**设计模式**

1. 设计原则

**1).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。开放-封闭原则的意思就是说，你设计的时候，时刻要考虑，尽量让这个类是足够好，写好了就不要去修改了，如果新需求来，我们增加一些类就完事了，原来的代码能不动则不动。这个原则有两个特性，一个是说“对于扩展是开放的”，另一个是说“对于更改是封闭的”。面对需求，对程序的改动是通过增加新代码进行的，而不是更改现有的代码。

**2).里氏代换原则**:所有引用基类的地方必须能透明的使用其子类的对象. 一个软件实体如果使用的是一个父类的话，那么一定适用于其子类，而且它察觉不出父类对象和子类对象的区别，也就是说，在软件里面，把父类都替换成它的子类，程序的行为没有变化.

里氏替换原则通俗的来讲就是：子类可以扩展父类的功能，但不能改变父类原有的功能。它包含以下4层含义：  
①.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。  
②.子类中可以增加自己特有的方法。  
③.当子类的方法重载父类的方法时，方法的前置条件（即方法的形参）要比父类方法的输入参数更宽松。  
④.当子类的方法实现父类的抽象方法时，方法的后置条件（即方法的返回值）要比父类更严格。

**3)依赖倒转原则:** 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。即针对接口编程，不要针对实现编程. 依赖倒转其实就是谁也不要依靠谁，除了约定的接口，大家都可以灵活自如。依赖倒转可以说是面向对象设计的标志，用哪种语言来编写程序不重要，如果编写时考虑的都是如何针对抽象编程而不是针对细节编程，即程序中所有的依赖关系都是终止于抽象类或者接口，那就是面向对象的设计，反之那就是过程化的设计了。如果设计的各个部件或类相互依赖，这样就是耦合度高，难以维护和扩展，这也就体现不出面向对象的好处了。

**4)接口隔离原则:** 建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。也就是说，我们要为各个类建立专用的接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。在程序设计中，依赖几个专用的接口要比依赖一个综合的接口更灵活。接口是设计时对外部设定的“契约”，通过分散定义多个接口，可以预防外来变更的扩散，提高系统的灵活性和可维护性。

5)迪米特法则: 强调了类之间的松耦合，类之间的耦合越弱,越有利于复用，一个处在弱耦合的类被修改，不会对有关系的类造成影响，也就是说，信息的隐藏促进了软件的复用。

**6)单一职责原则**: 不要存在多于一个导致类变更的原因。通俗的说，即一个类只负责一项职责，应该仅有一个引起它变化的原因.

遵循单一职责原的优点有：  
①.可以降低类的复杂度，一个类只负责一项职责，其逻辑肯定要比负责多项职责简单的多；  
②.提高类的可读性，提高系统的可维护性；  
③.变更引起的风险降低，变更是必然的，如果单一职责原则遵守的好，当修改一个功能时，可以显著降低对其他功能的影响。

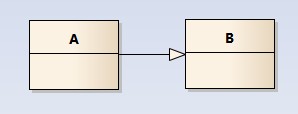
**7)组合/聚合复用原则**: 要尽量的使用合成和聚合，而不是继承关系达到复用的目的  
该原则就是在一个新的对象里面使用一些已有的对象，使之成为新对象的一部分：新的对象通过向这些对象的委派达到复用已有功能的目的。  
      其实这里最终要的地方就是区分“has-a”和“is-a”的区别。相对于合成和聚合，  
继承的缺点在于：父类的方法全部暴露给子类。父类如果发生变化，子类也得发生变化。聚合的复用的时候就对另外的类依赖的比较的少。。  
**合成/聚合复用**① 优点：  
新对象存取成分对象的唯一方法是通过成分对象的接口；  
 这种复用是黑箱复用，因为成分对象的内部细节是新对象所看不见的；  
 这种复用支持包装；  
这种复用所需的依赖较少；  
每一个新的类可以将焦点集中在一个任务上；  
 这种复用可以在运行时动态进行，新对象可以使用合成/聚合关系将新的责任委派到合适的对象。  
② 缺点：  
通过这种方式复用建造的系统会有较多的对象需要管理。  
**继承复用**① 优点：  
  新的实现较为容易，因为基类的大部分功能可以通过继承关系自动进入派生类；  
  修改或扩展继承而来的实现较为容易。  
② 缺点：  
  继承复用破坏包装，因为继承将基类的实现细节暴露给派生类，这种复用也称为白箱复用；如果基类的实现发生改变，那么派生类的实现也不得不发生改变；从基类继承而来的实现是静态的，不可能在运行时发生改变，不够灵活。

