程序设计目标

程序设计的基本目标

正确性

程序是否正确地实现了预订的目标。

健壮性(鲁棒性)

在接收不合法输入或在异常环境下正常运转的程度。

健壮性用来描述程序在异常外界环境下的行为特征，体现程序的容错能力和故障恢复能力。

可靠性

给定时间段内，给定环境下，程序无故障运行的概率。

有效性

程序以最佳方式使用可用资源。资源包括：时间、空间、网络带宽等。通常说的性能是有效性的一部分。

可理解性

程序必须直接反应问题的本来面目，逻辑清晰、结构简单、编排合理、注释简明。

可扩展性

- 新的功能可以很容易地加入到系统中，而不会破坏其底层结构。
- 通过增加新的代码或改动少量代码，即可实现功能新增或增强。

可维护性

程序正式交付使用后，对其进行修改的难易程度。

- 可理解性、可靠性、可扩展性直接影响程序的可维护性。
- 更高的可复用性，能改善可维护性。
- 相关文档的完善程度，也会影响程序的可维护性。

可复用性

- 较高的生产效率
- 较高的软件质量
- 改善系统的可维护性

> 以上内容摘自网络文章。

如何权衡设计目标的优先级

1. 正确性是基础，无法保证正确性，任何设计都是没有意义的，这是任何情况下都必须满足的硬性目标。大多数情况下，健壮性和可靠性也都是硬性目标，必须被满足。
2. 除开硬性目标之外，绝大多数的情况下，可理解性都应当是第一优先考虑的设计目标。难以理解的设计和代码，更容易出错，难以保证正确性，更不用说实现可维护、可扩展等等目标。
3. 绝大多数的情况下，性能往往都不是第一优先考虑的目标，原因如下：
    1. 性能的决定因素主要在于：架构、关键算法。
    2. 8-2原则，20%的关键代码决定了80%的性能。
    3. 简单清晰的代码更容易被优化，复杂的代码则更容易产生错误，并且低效。

    注意：这并非说性能不重要，而是大多数情况下，不是第一优先考虑的目标，特别是当性能与可理解性有冲突时，除非确实必要，否则应尽量避免以性能为由，牺牲可理解性。
4. 对可扩展性目标的追求，可能以增加额外的复杂性为代价，需要适度平衡，避免过度设计。
5. 重复会增加冗余，但是复用会增加依赖，有时候可能需要适当的容忍重复。

扩展性及影响因素

变化

我们对于可扩展性目标的追求，源于变化。

软件系统唯一不变的，就是变化。

解决变化的基本方法在于，分析并区分出系统的哪些部分是容易变化的，哪些是不容易变化的。对不容易变化的部分，进行抽象，使之稳定。对于容易变化的，则进行封装隔离。以稳定不易变化的部分作为系统的基础，在层次结构中，处于越底层的模块，应该是越稳定的。

面向对象方法学对于变化的解决手段主要是：抽象和封装变化，详细见基本设计原则的开闭原则。

耦合

耦合度(Coupling)是对模块间关联程度的度量。耦合度越高则模块间关联程度越高，当模块发生变化时，带来的影响越大。因此，为了减少模块变化带来的影响，应该降低模块间的耦合度，即所谓的**解耦**。由于模块间必然存在耦合关系，所以绝对的零耦合是做不到的。

面向对象方法学中，对象间的耦合关系，即指对象之间的关系，对象之间包含哪些关系及如何解耦，详细见面向对象基础概念中的对象关系。

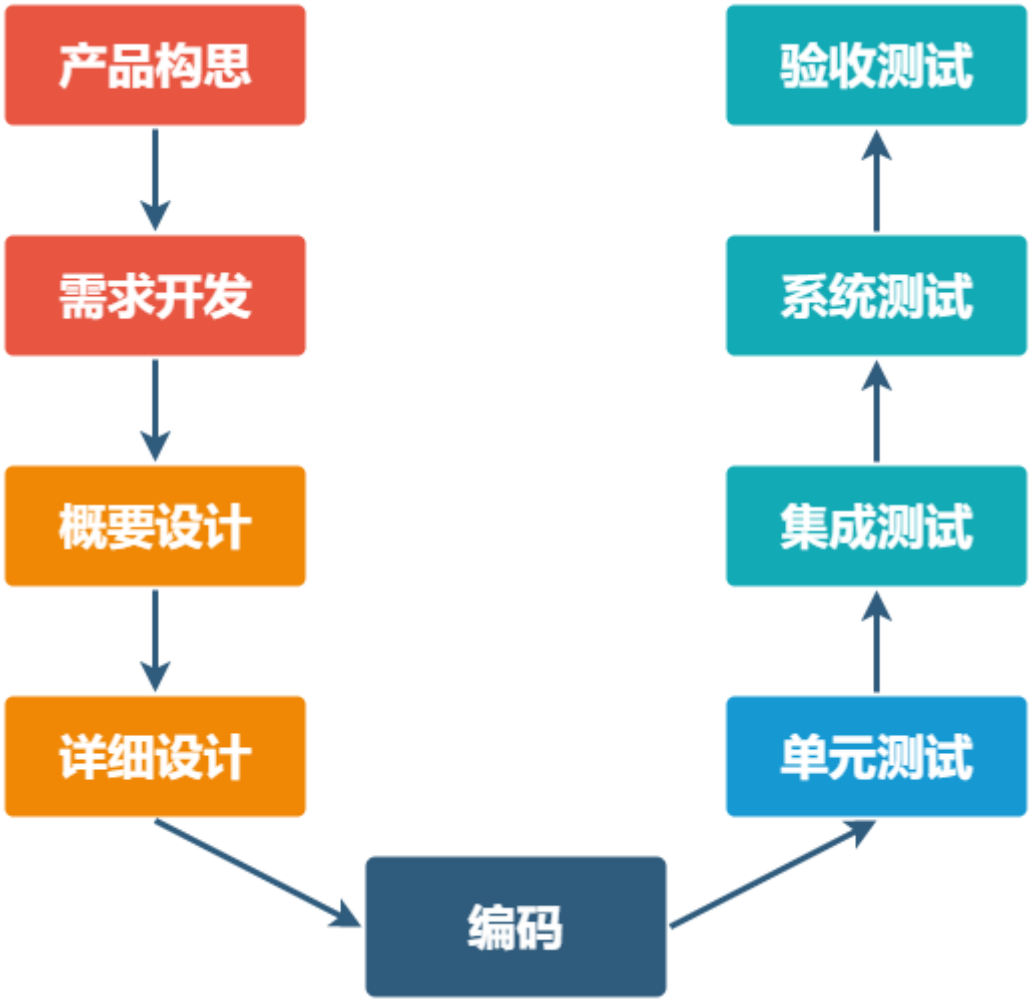
面向对象基础概念

面向对象

引例

我们可以从两个完全不同的视角来看待软件项目开发，包括：

1、开发过程视角，传统的瀑布模型：



2、项目开发团队的角色视角：

当我们站在开发团队的角色视角，来看项目开发时，实际上项目开发的流程，仍然是不变的。只不过这个流程隐藏起来，成为项目经理这个角色的一个主要职责。项目经理，负责控制并协调各个角色，以完成整个开发流程，并最终完成项目。但项目经理本身并不实际完成流程中的各项具体工作，而是将这些具体的工作交给具体负责的角色去完成。

项目团队中的各个角色都有明确的职责，相互之间不可替代。对整个团队来说，这种职责明确的分工，可以明确责任提高组织管理效率和质量；对具体角色来说，可以专注于自己的角色职责，提升技能，进而提高工作效率和质量。

面向过程

面向过程的分析和设计方法来源于人们思考和处理事情的一般逻辑。即分析问题的解决步骤，第一步该怎样->第二步该怎样->第三步该怎样...并用函数将这些步骤一步一步实现。因此面向过程，强调处理过程，以过程来分析、描述和构造软件系统。

面向过程方法具有如下的特点：

1、 与功能高度相关

使用面向过程方法构建系统，其本质是功能分解，从代表目标系统整体功能的单个处理着手，自顶向下不断把复杂的处理分解为子处理，这样一层一层的分解下去，直到仅剩下若干个容易实现的子处理功能为止，然后用相应的工具来描述各个最低层的处理。因此，面向过程方法是围绕实现处理功能的“过程”来构造系统的。

随着时间的推移，过程式的代码由于补丁不断，会变得伤痕累累（行数的不断增加，爆炸式的if...else...嵌套）。当行数到达一定极限时，程序员就很难阅读，也很难在限期内修改掉最新的BUG。同时补丁还可能带来执行效率的下降。

由于用户需求的变化大部分是针对功能的，因此，这种变化对于基于过程的设计来说是灾难性的。用这种方法设计出来的系统结构常常是不稳定的，用户需求的变化往往造成系统结构的较大变化，从而需要花费很大代价才能实现这种变化。

2、 数据与方法分离

在面向过程程序设计中，数据通常与方法分离，有时数据是全局，因此修改代码作用域以外的数据很容易。这说明对数据的访问是非受控而且不可预测的(也就是说可能多个函数都可以访问全局数据)。其次，由于对谁能访问数据无从控制，测试和调试就困难得多。

因此，使用面向过程方法开发的软件，其稳定性、可重用性、可维护性都比较差，特别是在软件规模、复杂度及项目团队规模变得越来越大的情况下，这种缺点会越来越明显。

面向对象

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。

它可以有不同层次的理解：

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。

面向对象方法的优点

面向对象如何克服面向过程的缺点?

