我所理解的面向对象

什么是面向对象?

首先，面向对象并不是说写一个class就是面向对象了。在Java里面Everything is class，全部都是Class，还有React也需要写class，所以很多人写class并不是他自己要写class，而是编程语言或者框架要求他写class。因此就会存在一个窘境，虽然是写的class，但是代码风格是面向结构的，只是套了一个class的外衣，真正面向对象的是所使用的框架。

所以**面向对象应该是一种思想，而不是你代码的组织形式**，甚至有时候你连一个class都没写。

面向对象的英文为Object Oriented，它的准确翻译应该为“面向物件”，而不是“面向对象”，只不过不知道是谁翻译了这么一个看似“高大上”但是不符合实际的名词。面向对象是对世界物件的抽象和封装，例如车子、房子和狗等。

面向对象的特点

面向对象有三个主要的特点：封装、继承和多态。

封装

现在我要研究下人，并且关注它的吃和睡行为，所以我封装了一个人的类，如下代码所示：

```
class Person {
  constructor (name) {
    this.name = name
  }
  eat () {
    console.log(`${this.name}正在大口大口地吃`)
  }
  sleep () {
    console.log(`${this.name}正在睡`)
  }
}
```

上面代码封装两个行为：吃、睡，和一个属性：姓名。

继承

然后我又要研究下小明，如下所示：

```
class Xiaoming extends Person {
  constructor (name) {
    super(name)
  }
  smile () {
    console.log(`${this.name}正在笑`)
  }
}
```

小明是人的一种，我让小明继承了人这个类，于是它就继承了父类的行为，如小明可以吃、睡，同时，小明有他自己的行为，例如他会笑：

```
let xiaoming = new Xiaoming('小明')
xiaoming.eat() // 小明正在大口大口地吃
xiaoming.smile() // 小明正在笑
```

多态

小明也会吃，但是它不是大口大口地吃，他是小口小口地吃，所以同样是吃的行为，但是小米有自己特点，这个就是多态，如下所示：

```
class Xiaoming extends Person {
  constructor (name) {
    super(name)
  }
  eat () {
    console.log(`${this.name}正在小口小口地吃`)
  }
  smile () {
    console.log(`${this.name}正在笑`)
  }
}
```

当调用Xiaoming的eat函数时就是小口小口地而不是大口大口地吃了：

```
let xiaoming = new Xiaoming('小明')
xiaoming.eat() // // 小明正在小口小口地吃
```

面向对象的实际例子

传进度条

一个页面会有多个上传图片的地方，每个上传地方都会生成一个进度条，所以考虑把进度条封装成一个类ProgressBar，如下代码所示：

```
class ProgressBar {
  constructor ($container) {
```

```
    this.fullWidth = $container.width()
    this.$bar = null
  }

  // 设置进度
  setProgress (percentage) {
    this.$bar.animate({width: this.fullWidth * percentage + 'px'})
  }

  // 完成
  finished () {
    this.$bar.hide()
  }

  // 失败
  failed () {
    this.addFailedText()
  }

  addFailedText () {

  }
}
```

ProgressBar封装了设置进度、完成、失败的函数，这个就是面向对象的封装。addFailedText函数是内部的实现，不希望实例直接调用，也就是说它应该是一个私有的、对外不可见的函数。但是由于JS没有私有属性、私有函数的概念，所以还是可以调的，如果要实现私有属性得通过闭包之类的技巧实现。

接着我想做一个带有百分比数字的进度条，于是我想到了面向对象的继承，写一个ProgressBarWithNumber的类，继承ProgressBar，如下代码所示：

```
class ProgressBarWithNumber extends ProgressBar {
  constructor ($container) {
    super($container)
  }

  // 多态
  setProgress (percentage) {
    // 先借助继承的父类的函数
    super.setProgress(percentage)
    this.showPercentageText(percentage)
  }

  showPercentageText (percentage) {

  }
}
```

子类继承了父类的函数，同时覆盖/实现了父类的某些行为。上面的setProgress函数既体现了多态又体现了继承。

继承和组合

继承是为了实现复用，组合其实也是为了实现复用。继承是is-a的关系，而组合是has-a的关系。可以把上面的ProgressBar改成组合的方式，如下代码所示：

```
class ProgressBarWithNumber {
  constructor ($container) {
    this.progressBar = new ProgressBar($container)
  }

  setProgress (percentage) {
    this.progressBar.setProgress(percentage)
    this.showPercentageText(percentage)
  }

  showPercentageText (percentage) {

  }
}
```

在构造函数里面组合了一个progressBar的实例，然后在setProgress函数里面利用这个实例去设置进度条的百分比。

也就是说带有数字的进度条里面有一条普通的进度条，这是组合，而当我们用继承的时候就变成了带数字的进度条是一种进度条。这两个都说得通。那么是继承好用一点，还是组合好用一点呢?

