# 一，变量绑定

1.var的作用域是function，会悬挂；let的作用域是块{}，不会悬挂。

2.const定义常量，不可修改。所以定义时必须初始化。否则报错.

3.总结：

var 作用域：function 会悬挂

let 作用域：块{} 不悬挂

const 作用域：块{} 不悬挂

4.和其他语言不同，如果const是一个object，那么它是可变的。（why design like this？）

5.typeof是个相对安全的命令，即使值未定义，它会返回undefinded，而不会报错。但是，在使用let或者const的时候，在它们的作用域块中，会存在一个TDZ（Temporal Dead Zone），TDZ保证在let或者const语句未运行之前，变量值不可以被调用。在变量作用域块中，变量声明语句之前，都是TDZ的作用范围，在TDZ中，即使使用安全的typeof指令，也会报错。

6.js经典的循环问题，通过let可以直接解决，问题如下：

var funcs = [];for (var i = 0; i < 10; i++) { funcs.push(function() { console.log(i); });}funcs.forEach(function(func) { func(); *// outputs the number "10" ten times*});

7.const用在for里会报错：

for (const i = 0; i < 10; i++)

但是用在for-in或者for-of里不会报错：

for (const key in object)

因为for循环新的值会赋值给原来的变量导致错误，而for-in每次都新建一个变量，所以不会报错

8.不像var，直接使用let或者const定义变量不会影响到全局作用域（window）。

let history=1;

console.log(window.history === histort); //false

9.最佳实践：通常用const，需要变的用let，基本不用var。

# 二，字符串与正则表达式

10.js的string编码是16位的，汉字之类的文字编码已经超出了16位，所以汉字会占用两个字符，如下：

var text = "??";console.log(text.length);  *//* 2console.log(/^.$/.test(text));  *// fals*econsole.log(text.charAt(0));  *// "*"console.log(text.charAt(1));  *// "*"console.log(text.charCodeAt(0));  *// 5536*2console.log(text.charCodeAt(1));  *// 5727*1

es6新增了一个string的方法， codePointAt()，可以返回32位字符的编码：

var text = "??a";console.log(text.charCodeAt(0));  *// 5536*2console.log(text.charCodeAt(1));  *// 5727*1console.log(text.charCodeAt(2));  *// 9*7console.log(text.codePointAt(0));  *// 13407*1console.log(text.codePointAt(1));  *// 5727*1console.log(text.codePointAt(2));  *// 9*7

通过这个方法可以判断字符是否是32位的：

function is32Bit(c) { return c.codePointAt(0) > 0xFFFF;}console.log(is32Bit("??"));  *// tru*econsole.log(is32Bit("a"));

11.新增string方法，与codePointAt()相反：String.fromCodePoint()

console.log(String.fromCodePoint(134071)); *// "??*"

text.charCodeAt（）在es5里也有相反的方法 String.fromCharCode()

12.新增string方法，normalize(),有些字符意义是相同的，但是编码不同，比如“æ” and “ae”。通过这个方法可以让他们相同，在排序等场景有用。在做国际化的时候可能会用到这个方法，平时不怎么用，这里不详细写了，用到了看原文吧。

13.es6正则表达式新增U标志，用来匹配32位的字符。

var text = "??";console.log(text.length);  *//* 2console.log(/^.$/.test(text));  *// fals*econsole.log(/^.$/u.test(text)); // true

14.es6并没有提供针对32位字符串的字符个数统计方法，现在只能用以下方法模拟，不过这个方法效率不高，如果字符串很长的话就不要用了：

function codePointLength(text) { var result = text.match(/[\s\S]/gu); return result ? result.length : 0;}console.log(codePointLength("abc")); *// 3*console.log(codePointLength("??bc"));  *//* 3

15.检测js环境是否支持u标志，在老浏览器里u标志会报错，可以用以下方法检测浏览器是否支持u标志。

function hasRegExpU() { try { var pattern = new RegExp(".", "u"); return true; } catch (ex) { return false; }}

16.es5时使用indexOf()来判断字符串中是否包含特定片段，es6新增了三个方法来做判断：

var msg = "Hello world!";console.log(msg.startsWith("Hello")); *// true*console.log(msg.endsWith("!")); *// true*console.log(msg.includes("o")); *// true*console.log(msg.startsWith("o")); *// false*console.log(msg.endsWith("world!")); *// true*console.log(msg.includes("x")); *// false*console.log(msg.startsWith("o", 4)); *// true*console.log(msg.endsWith("o", 8)); *// true*console.log(msg.includes("o", 8)); *// false*

这三个方法只能返回布尔值，想要特定片段的位置信息还得用indexOf() and lastIndexOf()，而且这三个方法里不能放正则表达式，index是可以的。

16.The repeat() Method，es6新增了字符串方法---重复。

console.log("x".repeat(3)); *// "xxx"*console.log("hello".repeat(2)); *// "hellohello"*console.log("abc".repeat(4)); *// "abcabcabcabc"*

17. template literals，多行字符串，用``号（esc下面那个键），字符串中换行以及空格都会保留。

let message = `Multiline string`;console.log(message); *// "Multiline* *// string"*console.log(message.length); *// 31*

18.template literals 之中可以运行 js。${这里是js部分}

let count = 10, price = 0.25, message = `${count} items cost $${(count \* price).toFixed(2)}.`;console.log(message); *// "10 items cost $2.50."*

19.Raw Values in Template Literals. 存储解析前的原始字符串

let message1 = `Multiline\nstring`, message2 = String.raw`Multiline\nstring`;console.log(message1); *// "Multiline* *// string"*console.log(message2); *// "Multiline\\nstring"*

