

课程主题

运行时数据区之方法区和字符串常量池

课程回顾

课程目标

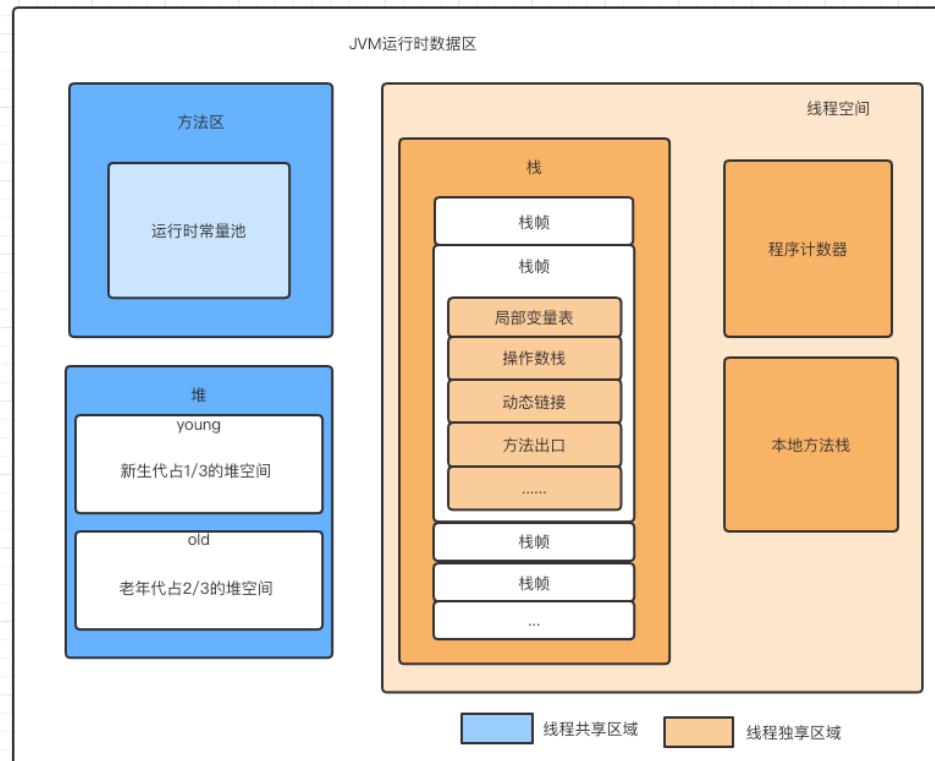
课程内容

一、运行时数据区概述

整个JVM构成里面，由三部分组成：类加载系统、运行时数据区、执行引擎

1、JVM运行时数据区规范

<https://www.processon.com/diagraming/60a390dde401fd399500f846>



按照线程使用情况和职责分成两大类：

- 线程独享（程序执行区域）
 - 不需要垃圾回收
 - 虚拟机栈、本地方法栈、程序计数器
- 线程共享（数据存储区域）
 - 垃圾回收
 - 存储类的静态数据和对象数据

2、分配JVM内存空间

分配堆的大小

`-Xms` (堆的初始容量)
`-Xmx` (堆的最大容量)

`-XX:InitialHeapSize=268435456`
`-XX:MaxHeapSize=4294967296`

如果为了提高性能，可以考虑去浪费空间，就是将初始容量和最大容量相等

开课吧

分配方法区的大小

`-XX:PermSize`

永久代的初始容量

`-XX:MaxPermSize`

永久代的最大容量

-XX:MetaspaceSize

元空间的初始大小，达到该值就会触发垃圾收集进行类型卸载，同时GC会对该值进行调整：如果释放了大量的空间，就适当降低该值；如果释放了很少的空间，那么在不超过MaxMetaspaceSize时，适当提高该值。

-XX:MaxMetaspaceSize

最大空间，默认是没有限制的。

除了上面两个指定大小的选项以外，还有两个与 GC 相关的属性：

-XX:MinMetaspaceFreeRatio

在GC之后，最小的Metaspace剩余空间容量的百分比，减少为分配空间所导致的垃圾收集

-XX:MaxMetaspaceFreeRatio

在GC之后，最大的Metaspace剩余空间容量的百分比，减少为释放空间所导致的垃圾收集

分配线程空间的大小

-Xss:

为jvm启动的每个线程分配的内存大小，默认JDK1.4中是256K，JDK1.5+中是1M

二、方法区

1、方法区存储什么数据

类型信息，比如Class (com.kkb.User类)

方法信息，比如Method (方法名称、方法参数列表、方法返回值信息)

字段信息，比如Field (字段类型，字段名称需要特殊设置才能保存的住)

Code区，存储的是方法执行对应的字节码指令

方法表（方法调用的时候） 在A类的main方法中去调用B类的method1方法，是根据B类的方法表去查找合适的方法，进行调用的。

静态变量（类变量） ---JDK1.7之后，转移到堆中存储

运行时常量池（字符串常量池） ---从class中的常量池加载而来---JDK1.7之后，转移到堆中存储

- * 字面量类型
 - * 双引号引起的字符串值，比如"kkb" ----- 会进入字符串常量池 (StringPool)
 - * final修饰的变量
 - * 非final修饰的变量，比如long、double、float
- * 引用类型-->内存地址
 - * 类的符号引用
 - * 方法
 - * 字段

JIT编译器编译之后的代码缓存

如果需要访问方法区中类的其他信息，都必须先获得Class对象，才能取访问该Class对象关联的方法信息或者字段信息。

存储示意图如下，下面的图片显示的是JVM加载类的时候，方法区存储的信息：



1.1、类型信息（重点）

- 类型的全限定名
- 超类的全限定名
- 直接超接口的全限定名
- 类型标志（该类是类类型还是接口类型）
- 类的访问描述符（public、private、default、abstract、final、static）

1.2、类型的常量池

存放该类型所用到的常量的有序集合，包括直接常量（如字符串、整数、浮点数的常量）和其他类型、字段、方法的符号引用。

常量池中每一个保存的常量都有一个索引，就像数组中的字段一样。因为常量池中保存着所有类型使用到的类型、字段、方法的字符引用，所以它也是动态连接的主要对象（在动态链接中起到核心作用）。

1.3、字段信息（重点）

- 字段修饰符（public、protect、private、default）
- 字段的类型
- 字段名称

1.4、方法信息（重点）

方法信息中包含类的所有方法，每个方法包含以下信息：

- 方法修饰符
- 方法返回类型
- 方法名
- 方法参数个数、类型、顺序等
- 方法字节码
- 操作数栈和该方法在栈帧中的局部变量区大小
- 异常表

1.5、类变量（重点）

指该类所有对象共享的变量，即使没有任何实例对象时，也可以访问的类变量。它们与类进行绑定。

1.6、指向类加载器的引用

每一个被JVM加载的类型，都保存这个类加载器的引用，类加载器动态链接时会用到。

1.7、指向Class实例的引用

类加载的过程中，虚拟机会创建该类型的Class实例，方法区中必须保存对该对象的引用。通过Class.forName(String className)来查找获得该实例的引用，然后创建该类的对象。

1.8、方法表（重点）

为了提高访问效率，JVM可能会对每个装载的非抽象类，都创建一个数组，数组的每个元素是实例可能调用的方法的直接引用，包括父类中继承过来的方法。这个表在抽象类或者接口中是没有的。

1.9、运行时常量池

(Runtime Constant Pool)

开课吧

class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池，用于存放编译器生成的各种字面常量和符号引用，这部分内容被类加载后进入方法区的运行时常量池中存放。

运行时常量池相对于class文件常量池的另外一个特征具有动态性，可以在运行期间将新的常量放入池中（典型的如String类的intern()方法）。

2、永久代和元空间的区别是什么？

1. JDK1.8之前使用的方法区实现是永久代，JDK1.8及以后使用的方法区实现是元空间。
2. 存储位置不同，永久代所使用的内存区域是JVM进程所使用的区域，它的大小受整个JVM的大小所限制。元空间所使用的内存区域是物理内存区域。那么元空间的使用大小只会受物理内存大小的限制。
3. 存储内容不同，永久代存储的信息基本上就是上面方法区存储内容中的数据。元空间只存储类的元信息，而静态变量和运行时常量池都挪到堆中。

3、为什么要使用元空间来替换永久代？

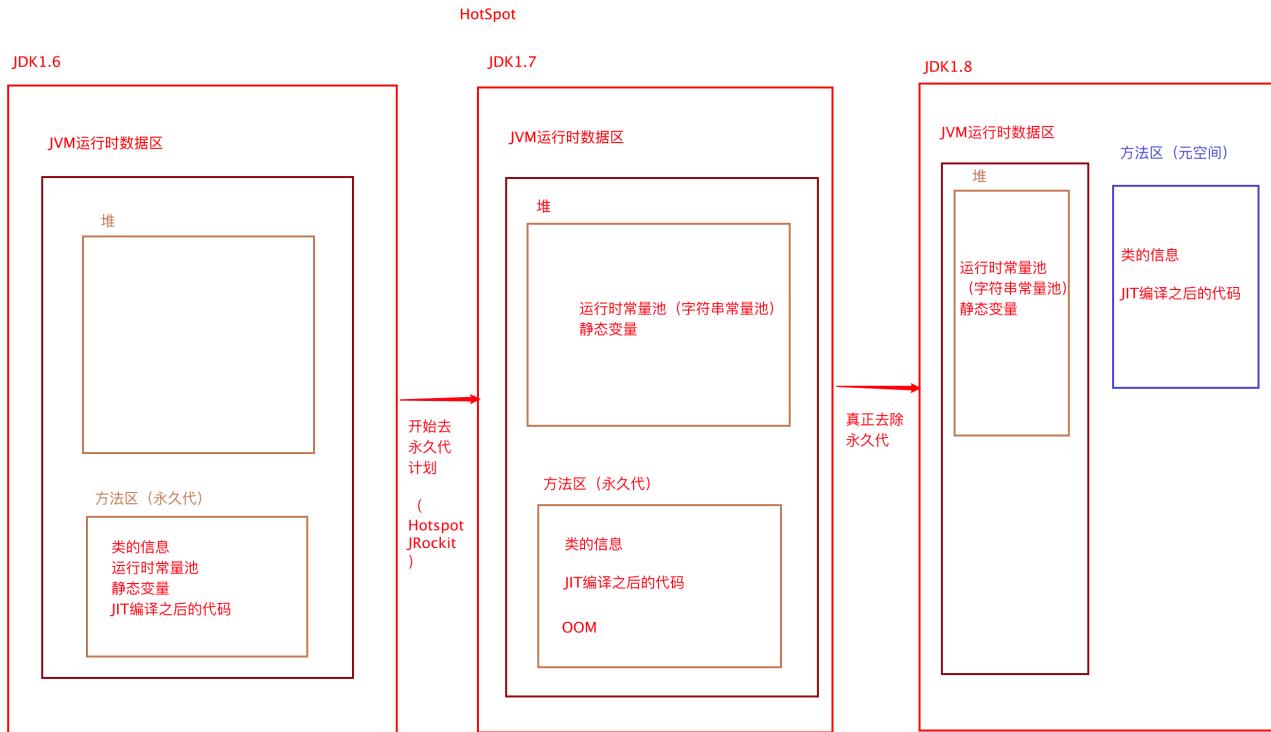
1. 字符串存在永久代中，容易出现性能问题和永久代内存溢出。
2. 类及方法的信息等比较难确定其大小，因此对于永久代的大小指定比较困难，太小容易出现永久代溢出，太大则容易导致老年代溢出。
3. 永久代会为GC带来不必要的复杂度，并且回收效率偏低。
4. Oracle计划将HotSpot与JRockit合二为一。

结论

其实，移除永久代的工作从JDK1.7就开始了。

JDK1.7中，存储在永久代的部分数据就已经转移到了Java Heap。

但永久代仍存在于JDK1.7中，并没完全移除，譬如字面量(interned strings)转移到了java heap；类的静态变量(class statics)转移到了java heap。



4、方法区异常演示

4.1、类加载导致OOM异常

1) 案例代码

我们现在通过动态生成类来模拟方法区的内存溢出：

```
package com.kkb.test.memory;
public class Test {}
```

```
package com.kkb.test.memory;
import java.io.File;
import java.net.URL;
import java.net.URLClassLoader;
import java.util.ArrayList;
```

```
import java.util.List;
public class PermGenOomMock{
    public static void main(String[] args) {
        URL url = null;
        List<ClassLoader> classLoaderList = new
ArrayList<ClassLoader>();
        try {
            url = new File("/tmp").toURI().toURL();

            URL[] urls = {url};
            while (true){
                ClassLoader loader = new
URLClassLoader(urls);
                classLoaderList.add(loader);
                loader.loadClass("com.kkb.test.memory.Test");
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

2) JDK1.7分析

指定的 `PermGen` 区的大小为 8M:

```
liuxpdeMacBook-Pro:classes liuxp$ java -version
java version "1.7.0_75"
Java(TM) SE Runtime Environment (build 1.7.0_75-b13)
Java HotSpot(TM) 64-Bit Server VM (build 24.75-b04, mixed mode)
liuxpdeMacBook-Pro:classes liuxp$ java -XX:PermSize=8m -XX:MaxPermSize=8m com.test.memory.PermGenOomMock
Exception in thread "main" java.lang.OutOfMemoryError: PermGen space
        at java.lang.ClassLoader.defineClass(ClassLoader.java:800)
        at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142)
        at java.net.URLClassLoader.defineClass(URLClassLoader.java:449)
        at java.net.URLClassLoader.access$100(URLClassLoader.java:71)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:361)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:355)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:354)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:425)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:358)
```

最典型的场景就是，在jsp页面比较多的情况下，容易出现永久代内存溢出。

3) JDK1.8+分析

现在我们在JDK 8下重新运行一下案例代码，不过这次不再指定 `PermSize` 和 `MaxPermSize`。而是指定 `MetaspaceSize` 和 `MaxMetaspaceSize` 的大小。输出结果如下：

