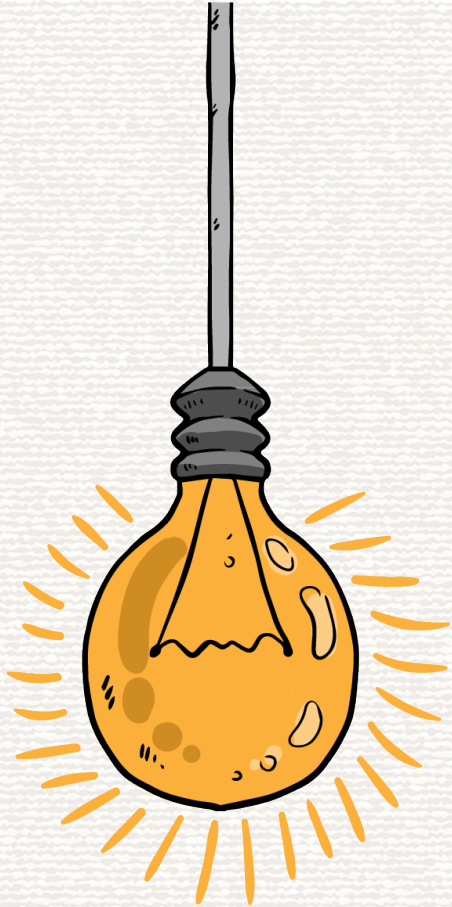


面向切面的编程思想

www.yidengxuetang.com



目录

CHUANGYI SHOUHUI

1.SOLID设计原则

2.依赖注入DI

3.控制反转IOC

4.面向切面编程AOP

编程思想的逐层提升

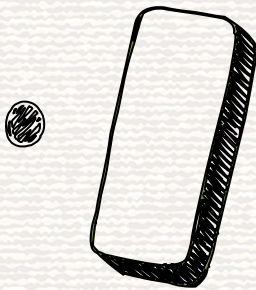
Improve programming thinking

面向对象



传统方式
动态的引入需要的类

工厂模式



业务层不需要关注实
例到底怎么生成的

面向切面



不用再写工厂类，
直接从IOC容器中
创建好的实例取



oop是静态的抽象
aop是动态的抽象

A large, stylized orange brushstroke graphic that serves as a background for the text.


