设计模式

简介

设计模式（Design pattern）代表了最佳的实践，通常被有经验的面向对象的软件开发人员所采用。设计模式是软件开发人员在软件开发过程中面临的一般问题的解决方案。这些解决方案是众多软件开发人员经过相当长的一段时间的试验和错误总结出来的。

设计模式是一套被反复使用的、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了重用代码、让代码更容易被他人理解、保证代码可靠性。 毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。项目中合理地运用设计模式可以完美地解决很多问题，每种模式在现实中都有相应的原理来与之对应，每种模式都描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是设计模式能被广泛应用的原因。

使用

设计模式在软件开发中的两个主要用途。

开发人员的共同平台

设计模式提供了一个标准的术语系统，且具体到特定的情景。例如，单例设计模式意味着使用单个对象，这样所有熟悉单例设计模式的开发人员都能使用单个对象，并且可以通过这种方式告诉对方，程序使用的是单例模式。

最佳的实践

设计模式已经经历了很长一段时间的发展，它们提供了软件开发过程中面临的一般问题的

最佳解决方案。学习这些模式有助于经验不足的开发人员通过一种简单快捷的方式来学习软件设计。

设计模式的类型

根据设计模式的参考书 **Design Patterns - Elements of Reusable Object-Oriented Software（中文译名：设计模式 - 可复用的面向对象软件元素）** 中所提到的，总共有 23 种设计模式。这些模式可以分为三大类：创建型模式（Creational Patterns）、结构型模式（Structural Patterns）、行为型模式（Behavioral Patterns）。当然，我们还会讨论另一类设计模式：J2EE 设计模式。

序号	模式 & 描述	包括
1	创建型模式这些设计模式提供了一种在创建对象的同时隐藏创建逻辑的方式，而不是使用 new 运算符直接实例化对象。这使得程序在判断针对某个给定实例需要创建哪些对象时更加灵活。	工厂模式（Factory Pattern） 抽象工厂模式（Abstract FactoryPattern） 单例模式（Singleton Pattern） 建造者模式（Builder Pattern） 原型模式（Prototype Pattern）
2	结构型模式这些设计模式关注类和对象的组合。继承的概念被用来组合接口和定义组合对象获得新功能的方式。	适配器模式（Adapter Pattern） 桥接模式（Bridge Pattern） 组合模式（Composite Pattern） 装饰器模式（Decorator Pattern） 外观模式（Facade Pattern） 享元模式（Flyweight Pattern） 代理模式（Proxy Pattern）
3	行为型模式这些设计模式特别关注对象之间的通信。	责任链模式（Chain of Responsibility Pattern） 命令模式（Command Pattern） 解释器模式（Interpreter Pattern） 迭代器模式（Iterator Pattern） 中介者模式（Mediator Pattern） 备忘录模式（Memento Pattern） 观察者模式（Observer Pattern） 状态模式（State Pattern） 策略模式（Strategy Pattern） 模板模式（Template Pattern） 访问者模

