目录

[设计模式 1](#_Toc14)

[一、 设计模式 3](#_Toc12630)

[1.1 设计模式的分类 3](#_Toc6864)

[1.2 设计模式的作用 3](#_Toc32606)

[1.3 历史 3](#_Toc25587)

[二、 工厂模式 3](#_Toc24555)

[2.1 简单工厂 3](#_Toc3467)

[2.2 寄生增强工厂 4](#_Toc1025)

[2.3 工厂方法 5](#_Toc14026)

[三、 原型与继承 5](#_Toc24418)

[3.1 原型 5](#_Toc22643)

[四、安全工厂 7](#_Toc13215)

[下午复习： 8](#_Toc12847)

[五、闭包类 9](#_Toc7352)

[5.1 单例模式 9](#_Toc10022)

[5.2 普通单例 10](#_Toc29614)

[5.3 惰性单例 11](#_Toc11470)

[六、静态变量 11](#_Toc16054)

# 设计模式

设计模式： 是一套编目分明、广为人知，是可复用的代码经验的总结

工具库：一些常用方法的集合体，例如jquery, underscore，这些方法之间通常是没有联系的

框架：一套半成品代码，它们里面也支持一些方法，但是通常是有联系的

架构：一套大型项目的设计思路

## 设计模式的分类

所有的设计模式可以分为三类：

1 创建型设计模式

2 结构型设计模式

3 行为型设计模式

## 设计模式的作用

创建型设计模式：解决了创建对象时候的问题

结构型设计模式： 解决了对象和类组合在一起时候的问题

行为型设计模式： 解决了对象和类耦合、职责之间的关系

## 历史

设计模式最初是由国外的GOF(Gang of four)合著， 设计模式一共有23种， 当设计模式发展至今，已经远远超过23种

# 工厂模式

## 2.1 简单工厂

|  |
| --- |
| 1. function createObject(name, sex, length, job) { 2. // 定义对象 3. var obj = { 4. name: name, 5. sex: sex, 6. length: length, 7. job: job 8. } 9. // 返回obj 10. return obj; 11. } 12. // 再次调用该方法 13. var laowang = createObject("老王", "男", 180, "IT"); 14. var laoli = createObject("老李", "男", 181, "厨师"); |

结果：

|  |
| --- |
|  |

## 2.2 寄生增强工厂

|  |
| --- |
| 1. // 定义工厂 2. function People(name, age, sex) { 3. this.name = name; 4. this.age = age; 5. this.sex = sex; 6. } |

另外一个工厂：

|  |
| --- |
| 1. // 定义新的工厂方法 2. function StrongPeople(name, age, sex) { 3. // 在这里实例化另外一个工厂 叫做寄生 4. var p = new People(name, age, sex); 5. // 添加属性 6. p.title = "nihao"; 7. // 添加方法 8. p.say = function() { 9. console.log("你好"); 10. } 11. // 返回p 12. return p; 13. } |

执行代码：

|  |
| --- |
| 1. // 调用StrongPeople 2. var p = new StrongPeople("小白", 12, "男"); |

结果：

|  |
| --- |
|  |

## 2.3 工厂方法

|  |
| --- |
| 1. // 工厂方法： 可以管理多个实例化对象的创建 2. // 定义类 3. function Dog() { 4. this.type = "dog"; 5. } 6. // 定义类 7. function Pig() { 8. this.type = "pig"; 9. } 10. // 定义类 11. function Cat() { 12. this.type = "cat"; 13. } 14. // 定义工厂方法 15. function FactoryMethod(type) { 16. // 判断传递的type 17. if (type == "dog") { 18. return new Dog(); 19. } else if (type == "pig") { 20. return new Pig(); 21. } else { 22. return new Cat(); 23. } 24. } 25. var dog = FactoryMethod("dog"); 26. var pig = FactoryMethod("pig"); 27. var cat = FactoryMethod("cat"); |

