1. 模板字符串替换

String.prototype.render=function(context){

return this.replace(/{{(.\*?)}}/g,(match,key)=>context[key.trim()]);

}

var str="{{name}}很厉害";

str=str.render({name:'js'});

console.log(str);//js很厉害

1. 静态作用域和动态作用域

var x=10;

function foo() {

var y=x+5;

console.log(y);

}

function bar(){

var x=2;

return foo();

}

function main(){

foo();//static scope:15 dynamic scope:15

bar();//static scope:15 dynamic:7

return 0;

}

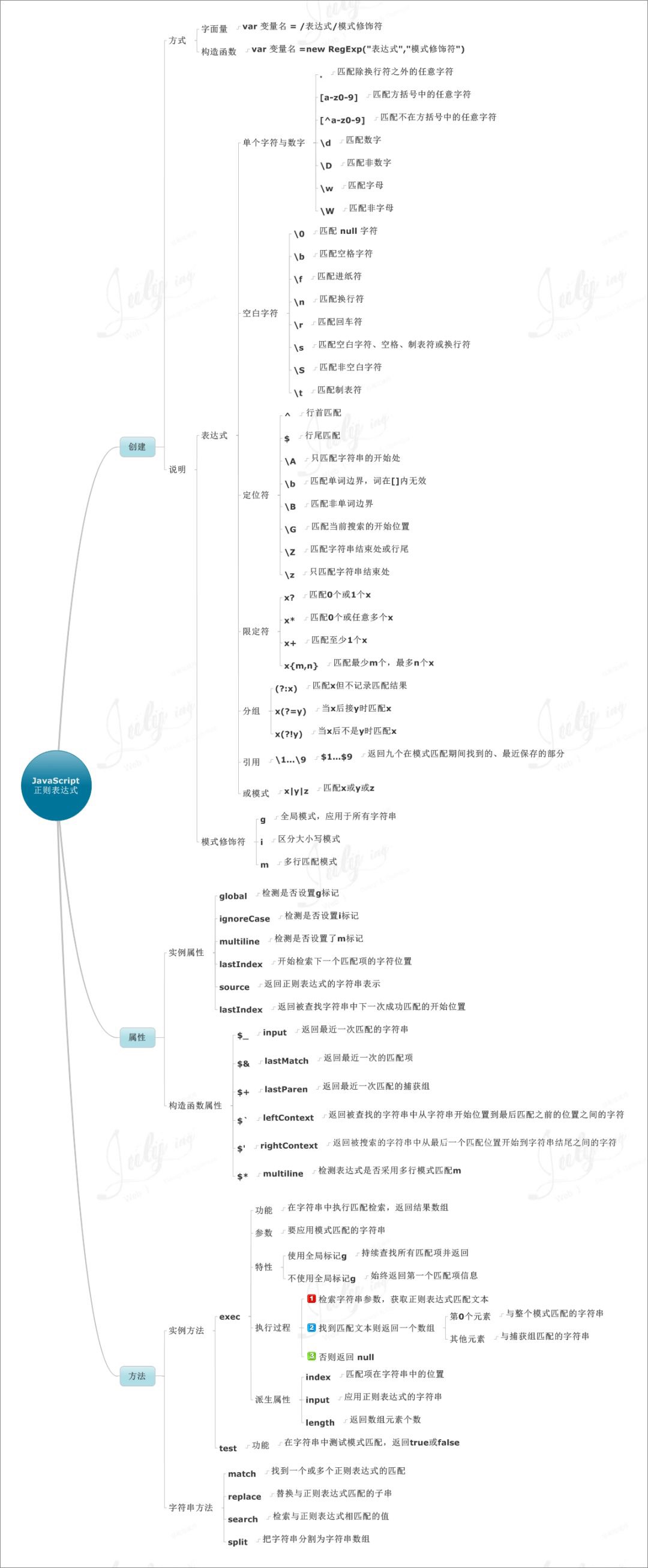
main();

