面向对象

- 首先，我们要明确，面向对象不是语法，是一个思想，是一种 **编程模式**
- 面向： 面（脸），向（朝着）
- 面向过程： 脸朝着过程 =》 关注着过程的编程模式
- 面向对象： 脸朝着对象 =》 关注着对象的编程模式
- 实现一个效果
  - 在面向过程的时候，我们要关注每一个元素，每一个元素之间的关系，顺序，。。。
  - 在面向过程的时候，我们要关注的就是找到一个对象来帮我做这个事情，我等待结果
- 例子 ： 我要吃面条
  - 面向过程
    - 用多少面粉
    - 用多少水
    - 怎么和面
    - 怎么切面条
    - 做开水
    - 煮面
    - 吃面
  - 面向对象
    - 找到一个面馆
    - 叫一碗面
    - 等着吃
  - 面向对象就是对面向过程的封装
- 我们以前的编程思想是，每一个功能，都按照需求一步一步的逐步完成
- 我们以后的编程思想是，每一个功能，都先创造一个 **面馆**，这个 **面馆** 能帮我们作出一个 **面（完成这个功能的对象）**，然后用 **面馆** 创造出一个 **面**，我们只要等到结果就好了

创建对象的方式

- 因为面向对象就是一个找到对象的过程
- 所以我们先要了解如何创建一个对象

调用系统内置的构造函数创建对象

- js 给我们内置了一个 Object 构造函数
- 这个构造函数就是用来创造对象的
- 当 构造函数 和 new 关键字连用的时候，就可以为我们创造出一个对象

- 因为 js 是一个动态的语言，那么我们就可以动态的向对象中添加成员了

```
// 就能得到一个空对象
var o1 = new Object()

// 正常操作对象
o1.name = 'Jack'
o1.age = 18
o1.gender = '男'
```

字面量的方式创建一个对象

- 直接使用字面量的形式，也就是直接写 `{}`
- 可以在写的时候就添加好成员，也可以动态的添加

```
// 字面量方式创建对象
var o1 = {
  name: 'Jack',
  age: 18,
  gender: '男'
}

// 再来一个
var o2 = {}
o2.name = 'Rose'
o2.age = 20
o2.gender = '女'
```

使用工厂函数的方式创建对象

- 先书写一个工厂函数
- 这个工厂函数里面可以创造出一个对象，并且给对象添加一些属性，还能把对象返回
- 使用这个工厂函数创造对象

```
// 1. 先创建一个工厂函数
function createObj() {
  // 手动创建一个对象
  var obj = new Object()

```

```
  // 手动的向对象中添加成员
  obj.name = 'Jack'
  obj.age = 18
  obj.gender = '男'

  // 手动返回一个对象
  return obj
}

// 2. 使用这个工厂函数创建对象
var o1 = createObj()
var o2 = createObj()
```

使用自定义构造函数创建对象

- 工厂函数需要经历三个步骤
  - 手动创建对象
  - 手动添加成员
  - 手动返回对象
- 构造函数会比工厂函数简单一下
  - 自动创建对象
  - 手动添加成员
  - 自动返回对象
- 先书写一个构造函数
- 在构造函数内向对象添加一些成员
- 使用这个构造函数创造一个对象（和 new 连用）
- 构造函数可以创建对象，并且创建一个带有属性和方法的对象
- 面向对象就是要想办法找到一个有属性和方法的对象
- 面向对象就是我们自己制造 **构造函数** 的过程

```
// 1. 先创造一个构造函数
function Person(name, gender) {
  this.age = 18
  this.name = name
```

```
    this.gender = gender
}

// 2. 使用构造函数创建对象
var p1 = new Person('Jack', 'man')
var p2 = new Person('Rose', 'woman')
```

构造函数详解

- 我们了解了对象的创建方式
- 我们的面向对象就是要么能直接得到一个对象
- 要么就弄出一个能创造对象的东西，我们自己创造对象
- 我们的构造函数就能创造对象，所以接下来我们就详细聊聊 **构造函数**

构造函数的基本使用

- 和普通函数一样，只不过 **调用的时候要和 new 连用**，不然就是一个普通函数调用

```
function Person() {}
var o1 = new Person() // 能得到一个空对象
var o2 = Person() // 什么也得不到，这个就是普通函数调用
```

  - 注意： **不写 new 的时候就是普通函数调用，没有创造对象的能力**

- 首字母大写

```
function person() {}
var o1 = new person() // 能得到一个对象

function Person() {}
var o2 = new Person() // 能得到一个对象
```

  - 注意： **首字母不大写，只要和 new 连用，就有创造对象的能力**

- 当调用的时候如果不需要传递参数可以不写 `()` ，建议都写上

```
function Person() {}
var o1 = new Person() // 能得到一个空对象
var o2 = new Person // 能得到一个空对象
```

