

JS权威指南学习分享

keep simple and fit

十年javascript

- 1995.4-1995.10: Brendan Eich(Live Script) -> JavaScript
- 1996.8: Microsoft IE3(JScript)
- 1996-1997.7: JavaScript1.1 -> ECMA-262 -> ECMAScript
- 1997.10: IE4(DHTML) -> HTML4.0&CSS&JavaScript 的结合物
- 1999: IE5 (AJAX)
- 2009.12: ES5 & Harmony

JavaScript运行期环境

JavaScript

JavaScript Core

JavaScript Client

权威指南

JavaScript

ECMAScript

DOM

BOM

高级程序设计

JavaScript运行期环境



Content

- JavaScript Core
- JavaScript Client

JavaScript core

- 词法结构
- 数据类型及转换
- 变量、作用域
- 表达式、语句
- 数组、对象
- 函数
- 类、构造器、原型
- 模块、正则

词法结构

- 大小写
- 空白&换行
- 可选分号
- 注释
- 直接量
- 标识符&保留字

例1: `(function(){})()
(function(){})()`

例2: `return
{ key:value };`


数据类型

- 基本数据类型
 - primitive value
- 引用数据类型
 - native object
 - built-in object
 - host object
- 内置数据类型

数据类型

- `String` 是基本类型还是引用类型？
- 何时传值，何时传址？

Primitive Value

- number
 - string
 - boolean
 - null
 - undefined
- 

primitive value is a datum that is represented directly at the lowest level of the language implementation, is a member of one of the following built-in types: Undefined, Null, Boolean, Number, and String;

内置数据类型

- Function
- Object
- Boolean, Number, String
- Date, Array, RegExp
- Math, Error

类型转换

```
if( [0] ) {  
  console.log([0] == true);  
  console.log(!![0]);  
}
```

```
if("potato") {  
  console.log("potato" == false);  
  console.log("potato"== true);  
}
```

```
if("false") {  
  console.log('true');  
}
```

类型转换(if..)

The construct `if (Expression) Statement` will coerce the result of evaluating the *Expression* to a boolean using the abstract method **ToBoolean** for which the [ES5 spec](#) defines the following algorithm:

Argument Type	Result	Conditionals(if..)
Undefined	false	
Null	false	
Boolean	The result equals the input argument (no conversion).	
Number	The result is false if the argument is +0 , -0 , or NaN ; otherwise the result is true .	
String	The result is false if the argument is the empty String (its length is zero); otherwise the result is true .	
Object	true .	

类型转换(==)

Just remember that `undefined` and `null` equal each other (and nothing else) and most other types get coerced to a number to facilitate comparison

Type(x)	Type(y)	Result	The Equals Operator (==)
x and y are the same type		See Strict Equality (===) Algorithm	
null	Undefined	true	
Undefined	null	true	
Number	String	x == toNumber(y)	
String	Number	toNumber(x) == y	
Boolean	(any)	toNumber(x) == y	
(any)	Boolean	x == toNumber(y)	
String or Number	Object	x == toPrimitive(y)	
Object	String or Number	toPrimitive(x) == y	
otherwise...		false	

类型转换 (===)

- different types always false
- object must reference the same object
- strings must contain identical character sets,
- other primitives must share the same value.
- NaN、 null、 undefined will never === another type. NaN does not even === itself.

类型转换 (+、>、<)

1 + 2	// 3
1 + "2"	// 12
"1" + "2"	// 12
11 > 3	//true
"11" > "3"	//false
"11" < 3	//false
"one" < 3	//false
var s = 1 + 2 + " test add";	//3 test add
var t = "test add " + 1 + 2;	//test add 12

传值和传址

	传值	传址
复制	复制的是值，存在两个不同的拷贝	复制的是引用，通过新的引用修改了数值，改变对原引用可见
传递	传递给函数的值是一个独立的拷贝，对它的改变在函数外部不受影响	传递给函数的是一个引用，修改对外部可见
比较	比较的是两个独立的值	比较的是两个引用，以判断他们引用的是否是同一个数值

传值和传址

```
var test = [1, 2, 3];  
function changeArray(a) {  
    a = new Array(1, 2);  
    return a;  
}  
console.log(test);  
console.log(changeArray(test));  
console.log(test);
```

