反 射

一、Java核心概念

> 本章节通过一些核心概念讲解，引出反射技术！
>
> 具体涉及虚拟机部分内容，Java高级课程会继续讲解！
>
> 参考内容：Java虚拟机规范https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-1.html

1.1 Java虚拟机

Java 虚拟机是 Java 平台的基石。它是该技术的组成部分，负责其硬件和操作系统独立性、编译代码的小尺寸以及保护用户免受恶意程序侵害的能力。

Java 虚拟机是一种抽象计算机。像真正的计算机一样，它具有指令集并在运行时操纵各种内存区域。关于内存区域，后续讲解！！使用虚拟机实现编程语言是相当普遍的；最著名的虚拟机可能是 UCSD Pascal 的 P-Code 机器。

Java 虚拟机不采用任何特定的实现技术、主机硬件或主机操作系统。它本身不是解释的，但也可以通过将其指令集编译为CPU 的指令集来实现CPU调用。

Java 虚拟机对 Java 编程语言一无所知，只知道一种特定的二进制格式，即`class`文件格式。文件`class`包含 Java 虚拟机指令（或字节码）和符号表，以及其他辅助信息。

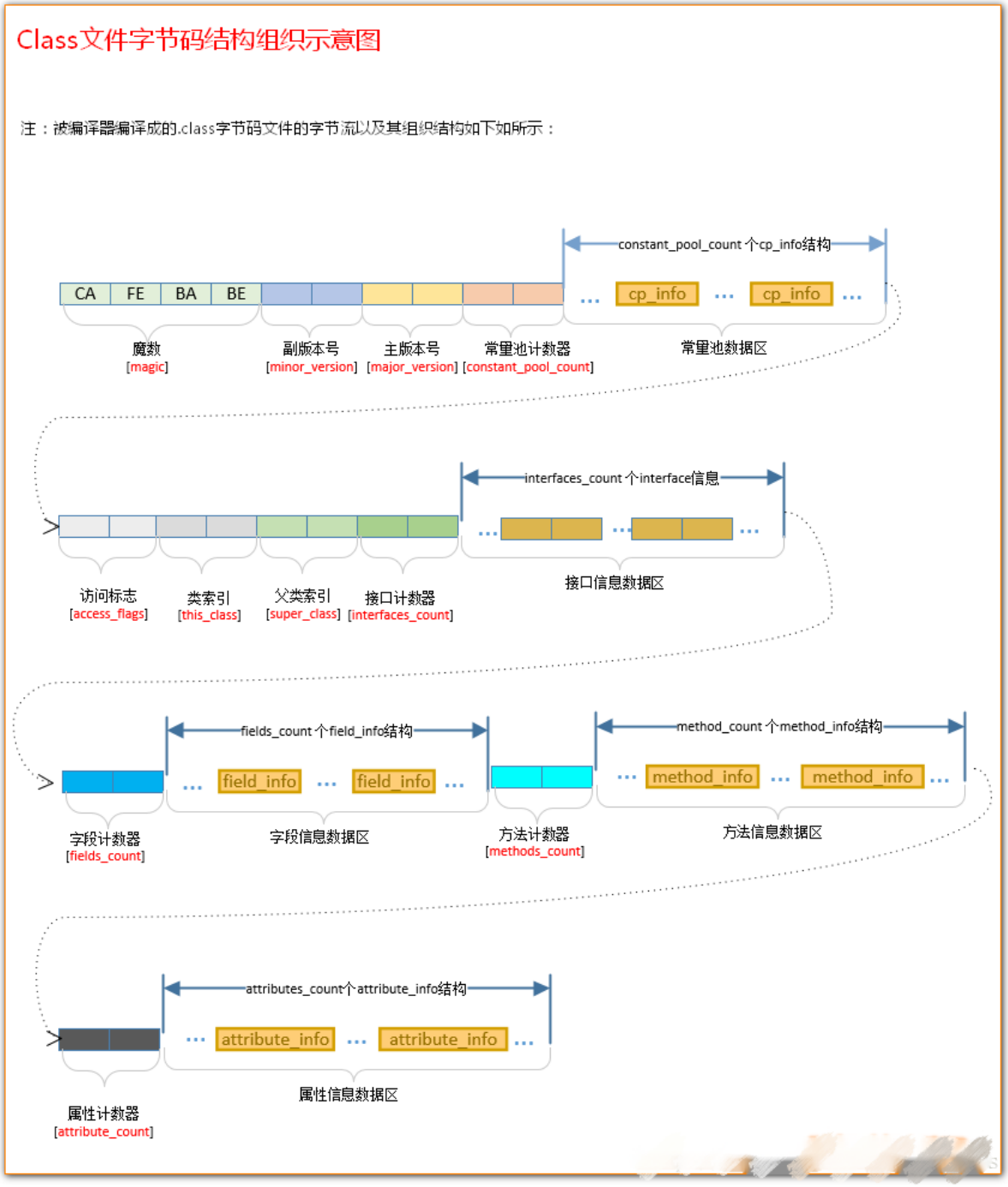
为了安全起见，Java 虚拟机对 `class` 文件中的代码施加了强大的语法和结构约束。但是，任何具有可以用有效 `class` 文件【本地，网络地址】表示的功能的语言都可以由 Java 虚拟机托管。