1. UML类图

类图的解释:

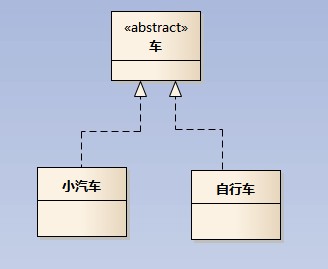
1).泛化关系: 类的继承结构表现在UML中为：泛化(generalize)与实现(realize), 继承关系为 is-a的关系；两个对象之间如果可以用 is-a 来表示，就是继承关系; **泛化关系表现为继承非抽象类**.

泛化关系用一条带空心箭头的直接表示



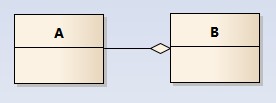
2).实现关系: 实现关系表现为继承抽象类(实现接口)

实现关系用一条带空心箭头的虚线表示



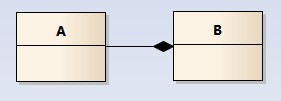
3).聚合关系: 表示实体对象之间的关系，表示整体由部分构成的语义；例如一个部门由多个员工组成；与组合关系不同的是，整体和部分不是强依赖的，即使整体不存在了，部分仍然存在；例如， 部门撤销了，人员不会消失，他们依然存在.

聚合关系用一条带空心菱形箭头的直线表示，如下图表示A聚合到B上，或者说B由A组成.

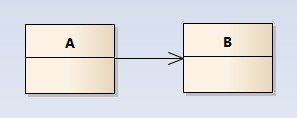


4).组合关系: 与聚合关系一样，组合关系同样表示整体由部分构成的语义；比如公司由多个部门组成, 但组合关系是一种强依赖的特殊聚合关系，如果整体不存在了，则部分也不存在了；例如， 公司不存在了，部门也将不存在了.

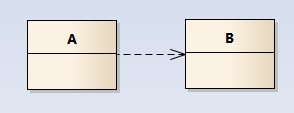
组合关系用一条带实心菱形箭头直线表示，如下图表示A组成B，或者B由A组成



4).关联关系: 描述不同类的对象之间的结构关系；它是一种静态关系， 通常与运行状态无关，一般由常识等因素决定的；它一般用来定义对象之间静态的、天然的结构； 所以，关联关系是一种“强关联”的关系, 比如，乘车人和车票之间就是一种关联关系；学生和学校就是一种关联关系, 关联关系默认不强调方向，表示对象间相互知道；如果特别强调方向，如下图，表示A知道B，但 B不知道A.( 在最终代码中，**关联对象通常是以成员变量的形式实现的**)



5). 依赖关系: 与关联关系不同的是，它是一种临时性的关系，通常在运行期间产生，并且随着运行时的变化；依赖关系也可能发生变化. 依赖也有方向，双向依赖是一种非常糟糕的结构，我们总是应该保持单向依赖，杜绝双向依赖的产生.( **在最终代码中，依赖关系体现为类构造方法及类方法的传入参数，箭头的指向为调用关系；依赖关系除了临时知道对方外，还是“使用”对方的方法和属性**)



1. 设计模式

1).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. 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。

类图:

代码: 四种实现线程安全的单例模式

i).Eager initialization:饿汉式

public class Singleton{

//JVM保证在任何线程访问singleton静态变量之前先创建此实例

private static Singleton singleton = new Singleton();

//私有构造函数,保证外界无法直接实例化

private Singleton(){}

//提供外部访问

public static Singleton getInstance(){

return singleton;

}

}

ii).Lazy initialization: 懒汉式, 在开销比较大,希望用到时才创建就要考虑延迟实例化

public class Singleton{

private static Singleton singleton;

//私有构造函数,保证外界无法直接实例化

private Singleton(){}

//提供外部访问

public static synchronized Singleton getInstance(){

if(singleton == null){

singleton = new Singleton();

}

return singleton;

}

}

iii).方法的同步会导致性能下降,使用双检锁,保证一次初始化成功,其余的快速返回.

双检锁:尽量将加锁推迟,只在需要时加锁.

public class Singleton{

//Java使用双重检查锁定机制,由于Java编译器和JIT的优化原因系统无法保证程序按期望的顺序执行,使用volatile声明的变量可以强制屏蔽编译器和JIT的优化工作(JVM根据CPU的特性,适当的重排序机器指令,使得最大限度的发挥CPU的性能,但是如果没有volatile可能出现一个线程t1的A操作和另一个线程的B操作之间对instance的读写没有happens-before,可能造成t1的A操作还没完全构造成功,但t2的B操作已经看到instance为空,这样t2就会返回未完全构造的instance的引用,就会导致t2对instance的引用出错.).

private volatile static Singleton singleton;

//私有构造函数,保证外界无法直接实例化

private Singleton(){}

//提供外部访问

public static Singleton getInstance(){

if(singleton == null){

synchronized(Singleton.class){

if(singleton == null){

singleton = new Singleton();

}

}

}

return singleton;

}

}

iv).Lazy initialization holder class方式实现: 满足双检锁但是没有显示的同步

public class Singleton{

//私有构造函数,保证外界无法直接实例化

private Singleton(){}

private static class SingletonHolder{

private static final Singleton INSTANCE = new Singleton();

}

//提供外部访问

public static Singleton getInstance(){

return SingletonHolder.INSTANCE;

}

}

v).枚举实现:

public enum Singleton {

// 定义枚举元素，他就是Singleton的一个实例

INSTANCE;

public void doSomething() {

// do something

}

}

调用方式:

Singleton singleton = Singleton.INSTANCE;

singleton.doSomething();

根据JVM规范,当某对象第一次调用Singleton. getInstance时, Singleton类被首次主动调用,JVM对其进行初始化, 此时先初始化类中的静态方法块(静态方法块先于构造函数调用),所以先调用getInstance方法, 然后该方法中需要用到SingletonHolder实例,所以要初始化SingletonHolder类,此时INSTANCE被赋值时调用了Singleton的构造函数,完成实例化过程并返回;当其他线程在调用Singleton的getInstance方法时,因为已经被初始化过了,不会再进行初始化步骤,所以直接返回INSTANCE常量即同一个Singleton实例.

**优点:** 提供了对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高系统的性能; 允许可变数目的实例。我们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”。因为单例类既充当了工厂角色，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的本身的功能融合到一起; 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.

2).工厂方法模式:

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(Polymorphic 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.

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

类图:

代码:

**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，无须关心创建细节，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，无须修改抽象工厂和抽象产品提供的接口，无须修改客户端，也无须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”.

**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，在一定程度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.

**使用场景:** 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.

3).抽象工厂模式:

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

类图:

代码:

**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，无须修改已有系统，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）.

**适用场景:** 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，而每次只使用其中某一产品族; 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.( 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)

4).模板方法模式:

**模板方法模式是类的行为模式。准备一个抽象类，将部分逻辑以具体方法以及具体构造函数的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现.**

**类图:**

**代码:**

**优点:** 模板方法模式在一个类中形式化地定义算法，而由它的子类实现细节的处理;模板方法是一种代码复用的基本技术。它们在类库中尤为重要，它们提取了类库中的公共行为;模板方法模式导致一种反向的控制结构，这种结构有时被称为“好莱坞法则” ，即“别找我们，,我们找你”通过一个父类调用其子类的操作(而不是相反的子类调用父类)，通过对子类的扩展增加新的行为，符合“开闭原则”.

