目 录

JDK 优化

Jvm 动态库有 client 和 server 两个版本，分别针对桌面应用和服务器应用做了相应的优化，client 版本加 载速度较快，server 版本加载速度较慢但运行起来较快。

%JAVA_HOME%/jre/bin/client/jvm.dll
%JAVA_HOME%/jre/bin/server/jvm.dll
可以看到这两个 jvm.dll 的大小不同。

如何修改默认 Java 调用的 JVM.dll 版本?

在命令行 java -version 可以看到 jvm 配置的是哪个版本。

更改默认 java.exe 调用的 jvm.dll, 这个由 jvm.cfg 决定。编辑 %JAVA_HOME%jre/lib/i386/jvm.cfg
里面第一行写的是 -client 默认就是 client 版本 ，把第二行的-server KNOWN 放到第一行，如下面所示：
-server KNOWN
-client KNOWN
-hotspot ALIASED_TO -client
-classic WARN
-native ERROR
-green ERROR

测试实例

测试环境：
Tomcat5.5.28
jdk1.6.0_13

测试应用程序：
jpetstore5

测试目的：
对比 jdk 中采用 client 和 server 版本的 jvm 的性能区别

测试场景：
访问主页面
登录
查看宠物
添加宠物到购物车
结账
退出

100 个并发用户运行 1 分钟左右

测试结果：
Client 版本的 JVM 比 Server 版本的在大部分事务的平均响应时间上反而略为快点。
在《J2EE 性能测试》这本书中也得到类似的结果：
JDK 1.3 HotSpot Server 2267ms
JDK 1.3 HotSpot Client 926ms
JDK 1.3 Classic 537ms
JDK 1.22 186ms
IBM 1.3 138ms
JRockit 3.1 137ms

选择合适的 JDK 版本

不同版本的 JDK，甚至不同厂家的 JDK 可能都存在着很大的差异，对于性能优化的程度不同。一般来说，尽可能选择最新发布的稳定的 JDK 版本。最新的稳定的 JDK 版本相对以前的 JDK 版本都会做一些 bug 的修改和性能的优化工作。

可以选择自己的需要选择不同的操作系统和对应的 JDK 的版本（只要是符合 Sun 发布的 Java 规范的），但推荐使用 Sun 公司发布的 JDK。确保所使用的版本是最新的，因为 Sun

公司和其它一些公司一直在为提高性能而对 java 虚拟机做一些升级改进。一些报告显示 JDK1.4 在性能上比 JDK1.3 提高了将近 10%到 20%。

垃圾收集 Java 堆的优化

垃圾收集就是自动释放不再被程序所使用的对象的过程。当一个对象不再被程序所引用时，它所引用的堆空间可以被回收，以便被后续的新对象所使用。垃圾收集器必须能够断定哪些对象是不再被引用的，并且能够把它们所占据的堆空间释放出来。如果对象不再被使用，但还有被程序所引用，这时是不能被垃圾收集器所回收的，此时就是所谓的“内存泄漏”。监控应用程序是否发生了内存泄漏，有一个非常优秀的监控工具推荐给大家——Quest 公司的 JProbe 工具，使用它来观察程序运行期的内存变化，并可产生内存快照，从而分析并定位内存泄漏的确切位置，可以精确定位到源码内。这个工具的使用我在后续的章节中还会做具体介绍。

Java 堆是指在程序运行时分配给对象生存的空间。通过-mx/-Xmx 和-ms/-Xms 来设置起始堆的大小和最大堆的大小。根据自己 JDK 的版本和厂家决定使用-mx 和-ms 或-Xmx 和-Xms。Java 堆大小决定了垃圾回收的频度和速度，Java 堆越大，垃圾回收的频度越低，速度越慢。同理，Java 堆越小，垃圾回收的频度越高，速度越快。要想设置比较理想的参数，还是需要了解一些基础知识的。 Java 堆的最大值不能太大，这样会造成系统内存被频繁的交换和分页。所以最大内存必须低于物理内存减去其他应用程序和进程需要的内存。而且堆设置的太大，造成垃圾回收的时间过长，这样将得不偿失，极大的影响程序的性能。以下是一些经常使用的参数设置：

1) 设置-Xms 等于-XmX 的值；
2) 估计内存中存活对象所占的空间的大小，设置-Xms 等于此值，-Xmx 四倍于此值；
3) 设置-Xms 等于-Xmx 的 1/2 大小；
4) 设置-Xms 介于-Xmx 的 1/10 到 1/4 之间；
5) 使用默认的设置。

需要根据自己的运行程序的具体使用场景，来确定最适合自己的参数设置。 除了-Xms 和-Xmx 两个最重要的参数外，还有很多可能会用到的参数，这些参数通常强烈的依赖于垃圾收集的算法，所以可能因为 JDK 的版本和厂家而有所不同。但这些参数一般在 Web 开发中用的比较少，就不做详细介绍了。在实际的应用中注意设置-Xms 和-Xmx 使其尽可能的优化应用程序就行了。对于性能要求很高的程序，就需要自己再多研究研究 Java 虚拟机和垃圾收集算法的机制了。可以看看曹晓钢翻译的《深入 Java 虚拟机》一书。

Tomcat 性能监控与优化

监控

D:\PrefTest\案例\Tomcat 调优\Tomcat 性能监控.pdf

Tomcat 监控页面

JMX

JMX(Java Management Extensions)是一个为应用程序植入管理功能的框架。JMX是一套标准的代理和服务，实际上，用户可以在任何Java应用程序中使用这些代理和服务实现管理。可以利用JDK的JConsole来访问Tomcat JMX接口实施监控，具体步骤如下：
首先，打开tomcat5的bin目录中的catalina.bat文件，在头部注释部分的后面加上：
set JAVA_OPTS=%JAVA_OPTS% -Dcom.sun.management.jmxremote.port=8999
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
如果已经配置好，则可使用 JConsole 打开监控平台查看 Tomcat 性能。

Linux 下配置 catalina.sh 的例子：
JAVA_OPTS='-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=8999
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.authenticate=false
-Djava.rmi.server.hostname=192.168.0.106'

由于 JMX 提供的接口是任何 Java 程序都可以调用访问的，因此我们可以编写 JAVA 程序来收集 Tomcat 性能数据，具体代码如下所示：

```
import java.lang.management.MemoryUsage;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Formatter;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import javax.management.MBeanAttributeInfo;
```

```
import javax.management.MBeanInfo;
import javax.management.MBeanServerConnection;
import javax.management.ObjectInstance;
import javax.management.ObjectName;
import javax.management.openmbean.CompositeDataSupport;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;

public class MonitorTomcat {
/**
* @param args
*/
public static void main(String[] args) {
try {
        String jmxURL = "service:jmx:rmi:///jndi/rmi://192.168.0.106:8999/jmxrmi";//tomcat jmx url
        JMXServiceURL serviceURL = new JMXServiceURL(jmxURL);
        Map map = new HashMap();
        String[] credentials = new String[] { "monitorRole" , "QED" };
        map.put("jmx.remote.credentials", credentials);
        JMXConnector connector = JMXConnectorFactory.connect(serviceURL, map);
        MBeanServerConnection mbsc = connector.getMBeanServerConnection();
        ObjectName threadObjName = new
ObjectName("Catalina:type=ThreadPool,name=http-8080");
        MBeanInfo mbInfo = mbsc.getMBeanInfo(threadObjName);
        String attrName = "currentThreadCount"; //tomcat 的线程数对应的属性值
        MBeanAttributeInfo[] mbAttributes = mbInfo.getAttributes();
        System.out.println("currentThreadCount:"+mbsc.getAttribute(threadObjName, attrName));
        //heap
        for(int j=0;j <mbsc.getDomains().length;j++){
            System.out.println("###########"+mbsc.getDomains()[j]);
        }
        Set MBeanset = mbsc.queryMBeans(null, null);
        System.out.println("MBeanset.size() : " + MBeanset.size());
        Iterator MBeansetIterator = MBeanset.iterator();
        while (MBeansetIterator.hasNext()) {
            ObjectInstance objectInstance = (ObjectInstance)MBeansetIterator.next();
            ObjectName objectName = objectInstance.getObjectName();
            String canonicalName = objectName.getCanonicalName();
            System.out.println("canonicalName : " + canonicalName);
            if (canonicalName.equals("Catalina:host=localhost,type=Cluster")) {
                // Get details of cluster MBeans
                System.out.println("Cluster MBeans Details:");
                System.out.println("==========================================");
```

```
                //getMBeansDetails(canonicalName);
                String canonicalKeyPropList = objectName.getCanonicalKeyPropertyListString();
            }
        }

        // - - - - - - - - - - - - - - - - - - - - - - - - - - - - system - - - - - - - - - - - - - - - - - - - -
        ObjectName runtimeObjName = new ObjectName("java.lang:type=Runtime");
        System.out.println("厂商:"+ (String)mbsc.getAttribute(runtimeObjName, "VmVendor"));
        System.out.println("程序:"+ (String)mbsc.getAttribute(runtimeObjName, "VmName"));
        System.out.println("版本:"+ (String)mbsc.getAttribute(runtimeObjName, "VmVersion"));
        Date starttime=new Date((Long)mbsc.getAttribute(runtimeObjName, "StartTime"));
        SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        System.out.println("启动时间:"+df.format(starttime));
        Long timespan=(Long)mbsc.getAttribute(runtimeObjName, "Uptime");
        System.out.println("连续工作时间:"+MonitorTomcat.formatTimeSpan(timespan));

        //------------------------JVM--------------------------------
        //堆使用率
        ObjectName heapObjName = new ObjectName("java.lang:type=Memory");
        MemoryUsage heapMemoryUsage = MemoryUsage.from((CompositeDataSupport)mbsc.getAttribute(heapObjName, "HeapMemoryUsage"));
        long maxMemory = heapMemoryUsage.getMax(); //堆最大
        long commitMemory = heapMemoryUsage.getCommitted(); //堆当前分配
        long usedMemory = heapMemoryUsage.getUsed();
        System.out.println("heap:"+(double)usedMemory*100/commitMemory+"%"); // 堆使用率
        MemoryUsage nonheapMemoryUsage = MemoryUsage.from((CompositeDataSupport)mbsc.getAttribute(heapObjName, "NonHeapMemoryUsage"));
        long noncommitMemory = nonheapMemoryUsage.getCommitted();
        long nonusedMemory = heapMemoryUsage.getUsed();
        System.out.println("nonheap:"+(double)nonusedMemory*100/noncommitMemory+"%");
        ObjectName permObjName = new ObjectName("java.lang:type=MemoryPool,name=PermGen");
        MemoryUsage permGenUsage = MemoryUsage.from((CompositeDataSupport)mbsc.getAttribute(permObjName, "Usage"));
        long committed = permGenUsage.getCommitted();//持久堆大小
        long used = heapMemoryUsage.getUsed();//
        System.out.println("perm gen:"+(double)used*100/committed+"%");//持久堆使用率

        // - - - - - - - - - - - - - - - - - - - - Session - - - - - - - - - - - - - - - - - - - -
        ObjectName managerObjName = new ObjectName("Catalina:type=Manager,*");
```

```
        Set<ObjectName> s=mbsc.queryNames(managerObjName, null);
        for (ObjectName obj:s){
            System.out.println("应用名:"+obj.getKeyProperty("path"));
            ObjectName objname=new ObjectName(obj.getCanonicalName());
            System.out.println("最大会话数:"+ mbsc.getAttribute( objname, "maxActiveSessions"));
            System.out.println("会话数:"+ mbsc.getAttribute( objname, "activeSessions"));
            System.out.println("活动会话数:"+ mbsc.getAttribute( objname, "sessionCounter"));
        }

        // - - - - - - - - - - - - - - - - - - Thread Pool - - - - - - - - - - - - - -
        ObjectName threadpoolObjName = new ObjectName("Catalina:type=ThreadPool,*");
        Set<ObjectName> s2=mbsc.queryNames(threadpoolObjName, null);
        for (ObjectName obj:s2){
            System.out.println("端口名:"+obj.getKeyProperty("name"));
            ObjectName objname=new ObjectName(obj.getCanonicalName());
            System.out.println("最大线程数:"+ mbsc.getAttribute( objname, "maxThreads"));
            System.out.println("当前线程数:"+ mbsc.getAttribute( objname, "currentThreadCount"));
            System.out.println("繁忙线程数:"+ mbsc.getAttribute( objname, "currentThreadsBusy"));
        }
        } catch (Exception e) {
            e.printStackTrace();
        }
}

public static String formatTimeSpan(long span){
        long minseconds = span % 1000;
        span = span /1000;
        long seconds = span % 60;
        span = span / 60;
        long mins = span % 60;
        span = span / 60;
        long hours = span % 24;
        span = span / 24;
        long days = span;
        return (new Formatter()).format("%1$d 天 %2$02d:%3$02d:%4$02d.%5$03d", days,hours,mins,seconds,minseconds).toString();
        }
}
```

内存使用调整（Out of Memery 问题）

Tomcat 默认可以使用的内存为 128MB ，在较大型的应用项目中，这点内存是不够的，需要调大。

Windows 下，在文件 {tomcat_home}/bin/catalina.bat ， Unix 下，在文件 {tomcat_home}/bin/catalina.sh 的前面，增加如下设置：
JAVA_OPTS='-Xms【初始化内存大小】 -Xmx【可以使用的最大内存】 '

需要把这个两个参数值调大。例如：
rem ----- Execute The Requested Command ----------
set JAVA_OPTS='-Xms256m -Xmx512m'

表示初始化内存为 256MB ，可以使用的最大内存为 512MB

可用 JDK 附带的 Jconsole 查看 tomcat 的内存使用情况。

连接线程数调整（cannot connect to server 问题）

优化 tomcat 配置：maxThreads="500" minSpareThreads="400" maxSpareThreads="450"。

maxThreads
This is the maximum number of threads allowed. This defines the upper bound to the concurrency, as Tomcat will not create any more threads than this. If there are more than maxThreads requests, they will be queued until the number of threads decreases. Increasing maxThreads increases the capability of Tomcat to handle more connections concurrently. However,threads use up system resources. Thus, setting a very high value might degrade performance, and could even cause Tomcat to crash. It is better to deny some incoming connections, rather that affect the ones that are being currently serviced.

maxSpareThreads
This is the maximum number of idle threads allowed. Any excess idle threads are shut down by Tomcat. Setting this to a large value is not good for performance; the default (50) usually works for most Web sites with an average load. The value of maxSpareThreads should be greater than minSpareThreads, but less than maxThreads.

minSpareThreads
This is the minimum number of idle threads allowed. On Tomcat startup, this is also the number of threads created when the Connector is initialized.If the number of idle threads falls below minSpareThreads, Tomcat creates new threads. Setting this to a large value is not good for performance, as each thread uses up resources. The default (4) usually works for most Web sites with an average load. Typically, sites with “bursty” traffic would need higher values for

minSpareThreads.

Tomcat6 使用默认的配置，在进行压力测试时，老是报错，用 Jmeter 模拟 40 个用户并发时，正常，当超过 60 个时，居然全部报错，打开 server.xml 可以看到如下配置：

```
<Connector port="8080" protocol="HTTP/1.1"
               connectionTimeout="20000"
               redirectPort="8443" />
```

Tomcat 的官方网站的解释如下， maxThread 如果没有 set，默认值为 200.

The maximum number of request processing threads to be created by this Connector , which therefore determines the maximum number of simultaneous requests that can be handled. If not specified, this attribute is set to 200. If an executor is associated with this connector, this attribute is ignored as the connector will execute tasks using the executor rather than an internal thread pool.

这个就产生问题了，打开 Tomcat 的源码发现默认值是 40，而不是 200，所以压力测试时遇到同类的问题，大家一定要注意，添加以下配置：

```
<Connector port="8080" protocol="HTTP/1.1"
               connectionTimeout="20000"
               redirectPort="8443" maxThreads="150"/>
```

Tomcat7 实验：

一、测试 JSP 页面：

http://192.168.1.101:8080/examples/jsp/jsp2/el/basic-arithmetic.jsp

1、默认配置（maxThreads=200）

VU：50

RT：0.072

2、maxThreads=10

VU：50

RT：0.072

3、maxThreads=5

VU：50

RT：0.081

4、maxThreads=1

VU：50

RT：10.423

二、测试 JPetStore 主页面：

http://192.168.1.101:8080/jpetstore/shop/index.shtml

1、maxThreads=5

VU：50

RT：0.355

2、maxThreads=50
VU：50
RT：0.369

3、maxThreads=3
VU：50
RT：0.559

4、默认配置（maxThreads=200）
VU：50
RT：0.390

5、maxThreads=3
VU：50（Rendezvous）
RT：2.609

6、maxThreads=50
VU：50（Rendezvous）
RT：0.597

connectionTimeout

单位是毫秒，Connector 从接受连接到提交 URI 的等待的时间。
http://tomcat.apache.org/tomcat-5.5-doc/config/http.html
The number of milliseconds this Connector will wait, after accepting a connection, for the request URI line to be presented. The default value is 60000 (i.e. 60 seconds).

现象：线上系统中tomcat的连接超时（connectionTimeout）设置成 60ms，造成第三方访问公司的服务，总是 502 异常（502 Bad Gateway）。

这个设置在$tomcat/conf/server.xml 中：

```
<!-- Define a non-SSL HTTP/1.1 Connector on port 8080 -->
    <Connector port="8080" maxHttpHeaderSize="8192"
               maxThreads="150" minSpareThreads="25"
maxSpareThreads="75"
               enableLookups="false" redirectPort="8443"
acceptCount="100"
               connectionTimeout="2000" disableUploadTimeout="true" />
```

是否是整个连接处理的时间？

写了一个 servlet，doGet 先 sleep 一段时间，再写一个输出，直接用浏览器访问。

经过测试，发现和这个时间无关。

用 telnet 收工测试

直接用 telnet 连上 tomcat，如果什么都不输入，socket 很快回断开，输入完整 GET。。。，能够获得输出。如果不保持输入，则连接很快会断开。如果一直不停输入，连接继续保持。

```
$ telnet localhost 8080

GET /index.jsp HTTP/1.1
Accept-Language: zh-cn
Connection: Keep-Alive
Host: 192.168.0.53
Content-Length: 36
```

直接写 Socket 做测试

还是 2s 超时，睡一秒能够正确获得输出，睡 2 秒，输出为空。

以下是 Groovy 代码

```
content =''GET /index.jsp HTTP/1.1
Accept-Language: zh-cn
Connection: Keep-Alive
Host: 192.168.0.53
Content-Length: 36

'''
def sleepTime=1000
Socket socket = new Socket('localhost',8080)

println 'is keep alive? ' + socket.getKeepAlive()

println 'sleep ' + sleepTime + ' ms.'
Thread.sleep(sleepTime)
println 'is closed? ' + socket.isClosed()

println 'sleep ' + sleepTime + ' ms.'
//Thread.sleep(sleepTime)

println 'write socket begin======'
writeStream(content, socket.getOutputStream())

println 'read socket begin======'
println readStream(socket.getInputStream())[0..300]

void writeStream(content, stream) {
```

```
        OutputStream buf = new BufferedOutputStream(stream);
        OutputStreamWriter out = new OutputStreamWriter(buf, "UTF-8");
        out.write(content)
        out.flush();
        print content
        //out.close();
}

String readStream(stream){
        String sResult=''
        byte[] buffer = new byte[1024];

        int readCount = stream.read(buffer);

        while (readCount != -1) {
                sResult += new String(buffer, 0,
                                readCount, "utf-8");
                readCount = stream.read(buffer);
        }
        stream.close()
        return sResult
}
```

acceptCount – 最大排队数

The maximum queue **length** for incoming connection requests when all possible request processing threads are in use. Any requests received when the queue is full will be refused. **The default value is 100**.

配置例子如下：

```
<Connector port="8080" protocol="HTTP/1.1"
                connectionTimeout="20000"
                redirectPort="8443"
                maxThreads="800" acceptCount="1000"/>
```

maxThreads：tomcat 起动的最大线程数，即同时处理的任务个数，默认值为 200

acceptCount：当 tomcat 起动的线程数达到最大时，接受排队的请求个数，默认值为 100

这两个值如何起作用，请看下面三种情况

情况 1：接受一个请求，此时 tomcat 起动的线程数没有到达 maxThreads，tomcat 会起动一个线程来处理此请求。

情况 2：接受一个请求，此时 tomcat 起动的线程数已经到达 maxThreads，tomcat 会把此请求放入等待队列，等待空闲线程。

情况 3：接受一个请求，此时 tomcat 起动的线程数已经到达 maxThreads，等待队列中的请

求个数也达到了 acceptCount，此时 tomcat 会直接拒绝此次请求，返回 connection refused

maxThreads 如何配置
一般的服务器操作都包括量方面：1 计算（主要消耗 cpu），2 等待（io、数据库等）

第一种极端情况，如果我们的操作是纯粹的计算，那么系统响应时间的主要限制就是 cpu 的运算能力，此时 maxThreads 应该尽量设的小，降低同一时间内争抢 cpu 的线程个数，可以提高计算效率，提高系统的整体处理能力。

第二种极端情况，如果我们的操作纯粹是 IO 或者数据库，那么响应时间的主要限制就变为等待外部资源，此时 maxThreads 应该尽量设的大，这样才能提高同时处理请求的个数，从而提高系统整体的处理能力。此情况下因为 tomcat 同时处理的请求量会比较大，所以需要关注一下 tomcat 的虚拟机内存设置和 linux 的 open file 限制。

现实应用中，我们的操作都会包含以上两种类型（计算、等待），所以 maxThreads 的配置并没有一个最优值，一定要根据具体情况来配置。
最好的做法是：在不断测试的基础上，不断调整、优化，才能得到最合理的配置。
acceptCount 的配置，我一般是设置的跟 maxThreads 一样大，这个值应该是主要根据应用的访问峰值与平均值来权衡配置的。
如果设的较小，可以保证接受的请求较快相应，但是超出的请求可能就直接被拒绝
如果设的较大，可能就会出现大量的请求超时的情况，因为我们系统的处理能力是一定的。

**web server 允许的最大连接数还受制于操作系统的内核参数设置，通常 Windows 是 2000 个左右，Linux 是 1000 个左右。*

CentOS 修改 Socke 最大连接数
查看
ulimit -a
中的 open files
CentOS 默认是 1024

修改
vi /etc/security/limits.conf
追加
* soft nofile 32768
* hard nofile 32768
保存，重启系统，就生效了。
说明：
* 对所有用户有效
soft 可以超过的配置数
hard 最大不能超过的配置数
nofile 对描述符的配置

试验:

修改/etc/security/limits.conf：
* soft nofile 200
* hard nofile 200

200 个 VU 并发（设置集合点），出现连接失败的错误：
Action.c(6): Error -27796: Failed to connect to server "192.168.1.101:8080": [10060] Connection timed out

在 Controller 的 Connection 和 Connection per Second 图中看到连接数偏少（相比起默认设置的 1024）：
1、200
TR:19.664
Connections:177.376
Connections per Second-New Connection：28.082
Connections per Second-Connection Shutdown：28.082
2、1024（默认设置）
TR:2.712
Connections:315.136
Connections per Second-New Connection：114.000
Connections per Second-Connection Shutdown：114.000

Tomcat Native library

如果不下载 Tomcat Native library，则启动 Tomcat 时会提示：
2011-7-3 23:59:16 org.apache.catalina.core.AprLifecycleListener init
信息: **The APR based Apache Tomcat Native library which allows optimal performanc e in production environments was not found on the java.library.path**: E:\Program Files\Java\jdk1.6.0_13\bin;.;E:\WINDOWS\Sun\Java\bin;E:\WINDOWS\system32;...

Linux 下可用如下命令查看到上述提示：
head　logs/catalina.out

所谓的Apache Tomcat Native library其实叫APR，全称为：**Apache Portable Runtime and Tomcat**。可以通过下面地址访问：http://tomcat.apache.org/tomcat-5.5-doc/apr.html

Tomcat can use the Apache Portable Runtime to provide superior scalability, performance, and better integration with native server technologies. The Apache Portable Runtime is a highly portable library that is at the heart of Apache HTTP Server 2.x. APR has many uses, including access to advanced IO functionality (such as sendfile, epoll and OpenSSL), OS level functionality (random number generation, system status, etc), and native process handling (shared memory, NT pipes and Unix sockets).

These features allows making Tomcat a general purpose webserver, will enable much better integration with other native web technologies, and overall make Java much more viable as a full fledged webserver platform rather than simply a backend focused technology.