# 三、Functions

20.方法参数的默认值，es6里可以在可选参数后加默认值，一般可选参数都放在最后，但也可以放在前面，调用的时候传个undefined就可以启用默认值，这里注意，传null不能启用默认值。

function makeRequest(url, timeout = 2000, callback) { *// the rest of the function*}

21.方法默认值对arguments对象的影响，在es6中，不管是不是严格模式，arguments对象的值是不会随着参数的变化而变化的（es5的非严格模式会变化），所以可以用arguments对象找到参数的初始值。

*// not in strict mode*function mixArgs(first, second = "b") { console.log(arguments.length); console.log(first === arguments[0]); console.log(second === arguments[1]); first = "c"; second = "d" console.log(first === arguments[0]); console.log(second === arguments[1]);}mixArgs("a");

22.方法的默认参数可以通过一个方法返回，这个方法还可以利用前面的参数值（不能用后面的，有TDZ）。

function getValue(value) { return value + 5;}function add(first, second = getValue(first)) { return first + second;}console.log(add(1, 1)); *// 2*console.log(add(1)); *// 7*

33.多余的参数，在es5中，可以用arguments对象来获取多余的参数，但是这样并不直观，es6提供了新的方法。用...name的写法来表示多个参数，多余的参数会被放在一个数组中，用name调用。多余参数的限制：只能有一个，而且只能放最后。

function pick(object, ...keys) { let result = Object.create(null); for (let i = 0, len = keys.length; i < len; i++) { result[keys[i]] = object[keys[i]]; } return result;}

34.多余参数对arguments对象的影响（多余参数是用来替换arguments的，尽管es6中还有arguments）

function checkArgs(...args) { console.log(args.length); console.log(arguments.length); console.log(args[0], arguments[0]); console.log(args[1], arguments[1]);}checkArgs("a", "b");

The call to checkArgs() outputs:

2 2 a a b b

35. spread operator,是rest parameters的相反操作，它可以在调用函数的时候把数组拆分成单个的项。

**let** values = [-25, -50, -75, -100]

console.log(Math.max(...values, 0)); *// 0*

36.由于匿名function在调试的时候会带来一些困难，es6给所有的function加了一个name属性。

**function** doSomething() {

*// ...*

}

**var** doAnotherThing = **function**() {

*// ...*

};

console.log(doSomething.name); *// "doSomething"*

console.log(doAnotherThing.name); *// "doAnotherThing"*

37.让构造函数只能通过new来使用（new.target）

**function** Person(name) {

**if** (**typeof** **new**.target !== "undefined") {

**this**.name = name; *// using new*

} **else** {

**throw** **new** Error("You must use new with Person.")

}

}

**var** person = **new** Person("Nicholas");

**var** notAPerson = Person.call(person, "Michael"); *// error!*

或者限定构造函数：

**function** Person(name) {

**if** (**new**.target === Person) {

**this**.name = name; *// using new*

} **else** {

**throw** **new** Error("You must use new with Person.")

}

}

**function** AnotherPerson(name) {

Person.call(**this**, name);

}

**var** person = **new** Person("Nicholas");

**var** anotherPerson = **new** AnotherPerson("Nicholas"); *// error!*

new.target出现在function外面会导致语法错误

38：块级方法

Es5严格模式中，块级方法是会报错的

"use strict";

**if** (**true**) {

*// Throws a syntax error in ES5, not so in ES6*

**function** doSomething() {

*// ...*

}

}

Es6严格模式 允许块级方法：

"use strict";

**if** (**true**) {

console.log(**typeof** doSomething); *// "function"*

**function** doSomething() {

*// ...*

}

doSomething();

}

console.log(**typeof** doSomething); *// "undefined"*

在块内依然会存在变量提升，块外方法不存在

块级方法也可以用let来写，不过这样写会存在TDZ

"use strict";

**if** (**true**) {

console.log(**typeof** doSomething); *// throws error*

**let** doSomething = **function** () {

*// ...*

}

doSomething();

}

console.log(**typeof** doSomething);

在es6的非严格模式中，块级方法会被提升至全局

*// ECMAScript 6 behavior*

**if** (**true**) {

console.log(**typeof** doSomething); *// "function"*

**function** doSomething() {

*// ...*

}

doSomething();

}

console.log(**typeof** doSomething); *// "function"*

39.箭头函数

No this, super, arguments, and new.target bindings

Cannot be called with new

No prototype

Can’t change this

No arguments object

No duplicate named parameters

40.尾部函数调用优化

迭代的函数可以考虑使用，不常用，详细见原著

# 四、Expanded Object Functionality

41.属性初始化简写

{name:name},可以简化为{name}

42.对象方法简写

Es5写法：

**var** person = {

name: "Nicholas",

sayName: **function**() {

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

}

};

Es6 新写法：

**var** person = {

name: "Nicholas",

sayName() {

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

}

};

42.动态属性名

**var** suffix = " name";

**var** person = {

["first" + suffix]: "Nicholas",

["last" + suffix]: "Zakas"

};

console.log(person["first name"]); *// "Nicholas"*

console.log(person["last name"]); *// "Zakas"*

43.新方法Object.is()

console.log(+0 == -0); *// true*

console.log(+0 === -0); *// true*

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

console.log(**NaN** == **NaN**); *// false*

console.log(**NaN** === **NaN**); *// false*

console.log(Object.is(**NaN**, **NaN**)); *// true*

console.log(5 == 5); *// true*

console.log(5 == "5"); *// true*

console.log(5 === 5); *// true*

console.log(5 === "5"); *// false*

console.log(Object.is(5, 5)); *// true*

console.log(Object.is(5, "5")); *// false*

除了+0 -0 和 NAN之外，其他行为和===相同

44.新方法Object.assign()

