反射

1. 引入
   1. 框架使用(基于反射)
2. 类的加载步骤
   1. 加载(将class文件加载到内存中，由ClassLoader创建该类的Class对象)
   2. 连接(内存分配,二进制文件转换)
   3. 初始化
3. ClassLoader(类加载器)
   1. 加载并创建class对象
   2. 分类
      1. 根类加载器
      2. 扩展类加载器
      3. 系统类加载器
4. 反射
   1. 反射概念
      1. 在运行阶段，动态获取类信息和动态调用方法的机制
         1. 在编译阶段不清楚需要加载类
         2. Class的”内省”机制(通过该类的Class对象可以获取所有的信息)
   2. 反射的API
      1. Class类: 封装类的所有信息
      2. Field：封装属性的所有信息
      3. Method:封装方法的所有信息
      4. Contructer：封装构造方法的信息。
   3. 获取Class对象
      1. 获取Class对象

|  |
| --- |
| //1.获取Class对象  Class clz = Class.*forName*("com.lee.reflect.Student");  //读取配置文件(动态创建调用)    //2.通过类名.class(加锁)  Class clz1 = Student.**class**;    //3.对象.getClass() (类的Class对象时唯一的)  Student stu = **new** Student();  Class clz2 = stu.getClass(); |

* + 1. 读取配置文件

|  |
| --- |
| //1.配置文件  //1.1 properties(属性文件 key,value)(Properties)  //1.2 xml文件 (标签<book></book>)    //读取配置文件  Properties pros = **new** Properties();  //.properties -流 classloader.getResourceAsStream()  InputStream is = ReflectTest.**class**  .getClassLoader()  .getResourceAsStream("fruit.properties");  //加载key/value属性列表  pros.load(is);  //根据key获取value  String fruit = pros.getProperty("fruit");  is.close();    Person p = **new** Person();  Class clz = Class.*forName*(fruit);  //创建对象(调用无参构造)  Fruit f = (Fruit) clz.newInstance();  p.buy(f); |

* 1. Field的使用

|  |
| --- |
| 2.查看属性  /\*Field[] fields = clz.getFields();  for (Field field : fields) {  System.out.println(field.getName());  }\*/  /\*Field[] fields = clz.getDeclaredFields();  for (Field field : fields) {  System.out.println(field.getName());  }\*/  //获取指定属性  /\*Field field = clz.getField("no");  System.out.println(field);\*/  /\*Field field = clz.getDeclaredField("no");  System.out.println(field);\*/    /\*FieldDemo demo = new FieldDemo();  demo.setNo(1);  System.out.println(demo.getNo());\*/  FieldDemo demo = **new** FieldDemo();  SetInt():设置基本类型的值  Set(obj,value):设置引用类型值  getInt(obj):获取基本类型的值  get(obj):获取引用类型的值  setAcceiable(true):设置访问权限 |

* 1. Method的使用
     1. Class对象获取(封装了类中方法的信息)

|  |
| --- |
| 2.获取method的对象  /\*Method[] methods = clz.getMethods();  for (Method method : methods) {  System.out.println(method);  }\*/  /\*Method[] declaredMethods = clz.getDeclaredMethods();  for (Method method : declaredMethods) {  System.out.println(method);  }\*/    /\*Method method = clz.getMethod("setName", String.class);  System.out.println(method);\*/    Method m2 = clz.getDeclaredMethod("show");  //执行方法(对象 参数)  m2.setAccessible(**true**);  m2.invoke(clz.newInstance()); |

* 1. Constructer使用

|  |
| --- |
| Constructor<Stu> constructor =clz.getConstructor(String.**class**,**int**.**class**);  Stu stu = constructor.newInstance("zs",12);  System.***out***.println(stu);    /\*clz.getConstructors()  clz.getDeclaredConstructors()  clz.getDeclaredConstructor();\*/ |

1. 代理
   1. 什么是代理

真正对象被访问之前和之后实现预处理，过滤等操作的方式被称为代理模式。代理实现对对象访问的控制。

* 1. 分类(接口+实现类)
     1. 静态代理

编译阶段为每个需要访问的方法设置代理方法。（创建代理类）

* + 1. 动态代理

运行阶段，为每个需要访问的方法创建代理方法。(运行时直接创建代理对象)

* + 1. 静态代理实现

|  |
| --- |
| 目标类和接口:  **public** **interface** Person {    **void** play();  **void** swim();    }  **public** **class** YifengLee **implements** Person{  @Override  **public** **void** play() {  System.***out***.println("play");  }  @Override  **public** **void** swim() {  System.***out***.println("swim");  }  }  代理类:  **public** **class** Songzhe **implements** Person {  Person p;    **public** Songzhe(Person p) {  **this**.p = p;  }  @Override  **public** **void** play() {  System.***out***.println("热身");  p.play();  System.***out***.println("擦汗");  }  @Override  **public** **void** swim() {  System.***out***.println("热身");  p.swim();  System.***out***.println("擦汗");  }  } |

优点:解决代码耦合问题

缺点:代码复用性差

* + 1. 动态代理(反射)