*上次我们学习了关于GC的一些东西，什么是****GC Root****？GC都有哪些算法？不同GC算法各自的****优劣****？堆中的对象是****如何分代****的？****为什么****要分代？有哪些****垃圾收集器****？不同垃圾收集器的****优劣****？现在****投入使用****的是哪种收集器？这其中又有很多的细节：如何处理****内存碎片****？如何看待GC停顿时间？还有，如何在GC中把堆和栈结合起来？(栈中的****OopMap****储存引用类型，作为GC Roots在堆中寻找****引用链****)堆和栈到底是如何交互的呢？*

这次我们来学习**类加载**的机制，我们用一个问题来作为本次学习的开端：一个**Java程序为什么可以跨平台运行**？

众所周知，在我们完成一个Java程序之后，这个程序就会被编译器编译为.**class字节码**文件。由于**不同的硬件平台有不同的JVM**，这些不同的JVM可以把字节码文件翻译为对应自己硬件平台能够执行的机器码。这里需要注意几个点：一个是Java程序被编译为**.class字节码**文件，另一个是**不同的硬件平台**有对应自己的JVM，最后一个是相同的.class字节码文件被不同的JVM翻译成为针对不同平台的不同机器语言(基本上是复述了一遍刚才的东西…)。

明白了上述问题，我们就可以思考一些细节了：一个Java程序，一个Java类，是如何被编译成.class字节码文件的？编译出来的字节码文件长啥样子？如何加载运行这个字节码文件？这个字节码文件在哪里运行？

1. **Java程序编译成为一个.class字节码文件**

一个写好的Java程序，是如何编译成为.class字节码文件的呢？

其实我也不了解其中的细节，网上查阅了一些资料，比较少也比较难懂…毕竟编译方面的东西，太底层了。我们需要知道的就是一个Java程序是由javac编译器编译成为.class字节码文件的即可。

那么编译出来的.class文件长啥样子呢？

这时候，我们就需要了解Class类文件的结构了。

以下是摘自《深入理解Java虚拟机的片段》:

Class类文件的结构

Class文件是一组以**8位字节**为基础单位的**二进制流**，各个数据项目严格按照顺序紧凑地**排列**在Class文件之中，中间没有添加任何分隔符，这使得整个Class文件中存储的内容几乎全部都是**程序运行的必要数据**，没有空隙存在。当遇到需要占用8位字节以上的空间的数据项时，则会按照高位在前的方式**分割成若干个8位字节**进行存储。

根据Java虚拟机规范的规定，Class文件格式采用一种类似于C语言结构体的伪结构来存储，这种伪结构中只有两种数据类型：**无符号数**和**表**。

无符号数属于基本的数据类型，以**u1**、**u2**、**u4**、**u8**来分别代表1个字节、2个字节、4个字节、8个字节的无符号数，无符号数可以用来描述数字、索引引用、数量值，或者按照UTF-8编码构成字符串值。

表是由多个无符号数或者其它表作为数据项构成的复合数据类型，所有表都习惯性地以"\_info"结尾。表用于描述有层次关系的复合结构的数据，整个Class文件本质上就是一张表。

其他的就不多说了，自己可以百度，太底层的东西看起来是挺有难度的。。。

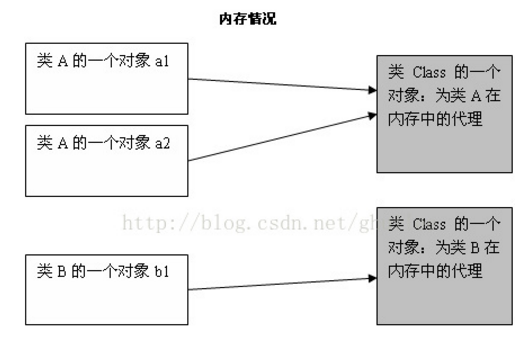
1. **Class加载**

当我们成功的把一个Java程序编译成为.class字节码文件之后，会对class进行装载，读取类的二进制流，然后把类的信息存储至方法区中，然后在堆中生成java.lang.Class对象。