SOLID

面向对象设计原则之一

单一功能、开闭原则、里氏替换、接口隔离以及依赖反转

原则基本概念

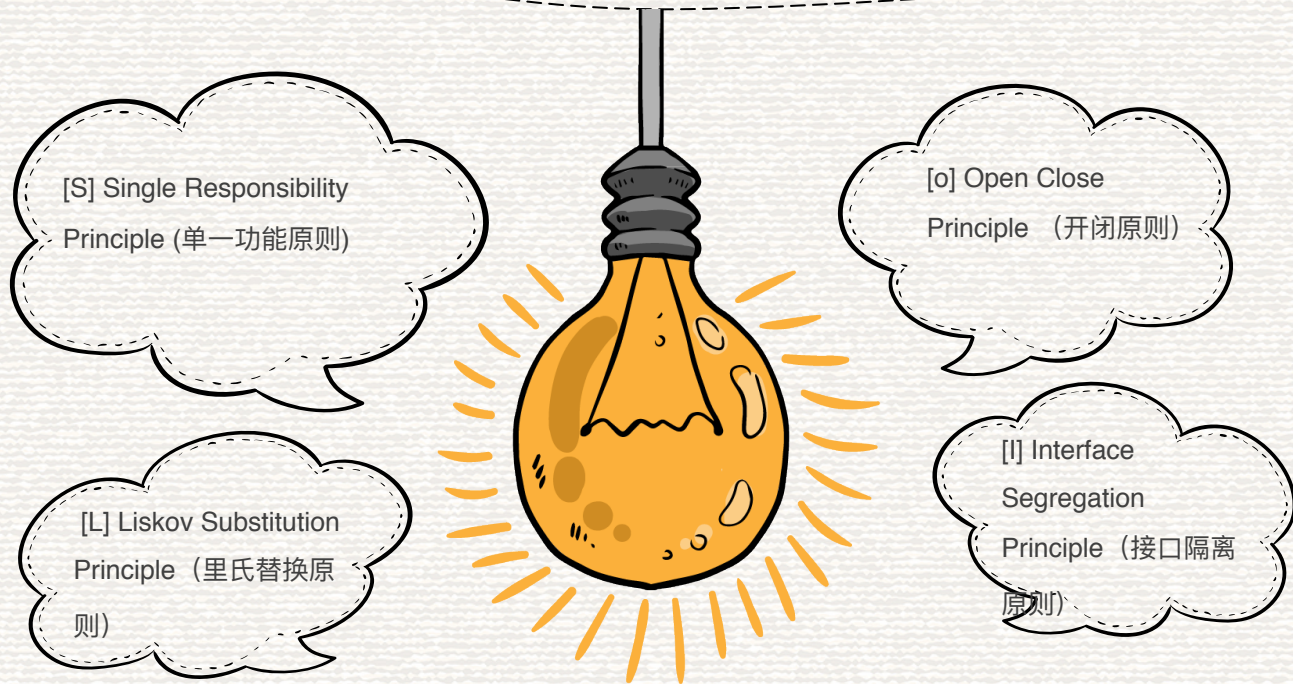
SOLID



程序设计领域， SOLID (单一功能、开闭原则、里氏替换、接口隔离以及依赖反转)是由罗伯特·C·马丁在21世纪早期 引入的记忆术首字母缩略字，指代了面向对象编程和面向对象设计的五个基本原则。当这些原则被一起应用时，它们使得一个程序员开发一个容易进行软件维护和扩展的系统变得更加可能SOLID被典型的应用在测试驱动开发上，并且是敏捷开发以及自适应软件开发的基本原则的重要组成部分。

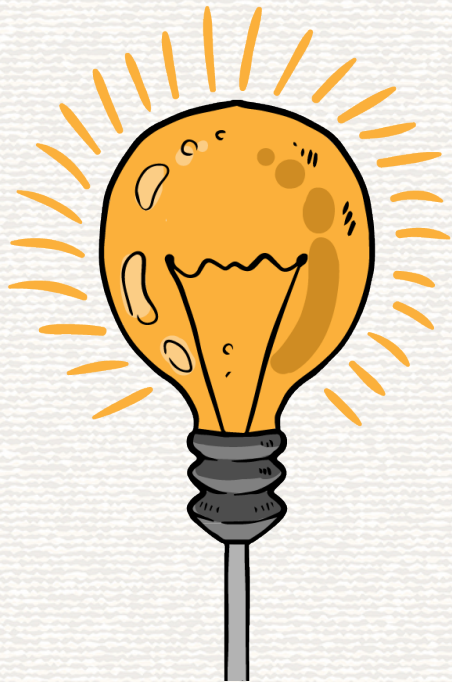
这几个字母的代表含义

SOLID



[S] 单一功能原则

Single Responsibility Principle



单一功能原则：单一功能原则认为对象应该仅具有一种单一功能的概念。

换句话说就是让一个类只做一种类型责任，当这个类需要承担其他类型的责任的时候，就需要分解这个类。在所有的SOLID原则中，这是大多数开发人员感到最能完全理解的一条。严格来说，这也可能是违反最频繁的一条原则了。单一责任原则可以看作是低耦合、高内聚在面向对象原则上的引申，将责任定义为引起变化的原因，以提高内聚性来减少引起变化的原因。责任过多，可能引起它变化的原因就越多，这将导致责任依赖，相互之间就产生影响，从而极大的损伤其内聚性和耦合度。单一责任，通常意味着单一的功能，因此不要为一个模块实现过多的功能点，以保证实体只有一个引起它变化的原因。

