作用域

全局作用域

函数作用域

块级作用域 用花括号包含的部分{}

var 定义的变量可以修改，如果不初始化会输出undefined，不会报错。作用域可提升。

let 是块级作用域，函数内部使用let定义后，对函数外部无影响。

const 是块级作用域 常量 只读 声明过后不可修改，而且必须初始化。

使用原则：不用var 主用const 配合 let

# **模板字符串**

1.反引号包裹 允许换行

const str = hello es2015, this is a string

2.可以通过 ${}

插入表达式，表达式的执行结果将会输出到对应位置

const msg = hey, ${name} --- ${1 + 2} ---- ${Math.random()}

# **带标签的模板字符串**

1.模板字符串的标签就是一个特殊的函数，

2.使用这个标签就是调用这个函数

示例：const str = console.loghello world

const name = 'tom'

const gender = false

function myTagFunc (strings, name, gender) {

参数 strings：由 ${} 分割的数组 其余参数 对应相应的 ${} 传的参数

// console.log(strings, name, gender)

// return '123'

const sex = gender ? 'man' : 'woman'

return strings[0] + name + strings[1] + sex + strings[2]

}

const result = myTagFunchey, ${name} is a ${gender}.

# **字符串的扩展方法**

const message = 'Error: foo is not defined.'

console.log(

message.startsWith('Error')

message.endsWith('.')

message.includes('foo') )

startsWith：以什么开始

endsWith：以什么结束

includes：是否包含

# **数组结构解析**

const arr = [1, 2, 3]

const [a1, a2] = arr

const [foo, ...rest] = arr

const [foo,,last='def'] = arr

const pat='/foo/bar/baz'

const [,rootdir]=path.split('/')

# **对象结构解析**

根据属性名称匹配

const object1={name:"aa"};

const {name}=object1;

如果有变量冲突 可使用重命名的方式

const {name:rename}=object1;

# **函数参数的默认值**

1.短路运算很多情况下是不适合判断默认参数的，例如 0 '' false null

例如：enable = enable || true

//默认参数一定是在形参列表的最后 不能混合写

**function** foo (**enable** = **true**) {

console.log('foo invoked - enable: ')

console.log(**enable**)

}

foo(**false**)

# **剩余参数**

**function** **foo** (first, ...args) {

参数：...args 需要放在最后

console.log(args)

}

foo(1, 2, 3, 4)

# **箭头函数**

1.极大简化了回调函数

2.不会改变this指向，没有this机制 指向当前作用的this

有且只有一个参数**const** fun = n => n+1**const** fun = () => {}**const** person = {

name: 'tom',

*// sayHi: function () {*

*// console.log(`hi, my name is ${this.name}`) name=tom*

*// }*

sayHi: () => {

console.log(`hi, my name is ${**this**.name}`) name=undefined

},

sayHiAsync: **function** () {

*// const \_this = this*

*// setTimeout(function () {*

放到全局作用域中调用

*// console.log(\_this.name)*

*// }, 1000)*

console.log(**this**)

setTimeout(() => {

*// console.log(this.name)*

console.log(**this**)

}, 1000)

}

}

person.sayHiAsync()

# **数组展开**

数组参数展开

**const** arr = ['foo', 'bar', 'baz']

*// console.log(*

*// arr[0],*

*// arr[1],*

*// arr[2],*

*// )*

*// console.log.apply(console, arr)*

console.log(...arr)

# **字面量的增强**

对象字面量的属性名 可以直接通过[]直接书写

# **object.assign**

将多个源对象中的属性复制到一个目标对象中

可用于复制对象不影响源数据

# **[object.is](http://object.is" \t "C:/Users/AAA/AppData/Local/Youdao/YNote/markdown/_blank)**

方法判断两个值是否是相同的值。

此方法的比较类似于 ‘===’ 但是还有一些不同之处，

比如：

// 使用 ‘===’

+0 === -0 //**true**

**NaN** === **NaN** // **false**

// 使用 **Object**.**is**()

**Object**.**is**(+0, -0) // **false**

**Object**.**is**(**NaN**, **NaN**) // **true**

# **对象代理**

第一个参数被代理的对象 第二个key名称 第三个value值

get:监视属性读取 set:监视属性设置

优势 1.可以监视读写以外的操作 2.可以很方便的监视数组操作 3.不需要侵入对象

# **Reflect**

静态类 内部封装了一系列对象的底层操作 Reflect成员方法就是Proxy处理对象的默认实现

地址：[https://www.cnblogs.com/houfee/p/11369609.html](https://www.cnblogs.com/houfee/p/11369609.html" \t "C:/Users/AAA/AppData/Local/Youdao/YNote/markdown/_blank)