		式（Visitor Pattern）

下面用一个图片来整体描述一下设计模式之间的关系：

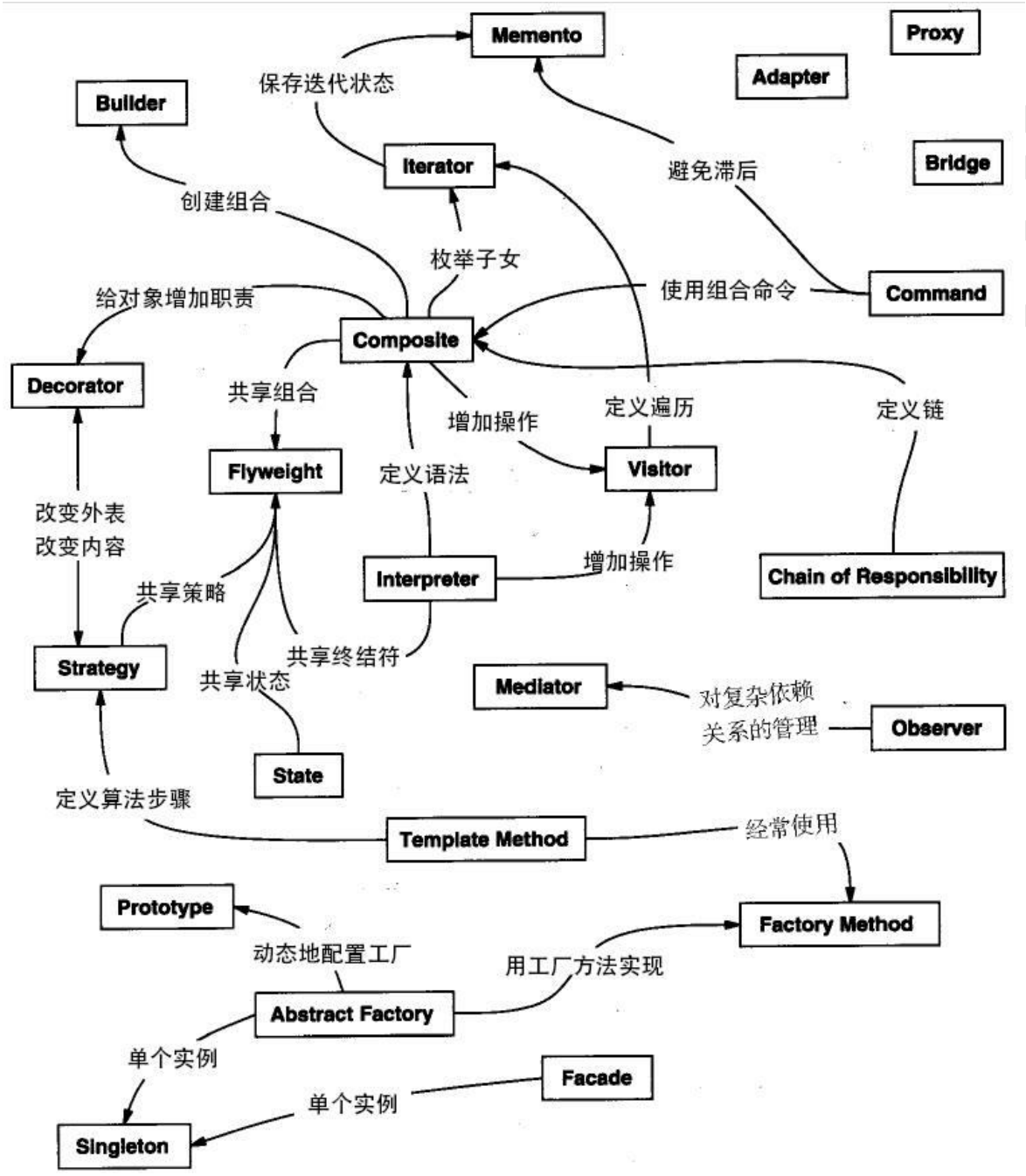

图　设计模式之间的关系

设计模式的六大原则

1. 设计模式之单一职责原则（Open Close Principle）

一个类只负责一项职责，不要存在 1 个以上导致类发生变更的原因。

- 优点： a. 降低类的复杂度，一个类只负责一项职责，逻辑简单清晰； b. 类的可读性，系统的可维护性更高； c. 因需求变更引起的风险更低，降低对其它功能的影响。

- 总结： 只有逻辑足够简单，才可以在代码级别上违反单一职责原则； 只有类中方法数量足够少，才可以在方法级别上违反单一职责原则； 模块化的程序设计以及在员工工作安排上面，都适用单一职责原则。

2. 设计模式之里式替换原则（Liskov Substitution Principle）

子类可以扩展父类的功能，不能改变父类原有的功能，子类可以替换父类，方法或者行为也没有改变

- 注意： a. 子类可以实现父类的抽象方法，但不能覆盖父类的非抽象方法； b. 子类中可以增加自己特有的方法； c. 当子类的方法重载父类的方法时，方法的前置条件（形参）要比父类方法更宽松； d. 当子类的方法实现父类的抽象方法时，方法的后置条件（返回值）要比父类更严格。