**缺点:** 每个不同的实现都需要定义一个子类，这会导致类的个数增加，系统更加庞大，设计也更加抽象，但是更加符合“单一职责原则”，使得类的内聚性得以提高

**使用场景**: 考虑一个计算存款利息的例子。假设系统需要支持两种存款账号，即货币市场(Money Market)账号和定期存款(Certificate of Deposite)账号。这两种账号的存款利息是不同的，因此，在计算一个存户的存款利息额时，必须区分两种不同的账号类型。这个系统的总行为应当是计算出利息，这也就决定了作为一个模板方法模式的顶级逻辑应当是利息计算。由于利息计算涉及到两个步骤：一个基本方法给出账号种类，另一个基本方法给出利息百分比。这两个基本方法构成具体逻辑，因为账号的类型不同，所以具体逻辑会有所不同。显然，系统需要一个抽象角色给出顶级行为的实现，而将两个作为细节步骤的基本方法留给具体子类实现

5).建造者模式:

将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. 建造者模式是一步一步创建一个复杂的对象，它允许用户只通过指定复杂对象的类型和内容就可以构建它们，用户不需要知道内部的具体构建细节。建造者模式属于对象创建型模式。根据中文翻译的不同，建造者模式又可以称为生成器模式.

类图:

代码:

**优点:** 在建造者模式中， 客户端不必知道产品内部组成的细节，将产品本身与产品的创建过程解耦，使得相同的创建过程可以创建不同的产品对象; 每一个具体建造者都相对独立，而与其他的具体建造者无关，因此可以很方便地替换具体建造者或增加新的具体建造者， 用户使用不同的具体建造者即可得到不同的产品对象; 可以更加精细地控制产品的创建过程 。将复杂产品的创建步骤分解在不同的方法中，使得创建过程更加清晰，也更方便使用程序来控制创建过程; 增加新的具体建造者无须修改原有类库的代码，指挥者类针对抽象建造者类编程，系统扩展方便，符合“开闭原则”.

**缺点:** 建造者模式所创建的产品一般具有较多的共同点，其组成部分相似，如果产品之间的差异性很大，则不适合使用建造者模式，因此其使用范围受到一定的限制; 如果产品的内部变化复杂，可能会导致需要定义很多具体建造者类来实现这种变化，导致系统变得很庞大.

**使用场景:** 需要生成的产品对象有复杂的内部结构，这些产品对象通常包含多个成员属性; 需要生成的产品对象的属性相互依赖，需要指定其生成顺序; 对象的创建过程独立于创建该对象的类。在建造者模式中引入了指挥者类，将创建过程封装在指挥者类中，而不在建造者类中; 隔离复杂对象的创建和使用，并使得相同的创建过程可以创建不同的产品.( 在很多游戏软件中，地图包括天空、地面、背景等组成部分，人物角色包括人体、服装、装备等组成部分，可以使用建造者模式对其进行设计，通过不同的具体建造者创建不同类型的地图或人物)

6).代理模式:

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.**( java.lang.reflect.Proxy)**

类图:

代码:

**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:** 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.

**适用场景:** 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; 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.

Java中的三种代理模式:

**静态代理:** 静态代理在使用时,需要定义接口或者父类,被代理对象与代理对象一起实现相同的接口或者是继承相同父类

代码示例:

**动态代理:** 代理对象,不需要实现接口; 代理对象的生成,是利用JDK的API,动态的在内存中构建代理对象(需要我们指定创建代理对象/目标对象实现的接口的类型); 动态代理也叫做:JDK代理,接口代理.

JDK中生成代理对象的API  
代理类所在包:java.lang.reflect.Proxy  
JDK实现代理只需要使用newProxyInstance方法,但是该方法需要接收三个参数,完整的写法是:

static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces,InvocationHandler h )

注意该方法是在Proxy类中是静态方法,且接收的三个参数依次为:

* ClassLoader loader,:指定当前目标对象使用类加载器,获取加载器的方法是固定的
* Class<?>[] interfaces,:目标对象实现的接口的类型,使用泛型方式确认类型
* InvocationHandler h:事件处理,执行目标对象的方法时,会触发事件处理器的方法,会把当前执行目标对象的方法作为参数传入

代码示例:

**Cglib代理:** 静态代理和动态代理模式都是要求目标对象是实现一个接口的目标对象,但是有时候目标对象只是一个单独的对象,并没有实现任何的接口,这个时候就可以使用以目标对象子类的方式类实现代理,这种方法就叫做:Cglib代理.

Cglib代理,也叫作子类代理,它是在内存中构建一个子类对象从而实现对目标对象功能的扩展.

* JDK的动态代理有一个限制,就是使用动态代理的对象必须实现一个或多个接口,如果想代理没有实现接口的类,就可以使用Cglib实现.
* Cglib是一个强大的高性能的代码生成包,它可以在运行期扩展java类与实现java接口.它广泛的被许多AOP的框架使用,例如Spring AOP和synaop,为他们提供方法的interception(拦截)
* Cglib包的底层是通过使用一个小而块的字节码处理框架ASM来转换字节码并生成新的类.不鼓励直接使用ASM,因为它要求你必须对JVM内部结构包括class文件的格式和指令集都很熟悉.

Cglib子类代理实现方法:  
①.需要引入cglib的jar文件,但是Spring的核心包中已经包括了Cglib功能,所以直接引入spring-core-3.2.5.jar即可.  
②.引入功能包后,就可以在内存中动态构建子类  
③.代理的类不能为final,否则报错  
④.目标对象的方法如果为final/static,那么就不会被拦截,即不会执行目标对象额外的业务方法.

示例代码:

7).原型模式:

用原型实例指定创建对象的种类，并通过拷贝这些原型创建新的对象**(Object)**

类图:

代码:

原型模式主要用于对象的复制，它的核心是就是类图中的原型类Prototype。Prototype类需要具备以下两个条件：

1. .实现Cloneable接口。在java语言有一个Cloneable接口，它的作用只有一个，就是在运行时通知虚拟机可以安全地在实现了此接口的类上使用clone方法。在java虚拟机中，只有实现了这个接口的类才可以被拷贝，否则在运行时会抛出CloneNotSupportedException异常。
2. .重写Object类中的clone方法。Java中，所有类的父类都是Object类，Object类中有一个clone方法，作用是返回对象的一个拷贝，但是其作用域protected类型的，一般的类无法调用，因此，Prototype类需要将clone方法的作用域修改为public类型。

**优点:**  使用原型模式创建对象比直接new一个对象在性能上要好的多，因为Object类的clone方法是一个本地方法，它直接操作内存中的二进制流，特别是复制大对象时，性能的差别非常明显; 简化对象的创建，使得创建对象就像我们在编辑文档时的复制粘贴一样简单.

