Java常用设计模式

作者：叶翠

目录

单例模式（Singleton Pattern）

单例模式是最常用的设计模式之一。它可以确保在整个应用程序中，某个类只有一个实例存在，并提供一种访问这个实例的全局访问点。单例模式在需要限制某些类的实例数量时非常有用。 它通常用于需要全局访问的资源，如配置文件、日志记录器、数据库连接等。

应用场景

1. 日志记录器 在一个应用程序中，通常会有多个模块或类需要记录日志。为了避免创建多个日志记录器实例，使用单例模式可以确保只有一个日志记录器实例，从而避免重复记录日志并提高应用程序的性能。
2. 数据库连接 在一个应用程序中，如果需要频繁地与数据库交互，使用单例模式可以确保只有一个数据库连接实例，从而减少数据库连接的数量，提高应用程序的性能。
3. 系统配置 在一个应用程序中，通常会有一些全局的配置参数，如数据库连接字符串、服务器地址、缓存大小等。使用单例模式可以确保只有一个配置实例，从而方便管理和修改配置参数。

代码实现

懒汉式

```
csharp
复制代码public class Singleton {
    private static Singleton instance;

    private Singleton() {
        // 私有构造函数，防止外部实例化
    }

    public static Singleton getInstance() {
        if (instance == null) {
```

```
            instance = new Singleton();
        }
        return instance;
    }
}
```

静态内部类方式

`SingletonHolder`是一个静态内部类，它包含一个静态的`INSTANCE`成员变量，用于存储单例对象。在第一次调用`getInstance`方法时，静态内部类会被加载，从而创建单例对象。这种方式既兼顾了线程安全又兼顾了延迟加载的需求。

```
csharp
复制代码public class Singleton {
    private Singleton() {
        // 私有构造函数，防止外部实例化
    }

    public static Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }

    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }
}
```

饿汉式

饿汉式在类加载时就创建了单例对象，所以不存在线程安全问题。不过，这种方式可能会导致不必要的资源浪费，因为单例对象的创建可能在应用程序启动时就完成了，而有些应用场景中可能并不需要使用单例对象。

```
csharp
复制代码public class Singleton {
    // 在类加载时就创建单例对象
    private static Singleton instance = new Singleton();

    // 将构造函数设为私有，禁止外部创建实例
    private Singleton() {}

    // 提供获取单例对象的方法
    public static Singleton getInstance() {
        return instance;
    }
}
```

双重检查锁

它可以在保证线程安全的同时实现延迟加载

```
csharp
复制代码public class Singleton {
    private static volatile Singleton instance;

    private Singleton() {}

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

枚举方式

使用枚举实现单例模式的好处是，可以避免反射和序列化攻击。因为枚举类型的构造函数是私有的，所以无法使用反射来创建实例；而且枚举类型的实例在序列化和反序列化时会自动处理好，所以也无法通过序列化和反序列化来破坏单例。

```
csharp
复制代码public enum Singleton {
    INSTANCE;

    public void doSomething() {
        // TODO: 实现单例对象的功能
    }
}
```

使用小结

- 对线程安全和性能要求较高，可以考虑使用**饿汉式**或**双重检查锁**方式实现单例模式。这两种方式都能保证线程安全，而且在大多数情况下性能也比较好。
- 如果你对线程安全要求不是很高，或者希望在第一次访问时才创建单例对象，可以考虑使用**懒汉式**或者**静态内部类方式**。这两种方式都是延迟加载的，只有在需要时才会创建单例对象。懒汉式不是线程安全的，需要通过加锁等方式来保证线程安全；而静态内部类方式则是天生线程安全的，不需要额外的处理。
- 希望实现简单、代码少，且不需要考虑线程安全和延迟加载的问题，可以考虑使用**枚举方式**。这种方式不仅代码简单，而且天生线程安全、单例对象创建和调用都很方便。

总之，选择哪种实现方式需要根据具体需求来决定，需要综合考虑线程安全、性能、代码复杂度、延迟加载等因素。

工厂模式（Factory Pattern）

工厂模式是一种创建型模式，它可以为开发人员提供一种在不直接实例化对象的情况下创建对象的方法。工厂模式通过提供一个通用的接口和一组实现，来隐藏具体实现的细节，从而降低了代码的耦合度和依赖性。

应用场景

1. 对象的创建过程比较复杂，需要进行封装：如果创建一个对象需要进行复杂的初始化过程，或者需要从多个地方获取数据才能创建对象，那么使用工厂模式可以将这些过程封装起来，让客户端代码更加简洁和易于理解。
2. 需要动态扩展或修改对象的创建过程：如果需要增加或修改某个对象的创建过程，而又不希望对客户端代码产生影响，那么使用工厂模式可以很方便地实现这个需求。
3. 需要统一管理对象的创建：如果需要统一管理对象的创建过程，或者需要对创建的对象进行某些统一的处理，那么使用工厂模式可以很好地实现这个需求。
4. 需要根据不同的条件创建不同的对象：如果需要根据不同的条件来创建不同类型的对象，那么使用工厂模式可以很方便地实现这个需求。

代码实现

通过一个工厂类来封装对象的创建过程，客户端只需要告诉工厂类需要创建哪种类型的对象即可。将对象的创建过程与客户端代码分离开来，使代码更加灵活和易于扩展

