day04-JVM&垃圾回收机制

1 jvm

1.1 我们为什么要对jvm做优化?

在本地开发环境中我们很少会遇到需要对jvm进行优化的需求，但是到了生产环境，我们可能将有下面的需求:

- 运行的应用“卡住了”，日志不输出，程序没有反应
- 服务器的CPU负载突然升高
- 在多线程应用下，如何分配线程的数量?
- ……

在本次课程中，我们将对jvm有更深入的学习，我们不仅要让程序能跑起来，而且是可以跑的更快！可以分析解决在生产环境中所遇到的各种“棘手”的问题。

> 说明：本套课程使用的jdk版本为1.8。

1.2 jvm的运行参数

在jvm中有很多的参数可以进行设置，这样可以让jvm在各种环境中都能够高效的运行。绝大部分的参数保持默认即可。

1.2.1 三种参数类型

jvm的参数类型分为三类，分别是:

- 标准参数(稳定的,在后面版本都会保留)
  - -help
  - -version
- -X参数 （非标准参数）
  - 了解
- -XX参数（使用率较高）
  - -XX:newSize
  - -XX:+UseSerialGC

1.2.2 标准参数

jvm的标准参数，一般都是很稳定的，在未来的JVM版本中不会改变，可以使用**java -help**检索出所有的标准参数。

```
[root@node01 ~]# java -help
用法: java [-options] class [args...]
           (执行类)
   或  java [-options] -jar jarfile [args...]
           (执行 jar 文件)
其中选项包括:
    -d32       使用 32 位数据模型 (如果可用)
    -d64       使用 64 位数据模型 (如果可用)
```

```
    -server    选择 "server" VM
                  默认 VM 是 server,
                  因为您是在服务器类计算机上运行。


    -cp <目录和 zip/jar 文件的类搜索路径>
    -classpath <目录和 zip/jar 文件的类搜索路径>
                  用 : 分隔的目录, JAR 档案
                  和 ZIP 档案列表, 用于搜索类文件。
    -D<名称>=<值>
                  设置系统属性
    -verbose:[class|gc|jni]
                  启用详细输出
    -version      输出产品版本并退出
    -version:<值>
                  警告: 此功能已过时, 将在
                  未来发行版中删除。
                  需要指定的版本才能运行
    -showversion  输出产品版本并继续
    -jre-restrict-search | -no-jre-restrict-search
                  警告: 此功能已过时, 将在
                  未来发行版中删除。
                  在版本搜索中包括/排除用户专用 JRE
    -? -help      输出此帮助消息
    -X            输出非标准选项的帮助
    -ea[:<packagename>...|:<classname>]
    -enableassertions[:<packagename>...|:<classname>]
                  按指定的粒度启用断言
    -da[:<packagename>...|:<classname>]
    -disableassertions[:<packagename>...|:<classname>]
                  禁用具有指定粒度的断言
    -esa | -enablesystemassertions
                  启用系统断言
    -dsa | -disablesystemassertions
                  禁用系统断言
    -agentlib:<libname>[=<选项>]
                  加载本机代理库 <libname>, 例如 -agentlib:hprof
                  另请参阅 -agentlib:jdwp=help 和 -agentlib:hprof=help
    -agentpath:<pathname>[=<选项>]
                  按完整路径名加载本机代理库
    -javaagent:<jarpath>[=<选项>]
                  加载 Java 编程语言代理, 请参阅 java.lang.instrument
    -splash:<imagepath>
                  使用指定的图像显示启动屏幕
```

1.2.2.1 实战

> **实战1：查看jvm版本**

```
[root@node01 ~]# java -version
java version "1.8.0_141"
Java(TM) SE Runtime Environment (build 1.8.0_141-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.141-b15, mixed mode)

# -showversion参数是表示，先打印版本信息，再执行后面的命令，在调试时非常有用，后面会使用到。
```

> 实战2：通过-D设置系统属性参数

```
public class TestJVM {

    public static void main(String[] args) {
        String str = System.getProperty("str");
        if (str == null) {
            System.out.println("itcast");
        } else {
            System.out.println(str);
        }
    }
}
```

进行编译、测试：

```
#编译
[root@node01 test]# javac TestJVM.java

#测试
[root@node01 test]# java TestJVM
itcast
[root@node01 test]# java -Dstr=123 TestJVM
123
```

1.2.2.2 -server与-client参数

可以通过-server或-client设置jvm的运行参数。

- 它们的区别是Server VM的初始堆空间会大一些，默认使用的是**并行垃圾回收器**，启动慢运行快。
- Client VM相对来讲会保守一些，初始堆空间会小一些，使用**串行的垃圾回收器**，它的目标是为了让JVM的启动速度更快，但运行速度会比Serverm模式慢些。
- JVM在启动的时候会根据硬件和操作系统自动选择使用Server还是Client类型的JVM。
- 32位操作系统
  - 如果是Windows系统，不论硬件配置如何，都默认使用Client类型的JVM。
  - 如果是其他操作系统上，机器配置有2GB以上的内存同时有2个以上CPU的话默认使用server模式，否则使用client模式。
- 64位操作系统
  - 只有server类型，不支持client类型。

1.2.3 -X参数

jvm的-X参数是非标准参数，在不同版本的jvm中，参数可能会有所不同，可以通过**java -X**查看非标准参数。

```
[root@node01 test]# java -X
    -Xmixed           混合模式执行 (默认)
    -Xint             仅解释模式执行
    -Xbootclasspath:<用 : 分隔的目录和 zip/jar 文件>
                      设置搜索路径以引导类和资源
    -Xbootclasspath/a:<用 : 分隔的目录和 zip/jar 文件>
                      附加在引导类路径末尾
    -Xbootclasspath/p:<用 : 分隔的目录和 zip/jar 文件>
```

```
                          置于引导类路径之前
    -Xdiag                显示附加诊断消息
    -Xnoclassgc           禁用类垃圾收集
    -Xincgc               启用增量垃圾收集
    -Xloggc:<file>        将 GC 状态记录在文件中 (带时间戳)
    -Xbatch               禁用后台编译
    -Xms<size>            设置初始 Java 堆大小
    -Xmx<size>            设置最大 Java 堆大小
    -Xss<size>            设置 Java 线程堆栈大小
    -Xprof                输出 cpu 配置文件数据
    -Xfuture              启用最严格的检查, 预期将来的默认值
    -Xrs                  减少 Java/VM 对操作系统信号的使用 (请参阅文档)
    -Xcheck:jni           对 JNI 函数执行其他检查
    -Xshare:off           不尝试使用共享类数据
    -Xshare:auto          在可能的情况下使用共享类数据 (默认)
    -Xshare:on            要求使用共享类数据, 否则将失败。
    -XshowSettings        显示所有设置并继续
    -XshowSettings:all
                          显示所有设置并继续
    -XshowSettings:vm     显示所有与 vm 相关的设置并继续
    -XshowSettings:properties
                          显示所有属性设置并继续
    -XshowSettings:locale
                          显示所有与区域设置相关的设置并继续

-X 选项是非标准选项, 如有更改, 恕不另行通知。
```

1.2.3.1 -Xms与-Xmx参数

-Xms与-Xmx分别是设置jvm的堆内存的**初始大小**和**最大大小**。

-Xms512m：等价于-XX:InitialHeapSize，设置JVM初始堆内存为512M。

-Xmx2048m：等价于-XX:MaxHeapSize，设置JVM最大堆内存为2048M。

适当的调整jvm的内存大小，可以充分利用服务器资源，让程序跑的更快。

示例：

```
[root@node01 test]# java -Xms512m -Xmx2048m TestJVM
itcast
```

1.2.4 -XX参数

-XX参数也是非标准参数，主要用于jvm的调优和debug操作。

-XX参数的使用有2种方式，一种是boolean类型，一种是非boolean类型：

- boolean类型
  - 格式：-XX:[+-]<name> 表示启用或禁用<name>属性
  - 如：-XX:+DisableExplicitGC 表示禁用手动调用gc操作，也就是说调用System.gc()无效
- 非boolean类型
  - 格式：-XX:<name>=<value> 表示<name>属性的值为<value>
  - 如：-XX:NewRatio=1 表示新生代和老年代的比值

用法：

```
[root@node01 test]# java -showversion -XX:+DisableExplicitGC TestJVM
java version "1.8.0_141"
Java(TM) SE Runtime Environment (build 1.8.0_141-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.141-b15, mixed mode)

itcast
```

1.2.4.1 查看jvm的运行参数

有些时候我们需要查看jvm的运行参数，这个需求可能会存在2种情况：

第一，运行java命令时打印出运行参数；

第二，查看正在运行的java进程的参数；

1.2.4.1.1 运行java命令时打印参数

运行java命令时打印参数，需要添加-XX:+PrintFlagsFinal参数即可。

```
[root@node01 test]# java -XX:+PrintFlagsFinal -version
[Global flags]
    uintx AdaptiveSizeDecrementScaleFactor          = 4
          {product}
    uintx AdaptiveSizeMajorGCDecayTimeScale         = 10
         {product}
    uintx AdaptiveSizePausePolicy                   = 0
          {product}
    uintx AdaptiveSizePolicyCollectionCostMargin    = 50
         {product}
    uintx AdaptiveSizePolicyInitializingSteps       = 20
         {product}
    uintx AdaptiveSizePolicyOutputInterval          = 0
          {product}
    uintx AdaptiveSizePolicyWeight                  = 10
         {product}
    uintx AdaptiveSizeThroughPutPolicy              = 0
          {product}
    uintx AdaptiveTimeWeight                        = 25
         {product}
     bool AdjustConcurrency                         = false
          {product}
     bool AggressiveOpts                            = false
          {product}
     intx AliasLevel                                = 3
          {C2 product}
     bool AlignVector                               = true
         {C2 product}
     intx AllocateInstancePrefetchLines             = 1
          {product}
     intx AllocatePrefetchDistance                  = 256
          {product}
     intx AllocatePrefetchInstr                     = 0
          {product}

     ..........略..........
```

```
     bool UseXmmI2D                                = false
          {ARCH product}
     bool UseXmmI2F                                = false
          {ARCH product}
     bool UseXmmLoadAndClearUpper                  = true
         {ARCH product}
     bool UseXmmRegToRegMoveAll                    = true
         {ARCH product}
     bool VMThreadHintNoPreempt                    = false
          {product}
     intx VMThreadPriority                         = -1
         {product}
     intx VMThreadStackSize                        = 1024
         {pd product}
     intx ValueMapInitialSize                      = 11
         {C1 product}
     intx ValueMapMaxLoopSize                      = 8
          {C1 product}
     intx ValueSearchLimit                         = 1000
         {C2 product}
     bool VerifyMergedCPBytecodes                  = true
         {product}
     bool VerifySharedSpaces                       = false
          {product}
     intx WorkAroundNPTLTimedWaitHang              = 1
          {product}
    uintx YoungGenerationSizeIncrement             = 20
          {product}
    uintx YoungGenerationSizeSupplement            = 80
          {product}
    uintx YoungGenerationSizeSupplementDecay       = 8
          {product}
    uintx YoungPLABSize                            = 4096
          {product}
     bool ZeroTLAB                                 = false
          {product}
     intx hashCode                                 = 5
          {product}
java version "1.8.0_141"
Java(TM) SE Runtime Environment (build 1.8.0_141-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.141-b15, mixed mode)
```

由上述的信息可以看出，参数有boolean类型和数字类型，值的操作符是=或:=，分别代表默认值和被修改的值。

示例：

```
java -XX:+PrintFlagsFinal -XX:+VerifySharedSpaces -version

     intx ValueMapInitialSize                      = 11
         {C1 product}
     intx ValueMapMaxLoopSize                      = 8
          {C1 product}
     intx ValueSearchLimit                         = 1000
         {C2 product}
     bool VerifyMergedCPBytecodes                  = true
         {product}
```

```
     bool VerifySharedSpaces                         := true
         {product}
     intx WorkAroundNPTLTimedWaitHang                 = 1
          {product}
    uintx YoungGenerationSizeIncrement                = 20
         {product}
    uintx YoungGenerationSizeSupplement               = 80
         {product}
    uintx YoungGenerationSizeSupplementDecay          = 8
          {product}
    uintx YoungPLABSize                               = 4096
         {product}
     bool ZeroTLAB                                    = false
          {product}
     intx hashCode                                    = 5
          {product}
java version "1.8.0_141"
Java(TM) SE Runtime Environment (build 1.8.0_141-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.141-b15, mixed mode)

#可以看到VerifySharedSpaces这个参数已经被修改了。
```

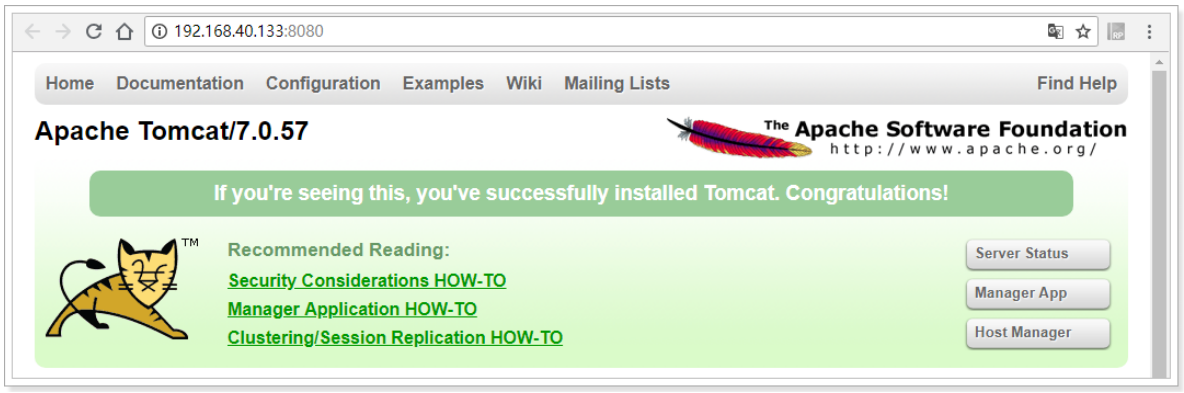
1.2.4.1.2 查看正在运行的jvm参数

如果想要查看正在运行的jvm就需要借助于jinfo命令查看。

首先，启动一个tomcat用于测试，来观察下运行的jvm参数。

```
cd /tmp/
rz 上传
tar -xvf apache-tomcat-7.0.103.tar.gz
cd apache-tomcat-7.0.103
cd bin/
./startup.sh

#http://192.168.40.133:8080/ 进行访问
```

访问成功：



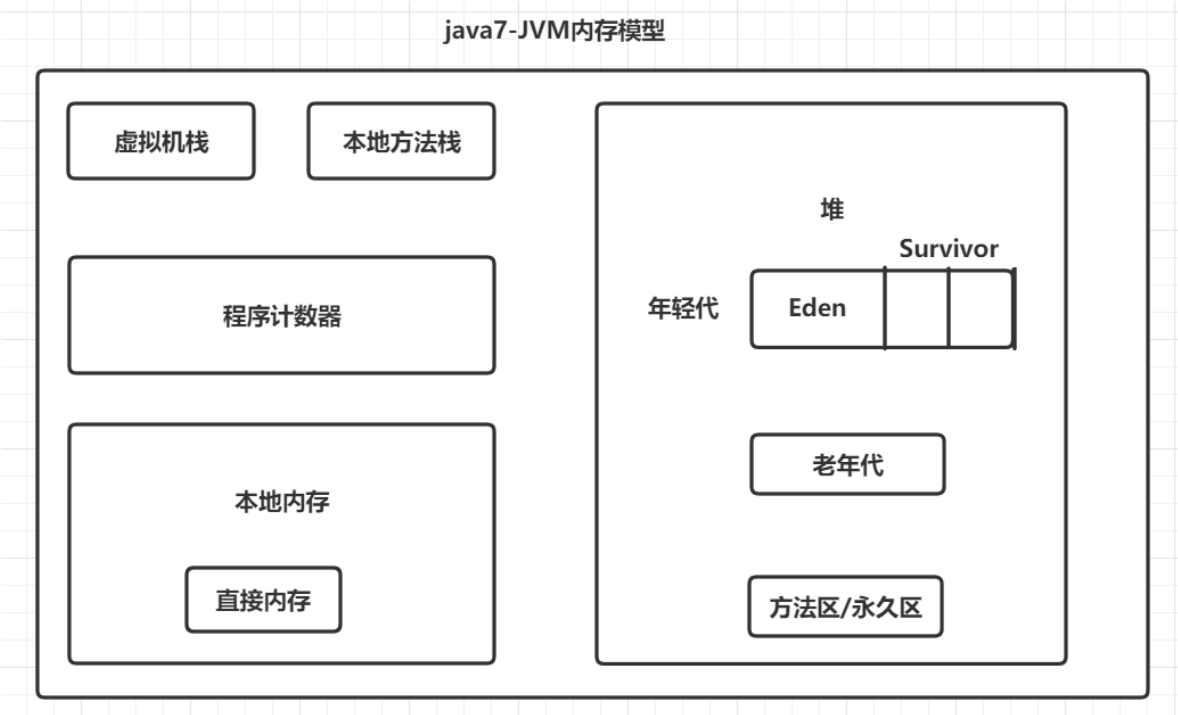

```
#查看所有的参数，用法：jinfo -flags <进程id>

#通过jps 或者  jps -l 查看java进程
[root@node01 bin]# jps
6346 Jps
6219 Bootstrap
[root@node01 bin]# jps -l
```