```
liuxpdeMacBook-Pro:classes liuxp$ java -version
java version "1.8.0_40"
Java(TM) SE Runtime Environment (build 1.8.0_40-b27)
Java HotSpot(TM) 64-Bit Server VM (build 25.40-b25, mixed mode)
liuxpdeMacBook-Pro:classes liuxp$ java -XX:MetaspaceSize=8m -XX:MaxMetaspaceSize=8m com.paddx.test.memory.PermGenOomMock
Exception in thread "main" java.lang.OutOfMemoryError: Metaspace
        at java.lang.ClassLoader.defineClass1(Native Method)
        at java.lang.ClassLoader.defineClass(ClassLoader.java:760)
        at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142)
        at java.net.URLClassLoader.defineClass(URLClassLoader.java:467)
        at java.net.URLClassLoader.access$100(URLClassLoader.java:73)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:368)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:362)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:361)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
```

从输出结果，我们可以看出，这次不再出现永久代溢出，而是出现了元空间的溢出。

4.2、字符串OOM异常

1) 案例代码

以下这段程序以2的指数级不断的生成新的字符串，这样可以比较快速的消耗内存：

```
package com.kkb.test.memory;
import java.util.ArrayList;
import java.util.List;
public class StringOomMock {
    static String base = "string";
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        for (int i=0;i< Integer.MAX_VALUE;i++){
            String str = base + base;
            base = str;
            list.add(str.intern());
        }
    }
}
```

2) JDK1.6

JDK 1.6 的运行结果：

```
LiuxpdeMacBook-Pro:classes liuxp$ java -version
java version "1.6.0_65"
Java(TM) SE Runtime Environment (build 1.6.0_65-b14-466.1-11M4716)
Java HotSpot(TM) 64-Bit Server VM (build 20.65-b04-466.1, mixed mode)
LiuxpdeMacBook-Pro:classes liuxp$ java -XX:PermSize=8m -XX:MaxPermSize=8m -Xmx16m com.paddx.test.memory.StringOomMock
Exception in thread "main" java.lang.OutOfMemoryError: PermGen space
    at java.lang.String.intern(Native Method)
    at com.paddx.test.memory.StringOomMock.main(StringOomMock.java:17)
```

在JDK 1.6下，会出现永久代的内存溢出。

3) JDK1.7

JDK 1.7的运行结果：

```
luxpdeMacBook-Pro:classes liuxp$ java -version
java version "1.7.0_75"
Java(TM) SE Runtime Environment (build 1.7.0_75-b13)
Java HotSpot(TM) 64-Bit Server VM (build 24.75-b04, mixed mode)
luxpdeMacBook-Pro:classes liuxp$ java -XX:PermSize=8m -XX:MaxPermSize=8m -Xmx16m com.paddx.test.memory.StringOomMock
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at java.util.Arrays.copyOf((Arrays.java:2367)
        at java.lang.AbstractStringBuilder.expandCapacity(AbstractStringBuilder.java:130)
        at java.lang.AbstractStringBuilder.ensureCapacityInternal(AbstractStringBuilder.java:114)
        at java.lang.AbstractStringBuilder.append(AbstractStringBuilder.java:415)
        at java.lang.StringBuilder.append(StringBuilder.java:132)
        at com.paddx.test.memory.StringOomMock.main(StringOomMock.java:15)
```

在JDK 1.7中，会出现堆内存溢出。结论是：JDK 1.7 已经将字符串常量由永久代转移到堆中。

4) JDK1.8+

JDK 1.8的运行结果：

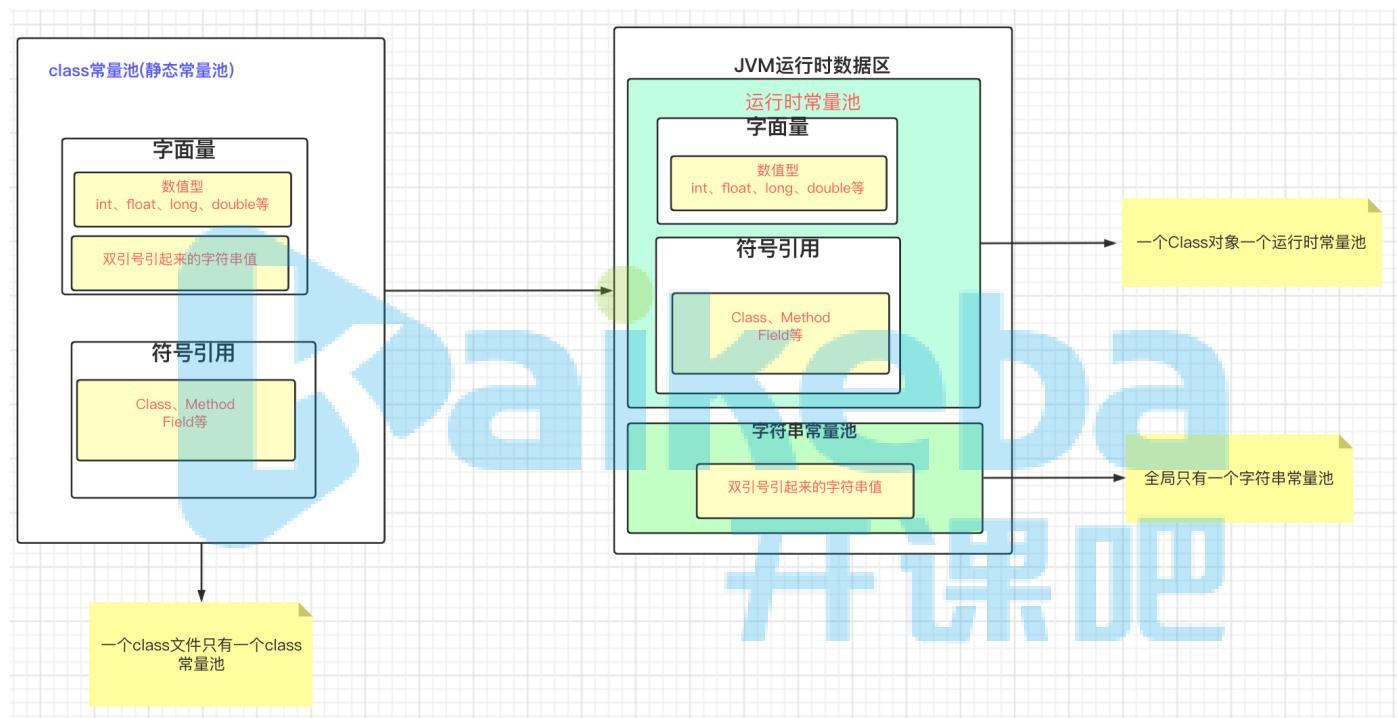
```
luxpdeMacBook-Pro:classes liuxp$ java -version
java version "1.8.0_40"
Java(TM) SE Runtime Environment (build 1.8.0_40-b27)
Java HotSpot(TM) 64-Bit Server VM (build 25.40-b25, mixed mode)
luxpdeMacBook-Pro:classes liuxp$ java -XX:PermSize=8m -XX:MaxPermSize=8m -Xmx16m com.paddx.test.memory.StringOomMock
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option PermSize=8m; support was removed in 8.0
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=8m; support was removed in 8.0
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at java.util.Arrays.copyOf((Arrays.java:3332)
        at java.lang.AbstractStringBuilder.expandCapacity(AbstractStringBuilder.java:137)
        at java.lang.AbstractStringBuilder.ensureCapacityInternal(AbstractStringBuilder.java:121)
        at java.lang.AbstractStringBuilder.append(AbstractStringBuilder.java:421)
        at java.lang.StringBuilder.append(StringBuilder.java:136)
        at com.paddx.test.memory.StringOomMock.main(StringOomMock.java:15)
```

在JDK 1.8 中，也会出现堆内存溢出，并且显示JDK 1.8中 PermSize 和 MaxPermGen 已经无效。因此，可以验证 JDK 1.8 中已经不存在永久代的结论。

三、字符串常量池

1、三种常量池区别

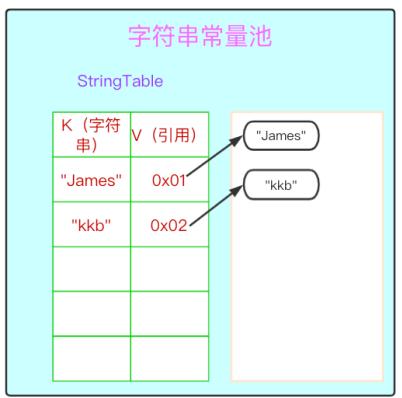
<https://www.processon.com/diagraming/60bc5453e0b34d09509f819b>



2、字符串常量池中如何存储数据

<https://www.processon.com/diagraming/60bc5453e0b34d09509f819b>

堆空间 (JDK1.7+)



字符串常量池查找字符串的方式：

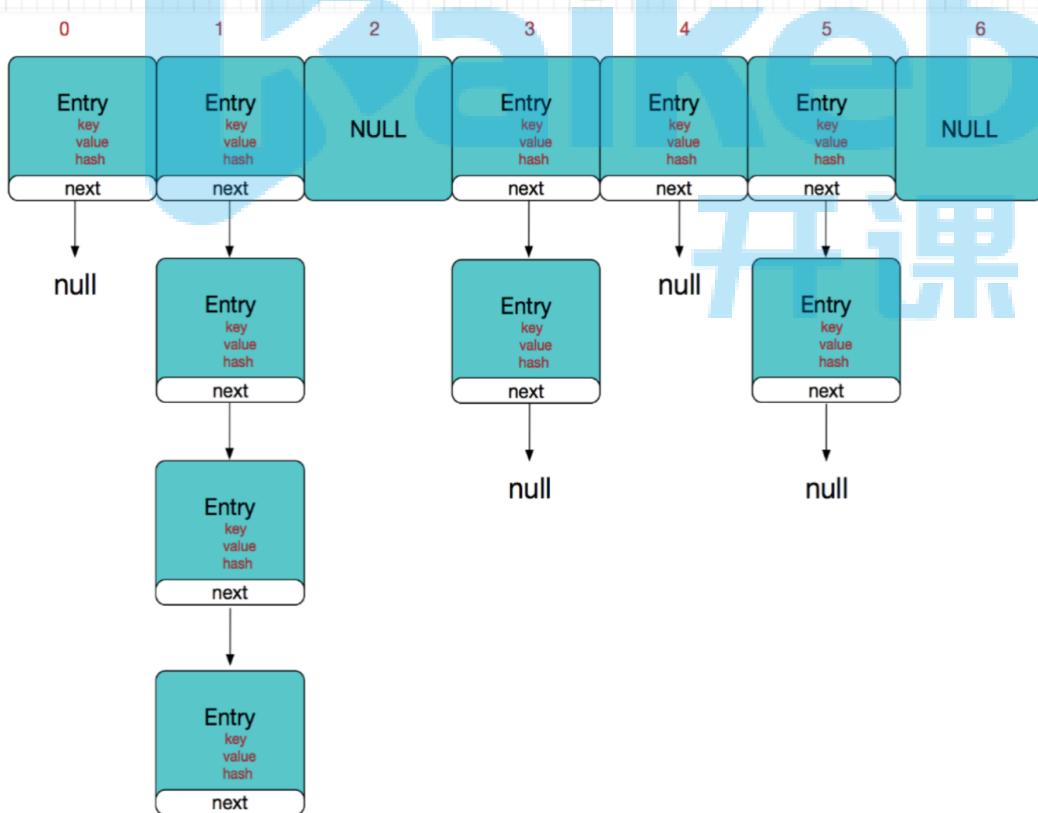
- 根据字符串的 hashCode 找到对应 entry。
- 如果没冲突，它可能只是一个 entry。
- 如果有冲突，它可能是一个 entry 链表，然后 Java 再遍历 entry 链表，匹配引用对应的字符串。
- 如果找不到字符串，在使用 intern 方法的时候，会将 intern 方法调用者的引用放入 stringtable 中。

为了提高匹配速度，即更快的查找某个字符串是否存在于常量池，Java 在设计字符串常量池的时候，还搞了一张 [stringtable]，`stringtable` 有点类似于我们的 hashtable，里面保存了 [字符串的引用]。

{ hashtable 中数据存储到数组中的位置是通过以下计算得来的：
hash(字符串值) / 数组长度 = 余数 (数组下标)
数组中存储的元素都是一个 Entry 对象 (next 指针)
Entry 通过 next 指针可以形成链表。 }

> hashtable 会存在两种问题：hash 冲突问题、rehash 问题
> hash 冲突问题：链地址法，也就是使用链表
> 一旦导致 hash 冲突之后，就会形成链表。
> 链表是增删快，查找慢。

在 jdk7+，`StringTable` 的长度可以通过一个参数指定：
-XX:StringTableSize=99991



3、字符串常量池案例分析

```
public class Test {  
    public void test() {  
        String str1 = "abc";  
        String str2 = new String("abc");  
        System.out.println(str1 == str2);  
  
        String str3 = new String("abc");  
        System.out.println(str3 == str2);  
  
        String str4 = "a" + "b";  
        System.out.println(str4 == "ab");  
  
        final String s = "a";  
        String str5 = s + "b";  
        System.out.println(str5 == "ab");  
  
        String s1 = "a";  
        String s2 = "b";  
        String str6 = s1 + s2;  
        System.out.println(str6 == "ab");  
  
        String str7 = "abc".substring(0, 2);  
        System.out.println(str7 == "ab");  
  
        String str8 = "abc".toUpperCase();  
        System.out.println(str8 == "ABC");  
  
        String s5 = "a";  
        String s6 = "abc";
```

```
String s7 = s5 + "bc";
System.out.println(s6 == s7.intern());
}
}
```

结论：

- 1、单独使用""引号创建的字符串都是常量，编译期就已经确定存储到String Pool中。
- 2、使用new String("")创建的对象会存储到heap中，是运行期新创建的。
- 3、使用只包含常量的字符串连接符如"aa"+“bb”创建的也是常量，编译期就能确定已经存储到String Pool中。
- 4、使用包含变量的字符串连接如"aa"+s创建的对象是运行期才创建的，存储到heap中。
- 5、运行期调用String的intern()方法可以向String Pool中动态添加对象。

4、String的Intern方法详解

4.1、intern的作用

1. 返回stringtable中对应字符串对象的引用值。
2. 如果stringtable中没有对应字符串对象的一条记录，则动态添加字符串对象到stringtable中。

先让大家做个面试题：

```
String c = "world";
System.out.println(c.intern() == c);

String d = new String("mike");
System.out.println(d.intern() == d);

String e = new String("jo") + new String("hn");
System.out.println(e.intern() == e);

String f = new String("ja") + new String("va");
System.out.println(f.intern() == f);
```

JDK1.7+测试的结果：

```
public class TestIntern {
    public static void main(String[] args) {
        // String f = new String("abs") + new String("tract");
        //t
        // String f = new String("br") + new String("eak"); //t
        // String f = new String("cat") + new String("ch"); //t
        // String f = new String("cla") + new String("ss"); //t
        // String f = new String("con") + new String("tinue");
        //t
        // String f = new String("d") + new String("o"); //t
        // String f = new String("el") + new String("se"); //t
        // String f = new String("ex") + new String("tends");
        //t
        // String f = new String("fin") + new String("al"); //t
```

