# Part 1 Summary

## 1.1 前端优化

* 合并css、js，减少Http请求
* 启用压缩，减少数据传输
* 使用浏览器缓存。生成新的js时，只需更新Html中引用。
* CSN加速：缓存静态资源，如图片、文件、CSS、JavaScript脚本、静态网页等。
* 反向代理

## 1.2 应用服务器

系统架构

* 缓存：加快数据读取，降低DB访问压力。网站遇到性能瓶颈，第一个想到的解决方案
* 异步操作：加快请求响应及实现削峰。
* 使用集群：使用负载均衡，有更好的响应延迟，提高吞吐能力。

代码优化

* 多线程,最大发挥CPU和处理IO阻塞。
* 使用NIO(非阻塞，基于事件驱动)
* 资源复用，比如单例和线程池。
* 合理的数据结构
* 减少垃圾回收。
* 尽量不使用锁,减小锁的力度。

## 1.4 Protocol

### HTTP 1.1

**最大并发数限制**

并发指建立的TCP连接数。可以一个连接发送多个请求。Server端按发送顺序处理。

# Part 2 性能优化

## 2.1 sum

### 2.1.1 性能指标

PV QPS TPS

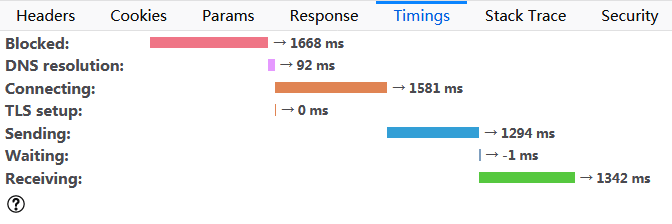
#### Concurrency

并发数。服务器能同时处理的并发TCP连接数。

#### Response

响应时间。发出请求到收到相应数据的时间

##### FireFox



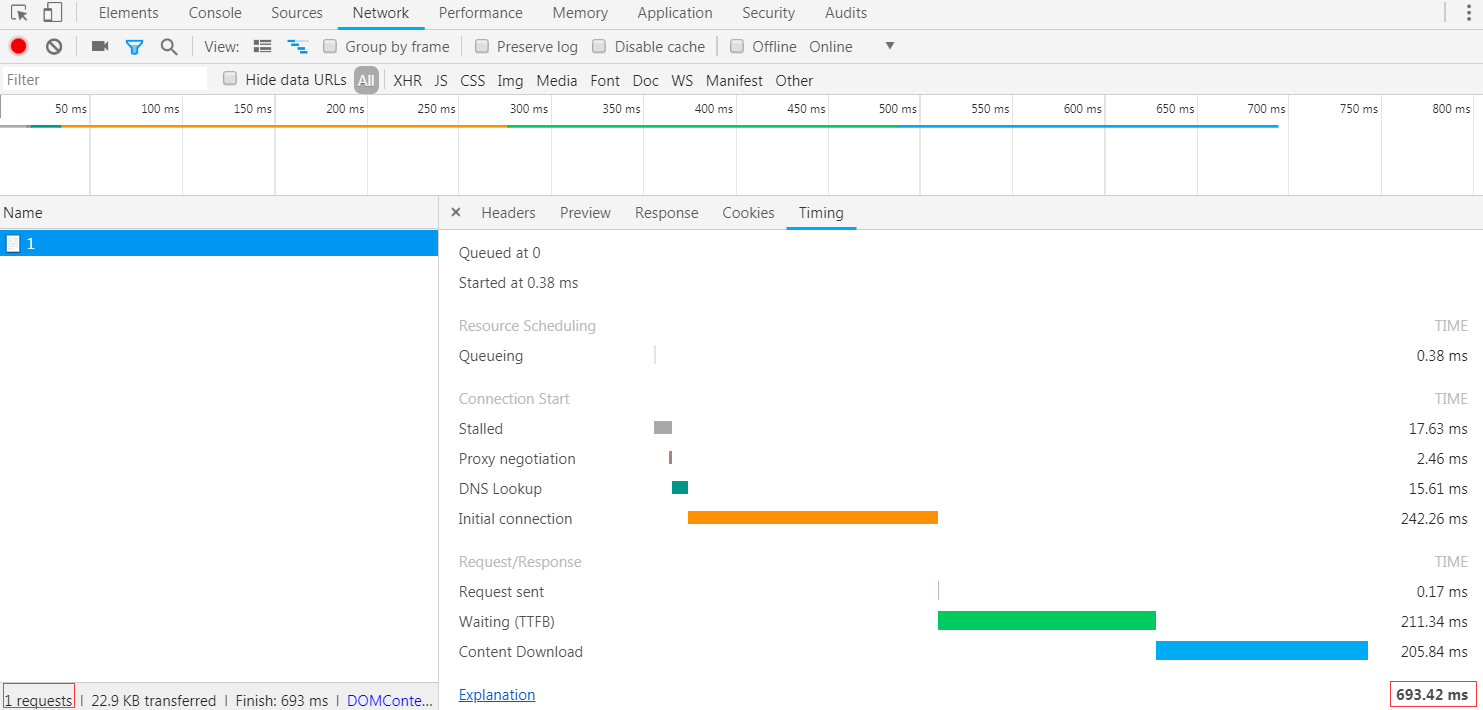
Connecting:TCP建立连接时间

Waiting : Waiting for a response from the server.

