面试宝典-设计模式

设计模式的六大原则

单一职责原理

应该有且仅有一个原因引起类的变更。

单一职责适用于接口、类，同时也适用于方法，什么意思呢？一个方法尽可能做一件事情。

单一职责原则有什么好处：

- 类的复杂性降低，实现什么职责都有清晰明确的定义；
- 可读性提高，复杂性降低，那当然可读性提高了；
- 可维护性提高，可读性提高，那当然更容易维护了；
- 变更引起的风险降低，变更是必不可少的，如果接口的单一原则做的好，一个接口修改只对相应的实现类有影响，对其他的接口无影响，这对系统的扩展性、维护性都有非常大的帮助。

单一职责原则最难划分的就是职责。一个职责一个接口，但问题是“职责”没有一个量化的标准，一个类到底要负责那些职责？这些职责该怎么细化？细化后是否都要有一个接口或类？这些都需要从实际的项目去考虑。

单一职责适用于接口、类，同时也适用于方法，什么意思呢？一个方法尽可能做一件事情。

对于单一职责原理，我的建议是接口一定要做到单一职责，类的设计尽量做到只有一个原因引起变化。

里氏替换原则

为了让单一继承原则的优势发挥最大的作用，减少弊端，解决方案就是引入里氏置换原则（Lishov Substitution Principle,LSP）。

什么是里氏置换原则？它有两种定义：

- 第一种定义，也是最正宗的定义：If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T,the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.（如果对每一个类型为 S 的对象 o1，都有类型为 T 的对象 o2，使得以 T 定义的所有程序 P 在所有的对象 o1 都代换成 o2 时，程序 P 的行为没有发生变化，那么类型 S 是类型 T 的子类型）。
- 第二种定义：Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.（所有引用基类的方法必须能透明地使用其子类的对象）。

第二个定义是最清晰明确的，通俗点讲，只要父类能出现的地方子类就可以出现，而且替换为子类也不会产生任何错误或异常，使用者可能根本就不需要知道是父类还是子类。但是，反过来就不行了，有子类出现的地方，父类未必就能适应。

依赖倒置原则

翻译过来，包含三层含义：

- 高层模块不应该依赖低层模块，两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。

依赖倒置原则在 Java 语言中的表现就是：

- 模块间的依赖通过抽象发生，实现类之间不发生直接的依赖关系，其依赖关系是通过接口或抽象类产生的；
- 接口或抽象类不依赖于实现类；
- 实现类依赖接口或抽象类。

接口隔离原则

建立单一接口，不要建立臃肿庞大的接口。通俗讲就是：接口尽量细化，同时接口中的方法尽量少。

接口隔离原则与单一职责的不同：接口隔离原则与单一职责的审视角度不相同，单一职责要求的是类和接口职责单一，注重的是职责，这是业务逻辑上的划分，而接口隔离原则要求接口的方法尽量少。

迪米特法则

迪米特法则（Law of Demeter, Lod，也称为最少知识原则（Least Knowledge Principle，LKP）：一个对象应该对其他对象有最少的了解。通俗地讲，一个类应该对自己需要耦合或调用地类知道得最少。

开闭原则

一个软件实体如类、模块和函数应该对扩展开放，对修改关闭。

简单工厂模式

模式动机

考虑一个简单的软件应用场景，一个软件系统可以提供多个外观不同的按钮（如圆形按钮、矩形按钮、菱形按钮等），这些按钮都源自同一个基类，不过在继承基类后不同的子类修改了部分属性从而使得它们可以呈现不同的外观，如果希望在使用这些按钮时，不需要知道这些具体按钮类的名字，只需要知道表示该按钮类的一个参数，并提供一个调用方便的方法，把该参数传入方法即可返回一个相应的按钮对象，此时，就可以使用简单工厂模式。

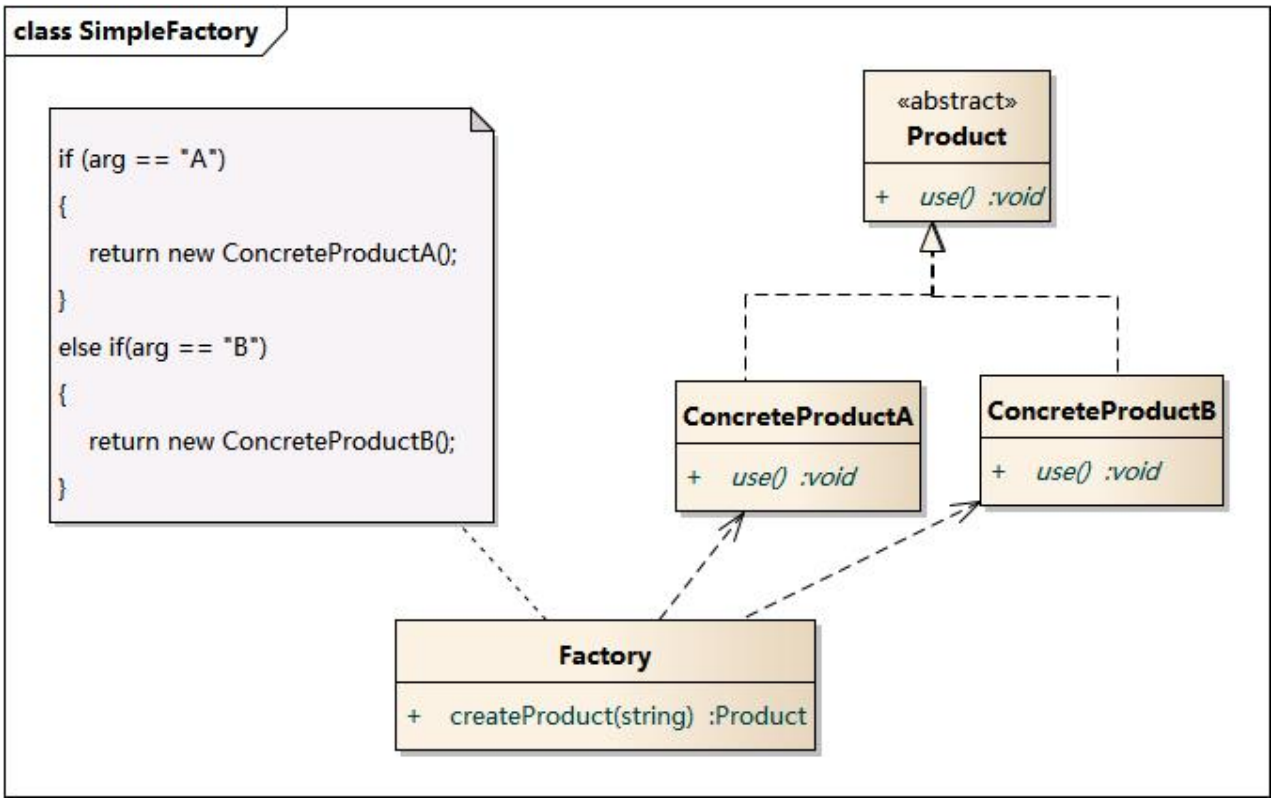
模式定义

简单工厂模式（Simple Factory Pattern）：又称为静态工厂方法（Static Factory Method）模式，它属于类创建型模式。在简单工厂模式中，可以根据参数的不同返回不同类的实例。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。

模式结构

简单工厂模式包含如下角色：

- Factory：工厂角色
  工厂角色负责实现创建所有实例的内部逻辑。
- Product：抽象产品角色
  抽象产品角色是所创建的所有对象的父类，负责描述所有实例所共有的公共接口。
- ConcreteProduct：具体产品角色
  具体产品角色是创建目标，所有创建的对象都充当这个角色的某个具体类的实例。



模式分析

- 将对象的创建和对象本身业务处理分离，可以降低系统的耦合度，使得两者修改起来都相对容易。
- 在调用工厂类的工厂方法时，由于工厂方法是静态方法，使用起来很方便，可通过类名直接调用，而且只需要传入一个简单的参数即可，在实际开发中，还可以在调用时将所传入的参数保存在 XML 等格式的配置文件中，修改参数时无须修改任何源代码。
- 简单工厂模式最大的问题在于工厂类的职责相对过重，增加新的产品需要修改工厂类的判断逻辑，这一点与开闭原则是相违背的。
- 简单工厂模式的要点在于：当你需要什么，只需要传入一个正确的参数，就可以获取你所需要的对象，而无须知道其创建细节。

简单工厂模式的优点

- 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，仅仅“消费”产品即可，简单工厂模式通过这种做法实现了对责任的分割，它提供了专门的工厂类用于创建对象。
- 客户端无须知道所创建的具体产品类的类名，只需要知道具体产品类所对应的参数即可，对于一些复杂的类名，通过简单工厂模式可以减少使用者的记忆量。

- 通过引入配置文件，可以在不修改任何客户端代码的情况下更换和增加新的具体产品类，在一定程度上提高了系统的灵活性。

简单工厂模式的缺点

- 由于工厂类集中了所有产品创建逻辑，一旦不能正常工作，这个系统都要受到影响。
- 使用简单工厂模式将会增加系统中类的个数，在一定程度上增加了系统的复杂度和理解难度。
- 系统扩展困难，一旦添加新产品就不得不修改工厂逻辑，在产品类型较多时，有可能造成工厂逻辑过于复杂，不利于系统的扩展和维护。
- 简单工厂模式由于使用了静态工厂方法，造成工厂角色无法形成基于继承的等级结构。