在《Effective Java》里面有一个条款：

> Item 16 : Favor composition over inheritance

意思为**偏向于使用组合而非继承**，为什么说组合比较好呢？因为继承的耦合性要大于组合，组合更加灵活。继承是编译阶段就决定了关系，而组合是运行阶段才决定关系。组合可以组合多个，而如果要搞多重继承系统的复杂性无疑会大大增加。

就上面的进度条的例子来说，使用组合会比使用继承的方式好吗？假设某一天，带数字的进度条不想复用普通的进度条了，要复用另外一种类型的进度条，使用继承就得改它的继承关系，万一带数字的进度条还派生了另外一个类，这个孙子类如果刚好用了普通进度条的一个函数，那这个条链就断了，导致孙子类也要改。所以可以看出组合的方式更加简易，继承相对比较复杂。

但是如果要我在这之上加一个条款的话我会这么加：

> Item 0: Favor Simple Ways over OOP

因为能用简单的方式解决问题就应该用简单的方式，而不是一着手就是各种面向对象的继承、多态的思想，带数字的LoadingBar其实不需要使用继承或者组合，只要带一个参数控制就好了，是否要显示数字。笔者认为应该先使用简洁的方式解决问题，然后再考虑性能、代码组织优化等。为了5%的效果，增加了系统50%的复杂度，其实不值得，除非那个问题是瓶颈问题，能够提升一点是一点。为了写一个小需求，封装了几十个类，最后需求一变这几十个类就都没用了。

接着重点说一下设计模式和OOP的编程原则。

面向对象编程原则和设计模式

单例模式

单例是一种比较简单也是比较常见的模式。例如现在要定义Task类，要实现它的单例，因为全局只能有一个数组存放Task，如果有任务就都放到这个队列里面，按先进先出的顺序执行。

于是我先写一个Task类：

```
class Task {
  constructor () {
    this.tasks = []
  }

  // 初始化
  draw () {
    var that = this
    window.requestAnimationFrame(function () {
      if (that.tasks.length) {
        var task = that.tasks.shift()
        task()
      }
    })
  }

  addTask (task) {
    this.tasks.push(task)
  }
}
```

现在要实现它的单例，可以这么实现：

```
var mapTask = {
  get: function () {
    if (!mapTask.aTask) {
      mapTask.aTask = new Task()
      mapTask.aTask.draw()
    }
    return this.aTask
  },
  add: function (task) {
    mapTask.get().addTask(task)
  }
}
```

每次get的时候先判断有mapTask有没有Task的实例了，如果没有则为第一次，先去实例化一个，并做些初始化工作，如果有则直接返回。然后执行mapTask.get()的时候就能够保证获取到的是一个单例。

但是这种实现其实不太安全，任何人可通过设置：

```
mapTask.aTask = null
```

去破坏你这个单例，那怎么办呢？一方面JS本身没有私有属性，另一方面要怎么解决留给读者去思考。

因为JS的Object本身就是单例的，所以可以把Task类改成一个taskWorker，如下代码所示：

```
var taskWorker = {
  tasks: [],
  draw () {

  },
  addTask (task) {
    Task.tasks.push(task)
  }
}

var mapTask = {
  add: function (task) {
    taskWorker.addTask(task)
  }
}
```

显然第二种方式比较简单，但是它只能有一个全局的task。而第一种办法可以拥有几种不同业务的Task，不同业务互不影响。例如除了mapTask之外，还可以再写一个searchTask的业务。

