编程泛型 - (Programming paradigm)

编程范型 又称 **编程范式**、**编程典范** 或 **程序设计法**，是指软件工程中的一类典型的编程风格。

常见的编程泛型有:

- **函数式编程**
- **指令式编程**
- **过程式编程**
- **面向对象编程**

等等...

> 正如软件工程中不同的群体会提倡不同的“方法学”一样，不同的编程语言也会提倡不同的“编程范型”。一些语言是专门为某个特定的范型设计的，如Smalltalk和Java支持面向对象编程，而Haskell和Scheme则支持函数式编程，同时还有另一些语言支持多种范型，如Ruby、Common Lisp、Python、Rust。
>
> 很多编程范型已经被熟知他们禁止使用哪些技术，同时允许使用哪些。例如，纯函数式编程不允许有副作用；大部分高端程序语言都期望用户进行结构化编程避免非结构化编程，结构化编程不允许使用goto。
>
> 编程范型和编程语言之间的关系可能十分复杂，由于一个编程语言可以支持多种范型。例如，C++设计时，支持过程式编程、面向对象编程以及泛型编程。然而，设计师和程序员们要考虑如何使用这些范型元素来构建一个程序。一个人可以用C++写出一个完全过程化的程序，另一个人也可以用C++写出一个纯粹的面向对象程序，甚至还有人可以写出杂揉了两种范型的程序。

请区别于 "泛型编程" 这个概念！它们是不同的两个概念！

OO-三大基本特性

封装 - (Encapsulation)

> 在面向对象编程方法中，封装 是指，一种将抽象性函数接口的实现细节部分包装、隐藏起来的方法。同时，它也是一种防止外界调用端，去访问对象内部实现细节的手段，这个手段是由编程语言本身来提供的。
>
> 适当的封装，可以将对象使用接口的程序实现部分隐藏起来，不让用户看到，同时确保用户无法任意更改对象内部的重要资料，若想接触资料只能通过公开接入方法（Publicly accessible methods）的方式（如："getters" 和"setters"）。它可以让代码更容易理解与维护，也加强了代码的安全性。

封装，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。一个类就是一个封装了数据以及操作这些数据的代码的逻辑实体。在一个对象内部，某些代码或某些数据可以是私有的，不能被外界访问。通过这种方式，对象对内部数据提供了不同级别的保护，以防止程序中无关的部分意外的改变或错误的使用了对象的私有部分。

信息隐藏

> 封装可以隐藏成员变量以及成员函数，对象的内部实现通常被隐藏，并用定义代替。举个例子，仅仅对象自身的方法能够直接接触或者操作这些成员变量。隐藏对象内部信息能供保证一致性，当用户擅自修改内部部件的数据，这可能造成内部状态不一致或者不可用。隐藏对象内部信息能阻止这种后果。一个众所周知的好处是，降低系统的复杂度和提高健壮性。
>
> 大多数语言（如：C++、C#、 Delphi、Java）通过设定等级去控制内部信息隐藏，经典的是通过保留字public 暴露信息和 private去隐藏信息。一些语言（如： Smalltalk 和 Ruby ）只允许对象去访问隐藏信息。
>
> 通常，也是存在方法去暴露隐藏信息，如：通过反射(Ruby、Java、C#、etc.)，名字修饰(Python)

继承 - (Inheritance)

> 继承 是面向对象软件技术当中的一个概念。继承可以使得子类具有父类别的各种属性和方法，而不需要再次编写相同的代码。在令子类别继承父类别的同时，可以重新定义某些属性，并重写某些方法，即覆盖父类别的原有属性和方法，使其获得与父类别不同的功能。另外，为子类追加新的属性和方法也是常见的做法。 一般静态的面向对象编程语言，继承属于静态的，意即在子类的行为在编译期就已经决定，无法在运行期扩展。
>
> 有些编程语言支持多重继承，即一个子类可以同时有多个父类，比如C++编程语言；而在有些编程语言中，一个子类只能继承自一个父类，比如Java编程语言，这时可以透过实现接口来实现与多重继承相似的效果。