也就是说，在加载阶段，JVM做了一下几件事：

1. 通过全限定类名，获取类的二进制流
2. 把这个转化为方法区的运行时数据结构
3. 将内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口

这里需要说一下java.lang.Class，每种不同的类型都有自己对应的Class对象，Class对象和对应类型的类的实例的关系可以用一张图来描述：



关于Class这个东西其实还有很多其他的东西需要摆，比如类的反射机制等等

**3. 连接**

连接总共分为三个步骤：验证，准备，解析。

1. 验证：验证的目的是为了**确保类的二进制流是正确的**，是**符合当前虚拟机规范**的，那么怎么判断类的二进制流是否正确呢？
2. **文件格式**验证：是否以**魔数0xCAFEBABE**开头？主次**版本号**是否合理？
3. **元数据**验证：检查类是**final**的类不拥有**子类**？检查**final**的**方法**不能被**覆盖**？
4. **字节码**验证：检查字节码的**完整性**，听说很复杂，就先不看了
5. **符号引用**验证：**常量池**中描述的类是否存在，访问的方法或字段是否有足够的权限。

这里要解释一下这个常量池，常量池分为两种，一种是运行时常量池，一种是静态常量池。

静态常量池是Class文件中的常量池，前面我们提到了**Class字节码文件的结构**，一个字节码文件是由魔数，版本号，紧接着就是常量池信息组成的。Class字节码文件中的这个常量池，就是静态常量池。

下面引申一下这两种常量池的概念：

1. **静态常量池**里保存了**字面量**和**符号引用量**，我们需要解释一下，什么是字面量，什么是符号引用量。

**字面量其实就是常量**，比如int a = 10,String str = “abc”，这里的**10**和**abc**就是字面量，再比如**声明为final的变量**也是一种常量：**final int** a = 10，那么这个a也可以作为字面量，常量。

**符号引用量**包括了三种类型的**常量**：**类和接口的全限定名，字段名称和描述符，方法名称和描述符**。(这部分不怎么看懂，以后再看的时候深入看一下)

2.**运行时常量池**：运行时常量池存在于方法区中，相对于静态的class常量池，运行时常量池注重的是**动态性**，有些常量不一定是编译时就产生的，运行时也会产生常量，比如String.intern()。常量池的好处之一就是节省内存空间，另一方面是在比较的时候直接比较值，而不是比较引用，节省运行时间。

1. 准备：

在方法区中，为各个对象和变量分配内存，并且为类设置初始值。

比如static int a = 1这种，在准备期间，a的值就是0，在初始化的时候才会变为1。

但像常量：static final int a = 1，在准备期间，a的值就是1。

1. 解析：

在这个阶段，**符号引用将会被转化为直接引用**。

刚才并没有说清什么是符号引用，举一个简单的例子：在Java程序编译期间，比如你的类java.exskil.A引用了java.exskil.B，但是编译的时候并不能直接获取到类B的引用，因此只能用一串描述符比如java.exskil.B(描述符)的**常量**来代替，说明这里应该是B类引用的地址，但是我现在还不知道，所以就用B类的一个描述符常量先来替代了。

直接引用就相当于指针，代表了B的地址。

**3. 初始化<clinit>(** **classinitialize阶段)**

把静态变量赋值为初始值，执行static{}静态代码块。

如果这个类有父类，而且父类还没有被加载，那么久停下来，先加载他的父类。

**Classinitialize方法是线程安全的，每次初始化一个类，都要加载Classinitialize方法。**

什么时候类会初始化？

1. new的时候
2. 使用反射机制的时候(Class.forName())
3. 初始化一个类的子类的时候
4. 调用类的静态方法的时候

**4. ClassLoader类加载器**

什么是ClassLoader？

1. ClassLoader是一个抽象类
2. ClassLoader的实例负责读入Java字节码文件，把类装载至JVM中
3. ClassLoader负责类的装载阶段中的加载部分
4. 你可以根据针对不同字节码流的不同获取方式，自定义自己的ClassLoader