适用环境

在以下情况下可以使用简单工厂模式：

- 工厂类负责创建的对象比较少：由于创建的对象较少，不会造成工厂方法中的业务逻辑太过复杂。
- 客户端只知道传入工厂类的参数，对于如何创建对象不关心：客户端既不需要关心创建细节，甚至连类名都不需要记住，只需要知道类型所对应的参数。

工厂方法模式

模式动机

对系统进行修改，不再设计一个按钮工厂类来统一负责所有产品的创建，而是将具体按钮的创建过程交给专门的工厂子类去完成，先定义一个抽象的按钮工厂类，再定义具体的工厂类来生成圆形按钮、矩形按钮、菱形按钮等，它们实现在抽象按钮工厂类中定义的方法。这种抽象化的结果使这种结构可以在不修改具体的工厂类的情况下引进新的产品，如果出现新的按钮类型，只需要为这种新类型的按钮创建一个具体的工厂类就可以获得该新按钮的实例，这一特点无疑使得工厂方法模式具有超越简单工厂模式的优越性，更加符合“开闭原则”。

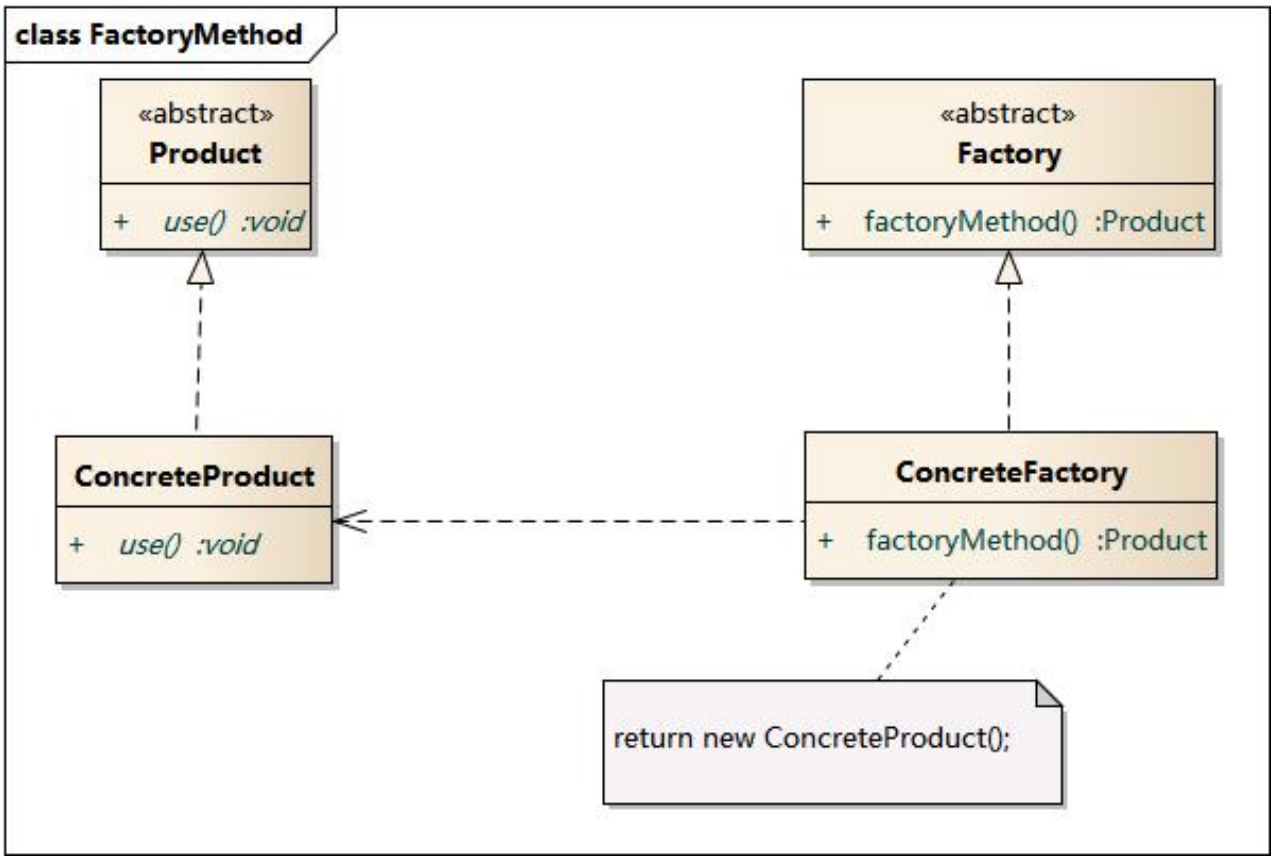
模式定义

工厂方法模式（Factory Method Pattern）又称为工厂模式，也叫虚拟构造器（Virtual Constructor）模式或者多态工厂（Polumorphic Factory）模式，它属于类创建型模式。在工厂方法模式中，工厂父类负责定义创建产品对象的公共接口，而工厂子类则负责生成具体的产品对象，这样做的目的是将产品类的实例化操作延迟到工厂子类中完成，即通过工厂子类来确定究竟应该实例化哪一个具体产品类。

模式结构

工厂方法模式包含如下角色：

- Product：抽象产品
- ConcreateProduct：具体产品
- Factory：抽象工厂
- ConcreateFactory：具体工厂



模式分析

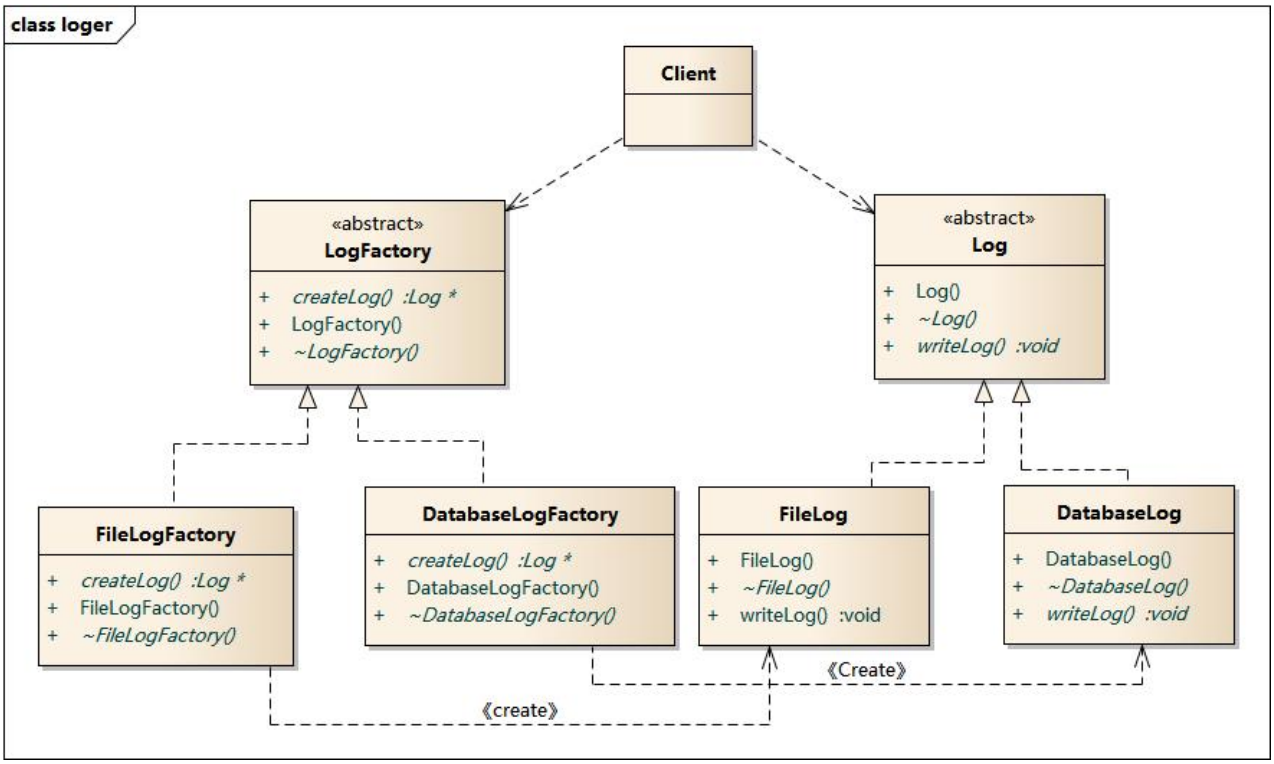
工厂方法模式是简单工厂模式的进一步抽象和推广。由于使用了面向对象的多态性，工厂方法模式保持了简单工厂模式的优点，而且克服了它的缺点。在工厂方法模式中，核心的工厂类不再负责所有产品的创建，而是将具体创建工作交给子类去做。这个核心类仅仅负责给出具体工厂必须实现的接口，而不负责哪一个产品类被实例化这种细节，这使得工厂方法模式可以允许系统在不修改工厂角色的情况下引进新产品。