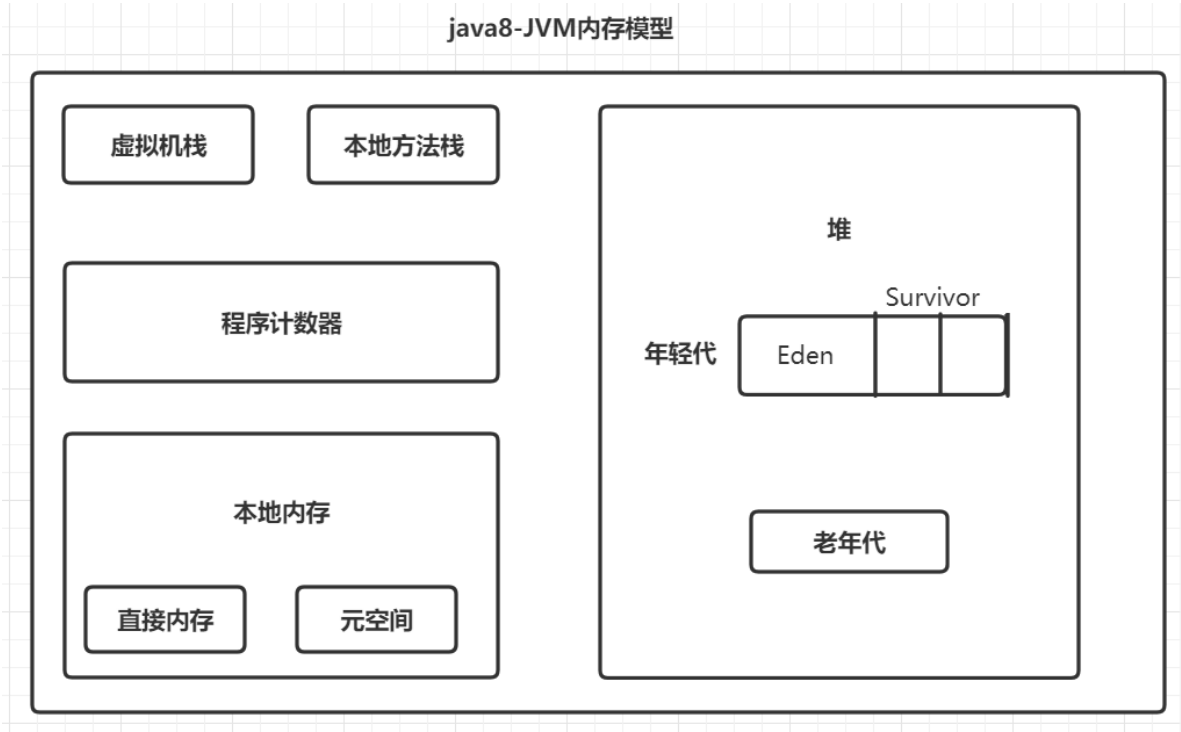
```
6358 sun.tools.jps.Jps
6219 org.apache.catalina.startup.Bootstrap
[root@node01 bin]#

[root@node01 bin]# jinfo -flags 6219
Attaching to process ID 6219, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.141-b15
Non-default VM flags: -XX:CICompilerCount=2 -XX:InitialHeapSize=31457280 -
XX:MaxHeapSize=488636416 -XX:MaxNewSize=162529280 -XX:MinHeapDeltaBytes=524288 -
XX:NewSize=10485760 -XX:OldSize=20971520 -XX:+UseCompressedClassPointers -
XX:+UseCompressedOops -XX:+UseFastUnorderedTimeStamps -XX:+UseParallelGC
Command line:  -Djava.util.logging.config.file=/tmp/apache-tomcat-
7.0.57/conf/logging.properties -
Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager -
Djava.endorsed.dirs=/tmp/apache-tomcat-7.0.57/endorsed -
Dcatalina.base=/tmp/apache-tomcat-7.0.57 -Dcatalina.home=/tmp/apache-tomcat-
7.0.57 -Djava.io.tmpdir=/tmp/apache-tomcat-7.0.57/temp

#查看某一参数的值，用法：jinfo -flag <参数名> <进程id>
[root@node01 bin]# jinfo -flag MaxHeapSize 6219
-XX:MaxHeapSize=488636416
```

1.3 jvm的内存模型

1.3.1 核心概念

JVM内存模型是JVM中非常重要的一部分，并且在JDK7和JDK8中也进行了一些改动。





虚拟机栈：

线程私有的，生命周期与线程相同的，保存执行方法时的**局部变量、动态连接信息、方法返回地址信息**等等。方法开始执行的时候会进栈，方法执行完会出栈【相当于清空了数据】。不需要进行GC。

本地方法栈：

与虚拟机栈类似。本地方法栈是为虚拟机**执行本地方法时提供服务的**。不需要进行GC。

程序计数器：

线程私有的。内部保存的字节码的行号。

```
C:\Users\weizhaohui>javap -verbose D:\workspace\restKeeper1\Restkeeper\restkeeper_service\restkeeper_order\target\classes\com\restkeeper\order\service\OrderDetailServiceImpl.class
Classfile /D:/workspace/restKeeper1/Restkeeper/restkeeper_service/restkeeper_order/target/classes/com/restkeeper/order/service/OrderDetailServiceImpl.class
  Last modified 2020-2-6; size 878 bytes
  MD5 checksum 89c5003553323c537f65d10056d4ea68
  Compiled from "OrderDetailServiceImpl.java"
public class com.restkeeper.order.service.OrderDetailServiceImpl extends com.baomidou.mybatisplus.extension.service.impl.ServiceImpl<com.restkeeper.order.mapper.OrderDetailMapper, com.restkeeper.entity.OrderDetailEntity>
restkeeper.service.IOrderDetailService
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #3.#25         // com/baomidou/mybatisplus/extension/service/impl/ServiceImpl."<init>":()V
   #2 = Class              #26            // com/restkeeper/order/service/OrderDetailServiceImpl
   #3 = Class              #27            // com/baomidou/mybatisplus/extension/service/impl/ServiceImpl
   #4 = Class              #28            // com/restkeeper/service/IOrderDetailService
   #5 = Utf8               <init>
   #6 = Utf8               ()V
   #7 = Utf8               Code
   #8 = Utf8               LineNumberTable
   #9 = Utf8               LocalVariableTable
  #10 = Utf8               this
  #11 = Utf8               Lcom/restkeeper/order/service/OrderDetailServiceImpl;
  #12 = Utf8               Signature
  #13 = Utf8               Lcom/baomidou/mybatisplus/extension/service/impl/ServiceImpl<Lcom/restkeeper/order/mapper/OrderDetailMapper;Lcom/restkeeper/entity/OrderDetailEntity;>;Lcom/restkeeper/service/IOrderDetailServic
  #14 = Utf8               SourceFile
  #15 = Utf8               OrderDetailServiceImpl.java
  #16 = Utf8               RuntimeVisibleAnnotations
  #17 = Utf8               Lorg/springframework/stereotype/Service;
  #18 = Utf8               value
  #19 = Utf8               orderDetailService
  #20 = Utf8               Lorg/apache/dubbo/config/annotation/Service;
  #21 = Utf8               version
  #22 = Utf8               1.0.0
  #23 = Utf8               protocol
  #24 = Utf8               dubbo
  #25 = NameAndType        #5:#6          // "<init>":()V
  #26 = Utf8               com/restkeeper/order/service/OrderDetailServiceImpl
  #27 = Utf8               com/baomidou/mybatisplus/extension/service/impl/ServiceImpl
  #28 = Utf8               com/restkeeper/service/IOrderDetailService

{
  public com.restkeeper.order.service.OrderDetailServiceImpl();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method com/baomidou/mybatisplus/extension/service/impl/ServiceImpl."<init>":()V
         4: return
      LineNumberTable:
        line 11: 0
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       5     0  this   Lcom/restkeeper/order/service/OrderDetailServiceImpl;
```

java虚拟机对于多线程是通过线程轮流切换并且分配线程执行时间。在任何的一个时间点上，一个处理器只会处理执行一个线程，如果当前被执行的这个线程它所分配的执行时间用完了【挂起】。处理器会切换到另外的一个线程上来进行执行。并且这个线程的执行时间用完了，接着处理器就会又来执行被挂起的这个线程。

那么现在有一个问题就是，当前处理器如何能够知道，对于这个被挂起的线程，它上一次执行到了哪里？那么这时就需要从程序计数器中来回去到当前的这个线程他上一次执行的行号，然后接着继续向下执行。

程序计数器是JVM规范中唯一一个没有规定出现OOM的区域，所以这个空间也不会进行GC。

本地内存：

它又叫做**堆外内存**，线程共享的区域，本地内存这块区域是不会受到JVM的控制的，也就是说对于这块区域是不会发生GC的。因此对于整个java的执行效率是提升非常大的。

堆：

线程共享的区域。主要用来保存**对象实例，数组**等，当堆中没有内存空间可分配给实例，也无法再扩展时，则抛出OutOfMemoryError异常。

在JAVA7中堆内会存在**年轻代、老年代和方法区(永久代)。**

1）Young区被划分为三部分，Eden区和两个大小严格相同的Survivor区，其中，Survivor区间中，某一时刻只有其中一个是被使用的，另外一个留做垃圾收集时复制对象用。在Eden区变满的时候， GC就会将存活的对象移到空闲的Survivor区间中，根据JVM的策略，在经过几次垃圾收集后，任然存活于Survivor的对象将被移动到Tenured区间。

2）Tenured区主要保存生命周期长的对象，一般是一些老的对象，当一些对象在Young复制转移一定的次数以后，对象就会被转移到Tenured区。

3）Perm代主要保存**保存的类信息、静态变量、常量、编译后的代码**，在java7中堆上方法区会受到GC的管理的。方法区【永久代】是有一个大小的限制的。如果大量的动态生成类，就会放入到方法区【永久代】，很容易造成OOM。

为了避免方法区出现OOM，所以在java8中将堆上的方法区【永久代】给移动到了本地内存上，重新开辟了一块空间，叫做**元空间**。那么现在就可以避免掉OOM的出现了。

1.3.2 为什么要废弃1.7中的永久代?

官网给出了解释：http://openjdk.java.net/jeps/122

```
This is part of the JRockit and Hotspot convergence effort. JRockit customers do not need to configure the permanent generation (since JRockit does not have a permanent generation) and are accustomed to not configuring the permanent generation.

移除永久代是为融合HotSpot JVM与 JRockit VM而做出的努力，因为JRockit没有永久代，不需要配置永久代。
```

- 字符串存在永久代中，容易出现性能问题和内存溢出。
- 类及方法的信息等比较难确定其大小，因此对于永久代的大小指定比较困难，太小容易出现永久代溢出，太大则容易导致老年代溢出。
- 将 HotSpot 与 JRockit 合二为一。

1.3.3 通过jstat命令进行查看堆内存使用情况

jstat命令可以查看堆内存各部分的使用量，以及加载类的数量。命令的格式如下：

jstat [-命令选项] [vmid] [间隔时间/毫秒] [查询次数]

1.3.3.1、查看class加载统计

```
[root@node01 ~]# jps
7080 Jps
6219 Bootstrap

[root@node01 ~]# jstat -class 6219
Loaded  Bytes  Unloaded  Bytes     Time
  3273  7122.3        0     0.0       3.98
```

说明：

- Loaded：加载class的数量
- Bytes：所占用空间大小
- Unloaded：未加载数量
- Bytes：未加载占用空间
- Time：时间

1.3.3.2、查看编译统计

```
[root@node01 ~]# jstat -compiler 6219
Compiled Failed Invalid   Time   FailedType FailedMethod
    2376      1       0     8.04          1
org/apache/tomcat/util/IntrospectionUtils setProperty
```

说明：

- Compiled：编译数量。
- Failed：失败数量
- Invalid：不可用数量
- Time：时间
- FailedType：失败类型
- FailedMethod：失败的方法

1.3.3.3、垃圾回收统计

```
[root@node01 ~]# jstat -gc 6219
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU
   CCSC   CCSU   YGC     YGCT    FGC    FGCT     GCT
9216.0 8704.0  0.0   6127.3 62976.0   3560.4   33792.0    20434.9   23808.0
23196.1 2560.0 2361.6      7    1.078   1      0.244    1.323

#也可以指定打印的间隔和次数，每1秒中打印一次，共打印5次
[root@node01 ~]# jstat -gc 6219 1000 5
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU
   CCSC   CCSU   YGC     YGCT    FGC    FGCT     GCT
9216.0 8704.0  0.0   6127.3 62976.0   3917.3   33792.0    20434.9   23808.0
23196.1 2560.0 2361.6      7    1.078   1      0.244    1.323
9216.0 8704.0  0.0   6127.3 62976.0   3917.3   33792.0    20434.9   23808.0
23196.1 2560.0 2361.6      7    1.078   1      0.244    1.323
9216.0 8704.0  0.0   6127.3 62976.0   3917.3   33792.0    20434.9   23808.0
23196.1 2560.0 2361.6      7    1.078   1      0.244    1.323
9216.0 8704.0  0.0   6127.3 62976.0   3917.3   33792.0    20434.9   23808.0
23196.1 2560.0 2361.6      7    1.078   1      0.244    1.323
9216.0 8704.0  0.0   6127.3 62976.0   3917.3   33792.0    20434.9   23808.0
23196.1 2560.0 2361.6      7    1.078   1      0.244    1.323
```

说明：

- S0C：第一个Survivor区的大小（KB）
- S1C：第二个Survivor区的大小（KB）
- S0U：第一个Survivor区的使用大小（KB）
- S1U：第二个Survivor区的使用大小（KB）
- EC：Eden区的大小（KB）
- EU：Eden区的使用大小（KB）
- OC：Old区大小（KB）
- OU：Old使用大小（KB）
- MC：方法区(元空间)大小（KB）
- MU：方法区使用大小（KB）
- CCSC：压缩类空间大小（KB）
- CCSU：压缩类空间使用大小（KB）
- YGC：年轻代垃圾回收次数
- YGCT：年轻代垃圾回收消耗时间
- FGC：老年代垃圾回收次数
- FGCT：老年代垃圾回收消耗时间
- GCT：垃圾回收消耗总时间

1.4 jmap的使用以及内存溢出分析

前面通过jstat可以对jvm堆的内存进行统计分析，而jmap可以获取到更加详细的内容，如：内存使用情况的汇总、对内存溢出的定位与分析。

1.4.1 查看内存使用情况

```
[root@node01 ~]# jmap -heap 6219
Attaching to process ID 6219, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.141-b15

using thread-local object allocation.
Parallel GC with 2 thread(s)

Heap Configuration: #堆内存配置信息
   MinHeapFreeRatio         = 0
   MaxHeapFreeRatio         = 100
   MaxHeapSize              = 488636416 (466.0MB)
   NewSize                  = 10485760 (10.0MB)
   MaxNewSize               = 162529280 (155.0MB)
   OldSize                  = 20971520 (20.0MB)
   NewRatio                 = 2
   SurvivorRatio            = 8
   MetaspaceSize            = 21807104 (20.796875MB)
   CompressedClassSpaceSize = 1073741824 (1024.0MB)
   MaxMetaspaceSize         = 17592186044415 MB
   G1HeapRegionSize         = 0 (0.0MB)

Heap Usage: # 堆内存的使用情况
PS Young Generation #年轻代
Eden Space:
   capacity = 123731968 (118.0MB)
   used     = 1384736 (1.320587158203125MB)
```

```
   free     = 122347232 (116.67941284179688MB)
   1.1191416594941737% used
From Space:
   capacity = 9437184 (9.0MB)
   used     = 0 (0.0MB)
   free     = 9437184 (9.0MB)
   0.0% used
To Space:
   capacity = 9437184 (9.0MB)
   used     = 0 (0.0MB)
   free     = 9437184 (9.0MB)
   0.0% used
PS Old Generation #年老代
   capacity = 28311552 (27.0MB)
   used     = 13698672 (13.064071655273438MB)
   free     = 14612880 (13.935928344726562MB)
   48.38545057508681% used

13648 interned Strings occupying 1866368 bytes.
```

1.4.2 查看内存中对象数量及大小

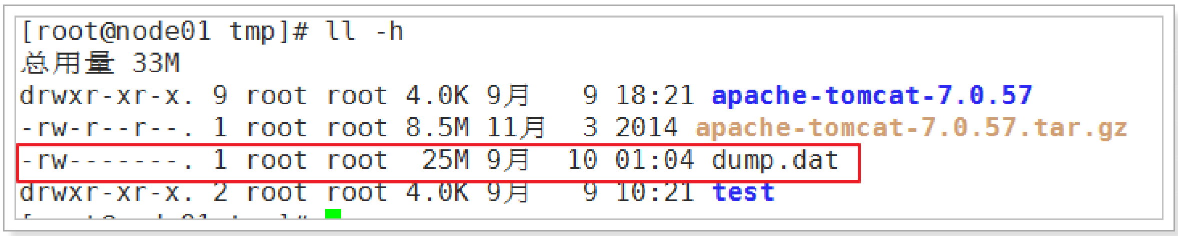
```
#查看所有对象，包括活跃以及非活跃的
jmap -histo <pid> | more

#查看活跃对象
jmap -histo:live <pid> | more

[root@node01 ~]# jmap -histo:live 6219 | more

 num     #instances         #bytes  class name
----------------------------------------------
   1:         37437        7914608  [C
   2:         34916         837984  java.lang.String
   3:           884         654848  [B
   4:         17188         550016  java.util.HashMap$Node
   5:          3674         424968  java.lang.Class
   6:          6322         395512  [Ljava.lang.Object;
   7:          3738         328944  java.lang.reflect.Method
   8:          1028         208048  [Ljava.util.HashMap$Node;
   9:          2247         144264  [I
  10:          4305         137760  java.util.concurrent.ConcurrentHashMap$Node
  11:          1270         109080  [Ljava.lang.String;
  12:            64          84128
[Ljava.util.concurrent.ConcurrentHashMap$Node;
  13:          1714          82272  java.util.HashMap
  14:          3285          70072  [Ljava.lang.Class;
  15:          2888          69312  java.util.ArrayList
  16:          3983          63728  java.lang.Object
  17:          1271          61008
org.apache.tomcat.util.digester.CallMethodRule
  18:          1518          60720  java.util.LinkedHashMap$Entry
  19:          1671          53472  com.sun.org.apache.xerces.internal.xni.QName
  20:            88          50880  [Ljava.util.WeakHashMap$Entry;
  21:           618          49440  java.lang.reflect.Constructor
  22:          1545          49440  java.util.Hashtable$Entry
  23:          1027          41080  java.util.TreeMap$Entry
```

```
  24:           846         40608  org.apache.tomcat.util.modeler.AttributeInfo
  25:           142         38032  [S
  26:           946         37840  java.lang.ref.SoftReference
  27:           226         36816  [[C
  。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。

#对象说明
B  byte
C  char
D  double
F  float
I  int
J  long
Z  boolean
[  数组，如[I表示int[]
[L+类名 其他对象
```

1.4.3 将内存使用情况dump到文件中

有些时候我们需要将jvm当前内存中的情况dump到文件中，然后对它进行分析，jmap也是支持dump到文件中的。

```
#用法：
jmap -dump:format=b,file=dumpFileName <pid>

#示例
jmap -dump:format=b,file=/tmp/dump.dat 6219
```



```
[root@node01 tmp]# ll -h
总用量 33M
drwxr-xr-x. 9 root root 4.0K 9月   9 18:21 apache-tomcat-7.0.57
-rw-r--r--. 1 root root 8.5M 11月  3 2014 apache-tomcat-7.0.57.tar.gz
-rw-------. 1 root root  25M 9月  10 01:04 dump.dat
drwxr-xr-x. 2 root root 4.0K 9月   9 10:21 test
```

