最近一直被方法区里面存着什么东西困扰着？

1.方法区里存class文件信息和class文件常量池是个什么关系。

2.class文件常量池和运行时常量池是什么关系。

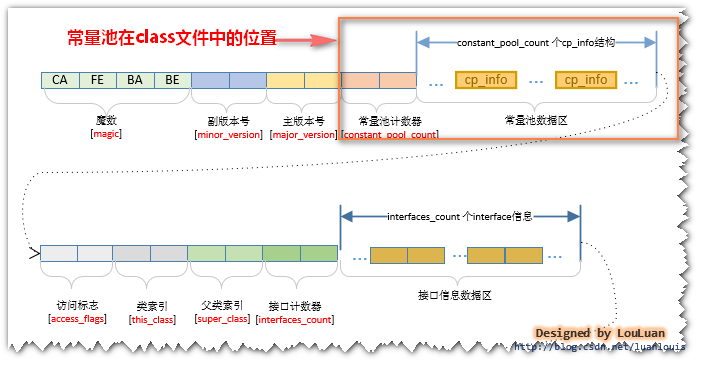
方法区存着类的信息，常量和静态变量，即类被编译后的数据。这个说法其实是没问题的，只是太笼统了。更加详细一点的说法是方法区里存放着类的版本，字段，方法，接口和常量池。常量池里存储着字面量和符号引用。

 符号引用包括：1.类的全限定名，2.字段名和属性，3.方法名和属性。

下面一张图是我画的方法区，class文件信息，class文件常量池和运行时常量池的关系



下面一张图用来表示方法区class文件信息包括哪些内容:

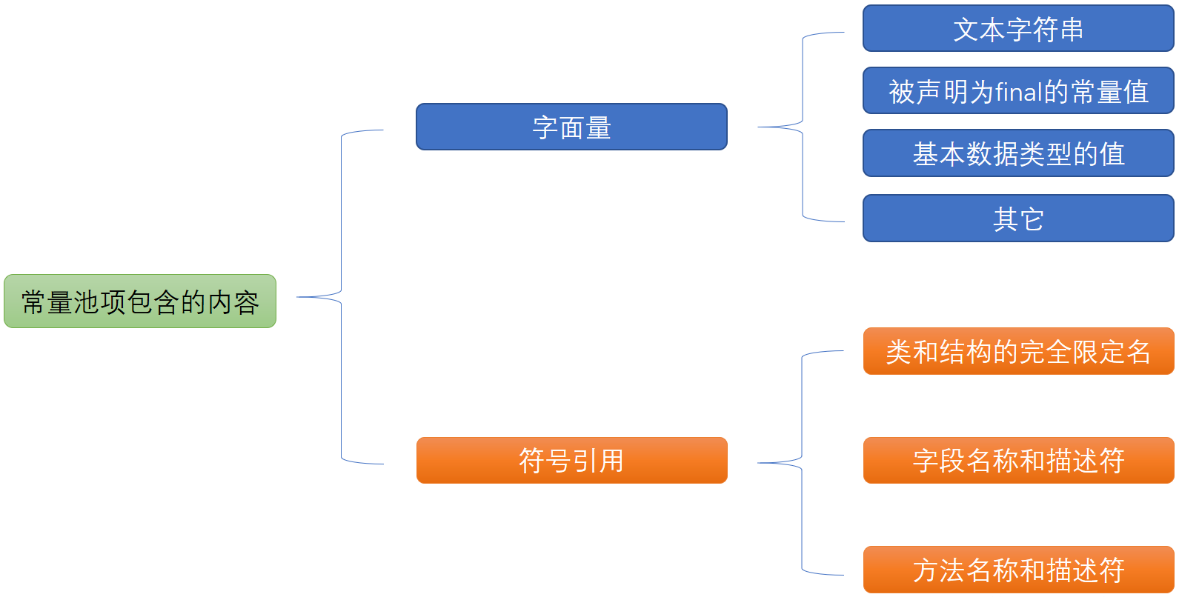


可以看到在方法区里的class文件信息包括：魔数，版本号，常量池，类，父类和接口数组，字段，方法等信息，其实类里面又包括字段和方法的信息。

下面的图表是class文件中存储的数据类型

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 数量 |
| u4 | magic | 1 |
| u2 | minor\_version | 1 |
| u2 | major\_version | 1 |
| u2 | constant\_pool\_count | 1 |
| cp\_info | constant\_pool | constant\_pool\_count - 1 |
| u2 | access\_flags | 1 |
| u2 | this\_class | 1 |
| u2 | super\_class | 1 |
| u2 | interfaces\_count | 1 |
| u2 | interfaces | interfaces\_count |
| u2 | fields\_count | 1 |
| field\_info | fields | fields\_count |
| u2 | methods\_count | 1 |
| method\_info | methods | methods\_count |
| u2 | attribute\_count | 1 |
| attribute\_info | attributes | attributes\_count |