策略模式

这个例子已经提过很多次，这里再简单提一下。假设现在要弹几个注册的框，每个注册的框只是顶部的文案不一样，而其它地方包括逻辑等都一样，所以，我就把文案当作一个个的策略，使用的时候根据不同的类型，映射到不同的策略，如下图所示：

```
var popType = {
    userReg: {
        title: 'Create Your Account',
    },
    favHouse: {
        title: 'Add Home to Favorite'
    },
    saveSearch: {
        title: 'Save this search'
    }
}

var tpl = `
<section>
    <h1>{{title}}</h1>
</section>`;

Mustache.render(tpl, popType['userReg']);
```

把不同的文案当作一个个策略

根据不同类型映射到不同的策略

注册完成后需要去执行不同的操作，把这些操作也封装成一个个的策略，同样地根据不同的类型映射到不同的策略，如下图所示：

```
var popCallback = {
    userReg: function(){
        //do nothing
    },
    favHouse: function(){
        //发一个收藏房源的请求
    },
    saveSearch: function(){
        //发一个保存搜索条件的请求
    }
};

util.ajax("/register", function(){
    var popType = "favHouse"; //获取popType
    popCallback[popType]
});
```

把一个回调操作封装成一个策略

根据不同的type映射到不同的策略

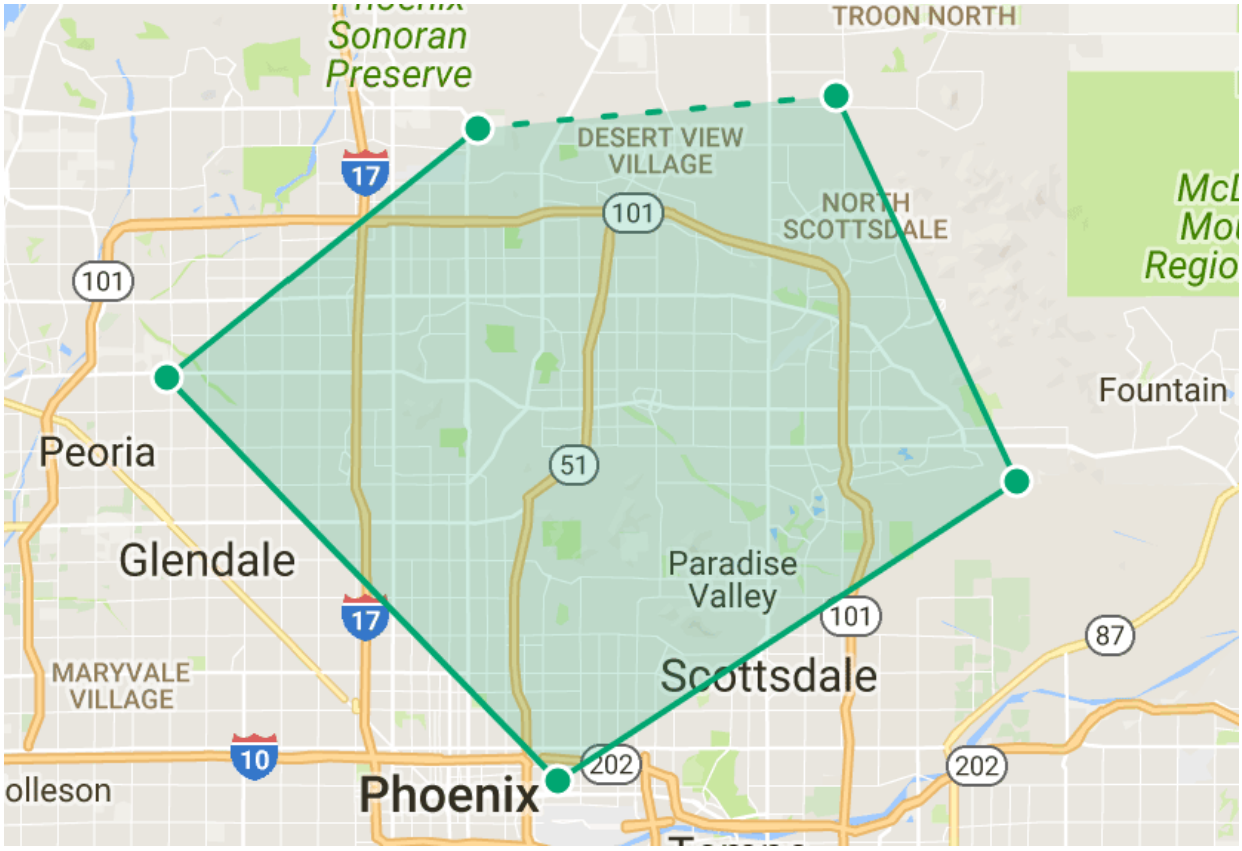
这样比写if-else或者switch-case的好处就在于：如果以后要增加或者删除某种类型的弹框，只需要去增删一个type就可以了，而不用去改动if-else的逻辑。这个就叫做开闭原则——对修改是封闭的，而对扩展是开放的。

