day21_设计模式和反射

一 内容回顾（列举前一天重点难点内容）

1.1 教学重点:

```
1.掌握NIO的原理
2.掌握NIO的基本使用
3.掌握JVM的工作原理
4.掌握JVM各区块的基本工作原理
5.掌握堆区的分区使用
6.掌握堆内存优化
```

1.2 教学难点:

```
1.NIO的一些高阶使用
2.JVM的垃圾回收机制GC
3.类加载器的工作原理
```

二 教学目标

```
1.掌握设计模式的分类
2.掌握设计模式的六大原则
3.掌握单例设计模式
4.掌握模板设计模式
5.了解适配器设计模式
6.了解代理设计模式
7.了解反射原理
```

三 教学导读

3.1. 设计模式

- 定义

前人总结出来的对一些常见问题的解决方案,后人直接拿来解决特定问题而存在的解题思路。

- 历史

Erich Gamma(艾里希戈莫) 博士是设计模式的开创者,Eclipse的总设计师,IBMOTL技术主管,JUnit共同创作者,敏捷开发的创始人

Ralph Johnso,康奈尔大学获得计算机博士学位,伊利诺伊大学教授,著有<重构与模式>

- 分类

常用的设计模式:单例,工厂,代理,适配器,装饰,模板,观察者等,一共有23种

第一:创建型模式:如何创建对象以及何时创建对象

```
包括:
工厂模式(FACTORY METHOD)
抽象工厂模式
建造(BUILDER)模式
代理模式(SINGLETON)
原型模式(Prototype)
```

第二:结构型模式:对象该如何组织以及采用什么样的结构更合理

```
包括：
适配器(Adapter)模式
合成(Composite)模式
装饰(Decorator)模式
代理(Proxy)模式
享元(Flyweight Pattern)模式
门面(Facade)模式
桥梁(Bridge)模式
```

第三:行为型模式:规定了各个对象应该具备的职责以及对象间的通信模式

```
包括：
策略(Strategy)模式
模板方法(Template Method)模式
观察者(observer)模式
迭代子(Iterator)模式
责任链模式
命令模式
备忘录模式
状态模式
访问者模式
解释器模式
调停者模式
```

- 六大基本原则

1.单一职责原则(SRP)

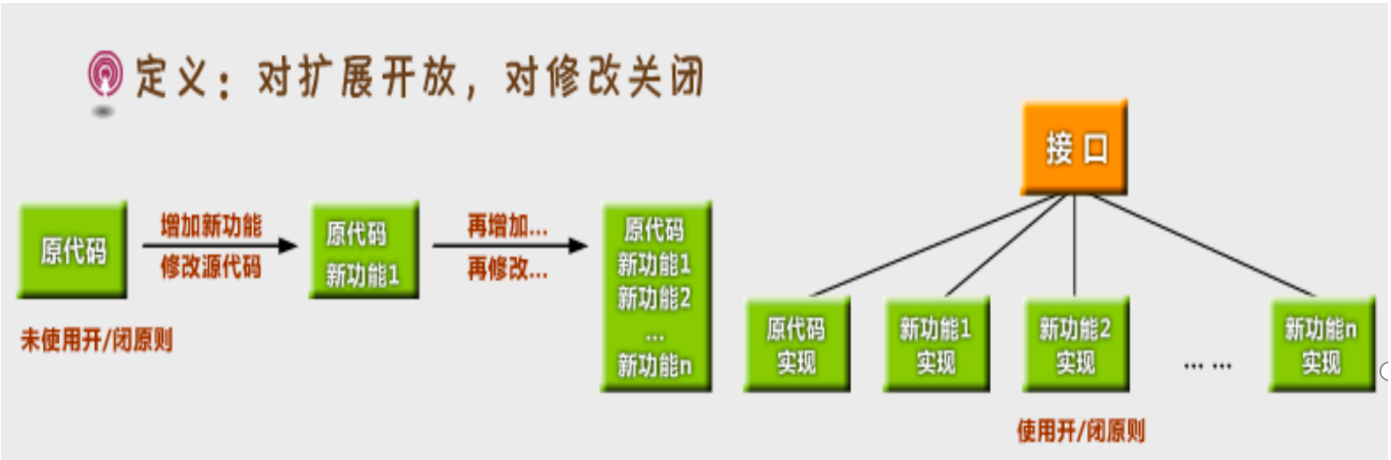
定义:系统中的每一个类都应该只有一个职责

好处:高内聚低耦合

2.开闭原则(OCP)

定义:对扩展开放,对修改关闭

好处:适应性和灵活性,稳定性和延续性,可复用性和可稳定性



3.里氏替换原则(LSP)