**类加载的动态性**：

一个Java程序肯定是由很多个类组成的，但是程序启动时，并不是一次性就把所有程序加载完成的，而是**先加载一部分保证程序能够正常运行的基础类**一次性加载至内存中，其他的类等到**需要**使用到时在加载，减少了**内存**的**开销**。

都有哪些**ClassLoader**？

1. App ClassLoader：加载自己写的类，也就是当前应用的classpath的所有类

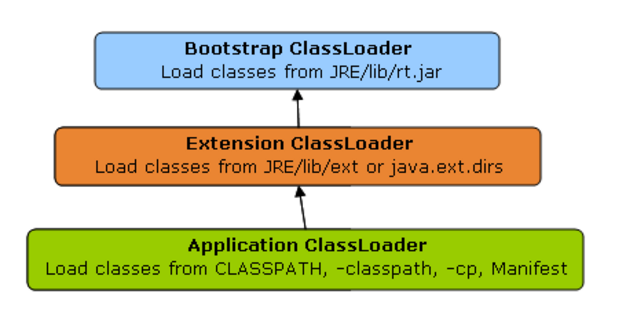
(classpath是什么? 它的作用是什么？

classpath是javac编译器的一个环境变量。

它的作用是指定类搜索路径，它与import、package关键字有关。

CLASSPATH 包含了**一个或多个目录**，它们作为一种特殊的“**根**”使用，从这里**展开对.class 文件的搜索**。从那个根开始，解释器会寻找包名，并将每个点号（句点）替换成一个斜杠，从而生成从CLASSPATH 根开始的一个路径名（所以package foo.bar.baz 会变成foo\bar\baz 或者foo/bar/baz；具体是正斜杠还是反斜杠由操作系统决定）。随后将它们连接到一起，成为CLASSPATH 内的各个条目（入口）。**以后搜索.class 文件时，就可从这些地方开始查找与准备创建的类名对应的名字**。)

1. BootStrap ClassLoader：最**顶层**的加载类，**由C++语言编写(所以Java中是看不到他的哦)**，主要加载核心类库。%JRE\_HOME%\lib，目录下的rt.jar，resources.jar的文件
2. Extention ClassLoader：**扩展**的类加载器，主要加载%JRE\_HOME%\lib\ext目录下的jar包和class文件
3. 一幅图描述三个ClassLoader之间的关系



在类的加载的时候，是从顶层向下，先从Bootstrap ClassLoader开始的。

在类的查找的时候，是从底层向上，先从AppClassLoader开始查找的，若AppClassLoader能够查找到该类，就不在向上继续查找了。

**双亲委派模型：**

**父加载器**的概念：AppClassLoader的父加载器是ExtClassLoader，ExtClassLoader的父加载器是BootstrapClassLoader，但是由于BootstrapClassLoader是由**C++**写的，所以**ExtClassLoader获取父加载器是null**的。

当你需要加载一个类的时候，比如java.exskil.A这个类，他会先在AppClassLoader的缓存里看看这个类**是否已经被加载**，如果没有，那么首先会将加载任务**委托给父类**，让父类去加载这个类，依递归。如果父类可以加载，那就成功返回；如果父类不能加载，那就让这个类自己加载。

这样做的好处？举个例子，java.lang.Object这个类在rt.jar下，这个类本来就是由BootstrapClassLoader加载的，在任何类加载器环境下都是唯一的这一个类，无论哪一个类加载器要加载这个类，最终都会到最顶层的BootstrapClassLoader中去加载。试想，如果用户自定义了一个名字为java.lang.Object的类，并且放在了ClassPath中，那么就会出现两个重名的类。