可以看到已经在/tmp下生成了dump.dat的文件。

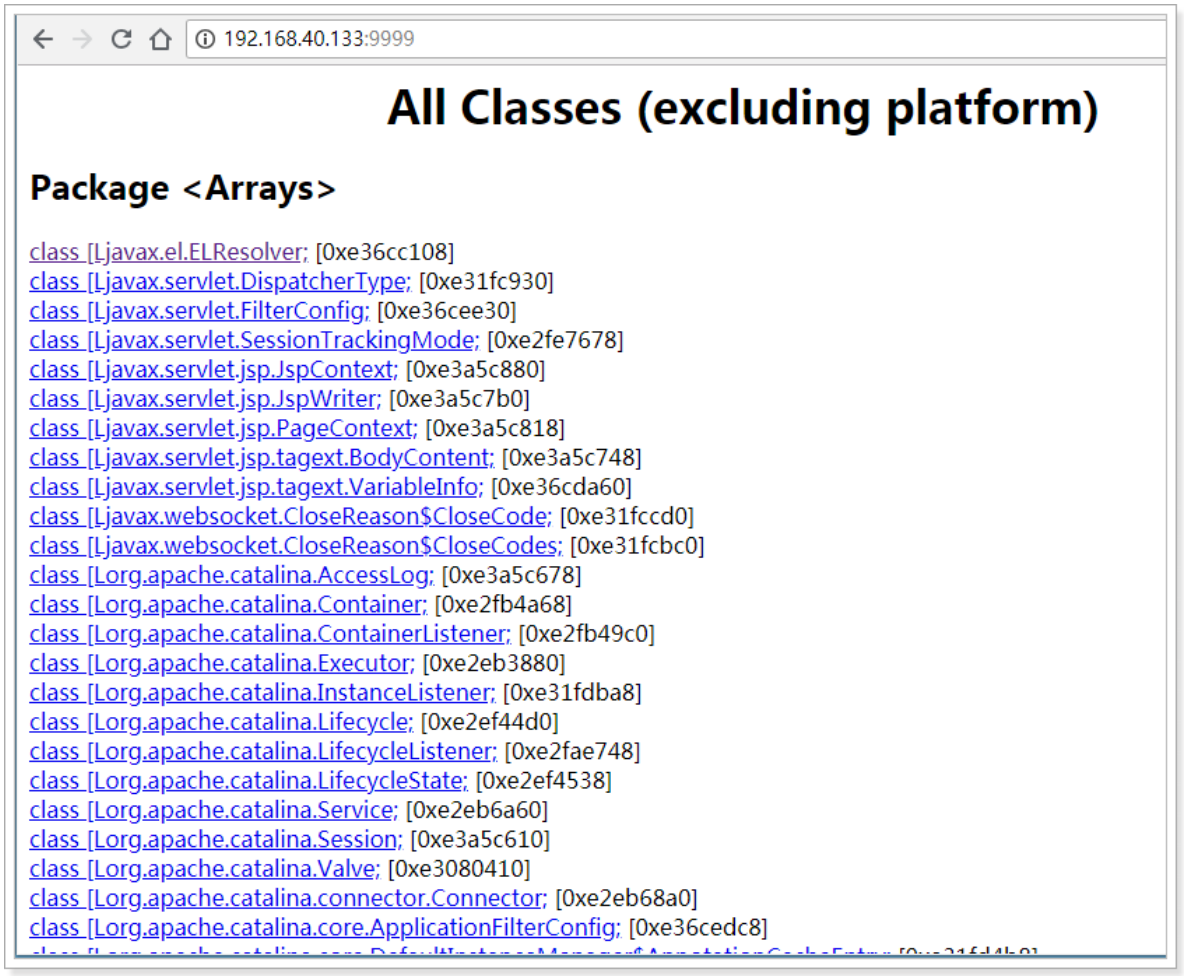
1.4.4 通过jhat对dump文件进行分析

在上一小节中，我们将jvm的内存dump到文件中，这个文件是一个二进制的文件，不方便查看，这时我们可以借助于jhat工具进行查看。

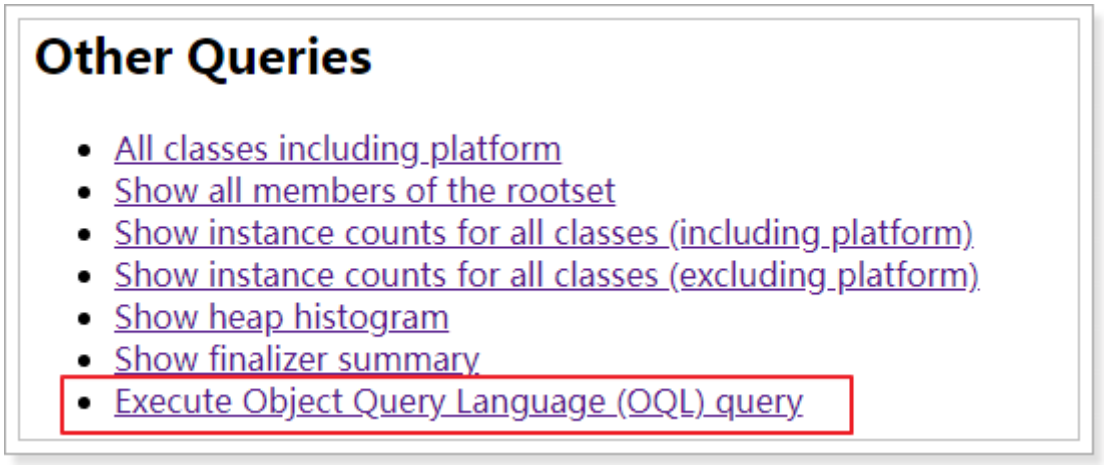
```
#用法：
jhat -port <port> <file>

#示例：
[root@node01 tmp]# jhat -port 9999 /tmp/dump.dat
Reading from /tmp/dump.dat...
Dump file created Mon Sep 10 01:04:21 CST 2018
Snapshot read, resolving...
Resolving 204094 objects...
Chasing references, expect 40 dots........................................
Eliminating duplicate references........................................
Snapshot resolved.
```

```
Started HTTP server on port 9999
Server is ready.
```

打开浏览器进行访问：http://192.168.40.133:9999/



在最后面有OQL查询功能。



1.5 VisualVM

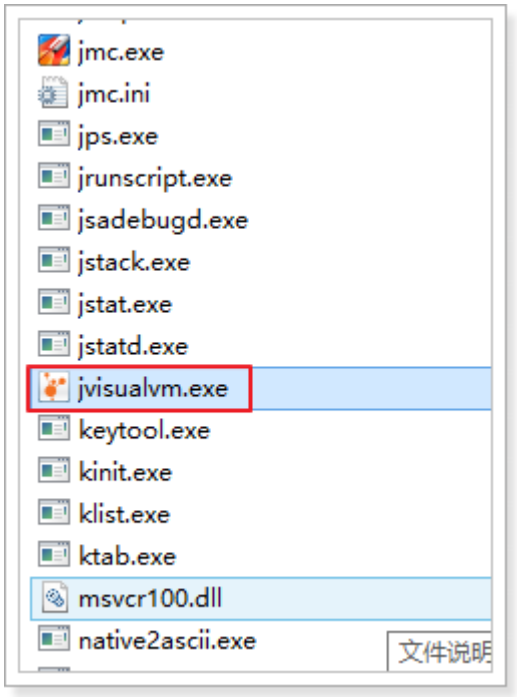
VisualVM，能够监控线程、内存情况，查看方法的CPU时间和内存中的对象、已被GC的对象、反向查看分配的堆栈(如100个String对象分别由哪几个对象分配出来的)。

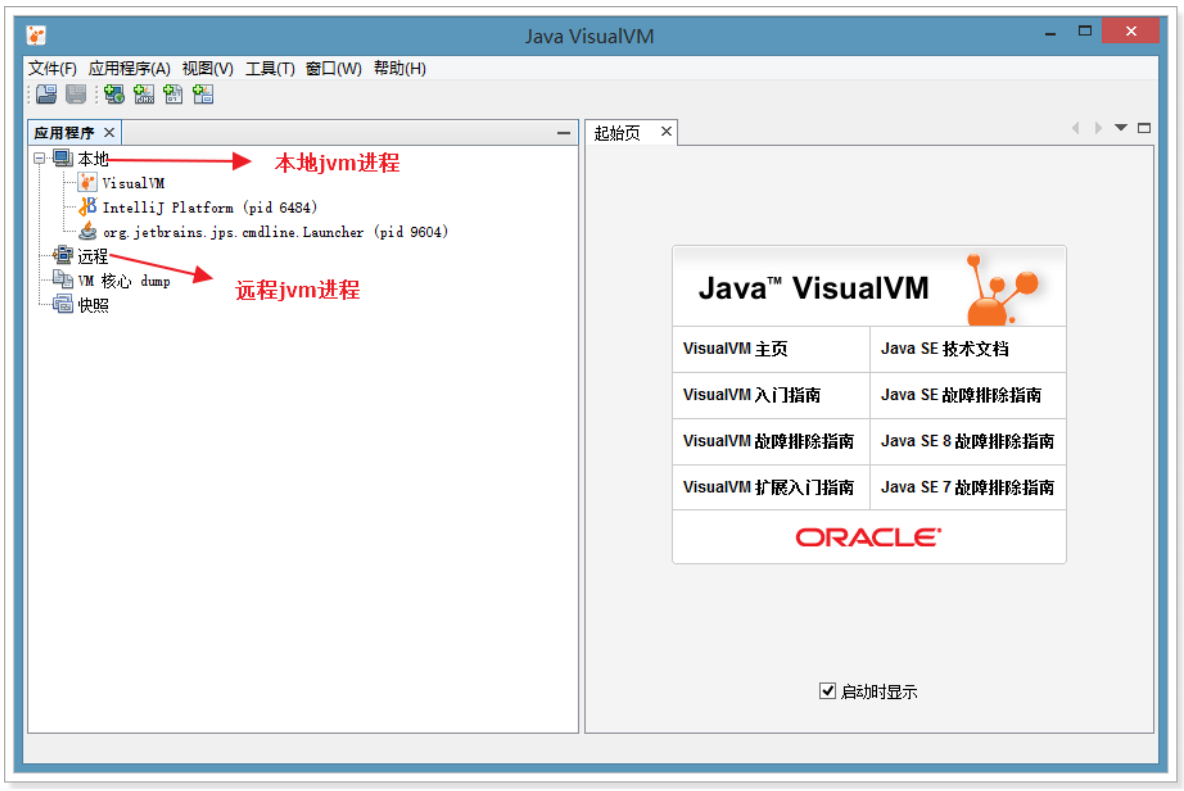
VisualVM使用简单，几乎0配置，功能还是比较丰富的，几乎囊括了其它JDK自带命令的所有功能。

- 内存信息
- 线程信息
- Dump堆（本地进程）
- Dump线程（本地进程）
- 打开堆Dump。堆Dump可以用jmap来生成。
- 打开线程Dump
- 生成应用快照（包含内存信息、线程信息等等）
- 性能分析。CPU分析（各个方法调用时间，检查哪些方法耗时多），内存分析（各类对象占用的内存，检查哪些类占用内存多）
- ……

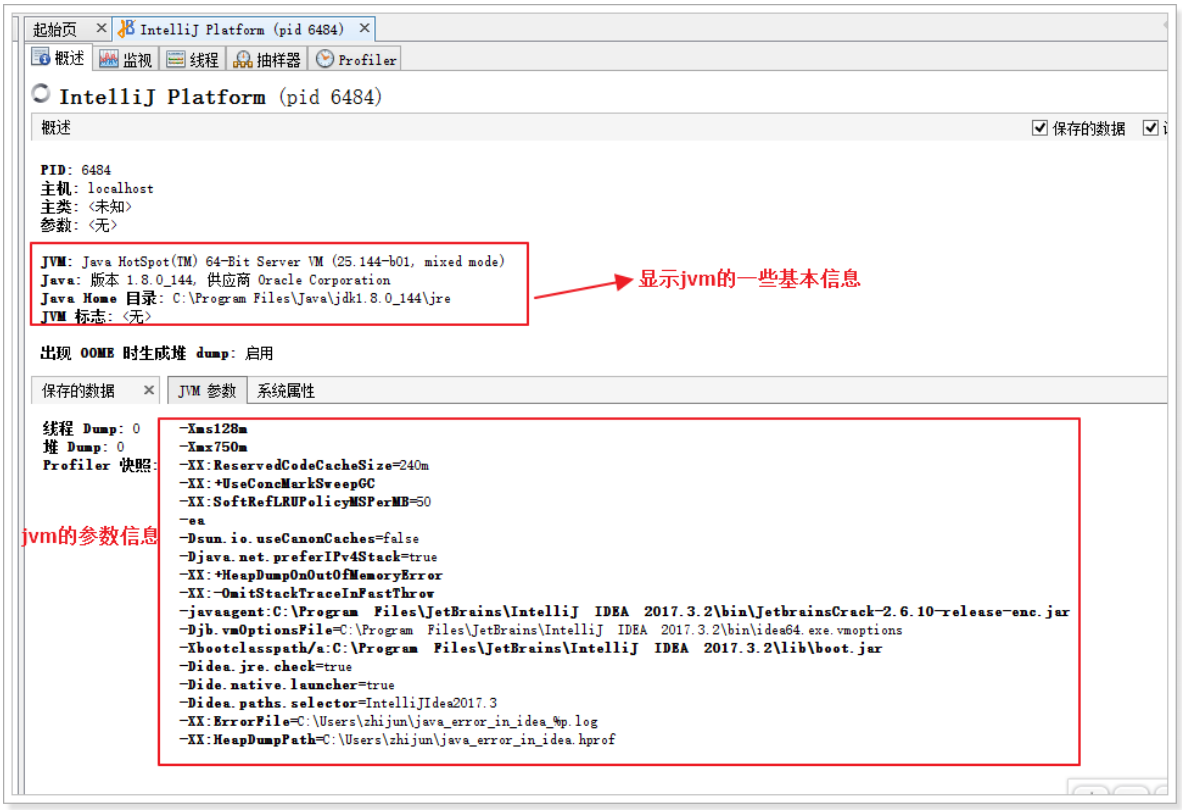
1.5.1 启动

在jdk的安装目录的bin目录下，找到**jvisualvm.exe**，双击打开即可。

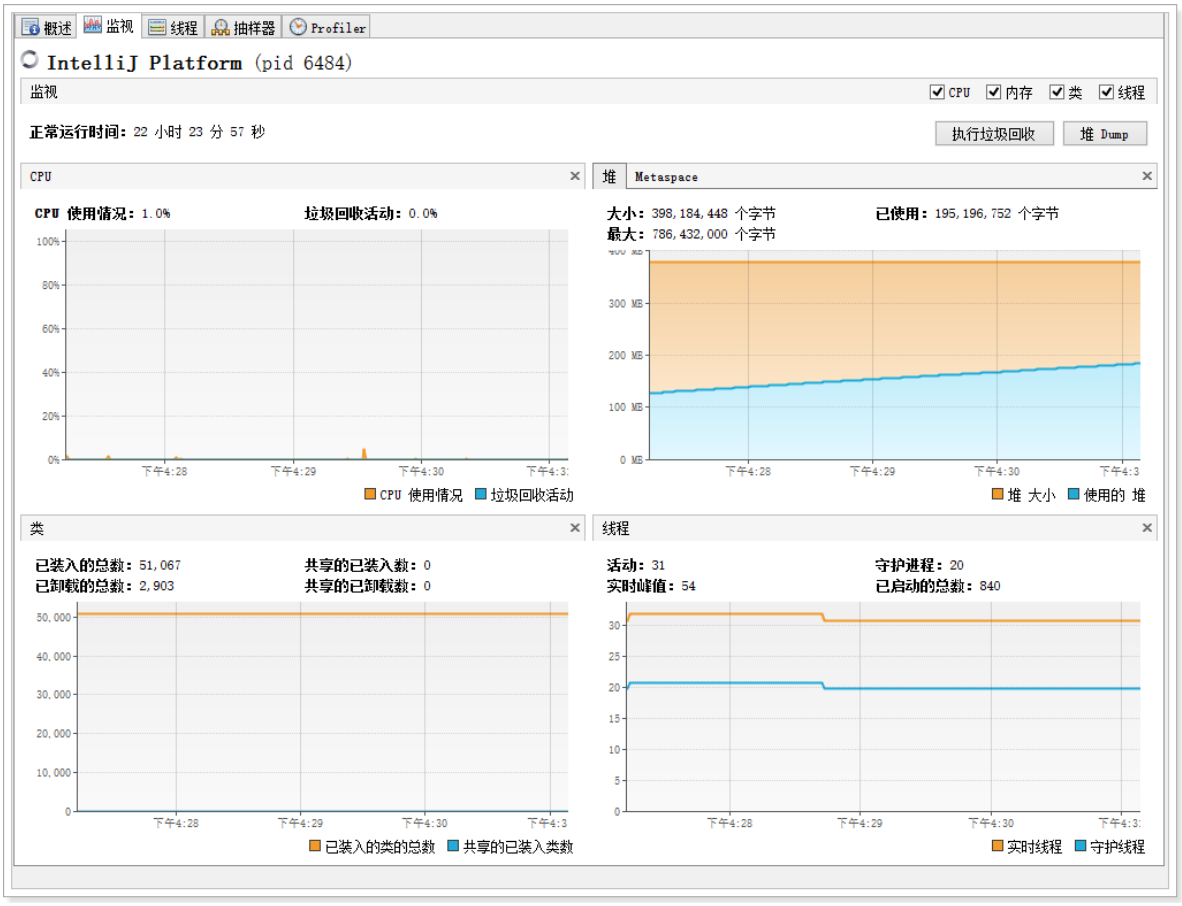

jmc.exe
jmc.ini
jps.exe
jrunscript.exe
jsadebugd.exe
jstack.exe
jstat.exe
jstatd.exe
jvisualvm.exe
keytool.exe
kinit.exe
klist.exe
ktab.exe
msvcr100.dll
native2ascii.exe
文件说明


Java VisualVM
文件(F) 应用程序(A) 视图(V) 工具(T) 窗口(W) 帮助(H)
应用程序
本地
VisualVM
IntelliJ Platform (pid 6484)
org.jetbrains.jps.cmdline.Launcher (pid 9604)
远程
VM 核心 dump
快照
本地jvm进程
远程jvm进程
起始页
Java™ VisualVM
VisualVM 主页
Java SE 技术文档
VisualVM 入门指南
Java SE 故障排除指南
VisualVM 故障排除指南
Java SE 8 故障排除指南
VisualVM 扩展入门指南
Java SE 7 故障排除指南
ORACLE
启动时显示

