课程主题

运行时数据区概述及方法区详解&运行时常量池&字符串常量池

课程目标

1. 类加载中的双亲委派模型
2. 类加载中的破坏双亲委派模型
3. 掌握JVM规范中的运行时数据区
4. 掌握不同JDK版本中的运行时数据区
5. 掌握不同JDK版本中运行时数据区的内存划分情况
6. 掌握不同版本的方法区的实现是什么
7. 掌握永久代和元空间的区别
8. 方法区的异常演示
9. 运行时常量池
10. 字符串常量池

课程回顾

1. class类文件概述
2. class常量池详解
3. 分析javap命令显示出来的结果
4. 类加载的过程
5. 类加载的时机
6. 类加载器

课程内容

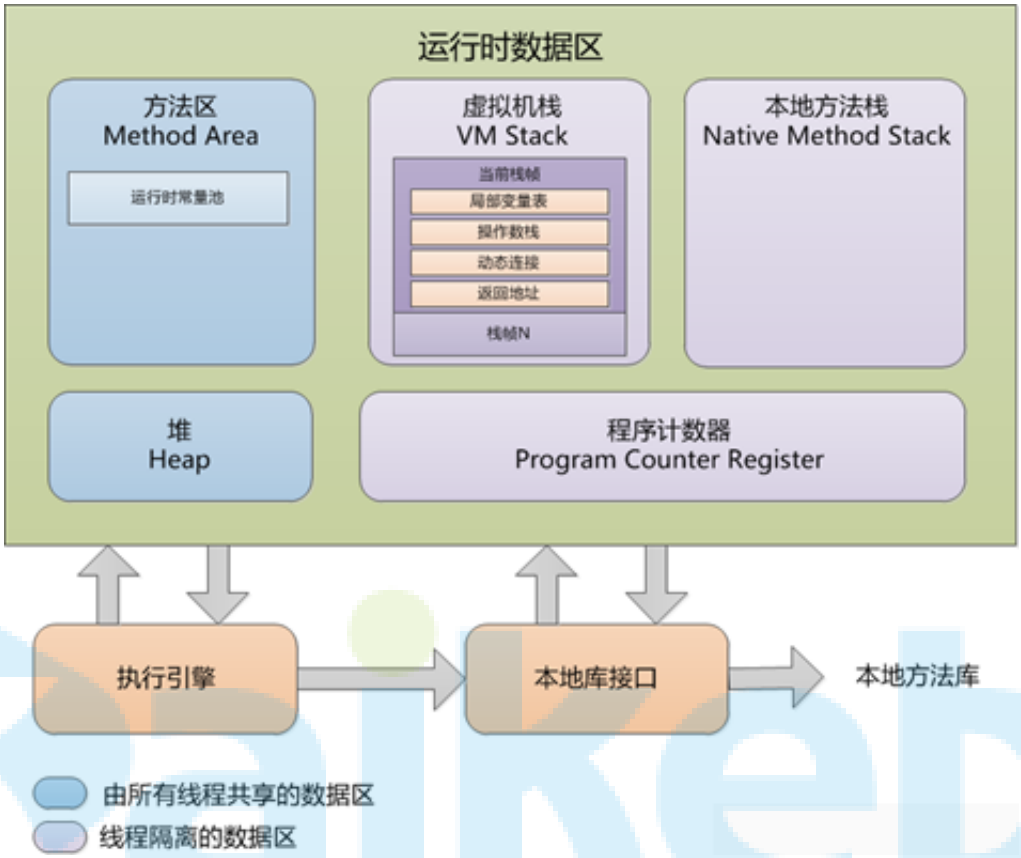
类加载机制

- 双亲委派模型（自上而下的去进行类加载）
  - 保证核心类的安全（String）
- 破坏双亲委派模型
  - 在上层类加载器去加载一个类的时候，去委托子类完成其他类的加载（破坏双亲委派模型）
  - OSGI（热加载技术-->动态模块化）
  - SPI机制（Driver加载、springboot、dubbo）
    - classpath:/META-INF/以接口全路径为名称的文件（内容是第三方的实现类）
      - ServiceLoader 接口类型

运行时数据区

概述

JVM运行时数据区规范



线程共享区域：JVM启动的时候，这块区域就开始分配空间。

线程私有区域：没有线程的时候，这块区域是不存在的。这块空间的生命周期特别短暂，不存在垃圾回收的问题

运行时数据区的使用顺序：

```
方法区（类信息、常量池信息等）==> 堆（对象或者数组）==>虚拟机栈、程序计数器、本地方法栈（不是必须的）
```

Hotspot运行时数据区

见图

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）
```

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
```

方法区

存储内容

1、类型信息

- 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）

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）和对其他类型、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）。

3、字段信息（该类声明的所有字段）

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

4、方法信息

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

5、类变量（静态变量）

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。

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。

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。

8、方法表

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。

9、运行时常量池(Runtime Constant Pool)

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）。

永久代和元空间的区别

永久代和元空间存储位置和存储内容的区别：

> 存储位置不同，永久代物理是是堆的一部分，和新生代，老年代地址是连续的，而元空间属于本地内存；
> 由于永久代它的大小是比较小的，而元空间的大小是决定于内地内存的。
> 所以说永久代使用不当，比较容易出现OOM异常。而元空间一般不会。
>
> 存储内容不同，元空间存储类的元信息，[静态变量]和[常量池]等并入堆中。相当于永久代的数据被分到了堆和元空间中。

为什么要把永久代替换为元空间?

1.原来Java是属于Sun公司的，后来Java被Oracle收购了。Sun公司实现的Java中的JVM是Hotspot。当时Oracle堆Java的JVM也有一个实现，交JRockit。后来Oracle收购了Java之后，也同时想把Hotspot和JRockit合二为一。他们俩很大的不同，就是方法区的实现。

2.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。

3.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。
4.永久代会为 GC 带来不必要的复杂度，并且回收效率偏低。