观察者模式

观察者模式也是经常和前端打交道的一种模式，事件监听就是一种观察者模式，如下实现一个观察者模式：

```
class Input {
  constructor (inputDom) {
    this.inputDom = inputDom
    this.visitors = {
      'click': [],
    }
  }

  // 添加访问者
  on (eventType, visitor) {
    this.visitors.push(visitor)
  }

  // 收到消息，把消息分发给访问者
  trigger (type, event) {
    if (this.visitors[type]) {
      for (var i = 0; i < this.visitors[type]; i++) {
        this.visitors[type]()
      }
    }
  }
}
```

观察者向消息的接收者订阅消息，一旦接收者收到消息后就把消息下发给它的观察者们。在一个地图绘制搜索的应用里面，点击最后一个点关闭路径，要触发搜索：



但其实不用再去手动调搜索的接口了，因为地图本身就监听了drag_end事件，在这个事件里面会去搜索，所以在绘制完成之后只要执行：

```
map.trigger("drag_end")
```

就可以了，即给drag_end事件的观察者们下发一个消息，让它们去执行。

适配器模式

在一个响应式的页面里面，假设小屏和大屏显示的分页样式不一样，小屏要这样显示：

而大屏要这样显示：

它们初始化和更新状态的函数都不一样，如下所示：

```
//小屏
var wap_pagination = new jqPagination({})
wap_pagination.showPage = function (curPage, totalPage) {
  wap_pagination.setPage(curPage, totalPage)
}

// 大屏
var pc_pagination = new Pagination({})
pc_pagination.showPage = function (curPage, totalPage) {
  pc_pagination.showItem(curPage, totalPage)
```

如果我每次用的时候都得先判断一下不同的屏幕大小然后去调不同的函数就显得有点麻烦，所以可以考虑用一个适配器，对外提供统一的接口，如下所示：

```
var screen = $(window).width() < 800 ? 'small' : 'large'
var paginationAdapter = {
  init: function () {
    this.pagination = screen === 'small'
      ? new jqPagination()
      : new Pagination()
    if (screen === 'large') {
      this.pagination.showItem = this.pagination.setPage
    }
  },
  showPage: function (curPage, totalPage) {
    this.pagination.showItem(curPage, totalPage)
```

```
  }
}
```

使用者只要调一下paginationAdapter.showPage就可以更新分页状态，它不需要去关心当前是大屏还是小屏，由适配器去处理这些细节。

工厂模式

工厂模式是把创建交给一个“工厂”，使用者无需要关心创建细节，如下代码所示：

```
var taskCreator = {
  createTask: function (type) {
    switch (type) {
      case 'map':
        return new MapTask()
      case 'search':
        return new SearchTask()
    }
  }
}

var mapTask = taskCreator.createTask('map')
```

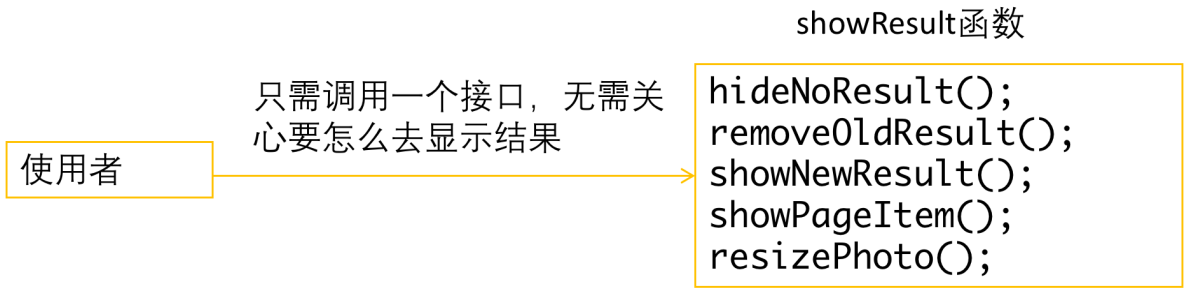
需要哪种类型的Task的时候就传一个类型或者产品名字给一个工厂，工厂根据名字去生产相应的产品给我，而我不需要关心它是怎么创建的，要不要单例之类的。

