设计模式

Ning Ding

February 2024

1 Design Pattern 设计模式

1.1 Creational Patterns 创建性模式

These patterns provide mechanisms for creating objects in a way that's flexible and appropriate to the situation.
这些模式提供了一些灵活的、适应不同场景的对象创建机制。

- 工厂模式(Factory Pattern): create objects without having to directly specify their concrete class
- 单例模式(Singleton Pattern): ensures only one instance of a class exists
- 原型模式(Builder Pattern): simplifies the construction of complex objects

1.2 Structural Patterns 结构式模式

Focus on how classes and objects are put together to form larger structures or relationships.
关注如何将类和对象组合成更大的结构或关系。

- 适配器模式(Adaptor Pattern): Lets you use classes with incompatible interfaces to work together. e.g. Converting an API that returns XML data into one that works with JSON objects.
  让具有不兼容接口的类可以协同工作。
- 装饰器模式(Decorator Pattern): Lets you dynamically add new behavior (responsibilities) to objects without changing the underlying objects themselves.
  允许你动态地为对象添加新的行为（职责），而不需要改变对象本身。
- 代理模式(Proxy Pattern): Provides a stand-in or placeholder for another object. It controls access to the original object, potentially adding extra functionality in the process.
  为另一个对象提供一个替身或占位符。它控制对原始对象的访问，并在过程中有可能添加额外的功能。

1.3 Behavioral Patterns 行为性模式

Address communication and interaction between objects, and ways to distribute responsibilities.
解决对象之间的通信和交互，以及如何分配职责。

- 策略模式(Strategy Pattern): Defines a family of algorithms, encapsulates each one, and makes them interchangeable. The strategy pattern lets the algorithm vary independently from clients that use it. Think of it like choosing different sorting algorithms (bubble sort, quick sort, etc.) for the same dataset.
  定义一系列的算法，把它们一个个封装起来，并且使它们可以相互替换。策略模式可以让算法独立于使用它的客户端发生变化。

- 模板模式(Template Pattern): Defines the basic skeleton or outline of an algorithm in a class, but lets subclasses provide specific steps of the algorithm.
  在一个类中定义一个算法的骨架，但是让子类来提供算法的具体步骤。

- 委派模式(Delegate Pattern): A way for an object to outsource some of its responsibilities to a helper object, known as the delegate. Think of a manager delegating tasks to an assistant.
  一种让对象将它的一些职责外包给一个助手对象（称之为代理）的方式。

- 观察者模式(Observer Pattern): A one-to-many dependency between objects. One object (the "subject") maintains a list of its dependents (the "observers"). When the subject changes its state, it automatically notifies all of its observers. Think of a news publisher and its subscribers.
  对象间一对多的依赖关系。一个对象（"主题"）维护一个它的依赖者（"观察者"）的列表。当主题的状态发生改变时，它会自动通知所有的观察者。

2 Factory Pattern 工厂模式

该模式通过向工厂传递类型来指定要创建的对象，它在创建对象时提供了一种封装机制，将实际创建对象的代码与使用代码分离。
The Factory Pattern is a creational design pattern that provides a centralized way to create objects, especially when the specific type of object to be created isn't known in advance. It involves a "factory" that handles the object creation logic.

2.1 Simple Factory Pattern

简单工厂模式提供了一种不需要直接暴露具体实例化逻辑就能创建对象的方法。这提升了代码的灵活性和易维护性。

```java
// The common blueprint
interface Shape {
   void draw();
}

// Concrete shapes
class Circle implements Shape { ... }
class Rectangle implements Shape { ... }

// Simple Factory
class ShapeFactory {
    public Shape getShape(String shapeType) {
        // Logic to decide which shape to create
        if (shapeType.equalsIgnoreCase("Circle")) {
            return new Circle();
        } else if (shapeType.equalsIgnoreCase("Rectangle")) {
            return new Rectangle();
        }
        return null;
    }
}

//Client calling the factory
class DrawingApplication {
    public static void main(String[] args) {
        ShapeFactory shapeFactory = new ShapeFactory();

        // Get a Circle object
        Shape circle = shapeFactory.getShape("Circle");
        circle.draw();  // Output: Drawing a Circle

        // Get a Rectangle object
        Shape rectangle = shapeFactory.getShape("Rectangle");
        rectangle.draw(); // Output: Drawing a Rectangle

        // Try to get a different shape
        Shape triangle = shapeFactory.getShape("Triangle");  // Might return
   null if not supported
    }
}
```

