

# 设计模式 策略模式 以角色游戏为背景

原创

鸿洋\_

于 2014-04-19 17:03:33 发布

29503

收藏 27

版权

分类专栏:


【Java 设计模式】

设计模式融入生活

文章标签:

设计模式

策略模式



【Java 设计模式】 同时关注

67 订阅 10 篇文章

订阅专栏

今天不想写代码，给大家带来一篇 **设计模式** 的文章，帮助大家可以把系统组织成容易了解、容易维护、具有弹性的架构。

先来看看策略模式的定义：

策略模式（Strategy Pattern）：定义了算法族，分别封装起来，让它们之间可相互替换，此模式让算法的变化独立于使用算法的客户。

好了，对于定义，肯定不是一眼就能看明白的，不然这篇文章就收尾了，对于定于大家简单扫一眼，知道个大概，然后继续读下面的文章，读完以后再来回味，效果嘎嘣脆。大家应该都玩过武侠角色游戏，下面我就以角色游戏为背景，为大家介绍：假设公司需要做一款武侠游戏，我们就是负责游戏的角色模块，需求是这样的：每个角色对应一个名字，每类角色对应一种样子，每个角色拥有一个逃跑、攻击、防御的技能。

初步的代码：

```
1 package com.zhy.bean;
2
3 /**
4  * 游戏的角色超类
5  *
6  * @author zhy
7  *
8  */
9 public abstract class Role
10 {
11     protected String name;
12
13     protected abstract void display();
14
15     protected abstract void run();
16
17     protected abstract void attack();
18
19     protected abstract void defend();
20 }
21 }
```

```
1 package com.zhy.bean;
2
3 public class RoleA extends Role
4 {
5     public RoleA(String name)
6     {
7         this.name = name;
8     }
9
10    @Override
11    protected void display()
12    {
13        System.out.println("样子1");
14    }
15
16    @Override
17    protected void run()
```

```

18         {19 |         System.out.println("金蝉脱壳");
19     }
20
21
22     @Override
23     protected void attack()
24     {
25         System.out.println("降龙十八掌");
26     }
27
28     @Override
29     protected void defend()
30     {
31         System.out.println("铁头功");
32     }
33
34 }

```

没几分钟，你写好了上面的代码，觉得已经充分发挥了OO的思想，正在窃喜，这时候项目经理说，再添加两个角色

RoleB(样子2，降龙十八掌，铁布衫，金蝉脱壳)。

RoleC(样子1，拥有九阳神功，铁布衫，烟雾弹)。

于是你觉得没问题，开始写代码，继续集成Role，写成下面的代码：

```

1 package com.zhy.bean;
2
3 public class RoleB extends Role
4 {
5     public RoleB(String name)
6     {
7         this.name = name;
8     }
9
10    @Override
11    protected void display()
12    {
13        System.out.println("样子2");
14    }
15
16    @Override
17    protected void run()
18    {
19        System.out.println("金蝉脱壳");//从RoleA中拷贝
20    }
21
22    @Override
23    protected void attack()
24    {
25        System.out.println("降龙十八掌");//从RoleA中拷贝
26    }
27
28    @Override
29    protected void defend()
30    {
31        System.out.println("铁布衫");
32    }
33
34 }

```

```

1 package com.zhy.bean;
2
3 public class RoleC extends Role
4 {

```

```
5 | public RoleC(String name) 6 | {
7 |     this.name = name;
8 | }
9 |
10 | @Override
11 | protected void display()
12 | {
13 |     System.out.println("样子1");//从RoleA中拷贝
14 | }
15 |
16 | @Override
17 | protected void run()
18 | {
19 |     System.out.println("烟雾弹");
20 | }
21 |
22 | @Override
23 | protected void attack()
24 | {
25 |     System.out.println("九阳神功");
26 | }
27 |
28 | @Override
29 | protected void defend()
30 | {
31 |     System.out.println("铁布衫");//从B中拷贝
32 | }
33 |
34 | }
```

写完之后，你自己似乎没有当初那么自信了，你发现代码中已经存在相当多重复的代码，需要考虑重新设计架构了。于是你想，要不把每个技能都写成接口，有什么技能的角色实现什么接口，简单一想，觉得这想法高大尚啊，但是实现起来会发现，接口并不能实现代码的复用，每个实现接口的类，还是必须写自己写实现。于是，we need change！遵循设计的原则，找出应用中可能需要变化的部分，把它们独立出来，不要和那些不需要变化的代码混在一起。我们发现，对于每个角色的display，attack，defend，run都是有可能变化的，于是我们必须把这写独立出来。再根据另一个设计原则：针对接口（超类型）编程，而不是针对实现编程，于是我们把代码改造成这样：

```
1 | package com.zhy.bean;
2 |
3 | public interface IAttackBehavior
4 | {
5 |     void attack();
6 | }
```

```
1 | package com.zhy.bean;
2 |
3 | public interface IDefendBehavior
4 | {
5 |     void defend();
6 | }
```

```
1 | package com.zhy.bean;
2 |
3 | public interface IDisplayBehavior
4 | {
5 |     void display();
6 | }
```

```

1 package com.zhy.bean;
2
3 public class AttackJY implements IAttackBehavior
4 {
5
6     @Override
7     public void attack()
8     {
9         System.out.println("九阳神功! ");
10    }
11
12 }

