

# 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

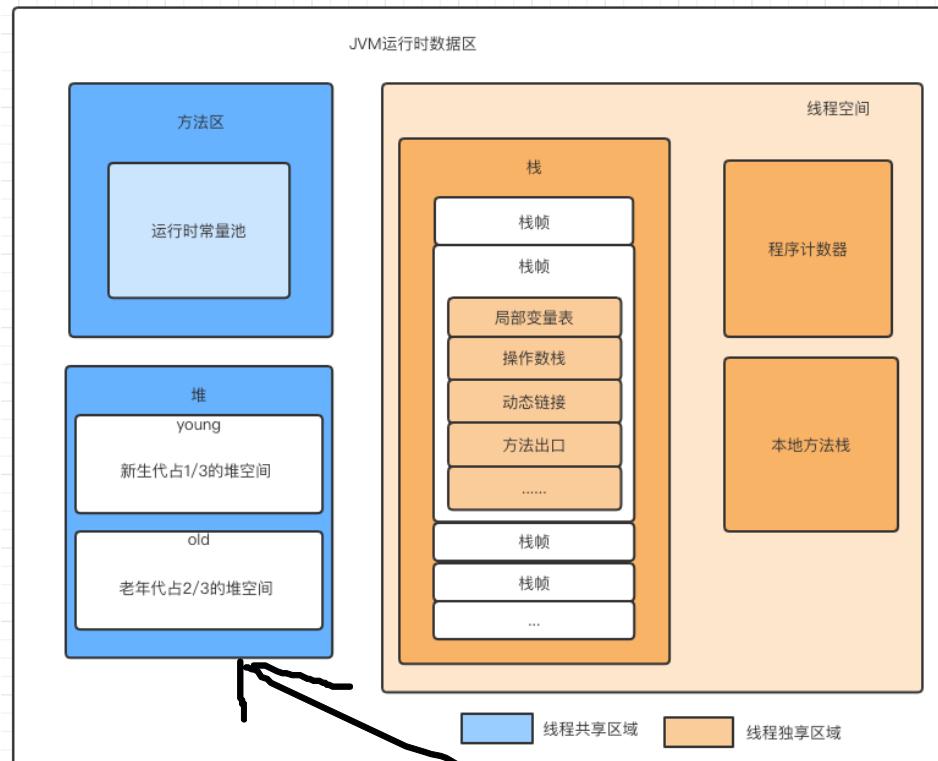
---

### 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
  - 堆和方法区

## 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

### 分配方法区的大小

jconsole打开~

`-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

1. 类型信息, 比如Class (com.kkb.User类)
2. 方法信息, 比如Method (方法名称、方法参数列表、方法返回值信息)
3. 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)
4. Code区, 存储的是方法执行对应的字节码指令
5. 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。
6. 静态变量 (类变量) ---JDK1.7之后, 转移到堆中存储
7. 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
8. 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？

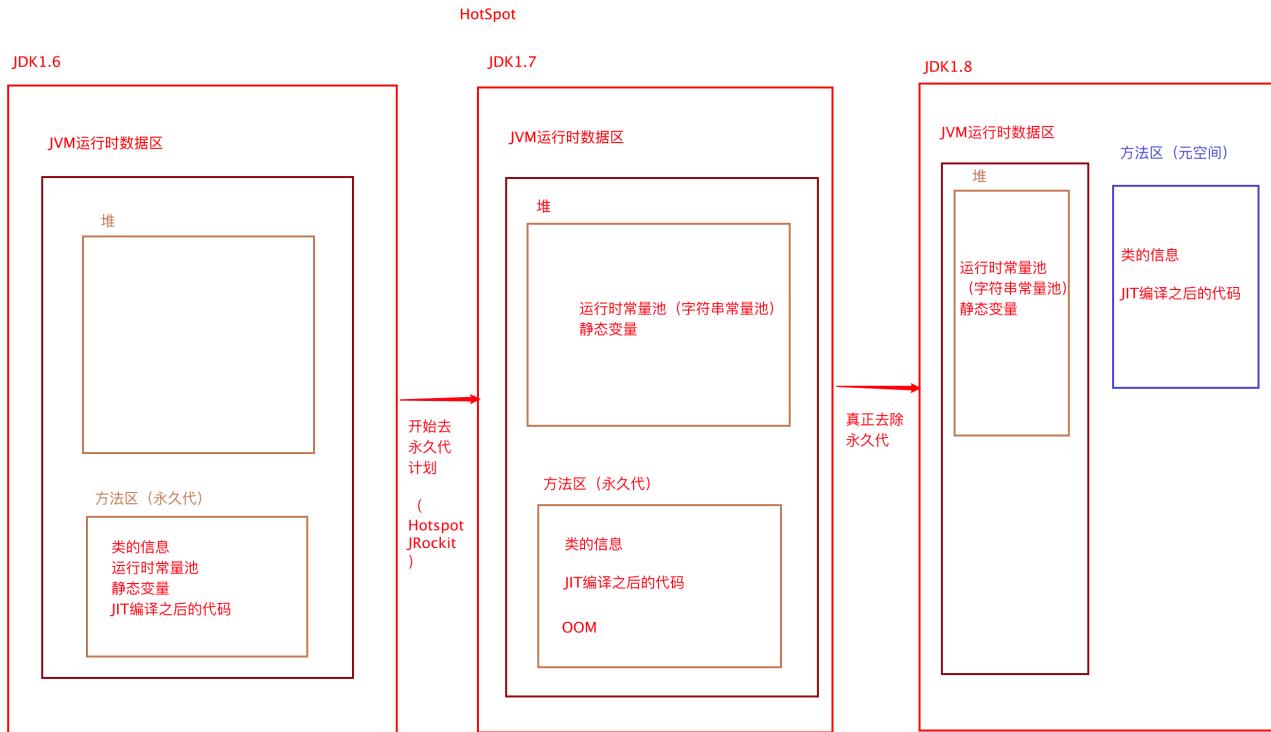
1. 字符串存在永久代中，容易出现性能问题和永久代内存溢出。
2. 类及方法的信息等比较难确定其大小，因此对于永久代的大小指定比较困难，太小容易出现永久代溢出，太大则容易导致老年代溢出。
3. 永久代会为GC带来不必要的复杂度，并且回收效率偏低。
4. Oracle 计划将HotSpot 与 JRockit 合二为一。

