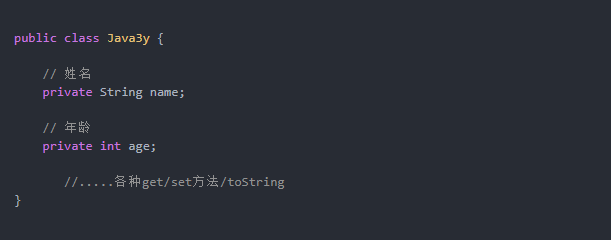


# 一、简单聊聊JVM

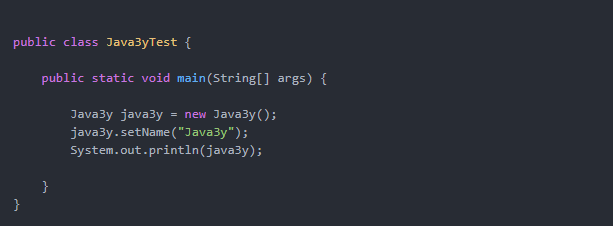
原文：<https://www.jianshu.com/p/904b15a8281f>

## 1.1我们现在有一个简单的Java程序

现在我有一个JavaBean：



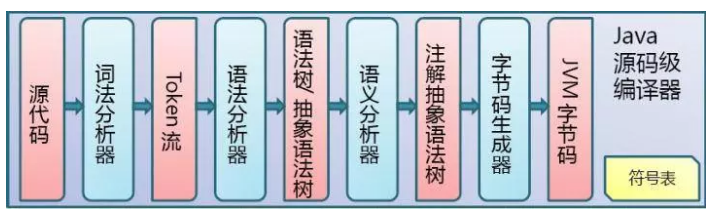
一个测试类：



我们在初学的时候肯定用过**javac来编译.java文件代码**，用过**java命令来执行编译后生成的.class文件**。

## 1.2编译过程

.Java文件是由Java源码编译器（javac.exe）来完成，流程图如下：



Java源码编译由以下三个过程组成：

1.分析和输入到符号表

2.注解处理

3.语义分析和生成class文件



### 1.2.1编译时期-语法糖

**语法糖可以看做是编译器实现的一些“小把戏”，这些“小把戏”可能会使得效率“大提升”。**

最值得说明的就是泛型了，这个语法糖可以说我们是经常会使用到的！

泛型只会在Java源码中存在，编译过后会被替换为原来的原生类型（Raw Type，也称为裸类型）了。这个过程也被称为：泛型擦除。

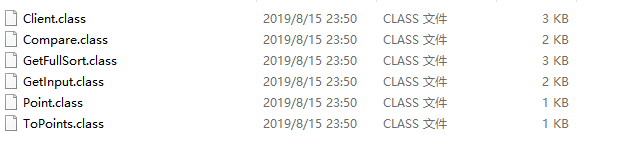
有了泛型这颗语法糖以后：

代码更加简洁【不用强制转换】

程序更加健壮【只要编译时期没有警告，那么运行时期就不会出现ClassCastException异常】

可读性和稳定性【在编写集合的时候，就限定了类型】

## 1.3JVM实现跨平台



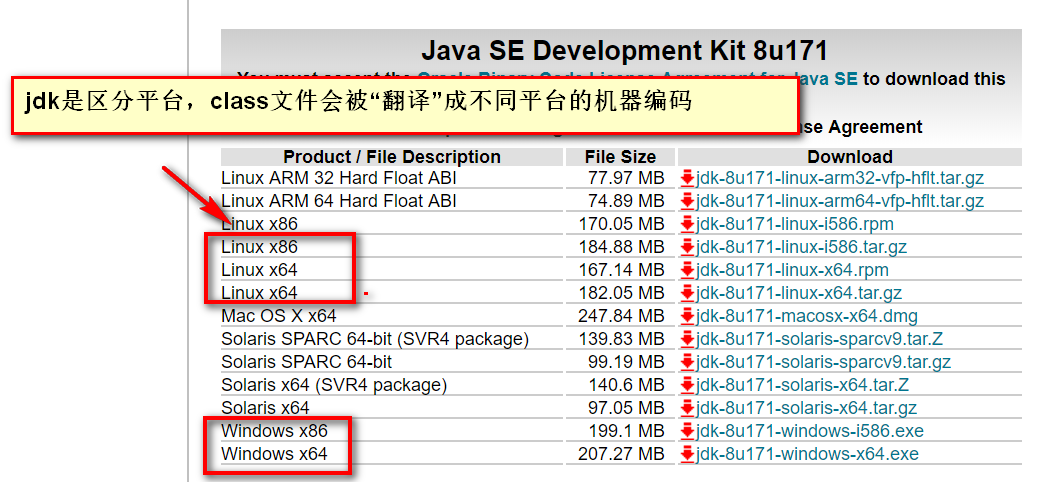
这些.class文件很明显是不能直接运行的，它不像C语言(编译cpp后生成exe文件直接运行)

