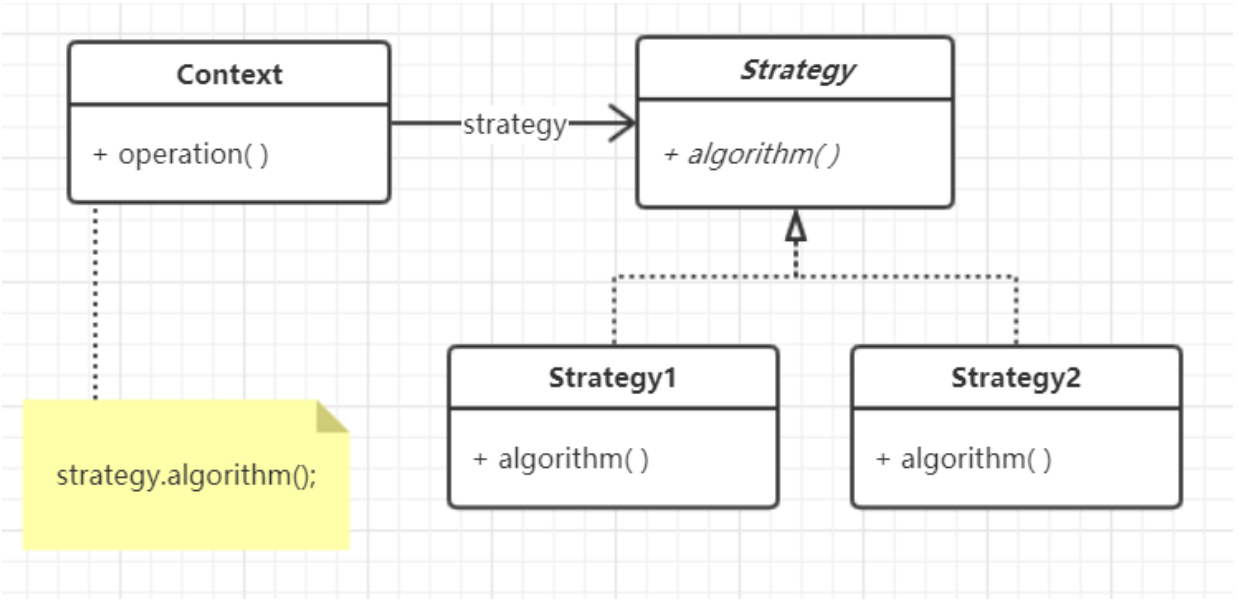
策略模式

Strategy

郭嘉

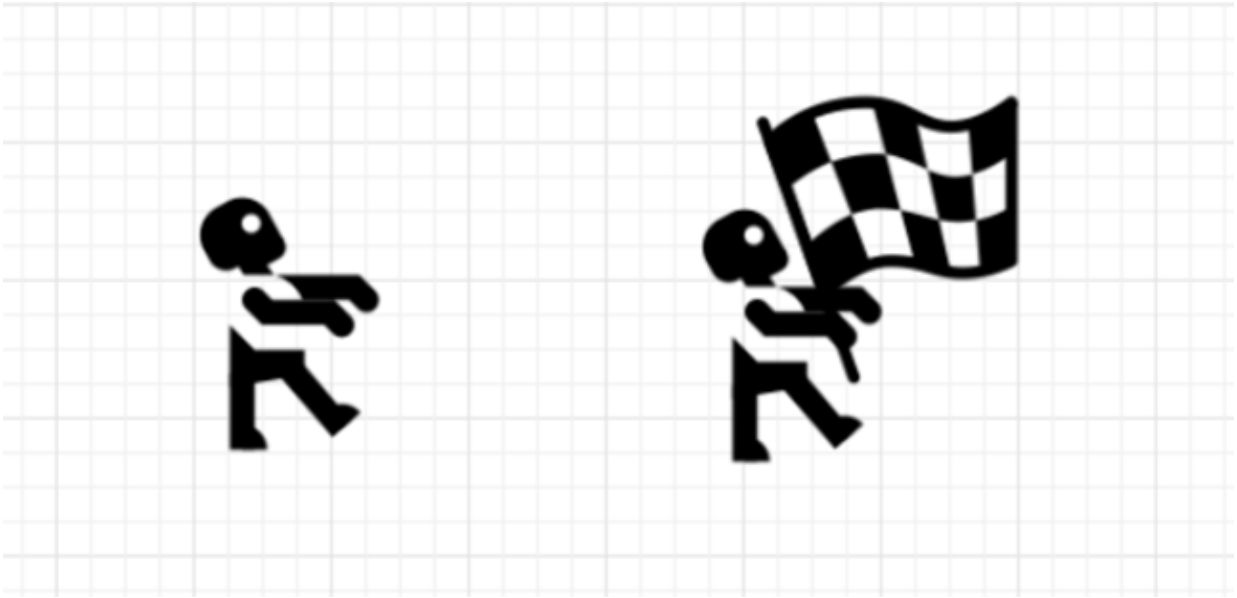
模式定义：

定义了算法族，分别封装起来，让它们之间可以互相替换，此模式的变化独立于算法的使用者。