```
//      String f = new String("fin") + new String("ally");
//t
//      String f = new String("f") + new String("or"); //t
//      String f = new String("i") + new String("f"); //t
//      String f = new String("imp") + new
String("lements"); //t
//      String f = new String("im") + new String("port");
//t
//      String f = new String("instance") + new
String("of"); //t
//      String f = new String("inter") + new String("face");
//t
//      String f = new String("na") + new String("tive");
//t
//      String f = new String("n") + new String("ew"); //t
//      String f = new String("pack") + new String("age");
//t
//      String f = new String("pri") + new String("vate");
//t
//      String f = new String("protect") + new String("ed");
//t
//      String f = new String("pub") + new String("lic");
//t
//      String f = new String("sta") + new String("tic");
//t
//      String f = new String("su") + new String("per"); //t
//      String f = new String("sw") + new String("itch");
//t
//      String f = new String("synchronize") + new
String("d"); //t
//      String f = new String("th") + new String("is"); //t
//      String f = new String("th") + new String("row"); //t
```

```
//      String f = new String("th") + new String("rows");
//t
//      String f = new String("trans") + new String("ient");
//t
//      String f = new String("tr") + new String("y"); //t
//      String f = new String("vola") + new String("tile");
//t
//      String f = new String("whi") + new String("le"); //t

// -----分割线-----

//      String f = new String("boo") + new String("lean");
//f
//      String f = new String("by") + new String("te"); //f
//      String f = new String("ch") + new String("ar"); //f
//      String f = new String("de") + new String("fault");
//f
//      String f = new String("dou") + new String("ble");
//f
//      String f = new String("fal") + new String("se"); //f
//      String f = new String("flo") + new String("at"); //f
//      String f = new String("in") + new String("t"); //f
//      String f = new String("l") + new String("ong"); //f
//      String f = new String("nu") + new String("ll"); //f
//      String f = new String("sh") + new String("ort"); //f
//      String f = new String("tr") + new String("ue"); //f
//      String f = new String("vo") + new String("id"); //f

String f = new String("ja") + new String("va"); //f
System.out.println(f.intern() == f);
}

}
```

4.2、intern方法的好处

测试案例：

```
// 字符串数组的长度
static final int MAX = 1000 * 10000;
// 字符串数组
static final String[] arr = new String[MAX];

public static void main(String[] args) throws Exception {
    // 随机数数组
    Integer[] DB_DATA = new Integer[10];
    // 随机数对象
    Random random = new Random(10 * 10000);
    // 产生10个随机数，放入DB_DATA数组中保存
    for (int i = 0; i < DB_DATA.length; i++) {
        DB_DATA[i] = random.nextInt();
    }
    long t = System.currentTimeMillis();
    // 存储1000*10000个字符串对象
    for (int i = 0; i < MAX; i++) {
        arr[i] = new String(String.valueOf(DB_DATA[i %
DB_DATA.length])).intern();
        // arr[i] = new String("1") + new String("11");
        // arr[i] = new String("111").intern();
        // arr[i] = new String("111").intern();
    }

    System.out.println((System.currentTimeMillis() - t) +
"ms");
    System.gc();
}
```

}

以上程序会有很多重复的相同的字符串产生，但是这些字符串的值都是只有在运行期才能确定的。所以，只能我们通过intern显示的将其加入常量池，这样可以减少很多字符串的重复创建。

Jdk6 中常量池位于PremGen区，大小受限，不建议使用String.intern()方法，不过Jdk7 将常量池移到了Java堆区，大小可控，可以重新考虑使用String.intern()方法，但是由对比测试可知，使用该方法的耗时不容忽视，所以需要慎重考虑该方法的使用；

String.intern() 方法主要适用于程序中需要保存有限个会被反复使用的值的场景，这样可以减少内存消耗，同时在进行比较操作时减少时耗，提高程序性能。



4.3、intern案例分析

画图解析

```
public static void main(String[] args) {  
    String s = new String("1");  
    s.intern();  
    String s2 = "1";  
    System.out.println(s == s2);  
  
    String s3 = new String("1") + new String("1");  
    s3.intern();  
    String s4 = "11";  
    System.out.println(s3 == s4);  
}
```

具体为什么稍后再解释，然后将 `s3.intern();` 语句下调一行，放到 `String s4 = "11";` 后面。将 `s.intern();` 放到 `String s2 = "1";` 后面。是什么结果呢？

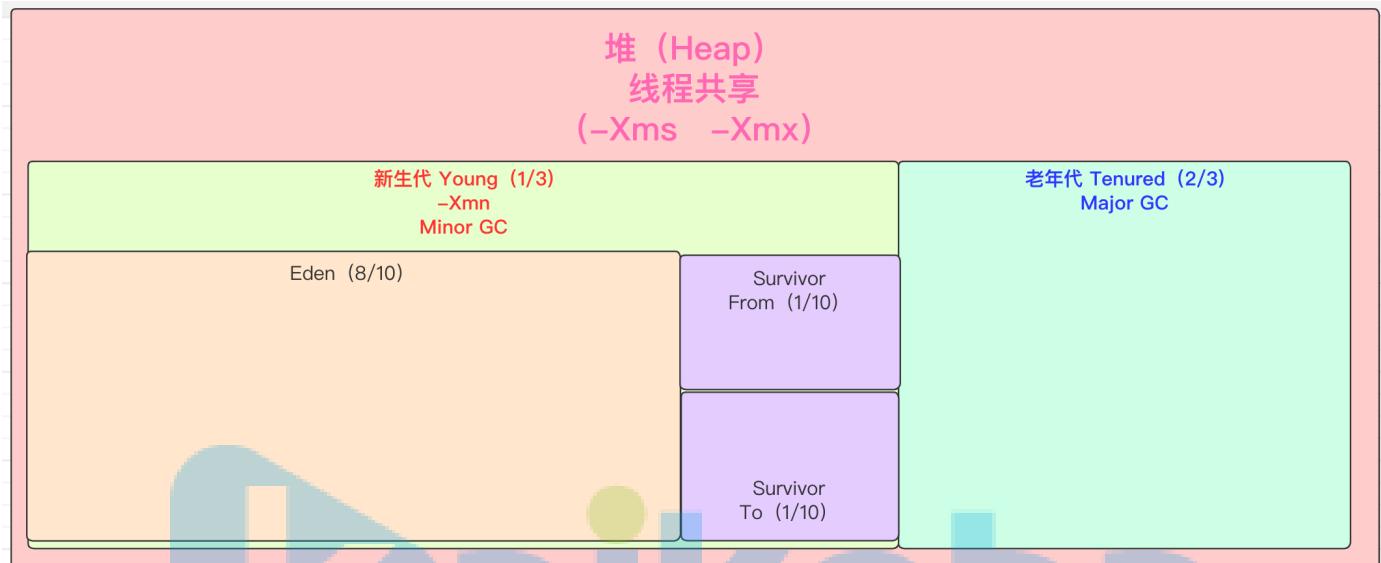
```
public static void main(String[] args) {  
    String s = new String("1");  
    String s2 = "1";  
    s.intern();  
    System.out.println(s == s2);  
  
    String s3 = new String("1") + new String("1");  
    String s4 = "11";  
    s3.intern();  
    System.out.println(s3 == s4);  
}
```

四、Java堆

堆内存分配

堆内存划分

<https://www.processon.com/diagraming/60bd7fe60e3e7468f4ba567b>

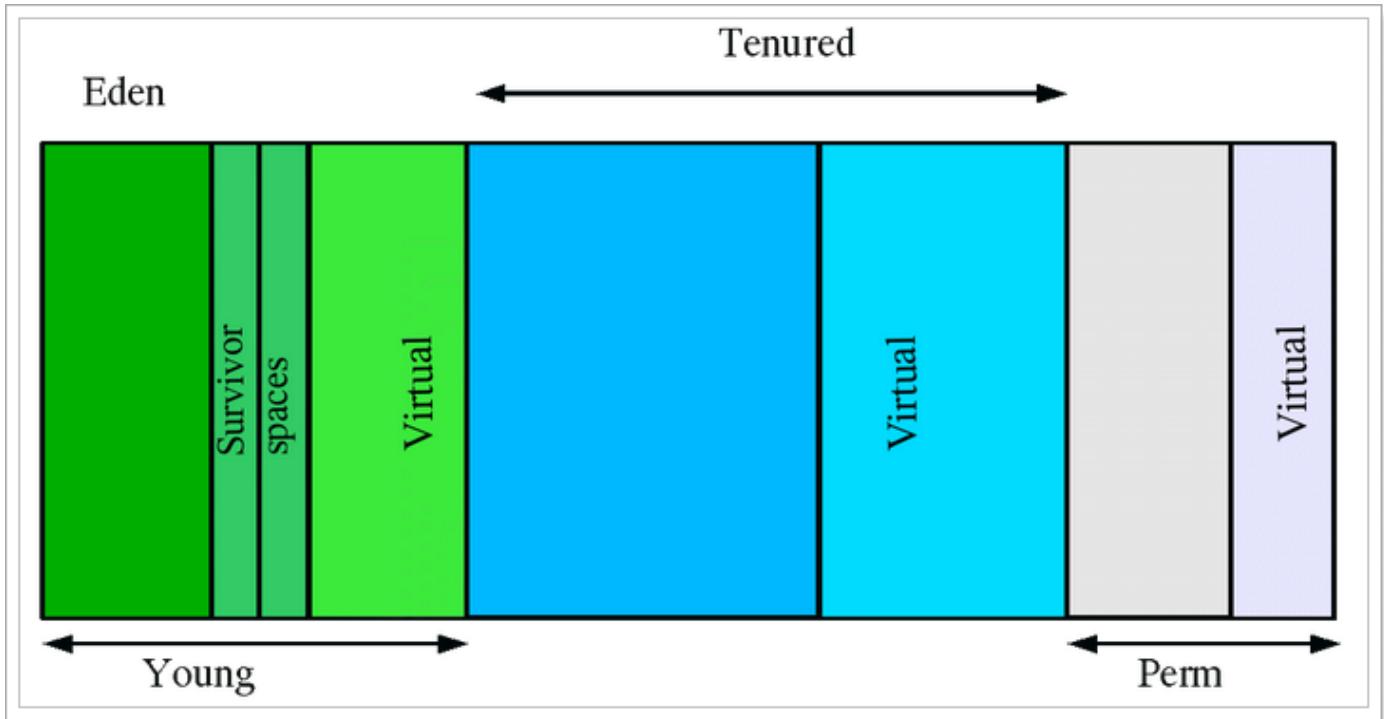


Java堆在JVM启动时创建，是通过【垃圾收集器】去实现内存分配。它是虚拟机管理最大的一块内存。也是垃圾回收的主要区域，而且主要采用【分代回收算法】。

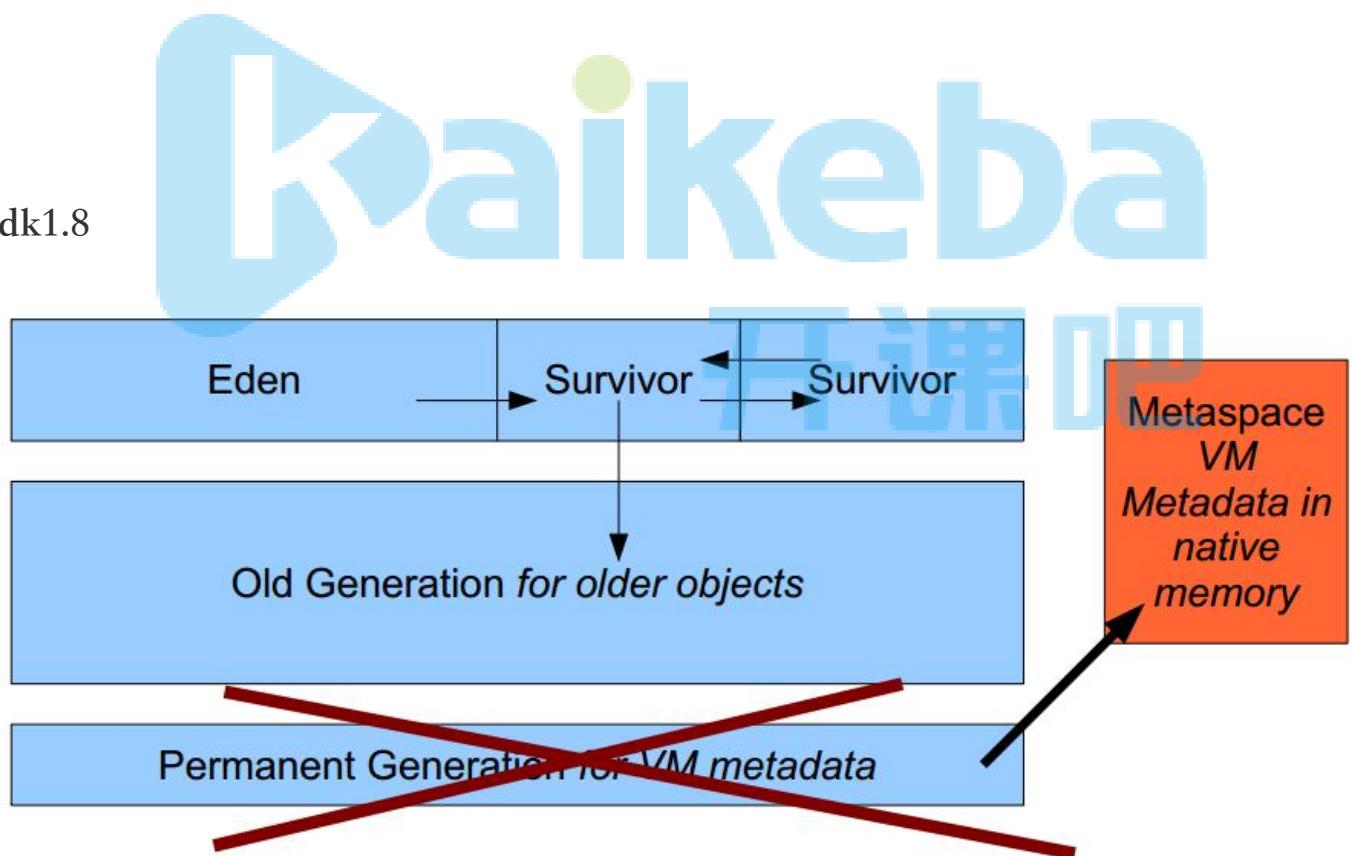
开课吧

堆内存模型

jdk1.7



jdk1.8

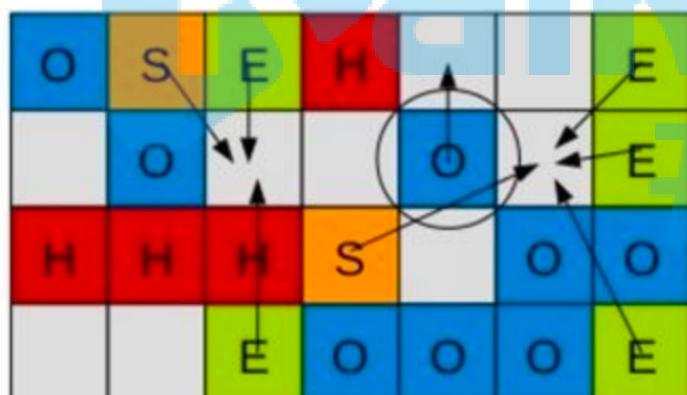


■ Familiar with this ?

- Eden
- Survivor
- Tenured

Young Generation

Old Generation



堆内存相关参数

通过工具查看堆内存信息

1、jvisualvm工具