外观/门面模式

在一个搜索逻辑里面，为了显示搜索结果需要执行以下这么多个操作：

```
hideNoResult()      //先隐藏没有结果的显示
removeOldResult()   //删除老的结果
showNewResult()     //显示新的结果
showPageItem()      //更新分页
resizePhoto()       //结果图片大小重置
```

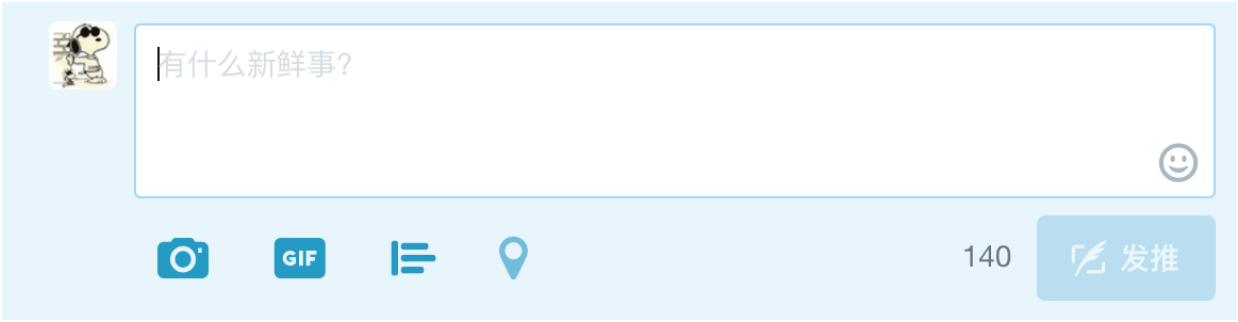
于是考虑用一个模块把它包起来，如下图所示：



把那么多个操作封装成一个模块，对外只提供一个门面叫showResult，使用者只要调一下这个showResult就可以了，它不需要知道究竟要怎么去显示结果。

状态模式

现在要实现一个发twitter的消息框，要求是当字数为0或者超过140的时候，发推按钮不可点击，并且剩余字数会跟着变，如下图所示：



我想用一个state来保存当前的状态，然后当用户输入的时候，这个state的数据会跟着变，同时更新发推按钮的状态，如下代码所示：

```
var tweetBox {
  init() {
    //初始化一个state
    this.state = {}
    tweetBox.bindEvent()
  }
  setState(key, value) {
    this.state[key] = value
  }
  changeSubmit() {
    // 通过获取当前的state
    $('#submit')[0].disabled = tweetBox.state.text.length === 0 ||
      tweetBox.state.text.length > 140
  }
  showLeftTextCount() {
    $('#text-count').text(140 - this.state.text.length)
  }
  bindEvent() {
    $('.tweet-textarea').on('input', function () {
      //改变当前的state
      tweetBox.setState({'text', this.value
    })
      tweetBox.changeSubmit()
      tweetBox.showLeftTextCount()
    })
  }
}
```

用一个state保存当前的状态，通过获取当前state进行下一步的操作。

可以把它改得更加智能一点，即在上面setState的时候，自动去更新DOM，如下代码所示：

```
var tweetBox{
    init(){
        //初始化一个state
        this.state = {};
        tweetBox.bindEvent();
    }
    setState(key, value){
        this.state[key] = value;
        tweetBox.changeSubmit();
        tweetBox.showLeftTextCount();
    }
    changeSubmit(){
        //获取当前的state
        $(“#submit”)[0].disabled = tweetBox.state.text.length === 0 ||
                                        tweetBox.state.text.length >
140;
    }
    showLeftTextCount(){
        $("#text-count").text(140 - this.state.text.length);
    }
    bindEvent(){
        $(".tweet-textarea").on("input", function(){
              tweetBox.setState({"text", this.value});
        });
    }
}
```