对象和数组是用传值的方式传递的，只不过传递的这个值实际上是一个引用

变量、作用域

- 变量的申明（重复、遗漏）
- 局部变量与变量提升
- 未定义、未声明、未赋值
- `global` && `window`

变量、作用域

- 无块级作用域
- Execution Context: 允许有多个执行环境并相互引用

表达式

- `&&`
- `||`
- `=`

- 1、`a=b`为一个表达式，值为赋值运算符右边的值
- 2、多个赋值运算，将从右向左运算 `a=b=0;`
- 3、避免：`if (a = b) {...}`

语句

- for in 语句
- with 语句
- try/catch/finally

with

```
var o = {x: 1, y:2, z:3 };  
var x = 4;  
var y = 5;  
with(o) {  
    x = z;  
    y = z;  
}  
console.log(o);  
console.log(x);
```

?

语句-delete

不带DontDelete标记

- eval代码块中声明的变量和方法
- 不存在的变量或属性直接赋值产生的对象

带有DontDelete标记

- 变量和函数的声明
- 函数内建的arguments对象
- 其它

对于宿主对象而言，**delete**操作的结果有可能是不可预料的

语句-function

函数创建的算法

内部行为可以描述成如下：

```
F = new NativeObject();

F.[[Class]] = "Function"

.... // 其它属性

F.[[Call]] = <reference to function> // function自身

F.[[Construct]] = internalConstructor // 普通的内部构造函数

.... // 其它属性

// F构造函数创建的对象原型
__objectPrototype = {};
__objectPrototype.constructor = F // {DontEnum}
F.prototype = __objectPrototype
```


语句-new

对象创建的算法

内部方法[[Construct]]的行为可以描述成如下:

```
F.[[Construct]](initialParameters):  
  
O = new NativeObject();  
  
// 属性[[Class]]被设置为"Object"  
O.[[Class]] = "Object"  
  
// 引用F.prototype的时候获取该对象g  
var __objectPrototype = F.prototype;  
  
// 如果__objectPrototype是对象, 就:  
O.[[Prototype]] = __objectPrototype  
// 否则:  
O.[[Prototype]] = Object.prototype;  
// 这里O.[[Prototype]]是Object对象的原型  
  
// 新创建对象初始化的时候应用了F.[[Call]]  
// 将this设置为新创建的对象O  
// 参数和F里的initialParameters是一样的  
R = F.[[Call]](initialParameters); this === O;  
// 这里R是[[Call]]的返回值  
// 在JS里看, 像这样:  
// R = F.apply(O, initialParameters);  
  
// 如果R是对象  
return R  
// 否则  
return O
```

数组与对象

✦ 常规数组和关联数组

```
[3]、 new Array()、 new Array(3)、 new Array(1, 2, 3);  
new Object() or {};
```

💡 <http://localhost/testObject.php>

数组与对象

- `Array.shift`
- `Array.unshift`
- `Array.push`
- `Array.pop`
- `Array.splice`

对象

- 对象动态性
- 由构造函数的内部方法[[Construct]]来创建
- 由构造函数的内部方法[[Call]]来初始化

对象

如何确定一个对象的类型？

确定对象类型

- `typeof`
- `instanceof` and `constructor`
- `toString`

```
function isArray(o) {  
    return Object.prototype.toString.call(o) === '[object Array]';  
}
```

```
TYPES = {  
    'undefined'      : 'undefined',  
    'number'         : 'number',  
    'boolean'        : 'boolean',  
    'string'         : 'string',  
    '[object Function]' : 'function',  
    '[object RegExp]'  : 'regexp',  
    '[object Array]'   : 'array',  
    '[object Date]'    : 'date',  
    '[object Error]'   : 'error'  
},
```

```
L.type = function(o) {  
    return TYPES[typeof o] || TYPES[TOSTRING.call(o)] || (o ? 'object' : 'null');  
};
```