方法区异常演示

类加载导致OOM异常

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：

需要被类加载器加载的测试类：

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

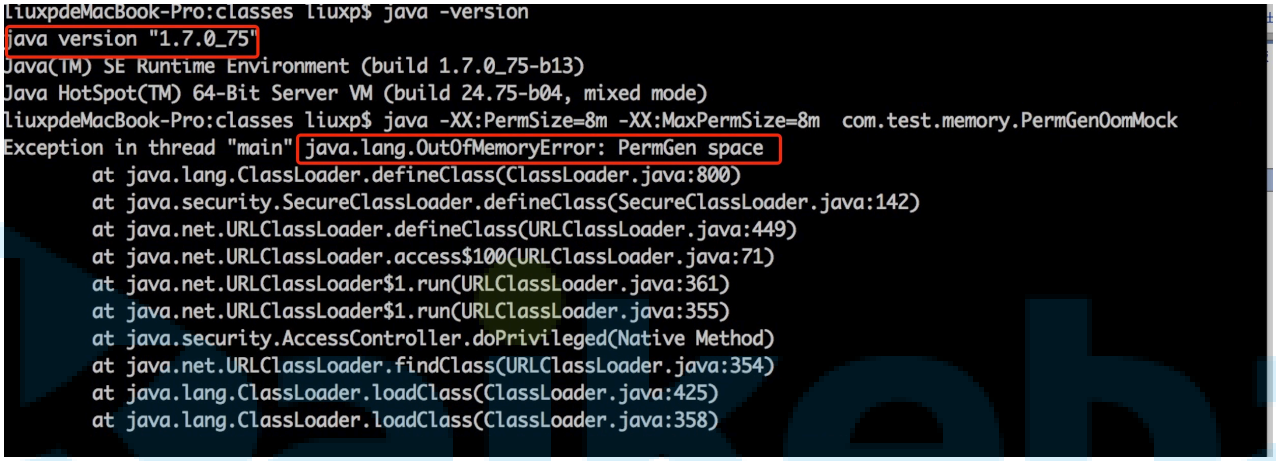
测试代码（*使用不同的类加载器对象对以上的类进行加载*）：

```
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);
                //每个ClassLoader对象，对同一个类进行加载，会产生不同的Class对象
                loader.loadClass("com.kkb.test.memory.Test");
```

```
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

JDK1.7分析

指定的 `PermGen` 区的大小为 8M。



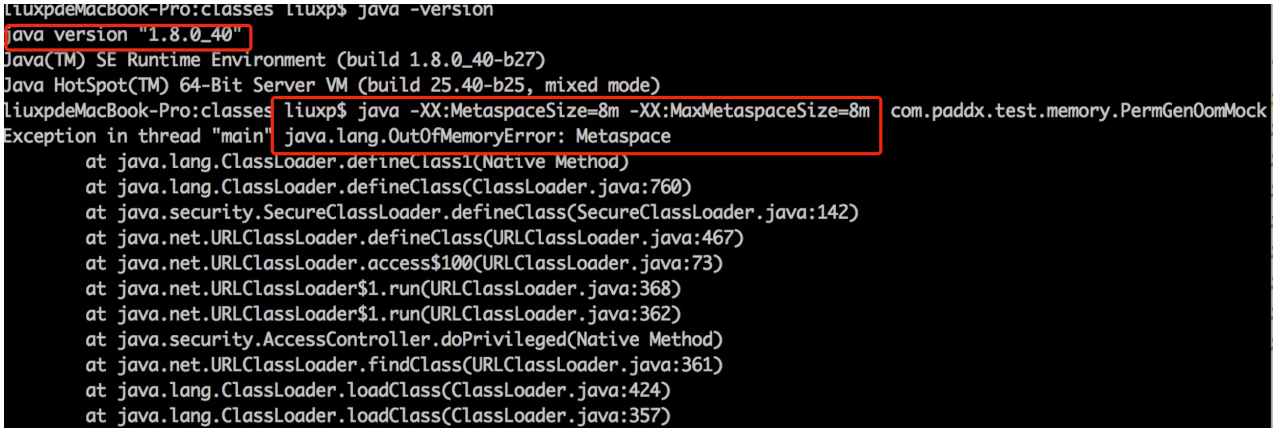
绝大部分 Java 程序员应该都见过 "java.lang.OutOfMemoryError: PermGen space "这个异常。这里的“`PermGen space`”其实指的就是方法区。由于方法区主要存储类的相关信息，所以对于动态生成类的情况比较容易出现永久代的内存溢出。

最典型的场景就是，在 jsp 页面比较多的情况，容易出现永久代内存溢出。

```
JSP页面，需要动态生成Servlet类class文件
```

JDK1.8+

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：



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。

字符串OOM异常

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());
        }
    }
}
```

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)
```

JDK 1.6下，会出现永久代的内存溢出。

JDK1.7

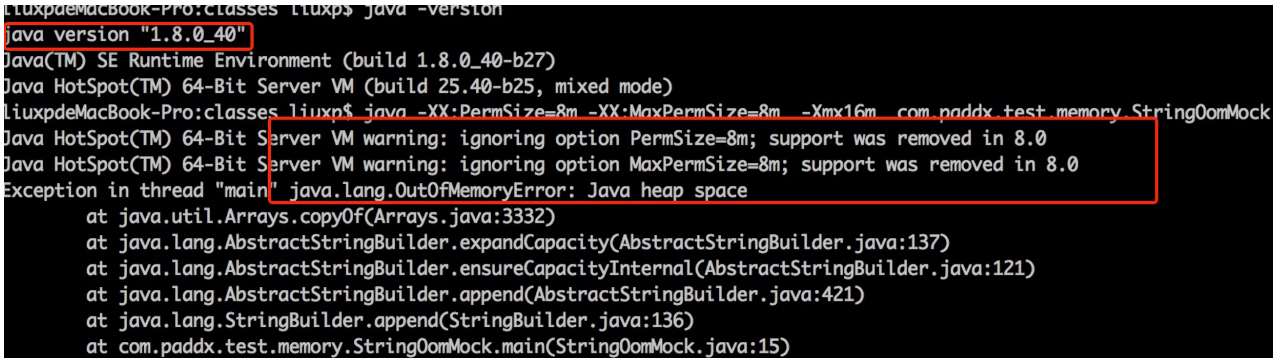
JDK 1.7的运行结果：

```
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  -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。

JDK1.8+

JDK 1.8的运行结果：



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。

运行时常量池和字符串常量池

存储内容

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，用于存放编译期生成的各种**字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中存放。**

- 字面量：
  - 双引号引起来的字符串值，“kkb”
  - 定义为final类型的常量的值。
- 符号引用：
  - 类或接口的全限定名（包括他的父类和所实现的接口）
  - 变量或方法的名称
  - 变量或方法的描述信息
    - 方法的描述：参数个数、参数类型、方法返回类型等等
    - 变量的描述信息：变量的返回值
  - this

运行时常量池相对于Class文件常量池的另外一个重要特征是**具备动态性**，Java语言并不要求常量一定只有编译期才能产生，也就是并非预置入Class文件中常量池的内容才能进入方法区运行时常量池，**运行期间也可能将新的常量放入池中，这种特性被开发人员利用比较多的就是String类的intern()方法。**

存储位置

在JDK1.6及以前，运行时常量池是方法区的一部分。

在JDK1.7及以后，运行时常量池在Java 堆（Heap）中。

运行时和class常量池一样，运行时常量池也是每个类都有一个。但是字符串常量池全只有一个

常量池区别

class常量池（静态常量池）、运行时常量池、字符串常量池区别：

- class常量池中存储的是符号引用，而运行时常量池存储的是被解析之后的直接引用。
- class常量池存在于class文件中，运行时常量池和字符串常量池是存在于JVM内存中。
- 运行时常量池具有动态性，java运行期间也可能将新的常量放入池中(**String#intern()**)，
- 字符串常量池逻辑上属于运行时常量池的一部分，但是它和运行时常量池的区别在于，字符串常量池是全局唯一的，而运行时常量池是每个类一个。