Apache Tomcat Native library 是 Apache 为了提升 Tomcat 的性能搞的一套本地化 Socket, Thread, IO 组件也就是说它有高级 IO 功能, 操作系统级别的功能调用, 以及本地进程处理等等, 这些都能使 Tomcat 更像一个 Web Server(像 Apache 那样), 而不是只能用来解释 JSP, 也就是说提升单独的 Tomcat 作为服务器的吞吐性能.

Windows 下安装 APR：

用于 Windows 的 APR 是一个名称为：tcnative-1.dll 的文件。下载匹配版本的 tcnative-1.dll：
http://www.apache.org/dist/tomcat/tomcat-connectors/native/1.1.20/binaries/win32/
放到<$JAVA_HOME>/bin 目录下

启动 Tomcat 可看到：
Loaded APR based Apache Tomcat Native library 1.1.20

Linux 下安装 APR：

1. 安装 APR
   http://apr.apache.org 下载 apr-1.2.12.tar.gz
   tar –xvf apr-1.2.12.tar.gz
   cd apr-1.2.12
   ./configure --prefix=/tomcat/apr
   make
   make install
2. 安装 APR-UTIL
   http://apr.apache.org 下载 apr-util-1.2.12.tar.gz
   tar –xvf apr-util-1.2.12.tar.gz
   cd apr-util-1.2.12
   ./configure --prefix=/tomcat/apr --with-apr=/tomcat/apr
   make
   make install
3. 安装 tomcat native library
   cd /usr/local/tomcat/bin
   tar zxvf tomcat-native.tar.gz
   cd tomcat-native-1.1.10-src/jni/native
   ./configure --prefix=/tomcat/apr -- with-apr=/tomcat/apr –with-java-home=/usr/jdk
   make
   make install
4. 编辑 tomcat/bin/catalina.sh

将 JAVA_OPTS="$CATALINA_OPTS -Djava.library.path= tomcat/apr/lib"
加在# ----- Execute The Requested Command ----------前面

5、添加环境变量
vi /etc/profile
添加：export LD_LIBRARY_PATH=/usr/local/apr/lib

6. 重启 tomcat 查看结果
vim /usr/local/tomcat/logs/catalina.out
信息: Loaded Apache Tomcat Native library 1.1.10.
2008-7-8 10:20:27 org.apache.catalina.core.AprLifecycleListener init
信息: APR capabilities: IPv6 [true], sendfile [true], accept filters [false], random [true].
2008-7-8 10:20:27 org.apache.coyote.http11.Http11AprProtocol init

Connector

Tomcat 从 5.5 版本开始，支持以下四种 Connector 的配置分别为：

1）

```
<Connector port="8081"
protocol="org.apache.coyote.http11.Http11NioProtocol"
              connectionTimeout="20000" redirectPort="8443"/>
```

2）

```
<Connector port="8081" protocol="HTTP/1.1"
connectionTimeout="20000"
               redirectPort="8443"/>
```

3）

```
<Connector executor="tomcatThreadPool"
               port="8081" protocol="HTTP/1.1"
               connectionTimeout="20000"
               redirectPort="8443" />
```

4）

```
<Connector executor="tomcatThreadPool"
               port="8081"
protocol="org.apache.coyote.http11.Http11NioProtocol"
```

```
connectionTimeout="20000"
redirectPort="8443" />
```

我们姑且把上面四种 Connector 按照顺序命名为 NIO, HTTP, POOL, NIOP
为了不让其他因素影响测试结果，我们只对一个很简单的 jsp 页面进行测试，这个页面仅仅是输出一个 Hello World。假设地址是 http://tomcat1/test.jsp
我们依次对四种 Connector 进行测试，测试的客户端在另外一台机器上用 ab 命令来完成，测试命令为： ab -c 900 -n 2000 http://tomcat1/test.jsp ，最终的测试结果如下表所示(单位: 平均每秒处理的请求数)：

NIO HTTP POOL NIOP
281 65 208 365
666 66 110 398
692 65 66 263
256 63 94 459
440 67 145 363

由这五组数据不难看出，HTTP 的性能是很稳定，但是也是最差的，而这种方式就是 Tomcat 的默认配置。NIO 方式波动很大，但没有低于 280 的，NIOP 是在 NIO 的基础上加入线程池，可能是程序处理更复杂了，因此性能不见得比 NIO 强；而 POOL 方式则波动很大，测试期间和 HTTP 方式一 样，不时有停滞。
由于 linux 的内核默认限制了最大打开文件数目是 1024，因此此次并发数控制在 900。

Tomcat 6.X 实现了 JCP 的 Servlet 2.5 和 JSP2.1 的规范，并且包括其它很多有用的功能，使它成为开发和部署 web 应用和 web 服务的坚实平台。

NIO （No-blocking I/O）从 JDK 1.4 起，NIO API 作为一个基于缓冲区，并能提供非阻塞 I/O 操作的 API 被引入。

作为开源 web 服务器的 java 实现，tomcat 几乎就是 web 开发者开发、测试的首选，有很多其他商业服务器的开发者也会优先选择 tomcat 作为开发时候使用，而在部署的时候，把应用发布在商业服务器上。也有许多商业应用部署在 tomcat 上，tomcat 承载着其核心的应用。但是很多开发者很迷惑，为什么在自己的应用里使用 tomcat 作为平台的时候，而并发用户超过一定数量，服务器就变的非常繁忙，而且很快就出现了 connection refuse 的错误。但是很多商业应用部署在 tomcat 上运行却安然无恙。

其中有个很大的原因就是，配置良好的 tomcat 都会使用 APR(Apache Portable Runtime),APR 是 Apache HTTP Server2.x 的核心，它是高度可移植的本地库，它使用高性能的 UXIN I/O 操

作，低性能的 java io 操作，但是 APR 对很多 Java 开发者而言可能稍稍有点难度，在很多 OS 平台上，你可能需要重新编译 APR。但是从 Tomcat6.0 以后, Java 开发者很容易就可以是用 NIO 的技术来提升 tomcat 的并发处理能力。

但是为什么 NIO 可以提升 tomcat 的并发处理能力呢，我们先来看一下 java 传统 io 与 java NIO 的差别。

Java 传统的 IO 操作都是阻塞式的(blocking I/O), 如果有 socket 的编程基础，你会接触过堵塞 socket 和非堵塞 socket，堵塞 socket 就是在 accept、read、write 等 IO 操作的的时候，如果没有可用符合条件的资源，不马上返回，一直等待直到有资源为止。而非堵塞 socket 则是在执行 select 的时候，当没有资源的时候堵塞，当有符合资源的时候，返回一个信号，然后程序就可以执行 accept、read、write 等操作，一般来说，如果使用堵塞 socket，通常我们通常开一个线程 accept socket，当读完这次 socket 请求的时候，开一个单独的线程处理这个 socket 请求；如果使用非堵塞 socket，通常是只有一个线程，一开始是 select 状，当有信号的时候可以通过 可以通过多路复用（Multiplexing)技术传递给一个指定的线程池来处理请求，然后原来的线程继续 select 状态。 最简单的多路复用技术可以通过 java 管道(Pipe)来实现。换句话说，如果客户端的并发请求很大的时候，我们可以使用少于客户端并发请求的线程数来处理这些请求，而这些来不及立即处理的请求会被阻塞在 java 管道或者队列里面，等待线程池的处理。请求 听起来很复杂，在这个架构当道的 java 世界里，现在已经有很多优秀的 NIO 的架构方便开发者使用，比如 Grizzly,Apache Mina 等等，如果你对如何编写高性能的网络服务器有兴趣，你可以研读这些源代码。

简单说一下，在 web 服务器上阻塞 IO(BIO)与 NIO 一个比较重要的不同是，我们使用 BIO 的时候往往会为每一个 web 请求引入多线程，每个 web 请求一个单独的线程，所以并发量一旦上去了，线程数就上去了，CPU 就忙着线程切换，所以 BIO 不合适高吞吐量、高可伸缩的 web 服务器；而 NIO 则是使用单线程(单个 CPU)或者只使用少量的多线程(多 CPU)来接受 Socket，而由线程池来处理堵塞在 pipe 或者队列里的请求.这样的话，只要 OS 可以接受 TCP 的连接，web 服务器就可以处理该请求。大大提高了 web 服务器的可伸缩性。

使用 NIO 连接器，只需要在 server.xml 里把 HTTP Connector 做如下更改：

```
<Connector port="8080" protocol="HTTP/1.1"
           connectionTimeout="20000"
           redirectPort="8443" />
```

改为

```
<Connector port="8080" protocol="org.apache.coyote.http11.Http11NioProtocol"
           connectionTimeout="20000"
           redirectPort="8443" />
```

然后启动服务器，在启动日志中会看到 org.apache.coyote.http11.Http11NioProtocol start 的信息，表示 NIO 已经启动。

---

*AJP 连接器

*由于 tomcat 的 html 和图片解析功能相对其他服务器如 apche 等较弱，所以，一般都是集成起来使

用，只有 jsp 和 servlet 服务交由 tomcat 处理，而 tomcat 和其他服务器的集成，就是通过 **ajp** 协议来完成的。

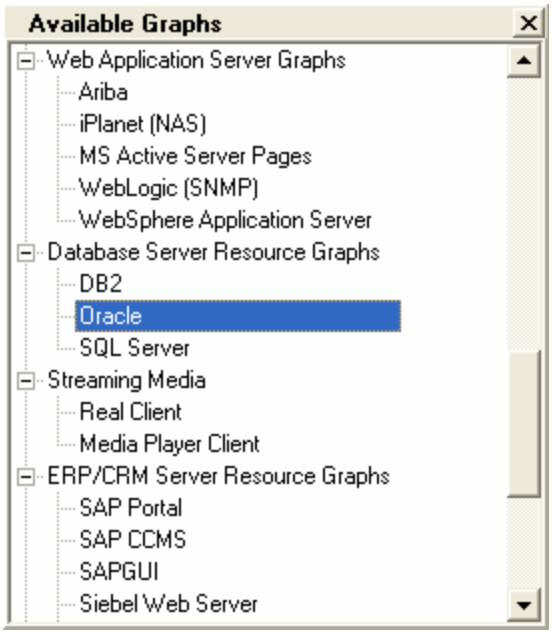
oracle 性能监控、诊断分析与调优

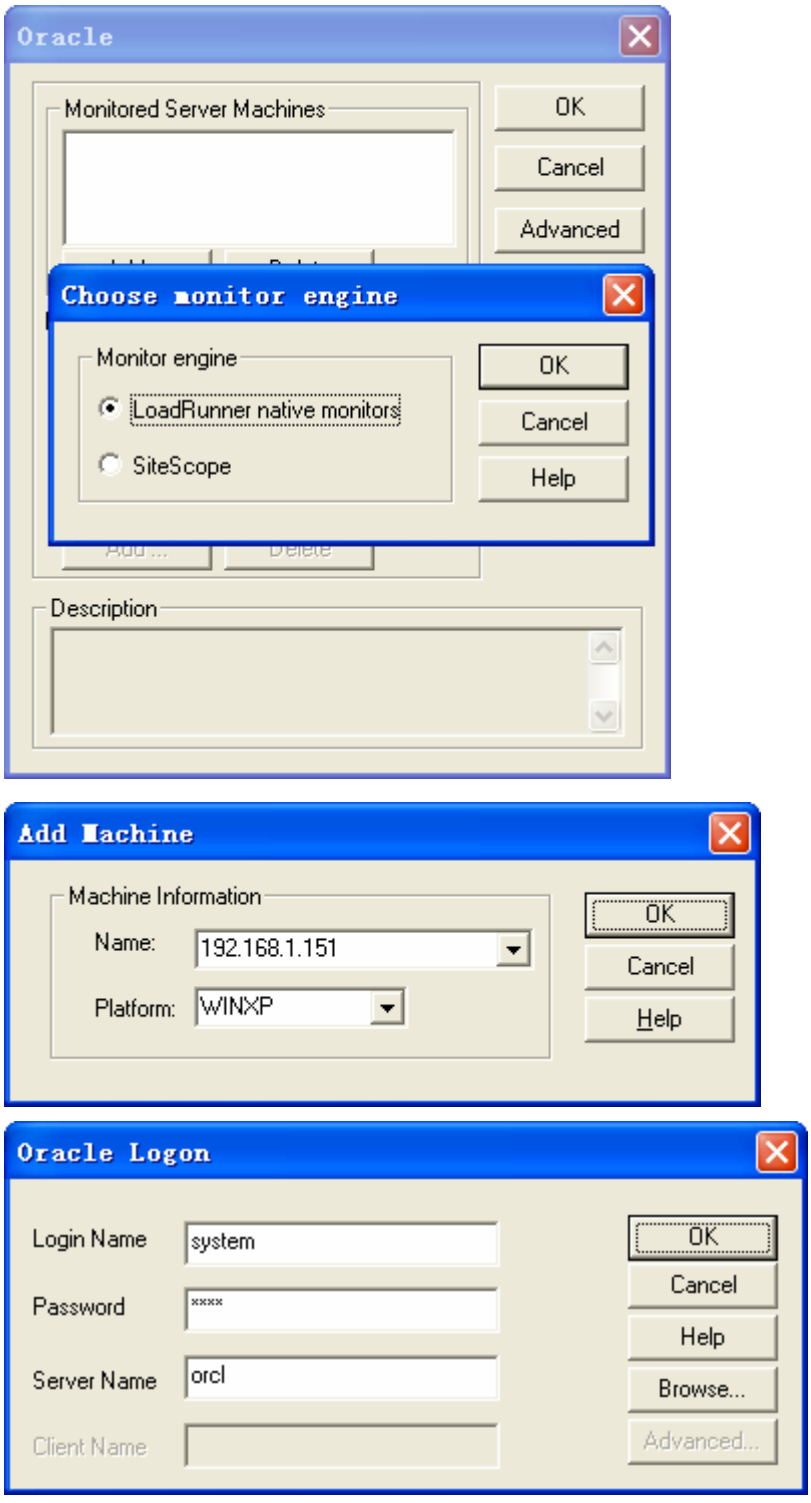
D:\PrefTest\Oracle 调优\Oracle 性能诊断与调优学习笔记.doc

监控方法

方法 1：在 LR 的 Controller 中配置监视 Oracle

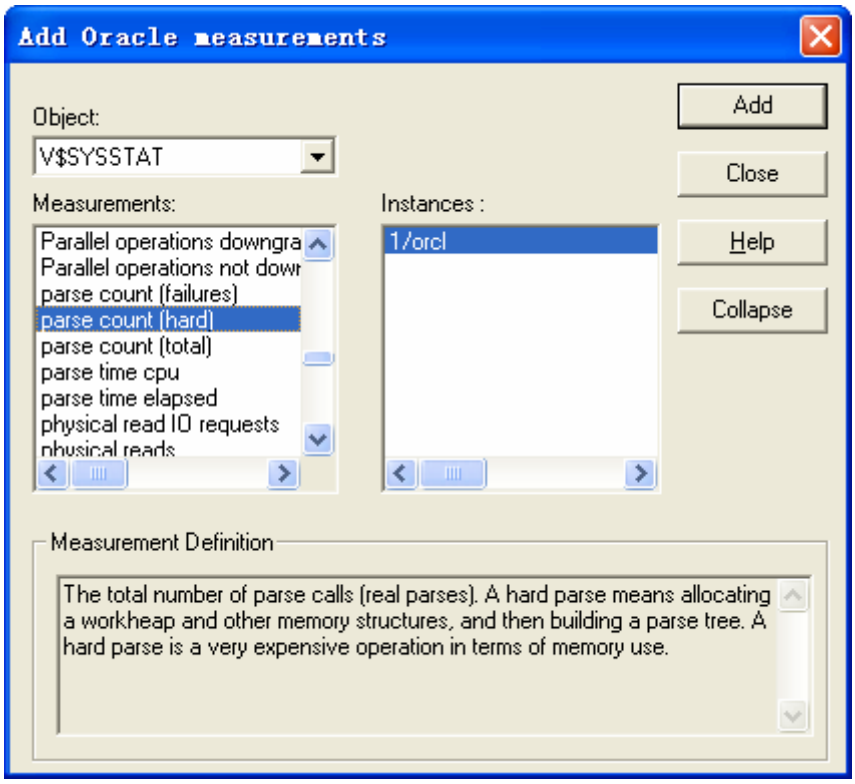
1、在 Controller 所在的机器上安装 Oracle 客户端
2、配置好服务名，用 sqlplus 确认可以连接 Oracle
3、在 Controller 中配置 Oracle 连接：

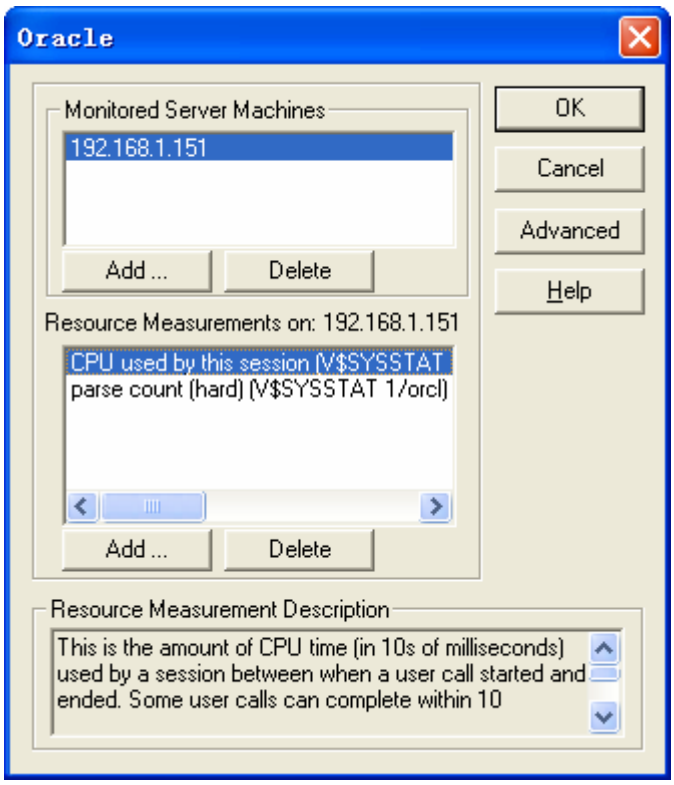


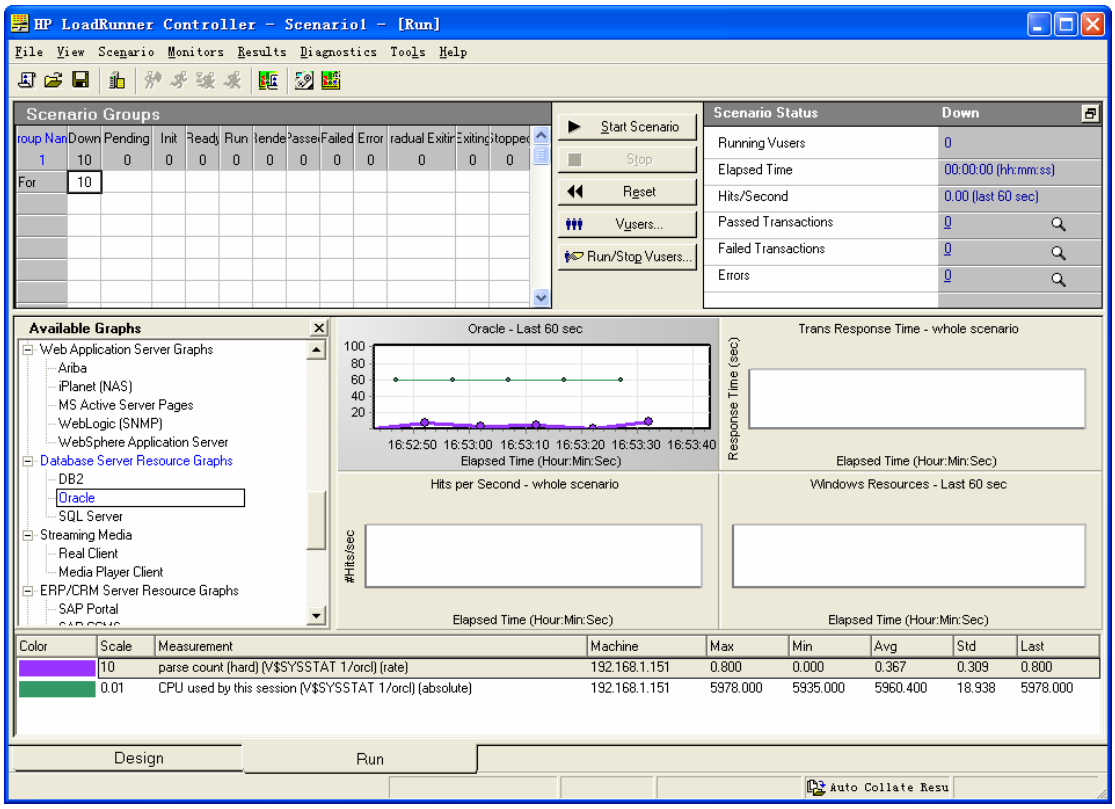

Oracle
Monitored Server Machines
OK
Cancel
Advanced
Choose monitor engine
Monitor engine
LoadRunner native monitors
SiteScope
OK
Cancel
Help
Description

Add Machine
Machine Information
Name:
192.168.1.151
Platform:
WINXP
OK
Cancel
Help

Oracle Logon
Login Name
system
Password
Server Name
orcl
Client Name
OK
Cancel
Help
Browse...
Advanced...


Add Oracle measurements
Object:
V$SYSSTAT
Measurements:
Parallel operations downgra
Parallel operations not dowr
parse count (failures)
parse count (hard)
parse count (total)
parse time cpu
parse time elapsed
physical read IO requests
physical reads
Instances :
1/orcl
Add
Close
Help
Collapse
Measurement Definition
The total number of parse calls (real parses). A hard parse means allocating a workheap and other memory structures, and then building a parse tree. A hard parse is a very expensive operation in terms of memory use.


Oracle
Monitored Server Machines
192.168.1.151
Add ...
Delete
Resource Measurements on: 192.168.1.151
CPU used by this session [V$SYSSTAT
parse count (hard) [V$SYSSTAT 1/orcl]
Add ...
Delete
OK
Cancel
Advanced
Help
Resource Measurement Description
This is the amount of CPU time (in 10s of milliseconds) used by a session between when a user call started and ended. Some user calls can complete within 10



修改 sample rate：

To change the length of each monitoring sample (in seconds), edit the **dat\monitors\vmon.cfg** file in the LoadRunner root folder. The default rate is 10 seconds.

The minimum sampling rate for the Oracle Monitor is 10 seconds. If you set the sampling rate at less than 10 seconds, the Oracle Monitor will continue to monitor at 10 second intervals.

添加自定义计数器：

在 LoadRunner 安装路径的\dat\monitors 找到 vmon.cfg 文件并修改：

[V$ Monitor]
Counters=150
CustomCounters=9
;How many seconds for each data sample?
SamplingRate=10

[Custom0]
;Name must be unique
Name=库快存命中率
Description=该计数器返回当前库快存命中率
Query=SELECT 100*((sum(pins-reloads))/sum(pins)) from v$librarycache

IsRate=0

[Custom1]
;Name must be unique
Name=高速缓存区命中率
Description=oracle database shoot straight
Query=SELECT round(1-SUM(PHYSICAL_READS)/(SUM(DB_BLOCK_GETS) + SUM(CONSISTENT_GETS)), 4) * 100 FROM (SELECT CASE WHEN NAME='physical reads' THEN VALUE END PHYSICAL_READS,CASE WHEN NAME = 'db block gets' THEN VALUE END DB_BLOCK_GETS,CASE WHEN NAME = 'consistent gets' THEN VALUE END CONSISTENT_GETS FROM V$SYSSTAT WHERE Name IN ('physical reads','db block gets','consistent gets'))
IsRate=0

[Custom2]
;Name must be unique
Name=共享区库缓存区命中率
Description=命中率应大于 0.99
Query=Select round(sum(pins-reloads)/sum(pins) * 100, 2) from v$librarycache
IsRate=0

[Custom3]
;Name must be unique
Name=共享区字典缓存区命中率
Description=命中率应大于 0.85
Query=Select round(sum(gets-getmisses-usage-fixed)/sum(gets) * 100, 2) from v$rowcache
IsRate=0

[Custom4]
;Name must be unique
Name=检测回滚段的争用
Description=应该小于 1%
Query=select round(sum(waits)/sum(gets) * 100, 2) from v$rollstat
IsRate=0

[Custom5]
;Name must be unique
Name=检测回滚段收缩次数
Description=应该小于 1%
Query=select sum(shrinks) from v$rollstat, v$rollname where v$rollstat.usn = v$rollname.usn
IsRate=0

[Custom6]
;Name must be unique

Name=监控表空间的 I/O 读总数
Description=监控表空间的 I/O
Query=select sum(f.phyrds) pyr from v$filestat f, dba_data_files df where f.file# = df.file_id
IsRate=0

[Custom7]
;Name must be unique
Name=监控表空间的 I/O 块读总数
Description=监控表空间的 I/O
Query=select sum(f.phyblkrd) pbr from v$filestat f, dba_data_files df where f.file# = df.file_id
IsRate=0

[Custom8]
;Name must be unique
Name=监控表空间的 I/O 写总数
Description=监控表空间的 I/O
Query=select sum(f.phywrts) pyw from v$filestat f, dba_data_files df where f.file# = df.file_id
IsRate=0

方法 2：使用 SiteScope

根据 sitescope 帮助文档，需要把 oracle 的 jdbc 驱动 classes12.zip 放到<SiteScope root directory>\WEB-INF\lib 和 <SiteScope root directory>\java\lib\ext。尝试连接测试，成功。 另需注意的是：数据库用户名必须是 DBA。

方法 3：使用 Oracle 企业管理器查看数据库性能

方法 4：使用 Spotlight

参考：

D:\PrefTest\Oracle调优\Spotlight For Oracle使用说明.pdf

计数器

sorts(disk)(V$SYSSTAT)

If the number of disk writes is non-zero for a given sort operation, then this statistic is incremented. Sorts that require I/O to disk are quite resource intensive. Try increasing the size of the initialization parameter SORT_AREA_SIZE. For more information, see "SORT_AREA_SIZE".

sorts (memory) and sorts (disk)：sorts(memory)是在SORT_AREA_SIZE(因此不需要在磁盘进行排序)的排序操作的数量。sorts(disk)则是由于排序所需空间太大，SORT_AREA_SIZE不能满足而不得不在磁盘进行排序操作的数量。这两项统计通常用于计算in-memory sort ratio。

sort(memory)(V$SYSSTAT)

If the number of disk writes is zero, then the sort was performed completely in memory and this statistic is incremented. This is more an indication of sorting activity in the application work load. You cannot do much better than memory sorts, except maybe no sorts at all. Sorting is usually caused by selection criteria specifications within table join SQL operations.