继承，指可以让某个类型的对象获得另一个类型的对象的属性的方法。它支持按级分类的概念。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。继承概念的实现方式有二类：实现继承与接口继承。实现继承是指直接使用基类的属性和方法而无需额外编码的能力；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。

> 现今面向对象程序设计技巧中，继承并非以继承类别的“行为”为主，而是继承类别的“类型”，使得组件的类型一致。另外在设计模式中提到一个守则，“多用合成，少用继承”，此守则也是用来处理继承无法在运行期动态扩展行为的遗憾。

多态 - (Polymorphism)

在编程语言和类型论中，多态 指为不同数据类型的实体提供统一的接口，或使用一个单一的符号来表示多个不同的类型。

多态，是指一个类实例的相同方法在不同情形有不同表现形式。多态机制使具有不同内部结构的对象可以共享相同的外部接口。这意味着，虽然针对不同对象的具体操作不同，但通过一个公共的类，它们（那些操作）可以通过相同的方式予以调用。

多态的最常见主要类别有：

- 特设多态：为个体的特定类型的任意集合定义一个共同接口。
- 参数多态：指定一个或多个类型不靠名字而是靠可以标识任何类型的抽象符号。
- 子类型（也叫做子类型多态或包含多态）：一个名字指称很多不同的类的实例，这些类有某个共同的超类。

依据实现时做出的选择，多态可分为：

- 动态多态（dynamic polymorphism）:生效于运行期。

- 静态多态（static polymorphism）：将不同的特殊行为和单个泛化记号相关联，由于这种关联处理于编译期而非运行期，因此被称为“静态”。可以用来实现类型安全、运行高效的同质对象集合操作。C++ STL不采用动态多态来实现就是个例子。

对于C++语言，带变量的宏和函数重载机制也允许将不同的特殊行为和单个泛化记号相关联。然而，习惯上并不将这种函数多态、宏多态展现出来的行为称为多态（或静态多态），否则就连C语言也具有宏多态了。(对于C++ :)谈及多态时，默认就是指动态多态，而静态多态则是指基于模板的多态。

> 关于“动态多态”与“静态多态”的笔记，将转至C++相关内容中 `->` File

子类型 - (我们常说的多态属于子类型多态)

概念：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。 简单的说:就是用基类的引用指向子类的对象。

在面向对象程序设计中，计算机程序运行时，相同的消息可能会送给多个不同的类别之对象，而系统可依据对象所属类别，引发对应类别的方法，而有不同的行为。简单来说，所谓多态意指相同的消息给予不同的对象会引发不同的动作。比如有动物之类别，而且由动物继承出类别猫和类别狗，并对同一源自类别动物（父类）之一消息有不同的响应，如类别动物有“叫”之动作，而类别猫会“喵喵”，类别狗则会“汪汪”，则称之为多态态。

在下面的这个例子中猫和狗都是动物的子类型。过程`letsHear()`接受一个动物，但在传递给它一个子类型的时候也能正确工作：

```
abstract class Animal {
    abstract String talk();
}

class Cat extends Animal {
    String talk() {
        return "Meow!";
    }
}

class Dog extends Animal {
    String talk() {
        return "Woof!";
    }
}

static void letsHear(final Animal a) {
    println(a.talk());
}

static void main(String[] args) {
    letsHear(new Cat());
    letsHear(new Dog());
}
```

特设多态

术语“**特设多态**”来指称一个多态函数可以应用于有不同类型的实际参数上，但是以来它们所应用到的实际参数类型而有不同的表现（也叫做为**函数重载**或**运算符重载**）。在这个上下文中术语“特设”（ad hoc）不意图表达贬义，它只是简单的指出这种多态不是类型系统的基本特征。在下面的Pascal/Delphi例子中，在查看`Add`函数的调用的时候，它好像通用的工作在各种类型之上，但编译器对所有意图和用途都把它们视为完全不同的两个函数:

```
program Adhoc;

function Add(x, y : Integer) : Integer;
begin
    Add := x + y
end;

function Add(s, t : String) : String;
begin
    Add := Concat(s, t)
end;

begin
    Writeln(Add(1, 2));                    (* 打印"3"              *)
    Writeln(Add('Hello, ', 'Mammals!'));    (* 打印"Hello, Mammals!" *)
end.
```