```
typescript
复制代码// 定义产品接口
public interface Product {
    void operation();
}

// 具体产品类A
public class ConcreteProductA implements Product {
    @Override
    public void operation() {
        System.out.println("ConcreteProductA operation.");
    }
}

// 具体产品类B
public class ConcreteProductB implements Product {
    @Override
    public void operation() {
        System.out.println("ConcreteProductB operation.");
    }
}

// 工厂类
public class SimpleFactory {
    public static Product createProduct(String type) {
```

```
        if ("A".equals(type)) {
            return new ConcreteProductA();
        } else if ("B".equals(type)) {
            return new ConcreteProductB();
        } else {
            throw new IllegalArgumentException("Invalid product type.");
        }
    }
}
```

客户端可以通过调用`SimpleFactory.createProduct`方法来创建不同类型的产品对象

```
ini
复制代码Product productA = SimpleFactory.createProduct("A");
productA.operation(); // 输出 "ConcreteProductA operation."

Product productB = SimpleFactory.createProduct("B");
productB.operation(); // 输出 "ConcreteProductB operation."
```

使用小结

在`Java`中，工厂模式广泛应用于各种框架和类库中，例如JDBC中的`DataSource`工厂、`Spring`框架中的`Bean`工厂、`MyBatis`框架中的`SqlSessionFactory`等等。

观察者模式（Observer Pattern）

观察者模式是一种行为型模式，它定义了对象之间的一种一对多的依赖关系。在这种模式中，一个对象发生变化时，所有依赖于它的对象都会得到通知并自动更新。观察者模式可以帮助开发人员创建可扩展的应用程序，减少对象之间的直接依赖关系。

应用场景

1. 事件处理机制：`Java`中的`Swing GUI`框架就是基于观察者模式实现的，当用户与组件交互时，组件会向注册的监听器发送事件通知，以触发相应的事件处理方法。
2. 日志记录：`Java`中的日志系统也是基于观察者模式实现的，当日志发生变化时，它会通知所有注册的观察者，例如文件输出流、控制台输出流等，从而实现日志的输出和记录。
3. 用户界面设计：在`Java`中，用户界面设计中的许多元素都可以使用观察者模式实现，例如菜单项、按钮、文本框等，当用户与这些元素交互时，它们会向注册的监听器发送事件通知，以触发相应的事件处理方法。
4. 多线程编程：在`Java`中，观察者模式还可以用于多线程编程中，当一个线程发生了某些变化时，它可以向其他线程发送通知，以实现线程间的协作和同步。

代码实现

这个示例中，`ConcreteSubject`实现了`Subject`接口，它维护了一个`observers`列表，用于保存注册的观察者对象。当被观察者发生变化时，它会遍历观察者列表，调用每个观察者的`update`方法。

`ConcreteObserver` 实现了 `Observer` 接口，它可以接收来自被观察者的通知，并执行相应的操作。

在测试类 `ObserverPatternDemo` 中，我们创建了一个具体的被观察者对象 `ConcreteSubject`，并注册了两个具体的观察者对象 `observer1` 和 `observer2`。当被观察者发生变化时，它会通知所有注册的观察者对象，并调用它们的 `update` 方法。

```
typescript
复制代码// 观察者接口
interface Observer {
    void update(String message);
}

// 被观察者接口
interface Subject {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers(String message);
}

