74 | 总结回顾23种经典设计模式的原理、背后的思想、应用场景等

2020-04-22 王争

讲述：冯永吉
时长 31:03　大小 28.44M

到今天为止，23 种经典的设计模式已经全部讲完了。咱们整个专栏也完成了 3/4，马上就要进入实战环节了。在进入新模块的学习之前，我照例带你做一下总结回顾。23 种经典设计模式共分为 3 种类型，分别是创建型、结构型和行为型。今天，我们把这 3 种类型分成 3 个对应的小模块，逐一带你回顾一下每一种设计模式的原理、实现、设计意图和应用场景。

和之前的总结文一样，今天的内容比较多，有近万字，但都是咱们之前学过的，看起来不会太费劲，但却能检验你是否真的掌握了这些内容。

还是那句话，如果你看了之后，感觉都有印象，那就说明学得还不错；如果还能在脑子里形成自己的知识架构，闭上眼睛都能回忆上来，那说明你学得很好；如果能有自己的理解，并且在项目开发中，开始思考代码质量问题，开始用已经学过的设计模式来解决代码问题，那说明你已经掌握这些内容的精髓。

话不多说，让我们正式开始今天的复习吧！

一、创建型设计模式

创建型设计模式包括：单例模式、工厂模式、建造者模式、原型模式。它主要解决对象的创建问题，封装复杂的创建过程，解耦对象的创建代码和使用代码。

1. 单例模式

单例模式用来创建全局唯一的对象。一个类只允许创建一个对象（或者叫实例），那这个类就是一个单例类，这种设计模式就叫作单例模式。单例有几种经典的实现方式，它们分别是：饿汉式、懒汉式、双重检测、静态内部类、枚举。

尽管单例是一个很常用的设计模式，在实际的开发中，我们也确实经常用到它，但是，有些人认为单例是一种反模式（anti-pattern），并不推荐使用，主要的理由有以下几点：

单例对 OOP 特性的支持不友好

单例会隐藏类之间的依赖关系

单例对代码的扩展性不友好

单例对代码的可测试性不友好

单例不支持有参数的构造函数

那有什么替代单例的解决方案呢？如果要完全解决这些问题，我们可能要从根上寻找其他方式来实现全局唯一类。比如，通过工厂模式、IOC 容器来保证全局唯一性。

有人把单例当作反模式，主张杜绝在项目中使用。我个人觉得这有点极端。模式本身没有对错，关键看你怎么用。如果单例类并没有后续扩展的需求，并且不依赖外部系统，那设计成单例类就没有太大问题。对于一些全局类，我们在其他地方 new 的话，还要在类之间传来传去，不如直接做成单例类，使用起来简洁方便。

除此之外，我们还讲到了进程唯一单例、线程唯一单例、集群唯一单例、多例等扩展知识点，这一部分在实际的开发中并不会被用到，但是可以扩展你的思路、锻炼你的逻辑思维。这里我就不带你回顾了，你可以自己回忆一下。

2. 工厂模式

工厂模式包括简单工厂、工厂方法、抽象工厂这 3 种细分模式。其中，简单工厂和工厂方法比较常用，抽象工厂的应用场景比较特殊，所以很少用到，不是我们学习的重点。

工厂模式用来创建不同但是相关类型的对象（继承同一父类或者接口的一组子类），由给定的参数来决定创建哪种类型的对象。实际上，如果创建对象的逻辑并不复杂，那我们直接通过 new 来创建对象就可以了，不需要使用工厂模式。当创建逻辑比较复杂，是一个“大工程”的时候，我们就考虑使用工厂模式，封装对象的创建过程，将对象的创建和使用相分离。

当每个对象的创建逻辑都比较简单的时候，我推荐使用简单工厂模式，将多个对象的创建逻辑放到一个工厂类中。当每个对象的创建逻辑都比较复杂的时候，为了避免设计一个过于庞

大的工厂类，我们推荐使用工厂方法模式，将创建逻辑拆分得更细，每个对象的创建逻辑独立到各自的工厂类中。

详细点说，工厂模式的作用有下面 4 个，这也是判断要不要使用工厂模式最本质的参考标准。

封装变化：创建逻辑有可能变化，封装成工厂类之后，创建逻辑的变更对调用者透明。

代码复用：□创建代码抽离到独立的工厂类之后可以复用。

隔离复杂性：封装复杂的创建逻辑，调用者无需了解如何创建对象。

控制复杂度：将创建代码抽离出来，让原本的函数或类职责更单一，代码更简洁。

除此之外，我们还讲了工厂模式一个非常经典的应用场景：依赖注入框架，比如 Spring IOC、Google Guice，它用来集中创建、组装、管理对象，跟具体业务代码解耦，让程序员聚焦在业务代码的开发上。DI 框架已经成为了我们平时开发的必备框架，在专栏中，我还带你实现了一个简单的 DI 框架，你可以再回过头去看看。