## 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：

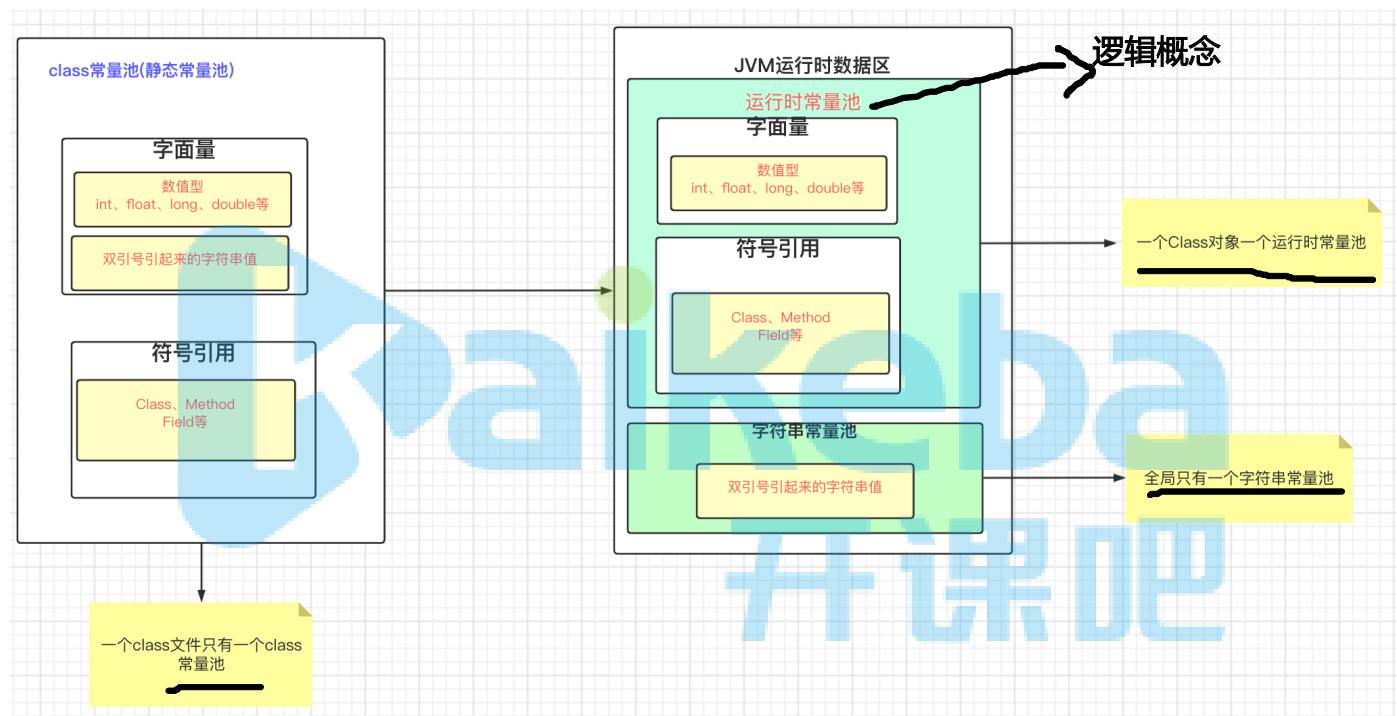
```
luxpdeMacBook-Pro:classes liuxp$ java -version
java version "1.8.0_40"
Java(TM) SE Runtime Environment (build 1.8.0_40-b27)
Java HotSpot(TM) 64-Bit Server VM (build 25.40-b25, mixed mode)
luxpdeMacBook-Pro:classes liuxp$ java -XX:PermSize=8m -XX:MaxPermSize=8m -Xmx16m com.paddx.test.memory.StringOomMock
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：

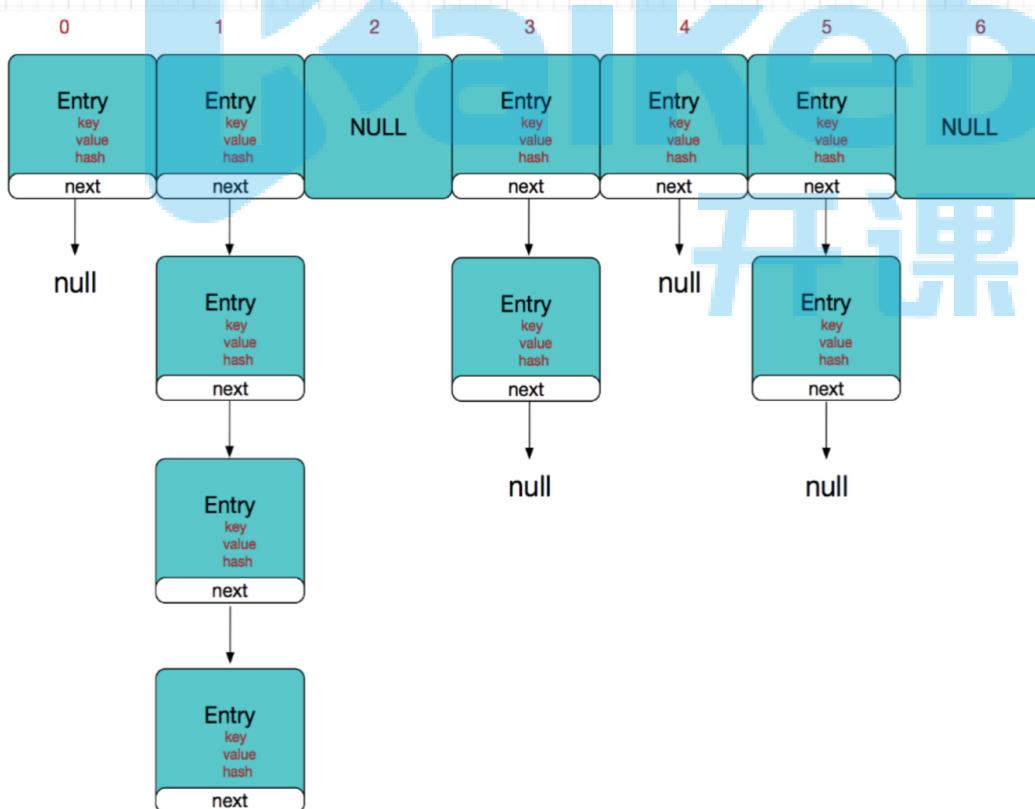
- 根据字符串的 `hashCode` 找到对应entry。
- 如果没冲突，它可能只是一个entry。
- 如果有冲突，它可能是一个entry链表，然后Java再遍历entry链表，匹配引用对应的字符串。
- 如果找得到字符串，返回引用。
- 如果找不到字符串，在使用intern方法的时候，会将intern方法调用者的引用放入stringtable中。

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：  
 $\text{hash}(\text{字符串值}) / \text{数组长度} = \text{余数} (\text{数组下标})$   
 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); f  
  
        String str3 = new String("abc");  
        System.out.println(str3 == str2); f  
  
        String str4 = "a" + "b";  
        System.out.println(str4 == "ab"); t  
  
        final String s = "a";  
        String str5 = s + "b";  
        System.out.println(str5 == "ab"); t  
  
        String s1 = "a";  
        String s2 = "b";  
        String str6 = s1 + s2; → 是否会编译优化?  
        System.out.println(str6 == "ab"); f 运行期间出来的，相当于new  
  
        String str7 = "abc".substring(0, 2); f 相当于new~  
        System.out.println(str7 == "ab");  
  
        String str8 = "abc".toUpperCase(); f 相当于new~  
        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。 new出来的对象的引用放进stringtable中

先让大家做个面试题：

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



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

在字符串常量池中  
stringtable  
找得到

五个对象。 String e = new String("jo") + new String("hn");
System.out.println(e.intern() == e); t

```
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
    }
}
```

字符串常量池

# StringTable

new · John

+  
new "Java"

默认加载了~

```
//      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

jps -l  
jmap 分析~有多少个string

```

public static void main(String[] args) {
    String s = new String("1");
    s.intern(); 这一行相当于啥也没做
    String s2 = "1";
    System.out.println(s == s2); f

    String s3 = new String("1") + new String("1");
    s3.intern(); stringtable的引用          5个对象，只有
                                         是上面new的11.
    String s4 = "11";                      一个new 的
                                         11, 没有常量
    System.out.println(s3 == s4);          池的11
}

}                                     JDK1.7以上是true~。

```

七

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(); 啥也没做，字
                                         符串常量池中
    System.out.println(s == s2);         有了 f

