




JAVASCRIPT设计模式(浅谈)



yuanzhijia@baidu.com

23种设计模式总结

- 
- 单例模式
 - 构造函数模式
 - 建造者模式
 - 工厂模式
 - 装饰者模式
 - 外观模式
 - 代理模式
 - 观察者模式
 - 策略模式
 - 命令模式
 - 迭代器模式

23种设计模式总结

- 
- 享元模式
 - 职责链模式
 - 适配器模式
 - 组合模式
 - 模板模式
 - 原型模式
 - 状态模式
 - 桥接模式
 - 中介者模式

单例模式



● 模式介绍

- 划分命名空间，组织代码（唯一的访问点）。
- 减少全局变量。
- 模块间通信。

● 使用场景及注意事项

- 代码格式清晰可见。
- 注意**this**的使用。
- 闭包容易造成内存泄露，不需要的赶快干掉。
- 注意**new**的成本。（继承）

Single code



```
var SingletonTester = (function () {  
  
    //参数: 传递给单例的一个参数集合  
    function Singleton(args) {  
  
        //设置args变量为接收的参数或者为空 (如果没有提供的话)  
        var args = args || {};  
        //设置name参数  
        this.name = 'SingletonTester';  
        //设置pointX的值  
        this.pointX = args.pointX || 6; //从接收的参数里获取, 或者设置为默认值  
        //设置pointY的值  
        this.pointY = args.pointY || 10;  
  
    }  
  
    //实例容器  
    var instance;  
  
    var _static = {  
        name: 'SingletonTester',  
  
        //获取实例的方法  
        //返回Singleton的实例  
        getInstance: function (args) {  
            if (instance === undefined) {  
                instance = new Singleton(args);  
            }  
            return instance;  
        }  
    };  
    return _static;  
})();
```



构造函数模式



● 模式介绍

- 用于创建特定类型的对象。
- 第一次声明的时候给对象赋值。
- 自己声明构造函数，赋予属性和方法。

● 使用场景及注意事项

- 声明函数的时候处理业务逻辑。
- 配合单例实现初始化。
- 构造函数大写字母开头。（建议）

Constructor code

```
function Car(model, year, miles) {  
    if (!(this instanceof Car)) {  
        return new Car(model, year, miles);  
    }  
    this.model = model;  
    this.year = year;  
    this.miles = miles;  
    this.output = function () {  
        return this.model + "走了" + this.miles + "公里";  
    }  
}  
Car.prototype.output = function () {  
    /*操作*/  
};  
var tom = new Car("zhijia", 2009, 20000);  
var dudu = Car("z", 2010, 5000);  
  
console.log(typeof tom); // "object"  
console.log(tom.output()); // "志佳走了20000公里"  
console.log(typeof dudu); // "object"  
console.log(dudu.output()); // "z走了5000公里"
```

工厂模式(制造模式)



● 模式介绍

- 分为简单工厂和抽象工厂。
- 创建对象的一个接口，延迟子类。
- 具体实现依赖具体的子类。

● 使用场景及注意事项

- 对象的处理十分复杂。
- 需要依赖具体的模块。
- 配合命令模式处理逻辑。
- 复杂模式使用较少。

Factory code

```
var XMLHttpRequestFactory = function(){}; //这是一个简单工厂模式
XMLHttpRequestFactory.createXMLHttpRequest = function(){
    var XMLHttpRequest = null;
    if (window.XMLHttpRequest){
        XMLHttpRequest = new XMLHttpRequest()
    }elseif (window.ActiveXObject){
        XMLHttpRequest = new ActiveXObject("Microsoft.XMLHTTP")
    }
    return XMLHttpRequest;
};
//XMLHttpRequestFactory.createXMLHttpRequest()这个方法根据当前环境的具体情况返回一个XHR对象。

var AjaxHandler = function(){
    var XMLHttpRequest = XMLHttpRequestFactory.createXMLHttpRequest();
    /*...具体的操作...*/
}
```