```
package com.kkb.example;

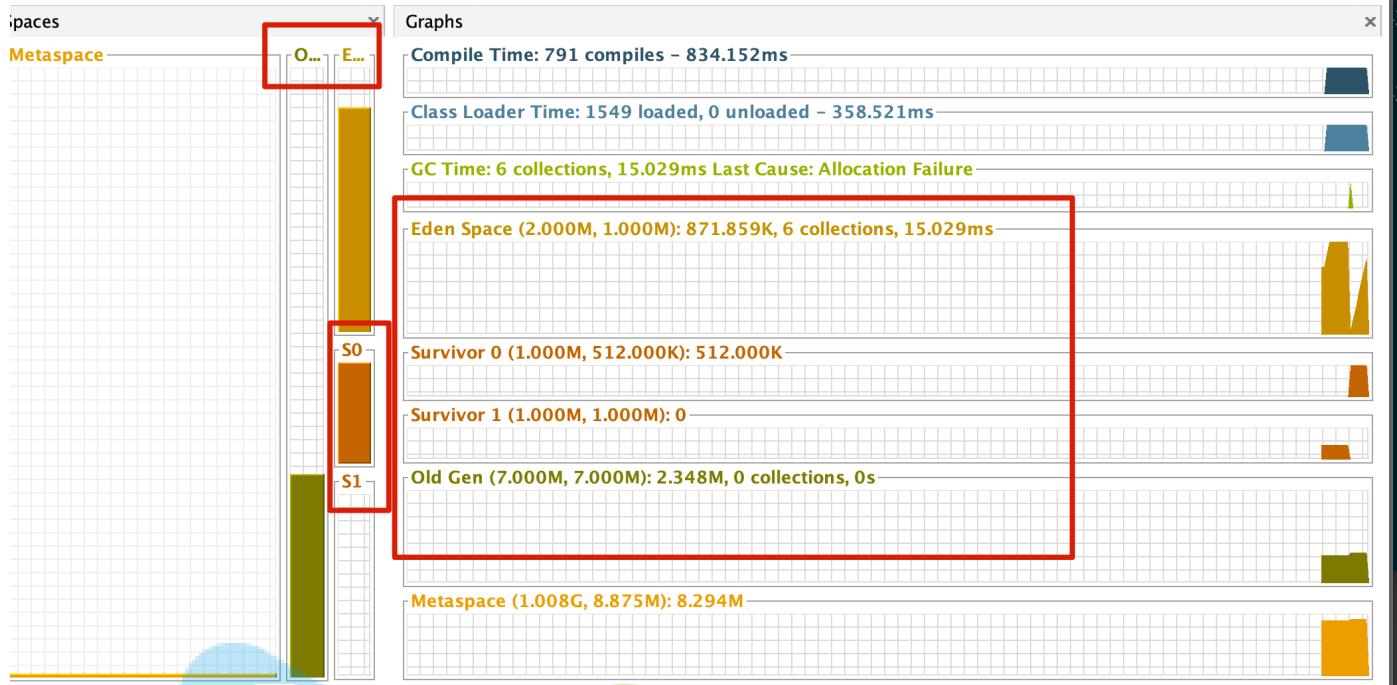
/**
 * -Xms10m -Xmx10m
 */
public class HeapDemo {
    public static void main(String[] args) {
        System.out.println("=====start=====");
        try {
            Thread.sleep(1000000L);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("=====end=====");
    }
}
```

com.kkb.example.HeapDemo (pid 50009)

Visual GC

Spaces Graphs Histogram

Refresh rate: Auto msec.



2、VM options -XX:+PrintGCDetails



```
package com.kkb.example;

/**
 * -Xms10m -Xmx10m -XX:+PrintGCDetails
 */
public class HeapDemo {
    public static void main(String[] args) {
        System.out.println("=====start=====");
        try {
            Thread.sleep(1000000L);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
        }
        System.out.println("=====end=====");
    }
}
```

停止进程即打印日志

```
Heap
PSYoungGen      total 2560K, used 1498K
[0x00000007bfd00000, 0x00000007c0000000,
0x00000007c0000000)
    eden space 2048K, 73% used
[0x00000007bfd00000,0x00000007bfe76bb8,0x00000007bff00000)
        from space 512K, 0% used
[0x00000007bff80000,0x00000007bff80000,0x00000007c0000000)
        to   space 512K, 0% used
[0x00000007bff00000,0x00000007bff00000,0x00000007bff80000)
ParOldGen       total 7168K, used 0K [0x00000007bf600000,
0x00000007bfd00000, 0x00000007bfd00000)
    object space 7168K, 0% used
[0x00000007bf600000,0x00000007bf600000,0x00000007bfd00000)
Metaspace        used 2713K, capacity 4486K, committed
4864K, reserved 1056768K
    class space     used 289K, capacity 386K, committed 512K,
reserved 1048576K
```

3、jstat 命令

```

→ ~ jps -l | grep 'HeapDemo'
52089 com.kkb.example.HeapDemo
→ ~ jstat -gc 52089
    S0C      S1C      S0U      S1U          EC          EU          OC
      OU        MC        MU      CCSC      CCSU      YGC      YGCT      FGC
      FGCT      GCT
 512.0   512.0    0.0    483.9   2048.0     59.9    7168.0
 464.0   4864.0  3202.6  512.0    349.4      1    0.006      0
      0.000    0.006
→ ~

```

- S0C: 第一个幸存区的大小
- S1C: 第二个幸存区的大小
- S0U: 第一个幸存区的使用大小
- S1U: 第二个幸存区的使用大小
- EC: 伊甸园区的大小
- EU: 伊甸园区的使用大小
- OC: 老年代大小
- OU: 老年代使用大小
- MC: 方法区大小
- MU: 方法区使用大小
- CCSC: 压缩类空间大小
- CCSU: 压缩类空间使用大小
- 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

4、jmap 命令

```
→ ~ jmap -heap 52089
```

```
Attaching to process ID 52089, please wait...
```

```
Debugger attached successfully.
```

```
Server compiler detected.
```

```
JVM version is 25.121-b13
```

```
using thread-local object allocation.
```

```
Parallel GC with 8 thread(s)
```

```
Heap Configuration:
```

MinHeapFreeRatio	= 0
MaxHeapFreeRatio	= 100
MaxHeapSize	= 10485760 (10.0MB)
NewSize	= 3145728 (3.0MB)
MaxNewSize	= 3145728 (3.0MB)
OldSize	= 7340032 (7.0MB)
NewRatio	= 2
SurvivorRatio	= 8
MetaspaceSize	= 21807104 (20.796875MB)
CompressedClassSpaceSize	= 1073741824 (1024.0MB)
MaxMetaspaceSize	= 17592186044415 MB
G1HeapRegionSize	= 0 (0.0MB)

```
Heap Usage:
```

```
PS Young Generation
```

```
Eden Space:
```

capacity	= 2097152 (2.0MB)
used	= 61336 (0.05849456787109375MB)
free	= 2035816 (1.9415054321289062MB)
2.9247283935546875% used	

From Space:

```
capacity = 524288 (0.5MB)
used      = 495544 (0.47258758544921875MB)
free      = 28744 (0.02741241455078125MB)
94.51751708984375% used
```

To Space:

```
capacity = 524288 (0.5MB)
used      = 0 (0.0MB)
free      = 524288 (0.5MB)
0.0% used
```

PS Old Generation

```
capacity = 7340032 (7.0MB)
used      = 475168 (0.453155517578125MB)
free      = 6864864 (6.546844482421875MB)
6.473650251116071% used
```

2143 interned Strings occupying 151616 bytes.

堆空间的参数设置

<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html>

-XX:+PrintFlagsInitial: 查看所有参数的默认初始值

-XX:+PrintFlagsFinal: 查看最终值 (初始值可能被修改掉)

-Xms: 初始堆空间内存(默认为物理内存的1/64)

-Xmx: 最大堆空间内存(默认为物理内存的1/4)

-Xmn: 设置新生代的大小。(初始值及最大值)

- XX:NewRatio: 配置新生代与老年代在堆结构的占比
- XX:SurvivorRatio: 设置新生代中Eden和S0/S1空间的比例
- XX:MaxTenuringThreshold 设置新生代垃圾的最大年龄
- XX:+PrintGCDetails: 输出详细的GC处理日志 打印GC简要信息: -XX:PrintGC -verbose:gc
- XX:HandlePromotionFailure: 是否设置空间担保

演示

```
package com.kkb.example;
```

```
/**
```

1. 设置堆空间大小的参数

-X 是jvm 的运行参数

-Xms 用来设置堆空间(年轻代+老年代)的初始内存大小

-Xmx 用来设置堆空间(年轻代+老年代)的最大内存大小

2. 手动设置 -Xms600m -Xmx600m

开发中建议将初始化堆内存和最大的堆内存设置成相同的值

3. 查看设置的参数: 方式一: jps / jstat -gc 进程id

方式二: -XX:+PrintGCDetails

4. 默认值

初始内存大小: 物理电脑内存大小 /64 ; 最大内存大小: 物理电脑内存大小 /4。

```
*/
```

```
public class HeapSpaceInitial {
```

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

```
long initMemory =  
Runtime.getRuntime().totalMemory() / 1024 / 1024;  
long maxMemory =  
Runtime.getRuntime().maxMemory() / 1024 / 1024;  
  
System.out.println("-Xms:" + initMemory + " M");  
System.out.println("-Xmx:" + maxMemory + " M");  
  
System.out.println("系统内存大  
小：" + initMemory * 64.0 / 1024 + " G");  
System.out.println("系统内存大  
小：" + maxMemory * 4.0 / 1024 + " G");
```

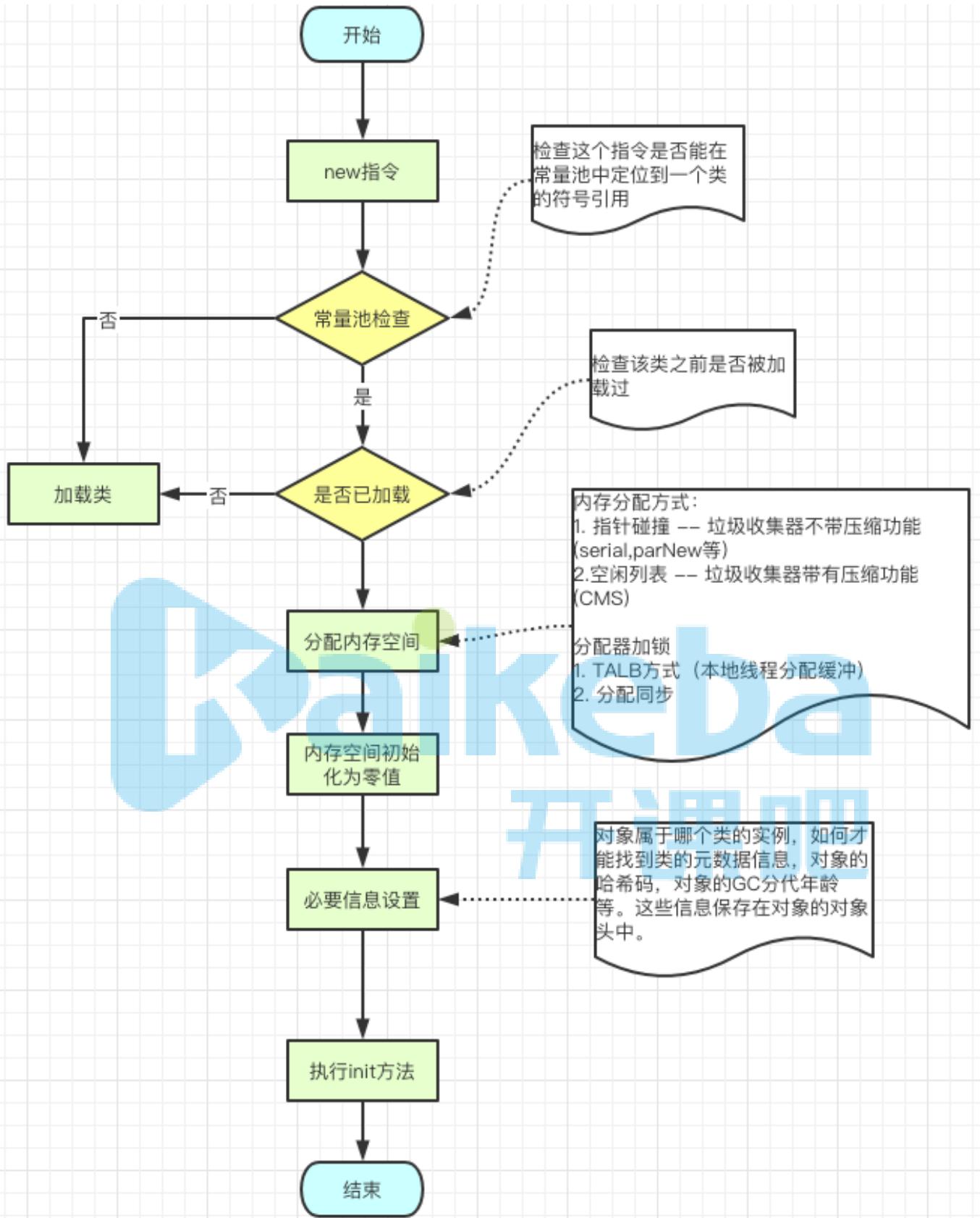
}



对象内存分配

对象创建流程

