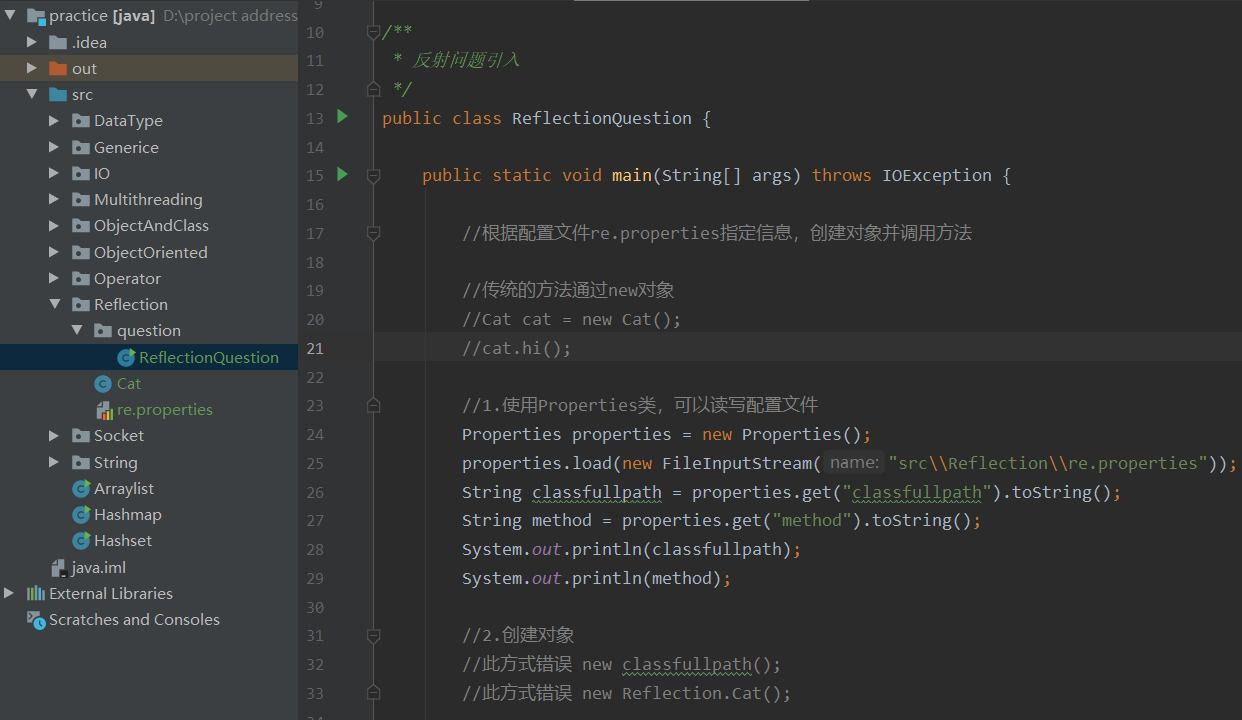
JAVA反射

1. **一个需求引出反射：**

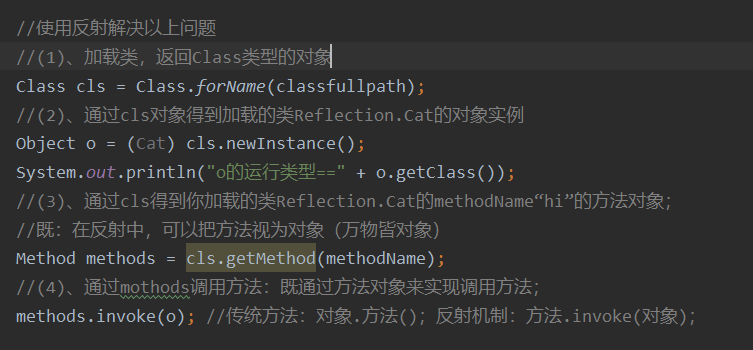
* 需求：根据配置文件“re.properties”指定信息，创建对象并调用方法