在动态类型语言中情况可能更加复杂，因为需要调用的正确函数只能在运行时间确定。

隐式类型转换也被定义为多态的一种形式，叫做“**强迫多态**”。

参数多态

参数多态允许函数或数据类型被一般性的书写，从而它可以“统一”的处理值而不用依赖于它们的类型。参数多态是使语言更加有表现力而仍维持完全的**静态类型安全**的一种方式。这种函数和数据类型被分别称为“泛化函数”和“泛化数据类型”从而形成了**泛型编程**的基础。

参数多态的概念适用于数据类型和函数二者。可以被求值或应用于不同类型的值之上的函数叫做“多态函数”。看起来具有泛化类型性质的数据类型（比如具有任意类型的元素的列表）被指认为“多态数据类型”，就像根据它来做特殊化的泛化类型那样。

参数多态在函数式编程之中是普遍的，在这里它经常被简称为“多态”。参数多态在很多面向对象语言中也能获得到。例如，C++和D的模板，和在C#、Delphi和Java中所称谓的泛型:

```
class List<T> {
    class Node<T> {
        T elem;
        Node<T> next;
    }
    Node<T> head;
    int length() { ... }
}
List<B> map(Func<A, B> f, List<A> xs) {
    //...
}
```

> 多态可分为变量多态与函数多态。变量多态是指：基类型的变量（对于C++是引用或指针）可以被赋值基类型对象，也可以被赋值派生类型的对象。函数多态是指，相同的函数调用界面（函数名与实参表），传送给一个对象变量，可以有不同的行为，这视该对象变量所指向的对象类型而定。多态也可定义为“一种将不同的特殊行为和单个泛化记号相关联的能力”，变量多态是函数多态的基础。

事务

OO-五大基本原则SOLID - (Solid - '稳定的')

SRP - (Single Responsibility Principle) 单一职责原则

OCP - (Open - Close Principle) 开放封闭原则

LSP - (the Liskov Substitution Principle LSP) 里氏替换原则

ISP - (the Interface Segregation Principle ISP) 接口分离原则

DIP - (the Dependency Inversion Principle DIP) 依赖倒置原则

补充 - 设计模式中 更多的一个原则

LOD - (Law of Demeter) 迪米特法则

只与你的直接朋友交谈，不跟“陌生人”说话

> Talk only to your immediate friends and not to strangers

其含义是：如果两个软件实体无须直接通信，那么就不应当发生直接的相互调用，可以通过第三方转发该调用。其目的是降低类之间的耦合度，提高模块的相对独立性。

过度使用迪米特法则会使系统产生大量的中介类，从而增加系统的复杂性，使模块之间的通信效率降低。所以，在采用迪米特法则时需要反复权衡，确保高内聚和低耦合的同时，保证系统的结构清晰。

从迪米特法则的定义和特点可知，它强调以下两点：

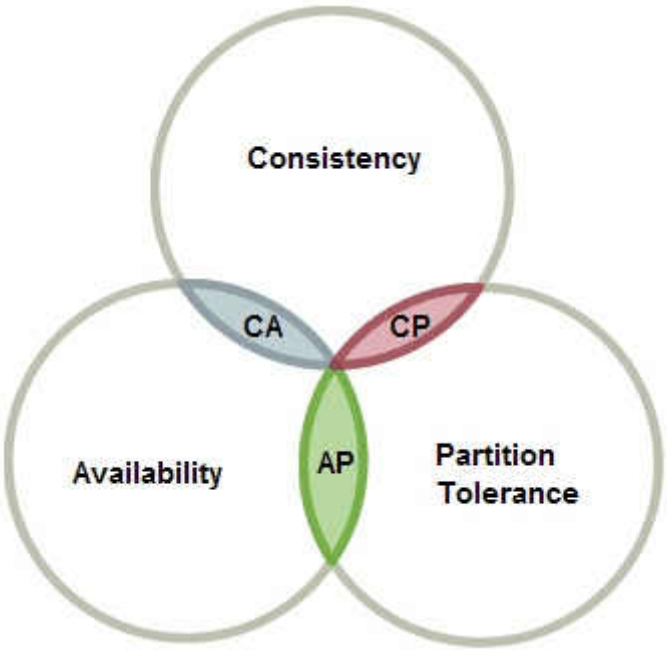
- 从依赖者的角度来说，只依赖应该依赖的对象。(一个中介，客户只要找中介要满足的楼盘 ，而不必跟每个楼盘发生联系。)
- 从被依赖者的角度说，只暴露应该暴露的方法。(无服务中的网关，前端都请求到网关，而不是直接请求具体的微服务。)