Reflect.apply(target, thisArg, args)Reflect.construct(target, args)Reflect.get(target, name, receiver)Reflect.set(target, name, value, receiver)Reflect.defineProperty(target, name, desc)Reflect.deleteProperty(target, name)Reflect.has(target, name)Reflect.ownKeys(target)Reflect.isExtensible(target)Reflect.preventExtensions(target)Reflect.getOwnPropertyDescriptor(target, name)Reflect.getPrototypeOf(target)Reflect.setPrototypeOf(target, prototype)

*// function Person (name) {// this.name = name// }// Person.prototype.say = function () {// console.log(`hi, my name is ${this.name}`)// }*

es6**class** **Person** {

**constructor** (name) {

**this**.name = name

}

say () {

console.log(`hi, my name **is** ${**this**.name}`)

}

<!--静态方法 不能通过实例化对象调用 **this**不会只想实例对象而是指向当前的类型 -->

static **fun** (name) {

}

}**const** p = new Person('tom')

p.say()

# **继承 extends**

**class** **Person** {

**constructor** (name) {

**this**.name = name

}

say () {

console.log(`hi, my name is ${**this**.name}`)

}

}

**class** **Student** **extends** **Person** {

**constructor** (name, number) {

**super**(name) *// 父类构造函数*

**this**.number = number

}

hello () {

**super**.say() *// 调用父类成员*

console.log(`my school number is ${**this**.number}`)

}

}**const** s = **new** Student('jack', '100')

s.hello()

# **Set**

所添加的值不可重复.add.has.size.delete.clear

*// 应用场景：数组去重*

const arr = [1, 2, 1, 3, 4, 1]

*// const result = Array.from(new Set(arr))*

const result = [...new Set(arr)]

console.log(result)

弱引用版本 WeakSet

差异就是 Set 中会对所使用到的数据产生引用

即便这个数据在外面被消耗，但是由于 Set 引用了这个数据，所以依然不会回收

而 WeakSet 的特点就是不会产生引用，

一旦数据销毁，就可以被回收，所以不会产生内存泄漏问题。

# **Map**

键值对集合 键可以是任意类型

**const** m = **new** Map()

**const** tom = { name: 'tom' }

m.set(tom, 90)

console.log(m)

console.log(m.get(tom))

*// m.has()*

*// m.delete()*

*// m.clear()*

m.forEach((value, key) => {

console.log(value, key)

})

# **Symobl**

永远都是唯一的 可以用过对象的私有成员

全局复用 使用 for Symbol.for：维护字符串和Symbol对应的关系 例：true 会转换成 'true'

Symbol 属性名获取 Object.getOwnPropertySymbols(obj):只能获取Symbol类型的属性名

# **Interable**

实现Interable接口就是for..of的前提

**const** set = new Set(['foo', 'bar', 'baz'])

**const** iterator = set[Symbol.iterator]()

// console.log(iterator.**next**())

// console.log(iterator.**next**())

// console.log(iterator.**next**())

// console.log(iterator.**next**())

// console.log(iterator.**next**())

**while** (true) {

**const** current = iterator.**next**()

**if** (current.done) {

break // 迭代已经结束了，没必要继续了

}

console.log(current.value)

}

# **实现可迭代接口**

在对象中挂载 Interable 方法 返回迭代器对象

*// const obj =* 此处 实现了可迭代接口 iterable 内部要有iterator方法用于返回内部迭代器

{*// [Symbol.iterator]: function () {// return*

迭代器接口 iterator 内部用于迭代的next 方法

{*// next: function () {// return*

迭代结果接口 interaitonResult

{*// value: 'zce', 当前别迭代的数组// done: true 迭代有没有结束// }// }// }// }// }*

**const** obj = {

store: ['foo', 'bar', 'baz'],

[Symbol.iterator]: **function** () {

**let** index = 0

**const** self = **this**

**return** {

next: **function** () {

**const** result = {

value: self.store[index],

done: index >= self.store.length

}

index++

**return** result

}

}

}

}

**for** (**const** item **of** obj) {

console.log('循环体', item)

