服务器端性能优化-提升QPS、RT

关注：CPU、内存、线程

业务平台 小邪

1、找到优化的方向
2、QPS/RT---线程(CPU/IO)的关系
3、最佳线程数
4、优化案例说明
5、找到瓶颈
6、线程本身的开销？什么时候需要我们关注，多线程的切换，线程本身占用的资源，以及线程的资源开销
7、内存瓶颈（FULL GC的停顿）
8、案例说明
9、内存优化方向

QPS：Query-per-second，1秒钟内完成的请求数量
RT：Response-time，1个请求完成的时间

QPS提升带来什么？
1.单台服务器资源的充分利用
2.QPS提升1倍，服务器资源减少1半

Detail 现在有236台 ShopSystem 115台，QPS提升一倍，则Detail只要118台机器，ShopSystem只要58台机器，或者说未来咱们淘宝的流量增加了一倍，detail和shopsystem的机器数量可以保持不变。

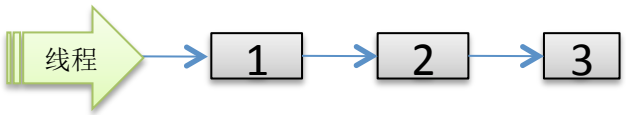
RT提升带来什么？
1.提高响应速度，提升用户的体验
2.反过来也会提升QPS

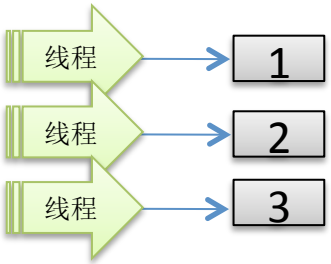
前言：

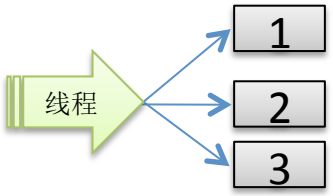
做hesper优化期间，发现一个有趣的事情，当时我们一伙人列出了很多优化点，有节省内存的，有节省CPU的，有节省IO时间的。性能测试过程中，发现响应时间提升非常大，从原来的200毫秒提升到了100ms，大喜。

总结一下有两个关键的改进：

1、多次搜索请求采用了异步IO，串行改并行







2、QP的查询结果做缓存

但是性能压测的结果QPS却提升很少：45提升到49，为什么？

继续。。。

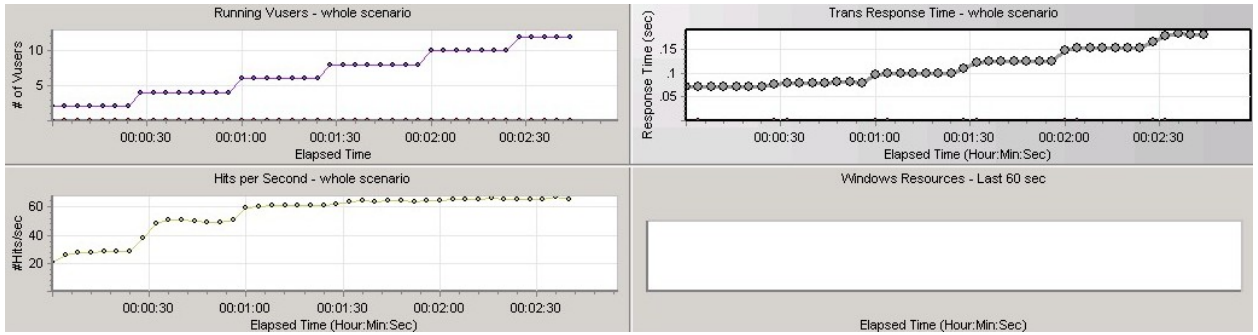
1、然后删除掉searchAuction.vm里面的所有模板代码，压测QPS几乎没有变化？

2、增加压测的用户数，发现QPS从49提升到了190？但是响应时间几乎没有变化，还是100ms左右，为什么？

QPS和线程数的关系

1、在性能压测刚开始之前，QPS和线程是互相递增的关系，线程数到了一定数量之后，QPS持平，不在上升，并且随着线程数量的增加，QPS开始略有下降，同时响应时间开始持续上升。

图1（hesper压测）：



2、单线程的QPS公式，QPS=1000/RT

同一个系统而言，支持的线程数越多，QPS越高，而能够支持线程数量的两个因素是，CPU数量，和线程等待

（对于单线程，公式QPS=1000/RT永远是正确的，所以线程能支持的越多，QPS越高）

最佳线程数量

定义：刚好消耗完服务器的瓶颈资源的临界线程数

公式1：最佳线程数量=((线程等待时间+线程cpu时间)/线程cpu时间) * cpu数量

特性：

1、在达到最佳线程数的时候，线程数量继续递增，则QPS不变，而响应时间变长，持续递增线程数量，则QPS开始下降

2、每个系统都有其最佳线程数量，但是不同状态下，最佳线程数量是会变化的

3、瓶颈资源可以是CPU，可以是内存，可以是锁资源，IO资源

最佳线程数的获取

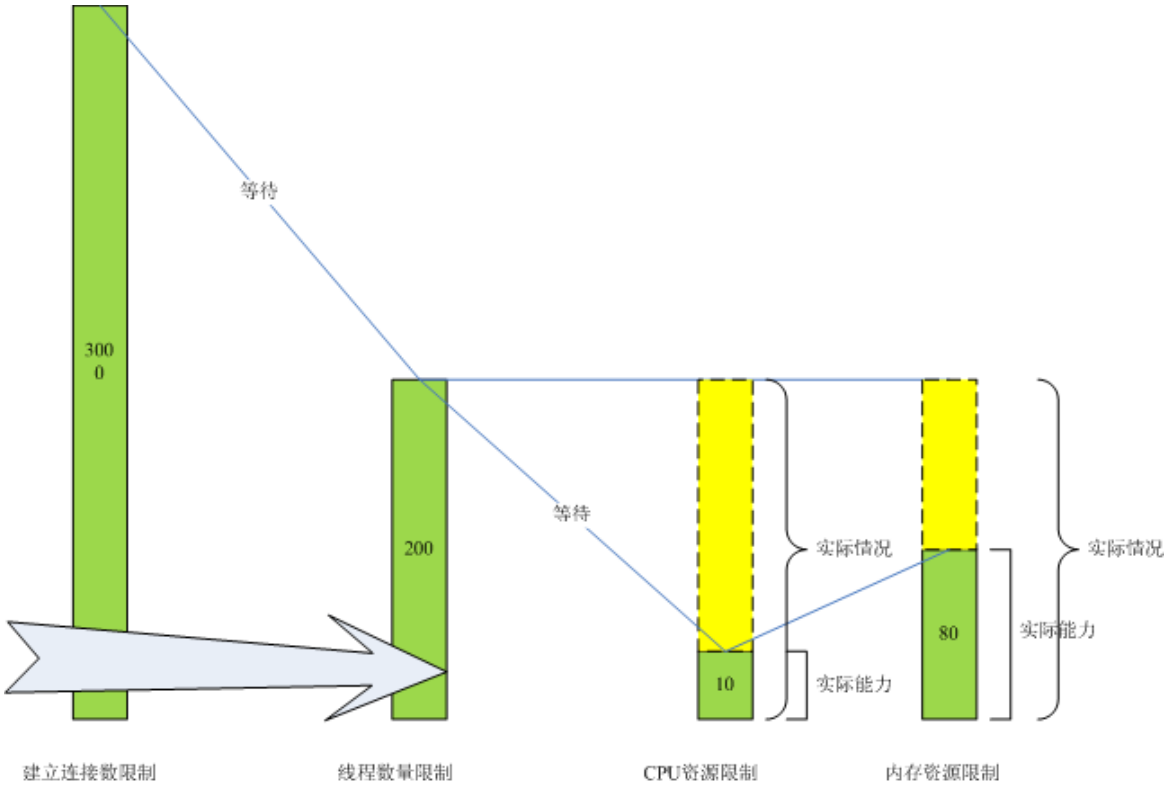
1、通过用户慢慢递增来进行性能压测 ，观察压测结果，按照定义很容易获得最佳线程数量
2、根据公式计算:服务器端最佳线程数量=((线程等待时间+线程cpu时间)/线程cpu时间) * cpu数量
3、单用户压测，查看CPU的消耗，然后直接乘以百分比，再进行压测，一般这个值的附近略作调整应该很容易得到最佳线程数量。

