**传智播客前端和移动开发学院**

**http://web.itcast.cn**



设计模式

第0天课堂笔记（本课程共0天）

**目录**

[一、 面向对象和设计模式作业 2](#_Toc432500234)

[二、 从武学角度看设计模式和面向对象 3](#_Toc432500235)

[三、 什么是设计模式 4](#_Toc432500236)

[基本概念 4](#_Toc432500237)

[如何学习设计模式 4](#_Toc432500238)

[设计模式是面向未来编程 5](#_Toc432500239)

[面向对象原则 6](#_Toc432500240)

[面向对象编程的代码结构分析 6](#_Toc432500241)

[四、 设计模式的本质 6](#_Toc432500243)

[五、 设计模式应用场景分析 复杂大型项目 7](#_Toc432500245)

[六、 复杂问题编程第一步 UML辅助思考 –核心核心 7](#_Toc432500247)

[七、 面向对象原则 8](#_Toc432500249)

[八、 设计模式需要的JS技术复习 8](#_Toc432500251)

[九、 单例模式 重点 9](#_Toc432500253)

[十、 工厂模式 重点 10](#_Toc432500255)

[十一、 策略模式 重点 14](#_Toc432500257)

[十二、 观察者模式 重点 19](#_Toc432500258)

[十三、 装饰者模式 – 重点 21](#_Toc432500259)

[十四、 代理模式 – 了解 21](#_Toc432500260)

[十五、 AOP面向切面编程 了解 21](#_Toc432500261)

[十六、 中介者模式 了解 21](#_Toc432500262)

[十七、 组合模式 了解 21](#_Toc432500263)

[十八、 门面模式 了解 21](#_Toc432500264)

[十九、 模板方法模式 了解 21](#_Toc432500265)

[二十、 管道模式 了解 21](#_Toc432500266)

[二十一、 ORM映射模式 了解 21](#_Toc432500267)

# 作业

第一 将百度星座等案例改成jquery版本 -- 提交日期 15号晚上之前

第二 完善第一次实战的代码，调整下bug，然后提交 15号晚上之前--项目经理

第三 完善二次实战项目分配（14号） – 14号晚上之前最终确定下来 -- 项目经理

第四 二次实战+百度案例改造 14（完成百度作业） 15 （不休息）16号（休息半天）

至少完成80% 剩余的部分可以在毕业前的晚上完成

第五:阅读面试题目，不懂的请教同学或者百度，到时候找同学讲解答案，同学可以自己整理一些面试题目，以后毕业后不断的扩充，如果想面试外企的可以准备英文面试题目

**二次实战要求：**

参考你们已经修改的百度页面 尽量使用面向对象开发，但是不做强制要求

可以使用任何jquery插件 别人写好的任何框架 HTML5 CSS3等绚丽效果框架

可以尝试使用**观察者模式**，但不做强制要求

可以尝试使用**工厂模式**，但不做强制要求

可以尝试使用**策略模式**，但不做强制要求

可以尝试使用**装饰者模式**，但不做强制要求

# 从武学角度看设计模式和面向对象

|  |
| --- |
| 面试：  第一阶段：海投 – 熟悉各个公司的面试方式和面试试题 10家公司 失败  智联 52job 英才网 。。。  面试技巧 面试题 笔试题  Widow document nagator BOM ：  String里面的某个方法。。array ，date ，。。。。  不下1万了  更深刻。。。  紧张 。。。不在乎成败了。。反而放下来了。。。  一天至少投5000  包装简历：经验。。。  笔试题目（这些题目一般和网上差不多，算法题目（数组排序，将一个字符串颠倒。。，一般用于初级））  还有面试题目 – 面向对象 ---设计模式--- 策略模式—策略模式，你还懂哪几种，技术经理（压力面试，宅男，也是需要沟通技巧的，打击）  第二阶段：真正的我想去的公司 |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

