http://blog.csdn.net/hexieshangwang/article/details/47251615

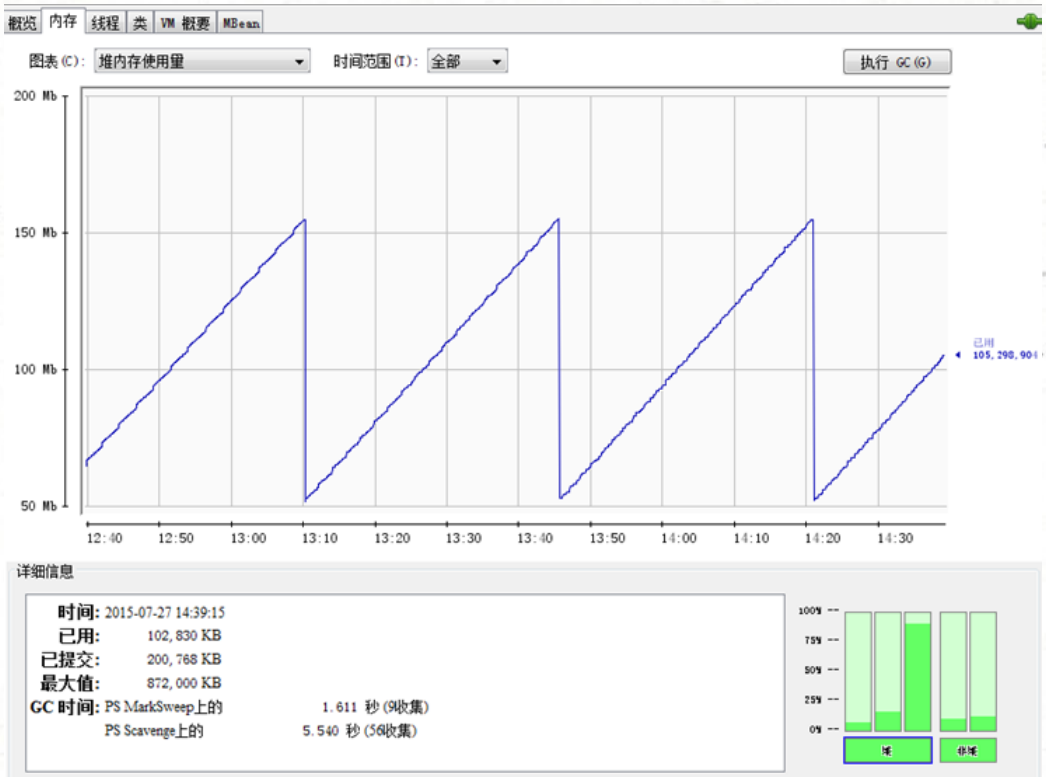
Java垃圾回收机制（GC）

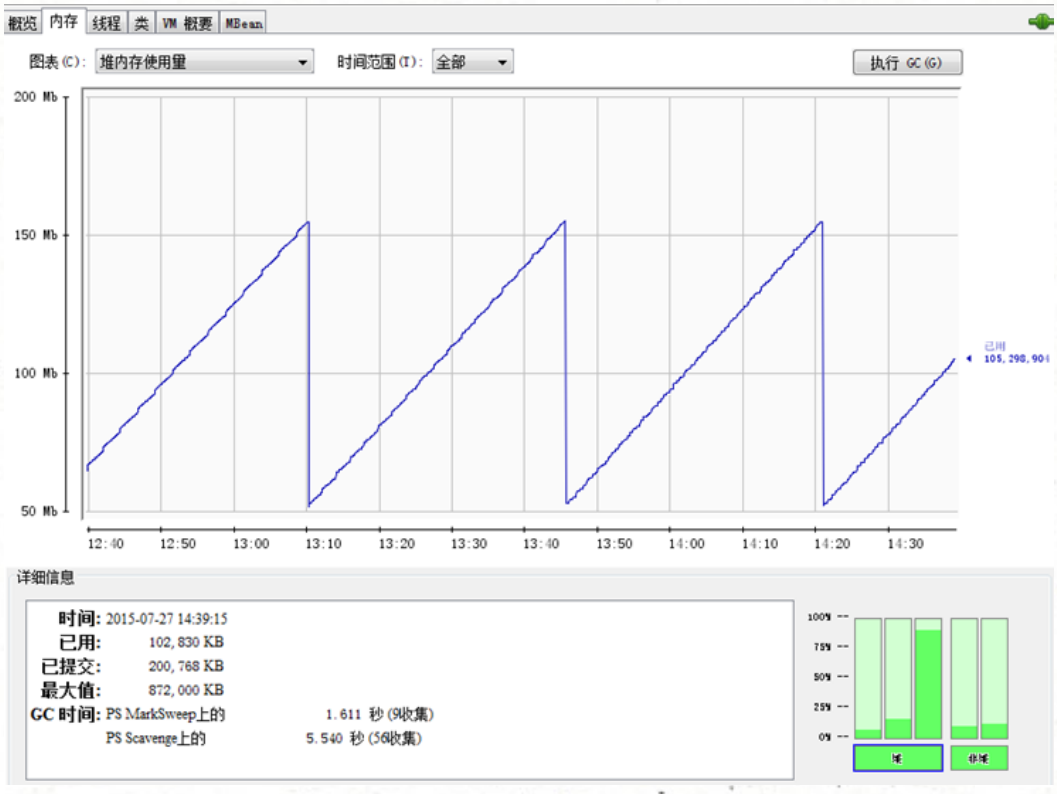
1.GC机制作用

1.1 JVM自动检测和释放不再使用的对象内存

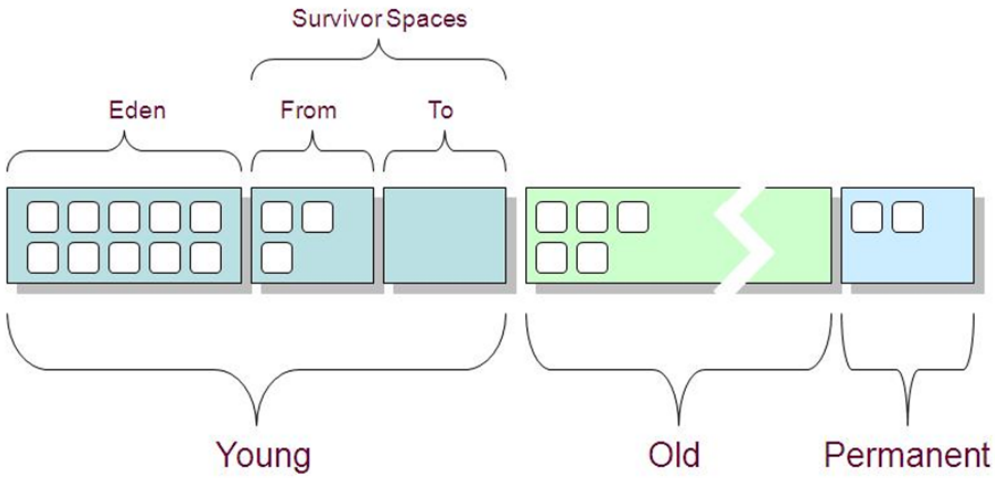
1.2 Java 运行时JVM会执行 GC，不再需要显式释放对象

例：Object.finallize()、 Windows.dispose()、 System.gc()





2.Java堆3代分布



关于Java堆3代分布情况，可通过命令：jmap –heap pid 查看

```
PS Young Generation
Eden Space:
   capacity = 104005632 (99.1875MB)
   used     = 37356664 (35.62609100341797MB)
   free     = 66648968 (63.56140899658203MB)
   35.917924137031358 used
From Space:
   capacity = 9109504 (8.6875MB)
   used     = 163840 (0.15625MB)
   free     = 8945664 (8.53125MB)
   1.7985611510791366% used
To Space:
   capacity = 8454144 (8.0625MB)
   used     = 0 (0.0MB)
   free     = 8454144 (8.0625MB)
   0.0% used
PS Old Generation
   capacity = 105381888 (100.5MB)
   used     = 56282912 (53.675567626953125MB)
   free     = 49098976 (46.824432373046875MB)
   53.40852500194341% used
PS Perm Generation
   capacity = 85262336 (81.3125MB)
   used     = 65188440 (62.168540954589844MB)
   free     = 20073896 (19.143959045410156MB)
   76.45631477889604% used
```