案例

超过最佳线程数-导致资源的竞争

以detail系统为例子，我们的apache连接数允许3000个，tomcat线程数允许200个，而CPU能支持的最佳线程数只有10个左右（CPU 85%以上），内存限制的线程数60个(否则Full GC频繁)。

可是CPU不认为自己只有处理10个线程的水平，它会认为自己有处理200个线程的能力，这是一个问题。这个问题存在可能使内存资源成为瓶颈。



超过最佳线程数，响应时间递增

观点：线程多时间消耗长，并不是说我们的代码执行效率下降了，而是资源的竞争，导致线程等待的时间上升了

公式2：平均响应时间 =（并发线程数/最佳线程数） * 最佳线程数的响应时间

例子：超过最佳线程数量之和，线程数量翻倍，响应时间翻倍，QPS不变：下面例子的QPS在1100左右，每次都在这个值左右，统一使用了1100

- ✓2个并发，每个申请1M内存，测试结果，qps在1100左右，rt在1.911ms左右

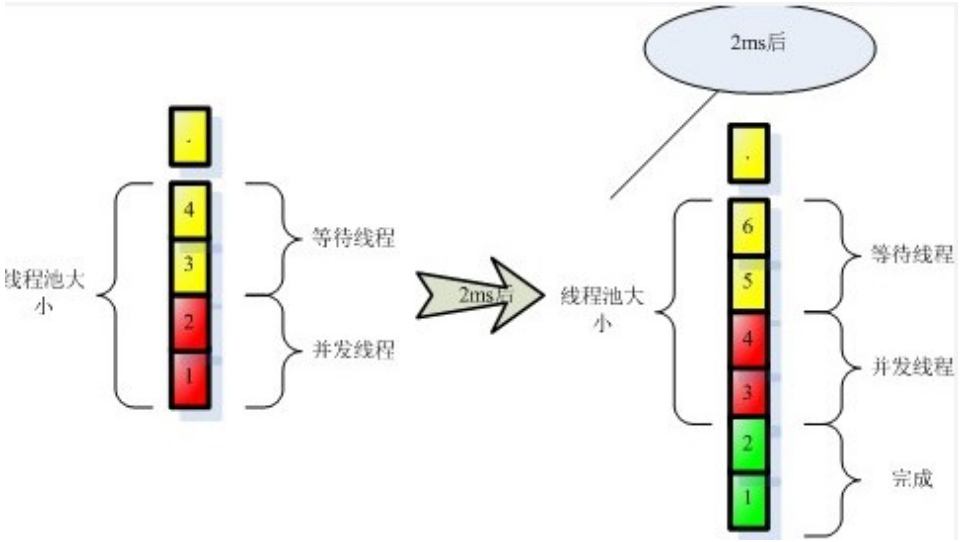
  ab -n10000 -c2 http://192.168.211.1:8080/test.jsp?thm=1024000

- ✓100个并发，每个申请1M内存，测试结果，qps在1100左右，rt在89ms左右

  ab -n10000 -c100 http://192.168.211.1:8080/test.jsp?thm=1024000

- ✓200个并发，每个申请1M内存，测试结果，qps在1100左右，rt在170ms左右

  ab -n100000 -c200 http://192.168.211.1:8080/test.jsp?thm=1024000



最佳线程数是在不断变化的

对一个高IO的系统而言，假设CPU时间10ms + IO 时间40ms =总时间 50ms

如果CPU被优化成了5ms，实际总的时间是45ms

响应时间从50ms减少到了45ms，成总体时间上看变化不大，单线程的qps从20提升到22也并不明显，但是实际上由于CPU时间减少了一半，线程数量几乎可以翻番，QPS也几乎可以提升1倍

反过来如果由于IO时间从40ms变成了80ms，则总时间变成了90ms，总体时间从50ms提升到了90ms，此时最佳线程数量几乎翻倍，QPS几乎没有什么变化。

我们的系统由于依赖的系统响应时间发生变化，会导致系统本身最佳线程数量的变化，这给系统线程资源的分配造成了难度

QPS和响应时间RT的关系

1、对于大部分的web系统，响应时间RT一般由CPU执行时间，线程等待时间（IO等待，sleep，wait）时间组成。如，hesper由 io-cpu-io-wait-cpu-io组成
表明上看，QPS和RT应该是反比例关系？满足，公式3：QPS=1000ms/RT
如图1：

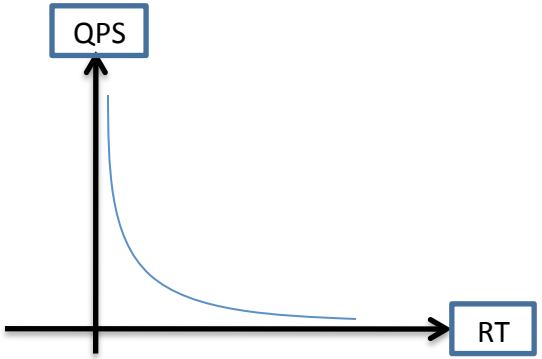


图1 单线程 or 纯CPU多线程

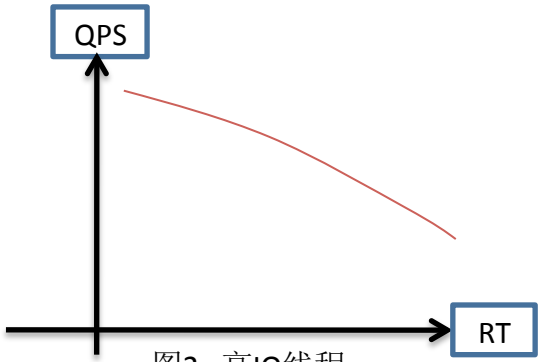


图2 高IO线程

2、 在实际环境中RT和QPS并不是非常直接的反比例关系

实际测试发现，RT的两个时间对QPS的影响并不一样。CPU的执行时间减少，对QPS有实质的提升（线性），IO时间的减少，对QPS提升不明显。所以在高IO的系统，QPS和RT的关系如图2