*Select * from v$sysstat where name like '%sort%';*
- Sort(disk):要求IO去临时表空间的排序数目
- Sort(memory)：完全在memory中完成的排序数目
- Sort(rows)：被排序的行数合计

In-memory sort ratio：该项显示内存中完成的排序所占比例。最理想状态下，在OLTP系统中，大部分排序不仅小并且能够完全在内存里完成排序。
公式：sorts (memory) / (sorts (memory) + sorts (disk))
执行：
```
select a.value/(b.value+c.value)
  from v$sysstat a,v$sysstat b,v$sysstat c
  where a.name='sorts (memory)' and
            b.name='sorts (memory)' and c.name='sorts (disk)';
```

对于要做大量排序操作的SQL语句的执行（例如select * from tt order by 1,2,3,4;），可监控到sort(disk)和sort(memory)都会有所上升。性能好的话，应该是大部分排序在内存中进行。
实验：
```
create table tt as select * from all_objects;
commit;
Set timing on;
select * from tt order by 1,2,3,4;
```

排序诊断和优化：
```
SELECT disk.Value disk,mem.Value mem,(disk.Value/mem.Value)*100 ratio
FROM v$sysstat disk,v$sysstat mem
WHERE mem.NAME='sorts (memory)' AND disk.NAME='sorts (disk)';
```

Sort(disk)/ Sort(memory)<5%，如果超过5%，增加sort_area_size的值(调整PGA)。

查询PGA统计信息：
SELECT * FROM v$pgastat;
查看bytes processed、extra bytes read/written的增量值和cache hit percentage的值
如果cache hit percentage偏低，则要考虑调整PGA

PGA
（Program Global Area程序全局区）是一块包含一个服务进程的数据和控制信息的内存区

域。它是Oracle在一个服务进程启动时创建的，是非共享的。一个Oracle进程拥有一个PGA内存区。一个PGA也只能被拥有它的那个服务进程所访问，只有这个进程中的Oracle代码才能读写它。因此，PGA中的结构是不需要Latch保护的。

调整PGA优化排序：
首先查看Oracle的v$pga_target_advice：

```
SELECT ROUND(pga_target_for_estimate/1024/1024) AS target_mb,
      estd_pga_cache_hit_percentage AS hit_ratio,
      estd_overalloc_count
FROM v$pga_target_advice
ORDER BY target_mb;
```

然后调整PGA

```
alter system set pga_aggregate_target=150M;
```

*在OLTP（联机事务处理）系统中，典型PGA内存设置应该是总内存的较小部分（例如20％），剩下80％分配给SGA。
OLTP：**PGA_AGGREGATE_TARGET = (total_mem * 80%) * 20%**
*在DSS（数据仓库）系统中，由于会运行一些很大的查询，典型的PGA内存最多分配70％的内存。
DSS：**PGA_AGGREGATE_TARGET = (total_mem * 80%) * 50%**

db block gets (V$SYSSTAT)

Number of blocks accessed in buffer cache for INSERT, UPDATE, DELETE, and SELECT FOR UPDATE. Represent block logical reads (from cache).The logical reads ALWAYS include the physical reads. Low number of physical reads is preferable.

在Oracle的文档中有这样一段解释：
db block gets：Number of times a CURRENT block was requested.
consistent gets：Number of times a consistent read was requested for a block.
physical reads：Total number of data blocks read from disk. This number equals the value of "physical reads direct" plus all reads into buffer cache.

针对以上3个概念进行的说明解释及关系如下：
1、DB Block Gets（当前请求的块数目）
当前模式块意思就是在操作中正好提取的块数目，而不是在一致性读的情况下而产生的块数。正常的情况下，一个查询提取的块是在查询开始的那个时间点上存在的数据块，当前块是在这个时刻存在的数据块，而不是在这个时间点之前或者之后的数据块数目。

2、Consistent Gets（数据请求总数在回滚段Buffer中的数据一致性读所需要的数据块）
这里的概念是在处理你这个操作的时候需要在一致性读状态上处理多少个块，这些块产生的主要原因是因为由于在你查询的过程中，由于其他会话对数据块进行操 作，而对所要查询的块有了修改，但是由于我们的查询是在这些修改之前调用的，所以需要对回滚段中的数据

块的前映像进行查询，以保证数据的一致性。这样就产 生了一致性读。

3、Physical reads（物理读）
就是从磁盘上读取数据块的数量，其产生的主要原因是：
1)、 在数据库高速缓存中不存在这些块
2)、 全表扫描
3)、 磁盘排序

它们三者之间的关系大致可概括为：
逻辑读指的是Oracle从内存读到的数据块数量。一般来说是'consistent gets' + 'db block gets'。
当在内存中找不到所需的数据块的话就需要从磁盘中获取，于是就产生了'physical reads'。

Buffer Cache Hit Ratio诊断：
查看oracle缓存的命中率（应该大于90％）

```
select 1 - ((physical.value - direct.value - lobs.value) / logical.value)   "Buffer Cache Hit Ratio"
  from v$sysstat physical,v$sysstat direct,v$sysstat lobs,v$sysstat logical
     where physical.name = 'physical reads'
        and direct.name='physical reads direct'
        and lobs.name='physical reads direct (lob)'
        and logical.name='session logical reads';
```

该项显示buffer cache大小是否合适

Buffer Cache优化：
通常在OLTP下，hit ratio应该高于0.9，否则如果低于0.9则需要增加buffer cache的大小。在考虑调整buffer cache hit ratio时，需要注意：如果上次增加buffer cache的大小以后，没有对提高hit ratio产生很大效果的话，不要盲目增加buffer cache的大小以提高性能。因为对于排序操作或并行读，oracle是绕过buffer cache进行的。在调整buffer cache时，尽量避免增加很多的内存而只是提高少量hit ratio的情况出现。

查看buffer cache size：

```
show parameter _size
```

db_16k_cache_size
db_2k_cache_size
db_32k_cache_size
db_4k_cache_size
db_8k_cache_size
db_block_size
db_cache_size
db_keep_cache_size

parse count (hard)(V$SYSSTAT)

Total number of parse calls (real parses). A hard parse is a very expensive operation in terms of memory use, because it requires Oracle to allocate a
workheap and other memory structures and then build a parse tree.Should be minimized. The ratio of Hard Parse to Total should be less than 20%.

parse count (hard)：在shared pool中解析调用的未命中次数。当sql语句执行并且该语句不在shared pool或虽然在shared pool但因为两者存在部分差异而不能被使用时产生硬解析。如果一条sql语句原文与当前存在的相同，但查询表不同则认为它们是两条不同语句，则硬解析即会发生。硬解析会带来cpu和资源使用的高昂开销，因为它需要oracle在shared pool中重新分配内存，然后再确定执行计划，最终语句才会被执行。

parse count (total)：解析调用总数，包括软解析和硬解析。当session执行了一条sql语句，该语句已经存在于shared pool并且可以被使用则产生软解析。当语句被使用(即共享)所有数据相关的现有sql语句(如最优化的执行计划)必须同样适用于当前的声明。这两项统计可被用于计算软解析命中率。

Soft parse ratio：这项将显示系统是否有太多硬解析。该值可与原始统计数据对比以确保精确。例如，软解析率仅为0.2则表示硬解析率太高，不过，如果总解析量(parse count total)偏低，这项值可以被忽略。
公式：1 - (parse count (hard) / parse count (total))

```
select 1 - (a.value/b.value)
  from v$sysstat a,v$sysstat b
    Where a.name='parse count (hard)' and b.name='parse count (total)';
```

The parse process includes the following phases（解析过程包括以下阶段）：
Checking that the SQL statement is syntactically valid (that is, that the SQL conforms to the rules of the SQL language, and that all keywords and operators are valid and correctly used).

Checking that the SQL is semantically valid. This means that all references to database objects (such as tables and columns) are valid.

Checking security (that is, that the user has permission to perform the specified SQL operations on the objects involved).

Determining an execution plan for the SQL statement. The execution plan describes the series of steps that Oracle performs in order to access and update the data involved.
Parsing can be an expensive operation. Its overhead is often masked by the greater overhead of high I/O requirements. However, eliminating unnecessary parsing is always desirable.

The parse/execute ratio reflects the ratio of parse calls to execute calls. Because parsing is an expensive operation, it is better to parse statements once and then execute them many times. High

parse ratios (greater than 20%) can result from the following circumstances:

If literals, rather than bind variables, are used as query parameters, the SQL must be re-parsed on every execution. You should use bind variables whenever possible, unless there is a pressing reason for using column histograms.

Some development tools or techniques result in SQL cursors being discarded after execution. If a cursor is discarded, then the parse is required before the statement can be re-executed.

Oracle 对 sql 的处理过程：

当你发出一条 sql 语句交付 Oracle，在执行和获取结果前，Oracle 对此 sql 将进行几个步骤的处理过程：

1、语法检查(syntax check)

检查此 sql 的拼写是否语法。

2、语义检查(semantic check)

诸如检查 sql 语句中的访问对象是否存在及该用户是否具备相应的权限。

3、对 sql 语句进行解析(prase)

利用内部算法对 sql 进行解析，生成解析树(parse tree)及执行计划(execution plan)。

4、执行 sql，返回结果(execute and return)

其中，软、硬解析就发生在第三个过程里。

Oracle 利用内部的 hash 算法来取得该 sql 的 hash 值，然后在 library cache 里查找是否存在该 hash 值；

假设存在，则将此 sql 与 cache 中的进行比较；

假设“相同”，就将利用已有的解析树与执行计划，而省略了优化器的相关工作。这也就是软解析的过程。

诚然，如果上面的 2 个假设中任有一个不成立，那么优化器都将进行创建解析树、生成执行计划的动作。这个过程就叫硬解析。

创建解析树、生成执行计划对于 sql 的执行来说是开销昂贵的动作，所以，应当极力避免硬解析，尽量使用软解析。

这就是在很多项目中，倡导开发设计人员对功能相同的代码要努力保持代码的一致性，以及要在程序中多使用绑定变量的原因。

查找不能被充分共享利用的SQL语句（查询LibraryCache中执行次数偏低的SQL语句）：

```
SELECT sql_text FROM v$sqlarea WHERE executions < 5 ORDER BY UPPER(sql_text);
```

查找SQL执行次数和SQL解析次数（hard parse），对比两个值的差：

```
SELECT sql_text , parse_calls , executions FROM v$sqlarea ORDER BY parse_calls;
```

查询**v$librarycache**视图的**Reloads**值（reparsing）的值，值应该接近0，否则应该考虑调整shared pool size,通过调整Shared Pool来调整Library Cache，**invalidations**的值也应该接近0

```
select namespace,gethitratio,pinhitratio,reloads,invalidations from v$librarycache;
```

```
SELECT gethitratio FROM v$librarycache WHERE namespace = 'SQL AREA';
```

SELECT sql_text , users_executing , executions , loads FROM v$sqlarea;
进一步查询该SQL的完整信息：
SELECT * FOM v$sqltext WHERE sql_text LIKE 'SELECT * FROM hr.employees WHERE %'

诊断：
1） 检查v$librarycache中sql area的gethitratio是否超过90%，如果未超过90%，应该检查应用代码，提高应用代码的效率：
Select gethitratio from v$librarycache where namespace='SQL AREA';
2) v$librarycache中reloads/pins的比率应该小于1%，如果大于1%，应该增加参数shared_pool_size的值:
Select sum(pins) "executions", sum(reloads) "cache misses",sum(reloads)/sum(pins) from v$librarycache;
reloads/pins>1%有两种可能，一种是library cache空间不足，一种是sql中引用的对象不合法。
3)查看某个session的hard parse个数：
select a.sid,a.value from v$sesstat a,v$session b ,v$statname c where a.sid=b.sid and a.statistic#=c.statistic# and a.sid = 137 and c.name='parse count (hard)';

调整 Library Cache（通过调整 Shared Pool 来调整）：
SELECT shared_pool_size_for_estimate AS **pool_size**,**estd_lc_size**,**estd_lc_time_saved** FROM **v$shared_pool_advice**;
- ESTD_LC_SIZE 估计librarycache的大小值
- ESTD_LC_TIME_SAVED 在当前指定共享池的大小中找到库缓存对象所节省的时间（秒）

调优：
1、书写程序时尽量使用变量，不要过多的使用常量
实验：
创建表格
SQL>CREATE TABLE m(x int);
创建存储过程proc1，使用绑定变量
SQL>CREATE OR REPLACE PROCEDURE proc1
AS
BEGIN
  FOR i IN 1..10000
  LOOP
Execute immediate
  'INSERT INTO m VALUES(:x)' USING i;
  END LOOP;
END;
/
创建存储过程proc2，不使用绑定变量
SQL>CREATE OR REPLACE PROCEDURE proc2
AS
BEGIN

```
   FOR i IN 1..10000
   LOOP
Execute immediate
   'INSERT INTO m VALUES('||i||')' ;
   END LOOP;
END;
/
```

执行proc2和proc1，对比执行效率

```
SQL>SET TIMING ON
SQL> exec proc2;
PL/SQL procedure successfully completed.
Elapsed: 00:00:08.93
SQL> select count(*) from m;
   COUNT(*)
----------
        10000
Elapsed: 00:00:00.01
SQL> TRUNCATE TABLE m;
Table truncated.
Elapsed: 00:00:01.76
SQL> exec proc1;
PL/SQL procedure successfully completed.
Elapsed: 00:00:01.85
SQL> select count(*) from m;
   COUNT(*)
----------
        10000
Elapsed: 00:00:00.00
```

2、修改cursor_sharing参数为similar，让类似的SQL语句不做hard parse

有时候我们的应用程序没有使用绑定变量，而修改程序可能有点困难，我们可能需要设置CURSOR_SHARING=SIMILAR来强制ORACLE使用绑定变量。

```
Show parameter cursor
Alter system set cursor_sharing=SIMILAR
```

实验：

```
SQL> show parameter cursor_sharing
  cursor_sharing string EXACT
SQL> select * from test where object_id=1;
no rows selected
SQL> select sql_text,parse_calls from v$sqlarea where sql_text like 'select * from test%';
select * from test where object_id=:"SYS_B_0" 2
select * from test where object_id=1 1
SQL> alter system flush shared_pool;
System altered.
```

SQL> alter system flush shared_pool;
System altered.
SQL> alter session set cursor_sharing=similar; ----second
Session altered.
SQL> select * from test where object_id=1;
no rows selected
SQL> select sql_text,parse_calls from v$sqlarea where sql_text like 'select * from test%';
select * from test where object_id=:"SYS_B_0" 1
SQL> select * from test where object_id=2;
no rows selected
SQL> select sql_text,parse_calls from v$sqlarea where sql_text like 'select * from test%';
select * from test where object_id=:"SYS_B_0" 2

CPU used by this session(V$SYSSTAT)

Amount of CPU time (in tens of milliseconds) used by a session from the time a user call starts until it ends. If a user call completes within 10 milliseconds, the start- and end-user call times are the same for purposes of this statistic, and 0 milliseconds are added

Parse CPU to total CPU ratio：该项显示总的CPU花费在执行及解析上的比率。如果这项比率较低，说明系统执行了太多的解析。
公式：1 - (parse time cpu / CPU used by this session)
执行：

```
select 1-(a.value/b.value)
  from v$sysstat a,v$sysstat b
     where a.name='parse time cpu' and
           b.name='CPU used by this session';
```

V$SYSSTAT shows Oracle CPU usage for all sessions. The statistic "CPU used by this session" shows the aggregate CPU used by all sessions.

V$SESSTAT shows Oracle CPU usage per session. You can use this view to determine which particular session is using the most CPU.

If you can, determine why the processes use so much CPU time and attempt to tune them. Possible areas to research include, but are not limited to, the following:
Reparsing SQL Statements
Read Consistency
Scalability Limitations Within the Application
Wait Detection
Latch Contention

诊断：

找使用CPU多的用户session：

select a.sid,spid,status,substr(a.program,1,40) prog,a.terminal,osuser,value/60/100 value
from v$session a,v$process b,v$sesstat c
where c.statistic#=12 and c.sid=a.sid and a.paddr=b.addr order by value desc;

*12是cpu used by this session

再找出使用CPU多的SQL语句：

查找指定SPID正在执行的SQL语句：

SELECT P.pid pid,S.sid sid,P.spid spid,S.username username,S.osuser osname,P.serial# S_#,P.terminal,P.program program,P.background,S.status,RTRIM(SUBSTR(a.sql_text, 1, 80)) SQL
FROM v$process P, v$session S,v$sqlarea A
WHERE P.addr = s.paddr AND S.sql_address = a.address (+) AND P.spid LIKE '%&1%';

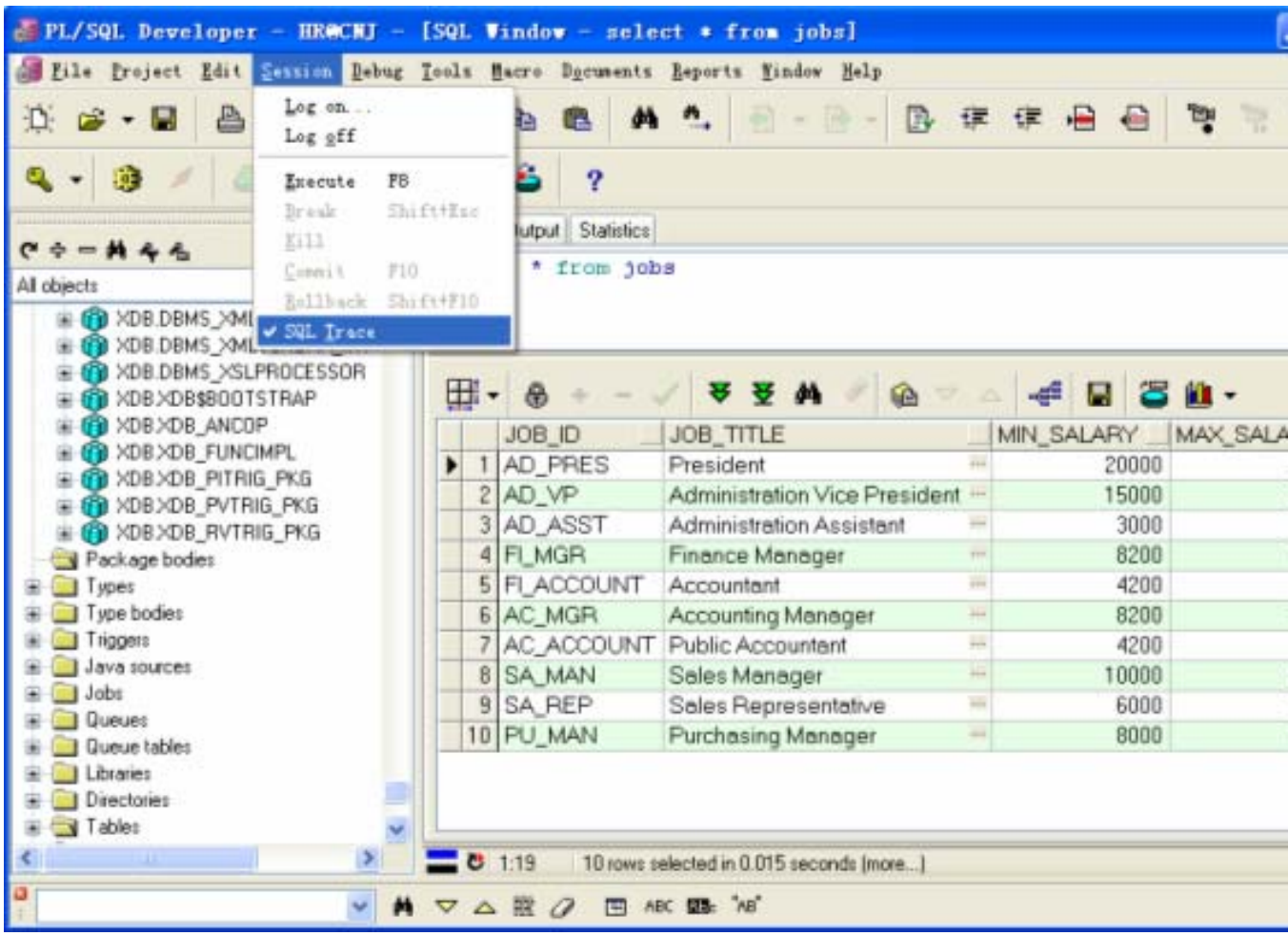
*在linux环境可以通过ps查看进程信息，包括pid,windows中任务管理器的PID与v$process中pid不能一一对应。windows是多线程服务器,每个进程包含一系列线程。这点与unix等不同，Unix每个Oralce进程独立存在，在NT上所有线程由Oralce进程衍生。

