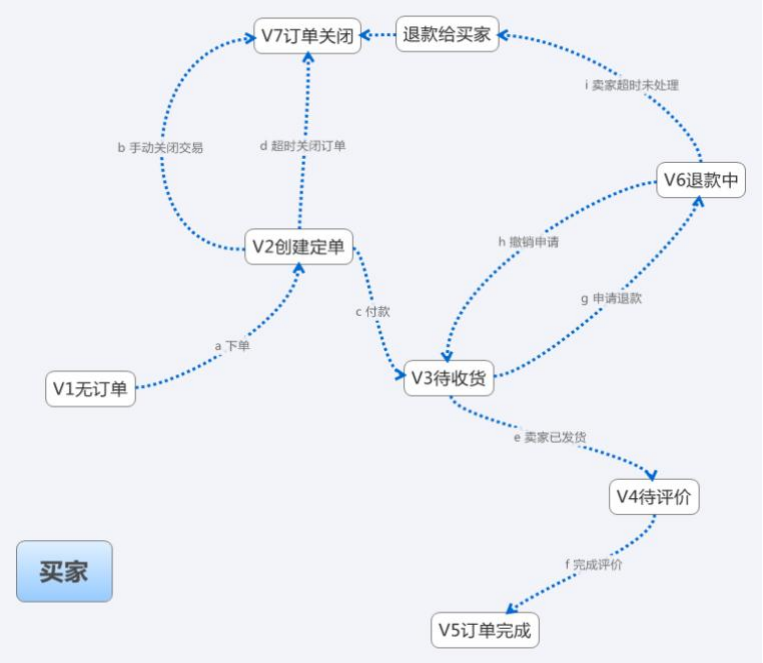


图 2.3 卖家的订单状态图

然而发现，这其中有一个问题，就是根据这样的状态图去设计测试用例，设计出的都是针对一方的，而在实际的买卖过程中，只有一方的操作时无法完成整个交易的。另外还会有一个问题就是，有些异常的情况是无法覆盖到的。举个例子：比如买家从 V2（待付款）到 V3（代收货）这个过程中，如果卖家关闭了订单，会发生什么呢？

因此我觉得应该将买家和卖家放到一起，将他们的操作流程给串起来，于是我又画出了如下的状态图：

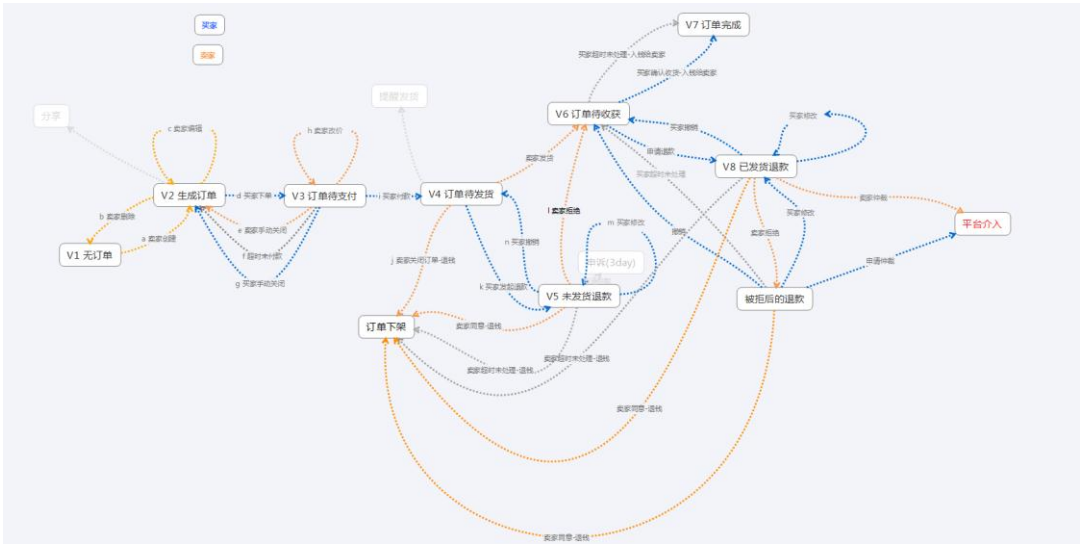


图 2.4 初步融合起来的状态图

然而这个图还是存在问题，当然这个问题在于我对一开始分析对象划分的不够细，

其实这里面，应该划分为“订单”，“物品”，“操作者”，这里操作者又分为“买家”“卖家”“系统”“管理员”，这里应该订单这个元素状态较多，并且贯穿于整个的交易流程中，因此我们就选择“订单”作为此次测试建模的一个对象。

2.4.3 状态机绘制

基于上面的分析，我们最终确定了状态机的对象是订单，于是我们对订单的状态进行了一个列举（从后台的代码中列举出所有订单的状态码），然后画出状态转化图:

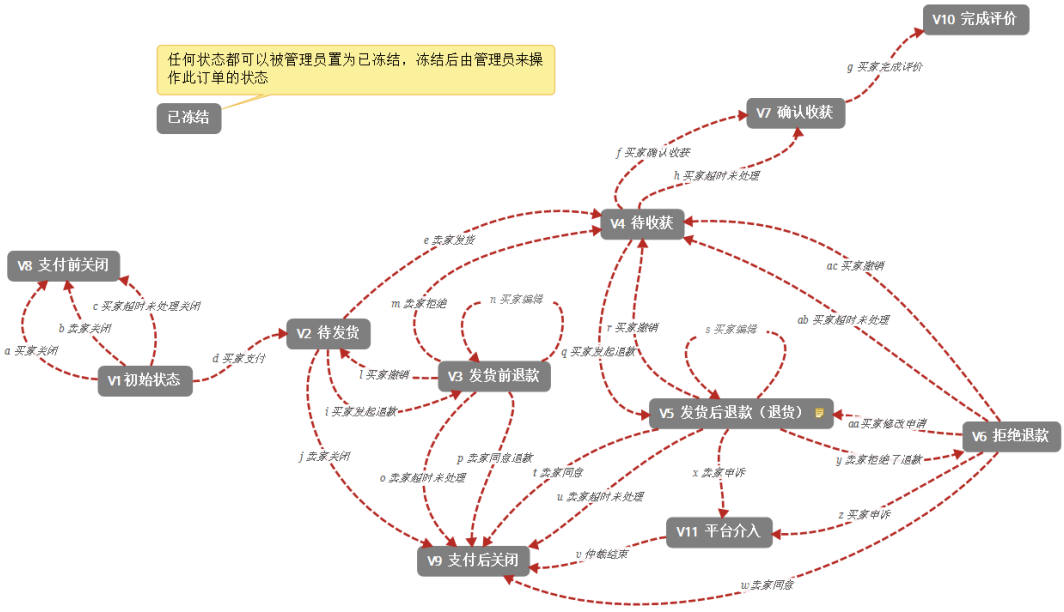


图 2.5 订单的状态图

三、从状态机到测试用例

在针对所测功能模块绘制完状态机后，下一步便是在状态机的基础上生成测试用例。由于状态机描述了系统状态的所有转换，所以在构造测试用例的时候，只要保证状态机中的状态转换均被覆盖了，就能保证功能的测试完整了。比较通用的方法是通过单一状态转换表和转换对，构造几条覆盖全部状态的路径，以这几条路径为基础，生成基础测试用例。

3.1 简单状态转换覆盖

对一个状态机进行全覆盖，最简单的方法就是取出所有状态转换的状态对，对其进行逐一覆盖，我们称这种测试用例生成方法为简单状态覆盖方法。

首先，我们根据已经生成的状态机，将所有状态两两组合，形成一个状态转换表。

如下表 3.1 所示:

表 3.1 辅助转化表

No.	Start	Event	End
1	V1	买家支付	V2
2	V2	买家发起退款	V3
3	V1	买家关闭	V8
4	V1	卖家关闭	V8
5	V1	买家超时未处理关闭	V8
6	V3	卖家同意退款	V9
…	…	…	…
n	V5	买家撤销	V4
...	…	…	…

如上表所示，根据转换表第一条，我们需要覆盖从订单初始化到待发货的状态转换，因此我们构造一条用例为:

1）订单创建成功后，买家付款，在卖家发货前，买家发起退款，卖家同意退款后，订单关闭

上面这个用例不仅覆盖了初始化订单到待发货的状态转换，同时也覆盖了 V2 到 V3、V3 到 V9 的转换，因此我们在辅助转换表中，将其标识，如下:

表 3.1 辅助转化表覆盖标识图

No.	Start	Event	End
1	V1	买家支付	V2
2	V2	买家发起退款	V3
3	V1	买家关闭	V8
4	V1	卖家关闭	V8
5	V1	买家超时未处理关闭	V8
6	V3	卖家同意退款	V9
…	…	…	…
n	V5	买家撤销	V4
...	…	…	…

按照这种方法，我们依次对辅助状态转换表中的所有转换进行覆盖用例设计，最终形成全量的测试用例集合。

3.2 手工状态机路径覆盖生成方法

简单状态转换覆盖方法的原理不复杂，但是在具体生成测试用例的时候比较困难，而且容易产生很多冗余的测试用例。在简单状态转换覆盖方法的基础上，我们结合状态

机的路径覆盖方法，将生成的覆盖路径转换成测试用例。

在表 3.1 的基础上，我们将所有的状态抽取出来生成转化对，也就是列出每一个状态的输入流和输出流，然后将输入流和输出流进行排列组合作为状态流，如下表:

表 3.2.辅助转化表

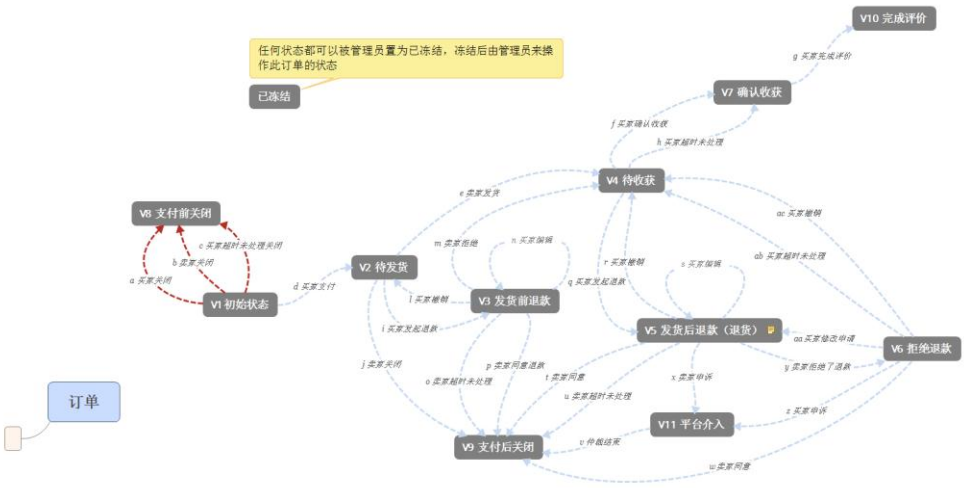
V2	输入	d	l				
	输出	e	i	j			
	状态流	d->e	l->e				
		d->i	l->i				
		d->j	l->j				
V3	输入	i	n				
	输出	m	l	n	o	p	
	状态流	i->m	n->m				
		i->l	n->l				
		i->n	n->n				
		i->o	n->o				
		i->p	n->p				
V4	输入	e	m	r	ab	ac	
	输出	q	f	h			
	状态流	e->q	m->q	r->q	ab->q	ac->q	
		e->f	m->f	r->f	ab->f	ac->f	
		e->h	m->h	r->h	ab->h	ac->h	
V5	输入	q	s	aa			
	输出	r	s	t	u	x	y
	状态流	q->r	s->r	aa->r			
		q->s	s->s	aa->s			
		q->t	s->t	aa->t			
		q->u	s->u	aa->u			
		q->x	s->x	aa->x			
		q->y	s->y	aa->y			
V6	输入	y					
	输出	aa	ab	ac	z	w	
	状态流	y->aa					
		y->ab					
		y->ac					
		y->z					
		y->w					
V7	输入	f	h				
	输出	g					
	状态流	f->g	h->g				
V11	输入	x	z				
	输出	v					

	状态流	x->v	z->v				

这时候“状态流”开始，然后每个终点的字母可以看它是否还是其他“状态流”的起点，如果是那么就继续往下，直到把所有的路径都覆盖完。这里举一个例子:

v2	in	a		
	out	b	c	d
	following	a->b		
		a->c		
		a->d		
v3	in	c	h	
	out	e	g	
	following	c->e	h->e	
		c->g	h->g	
v4	in	e		
	out	f		
	following	e->f	生成 a->c->e->f	
V6	in	g		
	out	h	i	
	following	g->h		
		g->i		

这样我们就可以在图中将覆盖掉的路径颜色进行一个标识，这样直到我们将图中所有的路径都覆盖掉，



这里我手动执行完成后，得到的所有路径为:

1	d	e	q	r	f	g		
2	d	i	m	q	s	t		
3	d	j						
4	d	i	n	o				
5	d	i	p					
6	d	i	l	e	h	g		
7	d	e	q	y	aa	y	z	v
8	d	e	q	y	ab	q	u	
9	d	e	q	y	ac	q	x	v
10	d	e	q	y	w			

3.3 扩充用例

在上述过程中，我们对订单正常状态的覆盖已经达到了。但是在实际的使用过程中，仍然存在这样的问题：卖家操作导致订单状态改变，而此时买家还停留在之前的界面，没有刷新 UI，此时操作的 case。因此针对这类的 case，又延伸出如下一些用例：

买家异常		
订单状态	非买家操作导致订单状态变为	买家
V1	V8	d
V2	V4	i
V3	V4	l
V3	V9	l
V3	V4	n
V3	V9	n
V5	V6	r
V5	V11	r
V5	V9	r
V5	V6	s
V5	V11	s
V5	V9	s
V6	V9	ac
V6	V9	aa
V6	V9	z
卖家异常		
V1	V2	b
V1	V8	b
V2	V3	e
V2	V3	j
V3	V2	m
V3	V2	p
V5	V4	t
V5	V4	x
V5	V4	y
V6	V4	w
V6	V5	w

V6	V11	w

四、基于 NModel 的状态机-测试用例转换方法

不管是简单状态转换覆盖，还是手工状态机路径覆盖方法，在对复杂的状态机进行测试用例转换的时候，都会遇到不小的困难。这个事情并不需要人工的分析过程，能不能让机器帮我们完成呢？答案是肯定的。

NModel（官方地址：http://nmodel.codeplex.com/）是基础状态测试中常用的一个工具，它可以在我们列出对象的状态和执行的动作之后，自动帮我们构建状态图，并且还可以生成用例。其模型创建的原理是:

1.程序是用来处理数据的，数据也可以称作状态（State）;

2.用户通过程序提供的操作界面来处理数据，操作界面也可以称作动作（Action）;

3.数据的更动又反过来影响一些动作是否可以执行。

首先第一步需要抽象状态，在代码中我们用 enmu 类型来表示:

public enum OrderNum { v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11 }

public enum OrderDesc { 初始状态，代发货，发货前退款，代收货，发货后退款，发货后拒绝退款，确认收货，支付前关闭，支付后关闭，完成评价，平台介入 }

接着是模拟一个动作

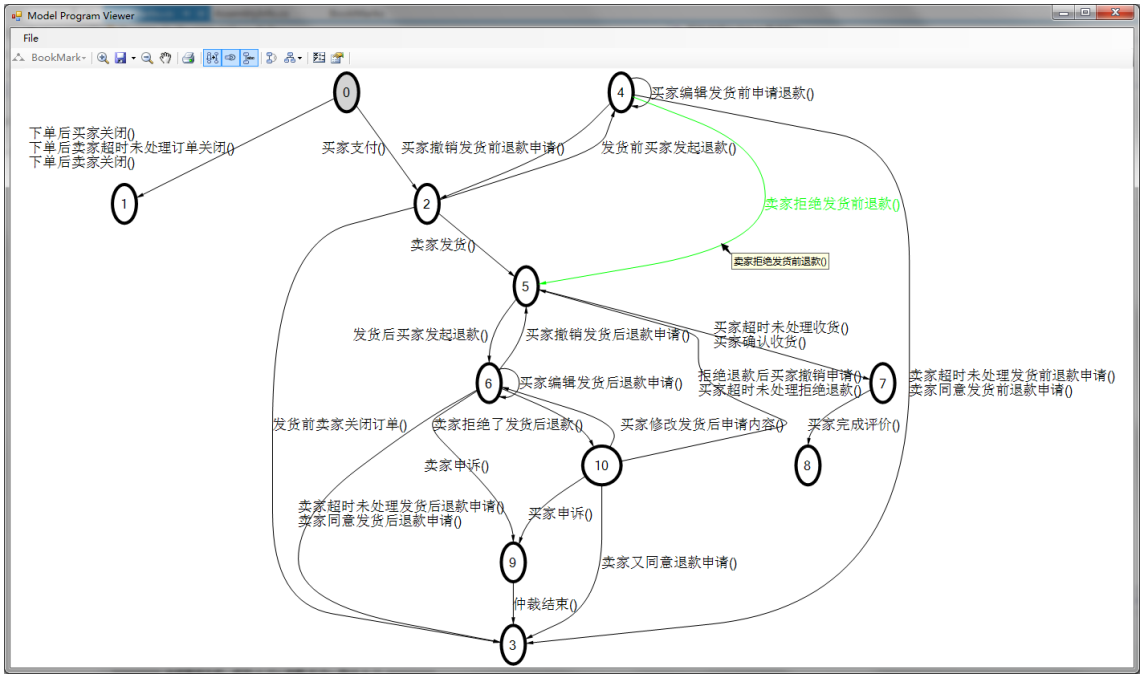
```
[Action]
        // a
        public static void 下单后买家关闭()
        {
            BookMarkShow.SetOrderState(OrderNum.v8);
        }
        public static bool 下单后买家关闭 Enabled()
        {
            return (WebSiteModel.ordernum == OrderNum.v1);
        }
```

标注了[Action]的函数，就是抽象出来的程序所支持的动作，例如 Logout；而在动

作函数名后面加上 Enabled 的函数，是 NModel 用来判定指定的动作是否可以执行。

当我们列出所有的动作之后，可以使用如下命令来生成状态图:

mpv.exe /r:case\BookMarks.dll BookMarks.WebSiteModel.CreateLoginModel



察看模型没有问题后，我们就可以运行如下命令来生成用例了：otg.exe /r:case\BookMarks.dll BookMarks.WebSiteModel.CreateLoginModel /file:TestSuit.txt

```
TestSuite(
    TestCase(
        买家支付(),
        发货前卖家关闭订单()
    ),
    TestCase(
        买家支付(),
        发货前买家发起退款(),
        卖家拒绝发货前退款(),
        买家确认收货()
    ),
    TestCase(
        买家支付(),
```

```
        卖家发货(),
        发货后买家发起退款(),
        卖家拒绝了发货后退款(),
        买家修改发货后申请内容(),
        卖家超时未处理发货后退款申请()
    ),
    TestCase(
        买家支付(),
        发货前买家发起退款(),
        卖家同意发货前退款申请()
    ),
    TestCase(
        买家支付(),
        发货前买家发起退款(),
        买家撤销发货前退款申请(),
        发货前买家发起退款(),
        买家编辑发货前申请退款(),
        卖家超时未处理发货前退款申请()
    ),
    TestCase(
        下单后卖家超时未处理订单关闭()
    ),
    TestCase(
        买家支付(),
        卖家发货(),
        发货后买家发起退款(),
        卖家申诉(),
        仲裁结束()
    ),
    TestCase(
        下单后卖家关闭()
    ),
    TestCase(
```

```
        买家支付(),
        卖家发货(),
        发货后买家发起退款(),
        卖家拒绝了发货后退款(),
        拒绝退款后买家撤销申请(),
        发货后买家发起退款(),
        买家编辑发货后退款申请(),
        买家撤销发货后退款申请(),
        发货后买家发起退款(),
        卖家拒绝了发货后退款(),
        买家超时未处理拒绝退款(),
        发货后买家发起退款(),
        卖家拒绝了发货后退款(),
        卖家又同意退款申请()
    ),
    TestCase(
        下单后买家关闭()
    ),
    TestCase(
        买家支付(),
        卖家发货(),
        发货后买家发起退款(),
        卖家拒绝了发货后退款(),
        买家申诉()
    ),
    TestCase(
        买家支付(),
        卖家发货(),
        发货后买家发起退款(),
        卖家同意发货后退款申请()
    ),
    TestCase(
        买家支付(),
```

```
            卖家发货(),
            买家超时未处理收货(),
            买家完成评价()
        )
    )
```

后台性能测试之时间都去哪儿了

◆ 搜狗测试：周海静

摘要： 通常在做后台性能测试的过程中，性能测试数据的统计大都是由客户端来完成，然而也会经常遇到，比如客户端统计的响应时间和服务端统计的响应时间相差较大的情况，这差的时间都去哪里了呢？是什么原因造成的呢？遇到这种问题我们该如何分析和定位？本文通过在测试推荐系统过程中遇到的一个真实的测试案例，和大家一起聊一聊在客户端和服务端数据相差较大情况下，如何分析和定位问题原因，同时也结合案例给出了一些建议。

一、背景介绍

首先要简单说下推荐系统，这是一个采用 thrift 协议为其他服务提供相关推荐数据的后台服务系统，该服务主要包含 q2u（为用户推荐感兴趣问题），u2u（为用户推荐感兴趣的用户）等多个子服务，要求针对每个子服务做负载测试，寻求系统最大所能承受压力。作为小白的我思考之后很快制定了测试方案：

采用并发的方式请求服务端，通过不断增加对系统的负载，获得系统在不同负载下的性能指标（吞吐量（QPS），平均响应时间，资源利用率等），按照业界常用的 2 的 N 次方并发的方式，快速找到系统最大吞吐量。方案制定好了，小白开心的编写性能测试客户端工具，性能指标直接在客户端中进行统计，测试完成直接就可以得出测试结果统计，看起来就是这么顺利！

然而问题出现了……

二、案例描述：

正当小白开心的测试 u2u 服务的时候，发现 4 个并发请求的情况下，平均响应时间达到了 500ms+以上，而这时候 QPS 却只有 7，当采用 8 个并发请求后，响应时间直接飙升到了 1000ms 以上，QPS 却基本保持 7 不变，按照负载测试原理，貌似已经达到了瓶颈，正准备发个测试报告，等等，时间这么高，看看服务端怎么样呢？

从客户端中打印的 log 可以看出，平均响应时间在 500ms+以上，吞吐量却只有 7

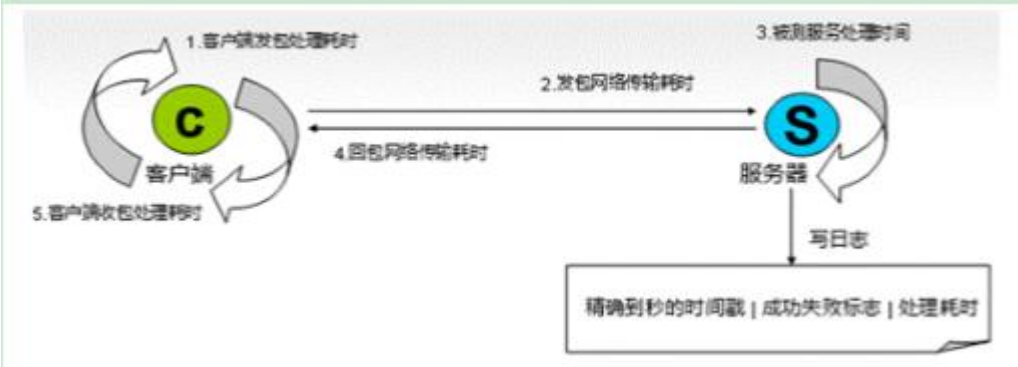
但是从服务端打印的 log 看出，服务的处理时间其实只有二三十毫秒

怎么会差异 500 多毫秒？这差异的时间都去哪了？小白陷入了深思之中……

三、问题分析与定位:

小白本着敬业精神，决定好好研究下问题到底出在哪里?