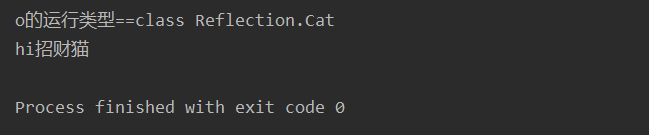


* 思考：使用现有技术，能做到吗？

以上需求在学习框架时特别多，既通过外部文件配置，在不修改源码的情况下，来控制程序，也符合设计模式的ocp原则（开闭原则）

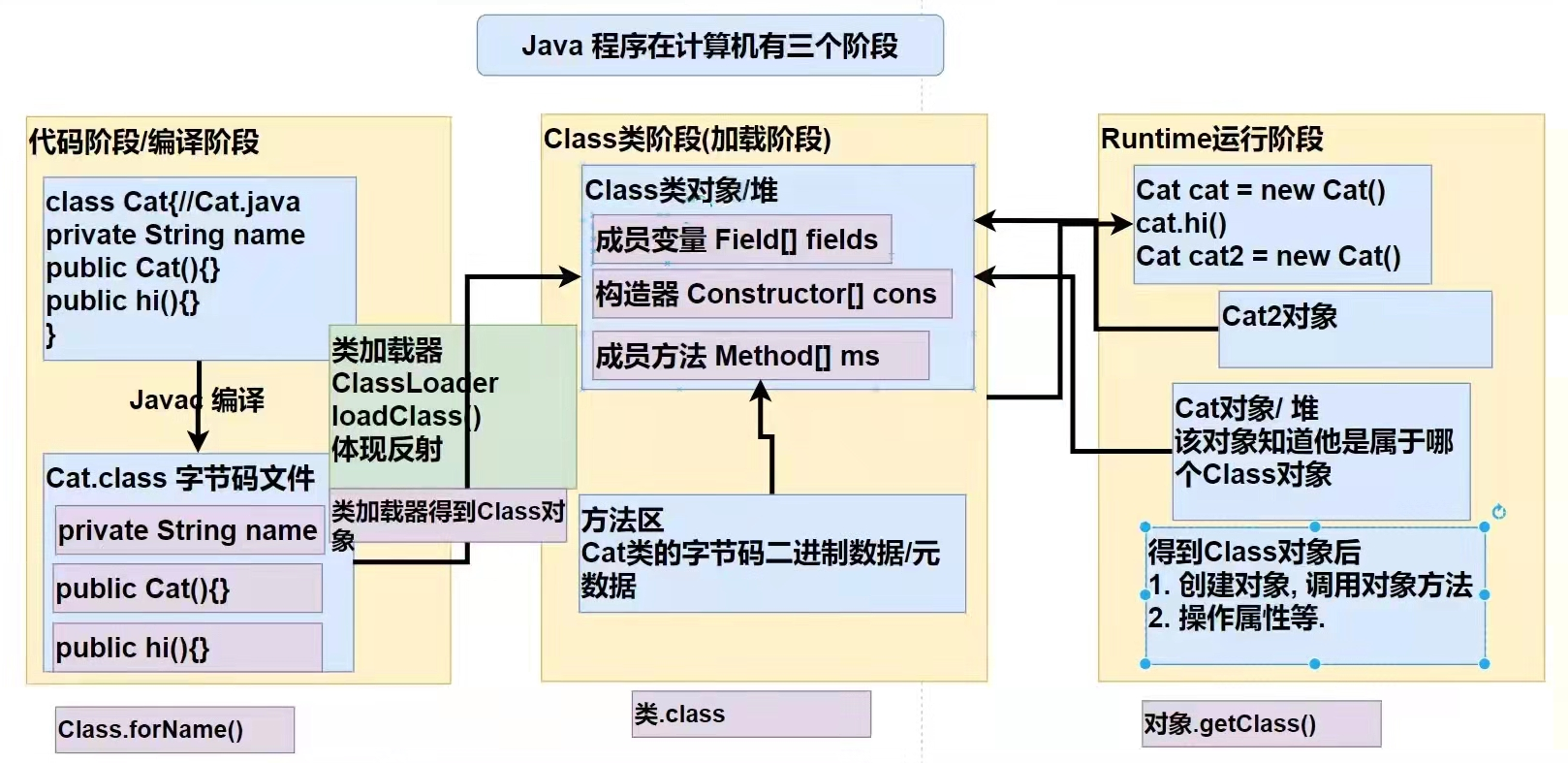
* 通过反射解决以上问题

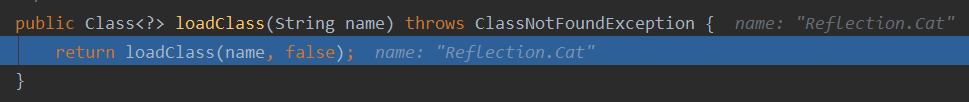




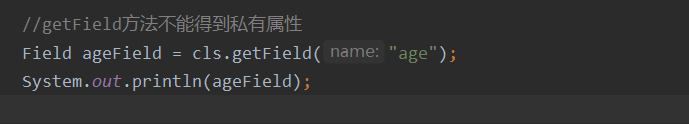
1. **反射原理**

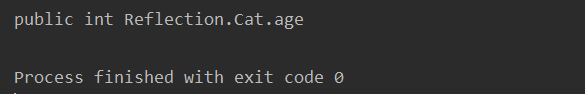
* 反射机制允许程序在执行期借助于**ReflectionAPI**取得任何类的内部信息（比如成员变量、构造器、成员方法等等）。并能操作对象的属性及方法、反射在设计模式和框架底层都会用到。
* 加载完类之后，在堆中就产生了一个**Class**类型的对象（一个类只有一个**Class**对象），这个对象包含了类的完整结构信息。通过这个对象得到类的结构、这个**Class**对象就像一面镜子，透过这个镜子看到类的结构，所以形象的称之为“反射”。





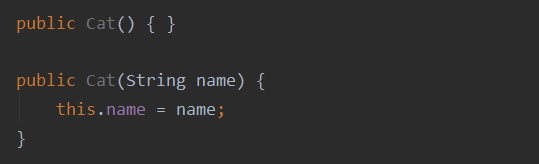
1. **反射相关的类**
2. **Java.lang.Class**：代表一个类，Class对象表示某个类的加载后在堆中的对象
3. **Java.lang.reflect.Method**：代表类的方法
4. **Java.lang.reflect.Field**：代表类的成员变量

