【讲义】第1讲：设计原则

〇、大纲

- **单一职责**——组装电脑 & 凤凰架构 & 微服务。
- **接口隔离**——接口不要过于臃肿，配合单一职责，仅提供职责内的方法。

- **依赖倒置**——国家购买出租车 & DDD。

- **里式替换**——多用组合，少用继承。
- **迪米特法则**——创建/获取账户功能 & DDD领域间调用。

- **开闭原则**——对扩展开发对修改关闭 & 活动积分功能 & SPI。

一、单一职责原则

1.1> 名词解释

- SRP

> Single Reposibility Principle：单一职责

- RBAC

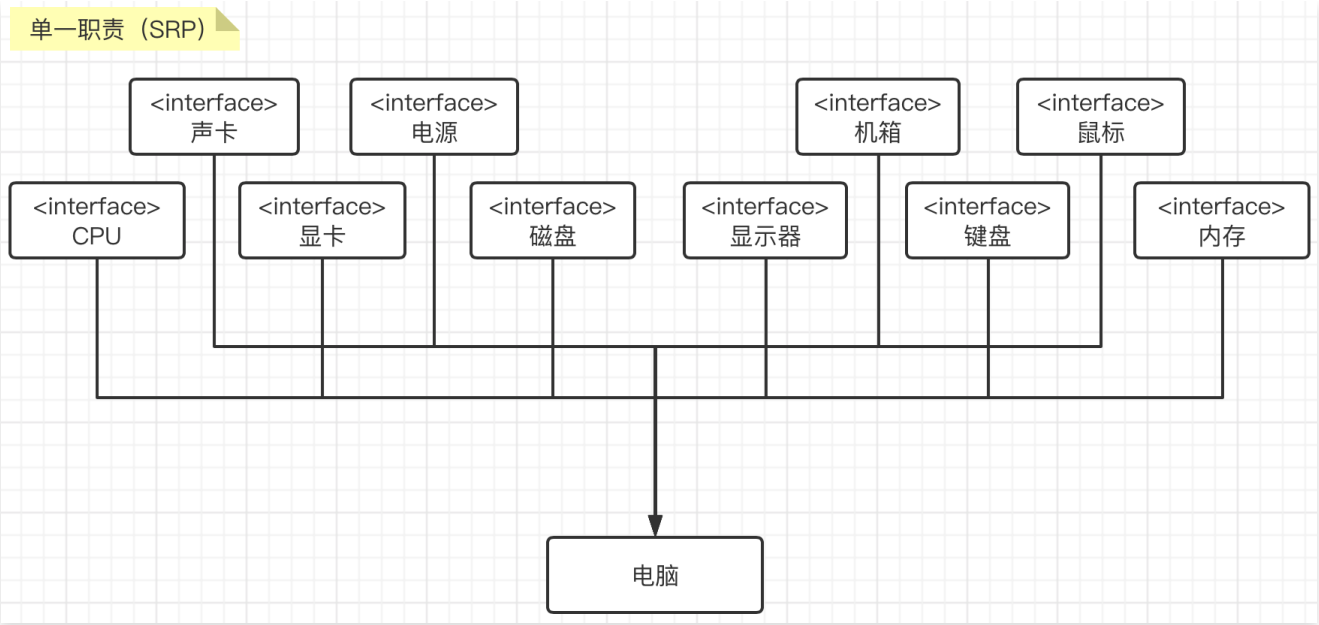
> Role-Based Access Control：基于角色的访问控制

1.2> 定义

- **一个类或者模块只负责完成一个职责**。

1.3> 最佳实践

- 接口一定要尽量做到单一职责。
- 类的设计不应该大而全，要设计**粒度小**、**功能单一**的类。**如果一个类中存在多个不相干的功能，那么我们就违背了单一职责原则**，应该将它拆分成多个功能单一、粒度更新的类。
- 如下图所示：



> 每个电脑组件**各司其职**。如果要升级，可以**针对某一模块组件升级**，而不用影响到整台电脑。而且是**基于接口开发**，可以扩展更换不同的实现类。参考：**凤凰架构**。

