面向对象 相关问题

1.SOLID原则

SOLID 原则其实是用来指导软件设计的，它一共分为五条设计原则，分别是：

单一职责原则（SRP）开闭原则（OCP）里氏替换原则（LSP）接口隔离原则（ISP）依赖倒置原则（DIP）

依赖倒置原则（DIP）单一职责原则（SRP）

单一职责原则（Single Responsibility Principle），它的定义是：应该有且仅有一个原因引起类的变更。简单地说：接口职责应该单一，不要承担过多的职责。用生活中肯德基的例子来举例：负责前台收银的服务员，就不要去餐厅收盘子。负责餐厅收盘子的就不要去做汉堡。

单一职责适用于接口、类，同时也适用于方法。例如我们需要修改用户密码，有两种方式可以实现，一种是用「修改用户信息接口」实现修改密码，一种是新起一个接口来实现修改密码功能。在单一职责原则的指导下，一个方法只承担一个职能，所以我们应该新起一个接口来实现修改密码的功能。

单一职责原则的重点在于职责的划分，很多时候并不是一成不变的，需要根据实际情况而定。单一职责能够使得类复杂性降低、类之间职责清晰、代码可读性提高、更加容易维护。但它的缺点也很明显，就是对技术人员要求高，有些时候职责难以区分。

我们在设计一个类的时候，可以先从粗粒度的类开始设计，等到业务发展到一定规模，我们发现这个粗粒度的类方法和属性太多，且经常修改的时候，我们就可以对这个类进行重构了，将这个类拆分成粒度更细的类，这就是所谓的持续重构。

开闭原则（OCP）

开闭原则（Open Closed Principle），它的定义是：一个软件实体，如类、模块和函数应该对扩展开放，对修改关闭。简单地说：就是当别人要修改软件功能的时候，使得他不能修改我们原有代码，只能新增代码实现软件功能修改的目的。

这听着有点玄乎，我来举个例子吧。

这段代码模拟的是对于水果剥皮的处理程序。如果是苹果，那么是一种拨皮方法；如果是香蕉，则是另一种剥皮方法。如果以后还需要处理其他水果，那么就会在后面加上很多 if else 语句，最终会让整个方法变得又臭又长。如果恰好这个水果中的不同品种有不同的剥皮方法，那么这里面又会有很多层嵌套。

if(type == apple){//deal with apple} else if (type == banana){//deal with banana} else if (type == ......){//......}

可以看得出来，上面这样的代码并没有满足「对拓展开放，对修改封闭」的原则。每次需要新增一种水果，都可以直接在原来的代码上进行修改。久而久之，整个代码块就会变得又臭又长。

如果我们对剥水果皮这件事情做一个抽象，剥苹果皮是一个具体的实现，剥香蕉皮是一个具体的实现，那么写出的代码会是这样的：

public interface PeelOff {void peelOff();}public class ApplePeelOff implement PeelOff{void peelOff(){//deal with apple}}public class BananaPeelOff implement PeelOff{void peelOff(){//deal with banan}}public class PeelOffFactory{private Map<String, PeelOff> map = new HashMap();private init(){//init all the Class that implements PeelOff interface}}.....public static void main(){String type = "apple";PeelOff peelOff = PeelOffFactory.getPeelOff(type); //get ApplePeelOff Class Instance.peelOff.pealOff();}

上面这种实现方式使得别人无法修改我们的代码，为什么？

因为当需要对西瓜剥皮的时候，他会发现他只能新增一个类实现 PeelOff 接口，而无法在原来的代码上修改。这样就实现了「对拓展开放，对修改封闭」的原则。

里氏替换原则（LSP）

里氏替换原则（LSP）的定义是：所有引用基类的地方必须能透明地使用其子类的对象。简单地说：所有父类能出现的地方，子类就可以出现，并且替换了也不会出现任何错误。例如下面 Parent 类出现的地方，可以替换成 Son 类，其中 Son 是 Parent 的子类。

Parent obj = new Son();等价于Son son = new Son();

这样的例子在 Java 语言中是非常常见的，但其核心要点是：替换了也不会出现任何的错误。这就要求子类的所有相同方法，都必须遵循父类的约定，否则当父类替换为子类时就会出错。这样说可能还是有点抽象，我举个例子。

public class Parent{// 定义只能扔出空指针异常public void hello throw NullPointerException(){}}public class Son extends Parent{public void hello throw NullPointerException(){// 子类实现时却扔出所有异常throw Exception;}}