函数

- Arguments对象
- 作为数据的函数
- 作为方法的函数
- 构造函数

函数

- callee
- length
- prototype
- call、 apply, undefined and null

内置的全局函数

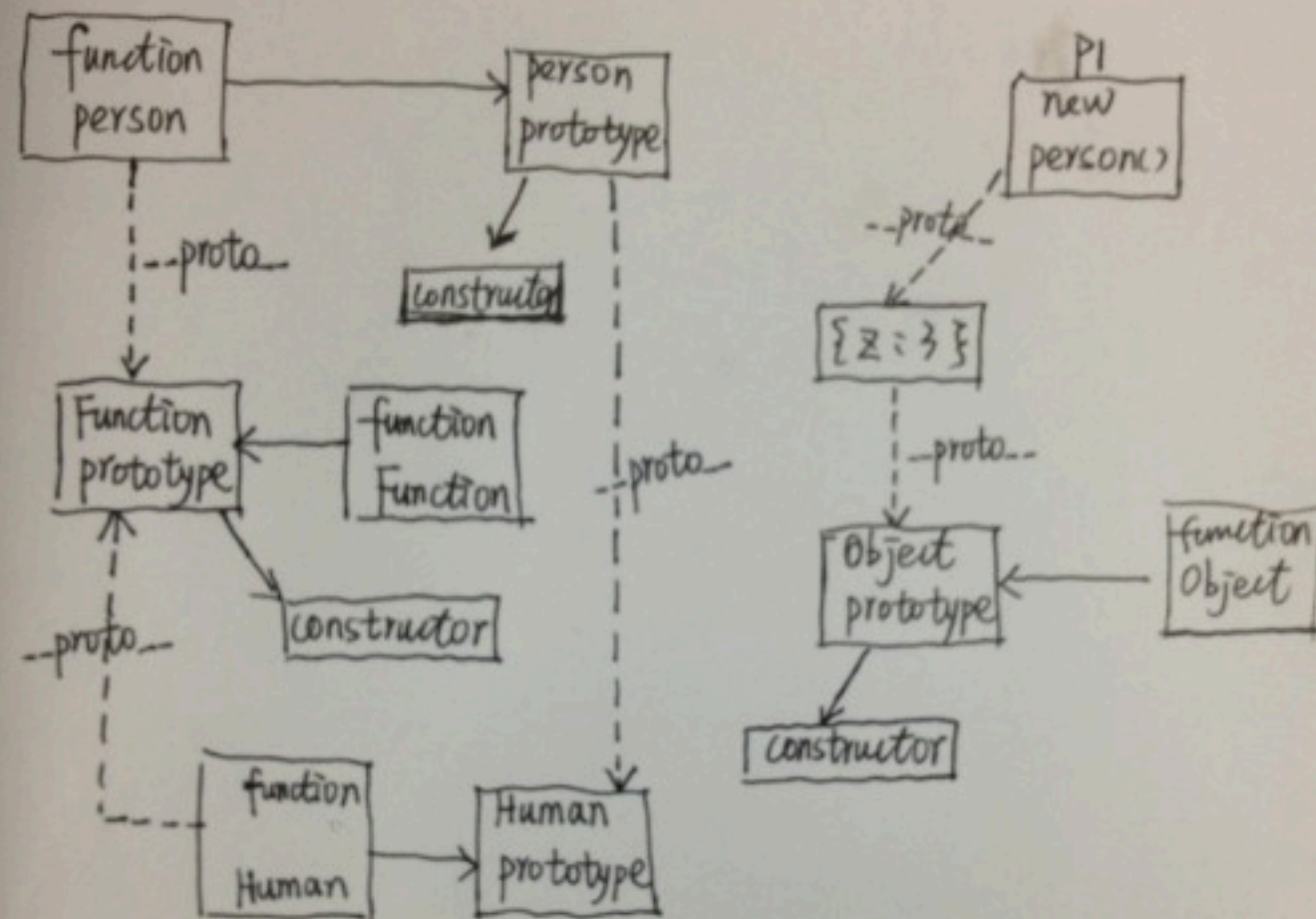
- eval
- number / string / boolean
- parseFloat / parseInt
- isFinite / isNaN
- escape / unescape
- encodeURIComponent / decodeURIComponent
- encodeURIComponent
decodeURIComponent

类、构造器、原型

- 类方法、实例方法，私有方法和原型方法
- 类属性、实例属性、私有属性和原型属性
- 内置类型可扩展
- 原型可以存放方法并共享属性
- 读写属性

原型及原型继承

```
1 function Person() {  
2     this.x = 1;  
3 }  
4 function Human() {  
5     this.y = 2;  
6 }  
7  
8 Person.prototype = {z:3};  
9  
10 var p1 = new Person();  
11  
12 console.log(p1.z);  
13  
14 Person.prototype = new Human();  
15  
16 console.log(p1.y);  
17 console.log(p1.z);  
18  
19 console.log(Person.constructor == Human.constructor);  
20 console.log(Person.constructor == Person);
```