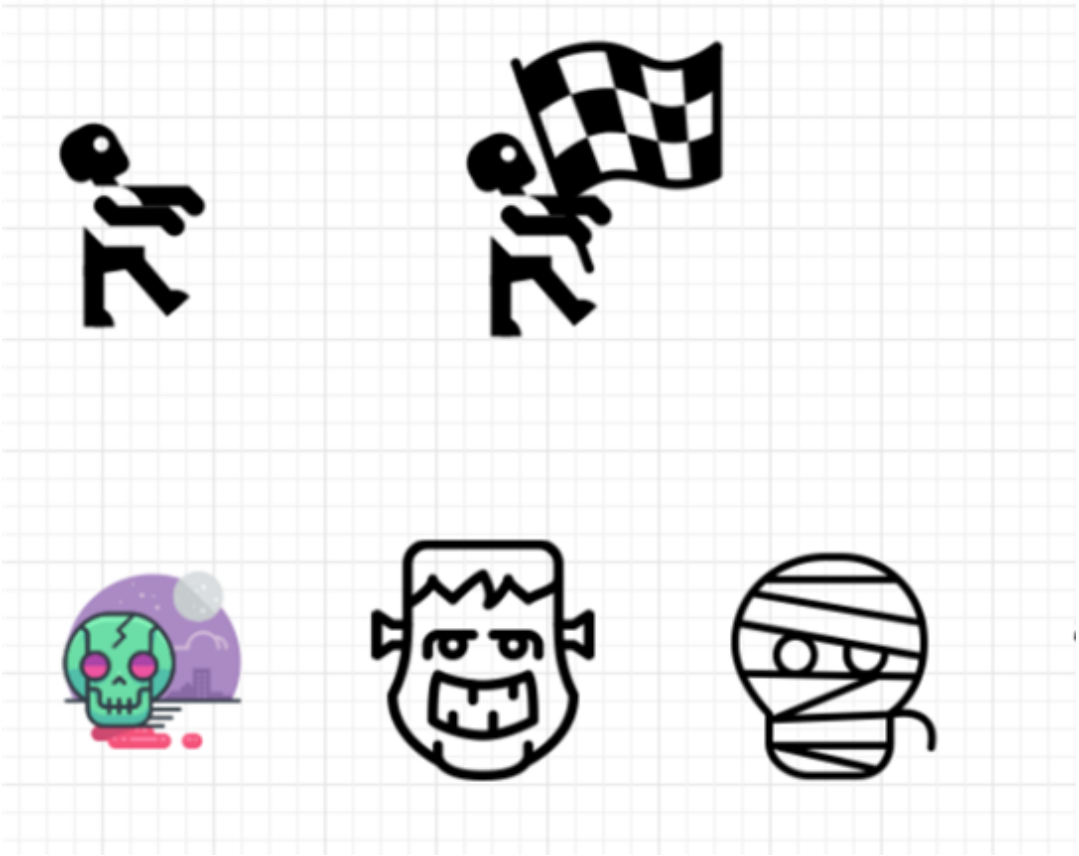


案例:

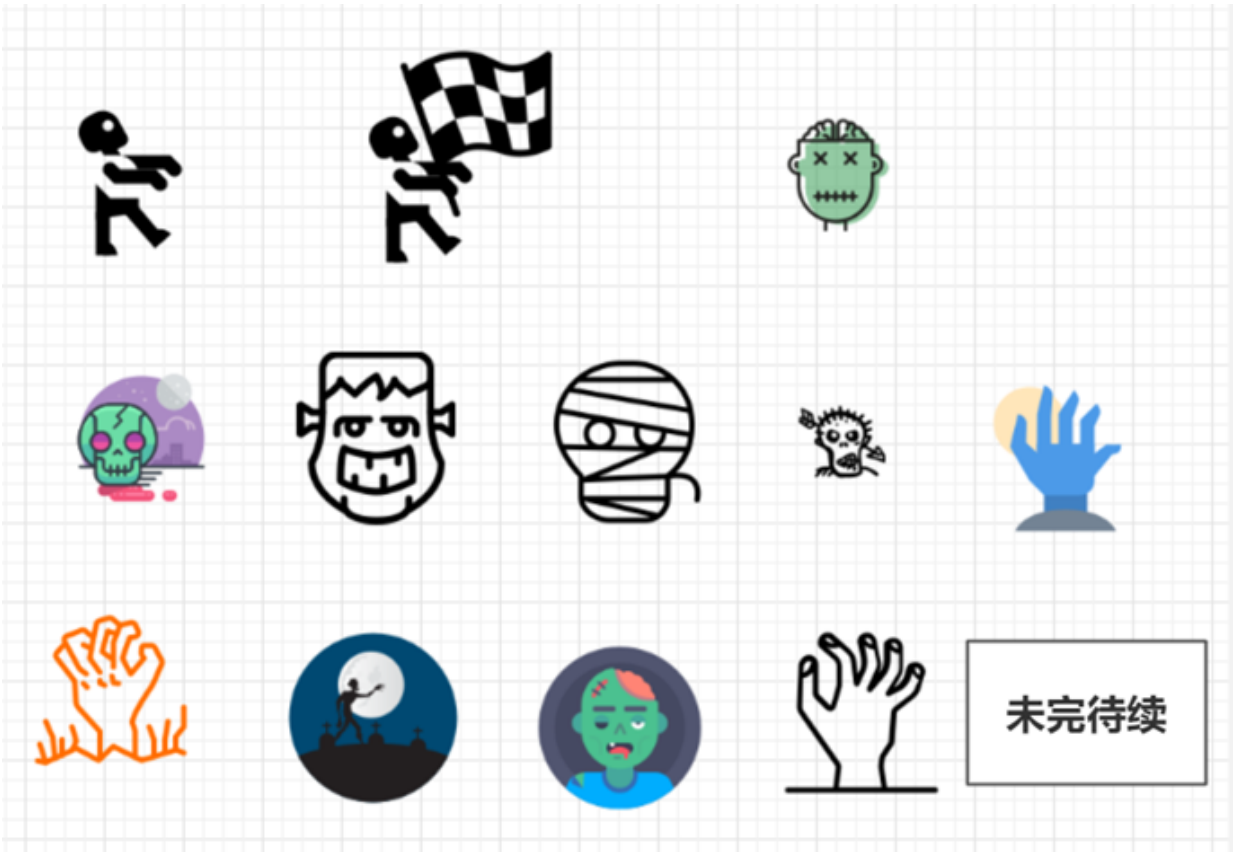
类型	外观	移动	攻击
普通僵尸	普通	朝着一个方向移动	咬
旗手僵尸	普通+手持旗子	朝着一个方向移动	咬