二、里氏替换原则

2.1> 名词解释

- LSP

> Liskov Substitution Principle

2.2> 定义

- LSP的原定义比较复杂，我们一般对里氏替换原则LSP的解释为：**子类对象能够替换父类对象，而程序逻辑不变。继承必须确保父类所拥有的性质在子类中仍然成立。**
- 继承的优点

> - 提高代码的**重用性**，子类拥有父类的方法和属性；
> - 提高代码的**可扩展性**，子类可形似于父类，但异于父类，保留自我的特性；

- 继承的缺点

> - **继承是侵入性的，只要继承就必须拥有父类的所有方法和属性**，在一定程度上约束了子类，降低了代码的灵活性；
> - **增加了耦合**，当父类的常量、变量或者方法被修改了，需要考虑子类的修改，所以一旦父类有了

变动，很可能会造成非常糟糕的结果，要重构大量的代码。

- 任何基类可以出现的地方，子类一定可以出现。**里氏替换原则是继承复用的基石，只有当衍生类可以替换基类，软件单位的功能不受到影响时，即基类随便怎么改动子类都不受此影响（即：子类可以拥有父类[非接口or抽象类]的方法，但是不要重写父类的方法），那么基类才能真正被复用**。因为继承带来的侵入性，增加了耦合性，也降低了代码灵活性，父类修改代码，子类也会受到影响，此时就需要里氏替换原则。
- 里氏替换原则有至少以下两种含义：

- 1> 里氏替换原则是针对继承而言的，**如果继承是为了实现代码重用，**也就是为了共享方法，那么共享的父类方法就应该保持不变，不能被子类重新定义。子类只能通过新添加方法来扩展功能，父类和子类都可以实例化，而子类继承的方法和父类是一样的，父类调用方法的地方，子类也可以调用同一个继承得来的，逻辑和父类一致的方法，这时用子类对象将父类对象替换掉时，当然逻辑一致，相安无事。

  eg：子类重写了父类，那么当父类被重写的方法发生了变动（比如：里面加入了鉴权方法），那么子类的重写方法没有鉴权，则会出现问题。（高耦合）
- 2> **如果继承的目的是为了多态**，而多态的前提就是子类覆盖并重新定义父类的方法，为了符合LSP，我们应该将父类定义为抽象类，并定义抽象方法，让子类重新定义这些方法，当父类是抽象类时，父类就是不能实例化，所以也不存在可实例化的父类对象在程序里。也就不存在子类替换父类实例（根本不存在父类实例了）时逻辑不一致的可能。

- 在实际应用中，里氏替换可以表现出如下**四种**情况：

- **情况一**：子类必须完全实现父类的方法

  eg：有一个WarGun的接口或抽象类，用来描述战场枪支，里面有kill()方法，来描述枪支的杀伤力，其子类AK47，AWM，M416都实现了kill()方法，但是ToyGun因为是玩具枪，它没有杀伤力，所以，无法实现kill()，那么针对这种情况，即：子类不能完整的实现父类的方法，或者父类的某些方法在子类中已经发生了“畸变”，则建议断开父子继承关系，采用依赖、聚集、组合等关系代替继承。
- **情况二**：子类中可以增加自己特有的方法，即：子类可以有自己的个性
- **情况三**：当子类覆盖或实现父类的方法时，方法的`前置条件`（即方法的形参）要比父类方法的输入参数**更宽松**

  eg：如果父类方法为doSomething(Map map)，子类方法为doSomething(HashMap map)，那么由于方法签名不同，其实不是重写而是重载了。那么当入参为HashMap时，调用父类对象的doSomething方法，则执行了父类的方法。调用子类的doSomething方法，则执行了子类。那么，这种情况下，我们往往会以为，调用的都是父类的doSomething。
- **情况四**：当子类的方法实现父类的抽象方法时，方法的`后置条件`（即方法的返回值）要比父类**更严格**，编译上也会提示有问题

```java
public class Parent {

    public ArrayList doSomething() {
        ArrayList list = new ArrayList<>();
        return list;
    }
}
```

```java
public class Son extends Parent {

    @Override
    public List doSomething() {
        return super.doSomething();
    }
}
```

