# 设计模式

## 设计原则

**单一职责原则（SRP）**

一个对象或方法只做一件事情。如果一个方法承担了过多的职责，那么在需求的变迁过程中，需要改写这个方法的可能性就越大。

应该把对象或方法划分成较小的粒度

**最少知识原则（LKP）**

一个软件实体应当 尽可能少地与其他实体发生相互作用

应当尽量减少对象之间的交互。如果两个对象之间不必彼此直接通信，那么这两个对象就不要发生直接的 相互联系，可以转交给第三方进行处理

**开放-封闭原则（OCP）**

软件实体（类、模块、函数）等应该是可以 扩展的，但是不可修改

当需要改变一个程序的功能或者给这个程序增加新功能的时候，可以使用增加代码的方式，尽量避免改动程序的源代码，防止影响原系统的稳定

## 什么是设计模式

假设有一个空房间，我们要日复一日地往里 面放一些东西。最简单的办法当然是把这些东西 直接扔进去，但是时间久了，就会发现很难从这 个房子里找到自己想要的东西，要调整某几样东 西的位置也不容易。所以在房间里做一些柜子也许是个更好的选择，虽然柜子会增加我们的成 本，但它可以在维护阶段为我们带来好处。使用 这些柜子存放东西的规则，或许就是一种模式

**学习设计模式，有助于写出可复用和可维护性高的程序**

设计模式的原则是“**找出 程序中变化的地方，并将变化封装起来**”，它的关键是意图，而不是结构。

不过要注意，使用不当的话，可能会事倍功半。

# 一、单例模式

什么是单例模式

单例模式也称作为单子模式，单体模式。单例模式的定义是产生一个类的唯一实例，是软件设计中较为简单但是很常用的一种设计模式。

**单例模式的核心是确保只有一个实例，并提供全局访问，也是单例模式的定义**

实现惰性单例(利用闭包和立即执行函数来实现)

```
// 实现单体模式创建 div

var createDiv= (function(){
        var div;
        return function(){
        if(!div) {
                div = document.createElement("div");
                div.style.width = '100px';
                div.style.height = '100px';
                div.style.background = '#e4e4e4';
                document.body.appendChild(div);
        }
        return div;
        }
})();
var div1=createDiv();
var div2=createDiv();
console.log(div1===div2); //true
```

**1. 定义**

保证一个类仅有一个实例，并提供一个访问它的全局访问点

**2. 核心**

确保只有一个实例，并提供全局访问

**3. 实现**

假设要设置一个管理员，多次调用也仅设置一次，我们可以使用闭包缓存一个内部变量来实现这个单例

```
function SetManager(name) {
```

```
    this.manager = name;
}

SetManager.prototype.getName = function() {
    console.log(this.manager);
};
var SingletonSetManager = (function() {
    var manager = null;

    return function(name) {
        if (!manager) {
            manager = new SetManager(name);
        }

        return manager;
    }
})();

SingletonSetManager('a').getName(); // a
SingletonSetManager('b').getName(); // a
SingletonSetManager('c').getName(); // a
```

这是比较简单的做法，但是假如我们还要设置一个 HR 呢？就得复制一遍代码了

所以，可以改写单例内部，实现地更通用一些

```
// 提取出通用的单例
function getSingleton(fn) {
    var instance = null;

    return function() {
        if (!instance) {
            instance = fn.apply(this, arguments);
        }

        return instance;
    }
}
```

再进行调用，结果还是一样

```
// 获取单例
var managerSingleton = getSingleton(function(name) {
    var manager = new SetManager(name);
    return manager;
});

managerSingleton('a').getName(); // a
managerSingleton('b').getName(); // a
managerSingleton('c').getName(); // a
```

这时，我们添加 HR 时，就不需要更改获取单例内部的实现了，仅需要实现添加 HR 所需要做的，再调用即可

```
function SetHr(name) {
    this.hr = name;
}
```

```
SetHr.prototype.getName = function() {
    console.log(this.hr);
};
var hrSingleton = getSingleton(function(name) {
    var hr = new SetHr(name);
    return hr;
});

hrSingleton('aa').getName(); // aa
hrSingleton('bb').getName(); // aa
hrSingleton('cc').getName(); // aa
```

或者，仅想要创建一个 div 层，不需要将对象实例化，直接调用函数

结果为页面中仅有第一个创建的 div

```
function createPopup(html) {
    var div = document.createElement('div');
    div.innerHTML = html;
    document.body.append(div);

    return div;
}
var popupSingleton = getSingleton(function() {
    var div = createPopup.apply(this, arguments);
    return div;
});

console.log(
    popupSingleton('aaa').innerHTML,
    popupSingleton('bbb').innerHTML,
    popupSingleton('bbb').innerHTML
); // aaa  aaa  aaa
```

总结

单例模式是一种简单但非常实用的模式，特别是惰性单例技术，在合适的时候才创建对像，并且只创建唯一的一个。更奇妙的是，创建对象和管理单例的职责被分布在两个不同的方法中，这两个方法组合起来才具有单例模式的威力。

