1. **构造函数**
2. 构造函数三要素

首字母大写；内部使用this对象指向即将要生成的实例对象；使用new运算符调用（不写new指向window）；充当类的角色

1. 实例化与内存

实例化：创建一个实例对象，分配内存；执行构造函数即对分配的内存空间进行初始化。

1. **普通函数的返回值**

没有return语句，默认返回undefined

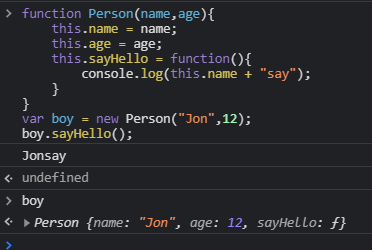
有return返回return

**构造函数的返回值**

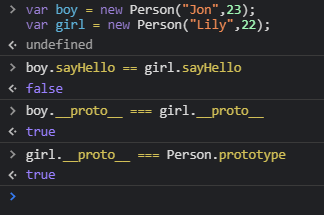
没有return，返回this实例化对象

有return+基本类型，返回this

有return+**引用**数据类型，返回return



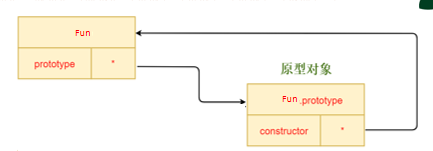
1. Protype/\_\_proto\_\_

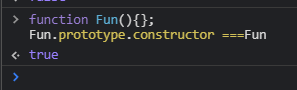


protype属性只有函数具有；构造函数prototype指向实例化对象的原型对象。

只要创建了一个函数Fun，JS引擎会为该函数创建一个prototype（指向实例化原型对象），并为该原型对象初始化一个属性constructor，用来引用该函数

作用：通过Prototype属性向实例化对象的原型对象添加属性和方法，实现属性和方法的共享；通过prototype拿到原型，然后分配。

1. 



Fun.prototype.constructor === Fun//函数对象的原型创造函数

**Constructor为什么定义在prototype上？**

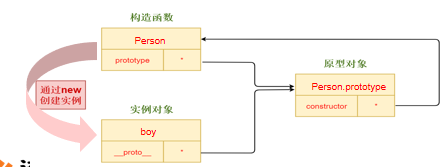
节约内存，只需有一个constructor就足够了。

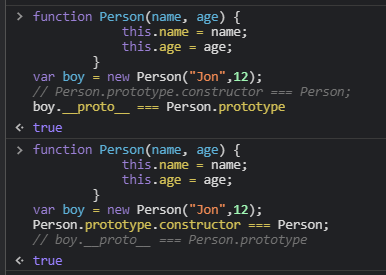
6、

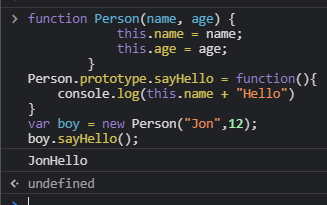
7、\_\_proto\_\_属性

每个实例对象都有一个属性\_\_proto\_\_指向它的构造函数的prototype属性即实例化对象的原型对象

实例对象可以访问这个原型对象中的属性，就像本身拥有一样



8、

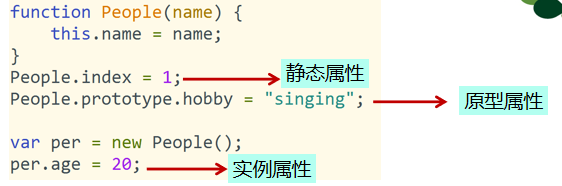
9、

在原型上添加的对象可以通过实例化对象访问。

原型对象：看作公共容器，将重复部分放入公共容器。

定义在原型对象上的属性会让所有实例对象都共享这个属性，但不会在每个实例化的内部重新定义这个方法，内存占用较小。

11、



实例对象可以访问本身对象上的实例属性，也可以访问原型对象上的原型对象，但不能访问构造函数上的静态属性

静态属性只能通过构造函数访问

1. new运算符：

创建一个空的简单JavaScript对象{}

该对象会被执行[prototype]连接

将该对象作为this的上下文，并执行构造函数中的代码

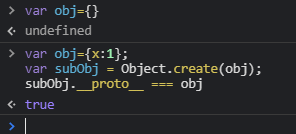
如果构造函数没有返回引用类型，返回this

**第九章 对象与对象属性**

1、对象是一种复合值：将很多值复合在一起

对象是若干无序属性的集合，可以直接通过属性名来访问对象的属性

函数作为某一个对象的属性时，称其为该对象的方法。

2、

3、**对象的重要性：**

一切数据都是通过变量保存；对象将相应的数据封装在一起统一管理

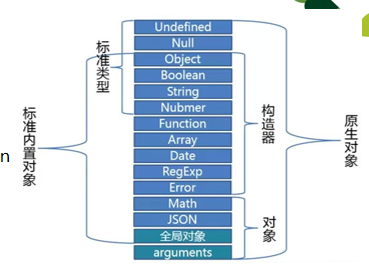
1. 属性的重要性：操作数据就是操作对象的属性。
2. 对象分类：

**内置对象**：非构造器对象（Math，JSON）或构造器对象（Object,Array,Function,Number,String,Boolean,Date）

**宿主对象：**由JavaScript解析器所嵌入的宿主环境定义（Window，document）

**自定义对象：**运行中用户的自定义JavaScript代码创建的对象；

通过3种方式创建（字面量，Object的create静态方法，构造函数实例化）

1. 

**Typeof Array/Function/Date //function**

**Typeof Math/JSON //Object**

1. 属性分类：

**数据属性**：对象中的普通属性键值映射

**访问器属性：**类似于读写属性的特殊方法

内置属性：只存在于ECMAScript语言规范中，不能直接访问，有可能存在间接访问方式。规范将内置属性的键置于[]中，[Scope][Prototype]

.\_\_proto\_\_可以访问[Prototype]

1. 数据属性：

Value（属性的值）:对应属性的值

Writable（可写特性）:确定属性是否可改写型

Configurable（可配置性）:确定属性是否能删除和其他特定是否可配置

Enumerable（可枚举特性）:属性是否可枚举

1. 属性描述符：
2. 用来查看对象属性的特性的对象，该对象包含4个属性，对应四个特性

封装了属性特性的对象，方便实现数据属性特性的设置和查询

1. Object.defineProperty(obj,prop,descriptor)

Obj:要定义属性的对象

Prop：要定义或修改属性的名称

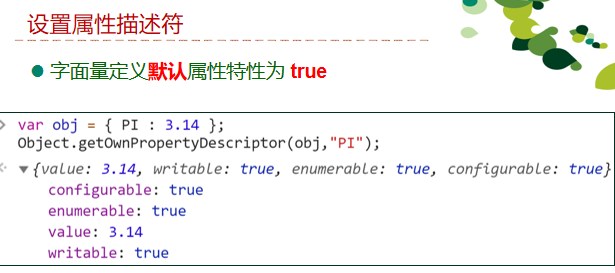
Descriptor:将被定义或修改的属性描述符

Value默认undefined

Enumerable、writable、configurable默认为false

1. 设置属性描述符：

Object.defineProperty(obj,**props**)



