面向对象的三大特征

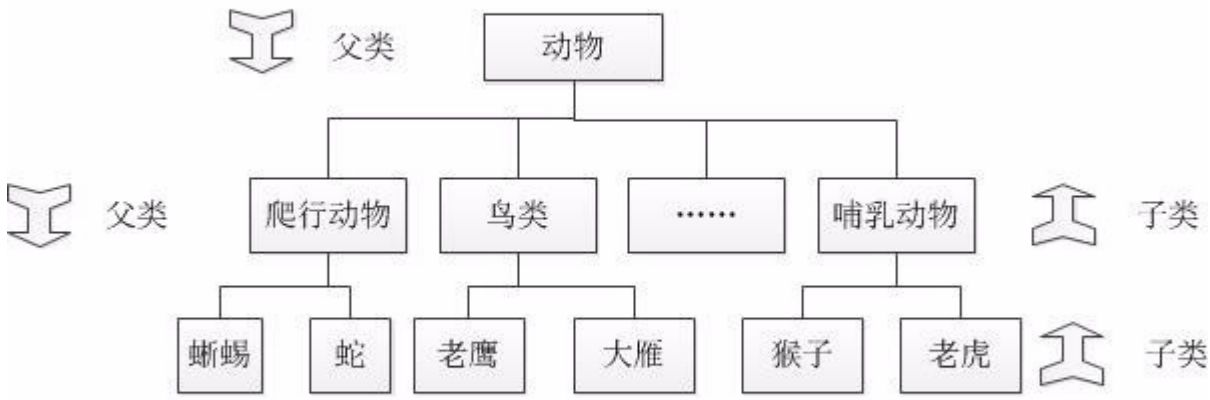
看法一

（一）封装

封装是类将内部数据隐藏。
封装为用户提供对象的属性和行为的接口，用户通过这些接口使用这些类，无须知道这些类内部是如何构成的，同时用户不能操作类中的内部数据。

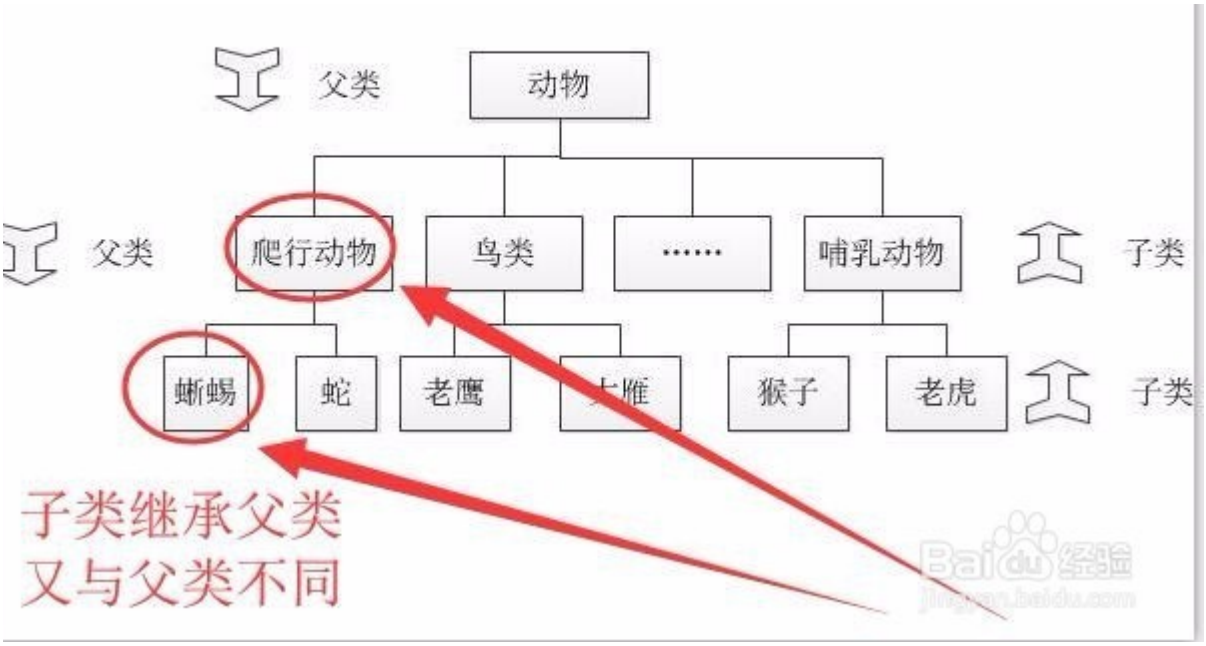
（二）继承

首先，我们定义一个父类，本例为动物。



（三）多态

将父类对象应用于子类的特征就是多态，每一个父类下都可以继承子类，如果将子类对象统一看作是父类的实例对象，子类都调用了父类的属性和方法，但又与父类不相同。



看法二

一个面向对象的语言在处理对象时，必须遵循的三个原则是：封装、继承和多态。

（1）封装

所谓“封装”，就是用一个框架把数据和代码组合在一起，形成一个对象。遵循面向对象数据抽象的要求，一般数据都被封装起来，也就是外部不能直接访问对象的数据，外部能见到的只有提供给外面访问的公共操作（也称接口，对象之间联系的渠道）。在C#中，类是对象封装的工具，对象则是封装的基本单元。

封装的对象之间进行通信的一种机制叫做消息传递。消息是向对象发出的服务请求，是面向对象系统中对象之间交互的途径。消息包含要求接收对象去执行某些活动的信息以及完成要求所需要的其他信息（参数）。发送消息的对象不需要知道接收消息的对象如何对请求予以响应。接收者接收了消息，它就承担了执行指定动作的责任，作为消息的答复，接收者将执行某个方法，来满足所接收的请求。

封装其实就是信息隐藏，隐藏一个对象的本质，让用户不再注意那些细节·提供一些向外的接口供别人使用。　就像电视的内部已经被封起来，你不需要知道它的内部是由哪些零件组成、如何工作。你只知道用遥控器来控制就好。

（2）继承

世界是复杂的，在大千世界中事物有很多的相似性，这种相似性是人们理解纷繁事物的一个基础。因数事物之间往往具有某种“继承”关系。比如，儿子与父亲往往有许多相似之处，因数儿子从父亲那里遗传了许多特性；汽车与卡车、轿车、客车之间存在着一般化与具体化的关系，它们都可以用继承来实现。

继承是面向对象编程技术的一块基石，通过它可以创建分等级层次的类。例如，创建一个汽车通用类，它定义了汽车的一般属性（如：车轮、方向盘、发动机、车门等）和操作方法（如：前进、倒退、刹车、转变等到）。从这个已有的类可以通过继承的方法派生出新的子类，卡车、轿车、客车等，它们都是汽车类的更具体的类，每个具体的类还可以增加自己一些特有的东西。继承是父类和子类之间共享数据和方法的机制，通常把父类称为基类，子类称为派生类。一个基类可以有任意数目的派生类，从基类派生出的类还可以被派生，一群通过继承相联系的类就构成了树型层次结构。