```
Student stu = new Student();
```



对象内存分配方式

内存分配的方法有两种:[指针碰撞\(Bump the Pointer\)](#)和[空闲列表\(Free List\)](#)

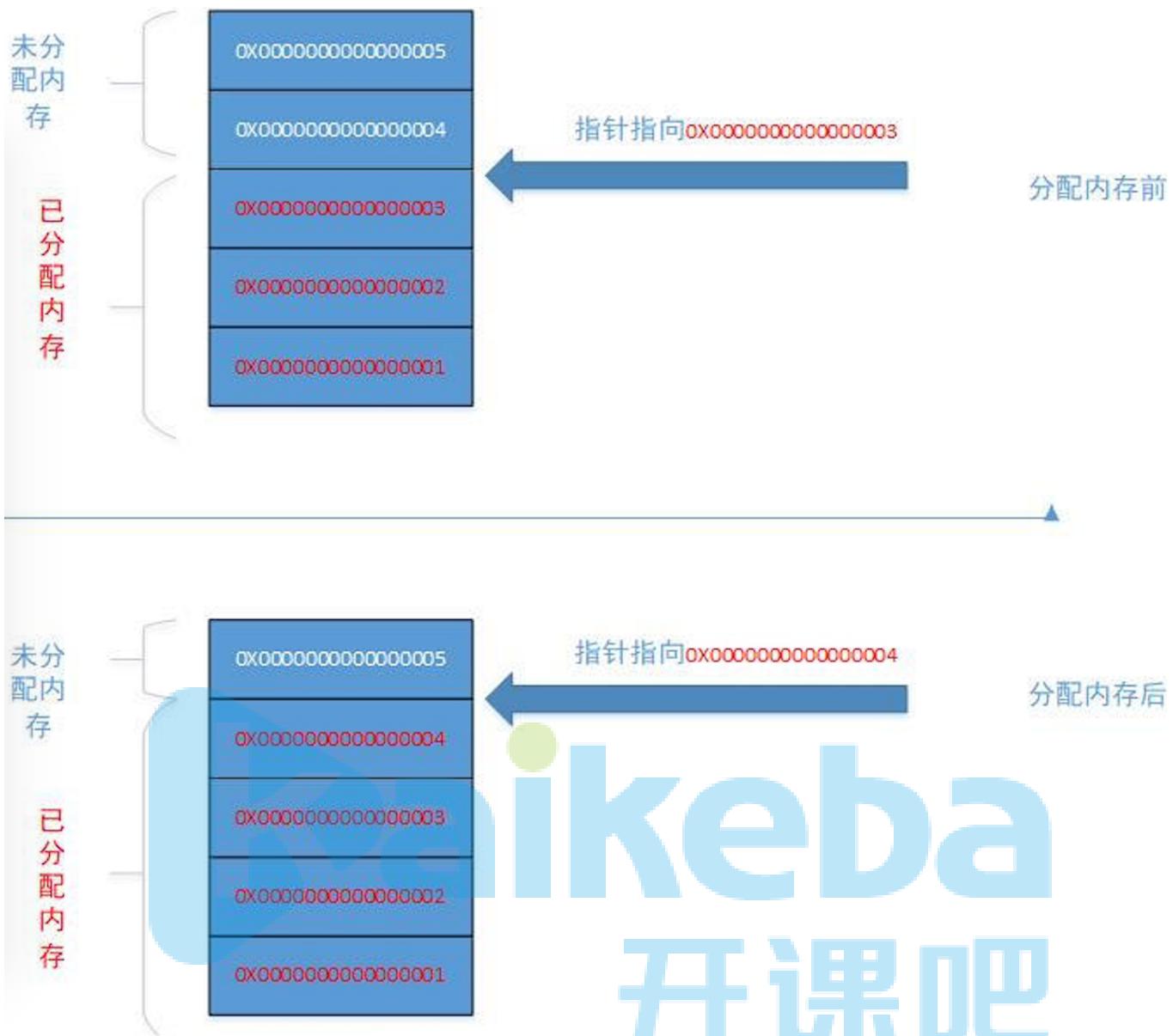
分配方法	说明	收集器
指针碰撞	内存地址是连续的（年轻代）	Serial 和 ParNew 收集器
空闲列表	内存地址不连续（年老代）	CMS 收集器和 Mark-Sweep 收集器





指针碰撞





对象内存分配安全问题

在分配内存的时候，虚拟机给A线程分配内存过程中，指针未修改。此时B线程同时使用了同样一块内存。

在JVM中有两种解决办法：

1. CAS 是乐观锁的一种实现方式。虚拟机采用 CAS 配上失败重试的方式保证更新操作的原子性。
2. TLAB, 本地线程分配缓冲(Thread Local Allocation Buffer即TLAB): 为

每一个线程预先分配一块内存

JVM在第一次给线程中的对象分配内存时，首先使用CAS进行TLAB的分配。

当对象大于TLAB中的剩余内存或TLAB的内存已用尽时，再采用上述的CAS进行内存分配。

对象内存分配流程

<https://www.processon.com/diagraming/60c5f6950791296f0ab4de73>

进入老年代的条件

<https://www.processon.com/mindmap/60c5e4771e085306cf77785d>

```

/**
 * 测试：大对象直接进入到老年代
 * -Xmx60m -Xms60m -XX:NewRatio=2 -XX:SurvivorRatio=8 -
XX:+PrintGCDetails
 * -XX:PretenureSizeThreshold
 *
 */
public class YoungOldAreaTest {
    public static void main(String[] args) {
        byte[] buffer = new byte[1024*1024*20]; //20M
    }
}

```

```

/Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/bin/java ...
Heap
PSYoungGen      total 18432K, used 2311K [0x00000007bec00000, 0x00000007c000000, 0x00000007c000000)
  eden space 16384K, 14% used [0x00000007bec00000, 0x00000007bee41c10, 0x00000007bfc00000)
  from space 2048K, 0% used [0x00000007bfe00000, 0x00000007bfe00000, 0x00000007c000000)
  to   space 2048K, 0% used [0x00000007bfc00000, 0x00000007bfc00000, 0x00000007bfe00000)
ParOldGen       total 40960K, used 20480K [0x00000007bc400000, 0x00000007bec00000, 0x00000007bec00000)
  object space 40960K, 50% used [0x00000007bc400000, 0x00000007bd800010, 0x00000007bec00000)
Metaspace        used 2711K, capacity 4486K, committed 4864K, reserved 1056768K
  class space   used 289K, capacity 386K, committed 512K, reserved 1048576K

Process finished with exit code 0

```

开课吧

代码演示对象分配过程

```

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

/*
-Xmx600m -Xms600m
*/

```

```
public class HeapInstanceTest {  
    byte[] buffer = new byte[new  
Random().nextInt(1024*200)];  
  
    public static void main(String[] args) {  
        List<HeapInstanceTest> list = new  
ArrayList<HeapInstanceTest>();  
        while (true){  
            list.add(new HeapInstanceTest());  
            try {  
                Thread.sleep(10);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```



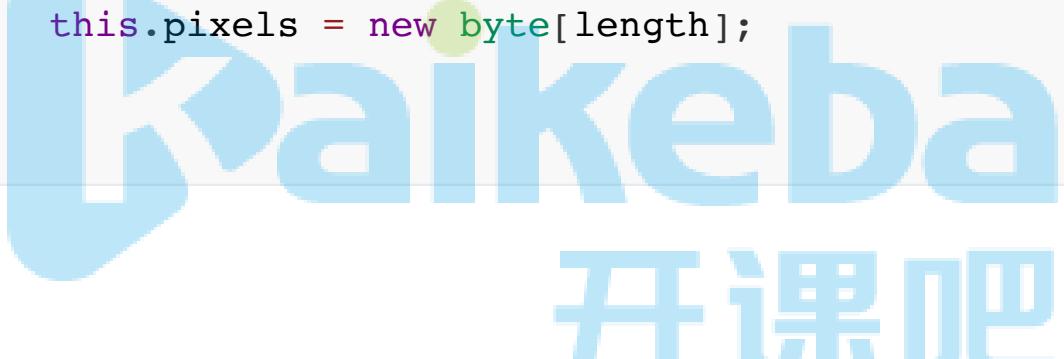


OOM的说明与举例

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
/*
-Xmx600m -Xms600m -XX:+PrintGCDetails
*/
public class OOMTest {
    public static void main(String[] args) {
        List<Picture> list = new ArrayList<>();
        while (true){
```

```
        try {
            Thread.sleep(20);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        list.add(new Picture(new Random().nextInt(1024
* 1024)));
    }
}

class Picture{
    private byte[] pixels;
    public Picture(int length){
        this.pixels = new byte[length];
    }
}
```





体会堆空间分代的思想

为什么需要把java堆分代? 不分代就不能正常工作了吗?

开课吧

经研究, 不同对象的生命周期不同, 70%-99%的对象是临时对象。

新生代: 有Eden、两块相同大小的Survivor(又称from/to,s0/s1)构成, to总为空

老年代: 存放新生代中经历多次GC仍然存活的对象。

其实不分代完全可以, 分代的唯一理由就是优化GC性能。如果没有分代, 那所有的对象都在一块, 就如同把一个学校的人都关在一个教室。GC的时候要找到哪些对象没用, 这样就会对堆的所有区域进行扫描。而很多对象都是朝生夕死的, 如果分代的话, 把新创建的对象放到某一地方, 当GC的

时候先把这块存储“朝生夕死”对象的区域进行回收，这样就会腾出很大的空间出来。

对象内存分配担保(老年代)

当新生代无法分配内存的时候，我们想把新生代的对象转移到老生代，然后把新对象放入腾空的新生代。此时就需要内存担保机制。

案例准备

1、JVM参数： `-Xms20M -Xmx20M -Xmn10M`

2、分配三个2MB的对象和一个4MB的对象

代码如下：

```
/*
 * 内存分配担保案例
 *
 * @author 灭霸詹
 *
 */
public class MemoryAllocationGuarantee {

    private static final int _1MB = 1024 * 1024;

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



```
public static void memoryAllocation() {  
  
    byte[] allocation1, allocation2, allocation3,  
allocation4;  
  
    allocation1 = new byte[2 * _1MB]; //2M  
    allocation2 = new byte[2 * _1MB]; //2M  
    allocation3 = new byte[2 * _1MB]; //2M  
    allocation4 = new byte[4 * _1MB]; //4M  
//    allocation4 = new byte[5 * _1MB]; //4M  
//    allocation4 = new byte[3 * _1MB]; //4M  
  
    System.out.println("完毕");  
}  
}
```

堆内存分配情况如下：

新生代总可用空间为9216KB

eden space (8192K)

from space (1024 K)	to space (1024 K)
------------------------------	----------------------------

Survivor区

串行垃圾收集器案例

设置JVM参数：

```
-Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -  
XX:SurvivorRatio=8 -XX:+UseSerialGC
```

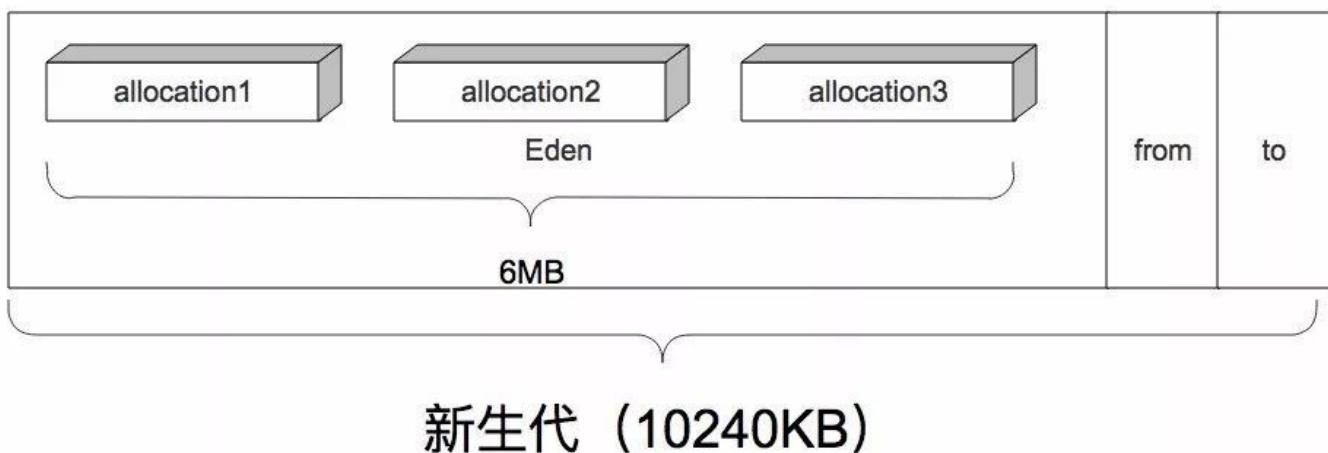
担保机制在JDK1.5以及之前版本中默认是关闭的，需要通过HandlePromotionFailure手动指定。

查看GC日志

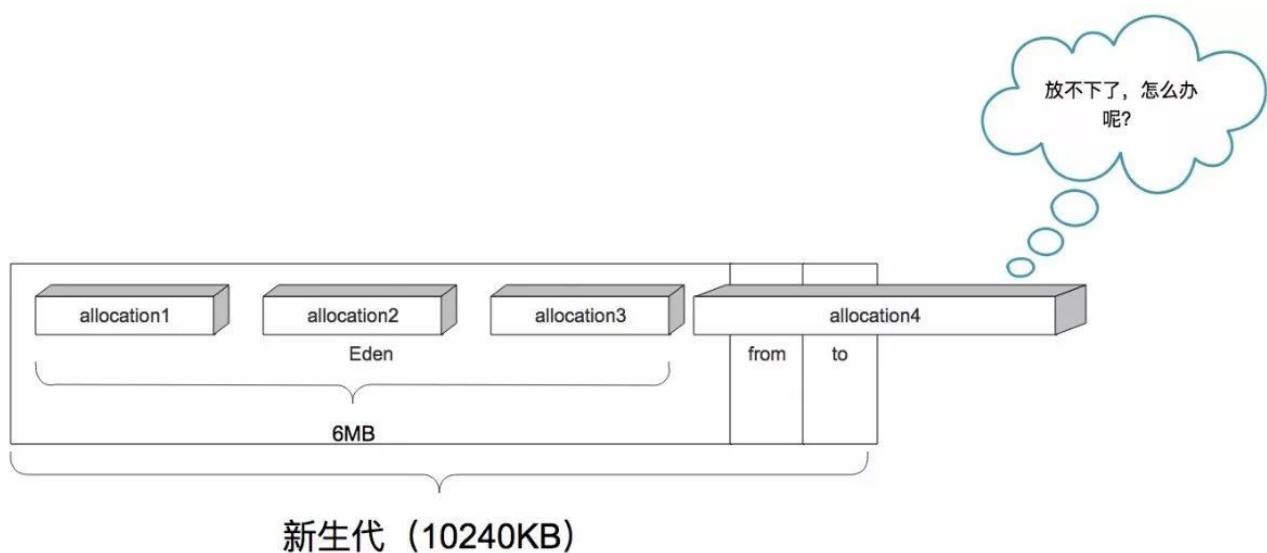
通过GC日志我们发现在分配allocation4的时候，发生了一次Minor GC，让新生代从7836K变为了472K，但是你发现整个堆的占用并没有多少变化。