1. 查询属性描述符

Object.getOwnPropertyDescriptor(obj,prop)



1. 获取一个对象的所有自身属性的描述符

Object.getOwnPropertyDescript(obj)



1. 枚举性enmerable

一般来说，系统创建的属性不可枚举，而用户创建的属性可枚举

通常不应该给内置对象添加属性和方法，如果需要添加，应该设置属性不可枚举，避免破坏现有代码

枚举性影响操作：

for in循环(遍历包括原型属性)

Object.keys()（遍历不包括原型属性）

JSON.stringify()

1. 指定属性是否可枚举

Object.propertyIsEnumerable(prop)

1. configurable:可配置性

确定属性是否能删除和其他特性是否可配置

1. 定义属性

如果**属性不存在**，会创建一个新的属性，特性由描述符指定，如果未指定，则使用默认值

如果**属性已存在，**会更新描述符的属性特征，但在描述符中的特性没有对应的特性，特定不变

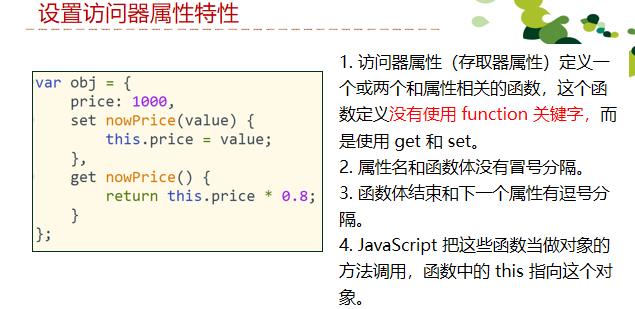
1. 访问器属性

Get：读取属性时，调用的函数，该函数计算读取的结果，默认undefined

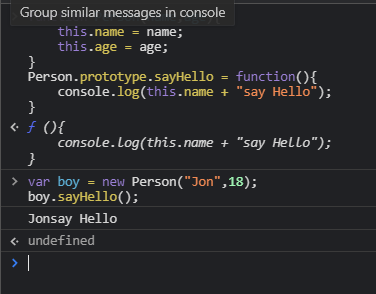
Set:设置属性值时调用的函数，该函数接受设置值的值作为参考，默认undefined

Configurable：确定属性是否能删除和其他特性是否可配置

Enumerable：属性是否可枚举

20、

**第十章 原型链及JS继承方式1**

1、

相当于在原型上添加一个函数，然后实例化出的新对象可以调用。

**2、属性访问机制**

**实例对象继承原型对象的属性。**

构造函数与原型对象方法中的this，指向调用该方法的对象。

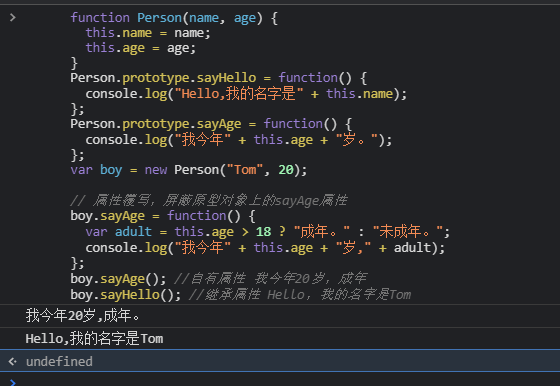
当通过对象访问属性时，JavaScript先从本对象查找，接着找原型，原型的原型，按原型链查找。

**3、属性屏蔽**

当本对象和原型对象具有相同的属性时，不会再去原型对象上访问。

**4、覆写**

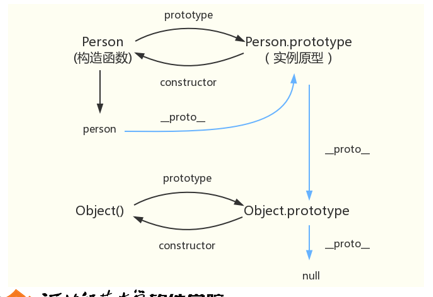
一个对象的属性可以覆写原型对象相同键的属性。此时，前者的属性最先被找到，所以隐藏原型对象的属性，使得原型对象的属性不能被访问。



**5、基于构造函数实现的原型继承-原型链**

**实例化出的对象.\_\_proto\_==> 指向原型，并确定原型**

**实例化对象.\_\_proto\_\_ === 构造函数.prototype**



1. 属性分类

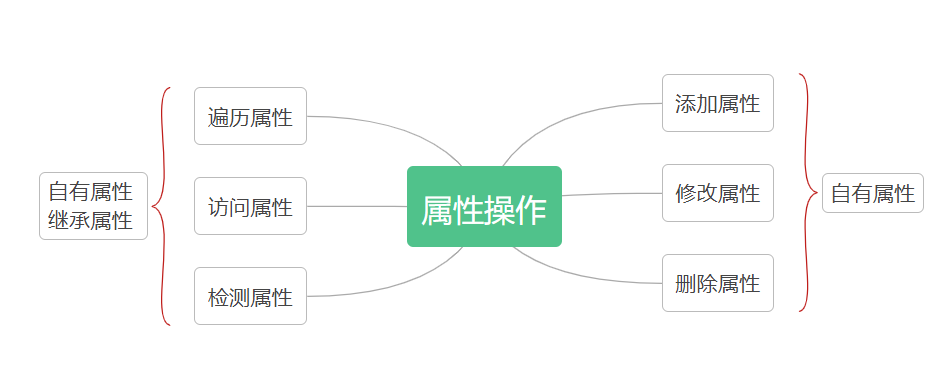
**自有属性：直接赋予给该对象的属性，不需在原型链上进行查找**

**继承属性：**

**从原型链上查找的属性**

**静态属性、直接属性对象**

1. 属性操作：



1. 检测属性

Obj.hasOwnProperty(prop)

返回布尔值，判断实例化出对象自身属性是否具有指定属性

Object.getOwnPropertyNames(obj)

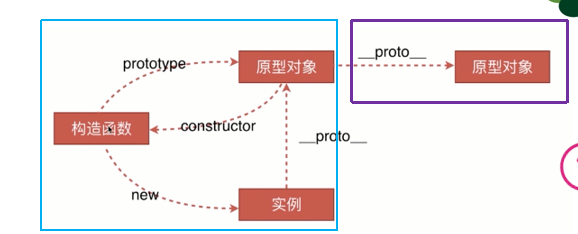
返回一个由指定对象的所有自身属性的属性名组成的数组

Prop in obj

返回值布尔类型，检查obj（包括原型链）是否包含具有指定名称的属性对象

1. 遍历属性

遍历一个对象所有可枚举属性键（包括自有和继承）

10、

11、JavaScript基于原型的语言

原型链：每个对象拥有一个原型对象，对象从原型继承方法和属性。

原型对象也可能拥有原型对象，并从中继承方法和属性，这种关系称为原型链

层层向上直到有一个对象的原型对象为null。

1. instance.\_\_proto\_\_ === constructor.prototype

所有对象都有\_\_proto\_\_属性，函数具有prototype属性，也具有\_\_proto\_\_属性

13、function Foo() {}

      var f1 = new Foo();

      var f2 = new Foo();

      f1.\_\_proto\_\_ === f2.\_\_proto\_\_; //true

      f1.\_\_proto\_\_ === Foo.prototype; //true

      Foo.prototype.constructor === Foo; //true

      f1.constructor === Foo; //true

      var o1 = new Object();