定义:在任何父类出现的地方都可以用它的子类来替换,且不影响功能

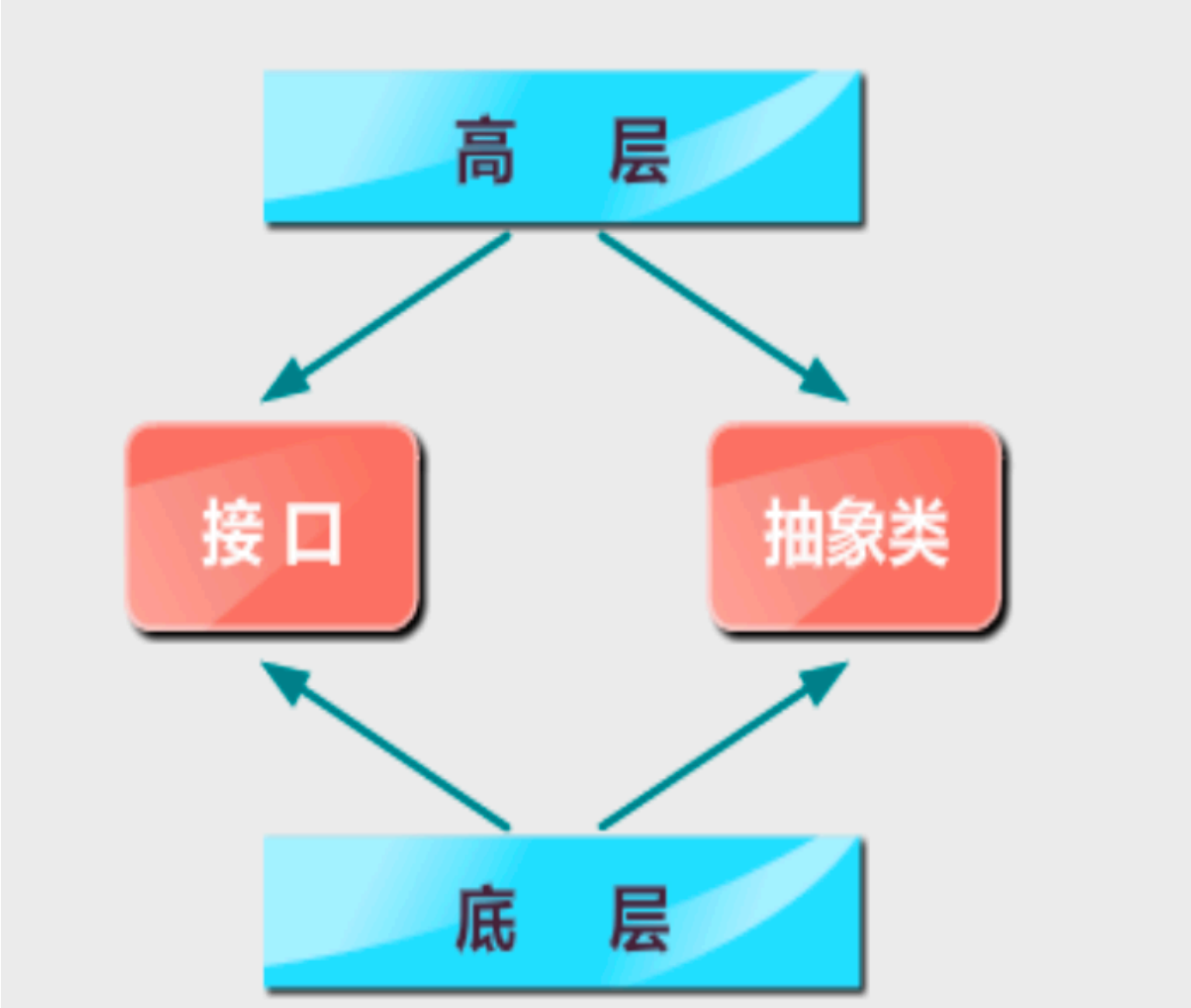
好处:是加强程序的健壮性，同时版本升级也可以做到非常好的兼容性，增加子

类，原有的子类还可以继续运行。

4.依赖倒置原则(DIP)