验证案例

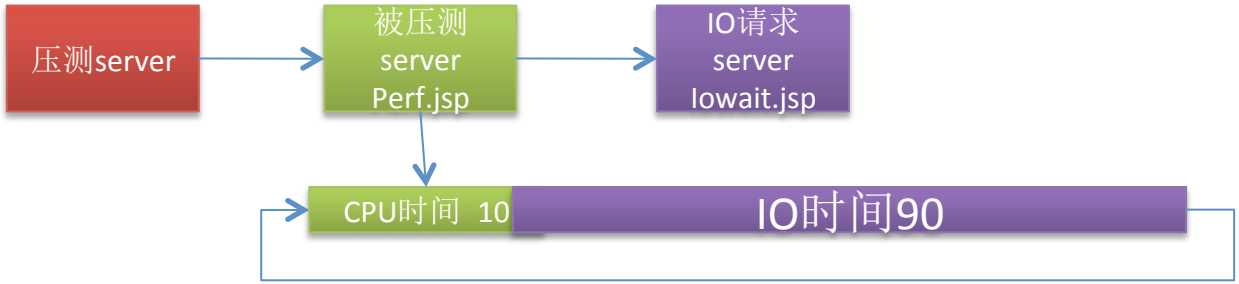
下面案例用来说明3个事情

1、最佳线程数经常变化

2、CPU的优化直接对QPS贡献

3、远程服务时间的缩短对RT贡献最大

案例
----关注QPS-RT的变化



例子简介：通过CPU和IO的测试模型，观察CPU和IO时间消耗变化，而带来的QPS和RT的变化

1、准备资源

3台linux服务器（保证服务器的资源不会互相干扰，服务器硬件情况）
a、压测服务器（安装了AB 工具） 4CPU
b、被压测的主程序部署的服务器 （安装了tomcat和jdk ） 2 CPU
c、主程序服务器会有IO请求到这台服务器 （安装了tomcat和jdk ）2CPU

2个jsp页面，分别是**perf.jsp**和**iowait.jsp** （perf.war）

Perf.jsp

```
<%
        // CPU执行时间
        long t1 = start;
          String cpucount = request.getParameter("cpuc");
          int count = Integer.parseInt(cpucount);

              int n = 0;
              double d = 0;
              long l = 0;
              for (int i = 0; i < count; i++) {
                    for(long b = 0; b < 100; b++) {
                          n++; d++; ++; n = n * 2; = d / 1;  n -=
          1; l --; d = l + n; d = l + n; d ++;
                    }
                    d ++; l ++; n ++;
              }
              out.println(n); out.println(d); out.println(l);
             long t2 = System.currentTimeMillis();
          out.println("cpu time:"); out.println((t2 - t1));
          out.println("ms<br>");
%>
```

```
<%
        // IO 时间
        long t3 = System.currentTimeMillis();
          String iosleep = request.getParameter("ios");
          HttpClient httpclient = new DefaultHttpClient();
          HttpGet httpget = new HttpGet("http://localhost/
perf/iowait.jsp?sleep=" + iosleep);
          HttpResponse response1 = httpclient.execute(httpget);
          HttpEntity entity = response1.getEntity();
          if (entity != null) {
             InputStream instream = entity.getContent();
             int s;
             byte[] tmp = new byte[2048];
             while ((s = instream.read(tmp)) != -1) {
             }
             instream.close();
          }
          httpclient.getConnectionManager().shutdown();
          long t4 = System.currentTimeMillis();
          out.println("io time:");
          out.println((t4 - t3));
          out.println("ms<br>");
%>
```

iowait.jsp

```
<%
// CPU sleep
%>

<%
		long t1 = System.currentTimeMillis();
		String sleep = request.getParameter("sleep");
		int sleepTime = Integer.parseInt(sleep);
		Thread.sleep(sleepTime);
		long t2 = System.currentTimeMillis();
		out.println("Thread sleep:");
		out.println((t2 - t1));
		out.println("ms<br>");
%>
```

硬件配置情况

机器1： 处理perf.jsp页面
CPU 2个
CPU参数
model name : Intel(R) Xeon(R) CPU 5120 @ 1.86GHz
stepping : 6
cpu MHz : 1862.650
cache size : 4096 KB

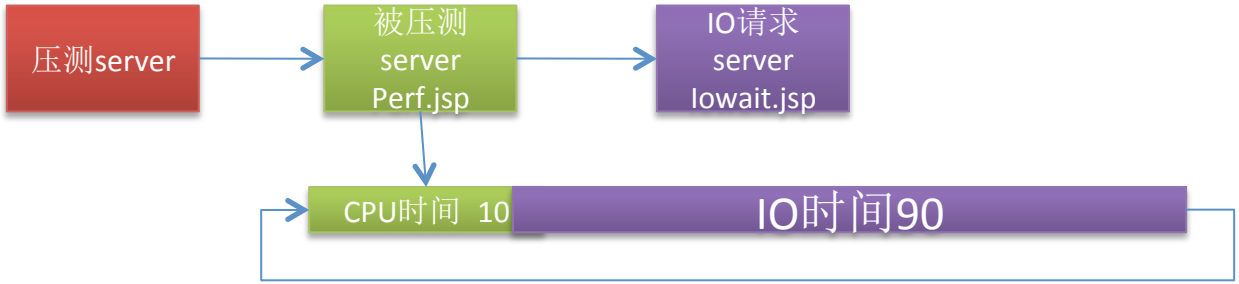
tomcat配置了jvm的参数
JAVA_OPTS="-Xms256m -Xmx256m -Xss1024K -Djava.library.path=/root/jnicache/cachemap/cachemap-0.1/src"
其他默认配置

机器2：处理iowait.jsp页面
2个CPU
model name : Intel(R) Xeon(R) CPU E5320 @ 1.86GHz
stepping : 11
cpu MHz : 1861.916
cache size : 64 KB
tomcat配置了jvm参数
JAVA_OPTS="-Xms256m -Xmx256m -Xss1024K -Djava.library.path=/root/jnicache/cachemap/cachemap-0.1/src"
其他默认配置

性能压测机器
CPU 4个
CPU参数：
model name : Intel(R) Xeon(TM) CPU 3.20GHz
stepping : 10
cpu MHz : 3192.783
cache size : 2048 KB

测试工具：AB

案例-基准测试



1、进行ab压测

ab -n10000 -c20 http://192.168.211.1:8080/perf/perf.jsp?cpuc=30000&ios=80

单次请求：cpu time: 10 ms 、io time: 86 ms、total: 96ms

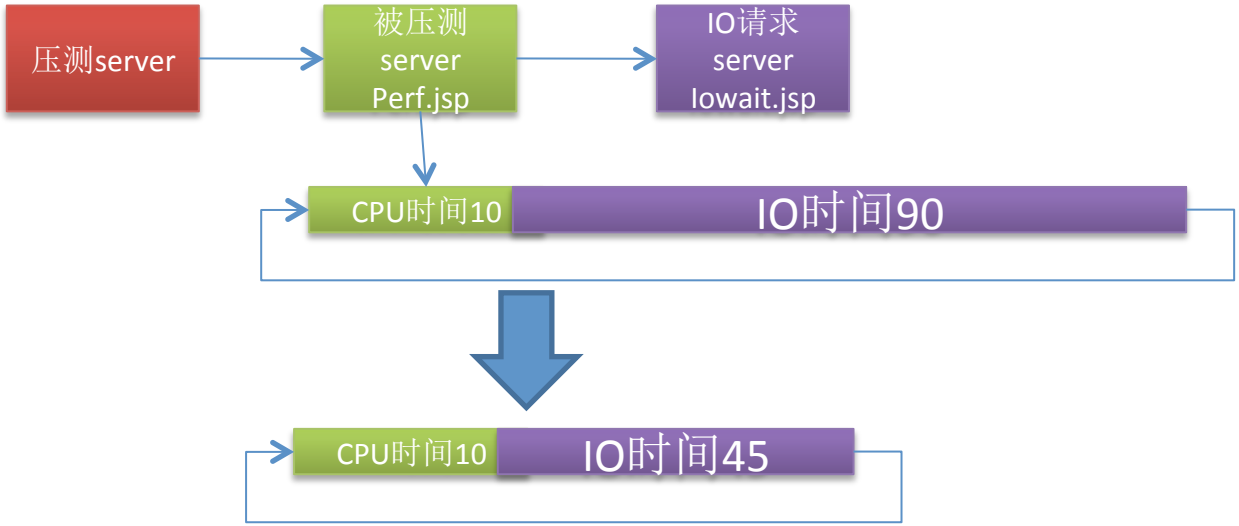
Ab压测结果：**QPS=174** ， RT = 114ms， 服务器CPU 85%， 最佳线程数20