2.3> 最佳实践

- 不符合LSP的最常见的情况是，父类和子类都是可实例化的非抽象类，且父类的方法被子类重新定义，这一类的实现继承会造成父类和子类间的强耦合，也就是实际上并不相关的属性和方法牵强附会在一起，不利于程序扩展和维护。
- **如何符合LSP？总结一句话 —— 就是尽量不要从可实例化的父类中继承，而是要使用基于抽象类和接口的继承。**

2.4> Java中的函数签名

- **方法签名的组成：**

> 1> 方法名
> 2> 参数列表（形参类别、个数、顺序）

- **特别注意：**

> 1> 与返回值、修饰符以及异常无关
> 2> 在Class文件格式之中，返回值不同，可以合法地共存于一个Class文件中。
> 3> 在泛型的使用中，参数List<String>与List<Integer>在经过类型擦除后，是相同参数。
> 4> 参数String... strings与参数String[] arr，是相同参数

- **重载和重写**

> 重载：同一个类中方法签名不同的方法。
> 重写：方法签名必定相同，返回值必定相同， 访问修饰符 子 > 父， 异常类 子 < 父

三、依赖倒置原则

3.1> 名词解释

- DIP

> Dependence Inversion Principle

3.2> 定义

- 下层模块引入上层模块的依赖，改变原有自上而下的依赖方向。
- 要面向接口编程，不要面向实现编程。

3.3> 最佳实践

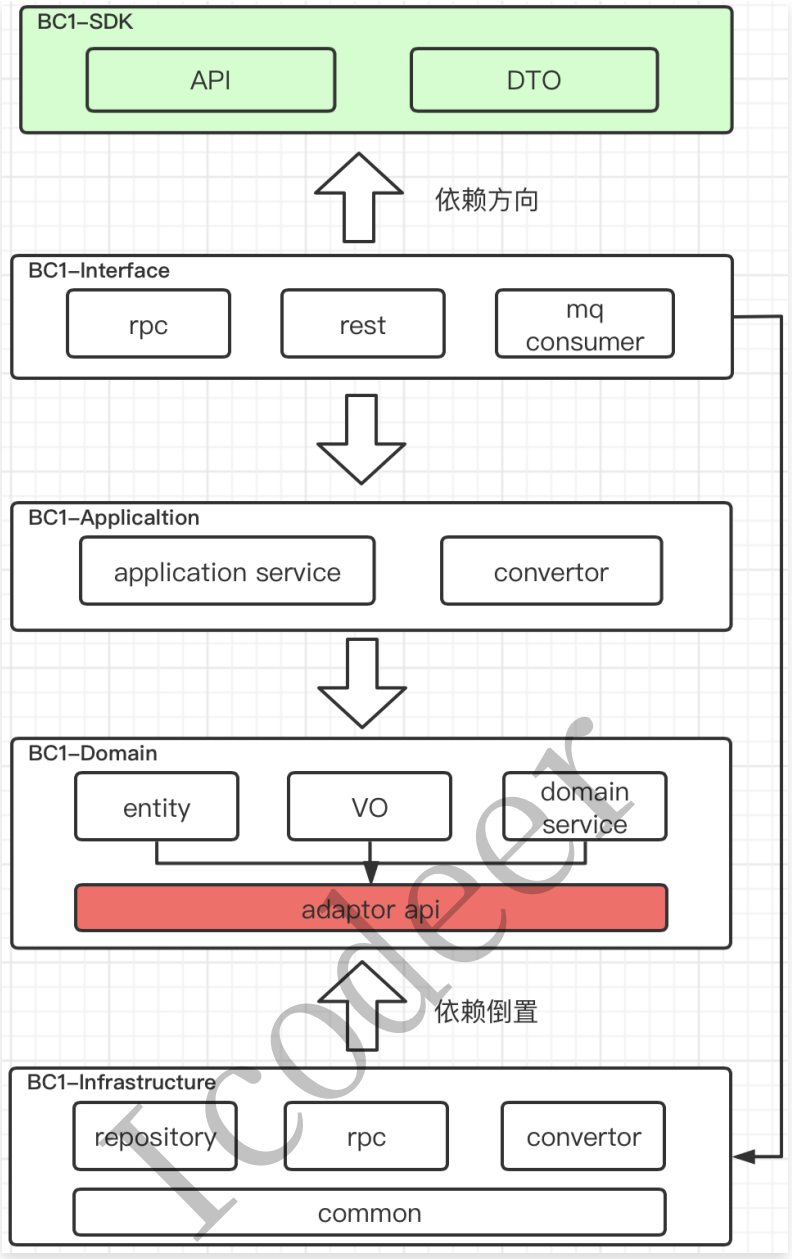
3.3.1> 针对于类直接的依赖

- 讲一个故事：建国初期，国家要培养一批出租车司机。那么需要会开车的人和汽车。那么，为了市容市貌，最初规定，汽车只能是大众生产的捷达汽车，并且喷上某种代表出租车的配色即可。期初需求不大，培养出1万出租车司机即可。随着国家富强了，很多二三四线城市也要培养出租车司机，那么对于汽车的订单量就大大提升了，第二年全国需要培养30万名出租车司机。但是德国大众集团每年针对捷达汽车的产量只有10万台，所以，明年培养30万名出租车司机的需求就无法实现了。因为我们严重的依赖了下游的德国大众公司。那么，针对这种情况，国家出台了新的政策。出租车司机，不是必须只能开大众捷达汽车了。只要作为出租车的汽车满足一定的规定要求即可。比如：四门四座、油耗、安全性、安装计费设备等等。那么，其他的汽车品牌，比如：现代、本田、丰田、比亚迪等等，为了抢占这块市场，纷纷按照国家出台的出租车标准，生产了一系列的轿车。那么，总的汽车产能一下子就满足要求了。