1.2 Class文件

由 Java 虚拟机执行的编译代码使用独立于硬件和操作系统的二进制格式表示，通常（但不一定）存储在文件中，称为 `class` 文件格式。文件格式精确地定义了类或接口的 `class` 表示，包括诸如字节顺序之类的细节等..

> 虚拟机和class的关系

1.3 Class File 结构【了解】



1. 魔数

> 所有的由Java编译器编译而成的class文件的前4个字节都是“0xCAFEBABE”
> 它的作用在于：当JVM在尝试加载某个文件到内存中来的时候，会首先判断此class文件有没有JVM认为可以接受的“签名”，即JVM会首先读取文件的前4个字节，判断该4个字节是否是“0xCAFEBABE”，如果是，则JVM会认为可以将此文件当作class文件来加载并使用。
>
> 天王盖地虎，宝塔镇河妖！

2. 版本号(minor_version,major_version)

> 主版本号和次版本号在class文件中各占两个字节，每个JDK的主版本都是固定的！虚拟机主要对比主版本！JDK1.0的主版本号为45，以后的每个新主版本都会在原先版本的基础上加1。若现在使用的是JDK1.7编译出来的class文件，则相应的主版本号应该是51,对应的7，8个字节的十六进制的值应该是 0x33。
>
> 一个 JVM实例只能支持特定范围内的主版本号，这个 Class 文件才可以被此 Java 虚拟机支持 。

> JVM在加载class文件的时候，会读取出主版本号，然后比较这个class文件的主版本号和JVM本身的版本号，如果JVM本身的版本号 < class文件的版本号，JVM会认为加载不了这个class文件，会抛出我们经常见到的"java.lang.UnsupportedClassVersionError: Bad version number in .class file " **Error 错误**；反之，JVM会认为可以加载此class文件，继续加载此class文件。

3. 常量池计数器

> 常量池是class文件中非常重要的结构，它描述着整个class文件的字面量信息。 常量池是由一组constant_pool结构体数组组成的，而数组的大小则由常量池计数器指定。常量池计数器!

4. 常量池数据区

> 常量池，constant_pool是一种表结构,它包含 Class 文件结构及其子结构中引用的所有字符串常量、 类或接口名、字段名和其它常量

5. 访问标志

> 访问标志，access_flags 是一种掩码标志，用于表示某个类或者接口的访问权限及基础属性。

标记名	值	含义
ACC_PUBLIC	0x0001	可以被包的类外访问。
ACC_FINAL	0x0010	不允许有子类。
ACC_SUPER	0x0020	当用到 invokespecial 指令时，需要特殊处理的父类方法。
ACC_INTERFACE	0x0200	标识定义的是接口而不是类。
ACC_ABSTRACT	0x0400	不能被实例化。
ACC_SYNTHETIC	0x1000	标识并非 Java 源码生成的代码。
ACC_ANNOTATION	0x2000	标识注解类型
ACC_ENUM	0x4000	标识枚举类型