几个SsO缩写涉及到的概念

AJAX

CAP

CAP原则又称CAP定理，指的是在一个分布式系统中，一致性（Consistency）、可用性（Availability）、分区容错性（Partition tolerance）。CAP 原则指的是，这三个要素最多只能同时实现两点，不可能三者兼顾。
CAP原则是NoSQL数据库的基石。



分布式系统的CAP理论：理论首先把分布式系统中的三个特性进行了如下归纳：

- 一致性（C）：在分布式系统中的所有数据备份，在同一时刻是否同样的值。（等同于所有节点访问同一份最新的数据副本）
- 可用性（A）：在集群中一部分节点故障后，集群整体是否还能响应客户端的读写请求。（对数据更新具备高可用性）
- 分区容忍性（P）：以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择。

ORM-(Object Relational Mapping)对象关系映射

> ORM是一种程序设计技术，用于实现面向对象编程语言里不同类型系统的资料之间的转换。从效果上说，它其实是创建了一个可在编程语言里使用的“虚拟对象数据库”。如今已有很多免费和付费的ORM产品，而有些程序员更倾向于创建自己的ORM工具。

面向对象是从软件工程基本原则（如耦合、聚合、封装）的基础上发展起来的，而关系数据库则是从数学理论发展而来的，两套理论存在显著的区别。为了解决这个不匹配的现象，对象关系映射技术应运而生。

简单的说：ORM相当于中继资料。具体到产品上，ORM是通过使用描述对象和数据库之间映射的元数据，将程序中的对象自动持久化到关系数据库中。那么，到底如何实现持久化呢？一种简单的方案是采用硬编码方式，为每一种可能的数据库访问操作提供单独的方法。

这种方案存在以下不足：

1. 持久化层缺乏弹性。一旦出现业务需求的变更，就必须修改持久化层的接口
2. 持久化层同时与域模型与关系数据库模型绑定，不管域模型还是关系数据库模型发生变化，毒药修改持久化曾的相关程序代码，增加了软件的维护难度。

ORM提供了实现持久化层的另一种模式，它采用映射元数据来描述对象关系的映射，使得ORM中间件能在任何一个应用的业务逻辑层和数据库层之间充当桥梁。Java典型的ORM中间件有:Hibernate,MyBatis,SpeedFramework。

ORM的方法论基于三个核心原则：

· 简单：以最基本的形式建模数据。

· 传达性：数据库结构被任何人都能理解的语言文档化。

· 精确性：基于数据模型创建正确标准化了的结构。

MVC-(Model–view–controller)MVC模式

> MVC模式（Model–view–controller）是软件工程中的一种软件架构模式，把软件系统分为三个基本部分：模型（Model）、视图（View）和控制器（Controller）。MVC模式的目的是实现一种动态的程序设计，使后续对程序的修改和扩展简化，并且使程序某一部分的重复利用成为可能。除此之外，此模式透过对复杂度的简化，使程序结构更加直观。软件系统透过对自身基本部分分离的同时也赋予了各个基本部分应有的功能。专业人员可以依据自身的专长分组：