首先小白编写的客户端工具中统计的响应时间：是从发送请求到请求处理，再到返回整个过程的时间，如下所示:



这个响应时间包含：客户端发包处理耗时+发包网络传输耗时+被测服务处理时间+回包网络传输耗时+客户端收包处理耗时。

服务端统计的耗时主要是被测服务处理时间，那既然两个结果差异很大的话，哪飞走的 500 多毫秒可能就存在以下两个方面:

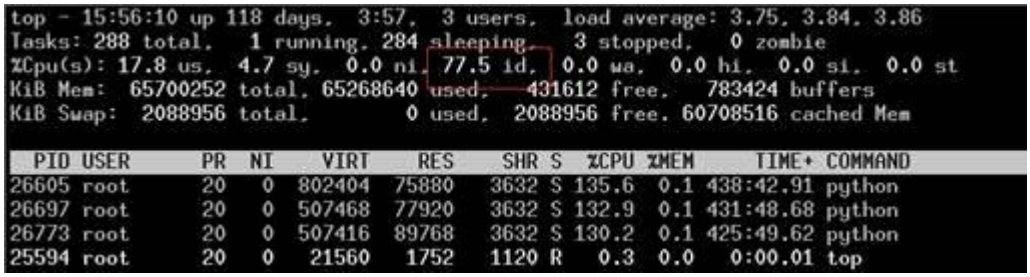
1. 网络传输消耗

一般来说网络传输消耗主要是消耗在不同的地域网络传输上。可是小白的压测客户端和和推荐服务均在深圳同一机房，从客户端机器 ping 服务端机器的时间消耗不到 2ms，可以看出网络消耗应该不是问题瓶颈，小白还是觉得不可靠，又将测试客户端放到和服务器相同的机器上进行负载测试，结果和之前测试结果基本一样，小白认为基本就可以排除是网络传输消耗的原因了。

2. 客户端瓶颈

1) 首先小白测试所在客户端机器在整个的负载测试过程中，机器的整体 cpu 消耗

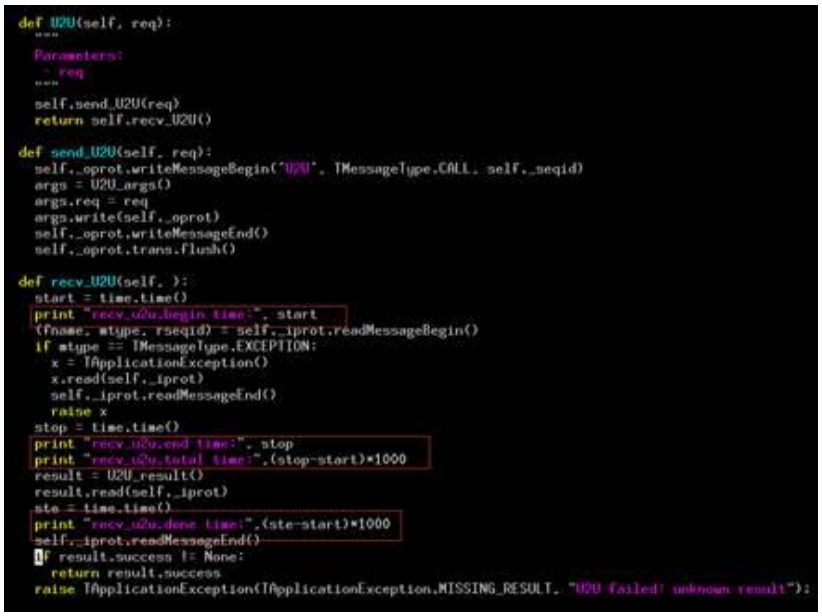
都很低，整体空闲率在 77%以上，如下图所示，基本排除了客户端机器瓶颈。



2) 小白灵思一动，想到一个妙招！决定用两台性能基本相似的机器，将负载均分的方式压测服务端，比如原来用单台 4 个并发的负载，现在采用两台各 2 个并发负载进行压测，对于服务端来说负载压力并没有改变，然而结果却大不相同！最终结果可以看出两台机器客户端的响应时间分别为 200ms+，吞吐量为 7，按照性能测试相同机器叠加的原理，两台机器总的吞吐量应该是 14，平均响应时间为 200ms+，可以看出较之前单台机器响应时间降低了一半多，吞吐量有所提升，所以最值得怀疑的就是客户端处理能力的问题。既然有了可疑点，查起来就方便了。

3) 为了验证瓶颈就在客户端收包处理耗时，小白决定将服务处理时间和解包时间进行分离查看耗时，由于整个采用的是 thrift 协议，在生成的 gen-py 中找到 Cupid.py（这个推荐系统的主接口文件）在他的接收函数中添加时间的统计。

由于之前 client 调用 rpc 接口 U2U 的时候，相当于直接实现了 send_U2U 和 recv_U2U 两个步骤，现在在 recv_U2U 调用过程中加入了时间点统计，接收网络数据包的时间和解析网络数据包的时间，如下所示:



结果可以看出，一共消耗时间是 247ms，其中 4ms 用于网络传输，超过 200ms 是数据包解析的时间。

```
#### 10.241.80.202:13003:316643292 ####
recv_u2u,begin time: 1462882440.24
recv_u2u,end time: 1462882440.27
recv_u2u,total time: 29.7961235046
recv_u2u,done time: 247.8120327
cost: 247.955083847
```