Factory code (抽象)

```
var XMLHttpRequestFactory = function(){};           //这是一个抽象工厂模式
XMLHttpRequestFactory.prototype = {
    //如果真的要调用这个方法会抛出一个错误，它不能被实例化，只能用来派生子类
    createFactory: function(){
        throw new Error('This is an abstract class');
    }
}
//派生子类，文章开始处有基础介绍那有讲解继承的模式，不明白可以去参考原理
var XMLHttpRequest = function(){
    XMLHttpRequestFactory.call(this);
};
XMLHttpRequest.prototype = new XMLHttpRequestFactory();
XMLHttpRequest.prototype.constructor = XMLHttpRequest;
//重新定义createFactory 方法
XMLHttpRequest.prototype.createFactory = function(){
    var XMLHttpRequest = null;
    if (window.XMLHttpRequest){
        XMLHttpRequest = new XMLHttpRequest();
    }else if (window.ActiveXObject){
        XMLHttpRequest = new ActiveXObject("Microsoft.XMLHTTP")
    }
}
```

代理模式



● 模式介绍

- 为其他对象提供一种代理以控制对这个对象的访问。
- 代理可以是任何对象：文件，资源，内存中的对象，或者是一些难以复制的东西。
- 推迟消耗大资源的实例化。

● 使用场景及注意事项

- 远程代理(一个对象在不同的地址空间提供局部代表)。
- 虚拟代理(根据需要创建开销很大的对象)。
- 安全代理(用来控制真实对象访问时的权限)。
- 智能指引(调用真实对象，代理处理一些事情)。

Proxy code

// 先声明美女对象

```
var girl = function (name) {  
    this.name = name;  
};
```

// 这是dudu

```
var dudu = function (girl) {  
    this.girl = girl;  
    this.sendGift = function (gift) {  
        alert("Hi " + girl.name + ", dudu送你一个礼物: " + gift);  
    }  
};
```

// 大叔是代理

```
var proxyTom = function (girl) {  
    this.girl = girl;  
    this.sendGift = function (gift) {  
        (new dudu(girl)).sendGift(gift); // 替dudu送花咯  
    }  
};  
var proxy = new proxyTom(new girl("酸奶小妹"));  
proxy.sendGift("999朵玫瑰");
```

桥梁模式（桥接模式）



● 模式介绍

- 抽象部分与实现部分相分离，均可对立变化。
- 将多个类连接在一起。
- 让API更加健壮，提高组件模块化程度，促成更简洁的实现，并提高抽象的灵活性。

● 使用场景及注意事项

- 连接公开的API代码和私有的实现代码。
- 业务逻辑复杂，抽象的间接实现，方便维护。
- 注意与代理模式的区别。



缺点：增加开发成本，性能影响！

Bridge code

/*

* 错误的方式

```
addEvent(element,'click',getBeerById);
```

*/

```
function getBeerById(id,callback){  
    asyncRequest('GET','beer.url?id='+ id,function(resp){  
        callback(resp.responseText)  
    });  
}
```

```
addEvent(element,'click',getBeerByIdBridge);
```

```
function getBeerByIdBridge(e){  
    getBeerById(this.id,function(beer){  
        console.log('Requested Beer: '+ beer);  
    });  
}
```

适配器器模式(包装器)



● 模式介绍

- 将一个类的接口转换成客户端希望的接口。
- 将一些不能在一起工作的接口组织在一起工作。

● 使用场景及注意事项

- 一个已经存在的对象，但其方法或属性不符合你的要求。
- 创建一个可复用的对象，可以和其他的不先关的对象或接口的方法或属性不兼容的对象协同工作。
- 想使用已经存在的对象，但是不能对每一个进行原型继承去匹配他的借口。对象适配器可以适配他的父对象接口方法和属性。

Adapter code

```
var clientObject = {  
    str1:'bat',  
    str2:'foo',  
    str3:'baz'  
}  
  
function interfaceMethod(str1,str2,str3){  
    alert(str1)  
}  
  
//配置器函数  
function adapterMethod(o){  
    interfaceMethod(o.str1, o.str2, o.str3);  
}  
  
adapterMethod(clientObject)
```


命令模式



● 模式介绍

- 将函数的封装、请求、调用结合为一体。
- 不同的请求对客户进行参数化。
- 调用具体函数解耦命令对象与接收对象。

● 使用场景及注意事项

- 不想让新增加的类影响到其他类。
- 想实现撤销、恢复等操作。
- 配合命令模式处理逻辑。
- 复杂模式使用较少。

Command code

```
car Calculator={
  add:function(x,y){
    return x+y;
  },
  subtract:function(x,y){
    return x-y;
  },
  multiply:function(x,y){
    return x*y;
  },
  divide:function(x,y){
    return x/y;
  }
}

Calculator.calc =function(command){
  return Calculator[command.type](command.op1,command.opd2)
};
```