用于合并对象

无法复制accessor properties，因为内部为赋值操作

**var** receiver = {},

supplier = {

get name() {

**return** "file.js"

}

};

Object.assign(receiver, supplier);

**var** descriptor = Object.getOwnPropertyDescriptor(receiver, "name");

console.log(descriptor.value); *// "file.js"*

console.log(descriptor.get); *// undefined*

45.重复对象字面量属性

在es5严格模式中，在字面量中重复声明属性会报错。Es6中两个模式都不会报错。

"use strict";

**var** person = {

name: "Nicholas",

name: "Greg" *// syntax error in ES5 strict mode*

};

46.自有属性枚举顺序

The basic order for own property enumeration is:

1. All numeric keys in ascending order
2. All string keys in the order in which they were added to the object
3. All symbol keys (covered in Chapter 6) in the order in which they were added to the object
4. **var** obj = {
5. a: 1,
6. 0: 1,
7. c: 1,
8. 2: 1,
9. b: 1,
10. 1: 1
11. };
12. obj.d = 1;
13. console.log(Object.getOwnPropertyNames(obj).join("")); *// "012acbd"*

es5之中枚举顺序是javascript决定的，不同的环境下实现的方式不同。为了统一。Es6中规定了排序规则。数字按照大小排序，并前置。字符串按照添加顺序排在后面。

47.改变对象原型Object.setPrototypeOf()

**let** person = {

getGreeting() {

**return** "Hello";

}

};

**let** dog = {

getGreeting() {

**return** "Woof";

}

};

*// prototype is person*

**let** friend = Object.create(person);

console.log(friend.getGreeting()); *// "Hello"*

console.log(Object.getPrototypeOf(friend) === person); *// true*

*// set prototype to dog*

Object.setPrototypeOf(friend, dog);

console.log(friend.getGreeting()); *// "Woof"*

console.log(Object.getPrototypeOf(friend) === dog); *// true*

48，引进super （like Java）

Es5中是这样模拟的：

Object.getPrototypeOf(this).getGreeting.call(this)

但是继承层数多了会出问题

Es6新语法这样写：

**let** friend = {

getGreeting() {

*// in the previous example, this is the same as:*

*// Object.getPrototypeOf(this).getGreeting.call(this)*

**return** **super**.getGreeting() + ", hi!";

}

};

注意super只能出现在简写的方法里，不然会报错：如下

**let** friend = {

getGreeting: **function**() {

*// syntax error*

**return** **super**.getGreeting() + ", hi!";

}

};

# 五，解构

49，对象解构

变量声明：

**let** node = {

type: "Identifier",

name: "foo"

};

**let** { type, name } = node;

console.log(type); *// "Identifier"*

console.log(name); *// "foo"*

变量修改：（注意这里的括号是必需的）

**let** node = {

type: "Identifier",

name: "foo"

},

type = "Literal",

name = 5;

*// assign different values using destructuring*

({ type, name } = node);

console.log(type); *// "Identifier"*

console.log(name); *// "foo"*

设置默认值：

**let** node = {

type: "Identifier",

name: "foo"

};

**let** { type, name, value = **true** } = node;

console.log(type); *// "Identifier"*

console.log(name); *// "foo"*

console.log(value); *// true*

设置新的变量名：

**let** node = {

type: "Identifier"

};

**let** { type: localType, name: localName = "bar" } = node;

console.log(localType); *// "Identifier"*

console.log(localName); *// "bar"*

解构嵌套对象：

**let** node = {

type: "Identifier",

name: "foo",

loc: {

start: {

line: 1,

column: 1

},

end: {

line: 1,

column: 4

}

}

};

*// extract node.loc.start*

**let** { loc: { start: localStart }} = node;

console.log(localStart.line); *// 1*

console.log(localStart.column); *// 1*

50. 数组解构

变量声明：

**let** colors = [ "red", "green", "blue" ];

**let** [ , , thirdColor ] = colors;

console.log(thirdColor); *// "blue"*

变量修改：

**let** colors = [ "red", "green", "blue" ],

firstColor = "black",

secondColor = "purple";

[ firstColor, secondColor ] = colors;

console.log(firstColor); *// "red"*

console.log(secondColor); *// "green"*

注意 这里不用加括号

可以用于调换两个值

*// Swapping variables in ECMAScript 6*

**let** a = 1,

b = 2;

[ a, b ] = [ b, a ];

console.log(a); *// 2*

console.log(b); *// 1*

设置默认值：

**let** colors = [ "red" ];

**let** [ firstColor, secondColor = "green" ] = colors;

console.log(firstColor); *// "red"*

console.log(secondColor); *// "green"*

解构嵌套数组：

**let** colors = [ "red", [ "green", "lightgreen" ], "blue" ];

*// later*

**let** [ firstColor, [ secondColor ] ] = colors;

console.log(firstColor); *// "red"*

console.log(secondColor); *// "green"*

解构剩余参数：

**let** colors = [ "red", "green", "blue" ];

**let** [ firstColor, ...restColors ] = colors;

console.log(firstColor); *// "red"*

console.log(restColors.length); *// 2*

console.log(restColors[0]); *// "green"*

console.log(restColors[1]); *// "blue"*

51.复制数组的两个方法

Js里没有专门用来复制数组的方法，这俩方法都是曲线救国

*// cloning an array in ECMAScript 5*

**var** colors = [ "red", "green", "blue" ];

**var** clonedColors = colors.concat();

console.log(clonedColors); *//"[red,green,blue]"*

*// cloning an array in ECMAScript 6*

**let** colors = [ "red", "green", "blue" ];

**let** [ ...clonedColors ] = colors;

console.log(clonedColors); *//"[red,green,blue]"*

…必须是最后一项，且后面不能有逗号

52.混合解构

**let** node = {

type: "Identifier",

name: "foo",

loc: {

start: {

line: 1,

column: 1

},

end: {

line: 1,

column: 4

}

},

range: [0, 3]