3. 建造者模式

建造者模式用来创建复杂对象，可以通过设置不同的可选参数，“定制化”地创建不同的对象。建造者模式的原理和实现比较简单，重点是掌握应用场景，避免过度使用。

如果一个类中有很多属性，为了避免构造函数的参数列表过长，影响代码的可读性和易用性，我们可以通过构造函数配合 set() 方法来解决。但是，如果存在下面情况中的任意一种，我们就要考虑使用建造者模式了。

我们把类的必填属性放到构造函数中，强制创建对象的时候就设置。如果必填的属性有很多，把这些必填属性都放到构造函数中设置，那构造函数就又会出现参数列表很长的问题。如果我们把必填属性通过 set() 方法设置，那校验这些必填属性是否已经填写的逻辑就无处安放了。

如果类的属性之间有一定的依赖关系或者约束条件，我们继续使用构造函数配合 set() 方法的设计思路，那这些依赖关系或约束条件的校验逻辑就无处安放了。

如果我们希望创建不可变对象，也就是说，对象在创建好之后，就不能再修改内部的属性值，要实现这个功能，我们就不能在类中暴露 set() 方法。构造函数配合 set() 方法来

设置属性值的方式就不适用了。

4. 原型模式

如果对象的创建成本比较大，而同一个类的不同对象之间差别不大（大部分字段都相同），在这种情况下，我们可以利用对已有对象（原型）进行复制（或者叫拷贝）的方式，来创建新对象，以达到节省创建时间的目的。这种基于原型来创建对象的方式就叫作原型模式。

原型模式有两种实现方法，深拷贝和浅拷贝。浅拷贝只会复制对象中基本数据类型数据和引用对象的内存地址，不会递归地复制引用对象，以及引用对象的引用对象……而深拷贝得到的是一份完完全全独立的对象。所以，深拷贝比起浅拷贝来说，更加耗时，更加耗内存空间。

如果要拷贝的对象是不可变对象，浅拷贝共享不可变对象是没问题的，但对于可变对象来说，浅拷贝得到的对象和原始对象会共享部分数据，就有可能出现数据被修改的风险，也就变得复杂多了。除非操作非常耗时，比较推荐使用浅拷贝，否则，没有充分的理由，不要为了一点点的性能提升而使用浅拷贝。

二、结构型设计模式

结构型模式主要总结了一些类或对象组合在一起的经典结构，这些经典的结构可以解决特定应用场景的问题。结构型模式包括：代理模式、桥接模式、装饰器模式、适配器模式、门面模式、组合模式、享元模式。

1. 代理模式

代理模式在不改变原始类接口的条件下，为原始类定义一个代理类，主要目的是控制访问，而非加强功能，这是它跟装饰器模式最大的不同。一般情况下，我们让代理类和原始类实现同样的接口。但是，如果原始类并没有定义接口，并且原始类代码并不是我们开发维护的。在这种情况下，我们可以通过让代理类继承原始类的方法来实现代理模式。

静态代理需要针对每个类都创建一个代理类，并且每个代理类中的代码都有点像模板式的“重复”代码，增加了维护成本和开发成本。对于静态代理存在的问题，我们可以通过动态代理来解决。我们不事先为每个原始类编写代理类，而是在运行的时候动态地创建原始类对应的代理类，然后在系统中用代理类替换掉原始类。

代理模式常用在业务系统中开发一些非功能性需求，比如：监控、统计、鉴权、限流、事务、幂等、日志。我们将这些附加功能与业务功能解耦，放到代理类统一处理，让程序员只需要关注业务方面的开发。除此之外，代理模式还可以用在 RPC、缓存等应用场景中。

2. 桥接模式

桥接模式的代码实现非常简单，但是理解起来稍微有点难度，并且应用场景也比较局限，所以，相对来说，桥接模式在实际的项目中并没有那么常用，你只需要简单了解，见到能认识就可以了，并不是我们学习的重点。