做得更智能一点，在状态变的时候，自动渲染DOM

然后还可以再做得更智能，状态变的时候自动去比较当前状态所渲染的虚拟DOM和真实DOM的区别，自动去改变真实DOM，如下代码示：

```
var tweetBox {
  setState(key, value) {
    this.state[key] = value
    renderDom($('.tweet'))
  }
  renderDom($currentDom) {
    diffAndChange($currentDom,
      renderVirtualDom(tweetBox.state))
  }
}

'<input type="submit" disabled={{this.state.text.length === 0 ||
this.state.text.length > 140}}>'
```

这个其实就是React的原型，不同的状态有不同的表现行为，所以可以认为是一个状态模式，并且通过状态去驱动DOM的更改。

代理模式

如下图所示：

使用React不直接操作DOM，而是把数据给State，然后委托给State和虚拟DOM去操作真实DOM，所以它又是一个代理模式。

状态模式的另一个例子

React的那个例子并不是很典型，这里再举一个例子，如下代码所示，改变一个房源的状态：

```
if (newState === 'sold') {
  if (currentSate === 'building' ||
    currentState === 'sold') {
    return 'error'
  } else if (currentSate === 'ready') {
    currentSate = 'sold'
    return 'ok'
  }
} else if (newState === 'ready') {
  if (currentState === 'building') {
    currentState = 'toBeSold'
    return 'ok'
  }
}
```

改一个房源的状态之前先要判断一下当前的状态，如果当前状态不支持的话那么不允许修改，要是像上面那样写的话就得写好多个if-else，我们可以用状态模式重构一下：

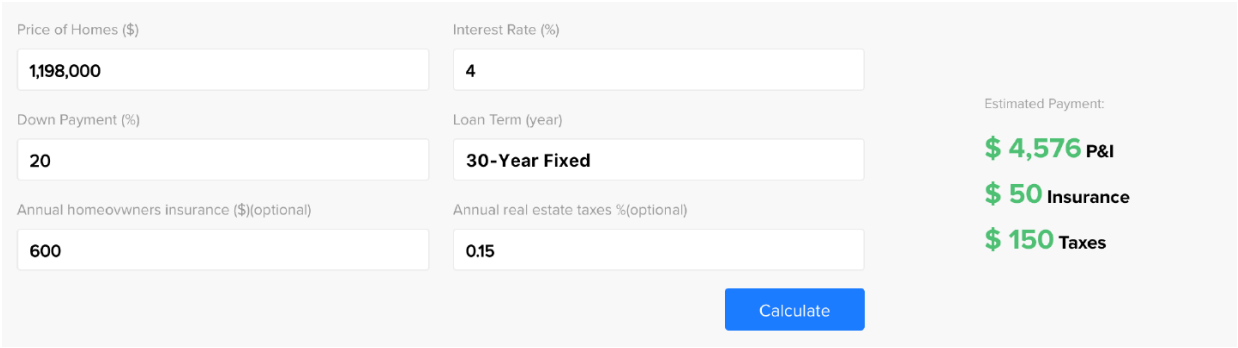
```
var stateChange = {
  'ready': {
    'buidling': 'error',
    'ready': 'error',
    'sold': 'ok',
  },
  'building': {
    'buidling': 'error',
    'ready': 'ok',
    'sold': 'error',
  },
}

if (stateChange[currentState][newState] !== 'error') {
  currentState = newState
```

```
}
return stateChange[currentState][newState]
```

你会发现状态模式和策略模式是孪生兄弟，它们的形式相同，只是目的不同，一个是封装成策略，一个是封装成状态。这样的代码就比写很多个if-else强多了，特别是当状态切换关系比较复杂的时候。

装饰者模式

要实现一个贷款的计算器，如下图所示：



点了计算的按钮之后，除了要计算结果，还要把结果发给后端做一个埋点，所以写了一个calculateResult的函数：

```
function calculateResult(form){
    var data = $(form).serializeForm();
    var l = data.rate / 1200;
    var o = Math.pow(1 + l, data.term * 12);
    var e = data.price * (1 - data.payment / 100);
    var result = (e * l * o / (o - 1);
    var formatResult = util.formatMoney((result).toFixed(0));

    var $calResult = $(".loan-cal .cal-result-con");
    $calResult.find(".pi-result").text(formatResult);

    return result;
}

//计算按钮click回调
var result = calculateResult(form);
//发一个埋点的请求
util.ajax("/cal-load", {result: result});
```

这个函数包含了两个功能，一个计算结果，一个改变DOM

因为要把结果返回出来，所以这个函数有两个功能，一个是计算结果，第二个是改变DOM，这样写在一起感觉不太好。那怎么办呢?