  - 注意： **如果不需要传递参数，那么可以不写 （），如果传递参数就必须写**

- 构造函数内部的 this，由于和 new 连用的关系，是指向当前实例对象的

```
function Person() {
```

```
    console.log(this)
  }
  var o1 = new Person() // 本次调用的时候，this => o1
  var o2 = new Person() // 本次调用的时候，this => o2
```

  - 注意： **每次 new 的时候，函数内部的 this 都是指向当前这次的实例化对象**

- 因为构造函数会自动返回一个对象，所以构造函数内部不要写 return
  - 你如果 return 一个基本数据类型，那么写了没有意义
  - 如果你 return 一个引用数据类型，那么构造函数本身的意义就没有了

使用构造函数创建一个对象

- 我们在使用构造函数的时候，可以通过一些代码和内容来向当前的对象中添加一些内容

```
function Person() {
  this.name = 'Jack'
  this.age = 18
}

var o1 = new Person()
var o2 = new Person()
```

  - 我们得到的两个对象里面都有自己的成员 **name** 和 **age**

- 我们在写构造函数的时候，是不是也可以添加一些方法进去呢？

```
function Person() {
  this.name = 'Jack'
  this.age = 18
  this.sayHi = function () {
    console.log('hello constructor')
  }
}

var o1 = new Person()
var o2 = new Person()
```

  - 显然是可以的，我们的到的两个对象中都有 `sayHi` 这个函数
  - 也都可以正常调用

- 但是这样好不好呢？缺点在哪里？

```
function Person() {
  this.name = 'Jack'
```

```
    this.age = 18
    this.sayHi = function () {
    console.log('hello constructor')
    }
}

// 第一次 new 的时候， Person 这个函数要执行一遍
// 执行一遍就会创造一个新的函数，并且把函数地址赋值给 this.sayHi
var o1 = new Person()

// 第二次 new 的时候， Person 这个函数要执行一遍
// 执行一遍就会创造一个新的函数，并且把函数地址赋值给 this.sayHi
var o2 = new Person()
```

  - 这样的话，那么我们两个对象内的 `sayHi` 函数就是一个代码一摸一样，功能一摸一样
  - 但是是两个空间函数，占用两个内存空间
  - 也就是说 `o1.sayHi` 是一个地址，`o2.sayHi` 是一个地址
  - 所以我们执行 `console.log(o1 === o2.sayHi)` 的到的结果是 `false`
  - 缺点： **一摸一样的函数出现了两次，占用了两个空间地址**

- 怎么解决这个问题呢？
  - 就需要用到一个东西，叫做 **原型**

原型

- 原型的出现，就是为了解决 **构造函数的缺点**
- 也就是给我们提供了一个给对象添加函数的方法
- 不然构造函数只能给对象添加属性，不能合理的添加函数就太 LOW 了

prototype

- **每一个函数天生自带一个成员，叫做 prototype，是一个对象空间**
- 即然每一个函数都有，构造函数也是函数，构造函数也有这个对象空间
- 这个 `prototype` 对象空间可以由函数名来访问

```
function Person() {}

console.log(Person.prototype) // 是一个对象
```

  - 即然是个对象，那么我们就可以向里面放入一些东西

```
function Person() {}

Person.prototype.name = 'prototype'
Person.prototype.sayHi = function () {}
```

- 我们发现了一个叫做 `prototype` 的空间是和函数有关联的
- 并且可以向里面存储一些东西
- 重点： **在函数的 prototype 里面存储的内容，不是给函数使用的，是给函数的每一个实例化对象使用的**
- 那实例化对象怎么使用能？

__proto__

- **每一个对象都天生自带一个成员，叫做 `__proto__` ，是一个对象空间**
- 即然每一个对象都有，实例化对象也是对象，那么每一个实例化对象也有这个成员
- 这个 `__proto__` 对象空间是给每一个对象使用的
- 当你访问一个对象中的成员的时候
  - 如果这个对象自己本身有这个成员，那么就会直接给你结果
  - 如果没有，就会去 `__proto__` 这个对象空间里面找，里面有的话就给你结果
  - 未完待续。。。
- 那么这个 `__proto__` 又指向哪里呢？
  - 这个对象是由哪个构造函数 new 出来的
  - 那么这个对象的 `__proto__` 就指向这个构造函数的 `prototype`

```
function Person() {}

var p1 = new Person()

console.log(p1.__proto__ === Person.prototype) // true
```