定义:高层模块不应该依赖底层模块,两者都应该依赖其抽象;抽象不应该依赖细节,细节应该依赖抽象

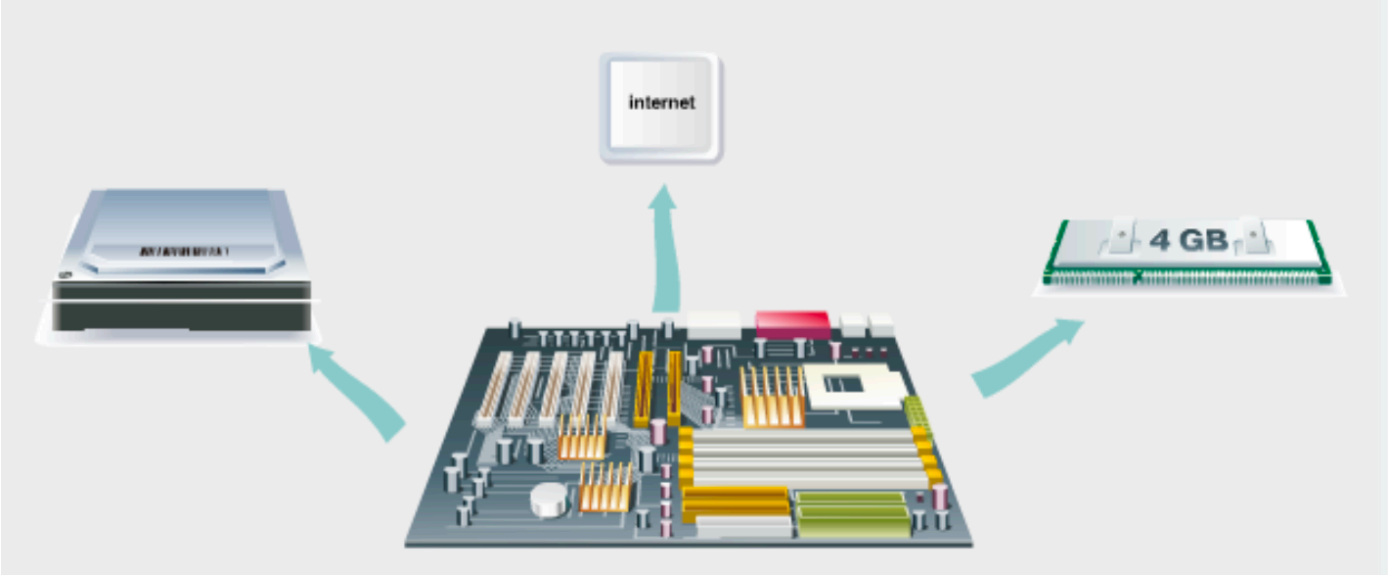
好处:提高程序的稳定性,可维护性,可扩展性



5.接口隔离原则(ISP)

定义:使用多个专门的接口比使用单一的总接口要好

好处:不强迫新功能实现不需要的方法



6.迪米特原则(LOP)

定义:一个对象应该对其他对象尽可能少的了解

好处:高内聚,低耦合

缺点:通信效率降低,产生大量的中介类

3.2. 反射

动态获取类的字节码文件,并对其成员进行抽象

整体的含义:就是想通过字节码文件直接创建对象.

JAVA反射机制是在运行状态中，对于任意一个实体类，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方法和属性；这种动态获取信息以及动态调用对象方法的功能称为java语言的反射机制。

JAVA反射（放射）机制："程序运行时，允许改变程序结构或变量类型，这种语言称为动态语言"。从这个观点看，Perl，Python，Ruby是动态语言，C++，Java，C#不是动态语言。但是JAVA有着一个非常突出的动态相关机制：Reflection，用在Java身上指的是我们可以于运行时加载、探知、使用编译期间完全未知的classes。换句话说，Java程序可以加载一个运行时才得知名称的class，获悉其完整构造（但不包括methods定义），并生成其对象实体、或对其fields设值、或唤起其method

Class: 不是用来声明类的时候用到的关键字。（注意：首字母大写）。他是一个类。

是一个比较特殊的类。继承自Object类的。可以用Class对象来描述一个类中的成员。

Class对象的获取：

1.通过对象.getClass()

通过类.class

3.通过Class类的静态方法 Class.forName(String name)

注意：参数name需要是类的全限定名

如果要访问的是一个内部类，则需要写编译后.class文件的名字

Class类常用方法：

```
setAccessible(boolean flag);
// 从字面非常容易被理解为设置一个属性、方法、构造方法的可访问性。
// 这个表示的是是否需要进行访问权限的检查。
// 如果参数是true：则在访问对应的成员的时候，不去进行访问权限的检查，直接访问。
// 如果参数是false：则在访问对应的成员之前，先去检查访问权限
```

四 教学内容

4.1. 设计模式

4.1.1. 单例设计模式(会)

- 定义

一个类只允许有一个对象,建立一个全局的访问点,提供出去供大家使用.

- 分析:

```
1.我们肯定要建立一个单例类来描述

2.只允许有一个对象

3.全局的访问点:说的就是当前的s----通过static实现的

4.提供出去

5.给大家使用
```

- 分类

```
饿汉式单例:在定义类中的变量时,直接给值(看下面的代码)

懒汉式单例:当通过公共方法调用s2的时候才给值(看下面的代码)
```

- 作用

总括:1.传值 .作为全局的访问点.

解决一个全局使用的类，频繁创建和销毁。拥有对象的唯一性，并保证内存中对象的唯一。可以节省内存，因为单例共用一个实例，有利于Java的垃圾回收机制。也就是控制资源使用，通过线程同步来控制资源的并发访问；