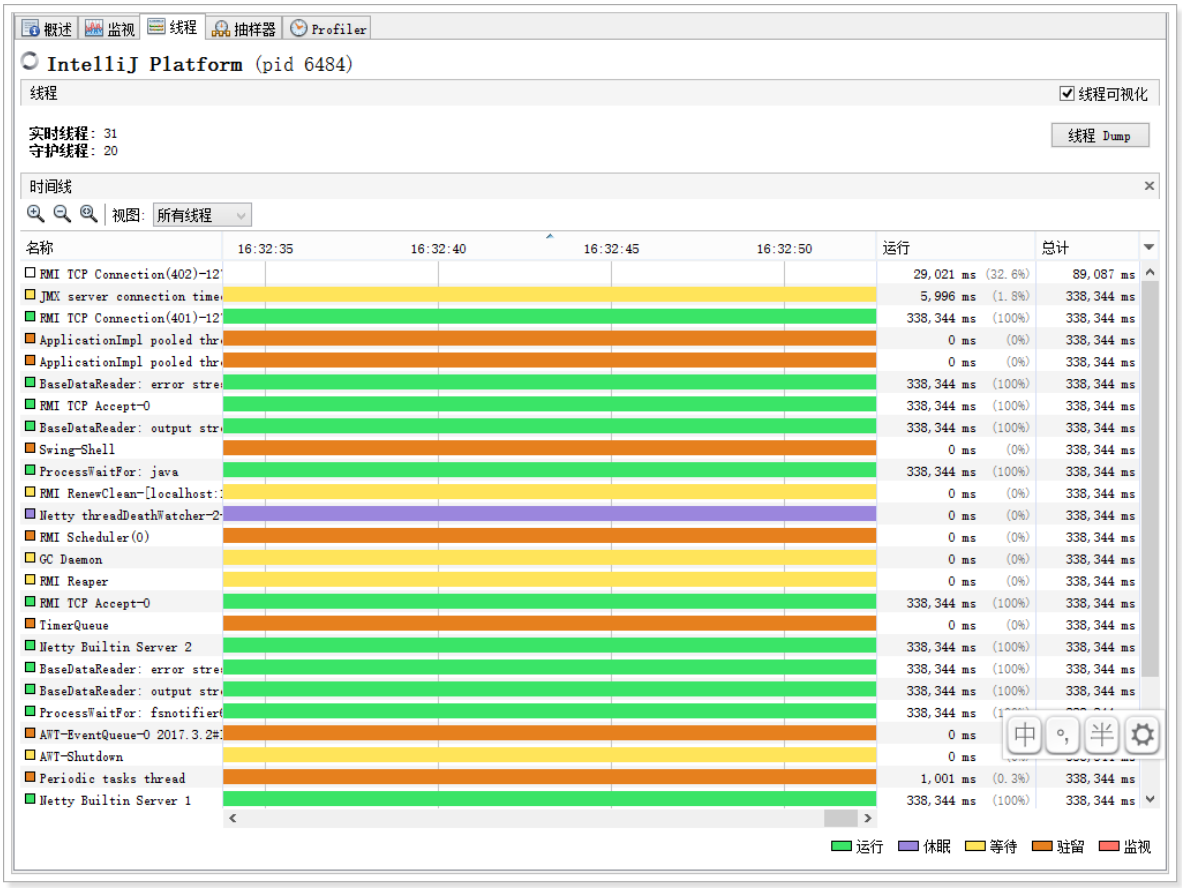
1.5.2 查看本地进程



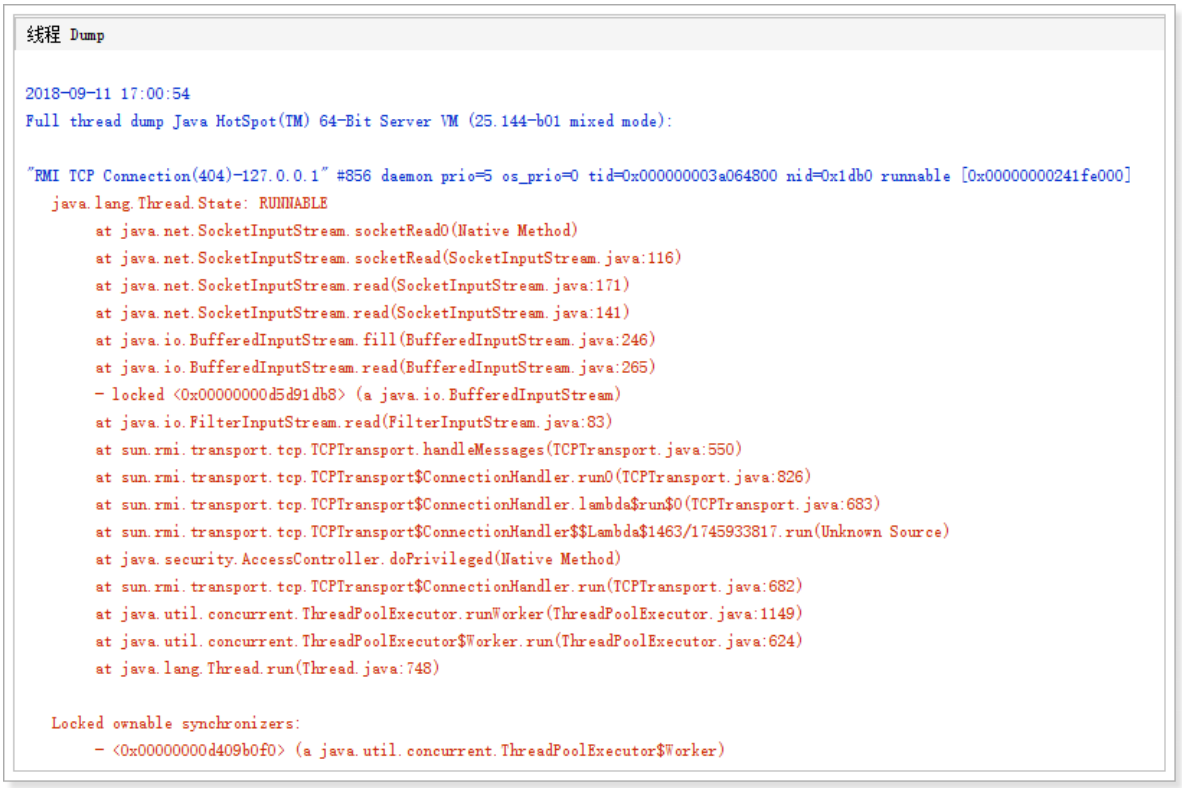
1.5.3 查看CPU、内存、类、线程运行信息



1.5.4 查看线程详情



也可以点击右上角Dump按钮，将线程的信息导出，其实就是执行的jstack命令。

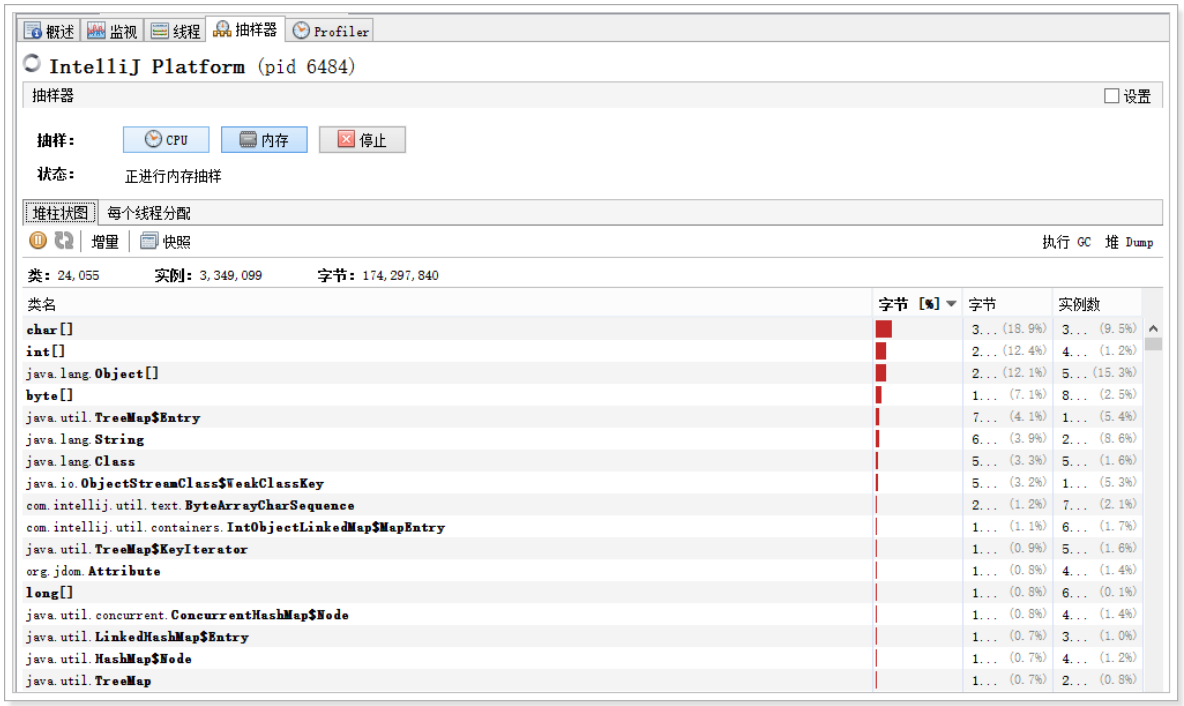


```
线程 Dump

2018-09-11 17:00:54
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.144-b01 mixed mode):

"RMI TCP Connection(404)-127.0.0.1" #856 daemon prio=5 os_prio=0 tid=0x000000003a064800 nid=0x1db0 runnable [0x00000000241fe000]
   java.lang.Thread.State: RUNNABLE
        at java.net.SocketInputStream.socketRead0(Native Method)
        at java.net.SocketInputStream.socketRead(SocketInputStream.java:116)
        at java.net.SocketInputStream.read(SocketInputStream.java:171)
        at java.net.SocketInputStream.read(SocketInputStream.java:141)
        at java.io.BufferedInputStream.fill(BufferedInputStream.java:246)
        at java.io.BufferedInputStream.read(BufferedInputStream.java:265)
        - locked <0x00000000d5d91db8> (a java.io.BufferedInputStream)
        at java.io.FilterInputStream.read(FilterInputStream.java:83)
        at sun.rmi.transport.tcp.TCPTransport.handleMessages(TCPTransport.java:550)
        at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run0(TCPTransport.java:826)
        at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.lambda$run$0(TCPTransport.java:683)
        at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler$$Lambda$1463/1745933817.run(Unknown Source)
        at java.security.AccessController.doPrivileged(Native Method)
        at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run(TCPTransport.java:682)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
        at java.lang.Thread.run(Thread.java:748)

   Locked ownable synchronizers:
        - <0x00000000d409b0f0> (a java.util.concurrent.ThreadPoolExecutor$Worker)
```

发现，显示的内容是一样的。

1.5.5 抽样器

抽样器可以对CPU、内存在一段时间内进行抽样，以供分析。

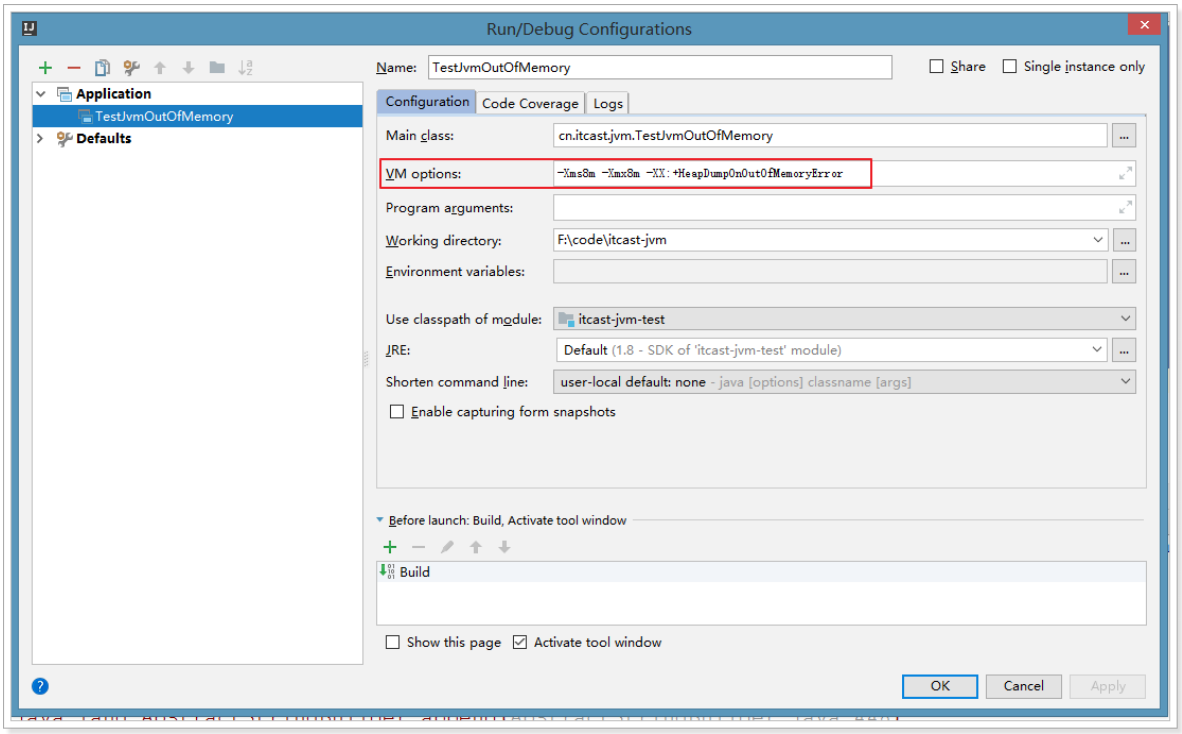


1.5.6 模拟内存溢出

编写代码，向List集合中添加100万个字符串，每个字符串由1000个UUID组成。如果程序能够正常执行，最后打印ok。

```
public class TestJvmOutOfMemory {

    public static void main(String[] args) {
        List<Object> list = new ArrayList<>();
        for (int i = 0; i < 10000000; i++) {
            String str = "";
            for (int j = 0; j < 1000; j++) {
                str += UUID.randomUUID().toString();
            }
            list.add(str);
        }
        System.out.println("ok");
    }
}
```

为了演示效果，我们将设置执行的参数，这里使用的是Idea编辑器。



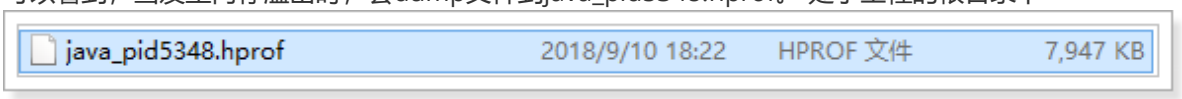

```
#参数如下：
-Xms8m -Xmx8m -XX:+HeapDumpOnOutOfMemoryError
```

运行测试

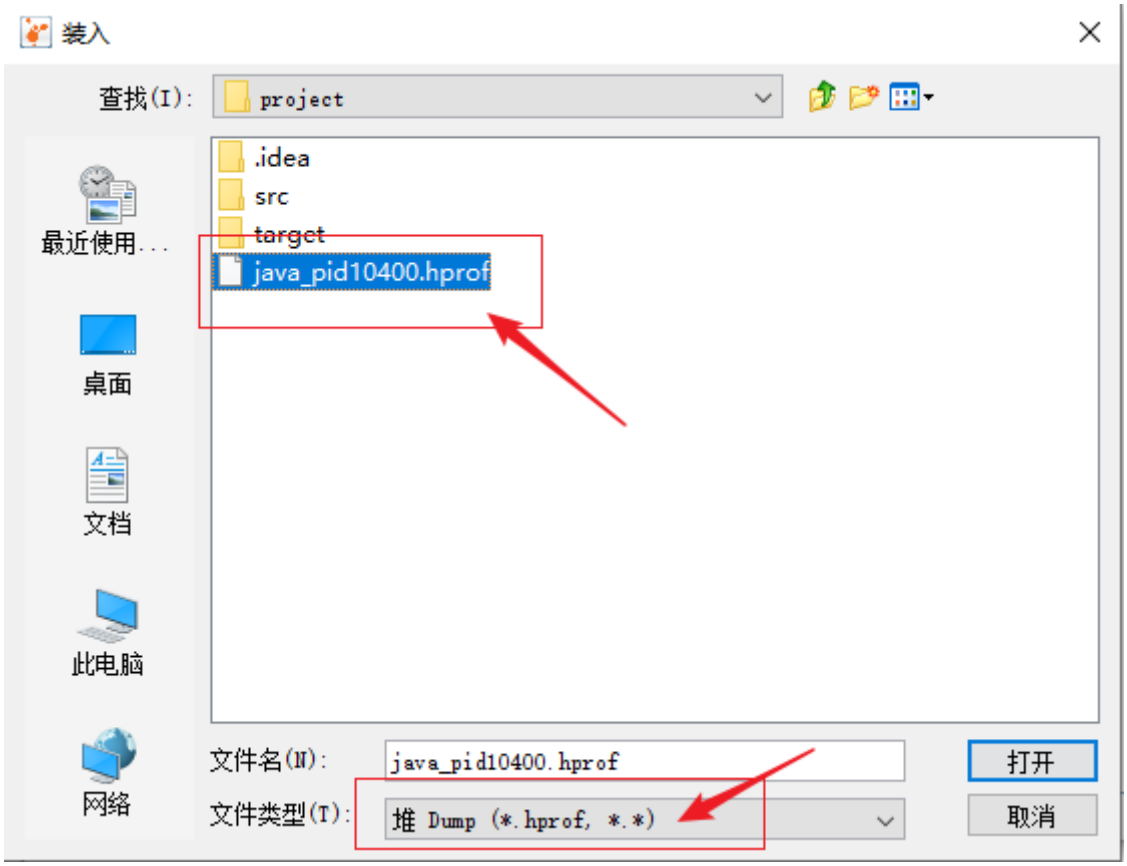
测试结果如下：

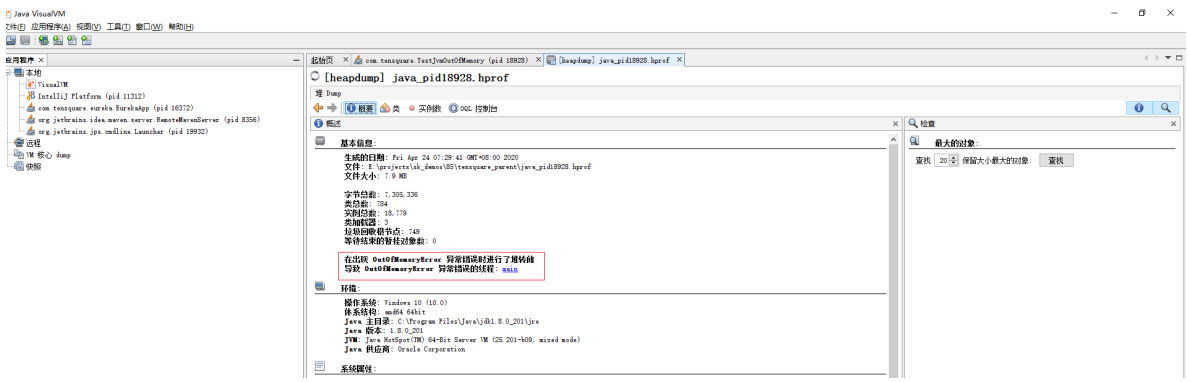
```
java.lang.OutOfMemoryError: Java heap space
Dumping heap to java_pid5348.hprof ...
Heap dump file created [8137186 bytes in 0.032 secs]
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at java.util.Arrays.copyOf(Arrays.java:3332)
    at
java.lang.AbstractStringBuilder.ensureCapacityInternal(AbstractStringBuilder.jav
a:124)
    at java.lang.AbstractStringBuilder.append(AbstractStringBuilder.java:448)
    at java.lang.StringBuilder.append(StringBuilder.java:136)
    at cn.itcast.jvm.TestJvmOutOfMemory.main(TestJvmOutOfMemory.java:14)

Process finished with exit code 1
```

