以下转载自：

<http://baike.baidu.com/link?url=huWIUQEsGxYIVHusfP2WWHeGLfOZlbmOW9VDUzIc72GPOKHG6Jqj8mJ1_KjVGmcjbsyAWnelHycxUqc9M3Nep_>

Liskov于1987年提出了一个关于继承的原则“Inheritance should ensure that any property proved about supertype objects also holds for subtype objects.”——“继承必须确保超类所拥有的性质在子类中仍然成立。”也就是说，当一个子类的实例应该能够替换任何其超类的实例时，它们之间才具有is-A关系。

OO(Object Oriented)的继承与日常生活中的继承的本质区别。举一个例子：生物学的分类体系中把企鹅归属为鸟类。我们模仿这个体系，设计出这样的类和关系。

类“鸟”中有个方法fly，企鹅自然也继承了这个方法，可是企鹅不能飞啊，于是，我们在企鹅的类中覆盖了fly方法，告诉方法的调用者：企鹅是不会飞的。这完全符合常理。但是，这违反了LSP，企鹅是鸟的子类，可是企鹅却不能飞！需要注意的是，此处的“鸟”已经不再是生物学中的鸟了，它是软件中的一个类、一个抽象。

有人会说，企鹅不能飞很正常啊，而且这样编写代码也能正常编译，只要在使用这个类的客户代码中加一句判断就行了。但是，这就是问题所在！首先，客户代码和“企鹅”的代码很有可能不是同时设计的，在当今软件外包一层又一层的开发模式下，你甚至根本不知道两个模块的原产地是哪里，也就谈不上去修改客户代码了。客户程序很可能是遗留系统的一部分，很可能已经不再维护，如果因为设计出这么一个“企鹅”而导致必须修改客户代码，谁应该承担这部分责任呢？（大概是上帝吧，谁叫他让“企鹅”不能飞的。^\_^）“修改客户代码”直接违反了OCP，这就是OCP的重要性。违反LSP将使既有的设计不能封闭！

注：OCP(开闭原则)

遵循开闭原则设计出的模块具有两个主要特征：[1]

（1）对于扩展是开放的（Open for extension）。这意味着模块的行为是可以扩展的。当应用的需求改变时，我们可以对模块进行扩展，使其具有满足那些改变的新行为。也就是说，我们可以改变模块的功能。

（2）对于修改是关闭的（Closed for modification）。对模块行为进行扩展时，不必改动模块的源代码或者二进制代码。模块的二进制可执行版本，无论是可链接的库、DLL或者.EXE文件，都无需改动。

为了满足LSP，当存在继承关系时，子类中方法的前置条件必须与超类中被覆盖的方法的前置条件相同或者更宽松；而子类中方法的后置条件必须与超类中被覆盖的方法的后置条件相同或者更为严格。

一些OO语言中的特性能够说明这一问题：

继承并且覆盖超类方法的时候，子类中的方法的可见性必须等于或者大于超类中的方法的可见性，子类中的方法所抛出的受检异常只能是超类中对应方法所抛出的受检异常的子类。

public class SuperClass{

public void methodA() throws Exception{}

}

public class SubClassA extends SuperClass{

//this overriding is illegal.

private void methodA() throws IOException{}

}

public class SubClassB extends SuperClass{

//this overriding is OK.

public void methodA() throws FileNotFoundException{}

}

从Java5开始，子类中的方法的返回值也可以是对应的超类方法的返回值的子类。这叫做“协变”(Covariant)

public class SuperClass {

public Number caculate(){

return null;

}

}

public class SubClass extends SuperClass{

//only compiles in Java 5 or later.

public Integer caculate(){

return null;

}

}

可以看出，以上这些特性都非常好地遵循了LSP。

以下转载自：

<http://www.cnblogs.com/sunwei2012/archive/2010/03/10/1682415.html>

里氏替换原则为良好的继承定义了一个规范，一句简单的定义包含了四层含义：

（1）子类必须完全实现父类的方法

我们在做系统设计时，经常会定义一个接口或抽象类，然后编码实现，调用类则直接传入接口或抽象类，其实这里已经使用了里氏替换原则。我们举个例子来说明这个原则，大家都打过CS吧，非常经典的FPS类游戏，我们来描述一下里面用到的枪，类图如图2-1所示。