桥接模式有两种理解方式。第一种理解方式是“将抽象和实现解耦，让它们能独立开发”。这种理解方式比较特别，应用场景也不多。另一种理解方式更加简单，等同于“组合优于继承”设计原则，这种理解方式更加通用，应用场景比较多。不管是哪种理解方式，它们的代码结构都是相同的，都是一种类之间的组合关系。

对于第一种理解方式，弄懂定义中“抽象”和“实现”两个概念，是理解它的关键。定义中的“抽象”，指的并非“抽象类”或“接口”，而是被抽象出来的一套“类库”，它只包含骨架代码，真正的业务逻辑需要委派给定义中的“实现”来完成。而定义中的“实现”，也并非“接口的实现类”，而是的一套独立的“类库”。“抽象”和“实现”独立开发，通过对象之间的组合关系组装在一起。

3. 装饰器模式

装饰器模式主要解决继承关系过于复杂的问题，通过组合来替代继承，给原始类添加增强功能。这也是判断是否该用装饰器模式的一个重要的依据。除此之外，装饰器模式还有一个特点，那就是可以对原始类嵌套使用多个装饰器。为了满足这样的需求，在设计的时候，装饰器类需要跟原始类继承相同的抽象类或者接口。

4. 适配器模式

代理模式、装饰器模式提供的都是跟原始类相同的接口，而适配器提供跟原始类不同的接口。适配器模式是用来做适配的，它将不兼容的接口转换为可兼容的接口，让原本由于接口不兼容而不能一起工作的类可以一起工作。适配器模式有两种实现方式：类适配器和对象适配器。其中，类适配器使用继承关系来实现，对象适配器使用组合关系来实现。

适配器模式是一种事后的补救策略，用来补救设计上的缺陷。应用这种模式算是“无奈之举”。如果在设计初期，我们就能规避接口不兼容的问题，那这种模式就无用武之地了。在实际的开发中，什么情况下才会出现接口不兼容呢？我总结下了下面这 5 种场景：

封装有缺陷的接口设计

统一多个类的接口设计

替换依赖的外部系统

兼容老版本接口

适配不同格式的数据

5. 门面模式

门面模式原理、实现都非常简单，应用场景比较明确。它通过封装细粒度的接口，提供组合各个细粒度接口的高层次接口，来提高接口的易用性，或者解决性能、分布式事务等问题。

6. 组合模式

组合模式跟我们之前讲的面向对象设计中的“组合关系（通过组合来组装两个类）”，完全是两码事。这里讲的“组合模式”，主要是用来处理树形结构数据。正因为其应用场景的特殊性，数据必须能表示成树形结构，这也导致了这种模式在实际的项目开发中并不那么常用。但是，一旦数据满足树形结构，应用这种模式就能发挥很大的作用，能让代码变得非常简洁。

组合模式的设计思路，与其说是一种设计模式，倒不如说是对业务场景的一种数据结构和算法的抽象。其中，数据可以表示成树这种数据结构，业务需求可以通过在树上的递归遍历算法来实现。组合模式，将一组对象组织成树形结构，将单个对象和组合对象都看作树中的节点，以统一处理逻辑，并且它利用树形结构的特点，递归地处理每个子树，依次简化代码实现。

7. 享元模式

所谓“享元”，顾名思义就是被共享的单元。享元模式的意图是复用对象，节省内存，前提是享元对象是不可变对象。

具体来讲，当一个系统中存在大量重复对象的时候，我们就可以利用享元模式，将对象设计成享元，在内存中只保留一份实例，供多处代码引用，这样可以减少内存中对象的数量，以起到节省内存的目的。实际上，不仅仅相同对象可以设计成享元，对于相似对象，我们也可以将这些对象中相同的部分（字段），提取出来设计成享元，让这些大量相似对象引用这些享元。

三、行为型设计模式

我们知道，创建型设计模式主要解决“对象的创建”问题，结构型设计模式主要解决“类或对象的组合”问题，那行为型设计模式主要解决的就是“类或对象之间的交互”问题。行为型模式比较多，有 11 种，它们分别是：观察者模式、模板模式、策略模式、职责链模式、迭代器模式、状态模式、访问者模式、备忘录模式、命令模式、解释器模式、中介模式。

1. 观察者模式

观察者模式将观察者和被观察者代码解耦。观察者模式的应用场景非常广泛，小到代码层面的解耦，大到架构层面的系统解耦，再或者一些产品的设计思路，都有这种模式的影子，比如，邮件订阅、RSS Feeds，本质上都是观察者模式。