// 具体的被观察者实现类
class ConcreteSubject implements Subject {
    private List<Observer> observers = new ArrayList<>();

    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }
}

// 具体的观察者实现类
class ConcreteObserver implements Observer {
    private String name;

    public ConcreteObserver(String name) {
        this.name = name;
    }
```

```
43
44      @Override
45      public void update(String message) {
46          System.out.println(name + " received message: " + message);
47      }
48  }
49
50  // 测试类
51  public class ObserverPatternDemo {
52      public static void main(String[] args) {
53          Subject subject = new ConcreteSubject();
54          Observer observer1 = new ConcreteObserver("Observer1");
55          Observer observer2 = new ConcreteObserver("Observer2");
56          subject.registerObserver(observer1);
57          subject.registerObserver(observer2);
58          subject.notifyObservers("Hello, World!");
59      }
60  }
```

使用小结

观察者模式的优点在于它提供了一种松耦合的方式，让观察者和主题之间的依赖关系变得更加灵活，同时也可以使得程序更易于扩展和维护。

观察者模式的应用场景包括：当一个抽象模型有两个方面，其中一个方面依赖于另一个方面时；当一个对象的改变需要同时改变其他对象的时候；当一个对象的改变需要通知其他对象而又不希望与被通知对象形成紧耦合关系时。

适配器模式（Adapter Pattern）

适配器模式是一种结构型模式，它可以将一个类的接口转换成客户端所期望的另一种接口。适配器模式可以帮助开发人员在不修改现有代码的情况下，将不兼容的类组合在一起。适配器模式包括以下几个组成部分：

- 目标接口（`Target Interface`）：客户端期望的接口。
- 适配器（`Adapter`）：充当两个不兼容接口之间的桥梁，使得它们可以互相通信。
- 适配者（`Adaptee`）：需要被适配的对象，它的接口与目标接口不兼容。
- 客户端（`Client`）：使用目标接口的对象。

应用场景

- 当需要将一个已有的类或接口与另一个不兼容的类或接口进行协同工作时。
- 当需要对一个已有的类或接口进行修改，以满足客户端的需求时，但是不希望修改该类或接口的源代码。
- 当需要重新使用一个已有的类或接口，但是不能直接使用该类或接口的方法时。

代码实现

在这个示例中，我们有一个目标接口 `Target` 和一个不兼容的适配者 `Adaptee`，我们需要创建一个适配器 `Adapter` 来让它们能够一起工作。

适配器实现了目标接口 `Target`，并在构造函数中接受一个适配者对象 `Adaptee`，然后在实现目标接口的 `request` 方法中调用适配者的 `specificRequest` 方法。

在客户端中，我们创建了一个适配者对象 `adaptee`，并将其传递给适配器的构造函数创建一个适配器对象 `adapter`。最后，我们使用目标接口 `Target` 中定义的方法 `request` 来访问适配器，从而调用适配者的方法。

```
csharp
复制代码// 目标接口
interface Target {
    void request();
}

// 适配者
class Adaptee {
    void specificRequest() {
        System.out.println("Adaptee specificRequest.");
    }
}

// 适配器
class Adapter implements Target {
    private Adaptee adaptee;