将以上数据作为基准数据，后面的例子均以这个基准展开

2、根据压测结果，如果要把QPS从174提升到300，我们需要怎么做？

接下去做两个例子，1，将IO时间从90ms改成45ms，2、CPU时间从10ms改成5ms

案例-提升RT能提升QPS？



1、减少IO时间：从90ms减少到45ms

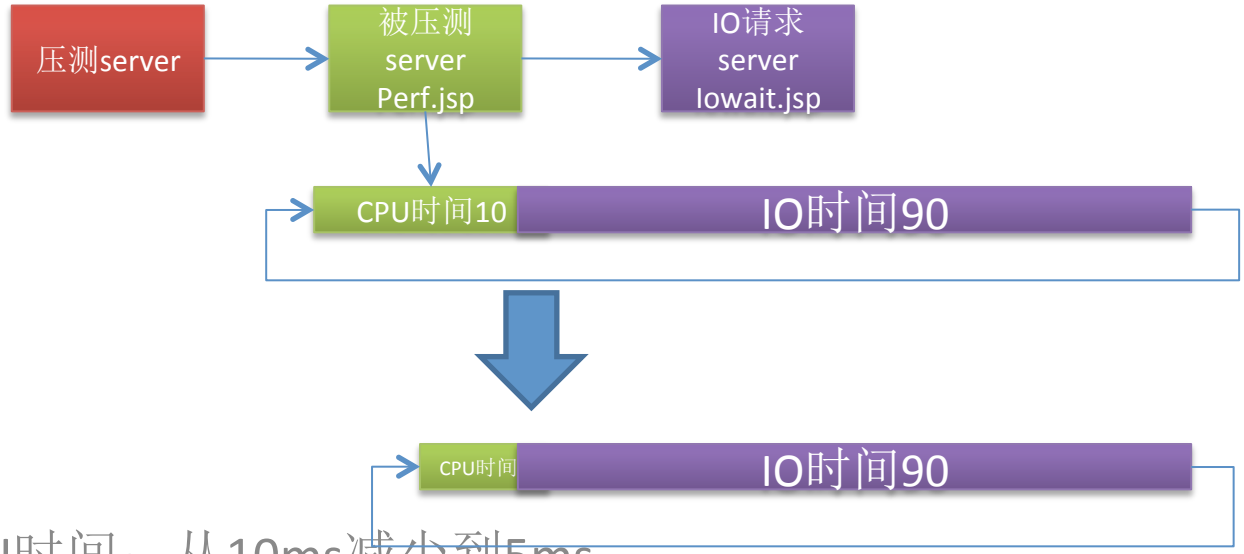
2、进行ab压测

ab -n10000 -c13
http://192.168.211.1:8080/perf/perf.jsp?cpuc=30000&ios=40

单次请求：cpu time: 10 ms 、io time: 45ms、total: 55ms

Ab压测结果：**QPS=176** ， RT = 74ms， 服务器CPU 85%， 最佳线程数13 **50用户的时候**

案例-提升CPU时间能提升QPS？



1、减少CPU时间：从10ms减少到5ms

2、进行ab压测

ab -n10000 -c40
http://192.168.211.1:8080/perf/perf.jsp?cpuc=15000&ios=80

单次请求：cpu time: 5ms 、io time: 85 ms、total: 90ms

Ab压测结果：**QPS=338** ， RT = 118ms， 服务器CPU 85%， 最佳线程数40

案例-压测结果汇总

AB测试，URL： ab -n10000 -c10 http://192.168.211.1:8080/perf/perf.jsp?cpuc=?&ios=?

大致时间消耗	AB 压测URL的参数	最佳线程数	QPS	RT	服务器CPU
CPU:10 ms IO:80 ms	perf.jsp?cpuc=30000&ios=80	20	175	114	85%~
CPU:5 ms IO:80 ms	perf.jsp?cpuc=15000&ios=80	40	339	118	85%~
CPU:10 ms IO：40 ms	perf.jsp?cpuc=30000&ios=40	13	176	73	85%~

- 减少响应时间，并不能有效的提升QPS
- 通过这个例子，有几点可以明确
  1、如果要提升服务器端的响应时间RT，采用减少IO的时间能达到最佳效果，比如合并多个IO请求
  2、如果要提升QPS，采用优化CPU的时间能达到最佳效果
  3、但是并不绝对，但是足以证明hepser优化的例子

前言解释：

做hesper优化期间，发现一个有趣的事情，当时我们一伙人列出了很多优化点，有节省内存的，有节省CPU的，有节省IO时间的。性能测试过程中，发现响应时间提升非常大，从原来的200毫秒提升到了100ms，大喜。

总结一下有两个关键的改进：

1、多次搜索请求采用了异步IO，串行改并行，画个图

2、QP的查询结果做缓存

但是性能压测的结果QPS提升很少：45提升到49，为什么？

因为IO并不是瓶颈资源，CPU才是瓶颈资源，减少的IO时间并不能使CPU时间增加，所以瓶颈依旧没有解决，QPS变化很少。

继续。。。

1、然后删除掉searchAuction.vm里面的所有模板代码，压测QPS几乎没有变化？响应时间略有减少，是90ms左右？

总QPS=线程数*单个线程的QPS，因为压测的时候没有改变用户数量所以线程数没有变化，而单个线程的QPS=1000ms/rt，显然QPS不变。但是此时如果细心你会发现系统的CPU消耗很低。

2、增加压测的用户数，发现QPS从49提升到了190？响应时间几乎没有变化，还是100ms左右？

QPS提升得益于模板CPU资源的释放，这里也说明了模板消耗了60%以上的CPU。

线程本身是否会影响QPS

通过之前的例子看到，如果线程的wait之间变长，则最佳线程数量也会变多，最佳线程数在多大的时候，线程本身的资源开销也需要被考虑。

首先，操作系统线程是一种资源，数量是有限制的，32位操作系统分配给一个进行的用户内存是2G，系统本身使用2G，所以jvm允许创建线程的最大值应该是 小于2G/256K=8000，在linux服务器上测试发现默认允许创建7409个线程 （-Xss的值会决定能创建多少线程)，如果设置-Xss128K，则可以最大创建14349个线程，设置-Xss1M，则最大可以创建1896个线程

超过这些线程数量的极限值，则抛出：

java.lang.OutOfMemoryError: unable to create new native thread

at java.lang.Thread.start0(Native Method)

其次，线程越多消耗内存越多，过多的线程直接将系统内存消耗殆尽。

其次，线程数量如果达不到500则不要太过于纠结，因为线程本身消耗的内存是os级别的内存，而非进程的用户内存，1000以下的线程对我们的性能几乎没有任何影响，当然前提是你需要这么多的线程来支撑你的业务。

Linux下，Tomcat 500个线程 同 4个线程对同一个url进行压测，线程本身消耗资源对qps的影响几乎是可以忽略不计的。如果你要维护上万以上的用户长连则需要重点关注线程本身的开销。

ab -n100000 -c850 http://192.168.211.1:8080/test.jsp?delay=1

Requests per second: 12692.86 [#/sec] (mean)