这些.class文件是交由JVM来解析运行！

（1）JVM是运行在操作系统之上的，每个操作系统的指令是不同的，而JDK是区分操作系统的，只要你的本地系统装了JDK，这个JDK就是能够和当前系统兼容的。

（2）而class字节码运行在JVM之上，所以不用关心class字节码是在哪个操作系统编译的，3.只要符合JVM规范，那么，这个字节码文件就是可运行的。

（3）所以Java就做到了跨平台--->一次编译，到处运行！



## 1.4class文件和jvm

### 1.4.1类的加载时机

虚拟机规范严格规定了有且只有5种情况必须对类进行初始化（class文件加载到JVM中）

（1）创建类的实例（new的方式）。访问某个类或接口的静态变量，或者对该静态变量赋值，调用类的静态方法

（2）反射的方式

（3）初始化某个类的子类，则其父类也会被初始化

（4）Java虚拟机启动时被表明为启动类的类，直接使用java.exe命令来运行某个主类（包含main方法的那个类）

（5）当使用JDK1.7的动态语言支持时…

### 1.4.2如何将类加载到JVM

Class文件是通过**类的加载器**装载到JVM中的

Java默认有三种类加载器：



各个加载器的工作责任：

1）Bootstrap ClassLoader：由C++编写，是JVM的一部分，负责加载$JAVA\_HOME中jre/lib/**rt.jar等**class，不是ClassLoader子类。**JVM启动后，BootStrap ClassLoader也随着启动，负责加载完核心类库后，并构造Extension ClassLoader 和App ClassLoader。**

不能被Java代码访问到。又叫**启动类加载器**

2）Extension ClassLoader：负责加载java平台中**扩展功能**的一些jar包，即$JAVA\_HOME中jre/lib/ext指定目录下的jar包。是Java代码可以访问到的类加载器的最顶端，**没有父加载器。又叫扩展类加载器**

3）App ClassLoader：负责记载**classpath**中指定的jar包及**目录中class**

#### JVM加载class文件到内存的两种方式

1.隐式加载：不通过在代码中调用ClassLoader来加载需要的类，而是通过JVM来自动加载需要的类到内存。例如，当类中继承或引用某个类时，JVM在解析当前这个类不在内存中时，就会自动将这些类加载到内存

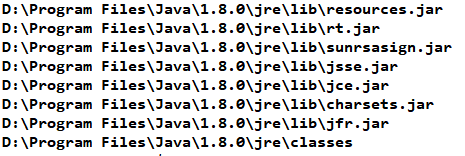
2.显示加载：在代码中通过ClassLoader类来加载一个类，例如调用this.getClass.getClassLoader().loadClass(…)或者class.forName(…)

#### 打印各加载器加载类的路径

1.BootStrap ClassLoader **启动类加载器**

1. System.getProperty("sun.boot.class.path").split(";");

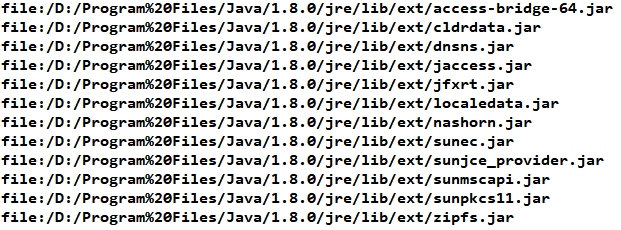
**打印结果：**



2.Extension ClassLoader**扩展类加载器**

1. URLClassLoader extClassLoader = (URLClassLoader)ClassLoader.getSystemClassLoader().getParent();
2. urls = extClassLoader.getURLs();
3. **for**(URL url : urls)
4. System.out.println(url);

**打印结果：**



3.App ClassLoader应用类加载器

1. URLClassLoader appClassLoader = (URLClassLoader)ClassLoader.getSystemClassLoader();
2. urls = appClassLoader.getURLs();
3. **for**(URL url : urls)
4. System.out.println(url);