控制实例产生的数量，达到节约资源的目的；

- 使用场景

1.统计当前在线人数（网站计数器）：用一个全局对象来记录。

2.打印机（设备管理器）：当有两台打印机，在输出同一个文件的时候只一台打印机进行输出。

3.数据库连接池（控制资源）：一般是采用单例模式，因为数据库连接是一种连接数据库资源，不易频繁创建和销毁。

数据库软件系统中使用数据库连接池，主要是节省打开或者关闭数据库连接所引起的效率损耗，这种效率的损耗还是非常昂贵的，因此用单例模式来维护，就可以大大降低这种损耗。

4.应用程序的日志（资源共享）：一般日志内容是共享操作，需要在后面不断写入内容所以通常单例设计。

- 示例代码

```
public class Demo11 {
```

```
    public static void main(String[] args) {
        SingleInstance1 singleInstance1 = SingleInstance1.getInstance();
        SingleInstance1 singleInstance2 = SingleInstance1.getInstance();
        System.out.println(singleInstance1 == singleInstance2);//true
    }
}
//饿汉式:在定义s1这个变量时,就直接给值
class  SingleInstance1{
    //2.定义一个私有化的静态的当前类类型的成员变量并完成赋值.
    private  static final SingleInstance1 s1 = new SingleInstance1();
    //1.先将构造方法私有化
    private SingleInstance1(){

    }
    //3.创建一个公共的方法,将当前的私有的成员变量提供出去
    public static  SingleInstance1 getInstance(){
        return s1;
    }

    //注意:一般我们在写单例的时候,下面的方法不考虑
    //克隆的意思:我们通过克隆也可以得到当前的对象,注意:克隆!=new
//    @Override
//    protected Object clone() throws CloneNotSupportedException {
//        return s1;
//    }

    //功能区
```

```
    //属性
    //行为
}
//懒汉式:当通过公共方法调用s2的时候才给值
class  SingleInstance2{
    //2.定义一个私有化的静态的当前类类型的成员变量并完成赋值.
    private  static  SingleInstance2 s2 = null;
    //1.先将构造方法私有化
    private SingleInstance2(){

    }
    //3.创建一个公共的方法,将当前的私有的成员变量提供出去
    public static  SingleInstance2 getInstance(){
        if (s2 == null){
            s2 = new SingleInstance2();
        }
        return s2;
    }

    //功能区
    //属性
    //行为
}
```

下面是功能实现演示代码:

```
//饿汉式
class  SingleInstance {
    private static  final SingleInstance s = new
SingleInstance();
    private SingleInstance(){}
    public  static SingleInstance getInstance(){
        return s;
    }
```

```
8
9       //功能区
10      //功能区(包括成员变量和成员方法)
11      //注意点:对于功能区来说:一般都是非静态的成员
12      int num;
13
14      //成员方法:实现方法的全局访问
15      public  void  show(){
16          System.out.println("show");
17      }
18  }
19
20  public class Demo2 {
21      public static void main(String[] args) {
22          //实例:
23          //传值:
24          //有一个A类和一个B类,A类中有一个变量num1,B类中有一个变
    量num2
25          //创建A类的对象a,给变量赋值num1=4 ,创建B类的对象b,要
    实现的功能:将num1的值传给num2
26
27          //第一种方法
28          //直接传值:不建议使用
29          A a = new A();
30          B b = new B();
31          //b.num2 = a.num1;
32
33          //第二种方法
34          //通过参数传值
35          //b.bText(a);
36
37          //第三种方法
38          //通过单例传值
```

```
        a.singleA();
        b.singleB();
    }
}

class A{
    String name;
    private int num1 = 4;
    public int getNum1() {
        return num1;}
    public void setNum1(int num1) {
        this.num1 = num1;}
    //用于单例
    public void singleA(){
        SingleInstance singleInstance =
SingleInstance.getInstance();
        singleInstance.num = num1;
        //在全局内访问show方法
        singleInstance.show();
    }
}
class B{
    private int num2;

    public void bText(A a) {
        num2 = a.getNum1();
    }
    //用于单例
    public void singleB(){
        SingleInstance singleInstance =
SingleInstance.getInstance();
        num2 = singleInstance.num;
        singleInstance.show();
```

```
70      }
71  }
```

4.1.2. 适配器设计模式(了解)

- 定义:

通常可以变相的理解成装饰设计模式

- 作用:

更好的复用性 如果功能已经存在，只是接口不兼容，通过适配器模式就可以让这些功能得到更好的复用。

更好的扩展性 在实现适配器功能时，可以调用自己开发的功能，从而自然的扩展系统的功能。

- 使用场景

系统需要使用现有已投产的类，而这些类的接口不符合系统的需要。 想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作。 Java中的数据库连接工具JDBC，JDBC定义一个客户端通用的抽象接口，每一个具体数据库引擎（如SQL Server、Oracle、MySQL等）的JDBC驱动软件都是一个介于JDBC接口和数据库引擎接口之间的适配器软件。

- 示例代码:要求在子类中只使用play方法

分析:Dog是继承了ZiMidel类,ZiMidel类实现了Inter接口

当Dog类想要实现Inter接口的一个方法的时候,如果直接实现Inter接口,就必须将所有的方法都实现,如果在Dog类与Inter接口之间插入一个类,让这个类去实现Inter接口的所有方法,作为这个类的子类只需要实现自己需要的方法.我们将中间的这个类就可以称为适配器类

```
interface Inter{
    public void play();
    public void song();
    public void run();
    public void eat();
    public void jump();
}
//适配器类
class ZiMidel implements Inter{

    @Override
    public void play() {
        // TODO Auto-generated method stub

    }

    @Override
    public void song() {
        // TODO Auto-generated method stub

    }

    @Override
    public void run() {
        // TODO Auto-generated method stub

    }

