01_反射(类的加载概述和加载时机)

- A:类的加载概述
  - 当程序要使用某个类时，如果该类还未被加载到内存中，则系统会通过加载，连接，初始化三步来实现对这个类进行初始化。
  - 加载
    - 就是指将class文件读入内存，并为之创建一个Class对象。任何类被使用时系统都会建立一个Class对象。
  - 连接
    - 验证 是否有正确的内部结构，并和其他类协调一致
    - 准备 负责为类的静态成员分配内存，并设置默认初始化值
    - 解析 将类的二进制数据中的符号引用替换为直接引用
  - 初始化 就是对变量的初始化
- B:加载时机
  - 创建类对象的实例
  - 访问类的静态变量，或者为静态变量赋值
  - 调用类的静态方法
  - 使用反射方式来强制创建某个类或接口对应的java.lang.Class对象
    - Class.forName("com.mysql.jdbc.Driver");创建了Driver类的运行时对象
  - 初始化某个类的子类
  - 直接使用java.exe命令来运行某个主类

02_反射(类加载器的概述和分类)

- A:类加载器的概述
  - 负责将.class文件加载到内存中，并为之生成对应的Class对象。虽然我们不需要关心类加载机制，但是了解这个机制我们就能更好的理解程序的运行。
- B:类加载器的分类
  - Bootstrap ClassLoader 根类加载器
  - Extension ClassLoader 扩展类加载器
  - Sysetm ClassLoader 系统类加载器
- C:类加载器的作用
  - Bootstrap ClassLoader 根类加载器
    - 也被称为引导类加载器，负责Java核心类的加载
    - 比如System,String等。在JDK中JRE的lib目录下rt.jar文件中
  - Extension ClassLoader 扩展类加载器
    - 负责JRE的扩展目录中jar包的加载。
    - 在JDK中JRE的lib目录下ext目录
  - Sysetm ClassLoader 系统类加载器
    - 负责在JVM启动时加载来自java命令的class文件，以及classpath环境变量所指定的jar包和类路径

03_反射(反射概述)

- A:反射概述
  - JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；
  - 对于任意一个对象，都能够调用它的任意一个方法和属性；
  - 这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。
  - 要想解剖一个类,必须先要获取到该类的字节码文件对象。
  - 而解剖使用的就是Class类中的方法，所以先要获取到每一个字节码文件对应的Class类型的对象。
- B:三种方式
  - a:Object类的getClass()方法,判断两个对象是否是同一个字节码文件
  - b:静态属性class,锁对象
  - c:Class类中静态方法forName(),读取配置文件
- C:案例演示
  - 获取class文件对象的三种方式

04_反射结合工厂模式

- 工厂模式
  - 工厂模式是我们最常用的实例化对象模式了，是用工厂方法代替new操作的一种模式。
- 榨汁的案例,使用工厂设计模式!

```
interface Fruit {
    public void squeeze();
}
class Apple implements Fruit {
 public void squeeze() {
    System.out.println("榨出一杯苹果汁儿");
 }
}

class Factory {
 public static Fruit getInstance(String className) {
    if ("apple".equals(className)) {
        return new Apple();
    }
    return null;
 }
}

public class Demo01 {
 public static void main(String[] args) {
    Fruit f = Factory.getInstance("apple");
    f.squeeze();
 }
}
```

- 如果在以上案例需求的基础上再多加几样水果:香蕉(Banana)桔子(Orange)

- 那么要怎么处理才能保证代码结构的合理性了?
- 使用反射解决,代码如下

```
class Factory {
    public static Fruit getInstance(String className) throws Exception {
        Class<?> clazz = Class.forName(className);
        Fruit fruit = (Fruit) clazz.newInstance();
        return fruit;
    }
 }

 public class Demo01 {
    public static void main(String[] args) {
        Fruit f;
        try {
            f = Factory.getInstance("com.qzw.demo.Orange");
            f.squeeze();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
 }
```

05_反射(通过反射获取构造方法并使用)

- Constructor
  - Class类的newInstance()方法是使用该类无参的构造函数创建对象, 如果一个类没有无参的构造函数, 就不能这样创建了,可以调用Class类的getConstructor(String.class,int.class)方法获取一个指定的构造函数然后再调用Constructor类的newInstance("张三",20)方法创建对象
  - 通过私有构造器

```
try {
    Class<?> clazz = Class.forName("com.qzw.bean.Student");
    Constructor<?> c1 = clazz.getDeclaredConstructor(long.class,String.class);
    c1.setAccessible(true);
    Object bj = c1.newInstance(1,"张三");
    System.out.println(bj);
} catch (Exception e) {
    e.printStackTrace();
}
```