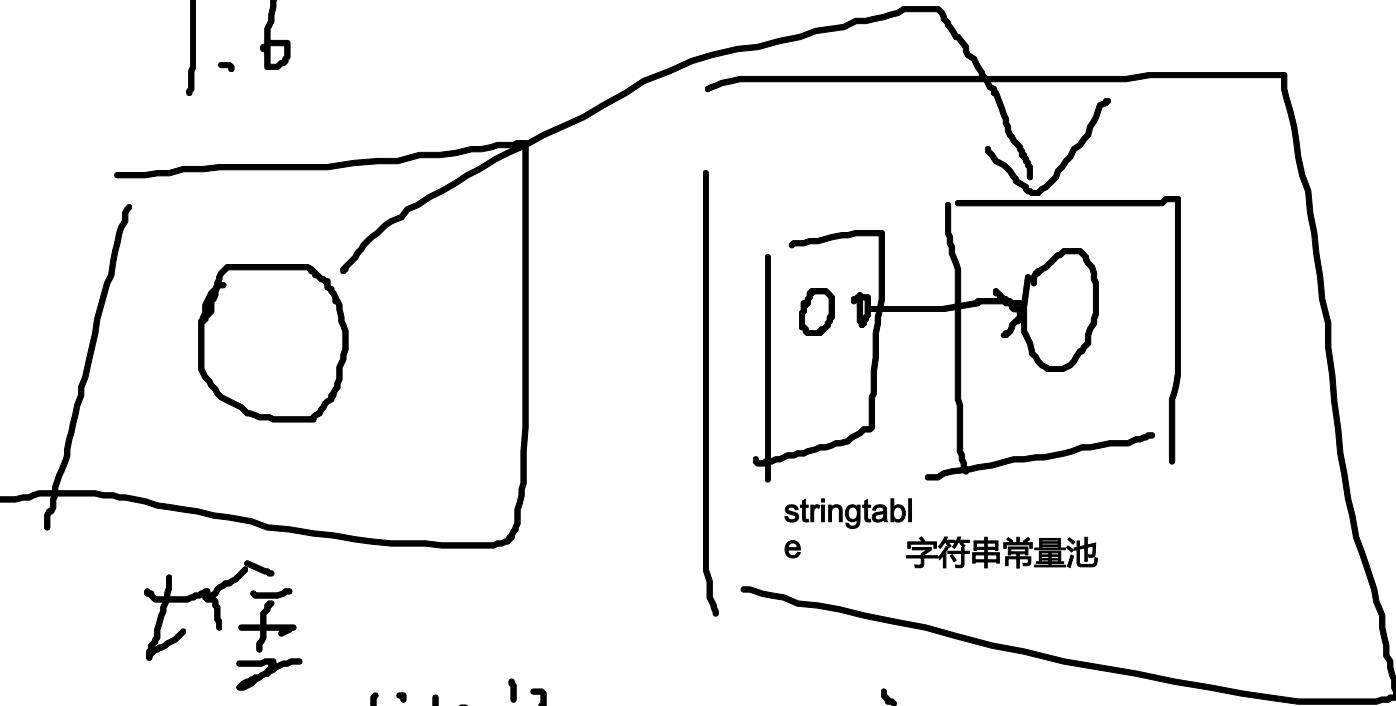
    String s3 = new String("1") + new String("1");
    String s4 = "11"; 字符串常量池中有了11
    s3.intern();
    System.out.println(s3 == s4); f
}

```

## 四、Java堆

Copy

1.6



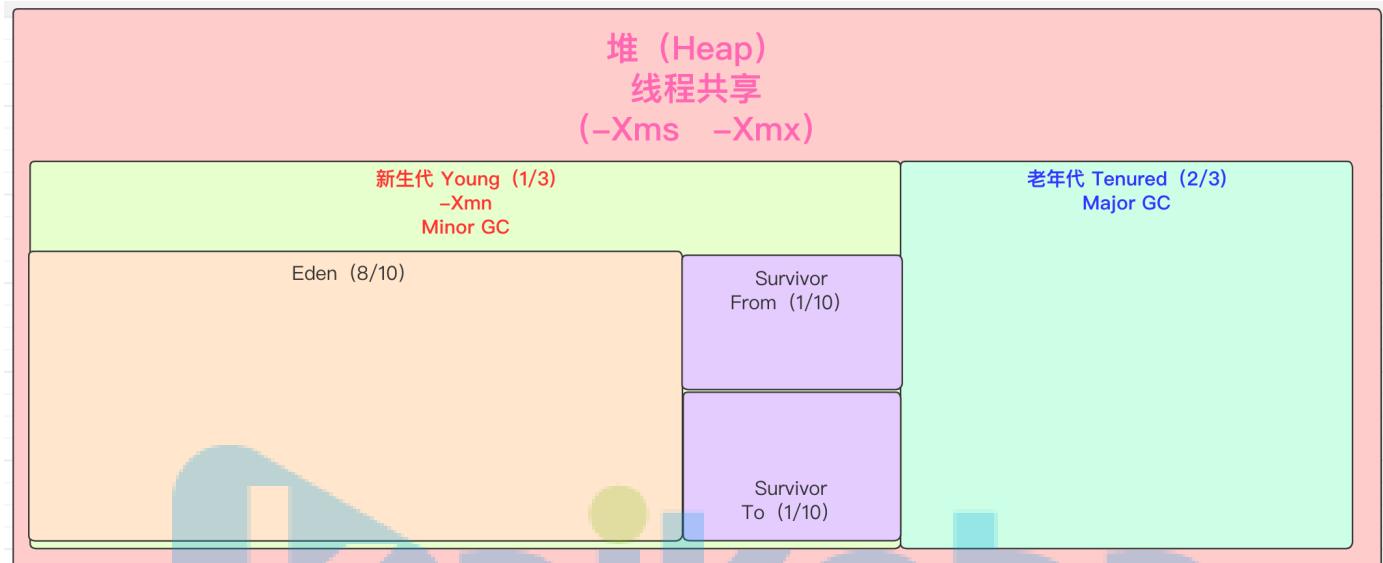
堆  
new "11"

方法

# 堆内存分配

## 堆内存划分

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



## 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)

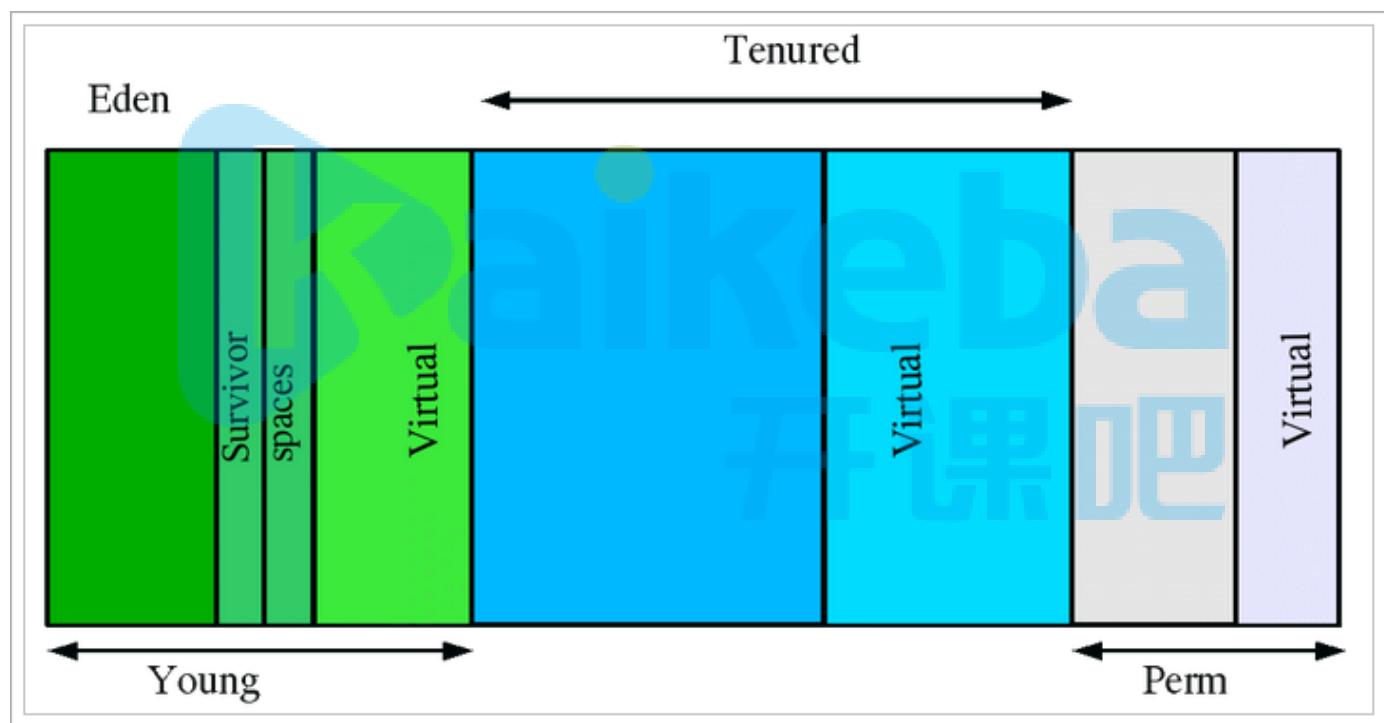