实例

日志记录器

某系统日志记录器要求支持多种日志记录方式，如文件记录、数据库记录等，且用户可以根据要求动态选择日志记录方式，现使用工厂方法模式设计该系统。

结构图：



工厂方法模式的优点

- 在工厂方法模式中，工厂方法用来创建客户所需要的产品，同时还向客户隐藏了哪种具体产品类将被实例化这一细节，用户只需要关心所需产品对应的工厂，无需关心创建细节，甚至无须知道具体产品类的类名。
- 基于工厂角色和产品角色的多态性设计是工厂方法模式的关键。它能够使工厂可以自主确定创建何种产品对象，而如何创建这个对象的细节则完全封装在具体工厂内部。工厂方法模式之所以又被称为多态工厂模式，是因为所有的具体工厂类都具有同一抽象父类。
- 使用工厂方法模式的另一个优点是在系统中加入新产品时，无需修改抽象工厂和抽象产品提供的接口，无需修改客户端，也无需修改其他的具体工厂和具体产品，而只要添加一个具体工厂和具体产品就可以了。这样，系统的可扩展性也就变得非常好，完全符合“开闭原则”。

工厂模式的缺点

- 在添加新产品时，需要编写新的具体产品类，而且还要提供与之对应的具体工厂类，系统中类的个数将成对增加，在一定程序上增加了系统的复杂度，有更多的类需要编译和运行，会给系统带来一些额外的开销。
- 由于考虑到系统的可扩展性，需要引入抽象层，在客户端代码中均使用抽象层进行定义，增加了系统的抽象性和理解难度，且在实现时可能需要用到 DOM、反射等技术，增加了系统的实现难度。

适用环境

在以下情况下可以使用工厂方法模式：

- 一个类不知道它所需要的对象的类：在工厂方法模式中，客户端不需要知道具体产品类的类名，只需要知道所对应的工厂即可，具体的产品对象由具体的工厂类创建；客户端需要知道创建具体产品的工厂类。
- 一个类通过其子类来指定创建哪个对象：在工厂方法模式中，对于抽象工厂类只需要提供一个创建产品的接口，而由其子类来确定具体要创建的对象，利用面向对象的多态性和里氏代换原则，在程序运行时，子类对象将覆盖父类对象，从而使得系统更容易扩展。

- 将创建对象的任务委托给多个工厂子类中的某一个，客户端在使用时可以无须关心是哪一个工厂子类创建产品子类，需要时再动态指定，可将具体工厂类的类名存储在配置文件或数据库中。

模式扩展

- 使用多个工厂方法：在抽象工厂角色中可以定义多个工厂方法，从而使具体工厂角色实现这些不同的工厂方法，这些方法可以包含不同的业务逻辑，以满足对不同的产品对象的需求。
- 产品对象的重复使用：工厂对象将已经创建过的产品保存到一个集合（如数组、List等）中，然后根据客户对产品的请求，对集合进行查询。如果有满足要求的产品对象，就直接将该产品返回客户端；如果集合中没有这样的产品对象，那么就创建一个新的满足要求的产品对象，然后将这个对象在增加到集合中，再返回给客户端。
- 多态性的丧失和模式的退化：如果工厂仅仅返回一个具体产品对象，便违背了工厂方法的用意，发生退化，此时就不再是工厂方法模式了。一般来说，工厂对象应当有一个抽象的父类型，如果工厂等级结构中只有一个具体工厂类的话，抽象工厂应当有一个抽象的父类型，如果工厂等级结构中只有一个具体工厂类的话，抽象工厂就可以省略，也将发生了退化。当只有一个具体工厂，在具体工厂中可以创建所有的产品对象，并且工厂方法设计为静态方法时，工厂方法模式就退化为简单工厂模式。

抽象工厂模式

模式动机

- 在工厂方法模式中具体工厂负责生产具体的产品，每一个具体工厂对应一种具体产品，工厂方法也具有唯一性，一般情况下，一个具体工厂中只有一个工厂方法或者一组重载的工厂方法。但是有时候需要一个工厂可以提供多个产品对象，而不是单一的产品对象。
  为了更清晰地理解工厂方法模式，需要先引入两个概念：
  - **产品等级结构**：产品等级结构即产品的继承结构，如一个抽象类是电视机，其子类有海尔电视机、海信电视机、TCL 电视机，则抽象电视机与具体品牌的电视机之间构成了一个产品等级结构，抽象电视机是父类，而具体品牌的电视机是其子类。
  - **产品族**：在抽象工厂模式中，产品族是指有同一个工厂生产的，位于不同产品等级结构中的一组产品，如海尔电器工厂生产的海尔电视机、海尔电冰箱，海尔电视机位于电视机产品等级结构中，海尔电冰箱位于电冰箱产品等级结构中。
- 当系统所提供的的工厂所需生产的具体产品并不是一个简单的对象，而是多个位于不同产品等级结构中属于不同类型的具体产品时需要使用抽象工厂模式。
- 抽象工厂模式是所有形式的工厂模式中最为抽象和最具一般性的一种形态。
- 抽象工厂模式与工厂方法模式最大的区别在于：工厂方法模式针对的是一个产品等级结构，而抽象工厂模式则需要面对多个产品等级结构，一个工厂等级结构可以负责多个不同产品等级结构中的产品对象的创建。当一个工厂等级结构可以创建出分属于不同产品等级结构的一个产品族中的所有对象时，抽象工厂模式比工厂方法模式更为简单、有效率。

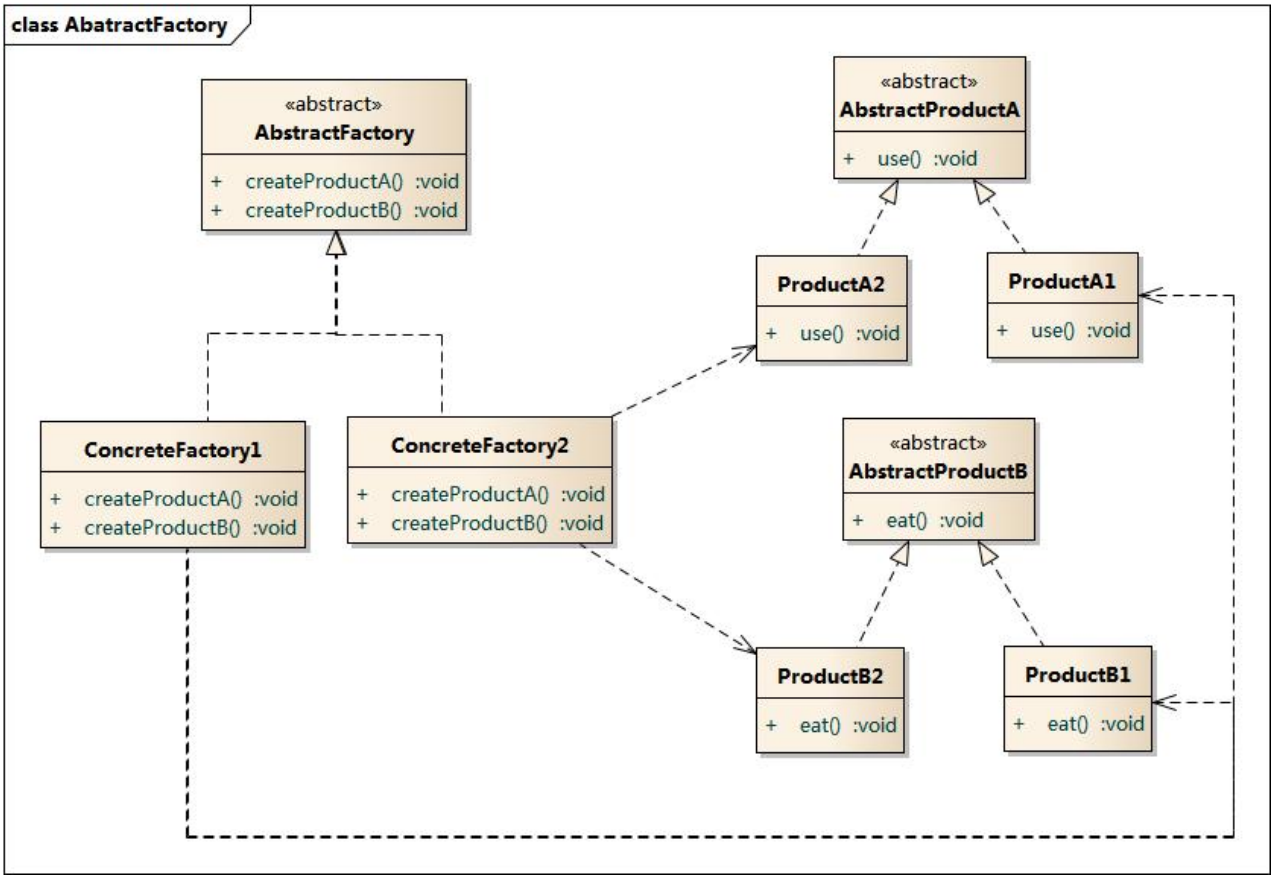
模式定义

抽象工厂模式（Abstract Factory Pattern）：提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们具体的类。抽象工厂模式又称为 Kit 模式，属于对象创建型模式。

模式结构

抽象工厂模式包含如下角色：

- AbstractFactory：抽象工厂
- ConcreateFactory：具体工厂
- AbstractProduct：抽象产品
- Product：具体产品