打印结果：如果是在eclipse环境下路径则是 **当前工程路径/bin**

### 1.4.3类加载详细过程

加载器加载到jvm中，接下来其实又分了好几个步骤：

加载，查找并加载类的二进制数据，在Java堆中也**创建一个java.lang.Class类的对象**。

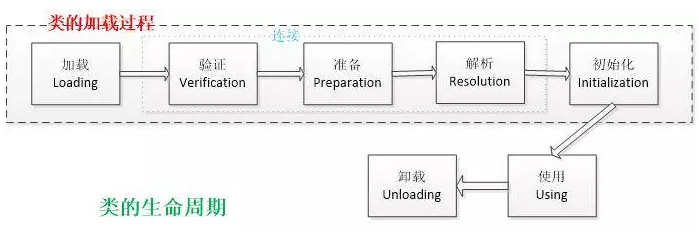
**连接，连接又包含三块内容：验证、准备、初始化。**

**1）验证，文件格式、元数据、字节码、符号引用验证；**

**2）准备，为类的静态变量分配内存，并将其初始化为默认值；**

**3）解析，把类中的符号引用转换为直接引用**

初始化，为类的静态变量赋予正确的初始值。



### 1.4.4JIT即时编辑器

一般我们可能会想：JVM在加载了这些class文件以后，针对这些字节码，逐条取出，逐条执行-->解析器解析。

但如果是这样的话，那就**太慢**了！

我们的JVM是这样实现的：

就是把这些**Java字节码重新编译优化，生成机器码，让CPU直接执行。这样编出来的代码效率会更高。**

编译也是要花费时间的，我们一般对热点代码做编译，非热点代码直接解析就好了。

**热点代码解释：一、多次调用的方法。二、多次执行的循环体**

使用热点探测来检测是否为热点代码，热点探测有两种方式：

采样

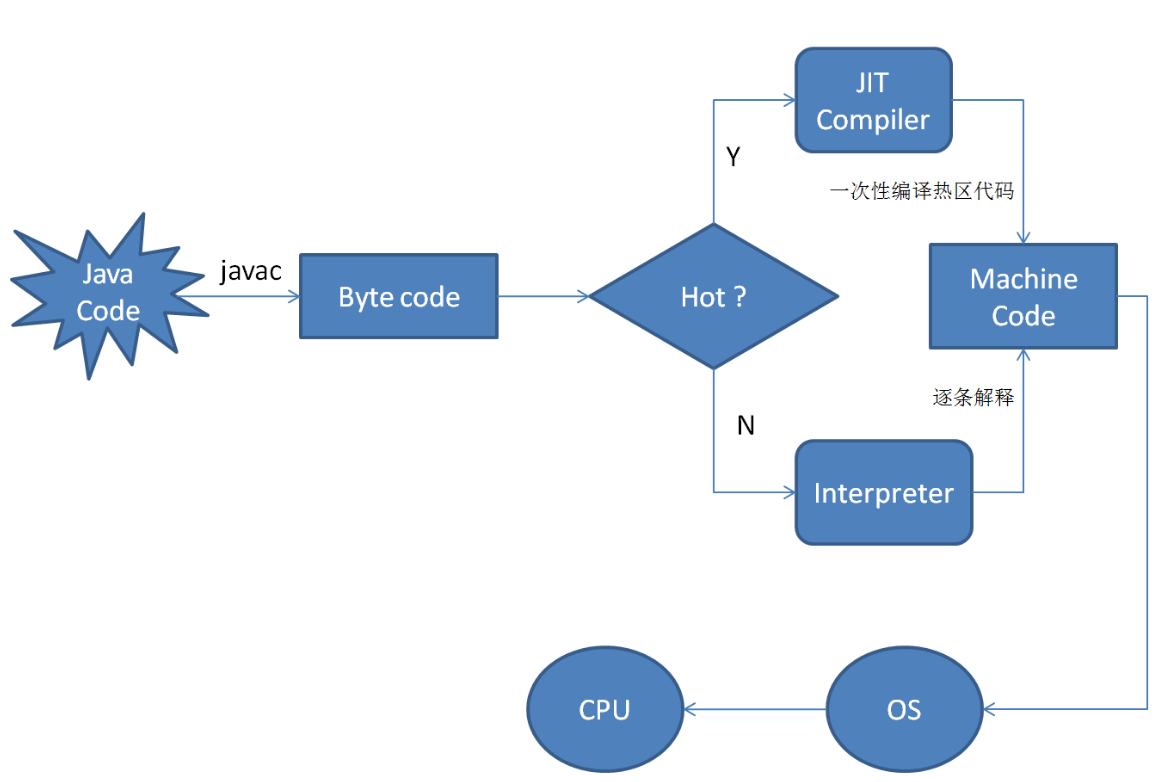
计数器

目前HotSpot使用的是计数器的方式，它为每个方法准备了两类计数器：

方法调用计数器（Invocation Counter）

回边计数器（Back EdgeCounter）。

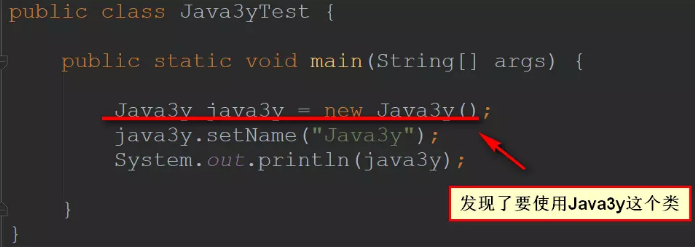
在确定虚拟机运行参数的前提下，这两个计数器都有一个确定的阈值，当计数器超过阈值溢出了，就会触发JIT编译。



### 1.4.5

按我们程序来走，我们自己写的**Java3yTest.class**文件会被AppClassLoader加载器(因为ExtClassLoader和BootStrap加载器都不会加载它[双亲委派模型])加载到JVM中。

随后发现了要使用Java3y这个类，我们的**Java3y.class**文件会被AppClassLoader加载器(因为ExtClassLoader和BootStrap加载器都不会加载它[双亲委派模型])加载到JVM中



### 1.4.6 ClassLoader.class部分源码分析

**Extension ClassLoader扩展类加载器，没有父加载器**

1. **protected** Class<?> loadClass(String name, **boolean** resolve)
2. **throws** ClassNotFoundException
3. {
4. **synchronized** (getClassLoadingLock(name)) {
5. // 首先检查类是否已经被当前类加载器加载