So : `pi.constructor = object();`

`pi.y = undefined ;`

`person.constructor == Human.constructor == Function() ;`

读写属性

例：

```
var a = 10;  
console.log(a.toString());  
a.test = 100;  
console.log(a.test);
```

例：

```
1 console.log(1.toString());  
2 console.log(1..toString());  
3 console.log((1).toString());  
4 console.log(1['toString']());  
_
```

获取属性

0.[[Get]](P):

```
// 如果是自己的属性，就返回
if (0.hasOwnProperty(P)) {
  return 0.P;
}
```

```
// 否则，继续分析原型
var __proto = 0.[[Prototype]];
```

```
// 如果原型是null，返回undefined
if (__proto === null) {
  return undefined;
}
```

```
// 否则，对原型链递归调用[[Get]]，在各层的原型中查找属性
// 直到原型为null
return __proto.[[Get]](P)
```

写入属性

```
O.[[Put]](P, V):
```

```
// 如果不能给属性写值，就退出
```

```
if (!O.[[CanPut]](P)) {  
    return;  
}
```

```
// 如果对象没有自身的属性，就创建它
```

```
// 所有的attributes特性都是false
```

```
if (!O.hasOwnProperty(P)) {  
    createNewProperty(O, P, attributes: {  
        ReadOnly: false,  
        DontEnum: false,  
        DontDelete: false,  
        Internal: false  
    });  
}
```

```
// 如果属性存在就设置值，但不改变attributes特性
```

```
O.P = V
```

```
return;
```

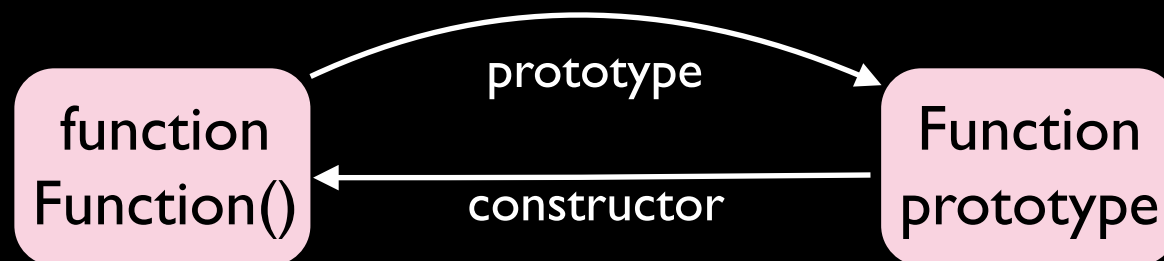
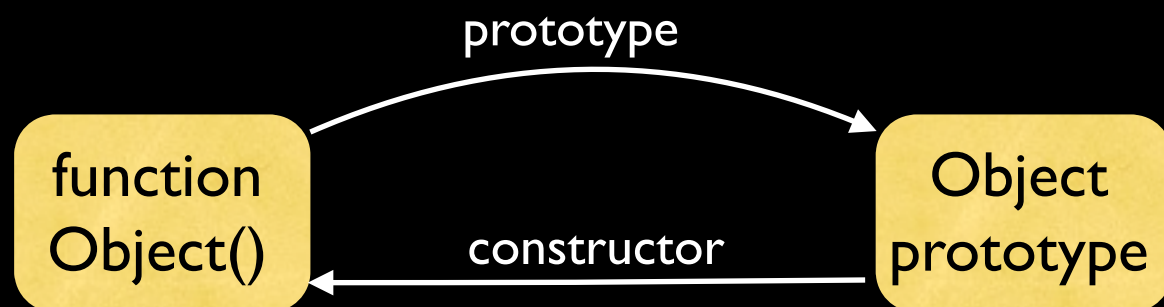
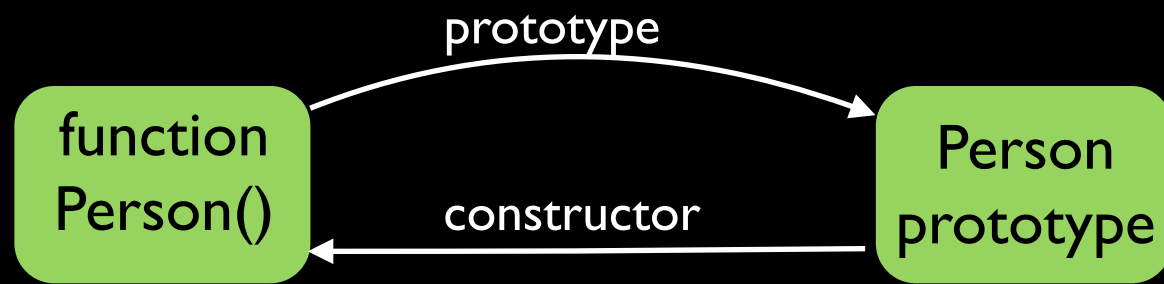