建造者模式（构造 模式）



● 模式介绍

- 封装对象隔离出复杂对象的各个部分。
- 保持系统稳定的构建算法。
- 无需知道具体需要展示的部分。
- 表相即回调（`jquery ajax`实现很多回调函数）。

● 使用场景及注意事项

- 分布构建一个复杂的对象。
- 该对象有一个稳定的算法进行支持。
- 解耦了封装过程和具体创建的组件。
- 复杂模式使用较少。

Builders code

```
function getBeerById(id, callback) {  
    // 使用ID来请求数据，然后返回数据。  
    asyncRequest('GET', 'beer.uri?id=' + id, function (resp) {  
        // callback调用 response  
        callback(resp.responseText);  
    });  
}
```

```
var el = document.querySelector('#test');  
el.addEventListener('click', getBeerByIdBridge, false);
```

```
function getBeerByIdBridge(e) {  
    getBeerById(this.id, function (beer) {  
        console.log('Requested Beer: ' + beer);  
    });  
}
```

外观模式（门面模式）



● 模式介绍

- 此模块定义了一个高层接口，这个接口值得这一子系统更加容易使用。
- 所有javascript类库核心原则。

● 使用场景及注意事项

- 封装一些算法于一身，提供统计的门面。
- 消除类和代码之间的耦合。
- 设计初期注意分层。
- 开发阶段注意减少依赖。
- 维护别人的代码，把以前的写到一起成为一个门面。不影响自己新建的模块。

Facade code

```
var mobileEvent = {  
  stop: function (e) {  
    e.preventDefault();  
    e.stopPropagation();  
  }  
};
```

demo1

```
var addEvent=function(el,type,fn){  
  if(window.addEventListener){  
    el.addEventListener(type,fn);  
  }elseif(window.attachEvent){  
    el.attachEvent('on'+type,fn);  
  }else{  
    el['on'+type]=fn;  
  }  
};
```

demo2

装饰者模式



● 模式介绍

- 比继承更有弹性。
- 用于包装同接口对象，通过重写添加新功能。
- 不修改原对象，在派生子类增加方法。
- 运作透明，可以增加其他方法。

● 使用场景及注意事项

- 为类添加属性或方法，但是该类的子类添加并不实际。



缺点：实现的时候需要大量的子类！

Decorator code

```
//创建一个命名空间为myText.Decorations
var myText= {};
myText.Decorations={};
myText.Core=function(myString){
    this.show =function(){return myString;}
}
//第一次装饰
myText.Decorations.addQuestuibMark =function(myString){
    this.show =function(){return myString.show()+'?';};
}
//第二次装饰
myText.Decorations.makeItalic =function(myString){
    this.show =function(){return '<li>'+myString.show()+'</li>';}
}
//得到myText.Core的实例
var theString =new myText.Core('this is a sample test String');
alert(theString.show());    //output 'this is a sample test String'
theString =new myText.Decorations.addQuestuibMark(theString);
```


观察者模式（发布订阅模式）



● 模式介绍

- 定义一对多的关系，多个对象同时监听同一主题对象。
- 主题对象发生变化时，监听对象及时更新。
- 简单的广播机制。
- 页面载入后，观察者与被观察者动态关联，增加灵活性。

● 使用场景及注意事项

- 一个对象改变需要通知其他对象。
- 不知道具体有多少个更新的对象。
- 解耦，让解耦的双方都依赖抽象。
- 借助事件监听器处理各种行为，减少内存的损耗。

Observer code



```
(function ($) {  
  
    var o = $({});  
  
    $.subscribe = function () {  
        o.on.apply(o, arguments);  
    };  
  
    $.unsubscribe = function () {  
        o.off.apply(o, arguments);  
    };  
  
    $.publish = function () {  
        o.trigger.apply(o, arguments);  
    };  
  
} (jQuery));  
//回调函数  
function handle(e, a, b, c) {  
    // `e` 是事件对象，不需要关注  
    console.log(a + b + c);  
};  
  
//订阅  
$.subscribe("/some/topic", handle);  
//发布  
$.publish("/some/topic", ["a", "b", "c"]); // 输出abc  
  
$.unsubscribe("/some/topic", handle); // 退订  
  
//订阅  
$.subscribe("/some/topic", function (e, a, b, c) {  
    console.log(a + b + c);  
});  
  
$.publish("/some/topic", ["a", "b", "c"]); // 输出abc
```