Time per request: 66.967 [ms] (mean)

ab -n100000 -c50 http://192.168.211.1:8080/test.jsp?delay=1

Requests per second: 13592.44 [#/sec] (mean)

Time per request: 3.679 [ms] (mean)

总结：CPU瓶颈下的QPS计算

对一个高IO的系统而言，假设CPU时间10ms + IO 时间40ms =总时间 50ms
如果CPU被优化成了5ms，实际总的时间是45ms
响应时间从50ms减少到了45ms，变化不大，单线程的qps从20提升到22也并不明显
CPU在5个线程时候达到了100%，根据CPU资源恒定原则：
1、CPU的时间从10ms优化到5ms

线程数	单线程QPS	RT	CPU处理时间（每秒）	QPS
5	20	50	10ms*5*20=1000	5*20=100
X=9.1	22	45	5ms*x*22=1000	9.1 *22 = 200.2

2、IO的时间从40ms优化到20ms

线程数	单线程QPS	RT	CPU处理时间（每秒）	QPS
5	20	50	10ms*5*20=1000	5*20=100
X=3	33.3	30	10ms*x*33.3=1000	33.3*3=100

总结：两种极端的应用

1、proxy应用（耗IO的）线程越多越好，当线程达到过多时线程本身的开销也会成为瓶颈，线程本身也是一个资源。所以这类应用一般采用轻程模型，NIO解决，如nginx

2、计算型应用（耗CPU的），线程数量就是CPU的数量。如搜索索引服务器

耗时热点查找工具

1、jprof

2、jprof.py

3、TimeFilter

4、visualVM、jprofiler、 YourKit

怎么来提升RT

1、减少IO的响应时间

减少IO的调用次数

并发HTTP请求（无上下文依赖，多个连接，一个线程）

HTTP连接池（长连接）

2、减少CPU的使用时间

forest循环的例子

怎么来提升QPS

1、减少CPU的使用时间

2、增加CPU的数量

3、减少同步锁

如果CPU不能被压到85%以上，并且此时的QPS已经达到了峰值，则说明另有瓶颈，接下去关注内存

内存是否是瓶颈

判断依据？

在最佳线程数量*5~6的情况下，进行压测，Old区内存增长是否正常。（性能压测要关注使用了多少用户数，目前我们的压测方式容易遗漏内存瓶颈。）

堆内存的结构

jvm的堆分为3个部分，young和Old区是我们需要重点关注的

✓young（eden+2survivor）

所有对象的创建都是在Eden区完成的，Eden满了之后会进行minorGC，将不能回收的对象放入到survivor区。

young区通过 -XX:NewRatio=n或者-XX:NewSize=n或者–Xmn设置

Survivor通过-XX:SurvivorRatio设置

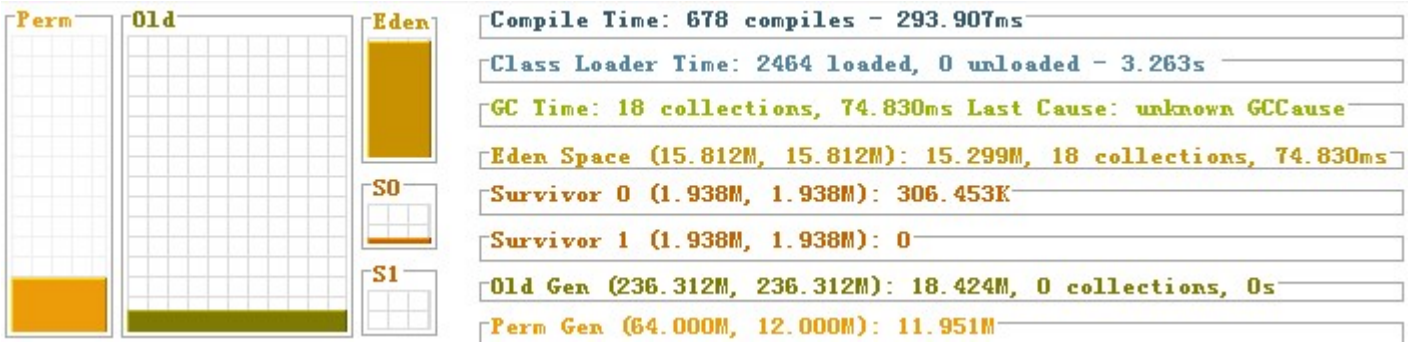
http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html

✓Old

survivor区满了之后，或者对象已经足够的old，则放入Old区，这个行为也是由minorGC触发

✓Perm

主要存放类的一些数据，类的频繁创建会导致Perm OOM （不属于-Xms的设置的空间）



堆内存的分配和回收步骤

1、对象在Eden区完成内存分配， String str = new String(“helloWorld”)

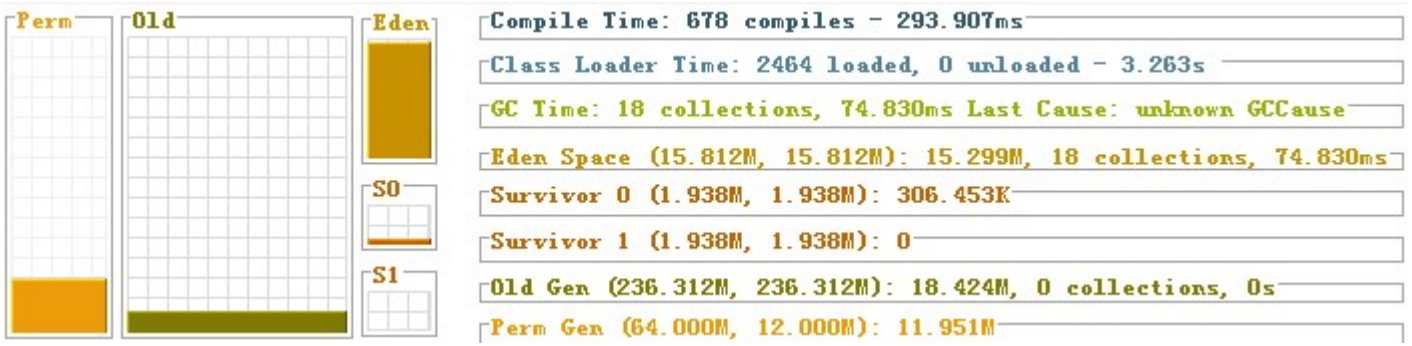
2、当Eden区满了，再创建对象，会因为申请不到空间，触发minorGC，进行young(eden+1survivor)区的垃圾回收

3、minorGC时，Eden不能被回收的对象被放入到空的survivor（Eden肯定会被清空），另一个survivor里不能被GC回收的对象也会被放入这个survivor，始终保证一个survivor是空的

4、当做第3步的时候，如果发现survivor满了，则这些对象被copy到old区，或者survivor并没有满，但是有些对象已经足够Old，也被放入Old区

XX:MaxTenuringThreshold

5、当Old区被放满的之后，进行完整的垃圾回收



堆内存的分配

下面用例子来说明jvm，堆得分配和回收：

memory2.jsp 这个jsp页面主要包含2个功能，1，申请指定大小的内存 2，线程等待指定的时间

```
<%
// 申请内存
%>
<%

                long start = System.currentTimeMillis();
                String msize = request.getParameter("m");
                int m = Integer.parseInt(msize);
                // 申请内存
                int size = 1024 * 1024 * m; // 1M byte
                byte[] allocate = new byte[size];
%>

<%
// Thread sleep
%>
<%
                String sleep = request.getParameter("s");
                int s = Integer.parseInt(sleep);
                Thread.sleep(s);
%>
<%
                // 使用申请的内存
                out.println(allocate);
%>
<%
// 输出耗时

                long end = System.currentTimeMillis();
                out.println("elapsed:");
                out.println(end - start);
                out.println("ms<br>");
%>
```