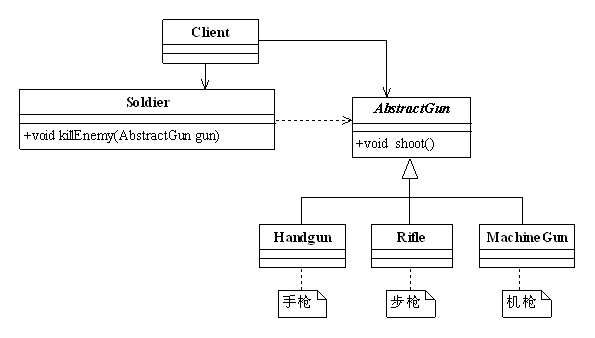


图2-1 CS游戏中的枪械类图

枪的主要职责是射击，如何射击在各个具体的子类中定义，手 枪是单发射程比较近，步枪威力大射程远，机枪用于扫射。在士兵类中定义了一个方法killEnemy，使用枪来杀敌人，具体使用什么枪来杀敌人，调用的时候才知道，AbstractGun类的源程序如代码清单2-1所示。

代码清单2-1 枪支的抽象类

1 public abstract class AbstractGun {

2

3 //枪用来干什么的？射击杀戮！

4

5 public abstract void shoot();

6

7 }

手枪、步枪、机枪的实现类如代码清单2-2所示。

代码清单2-2 手 枪、步枪、机枪的实现类

01 public class Handgun extends AbstractGun {

02

03 //手枪的特点是携带方便，射程短

04

05 @Override

06

07 public void shoot() {

08

09 System.out.println("手枪射击...");

10

11 }

12

13 }

14

15 public class Rifle extends AbstractGun{

16

17 //步枪的特点是射程远，威力大

18

19 public void shoot(){

20

21 System.out.println("步枪射击...");

22

23 }

24

25 }

26

27 public class MachineGun extends AbstractGun{

28

29 public void shoot(){

30

31 System.out.println("机枪扫射...");

32

33 }

34

35 }

有了枪械，还要有能够使用这些枪械的士兵，其源程序如代码清单2-3所示。

代码清单2-3 士兵的实现类

01 public class Soldier {

02

03 public void killEnemy(AbstractGun gun){

04

05 System.out.println("士兵开始杀人...");

06

07 gun.shoot();

08

09 }

10

11 }

注意粗体部分，定义士兵使用枪来杀敌，但是这把枪是时抽象的，具体是手 枪还是步枪需要在上战场前（也就是场景中）前确定。

场景类Client的源代码如代码清单2-4所示。

代码清单2-4 场景类

01 public class Client {

02

03 public static void main(String[] args) {

04

05 //产生三毛这个士兵

06

07 Soldier sanMao = new Soldier();

08

09 sanMao.killEnemy(new Rifle());

10

11 }

12

13 }

有人，有枪，也有场景，运行结果如下所示。

士兵开始杀人...

步枪射击...

在这个程序中，我们给三毛这个士兵一把步枪，然后就开始杀敌了，如果三毛要使用机枪当然也可以,直接把sanMao.killEnemy(new Rifle())修改为 sanMao.killEnemy(new MachineGun())就可以了，在编写程序时Solider士兵类根本就不用知道是哪个型号的枪（子类）被传入。

注意，在类中调用其他类时务必要使用父类或接口，如果不能使用父类或接口，则说明类的设计已经违背了LSP原则。

我们再来想一想，如果我们有一个玩具手枪，该如何定义呢？我们先在类图2-1上增加一个类ToyGun，然后继承于AbstractGun类，修改后的类图如图2-2所示。

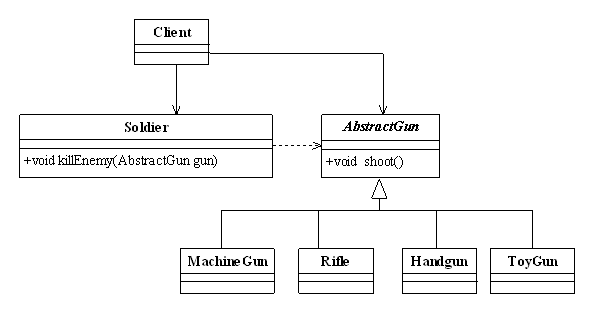


图2-2 枪械类图

首先我们想，玩具枪是不能用来射击的，杀不死人的，这个不应该写在shoot方法中。新增加的ToyGun的源代码如代码清单2-5所示。

代码清单2-5 玩具枪源代码

01 public class ToyGun extends AbstractGun {

02

03 //玩具枪是不能射击的，但是编译器又要求实现这个方法，怎么办？虚构一个呗！

04

05 @Override

06

07 public void shoot() {

08

09 //玩具枪不能射击，这个方法就不实现了

10

11 }

12

13 }

由于引入了新的子类，场景类中也使用了该类，Client稍作修改，源代码如代码清单2-6所示。

代码清单2-6 玩具枪源代码

01 public class Client {

02

03 public static void main(String[] args) {

04

05 //产生三毛这个士兵

06

07 Soldier sanMao = new Soldier();

08

09 sanMao.killEnemy(new ToyGun());