# 什么是设计模式

|  |
| --- |
| 后台：23种基础 –后台 但是后来又出现了很多新的设计模式  前台：12种左右。  对同学的要求：  一般要求掌握2-4种。。  精通2种：单例模式 工厂模式  了解：观察者模式 适配器模式 装饰者模式 策略模式  其他：知道 AOP 控制反转 Struts spring hibernate 映射模式 MVC 观察模式  复习一下前面做的案例 |

|  |
| --- |
| 设计模式本质 – 解耦合 模块化 |

|  |
| --- |
| 设计模式如何解决复杂问题  设计模式的原则  设计模式需要用到的知识点复习  单例模式  不贪多，将一个练一个，  设计模式重点不是学习这些模式，  而是学习人家是如何模块化的。。。。 |

体会设计模式 其实也不是我们想的这么难

### 基本概念

设计模式和建筑学一样，是软件工程领域的设计前辈（四个人 Gang of Four 简称GOF）把建筑学的理论抄袭模仿创新发展成为软件开发领域的通用**解决方案**，我们称之为设计模式。

### 如何学习设计模式

* 第一：**知道**
* 第二：学习面向对象设计原则
* 第三**：生搬硬套**阶段，为了模式而模式
* 第四：走出为了模式而模式误区
* 第五：忘记模式
* 第六：行云流水，信手拈来

### 设计模式是面向未来编程

设计模式如果只从代码上看是将**代码复杂化**,比如策略模式：

我明明可以只用一个函数就可以搞定，为什么偏偏定义这么多对象，这不是SB吗？？

那么为什么全世界人都这么做？？那就是面向未来编程。

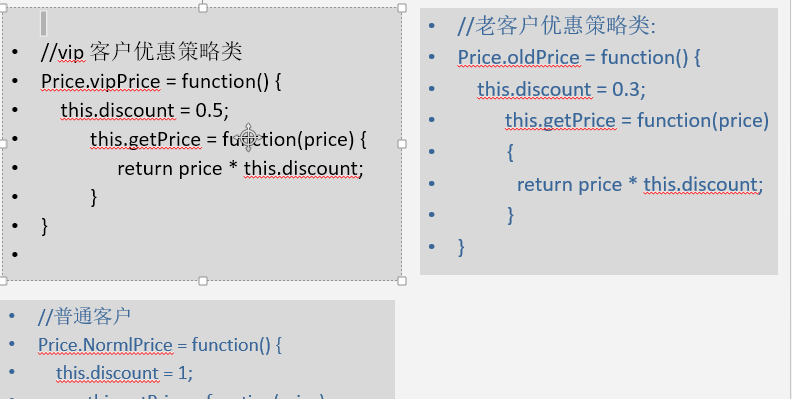
如果单从今天来看，确实是啰嗦，但是从未来需求变更，代码可维护性来看，从整个历史长河来看，他减轻的不是一天的工作量，有时候甚至是决定性的。。。。

**这就是面向未来编程**

**通过代码来体会设计模式是如何将代码复杂化的：**

如下是策略模式，体会设计模式其实是将**代码复杂化**，明明一个函数搞定，用了设计模式结果定义了多个对象，多个方法。。。。简直是啰嗦中的啰嗦。。和唐僧一样。。。脱裤子放屁。。

* function Price(personType, price) {
* //大客户 5 折
* if (personType == 'vip') {
* return price \* 0.5;
* }
* else if (personType == 'old'){ //老客户 3 折
* return price \* 0.3;
* } else {
* return price; //其他都全价
* }
* }
* 只扩充不修改原则



### 面向对象原则

设计模式只不过是面向对象编程的延迟

以上设计模式其实是面**向对象编程原则**的具体运用

也就是设计模式的代码为什么这么写：是有个依据的，就是：**面向对象编程原则**

只要按照这些原则去实现代码，就能保证代码**健壮性，可维护性，可扩展性**

### 面向对象编程的代码结构分析

学习设计模式其实是为了掌握**面向对象编程方法**，而不是为了学习这些模式，通过模式来理解面向对象的编程方法，这才是重要的。

所有的**面向对象编程**的代码结构都是按照如下两点拼凑出来的（包括设计模式）：

**面向对象原则**

**对象之间的关系**



# 设计模式的本质

|  |
| --- |
| 模块化 解耦合  搭积木开发模式---  先去开发一些积木。。。  产品详细页面  产品基本信息模块 产品详细信息模块 评论模块 –  Var page =function(){  guanlianguanxi  This.module1.  This.module2  }  关联关系：1:1 1：n  购物车 var shoppingCar ={  1:m=n  List【product】  }  **耦合的概念** |