};

**let** {

loc: { start },

range: [ startIndex ]

} = node;

console.log(start.line); *// 1*

console.log(start.column); *// 1*

console.log(startIndex); *// 0*

53.解构参数

当函数具有多个可选参数时建议使用

**function** setCookie(name, value, { secure, path, domain, expires } = {}) {

*// code to set the cookie*

}

setCookie("type", "js", {

secure: **true**,

expires: 60000

});

# 六、Symbols

54.创建symbols

**let** firstName = Symbol("first name");

**let** person = {};

person[firstName] = "Nicholas";

console.log("first name" **in** person); *// false*

console.log(person[firstName]); *// "Nicholas"*

console.log(firstName); *// "Symbol(first name)"*

55.symbols可以用作计算属性的名字

**let** firstName = Symbol("first name");

*// use a computed object literal property*

**let** person = {

[firstName]: "Nicholas"

};

*// make the property read only*

Object.defineProperty(person, firstName, { writable: **false** });

**let** lastName = Symbol("last name");

Object.defineProperties(person, {

[lastName]: {

value: "Zakas",

writable: **false**

}

});

console.log(person[firstName]); *// "Nicholas"*

console.log(person[lastName]); *// "Zakas"*

56.共享symbols Symbol.for()

Symbols可以被全局共享，使用Symbol.for()创建symbol，如果这个symbol已经被注册，则返回值。如果没有注册，则先注册在返回值。

**let** uid = Symbol.**for**("uid");

**let** object = {

[uid]: "12345"

};

console.log(object[uid]); *// "12345"*

console.log(uid); *// "Symbol(uid)"*

**let** uid2 = Symbol.**for**("uid");

console.log(uid === uid2); *// true*

console.log(object[uid2]); *// "12345"*

console.log(uid2); *// "Symbol(uid)"*

利用Symbol.keyFor()可以通过symbol反差注册字符串，可以利用这个方法避免冲突：

**let** uid = Symbol.**for**("uid");

console.log(Symbol.keyFor(uid)); *// "uid"*

**let** uid2 = Symbol.**for**("uid");

console.log(Symbol.keyFor(uid2)); *// "uid"*

**let** uid3 = Symbol("uid");

console.log(Symbol.keyFor(uid3)); *// undefined*

57.symbols 类型强制转换

用String()、toString()可以将symbol转为字符串：

**let** uid = Symbol.**for**("uid"),

desc = String(uid);

console.log(desc); *// "Symbol(uid)"*

但是直接串联字符串会报错：

**let** uid = Symbol.**for**("uid"),

desc = uid + ""; *// error!*

尝试将symbol转化为数字也会报错：

**let** uid = Symbol.**for**("uid"),

sum = uid / 1; *// error!*

将symbol转化Boolean是正常的true

58.获取symbol对象属性Object.getOwnPropertySymbols();

使用Object.keys()可以获取所有可枚举对象属性；

使用Object.getOwnPropertyNames()可以获取所有对象属性无论可不可枚举；

但是这两个方法都不会返回symbol属性，es6中添加了一个方法专门返回symbol属性：

**let** uid = Symbol.**for**("uid");

**let** object = {

[uid]: "12345"

};

**let** symbols = Object.getOwnPropertySymbols(object);

console.log(symbols.length); *// 1*

console.log(symbols[0]); *// "Symbol(uid)"*

console.log(object[symbols[0]]); *// "12345"*

59，symbol的其他诡异用法

The well-known symbols are:

* Symbol.hasInstance - A method used by instanceof to determine an object’s inheritance.
* Symbol.isConcatSpreadable - A Boolean value indicating that Array.prototype.concat() should flatten the collection’s elements if the collection is passed as a parameter to Array.prototype.concat().
* Symbol.iterator - A method that returns an iterator. (Iterators are covered in Chapter 7.)
* Symbol.match - A method used by String.prototype.match() to compare strings.
* Symbol.replace - A method used by String.prototype.replace() to replace substrings.
* Symbol.search - A method used by String.prototype.search() to locate substrings.
* Symbol.species - The constructor for making derived objects. (Derived objects are covered in Chapter 8.)
* Symbol.split - A method used by String.prototype.split() to split up strings.
* Symbol.toPrimitive - A method that returns a primitive value representation of an object.
* Symbol.toStringTag - A string used by Object.prototype.toString() to create an object description.
* Symbol.unscopables - An object whose properties are the names of object properties that should not be included in a with statement.

这里不详细写了，具体见原著

# 七、Sets and Maps

60.set 创建、添加、大小、检测

**let** set = **new** Set();

set.add(5);

set.add("5");

console.log(set.size); *// 2*

多次重复添加，后面添加的会被忽略掉

**let** set = **new** Set();

set.add(5);

set.add("5");

set.add(5); *// duplicate - this is ignored*

console.log(set.size); *// 2*

可以利用数组初始化set，重复项会被自动删除：

**let** set = **new** Set([1, 2, 3, 4, 5, 5, 5, 5]);

console.log(set.size); *// 5*

可以利用has检测set是否存在某个值：

**let** set = **new** Set();

set.add(5);

set.add("5");

console.log(set.has(5)); *// true*

console.log(set.has(6)); *// false*

61.set移除值

delete() 删除某个值；

clear() 清空set

**let** set = **new** Set();

set.add(5);

set.add("5");

console.log(set.has(5)); *// true*

set.**delete**(5);

console.log(set.has(5)); *// false*

console.log(set.size); *// 1*

set.clear();

console.log(set.has("5")); *// false*

console.log(set.size); *// 0*

62.set的forEach方法

**let** set = **new** Set([1, 2]);

set.forEach(**function**(value, key, ownerSet) {

console.log(key + " " + value);

console.log(ownerSet === set);

