# 咕泡学院 JavaVIP 高级课程教案

# Spring5 源码分析(第 2 版)

# 第一章

# 从现实生活理解 Spring 中

# 常用的设计模式

关于本文档

| | |
|---|---|
| 主题 | 咕泡学院 Java VIP 高级课程教案--Spring5 源码分析（第二版） |
| 主讲 | Tom 老师 |
| 适用对象 | 咕泡学院 Java 高级 VIP 学员及 VIP 授课老师 |
| 源码版本 | Spring 5.0.2.RELEASE |
| IDE 版本 | IntelliJ IDEA 2017.1.4 |

## 一、Spring 中常用的设计模式

1、我们通常说的 23 中经典设计模式可以通过下表一目了然：

| 分类 | 设计模式 |
| --- | --- |
| 创建型 | 工厂方法模式（Factory Method）、抽象工厂模式（Abstract Factory）、建造者模式（Builder）、原型模式（Prototype）、单例模式(Singleton) |
| 结构型 | 适配器模式(Adapter)、桥接模式（Bridge）、组合模式（Composite）、装饰器模式（Decorator）、门面模式（Facade）、享元模式（Flyweight）、代理模式（Proxy） |
| 行为型 | 解释器模式（Interpreter）、模板方法模式（Template Method）、责任链模式（Chain of Responsibility）、命令模式（Command）、迭代器模式（Iterator）、调解者模式（Mediator）、备忘录模式（Memento）、观察者模式（Observer）、状态模式（State）、策略模式（Strategy）、访问者模式（Visitor） |

通常来说，设计模式都是混合使用，不会独立应用。利用穷举法充分理解设计模式的应用场景。在平时的应用中，不是用设计模式去生搬硬套，而是根据具体业务问题需要时借鉴。

2、设计模式在应用中遵循六大原则：

a、开闭原则（Open Close Principle）

开闭原则就是说对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。

b、里氏代换原则（Liskov Substitution Principle）

里氏代换原则(Liskov Substitution Principle LSP)面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。 LSP 是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

