木桶理论与性能的瓶颈：

木桶理论有称为短板理论. 其核心思想是：一只木桶装水的多少, 并不取决于桶壁最高的那块木板, 而是取决于最短的那块木板. 将这个理论应用到系统性能优化上, 可以这么理解, 即使系能拥有充足的内存资源和CPU资源. 但是如果磁盘I/O性能低下, 那么系统总体性能是取决于当前最慢的磁盘I/O速度. 而不是当前最优的CPU或者内存. 在这种情况下. 如果需要进一步提升系统性能, 优化内存或者CPU是毫无用途的. 只有提高磁盘I/O 性能才能对系统的整体性能进行优化. 而此时, 磁盘I/O就是系统性能的瓶颈.

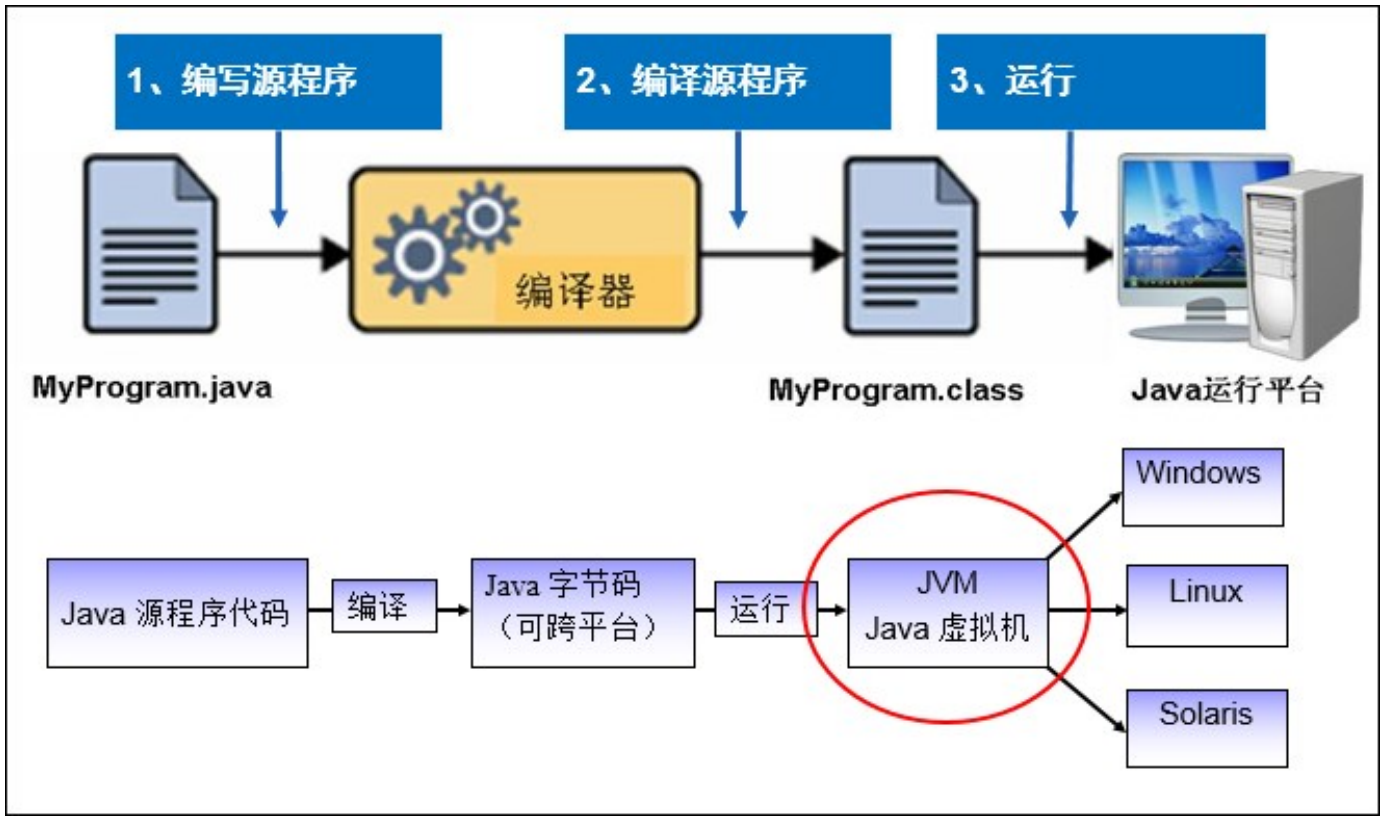
根据应用的特点不同, 任何的计算机资源都有可能成为系统的瓶颈. 其中最有可能成为系统瓶颈的计算资源如下：

1. 磁盘I/O
2. 网络操作
3. CPU
4. 锁竞争
5. 内存

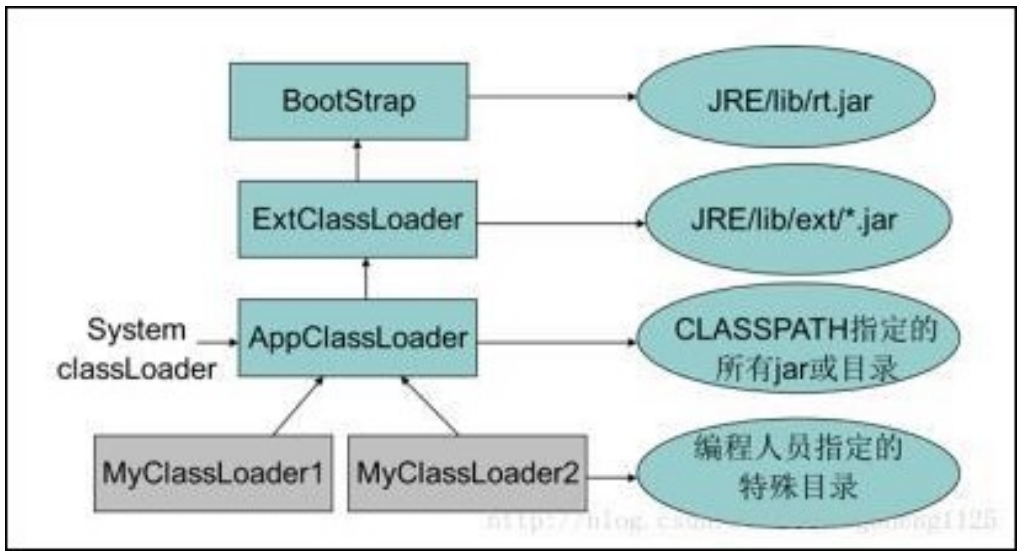
性能调优的层次：

- 设计调优：单例、工厂、代理、监听模式
- 代码调优：Stirng、StringBuffer、LinkedList、ArrayList
- JVM调优：堆、栈、方法区、常量池.....
- 数据库调优：索引、分区表、缓存
- 操作系统调优：虚拟内存大小、磁盘块大小

首先我们先来回顾 Java的整个执行流程



Java虚拟机中可以安装多个类加载器, 系统默认三个主要的类加载器, 每个类加载器各有分工.不同的类加载器负责加载特定位置的类, 类加载器本身也是一个Java类. 但是BootStrap本身不是类, 因此它本身不需要被类加载器加载. BootStrap本身在Java内核中.当然我们也可以自定义类加载器.类加载器的层次结构如下:



获取类加载器的代码如下：

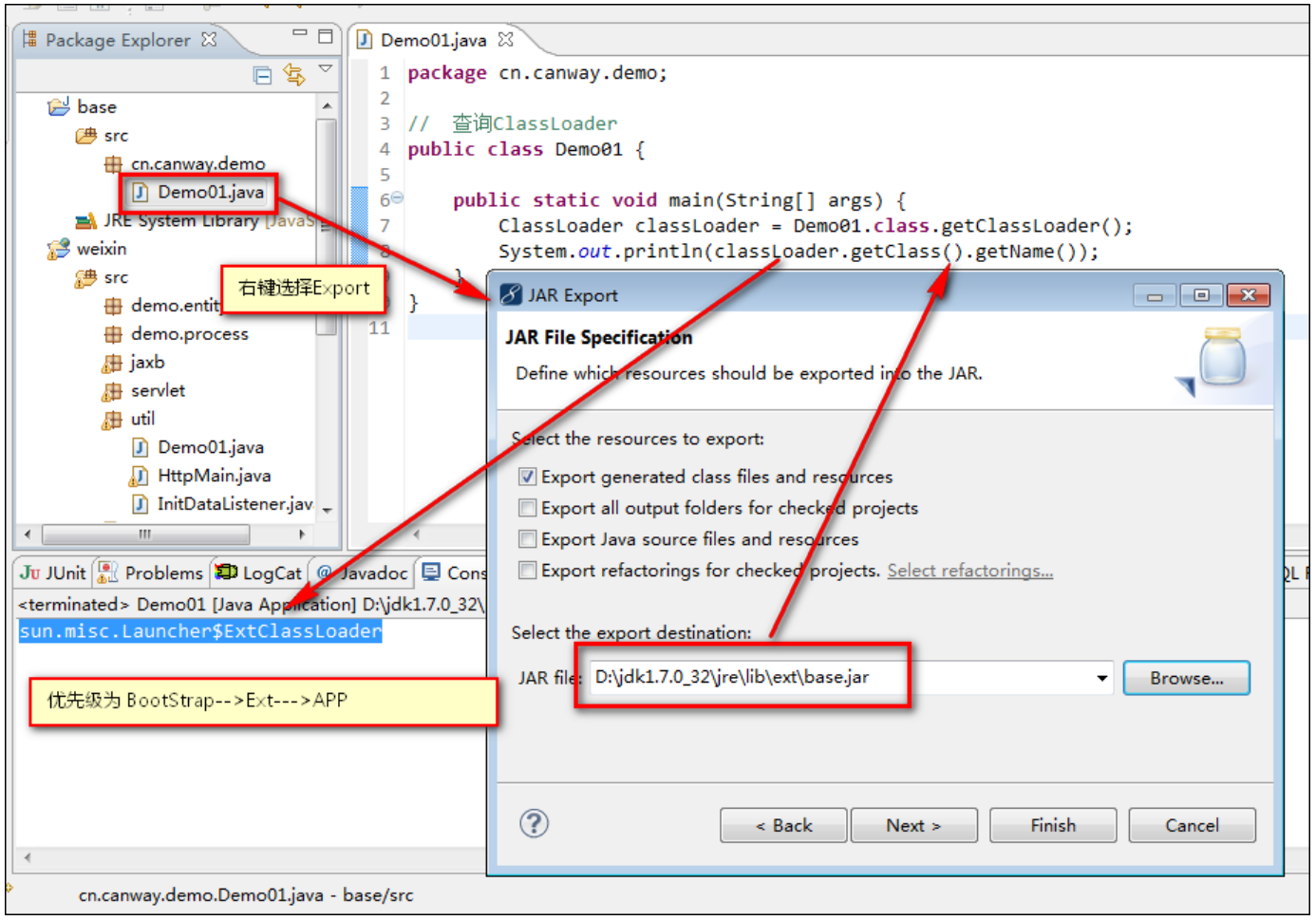
```
1.  public class MyClass {
2.      // 通过加载获取当前类的相关类加载器,我们会发现BootStrap并不是一个类. 而是用
    C/C++编写的引导程序
3.      public static void main(String[] args) {
4.          ClassLoader classLoader = MyClass.class.getClassLoader();
5.          while(classLoader!=null){
6.              System.out.println(classLoader.getClass().getName());
7.              // 如果类加载器的父类加载器就是引导类加载器，则此方法将在这样的实现
    中返回 null
8.              classLoader = classLoader.getParent();
```

```
        }
    }
}
```