* 耦合定义来源于电路学。
* 当一个模块直接修改或者损坏，会导致另一个模块也损坏或者不起作用，则称二者存在耦合。

解耦合---高耦合 低耦合 降低耦合 、、

|  |
| --- |
| 解耦合 |

|  |
| --- |
|  |



# 设计模式应用场景分析 复杂大型项目

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |



# 复杂问题编程第一步 UML辅助思考 –核心核心

|  |
| --- |
| 主要是了解对象之间的关系，昨天我们已经讲解了对象之间的各种关系：  没有关系  关联关系 – 产品详细信息模块化  继承关系 – 计算周长案例  聚合关系 – 产品详细信息模块化  依赖关系：  对象作为另一个函数的参数  对象的方法中使用了另一个对象 |
|  |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |



# 面向对象原则

|  |
| --- |
| 面向对象 设计模式他们都是依据 这些原则写出来的：  同样，也是面向对象三大特征的运用：封装，多态，继承  单一职责原则 -- 面向未来编程 。。。。 |

|  |
| --- |
| **面向未来编程**：现在编程 现在的功能  面向未来：健壮性 可维护性 可扩展性 高可用（硬件）  关系网 --- InfoSys |

|  |
| --- |
| 只增加 不修改（少修改）原则 -- 开放封闭法则  七八个。。。 |



# 设计模式需要的JS技术复习

|  |
| --- |
| 高阶函数 --- 函数的参数是函数 – 参考ajax，框架，设计模式基础篇等  Btn.on(‘id’,function(){  })    回调函数 |

|  |
| --- |
| This – 参考设计模式基础篇 |

|  |
| --- |
| Apply call -- 参考框架，面向对象中高级  链式访问 -- 参考框架，设计模式基础篇  命名空间 – 参考框架，面向对象中高级  var baidu={};  baidu.basice={  isArray();  ajax();  isString();  each  }  Baidu.dom={  Shijian  Xuanze  removeClass  }  baidu.controls={  alert();  calender;  tab;  Layer();  };  Baidu.controls.alert(); |



# 单例模式 重点

|  |
| --- |
| 单例模式定义  就是保证只有一个实例。。就类似一个全局变量或者字面量  A团队：  Function gouwuce(){  This.price=0;  This.products=[];  }  Var g1= new gouwuce();  Var g2 = new gouwuce();  产品详细信息页面，团购Tuan，秒购Miao，。。。。。  Var g1 = new gouwuche();  G1.price+=1000;  Fuwulei  全局变量  Window.price=111;  Var g2 = new gouwuce(); |

