程序主动使用某个类时，就会通过加载、连接、初始化三个步骤对该类进行初始化；

> 类加载指：把类的class文件读入内存，并为之创建一个java.lang.Class对象；**系统中所有的类实际上都是java.lang.Class的实例**。 类加载器完成类的加载，类加载器由JVM提供，开发者可以通过集成ClassLoader基类来创建自己的类加载器。 用不同的类加载器，可从不同途径加载类的二进制数据：

1. 本地文件系统加载class文件；（自己写的）
2. 从jar包中加载；（下载好的，调用的jar包）
3. 对java源文件（.java文件）动态编译后加载。

> 通常会预先加载某些类。

类的连接

> 连接阶段负责把类的二进制数据合并到JRE中。 类的连接分为三个阶段：验证、准备、解析。

类的初始化

> 对声明的类变量时指定初始值，以及静态初始化块两种情况进行初始化（类初始化）； **先初始化其父类**； 判断一个类是否被初始化的简易方法

```
//加入一个静态初始化块
static {
        System.out.println("该类被静态初始化了.....");
}
```

> 以下6种情况下进行初始化类
>
> 1. 创建类的实例：new、反射、反序列化？
> 2. 调用该类的静态方法。
> 3. 调用类变量。
> 4. 使用反射方式来强制创建某个类或者看接口对应的java.lang.Class对象
> 5. **初始化其子类**。
> 6. 运行该类。 **值得注意的是：final型变量**，访问final型变量**可能不会导致类初始化**，因为如果在编译时就能**确定其值**，就会转换为初始值，相当于使用了常量；而如果不能（例如赋值时采用别的类方法，当前时间这样）就会导致该类初始化。**final类变量为枚举类也会初始化哦** **ClassLoader类的loadClass()方法**加载某个类，并不会执行该类的初始化；使用Class.forName()方法才会初始化？

类加载器

> 任务：把class文件加载到内存中，并生成对应的java.lang.Class对象。 在JVM中，唯一标识一个类是用其全限定类名（**即包名+类名**）和其**类加载器名**。 JVM在启动时，会形成由三个类加载器组成的初始类夹子阿奇层次结构,分别为：
>
> 1. 根类加载器：Bootstrap ClassLoader；
> 2. 扩展类加载器:Extension ClassLoader；
> 3. 系统类加载器:System ClassLoader； 根类加载器加载的不是ClassLoader的子类，而是JVM自身实现的核心类库，位于jdk文件的lib文件中的jar包。 扩展类加载器加载加载JRE扩展目录

> （%JAVA_HOME%/jre/lib/ext）中的jar包； ps:JAVA_HOME、Path、CLASSPATH三种概念:
>
> 1. PATH环境变量设置，一般设置是JDK文件（不是jre文件）中的bin文件，为的只是让cmd能识别到Java这个命令，执行path路径下的java.exe文件
> 2. JAVA_HOME只是为了方便设置path，一般写成JAVA_HOME = jdk文件所在,path = %JAVA_HOME%\bin;（只是起到方便修改jdk版本的作用）
> 3. CLASSPATH是查找java类的路径，执行java命令时，会自动根据这个设置的值去查找，一般设为：.;%JAVA_HOME%\lib\dt.jar;%JAVA_HOME%\lib\tools.jar。这里可以分为三个路径，每个分号一个，第一个（.）代表的是当前命令框cmd的执行路径，后面两个就是自己定义的。还有可以自定义输入的-classpath 路径参数 与 set classpath 路径参数，起同样效果。 综上所述，其实在cmd中执行 java+类名A 这个命令，是启动了java.exe文件，传入classpath和类名A作为两个参数，执行A类。 系统类加载器架子阿德就是classpath的类，即类A。 **程序可以通过ClassLoader.getSystemClassLoader()**这个方法来获取系统类加载器。 如：

```
//获取系统加载类
ClassLoader s = ClassLoader.getSystemClassLoader();
//仅仅只是加载Hellow类，不初始化
s.loadClass("Hellow");
```

类加载机制

> 全盘负责：当加载一个类时，把所有相关引用和依赖的都加载。 父类委托：先让父类加载器试图加载该class。**这里的父子关系是类加载器实例之间的，不是继承上的？** 缓存机制：保证所有加载过的Class都会被缓存。所以修改了Class后需要重启JVM。

ClassLoader类

反射

> 使用反射来创建对象的方法适用于**读取配置文件获取类名，然后创建对应的类**。其步骤为：**获取类的Class对象，利用Class对象获取构造器创建实例。 开始前，先说明：类的实例和类的Class对象是两个不同概念！**

1. 获取Class对象:

> - 使用Class.forName(sting)**方法**,传入某个类的完整类名（包含包名）；
> - 使用某个类的class**属性**，如：Person.class；
> - 调用**某个对象的**getClass()方法，是java.lang.Object类的一个方法；

