# 基于面向对象浅谈

1903杜怡围

前言：在整理答辩内容的过程，我是以一种整理资料，补充盲点的心态去找资料。收益良多。不仅让自己对于面向对象概念清晰了一些。收益最大的是明白了组合复用规则的具体含义，彻底读懂天龙八部中该思想的灵活应用。对于三大特征，六大原则也有了更深的理解。本文结构上，仔细想想，借用了老师面向对象文档基础上进行扩展，不就是运用了面向对象的思想吗？继承？在六大原则中我添加了sunny关于CRM系统中遇到问题的处理方式，分别附上各个项目实例在各规则中的具体应用。

Python从设计之初就已经是一门面向对象、解释型计算机程序设计的语言，正因为如此，在Python中创建一个类和对象是很容易的。

函数、模块、数字、字符串都是对象，在 Python中一切皆对象。

完全支持继承、重载、多重继承。

支持重载运算符，也支持泛型设计。

# 定义（未明确定义）

程序 = 对象 + 交互

面向对象的三大特点（封装，继承，多态）缺一不可。

面向对象(Object Oriented)是当前计算机界关心的重点，它是90年代软件开发方法的主流。面向对象的概念和应用已超越了程序设计和软件开发，扩展到很宽的范围。如数据库系统、交互式界面、应用结构、应用平台、分布式系统、网络管理结构、CAD技术、人工智能等领域。

谈到面向对象，这方面的文章非常多。但是，明确地给出对象的定义或说明对象的定义的非常少——至少我还没有发现。起初，“面向对象”是专指在程序设计中采用封装、继承、多态和抽象等设计方法。可是，这个定义显然不能再适合现在情况。面向对象的思想已经涉及到软件开发的各个方面。如，面向对象的分析（OOA，Object Oriented Analysis），面向对象的设计（OOD，Object Oriented Design）、以及我们经常说的面向对象的编程实现（OOP，Object Oriented Programming）。许多有关面向对象的文章都只是讲述在面向对象的开发中所需要注意的问题或所采用的比较好的设计方法。

面向对象的最主要目的不能仅仅只是提高程序的重复使用性。

# 面向对象思想

从世界观的角度可以认为：面向对象的基本哲学是认为世界是由各种各样具有自己的运动规律和内部状态的对象所组成的；不同对象之间的相互作用和通讯构成了完整的现实世界。因此，人们应当按照现实世界这个本来面貌来理解世界，直接通过对象及其相互关系来反映世界。这样建立起来的系统才能符合现实世界的本来面目。

从方法学的角度可以认为：面向对象的方法是面向对象的世界观在开发方法中的直接运用。它强调系统的结构应该直接与现实世界的结构相对应，应该围绕现实世界中的对象来构造系统，而不是围绕功能来构造系统。

# 面向对象的分析

面向对象分析的目的是对客观世界的系统进行建模。

系统分析的第一步是：陈述需求。分析者必须同用户一块工作来提炼需求，因为这样才表示了用户的真实意图，其中涉及对需求的分析及查找丢失的信息。

简谈：

我的理解是，在大多数情况下，人类是一种感官动物，最初都是以感官作为切入点，眼睛看到的东西就是真实的对象，不能脱离现实。

从现实世界中客观存在的事物（即对象）出发来构造软件系统，并在系统构造中尽可能运用人类的自然思维方式，强调直接以问题域（现实世界）中的事物为中心来思考问题，认识问题，并根据这些事物的本质特点，把它们抽象地表示为系统中的对象，作为系统的基本构成单位（而不是用一些与现实世界中的事物相关比较远，并且没有对应关系的其它概念来构造系统）。这可以使系统直接地映射问题域，保持问题域中事物及其相互关系的本来面貌。