注：

- 带有 ACC_SYNTHETIC 标志的类，意味着它是由编译器自己产生的而不是由程序员编写的源代码生成的。
- 带有 ACC_ENUM 标志的类，意味着它或它的父类被声明为枚举类型。
- 带有 ACC_INTERFACE 标志的类，意味着它是接口而不是类，反之是类而不是接口。如果一个 Class 文件被设置了 ACC_INTERFACE 标志，那么同时也得设置 ACC_ABSTRACT 标志。同时它不能再设置 ACC_FINAL、ACC_SUPER 和 ACC_ENUM 标志。
- 注解类型必定带有 ACC_ANNOTATION 标记，如果设置了 ANNOTATION 标记，ACC_INTERFACE 也必须被同时设置。如果没有同时设置 ACC_INTERFACE 标记，那么这个 Class 文件可以具有表中的除 ACC_ANNOTATION 外的所有其它标记。当然 ACC_FINAL 和 ACC_ABSTRACT 这类互斥的标记除外。
- ACC_SUPER 标志用于确定该 Class 文件里面的 invokespecial 指令使用的是哪一种执行语义。目前 Java 虚拟机的编译器都应当设置这个标志。ACC_SUPER 标记是为了向后兼容旧编译器编译的 Class 文件而存在的，在 JDK1.0.2 版本以前的编译器产生的 Class 文件中，access_flag 里面没有 ACC_SUPER 标志。同时，JDK1.0.2 前的 Java 虚拟机遇到 ACC_SUPER 标记会自动忽略它。
- 在表中没有使用的 access_flags 标志位是为未来扩充而预留的 这些预留的标志为在编译器中会被设置为 0，Java 虚拟机实现也会自动忽略它们.

6. 类索引

> 类索引，表示这个 Class 文件所定义的类或接口。

7. 父类索引

> 父类索引，表示这个 Class 文件所定义的类的直接父类。当前类的直接父类，以及它所有间接父类的access_flag 中都不能带有ACC_FINAL标记。

> 如果 Class 文件的 super_class的值为 0【没有父类】，那这个Class文件只可能是定义的是java.lang.Object类，只有它是唯一没有父类的类。

8. 接口计数器

> 接口计数器，interfaces_count的值表示当前类或接口的直接饮用接口数量。

9. 接口信息数据区

> 接口表【interfaces[]数组】，在interfaces[]数组中，成员所表示的接口顺序和对应的源代码中给定的接口顺序（从左至右）一样，即interfaces[0]对应的是源代码中最左边的接口。

10. 字段计数器(fields_count)

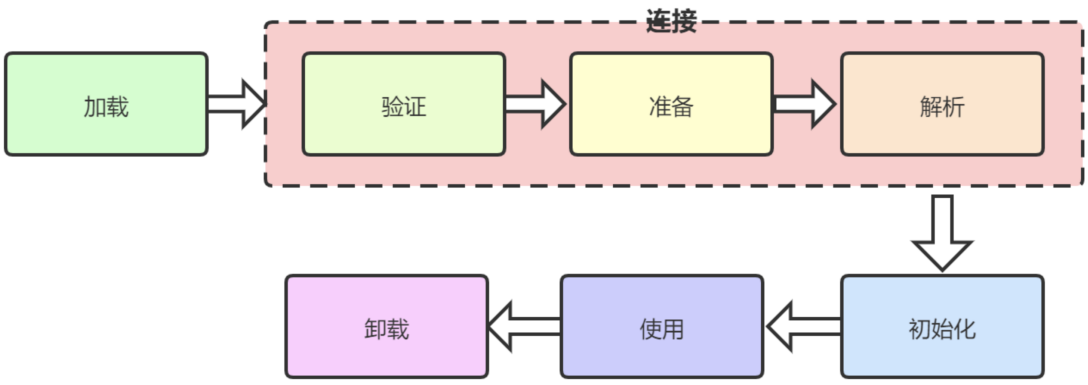
> 字段计数器，`fields_count`的值表示当前 `Class` 文件 `fields[]`数组的成员个数。`fields[]`数组中每一项都是一个`field_info`结构的数据项，它用于表示该类或接口声明的类字段或者实例字段。

11. 字段信息数据区(fields[fields_count]

> 字段表，fields[]数组中的每个成员都必须是一个fields_info结构的数据项，用于表示当前类或接口中某个字段的完整描述。 fields[]数组描述当前类或接口声明的所有字段，但不包括从父类或父接口继承的部分。

1.4 类加载过程

> 类加载过程就是class文件加载到虚拟机的内存过程！
>
> Class对象 == Class文件



1. 加载： 将Java的字节码文件从不同的源读取到JVM中，并映射为JVM认可的类数据结构[Class对象]！

   注意注意注意：Class对象全局唯一！它是实例化具体对象的模板，也是反射技术的起源对象！

   例如： Student.class - 1:1 -> StudentClass对象 -1:n-》 Student对象

2. 链接：将原始的类定义信息平滑的转入到JVM的过程，有三个子步骤：
   1. 验证：虚拟机的安全保障，JVM需要验证字节信息格式是否符合Java虚拟机规范，防止恶意信息和不符合规则的信息危害JVM运行！
   2. 准备：创建类或者接口中的静态变量，并初始化静态变量值，注意，这初始化并不是真正的赋于Java中声明的值，重点是开辟空间，赋类型变量的默认值，例如：boolean=false!