# 二、策略模式

策略模式的定义是：定义一系列的算法，把它们一个个封装起来，并且使它们可以相互替换。

使用策略模式的优点如下：

优点：1. 策略模式利用组合，委托等技术和思想，有效的避免很多 if 条件语句。

2. 策略模式提供了开放-封闭原则，使代码更容易理解和扩展。

3. 策略模式中的代码可以复用。

**1. 定义**

定义一系列的算法，把它们一个个封装起来，并且使它们可以相互替换。

**2. 核心**

**将算法的使用和算法的实现分离开来。**

一个基于策略模式的程序至少由两部分组成：

第一个部分是一组策略类，策略类封装了具体的算法，并负责具体的计算过程。

第二个部分是环境类 Context，Context 接受客户的请求，随后把请求委托给某一个策略类。要做到这点，说明 Context 中要维持对某个策略对象的引用

**3. 实现**

策略模式可以用于组合一系列算法，也可用于组合一系列业务规则

假设需要通过成绩等级来计算学生的最终得分，每个成绩等级有对应的加权值。我们可以利用对象字面量的形式直接定义这个组策略

比如公司的年终奖是根据员工的工资和绩效来考核的，绩效为 A 的人，年终奖为工资的 4 倍，绩效为 B 的人，年终奖为工资的 3 倍，绩效为 C 的人，年终奖为工资的 2 倍；现在我们使用一般的编码方式会如下这样编写代码：

```
var calculateBouns = function(salary,level) {
    if(level === 'A') {
        return salary * 4;
    }
    if(level === 'B') {
        return salary * 3;
    }
    if(level === 'C') {
        return salary * 2;
    }
};// 调用如下：
console.log(calculateBouns(4000,'A')); // 16000
console.log(calculateBouns(2500,'B')); // 7500
```

第一个参数为薪资，第二个参数为等级；

代码缺点如下：

1. calculateBouns 函数包含了很多 if-else 语句。
2. calculateBouns 函数缺乏弹性，假如还有 D 等级的话，那么我们需要在 calculateBouns 函数内添加判断等级 D 的 if 语句；
3. 算法复用性差，如果在其他的地方也有类似这样的算法的话，但是规则不一样，我们这些代码不能通用。

**2. 使用策略模式重构代码**

策略模式指的是 定义一系列的算法，把它们一个个封装起来，将不变的部分和变化的部分隔开，**实际就是将算法的使用和实现分离出来**；算法的使用方式是不变的，都是根据某个算法取得计算后的奖金数，而算法的实现是根据绩效对应不同的绩效规则；

一个基于策略模式的程序至少由 2 部分组成，第一个部分是一组策略类，策略类封装了具体的算法，并负责具体的计算过程。第二个部分是环境类 Context，该 Context 接收客户端的请求，随后把请求委托给某一个策略类。我们先使用传统面向对象来实现；

```
// 加权映射关系
var levelMap = {
    A: 4,
    B: 3,
    C: 2
};
// 组策略
var strategies= {

   A: function(salary) {
      return salary * levelMap['A'];
   },

   B: function(salary) {
      return salary *levelMap['B'];
   },

   C: function(salary) {
      return salary *levelMap['C'];
   }
}

var calculateBonus = function(level,salary ){
       return strategies[level](salary);
 };
   var getBone = calculateBonus('A',5000);

// 每个策略对象负责的算法已被各自封装在对象内部。
// 当我们对这些策略对象发出“计算奖金”的请求时，它们会返回各自不同的计算结果，
 // 这正是对象多态性的体现，也是“它们可以相互替换”的目
```

在组合业务规则方面，比较经典的是表单的验证方法。这里列出比较关键的部分

```
// 错误提示
var errorMsgs = {
    default: '输入数据格式不正确',
    minLength: '输入数据长度不足',
    isNumber: '请输入数字',
    required: '内容不为空'
};
// 规则集
var rules = {
    minLength: function(value, length, errorMsg) {
        if (value.length < length) {
            return errorMsg || errorMsgs['minLength']
        }
    },
    isNumber: function(value, errorMsg) {
        if (!/\d+/.test(value)) {
            return errorMsg || errorMsgs['isNumber'];
        }
    },
    required: function(value, errorMsg) {
        if (value === '') {
            return errorMsg || errorMsgs['required'];
        }
    }
};
// 校验器
function Validator() {
    this.items = [];
};

Validator.prototype = {
    constructor: Validator,

    // 添加校验规则
    add: function(value, rule, errorMsg) {
        var arg = [value];

        if (rule.indexOf('minLength') !== -1) {
            var temp = rule.split(':');
            arg.push(temp[1]);
            rule = temp[0];
        }

        arg.push(errorMsg);

        this.items.push(function() {
            // 进行校验
            return rules[rule].apply(this, arg);
        });
    },

    // 开始校验
    start: function() {
        for (var i = 0; i < this.items.length; ++i) {
            var ret = this.items[i]();

            if (ret) {
                console.log(ret);
                // return ret;
```

```
            }
        }
    }
};
// 测试数据
function testTel(val) {
    return val;
}
var validate = new Validator();

validate.add(testTel('ccc'), 'isNumber', '只能为数字'); // 只能为数字
validate.add(testTel(''), 'required'); // 内容不为空
validate.add(testTel('123'), 'minLength:5', '最少 5 位'); // 最少 5 位
validate.add(testTel('12345'), 'minLength:5', '最少 5 位');
var ret = validate.start();

console.log(ret);
```