可以看到，当发生内存溢出时，会dump文件到java_pid5348.hprof。 处于工程的根目录下



导入dump文件到visualJvm中





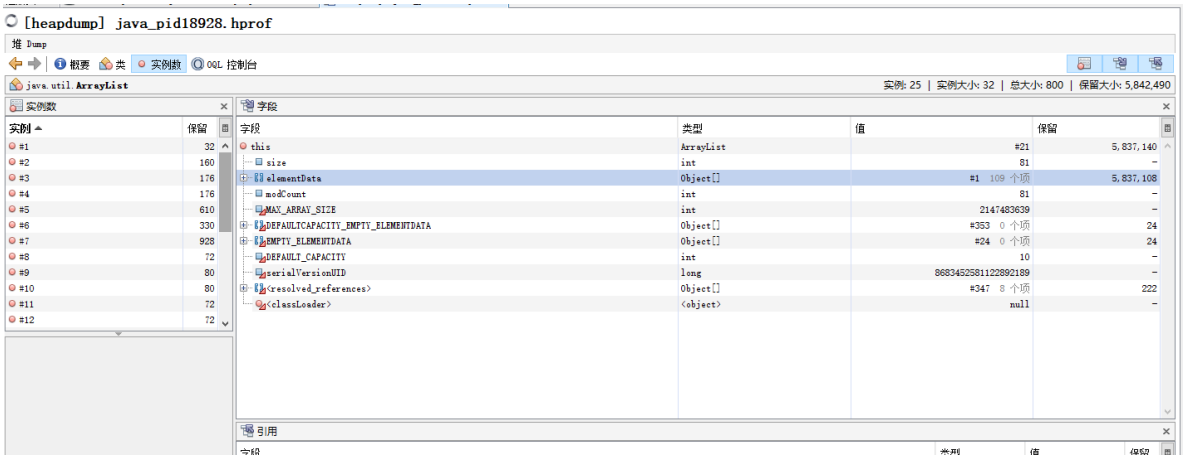
在概览中点击异常对应的方法

```
at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:153)

"Monitor Ctrl-Break" daemon prio=5 tid=6 RUNNABLE
at java.net.SocketInputStream.socketRead0(Native Method)
at java.net.SocketInputStream.socketRead(SocketInputStream.java:116)
at java.net.SocketInputStream.read(SocketInputStream.java:171)
   Local Variable: java.io.FileDescriptor#4
at java.net.SocketInputStream.read(SocketInputStream.java:141)
   Local Variable: java.net.SocketInputStream#1
   Local Variable: byte[]#97
at sun.nio.cs.StreamDecoder.readBytes(StreamDecoder.java:284)
at sun.nio.cs.StreamDecoder.implRead(StreamDecoder.java:326)
   Local Variable: java.nio.HeapCharBuffer#1
   Local Variable: java.nio.charset.CoderResult#1
at sun.nio.cs.StreamDecoder.read(StreamDecoder.java:178)
   Local Variable: sun.nio.cs.StreamDecoder#1
at java.io.InputStreamReader.read(InputStreamReader.java:184)
   Local Variable: char[]#1512
at java.io.BufferedReader.fill(BufferedReader.java:161)
at java.io.BufferedReader.readLine(BufferedReader.java:324)
   Local Variable: java.io.InputStreamReader#1
at java.io.BufferedReader.readLine(BufferedReader.java:389)
at com.intellij.rt.execution.application.AppMainV2$1.run(AppMainV2.java:64)
   Local Variable: java.net.Socket#1
   Local Variable: java.io.BufferedReader#1

"main" prio=5 tid=1 RUNNABLE
at java.lang.OutOfMemoryError.<init>(OutOfMemoryError.java:48)
at java.util.Arrays.copyOf(Arrays.java:3332)
   Local Variable: char[]#20
at java.lang.AbstractStringBuilder.ensureCapacityInternal(AbstractStringBuilder.java:124)
at java.lang.AbstractStringBuilder.append(AbstractStringBuilder.java:448)
   Local Variable: java.lang.String#19
at java.lang.StringBuilder.append(StringBuilder.java:136)
   Local Variable: java.lang.StringBuilder#1
at com.tensquare.TestJvmOutOfMemory.main(TestJvmOutOfMemory.java:14)
   Local Variable: java.util.ArrayList#21

"Finalizer" daemon prio=8 tid=3 WAITING
at java.lang.Object.wait(Native Method)
at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:144)
at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:165)
```

发生异常

最终的位置

可以点进ArrayList中查看更详细的信息



2 垃圾回收机制

2.1 Java语言的垃圾回收

为了让程序员更专注于代码的实现，而不用过多的考虑内存释放的问题，所以，在Java语言中，有了自动的垃圾回收机制，也就是我们熟悉的GC。

有了垃圾回收机制后，程序员只需要关心内存的申请即可，内存的释放由系统自动识别完成。

换句话说，自动的垃圾回收的算法就会变得非常重要了，如果因为算法的不合理，导致内存资源一直没有释放，同样也可能会导致内存溢出的。

当然，除了Java语言，C#、Python等语言也都有自动的垃圾回收机制。

2.2 什么是垃圾&垃圾定位

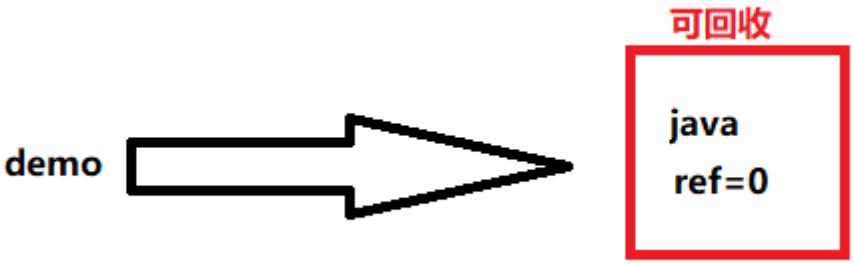
简单一句就是：**如果一个或多个对象没有任何的引用指向它了，那么这个对象现在就是垃圾。**

2.2.1 引用计数法

一个对象被引用了一次，在当前的对象头上递增一次引用次数，如果这个对象的引用次数为0，代表这个对象可回收

```
String demo = new String("123");
```


```
String demo = null;
```



当对象间出现了循环引用的话，则引用计数法就会失效

```
public static void main(String[] args) {

    Demo a = new Demo( info: "a");
    Demo b = new Demo( info: "b");

    a.instance = b;
    b.instance = a;


    a=null;
    b=null;

}
```

虽然a和b都为null，但是由于a和b存在循环引用，这样a和b永远都不会被回收。

优点：

- 实时性较高，无需等到内存不够的时候，才开始回收，运行时根据对象的计数器是否为0，就可以直接回收。
- 在垃圾回收过程中，应用无需挂起。如果申请内存时，内存不足，则立刻报OOM错误。

- 区域性，更新对象的计数器时，只是影响到该对象，不会扫描全部对象。

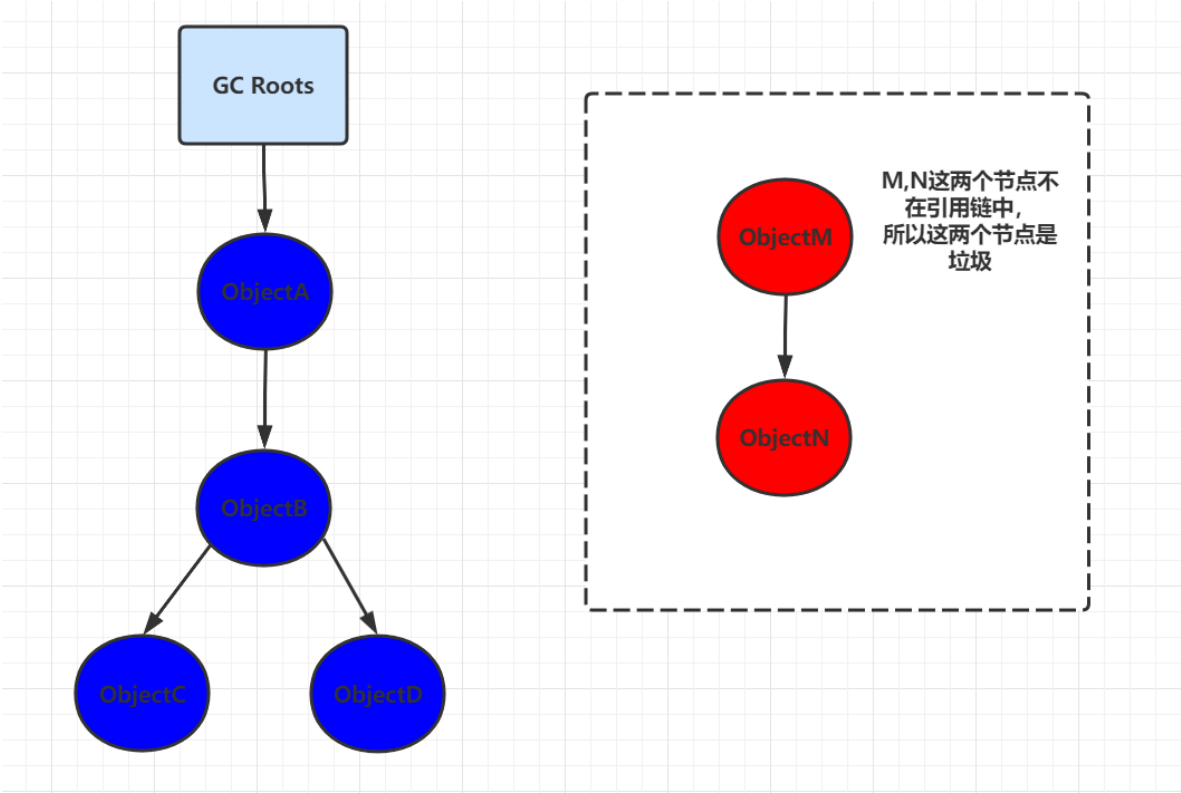
缺点：

- 每次对象被引用时，都需要去更新计数器，有一点时间开销。
- **浪费CPU资源**，即使内存够用，仍然在运行时进行计数器的统计。
- **无法解决循环引用问题，会引发内存泄露。**（最大的缺点）

2.2.2 可达性分析算法

现在的虚拟机采用的都是通过可达性分析算法来确定哪些内容是垃圾。

会存在一个根节点【GC Roots】，引出它下面指向的下一个节点，再以下一个节点节点开始找出它下面的节点，依次往下类推。直到所有的节点全部遍历完毕。



M,N这两个节点是可回收的，但是**并不会马上的被回收！！** 对象中存在一个方法【finalize】。当对象被标记为可回收后，当发生GC时，首先**会判断这个对象是否执行了finalize方法**，如果这个方法还没有被执行的话，那么就会先来执行这个方法，接着在这个方法执行中，可以设置当前这个对象与GC ROOTS产生关联，那么这个方法执行完成之后，GC会再次判断对象是否可达，如果仍然不可达，则会进行回收，如果可达了，则不会进行回收。

finalize方法对于每一个对象来说，只会执行一次。如果第一次执行这个方法的时候，设置了当前对象与RC ROOTS关联，那么这一次不会进行回收。 那么等到这个对象第二次被标记为可回收时，那么该对象的finalize方法就不会再次执行了。

GC ROOTS：

虚拟机栈中引用的对象

本地方法栈中引用的对象

方法区中类静态属性引用的对象

方法区中常量引用对象

2.3 垃圾回收算法

2.3.1 标记清除算法

标记清除算法，是将垃圾回收分为2个阶段，分别是**标记和清除。**

1.根据可达性分析算法得出的垃圾进行标记

2.对这些标记为可回收的内容进行垃圾回收

可以看到，标记清除算法解决了引用计数算法中的循环引用的问题，没有从root节点引用的对象都会被回收。

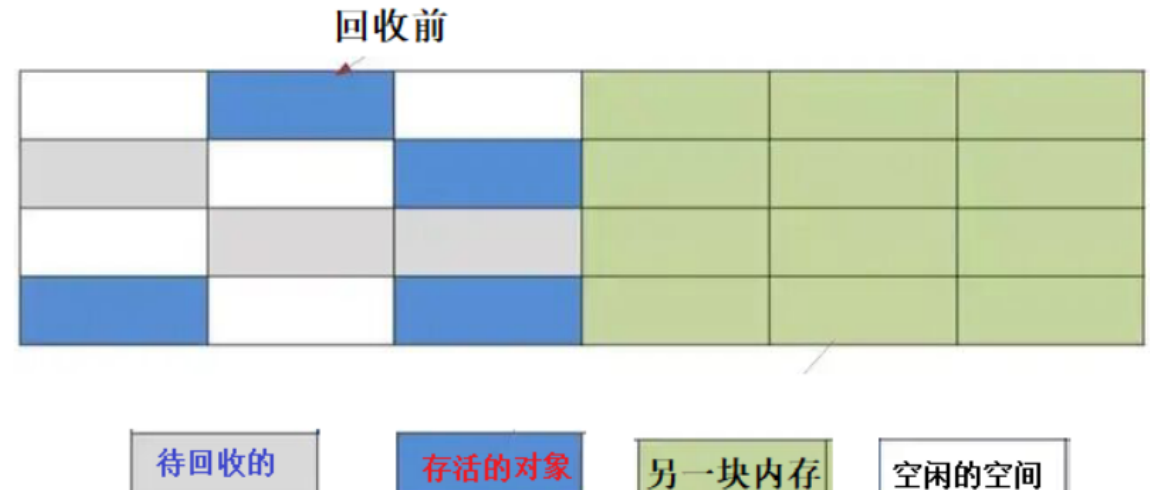
同样，标记清除算法也是有缺点的：

- 效率较低，**标记和清除两个动作都需要遍历所有的对象**，并且在GC时，**需要停止应用程序**，对于交互性要求比较高的应用而言这个体验是非常差的。
- （**重要**）通过标记清除算法清理出来的内存，碎片化较为严重，因为被回收的对象可能存在于内存的各个角落，所以清理出来的内存是不连贯的。

2.3.2 复制算法

复制算法的核心就是，**将原有的内存空间一分为二，每次只用其中的一块**，在垃圾回收时，将正在使用的对象复制到另一个内存空间中，然后将该内存空间清空，交换两个内存的角色，完成垃圾的回收。

如果内存中的垃圾对象较多，需要复制的对象就较少，这种情况下适合使用该方式并且效率比较高，反之，则不适合。



1）将内存区域分成两部分，每次操作其中一个。

2）当进行垃圾回收时，将正在使用的内存区域中的存活对象移动到未使用的内存区域。当移动完对这部分内存区域一次性清除。

3）周而复始。

优点：

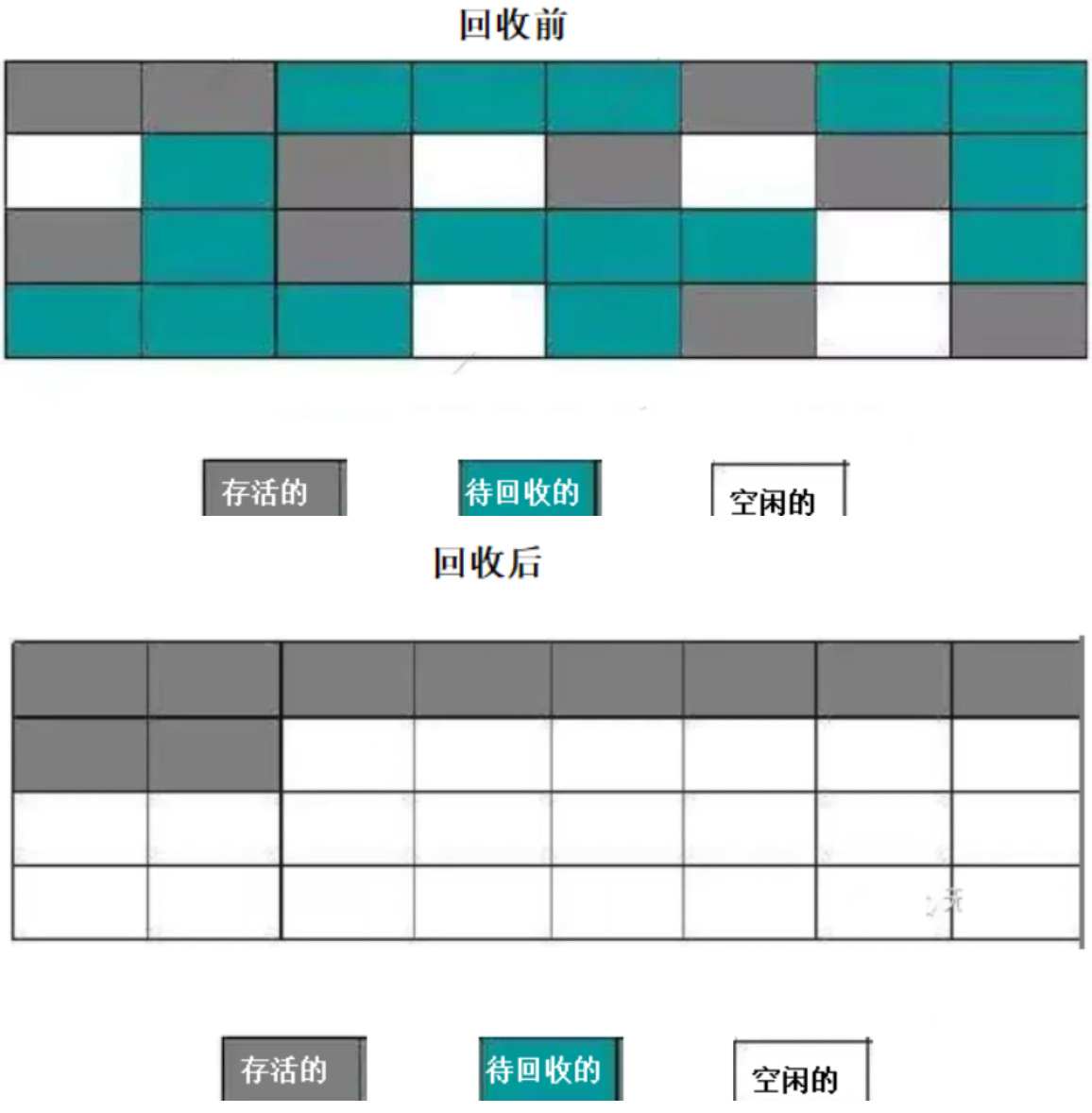
- 在垃圾对象多的情况下，效率较高
- 清理后，内存无碎片

缺点：

- 分配的2块内存空间，在同一个时刻，只能使用一半，内存使用率较低

2.3.3 标记压缩算法

标记压缩算法是在标记清除算法的基础之上，做了优化改进的算法。和标记清除算法一样，也是从根节点开始，对对象的引用进行标记，在清理阶段，并不是简单的直接清理可回收对象，而是将存活对象都向内存另一端移动，然后清理边界以外的垃圾，从而解决了碎片化的问题。



