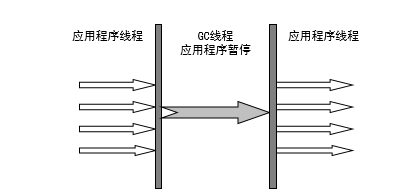
#### 一:回收器

##### 1.1串行收集器:

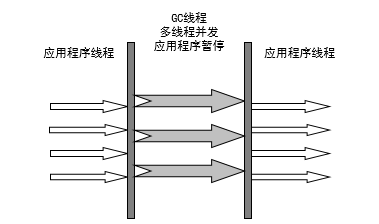
单线程回收，所以性能相对比较慢,可以用-XX:+UseSerialGC来启动串行回收器



如图所示，暂停应用线程，只有一个GC回收线程。

##### 1.2 ParNew并行收集器:

并行收集器:-XX:+UseParNewGC来开启并行收集器。可以使用-XX:ParallelGCThreads来指定线程的数量。这样在多核的情况下它的回收效率更快，但是如果单核的情况下也并不见得快。



##### 1.3 Parallel并行收集器

-XX:+UserParanllelGC 新生代开启并行收集器。

-XX:+UseParallelOldGC 在老年代开启并行收集器。

-XX:MaxGCPauseMills:最大停顿时间,GC尽力保证回收时间不超过设定值。

-XX:GCTimeRatio垃圾回收时间占总时间比默认99，即最大允许1%时间做GC.

1.4 CMS收集器(concurrent Mark weep);并发标记清楚

-XX:+UseConcMarkSweepGC开启。

可以跟应用程序一起进行，这样的话吞吐量就会降低。

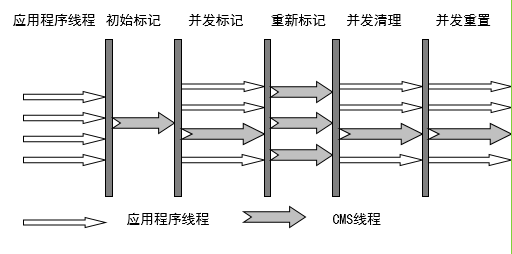
工作步骤:

1 初始标记 直接关联到对象，速度快。

2 并发标记 主要标记过程，标记全部对象。

3 重新标记 由于并发标记时，用户线程在运行，重新标记。

4 并发清理 基于标记结果，直接清理对象。



CMS是一个老年代的回收器,如下面的GC参数所示的，经历了五个阶段。

1.662: [GC [1 **CMS-initial-mark**: 28122K(49152K)] 29959K(63936K), 0.0046877 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

1.666: [CMS-concurrent-mark-start]

1.699: [**CMS-concurrent-mark**: 0.033/0.033 secs] [Times: user=0.25 sys=0.00, real=0.03 secs]

1.699: [CMS-concurrent-preclean-start]

1.700: [CMS-concurrent-preclean: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

1.700: [GC[YG occupancy: 1837 K (14784 K)]1.700: [Rescan (parallel) , 0.0009330 secs]1.701: [weak refs processing, 0.0000180 secs] [1 **CMS-remark**: 28122K(49152K)] 29959K(63936K), 0.0010248 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

1.702: [CMS-concurrent-sweep-start]

1.739: [**CMS-concurrent-sweep**: 0.035/0.037 secs] [Times: user=0.11 sys=0.02, real=0.05 secs]

1.739: [CMS-concurrent-reset-start]

1.741: [**CMS-concurrent-reset**: 0.001/0.001 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

CMS尽可能的降低停顿，所以清理不彻底，因为和用户线程一起运行，所以不能再空间快满时再清理。CMS用标记清楚算法，而不用标记压缩算法，就是因为要与用户程序并发执行，没有时间去整理碎片。