3.GC分类

3.1 Young GC（Minor GC）：收集生命周期短的区域(Young)

(1) 清空Eden+from survivor中所有no ref的对象占用的内存

(2) 将Eden+from survivor中所有存活的对象copy到to survivor中

(3) 一些对象将晋升到old中: to survivor放不下的或存活次数超过turning threshold中的

3.2 Full GC（Major GC）：收集生命周期短的区域(Young)和生命周期比较长的区域(Old)，对整个堆进行垃圾收集，有时也会回收持久区(Perm)

(1) 清空heap中no ref的对象

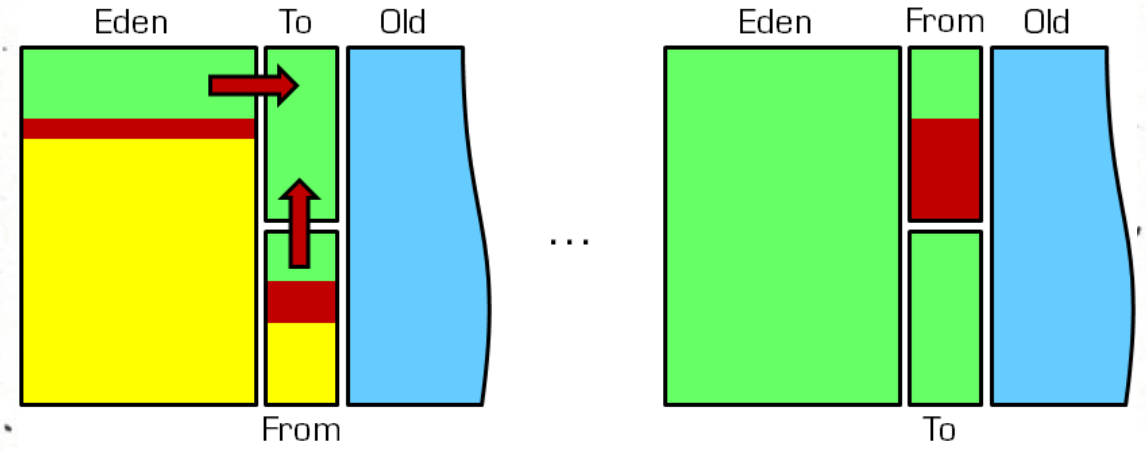
(2) 清空permgen中已经被卸载的class信息

4.GC过程

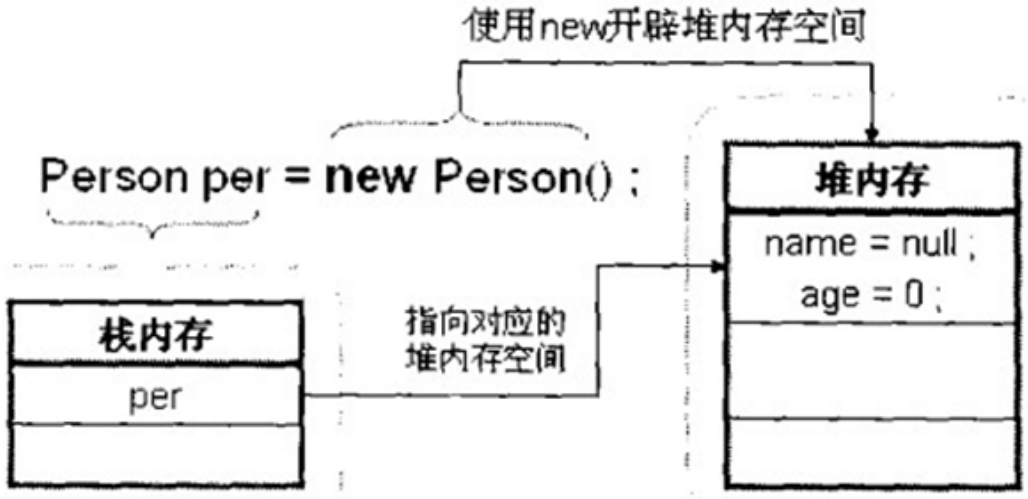
(1) 新生成的对象在Eden区完成内存分配

(2) 当Eden区满，再创建对象，会因为申请不到空间触发YGC，进行young(eden+1survivor)区的垃圾回收（为什么是eden+1survivor：两个survivor中始终有一个survivor是空的，空的那个被标记成To Survivor）

(3) YGC时，Eden不能被回收的对象被放入到空的survivor（也就是放到To Survivor，此时Eden被清空），另一个survivor（From Survivor）里不能被GC回收的对象也会被放入To Survivor，始终保证一个survivor是空的（YGC完成之后，To Survivor 和 From Survivor的标记互换）
(4) YGC结束后，若存放对象的survivor满，则这些对象被copy到old区，或者survivor区没有满，但是有些对象已经足够Old（超过XX:MaxTenuringThreshold），也被放入Old区
(5) 当Old区被放满的之后，进行完整的垃圾回收，即 FGC
(6) FGC后，若Survivor及old区仍然无法存放从Eden复制过来的部分对象，导致JVM无法在Eden区为新对象创建内存区域，则出现OOM错误



Java应用内存问题分析方法
1.Java内存划分
可粗略划分三类：
1.1 堆内存
存放由 new 创建的对象和数组，在堆中分配的内存，由 Java 虚拟机的自动垃圾回收器来管理