10

11 }

12

13 }

修改了黑体部分，把玩具枪传递给三毛用来杀敌，代码运行结果如下所示：

士兵开始杀人...

坏了，士兵拿着玩具枪来杀人，射不出子弹呀！如果在CS游戏中有这种事情发生，那你就等着被人爆头吧，然后看着自己凄凉的倒地。在这种情况下，我们发现业务调用类已经出现了问题，正常的业务逻辑已经不能运行，那怎么办？好办，有两种解决办法：

Number one：在Soldier类中增加instanceof的判断，如果是玩具枪，就不用来杀敌人。这个方法可以解决问题，但是你要知道，在程序中，每增加一个类，所有与这个父类有关系的类都必须修改，你觉得可行吗？如果你的产品出现了这个问题，因为修正了这样一个Bug，就要求所有与这个父类有关系的类都增加一个判断，客户非跳起来跟你干架不可！你还想要你的客户忠诚你吗？显然，这个方案被否定了。

Number two：ToyGun脱离继承，建立一个独立的父类，为了做到代码可以复用，可以与AbastractGun建立关联委托关系，如图2-3所示。

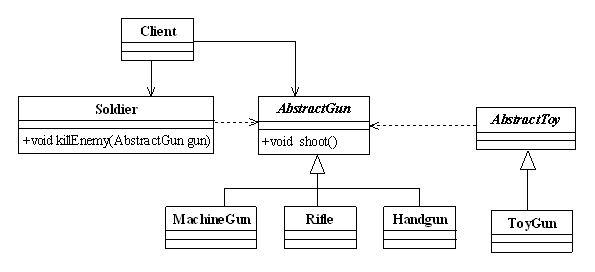


图2-3 玩具枪与真实枪分离的类图

例如，可以在AbstractToy中声明声音、形状都委托给AbstractGun处理，仿真枪嘛，形状和声音都要和真实的枪一样了，然后两个基类下的子类自由延展，互不影响。

在Java的基础知识中，每位老师都会讲继承，Java的三大特征嘛，继承、封装、多态。继承就是告诉你拥有父类的方法和属性，然后你就可以重写父类的方法。按照继承原则，我们上面的玩具枪继承AbstractGun是绝对没有问题的，玩具枪也是枪嘛，但是在具体应用场景中就要考虑下面这个问题了：子类是否能够完整地实现父类的业务，否则就会出现像上面的拿枪杀敌人时却发现是把玩具枪的笑话。

注意，如果子类不能完整地实现父类的方法，或者父类的某些方法在子类中已经发生“畸变”，则建议断开父子继承关系，采用依赖、聚集、组合等关系代替继承。

（2）子类可以有自己的个性

子类当然可以有自己的行为和外观了，也就是方法和属性，那这里为什么要再提呢？是因为里氏替换原则可以正着用，但是不能反过来用。在子类出现的地方，父类未必就可以胜任。还是以刚才的关于枪械的例子为例，步枪有几个比较“响亮”的型号，比如AK47、G3狙击步枪等，把这两个型号的枪引入后的Rifle子类图如图2-4所示。

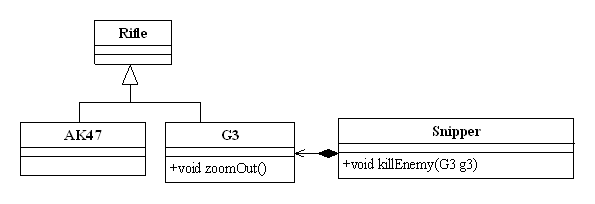


图2-4 增加AK47和G3后的Rifle子类图

很简单，G3继承了Rifle类，狙击手(Snipper)则直接使用G3狙击步枪，源代码如代码清单2-7所示。

代码清单2-7 G3狙击枪源码代码

01 public class G3 extends Rifle {

02

03 //狙击枪都是携带一个精准的望远镜

04

05 public void zoomOut(){

06

07 System.out.println("通过望远镜查看敌人...");

08

09 }

10

11 public void shoot(){

12

13 System.out.println("G3射击...");

14

15 }

16

17 }

有狙击枪就有狙击手，狙击手类的源代码如代码清单2-8所示。

代码清单2-8 G3狙击手类的源码代码

01 public class Snipper {

02

03 public void killEnemy(G3 g3){

04

05 //首先看看敌人的情况，别杀死敌人，自己也被人干掉

06

07 g3.zoomOut();