# 优点

采用面向对象方法开发软件的基本目的和主要优点是通过重用提高软件的生产率。因此，应该优先选用能够最完善、最准确地表达问题域语义的面向对象语言。

优点：(1)提高重用性。

(2)提高可扩充性。

(3)提高健壮性。

# 基本概念

## (1)对象

对象是人们要进行研究的任何事物，从最简单的整数到复杂的飞机等均可看作对象，它不仅能表示具体的事物，还能表示抽象的规则、计划或事件。

## (2)对象的状态和行为

对象具有状态，一个对象用数据值来描述它的状态。

对象还有操作，用于改变对象的状态，操作就是对象的行为。

对象实现了数据和操作的结合，使数据和操作封装于对象的统一体中

## (3)类

具有相同或相似性质的对象的抽象就是类。因此，对象的抽象是类，类的具体化就是对象，也可以说类的实例是对象。

类具有属性，它是对象的状态的抽象，用数据结构来描述类的属性。

类具有操作，它是对象的行为的抽象，用操作名和实现该操作的方法来描述。

## (4)类的结构

在客观世界中有若干类，这些类之间有一定的结构关系。通常有两种主要的结构关系，即一般--具体结构关系，整体--部分结构关系。

（1）一般——具体结构称为分类结构，也可以说是“或”关系，或者是“is a”关系，泛化。

（2）整体——部分结构称为组装结构，它们之间的关系是一种“与”关系，或者是“has a”关系，关联。

（3）作为参数，依赖关系。

## (5)消息和方法

对象之间进行通信的结构叫做消息。在对象的操作中，当一个消息发送给某个对象时，消息包含接收对象去执行某种操作的信息。发送一条消息至少要包括说明接受消息的对象名、发送给该对象的消息名（即对象名、方法名）。一般还要对参数加以说明，参数可以是认识该消息的对象所知道的变量名，或者是所有对象都知道的全局变量名。

# 创建对象（实例化）

根据类创建对象称之为实例化。

方法：类中定义的函数。

对象：通过类定义的数据结构实例。

对象包括两个数据成员（类变量和实例变量）和方法

变量 = 构造函数 (参数列表)

# 类与对象

类是创建实例的模板，而实例则是一个一个具体的对象，各个实例拥有的数据都互相独立，互不影响；方法就是与实例绑定的函数，和普通函数不同，方法可以直接访问实例的数据；通过实例调用方法，我们就直接操作了对象内部的数据，但无需知道方法内部的实现细节。和静态语言不同，Python允许对实例变量绑定任何数据，也就是说，对于两个实例变量，虽然它们都是同一个类的不同实例，但拥有的变量名称都可能不同。

# 封装

## 定义

隐藏对象的属性和实现细节，仅对外提供公共访问方式。

1. 将不需要对外提供的内容都隐藏起来；

2. 把属性都隐藏，提供公共方法对其访问。

## 分析

1. 数据角度讲，将一些基本数据类型复合成一个自定义类型。

2. 行为角度讲，向类外提供功能，隐藏实现的细节。

3. 设计角度讲：

（1）分而治之

-- 将一个大的需求分解为许多类，每个类处理一个独立的功能。

-- 拆分好处：便于分工，便于复用，可扩展性强。

(2) 封装变化

-- 变化的地方独立封装，避免影响其他类。

(3) 高 内 聚

-- 类中各个方法都在完成一项任务(单一职责的类)。

(4) 低 耦 合

-- 类与类的关联性与依赖度要低(每个类独立)，让一个类的改变，尽少影响其他类。

## 作用

1. 简化编程，使用者不必了解具体的实现细节，只需要调用对外提供的功能。

2. 松散耦合，降低了程序各部分之间的依赖性。

3. 数据和操作相关联，方法操作的是自己的数据。

# 继承

## 定义

重用现有类的功能与概念，并在此基础上进行扩展。

## 作用

隔离客户端代码与功能的实现方式。

## 适用性

多个类在概念上是一致的，且需要进行统一的处理。

# 特点

1. 在继承中基类的构造（\_\_init\_\_()方法）不会被自动调用，它需要在其派生类的构造中亲自专门调用。

2. 在调用基类的方法时，需要加上基类的类名前缀，且需要带上self参数变量。区别于在类中调用普通函数时并不需要带上self参数。

3. Python总是首先查找对应类型的方法，如果它不能在派生类中找到对应的方法，它才开始到基类中逐个查找，即先在本类中查找调用的方法，找不到才去基类中找。

# 优缺点

面向对象编程的优点是代码的复用，而继承就是实现代码复用的方法之一。继承完全可以理解成类之间的类型和子类型关系。

继承的优点：

如果需要定义几个类，而类与类之间有一些公共的属性和方法，这时就可以把相同的属性和方法作为基类的成员，而特殊的方法及属性则在本类中定义，这样只需要继承基类这个动作，就可以访问到基类的属性和方法了，它提高了代码的可扩展性。

继承的缺点：

继承的一个缺点就是，可能特殊的本类又有其他特殊的地方，又会定义一个类，其下也可能再定义类，这样就会造成继承的那条线越来越长，使用继承的话,任何一点小的变化也需要重新定义一个类,很容易引起类的爆炸式增长,产生一大堆有着细微不同的子类. 所以尽可能遵守“多用组合少用继承”的原则。

