

[首页](#) [资讯](#) [精华](#) [论坛](#) [问答](#) [博客](#) [专栏](#) [群组](#) [更多 ▼](#)  
[您还未登录！](#) [登录](#) [注册](#)

## [滩涂曳尾](#)

- [博客](#)
- [微博](#)
- [相册](#)
- [收藏](#)
- [留言](#)
- [关于我](#)

### [状态模式——State \(更好的实现状态机\)](#)

博客分类：

- [设计模式\(抽象&封装\)](#)

#### 1. 概述

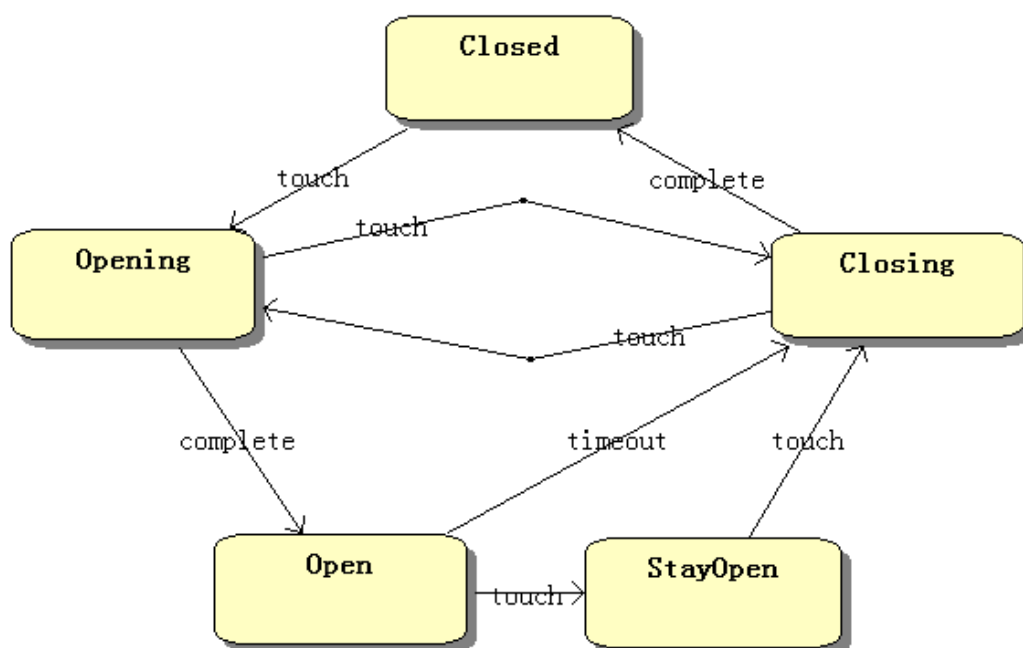
The intent of the STATE pattern is to distribute state-specific logic across classes that represent an object's state.

**STATE** 设计模式的目的 是：将特定状态相关的逻辑分散到一些类的状态类中。

#### 2. 实例

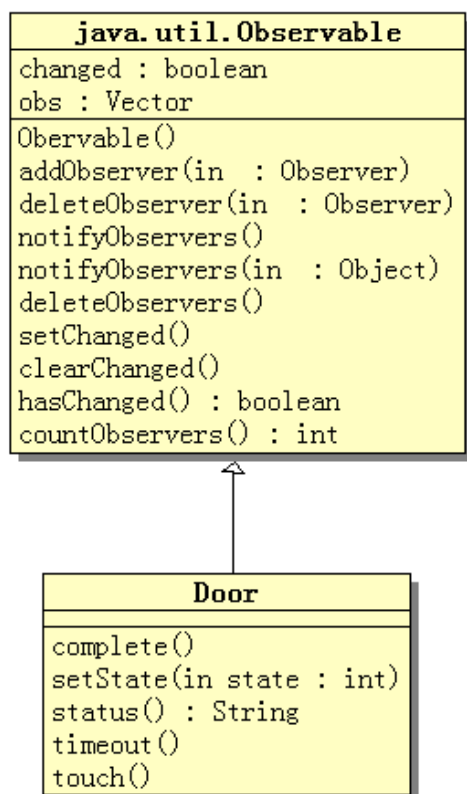
旋转门： Consider the model of the state of a carousel door(revolving door). A carousel is a large, smart rack that accepts material through a doorway and stores the material according to a bar code ID on it. The door operates with a single button. See the state diagram below for some detail.