或者指定SID查看正在执行的SQL语句：

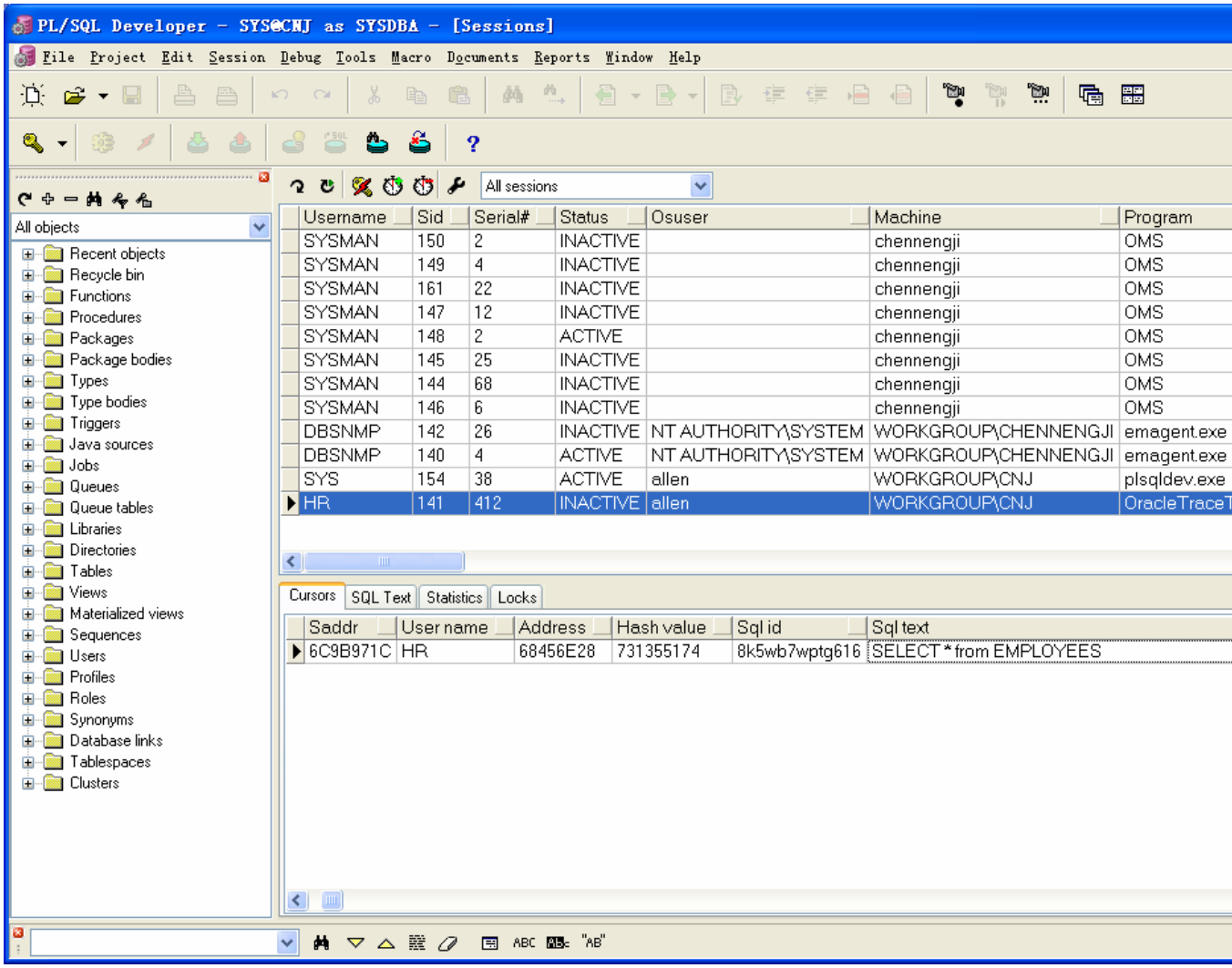
SELECT P.pid pid,S.sid sid,P.spid spid,S.username username,S.osuser osname,P.serial# S_#,P.terminal,P.program program,P.background,S.status,RTRIM(SUBSTR(a.sql_text, 1, 80)) SQL
FROM v$process P, v$session S,v$sqlarea A
WHERE P.addr = s.paddr AND S.sql_address = a.address (+) AND s.sid = '136';

跟踪诊断和优化 SQL 语句

PL/SQL Developer：



指定跟踪某个Session的SQL语句：



修改全局参数 sql_trace，产生 Trace 文件（少用）：

修改自己的 Session，产生 Trace 文件：
ALTER SESSION SET sql_trace=TRUE;

针对指定 Session 产生 Trace 文件（以下操作需要在 SYS 用户下进行）：
（1）首先使用以下语句从 Oracle 的 v$session 表查出所有跟 Oracle 连接的客户端程序的进程：

```
select SID, SERIAL#, USERNAME,OSUSER,MACHINE,TERMINAL,PROGRAM from v$Session
```

（2）然后利用 DBMS_SYSTEM.SET_SQL_TRACE_IN_SESSION 包来跟踪特定的进程向 Oracle 提交的 SQL 语句。
例如，下面语句为 SID 为 13，SERIAL#为 96 的进程执行 SQL 语句的跟踪功能：

```
execute dbms_system.set_sql_trace_in_session(13,96,TRUE);
```

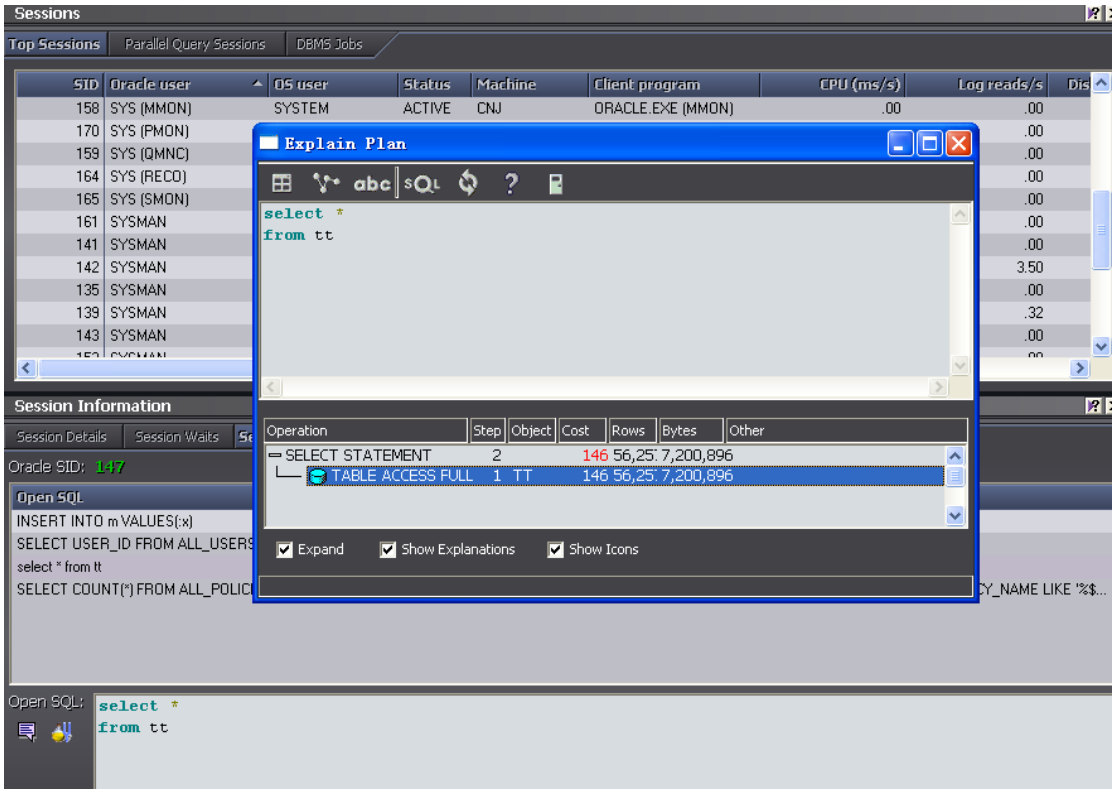
（3）如果要停止跟踪，则提交以下语句：

```
execute dbms_system.set_sql_trace_in_session(13,96,FALSE);
```

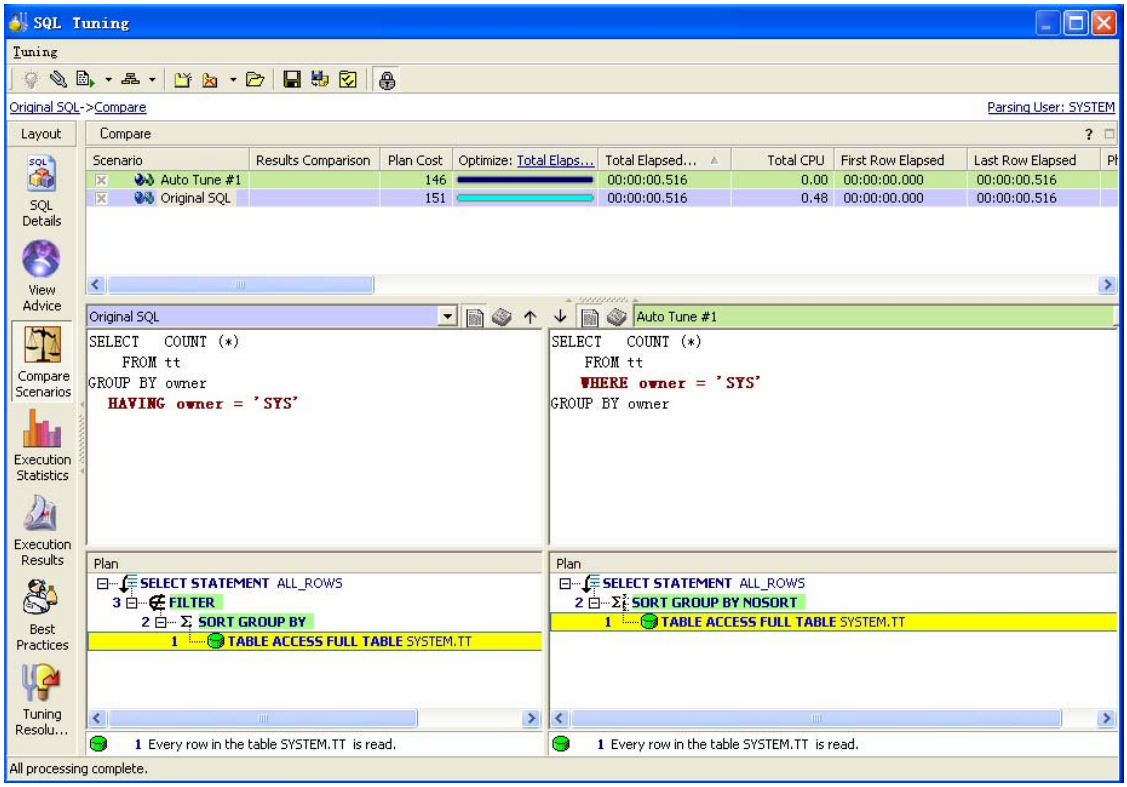
tkprof：

```
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\allen>tkprof
Usage: tkprof tracefile outputfile [explain= ] [table= ]
              [print= ] [insert= ] [sys= ] [sort= ]
  table=schema.tablename   Use 'schema.tablename' with 'explain=' option.
  explain=user/password    Connect to ORACLE and issue EXPLAIN PLAN.
  print=integer    List only the first 'integer' SQL statements.
  aggregate=yes|no
  insert=filename  List SQL statements and data inside INSERT statements.
  sys=no           TKPROF does not list SQL statements run as user SYS.
  record=filename  Record non-recursive statements found in the trace file.
  waits=yes|no     Record summary for any wait events found in the trace file.
  sort=option      Set of zero or more of the following sort options:
    prscnt  number of times parse was called
    prscpu  cpu time parsing
    prsela  elapsed time parsing
    prsdsk  number of disk reads during parse
    prsqry  number of buffers for consistent read during parse
    prscu   number of buffers for current read during parse
    prsmis  number of misses in library cache during parse
    execnt  number of execute was called
    execpu  cpu time spent executing
    exeela  elapsed time executing
    exedsk  number of disk reads during execute
    exeqry  number of buffers for consistent read during execute
    execu   number of buffers for current read during execute
    exerow  number of rows processed during execute
    exemis  number of library cache misses during execute
    fchcnt  number of times fetch was called
    fchcpu  cpu time spent fetching
    fchela  elapsed time fetching
    fchdsk  number of disk reads during fetch
    fchqry  number of buffers for consistent read during fetch
    fchcu   number of buffers for current read during fetch
    fchrow  number of rows fetched
    userid  userid of user that parsed the cursor
```

在Spotlight中跟踪执行session的SQL语句，并查看执行计划：



还可以利用SQL Tuning的调优建议功能：



参考：

D:\PrefTest\案例\oracle性能调优\SQL优化.doc

Oracle 索引问题诊断与优化

Usage of Functions（在 where 关键字后要尽量避免使用函数）
Using functions in the where clause suppresses the use of indexes.

SELECT * FROM dwtable2 WHERE to_number(empno)=783;
Elapsed: 00:00:00.15

This query could be written more efficiently as:
SELECT * FROM dwtable2 WHERE empno=783;
Elapsed: 00:00:00.01

实验
create table s1 as select * from SH.SALES;
create table s2 as select * from SH.SALES;

s1 表没有建立索引
s2 表有建立索引

set timing on;
select * from s1 where prod_id=1;
2.45s
select * from s2 where prod_id=1;
0.59s

可见索引对于表查询速度的重要性。

索引性能测试与诊断：
1、查看数据库 Index 信息：

```
SELECT A.OWNER, A.TABLE_OWNER, A.TABLE_NAME, A.INDEX_NAME,
A.INDEX_TYPE,
            B.COLUMN_POSITION, B.COLUMN_NAME, C.TABLESPACE_NAME,
            A.TABLESPACE_NAME, A.UNIQUENESS
      FROM DBA_INDEXES A, DBA_IND_COLUMNS B, DBA_TABLES C
     WHERE A.OWNER = UPPER ('hr')
       AND A.OWNER = B.INDEX_OWNER
       AND A.OWNER = C.OWNER
       AND A.TABLE_NAME LIKE UPPER ('DEPARTMENTS')
       AND A.TABLE_NAME = B.TABLE_NAME
       AND A.TABLE_NAME = C.TABLE_NAME
       AND A.INDEX_NAME = B.INDEX_NAME
ORDER BY A.OWNER, A.TABLE_OWNER, A.TABLE_NAME, A.INDEX_NAME,
```

B.COLUMN_POSITION

2、查出没有建立 index 的表：

```
SELECT OWNER, TABLE_NAME
   FROM ALL_TABLES
 WHERE OWNER NOT IN ('SYS','SYSTEM','OUTLN','DBSNMP') AND OWNER = UPPER ('scott')
MINUS
SELECT OWNER, TABLE_NAME
   FROM ALL_INDEXES
 WHERE OWNER NOT IN ('SYS','SYSTEM','OUTLN','DBSNMP')
```

3、查出建立了过量 index 的表：

```
SELECT      OWNER, TABLE_NAME, COUNT (*) "count"
   FROM ALL_INDEXES
     WHERE OWNER NOT IN ('SYS','SYSTEM','OUTLN','DBSNMP') AND OWNER = UPPER ('hr')
GROUP BY OWNER, TABLE_NAME
   HAVING COUNT (*) > ('4')
```

一个表可以有几百个索引，但是对于频繁插入和更新表，索引越多系统 CPU，I/O 负担就越重；建议每张表不超过 5 个索引。

实验：

```
create table table1 as select * from SH.SALES;
create table table2 as select * from SH.SALES;
```

table1 只在 prod_id 列建索引
table2 在所有列建索引

```
SELECT count(*) FROM table1 where prod_id=30;
29282

set timing on;
update table1 set cust_id=1 where prod_id=30;
10.56s
update table2 set cust_id=1 where prod_id=30;
11.35s
```

4、找出全表扫描（Full Scan）的 Sid 和 SQL

The following query reports how many full table scans are taking place:

```
SELECT name, value
FROM v$sysstat
WHERE name LIKE '%table %'
```

ORDER BY name;

The values relating to the full table scans are:
table scans (long tables) - a scan of a table that has more than five database blocks
table scans (short tables) - a count of full table scans with five or fewer blocks
If the number of long table scans is significant, there is a strong possibility that SQL statements in your application need tuning or indexes need to be added.

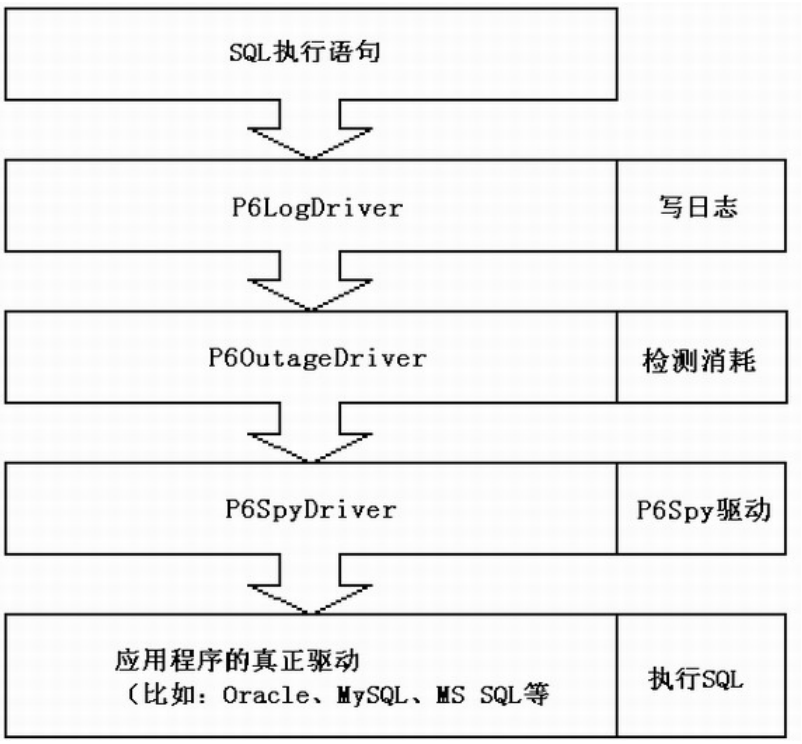
If you can identify the users who are experiencing the full table scans, you can find out what they were running to cause these scans. Below is a script that allows you to do this:

```
    SELECT          ss.username
                || '('
                || se.sid
                || ') ' "User Process",
SUM (DECODE (NAME, 'table scans (short tables)', VALUE)) "Short Scans",
SUM (DECODE (NAME, 'table scans (long tables)', VALUE)) "Long Scans",
SUM (DECODE (NAME, 'table scan rows gotten', VALUE)) "Rows Retrieved"
              FROM v$session ss, v$sesstat se, v$statname sn
             WHERE se.statistic# = sn.statistic#
               AND ( NAME LIKE '%table scans (short tables)%'
                     OR NAME LIKE '%table scans (long tables)%'
                     OR NAME LIKE '%table scan rows gotten%'
                     )
               AND se.sid = ss.sid
               AND ss.username IS NOT NULL
      GROUP BY ss.username
                     || '('
                     || se.sid
                     || ') ';
```

参考：
D:\PrefTest\案例\oracle 性能调优\Oracle 索引问题诊断与优化

p6spy 监控和跟踪 SQL 语句

参考：
D:\PrefTest\案例\p6spy 监控\



1、把 P6Spy 的 jar 包 p6spy.jar 放到 CLASSPATH 中，如果是 Web 应用程序则放在 YourWebApp/WEB-INF/lib/ 目录下；

2、把 spy.properties 放到 CLASSPATH 目录下，如果是 Web 应用程序放在 YourWebApp/WEB-INF/classess/ 目录下，注意不是 lib/ 目录

3、修改你应用系统中的数据库驱动名称为 P6Spy 的驱动程序名称 com.p6spy.engine.spy.P6SpyDriver 其它的全部使用默认值，暂时先都不用修改；

4、打开配置文件 spy.properties 文件，找到 realdriver，把它的值改为你的应用系统的真正的数据库驱动名称；

5、运行你的应用程序或 Web 应用程序，可以在 tomcat 的 bin 目录下的 spy.log 里看到 P6Spy 监测到的 SQL 详细的执行与操作的记录信息了，包含有完整的 SQL 执行参数。

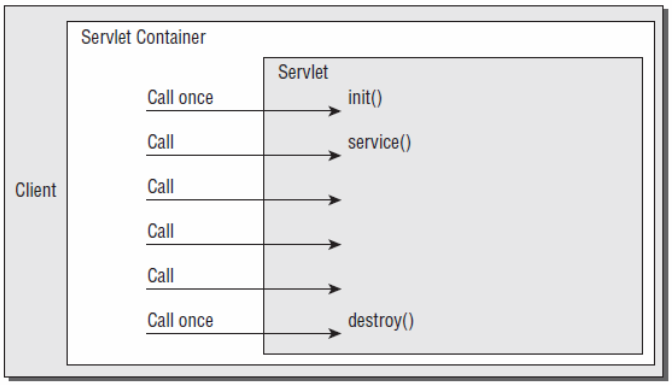
JSP、Servlet 性能优化

D:\PrefTest\案例\servlet 性能优化

Servlet 常见性能问题分析与优化

Servlet 中利用 init()方法进行高速缓存

当应用服务器初始化 servlet 实例之后，为客户端请求提供服务之前，它会调用这个 servlet 的 init()方法。在一个 servlet 的生命周期中，init()方法只会被调用一次。如图所示：



通过在 init()方法中缓存一些静态的数据或完成一些只需要执行一次的、耗时的操作，就可大大地提高系统性能，例如下面的 Servlet 代码：

```
public class TestInit2 extends HttpServlet {
    String driverName = "com.microsoft.jdbc.sqlserver.SQLServerDriver";  //加载 JDBC 驱动
    String dbURL = "jdbc:sqlserver://localhost:1433; DatabaseName=Northwind";  //连接服务器和数据库
    String userName = "sa";  //默认用户名
    String userPwd = "";  //密码
    Connection dbConn;
    Statement stmt = null;
    ResultSet rs = null;

    public TestInit2() {
        super();
    }

    public void destroy() {
        super.destroy();
        try {
            dbConn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
```

```
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        try {
            rs = stmt.executeQuery( "select * from Northwind.dbo.Orders;" ); //得到结果集
            while(rs.next()){//遍历结果集
                out.println(" ");
                out.println(" " + rs.getString("orderid") + " ");//取出列值
                out.println(" ");
            }
            out.println(" ");

        } catch (Exception e) {
            e.printStackTrace();
            out.println("Connection Fail!" + e.toString());
        }
        out.flush();
        out.close();
    }

    public void init() throws ServletException {
        try {
            Class.forName(driverName);
            try {
                dbConn = DriverManager.getConnection(dbURL, userName, userPwd);
            } catch (SQLException e) {
                e.printStackTrace();
            }
            stmt = dbConn.createStatement();//创建 statement
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

在 init 和 destroy 函数中处理数据库连接建立和关闭操作。试验结果表明，用 Init 比不用 Init 要快点：0.465 VS. 0.591。

没有使用 init 的 Servlet 代码如下所示，在 doGet 中连接数据库、执行查询语句，这样每次请求 Servlet 都将执行这些代码，耗费大量资源：

```
    public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
```

```
        String driverName = "com.microsoft.jdbc.sqlserver.SQLServerDriver";  //加载 JDBC 驱动
        String dbURL = "jdbc:sqlserver://localhost:1433; DatabaseName=Northwind";  //连接服务器和数据库
        String userName = "sa";  //默认用户名
        String userPwd = "";  //密码
        Connection dbConn;
        Statement stmt = null;
        ResultSet rs = null;

        try {
            Class.forName(driverName);
            dbConn = DriverManager.getConnection(dbURL, userName, userPwd);

            stmt = dbConn.createStatement();//创建 statement
            rs = stmt.executeQuery( "select * from Northwind.dbo.Orders;" ); //得到结果集
            while(rs.next()){//遍历结果集
                out.println(" ");
                out.println(" " + rs.getString("orderid") + " ");//取出列值
                out.println(" ");
            }
            out.println(" ");

            dbConn.close();
           } catch (Exception e) {
            e.printStackTrace();
            out.println("Connection Fail!" + e.toString());
        }

        out.flush();
        out.close();
    }
```

我们还可在 Jconsole 中观察 JVM 的内存使用情况，没用 Init 的话，内存逐渐上升，用了 Init 的话，内存使用比较平稳。

Servlet 压缩输出

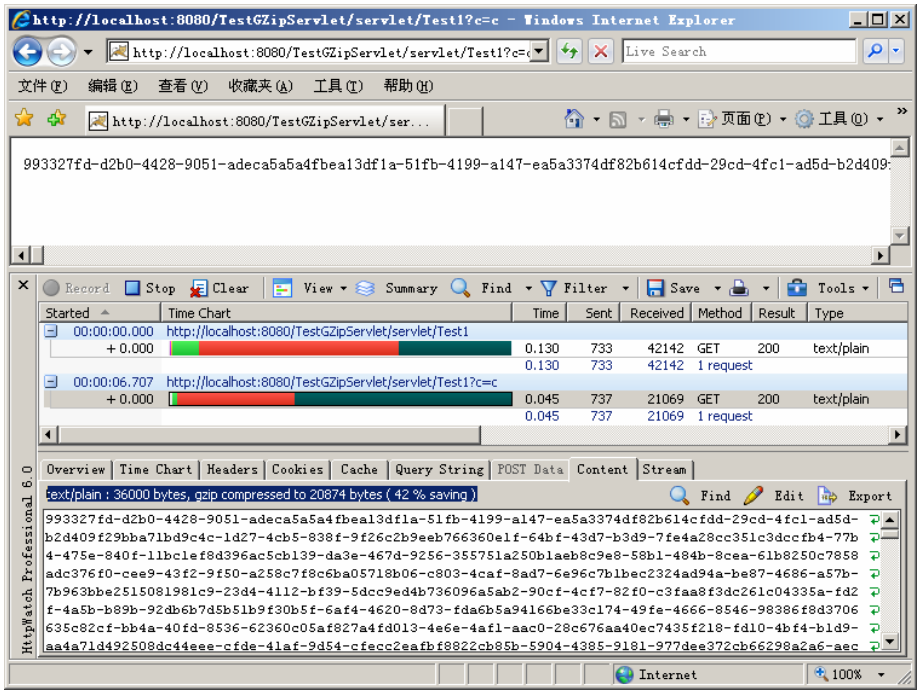
HTTP 压缩可以大大提高浏览网站的速度，它的原理是：在客户端请求网页后，从服务器端将网页文件压缩，再下载到客户端，由客户端的浏览器负责解压缩后再解析呈现。HTTP 压缩大概可以节省 40%左右的流量。

在 Servlet 中，可以编写压缩过滤器，对输出的页面内容进行压缩处理，例如下面的代码：

