原型链

单例模式

- 单例模式定义：
  - 单例模式功能相对单一,用来描述具体一个事务。
  - 单例 — 单独的实例
  - 实例:把相同的事务总结(归纳、抽象)出来，形成一类事务，把描述事务的属性和方法具体化，这个具体的描述的对象就是实例
- 单例模式的优势:

> 1.解决命名冲突
> 2.把相同事务归为了一类，并且把这些属性或者方法
> 放到了一个堆内存空间中存储。
> 3.模块化的开发

- 命名冲突:
  - 1.封闭空间

> 把一段代码放到一个函数内,当执行函数的时候
> 函数内的域和外界是互不干扰的
> (function(){})()

  - 2.命名空间

> 把一些变量或者函数变成某个对象下的属性和方法
> 对象与对象之间空间地址是不一样的，所以可以解决
> 命名冲突的问题

- 表示方法：

```
变量 let a = 20;
属性 obj.a = 20;

函数:function fn(){}
方法:obj.fn = function(){}
```

高级单例模式

- 定义：
  - 高级单例模式，可以实现高内聚、低耦合
- 匿名函数自执行：使用一个匿名函数自执行函数，这个函数返回一个对象

工厂模式

- 定义： 目的是批量生成多个实例, 通过传参去描述具体的实例, 把生产后的对象返回到外界使用。

```
function fn(name,age){
    let obj = {}; //原材料
    //加工
    obj.name = name;
    obj.age = age;
    //出厂
    return obj;
}

let 机器人1号 = fn('李永梅',8);
let 机器人2号 = fn('席超',8);

console.log(机器人1号);
console.log(机器人2号)
```

构造函数

- 定义：把属性或者方法挂在this上，然后去new这个函数
  浅规则是构造函数首字母大写。

- 执行函数：不使用()调用也是可以执行函数的，此时的()只是为了传参
- **构造函数(fn)中的this指向了当前实例。** 跟普通函数比较把默认的window转成当前实例

```
function Fn(){}  构造函数
fn{}  实例化对象
```

- **return 的结果默认指向当前实例this,**有return ， 如果后面跟着的是一个基本类型结果依然是实例,如果后面跟着的是一个引用 类型，那么结果就是这个return后的引用类型

面向对象

- 定义：把描述相同的事务抽象出来，归为一类，把描述这个类的属性和方法挂在这个类的原型(prototype)上的一种编程方式就叫面向对象
- 抽象: 抽离出长的相像的部分。
- js的面向对象有特征:
  - 抽象
  - 封装
  - 继承
  - 多态
- 类： 构造函数 -> 把相同的代码抽离出来归纳在一个函数中

```
//类
function Fn(name,age){
    this.name = name;
    this.age = age;
}

//实例
let 机器人1号 = new Fn('李永梅',8);

console.log(机器人1号);
```

原型

- 定义：当定义一个函数的时候,这个函数自身有一些属性 或者方法，其中有一个属性 叫做prototype,这个 属性就叫原型
- **prototype它是一个对象**
- 使用原型的目的:

> 当构造函数中添加方法的时候，每new一次就生成一个同类
> 方，这些方法虽然同类但是各自不相等，这就导致如果new若干次，
> 那么就会生成若干个一摸一样的方法，这样对性能是不友好的
> 所以我们要使用原型的方式去把方法挂在原型上。

- 使用原型的用处：

> 它的用处是如果实例化对象上没有某个属性或者方法，还会去这个实例化对象
> 的构造函数中的原型下去查找该属性或者方法.

- 如果构造函数的原型上没有这个方法，那么还会去原型下的原型链（***_proto_***）中查找，找到Object.prototype
- **构造函数的原型下的方法只给它的实例化对象使用**
- 以构造函数模式 + 原型模式 = js面向对象模式

小复习

- 面向对象定义：

> 把相同的事务抽象出来归为一类，把描述它具体特征的属性或者方法
> 挂在这个类的原型上的一种编程方式。

- 原型:

> 只要是个函数，函数下都有一个叫prototype的属性
> **值为对象**，这个原型下的方法或者属性只有2种方式可以 获取

```
1.obj.prototype.xx();
2.new obj().xx()
```

- 原型链
  proto 实例上一定有原型链
  - 重点：**实例化对象上的原型链 === 构造函数的原型**
  - 相当于：对象.***_proto_*** === 函数.prototype
    代码：找原型链：

```
function Gui(name){
  this.name = name;
    this.jiao = '4只';
    this.keer = '椭圆壳儿';
```

```
    this.zishi = '爬';
}

 Gui.prototype.shi = function(食物){
     console.log(this.name+'吃'+食物);
 }

 Object.prototype.shi = function(){
    console.log(this.name+'喝水');
 };
 let g1 = new Gui('草龟');//墨
 let g2 = new Gui('鳄龟');
 g1.shi('啥都吃');
 g2.shi('泥鳅');
 console.dir(g1);
```

例题及定义

> 例题

```
function fn(name) {
     this.name = name;
      this.say = function () {
          console.log(this.name);
      }
  }
  fn.prototype.say = function () {
      console.log('原型下的');
  }
  Object.prototype.say = function () {
      console.log('对象的');
  }
  fn.say = function () {
      console.log('fn的')
  }
  Function.prototype.say = function () {
      console.log('Func');
  }
  let f = new fn();
  f.say = function () {
      console.log('自己的');
  }
  fn.say();
  Function.say();
```

- 结论方法：
- 结论一：
  - **1.先看对象自身有没有这个属性或者方法**
    有就不找了
    没有的话就接着找
  - **2.通过对象的原型链找构造函数的原型**
    有就不找了
    没有的话就接着找
  - **3.因为函数的原型是个对象，对象身上有原型链，通过原型链又去找构造函数原型**
    有就不找了
    没有的话就接着找
- 结论二：

```
new Function -> function.say
fn.prototyoe.say -> new fn
Function.prototype.say -> new Function -> function
function fn(){} -> new Function
```

- 结论三：
  函数即是函数，又是对象(它是Function的实例化对象)
  函数即有原型，也有原型链，函数的原型上的属性或者方法只给它的
  实例化对象使用。