JVM知识梳理之二_JVM的常量池

下塘烧饼 发布于 2021-01-04

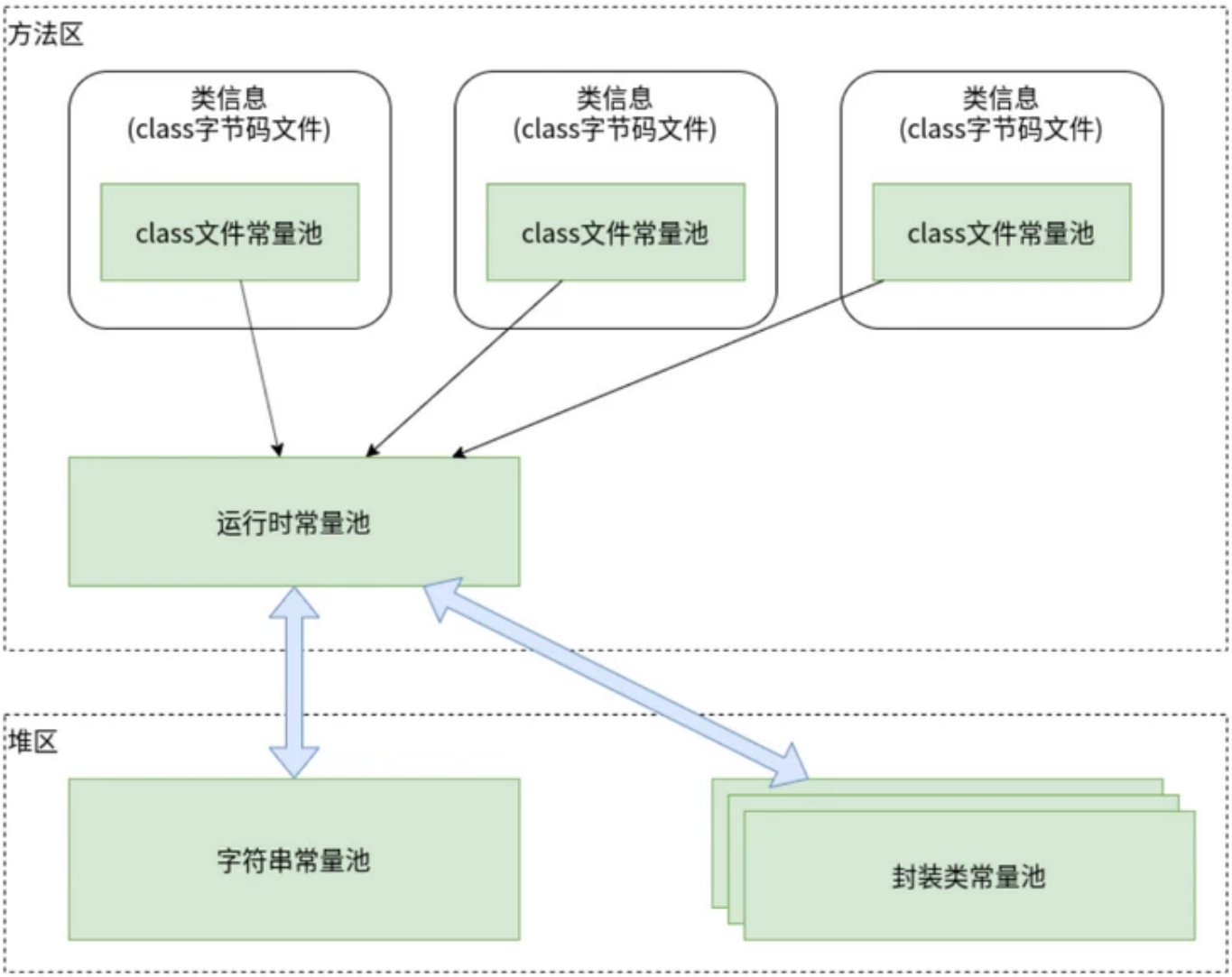
在上一篇《JVM知识梳理之一_JVM运行时内存区域与Java内存模型》中，提到了JVM的各种常量池，但没有展开讲述。本文就JVM的各种常量池进行一些简单的梳理。

