当程序使用某个类时，如果该类还未被加载到内存中，则JVM会通过加载、链接、初始化三个步骤对该类进行类加载。

一、类的加载、链接、初始化

### 1、加载

类加载：将类的class文件读入内存，并为之创建一个java.lang.Class对象。

类的加载过程由类加载器来完成，类加载器由JVM提供。开发人员也可以通过继承ClassLoader来实现自己的类加载器。

加载的class来源：

* 从本地文件系统内加载class文件
* 从JAR包加载class文件
* 通过网络加载class文件
* 把一个java源文件动态编译，并执行加载

### 2、类的链接

通过类的加载，内存中已经创建了一个Class对象。链接负责将二进制数据合并到JRE中。链接需要通过验证、准备、解析三个阶段。

验证：

检查被加载的类是否有正确的内部结构，并和其他类协调一致。即是否满足

java虚拟机的约束。

准备：

为类的类变量分配内存，并设置默认初始值。

解析：

引用其实对应于内存地址。思考这样一个问题，在编写代码时，使用引用，方

法时，类知道这些引用方法的内存地址吗？显然是不知道的，因为类还未被加载到虚拟机中，你无法获得这些地址。

举例来说，对于一个方法的调用，编译器会生成一个包含目标方法所在的类、目标方法名、接收参数类型以及返回值类型的符号引用，来指代要调用的方法。

解析阶段的目的，就是将这些符号引用解析为实际引用。如果符号引用指向一个未被加载的类，或者未被加载类的字段或方法，那么解析将触发这个类的加载（但未必会触发解析与初始化）。

### 3、类的初始化

类的初始化阶段，虚拟机主要对类变量进行初始化。虚拟机调用<clinit>方法，进行类变量的初始化。

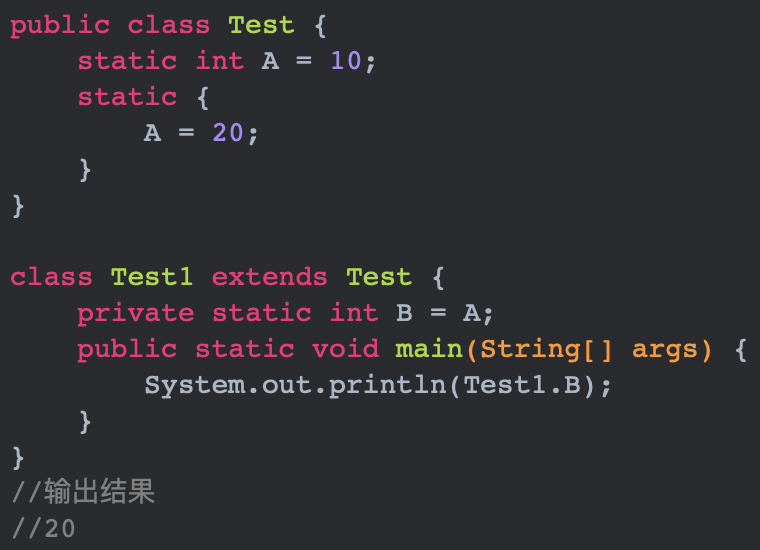
java类中对类变量进行初始化的两种方式：

1. 在定义时初始化
2. 在静态初始化块内初始化

#### 3.1 < clinit>方法相关

虚拟机会收集类及父类中的类变量及类方法组合为< clinit>方法，根据定义的顺序进行初始化。虚拟机会保证子类的< clinit>执行之前，父类的< clinit>方法先执行完毕。

因此，虚拟机中第一个被执行完毕的< clinit>方法肯定是java.lang.Object方法。



父类的静态初始化块在子类静态变量初始化之前初始化完毕，所以输出结果是20，不是10。

如果类或者父类中都没有静态变量及方法，虚拟机不会为其生成< clinit>方法。

接口与类不同的是，执行接口的＜clinit＞方法不需要先执行父接口的

＜clinit＞方法。只有当父接口中定义的变量使用时，父接口才会初始化。另外，接口的实现类在初始化时也一样不会执行接口的＜clinit＞方法。



对于接口，只有真正使用父接口的类变量才会真正的加载父接口。这跟普通类加载不一样。

虚拟机会保证一个类的< clinit>方法在多线程环境中被正确地加锁和同步，如果多个线程同时去初始化一个类，那么只有一个线程去执行这个类的< clinit>方法，其他线程都需要阻塞等待，直到活动线程执行< clinit>方法完毕。