    3. 解析：这一步会讲常量池的引用符号替换直接引用，例如，我得父类或者接口，从符号变成具体地址指向！
3. 初始化：这一步真正的执行类的初始化的代码逻辑，包括静态字段赋值的动作，以及执行类定义中的静态代码块等，注意父类的初始化逻辑高于当前类！

> 注意：每个类加载到内存都会生成一个唯一的类对象。

1.5 触发类加载的时机

1. 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；
2. 当遇到调用静态方法的指令时，初始化该静态方法所在的类；
3. 当遇到访问静态字段的指令时，初始化该静态字段所在的类；
4. 子类的初始化会触发父类的初始化；
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，会触发该接口的初始化；
6. 使用反射 API 对某个类进行反射调用时，初始化这个类；
7. 当虚拟机启动时，初始化用户指定的主类；

1.6 Java中实例化对象

> Java中想使用类的非静态方法和属性，需要将类进行对象实例化，
>
> 利用具体对象进行调用！接下来介绍下Java实例化对象以及对象属性和方法操作的常见方式！

1. new关键字方式
    1. 在编译期确定实例类型和方法属性调用位置
    2. 程序运行时无法动态修改
    3. new实例化的动作底层也依赖当前Class模板
    4. 程序的主要对象实例化方式
2. Reflect[反射]机制方式
    1. 反射是一种动态机制，运行时可以根据Class模板完成类的实例化

       以及属性和方法的操作
    2. 反射还可以通过方法打破修饰符的限制。例如：类外访问私有方法
    3. 第三方工具和框架都大量使用反射，保证框架的灵活和可扩展性，他们往往通过配置，可以动态的实例不同的类，例如：数据库框架MyBatis支持动态查询，结果是Map还是Java类都可以动态指定，原理就是反射！反射号称第三方神器
    4. 反射的操作也是基于Class模板对象这点和new一样
    5. 反射的学习就是学习如何动态常见对象，操作方法和属性
    6. 反射也是存在缺点，就是速度较慢，甚至甲骨文关于反射教学也提出反射性能开销大的缺点！
    7. 瑕不掩瑜，退役的小井老师携带1亿美金要嫁给你，你会不会在意她的过往呢？

二、Class模版对象

> class模板对象虚拟机唯一，反射和new对象的模板对象！
>
> 反射必须先要获取模板对象！Class对象是反射的根源。

2.1 获取Class模板方法

1. 类型名.class

   要求编译期间已知类型，所有的Java类型都可通过此方式获取Class对象

2. 对象.getClass()

   获取对象的运行时类型

3. Class.forName(String 类全限定符)

   可以获取编译期间未知的类型！

   更推荐这种，动态传入字符串，更加灵活！

4. ClassLoader的类加载器对象.loadClass(String 类型全名称)

   可以用系统类加载对象或自定义加载器对象加载指定路径下的类型

2.2 方法实践练习

```
public class TestClass {
    @Test
    public void test05() throws ClassNotFoundException{
        Class c = TestClass.class;
        ClassLoader loader = c.getClassLoader();

        Class c2 = loader.loadClass("com.atguigu.test05.Employee");
        Class c3 = Employee.class;
        System.out.println(c2 == c3);
    }

    @Test
    public void test03() throws ClassNotFoundException{
        Class c2 = String.class;
        Class c1 = "".getClass();
        Class c3 = Class.forName("java.lang.String");

        System.out.println(c1 == c2);
        System.out.println(c1 == c3);
    }
}
```

三、反射通用操作【重点】

3.1 常见方法

方法名	描述
public String getName()	获取类的完全名称
public Package getPackage()	获取包信息
public Class<? super T> getSuperclass()	获取父类
public Class<?>[] getInterfaces()	获取实现父接口
public Field[] getFields()	获取字段信息
public Method[] getMethods()	获取方法信息
public Constructor<?>[] getConstructors()	获取构造方法
public T newInstance()	反射创建对象
public int getModifiers();	获取修饰符