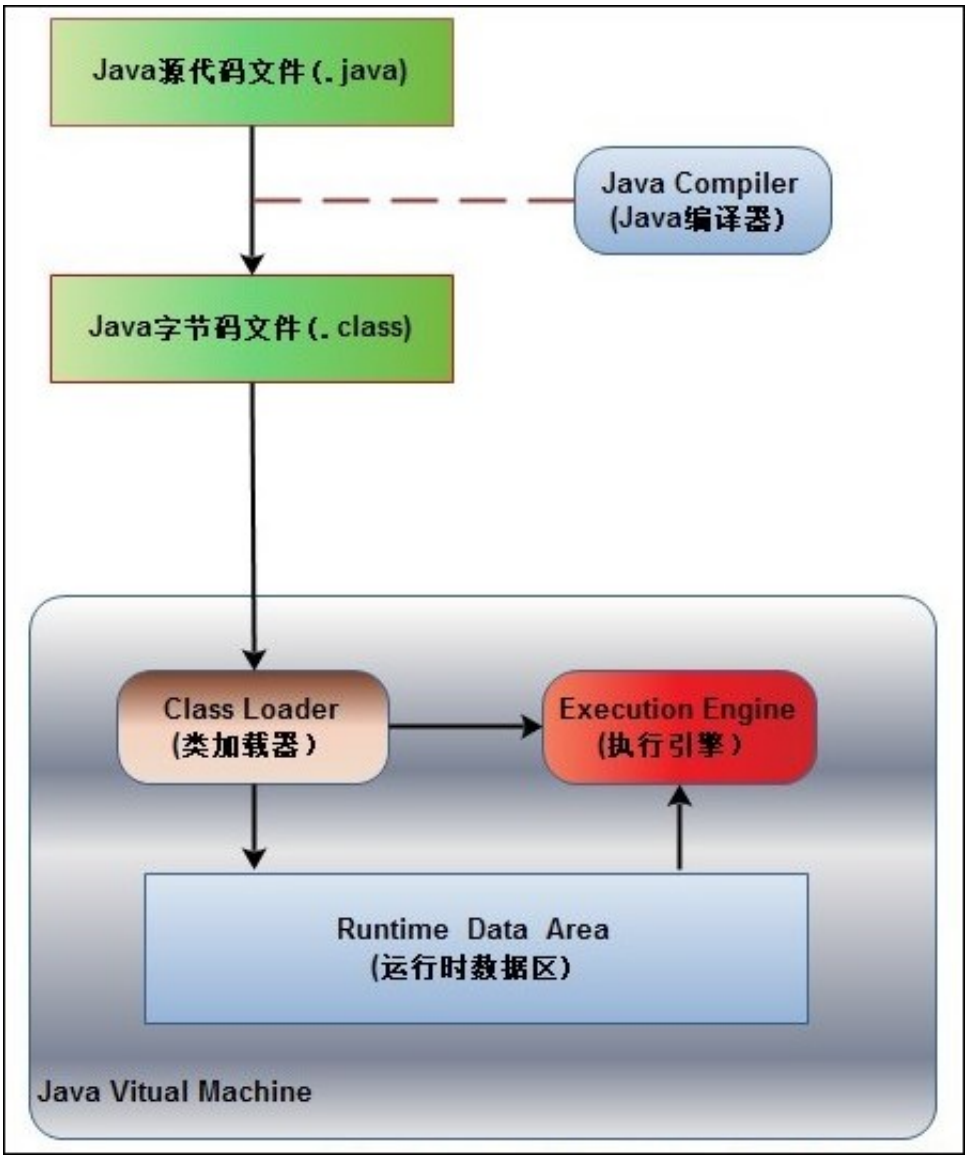
- JRE/lib/rt.jar：Java运行时核心的Jar包, 里面有 String、System、Connection等我们常见的API
- JRE/lib/ext.jar：ExtClassLoader从名称大家可以看出来用来加载ext的类加载器. 主要用来加载JRE/lib/ext/*.jar文件 (一般用来放一些公司自己设计的Jar文件)
- AppClassLoader主要用来加载我们自己项目的class文件通过ClassLoader的源码可以看出来. LoadClass会先调用父类的loadClass类加载器来加载的

```
protected Class<?> loadClass(String name, boolean resolve)
        Class c = findLoadedClass(name);
                if (parent != null) {
                        c = parent.loadClass(name, false);
                } else {
                        c = findBootstrapClassOrNull(name);
                }
        }
}
```

他们之间的优先级顺序如下：BootStrap--->Ext--->App

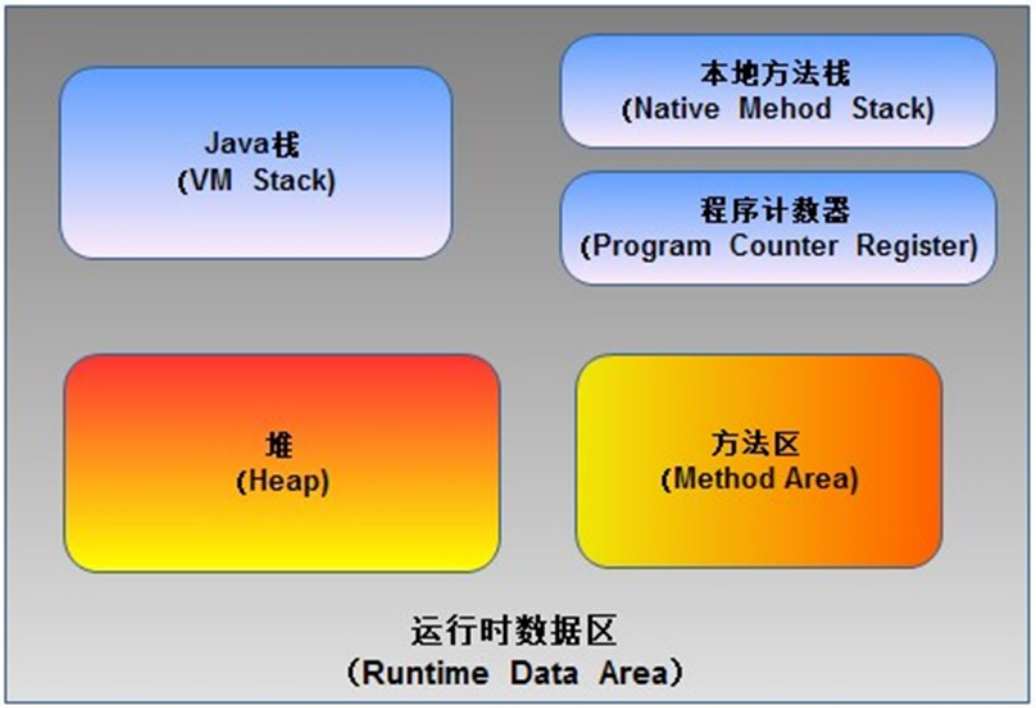


通过类加载器加载类的流程图如下：



思考：我们自己是否可以定义类加载器，有什么作用？

JVM内存结构图介绍：可以参考如下博客文章：http://www.cnblogs.com/dolphin0520/p/3613043.html



程序计数器：程序计数器（Program Counter Register），也有称作为PC寄存器

方法区：它与堆一样，是被线程共享的区域.在方法区中，存储了每个类的信息（包括类的名称、方法信息、字段信息）、静态变量、常量以及编译器编译后的代码等

在方法区中有一个非常重要的部分就是运行时常量池，它是每一个类或接口的常量池的运行时表示形式，在类和接口被加载到JVM后，对应的运行时常量池就被创建出来。当然并非Class文件常量池中的内容才能进入运行时常量池，在运行期间也可将新的常量放入运行时常量池中，比如String的intern方法

堆：Java中的堆是用来存储对象本身的以及数组（当然，数组引用是存放在Java栈中的). 只不过和C语言中的不同，在Java中，程序员基本不用去关心空间释放的问题，Java的垃圾回收机制会自动进行处理。因此这部分空间也是Java垃圾收集器管理的主要区域。另外, 堆是被所有线程共享的，在JVM中只有一个堆

栈：Java栈中存放的是一个个的栈帧，每个栈帧对应一个被调用的方法，在栈帧中包括局部变量表(Local Variables)、操作数栈(Operand Stack)、指向当前方法所属的类的运行时常量池（运行时常量池的概念在方法区部分会谈到）的引用(Reference to runtime constant pool)、方法返回地址(Return Address)和一些额外的附加信息。当线程执行一个方法时，就会随之创建一个对应的栈帧，并将建立的栈帧压栈. 由于每个线程正在执行的方法可能不同，因此每个线程都会有一个自己的Java栈，互不干扰

为了了解Java的内存结构与大家的Java基础. 我们来看一个小的Demo

```
1.  public class StringDemo {
2.      /* 思考两个问题: 打印的结果? 代码中一共创建的几个对象? */
3.      public static void main(String[] args) {
4.          String str1 = "hello";
5.          String str2 = "hello";
6.          String str3 = new String("hello");
7.          System.out.println(str1 == str2);
8.          System.out.println(str1.equals(str2));
9.          System.out.println(str2 == str3);
10.         System.out.println(str2.equals(str3));
11.         String str4 = str3;
12.         System.out.println(str3 == str4);
13.         System.out.println(str3.equals(str4));
14.         str3 += " world";
15.         System.out.println(str3 == str4);
16.         System.out.println(str3.equals(str4));
17.         StringBuffer str5 = new StringBuffer("hello");
18.         StringBuffer str6 = str5;
19.         str5.append(" world");
20.         System.out.println(str5 == str6);
```

```
        System.out.println(str5.equals(str6));
    }
}
```