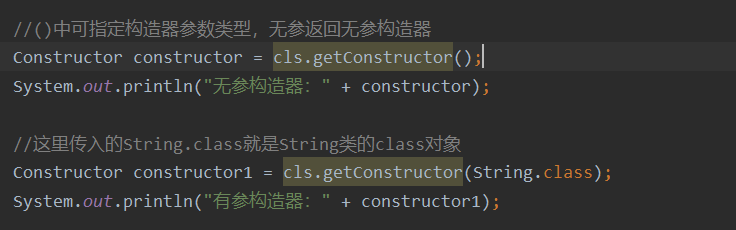


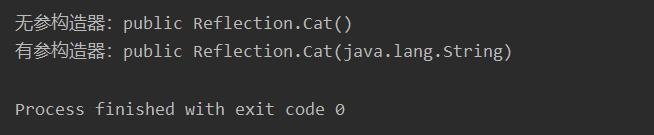




1. **Java.lang.reflect.Constructor**：代表类的构造方法

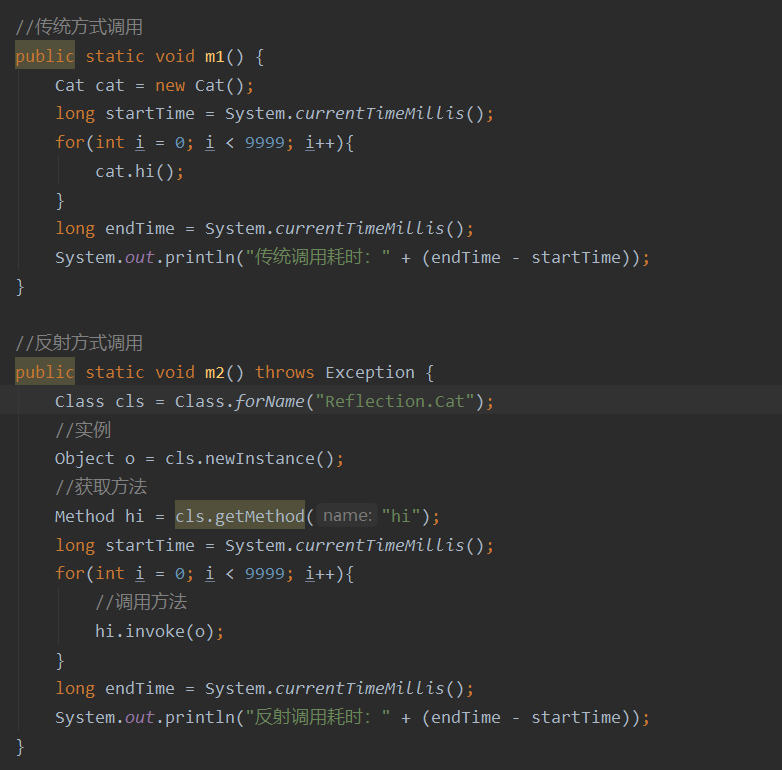
****

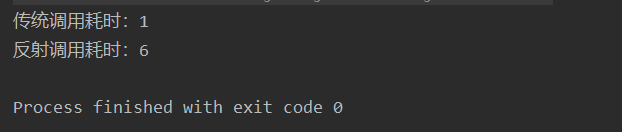
****

****

1. **反射调用优化**

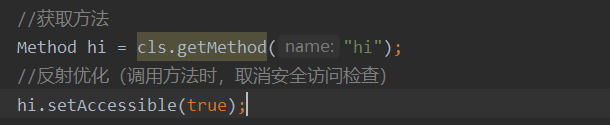
* **优点：**可以动态的创建和使用对象（也是框架底层的核心），使用灵活，没有反射机制，框架技术就失去底层的支撑。
* **缺点：**使用反射基本是解释执行，对执行速度有影响

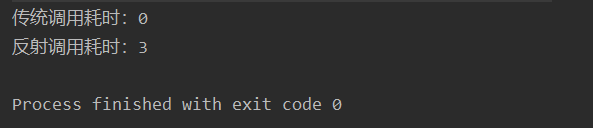
****

****

* **优化（关闭访问检查）**

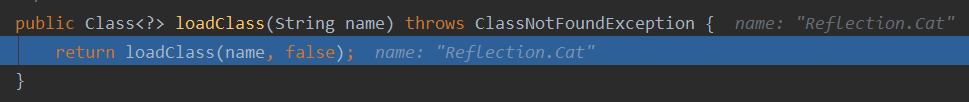