字符串常量池如何存储数据

实际上，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。

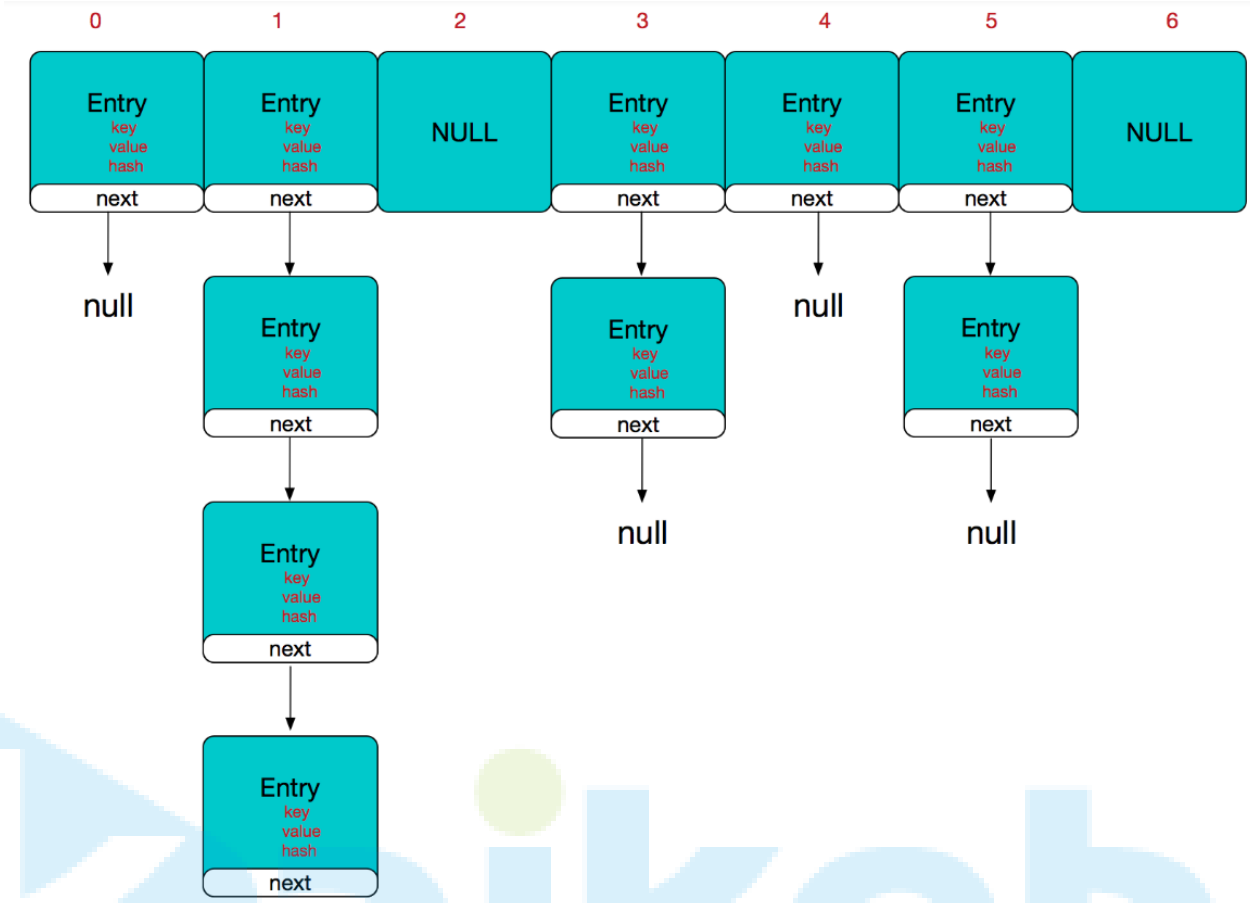
在jdk6中`StringTable`的长度是固定的，就是1009，因此如果放入String Pool中的String非常多，就会造成hash冲突，导致链表过长。此时当调用`String.intern()`时会需要到链表上一个一个找，从而导致性能大幅度下降；