- 模型（Model） - 程序员编写程序应有的功能（实现算法等等）、数据库专家进行数据管理和数据库设计(可以实现具体的功能),负责资料访问。 用于封装与应用程序的业务逻辑相关的数据以及对数据的处理方法。“ Model ”有对数据直接访问的权力，例如对数据库的访问。“Model”不依赖“View”和“Controller”，也就是说， Model 不关心它会被如何显示或是如何被操作。但是 Model 中数据的变化一般会通过一种刷新机制被公布。为了实现这种机制，那些用于监视此 Model View 必须事先在此 Model 上注册，从而，View 可以了解在数据 Model 上发生的改变。（比如：观察者模式）

- 视图（View） - 界面设计人员进行图形界面设计。能够实现数据有目的的显示（理论上，这不是必需的）,负责处理消息。在 View 中一般没有程序上的逻辑。为了实现 View 上的刷新功能，View 需要访问它监视的数据模型（Model），因此应该事先在被它监视的数据那里注册。
- 控制器（Controller）- 负责转发请求，对请求进行处理。起到不同层面间的组织作用，用于控制应用程序的流程。它处理事件并作出响应。“事件”包括用户的行为和数据 Model 上的改变。

MVC模式在概念上强调 Model, View, Controller 的分离，各个模块也遵循着由 Controller 来处理消息，Model 掌管数据源，View 负责资料显示的职责分离原则，因此在实现上，MVC 模式的 Framework 通常会将 MVC 三个部分分离实现，，MVC-based 的应用程序在良好的职责分离的设计下，各个部分可独立行使单元测试，有利于与企业内的自动化测试、持续集成 (Continuous Integration) 与持续交付 (Continuous Delivery) 流程集成，减少应用程序改版部署所需的时间。

AOP-(Aspect-oriented Programming)面向切面程序设计

>AOP，又译作面向方面的程序设计、剖面导向程序设计,是计算机科学中的一种程序设计思想，旨在将横切关注点与业务主体进行进一步分离，以提高程序代码的模块化程度。通过在现有代码基础上增加额外的通知（Advice）机制，能够对被声明为“**切点**（Pointcut）”的代码块进行统一管理与装饰，如“对所有方法名以‘set*’开头的方法添加后台日志”。该思想使得开发人员能够将与代码核心业务逻辑关系不那么密切的功能（如日志功能）添加至程序中，同时又不降低业务代码的可读性。面向切面的程序设计思想也是面向切面软件开发的基础。

面向切面的程序设计将代码逻辑切分为不同的模块（即**关注点**（Concern）：一段特定的逻辑功能)。几乎所有的编程思想都涉及代码功能的分类，将各个关注点封装成独立的抽象模块（如函数、过程、模块、类以及方法等），后者又可供进一步实现、封装和重写。部分关注点“横切”程序代码中的数个模块，即在多个模块中都有出现，它们即被称作**横切关注点**（Cross-cutting concerns, Horizontal concerns）。

日志功能即是横切关注点的一个典型案例，因为日志功能往往横跨系统中的每个业务模块，即“横切”所有有日志需求的类及方法体。“而对于一个信用卡应用程序来说，存款、取款、帐单管理是它的核心关注点，日志和持久化将成为横切整个对象结构的横切关注点。”

切面的概念源于对面向对象的程序设计和计算反射的融合，但并不只限于此，它还可以用来改进传统的函数。与切面相关的编程概念还包括元对象协议、主题（Subject）、混入（Mixin）和委托（Delegate）。

从核心关注点中分离出横切关注点是面向切面的程序设计的核心概念。分离关注点使得解决特定领域问题的代码从业务逻辑中独立出来，业务逻辑的代码中不再含有针对特定领域问题代码的调用，业务逻辑同特定领域问题的关系通过**切面**来封装、维护，这样原本分散在在整个应用程序中的变动就可以很好的管理起来。

切面（Aspect）：在支配性分解的基础上，提供的一种辅助的模块化机制，这种新的模块化机制可以捕捉横切关注点。

IOC - (Inversion of Control)控制反转

> IOC是面向对象编程中的一种设计原则，可以用来减低计算机代码之间的耦合度。其中最常见的方式叫做依赖注入（Dependency Injection，简称DI），还有一种方式叫“依赖查找”（Dependency Lookup）。通过控制反转，对象在被创建的时候，由一个调控系统内所有对象的外界实体将其所依赖的对象的引用传递给它。也可以说，依赖被注入到对象中。