在Java中其实==和equals本来是相同的, 只是String重写的equals

```
public boolean equals(Object anObject) {
        if (this == anObject) {
            return true;
        }
        if (anObject instanceof String) {
            String anotherString = (String) anObject;
            int n = value.length;
            if (n == anotherString.value.length) {
                char v1[] = value;
                char v2[] = anotherString.value;
                int i = 0;
                while (n-- != 0) {
                    if (v1[i] != v2[i])
                            return false;
                    i++;
                }
                return true;
            }
        }
        return false;
    }
```

练习：画出文章开头部分String测试代码的内存结构图

接下来我们通过一些JVM运行时常见的参数设置来深刻了解JVM各部分的结构

`java -XX:+TraceClassLoading`

```
[Loaded java.lang.Object from shared objects file]
[Loaded java.io.Serializable from shared objects file]
[Loaded java.lang.Comparable from shared objects file]
...
[Loaded cn.jxy.demo.MenuOut2 from file:/D:/workspace/demo/bin/]
[Loaded java.lang.Void from shared objects file]
[Loaded java.lang.Shutdown from shared objects file]
[Loaded java.lang.Shutdown$Lock from shared objects file]
```

在Java命令后面添加参数： -verbose:gc 则可以打印GC回收信息

```
// [Full GC 1538K->369K(15872K), 0.0039587 secs]
// [Full GC 1538K->1393K(15872K), 0.0040752 secs]
public static void main(String[] args) {
    byte[] b=new byte[1024*1024];
    // b = null;
    System.gc();
}
```

箭头前后的数据1538K和369K分别表示垃圾收集GC前后所有存活对象使用的内存容量，说明有1538K-369K=1169K的对象容量被回收，括号内的数据15872K为堆内存的总容量，收集所需要的时间是0.039578秒（时间在每次执行的时候会有所不同）

采用java –Xmx -Xms 来分配堆的大小[代码为: java -Xmx20m -Xms5m]

```
// 手动指定 java - -Xmx20m -Xms5m 的空间大小
public static void main(String[] args) throws Exception {
        byte[] b = new byte[1*1024*1024];
        // 返回JVM可以使用的最大可以分配内存量,字节为单位
        System.out.println("Xmx=" + Runtime.getRuntime().maxMemory() / 1024.0 / 1024 + "mb");
        // 返回JVM系统总分配到的内存数量
        System.out.println("total=" + Runtime.getRuntime().totalMemory()/1024.0/1024 + "mb");
        // 目前free空闲可以使用的堆内存数量  total-free 可以核算出已经使用的堆大小
        System.out.println("free=" + Runtime.getRuntime().freeMemory()/1024.0/1024 + "m");
}
```

显示效果如下, 默认情况下jvm在初始的时候会根据Xms来分配空间.

```
Xmx=19.375mb
total=4.875mb
free=3.4565048217773438m
```

在jvm堆不够的情况下才会显示扩容. 测试代码如下.

```
public static void main(String[] args) throws Exception {
        byte[] b = new byte[4*1024*1024];
        // 返回JVM可以使用的最大可以分配内存量,字节为单位
        System.out.println("Xmx=" + Runtime.getRuntime().maxMemory() / 1024.0 / 1024 + "mb");
        // 返回JVM系统总分配到的内存数量
        System.out.println("total=" + Runtime.getRuntime().totalMemory()/1024.0/1024 + "mb");
        // 目前free空闲可以使用的堆内存数量  total-free 可以核算出已经使用的堆大小
        System.out.println("free=" + Runtime.getRuntime().freeMemory()/1024.0/1024 + "mb");
```

显示效果如下:

```
Xmx=19.375mb
total=9.00390625mb
free=4.580406188964844mb
```

当然如果我们的程序超过了堆设置的Xmx则会立即抛出内存溢出异常： `java.lang.OutOfMemoryError: Java heap space`

```
// 手动指定 java - -Xmx20m -Xms5m 的空间大小
public static void main(String[] args) throws Exception {
    byte[] b = new byte[30*1024*1024];
}
```

栈的空间分配 -Xss 首先了解它的一些概念：

- 通常只有几百KB,
- JDK5.0以后每个线程堆栈大小为1M,以前每个线程堆栈大小为256K.更具应用的线程所需内存大小进行 调整.在相同物理内存下,减小这个值能生成更多的线程.
- 栈的大小直接影响到函数调用的深度
- 每个线程都有独立的栈空间 (如果栈空间定太大, 则可以同时所运行的线程数量会减少, 如果栈空间太少.则函数调用的深度又会受到影响)
- 局部变量、参数、分配在栈上

如果该值设置过大,就有影响到创建栈的数量,如果是多线程的应用,就会出现内存溢出的错误,我们来显示一个通过修改-Xss大小影响创建线程数量的案例

```
public static class MyThread extends Thread{

    @Override
    public void run() {
        try{
            Thread.sleep(10000); // 保证线程不退出
        }catch (Exception e) {
            e.printStackTrace();
        }
    }
}

public static void main(String args[]){
    int i = 0;
    try{
        while(i<10000){
            new MyThread().start();  // 开启大量新线程
            i++;
        }
    }catch (OutOfMemoryError e) {
        System.out.println("count thread is " + i);
    }
}
```

我们测试下默认未分配时支持调用函数调用深度(溢出的深度:7863)

```
public class MemoryOut {

    private int count = 0;

    public void rec() {
```

```
        count++;
        rec();
    }

    public void test() {
        try {
            rec();
            // 思考写Exception为什么不捕获异常
        } catch (Throwable e) {
            System.out.println("溢出的深度:" + count);
        }
    }

    public static void main(String[] args) {
        new MemoryOut().test();
    }
}
```

如果栈空间不变的情况下. 如果给递归函数添加变量. 则调用深度会减少(溢出的深度:3438) 原理很简单, 局部变量都会存储在栈帧中

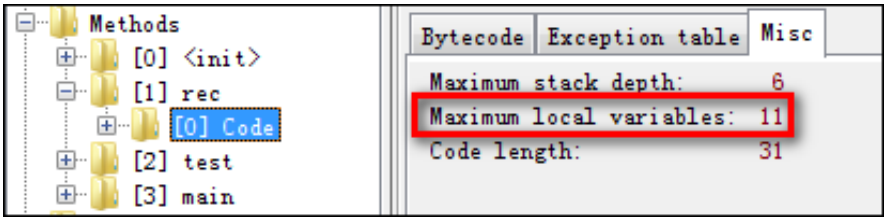
```
public class MemoryOut {

    private int count = 0;
    // 如果栈空间不变的情况下. 如果给递归函数添加变量. 则调用深度会减少(溢出的深度:3438) 原理很简单, 局部变量都会存储在栈帧中
    public void rec(int a,int b,int c) {
        int d,e,f;
        d = a; e = b; f= c;
        count++;
        rec(a,b,c);
    }

    public void test() {
        try {
            rec(1,2,3);
            // 思考写Exception为什么不捕获异常
        } catch (Throwable e) {
            System.out.println("溢出的深度:" + count);
        }
    }

    public static void main(String[] args) {
        new MemoryOut().test();
    }
}
```

在栈帧中, 与性能调优关系最密切的部分就是局部变量表. 局部变量表用于存放方法的参数和方法内部的局部变量, 局部变量表与 "字" 为单位进行划分 , 一个字为32位长度(也就是4个字节). 对于long和double类型的变量. 则占用2个字. 其余类型使用1个字. 在方法执行时, 虚拟机使用局部变量完成数据的传递. 注意虚拟机还会将this 作为参数传递给当前方法. 使用jclasslib工具可以查看class文

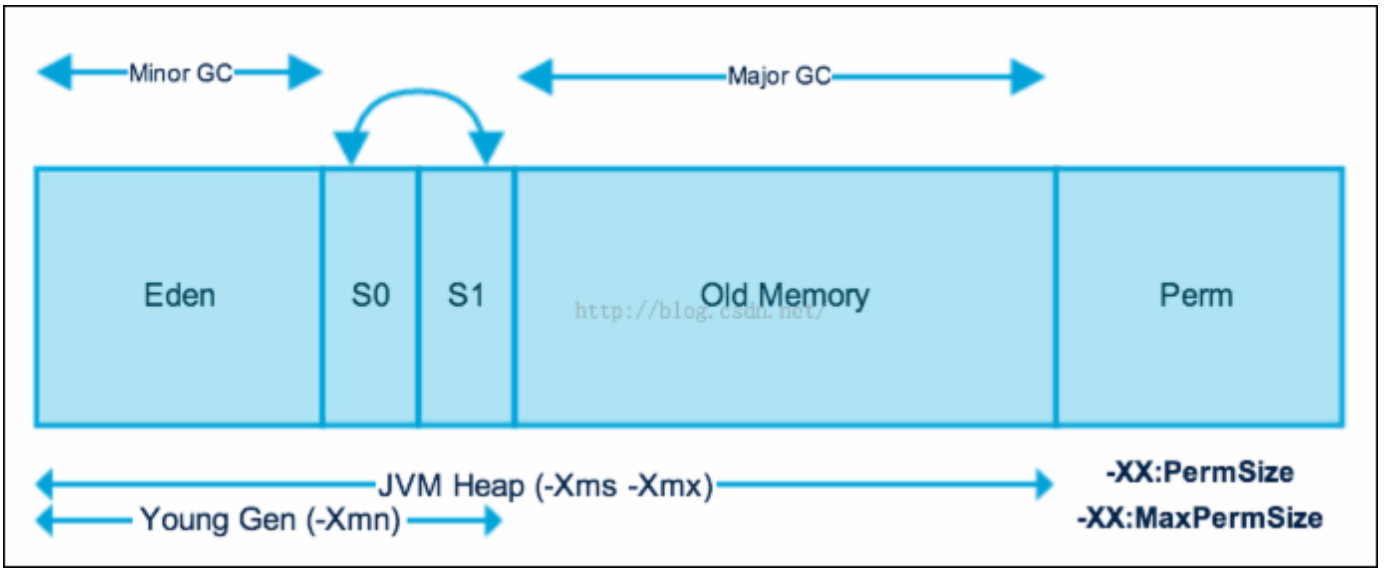
件中每个方法所分配的最大局部变量表的容量. jclasslib工具是开源软件. 它可以查看class文件的结构, 包括常量池、接口、属性、方法. 还可以用于查看方法的字节码. 可以帮助读者对class文件做较为深入的研究.

```
1.  // 如果栈空间不变的情况下. 如果给递归函数添加变量. 则调用深度会减少(溢出的深度:343
    8) 原理很简单, 局部变量都会存储在栈帧中
2.  public void rec(int a, double b, long c) {
3.      int d; double e; long f;
4.      d = a;
5.      e = b;
6.      f = c;
7.      count++;
8.      rec(d, e, f);
9.  }
```