分析过程

担保前的堆空间：



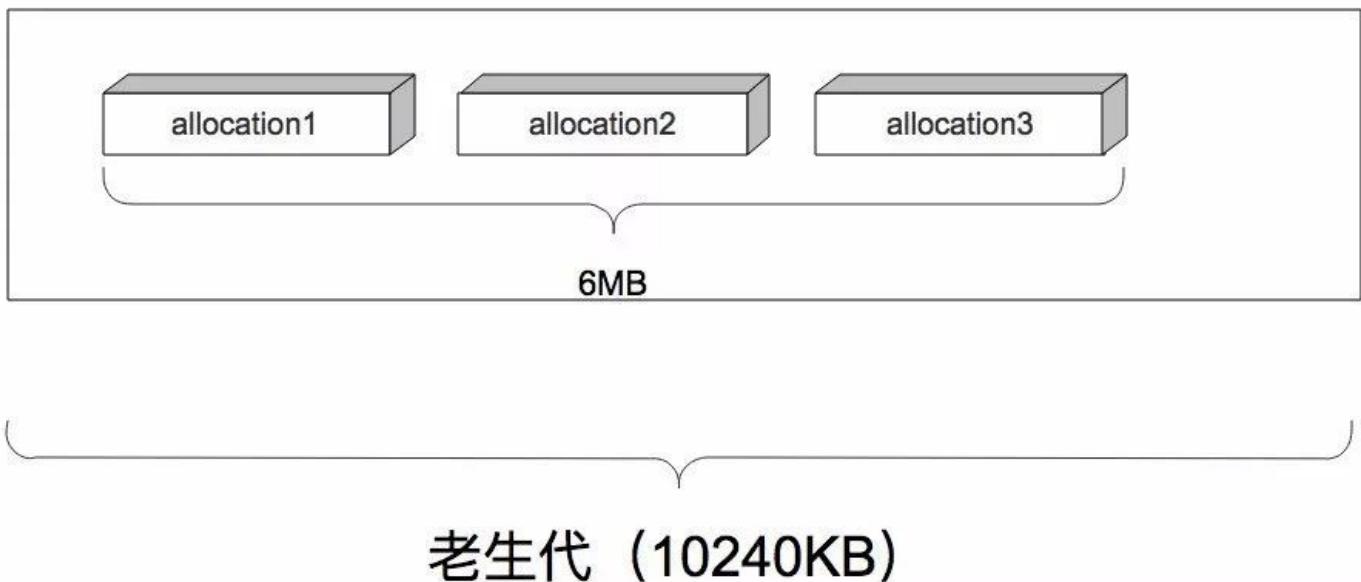
发生Minor GC，触发担保机制：



担保后的新生代：



担保后的老年代：



串行垃圾收集器案例结论

1. 当Eden区存储不下新分配的对象时，会触发minorGC。
2. GC之后，还存活的对象，按照正常逻辑，需要存入到Survivor区(幸存区)。
3. 当无法存入到幸存区时，此时会触发担保机制
4. 发生内存担保时，需要将Eden区GC之后还存活的对象放入老年带。后来的新对象或者数组放入Eden区。

并行垃圾收集器案例

设置JVM参数：

```
-Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -
XX:SurvivorRatio=8 -XX:+UseParallelGC
```

修改GC组合为 (Parallel Scavenge+Serial Old的组合)

查看GC日志

- 第四个对象是4MB的情况下：

- 第四个对象是3MB的情况下：

对象内存布局

内存布局概述

对象在内存中存储的布局可以分为三块区域：对象头 (Header)，实例数据 (Instance Data) 和 对齐填充 (Padding)。

开课吧

堆内对象

对象头

Markword

类型指针

数据区域



对其填充

对象头的大小

<https://www.processon.com/diagraming/60c787665653bb7a324408fe>

对象头在32位系统上占用8B，64位系统上占16B。

无论是32位系统还是64位系统，对象都采用8字节对齐。

Java对象头里的Mark Word里默认存储对象的HashCode，分代年龄和锁标记位。32位JVM的Mark Word的默认存储结构如下：

	25 bit 对象的 hashCode	4bit 对象分代年 龄	1bit 是否是偏 向锁	2bit 锁标志 位
无锁状 态			0	01

在运行期间Mark Word里存储的数据会随着锁标志位的变化而变化。Mark Word可能变化为存储以下4种数据：

锁状态	25 bit		4bit	1bit	2bit	
	23bit	2bit		是否是偏向锁	锁标志位	
轻量级锁	指向栈中锁记录的指针				00	
重量级锁	指向互斥量（重量级锁）的指针				10	
GC标记	空				11	
偏向锁	线程ID	Epoch	对象分代年龄	1	01	

开课吧

原生类型(primitive type)的内存占用如下：

Primitive Type	Memory Required(bytes)
boolean	1
byte	1
short	2
char	2
int	4
float	4
long	8
double	8

包装类型(初始化后)的内存占用如下：

Primitive Type	Memory Required(bytes)
Boolean	4
Byte	4
Short	4
Character	4
Integer	4
Float	4
Long	4
Double	4

案例验证1

示例代码

```
<dependency>
    <groupId>org.openjdk.jol</groupId>
    <artifactId>jol-core</artifactId>
    <version>0.9</version>
</dependency>
```

```
public class KKBObjLockTest {
    public static void main(String[] args) {
        Object o = new Object();
        System.out.println("new Object:" +
ClassLayout.parseInstance(o).toPrintable());
    }
}
```

控制台输出

new Object:
Kaikeba
开课吧

```

new Object:java.lang.Object object internals:
OFFSET  SIZE   TYPE DESCRIPTION
VALUE
    0      4       (object header)
01 00 00 00 (00000001 00000000 00000000 00000000) (1)
    4      4       (object header)
00 00 00 00 (00000000 00000000 00000000 00000000) (0)
    8      4       (object header)
e5 01 00 f8 (11100101 00000001 00000000 11111000)
(-134217243)

    12     4       (loss due to the next object
alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4
bytes total

```

分析

首先对象头是包含MarkWord和类型指针这两部分信息的；

根据64位开启指针压缩的情况下，存放Class指针的空间大小是4字节，MarkWord是8字节，对象头为12字节；

根据公式计算 【对象实例的大小】 = 对象头 (12) + 实例数据 (0) + 对齐填充 (4)。

结论：新建Object对象，会在内存占用16个字节，其中Header占12个（markword占8个+classpointer占4个），没有实例数据，补充对齐4个。

案例代码2

示例代码

```
public class KKObjLockTest {  
    public static void main(String[] args) {  
        A a = new A();  
        System.out.println("new A:" +  
ClassLayout.parseInstance(a).toPrintable());  
        a.setFlag(true);  
        a.setI(1);  
        a.setStr("ABC");  
        System.out.println("赋值 A:" +  
ClassLayout.parseInstance(a).toPrintable());  
    }  
  
    static class A {  
        private boolean flag;  
        private int i;  
        private String str;  
  
        public void setFlag(boolean flag) {  
            this.flag = flag;  
        }  
  
        public void setStr(String str) {  
            this.str = str;  
        }  
  
        public void setI(int i) {  
            this.i = i;  
        }  
    }  
}
```

```
}
```

控制台输出

```
new A:com.lock.ObjLockTest$A object internals:
OFFSET  SIZE          TYPE DESCRIPTION
              VALUE
 0        4          (object header)
           01 00 00 00 (00000001 00000000 00000000
00000000) (1)
 4        4          (object header)
           00 00 00 00 (00000000 00000000 00000000
00000000) (0)
 8        4          (object header)
           73 c9 00 f8 (01110011 11001001 00000000
11111000) (-134166157)
12        4          int A.i
           0
16        1          boolean A.flag
           false
17        3          (alignment/padding gap)

20        4          java.lang.String A.str
           null

Instance size: 24 bytes
Space losses: 3 bytes internal + 0 bytes external = 3
bytes total
```

赋值 A:com.lock.ObjLockTest\$A object internals:

```
OFFSET  SIZE          TYPE DESCRIPTION
              VALUE
```

0	4	(object header)
	01 00 00 00 (00000001 00000000 00000000	
00000000)	(1)	
4	4	(object header)
	00 00 00 00 (00000000 00000000 00000000	
00000000)	(0)	
8	4	(object header)
	73 c9 00 f8 (01110011 11001001 00000000	
11111000)	(-134166157)	
12	4	int A.i
	1	
16	1	boolean A.flag
	true	
17	3	(alignment/padding gap)
20	4	java.lang.String A.str (object)
Instance size: 24 bytes		
Space losses: 3 bytes internal + 0 bytes external = 3 bytes total		

分析

新建对象A时，Header占12个（markword占8个+classpointer占4个），实例数据中 boolean占一个字节，会补齐三个，int占4个，String占4个，无需补充对齐。

例如

```
class C{
    A a;
    B b;
}
```

对象的大小 = 12B对象头 + 4B*2的实例数据 + 4B的填充 = 24B

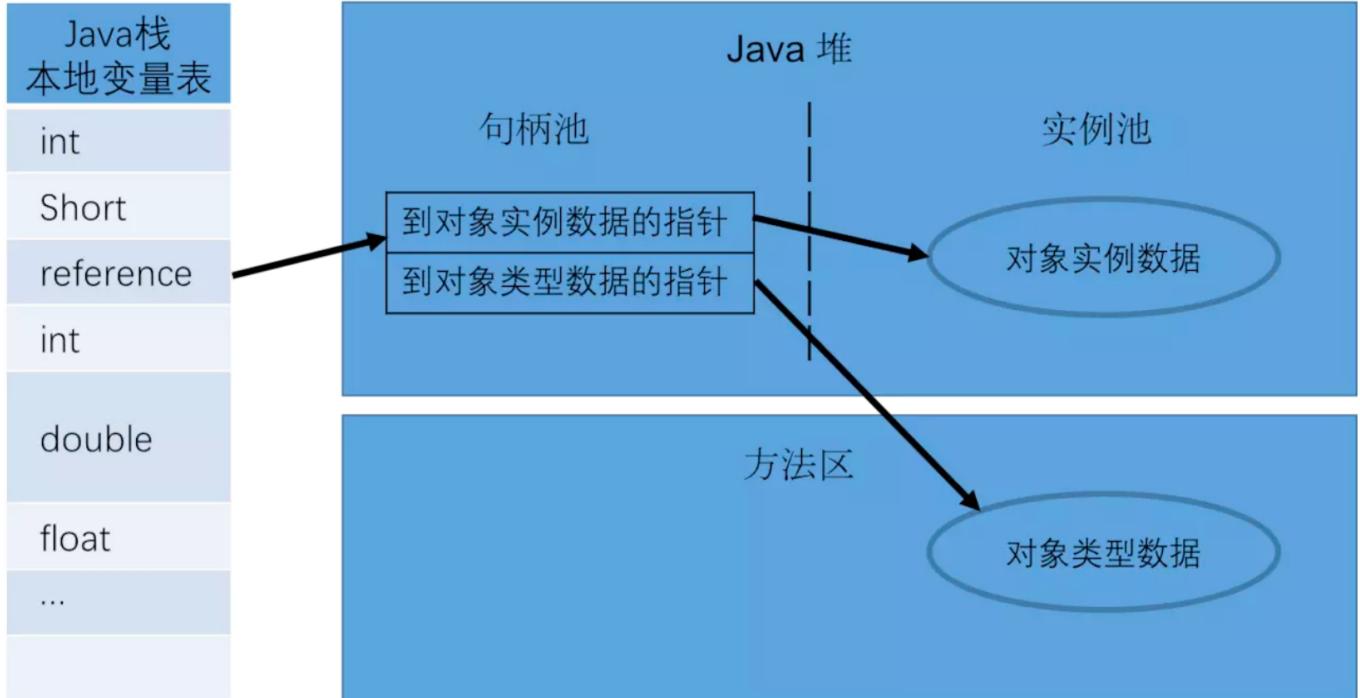
```
class C{  
    A a;  
    B b;  
    D d;  
}
```