       下面用一张图来表示常量池里存储的内容：



用一个class文件实际反编译一下，下面是原java代码：

public class TestInt {

private String str = "hello";

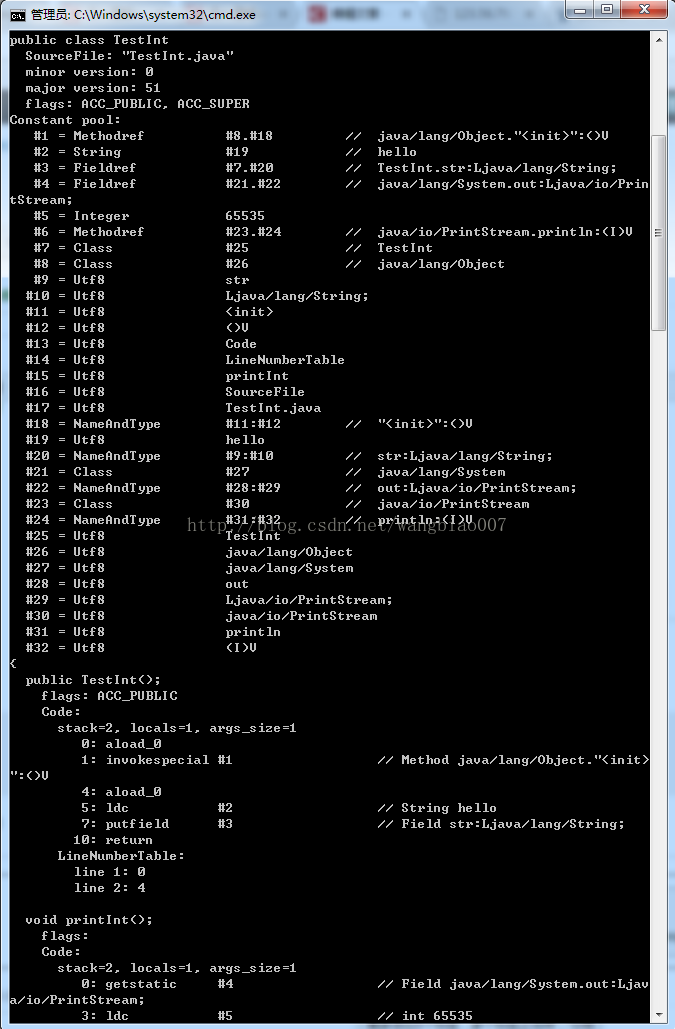
void printInt(){

System.out.println(65535);

}

}

经过反编译后获得class文件是下面这样的



可以看出被反编译的class文件中的内容和上面所说的是能对应上的。这就解答了class文件和静态常量池（class文件常量池）的关系

静态常量池和动态常量池的关系以及区别

静态常量池存储的是当class文件被java虚拟机加载进来后存放在方法区的一些字面量和符号引用，字面量包括字符串，基本类型的常量，符号引用其实引用的就是常量池里面的字符串，但符号引用不是直接存储字符串，而是存储字符串在常量池里的索引。

动态常量池是当class文件被加载完成后，java虚拟机会将静态常量池里的内容转移到动态常量池里，在静态常量池的符号引用有一部分是会被转变为直接引用的，比如说类的静态方法或私有方法，实例构造方法，父类方法，这是因为这些方法不能被重写其他版本，所以能在加载的时候就可以将符号引用转变为直接引用，而其他的一些方法是在这个方法被第一次调用的时候才会将符号引用转变为直接引用的。

总结：

方法区里存储着class文件的信息和动态常量池,class文件的信息包括类信息和静态常量池。可以将类的信息是对class文件内容的一个框架，里面具体的内容通过常量池来存储。