**4. 优缺点**

**优点**

可以有效地避免多重条件语句，将一系列方法封装起来也更直观，利于维护

**缺点**

往往策略集会比较多，我们需要事先就了解定义好所有的情况

# 三、代理模式

**1. 定义**

为一个对象提供一个代用品或占位符，以便控制对它的访问

**2. 核心**

当客户不方便直接访问一个 对象或者不满足需要的时候，提供一个替身对象 来控制对这个对象的访问，客户实际上访问的是 替身对象。

替身对象对请求做出一些处理之后， 再把请求转交给本体对象

代理和本体的接口具有一致性，本体定义了关键功能，而代理是提供或拒绝对它的访问，或者在访问本体之前做一 些额外的事情

**3. 实现**

代理模式主要有三种：保护代理、虚拟代理、缓存代理

保护代理主要实现了访问主体的限制行为，以过滤字符作为简单的例子

# 保护代理

```
// 主体，发送消息
function sendMsg(msg) {
    console.log(msg);
}
// 代理，对消息进行过滤
function proxySendMsg(msg) {
    // 无消息则直接返回
    if (typeof msg === 'undefined') {
        console.log('deny');
        return;
    }

    // 有消息则进行过滤
    msg = ('' + msg).replace(/泥\s*煤/g, '');

    sendMsg(msg);
}


sendMsg('泥煤呀泥 煤呀'); // 泥煤呀泥 煤呀
proxySendMsg('泥煤呀泥 煤'); // 呀
proxySendMsg(); // deny
```

**它的意图很明显，在访问主体之前进行控制，没有消息的时候直接在代理中返回了，拒绝访问主体，这数据保护代理的形式**

**有消息的时候对敏感字符进行了处理，这属于保护代理的模式。**

# 虚拟代理

**虚拟代理在控制对主体的访问时，加入了一些额外的操作**

在滚动事件触发的时候，也许不需要频繁触发，

一般情况一般可以这么处理,暴露全局变量

```
//业务代码
        function scrollHander(){
           console.log('ppp');
        }
        var timerK = null; //污染全局变量了
        function testScroll(){
           clearTimeout(timerK);
           timerK = setTimeout(function(){
```

```
            console.log('pppoooo');
            scrollHander(); //每个页面不同的业务函数
        },300)
    }
    window.onscroll = function(){
        testScroll()
    };
```

我们可以引入函数节流，这是一种虚拟代理的实现

```
// 函数防抖，频繁操作中不处理，直到操作完成之后（再过 delay 的时间）才一次性处理

//业务代码
function scrollHander(name,age){
    console.log(name,age);
    console.log('ppp');
}

//闭包的形式
    function debounce(fn,delay){
        delay = delay || 200;
        var timer=null;

        return function(){
            var arg = arguments;
           // 每次操作时，清除上次的定时器
            clearTimeout(timer);
         // 定义新的定时器，一段时间后进行操作
            timer = setTimeout(function(){
                fn.apply(this,arg);
            },delay);
        }
    }

var proxyScrollHandle = scrollHander;  //无代理

// 代理
 var proxyScrollHandle = debounce(scrollHander,500);  //有代理

//var proxyScrollHandle = (function() {
 //   return debounce(scrollHander, 500);
//})();   //有代理


 // window.onscroll = proxyScrollHandle;   //1
window.onscroll = function(){   //2
```

```
        proxyScrollHandle('laney','age');
   };
//1 和 //2 等价

// 函数的静态属性
 function debounce2(fn,delay){
       delay = delay || 200;
       console.log('0000');
       clearTimeout(fn.id);
       fn.id = setTimeout(function(){
            fn();
       },delay);
         }
   var proxyScrollHandle = debounce2(scrollHander,500);  //有代理，
       // // window.onscroll = debounce2(scrollHander,500); //不能工作
        window.onscroll = function(){  //能工作
           debounce2(scrollHander,500)
        };
```

# 缓存代理

**缓存代理可以为一些开销大的运算结果提供暂时的缓存，提升效率**

来个栗子，缓存加法操作

```
function add() {
    var arg = [].slice.call(arguments);
    //Array.prototype.slice.call(arguments)
    return arg.reduce(function(a, b) {
        return a + b;
    });
}

// 主体
// 代理
var proxyAdd = (function() {
    var cache = [];

    return function() {
        var arg = [].slice.call(arguments).join(',');

        // 如果有，则直接从缓存返回
        if (cache[arg]) {
            return cache[arg];
        } else {
            var ret = add.apply(this, arguments);
```

```
            return ret;
        }
    };
})();

console.log(
    add(1, 2, 3, 4),
    add(1, 2, 3, 4),

    proxyAdd(10, 20, 30, 40),
    proxyAdd(10, 20, 30, 40)
); // 10 10 100 100
```

# 四、迭代器模式

**1. 定义**

迭代器模式是指提供一种方法顺序访问一个聚合对象中的各个元素，而又不需要暴露该对象的内部表示。

**2. 核心**

在使用迭代器模式之后，即使不关心对象的内部构造，也可以按顺序访问其中的每个元素

**3. 实现**

JS 中数组的 map forEach 已经内置了迭代器

```
[1, 2, 3].forEach(function(item, index, arr) {
    console.log(item, index, arr);
});
```