2.1.1 使用场景

- 工厂类负责创建的对象较少
- 客户端只需要传入工厂类的参数，对于如何创建对象的逻辑不需要关心。

2.1.2 优点

- 只需要传入一个正确的参数，就可以获取你所需要的对象，无须知道其创建的细节。

2.1.3 缺点

- 工厂类的职责相对过重，增加新的产品时需要修改工厂类的判断逻辑，违背开闭原则
- 不易于扩展过于复杂的产品结构

2.2 Factory Method Pattern 工厂方法模式

Provides a way to create objects without directly specifying their exact concrete class. This means you can decide which type of object to create at runtime.
定义一个创建对象的接口，但让实现这个接口的类来决定实例化哪个类，工厂方法让类的实例化推迟到子类中进行。

将职能继续拆分，对工厂本身也做一个抽象，专门的类由专门的工厂创建。

```java
// Product
interface Shape {
    void draw();
}

// Concrete Products
class Circle implements Shape { ... }
class Rectangle implements Shape { ... }

// Creator
```

```java
abstract class ShapeFactory {
    public abstract Shape createShape(String shapeType);
}

// Concrete Creator
class CircleFactory extends ShapeFactory {
    @Override
    public Shape createShape(String shapeType) {
       if ("Circle".equalsIgnoreCase(shapeType)) {
           return new Circle();
       }
       // Add cases for other shapes
       return null;
   }
}
```

2.2.1 使用场景

主要解决产品扩展的问题，在简单工厂中，随着产品链的丰富，工厂的职责会越来越多，不方便维护，即可使用工厂方法模式进一步拆分。

2.2.2 优点

- 用户只需关心所需产品对应的工厂，无需关心创建细节
- 加入新产品符合开闭原则，提高了系统的可扩展性

2.2.3 缺点

- 类的个数容易过多，增加了代码结构的复杂度。
- 增加了系统的抽象性和理解难度。

2.3 Abstract Factory Pattern 抽象工厂模式

The Abstract Factory pattern provides a way to create families of related or dependent objects without having to know their concrete classes. Think of it as a factory that produces other factories.
提供一个创建一系列相关或相互依赖对象的接口，无需指定他们具体的类。

```java
// Abstract Products
interface Chair { ... }
interface Table { ... }

// Concrete Products
class ModernChair implements Chair { ... }
class VictorianChair implements Chair { ... }
class ModernTable implements Table { ... }
class VictorianTable implements Table { ... }

// Abstract Factory: An interface or abstract class that declares methods for creating
    each product type within a family
interface FurnitureFactory {
    Chair createChair();
    Table createTable();
}
```

```java

// Concrete Factories:  Classes that implement the Abstract Factory interface, each
    tailored to produce a specific family of products
class ModernFurnitureFactory implements FurnitureFactory { ... }
class VictorianFurnitureFactory implements FurnitureFactory { ... }
```

2.3.1 使用场景

在工厂方法模式的基础上，想要创建一组相关或依赖的对象时。

2.3.2 优点

同工厂方法模式

2.3.3 缺点

同工厂方法模式

2.4 工厂方法和抽象工厂的区别

工厂方法模式的工厂是创建出一种产品，而抽象工厂是创建出一类产品。

- 一类的产品我们称之为产品族。
- 产品的继承结构称之为产品等级。

在工厂方法模式中，同一产品等级下的每种不同产品都需要一个工厂创建。而在抽象工厂模式中，同一产品等级下的每一产品族（每一系列）的产品都可以以一个工厂创建（每个工厂可以创建多个具有共同属性的产品）。

3 Singleton Pattern 单例模式

The Singleton pattern ensures that there is only one single instance of a class throughout your entire application and provides a global way to access that instance.
指确保一个类在任何情况下都绝对只有一个实例，并提供一个全局访问点。隐藏其所有的构造方法。