**Method、Field、Constructor**对象都有**setAccessible()**方法。此方法作用是启动和禁用访问安全检查的开关，参数值为**true**表示 反射的对象在使用时取消访问检查，提高反射效率。参数值为**false**则表示反射的对象执行访问检查

****

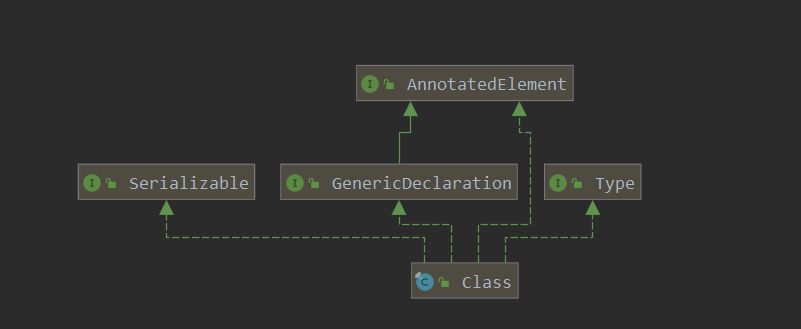
****

1. **Class类分析**

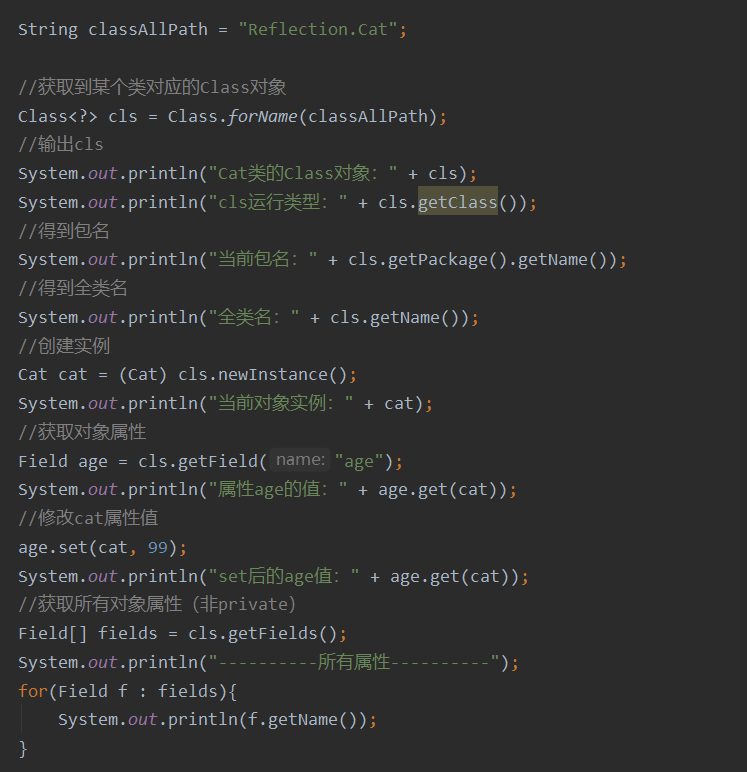
* **介绍**
* **Class也是类，因此也继承了Object类；**
* **Class类对象不是new出来的，而是系统创建的；**
* **对于某个类的Class类对象，在内存中只有一份，因为类只加载一次；**
* **每个类的实例都会记得自己是由哪个Class实例生成；**
* **通过Class对象可以完整的得到一个类的完整结构，通过一系列API；**
* **Class对象是存放在堆中的；**
* **类的字节码二进制数据，是放在方法区的，有的地方称为类的元数据（包括 方法代码、变量名、方法名、访问权限等等）；**

