Part 1 es6 + 异步编程 + TS

笔记本: JavaScript
创建时间: 2020/5/14 14:21 **更新时间:** 2020/5/30 21:19
作者: 兔子先生
URL: https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Functions/...

---

1. **初识ECMAScript**

    1. ECMAScript是一种标准化的脚本语言
    2. javascript基于ECMAScript规范实现，是ECMAScript的扩展语言（es只定义了语法，如条件语句，js实现了es这种标准的语言）

    浏览器中 js = es + 浏览器提供的api（dom +bom等）
    node中 js = es + node中的api(fs + net等)

2. **ES6**

    2-1. let和const

    1. 块级作用域：
       问题：js原本只有全局和函数作用域，
       1. var声明的变量是该变量所在作用域中的局部变量；省略var操作符可以直接创建一个全局变量；（函数中创建的变量只在函数被调用时创建，函数调用完成，则定义的局部变量被销毁，全局变量仍可以被访问）
       2. 通过var声明的变量始终会被提升至当前作用域顶部（同一个作用域中有多个代码块时同名变量就会冲突）
       3. 在if或for这种代码块中var的变量就会自动存在于当前作用域顶部；省略var定义的变量还会提升至全局，容易出现变量冲突

       方案：为所有代码块划定作用域；
    2. 定义全局对象：
        1. let 和 const在全局作用域定义变量时，会生成一个新的绑定，而不会直接挂在window上，也就不会覆盖原有的全局变量，但会遮蔽原来的全局变量；

```
let RegExp = 'hello'
console.log(RegExp) // hello
console.log(RegExp === window.RegExp) // false

const ncz = 'hi'
console.log(ncz) // hi
console.log(ncz in window) // false
```

    3. 循环

        1. 在循环中使用let和const定义的循环变量在每次循环中都会被重新创建；注意：const定义的循环变量不能被更改，所以for循环中的循环变量不能使用const

    2-2 解构赋值

    2-3 模板字符串

    1. 标签模板：每个模板标签都可以执行模板字面量上的转换并返回一个字符串；（相当于自定义``模板字符串的替换规则）

1. 标签可以是一个函数，接收不定参数

```
let message = tag`hello, ${a} is a ${b}`
// 可以控制字符替换的结果
// return string 替换的结果
function tag(literals, ...substitutions) {
    // literals 数组，以${}为分隔得到其他静态部分的数组
    // substitutions 所有${}部分的值
}
```

2-4 字符串扩展方法

1. startsWith() 'hello. u'.startsWith('h') // true
2. endsWith()
3. indexOf()

2-5 参数默认值

1. 带默认值的形参放在参数列表末尾，否则影响取值
2. 不定参数：...args 获取不定形参
    1. 只能用在末尾
    2. arguments.length：始终包含函数**实际接收到**的所有参数，因此length长度 = 命名参数个数 + 不定参数个数
    3. 对象字面量的setter中不能使用不定参数，因为setter只允许接受一个参数
       let object = {
          set name(...value) { // 语法错误
             // 一些逻辑
          }
       }
3. arguments:
    1. es5非严格模式：命名参数的变化**同步更新**到arguments，即改变参数值，arguments中的值也变化
    2. es5严格模式：**不会**同步更新，即arguments始终是函数调用时形参的初始化值
    3. es6严格和非严格模式：**均不会**同步更新
    4. **函数传递参数时不适用arguments，原因1：es6和es5严格模式中值不会同步更新；原因2：箭头函数没有arguments**
4. 默认参数取值：在函数调用时，参数默认值才会真正赋给函数形参，所以
    1. 位置在前的形参可以作为后面形参的默认值，反过来不行（TDZ）
5. TDZ：默认参数的临时死区
    1. 函数参数有自己的作用域和TDZ，独立于函数作用域
    2. 标识符：定义参数时会为每个参数创建一个新的标识符绑定，且在函数被调用时，才初始化；形参在初始化之前不可引用
6. 函数的length属性：**只统计函数命名参数的数量**

```
function foo(a, ...args) {
    console.log(arguments.length) // 3
}

foo(1,2,3)

console.log(foo.length) // 1

Math.max.apply(Math, [1,2,3,4]) // apply的机制是什么？？
```