果真大部分时间都消耗在了数据包的解析上面。

小白终于可以稍稍放松下了，可是小白又不明白了？按照以往性能测试，都没有遇到过有如此高的解包时间，这个服务之所以解包耗时能超过 200ms，是因为结果太大吗？

小白没有放弃，决定一探究竟。U2U 接口在请求的时候返回结果个数可以定制，之前测试过程中采用的是从 10-100 中随机取一个数（因为线上系统实际是每次取 10 个结果回来），如下图所示：

```
query=self.queryList[random.randint(0,max_)]
req = Req_U2U()
Info= SceneInfo()
Info.time =int(time.time())
Info.req_src = "wenwen"
#Info.channel_id = 0
#Info.sceneType = 1
#Info.uuid =
Info.debug_level=0
req.sceneInfo = Info
req.uid = random.randint(10000,20000)
req.qq = int(query)
req.rank_num = random.randint(10,100)
#req.mix_method = 0
```

小白将返回结果数降低到 10，再次采用单机 4 个并发对服务器进行负载测试，结果响应时间降低到 160ms 左右，吞吐量提升到 20 左右，如下所示，看来果真是返回结果太多了，加之客户端需要对返回的结果进行反序列化，每条结果的结构体复杂度较高，所以导致了解包过程十分缓慢。

最终小白给出了测试结论

（1） 客户端请求的返回结果条数大于 10 以上，系统的性能出现明显的降低

（2） 服务端接口返回的结果复杂度太高，在客户端解包的时候需要反序列化，消耗了大量的解包时间

建议：优化返回结果包，降低复杂度。

小白长长的舒了一口气，问题原因终于找到了！

四、总结

服务负载测试的时候，便于统计，常采用的方法是通过客户端来统计相应的性能参数，这本身无可厚非，但是这部分统计时间里面包含了网络传输和客户端解包时间，一般情况下这两部分时间消耗都可以忽略不计（基本在 5ms 以内），往往忽略了服务端真正处理时间和客户端响应时间的比对，两者差异较大情况下，就需要特殊关注一下。

性能测试过程中同样也需要和产品，开发进行详细的需求评审，针对服务的各个参数在实际使用中的范围进行确定，比如在这种返回结果条数可控的服务的测试过程中，一般来说都要测试下线上真实拉取返回结果条数情况下，和大于实际情况条件下服务的

性能，并进行比对观察服务性能有没有较大差异。

在发现客户端响应时间和服务端统计时间差异较大情况下，可以将客户端统计的响应时间逐个分布进行统计，比如上述的 thrift 协议中可以借助于 thrift 生成的 send 和 recv 函数进行过程干预，分别打印各个阶段的时间进行分析，快速发现问题。

Nginx 的 proxy_pass 之斜杠的作用

◆ 搜狗测试：曹承臻

近期在配置 nginx 转发功能时。遇到一点小问题，在这里和大家分享一下

需求:

请求先打到加密墙 get.sogou.com，解密后，判断如果 host 是 ping.android.shouji.com 且为 log.gif 的请求，则需要转发到另一台接收服务器。所以就需要在解密服务器上做一下判断转发:

```
    location ~ ^/log.gif
{
    proxy_pass http://hostserver/log.gif?$args;
    access_log logs/${host}_access_log ie;
}
```

重启 nginx 后运行了一下，发现真正 hostserver 收到了 log.gif 的转发请求，但是收到的请求是这样的:

```
10.134.114.165 - - [22/Apr/2016:11:32:08 +0800] ? "POST /log.gif? HTTP/1.0" 200 59 "-" "-"
10.134.114.165 - - [22/Apr/2016:11:32:10 +0800] ? "POST /log.gif? HTTP/1.0" 200 59 "-" "-"
10.134.114.165 - - [22/Apr/2016:11:32:12 +0800] ? "POST /log.gif? HTTP/1.0" 200 59 "-" "-"
10.134.114.165 - - [22/Apr/2016:11:37:26 +0800] ? "POST /log.gif? HTTP/1.0" 200 59 "-" "-"
10.134.114.165 - - [22/Apr/2016:11:37:27 +0800] ? "POST /log.gif? HTTP/1.0" 200 59 "-" "-"
10.134.114.165 - - [22/Apr/2016:11:37:29 +0800] ? "POST /log.gif? HTTP/1.0" 200 59 "-" "-"
10.134.114.165 - - [22/Apr/2016:11:37:58 +0800] ? "POST /log.gif? HTTP/1.0" 200 59 "-" "-"
10.134.114.165 - - [22/Apr/2016:11:38:01 +0800] ? "POST /log.gif? HTTP/1.0" 200 59 "-" "-"
```