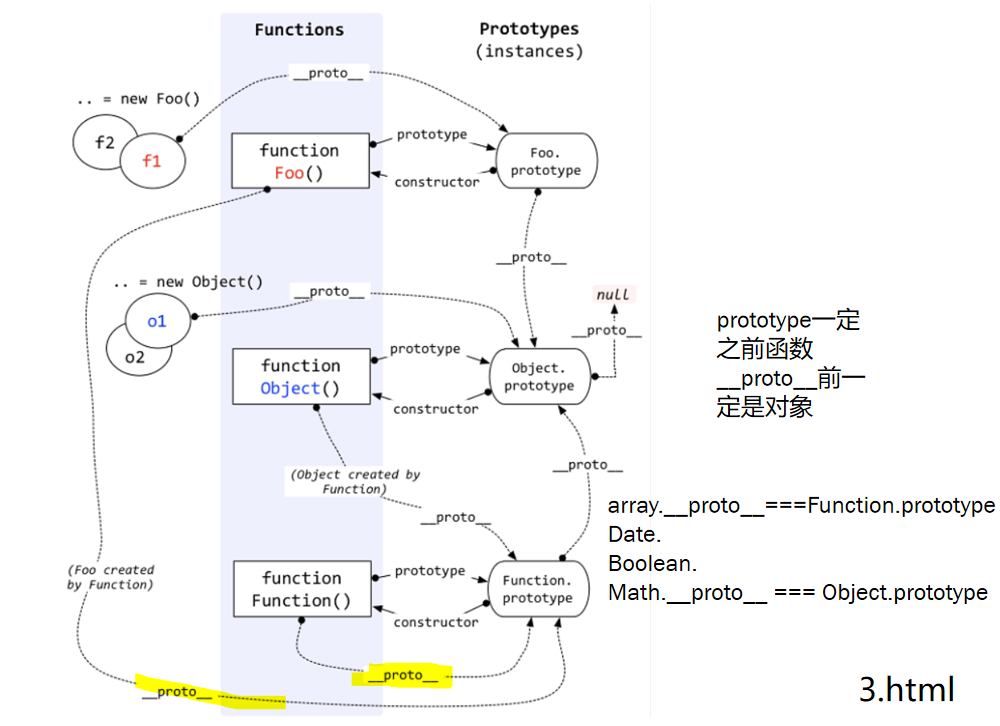
      var o2 = new Object();

      o1.\_\_proto\_\_ === o2.\_\_proto\_\_; //true

      o1.\_\_proto\_\_ === Object.prototype; //true

      Object.prototype.constructor === Object; //true

      o1.constructor === Object; //true

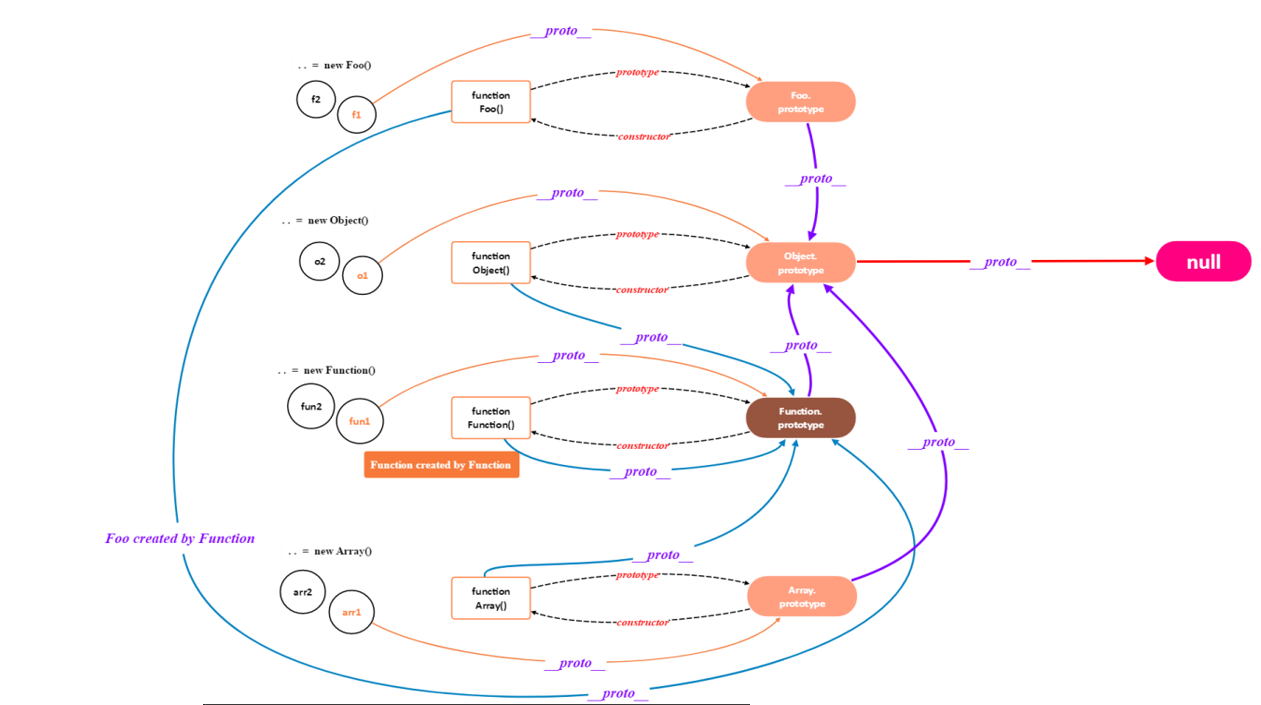


Function.\_\_proto\_\_ == Function.prototype(Function created by Function)  
Object.\_\_proto\_\_ == Function.prototype(Object created by Function)

Function.prototype.\_\_proto\_\_ == Object.prototype

Function.protype是函数起源，Object.prototype是对象起源

14、



15、

获取/设置/判断原型对象

Object.getPrototypeOf(obj)

返回指定对象的原型，如果没有继承属性，则返回null

Object.create(proto)

创建一个新对象，使用现有的对象来提供新创建的对象的\_\_proto\_\_（作为原型）

prototypeObj.isPrototypeOf(object)

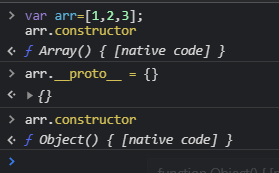
测试prototypeObj对象对否存在在object对象的原型链上

1. 数据类型检测

（1）typeof

（2）constructor对象的constructor属性用于返回创建该对象的函数（Array,Boolean,Date,Function,Number,Object）