#### 二 jvm参数整理

-XX:+ UseCMSCompactAtFullCollection Full GC后，进行一次整理,整理过程是独占的，会引起停顿时间变长

-XX:+CMSFullGCsBeforeCompaction 设置进行几次Full GC后，进行一次碎片整理

-XX:ParallelCMSThreads设定CMS的线程数量

-XX:+UseSerialGC：在新生代和老年代使用串行收集器

-XX:SurvivorRatio：设置eden区大小和survivior区大小的比例

-XX:NewRatio:新生代和老年代的比

-XX:+UseParNewGC：在新生代使用并行收集器

-XX:+UseParallelGC ：新生代使用并行回收收集器

-XX:+UseParallelOldGC：老年代使用并行回收收集器

-XX:ParallelGCThreads：设置用于垃圾回收的线程数

-XX:+UseConcMarkSweepGC：新生代使用并行收集器，老年代使用CMS+串行收集器

-XX:ParallelCMSThreads：设定CMS的线程数量

-XX:CMSInitiatingOccupancyFraction：设置CMS收集器在老年代空间被使用多少后触发

-XX:+UseCMSCompactAtFullCollection：设置CMS收集器在完成垃圾收集后是否要进行一次内存碎片的整理

-XX:CMSFullGCsBeforeCompaction：设定进行多少次CMS垃圾回收后，进行一次内存压缩

-XX:+CMSClassUnloadingEnabled：允许对类元数据进行回收

-XX:CMSInitiatingPermOccupancyFraction：当永久区占用率达到这一百分比时，启动CMS回收

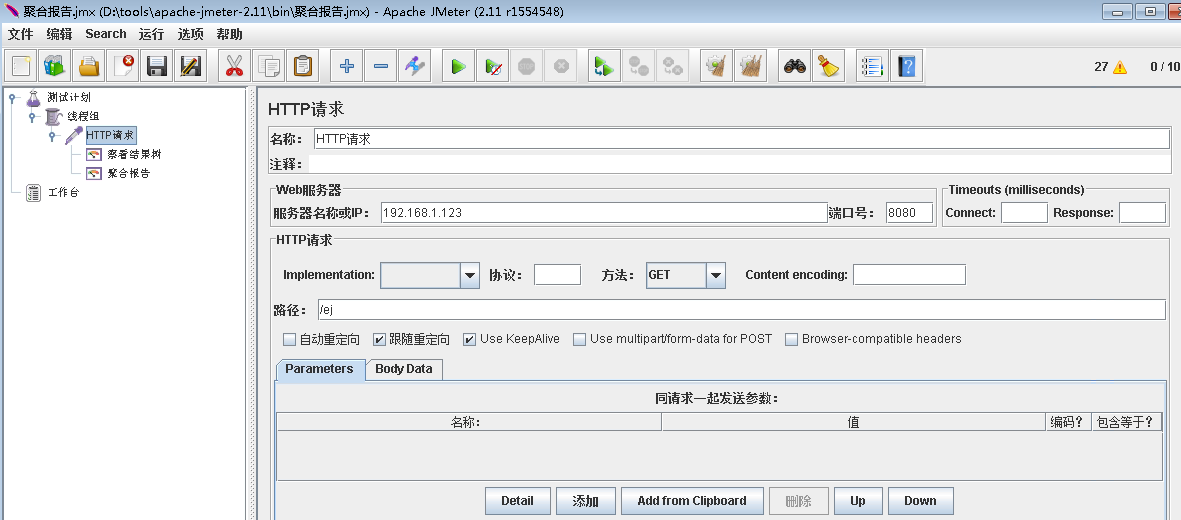
-XX:UseCMSInitiatingOccupancyOnly：表示只在到达阀值的时候，才进行CMS回收

#### 三 案例

##### 3.1场景：性能测试工具JMeter,测试吞吐量,目标让tomcat有一个不错的吞吐量。

性能测试工具

建立10个线程，每个线程请求Tomcat 1000次 共10000次请求

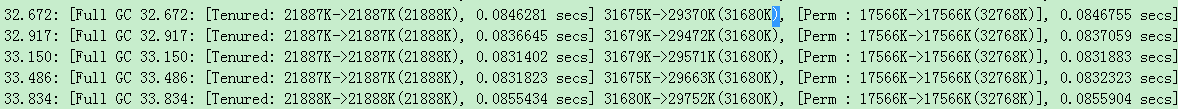


##### 3.2案例一 JDK6：使用32M堆处理请求

set CATALINA\_OPTS=-server -Xloggc:gc.log -XX:+PrintGCDetails -Xmx32M -Xms32M -XX:+HeapDumpOnOutOfMemoryError -XX:+UseSerialGC -XX:PermSize=32M

设置如上参数：打印日志详情，最大堆内存，最小堆内存，堆溢出输出日志，串行化进行垃圾回收，老年代内存。如下所示10000次请求，平均6毫秒，半数4毫秒，吞吐量540