#### 3.2 类初始化时机

1. 当虚拟机启动时，初始化用户指定的主类；
2. 当遇到用以新建目标类实例的new指令时，初始化new指令的目标类；
3. 当遇到调用静态方法或者使用静态变量，初始化静态变量或方法所在的类；
4. 子类初始化过程会触发父类初始化；
5. 如果一个接口定义了default方法，那么直接实现或者间接实现该接口的类的初始化，会触发该接口初始化；
6. 使用反射API对某个类进行反射调用时，初始化这个类；
7. Class.forName()会触发类的初始化

#### 3.3 final定义的初始化

注意：对于一个使用final定义的常量，如果在编译时就已经确定了值，在引用时不会触发初始化，因为在编译的时候就已经确定下来，就是“宏变量”。如果在编译时无法确定，在初次使用才会导致初始化。

如果final定义的变量在编译时无法确定，则在使用时还是会进行类的初始化。

#### 3.4 ClassLoader只会对类进行加载，不会进行初始化

ClassLoader只会对类进行加载，不会进行初始化；使用Class.forName()会强制导致类的初始化。

二、类加载器

类加载器负责将.class文件（不管是jar,还是本地磁盘，还是网络获取等等）加载到内存中，并为之生成对应的java.lang.Class对象。一个类被加载到JVM中，就不会第二次加载了。

那怎么判断是同一个类呢？

每个类在JVM中使用全限定类名（包名+类名）与类加载器联合为唯一的ID，所以如果同一个类使用不同的类加载器，可以被加载到虚拟机，但彼此不兼容。

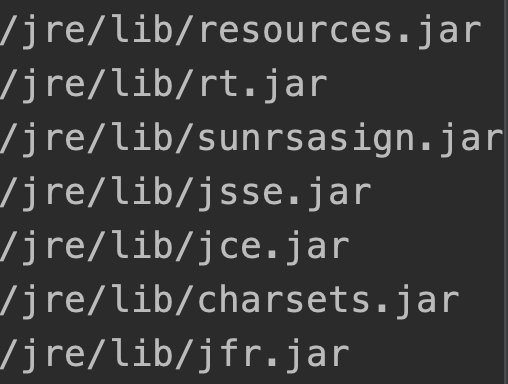
### 1、JVM类加载器分类

#### 1.1 Bootstrap ClassLoader根类加载器

根类加载器，负责加载java的核心类库。根加载器不是ClassLoader的子类，是由C++实现的。

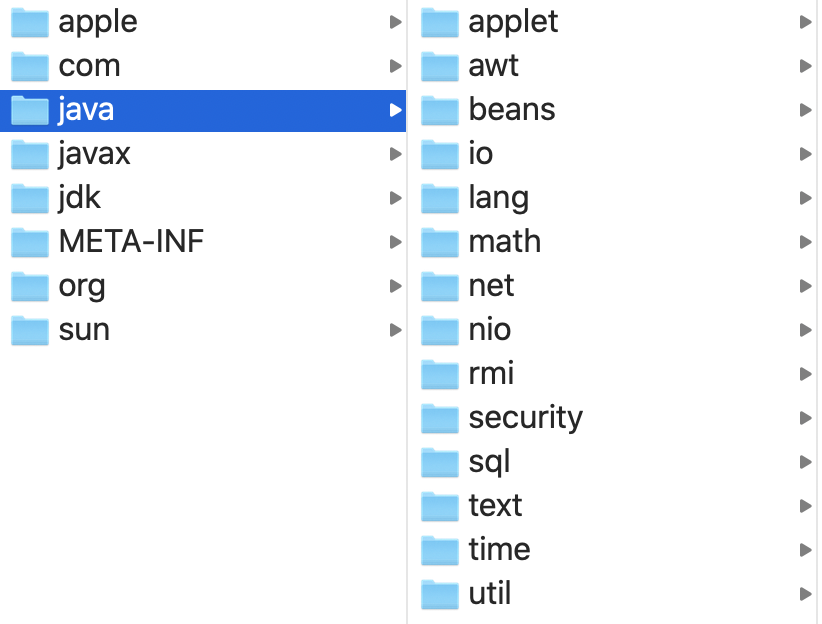
// 获取根类加载器所加载的全部URL数组

URL[] urLs = Launcher.*getBootstrapClassPath*().getURLs();  
Arrays.*stream*(urLs).forEach(System.*out*::println);



根类加载器负责加载%JAVA\_HOME%/jre/lib下的jar包（以及由虚拟机参数 -Xbootclasspath 指定的类）。

解压rt.jar，可以看到我们经常使用的类库就在这个jar包中。



#### 1.2 Extension ClassLoader扩展类加载器

负责加载%JAVA\_HOME%/jre/ext或者java.ext.dirs系统熟悉指定的目录的jar包。大家可以将自己写的工具包放到这个目录下，可以方便自己使用。