上面的代码中，父类对于 hello 方法的定义是只能扔出空指针异常，但子类覆盖父类的方法时，却扔出了其他异常，违背了父类的约定。那么当父类出现的地方，换成了子类，那么必然会出错。

其实这个例子举得不是很好，因为这个在编译层面可能就有错误。但表达的意思应该是到位了。

而这里的父类的约定，不仅仅指的是语法层面上的约定，还包括实现上的约定。有时候父类会在类注释、方法注释里做了相关约定的说明，当你要覆写父类的方法时，需要弄懂这些约定，否则可能会出现问题。例如子类违背父类声明要实现的功能。比如父类某个排序方法是从小到大来排序，你子类的方法竟然写成了从大到小来排序。

里氏替换原则 LSP 重点强调：对使用者来说，能够使用父类的地方，一定可以使用其子类，并且预期结果是一致的。

接口隔离原则（ISP）

接口隔离原则（Interface Segregation Principle）的定义是：类间的依赖关系应该建立在最小的接口上。简单地说：接口的内容一定要尽可能地小，能有多小就多小。

举个例子来说，我们经常会给别人提供服务，而服务调用方可能有很多个。很多时候我们会提供一个统一的接口给不同的调用方，但有些时候调用方 A 只使用 1、2、3 这三个方法，其他方法根本不用。调用方 B 只使用 4、5 两个方法，其他都不用。接口隔离原则的意思是，你应该把 1、2、3 抽离出来作为一个接口，4、5 抽离出来作为一个接口，这样接口之间就隔离开来了。

那么为什么要这么做呢？我想这是为了隔离变化吧！想想看，如果我们把 1、2、3、4、5 放在一起，那么当我们修改了 A 调用方才用到 的 1 方法，此时虽然 B 调用方根本没用到 1 方法，但是调用方 B 也会有发生问题的风险。而如果我们把 1、2、3 和 4、5 隔离成两个接口了，我修改 1 方法，绝对不会影响到 4、5 方法。

除了改动导致的变化风险之外，其实还会有其他问题，例如：调用方 A 抱怨，为什么我只用 1、2、3 方法，你还要写上 4、5 方法，增加我的理解成本。调用方 B 同样会有这样的困惑。

在软件设计中，ISP 提倡不要将一个大而全的接口扔给使用者，而是将每个使用者关注的接口进行隔离。

依赖倒置原则（DIP）

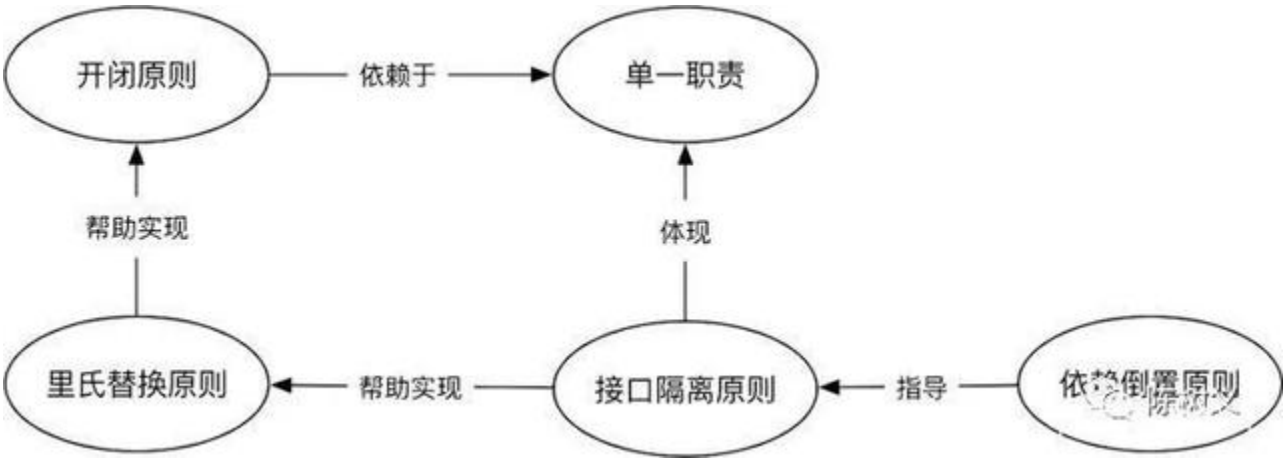
依赖倒置原则（Dependence Inversion Principle）的定义是：高层模块不应该依赖底层模块，两者都应该依赖其抽象。抽象不应该依赖细节，即接口或抽象类不依赖于实现类。细节应该依赖抽象，即实现类不应该依赖于接口或抽象类。简单地说，就是说我们应该面向接口编程。通过抽象成接口，使各个类的实现彼此独立，实现类之间的松耦合。

