|  |  |  |  |
| --- | --- | --- | --- |
| 范围 | 创建型 | 结构型 | 行为型 |
| 类 | Factory Method（工厂方法） | Adapter(类) （适配器） | Interpreter（解释器）  Template Method（模版方法） |
| 对象 | Abstract Factory（抽象工厂）  Builder（建造者）  Prototype（原型）  Singleton（单例） | Adapter(对象)（适配器）  Bridge（桥接）  Composite（组合）  Decorator（装饰者）  Façade（外观）  Flyweight（享元）  Proxy（代理） | Chain of Responsibility（职责链）  Command（命令）  Iterator（迭代器）  Mediator（中介者）  Memento（备忘录）  Observer（观察者）  State（状体）  Strategy（策略）  Visitor（访问者） |

**设计模式分类**

通常的设计模式可以概括为23种,按照特点可以将其分为三大类型：创建型、结构型、行为型。

**创建型**

创建型模式是用来创建对象的模式，抽象了实例化的过程，帮助一个系统独立于其他关联对象的创建、组合和表示方式。所有的创建型模式都有两个主要功能：

　　1.将系统所使用的具体类的信息封装起来

　　2.隐藏类的实例是如何被创建和组织的。外界对于这些对象只知道他们共同的接口，而不清楚其具体的实现细节。

正因为以上两点，创建型模式在创建什么（what）、由谁来创建（who）、以及何时创建（when）这些方面，都为设计者提供了尽可能大的灵活性。

创建型模式的作用可以概括为：A.封装创建逻辑，不仅仅是new一个对象那么简单。B.封装创建逻辑变化,客户代码尽量不改变和尽量少修改。

创建型模式有以下几种：

　　单例模式（Singleton Pattern）：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,而且自行实例化并向整个系统提供这个实例。

　　工厂方法模式（Factory Pattern）：在工厂方法模式中，工厂类成为了抽象类，实际的创建工作将由其具体子类来完成。工厂的用意是定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类中，它强调的是“单个对象”的变化。

　　抽象工厂模式（Abstract Factory）：抽象工厂是所有工厂模式中最为抽象且最具有一般性的一种形态。抽象工厂可以向客户提供一个接口，是的客户可以在没有指定产品的情况下，创建多个产品族中的产品对象，强调的是“对象系列”的变化。

　　建造者模式（Builder Pattern）：把构造对象实例的逻辑移到了类的内部，在类的外部定义了该类的构造逻辑。它把一个复杂对象的构造过程从对象的表示中分离出来，其直接效果是将一个复杂的对象简化为一个比较简单的目标对象，强调的是产品的构造过程。

　　原型模式（Prototype Pattern）：从一个对象再创建另一个可定制的对象，而且不需知道任何创建的细节。原型模式和工厂模式一样，同样对客户隐藏了对象创建工作具体的实现细节，但与通过对一个类进行实例化的构造心对象不同的是，原型模式通过复制一个现有的对象生成新对象。

**结构型**

　　结构型模式讨论的是类和对象的结构，它采用继承机制来组合接口或实现（类结构型模式），或者通过组合一些对象实现新的功能（对象结构型模式）。这些结构型模式在某些方面具有很大的相似性，但侧重点各有不同。

　　代理模式（Proxy）：为其他对象提供一种代理以控制对该对象的访问。

　　装饰模式（Decorator）：动态的给一个对象添加一些额外的职责。就增加功能来说，装饰模式比生成子类更灵活。

　　适配器模式（Adapter）：将一个类的接口变换成客户端所期待的接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。

　　组合模式（Composite）：也叫合成模式，将对象组合成属性结构一表示“部分-整体”的层次结构，使得用户对单个对象和组成对象的使用具有一致性。

　　桥梁模式（Bridge）：也叫桥接模式，将抽象和实现解耦，是的两者可以独立的变化。

　　外观模式（Facade）：也叫门面模式，要求一个子系统的外部与其内部的通信必须通过一个统一的对象进行，外观模式提供一个高层次的接口，使得子系统更易于使用。

　　亨元模式（Flyweight）：是池技术的重要实现方式，使用共享对象可有效的支持大量的细粒度的对象。

**行为型**

　　行为型设计模式关注的是对象的行为，用来解决对象之间的联系问题。

　　模板方法模式（Template Method）：定义一个操作中的算法的框架，而将一些步骤延迟到子类中，使得子类可以在不改变算法的结构的情况下重新定义该算法的某些特定步骤。