正常请求是这样滴:

```
10.142.85.133 - - [21/Apr/2016:19:13:32 +0800]  "POST /log.gif HTTP/1.0" 200 59 "-" "-"
10.142.85.133 - - [21/Apr/2016:19:13:43 +0800]  "POST /log.gif HTTP/1.0" 200 59 "-" "-"
10.142.85.133 - - [21/Apr/2016:19:13:46 +0800]  "POST /log.gif HTTP/1.0" 200 59 "-" "-"
10.142.85.133 - - [21/Apr/2016:19:14:39 +0800]  "POST /log.gif HTTP/1.0" 200 59 "-" "-"
10.142.85.133 - - [21/Apr/2016:19:15:19 +0800]  "POST /log.gif HTTP/1.0" 200 59 "-" "-"
10.142.85.133 - - [21/Apr/2016:19:16:00 +0800]  "POST /log.gif HTTP/1.0" 200 59 "-" "-"
10.142.85.133 - - [21/Apr/2016:19:16:02 +0800]  "POST /log.gif HTTP/1.0" 200 59 "-" "-"
10.142.85.133 - - [21/Apr/2016:19:16:05 +0800]  "POST /log.gif HTTP/1.0" 200 59 "-" "-"
```

可以看到，手动拼的请求和原来的请求有了变化，这可能就是导致接收服务器收到

数据异常的原因。继续挖~

网上查了一下，原来 proxy_pass 方法是可以将请求的参数，log.gif 和 postbody 都带过去的，不需要手动拼。

于是改了一下 nginx 配置:

```
    location ~ ^/log.gif
{
    proxy_pass http://hostserver;
    access_log logs/${host}_access_log ie;
}
```

重新运行一下，再看接收服务器的 log

```
10.134.114.165 - - [22/Apr/2016:11:55:43 +0800]  "POST /log.gif HTTP/1.0" 200 59 "-" "-"
10.134.114.165 - - [22/Apr/2016:11:55:47 +0800]  "POST /log.gif HTTP/1.0" 200 59 "-" "-"
10.134.114.165 - - [22/Apr/2016:11:55:51 +0800]  "POST /log.gif HTTP/1.0" 200 59 "-" "-"
```

不再有前面的？请求正常了，接收服务器也可以正常接收了。

但是细想了一下，对于 proxy_pass 的转发原理还是有困惑：nginx 怎么知道要讲 log.gif 及后面的所有字段都带过去？如果接收到的请求是 http://server/A/B/C/log.gif，那么通过 location ~ log.gif$匹配后，转发过去的是/A/B/C/log.gif 及参数，还是 log.gif 及参数？决定实验一把试试。

测试网址:

http://10.129.156.137/A/B/C/background.html

1、在 137 服务器做如下配置:

```
        location ~* /A/
        {
            proxy_pass http://10.134.68.233;
        }
```

访问网址后，233 服务器接收到的请求为:

```
10.129.156.137 - - [23/Apr/2016:10:21:18 +0800] "GET /A/B/C/background.html H
```

2、在 137 服务器做如下配置:

```
location ~* /B/
{
    proxy_pass http://10.134.68.233;
}
```

访问网址后，233 服务器接收到的请求为:

```
10.129.156.137 - - [23/Apr/2016:10:22:49 +0800] "GET /A/B/C/background.html
```

通过实验，不管 location 匹配的是/A/、/B/、/C/、background.html$，最后转发到 233 服务器时的请求都为

```
10.129.156.137 - - [23/Apr/2016:10:21:18 +0800] "GET /A/B/C/background.html H
```

这说明通过 proxy_pass 转发的是除了 hostname 之外所有的部分。

这样就有另外一个问题了，如果我们就想要 background.html 及后面的部分，怎么办呢?

网上查了一下，proxy_pass 中的/就是用来解决这个问题的。继续尝试:

将 137 服务器做如下修改:

```
location ~* /B/
{
    proxy_pass http://10.134.68.233/;
}
```

运行一下，发现 nginx 报错了:

```
nginx: [emerg] "proxy_pass" cannot have URI part in location given by regular expression, or inside named location, or inside "if" statement, or inside "limit_except" block in /usr/local/nginx/conf/nginx.conf:53
```

根据提示，将正则规则去掉:

```
location /B/
{
    proxy_pass http://10.134.68.233/;
}
```

再运行就没有报错了。

访问以下请求，发现 233 服务器收不到转发请求了，说明没有命中这个 location。细想了一下，原来这个 location 是没有正则匹配的，所以需要些完整匹配:

```
location /A/B/C/
{
    proxy_pass http://10.134.68.233/;
}
```

重新请求一下，发现 233 服务器收到的请求为:

```
10.129.156.137 - - [23/Apr/2016:10:35:29 +0800] "GET /background.html HT
```

从此可以得出结论:

Proxy_pass 末尾带“/”和不带是有区别的:

（1） 不带斜杠转发的是除 hostname 以外的部分，包括目录。可以使用正则表达式匹配 location，且任意正则匹配成功后，转发的都是完整目录路径。

（2） 带斜杠转发的是除 hostname 及目录外的所有部分。不能使用正则表达式匹配 location 块，只能使用完整路径名准确匹配。