如果我们每个人都能通过接口编程，那么我们只需要约定好接口定义，我们就可以很好地合作了。软件设计的 DIP 提倡使用者依赖一个抽象的服务接口，而不是去依赖一个具体的服务执行者，从依赖具体实现转向到依赖抽象接口，倒置过来。

02 SOLID 原则的本质

我们总算把 SOLID 原则中的五个原则说完了。但说了这么一通，好像是懂了，但是好像什么都没记住。那么我们就来盘一盘他们之间的关系。ThoughtWorks 上有一篇文章说得挺不错，文中说：

单一职责是所有设计原则的基础，开闭原则是设计的终极目标。里氏替换原则强调的是子类替换父类后程序运行时的正确性，它用来帮助实现开闭原则。而接口隔离原则用来帮助实现里氏替换原则，同时它也体现了单一职责。依赖倒置原则是过程式编程与面向对象编程的分水岭，同时它也被用来指导接口隔离原则。



2.JS是如何实现 发布订阅者模式的 在项目中用到过么 js中是先发布还是订阅 如何实现先发布

```
var Event = (function(){
    var clientList = {},/ /
          listen,
         trigger,
         remove;
    //
    listen = function( key, fn ){
        if ( !this.clientList[ key ] ){  //
            this.clientList[ key ] = [];
        }
        this.clientList[ key ].push( fn ); //
     },
    //
     trigger =  function(){
        var key = Array.prototype.shift.call( arguments ),  //
        fns = this.clientList[ key ];//
        if ( !fns || fns.length === 0 ){  //
            return false;
        }
        for( var i = 0, fn; fn = fns[ i++ ]; ){
            fn.apply( this, arguments ); // (2) // arguments  trigger
        }
     }
   //
   remove = function( key, fn ){
     var fns = clientList[ key ];
     if ( !fns ){ return false; } //  key
     if ( !fn ){ // key .
         fns && ( fns.length = 0 );
     }else{   //
          for ( var l = fns.length - 1; l >=0; l-- ){  //
               var _fn = fns[ l ];
               if ( _fn === fn ){  //
                    fns.splice( l, 1 );
               }
          }
     }
  };
  return {
     listen: listen,
     trigger: trigger,
     remove: remove
  }
})();
```

3.JS是如何实现职责链

```
Function.prototype.after = function (fn) {
        let self = this;
        return async function () {
            let ret = await self.apply(this, arguments);
            if (ret === "nextSuccessor") {
                return fn.apply(this, arguments);
            }
            return ret;
        }
    };
    Function.prototype.before = function (fn) {
        let self = this;
        return function () {
            fn.apply(this, arguments);
            return self.apply(this, arguments);
        }
    }
```

4.

工厂模式 （创建对象）

也是一种是软件工程领域一种广为人知的设计模式

：在一个函数内创建一个空对象，给空对象添加属性和属性值，return这个对象。然后调用这个函数并传入参数来使用。

```
function createPerson(name, age, job){
    var o = new Object();
    o.name = name;
    o.age = age;
    o.sayName = function(){ alert(this.name); };
    return o;
}
var person1 = createPerson("xm", 22);
var person2 = createPerson("Greg", 27);
console.log(person1.name) //xm
console.log(person1.sayName()) //xm
```

：解决了创建 多个相似对象的问题
：没有解决对象识别的问题（即怎样知道一个对象的类型）

构造函数模式（创建对象）

：创建一个构造函数，然后用new 创建构造函数的实例。

```
function Person(name, age, job){  //
    this.name = name;
    this.age = age;
    this.sayName = function(){
        console.log(this.name);
    };
}

var person1 = new Person("xm", 22);
var person2 = new Person("Greg", 27);
console.log(person1.name) //xm
console.log(person1.sayName()) //xm
```