1、对象具有更好的稳定性

对象是对客观事物的抽象，更接近事物的本质，因此具有更好的稳定性。这意味着对象能够被更多的复用，并且不容易被改变。（如：引例中，不同项目，项目开发流程可能不同，但大部分角色的职责却始终保持不变。）

2、面向对象符合人们对于世界的认知，有助于组织和构建更为复杂、规模更为庞大的软件系统。这也是面向对象相对于纯粹的面向过程，所具备的最大优势。

3、面向对象中，数据和方法都封装在对象中，对数据的访问是可控的。并且如果设计得当，不会存在全局数据之类的东西，因此可以保证系统中高度的数据完整性。

面向过程与面向对象

面向对象从另一个角度来分析和看待软件系统，而并非是面向过程的替代。两者之间不是相互替代的关系。事实上，面向过程作为一种基础的分析和设计方法，是不可替代的。对象行为特征，仍然需要通过过程的方法去分析、设计和实现。

如果项目规模比较小，复杂度不高，使用面向过程方法未尝不可。

面向对象与语言

面向对象是一种思想和方法，而语言，包括各种建模语言和编程语言(如UML、C、C++、PHP等)，是实现工具。

虽然学习一门建模语言是很重要的一步，但首先学习面向对象技巧更重要，在没有面向对象思想之前学习UML就和没有一点建筑知识就去学习如何阅读一个建筑图纸有点类似。

学习编程语言，也是如此，许多C程序员是通过移植C++而接触到面向对象思想的，通常，C++开发人员只是使用C++编译器的C程序员。

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。看这些文章只有真正懂得什么是对象，什么是面向对象，才能最大程度地对自己有所裨益。

类与对象

类：描述了一组有相同属性（状态）和相同操作（行为）的对象。对象是类的实例。它为属于该类的所有对象提供了统一的抽象描述，其内部包括属性和操作两个主要部分。

对象：是系统中用来描述客观事物的一个实体。一个对象由一组属性和对这些属性进行操作的一组方法组成。对象是问题域或实现域中某些事物的一个抽象，它反映该事物在系统中要保存的信息和发挥的作用。

类与对象的关系就如模具和铸件的关系，类的实例化结果就是对象，而对一类对象的抽象(泛化)就是类。类描述了一组有相同特性（属性）和相同行为（方法）的对象。

为了描述的简便，下面的内容，将不明确区分类和对象。

对象是属性和行为的集合体

一个对象由两个术语定义：属性和行为。比如，人有属性，如眼睛颜色，年龄，身高等，人也有行为，如走路，说话，呼吸等。

在编程语言的语法上面，类也是由成员变量和成员方法组成的，它们分别定义了对象的属性和行为。

对象是一个包括数据和行为的整体，这是传统的面向过程编程方法和面向对象编程方法之间的最大不同。

对象是属性和行为的集合体，但不能简单的认为将任意数据和行为捆绑成为一个整体，就是对象。组成对象的属性和行为，必定存在内在的关联性。

对象是一组自我负责的职责

将对象看成是一组职责，强调的是对象能够做什么。

自我负责的意思是，对象知道怎么做，并且能够很好的完成自己的职责。外部对象不需要关心对象是怎么做的，并且能够信任对象所做的，跟其承诺的职责是相匹配的。

对象专注于自己的职责，因此可简化单一对象的设计和编写，同时有利于保持对象的稳定，避免由于需求和设计的变更而带来的修改。

封装

面向对象的三个基本特征：封装、继承和多态，当中封装是最基础也是最强有力的工具。

封装的含义

封装是指将属性和操作封进一个对象中，它的内部信息对外界隐藏，不允许外界直接存取对象的属性，只能通过对象提供的有限接口对对象的属性数据进行操作。封装是面向对象，最基础也是最强有力的工具。

封装有两个特性：

1、封装是将属性与方法结合起来，成为一个不可分割的整体。

2、信息隐藏。封装是隐藏实现，即隐藏内部实现细节，只提供有限的接口，与外部交互。

类的封装是通过类定义及访问控制来实现的。

封装的优点

1、模块化、易于使用(职责明确清晰、使用者不需要了解实现细节)。

2、封装使对象对外形成一个边界，只通过有限的接口与外部交互，减少与外部对象的依赖关系(高内聚低耦合)。同时能够隔绝错误的交叉感染，使错误被局部化，降低错误排查的难度。

3、隐藏实现易于应对可能的变化。封装使对象能够自我负责，只要接口保持不变，内部实现可以任意调整，而不会影响对象的使用者。

继承

基本概念

继承是指子类自动拥有父类的属性和方法的机制。

继承是类之间的一种关系(继承关系)。

继承的优点

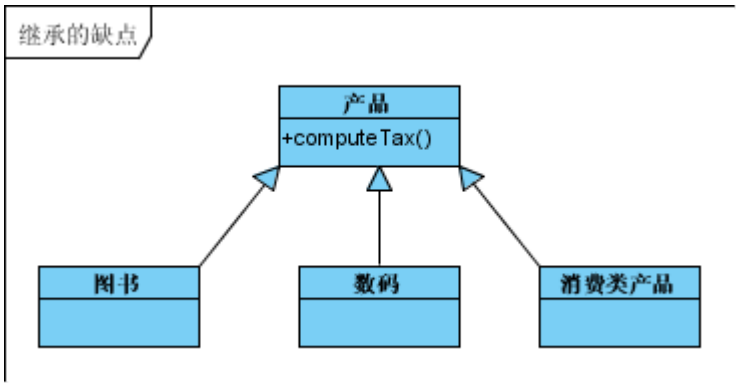
1、通过继承机制，可以使用已有的类来定义新的类。新的类自动拥有已有类的属性和方法，并可在此基础上进行扩展。因此简化了类的创建工作。

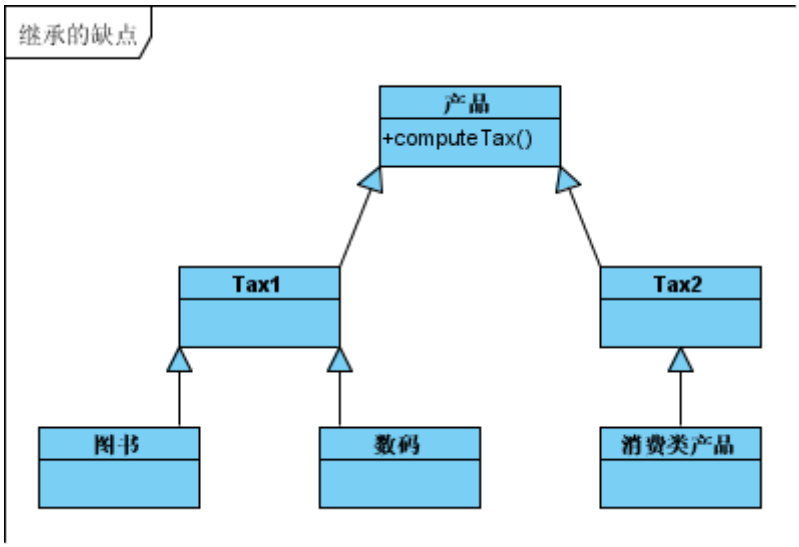
2、继承机制使公共的特性可以被共享，提高了软件的重用性。

继承的代价

1、继承破坏封装。继承使子类可以访问父类的保护成员，相比其它类，子类获得更多对父类的访问权限。子类与父类之间紧密耦合，相互依赖。

2、继承支持扩展，但往往以增加系统结构的复杂度为代价，特别是继承层次教深时，会使系统结构变得复杂，难以理解和控制。





何时使用继承

由于继承能够使子类自动拥有父类的属性和方法，很容易导致为使用父类的功能而随意继承。

1、当类B与类A的关系是“is-a”的关系时，才让B从A继承。

“is-a”的关系说明，子类是父类的一种特例，父类的所有接口都对子类同样适用。子类通过重写父类的同名函数，或适当增加特有接口来实现自己区别于其它同级子类的特性。

当子类拥有太多自己的特有的接口，进而掩盖从父类继承的特性时，需要考虑重新设计。

2、尽量使用组合来代替继承。

多态

基本概念

多态是面向对象技术的一个重要概念和技术。

多态是指同一操作作用于不同的对象，可以有不同的解释，产生不同的结果。

编译型语言的多态性分为：编译时多态和运行时多态。

编译时多态通过函数重载实现，编译器在编译时根据传递的参数决定调用哪个函数，实现何种操作。

运行时多态通常通过派生类重写基类中的同名函数来实现的。不同语言实现不同。

重写与重载

重写是面向对象的一个概念，它规定一个派生类可以创建其基类某个方法的不同实现代码。

重载规定一个方法可以具有许多不同的接口，但方法的名称是相同的。

运行时多态的实现方式

通过继承实现多态性

通过抽象类实现多态性

通过接口实现多态性

消息

消息就是向对象发出的请求，当一个消息发送给某个对象时，接收到消息的对象经过解析，然后予以响应。

消息传递是对象间的通信机制。

消息的本质是函数调用。

对象关系

对象间的关系包括：

依赖(Dependency)

一个模型元素（独立模型元素）发生变化会影响依赖它的另一个模型元素（依赖模型元素）的变化。体现使用(use)关系，这种使用关系是具有偶然性的、临时性的、非常弱的。

语言层面上，依赖关系一般通过局域变量、方法的形参，或者对静态方法的调用来实现。

关联(Association)

关联是类与类之间的联接，它使一个类知道另一个类的属性和方法。关联可以是双向的，也可以是单向的。

关联关系，体现has a关系，关联与被关联双方的关系一般是平等的。关联的两个对象彼此间没有任何强制性的约束，只要二者同意，可以随时解除关系或是进行关联，它们在生命周期问题上没有任何约定。被关联的对象还可以再被别的对象关联，所以关联是可以共享的。

语言层面上，关联关系一般通过成员变量实现的(持有另一个对象的引用)。

聚合(Aggregation)

是关联（Association）关系的一种，是一种强的关联关系。

聚合关系，体现"owns a"关系，聚合双方是不平等的，是整体和部分的关系。

聚合关系中，整体和部分可以具有各自的生命周期。部分可以属于多个整体对象，也可以为多个整体对象共享。

语言层面上，聚合关系同样通过成员变量实现的(持有另一个对象的引用)。

组合(Composition)

是关联（Association）关系的一种，是一种比聚合更强的关联关系，也称为强聚合。

组合关系，体现"contains a"的关系。组合双方是整体和部分的关系，部分属于整体，整体不存在，部分一定不存在，然而部分不存在整体是可以存在的。

组合关系中，部分和整体的生命周期一样。组合对象完全支配其组成部分，包括它们的创建和湮灭等。一个组合关系的成分对象是不能与另一个组合关系共享的。

语言层面上，组合关系同样通过成员变量实现的(持有另一个对象的值)。

泛化(继承)

一般到具体，"is a" 的关系。

解耦

上述的对象关系也就是对象间的耦合关系，这些关系：依赖、关联、聚合、组合、泛化，它们的耦合度依次增强。降低对象之间的耦合度，即对象的解耦，应遵循下面的原则：

- 优先使用耦合度低的关系。
- 减少依赖，零依赖最佳。
- 当依赖关系不可避免的时候，必须确保单向依赖。

面向对象基本设计原则

原则通常是软件开发过程中，被大量实践证明了能够有效改善设计的一般准则。原则并非定律，你不一定要遵守它，但如果没有足够充分的理由，你也不应该违背它。

“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(Software entities should be open for extension,but closed for modification.)。即在设计一个模块的时候，应当使这个模块可以在不被修改的前提下被扩展。

满足“开-闭”原则的系统的优点

a)通过扩展已有的软件系统，可以提供新的行为，以满足对软件的新需求，使变化中的软件系统有一定的适应性和灵活性。

b)已有的软件模块，特别是最重要的抽象层模块不能再修改，这就使变化中的软件系统有一定的稳定性和延续性。

c)这样的系统同时满足了可复用性与可维护性。

如何实现“开-闭”原则

抽象

在面向对象设计中，不允许更改的是系统的抽象层，而允许扩展的是系统的实现层。换言之，定义一个一劳永逸的抽象设计层，允许尽可能多的行为在实现层被实现。

解决问题关键在于抽象化，抽象化是面向对象设计的第一个核心本质。

对一个事物抽象化，实质上是在概括归纳总结它的本质。抽象让我们抓住最最重要的东西，从更高一层去思考。这降低了思考的复杂度，我们不用同时考虑那么多的东西。换言之，我们封装了事物的本质，看不到任何细节。

在面向对象编程中，通过抽象类及接口，规定了具体类的特征作为抽象层，相对稳定，不需更改，从而满足“对修改关闭”；而从抽象类导出的具体类可以改变系统的行为，从而满足“对扩展开放”。

对实体进行扩展时，不必改动软件的源代码或者二进制代码。关键在于抽象。

封装可变性

“开-闭”原则也就是“对可变性的封装原则”（Principle of Encapsulation of Variation，EVP）。即找到一个系统的可变因素，将之封装起来。换言之，在你的设计中什么可能会发生变化，应使之成为抽象层而封装，而不是什么会导致设计改变才封装。

“对可变性的封装原则”意味着：

a)一种可变性不应当散落在代码的许多角落，而应当被封装到一个对象里面。同一可变性的不同表象意味着同一个继承等级结构中的具体子类。因此，此处可以期待继承关系的出现。继承是封装变化的方法，而不仅仅是从一般的对象生成特殊的对象。

b)一种可变性不应当与另一种可变性混合在一起。类图的继承结构如果超过两层，很可能意味着两种不同的可变性混合在了一起。

使用“可变性封装原则”来进行设计可以使系统遵守“开-闭”原则。

即使无法百分之百的做到“开-闭”原则，但朝这个方向努力，可以显著改善一个系统的结构。

单一职责原则 (Single Responsibility Principle,\SRP)

概念

一个类应该有且只有一个变化的原因。

为什么将不同的职责分离到单独的类中是如此的重要呢?