3.2 获取类型的信息

> 可以获取：包、修饰符、类型名、父类、父接口、成员（属性、构造器、方法）、注解（类上的、方法上的、属性上的）【注解完事再讲】

```
public class TestClassInfo {
    public static void main(String[] args) throws NoSuchFieldException,
SecurityException {
        //1、先得到某个类型的Class对象
        Class clazz = String.class;
        //比喻clazz好比是镜子中的影子

        //2、获取类信息
        //（1）获取包对象，即所有java的包，都是Package的对象
        Package pkg = clazz.getPackage();
        System.out.println("包名：" + pkg.getName());

        //（2）获取修饰符
        //其实修饰符是Modifier，里面有很多常量值
        /*
         * 0x是十六进制
         * PUBLIC           = 0x00000001;  1    1
         * PRIVATE          = 0x00000002;  2    10
         * PROTECTED        = 0x00000004;  4    100
         * STATIC           = 0x00000008;  8    1000
         * FINAL            = 0x00000010;  16   10000
         * ...
         *
         * 设计的理念，就是用二进制的某一位是1，来代表一种修饰符，整个二进制中只有一位是1，其
余都是0
         *
         * mod = 17          0x00000011
         * if ((mod & PUBLIC) != 0)  说明修饰符中有public
         * if ((mod & FINAL) != 0)   说明修饰符中有final
         */
        int mod = clazz.getModifiers();
        System.out.println(Modifier.toString(mod));
```

```
        //（3）类型名
        String name = clazz.getName();
        System.out.println(name);

        //（4）父类，父类也有父类对应的Class对象
        Class superclass = clazz.getSuperclass();
        System.out.println(superclass);

        //（5）父接口们
        Class[] interfaces = clazz.getInterfaces();
        for (Class class1 : interfaces) {
            System.out.println(class1);
        }

        //（6）类的属性，   你声明的一个属性，它是Field的对象
/*      Field clazz.getField(name)  根据属性名获取一个属性对象，但是只能得到公共的
        Field[] clazz.getFields();  获取所有公共的属性
        Field clazz.getDeclaredField(name)  根据属性名获取一个属性对象，可以获取已声明的
        Field[] clazz.getDeclaredFields()   获取所有已声明的属性
        */
        Field valueField = clazz.getDeclaredField("value");
//      System.out.println("valueField = " +valueField);

        Field[] declaredFields = clazz.getDeclaredFields();
        for (Field field : declaredFields) {
            //修饰符、数据类型、属性名
            int modifiers = field.getModifiers();
            System.out.println("属性的修饰符：" + Modifier.toString(modifiers));

            String name2 = field.getName();
            System.out.println("属性名：" + name2);

            Class<?> type = field.getType();
            System.out.println("属性的数据类型：" + type);
        }
        System.out.println("--------------------------");
        //（7）构造器们
        Constructor[] constructors = clazz.getDeclaredConstructors();
        for (Constructor constructor : constructors) {
            //修饰符、构造器名称、构造器形参列表  、抛出异常列表
            int modifiers = constructor.getModifiers();
            System.out.println("构造器的修饰符：" + Modifier.toString(modifiers));

            String name2 = constructor.getName();
            System.out.println("构造器名：" + name2);

            //形参列表
            System.out.println("形参列表：");
            Class[] parameterTypes = constructor.getParameterTypes();
            for (Class parameterType : parameterTypes) {
                System.out.println(parameterType);
            }

            //异常列表
            System.out.println("异常列表：");
            Class<?>[] exceptionTypes = constructor.getExceptionTypes();
            for (Class<?> exceptionType : exceptionTypes) {
                System.out.println(exceptionType);
```

```
            }
        }
        System.out.println("=---------------------------------");
        //(8)方法们
        Method[] declaredMethods = clazz.getDeclaredMethods();
        for (Method method : declaredMethods) {
            //修饰符、返回值类型、方法名、形参列表 、异常列表
            int modifiers = method.getModifiers();
            System.out.println("方法的修饰符：" + Modifier.toString(modifiers));

            Class<?> returnType = method.getReturnType();
            System.out.println("返回值类型:" + returnType);

            String name2 = method.getName();
            System.out.println("方法名：" + name2);

            //形参列表
            System.out.println("形参列表：");
            Class[] parameterTypes = method.getParameterTypes();
            for (Class parameterType : parameterTypes) {
                System.out.println(parameterType);
            }

            //异常列表
            System.out.println("异常列表：");
            Class<?>[] exceptionTypes = method.getExceptionTypes();
            for (Class<?> exceptionType : exceptionTypes) {
                System.out.println(exceptionType);
            }
        }

    }
}
```