```

```
29      @Override
30      public void eat() {
31          // TODO Auto-generated method stub
32
33      }
34
35      @Override
36      public void jump() {
37          // TODO Auto-generated method stub
38
39      }
40
41  }
42
43
44  //创建狗类,我只想让她实现play方法?
45  class Dog extends ZiMidel{
46      public void play() {
47          // TODO Auto-generated method stub
48
49      }
50  }
51
52  class Cat extends ZiMidel{
53      public void song() {
54
55      }
56  }
```

4.1.3. 模板设计模式(会)

- 定义

我们在实现一个功能的时候,功能分成两部分,一部分是确定的,一部分是不确定的.将确定的部分交给当前类实现,将不确定的部分交给子类实现.子类实现的结果又会反过来影响确定部分的功能.

- 作用

模板设计模式是通过把不变的行为挪到一个统一的父类，从而达到去除子类中重复代码的目的、子类实现模板父类的某些细节，有助于模板父类的扩展.通过一个父类调用子类实现的操作，通过子类扩展增加新的行为，符合“开放-封闭原则”

- 使用场景

多个子类有共有的方法，并且逻辑基本相同.重要、复杂的算法，可以把核心算法设计为模板方法，周边的相关细节功能则由各个子类实现重构时，模板方法是一个经常使用的方法，把相同的代码抽取到父类中，然后通过构造函数约束其行为

- 代码演示

```
/*
 * 实例:计算一个功能的耗时
 * 分析:固定的功能:开始时间,结束时间
 * 不固定的功能:程序运行的时间
 */
public class Demo4 {
    public static void main(String[] args) {
        //测试
        Zi zi = new Zi();
        long value = zi.getTime();
        System.out.println(value);
    }
}

```

```
abstract class Fu{
    abstract public void function();
    public long getTime() {
        //开始时间
        long startTime = System.nanoTime();//获取的系统时间,单位纳秒
        //程序运行的时间
        function();
        //结束时间
        long endTime = System.nanoTime();

        return endTime-startTime;
    }
}

class Zi extends Fu{
    public void function() {
        for (int i = 0; i < 10; i++) {
            System.out.println("i:"+i);
        }
    }
}
```

4.1.4. 代理设计模式(了解)

4.1.4.1. 什么是代理模式

- 定义

我很忙，忙的没空理你，那你要找我呢就先找我的代理人吧，那代理人总要知道被代理人能做哪些事情不能做哪些事情吧，那就是两个人具备同一个接口，代理人虽然不能干活，但是被代理的人能干活呀。

- 作用

代理模式主要使用了 Java 的多态，干活的是被代理类，代理类主要是接活，你让我干活，好，我交给幕后的类去干，你满意就成，那怎么知道被代理类能不能干呢？同根就成，大家知根知底，你能做啥，我能做啥都清楚的很，同一个接口呗。

- 使用场景

1.一个对象，比如很大的一张图像，加载前可以用一个占位的图像来替代。

2.一个过程计算需要等待很长时间，并且需要再计算过程中展示结果。

3.一个存在于远程的对象，通过网络载入需要较长的时间，

4.验证用户对对象的访问权限。

4.1.4.2. 代理方法分类

- 静态代理
- 动态代理

4.1.4.3. 静态代理

- 作用:可以实现简单代理 根据OCP(对扩展开放,对修改关闭)的原则,在不改变原来类的基础上,给这个类增加额外的功能
- 缺点:代理对象要保证跟目标对象实现同样的接口,在维护的时候两个对象都要维护,而且代理对象实现的接口是死的,这时如果要给想实现不同功能的多个目标对象添加代理对象的话,要添加很多个类

针对这个确定,我们可以选择动态代理

- 代码实现

模拟功能:bingbing和chenchen找房住

解释说明:

```
分析:开始bingbing和chenchen自己找房,但是都有工作,而且没有房源,不好找.
然后找到有可以找房功能的代理类帮忙找房

我们发现:代理类在实现了找房功能后,还可以在找房前找房后再实现额外的功能,这就是代理的最终目的
```

定义找房功能的接口

```
public interface TestInter {
  public void findHouse();
}
```

定义bingbing类

```
public class Bingbing implements TestInter {
  public void findHouse() {
    System.out.println("冰冰来西三旗找房");
  }
}
```

定义chenchen类

```
public class ChenChen implements TestInter {
  public void findHouse() {
    System.out.println("晨晨去日本找房");
  }
}
```

定义代理类Agent

```
public class Agent implements TestInter{
```

```
  //先给他个人
  TestInter person;
  public Agent(TestInter person) {
    super();
    this.person = person;
  }
  //代理类在实现了找房功能后,还可以在找房前找房后再实现额外的功能扣中介费和哈哈大笑,这就是代理的最终目的
  public void findHouse() {
    System.out.println("扣一个月的房租作为中介费");
    person.findHouse();
    System.out.println("哈哈大笑");
  }
}
```