# 多态

## 定义

父类的同一种动作或者行为，在不同的子类上有不同的实现。

## 作用

1. 继承将相关概念的共性进行抽象，多态在共性的基础上，体现类型的个性化（一个行为有不同的实现）。

2. 增强程序扩展性，体现开闭原则。

## 重写

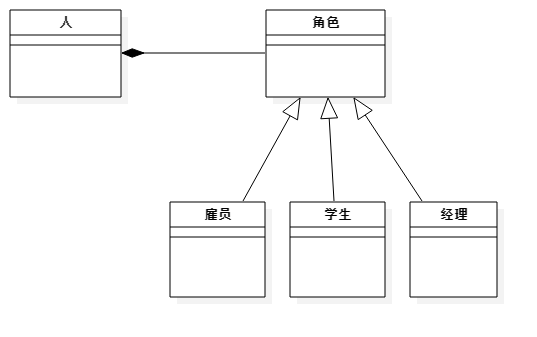
子类实现了父类中相同的方法（方法名、参数），在调用该方法时，实际调用的是子类的方法。

# 六大设计原则（心中的一把尺）

## 开闭原则、里氏代换原则和依赖倒转原则的关系

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。

## 使用组合还是继承



**注释：人被继承到雇员，学生，经理子类。而实际上，雇员、学生和经理分别描述一种角色，而人可以同时有几种不同的角色。比如，一个人既然是经理了就一定是雇员，使用继承来实现角色，则只能使用每一个人具有一种角色，这显然是不合理的。错误的原因就是把角色的等级结构和人的等级结构混淆起来，把Has-A的关系误认为是Is-A的关系，通过下面的改正就可以正确的做到这一点。**

组合复用原则我们应该首选组合，然后才是继承，使用继承时应该严格的遵守里氏替换原则，必须满足“Is-A”的关系是才能使用继承，而组合却是一种“Has-A”的关系。导致错误的使用继承而不是使用组合的一个重要原因可能就是错误的把“Has-A”当成了“Is-A”。

## 开-闭原则（目标、总的指导思想）

**Sunny软件公司开发的CRM（客户关系管理）系统可以显示各种类型的图表**

**注释：**

**(1) 增加一个抽象图表类AbstractChart，将各种具体图表类作为其子类；**

**(2) ChartDisplay类针对抽象图表类进行编程，由客户端来决定使用哪种具体图表。**

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。

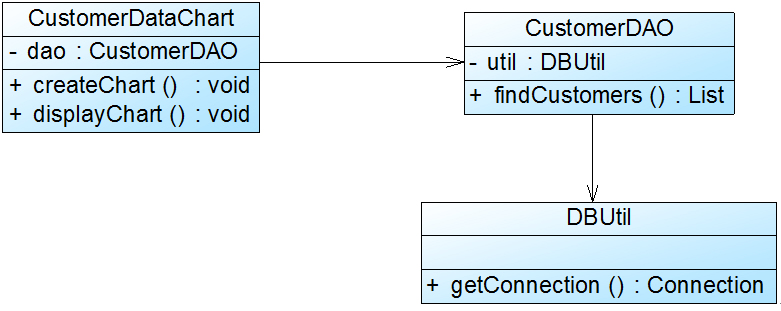
      任何软件都需要面临一个很重要的问题，即它们的需求会随时间的推移而发生变化。当软件系统需要面对新的需求时，我们应该尽量保证系统的设计框架是稳定的。如果一个软件设计符合开闭原则，那么可以非常方便地对系统进行扩展，而且在扩展时无须修改现有代码，使得软件系统在拥有适应性和灵活性的同时具备较好的稳定性和延续性。随着软件规模越来越大，软件寿命越来越长，软件维护成本越来越高，设计满足开闭原则的软件系统也变得越来越重要。

      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。

对扩展开放，对修改关闭。

增加新功能，不改变原有代码。

## 类的单一职责（一个类的定义）



**Sunny软件公司开发人员针对某CRM系统中客户信息图形统计模块**

**注释：**

**(1) DBUtil：负责连接数据库，包含数据库连接方法getConnection()；**

**(2) CustomerDAO：负责操作数据库中的Customer表，包含对Customer表的增删改查等方法，如findCustomers()；**

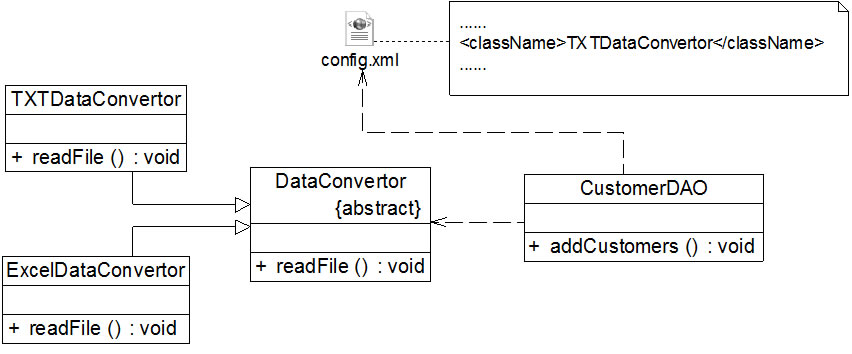
**(3) CustomerDataChart：负责图表的生成和显示，包含方法createChart()和displayChart()**