动态常量池里的内容除了是静态常量池里的内容外，还将静态常量池里的符号引用转变为直接引用，而且动态常量池里的内容是能动态添加的。例如调用String的intern方法就能将string的值添加到String常量池中，这里String常量池是包含在动态常量池里的，但在jdk1.8后，将String常量池放到了堆中。

**《java虚拟机规范》中写道：“确切来说，java虚拟机为每个类型都维护着一个常量池”。其中常量池以“常量池表”表示，包括的常量类型有：CONSTANT\_Class / Fieldref / Methodref / InterfaceMethodref等，表示类，字段，方法，接口的符号信息。**

**常量池的好处：**

常量池是为了避免频繁的创建和销毁对象而影响系统性能，其实现了对象的共享。例如字符串常量池，在编译阶段就把所有的字符串文字放到一个常量池中。

节省内存空间：常量池中所有相同的字符串常量被合并，只占用一个空间。

节省运行时间：比较字符串时，==比equals()快。对于两个引用变量，只用==判断引用是否相等，也就可以判断实际值是否相等。

## 1.字符串常量池(String Constant Pool):

### 1.1:字符串常量池在Java内存区域的位置

在JDK6.0及之前版本，字符串常量池是放在Perm Gen区(也就是方法区)中；

在JDK7.0版本，字符串常量池被移到了堆中了。至于为什么移到堆内，大概是由于方法区的内存空间太小了。

### 1.2:字符串常量池是什么？

**在HotSpot VM里实现的string pool功能的是一个StringTable类，它是一个Hash表，默认值大小长度是1009**；**这个StringTable在每个HotSpot VM的实例只有一份，被所有的类共享**。字符串常量由一个一个字符组成，放在了StringTable上。

在JDK6.0中，StringTable的长度是固定的，长度就是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，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JDK7.0中，StringTable的长度可以通过参数指定：

-XX:StringTableSize=66666

### 1.3:字符串常量池里放的是什么？

在JDK6.0及之前版本中，String Pool里放的都是字符串常量；

在JDK7.0中，由于String.intern()发生了改变，因此String Pool中也可以存放（放于堆内的）字符串对象的引用。

关于String在内存中的存储和String.intern()方法的说明，可以参考我的另外一篇博客：

需要说明的是：字符串常量池中的字符串只存在一份！

如：

1 String s1 = "hello,world!";

2 String s2 = "hello,world!";

即执行完第一行代码后，常量池中已存在 “hello,world!”，那么 s2不会在常量池中申请新的空间，而是直接把已存在的字符串内存地址返回给s2。(这里具体的字符串如何分配就不细说了，可以看我的另一篇博客)

## 2.class常量池(Class Constant Pool)：

### 2.1:class常量池简介：

我们写的每一个Java类被编译后，就会形成一份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，还有一项信息就是常量池(constant pool table)，用于存放编译器生成的各种字面量(Literal)和符号引用(Symbolic References)；

**每个class文件都有一个class常量池。**

### 2.2:什么是字面量和符号引用：

**字面量**包括：1.文本字符串 2.八种基本类型的值 3.被声明为final的常量等;

**符号引用**包括：1.类和方法的全限定名 2.字段的名称和描述符 3.方法的名称和描述符。

**PS:**

**限定类名**，就是类名全称，带包路径的用点隔开，如(java.lang.String)

**非限定(non-qualified)类名也叫短名**，就是我们平时说的类名，不带包的，如(String)

非限定类名是相对于限定类名来说的，在Java中有很多类，不同的类之间会存在相同的函数或者方法，所以有时候就需要限定类名来调包。 而如果不存在相同的函数或者方法 ，就可以使用非限定(non-qualified)类名。

## 3.运行时常量池(Runtime Constant Pool)：

运行时常量池存在于内存中，也就是class常量池被加载到内存之后的版本，不同之处是：它的字面量可以动态的添加(String#intern()),符号引用可以被解析为直接引用

JVM在执行某个类的时候，必须经过加载、连接、初始化，而连接又包括验证、准备、解析三个阶段。而当类加载到内存中后，jvm就会将class常量池中的内容存放到运行时常量池中，由此可知，运行时常量池也是每个类都有一个。在解析阶段，会把符号引用替换为直接引用，解析的过程会去查询字符串常量池，也就是我们上面所说的StringTable，以保证运行时常量池所引用的字符串与字符串常量池中是一致的。