> 简单来说，依赖查找是主动和手动的依赖查找方式，通常需要依赖容器和标准API去实现；而依赖注入则是手动或自动依赖绑定的方式，无需依赖特定的容器和API。

IOC不是一种技术，只是一种思想，一个重要的面向对象编程的法则，它能指导我们如何设计出松耦合、更优良的程序。传统应用程序都是由我们在类内部主动创建依赖对象，从而导致类与类之间高耦合，难于测试；有了IOC容器后，把创建和查找依赖对象的控制权交给了容器，由容器进行注入组合对象，所以对象与对象之间是松散耦合，这样也方便测试，利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活

IOC很好的体现了面向对象设计法则之一 ——好莱坞法则：“别找我们，我们找你”；即由IOC容器帮对象找相应的依赖对象并注入，而不是由对象主动去找

DI-(Dependency Injection)依赖注入

依赖注入：是组件之间依赖关系由容器在运行期决定，形象的说，即由容器动态的将某个依赖关系注入到组件之中。依赖注入的目的并非为软件系统带来更多功能，而是为了提升组件重用的频率，并为系统搭建一个灵活、可扩展的平台。通过依赖注入机制，我们只需要通过简单的配置，而无需任何代码就可指定目标需要的资源，完成自身的业务逻辑，而不需要关心具体的资源来自何处，由谁实现。

理解DI的关键是：“谁依赖谁，为什么需要依赖，谁注入谁，注入了什么”，那我们来深入分析一下：

- 谁依赖于谁：当然是应用程序依赖于IOC容器；
- 为什么需要依赖：应用程序需要IOC容器来提供对象需要的外部资源；
- 谁注入谁：很明显是IOC容器注入应用程序某个对象，应用程序依赖的对象；
- 注入了什么：就是注入某个对象所需要的外部资源（包括对象、资源、常量数据）

DL-(Dependency Lookup)依赖查找

依赖查找，它是控制反转设计原则的一种实现方式。它的大体思路是：容器中的受控对象通过容器的 API 来查找自己所依赖的资源和协作对象。这种方式虽然降低了对象间的依赖，但是同时也使用到了容器的 API，造成了我们无法在容器外使用和测试对象。依赖查找是一种更加传统的 IOC 实现方式。

依赖查找也有两种方式：

- 依赖拖拽：注入的对象如何与组件发生联系，这个过程就是通过依赖拖拽实现；
- 上下文依赖查找：在某些方面跟依赖拖拽类似，但是上下文依赖查找中，查找的过程是在容器管理的资源中进行的，而不是从集中注册表中，并且通常是作用在某些设置点上；

本质上IOC和DI是同一思想下不同维度的表现,用通俗的话说就是，IOC是bean的注册，DI是bean的初始化

DRY & WET

一次且仅一次（英语：Once and only once，简称OAOO）又称为**Don't repeat yourself**（不要重复你自己，简称DRY）或**一个规则，实现一次**（One rule, one place）是面向对象编程中的基本原则，程序员的行事准则。旨在软件开发中，减少重复的信息。

DRY的原则是“系统中的每一部分，都必须有一个单一的、明确的、权威的代表”，指的是（由人编写而非机器生成的）代码和测试所构成的系统，必须能够表达所应表达的内容，但是不能含有任何重复代码。当DRY原则被成功应用时，一个系统中任何单个元素的修改都不需要与其逻辑无关的其他元素发生改变。此外，与之逻辑上相关的其他元素的变化均为可预见的、均匀的，并如此保持同步。

违反**DRY**原则的解决方案通常被称为**WET**，其有多种全称，包括“Write everything twice”（把每个东西写两次）、“We enjoy typing”（我们就是喜欢打字）或“Waste everyone's time”（浪费大家的时间）。