- 内部方法[[Get]]和[[Put]]是通过点符号或索引法来激活
- 属性访问器总是使用ToObject规范来对待“.”左边的值

构造函数与对象之间的关系

```
1 var Person = function() {}  
2 var person = new Person();  
3  
4 var obj1 = new Object();  
5 var obj2 = new String();  
6  
7 console.log(person instanceof Person);  
8 console.log(Person instanceof Function);  
9 console.log(Person instanceof Object);  
10  
11 console.log(obj1 instanceof Object);  
12 console.log(obj2 instanceof String);
```



对象模型



`new Person()`

`null`

`Number/String/.../RegExp`

`constructor`

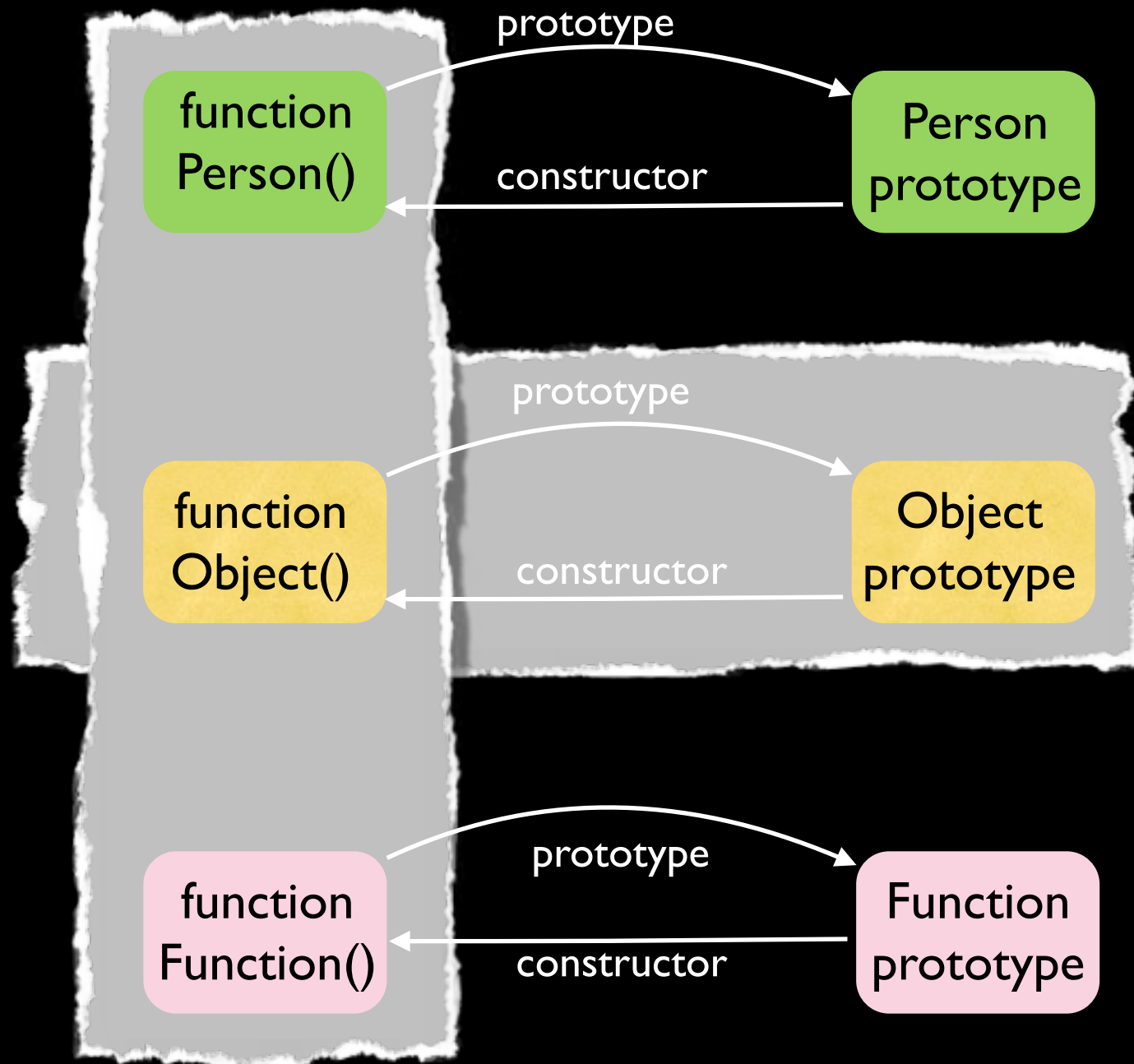
`prototype`

`constructor`