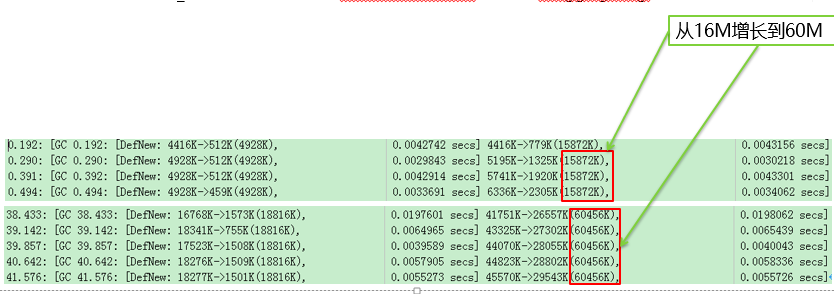
Full gc次数5次。

##### 3.3案例二 对比案例一 JDK6：使用最大堆512M堆处理请求

set CATALINA\_OPTS=-Xmx512m -XX:MaxPermSize=32M -Xloggc:gc.log -XX:+PrintGCDetails

调整堆的大小,吞吐量有所增长651





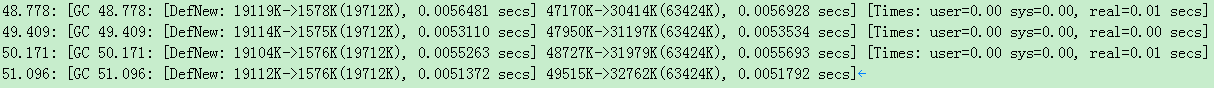
但是GC的数据从16G到了60M可见新生代最小堆的负载已经不足,吞吐量虽然上来了，但GC的次数更多了。

##### 3.4案例三 对比案例二: JDK6使用最大堆512M堆处理请求

set CATALINA\_OPTS=-Xmx512m -Xms64m -XX:MaxPermSize=32M -Xloggc:gc.log -XX:+PrintGCDetails

调整最小堆大小为64M，结果 GC数量减少 大部分是Minor GC，吞吐量上升.





##### 3.5案例四 对比案例三: JDK6：使用最大堆512M堆处理请求

set CATALINA\_OPTS=-Xmx512m -Xms64m -XX:MaxPermSize=32M -Xloggc:gc.log -XX:+PrintGCDetails -XX:+UseParallelGC -XX:+UseParallelOldGC -XX:ParallelGCThreads=4

同上，只是新生代与老年代都采用并行机制，并启动4个并行线程。GC压力原本不大，修改GC方式影响很小



##### 3.6案例五 JDK 6 Xmx40m 增大GC压力

set CATALINA\_OPTS=-Xmx40m -Xms40m -XX:MaxPermSize=32M -Xloggc:gc.log -XX:+PrintGCDetails

减小堆大小，增加GC压力，使用Serial回收器



##### 3.7案例六 对比案例五: JDK 6 Xmx40m 增大GC压力

set CATALINA\_OPTS=-Xmx40m -Xms40m -XX:MaxPermSize=32M -Xloggc:gc.log -XX:+PrintGCDetails -XX:+UseParallelOldGC -XX:ParallelGCThreads=4

减小堆大小，增加GC压力，使用并行回收器,可以发现并行回收器在相同压力下相对于串行回收器，吞吐量明显提高。



##### 3.8案例七

set CATALINA\_OPTS=-Xmx40m -Xms40m -XX:MaxPermSize=32M -Xloggc:gc.log -XX:+PrintGCDetails -XX:+UseParNewGC