对象的大小 = 12B对象头 + 4B*3的实例数据 + 0B的填充 = 24B

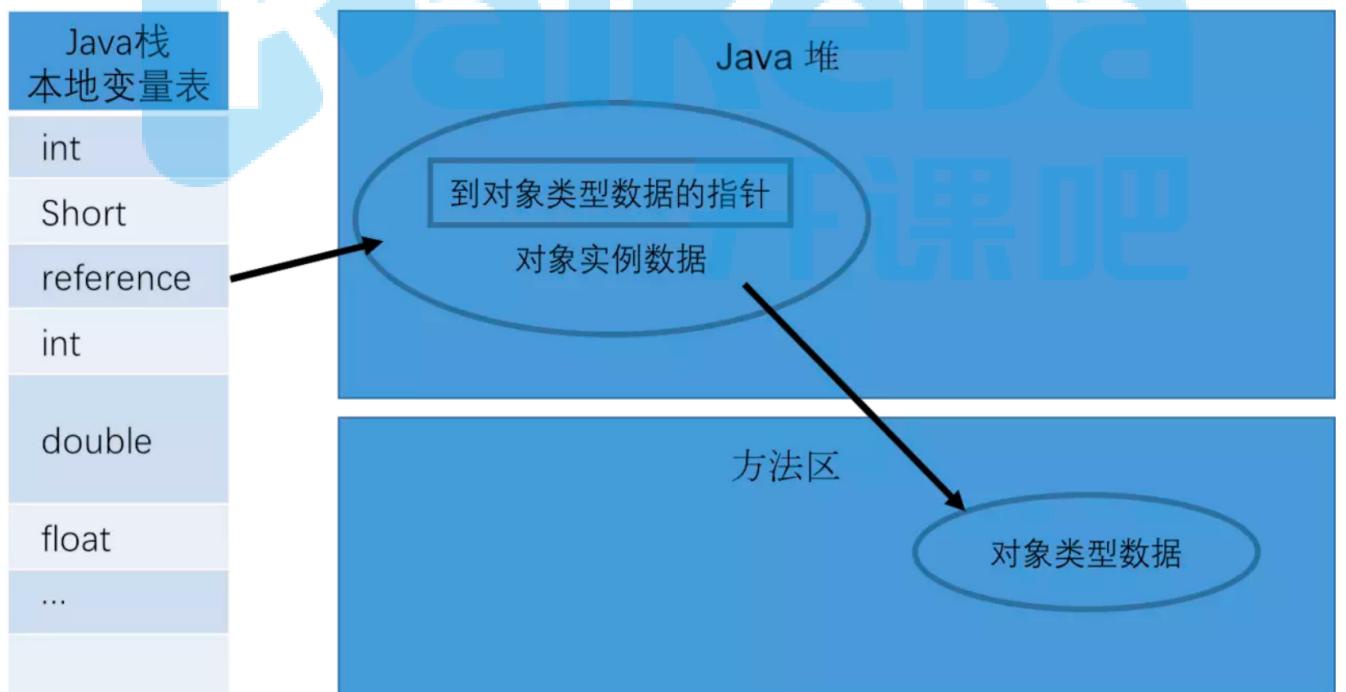
对象访问方式

方式	优点
句柄	稳定，对象被移动只要修改句柄中的地址
直接指针	访问速度快，节省了一次指针定位的开销

开课吧



通过句柄访问对象

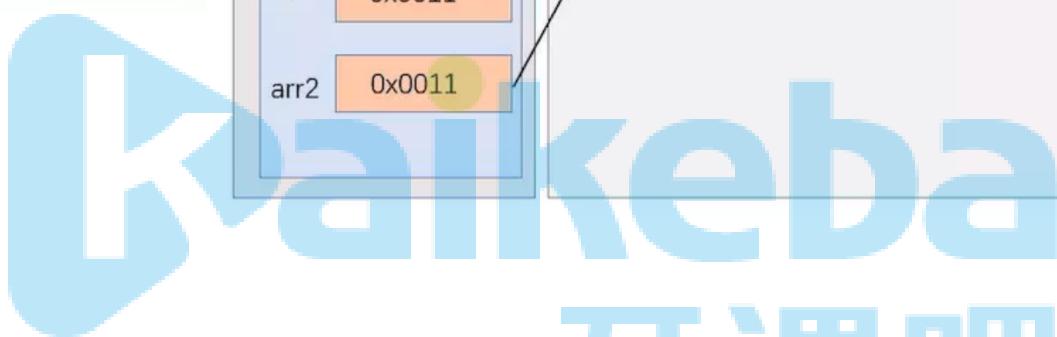


通过直接指针访问对象

数组的内存分析

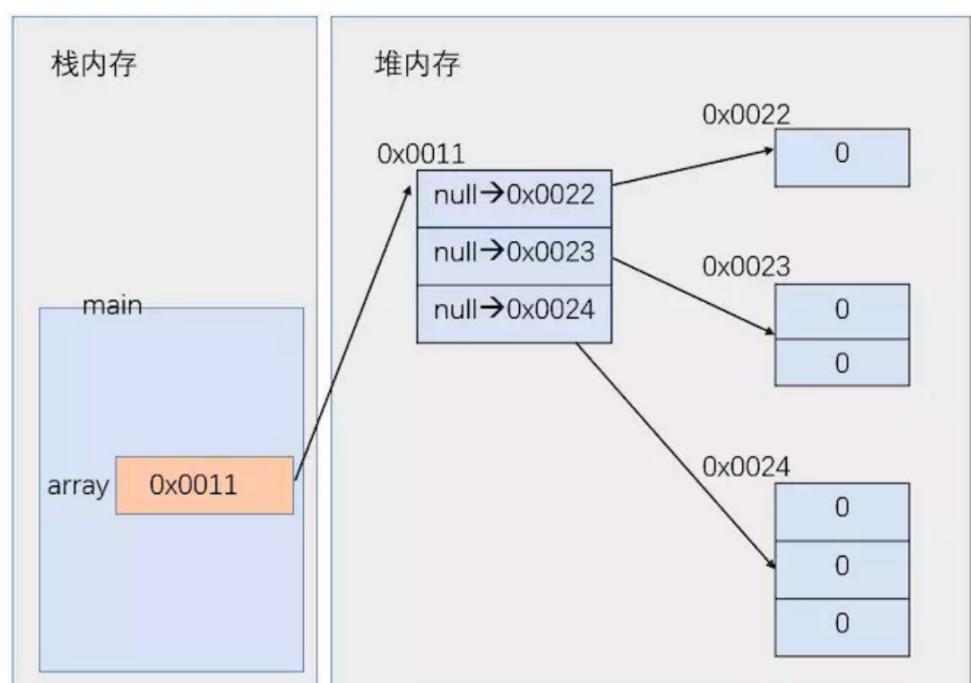
一维数组

```
int[ ] arr1 = new int[3];  
int[ ] arr2 = arr1;  
  
arr2[0] = 20;
```



二维数组

```
int[ ][ ] array = new int[3][ ];  
array[0][ ] = new int[1];  
array[1][ ] = new int[2];  
array[2][ ] = new int[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]
```

3.4.1、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

3.4.2、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

3.4.3、垃圾回收统计

```
[root@node01 ~]# jstat -gc 6219
SOC S1C SOU 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
SOC S1C SOU 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
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



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。

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.
```

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[[表示int[] ]L+类名 其他对象
```

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=fileName <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。

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/>



All Classes (excluding platform)

Package <Arrays>