插入：函数

1. name属性：
    1. es6中的所有函数都有一个name属性
    2. 声明式函数：name为函数名

3. 表达式式函数：name为赋值的变量名；
    1. 特殊情况：var foo = bar() {} foo.name // bar 因为函数表达式有名字，且权重高于被赋值的变量
    4. bind创建的：带bound前缀 foo.bind().name // bound foo
    5. Function构建的：anonymous
2. call和construct： js函数有这两个内部方法
    1. new调用函数时：执行[[construct]]方法，创建一个新对象（实例），再执行函数体，把this绑定到实例上
    2. 普通调用：执行[[call]]方法，直接执行函数体
    3. 判断函数是否通过new调用：
        1. es5 ： a instanceof Foo
           function Foo() { console.log(this instanceof Foo) }
           特殊情况：var foo = new Foo() bar.call(foo)
        2. es6 : 元属性new.target
            1. 调用[[construct]]方法时，会将new.target赋值为构造函数 因此在构造函数内部可以判断new.target === Foo；普通调用时，new.target为undefined
3. 块级函数：在代码块如if中定义的函数（es5中不支持这种定义，es6支持，会把函数当做一个块级声明）
    1. 声明式定义函数，函数会被提升到块的顶部（严格模式）；非严格模式会提升至外围函数或全局作用域顶部
    2. 表达式式定义函数，不会提升

2-6 数组展开

1. ...[1,2,3]

2-7 箭头函数（设计用来替换匿名函数表达式）

1. 简化函数写法
2. 没有单独的this、super、arguments、new.target：只会从自己的作用域链的上一层继承这几个（即外围最近一层非箭头函数）；箭头函数作为对象的方法要注意this不指向对象自己
3. 通过call或apply调用：因为没有自己的this，所以改变this指向不生效，只会去取上层this
4. 不绑定arguments：只会引用了封闭作用域内的arguments（上层）
5. 不绑定constructor：不能使用new
6. 没有prototype
7. 不能使用yield：除非嵌套在允许使用的函数内
8. 特殊的解析顺序：

```
var name= 'tom';
var Person = {
    name: 'li',
    say: function() {
      setTimeout(() => {console.log(123, this.name)},1000)
    }
}

Person.say() // 123 "li"  this指向外层及作用域链的上一层

var name= 'tom';
var Person = {
    say: function() {
```

```
      setTimeout(() => {console.log(123, this.name)},1000)
    }
}

Person.say() // 123 undefined this仅指向上一层，并不会一直向上查找


let callback;

callback = callback || function() {}; // ok

callback = callback || () => {};
// Uncaught SyntaxError: Malformed arrow function parameter list

callback = callback || (() => {});    // ok
```

2-8 对象字面量的加强

1. 对象属性同名变量简写　var a = 1, var obj = {　a,　b　}
2. 函数属性定义简写　var obj = {　a,　b, foo() {}　}
3. 计算属性名直接在对象定义时添加　var obj = {　a,　b, foo() {}, [1+2]: 'abc' }