减小堆大小，增加GC压力，使用ParNew回收器



##### 3.9案例八:启动Tomcat 7,使用JDK6,不加任何参数启动测试。



案例九对比案例八: 启动Tomcat 7,使用JDK7,不加任何参数启动测试。

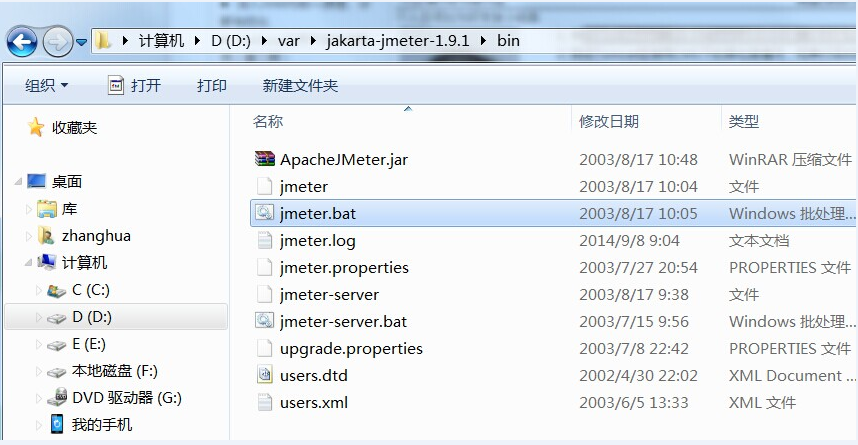


可见升级jdk也是可以提高吞吐量的。

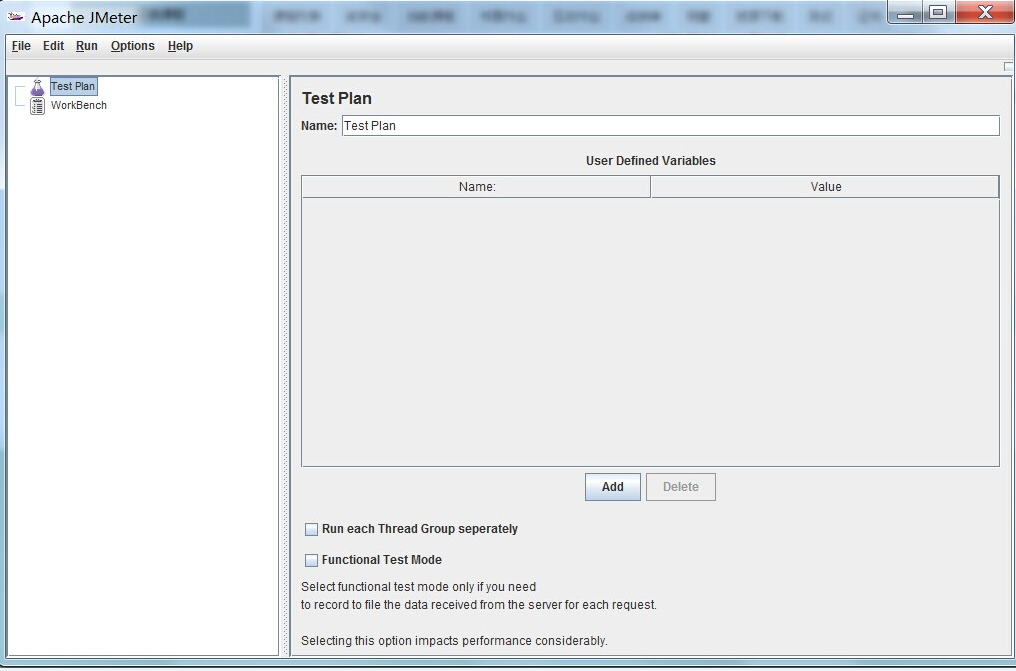
1. 学习使用JMeter，并测试Tomcat的吞吐率，给出有关配置和测试结果的至少5个截图

我的jmeter和tomcat都在同一台机器

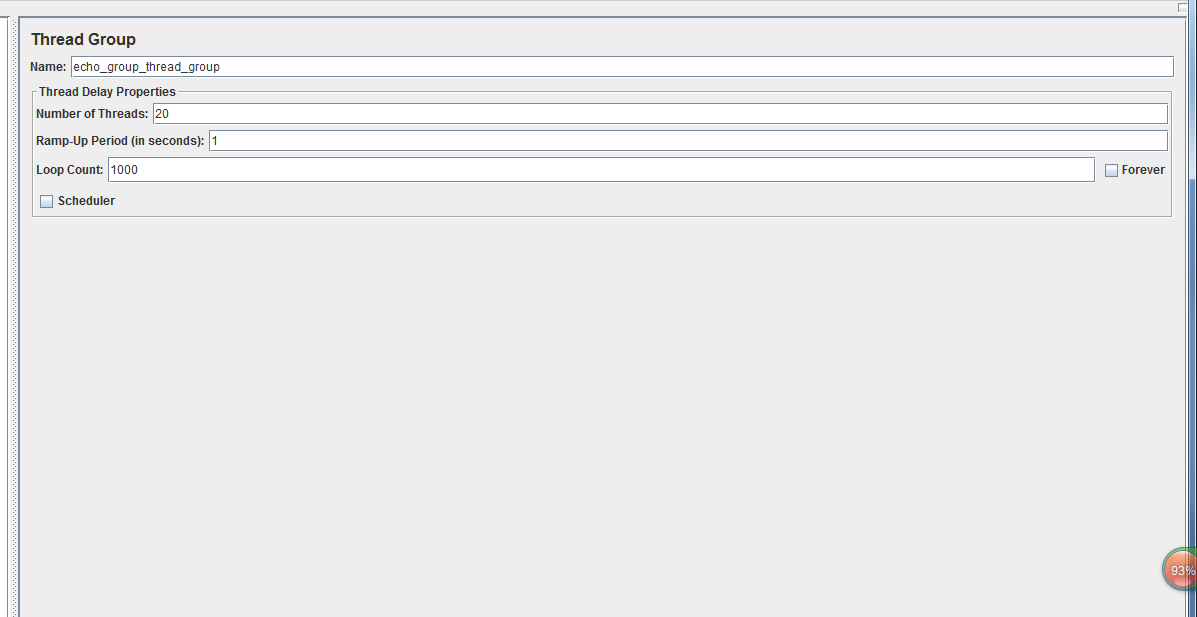
1、首先要安装jmeter,安装过程非常简单，将下载好的zip包解压，并配置环境变量就可以了，此时点击是jmeter就可以运行