}

# **迭代器模式 iterator**

对外实现统一的接口

对外提供统一的遍历接口

# **生成器 Generator**

避免异步编程中回调函数嵌套过深

定义在普通的函数后 加 \* **function** \* **foo** () {

console.log('1111')

yield 100

console.log('2222')

yield 200

console.log('3333')

yield 300

}

const generator = foo()

console.log(generator.next()) // 第一次调用，函数体开始执行，遇到第一个 yield 暂停

console.log(generator.next()) // 第二次调用，从暂停位置继续，直到遇到下一个 yield 再次暂停

console.log(generator.next()) // 。。。

console.log(generator.next()) // 第四次调用，已经没有需要执行的内容了，所以直接得到 undefined

# **生成器应用**

*// 案例1：发号器***function** \* **createIdMaker** () {

**let** id = 1

**while** (true) {

**yield** id++

}

}**const** idMaker = createIdMaker()console.log(idMaker.next().value)console.log(idMaker.next().value)console.log(idMaker.next().value)console.log(idMaker.next().value)

*// 案例2：使用 Generator 函数实现 iterator 方法***const** todos = {

life: ['吃饭', '睡觉', '打豆豆'],

learn: ['语文', '数学', '外语'],

work: ['喝茶'],

[Symbol.iterator]: **function** \* () {

**const** all = [...this.life, ...this.learn, ...this.work]

**for** (**const** item **of** all) {

**yield** item

}

}

}

**for** (**const** item **of** todos) {

console.log(item)

}

JS是一门单线程的语言，即代码只会一行一行的执行，假如中间执行函数需要等待五秒，那么就会阻塞进程，为了解决这个问题，就出现了异步编程，让程序中将来执行的部分并不一定在现在运行的部分执行后立即执行，换句话说，就是现在无法完成的任务将会异步完成，即JS引擎会将程序分为同步任务和异步任务，只有当同步任务完成之后，才会去执行异步任务，异步任务即将来再完成的任务。

我们都知道，setTimeout,ajax请求都是异步操作，JS知道他们是异步操作，所以放到异步任务里，那如果我们有一个操作是将来才能完成的，或者需要将来得到某个结果之后再完成的，我们要怎么让JS知道这是个异步操作呢，通常处理异步操作我们经历了以下进程，从一开始的回调方式，再到Promise，再到Generator，再到async/await，因为asyinc/await本质上是Generator的语法糖，所以，本文会略过Generator，着重讲Promise和async/awit。

# **同步 Syncronous**

排队执行 按照代码顺序一个一个执行

可能会引起阻塞console.log('1')**function** **bar** () {

console.log('2')

}

**function** **foo** () {

console.log('3')

bar()

}

foo()console.log('4')

输出 1324

# **异步 Asyncronous**

不会等待任务的结束，来执行下一个任务，通过回函数的方式定义

console.log('global begin')

setTimeout(**function** **timer1** () {

console.log('timer1 invoke')

}, 1800)

setTimeout(**function** **timer2** () {

console.log('timer2 invoke')

setTimeout(**function** **inner** () {

console.log('inner invoke')

}, 1000)

}, 1000)

console.log('global end')

输出：1.**global** begin

2.**global** **end**

3.timer2 invoke

4.timer1 invoke

5.inner invoke

# **# 所有异步编程方法的根基**

由调用者定义，交给执行者执行的函数

# **Promise**

commonJS社区提出Promise规范，在ES2015中被标准化，成为了语言规范。

**const** promise = **new** Promise(**function** (resolve, reject) {

*// 这里用于“兑现”承诺*

*// resolve(100) // 承诺达成*

reject(**new** Error('promise rejected')) *// 承诺失败*

})

promise.then(**function** (value) {

*// 即便没有异步操作，then 方法中传入的回调仍然会被放入队列，等待下一轮执行*

console.log('resolved', value)

}, **function** (error) {

console.log('rejected', error)

})

# **resolve**

描述： 静态方法 返回一个解析过的Promise对象

value 将被Promise对象解析的参数，也可以是一个Promise对象，或者是一个thenable。

返回一个带着给定值解析过的Promise对象，如果参数本身就是一个Promise对象，则直接返回这个Promise对象。

示例：**const** promise1 = Promise.resolve(123);

promise1.then((value) => {

console.log(value);

*// expected output: 123*

})

# **catch**

描述：返回一个Promise，并且处理拒绝的情况

参数

onRejected

当Promise 被rejected时,被调用的一个Function。 该函数拥有一个参数：

reason rejection 的原因。

如果 onRejected抛出一个错误或返回一个本身失败的 Promise ， 通过 catch() 返回的Promise 被rejected；否则，它将显示为成功（resolved）。

返回值 一个Promise.