```
        String c = request.getParameter("c");
```

```
if ("c".equals(c)) { // 压缩输出
        response.setHeader("Content-Encoding" , "gzip");
        CompressionResponse cResponse = new CompressionResponse(response);
        response.setContentType("text/plain");
        final OutputStream out = cResponse.getOutputStream();
        out.write("Compress!".getBytes());
        for ( int i=0; i<1000; i++ ) {
                out.write(UUID.randomUUID().toString().getBytes());
                out.flush();
        }
        out.close();
}
else { // 不压缩输出
        response.setContentType("text/plain");
        PrintWriter out = response.getWriter();
        out.print("NOT Compress!");
        for ( int i=0; i<1000; i++ ) {
                out.print(UUID.randomUUID().toString());
                out.flush();
        }
        out.close();
}
```

在 HTTPWatch 中观察压缩前跟压缩后数据传输量和传输速度的差异，可以看到不压缩输出的传输时间需要 0.130 秒，压缩输出所需要的时间是 0.045，压缩大概降低了一半的传输数据量，从而大大提高了传输速度：



需要注意 WEB 容器在处理压缩时需要占有较多内存，原始响应被复制到 JVM 内存中，然后

运行压缩算法，算法本身需要动用一定量的内存来创建并处理中间数据。

JSP 常见性能问题分析与优化

选择正确的页面包含机制

在 JSP 中有两种方法可以用来包含另一个页面：
（1）使用 jsp 包括指令（include directive）
<%@ include file="test.jsp" %>
（2）使用 jsp 包括动作（include action）
<jsp:include page= "test.jsp" flush="true"/>
一般而言，使用第一种方法的话，可以使得系统性能更高。因为第一种是在编译阶段执行 include，将与主 jsp 的 Servlet 融合在一起。而第二种则是在请求处理阶段执行。包括指令在编译阶段包括一个指定文件的内容，例如，当一个页面编译成一个 servlet 时。包括动作是指在请求阶段包括文件内容，例如，当一个用户请求一个页面时。包括指令要比包括动作快些，因此除非被包括的文件经常变动，否则使用包括指令将会获得更好的性能。

屏蔽 Page Session

JSP 引擎默认会为 JSP 页面创建 Session 对象。如果不需要使用 HttpSession 的话，把 session 属性设置为 false。这样可避免创建 session 对象，降低内存占有、减少垃圾回收，从而提高性能。
例如，我们用 MyEclipse 创建一个简单的 WEB 项目，添加一个最简单的 JSP 页面：

```
<%@ page language="java" import="java.util.*" pageEncoding="ISO-8859-1"%>
<%
String path = request.getContextPath();
String basePath = request.getScheme()+"://"+request.getServerName()+":"+request.getServerPort()+path+"/";
%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <base href="<%=basePath%>">

    <title>My JSP 'MyJsp.jsp' starting page</title>

    <meta http-equiv="pragma" content="no-cache">
    <meta http-equiv="cache-control" content="no-cache">
    <meta http-equiv="expires" content="0">
    <meta http-equiv="keywords" content="keyword1,keyword2,keyword3">
```

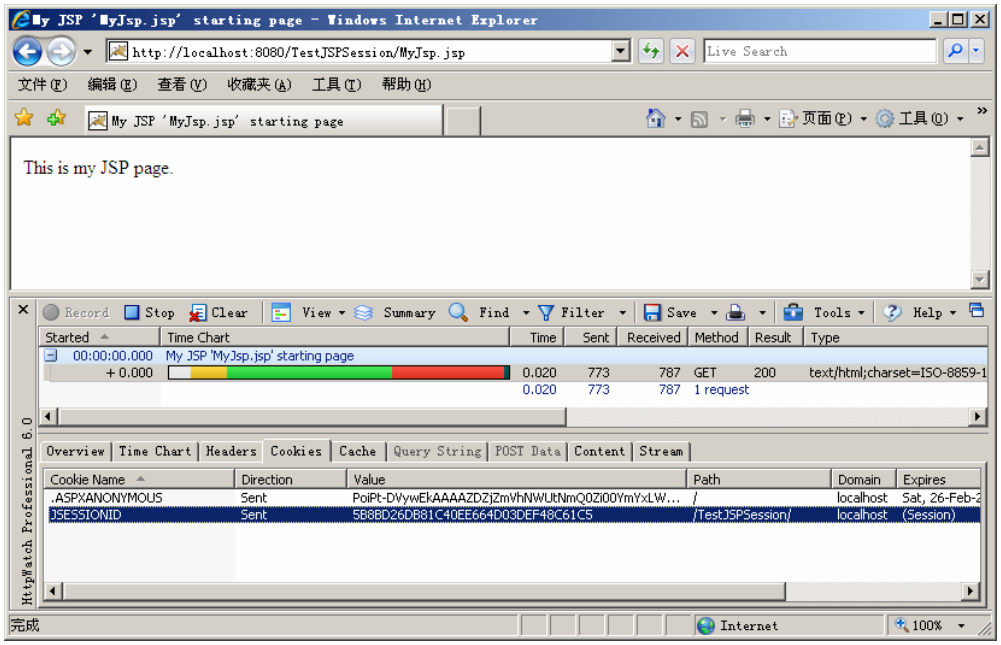
```
    <meta http-equiv="description" content="This is my page">
    <!--
    <link rel="stylesheet" type="text/css" href="styles.css">
    -->

  </head>

  <body>
    This is my JSP page. <br>
  </body>
</html>
```

默认不会设置<%@ page session="false"%>，也就是说默认创建的 JSP 页面是创建 session 的，为了证实这一点，我们可以把这个简单的 WEB 项目部署到 Tomcat 中，用浏览器访问这个 JSP 页面。

在 Tomcat 的临时目录（<*tomcat*>\work\Catalina\localhost\TestJSPSession\org\apache\jsp）可以找到 MyJsp_jsp.java 文件，打开这个临时文件可以看到“_jspService”函数中包含了 HttpSession 对象创建的代码，例如：

```
  public void _jspService(HttpServletRequest request, HttpServletResponse response)
        throws java.io.IOException, ServletException {
    JspFactory _jspxFactory = null;
    PageContext pageContext = null;
    HttpSession session = null;
    ServletContext application = null;
    ServletConfig config = null;
    JspWriter out = null;
    Object page = this;
    JspWriter _jspx_out = null;
    PageContext _jspx_page_context = null;
    try {
      _jspxFactory = JspFactory.getDefaultFactory();
      response.setContentType("text/html;charset=ISO-8859-1");
      pageContext = _jspxFactory.getPageContext(this, request, response,
                null, true, 8192, true);
      _jspx_page_context = pageContext;
      application = pageContext.getServletContext();
      config = pageContext.getServletConfig();
      session = pageContext.getSession();
      out = pageContext.getOut();
      _jspx_out = out;
      out.write('\r');
      out.write('\n');
…
```

在浏览器端用 HTTPWatch 访问页面并截获 HTTP 包进行查看，可发现 JSESSIONID 被保存到 Cookies 中了，如图所示：



如果在 JSP 页面中添加代码：

<%@ page session="false"%>

那么 JSP 引擎就不会自动创建和发送 Session 对象，但是由于默认不添加 page session 属性的话，这个值是默认为 true 的，因此我们如果漏了添加这个代码的话，往往会造成大量 Session 对象的创建，占有内存，影响性能。

正确地确定 javabean 的生命周期

JSP 的一个强大的地方就是对 javabean 的支持。通过在 JSP 页面中使用＜jsp:useBean＞标签，可以将 javabean 直接插入到一个 JSP 页面中。它的使用方法如下：

<jsp:useBean id="name" scope="page|request|session|application" class="package.className" type="typeName"></jsp:useBean>

其中 scope 属性指出了这个 bean 的生命周期。缺省的生命周期为 page。如果你没有正确地选择 bean 的生命周期的话，它将影响系统的性能。

举例来说，如果你只想在一次请求中使用某个 bean，但你却将这个 bean 的生命周期设置成了 session，那当这次请求结束后，这个 bean 将仍然保留在内存中，除非 session 超时或用户关闭浏览器。这样会耗费一定的内存，并无谓的增加了 JVM 垃圾收集器的工作量。因此为 bean 设置正确的生命周期，并在 bean 的使命结束后尽快地清理它们，会使用系统性能有一个提高。

下面简要介绍 scope 的 4 种设置所对应的 bean 生命周期范围：

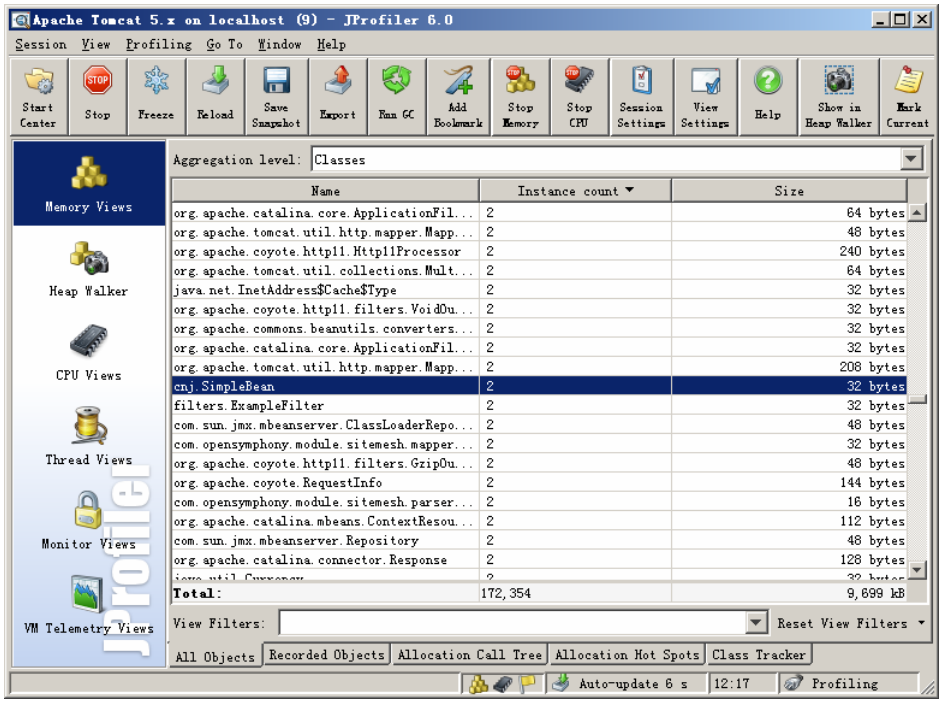
（1）page - 在包含<jsp:useBean>元素的 JSP 文件以及此文件中的所有静态包含文件中使用 Bean,直到页面执行完毕向客户端发回响应或转到另一个文件为止。

（2）request - 在任何执行相同请求的 Jsp 文件中使用 Bean,直到页面执行完毕向客户端发回响应或转到另一个文件为止。可以使用 Request 对象访问 Bean，比如 request.getAttribute(beanInstanceName)。

（3）session - 从创建 Bean 开始，就能在任何使用相同 session 的 Jsp 文件中使用 Bean。这个 Bean 存在于整个 Session 生存周期内，任何在分享此 Session 的 Jsp 文件都能使用同一

Bean。注意在创建 Bean 的 Jsp 文件中<% @ page %>指令中必须指定 session=true。
（4）application - 从创建 Bean 开始，就能在任何使用相同 application 的 Jsp 文件中使用 Bean。这个 Bean 存在于整个 application 生存周期 内，任何在分享此 application 的 Jsp 文件都能使用同一 Bean。

为了加深对 javabean 使用的生命周期的理解，可自行编写 JSP 页面引用 Bean，用不同的 scope 指出这个 bean 的生命周期，然后用 JProfiler 等工具查看对象在内存中的存活周期。例如图中显示的是分别访问了采用 4 种范围引用 bean 的 JSP 页面，按 F4 回收对象后的情况：



可以看到 Application 和 Session 的作用范围和生命周期与其在 JVM 内存中的存活时间是对应的，如果某些 Bean 本应及时撤销，但是由于被引用而无法进行回收，则会造成内存的占有，如果存在大量这类对象则造成内存泄漏，影响性能。

控制 Session 的时间

如果在 JSP 中使用 Session 对象，那么就需要了解清楚 Session 的时间范围，以便控制好 Session 的存活时间。一般而言，应用服务器引擎会有默认的 Session 超时的设置，例如 Tomcat 默认设置为 30 分钟，可在 conf 文件夹的 web.xml 文件中设置：

```
<!-- ==================== Default Session Configuration ================ -->
<!-- You can set the default session timeout (in minutes) for all newly     -->
<!-- created sessions by modifying the value below.                         -->
    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>
```

这意味着，如果在 30 分钟内不使用 session 或移除 session，应用服务器引擎会自动从内存中清除 session。如果设置的 timeout 时间很长，例如 3 小时，那么 session 中的所有对象都将保持 3 小时，这将影响内存垃圾回收和性能。为了提高性能，最好能在代码中使用 session.invalidate()方法显式地移除 session（当不再需要 session 对象时）。最好能根据每个应

用的需要，在应用级别调整 session 的 time out 时间。

可延续上一个例子，尝试设置 session 的 time out 时间，然后在 JProfiler 中观察对于范围是 session 这一级别的 jsp:useBean 所引用的对象是否能在 session 的 time out 时间到了之后可被垃圾回收。

Java 代码性能监控与性能问题定位

内存泄漏检测

JAVA 是托管语言，有优秀的垃圾回收机制，但是如果代码编写不恰当，也会引起内存泄漏问题，导致软件系统性能缓慢甚至“当机”。例如，下面代码中就包含了两个存在内存泄漏问题的函数：

```
import java.io.IOException;
import java.util.HashSet;
import java.util.Random;
import java.util.Vector;

public class LeakExample {
    static Vector myVector = new Vector();
    static HashSet pendingRequests = new HashSet();

    public void slowlyLeakingVector(int iter, int count) {
        for (int i=0; i<iter; i++) {
            for (int n=0; n<count; n++) {
                myVector.add(Integer.toString(n+i));
            }
            for (int n=count-1; n>0; n--) {
                //应该是 n>=0
                myVector.removeElementAt(n);
            }
        }
    }

    public void leakingRequestLog(int iter) {
        Random requestQueue = new Random();
        for (int i=0; i<iter; i++) {
            int newRequest = requestQueue.nextInt();
            pendingRequests.add(new Integer(newRequest));
            //忘记移除对象了
```

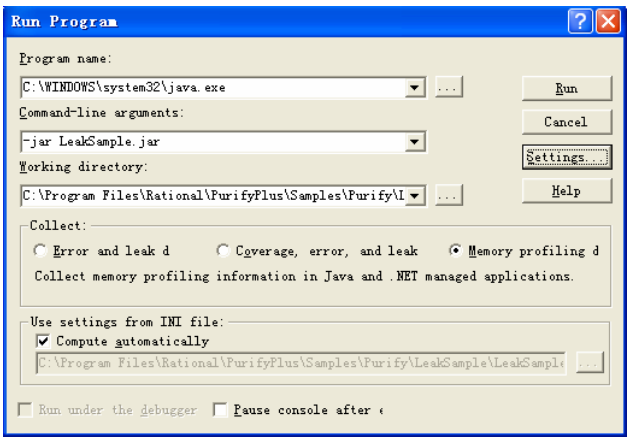
```
        }
    }

    public void noLeak(int size) {
        HashSet tmpStore = new HashSet();
        for (int i=0; i<size; ++i) {
            String leakingUnit = new String("Object: " + i);
            tmpStore.add(leakingUnit);
        }
        // 虽然在函数内分配了很多内存，但是函数执行完后，这些内存都能被“垃圾回
收”
    }

    public static void main(String[] args) throws IOException {
        LeakExample javaLeaks = new LeakExample();
        for (int i=0; true; i++) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) { /*  不做任何事情  */ }
            System.out.println("Iteration: " + i);
            javaLeaks.slowlyLeakingVector(1000,10);
            javaLeaks.leakingRequestLog(5000);
            javaLeaks.noLeak(100000);
        }
    }
}
```

上面的代码中，slowlyLeakingVector 函数中的第一个循环往 myVector 添加了很多对象，第二个循环负责将对象从 myVector 移除，但是循环条件写错了（应该是 n>=0），导致对象移除不干净，发生缓慢的内存泄漏。

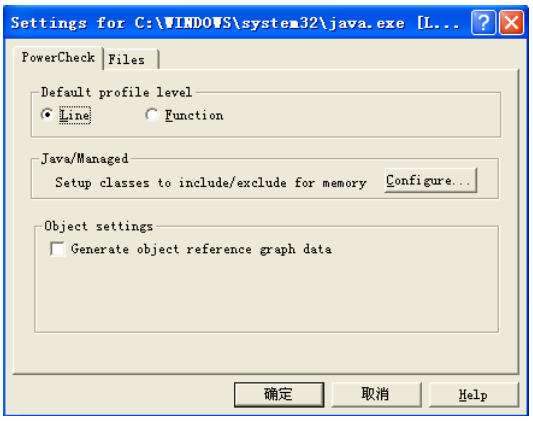
第二个函数 leakingRequestLog 干脆忘记移除对象了，pendingRequests 这个成员集合变量被不断地添加对象进去，只有整个程序运行结束，这些对象才有可能被垃圾回收。

JAVA 代码的内存泄漏问题可利用 JDK 附带的 JConsole 来查看分析，也可以用 Purify 这类专门的内存问题检测工具来分析。下面以 Purify 附带的例子（LeakSample）介绍如何用 Purify 检测 JAVA 代码内存泄漏问题。

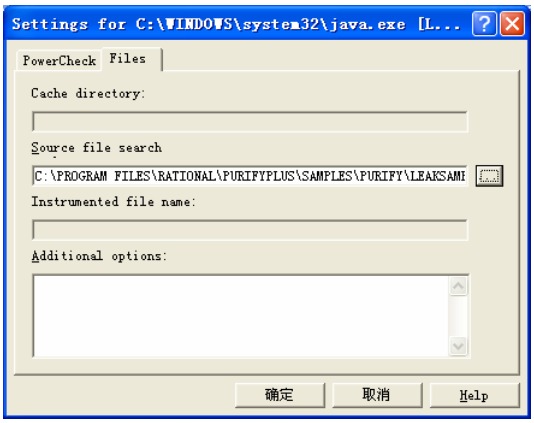
（1）打开 Purify，指定运行 LeakSample 程序，如图所示



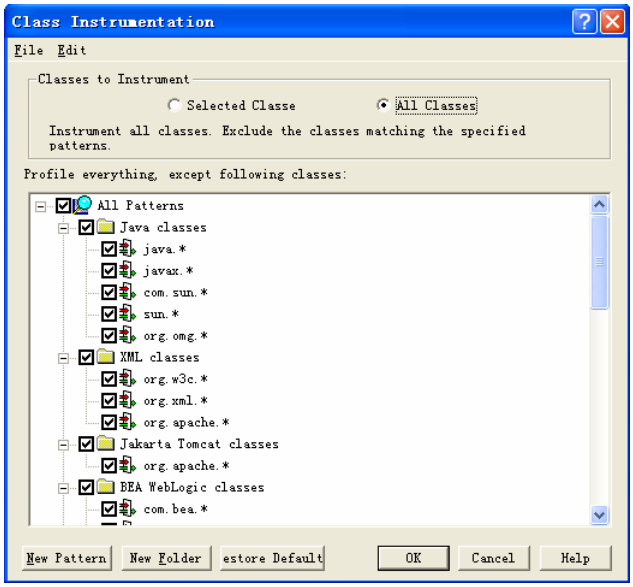
（2）点击“Settings”进行设置，例如设置分析级别，如图所示：



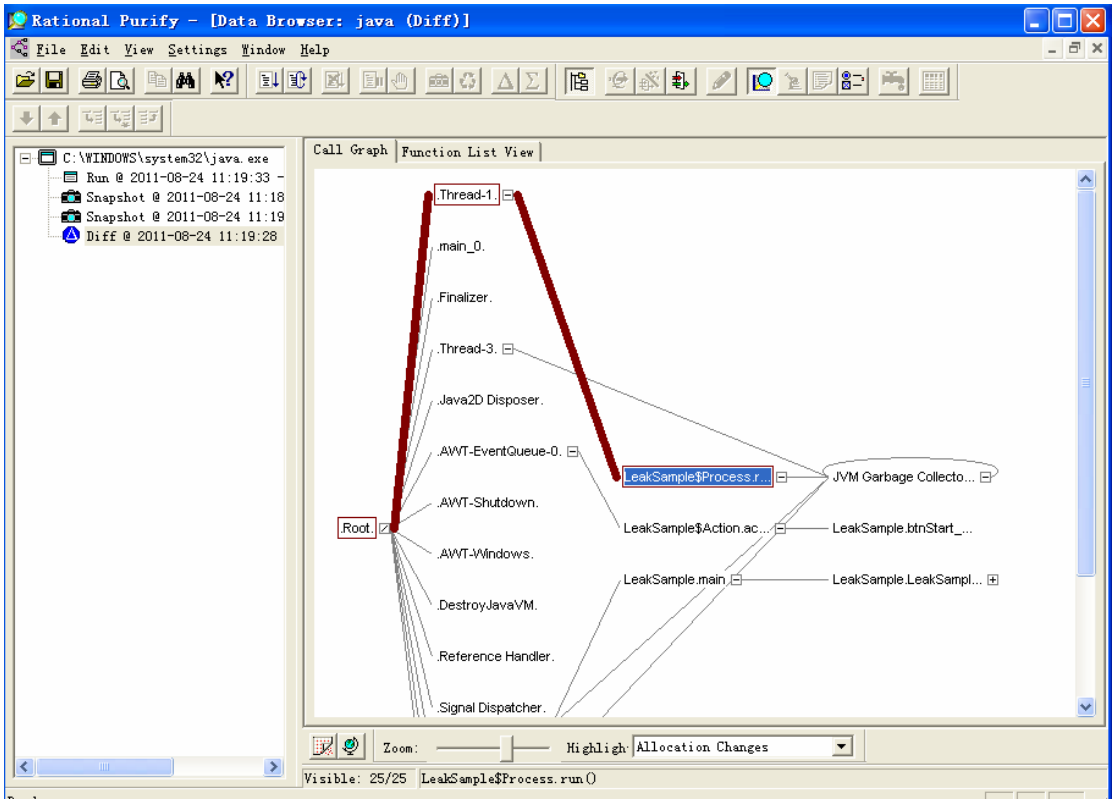
设置与源代码关联，如图所示：



设置类过滤，如图所示：



（3）用 Purify 执行程序进行分析，待程序初始化完成之后，先截取一个内存快照，然后执行程序的相关功能，尤其是那些怀疑有内存泄漏的功能，同时观察“Call Graph”界面中的内存变化情况，如果看到有内存使用持续增长，则进行一次垃圾回收，如果内存使用曲线没有明显的下降，这时可以截取第二个快照，与第一个快照进行比较，如图所示：



（4）对比图会将内存使用变化的地方高亮显示，内存使用量下降的函数和路径会用绿色标注，内存使用量上升则用红色标注。

（5）通过分析调用图（Call Graph）和函数列表中的内存使用变化情况，可初步定位到内存泄漏的地方，结合 LeakSample 的源代码分析，可知，线程体中的第一个循环添加的对象，在第二个循环中并未全部释放，第一个循环的条件是“<=cnt”，而第二个循环的条件是

“<cnt”：

```
for (i=0; i<=cnt; i++)
{
    vBytes.addElement (new byte[8196]);
}

for (i=0; i<cnt; i++)
{
    vBytes.removeElementAt (0);
}
```

这样就会每次都有一个对象不能得到及时的释放，造成缓慢的内存泄漏现象。下面附上 LeakSample 的全部源代码：

```
import java.util.*;
import java.awt.*;

public class LeakSample extends Frame
{
    Runtime rt;
    public boolean bLeakMemory;
    public boolean bLeakOnce;
    public Object oLeakObject;

    public LeakSample (String title)
    {
        super (title);

        bLeakMemory = false;
        bLeakOnce = false;
        oLeakObject = new Object ();

        rt = java.lang.Runtime.getRuntime ();

        setLayout(null);
        setSize(344,240);
        label1 = new java.awt.Label("Free Memory");
        label1.setBounds(36,24,120,24);
        add(label1);
        label2 = new java.awt.Label("Total Memory");
        label2.setBounds(36,60,120,24);
        add(label2);
        txtFreeMemory = new java.awt.TextField();
        txtFreeMemory.setBounds(180,24,144,24);
```