优点

- 抽象工厂模式隔离了具体类的生成，使得客户并不需要知道什么被创建。由于这种隔离，更换一个具体工厂就变得相对容易。所有的具体工厂都实现了抽象工厂中定义的那些公共接口，因此只需改变具体工厂的实例，就可以在某种程度上改变整个软件系统的行为。另外，应用抽象工厂模式可以实现高内聚低耦合的设计目的，因此抽象工厂模式得到了广泛的应用。
- 当一个产品族中的多个对象被设计成一起工作时，它能够保证客户端始终只使用同一个产品族中的对象。这对一些需要根据当前环境来决定其行为的软件系统来说，是一种非常实用的设计模式。
- 增加新的具体工厂和产品族很方便，无需修改已有系统，符合“开闭原则”。

缺点

- 在添加新的产品对象时，难以扩展抽象工厂来生产新种类的产品，这是因为在抽象工厂角色中规定了所有可能被创建的产品集合，要支持新种类的产品就意味着要对该接口进行扩展，而这将涉及到对抽象工厂角色及其所有子类的修改，显然会带来较大的不便。
- 开闭原则的倾斜性（增加新的工厂和产品族容易，增加新的产品等级结构麻烦）。

适用模式

在以下情况下可以使用抽象工厂模式：

- 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，这对于所有类型的工厂模式都是重要的。
- 系统中有多于一个的产品族，而每次只是用其中某一个产品族。
- 属于同一个产品族的产品将在一起使用，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具体实现。

模式应用

在很多软件系统中需要更换界面主题，要求界面中的按钮、文本框、背景色等一起发生改变时，可以使用抽象工厂模式进行设计。

模式扩展

“ 开闭原则 ” 的倾斜性

- “ 开闭原则 ” 要求系统对扩展开放，对修改封闭，通过扩展达到增强其功能的目的。对于涉及到多个产品族与多个产品等级结构的系统，其功能增强包括两方面：
    - 1.增加产品族：对于增加新的产品族，工作方法模式很好的支持了 “ 开闭原则 ”，对于新增加的产品族，只需要对应增加一个新的具体工厂即可，对已有代码无需做任何修改。
    - 2.增加新的产品等级结构：对于增加新的产品等级结构，需要修改所有的工厂角色，包括抽象工厂类，在所有的工厂类中都需要增加生产新产品的方法，不能很好的支持“开闭原则”。
- 抽象工厂模式的这种性质称为 “ 开闭原则 ” 的倾斜性，抽象工厂模式以一种倾斜的方式支持增加新的产品，它为新产品族的增加提供方便，但不能为新的产品等级结构的增加提供这样的方便。

工厂模式的退化

- 当抽象工厂模式中每一个具体工厂类只创建一个产品对象，也就是只存在一个产品等级结构时，抽象工厂模式退化成工厂方法模式；当工厂方法模式中抽象工厂与具体工厂合并，提供一个统一的工厂来创建产品对象，并将创建对象的工厂方法设计为静态方法时，工厂方法模式退化成简单工厂模式。

单例模式

模式动机

对于系统中的某些类来说，只有一个实例很重要。例如，一个系统中可以存在多个打印任务，但是只能有一个正在工作的任务；一个系统只能有一个窗口管理器或文件系统；一个系统只能有一个计时工具或 ID(序号)生成器。

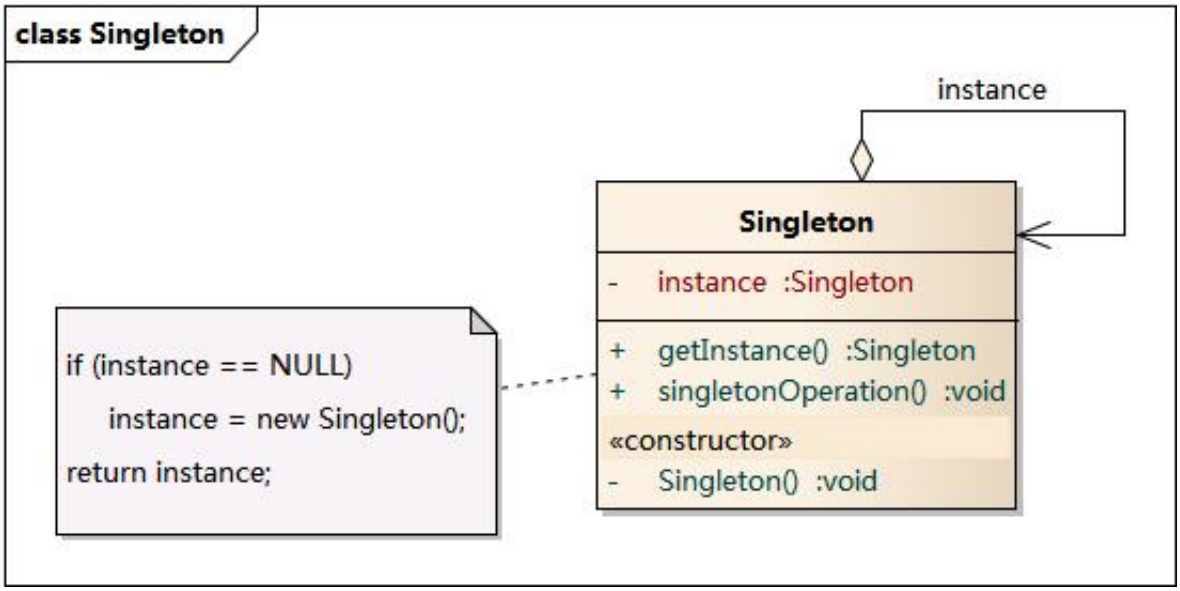
如何保证一个类只有一个实例，并且这个实例易于被访问呢？定义一个全局变量可以确保对象随时可以被访问，但不能防止实例化多个对象。一个更好的解决方法是让类自身负责保存它的唯一实例，这个类可以保证没有其他实例被创建，并且它可以提供一个访问该实例的方法，这就是单例模式的模式动机。

模式定义

单例模式（Singleton Pattern）：单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例，这个类称为单例类，它提供全局访问的方法。

单例模式的要点有三个：一是某个类只能有一个实例；二是它必须自行创建这个实例；三是它必须自行向整个系统提供这个实例。单例模式是一种对象创建型模式。单例模式又名单件模式或单态模式。

模式结构



模式分析

单例模式的目的是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。单例模式包含的角色只有一个，就是单例类 -- Singleton。单例类拥有一个私有构造函数，确保用户无法通过 new 关键字直接实例化它。除此之外，该模式中包含一个静态私有成员变量与静态公有的工厂方法，该工厂方法负责检验实例的存在性并实例化自己，然后存储在静态成员变量中，以确保只有一个实例被创建。

在单例模式的实现过程中，需要注意如下三点：

- 单例类的构造函数为私有；
- 提供一个自身的静态私有成员变量；
- 提供一个公有的静态工厂方法。

优点

- 提供了唯一实例的受控访问。因为单例类封装了它的唯一实例，所以它可以严格控制客户怎样以及何时访问它，并为设计及开发团队提供了共享的概念。
- 由于在系统内存中只存在一个对象，因此可以节约系统资源，对于一些需要频繁创建和销毁的对象，单例模式无疑可以提供系统的性能。
- 允许可变数目的实例。可以基于单例模式进行扩展，使用与单例控制相似的方法来获得指定个数的对象实例。

缺点

- 由于单例模式中没有抽象层，因此单例类的扩展有很大的困难。
- 单例类的职责过重，在一定程度上违背了“ 单一职责原则 ”。因为单例类即充当了工厂角色，提供了工厂方法，同时又充当了产品角色，包含一些业务方法，将产品的创建和产品本身的功能融合在一起。
- 滥用单例将带来一些负面问题，如为了节省资源将数据库连接池对象设计为单例类，可能会导致共享连接池对象的程序过多而出现连接池溢出；现在很多面向对象语言（如 Java、c#）的运行环境都提供了自动垃圾回收的技术，因此，如果实例化的对象长时间不被利用，系统会认为它是垃圾，会自动销毁并回收资源，下次利用时又将重新实例化，这将导致对象状态的丢失。

适用环境

在以下情况下可以使用单例模式：

- 系统只需要一个实例对象，如系统要求提供一个唯一的序列号生成器，或者需要考虑资源消耗太大而只允许创建一个对象。
- 客户调用类的单个实例只允许使用一个公共访问点，除了该公共访问点，不能通过其他途径访问该实例。
- 在一个系统中要求一个类只有一个实例时才应当使用单例模式。反过来，如果一个类可以有几个实例共存，就需要对单例模式进行改进，使之成为多例模式。

模式应用

一个具有自动编号主键的表可以有多个用户同时使用，但数据库中只能有一个地方分配下一个主键编号，否则会出现主键重复，因此该主键编号生成器必须具备唯一性，可以通过单例模式来实现。

适配器模式

模式动机

在软件开发中采用类似于电源适配器的设计和编码技巧被称为适配器模式。

通常情况下，客户端可以通过目标类的接口访问它所提供的服务。有时，现有的类可以满足客户类的功能需要，但是它所提供的接口不一定是客户类所期望的，这可能是因为现有类中方法名与目标类中定义的方法名不一致等原因所导致的。