旋转门的状态图：（状态图细节见 4.）




### 3. 状态模型的两两种实现方法

#### 3.1 方法一： switch



Observable是java.util中的类^^居然以前都不晓得哦，该打！

Door的具体实现如下:

Java代码 

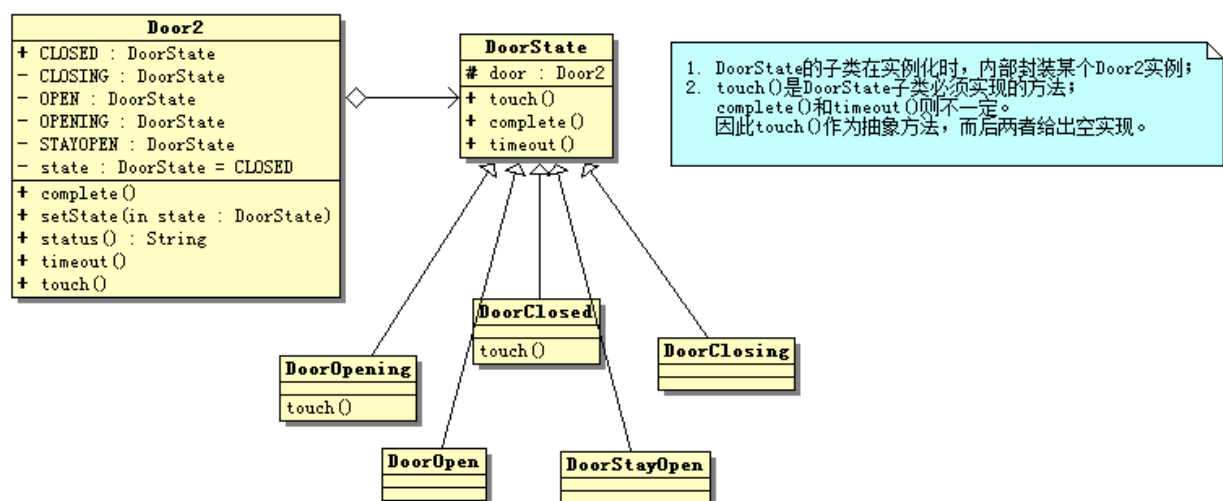
```
1. import java.util.Observable;
2.
3. /**
4.  * This class provides an initial model of a carousel door
5.  * that manages its state without moving state-specific
6.  * logic out to state classes.
7.  */
8. public class Door extends Observable {
9.     public final int CLOSED = -1;
10.    public final int OPENING = -2;
11.    public final int OPEN = -3;
12.    public final int CLOSING = -4;
13.    public final int STAYOPEN = -5;
14.
15.    private int state = CLOSED;
16.
17.    /**
18.     * The carousel user has touched the carousel button. This "one touch"
19.     * button elicits different behaviors, depending on the state of the door.
20.     */
21.    public void touch() {
22.        switch (state)
23.        {
24.            case OPENING:
25.            case STAYOPEN:
26.                setState(CLOSING);
27.                break;
28.            case CLOSING:
29.            case CLOSED:
30.                setState(OPENING);
31.                break;
32.            case OPEN:
33.                setState(STAYOPEN);
34.                break;
35.            default:
36.                throw new Error("can't happen");
37.        }
38.    }
39.
40.    /**
41.     * This is a notification from the mechanical carousel that
42.     * the door finished opening or shutting.
43.     */
44.    public void complete() {
45.        if (state == OPENING)
46.            setState(OPEN);
47.        else if (state == CLOSING)
48.            setState(CLOSED);
49.    }
50.
51.    /**
52.     * This is a notification from the mechanical carousel that the
53.     * door got tired of being open.
54.     */
55.    public void timeout() {
56.        setState(CLOSING);
57.    }
58.
59.    /**
60.     * @return a textual description of the door's state
```

```
61.  */
62.  public String status()
63.  {
64.      switch (state)
65.      {
66.          case OPENING:
67.              return "Opening";
68.          case OPEN:
69.              return "Open";
70.          case CLOSING:
71.              return "Closing";
72.          case STAYOPEN:
73.              return "StayOpen";
74.          default:
75.              return "Closed";
76.      }
77.  }
78.
79.  private void setState(int state)
80.  {
81.      this.state = state;
82.      setChanged();
83.      notifyObservers();
84.  }
85. }
```