```

```

1 package com.zhy.bean;
2
3 public class DefendTBS implements IDefendBehavior
4 {
5
6     @Override
7     public void defend()
8     {
9         System.out.println("铁布衫");
10    }
11
12 }

```

```

1 package com.zhy.bean;
2
3 public class RunJCTQ implements IRunBehavior
4 {
5
6     @Override
7     public void run()
8     {
9         System.out.println("金蝉脱壳");
10    }
11
12 }

```

这时候需要对Role的代码做出改变：

```

1 package com.zhy.bean;
2
3 /**
4  * 游戏的角色超类
5  *
6  * @author zhy
7  *
8  */
9 public abstract class Role
10 {
11     protected String name;
12
13     protected IDefendBehavior defendBehavior;
14     protected IDisplayBehavior displayBehavior;
15     protected IRunBehavior runBehavior;
16     protected IAttackBehavior attackBehavior;
17
18
19     public Role setDefendBehavior(IDefendBehavior defendBehavior)
20     {
21         this.defendBehavior = defendBehavior;
22     }
23 }

```

```
21 |         return this;22 |     }
23 |
24 |
25 |     public Role setDisplayBehavior(IDisplayBehavior displayBehavio
26 |     {26 |         this.displayBehavior = displayBehavior;
27 |         return this;
28 |     }
29 |
30 |     public Role setRunBehavior(IRunBehavior runBehavior)
31 |     {
32 |         this.runBehavior = runBehavior;
33 |         return this;
34 |     }
35 |
36 |
37 |     public Role setAttackBehavior(IAttackBehavior attackBehavior)
38 |     {38 |         this.attackBehavior = attackBehavior;
39 |         return this;
40 |     }
41 |
42 |     protected void display()
43 |     {
44 |         displayBehavior.display();
45 |     }
46 |
47 |     protected void run()
48 |     {
49 |         runBehavior.run();
50 |     }
51 |
52 |     protected void attack()
53 |     {
54 |         attackBehavior.attack();
55 |     }
56 |
57 |     protected void defend()
58 |     {
59 |         defendBehavior.defend();
60 |     }
61 |
62 | }
```

每个角色现在只需要一个name了:

```
1 | package com.zhy.bean;
2 |
3 | public class RoleA extends Role
4 | {
5 |     public RoleA(String name)
6 |     {
7 |         this.name = name;
8 |     }
9 |
10 | }
```

现在我们需要一个金蝉脱壳，降龙十八掌！，铁布衫，样子1的角色A只需要这样:

```
1 | package com.zhy.bean;
2 |
3 | public class Test
4 | {
5 |     public static void main(String[] args)
6 |     {
7 |
```

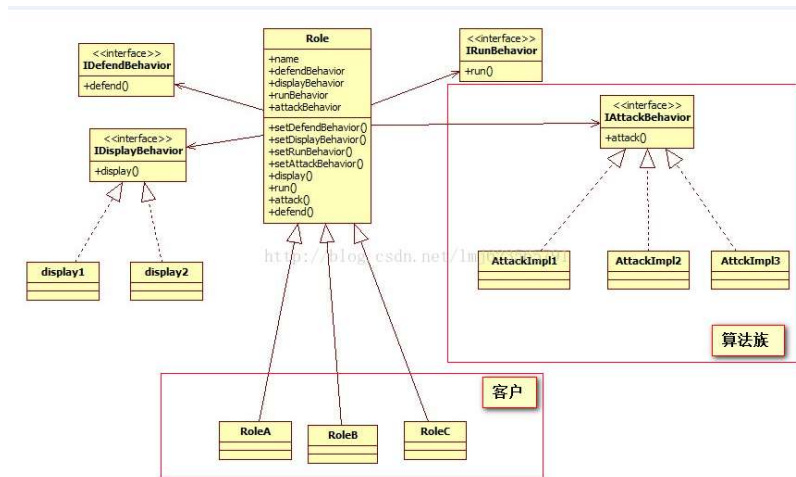
```

8 |         Role roleA = new RoleA("A");
9 |
10 |         roleA.setAttackBehavior(new AttackXL());
11 |         roleA.setDefendBehavior(new DefendTBS());
12 |         roleA.setDisplayBehavior(new DisplayA());
13 |         roleA.setRunBehavior(new RunJCTQ());
14 |         System.out.println(roleA.name + ":");
15 |         roleA.run();
16 |         roleA.attack();
17 |         roleA.defend();
18 |         roleA.display();
19 |     }
20 | }

```

经过我们的修改，现在所有的技能的实现做到了100%的复用，并且随便项目经理需要什么样的角色，对于我们来说只需要动态设置一下技能和展示方式，是不是很完美。恭喜你，现在你已经学会了策略模式，现在我们回到定义，定义上的算法族：其实就是上述例子的技能；定义上的客户：其实就是RoleA，RoleB...；我们已经定义了一个算法族（各种技能），且根据需求可以进行相互替换，算法（各种技能）的实现独立于客户（角色）。现在是不是很好理解策略模式的定义了。

附上一张UML图，方便大家理解：



最后总结一下OO的原则：

- 1、封装变化（把可能变化的代码封装起来）
- 2、多用组合，少用继承（我们使用组合的方式，为客户设置了算法）
- 3、针对接口编程，不针对实现（对于Role类的设计完全的针对角色，和技能的实现没有关系）

[点击此处下载源码](#)

显示推荐内容

觉得还不错? [一键收藏](#) ×



鸿洋\_

关注



80



27



58



专栏目录