一、常量池概述

JVM的常量池主要有以下几种：

- class文件常量池
- 运行时常量池
- 字符串常量池
- 基本类型包装类常量池

它们相互之间关系大致如下图所示：



1. 每个class的字节码文件中都有一个常量池，里面是编译后即知的该class会用到的字面量与符号引用，这就是`class文件常量池`。JVM加载class，会将其类信息，包括class文件常量池置于方法区中。
2. class类信息及其class文件常量池是字节码的二进制流，它代表的是一个类的静态存储结构，JVM加载类时，需要将其转换为方法区中的`java.lang.Class`类的对象实例；同时，会将class文件常量池中的内容导入运行时常量池。
3. 运行时常量池中的常量对应的内容只是字面量，比如一个"字符串"，它还不是String对象；当Java程序在运行时执行到这个"字符串"字面量时，会去字符串常量池里找该字面量的对象引用是否存在，存在则直接返回该引用，不存在则在Java堆里创建该字面量对应的String对象，并将其引用置于字符串常量池中，然后返回该引用。
4. Java的基本数据类型中，除了两个浮点数类型，其他的基本数据类型都在各自内部实现了常量池，但都在[-128~127]这个范围内。

下面分别对每个常量池具体梳理一下。

二、class文件常量池

java的源代码.java文件在编译之后会生成.class文件，class文件需要严格遵循JVM规范才能被JVM正常加载，它是一个二进制字节流文件，里面包含了class文件常量池的内容。

2.1 查看一个class文件内容

jdk提供了javap命令，用于对class文件进行反汇编，输出类相关信息。该命令用法如下：

```
用法: javap <options> <classes>
其中, 可能的选项包括:
  -help  --help  -?        输出此用法消息
  -version                 版本信息
  -v  -verbose             输出附加信息
  -l                       输出行号和本地变量表
  -public                  仅显示公共类和成员
  -protected               显示受保护的/公共类和成员
  -package                 显示程序包/受保护的/公共类
                           和成员 (默认)
  -p  -private             显示所有类和成员
  -c                       对代码进行反汇编
  -s                       输出内部类型签名
  -sysinfo                 显示正在处理的类的
                           系统信息 (路径, 大小, 日期, MD5 散列)
  -constants               显示最终常量
  -classpath <path>        指定查找用户类文件的位置
  -cp <path>               指定查找用户类文件的位置
  -bootclasspath <path>    覆盖引导类文件的位置
```

例如，我们可以编写一个简单的类，如下：

```
        return evaluate;
    }

    public void setEvaluate(String evaluate) {
        String tmp = "+";
        this.evaluate = evaluate + tmp;
    }

    public int getScores() {
        return scores;
    }

    public void setScores(int scores) {
        final int base = 10;
```

```
    public Integer getLevel() {
        return level;
    }

    public void setLevel(Integer level) {
        this.level = level;
    }
}
```