|  |
| --- |
| 单例模式应用场景分析  其他语言： |

|  |
| --- |
| 单例模式代码结构 |



单例模式实战

总结：模式重要的真不是代码，而是理解以及应用场景。

全局变量，或者一个字面量对象。。。。。

在js中，完全可以用字面量实现单例模式。。。。。。

单例模式其实我们完全可以用字面量或者一个全局变量来实现。

但是为什么还用闭包的形式包裹起来呢。。。。

解决污染。。。

实现公有私有。。。。

如何访问一个对象里面的属性和方法：

Widows.$

Return{}

Call apply 神奇的东西。。。。。

和json字面量的区别

使用单例模式其实是一个完整的对象，可以定义公有和私有方法。。。

# 工厂模式 重点

|  |
| --- |
| 定义 – 使用工厂来专门实例化的工作。。。  使用场景  代码结构  //团队1  //算法家族 --放在同一的命名空间中  var Price={};  //vip 客户优惠策略类  Price.vipPrice = function() {  this.**discount** = 0.5;  this.getPrice = function(price) {  return price \* this.**discount**;  }  }  //老客户优惠策略类:  Price.oldPrice = function() {  this.**discount** = 0.3;  this.getPrice = function(price)  {  return price \* this.**discount**;  }  }  //普通客户  Price.NormlPrice = function() {  this.**discount** = 1;  this.getPrice = function(price)  {  return price ;  }  }  //核心点：面向未来编程： 以后在维护的时候需求编程：比如新增钻石客户  Price.DiamondPrice = function() {  this.**discount** = 0.98;  this.getPrice = function(price)  {  return price \*discount;  //送我京东豆 // 专属的电话频道。。。  }  }   //工厂 -- 专门负责生成不同的实例 --就比如 吃货不用关心东西是如何制造的一样。。。  Price.factory = function (typeType) {  return new Price[typeType];  }  // json. //json对象的调用方法的第二种； json['属性名称']  代码结构分析 工厂模式分析  //团队2：使用 根本不用关心制造过程 ，只需要学会使用， // 其实就是面向对象的封装性，多态性 var normalPrice = Price.factory("NormlPrice").getPrice(1000); var oldPrice = Price.factory("oldPrice").getPrice(1000); alert(normalPrice); alert(oldPrice);  //未来新增一个新的优惠方式，如何改动最少，---只添加不修改（或者改动少）， // 这就是传说中的开放封闭法则（只新增不修改或者少修改法则） var normalPrice = Price.factory("DiamondPrice").getPrice(1000);  实战 |

|  |
| --- |
|  |

|  |
| --- |
|  |



# 策略模式 重点

策略模式定义

算法家族。。。

打折优惠：vip 钻石 普通。。

代码

场景

<script>  
  
//算法家族 --放在同一的命名空间中   
var Price={};  
  
//vip 客户优惠策略类  
Price.vipPrice = function() {  
 this.**discount** = 0.5;  
 this.getPrice = function(price) {  
　 　return price \* this.**discount**;  
 }  
}  
   
//老客户优惠策略类:  
Price.oldPrice = function() {  
 this.**discount** = 0.3;  
 this.getPrice = function(price)  
 {  
 return price \* this.**discount**;  
 }  
}  
  
//普通客户  
Price.NormlPrice = function() {  
 this.**discount** = 1;  
 this.getPrice = function(price)  
 {  
 return price ;  
 }  
}  
  
  
//后台的数据库  
// 关系型数据库：oracel MS SQL SERVER MYSQL DB2 1:1 1：n  
// 1：： 班级表 讲师 讲师 和学生  
  
  
  
  
  
//2. 上下文- 类似工厂模式中工厂  
//这里上下文的目的只是方便第三方使用，不用关心算法是怎么实现的，只需要会使用即可  
//算法对象的引入是方便我们思考问题，应对客户每日三疯 -- 需求变化频繁  
//上下文是方便用户使用方便，封装了算法的细节  
function Context() {  
 this.**name** = '';  
// 关联关系  
 this.**strategy** = null;  
 this.**price** = 0;  
}  
   
Context.**prototype**.set = function(name, strategy, price) {  
 this.**name** = name;  
 this.**strategy** = strategy;  
 this.**price** = price;  
}  
   
Context.**prototype**.getResult = function() {  
 //alert('123');  
 //alert(this.name);  
 //alert(this.price);  
 //alert(this.strategy.getPrice(this.price));  
 **document**.getElementById('type').**innerHTML**=this.**name**;;  
 **document**.getElementById('mydiv').**innerHTML**=this.**strategy**.getPrice(this.**price**);;  
 alert(this.**strategy**.getPrice(this.**price**));  
}  
  
  
  
//3. 客户端使用;  
// 上下文  
var context = new Context();  
   
//vip客户  
var vip = new Price.vipPrice();  
context.set ('vip客户', vip, 200);  
context.getResult();  
   
//老客户  
var old = new Price.oldPrice();  
context.set ('老客户', old, 200);  
context.getResult();  
   
//普通客户  
//...  
  