1）标记垃圾。

2）需要清除向右边走，不需要清除的向左边走。

3）清除边界以外的垃圾。

优缺点同标记清除算法，解决了标记清除算法的碎片化的问题，同时，标记压缩算法多了一步，对象移动内存位置的步骤，其效率也有有一定的影响。

2.2.6 分代收集算法

在java8时，堆被分为了两份：**新生代和老年代【1：2】**，在java7时，还存在一个永久代。

对于新生代，内部又被分为了三个区域。Eden区，S0区，S1区【8：1：1】

当对新生代产生GC：MinorGC【young GC】

当对老年代产生GC：FullGC【OldGC】

2.2.6.1 工作机制

1）当创建一个对象的时候，那么这个对象会被分配在新生代的Eden区。当Eden区要满了时候，触发YoungGC。

2）当进行YoungGC后，此时在Eden区存活的对象被移动到S0区，并且**当前对象的年龄会加1**，清空Eden区。

3）当再一次触发YoungGC的时候，会把Eden区中存活下来的对象和S0中的对象，移动到S1区中，这些对象的年龄会加1，清空Eden区和S0区。

4）当再一次触发YoungGC的时候，会把Eden区中存活下来的对象和S1中的对象，移动到S0区中，这些对象的年龄会加1，清空Eden区和S1区。

2.2.6.2 对象何时晋升到老年代

1）对象的年龄达到了某一个限定的值（**默认15岁**，CMS默认6岁 ），那么这个对象就会进入到老年代中。

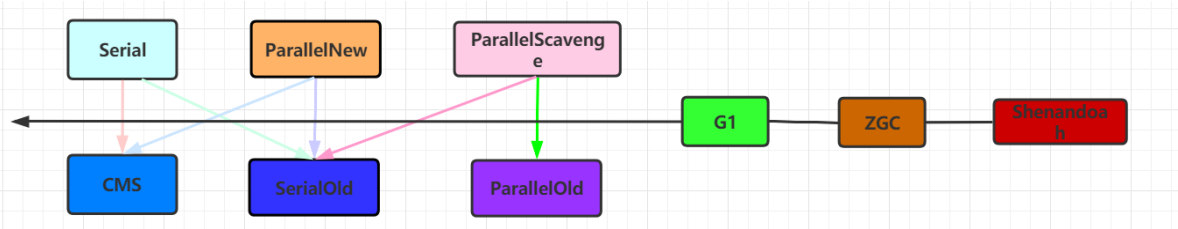
2）大对象。

3）如果在Survivor区中相同年龄的对象的所有大小之和超过Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代。

当老年代满了之后，**触发FullGC。FullGC同时回收新生代和老年代**，当前只会存在一个FullGC的线程进行执行，其他的线程全部会被挂起。

2.3 垃圾收集器以及内存分配

前面我们讲了垃圾回收的算法，还需要有具体的实现，在jvm中，实现了多种垃圾收集器，包括：串行垃圾收集器、并行垃圾收集器、CMS（并发）垃圾收集器、G1垃圾收集器，接下来，我们一个个的了解学习。

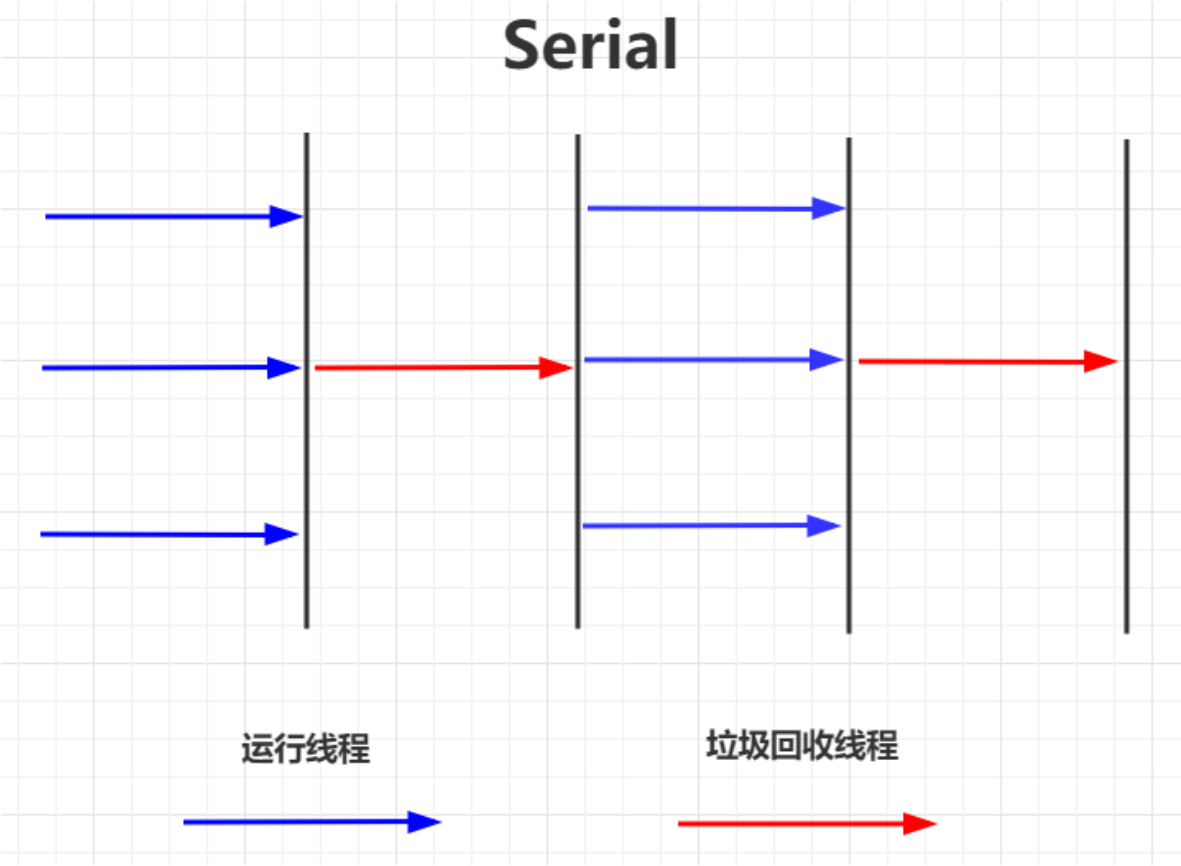


2.3.1 串行垃圾收集器Serial

2.3.1.1 概述

串行垃圾收集器，是指使用单线程进行垃圾回收，垃圾回收时，只有一个线程在工作，并且java应用中的所有线程都要暂停，等待垃圾回收的完成。这种现象称之为STW（Stop-The-World）。**其应用在年轻代**

对于交互性较强的应用而言，这种垃圾收集器是不能够接受的。因此一般在Javaweb应用中是不会采用该收集器的。



2.3.1.2 编写测试代码

```
public class SerialDemo {

    public static void main(String[] args) throws Exception {
        List<Object> list = new ArrayList<Object>();
        while (true){
            int sleep = new Random().nextInt(100);
            if(System.currentTimeMillis() % 2 ==0){
                list.clear();
            }else{
                for (int i = 0; i < 10000; i++) {
                    Properties properties = new Properties();
```

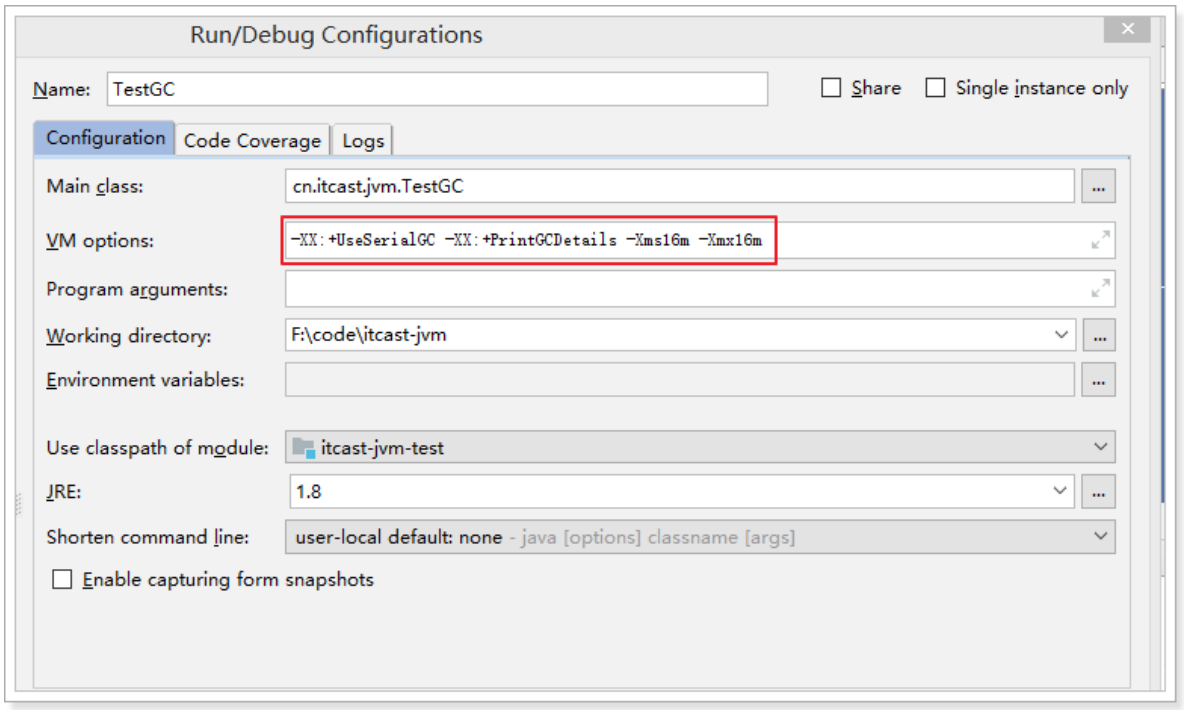
```
                    properties.put("key_"+i, "value_" +
System.currentTimeMillis() + i);
                    list.add(properties);
                }
            }

            Thread.sleep(sleep);
        }
    }
}
```

2.3.1.3 设置垃圾回收为串行收集器

在程序运行参数中添加2个参数，如下：

- **-XX:+UseSerialGC**
  - 指定年轻代和老年代都使用串行垃圾收集器
- **-XX:+PrintGCDetails**
  - 打印垃圾回收的详细信息

```
# 为了测试GC，将堆的初始和最大内存都设置为16M
-XX:+UseSerialGC -XX:+PrintGCDetails -Xms16m -Xmx16m
```



启动程序，可以看到下面信息：

```
[GC (Allocation Failure) [DefNew: 4416K->512K(4928K), 0.0046102 secs] 4416K-
>1973K(15872K), 0.0046533 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

[Full GC (Allocation Failure) [Tenured: 10944K->3107K(10944K), 0.0085637 secs]
15871K->3107K(15872K), [Metaspace: 3496K->3496K(1056768K)], 0.0085974 secs]
[Times: user=0.02 sys=0.00, real=0.01 secs]
```

GC日志信息解读：

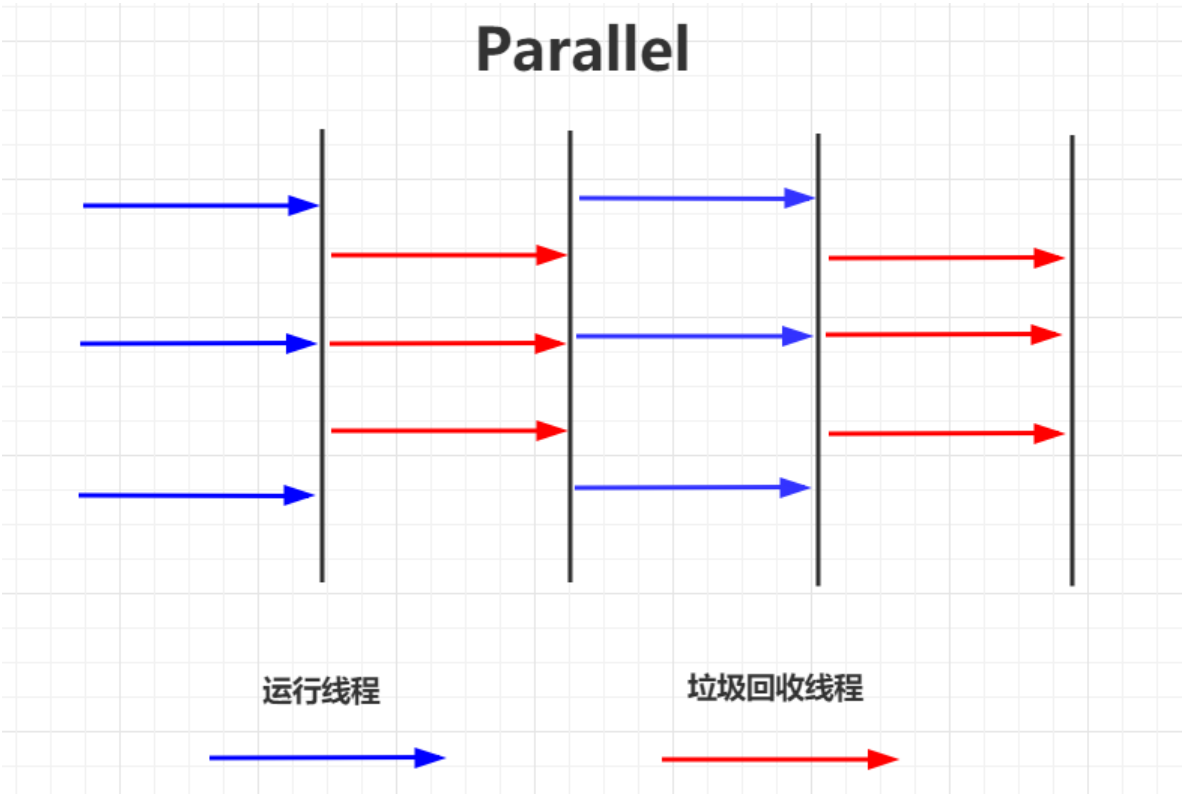
年轻代的内存GC前后的大小：

- DefNew
  - 表示使用的是串行垃圾收集器。
- 4416K->512K(4928K)
  - 表示，年轻代GC前，占有4416K内存，GC后，占有512K内存，总大小4928K
- 0.0046102 secs
  - 表示，GC所用的时间，单位为毫秒。
- 4416K->1973K(15872K)
  - 表示，GC前，堆内存占有4416K，GC后，占有1973K，总大小为15872K
- Full GC
  - 表示，内存空间全部进行GC

2.3.2 并行垃圾收集器Parallel

并行垃圾收集器在串行垃圾收集器的基础之上做了改进，将单线程改为了多线程进行垃圾回收，这样可以缩短垃圾回收的时间。（这里是指，并行能力较强的机器）。同样也是应用在年轻代。

当然了，并行垃圾收集器在收集的过程中也会暂停应用程序，这个和串行垃圾回收器是一样的，只是并行执行，速度更快些，暂停的时间更短一些。



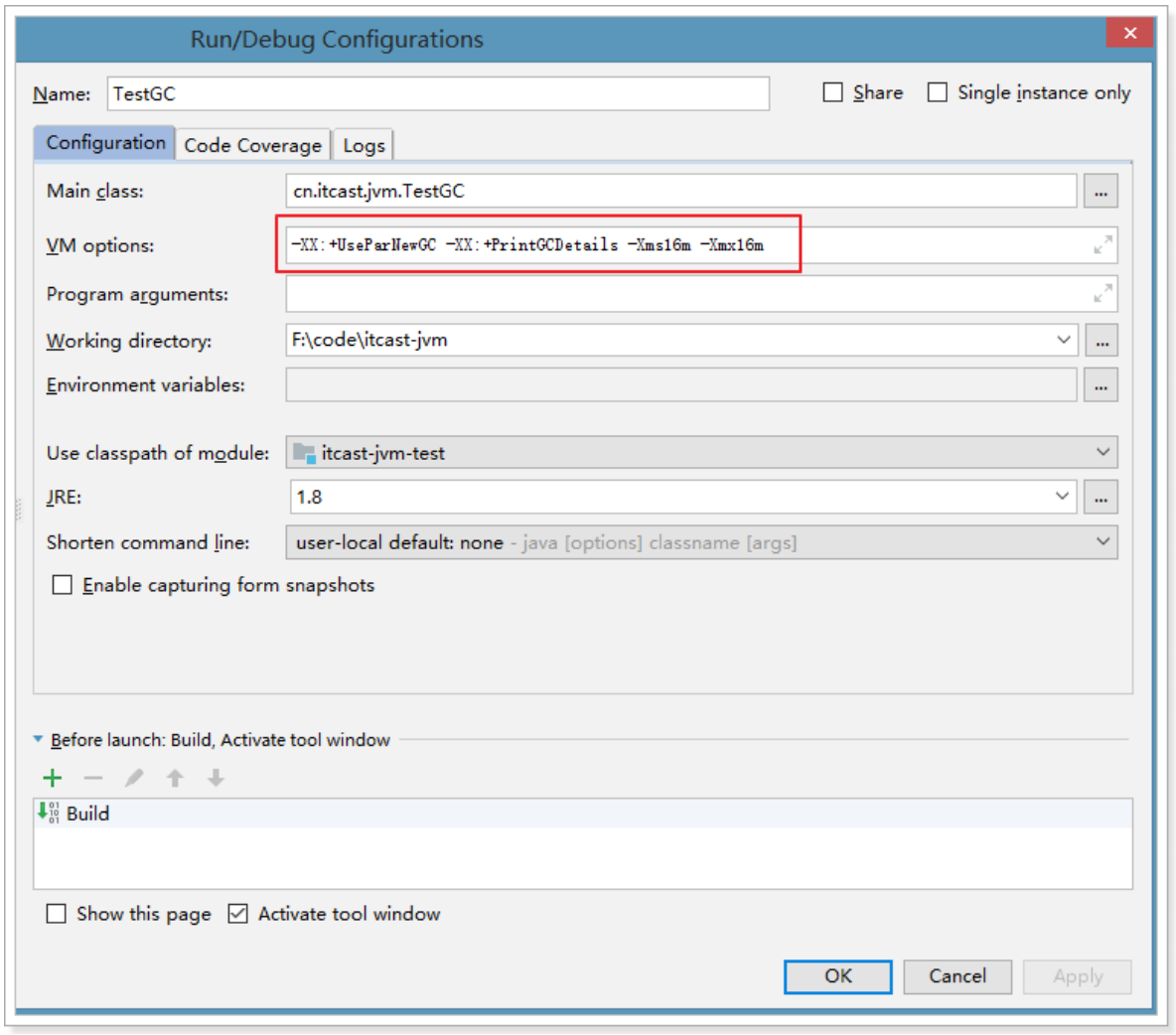
JDK8默认使用此垃圾回收器

2.3.2.1 ParNew垃圾收集器

ParNew垃圾收集器是工作在年轻代上的，只是将串行的垃圾收集器改为了并行。

通过**-XX:+UseParNewGC**参数设置年轻代使用ParNew回收器，老年代使用的依然是串行收集器。

测试：




```
#参数
-XX:+UseParNewGC -XX:+PrintGCDetails -Xms16m -Xmx16m

#打印出的信息
[GC (Allocation Failure) [ParNew: 4416K->512K(4928K), 0.0032106 secs] 4416K-
>1988K(15872K), 0.0032697 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
```