08

09 //开始射击

10

11 g3.shoot();

12

13 }

14

15 }

狙击手使用狙击枪来杀死敌人，业务场景Client类的源代码如代码清单2-9所示。

代码清单2-9 狙击手使用G3杀死敌人

01 public class Client {

02

03 public static void main(String[] args) {

04

05 //产生三毛这个狙击手

06

07 Snipper sanMao = new Snipper();

08

09 sanMao.killEnemy(new G3());

10

11 }

12

13 }

狙击手使用G3杀死敌人，运行结果如下所示：

通过望远镜查看敌人...

G3射击...

在这里系统直接调用了子类，一个狙击手是很依赖枪支的，别说换一个型号的枪了，就是换一个同型号的枪也会影响射击，所以这里就直接把子类传递了进来。这个时候，我们能不能直接使用父类传递进来呢？修改一下Client类，如代码清单2-10所示。

代码清单2-10 使用父类作为参数

01 public class Client {

02

03 public static void main(String[] args) {

04

05 //产生三毛这个狙击手

06

07 Snipper sanMao = new Snipper();

08

09 Rifle rifle = new Rifle();

10

11 sanMao.killEnemy((G3)rifle);

12

13 }

14

15 }

显示是不行的，会在运行期抛出java.lang.ClassCastException异常，这也是大家经常说的向下转型（downcast）是不安全的，从里氏替换原则来看，就是有子类出现的地方父类未必就可以出现。

（3）覆盖或实现父类的方法时输入参数可以被放大

方法中的输入参数称为前置条件，这是什么意思呢？大家做过Web Service开发就应该知道有一个“契约优先”的原则，也就是先定义出WSDL接口，制定好双方的开发协议，然后再各自实现。里氏替换原则也要求制定一个契约，就是父类或接口，这种设计方法也叫做Design by Contract，契约设计，是与里氏替换原则融合在一起的。契约制定了，也就同时制定了前置条件和后置条件，前置条件就是你要让我执行，就必须满足我的条件；后置条件就是我执行完了需要反馈，标准是什么。这个比较难理解，我们来看一个例子，我们先定义个Father类，如代码清单2-11所示。

代码清单2-11 Father类源代码

01 public class Father {

02

03 public Collection doSomething(HashMap map){

04

05 System.out.println("父类被执行...");

06

07 return map.values();

08

09 }

10

11 }

这个类非常简单，就是把HashMap转换为Collection集合类型，然后再定义一个子类，源代码如代码清单2-12所示。

代码清单2-12 子类源代码

01 public class Son extends Father {

02

03 //放大输入参数类型

04

05 public Collection doSomething(Map map){

06

07 System.out.println("子类被执行...");

08

09 return map.values();

10

11 }

12

13 }

请注意粗体部分，与父类的方法名相同，但又不是覆写（Override）父类的方法。你加个@Override试试看，会报错的，为什么呢？方法名虽然相同，但方法的输入参数不同，就不是覆写，那这是什么呢？是重载（Overload）！不用大惊小怪的，不在一个类就不能是重载了？继承是什么意思，子类拥有父类的所有属性和方法，方法名相同，输入参数类型又不相同，当然是重载了。父类和子类都已经声明了，场景类的调用如代码清单2-13所示。

代码清单2-13 场景类源代码

01 public class Client {

02

03 public static void invoker(){

04

05 //父类存在的地方，子类就应该能够存在

06

07 Father f = new Father();

08

09 HashMap map = new HashMap();

10

11 f.doSomething(map);

12

13 }

14

15 public static void main(String[] args) {

16

17 invoker();

18

19 }

20

21 }

代码运行后的结果如下所示：

父类被执行...

