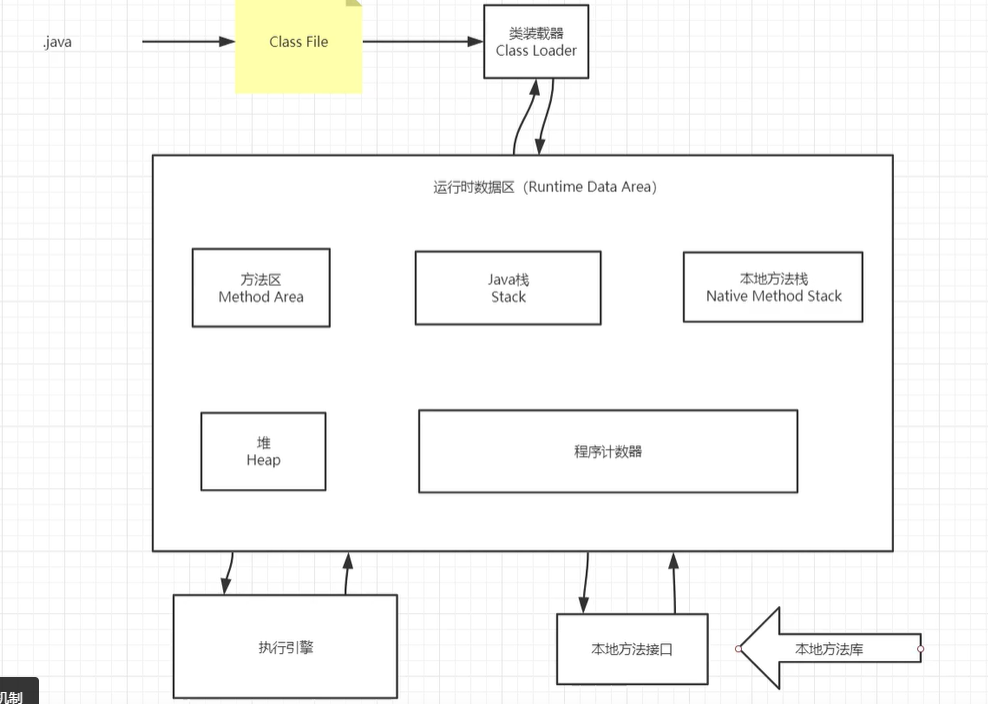
.java文件经过javac编译成class file，被class Loader装载，



STACK， Native Method Stack,程序计数器是不会存在垃圾回收的，因为有垃圾Stack就卡住了，就无法进入到main，垃圾回收主要是Heap

ClassLoader:类加载器

把Class文件加载到Java虚拟机中，但是他不会一次性把所有class文件都加载，而是按需动态加载，

什么时候类加载

(1)、调用静态成员时，会加载静态成员真正所在的类及其父类。

(2)、第一次 new 对象的时候 加载(第二次再 new 同一个类时，不需再加载)。

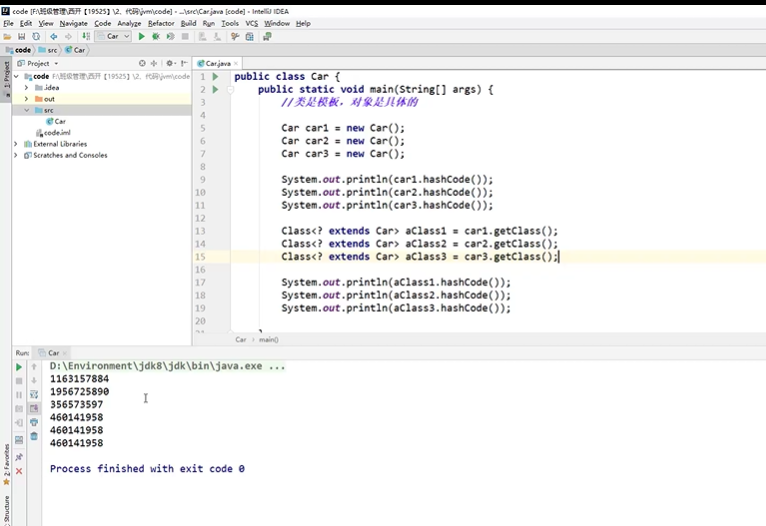
(3)、加载子类会先加载父类。（覆盖父类方法时所抛出的异常不能超过父类定义的范围）