如果一个类继承两个或两个以上直接基类，这样的继承结构被称为多重继承或多继承。

尽管多继承从形式上看比较直观，但在现实上多继承可能引起继承操作或属性的冲突。当今的很多语言已不再支持多继承，C#语言也对多继承的使用进行了限制，它通过接口来实现。接口可以从多个基接口继承。接口可以包含方法、属性、事件和索引器。一个典型的接口就是一个方法声明的列表，接口本身不提供它所定义的成员的实现。所以接口不能被实例化，一个实现接口的类再以适当的方法定义接口中声明的方法。

有仅如此，C#的接口概念十分适用于组件编程。在组件和组件之间、组件和客户之间都通过接口进行交互。继承可以理解为基类代码的复用. 当一个对象可以描述为另外一个对象的时候用继承(is-a)的关系. 当一个可以可以有另外一个对象的时候用组合(has-a)的关系. 当一个对象可以包含某个行为的时候用接口(can-do)的关系.

(3) 多态性

多态性就其字面上的意思是：多种形式或多种形态。在面向对象编程中，多态是指同一个消息或操作作用于不同的对象，可以有不同的解释，产生不同的执行结果。例如，问甲同学：“现在几点钟”，甲看一看表回答说：“3 点 15 分”，又问乙同学：“现在几点钟”，乙想一想回答说：“大概 3 点多钟”，又问丙同学：“现在几点钟”，两干脆回答说：“不知道”。这就是同一个消息发给不同的对象，不同的对象可以做出不同的反应。

同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果， 这就是多态性。多态性通过派生类重载基类中的虚函数型方法来实现。 C#支持两种类型的多态性 编译时的多态性为我们提供了运行速度快的特点，而运行时的多态性则带来了高度灵活和抽象的特点。 编译时的多态性 编译时的多态性是通过重载来实现的。对于非虚的成员来说，系统在编译时，根据传递的参数、返回的类型等信息决定实现何种操作。 运行时的多态性 运行时的多态性就是指直到系统运行时，才根据实际情况决定实现何种操作。C#中，运行时的多态性通过虚成员实现。

看法三

面向对象程序设计的 4 个主要特点是什么？

抽象

封装

继承

多态

类和对象体现了**抽象**和**封装**

基于以上四点的程序称为面向对象的程序设计。

什么是**类**？

前边我们学习过c++中的结构体，struct的使用是先声明，然后定义一个该结构体的变量，最后再初始化（赋值）

其实结构体的声明当中有很多不同数据类型的成员，在一个结构体当中，这就是对属性的封装，但是结构体与类相比较，还缺少了一个重要的环节，结构体当中只有属性，只封装了属性，但没有封装相应的操作，也就是没有封装操作成员属性的函数。

类当中有两个方面，一是封装属性，二是封装函数。在自己声明的类当中要既有属性又有函数，这就是类。

什么是**对象**？

在结构体中声明一个结构体就相当于我们设计了一个类，定义一个结构体就相当于我们定义了一个对象

客观世界中任何一个事物都可以看成一个对象（Object），对象可以是自然物体（如汽车、狗熊）也可以是一种逻辑(如班级、连队）。

任何一个对象都有两个要素：一个是静态特征，如班级专业，班级人数，所在教室，这种静态特征称为属性。

一个是动态特征，如班级学习、班级开会、班级体育比赛，这种动态特征称为行为（或功能）。

封装的意义？

把一部分或全部属性和部分功能（函数）对外界屏蔽，就是从外界（类的大括号之外）看不到，不可知，这就是封装的意义。

实际上就是用了一个后边会介绍的关键字 private 私有化关键字来完成，就是隐蔽，比如账号的密码，你如果玩网络游戏，你的账号密码别人都可以看到，访问，所以要把它封装。

封装两个方面的含义：一是将有关数据和操作代码封装在对象当中，形成一个基本单位，各个对象之间相对独立互不干扰。

二是将对象中某些属性和操作私有化，已达到数据和操作信息隐蔽，有利于数据安全，防止无关人员修改。

类与对象之间的关系是什么？

程序设计当中，常用到抽象这个词，这个词就是解释类与对象之间关系的词。

类与对象之间的关系就是抽象的关系。

一句话来说明：类是对象的抽象，而对象则是类得特例，即类的具体表现形式。

举例说明上面的那句话：

我们都是人类，人类是不是有自己的属性，属性有年龄、名字、ID、性别，人类是不是有功能，即操作能力，能说话、能吃饭、能睡觉，能思考解决问题。

我们大家都是人类，我们彼此却又不相同，我们每一个人具有自己的属性的能力，每一个人就是人类的某一个对象。

把我们共同的属性和功能向上抽取，这就是抽象，抽象程一个人类。

继承

面向对象的继承是为了软件重用，简单理解就是代码复用，把重复使用的代码精简掉的一种手段。

如何精简，当一个类中已经有了相应的属性和操作的代码，而另一个类当中也需要写重复的代码，那么就用继承方法，把前面的类当成父类，后面的类当成子类，子类继承父类，理所当然。就用一个关键字 extends 就完成了代码的复用。

多态

没有继承就没有多态，继承是多态的前提。

举例来说明多态

猫、狗、鸟都是动物

动物不同于植物，猫、狗、鸟都可以叫、都可以吃

这些功能可以同时继承自动物类，猫类、狗类、鸟类都是动物类的子类

但是猫的叫声是“喵喵喵” ，狗的叫声是“汪汪汪”，鸟的叫声是“吱吱支”

猫爱吃鱼，狗爱吃骨头，鸟爱吃虫子

这就是虽然继承自同一父类，但是相应的操作却各不相同，这叫多态。

由继承而产生的不同的派生类，其对象对同一消息会做出不同的响应。

知道了以上概念就可以进行面向对象编程（Object Oriented Programming，简称OOP）

看法四

基本特征

抽象

将一些事物的共性抽离出来归为一个类。

如对于动物，具有生命体征、活动能力等区别于其它事物的共同特征

封装

有选择地隐藏和暴露数据和方法

比如有U盘这个类，我希望隐藏内部组成和实现，只暴露USB接口以供使用

继承

子类可以直接使用父类的部分数据和方法，可以有选择的扩展

比如鸟是动物，但鸟扩展了飞行的能力。

多态

同一类的对象调用相同方法可以表现出不同的行为

比如动物实现了 say()方法，猴子、马等动物重写了 say()方法来表现不同的交流语言。

看法五

面向对象的三个基本特征

原文网址：http://blog.163.com/gogles@yeah/blog/static/161688177201031164026 3/

1.抽象与封装:

抽象是把系统中需要处理的数据和在这些数据上的操作结合在一起，根据功能、性质和用途等因素抽象成不同的抽象数据类型。每个抽象数据类型既包含了数据，又包含了针对这些数据的授权操作。在面向对象的程序设计中，抽象数据类型是用“类”这种结构来实现的，每个类里都封装了相关的数据和操作。
封装是指利用抽象数据类型和基于数据的操作结合在一起，数据被保护在抽象数据类型的内部，系统的其他部分只有通过包裹在数据之外被授权的操作，才能与这个抽象数据类型进行交互。

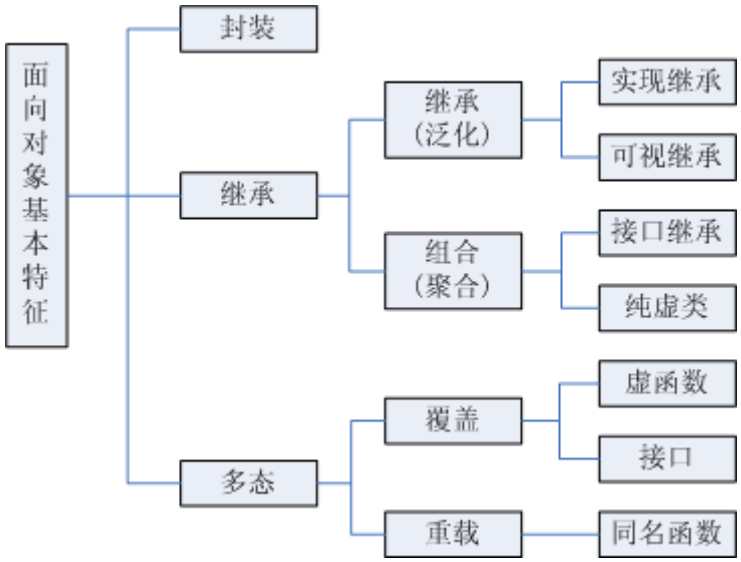
2. 继承：
它是与传统方法不同的一个最有特色的方法。它是面向对象的程序中两个类之间的一种关系，即一个类可以从另一个类（即它的父类）继承状态和行为。继承父类的类称为子类。
继承的优越性：通过使用继承，程序员可以在不同的子类中多次重新使用父类中的代码，使程序结构清晰，易于维护和修改，而子类又可以提供一些特殊的行为，这些特殊的行为在父类中是没有的 。

3.多态：
是指一个程序中同名的方法共存的情况，调用者只需使用同一个方法名，系统会根据不同情况，调用相应的不同方法，从而实现不同的功能。多态性又被称为“**一个名字，多个方法**”。

面向对象的三个基本特征 和 五种设计原则

三个基本特征

面向对象的三个基本特征是：封装、继承、多态。



封装

封装最好理解了。封装是面向对象的特征之一，是对象和类概念的主要特性。

封装，也就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。

继承

面向对象编程（OOP）语言的一个主要功能就是“继承”。继承是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。

通过继承创建的新类称为“子类”或“派生类”。

被继承的类称为“基类”、“父类”或“超类”。

继承的过程，就是从一般到特殊的过程。

要实现继承，可以通过“继承”（Inheritance）和“组合”（Composition）来实现。

在某些 OOP 语言中，一个子类可以继承多个基类。但是一般情况下，一个子类只能有一个基类，要实现多重继承，可以通过多级继承来实现。

继承概念的实现方式有三类：实现继承、接口继承和可视继承。

Ø 实现继承是指使用基类的属性和方法而无需额外编码的能力；

Ø 接口继承是指仅使用属性和方法的名称、但是子类必须提供实现的能力；

Ø 可视继承是指子窗体（类）使用基窗体（类）的外观和实现代码的能力。

在考虑使用继承时，有一点需要注意，那就是两个类之间的关系应该是“属于”关系。例如，Employee 是一个人，Manager 也是一个人，因此这两个类都可以继承 Person 类。但是 Leg 类却不能继承 Person 类，因为腿并不是一个人。

抽象类仅定义将由子类创建的一般属性和方法，创建抽象类时，请使用关键字 Interface 而不是 Class。

OO 开发范式大致为：划分对象→抽象类→将类组织成为层次化结构(继承和合成) →用类与实例进行设计和实现几个阶段。

多态

多态性（polymorphisn）是允许你将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。简单的说，就是一句话：**允许将子类类型的指针赋值给父类类型的指针**。
实现多态，有二种方式，覆盖，重载。

覆盖，是指子类重新定义父类的虚函数的做法。

重载，是指允许存在多个同名函数，而这些函数的参数表不同（或许参数个数不同，或许参数类型不同，或许两者都不同）。

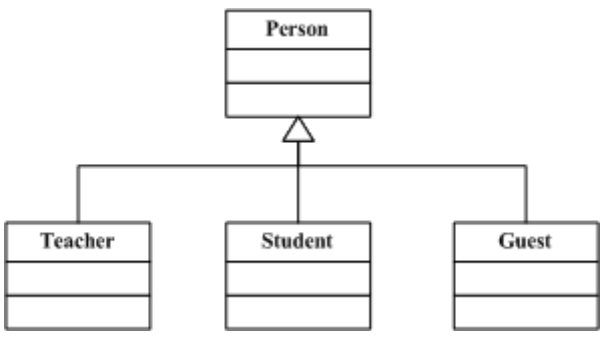
其实，重载的概念并不属于“面向对象编程”，重载的实现是：编译器根据函数不同的参数表，对同名函数的名称做修饰，然后这些同名函数就成了不同的函数（至少对于编译器来说是这样的）。如，有两个同名函数：function func(p:integer):integer;和 function func(p:string):integer;。那么编译器做过修饰后的函数名称可能是这样的：int_func、str_func。对于这两个函数的调用，在编译器间就已经确定了，是静态的（记住：是静态）。也就是说，它们的地址在编译期就绑定了（早绑定），因此，重载和多态无关！真正和多态相关的是“覆盖”。当子类重新定义了父类的虚函数后，父类指针根据赋给它的不同的子类指针，动态（记住：是动态！）的调用属于子类的该函数，这样的函数调用在编译期间是无法确定的（调用的子类的虚函数的地址无法给出）。因此，这样的函数地址是在运行期绑定的（晚邦定）。结论就是：重载只是一种语言特性，与多态无关，与面向对象也无关！引用一句 Bruce Eckel 的话：“不要犯傻，如果它不是晚邦定，它就不是多态。”

那么，多态的作用是什么呢？

我们知道，封装可以隐藏实现细节，使得代码模块化；继承可以扩展已存在的代码模块（类）；它们的目的都是为了——代码重用。而多态则是为了实现另一个目的——接口重用！多态的作用，就是为了类在继承和派生的时候，保证使用“家谱”中任一类的实例的某一属性时的正确调用。

概念讲解

泛化（Generalization）



图表 1 泛化

在上图中，空心的三角表示继承关系（类继承），在 UML 的术语中，这种关系被称为泛化（Generalization）。Person(人)是基类，Teacher(教师)、Student(学生)、Guest(来宾)是子类。