# **all**

通用：在启动多个异步任务并发运行并为其结果创建承诺之后使用，以便人们可以等待所有任务完成。

参数

iterable

一个可迭代对象，如 Array 或 String。

返回值

如果传入的参数是一个空的可迭代对象，则返回一个已完成（already resolved）状态的 Promise。

如果传入的参数不包含任何 promise，则返回一个异步完成（asynchronously resolved） Promise。注意：Google Chrome 58 在这种情况下返回一个已完成（already resolved）状态的 Promise。

其它情况下返回一个处理中（pending）的Promise。这个返回的 promise 之后会在所有的 promise 都完成或有一个 promise 失败时异步地变为完成或失败。 见下方关于“Promise.all 的异步或同步”示例。返回值将会按照参数内的 promise 顺序排列，而不是由调用 promise 的完成顺序决定。

# **race(iterable)**

方法返回一个promise，一旦迭代器中的某个promise解决或拒绝，返回的promise就会解决或拒绝。

# **TypeScript**

* 解决了JavaScript类型系统的问题
* 大大提高了代码的可靠程度

类型系统

强类型：语言层面限制函数的实参类型，必须与形参的类型相同，不润许任意的意识类型转换

优势

1. 错误更早暴露
2. 代码更智能，编码更准确
3. 重构更牢靠
4. 减少不必要的类型判断

弱类型：不会限制实参的类型

类型检查

静态类型：一个变量声明是他的类型就是明确的，生命过后类型不可修改

动态类型：运行阶段才能后明确变量的类型，变化的类型可随时改变

JavaScript:弱类型 静态类型语言 没有编译的环节

# **Flow**

安装 flow-bin

添加 //@flow 在js

移除注解

安装 flow-remove-types

node flow-remove-types 源代码所在目录 -d 目标目录

babel

add @babel/core @babel/cli

开发插件

vscode 下载插件：Flow Language Support

类型推断：可自己推断类型

类型注解**function** **sum** (a: string): **number**{

**return** a

}

数组类型**const** arr1: Array<number> = [1, 2, 3]**const** arr2: number[] = [1, 2, 3]*// 元组***const** foo: [string, number] = ['foo', 100]

对象类型**const** obj1: { foo: string, bar: number } = { foo: 'string', bar: 100 }**const** obj2: { foo?: string, bar: number } = { bar: 100 }**const** obj3: { [string]: string } = {}

obj3.key1 = 'value1'

obj3.key2 = 'value2'

函数类型**function** **foo** (callback: (string, number) => void) {

callback('string', 100)

}

特殊类型

字面量类型**const** a: 'foo' = 'foo'**const** **type**: 'success' | 'warning' | 'danger' = 'success'

*// 声明类型***type** StringOrNumber = string | number**const** b: StringOrNumber = 'string' *// 100*

*// Maybe 类型***const** gender: ?number = undefined*// 相当于// const gender: number | null | void = undefined*

Mixed 强类型 可以传入任意类型的参数

Any 弱类型 兼容老代码 可以传入任意类型的参数

# **TypeScript**

1. 任何一种JavaScript运行环境都支持
2. 功能更为强大，生态也更健全、更完善。 缺点：
3. 语言本身多了很多概念 提高学习成本
4. 项目初期，ts会增加一些成本

使用

1. add typescript --dev
2. tsc xx.ts
3. 配置文件 tsc --init

原始数据类型**const** a: string = 'foobar'**const** b: number = 100 *// NaN Infinity***const** c: boolean = true *// false// 在非严格模式（strictNullChecks）下，// string, number, boolean 都可以为空// const d: string = null// const d: number = null// const d: boolean = null***const** e: void = undefined**const** f: null = null**const** g: undefined = undefined*// Symbol 是 ES2015 标准中定义的成员，// 使用它的前提是必须确保有对应的 ES2015 标准库引用// 也就是 tsconfig.json 中的 lib 选项必须包含 ES2015***const** h: symbol = Symbol()

*// Promise// const error: string = 100*

标准库声明

内置对象对应的声明文件

作用域问题 // 默认文件中的成员会作为全局成员 // 多个文件中有相同成员就会出现冲突 // const a = 123 // 解决办法1: IIFE 提供独立作用域 // (function () { // const a = 123 // })()

*// 解决办法2: 在当前文件使用 export，也就是把当前文件变成一个模块// 模块有单独的作用域***const** a = 123

export {}

object 类型

object 类型是指除了原始类型以外的其它类型

const foo: object = function(){}\\[]\\{}

*// 如果需要明确限制对象类型，则应该使用这种类型对象字面量的语法，或者是「接口」*

const obj: { foo: number, bar: string } = { foo: 123, bar: 'string' }

数组类型

数组类型的两种表示方式**const** arr1: Array<number> = [1, 2, 3]**const** arr2: number[] = [1, 2, 3]*// 案例 ----------------------// 如果是 JS，需要判断是不是每个成员都是数字// 使用 TS，类型有保障，不用添加类型判断***function** **sum** (...args: number[]) {