# 原型与继承

## 3.1 原型

|  |
| --- |
| 1. /\*\* 2. \* 原型: prototype 3. \* 每一个函数天生都可以调用原型属性， 它的值是一个对象 4. \* 原型就好像是一个公共区域， 当我们使用构造函数创建对象的时候，我们要将对象的方法添加在原型中 5. \* 这样的话就不需要每次为不同的实例化对象去添加方法了，还能够使每一个对象都可以访问， 6. \* 因此，我们要将公用的属性和方法统一设置在原型中 7. \* 8. \* 当一个对象去调用某个属性和方法的时候，首先从自身开始查找，如果有就直接使用，如果没有 9. \* 到原型中去查找，如果有则使用，如果还没有就到原型的原型中去查找 10. \* 11. \* 原型的作用: 实例共享属性和方法 12. \*\*/ 13. // 定义构造函数 14. function People(name, age, sex) { 15. this.name = name; 16. this.age = age; 17. this.sex = sex; 18. } 19. // 将方法写在原型中 20. People.prototype.sayHello = function() { 21. console.log("你好，朋友"); 22. } 23. // 添加属性 24. People.prototype.title = "nihao"; 25. // 实例化对象 26. var xiaoming = new People("小明", 12, "男"); 27. xiaoming.title = "你好"; 28. xiaoming.sayHello(); 29. console.log(xiaoming.title); |

继承：

|  |
| --- |
| 1. // 定义父类 2. function People(name, age, sex) { 3. this.name = name; 4. this.age = age; 5. this.sex = sex; 6. } 7. // 方法要写在原型中 8. People.prototype.sayHello = function() { 9. console.log("大家好， 我的名字叫做" + this.name + "我的年龄" + this.age); 10. } 11. // 定义子类 12. function Student(name, age, sex, grade) { 13. // 转为数组 14. var arr = [].slice.call(arguments); 15. // 使用构造函数式继承 16. People.apply(this, arr); 17. // 子类独有的属性要单独定义 18. this.grade = grade; 19. } 20. // 继承: 使用类式继承 21. // Student.prototype = new People(); 22. // 在ES5中提供了一个方法: Object.create() 23. // Student.prototype = Object.create(People.prototype); 24. // 使用类式继承会丢失原本的constructor属性 25. // 手工补回 26. // Student.prototype.constructor = Student; 27. // 实例化对象 28. // var s = new Student("小明", 12, "男", 6); 29. // console.log(s.constructor); 30. // s.sayHello(); 31. // 寄生组合式继承 32. // 定义一个继承函数 33. function jisheng(father, child) { 34. // 定义新的构造函数 35. var F = function() {}; 36. // 将该构造函数的原型执行父类的原型 37. F.prototype = father.prototype; 38. // 将子类的原型执行该构造函数的实例 39. child.prototype = new F(); 40. // 补回child的constructor属性 41. child.prototype.constructor = child; 42. // 返回child 43. return child; 44. } 45. // 执行 46. jisheng(People, Student); 47. // 实例化对象 48. var s = new Student("小明", 12, "男", 6); 49. // 检测s是否是Student的实例 50. console.log(s instanceof Student); 51. // 检测s是否是People的实例 52. console.log(s instanceof People); |

# 四、安全工厂

|  |
| --- |
| 1. function People(name, age, sex) { 2. // 通过判断this指向决定代码如何执行 3. if (this instanceof People) { 4. // 说明作为构造函数来使用, 一切照旧 5. this.name = name; 6. this.age = age; 7. this.sex = sex; 8. } else { 9. // 说明当做普通函数来使用， 普通函数想要返回内容要使用return 10. return new People(name, age, sex); 11. } 12. } 13. // 测试 14. // 通过new 使用 15. var p = new People("小明", 12, "男"); 16. // 不使用new 调用 17. var p1 = People("小明1", 12, "男") |

