## 一．Http协议

## 二．Javascript中延长作用域链

try-catch语句的catch块

With语句

## 三．Js实现富文本编辑器

编辑部分是一个div，contenteditable=”true” 使得div成为可编辑状态。

按钮功能的实现：

核心函数：

document.execCommand(aCommandName, aShowDefaultUI, aValueArgument);

三个参数：

参数1：命令

参数2：是否展示用户界面（true/false）

参数3： 一些命令需要一些额外的参数值，默认为null，然后通过onclick赋予相应的按·钮， 实现相应按钮的功能。

Selection对象和range对象：

## 四．Require.js模块化开发

1. **require.js的作用：**
2. 实现js文件的异步加载，避免网页失去响应；

(2) 管理模块之间的依赖性，便于 代码的编写和维护。

1. **使用：**

<script src="js/require.js" defer="defer" async="true" data-main="js/main"></script>

引入require.js文件; 其中async属性表明这个文件需要异步加载，避免网页失去响应，IE不支持这个属性，所以用defer属性代替; data-main: 指定网页程序的主模块 -> [ 整个网页的入口代码 ]。

**3. 主模块的写法：**

1. **不依赖任何模块的话直接写js代码。**
2. **主模块依赖于其他模块，使用AMD规范定义的require()函数。**

require()函数接受两个参数，第一个参数是数组，表示所依赖的模块；第二个参数是一个回调 函数，当前面指定的模块都加载成功后他将被调用。加载的模块会以参数的形式传入该函数， 从而在回调函数内部就可以使用这些模块。

require(['moduleA', 'moduleB', 'moduleC'], function(moduleA, moduleB, moduleC) {});

require()异步加载moduleA, moduleB, moduleC, 浏览器不会失去响应；它指定的回调函数 只有前面的模块都加载成功后，才会运行，解决了依赖性的问题。

例：

require(['jquery'], function($) {

console.log($("#context").html());

});

**4. 模块的加载：**

使用require.config()对模块的加载行为进行自定义。Require.config()就写在主模块的头部， 参数是一个对象。这个对象的path属性指定各个模块的加载路径。

(1) jquery.min.js与main.js在同一目录下:

require.config({

paths: {

"jquery": "jquery.min",

}

});

(2) 加载模块与main.js不在同一目录下:

1) require.config({

paths: {

"jquery": "lib/jquery.min",

}

});

2) require.config({

**baseUrl: "js/lib",**

paths: {

"jquery": "jquery.min",

}

});

(3) 如果某个模块在另一台主机上，也可以直接指定他的网址

require.config({

paths: {

jquery: "https://ajax.googleapis.com/jquery.min"

}

});

**5. AMD模块的写法：**

1. 定义一个模块:

define(*function*() {

*var* add = *function*(*x*, *y*) {

return x+y;

};

return {

add: add

}

});

1. 加载该模块:

require(['jquery', 'math'], *function*(*$*, *math*) {

*console*.log($("#context").html());

alert(*math*.add(1,2));

});

如果这个模块还依赖其他模块，那么define()函数的第一个参数必须是一个数组，指明该模块 的依赖性。如下，当require.js加载这个模块的时候就会先加载myLib.js模块。

define(['myLib'], *function*(*myLib*) {

*var* add = *function*(*x*, *y*) {

return x+y;

};

*var* word = *function*() {

return myLib.*text*;

}

return {

add: add,

word: word

}

});

**6. 加载非规范的模块：**

require.config()接受一个配置对象，这个对象除了有前面说过的paths属性之外还有一个shim属性，专门用来配置不兼容的模块。

每个模块要定义：

exports值：输出的变量名，表明这个模块外部调用时的名称。

deps数组：表明该模块的依赖性。

**require.js插件：**

(1) domready插件可以让回调函数在页面DOM结构加载完成后在运行;

(2) text和image插件允许require.js加载文本和图片文件;

(3) json和mdown，用于加载json文件和markdowm文件。

## 五．SASS

Sass是一种css的开发工具，提供了许多便利的方法。

## Js面向对象

1. Object.defineProperty() => 修改属性默认的特性。

接收三个参数：

Object.defineProperty(属性所在的对象，属性的名字，描述符对象);

描述符对象的属性必须是：configurable, enumerable, writable, value

var person = {};

Object.defineProperty(person, "name", {

configurable: false/true, //是否可配置

enumerable: false/true, //是否可枚举

writable: false/true, // 是否可写

value: "Beijing"

});

2. 创建对象：

(1) 工厂模式

function createPerson(name, age, job) {

var o = new Object();

o.name = name;

o.age = age;

o.job = job;

o.sayName = function() {

alert(this.name);

};

return o;

}

var person1 = createPerson("Jem", 26, "web安全");

var person2 = createPerson("Sunny", 23, "web前端");

console.log(person1, person2);