****

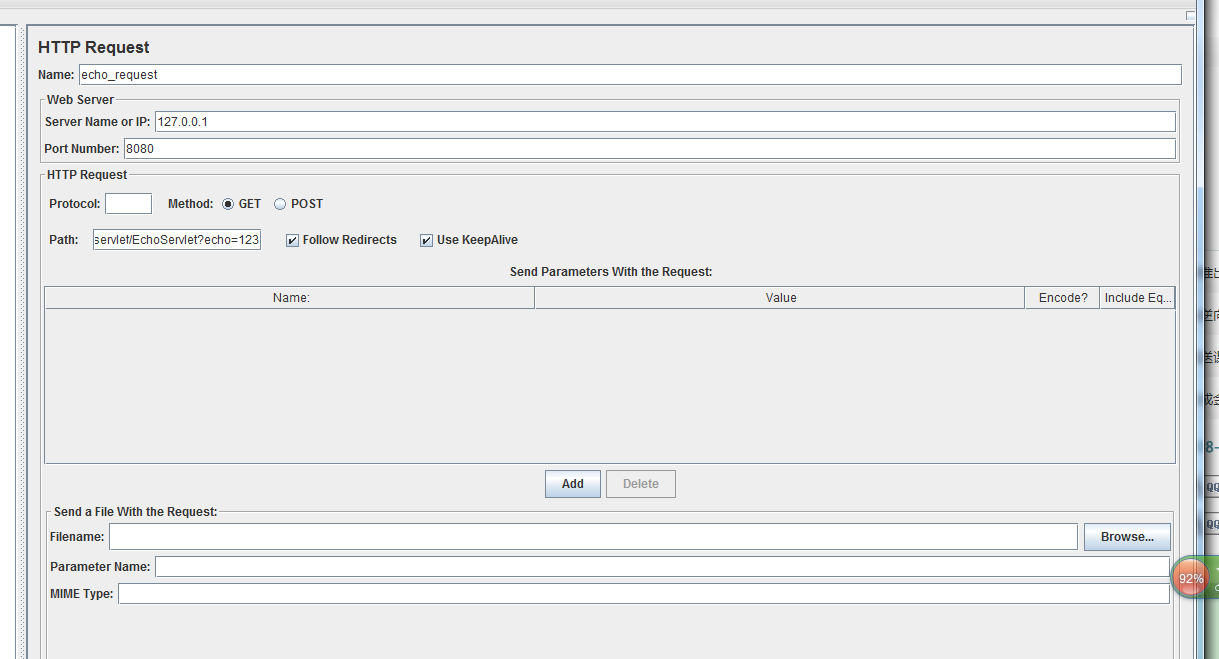
**2、运行Jmeter.bat：**

****

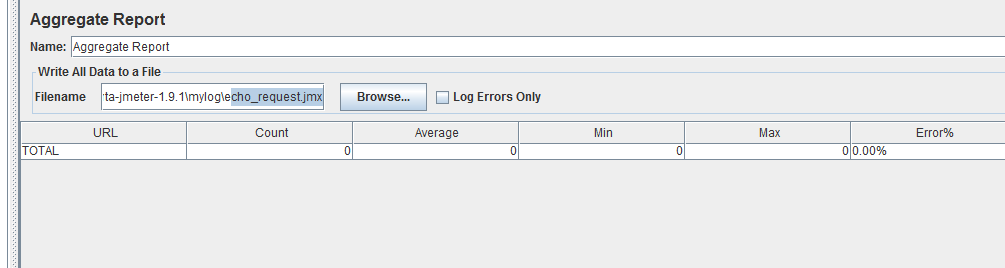
**3、增加Jmeter线程组**

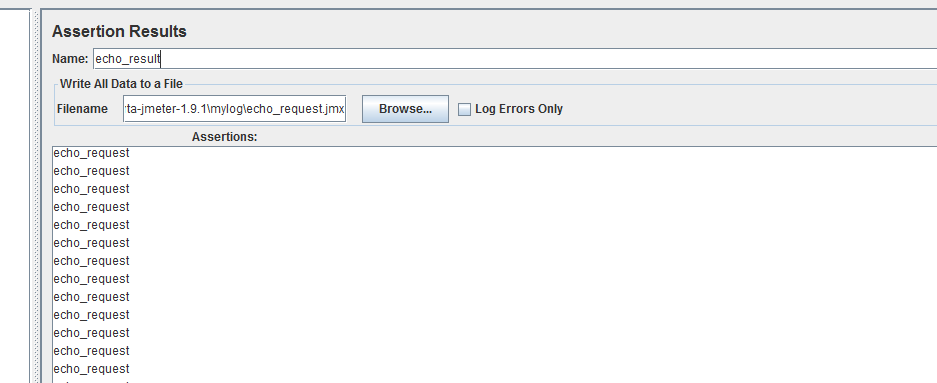
****

**4、设置http request**

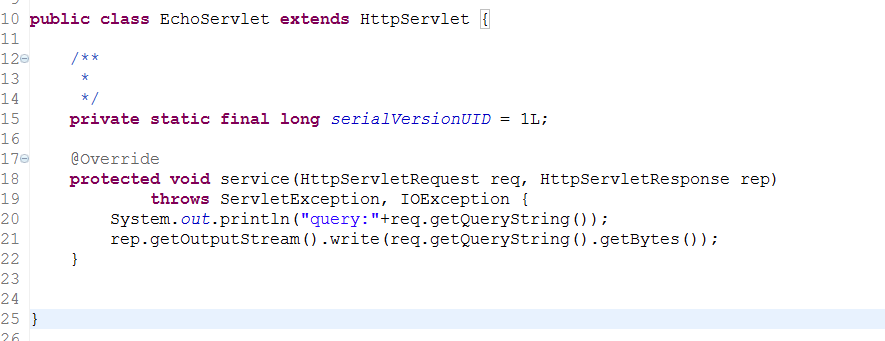
****

**5、增加聚合报表和聚合的result：**

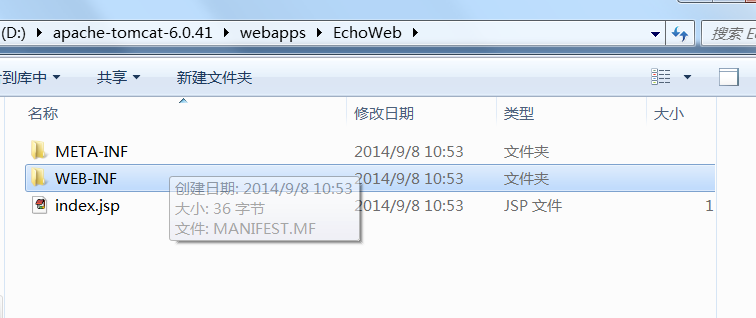
****

****

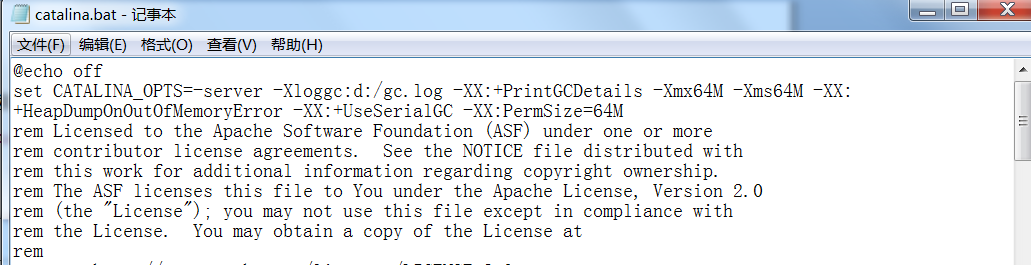
