|  |  |
| --- | --- |
| 目录 | [外观模式与命令模式外观模式实例代码命令模式实例代码区别](javascript:;) |

# 外观模式与命令模式

## 外观模式

* 概念为一个子系统中的一组接口，提供一个一致的交互界面。定义了一个高层接口，使得这一个子系统更容易使用。
* 核心思想用一个简单的接口，封装一个复杂的系统
* 设计要点：明确角色外观（Facade）：封装后的对外接口子系统（SubSystem）：由多个类组成，具有特定功能的子系统
* 优点1 方便子系统的调用、管理和维护2 简化子系统使用难度，调用方无须关心子系统内部细节3 降低系统耦合度
* 缺点系统灵活度降低
* 应用场景1 要为某复杂系统提供对外调用的接口2 调用方与子系统存在依赖关系，引入外观类将子系统独立3 在多层次结构中，为每一层都定义一个外观类，将层次间解耦，通过外观类交互

### 实例代码

1. """
2. Rectangle Square Circle
3. 都是某一子系统内的功能点
4. ShapeMaker
5. 是我们为这个子系统设计的外观类
6. """
7. class Rectangle(object):
8. def draw(self):
9. print("Rectangle::draw() 画了个长方形")
10. class Square(object):
11. def draw(self):
12. print("Square::draw() 画了个正方形")
13. class Circle(object):
14. def draw(self):
15. print("Circle::draw() 画了个圆形")
16. class ShapeMaker(object):
17. def \_\_init\_\_(self):
18. self.rectangle = Rectangle()
19. self.square = Square()
20. self.circle = Circle()
21. def draw\_rectangle(self):
22. self.rectangle.draw()
23. def draw\_square(self):
24. self.square.draw()
25. def draw\_circle(self):
26. self.circle.draw()
27. if \_\_name\_\_ == '\_\_main\_\_':
28. # 外观交互方式
29. shape\_maker = ShapeMaker()
30. shape\_maker.draw\_rectangle()
31. shape\_maker.draw\_square()
32. shape\_maker.draw\_circle()
33. print("--- 分割线 ---")
34. # 子系统交互方式
35. rectangle = Rectangle()
36. rectangle.draw()
37. square = Square()
38. square.draw()
39. circle = Circle()
40. circle.draw()

## 命令模式

* 概念将客户端的请求封装成对象，从而使用不同的请求把客户端参数化，可以对请求排队或记录日志，也可以提供命令的撤销和恢复功能。最大的特点是将命令与对应的命令接收者相关联，使调用方不用关心执行者是谁以及具体的执行过程，只需要发送正确的命令。
* 设计要点：明确角色1 用户（Client）：命令的发送者2 命令（Command）：要做的操作或者任务3 命令调度者（Invoker）：接收外部命令，分配给对应接收者4 命令接收者（Receiver）：命令的真实执行者
* 优点1 命令发送者与命令执行者解耦，不再相互依赖2 可以很方便地新增命令，组合命令
* 缺点每一个命令都需要一个命令接收者，使类数目大增
* 应用场景1 需要将调用方和被调方解耦2 某些操作可抽象为命令，比如GUI中的按键操控3 需要将一系列操作进行排列组合成各种组合操作

### 实例代码

1. # coding: utf-8
2. import abc
3. class Command(metaclass=abc.ABCMeta):
4. """ 命令抽象类 """
5. @abc.abstractmethod
6. def execute(self):
7. """ 命令如何执行 """
8. ...
9. class CommandReceiver():
10. """ 命令接收者 处理命令 """
11. def do\_command1(self):
12. print("命令1 执行完成")
13. def do\_command2(self):
14. print("命令2 执行完成")
15. def \_\_repr\_\_(self) -> str:
16. return "命令执行者"
17. class Command\_1(Command):
18. """ 每一个命令，都要有一个命令接收者 """
19. def \_\_init\_\_(self, receiver: CommandReceiver = CommandReceiver()):
20. self.receiver = receiver
21. def execute(self):
22. print(f"{receiver} 执行命令")
23. self.receiver.do\_command1()
24. def \_\_repr\_\_(self) -> str:
25. return "命令1"
26. class Command\_2(Command):
27. """ 每一个命令，都要有一个命令接收者，此处不指定默认接收者 """
28. def \_\_init\_\_(self, receiver):
29. self.receiver = receiver
30. def execute(self):
31. print(f"{receiver} 执行命令")
32. self.receiver.do\_command2()
33. def \_\_repr\_\_(self) -> str:
34. return "命令2"
35. class Invoker(object):
36. def \_\_init\_\_(self):
37. self.command\_list = []
38. def take\_command(self, command: Command):
39. print(f"{self} 收到命令： {command}")
40. self.command\_list.append(command)
41. def take\_commands(self, commands: [Command]):
42. for command in commands:
43. print(f"{self} 收到命令： {command}")
44. self.command\_list.append(command)
45. def dispatch\_command(self):
46. for command in self.command\_list:
47. command.execute()
48. self.command\_list.clear()
49. def \_\_repr\_\_(self) -> str:
50. return "命令接收者"
51. if \_\_name\_\_ == '\_\_main\_\_':
52. invoker = Invoker()
53. receiver = CommandReceiver()
54. command1 = Command\_1()
55. invoker.take\_command(command1)
56. invoker.dispatch\_command()
57. command2 = Command\_2(receiver)
58. invoker.take\_command(command2)
59. invoker.dispatch\_command()
60. invoker.take\_commands([command1, command2])
61. invoker.dispatch\_command()

## 区别

1. 外观模式引入了外观类与调用方交互命令模式引入了命令调度者与调用方交互
2. 外观模式中，调用方把要做的事情直接告诉子系统，没有Command概念命令模式中，调用方把要做的事，封装成命令，把命令发给命令调度者
3. 命令模式中可以做到命令的撤销和恢复
4. 命令模式中，Command的子类都要实现execute方法，执行方式统一，调用方做不同的事情，都只需要发请求外观模式中，要做不同的事，要调用外观类中不同的接口