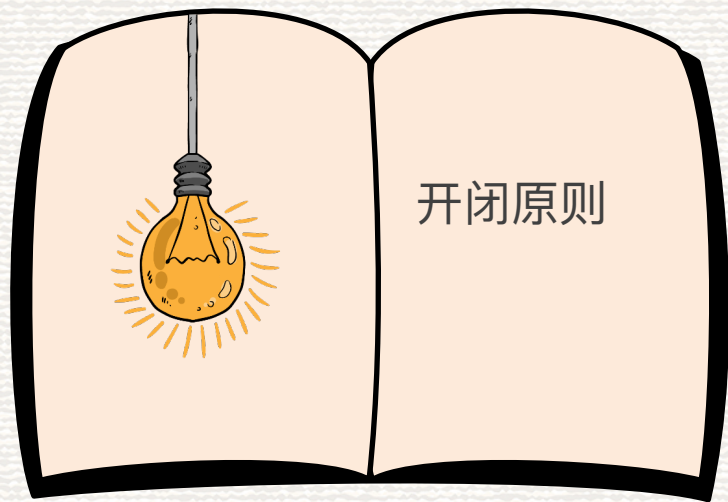
[S] 单一功能原则

Single Responsibility Principle



[O] 开闭原则

Open Close Principle



开闭原则(ocp) 认为“软件体应该是对于扩展开放的，但是对于修改封闭的”的概念。

软件实体应该是可扩展，而不可修改的。也就是说，对扩展是开放的，而对修改是封闭的（“开”指的就是类、模块、函数都应该具有可扩展性，“闭”指的是它们不应该被修改。也就是说你可以新增功能但不能去修改源码）。这个原则是诸多面向对象编程原则

中最抽象、最难理解的一个。

对扩展开放，意味着有新的需求或变化时，可以对现有代码进行扩展，以适应新的情况。对修改封闭，意味着类一旦设计完成，就可以独立完成其工作，而不要对类进行任何修改。可以使用变化和不变来说明：封装不变部分，开放变化部分，一般使用接口继承实现方式来实现“开放”应对变化，说大白话就是：你不是要变化吗？，那么我就让你继承实现一个对象，用一个接口来抽象你的职责，你变化越多，继承实现的子类就越多。

[O] 开闭原则

Open Close Principle



[L] 里氏替换原则

Liskov Substitution Principle



里氏替换原则：里氏替换原则认为“程序中的对象应该是可以在不改变程序正确性的前提下被它的子类所替换的”的概念。

子类必须能够替换成它们的基类。即：子类应该可以替换任何基类能够出现的地方，并且经过替换以后，代码还能正常工作。另外，不应该在代码中出现if/else之类对子类类型进行判断的条件。里氏替换原则LSP是使代码符合开闭原则的一个重要保证。正是由于子类型的可替换性才使得父类型的模块在无需修改的情况下就可以扩展。在很多情况下，在设计初期我们类之间的关系不是很明确，LSP则给了我们一个判断和设计类之间关系的基准：需不需要继承，以及怎样设计继承关系。

当一个子类的实例应该能够替换任何其超类的实例时，它们之间才具有is-A关系。继承对于OCP，就相当于多态性对于里氏替换原则。子类可以代替基类，客户使用基类，他们不需要知道派生类所做的事情。这是一个针对行为职责可替代的原则，如果S是T的子类型，那么S对象就应该在不改变任何抽象属性情况下替换所有T对象。