  - 通过公共构造器

```
try {
    Class<?> clazz = Class.forName("com.qzw.bean.Student");
```

```
        Constructor<?> c1 = clazz.getConstructor(long.class,String.class);
        Object bj = c1.newInstance(1,"张三");
        System.out.println(bj);
    } catch (Exception e) {
        e.printStackTrace();
    }
```

06_反射(通过反射获取成员变量并使用)

- Field
  - Class.getField(String)方法可以获取类中的指定字段(可见的), 如果是私有的可以用getDeclaedField("name")方法获取,通过set(obj, "李四")方法可以设置指定对象上该字段的值, 如果是私有的需要先调用setAccessible(true)设置访问权限,用获取的指定的字段调用get(obj)可以获取指定对象中该字段的值
  - 操作公共成员变量

```
Class<?> clazz = Class.forName("com.qzw.bean.MyClass02");
//1,获取到MyClass02中的成员变量
Field field1 = clazz.getField("name1");
MyClass02 obj = (MyClass02) clazz.newInstance();
//获取该字段对象的值，是要获取哪个对象中的name1属性值
Object name = field1.get(obj);
field1.set(obj, "小邱");
```

  - 操作私有成员变量

```
Class<?> clazz = Class.forName("com.qzw.bean.MyClass02");
MyClass02 obj = (MyClass02) clazz.newInstance();
//1,获取到name2字段对象
Field field = clazz.getDeclaredField("name2");
field.setAccessible(true);
field.set(obj, "大球球");
```

07_反射(通过反射获取方法并使用)

- Method
  - Class.getMethod(String, Class...) 和 Class.getDeclaredMethod(String, Class...)方法可以获取类中的指定方法,调用invoke(Object, Object...)可以调用该方法,

```
Class.getMethod("eat").invoke(obj)
Class.getMethod("eat",int.class).invoke(obj,10)
```

08_反射(通过反射越过泛型检查)

- A:案例演示

- ArrayList<Integer>的一个对象，在这个集合中添加一个字符串数据，如何实现呢?

```
List<Integer> list = new ArrayList<>();
list.add(1);
list.add(2);
list.add(3);
System.out.println(list);
//往list中添加一段字符串
//可以使用反射来做
//泛型仅仅存在于编译阶段,当程序运行时,泛型都已经擦除了!!
Class clazz = Class.forName("java.util.List");
//获取到list类中add方法
Method method = clazz.getMethod("add", Object.class);
method.invoke(list, "hello");
System.out.println(list);
```

09_反射通用方法

- A.需求
  - 给指定对象的指定属性设置指定值
- B:案例演示

```
public void setProperty(Object obj, String fieldName, Object fieldValue){

}
```

10_反射结合配置文件

- 已知一个类，定义如下:

```
package cn.qzw.demo;
public class DemoClass {
    public void run() {
        System.out.println("welcome!");
    }
}
```

  - (1) 写一个Properties格式的配置文件，配置类的完整名称。
  - (2) 写一个程序，读取这个Properties配置文件，获得类的完整名称并加载这个类，用反射的方式运行run方法。

11_静态代理设计模式

- A.前提
  - 1,代理类和被代理类实现同一个接口
  - 2,代理类中持有被代理类的对象

- 3,在增强方法中使用被代理类对象调用方法

12_装饰者设计模式

- 前提
  - 1,代理类和被代理类实现同一个接口
  - 2,代理类中持有被代理类的引用
  - 3,在增强方法中使用被代理类对象调用方法

13_动态代理的概述和实现

- A:动态代理概述
  - 代理：本来应该自己做的事情，请了别人来做，被请的人就是代理对象。
  - 举例：春节回家买票让人代买
  - 动态代理：在程序运行过程中产生的这个代理对象,而程序运行过程中产生对象其实就是我们刚才反射讲解的内容，所以，动态代理其实就是通过反射来生成一个代理
  - 在Java中java.lang.reflect包下提供了一个Proxy类和一个InvocationHandler接口，通过使用这个类和接口就可以生成动态代理对象。JDK提供的代理只能针对接口做代理。我们有更强大的代理cglib，Proxy类中的方法创建动态代理类对象
  - 代理对象

```
public class MyInvocationHandler implements InvocationHandler {

    private Object target;

    public MyInvocationHandler() {
        super();
    }

    public MyInvocationHandler(Object target) {
        super();
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        System.out.println("权限校验");
        method.invoke(target, args);
        System.out.println("日志记录");
        return null;
    }

}
```

- 被代理对象

```
public interface Person {
    public void add();
}

public class Teacher implements Person {
    @Override
    public void add() {
        System.out.println("添加老师");
    }
}
```

- 测试代码

```
Teacher t = new Teacher();
Person t1 = (Person) Proxy.newProxyInstance(t.getClass().getClassLoader(),
t.getClass().getInterfaces(), new MyInvocationHandler(t));
t1.add();
```