c、依赖倒转原则（Dependence Inversion Principle）

这个是开闭原则的基础，具体内容：针对接口编程，依赖于抽象而不依赖于具体。

d、接口隔离原则（Interface Segregation Principle）

这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖，降低耦合。

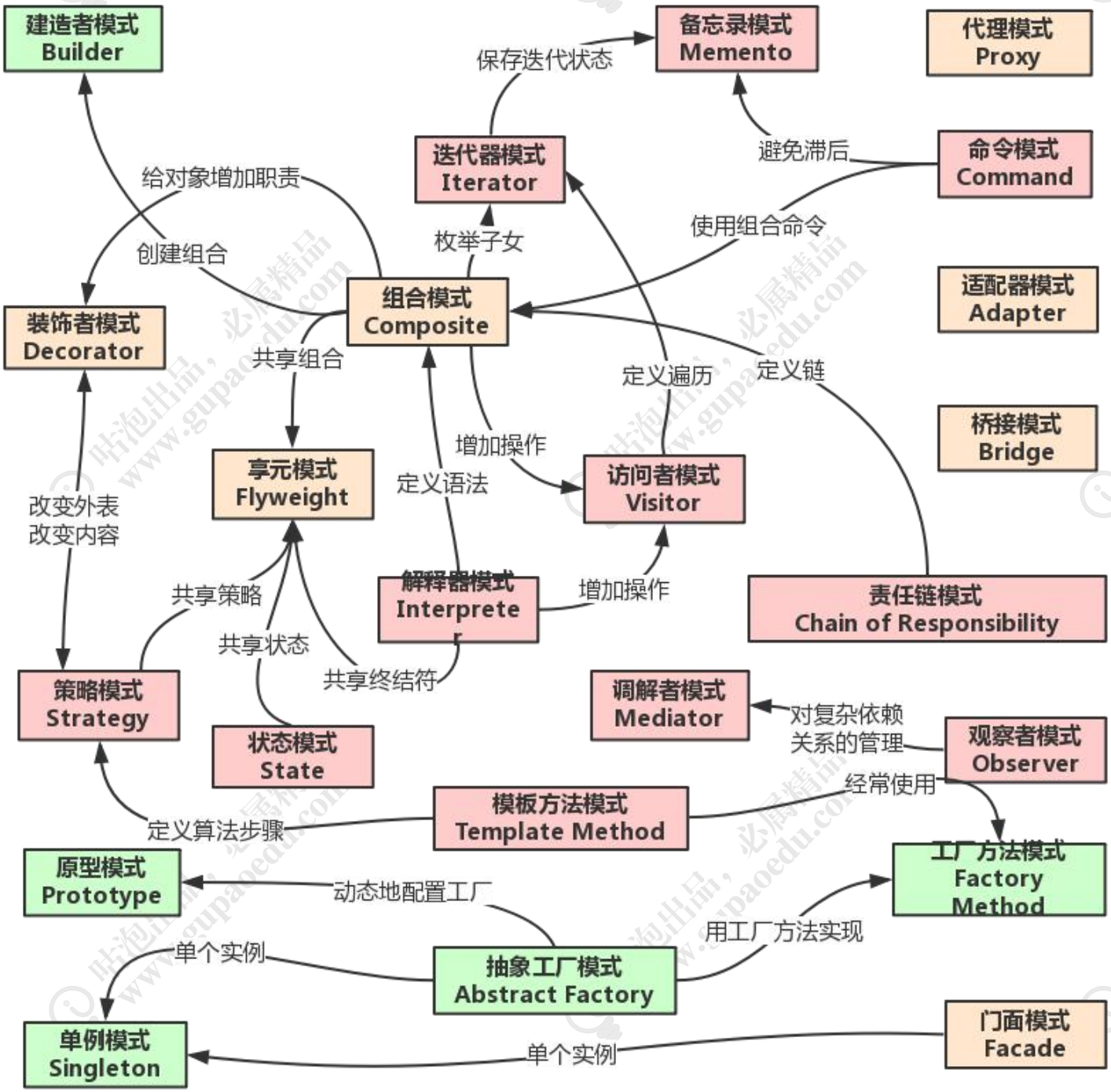
e、迪米特法则（最少知道原则）（Demeter Principle）

为什么叫最少知道原则，就是说：一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。

f、合成复用原则（Composite Reuse Principle）

原则是尽量使用合成/聚合的方式，而不是使用继承。

设计模式之间的关系图




接下来我们只介绍在 Spring 中常用的设计模式。

1.1、简单工厂模式（Factory）

应用场景：又叫做静态工厂方法（StaticFactory Method）模式，但不属于 23 种设计模式之一。

简单工厂模式的实质是由一个工厂类根据传入的参数，动态决定应该创建哪一个产品类。

Spring 中的 BeanFactory 就是简单工厂模式的体现，根据传入一个唯一的标识来获得 Bean 对象，但是否是在传入参数后创建还是传入参数前创建这个要根据具体情况来定。

| 归类 | 特点 | 穷举 |
| --- | --- | --- |
| 创建型模式 | 是复杂工厂模式的思维模型 | 批量生产、标准化 |

### 1.2、工厂方法模式（Factory Method）

应用场景：通常由应用程序直接使用 new 创建新的对象，为了将对象的创建和使用相分离，采用工厂模式,即应用程序将对象的创建及初始化职责交给工厂对象。

一般情况下,应用程序有自己的工厂对象来创建 Bean.如果将应用程序自己的工厂对象交给 Spring 管理,那么 Spring 管理的就不是普通的 Bean,而是工厂 Bean。

| 归类 | 特点 | 穷举 |
| --- | --- | --- |
| 创建型模式 | 对于调用者来说，隐藏了复杂的逻辑处理过程，调用者只关心执行结果。<br>对于工厂来说要对结果负责，保证生产出符合规范的产品。 | 流水线生产 |

### 1.3、单例模式（Singleton）

应用场景：保证一个类仅有一个实例，并提供一个访问它的全局访问点。

Spring 中的单例模式完成了后半句话，即提供了全局的访问点 BeanFactory。但没有从构造器级别去控制单例，这是因为 Spring 管理的是是任意的 Java 对象。 Spring 下默认的 Bean 均为单例。

| 归类 | 特点 | 穷举 |
| --- | --- | --- |
| 创建型模式 | 保证从系统启动到系统终止，全过程只会产生一个实例。<br>当我们在应用中遇到功能性冲突的时候，需要使用单例模式。 | 配置文件、日历、IOC 容器 |