2-9 对象方法加强

1. Object.assign(target, source1, source2):
    1. 将source的属性copy给target，并覆盖同名的属性；
    2. 返回新对象，新对象与target同引用
    3. 如果source对象中有访问器属性（get set）,不会被赋值给target，但会在target中添加一个对应的数据属性
2. 比较像等
    1. ===（现在）
        1. +0 === -0 // true 在数学问题中不合理
        2. NaN === NaN // false　应该相等
    2. Object.is()　（新增）
        1. +0 === -0 // false
        2. NaN === NaN // true
3. 自由属性枚举顺序：
    1. 即Object.getOwnPropertyNames()　Reflect.ownKeys获取的key值顺序，Object.assign生成的对象的属性会自动按如下规则排序　；es5时浏览器各自实现
        1. 数字属性升序排列
        2. 字符串键按加入顺序
        3. symbol键按加入顺序
    2. for in没有明确规则：Object.keys，　JSON.stringify与for in保持一致
4. 增强对象原型：通过原型继承（Object.create(a)或Objecct(a)）得到的对象b，原型指向a，且a的属性成为b的原型属性，
    1. Object.setPrototypeOf(b, c)：将b的原型指向c，操作的其实是对象的内部属性[[Prototype]]
    2. super：指向指向原型对象的指针
    3. 对象的方法属性：在es6中定义为函数，有一个内部属性[[HomeObject]]，指向对象
5. 对象属性读写
    1. Object.defineProperty() (es5)：定义不可枚举不可写属性
    2. Proxy(es6)：拦截并改变底层js引擎操作的包装器
    3. 对比：
        1. object.defineProperty不能监听数组变化；必须遍历对象的每个属性；必须深层遍历嵌套的对象

2. Proxy可以监听到对象上更多的操作，例如：

handler方法	触发方式
get	读取某个属性
set	写入某个属性
has	in 操作符
deleteProperty	delete 操作符
getPrototypeOf	Object.getPrototypeOf()
setPrototypeOf	Object.setPrototypeOf()
isExtensible	Object.isExtensible()
preventExtensions	Object.preventExtensions()
getOwnPropertyDescriptor	Object.getOwnPropertyDescriptor()
defineProperty	Object.defineProperty()
ownKeys	Object.getOwnPropertyNames()、Object.getOwnPropertySymbols()
apply	调用一个函数
construct	用 new 调用一个函数

4. Reflect：一系列操作对象的方法
5. Promise
6. class
    1. 定义类型的新方法：替代了es5中先定义 function Person() {} 再添加prototype方法的方式
    2. static：静态方法
7. extends
8. Set集合：成员不重复
    1. 方法： add size has delete clear
9. Map键值对集合：
    1. 不同对象键只能是字符串（非字符串会被toString()后作为键）
    2. Map中的键不会被做转化；方法：set get has delete clear new Map().forEach((value, key) => {})
10. Array.from()

6. Symbol： 生成独一无二的字符
    1. 作为对象属性名：防止重复
    2. 作为内部属性名：外部不知道属性标识符，不能访问
    3. Symbol.for(sign): 生成全局变量，
    4. 不可枚举：Object.keys for in JSON.stringify 访问不到Smybol属性; `Object.getOwnPropertySymbols()`来获取
    5. Symbol.toStringTag: 'xxx'：为对象定义toString()的标识符； obj = {[Symbol.toStringTag]: 'aaa' } obj.toString() ===> [object aaa]
    6. Symbol.iterator：Symbol常量
7. 迭代器：
    1. `for...of`：遍历可迭代数据类型`Interable`（内部实现了可迭代接口`Iterator`：拥有`Symbol.iterator`这个`Symbol`属性，函数类型，返回值为迭代器对象，迭代器对象一定有一个用来迭代的`next`方法，返回`InteratorResult`对象`{value: '', done: true}`）？？？？
    2. for...in： 键值对
    3. 循环变量item是数组（伪数组，元素集合，Set）的值；可break
    4. 遍历Map：item是数组【key，value】
8. 生成器：解决异步回调嵌套
    1. 生成器函数： function * （）{}
        1. 自动返回一个生成器对象，调用生成器对象的next方法，触发函数执行，遇到yield暂停执行，yield的值是next返回值的value

3. **ES7**

3-1. includes()： 数组是否包含指定元素；返回 boolean；可查找NaN
3-2. **：指数运算符　2 ** 10 ====》 Math.pow（2,10）