```
class [Ljavax.el.ELResolver; [0xe36cc108]
class [Ljavax.servlet.DispatcherType; [0xe31fc930]
class [Ljavax.servlet.FilterConfig; [0xe36cee30]
class [Ljavax.servlet.SessionTrackingMode; [0xe2fe7678]
class [Ljavax.servlet.jsp.JspContext; [0xe3a5c880]
class [Ljavax.servlet.jsp.JspWriter; [0xe3a5c7b0]
class [Ljavax.servlet.jsp.PageContext; [0xe3a5c818]
class [Ljavax.servlet.jsp.tagext.BodyContent; [0xe3a5c748]
class [Ljavax.servlet.jsp.tagext.VariableInfo; [0xe36cda60]
class [Ljavax.websocket.CloseReason$CloseCode; [0xe31fccd0]
class [Ljavax.websocket.CloseReason$CloseCodes; [0xe31fcbc0]
class [Lorg.apache.catalina.AccessLog; [0xe3a5c678]
class [Lorg.apache.catalina.Container; [0xe2fb4a68]
class [Lorg.apache.catalina.ContainerListener; [0xe2fb49c0]
class [Lorg.apache.catalina.Executor; [0xe2eb3880]
class [Lorg.apache.catalina.InstanceListener; [0xe31fdbaa8]
class [Lorg.apache.catalina.Lifecycle; [0xe2ef44d0]
class [Lorg.apache.catalina.LifecycleListener; [0xe2fae748]
class [Lorg.apache.catalina.LifecycleState; [0xe2ef4538]
class [Lorg.apache.catalina.Service; [0xe2eb6a60]
class [Lorg.apache.catalina.Session; [0xe3a5c610]
class [Lorg.apache.catalina.Valve; [0xe3080410]
class [Lorg.apache.catalina.connector.Connector; [0xe2eb68a0]
class [Lorg.apache.catalina.core.ApplicationFilterConfig; [0xe36cedc8]
```

在最后面有OQL查询功能。

Other Queries

- [All classes including platform](#)
- [Show all members of the rootset](#)
- [Show instance counts for all classes \(including platform\)](#)
- [Show instance counts for all classes \(excluding platform\)](#)
- [Show heap histogram](#)
- [Show finalizer summary](#)
- [Execute Object Query Language \(OQL\) query](#)

The screenshot shows a web-based OQL query interface. At the top, the URL is 192.168.40.133:9999/oql/?query=select+s+from+java.lang.String+s+where+s.value.length+>%3D+10000%0D%0A%0... . The main title is "Object Query Language (OQL) query". Below it are links for "All Classes (excluding platform)" and "OQL Help". The query entered is "select s from java.lang.String s where s.value.length >= 10000" with a red annotation "查询字符串长度大于10000的内容". A "Execute" button is present. The results are listed in a table with a red border, showing five entries: "java.lang.String@0xe321a8d0", "java.lang.String@0xe338e038", "java.lang.String@0xe3684f98", "java.lang.String@0xe359f8c0", and "java.lang.String@0xe329f458". A red arrow points from the text "查询到的结果" to this table.

4.5、通过MAT工具对dump文件进行分析

4.5.1、MAT工具介绍

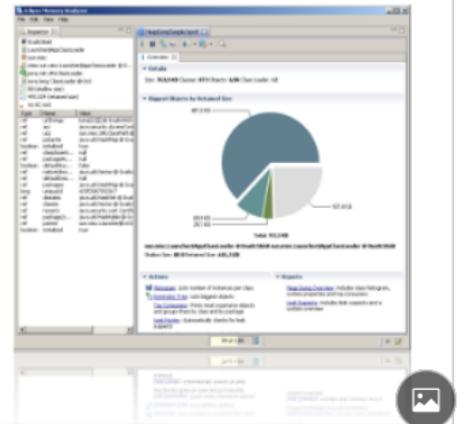
MAT(Memory Analyzer Tool)，一个基于Eclipse的内存分析工具，是一个快速、功能丰富的JAVA heap分析工具，它可以帮助我们查找内存泄漏和减少内存消耗。使用内存分析工具从众多的对象中进行分析，快速的计算出在内存中对象的占用大小，看看是谁阻止了垃圾收集器的回收工作，并可以通过报表直观的查看到可能造成这种结果的对象。

官网地址：<https://www.eclipse.org/mat/>

Memory Analyzer (MAT)

The Eclipse Memory Analyzer is a fast and feature-rich **Java heap analyzer** that helps you find memory leaks and reduce memory consumption.

Use the Memory Analyzer to analyze productive heap dumps with hundreds of millions of objects, quickly calculate the retained sizes of objects, see who is preventing the Garbage Collector from collecting objects, run a report to automatically extract leak suspects.



4.5.2、下载安装

下载地址：<https://www.eclipse.org/mat/downloads.php>

The **stand-alone** Memory Analyzer is based on Eclipse RCP. It is useful if you do not want to install a full-fledged IDE on the system you are running the heap analysis.

To install the Memory Analyzer **into an Eclipse IDE** use the update site URL provided below. The *Memory Analyzer (Chart)* feature is optional. The chart feature requires the BIRT Chart Engine (Version 2.3.0 or greater).

The minimum Java version required to run Memory Analyzer is 1.8

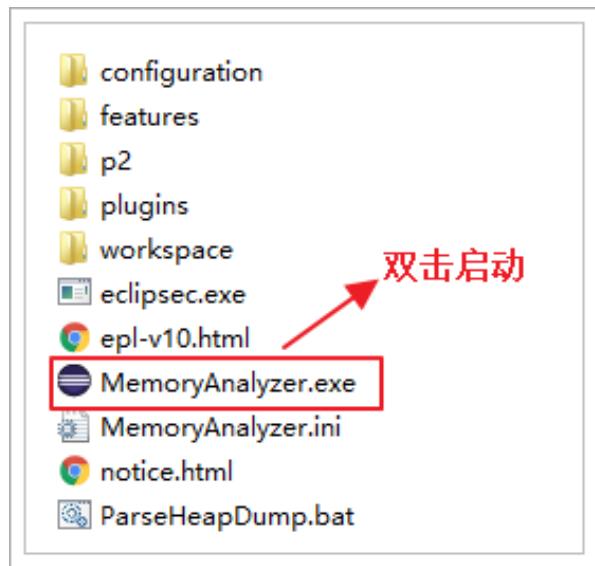
Memory Analyzer 1.8.0 Release

- **Version:** 1.8.0.20180604 | **Date:** 27 June 2018 | **Type:** Released
 - **Update Site:** <http://download.eclipse.org/mat/1.8/update-site/>
 - **Archived Update Site:** [MemoryAnalyzer-1.7.0.201706130745.zip](#)
 - **Stand-alone Eclipse RCP Applications**
 - Windows (x86)
 - Windows (x86_64)**
 - Mac OSX (Mac/Cocoa/x86_64)
 - Linux (x86/GTK+)
 - Linux (x86_64/GTK+)
 - Linux (PPC64/GTK+)
 - Linux (PPC64le/GTK+)

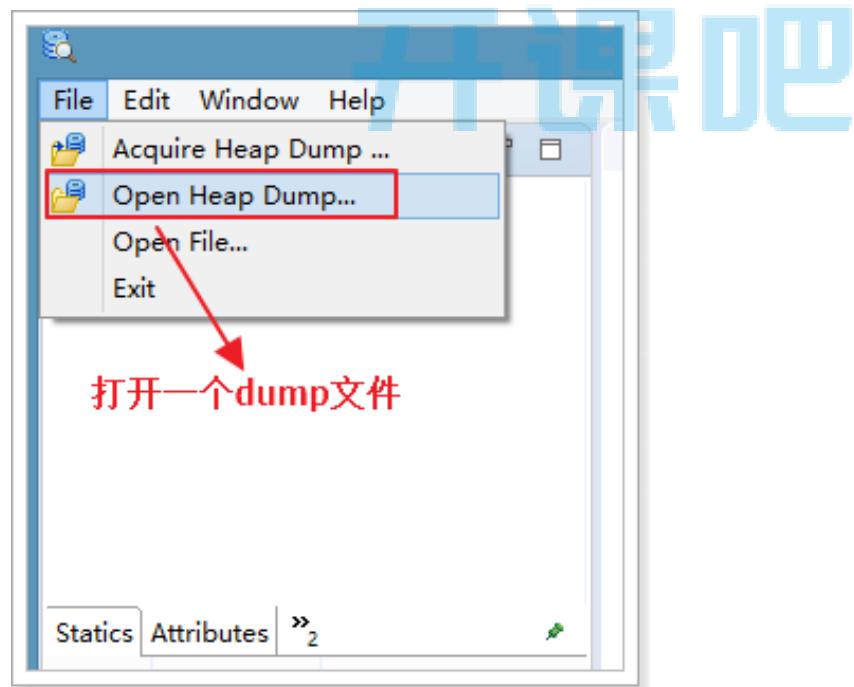
Other Releases

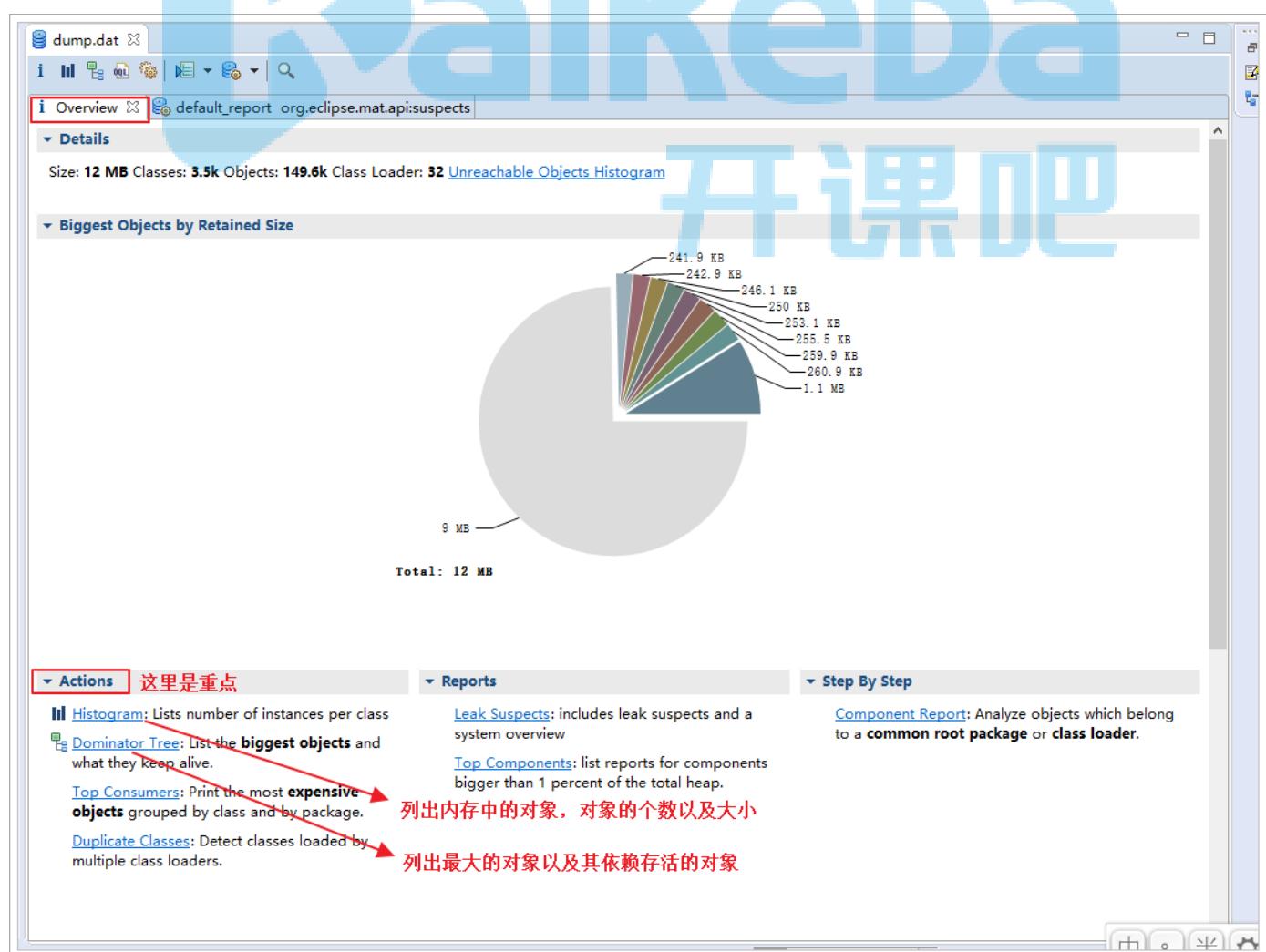
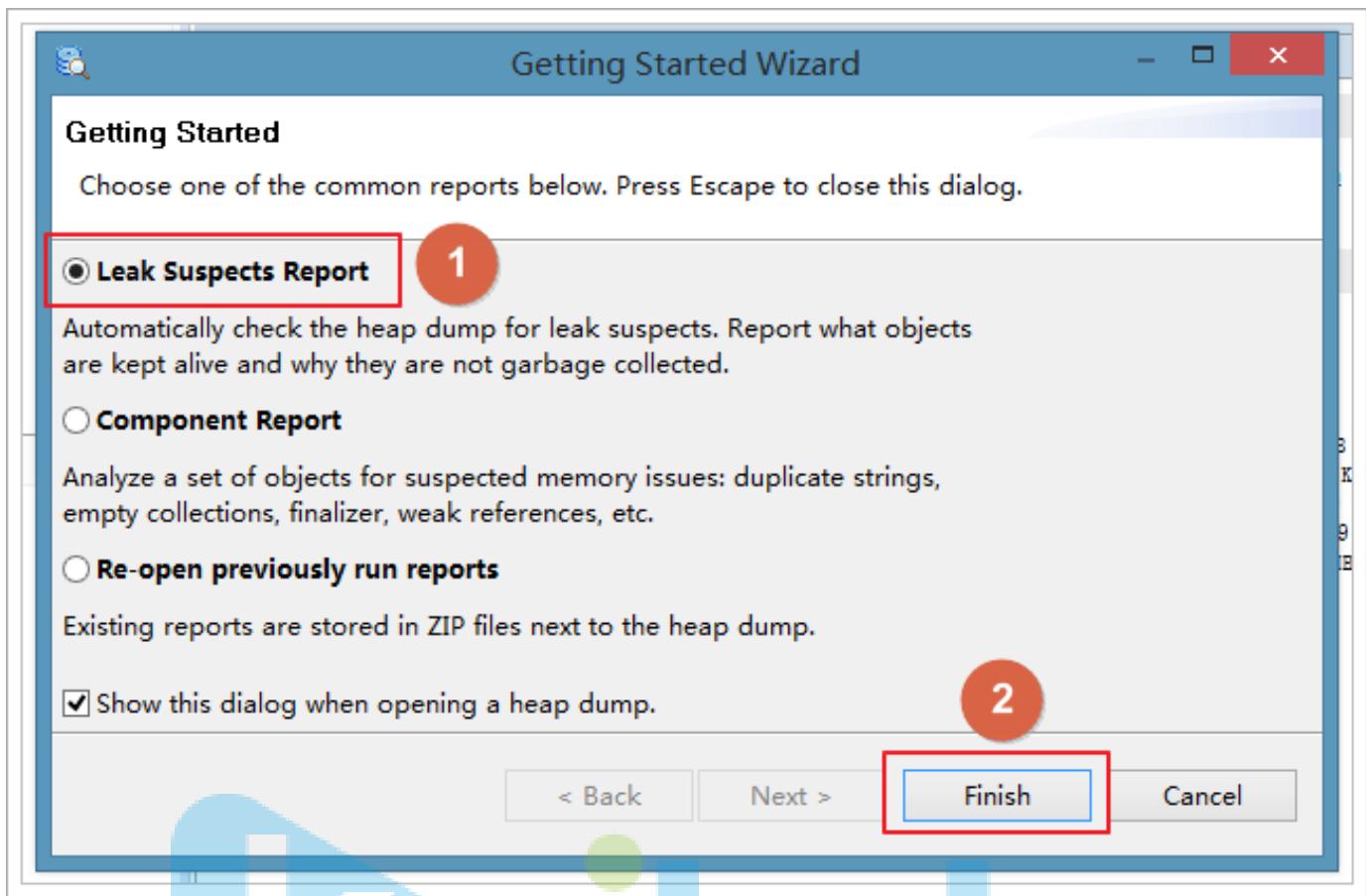
- [Previous Releases](#)
- [Snapshot Builds](#)

将下载得到的MemoryAnalyzer-1.8.0.20180604-win32.win32.x86_64.zip进行解压：



4.5.3、使用





在这里通过正则进行搜索

Class Name	Objects	Shallow He...	Retained H...
C char[]	32,094	7,575,720	>= 7,575,7...
C java.lang.String	29,573	709,752	>= 4,907,7...
C byte[]	878	654,704	>= 654,704
C java.util.HashMap\$N...	17,161	549,152	>= 1,911,5...
C java.lang.reflect.Meth...	3,720	327,360	>= 441,480
C java.lang.Object[]	4,712	259,504	>= 1,574,8...
C java.util.HashMap\$N...	995	206,472	>= 2,115,3...
C java.util.concurrent.C...	4,305	137,760	>= 1,759,4...
C int[]	1,423	130,568	>= 130,568
C java.lang.String[]	1,269	109,064	>= 549,040
C java.util.concurrent.C...	64	84,128	>= 1,834,3...
C java.util.HashMap	1,674	80,352	>= 2,166,6...
C java.util.ArrayList	2,871	68,904	>= 308,840
C java.lang.Class[]	2,957	64,744	>= 64,744
C java.lang.Object	3,979	63,664	>= 63,664
C org.apache.tomcat.ut...	1,271	61,008	>= 91,456
C java.util.LinkedHashM...	1,487	59,480	>= 125,576
C com.sun.org.apache....	1,671	53,472	>= 53,472

查看对象以及它的依赖：

i Overview default_report org.eclipse.mat.apis:suspects dominator_tree

Class Name	Shallow Heap	Retained Heap	Percentage
	<Numeric>	<Numeric>	<Numeric>
↳ <Regex>			
↳ org.apache.catalina.loader.StandardClassLoader @ 0xe345e5f0	80	1,129,024	8.97%
↳ java.util.Vector @ 0xe2e6dc68	32	1,071,288	8.51%
↳ java.util.HashMap @ 0xe2e6ddb0	48	21,360	0.17%
↳ java.util.Hashtable @ 0xe2e6d4d0	48	12,232	0.10%
↳ sun.misc.URLClassPath @ 0xe2e6f008	48	11,464	0.09%
↳ java.util.WeakHashMap @ 0xe2e75d48	48	5,720	0.05%
↳ java.util.HashSet @ 0xe2e6dcf0	16	624	0.00%
↳ java.util.HashMap @ 0xe2e6ef58	48	608	0.00%
↳ java.security.ProtectionDomain @ 0xe2fa0860	40	536	0.00%
↳ java.security.ProtectionDomain @ 0xe2fa9c78	40	536	0.00%
↳ java.security.ProtectionDomain @ 0xe31a7510	40	536	0.00%
↳ java.security.ProtectionDomain @ 0xe2fa9ed0	40	528	0.00%
↳ java.security.ProtectionDomain @ 0xe2faa270	40	528	0.00%
↳ java.security.ProtectionDomain @ 0xe353b5a0	40	528	0.00%
↳ java.security.ProtectionDomain @ 0xe353dd68	40	528	0.00%
↳ java.security.ProtectionDomain @ 0xe336e248	40	520	0.00%
↳ java.net.URL @ 0xe2e6f1b0 file:/tmp/apache-tomcat-7.0.57/lib/t	64	112	0.00%
↳ java.net.URL @ 0xe2e6f590 file:/tmp/apache-tomcat-7.0.57/lib/j	64	112	0.00%
↳ java.net.URL @ 0xe2e6f878 file:/tmp/apache-tomcat-7.0.57/lib/v	64	112	0.00%
↳ java.net.URL @ 0xe2e6fb68 file:/tmp/apache-tomcat-7.0.57/lib/e	64	112	0.00%
↳ java.net.URL @ 0xe2e6fc50 file:/tmp/apache-tomcat-7.0.57/lib/j	64	112	0.00%
↳ java.net.URL @ 0xe2e6ff20 file:/tmp/apache-tomcat-7.0.57/lib/e	64	112	0.00%
↳ java.net.URL @ 0xe2fa08a8 file:/tmp/apache-tomcat-7.0.57/lib/c	64	112	0.00%
↳ java.net.URL @ 0xe2fa9a00 file:/tmp/apache-tomcat-7.0.57/lib/t	64	112	0.00%
↳ java.security.ProtectionDomain @ 0xe2e6dc88	40	104	0.00%
↳ java.util.Vector @ 0xe2e6ef00	32	88	0.00%
Σ Total: 25 of 34 entries; 9 more			

查看可能存在内存泄露的分析：

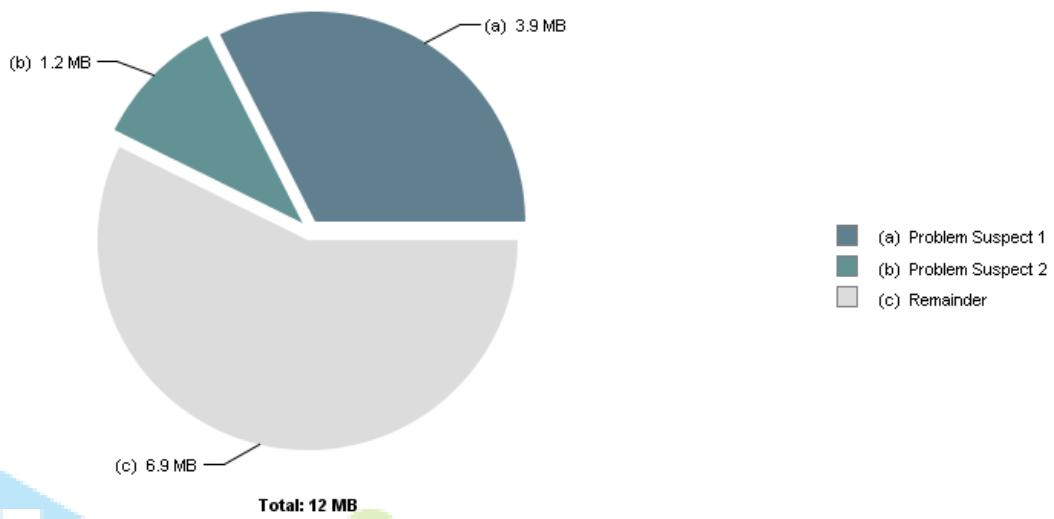
Leak Suspects

Leak Suspects

System Overview

▼ Leaks

▼ Overview



▼ Problem Suspect 1

Baikeba
开课吧