3.3 创建任意类型的对象

> 两种方式：
>
> 1、直接通过Class对象来实例化（要求必须有无参 public构造）
>
> 2、通过获取构造器对象来进行实例化

1. 直接实例化对象

```
@Test
    public void test2()throws Exception{
        Class<?> clazz = Class.forName("com.atguigu.test.Student");
        //Caused by: java.lang.NoSuchMethodException:
com.atguigu.test.Student.<init>()
        //即说明Student没有无参构造，就没有无参实例初始化方法<init>
        Object stu = clazz.newInstance();
        System.out.println(stu);
    }

    @Test
    public void test1() throws ClassNotFoundException,
InstantiationException, IllegalAccessException{
//      AtGuigu obj = new AtGuigu();//编译期间无法创建
```

```
        Class<?> clazz = Class.forName("com.atguigu.test.AtGuigu");
        //clazz代表com.atguigu.test.AtGuigu类型
        //clazz.newInstance()创建的就是AtGuigu的对象
        Object obj = clazz.newInstance();
        System.out.println(obj);
    }
```

2. 构造函数实例化

> 如果构造器的权限修饰符修饰的范围不可见，也可以调用setAccessible(true)

```
public class TestNewInstance {
    @Test
    public void test3()throws Exception{
        //(1)获取Class对象
        Class<?> clazz = Class.forName("com.atguigu.test.Student");
        /*
         * 获取Student类型中的有参构造
         * 如果构造器有多个，我们通常是根据形参【类型】列表来获取指定的一个构造器的
         * 例如：public Student(int id, String name)
         */
        //(2)获取构造器对象
        Constructor<?> constructor =
clazz.getDeclaredConstructor(int.class,String.class);

        //(3)创建实例对象
        // T newInstance(Object... initargs)  这个Object...是在创建对象时，给有参构造的实参列表
        Object obj = constructor.newInstance(2,"张三");
        System.out.println(obj);
    }
}
```

3.4 操作任意类型的属性

> (1) 获取该类型的Class对象
> Class clazz = Class.forName("com.atguigu.bean.User");
>
> (2) 获取属性对象
> Field field = clazz.getDeclaredField("username");
>
> (3) 设置属性可访问
>
> field.setAccessible(true);
>
> (4) 创建实例对象：如果操作的是非静态属性，需要创建实例对象
> Object obj = clazz.newInstance();
>
> (4) 设置属性值
>
> field.set(obj,"chai");
> (5) 获取属性值
> Object value = field.get(obj);
>
> 如果操作静态变量，那么实例对象可以省略，用null表示!

```
public class TestField {
    public static void main(String[] args)throws Exception {
        //1、获取Student的Class对象
        Class clazz = Class.forName("com.atguigu.test.Student");

        //2、获取属性对象，例如：id属性
        Field idField = clazz.getDeclaredField("id");

        //3、如果id是私有的等在当前类中不可访问access的，我们需要做如下操作
        idField.setAccessible(true);

        //4、创建实例对象，即，创建Student对象
        Object stu = clazz.newInstance();

        //5、获取属性值
        /*
         * 以前：int 变量= 学生对象.getId()
         * 现在：Object id属性对象.get(学生对象)
         */
        Object value = idField.get(stu);
        System.out.println("id = "+ value);

        //6、设置属性值
        /*
         * 以前：学生对象.setId(值)
         * 现在：id属性对象.set(学生对象,值)
         */
        idField.set(stu, 2);

        value = idField.get(stu);
        System.out.println("id = "+ value);

    //静态属性  xx.set(null 表示静态, 11);  xx.get(null)
    }
}
```