## 下午复习：

设计模式：

是一套编目分明、广为人知，可复用的代码经验的总结

工具库：

是一些常用方法的集合体，例如jquery、underscore，这些方法之间通常是没有联系的

框架：

一套半成品代码，里面也支持一些方法，但是这些方法之间是有联系的

架构：

一套大型项目的设计思路

设计模式的分类以及作用：

所有的设计模式可以分为三类：

1 创建型设计模式： 解决了创建对象时候的问题

2 结构型设计模式： 解决了对象和类组合在一起时候的问题

3 行为型设计模式： 解决了对象和类耦合、职责之间的关系

简单工厂： 其实就是创建对象的函数

寄生增强工厂：定义一个新的工厂，在该工厂内部去实例化另外一个工厂并且对其改造

工厂方法: 可以管理多个工厂的实例化创建

原型：是每一个函数天生可以拥有的属性，它的值是一个对象

特点： 原型中的所有内容都可以被实例化对象所访问到

继承：类与类之间的继承

继承的方式有：

1 类式继承（原型式继承）

2 构造函数式继承

3 组合式继承： 类式继承 + 构造函数式继承

4 寄生式继承

5 寄生组合式继承： 定义类的时候使用构造函数式继承

安全工厂： 其实就是安全类

# 五、闭包类

|  |
| --- |
| 1. // 一个闭包 2. (function() { 3. })() 4. // 一个类 5. function People() { 6. } 7. // 一个闭包类 8. (function() { 9. function People() { 10. } 11. })() 12. // 如果把People放在全局中，就可以被别人访问到，就可以无限初始化 13. // 如果将People放在闭包中，别人就无法访问到，也就无法初始化 |

## 5.1 单例模式

|  |
| --- |
| 1. var single = (function() { 2. function People(name, age, sex) { 3. this.name = name; 4. this.age = age; 5. this.sex = sex; 6. } 7. // 此时，将People放入到闭包中，别人无法访问到也就无法实例化了 8. // 但是，我们自己也无法访问到，我们要向外部提供一个接口用来访问该People类 9. // 定义变量 用于存储单例 10. var instance = null; 11. // 定义接口函数 12. return function(name, age, sex) { 13. // 判断单例是否存在 14. if (instance == null) { 15. return instance = new People(name, age, sex); 16. } 17. // 如果存在 直接返回 18. return instance; 19. } 20. })() 21. // 实例化对象 22. var p = single("小明", 12, "男"); 23. var p1 = single("小红", 12, "女"); 24. var p2 = single("小红", 12, "女"); 25. var p3 = single("小红", 12, "女"); 26. var p4 = single("小红", 12, "女"); |

结果：

|  |
| --- |
|  |

## 5.2 普通单例

|  |
| --- |
| 1. // 定义闭包类 2. var single = (function() { 3. // 定义一个类 4. function People(name, age, sex) { 5. this.name = name; 6. this.age = age; 7. this.sex = sex; 8. console.log("普通单例"); 9. } 10. // 定义变量用于保存单例 11. var instance = new People("小明", 12, "男"); 12. // 定义接口函数 13. return function() { 14. // 由于已经初始化一次，所以不要在初始化了 15. return instance; 16. } 17. })() |

执行代码：

|  |
| --- |
| 1. // 执行single 2. var p = single(); 3. var p1 = single(); 4. var p2 = single(); |

结果：

|  |
| --- |
|  |

## 5.3 惰性单例

|  |
| --- |
| 1. // 闭包类 2. var single = (function() { 3. // 定义类 4. function People(name, age, sex) { 5. this.name = name; 6. this.age = age; 7. this.sex = sex; 8. console.log("惰性单例"); 9. } 10. // 定义变量用于保存单例 11. var instance = null; 12. // 定义返回的接口 13. return function(name, age, sex) { 14. // 判断单例是否存在 15. if (!instance) { 16. return instance = new People(name, age, sex); 17. } 18. // 如果单例存在， 直接返回 19. return instance; 20. } 21. })() |

执行代码：

|  |
| --- |
| 1. // 执行single 2. var p = single("小明", 12, "男"); 3. var p1 = single("小明1", 12, "男"); |

结果：

|  |
| --- |
|  |

# 六、静态变量

|  |
| --- |
| 1. // 在ES5中 2. var obj = { 3. name: "老王", 4. length: 180, 5. age: 30 6. } 7. // 如何能够使obj中的结构不被改变 8. // 冻结对象 9. Object.freeze(obj); 10. // 拓展属性 11. obj.a = 123; 12. // 修改属性值 13. obj.age = 40; 14. // 删除属性 15. delete obj.name; 16. // 查看对象是否被冻结 17. console.log(Object.isFrozen(obj)); 18. obj = 123; 19. // 在ES3.1中 20. var Conf = (function() { 21. var obj = { 22. name: "老王", 23. length: 180, 24. age: 30, 25. a: { 26. } 27. } 28. // 定义接口 29. return function(name) { 30. return obj[name]; 31. } 32. })() |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |