正则

JavaScript 中使用正则表达式

创建正则对象

方式1:

```
var reg = new Regex('\d', 'i');
var reg = new Regex('\d', 'gi');
```

方式2:

```
var reg = /\d/i;
var reg = /\d/gi;
```

参数

标志	说明
i	忽略大小写
g	全局匹配
gi	全局匹配+忽略大小写

正则匹配

```
// 匹配日期
var dateStr = '2015-10-10';
var reg = /^\d{4}-\d{1,2}-\d{1,2}$/
console.log(reg.test(dateStr));
```

常用元字符串

元字符	说明
\d	匹配数字
\D	匹配任意非数字的字符
\w	匹配字母或数字或下划线
\W	匹配任意不是字母，数字，下划线
\s	匹配任意的空白符
\S	匹配任意不是空白符的字符
.	匹配除换行符以外的任意单个字符
^	表示匹配行首的文本(以谁开始)
$	表示匹配行尾的文本(以谁结束)

限定符

限定符	说明
*	重复零次或更多次
+	重复一次或更多次
?	重复零次或一次
{n}	重复n次
{n,}	重复n次或更多次
{n,m}	重复n到m次

其它

[] 字符串用中括号括起来，表示匹配其中的任一字符，相当于或的意思
[^] 匹配除中括号以内的内容
\ 转义符

| 或者，选择两者中的一个。注意|将左右两边分为两部分，而不管左右两边有多长多乱
() 从两个直接量中选择一个，分组
eg：gr(ale)y匹配gray和grey
[\u4e00-\u9fa5] 匹配汉字

正则表达式的作用

1. 给定的字符串是否符合正则表达式的过滤逻辑(**匹配**)
2. 可以通过正则表达式，从字符串中获取我们想要的特定部分(**提取**)
3. 强大的字符串替换能力(**替换**)

[在线测试正则]

递归应用场景

- 深拷贝
- 菜单树
- 遍历 DOM 树

函数闭包

```
function fn () {
 var count = 0
 return {
   getCount: function () {
     console.log(count)
   },
   setCount: function () {
     count++
   }
 }
}

var fns = fn()

fns.getCount() // => 0
fns.setCount()
fns.getCount() // => 1
```

```
准确获取对象类型
console.log(Object.prototype.toString.call(obj));
```

call bind apply 一个program解释清楚

```
function addNumbers(a, b) {
    console.log(this.num);
    console.log(this.num + a + b);
  }

var obj = {
    num: 10
};

addNumbers.apply(obj, [1,2]); //13 // 后面是数组为参数，参数个数不够则报错，超出则舍弃超出的部分，运行
addNumbers.call(obj, 1, 2);   //13 'call' is most same as 'apply'
let bindResult = addNumbers.bind(obj, 1, 2);
bindResult()
```

函数的调用方式

- 普通函数
- 构造函数
    - 对象方法

函数内 `this` 指向的不同场景

函数的调用方式决定了 `this` 指向的不同：

调用方式	非严格模式	备注
普通函数调用	window	严格模式下是 undefined
构造函数调用	实例对象	原型方法中 this 也是实例对象
对象方法调用	该方法所属对象	紧挨着的对象
事件绑定方法	绑定事件对象	
定时器函数	window	

这就是对函数内部 this 指向的基本整理，写代码写多了自然而然就熟悉了。

函数也是对象

- 所有函数都是 `Function` 的实例

继承

构造函数的属性继承：借用构造函数

```
function Person (name, age) {
 this.type = 'human'
 this.name = name
 this.age = age
```

```
}

function Student (name, age) {
 // 借用构造函数继承属性成员
 Person.call(this, name, age)
}

// 原型对象拷贝继承原型对象成员
for(var key in Person.prototype) {
  Student.prototype[key] = Person.prototype[key]
}

var s1 = Student('张三', 18)

s1.sayName() // => hello 张三

// 利用原型的特性实现继承
Student.prototype = new Person()

var s1 = Student('张三', 18)
```

贪吃蛇例子，写一个js贪吃蛇，以此来熟悉js用法。

重置原型对象的写法

```
function Person (name, age) {
  this.name = name
  this.age = age
}

Person.prototype = {
  constructor: Person, // => 手动将 constructor 指向正确的构造函数
  type: 'human',
  sayHello: function () {
    console.log('我叫' + this.name + ', 我今年' + this.age + '岁了')
  }
}
```