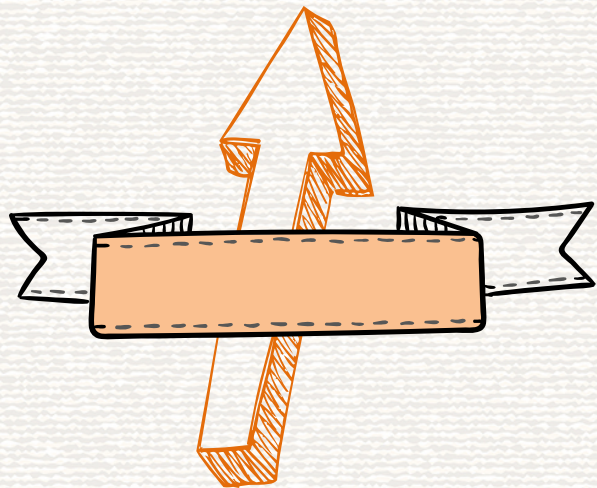
[L] 里氏替换原则

Liskov Substitution Principle



[I] 接口隔离原则

Interface Segregation Principle



接口隔离原则：接口隔离原则认为“多个特定客户端接口要好于一个宽泛用途的接口”的概念。

不能强迫用户去依赖那些他们不使用的接口。换句话说，使用多个专门的接口比使用单一的总接口总要好(JavaScript 几乎没有接口的概念所以使用ts)。注意：在代码中应用ISP并不意味着服务就是绝对安全的。仍然需要采用良好的编码实践，以确保正确的验证与授权。

这个原则起源于施乐公司，他们需要建立了一个新的打印机系统，可以执行诸如装订的印刷品一套，传真多种任务。此系统软件创建从底层开始编制，并实现了这些任务功能，但是不断增长的软件功能却使软件本身越来越难适应变化和维护。每一次改变，即使是最小的变化，有人可能需要近一个小时的重新编译和重新部署。这是几乎不可能再继续发展，所以他们聘请罗伯特Robert帮助他们。他们首先设计了一个主要类Job,几乎能够用于实现所有任务功能。只要调用Job类的一个方法就可以实现一个功能，Job类就变动非常大，是一个胖模型啊，对于客户端如果只需要一个打印功能，但是其他无关打印的方法功能也和其耦合，ISP原则建议在客户端和Job类之间增加一个接口层，对于不同功能有不同接口，比如打印功能就是Print接口，然后将大的Job类切分为继承不同接口的子类，这样有一个Print Job类，等等。

[I] 接口隔离原则

Interface Segregation Principle



A large, horizontal, orange brushstroke graphic that serves as a background for the text.

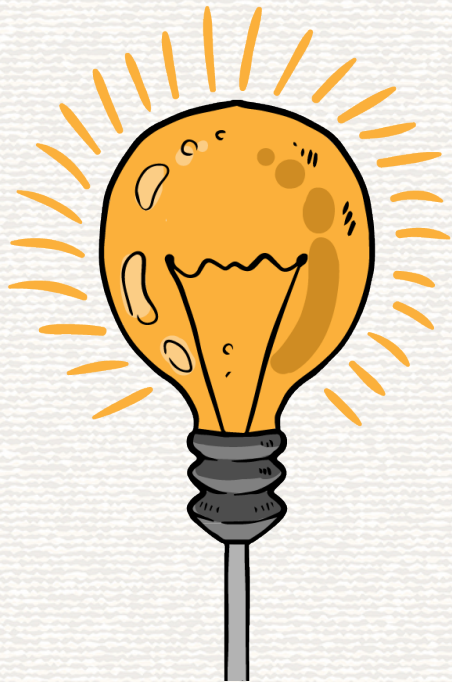
DI

依赖注入 (Dependency Injection)

为一个方法应该遵从“依赖于抽象而不是一个实例”的概念。依赖注入是该原则的一种实现方式。

[D] 依赖反转原则

Dependency Inversion Principle



依赖倒置原则(Dependency Inversion Principle, DIP)规定：代码应当取决于抽象概念，而不是具体实现。

高层模块不应该依赖于低层模块，二者都应该依赖于抽象

抽象不应该依赖于细节，细节应该依赖于抽象 (总结解耦)