定义测试类

```
public class Test {
  public static void main(String[] args) {
    Bingbing bingbing = new Bingbing();
    //冰冰自己找房
    //bingbing.findHouse();
    Agent agent = new Agent(bingbing);
    //通过代理帮冰冰找房
    agent.findHouse();
  }
}
```

4.1.4.4. 动态代理

更新找房功能的接口,添加找妹子方法

```
public interface TestInter {
  public void findHouse();
  public void findMeizi();
}
```

更新bingbing类,添加找妹子方法实现

```
public class Bingbing implements TestInter {
  public void findHouse() {
    System.out.println("冰冰来西三旗找房");
  }
  @Override
  public void findMeizi() {
    // TODO Auto-generated method stub
    System.out.println("玩儿");
  }
}
```

更新chenchen类,添加找妹子方法实现

```
public class ChenChen implements TestInter {
  public void findHouse() {
    System.out.println("晨晨去日本找房");
  }
  @Override
  public void findMeizi() {
    // TODO Auto-generated method stub
    System.out.println("过七夕");
  }
}
```

新增吃饭接口

```
public interface TestEat {
  public void eat();
}
```

新增LangLang类

```
public class Langlang implements TestEat{
  @Override
  public void eat() {
    System.out.println("郎朗吃饭");

  }
}
```

更新代理类Agent

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

public class Agent implements InvocationHandler{
  //先给他个人
  //注意:指定给构造方法的参数要使用Object
  Object person;
  public Agent(Object person) {
    super();
    this.person = person;
  }

//  public void findHouse() {
//    System.out.println("扣一个月的房租作为中介费");
//    person.findHouse();
//    System.out.println("哈哈大笑");
```

```
//  }
  /**
   * 接口中的方法
   * 主要:这个方法在调用接口方法的时候,会被自动调动
   * 参数一:代理对象的引用
   * 参数二:目标对象的方法
   * 参数三:目标对象的方法参数
   *
   */
  @Override
  public Object invoke(Object proxy, Method method,
Object[] args) throws Throwable {
    System.out.println("扣一个月的房租作为中介费");
    Object object = method.invoke(person, args);
    System.out.println("哈哈大笑");
    return object;
  }
}
```

更新测试类

```
//使用动态代理
public class Test {
  //模拟功能:bingbing和chenchen找房住
  public static void main(String[] args) {
    //静态代理
//    Bingbing bingbing = new Bingbing();
//    //bingbing.findHouse();
//    Agent agent = new Agent(bingbing);
//    agent.findHouse();

    //动态代理

    //调用动态代理的方法实现功能
```

```
14      /**
15       *动态生成代理对象的方法--通过JDK内置的
java.lang.reflect.Proxy动态代理类完成代理对象的创建
16       *参数一:这里代表类加载器,代理类的类加载器要与目标类的类加载
器一致,类加载器用来装载内存中的字节码文件
17       *参数二:代理类与目标类实现的接口必须有相同的,即指定给代理类
的接口,目标类必须实现了
18       *参数三:代理类的构造方法生成的对象--注意:指定给构造方法的参
数要使用Object
19       *
20       */
21      //设置不同的接口,可以方便的实现不同的代理功能
22      System.out.println("**********bingbing使用Agent实现买
房*************");
23      TestInter testInter = new Bingbing();
24      TestInter object =
(TestInter)Proxy.newProxyInstance(testInter.getClass().
getClassLoader(), new Class[] {TestInter.class}, new
Agent(testInter));
25      //代理对象调动方法的时候,invoke方法会自动被调用
26      object.findHouse();
27
28      System.out.println("*****************langlnag使用
Agent实现吃饭************");
29      TestEat testEat = new Langlang();
30      TestEat object1 =
(TestEat)Proxy.newProxyInstance(testEat.getClass().getC
lassLoader(), new Class[]
{TestInter.class,TestEat.class}, new Agent(testEat));
31      object1.eat();
32    }
33  }
34
```

```
执行结果:
**********bingbing使用Agent实现买房******************
扣一个月的房租作为中介费
冰冰来西三旗找房
哈哈大笑
*****************langlnag使用Agent实现吃饭***********
扣一个月的房租作为中介费
郎朗吃饭
哈哈大笑
```

功能进一步优化:真正的动态代理

直接使用InvocationHandler创建匿名内部类干活儿,不再需要Agent类

```java
//使用动态代理
public class Test {
  //模拟功能:bingbing和chenchen找房住
  public static void main(String[] args) {
    //静态代理
//    Bingbing bingbing = new Bingbing();
//    //bingbing.findHouse();
//    Agent agent = new Agent(bingbing);
//    agent.findHouse();