  - 我们发现实例化对象的 `__proto__` 和所属的构造函数的 `prototype` 是一个对象空间
  - 我们可以通过构造函数名称来向 `prototype` 中添加成员
  - 对象在访问的时候自己没有，可以自动去自己的 `__proto__` 中查找
  - 那么，我们之前构造函数的缺点就可以解决了
    - 我们可以把函数放在构造函数的 `prototype` 中
    - 实例化对象访问的时候，自己没有，就会自动去 `__proto__` 中找
    - 那么也可以使用了

```
function Person() {}

Person.prototype.sayHi = function () {
 console.log('hello Person')
}

var p1 = new Person()
p1.sayHi()
```

- `p1` 自己没有 `sayHi` 方法，就会去自己的 `__proto__` 中查找
- `p1.__proto__` 就是 `Person.prototype`
- 我们又向 `Person.prototype` 中添加了 `sayHi` 方法
- 所以 `p1.sayHi` 就可以执行了

- 到这里，当我们实例化多个对象的时候，每个对象里面都没有方法
  - 都是去所属的构造函数的 `prototttype` 中查找
  - 那么每一个对象使用的函数，其实都是同一个函数
  - 那么就解决了我们构造函数的缺点

```
function Person() {}

Person.prototype.sayHi = function () {
 console.log('hello')
}

var p1 = new Person()
var p2 = new Person()

console.log(p1.sayHi === p2.sayHi)
```

  - `p1` 是 `Person` 的一个实例
  - `p2` 是 `Person` 的一个实例
  - 也就是说 `p1.__proto__` 和 `p2.__proto__` 指向的都是 `Person.prototype`
  - 当 `p1` 去调用 `sayHi` 方法的时候是去 `Person.prototype` 中找
  - 当 `p2` 去调用 `sayHi` 方法的时候是去 `Person.prototype` 中找
  - 那么两个实例化对象就是找到的一个方法，也是执行的一个方法

- 结论
  - 当我们写构造函数的时候
  - **属性我们直接写在构造函数体内**
  - **方法我们写在原型上**

原型链

原型链

- 我们刚才聊过构造函数了，也聊了原型
- 那么问题出现了，我们说构造函数的 `prototype` 是一个对象
- 又说了每一个对象都天生自带一个 `__proto__` 属性
- 那么 **构造函数的 prototype** 里面的 `__proto__` 属性又指向哪里呢？

一个对象所属的构造函数

- 每一个对象都有一个自己所属的构造函数
- 比如： 数组

```
// 数组本身也是一个对象
var arr = []
var arr2 = new Array()
```

  - 以上两种方式都是创造一个数组
  - 我们就说数组所属的构造函数就是 `Array`

- 比如： 函数

```
// 函数本身也是一个对象
var fn = function () {}
var fun = new Function()
```

  - 以上两种方式都是创造一个函数
  - 我们就说函数所属的构造函数就是 `Function`

constructor

- 对象的 `__proto__` 里面也有一个成员叫做 **`constructor`**
- 这个属性就是指向当前这个对象所属的构造函数

链状结构

- 当一个对象我们不知道准确的是谁构造的时候，我们呢就把它看成 `Object` 的实例化对象
- 也就是说，我们的 **构造函数 的 prototype 的** `__proto__` 指向的是 `Object.prototype`
- 那么 `Object.prototype` 也是个对象，那么它的 `__proto__` 又指向谁呢？
- 因为 `Object` 的 js 中的顶级构造函数，我们有一句话叫 **万物皆对象**
- 所以 `Object.prototype` 就到顶了，`Object.prototype` 的 `__proto__` 就是 null

原型链的访问原则

- 我们之前说过，访问一个对象的成员的时候，自己没有就会去 `__proto__` 中找
- 接下来就是，如果 `__proto__` 里面没有就再去 `__proto__` 里面找
- 一直找到 `Object.prototype` 里面都没有，那么就会返回 `undefiend`

对象的赋值

- 到这里，我们就会觉得，如果是赋值的话，那么也会按照原型链的规则来
- 但是： **并不是！并不是！并不是！** 重要的事情说三遍
- 赋值的时候，就是直接给对象自己本身赋值
  - 如果原先有就是修改
  - 原先没有就是添加
  - 不会和 `__proto__` 有关系

总结

- 到了这里，我们就发现了面向对象的思想模式了

  - 当我想完成一个功能的时候
  - 先看看内置构造函数有没有能给我提供一个完成功能对象的能力
  - 如果没有，我们就自己写一个构造函数，能创造出一个完成功能的对象
  - 然后在用我们写的构造函数 new 一个对象出来，帮助我们完成功能就行了

- 比如： tab选项卡

  - 我们要一个对象
  - 对象包含一个属性：是每一个点击的按钮
  - 对象包含一个属性：是每一个切换的盒子
  - 对象包含一个方法：是点击按钮能切换盒子的能力
  - 那么我们就需要自己写一个构造函数，要求 new 出来的对象有这些内容就好了
  - 然后在用构造函数 new 一个对象就行了