JavaScript基础

变量类型和计算

q:JS中使用 `typeof` 的类型?

1. 基本类型

```
`undefined` `null` `boolean` `number` `string`
```

2.引用类型

```
`object`
`对象` `数组` `函数`
```

```
typeof undefined; //undefined
typeof 'abc'; //string
typeof 123; //number
typeof true; //boolean
typeof {}; //object
typeof []; //object
typeof null; //object 引用类型
typeof console.log; //function
//typeof 只能区分基本类型，无法区分对象、数组、null这三种引用类型
```

q:何时使用 `===` 何时使用 `==`

```
//字符串拼接类型转换
var a = 100 + 10 //110
var b = 100 + '10' //10010
```

```
// ==号
100 == '100' //true
0 == '' //true
null == undefined //true
```

```
//if语句
var a = true;
if (a) {
  //
}
var b = 100;
if (b) {
  //b=true
}
var c = '';
if (c) {
  //c=false
}
```

```
//逻辑运算符
console.log(10 && 0) // 0
console.log('' || 'abc') // 'abc'
console.log(!window.abc) // true (当window.abc=undefined时)
// 判断一个变量是被当做 `true` 还是 `false`
var m = 100;
console.log(!m) //false
console.log(!!m) //true
```

```
// a:
if (obj.a == null) {
  // 相当于obj.a=== null||obj.a=== undefined, 简写形式
  // 这是jquery源码中推荐的写法
  // 其他情况全部使用 `===`
}
```

q:JS中有哪些 内置函数 -数据封装类对象

```
Object
Array
Boolean
Number
String
Function
Date
RegExp
Error
```

q:JS按照 存储方式 区分为哪些类型，并描述其特点

```
//值类型
var a = 10;
b = a;
a = 11;
console.log(b) //10
//复制不会相互干预
** ** ** ** ** ** ** ** ** *
//引用类型
var obj1 = { x: 100 };
var obj2 = obj1;
obj1.x = 200;
console.log(obj2.x); //200
// 复制是引用类型的指针，会相互干预
```

q:如何理解 JSON

```
// JSON只不过是一个内置的JS对象而已
// JSON也是一种数据格式
JSON.stringify({ a: 100, b: 200 }); // "{"a":100,"b":200}"
JSON.parse('{"a":100,"b":200}'); // {a:100,b:200}
```

原型&&原型链

构造函数

```
function Foo(name, age) {
  this.name = name;
  this.age = age;
  this.class = 'class-1';
  //return this ; //默认有这一行
}
var f = new Foo('张三', 22);
var f1 = new Foo('李四', 29);
```

构造函数 - 扩展

```
var a={} 其实是 var a=new Object() 的语法糖
var a=[] 其实是 var a=new Array() 的语法糖
function Foo() {....} 其实是 var Foo=new Function(...)
使用 instanceof 判断一个函数是否是一个变量的构造函数
```

原型规则和示例

- 所有的引用类型(数组、对象、函数)，都具有对象属性(即可自有扩展的属性)，`null` 除外
- 所有的引用类型(数组、对象、函数)，都有一个 `__proto__` 属性(隐式原型)，属性值是一个普通的对象

```
var obj = { };
obj.x=100;
console.log(obj.__proto__);
// {constructor: f, __defineGetter__: f, __defineSetter__: f, hasOwnProperty: f,
__lookupGetter__: f, …}
var arr = [];
arr.x = 200;
console.log(arr.__proto__);
// [constructor: f, concat: f, find: f, findIndex: f, pop: f, …]
function fn() {};
fn.x = 300;
console.log(fn.__proto__);
// f () { [native code] }
var d = null;
console.log(d.__proto__);
// Uncaught TypeError: Cannot read property '__proto__' of null
```

- 所有的 `函数` ，都有一个 `prototype` 属性(显式原型)，属性值也是一个普通对象

```
console.log(fn.prototype);
// {constructor: f}
```

- 所有的引用类型(数组、对象、函数)， `__proto__` 属性值指向它的构造函数的 `prototype` 属性值

```
console.log(obj.__proto__ === Object.prototype);
// true
```

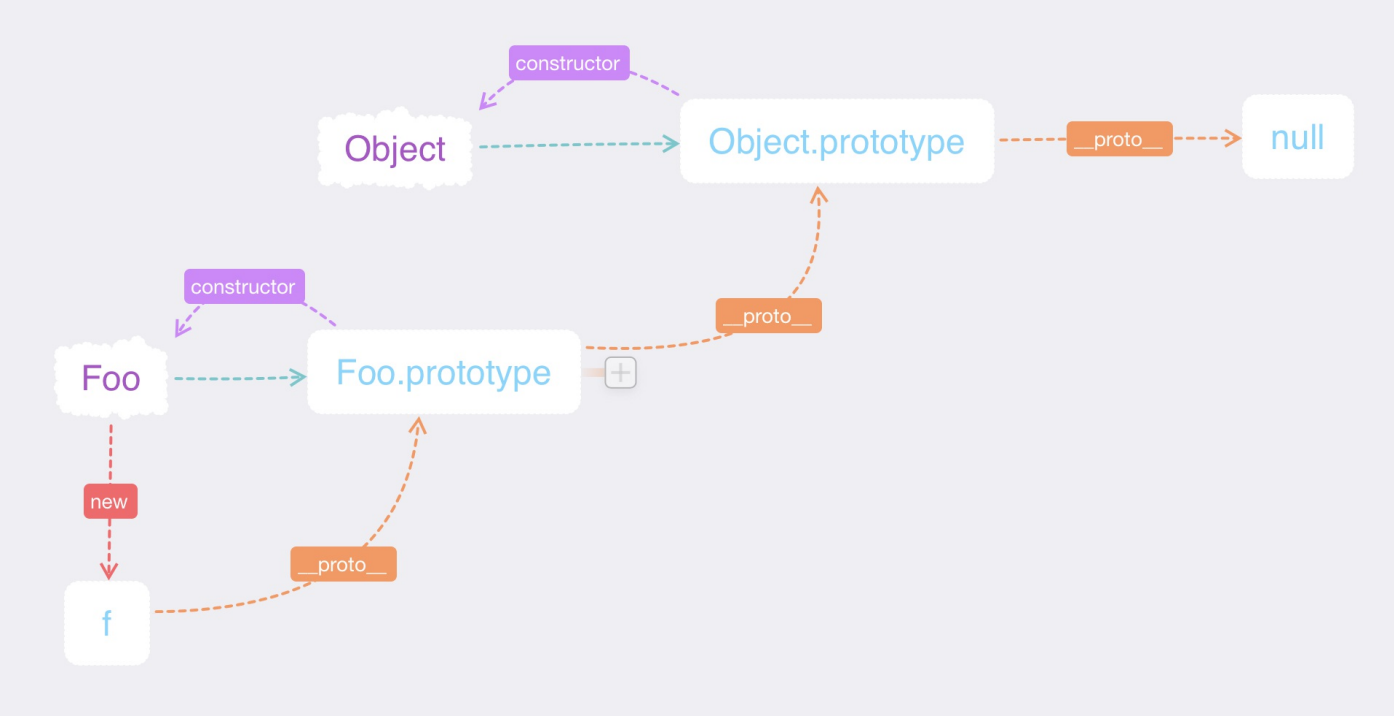
- 当视图得到一个对象(所有的引用类型)的某个属性时，如果这个对象本身没有这个属性，那么会去它的 `__proto__` (即它的构造函数的 `prototype`)中寻找。

```
// 构造函数
function Foo(name, age) {
  this.name = name;
}
Foo.prototype.alertName = function() {
  console.log('alertName' + this.name);
}
// 创建示例
var f = new Foo('张三');
f.prientname = function() {
  console.log('prientname' + this.name);
}
// 测试
f.prientname(); // prientname张三
f.alertName(); // alertName张三
```

原型链

```
// 构造函数
function Foo(name, age) {
  this.name = name;
}
Foo.prototype.alertName = function() {
  console.log('alertName' + this.name);
}
// 创建示例
var f = new Foo('张三');
f.prientname = function() {
  console.log('prientname' + this.name);
}
// 测试
f.prientname(); // prientname张三
f.alertName(); // alertName张三

f.toString(); // "[object Object]" 在f.__proto__.__proto__中查找，即Object的显式原型中寻找
```



instanceof

- `instanceof` 用于判断 `引用类型` 属于哪个 `构造函数` 的方法

```
// f的 __proto__ 一层一层网上找，找到对应的 Foo.prototype
f instanceof Foo //true
f instanceof Object //true
```

q:如何准确判断一个变量是数组类型

```
var arr=[]
// 可以正确判断的情况
arr instanceof Array //true
Object.prototype.toString.call(arr) // "[object Array]"
Object.prototype.toString.apply(arr) // "[object Array]"
Array.isArray(arr) // true
// 不能判断的情况
typeof arr // object 是无法判断是否是数组的
// 不准确
arr.constructor === Array //true 但是原型链可以被改写，这样判断不安全
```

```
// 扩展 兼容老版本浏览器，isArray的写法
if(!Array.isArray){
  Array.isArray = function(arg){
    return Object.property.toString.call(arg) === '[object Array]'
  }
}
```

q:写一个原型链继承的例子

```
function Elem(id) {
  this.elem = document.getElementById(id);
}
Elem.prototype.html = function(val) {
```

```
  var elem = this.elem;
  if (val) {
    elem.innerHTML = val;
    return this; // 后续的链式操作
  } else {
    return elem.innerHTML;
  }
}
Elem.prototype.on = function(type, fn) {
  var elem = this.elem;
  elem.addEventListener(type, fn);
  return this;
}
var main = new Elem('main')
main.html('<p>Hello World</p>').on('click', function() {
  alert('Hello javascript')
})
```

q:描述 `new` 一个对象的过程

- 创建一个对象
- `this` 指向这个新对象
- 执行代码，即对 `this` 赋值
- 返回 `this`

```
function Foo(name, age) {
  this.name = name;
  this.age = age;
  this.class = 'class-1';
  //return this ; //默认有这一行
}
var f = new Foo('张三', 22);
var f1 = new Foo('李四', 29);
```