但是采用这种实现，有一个缺陷：`state`变量在Door类的实现中浑身扩散，就像癌症一般！

### 3.2 方法二：State Pattern

#### A. 基本的 State Pattern 实现



以上设计方式要求每个状态子类实例内部“hold住”一个Door2实例的引用，这样才能完成Door2实例和它的各个状态实例时间的互相通信。这种设计要求一个状态实例对应一个Door2实例，这样一来，一个

状态实例就只能为一个 Door2 实例服务 (ノ▽ノ) 。

客户端这样调用：

Java代码 ☆

```
1. public static void main(String[] args){
2.     Door2 door=new Door2();
3.
4.     //1. 初始状态
5.     System.out.println(door.status());
6.
7.     //2. 转移到Opening状态
8.     door.touch();
9.     System.out.println(door.status());
10.
11.    //3. 转移到Open状态
12.    door.complete();
13.    System.out.println(door.status());
14.
15.    //4. 转移到Closing状态
16.    door.timeout();
17.    System.out.println(door.status());
18.
19.    //5. 回到Closed状态
20.    door.complete();
21.    System.out.println(door.status());
22. }
```

下面给出Door2类、DoorState抽象类、DoorStayOpen类的实现：

Door2:

Java代码 ☆

```
1. public class Door2 extends Observable {
2.     public final DoorState CLOSED = new DoorClosed(this);
3.     public final DoorState CLOSING = new DoorClosing(this);
4.     public final DoorState OPEN = new DoorOpen(this);
5.     public final DoorState OPENING = new DoorOpening(this);
6.     public final DoorState STAYOPEN = new DoorStayOpen(this);
7.
8.     private DoorState state = CLOSED;
9.
10.    public void touch() {
11.        state.touch();
12.    }
13.
14.    public void complete() {
15.        state.complete();
16.    }
17.
18.    public void timeout() {
19.        state.timeout();
20.    }
21.
22.    public String status() {
23.        return state.status();
24.    }
25. }
```

```
26. protected void setState(DoorState state) {  
27.     this.state = state;  
28.     setChanged();  
29.     notifyObservers();  
30. }
```

DoorState抽象类:

Java代码 ☆

```
1. public abstract class DoorState {  
2.     protected Door2 door;  
3.  
4.     public abstract void touch();  
5.  
6.     public void complete() {  
7.     }  
8.  
9.     public void timeout() {  
10.    }  
11.  
12.    public String status() {  
13.        String s = getClass().getName();  
14.        return s.substring(s.lastIndexOf('.') + 1);  
15.    }  
16.  
17.    public DoorState(Door2 door) {  
18.        this.door = door;  
19.    }  
20. }
```

DoorStayOpen类:

Java代码 ☆

```
1. public class DoorStayOpen extends DoorState {  
2.     public DoorStayOpen(Door2 door) {  
3.         super(door);  
4.     }  
5.  
6.     public void touch() {  
7.         door.setState(door.CLOSING);  
8.     }  
9. }
```

## B. State Pattern 实现 2 —— 让状态实例（DoorState 的子类实例）为多个 Door2 实例服务

子状态 DoorOpen 实现转移时只负责返回下目标状态是什么，将状态转移的 action 留给 Door2 实例自己来做；而不是像“A. 基本的 State Pattern 实现”那样在 DoorOpen 内部保存一个 Door2 实例的引用 door，亲自调用 door.setState(door.STAYOPEN); 来实现状态转移