　　命令模式（Command）：是一种高内聚的模式，讲一个请求封装成一个对象，从而使用不同的请求吧客户端参数化。对请求排队或者记录请求日志，可以提供命令的撤销和恢复功能。

　　责任链模式（Chain of Responsibility）：使多个对象都有机会处理请求，从而避免了请求的发送者和接受者之间的耦合关系。将这些对象连城一条链，并沿着这条链传递该请求，直到有对象处理它为止。

　　策略模式（Strategy）：也叫政策模式，定义一组算法，将每个算法都封装起来，并且使他们之间可以互换。

　　迭代器模式（Iterator）：提供一种方法访问一个容器对象中的各个元素，而又不需要暴露该对象的内部细节。

　　中介者模式（Mediator）：用一个中介对象封装一系列对象交互，中介者使各对象不需要显示相互作用，从而使其耦合松散，而且可以独立的改变它们之间的交互。

　　观察者模式（Observer）：也叫发布订阅模式，定义对象间的一对多的依赖关系，是的每当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新。

　　备忘录模式（Memento）：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保持这个状态。

　　访问者模式（Visitor）：封装一些作用于某种数据结构中的各元素的操作，它可以在不改变数据结构的前提下定义作用于这些元素的新的操作。

　　状态模式（State）：当一个对象内在状态改变时允许其改变行为，这个对象看起来像改变了其类型，状态模式的核心是封装，状态的变更会引起行为的变更。

　　解释器模式（Interpreter）：给定一门语言，定义它的文法的一种表示，并定义一个解释器，该解释器使用该文法表示来解释语言中的句子。

**二、设计模式的六大原则**

**总原则：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，而是要扩展原有代码，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等，后面的具体设计中我们会提到这点。

**1、单一职责原则**

单一职责原则（SRP：Single responsibility principle）又称单一功能原则，面向对象五个基本原则（SOLID）之一。它规定一个类应该只有一个发生变化的原因。该原则由罗伯特·C·马丁（Robert C. Martin）于《敏捷软件开发：原则、模式和实践》一书中给出的。马丁表示此原则是基于汤姆·狄马克(Tom DeMarco)和Meilir Page-Jones的著作中的内聚性原则发展出的。 就一个类而言，应该仅有一个引起它变化的原因。不要存在多于一个导致类变更的原因，也就是说每个类应该实现单一的职责，如若不然，就应该把类拆分。

**2、里氏替换原则（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。—— From Baidu 百科

历史替换原则中，子类对父类的方法尽量不要重写和重载。因为父类代表了定义好的结构，通过这个规范的接口与外界交互，子类不应该随便破坏它。

**3、依赖倒转原则（Dependence Inversion Principle）**

这个是开闭原则的基础，具体内容：面向接口编程，依赖于抽象而不依赖于具体。写代码时用到具体类时，不与具体类交互，而与具体类的上层接口交互。

**4、接口隔离原则（Interface Segregation Principle）**

这个原则的意思是：每个接口中不存在子类用不到却必须实现的方法，如果不然，就要将接口拆分。使用多个隔离的接口，比使用单个接口（多个接口方法集合到一个的接口）要好。

**5、迪米特法则（最少知道原则）（Demeter Principle）**

就是说：一个类对自己依赖的类知道的越少越好。也就是说无论被依赖的类多么复杂，都应该将逻辑封装在方法的内部，通过public方法提供给外部。这样当被依赖的类变化时，才能最小的影响该类。

最少知道原则的另一个表达方式是：只与直接的朋友通信。类之间只要有耦合关系，就叫朋友关系。耦合分为依赖、关联、聚合、组合等。我们称出现为成员变量、方法参数、方法返回值中的类为直接朋友。局部变量、临时变量则不是直接的朋友。我们要求陌生的类不要作为局部变量出现在类中。

**6、合成复用原则（Composite Reuse Principle）**

原则是尽量首先使用合成/聚合的方式，而不是使用继承

建造者模式：一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。创建一些复杂的对象时，这些对象的内部组成构件间的建造顺序是稳定的，但是对象的内部组成构件面临着复杂的变化。

装饰模式：装饰模式是为已有功能动态地添加更多功能的一种方式。当系统需要新功能的时侯，是向旧的类中添加新的代码。这些新加的代码通常装饰了原有类的核心职责或主要行为，但这种做法的问题在于，它们在主类中加入新的字段，新的方法，和新的逻辑，从而增加了主类的复杂度，而这些新加入的东西仅 仅是为了满足一些只在某种特定情况下才会执行的特殊行的需要。而装饰模饰模式却提供了一个非常好的解决方案，它把每个要装饰的功能放在单独的类中，并让这个类包装它所要装饰的对象。因此，当需要执行特殊为时，客户代码就可以在运行时根据需要有选择地、按顺序地使用装饰功能包装对象了。