- 我们发现没有，解决这个问题的关键就是——**依赖方向变化了**（产生了依赖的倒置）。

> 从最初的**依赖具体对象**：
>
> 国家（上游）——>依赖——>大众集团公司（下游）
>
> 变为了**依赖接口规范**：
>
> 大众和其他汽车品牌集团公司（下游）——>依赖——>国家出台的出租汽车标准（上游）

3.3.2> 针对于项目模块的依赖

- 图如下所示：



四、接口隔离原则

4.1> 定义

- 建立单一接口，**不要建立臃肿庞大的接口**。接口尽量细化，同时接口中的方法尽量少。
- 单一职责原则和接口隔离的区别？答：它们的审视角度不同。

> - 单一职责要求的是**类和接口职责单一**，注重的是**职责**，这是业务逻辑上的划分。
> - 接口隔离原则要求**接口的方法尽量少**。

- 保证接口的纯洁性

- 接口要尽量小
  - 不要违反单一职责原则。
  - 要适度的小，要适度。
- 接口要高内聚
  - 提高接口、类、模块的处理能力，减少对外的交互。
- 定制服务
  - 通过对高质量接口的组装，实现服务的定制化。

五、迪米特法则/最少知识原则

5.1> 名词解释

- LoD

> Law of Demeter

5.2> 定义

- 迪米特法则，也称为最少知识原则。
- 描述的是：一个类应该对自己需要耦合或调用的类知道得最少，你（被耦合或调用的类）的内部是如何复杂，那是你的事儿，和我没关系，我就知道你提供的这么多public方法，我就调用这么多，其他的我一概不关系。

  eg:当一个微信用户要在我们系统操作业务逻辑的时候，我们的需求是，如果微信用户没有注册我们系统的话，我们默认的调用注册接口去注册它，注册成功后，把用户信息返回给业务系统；如果她已经注册了，即已经存在于我们的用户表，则把用户信息返回给业务系统，再通过它的用户信息，进行下一步业务操作。那么针对于这个需求，我们其实应该是希望负责用户信息的研发团队，给我们提供一个接口，即：获得用户信息的接口。而不应该提供用户信息查询接口，用户信息注册接口，甚至底层还涉及到其他安全性接口，由我们一步一步的去调用。这样就违反了迪米特法则了。
- 类似DDD里面领域划分的概念。是我域负责的业务我负责，不是我域的业务，由相关领域负责。