若在逻辑上 B 是 A 的“一种”，并且 A 的所有功能和属性对 B 而言都有意义，则允许 B 继承 A 的功能和属性。

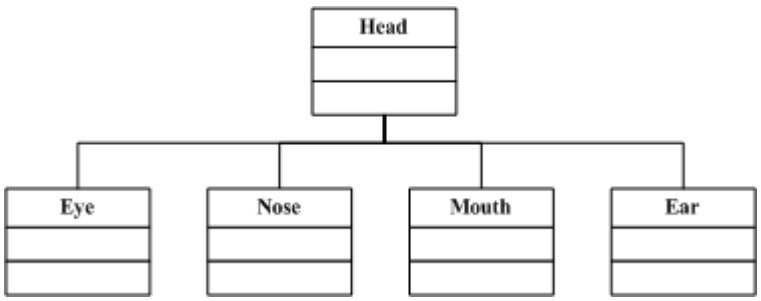
例如，教师是人，Teacher 是 Person 的“一种”（a kind of ）。那么类 Teacher 可以从类 Person 派生（继承）。

如果 A 是基类，B 是 A 的派生类，那么 B 将继承 A 的数据和函数。

如果类 A 和类 B 毫不相关，不可以为了使 B 的功能更多些而让 B 继承 A 的功能和属性。

若在逻辑上 B 是 A 的“一种”（a kind of ），则允许 B 继承 A 的功能和属性。

聚合（组合）



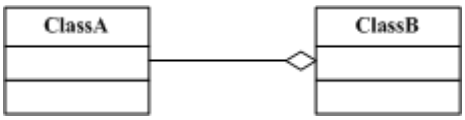
图表 2 组合

若在逻辑上 A 是 B 的“一部分”（a part of），则不允许 B 从 A 派生，而是要用 A 和其它东西组合出 B。

例如，眼（Eye）、鼻（Nose）、口（Mouth）、耳（Ear）是头（Head）的一部分，所以类 Head 应该由类 Eye、Nose、Mouth、Ear 组合而成，不是派生（继承）而成。

聚合的类型分为无、共享(聚合)、复合(组合)三类。

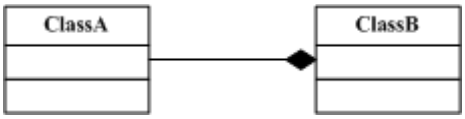
聚合（aggregation）



图表 3 共享

上面图中，有一个菱形（空心）表示聚合（aggregation）（聚合类型为共享），聚合的意义表示 has-a 关系。聚合是一种相对松散的关系，聚合类 B 不需要对被聚合的类 A 负责。

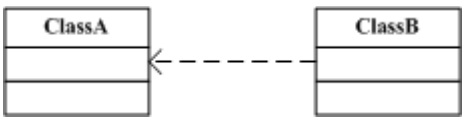
组合（composition）



图表 4 复合

这幅图与上面的唯一区别是菱形为实心的，它代表了一种更为坚固的关系——组合（composition）（聚合类型为复合）。组合表示的关系也是 has-a，不过在这里，A 的生命期受 B 控制。即 A 会随着 B 的创建而创建，随 B 的消亡而消亡。

依赖(Dependency)



图表 5 依赖

这里 B 与 A 的关系只是一种依赖(Dependency)关系，这种关系表明，如果类 A 被修改，那么类 B 会受到影响。

看法六

面向对象的三大基本特征：封装、继承和多态

一、封装

利用抽象数据类型将数据和基于数据的操作封装在一起，使其构成一个不可分割的独立实体。数据被保护在抽象数据类型的内部，尽可能地隐藏内部的细节，只保留一些对外接口使之与外部发生联系。用户无需知道对象内部的细节，但可以通过对象对外提供的接口来访问该对象。

优点：

减少耦合：可以独立地开发、测试、优化、使用、理解和修改

减轻维护的负担：可以更容易被程序员理解，并且在调试的时候可以不影响其他模块

有效地调节性能：可以通过剖析确定哪些模块影响了系统的性能

提高软件的可重用性

降低了构建大型系统的风险：即使整个系统不可用，但是这些独立的模块却有可能是可用的

二、继承

继承实现了 IS-A 关系，例如 Cat 和 Animal 就是一种 IS-A 关系，因此 Cat 可以继承自 Animal，从而获得 Animal 非 private 的属性和方法。

Cat 可以当做 Animal 来使用，也就是说可以使用 Animal 引用 Cat 对象。父类引用指向子类对象称为 向上转型 。

```
Animal animal = new Cat();
```

继承应该遵循里氏替换原则，子类对象必须能够替换掉所有父类对象。

三、多态

多态分为编译时多态和运行时多态。编译时多态主要指方法的重载，运行时多态指程序中定义的对象引用所指向的具体类型在运行期间才确定。

运行时多态有三个条件:

- 继承
- 覆盖（重写）

•向上转型

面向对象的设计原则

程序设计六大原则

看法一

1.单一职责

简单来说单一职责就是一个类只负责一个功能。更加具体的说就是对一个类而言，应该是一组相关性很高的函数、数据的封装，是高内聚低耦合的，对外界而言应该仅有一个引起它变化的原因。

单一职责在项目中的使用:

1.项目中的新手引导变量的管理可以统一在各自的Modle中用单独的类来管理

2.MVP模式P层生命周期与V层生命周期的同步可以用单独的包装类来实现，

3.各种基础框架功能的定义，例如：图片的加载、缓存、显示等都应该在各自的类中去做。

2.开闭原则

开闭原则的英文全称是Open Close Principle缩写即OCP。开闭原则的定义是：软件中的对象(类、模块、函数等)应该对于扩展是开放的，但是对于修改是封闭的。在软件的生命周期内，因为变化、升级和维护等原因需要对软件的原有代码进行修改时，可能会将错误的代码引入，从而破坏原有系统。因此当软件需求发生变化时，我们应该尽量通过扩展的方式 来实现变化，而不是通过修改已有的代码。

开闭原则在项目中的使用:

1.基类与子类，子类可以继承父类并扩展父类的功能

2.接口与实现类，接口定义功能，实现类按照各自的需求实现

3.里氏替换原则

里氏替换原则的定义:如果对每一个类型为 S 的对象 01，都有类型为 T 的对象 02，程序 P 在所有的对象 01 都带换成 02 时，程序 P 的行为没有发生变化，那么类型 S 是类型 T 的子类型换言之就是所有引用基类的地方必须能透明的使用其子类的对象。更通俗的讲就是只要父类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的错误或者异常。

里氏替换原则的核心是抽象，而抽象又依赖于继承这个特性，在 OOP 当中，继承的优缺点都相当明显。

优点：

1.代码重用，减少创建类的成本，每个子类都拥有父类的方法和属性

2.子类与父类基本相似，但又与父类有所区别

3.提高代码的可扩展性

缺点:

1.继承是侵入性的，只要继承就必须拥有父类的方法和属性

2.可能造成子类代码冗余，灵活性降低，因为子类必须拥有父类的属性和方法

在上面的例子中，我们通过 ImageCache 建立起了一套缓存的规范，在通过 setImageCache 注入不同的具体实现，保证了系统的扩展性和灵活性。因此开闭原则和里氏替换原则往往是生死相依，形影不离的，通过里氏替换原则来达到对扩展开放，对修改关闭的效果。

4.依赖倒置原则

依赖倒置原则指定了一种特定的解耦形式，使得高层次的模块不依赖与低层次模块的实现细节的目的，依赖模块被颠倒了。依赖倒置原则有以下几个关键点:

1.高层模块不应该依赖于低层模块，两者都应该依赖其抽象

2.抽象不应该依赖于细节

3.细节应该依赖于抽象

在 Java 语言中，抽象就是指接口或者抽象类，二者都是不能够被直接实例化的：细节就是实现类，实现接口或者抽象类而产生的类就是细节，其特点就是可以直接被实例化，也就是可以使用关键字 new 产生一个对象。高层模块就是指调用端，底层模块就是指具体的实现类。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产生，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者抽象类产生的。使用一句话概括就是:面向接口编程或者说是面向抽象编程。

5.接口隔离原则

接口隔离原则的定义是:客户端不应该依赖于他不需要的接口。另一种定义是:类之间的依赖关系应该建立在最小的接口上。接口隔离原则将非常庞大，臃肿的接口拆分成更小的和更具体的接口，这样客户端将会值需要知道它们感兴趣的方法。接口隔离原则的目的是系统解开耦合，从而容易重构、更改和部署。

6.迪米特原则

迪米特原则：一个对象应该对其他对象有最少的了解，通俗的讲，一个类应该对自己需要耦合或调用的类知道的最少，类的内部如何实现与调用者或者依赖者没有关系，调用者或者依赖者只需要知道他需要的方法即可，其他的一概不管。类与类之间的关系越密切，耦合度越大，当一个类发生改变时，对另一个类的影响也越大。

小结:

在应用开发过程中，最难的不是完成应用的开发工作，而是在后续的升级、维护过程中让应用系统能够拥抱变化。拥抱变化也意味着在满足需求而且不破坏系统稳定的前提下保持高可扩展性、高内聚、低耦合，在经历了各版本的变更之后依然保持着清晰、灵活、稳定的系统架构。

看法二

单一职责原则是最简单的面向对象设计原则，它用于控制类的粒度大小。单一职责原则定义如下:

> 单一职责原则(Single Responsibility Principle, SRP)：一个类只负责一个功能领域中的相应职责，或者可以定义为：就一个类而言，应该只有一个引起它变化的原因。

单一职责原则告诉我们：一个类不能太“累”！在软件系统中，一个类（大到模块，小到方法）承担的职责越多，它被复用的可能性就越小，而且一个类承担的职责过多，

就相当于将这些职责耦合在一起，当其中一个职责变化时，可能会影响其他职责的运作，因此要将这些职责进行分离，将不同的职责封装在不同的类中，即将不同的变化原因封装在不同的类中，如果多个职责总是同时发生改变则可将它们封装在同一类中。

单一职责原则是实现高内聚、低耦合的指导方针，它是最简单但又最难运用的原则，需要设计人员发现类的不同职责并将其分离，而发现类的多重职责需要设计人员具有较强的分析设计能力和相关实践经验。

二.开闭原则

原文链接：http://blog.csdn.net/lovelion/article/details/7537584

开闭原则是面向对象的可复用设计的第一块基石，它是最重要的面向对象设计原则。开闭原则由Bertrand Meyer 于 1988 年提出，其定义如下：

> 开闭原则(Open-Closed Principle, OCP)：一个软件实体应当对扩展开放，对修改关闭。即软件实体应尽量在不修改原有代码的情况下进行扩展。

在开闭原则的定义中，软件实体可以指一个软件模块、一个由多个类组成的局部结构或一个独立的类。

任何软件都需要面临一个很重要的问题，即它们的需求会随时间的推移而发生变化。当软件系统需要面对新的需求时，我们应该尽量保证系统的设计框架是稳定的。如果一个软件设计符合开闭原则，那么可以非常方便地对系统进行扩展，而且在扩展时无须修改现有代码，使得软件系统在拥有适应性和灵活性的同时具备较好的稳定性和延续性。随着软件规模越来越大，软件寿命越来越长，软件维护成本越来越高，设计满足开闭原则的软件系统也变得越来越重要。

为了满足开闭原则，需要对系统进行抽象化设计，抽象化是开闭原则的关键。在Java、C#等编程语言中，可以为系统定义一个相对稳定的抽象层，而将不同的实现行为移至具体的实现层中完成。在很多面向对象编程语言中都提供了接口、抽象类等机制，可以通过它们定义系统的抽象层，再通过具体类来进行扩展。如果需要修改系统的行为，无须对抽象层进行任何改动，只需要增加新的具体类来实现新的业务功能即可，实现在不修改已有代码的基础上扩展系统的功能，达到开闭原则的要求。

里氏代换原则由 2008 年图灵奖得主、美国第一位计算机科学女博士 Barbara Liskov 教授和卡内基·梅隆大学 Jeannette Wing 教授于 1994 年提出。其严格表述如下：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 代换 o2 时，程序 P 的行为没有变化，那么类型 S 是类型 T 的子类型。这个定义比较拗口且难以理解，因此我们一般使用它的另一个通俗版定义：

> 里氏代换原则(Liskov Substitution Principle, LSP)：所有引用基类（父类）的地方必须能透明地使用其子类的对象。

里氏代换原则告诉我们，在软件中将一个基类对象替换成它的子类对象，程序将不会产生任何错误和异常，反过来则不成立，如果一个软件实体使用的是一个子类对象的话，那么它不一定能够使用基类对象。例如：我喜欢动物，那我一定喜欢狗，因为狗是动物的子类；但是我喜欢狗，不能据此断定我喜欢动物，因为我并不喜欢老鼠，虽然它也是动物。

例如有两个类，一个类为 BaseClass，另一个是 SubClass 类，并且 SubClass 类是 BaseClass 类的子类，那么一个方法如果可以接受一个 BaseClass 类型的基类对象 base 的话，如：method1(base)，那么它必然可以接受一个 BaseClass 类型的子类对象

sub，method1(sub)能够正常运行。反过来的代换不成立，如一个方法 method2 接受 BaseClass 类型的子类对象 sub 为参数：method2(sub)，那么一般而言不可以有 method2(base)，除非是重载方法。

里氏代换原则是实现开闭原则的重要方式之一，由于使用基类对象的地方都可以使用子类对象，因此在程序中尽量使用基类类型来对对象进行定义，而在运行时再确定其子类类型，用子类对象来替换父类对象。