在JavaScript中，通过构造函数创建实例化对象可以访问原型对象的constructor属性



1. obj instanceof constructor

返回布尔值，检测构造函数的prototype属性值是否出现在某个实例对象上。

1. toString

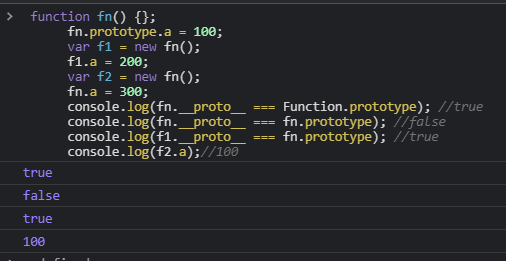
toString是定义在Object.prototype对象上的方法，返回一个表示该对象的字符串。

按照 JavaScript **属性访问规则**，所有对象都可以访问到 Object.prototype 上的的 toString 方法，而事实上，大部分的对象都**实现了自身的 toString** 方法，实现了**属性覆写**，这可能会导致 Object.prototype的 toString 被屏蔽。所以，需要**通过 call/apply 来借用 Object.prototype 上的 toString 方法。**

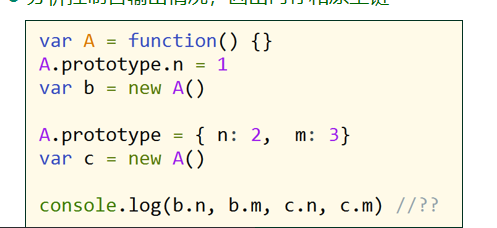
使用 call 来借用该方法会返回调用者的类型字符串，格式为 [object xxx]，xxx 是调用者的数据类型。