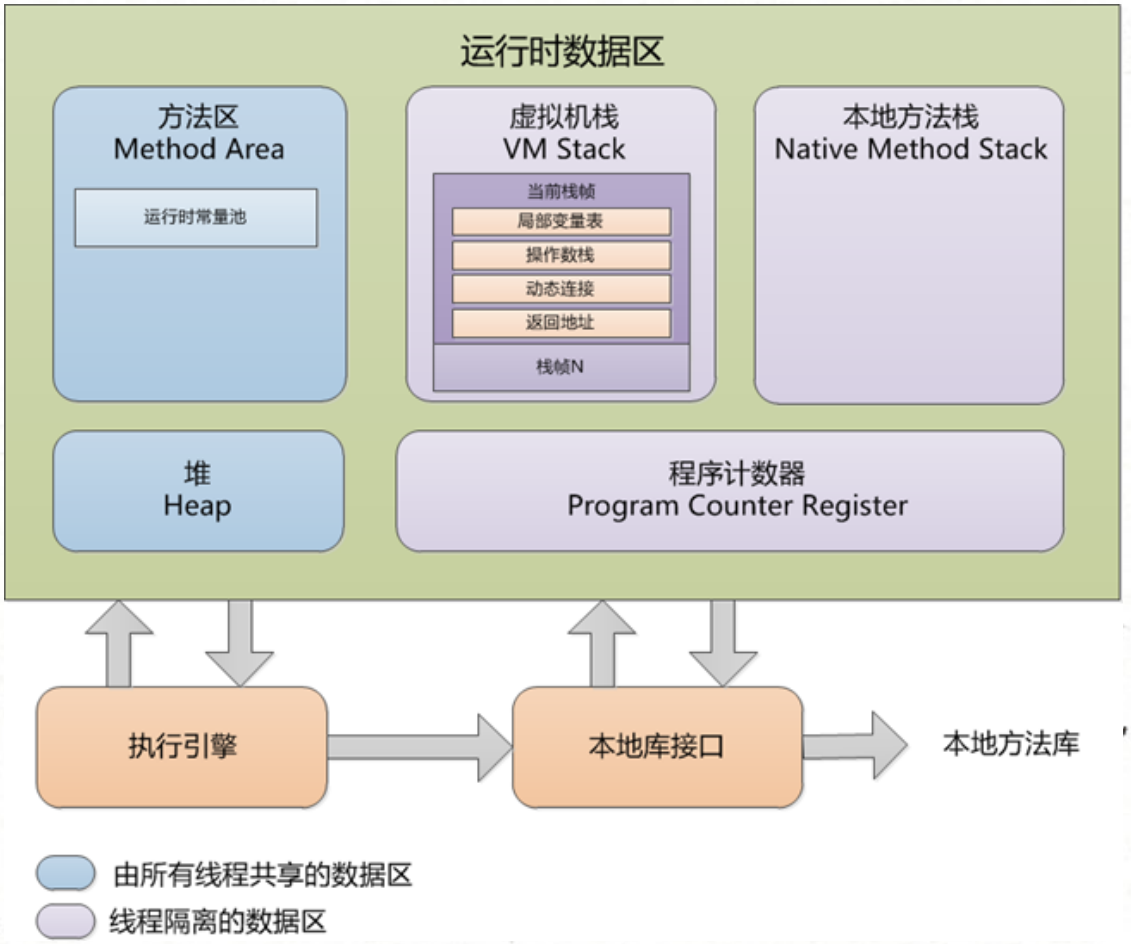
1.2 栈内存

在函数中定义的一些基本类型的变量和对象的引用变量都是在函数的栈内存中分配

（更准确地说是保存了引用的堆内存空间的地址,java中的“指针”）

1.3 永久保存区、方法区（Permanent Generation）

用于存储已被虚拟机加载的类信息、常量、静态变量等



2.Java常见的内存问题表现形式：

2.1 OutOfMemory：内存溢出

2.2 Memory Leak：内存泄露

二者共同点：

(1) 通常最终的状态就会导致OOM错误

(2) 在Java堆或本地内存中都可能发生

二者不同点：

(1) ML是已经分配好的内存或对象，当不再需要，没有得到释放 而OOM则是没有足够的空间来供jvm分配新的内存块

(2) ML的内存曲线总体上是一条斜向上的曲线而OOM不是，反之未必

3.内存溢出类型：

虚拟机栈溢出、本地方法栈溢出、方法区溢出、堆溢出、运行时常量池溢出

异常类型：
(1) java.lang.OutOfMemoryError: **Java** heap space
堆内存溢出
优化：通过-Xmn（最小值）–Xms（初始值） -Xmx（最大值）参数手动设置Heap（堆）的大小。
(2) java.lang.OutOfMemoryError: PermGen space
PermGen Space溢出（方法区溢出、运行时常量池溢出）
优化：通过MaxPermSize参数设置PermGen space大小。
(3) java.lang.StackOverflowError
栈溢出（虚拟机栈溢出、本地方法栈溢出）
优化：通过Xss参数调整

```
// Java 堆溢出
    public static void main(String[] args) {
        List<OOMObject> list = new    ArrayList<JavaHeapSpace.OOMObject>();
        while (true) {
            list.add(new OOMObject());
        }
    }
    static class OOMObject {

    }
```