根据上面的推断：在局部变量表中 a b c d e f + this = 11个字(占据了44个字节)



进一步了解堆的分布：新生代、老生代，我们先看一张比较经典的图



JVM区域总体分两类，heap区和非heap区。heap区又分：Eden Space（伊甸园）、S0 S1 Space(幸存者区)、Tenured Gen（老年代-养老区）。 非heap区又分：Code Cache(代码缓存区)、Perm Gen（永久代）、Jvm Stack(java虚拟机栈)、Local Method Statck(本地方法栈),为了帮助大家理解新生代和老年代, 此处举一个类似的案例:

1. 一个人（对象）出来（new 出来）后会在Eden Space（伊甸园）无忧无虑的生活，直到GC到来打破了他们平静的生活. GC会逐一问清楚每个对象的情况，有没有钱（此对象的引用）啊，因为GC想赚钱呀，有钱的才可以敲诈嘛。然后富人就会进入Survivor Space（幸存者区），穷人的就直接kill掉
2. 并不是进入Survivor Space（幸存者区）后就保证人身是安全的，但至少可以活段时间。GC会定期

（可以自定义）会对这些人进行敲诈，亿万富翁每次都给钱，GC很满意，就让其进入了Genured Gen(养老区)。万元户经不住几次敲诈就没钱了，GC看没有啥价值啦，就直接kill掉了

3. 进入到养老区的人基本就可以保证人身安全啦，但是亿万富豪有的也会挥霍成穷光蛋，只要钱没了，Full GC还是kill掉

JVM 为什么要采用分区策略, 方便分代收集

- 即将内存分为几个区域，将不同生命周期的对象放在不同区域里；
- 在GC收集的时候，频繁收集生命周期短的区域(Young area)；
- 比较少的收集生命周期比较长的区域(Old area)；
- 基本不收集的永久区(Perm area)，此处的永久区就是我们说的方法区.

思考：一个有内存泄露的程序, 它的新生代、年老代、符合什么样的特征？

---

jps(Java Virtual Machine Process Status Tool)是JDK1.5提供的一个显示当前所有java进程pid的命令，它的作用是显示当前系统的java进程情况及进程id。我们可以通过它来查看我们到底启动了几个java进程（因为每一个java程序都会独占一个java虚拟机实例），并可通过opt来查看这些进程的详细启动参数

- -m 输出传递给main方法的参数，在嵌入式jvm上可能是null
- -l 输出应用程序main class的完整package名或者应用程序的jar文件完整路径名

```
C:\Users\Administrator>jps
5312 Jps
4836 JavaDemo
1996 PULSEI~1.JAR
```

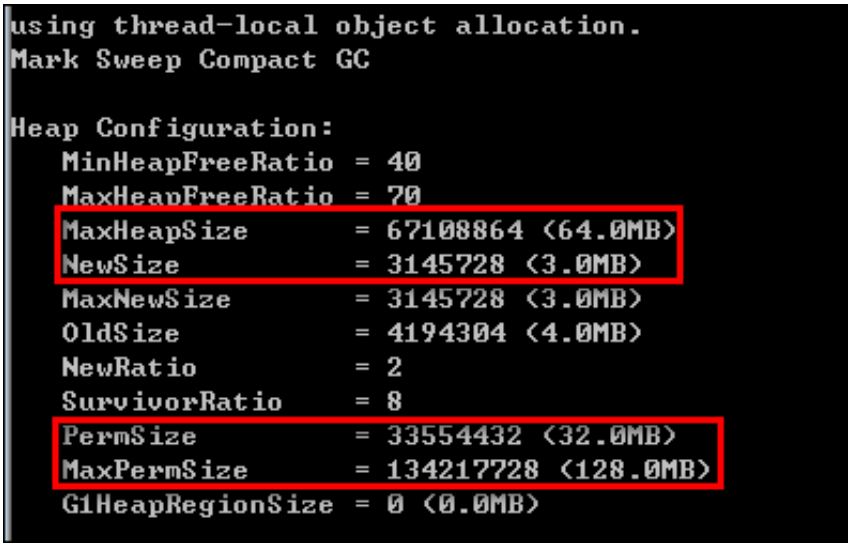
添加了 -m -l 的参数如下

```
C:\Users\Administrator>jps -m -l
4728 sun.tools.jps.Jps -m -l
4836 cm.jvm.demo.JavaDemo 1 2 3
```

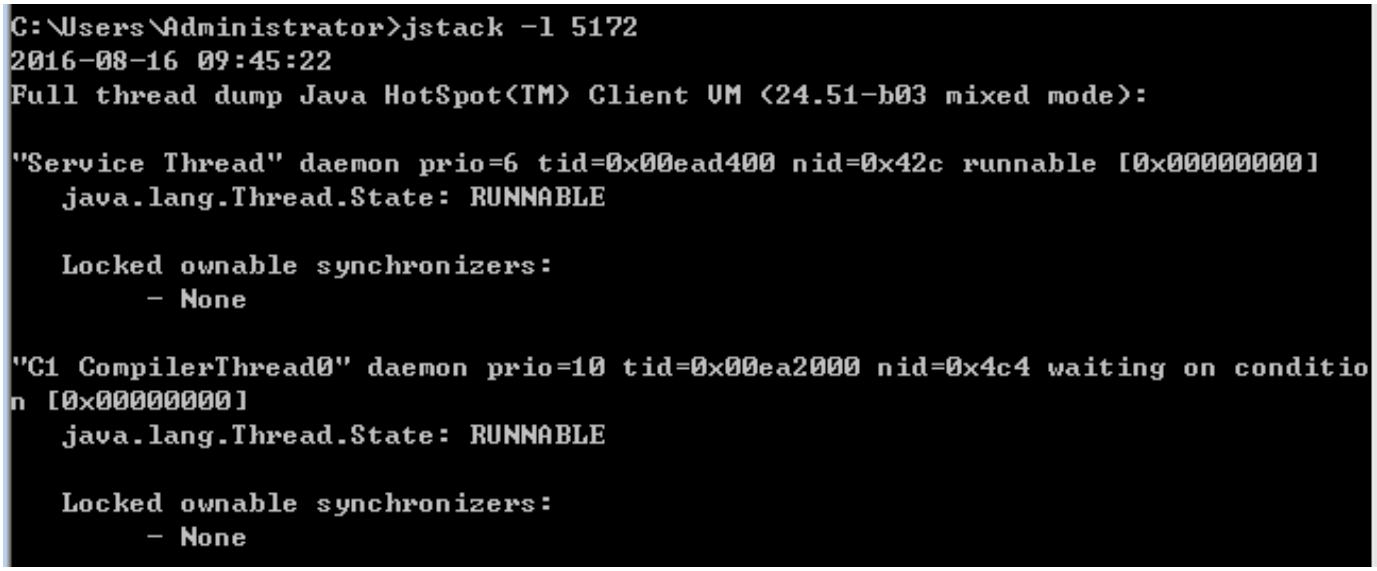
jinfo打印出java进程的配置信息, 包括 jvm参数, 系统属性等 常用格式,

```
C:\Users\Administrator>jinfo
Usage:
    jinfo [option] <pid>
        (to connect to running process)
    jinfo [option] <executable <core>
        (to connect to a core file)
    jinfo [option] [server_id@]<remote server IP or hostname>
        (to connect to remote debug server)

where <option> is one of:
    -flag <name>         to print the value of the named VM flag
    -flag [+|-]<name>    to enable or disable the named VM flag
    -flag <name>=<value> to set the named VM flag to the given value
    -flags               to print VM flags
    -sysprops            to print Java system properties
    <no option>          to print both of the above
    -h | -help           to print this help message
```

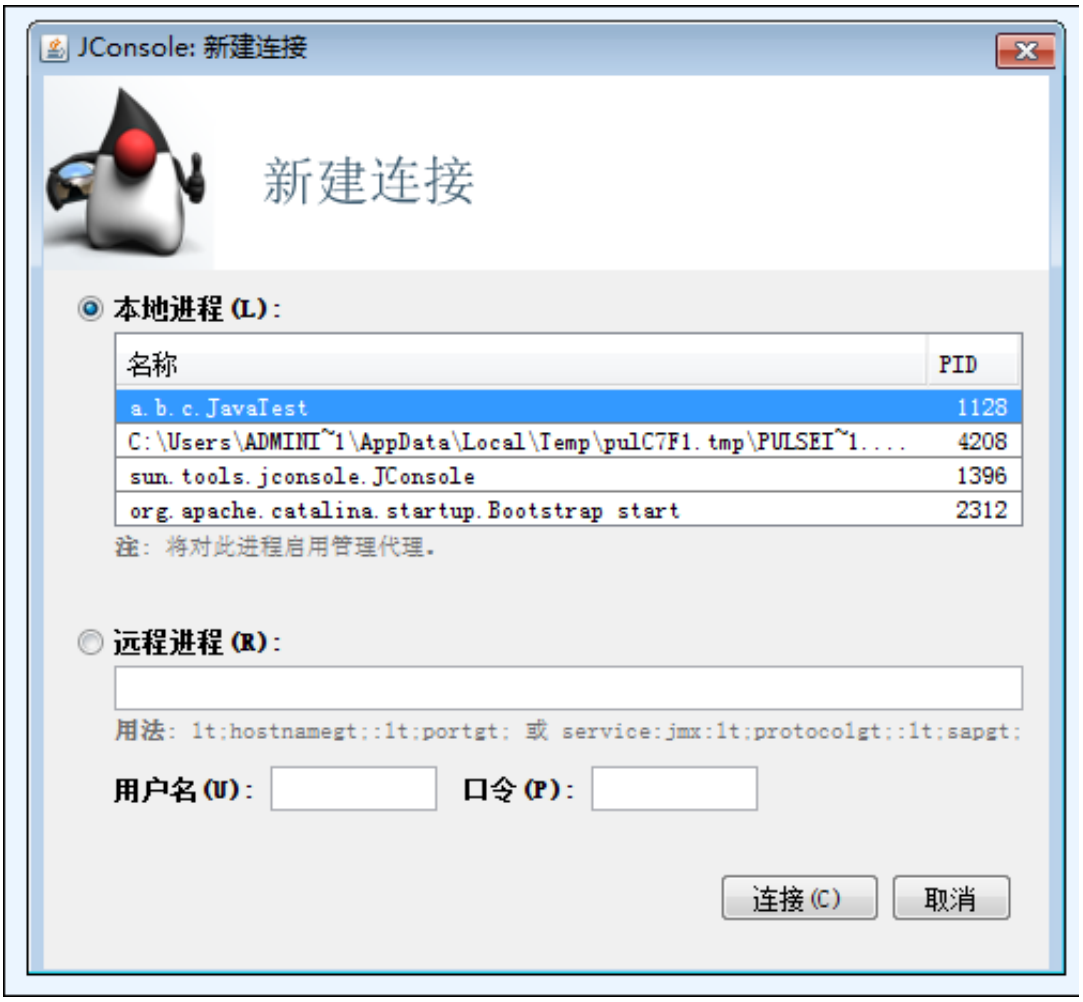
Jmap：可以生成堆快照, 查看内存的各种信息, 注意此快照非实时 常用格式：jmap - heap pid 查看java进程的堆内存分布情况，运行时添加的参数如下：-Xmx64m -Xms32m -Xmn3m -XX:PermSize=32m -XX:MaxPermSize=128m