在这种情况下，现有的接口需要转化为客户类期望的接口，这样保证了对现有类的重用。如果不进行这样的转化，客户类就不能利用现有类所提供的功能，适配器模式可以完成这样的转化。

在适配器模式中可以定义一个包装类，包装不兼容接口的对象，这个包装类指的就是适配器（Adapter），它所包装的对象就是是适配者（Adaptee），即被适配的类。

适配器提供客户类需要的接口，适配器的实现就是把客户类的请求转化为对适配者的相应接口的调用。也就是说：当客户类调用适配器的方法时，在适配器类的内部将调用适配者类的方法，而这个过程对客户类是透明的，客户类并不直接访问适配者类。因此，适配器可以使由于接口不兼容而不能交互的类可以一起工作。这就是适配器模式的模式动机。

模式定义

适配器模式（Adapter Pattern）：将一个接口转换成客户希望的另一个接口，适配器模式使接口不兼容的那些类可以一起工作中，其别名为包装器（Wrapper）。适配器模式既可以作为类结构性模式，也可以作为对象结构型模式。

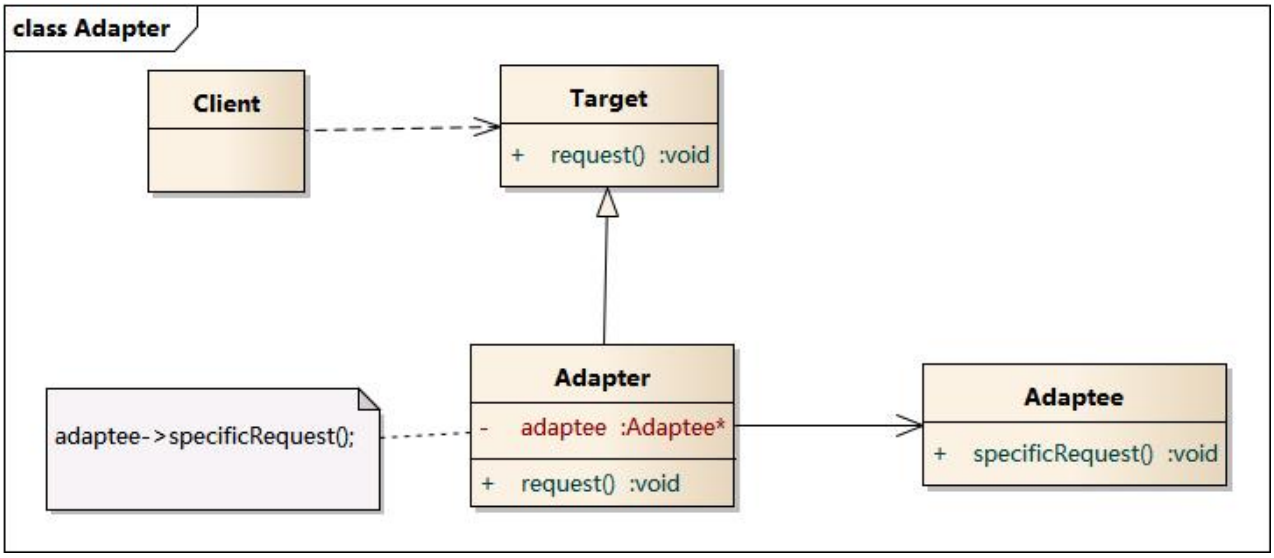
模式结构

适配器模式包含如下角色：

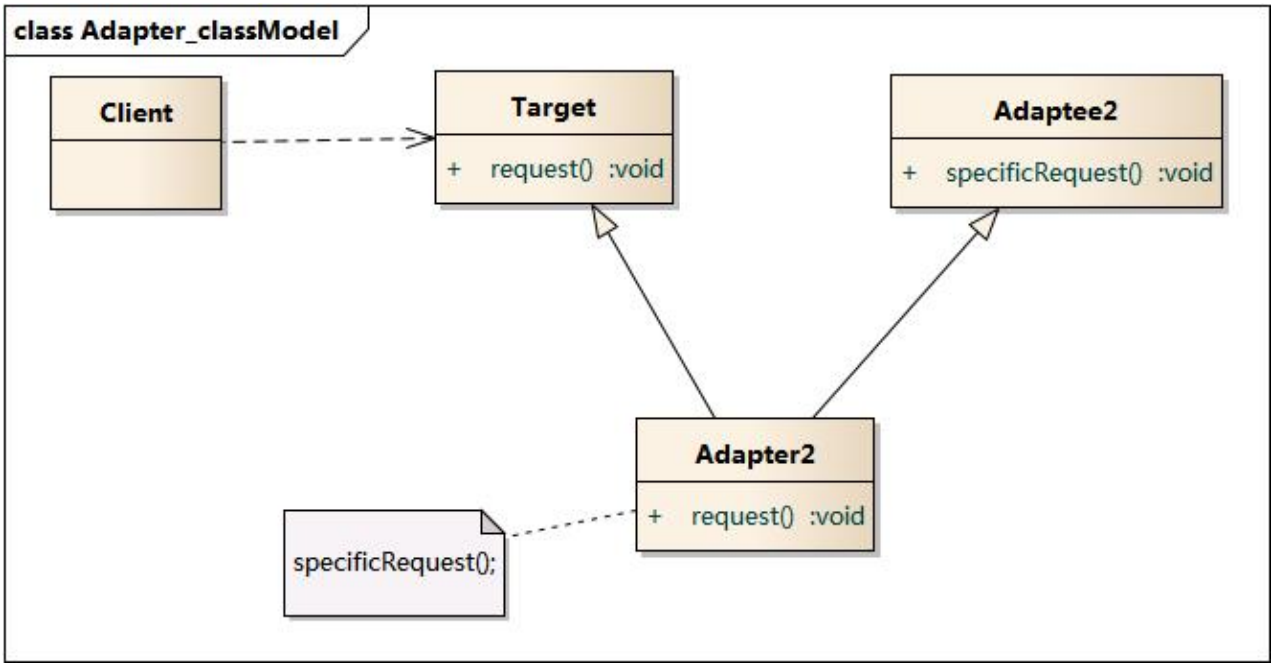
- Target：目标抽象类
- Adapter：适配器类
- Adaptee：适配者类
- Client：客户类

适配器模式有对象适配器和类适配器两种实现：

对象适配器：



类适配器：



优点

- 将目标类和适配者类解耦，通过引入一个适配器类来重现现有的适配者类，而无须修改原有代码。
- 增加了类的透明性和复用性，将具体的实现封装在适配者类中，对于客户款来说是透明的，而且提高了适配者的复用性。
- 灵活性和扩展性都非常好，通过使用配置文件，可以很方便地更换适配器，也可以在不修改原有代码的基础上增加新的适配器类，完全符合“开闭原则”。

类适配器模式还具有如下优点：由于适配器类是适配者类的子类，因此可以在适配器类中置换一些适配者的方法，使得适配器的灵活性更强。

对象适配器模式还具有如下优点：一个对象适配器可以把多个不同的适配者适配到同一个目标，也就是说，同一个适配器可以把适配者类和它的子类都适配到目标接口。

缺点

类适配器模式的缺点：对于 Java 、C# 等不支持多重继承的语言，一次最多只能适配一个适配者类，而且目标抽象类只能为抽象类，不能为具体类，其使用有一定的局限性，不能将一个适配者类和它的子类都适配到目标接口。

对象适配器模式的缺点如下：与类适配器模式相比，要想置换适配者类的方法就不容易。如果一定要置换掉适配者类的一个或多个方法，就只好先做一个适配者类的子类，将适配者类的方法置换掉，然后再把适配者类的子类当做真正的适配者进行适配，实现过程较为复杂。

适用环境

在以下情况下可以使用适配器模式：

- 系统需要使用现有的类，而这些类的接口不符合系统的需要。
- 想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作。

模式扩展

默认适配器模式（Default Adapter Pattern） 或**缺省适配器模式**
当不需要全部实现接口提供的方法时，可先设计一个抽象类实现接口，并为该接口中每个方法提供一个默认实现（空方法），那么该抽象类的子类可有选择地覆盖父类的某些方法来实现需求，它适用于一个接口不想使用其所有的方法的情况。因此也称为 **单接口适配器模式** 。

代理模式

模式动机

在某些情况下，一个客户不想或者不能直接引用一个对象，此时可以通过一个称之为“代理”的第三者来实现间接引用。代理对象可以在客户端和目标对象之间起到中介的作用，并且可以通过代理对象去掉客户不能看到的内容和服务或者添加客户需要的额外服务。

通过引入一个新的对象（如小图片和远程代理对象）来实现对真实对象的操作或者将新的对象作为真实对象的一个替身，这种实现机制即为代理模式，通过引入代理对象来间接访问一个对象，这就是代理模式的模式动机。