-Xmn: 设置新生代的大小。(初始值及最大值)

-XX:NewRatio: 配置新生代与老年代在堆结构的占比

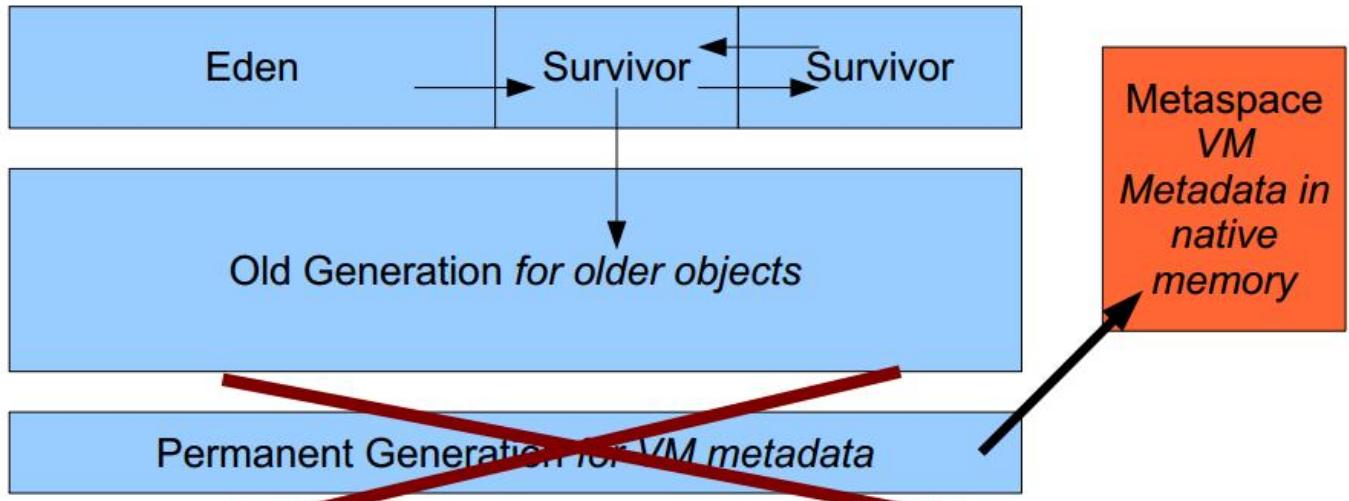
- XX:SurvivorRatio: 设置新生代中Eden和S0/S1空间的比例
- XX:MaxTenuringThreshold 设置新生代垃圾的最大年龄
- XX:+PrintGCDetails: 输出详细的GC处理日志 打印GC简要信息: -XX:PrintGC -verbose:gc
- XX:HandlePromotionFailure: 是否设置空间担保

## 堆内存模型

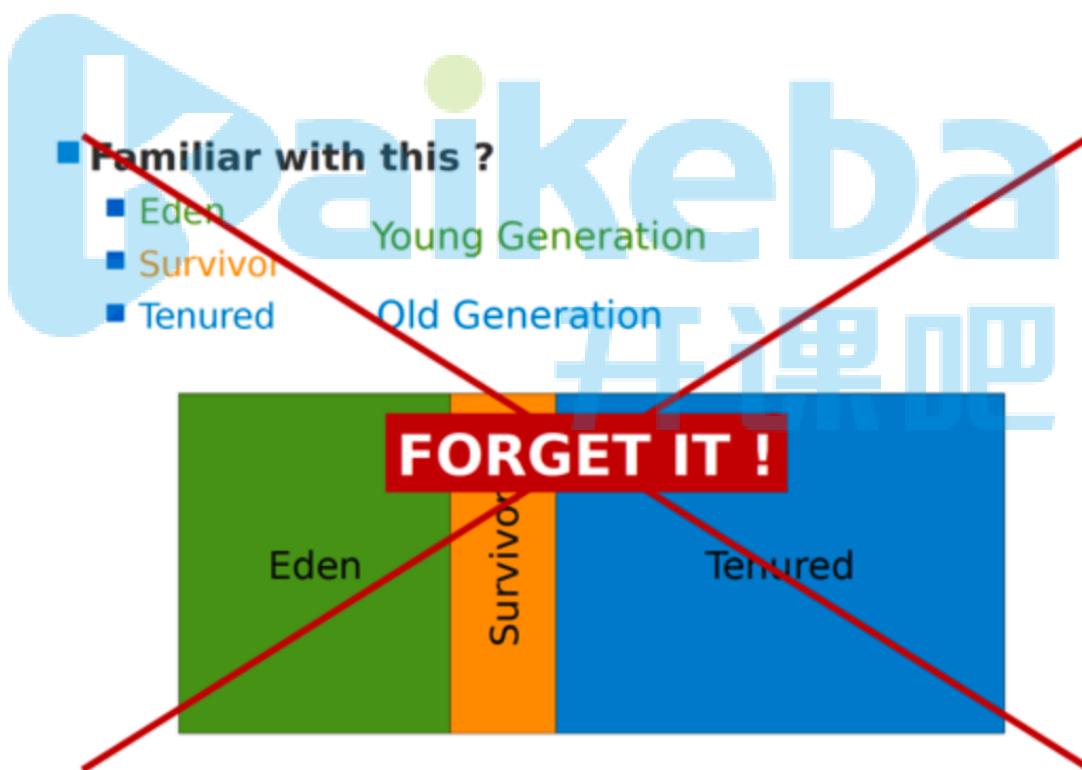
jdk1.7

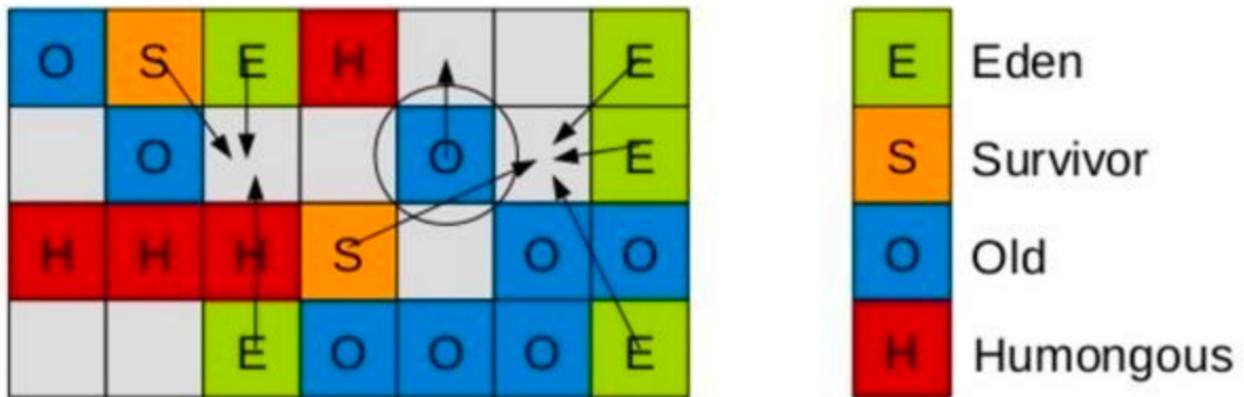


jdk1.8



jdk1.9





## 通过工具查看堆内存信息

### 1、jvisualvm工具