因为每一个职责都是一个变化的中心。当需求变化时，这个变化将通过更改职责相关的类来体现。

如果一个类拥有多于一个的职责，则这些职责就耦合到在了一起，那么就会有多于一个原因来导致这个类的变化。对于某一职责的更改可能会损害类满足其他耦合职责的能力。这样职责的耦合会导致设计的脆弱，以至于当职责发生更改时产生无法预期的破坏。

单一职责的优点

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。

提高类的可读性，提高系统的可维护性。

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。

里氏替换原则（Liskov Substitution Principle, LSP）

概念

定义：如果对每一个类型为T1的对象O1，都有类型为T2 的对象O2，使得以T1定义的所有程序P在所有的对象O1都代换为O2时，程序P的行为没有变化，那么类型T2是类型T1的子类型。

即，一个软件实体如果使用的是一个基类的话，那么一定适用于其子类。而且它觉察不出基类对象和子类对象的区别。也就是说，在软件里面，把基类都替换成它的子类，程序的行为没有变化。

反过来的代换不成立，如果一个软件实体使用的是一个子类的话，那么它不一定适用于基类。

任何基类可以出现的地方，子类一定可以出现。

基于契约的设计、抽象出公共部分作为抽象基类的设计。

里氏代换原则与“开-闭”原则的关系

实现“开-闭”原则的关键步骤是抽象化。基类与子类之间的继承关系就是抽象化的体现。因此里氏代换原则是对实现抽象化的具体步骤的规范。

违反里氏代换原则意味着违反了“开-闭”原则，反之未必。

依赖倒转原则（Dependence Inversion Principle, DIP）

概念

依赖倒转原则就是要依赖于抽象，不要依赖于实现。（Abstractions should not depend upon details. Details should depend upon abstractions.）要针对接口编程，不要针对实现编程。（Program to an interface, not an implementation.）

也就是说应当使用接口和抽象类进行变量类型声明、参数类型声明、方法返还类型说明，以及数据类型的转换等。而不要用具体类进行变量的类型声明、参数类型声明、方法返还类型说明，以及数据类型的转换等。要保证做到这一点，一个具体类应当只实现接口和抽象类中声明过的方法，而不要给出多余的方法。

传统的过程性系统的设计办法倾向于使高层次的模块依赖于低层次的模块，抽象层次依赖于具体层次。倒转原则就是把这个错误的依赖关系倒转过来。

面向对象设计的重要原则是创建抽象化，并且从抽象化导出具体化，具体化给出不同的实现。继承关系就是一种从抽象化到具体化的导出。

抽象层包含的应该是应用系统的商务逻辑和宏观的、对整个系统来说重要的战略性决定，是必然性的体现。具体层次含有的是一些次要的与实现有关的算法和逻辑，以及战术性的决定，带有相当大的偶然性选择。具体层次的代码是经常变动的，不能避免出现错误。

从复用的角度来说，高层次的模块是应当复用的，而且是复用的重点，因为它含有一个应用系统最重要的宏观商务逻辑，是较为稳定的。而在传统的过程性设计中，复用则侧重于具体层次模块的复用。

依赖倒转原则则是对传统的过程性设计方法的“倒转”，是高层次模块复用及其可维护性的有效规范。

特例：对象的创建过程是违背“开—闭”原则以及依赖倒转原则的，但通过工厂模式，能很好地解决对象创建过程中的依赖倒转问题。

与“开-闭”原则关系

“开-闭”原则与依赖倒转原则是目标和手段的关系。如果说开闭原则是目标,依赖倒转原则是到达"开闭"原则的手段。如果要达到最好的"开闭"原则，就要尽量的遵守依赖倒转原则，依赖倒转原则是对"抽象化"的最好规范。

里氏代换原则是依赖倒转原则的基础，依赖倒转原则是里氏代换原则的重要补充。

耦合（或者依赖）关系的种类：

零耦合（Nil Coupling）关系：两个类没有耦合关系

具体耦合（Concrete Coupling）关系：发生在两个具体的（可实例化的）类之间，经由一个类对另一个具体类的直接引用造成。

抽象耦合（Abstract Coupling）关系：发生在一个具体类和一个抽象类（或接口）之间，使两个必须发生关系的类之间存有最大的灵活性。

如何把握耦合

我们应该尽可能的避免实现继承，原因如下：

1 失去灵活性，使用具体类会给底层的修改带来麻烦。

2 耦合问题，耦合是指两个实体相互依赖于对方的一个量度。程序员每天都在(有意识地或者无意识地)做出影响耦合的决定：类耦合、API耦合、应用程序耦合等等。在一个用扩展的继承实现系统中，派生类是非常紧密的与基类耦合，而且这种紧密的连接可能是被不期望的。如B extends A ，当B不全用A中的所有methods时，这时候，B调用的方法可能会产生错误!

我们必须客观的评价耦合度，系统之间不可能总是松耦合的，那样肯定什么也做不了。

我们决定耦合的程度的依据何在呢?

简单的说，就是根据需求的稳定性，来决定耦合的程度。对于稳定性高的需求，不容易发生变化的需求，我们完全可以把各类设计成紧耦合的(我们虽然讨论类之间的耦合度，但其实功能块、模块、包之间的耦合度也是一样的)，因为这样可以提高效率，而且我们还可以使用一些更好的技术来提高效率或简化代码，例如c# 中的内部类技术。可是，如果需求极有可能变化，我们就需要充分的考虑类之间的耦合问题，我们可以想出各种各样的办法来降低耦合程度，但是归纳起来，不外乎增加抽象的层次来隔离不同的类，这个抽象层次可以是抽象的类、具体的类，也可以是接口，或是一组的类。我们可以用一句话来概括降低耦合度的思想："针对接口编程，而不是针对实现编程。

在我们进行编码的时候，都会留下我们的指纹，如public的多少，代码的格式等等。我们可以耦合度量评估重新构建代码的风险。因为重新构建实际上是维护编码的一种形式，维护中遇到的那些麻烦事在重新构建时同样会遇到。我们知道在重新构建之后，最常见的随机bug大部分都是不当耦合造成的 。

如果不稳定因素越大，它的耦合度也就越大。

某类的不稳定因素=依赖的类个数/被依赖的类个数

依赖的类个数 = 在编译此类的时被编译的其它类的个数总和

怎样将大系统拆分成小系统

解决这个问题的一个思路是将许多类集合成一个更高层次的单位,形成一个高内聚、低耦合的类的集合，这是我们设计过程中应该着重考虑的问题!

耦合的目标是维护依赖的单向性，有时我们也会需要使用坏的耦合。在这种情况下，应当小心记录下原因，以帮助日后该代码的用户了解使用耦合真正的原因。

3.4怎样做到依赖倒转?

以抽象方式耦合是依赖倒转原则的关键。抽象耦合关系总要涉及具体类从抽象类继承，并且需要保证在任何引用到基类的地方都可以改换成其子类，因此，里氏代换原则是依赖倒转原则的基础。

在抽象层次上的耦合虽然有灵活性，但也带来了额外的复杂性，如果一个具体类发生变化的可能性非常小，那么抽象耦合能发挥的好处便十分有限，这时可以用具体耦合反而会更好。

层次化：所有结构良好的面向对象构架都具有清晰的层次定义，每个层次通过一个定义良好的、受控的接口向外提供一组内聚的服务。

依赖于抽象：建议不依赖于具体类，即程序中所有的依赖关系都应该终止于抽象类或者接口。尽量做到：

1、任何变量都不应该持有一个指向具体类的指针或者引用。

2、任何类都不应该从具体类派生。

3、任何方法都不应该覆写它的任何基类中的已经实现的方法。

依赖倒转原则的优缺点

依赖倒转原则虽然很强大，但却最不容易实现。因为依赖倒转的缘故，对象的创建很可能要使用对象工厂，以避免对具体类的直接引用，此原则的使用可能还会导致产生大量的类，对不熟悉面向对象技术的工程师来说，维护这样的系统需要较好地理解面向对象设计。

依赖倒转原则假定所有的具体类都是会变化的，这也不总是正确。有一些具体类可能是相当稳定，不会变化的，使用这个具体类实例的应用完全可以依赖于这个具体类型，而不必为此创建一个抽象类型。

合成/聚合复用原则（Composite/Aggregate Reuse Principle或CARP）

概念

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这些对象的目的。

应首先使用合成/聚合，合成/聚合则使系统灵活，其次才考虑继承，达到复用的目的。而使用继承时，要严格遵循里氏替换原则。有效地使用继承会有助于对问题的理解，降低复杂度，而滥用继承会增加系统构建、维护时的难度及系统的复杂度。

如果两个类是“Has-a”关系应使用合成、聚合，如果是“Is-a”关系可使用继承。"Is-A"是严格的分类学意义上定义，意思是一个类是另一个类的"一种"。而"Has-A"则不同，它表示某一个角色具有某一项责任。

什么是合成？什么是聚合？

合成（或组合Composition）和聚合（Aggregation）都是关联（Association）的特殊种类。

聚合表示整体和部分的关系，表示“拥有”。如奔驰S360汽车，对奔驰S360引擎、奔驰S360轮胎的关系是聚合关系，离开了奔驰S360汽车，引擎、轮胎就失去了存在的意义。在设计中, 聚合不应该频繁出现，这样会增大设计的耦合度。

合成则是一种更强的“拥有”，部分和整体的生命周期一样。合成的新的对象完全支配其组成部分，包括它们的创建和湮灭等。一个合成关系的成分对象是不能与另一个合成关系共享的。