不过对于对象的遍历，往往不能与数组一样使用同一的遍历代码

我们可以封装一下

```
function each(obj, cb) {
    var value;

    if (Array.isArray(obj)) {
        for (var i = 0; i < obj.length; ++i) {
            value = cb.call(obj[i], i, obj[i]);

            if (value === false) {
                break;
            }
        }
    } else {
        for (var i in obj) {
            value = cb.call(obj[i], i, obj[i]);
```

```
            if (value === false) {
                break;
            }
        }
    }
}

each([1, 2, 3], function(index, value) {
    console.log(index, value);
});

each({a: 1, b: 2}, function(index, value) {
    console.log(index, value);
});
// 0 1
// 1 2
// 2 3
// a 1
// b 2
```

再来看一个例子，强行地使用迭代器，来了解一下迭代器也可以替换频繁的条件语句

虽然例子不太好，但在其他负责的分支判断情况下，也是值得考虑的

```
function getManager() {
    var year = new Date().getFullYear();

    if (year <= 2000) {
        console.log('A');
    } else if (year >= 2100) {
        console.log('C');
    } else {
        console.log('B');
    }
}

getManager(); // B
```

将每个条件语句拆分出逻辑函数，放入迭代器中迭代

```
function year2000() {
    var year = new Date().getFullYear();

    if (year <= 2000) {
        console.log('A');
    }

    return false;
}
function year2100() {
    var year = new Date().getFullYear();

    if (year >= 2100) {
        console.log('C');
    }
```

```
    return false;
}
function year() {
    var year = new Date().getFullYear();

    if (year > 2000 && year < 2100) {
        console.log('B');
    }

    return false;
}
function iteratorYear() {
    for (var i = 0; i < arguments.length; ++i) {
        var ret = arguments[i]();

        if (ret !== false) {
            return ret;
        }
    }
}
var manager = iteratorYear(year2000, year2100, year); // B
```

# 五、发布-订阅模式

**1. 定义**

也称作观察者模式，定义了对象间的一种一对多的依赖关系，当一个对象的状态发 生改变时，所有依赖于它的对象都将得到通知

**2. 核心**

取代对象之间硬编码的通知机制，一个对象不用再显式地调用另外一个对象的某个接口。

与传统的发布-订阅模式实现方式（将订阅者自身当成引用传入发布者）不同，在 JS 中通常使用注册回调函数的形式来订阅

**3. 实现**

JS 中的事件就是经典的发布-订阅模式的实现

```
// 订阅
document.body.addEventListener('click', function() {
    console.log('click1');
}, false);

document.body.addEventListener('click', function() {
    console.log('click2');
}, false);
// 发布
document.body.click(); // click1  click2
```

自己实现一下

小 A 在公司 C 完成了笔试及面试，小 B 也在公司 C 完成了笔试。他们焦急地等待结果，每隔半天就电话询问公司 C，导致公司 C 很不耐烦。

一种解决办法是 AB 直接把联系方式留给 C，有结果的话 C 自然会通知 AB

这里的“询问”属于显示调用，“留给”属于订阅，“通知”属于发布

```
// 观察者
var observer = {
    // 订阅集合
    subscribes: [],

    // 订阅
    subscribe: function(type, fn) {
        if (!this.subscribes[type]) {
            this.subscribes[type] = [];
        }

        // 收集订阅者的处理
        typeof fn === 'function' && this.subscribes[type].push(fn);
    },

    // 发布  可能会携带一些信息发布出去
    publish: function() {
        var type = [].shift.call(arguments),
            fns = this.subscribes[type];

        // 不存在的订阅类型，以及订阅时未传入处理回调的
        if (!fns || !fns.length) {
            return;
        }

        // 挨个处理调用
        for (var i = 0; i < fns.length; ++i) {
            fns[i].apply(this, arguments);
        }
    },

    // 删除订阅
    remove: function(type, fn) {
        // 删除全部
        if (typeof type === 'undefined') {
            this.subscribes = [];
            return;
        }

        var fns = this.subscribes[type];

        // 不存在的订阅类型，以及订阅时未传入处理回调的
        if (!fns || !fns.length) {
            return;
        }
```

```
        if (typeof fn === 'undefined') {
            fns.length = 0;
            return;
        }

        // 挨个处理删除
        for (var i = 0; i < fns.length; ++i) {
            if (fns[i] === fn) {
                fns.splice(i, 1);
            }
        }
    }
};
// 订阅岗位列表
function jobListForA(jobs) {
    console.log('A', jobs);
}
function jobListForB(jobs) {
    console.log('B', jobs);
}
// A 订阅了笔试成绩
observer.subscribe('job', jobListForA);
// B 订阅了笔试成绩
observer.subscribe('job', jobListForB);

// A 订阅了笔试成绩
observer.subscribe('examinationA', function(score) {
    console.log(score);
});
// B 订阅了笔试成绩
observer.subscribe('examinationB', function(score) {
    console.log(score);
});
// A 订阅了面试结果
observer.subscribe('interviewA', function(result) {
    console.log(result);
});

observer.publish('examinationA', 100); // 100
observer.publish('examinationB', 80); // 80
observer.publish('interviewA', '备用'); // 备用
observer.publish('job', ['前端', '后端', '测试']); // 输出 A 和 B 的岗位

// B 取消订阅了笔试成绩
observer.remove('examinationB');
// A 都取消订阅了岗位
observer.remove('job', jobListForA);

observer.publish('examinationB', 80); // 没有可匹配的订阅，无输出
observer.publish('job', ['前端', '后端', '测试']); // 输出 B 的岗位
```

