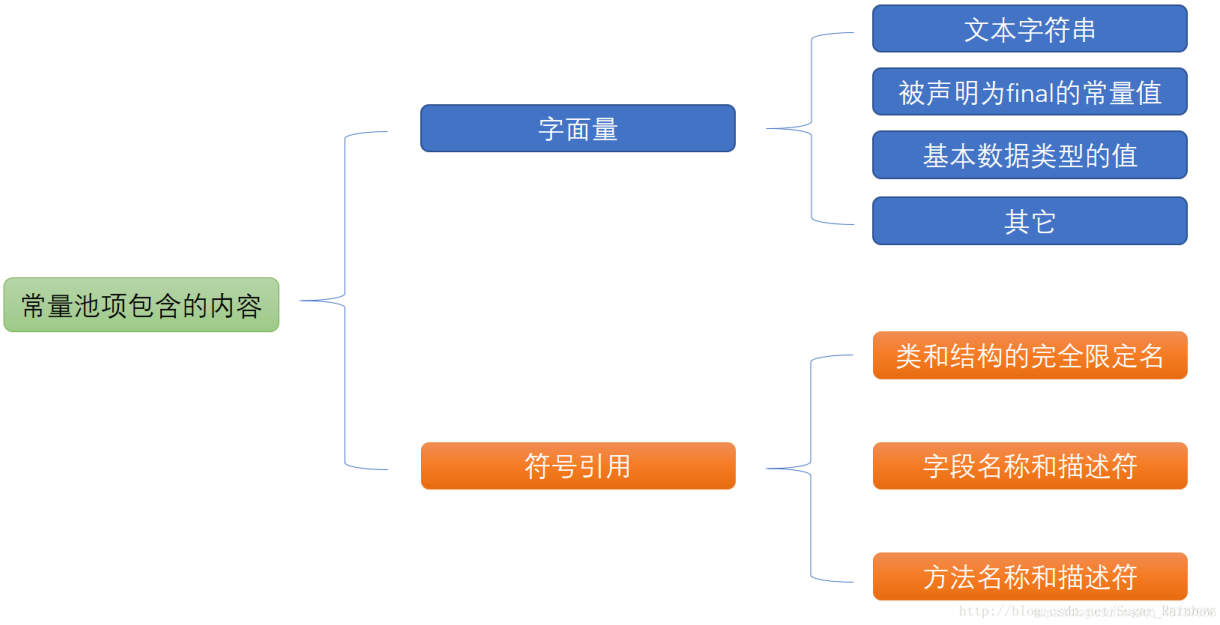
详解JVM常量池、Class常量池、运行时常量池、字符串常量池

1.常量池

常量池,也叫 Class 常量池(常量池==Class常量池)。Java文件被编译成 Class文件，Class文件中除了包含类的版本、字段、方法、接口等描述信息外，还有一项就是常量池，常量池是当Class文件被Java虚拟机加载进来后存放在方法区 各种字面量 (Literal)和 符号引用。

在Class文件结构中，最头的4个字节用于 存储魔数 (Magic Number)，用于确定一个文件是否能被JVM接受，再接着4个字节用于 存储版本号，前2个字节存储次版本号，后2个存储主版本号，再接着是用于存放常量的常量池常量池主要用于存放两大类常量：字面量和符号引用量，字面量相当于Java语言层面常量的概念，如文本字符串，声明为final的常量值等，符号引用则属于编译原理方面的概念。如下