在jdk7+，`StringTable`的长度可以通过一个参数指定：

```
-XX:StringTableSize=99991
```

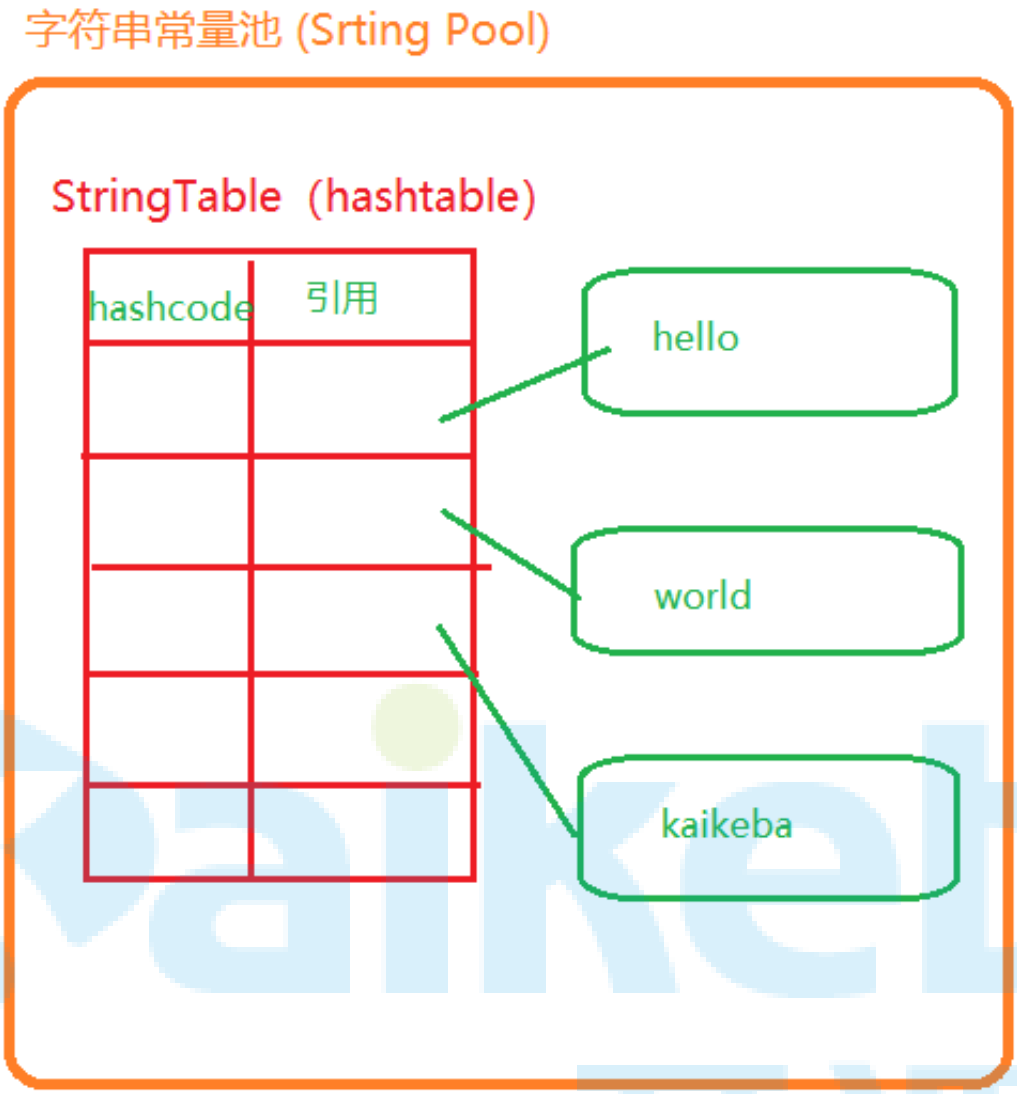
hash冲突问题：形成链表（增删快、查找慢的数据结构）

stringtable是类似于hashtable的数据结构，hashtable数据结构如下：



intern方法在进行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，会把字符串放到常量池，并把引用保存到stringtable里。



字符串常量池介绍

上面我们已经稍微了解过字符串常量池了，它是java为了节省空间而设计的一个内存区域，java中所有的类共享一个字符串常量池。

> 比如A类中需要一个“hello”的字符串常量，B类也需要同样的字符串常量，他们都是从字符串常量池中获取的字符串，并且获得得到的字符串常量的地址是一样的。

字符串常量池案例分析

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。

```
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 s3 = "ab";
    String s4 = "ab" + getString();
    System.out.println(s3 == s4);

        String s5 = "a";
```

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

  private String getString(){
    return "c";
  }

}
```

分析1

```
String str1 = "abc";
System.out.println(str1 == "abc");
```

步骤：
1）栈中开辟一块空间存放引用str1，
2）String池中开辟一块空间，存放String常量"abc"，
3）引用str1指向池中String常量"abc"，
4）str1所指代的地址即常量"abc"所在地址，输出为true

分析2

```
String str2 = new String("abc");
System.out.println(str2 == "abc");
```

步骤：
1）栈中开辟一块空间存放引用str2，
2）堆中开辟一块空间存放一个新建的String对象"abc"，
3）引用str2指向堆中的新建的String对象"abc"，
4）str2所指代的对象地址为堆中地址，而常量"abc"地址在池中，输出为false

分析3

```
String str2 = new String("abc");
String str3 = new String("abc");
System.out.println(str3 == str2);
```

步骤：
1）栈中开辟一块空间存放引用str3，
2）堆中开辟一块新空间存放另外一个(不同于str2所指)新建的String对象，
3）引用str3指向另外新建的那个String对象
4）str3和str2指向堆中不同的String对象，地址也不相同，输出为false

分析4

```
String str4 = "a" + "b";
System.out.println(str4 == "ab");
```

步骤：
1）栈中开辟一块空间存放引用str4，
2）根据编译器合并已知量的优化功能，池中开辟一块空间，存放合并后的String常量"ab"，
3）引用str4指向池中常量"ab"，
4）str4所指即池中常量"ab"，输出为true

分析5

```
final String s = "a";
String str5 = s + "b";
System.out.println(str5 == "ab");
```

步骤：
同4

分析6

```
String s1 = "a";
String s2 = "b";
String str6 = s1 + s2;
System.out.println(str6 == "ab");
```

步骤：
1）栈中开辟一块中间存放引用s1，s1指向池中String常量"a"，
2）栈中开辟一块中间存放引用s2，s2指向池中String常量"b"，
3）栈中开辟一块中间存放引用str6，
4）s1 + s2通过StringBuilder的最后一步toString()方法还原一个新的String对象"ab"，因此堆中开辟一块空间存放此对象，
5）引用str6指向堆中(s1 + s2)所还原的新String对象，
6）str6指向的对象在堆中，而常量"ab"在池中，输出为false

分析7

```
String str7 = "abc".substring(0,  2);
System.out.println(str7 == "ab");
```

步骤：
1）栈中开辟一块空间存放引用str7，
2）substring()方法还原一个新的String对象"ab"（不同于str6所指），堆中开辟一块空间存放此对象，
3）引用str7指向堆中的新String对象，

分析8

```
String str8 = "abc".toUpperCase();
System.out.println(str8 == "ABC");
```

步骤：
1）栈中开辟一块空间存放引用str8，
2）toUpperCase()方法还原一个新的String对象"ABC"，池中并未开辟新的空间存放String常量"ABC"，
3）引用str8指向堆中的新String对象

String的Intern方法详解

先让大家做个面试题：

```
        String a = "hello";
        String b = new String("hello");
        System.out.println(a == b);

        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);
```

如果大家能一题不差的全做对，接下来的内容应该不用看了。如果不能并且有兴趣的话，可以稍微了解一下以下

内容。

```
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);
  }
}
```

intern的作用

intern的作用是在运行期间把new出来的字符串的引用添加到stringtable中，java会先计算string的hashcode，查找stringtable中是否已经有string对应的引用了，如果有返回引用（地址），然后没有把字符串的地址放到stringtable中，并返回字符串的引用（地址）。

我们继续看例子：

```
        String a = new String("haha");
        System.out.println(a.intern() == a);//false
```

> 因为有双引号括起来的字符串，所以会把ldc命令，即"haha"会被我们添加到字符串常量池，它的引用是string的char数组的地址，会被我们添加到stringtable中。所以a.intern的时候，返回的其实是string中的char数组的地址，和a的string实例化地址肯定是不一样的。

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

> new String("jo") + new String("hn")实际上会转为stringbuffer的append 然后tosring()出来，实际上是new 一个新的string出来。在这个过程中，并没有双引号括起john，也就是说并不会执行ldc然后把john的引用添加到stringtable中，所以intern的时候实际就是把新的string地址（即e的地址）添加到stringtable中并且返回回来。

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

> 或许很多朋友感觉很奇怪，这跟上面的例子2基本一模一样，但是却是false呢？这是因为java在启动的时候，会把一部分的字符串添加到字符串常量池中，而这个“java”就是其中之一。所以intern回来的引用是早就添加到字符串常量池中的”java“的引用，所以肯定跟f的原地址不一样。

JDK6中的理解

Jdk6中字符串常量池位于PermGen（永久代）中，PermGen是一块主要用于存放已加载的类信息和字符串池的大小固定的区域。

执行intern()方法时，若常量池中不存在等值的字符串，JVM就会在常量池中创建一个等值的字符串，然后返回该字符串的引用。除此以外，JVM 会自动在常量池中保存一份之前已使用过的字符串集合。

Jdk6中使用intern()方法的主要问题就在于字符串常量池被保存在PermGen中：

> - **首先，PermGen是一块大小固定的区域，一般不同的平台PermGen的默认大小也不相同，大致在32M到96M之间。**所以不能对不受控制的运行时字符串（如用户输入信息等）使用intern()方法，否则很有可能会引发PermGen内存溢出；
> - **其次String对象保存在Java堆区，Java堆区与PermGen是物理隔离的，因此如果对多个不等值的字符串对象执行intern操作，则会导致内存中存在许多重复的字符串，会造成性能损失。**

JDK7+的理解

Jdk7将常量池从PermGen区移到了Java堆区。堆区的大小一般不受限，所以将常量池从PremGen区移到堆区使得常量池的使用不再受限于固定大小。可以使用 `-XX:StringTableSize` 虚拟机参数设置字符串池的map大小。

> 字符串池内部实现为一个HashMap，所以当能够确定程序中需要intern的字符串数目时，可以将该map的size设置为所需数目*2（减少hash冲突），这样就可以使得String.intern()每次都只需要常量时间和相当小的内存就能够将一个String存入字符串池中。

执行intern操作时，如果常量池已经存在该字符串，则直接返回字符串引用，否则复制该字符串对象的引用到常量池中并返回。

除此之外，位于堆区的常量池中的对象可以被垃圾回收。当常量池中的字符串不再存在指向它的引用时，JVM就会回收该字符串。

intern案例分析

```
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);
}
```

打印结果是

- jdk6 下`false false`
- jdk7 下`false true`

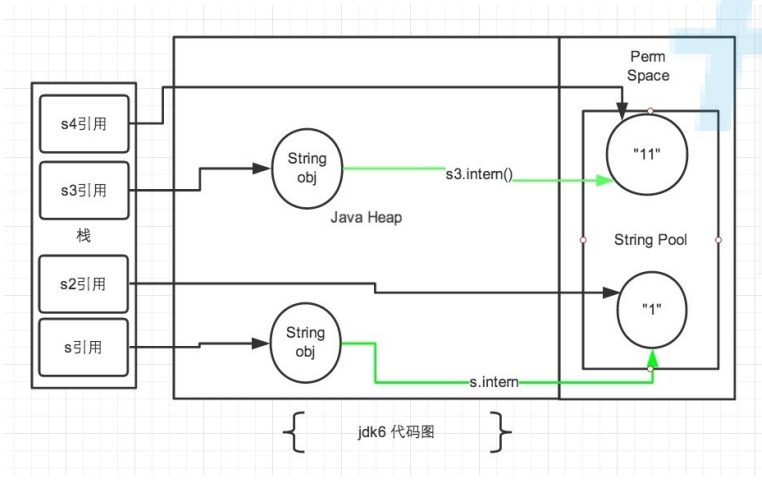
具体为什么稍后再解释，然后将`s3.intern();`语句下调一行，放到`String s4 = "11";`后面。将`s.intern();`放到`String s2 = "1";`后面。是什么结果呢?

```
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);
}
```

打印结果为:

- jdk6 下`false false`
- jdk7 下`false false`

jdk6中的解释




```
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);
}
```

注:

图中绿色线条代表 string 对象的内容指向。

黑色线条代表地址指向。

打印结果是

- jdk6 下`false false`
- jdk7 下`false true`

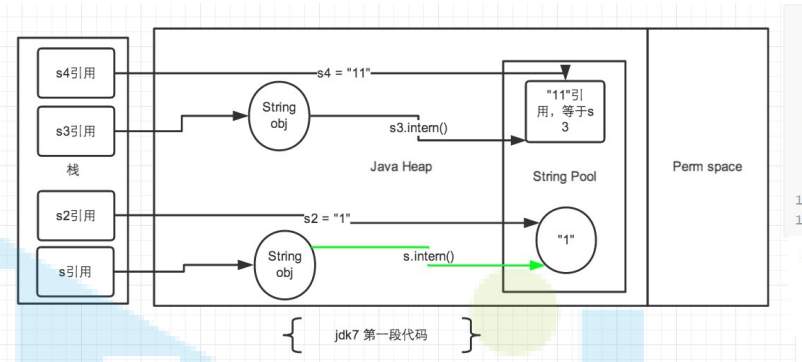
如上图所示。在 jdk6中上述的所有打印都是 false 的，因为 jdk6中的常量池是放在 Perm 区中的，Perm 区和正常的 JAVA Heap 区域是完全分开的。上面说过如果是使用引号声明的字符串都是会直接在字符串常量池中生成，而 new 出来的 String 对象是放在 JAVA Heap 区域。所以拿一个 JAVA Heap 区域的对象地址和字符串常量池的对象地址进行比较肯定是不相同的，即使调用`String.intern`方法也是没有任何

关系的。

jdk7中的解释

在 Jdk6 以及以前的版本中，字符串的常量池是放在堆的Perm区的，Perm区是一个类静态的区域，主要存储一些加载类的信息，常量池，方法片段等内容，默认大小只有4m，一旦常量池中大量使用 intern 是会直接产生`java.lang.OutOfMemoryError:PermGen space`错误的。

在 jdk7 的版本中，字符串常量池已经从Perm区移到正常的Java Heap区域了。为什么要移动，Perm 区域太小是一个主要原因，而且jdk8已经直接取消了Perm区域，而新建立了一个元区域。应该是jdk开发者认为Perm区域已经不适合现在 JAVA 的发展了。正式因为字符串常量池移动到JAVA Heap区域后，再来解释为什么会有上述的打印结果。

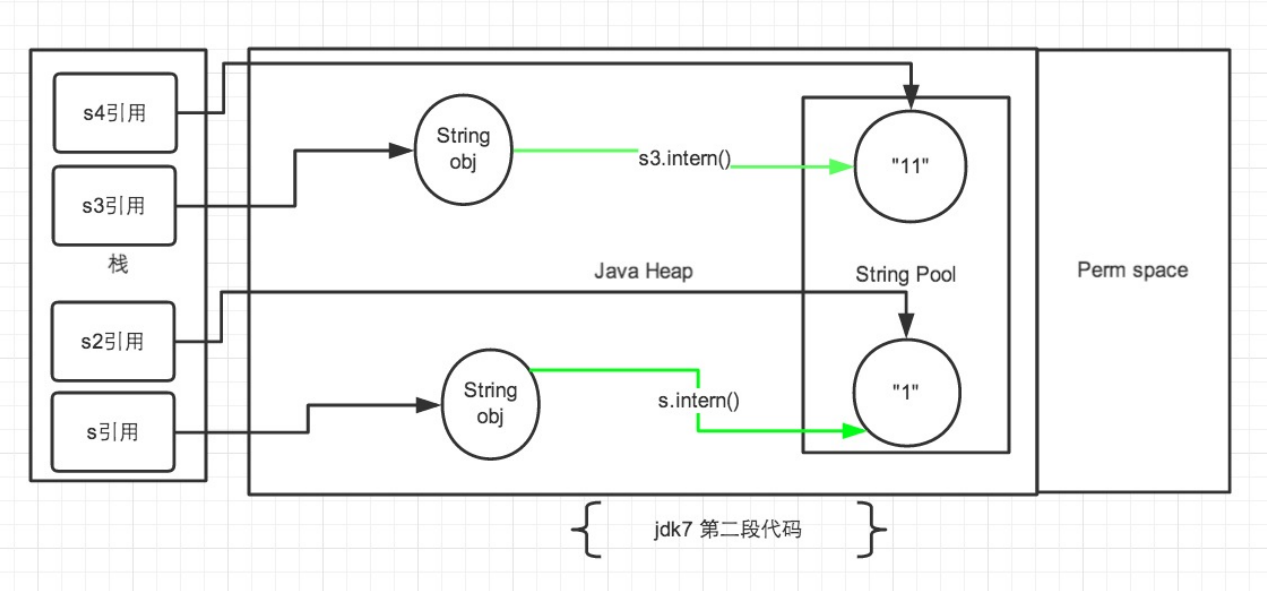