#### 4. 优缺点

**优点**

一为时间上的解耦，二为对象之间的解耦。可以用在异步编程中与 MV*框架中

**缺点**

创建订阅者本身要消耗一定的时间和内存，订阅的处理函数不一定会被执行，驻留内存有性能开销

弱化了对象之间的联系，复杂的情况下可能会导致程序难以跟踪维护和理解

# 六、模板方法模式

**1. 定义**

模板方法模式由两部分结构组成，第一部分是抽象父类，第二部分是具体的实现子类。

**2. 核心**

在抽象父类中封装子类的算法框架，它的 init 方法可作为一个算法的模板，指导子类以何种顺序去执行哪些方法。

由父类分离出公共部分，要求子类重写某些父类的（易变化的）抽象方法

**3. 实现**

**模板方法模式一般的实现方式为继承**

以运动作为例子，运动有比较通用的一些处理，这部分可以抽离开来，在父类中实现。具体某项运动的特殊性则有子类来重写实现。

最终子类直接调用父类的模板函数来执行

```
// 体育运动
function Sport() {

}

Sport.prototype = {
    constructor: Sport,

    // 模板，按顺序执行
    init: function() {
        this.stretch();
        this.jog();
        this.deepBreath();
        this.start();

        var free = this.end();

        // 运动后还有空的话，就拉伸一下
        if (free !== false) {
            this.stretch();
        }

    },

    // 拉伸
    stretch: function() {
        console.log('拉伸');
```

```
    },

    // 慢跑
    jog: function() {
        console.log('慢跑');
    },

    // 深呼吸
    deepBreath: function() {
        console.log('深呼吸');
    },

    // 开始运动
    start: function() {
        throw new Error('子类必须重写此方法');
    },

    // 结束运动
    end: function() {
        console.log('运动结束');
    }
};
// 篮球
function Basketball() {
    Sport.apply(this,arguments)
}

//Basketball.prototype = new Sport();
Basketball.prototype = Object.create(Sport.prototype);
// 重写相关的方法
Basketball.prototype.start = function() {
    console.log('先投上几个三分');
};

Basketball.prototype.end = function() {
    console.log('运动结束了，有事先走一步');
    return false;
};

// 马拉松
function Marathon() {

}

Marathon.prototype =  Object.create(Sport.prototype);
var basketball = new Basketball();
var marathon = new Marathon();
// 子类调用，最终会按照父类定义的顺序执行
basketball.init();
marathon.init();
```

# 七、外观模式

**1. 定义**

为子系统中的一组接口提供一个一致的界面，定义一个高层接口，这个接口使子系统更加容易使用

**2. 核心**

可以通过请求外观接口来达到访问子系统，也可以选择越过外观来直接访问子系统

**3. 实现**

外观模式在 JS 中，可以认为是一组函数的集合

```
// 三个处理函数
function start() {
    console.log('start');
}
function doing() {
    console.log('doing');
}
function end() {
    console.log('end');
}
// 外观函数，将一些处理统一起来，方便调用
function execute() {
    start();
    doing();
    end();
}

// 调用 init 开始执行
function init() {
    // 此处直接调用了高层函数，也可以选择越过它直接调用相关的函数
    execute();
}

init(); // start doing end
```

# 八、命令模式

**1. 定义**

用一种松耦合的方式来设计程序，使得请求发送者和请求接收者能够消除彼此之间的耦合关系

命令（command）指的是一个执行某些特定事情的指令

**2. 核心**

命令中带有 execute 执行、undo 撤销、redo 重做等相关命令方法，建议显示地指示这些方法名

**3. 实现**

简单的命令模式实现可以直接使用对象字面量的形式定义一个命令

```
var incrementCommand = {
    execute: function() {
        // something
```

```
 }

};
```

再实例化进行测试，模拟执行、撤销、重做的操作

```
var incrementCommand = new IncrementCommand();
    // 模拟事件触发，执行命令
    var eventTrigger = {
     // 某个事件的处理中，直接调用命令的处理方法
     increment: function() {
         incrementCommand.execute();
     },

     incrementUndo: function() {
         incrementCommand.undo();
     },

     incrementRedo: function() {
         incrementCommand.redo();
     }
};


eventTrigger['increment'](); // 2
eventTrigger['increment'](); // 4
eventTrigger['incrementUndo'](); // 2
eventTrigger['increment'](); // 4
eventTrigger['incrementUndo'](); // 2
eventTrigger['incrementUndo'](); // 0
eventTrigger['incrementUndo'](); // 无输出
eventTrigger['incrementRedo'](); // 2
eventTrigger['incrementRedo'](); // 4
eventTrigger['incrementRedo'](); // 无输出
eventTrigger['increment'](); // 6
```