```

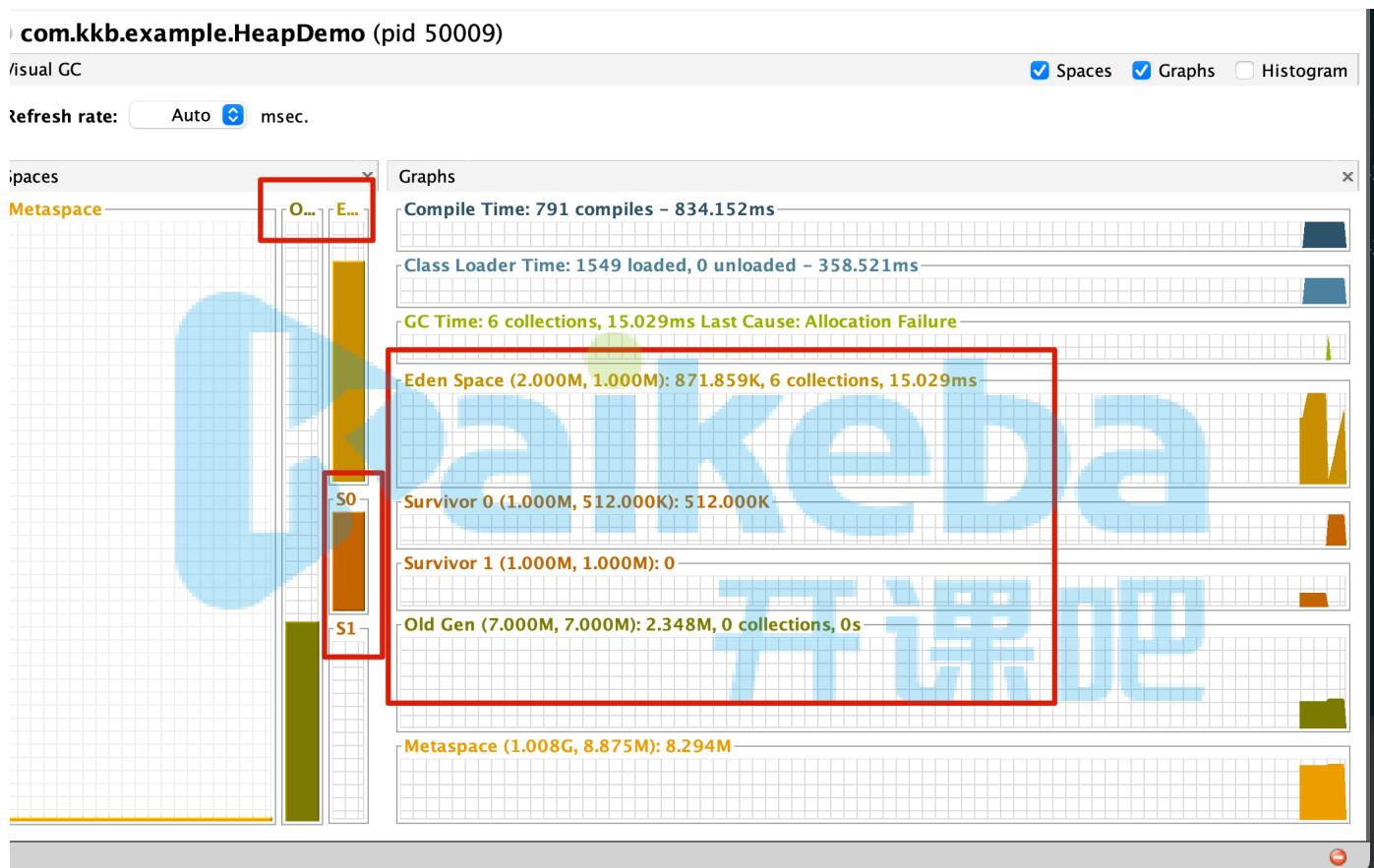
/*
-Xmx600m -Xms600m
*/
public class OOMTest {
    public static void main(String[] args) {
        List<Picture> list = new ArrayList<>();
        while (true){
            try {
                Thread.sleep(20);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            list.add(new Picture(new Random().nextInt(1024
* 1024)));
        }
    }
}

```