```java
public static void main(String[] args) {
    String s = new String("1");
    String s2 = "1";
    s.intern();
    System.out.println(s == s2);

    String s3 = new String("1") + new String("1");
    String s4 = "11";
    s3.intern();
    System.out.println(s3 == s4);
}
```

打印结果为：

- jdk6 下`false false`
- jdk7 下`false false`

- 在第一段代码中，先看 s3和s4字符串。`String s3 = new String("1") + new String("1");`，这句代码中现在生成了2最终个对象，是字符串常量池中的“1” 和 JAVA Heap中的s3引用指向的对象。中间还有2个匿名的`new String("1")`我们不去讨论它们。此时s3引用对象内容是”11"，但此时常量池中是没有 “11”对象的。
- 接下来`s3.intern();`这一句代码，是将 s3中的"11"字符串放入String 常量池中，因为此时常量池中不存在"11"字符串，因此常规做法是跟 jdk6 图中表示的那样，在常量池中生成一个"11"的对象，关键点是 jdk7 中常量池不在Perm区域了，这块做了调整。常量池中不需要再存储一份对象了，可以直接存储堆中的引用。这份引用指向s3引用的对象。 也就是说引用地址是相同的。
- 最后`String s4 = "11";`这句代码中”11"是显示声明的，因此会直接去常量池中创建，创建的时候发现已经有这个对象了，此时也就是指向s3引用对象的一个引用。所以s4引用就指向和s3一样了。因此最后的比较`s3 == s4`是 true。
- 再看s和 s2 对象。`String s = new String("1");`第一句代码，生成了2个对象。常量池中的“1” 和 JAVA Heap 中的字符串对象。`s.intern();`这一句是 s 对象去常量池中寻找后发现 “1” 已经在常量池里了。
- 接下来`String s2 = "1";`这句代码是生成一个 s2的引用指向常量池中的“1”对象。 结果就是 s 和 s2 的引用地址明显不同。图中画的很清晰。



- 来看第二段代码，从上边第二幅图中观察。第一段代码和第二段代码的改变就是 `s3.intern();` 的顺序是放在 `String s4 = "11";` 后了。这样，首先执行 `String s4 = "11";` 声明 s4 的时候常量池中是不存在“11”对象的，执行完毕后，“11“对象是 s4 声明产生的新对象。然后再执行 `s3.intern();` 时，常量池中“11”对象已经存在了，因此 s3 和 s4 的引用是不同的。
- 第二段代码中的 s 和 s2 代码中， `s.intern();` ，这一句往后放也不会有什么影响了，因为对象池中在执行第一句代码 `String s = new String("1");` 的时候已经生成“1”对象了。下边的s2声明都是直接从常量池中取地址引用的。 s 和 s2 的引用地址是不会相等的。

小结

从上述的例子代码可以看出 jdk7 版本对 intern 操作和常量池都做了一定的修改。主要包括2点：

- 将String常量池从Perm区移动到了Java Heap区
- `String#intern` 方法时，如果存在堆中的对象，会直接保存对象的引用，而不会重新创建对象。

intern方法的好处

如果在字符串拼接中，有一个参数是非字面量，而是一个变量的话，整个拼接操作会被编译成StringBuilder.append，这种情况编译器是无法知道其确定值的。只有在运行期才能确定。

```
String s3 = new String("1") + new String("1");
```

那么，有了这个特性了，intern就有用武之地了。那就是很多时候，我们在程序中得到的字符串是只有在运行期才能确定的，在编译期是无法确定的，那么也就没办法在编译期被加入到常量池中。

这时候，对于那种可能经常使用的字符串，使用intern进行定义，每次JVM运行到这段代码的时候，就会直接把常量池中该字面值的引用返回，这样就可以减少大量字符串对象的创建了。

```
static final int MAX = 1000 * 10000;
static final String[] arr = new String[MAX];

public static void main(String[] args) throws Exception {
    Integer[] DB_DATA = new Integer[10];
    Random random = new Random(10 * 10000);
```

```
    for (int i = 0; i < DB_DATA.length; i++) {
        DB_DATA[i] = random.nextInt();
    }
    long t = System.currentTimeMillis();
    for (int i = 0; i < MAX; i++) {
         arr[i] = new String(String.valueOf(DB_DATA[i %
DB_DATA.length])).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。

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。

2.1、三种参数类型

jvm的参数类型分为三类，分别是：

1、标准参数

- -help
- -version

2、-X参数（非标准参数）

- -Xint
- -Xcomp

3、-XX参数（使用率较高）

- -XX:newSize
- -XX:+UseSerialGC

2.2、标准参数

2.2.1、参数介绍

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
```

2.2.2、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("kkb");
        } 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
kkb
[root@node01 test]# java -Dstr=123 TestJVM
123
```

2.2.3、-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。

测试：

```
[root@node01 test]# java -client -showversion 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)
kkb

[root@node01 test]# java -server -showversion 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)
kkb

#由于机器是64位系统，所以不支持client模式
```

2.3、-X参数

2.3.1、参数介绍

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。
```

2.3.2、-Xint、-Xcomp、-Xmixed

- 在解释模式(interpreted mode)下，-Xint标记会强制JVM执行所有的字节码，当然这会降低运行速度，通常低10倍或更多。
- -Xcomp参数与它（-Xint）正好相反，JVM在第一次使用时会把所有的字节码编译成本地代码，从而带来最大程度的优化。

  然而，很多应用在使用-Xcomp也会有一些性能损失，当然这比使用-Xint损失的少，原因是-xcomp没有让JVM启用JIT编译器的全部功能。JIT编译器可以对是否需要编译做判断，如果所有代码都进行编译的话，对于一些只执行一次的代码就没有意义了。
- -Xmixed是混合模式，将解释模式与编译模式进行混合使用，由jvm自己决定，这是jvm默认的模式，也是推荐使用的模式。

示例：强制设置运行模式