**缺点:** 使用原型模式复制对象不会调用类的构造方法。因为对象的复制是通过调用Object类的clone方法来完成的，它直接在内存中复制数据，因此不会调用到类的构造方法。不但构造方法中的代码不会执行，甚至连访问权限都对原型模式无效; 深拷贝与浅拷贝。Object类的clone方法只会拷贝对象中的基本的数据类型，对于数组、容器对象、引用对象等都不会拷贝，这就是浅拷贝。如果要实现深拷贝，必须将原型模式中的数组、容器对象、引用对象等另行拷贝

**适用场景:** 在需要重复地创建相似对象时可以考虑使用原型模式。比如需要在一个循环体内创建对象，假如对象创建过程比较复杂或者循环次数很多的话，使用原型模式不但可以简化创建过程，而且可以使系统的整体性能提高很多.

8).中介者模式:

用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。中介者模式又称为调停者模式，它是一种对象行为型模式.**(JDK并发包中的Executor,ExecutorService,ScheduledExecutorService)**

类图:

代码:

**优点:** 简化了对象之间的交互; 将各同事解耦; 减少子类生成; 可以简化各同事类的设计和实现.

**缺点:** 在具体中介者类中包含了同事之间的交互细节，可能会导致具体中介者类非常复杂，使得系统难以维护.

**适用场景:** 系统中对象之间存在复杂的引用关系，产生的相互依赖关系结构混乱且难以理解; 一个对象由于引用了其他很多对象并且直接和这些对象通信，导致难以复用该对象; 想通过一个中间类来封装多个类中的行为，而又不想生成太多的子类。可以通过引入中介者类来实现，在中介者中定义对象; 交互的公共行为，如果需要改变行为则可以增加新的中介者类.( 某论坛系统欲增加一个虚拟聊天室，允许论坛会员通过该聊天室进行信息交流，普通会员(CommonMember)可以给其他会员发送文本信息，钻石会员(DiamondMember)既可以给其他会员发送文本信息，还可以发送图片信息。该聊天室可以对不雅字符进行过滤，如“日”等字符；还可以对发送的图片大小进行控制。用中介者模式设计该虚拟聊天室)

9).命令模式:

将一个请求封装为一个对象，从而使我们可用不同的请求对客户进行参数化；对请求排队或者记录请求日志，以及支持可撤销的操作。命令模式是一种对象行为型模式，其别名为动作(Action)模式或事务(Transaction)模式.**(所有实现Runnable)**

* 每一个命令都是一个操作：请求的一方发出请求，要求执行一个操作；接收的一方收到请求，并执行操作。
* 命令模式允许请求的一方和接收的一方独立开来，使得请求的一方不必知道接收请求的一方的接口，更不必知道请求是怎么被接收，以及操作是否被执行、何时被执行，以及是怎么被执行的。
* 命令模式使请求本身成为一个对象，这个对象和其他对象一样可以被存储和传递。
* 命令模式的关键在于引入了抽象命令接口，且发送者针对抽象命令接口编程，只有实现了抽象命令接口的具体命令才能与接收者相关联。

类图:

代码:

**优点:** 降低系统的耦合度; 新的命令可以很容易地加入到系统中; 可以比较容易地设计一个命令队列和宏命令（组合命令）; 可以方便地实现对请求的Undo和Redo.

**缺点:** 使用命令模式可能会导致某些系统有过多的具体命令类。因为针对每一个命令都需要设计一个具体命令类，因此某些系统可能需要大量具体命令类，这将影响命令模式的使用.**适用场景:** 系统需要将请求调用者和请求接收者解耦，使得调用者和接收者不直接交互; 系统需要在不同的时间指定请求、将请求排队和执行请求; 系统需要支持命令的撤销(Undo)操作和恢复(Redo)操作; 系统需要将一组操作组合在一起，即支持宏命令.( 很多系统都提供了宏命令功能，如UNIX平台下的Shell编程，可以将多条命令封装在一个命令对象中，只需要一条简单的命令即可执行一个命令序列，这也是命令模式的应用实例之一)

10).观察者模式:

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。观察者模式又叫做发布-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(publishsubscribe)。目标是通知的发布者，它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。

类图:

代码:

**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:**

* 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，可能导致系统崩溃。
* 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。

**适用场景:**

* 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。

应用实例:如某电子商务网站可以在执行发送操作后给用户多个发送商品打折信息，某团队战斗游戏中某队友牺牲将给所有成员提示等等，凡是涉及到一对一或者一对多的对象交互场景都可以使用观察者模式.