Jstack 用来查看某个Java进程内线程堆栈信息格式为：jstack -l pid. jstack 定位到线程堆栈, 根据堆栈信息我们可以定位到具体代码. 例如CPU问题, 死锁问题等. 实际应用中建议多几次jstack来捕捉信息



Jconsole 可视化工具的使用：在cmd命令中输入Jconsole, 如果是远程, 则采用 ip:port的形式来链接



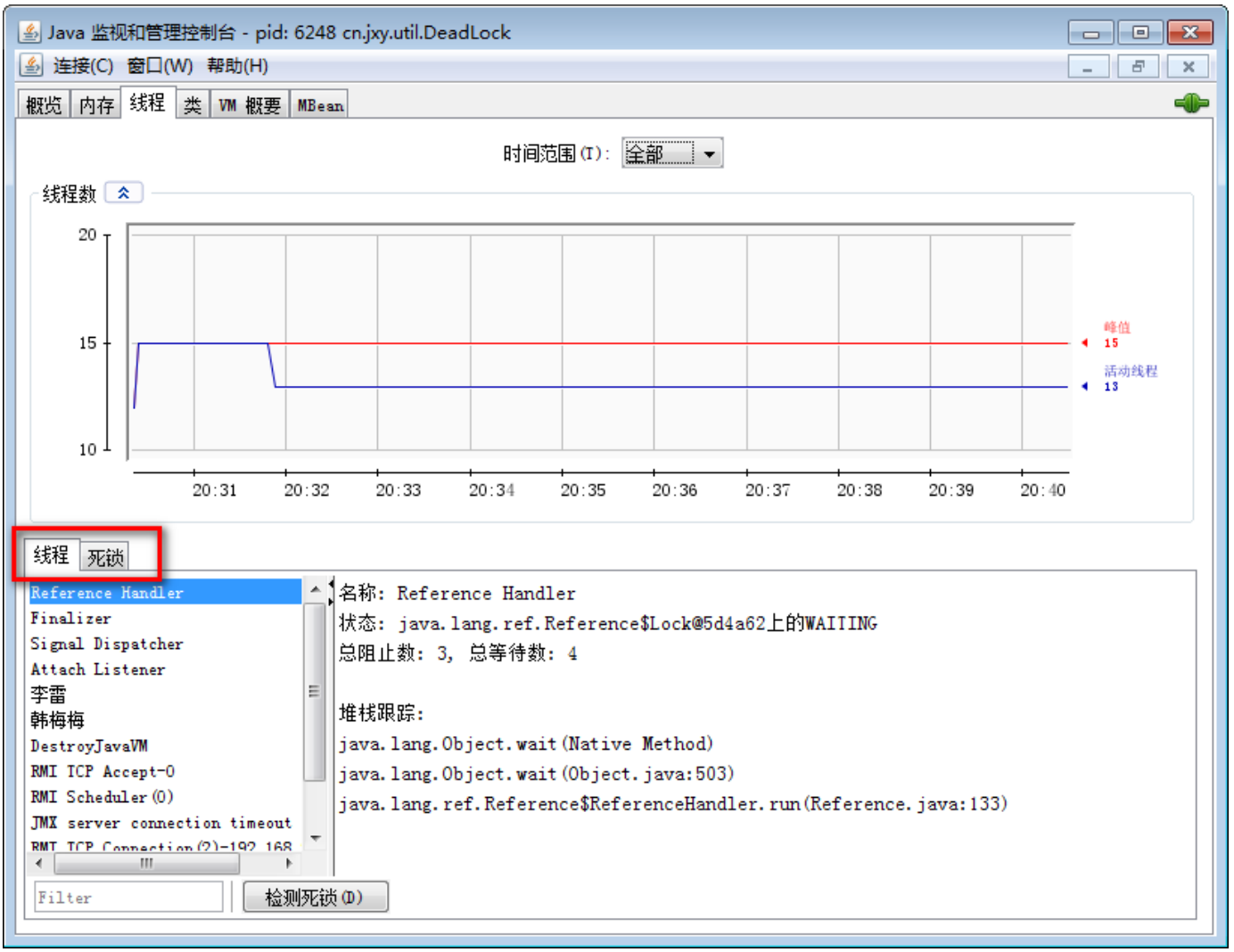
可以通过JConsole来查看线程、内存、VM概要相关信息，思考这些信息是之前哪个命令提供的

VM 概要

2016年8月21日 星期日 下午08时38分13秒 CST

连接名称: pid: 6248 cn.jxy.util.DeadLock	**运行时间:** 8 分钟
虚拟机: Java HotSpot(TM) Client VM版本 24.51-b03	**进程 CPU 时间:** 2.808 秒
供应商: Oracle Corporation	**JIT 编译器:** HotSpot Client Compiler
名称: 6248@PC-20160607LGGU	**总编译时间:** 0.298 秒
活动线程: 13	**已加装当前类:** 1, 446
峰值: 15	**已加载类总数:** 1, 446
守护程序线程: 10	**已卸载类总数:** 0
启动的线程总数: 16	
当前堆大小: 5, 239 KB	**提交的内存:** 15, 872 KB
最大堆大小: 253, 440 KB	**暂挂最终处理:** {0}对象
垃圾收集器: 名称 = 'Copy', 收集 = 8, 总花费时间 = 0.034 秒	
垃圾收集器: 名称 = 'MarkSweepCompact', 收集 = 0, 总花费时间 = 0.000 秒	
操作系统: Windows 7 6.1	**总物理内存:** 3, 607, 920 KB
体系结构: x86	**空闲物理内存:** 353, 912 KB
处理程序数: 4	**总交换空间:** 4, 194, 303 KB
提交的虚拟内存: 49, 184 KB	**空闲交换空间:** 3, 855, 732 KB

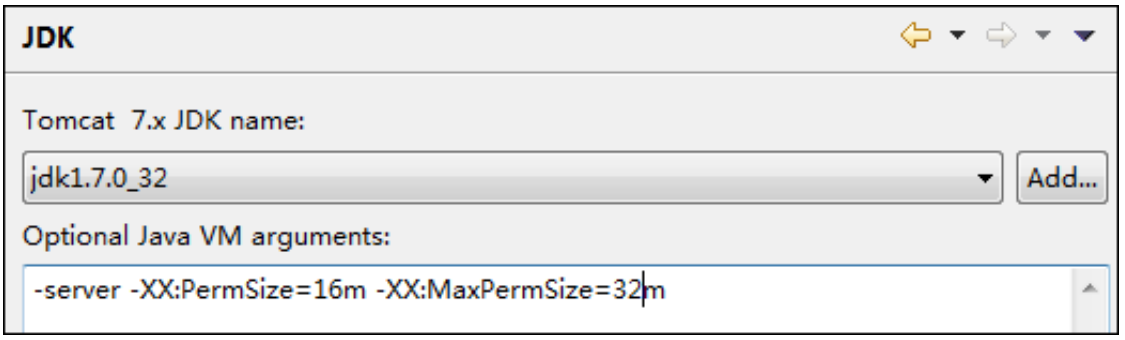
可以查看线程与死锁的相关信息



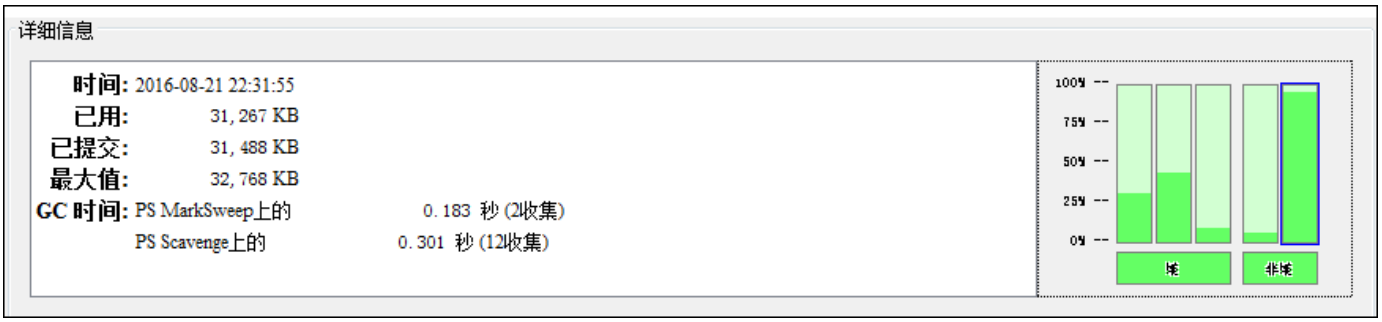
1. "Reference Handler" ： 负责处理引用
2. "Finalizer" 负责调用Finalizer方法
3. "Signal Dispatcher" 负责分发内部事件

还可以通过此工具来查看方法区的占用率. 请注意：目前的Java程序, 一般在启动的时候会大量的导入class文件, 后面只会缓慢增长. 因此持久代(方法区) 一般在运行稳定之后的占用率一般是不变的 (通过测试一个web项目设置合适的持久区的大小)

