

设计模式 模版方法模式 展现程序员的一天

原创 鸿洋_ 于 2014-05-19 19:33:09 发布 20544 收藏 13 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](#) [模版方法模式](#)



【Java 设计模式】 同时关注

67 订阅

10 篇文章

订阅专栏

转载请标明出处: <http://blog.csdn.net/Imj623565791/article/details/26276093>

继续 [设计模式](#) ~ 模版方法模式

老套路, 先看下定义: 定义了一个算法的骨架, 而将一些步骤延迟到 **子类** 中, 模版方法使得子类可以在不改变算法结构的情况下, 重新定义算法的步骤。

简单看下定义, 模版方法定义了一个算法的步骤, 并且允许子类为一个或多个步骤提供实现。定义还算清晰, 下面来个例子展示下本公司的上班情况 (纯属娱乐, 如有雷同, 请对号入座)。简单描述一下: 本公司有程序猿、测试、HR、项目经理等人, 下面使用模版方法模式, 记录下所有人员的上班情况:

首先来个超类, 超类中定义了一个workOneDay方法, 设置为作为算法的骨架:

```
1 package com.zhy.pattern.template;
2
3 public abstract class Worker
4 {
5     protected String name;
6
7     public Worker(String name)
8     {
9         this.name = name;
10    }
11
12    /**
13     * 记录一天的工作
14     */
15    public final void workOneDay()
16    {
17
18        System.out.println("-----work start -----");
19        enterCompany();
20        computerOn();
21        work();
22        computerOff();
23        exitCompany();
24        System.out.println("-----work end -----");
25    }
26
27
28    /**
29     * 工作
30     */
31    public abstract void work();
32
33    /**
34     * 关闭电脑
35     */
36    private void computerOff()
37    {
38        System.out.println(name + "关闭电脑");
39    }
40
41    /**
42     * 打开电脑
43     */
44    private void computerOn()
```

```
45 | {
46 |     System.out.println(name + "打开电脑");
47 | }
48 |
49 | /**
50 |  * 进入公司
51 |  */
52 | public void enterCompany()
53 | {
54 |     System.out.println(name + "进入公司");
55 | }
56 |
57 | /**
58 |  * 离开公司
59 |  */
60 | public void exitCompany()
61 | {
62 |     System.out.println(name + "离开公司");
63 | }
64 |
65 | }
```

定义了一个上班（算法）的骨架，包含以下步骤：

- a、进入公司
- b、打开电脑
- c、上班情况
- d、关闭电脑
- e、离开公司

可以看到，a、b、d、e我们在超类中已经实现，子类仅实现work这个抽象方法，记录每天的上班情况。下面各类屌丝入场：

程序猿：

```
1 | package com.zhy.pattern.template;
2 |
3 | public class ITWorker extends Worker
4 | {
5 |
6 |     public ITWorker(String name)
7 |     {
8 |         super(name);
9 |     }
10 |
11 |     @Override
12 |     public void work()
13 |     {
14 |         System.out.println(name + "写程序-测bug-fix bug");
15 |     }
16 |
17 | }
```

HR：

```
1 | package com.zhy.pattern.template;
2 |
3 | public class HRWorker extends Worker
4 | {
5 |
6 |     public HRWorker(String name)
```

```
7 | { 8 |         super(name);
9 |     }
10 |
11 |     @Override
12 |     public void work()
13 |     {
14 |         System.out.println(name + "看简历-打电话-接电话");
15 |     }
16 |
17 | }
```

测试人员：

```
1 | package com.zhy.pattern.template;
2 |
3 | public class QAWorker extends Worker
4 | {
5 |
6 |     public QAWorker(String name)
7 |     {
8 |         super(name);
9 |     }
10 |
11 |     @Override
12 |     public void work()
13 |     {
14 |         System.out.println(name + "写测试用例-提交bug-写测试用例");
15 |     }
16 |
17 | }
```

项目经理：

```
1 | package com.zhy.pattern.template;
2 |
3 | public class ManagerWorker extends Worker
4 | {
5 |
6 |     public ManagerWorker(String name)
7 |     {
8 |         super(name);
9 |     }
10 |
11 |     @Override
12 |     public void work()
13 |     {
14 |         System.out.println(name + "打dota...");
15 |     }
16 |
17 | }
```

下面我们测试下：

```
1 | package com.zhy.pattern.template;
2 |
3 | public class Test
4 | {
5 |     public static void main(String[] args)
6 |     {
7 |
8 |         Worker it1 = new ITWorker("鸿洋");
```

```

9 |         it1.workOneDay();
10 |
    | Worker it2 = new ITWorker("老张");11 |
    | it2.workOneDay();12 |
    | Worker hr = new HRWorker("迪迪");13 |
    | hr.workOneDay();14 |
    | Worker qa = new QAWorker("老李");15 |
    | qa.workOneDay();16 |
    | Worker pm = new ManagerWorker("坑货");17 |
    | pm.workOneDay();18 |
19 |     }
20 | }