3.5 操作任意类型的方法

> (1) 获取该类型的Class对象
>
> Class clazz = Class.forName("com.atguigu.service.UserService");
>
> (2) 获取方法对象
>
> Method method = clazz.getDeclaredMethod("login",String.class,String.class);
>
> (3) 创建实例对象
>
> Object obj = clazz.newInstance();
>
> (4) 调用方法
>
> Object result = method.invoke(obj,"chai","123");
>
> 如果方法的权限修饰符修饰的范围不可见，也可以调用setAccessible(true)
>
> 如果方法是静态方法，实例对象也可以省略，用null代替

```
public class TestMethod {
    @Test
    public void test()throws Exception {
        // 1、获取Student的Class对象
        Class<?> clazz = Class.forName("com.atguigu.test.Student");
```

```
        //2、获取方法对象
        /*
         * 在一个类中，唯一定位到一个方法，需要：（1）方法名（2）形参列表，因为方法可能重载
         *
         * 例如：void setName(String name)
         */
        Method method = clazz.getDeclaredMethod("setName", String.class);

        //3、创建实例对象
        Object stu = clazz.newInstance();

        //4、调用方法
        /*
         * 以前：学生对象.setName(值)
         * 现在：方法对象.invoke(学生对象，值)
         */
        method.invoke(stu, "张三");

        System.out.println(stu);
    }
}
```

四、注解讲解

6.1 概念

> 注解(Annotation)：是代码里的特殊标记, 程序可以读取注解，一般用于替代配置文件。
>
> 开发人员可以通过注解告诉类如何运行。
>
> 在Java技术里注解的典型应用是：可以通过反射技术去得到类里面的注解，以决定怎么去运行类。

6.2 定义注解

> 定义注解使用@interface关键字，注解中只能包含属性。
>
> 常见注解：@Override、@Deprecated

> 案例演示：

```
public @interface MyAnnotation {
    //属性(类似方法)
    String name() default "张三";
    int age() default 20;
}
```

6.3 注解属性类型

> - String类型
> - 基本数据类型
> - Class类型
> - 枚举类型
> - 注解类型

- 以上类型的一维数组

6.4 元注解

元注解：用来描述注解的注解。

@Retention:用于指定注解可以保留的域。

- RetentionPolicy.CLASS：
  注解记录在class文件中，运行Java程序时, JVM不会保留，此为默认值。
- RetentionPolicy.RUNTIME： RetensionPolicy.RUNTIME
  注解记录在 class文件中，运行Java程序时，JVM会保留，程序可以通过反射获取该注释
- RetentionPolicy.SOURCE:
  编译时直接丢弃这种策略的注释。

@Target：指定注解用于修饰类的哪个成员。

- ElementType.ANNOTATION_TYPE 可以给一个注解进行注解
- ElementType.CONSTRUCTOR 可以给构造方法进行注解
- ElementType.FIELD 可以给属性进行注解
- ElementType.LOCAL_VARIABLE 可以给局部变量进行注解
- ElementType.METHOD 可以给方法进行注解
- ElementType.PACKAGE 可以给一个包进行注解
- ElementType.PARAMETER 可以给一个方法内的参数进行注解
- ElementType.TYPE 可以给一个类型进行注解，比如类、接口、枚举

6.5 案例演示

注解声明和反射获取注解

PersonInfo注解类:

```
@Retention(value=RetentionPolicy.RUNTIME)
@Target(value= {ElementType.METHOD})
public @interface PersonInfo {
    String name();
    int age();
    String sex();
}
```

Person类：

```
public class Person {

    @MyAnnotation()
    public void show() {

    }

    //@MyAnnotation2(value="大肉",num=25)
    public void eat() {

    }

    @PersonInfo(name="小岳岳",age=30,sex="男")
    public void show(String name,int age,String sex) {
        System.out.println(name+"==="+age+"===="+sex);
```

```
    }

}
```

> TestAnnotation类:

```
public class Demo {
    public static void main(String[] args) throws Exception{
        //(1)获取类对象
        Class<?> class1=Class.forName("com.qf.chap17_5.Person");
        //(2)获取方法
        Method method=class1.getMethod("show",
String.class,int.class,String.class);
        //(3)获取方法上面的注解信息 personInfo=null
        PersonInfo personInfo=method.getAnnotation(PersonInfo.class);
        //(4)打印注解信息
        System.out.println(personInfo.name());
        System.out.println(personInfo.age());
        System.out.println(personInfo.sex());
        //(5)调用方法
        Person yueyue=(Person)class1.newInstance();
        method.invoke(yueyue,
personInfo.name(),personInfo.age(),personInfo.sex());

    }
}
```