常用单例模式写法：饿汉式、懒汉式、注册式、序列化。

### 1.4、原型模式（Prototype）

应用场景：原型模式就是从一个对象再创建另外一个可定制的对象，而且不需要知道任何创建的细节。

所谓原型模式，就是 Java 中的克隆技术，以某个对象为原型。复制出新的对象。显然新的对象具备原型对象的特点，效率高（避免了重新执行构造过程步骤）。

| 归类 | 特点 | 穷举 |
| --- | --- | --- |
| 创建型模式 | 首先有一个原型。<br>数据内容相同，但对象实例不同（完全两个个体）。 | 孙悟空吹毫毛 |

### 1.5、代理模式（Proxy）

应用场景：为其他对象提供一种代理以控制对这个对象的访问。从结构上来看和 Decorator 模式类似，但 Proxy 是控制，更像是一种对功能的限制，而 Decorator 是增加职责。

Spring 的 Proxy 模式在 AOP 中有体现，比如 JdkDynamicAopProxy 和 Cglib2AopProxy。

| 归类 | 特点 | 穷举 |
| --- | --- | --- |
| 结构型模式 | 执行者、被代理人<br>对于被代理人来说，这件事情是一定要做的，但是我自己又不想做或者没有时间做。<br>对于代理人而言，需要获取到被代理的人个人资料，只是参与整个过程的某个或几个环节。 | 租房中介、售票黄牛、婚介、经纪人、快递、事务代理、非侵入式日志监听 |

### 1.6、策略模式（Strategy）

应用场景：定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。

Spring 中在实例化对象的时候用到 Strategy 模式，在 SimpleInstantiationStrategy 有使用。

| 归类 | 特点 | 穷举 |
| --- | --- | --- |
| 行为型模式 | 最终执行结果是固定的。<br>执行过程和执行逻辑不一样。 | 旅游出行方式 |

### 1.7、模板方法模式（Template Method）

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。Template Method 使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

Template Method 模式一般是需要继承的。这里想要探讨另一种对 Template Method 的理解。Spring 中的 JdbcTemplate，在用这个类时并不想去继承这个类，因为这个类的方法太多，但是我们还是想用到 JdbcTemplate 已有的稳定的、公用的数据库连接，那么我们怎么办呢？我们可以把变化的东西抽出来作为一个参数传入 JdbcTemplate 的方法中。但是变化的东西是一段代码，而且这段代码会用到 JdbcTemplate 中的变量。怎么办？那我们就用回调对象吧。在这个回调对象中定义一个操纵 JdbcTemplate 中变量的方法，我们去实现这个方法，就把变化的东西集中到这里了。然后我们再传入这个回调对象到 JdbcTemplate，从而完成了调用。这就是 Template Method 不需要继承的另一种实现方式。

| 归类 | 特点 | 穷举 |
| --- | --- | --- |
| 行为型模式 | 执行流程固定，但中间有些步骤有细微差别（运行时才确定）。<br>可实现批量生产。 | Spring ORM 数据模型 |

### 1.8、委派模式（Delegate）

应用场景：不属于 23 种设计模式之一，是面向对象设计模式中常用的一种模式。这种模式的原理为类 B 和类 A 是两个互相没有任何关系的类，B 具有和 A 一模一样的方法和属性；并且调用 B 中的方法，属性

就是调用 A 中同名的方法和属性。B 好像就是一个受 A 授权委托的中介。第三方的代码不需要知道 A 的存在，也不需要和 A 发生直接的联系，通过 B 就可以直接使用 A 的功能，这样既能够使用到 A 的各种功能，又能够很好的将 A 保护起来了，一举两得。

| 归类 | 特点 | 穷举 |
| --- | --- | --- |
| 行为型模式 | 要和代理模式区分开来。<br>持有被委托人的引用。<br>不关心过程，只关心结果。 | 经理派发工作任务、Dispatcher |

1.9、适配器模式（Adapter）

Spring AOP 模块对 BeforeAdvice、AfterAdvice、ThrowsAdvice 三种通知类型的支持实际上是借助适配器模式来实现的，这样的好处是使得框架允许用户向框架中加入自己想要支持的任何一种通知类型，上述三种通知类型是 Spring AOP 模块定义的，它们是 AOP 联盟定义的 Advice 的子类型。

| 归类 | 特点 | 穷举 |
| --- | --- | --- |
| 结构型模式 | 注重兼容、转换。<br>适配者与被适配这之间没有层级关系，也没有必然联系。<br>满足 has-a 的关系。 | 编码解码、一拖三充电头、HDMI 转 VGA、Type-C 转 USB |