由以上信息可以看出，`ParNew:` 使用的是ParNew收集器。其他信息和串行收集器一致。

2.3.2.2 ParallelGC垃圾收集器（jdk8默认垃圾收集器）

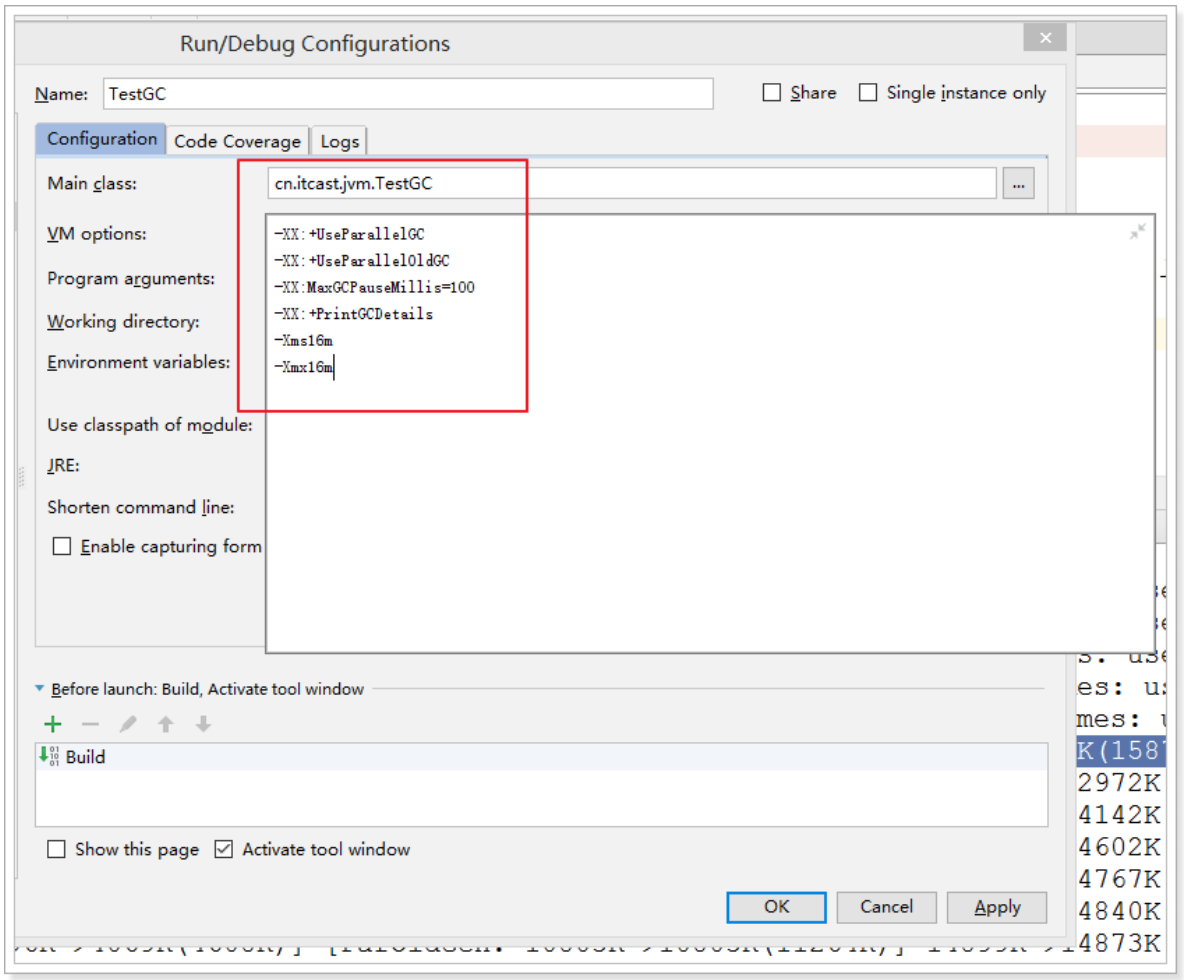
ParallelGC收集器工作机制和ParNewGC收集器一样，只是在此基础之上，新增了两个和系统吞吐量相关的参数，使得其使用起来更加的灵活和高效。

相关参数如下：

- **-XX:+UseParallelGC**
  - 年轻代使用ParallelGC垃圾回收器，老年代使用串行回收器。
- **-XX:+UseParallelOldGC**
  - 年轻代使用ParallelGC垃圾回收器，老年代使用ParallelOldGC垃圾回收器。
- **-XX:MaxGCPauseMillis(避免stm时间较长，会自动适当调小内存)**
  - 设置最大的垃圾收集时的停顿时间，单位为毫秒

    - 需要注意的时，ParallelGC为了达到设置的停顿时间，可能会调整堆大小或其他的参数，如果堆的大小设置的较小，就会导致GC工作变得很频繁，反而可能会影响到性能。
    - 该参数使用需谨慎。
- **-XX:UseAdaptiveSizePolicy**
    - 自适应GC模式，垃圾回收器将自动调整年轻代、老年代等参数，达到吞吐量、堆大小、停顿时间之间的平衡。
    - 一般用于，手动调整参数比较困难的场景，让收集器自动进行调整。

测试：




```
#参数
-XX:+UseParallelGC -XX:+UseParallelOldGC -XX:MaxGCPauseMillis=100 -
XX:+PrintGCDetails -Xms16m -Xmx16m

#打印的信息
[GC (Allocation Failure) [PSYoungGen: 4096K->480K(4608K)] 4096K->1840K(15872K),
0.0034307 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

[Full GC (Ergonomics) [PSYoungGen: 505K->0K(4608K)] [ParOldGen: 10332K-
>10751K(11264K)] 10837K->10751K(15872K), [Metaspace: 3491K->3491K(1056768K)],
0.0793622 secs] [Times: user=0.13 sys=0.00, real=0.08 secs]
```

有以上信息可以看出，年轻代和老年代都使用了ParallelGC垃圾回收器。

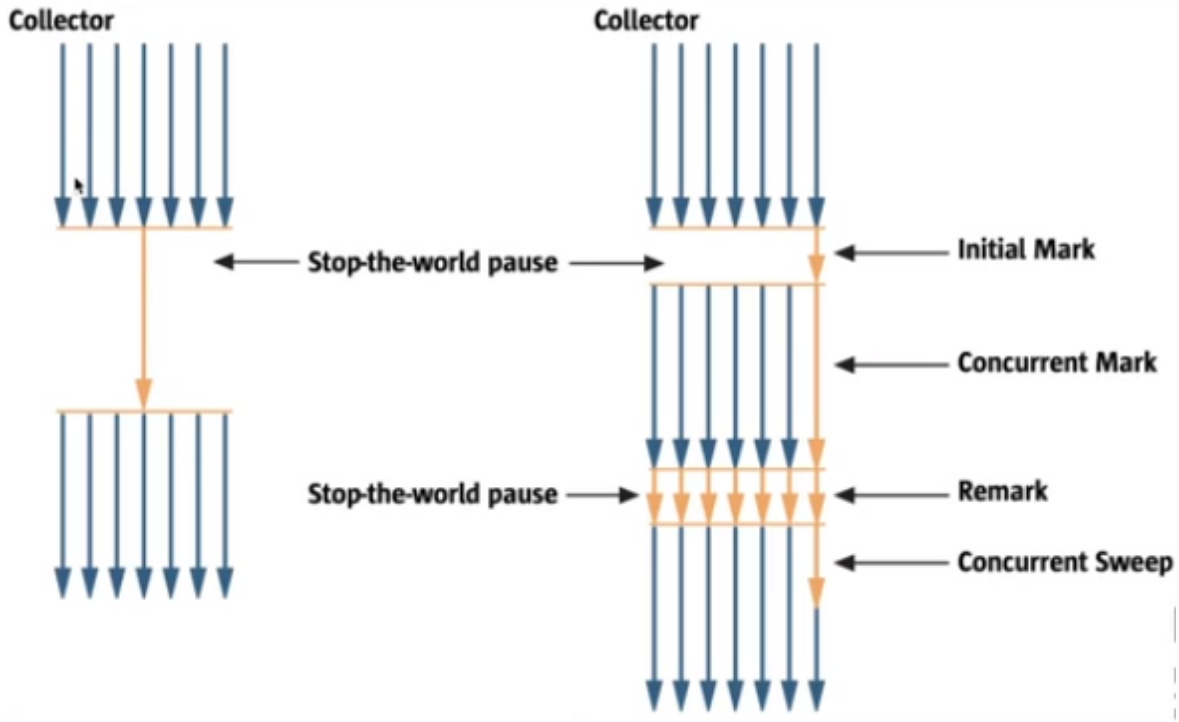
2.3.3 CMS垃圾收集器

2.3.3.1 概述

CMS全称 Concurrent Mark Sweep，是一款**并发**的、使用**标记-清除**算法的垃圾回收器，该回收器是**针对老年代垃圾回收的**，gc的时间能够在200ms内完成。通过参数**-XX:+UseConcMarkSweepGC**进行设置。

其最核心的特点是在进行垃圾回收时，应用仍然能正常运行。

CMS垃圾回收器的执行过程如下：



- 初始化标记(CMS-initial-mark) ,标记root，会导致stw；
- 并发标记(CMS-concurrent-mark)，与用户线程同时运行；
- 预清理（CMS-concurrent-preclean），与用户线程同时运行；
- 重新标记(CMS-remark)，会导致stw；
- 并发清除(CMS-concurrent-sweep)，与用户线程同时运行；
- 调整堆大小，设置CMS在清理之后进行内存压缩，目的是清理内存中的碎片；
- 并发重置状态等待下次CMS的触发(CMS-concurrent-reset)，与用户线程同时运行；

Initial Mark：初始标记。通过很短的STW进行初始标记。

Concurrent Mark：并行标记。用户线程运行的同时，进行垃圾的收集。

Remark：重新标记。进行STW，通过很多的线程进行重新标记。因为在ConcurrentMark阶段数据不一定是正确的，因为用户线程处于运行中。

Concurrent Sweep：并发清除。 通过和用户线程并发执行的线程，将垃圾清除掉。

2.3.3.2 测试

```
#设置启动参数
-XX:+UseConcMarkSweepGC -XX:+PrintGCDetails -Xms16m -Xmx16m

#运行日志
[GC (Allocation Failure) [ParNew: 4926K->512K(4928K), 0.0041843 secs] 9424K-
>6736K(15872K), 0.0042168 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
```

```
#第一步，初始标记
[GC (CMS Initial Mark) [1 CMS-initial-mark: 6224K(10944K)] 6824K(15872K),
0.0004209 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
#第二步，并发标记
[CMS-concurrent-mark-start]
[CMS-concurrent-mark: 0.002/0.002 secs] [Times: user=0.00 sys=0.00, real=0.00
secs]
#第三步，预处理
[CMS-concurrent-preclean-start]
[CMS-concurrent-preclean: 0.000/0.000 secs] [Times: user=0.00 sys=0.00,
real=0.00 secs]
#第四步，重新标记
[GC (CMS Final Remark) [YG occupancy: 1657 K (4928 K)][Rescan (parallel) ,
0.0005811 secs][weak refs processing, 0.0000136 secs][class unloading, 0.0003671
secs][scrub symbol table, 0.0006813 secs][scrub string table, 0.0001216 secs][1
CMS-remark: 6224K(10944K)] 7881K(15872K), 0.0018324 secs] [Times: user=0.00
sys=0.00, real=0.00 secs]
#第五步，并发清理
[CMS-concurrent-sweep-start]
[CMS-concurrent-sweep: 0.004/0.004 secs] [Times: user=0.00 sys=0.00, real=0.00
secs]
#第六步，重置
[CMS-concurrent-reset-start]
[CMS-concurrent-reset: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00
secs]
```

由以上日志信息，可以看出CMS执行的过程。

2.3.4 G1垃圾收集器（重点）

2.3.4.1 概述

对于垃圾回收器来说，前面的三种要么一次性回收年轻代，要么一次性回收老年代。而且现代服务器的堆空间已经可以很大了。为了更加优化GC操作，所以出现了G1。

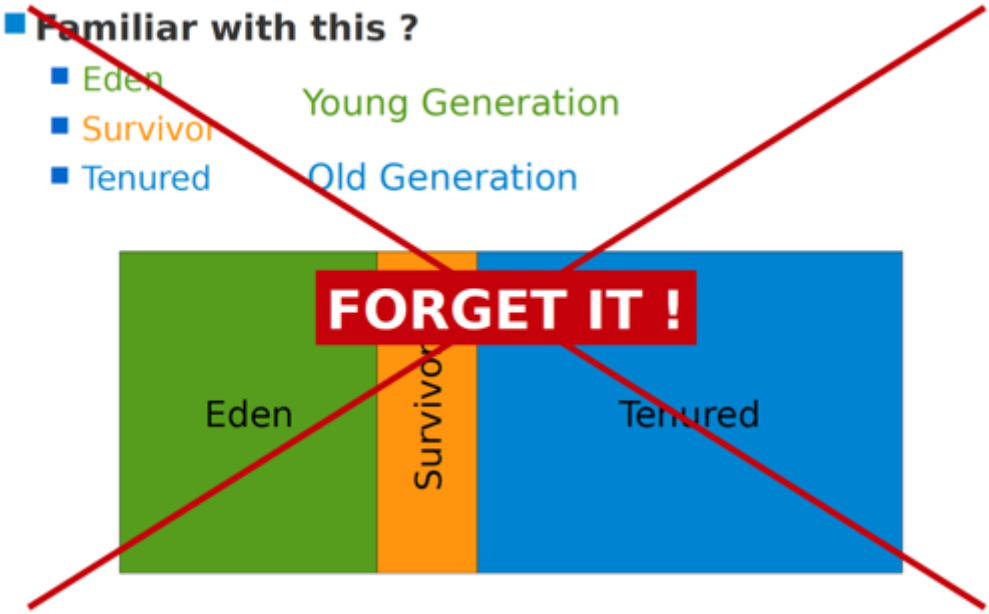
它是一款软实时、低延迟、可**设定目标(最大STW停顿时间)**的垃圾回收器，用于代替CMS，适用于较大的堆(>4~6G)，**在JDK9之后默认使用G1**。其垃圾回收的时间能控制在10ms内。

G1的设计原则就是简化JVM性能调优，开发人员只需要简单的三步即可完成调优：

1. 第一步，开启G1垃圾收集器
2. 第二步，设置堆的最大内存
3. 第三步，设置最大的停顿时间（stw）

2.3.4.2 G1的内存布局

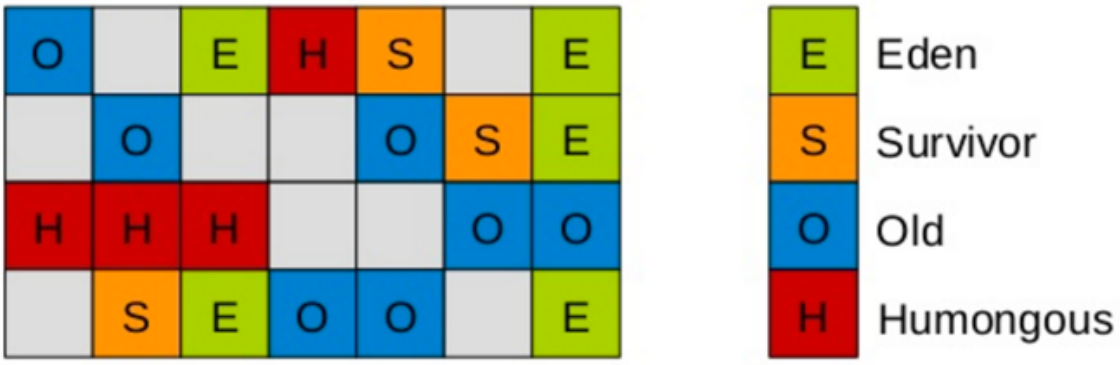
G1垃圾收集器相对比其他收集器而言，最大的区别在于它**取消了年轻代、老年代的物理划分。**



取而代之的是将堆划分为**若干个区域（Region）**，这些区域中包含了有**逻辑上的年轻代、老年代区域。**这样做的好处就是，我们再也不用单独的空间对每个代进行设置了，不用担心每个代内存是否足够。

此时可以看到，现在出现了一个**新的区域Humongous**，它本身属于老年代区。当现在出现了一个巨大的对象，超出了分区容量的一半，则这个对象会进入到该区域。如果一个H区装不下一个巨型对象，那么G1会寻找连续的H分区来存储。为了能找到连续的H区，有时候不得不启动Full GC。

同时G1会估计每个Region中的垃圾比例，优先回收垃圾较多的区域。



在G1划分的区域中，年轻代的垃圾收集依然采用**暂停所有应用线程的方式**，将存活对象拷贝到老年代或者Survivor空间，G1收集器通过将**对象从一个区域复制到另外一个区域，完成了清理工作。**

这就意味着，在正常的处理过程中，G1完成了堆的压缩（至少是部分堆的压缩），这样也就不会有cms内存碎片问题的存在了。

2.3.4.3 垃圾回收模式

其提供了三种模式垃圾回收模式： **young GC、Mixed GC、Full GC。**在不同的条件下被触发。

2.3.4.3.1 Young GC

发生在年轻代的GC算法，一般对象（除了巨型对象）都是在eden region中分配内存，当所有eden region被耗尽无法申请内存时，就会触发一次young gc，这种触发机制和之前的young gc差不多，执行完一次young gc，活跃对象会被拷贝到survivor region或者晋升到old region中，空闲的region会被放入空闲列表中，等待下次被使用。

2.3.4.3.2 Mixed GC

当越来越多的对象晋升到老年代old region时，为了避免堆内存被耗尽，虚拟机会触发一个混合的垃圾收集器，即**mixed gc**，该算法并不是一个old gc，除了回收整个young region，还会回收一部分的old region，这里需要注意：**是一部分老年代，而不是全部老年代**，可以选择哪些old region进行收集，从而可以对垃圾回收的耗时时间进行控制。

在CMS中，当老年代的使用率达到80%就会触发一次cms gc。在G1中，mixed gc也可以通过`-XX:InitiatingHeapOccupancyPercent`设置阈值，**默认为45%。**当老年代大小占整个堆大小百分比达到该阈值，则触发mixed gc。

其执行过程和cms类似：

1. initial mark: 初始标记过程，整个过程STW，标记了从GC Root可达的对象。
2. concurrent marking: 并发标记过程，整个过程gc collector线程与应用线程可以并行执行，标记出GC Root可达对象衍生出去的存活对象，并收集各个Region的存活对象信息。
3. remark: 最终标记过程，整个过程STW，标记出那些在并发标记过程中遗漏的，或者内部引用发生变化的对象。
4. clean up: 垃圾清除过程，如果发现一个Region中没有存活对象，则把该Region加入到空闲列表中。

2.3.4.3.3 Full GC

如果对象内存分配速度过快，mixed gc来不及回收，导致老年代被填满，就会触发一次full gc，G1的full gc算法就是单线程执行的serial old gc，会导致异常长时间的暂停时间，需要进行不断的调优，尽可能的避免full gc.