  
  
//如果未来针对老顾客的优惠方式变了，只需要修改单个对象，而且修改这个对象不会影响其他对象  
//如果未来针对vip的优惠方式变了，只需要修改单个对象  
//如果以后客户又疯了，需要新增一种优惠方式，只需要新增一个对象，原来的代码丝毫不影响，也不需要修改  
  
</script>

# 观察者模式 重点

什么是观察者模式 – 生活场景分析 水壶烧水 购买楼房 监听洪水的水位。

开发场景分析 -- 新浪体育板块

观察者模式的三要素： 观察者（链家地产 售楼中心） 监听者（就是想要购买房子的人，数组） 通知（链家地产就会根据监听列表以此通知每个已经注册的人）

监听者有可能：买房，借钱，抢劫银行，打劫。。。

代码讲解

观察者 售楼中心

Var ShouLouCenger = function（）{

//监听者

Var Oberservers=[];

AddOberserver:function(JianTingZhe){

Oberservers.push(JianTingZhe)

}

Notify:function(){

For(var obj in Oberservers)

{

Obj.upate();

}

}

}

定义监听者对象

Function xiaowang=function（）{

This.do=function(){

抢劫银行

}

}

Function xiaoLi=function（）{

This.do=function(){

直接买房

}

}

新闻板块

Function News=function（）{

This.update=function(){

更新最新新闻

}

}

球队的最新排名

Function News=function（）{

This. update =function(){

火箭 –勇士 --- 骑士 – 热火

}

}

精彩回放

Function News=function（）{

This. update =function(){

火箭 –勇士 --- 骑士 – 热火

}

}

# 装饰者模式 – 重点

## 装饰模式使用场景分析 -- 用于递进式场景

什么是递进式 什么是并列式？？？

很简单—

打折优惠有两种模式：

什么是并列式，什么是递进式

Var price = 1000；

并列式 一般用于会员： 铜牌 9 金牌8 钻石7

Price = 900X90%；

Price = 800X90%；

Price = 700X90%；

Price = 1000；

Price = 900；

Price = 900X90%；

900X90%X90% ;

**装饰着模式一般用于递进式包装**

**而策略模式用于并列式算法家族**

## 代码分析

//////////////////基础的产品对象//////////////////  
// 打折优惠的两种形式 -- 并列式 -- 递进式  
// 什么是并列式，什么是递进式  
//产品基础对象 -- 基类  
function Product() {  
 this.**description**="基础对象 记录产品基本价格";  
 this.cost=function()  
 {  
 return 1000;  
 }  
}  
////////////各种优惠方式////////////////  
//打折优惠 -- 传递了一个对象，这个对象和基类都有cost方法，这也是多态的一种体现  
function Discount(obj) {  
 this.**description**="打折优惠--比如5折";  
 this.cost=function()  
 {  
 return obj.cost() \* 0.5;   
 }  
}  
//钻石会员 -- 抵消10元  
function VIP(obj) {  
 this.**description**="vip用户抵消10元";  
 this.cost=function()  
 {  
 return obj.cost()-200;//500-200=300  
 }  
}  
//京东10周年庆典日 -- 优惠10元  
function HappyDay(obj) {  
 this.**description**="庆典日 -- 优惠10元";  
 this.cost=function()  
 {  
 return obj.cost() - 300;  
 }  
}  
//////////////////使用///////////////////  
 //基础产品 - 原价  
var product=new Product();  
**console**.log(product.cost())

//打折促销之后产品的价格  
var p\_Discount=new Discount(product);  
**console**.log(p\_Discount.cost())

//打折促销，并且这个人还是会员，进一步优惠  
var p\_Discount\_VIP = new VIP(p\_Discount);  
**console**.log(p\_Discount\_VIP.cost())

//到了庆典，在原来的基础上再优惠  
var P\_Discount\_VIP\_HappyDay = new HappyDay(p\_Discount\_VIP);  
**console**.log(P\_Discount\_VIP\_HappyDay.cost())  
alert("产品经过打折优惠，VIP会员，恰逢京东十周年庆典之后的价格："+P\_Discount\_VIP\_HappyDay.cost()+"元");  
</script>

# AOP面向切面编程 了解

## 通过计算函数执行时间来熟悉AOP

目前统治世界的一种编程思维

Spring JAVA

C# SPRING.NET

注入器

控制反转。。。。。。

面向切片编程

举个例子：

比如如何去计算函数的执行时间。。。。。

Function（）{

Var time = new date。Time（）；当前时间

///

///

//

Var timeEnd = new date。Time（）； 当前时间

TimeEnd – time；

}

下面是详细的代码：

//统计函数运行时间  
function Func6() {  
 //计算时间的代码  
 var now = +new Date();  
 //代码  
 sleep(1000);  
 //计算时间的代码  
 var time = +new Date() - now;  
 alert(time);  
}