换句话说，合成是值的聚合（Aggregation by Value），而一般说的聚合是引用的聚合（Aggregation by Reference）。

明白了合成和聚合关系，再来理解合成/聚合原则应该就清楚了，要避免在系统设计中出现，一个类的继承层次超过3层，则需考虑重构代码，或者重新设计结构。当然最好的办法就是考虑使用合成/聚合原则。

通过合成/聚合的优缺点

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) 这种复用所需的依赖较少。

5) 每一个新的类可以将焦点集中在一个任务上。

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

7) 作为复用手段可以应用到几乎任何环境中去。

缺点：

就是系统中会有较多的对象需要管理。

通过继承来进行复用的优缺点

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的关系自动进入子类。

修改和扩展继承而来的实现较为容易。

缺点：

继承复用破坏封装，因为继承将基类的实现细节暴露给子类。由于基类的内部细节常常是对于子类透明的，所以这种复用是透明的复用，又称“白箱”复用。

如果基类发生改变，那么子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，不可能在运行时间内发生改变，没有足够的灵活性。

继承只能在有限的环境中使用。

迪米特法则（Law of Demeter，LoD）或最少知识原则（Least Knowledge Principle，LKP）

概述

定义：一个软件实体应当尽可能少的与其他实体发生相互作用。

这样，当一个模块修改时，就会尽量少的影响其他的模块。扩展会相对容易。

这是对软件实体之间通信的限制。它要求限制软件实体之间通信的宽度和深度。

迪米特法则的其他表述

1）只与你直接的朋友们通信。

2）不要跟“陌生人”说话。

3）每一个软件单位对其他的单位都只有最少的知识，而且局限于那些与本单位密切相关的软件单位。

狭义的迪米特法则

如果两个类不必彼此直接通信，那么这两个类就不应当发生直接的相互作用。如果其中的一个类需要调用另一个类的某一个方法的话，可以通过第三者转发这个调用。

朋友圈的确定

“朋友”条件：

1）当前对象本身（this）

2）以参量形式传入到当前对象方法中的对象

3）当前对象的实例变量直接引用的对象

4）当前对象的实例变量如果是一个聚集，那么聚集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”。

缺点：会在系统里造出大量的小方法，散落在系统的各个角落。

与依赖倒转原则互补使用

狭义的迪米特法则的缺点

在系统里造出大量的小方法，这些方法仅仅是传递间接的调用，与系统的商务逻辑无关。

遵循类之间的迪米特法则会使一个系统的局部设计简化，因为每一个局部都不会和远距离的对象有直接的关联。但是，这也会造成系统的不同模块之间的通信效率降低，也会使系统的不同模块之间不容易协调。

迪米特法则与设计模式

门面（外观）模式和调停者（中介者）模式实际上就是迪米特法则的具体应用。

广义迪米特法则

迪米特法则的主要用意是控制信息的过载。在将迪米特法则运用到系统设计中时，要注意下面的几点：

1）在类的划分上，应当创建有弱耦合的类。

2）在类的结构设计上，每一个类都应当尽量降低成员的访问权限。

3）在类的设计上，只要有可能，一个类应当设计成不变类。

4）在对其他类的引用上，一个对象对其对象的引用应当降到最低。

广义迪米特法则在类的设计上的体现

1）优先考虑将一个类设置成不变类

2）尽量降低一个类的访问权限

3）谨慎使用Serializable

4）尽量降低成员的访问权限

5）取代C Struct

迪米特法则又叫作最少知识原则（Least Knowledge Principle或简写为LKP），就是说一个对象应当对其他对象有尽可能少的了解。

如何实现迪米特法则

迪米特法则的主要用意是控制信息的过载，在将其运用到系统设计中应注意以下几点：

1) 在类的划分上，应当创建有弱耦合的类。类之间的耦合越弱，就越有利于复用。

2) 在类的结构设计上，每一个类都应当尽量降低成员的访问权限。一个类不应当public自己的属性，而应当提供取值和赋值的方法让外界间接访问自己的属性。

3) 在类的设计上，只要有可能，一个类应当设计成不变类。

4) 在对其它对象的引用上，一个类对其它对象的引用应该降到最低。

接口隔离原则（interface separate principle, ISP）

概念

接口隔离原则：使用多个专门的接口比使用单一的总接口要好。也就是说，一个类对另外一个类的依赖性应当是建立在最小的接口上。

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c# 语言里面的Interface结构。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。(上面说的一个类型，可以理解成一个类，我们定义了一个类，也就是定义了一种新的类型)。

当我们把"接口"理解成一个类所提供的所有方法的特征集合的时候，这就是一种逻辑上的概念。接口的划分就直接带来类型的划分。这里，我们可以把接口理解成角色，一个接口就只是代表一个角色，每个角色都有它特定的一个接口，这里的这个原则可以叫做"角色隔离原则"。

如果把"接口"理解成狭义的特定语言的接口，那么ISP表达的意思是说，对不同的客户端，同一个角色提供宽窄不同的接口，也就是定制服务，个性化服务。就是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。

这也是对软件实体之间通信的限制。但它限制的只是通信的宽度，就是说通信要尽可能的窄。

遵循迪米特法则和接口隔离原则，会使一个软件系统功能扩展时，修改的压力不会传到别的对象那里。

如何实现接口隔离原则

不应该强迫用户依赖于他们不用的方法。

1、利用委托分离接口。

2、利用多继承分离接口。

> 以上内容主要摘自网络文章。

面向对象设计实践

可以工作的类

设计和编写可以工作的类，需要考虑并回答如下的问题：

1. 如何命名？

- 怎样才是好的命名？
- 应当遵守怎样的命名规则？

2. 类的职责是什么？

- 能够做什么？不能够做什么？
- 类在系统中处于什么样的层次？
- 类与其它类是什么关系？

3. 成员变量怎么设置？

- 什么样的变量可以成为成员变量？
- 什么样的变量不应该成为成员变量？

4. 成员方法如何设置？

- 如何确定成员应该是私有、公有或保护的？
- 如何区分接口与实现？
- 什么样的接口才是好的接口？怎样设计实现好的接口？

5. 如何使用继承和多态？

好的命名

怎样才是好的命名

好的命名要能够完全、准确的描述出命名对象(类、方法、变量等)所代表的事物、含义。

好的命名遵循的原则

1. 简单、简洁

尽量使用简单的单词。

尽量使用简洁的名称，避免冗余，特别是需要结合上下文环境，去掉不必要的修饰单词。

2. 直观

命名应能够直观体现变量、方法、类的职责。

3. 一致性

在多个地方出现的具有相同意义的变量，应该使用一致的命名。相同性质的类或方法应该使用一致的命名方式。命名应遵循项目的规范。

例1：

```
class CustomerCart
{
    public function getCustomerCoupons()
    {
    }
}
```

这里`getCustomerCoupons()`方法名中的`Customer`是冗余的。`CustomerCart`类本身就是`Customer`的购物车，其处理的都是跟`Customer`购物车有关的逻辑，优惠券也一定是属于`Customer`范畴内的优惠券，所以应该直接命名为`getCoupons()`，更加简洁，也没有歧义。

例2：

```
public function createBuyOrder()
public function createRenewOrder()
public function createUpgradeOrder()
```

某一项SAAS产品其业务，包括了购买、续费、升级等业务操作。这些业务的服务类，都有创建业务操作订单的接口。它们在性质上都是一样的，所以命名上应当保持一致，即统一使用`createXxxOrder`命名。

例3：

```
class Category extends ActiveRecord
{
    public static function getAllCategoryByNumber() {
        $items = self::find()->all();

        $names =[];
        foreach ($items as $item) {
            $names[$item->number] = $item->name;
        }

        return $names;
    }

    public static function getAllCategory() {
        $items = self::find()->all();

        $names =[];
        foreach ($items as $item) {
            $names[$item->id] = $item->name;
        }

        return $names;
    }
}
```

这个例子中，`getAllCategoryByNumber()`接口实现的功能是查询并返回所有分类的名称，并且以分类编号作为索引。所以重点应体现名称和索引方式，至于名称中的`Category`是多余，因为对象本身就是分类，没有必要再体现。

改进后的代码：

```
class Category extends ActiveRecord
{
    /**
     * @return array 返回所有分类名称([number=>name,...])
     */
    public static function getAllNamesIndexByNumber() {
        ...
    }

    /**
     * @return array 返回所有分类名称([id=>name,...])
```

```
     */
    public static function getAllNames() {
        ...
    }
}
```

名称应该直观体现所命名的类、方法、变量的含义，对于类名和方法名，应该体现类和方法的核心职责，需要注意，大部分情况下都不要将类或方法的内部实现体现在名称上面。

比如：有这样一个类，负责将话单数据按照一定的格式生成报表，并将报表上传到指定的服务器上。程序员将这个类命名为：`FtpVoiceBillReportsCommand`。这里名称中带有Ftp，程序员的本意是表达通过Ftp上传报表。但是，经过分析，我们发现，类的核心职责是生成报表，并上传到指定服务器，FTP上传只是当前的采用的上传方式，即属于内部实现，因此不应该体现在类名中。将内部实现体现在命名中，当内部实现修改时，比如这里由FTP上传改为其它上传方式时，则名称将会变得不合适。此时如果不修改类名，则容易造成误导，如果修改类名，则大大增加工作量。

> 《代码整洁之道》、《代码大全2》有关于命名的专门章节，详细介绍了如何命名，并有更多详细的例子。

变量使用原则

尽可能推迟变量定义

尽可能将变量的定义推迟到第一次使用该变量，并且有足够信息初始化该变量的时候。原因如下：

避免不必要的开销

大部分高级语言，特别是面向对象编程语言(比如C++、Java、PHP等等)，对象的创建和销毁都需要额外的开销。比如C++中对象在创建和销毁时，需要承担对象构造、析构的开销；如果对象创建时，没有足够的信息来初始化对象，只能使用默认构造函数进行初始化，等到有足够信息时，需要对对象进行二次赋值，则默认构造函数的开销将是无效开销。

例子：

```
// 此函数太早定义了变量"encrypted"
string encryptPassword(const string& password)
{
  string encrypted;

  if (password.length() < MINIMUM_PASSWORD_LENGTH) {
     throw logic_error("Password is too short");
  }

  // 进行必要的操作，将口令的加密版本放进encrypted之中;

  return encrypted;
}
```

对象encrypted在函数中并非完全没用，但如果有异常抛出时，就是无用的。但是，即使encryptPassword抛出异常，程序也要承担encrypted构造和析构的开销。所以，最好将encrypted推迟到确实需要它时才定义：

```
string encryptPassword(const string& password)
{
  if (password.length() < MINIMUM_PASSWORD_LENGTH) {
    throw logic_error("Password is too short");
  }

  string encrypted;

  // 进行必要的操作，将口令的加密版本放进encrypted之中;

  return encrypted;
}
```