如果静态属性有 final 修饰时，则不会加载，当成常量使用

例如new Student()的时候

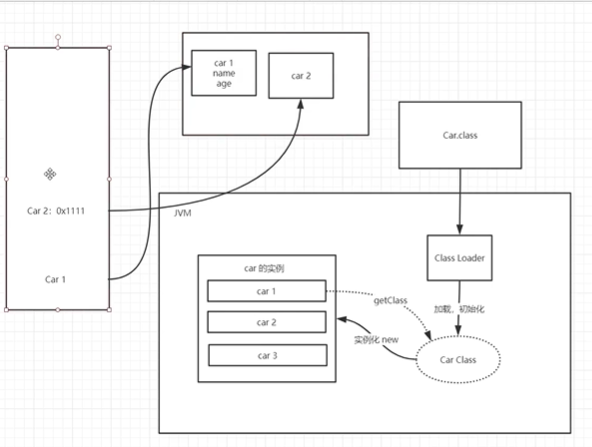
Student类本质上也是一个对象，某种程度上对象分为两种：实例对象和Class对象。每个类的运行时的**类型信息**就是用Class对象表示的。它包含了与类有关的信息。Class对象是不能直接用Constructor构造的，他只能通过在ClassLoader加载时由JVM悄悄摸摸的构造。相当于一个**模板**，//加载 然后通过new实例化具体的常规对象//初始化

先加载，再初始化



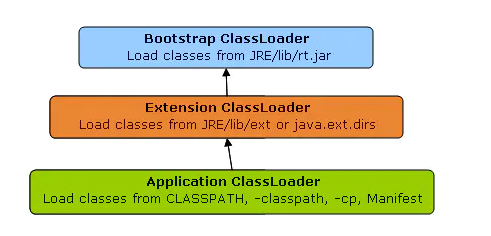
我们看这里，他们getClass得到的Class Object实际上指向的都是同一个模板，具体对象hashCode出来是不一样的

常规对象Car1引用是在栈Stack里，具体内容在Heap里



双亲委派机制

加载器是分层的



应用程序加载器Application ClassLoader，拓展类加载器，启动类/根加载器

常规的你写的Student啥由最底层的Application ClassLoader加载

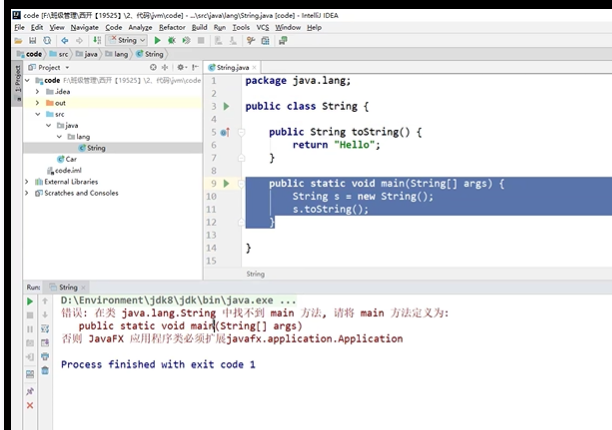
**工作原理**

（1）如果一个类加载器收到了类加载请求，它并不会自己先加载，而是把这个请求**委托给父类的加载器**去执行 //Application ClassLoader给Extension ClassLoader，Extension 给BootStrap，从下至上传递请求

（2）如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终将到达顶层的引导类加载器；

（3）父类加载器检查是否能完成类加载任务，如果可以，就加载对应Class，倘若父类加载器无法完成加载任务，子加载器才会尝试自己去加载，这就是双亲委派机制 //而处理则是从上往下