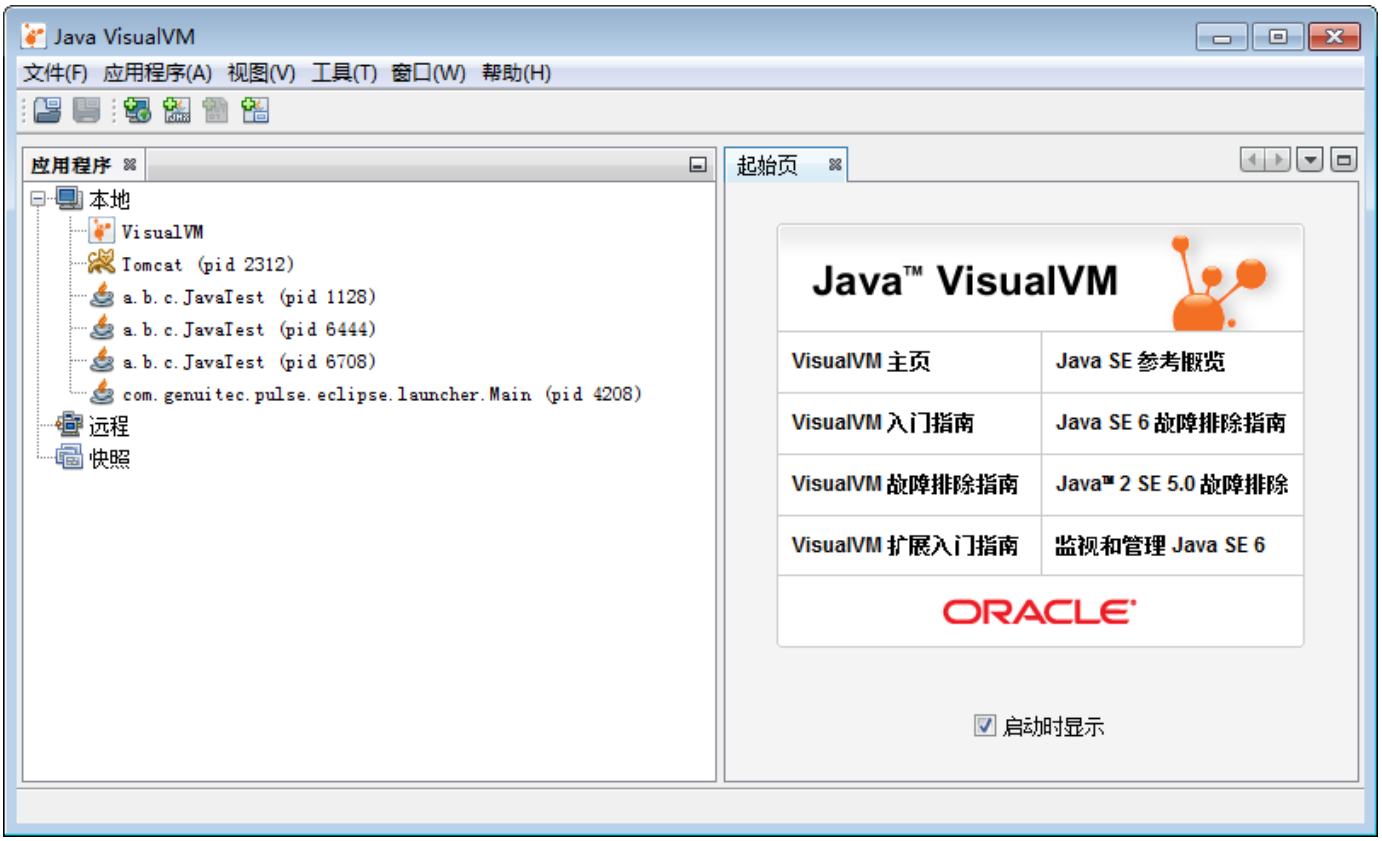
可以在tomcat 多依赖的JDK中配置如下参数：-server -XX:PermSize=16m -XX:MaxPermSize=32m



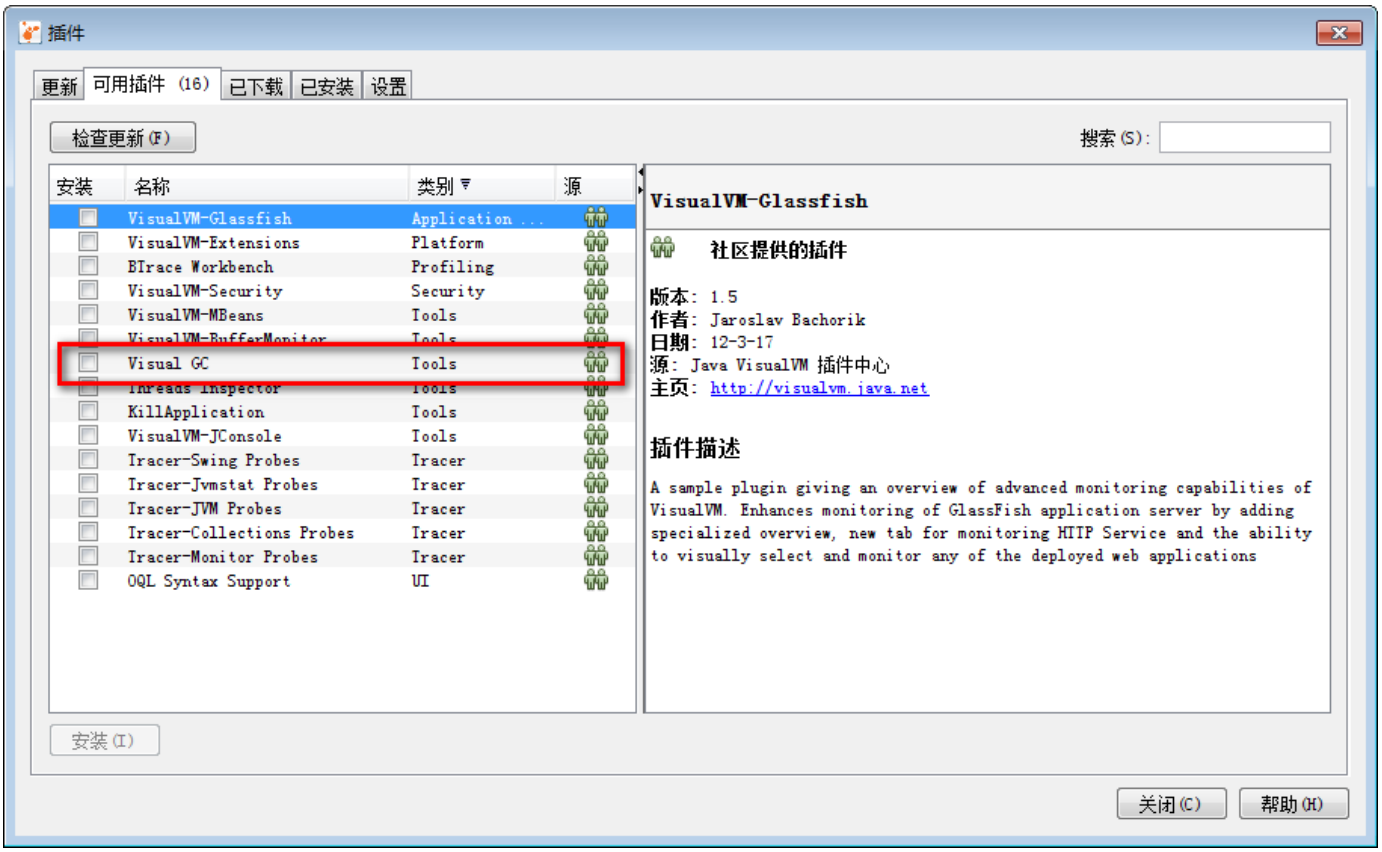
通过Jconsole检测到的Perm Space截图如下：



Jvisualvm 可以理解为Jcosole的升级版本, 可以采用它来查看当前运行时候CPU、线程的相关资源. 第一次启动时候会比较慢, 因为要验证JDK, 具体的界面截图如下：



而且在工具-->插件有很多的可用插件.



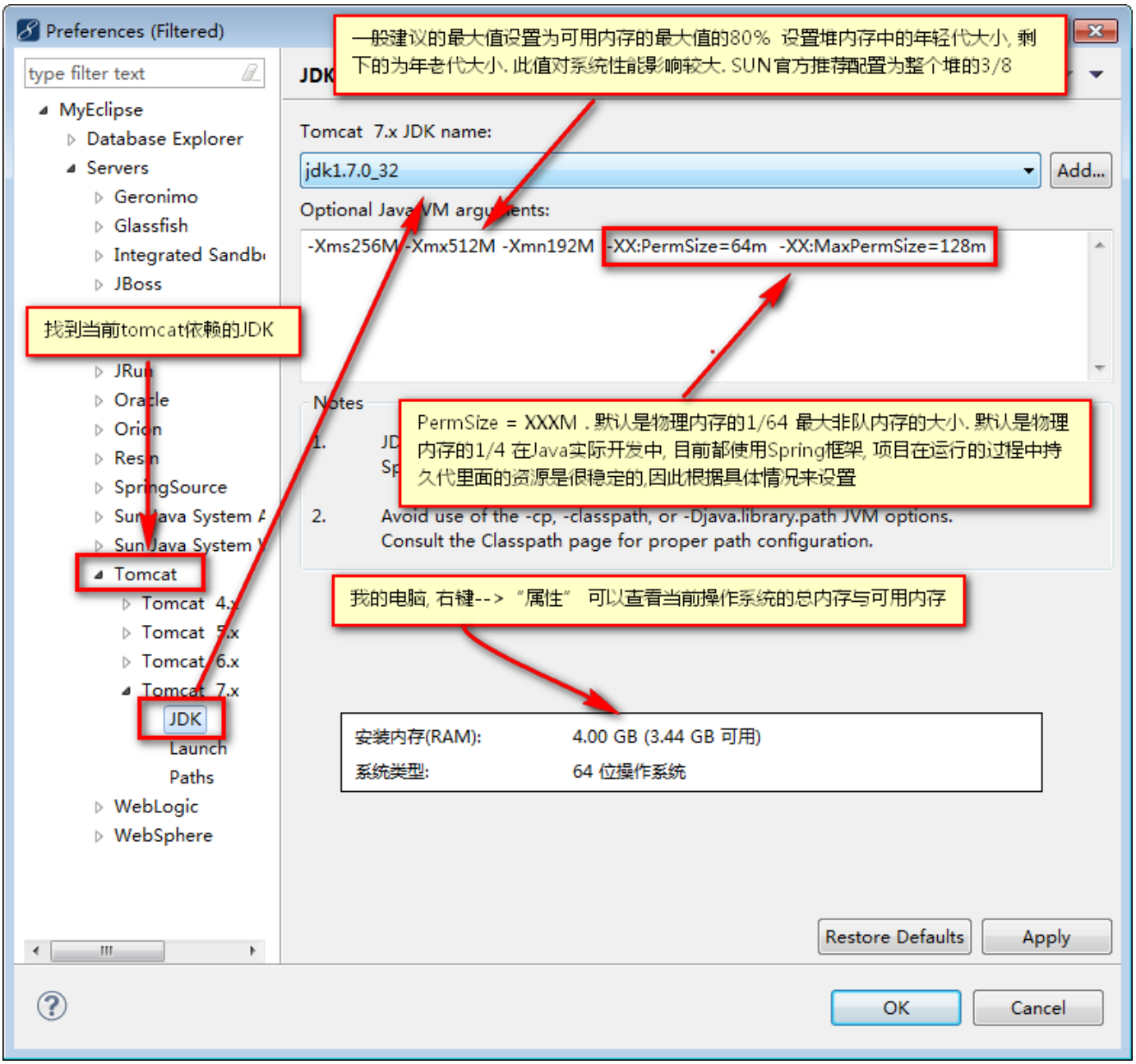
如何配置堆大小： windwo查询可用内存的80% . 堆大小、线上的堆的最大值与最小值必须一致. 省的JVM调整堆大小浪费性能. 一般建议的最大值设置为可用内存的最大值的80% 设置堆内存中的年轻代大小, 剩下的为年老代大小. 此值对系统性能影响较大. SUN官方推荐配置为整个堆的3/8