11).状态模式:

允许一个对象在其内部状态改变时改变它的行为，对象看起来似乎修改了它的类。其别名为状态对象(Objects for States)，状态模式是一种对象行为型模式.

* 状态模式描述了对象状态的变化以及对象如何在每一种状态下表现出不同的行为。
* 状态模式的关键是引入了一个抽象类来专门表示对象的状态，这个类我们叫做抽象状态类，而对象的每一种具体状态类都继承了该类，并在不同具体状态类中实现了不同状态的行为，包括各种状态之间的转换。

类图:

代码:

**优点:**

* 封装了转换规则。
* 枚举可能的状态，在枚举状态之前需要确定状态种类。
* 将所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为。
* 允许状态转换逻辑与状态对象合成一体，而不是某一个巨大的条件语句块。
* 可以让多个环境对象共享一个状态对象，从而减少系统中对象的个数。

**缺点:**

* 状态模式的使用必然会增加系统类和对象的个数。
* 状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱。
* 状态模式对“开闭原则”的支持并不太好，对于可以切换状态的状态模式，增加新的状态类需要修改那些负责状态转换的源代码，否则无法切换到新增状态；而且修改某个状态类的行为也需修改对应类的源代码。

**适用场景:**

* 对象的行为依赖于它的状态（属性）并且可以根据它的状态改变而改变它的相关行为。
* 代码中包含大量与对象状态有关的条件语句，这些条件语句的出现，会导致代码的可维护性和灵活性变差，不能方便地增加和删除状态，使客户类与类库之间的耦合增强。在这些条件语句中包含了对象的行为，而且这些条件对应于对象的各种状态。

如在政府OA办公系统中，一个批文的状态有多种：尚未办理；正在办理；正在批示；正在审核；已经完成等各种状态，而且批文状态不同时对批文的操作也有所差异。使用状态模式可以描述工作流对象（如批文）的状态转换以及不同状态下它所具有的行为.

12).策略模式:

定义一系列算法，将每一个算法封装起来，并让它们可以相互替换。策略模式让算法独立于使用它的客户而变化，也称为政策模式(Policy)。

* 策略模式是一个比较容易理解和使用的设计模式，策略模式是对算法的封装，它把算法的责任和算法本身分割开，委派给不同的对象管理。策略模式通常把一个系列的算法封装到一系列的策略类里面，作为一个抽象策略类的子类。用一句话来说，就是“准备一组算法，并将每一个算法封装起来，使得它们可以互换”。
* 在策略模式中，应当由客户端自己决定在什么情况下使用什么具体策略角色。
* 策略模式仅仅封装算法，提供新算法插入到已有系统中，以及老算法从系统中“退休”的方便，策略模式并不决定在何时使用何种算法，算法的选择由客户端来决定。这在一定程度上提高了系统的灵活性，但是客户端需要理解所有具体策略类之间的区别，以便选择合适的算法，这也是策略模式的缺点之一，在一定程度上增加了客户端的使用难度。

类图:

代码:

**优点**

* 策略模式提供了对“开闭原则”的完美支持，用户可以在不修改原有系统的基础上选择算法或行为，也可以灵活地增加新的算法或行为。
* 策略模式提供了管理相关的算法族的办法。
* 策略模式提供了可以替换继承关系的办法。
* 使用策略模式可以避免使用多重条件转移语句。

**缺点**

* 客户端必须知道所有的策略类，并自行决定使用哪一个策略类。
* 策略模式将造成产生很多策略类，可以通过使用享元模式在一定程度上减少对象的数量。

**适用场景:**

如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用策略模式可以动态地让一个对象在许多行为中选择一种行为。