```
#强制设置为解释模式
[root@node01 test]# java -showversion -Xint 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, interpreted mode)

kkb

#强制设置为编译模式
[root@node01 test]# java -showversion -Xcomp 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, compiled mode)

kkb
#注意：编译模式下，第一次执行会比解释模式下执行慢一些，注意观察。

#默认的混合模式
[root@node01 test]# java -showversion 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)
kkb
```

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:[+-]

  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:

  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)

kkb
```

2.5、-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。

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

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

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
kkb
```

2.6、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；

2.6.1、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。
```

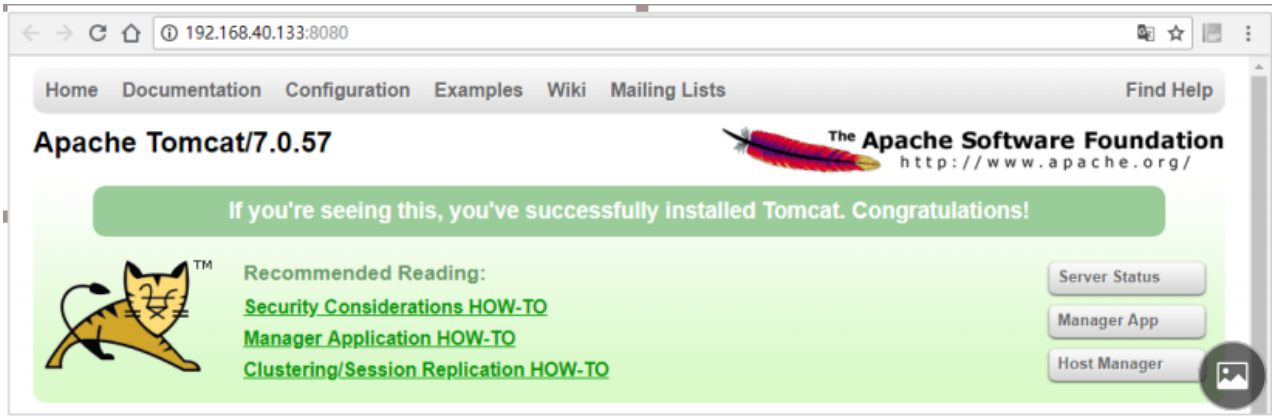
2.6.2、查看正在运行的jvm参数

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-7.0.57.tar.gz
cd apache-tomcat-7.0.57
cd bin/
./startup.sh

#http://192.168.40.133:8080/ 进行访问
```

访问成功：

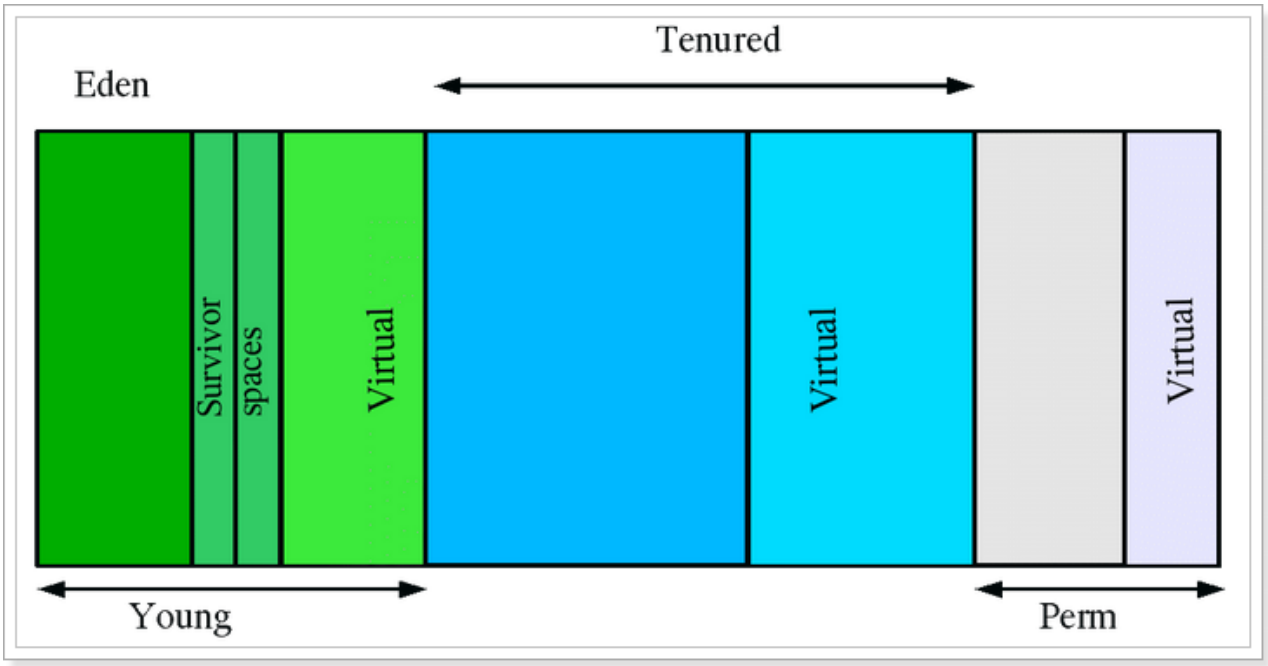


```
#查看所有的参数，用法：jinfo -flags <进程id>
#通过jps 或者 jps -l 查看java进程
[root@node01 bin]# jps
6346 Jps
6219 Bootstrap
[root@node01 bin]# jps -l
6358 sun.tools.jps.Jps
6219 org.apache.catalina.startup.Bootstrap
[root@node01 bin]#