});

1 1

true

2 2

true

和array和map的方法差不多，区别是他的前两个参数是一样的，为了和array与map保持一致。像array一样，也可以传入第二个参数作为this

63.将set转换为array

Array转set前面说过了，直接传入构造函数即可。

Set转array，使用…方法：

**let** set = **new** Set([1, 2, 3, 3, 3, 4, 5]),

array = [...set];

console.log(array); *// [1,2,3,4,5]*

配合前面的方法，可以实现去除数组重复项：

**function** eliminateDuplicates(items) {

**return** [...**new** Set(items)];

}

**let** numbers = [1, 2, 3, 3, 3, 4, 5],

noDuplicates = eliminateDuplicates(numbers);

console.log(noDuplicates); *// [1,2,3,4,5]*

64. Weak Sets

在set里存放原始变量一切ok，但是当把对象放进set里面的时候，set会保留这个对象的引用导致垃圾回收无法进行。所以如果要放一堆对象到set里，就使用weak set。

Strong set阻塞垃圾回收：

**let** set = **new** Set(),

key = {};

set.add(key);

console.log(set.size); *// 1*

*// eliminate original reference*

key = **null**;

console.log(set.size); *// 1*

*// get the original reference back*

key = [...set][0];

weak set不阻塞垃圾回收：

**let** set = **new** WeakSet(),

key = {};

*// add the object to the set*

set.add(key);

console.log(set.has(key)); *// true*

*// remove the last strong reference to key, also removes from weak set*

key = **null**;

不同点：

1. Weak set只能存储对象，add原始对象会报错
2. 没有迭代器，不能用于for – of，forEach，无法动态监测内容
3. 没有size属性，无法检测大小。

65,map的创建、添加、取回

**let** map = **new** Map();

map.set("title", "Understanding ES6");

map.set("year", 2016);

console.log(map.get("title")); *// "Understanding ES6"*

console.log(map.get("year")); *// 2016*

map可以使用对象做键值：

**let** map = **new** Map(),

key1 = {},

key2 = {};

map.set(key1, 5);

map.set(key2, 42);

console.log(map.get(key1)); *// 5*

console.log(map.get(key2)); *// 42*

66，map的方法

为了故意和set保持一致，map的很多方法和set是一样的

* has(key) - Determines if the given key exists in the map
* delete(key) - Removes the key and its associated value from the map
* clear() - Removes all keys and values from the map
* size() – Retrun the length of the map
* forEach()

67.map初始化

**let** map = **new** Map([["name", "Nicholas"], ["age", 25]]);

console.log(map.has("name")); *// true*

console.log(map.get("name")); *// "Nicholas"*

console.log(map.has("age")); *// true*

console.log(map.get("age")); *// 25*

console.log(map.size); *// 2*

68，weak maps

创建原因和set一样，键只能是对象，不能迭代。

Weak map只有两个方法，has和delete。Size、clear、forEach全都不能用，因为无法迭代。

**let** map = **new** WeakMap(),

element = document.querySelector(".element");

map.set(element, "Original");

**let** value = map.get(element);

console.log(value); *// "Original"*

*// remove the element*

element.parentNode.removeChild(element);

element = **null**;

*// the weak map is empty at this point*

69.weak maps在私有对象数据中的应用

我们用闭包创建私有变量的时候，如果变量是个对象，那么这个对象永远不会被释放：

**var** Person = (**function**() {

**var** privateData = {},

privateId = 0;

**function** Person(name) {

Object.defineProperty(**this**, "\_id", { value: privateId++ });

privateData[**this**.\_id] = {

name: name

};

}

Person.prototype.getName = **function**() {

**return** privateData[**this**.\_id].name;

};

**return** Person;

}());

在上述程序中，即使person被摧毁，privateData依然存在于内存之中。

我们可以使用weak map来解决这个问题：

**let** Person = (**function**() {

**let** privateData = **new** WeakMap();

**function** Person(name) {

privateData.set(**this**, { name: name });

}

Person.prototype.getName = **function**() {

**return** privateData.get(**this**).name;

};

**return** Person;

}());

# 八、Iterators and Generators

70.es5模拟迭代器

**function** createIterator(items) {

**var** i = 0;

**return** {

next: **function**() {

**var** done = (i >= items.length);

**var** value = !done ? items[i++] : **undefined**;

**return** {

done: done,

value: value

};

}

};

}

**var** iterator = createIterator([1, 2, 3]);

console.log(iterator.next()); *// "{ value: 1, done: false }"*

console.log(iterator.next()); *// "{ value: 2, done: false }"*

console.log(iterator.next()); *// "{ value: 3, done: false }"*

console.log(iterator.next()); *// "{ value: undefined, done: true }"*

*// for all further calls*

console.log(iterator.next()); *// "{ value: undefined, done: true }"*

71.生成器

生成器可以生成一个迭代器：

生成器在function之后加\*号

*// generator*

**function** \*createIterator() {

**yield** 1;

**yield** 2;

**yield** 3;

}

*// generators are called like regular functions but return an iterator*

**let** iterator = createIterator();

console.log(iterator.next().value); *// 1*

console.log(iterator.next().value); *// 2*

console.log(iterator.next().value); *// 3*

yield只能出现在生成器之中。即使出现在生成器的内部方法之中也会报错：

**function** \*createIterator(items) {

items.forEach(**function**(item) {

*// syntax error*

**yield** item + 1;

});

}

箭头函数不能创建生成器

72.For-of

推荐使用for-of来循环迭代器，它会内部调用迭代器的方法：

**let** values = [1, 2, 3];

**for** (**let** num **of** values) {

console.log(num);