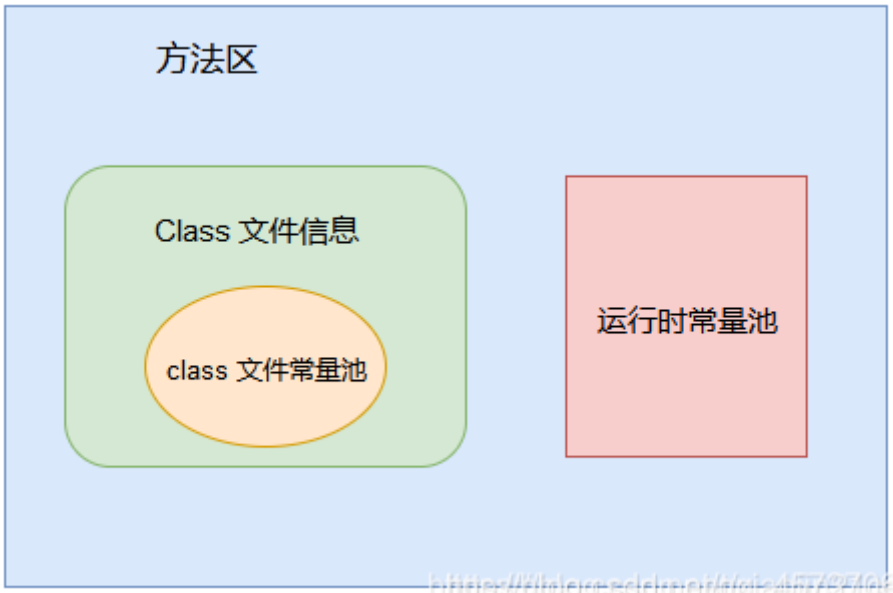


2.运行时常量池

2.1运行时常量池的简介

运行时常量池是方法区的一部分。运行时常量池是当Class文件被加载到内存后，Java虚拟机会 将Class文件常量池里的内容转移到运行时常量池里(运行时常量池也是每个类都有一个)。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

