# 设计模式，MVC，MVVM

第1天课堂笔记（本课程共4天）

[设计模式，MVC，MVVM 1](#_Toc5044)

[一，设计模式 3](#_Toc18457)

[1 四人帮 3](#_Toc15325)

[2 23种设计模式 3](#_Toc7402)

[3 模式类型 3](#_Toc14117)

[4 设计模式，框架，架构与库 3](#_Toc6398)

[二，工厂模式 4](#_Toc26569)

[1 简单工厂模式。 4](#_Toc19604)

[2 寄生增强工厂 5](#_Toc24600)

[3 安全工厂模式 6](#_Toc32657)

[4 工厂方法 6](#_Toc29706)

[三 继承 7](#_Toc15135)

[四 原型模式 7](#_Toc24787)

[五 单例模式 7](#_Toc1423)

[1 静态变量 8](#_Toc445)

[2 惰性单例 8](#_Toc2623)

[六：适配器模式 8](#_Toc1286)

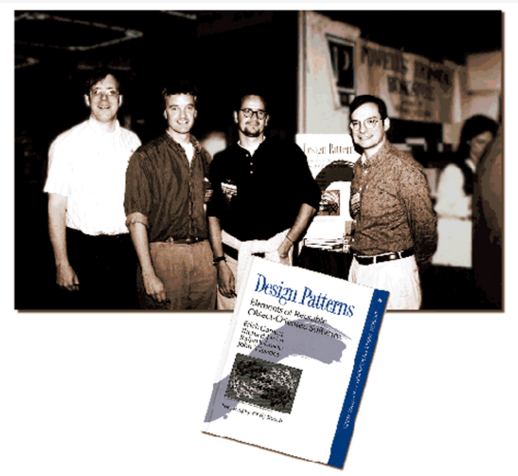
[1 适配代码库 8](#_Toc30077)

# 一，设计模式

是一套可复用的，被众人知晓，经过编目分明的，经验的总结。

作用：使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性

## 1 四人帮



1995年 Erich Gamma， Richard Helm, Ralph Johnson ,John Vlissides合作出版了Design Patterns - Elements of Reusable Object-Oriented Software 一书，在此书中共收录了23个设计模式。这四位作者在软件开发领域里也以他们的匿名著称Gang of Four(四人帮，简称GoF),并且是他们在此书中的协作导致了软件设计模式的突破。有时这个匿名GoF也会用于指代前面提到的那本书，第一次将设计模式提升到理论高度，并将之规范化，本书提出了23种基本设计模式，自此，在可复用面向对象软件的发展过程中，新的大量的设计模式不断出现。

## 2 23种设计模式

创建型模式：单例模式、抽象工厂模式、建造者模式、工厂模式、原型模式。（5）

结构型模式：适配器模式、桥接模式、装饰模式、组合模式、外观模式、享元模式、代理模式。（7）

行为型模式：模版方法模式、命令模式、迭代器模式、观察者模式、中介者模式、备忘录模式、解释器模式（Interpreter模式）、状态模式、策略模式、职责链模式(责任链模式)、访问者模式。（11）

## 3 模式类型

创建型设计模式：解决对象在创建时产生的一系列问题。

结构型设计模式：解决类或对象之间组合时产生的一系列问题

行为型设计模式：解决类或对象之间的交互以及职责分配的一些列问题

## 4 设计模式，框架，架构与库

**设计模式**：是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结，它强调的是一个设计问题的解决方法

**框架**：软件框架是项目软件开发过程中提取特定领域软件的共性部分形成的体系结构，不同领域的软件项目有着不同的框架类型。框架不是现成可用的应用系统。而是一个半成品，提供了诸多服务，开发人员进行二次开发，实现具体功能的应用系统。

设计模式研究的是针对单一问题的设计思路和解决方法，一个模式可应用于不同的框架和被不同的语言所实现；而框架则是一个应用的体系结构，是一种或多种设计模式和代码的混合体虽然它们有所不同，但却共同致力于使人们的设计可以被重用，在思想上存在着统一性的特点，因而设计模式的思想可以在框架设计中进行应用。