6. Class<?> c = findLoadedClass(name);
7. **if** (c == **null**) {
8. //没有被加载
9. **long** t0 = System.nanoTime();
10. **try** {
11. **if** (parent != **null**) {  //先让父加载类加载类
12. c = parent.loadClass(name, **false**);
13. } **else** {  //父加载器为空，说明当前的类加载器是Extension ClassLoader
14. c = findBootstrapClassOrNull(name);  //尝试让启动类加载器加载类
15. }
16. } **catch** (ClassNotFoundException e) {
17. // ClassNotFoundException thrown if class not found
18. // from the non-null parent class loader
19. }
21. **if** (c == **null**) {
22. // If still not found, then invoke findClass in order
23. // to find the class.
24. **long** t1 = System.nanoTime();
25. //尝试自己加载类
26. c = findClass(name);
28. // this is the defining class loader; record the stats
29. sun.misc.PerfCounter.getParentDelegationTime().addTime(t1 - t0);
30. sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);
31. sun.misc.PerfCounter.getFindClasses().increment();
32. }
33. }
34. **if** (resolve) {
35. resolveClass(c);
36. }
37. **return** c;
38. }
39. }

**工作过程：**

1.系统首先会默认调用底层类加载器，App ClassLoader。先检查是否已经被当前类加载器加载，如果已经被加载，则直接返回类的Class<?>实例

2.没有被加载，则看是否有父加载器，这里肯定会有，因为Extension ClassLoader就是App ClassLoader的父加载器

3.调用Extension ClassLoader，首先检查当前类加载器是否已经加载过，有的话就直接返回类的Class<?>实例。否则就看有没有父加载器，这里肯定没有，因为Extension Class Loader没有父加载器，没有父加载器，就看看启动器类加载器有没有加载过，加载过的话就返回Class<?>实例，没有的话返回null。