```

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

```



## 2、VM options -XX:+PrintGCDetails

```

/*
-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];  
    }  
}
```

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.

## 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

## 对象内存的分配原则

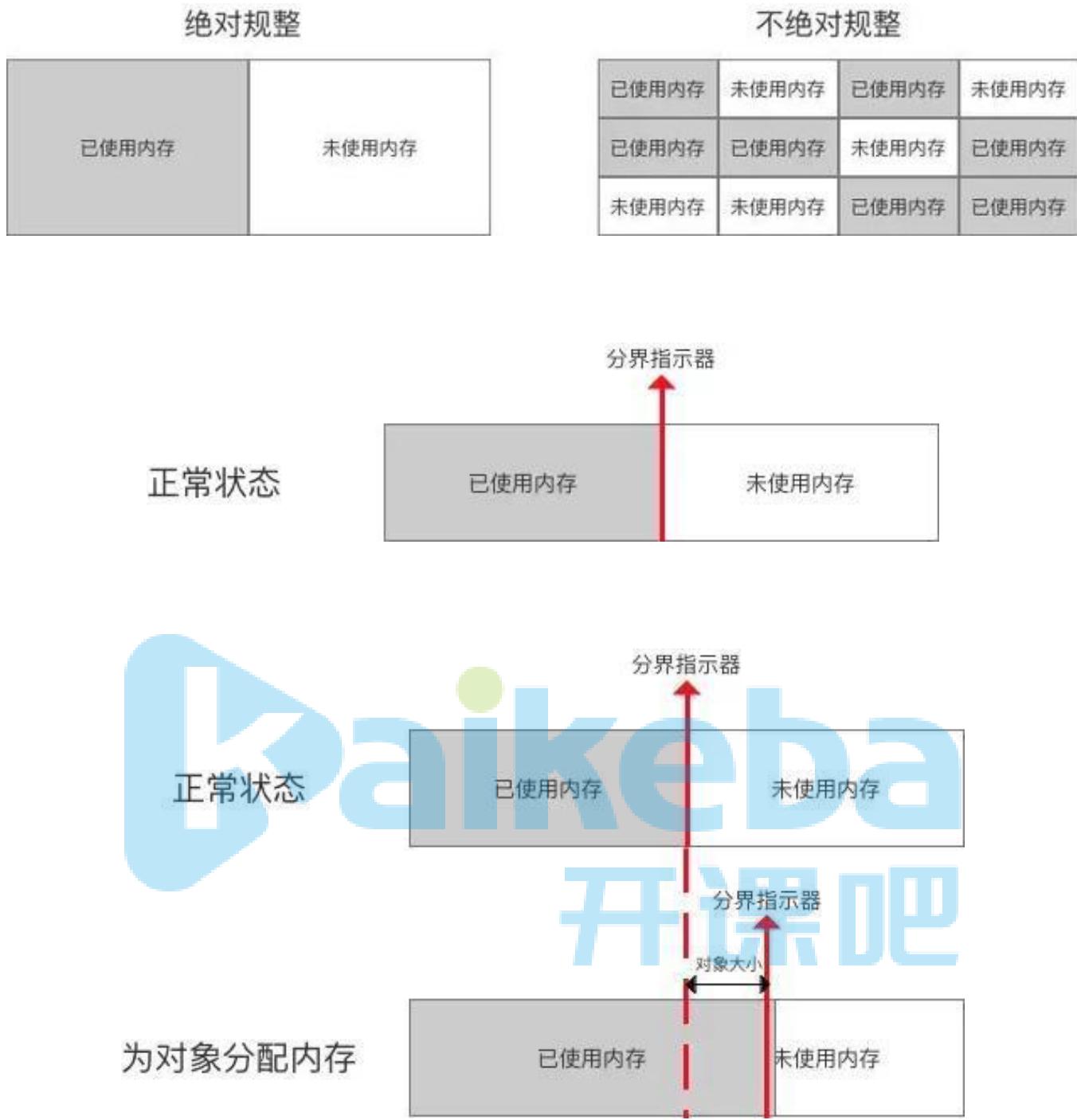
序号	介绍
1	优先在 <code>Eden</code> 分配,如果 <code>Eden</code> 空间不足虚拟机则会进行一次 <code>MinorGC</code>
2	【大对象】直接接入老年代， 【大对象】一般指的是【很长的字符串或数组】，默认是0.
3	【长期存活的对象】也会进入老年代，每个对象都有一个 <code>【age】</code> ，当 <code>age</code> 到达设定的年龄的时候就会进入老年代，默认是15岁。

## 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）	<code>Serial</code> 和 <code>ParNew</code> 收集器
空闲列表	内存地址不连续（年老代）	<code>CMS</code> 收集器和 <code>Mark-Sweep</code> 收集器



## 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年代，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



## 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，JDK1.6之后就默认开启。

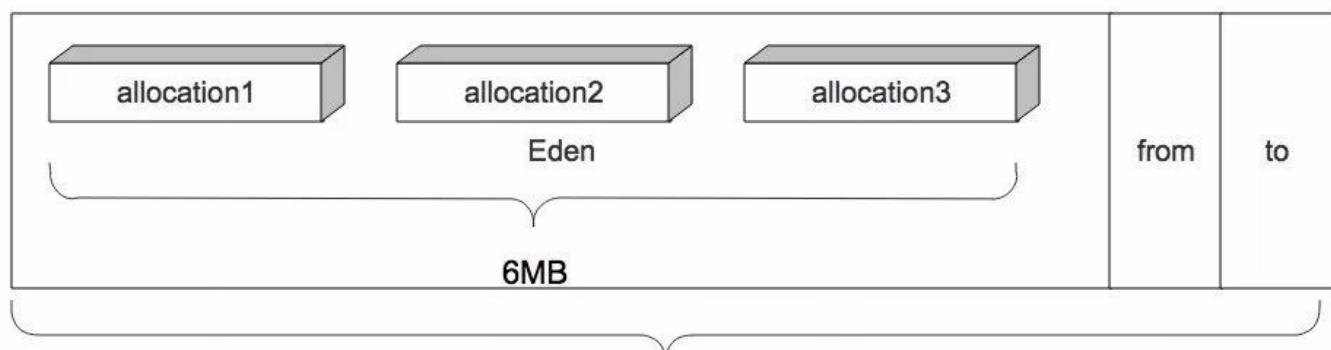
这里我们使用的是JDK1.8，所以不用再手动去开启担保机制。

## 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。

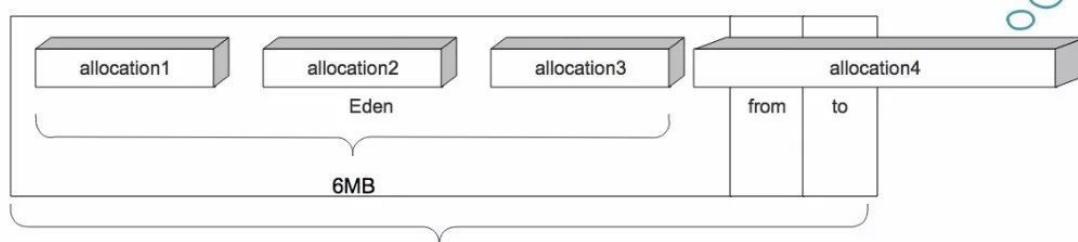
## 分析过程

担保前的堆空间：



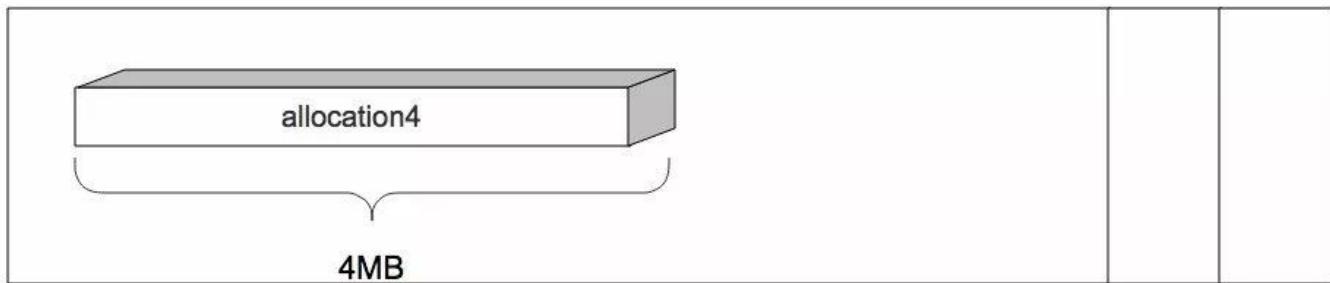
新生代 (10240KB)