    public Adapter(Adaptee adaptee) {
        this.adaptee = adaptee;
    }

    @Override
    public void request() {
        adaptee.specificRequest();
    }
}

// 客户端
public class AdapterPatternDemo {
    public static void main(String[] args) {
        Adaptee adaptee = new Adaptee();
        Target target = new Adapter(adaptee);
        target.request();
    }
}
```

使用小结

适配器模式是一种非常有用的设计模式，在`JDK`中被广泛应用，可以提供一致的接口，比如：

1. `Java IO` 流是一个常见的适配器模式的例子。它提供了一组标准的接口来访问各种类型的数据源，包括文件、网络连接、内存等等。每个数据源都有自己的接口，但是 `Java IO` 流可以将这些不同的接口转换为标准的接口，从而提供一致的访问方式。
2. `Java Servlet API` 也是一个常见的适配器模式的例子。它定义了一组接口来处理 `HTTP` 请求和响应，包括 `doGet()、doPost()、doPut()` 等等。每个 `Servlet` 都必须实现这些接口，但是用户只需要实现其中的一部分即可。这些 `Servlet` 之间的适配工作由 `Servlet` 容器完成。

装饰器模式（Decorator Pattern）

装饰器模式是一种结构型模式，它可以允许开发人员在不修改现有对象的情况下，动态地添加新功能。装饰器模式通过将一个对象包装在另一个对象中来扩展它的行为，从而提高了代码的灵活性和可重用性。

应用场景

1. 当需要在不修改现有对象结构的前提下增加新的功能或特性时，可以使用装饰器模式。这样可以保持原有代码的稳定性和兼容性，同时也可以增加代码的灵活性和可扩展性。
2. 当需要动态地向对象添加或删除功能时，可以使用装饰器模式。这样可以在运行时动态地添加或删除功能，而不需要修改现有的代码。
3. 当需要为多个对象添加相同的功能时，可以使用装饰器模式。这样可以将相同的功能封装在装饰器中，以便于复用和管理。

代码实现

该示例代码中，`Shape` 是一个接口，定义了一个 `draw` 方法，表示绘制图形的操作。`Circle` 是一个实现 `Shape` 接口的类，表示一个圆形。

`ShapeDecorator` 是一个装饰器抽象类，实现了 `Shape` 接口，并包含一个 `Shape` 类型的变量 `decoratedShape`，表示要装饰的对象。`RedShapeDecorator` 是一个具体的装饰器类，继承了 `ShapeDecorator` 类，并实现了 `draw` 方法，在绘制图形时添加了一个红色的边框。

在 `main` 方法中，我们创建了原始对象 `Circle`，以及两个装饰器对象 `RedShapeDecorator`，分别装饰了 `Circle` 和 `Rectangle` 对象。通过调用 `draw` 方法，我们可以看到对象被动态地添加了一个红色的边框，而不需要修改原有的代码。

```
csharp
复制代码// 定义接口
interface Shape {
    void draw();
}

// 实现接口
class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Shape: Circle");
    }
}
```

```java

class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Shape: Rectangle");
    }
}

// 装饰器抽象类
abstract class ShapeDecorator implements Shape {
    protected Shape decoratedShape;

    public ShapeDecorator(Shape decoratedShape){
        this.decoratedShape = decoratedShape;
    }

    public void draw(){
        decoratedShape.draw();
    }
}

// 具体装饰器类
class RedShapeDecorator extends ShapeDecorator {
    public RedShapeDecorator(Shape decoratedShape) {
        super(decoratedShape);
    }

    @Override
    public void draw() {
        decoratedShape.draw();
        setRedBorder(decoratedShape);
    }

    private void setRedBorder(Shape decoratedShape){
        System.out.println("Border Color: Red");
    }
}

// 测试代码
public class DecoratorPatternDemo {
    public static void main(String[] args) {
        // 创建原始对象
        Shape circle = new Circle();

        // 创建装饰器对象
        Shape redCircle = new RedShapeDecorator(new Circle());
        Shape redRectangle = new RedShapeDecorator(new Rectangle());

        // 调用方法
```

```
        System.out.println("Circle with normal border");
        circle.draw();

        System.out.println("\nCircle of red border");
        redCircle.draw();

        System.out.println("\nRectangle of red border");
        redRectangle.draw();
    }
}
```

使用小结

在实际应用中，装饰器模式经常用于图形界面(`GUI`)开发、输入/输出流处理、缓存机制、日志记录等领域，可以有效地提高程序的可扩展性和可维护性。比如

1. 装饰器模式被广泛应用于`Java IO`流中，以提供各种不同的功能，如缓存、压缩、加密等等。例如，可以使用`BufferedReader`来缓存读取文件的数据，使用`GZIPOutputStream`来压缩数据，使用`CipherOutputStream`来加密数据等等。
2. Java Swing 组件是一个经典的装饰器模式的例子。它允许在运行时动态地向组件添加功能，如边框、背景、文本等等。例如，可以使用`BorderFactory`来向组件添加边框，使用`Color`来设置组件的背景颜色，使用`Font`来设置组件的字体等等。
3. 在`Spring`框架中，装饰器模式被广泛应用于实现`AOP`。`AOP`通过代理模式和装饰器模式实现。`JDK`动态代理和`CGLIB`动态代理两种方式实现代理模式，使用装饰器模式对目标对象进行包装，从而实现通知(`Advice`)的织入。例如，可以使用`@Transactional`来添加事务处理的功能，使用`@Cacheable`来添加缓存处理的功能，等等。