    //动态代理
    TestInter testInter = new Bingbing();
    //调用动态代理的方法实现功能

    //进一步优化----直接使用InvocationHandler创建匿名内部类干活儿,不再需要Agent类

```

```
17      System.out.println("************bingbing租房
        ************");
18
        ((TestInter)Proxy.newProxyInstance(testInter.getClass()
        .getClassLoader(), new Class[] {TestInter.class},
19          new InvocationHandler() {
20            public Object invoke(Object proxy, Method
              method, Object[] args) throws Throwable {
21              System.out.println("扣一个月的房租作为中介费");
22              Object object = method.invoke(testInter,
                args);
23              System.out.println("哈哈大笑");
24              return object;
25            }
26          }
27      )).findHouse();
28
29      System.out.println("*************bingbing找妹子
        ******************");
30
        ((TestInter)Proxy.newProxyInstance(testInter.getClass()
        .getClassLoader(), new Class[] {TestInter.class},
31            new InvocationHandler() {
32              public Object invoke(Object proxy, Method
                method, Object[] args) throws Throwable {
33                System.out.println("扣一个月的房租作为中介费");
34                Object object = method.invoke(testInter,
                  args);
35                System.out.println("哈哈大笑");
36                return object;
37              }
38            }
39          )).findMeizi();
```

```
40
41      System.out.println("***********langlang洗手吃饭
*******");
42
43      TestEat testEat = new Langlang();
44      TestEat object3 =
(TestEat)Proxy.newProxyInstance(testEat.getClass().getC
lassLoader(), new Class[]
{TestInter.class,TestEat.class}, new
InvocationHandler() {
45          public Object invoke(Object proxy, Method method,
Object[] args) throws Throwable {
46              System.out.println("先洗后");
47              Object object = method.invoke(testEat, args);
48              System.out.println("哈哈大笑");
49              return object;
50          }
51      });
52      object3.eat();
53    }
54 }
55
56 执行结果:
57 ***********bingbing租房***********
58 扣一个月的房租作为中介费
59 冰冰来西三旗找房
60 哈哈大笑
61 ************bingbing找妹子******************
62 扣一个月的房租作为中介费
63 玩儿
64 哈哈大笑
65 ***********langlang洗手吃饭*******
66 先洗后
```

```
67 郎朗吃饭
68 哈哈大笑
69
70 总结:直接使用InvocationHandler实现动态代理,我们可以更加灵活的
   让代理添加功能.
```

4.2. 反射(了解)

实现原理分析

实现过程:

1.获取字节码文件对象

2.通过字节码文件对象获取对应的实例对象

3.给属性赋值(通过从属性中提取出来的类--Field)

4.调用方法(通过从方法中提取出来的类--Method)

4.2.1. 首先创建Person类

```
1  package com.qf.refect;
2
3  public class Person {
4    String name;
5    int age;
6    public String getName() {
7      return name;
8    }
9    public void setName(String name) {
10     this.name = name;
11   }
```

```java
  public int getAge() {
    return age;
  }
  public void setAge(int age) {
    this.age = age;
  }
  @Override
  public String toString() {
    return "Person [name=" + name + ", age=" + age + "]";
  }
  public Person(String name, int age) {
    super();
    this.name = name;
    this.age = age;
  }
  public Person() {
    super();
    // TODO Auto-generated constructor stub
  }
  //非静态的无参方法
  public int show() {
    System.out.println("show");
    return 3;
  }
  //非静态的有参方法
  public  void callPhone(String tel) {
    System.out.println("打电话给"+tel);
  }
  //静态的有参方法
  public static void run(int num) {
    System.out.println("run");
  }
```

```
}

```

4.2.2. 获取字节码文件对象

```
//1.通过Object提供的getClass()方法
//  首先必须要有一个对象     XXX
//2.通过每种数据类型都有的一个class属性
//  在使用的位置必须当前的类是可见的,因为这里要显示的使用这个类名,对类的依赖性太强,使用不方便     XXX
//3.Class类提供的一个静态方法forName(字符串)   字符串:包名+类名
//  我们只需要提供一个当前类的字符串形式即可
public class Demo1 {
    public static void main(String[] args) throws ClassNotFoundException {
        //1.通过Object提供的getClass()方法
        fun1();
        //2.通过每种数据类型都有的一个class属性
        fun2();
        //3.Class类提供的一个静态方法forName(字符串)   字符串:包名+类名
        fun3();
    }
    public static void fun1() {
        Person person = new Person();
        Person person1 = new Person();
        Class<?> class1 = person.getClass();
        Class<?> class2 = person.getClass();
        System.out.println(class1 == class2);//true
        System.out.println("value:"+(person.getClass() == person1.getClass()));
```

```
    }
    public static void fun2() {
        Class<?> class1 = Person.class;
        System.out.println(class1.getName());
    }
    public static void fun3() throws ClassNotFoundException {
        //注意:要保证至少字符串对应的类是存在的
        Class<?> class1 = Class.forName("com.qf.refect.Person");
    }
}
```

4.2.3. 通过字节码文件对象获取对应的实例对象

```
ackage com.qf.refect;
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

//2.通过字节码文件对象获取对应的实例对象
public class Demo2 {
    public static void main(String[] args) throws ClassNotFoundException, InstantiationException, IllegalAccessException, NoSuchMethodException, SecurityException, IllegalArgumentException, InvocationTargetException {
        //普通方式
        //Person person = new Person();