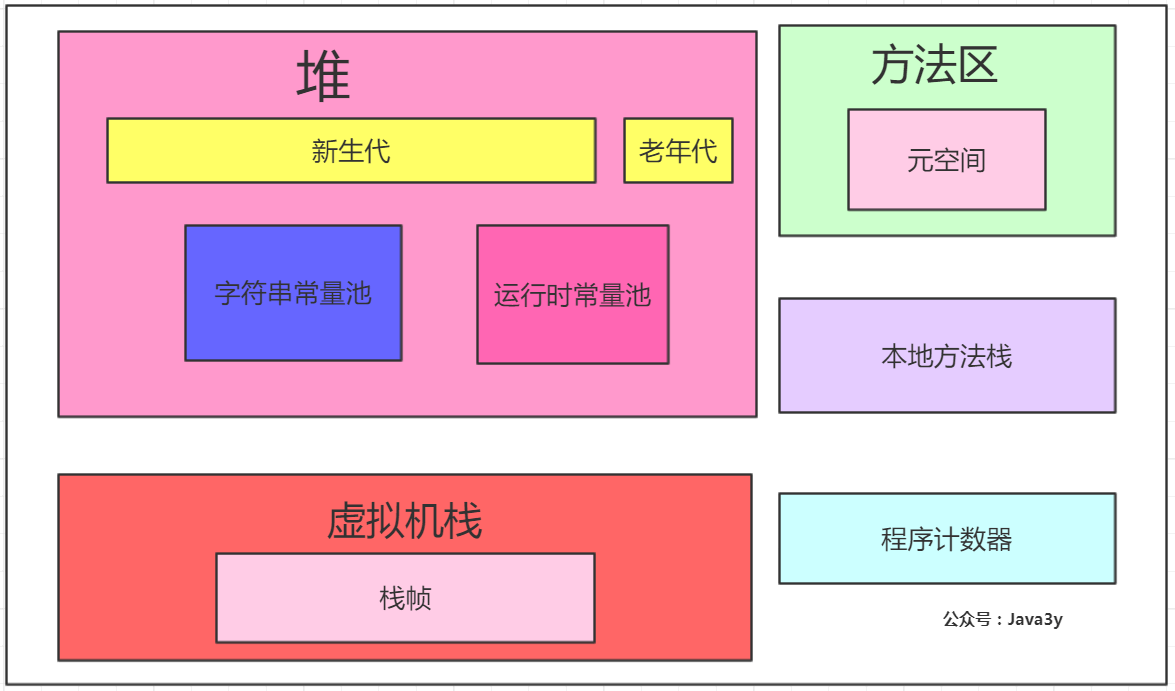
4.如果BootStrap ClassLoader没有加载过，Extension ClassLoader会尝试进行加载，加载成功的话就返回Class<?>实例，失败的话就返回null。

5.如果Extension ClassLoader加载失败，则App ClassLoader会自己尝试进行加载

## 1.5类加载完以后JVM干了什么

在类加载检查通过后，接下来虚拟机将为新生对象分配内存。

### 1.51JVM的内存模型（基于JDK1.8



简单看看每个区域究竟存储的是什么(干的是什么)：

1）堆：存放对象实例，几乎所有的对象实例都在这里分配内存

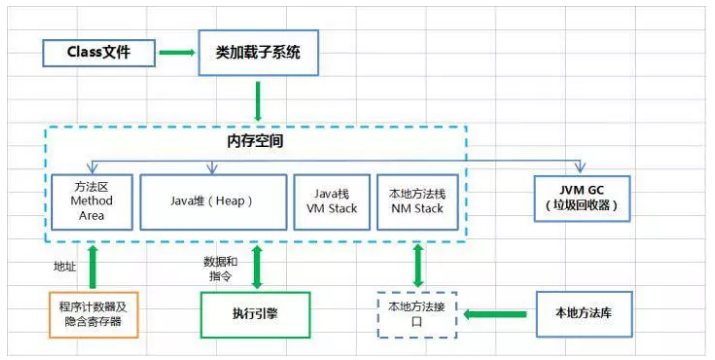
2）虚拟机栈：虚拟机栈描述的是Java方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于存储局部变量表、操作栈、动态链接、方法出口等信息

3）本地方法栈：本地方法栈则是为虚拟机使用到的Native方法服务。

4）方法区：存储已被虚拟机加载的类元数据信息(元空间)

5）程序计数器：当前线程所执行的字节码的行号指示器

### 1.5.2例子中的流程



我来宏观简述一下我们的例子中的工作流程：

1、通过**java.exe**运行**Java3yTest.class**，随后被加载到JVM中，元空间存储着类的信息(包括类的名称、方法信息、字段信息..)。

2、然后JVM找到**Java3yTest**的主函数入口(main)，**为main函数创建栈帧**，开始执行main函数

3、main函数的第一条命令是**Java3y java3y = new Java3y()**;就是让JVM创建一个Java3y对象，**但是这时候方法区中没有Java3y类的信息，所以JVM马上加载Java3y类，把Java3y类的类型信息放到方法区中(元空间)**