1. 总结：属性继承只发生在读属性时，写属性值时不会发生
2. 课后：如何准确判断一个变量是数组类型？

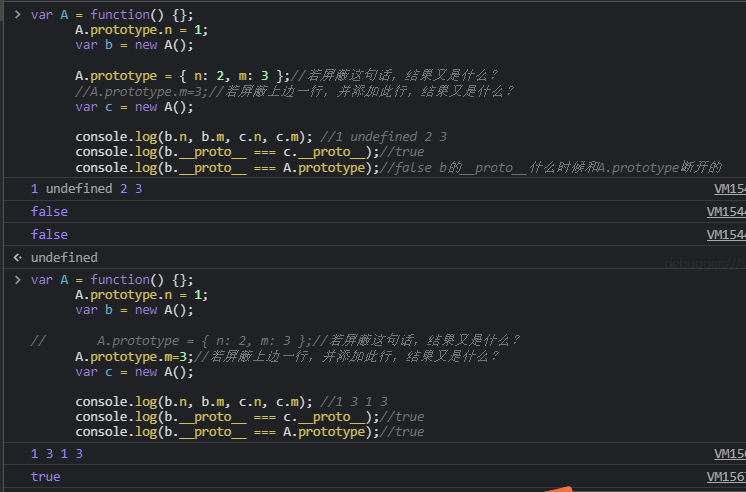
Instanceof



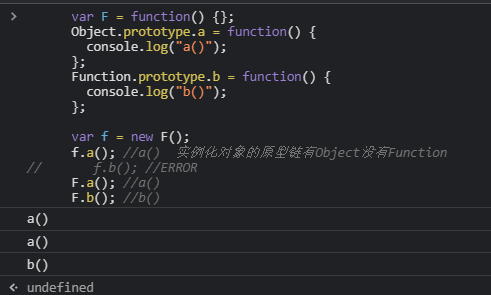
19、



20、



21、



22、



23、

**第十章 原型链及JS继承方式2**

**1、JS对象-对象原型继承**

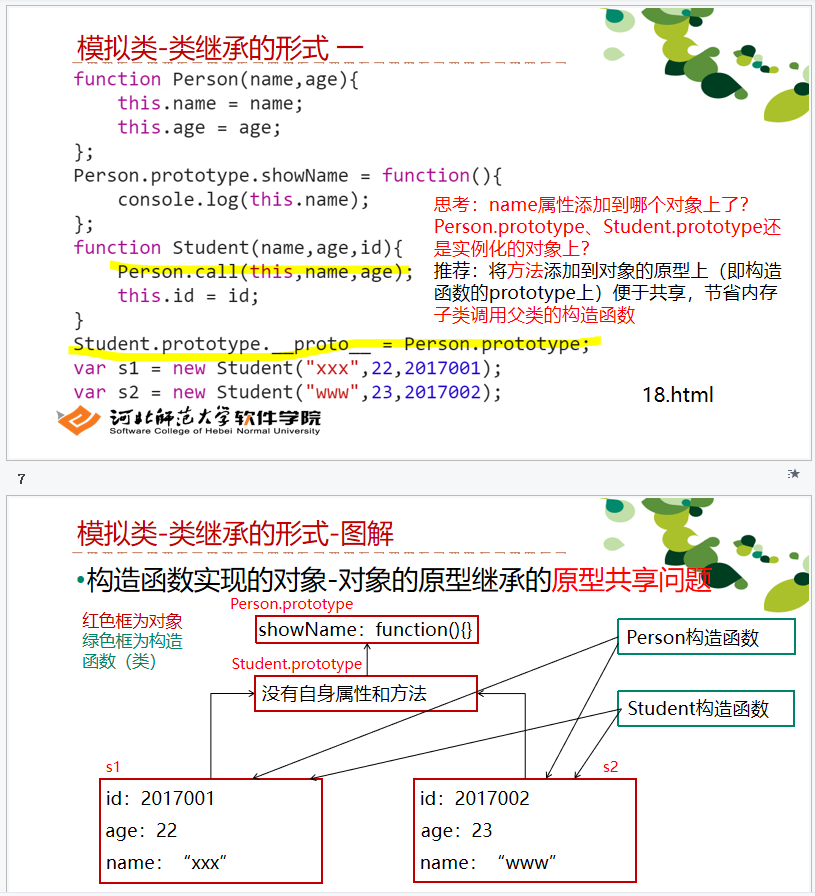
JavaScript的原型继承是对象-对象的继承

每个对象都有一个原型（可动态指定原型，最原始原型是null）

多个对象继承于一个原型时，存在原型共享



**2、通过构造函数模拟类-类的继承**



**3、JS继承补充部分**

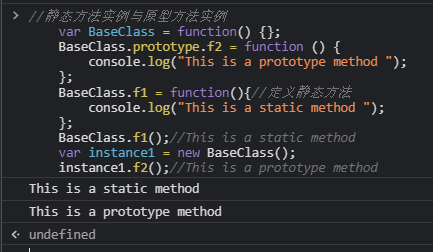
**静态方法与原型方法的区别：**

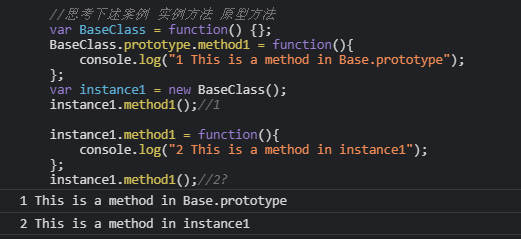
静态方法是**构造器函数对象的方法**，原型方法是实例化对象的原型方法。

**对象原型的constructor属性**

**对象实例从原型中继承了constructor**，所以通过constructor得到实例的构造函数，

确定对象的构造函数名、创建相似对象、constructor可用于指定构造函数





1. 对象的公有属性、私有属性



**第十章 JS继承扩展知识**

**原型继承（原型链继承）**

**构造继承（call继承）**

**\*\*\*组合继承（原型继承+构造继承）**

**原型式继承（类似于原型继承）**

**寄生式继承（原型式继承进阶）**

**\*\*\*寄生组合式继承（寄生式继承+构造继承）**

**原型继承：将子及的原型对象设置为父级的一个实例**

优点：实例是子级的实例，也是父级的实例。**父级**新增原型方法/属性，**子级都能访问。**

缺点：所有实例都会公用一个原型链，如果**通过一个实例修改原型**属性、那么所有实例访问的值都会被修改。

**构造继承：通过使用 call、apply 方法可以在子级构造函数上执行父级构造函数**

优点：直接继承父级构造函数中的属性和方法

缺点：构造函数的继承相当于将父级复制给子级，在子级中实现了所有父级的方法，影响子级的性能。另外无法继承原型链上的属性和方法

原型链继承能继承父级原型链上的属性，但是可能会存在篡改的问题；而构造函数继承不会存在篡改的问题，但是不能继承原型上面的属性

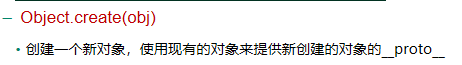
**组合继承：**构造继承 + 原型继承

优点：继承父级构造函数和原型上的属性和方法

缺点：调用了两次父级的构造函数，有些许损耗性能，并且子级的构造函数的属性会和原型上面的属性重合

**原型式继承（类似于原型继承）**

原型式继承并不需要定义一个构造函数，传入参数obj，生成一个继承 obj 对象的对象



**十一章 递归与深浅拷贝**

1. 递归：

一个方法重复调用自身

递归一定要有结束条件

1. 拷贝

**浅拷贝：拷贝基本类型的值；拷贝引用类型的引用**

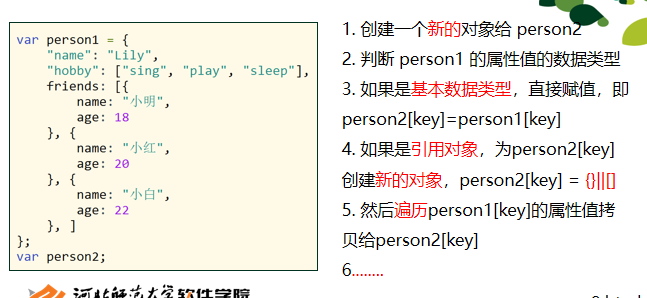
对象拷贝时，如果属性值是引用数据类型时，这时候我们传递的只是一个地址。因此子对象在访问该属性时，会根据地址回溯到父对象指向的堆内存中，即父子对象发生了关联，两者的属性值会指向**同一内存空间。**

浅拷贝新建一个对象，将源对象的属性一一复制

**深拷贝：浅拷贝+递归**

拷贝基本类型的值；判断类型

拷贝引用类型时，使用**递归**，把父对象所有属于引用类型的对象遍历父与子对象

3、

4、**浅拷贝**： 将原对象或原数组的引用直接赋给新对象的属性，新对象／数组的属性只是原对象属性的一个引用

**深拷贝**： 创建一个新的对象和数组，将原对象的各项属性的“值”拷贝过来，是“值”而不是“引用”

深拷贝和浅拷贝**最根本的区别**在于是**否是真正获取了一个对象的复制实体，而不是引用**

**深拷贝的缺点**

虽然深拷贝能够避免浅拷贝出现的问题，但是却会带来性能上的问题，如果一个对象非常复杂或数据庞大，性能消耗也会增加

1. 拷贝三问

**为什么要使用拷贝？**

我们希望在改变新的数组（对象）的时候，不改变原数组（对象）

**拷贝的要求程度**

是仅拷贝第一层级的对象属性或数组元素

还是递归拷贝所有层级的对象属性和数组元素

**检验深拷贝与浅拷贝**

改变一个新对象/数组中的属性/元素，看是否改变原对象/数组中的属性/元素

**第十二章 标准内置对象1（构造器）**

**1、Number 包装对象**

当访问 Number 基本数据类型属性或方法时创建临时包装对象，访问的都是对象中的属性或方法

访问对象属性时，首先访问自身属性，访问不到时，则会在原型链上寻找对应的属性和方法

原型方法

Number.prototype.toFixed()

Number.prototype.toPrecision()

Number.prototype.toString()

Number.prototype.toExponential()

静态属性

Number.MAX\_VALUE Number.MIN\_VALUE

Number.NaN Number.NEGATIVE\_INFINITY

Number.POSITIVE\_INFINITY

**2、Boolean 包装对象**

当访问 Boolean 基本数据类型属性或方法时创建临时包装对象，访问的都是对象中的属性或方法

访问对象属性时，首先访问自身属性，访问不到时，则会在原型链上寻找对应的属性和方法

**所有对象都是真值**

1. **String包装对象**

当访问 String 基本数据类型属性或方法时创建临时包装对象，访问的都是对象中的属性或方法

访问对象属性时，首先访问自身属性，访问不到时，则会在原型链上寻找对应的属性和方法

**静态方法**

String.fromCharCode(97,98,99);

**原型方法**

String.prototype.trim( );

String.prototype.concat(str1?,str2?);

String.prototype.toLowerCase( ); String.prototype.toLocaleLowerCase( );

String.prototype.toUpperCase( ); String.prototype.toLocaleUpperCase( );

String.prototype.indexOf(searchingString,position?);

String.prototype.lastIndexOf(searchingString,position?);

String.prototype.search(regexp);

String.prototype.match(regexp);

String.prototype.replace(regexp);