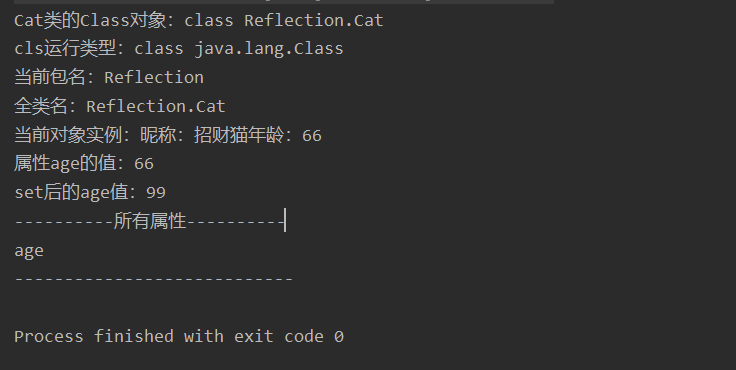
****

* **查看Class类图**

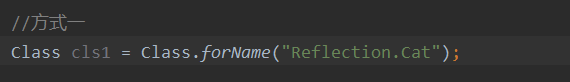
****

1. **Class类常用方法**

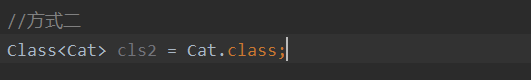
****

****

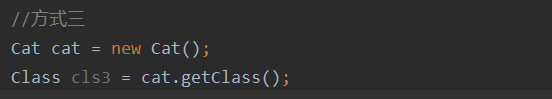
1. **获取Class对象的几种方式**
2. **Class.forName()：**多用于配置文件，读取类全路径，加载类（前提：已知一个类的全名，且该类在类路径下，可通过Class类的静态方法forName获取）；

****

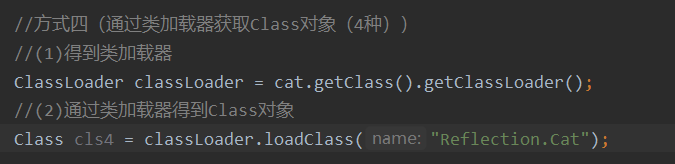
1. **Person.class：**多用于传递参数，比如通过反射得到对应的构造器（前提：若已知具体的类，通过类的class获取，该方式最为安全可靠，程序性能最高）；

****

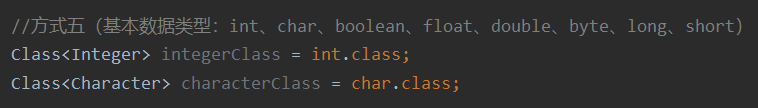
1. **对象.getClass()：**通过创建好的对象，获取Class对象；