Receiving :read entire response from server

##### Chrome

[Reference](https://developers.google.com/web/tools/chrome-devtools/network-performance/understanding-resource-timing)



Queuing：阻塞的原因

Finish： timeline里所有request总时间

DOMContentLoaded ： 加载html dom内容，不包括js，css等

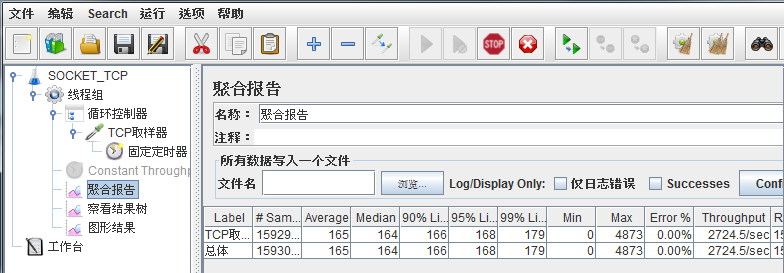
Load： 整个页面的时间

#### 吞吐量

系统单位处理的请求数（QPS）。一个TCP连接可同时发出多个请求，吞吐量是衡量服务器性能的指标。

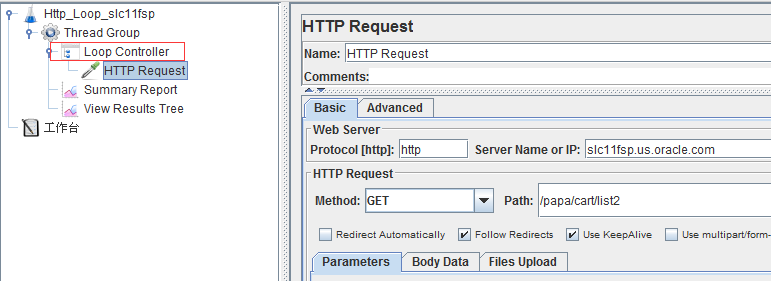
##### JMeter

TCP



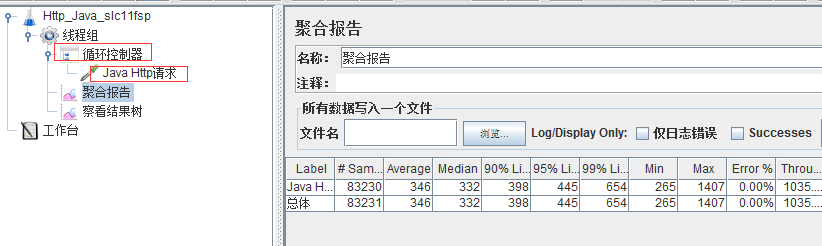
HTTP

短连接



此处是自带的HTTP Request，虽然Loop，但每个request都会重新建立TCP连接，使用完close。

长连接



java中不要close连接

性能计数器

包括System Load、对象和线程数、内存、CPU、磁盘、网络IO

### 2.1.2 性能测试方法

* 性能测试：性能预期
* 负荷测试：找到性能安全临界线
* 压力测试：超过安全符合继续施压，获得最大压力承受能力

压力测试

<https://blog.csdn.net/wyingquan/article/details/414608>

### 2.1.3 测试流程

很多时候在线上出现问题时，往往会伴随着某些指标的异常。大部分情况下，在问题发生之前，某些指标就会提前有异常显示。

step 1 概括

top查看整个系统负荷，正常情况单核负荷在0.7，四核就是3一下。

## 2.2 系统卡顿

系统卡顿，响应慢，往往会伴随一些系统指标异常，第一步应找到系统瓶颈。可能在CPU，IO，也可能在内存。一些常见引起卡顿的原因

* 操作数据库 %idle值高
* 大量计算 %idle值低，
* 大量IO。%idle值高，%iowait值高，详细信息看iostat
* 死锁：大量线程执行同一方法？方法可能有request请求
* 大量频繁线程切换
* 内存：内存分配

定位线程

ps或jps找java进程pid🡪top -Hp $pid H 查看耗cpu时间最多的几个线程🡪线程ID转换成16进制🡪jstack thread dump

### 场景分析

CPU使用率高，响应慢

CPU持续飙高，定位到占用CPU最大的线程分析。可能的原因，

CPU处理能力相对较低/计算量大。

CPU使用率不高，响应慢

特征：%idle值高，dump线程，分析线程stuck原因。

**IO**：%iowait值高。分析IO，详细信息看iostat

**数据库**

**内存**：内存分配

线程切换：

请求无响应

各项指标正常。是否所有的Runnable线程都一直在执行相同的方法，如果是的，恭喜你，锁住了！

## 2.2 线程优化

系统负载高原因

1.线程切换

理想状态CPU，内存，IO外设都满负荷工作，大部分情况瓶颈都在IO的读写部分，多线程适合计算性case。

CPU主频

CPU内核工作的时钟频率，单位之间产生脉冲个数。用于同步指令执行。数值不代表运算速度(和运算速度相关：cache，指令集 ，CPU位数)，但对提高运算速度至关重要。

cat /boot/config-`uname -r` | grep 'CONFIG\_HZ='

cat /proc/interrupts

中断

TODO 操作系统时间片和中断、主频的关系？

#### Thread Dump

tomcat

dump分析发现大量WAITING

### 2.2.1 线程优化

线程切换开销

[Reference](http://ifeve.com/java-context-switch/)

上下文切换会带来直接和间接两种性能开销

**直接消耗**: CPU寄存器需要保存和加载, 系统调度器的代码需要执行, 刷新TLB(TLB：translation lookaside buffer，快表，可以理解为页表缓冲，地址变换高速缓存), CPU 的pipeline需要刷掉;

**间接消耗**：多核的cache之间得共享数据,对于程序的影响要看线程工作区操作数据的大小。

切换时间实测

线程数量

**1) Max Number of threads:**

(MaxProcessMemory - JVMMemory - ReservedOsMemory) / ThreadStackSize

* MaxProcessMemory 指的是一个进程的最大内存。在32位的 windows下是 2G
* JVMMemory         JVM内存
* ReservedOsMemory  保留的操作系统内存
* ThreadStackSize      线程栈的大小(-Xss)