这段代码还不是那么严谨，因为encrypted定义时没有带任何初始化参数。这将导致它的缺省构造函数被调用。 更好的方法是用password来初始化encrypted，从而绕过了对缺省构造函数不必要的调用：

```
// 定义和初始化encrypted的最好方式
string encryptPassword(const string& password)
{
  if (password.length() < MINIMUM_PASSWORD_LENGTH) {
    throw logic_error("Password is too short");
  }

  string encrypted(password);     // 通过拷贝构造函数定义并初始化

  encrypt(encrypted);
  return encrypted;
}
```

这样做，不仅可以避免对不必要的对象进行构造和析构，还可以避免无意义的对缺省构造函数的调用。

更有利于阅读

在变量被第一次使用，并且有足够信息对变量进行初始化时定义变量，变量本身的用途将不言自明，在这时定义变量有益于表明变量的含义。 另外，人在阅读代码过程中，能够有效记忆的行数是有限制的，将变量推迟到第一次被使用的时候才定义，可以尽量减少往前翻代码来查找确定变量含义的需要，减少阅读过程中的心智负担。

> 以上内容主要参考自《Effective C++》第二版。

尽可能缩短变量生命周期

本原则实际上包含了上一条“尽可能推迟变量定义”原则。

跨度

跨度表明对一个变量引用的集中程度。

```
a = 0
b = 0
c = 0
a = b + c
b = a + 1
b = b / c
```

a的第一次引用和第二次引用之间存在两行代码，因此变量的跨度是2，b的第一次引用和第二次引用之间存在一行代码，因此跨度是1。

平均跨度可以通过计算各个跨度的平均值得到。还是上面的例子，b的第一次引用和第二次引用的跨度为1，第二次引用和第三次引用的跨度为0，因此平均跨度为(1 + 0) / 2 = 0.5。

存活时间

存活时间是指一个变量存在期间所跨越的语句总数。变量的存活时间开始于引用它的第一条语句，结束于引用它的最后一条语句。

注意：存活时间只关注第一次引用和最后一次引用，而不关注中间有多少次引用。长存活时间意味着一个变量历经了许多条语句，而短存活时间意味着它只历经很少的语句。

减少跨度和存活时间

当把变量的引用点集中在一起的时候，阅读者能每次只关注一部分代码，如果这些引用点之间的距离非常远，那里就要迫使阅读者的目光在程序里跳来跳去。因此减少跨度，可以增强代码的可读性。

保持较低的存活时间的好处在于能够**减小攻击窗口**，因为存活时间短，该变量被错误或无意修改的可能性就变小了。

> 以上内容主要摘自《代码大全2》。

单一职责

单一职责原则，同样适用于变量。即一个变量，应该只有一个用途。遵循单一职责，可以提高代码的可读性，减少犯错的机会，代码更容易维护，也更容易重构。

一致性

相同含义的变量，应该有一致的命名。一致性的命名，可以提高代码的可阅读性；相反，不一致的命名，增加阅读难度，并且容易造成误导。

确定对象职责

职责的含义

职责的含义：

能够做什么?

不能够做什么?

面向对象构造的系统是由对象组成，通过对象之间的交互完成相应的功能。对象的主要价值或者意义，在于能够为其它的对象提供什么服务，或者说承担什么样的职责。用户并不关心对象是如何实现这些服务的。

职责不仅界定对象能够做什么，更重要的是需要界定对象不能够做什么，也就是需要为对象设定一个边界。每个对象都应当只有一个核心职责，对象的所有元素都应当专注于这个核心职责，即高内聚。

对象应当是自我负责的，即对象知道怎么实现自己的职责。外部对象不需要关心对象是怎么做的，并且能够信任对象所做的，跟其承诺的职责是相匹配的。

对象应当知道完成自身职责所需的外部依赖，并在接口中明确声明这些依赖。

对象职责强调接口是对象价值的主要体现。在设计过程中，类的接口应该要重点考虑。

抽象

抽象即探寻对象的本质。

当我们进行程序设计时，往往需要从需求入手，分析并找出特定功能或相近的功能集合，封装得到类。需求想要实现的功能，转化为类的职责。但仅仅如此，是远远不够的。

因为功能与需求接近，而需求往往容易变化。因此我们需要在这个基础上，做进一步的分析和处理，探寻我们所获得的对象的本质，以便能够使最终的设计和编码尽可能稳定、具有灵活性和适应性。这个过程即抽象的过程。

当类越接近需求，就越容易受到需求变更的影响，并且不容易被复用。而抽象的意义，在于找出本质的东西，使得类及其接口更加稳定、更具有适用性。从而减少需求变更造成的影响，并增加被复用的机会。

例1:

```php
class ProductCreator extends CComponent
{
    public function __construct(array $data) {...}
    public function getAttributes() {...}
    public function __set($name, $value) {...}
}
```

改为:

```php
class ArrayToObject extends ArrayObject
{
    ...
}
```

例2:

下面是一个SAAS产品提供的客户端接口，当客户对订购的该SAAS产品进行续费时，业务系统使用该接口来变更产品的到期日期：

```
XxxProduct::renew($username, $currentExpireDate, $newExpireDate)
```

对于SAAS产品本身来说，只有到期日期的概念，没有续费的概念。续费是一项业务操作，是在业务系统中操作的，所以将接口改为`modifyExpireDate`更合适：

```
XxxProduct::modifyExpireDate($username, $currentExpireDate, $newExpireDate)
```

对象边界

为对象设定边界，即要明确对象不能够做什么。

对象能够做的越多，则引起变化的因素就越多，越不稳定。反之，对象能够做的越少，则引起变化的因素就越少，越稳定。如果对象什么都不做，也就不会受变化影响，当然现实当中不可能有这样的对象。

为对象设定边界，是为了使对象专注于某个核心职责，提高对象的内聚性，使对象更稳定。

内聚性

对象的内聚性，是对象内部各个元素彼此结合的紧密程度的度量。内聚性越高，各个元素结合的紧密程度越高。

高内聚，则说明对象内部各个元素全部专注于对象的核心职责。高内聚的对象，职责更单一，也更稳定。易于理解、使用，有更好的复用性，也更容易维护。

例1：

```
class HostingMessageHandler
{
    ...
    /**
     * 发送商城端虚拟主机购买邮件
     * @param BusinessEvent $event
     */
    public static function buyShop($event)
    {
        $business = $event->sender;

        $customerId = $business->getCustomerId();
        $busiduct   = $business->getBusiduct()->getData();
        $hostingModel= $business->getProduct();
        $hostingData = $hostingModel->getData();

        $period = $business->getParam('period');
        $keys = HostingShopexKeyModel::useKey($customerId, $hostingData->id,
                                              $period->getYears());
```

```
        if (CError::isError($keys)) {
            self::log('网店主机购买通知信发送失败：'.$keys->getFirstError());
            return;
        }

        $info = array();
        $info['USERID'] = $customerId;
        $info['BUSIDUCT'] = $busiduct->title;
        $info['WEBSPACE'] = $hostingData->MaxSize;
        $info['START_DATE'] = $hostingData->create_date;
        $info['EXPIRE_DATE'] = $hostingData->expire_date;
        $info['KEY'] = implode("<br />\n", $keys);
        $info['DATE'] = date('Y-m-d');

        return MessageHandler::post($customerId, 'SHOPEX_BUY', $info);
    }
    ...
}
```

`HostingMessageHandler`的核心职责是与主机业务相关的消息发送，`HostingShopexKeyModel::useKey()`方法所实现逻辑，显然已经超出`HostingMessageHandler`的职责范围，不应该在`buyShop()`方法中调用该方法。

例2：

CustomerModel是一个商城项目中的会员信息业务模型类，提供会员信息的相关核心业务逻辑。那么CustomerModel是否应该提供类似下面的检查账户余额、支付、扣款、退款等操作接口？

```
public function checkBalance($amount)
public function deduct($details)
public function refunds($details)
public function pay($orderItem)
```

```
class CustomerModel
{
    ...
    public function checkBalance($amount)
    {
        $customer = $this->getModel();
        $model = FundsAccount::getAccountById($customer->id);
        if ($model!= null && $model->status!=FundsAccount::STATUS_HOLD){
            if($model->getTotalBalance()>=$amount) {
                $this->account = $model;
                return true;
            }
        }

        return false;
    }
}
```

设计原则

一个类只表示一个关键抽象(单一职责)

一个类应该只表示一个关键抽象，这样的类，其职责明确，并且简单。这使类更易于理解、维护和使用，更重要的是能够简化类之间的关系，减少依赖。

> 该原则来自《OOD启思录》。

把相关的数据和行为集中放置

设计者应当留意那些通过get之类操作从别的对象中获取数据的对象。这种类型的行为暗示着这条经验原则被违反了。

> 以上内容摘自《OOD启思录》。

把不相关的信息放在另一个类中(也即：互不沟通的行为)

开发者应当留意这样的类：方法的一个子集操作数据成员的一个真子集。极端情况是，一个类有一半的方法操作一半数据成员，另一半方法则操作另一半数据成员。

> 以上内容摘自《OOD启思录》，《代码大全2》也同样有该原则的描述。

信息专家

某项职责应当由掌握完成该项职责所需数据的对象完成。

正确使用成员变量

成员变量

成员变量用于描述对象的属性特征。

成员变量的生命周期，与对象的生命周期是一致的。因此成员变量的生命周期长于成员方法中的局部变量。从类的范围来看，其具有全局变量的特性。

成员变量体现对象的特征、状态，因此在对象的整个生命周期中，成员变量都必须保持有意义的取值。并且通常来说成员变量，会被大部分的成员方法直接或间接使用。

什么时候使用成员变量

- 变量是不是能够体现对象的属性、状态特征。
- 是不是被大部分成员方法所使用。

- 变量是不是在对象生命周期内都是有意义，即任意的成员方法都可以随时访问该变量，并且获得有意义的取值。特别是在对象创建时，变量是否能够合理初始化(关于这点，在下面的构造函数中将详细说明)。特例：有时候，因为性能问题，需要使用延迟加载或缓存技术，对这些变量的访问必须通过特定的方法，否则无法保证取到有意义的取值。这类技术的使用，会增加变量管理的复杂度，因此应当有通用的设计解决方法，避免在多个类中单独实现这类技术。
- 避免将成员变量用来存放临时数据，因为这种情况下，变量是否有合理的取值，取决于某些成员方法是否被调用，这将给维护造成困难。
- 应该尽量缩短变量的生命周期，这一原则，同样适用于成员变量，如果不是必须的话，应该尽量避免将变量定义为成员变量。

设计原则

- 类中定义的大多数方法都应当在大多数时间里使用大多数数据成员。
- 类包含的对象数目不应当超过开发者短期记忆的容量。这个数目常常是6。
- 当类包含多于6个数据成员时，可以把逻辑相关的数据成员划分为一组，然后用一个新的包含类去包含这一组成员。