(2) 构造函数模式

function Person(name, age, job) {

this.name = name;

this.age = age;

this.job = job;

this.sayName = function() {

alert(this.name);

};

}

var person = new Person("Sunny", 23, "web");

person.sayName();

(3) 原型模式

function Person() {}

Person.prototype.name = "Sunny";

Person.prototype.age = 23;

Person.prototype.job = "web前端";

Person.prototype.sayName = function() {

alert(this.name);

}

var person1 = new Person();

person1.sayName();

var person2 = new Person();

person2.sayName();

使用hasOwnProperty()方法可以检测一个属性是否存在于实例中，还是原型 中。这个方法只在给定属性存在于对象实例中时，才会返回true。

function hasPrototypeProperty(object, name) {

return !object.hasOwnProperty(name) && (name in object);

}

使用hasPrototypeProperty()方法可以检测一个属性是否存在于原型中。

hasPrototypeProperty(对象名, 属性名);

有两种方式使用in操作符：

(1) 单独使用: in操作符会在通过对象能够访问给定属性时返回true，无论该属性存在于实例中还是原型中;

function hasPrototypeProperty(object, name) { return !object.hasOwnProperty(name) && (name in object);

}

(2) 在for-in循环中使用。

var o = {

toString: function() {

return "My Object";

}

}

for(var prop in o) {

if(prop == "toString") {

alert("Found toString!");

}

}

***扩展：***

对象可枚举和不可枚举属性：

在js中对象分为可枚举和不可枚举之分，他们是由属性的enumerable值决定的，可枚举属性决定了这个属性能否被for in查找遍历到。

怎么判断属性是否可枚举：

js基本包装类型的原型属性是不可枚举的，如Object, Array, Number【Number中内置的属性是不可枚举的,所以不能被for in访问到】

Object对象的propertyIsEnumerable()方法可以判断此对象是否包含某个属性，并且这个属性是否可枚举。但是注意：如果判断的属性存在于Object对象的原型内，不管它是否可枚举都会返回false。

枚举性的作用：

属性的枚举性会影响以下三个函数的结果：

for in

Object.keys()

JSON.stringify()

要取得对象上所有可枚举的实例属性,可以使用es5的Object.keys()方法。这个方法接收一个对象作为参数，返回一个包含所有可枚举属性的字符串数组。

function Person() {}

Person.prototype.name = "Sunny";

Person.prototype.age = 23;

Person.prototype.job = "web前端";

Person.prototype.sayName = function() {

alert(this.name);

}

var keys = Object.keys(Person.prototype);

console.log(keys); // ["name", "age", "job", "sayName"]

var p1 = new Person();

p1.name = "Rob";

p1.age = 26;

var p1keys = Object.keys(p1);

console.log(p1keys); // ["name", "age"]

如果想要得到所有的实例属性，无论是否可枚举，都可以使用Object.getOwnPropertyNames()方法。

function Person() {}

Person.prototype.name = "Sunny";

Person.prototype.age = 23;

Person.prototype.job = "web前端";

Person.prototype.sayName = function() {

alert(this.name);

}

var keys = Object.getOwnPropertyNames(Person.prototype);

console.log(keys); //["constructor", "name", "age", "job", "sayName"]

(4) 简单的原型语法：

function Person() {}

Person.prototype = {

"name": "Sunny",

"age": 23,

"job": "Web Security",

"sayName": function() {

console.log(this.name);

}

}

var p1 = new Person();

p1.sayName(); //Sunny

(5) 组合使用构造函数模式和原型模式

构造函数模式用于定义实例属性，而原型模式用于定义方法和共享的属性。

function Person(name, age, job) {

this.name = name;

this.age =age;

this.job = job;

this.friends = ["Jem", "Sunny"];

}

Person.prototype = {

constructor: Person,

sayName: function() {

alert(this.name);

}

}

var person1 = new Person("Sunny", 23, "web前端");

var person2 = new Person("fYH", 23, "Java开发");

person1.friends.push("Van");

console.log(person1.friends); //["Jem", "Sunny", "Van"]

console.log(person2.friends); //["Jem", "Sunny"]

console.log(person1.friends === person2.friends); //false

console.log(person1.sayName === person2.sayName); //true

(6) 动态原型模式

(7) 寄生构造函数模式

创建一个函数，该函数的作用仅仅是封装创建对象的代码，然后再返回新创建的对象，但从表面上看，这个函数又很像典型的构造函数。

function Person(name, age, job) {

var o = new Object();

o.name = name;

o.age = age;

o.job = job;

o.sayName = function() {

alert(this.name);

}

return o;

}

var friend = new Person("Sunny", 23, "web前端");

friend.sayName();

(7) 稳妥构造函数模式

3. 继承：

原型链继承

借用构造函数