3. 设计模式之依赖倒置原则（Dependence Inversion Principle）

高层模块不应该依赖低层模块，二者都应该依赖其抽象 抽象不应该依赖细节，细节应该依赖抽象

- 理解： a. 相对于细节的多变性，抽象的东西要稳定的多，以抽象为基础搭建起来的架构比以细节为基础搭建起来的架构要稳定的多。这里，抽象指的是接口或者抽象类，细节就是具体的实现类，使用抽象类或者接口的目的是，制定好规范和契约，不去涉及任何具体的操作，把展现细节的任务交给实现类来完成；

  b. 依赖倒置原则的核心思想是面向接口编程，达到解耦的过程。

- 注意： a. 底层模块尽量都要有抽象类或者接口； b. 变量的声明类型尽量是抽象类或接口； c. 使用继承时遵循里式替换原则。

4. 设计模式之接口隔离原则（Interface Segregation Principle）

客户端不应该依赖它不需要的接口 一个类对另一个类的依赖应该建立在最小的接口上面

- 理解： a. 建立单一接口，尽量细化接口，接口中的方法尽量少； b. 为单个类建立专用的接口，不要包含太多； c. 依赖几个专用的接口要比依赖一个综合的接口更灵活，提高系统的灵活性和可维护性。

- 注意： a. 接口尽量小，但是要有限度，过小则导致接口数量过多，设计复杂化； b. 为依赖接口的类定制服务，只暴露给调用类需要的方法，建立最小的依赖关系； c. 提高内聚，减少对外交互，用最少的方法去完成最多的事情。

- 和单一职责原则的对比： a. 单一职责原则注重的是职责，而接口隔离原则注重对接口依赖的隔离； b. 单一职责原则主要是约束类，其次才是接口和方法，它针对的是程序中的实现和细节；而接口隔离原则主要约束接口，针对抽象和程序整体框架的构建。

5. 设计模式之迪米特法则（Demeter Principle）

迪米特法则在于降低类之间的耦合，每个类尽量减少对其他类的依赖，尽量减少对外暴露的方法，使得功能模块独立，低耦合

- 理解： a. 只直接的朋友交流（成员变量、方法的输入输出参数中的类）； b. 减少对朋友的理解（减少一个类对外暴露的方法）。
- 注意： a. 虽然可以避免和非直接的类通信，但是要通信，必然会通过一个"中介"来发生联系，过分的使用迪米特原则，会产生大量的中介和传递类，导致系统复杂度变高。

6. 设计模式之开闭原则 （Composite Reuse Principle）

软件中的对象（类、模块、函数等）应该对于扩展是开放的，对于修改是封闭的

- 理解： a. 当需求发生变化时，尽量扩展实体的行为来变化，而不是通过修改已有的代码来实现变化； b. 低层模块的变化，必然有高层模块进行耦合，它并不意味着不做任何修改； c. 这个原则比较虚，可以通过具体的设计模式的设计思维去加深理解。

7. 总结

- 单一职责原则告诉我们实现类要职责单一；
- 里氏替换原则告诉我们不要破坏继承体系；
- 依赖倒置原则告诉我们要面向接口编程；

- 接口隔离原则告诉我们在设计接口的时候要精简单一；
- 迪米特法则告诉我们要降低耦合；
- 而开闭原则是总纲，他告诉我们要对扩展开放，对修改关闭。

Spring 中使用的设计模式

1. 工厂模式 ： BeanFactory
2. 装饰器模式：BeanWrapper
3. 代理模式：AopProxy
4. 单例模式：ApplicationContext
5. 委派模式：DispatcherServlet
6. 策略模式：SimpleInstantiationStrategy
7. 适配器模式：HandlerApdapter
8. 模板方法模式：JdbcTemplate
9. 观察者模式：ContextLoaderListener