内存泄露一搬可以理解为系统资源在错误使用的情况下, 导致使用完毕的资源无法回收(或者没有回收), 从而导致新的资源分配请求无法完成.引起的系统错误. 内容泄露对系统危害比较大. 因为他可以直接导致系统崩溃

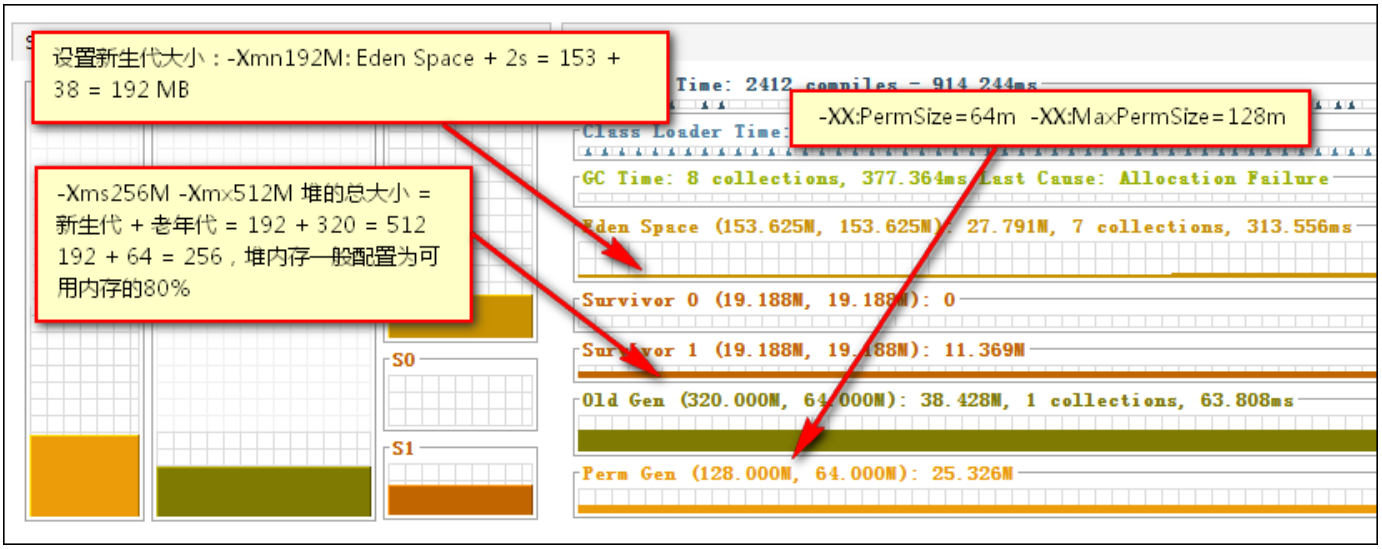
内存泄露和系统超负荷两者是有区别的, 虽然可能导致的最终结构是一样的 内存泄露是用完的资源没有回收引起错误, 而系统负荷则是系统确实没有那么多资源可以分配了.

持久代被占满情况：异常：java.lang.OutOfMemoryError：PermGen space
说明：无法为新的clas分配存储空间而引发的异常. 这个异常以前是没有的. 但是在Java反射大量使用的今天这个异常比较常见了. 主要原因就是大量动态反射生成的类不断被加载. 最终导致Perm区被占满
解决方案：增加-XX:MaxPermSize = XXXM . 默认是物理内存的1/64 最大非队内存的大小. 默认是物理内存的1/4 在数据量的很大的文件导出时. 一定要吧这两个值设置上, 否则会出现内存溢出的错误.

案例分析：在tomcat启动的时候依赖的JVM配置相应的的参数, 然后在Jvisualvm 进行查看：



Jvisualvm 显示的CG图为：



此代码可以测试堆的对象溢出的状态：

```
1.  public class ByteTest {
2.
```

```
    // -Xms32M -Xmx64M -Xmn16M  -XX:PermSize=32m  -XX:MaxPermSize=64m
    // i = 60 i = 70 分别测试
    public static void main(String[] args) throws Exception {
        List<Object> oList=new ArrayList<Object>();
        for(int i=0;i<70;i++){
            byte[] b = new byte[1024*1024];
            oList.add(b);
            System.out.println("i:" + i);
            Thread.sleep(1000);
        }
        System.out.println("----------------------测试完毕-----------------------");
        System.out.println(oList.size());
        Thread.sleep(1000*60*10);
    }
}
```

可以用它的测试Connection

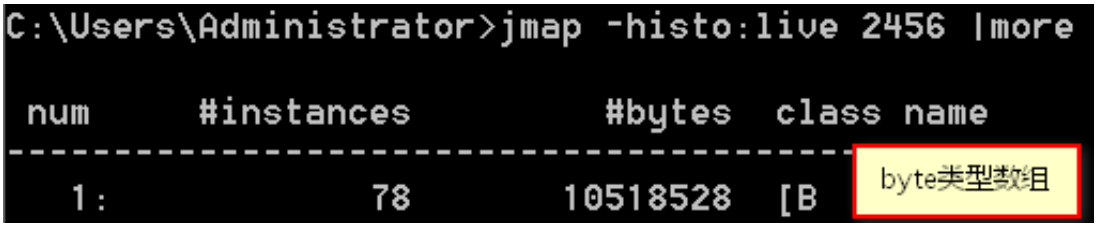
```
public class ByteTest {

    // -Xms32M -Xmx64M -Xmn16M  -XX:PermSize=32m  -XX:MaxPermSize=64m
    // i = 60 i = 70 分别测试
    // -Xms32M -Xmx64M -Xmn16M  -XX:PermSize=32m  -XX:MaxPermSize=64m
    // 测试connection
    public static void main(String[] args) throws Exception {
        List<Object> oList=new ArrayList<Object>();
        for(int i=0;i<70;i++){
            byte[] b = new byte[1024*1024];
            oList.add(b);
            System.out.println("i:" + i);
            Thread.sleep(500);
        }
        System.out.println("----------------------测试完毕-----------------------");
        System.out.println(oList.size());
        Thread.sleep(1000*60*10);
    }

```

如果在测试的时候发现有内存泄露的情况可以采用如下命令，查询对象所占据的空间，例如如下代码，很明显byte是占据的10MB的大小

```
public static void main(String[] args) throws Exception {
        List<Object> oList = new ArrayList<Object>();
        for(int i=0;i<10;i++){
            // 10M空间
            byte[] b=new byte[1*1024*1024];
            oList.add(b);
            System.out.println("i:" + i);
            Thread.sleep(1000);
        }
        System.out.println("----over-----");
```

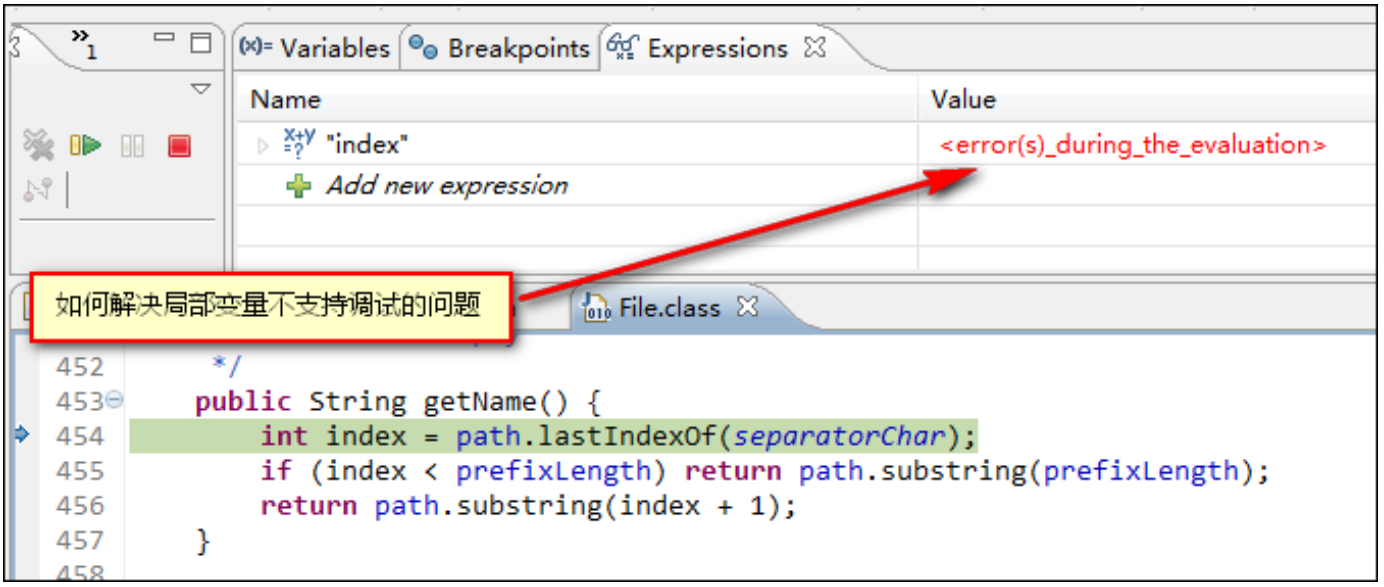
```
11.             Thread.sleep(1000*60*60);
12.         }
```

