

ES6

Babel

Setup

- Browser
- Cli
- Webstorm
- Webpack
- <https://www.babeljs.cn/setup#installation>

ES6变化...

let&const

- 块级作用域
- 不存在作用域提升的问题

新的运算符及定界符

- 反引：模板字符串，可以使用多行文本，`${}:`模板字符串中的变量引用
- `... :rest`运算符：表示把余下的东西强制解构到数组中
 - `fn(...rest)`, 将所有的实参解构到`rest`数组中
 - `[a,...b]=[1,2,3,4]`, 将`2,3,4`解构到数组中
 - `[...a,...b]`: `a,b`两个数组合并
 - `let mm=[...set]`: 将`Set`解构到数组`mm`中

解构, spread运算符, rest参数

```
1 // 解构赋值演示
2 let [a,b,c]=[1,2,3]//按顺序解构
3
4 let {name,age}={age:100,name:'john'}//按名称解构
5
6 function add({a,b}){
7     return a+b
8 }
9 let rs=add({a:1,b:2})//此处发生解构,但变更名称不可变化 (按名称解构)
10
11 //spread运算符
12 let as=[1,...[2,3],...[4,5]]//数组变列表[1,2,3,4,5]
13
14 //rest参数(注意: 此处...不是spread运算符)
15 function add(...rest){
16     return rest.reduce((a,c)=>a+c,0)
17 }
18 //可以这样调用
19 add( rest: 1,2,3)
20 let m=add(...[1,2,3])
```

destructuring

```
1 // 解构赋值演示
2 let [a,b,c]=[1,2,3]//按顺序解构
3 let {name,age}={age:100,name:'john'}//按名称解构
4
5 function add({a,b}){
6     return a+b
7 }
8 let rs=add({a:1,b:2})//此处发生解构,但变更名称不可变化 (按名称解构)
```


Symbol

- 第七种数据类型
- 解决：对象的属性名称重复问题
- `obj[Symbol('name')]='john'`

ES6的字符串

模板语法和模板标签

- `let s=`长文本中间可以换行或加入html标签``
- `let a=10;s=`a自增的值是${++a}`;`

新增加的函数

- 对原有的indexOf的搜索增强：
 - .startsWith/.endsWith/.includes
 - 参数：(target, start(0))
- 自动生成字符串
 - .repeat/.padStart/.padEnd
- for...of遍历
- trim

ES6的集合

数组的新玩法

- 兼具: `array-like`和`iterable` 特征(`for..in`/`for..of`)
- 生成数组的新方法:
 - `Array.from(array-like/iterable)` : 针对`nodeList`, `arguments`, `set`, `map`
 - `Array.of(item...)`: 取代`new Array(item...)`
 - `.copyWithin(target, start, end(len))`
- 查找函数（高阶）：
 - `.find(fn)/findIndex(fn)`: 比`indexOf`更加强大
- 其它
 - `.includes(ele)`: return T/F
 - `.fill(x, start(0), end(len))`
 - `.keys()`, `values()`, `entries()`

indexOf VS findIndex

- indexOf是基于===逻辑
 - 对于如NaN的判断，无能
 - 对象的自定制判断，无能
- findIndex使用回调函数进行判断

别忘了

- Queue/Stack的操作
 - pop, push, shift, unshift
- 数据库操作
 - slice/splice/indexof
- 排序
 - sort(fn), reverse
- 高阶遍历(回调中的三个参数: item, index, ary)
 - forEach/filter/every/some/map/reduce

Set/WeakSet

- 代表着无重复的集合：
 - `new Set([1,2,2])`: set的size为2
 - `let m=[...set]`, m为数组
- **WeakSet**: 代表着对象的引用集合，其引用为弱引用，不影响垃圾回收。

Map/WeakMap

- 为解决Object只能使用字符串做为“键”的不足

```
const m = new Map();
const o = {p: 'Hello World'};

m.set(o, 'content')
m.get(o) // "content"

m.has(o) // true
m.delete(o) // true
m.has(o) // false
```

函数

形参变化

- 默认形参
 - `function fn(a=0,b=0){}`
 - 副作用：此形参个数不会计入`fn.length`
- **rest形参(无需arguments)**
 - `function fn(...values){values为数组形式}`

箭头函数

- 用来取代匿名函数的传入方式
- 形式如下:
 - `(a,b)=>{...;return result}`
 - `a=>{...;return result}`
 - `a=>result`
 - `()=>result`
- 注意`this`
 - 箭头函数中没有`this`, 它会把`this`当做变量一直向上查找
 - `this`始终指向它的定义位置

```
2 global.name="tom";
3 let person={
4   name:"john",
5   //箭头函数法
6   sayName:function () {
7     return ()=>this.name;
8   },
9   //闭包法
10  sayName1:function () {
11    let name=this.name;
12    return function () {
13      return name;
14    };
15  },
16  //绑定法
17  sayName2:function () {
18    function inner() {
19      return this.name;
20    };
21    return inner.bind(this);
22  },
23  sayNameFail:function () {
24    return function () {
25      return this.name;
26    };
27  }
28 }
```

```
29 //测试
30 let sayNameFunction=person.sayName2();
31 let name=sayNameFunction();
32 console.log(name);
```

Promise

<https://www.jianshu.com/p/9b80bfaf7b9d>

Promise的由来

1. “异步函数+回调函数的古怪”调用方式，让多数码民接受不易
2. **Callback Hell** 让代码结构难以被理解和管理
3. JS社区有一部分程序员，想以”祖传的同步”方式，调用异步函数！！！！
4. ES6顺应了这个要求，提出了**Promise**对象，做为异步函数的返回结果（传统异步函数是没有返回结果的）
5. 《人类文明史之ES篇》又完成一次进化，离底层又远一步。