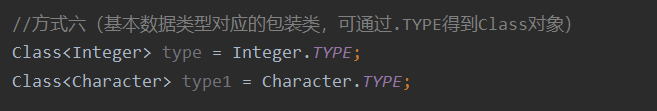
1. *通过类加载器：*

**

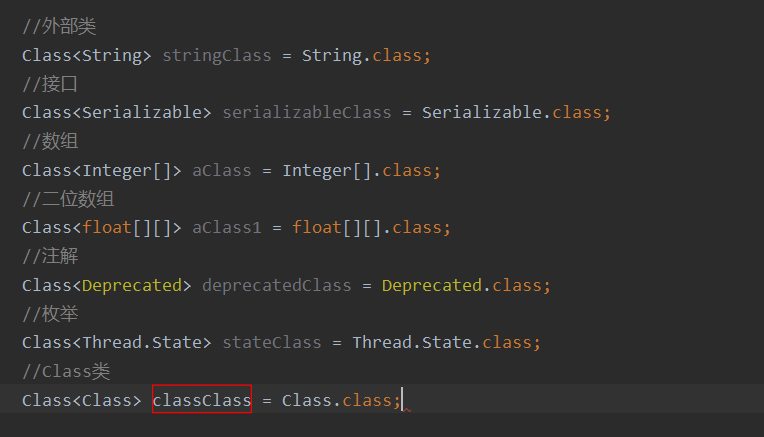
1. *基本数据类型*

**

1. *基本数据类型包装类*

**

1. **哪些类型有Class对象**

****

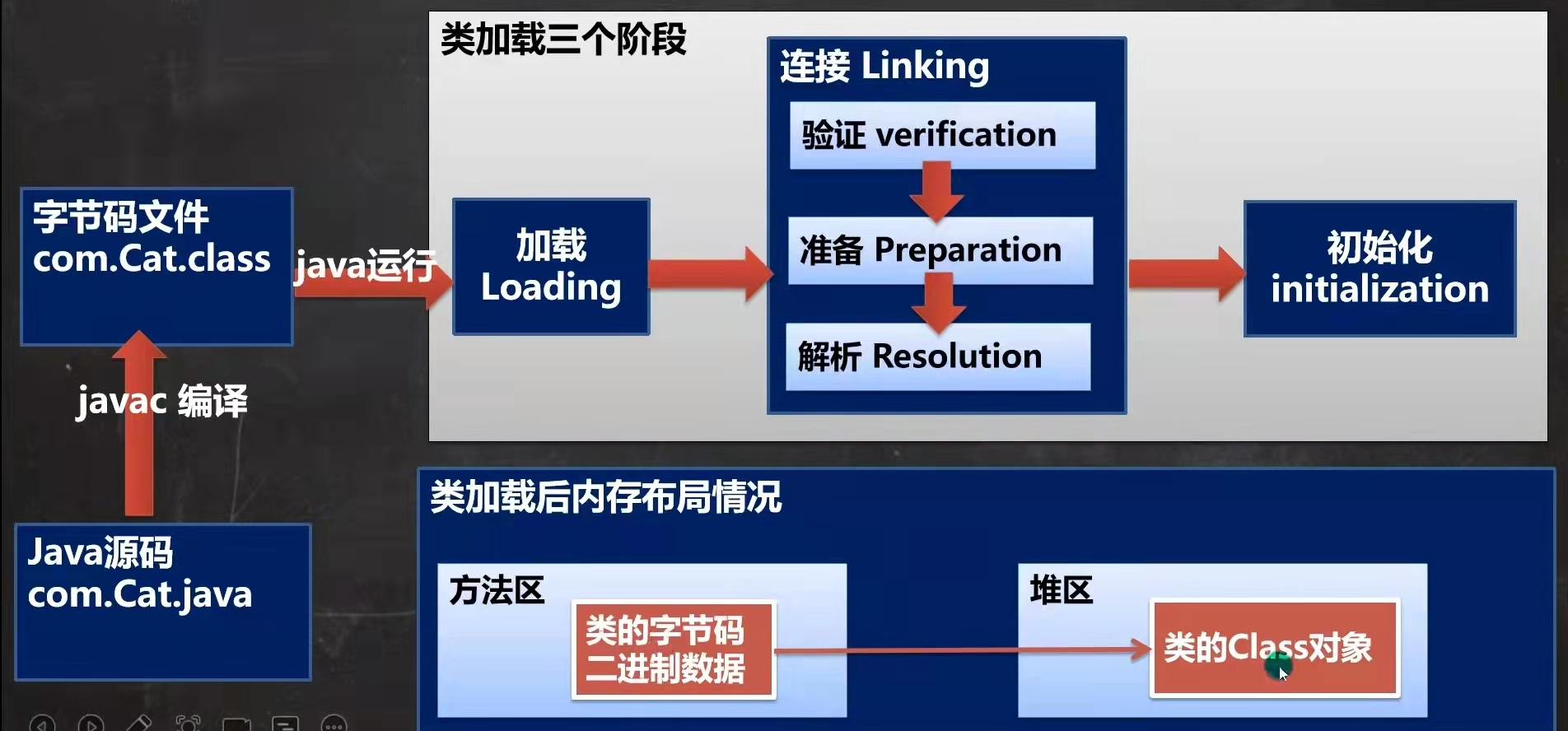