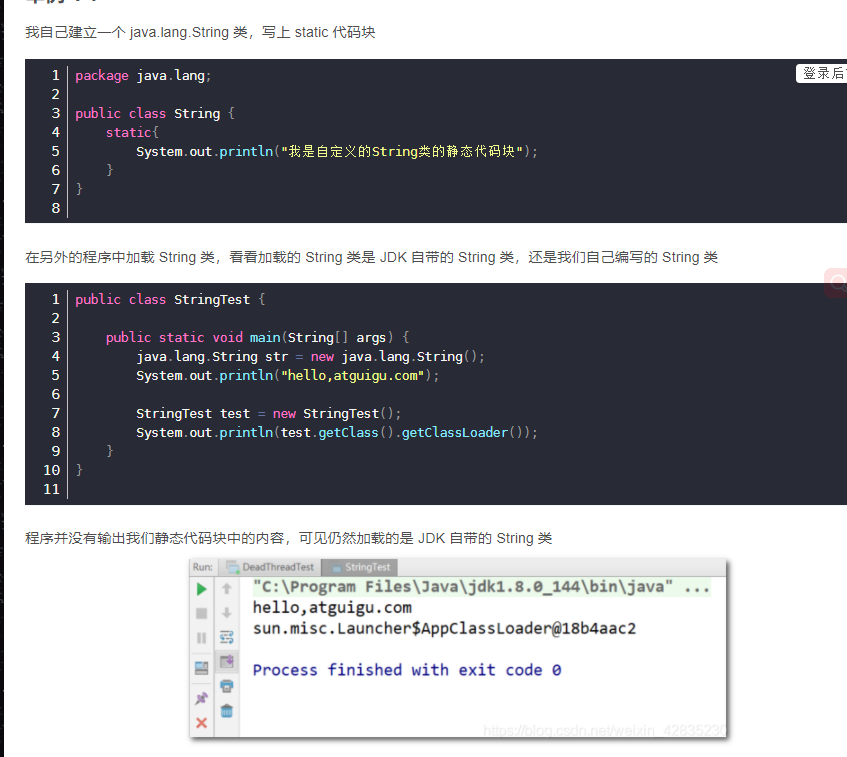
（4）父类加载器一层一层往下分配任务，如果子类加载器能加载，则加载此类，如果将加载任务分配至系统类加载器也无法加载此类，则抛出异常



为什么报错，因为他在他的上级ClassLoader找到了java.lang.String这个class

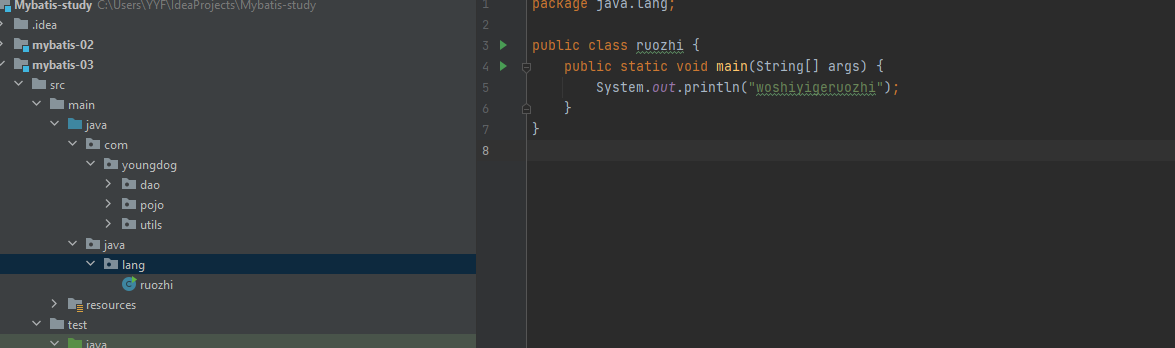
他这里第一次new对象，要加载String类,先上传一直到引导类加载器，因为他这里packet名字是java.lang，引导类加载器会处理java.lang包底下的class，因此它实际上加载的是Java本身的String类，而这个类是没有main方法的，因此无法运行

换一个例子

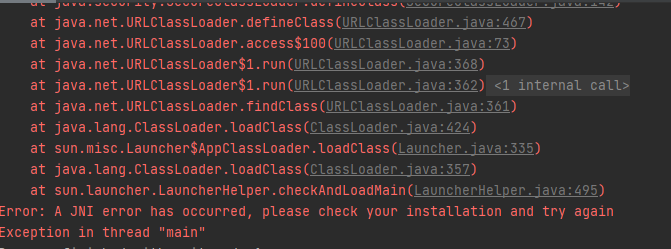


我们自己的String类用了static<

说明实际上用的是JDK自带的String类



我自己定义了一个java.lang.ruozhi



报错

为什么？

出于保护机制，java.lang 包下不允许我们自定义类

## 双亲委派机制优势

通过上面的例子，我们可以知道，双亲机制可以

* 避免类的重复加载
* 保护程序安全，防止核心API被随意篡改
  + 自定义类：java.lang.String (没用)
  + 自定义类：java.lang.ShkStart（报错：阻止创建 java.lang开头的类）

native关键字

18818791976