不同的应用场景和需求下，这个模式也有截然不同的实现方式：有同步阻塞的实现方式，也有异步非阻塞的实现方式；有进程内的实现方式，也有跨进程的实现方式。同步阻塞是最经典的实现方式，主要是为了代码解耦；异步非阻塞除了能实现代码解耦之外，还能提高代码的执行效率；进程间的观察者模式解耦更加彻底，一般是基于消息队列来实现，用来实现不同进程间的被观察者和观察者之间的交互。

框架的作用有隐藏实现细节，降低开发难度，实现代码复用，解耦业务与非业务代码，让程序员聚焦业务开发。针对异步非阻塞观察者模式，我们也可以将它抽象成 EventBus 框架来达到这样的效果。EventBus 翻译为“事件总线”，它提供了实现观察者模式的骨架代码。我们可以基于此框架非常容易地在自己的业务场景中实现观察者模式，不需要从零开始开发。

2. 模板模式

模板方法模式在一个方法中定义一个算法骨架，并将某些步骤推迟到子类中实现。模板方法模式可以让子类在不改变算法整体结构的情况下，重新定义算法中的某些步骤。这里的“算法”，我们可以理解为广义上的“业务逻辑”，并不特指数据结构和算法中的“算法”。这

里的算法骨架就是“模板”，包含算法骨架的方法就是“模板方法”，这也是模板方法模式名字的由来。

模板模式有两大作用：复用和扩展。其中复用指的是，所有的子类可以复用父类中提供的模板方法的代码。扩展指的是，框架通过模板模式提供功能扩展点，让框架用户可以在不修改框架源码的情况下，基于扩展点定制化框架的功能。

除此之外，我们还讲到回调。它跟模板模式具有相同的作用：代码复用和扩展。在一些框架、类库、组件等的设计中经常会用到，比如 JdbcTemplate 就是用了回调。

相对于普通的函数调用，回调是一种双向调用关系。A 类事先注册某个函数 F 到 B 类，A 类在调用 B 类的 P 函数的时候，B 类反过来调用 A 类注册给它的 F 函数。这里的 F 函数就是“回调函数”。A 调用 B，B 反过来又调用 A，这种调用机制就叫作“回调”。

回调可以细分为同步回调和异步回调。从应用场景上来看，同步回调看起来更像模板模式，异步回调看起来更像观察者模式。回调跟模板模式的区别，更多的是在代码实现上，而非应用场景上。回调基于组合关系来实现，模板模式基于继承关系来实现。回调比模板模式更加灵活。

3. 策略模式

策略模式定义一族算法类，将每个算法分别封装起来，让它们可以互相替换。策略模式可以使算法的变化独立于使用它们的客户端（这里的客户端代指使用算法的代码）。策略模式用来解耦策略的定义、创建、使用。实际上，一个完整的策略模式就是由这三个部分组成的。

策略类的定义比较简单，包含一个策略接口和一组实现这个接口的策略类。策略的创建由工厂类来完成，封装策略创建的细节。策略模式包含一组策略可选，客户端代码选择使用哪个策略，有两种确定方法：编译时静态确定和运行时动态确定。其中，“运行时动态确定”才是策略模式最典型的应用场景。

在实际的项目开发中，策略模式也比较常用。最常见的应用场景是，利用它来避免冗长的 if-else 或 switch 分支判断。不过，它的作用还不止如此。它也可以像模板模式那样，提供框架的扩展点等等。实际上，策略模式主要的作用还是解耦策略的定义、创建和使用，控制代码的复杂度，让每个部分都不至于过于复杂、代码量过多。除此之外，对于复杂代码来

说，策略模式还能让其满足开闭原则，添加新策略的时候，最小化、集中化代码改动，减少引入 bug 的风险。

4. 职责链模式

在职责链模式中，多个处理器依次处理同一个请求。一个请求先经过 A 处理器处理，然后再把请求传递给 B 处理器，B 处理器处理完后再传递给 C 处理器，以此类推，形成一个链条。链条上的每个处理器各自承担各自的处理职责，所以叫作职责链模式。

在 GoF 的定义中，一旦某个处理器能处理这个请求，就不会继续将请求传递给后续的处理器了。当然，在实际的开发中，也存在对这个模式的变体，那就是请求不会中途终止传递，而是会被所有的处理器都处理一遍。

职责链模式常用在框架开发中，用来实现过滤器、拦截器功能，让框架的使用者在不需要修改框架源码的情况下，添加新的过滤、拦截功能。这也体现了之前讲到的对扩展开放、对修改关闭的设计原则。