堆内存的分配

例子1：单线程情况下，只要eden区的内存大小 > 线程每次申请的内存大小

tomcat中jvm的配置：JAVA_OPTS=-Xms256m -Xmx256m -Xss128k -XX:MaxTenuringThreshold=15

堆内存分布：Eden 15.812M　S0 S1 1.938M　Old 236.312M

AB请求：ab -n1000 -c1 "http://localhost/perf/memory2.jsp?m=15&s=10"

申请15M内存，10ms sleep，1个线程

单线程情况下，堆内存分配，回收的步骤：

1、A线程在Eden申请了15M内存，A线程需要10ms左右的时间才能释放15M内存

2、10ms之后，A线程执行结束，这个时候新的请求进来，接下去可能是A线程也可能是其他线程接收这个请求，同样需要在Eden申请15M内存

3、由于Eden区内存不够，因为只有0.812M空闲内存，所以触发minorGC，由于Eden区的先前A线程申请的15M内存已经没有引用，所以全部收回，几乎没有数据进入survivor区，所以也几乎没有数据进入Old区。

4、Eden又空出15.812M内存空间，给新的A线程分配15M内存

5、新的A线程内存分配完成，Eden区又只剩下0.812M的空闲内存，重复1~5的过程

....

结果：无Full GC ，内存分配，回收正常（同时也说明单线程情况下内存是不可能成为瓶颈的）

查看“视频1” ，

堆内存的分配

例子2：2个并发线程

tomcat中jvm的配置：JAVA_OPTS=-Xms256m -Xmx256m -Xss128k -XX:MaxTenuringThreshold=15

ab -n1000 -c2 "http://localhost/perf/memory2.jsp?m=15&s=10"

2个线程并发的时候，堆内存分配，回收步骤：

1、A线程在Eden申请了15M内存，A线程需要10ms左右的时间才能释放15M内存

2、B线程此时也申请了15M内存，因为Eden区总共只有15.812M内存，除去被A申请的15M，所以只剩下0.812M

3、由于Eden区内存不够，触发minorGC，由于此时A线程并没有执行完成，那个15M内存不能被回收。所以将A线程的15M内存copy到其中一个Survivor区

4、由于Survivor区只有1.938M空间，放不下A线程的15M内存，则又将A线程的15M内存直接拷贝到Old区

5、拷贝完成之后，Eden又有了15.812M内存空间

6、B线程的15M内存在Eden完成了分配

....

结果：Full GC频繁 （同时也说明了一旦到了某个临界值，则Old区再大也是杯水车薪）

查看“视频2”

堆内存的分配

看了上面的例子，如果Eden区的空间有30M，是否内存分配正常？或者说满足：(并发线程数*线程占用内存) < Eden 内存分配就是健康的？

实际上这个是错误的：10个并发线程，每个线程占用内存10M，假设Eden有101M内存，由于第11个线程来申请内存的时候，Eden已经占用了100M内存，没有多余的空间，则发生MinorGC，但是先前的线程，即便是并发也是有先后顺序的，而最近执行的线程很可能还没有结束掉，则100M内存并不能被全部释放，可能有10M~NM是需要被放到survivor区的。如果这个时候survivor区的空间太小，则会直接被放到Old区，具体看后面的**例子3**和**例子4**

堆内存的分配

例子3：2个并发线程，线程占有内存调整到1M

ab -n1000 -c2 "http://localhost/perf/memory2.jsp?m=1&s=10"

2个线程并发的时候，堆内存分配，回收步骤：

1、A线程在Eden申请了1M内存，B线程也申请了1M内存，持续轮流申请，当第8个B线程开始申请1M内存的时候，发现Eden没有足够的空间，此时Eden已被占用了15M

2、触发minorGC，首先所有之前B线程申请的内存被全部回收，前7个A线程申请的内存也全部被回收，第8个A线程申请的内存有可能被回收，也可能因为没有执行完成，无法被回收

3、由于每个线程的执行时间是10ms，所以第8个A线程没有执行完成的概率非常大

4、A线程的1M内存由于不能被回收，被copy放入survivor区，survivor区空间是1.938M，所以没有问题

5、重复执行以上步骤，当再次触发minorGC的时候，survivor区的之前的A线程留下的1M内存也会被这次gc所回收

6、没有发生往Old区copy数据的事件

....

结果：无Full GC，则内存分配，回收正常

查看“视频3”

堆内存的分配

例子4：2个并发线程，线程占有内存调整到2M

ab -n1000 -c2 "http://localhost/perf/memory2.jsp?m=2&s=10"

2个线程并发的时候，堆内存分配，回收步骤：

1、A线程在Eden申请了2M内存，B线程也申请了2M内存，持续重复之前的申请，当第4个B线程开始申请2M内存的时候，发现Eden没有足够的空间，此时Eden已被占用了14M

2、触发minorGC，首先所有之前B线程申请的内存被全部回收，前3个A线程申请的内存也全部被回收，第4个A线程申请的内存有可能被回收，也可能没有执行完成，无法被回收

3、由于每个线程的执行时间是10ms，所以第4个A线程没有执行完成的概率非常大

4、A线程的2M内存由于不能被回收，被copy放入survivor区，survivor区空间是1.938M，内存不够，直接被放入Old区

5、重复执行以上步骤，不停有2M的内存被放入Old区

....

结果：Full GC频繁

查看“视频4”

堆内存的分配

例子5：在例子4的基础之上，调整survivor区的大小由1.938M调整为2.438M （通过设置-XX:SurvivorRatio=6实现 ）

tomcat中jvm的配置：JAVA_OPTS=-Xms256m -Xmx256m -Xss128k -XX:MaxTenuringThreshold=15 -XX:SurvivorRatio=6

ab -n1000 -c2 "http://localhost/perf/memory2.jsp?m=2&s=10"

2个线程并发的时候，内存分配，回收步骤：

1、A线程在Eden申请了2M内存，B线程也申请了2M内存，持续重复之前的申请，当第4个B线程开始申请2M内存的时候，发现Eden没有足够的空间，此时Eden已被占用了14M

2、触发minorGC，首先所有之前B线程申请的内存被全部回收，前3个A线程申请的内存也全部被回收，第4个A线程申请的内存有可能被回收，也可能没有执行完成，无法被回收

3、由于每个线程的执行时间是10ms，所以第4个A线程没有执行完成的概率非常大

4、A线程的2M内存由于不能被回收，被copy放入survivor区，survivor区空间是2.438M，所以没有问题

5、重复执行以上步骤，当再次触发minorGC的时候，survivor区的之前的A线程留下的2M内存也会被这次gc所回收

6、没有发生往Old区copy数据的事件

....