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。

一个类有且只有一个改变它的原因。

## 依赖倒置（依赖抽象）



**系统经常需要将存储在TXT或Excel文件中的客户信息转存到数据库中，因此需要进行数据格式转换。**

**注释：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。**

定义：

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者抽象类产生的；

理解：

假如我现在需要租个房子，然后我把身边的朋友叫过来，说给我帮忙找个房子。然后朋友们就去找了。还有一种方案，我可以打个电话给中介，让中介给我找。这样的优势就是，我和真正去找房子的人没有直接的联系。如果朋友临时有事，可能就不能帮忙给我找了。但是，中介就不会发生这种情况；所以，当我们要做一件事的时候，并不需要把做事的人拉进来，而只需要下订单，把事情外包给专门做这种事情的人去做，具体是谁就不要管那么多；这样，可以随时更换外包公司,而不影响自身业务。

客户端代码(调用的类)尽量依赖(使用)抽象的组件。

抽象的是稳定的。实现是多变的。

## 组合复用原则（复用的最佳实践）

### 定义

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

另外一种说法是：尽量使用（对象）组合，而非继承如果仅仅为了代码复用优先选择组合复用，而非继承复用。

组合的耦合性相对继承低。

### 优缺点

优点：

（1）新对象存取成分对象的唯一方法是通过成分对象的接口。

（2）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。

（3）这种复用支持包装。

（4）这种复用所需要的依赖较少。

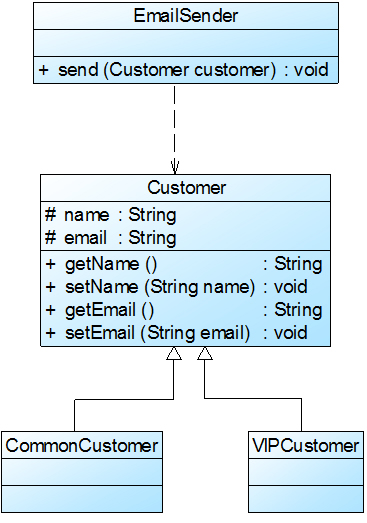
（5）每一个新的类可以将焦点集中到一个任务上。

（6）这种复用可以在运行时间动态进行，新对象可以动态的引用与成分对象类型相同的对象。

缺点：

就是用组合复用建造的系统会有较多的对象需要管理。

## 里氏替换（继承后的重写，指导继承的设计）



**注释：增加一个新的抽象客户类Customer，而将CommonCustomer和VIPCustomer类作为其子类，邮件发送类EmailSender类针对抽象客户类Customer编程，根据里氏代换原则，能够接受基类对象的地方必然能够接受子类对象，因此将EmailSender中的send()方法的参数类型改为Customer，如果需要增加新类型的客户，只需将其作为Customer类的子类即可。**

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。

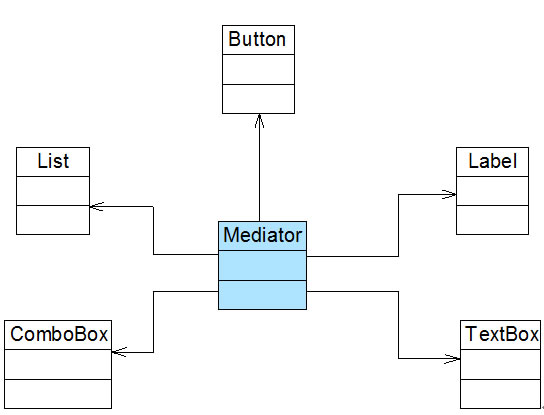
里氏代换原则是实现开闭原则的重要方式之一，由于使用基类对象的地方都可以使用子类对象，因此在程序中尽量使用基类类型来对对象进行定义，而在运行时再确定其子类类型，用子类对象来替换父类对象。

父类出现的地方可以被子类替换，在替换后依然保持原功能。

子类要拥有父类的所有功能。

子类在重写父类方法时，尽量选择扩展重写，防止改变了功能。

## 迪米特法则（类与类交互的原则）



**Sunny软件公司所开发CRM系统包含很多业务操作窗口，在这些窗口中，某些界面控件之间存在复杂的交互关系。**

**注释：可以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。**

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。

不要和陌生人说话。

类与类交互时，在满足功能要求的基础上，传递的数据量越少越好。因为这样可能降低耦合度。