改进后的关键代码:

Java代码 ☆

```

1. public class DoorOpen extends DoorState{
2.     public DoorState touch(){
3.         return DoorState.STAYOPEN;
4.         // 以前是 door.setState(door.STAYOPEN);
5.     }
6.     ...
7. }
8.
9.
10.
11. public class Door2 extends Observable{
12.     public void touch(){
13.         state=state.touch();
14.         // 以前是 state.touch();
15.         // 即将转移状态的工作留给状态实例来做，事不关己高高挂起
16.     }
17. }

```

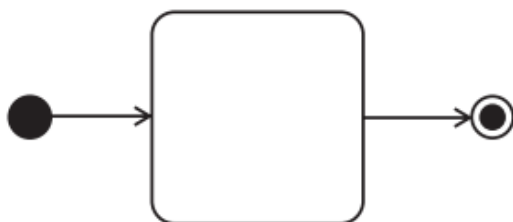
### C. State Pattern 实现 3 ——让状态实例（DoorState 的子类实例）为多个 Door2 实例服务

另一种实现这种效果的方法是：将 Door2 实例作为参数传递给 DoorState 的状态转移方法，而非建立 Composite 的关联关系（将 DoorState 的子类对象作为 Door2 的属性）。

也即，用“Dependency 依赖”（弱依赖，如调用）代替了“Association 关联”（强依赖，如作为属性进行组合）。

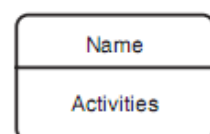
## 4. 状态图细节

何谓 State 状态：Generally speaking, the state of an object depends on the collective value of the object's instance variables. In some cases, most of an object's attributes are fairly static once set, and one attribute is dynamic and plays a prominent role in the class's logic. This attribute may represent the state of the entire object and may even be named state.

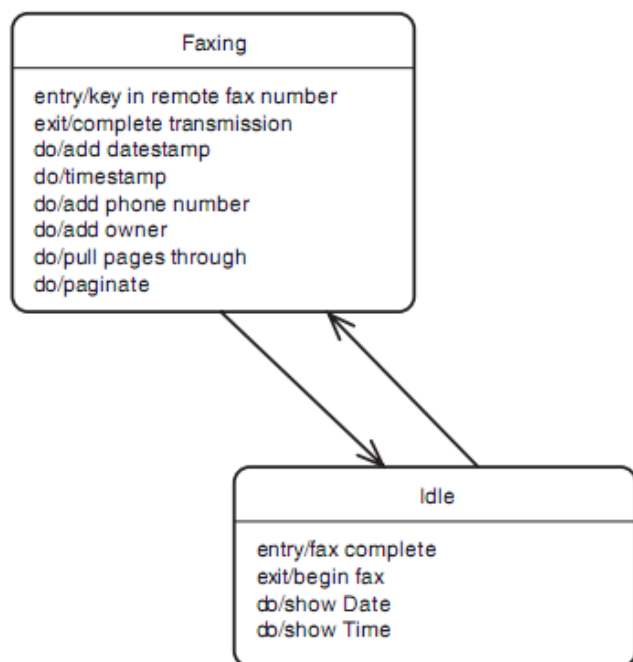


### 4.1 State

You can subdivide a state icon into areas that show the state's name and activities 活动.



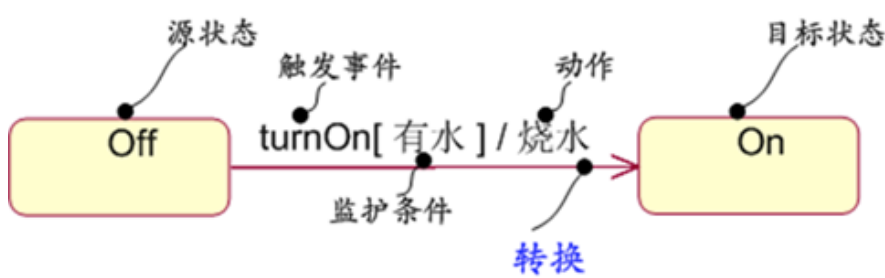
3 frequently used categories of activities are **entry** (what happens when the system enters the state), **exit** (what happens when the system leaves the state), and **do** (what happens while the system is in the state).