我们把这个函数拆了，首先有一个LoanCalculator的类专门负责计算小数的结果，如下代码所示：

```
//计算结果
class LoanCalculator {
```

```
  constructor (form) {
    this.form = form
  }

  calResult () {
    var result = …
    this.result = result
    return result
  }

  getResult () {
    if (!this.result) this.result = this.calResult()
    return this.result
  }
}
```

它还提供了一个getResult的函数，如果结果没算过那先算一下保存起来，如果已经算过了那就直接用算好的那个。

然后再写一个NumberFormater，它负责把小数结果格式化成带逗号的形式：

```
// 格式化结果
class NumberFormator {
  constructor (calculator) {
    this.calculator = calculator
  }

  calResult () {
    var result = this.calculator.calResult()
    this.result = result
    return util.formatMoney(result)
  }
}
```

在它的构造函数里面传一个calculator给它，这个calculator可以是上面的LoanCalculator，获取到它的计算结果然后格式化。

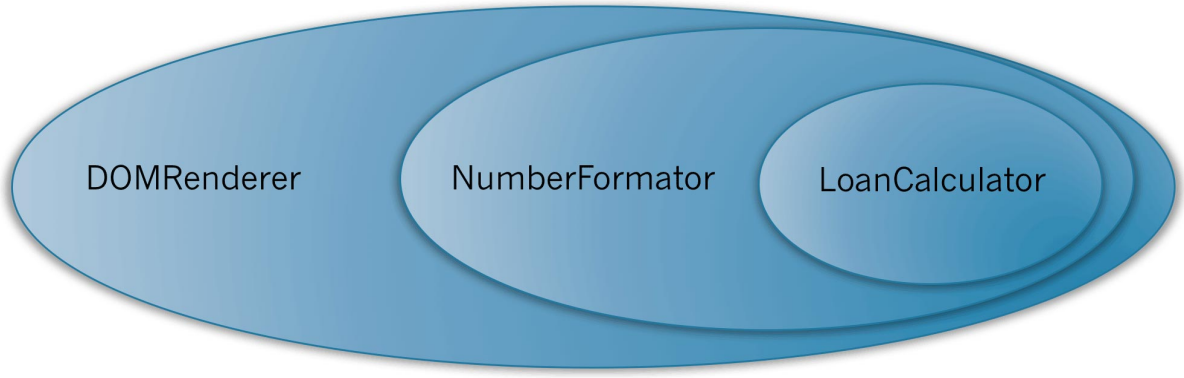
接着写一个DOMRenderer的类，它负责把结果显示出来：

```
// 显示结果
class DOMRenderer {
  constructor (calculator) {
    this.calculator = calculator
  }

  calResult () {
    var result = this.calculator.calResult()
    $('.pi-result').text(result)
  }
}
```

最后可以这么用：

```
var loadCalculator = new LoanCalculator(form)
var numberFormator = new NumberFormator(loadCalculator)
var domRenderer = new DOMRenderer(numberFormator)
domRenderer.calResult()

util.ajax('/cal-loan', {result: loadCalculator.getResult()})
```

可以看到它就是一个装饰的过程，一层一层地装饰，如下图所示：



下一个装饰者调上一个的calResult函数，对它的结果进一步地装饰。如果这些装饰者的返回结果类型比较平行的时候，可以一层层地装饰下去。

使用装饰者模式，逻辑是清晰了，代码看起来高大上了，但是系统复杂性增加了，有时候能用简单的还是先用简单的方式实现。

总结一下本文提到的面向对象的编程原则：

1. 把共性和特性或者会变和不变的分离出来
2. 少用继承，多用组合
3. 低耦高聚
4. 开闭原则
5. 单一职责原则