类型	外观	移动	攻击
普通僵尸	普通	朝着一个方向移动	咬
旗手僵尸	普通+手持旗子	朝着一个方向移动	咬
大头僵尸	大头	朝着一个方向移动	头撞
石膏僵尸	石膏装	一拐一瘸	武器
XXX僵尸	...	...	...



未完待续 . . . .



```java
package com.tuling.designpattern.strategy.v2;

/**
 * @author 腾讯课堂-图灵学院 郭嘉
 * @Slogan 致敬大师，致敬未来的你
 */
public class StrategyTest {
 public static void main(String[] args) {
 Zombie zombie=new NormalZombie( );
  zombie.display();

  zombie.attack();
  zombie.move();

  zombie.setAttackable( new BiteAttack() );
  zombie.attack();
  }
}
interface Moveable{
  void move();
}
interface Attackable{
  void attack();
}
abstract class Zombie{
  abstract public void display();
  Moveable moveable;
  Attackable attackable;

  public Zombie(Moveable moveable, Attackable attackable) {
  this.moveable=moveable;
  this.attackable=attackable;
  }

  abstract void move();
  abstract void attack();

  public Moveable getMoveable() {
  return moveable;
  }

```

```
 public void setMoveable(Moveable moveable) {
 this.moveable=moveable;
 }

 public Attackable getAttackable() {
 return attackable;
 }

 public void setAttackable(Attackable attackable) {
 this.attackable=attackable;
 }
}


class StepByStepMove implements Moveable{

 @Override
 public void move() {

 System.out.println("一步一步移动.");
 }
}

class BiteAttack implements Attackable{

 @Override
 public void attack() {
 System.out.println("咬.");
 }
}

class HitAttack implements Attackable{

 @Override
 public void attack() {
 System.out.println("打.");
 }
}

class FlagZombie extends Zombie{

```

```
83  public FlagZombie(){
84  super(new StepByStepMove(),new BiteAttack());
85  }
86
87  public FlagZombie(Moveable moveable, Attackable attackable) {
88  super( moveable, attackable );
89  }
90
91  @Override
92  public void display() {
93  System.out.println("我是旗手僵尸.");
94  }
95
96  @Override
97  void move() {
98  moveable.move();
99  }
100
101  @Override
102  void attack() {
103  attackable.attack();
104  }
105 }
106
107
108 class NormalZombie extends Zombie{
109
110  public NormalZombie(){
111  super(new StepByStepMove(),new BiteAttack());
112  }
113
114  public NormalZombie(Moveable moveable, Attackable attackable) {
115  super( moveable, attackable );
116  }
117
118  @Override
119  public void display() {
120  System.out.println("我是普通僵尸.");
121  }
122
123  @Override
```

```
124     void move() {
125     moveable.move();
126     }
127
128     @Override
129     void attack() {
130     attackable.attack();
131     }
132 }
133
134
```

应用场景

1. 当你有很多类似的类，但它们执行某些行为的方式不同时，请使用此策略。

2. 使用该模式将类的业务逻辑与算法的实现细节隔离开来，这些算法在逻辑上下文中可能不那么重要。

3. 当你的类具有大量的条件运算符，并且在同一算法的不同变体之间切换时，请使用此模式。

优点：

1. 可以将算法的实现细节与使用它的代码隔离开来。

2. 符合开闭原则

Spring &JDK 源码中的应用

```
1 java.util.Comparator
2 org.springframework.beans.factory.support.InstantiationStrategy
```