[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
```

3、jvm的内存模型

jvm的内存模型在1.7和1.8有较大的区别，虽然本套课程是以1.8为例进行讲解，但是我们也是需要对1.7的内存模型有所了解，所以接下里，我们将先学习1.7再学习1.8的内存模型。

3.1、jdk1.7的堆内存模型



- Young 年轻区（代）

  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区间变满的时候，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。

- Tenured 年老区

  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，一般如果系统中用了application级别的缓存，缓存中的对象往往会被转移到这一区间。

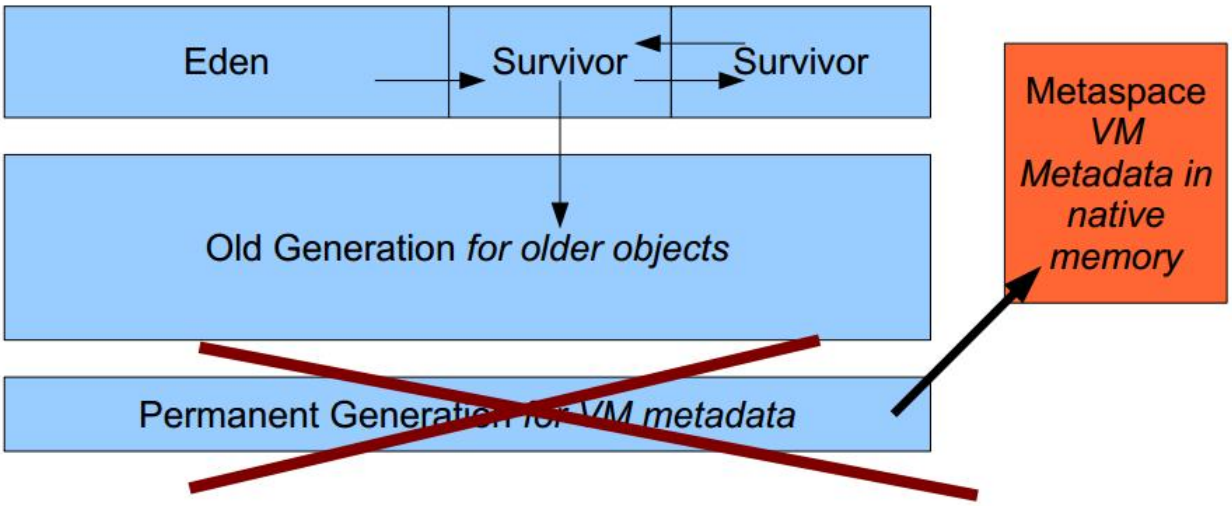
- Perm 永久区

  Perm代主要保存class,method,filed对象，这部份的空间一般不会溢出，除非一次性加载了很多的类，不过在涉及到热部署的应用服务器的时候，有时候会遇到java.lang.OutOfMemoryError : PermGen space 的错误，造成这个错误的很大原因就有可能是每次都重新部署，但是重新部署后，类的class没有被卸载掉，这样就造成了大量的class对象保存在了permq中，这种情况下，一般重新启动应用服务器可以解决问题。

- Virtual区：

  最大内存和初始内存的差值，就是Virtual区。

3.2、jdk1.8的堆内存模型



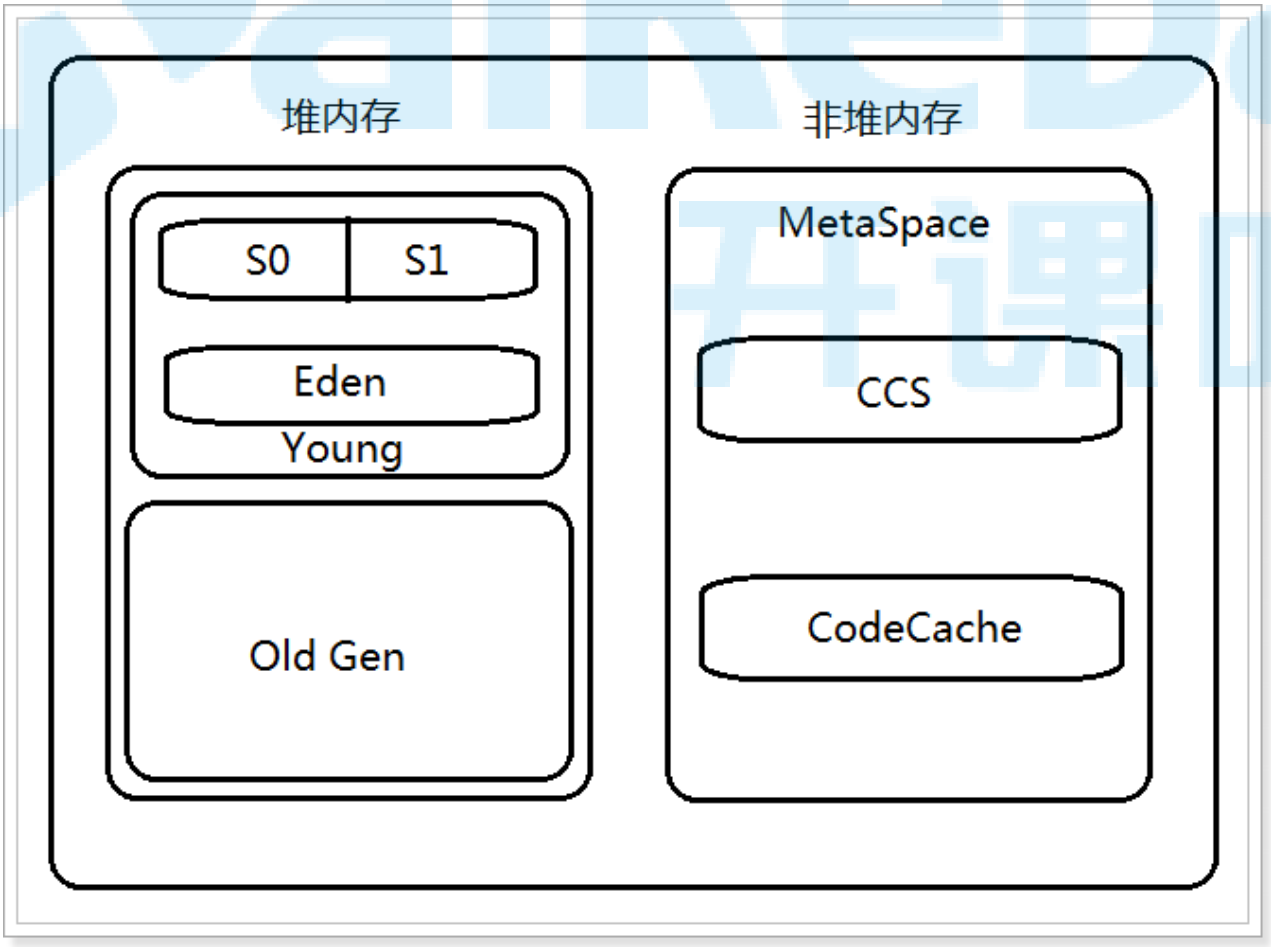
由上图可以看出，jdk1.8的内存模型是由2部分组成，年轻代 + 年老代。

年轻代：Eden + 2*Survivor

年老代：OldGen

在jdk1.8中变化最大的Perm区，用Metaspace（元数据空间）进行了替换。

需要特别说明的是：Metaspace所占用的内存空间不是在虚拟机内部，而是在本地内存空间中，这也是与1.7的永久代最大的区别所在。



3.3、为什么要废弃1.7中的永久区?

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。

现实使用中，由于永久代内存经常不够用或发生内存泄露，爆出异常java.lang.OutOfMemoryError:PermGen。

基于此，将永久区废弃，而改用元空间，改为了使用本地内存空间。