//退订（退订使用的是/some/topic名称，而不是回调函数哦，和版本一的例子不一样



策略模式



● 模式介绍

- 定义算法家族，分别封装起来。
- 算法的变化不会影响到客户的使用算法的客户。

● 使用场景及注意事项

- 做同样的事情，实践不同。
- 不同的事件，应对不同的业务规则。

Strategy code

```
var validator = {  
  
    // 所有可以验证规则处理类存放的地方，后面会单独定义  
    types: {},  
  
    // 验证类型所对应的错误消息  
    messages: [],  
  
    // 当然需要使用的验证类型  
    config: {},  
  
    // 暴露的公开验证方法  
    // 传入的参数是 key => value 对  
    validate: function (data) {  
  
        var i, msg, type, checker, result_ok;  
  
        // 清空所有的错误信息  
        this.messages = [];  
  
        for (i in data) {  
            if (data.hasOwnProperty(i)) {  
  
                type = this.config[i]; // 根据key查询是否有存在的验证规则  
                checker = this.types[type]; // 获取验证规则的验证类  
  
                if (!type) {  
                    continue; // 如果验证规则不存在，则不处理  
                }  
                if (!checker) { // 如果验证规则类不存在，抛出异常  
                    throw {  
                        name: "ValidationError",  
                        message: "No handler to validate type " + type  
                    };  
                }  
  
                result_ok = checker.validate(data[i]); // 使用查到的单个验证类进行验证  
                if (!result_ok) {  
                    msg = "Invalid value for *" + i + "*" + checker.instructions;  
                    this.messages.push(msg);  
                }  
            }  
        }  
    }  
};
```

迭代器模式



● 模式介绍

- 访问一个内聚对象，而无需暴露他的内部表示。
- 为遍历不同的集合提供一个统一的接口。

● 使用场景及注意事项

- 对于集合内部结果常常变化各异，不想暴露内部结构，但是又像让客户透明彻底访问其中的元素。
- 遍历迭代器的时候同时更改迭代器的集合机构，肯能会出现问题。

Iterator code



```
var agg = (function () {  
    var index = 0;  
    data = [1, 2, 3, 4, 5];  
    length = data.length;  
  
    return {  
        next: function () {  
            var element;  
            if (!this.hasNext()) {  
                return null;  
            }  
            element = data[index];  
            index = index + 1;  
            return element;  
        },  
  
        hasNext: function () {  
            return index < length;  
        },  
  
        rewind: function () {  
            index = 0;  
        },  
  
        current: function () {  
            return data[index];  
        }  
    };  
}());
```

// 迭代的结果是: 1,3,5

```
while (agg.hasNext()) {  
    console.log(agg.next());  
}
```



模版模式



● 模式介绍

- 定义一个操作的算法骨架，将一些步骤延迟到子类中。
- 不改变算法的结构，即可重新定义该算法的具体步骤。
- 是一些类库复用的基本技术（提取类库的公共行为）。
- 实现真正的控制反转（spring）。
- 好莱坞法则（别找我们，我们找你）。

● 使用场景及注意事项

- 一次性实现算法不变的部分，可变的行为留给子类实现。
- 提取公共行为，不同之处分离操作。
- 控制子类扩展，末班方法只在特定点调用“hook”操作。

TemplateMethod code

```
var CaffeineBeverage = function () {  
  
};  
CaffeineBeverage.prototype.prepareRecipe = function () {  
    this.boilWater();  
    this.brew();  
    this.pourOnCup();  
    if (this.customerWantsCondiments()) {  
        // 如果可以想加小料，就加上  
        this.addCondiments();  
    }  
};  
CaffeineBeverage.prototype.boilWater = function () {  
    console.log("将水烧开!");  
};  
CaffeineBeverage.prototype.pourOnCup = function () {  
    console.log("将饮料到再杯子里!");  
};  
CaffeineBeverage.prototype.brew = function () {  
    throw new Error("该方法必须重写!");  
};  
CaffeineBeverage.prototype.addCondiments = function () {  
    throw new Error("该方法必须重写!");  
};  
// 默认加上小料  
CaffeineBeverage.prototype.customerWantsCondiments = function () {  
    return true;  
};  
  
// 冲咖啡  
var Coffee = function () {  
    CaffeineBeverage.apply(this);  
};  
Coffee.prototype = new CaffeineBeverage();  
Coffee.prototype.brew = function () {  
    console.log("从咖啡机想咖啡倒进去!");  
};  
Coffee.prototype.addCondiments = function () {  
    console.log("添加糖和牛奶");  
};
```