5. 迭代器模式

迭代器模式也叫游标模式，它用来遍历集合对象。这里说的“集合对象”，我们也可以叫“容器”“聚合对象”，实际上就是包含一组对象的对象，比如，数组、链表、树、图、跳表。迭代器模式主要作用是解耦容器代码和遍历代码。大部分编程语言都提供了现成的迭代器可以使用，我们不需要从零开始开发。

遍历集合一般有三种方式：for 循环、foreach 循环、迭代器遍历。后两种本质上属于一种，都可以看作迭代器遍历。相对于 for 循环遍历，利用迭代器来遍历有 3 个优势：

迭代器模式封装集合内部的复杂数据结构，开发者不需要了解如何遍历，直接使用容器提供的迭代器即可；

迭代器模式将集合对象的遍历操作从集合类中拆分出来，放到迭代器类中，让两者的职责更加单一；

迭代器模式让添加新的遍历算法更加容易，更符合开闭原则。除此之外，因为迭代器都实现自相同的接口，在开发中，基于接口而非实现编程，替换迭代器也变得更加容易。

在通过迭代器来遍历集合元素的同时，增加或者删除集合中的元素，有可能会导致某个元素被重复遍历或遍历不到。针对这个问题，有两种比较干脆利索的解决方案，来避免出现这种不可预期的运行结果。一种是遍历的时候不允许增删元素，另一种是增删元素之后让遍历报错。第一种解决方案比较难实现，因为很难确定迭代器使用结束的时间点。第二种解决方案更加合理，Java 语言就是采用的这种解决方案。增删元素之后，我们选择 fail-fast 解决方式，让遍历操作直接抛出运行时异常。

6. 状态模式

状态模式一般用来实现状态机，而状态机常用在游戏、工作流引擎等系统开发中。状态机又叫有限状态机，它由 3 个部分组成：状态、事件、动作。其中，事件也称为转移条件。事件触发状态的转移及动作的执行。不过，动作不是必须的，也可能只转移状态，不执行任何动作。

针对状态机，我们总结了三种实现方式。

第一种实现方式叫分支逻辑法。利用 if-else 或者 switch-case 分支逻辑，参照状态转移图，将每一个状态转移原模原样地直译成代码。对于简单的状态机来说，这种实现方式最简单、最直接，是首选。

第二种实现方式叫查表法。对于状态很多、状态转移比较复杂的状态机来说，查表法比较合适。通过二维数组来表示状态转移图，能极大地提高代码的可读性和可维护性。

第三种实现方式就是利用状态模式。对于状态并不多、状态转移也比较简单，但事件触发执行的动作包含的业务逻辑可能比较复杂的状态机来说，我们首选这种实现方式。

7. 访问者模式

访问者模式允许一个或者多个操作应用到一组对象上，设计意图是解耦操作和对象本身，保持类职责单一、满足开闭原则以及应对代码的复杂性。

对于访问者模式，学习的主要难点在代码实现。而代码实现比较复杂的主要原因是，函数重载在大部分面向对象编程语言中是静态绑定的。也就是说，调用类的哪个重载函数，是在编译期间，由参数的声明类型决定的，而非运行时，根据参数的实际类型决定的。除此之外，

我们还讲到 Double Disptach。如果某种语言支持 Double Dispatch，那就不需要访问者模式了。

正是因为代码实现难理解，所以，在项目中应用这种模式，会导致代码的可读性比较差。如果你的同事不了解这种设计模式，可能就会读不懂、维护不了你写的代码。所以，除非不得已，不要使用这种模式。

8. 备忘录模式

备忘录模式也叫快照模式，具体来说，就是在不违背封装原则的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，以便之后恢复对象为先前的状态。这个模式的定义表达了两部分内容：一部分是，存储副本以便后期恢复；另一部分是，要在不违背封装原则的前提下，进行对象的备份和恢复。

备忘录模式的应用场景也比较明确和有限，主要用来防丢失、撤销、恢复等。它跟平时我们常说的“备份”很相似。两者的主要区别在于，备忘录模式更侧重于代码的设计和实现，备份更侧重架构设计或产品设计。

对于大对象的备份来说，备份占用的存储空间会比较大，备份和恢复的耗时会比较长。针对这个问题，不同的业务场景有不同的处理方式。比如，只备份必要的恢复信息，结合最新的数据来恢复；再比如，全量备份和增量备份相结合，低频全量备份，高频增量备份，两者结合来做恢复。