3.1 Eager Initialization 饿汉式单例

The single instance is created immediately when the class is loaded, even before anyone asks for it.
类被加载的时候就立刻创建这个类的唯一实例，无论之后是否会真正用到这个实例。

```java
class EagerSingleton {
    private static final EagerSingleton instance = new EagerSingleton();

    private EagerSingleton() {} // Private constructor that can only be called by
    this class

    public static EagerSingleton getInstance() {
        return instance;
    }
}

public class Main {
    public static void main(String[] args) {
        // Get the single instance of the EagerSingleton
        EagerSingleton singleton = EagerSingleton.getInstance();
    }
}
```

the `EagerSingleton` class is created (more accurately, it's loaded) at the very beginning of your program's execution, even before the main function is called.

This is because the JVM loads the class containing your main function, then examines the main class to see what other classes it depends on. Since the `EagerSingleton` class has a static field (instance) that's initialized immediately, the JVM must load the `EagerSingleton` class at this point to create that instance.

3.1.1 优点

- 执行效率高，性能高
- 由于在类加载的时候就创建了实例，所以不存在多线程竞争的问题

3.1.2 缺点

- 浪费资源：即使这个单例对象可能永远不会被用到，它也会被创建，可能会造成资源浪费。

3.2 Lazy Initialization 懒汉式单例

被外部类调用时才创建实例

```java
public class LazySingleton {
    private static LazySingleton instance = null;

    private LazySingleton() {} // Private constructor

    public static LazySingleton getInstance() {
        if (instance == null) {
            instance = new LazySingleton();   // Create on first use
        }
        return instance;
    }
}
```

3.2.1 优点

只有此类被使用的时候创建实例，节省内存

3.2.2 缺点

线程不安全：如果此类同时被两个进程调用，有可能同时创建两个实例

3.2.3 Double Check Lock

降低锁对性能影响的情况下使线程安全

```java
class DclSingleton {
    private static volatile DclSingleton instance = null;

    // ...

    public static DclSingleton getInstance() {
        if (instance == null) {                    // First Check (Unsynchronized):  A quick check if the instance is already created.  If it is, most threads can return it without entering the expensive synchronized block.
            synchronized (DclSingleton.class) { // Lock
                if (instance == null) {        // Second check:  We check again within the synchronized block to ensure another thread didn't already create the instance while we waited for the lock.
                    instance = new DclSingleton();
                }
            }
        }
        return instance;
    }
}
```

3.3 Bill Pugh Singleton(Lazy Static Inner class)

An Improvement over the lazy initialization that is thread safe without lock.

```java
public class Logger {
    private Logger() {
        // private constructor:  Only this class may construct its instance
    }

    // static inner class - inner classes are not loaded until they are referenced.
    private static class LoggerHolder {
        private static Logger logger = new Logger();
    }

    // global access point
    public static Logger getInstance() {
        return LoggerHolder.logger;
    }

    //Other methods
}
```

可以通过反射强行获取到私有构造方法并初始化两次

```java
Class<?> clazz = LazyStaticInnerClassSingleton.class;

Constructor c = clazz.getDeclaredConstructor(null);
c.setAccessible(true);

Object instance1 = c.newInstance();
Object instance2 = c.newInstance();
```

可以在构造时检测一下是否已经存在实例，如果有即报错。

3.4 Registry of Singletons 注册式单例

每一个实例都登记到一个地方，使用唯一的标识获取实例。

3.4.1 Enum Singleton 枚举式单例

```java
enum EnumSingleton {
    INSTANCE;  // Single enum value implicitly created

    public void doSomething() {
        System.out.println("Doing something in the Enum Singleton!");
    }
}

EnumSingleton.INSTANCE.doSomething(); // Accessing the Singleton instance
```

枚举(Enum)的特性就保证每个枚举值只会有一个单一实例，这是由JVM 所保证的。枚举天生线程安全，不需要额外的同步处理, 也不能通过反序列化破坏。

序列化就是把内存中的状态通过转换成字节码的形式，从而转换一个I/O流，写入其他地方（磁盘，网络），反序列化就是将已经持久化的字节码内容转换为I/O流，通过I/O流的读取，进而将读取的内容转换为java对象。在转换中会重新创建对象new。