2)线程数量

((CUP时间+CUP等待时间)/CUP时间)\*CUP数量

### 2.2.2 TOMCAT

tomcat的线程优化，本质上就是java 的线程池优化。结合socket特点来优化。实现线程资源和任务的一种平衡。

Server

超过最大连接数，响应时间变长

最大连接数越大，cup load约高

线程数越大，超过500，响应时间变长。线程切换时间？

吞吐量是否应该在服务端测，排除传输时间干扰。

#### 线程池

**thread busy**: jstack pid | grep " RUNNABLE" | wc –l

**waitting**: jstack pid | grep "WAITING" | wc –l

如果tomcat线程池minSpareThreads设置较大，大量线程将parking

#### maxThreads

Tomcat7默认值200。

实际应用中线程真正用于计算的时间可能很少，大多数线程都在等待或阻塞，如等待数据库返回数据、等待硬盘读写数据等。maxThreads应设大点，能支持更改的并发，最大化利用cpu闲置资源。但并不是越大越好，过大CPU会花费大量的时间用于线程的切换，整体效率会降低。

平衡点

只能做性能测试？

#### 设置策略

默认有12个系统线程负责。。。大量的线程用于处理sock，由于web one thread per one requst，看作一个线程和一个sock对应。线程池设置策略看作针对sock的设置

常设的参数

maxThreads：最大的线程数。看作最大连接数。

maxConnections

maxConnections的设置与Tomcat的运行模式有关。如果tomcat使用的是BIO，那么maxConnections的值应该与maxThreads一致；如果tomcat使用的是NIO，那么类似于Tomcat的默认值，maxConnections值应该远大于maxThreads

acceptCount

所有可用的线程数都被使用时，可以放到处理队列中的请求数。默认值10。

通过前面的介绍可以知道，虽然tomcat同时可以处理的连接数目是maxConnections，但服务器中可以同时接收的连接数为maxConnections+acceptCount 。

acceptCount的设置，与应用在连接过高情况下希望做出什么反应有关系。如果设置过大，后面进入的请求等待时间会很长；如果设置过小，后面进入的请求立马返回connection refused

KeepAlive

maxKeepAliveRequests： 保持请求数量，默认值100。

keepAliveTimeout 默认值60000

keepAliveTimeout VS connnectionTimeout

**connnectionTimeout**： 网络连接超时，默认值60000，单位：毫秒。通常可设置为30000毫秒。设置为0表示永不超时，这样设置有隐患的

minSpareThreads：初始创建数。默认25。

maxSpareThreads： max，超过将关闭socket。默认值75。

VS maxThread最大创建数

**acceptorThreadCount**： acceptor数量

bufferSize： 输入流缓冲大小，默认值2048 bytes。

compression： 压缩传输，取值on/off/force，默认值off。 其中和最大连接数相关的参数为

enableLookups： 是否反查域名，默认值为true。为了提高处理能力，应设置为false

server.xml

1. **<Connector** port="8080" protocol="HTTP/1.1"
2. connectionTimeout="20000" maxThreads="1000" minSpareThreads="60"
3. maxSpareThreads="600" acceptCount="120"  redirectPort="8443" URIEncoding="utf-8"**/>**

## 2.3 内存优化

JVM的编译优化

指令重排序，内存栅栏等

#### JAVA OPS优化

**Heap**

* -Xms heap的最小内存；
* -Xmx heap的最大内存；
* -Xmn young的大小，一般设置为Xmx的3、4分之一
* -XX:NewSize: young的大小
* –XX:MaxNewSize
* NewRatio: young:old,2表示1:2,即young/heap=1/5
* -server 启用JDK 的 server 版；一定要作为第一个参数，在多个CPU时性能佳

No Heap

* -XX:MaxNewSize:
* -XX:SurvivorRatio:2\*Survivor:eden,8代表1:1:8
* -XX:+UseParNewGC ：缩短minor收集的时间
* -XX:+UseConcMarkSweepGC ：缩短major收集的时间。此选项在Heap Size 比较大而且Major收集时间较长的情况下使用更合适

**方法区Perm**

* -XX:PermSize=5M -XX:MaxPermSize=7M

**32G 内存配置示例：**

-Xms10g -Xmx10g -Xmn1024m -Xshare:off

**jvm性能调优**

* 控制GC的行为. 频繁GC导致性能下降,因此经常会根据系统运行的程序的特性来更改GC行为
* 调整JVM堆栈大小.xmx xxx
* 控制JVM线程的内存分配.xss

**JVM调试方法：**

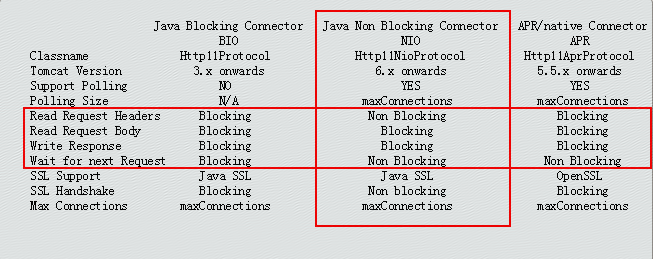
1.Jconsole查看内存各部分变化

2.gc 用jstat

## 2.4 IO优化

配置文件server.xml，略。

IO模型支持



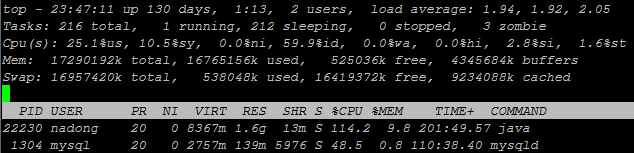
tomcat要实现servlet规范所以不能最大发挥NIO的特性，servlet3.0之前完全是同步阻塞模型，Read http body 以及 response还是同步阻塞，因为servlet规范规定的就是这样。

## 2.5 监控Tools

### 2.5.1 Linux

#### top

动态实时显示。关注%CPU %MEM



**Cpu(s)**

%us user CPU time，这里指整个CPU计算能力的百分比。标题中%CPU指单核的。

%ni：改变过优先级的进程

%id：idle

%wa ：I/O waiting

%hi：硬中断

%si：软中断

%st: steal time

**S**：R=运行 S=睡眠 D=不可中断的睡眠状态T=跟踪/停止 Z=僵尸进程

**快捷键：**P:CPU排序。M:内存排序，H: show thread，1:show cups 列出每个cpu负荷

**Load Average**

一段时间 (1 分钟、5分钟、15分钟) 内平均 Load

系统负荷正常范围：

* 大于0.7，必须开始调查了，问题出在哪里，防止情况恶化。
* 持续大于1.0，必须动手寻找解决办法，把这个值降下来。
* 当系统负荷达到5.0，就表明系统有很严重的问题，长时间没有响应，或者接近死机了。

以上指标基于单CPU的，四核机器的负载最好保持在3(4\*0.7 = 2.8)以下。

#### strace

#### netstat

1.Linux下查看并发连接数，一个浏览器可能建立几个连接，非并发用户数

netstat -na | grep ESTABLISHED | grep 8080 | wc -l

### 2.5.1 Sysstat

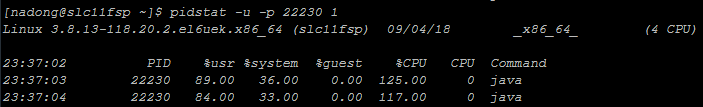
#### pidstat

–p pid.

