单例模式

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

主要解决：一个全局使用的类频繁地创建与销毁

实现原理：用一个变量来标识当前是否已经为某个类创建过对象，如果是，则在下一次获取该类的实例时，直接返回之前创建的对象.

优点：

1. **在内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例**
2. **避免对资源的多重占用**

缺点：没有接口，不能继承，与单一职责原则冲突，一个类应该只关心内部逻辑，而不关心外面怎么样来实例化

使用场景：

1. **可以用它来划分命名空间**
2. **可以把代码组织的更为一致**

```
var single = (function(){
    var unique;
    function getInstance(){
       // 如果该实例存在，则直接返回，否则就对其实例化
        if( !unique ){
            unique = new Construct();
        }
        return unique;
    }

    function Construct(){
        console.log('this is single')
        // ... 生成单例的构造函数的代码
    }

    return {
        getInstance : getInstance
    }
})();
```

策略模式

定义：定义一系列算法，然后逐个封装起来，并且使它们可以相互替换

主要解决：在有多种算法相似的情况下，使用 if...else 所带来的复杂和难以维护

实现方式：通过一个hash对象，来映射不同的策略

优点：算法可以自由切换，避免多重循环，扩展性好

缺点：所有策略类都需要对外暴露，需要了解所有的策略，知道它们之间的不同点

```
const strategy = {
  'A': function(salary) {
    return salary * 4
  },
  'B': function(salary) {
    return salary * 3
  },
  'C': function(salary) {
    return salary * 2
  }
}

const calculateBonus = function(level, salary) {
  return strategy[level](salary)
}

console.log(calculateBonus('A', 30000)) // 120000
```

命令模式

定义：将一个请求封装成一个对象，从而使你可以用不同的请求进行参数化

主要解决：相互之间的耦合

如何解决：通过调用者调用接受者执行命令，顺序：调用者→接受者→命令

使用场景：发布一些命令，但不清楚接受者和请求的操作。命令是指执行某些事情的指令。即只用知道发布了一个指令就行，具体做什么谁来做不用关心。其实是回调函数面向对象的替代品。最常见的例子就是事件绑定了

```
// 发送者：不关心给哪个button绑定，也不关心绑定的是什么方法
const setCommand = function(button, fn) {
    button.onClick = function() {
      fn()
    }
}

// --------------------  上面的界面逻辑由A完成，下面的由B完成

// 执行命令者：无需关心在哪里调用被谁调用，只需要按需执行就好
const menu = {
    updateMenu: function() {
        console.log('更新菜单')
    },
}
```

```
// 命令对象：只需要接收到接受者的参数，当发送者发出命令执行就好
const UpdateCommand = function(receive) {
    return receive.updateMenu()
}

const updateCommand = UpdateCommand(menu) // 创建命令

const button1 = document.getElementById('button1')
setCommand(button1, updateCommand)    // 更新菜单
```

代理模式

定义：为其他对象提供一种代理以控制对这个对象的访问

主要解决：在直接访问对象时带来的问题，比如说：要访问的对象在远程的机器上。在面向对象系统中，有些对象由于某些原因（比如对象创建开销很大，或者某些操作需要安全控制，或者需要进程外的访问），直接访问会给使用者或者系统结构带来很多麻烦，我们可以在访问此对象时加上一个对此对象的访问层。

优点： 1、职责清晰。 2、高扩展性。 3、智能化。

缺点：

1. **由于在客户端和真实主体之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢。**
2. **实现代理模式需要额外的工作，有些代理模式的实现非常复杂**

使用场景：按职责来划分，通常有以下使用场景： 1、远程代理。 2、虚拟代理。 3、保护（Protector Access）代理。 4、Cache代理。 5、防火墙（Firewall）代理。 6、同步化（Synchronization）代理等

虚拟代理实现图片懒加载

```
// 创建图片DOM
const myImage = (function() {
  const imgNode = document.createElement('img')
  document.body.appendChild(imgNode)
  return {
    setSrc: function(src) {
      imgNode.src = src
    }
  }
})()

// 代理
const proxyImage = (function() {
  const img = new Image()
  img.onload = function() { // http 图片加载完毕后才会执行
```

```
    myImage.setSrc(this.src) // 此处的this指的是img,当img加载完成后，将img.src传递给myImage
  }
  return {
    setSrc: function(src) {
      myImage.setSrc('loading.jpg') // 本地 loading 图片
      img.src = src
    }
  }
})()
proxyImage.setSrc('http://loaded.jpg')
```

缓存代理

```
var mult = function() {
    var result = 1;
    for(var i = 0, l = arguments.length; i < l; i++) {
        result= result * arguments[i];
    }
    return result;
};

var proxyFactory = function(fn) {
    var cache = {};
    return function() {
        var args = Array.from(arguments).join(',')
        // 判断当前是否存在缓存里面
        if(args in cache) {
            return cache[args];
        }
        return cache[args] = fn.apply(this, arguments);
    }
};

console.log(proxyFactory(mult)(1, 2, 8));
```

模板模式

定义：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤

主要解决：一些方法通用，却在每一个子类都重新写了这一方法

优点：

1. **封装不变部分，扩展可变部分。**
2. **提取公共代码，便于维护。**

3. 行为由父类控制，子类实现。

缺点：

1. 每一个不同的实现都需要一个子类来实现，导致类的个数增加，使得系统更加庞大

应用场景：主要应用在一些代码刚开始要一次性实现不变的部分。但是将来页面有修改，需要更改业务逻辑的部分或者重新添加新业务的情况。主要是通过子类来改写父类的情况，其他不需要改变的部分继承父类。

```
var Interview = function(){};
// 笔试
Interview.prototype.writtenTest = function(){
    console.log("这里是前端笔试题");
};
// 技术面试
Interview.prototype.technicalInterview = function(){
    console.log("这里是技术面试");
};
// 领导面试
Interview.prototype.leader = function(){
    console.log("领导面试");
};
// HR面试
Interview.prototype.HR = function(){
    console.log("HR面试");
};
// 代码初始化
Interview.prototype.init = function(){
    this.writtenTest();
    this.technicalInterview();
    this.leader();
    this.HR();
};

// 阿里巴巴的笔试和技术面不同，重写父类方法，其他继承父类方法。
var AliInterview = function(){};
AliInterview.prototype = new Interview();

// 子类重写方法 实现自己的业务逻辑
AliInterview.prototype.writtenTest = function(){
    console.log("阿里的技术题就是难啊");
}
AliInterview.prototype.technicalInterview = function(){
    console.log("阿里的技术面就是牛皮啊");
}
var AliInterview = new AliInterview();
AliInterview.init();

// 阿里的技术题就是难啊
```

```
// 阿里的技术面就是牛皮啊
// 领导面试
// HR面试
```