}

73.获取默认迭代器 Symbol.iterator

通过Symbol.iterator可以获取到默认迭代器，for-of内部就是这样做的：

**let** values = [1, 2, 3];

**let** iterator = values[Symbol.iterator]();

console.log(iterator.next()); *// "{ value: 1, done: false }"*

console.log(iterator.next()); *// "{ value: 2, done: false }"*

console.log(iterator.next()); *// "{ value: 3, done: false }"*

console.log(iterator.next()); *// "{ value: undefined, done: true }"*

也可以通过它来检测对象是否可迭代：、

**function** isIterable(object) {

**return** **typeof** object[Symbol.iterator] === "function";

}

console.log(isIterable([1, 2, 3])); *// true*

console.log(isIterable("Hello")); *// true*

console.log(isIterable(**new** Map())); *// true*

console.log(isIterable(**new** Set())); *// true*

console.log(isIterable(**new** WeakMap())); *// false*

console.log(isIterable(**new** WeakSet())); *// false*

74.创建自定义迭代器

用户定义的对象默认是不可迭代的，但是可以通过创建Symbol.iterator属性来包含一个生成器。

**let** collection = {

items: [],

\*[Symbol.iterator]() {

**for** (**let** item **of** **this**.items) {

**yield** item;

}

}

};

collection.items.push(1);

collection.items.push(2);

collection.items.push(3);

**for** (**let** x **of** collection) {

console.log(x);

}

1

2

3

75.集合迭代器

Es6的集合，array、set、map拥有以下默认迭代器：

* entries() - Returns an iterator whose values are a key-value pair
* values() - Returns an iterator whose values are the values of the collection
* keys() - Returns an iterator whose values are the keys contained in the collection

以entries为例：

**let** colors = [ "red", "green", "blue" ];

**let** tracking = **new** Set([1234, 5678, 9012]);

**let** data = **new** Map();

data.set("title", "Understanding ECMAScript 6");

data.set("format", "ebook");

**for** (**let** key **of** colors.keys()) {

console.log(key);

}

**for** (**let** key **of** tracking.keys()) {

console.log(key);

}

**for** (**let** key **of** data.keys()) {

console.log(key);

}

0

1

2

1234

5678

9012

"title"

"format"

如果省略迭代器名称，则会启用默认迭代器，

Array和set的默认迭代器为value

Map的默认迭代器为entries：

**let** colors = [ "red", "green", "blue" ];

**let** tracking = **new** Set([1234, 5678, 9012]);

**let** data = **new** Map();

data.set("title", "Understanding ECMAScript 6");

data.set("format", "print");

*// same as using colors.values()*

**for** (**let** value **of** colors) {

console.log(value);

}

*// same as using tracking.values()*

**for** (**let** num **of** tracking) {

console.log(num);

}

*// same as using data.entries()*

**for** (**let** entry **of** data) {

console.log(entry);

}

"red"

"green"

"blue"

1234

5678

9012

["title", "Understanding ECMAScript 6"]

["format", "print"]

小提示：使用entries的时候可以配合解构使用：

**let** data = **new** Map();

data.set("title", "Understanding ECMAScript 6");

data.set("format", "ebook");

*// same as using data.entries()*

**for** (**let** [key, value] **of** data) {

console.log(key + "=" + value);

}

76.字符串迭代器

从es5之后，js的字符串就变得越来越像数组。Es5中可以使用中括号来获取字符串中的值 ,text[0],但是这种方法遇到双字节字符的时候就会出错。Es6解决了这个问题。

**var** message = "A ð ®· B";

**for** (**let** c **of** message) {

console.log(c);

}

77. Spread Operator

前面章节讲过可以用spread operator将set转化为array

array = [...set];

这里spread operator利用的也是迭代器，也就是说不光是set，只要拥有迭代器的对象都可以通过这种方法转成数组，甚至字符串也可以。

array = [...’一二三四’];

78.给迭代器传参数

**function** \*createIterator() {

**let** first = **yield** 1;

**let** second = **yield** first + 2; *// 4 + 2*

**yield** second + 3; *// 5 + 3*

}

**let** iterator = createIterator();

console.log(iterator.next()); *// "{ value: 1, done: false }"*

console.log(iterator.next(4)); *// "{ value: 6, done: false }"*

console.log(iterator.next(5)); *// "{ value: 8, done: false }"*

console.log(iterator.next()); *// "{ value: undefined, done: true }"*

next()执行到yield 1就停止了

next（4）的时候继续执行，将4作为返回值赋值给first，以此类推。

79.在迭代器中抛出错误

**function** \*createIterator() {

**let** first = **yield** 1;

**let** second = **yield** first + 2; *// yield 4 + 2, then throw*

**yield** second + 3; *// never is executed*

}

**let** iterator = createIterator();

console.log(iterator.next()); *// "{ value: 1, done: false }"*

console.log(iterator.next(4)); *// "{ value: 6, done: false }"*

console.log(iterator.**throw**(**new** Error("Boom"))); *// error thrown from generator*