根据里氏替换原则，父类出现的地方子类就可以出现，我们把上面的粗体部分修改为子类，如下代码清单2-14所示。

代码清单2-14 子类替换父类后的源代码

01 public class Client {

02

03 public static void invoker(){

04

05 //父类存在的地方，子类就应该能够存在

06

07 Son f =new Son();

08

09 HashMap map = new HashMap();

10

11 f.doSomething(map);

12

13 }

14

15 public static void main(String[] args) {

16

17 invoker();

18

19 }

20

21 }

运行结果还是一样，看明白是怎么回事了吗？父类方法的输入参数是HashMap类型，子类的输入参数是Map类型，也就是说子类的输入参数类型的范围扩大了，子类代替父类传递到调用者中，子类的方法永远都不会被执行。这是正确的，如果你想让子类的方法运行，就必须覆写父类的方法。大家可以这样想，在一个Invoker类中关联了一个父类，调用了一个父类的方法，子类可以覆写这个方法，也可以重载这个方法，前提是要扩大这个前置条件，就是输入参数的类型大于父类的类型覆盖范围。这样说可能比较理难理解，我们再反过来想一下，如果Father类的输入参数类型大于子类的输入参数类型，会出现什么问题呢？会出现父类存在的地方，子类就未必可以存在，因为一旦把子类作为参数传入，调用者就很可能进入子类的方法范畴。我们把上面的例子修改一下，扩大父类的前置条件，源代码如代码清单2-15所示。

代码清单2-15 父类的前置条件较大

01 public class Father {

02

03 public Collection doSomething(Map map){

04

05 System.out.println("Map 转Collection被执行");

06

07 return map.values();

08

09 }

10

11 }

把父类的前置条件修改为Map类型，我们再修改一下子类方法的输入参数，相对父类缩小输入参数的类型范围，也就是缩小前置条件，源代码如代码清单2-16所示。

代码清单2-16 子类的前置条件较小

01 public class Son extends Father {

02

03 //缩小输入参数范围

04

05 public Collection doSomething(HashMap map){

06

07 System.out.println("HashMap转Collection被执行...");

08

09 return map.values();

10

11 }

12

13 }

在父类的前置条件大于子类的前置条件的情况下，业务场景的源代码如代码清单2-17所示。

代码清单2-17 子类的前置条件较小

01 public class Client {

02

03 public static void invoker(){

04

05 //有父类的地方就有子类

06

07 Father f= new Father();

08

09 HashMap map = new HashMap();

10

11 f.doSomething(map);

12

13 }

14

15 public static void main(String[] args) {

16

17 invoker();

18

19 }

20

21 }

代码运行结果如下所示：

父类被执行...

那我们再把里氏替换原则引入进来会有什么问题？有父类的地方子类就可以使用，好，我们把这个Client类修改一下，源代码如代码清单2-18所示。

代码清单2-18 采用里氏替换原则后的业务场景类

01 public class Client {

02

03 public static void invoker(){

04

05 //有父类的地方就有子类

06

07 Son f =new Son();

08

09 HashMap map = new HashMap();

10

11 f.doSomething(map);

12

13 }

14

15 public static void main(String[] args) {

16

17 invoker();

18

19 }

20

21 }

代码运行后的结果如下所示：

子类被执行...

完蛋了吧？！子类在没有覆写父类的方法的前提下，子类方法被执行了，这会引起业务逻辑混乱，因为在实际应用中父类一般都是抽象类，子类是实现类，你传递一个这样的实现类就会“歪曲”了父类的意图，引起一堆意想不到的业务逻辑混乱，所以子类中方法的前置条件必须与超类中被覆盖的方法的前置条件相同或者更宽松。

覆盖或实现父类的方法时输出结果可以被缩小

这是什么意思呢，父类的一个方法的返回值是一个类型T，子类的相同方法（重载或覆写）的返回值为S，那么里氏替换原则就要求S必须小于等于T，也就是说要么S和T是同一个类型，要么S是T的子类，为什么呢？分两种情况，如果是覆写，父类和子类的同名方法的输入参数是相同的，两个方法的范围值S小于等于T，这是覆写的要求，这才是重中之重，子类覆写父类的方法，天经地义。如果是重载，则要求方法的输入参数类型或数量不相同，在里氏替换原则要求下，就是子类的输入参数大于或等于父类的输入参数，也就是说你写的这个方法是不会被调用到的，参考上面讲的前置条件。

采用里氏替换原则的目的就是增强程序的健壮性，版本升级时也可以保持非常好的兼容性。即使增加子类，原有的子类还可以继续运行。在实际项目中，每个子类对应不同的业务含义，使用父类作为参数，传递不同的子类完成不同的业务逻辑，非常完美！