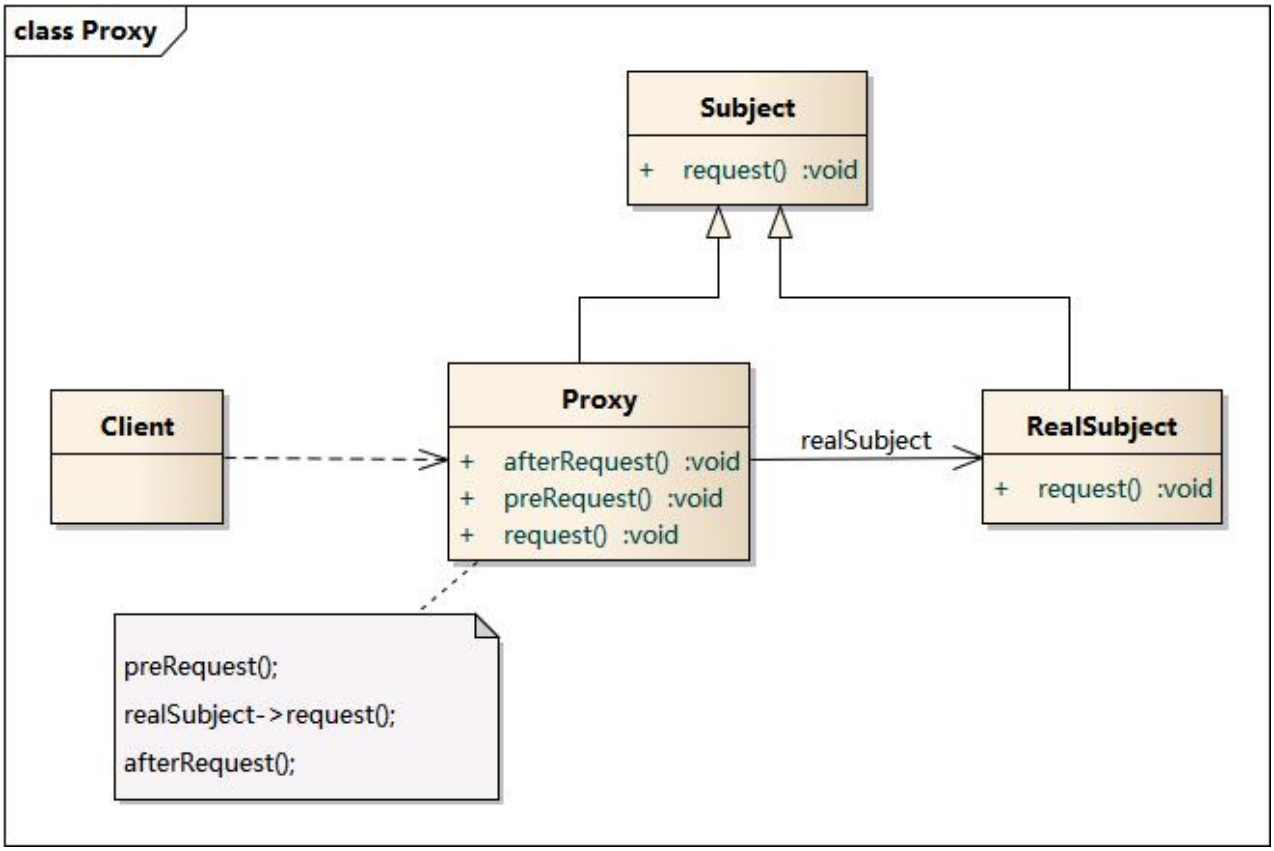
模式定义

代理模式（Proxy Pattern）：给某一个对象提供一个代理，并由代理对象控制对原对象的引用。代理模式的英文叫做 Proxy 或 Surrogate，它是一种对象结构型模式。

模式结构

代理模式包含如下角色：

- Subject：抽象主题角色
- Proxy：代理主题角色
- RealSubject：真实主题角色



优点

代理模式的优点：

- 代理模式能够协调调用者和被调用者，在一定程度上降低了系统的耦合度。
- 远程代理使得客户端可以访问在远程机器上的对象，远程机器可能具有更好的计算性能与处理速度，可以快速响应并处理客户端请求。
- 虚拟代理通过使用一个小对象来代表一个大对象，可以减少系统资源的消耗，对系统进行优化并提高运行速度。
- 保护代理可以控制对真实对象的使用权限。

缺点

代理模式的缺点：

- 由于在客户端和真实主题之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢。
- 实现代理模式需要额外的工作，有些代理模式的实现非常复杂。

适用环境

根据代理模式的使用目的，常见的代理模式有以下几种类型：

- 远程（Remote）代理：为一个位于不同的地址空间的对象提供一个本地的代理对象，这个不同的地址空间可以是在同一台主机中，也可是在另一台主机中，远程代理又叫做大使（Ambassador）。
- 虚拟（Virtual）代理：如果需要创建一个资源消耗较大的对象，先创建一个消耗相对较小的对象来表示，真实对象只在需要时才会被真正创建。
- Copy-on-Write 代理：它是虚拟代理的一种，把复制（克隆）操作延迟到只有在客户端真正需要时才执行。一般来说，对象的深克隆是一个开销较大的操作，Copy-on-Write 代理可以让这个操作延迟，只有对象被用到的时候才被克隆。
- 保护（Protect or Access）代理：控制对一个对象的访问，可以给不同的用户提供不同级别的使用权限。
- 缓冲（Cache）代理：为某一个目标操作的结果提供临时的存储空间，以便多个客户端可以共享这些结果。
- 防火墙（Firewall）代理：保护目标不让恶意用户接近。
- 同步化（Synchronization）代理：使几个用户能够同时使用一个对象而没有冲突。
- 智能引用（Smart Reference）代理：当一个对象被引用时，提供一些额外的操作。如将此对象被调用的次数记录下来等。

模式扩展

几种常见的代理模式：

- 图片代理：一个很常见的代理模式的应用实例就是对大图浏览的控制。

  用户通过浏览器访问网页时先不加载真实的大图，而是通过代理对象的方法来进行处理，在代理对象的方法中，先使用一个线程向客户端浏览器加载一个小图片，然后在后台使用另一个线程来调用大图片的加载方法将大图片加载到客户端。当需要浏览大图片时，再将大图片在新网页中显示。如果用户在浏览大图时加载工作还没有完成，可以再启动一个线程来显示相应的提示信息。通过代理技术结合多线程编程将真实图片的加载放到后台来操作，不影响前台图片的浏览。

- 远程代理：远程代理可以将网络的细节隐藏起来，使得客户端不必考虑网络的存在。客户完全可以认为被代理的远程业务对象是局域的而不是远程的，而远程代理对象承担了大部分的网络通信工作。
- 虚拟代理：当一个对象的加载非常耗费资源的时候，虚拟代理的优势就非常明显地体现出来了。虚拟代理模式是一种内存节省技术，那些占用大量内存或处理复杂的对象将推迟到使用它的时候才创建。

在应用程序启动的时候，可以用代理对象代理真正对象初始化，节省了内存的占用，并大大加速了系统的启动时间。

动态代理：

- 动态代理是一种较为高级的代理模式，它的典型应用就是 Spring AOP.
- 在传统的代理模式中，客户端通过 Proxy 调用 RealSubject 类的 request() 方法，同时还在代理类中封装了其他方法（如 preRequest() 和 postRequest()），可以处理一些其他问题。
- 如果按照这种方法使用代理模式，那么真实主题角色必须是事先已经存在的，并将其作为代理对象的内部成员属性。如果一个真实主题角色必须对应一个代理主题角色，这将导致系统中的类个数急剧增加，因此需要想办法减少系统中类的个数，此外，如何在实现不知道真实主题角色的情况下使用代理主题角色，这都是动态代理需要解决的问题。

责任链模式

介绍

责任链模式（Chain of Responsibility Pattern）为请求创建了一个接收者对象的链。责任链模式给与请求的类型，对请求的发送者和接收者进行解耦。责任链模式属于行为型模式。

在责任链模式中，通常每个接收者都包含对另一个接收者的引用。如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者，依次类推。

意图

避免请求发送者与接收者耦合在一起，并且让多个对象都有可能接收请求，沿着这条责任链传递请求，直到有对象处理它为止。

主要解决

责任链上的处理者负责处理请求，客户只需要将请求发送到责任链上即可，无需关心请求的处理细节和请求的传递，所以责任链将请求的发送者和请求的处理者解耦了。

使用

如何使用：在处理消息的时候经过多个拦截类处理。

如何解决：拦截的类都实现统一接口。

关键代码：在处理端自由的添加或删除拦截类，拦截类提供统一的接口接收传递的消息，拦截者自行判断是否可以处理消息，并将消息传递给下一个拦截者。

优点

1. 降低耦合度。它将请求的发送者和接收者解耦。
2. 简化了对象。使得对象不需要知道链的结构，每个链（处理类）也只需要处理自己负责的部分。
3. 增强给对象指派职责的灵活性。通过改变链内的成员或者调动它们的次序，允许动态地增加或者删除责任。
4. 增加新的请求处理类很方便。

缺点

1. 不能保证请求一定被接收。可能一个处理者就拦截消息直接返回了，或者出现了异常，导致消息无法发送给下一个处理者。
2. 系统性能将受到一定影响，并且在进行代码调试时不太方便，可能会造成循环调用。
3. 可能不容易观察运行时的特征，有碍于除错。

使用场景

1. 有多个对象可以处理同一个请求，具体哪个对象处理该请求由运行时刻自动确定。
2. 在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。
3. 可动态指定一组对象处理请求。