-u/r/d cpu,内存, io. 默认显示CPU信息.

pidstat 2 10 2s为间隔显示10次。

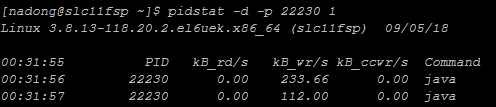
-u



CPU:在那个核运行

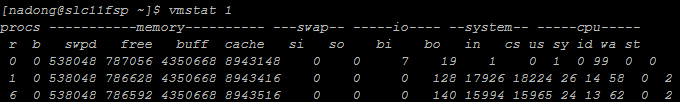
针对单个的线程stat，和top类似

-d



#### vmstat

[Reference](https://linuxtools-rst.readthedocs.io/zh_CN/latest/tool/vmstat.html)



**proces**: r 分配到CPU进程数，b阻塞进程

**memory**: swpd虚拟内存已用大小。

**Swap**: si 每秒从交换区写到内存的大小，so，每秒写入交换区的内存大小

io

单位：blocks/s，Linux块的大小为1M。从内存角度，bi 从block device 输入, bo 输出到device,

system

in中断数/每秒，cs 上下文切换/s。context指process context

**cpu**: us,sy,id,wa

#### iostat

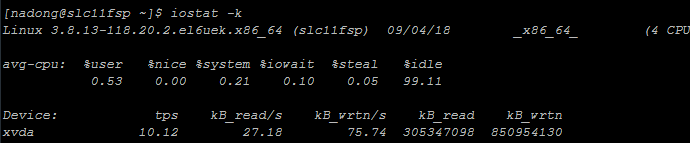
统计CPU和IO。都是针对整个系统的，不能对单个线程统计，区别于top/pidstat。

[Reference](https://linuxtools-rst.readthedocs.io/zh_CN/latest/tool/iostat.html)

-C/d 显示CPU/磁盘 使用情况

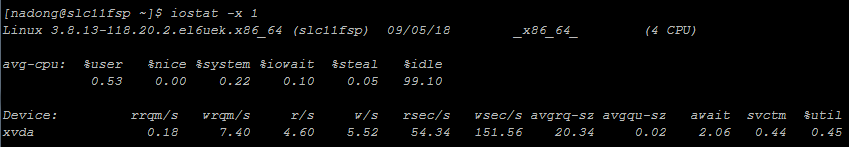
-k/m 以kb/M为单位显示

-d



TPS 每秒I/O请求。可理解为吞吐量

-x 详细信息。



r/s: 每秒完成的读 I/O 设备次数。即 rio/s

w/s: 每秒完成的写 I/O 设备次数。即 wio/s

avgqu-sz: 平均I/O队列长度。

await: 平均每次设备I/O操作的等待时间 (毫秒)。

svctm: 平均每次设备I/O操作的服务时间 (毫秒)。

%io消耗的cpu百分比

#### sar

### 2.5.2 Java

#### jstack

生成线程快照。

[Reference](http://www.hollischuang.com/archives/110)

**用法**：jstack <pid>

**主要目的**：定位线程长时间停顿原因，如线程间死锁、请求外部资源导致的长时间等。

Thread Dump分析

tid线程id，nid:native id，操作系统记录的id，thread state

**RUNNABLE**

locked <地址> (目标)。进入synchronized方法, 申请对象锁成功才会出现此句



BLOCKED(on object monitor)

waiting to lock <地址> (目标)：使用synchronized申请对象锁未成功,在迚入区等待。

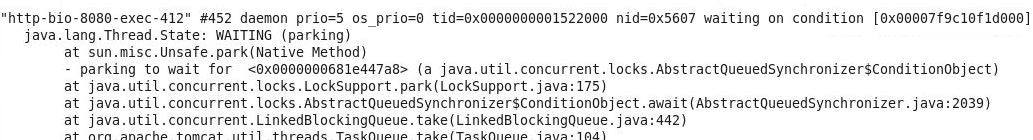


WATTING(on object monitor)

waiting on <地址> (目标)。synchronized修饰的方法调用wait()。Thread释放锁后会**等待锁**



WATTING(parking)

- parking to wait for <地址> (Object)

TIMED\_WAITING(parking)



这里tomcat线程池minSpareThreads设置较大，大量线程将parking

TIMED\_WAITING(sleeping)/ TIMED\_WAITING(on object monitor)

at java.lang.Thread.sleep

**Thread State**

Wait on condition

死锁

http://www.hollischuang.com/archives/2409

http://www.hollischuang.com/archives/110

#### jstat

jstat -option pic < interval >

**gcutil**



S0：年轻代中第一个survivor（幸存区）已使用的占当前容量百分比   
S1：年轻代中第二个survivor（幸存区）已使用的占当前容量百分比   
E：年轻代中Eden（伊甸园）已使用的占当前容量百分比   
O：old代已使用的占当前容量百分比

P：perm代已使用的占当前容量百分比   
YGC：从应用程序启动到采样时年轻代中gc次数   
YGCT：从应用程序启动到采样时年轻代中gc所用时间(s)

FGC：从应用程序启动到采样时old代(全gc)gc次数   
FGCT：从应用程序启动到采样时old代(全gc)gc所用时间(s)   
GCT：从应用程序启动到采样时gc用的总时间(s)   
**gc**