1. Function

函数定义：

通过函数声明的形式来定义

通过函数表达式的形式来定义

通过 Function 构造函数实例化的形式来定义

函数调用

作为函数直接调用

作为对象方法调用

作为构造函数调用

通过 call/apply 间接调用

**静态属性**

Function.name

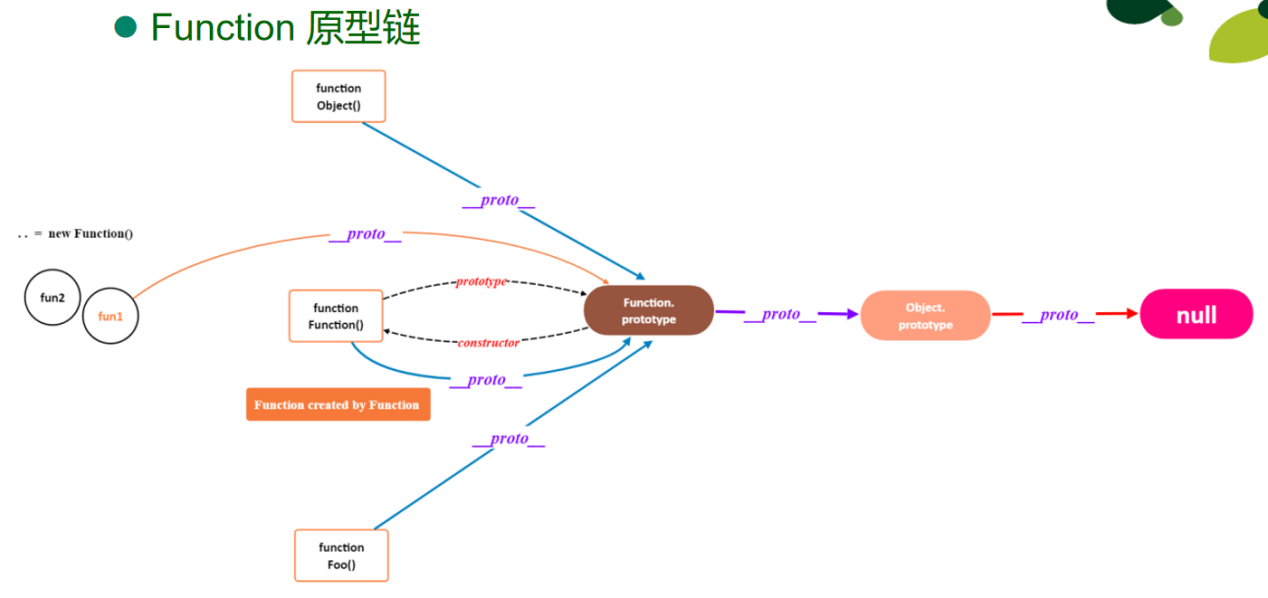
**原型方法**

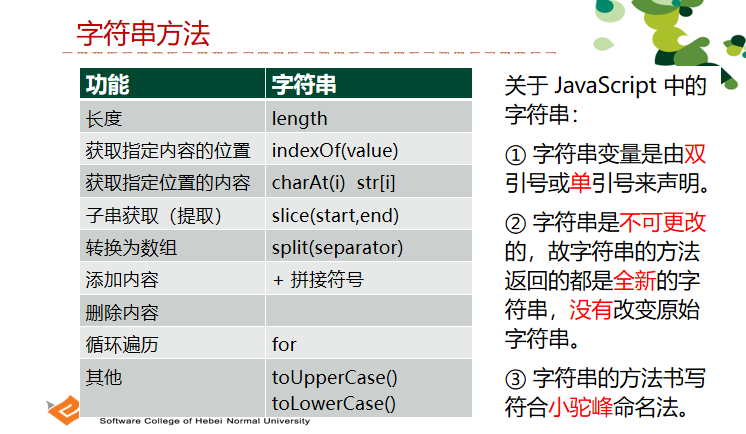
Function.prototype.apply()

Function.prototype.bind()

Function.prototype.call()

Function.prototype.toString()





5、Object

**静态方法**

Object.create()

Object.defineProperties()

Object.defineProperty()

Object.getOwnPropertyDescriptor()

Object.getOwnPropertyDescriptors()

Object.getOwnPropertyNames()

Object.getPrototypeOf()

Object.setPrototypeOf()

**原型方法**

Object.prototype.hasOwnProperty()

Object.prototype.isPrototypeOf()

Object.prototype.propertyIsEnumerable()

Object.prototype.toString()

Object.prototype.valueOf()

**6、Date**

Date 提供了解析、管理和展示时间的功能

Date 对象基于1970年1月1日（世界标准时间）起的**毫秒数**

以**常规函数调用**时（即不加 new 操作符）将会返回一个**字符串**，而不是一个日期对象，**通过new调用**，返回的是Date对象

Date 对象没有字面量格式



**时间戳(timestamp)**

时间戳是指格林尼治时间1970年01月01日00时00分00秒(北京时间1970年01月01日08时00分00秒)起至现在的**总毫秒数**

**时间转换成时间戳**

Date.now()

(new Date()).getTime()

**时间戳转换成时间**

new Date(timestamp)



**通过构造函数创建Date对象的4种形式**new Date(year,month,date?,hours?,minutes?,seconds?,milliseconds?)//注意起始索引   
new Date(dateTimeStr)//参数为字符串类型，注意格式，参见日期格式  
new Date(timeValue)//参数为数字类型，以毫秒为单位的时间戳  
new Date()//返回当前时间

**Date构造器方法（静态方法）**

Date.now()

Date.parse() //转成毫秒，从1970年1月1日 00:00:00开始计算

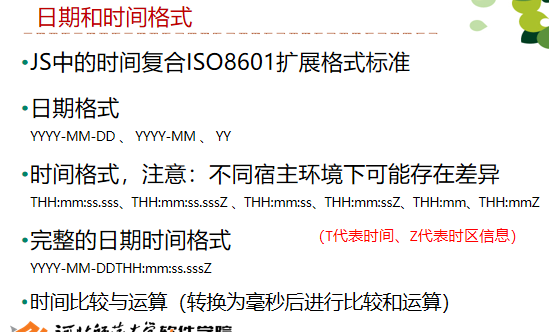
**原型方法**

Date.prototype.get<Unit>( )

Date.prototype.set<Unit>( )

Date.prototype.toLocalTimeString( )

Date.prototype.toString( )

Date.prototype.toLocalString( )   


**第十三章 标准内置对象**

1、Array

**静态方法( ES6 )**

Array.from(...) 、Array.isArray(...) 、Array.of(...)等

**原型方法（添加和删除元素—— 破坏性）**

Array.prototype.shift() Array.prototype.unshift(elem1?,elem2?,...)

Array.prototype.pop() Array.prototype.push(elem1?,elem2?,...)

Array.prototype.splice(start,deleteCount?,elem1?,elem2?)  
 **原型方法（排序和颠倒元素顺序 —— 破坏性）**

Array.prototype.reverse()

Array.prototype.sort(compareFunction？)

**原型方法（合并、切分和连接 —— 非破坏性）**

Array.prototype.concat(arr1?,arr2?,...)

Array.prototype.slice(begin?,end?)

Array.prototype.join(separator?) //注意返回的类型

**原型方法（值的查找 —— 非破坏性）**

Array.prototype.indexOf(searchValue,startIndex?)

Array.prototype.lastIndexOf(searchElement,startIndex?)

**原型方法（迭代 —— 非破坏性）**

Array.prototype.forEach(callback,thisValue?) //遍历

Array.prototype.every(callback,thisValue?)//是否为偶数

Array.prototype.some(callback,thisValue?) //是否为

**原型方法（迭代-转换方法 —— 非破坏性）**

Array.prototype.map(callback,thisValue?)

Array.prototype.filter(callback,thisValue?)

**原型方法（迭代-归约方法 —— 非破坏性）**

Array.prototype.reduce(element,initialValue?)

Array.prototype.reduceRight(callback,initialValue?)