此外，还可以实现简单的宏命令（一系列命令的集合）

```
var MacroCommand = {
    commands: [],

    add: function(command) {
        this.commands.push(command);

        return this;
    },

    remove: function(command) {
        if (!command) {
            this.commands = [];
            return;
        }

        for (var i = 0; i < this.commands.length; ++i) {
            if (this.commands[i] === command) {
                this.commands.splice(i, 1);
            }
        }
    },
```

```
    execute: function() {
        for (var i = 0; i < this.commands.length; ++i) {
            this.commands[i].execute();
        }
    }
};
var showTime = {
    execute: function() {
        console.log('time');
    }
};
var showName = {
    execute: function() {
        console.log('name');
    }
};
var showAge = {
    execute: function() {
        console.log('age');
    }
};

MacroCommand.add(showTime).add(showName).add(showAge);

MacroCommand.remove(showName);

MacroCommand.execute(); // time age
```

# 九、组合模式

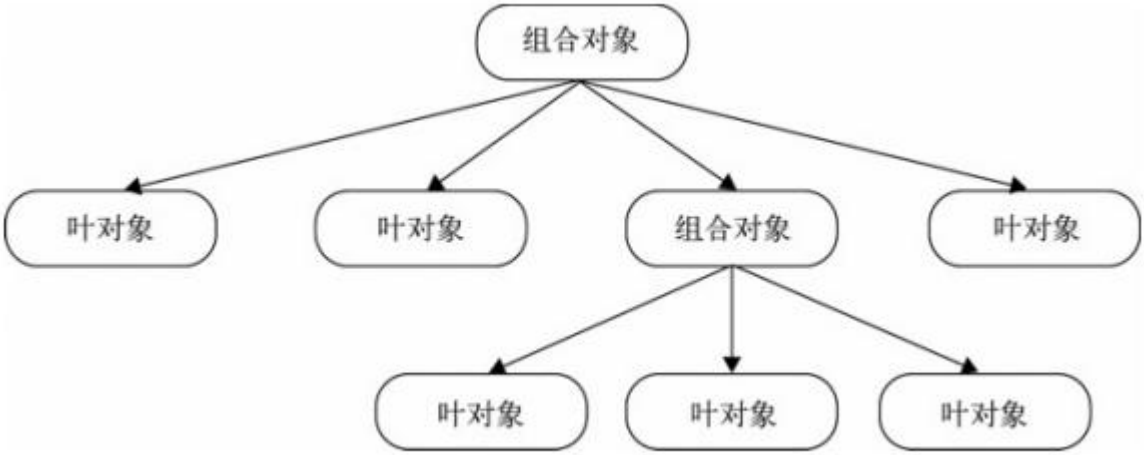
**1. 定义**

是用小的子对象来构建更大的 对象，而这些小的子对象本身也许是由更小 的“孙对象”构成的。

**2. 核心**

可以用树形结构来表示这种“部分- 整体”的层次结构。

调用组合对象 的 execute 方法，程序会递归调用组合对象 下面的叶对象的 execute 方法




但要注意的是，组合模式不是父子关系，它是一种 HAS-A（聚合）的关系，将请求委托给 它所包含的所有叶对象。基于这种委托，就需要保证组合对象和叶对象拥有相同的 接口

此外，也要保证用一致的方式对待 列表中的每个叶对象，即叶对象属于同一类，不需要过多特殊的额外操作

**3. 实现**

使用组合模式来实现扫描文件夹中的文件

```
// 文件夹 组合对象
function Folder(name) {
    this.name = name;
    this.parent = null;
    this.files = [];
}

Folder.prototype = {
    constructor: Folder,

    add: function(file) {
        file.parent = this;
        this.files.push(file);

        return this;
    },

    scan: function() {
        // 委托给叶对象处理
        for (var i = 0; i < this.files.length; ++i) {
            this.files[i].scan();
        }
```

```
    },

    remove: function(file) {
        if (typeof file === 'undefined') {
            this.files = [];
            return;
        }

        for (var i = 0; i < this.files.length; ++i) {
            if (this.files[i] === file) {
                this.files.splice(i, 1);
            }
        }
    }
};
// 文件叶对象
function File(name) {
    this.name = name;
    this.parent = null;
}

File.prototype = {
    constructor: File,

    add: function() {
        console.log('文件里面不能添加文件');
    },

    scan: function() {
        var name = [this.name];
        var parent = this.parent;

        while (parent) {
            name.unshift(parent.name);
            parent = parent.parent;
        }

        console.log(name.join(' / '));
    }
};
```

构造好组合对象与叶对象的关系后，实例化，在组合对象中插入组合或叶对象

```
var web = new Folder('Web');
var fe = new Folder('前端');
var css = new Folder('CSS');
var js = new Folder('js');
var rd = new Folder('后端');

web.add(fe).add(rd);
var file1 = new File('HTML 权威指南.pdf');
var file2 = new File('CSS 权威指南.pdf');
var file3 = new File('JavaScript 权威指南.pdf');
var file4 = new File('MySQL 基础.pdf');
var file5 = new File('Web 安全.pdf');
var file6 = new File('Linux 菜鸟.pdf');

css.add(file2);
fe.add(file1).add(file3).add(css).add(js);
```

```
rd.add(file4).add(file5);
web.add(file6);

rd.remove(file4);
// 扫描
web.scan();
```