在使用里氏代换原则时需要注意如下几个问题：

(1)子类的所有方法必须在父类中声明，或子类必须实现父类中声明的所有方法。根据里氏代换原则，为了保证系统的扩展性，在程序中通常使用父类来进行定义，如果一个方法只存在子类中，在父类中不提供相应的声明，则无法在以父类定义的对象中使用该方法。

(2) 我们在运用里氏代换原则时，尽量把父类设计为抽象类或者接口，让子类继承父类或实现父接口，并实现在父类中声明的方法，运行时，子类实例替换父类实例，我们可以很方便地扩展系统的功能，同时无须修改原有子类的代码，增加新的功能可以通过增加一个新的子类来实现。里氏代换原则是开闭原则的具体实现手段之一。

(3) Java 语言中，在编译阶段，Java 编译器会检查一个程序是否符合里氏代换原则，这是一个与实现无关的、纯语法意义上的检查，但 Java 编译器的检查是有局限的。

里氏代换原则是实现开闭原则的重要方式之一。在本实例中，在传递参数时使用基类对象，除此以外，在定义成员变量、定义局部变量、确定方法返回类型时都可使用里氏代换原则。针对基类编程，在程序运行时再确定具体子类。

四.依赖倒置原则

如果说开闭原则是面向对象设计的目标的话，那么依赖倒转原则就是面向对象设计的主要实现机制之一，它是系统抽象化的具体实现。依赖倒转原则是 Robert C. Martin 在 1996 年为“C++Reporter”所写的专栏 Engineering Notebook 的第三篇，后来加入到他在 2002 年出版的经典著作“Agile Software Development, Principles, Patterns, and Practices”一书中。依赖倒转原则定义如下：

> 依赖倒转原则(Dependency Inversion Principle, DIP)：抽象不应该依赖于细节，细节应当依赖于抽象。换言之，要针对接口编程，而不是针对实现编程。

依赖倒转原则要求我们在程序代码中传递参数时或在关联关系中，尽量引用层次高的抽象层类，即使用接口和抽象类进行变量类型声明、参数类型声明、方法返回类型声明，以及数据类型的转换等，而不要用具体类来做这些事情。为了确保该原则的应用，一个具体类应当只实现接口或抽象类中声明过的方法，而不要给出多余的方法，否则将无法调用到在子类中增加的新方法。

在引入抽象层后，系统将具有很好的灵活性，在程序中尽量使用抽象层进行编程，而将具体类写在配置文件中，这样一来，如果系统行为发生变化，只需要对抽象层进行扩展，并修改配置文件，而无须修改原有系统的源代码，在不修改的情况下来扩展系统的功能，满足开闭原则的要求。

在实现依赖倒转原则时，我们需要针对抽象层编程，而将具体类的对象通过依赖注入(DependencyInjection, DI)的方式注入到其他对象中，依赖注入是指当一个对象要与其他对象发生依赖关系时，通过抽象来注入所依赖的对象。常用的注入方式有三种，分别是：构造注入，设值注入（Setter 注入）和接口注入。构造注入是指通过构造函数来传入具体

类的对象，设值注入是指通过 Setter 方法来传入具体类的对象，而接口注入是指通过在接口中声明的业务方法来传入具体类的对象。这些方法在定义时使用的是抽象类型，在运行时再传入具体类型的对象，由子类对象来覆盖父类对象。

由于 CustomerDAO 针对具体数据转换类编程，因此在增加新的数据转换类或者更换数据转换类时都不得不修改 CustomerDAO 的源代码。我们可以通过引入抽象数据转换类解决该问题，在引入抽象数据转换类 DataConvertor 之后，CustomerDAO 针对抽象类 DataConvertor 编程，而将具体数据转换类名存储在配置文件中，符合依赖倒转原则。根据里氏代换原则，程序运行时，具体数据转换类对象将替换 DataConvertor 类型的对象，程序不会出现任何问题。更换具体数据转换类时无须修改源代码，只需要修改配置文件；如果需要增加新的具体数据转换类，只要将新增数据转换类作为 DataConvertor 的子类并修改配置文件即可，原有代码无须做任何修改，满足开闭原则。

开闭原则、里氏代换原则和依赖倒转原则，在大多数情况下，这三个设计原则会同时出现，开闭原则是目标，里氏代换原则是基础，依赖倒转原则是手段，它们相辅相成，相互补充，目标一致，只是分析问题时所站角度不同而已。

五.接口隔离原则

接口隔离原则定义如下：

> 接口隔离原则(Interface Segregation Principle, ISP)：使用多个专门的接口，而不使用单一的总接口，即客户端不应该依赖那些它不需要的接口。

根据接口隔离原则，当一个接口太大时，我们需要将它分割成一些更细小的接口，使用该接口的客户端仅需知道与之相关的方法即可。每一个接口应该承担一种相对独立的角色，不干不该干的事，该干的事都要干。这里的“接口”往往有两种不同的含义：一种

是指一个类型所具有的方法特征的集合，仅仅是一种逻辑上的抽象；另外一种是指某种语言具体的“接口”定义，有严格的定义和结构，比如 Java 语言中的 interface。对于这两种不同的含义，ISP 的表达方式以及含义都有所不同：

(1) 当把“接口”理解成一个类型所提供的所有方法特征的集合的时候，这就是一种逻辑上的概念，接口的划分将直接带来类型的划分。可以把接口理解成角色，一个接口只能代表一个角色，每个角色都有它特定的一个接口，此时，这个原则可以叫做“角色隔离原则”。

(2) 如果把“接口”理解成狭义的特定语言的接口，那么 ISP 表达的意思是指接口仅仅提供客户端需要的行为，客户端不需要的行为则隐藏起来，应当为客户端提供尽可能小的单独的接口，而不要提供大的总接口。在面向对象编程语言中，实现一个接口就需要实现该接口中定义的所有方法，因此大的总接口使用起来不一定很方便，为了使接口的职责单一，需要将大接口中的方法根据其职责不同分别放在不同的小接口中，以确保每个接口使用起来都较为方便，并都承担某一单一角色。接口应该尽量细化，同时接口中的方法应该尽量少，每个接口中只包含一个客户端（如子模块或业务逻辑类）所需的方法即可，这种机制也称为“定制服务”，即为不同的客户端提供宽窄不同的接口。

在使用接口隔离原则时，我们需要注意控制接口的粒度，接口不能太小，如果太小会导致系统中接口泛滥，不利于维护；接口也不能太大，太大的接口将违背接口隔离原则，灵活性较差，使用起来很不方便。一般而言，接口中仅包含为某一类用户定制的方法即可，不应该强迫客户依赖于那些它们不用的方法。

六.迪米特法则

迪米特法则来自于 1987 年美国东北大学(Northeastern University)一个名为“Demeter”的研究项目。迪米特法则又称为最少知识原则(LeastKnowledge Principle, LKP)，其定义如下：