 S0C：年轻代中第一个survivor（幸存区）的容量 (byte)   
 S1C：年轻代中第二个survivor（幸存区）的容量 (byte)   
 S0U：年轻代中第一个survivor（幸存区）目前已使用空间 (byte)   
 S1U：年轻代中第二个survivor（幸存区）目前已使用空间 (byte)   
 EC：年轻代中Eden（伊甸园）的容量 (byte)   
 EU：年轻代中Eden（伊甸园）目前已使用空间 (byte)   
OC：Old代的容量 (byte)   
OU：Old代目前已使用空间 (byte)   
PC：Perm(持久代)的容量 (byte)   
PU：Perm(持久代)目前已使用空间 (字节)

**gcnew**

年轻代对象的信息



**gcold**

 old代对象的信息



**gcoldcapacity**

old代对象的信息及其占用量



OGCMN：old代中初始化(最小)的大小 (字节)   
OGCMX：old代的最大容量 (字节)   
OGC：old代当前新生成的容量 (字节)   
OC：Old代的容量 (byte)

**gcnewcapacity:**



 NGCMN：young初始化(最小)的大小 (字节)   
 NGCMX：年轻代(young)的最大容量 (字节)   
 NGC：年轻代(young)中当前的容量 (字节)

S0CMX：年轻代中第一个survivor（幸存区）的最大容量 (字节)   
S1CMX ：年轻代中第二个survivor（幸存区）的最大容量 (字节)

ECMX：年轻代中Eden（伊甸园）的最大容量 (字节)   
**gcpermcapacity**

 PGCMN：perm代中初始化(最小)的大小 (字节)   
  PGCMX：perm代的最大容量 (字节)     
  PGC：perm代当前新生成的容量 (字节)   
PC：Perm(持久代)的容量 (byte)

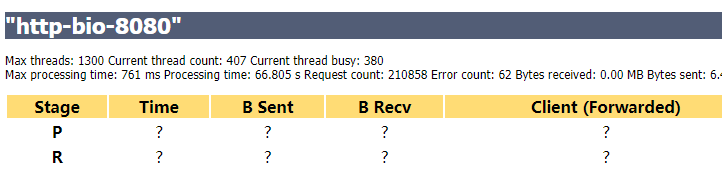
#### jmap

#### Tomcat Mgr

[Reference](https://tomcat.apache.org/tomcat-7.0-doc/manager-howto.html)