例1：

```
class REGISTRAR_TRANSFER
{
    ...
    var $type;
    var $domain;
    var $race_encode;
    var $customer_id;

    ...
    function REGISTRAR_TRANSFER()
    {
        $this->type        = "";
        $this->domain      = "";
        $this->race_encode = "";
        $this->customer_id = 0;
    }

    function setDomain($domain) {...}

    function setCustId($cust_id) {...}

    function trnDomain() {...}
    ...
}
```

这里的几个成员变量，在对象初始化时，均没有给出合理的取值，而是需要通过`setDomain()`等接口赋值。这意味着，`trnDomain()`等接口在执行之前，必须先调用`setDomain()`等接口对成员变量进行赋值，否则将无法正确执行。这极易造成误用，大大增加了使用难度。

例子2：

```
abstract class ClientBase
{
```

```
    /**
     * @var array 连接参数，对于各类通讯需要的参数集，具体情况查看子类
     */
    protected $options;
    /**
     * @var string 主机名，服务器IP或者域名
     */
    protected $server;
    /**
     * @var string 端口
     */
    protected $port;

    ...
    // 没有构造函数，也未有成员方法使用到这些变量
}
```

这里`ClientBase`被设计作为与远程服务器通讯的客户端基类，每个服务器都应当有地址、端口及连接参数信息，因此在基类中定义了相关的成员变量。这看起来似乎很合理。但实际上，在基类中，这些变量未被初始化，也未被使用。事实上，我们也很难保证派生类，会去使用这些成员变量。所以这些成员变量的定义是毫无意义的。

良好的类接口

接口的含义

所有暴露给用户的公有或保护成员。

接口的组成：命名、参数、职责。

设计友好的接口

友好的接口

- 简单
- 易于理解
- 易于使用
- 易于正确使用

总之应该尽可能让使用者通过函数名及其参数列表，就可以了解函数所要实现的功能及如何使用，而不必查看注释或手册。

类应当提供最小化并且足够丰富的接口

最小化并且足够丰富的接口，指类的接口应当尽可能少，并且同时通过这些接口或接口间的组合可以实现类所表示的抽象所具有的所有功能。

类的接口可以看成是类对于使用者的一种承诺或契约，我们都知道，做出承诺很容易，但想收回或修改承诺则很难。当一个类开始被广泛使用的时候，修改或撤销接口，将付出很大的代价。

> 以上内容主要参考自《Effective C++》第二版。

一个接口应该尽量只做一件事情(单一职责)

一个接口应该尽量只做一件事情。这样的接口，其实现代码，通常都比较简短，容易阅读和理解。同时简单单一的功能，能够减少变化带来的影响，并提高接口被复用的概率。

这里所说的一件事情，是一个相对的概念，相对于对象所处的抽象层次。比如：常见的MVC模型中，C层所处的抽象层次要高于M层所处的抽象层次，因此C层对象的接口，所做的一件事情可能要远比M层对象的接口所做的一件事情来得"多"。

例1：

```php
public function searchAudit($auditAction)
{
    ...
    if($auditAction == self::ACTION_SETTLED_AUDIT){
        ...
    }
    elseif($auditAction == self::ACTION_UNSETTLED_AUDIT){
        ...
    }
    ...
}
```

这个例子根据传递进来的`$auditAction`参数的不同取值，分成两个不同的逻辑分支进行处理。当两个分支所处理的逻辑，并无太紧密的联系时，它们实际上在做两件不同的事情，应该拆分成两个不同的接口。

确保接口的实现与接口承诺的一致

对象应当确保接口的实现与接口承诺的一致。接口实现的功能，应该与其声明一致。不同的功能，即使内部实现完全一致，也应当使用不同的接口，而不是同一个接口(一致的实现，在未来可能变得不一致)。

对象的使用者，同样应当遵守接口的承诺，而不应当窥探接口的实现。

例1：

```php
class ServiceCategory extends ActiveRecord
{
    /**
     * 通过通过外部ID获取服务分类的ID
     * @param integer $externalId 外部id
     * @return integer 返回服务分类ID
     */
    public static function getIdByExternalId($externalId) {
        $query = static::find();
        $query->select('id');
        $query->where(['external_id' => $externalId]);
        return $query->one();
```

```
    }
    ...
}
```

这个例子中，接口 `getIdByExternalId()` 所返回的值与所承诺的不一致。应重构为：

```
class ServiceCategory extends ActiveRecord
{
    /**
     * 通过通过外部ID获取服务分类的ID
     * @param integer $externalId 外部id
     * @return integer 返回服务分类ID
     */
    public static function getIdByExternalId($externalId) {
        $query = static::find();
        $query->select('id');
        $query->where(['external_id' => $externalId]);
        $res = $query->one();

        return ($res)? $res->id : 0;
    }
    ...
}
```

接口只包含最少的必要的参数

1、提供最少的必要的参数：

减少使用者的负担

减少依赖

增加被复用的机会。

2、参数命名应该能够表达参数的意义。

3、越重要的参数，放在前面。

例1：

```
/**
 * 激活用户
 * @param integer $id 用户id
 * @param sting $authcode 随机密匙和customer_activate中的authcode对应
 */
public function activate($id,$authcode)
{
    $expire_time = $this->getModel()->activation->expire_time;
    if(strtotime($expire_time) < time() || $authcode != $this->getModel()->activation->authcode){
        return $this->setError('链接已经失效，请重新发送激活邮件！');
    }
    //更改会员状态
    $this->getModel()->status = Customer::STATUS_OK;
    if(!$this->getModel()->save()){
```

```
        return $this->addErrors($this->getModel()->getErrors());
    }
    //删除激活表中数据
    CustomerActivation::model()->deleteAll('id=:id',array(':id'=>$id));
    return $this->getModel();
}
```

这个例子中，`activate()`接口是由用户对象提供的接口，对象本身是持有用户的所有信息的，当然也包括用户ID，因此这里的`$id`参数就属于不必要的参数。增加这样不必要的参数，一方面增加了使用者的使用难度，另外一方面，也可能造成逻辑错误。想象一下，如果使用者传递了一个与当前用户对象不同的用户ID，会造成什么结果。

例2：

```
class CustomerModel
{
    /**
     * 修改邮箱
     */
    public function editEmail(CustomerForm $form)
    {
        $customer = $this->getModel();
        $customer->email = $form->email;
        if(!$customer->save()){
            return $this->addErrors($customer->getErrors());
        }
        return $customer;
    }
}
```

这个例子中`editEmail()`接口中的`$form`参数，已经远远超出接口所需要，接口只需要新的邮箱地址，而不需要一个完整的`CustomerForm`对象。事实上`CustomerForm`对象高于`CustomerModel`对象所属的层次，根据层次间的单向依赖原则，`CustomerForm`对象也不应该出现在`CustomerModel`的接口参数中。

```
class CustomerModel
{
    /**
     * 修改邮箱
     */
    public function editEmail(string $email)
    {
        $customer = $this->getModel();
        $customer->email = $email;
        if(!$customer->save()){
            return $this->addErrors($customer->getErrors());
        }
        return $customer;
    }
}
```

修改后的接口，不仅使用上更简单方便，也减少了不必要的依赖，避免了因为展示层的表单变更而导致接口变更。

例3：

```
@Injectable()
export class ItemService extends Service {
    ...
    getDetail(params) {
        return this.http.post('/base/service/item/detail', params);
    }
}

// getDetail()接口的使用
this.service.getDetail({ id: this.itemId }).subscribe((data) => {
    ...
})
```

这个例子中，`getDetail()`接口实际上只需要指定想要获取的item的id即可，并且在可预见的未来，也不太可能需要根据额外的参数来获取item信息。因此将参数设置为对象，无疑增加了接口的使用难度。改进后的代码：

```
@Injectable()
export class ItemService extends Service {
    ...

    getDetail(id: number) {
         return this.http.post('/base/service/item/detail', {id:id});
    }
}

// getDetail()接口的使用
this.service.getDetail(this.itemId).subscribe((data) => {
    ...
})
```

简单易于使用的接口

接口及其参数的设置和命名，应该具有可读性。

例1：

```
void printText( const char *msg, bool sendNewLine);

int main(int argc, char *argv[])
{
    printText("Hello, world", true);

    return 0;
}
```

改进后的代码：

```
enum NewLineDisposition { sendNewLine, noNewLine };
void printText(const char *msg, NewLineDisposition format);

int main(int argc, char *argv[])
{
    printText("Hello, world", NewLineDisposition::sendNewLine);

    return 0;
}
```

保持接口一致性

很少有其它性质比得上“一致性”更能导致“接口容易被正确使用”，也很少有其它性质比得上“不一致性”更加剧接口的恶化。

保持接口的一致性，包括但不限于：

- 相同含义的接口，在命名上应该保持一致。比如，C++ STL容器的接口十分一致，每个STL容器都有一个名为size的成员函数，它会告诉调用者，目前容器内有多少对象。相反的，Java允许你针对数组使用length property，对Strings使用length method，而对Lists使用size method。一致性的接口，能大大减少开发人员的心智负担，使得接口能够更容易被正确使用。
- 同一个类的接口，应该展现一致的抽象层次。

列1：

```
class CustomerModel
{
    /**
     * 获取资金账户模型对象
     * @return FundsAccountModel
     */
    public function getAccountModel()
    {
        if(!isset($this->account) || $this->account == null)
            return FundsAccountModel::createById($this->getModel()->id);
        return $this->account;
    }

    /**
     * 获取积分
     * @return IntegralAccount
     */
    public function getIntegralAccount() {
        if($this->integralAccount == null) {
            $customer = $this->getModel();
            $model = IntegralAccount::getAccountById($customer->id);
            $this->integralAccount = $model;
        }

        return $this->integralAccount;
    }
    ...
}
```

这个例子中，`CustomerModel`是一个客户业务模型类，它提供了两个接口`getAccountModel()`和`getIntegralAccount()`分别用来获取客户对应的资金账户模型对象和积分账户模型对象。但`getAccountModel()`返回的是Model对象，`getIntegralAccount()`则返回更底层的ActiveRecord对象。两个接口展现的抽象层次不一致。

良好的封装

区分接口与实现

这里的接口指的是所有暴露给外部的成员变量和成员方法。区分接口与实现，即区分类中哪些部分属于接口，哪些部分属于内部实现。内部实现是使用者不需要关心的，应该被隐藏。在前面已经介绍过，信息隐藏是封装的重要特性之一，隐藏实现细节，不仅可以简化类的使用，更重要的是能够应对可能的变化，只要保持接口不变，内部实现可以任意修改，而不影响使用者。

区分接口与实现是良好封装的第一步。

尽可能地限制类和成员的可访问性

让可访问性尽可能低是促成封装的原则之一(即信息隐藏，可访问性越低，隐藏的越多)。

如何设置成员的可访问性，建议按下面的原则：

- 采用最严格且可行的访问级别。
- 考虑哪种可访问性，能最好地保护接口抽象的完整性，如果不确定，那么多隐藏通常比少隐藏要好。

> 以上内容摘自《代码大全2》。

所有数据都应该隐藏在类的内部

所有数据都应该隐藏在所在的类的内部，避免public接口出现数据成员。首先，从“一致性”的角度来看这个问题。如果public接口里都是函数，用户每次访问类的成员时就用不着抓脑袋去想：是该用括号还是不该用括号呢？——用括号就是了！因为每个成员都是函数。一生中，这可以避免你多少次抓脑袋啊！