> 迪米特法则(Law of Demeter, LoD)：一个软件实体应当尽可能少地与其他实体发生相互作用。

如果一个系统符合迪米特法则，那么当其中某一个模块发生修改时，就会尽量少地影响其他模块，扩展会相对容易，这是对软件实体之间通信的限制，迪米特法则要求限制软件实体之间通信的宽度和深度。迪米特法则可降低系统的耦合度，使类与类之间保持松散的耦合关系。

迪米特法则还有几种定义形式，包括：不要和“陌生人”说话、只与你的直接朋友通信等，在迪米特法则中，对于一个对象，其朋友包括以下几类：

(1) 当前对象本身(this)；

(2) 以参数形式传入到当前对象方法中的对象；

(3) 当前对象的成员对象；

(4) 如果当前对象的成员对象是一个集合，那么集合中的元素也都是朋友；

(5) 当前对象所创建的对象。

任何一个对象，如果满足上面的条件之一，就是当前对象的“朋友”，否则就是“陌生人”。在应用迪米特法则时，一个对象只能与直接朋友发生交互，不要与“陌生人”发生直接交互，这样做可以降低系统的耦合度，一个对象的改变不会给太多其他对象带来影响。

迪米特法则要求我们在设计系统时，应该尽量减少对象之间的交互，如果两个对象之间不必彼此直接通信，那么这两个对象就不应当发生任何直接的相互作用，如果其中的

一个对象需要调用另一个对象的某一个方法的话，可以通过第三者转发这个调用。简言之，就是通过引入一个合理的第三者来降低现有对象之间的耦合度。

在将迪米特法则运用到系统设计中时，要注意下面的几点：在类的划分上，应当尽量创建松耦合的类，类之间的耦合度越低，就越有利于复用，一个处在松耦合中的类一旦被修改，不会对关联的类造成太大波及；在类的结构设计上，每一个类都应当尽量降低其成员变量和成员函数的访问权限；在类的设计上，只要有可能，一个类型应当设计成不变类；在对其他类的引用上，一个对象对其他对象的引用应当降到最低。

通过引入一个专门用于控制界面控件交互的中间类(Mediator)来降低界面控件之间的耦合度。引入中间类之后，界面控件之间不再发生直接引用，而是将请求先转发给中间类，再由中间类来完成对其他控件的调用。当需要增加或删除新的控件时，只需修改中间类即可，无须修改新增控件或已有控件的源代码.

看法三

单一职责原则

一个类应该仅有一个引起它变化的原因

如果一个类承担的职责过多，那么这些职责就会相互依赖，一个职责的变化可能会影响另一个职责的履行。

比如 andoird 里的 Canvas 负责图形布局，Paint 负责绘制风格。如果合为一个类，比如传入颜色，坐标，形状来绘制图形，当添加透明度等功能时，所有绘制图形的方法都需要修改，而分开之后只需修改 Paint 类。

开放封闭原则

对扩展开放，对更改封闭。

为一个软件系统增加新功能时，只需要从原来的系统派生出一些新类就可以，不需要修改原来的任何一行代码。

里氏替换原则

子类必须能够替换任何地方的父类

比如玩具枪继承枪，但在战争中无法代替枪用于攻击

合成/聚合原则

尽量使用合成/聚合而不使用类继承

比如把”总监”，”经理”，”学生”当成”人”的子类。错误在于把”职位”和”人”的等级结构混淆了。一个人可以同时拥有多个职位。如果按继承来设计，一个人只能有一个职位，而且无法再改变。

正确的是建立抽象类”职位”，”总监”，”经理”，”学生”是”职位”的子类，”人”可以拥有多个”职位”。

迪米特法则（最少知道原则）

一个对象应当对其他对象有尽可能少的了解

比如你去买一辆车，给钱就行，不需要知道车是怎么生产的。

依赖倒置原则

让高层模块不直接依赖低层模块

若高层模块A直接依赖低层模块B，假如要将A改为依赖C，则必须修改A的代码。而将A修改为依赖接口I，更换底层模块时只需实现接口I

接口隔离原则

一个接口只应该描述一种能力

例如，琴棋书画应该设计为四个接口，而不是一个接口中的四个方法，因为琴棋书画四样都精通的人还是少数，如果按能力分开，会几个就实现几个接口，会使每个接口更好地被复用.

看法四

五种设计原则

"面向对象设计五大原则"和良性依赖原则在应付变化方面的作用。

单一职责原则(Single-Resposibility Principle)。"对一个类而言，应该仅有一个引起它变化的原因。"本原则是我们非常熟悉地"高内聚性原则"的引申，但是通过将"职责"极具创意地定义为"变化的原因"，使得本原则极具操作性，尽显大师风范。同时，本原则还揭示了内聚性和耦合生，基本途径就是提高内聚性；如果一个类承担的职责过多，那么这些职责就会相互依赖，一个职责的变化可能会影响另一个职责的履行。其实OOD的实质，就是合理地进行类的职责分配。

开放封闭原则(Open-Closed principle)。"软件实体应该是可以扩展的，但是不可修改。"本原则紧紧围绕变化展开，变化来临时，如果不必改动软件实体裁的源代码，就能扩充它的行为，那么这个软件实体设计就是满足开放封闭原则的。如果说我们预测到某种变化，或者某种变化发生了，我们应当创建抽象类来隔离以后发生的同类变化。在Java中，这种抽象是

指抽象基类或接口；在C++中，这各抽象是指抽象基类或纯抽象基类。当然，没有对所有情况都贴切的模型，我们必须对软件实体应该面对的变化做出选择。

Liskov替换原则(Liskov-Substituion Principle)。"子类型必须能够替换掉它们的基类型。"本原则和开放封闭原则关系密切，正是子类型的可替换性，才使得使用基类型模块无需修改就可扩充。Liskov替换原则从基于契约的设计演化而来，契约通过为每个方法声明"先验条件"和"后验条件"；定义子类时，必须遵守这些"先验条件"和"后验条件"。当前基于契的设计发展势头正劲，对实现"软件工厂"的"组装生产"梦想是一个有力的支持。

依赖倒置原则(Dependecy-Inversion Principle)。"抽象不应依赖于细节，细节应该依赖于抽象。"本原则几乎就是软件设计的正本清源之道。因为人解决问题的思考过程是先抽象后具体，从笼统到细节，所以我们先生产出的势必是抽象程度比较高的实体，而后才是更加细节化的实体。于是，"细节依赖于抽象"就意味着后来的依赖于先前的，这是自然而然的重用之道。而且，抽象的实体代表着笼而统之的认识，人们总是比较容易正确认识它们，而且本身也是不易变的，依赖于它们是安全的。依赖倒置原则适应了人类认识过程的规律，是面向对象设计的标志所在。

接口隔离原则(Interface-Segregation Principle)。"多个专用接口优于一个单一的通用接口。"本原则是单一职责原则用于接口设计的自然结果。一个接口应该保证，实现该接口的实例对象可以只呈现为单一的角色；这样，当某个客户程序的要求发生变化，而迫使接口发生改变时，影响到其他客户程序的可能生性小。