享元模式



● 模式介绍

- 定义一个操作的算法骨架，将一些步骤延迟到子类中。
- 不改变算法的结构，即可重新定义该算法的具体步骤。
- 是一些类库复用的基本技术（提取类库的公共行为）。
- 实现真正的控制反转（spring）。
- 好莱坞法则（别找我们，我们找你）。

● 使用场景及注意事项

- 一次性实现算法不变的部分，可变的行为留给子类实现。
- 提取公共行为，不同之处分离操作。
- 控制子类扩展，末班方法只在特定点调用“hook”操作。

Flyweight code

//数据量小到没多大的影响，数据量大的时候对计算机内存会产生压力，下面介绍享元模式优化后

//包含核心数据的Car类

```
var Car = function (make, model, year) {  
  this.make = make;  
  this.model = model;  
  this.year = year;  
}
```

```
Car.prototype = {  
  getMake: function () {  
    return this.make;  
  },  
  getModel: function () {  
    return this.model;  
  },  
  getYear: function () {  
    return this.year;  
  }  
}
```

//中间对象，用来实例化Car类

```
var CarFactory = (function () {  
  var createdCars = {};  
  return {  
    createCar: function (make, model, year) {  
      var car = createdCars[make + "-" + model + "-" + year];  
      return car ? car : createdCars[make + '-' + model + '-' + year] = (new Car(make, model, year));  
    }  
  }  
})();
```

//数据工厂，用来处理Car的实例化和整合附加数据

```
var CarRecordManager = (function () {  
  var carRecordDatabase = {};  
  return {  
    addCarRecord: function (make, model, year, owner, tag, renewDate) {  
      var car = CarFactory.createCar(make, model, year);  
      carRecordDatabase[tag] = {  
        owner: owner,  
        tag: tag,  
        renewDate: renewDate,  
        car: car  
      }  
    }  
  }  
})();
```

职责链模式



● 模式介绍

- 使多个对象都有机会处理请求，解耦发送者和接受者。
- 将对象连成一条链，直到有处理的请求为止。
- 候选者的数量是任意的。
- 任意候选者都有权利处理请求。

● 使用场景及注意事项

- 有多个的对象可以处理一个请求，哪个对象处理该请求运行时刻自动确定。
- 想在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。
- 可处理一个请求的对象集合需要被动态指定。

Chain of responsibility code

```
var NO_TOPIC = -1;
var Topic;
function Handler(s, t) {
    this.successor = s || null;
    this.topic = t || 0;
}
Handler.prototype = {
    handle: function () {
        if (this.successor){
            console.log(this.topic);
            this.successor.handle();
        }
    },
    has: function () {
        return this.topic != NO_TOPIC;
    }
};
var app = new Handler({
    handle: function () {
        console.log('我是app');
    }
}, 3);
var dialog = new Handler(app, 1);
dialog.handle = function () {
    console.log('dialog before ...')
    // 这里做具体的处理操作
    Handler.prototype.handle.call(this);
    console.log('dialog after ...')
};
```

组合模式



● 模式介绍

- 一种专为创建web上动态用户界面的模式。
- 擅长处理大批量的对象。
- 将对象组合成树形结构，表示整体与

● 使用场景及注意事项

- 存在一批组织成某处层次体系的对象（具体结构可能在开发期间无法知道）。

Composite code

```
// DynamicGallery Class
var DynamicGallery = function (id) { // 实现Composite, GalleryItem组合对象类
    this.children = [];
    this.element = document.createElement('div');
    this.element.id = id;
    this.element.className = 'dynamic-gallery';
}
DynamicGallery.prototype = {
    // 实现Composite组合对象接口
    add: function (child) {
        this.children.push(child);
        this.element.appendChild(child.getElement());
    },
    remove: function (child) {
        for (var node, i = 0; node = this.getChild(i); i++) {
            if (node == child) {
                this.children.splice(i, 1);
                break;
            }
        }
        this.element.removeChild(child.getElement());
    },
    getChild: function (i) {
        return this.children[i];
    },
    // 实现DynamicGallery组合对象接口
    hide: function () {
        for (var node, i = 0; node = this.getChild(i); i++) {
            node.hide();
        }
        this.element.style.display = 'none';
    },
    show: function () {
        this.element.style.display = 'block';
        for (var node, i = 0; node = this.getChild(i); i++) {
            node.show();
        }
    }
}
```