对其进行编译和反汇编:

```
javac Student.java

javap -v Student.class

# over
```

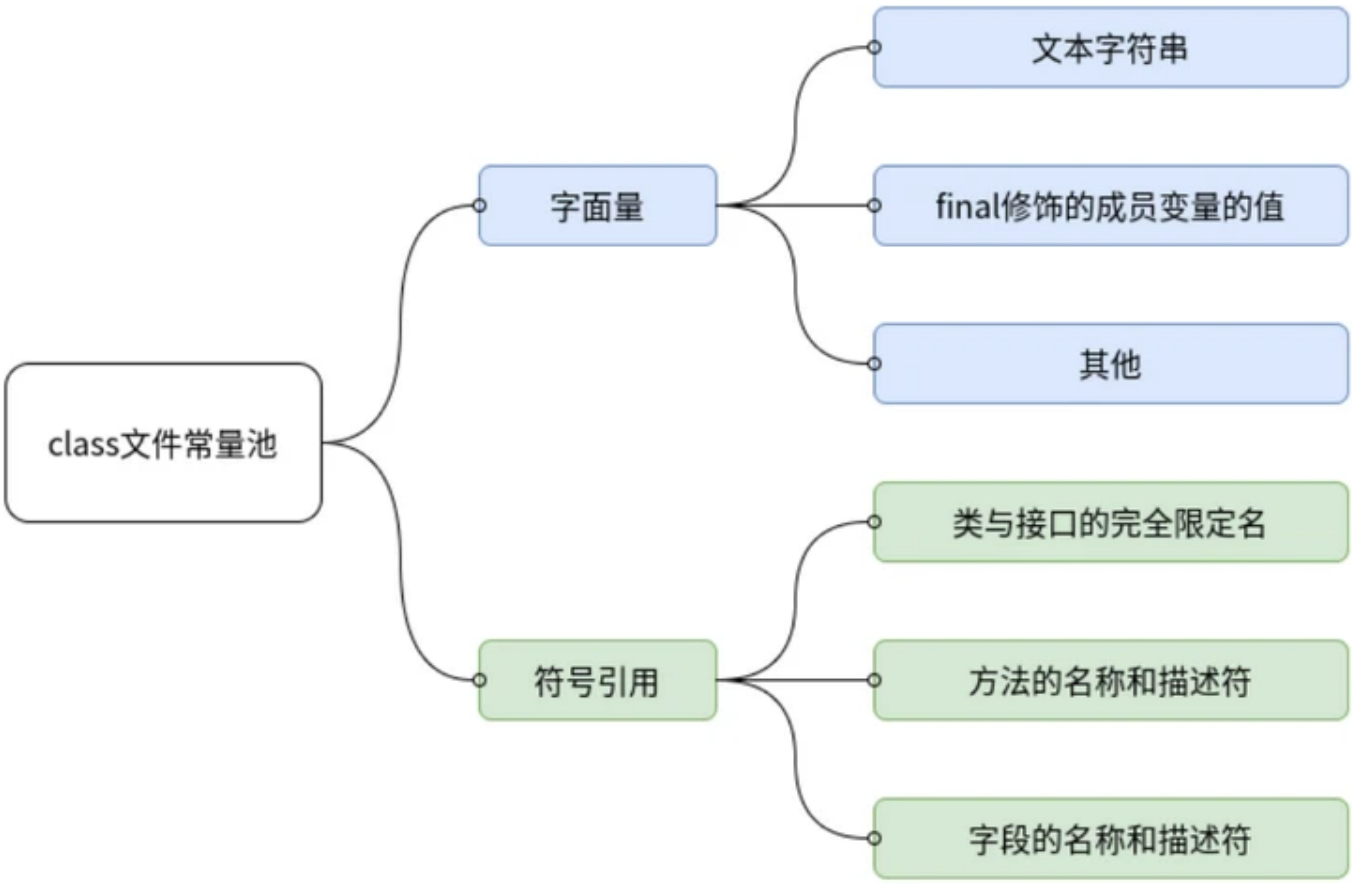
得到以下反汇编结果:

```
Classfile /home/work/sources/open_projects/lib-zc-crypto/src/test/java/Student.c
  Last modified 2021-1-4; size 1299 bytes
  MD5 checksum 06dfdad9da59e2a64d62061637380969
  Compiled from "Student.java"
public class Student
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #19.#48        // java/lang/Object."<init>":()V
   #2 = String             #49            // 张三
   #3 = Fieldref           #18.#50        // Student.name:Ljava/lang/String;
   #4 = Fieldref           #18.#51        // Student.entranceAge:I
   #5 = String             #52            // 优秀
   #6 = Fieldref           #18.#53        // Student.evaluate:Ljava/lang/String;
   #7 = Fieldref           #18.#54        // Student.scores:I
   #8 = Methodref          #55.#56        // java/lang/Integer.valueOf:(I)Ljava/
   #9 = Fieldref           #18.#57        // Student.level:Ljava/lang/Integer;
  #10 = String             #58            // +
  #11 = Class              #59            // java/lang/StringBuilder
  #12 = Methodref          #11.#48        // java/lang/StringBuilder."<init>":()
  #13 = Methodref          #11.#60        // java/lang/StringBuilder.append:(Lja
  #14 = Methodref          #11.#61        // java/lang/StringBuilder.toString:()
  #15 = Fieldref           #62.#63        // java/lang/System.out:Ljava/io/Print
```

其中的Constant pool就是class文件常量池，使用#加数字标记每个“常量”。

2.2 class文件常量池的内容

class文件常量池存放的是该class编译后即知的，在运行时将会用到的各个“常量”。注意这个常量不是编程中所说的final修饰的变量，而是字面量和符号引用，如下图所示：



2.2.1 字面量

字面量大约相当于Java代码中的双引号字符串和常量的实际的值，包括：

1.文本字符串，即代码中用双引号包裹的字符串部分的值。例如刚刚的例子中，有三个字符串："张三"，"优秀"，"+"，它们在class文件常量池中分别对应：

```
#49 = Utf8               张三
#52 = Utf8               优秀
#58 = Utf8               +
```