**架构**：简单的说架构就是一个蓝图，是一种设计方案，将客户的不同需求抽象成为抽象组件，并能够描述这些抽象组件之间的通信和调用，偏重于设计。框架比架构更具体，更偏重于技术

**工具库**：是类（方法）的集合，这些类之间可能是相互独立的。可以直接调用它，而不必再写一个。框架也往往是类（方法）的集合；但框架中的各个类并不是孤立的，而框架中的业务逻辑代码是将不同的类“连”在一起，在它们之间建立协作关系

# 二，工厂模式

## 1 简单工厂模式。

定义：用来创建一种类型的产品实例，所以他创建的对象单一，

特点：

通过将创建过程寄生在工厂内，可以解决全局变量污染问题。

他们创建的产品对象单一。

我们对工厂进行改造可以适应不同的需求。

|  |
| --- |
| 1. function BallFectory () { 2. var x = Math.random() \* w; 3. var y = Math.random() \* h; 4. var r = Math.random() \* 10 + 10; 5. var ball = new Ball(x, y, r); 6. return ball; 7. } 8. function BallFectory (x, y, r) { 9. var x = x || Math.random() \* w; 10. var y = y || Math.random() \* h; 11. var r = r || Math.random() \* 10 + 10; 12. var ball = new Ball(x, y, r); 13. return ball; 14. } |

批量生产

第一种：

在外部创建一个批量生产的工厂，然后再内部调用工厂，他的问题是需要另外创建一个工厂

第二种

通过在工厂方法内部进行分支判断，决定创建单个产品或者品量单品。

这种方式在工厂内部调用该厂，依赖了工厂名称。

第三种

通过在工厂方法内部进行分支判断，决定创建单个产品或者品量单品。

这种方式在工厂内部调用了，arguments.callee解决了对工厂名称的依赖。

|  |
| --- |
| 1. function BallFectory (num) { 2. if (num) { 3. var result = []; 4. for (var i = 0; i < num; i++) { 5. // result.push(BallFectory()) // 第二种 6. result.push(arguments.callee()) // 第三种 7. } 8. return result 9. } else { 10. var x = Math.random() \* w; 11. var y = Math.random() \* h; 12. var r = Math.random() \* 10 + 10; 13. var ball = new Ball(x, y, r); 14. return ball; 15. } 16. } |

## 2 寄生增强工厂

特点

通过对寄生在工厂内部的对象增添方法属性，使在不改变原类的基础上，完成对类的拓展

1 在工厂内部实例化类 这一步叫做寄生

2 对实例化的类拓展方法和属性 这一步叫做增强

3 将这个对象返回。 这一步是工厂。

|  |
| --- |
| 1. function BallFectory () { 2. // 这一步对类的实例化--寄生 3. var x = Math.random() \* w; 4. var y = Math.random() \* h; 5. var r = Math.random() \* 10 + 10; 6. var ball = new Ball(x, y, r); 7. // 这一步对实例化对象的拓展--增强 8. ball.vx = Math.random() \* 5; 9. ball.vy = Math.random() \* 5; 10. ball.changeColor = function () { 11. ctx.fillStyle = Math.random() > .5 ? 'red' : 'blue' 12. } 13. ball.move = function () { 14. var that = this; 15. setInterval(function () { 16. // 水平方向边界处理 17. if (that.x - that.r < 0 || that.x + that.r > w) { 18. that.vx \*= -1; 19. } 20. // 垂直方向边界处理 21. if (that.y - that.r < 0 || that.y + that.r > h) { 22. that.vy \*= -1; 23. } 24. // 让球运动 25. that.x += that.vx; 26. that.y += that.vy; 27. // 清除上一帧，并重新绘制小球 28. ctx.clearRect(0, 0, w, h); 29. that.changeColor(); // 改变小球的颜色 30. that.draw() 31. }, 33); 32. } 33. // 将实例化对象返回--工厂 34. return ball; 35. } |