发生Minor GC，触发担保机制：



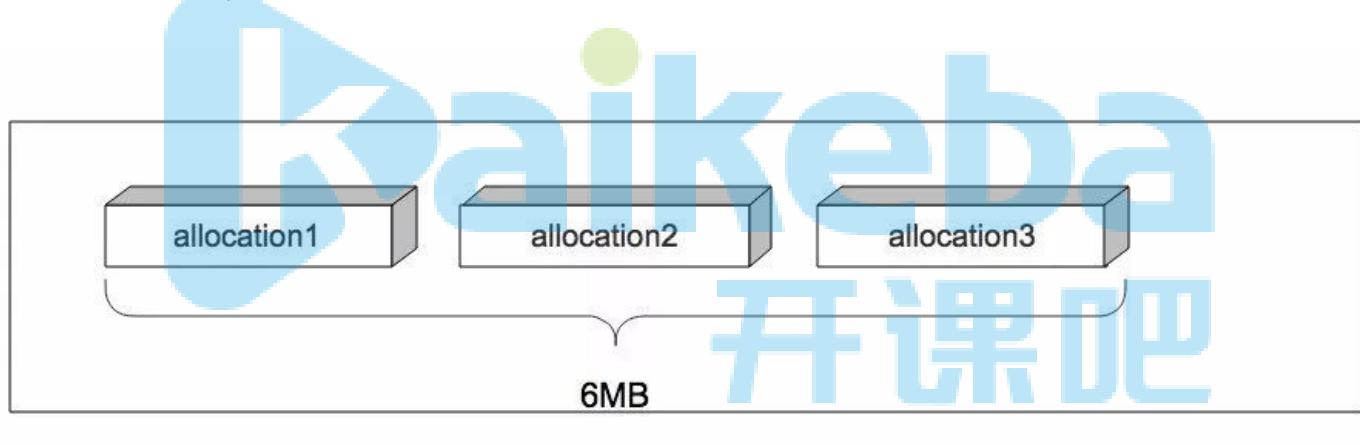
新生代 (10240KB)

担保后的新生代：



新生代 (10240KB)

担保后的老年代:



老生代 (10240KB)

## 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：

## 并行垃圾收集器案例结论

| 发现当我们使用ParallelGC收集器组合（Parallel Scavenge+Serial Old的组合）下，担保机制的实现和之前的Client模式下（SerialGC收集器组合）有所变化。

| 注意点：

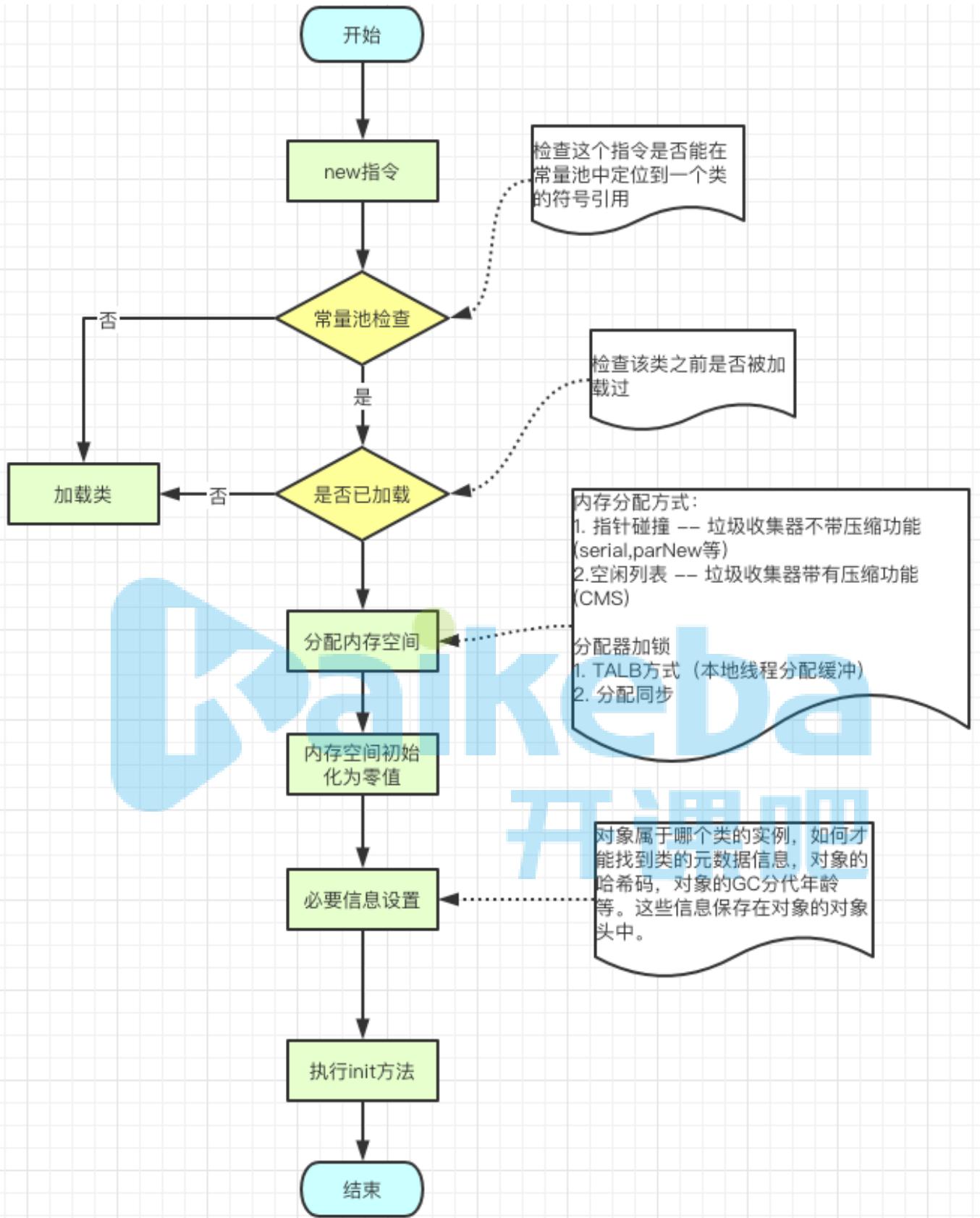
在GC前还会进行一次判断，如果要分配的内存 $\geq$ Eden区大小的一半，那么会直接把要分配的内存放入老年代中。否则才会进入担保机制。

## 对象创建与访问

### 对象创建

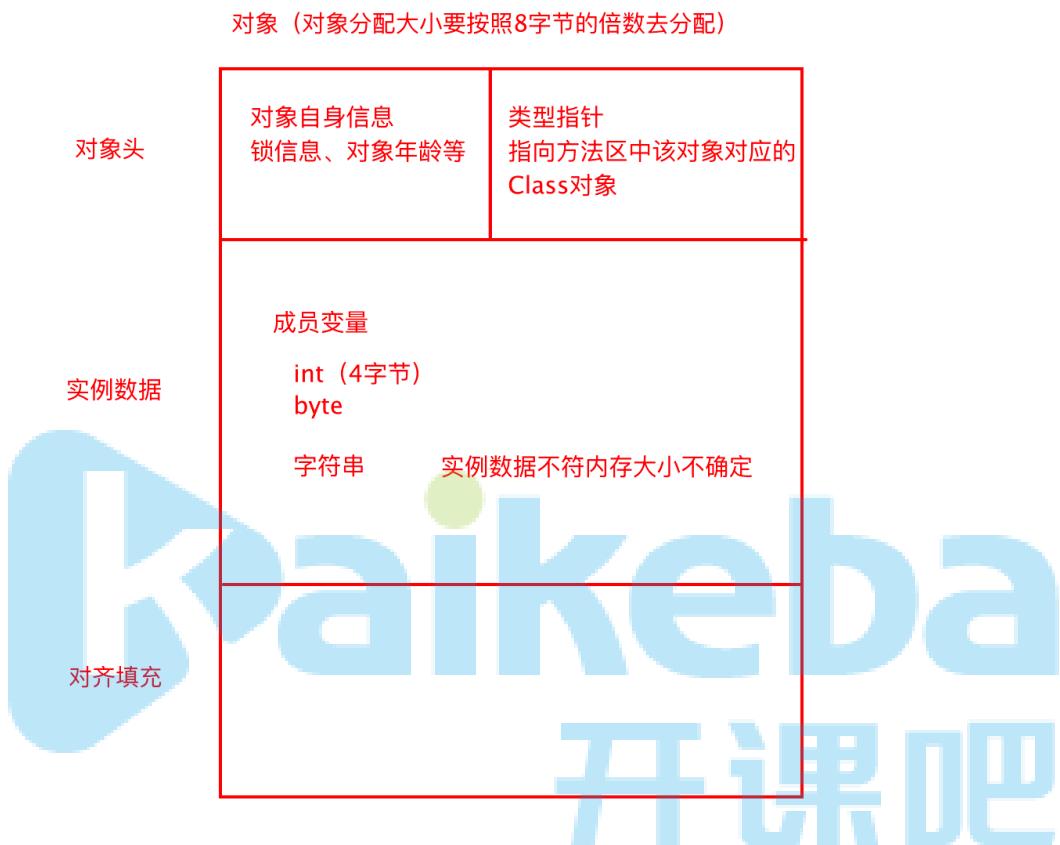
```
Student stu = new Student();
```





# 对象的内存布局

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 \(Padding\)](#)。



## 1) 对象头

对象头包括两部分信息：

一部分是用于[存储对象自身的运行数据](#)，如[哈希码 \(HashCode\)](#)，[GC分代年龄](#)，[锁状态标志](#)，[线程持有的锁](#)，[偏向线程ID](#)，[偏向时间戳](#)等。

另一部分是[类型指针](#)，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪一个类的实例。当对象是一个java数组的时候，那么对象头还必须有一块用于记录数组长度的数据，因此虚拟机可以通过普通java对象的元数据信息确定java对象的大小，但是从数组的元数据中无法

确定数组的大小。

## 2) 实例数据

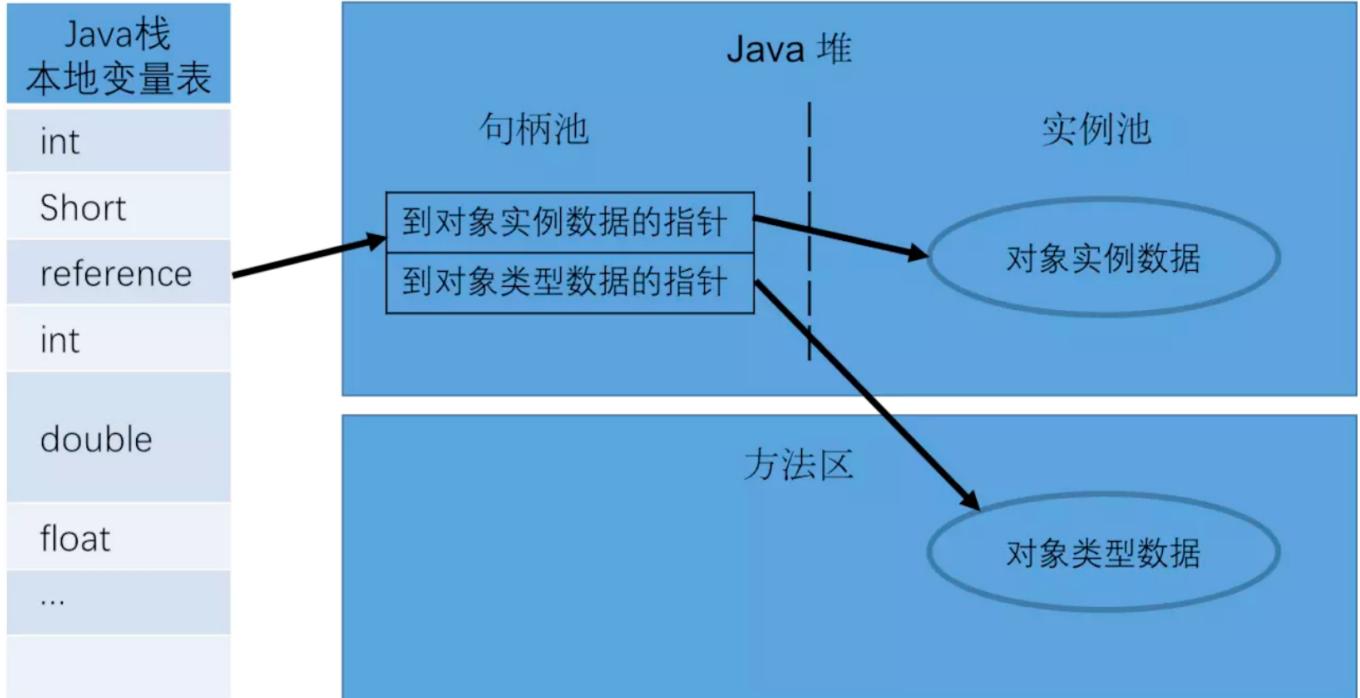
存储的是对象真正有效的信息。

## 3) 对齐填充

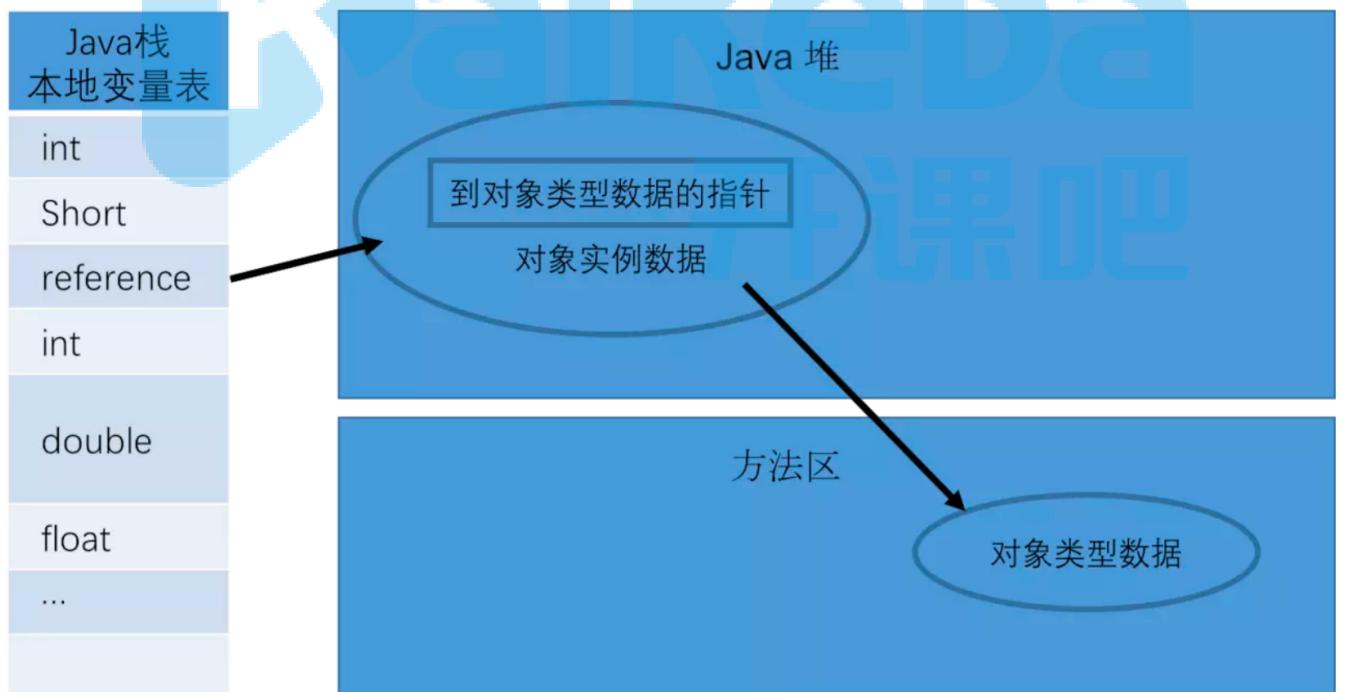
这部分并不是必须要存在的，没有特别的含义，在jvm中对象的大小必须是8字节的整数倍，而对象头也是8字节的倍数，当对象实例数据部分没有对齐时，就需要通过对齐填充来补全。

## 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



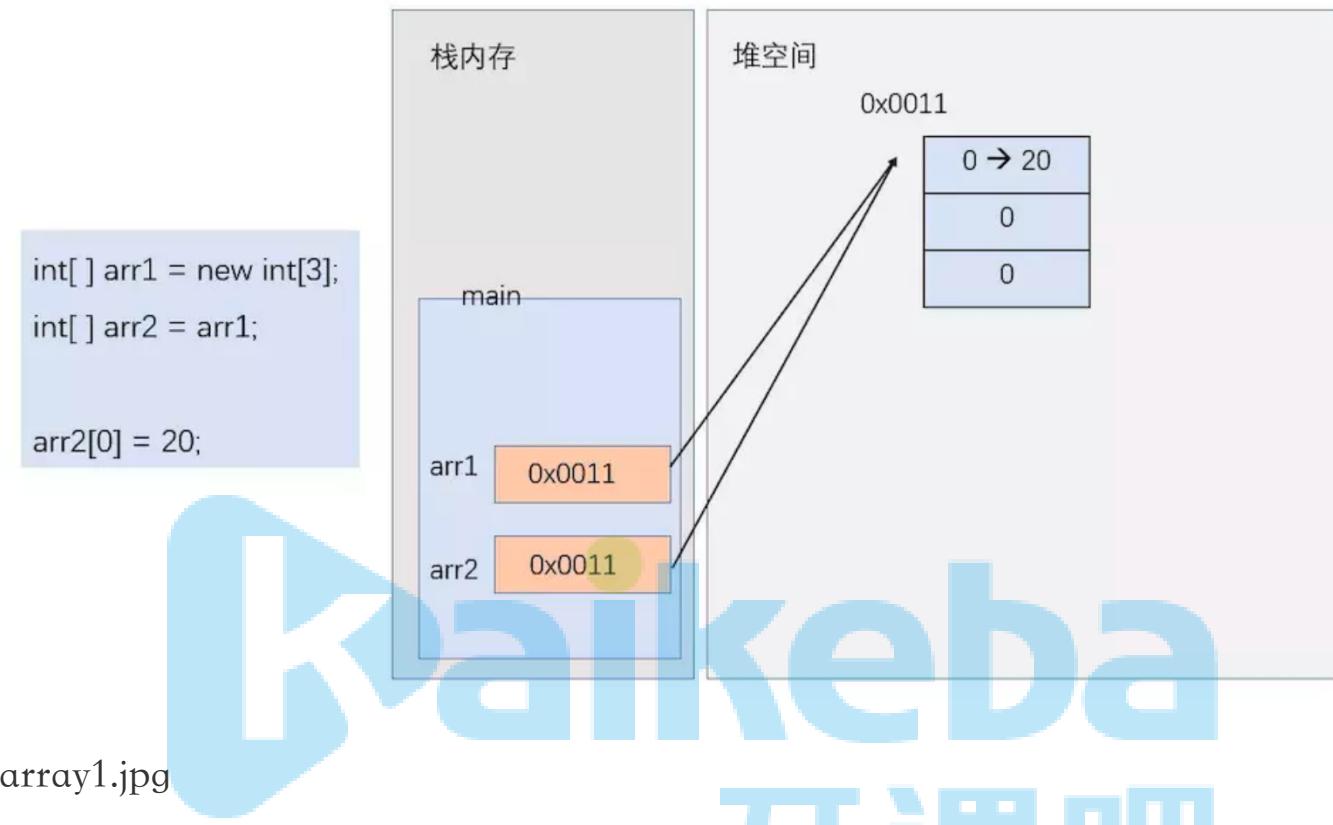