> 这里的#49就是"张三"的字面量，它不是一个String对象，只是一个使用utf8编码的文本字符串而已

2.用final修饰的成员变量，例如，`private static final int entranceAge = 18;`这条语句定义了一个final常量`entranceAge`，它的值是`18`，对应在class文件常量池中就有：

```
#25 = Integer             18
```

> 注意，只有final修饰的成员变量如`entranceAge`，才会在常量池中存在对应的字面量。而非final的成员变量`scores`，以及局部变量`base`(即使使用final修饰了)，它们的字面量都不会在常量池中定义。

2.2.2 符号引用

符号引用包括：

1.类和接口的全限定名，例如：

```
#11 = Class               #59            // java/lang/StringBuilder
#59 = Utf8                java/lang/StringBuilder
```

2.方法的名称和描述符，例如：

```
#38 = Utf8                getScores
#39 = Utf8                ()I

#40 = Utf8                setScores
#41 = Utf8                (I)V
```

以及这种对其他类的方法的引用：

```
#8 = Methodref          #55.#56        // java/lang/Integer.valueOf:(I)Ljava/lan

#55 = Class               #69            // java/lang/Integer
#69 = Utf8                java/lang/Integer

#56 = NameAndType         #70:#71        // valueOf:(I)Ljava/lang/Integer;
#70 = Utf8                valueOf
#71 = Utf8                (I)Ljava/lang/Integer;
```

3.字段的名称和描述符，例如：

```
#3 = Fieldref           #18.#50        // Student.name:Ljava/lang/String;

#18 = Class             #67            // Student
#67 = Utf8              Student

#50 = NameAndType       #20:#21        // name:Ljava/lang/String;
#20 = Utf8              name
#21 = Utf8              Ljava/lang/String;
```

以及这种局部变量：

```
#64 = Utf8              base:10
```

三、运行时常量池

JVM在加载某个class的时候，需要完成以下任务：

1. 通过该class的全限定名来获取它的二进制字节流，即读取其字节码文件。其内容包括在章节二、class文件常量池中介绍的class文件常量池。
2. 将读入的字节流从静态存储结构转换为方法区中的运行时的数据结构。
3. 在Java堆中生成该class对应的类对象，代表该class原信息。这个类对象的类型是java.lang.Class，它与普通对象不同的地方在于，普通对象一般都是在new之后创建的，而类对象是在类加载的时候创建的，且是单例。

而上述过程的第二步，就包含了将class文件常量池内容导入运行时常量池。class文件常量池是一个class文件对应一个常量池，而运行时常量池只有一个，多个class文件常量池中的相同字符串只会对应运行时常量池中的一个字符串。

运行时常量池除了导入class文件常量池的内容，还会保存符号引用对应的直接引用(实际内存地址)。这些直接引用是JVM在类加载之后的链接(验证、准备、解析)阶段从符号引用翻译过来的。

此外，运行时常量池具有动态性的特征，它的内容并不是全部来源与编译后的class文件，在运行时也可以通过代码生成常量并放入运行时常量池。比如String.intern()方法。

> String.intern()方法的分析见后续章节。

要注意的是，运行时常量池中保存的“常量”依然是字面量和符号引用。比如字符串，这里放的仍然是单纯的文本字符串，而不是String对象。

四、字符串常量池

如前所述，class文件常量池和运行时常量池中，都没有直接存储字面量对应的实际对象，比如String对象。那么String对象到底是什么时候在哪里创建的呢?

4.1 字面量赋值创建String对象

我们以下面这个简单的例子来说明使用字面量赋值方法来创建一个String对象的大致流程：

```
String s = "黄河之水天上来";
```

当Java虚拟机启动成功后，上面的字符串"黄河之水天上来"的字面量已经进入运行时常量池；

然后主线程开始运行，第一次执行到这条语句时，JVM会根据运行时常量池中的这个字面量去字符串常量池寻找其中是否有该字面量对应的String对象的引用。注意是引用。

如果没找到，就会去Java堆创建一个值为"黄河之水天上来"的String对象，并将该对象的引用保存到字符串常量池，然后返回该引用；如果找到了，说明之前已经有其他语句通过相同的字面量赋值创建了该String对象，直接返回引用即可。