**4. 优缺点**

**优点**

可 以方便地构造一棵树来表示对象的部分-整体 结构。在树的构造最终 完成之后，只需要通过请求树的最顶层对 象，便能对整棵树做统一一致的操作。

**缺点**

创建出来的对象长得都差不多，可能会使代码不好理解，创建太多的对象对性能也会有一些影响

# 十、享元模式

**1. 定义**

享元（flyweight）模式是一种用于性能优化的模式，它的目标是尽量减少共享对象的数量

**2. 核心**

运用共享技术来有效支持大量细粒度的对象。

强调将对象的属性划分为内部状态（属性）与外部状态（属性）。内部状态用于对象的共享，通常不变；而外部状态则剥离开来，由具体的场景决定。

**3. 实现**

在程序中使用了大量的相似对象时，可以利用享元模式来优化，减少对象的数量

举个栗子，要对某个班进行身体素质测量，仅测量身高体重来评判

```
// 健康测量
function Fitness(name, sex, age, height, weight) {
    this.name = name;
    this.sex = sex;
    this.age = age;
    this.height = height;
    this.weight = weight;
}
// 开始评判
Fitness.prototype.judge = function() {
    var ret = this.name + ': ';

    if (this.sex === 'male') {
```

```
        ret += this.judgeMale();
    } else {
        ret += this.judgeFemale();
    }

    console.log(ret);
};
// 男性评判规则
Fitness.prototype.judgeMale = function() {
    var ratio = this.height / this.weight;

    return this.age > 20 ? (ratio > 3.5) : (ratio > 2.8);
};
// 女性评判规则
Fitness.prototype.judgeFemale = function() {
    var ratio = this.height / this.weight;

    return this.age > 20 ? (ratio > 4) : (ratio > 3);
};

var a = new Fitness('A', 'male', 18, 160, 80);
var b = new Fitness('B', 'male', 21, 180, 70);
var c = new Fitness('C', 'female', 28, 160, 80);
var d = new Fitness('D', 'male', 18, 170, 60);
var e = new Fitness('E', 'female', 18, 160, 40);

// 开始评判
a.judge(); // A: false
b.judge(); // B: false
c.judge(); // C: false
d.judge(); // D: true
e.judge(); // E: true
```

评判五个人就需要创建五个对象，一个班就几十个对象

可以将对象的公共部分（内部状态）抽离出来，与外部状态独立。将性别看做内部状态即可，其他属性都属于外部状态。

这么一来我们只需要维护男和女两个对象（使用 factory 对象），而其他变化的部分则在外部维护（使用 manager 对象）

```
// 健康测量
function Fitness(sex) {
    this.sex = sex;
}
// 工厂，创建可共享的对象
var FitnessFactory = {
    objs: [],

    create: function(sex) {
        if (!this.objs[sex]) {
            this.objs[sex] = new Fitness(sex);
        }

        return this.objs[sex];
    }
```

```
};
// 管理器，管理非共享的部分
var FitnessManager = {
    fitnessData: {},

    // 添加一项
    add: function(name, sex, age, height, weight) {
        var fitness = FitnessFactory.create(sex);

        // 存储变化的数据
        this.fitnessData[name] = {
            age: age,
            height: height,
            weight: weight
        };

        return fitness;
    },

    // 从存储的数据中获取，更新至当前正在使用的对象
    updateFitnessData: function(name, obj) {
        var fitnessData = this.fitnessData[name];

        for (var item in fitnessData) {
            if (fitnessData.hasOwnProperty(item)) {
                obj[item] = fitnessData[item];
            }
        }
    }
};
// 开始评判
Fitness.prototype.judge = function(name) {
    // 操作前先更新当前状态（从外部状态管理器中获取）
    FitnessManager.updateFitnessData(name, this);

    var ret = name + ': ';

    if (this.sex === 'male') {
        ret += this.judgeMale();
    } else {
        ret += this.judgeFemale();
    }

    console.log(ret);
};
// 男性评判规则
Fitness.prototype.judgeMale = function() {
    var ratio = this.height / this.weight;

    return this.age > 20 ? (ratio > 3.5) : (ratio > 2.8);
};
// 女性评判规则
Fitness.prototype.judgeFemale = function() {
    var ratio = this.height / this.weight;

    return this.age > 20 ? (ratio > 4) : (ratio > 3);
};

var a = FitnessManager.add('A', 'male', 18, 160, 80);
var b = FitnessManager.add('B', 'male', 21, 180, 70);
```

```
var c = FitnessManager.add('C', 'female', 28, 160, 80);
var d = FitnessManager.add('D', 'male', 18, 170, 60);
var e = FitnessManager.add('E', 'female', 18, 160, 40);
// 开始评判
a.judge('A'); // A: false
b.judge('B'); // B: false
c.judge('C'); // C: false
d.judge('D'); // D: true
e.judge('E'); // E: true
```

不过代码可能更复杂了，这个例子可能还不够充分，只是展示了享元模式如何实现，它节省了多个相似的对象，但多了一些操作。

factory 对象有点像单例模式，只是多了一个 sex 的参数，如果没有内部状态，则没有参数的 factory 对象就更接近单例模式了

# 十一、职责链模式

**1. 定义**

使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系，将这些对象连成一条链，并沿着这条链 传递该请求，直到有一个对象处理它为止