观察者模式

模式动机

建立一种对象与对象之间的依赖关系，一个对象发生改变时将自动通知其他对象，其他对象将相应作出反应。在此，发生改变的对象称为**观察目标**，而被通知的对象称为**观察者**，一个观察目标可以对应多个观察者，而且这些观察者之间没有相互联系，可以根据需要增加和删除观察者，使得系统更易于扩展，这就是观察者模式的模式动机。

模式定义

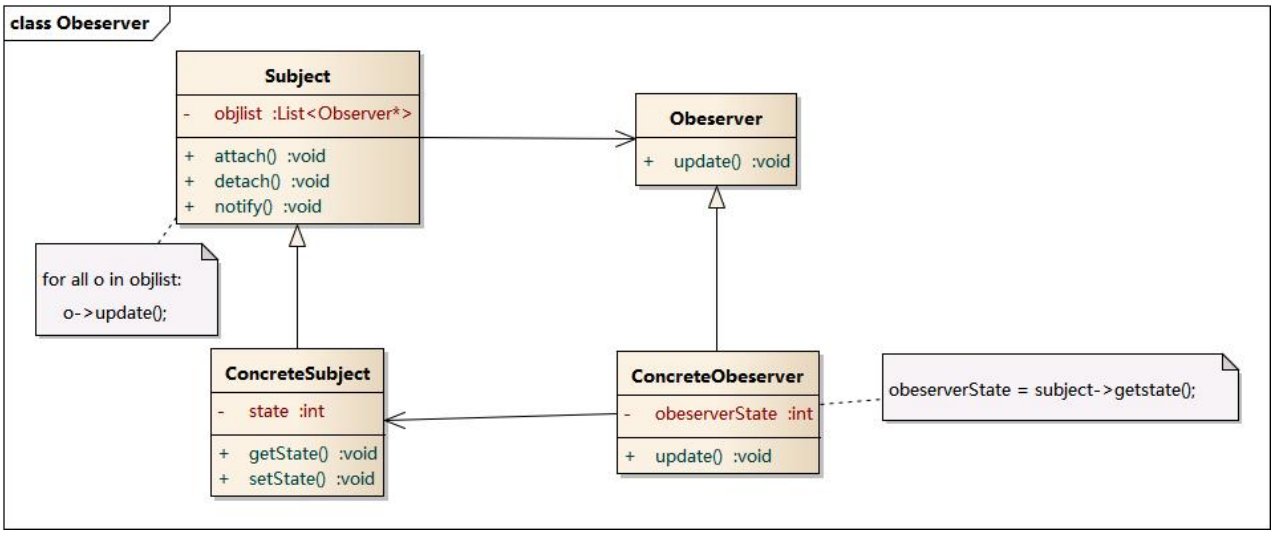
观察者模式（Observer Pattern）：定义对象间的一种一对多依赖关系，使得每当一个对象状态发生改变时，其相关依赖对象皆得到通知并被自动更新。观察者又叫做发布-订阅（Publish/Subscribe）模式、模型-视图（Model/View）模式、源-监听器（Source/Listener）模式或从属者（Dependents）模式。

观察者模式是一种对象行为型模式。

模式结构

观察者模式包含如下角色：

- Subject：目标
- ConcreteSubject：具体目标
- Observer：观察者
- ConcreteObserver：具体观察者



模式分析

- 观察者模式描述了如何建立对象与对象之间的依赖关系，如何构造满足这种需求的系统。
- 这一模式中的关键对象是观察目标和观察者，一个目标可以有任意数目的与之相依赖的观察者，一旦目标的状态发生改变，所有的观察者都将得到通知。
- 作为对这个通知的响应，每个观察者都将即时更新自己的状态，以与目标状态同步，这种交互也称为发布-订阅（publish - subscribe）。目标是通知的发布者发出通知时并不需要知道谁是它的观察者，可以有任意数目的观察者订阅它并接收通知。

优点

观察者模式的优点：

- 观察者模式可以实现表示层和数据逻辑层的分离，并定义了稳定的消息更新传递机制，抽象了更新接口，使得可以有各种各样不同的表示层作为具体观察者角色。
- 观察者模式在观察目标和观察者之间建立一个抽象的耦合。
- 观察者模式支持广播通信。
- 观察者模式符合“开闭原则”的要求。

缺点

观察者模式的缺点：

- 如果一个观察目标对象有很多直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。
- 如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统奔溃。
- 观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化。

适用环境

在以下情况可以使用观察者模式：

- 一个抽象模型有两个方面，其中一个方面依赖于另一个方面，将这些方面封装在独立的对象中，使它们可以各自独立地改变和复用。
- 一个对象的改变将导致其他一个或多个对象也发生改变，而不知道具体有多少对象将发生改变，可

以降低对象之间的耦合度。
- 一个对象必须通知其他对象，而并不知道这些对象是谁。
- 需要在系统中创建一个触发链，A 对象的行为将影响 B 对象，B对象的行为将影响 C 对象...，可以使用观察者模式创建一种链式触发机制。

装饰者模式

模式动机

一般有两种方式可以实现给一个类或对象增加行为：

- 继承机制，使用继承机制是给现有类添加功能的一种有效途径，通过继承一个现有类可以使得子类在拥有自身方法的同时还拥有父类的方法。但是这种方法是静态的，用户不能控制增加行为的方式和时机。
- 关联机制，即将一个类的对象嵌入另一个对象中，由另一个对象来决定是否调用嵌入对象的行为以便扩展自己的行为，我们称这个嵌入的对象为装饰器(Decorator)

装饰模式以对客户透明的方式动态地给一个对象附加上更多的责任，换言之，客户端并不会觉得对象在装饰前和装饰后有什么不同。装饰模式可以在不需要创造更多子类的情况下，将对象的功能加以扩展。这就是装饰模式的模式动机。

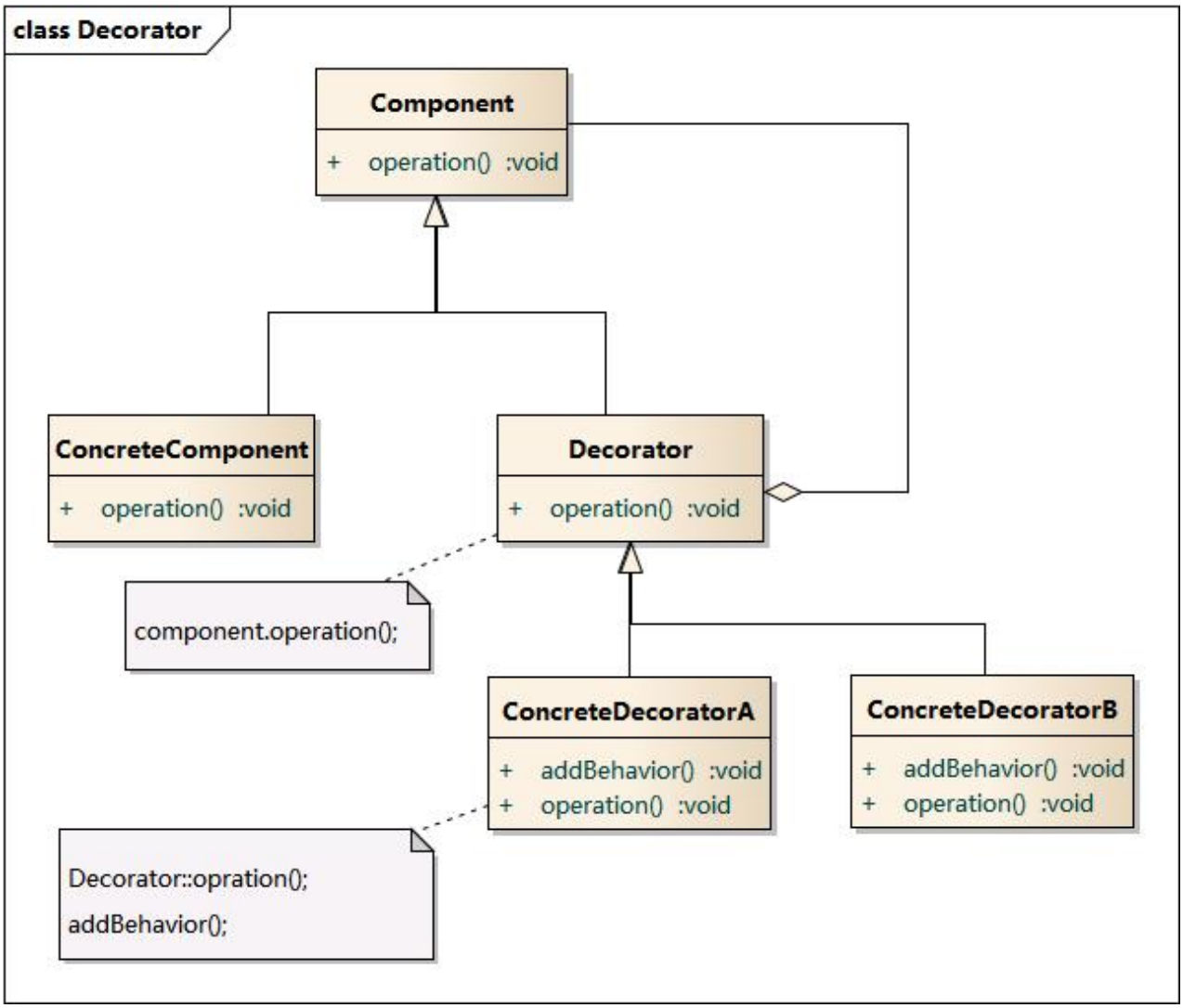
模式定义

装饰模式(Decorator Pattern)： 动态地给一个对象增加一些额外的职责(Responsibility)，就增加对象功能来说，装饰模式比生成子类实现更为灵活。其别名也可以称为包装器(Wrapper)，与适配器模式的别名相同，但它们适用于不同的场合。根据翻译的不同，装饰模式也有人称之为“油漆工模式”，它是一种对象结构型模式。