**6、开发echo servlet，代码如下：**

****

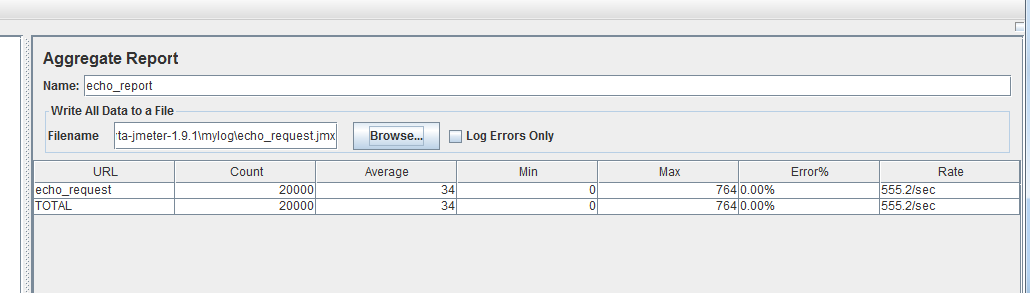
**7、配置web.xm并发包到tomcat下面：**

****

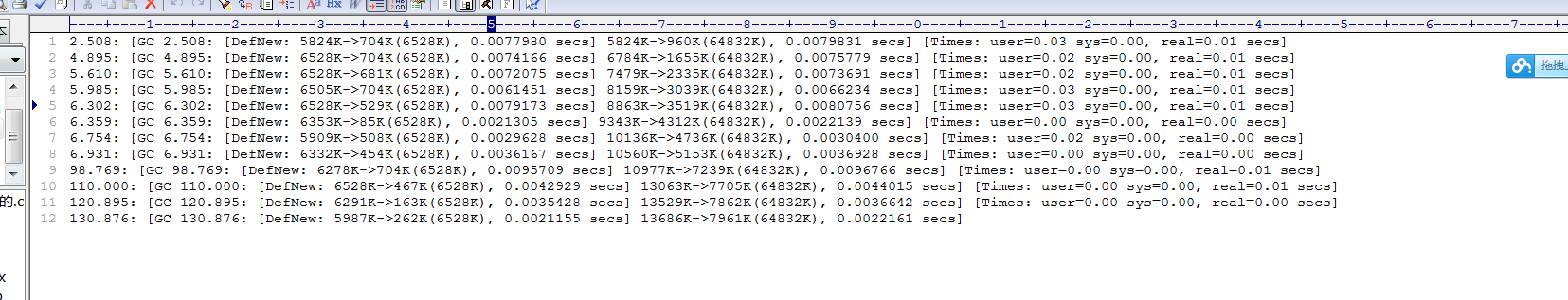
**8、在catalina里面设置jvm启动参数：我初始设置比老师大，gcLog存放d盘**

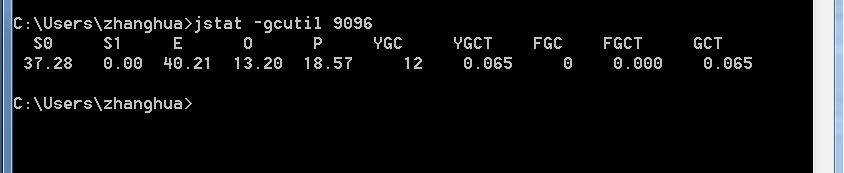
****

**9、启动tomcat,并运行测试用例，查看聚合结果：可以看到Min最在0，最大在764ms,我想因为是本机所以min可以达到0ms,大概每秒处理550，没有发生错误**

****

**10、看gc.log日志：因为我这个是servlet所以在内存需要方面非常少，以至于没有产生full