**forEach 遍历所有元素**

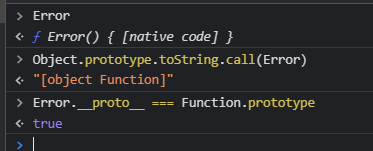
**every 判断所有元素是否都符合条件**

**some 判断是否有至少一个元素符合条件**

**map 对元素重新组装，生成新的数组**

**filter 过滤符合条件的元素**

2、ERROR



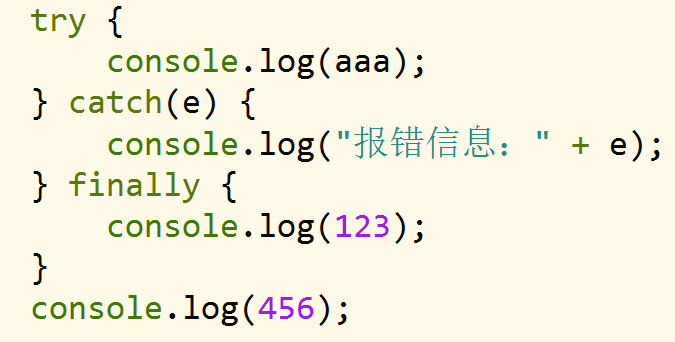
异常：如果执行语句中有一句产生错误，后面的语句不会继续执行

**异常捕获(try catch finally)**

try catch finally

try 是必选的，catch 和 finally 两者至少要有其一

如果没有 catch 块，程序将会终止



**异常捕获**

catch 块指定一个标识符（在上面的示例中为 e），该标识符保存抛出异常信息

标识符仅在 catch 块执行时存在；catch 块执行完成后，标识符不再可用

**对所有内置错误，catch 内的错误对象主要有两个属性：**

name ：错误名称

message ：错误信息的文本描述

stack**(非标准)**：当前调用栈：关于导致错误的嵌套调用序列，用于调试目的

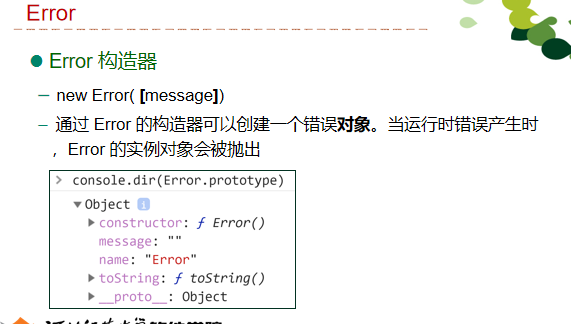
**异常抛出**

throw <expression >

抛出一个用户自定义的异常。

expression 指定了异常的内容，可以为任何的 JavaScript 值





**ECMAScript 定义了七种类型的错误。**

Error

ReferenceError: 引用错误

TypeError: 类型错误

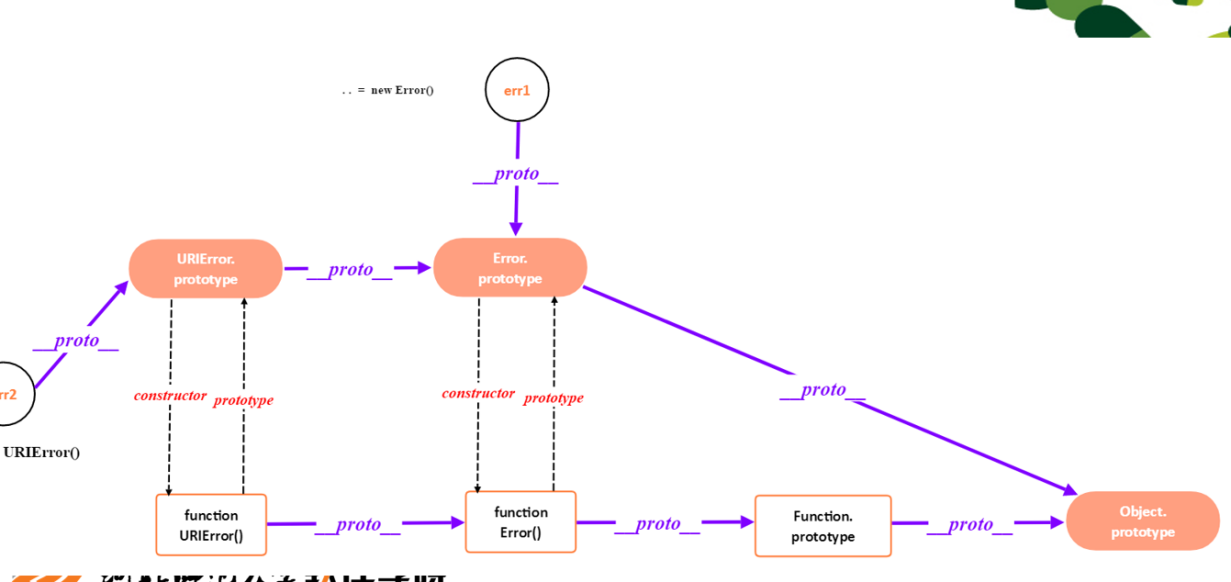
RangeError: 范围错误

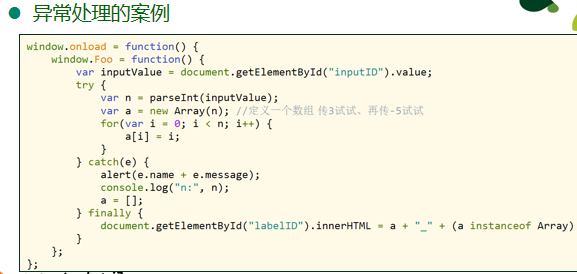
SyntaxError: 语法错误

EvalError: Eval错误

URIError: URI错误

**Error 是基类型，其他错误类型都继承自该类型。**





**第十四章正则**

1、JavaScript中的RegExp对象

正则：某种特定规则

字符串匹配工具

2、表达式是对字符串操作的一种逻辑公式；用事先定义好的一些特定字符、及这些特定字符的组合组成一个规则字符串，这个规则字符串用来对字符串进行匹配，然后做相应的操作，如过滤、替换、判定等。

3、正则表达式是匹配模式，包括**字符匹配**和**位置匹配**

**4、正则表达式用途：**

**批量提取/替换有规律的字符串**

**验证客户端的输入数据**

**各类办公软件中使用**

**各类开发语言中使用（C#/Java/Perl/PHP/Python）**

**网络爬虫（抓取机器人）的开发**

5、正则表达式创建：

字面量方式：

var box = /box/;

Var box = /box/ig;

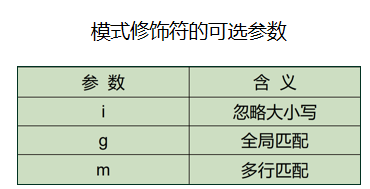
New运算符：

Var box = new RegExp(‘box’);

Var box = new RegExp(/box/);

Var box = new RegExp(‘box’,’ig’);



6、

7、构造函数的匹配模式可以使用变量，但是字面量不能够使用变量替代。

8、 RegExp 原型方法

test()//检索字符串中的指定值，返回true或false

exec()//检索字符串中的指定值，返回找到的值并确定位置

String 原型方法

replace()// strObject.replace(regexp,'replaceString'); 对字符串中特定字串进行替换，返回替换后的结果

match()//一个或多个子串、正则表达式的匹配，放入数组

search()//返回**第一次**出现匹配正则表达式的**下标**，没有回-1

split()//用指定字符串后数组，对源字符串进行拆分，拆得子串数组。

9、

10、

    //转义字符 双（单）引号里面的第一个 \ 识别成转义字符  解析之后不会在浏览器输出

    //字符串

    var str1 = "How are you?\nI'm fine!";

    var str2 = "How are you?\\nI'm fine!";

    var str3 = "How are you?\\\\nI'm fine!";//输出\\

    console.log(str1);

    //How are you?