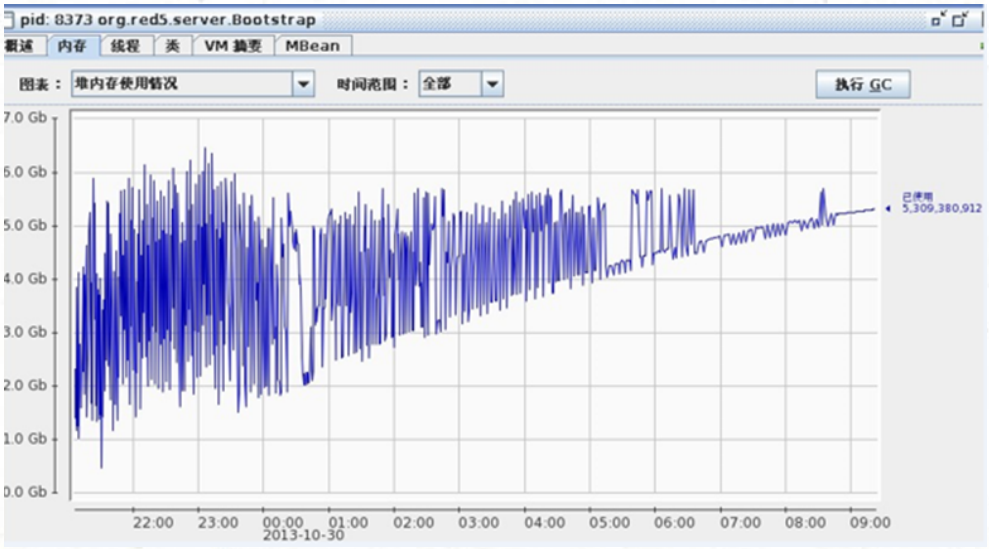
```
// 虚拟机栈溢出
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println(add());
    }

    public static int add(){
        return add();
    }
```

```
// 方法区溢出
    public static void main(String[] args) {
        while (true) {
            Enhancer enhancer = new Enhancer();
            enhancer.setSuperclass(OOMObject.class);
            enhancer.setUseCache(false);
```

```
            enhancer.setCallback(new MethodInterceptor() {
                @Override
                public Object intercept(Object obj, Method method,
                        Object[] args, MethodProxy proxy) throws Throwable {
                    return proxy.invoke(obj, args);
                }
            });
            enhancer.create();
        }
    }

    static class OOMObject {

    }

// 运行时常量池溢出
    public static void main(String[] args){
        // TODO Auto-generated method stub
        List<String> list = new ArrayList<String>();
        int i = 0;
        while (true ){
            list.add(String. valueOf(i++).intern());
        }
    }

// 内存泄露模拟
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        List<int[]> list = new ArrayList<int[]>();

        Runtime run = Runtime.getRuntime();

        int i=1;

        while(true){