gc**

**通过jstat也能对上关系：**

****

1. 测试Tomcat在使用CMS下的吞吐率情况，和串行回收器和并行回收器在相同环境下做比较，给出几种回收器的报表截图

由于使用的是servlet导致内存消耗比较小，所以servlet代码改造：

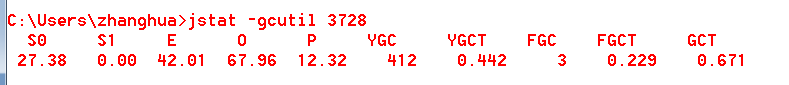


通过以上方式可以产生对象

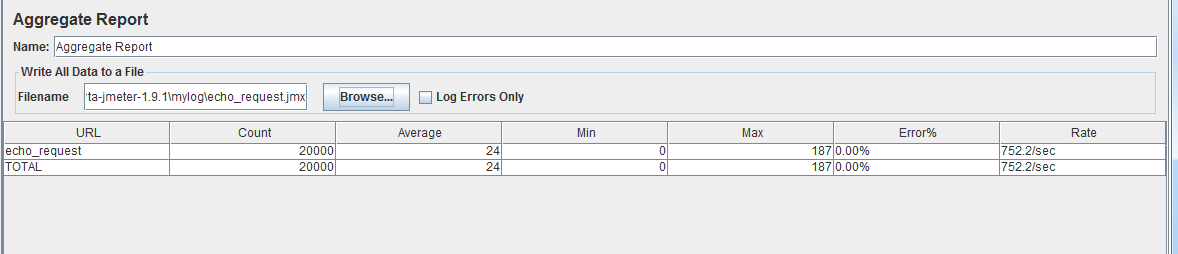
**2.1串行回收器参数：**

set CATALINA\_OPTS=-Xloggc:d:/gc.log -XX:+PrintGCDetails -Xmx40M -Xms4M -XX:+UseSerialGC -XX:PermSize=64M