Spring 四大模块中的典型的设计模式

1. Spring IOC 工厂模式、单例模式、装饰器模式
2. Spring AOP 代理模式、观察者模式

3. Spring MVC 委派模式、适配器模式

4. Spring JDBC 模板方法模式

Spring 中常见设计模式分类

设计模式类型	设计模式	Spring 组件
创建型	工厂模式	BeanFactoty
	单例模式	ApplicationContext
	原型模式	BeanWrapper
结构模式	适配器模式	HandlerApdapter
	装饰器模式	BeanWrapper
	代理模式	AopProxy
行为模式	策略模式	SimpleInstantiationStrategy
	模板模式	JdbcTemplate
	委派模式	DispatcherServlet
	观察者模式	ContextLoaderListener

JDK 中使用的设计模式

1.Singleton（单例）

作用：保证类只有一个实例；提供一个全局访问点

JDK 中体现：

（1）Runtime

（2）NumberFormat

2.Factory（静态工厂）

作用：

（1）代替构造函数创建对象

（2）方法名比构造函数清晰

JDK 中体现：

（1）Integer.valueOf

（2）Class.forName

3.Factory Method（工厂方法）

作用：子类决定哪一个类实例化

JDK 中体现：Collection.iterator 方法

4.Abstract Factory（抽象工厂）

作用：创建某一种类的对象

JDK 中体现：

（1）java.sql 包

（2）UIManager（swing 外观）

5.Builder（构造者）

作用：

（1）将构造逻辑提到单独的类中

（2）分离类的构造逻辑和表现

JDK 中体现：DocumentBuilder(org.w3c.dom)

6.Prototype（原型）

作用：

（1）复制对象

（2）浅复制、深复制

JDK 中体现：Object.clone；Cloneable

7.Adapter（适配器）

作用：使不兼容的接口相容

JDK 中体现：

（1）java.io.InputStreamReader(InputStream)

（2）java.io.OutputStreamWriter(OutputStream)

8.Bridge（桥接）

作用：将抽象部分与其实现部分分离，使它们都可以独立地变化

JDK 中体现：java.util.logging 中的 Handler 和 Formatter

9.Composite（组合）

作用：一致地对待组合对象和独立对象

JDK 中体现：

（1）org.w3c.dom

（2）javax.swing.JComponent#add(Component)

10.Decorator（装饰器）

作用：为类添加新的功能；防止类继承带来的爆炸式增长

JDK 中体现：

（1）java.io 包

（2）java.util.Collections#synchronizedList(List)

11.Façade（外观）

作用：

（1）封装一组交互类，一致地对外提供接口

（2）封装子系统，简化子系统调用

JDK 中体现：java.util.logging 包

12.Flyweight（享元）

作用：共享对象，节省内存

JDK 中体现：

（1）Integer.valueOf(int i)；Character.valueOf(char c)

（2）String 常量池

14.Proxy（代理）

作用：

（1）透明调用被代理对象，无须知道复杂实现细节

（2）增加被代理类的功能

JDK 中体现：动态代理；RMI

15.Iterator（迭代器）

作用：将集合的迭代和集合本身分离

JDK 中体现：Iterator、Enumeration 接口

16.Observer（观察者）

作用：通知对象状态改变

JDK 中体现：

（1）java.util.Observer,Observable

（2）Swing 中的 Listener

17.Mediator（协调者）

作用：用于协调多个类的操作

JDK 中体现：Swing 的 ButtonGroup

18.Template method（模板方法）

作用：定义算法的结构，子类只实现不同的部分

JDK 中体现：ThreadPoolExecutor.Worker

19.Strategy（策略）

作用：提供不同的算法

JDK 中的体现：ThreadPoolExecutor 中的四种拒绝策略

20.Chain of Responsibility（责任链）

作用：请求会被链上的对象处理，但是客户端不知道请求会被哪些对象处理

JDK 中体现：

（1）java.util.logging.Logger 会将 log 委托给 parent logger

（2）ClassLoader 的委托模型

21.Command（命令）

作用：

（1）封装操作，使接口一致

（2）将调用者和接收者在空间和时间上解耦合

JDK 中体现：Runnable；Callable；ThreadPoolExecutor

22.Interpreter（解释器）

作用：用一组类代表某一规则

JDK 中体现：java.util.regex.Pattern