结果：无Full GC，同时也说明有时候略微做一下jvm参数调整可以让内存分配和回收更加健康

查看“视频5”

堆内存的分配

1、eden区太小，则导致minorGC频繁

2、survivor太小，则非常容易导致对象被直接copy到old区(survivor只存放eden区无法被回收的对象，并不能直接说明这些对象相对较老，很多刚刚创建的对象也可能被直接拷贝进来)

3、young区太大，则容易导致一次minorGC 耗时

GC的时候，jvm是不允许内存分配的，所以GC时间越短越好

一般建议young区为整个堆的1/4，如堆为2g，则young区分配500M

sun推荐的配置，survivor区一般设置为young区的1/8，如果young区为500M，则survivor可以设置为60M

如果一个线程占用的内存为2M，则50M的survivor 支持25个并发线程是肯定OK的

（实际上jvm调优应该是以减少GC的时间和系统停顿时间为目的而进行堆内存的各个区段的分配，以及GC策略的调整）

堆内存的分配

按照2G的堆内存配置，young区应该在500M左右，survivor区一般配置60M左右，由此参数来计算并发线程支持的数量，以及每个线程所需要申请的内存对象

最保险的公式：线程平均占用内存=(survivor区大小/并发线程数量) ，　survivor区大小=young/(6~8) ，young区大小=堆总大小(-Xmx) / 4

在这个公式下，理论上不会有对象被迁移到Old区

所以根据最保险公司可以得到结论：

如果要支持60个并发，则每个线程申请的对象平均内存占用不能超过60/60=1M

如果要支持30个并发，则每个线程申请的对象平均内存占用不能超过60/30=2M

并不一定要如此严格的执行这个标准：

1、线程一般来说就算其还没有执行完成，但是也是有大部分临时对象是可以被回收的，只有少部生命周期比较长的临时对象才不能被回收

2、实际情况是允许对象迁移到Old区的

所以公式

线程平均占用内存=(survivor区大小/并发线程数量)

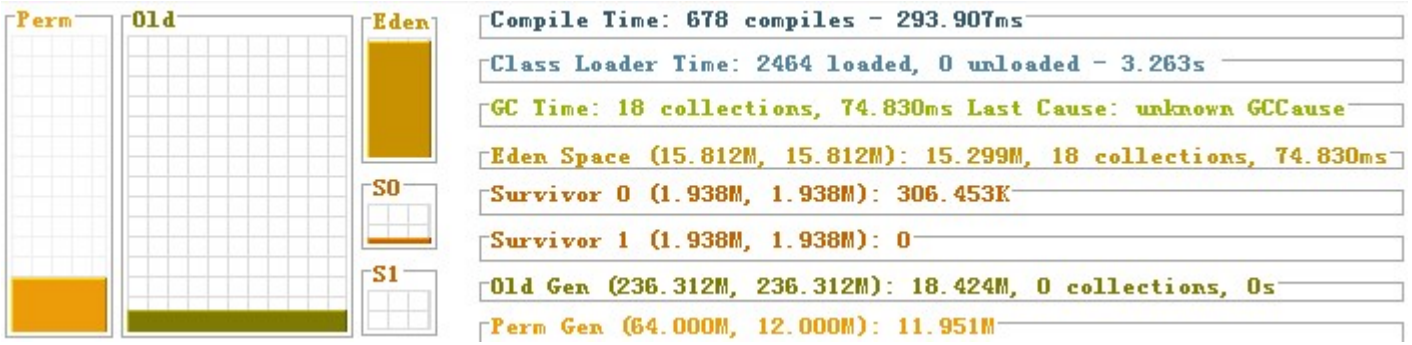
可以再乘以一个系数

线程平均占用内存=(survivor区大小/并发线程数量) * 系数

实际情况还是通过不断增加用户进行性能压测，获取gc日志来分析内存分配回收是否合理为准。同时配以不断调整jvm的参数，达到最佳情况

每个请求占用多少内存计算

根据：
1、minorGC的触发条件，当eden区申请空间失败，则进行minorGC
2、所有对象都在eden区创建
得出：
公式：每个请求占用的内存= Eden /（QPS * minorGC的平均间隔时间(秒)）
...计算方法：
平均两次minor gc间隔消耗(第二次minor gc开始时占用的空间-第一次minor gc时剩下的空间)的内存/(平均x秒触发一次minor gc * QPS)



Detail 的GC.log：

2010-08-18T14:48:42.637+0800: 767442.589: [GC [PSYoungGen: 504824K->1526K(507904K)] 981442K->478201K(1671168K), 0.0231660 secs] [Times: user=0.01 sys=0.00, real=0.03 secs]

2010-08-18T14:48:46.751+0800: 767446.704: [GC [PSYoungGen: 504644K->3330K(507072K)] 981319K->480109K(1670336K), 0.0120540 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]

Gc.log

根据：Detail 的GC.log：

2010-08-18T14:48:42.637+0800: 767442.589: [GC [PSYoungGen: 504824K->1526K(507904K)] 981442K->478201K(1671168K), 0.0231660 secs] [Times: user=0.01 sys=0.00, real=0.03 secs]

时间：14:48:42
young区从504824K减少到1526K，减少了503298K
整个堆　从981442K减少到478201K，减少了503241K
差额503298K - 503241K = 57K
进入Old区 57K

2010-08-18T14:48:47.551+0800: 767446.704: [GC [PSYoungGen: 504644K->3330K(507072K)] 981319K->480109K(1670336K), 0.0120540 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]
时间：14:48:47
young区从504644K减少到3330K，减少了501314K
整个堆　从981319K减少到480109K，减少了501210K
差额501314K - 501210K = 104K
进入Old区 104K

两次GC时间差为5s
每个5s 有大约104K 进入old 区 ， 1小时Old区产生 60/5 * 104 * 60 /1024= 73.125M， 正常

内存优化

1、32位系统jvm cache控制在400M以下，Old区按1G计算，则保留600M的空闲空间
如果超出则考虑使用一些技术进行优化

1、利用NIO的mmap机制，直接使用os内存来代替jvm 的堆
2、90%经常被使用的进行cache，10%不经常使用的进行文件化 ，forest案例
3、jvm堆对象被直接迁移到OS内存区（风险）

2、减少每次请求的内存占用(**很重要**)

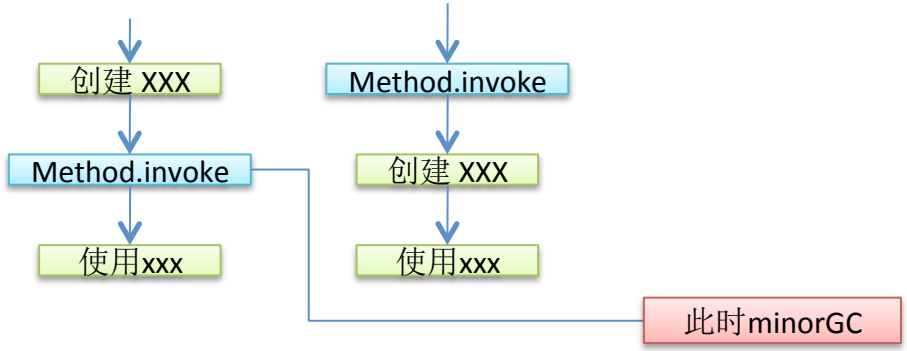
缩短对象的生命周期，减少线程生命周期的对象创建，大量使用局部对象，对象从创建到释放的时间尽可能短，适时使用内存回收优化的对象释放方式，如及时将对象的引用设置为null

✓1、尽量减少线程请求生命周期里的对象的数量

如：request.setAttribute("a",new XXX());
还有一些web框架本身的上下文Map，尽量不要放入大数据量的对象

✓2、对象创建到可回收的时间要尽可能短

```
//大内存数据
1. Object xxx = new XXX();
// 执行消耗时间的工作，包括线程等待或者CPU消耗
2. Object yyy = method.invoke();//耗时10ms
3.zzz = yyy + xxx;
4.xxx=null;
可以修改成：
// 执行消耗时间的工作，包括线程等待或者CPU消耗
1. Object yyy = method.invoke();//耗时10ms
//大内存数据
2. Object xxx = new XXX();
3. zzz = yyy + xxx;
4. xxx= null;
```