## 3 安全工厂模式

又叫安全类

特点，不论在调用的时候有没有new关键字，得到的结果都是一样的。

1 判断this是否是只想当前对象的。通过instanceof

2 如果是通过new关键字创建的，就直接对this赋值

3 如果不是，主动创建，并返回该实例。

|  |
| --- |
| 1. var Book = function (title) { 2. // 判断是否调用new关键字的时候， 3. if (this instanceof Book) { 4. this.title = title; 5. } else { 6. return new Book(title); 7. } 8. } |

## 4 工厂方法

定义：通过对产品类的抽象使其创建业务主要负责用于创建多类产品的实例

特点：

是创建多类对象

也是对类的再封装

步骤：

1 声明参数

2 循环创建类

3 对类的添加方法

4 将实例化对象返回

# 三 继承

我们学习三类继承

第一类：构造函数式的继承

实现，通过在子类中的构造函数类调用call方法执行父类的构造函数，来实现对父类的构造函数的继承，

第二类：原型式继承

实现。通过将父类的实例化对象赋值给子类的原型，让子类继承父类的原型

第三类：组合式继承

实现：通过在子类中的构造函数类调用call方法执行父类的构造函数，来实现对父类的构造函数的继承，并且通过将父类的实例化对象赋值给子类的原型，让子类继承父类的原型

总结

构造函数式继承，它继承了父类的构造函数，但不会继承父类的原型

原型式继承，继承了父类的原型，不会继承父类的构造函数

组合式继承：既继承了父类的构造函数，又继承了父类的原型

注意

1 对于构造函数式的继承，我们在构造函数中添加属性的时候，一定要在call（父类的构函数）执行的后面添加

2 对于原型式继承。为原型添加属性和方法的时候，一定在继承的后面，并且不能用{}语法

# 四 原型模式

定义：通过将对象的原型指向类的原型实现对该对象原型的属性与方法的共享。

是基于继承的一种模式，它也是创建型模式

|  |
| --- |
| 1. var Rect = function (x, y) { 2. Base.call(this, x, y); 3. this.dom.className = 'rect'; 4. } 5. Rect.prototype = new Base(); |

总结：

原型模式一种创建型设计模式

它基于javascript原型链原理实现的

是一种组合式继承

对于处理复杂的类，通过提取公共部分实现对类优化

# 五 单例模式

定义：只能被实例化一次的对象。

## 1 静态变量

定义：定以后不能被修改的一类变量。

实现：

Javascript中的变量是可以任意更改的，前提是这个变量必须可以获取到，如果我们可以将这类变量存储在封闭作用域里，不被外界访问到，此时这个变量就获取不到了。这个封闭的作用域我们可以通过闭包来实现。闭包的返回对象或者返回函数执行时是在闭包的作用域内，所以这类函数是可以访问到这类变量的。我们将这个函数封装成取值器，就是先对静态变量的获取。

|  |
| --- |
| 1. var Conf = (function () { 2. // 定义在闭包内部的变量，在外界是访问不到的。 3. var \_conf = { 4. MAX\_NUM: 100, 5. COUNT: 1000 6. } 7. // 通过将返回函数定义成取值器，这样我们在外界就能通过该函数访问到该变量了 8. return function (name) { 9. return \_conf[name]; 10. } 11. })() |

特别注意

静态变量存储的数据一定值类型的数据

## 2 惰性单例

作用

推迟复杂的类实例化时期、（晚点实例化）

实现： 通过闭包将我们的single封装起来，避免外部访问而实例化，这样可保证实例化一次，闭包的返回函数的作用域在闭包里面，所以可以访问到single类对其实例化，这样在执行LazySingle才会对single类实例化

特点

1 只能被实例化一次

2 推迟了实例化时间。

# 六：适配器模式

定义：将一个类或对象的接口方法适配到另外一个类或对象的接口方法，来完成需求。

## 1 适配代码库