普通异步函数的编写

```
//一个普通的异步函数设计
function openFile(fileName,callback) {
  console.log("正在打开文件...")
  setTimeout(function () {
    let fd=23;
    callback(fd);
  },500);
}

//callback hell 方式的调用
openFile("abc.txt",function (fd) {
  console.log(fd);
});
```

Promise函数的编写

```
//使用Promise设计的异步函数
function openFileWithPromise(fileName){
    let promise=new Promise(function (resolve,reject) {
        console.log("正在打开文件, pending状态开始....")
        setTimeout(function () {
            //获取一个随机正负的文件描述符,
            let fd=10-Math.floor(Math.random()*20)
            if(fd>0) resolve(fd) //状态变为fulfilled
            else reject(new Error("文件打开失败")) //状态变为rejected
        },500);
    })
    return promise
}

//异步函数, 使用同步调用的方法, 容易被理解
let promise=openFileWithPromise("ab.txt");

promise
    .then(function (fd) {
        console.log("获取文件: "+fd);
    }).catch(function (err) {
        console.log(err)
    })
```

连接多个Promise函数

- 解决回调地狱的有效手段

```
function fopen(filename) {  
    return new Promise(function (resolve, reject) {  
        setTimeout(function () {  
            //随机概率，表示成功75%与失败25%  
            let fd=10-Math.floor(Math.random()*16)  
            if(fd>=0) resolve(fd);  
            else reject(new Error("无读取权限"))  
        }, 500)  
    })  
}  
  
function fwrite(fd) {  
    return new Promise(function (resolve, reject) {  
        //随机概率，表示成功75%与失败25%  
        let len=10-Math.floor(Math.random()*16)  
        if(len>=0) resolve(len);  
        else reject(new Error("磁盘已满"))  
    })  
}
```

```
fopen("abc.txt")  
    .then(function (fd) {  
        console.log("打开文件号: "+fd)  
        return fwrite(fd);  
    })  
    .then(function (len) {  
        console.log(`写入了${len}个字节`)  
    })  
    .catch(function (err) {  
        console.log(err)  
    })  
    .catch(function (err) {  
        console.log(err)  
    })
```

async函数

- 语法糖：是一个以**promise**为返回结果的异步函数
- 语法糖：返回的结果，将自动的被包装成为**promise**对象
- 内部对**async**函数的调用，可以采用**await**方式，表示：
 - 下一步操作要等待本函数执行结束。
 - 本函数的返回结果就是**resolve()**的结果

```
1  async function f1() {  
2      return 100;  
3  }  
4  async function f2(){  
5      let rs=await f1();  
6      return rs*100;  
7  }  
8  f2().then(x=>console.log(x))  
9  console.log('end')
```

基于类的编程

本质

- 是语法糖
- 原理：是基于原型链和特性定义
- 适用：应用级项目
- 对于库级项目，坚持使用函数

```
//类定义, 无变更作用域提升
class Human{
    //构造器, 默认存在无参的, 并且不会被覆盖
    constructor(name){
        this.inner={}; //用来存放成员的
        this.name=name;
    }
    //静态方法, 但实例无法使用
    static staticFn(){
        console.log("static fn..")
    }
    //普通方法
    sayName(){
        console.log(this.name);
    }
    //添加成员的setter和getter
    set job(job){
        this.inner.job=job;
    }
    get job(){
        return this.inner.job;
    }
}
```














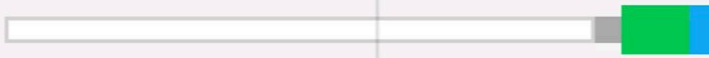










```
//关于继承
class Animal{
    constructor(){
        this.age=100; //子类继承得到
    }
    breath(){ //子类可以得到
        console.log("Animal breath....");
    }
    static staticFnPro(){ //静态方法同样获得继承
        console.log("Animal staticFnPro....")
    }
}
Animal.ff=function () {
    console.log("ff");
}

class Human extends Animal{
    constructor(name){
        super(); //默认调用
        this.name=name;
    }
    //Override Super breath
    breath(){
        console.log("Human breath..")
    }
}
```

模块系统

需求

- 全局环境的污染问题
- 浏览器的多JS加载问题
- 前端项目，需要更大的规模，更多的“源文件”
- 如何管理
- 目前的三种手段：AMD/CMD, ES6, 前者为异步加载，后两者需要借助于工具完成。

Name	St...	Type	Initiator	...	Ti	Waterfall			
 index.html?_i...	200	d...	Other	...	14				
 t1.js	200	sc...	index.html?...	...	23				
 t9.js	404	sc...	index.html?...	...	28				
 t3.js	200	sc...	index.html?...	...	21				
 jquery.js	200	sc...	index.html?...	...	25				
 a.jpg	200	jpeg	index.html?...	...	6 r				
 b.jpg	200	jpeg	index.html?...	...	6 r				
 c.jpg	200	jpeg	index.html?...	...	4 r				
 t4.js	200	sc...	index.html?...	...	24				
 t5.js	200	sc...	index.html?...	...	28				
 t6.js	200	sc...	index.html?...	...	18				
 t7.js	200	sc...	index.html?...	...	18				

我的webpack+babel脚手架

- https://gitee.com/bj_java_161221/
- `$:git clone url`
- `$:npm intall`
- `$:npm start`

几种导出，引入的方式

```
//module1.js
export var m=100;
export function fn(){}
```

```
//main.js
//原名
import {m,fn} from './module1'
import {fn as util} from './module1'
```

```
//module2.js
//默认导出，相当于export {fn as default}
export default fn
function fn(){}
```

```
//main.js
//相当于import {default as MyUtil}
import MyUtil from './module2'
```