状态模式



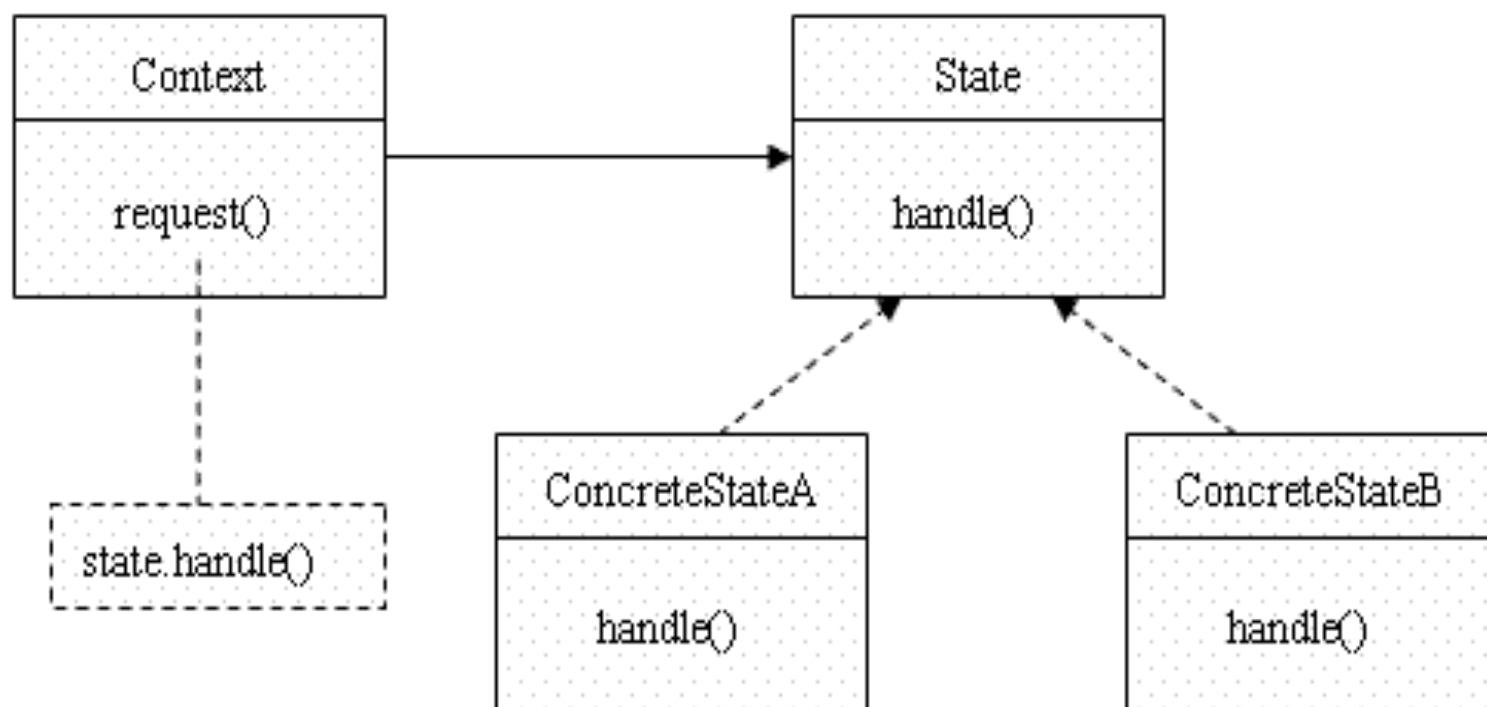
● 模式介绍

- 在状态改变时，改变内部对象的值。
- 类看起来被修改一样。
- 每个状态只做当前状态该做的事。

● 使用场景及注意事项

- 一个对象的行为取决于它的状态，并且它必须在运行时刻根据状态改变它的行为。
- 操作中含有大量的分支语句，而且这些分支语句依赖于该对象的状态。状态通常为一个或多个枚举常量的表示。

State code





结束语



用之无形



剑在心中





谢谢大家！