三种工厂模式优缺点解析

简单工厂模式，工厂方法模式和抽象工厂模式都是属于创建型设计模式，这三种创建型模式都不需要知道具体类。我们掌握一种思想，就是在创建一个对象时，需要把容易发生变化的地方给封装起来，来控制变化（哪里变化，封装哪里），以适应客户的变动，项目的扩展。用这三种设计模式都可以实现，那究竟这三种设计模式有什么异同呢？下面根据这三者之间的特点，优点，缺点，适用范围进行比较。

1. 特点

　　简单工厂模式：专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。它又称为静态工厂方法模式。它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（这些产品类继承自一个父类或接口）的实例。简单工厂模式的创建目标，所有创建的对象都是充当这个角色的某个具体类的实例。在这个模式中，工厂类是整个模式的关键所在。它包含必要的判断逻辑，能够根据外界给定的信息，决定究竟应该创建哪个具体类的对象。用户在使用时可以直接根据工厂类去创建所需的实例，而无需了解这些对象是如何创建以及如何组织的。有利于整个软件体系结构的优化。

　　工厂方法模式：工厂方法是粒度很小的设计模式，因为模式的表现只是一个抽象的方法。提前定义用于创建对象的接口，让子类决定实例化具体的某一个类，即在工厂和产品中间增加接口，工厂不再负责产品的创建，由接口针对不同条件返回具体的类实例，由具体类实例去实现。工厂方法模式是简单工厂模式的衍生，解决了许多简单工厂模式的问题。首先完全实现‘开－闭原则’，实现了可扩展。其次实现更复杂的层次结构，可以应用于产品结果复杂的场合。工厂方法模式是对简单工厂模式进行了抽象。有一个抽象的Factory类（可以是抽象类和接口），这个类将不在负责具体的产品生产，而是只制定一些规范，具体的生产工作由其子类去完成。在这个模式中，工厂类和产品类往往可以依次对应。即一个抽象工厂对应一个抽象产品，一个具体工厂对应一个具体产品，这个具体的工厂就负责生产对应的产品。

　　抽象工厂模式：抽象工厂模式是所有形态的工厂模式中最为抽象和最具一般性的一种形态。抽象工厂模式是指当有多个抽象角色时，使用的一种工厂模式。抽象工厂模式可以向客户端提供一个接口，使客户端在不必指定产品的具体的情况下，创建多个产品族中的产品对象。它有多个抽象产品类，每个抽象产品类可以派生出多个具体产品类，一个抽象工厂类，可以派生出多个具体工厂类，每个具体工厂类可以创建多个具体产品类的实例。每一个模式都是针对一定问题的解决方案，工厂方法模式针对的是一个产品等级结构；而抽象工厂模式针对的是多个产品等级结果。

2. 优点

　　简单工厂模式：工厂类含有必要的判断逻辑，可以决定在什么时候创建哪一个产品类的实例，客户端可以免除直接创建产品对象的责任，而仅仅"消费"产品。简单工厂模式通过这种做法实现了对责任的分割。简单工厂模式能够根据外界给定的信息，决定究竟应该创建哪个具体类的对象。通过它，外界可以从直接创建具体产品对象的尴尬局面中摆脱出来。外界与具体类隔离开来，偶合性低。明确区分了各自的职责和权力，有利于整个软件体系结构的优化。

　　工厂方法模式：工厂方法模式是为了克服简单工厂模式的缺点（主要是为了满足OCP）而设计出来的。简单工厂模式的工厂类随着产品类的增加需要增加很多方法（或代码），而工厂方法模式每个具体工厂类只完成单一任务，代码简洁。工厂方法模式完全满足OCP，即它有非常良好的扩展性。