```
        add(txtFreeMemory);
        txtTotalMemory = new java.awt.TextField();
        txtTotalMemory.setBounds(180,60,144,24);
        add(txtTotalMemory);
        btnStart = new java.awt.Button();
        btnStart.setLabel("Start");
        btnStart.setBounds(108,144,144,36);
        btnStart.setBackground(new Color(12632256));
        add(btnStart);
        btnExit = new java.awt.Button();
        btnExit.setLabel("Exit");
        btnExit.setBounds(108,190,144,36);
        btnExit.setBackground(new Color(12632256));
        add(btnExit);
        Group1 = new CheckboxGroup();
        radioLeakSome = new java.awt.Checkbox("Leak Some", Group1, false);
        radioLeakSome.setBounds(48,108,96,24);
        add(radioLeakSome);
        radioLeakContinuous = new java.awt.Checkbox("Leak Continuously", Group1, false);
        radioLeakContinuous.setBounds(180,108,144,24);
        add(radioLeakContinuous);

        // make LeakSome the default state
        radioLeakSome.setState (true);
        //}}

        //{{REGISTER_LISTENERS
        Action lAction = new Action ();
        btnStart.addActionListener (lAction);
        btnExit.addActionListener (lAction);
        //}}

        // start the process thread
        Process procThread = new Process (10);

        this.show ();
}

//{{DECLARE_CONTROLS
java.awt.Label label1;
java.awt.Label label2;
java.awt.TextField txtFreeMemory;
java.awt.TextField txtTotalMemory;
java.awt.Button btnStart;
```

```
java.awt.Button btnExit;
java.awt.Checkbox radioLeakSome;
CheckboxGroup Group1;
java.awt.Checkbox radioLeakContinuous;
//}}

void btnStart_Clicked (java.awt.event.ActionEvent event)
{
	if (radioLeakSome.getState () == true)
	{
		synchronized(oLeakObject)
		{
			bLeakMemory = true;
			bLeakOnce = true;
		}
	}
	else
	{
		if (btnStart.getLabel () == "Start")
			btnStart.setLabel ("Stop");
		else
		{
			btnStart.setLabel ("Start");

			// stop the leak
			synchronized(oLeakObject)
			{
				bLeakMemory = false;
			}

			return;
		}

		synchronized(oLeakObject)
		{
			bLeakMemory = true;
			bLeakOnce = false;
		}
	}
}

/**
 * Process
 */
```

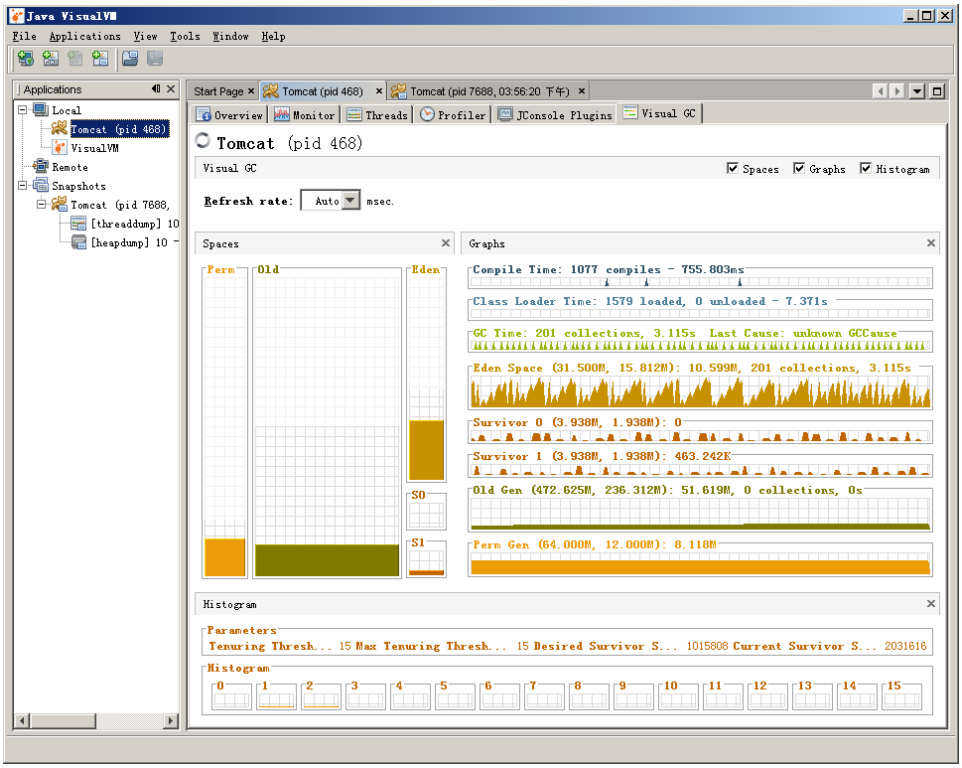
```
class Process extends Thread
{
	public Vector vBytes;
	public int cnt;

	public Process (int id)
	{
		// pre-allocate enough space for specified no of strings
		vBytes = new Vector (id);
		cnt = id;

		this.start ();
	}

	public void run ()
	{
		int i = 0;

		while (true)
		{
			synchronized(oLeakObject)
			{
				if (bLeakMemory == true)
				{
					// allocate memory and add a reference to it
					for (i=0; i<=cnt; i++)
					{
						vBytes.addElement (new byte[8196]);
					}

					for (i=0; i<cnt; i++)
					{
						vBytes.removeElementAt (0);
					}

					if (bLeakOnce == true)
					{
						bLeakMemory = false;
					}

					txtFreeMemory.setText (String.valueOf (rt.freeMemory ()));
					txtTotalMemory.setText (String.valueOf (rt.totalMemory ()));
				}
			}
```

```
                    // kill some time between processing
                    try   {
                        java.lang.Thread.sleep (100);
                    } catch (InterruptedException e)
                    {
                    }
                }

            }
        }

        class Action implements java.awt.event.ActionListener
        {
            public void actionPerformed (java.awt.event.ActionEvent event)
            {
                Object object = event.getSource ();
                if (object == btnStart)
                    btnStart_Clicked (event);
                else if (object == btnExit)
                    System.exit (0);
            }
        }

        public static void main (String args[])
        {
            LeakSample app = new LeakSample ("LeakSample");
        }
}
```

JVM 内存泄漏分析

认识 JVM

首先我们来看一张图，这是目前 JDK1.6 版本自带的 JVM 性能监控工具 VisualVM 的一个插件 VisualGC 的显示情况。让我们先来了解 JVM 的内存堆 Heap 管理模式，要调整 JVM，自然要知道它的内部结构和运作，此乃“知己知彼，百战不殆”！



注意：需要在 VisualVM 中选择 Tools->Plugins 进行 VisualGC 的安装。

JVM 的 Heap 包括三部分，分别是 Permanent Generation（简称 PermGen）、New Generation 和 Tenured Generation（又叫 Old ）。

PermGen 是 JVM 自用的区域，是内存的永久保存区，用于存放反射代理和 Class，Class 在被装载时就会被放到 PermGen 中，所以如果 WEB 服务器里应用的 Class 相当多时，就可以考虑将这一块区域放大一些。

New 和 Old 是 Java 应用的 Heap 区，用来存放类的实例 Instance 的；其中 New Generation 的目标就是尽可能快速的收集掉那些生命周期短的对象；New Generation 又分为 Eden Space，From Space 和 To Space 三块，Eden Space 用于存放新创建的对象，From 区和 To 区都是救助空间 Survivor Space（图中的 S0 和 S1）；当 Eden 区满时，JVM 执行垃圾回收 GC（Garbage Collection），垃圾收集器暂停应用程序，并会将 Eden Space 还存活的对象复制到当前的 From 救助空间，一旦当前的 From 救助空间充满，此区的存活对象将被复制到另外一个 To 区，当 To 区也满了的时候，从 From 区复制过来并且依然存活的对象复制到 Old 区，从而 From 和 To 救助空间互换角色，维持活动的对象将在救助空间不断复制，直到最终转入 Old 域。需要注意，Survivor 的两个区是对称的，没先后关系，所以同一个区中可能同时存在从 Eden 复制过来对象，和从前一个 Survivor 复制过来的对象，而且，Survivor 区总有一个是空的。同时，根据程序需要，Survivor 区是可以配置为多个的(多于两个)，这样可以增加对象在年轻代中的存在时间，减少被放到年老代的可能。每一次垃圾回收后，Eden Space 都会被清空。

Old 区用于存放长寿的对象，在 New 区中经历了 N 次垃圾回收后仍然存活的对象，就会被放到 Old 区中；如那些与业务信息相关的对象，包括 Http 请求中的 Session 对象、线程、Socket 连接，这类对象跟业务直接挂钩，因此生命周期比较长。当任何一个空间不够用时，都会促使 JVM 执行垃圾回收，而垃圾回收也需要消耗一定的时间，从而造成应用程序卡住；因此，合理的设置各

个区的大小，可以进行快速 GC 即 Mimor GC，避免频繁的 Full GC。

所谓 Minor GC，一般情况下，当新对象生成，并且在 Eden 申请空间失败时，就会触发 Minor GC，对 Eden 区域进行 GC，清除非存活对象，并且把尚且存活的对象移动到 Survivor 区，然后整理 Survivor 的两个区。这种方式的 GC 是对 New 区的 Eden 区进行，不会影响到 Old 区。因为大部分对象都是从 Eden 区开始的，同时 Eden 区不会分配的很大，所以 Eden 区的 Minor GC 会频繁进行。因而，一般在这里需要使用速度快、效率高的算法，使 Eden 去能尽快空闲出来。

而 Full GC 要对整个 Heap 区进行回收，包括 New、Old 和 PermGen，所以比 Minor GC 要慢，因此应该尽可能减少 Full GC 的次数。在对 JVM 性能调优的过程中，很大一部分工作就是对于 Full GC 的调节。有如下原因可能导致 Full GC：

- Tenured 区被写满
- PermGen 区被写满
- System.gc()被显示调用
- 上一次 GC 之后 Heap 的各域分配策略动态变化

JVM 启动参数介绍

-Xmn －Eden Generation 的 Heap 大小，一般设置为 Xmx 的 1/3 或 1/4
-Xmx －设置 JVM Heap 大小最大值，这里的 heap = New Generation + Old Generation，但不包括 PermGen
-Xms －设置 JVM Heap 大小初始值
-XX:NewRatio －New/Old 的大小比率
-XX:NewSize －New Generation Heap 的大小
-XX:MaxNewSize －可以通过 NewRatio 和-Xmx 计算得到
-XX:SurvivorRatio －Eden/Survivor Space 大小比率
-XX:PermSize －PermGen 的初始值
-XX:MaxPermSize －PermGen 最大值
-Xss －设置每个线程的 Stack 大小
-XX:+UseParNewGC －表示多 CPU 下缩短 Minor GC 的时间
-XX:+UseParallelGC －设置后可以使用并行清除收集器【多 CPU】
-XX:+ParallelGCThreads －可用来增加并行度【多 CPU】
-XX:+AggressiveOpts －是否激活最近的试验性性能调整
-XX:-Xnoclassgc －是否允许类垃圾收集，默认设置是允许类 GC
-XX:+UseLargePages －是否支持大页面堆
-XX:+UseFastAccessorMethods －在指定了这个参数后，JDK 会将所有的 get/set 方法都转为本地代码
-XX:+UseConcMarkSweepGC －缩短 major 收集的时间，此选项在 Heap 比较大而且 Full GC 时间较长的情况下使用更合适

另外，JVM 的一些参数可以输出有效的日志文件：
-verbose:gc －输出一些 gc 信息

-XX:+PrintGCDetails －输出 gc 详细信息
-XX:+PrintGCTimeStamps －包含时间戳信息
-XX:+PrintHeapAtGC －包括 gc 前后 Heap 状况
-XX:+PrintTenuringDistribution －输出对象存活时间和 Tenured Generation 的其他信息
-XX:+PrintHeapUsageOverTime －以时间戳输出 heap 利用率和容量信息
-Xloggc:filename －输出 gc 信息到日志文件 例如：
set JAVA_OPTS=%JAVA_OPTS% -verbose:gc -Xloggc:gc.log -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintHeapAtGC

如果输出的日志文件非常大，可以后续采用其它工具分析；或者干脆使用 Jstat 或 Jconsole 这些成熟的工具直接监控 JVM 的 GC 问题。当然，前边图示的 VisualVM 更是一款漂亮的、即时监控 JVM GC 的工具。

下面的命令把整个堆设置成 128m，New/Old 比率设置成 3，即 New 与 Old 比例为 1：3，New Space 为 Heap 的 1/4 即 32M，缺省值为 NewSize=2m MaxNewSize=32m SurvivorRatio=2
java –Xms128m –Xmx128m –XX:NewRatio=3
但是，如果 JVM 的堆 Heap 大小大于 1GB，则应该使用值：-XX:newSize=640m -XX:MaxNewSize=640m -XX:SurvivorRatio=16，或者将堆的总大小的 50%到 60%分配给新生成的池。

JVM 性能瓶颈

在 JAVA 应用程序中，虽然不太容易出现内存泄漏的问题，因为 JVM 会不定期的进行垃圾回收。但是因为程序的不合理写法，也会导致一些数据不能被收集。典型的状况：

- 在 HashMap 中放置大量不用的数据，而没有及时的清理。
- 在 web 应用中，很多人喜欢在 Session 放置状态数据，而没有清理。
- 在 Session 中存放数据还好，因为 Session 终究会有过期时间，但是如果在 Class 的 Static 变量中放置数据，那就怎么样也没办法了。

诊断应用中是否存在内存泄漏也有一些方法，通过分析 JVM GC Log 就是一个直观的方式。通过分析 GC After Heap 的变化趋势，如果 GC After Heap 稳步上升，立刻进行 Full GC 后，仍然不能降下来，通常就意味着存在内存泄漏了。当然也有情况是，的确有一些数据是应用程序的需求，我们要确认除了这些数据，是否还存在一些异常数据一直占据内存。可以通过 JProbe 的 Memory Views 来观察 Class 的对象数，在一段请求过后，如果还存在一些 Class 的 Instance 数目相当多，就可以判断这个 Class 可能会是问题的根源。

JVM 内存泄漏实例一 - PermGen 溢出

java.lang.OutOfMemoryError PermGen space
由于GC不会在主程序运行期对PermGen进行清理，所以如果你的应用中有很多CLASS的话，就很可能出现PermGen space错误，这种错误常见在WEB服务器对JSP进行预编译的时候发生。如果你的WEB APP下都用了大量的第三方JAR包，其大小超过了JVM默认的大小(4M)那么就会产生此错误信息了。

解决方法： 手动设置MaxPermSize大小。修改TOMCAT_HOME/bin/catalina.bat，在echo Using CATALINA_base:%CATALINA_base%上面加入以下行：
JAVA_OPTS=%JAVA_OPTS% -server -XX:PermSize=64M
-XX:MaxPermSize=128m
建议：将相同的第三方 jar 文件移置到 tomcat/shared/lib 目录下，这样可以达到减少 jar 文档重复占用内存的目的。

JVM 内存泄漏实例二 - Heap 溢出

java.lang.OutOfMemoryError: Java heap space
JVM Heap堆的设置是指java程序运行过程中JVM可以调配使用的内存空间。JVM在启动的时候会自动设置Heap size的值，其初始空间(即-Xms)是物理内存的1/64，最大空间(-Xmx)是物理内存的1/4。可以利用JVM提供的-Xmn -Xms -Xmx等选项可进行设置。提示：在JVM中如果98%的时间是用于GC且可用的Heap大小不足2%的时候将抛出此异常信息。Heap最大不要超过可用物理内存的80%，一般的要将-Xms和-Xmx选项设置为相同，而-Xmn为1/4的-Xmx值。 解决方法：手动设置Heap size，例如：
JAVA_OPTS=%JAVA_OPTS% -server -Xms800m -Xmx800m
但是有的时候可能这样的设置还会不生效，例如当JVM加载大量Class类时，永久域中的对象急剧增加，从而使JVM不断调整永久域大小。为了避免自动调整，可以使用前面的参数结合设置，如：
JAVA_OPTS=%JAVA_OPTS% -server -Xms800m -Xmx800m -XX:PermSize=64m
-XX:MaxPermSize=128m
基于Sun 的Java的垃圾回收机制，允许分隔内存池以包含不同时效的对象，垃圾回收循环根据时效收集与其他对象彼此独立的对象。使用其他参数，您可以单独设置内存池的大小。为了实现更好的性能，可以对包含短期存活对象的池的大小进行设置，以使该池中的对象的存活时间不会超过一个垃圾回收循环。新生成的池的大小由 NewSize和MaxNewSize参数确定。如：
JAVA_OPTS=%JAVA_OPTS% -server -Xms800m -Xmx800m -XX:PermSize=64m
-XX:MaxPermSize=128m -XX:NewSize=8m -XX:MaxNewSize=16m

JVM 内存泄漏实例三 - 垃圾回收时 promotion failed

这个问题一般由两种原因引起，第一个原因是 Survivor 空间不够，里面的对象还不应该被移动到 Old 区，但 New 区又有很多对象需要放入 Survivor；第二个原因是 Old 区没有足够的空间接纳来自 New 区的对象；这两种情况都会转向 Full GC，造成系统卡住时间较长。第一个原因可以通过去掉救助空间解决，设置 -XX:SurvivorRatio=65536 -XX:MaxTenuringThreshold=0 即可，第二个原因可以设置 CMSInitiatingOccupancyFraction 为某个值（假设 70），这样 Old 区的 70%时就开始执行 CMS，Old 区有足够的空间接纳来自 New 的对象。但是不管怎样，PermGen 还是会逐渐变满，所以隔三差五重起 java 服务器是必要的。需要注意的是，Old 区用的是并发回收，可以设置 New 区小一点，Old 区大些，可以减少系统的卡住。
解决方法：

$JAVA_ARGS .= " -Dresin.home=$SERVER_ROOT -server -Xms6000M -Xmx6000M -Xmn500M -XX:PermSize=500M -XX:MaxPermSize=500M -XX:SurvivorRatio=65536 -XX:MaxTenuringThreshold=0 -Xnoclassgc -XX:+DisableExplicitGC -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:+UseCMSCompactAtFullCollection -XX:CMSFullGCsBeforeCompaction=0 -XX:+CMSClassUnloadingEnabled -XX:-CMSParallelRemarkEnabled -XX:CMSInitiatingOccupancyFraction=90 -XX:SoftRefLRUPolicyMSPerMB=0 -XX:+PrintClassHistogram -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintHeapAtGC -Xloggc:log/gc.log"

-XX:SurvivorRatio=65536 -XX:MaxTenuringThreshold=0 就是去掉了救助空间

-Xnoclassgc 禁用类垃圾回收，性能会高一点

-XX:+DisableExplicitGC 禁止 System.gc()，免得程序员误调用 gc 方法影响性能

-XX:+UseParNewGC 对 New 采用多线程并行回收，这样收得快

带 CMS 参数的都是和并发回收相关的:

CMSInitiatingOccupancyFraction，这个参数设置有很大技巧，基本上满足(Xmx-Xmn)*(100-CMSInitiatingOccupancyFraction)/100>=Xmn 就不会出现 promotion failed。在这里 Xmx 是 6000，Xmn 是 500，那么 Xmx-Xmn 是 5500m，也就是 Old 有 5500m，CMSInitiatingOccupancyFraction=90 说明 Old 到 90%满的时候开始执行对 Old 区的并发垃圾回收（CMS），这时还剩 10%的空间是 5500*10%=550m，所以即使 Xmn（也就是 New 共 500m）里所有对象都搬到 Old 里，550 兆的空间也足够了，所以只要满足上面的公式，就不会出现垃圾回收时的 promotion failed。

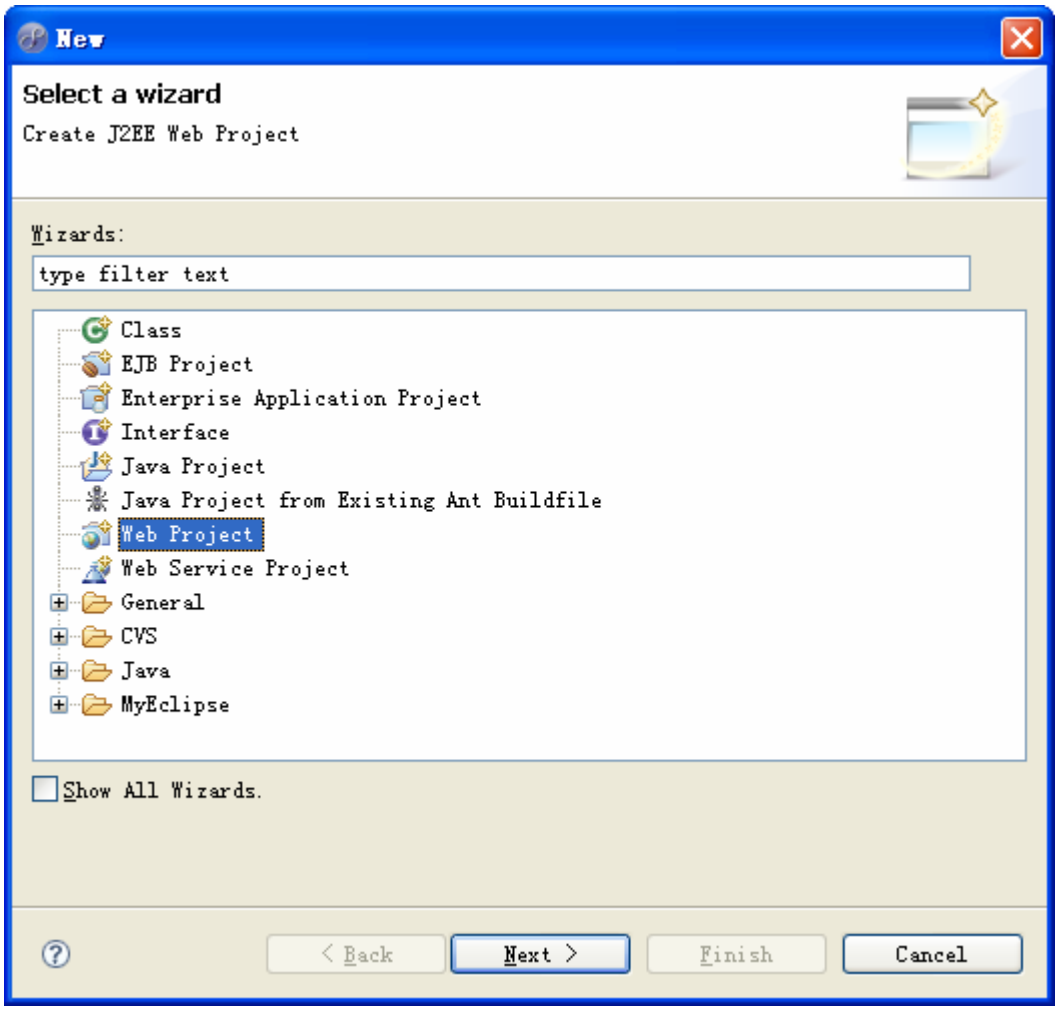
SoftRefLRUPolicyMSPerMB 这个参数是 softly reachable objects will remain alive for some amount of time after the last time they were referenced. The default value is one second of lifetime per free megabyte in the heap。

但是，这种设置方法因为没有用到 Survivor，所以 Old 区容易满，CMS 执行会比较频繁。建议还是用 Survivor 并把加大，这样也不会有 promotion failed 错误。配置上 32 位和 64 位不一样，64 位系统只要配置 MaxTenuringThreshold 参数，CMS 还是有暂停。为了解决暂停问题和 promotion failed 问题，最后我设置 -XX:SurvivorRatio=1，并把 MaxTenuringThreshold 去掉，这样即没有暂停又不会有 promotoin failed，而且更重要的是，Old 区和 PermGen 上升非常慢（因为好多对象到不了年老代就被回收了），所以 CMS 执行频率非常低，好几个小时才执行一次，这样，服务器都不用重启了。下面是 64 位的配置，系统 8G 内存：

-Xmx4000M -Xms4000M -Xmn600M -XX:PermSize=500M -XX:MaxPermSize=500M -Xss256K -XX:+DisableExplicitGC -XX:SurvivorRatio=1 -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+CMSParallelRemarkEnabled -XX:+UseCMSCompactAtFullCollection -XX:CMSFullGCsBeforeCompaction=0 -XX:+CMSClassUnloadingEnabled -XX:LargePageSizeInBytes=128M -XX:+UseFastAccessorMethods -XX:+UseCMSInitiatingOccupancyOnly -XX:CMSInitiatingOccupancyFraction=80 -XX:SoftRefLRUPolicyMSPerMB=0 -XX:+PrintClassHistogram -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintHeapAtGC -Xloggc:log/gc.log

实例-JProfile 跟踪内存泄漏

1、在 MyEclipse 中建立一个 Web 项目



2、在 cnj.com 包中新建两个类文件

TestBean.java：