    //I'm fine!

    console.log(str2);

    //How are you?\nI'm fine!

    console.log(str3);

    //How are you?\\nI'm fine!

11、

    //匹配\

    var pattern1 = /\\/;

    var pattern2 = new RegExp('\\\\'); //先解析为字符串 再解析为正则

    console.log(pattern1);//   /\\/

    //就是pattern1

    console.log(pattern2);//   /\\/

    //第一个\带走一个\，剩下直接写下来

12、

    //正则表达式匹配一个不确定的子串

    var a = "123";

    // var a = prompt();

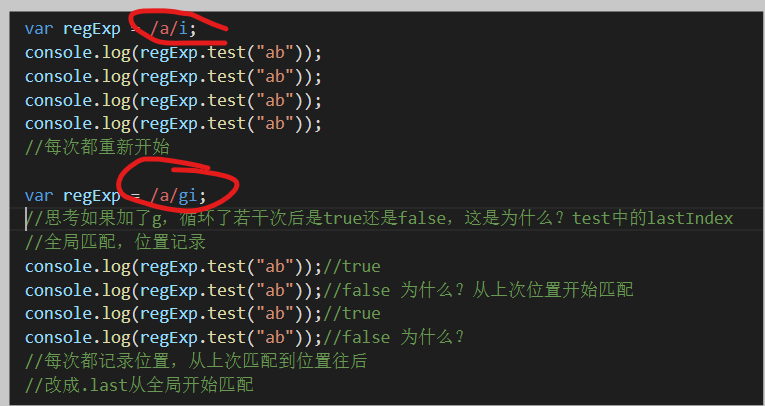
    var pattern = /a/; //在定义的时候就确定了

    var pattern1 = new RegExp(a); //定义时可以不确定，动态改变变量a的值

    console.log(pattern);//  /a/

    console.log(pattern1);//   /123/

13、



14、

\b是边界

\B不是边界

    //案例：结合定位匹配

    console.log(/oo/.test("moon"));//true

    console.log(/oo\b/.test("moon"));//false

    console.log(/oon\b/.test("moon"));//true

    console.log(/\boo/.test("moon"));//false

    console.log(/oo\B/.test("moon"));//true

    console.log(/oon\B/.test("moon"));//false

    console.log(/oo\B/.test("moon"));//true

    console.log(/\Boo\B/.test("moon"));//true

15、

字符串、正则/i replace只替换第一个匹配串

/ig全部替换

 //替换——返回替换后的结果

    //字符串替换

    var str = "This is a box This is a box!";

    console.log(str);//This is a box This is a box!

    console.log(str.replace("This", "1111"));//1111 is a box This is a box!

 //正则表达式替换

    var pattern = /This/;

    // var pattern = /(?!^)This/g;//只匹配后边的This

    var str = "This is a box This is a box!";

    console.log(str);//This is a box This is a box!

    console.log(str.replace(pattern, "1111"));//"1111 is a box 1111 is a box!

16、

   var regExp = /a\*b/gi; //注意\*和.的区别

var matchResult = "xxAbcaaBbxyz".match(regExp);

   console.log(matchResult);//**["Ab","aaB","b"]**

//必须要有b,前面可重复，可没有

    var regExp = /a.b/gi;//注意\*和.的区别 ，参见在线分析工具 https://regexper.com或https://jex.im/regulex

    var matchResult = "xxAbcaaBbxyz".match(regExp);

    console.log(matchResult);//["aaB"] a加任意字符加B

  </script>

17、

    //横杠 - 拆分

    var pattern = /-/;

    var str = "This-is-a-box-This-is-a-box!";

    console.log(str);

    console.log(str.split(pattern));//["This","is","a","box","This","is","a","box!"]

    //指定数组返回个数

    // var pattern = /-/;

    var pattern = /-\d\*/;////-没有数字 或 -加若干数字 进行分割

    var str = "This-is-3a-box-This-4is-a-6box!";

    console.log(str);

    console.log(str.split(pattern));//["This","is","a","box","This","is","a","box!"]

    // console.log(str.split(pattern, 3)); //["This","is","a"] 前三个

18、

19、锚字符用于查找某一个位置

位置是指字符串每行第一个字符的左边、最后一个字符的右边以及相邻字符的中间

位置也可以理解为空字符

20、

^ 从字符开头开始匹配，即从左向右匹配（https协议检验）

$ 从字符串**结尾开**始匹配（验证文件是否有css）

边界 \b

\w是字符组[0-9a-zA-Z\_]的简写

\W排除[0-9a-zA-Z\_]的

^$位置也是边界

21、零宽断言

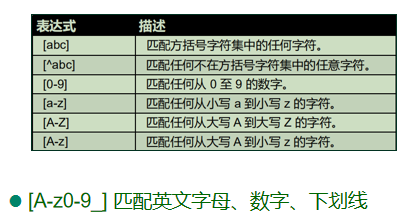
(?=exp)零宽断言正预测先行断言

断言此位置的后面能匹配表达式exp

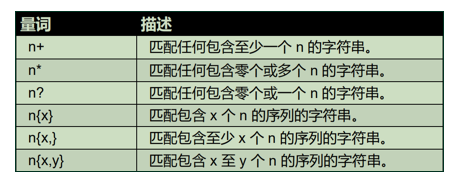
(?!exp)零宽负预测先行断言

断言此位置的后面**不**能匹配表达式exp

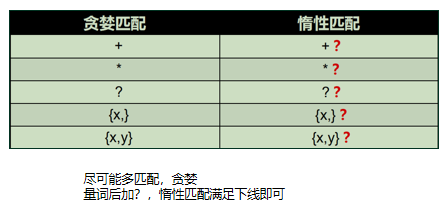
1. **方括号用于查找某个范围内的单个字符**



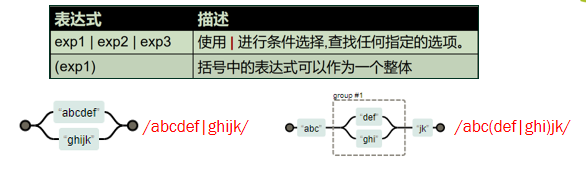
1. 量词是定义字符出现频次的字符



1. 贪婪和惰性



1. 选择和分组



1. 正则静态属性

RegExp.$1-$9

RegExp.input ($\_)

RegExp.lastMatch ($&)

RegExp.lastParen ($+)

RegExp.leftContext ($`)

patten.flags

patten.global

patten.ignoreCase

patten.multiline

patten.lastIndex

patten.source