2. 运用Class对象获取信息

> - 获取构造器（Constructor）、方法（Method）、成员变量（Field）、注解（Annotation）、内部类/外部类/父类/接口（Class）
> - 判断该类是否为接口、枚举、注解类型等
> - 只能保留在源代码上的注解，使用运行时获得的Class对象无法访问到。

Executable抽象基类，派生了Contructor和Method两个子类

> 提供获取方法的形参个数或者形参名的方法：int getParameterCount();Parameter getParamters();

程序可以通过**Contructor**对象来创建实例，通过**Method**对象来执行对应的方法，通过**Field**对象直接访问并修改对象的成员变量值。

3. 创建实例：

> - 使用Class对象的newInstance()方法,要求该类用默认的构造器，使用默认构造器来创建实例。

```
//获取一个Controller类的Class对象
Class<?> clazz = Class.forName("com.dwh.Controller");
//获取Controller类的实例
Object object = clazz.newInstance();
```

> - 使用Constructor对象的newInstance()方法来创建实例，先用**Class**对象获取一个**Constructor**对象，在用**Constructor**对象的**newInstance()**方法，能使用指定的构造器来创建实例。

```
//获取一个Controller类的Class对象
Class<?> clazz = Class.forName("com.dwh.Controller");
//获取了类的带一个String类型参数的构造器
Constructor ctor = clazz.getConstructor(String.class);
Object object = ctor.newInstance("测试窗口");
```

使用泛型可以避免类型转换，如：

```
/*
*第一种，不使用泛型
*/
//返回值类型为Object
public static Object getInstance(String className){
    class cls = class.forName(className);
    return cls.newInstance();
}
//调用，需要类型转换
Date d = (Date)xxx.getInstance("java.util.Date");

/*
*第二种，使用泛型
*/
//返回值类型为T对象
public static <T> T getInstance(String className){
    class cls = class.forName(className);
    return cls.newInstance();
}
//调用，不需要类型转换
Date d = xxx.getInstance("java.util.Date");
```

4. 使用Method对象调用方法

> - 第一步：使用3中的方法创建实例后，使用**Map<String,Object>来放置实例**，key为实例名（变量名）即等价于：Object a = new Object()；Map为（a,Object）；
> - 第二步：通过变量名获取实例后，再通过方法名获取实例中的某一个Method对象（即某个方法，例如Setter）；
> - 第三步：调用该Method对象的Object invoke(Object obj,.....args)方法执行该Method对象对应的方法(Setter)。其中Obj是执行方法的实例（即上面的a对应的Object），args是对应方法(Setter)的参数。 同样会有权限问题，**比如执行一个private修饰的方法**，这时要用setAccessible(true)方法取消访问权限检查。

5. 访问成员变量

> 使用Field类，有两个方法:getXXX(Object obj)和setXXX(Object obj,val)来获取和设置成员变量，XXX是8种基础类型；如果成员变量是引用类型，则使用get()和set()即可。

```
public class TestField {
    public static void main(String[] args) throws NoSuchFieldException,
IllegalAccessException {
        //创建一个实例
        Person p = new Person();
        //获取类的Class对象
        Class<Person> clazz = Person.class;
        //获取a这个成员变量，getDeclaredField无视访问控制符
        Field testa = clazz.getDeclaredField("a");
        //解除访问控制
        testa.setAccessible(true);
        System.out.println(testa.getInt(p));
        Field testt = clazz.getDeclaredField("t");
        System.out.println(testt.get(p));
    }
}

class Person{
    private int a;
    Test t;
    Person(){
        t = new Test();
    }
}

class Test{
    private int b;
    @Override
    public String toString(){
        return "hhh";
    }
}
```

> clazz.getField()和clazz.getDeclaredField()两个方法，前者只能获取**public修饰的**，后者无视访问控制。

利用反射获取成员变量的泛型信息

> 获取普通类型的方法：class<?> a = f.getType;f为Field对象 对于泛型类型成员变量：
>
> - Type是一个类，代表数据类型
> - 先用getGenericType()转换为Type，再强制转成ParameterizedType，从而使用**getRawType()**和**getActualTypeArguments()**两个方法

```
//f为某一个泛型类型，具体为map<String,Integer>
Type gType = f.getGenericType();
//因为后面需要强制转换成ParameterizedType类型所以先做判断
if(gType instanceof ParameterizedType){
    ParameterizedType p = (ParameterizedType)gType;
    //可以直接print，返回的是原始Map类型
    Type t = p.getRawType();
    //Type[] t2 = p.getActualTypeArguments();
    for(int i = 0; i < t2.lenth; i++){
        System.out.println("第" + i + "个泛型类型是："+t2[i]);
    }
    //输出String和Integer
}
else{
    ;
}
```