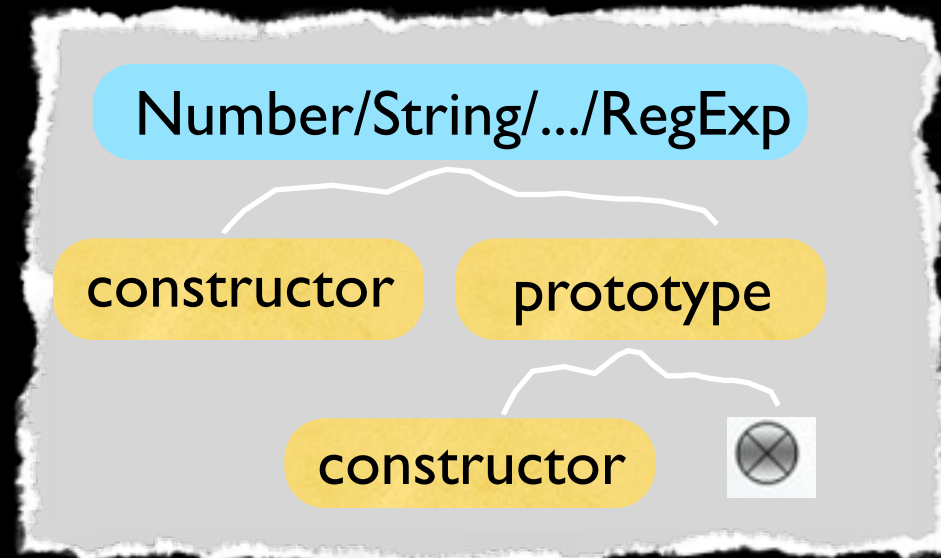
`obj1/obj2/obj3..`

对象模型



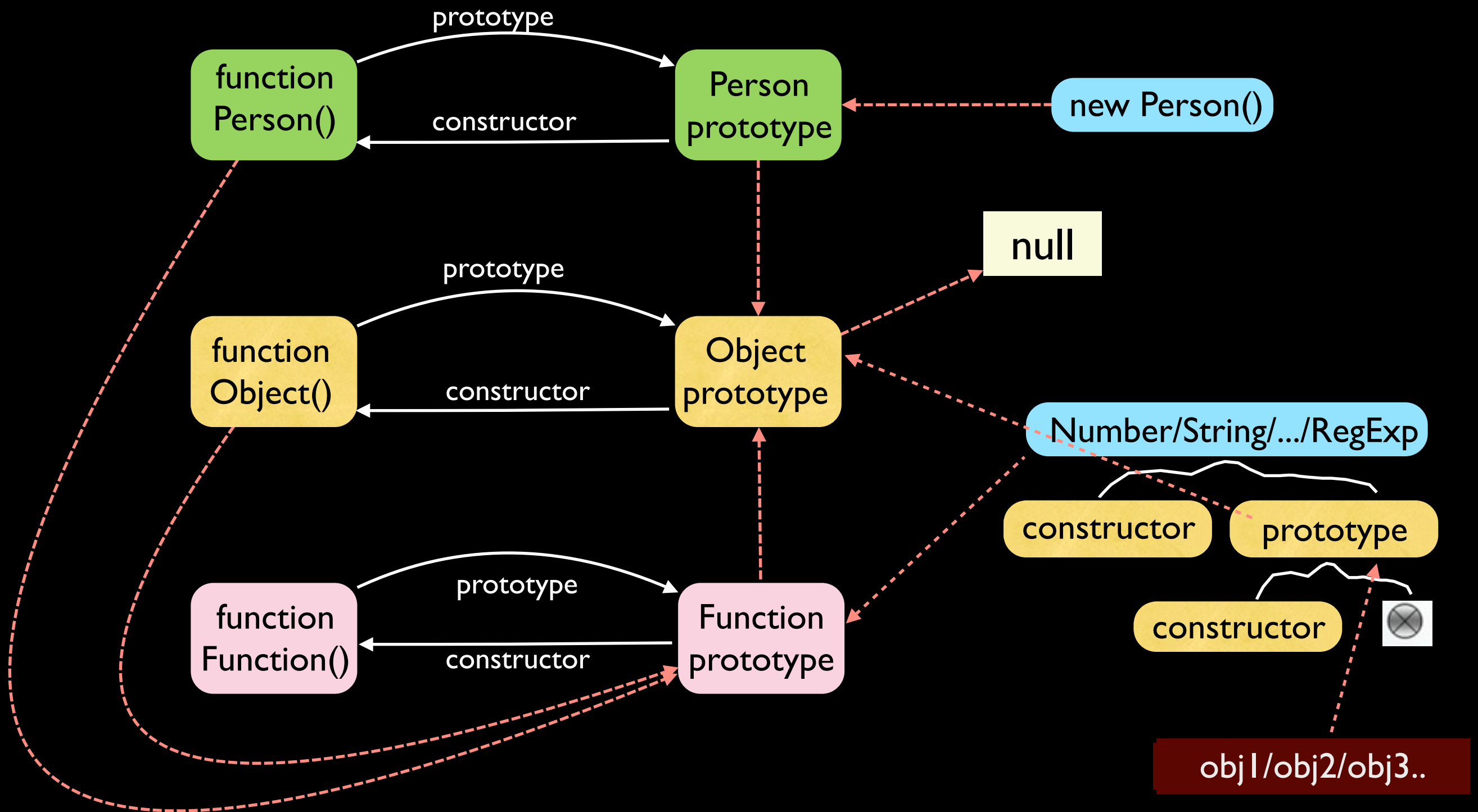
new Person()

null



obj1/obj2/obj3..

对象模型



模块

- 避免定义全局变量
- 向全局对象新增一条标记
 - 其文档应该清楚地描述该标记是什么
 - 标记的名字和载入模块的文件名应该有清楚的关系
- 使用闭包作为私有名字空间和作用域

正则

- `? * + {m,n}`
- 非贪婪与贪婪的重复 (`?? +? *?`)
- 选择、分组和引用 (`| () (?:) \n`)
- 指定匹配位置 (`?= ?!`)

- ✦ search `"javascript".search(/script/i);`
- ✦ replace `"javascript".replace(/(?:java)(scri)/g, '$1xx');`
- ✦ match `index、input`

```
console.log("javascript".replace(/(?:java)(scri)(pt)/g, function(x, y, z, m, n){  
    console.log(x, y, z, m, n);  
    return 'xiajiaojiao'  
}));
```


RegExp

- `exec`
- `test`
- `lastIndex`

Error对象

- `SyntaxError` 语法错误
- `EvalError` `eval`不可被赋值
- `RangeError` 常见于过大的数组定义
- `TypeError` 值的类型与要求不符
- `URLError` 常见于url decode中
- `ReferenceError` 读取一个不存在的值的时候

ES5 的新特性和新概念

- The Strict Mode
- 增强的对象模型 – `create`、`defineProperty`、`keys`、`preventExtensions`、`seal`、`freeze`
- 增强的数组模型 – `indexOf`、`lastIndexOf`、`every`、`some`
- JSON – `stringify`、`parse`
- `Function.prototype.bind`
- `Date.now()`

JavaScript Client

- BOM
- DOM
- EVENT
- XMLHttpRequest
- cookie和客户端持久性

thanks!