**return** args.reduce((prev, current) => prev + current, 0)

}

sum(1, 2, 3) *// => 6*

元组 Tuple

明确元素数量，每个元素类型的数组

const tuple: [number, string] = [18, 'zce']

const entries: [string, number][] = Object.entries({

foo: 123,

bar: 456

})

枚举类型 Enum

1. 可以给一组数值分别取上比较好的名字
2. 一个枚举中出现几个固定的值不会出现别的值 3.如果是数字枚举，不设置值，会自动累加

函数类型

1. 函数声明方式 声明函数

**function** **func1** (a: number, b: number = 10, ...rest: number[]): string {

**return** 'func1'

}

2. 函数表达式声明

**const** func2: (a: number, b: number) => string = **function** (a: number, b: number): string {

**return** 'func2'

}

任意类型 any

function stringify (value: any) {

return JSON.stringify(value)

}**stringify**('string')**stringify**(100)**stringify**(true)

let foo: any = 'string'

foo = 100

foo.bar()

隐式类型推断

1. 如果没有定义类型，会根据使用情况推断这个变量的类型

类型断言

使用 **as**

**const** num1 = res **as** number

使用<>

**const** num1 = <number>res

JSX下不能使用 会产生标签的冲突

接口 Interfaces

1. 约束对象的结构

**interface** Post {

title: string

content: string

}

**function** **printPost** (post: Post) {

console.log(post.title)

console.log(post.content)

}

printPost({

title: 'Hello TypeScript',

content: 'A javascript superset'

})

2.可选成员**interface** Post {

subtitle?: string 可选成员

}

3.只读成员**interface** Post {

readonly summary: string 只读不可修改

}4.动态成员例如缓存对象，**interface** Cache {

[prop: string]: string

}**const** cache: Cache = {}

cache.foo = 'value1'

cache.bar = 'value2'

类 class

描述一类具体事务的抽象特称

类的属性必须要有初始值**class** Person {

name: string *// = 'init name'*

age: number

**constructor** (name: string, age: number) {

**this**.name = name

**this**.age = age

}

sayHi (msg: string): void {

console.log(`I am ${this.name}, ${msg}`)

}

}

类的访问修改符

**public**：共有属性 默认就是**publicprivate**：私有属性 只有内部才能访问**protected**：受保护属性 外部不能访问 子成员可以访允许继承

**class** Person {

**public** name: string *// = 'init name'*

**private** age: number

**protected** gender: boolean

**constructor** (name: string, age: number) {

**this**.name = name

**this**.age = age

**this**.gender = true

}

sayHi (msg: string): void {

console.log(`I am ${this.name}, ${msg}`)

console.log(**this**.age)

}

}

**class** Student **extends** Person {

**private** **constructor** (name: string, age: number) {

**super**(name, age)

console.log(**this**.gender)

}

**static** create (name: string, age: number) {

**return** **new** Student(name, age)

}

}

**const** tom = **new** Person('tom', 18)console.log(tom.name)**const** jack = Student.create('jack', 18)

只读 readonly

类与接口 不同的类型相同的接口

一个接口约束一个类

interface Eat {

eat (food: string): void

}

interface Run {

run (distance: number): void

}

class Person implements Eat, Run {

eat (food: string): void {

console.log(`优雅的进餐: ${food}`)

}

run (distance: number) {

console.log(`直立行走: ${distance}`)

}

}

class Animal implements Eat, Run {

eat (food: string): void {

console.log(`呼噜呼噜的吃: ${food}`)

}

run (distance: number) {

console.log(`爬行: ${distance}`)

}

}

抽象类

1. 约束子类当中必须要有的成员
2. 比较大的类 最好使用抽象类
3. 使用 abstract 修饰

**abstract** **class** Animal {

eat (food: string): void {

console.log(`呼噜呼噜的吃: ${food}`)

}

**abstract** run (distance: number): void

}

**class** Dog **extends** Animal {

run(distance: number): void {

console.log('四脚爬行', distance)

}

}

**const** d = **new** Dog()

d.eat('嗯西马')

d.run(100)

泛型 Generics

1. 定义函数接口或者类型的时候没有指定类型，使用的时候传递参数

**function** createArray<T> (length: **number**, **value**: T): T[] {

const arr = Array<T>(length).fill(**value**)

**return** arr

}

类型声明

成员在定义的时候没有声明类型 在使用的时候再明确的做出类型说明

1. 使用 declare

**import** { camelCase } **from** 'lodash'

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