```
package cnj.test;
public class TestBean {
    String name = "";
}
```

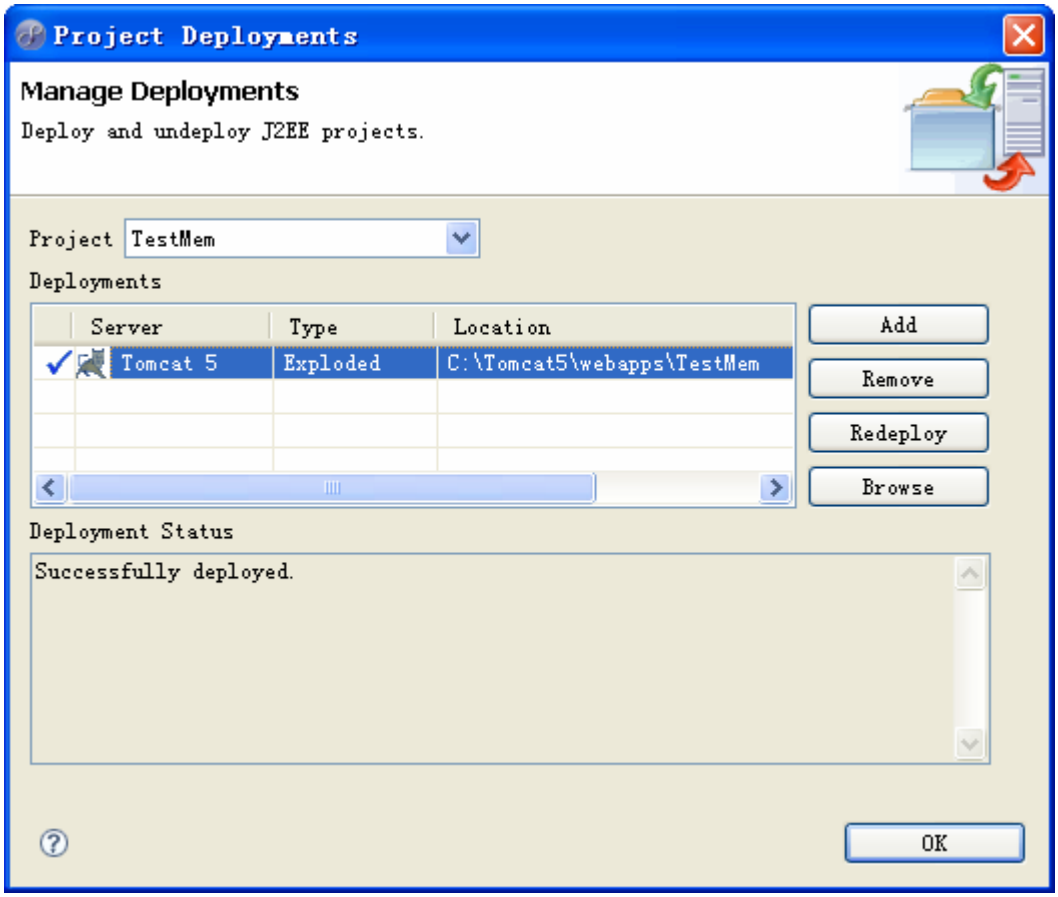
TestMain.java：

```
package cnj.test;
import java.util.ArrayList;
public class TestMain {
    public static ArrayList list = new ArrayList(); //存放对象的容器
    public static int counter = 0;    //作统计用
     }
```

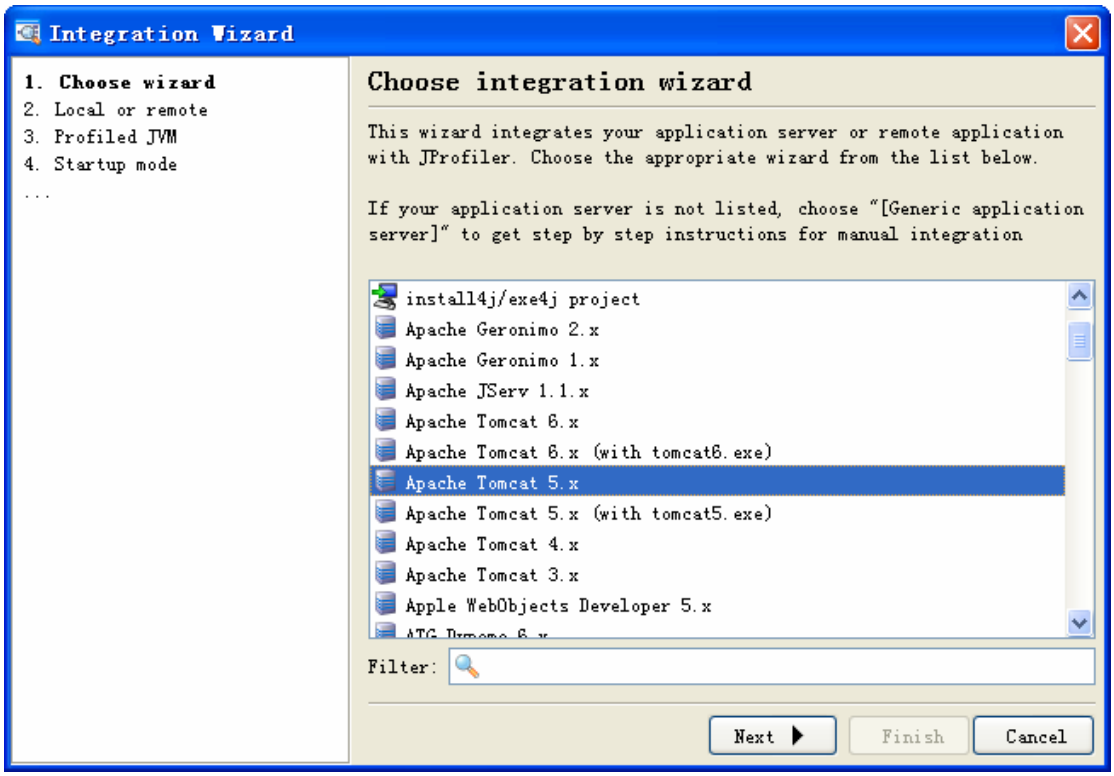
3、添加两个使用了TestBean类和TestMain类的JSP文件
两个JSP文件具有相同的内容：

```
<%@ page language="java" import="cnj.test.*"
pageEncoding="ISO-8859-1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>init</title>
  </head>
  <body><%
    for(int i=0;i<10000;i++){
       TestBean b = new TestBean();
       TestMain.list.add(b);
    }
    %>
    SIZE:<%=TestMain.list.size()%><br/>
    counter:<%=TestMain.counter++%>
  </body>
</html>
```

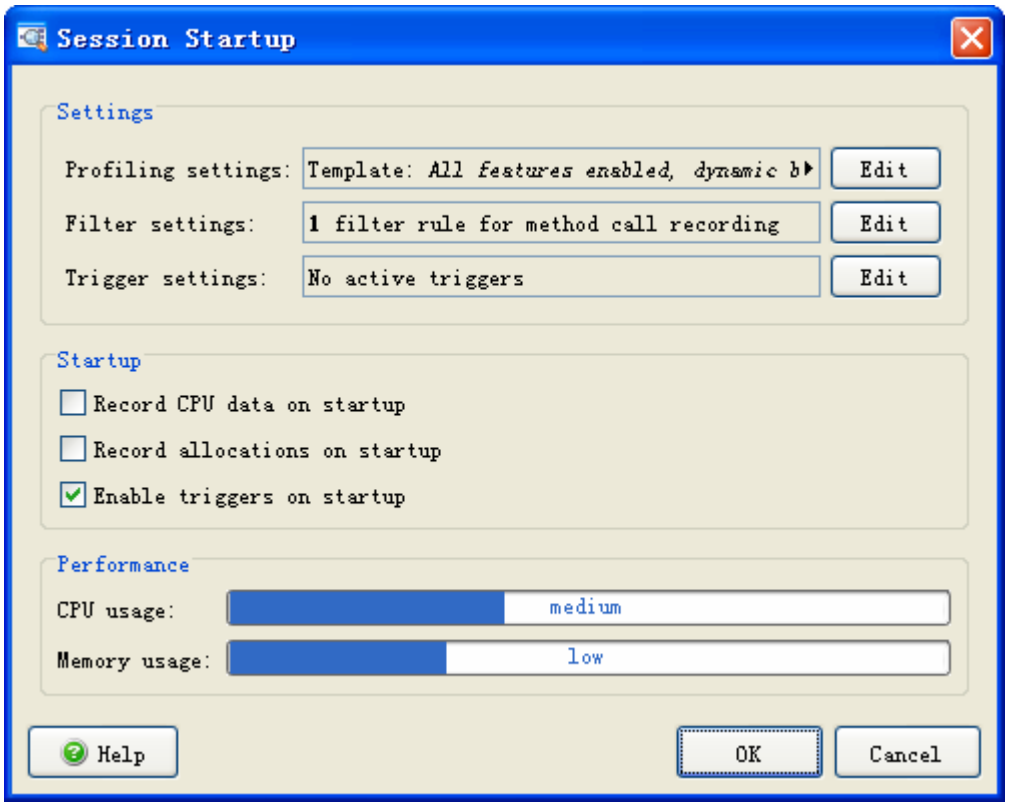
4、把项目部署到 Tomcat 服务器



5、启动 Jprofiler，选择监视和跟踪 Tomcat5.x

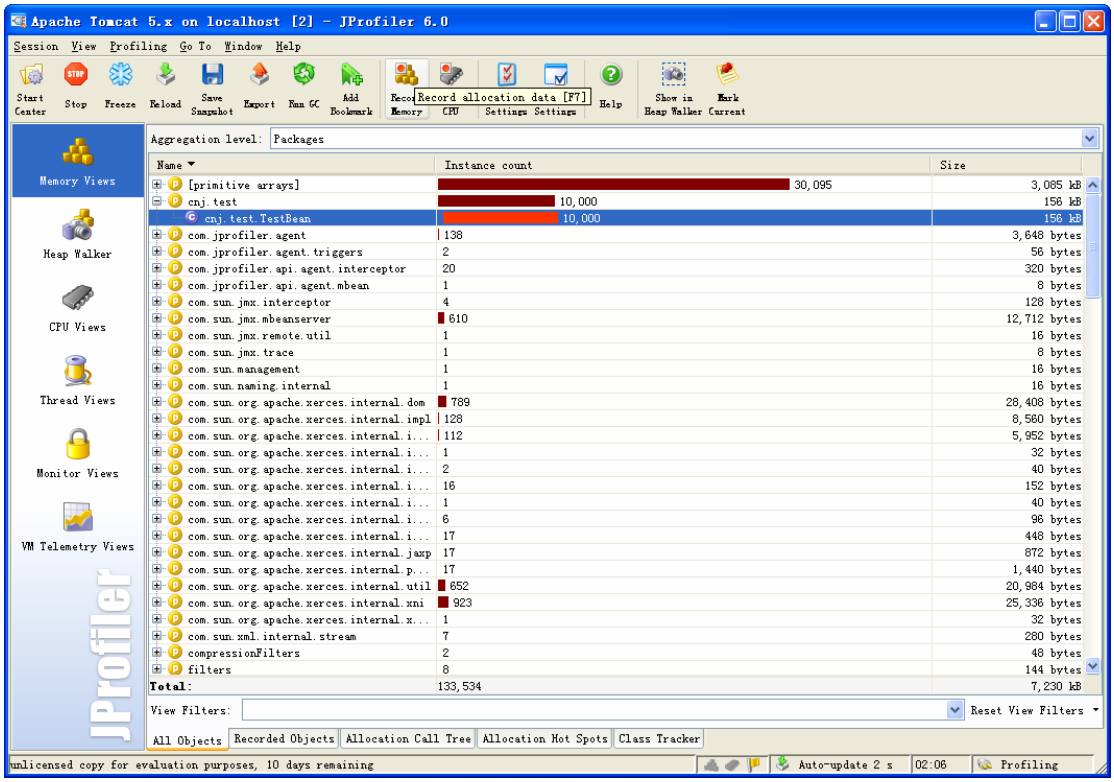


6、开始跟踪会话



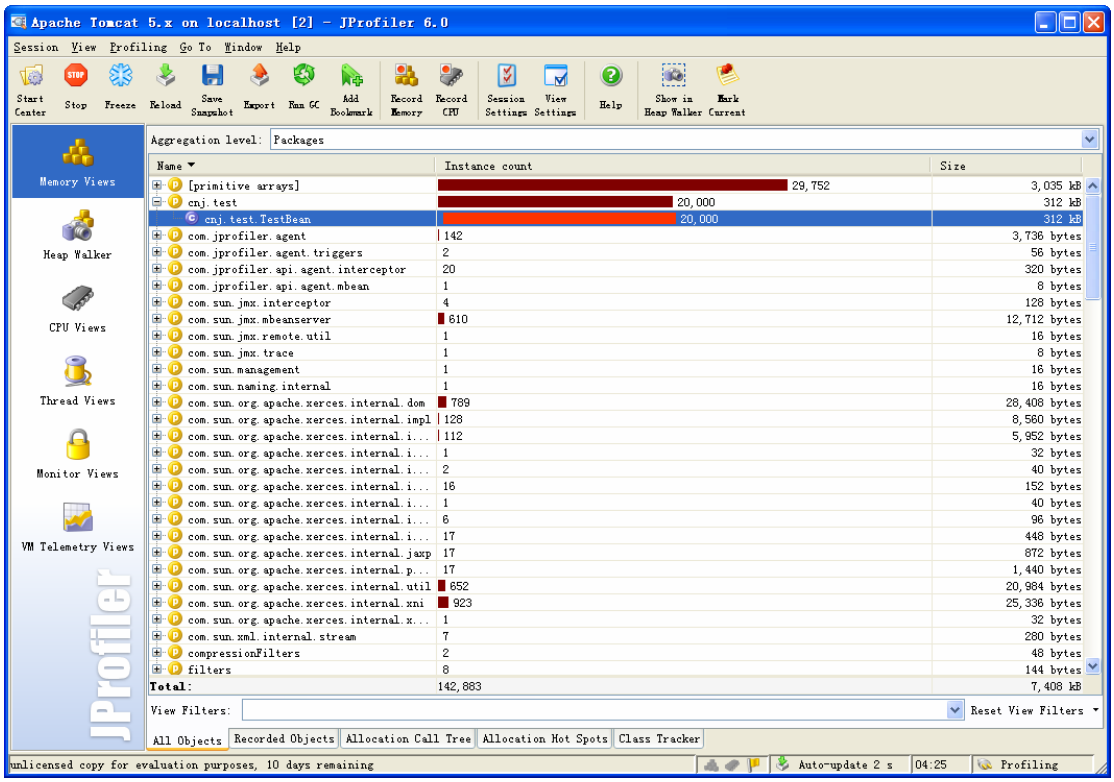
在浏览器中访问http://localhost:8080/TestMem/init1.jsp

在 JProfiler 中查看内存分配的情况：



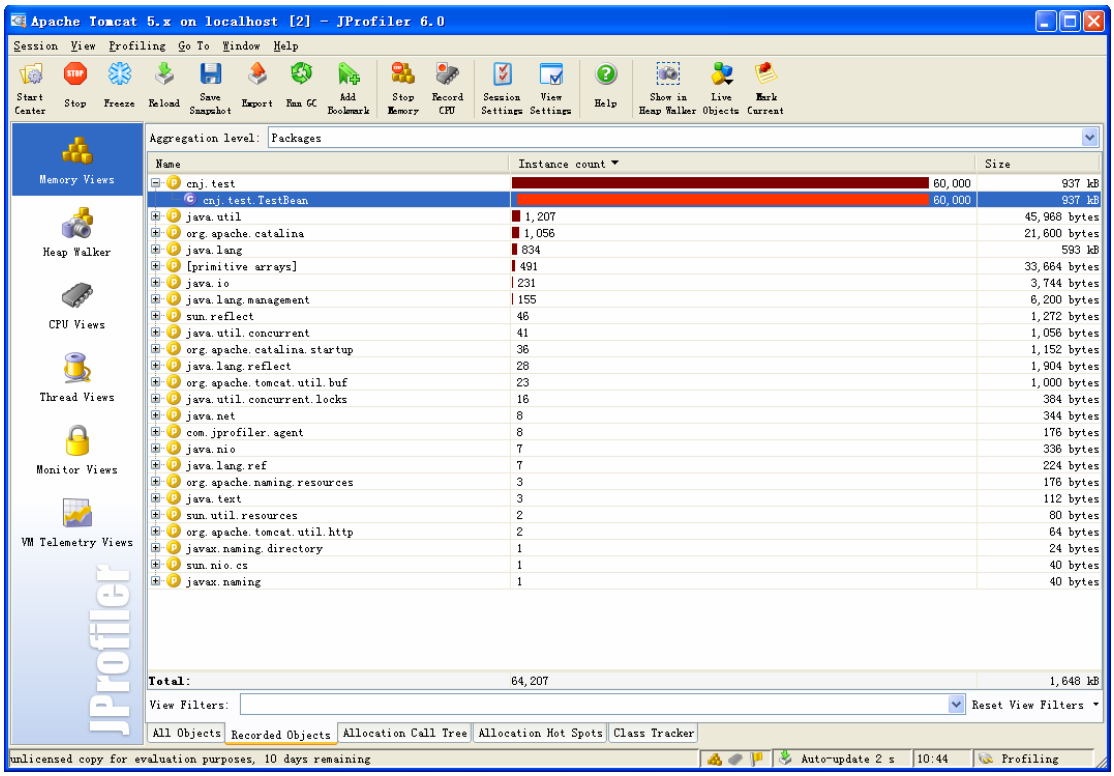
然后访问init2.jsp页面：http://localhost:8080/TestMem/init2.jsp

查看 JProfiler，可以看到内存使用增加：

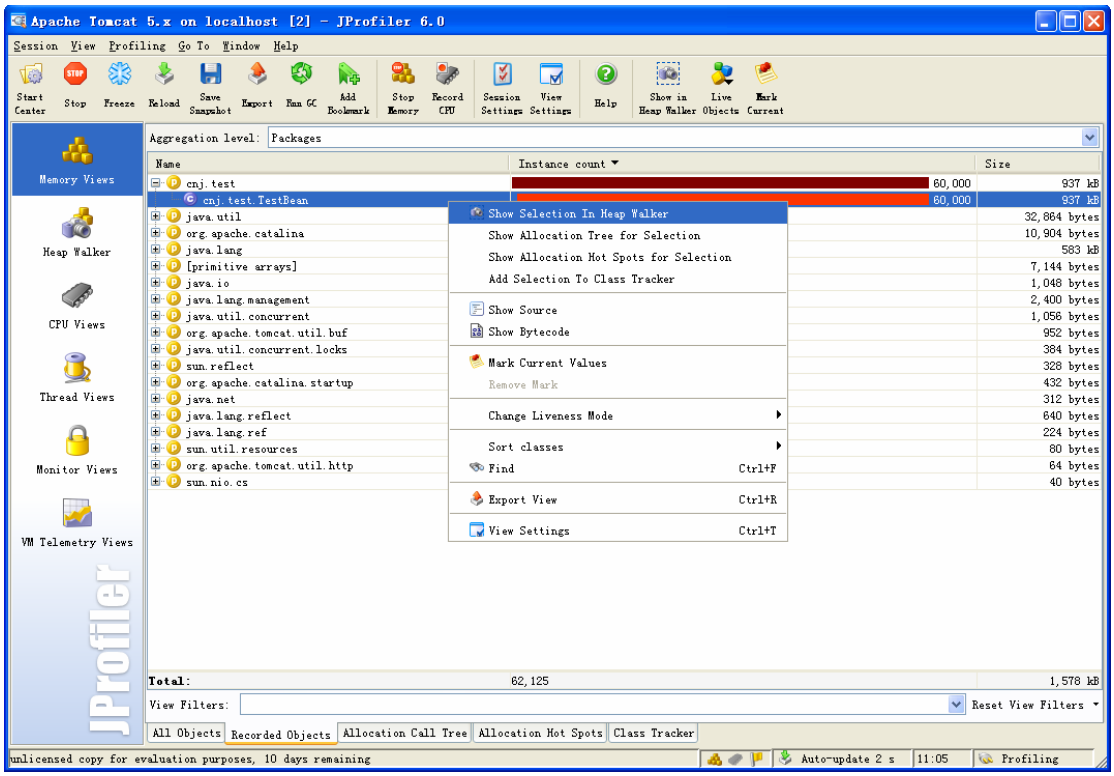


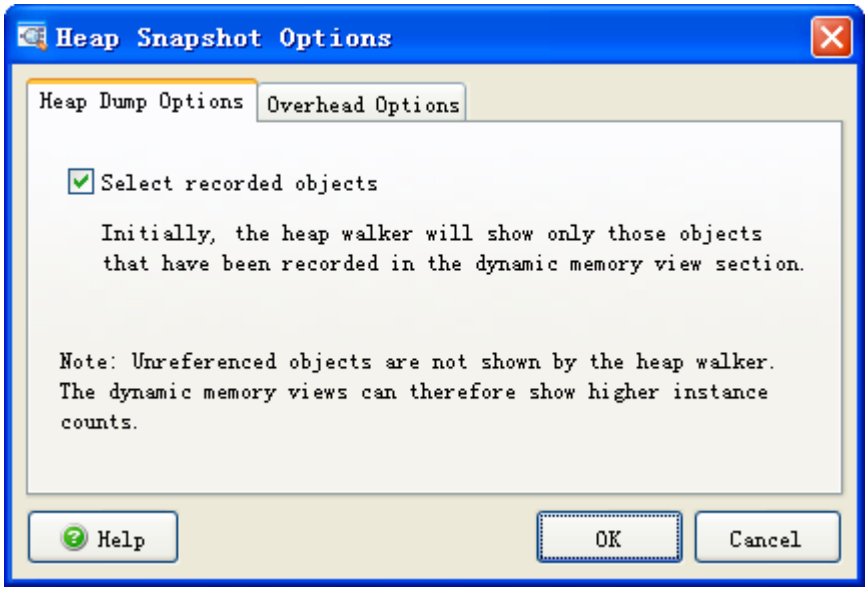
并且按快捷键 F4 进行垃圾回收，发现 cnj.com.TestBean 所占用的内存并没有释放，表明这些对象仍然存在，某些地方对这些对象的引用并没有释放。

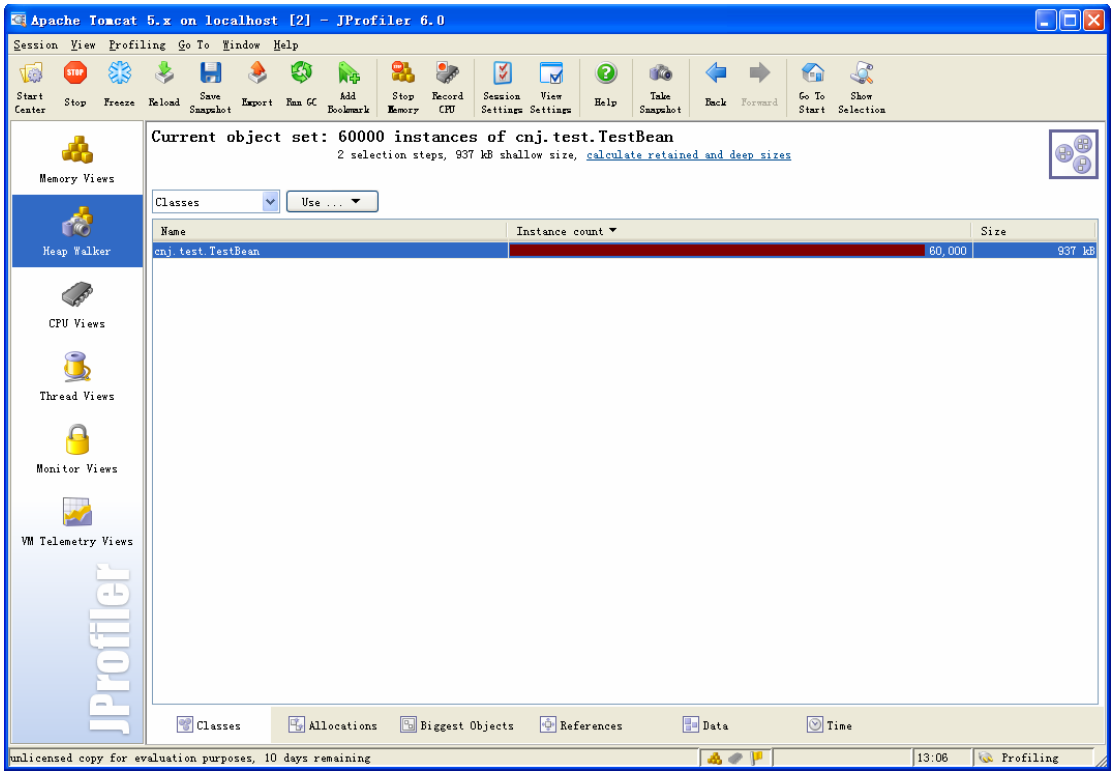
7、切换到 Recorded Objects 选项卡，按快捷键 F7 开始记录分配的数据，再次访问 Init1.jsp 和 Init2.jsp 页面若干次。



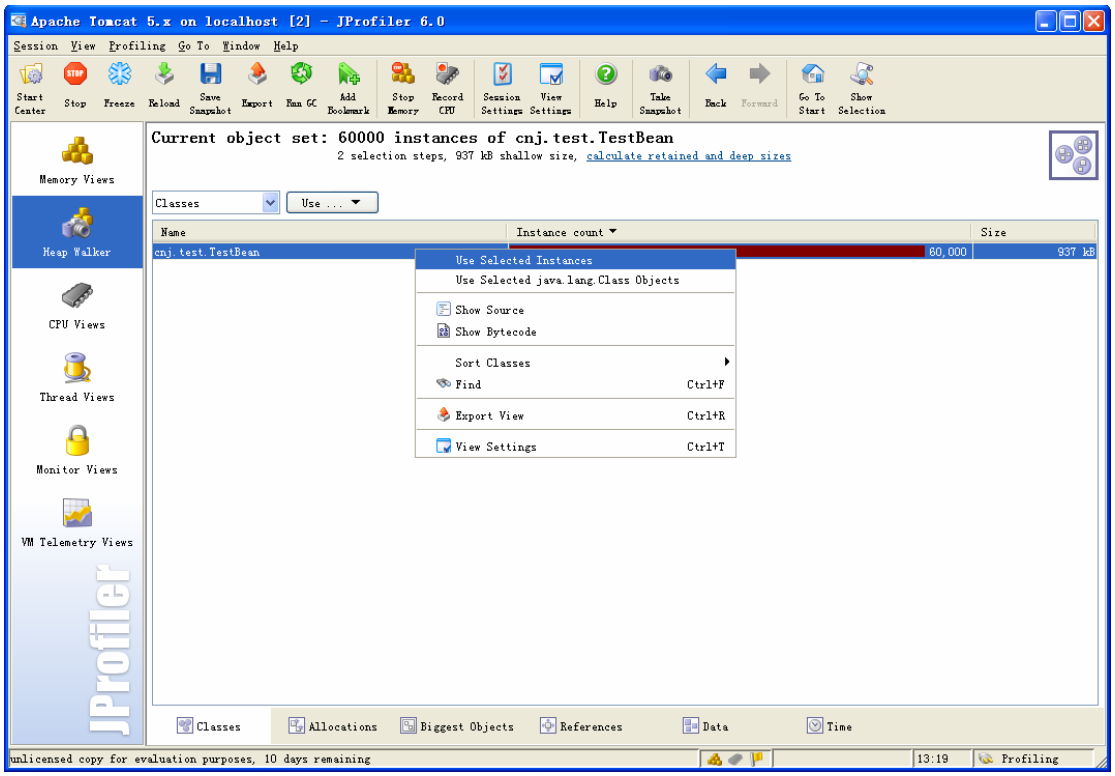
针对 cnj.test.TestBean 的内存对象，查看 Heap 的情况：



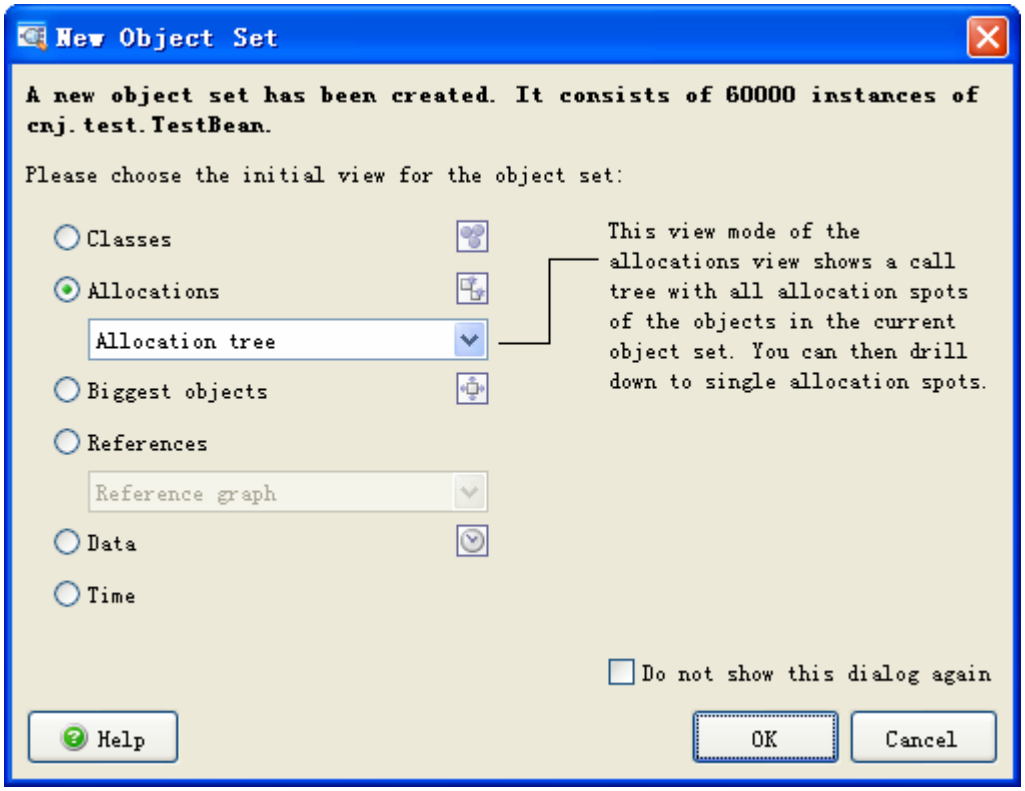




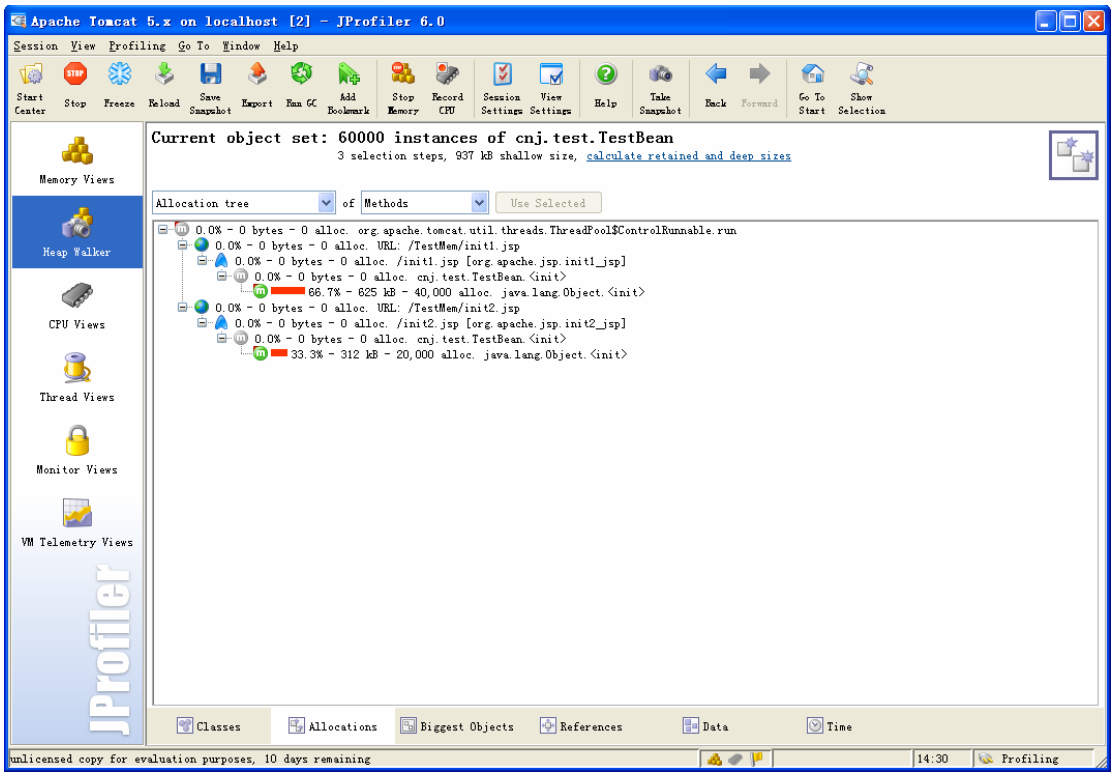
选择右键的“Use Selected Instances”



查看分配树：



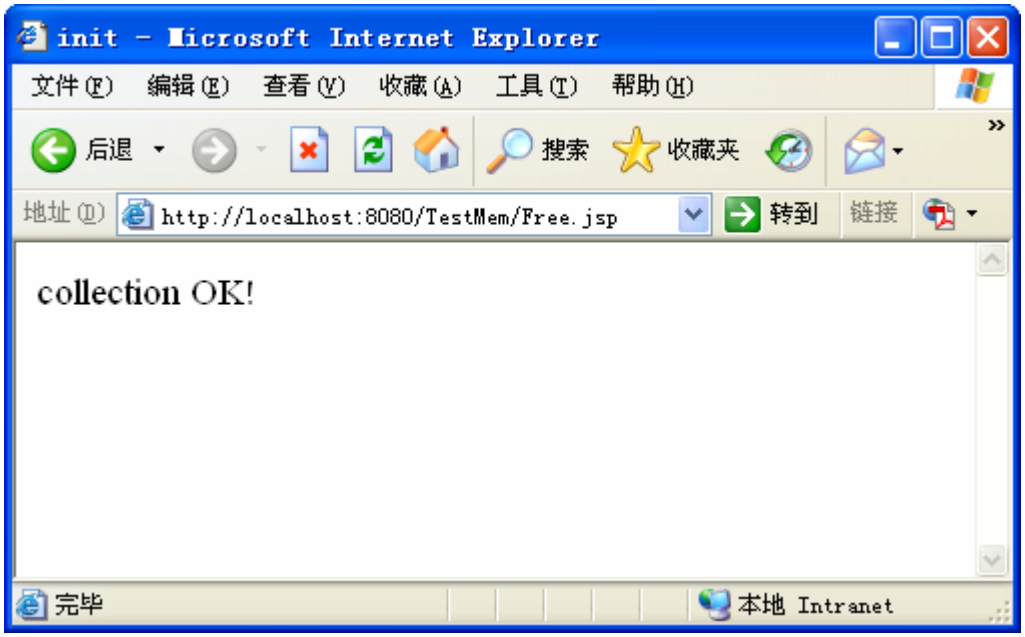
可以看到调用 cnj.TestBean 类的地方是 init1.jsp 和 init2.jsp，并且把各自占用的比率都列出来了：



8、在 Web 项目中添加一个 Free.jsp 文件用于释放内存