9. 命令模式

命令模式在平时工作中并不常用，你稍微了解一下就可以。

落实到编码实现，命令模式用到最核心的实现手段，就是将函数封装成对象。我们知道，在大部分编程语言中，函数是没法作为参数传递给其他函数的，也没法赋值给变量。借助命令模式，我们将函数封装成对象，这样就可以实现把函数像对象一样使用。

命令模式的主要作用和应用场景，是用来控制命令的执行，比如，异步、延迟、排队执行命令、撤销重做命令、存储命令、给命令记录日志等，这才是命令模式能发挥独一无二作用的地方。

10. 解释器模式

解释器模式为某个语言定义它的语法（或者叫文法）表示，并定义一个解释器用来处理这个语法。实际上，这里的“语言”不仅仅指我们平时说的中、英、日、法等各种语言。从广义上来讲，只要是能承载信息的载体，我们都可以称之为“语言”，比如，古代的结绳记事、盲文、哑语、摩斯密码等。

要想了解“语言”要表达的信息，我们就必须定义相应的语法规则。这样，书写者就可以根据语法规则来书写“句子”（专业点的叫法应该是“表达式”），阅读者根据语法规则来阅读“句子”，这样才能做到信息的正确传递。而我们要讲的解释器模式，其实就是用来实现根据语法规则解读“句子”的解释器。

解释器模式的代码实现比较灵活，没有固定的模板。我们前面说过，应用设计模式主要是应对代码的复杂性，解释器模式也不例外。它的代码实现的核心思想，就是将语法解析的工作拆分到各个小类中，以此来避免大而全的解析类。一般的做法是，将语法规则拆分一些小的独立的单元，然后对每个单元进行解析，最终合并为对整个语法规则的解析。

11. 中介模式

中介模式的设计思想跟中间层很像，通过引入中介这个中间层，将一组对象之间的交互关系（或者说依赖关系）从多对多（网状关系）转换为一对多（星状关系）。原来一个对象要跟 n 个对象交互，现在只需要跟一个中介对象交互，从而最小化对象之间的交互关系，降低了代码的复杂度，提高了代码的可读性和可维护性。

观察者模式和中介模式都是为了实现参与者之间的解耦，简化交互关系。两者的不同在于应用场景上。在观察者模式的应用场景中，参与者之间的交互比较有条理，一般都是单向的，一个参与者只有一个身份，要么是观察者，要么是被观察者。而在中介模式的应用场景中，参与者之间的交互关系错综复杂，既可以是消息的发送者、也可以同时是消息的接收者。

课堂讨论

终于学完了这 23 种设计模式，针对这些设计模式，你还有哪些疑问？可以在留言区说一说。

如果有收获，欢迎你收藏这篇文章，反复阅读，并把它分享给你的朋友。

上一篇　73 | 中介模式：什么时候用中介模式？什么时候用观察者模式？

下一篇　加餐一 | 用一篇文章带你了解专栏中用到的所有Java语法

精选留言 (10)

写留言

小晏子
2020-04-22

看完了文章，总结的很赞，产生了一个疑问，23种设计模式这里只提到了22种，创建型4加上结构型7加上行为性11共22种，那么还缺哪种模式呢？原来传统的23种是把抽象工厂单独拎出来算的，这22种加上抽象工厂就是23种设计模式了。

展开

5

强哥
2020-04-22

时间过得好快，不知不觉已经学了74讲，收获满满，感谢作者！

5

忆水寒
2020-04-22

虽然学完了一遍，但是感觉只掌握了60%，还要多回头看看。

2

Heaven
2020-04-22

相比设计思想,设计模式更加具体化,但是在实际开发中,往往更加具体的东西不好套,我在学完之后,感觉最有感受的,实际上是之前的设计原则和思想,很有帮助

展开

1

Geek_27a248
2020-04-22

学完之后一头雾水，只有模糊概念，重头继续学起，，

展开

1

do it
2020-04-22

仍需努力💪💪

展开

1

荀麒睿
2020-04-23

设计模式真的要多复习，还要多思考自己的项目里哪些可以用到，毕竟之前用的比较少，还得不断加强

jaryoung
2020-04-22

没有问题怎么办？看来我的学的还不够好

展开

守拙
2020-04-22

对于创建型, 结构型和行为型模式的定义:

创建型模式提供对象的创建机制, 解耦对象的创建和使用.

结构型模式让对象与对象之间组合成新的结构. ...

展开

张德

2020-04-22

终于更新完了 找个时间得好好看看了

展开