            int[] arr = new int[1024];
            list.add(arr);

            if(i++ % 1000 == 0 ){
```

```
            System.out.print("最大堆内存=" + run.maxMemory() / 1024 / 1024 + "M, ");
            System.out.print("已分配内存=" + run.totalMemory() /1024 / 1024 + "M, ");
            System.out.print("剩余空间内存=" + run.freeMemory() / 1024 / 1024 + "M, ");
            System.out.println("最大可用内存=" + ( run.maxMemory() - run.totalMemory() +
run.freeMemory() ) / 1024 / 1024 + "M");
            sleep(1000);
        }
    }
}

public static void sleep(long time) {
    try {
        Thread.sleep(time);
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

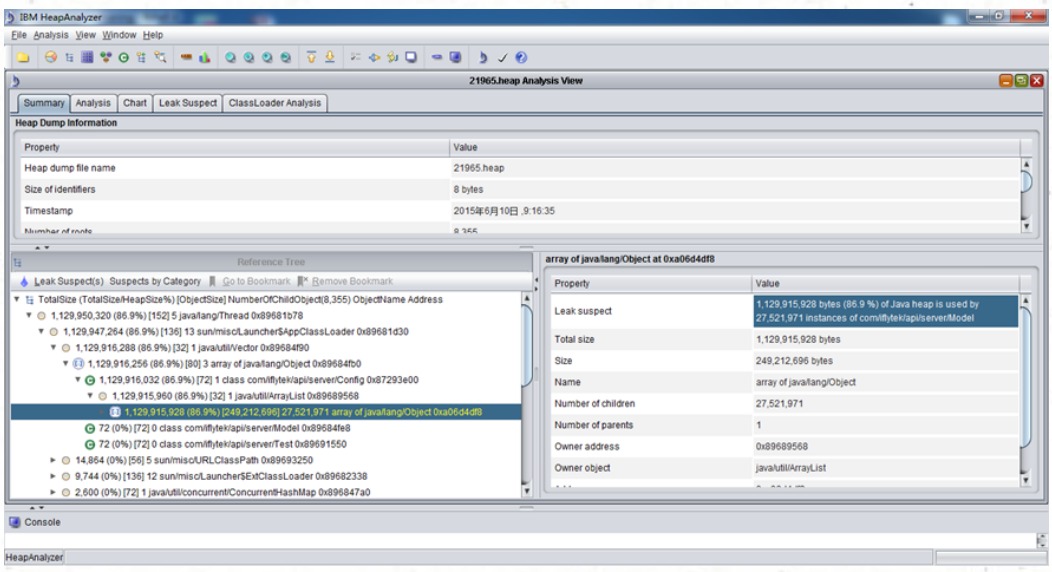
4.内存泄露现象



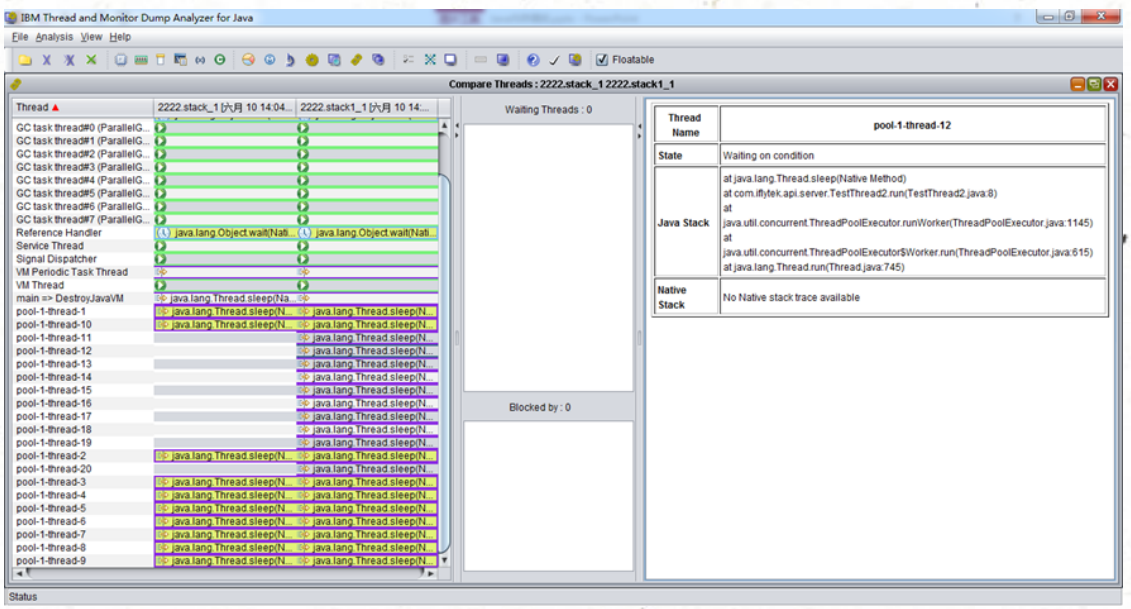
heapspace:OutOfMemoryError

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at java.util.Arrays.copyOf(Arrays.java:2245)
        at java.util.Arrays.copyOf(Arrays.java:2219)
        at java.util.ArrayList.grow(ArrayList.java:242)
        at java.util.ArrayList.ensureExplicitCapacity(ArrayList.java:216)
        at java.util.ArrayList.ensureCapacityInternal(ArrayList.java:208)
        at java.util.ArrayList.add(ArrayList.java:440)
        at com.iflytek.api.server.Config.setConfig(Config.java:19)
        at com.iflytek.api.server.Test.main(Test.java:27)
```