Server Status

<http://localhost:8080/manager/status>



Current thread busy: Runnable状态的线程

**Stage**

P: Parse and prepare request 正在处理请求

R: Ready. 线程sleep状态

S: Service

F: Finishing

K: Keepalive

TODO2 linux 命令 tomcat 每次只能accept一个request？

Status

<http://localhost:8080/manager/status?XML=true>

Thread Dump

<http://localhost:8080/manager/text/threaddump>

VM Info

<http://localhost:8080/manager/text/vminfo>

<http://localhost:8080/manager/status?XML=true>

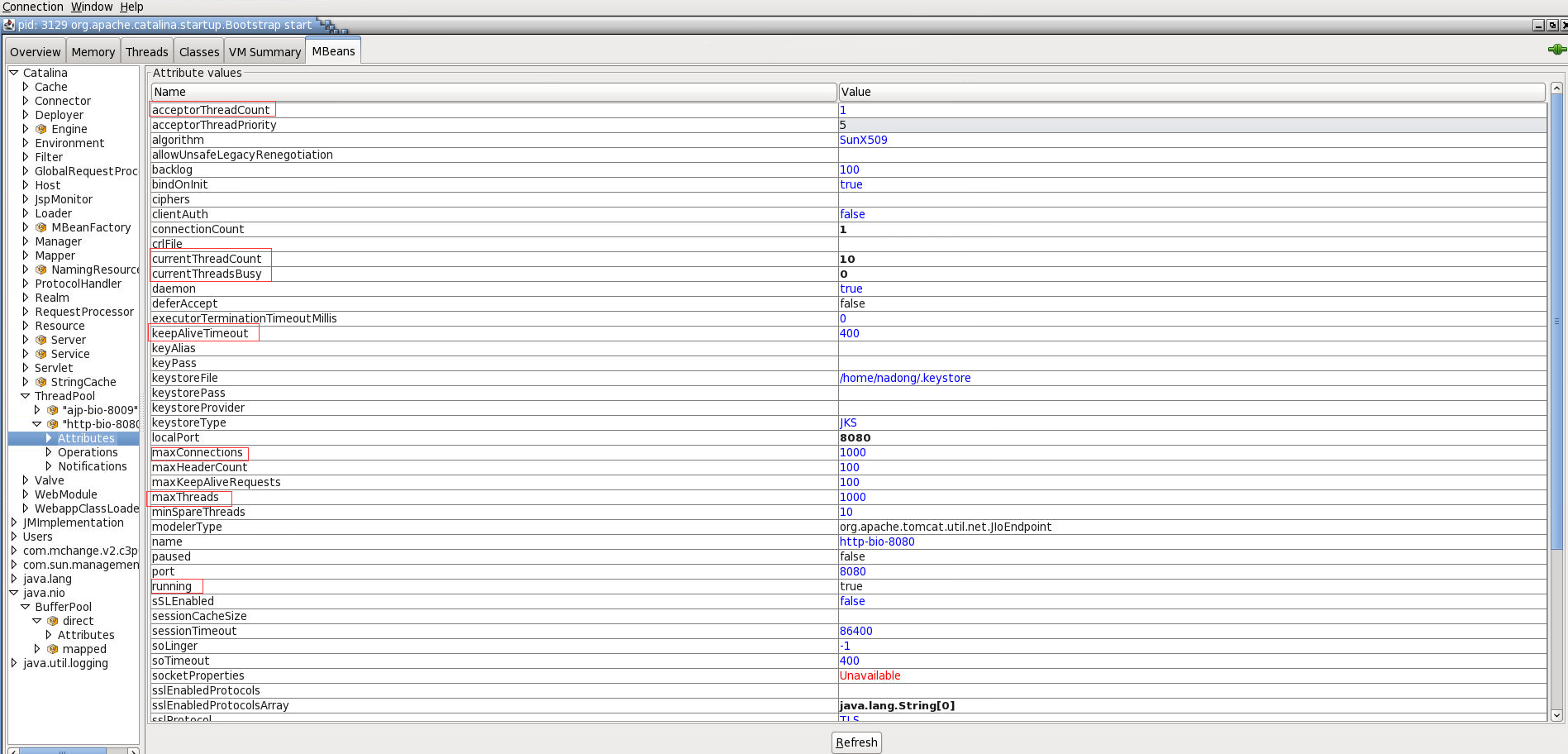
#### jconsole

配置

JAVA\_OPTS

-Djava.rmi.server.hostname=127.0.0.1 -Dcom.sun.management.jmxremote.port=8090 -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false