#### 4.2 Transition s (Details: Event[Guard Condition]/Action)

You can also add some details to the transition lines. You can indicate an event that causes a transition to occur (a **trigger event**) and the computation (the **action**) that executes and makes the state change happen.

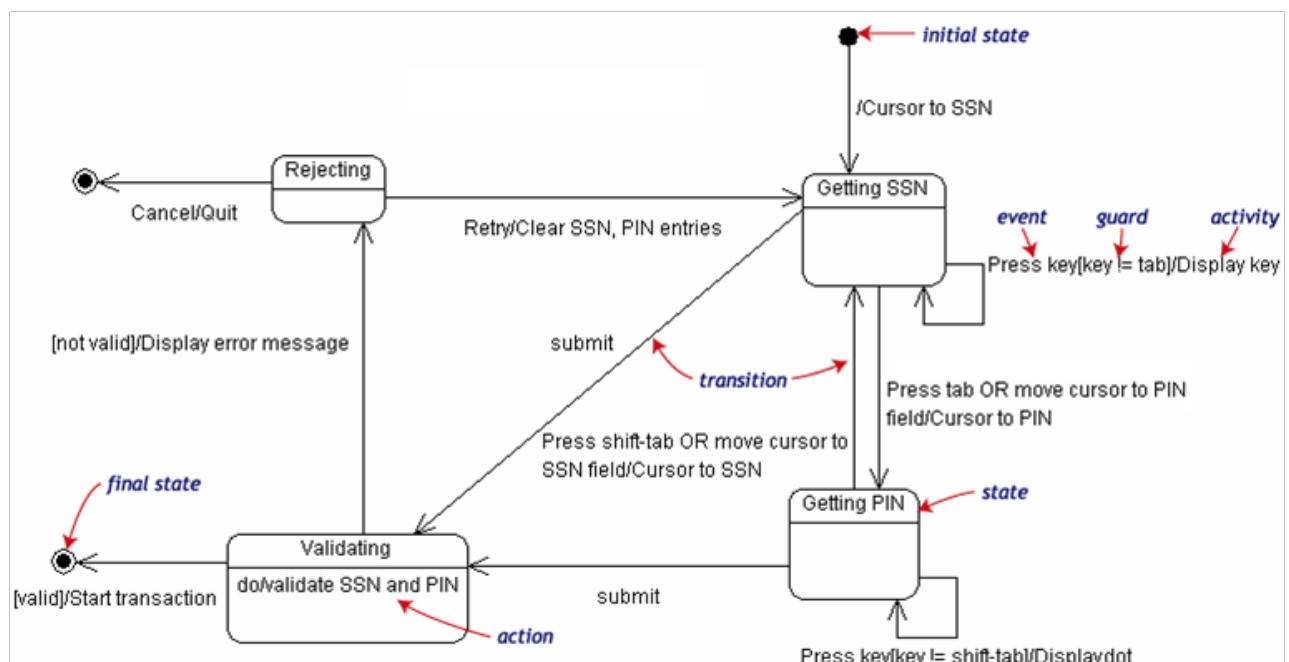
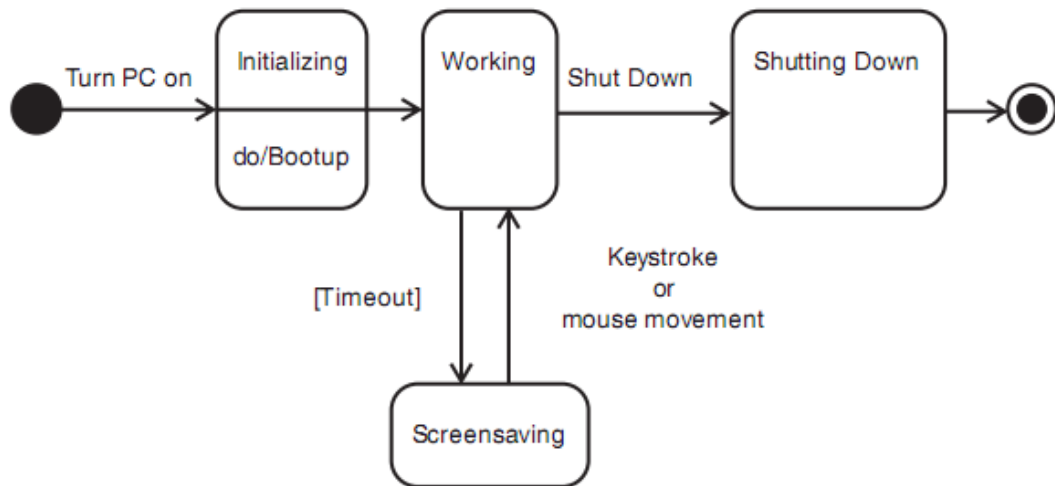
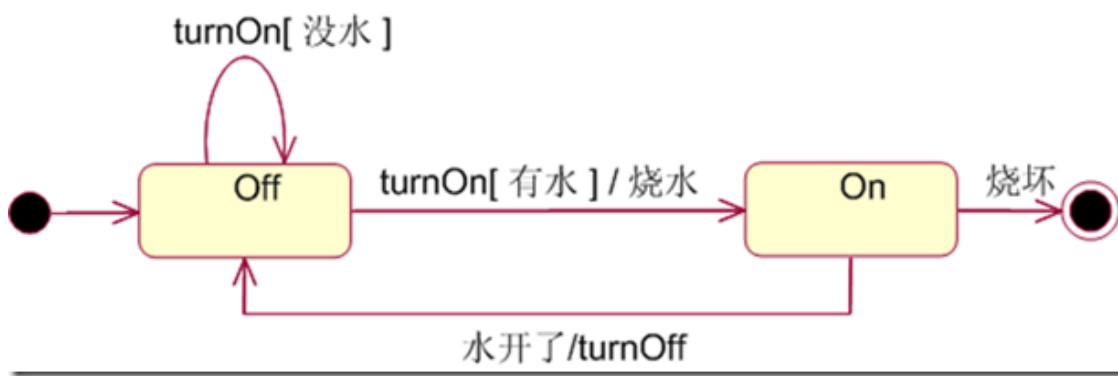
A **guard condition**: when it's met, the transition takes place. 通常将超时作为监护条件， $\therefore$  可以认为此时没有任何 event.