枚举式单例模式在静态代码块中就给INSTANCE进行了赋值，是饿汉(Eager Initialization)单例模式的实现。

3.5 容器式单例

适用于需要大量创建单例对象的场景，便于管理。但它是非线程安全的。

```java
public class ContainerSingleton {
    private ContainerSingleton(){}

    private static Map<String,Object> ioc = new ConcurrentHashMap<>();

    public static Object getBean(String className){
            if (!ioc.containsKey(className)){
                Object obj = null;
                try{
                    obj = Class.forName(className).newInstance();
                    ioc.put(className,obj);
                }catch (Exception e){
                    e.printStackTrace();
                }
                return obj;
            }else {
                return ioc.get(className);
            }
        }
}
```

3.6 Threadlocal单例

ThreadLocal不能保证其创建的对象是全局唯一的，但是能保证在单个线程中式唯一的，天生是线程安全的。

```java
public class ThreadLocalSingleton {
  private ThreadLocalSingleton() {}

  private static final ThreadLocal<ThreadLocalSingleton> threadLocal =
   ThreadLocal.withInitial(ThreadLocalSingleton::new);

  public static ThreadLocalSingleton getInstance(){
    return threadLocal.get();
  }
}
```

ThreadLocal将所有的对象全部放在ThreadLocalMap中，为每个线程都提供一个对象，实际上是以空间换时间来实现线程隔离的。

4 Proxy Pattern 代理模式

It introduces a middleman (the proxy) to control access to another object (the real subject). Think of it like a gatekeeper or a representative.
指为其他对象提供一种代理，以控制对这个对象的访问。在某些情况下，一个对象不适合或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。使用代理主要有两个目的：保护对象或增强目标对象。

在代码中，一般代理会被理解为代码增强，实际上就是在源代码逻辑前后增加一些代码逻辑，而使调用者无感知。分为静态代理和动态代理。
代理模式一般包含三种角色：

- 抽象主题角色(Subject)：主要职责是声明真实主题与代理的共同接口方法
- 真实主题角色(Real Subject)：也被称为被代理类定义了代理所表示的真实对象，是负责执行系统真正的逻辑业务对象
- 代理主题角色(Proxy)：也被称为代理类，其内部持有Real Subject的引用，因此具备完全的对Real Subject的代理权。客户端调用代理对象的方法，同时也调用被代理对象的方法，但是会在代理对象前后增加一些处理代码。

4.1 静态代理

这种代理方式需要代理对象和目标对象实现一样的接口。

```java
// Common interface
interface Image {
    void display();
}

// Real Subject
class RealImage implements Image {
    private String filename;

    public RealImage(String filename) {
        this.filename = filename;
        loadImage();
    }

    private void loadImage(){
        System.out.println("Loading image: " + filename);
    }

    @Override
    public void display() {
        System.out.println("Displaying: " + filename);
    }
}

```

```java
// Proxy
class ImageProxy implements Image {
    private RealImage realImage;
    private String filename;

    public ImageProxy(String filename) {
        this.filename = filename;
    }

    @Override
    public void display() {
        if (realImage == null) {
            realImage = new RealImage(filename);
        }
        realImage.display();
    }
}
```

4.1.1 优点

可以在不修改目标对象的前提下扩展目标对象的功能。

4.1.2 缺点

- 冗余。由于代理对象要实现与目标对象一致的接口，会产生过多的代理类。
- 不易维护。一旦接口增加方法，目标对象与代理对象都要进行修改。
- 在客户端与目标中增加一个代理对象，会导致请求速度变慢。

4.2 动态代理(使用JDK Proxy)

动态代理通常借助于Java 的反射机制实现，要求目标对象必须实现接口。

静态代理与动态代理的区别主要在:

- 静态代理在编译时就已经实现，编译完成后代理类是一个实际的class文件
- 动态代理是在运行时动态生成的，即编译完成后没有实际的class文件，而是在运行时动态生成类字节码，并加载到JVM中

```java
import java.lang.reflect.Proxy;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

// Image接口
interface Image {
    void display();
}

// 真实对象
class RealImage implements Image {
    // ...
}

// InvocationHandler实现
class ImageInvocationHandler implements InvocationHandler {
    private Object target;

    public ImageInvocationHandler(Object target) {
        this.target = target;
    }

    // 在代理实例上处理方法调用并返回结果。
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
   Throwable {
    //在代理对象前后增加一些处理代码
        System.out.println("before: " + method.getName());//would output
   "display" when image.display() is called
        Object result = method.invoke(target, args);
        System.out.println("after");
        return result;
    }
}

//返回一个指定接口的代理类实例，该接口可以将方法调用指派到指定的调用处理程序。
Image image = (Image) Proxy.newProxyInstance(RealImage.class.getClassLoader
   (), RealImage.class.getInterfaces(), new ImageInvocationHandler(new
   RealImage("test.jpg")));

image.display();
```

4.2.1 工作原理

1. 接口：目标对象（也就是要被代理的对象）必须实现一个或多个接口。动态代理会基于这些接口生成代理对象。
2. Invocation Handler: 你需要创建一个`InvocationHandler` 的实现类。这个类负责处理对代理对象的所有方法调用，它可以灵活地决定是在调用前做额外处理、调用目标对象的方法、还是在调用后进行处理。
3. 生成代理对象: 使用Java 内置的`Proxy` 类可以动态生成代理对象。`Proxy.newProxyInstance` 方法需要三个参数:
   - 目标对象的类加载器
   - 目标对象所实现的接口数组
   - 你的InvocationHandler 实现

4.3 cglib代理

CGLIB is a powerful bytecode generation library used in Java. Its proxy mechanism provides a way to dynamically create proxies at runtime by subclassing a target class.

1. Beyond Interfaces: Unlike Java's standard dynamic proxies, which require the target object to implement interfaces, CGLIB can create proxies for classes that don't have any interfaces.
2. Subclassing: CGLIB dynamically generates a subclass of your target class and overrides its methods. This subclass is the proxy object.

3. Intercepting Calls: When you call methods on the proxy object, CGLIB intercepts those calls and lets you add your own behavior before, after, or instead of the regular method execution.

```java
//Create an Interceptor
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

public class ImageMethodInterceptor implements MethodInterceptor {
    private Object target;

    public ImageMethodInterceptor(Object target) {
        this.target = target;
    }

    @Override
    public Object intercept(Object obj, Method method, Object[] args,
   MethodProxy proxy) throws Throwable {
        System.out.println("before: " + method.getName());
        Object result = proxy.invokeSuper(obj, args); // Important!!
        System.out.println("after");
        return result;
    }
}

//Generate Proxy with CGLIB
import net.sf.cglib.proxy.Enhancer;

// ...
Enhancer enhancer = new Enhancer();
enhancer.setSuperclass(RealImage.class); // Target class
enhancer.setCallback(new ImageMethodInterceptor(new RealImage("test.jpg")));
Image image = (Image) enhancer.create();
image.display();
```

4.3.1 CGLib和JDK动态代理对比

1. JDK动态代理实现了被代理对象的接口，CGLib代理继承了被代理对象
2. JDK动态代理和CGLib代理都在运行期间生成字节码，JDK动态代理直接写Class字节码，CGLib代理使用ASM框架写Class字节码，CGLib代理实现更复杂，生成代理类比JDK动态代理效率低。
3. JDK动态代理调用代理方法是通过反射机制调用的，CGLib代理是通过FastClass机制直接调用方法的，CGLib代理的执行效率更高

4.4 JDK动态代理实现原理

JDK动态代理生成对象的步骤如下：

1. 获取被代理对象的引用，并且获取它的所有接口，反射获取
2. JDK动态代理类重新生成一个新的类，同时新的类要实现被代理类实现的所有接口
3. 动态生成Java代码，新加的业务逻辑方法由一定的逻辑代码调用
4. 编译新生成的Java代码.class文件
5. 重新加载到JVM中运行