//在静态作用域中，bar 的返回值是基于函数foo 创建时x的值

//在动态作用域中，x的值取决于在当前作用域中的实际定义

js采用词法作用域，即静态作用域

1. 闭包
   1. 在任意时间点，只能有一个执行上下文
   2. 闭包的一个重要用途是保留外部作用域对一个变量的私有引用
2. 正则



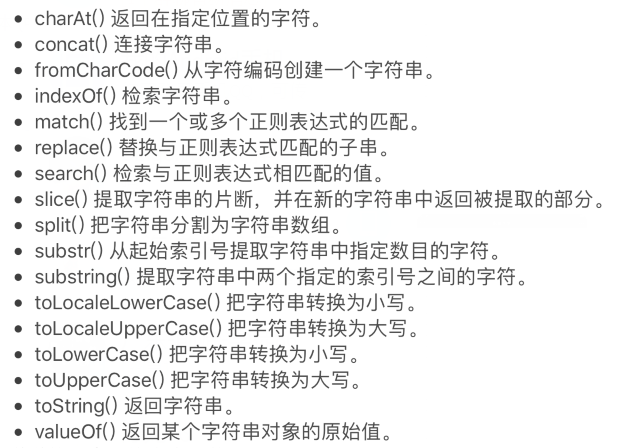
1. a标签的点击事件里，this指向window

this指调用函数的那个对象

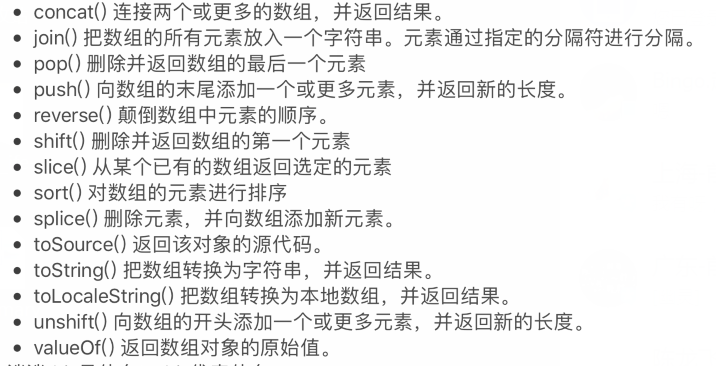
1. 改变this指向的方法

call apply bind

1. 绑定事件方法
   1. 嵌入dom
   2. 直接绑定 onclick
   3. 事件监听 addEventListener/attachEvent
2. 字符串常用函数



1. 数组常用函数



1. 对象-封装

构造函数模式

function Cat(name,color){

this.name=name;

this.color=color;

}

var cat1=new Cat("大毛","黄色");

var cat2=new Cat("二毛","黑色");

alert(cat1.name);//大毛

console.log(cat1.constructor==Cat);//true

console.log(cat2.constructor==Cat);//true

console.log(cat1 instanceof Cat);//true

缺点是浪费内存，如果有一样的内容，每生成一个实例都必须重复

Prototype模式

每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。

function Cat(name,color){

this.name=name;

this.color=color;

}

Cat.prototype.type="猫科动物";

var cat1=new Cat("大毛","黄色");

var cat2=new Cat("二毛","黑色");

console.log(cat1.type);//猫科动物

console.log(cat1.type==cat2.type);//true

console.log(Cat.prototype.isPrototypeOf(cat1));//true

console.log(cat1.hasOwnPrototype("name"));//true

console.log(cat1.hasOwnPrototype("type"));//false

console.log("name" in cat1);//true

console.log("type" in cat1);//true

1. 对象-继承

构造函数模式

function Animal(){

this.species="动物";

}

function Cat(name,color){

Animal.apply(this,arguments);

this.name=name;

this.color=color;

}

var cat1=new Cat("大毛","黄色");

console.log(cat1.species);//动物

prototype模式 最常用

function Animal(){

this.species="动物";

}

function Cat(name,color){

this.name=name;

this.color=color;

}

Cat.prototype=new Animal();

Cat.prototype.constructor=Cat;

var cat1=new Cat("大毛","黄色");

console.log(cat1.species);//动物

Cat.prototype=new Animal();

将Cat的prototype对象指向一个Animal的实例。

任何一个prototype对象都有一个constructor属性，执行它的构造函数，如果没有这一行，Cat.prototype.constructor是指向Cat的，加了之后指向Animal，所以要手动纠正

Cat.prototype.constructor=Cat;

直接继承prototype

function Animal(){}

Animal.prototype.species="动物";

function Cat(name,color){

this.name=name;

this.color=color;

}

Cat.prototype=Animal.prototype;

Cat.prototype.constructor=Cat;

var cat1=new Cat("大毛","黄色");

console.log(cat1.species);//动物

缺点：Cat.prototype的修改会影响Animal.prototype

利用空对象作中介

function extend(Child,Parent){

var F=function(){};

F.prototype=Parent.prototype;

Child.prototype=new F();

Child.prototype.constructor=Child;

Child.uber=Parent.prototype;

}

function Animal(){}

Animal.prototype.species="动物";

function Cat(name,color){

this.name=name;

this.color=color;

}

extend(Cat,Animal);

var cat1=new Cat("大毛","黄色");

console.log(cat1.species);//动物

拷贝继承

function Animal(){}

Animal.prototype.species="动物";

function Cat(name,color){

this.name=name;

this.color=color;

}

function extend2(Child,Parent){

var p=Parent.prototype;

var c=Child.prototype;

for(var i in p){

c[i]=p[i];

}

c.uber=p;

}

extend2(Cat,Animal);

var cat1=new Cat("大毛","黄色");

console.log(cat1.species);//动物

1. value

<input type="button" value="123" onclick="alert(value)"/>

input的value属性，默认是””，一直存在，不会沿作用域链向上找

<form action="#">

<input type="text" name="username" value="123" />

<input type="button" value="btn" onclick="alert(username.value)" />

</form>

表单元素的作用域链：this->this.form->document