2.2方法区的Class文件信息，Class常量池和运行时常量池的三者关系



字符串常量池

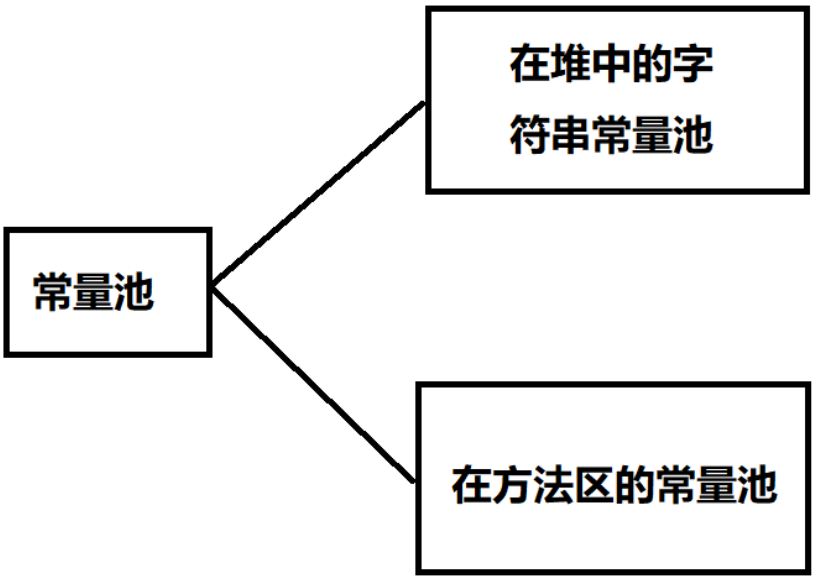
3.1字符串常量池的简介

字符串常量池

3.1字符串常量池的简介

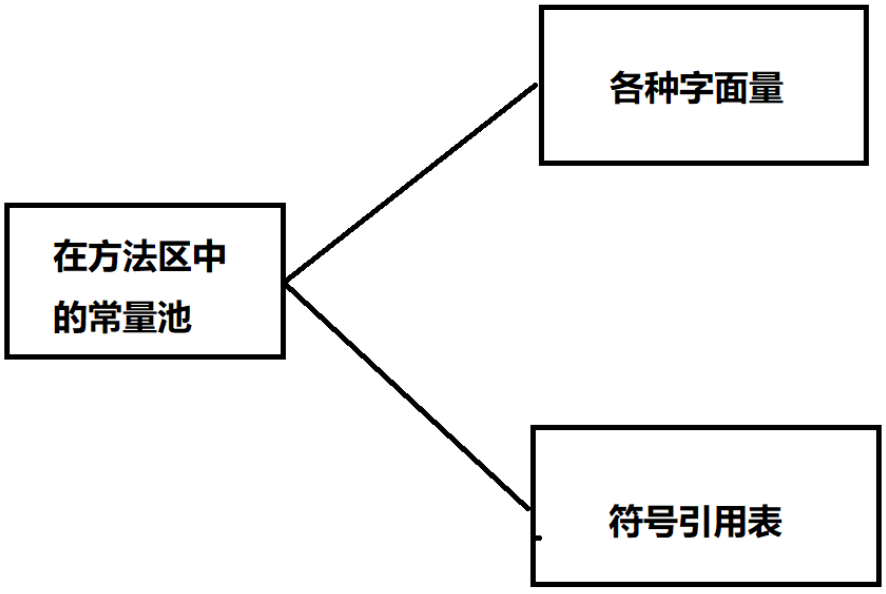
字符串常量池又称为：字符串池，全局字符串池,英文也叫String Pool。 在工作中，String类是我们使用频率非常高的一种对象类型。JVM为了提升性能和减少内存开销，避免字符串的重复创建，其维护了一块特殊的内存空间，这就是我们今天要讨论的核心：字符串常量池。字符串常量池由String类私有的维护。

在JDK7之前字符串常量池是在永久代里边的，但是在JDK7之后，把字符串常量池分进了堆里边。看下面两张图：



https://blog.csdn.net/qq_45737068

在堆中的字符串常量池： 堆里边的字符串常量池存放的是字符串的引用或者字符串(两者都有)，下面例子会有具体的讲解



https://blog.csdn.net/qq_45737068

我们知道，在Java中有两种创建字符串对象的方式：

1. 采用字面值的方式赋值
2. 采用new关键字新建一个字符串对象。这两种方式在性能和内存占用方面存在着差别。

3.2采用字面值的方式创建字符串对象

```
package Oneday;
public class a {
    public static void main(String[] args) {
        String str1="aaa";
```

```
        String str2="aaa";
        System.out.println(str1==str2);
    }
}
```

运行结果:

```
true
```

采用字面值的方式创建一个字符串时，JVM首先会去字符串池中查找是否存在"aaa"这个对象，如果不存在，则在字符串池中创建"aaa"这个对象，然后将池中"aaa"这个对象的引用地址返回给字符串常量str，这样str会指向池中"aaa"这个字符串对象；如果存在，则不创建任何对象，直接将池中"aaa"这个对象的地址返回，赋给字符串常量。

对于上述的例子：这是因为，创建字符串对象str2时，字符串池中已经存在"aaa"这个对象，直接把对象"aaa"的引用地址返回给str2，这样str2指向了池中"aaa"这个对象，也就是说str1和str2指向了同一个对象，因此语句System.out.println(str1== str2)输出：true

3.3采用new关键字新建一个字符串对象

```
package Oneday;
public class a {
    public static void main(String[] args) {
        String str1=new String("aaa");
        String str2=new String("aaa");
        System.out.println(str1==str2);
    }
}
```

运行结果:

```
false
```

采用new关键字新建一个字符串对象时，JVM首先在字符串常量池中查找有没有"aaa"这个字符串对象，如果有，则不在池中再去创建"aaa"这个对象了，直接在堆中创建一个"aaa"字符串对象，然后将堆中的这个"aaa"对象的地址返回赋给引用str1，这样，str1就指向了堆中创建的这个"aaa"字符串对象；如果没有，则首先在字符串常量池池中创建一个"aaa"字符串对象，然后再在堆中创建一个"aaa"字符串对象，然后将堆中这个"aaa"字符串对象的地址返回赋给str1引用，这样，str1指向了堆中创建的这个"aaa"字符串对象。