4、加载完Java3y类之后，Java虚拟机做的第一件事情就是在堆区中为一个**新的Java3y实例分配内存, 然后调用构造函数初始化Java3y实例**，**这个Java3y实例持有着指向方法区的Java3y类的类型信息（其中包含有方法表，java动态绑定的底层实现）的引用**

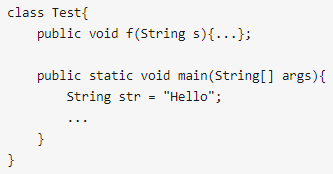
5、当使用java3y.setName("Java3y");的时候，JVM根据java3y引用找到Java3y对象，然后根据Java3y对象持有的**引用定位到方法区中Java3y类的类型信息的方法表，获得setName()函数的字节码的地址**

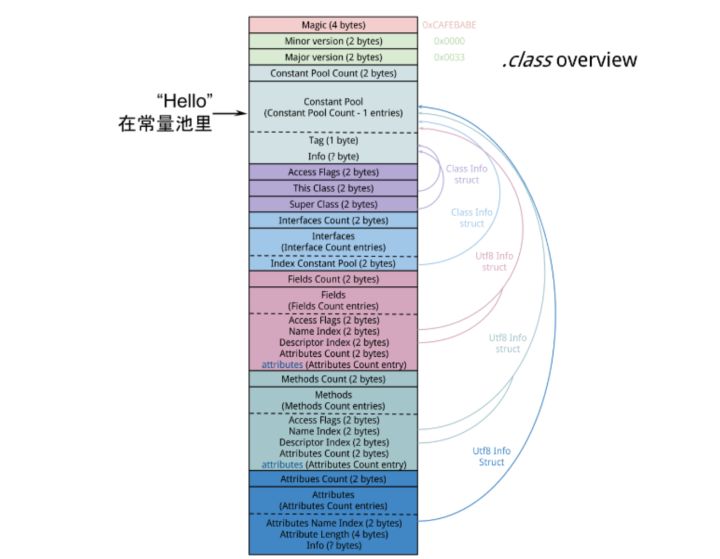
6、为setName()函数创建栈帧，开始运行setName()函数

从微观上其实还做了很多东西，正如上面所说的类加载过程（加载-->连接(验证，准备，解析)-->初始化)，在类加载完之后jvm为其分配内存(分配内存中也做了非常多的事)。由于这些步骤并不是一步一步往下走，会有很多的“混沌bootstrap”的过程，所以很难描述清楚。

## 1.6 .class文件总览

例子：





**编译成.class文件后，除了版本、字段、方法、接口等描述信息外，还有一个叫“常量池（Constant Pool Tables）”的东西。主要有两个东西：“字面量（Literal）”和“符号引用量（Symbolic Reference）”。**

**字面量——其中字面量就包括类中定义的一些常量，因为String是不可变的，由final关键字修饰过了，所以代码里的“Hello”字符串，就是作为字面量（常量）写在class的常量池里。**

**符号引用——1.类与接口的全限定名**

**2.字段的名称和描述符**

**3.方法的名称和描述符**

**运行程序用到类的时候，.class文件的信息就会被解析到内存的方法区里。.class文件里常量池的大部分数据会被加载到 运行时常量池 。String不是，本体还是和所有对象一样创建在heap堆区，其引用会被存到 字符串常量池 里。**

**注意：这只是类被类加载器加载时候的情形。主线程中的str变量这时候都还没被创建，但“hello”的实例已经在heap堆中了，对它的引用也已经在字符串常量池中了**

**等主线程开始创建str变量的时候，虚拟机就回到字符串常量池中找，看有没有能equal（“hello”）的String。如果找到了，就在栈区当前栈帧的局部变量表里创建str变量，然后把字符串常量池里对“hello”对象的引用赋值给str变量。找不到的话，才会在heap堆重新创建一个变量，然后把引用驻留到字符串常量池，然后再把引用复制给栈帧的局部变量表**

**类加载后，常量池中的数据会在运行时常量池中存放！**

## 1.7各种常量池

**JDK1.7后**

**运行时常量池位于堆中——它是每个类私有的。后面会说到，每个class文件里的“常量池”在类被加载器加载之后，就映射存放在这个地方。**

**字符串常量池也位于堆中——它是全局共享的。字符串常量池只存储引用，不存储内容！**

**常量池（Constant Pool Table）（每个class文件里的“常量池”）位于方法区，JDK8后在元空间里**