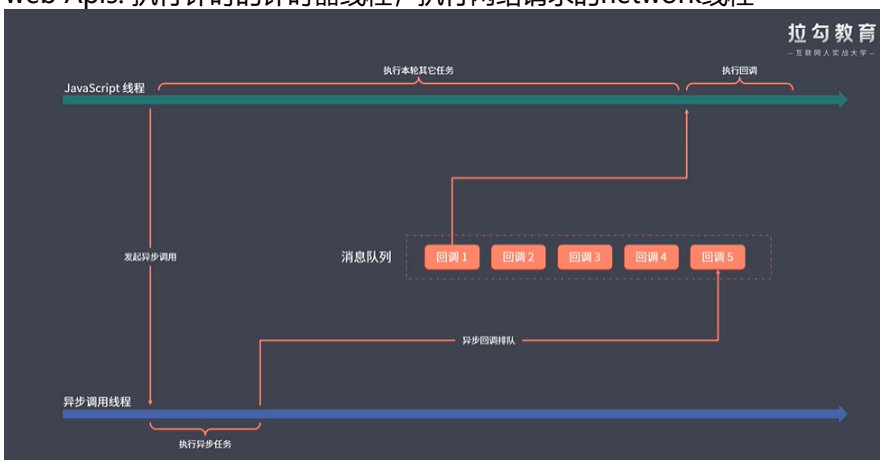
4. **ES8**

4-1. Object.values： 返回对象所有值组成的数组
4-2. Object.entries: 返回一个二维数组，每一项是数组的【key, value】

4-3. Object.getOwnPropertyDescriptors(): 返回一个对象的完整描述，包含get set
4-4. String.prototype.padEnd / String.prototype.padStart：使用指定字符填充字符串到指定位数　var a = 6; a.toString().padStart(2, '0')
4-5. 函数参数可以添加尾逗号
4-6： Async/await

5. **Js异步编程**

1. 异步编程的类型
    1. ES 6以前： 回调函数； 事件监听（click）；定时器setTimeout() setInterval()
    2. ES 6：Promise
    3. ES7 ： Generator函数(协程coroutine)
    4. ES8 ： async和await
2. 同步模式 / 异步模式
    1. event loop: 监听调用栈（call stack）和消息队列（queue）
    2. web Apis: 执行计时的计时器线程；执行网络请求的network线程



    3. 回调函数：告诉执行者要做的实情
3. promise
    1. then：
        1. 返回一个全新的promise对象=====》**链式调用**
        2. 每一个then方法是在为上一个promise对象添加状态确认后的回调、
        3. 前一个then的返回值就是后一个then的参数
        4. 前一个返回的是个promise，则后一个就作为前一个的回调
    2. catch:
        1. 为前一个promise（一般是then返回的）指定异常回调，因为promise异常会在链式中传递，前面没处理，后面就可以捕捉
        2. 对比：then第二个异常捕获参数
    3. 静态方法
        1. Promise.resolve():
            1. 普通参数：传递给promise的回调函数
            2. promise参数：直接返回这个参数promise

3. thenable：带有then方法的对象，接收 onFulfilled和 onRejected 回调，返回值可以在then中接收
2. Promise.reject()
3. Promise.all(): all ok
4. Promise.race(): first ok
4. 执行时序：
1. Promise中没有一步操作，回调也会进入异步队列排队
2. 微任务：promise MutationObserve process.nextTick

4. Generator：让异步代码扁平化
1. function * main() { yield ajax } const g = main() const result = g.next()
2. next参数可以带一个返回值，作为上一个yield表达式的返回值
3. 一个封装生成器函数调用的包： co

```
>    function * gene(a, b) {
    console.log(a, b);
    var c = yield 'c';
    console.log("c====", c);
    var d = yield "d";
    console.log("d====", d)
    return 6;
  }
  var g = gene("a", "b");
undefined
> g.next('1')
  a b
▶{value: "c", done: false}
> g.next('2')
  c==== 2
▶{value: "d", done: false}
> g.next('33')
  d==== 33
▶{value: 6, done: true}
>
```

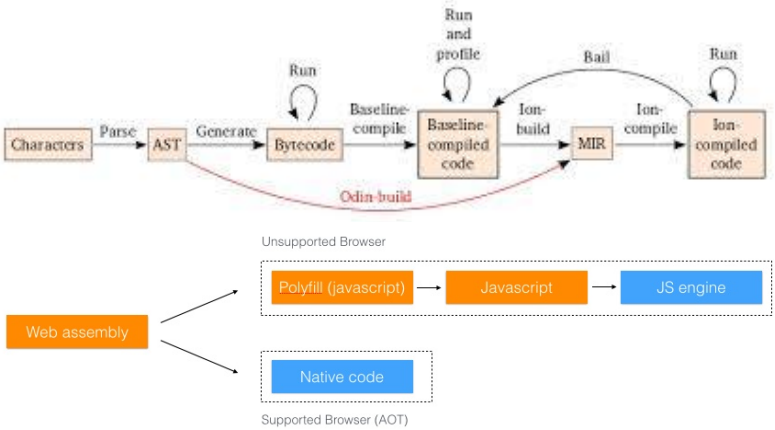
6. **TypeScript: JS类型系统**
1. 强类型 vs 弱类型（类型安全）：函数形参与实参是否强制一致，弱类型允许隐式类型转换
2. 静态类型 vs 动态类型（类型检查）：类型在声明变量时确定，且不能修改；反之
3. 为什么js是弱类型和动态类型
1. 引用简单：设计之初应用于简单的项目
2. 不需要编译：js是脚本语言，可直接执行；强类型是在编译阶段进行类型检查
3. 问题：
1. 类型异常需要等待运行时发现
2. 类型异常可能导致执行结果异常
3. 规避变量定义时的类型异常，如对象的属性名类型异常导致取值不准确

4. 强类型的优点：
1. 类型检测：编码时发现可能的类型异常错误
2. 智能提示：为确定类型的变量提供代码提示
3. 方便重构：方便全局替换
4. 类型判断：类型不匹配不能接收，避免因类型不匹配导致调用、执行错误

5. JS类型检测
1. Flow：FB提供的小工具，为变量定义类型，实现编译时类型检测，可以在发布之前通过babel等工具去掉类型定义相关代码
1. 检测：开发阶段

2. 生产：使用插件去掉types 问题：侵入式和破坏式的代码编译是否会有隐患？？？？？
3. 类型推断：主动推断为定义的异常类型，给出提示

2. TypeScript：js的超集
    1. 生态好，支持不错
    2. js是解释性语言执行较慢，所以v8对js做了个JIT（即时编译），即运行时编译来提高运行速度，JIT编译时需要对变量做类型定义，但是js是动态类型，导致编译时遇到被改变类型的变量需要重编编译，这各重编的开销不小，所以产生了
        1. TS、Dart、JSX等，支持强类型定义，再编译成js即可（去掉类型定义）解决了大部分JIT重编问题
        2. FF的asm.js：做标记说明类型 如 function (x) { var x = x|0 // x int 或的语法 } 潜力更大，可以直接编译（ahead of time） 性能阶级原生
        3. web assembly：基于asm的思想，将代码（可以使任意编程语言） 直接编译成AST（不直接编译成字节码是因为ast更容易压缩也更容易翻译），然后编译成字节码
        https://www.zhihu.com/question/31415286