利用反射生成动态代理

> java.lang.reflect包下有Proxy类和InvocationHandler接口，主要是利用这两个来生成动态代理对象。
> Proxy类是所有动态代理类的父类，通过它的两个静态方法来创建动态代理类和动态代理类对象。
>
> - 方法一：Class getProxyClass(ClassLoader loader,Class<?>...interfaces):创建一个动态代理类的**Class**对象。第一个参数为：类加载器；第二个参数为：实现的多个接口。
> - 方法二：Object Proxy.newProxyInstance(ClassLoader loader,Class<?>...interfaces,InvocationHandler h):创建动态代理类对象，第一二参数同上，第三个参数实际传入自定义的实现**InvocationHandler**接口的类对象。 代理类对象执行接口interfaces的任何方法实际上都是执行**InvocationHandler**对象的**invoke**方法，所以自定义的重点在于重写**invoke**方法。 因为调用第一个方法后，也需要利用**Class**创建实例，所以一般直接使用第二个方法，实际上第二个方法的实现也调用了第一个方法，所以才称之为利用反射生成的动态代理。下面代码中获取参数加载类，使用的是**Person.class.getClassLoader()**，接口类的构造器。

```
//使用第一种方法
public class TestProxy {
    public static void main(String[] args){
        InvocationHandler Handler = new MyInvocationHandler();
        //获取代理类的Class对象，注意第二个参数
        Class<?> clazz = Proxy.getProxyClass(Person.class.getClassLoader(),new
Class[]{Person.class});
        try {
            //获取构造器
```

```
            Constructor sot = clazz.getConstructor(new Class[]
{InvocationHandler.class});
            //创建对象，注意要转类型
            Person s = (Person)sot.newInstance(new Object[]{Handler});
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
        /*
        Person s =
(Person)Proxy.newProxyInstance(Person.class.getClassLoader(),new Class[]
{Person.class},Handler);
        //同样可以写成上面的形式
        */
    }
}

//设计一个接口
interface Person{
    void work();
    void say();
}

class MyInvocationHandler implements InvocationHandler{
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        return null;
    }
}
```

上面代码中，**生成的代理实例对象调用say或者work方法，都是执行invoke方法**。

上述只是单纯的说明了代理类会使抽象类的实例都实际执行invoke方法，实际上要运用其invoke方法的参数

> invoke参数列表
>
> - proxy:是指代理对象本身，实际上是一个**Proxy的子类的实例，实现了我们输入的接口**（第一句话）；
> - method:代表正在执行的方法。如：s.say()，这时候method代表的是say()方法。
> - args:代表调用目标方法时传入的参数。 主要是使用method这个参数来区分不同方法。

> 动态代理 动态代理主要是为了解决当有一段代码

```
public class TestProxy_2 {
    public static void main(String[] args){
```

```
        //被代理的实体类
        Dog target = new GunDog();
        //利用工厂生产代理类
        Dog dog = (Dog) ProxyFactory.getProxy(target);
        dog.info();
        dog.run();
    }
}

//接口
interface Dog{
    void info();
    void run();
}

//实体类
class GunDog implements Dog{

    @Override
    public void info() {
        System.out.println("猎狗");
    }

    @Override
    public void run() {
        System.out.println("极速奔跑");
    }
}

//增强的功能模块代码
class DogUtil{
    public void method1(){
        System.out.println("---------第一个增强功能--------------");
    }
    public void method2(){
        System.out.println("---------第二个增强功能--------------");
    }
}

class MyInvocationHandler2 implements InvocationHandler {
    //需要被代理（增强）的对象
    //用来作为Invoke方法中调用方法method的对象
    private Object target;

    public void setTarget(Object target) {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        //创建增强类
        DogUtil d = new DogUtil();
        d.method1();
```

```
        //被代理对象target执行其方法
        Object result = method.invoke(target,args);
        d.method2();
        return result;
    }

}

//创建代理类工厂
class ProxyFactory{
    public static Object getProxy(Object target){
        MyInvocationHandler2 handler = new MyInvocationHandler2();
        handler.setTarget(target);
        //注意第一个参数和第二个参数
        //由于target传入是一个Object所以和前面的不一样
        //第一个加载器参数：先使用getClass()来获取Class对象。
        //第二个加载器参数：使用getInterfaces()来获取对象实现的接口组。
        return Proxy.newProxyInstance(target.getClass().getClassLoader(),
                target.getClass().getInterfaces(),handler);
    }
}
```

上述代码实际上是为了给实体类**GunDog**增加某些功能（**DogUtil**），为了解耦，即把增强功能的代码从实体类中提取出来。 作用：1.解耦，方便维护。2.增强类功能（**DogUtil**）可作为公共代码，给其他类使用。反过来说如果存在多个类使用一段公共代码，则可以考虑使用代理模式提取公共代码。