模式结构

装饰模式包含如下角色：

- Component: 抽象构件
- ConcreteComponent: 具体构件
- Decorator: 抽象装饰类
- ConcreteDecorator: 具体装饰类



模式分析

- 与继承关系相比，关联关系的主要优势在于不会破坏类的封装性，而且继承是一种耦合度较大的静态关系，无法在程序运行时动态扩展。在软件开发阶段，关联关系虽然不会比继承关系减少编码量，但是到了软件维护阶段，由于关联关系使系统具有较好的松耦合性，因此使得系统更加容易维护。当然，关联关系的缺点是比继承关系要创建更多的对象。
- 使用装饰模式来实现扩展比继承更加灵活，它以对客户透明的方式动态地给一个对象附加更多的责任。装饰模式可以在不需要创造更多子类的情况下，将对象的功能加以扩展。

优点

- 装饰模式与继承关系的目的都是要扩展对象的功能，但是装饰模式可以提供比继承更多的灵活性。
- 可以通过一种动态的方式来扩展一个对象的功能，通过配置文件可以在运行时选择不同的装饰器，从而实现不同的行为。
- 通过使用不同的具体装饰类以及这些装饰类的排列组合，可以创造出很多不同行为的组合。可以使用多个具体装饰类来装饰同一对象，得到功能更为强大的对象。
- 具体构件类与具体装饰类可以独立变化，用户可以根据需要增加新的具体构件类和具体装饰类，在使用时再对其进行组合，原有代码无须改变，符合“开闭原则”。

缺点

- 使用装饰模式进行系统设计时将产生很多小对象，这些对象的区别在于它们之间相互连接的方式有所不同，而不是它们的类或者属性值有所不同，同时还将产生很多具体装饰类。这些装饰类和小对象的产生将增加系统的复杂度，加大学习与理解的难度。
- 这种比继承更加灵活机动的特性，也同时意味着装饰模式比继承更加易于出错，排错也很困难，对于多次装饰的对象，调试时寻找错误可能需要逐级排查，较为烦琐。

适用环境

- 在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。
- 需要动态地给一个对象增加功能，这些功能也可以动态地被撤销。
- 当不能采用继承的方式对系统进行扩充或者采用继承不利于系统扩展和维护时。不能采用继承的情况主要有两类：第一类是系统中存在大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长；第二类是因为类定义不能继承（如final类）.

模式扩展

装饰模式的简化-需要注意的问题:

- 一个装饰类的接口必须与被装饰类的接口保持相同，对于客户端来说无论是装饰之前的对象还是装饰之后的对象都可以一致对待。
- 尽量保持具体构件类Component作为一个“轻”类，也就是说不要把太多的逻辑和状态放在具体构件类中，可以通过装饰类对其进行扩展。
- 如果只有一个具体构件类而没有抽象构件类，那么抽象装饰类可以作为具体构件类的直接子类。

关于设计模式的常见编程题

替换多重嵌套的 if-else 写法

使用工厂模式来实现对多重嵌套的 if else 的替换。

根据不同的条件，使用不同的方法。

```
public class StrategyFactory {
    public static final int KEY_GESTURE_PWD = R.id.tv_forget_gestrue_pwd;
    public static final int KEY_OTHER_ACCOUNT_LOGIN =
R.id.tv_other_account_login;

    private static StrategyFactory factory = new StrategyFactory();
    private StrategyFactory(){
    }
    private static Map<Integer,Strategy> strategyMap = new
HashMap<Integer,Strategy>();
    static{
        strategyMap.put(KEY_GESTURE_PWD, new GesturePwdStrategy());
        strategyMap.put(KEY_OTHER_ACCOUNT_LOGIN, new
OtherAccountLoginStrategy());
    }
    public Strategy creator(int pType){
```

```
        return strategyMap.get(pType);
    }
    public static StrategyFactory getInstance(){
        return factory;
    }
}
```

写一个单例

```
public class Single {
    /**
     * 懒汉模式，线程不安全
     * 这种方式是最基本的实现方式，这种实现最大的问题就是不支持多线程。因为没有加锁 synchronized，所以严格意义上它并不算单例模式。
     * 这种方式 lazy loading 很明显，不要求线程安全，在多线程不能正常工作。
     */
    public static class Single1 {
        private static Single1 instance;

        private Single1() { }

        public static Single1 getInstance() {
            if (instance == null) {
                instance = new Single1();
            }
            return instance;
        }
    }

    /**
     * 饿汉模式，线程安全
     * 这种方式比较常用，但容易产生垃圾对象。
     * 优点：没有加锁，执行效率会提高。
     * 缺点：类加载时就初始化，浪费内存。
     * 它基于 classloder 机制避免了多线程的同步问题，不过，instance 在类装载时就实例化，虽然导致类装载的原因有很多种，在单例模式中大多数都是调用 getInstance 方法，
     * 但是也不能确定有其他的方式（或者其他的静态方法）导致类装载，这时候初始化 instance 显然没有达到 lazy loading 的效果。
     */
    public static class Single2 {
        private static Single2 instance = new Single2();

        private Single2() { }

        public static Single2 getInstance() {
            return instance;
        }
    }
```

```
    /**
     * 线程安全,双重锁
     * 这种方式采用双锁机制，安全且在多线程情况下能保持高性能。
     * getInstance() 的性能对应用程序很关键。
     */
    public static class Single3 {
        private volatile static Single3 instance;

        private Single3() { }

        public static Single3 getInstance() {
            if (instance == null) {
                synchronized (Single3.class) {
                    if (instance == null) {
                        instance = new Single3();
                    }
                }
            }
            return instance;
        }
    }

    /**
     * 静态内部类
     * 这种方式能达到双检锁方式一样的功效，但实现更简单。
     * 对静态域使用延迟初始化，应使用这种方式而不是双检锁方式。
     * 这种方式只适用于静态域的情况，双检锁方式可在实例域需要延迟初始化时使用。
     * 这种方式同样利用了 classloder 机制来保证初始化 instance 时只有一个线程，
     * 它跟饿汉模式方式不同的是：饿汉模式方式只要 Singleton 类被装载了，那么 instance 就会被实例化（没有达到 lazy loading 效果），
     * 而这种方式是 SingletonHolder 类被装载了，instance 不一定被初始化。因为 SingletonHolder 类没有被主动使用，只有显示通过调用 getInstance 方法时，
     * 才会显示装载 SingletonHolder 类，从而实例化 instance。想象一下，如果实例化 instance 很消耗资源，所以想让它延迟加载，
     * 另外一方面，又不希望在 Singleton 类加载时就实例化，因为不能确保 Singleton 类还可能在其他的地方被主动使用从而被加载，那么这个时候实例化 instance 显然是不合适的。
     * 这个时候，这种方式相比饿汉模式方式就显得很合理。
     */
    public static class Single5 {

        private static class SingleHolder {
            private static final Single5 instance = new Single5();
        }

        private Single5() { }

        public static final Single5 getInstance() {
            return SingleHolder.instance;
```

```
        }
    }


    /**
     * 懒汉模式，线程安全
     * 这种方式具备很好的 lazy loading，能够在多线程中很好的工作，但是，效率很低，99% 情况下不需要同步。
     * 优点：第一次调用才初始化，避免内存浪费。
     * 缺点：必须加锁 synchronized 才能保证单例，但加锁会影响效率。getInstance() 的性能对应用程序不是很关键（该方法使用不太频繁）。
     */
    public static class Single6{
        private static Single6 instance;
        private Single6(){}
        public static synchronized Single6 getInstance(){
            if (instance == null){
                instance = new Single6();
            }
            return instance;
        }

    }

    /**
     * 枚举
     *
     * 这种实现方式还没有被广泛采用，但这是实现单例模式的最佳方法。它更简洁，自动支持序列化机制，绝对防止多次实例化。
     * 这种方式是 Effective Java 作者 Josh Bloch 提倡的方式，它不仅能避免多线程同步问题，而且还自动支持序列化机制，
     * 防止反序列化重新创建新的对象，绝对防止多次实例化。
     * 不过，由于 JDK1.5 之后才加入 enum 特性，用这种方式写不免让人感觉生疏，在实际工作中，也很少用。
     * 不能通过 reflection attack 来调用私有构造方法。
     */
    public enum Single7{
        INSTANCE;
        private void method(){

        }
    }
    /**
     * 一般情况下，不建议使用懒汉方式，建议使用第 3 种饿汉方式。
     * 只有在要明确实现 lazy loading 效果时，才会使用静态内部类登记方式。
     * 如果涉及到反序列化创建对象时，可以尝试使用枚举方式。如果有其他特殊的需求，可以考虑使用双检锁方式。
     */
}
```