你不买“一致性”的帐？那你总得承认采用函数可以更精确地控制数据成员的访问权这一事实吧？如果使数据成员为public，每个人都可以对它读写；如果用函数来获取或设定它的值，就可以实现禁止访问、只读访问和读写访问等多种控制。甚至，如果你愿意，还可以实现只写访问：

```
class AccessLevels {
public:
  int getReadonly() const{ return readonly; }
  void setReadwrite(int value) { readwrite = value; }
  int getReadwrite() const { return readwrite; }
  void setWriteonly(int value) { writeonly = value; }
private:
  int noaccess;             // 禁止访问这个int
  int readonly;             // 可以只读这个int
  int readwrite;            // 可以读/写这个int
  int writeonly;            // 可以只写这个int
};
```

还没说服你？那只得搬出这门重型大炮：功能分离（functional abstraction）。如果用函数来实现对数据成员的访问，以后就有可能用一段计算来取代这个数据成员，而使用这个类的用户却一无所知。

例如，假设写一个用自动化仪器检测汽车行驶速度的应用程序。每辆车行驶过来时，计算出的速度值添加到一个集中了当前所有的汽车速度数据的集合里：

```
class SpeedDataCollection {
public:
  void addValue(int speed);       // 添加新速度值
  double averageSoFar() const;    // 返回平均速度
};
```

现在考虑怎么实现成员函数averageSoFar。一种方法是用类的一个数据成员来保存当前收集到的所有速度数据的运行平均值。只要averageSoFar被调用，就返回这个数据成员的值。另一个不同的方法则是在averagesofar每次被调用时才通过检查集合中的所有的数据值计算出结果。

第一种方法——保持一个运行值——使得每个SpeedDataCollection对象更大，因为必须为保存运行值的数据成员分配空间。但averageSoFar实现起来很高效：它可以是一个仅用返回数据成员值的内联函数。相反，每次调用时都要计算平均值的方案则使得averageSoFar运行更慢，但每个SpeedDataCollection对象会更小。

谁能说哪个方法更好？在内存很紧张的机器里，或在不是频繁需要平均值的应用程序里，每次计算平均值是个好方案。在频繁需要平均值的应用程序里，速度是最根本的，内存不是主要问题，保持一个运行值的方法更可取。重要之处在于，用成员函数来访问平均值，就可以使用任何一种方法，它具有极大价值的灵活性，这是那个在public接口里包含平均值数据成员的方案所不具有的。

所以，结论是，在public接口里放上数据成员无异于自找麻烦，所以要把数据成员安全地隐藏在与功能分离的高墙后。如果现在就开始这么做，那我们就可以无需任何代价地换来一致性和精确的访问控制。

> 以上内容摘自《Effective C++》第二版。

避免把私用的实现细节放入类的公有接口中

不要把实现细节，放入类的公有接口中，包括但不限于以下情况：

- 将放置共用代码的私有函数放到公有接口中。
- 将用户不应该关心的实现细节，放入到接口参数中。

例1：

主机接口开设：

```
function getFeature($item_id) {
    /*
       DomainBindings = 0x1,     域名绑定          1
       DefaultDocs = 0x2,        默认首页设置      2
       HttpRedirects = 0x8,      重定向        8
       CustomErrors = 0x10,      自定义错误        16
       MimeMaps = 0x20,          MIME类型          32
       IPSecurity = 0x40,        IP地址访问限制    64
       FileProtect = 0x100,      文件保护          256
       FileManager = 0x200,      文件管理          512
       DatabaseManager = 0x400,  数据库管理        1024
       Analyzer = 0x800,         访问统计          2048
       HostMail = 0x1000,        企业邮局          4096
       UserControls = 0x2000,    组件上传          8192
       SubDomain = 0x4000,      子站点管理         16384
       BackupAgent = 0x8000,     网站备份管理      32768
       页面加密为FileEncrypt= 0x20000 页面加密 131072
     */
    ...
    $base_feature = 3963;// 企业邮局以上所有功能的总和
    $base_feature += 131072; //加入页面加密，by liuwei
    $features = $base_feature;
    ...
    return $features;
}
$nCPFeatures = getFeature($host_type);
$h_hosting->CreateHost(..., $nCPFeatures, ...);


$hostingFeature = new HostingFeature($features);
$features = $hostingFeature->getFeatureValue();
$roHosting = RoHosting::create($this->name, $hostIP, $this->password,
$domains, $maxSize, $maxBandwidth, $maxConnections, $exDate, $features,
$logSize);
```

> 该原则来自《OOD启思录》。

不要以用户无法使用或不感兴趣的东西扰乱类的公有接口

这条经验原则与前一条“避免把私用的实现细节放入类的公有接口中”是相关的，因为类的用户不会想调用放置公用代码的函数，或者其它任何实现细节。所以把这些实现细节放入类的公有接口中，只会扰乱类的公有接口。

> 该原则来自《OOD启思录》。

不要对类的使用者做出任何假设

类的设计和实现应该符合在类的接口中所隐含的契约。它不应该对接口会被如何使用或不会被如何使用做出任何假设，除非在接口中有过明确说明。像下面这样一段注释就显示出这个类过多地假定了它的使用者：

```
// 请把x，y和z初始化为1.0，因为如果把它们初始化为0.0的话，DerivedClass就会崩溃！
```

> 以上内容摘自《代码大全2》。

依赖于接口而不依赖于实现

良好的抽象

类的接口应该展现一致的抽象层次

接口的抽象层次，应该保持与所属对象的抽象层次完全一致。同一个类，提供不同抽象层次的接口，意味着不同抽象层次的混杂。这种混杂，不仅增加正确使用接口的难度，而且也破坏了对象的内聚性，导致程序难以理解和维护。

不要添加与接口抽象不一致的公共成员

每次你向类的接口中添加新的方法时，问问“这个方法与现有接口所提供的抽象一致吗？”如果发现不一致，就要换另一种方法来进行修改，以便能够保持抽象的完整性。

同时考虑抽象性和内聚性

抽象性和内聚性这两个概念之间的关系非常紧密。一个呈现出很好的抽象的类接口通常也有很高的内聚性。而具有很强内聚性的类往往也会呈现出很好的抽象，尽管这种关系并不如前者那么强。

关注类的接口所表现出来的抽象，比关注类的内聚性更有助于深入理解类的设计。如果你发现某个类的内聚性很弱，也不知道该怎么改，那就换一种方法，问问你自己这个类是否表现为一致的抽象。

> 以上内容摘自《代码大全2》。

确定对象间的关系

耦合与解耦

前面已经介绍过对象间的关系，包括：依赖、关联、聚合、组合、泛化，这些关系的耦合度依次增强。确定对象之间的关系，必须考虑尽量降低对象之间的耦合度，即解耦。关于解耦的具体原则已在前面介绍过，此处不再赘述。

设计原则

类之间应该零耦合，或者只有导出耦合关系

类之间应该零耦合，或者只有导出耦合关系。也即，一个类要么同另一个类毫无关系，要么只使用另一个类的公有接口中的操作。

> 以上内容摘自《OOD启思录》。

使用构造与析构

构造函数

大部分面向对象语言，都提供了构造函数。构造函数用于确保在实例化对象时，能够正确的初始化对象。构造函数在对象被实例化的时候，执行(对象实例化时，通常需要先分配对象所需内存，执行初始化序列，之后再调用构造函数，完成初始化工作)。

构造函数由编译器或解析器自动调用(取决于具体使用的语言)，保证了构造函数始终能够被正确执行。

构造函数的引入，一开始是为了解决面向过程语言中，由于程序员疏忽导致的数据未初始化的问题。在传统的面向过程语言中(如C)，数据未初始化，导致了许多软件问题。特别是在C、C++等语言中，如果变量未初始化，则其状态是不确定的。未初始化的变量，其取值将取决于变量所占据的内存，上次使用后遗留下来的信息。这通常会导致内存错误，使程序崩溃。更糟糕的情况则是，这个不确定的取值，刚好是合法的，程序正常运行，但结果却是错误的。

在类似PHP这样的语言中，由于语言本身的特点，变量未初始化导致的问题跟C、C++这类语言会有很大的不同。在PHP中，未初始化的变量，其取值为null，这比C、C++中的不确定取值要来得好。但是如果PHP的register_globals选项开启时，则情况将可能变得很糟糕。如果未初始化的是全局变量，其取值将可能被客户篡改，进而导致安全问题。因此适当的初始化变量，仍然是一个好习惯。

PHP语言允许在类定义中，直接初始化成员变量，这使得不使用构造函数，也能确保对象正确初始化。但对于需要从外界获取参数，来完成初始化，或者初始化操作比较复杂的情况，仍然需要构造函数。

析构函数

大部分面向对象语言，都提供了析构函数(并非所有，比如Java就没有析构函数)。析构函数用于在对象生命周期结束后，做一些清理工作，比如释放对象持有的资源(内存资源、文件句柄、数据库连接等)等。

析构函数一般在对象生命周期结束后，由编译器或解析器自动调用执行。

关于对象生命周期，不同语言，会有些微的区别：

1、C++中，栈上创建的对象，其生命周期在离对象定义最近的一个 } 结束。堆上创建的对象，其生命周期在对象被delete时结束。静态对象，其生命周期则贯穿整个进程的生命周期。

2、PHP中，对象的所有引用都被删除或者对象被显示销毁时，对象生命周期结束。

构造与析构的其它讨论

何时实例化对象

一般来说应该尽可能推迟对象的实例化，直到真正需要这个对象的时候。原因如下：

(1)对于有构造参数的对象，往往只有在真正需要这个对象的时候，才能获取足够的信息，来完成对象的初始化。

(2)对象实例化时，将产生相应的开销，包括分配对象所需内存的开销及构造函数运行的开销(取决于具体的实现语言)。过早实例化对象可能导致：一是由于没有足够信息初始化对象，因此只能使用默认构造函数来构造对象，而在真正使用到该对象时，往往需要对对象再次赋值，造成额外开销；二是对象实际仅在某个条件分支中使用，对于其它条件分支，对象实例化造成的开销实际上是无效开销。

(3)对象实例化的代码与使用对象代码尽可能接近，有利于代码阅读和理解。

关于第(2)点的开销问题，可能这种开销大部分情况下对于效率的影响，微乎其微，但在某些情况下，却可能对效率产生很大影响，特别是复杂对象(这种对象往往持有大量资源或者初始化开销很大)或者被频繁使用的对象。而避免这种开销，实际上很简单。

构造及成员变量的设置

一般来说，我们希望在实例化后，对象处于就绪状态，可以立即提供服务，而不需要额外的工作。就好像我们总是希望新招聘的员工，已经掌握基本工作技能，入职后能够立即投入工作，而无需更多的培训工作。