1.10、装饰器模式（Decorator）

应用场景：在我们的项目中遇到这样一个问题：我们的项目需要连接多个数据库，而且不同的客户在每次访问中根据需要会去访问不同的数据库。我们以往在 Spring 和 Hibernate 框架中总是配置一个数据源，因而 SessionFactory 的 DataSource 属性总是指向这个数据源并且恒定不变，所有 DAO 在使用 SessionFactory 的时候都是通过这个数据源访问数据库。但是现在，由于项目的需要，我们的 DAO 在访问 SessionFactory 的时候都不得不在多个数据源中不断切换，问题就出现了：如何让 SessionFactory 在执行数据持久化的时候，根据客户的需求能够动态切换不同的数据源？我们能不能在 Spring 的框架下通过少量修改得到解决？是否有什么设计模式可以利用呢？

首先想到在 Spring 的 ApplicationContext 中配置所有的 DataSource。这些 DataSource 可能是各种不同类型的，比如不同的数据库：Oracle、SQL Server、MySQL 等，也可能是不同的数据源：比如 Apache 提供的 org.apache.commons.dbcp.BasicDataSource、Spring 提供的 org.springframework.jndi.JndiObjectFactoryBean 等。然后 SessionFactory 根据客户的每次请求，将 DataSource 属性设置成不同的数据源，以到达切换数据源的目的。

Spring 中用到的包装器模式在类名上有两种表现：一种是类名中含有 Wrapper，另一种是类名中含有 Decorator。基本上都是动态地给一个对象添加一些额外的职责。

| 归类 | 特点 | 穷举 |
| --- | --- | --- |
| 结构型模式 | 1、注重覆盖、扩展。<br>2、装饰器和被装饰器都实现同一个接口，主要目的 | IO 流包装、数据源包装、简历包装 |

| | | |
|---|---|---|
| | 是为了扩展之后依旧保留 OOP 关系（同宗同源）。<br>3、满足 is-a 的关系。 | |

1.11、观察者模式（Observer）

应用场景：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

Spring 中 Observer 模式常用的地方是 Listener 的实现。如 ApplicationListener。

| 归类 | 特点 | 穷举 |
|---|---|---|
| 行为型模式 | 一般由两个角色组成：发布者和订阅者（观察者）。<br>观察者通常有一个回调，也可以没有。 | 监听器、日志收集、短信通知、邮件通知 |

1.12、各设计模式对比及编程思想总结

| 设计模式 | 一句话归纳 |
|---|---|
| 工厂模式（Factory） | 只对结果负责，不要三无产品。 |
| 单例模式（Singleton） | 保证独一无二。 |
| 适配器模式（Adapter） | 需要一个转换头（兼容）。 |
| 装饰器模式（Decorator） | 需要包装，但不改变本质(同宗同源)。 |
| 代理模式（Proxy） | 办事要求人，所以找代理。 |
| 观察者模式（Observer） | 完成时通知我。 |
| 策略模式（Strategy） | 我行我素，达到目的就行。 |
| 模板模式（Template） | 流程标准化，原料自己加。 |
| 委派模式（Delegate） | 干活是你的（普通员工），功劳是我的（项目经理）。 |
| 原型模式（Prototype） | 拔一根猴毛，吹出千万个。 |

编程思想总结

| Spring 思想 | 应用场景（特点） | 一句话归纳 |
|---|---|---|
| AOP | Aspect Oriented Programming(面向切面编程)<br>找出多个类中有一定规律的代码，开发时拆开，运行时再合并。<br>面向切面编程，即面向规则编程。 | 解耦，专人做专事。 |
| OOP | Object Oriented Programming（面向对象编程）<br>归纳总结生活中一切事物。 | 封装、继承、多态。 |
| BOP | Bean Oriented Programming（面向 Bean 编程）<br>面向 Bean（普通的 java 类）设计程序。 | 一切从 Bean 开始。 |
| IOC | Inversion of Control（控制反转） | 转交控制权(即控制 |

| | | |
|---|---|---|
| | 将 new 对象的动作交给 Spring 管理，并由 Spring 保存已创建的对象（IOC 容器）。 | 权反转）。 |
| DI/DL | Dependency Injection（依赖注入）或者 Dependency Lookup（依赖查找）<br>依赖注入、依赖查找，Spring 不仅保存自己创建的对象，而且保存对象与对象之间的关系。<br>注入即赋值，主要三种方式构造方法、set 方法、直接赋值。 | 先理清关系再赋值。 |