上面的函数在给second返回值的时候返回了一个错误，可以在生成器中处理这个错误：

**function** \*createIterator() {

**let** first = **yield** 1;

**let** second;

**try** {

second = **yield** first + 2; *// yield 4 + 2, then throw*

} **catch** (ex) {

second = 6; *// on error, assign a different value*

}

**yield** second + 3;

}

**let** iterator = createIterator();

console.log(iterator.next()); *// "{ value: 1, done: false }"*

console.log(iterator.next(4)); *// "{ value: 6, done: false }"*

console.log(iterator.**throw**(**new** Error("Boom"))); *// "{ value: 9, done: false }"*

console.log(iterator.next()); *// "{ value: undefined, done:\*

**true** }"

80.在生成器中使用return语句

**function** \*createIterator() {

**yield** 1;

**return** 24;

**yield** 2;

**yield** 3;

}

**let** iterator = createIterator();

console.log(iterator.next()); *// "{ value: 1, done: false }"*

console.log(iterator.next()); *// "{ value: 24, done: true }"*

81. Delegating Generators

生成器中可以引用别的生成器：

**function** \*createNumberIterator() {

**yield** 1;

**yield** 2;

**return** 3;

}

**function** \*createRepeatingIterator(count) {

**for** (**let** i=0; i < count; i++) {

**yield** "repeat";

}

}

**function** \*createCombinedIterator() {

**let** result = **yield** \*createNumberIterator();

**yield** result;

**yield** \*createRepeatingIterator(result);

}

**var** iterator = createCombinedIterator();

console.log(iterator.next()); *// "{ value: 1, done: false }"*

console.log(iterator.next()); *// "{ value: 2, done: false }"*

console.log(iterator.next()); *// "{ value: 3, done: false }"*

console.log(iterator.next()); *// "{ value: "repeat", done: false }"*

console.log(iterator.next()); *// "{ value: "repeat", done: false }"*

console.log(iterator.next()); *// "{ value: "repeat", done: false }"*

console.log(iterator.next()); *// "{ value: undefined, done: true }"*

这里注意被引用的生成器之中的retrun不会被迭代出来，如果需要迭代，则必须在代码中写明。

提示：字符串也可以用在生成器中yield \* "hello"

82.异步任务运行

一个简单的异步任务运行器

**function** run(taskDef) {

*// create the iterator, make available elsewhere*

**let** task = taskDef();

*// start the task*

**let** result = task.next();

*// recursive function to keep calling next()*

**function** step() {

*// if there's more to do*

**if** (!result.done) {

result = task.next();

step();

}

}

*// start the process*

step();

}

这样使用：

run(**function**\*() {

console.log(1);

**yield**;

console.log(2);

**yield**;

console.log(3);

});

83.伴随参数传递的异步任务

**function** run(taskDef) {

*// create the iterator, make available elsewhere*

**let** task = taskDef();

*// start the task*

**let** result = task.next();

*// recursive function to keep calling next()*

**function** step() {

*// if there's more to do*

**if** (!result.done) {

result = task.next(result.value);

step();

}

}

*// start the process*

step();

}

这样使用：

run(**function**\*() {

**let** value = **yield** 1;

console.log(value); *// 1*

value = **yield** value + 3;

console.log(value); *// 4*

});

84.改进版异步任务运行器.

**function** run(taskDef) {

*// create the iterator, make available elsewhere*

**let** task = taskDef();

*// start the task*

**let** result = task.next();

*// recursive function to keep calling next()*

**function** step() {

*// if there's more to do*

**if** (!result.done) {

**if** (**typeof** result.value === "function") {

result.value(**function**(err, data) {

**if** (err) {

result = task.**throw**(err);

**return**;

}

result = task.next(data);

step();

});

} **else** {

result = task.next(result.value);

step();

}

}

}

*// start the process*

step();

}

异步方法：

**let** fs = require("fs");

**function** readFile(filename) {

**return** **function**(callback) {

fs.readFile(filename, callback);

};

}

以同步风格调用异步方法，不写callback

run(**function**\*() {

**let** contents = **yield** readFile("config.json");

doSomethingWith(contents);

console.log("Done");

});

# 九，js class

85.es5中类似class的写法

**function** PersonType(name) {

**this**.name = name;

}

PersonType.prototype.sayName = **function**() {

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

};

**let** person = **new** PersonType("Nicholas");

person.sayName(); *// outputs "Nicholas"*

console.log(person **instanceof** PersonType); *// true*

console.log(person **instanceof** Object); *// true*

86.基本的class声明

**class** PersonClass {

*// equivalent of the PersonType constructor*

constructor(name) {

**this**.name = name;

}

*// equivalent of PersonType.prototype.sayName*

sayName() {

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

}

}

**let** person = **new** PersonClass("Nicholas");

person.sayName(); *// outputs "Nicholas"*

console.log(person **instanceof** PersonClass); *// true*

console.log(person **instanceof** Object); *// true*

console.log(**typeof** PersonClass); *// "function"*

console.log(**typeof** PersonClass.prototype.sayName); *// "function"*

es6 中的class只是一个语法糖，其内部原理和上面的es5的方法是一样的。所以混合使用也没有问题

87.class和function的区别

1.class定义不提升，比较像let。

2.内部代码自动严格模式。

3.所有方法不可枚举。

4.所以方法不可以使用new调用，会报错。

5.class必须用new调用，不然报错。

6.在类内部覆盖类名报错。

用es5 语法可以完全模拟class：

*// direct equivalent of PersonClass*

**let** PersonType2 = (**function**() {

"use strict";

**const** PersonType2 = **function**(name) {

*// make sure the function was called with new*

**if** (**typeof** **new**.target === "undefined") {

**throw** **new** Error("Constructor must be called with new.");

}

**this**.name = name;

}

Object.defineProperty(PersonType2.prototype, "sayName", {

value: **function**() {

*// make sure the method wasn't called with new*

**if** (**typeof** **new**.target !== "undefined") {

**throw** **new** Error("Method cannot be called with new.");

}

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

},

enumerable: **false**,

writable: **true**,

configurable: **true**

});

**return** PersonType2;

}());