上述代码的缺点：

业务逻辑和测试代码都放在了一起，破化了单一职责原则。

那么我们可以用AOP思想动态的注入代码。。。。

那么如何做呢，我们使用外国开发的一个AOP框架 aop.js

//将时间记录函数封装一下  
function logTime (func) {  
 //相当于重新包装func，在前面植入一些代码，在后面又植入一些代码  
 //这里： 1 在前面植入：当前时间  
 //2 函数后面植入统计函数执行事件代码  
 return func = (function() {  
 var d;  
 return func.before(function() {  
 d = +new Date();  
 }).after(function() {  
 alert(new Date() - d);  
 });  
 })()  
}

## AOP概念来源

* 在利用高级语言，比如java,C#开发后台的时候，发现很多函数，方法，模块中都需要以下代码：
* 基础代码
  + 权限控制
  + 日志记录
  + 数据校验
  + 异常处理
  + 主动通知
  + 也就是正规意义上编程，有时候除了**核心代码**，可能还包含其他一些所有函数都通用的代码

## AOP概念

* AOP是Aspect-Oriented Programming（面向方面编程或面向切面）的简称
* AOP实际是GoF设计模式的延续，设计模式孜孜不倦追求的是调用者和被调用者之间的解耦，AOP可以说也是这种目标的一种实现。
* 可以在不修改原有代码的情况下增加新功能

## AOP植入分类

* AOP通过 “通知”来动态植入功能。具体分为
  + Before切面（前置通知）
    - 可以在函数执行前自动调用某个函数
  + After界面（后置通知）
    - 可以在函数执行后自动调用某个函数
  + Around切面（环绕通知）
    - Around是可以同时在所拦截方法的前后执行一段逻辑。
* 通俗讲：就是让一个函数在另一个函数之前或者之后执行
* 用过spring的同学肯定对它非常熟悉

## AOP的实现机制—了解

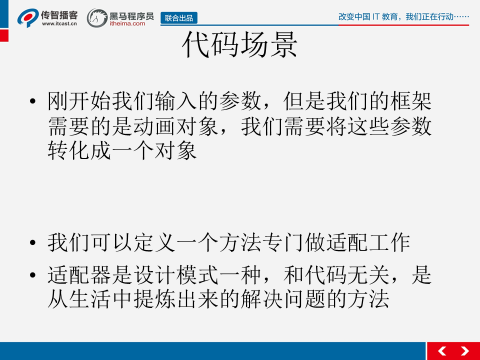
//给函数扩展方法  
**Function**.**prototype**.before = function( func ) {  
 var \_\_self = this;  
 return function() {  
 if ( func.apply( this, arguments ) === false ) {  
 return false;  
 }  
 return \_\_self.apply( this, arguments );  
 }  
}  
  
**Function**.**prototype**.after = function( func ) {  
 var \_\_self = this;  
 return function() {  
 var ret = \_\_self.apply( this, arguments );  
 if( ret === false) {  
 return false;  
 }  
 func.apply( this, arguments );  
 return ret;  
 }  
}

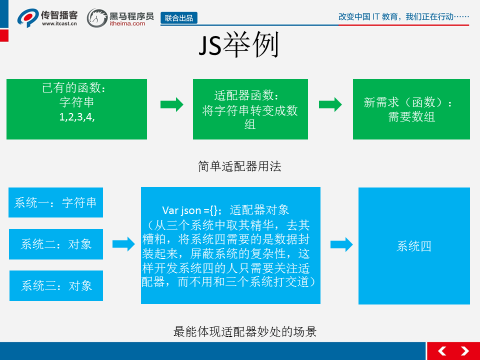
# 模板方法模式 了解

1. //模拟生物的成长 -- 做成一个模板  
   //有些是所有生物公有的  
   //有些是部分生物不一样的。。。  
   //一般越往底层越一样  
   // 比如 人 狗 --- 从上层看 一个是人 一个是狗  
   //但是从底层构造来看则制造方式一样：头部，心脏，胸腔，大肠，小肠，生殖器官等  
   //再往底层 --细胞 血液 -- 可能存在差异  
   //再往底层 --原子，分子，质子，中子 --- 一样  
   //同样这里 DNA复制 比较靠近底层，基本所有生物都一样。。。。一般在父类实现，并且设置不允许修改  
   //其他比如出生则属于高层，每个生物是不一样的， 允许子类修改  
   function Life(){  
   }  
   Life.**prototype**.init = function(){  
    this.DNA复制();  
    this.出生();  
    this.成长();  
    this.衰老();  
    this.死亡();  
   }  
     