```

输出结果：

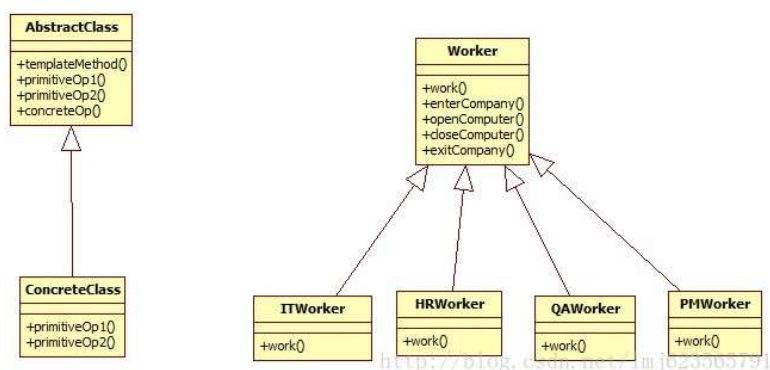
```

1 | -----work start -----
2 | 鸿洋进入公司
3 | 鸿洋打开电脑
4 | 鸿洋写程序-测bug-fix bug
5 | 鸿洋关闭电脑
6 | 鸿洋离开公司
7 | -----work end -----
8 | -----work start -----
9 | 迪迪进入公司
10 | 迪迪打开电脑
11 | 迪迪看简历-打电话-接电话
12 | 迪迪关闭电脑
13 | 迪迪离开公司
14 | -----work end -----
15 | -----work start -----
16 | 老李进入公司
17 | 老李打开电脑
18 | 老李写测试用例-提交bug-写测试用例
19 | 老李关闭电脑
20 | 老李离开公司
21 | -----work end -----
22 | -----work start -----
23 | 坑货进入公司
24 | 坑货打开电脑
25 | 坑货打dota...
26 | 坑货关闭电脑
27 | 坑货离开公司
28 | -----work end -----

```

好了，恭喜你，又学会一个设计模式，模版方法模式。

下面看下模版方法模式类图，和我们程序的类图：



模版方式里面也可以可选的设置钩子，比如现在希望记录程序员离开公司的时间，我们就可以在超类中添加一个钩子：

```

1 public boolean isNeedPrintDate()
2 {
3     return false;
4 }
5 /**
6  * 离开公司
7  */
8 public void exitCompany()
9 {
10     if (isNeedPrintDate())
11     {
12
13         System.out.print(new Date().toLocaleString()+"-->");
14         System.out.println(name + "离开公司");
15     }

```

超类中添加了一个isNeedPrintDate方法，且默认返回false，不打印时间。如果某子类需要调用打印时间，可以复写改钩子方法，返回true，比如，程序猿复写了这个方法：

```

1 package com.zhy.pattern.template;
2
3 public class ITWorker extends Worker
4 {
5
6     public ITWorker(String name)
7     {
8         super(name);
9     }
10
11     @Override
12     public void work()
13     {
14         System.out.println(name + "写程序-测bug-fix bug");
15     }
16
17     @Override
18     public boolean isNeedPrintDate()
19     {
20         return true;
21     }
22
23 }

```

最后再看下测试结果：

```

1 -----work start -----
2 鸿洋进入公司
3 鸿洋打开电脑
4 鸿洋写程序-测bug-fix bug
5 鸿洋关闭电脑
6 2014-5-19 19:17:05-->鸿洋离开公司
7 -----work end -----

```

好了，关于钩子，超类中可提供默认实现或者空实现，子类可覆盖或者不覆盖，具体根据需求来定。

最近恰好，再写一个爬虫程序，用到了模版方法模式，给大家分享下：

需求分析：程序需要对特定的20个网站进行抓取数据；每个网站页面返回的结果数据不同，url不同，参数不同等；但是抓取的过程是一致的。

于是我就这样的设计：

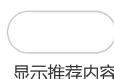
- a、定义一个规则Rule类（包含了：url, params, request_method, 以及返回哪块数据【根据选择器】）
- b、通过Rule进行抓取数据
- c、对数据进行处理

我把上面3个步骤定义了算法的骨架，b为超类实现，a、c由子类实现：

```
1 package com.zhy.pattern.template;
2
3 public abstract class AbsExtractInfo
4 {
5
6
7     /**
8      * 抓取的算法骨架
9      */
10    public void extract()
11    {
12        Rule rule = generateRule() ;
13        List<Element> eles = getInfosByRule(rule);
14        dealResult(eles);
15    }
16
17    /**
18     * 生成一个Rule
19     * @return
20     */
21    public abstract Rule generateRule();
22
23    /**
24     * 抓取的实现
25     * @param rule
26     * @return
27     */
28    private List<Element> getInfosByRule(Rule rule)
29    {
30        // the implements omitted
31    }
32
33    /**
34     * 处理抓取的结果
35     * @param results
36     */
37    public void dealResult(List<Element> results);
38 }
```

其中GenerateRule这个方法，恰好是工厂模式中的抽象方法模式（定义一个创建对象的接口，但由子类决定要实例化的类是哪一个。工厂方法模式把类实例化的过程推迟到子类），如果你忘记了，可以查看[设计模式 工厂模式 从卖肉夹馍说起](#)


好了，就到这里，最后欢迎大家留言。




觉得还不错？ [一键收藏](#)


 鸿洋_

关注

 56



 13



 21



专栏目录