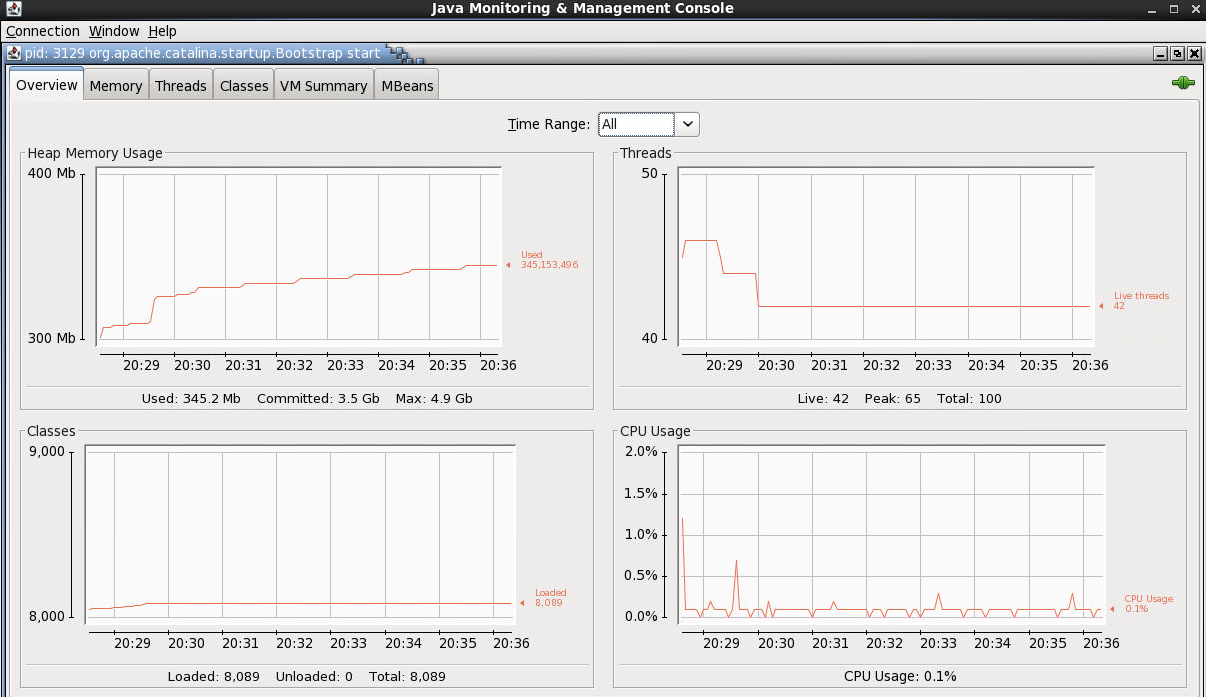
查询Acceptor属性



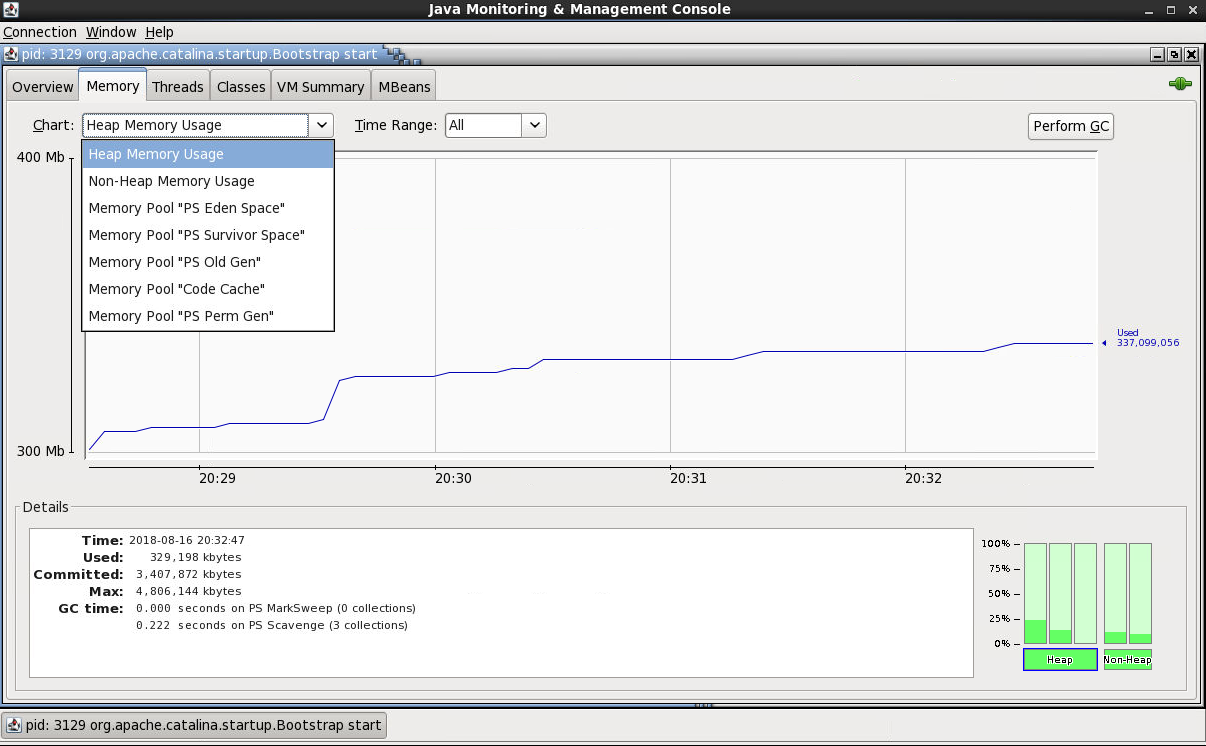
currentThreadCount

currentThreadBusy 线程池

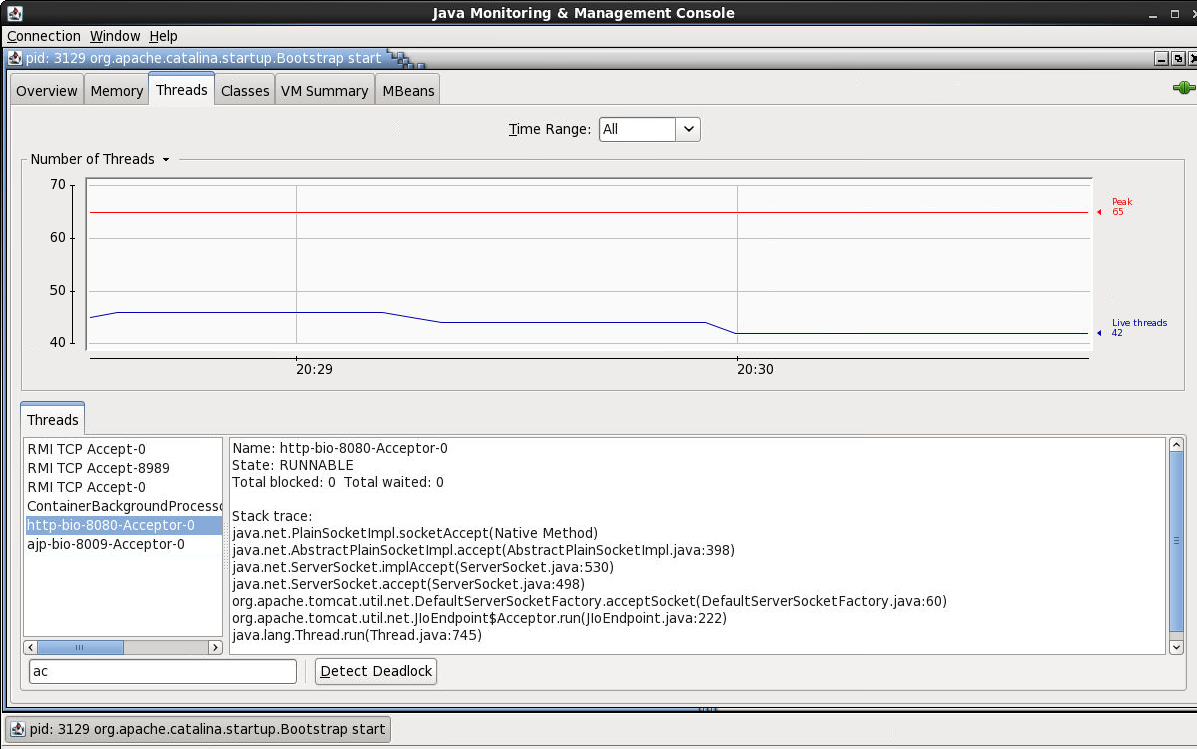
Overview



Memory



Thread



Total blocked

Total waited:

**state**

Wait on condition

#### Jvisual

JDK自带性能检测工具

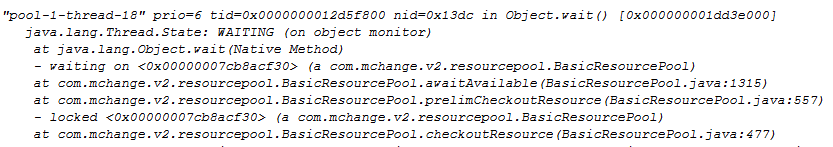


**Running**: thread is still running.   
**Sleeping**: thread is sleeping (method yield () was called on the thread object)  
**Wait**: thread was blocked by a mutex or a barrier, and is waiting for another thread to release the lock

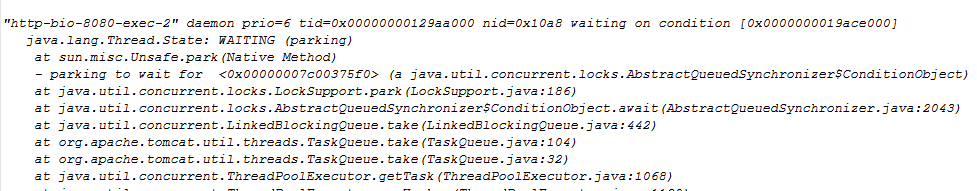
**Park**: parked threads are suspended until they are given a permit. Unparking a thread is usually done by calling method unpark () on the thread object  
**Monitor**: threads are waiting on a condition to become true to resume execution

# Part 3 MYSQL

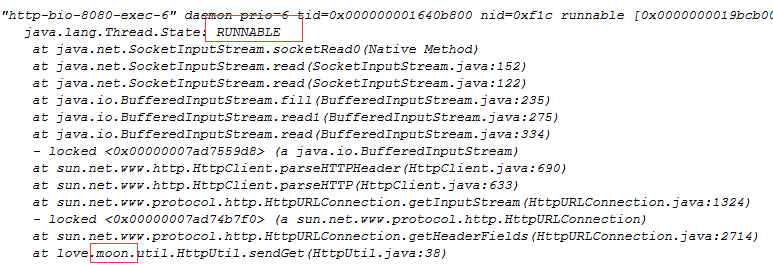
## 日志分析



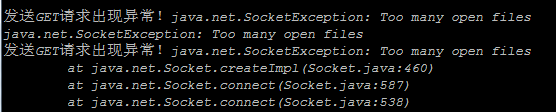
WAITING等待连接池连接。



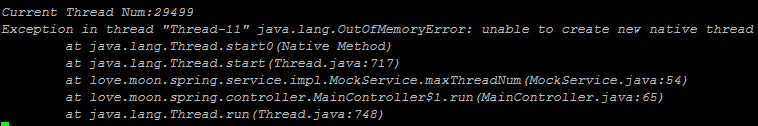
tomcat Acceptor连接后， 线程池启动2个线程，一个read，一个调用业务handler