- 源状态 Source State：即受转换影响的状态
- 目标状态 Target State：当转换完成后，对象的状态
- 触发事件 (Trigger) Event：用来为转换定义一个事件，包括调用、改变、信号、时间四类事件
- 监护条件 (Guard Condition)：布尔表达式，决定是否激活转换、
- 动作 (Action)：转换激活时的操作

几个实例：



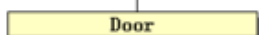




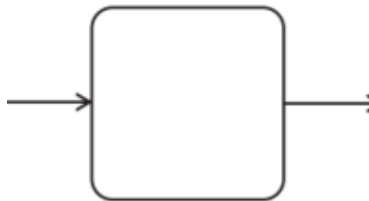
- 
- 大小: 12 KB

```

deleteObserver() : Observer
notifyObservers()
notifyObservers(in : Object)
deleteObservers()
setChanged()
clearChanged()
hasChanged() : boolean
countObservers() : int
    
```



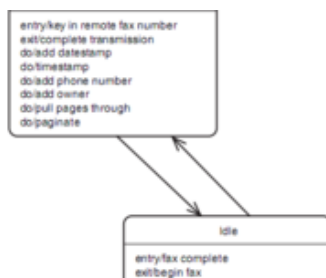
- 
- 大小: 11.4 KB



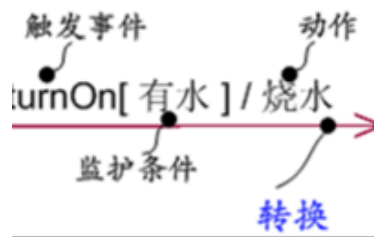
- 
- 大小: 4.9 KB




- 
- 大小: 2.8 KB

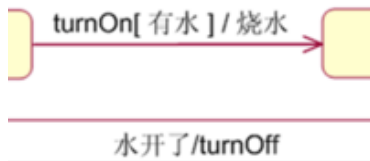


- 
- 大小: 20.5 KB

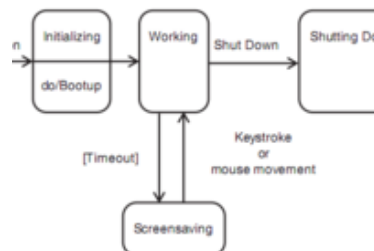


- 
- 大小: 23.8 KB

【水】



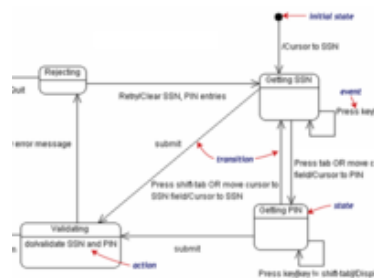
- 大小: 27.9 KB





- 大小: 21.6 KB



- 大小: 19.4 KB



- 大小: 57.8 KB

分享到:  

## 适配器模式——Adaptor(Adapter) | 装饰者模式——Decorator

- 2011-12-28 14:10
- 浏览 10322
- [评论\(0\)](#)
- 分类:[行业应用](#)
- [相关推荐](#)

评论

发表评论



[您还没有登录,请您登录后再发表评论](#)



chuanwang66

- 浏览: 175665 次
- 性别:
- 来自: 上海
- 我现在离线

最近访客

[更多访客>>](#)



[undead244](#)



[zhoutonglx](#)



[Super\\_Pippo](#)



[tianzx](#)

文章分类

- [全部博客 \(232\)](#)
- [一个操作系统的实现 \(20\)](#)
- [汇编\(NASM\) \(12\)](#)
- [Linux编程 \(11\)](#)
- [项目管理 \(4\)](#)
- [计算机网络 \(8\)](#)
- [设计模式\(抽象&封装\) \(17\)](#)
- [数据结构和算法 \(32\)](#)
- [java基础 \(6\)](#)
- [UML细节 \(2\)](#)
- [C/C++ \(31\)](#)
- [Windows \(2\)](#)

- [乱七八糟 \(13\)](#)
- [MyLaB \(6\)](#)
- [系统程序员-成长计划 \(8\)](#)
- [POJ部分题目 \(10\)](#)
- [数学 \(6\)](#)
- [分布式 & 云计算 \(2\)](#)
- [python \(13\)](#)
- [面试 \(1\)](#)
- [链接、装载与库 \(11\)](#)
- [java并行编程 \(3\)](#)
- [数据库 \(0\)](#)
- [体系结构 \(3\)](#)
- [C++ template / STL \(4\)](#)
- [Linux环境和脚本 \(6\)](#)

#### 社区版块

- [我的资讯 \(0\)](#)
- [我的论坛 \(0\)](#)
- [我的问答 \(0\)](#)

#### 存档分类

- [2014-09 \(1\)](#)
- [2014-03 \(1\)](#)
- [2014-02 \(3\)](#)
- [更多存档...](#)

#### 最新评论

- [chuanwang66](#): 默默水塘 写道typedef void(\*Fun)(void) ...  
[C++虚函数表 \(转\)](#)
- [默默水塘](#): typedef void(\*Fun)(void);  
[C++虚函数表 \(转\)](#)
- [lishaogingmn](#): 写的很好, 例子简单明了, 将观察者模式都表达了出来。 这里是ja ...  
[观察者模式——Observer](#)

---

声明: ITeye文章版权属于作者, 受法律保护。没有作者书面许可不得转载。若作者同意转载, 必须以超链接形式标明文章原始出处和作者。

© 2003-2015 ITeye.com. All rights reserved. [ 京ICP证110151号 京公网安备110105010620 ]