        //通过反射创建普通对象
        Class<?> class1 = Class.forName("com.qf.refect.Person");
        //方法一:通过无参的构造方法创建实例对象
```

```
14          fun1(class1);
15          //方法二:通过有参的构造方法创建实例对象
16          fun2(class1);
17      }
18      //方法一:通过无参的构造方法创建实例对象
19      public static void fun1(Class<?> cls) throws
InstantiationException, IllegalAccessException {
20          //创建实例对象
21          //这里相当于在newInstance方法的内部调用了无参的构造方
法
22          Object object = cls.newInstance();
23          Person person = (Person)object;
24          person.setName("bingbing");
25          System.out.println(person.getName());
26      }
27      //方法二:通过有参的构造方法创建实例对象
28      public static void fun2(Class<?> cls) throws
NoSuchMethodException, SecurityException,
InstantiationException, IllegalAccessException,
IllegalArgumentException, InvocationTargetException {
29          //先得到有参的构造方法
30          //这里要写参数的字节码文件对象形式          所有的类型都
有字节码文件对象
31          //相当于  public Person(String name, int age)
32          Constructor constructor =
cls.getConstructor(String.class,int.class);
33          Object object =
constructor.newInstance("bingbing",18);
34          System.out.println(object);
35      }
36  }
37
38
```

4.2.4. 给属性赋值(通过从属性中提取出来的类--Field)

```
1  package com.qf.refect;
2  //3.给属性赋值(通过从属性中提取出来的类--Field)
3
4  import java.lang.reflect.Field;
5
6  public class Demo3 {
7      public static void main(String[] args) throws
   ClassNotFoundException, InstantiationException,
   IllegalAccessException, NoSuchFieldException,
   SecurityException {
8          Person person = new Person();
9          //person.name = "bingbing";
10
11         //使用反射实现
12         //1.获取字节码文件对象
13         Class<?> class1 =
   Class.forName("com.qf.refect.Person");
14
15         //2.获取实例对象
16         Object object = class1.newInstance();
17
18         //3.调用属性
19         //注意:如果想使用getField,name属性必须是public的
20         //Field field1 = class1.getField("name");
21         //如果name是私有的,我们可以这样做    ,忽略权限
22         Field field1 = class1.getDeclaredField("name");
23         field1.setAccessible(true);
24         //赋值
25         //第一个参数:关联的具体对象
```

```
        //第二个参数:赋的值
        field1.set(object, "bing");

        System.out.println(field1.get(object));
    }
}


```

4.2.5. 调用方法(通过从方法中提取出来的类--Method)

```
package com.qf.refect;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

//4.调用方法(通过从方法中提取出来的类--Method)
//invoke的返回值就是内部对应方法的返回值,如果内部方法没有返回值这里就返回null
public class Demo4 {
    public static void main(String[] args) throws ClassNotFoundException, InstantiationException, IllegalAccessException, NoSuchMethodException, SecurityException, IllegalArgumentException, InvocationTargetException {
        //使用反射实现
        //1.获取字节码文件对象
        Class<?> class1 = Class.forName("com.qf.refect.Person");

        //调用非静态无参
```

```
16          fun1(class1);
17          //调用非静态有参
18          //fun2(class1);
19          //调用静态有参
20          //fun3(class1);
21      }
22      //调用非静态无参
23      public static void fun1(Class<?> cla) throws
InstantiationException, IllegalAccessException,
NoSuchMethodException, SecurityException,
IllegalArgumentException, InvocationTargetException {
24          //2.获取实例对象
25          Object object = cla.newInstance();
26          //3.通过反射得到方法
27          Method method = cla.getMethod("show");
28          //4.调用方法,通过调用invoke方法实现
29          Object obj = method.invoke(object);
30          System.out.println("obj:"+obj);//3
31      }
32      //调用非静态有参
33      public static void fun2(Class<?> cla) throws
NoSuchMethodException, SecurityException,
InstantiationException, IllegalAccessException,
IllegalArgumentException, InvocationTargetException {
34          //2.先得到有参的构造方法
35          Constructor<?> constructor =
cla.getConstructor(String.class,int.class);
36          Object object =
constructor.newInstance("bingibn",10);
37
38          //3.通过反射得到方法
39          Method method =
cla.getMethod("callPhone",String.class);
```

```
40          //4.调用方法,通过调用invoke方法实现
41          Object obj = method.invoke(object,"110");
42          System.out.println("obj:"+obj);//null
43      }
44      //调用静态有参
45      public static void fun3(Class<?> cla) throws
    NoSuchMethodException, SecurityException,
    IllegalAccessException, IllegalArgumentException,
    InvocationTargetException {
46          //3.通过反射得到方法
47          Method method = cla.getMethod("run",int.class);
48          //4.调用方法,通过调用invoke方法实现
49          method.invoke(null,11);
50      }
51  }
52
```