开发人员的分析、解决思路
内存对象申请未释放（未及时释放）
线程问题
分别从堆dump和线程dump进行分析：
jmap -dump:format=b,file=heap.dump pid
jstack pid >> thread.dump
5.JAVA DUMP分析工具
IBM HeapAnalyzer:ha456.jar
IBM Thread and Monitor Dump Analyzer:jca457.jar
堆dump分析
占用内存较多代码块
分析代码快上下文
分析占用内存的对象内容



线程dump分析
活跃线程
阻塞线程
等待资源线程



Java应用CPU问题分析方法

1.程序响应慢，CPU高

(1) ThreadDump

jstack pid >> thread.dump

(2) 找到导致cpu高的线程 top -H -p pid

(3) pid 十进制转十六进制

http://tool.oschina.net/hexconvert/

(4) 找到对应的线程UE打开 threaddump文件查找：按十六进制关键字找到对应的线程，把相关的方法找出来，可以精确到代码的行号

2.程序响应慢，CPU不高

一般表现为thread struck在了i/o、db等

实例：

IO阻塞（程序表现为响应慢）

线程状态为“in Object.wait()”，说明正在等待线程池可用资源，由于线程池满导致新的IO请求处于排队等待状态，且发生在：at com.iflytek.diange.data.provider.sendsong.impl.SendSongImpl.getSendSongInfosByUserId(SendSongImpl.java:92)行

```
erverHandler-192.168.57.144:20882-thread-200" daemon prio=10 tid=0x00007fd190281800 nid=0x5af6 in Object.wait()
.lang.Thread.State: BLOCKED (on object monitor)
 at java.lang.Object.wait(Native Method)
 at com.mchange.v2.resourcepool.BasicResourcePool.awaitAvailable(BasicResourcePool.java:1315)
 at com.mchange.v2.resourcepool.BasicResourcePool.prelimCheckoutResource(BasicResourcePool.java:557)
 - locked <0x00000000e77835a0> (a com.mchange.v2.resourcepool.BasicResourcePool)
 at com.mchange.v2.resourcepool.BasicResourcePool.checkoutResource(BasicResourcePool.java:477)
 at com.mchange.v2.c3p0.impl.C3P0PooledConnectionPool.checkoutPooledConnection(C3P0PooledConnectionPool.java:525
 at com.mchange.v2.c3p0.impl.AbstractPoolBackedDataSource.getConnection(AbstractPoolBackedDataSource.java:128)
 at org.springframework.jdbc.datasource.DataSourceUtils.doGetConnection(DataSourceUtils.java:111)
 at org.springframework.jdbc.datasource.DataSourceUtils.getConnection(DataSourceUtils.java:77)
```

3.程序无响应

死锁（程序表现为无响应）

线程状态为“waiting to lock”：两个线程各持有一个锁，又在等待另一个锁，故造成死锁，且发生在DeadLockTest.java:39行

```
"DestroyJavaVM" prio=6 tid=0x019f1400 nid=0x19d8 waiting on condition [0x0000000
0..0x002afd20]
   java.lang.Thread.State: RUNNABLE

"Thread-1" prio=6 tid=0x01a80000 nid=0x1b04 waiting for monitor entry [0x03e4f00
0..0x03e4fae8]
   java.lang.Thread.State: BLOCKED (on object monitor)
        at T1.run(DeadLockTest.java:39)
        - waiting to lock <0x276ea028> (a java.lang.String)
        - locked <0x2c4c4f28> (a java.lang.String)
        at java.lang.Thread.run(Unknown Source)

"Thread-0" prio=6 tid=0x01a7f800 nid=0x1fbc waiting for monitor entry [0x03dff00
0..0x03dffb68]
   java.lang.Thread.State: BLOCKED (on object monitor)
        at T1.run(DeadLockTest.java:39)
        - waiting to lock <0x2c4c4f28> (a java.lang.String)
        - locked <0x276ea028> (a java.lang.String)
        at java.lang.Thread.run(Unknown Source)

"Low Memory Detector" daemon prio=6 tid=0x01a6a000 nid=0x1024 runnable [0x000000
00..0x00000000]
   java.lang.Thread.State: RUNNABLE

"CompilerThread0" daemon prio=10 tid=0x01a5fc00 nid=0x16ac waiting on condition
[0x00000000..0x03d0f8f0]
   java.lang.Thread.State: RUNNABLE

"Attach Listener" daemon prio=10 tid=0x01a5ec00 nid=0x3b0 runnable [0x00000000..
```