   //其中DNA复制是预先定义的算法中不变部分. 所有子类都不能改写它. 如果需要我们可以写成protected的类型.  
   Life.**prototype**.DNA复制 = function(){  
    // &\*$%&^%^&(&(&(&&(^^(\*)   
   //看不懂的代码  
    alert('DNA');  
   }  
     
   //而其他的函数在父类中会被先定义成一个空函数(钩子). 然后被子类重写，这就是模版方法中所谓的可变的步骤。  
   Life.**prototype**.出生 = function(){  
    alert('出生');  
   }  
   Life.**prototype**.成长 = function(){  
    alert('成长');  
   }  
   Life.**prototype**.衰老 = function(){  
    alert('衰老');  
   }  
   Life.**prototype**.死亡 = function(){  
    alert('死亡');  
   }  
     
     
     
   //var life = new Life();  
   //life.init();  
     
     
     
     
     
   //子类 --哺乳动物  
   function Mammal(){  
   }  
     
   Mammal.**prototype** = new Life();   
   //继承Life  
     
     
   Mammal.**prototype**.出生 = function(){  
    alert('重写哺乳动物的出生');  
   }  
     
   Mammal.**prototype**.成长 = function(){  
   //再留给子类去实现  
    alert('成长');  
   }  
     
   Mammal.**prototype**.衰老 = function(){  
    alert('重写哺乳动物的衰老');  
   }  
   Mammal.**prototype**.死亡 = function(){  
    alert('死亡');  
   //再留给子类去实现  
   }  
     
     
   //var mammal = new Mammal();  
   //mammal.init();  
     
     
     
     
   //再实现一个Dog类  
   var Dog = function(){  
   }  
   //Dog继承自哺乳动物.  
   Dog.**prototype** = Mammal.**prototype**;  
     
   var dog = new Dog();  
   dog.init();

# 适配器模式

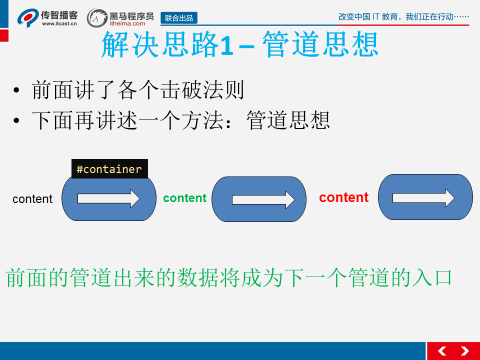


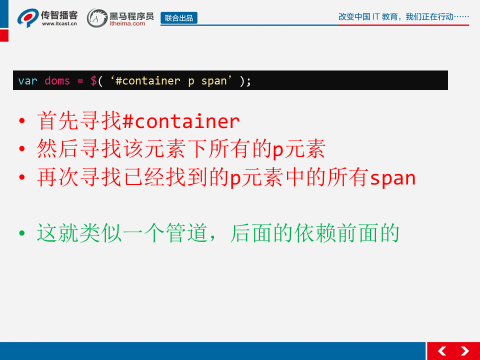






# 管道模式 了解





# 代理模式和中介模式 – 了解

从生活中先去理解代理模式

能力

# ORM映射模式 知道即可

本质就是适配器模式

传统方式：Sql : select\*from students where id=1

存储过程 。。。。。。。触发器 。。。。

优点：快

缺点：不是面向编程编程。。

ORM框架

学生表

Var students = [{},{},{},{}]

New students[0].name

Delete

增删查改

Hibernate C# ado.net enity framework Nhibernate

有点慢。。。

Sql

Hibernate

Java .NET

C++ -- 游戏。。

# 组合模式 知道即可

# 门面模式 知道即可