4.2 字符串常量池

字符串常量池，是JVM用来维护字符串实例的一个引用表。在HotSpot虚拟机中，它被实现为一个全局的StringTable，底层是一个c++的hashtable。它将字符串的字面量作为key，实际堆中创建的String对象的引用作为value。

字符串常量池在逻辑上属于方法区，但JDK1.7开始，就被挪到了堆区。

String的字面量被导入JVM的运行时常量池时，并不会马上试图在字符串常量池加入对应String的引用，而是等到程序实际运行时，要用到这个字面量对应的String对象时，才会去字符串常量池试图获取或者加入String对象的引用。因此它是懒加载的。

4.3 new String()与String.intern()

通过下面的例子，可以帮助我们加深对字符串常量池的理解。

```
// 语句1
String s1 = new String("asdf");

// 语句2
System.out.println(s1 == "asdf");
```

这个例子中假设"asdf"是首次被执行，那么语句1会创建两个String对象。一个是JVM拿字面量"asdf"去字符串常量池试图获取其对应String对象的引用，因为是首次执行，所以没找到，于是在堆中创建了一个"asdf"的String对象，并将其引用保存到字符串常量池中，然后返回；返回之后，因为new的存在，JVM又在堆中创建了与"asdf"等值的另一个String对象。因此这条语句创建了两个String对象，它们值相等，都是"asdf"，但是引用(内存地址)不同，所以语句2返回false。

例2：

```
// 语句3
String s3 = new String("a") + new String("b");

// 语句4
s3.intern();

// 语句5
String s4 = "ab";

// 语句6
System.out.println(s3 == s4);
```

这个例子也假设相关字符串字面量都是首次被执行到，那么语句3会创建5个对象：两个"a"，其中一个的引用被保存在字符串常量池；两个"b"，其中一个的引用被保存在字符串常量池；一个"ab"，其引用没有被保存在字符串常量池。

> 两个String对象用"+"拼接会被优化为StringBuffer的append拼接，然后toString方法，与new一样会直接在堆中创建对象。

语句4要注意，JDK1.6和JDK1.7开始，String.intern()的执行逻辑是不一样的。

- JDK1.6会判断"ab"在字符串常量池中不存在，于是创建新的"ab"对象并将其引用保存到字符串常量池。
- JDK1.7开始，判断"ab"在字符串常量池里不存在的话，会直接把s3的引用保存到字符串常量池。

因此对于语句6，如果是JDK1.6及以前的版本，结果就是false；而如果是JDK1.7开始的版本，结

> 如果没有语句4，那么语句6结果一定是false。

4.4 字符串常量池是否会被GC

字符串常量池本身不会被GC，但其中保存的引用所指向的String对象们是可以被回收的。否则字符串常量池总是"只进不出"，那么很可能会导致内存泄露。

在HotSpot的字符串常量池实现StringTable中，提供了相应的接口用于支持GC，不同的GC策略会在适当的时候调用它们。一般实在Full GC的时候，额外调用StringTable的对应接口做可达性分析，将不可达的String对象的引用从StringTable中移除掉并销毁其指向的String对象。

五、封装类常量池

除了字符串常量池，Java的基本类型的封装类大部分也都实现了常量池。包括`Byte,Short,Integer,Long,Character,Boolean`，注意，浮点数据类型`Float,Double`是没有常量池的。

封装类的常量池是在各自内部类中实现的，比如`IntegerCache`(`Integer`的内部类)，自然也位于堆区。

要注意的是，这些常量池是有范围的：

- Byte,Short,Integer,Long : [-128~127]
- Character : [0~127]
- Boolean : [True, False]

例如下面的代码，注意其结果：

```
Integer i1 = 127;
Integer i2 = 127;
System.out.println(i1 == i2);

Integer i3 = 128;
Integer i4 = 128;
System.out.println(i3 == i4);

Integer i5 = -128;
Integer i6 = -128;
System.out.println(i5 == i6);
```

```
Integer i8 = -129;
System.out.println(i7 == i8);
```

java

阅读 2.1k · 更新于 2021-01-05

赞 收藏 分享

下塘烧饼

路漫漫其修远兮，吾将上下而求索。

52 声望 **11** 粉丝

关注作者

0 条评论

得票 最新

撰写评论 ...

提交评论

继续阅读

JVM知识梳理之一_JVM运行时内存区域与Java内存模型

Java虚拟机在执行Java程序时，会将分配给JVM的内存划分为几个不同的区域。有些区域在JVM启动之后就...

下塘烧饼 阅读 727