一个系统需要动态地在几种算法中选择一种。

如果一个对象有很多的行为，如果不用恰当的模式，这些行为就只好使用多重的条件选择语句来实现。

不希望客户端知道复杂的、与算法相关的数据结构，在具体策略类中封装算法和相关的数据结构，提高算法的保密性与安全性。

13).适配器模式:

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，其别名为包装器(Wrapper)。适配器模式既可以作为类结构型模式，也可以作为对象结构型模式.**(集合类, IO流)**

类图:

代码:

**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用现有的适配者类，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，对于客户端类来说是透明的，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”。

**缺点:**

类适配器模式的缺点如下：

对于Java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。

**适用场景:**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。(JDBC使得Java语言程序能够与数据库连接，并使用SQL语言来查询和操作数据。JDBC给出一个客户端通用的抽象接口，每一个具体数据库引擎（如SQL Server、Oracle、MySQL等）的JDBC驱动软件都是一个介于JDBC接口和数据库引擎接口之间的适配器软件。抽象的JDBC接口和各个数据库引擎API之间都需要相应的适配器软件，这就是为各个不同数据库引擎准备的驱动程序)

14).桥梁模式:

将抽象部分与它的实现部分分离，使它们都可以独立地变化。它是一种对象结构型模式，又称为柄体(Handle and Body)模式或接口(Interface)模式.**(JDBC)**

理解桥接模式，重点需要理解如何将抽象化(Abstraction)与实现化(Implementation)脱耦，使得二者可以独立地变化。

* 抽象化：抽象化就是忽略一些信息，把不同的实体当作同样的实体对待。在面向对象中，将对象的共同性质抽取出来形成类的过程即为抽象化的过程。
* 实现化：针对抽象化给出的具体实现，就是实现化，抽象化与实现化是一对互逆的概念，实现化产生的对象比抽象化更具体，是对抽象化事物的进一步具体化的产物。
* 脱耦：脱耦就是将抽象化和实现化之间的耦合解脱开，或者说是将它们之间的强关联改换成弱关联，将两个角色之间的继承关系改为关联关系。桥接模式中的所谓脱耦，就是指在一个软件系统的抽象化和实现化之间使用关联关系（组合或者聚合关系）而不是继承关系，从而使两者可以相对独立地变化，这就是桥接模式的用意。

类图:

代码:

**优点:**

* 分离抽象接口及其实现部分。
* 桥接模式有时类似于多继承方案，但是多继承方案违背了类的单一职责原则（即一个类只有一个变化的原因），复用性比较差，而且多继承结构中类的个数非常庞大，桥接模式是比多继承方案更好的解决方法。
* 桥接模式提高了系统的可扩充性，在两个变化维度中任意扩展一个维度，都不需要修改原有系统。
* 实现细节对客户透明，可以对用户隐藏实现细节。

**缺点:**

* 桥接模式的引入会增加系统的理解与设计难度，由于聚合关联关系建立在抽象层，要求开发者针对抽象进行设计与编程。 - 桥接模式要求正确识别出系统中两个独立变化的维度，因此其使用范围具有一定的局限性。

**适用场景:**

* 如果一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的继承联系，通过桥接模式可以使它们在抽象层建立一个关联关系。
* 抽象化角色和实现化角色可以以继承的方式独立扩展而互不影响，在程序运行时可以动态将一个抽象化子类的对象和一个实现化子类的对象进行组合，即系统需要对抽象化角色和实现化角色进行动态耦合。
* 一个类存在两个独立变化的维度，且这两个维度都需要进行扩展。
* 虽然在系统中使用继承是没有问题的，但是由于抽象化角色和具体化角色需要独立变化，设计要求需要独立管理这两者。
* 对于那些不希望使用继承或因为多层次继承导致系统类的个数急剧增加的系统，桥接模式尤为适用。

15).装饰模式:

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.( **所有子类java.io.InputStream，OutputStream，Reader并Writer有一个构造函数取相同类型的实例**)

* 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。

类图:

代码:

**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。