类可能依赖于其他类来执行其工作。但是，它们不应当依赖于该类的特定具体实现，而应当是它的抽象。这个原则实在是太重要了，社会的分工化，标准化都是这个设计原则的体现。

显然，这一概念会大大提高系统的灵活性。如果类只关心它们用于支持特定契约而不是特定类型的组件，就可以快速而轻松地修改这些低级服务的功能，同时最大限度地降低对系统其余部分的影响。

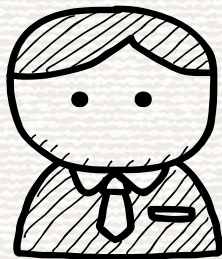
[D] 依赖反转原则

Dependency Inversion Principle



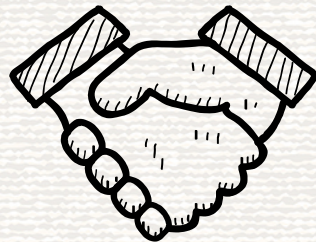
依赖注入

当某个角色要另一个角色协助时，通常由调用者来创建被调用者的实例。现在创建实例由容器来完成注入调用者。



注入过程

如果需要调用另一个对象协助时，无须在代码中创建被调用者，而是依赖于外部的注入



两种方式

设值注入、构造注入

[D] 依赖反转原则

Dependency Inversion Principle



Code演示

IOC

控制反转 (Inversion of Control)

IoC可以认为是一种全新的设计模式，但是理论和时间成熟相对较晚。

[IOC] 控制反转



什么是控制反转

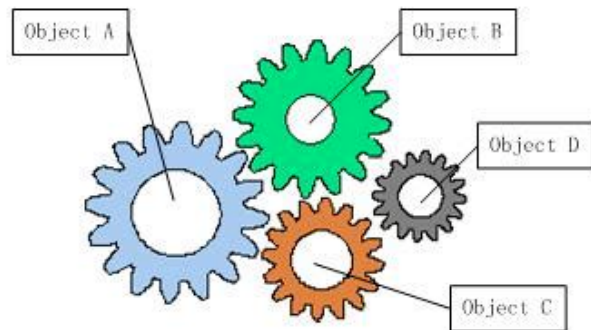
控制反转（Inversion of Control，缩写为IoC），是面向对象编程中的一种设计原则，可以用来减低计算机代码之间的耦合度。其中最常见的方式叫做依赖注入（Dependency Injection，简称DI），还有一种方式叫“依赖查找”（Dependency Lookup）。通过控制反转，对象在被创建的时候，由一个调控系统内所有对象的外界实体，将其所依赖的对象的引用传递给它。也可以说，依赖被注入到对象中。