而对象的这种就绪状态，要求表示对象的某一状态或者特征的成员变量，必须已经合理的初始化(有意义的取值)。因此在对象实例化时，成员变量应该总是能够被合理的初始化。

如果某个成员变量，在对象实例化时，无法被合理的初始化，则这个变量可能并不适合作为成员变量。

下面的实例中，成员变量的设置不太合理：

```
class REGISTRAR_TRANSFER
{
    ...
    var $type;
    var $domain;
    var $race_encode;
    var $customer_id;
    ...
    function REGISTRAR_TRANSFER()
    {
        $this->type        = "";
        $this->domain      = "";
        $this->race_encode = "";
        $this->customer_id = 0;
    }
    function trnDomain() {...}
    ...
}
```

使用对象来管理资源

构造和析构函数由编译器或解析器自动调用并执行，这一特性，使得我们可以使用对象来管理资源，在构造函数中初始化资源，在析构函数中释放。这可以避免由于程序员的疏忽而导致的资源未初始化就被使用，或者资源没有正确释放而泄露的情况。

包括内存、文件句柄、网络连接、数据库连接等等资源，均可以使用对象来管理。

以文件处理为例，使用fopen()打开文件，并使用完后，需要关闭文件。不使用对象的情况下，为了确保做到这一点，我们需要像下面的代码那样，在每一个return之前调用fclose()来关闭文件：

```php
function f() {
    $fp = fopen('test.txt');
    if (!$fp) {
        return false;
    }

    ...
    if() {
        fclose($fp);
        return false;
    }
    ...
    if() {
        fclose($fp);
        return false;
    }
    ...

    fclose($fp);
    return true;
}
```

如果使用对象来管理文件资源，则代码如下：

```php
class File {
    public function __construct($file) {
        $this->fp = fopen($file);
        if (!$this->fp) {
            throw new Exception('Can not open file.');
        }
    }

    public function __destruct() {
        if ($this->fp) {
            fclose($this->fp);
        }
    }
    ...
    protected $fp;
}

function f() {
    $file = new File('test.txt');
    ...
    if() {
        return false;
    }
    ...
    if() {
        return false;
    }
    ...
```

```
29
30      return true;
31  }
```

可以发现使用File类来操作文件，文件始终能够正确关闭，而无须手工调用方法来关闭文件。这样将不用担心由于疏忽，忘记在某个return之前调用fclose()，而导致文件没有正确关闭。

当使用异常时，这一特性将更加有用。关于异常，将在下面的章节中详细介绍。

使用对象来完成某些需要自动完成的操作

同样可以使用构造和析构函数来完成，那些需要在对象生命周期开始和结束时，自动完成的操作。比如事务自动回滚。

使用继承与多态

设计原则

替换原则

替换原则，即派生类对象可以替换基类对象。

使用组合代替继承

除非两个类，明确体现出“is-a”的关系，否则不应该使用继承，而应该尽量使用组合来代替继承。

复用派生类而非复用基类

继承机制使得派生类可以很容易的就能够复用基类的方法。但是因为继承本身是有代价的，单纯的使用继承来复用基类，并不是好的做法，此种场景下应当尽量使用组合而非继承。

继承机制最有价值的应用，应当是与多态结合，派生类通过重写基类的方法，实现对基类的扩展。此种场景下，使用者直接使用的是基类接口，并不需要知道派生类的存在。基类可以复用派生类，来扩展自身的能力。

使用异常处理机制

异常处理机制，并不属于面向对象概念范畴里面。但由于异常处理机制，被大部分的面向对象语言支持，并且在面向对象编程中有着广泛并且重要的应用，因此我们将异常处理机制作为一部分来讲述。

需要注意的是，不同编程语言对于异常处理机制的支持及实现，并不完全一致。因此以下讲述的内容只针对一般情况。

基本概念

异常指程序运行中，所发生的会中断程序正常运行的错误。比如除零、溢出、数组下标越界等。异常抛出后，正常的程序处理过程即中断，直到异常被捕获。需要注意的是，是否将错误作为异常来处理，取决于具体语言和设计需要。

异常处理机制是一种错误处理机制。引入异常处理机制的意图在于，实现错误处理逻辑与正常逻辑的分离，以解决传统错误处理方式的不足。

以下代码，展示了传统的错误处理方式的特点：正常的逻辑代码与错误处理代码是混合在一起的。

```
$sql1 = ...;
if (getFirst($row, $sql1, $db_conn)) {
    ...
    return false;
}
...
$sql2 = ...;
if (getFirst($row, $sql2, $db_conn)) {
    ...
    return false;
}
...
$sql3 = ...;
if (getFirst($row, $sql3, $db_conn)) {
    ...
    return false;
}
...
```

如果使用异常，则上面的代码，可变得更为简洁，如下：

```
try {
    ...
    $sql1 = ...;
    getFirst($row, $sql1, $db_conn);

    ...
    $sql2 = ...;
    getFirst($row, $sql2, $db_conn);

    ...
    $sql3 = ...;
    getFirst($row, $sql3, $db_conn);
    ...
}
catch (Exception $e) {
    ...// 错误处理
}
```

异常处理机制的优点

相对于传统的错误处理方式，异常处理机制具有如下优点：

1. 异常处理机制将正常处理代码与错误处理代码相互分离，使程序结构更加清晰。同时也有利于异常错误的统一处理。
2. 异常处理机制将内层错误的处理直接转移到适当的外层来处理，简化了处理流程。传统的错误处理方式是通过一层层返回错误码把错误处理转移到上层，上层再转移到上上层，当层数很多时将需要非常多的判断。

如下代码展示了传统错误处理方式，错误的传递方式：

```
function f() {
    ...
    if (!g()) {
        return false;
    }
    ...
}

function g() {
    ...
    if (!h()) {
        return false;
    }
    ...
}

function h() {
    ...
    return false;
}
```

3. 局部出现异常时，在执行处理代码之前，会执行栈回退，即为所有局部对象调用析构函数，保证局部对象行为良好(注：并非所有语言都提供析构函数)。
4. 异常不能被忽略，如果异常没有被捕获，程序会立即退出。如果一个函数通过设置一个状态变量或返回错误代码来表示一个异常状态，没有办法保证函数调用者一定会检测变量或测试错误代码(而这种情况一旦发生，程序将忽略异常状态，继续执行，这将导致未知错误)。通过语言本身的机制或应用框架，通常能够确保异常错误，始终能够被处理，即使程序没有明确的对异常进行捕获处理。相对于传统的错误处理方式，可以避免由于程序员疏忽，未判断错误返回或未测试错误代码而导致的未知错误。
5. 可以在处理块中尝试错误恢复，保证程序几乎不会崩溃。通过适当处理，即使出现除0异常，内存访问违例等致命性错误，也能让程序不崩溃，继续运行，这种能力在某些情况下及其重要(根据具体的语言和使用的系统，像除0异常和内存越界访问这样的致命错误，不一定会抛出异常)。
6. 可以将任意对象作为异常抛出(取决于具体语言，如PHP要求异常对象必须是Exception及其派生类的实例，而C++则允许任意对象类型)，因此可对异常对象进行封装处理以更好的对异常错误进行分类处理，并携带所需的错误信息。有些语言(如PHP、Java)能够在异常对象中提供路径信息，利于跟踪调试。

异常处理机制的代价及缺点

相对于传统的错误处理方式，异常机制有着很多优势，但也需要付出更多的代价，同时也有着自己的缺点。

1. 异常机制的实现需要额外的代价，包括空间代价和时间代价，这取决于使用的编程语言。异常处理与函数调用类似，但代价更高。同时对于编译型语言，为了在异常抛出后，能够正确到达异常处理，需要额外的空间用于记录抛出点与捕获点之间的路径信息。

2. 由于捕获并处理异常的点，往往远离异常抛出点，异常处理程序无法获取足够的错误现场信息，因此一般情况下，异常处理无法使程序恢复到错误前的状态。

何时使用异常

异常处理机制并非适用于所有的错误处理，根据异常处理机制的特点，以下情况适合使用异常处理：

1. 方法A需要返回错误，并且A被大量调用，并且我们希望使用同样的方式来处理这个错误时。这种情况需要权衡付出的代价。
2. 方法A被使用的层次很深时，使用异常可以立即从错误点上跳出，而不必一层一层返回。
3. 错误发生时，程序已经无法恢复正常。如果程序需要根据错误返回，进行相应处理，则不适合使用异常。
4. 由于使用异常的代价比较高，通常情况下，应该只对灾难性的错误(正常情况下不应该发生的错误)，使用异常处理。

上面所列情况，并非完全绝对。比如在C++ Unit中，就使用异常来报告用例失败。但在正常情况下，不应该将异常处理作为唯一的错误处理方式。

如果我们能遵循只对灾难性错误使用异常处理，那么异常发生的概率应该是很低的，这种情况下，对于类似C++这样的编译型语言，使用异常所带来的运行效率开销基本可以忽略不计，主要的开销是空间上的开销。

其它设计原则

类的使用者必须依赖类的共有接口，但类不能依赖它的使用者

这条经验原则背后的基本原理是可复用性。闹钟可以用于卧室。使用闹钟的人显然依赖于闹钟的公有界面。但是，闹钟不应当依赖于那个人。如果闹钟依赖于使用的人，那么闹钟就无法被用来制造定时锁保险箱，除非把使用的人也绑定在保险箱上。这样的依赖性是不受欢迎的，因为我们想要把闹钟用于其它的领域，而不想为此依赖于使用者。所以，最好把闹钟看作一个小型机器，这个小型机器对它的使用者一无所知，它仅仅是执行定义于公有界面的行为，而不管发送消息的是谁。

> 以上内容摘自《OOD启思录》。

避免创建全能类

- 在你的系统中不要创建全能类/对象。对名字包含Driver、Manager、System、Susystem的类要特别多加小心。
- 对公共接口中定义了大量访问方法的类多加小心。大量访问方法意味着相关数据和行为没有集中存放。
- 对包含太多互不沟通的行为的类多加小心。互不沟通的行为是指在的数据成员的一个真子集上进行操作的方法。全能类经常有很多互不沟通的行为。

这个问题的另一表现是在你的应用程序中的类的公有接口中创建了很多的get和set函数。

> 以上内容摘自《OOD启思录》。

尽可能地按照现实世界建模

我们常常为了遵守系统功能分布原则、避免全能类原则以及集中放置相关数据和行为的原则而违背这条原则。

从你的设计中去除不需要的类

一般来说，我们会把这个类降级成一个属性。

尽量减少类的协作者的数量

一个类用到的其他类的数目应当尽量少。

尽量减少类和协作者之间传递的消息的数量

如果类包含另一个类的对象，那么包含类应当给被包含的对象发送消息

即：包含关系总是意味着使用关系。

类必须知道它包含什么，但是不能知道谁包含它

不要把全局数据或全局函数用于类的对象的薄记工作

不要把全局数据或全局函数用于类的对象的薄记工作，应当使用类变量或类方法。

不要绕开公共接口去修改对象的状态