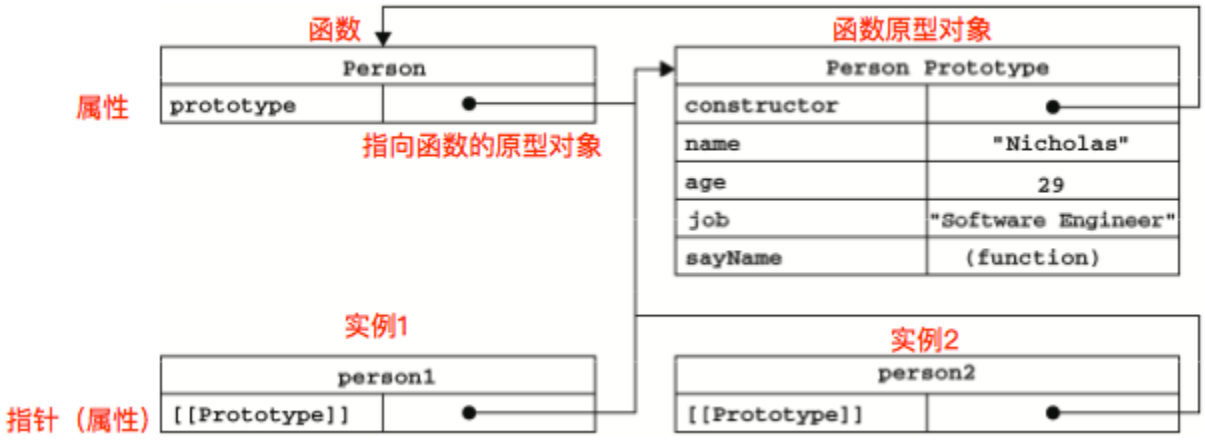
：1.子类型构造函数中可向超类型构造函数传递参数。

2.方法都在构造函数中定义，对于属性值是引用类型的就可通过在每个实例上重新创建一遍，避免所有实例的该属性值指向同一堆内存地址，一个改其他也跟着改。
：对于一些可共用的属性方法（比如这边的this.sayName）没必要都在每个实例上重新创建一遍，占用内存。(无法复用)

原型模式（创建对象）

：创建一个函数，给函数原型对象赋值。

具体一点就是利用函数的prototype属性指向函数的原型对象，从而在原型对象添加所有实例可共享的属性和方法。




```
function Person(){ }
Person.prototype.name = "xm";
Person.prototype.age = 22;
Person.prototype.sayName = function(){
     console.log(this.name);
};
var person1 = new Person();
console.log(person1.name) //xm
console.log(person1.sayName()) //xm
```

：可以让所有对象实例共享它所包含的属性和方法(复用性)。
：1.在创建子类型的实例时，不能向超类型的构造函数中传递参数。

2.如果包含引用类型值的属性，那一个实例改了这个属性（引用类型值），其他实例也跟着改变。

组合使用构造函数模式和原型模式（创建对象，推荐）

：构造函数模式用于定义实例属性，而原型模式用于定义方法和共享的属性。简单来说就是属性值是引用类型的就用构造函数模式，方法和属性能共享的就用原型模式，取精去糟。

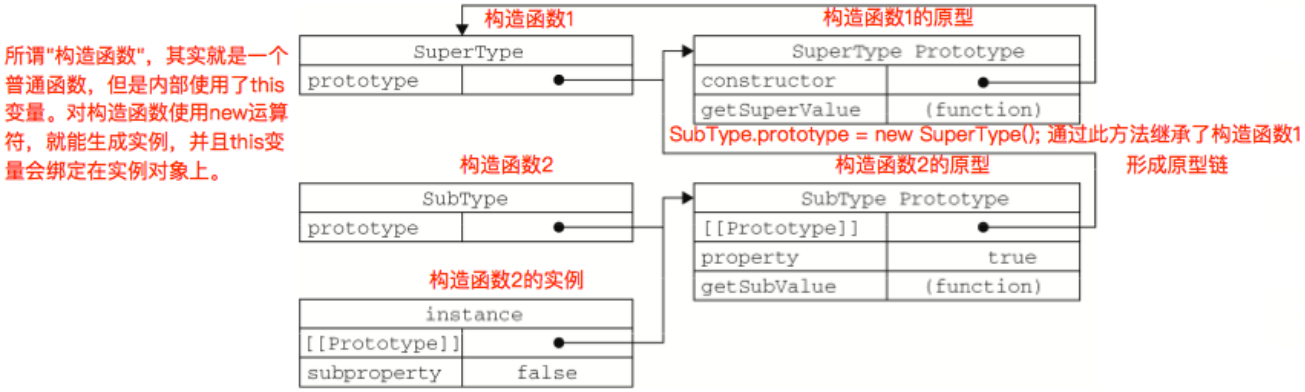
```
function Person(name, age){ //
    this.name = name;
    this.age = age;
    this.friends = ["aa", "bb"];
}
Person.prototype = {  //
    constructor : Person,
    sayName : function(){
        console.log(this.name);
    }
}
Person.prototype.hobby = {exercise:"ball"}; //

var person1 = new Person("xm", 22);
var person2 = new Person("Greg", 27)
person1.friends.push("cc");
console.log(person1.friends);   //"aa,bb,cc"
console.log(person2.friends);   //"aa,bb"
person1.sayName = function(){console.log(this.age)};
person1.sayName();  //22
person2.sayName();  //Greg
person1.hobby.exercise = "running";
console.log(person1.hobby);  //{exercise: "running"}
console.log(person2.hobby); //{exercise: "running"}
```

：构造函数模式和原型模式的取精去糟。

继承

原型链（继承）

：利用原型让一个引用类型继承另一个引用类型的属性和方法。




```
SubType.prototype = new SuperType();
```

```
function SuperType(){
    this.colors = ["red", "blue", "green"];
    this.property = true;
}

SuperType.prototype.getSuperValue = function(){
  return this.property;
};

function SubType(){};
// SuperType
SubType.prototype = new SuperType();
//
SubType.prototype.getSubValue = function (){
  return this.subproperty;
};
//
SubType.prototype.getSuperValue = function (){
  return false;
};
var instance = new SubType();
console.log(instance.getSuperValue());  // false
console.log(SuperType.prototype.getSuperValue); // ƒ (){ return this.property; }

var instance1 = new SubType();
var instance2 = new SubType();
console.log(instance1.colors); //["red", "blue", "green"]
console.log(instance2.colors);//["red", "blue", "green"]

instance1.colors.push("black");
console.log(instance1.colors);//["red", "blue", "green", "black"]
console.log(instance2.colors);//["red", "blue", "green", "black"]

instance1.property = 'test';
console.log(instance1.property);// 'test'
console.log(instance2.property);// true
```

优点：可以让所有对象实例共享它所包含的属性和方法(复用性)

缺点：
1.在创建子类型的实例时，不能向超类型的构造函数中传递参数。

2.如果包含引用类型值的属性，那一个实例改了这个属性（引用类型值），其他实例也跟着改变。

总结：一个实例的__proto__ 等于这个实例所属的构造函数的prototype

所以解释下下面为何为true？

```
function Point(x, y) {
    this.x = x;
    this.y = y;
}
var myPoint = new Point();
// the following are all true
myPoint.__proto__ == Point.prototype // true
myPoint.__proto__.__proto__ == Object.prototype // true
```

这边是因为myPoint.**proto.**等于Point.prototype，而Point.prototype是Object的实例{}, 所以等于Object.prototype

借用构造函数（继承）

：在子类型构造函数的内部调用超类型构造函数。

```
function SubType(){ //SubType SuperType
     SuperType.call(this,argument);
}
```

```
function SuperType(){
    this.colors = ["red", "blue", "green"];
}
function SubType(){ // SuperType
    SuperType.call(this);
}

var instance1 = new SubType();
instance1.colors.push("black");
console.log(instance1.colors); //"red,blue,green,black"
var instance2 = new SubType();
console.log(instance2.colors); //"red,blue,green"
```

： 1.子类型构造函数中可向超类型构造函数传递参数。

2.方法都在构造函数中定义，对于属性值是引用类型的就可通过在每个实例上重新创建一遍，避免所有实例的该属性值指向同一堆内存地址，一个改其他也跟着改。
：无法避免构造函数模式存在的问题——方法都在构造函数中定义（无法复用）；在超类型的原型中定义的方法，对子类型而言也是不可见的。

组合继承

：使用原型链实现对原型属性和方法的继承，而通过借用构造函数来实现对实例属性的继承。

```
function SuperType(name){  //
    this.name = name;
}

SuperType.prototype.sayName = function(){  //
    console.log(this.name);
}

function SubType(name, age){  //
    SuperType.call(this, name);    // this.name = name;
    this.age = age;     //
}

SubType.prototype = new SuperType();    //
SubType.prototype.constructor = SubType;
SubType.prototype.sayAge = function(){
    alert(this.age);
};
var instance1 = new SubType("xm", 22);
console.log(instance1.age)   // 22
console.log(instance1.name)   // xm
instance1.sayName(); //xm
instance1.sayAge(); //22
```