原生对象的原型

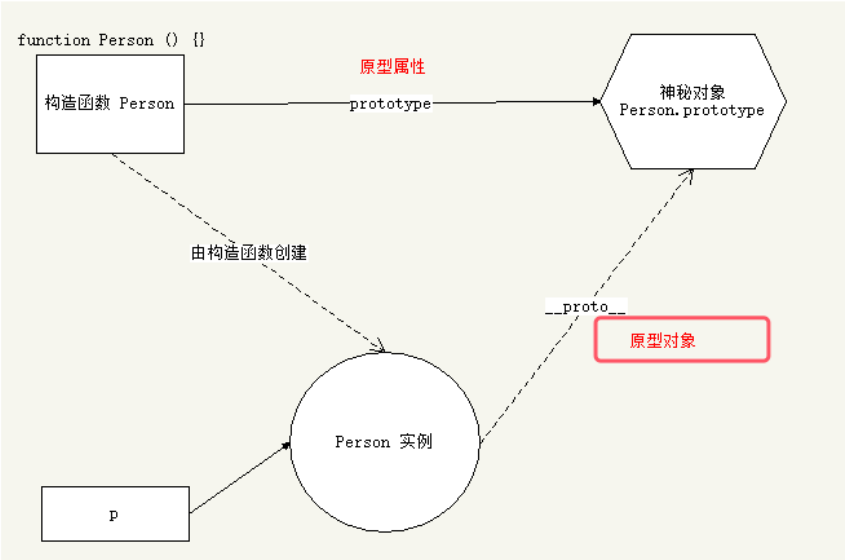
所有函数都有 prototype 属性对象。

- Object.prototype
- Function.prototype
- Array.prototype
- String.prototype
- Number.prototype
- Date.prototype
- ...

练习：为数组对象和字符串对象扩展原型方法。

原型链

构造函数、实例、原型三者之间的关系



实例对象读写原型对象成员

读取：

- 先在自己身上找，找到即返回

- 自己身上找不到，则沿着原型链向上查找，找到即返回
- 如果一直到原型链的末端还没有找到，则返回 undefined

值类型成员写入（实例对象.值类型成员 = xx）：

- 当实例期望重写原型对象中的某个普通数据成员时实际上会把该成员添加到自己身上
- 也就是说该行为实际上会屏蔽掉对原型对象成员的访问

引用类型成员写入（实例对象.引用类型成员 = xx）：

- 同上

复杂类型修改（实例对象.成员.xx = xx）：

- 同样会先在自己身上找该成员，如果自己身上找到则直接修改
- 如果自己身上找不到，则沿着原型链继续查找，如果找到则修改
- 如果一直到原型链的末端还没有找到该成员，则报错（实例对象.undefined.xx = xx）

更好的解决方案： prototype

Javascript 规定，每一个构造函数都有一个 prototype 属性，指向另一个对象。

这个对象的所有属性和方法，都会被构造函数的实例继承。

这也就意味着，我们可以把所有对象实例需要共享的属性和方法直接定义在 prototype 对象上。

Person.prototype.type = 'human'

```
Person.prototype.sayName = function () {
  console.log(this.name)
}
```

任何函数都具有一个 prototype 属性，该属性是一个对象。

```
> var man = new Person(1,2)
< undefined
> man.character
< 'saber'
> man.__proto__
< ▶ {character: 'saber', constructor: f}
> Person.prototype
< ▶ {character: 'saber', constructor: f}
>
```

Object Oriented

简单方式的改进：工厂函数

我们可以写一个函数，解决代码重复问题：

```
function createPerson (name, age) {
  return {
    name: name,
    age: age,
    sayName: function () {
      console.log(this.name)
    }
  }
}
```

更优雅的工厂函数：构造函数

一种更优雅的工厂函数就是下面这样，构造函数：

```
function Person (name, age) {
  this.name = name
  this.age = age
  this.sayName = function () {
    console.log(this.name)
  }
}

var p1 = new Person('Jack', 18)
p1.sayName() // => Jack

var p2 = new Person('Mike', 23)
p2.sayName() // => Mike
```

通过工厂模式我们解决了创建多个相似对象代码冗余的问题，但却没有解决对象识别的问题（即怎样知道一个对象的类型）。

下面是具体的伪代码：

```
function Person (name, age) {
  // 当使用 new 操作符调用 Person() 的时候，实际上这里会先创建一个对象
  // var instance = {}
  // 然后让内部的 this 指向 instance 对象
  // this = instance
  // 接下来所有针对 this 的操作实际上操作的就是 instance

  this.name = name
  this.age = age
  this.sayName = function () {
    console.log(this.name)
  }

  // 在函数的结尾处会将 this 返回，也就是 instance
  // return this
}
```

构造函数和实例对象的关系

使用构造函数的好处不仅仅在于代码的简洁性，更重要的是我们可以识别对象的具体类型了。

在每一个实例对象中的*__proto__*中同时有一个 constructor 属性，该属性指向创建该实例的构造函数：

```
console.log(p1.constructor === Person) // => true
console.log(p2.constructor === Person) // => true
console.log(p1.constructor === p2.constructor) // => true
```

对象的 constructor 属性最初是用来标识对象类型的，

但是，如果要检测对象的类型，还是使用 instanceof 操作符更可靠一些：

```
console.log(p1 instanceof Person) // => true
console.log(p2 instanceof Person) // => true
```

推荐使用 instanceof 操作符，后面学原型会解释为什么

对于这种问题我们可以把需要共享的函数定义到构造函数外部：

```
function sayHello = function () {
  console.log('hello ' + this.name)
}

function Person (name, age) {
  this.name = name
  this.age = age
  this.type = 'human'
  this.sayHello = sayHello
}

var p1 = new Person('lpz', 18)
var p2 = new Person('Jack', 16)

console.log(p1.sayHello === p2.sayHello) // => true
```