对于上述的例子：

因为，采用new关键字创建对象时，每次new出来的都是一个新的对象，也即是说引用str1和str2指向的是两个不同的对象，因此语句

System.out.println(str1 == str2)输出：false

字符串池的实现有一个前提条件：String对象是不可变的。因为这样可以保证多个引用可以同时指向字符串池中的同一个对象。如果字符串是可变的，那么一个引用操作改变了对象的

值，对其他引用会有影响，这样显然是不合理的。

3.4字符串池的优缺点

字符串池的优点就是避免了相同内容的字符串的创建，节省了内存，省去了创建相同字符串的时间，同时提升了性能；另一方面，字符串池的缺点就是牺牲了JVM在常量池中遍历对象所需要的时间，不过其时间成本相比而言比较低。

4 字符串常量池和运行时常量池之间的藕断丝连

4.1常量池和字符串常量池的版本变化

- 在JDK1.7之前运行时常量池逻辑包含字符串常量池存放在方法区, 此时hotspot虚拟机对方法区的实现为永久代
- 在JDK1.7 字符串常量池被从方法区拿到了堆中, 这里没有提到运行时常量池,也就是说 字符串常量池被单独拿到堆,运行时常量池剩下的东西还在方法区, 也就是hotspot中的永久代
- 在JDK1.8 hotspot移除了永久代用元空间(Metaspace)取而代之, 这时候字符串常量池还在堆, 运行时常量池还在方法区, 只不过方法区的实现从永久代变成了元空间(Metaspace)

4.2 String.intern在JDK6和JDK7之后的区别(重点)

JDK6和JDK7中该方法的功能是一致的，不同的是常量池位置的改变（JDK7将常量池放在了堆空间中），下面会具体说明。intern的方法返回字符串对象的规范表示形式。其中它做的事情是：首先去判断该字符串是否在常量池中存在，如果存在返回常量池中的字符串，如果在字符串常量池中不存在，先在字符串常量池中添加该字符串，然后返回引用地址

例子1：

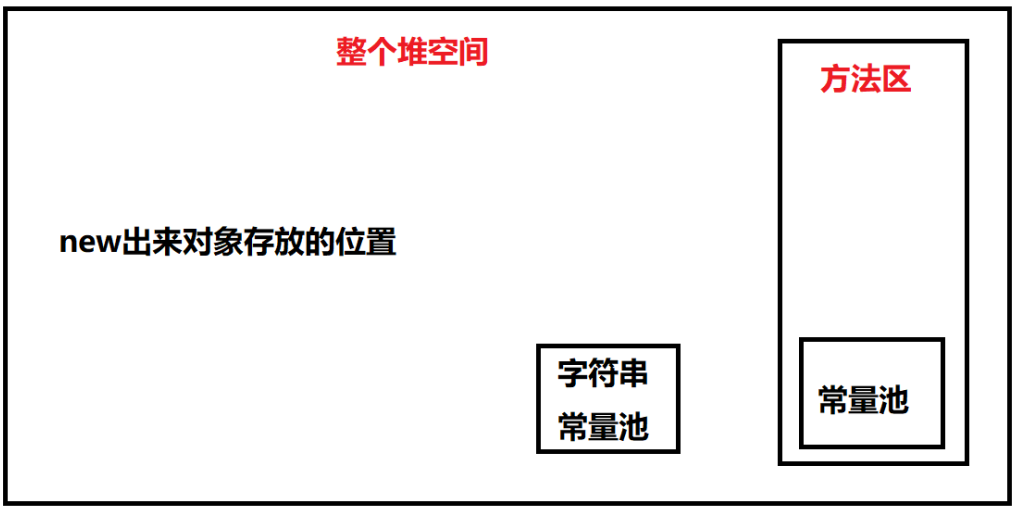
```
String s1 = new String("1");
s1.intern();
String s2 = "1";
System.out.println(s1 == s2);
```