jstat汇总结果：



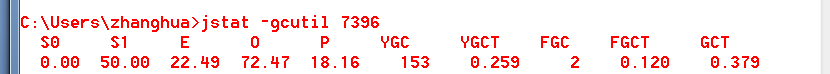
聚合报表结果：



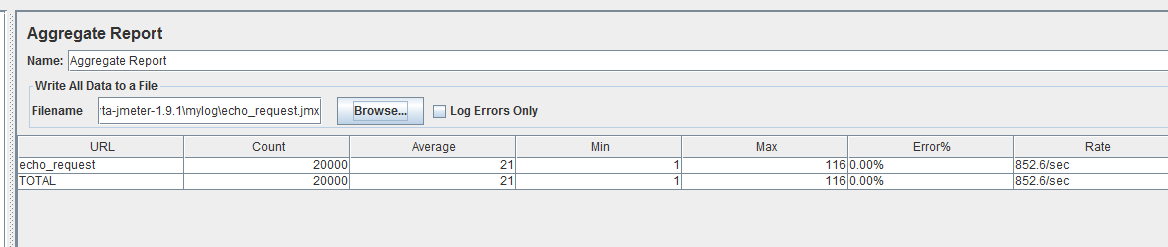
**2.2并行回收器：**

set CATALINA\_OPTS=-Xloggc:d:/gc.log -XX:+PrintGCDetails -Xmx40M -Xms4M -XX:+UseParallelGC -XX:+UseParallelOldGC -XX:ParallelGCThreads=2 -XX:PermSize=64M

jstat汇总结果：

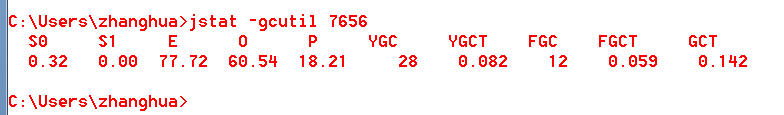


聚合报表结果

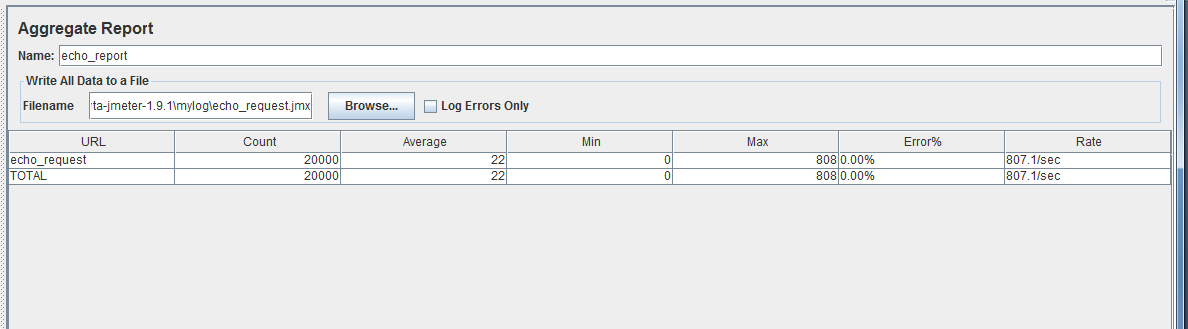


**2.3cms垃圾收集器：**

jstat汇总结果：



聚合报表结果



关于对cms评价：因为我是做nio方面的，对系统的响应时间和垃圾停顿时间比较在意，所以cms还是比较熟的，cms相对来说有以下特点：

1. 能保证系统的响应时间,减少垃圾收集时的停顿时间，对一些做监控的应用优其重要。
2. Cms采用Mark-Sweep会对堆产生空间碎片，但是可以指定在经过几次垃圾回收后再做总的空间整理
3. 从上面图中也可以看到，虽然fgc回收时间相对比并行收集器少，但总的回收次数确是多的，所以会导致系统停顿次数变多，导致吞吐量下降。