### 通过句柄访问对象



### 通过直接指针访问对象

# 数组的内存分析

## 一维数组



先把 `arr1` 压进栈，然后在堆空间中开辟一个空间，并把值初始化为 0 (`arr1` 为引用变量，但是内部数据是 `int` 类型，默认值为 0)，最后把开辟的堆空间地址 赋值给 `arr1`

int[ ] arr2 = arr1;

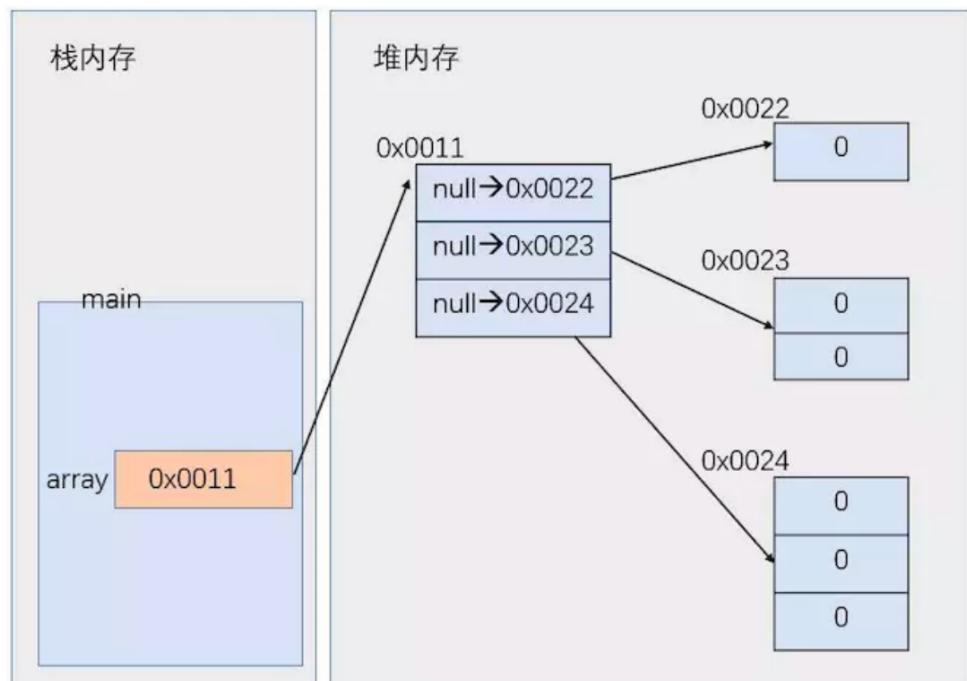
把 `arr1` 中的 地址 赋值给 `arr2`，此时 `arr2` 和 `arr1` 指向同一块空间。

arr2[0] = 20;

此时，`arr1[0]` 值为 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];
```



```
int[ ][ ] array = new int[3][];
```

这条语句会先把 array 压栈，然后在堆中开辟一个空间，初始值为 null (array 为引用变量，第一维同样是引用类型)，最后把开辟的堆空间地址赋值给 array。

```
array[0][ ] = new int[1]
```

这条语句会在堆空间中开辟一个只有一个 int 类型大小的空间，并初始化为 0，然后把自己的地址赋值给 array[0][ ]。

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

这两条语句和上一条意义一样，就不再做解释

# 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
```

#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

#### 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](#)

Object Query Language (OQL) query

All Classes (excluding platform) OQL Help

```
select s from java.lang.String s where s.value.length >= 10000
```

查询字符串长度大于10000的内容

Execute

java.lang.String@0xe321a8d0  
java.lang.String@0xe338e038  
java.lang.String@0xe3684f98  
java.lang.String@0xe359f8c0  
java.lang.String@0xe329f458

查询到的结果