　　抽象工厂模式：抽象工厂模式主要在于应对“新系列”的需求变化。分离了具体的类，抽象工厂模式帮助你控制一个应用创建的对象的类，因为一个工厂封装创建产品对象的责任和过程。它将客户和类的实现分离，客户通过他们的抽象接口操纵实例，产品的类名也在具体工厂的实现中被分离，它们不出现在客户代码中。它使得易于交换产品系列。一个具体工厂类在一个应用中仅出现一次——即在它初始化的时候。这使得改变一个应用的具体工厂变得很容易。它只需改变具体的工厂即可使用不同的产品配置，这是因为一个抽象工厂创建了一个完整的产品系列，所以整个产品系列会立刻改变。它有利于产品的一致性。当一个系列的产品对象被设计成一起工作时，一个应用一次只能使用同一个系列中的对象，这一点很重要，而抽象工厂很容易实现这一点。抽象工厂模式有助于这样的团队的分工，降低了模块间的耦合性，提高了团队开发效率。

3. 缺点

　　简单工厂模式：当产品有复杂的多层等级结构时，工厂类只有自己，以不变应万变，就是模式的缺点。因为工厂类集中了所有产品创建逻辑，一旦不能正常工作，整个系统都要受到影响。系统扩展困难，一旦添加新产品就不得不修改工厂逻辑（如果要增加一个产品，则需要修改工厂类，增加if/else分支，或者增加一个case分支），有可能造成工厂逻辑过于复杂,违背了"开放--封闭"原则(OCP).另外，简单工厂模式通常使用静态工厂方法，这使得无法由子类继承，造成工厂角色无法形成基于继承的等级结构。

　　工厂方法模式：不易于维护，假如某个具体产品类需要进行一定的修改，很可能需要修改对应的工厂类。当同时需要修改多个产品类的时候，对工厂类的修改会变得相当麻烦（对号入座已经是个问题了）。

　　抽象工厂模式：抽象工厂模式在于难于应付“新对象”的需求变动。难以支持新种类的产品。难以扩展抽象工厂以生产新种类的产品。这是因为抽象工厂几乎确定了可以被创建的产品集合，支持新种类的产品就需要扩展该工厂接口，这将涉及抽象工厂类及其所有子类的改变。

4. 适用范围

　　简单工厂模式：工厂类负责创建的对象比较少，客户只知道传入了工厂类的参数，对于始何创建对象（逻辑）不关心。

　　工厂方法模式：当一个类不知道它所必须创建对象的类或一个类希望由子类来指定它所创建的对象时，当类将创建对象的职责委托给多个帮助子类中的某一个，并且你希望将哪一个帮助子类是代理者这一信息局部化的时候，可以使用工厂方法。

　　抽象工厂模式：一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节，这对于所有形态的工厂模式都是重要的。这个系统有多于一个的产品族，而系统只消费其中某一产品族。同属于同一个产品族的产品是在一起使用的，这一约束必须在系统的设计中体现出来。系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于实现。

无论是简单工厂模式、工厂模式还是抽象工厂模式，它们本质上都是将不变的部分提取出来，将可变的部分留作接口，以达到最大程度上的复用。拿一个生产水杯（cup）的工厂举例：起初，不用工厂模式，我必须在生产水杯之前知道水杯的材料和形状等水杯的所有特征才能生产，这就是我们的new Cup();这个Cup必须是具体的。厂主发现同一形状的被子，只是材料不同,如一个是玻璃(glass)的，一个是瓷(china)的,但是确要两条生产线，显然有资源浪费的嫌疑。现在厂主生产杯子时先不让生产线知道我要产的是玻璃的还是瓷的，而是让它在不知道具体材料的情况下先做它能做的，等到它把模具做好，只需要向其中填充玻璃原料或者瓷原料就可以造出同一形状的具体杯子了。但是很可惜，C#并不能new一个抽象的Cup，所以就有了简单工厂模式。原来是Cup cup=new Cup;现在是SimpleCupFactory.createCup(String cupName),根据cup的名字生产Cup,而createCup返回的是一个实现了 Cup接口或抽象类的具体Cup。简单抽象工厂模式有一个问题，就是当我现在想生产一个同样形状的铁杯时，工厂里并没有定义相应的处理流程，只能更改createCup方法，这就不合理了。我现在只是想生产铁杯，你只要在最后的时候把玻璃原料换成铁的不就行了吗，干嘛还要更改整条生产线呢？于是就有了工厂模式。原来生产线在生产模具的时候还要考虑是为玻璃杯生产的模具还是为铁杯生产的模具，现在它不用管了。CupFactory.createCup()创建Cup.CupFactory是接口或抽象类。实现它的具体子类会创建符合Cup接口的具体Cup。那么现在厂主想要生产水壶(kettle)，用工厂模式就不得不再造一条水壶生产线，能不能在水杯生产线同时生产水壶呢？这就是抽象工厂模式。