运行结果:

JDK6运行结果：false

JDK7运行结果：false

我们首先看一张图：



https://blog.csdn.net/qq_45737068

上边例子中s1是new出来对象存放的位置的引用，s2是存放在字符串常量池的字符串的引用，所以两者不同

例子2：

```
String s1 = new String("1");
System.out.println(s1.intern() == s1);
```

运行结果：
JDK6运行结果：false
JDK7运行结果：false

上边例子中s1是new出来对象存放的位置的引用，s1.intern()返回的是字符串常量池里字符串的引用

例子3：

```
String s1 = new String("1") + new String("1");
s1.intern();
String s2 = "11";
System.out.println(s1 == s2);
```

运行结果：
JDK6运行结果：false
JDK7运行结果：true

JDK6中，s1.intern()运行时，首先去常量池查找，发现没有该常量，则在常量池中开辟空间存储"11"，返回常量池中的值（注意这里也没有使用该返回值），第三行中，s2直接指向常量池里边的字符串，所以s1和s2不相等。有可能会有小伙伴问为啥s1.intern()发现没有该常量呢，那是因为：

String s1 = new String(“1”) + new String(“1”);这行代码实际操作是，创建了一个StringBuilder对象，然后一路append，最后toString，而toString其实是又重新new了一个String对象，然后把对象给s1，此时并没有在字符串常量池中添加常量

JDK7中，由于字符串常量池在堆空间中，所以在s1.intern()运行时，发现字符串 常量池没有常量，则添加堆中“11”对象的引用到字符串常量池，这个引用返回堆空间“11”地址（注意这里也没有使用该返回值），这时s2通过查找字符串常量池中的常量，查到的是s1.intern()存在字符串常量池里的“11”对象的引用，既然都是指向堆上的“11”对象，所以s1和s2相等。

例子4：

```
String s1 = new String("1") + new String("1");
System.out.println(s1.intern() == s1);
```

JDK6中，常量池在永久代中，s1.intern()去常量池中查找"11"，发现没有该常量，则在常量池中开辟空间存储"11"，返回常量池中的值，s1指向堆空间地址，所以二者不相等。

JDK7中，常量池在堆空间，s1.intern()去常量池中查找"11"，发现没有该常量，则在字符串常量池中开辟空间，指向堆空间地址，则返回字符串常量池指向的堆空间地址，s1也是堆空间地址，所以二者相等。

另外美团的团队写了一篇关于intern()的博客，我觉得很好可以参考一下
深入解析String#intern

4.3字符串常量池里存放的是引用还是字面量
我在例子3中讲了在JDK7中字符串常量池在堆上，仔细看看例3啥时候会放引用

那么啥时候会放字面量在字符串常量池呢，那就是在我们new一个String对象的时候如果字符串常量池里边有字面量那么就不会放，如果字符串常量池没有就会放字面量。看一个例子：

```
package Oneday;
import java.util.HashSet;
import java.util.Set;
public class a {
    public static void main(String[] args) {
        String str1= new String("123");
        String str2=new String("123");
        System.out.println(str1==str2);
        System.out.println(str1.intern()==str2.intern());
```

```
    }
}
```

运行结果：

首先 String str1= new String("123");会在堆中创建一个对象，返回这个对象的引用给str1，同时它还会在字符串常量池中检查有没有有没有123这个对象，如果没有就再创建一个对象(也就是123这个字面量)在字符串常量池中

注意这里是创建了两个对象

但是当我们字符串常量池里边有123这个对象，那么就不用继续创建了

上面例子的false那是因为堆中的123对象不是同一个对象，但是第二个str1.intern和s2.intern指的都是字符串常量池里的123对象所以是true