#### 1.3 System ClassLoader系统(应用)类加载器

System ClassLoader为系统（应用）类加载器，负责加载加载来自java命令的-classpath选项、java.class.path系统属性，或者CLASSPATH环境变量所指定的JAR包和类路径。程序可以通过ClassLoader.getSystemClassLoader()来获取系统类加载器。如果没有特别指定，则用户自定义的类加载器默认都以系统类加载器作为父加载器。

三、类加载机制

### 1、JVM主要的类加载机制

* 全盘负责：当一个类加载器负责加载某个Class时，该Class所依赖和引用的其他Class也由该类加载器负责载入，除非显示使用另一个类加载器来载入。
* 父类委托（双亲委派）：先让父加载器试图加载该Class，只有在父加载器无法加载时该类加载器才会尝试从自己的类路径中加载该类。
* 缓存机制：缓存机制会将已经加载的class缓存起来，当程序中需要使用某个Class时，类加载器先从缓存区中搜寻该Class，只有当缓存中不存在该Class时，系统才会读取该类的二进制数据，并将其转换为Class对象，存入缓存中。这就是为什么更改了class后，需要重启JVM才生效的原因。

注意：类加载器之间的父子关系并不是类继承上的父子关系，而是实例之间的父子关系。



系统类加载器的父加载器是扩展类加载器。但输出中扩展类加载器的父加载器是null，这是因为父加载器不是java实现的，是C++实现的，所以获取不到。但扩展类加载器的父加载器是根加载器。

### 2、类加载流程图



红色部分可以是我们自定义实现的类加载器来进行加载。

四、创建并使用自定义类加载器

### 1、自定义类加载分析

除了根类加载器，所有类加载器都是ClassLoader的子类。所以我们可以通过继承ClassLoader来实现自己的类加载器。

ClassLoader类有两个关键的方法：

* protected Class loadClass(String name, boolean resolve)：name为类名，resove如果为true，在加载时解析该类。
* protected Class findClass(String name) ：根据指定类名来查找类。

所以，如果要实现自定义类，可以重写这两个方法来实现。但推荐重写findClass方法，而不是重写loadClass方法，因为loadClass方法内部会调用findClass方法。

loadClass()源码：

protected Class<?> loadClass(String name, boolean resolve)

throws ClassNotFoundException  
{  
 synchronized (getClassLoadingLock(name)) {  
 // 第一步，先从缓存里查看该类是否已经加载  
 Class<?> c = findLoadedClass(name);  
 if (c == null) {  
 long t0 = System.*nanoTime*();  
 try {

// 第二步，判断父加载器是否为null  
 if (parent != null) {  
 c = parent.loadClass(name, false);  
 } else {  
 c = findBootstrapClassOrNull(name);  
 }  
 } catch (ClassNotFoundException e) {  
 // ClassNotFoundException thrown if class not found  
 // from the non-null parent class loader  
 }  
  
 if (c == null) {  
 // 第三步，如果前面都没找到，就会调用findClass方法  
 // to find the class.  
 long t1 = System.*nanoTime*();  
 c = findClass(name);  
  
 // this is the defining class loader; record the stats  
 sun.misc.PerfCounter.*getParentDelegationTime*().addTime(t1 - t0);  
 sun.misc.PerfCounter.*getFindClassTime*().addElapsedTimeFrom(t1);  
 sun.misc.PerfCounter.*getFindClasses*().increment();  
 }  
 }  
 if (resolve) {  
 resolveClass(c);  
 }  
 return c;  
 }  
}

loadClass()加载流程：

1. 判断此类是否已经加载；
2. 如果父类加载器不为null，则使用父加载器进行加载；反之使用根加载器进行加载；
3. 如果前面都没有加载成功，则使用findClass()方法进行加载。

所以，为了不影响类的加载过程，我们重写findClass方法即可简单方便的实现自定义类加载。

### 2、实现自定义加载器

写了一个Hello测试类，并且编译过后放在了当前路径下（大家可以在findClass中加入判断，如果没有此文件，可以尝试查找.java文件，并进行编译得到.class文件；或者判断.java文件的最后更新时间大于.class文件最后更新时间，再进行重新编译等逻辑）。

ClassLoader还有一个重要的方法defineClass(String name, byte[] b, int off, int len)。此方法的作用是将class的二进制数组转换为Calss对象。