命令如下，和相关截图如下：10518528 / 1024 / 1024 = 10.03125MB



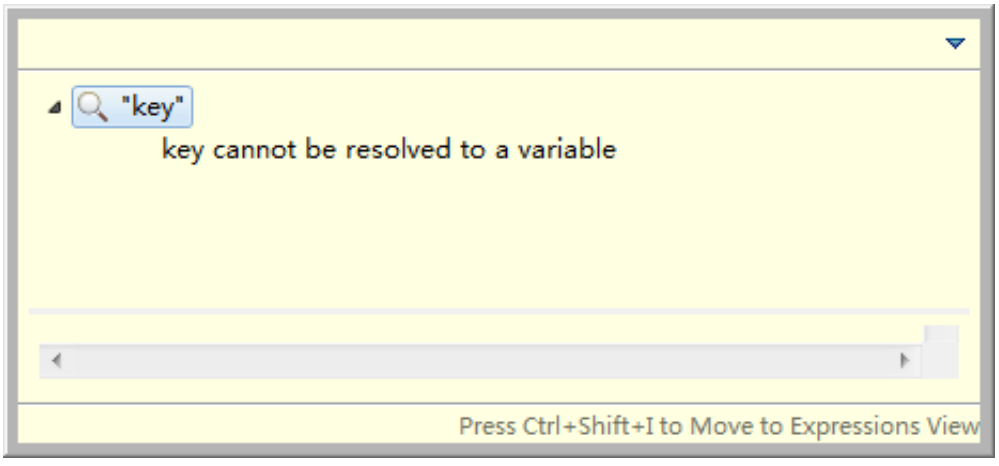
更多的JVM详细参数参考如下：http://www.cnblogs.com/redcreen/archive/2011/05/04/2037057.html

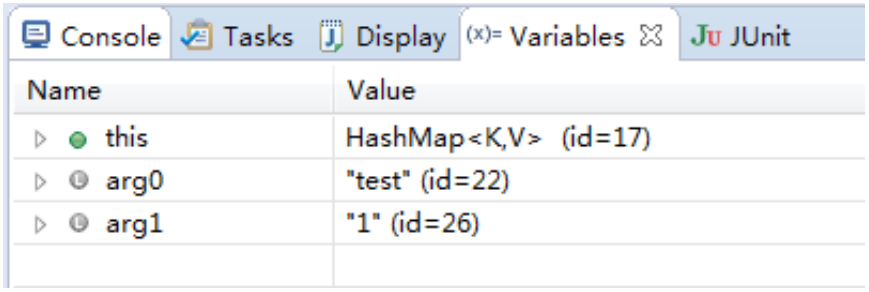
---



解决方案如下：http://my.oschina.net/xionghui/blog/497361

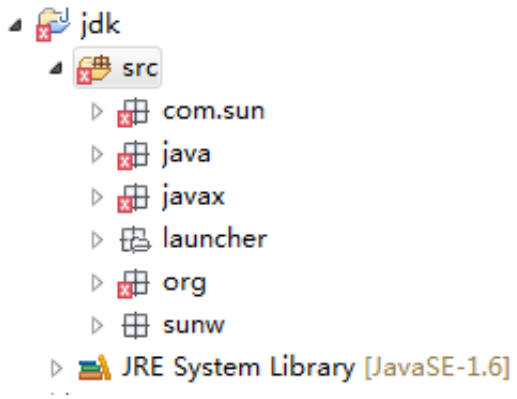
java是一门开源的程序设计语言，喜欢研究源码的java开发者总会忍不住debug一下jdk源码。虽然官方的jdk自带了源码包src.zip，然而在debug时查看变量却十分麻烦。例如调试HashMap的 public V put(K key, V value) 方法并查看key的值时会提示：



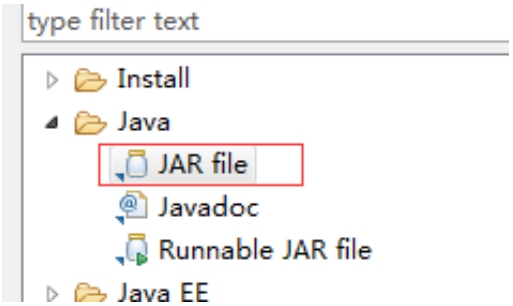


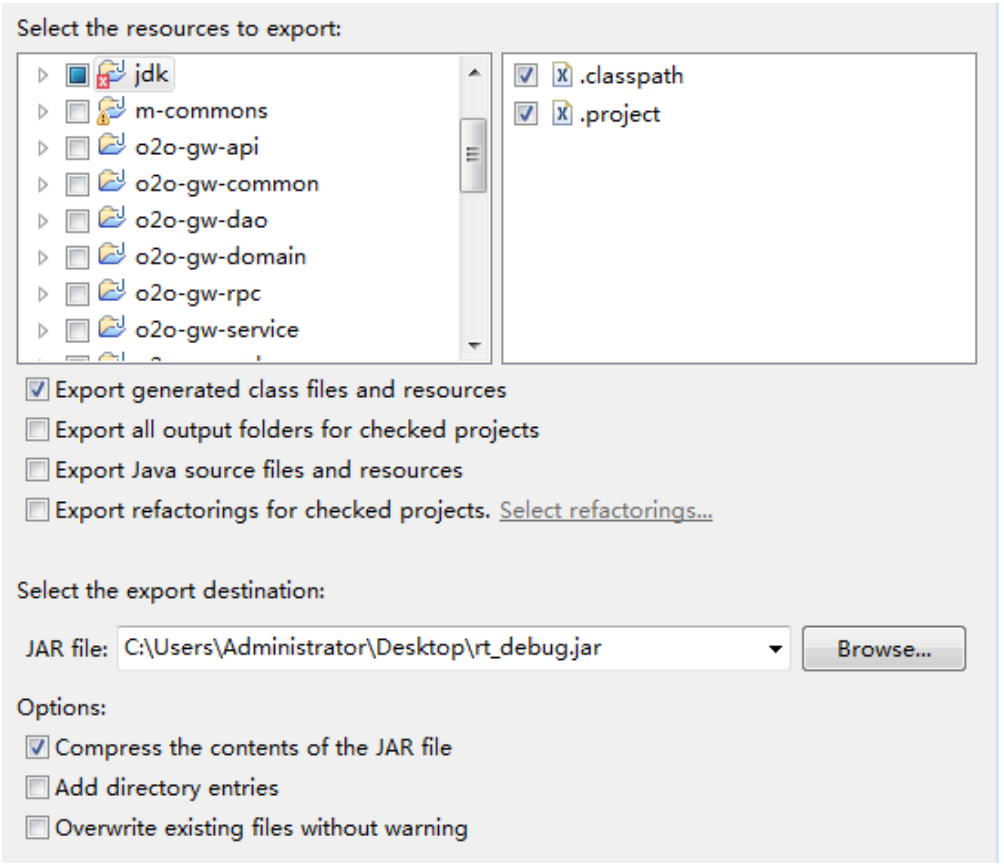
可以看到不能显示变量的值了，原因在于oracle提供的jre中rt.jar不带debug信息：orcale在编译src时使用了 javac -g:none，意思是不带任何调试信息，这样可以减小rt.jar的大小。若想正常调试jdk，就只能重新编译src.zip。这里介绍下编译src.zip的方法。

1. 在eclipse中新建一个java项目“jdk”，然后在src目录上导入"Archive File"，选择源码src.zip导入，导完目录结构如下（不用管编译报错）：

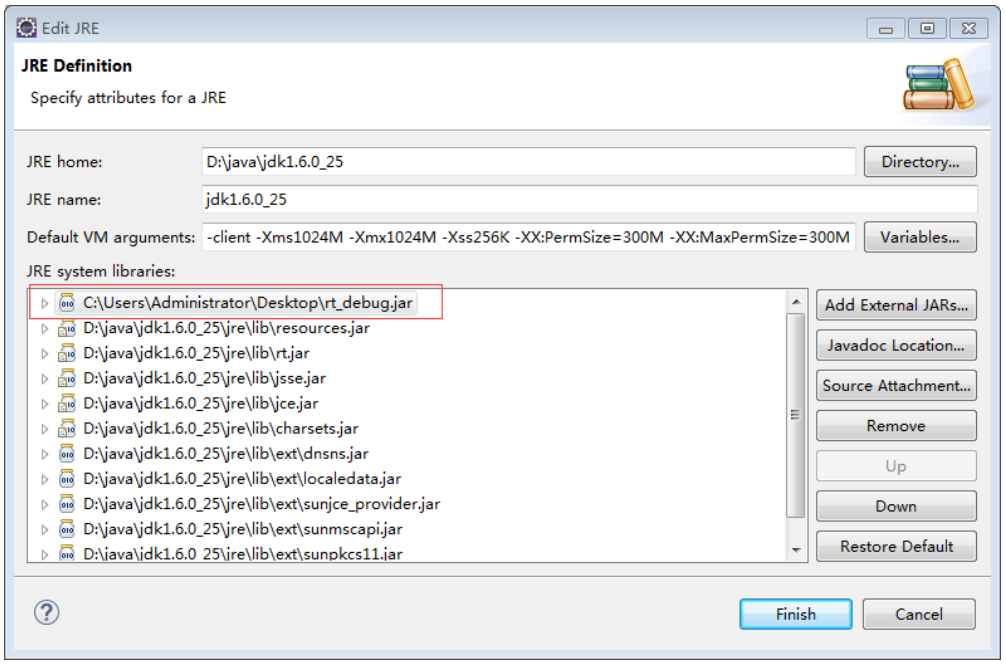


2. 右键项目export...，然后导出为jar包，起名为rt_debug.jar：



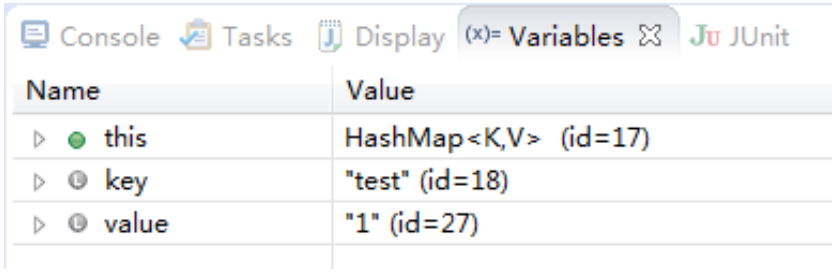


3. 修改eclipse的jre设置，将rt_debug.jar添加到jre中，并移动到最前面：



4. 最后再查看debug变量，可以看到变量值了：


key= "test" (id=18)
count= 4
hash= 0
offset= 0
value= (id=23)
test


Console
Tasks
Display
(x)= Variables
JUnit
Name
Value
this
HashMap<K,V> (id=17)
key
"test" (id=18)
value
"1" (id=27)