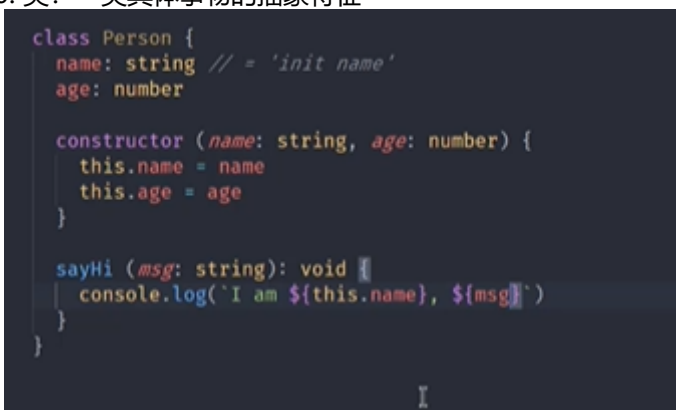
3. 增加较多新概念，学习成本增加
4. 小项目而言增加开发成本
5. 使用
    1. ts可以将es6+转化为es5，wp也可以
6. 语法：
    1. 对象
        1. object不单纯指对象：obj:object = function () {} // or [ ] // or {}
        2. 最想字面量形式类型定义：obj: { foo: number} = { foo: 1 }
    2. 数组
        1. arr: Array<number>
        2. arr: number[]
        3. 编译阶段类型解决了，运行时的类型问题怎么解决？？
    3. 元祖：明确元素数量及类型的数组
        1. arr: [number, string] = [1, 'lee']
    4. 枚举
        1. 普通枚举：
            1. enmu a { b = 0, c, d} 枚举的变量的值默认从0开始累加，如果第一个指定了数值类型，则从第一个元素的值开始累加； obj = { status： a.b }
            2. tsc编译后会生成一个枚举对象，以枚举值为属性保存枚举的key，以key为属性保存枚举的值
        2. 常量枚举：
            1. const enmu a{}

2. 编译后会使用枚举属性的地方会被直接替换为枚举的值，枚举的key以注释的形式标注

5. 函数
    1. 声明式 function fun(a: number, b:string, c?: object): string{}　c? ： number可选；c:number = 10 可选 ...rest: number[] 不定参数
    2. 表达式：可以通过箭头函数形式表示变量被赋值的函数所接受的参数和返回值类型
6. 任意类型： a: any
7. 隐式类型推断：let a = 10 // number let b; // any
8. 类型断言：nums = [1,2] const num = nums.find(i => i >0) // num推断为number或undefined
    1. const num1 = num as number
    2. const num2 = <number>num // 与jsx标签在格式上会冲突
9. 接口 interface

```
interface interfaceName {
    title: string,
    name: string
    age: number;
    subTitle?: string;
    readonly summary: string
    [prop: string] : string // 动态成员
}
```

10. 类：一类具体事物的抽象特征



    1. 类的属性在使用前要先定义类型
    2. 类的属性必须有初始值

11. 类属性修饰符 private（私有）protected（受保护的，可被继承）public（公有）

12. 类的只读属性：readonly a a只能在定义时或者constructor中被修改

13. 类与接口：interface
    1. 接口定义可以限制类中的成员定义：interface Eat{ eat (food: string): void }　class People implements Eat () {} People类必须实现eat方法
    2. 接口定义粒度应该尽可能小，方便组合使用 class People implements Eat, Run() {}

14. 抽象类：abstract
    1. 只能被继承，不能new
    2. 抽象方法，必须在子类中被实现

15. 泛型：在声明是不指定类型，调用时传入类型　动态确定类型

```ts
function createNumberArray (length: number, value: number): number[] {
  const arr = Array<number>(length).fill(value)
  return arr
}

function createStringArray (length: number, value: string): string[] {
  const arr = Array<string>(length).fill(value)
  return arr
}

function createArray<T> (length: number, value: T): T[] {
  const arr = Array<T>(length).fill(value)
  return arr
}

// const res = createNumberArray(3, 100)
// res => [100, 100, 100]

const res = createArray<string>(3, )
```

createArray(length: number, value: string): string[]

16. 类型声明： declare，为三方模块添加类型声明

```ts
// 类型声明

import { camelCase } from 'lodash'

// declare function camelCase (input: string): string

const res = camelCase('hello typed')
```