1. **动态和静态加载**

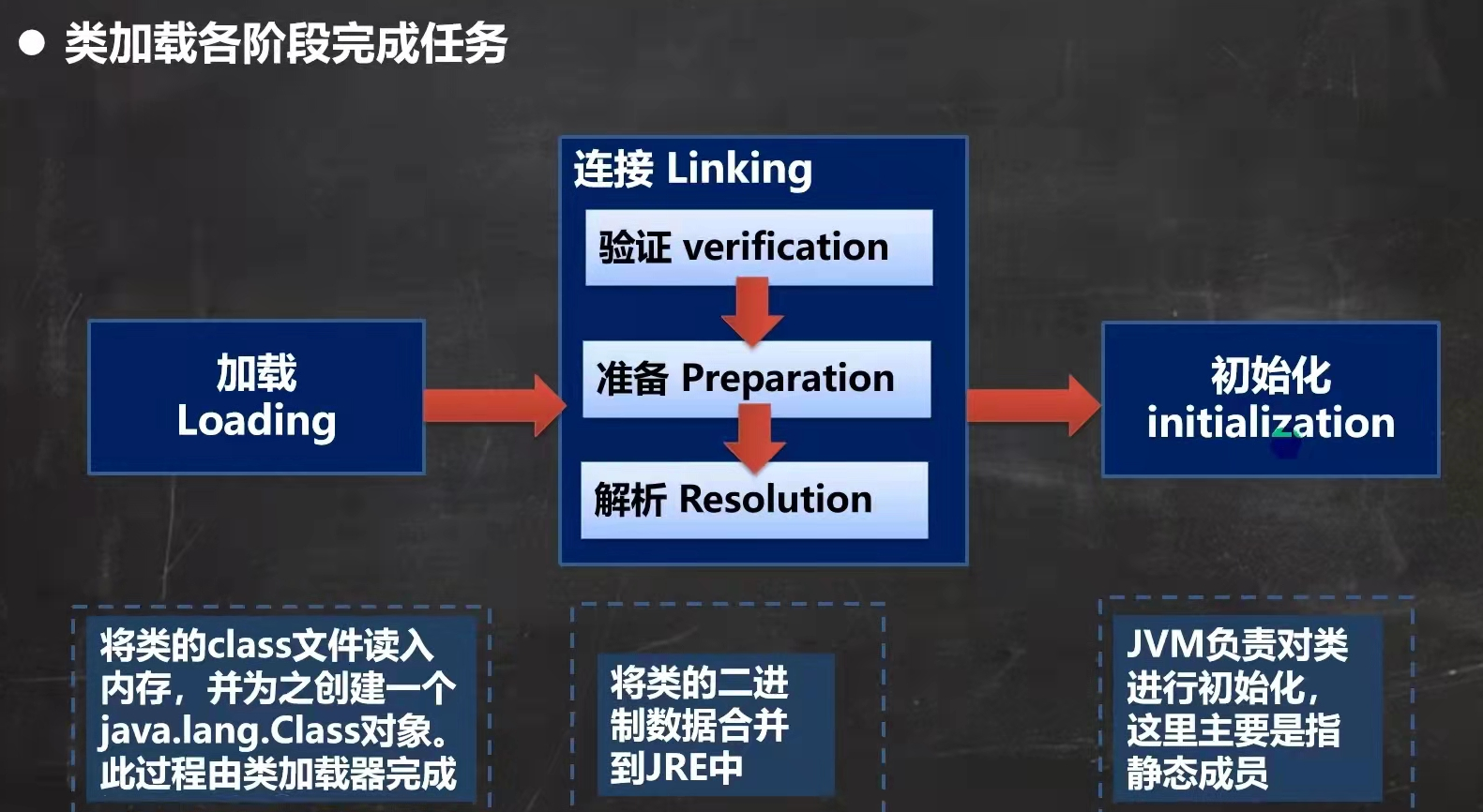
反射机制是java实现动态语言的关键，也就是通过反射实现类的动态加载

1. **静态加载：**编译时加载相关的类，如果没有则报错，依赖性太强；
2. **动态加载：**运行时加载需要的类，如果运行时用不到该类，则不报错，降低了依赖性