```
<%@ page language="java" import="java.util.*,cnj.test.*"
pageEncoding="ISO-8859-1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>init</title>
  </head>
  <body>
    <%TestMain.list.clear(); %>  collection OK!
  </body>
</html>
```

重新部署 TestMem 项目到 Tomcat 中。再次用 JProfiler 跟踪内存使用情况



在访问 init1.jsp 或 init2.jsp 后，可以看到内存是 cnj.TestBean 对象的数量增加了，然后执行 Free.sjp，接着再执行 F4 进行垃圾回收，立刻可以看到 TestBean 对象被释放掉了。

log4j 性能诊断与优化

实验：
Tomcat5.5.28
Log4j 1.2.16
JDK1.6

LoadRunner11

Intel Core 2 Duo CPU
2G 内存

Windows 2003

测试的 Servelet：

```
package servlet;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.UUID;
import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
```

```
import org.apache.log4j.xml.DOMConfigurator;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HelloWorld extends HttpServlet {

    /**
     * Constructor of the object.
     */
    public HelloWorld() {
        super();
    }

    /**
     * Destruction of the servlet. <br>
     */
    public void destroy() {
        super.destroy(); // Just puts "destroy" string in log
        // Put your code here
    }

    /**
     * The doGet method of the servlet. <br>
     *
     * This method is called when a form has its tag value method equals to get.
     *
     * @param request the request send by the client to the server
     * @param response the response send by the server to the client
     * @throws ServletException if an error occurred
     * @throws IOException if an error occurred
     */
    public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        String c = request.getParameter("c");
        if (c == null) {
            c = "Empty," + UUID.randomUUID().toString() + ".";
        }
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        out.println("<HTML><HEAD><TITLE>Test
Log4j</TITLE></HEAD><BODY><H1>");
```

```
        out.println("You Input:<BR>" + c);
        out.println("</H1></BODY></HTML>");
        out.flush();
        out.close();
        DOMConfigurator.configure("D:/log4j.xml");//加载.xml 文件
            Logger log=Logger.getLogger("appender1");
            log.info(c);

    }



    /**
     * The doPost method of the servlet. <br>
     *
     * This method is called when a form has its tag value method equals to post.
     *
     * @param request the request send by the client to the server
     * @param response the response send by the server to the client
     * @throws ServletException if an error occurred
     * @throws IOException if an error occurred
     */
    public void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out
                .println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01
Transitional//EN\">");
        out.println("<HTML>");
        out.println("  <HEAD><TITLE>A Servlet</TITLE></HEAD>");
        out.println("  <BODY>");
        out.print("    This is ");
        out.print(this.getClass());
        out.println(", using the POST method");
        out.println("  </BODY>");
        out.println("</HTML>");
        out.flush();
        out.close();
    }

    /**
     * Initialization of the servlet. <br>
```

```
     *
     * @throws ServletException if an error occure
     */
    public void init() throws ServletException {
        // Put your code here
    }

}
```

log4j 的配置文件：

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

<appender name="appender1" class="org.apache.log4j.DailyRollingFileAppender">
<!-- 设置通道 ID:org.zblog.all 和输出方式：org.apache.log4j.DailyRollingFileAppender -->
    <param name="File" value="/log4j_1.log" /><!-- 设置 File 参数：日志输出文件名 -->
    <param name="Append" value="true" /><!-- 设置是否在重新启动服务时，在原有日志的基础添加新日志 -->

<param name="BufferSize" value="8192" />
<param name="ImmediateFlush" value="false" />
<param name="BufferedIO" value="true" />

    <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern" value="%p (%c:%L)- %m%n" /><!-- 设置输出文件项目和格式 -->
    </layout>
</appender>


<appender name="appender2" class="org.apache.log4j.DailyRollingFileAppender">
    <param name="File" value="/log4j_2.log" />
    <param name="Append" value="true" />



    <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern" value="%p (%c:%L)- %m%n" />
    </layout>
</appender>
```

```
<root> <!-- 设置接收所有输出的通道 -->
    <appender-ref ref="appender1" /><!-- 与前面的通道 id 相对应 -->
</root>

</log4j:configuration>
```

添加 BufferSize，对比监控 Windows 资源监控磁盘队列。

```
<param name="BufferSize" value="8192" />
<param name="ImmediateFlush" value="false" />
<param name="BufferedIO" value="true" />
```

如果把 BufferSize 调大，例如 819200，反而响应时间会加大。

代码效率性能测试与优化

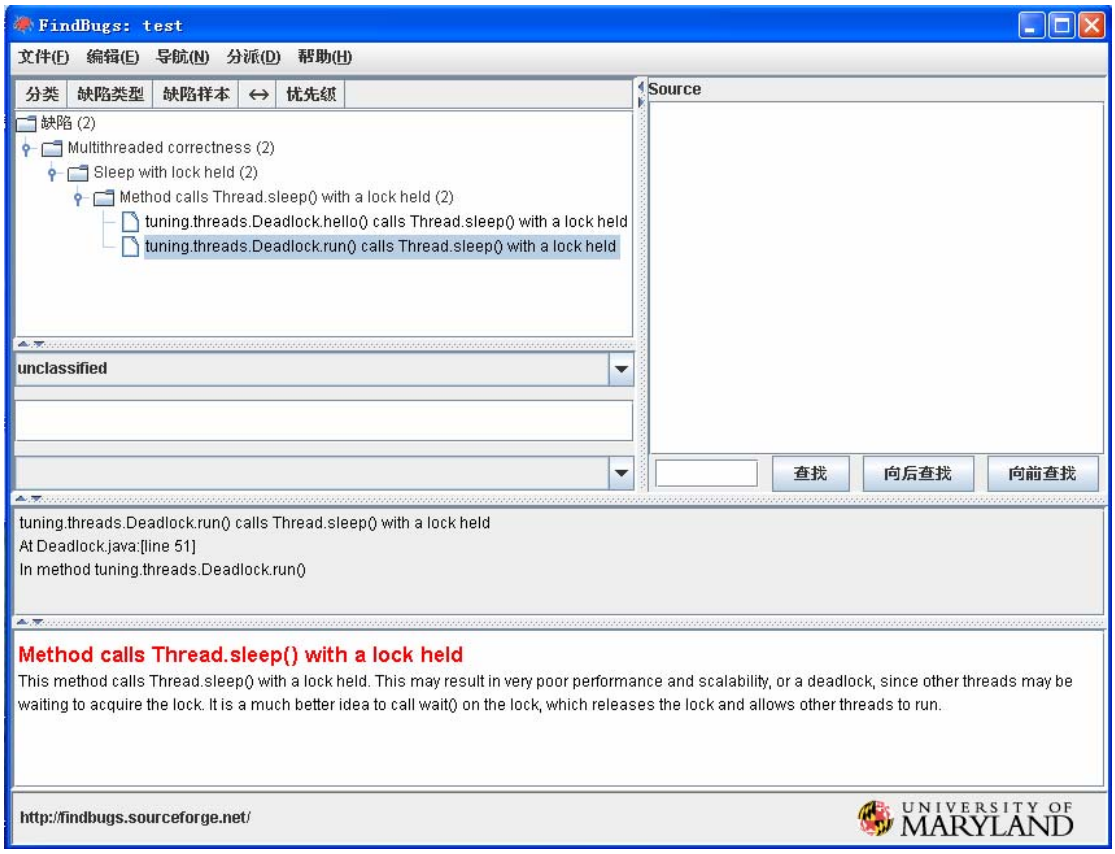
D:\PrefTest\案例\java 代码性能优化

线程死锁

```
package tuning.threads;

public class Deadlock implements Runnable
{
   String me;
   Deadlock other;

   public synchronized void hello()
   {
      //print out hello from this thread then sleep one second.
      System.out.println(me + " says hello");
      try {Thread.sleep(1000);}
      catch (InterruptedException e) {}
   }

   public void init(String name, Deadlock friend)
   {
      //We have a name, and a reference to the other Deadlock object
      //so that we can call each other
```

```
        me = name;
        other = friend;
    }

    public static void main(String args[])
    {
        //wait as long as the argument suggests (or use 20 ms as default)
        int wait = args.length == 0 ? 20 : Integer.parseInt(args[0]);

        Deadlock d1 = new Deadlock();
        Deadlock d2 = new Deadlock();

        //make sure the Deadlock objects know each other
        d1.init("d1", d2);
        d2.init("d2", d1);

        Thread d1Thread = new Thread(d1);
        Thread d2Thread = new Thread(d2);

        //Start the first thread, then wait as long as
        //instructed before starting the other
        d1Thread.start();
        try {Thread.sleep(wait);}
        catch (InterruptedException e) {}
        d2Thread.start();
    }

    public synchronized void run()
    {
        //We say we are starting, then sleep half a second.
        System.out.println("Starting thread " + me);
        try {Thread.sleep(500);}
        catch (InterruptedException e) {}

        //Then we say we are calling the other guy's hello(), and do so
        System.out.println("Calling hello from " + me + " to " + other.me);
        other.hello();
        System.out.println("Ending thread " + me);
    }
}
```

通过代码审查可以发现这类代码问题，FindBugs 可以找出其中的问题：



线程竞争

```
package tuning.threads;

public class ThreadRace
   implements Runnable
{
   //global counter
   static int num=0;

   //public static synchronized void increment()
   public static void increment()
   {
      int n = num;
      //This next line gives the context switcher an ideal
      //place to switch context.
      System.out.print(num+" ");
      //And when it switches back, n will still be the old
      //value from the old thread.
      num = n + 1;
   }
```

```
    public static void main(String args[])
    {
        ThreadRace d1 = new ThreadRace();
        ThreadRace d2 = new ThreadRace();

        Thread d1Thread = new Thread(d1);
        Thread d2Thread = new Thread(d2);

        d1Thread.start();
        d2Thread.start();
    }

    public void run()
    {
        for (int i = 50; i >= 0 ; i--)
        {
            increment();
        }
    }
}
```

算法效率

代码算法的优化与代码编写的细节有很大关系。

```
package cnj;

/*
 * 需求：给定一个 String 对象，过滤掉除数字（字符'0'-'9'）以外的其它字符。要求时间开销尽可能小。
 * 过滤函数的原型如下：String filter(String str);
 */

public class StringFilter {

    /**
     * @param args
     */
    public static void main(String[] args) {
        String input = "D186783E36B721651E8AF96AB1C4000B";
        String str = "";
```

```
        long nBegin = System.currentTimeMillis();
        for(int i=0; i<1024*1024; i++){
            str = filter5(input);   //此处调用某个具体的过滤函数
        }
        long nEnd = System.currentTimeMillis();
        System.out.println(nEnd-nBegin);
        System.out.println(str);
}
```

```
// ----------------------------------------------------------------------

private static String filter1(String strOld) {
        String strNew = new String();
        for (int i = 0; i < strOld.length(); i++) {
            if ('0' <= strOld.charAt(i) && strOld.charAt(i) <= '9') {
                strNew += strOld.charAt(i);
            }
        }
        return strNew;
}
```

```
// 连字符串拼接需要用 StringBuffer 来优化
private static String filter2(String strOld) {
        StringBuffer strNew = new StringBuffer();
        for (int i = 0; i < strOld.length(); i++) {
            if ('0' <= strOld.charAt(i) && strOld.charAt(i) <= '9') {
                strNew.append(strOld.charAt(i));
            }
        }
        return strNew.toString();
}
```

```
//先把 strOld.charAt(i)赋值给 char 变量，节省了重复调用 charAt()方法的开销；
//另外把 strOld.length()先保存为 nLen，也节省了重复调用 length()的开销
private static String filter3(String strOld) {
        StringBuffer strNew = new StringBuffer();
```

```
        int nLen = strOld.length();
        for (int i = 0; i < nLen; i++) {
            char ch = strOld.charAt(i);
            if ('0' <= ch && ch <= '9') {
                strNew.append(ch);
            }
        }
        return strNew.toString();
    }
```

```
    // 通过 StringBuffer 的构造函数设置初始的容量大小，可以有效避免 append()追加字符时重新分配内存，从而提高性能。
    private static String filter4(String strOld) {
        int nLen = strOld.length();
        StringBuffer strNew = new StringBuffer(nLen);
        for (int i = 0; i < nLen; i++) {
            char ch = strOld.charAt(i);
            if ('0' <= ch && ch <= '9') {
                strNew.append(ch);
            }
        }
        return strNew.toString();
    }
```

```
    // 首先，虽然 filter5 有一个字符数组的创建开销，但是相对于 filter4 来说，StringBuffer 的构造函数内部也会有字符数组的创建开销。两相抵消。
    // 所以 filter5 比 filter4 还多节省了 StringBuffer 对象本身的创建开销。
    // 其次，由于 StringBuffer 是线程安全的（它的方法都是 synchronized），因此调用它的方法有一定的同步开销，而字符数组则没有，
    // 这又是一个性能提升的地方。
    private static String filter5(String strOld) {
        int nLen = strOld.length();
        char[] chArray = new char[nLen];
        int nPos = 0;
        for (int i = 0; i < nLen; i++) {
            char ch = strOld.charAt(i);
            if ('0' <= ch && ch <= '9') {
                chArray[nPos] = ch;
                nPos++;
```

```
                }
            }
            return new String(chArray, 0, nPos);
        }

}
```

FindBugs 能指出 filter1 方法中的代码性能问题：