Java设计模式

1. Builder模式

创建个辅助类（一般使用内部类）`Builder`，先将属性设置到`Builder`中，然后调用`Builder#build`方法生成实例；

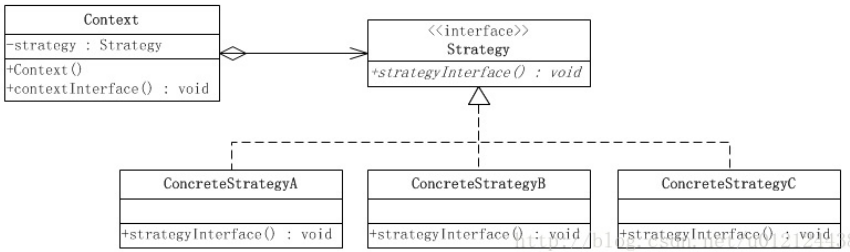
```
public class Toy {
    private String head;
    private String body;
    private ArrayList legs;
    private ArrayList hands;

    public String getHead() {
        return head;
    }

    public void setHead(String head) {
        this.head = head;
    }

    public String getBody() {
        return body;
    }

    public void setBody(String body) {
        this.body = body;
    }

    public ArrayList getLegs() {
        return legs;
    }

    public void setLegs(ArrayList legs) {
        this.legs = legs;
    }

    public ArrayList getHands() {
        return hands;
    }

    public void setHands(ArrayList hands) {
        this.hands = hands;
    }

    static class Builder {
        private Toy toy;

        public Builder() {
            toy = new Toy();
        }

        public Builder setHead(String head) {
            toy.setHead(head);
            return this;
        }

        public Builder setBody(String body) {
            toy.setBody(body);
            return this;
        }

        public Builder setLegs(ArrayList legs) {
            toy.setLegs(legs);
            return this;
        }

        public Builder setHands(ArrayList hands) {
```

```
            toy.setHands(hands);
            return this;
        }

        public Toy build() {
            return toy;
        }
    }

    public static void main(String[] hh) {
        ArrayList hands = new ArrayList();
        hands.add("left");
        hands.add("right");

        ArrayList legs = new ArrayList();
        legs.add("left");
        legs.add("right");

        Toy toy = new Toy.Builder()
                .setBody("body")
                .setHands(hands)
                .setLegs(legs)
                .setHead("head")
                .build();
    }

}
```

2. 策略模式

2.1 定义

- 策略模式定义了一系列的算法，并将每一个算法封装起来，而且使他们可以相互替换，让算法独立于使用它的客户而独立变化



- 行为参数化模式 与 策略模式 比较相近

3. 单例模式

3.1 饱汉模式

缺点：非线程安全；

```
public class SingletonLazy {

    private static volatile SingletonLazy instance;

    private SingletonLazy() {
    }

    public static synchronized SingletonLazy getInstance() {
        if (instance == null) {
            instance = new SingletonLazy();
```

```
        }
        return instance;
    }
}
```

解决线程安全问题的简单方法：使用synchronized关键字

```
public class SingletonLazy {

    private static volatile SingletonLazy instance;

    private SingletonLazy() {
    }

    public static synchronized SingletonLazy getInstance() {
        if (instance == null) {
            instance = new SingletonLazy();
        }
        return instance;
    }
}
```

但是这样直接使用synchronized效率就大打折扣，可以使用双重检查锁提高性能：

```
public class Singleton {
    private static volatile Singleton singleton = null;

    private Singleton(){}

    public static Singleton getSingleton(){
        if(singleton == null){
            synchronized (Singleton.class){
                if(singleton == null){
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

扩展：关键字volatile作用：

1. 会将高速缓存（CPU内核缓存，线程独享）的数据立马同步到主内存（内存条，线程共享）中；
2. 禁止指令重排序优化

> 我们写的代码（特别是多线程代码），由于编译器优化，在实际执行的时候可能与我们编写的顺序不同。编译器只保证程序执行结果与源代码相同，却不保证实际指令的顺序与源代码相同，这在单线程并没什么问题，然而一旦引入多线程环境，这种乱序就可能导致严重问题。

3. 深入分析Volatile的实现原理

> 了解文中的`Volatile 的使用优化`

3.2 饿汉模式

缺点：创建不必要的对象；

```
public class SingletonHungry {

    private static SingletonHungry instance = new SingletonHungry();

    private SingletonHungry() {
```

```
    }

    public static SingletonHungry getInstance() {
        return instance;
    }
}
```

3.3 静态内部类

```
public class SingletonInner {
    private static class Holder {
        private static SingletonInner singleton = new SingletonInner();
    }

    private SingletonInner(){}

    public static SingletonInner getSingleton(){
        return Holder.singleton;
    }
}
```

3.4 枚举单例模式

前面的几种模式有2个缺点：

1. 序列化可能会破坏单例模式，每次反序列化都会创建一个新的实例；

   解决方法是自己动手。

2. 使用反射强行调用私有构造器；

   解决方式是可以修改构造器，让它在创建第二个实例的时候抛异常。

枚举单例模式：

```
public enum  SingletonEnum {
    // 可以在这里实现单例需要的方法，而INSTANCE就保证了绝对的单例
    INSTANCE;
    private String name;
    public String getName(){
        return name;
    }
    public void setName(String name){
        this.name = name;
    }
}
```

既可防止反射攻击，也可防止反序列化产生多实例;

4. 适配器模式

4.1 优秀博客

适配器模式| 菜鸟教程

4.2 理解

> 适配器模式（Adapter Pattern）是作为两个不兼容的接口之间的桥梁。这种类型的设计模式属于结构型模式；

个人理解：就是在一个类中创建你想兼容的功能的对象，然后封装这些功能，再提供出去，最后达到不兼容对象的功能集成的目的；

5. 修饰器模式

5.1 博客/教程

装饰器模式| 菜鸟教程

4. 责任链模式

类型：行为型模式

通俗理解：通常每个接收者都包含对另一个接收者的引用。如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者

5.2 理解

> 装饰器模式（Decorator Pattern）允许向一个现有的对象添加新的功能，同时又不改变其结构。这种类型的设计模式属于结构型模式，它是作为现有的类的一个包装。

个人理解（助记）：就是创建一个有基本功能的类（一般是抽象类，但实现了某些通用的、基本的逻辑），然后，再根据自己的功能需求，继承这个抽象类并进行扩展和修饰，实现更多的功能；