* *类加载时机：*
* *当创建对象时（new）*
* *当子类被加载时*
* *调用类中的静态成员时*
* *通过反射获取时*

1. **类加载过程**



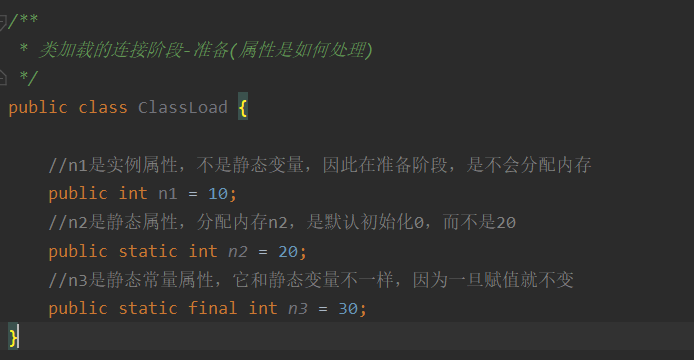


* **类加载5个阶段介绍：**
* **加载阶段：**

JVM在该阶段的主要目的是将字节码从不同的数据源（可能时class文件，也可能是jar包，甚至网络）转化为二进制字节流加载到内存中，并生成该类的java.lang.Class对象

* **连接阶段-验证：**
* 目的是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身安全。
* 包括：文件格式验证（是否以魔数oxcafebabe开头）、元数据验证、字节码验证、符号引用验证
* 可以考虑使用-Xverify：none参数关闭大部分的验证措施，缩短虚拟机类的加载时间。
* **连接阶段-准备：**

JVM会在该阶段对静态变量，分配内存并默认初始化（对应数据类型的默认初始值，如0、0L、null、 false等）。这些变量所使用的内存都将在方法区中进行分配。

****

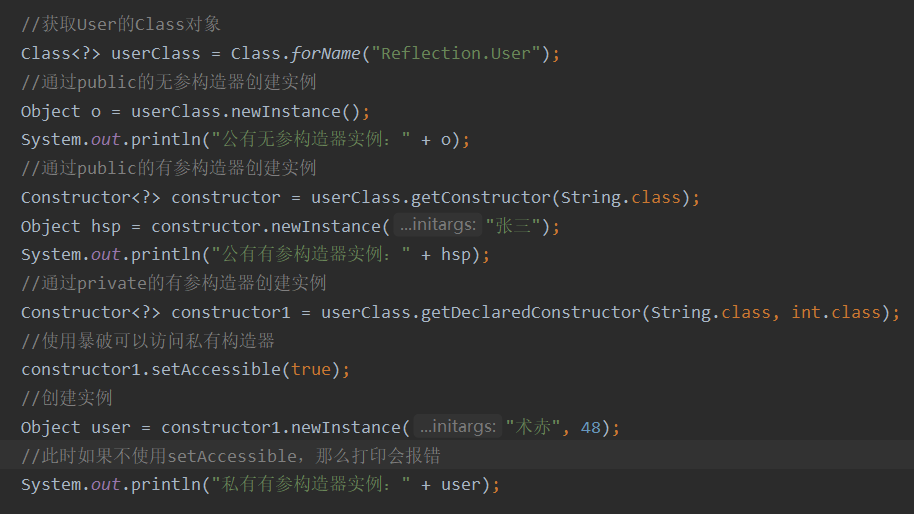
* **连接阶段-解析：**

虚拟机将常量池内的符号引用替换为直接引用的过程。

* **初始化阶段：**
* 到初始化阶段，才真正开始执行类中定义的java程序代码，此阶段是执行<clinit>()方法的过程。
* <clinit>()方法是由编译器按语句在源文件中出现的顺序，依次自动收集类中的所有静态变量的赋值动作和静态代码块中的语句，并进行合并
* 虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确的加锁、同步，如何多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的<clinit>()方法，其它线程都需要阻塞等待，直到活动线程执行<clinit>()方法完毕。

1. **获取类结构信息**
2. **反射爆破创建对象**
3. **Class类相关方法：**
   * **newInstance：**调用类中的无参数构造器，获取对应的对象；
   * **getConstructor(Class…clazz)：**根据参数列表，获取对应的构造器对象；
   * **getDecalaredConstructor(Class…clazz)：**根据参数列表，获取对应的构造器对象

1. **Constructor类相关方法**
   * + - **setAccessible：**暴破；
       - **newInstance(Object…obj)：**调用构造器

****

**十三、反射暴破操作属性**

****

**十四、反射暴破操作方法**

****