[IOC] 控制反转

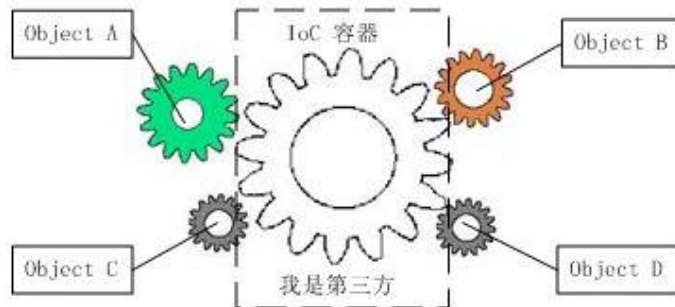
依赖查找:
容器提供回调接口
和上下文条件给组件。

依赖注入:
组件不做定位查询, 只
提供普通的方法让容器
去决定依赖关系。

[IOC] 控制反转



未使用IOC



开始使用IOC

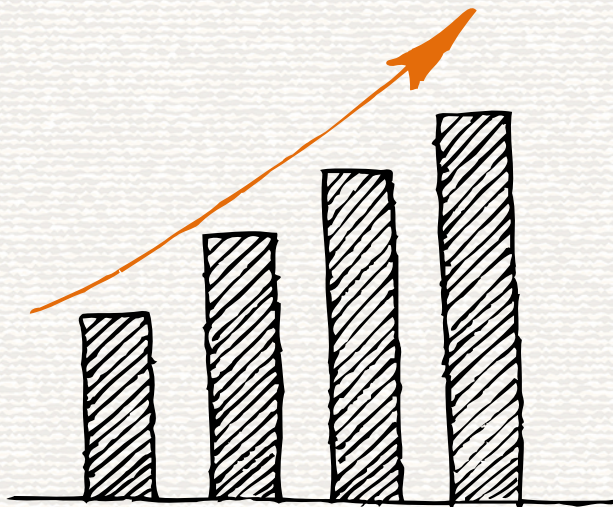
AOP

面向切面 (Aspect Oriented Programming)

面向切面编程 (aop) 是对面向对象编程 (oop) 的补充

面向切面编程

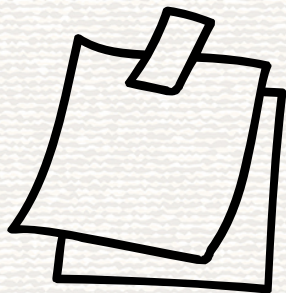
Aspect Oriented Programming



在软件业，AOP为Aspect Oriented Programming的缩写，意为：面向切面编程，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。AOP是OOP的延续，是软件开发中的一个热点，也是Spring框架中的一个重要内容，是函数式编程的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

面向切面编程

Aspect Oriented Programming



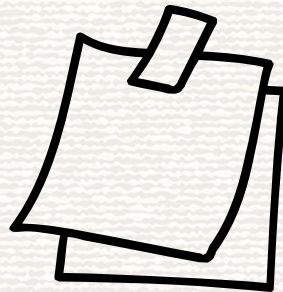
基础概念

AOP完善spring的依赖注入 (DI) 面向对象编程将程序分解成各个层次的对象, 面向切面编程将程序运行过程分解成各个切面。



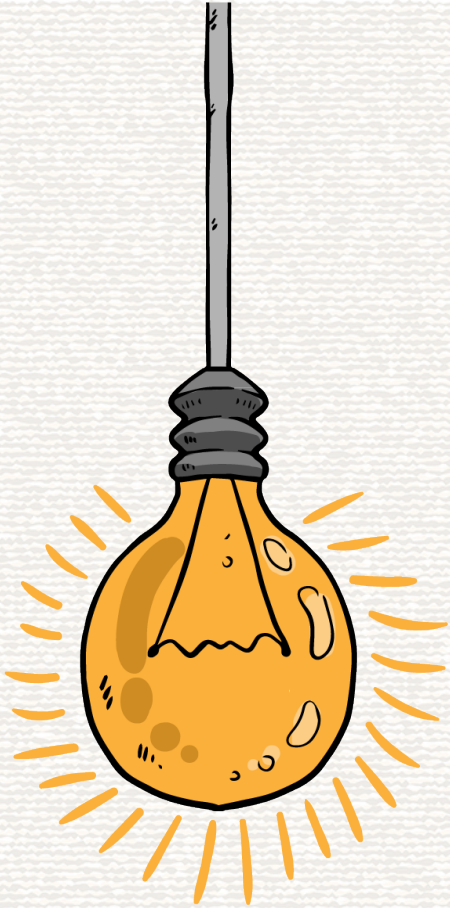
Filter

Filter (过滤器) 也是一种AOPA 它利用一种称为"横切"的技术, 剖开封装的对象内部, 并将那些影响了多个类的公共行为封装到一个可重用模块, 并将其命名为 "Aspect", 即切面。所谓"切面"。



优点

AOP的好处就是你只需要干你的正事, 其它事情别人帮你干。在你访问数据库之前, 自动帮你开启事务, 当你访问数据库结束之后, 自动帮你提交/回滚事务!



Q&&A