良性依赖原则。"不会在实际中造成危害的依赖关系，都是良性依赖。"通过分析不难发现，本原则的核心思想是"务实"，很好地揭示了极限编程(Extreme Programming)中"简单设计"各"重构"的理论基础。本原则可以帮助我们抵御"面向对象设计五大原则"以及设计模式的诱惑，以免陷入过度设计(Over-engineering)的尴尬境地，带来不必要的复杂性。

看法五

一、单一职责原则

单一职责原则准确的解释就是，就一个类而言，应该仅有一个引起它变化的原因。我们在编程的时候，很自然的就会给一个类加各种各样的功能，比如我们写一个窗体应用程序，一般都会生成一个Form1这样的类，

于是我们就把各种各样的代码，就像某种商业运算的算法呀，像数据库访问的SQL语句呀什么的都写到这样的类当中，这就意味着，无论任何需要要来，你都需要更改这个窗体类，这其实是很糟糕的，维护麻烦，复用不可能，也缺乏灵活性。

如果一个类承担的职责过多，就等于把这些职责耦合在一起，一个职责的变化可能会削弱或者抑制这个类完成其它职责的能力。这种耦合会导致脆弱的设计，设计会有意想不到的破坏。比如：设计游戏时应该把“界面”和“游戏逻辑”进行分离。

软件设计眞正要做的许多内容，就是发现职责并把那些职责相互分离。其实要去判断是否应该分离出类来，也不难，那就是如果你能够想到多于一个动机去改变一个类，那么这个类就具有多于一个的职责。那么就应该考虑类的职责分离。

二、开放封闭原则

开放-封闭原则就是说，软件实体（类、模板、函数等等）应该可以扩展，但是不可以修改。

对于扩展是开放的，对于更改是封闭的。

你设计的时候，时刻要考虑，尽量让这个类足够好，写好了就不要去修改了（因为任何需求的变更都是需要成本的），如果新需求来了，我们增加一些类就完事了，原来的代码能不动就不动。

无论模块多么“封闭”都会存在一些无法对之封闭的变化。既然不能完全封闭，设计人员必须对于设计的模块对哪些封闭做出选择。他必须猜出最有可能发生什么变化，然后构造抽象来隔离那些变化。

我们希望的是在开发展开不久就知道可能发生的变化，查明可能发生变化所等待的时间越长，要创建正确的抽象就越困难。所以变化发生时立刻采取行动。

三、里氏代换原则 依赖倒转原则

里氏代换原则：子类型必须能够替换掉它们的父类型。

也就是说一个软件实体如果使用一个父类的话，那么一定适用于其子类，而且觉察不出父类对象和子类对象的区别。就是说，用子类替换父类，程序的行为完全没有变化。

里氏代换原则，使得继承复用成为了可能，只有当子类可以替换掉父类，软件单位的功能不受影响时，父类才能眞正被复用，而子类也能够在父类的基础上增加新的行为。

里氏代换原则使开放-封闭成为了可能。使得模块在无需修改的情况下进行扩展。

B.抽象不应该依赖细节，细节应该依赖于抽象。

要针对接口编程，不要对实现编程。比如：电脑是由CPU、硬盘、显卡、光驱、内存组成，当其中有一个部分损坏时，只要换掉损坏部件即可。因为各个部件都是针脚式或触点式子的标准接口。这个增加了部件复用的可能。在这个例子中，“针脚”就相当于程序的“接口”，各个部件就相当于不同的“软件实体”，这样只要依赖接口，程序就可以无限的扩展。

看法六

1.优化代码的第一步---------单一职责原则(Single Responsibility Principle - SRP)

就一个类而言，应该仅有一个引起它变化的原因。简单来说，一个类中应该是一组相关性很高的函数、数据的封装。单一职责的划分界限并不是总那么清晰，很多时候都是需要靠个人经验来界定。当然，最大的问题就是对职责的定义，什么是类的职责，以及怎么划分类的职责。

2.让程序更稳定、更灵活-------开闭原则(Open Close Principle - OCP)

3.构建扩展性更好的系统-----里氏替换原则(Liskov Subsitution Principle - LSP)

面向对象的言语的三大特点是继承、封装、多态，里氏替换原则就是依赖于继承、多态这两大特性。里氏替换原则简单来说就是，所有引用基类的地方必须能透明地使用其子类的对象。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。说了那么多，其实最终总结就两个字：抽象。

里氏替换原则的核心原理是抽象，抽象又依赖于继承这个特性，在OOP当中，继承的优缺点都相当明显。优点有以下几点：

●代码重用，减少创建类的成本，每隔子类都拥有父类的函数和属性；
●子类于父类基本相似，但又与父类有所却别；

●提高代码的可扩展性。

继承的缺点：

●继承是侵入性的，只要继承就必须拥有父类所有属性和函数；
●可能造成子类代码冗余、灵活性降低，因为子类必须拥有父类的属性和函数。

事物总是具有两面性，如何权衡利与弊都是需要根据具体情况来做出选择并加以处理。我们还是接着上面的 ImageLoader 来做说明。
ImageLoader 很好的反应了里氏替换原则，即
MemoryCache、DiskCache、DoubleCache 都可以替换 ImageCache 的工作，并且能够保证行为的正确性。ImageCache 建立了获取缓存图片、保存缓存图片的接口规范，MemoryCache 等根据接口规范实现了相应的功能，用户只需要在使用时指定据图的缓存对象就可以动态的替换 ImageLoader 中的缓存策略。这就使得 ImageLoader 的缓存系统具有了无限的可能性，也就是保证了可扩展性。

4.让项目拥有变化的能力----依赖倒置原则(Dependence Inversion Principle - DIP)

依赖倒置原则指代了一种特定的解耦形式，使得高层次的模块不依赖于低层次的模块的实现细节的目的，依赖模块被颠倒了。

依赖倒置原则有以下几个关键点：

●高层模块不应该依赖低层模块，两者都应该依赖其抽象；
●抽象不应该依赖于细节；
●细节应该依赖于抽象。

这里说的高层模块就是调用端，低层模块就是具体实现类。其实一句话就可以概括：面向接口编程、或者说是面向抽象编程。

5.系统有更高的灵活性----接口隔离原则(Interface Segregation Principles - ISP)

类间的依赖关系应该建立在最小的接口上。

接口隔离原则将非常庞大、臃肿的接口拆分成更小的和更具体的接口，目的是系统解开耦合，从而容易重构、更改和重新部署。

使得每个抽象的职责更加单一，更细致。

6.更好的可扩展性-----迪米特原则(Law Of Demeter - LOD)

也称最少知道原则。通俗的将，一个类应该对自己需要耦合或调用的类知道得最少，类的内部如何实现与调用者或者依赖者没关系，调用者或者依赖者只需要知道它需要的方法即可，其他的可一概不管。类于类之间的关系越密切，耦合度越大，当一个类发生改变时，对另一个类的影响也越大。