2.3.4.4 G1收集器相关参数

-XX:+UseG1GC	**使用 G1 垃圾收集器**
-XX:MaxGCPauseMillis=200	设置期望达到的最大GC停顿时间指标（JVM会尽力实现，但不保证达到）
-XX:InitiatingHeapOccupancyPercent=45	启动并发GC周期时的堆内存占用百分比. G1之类的垃圾收集器用它来触发并发GC周期,基于整个堆的使用率,而不只是某一代内存的使用比. 值为 0 则表示”一直执行GC循环”. 默认占用率是整个 Java 堆的 45%
-XX:NewRatio=n	新生代与老生代(new/old generation)的大小比例(Ratio). 默认值为 2.
-XX:SurvivorRatio=n	eden/survivor 空间大小的比例(Ratio). 默认值为 8.
-XX:MaxTenuringThreshold=n	提升年老代的最大临界值(tenuring threshold). 默认值为 15.
-XX:ParallelGCThreads=n	设置垃圾收集器在并行阶段使用的线程数,默认值随JVM运行的平台不同而不同.最多为8
-XX:ConcGCThreads=n	并发垃圾收集器使用的线程数量. 默认值随JVM运行的平台不同而不同.

-XX:+UseG1GC	**使用 G1 垃圾收集器** 使用G1时Java堆会被分为大小统一的的区
-XX:G1HeapRegionSize=n	(region)。此参数可以指定每个heap区的大小. 默认值将根据 heap size 算出最优解. 最小值为 1Mb, 最大值为 32Mb.

2.3.4.5 测试

```
-XX:+UseG1GC -XX:MaxGCPauseMillis=100 -XX:+PrintGCDetails -Xmx256m

#日志
[GC pause (G1 Evacuation Pause) (young), 0.0044882 secs]
   [Parallel Time: 3.7 ms, GC Workers: 3]
      [GC Worker Start (ms): Min: 14763.7, Avg: 14763.8, Max: 14763.8, Diff:
0.1]
      #扫描根节点
      [Ext Root Scanning (ms): Min: 0.2, Avg: 0.3, Max: 0.3, Diff: 0.1, Sum:
0.8]
      #更新RS区域所消耗的时间
      [Update RS (ms): Min: 1.8, Avg: 1.9, Max: 1.9, Diff: 0.2, Sum: 5.6]
         [Processed Buffers: Min: 1, Avg: 1.7, Max: 3, Diff: 2, Sum: 5]
      [Scan RS (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
      [Code Root Scanning (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum:
0.0]
      #对象拷贝
      [Object Copy (ms): Min: 1.1, Avg: 1.2, Max: 1.3, Diff: 0.2, Sum: 3.6]
      [Termination (ms): Min: 0.0, Avg: 0.1, Max: 0.2, Diff: 0.2, Sum: 0.2]
         [Termination Attempts: Min: 1, Avg: 1.0, Max: 1, Diff: 0, Sum: 3]
      [GC Worker Other (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
      [GC Worker Total (ms): Min: 3.4, Avg: 3.4, Max: 3.5, Diff: 0.1, Sum: 10.3]
      [GC Worker End (ms): Min: 14767.2, Avg: 14767.2, Max: 14767.3, Diff: 0.1]
   [Code Root Fixup: 0.0 ms]
   [Code Root Purge: 0.0 ms]
   [Clear CT: 0.0 ms] #清空CardTable
   [Other: 0.7 ms]
      [Choose CSet: 0.0 ms] #选取CSet
      [Ref Proc: 0.5 ms] #弱引用、软引用的处理耗时
      [Ref Enq: 0.0 ms] #弱引用、软引用的入队耗时
      [Redirty Cards: 0.0 ms]
      [Humongous Register: 0.0 ms] #大对象区域注册耗时
      [Humongous Reclaim: 0.0 ms] #大对象区域回收耗时
      [Free CSet: 0.0 ms]
   [Eden: 7168.0K(7168.0K)->0.0B(13.0M) Survivors: 2048.0K->2048.0K Heap:
55.5M(192.0M)->48.5M(192.0M)] #年轻代的大小统计
 [Times: user=0.00 sys=0.00, real=0.00 secs]
```

2.3.4.5 G1的最佳实践

不断调优暂停时间指标

通过**XX:MaxGCPauseMillis=x**可以设置启动应用程序暂停的时间，G1在运行的时候会根据这个参数选择CSet来满足响应时间的设置。一般情况下这个值设置到100ms或者200ms都是可以的(不同情况下会不一样)，但如果设置成50ms就不太合理。暂停时间设置的太短，就会导致出现G1跟不上垃圾产生的速度。最终退化成Full GC。所以对这个参数的调优是一个持续的过程，逐步调整到最佳状态。

不要设置新生代和老年代的大小

G1收集器在运行的时候会调整新生代和老年代的大小。通过改变代的大小来调整对象晋升的速度以及晋升年龄，从而达到我们为收集器设置的暂停时间目标。设置了新生代大小相当于放弃了G1为我们做的自动调优。我们需要做的只是设置整个堆内存的大小，剩下的交给G1自己去分配各个代的大小。

2.4 可视化GC日志分析工具-GC Easy

设置参数输出gc日志：-XX:+UseParallelGC -XX:+UseParallelOldGC -XX:+PrintGCDetails -Xms128m -Xmx128m -Xloggc:gc.log

GC Easy是一款在线的可视化工具，易用、功能强大，网站：

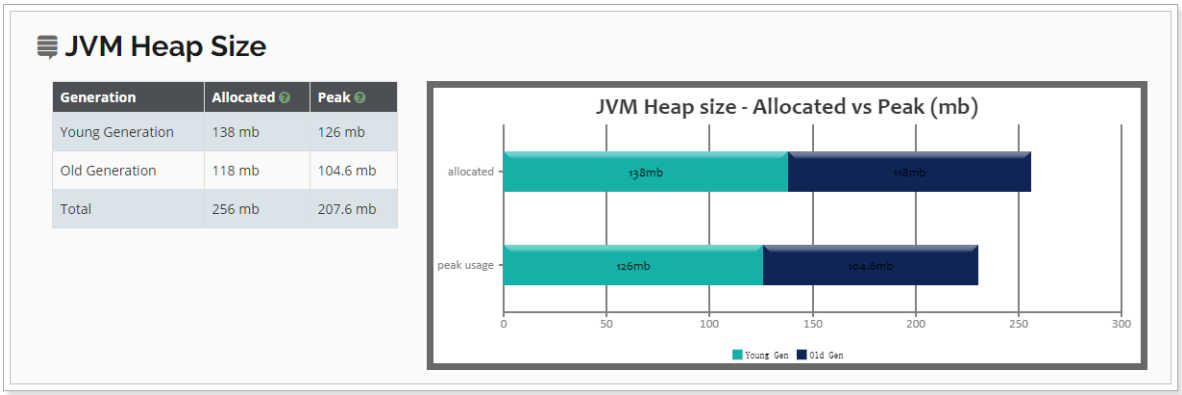
http://gceasy.io/

上传后，点击“Analyze”按钮，即可查看报告。

JVM Heap Size

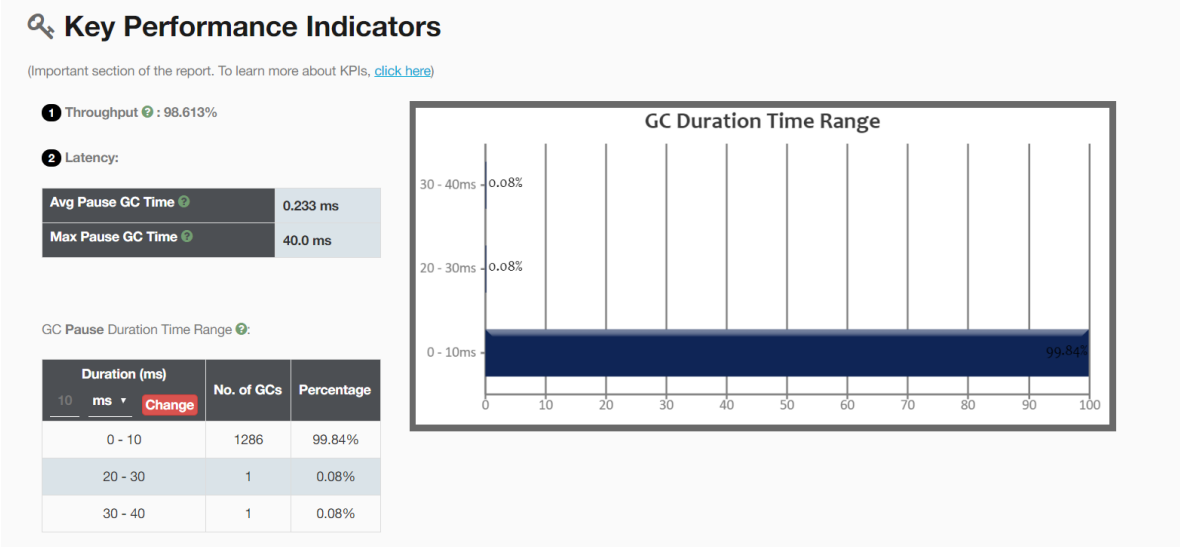
这一部分分别使用了表格和图形界面来展示了JVM堆内存大小

左侧分别展示了年轻代的内存分配分配空间大小（Allocated）和年轻代内存分配空间大小的最大峰值（Peek），然后依次是老年代（Old Generation）、元数据区（Meta Space）、堆区和非堆区（Young + Old + Meta Space）总大小。



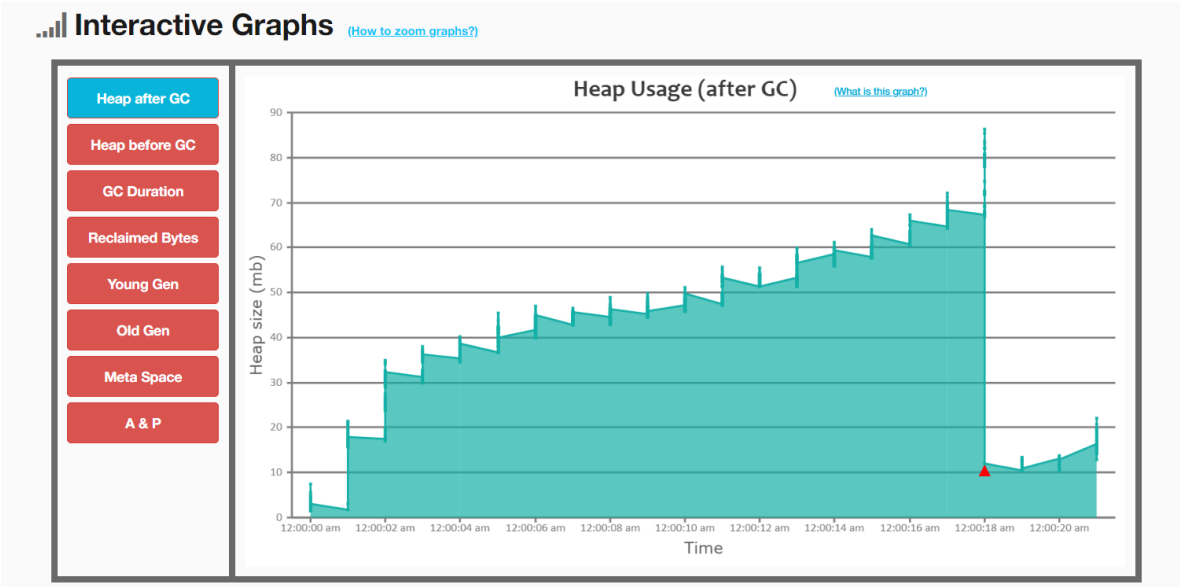
Key Performance Indicators

这一部分是关键的性能指标



- Throughput表示的是吞吐量
- Latency表示响应时间
    - Avg Pause GC Time 平均GC时间
    - Max Pause GC TIme 最大GC时间

Interactive Graphs



第一部分为：回收后堆的内存图，从图中可以看出，随着GC的进行，垃圾回收器把对象都回收掉了，因此堆的大小主键增大。

第二部分为：回收前堆的使用率，随着程序的运行，堆的使用率越来越高，堆被对象占用的内存越来越大。

第三部分为：GC的持续时间。

第四部分为：GC回收掉的垃圾对象的大小。

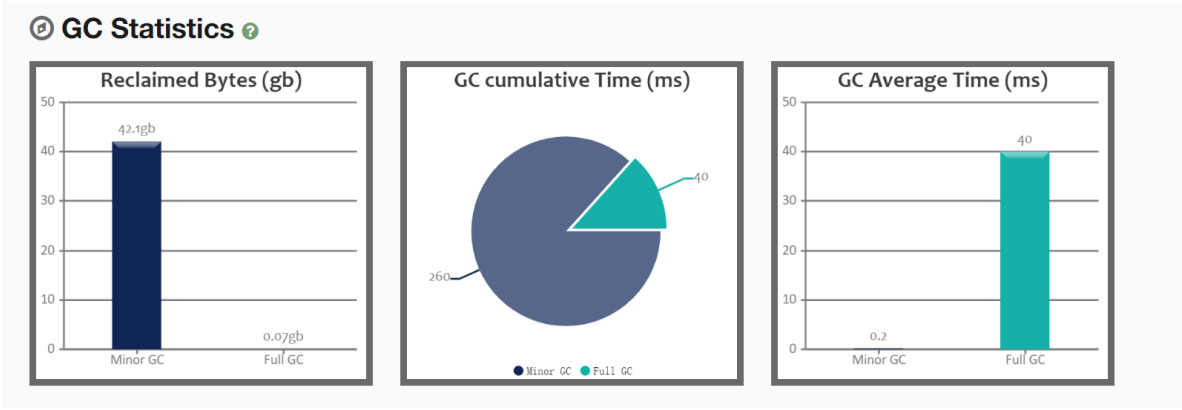
第五部分为：年轻代的内存分配情况

第六部分为：老年代的内存分配情况

第七部分为：元数据区内存分配情况

第八部分为：堆内存分配和晋升情况

GC Statistics



左图：表示的是堆内存中Minor GC和Full GC回收垃圾对象的内存。
中图：总计GC时间，包括Minor GC和Full GC，时间单位为ms。
右图：GC平均时间，包括了Minor GC和Full GC。

其他

Total GC stats

Total GC count	1288
Total reclaimed bytes	42.18 gb
Total GC time	300 ms
Avg GC time	0.233 ms
GC avg time std dev	1.92 ms
GC min/max time	0 / 40.0 ms
GC Interval avg time	16.0 ms

Minor GC stats

Minor GC count	1287
Minor GC reclaimed	42.1 gb
Minor GC total time	260 ms
Minor GC avg time	0.202 ms
Minor GC avg time std dev	1.56 ms
Minor GC min/max time	0 / 30.0 ms
Minor GC Interval avg	16.0 ms

Full GC stats

Full GC Count	1
Full GC reclaimed	75.74 mb
Full GC total time	40.0 ms
Full GC avg time	40.0 ms
Full GC avg time std dev	0
Full GC min/max time	40.0 ms / 40.0 ms
Full GC Interval avg	n/a

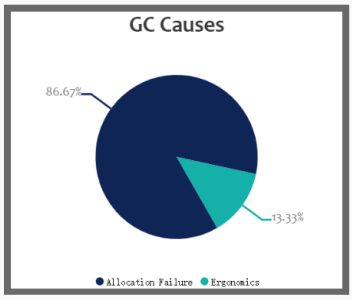
GC Pause Statistics

Pause Count	1288
Pause total time	300 ms
Pause avg time	0.233 ms
Pause avg time std dev	0.0
Pause min/max time	0 / 40.0 ms

分别表示的是总GC统计，MinorGC的统计，FullGC的统计，GC暂停程序的统计。

GC Causes

Cause	Count	Avg Time	Max Time	Total Time	Time %
Allocation Failure	1287	0.202 ms	30.0 ms	260 ms	86.67%
Ergonomics	1	40.0 ms	40.0 ms	40.0 ms	13.33%
Total	1288	n/a	n/a	300 ms	100.0%



GC花费的时间统计