**2. 核心**

请求发送者只需要知道链中的第一个节点，弱化发送者和一组接收者之间的强联系，可以便捷地在职责链中增加或删除一个节点，同样地，指定谁是第一个节点也很便捷

**3. 实现**

以展示不同类型的变量为例，设置一条职责链，可以免去多重 if 条件分支

```
// 定义链的某一项
function ChainItem(fn) {
    this.fn = fn;
    this.next = null;
}

ChainItem.prototype = {
    constructor: ChainItem,

    // 设置下一项
    setNext: function(next) {
        this.next = next;
        return next;
    },

    // 开始执行
    start: function() {
        this.fn.apply(this, arguments);
```

```
    },

    // 转到链的下一项执行
    toNext: function() {
        if (this.next) {
            this.start.apply(this.next, arguments);
        } else {
            console.log('无匹配的执行项目');
        }
    }
};
// 展示数字
function showNumber(num) {
    if (typeof num === 'number') {
        console.log('number', num);
    } else {
        // 转移到下一项
        this.toNext(num);
    }
}
// 展示字符串
function showString(str) {
    if (typeof str === 'string') {
        console.log('string', str);
    } else {
        this.toNext(str);
    }
}
// 展示对象
function showObject(obj) {
    if (typeof obj === 'object') {
        console.log('object', obj);
    } else {
        this.toNext(obj);
    }
}
var chainNumber = new ChainItem(showNumber);
var chainString = new ChainItem(showString);
var chainObject = new ChainItem(showObject);
// 设置链条
chainObject.setNext(chainNumber).setNext(chainString);

chainString.start('12'); // string 12
chainNumber.start({}); // 无匹配的执行项目
chainObject.start({}); // object {}
chainObject.start(123); // number 123
```

这时想判断未定义的时候呢，直接加到链中即可

```
// 展示未定义
function showUndefined(obj) {
    if (typeof obj === 'undefined') {
        console.log('undefined');
    } else {
        this.toNext(obj);
    }
}
var chainUndefined = new ChainItem(showUndefined);
```

```
chainString.setNext(chainUndefined);

chainNumber.start(); // undefined
```

由例子可以看到，使用了职责链后，由原本的条件分支换成了很多对象，虽然结构更加清晰了，但在一定程度上可能会影响到性能，所以要注意避免过长的职责链。

# 十二、装饰者模式

**1. 定义**

以动态地给某个对象添加一些额外的职责，而不会影响从这个类中派生的其他对象。
是一种“即用即付”的方式，能够在不改变对 象自身的基础上，在程序运行期间给对象动态地 添加职责

**2. 核心**

是为对象动态加入行为，经过多重包装，可以形成一条装饰链

**3. 实现**

最简单的装饰者，就是重写对象的属性

```
var A = {
    score: 10
};

A.score = '分数: ' + A.score;
```

可以使用传统面向对象的方法来实现装饰，添加技能

```
function Person() {}

Person.prototype.skill = function() {
    console.log('数学');
};
// 装饰器，还会音乐
function MusicDecorator(person) {
    this.person = person;
}

MusicDecorator.prototype.skill = function() {
    this.person.skill();
    console.log('音乐');
};
// 装饰器，还会跑步
function RunDecorator(person) {
    this.person = person;
}

RunDecorator.prototype.skill = function() {
```

```
    this.person.skill();
    console.log('跑步');
};
var person = new Person();
// 装饰一下
var person1 = new MusicDecorator(person);
person1 = new RunDecorator(person1);

person.skill(); // 数学
person1.skill(); // 数学 音乐 跑步
```

在 JS 中，函数为一等对象，所以我们也可以使用更通用的装饰函数

```
// 装饰器，在当前函数执行前先执行另一个函数
function decoratorBefore(fn, beforeFn) {
    return function() {
        var ret = beforeFn.apply(this, arguments);

        // 在前一个函数中判断，不需要执行当前函数
        if (ret !== false) {
            fn.apply(this, arguments);
        }
    };
}

function skill() {
    console.log('数学');
}
function skillMusic() {
    console.log('音乐');
}
function skillRun() {
    console.log('跑步');
}
var skillDecorator = decoratorBefore(skill, skillMusic);
skillDecorator = decoratorBefore(skillDecorator, skillRun);

skillDecorator(); // 跑步 音乐 数学
```

# 十四、适配器模式

**1. 定义**

是解决两个软件实体间的接口不兼容的问题，对不兼容的部分进行适配

**2. 核心**

解决两个已有接口之间不匹配的问题

**3. 实现**

比如一个简单的数据格式转换的适配器

```
// 渲染数据，格式限制为数组了
function renderData(data) {
    data.forEach(function(item) {
        console.log(item);
    });
}
// 对非数组的进行转换适配
function arrayAdapter(data) {
    if (typeof data !== 'object') {
        return [];
    }

    if (Object.prototype.toString.call(data) === '[object Array]') {
        return data;
    }

    var temp = [];

    for (var item in data) {
        if (data.hasOwnProperty(item)) {
            temp.push(data[item]);
        }
    }

    return temp;
}
var data = {
    0: 'A',
    1: 'B',
    2: 'C'
};

renderData(arrayAdapter(data)); // A B C
```