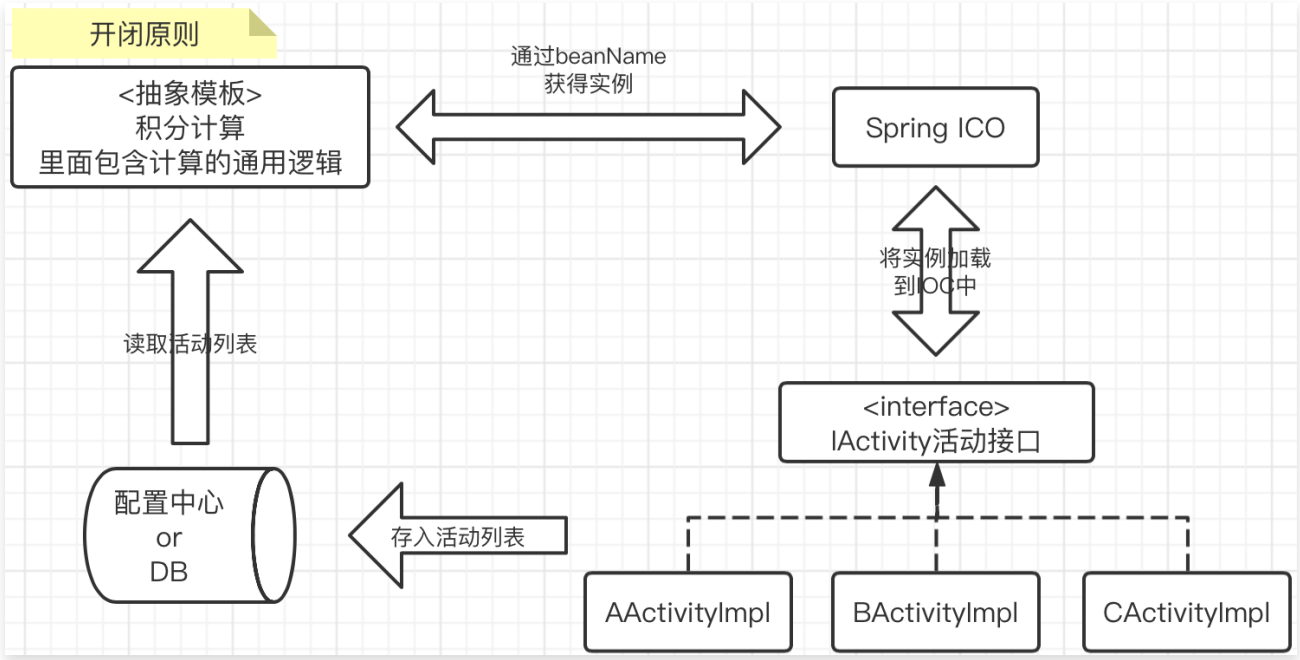
六、开闭原则

6.1> 定义

- 类、方法、模块应该对**扩展开放**，对**修改关闭**。例子：macbook的外接扩展器。
- 通俗讲：添加一个功能应该是在已有的代码基础上进行扩展，而不是修改已有的代码。

6.2> 最佳实践

- 活动积分功能



- 通过扩展配置，或者扩展原有接口的方式，应对新的需求变更。现实中，还是很难实现完全开闭原则的，但是我们在开发设计中，应该贯彻这种思想。

吾尝终日而思矣，不如须臾之所学也；
吾尝跂而望矣，不如登高之博见也。
登高而招，臂非加长也，而见者远；
顺风而呼，声非加疾也，而闻者彰。
假舆马者，非利足也，而致千里；
假舟楫者，非能水也，而绝江河。
君子生非异也，善假于物也。
------ 摘自《劝学》

愿本文可以成为大家的“山”、“风”、“马”、“舟”，助大家在技术之路上乘风破浪，大展宏图~~

同时，也欢迎大家关注我的公众号“爪哇缪斯”(^o^)/~ 「干货分享，每天更新」