请求第三方，如果不返回，线程一直RUNNABLE



超过4000个连接后



## 优化Case

1. insert时间长 (远程)

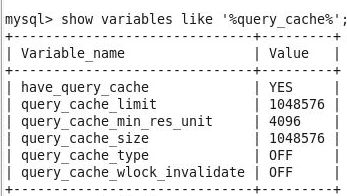
每个请求占用一个request/缓解mysql压力，将数据存入redies。然后批量插入mysql

2．未关闭事务，导致系统无响应。

## Cache

### Query

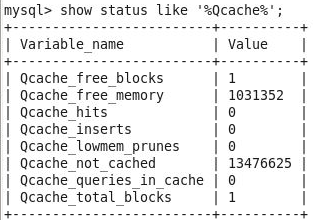
SQL:show variables like '%query\_cache%'



query\_cache\_limit 缓冲区大小，缺省为1M；

query\_cache\_min\_res\_unit缓存块大小，默认4KB，设置值大对大数据查询有好处，但如果你的查询都是小数据 查询，就容易造成内存碎片和浪费；

### Cache Status



Qcache\_free\_blocks：缓存中相邻内存块的个数。数目大说明可能有碎片。FLUSH QUERY CACHE会对缓存中的碎片进行整理，从而得到一个空闲块。

Qcache\_free\_memory：缓存中的空闲内存。

Qcache\_hits：每次查询在缓存中命中时就增大

Qcache\_inserts：每次插入一个查询时就增大。命中次数除以插入次数就是不中比率。

Qcache\_lowmem\_prunes：缓存出现内存不足并且必须要进行清理以便为更多查询提供空间的次数。这个数字最好长时间来看;如果这个 数字在不断增长，就表示可能碎片非常严重，或者内存很少。(上面的 free\_blocks和free\_memory可以告诉您属于哪种情况)

Qcache\_not\_cached：不适合进行缓存的查询的数量，通常是由于这些查询不是 SELECT 语句或者用了now()之类的函数。

Qcache\_queries\_in\_cache：当前缓存的查询(和响应)的数量。

Qcache\_total\_blocks：缓存中块的数量。

## 3.1 Information

连接

mysql –uroot [-h slc11fsp] –p

查看版本mysql -V;



### 概括

查看mysql占用系统资源

top –p pis

pidstat |grep mysql

### information\_schema

show databases/tables;

**Status**：Syntax：show [global/session] status like '%xxx%';

**Variables** :Syntax：show variables like '%max\_connections%';

**Innodb** :Syntax : show [global] status like 'Innodb\_%'

## 统计

### Questions

Slow\_queries： 超过long\_query\_time数量。（show variables like "%long\_query\_time%"）;

Questions ： 发往服务器的查询的数量。统计开始时间点为service start。

DDL统计

Innodb\_rows\_read/inserted/deleted/updated

Innodb\_data\_read/writes

buffer

Threads\_running 不在睡眠的线程数量。   
Uptime 服务器工作了多少秒。

### 数据大小

select data\_length,index\_length from information\_schema.tables where table\_schema='bigdata' and table\_name = 'ld\_cart';

单位是byte，优化显示

select concat(round(sum(data\_length/1024/1024),2),'MB') as data\_length\_MB,concat(round(sum(index\_length/1024/1024),2),'MB') as index\_length\_MB from information\_schema.tables where table\_schema='bigdata' and table\_name = 'fund\_history;

行数select count(1) from ld\_cart;

idex

show index from bigdata.fund;

## 监控

### Connections

show full processlist;



每一条代表一个Thread，一个Tcp连接

Threads\_connected 当前连接数。只有数量值，show processlist有更详细的信息;

connections 试图连接mySQL服务器的次数

Max\_used\_connections

### Lock

Syntax: show status like '%Innodb\_row\_lock%';



Innodb\_row\_lock\_current\_waits 当前阻塞线程数

Innodb\_row\_lock\_waits 总共wait次数

2.select \* from information\_schema.innodb\_locks;

查看lock具体数据



lock\_trx\_id找到事务

lock\_table+lock\_data找到具体锁定的数据。

Lock\_mode: X/S

lock\_type： RECORD行锁。

Timeout

show variables like 'innodb\_lock\_wait\_timeout'; 等待获取锁

### Transaction

开启事务 start transaction;

select \* FROM information\_schema.innodb\_trx\G;



trx\_id

trx\_state: RUNNING/LOCK WAIT

trx\_mysql\_thread\_id

trx\_stared 事务开始时间

trx\_wait\_stated:block事务wait stated

trx\_query:

trx\_rows\_locked 锁定row行数

trx\_tables\_locked 锁定table数量

Autocommit



show variables like 'autocommit';

set autocommit = 0;仅对当前连接生效

autocommit VS start transaction

[Reference](https://blog.csdn.net/qq_26941173/article/details/77872038)

set autocommit=0 所有语句自动开启事务，commit结束事务。  
start transaction 启动一个新事务。语句"挂起"自动提交模式

ReadOnly

查看 show global variables like "%read\_only%";

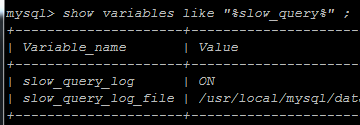
设置

## 日志

show variables like 'log%';

### show query

show variables like "%slow\_query%" ;



开启日志

set global slow\_query\_log=ON;

分析

mysqldumpslow /usr/local/mysql/data/mysql/slc11fsp-slow.log