✓3、其他如全局对象，静态对象的数据量要进行控制，不过这种相对比较容易通过压力测试发现问题

✓4、对于通过网络或者其他第三方获取的数据值，要保持怀疑态度，比如搜索引擎返回的搜索字段的个数，理论上是一个100以内的值，但是有可能会得到一个你无法想象的值。

要点回顾

1、找到优化的方向
提升QPS还是提升RT，这两者的优化思路截然不同
2、QPS的提升，RT的提升
我们知道了QPS的优化必须针对瓶颈进行优化(目前我们的系统瓶颈基本上都应该是CPU)，而响应时间缩短更多依赖减少IO，减少线程等待，减少线程数量
公式：平均响应时间 =（并发线程数/最佳线程数） * 最佳线程数的响应时间

3、最佳线程数
公式：最佳线程数量=((线程等待时间+线程cpu时间)/线程cpu时间) * cpu数量

4、线程
线程是一种有限资源，何时需要关心线程本身的开销
5、服务器需要同时处理大量的连接请求，并且服务器本身依赖其他IO资源，依赖的IO资源的响应时间远远超过服务器的计算时间，并且IO的响应时间可能变化很大
典型：proxy服务器，webww
6、服务的计算量非常少，但是要请求大量的IO资源，并且IO资源的响应时间不确定，并且大大于服务计算时间，此时如果需要提升并发能力，则NIO吧
典型：网页爬虫

7、堆的分配和回收
最保险的公式（保险到什么程度，如果是10个并发线程，那么即便是这10个线程全部非常非常的缓慢，也不会有任何内存分配和回收问题）
并发线程数量 = (survivor区大小/线程平均占用内存)
由于太过保守，做一下调整，增加一个系数(按照detail系统来说这个系数应该在5左右)
并发线程数量 = (survivor区大小/线程平均占用内存) * 系数
系数受平均响应时间影响

8、内存瓶颈
内存大部分情况下不应该是瓶颈，但是线上情况复杂，出现内存瓶颈的概率很高，我们要做好性能压测的时候，用5~6倍的最佳线程数来对内存进行考验，以防止内存瓶颈被遗漏
公式：每个请求占用的内存= Eden /(QPS * minorGC的平均间隔时间(秒))

Time for Q&A

讨论：Detail性能测试-线程数对QPS和GC的影响

如果出现内存瓶颈，优化相对较容易，内存不应该是我们的瓶颈

1、我们的系统基本上最佳线程数量都比较少，内存基本上都被用来单次请求的临时开销

2、如果出现瓶颈，则每个请求占用的内存略作优化就可以产生极大的飞跃

性能点	性能测试场景	测试类型	耗时H	配置项		TPS	RT(s)	CPU Util %	Load	FGC	JVM	备注
				Apache	Jboss							
Detail宝贝详情页	访问Detail300个页面	负载测试15用户	1	默认	ajp=200	69.95	0.17	66.15%	4.97	1小时	N.A.	
	访问Detail300个页面	稳定性测试100用户	18	默认	ajp=200	55.35	1.80	74.60%	7.10	31.9秒	JVM1	2031次FGC
	访问Detail300个页面	负载测试15用户	1	默认	ajp=15	66.45	0.17	63.51%	4.64	1小时	N.A.	
	访问Detail300个页面	稳定性测试100用户	15	默认	ajp=15	69.86	1.43	71.11%	5.53	1小时	JVM2	15次FGC

两个和线程相关公式

公式1：服务器端最佳线程数量=((线程等待时间+线程cpu时间)/线程cpu时间) * cpu数量

公式2：(小GC时间间隔/rt)*(并发线程数量 * thm) <=young

Rt = 平均响应时间

Thm =线程完成一次请求生命周期内占用的平均内存

Young=年轻代内存设置的大小

影响这个公式准确性的关键因素：RT、线程内对象GC的友好性

内存会是瓶颈？

1、在模板里加一个sleep，看看开满200个ajp线程，会占用多少堆内存？

Hesper的测试发现一个线程占用2M内存，在10-20个线程下，内存还不会是瓶颈。线上hesper的最佳线程数量是9，所以hesper的内存优化可以暂时不考虑。

2、内存的越少使用本身会降低CPU的消耗

例子

测试例子1代码：

```
<%
// 线程延迟时间ms
String delay = request.getParameter("delay");
int delayInt = 0;
if (delay != null ) {
                delayInt = Integer.parseInt(delay);
}
// 线程占有heap内存字节
String thm = request.getParameter("thm");
int thmInt = 0;
if (thm != null ) {
                thmInt = Integer.parseInt(thm);
}

%>

<%
// mem
byte[] thmByte = new byte[thmInt];
// wait
if(delayInt != 0) {
                Thread.sleep(delayInt);
}
// cpu

%>
hello world! <br>
delay=<%=delayInt%>ms<br>
thm=<%=thmInt%>byte<br>
freememory=<%=(Runtime.getRuntime().freeMemory() / 1024)%>K<br>
```

服务信息：

linux Linux dev211001.sqa 2.6.9-67.ELsmp #1 SMP Wed Nov 7 13:58:04 EST 2007 i686 i686 i386 GNU/Linux

tomcat 6.0.26

jdk 1.6

服务器ip：192.168.211.1

测试机器ip:192.168.211.20

性能监控工具：java visualVM

测试工具ab

我们的系统需要多少线程

执行 ab -n100000 -c10 http://192.168.211.1:8080/test.jsp

发现服务不断full gc，停止执行之后还是这样

原因是因为jsp创建了session对象，导致:98%都是session占用，session过期是30分钟，所以导致oom

One instance of "org.apache.catalina.session.StandardManager" loaded by "org.apache.catalina.loader.StandardClassLoader @ 0xa4e50650" occupies 284,825,208 (98.48%) bytes. The memory is accumulated in one instance of "java.util.concurrent.ConcurrentHashMap$Segment[]" loaded by "<system class loader>".

<%@ page session=”false”%>

再次进行测试，一切正常

我们的系统需要多少线程

1、hesper 9个

2、detail 11个

我们的系统需要多少线程

1、hesper 9个

2、detail 11个

超过最佳线程数，响应时间递增？

再试一下2个并发：

ab -n10000 -c2 http://192.168.211.1:8080/test.jsp?thm=1024000

结果显示：

2个并发，每个申请1M内存，测试结果，qps在1100左右，rt在1.911ms左右 ，qps和前面的测试结果相当。

100个并发，每个申请1M内存，测试结果，qps在1100左右，rt在89ms左右

ab -n10000 -c100 http://192.168.211.1:8080/test.jsp?thm=1024000

200个并发，每个申请1M内存，测试结果，qps在1100左右，rt在170ms左右

ab -n100000 -c200 http://192.168.211.1:8080/test.jsp?thm=1024000

线程数量翻倍，响应时间翻倍，QPS不变，这里在深入说明原因：

刚刚之前说过，这个例子的最佳线程数量为2，所以第一个用例有98个线程在瞬间是等待，第二个例子有198个线程在瞬间是等待的

因为并发数量是2（2个CPU），所以第1,2个请求，分别只需要2ms（实际是1.91ms，为了计算方便用2ms），同时有98个线程在等待，第3,4个请求，因为已经等待了2ms，所以等到完成需要4ms，以此类推，第99,100个线程需要100ms，第101,102线程也是100ms，因为线程池数量是100个。同理，200个线程的情况，在第199,200个线程的时候需要200ms。这个结果和实际结果的89ms和170ms还有比较接近的