88.访问者属性

Class允许在原型上定义访问者属性：

**class** CustomHTMLElement {

constructor(element) {

**this**.element = element;

}

get html() {

**return** **this**.element.innerHTML;

}

set html(value) {

**this**.element.innerHTML = value;

}

}

**var** descriptor = Object.getOwnPropertyDescriptor(CustomHTMLElement.prototype,\

"html");

console.log("get" **in** descriptor); *// true*

console.log("set" **in** descriptor); *// true*

console.log(descriptor.enumerable); *// false*

89.计算属性名

**let** propertyName = "html";

**class** CustomHTMLElement {

constructor(element) {

**this**.element = element;

}

get [propertyName]() {

**return** **this**.element.innerHTML;

}

set [propertyName](value) {

**this**.element.innerHTML = value;

}

}

90.生成器方法

**class** MyClass {

\*createIterator() {

**yield** 1;

**yield** 2;

**yield** 3;

}

}

**let** instance = **new** MyClass();

**let** iterator = instance.createIterator();

可以这样创建生成器，但这样不是最优解。

**class** Collection {

constructor() {

**this**.items = [];

}

\*[Symbol.iterator]() {

**yield** \***this**.items.values();

}

}

**var** collection = **new** Collection();

collection.items.push(1);

collection.items.push(2);

collection.items.push(3);

**for** (**let** x **of** collection) {

console.log(x);

}

*// Output:*

*// 1*

*// 2*

*// 3*

这样写相对好一点，可以直接用来循环

91.静态成员

Es5中实现静态成员的方法：

**function** PersonType(name) {

**this**.name = name;

}

*// static method*

PersonType.create = **function**(name) {

**return** **new** PersonType(name);

};

*// instance method*

PersonType.prototype.sayName = **function**() {

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

};

**var** person = PersonType.create("Nicholas");

Es6的方法：

**class** PersonClass {

*// equivalent of the PersonType constructor*

constructor(name) {

**this**.name = name;

}

*// equivalent of PersonType.prototype.sayName*

sayName() {

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

}

*// equivalent of PersonType.create*

**static** create(name) {

**return** **new** PersonClass(name);

}

}

**let** person = PersonClass.create("Nicholas");

92.继承与子类

Es5中的写法：

**function** Rectangle(length, width) {

**this**.length = length;

**this**.width = width;

}

Rectangle.prototype.getArea = **function**() {

**return** **this**.length \* **this**.width;

};

**function** Square(length) {

Rectangle.call(**this**, length, length);

}

Square.prototype = Object.create(Rectangle.prototype, {

constructor: {

value:Square,

enumerable: **true**,

writable: **true**,

configurable: **true**

}

});

**var** square = **new** Square(3);

console.log(square.getArea()); *// 9*

console.log(square **instanceof** Square); *// true*

console.log(square **instanceof** Rectangle); *// true*

es6中的写法：

**class** Rectangle {

constructor(length, width) {

**this**.length = length;

**this**.width = width;

}

getArea() {

**return** **this**.length \* **this**.width;

}

}

**class** Square **extends** Rectangle {

constructor(length) {

*// same as Rectangle.call(this, length, length)*

**super**(length, length);

}

}

**var** square = **new** Square(3);

console.log(square.getArea()); *// 9*

console.log(square **instanceof** Square); *// true*

console.log(square **instanceof** Rectangle); *// true*

注意：如果不写明，子类会自动调用super。以下两种写法是相同的：

**class** Square **extends** Rectangle {

*// no constructor*

}

*// Is equivalent to*

**class** Square **extends** Rectangle {

constructor(...args) {

**super**(...args);

}

}

使用super的注意事项：

1. 只能在子类中使用，否则报错
2. Super必须在this之前调用，因为super是返回this的
3. 唯一可以避免调用super的方法是在constructor中返回对象

93.覆盖父类方法

这一点和java一样，可以用super调用父类方法：

**class** Square **extends** Rectangle {

constructor(length) {

**super**(length, length);

}

*// override, shadow, and call Rectangle.prototype.getArea()*

getArea() {

**return** **super**.getArea();

}

}

94.静态成员继承

子类可以继承父类的静态资源：

**class** Rectangle {

constructor(length, width) {

**this**.length = length;

**this**.width = width;

}

getArea() {

**return** **this**.length \* **this**.width;

}

**static** create(length, width) {

**return** **new** Rectangle(length, width);

}

}

**class** Square **extends** Rectangle {

constructor(length) {

*// same as Rectangle.call(this, length, length)*

**super**(length, length);

}

}

**var** rect = Square.create(3, 4);

console.log(rect **instanceof** Rectangle); *// true*

console.log(rect.getArea()); *// 12*

console.log(rect **instanceof** Square); *// false*

95.父类可以是一个表达式

1.父类是一个function：

**function** Rectangle(length, width) {

**this**.length = length;

**this**.width = width;

}

Rectangle.prototype.getArea = **function**() {

**return** **this**.length \* **this**.width;

};

**class** Square **extends** Rectangle {

constructor(length) {

**super**(length, length);

}

}

**var** x = **new** Square(3);

console.log(x.getArea()); *// 9*

console.log(x **instanceof** Rectangle); *// true*

2.父类是一个执行函数，返回一个function：

**function** Rectangle(length, width) {

**this**.length = length;

**this**.width = width;

}

Rectangle.prototype.getArea = **function**() {

**return** **this**.length \* **this**.width;

};

**function** getBase() {

**return** Rectangle;

}

**class** Square **extends** getBase() {

constructor(length) {

**super**(length, length);

}

}

**var** x = **new** Square(3);

console.log(x.getArea()); *// 9*

console.log(x **instanceof** Rectangle); *// true*

1. 用执行函数做父类，可以实现混入（mixin）：
2. **let** SerializableMixin = {
3. serialize() {
4. **return** JSON.stringify(**this**);
5. }
6. };
7. **let** AreaMixin = {
8. getArea() {
9. **return** **this**.length \* **this**.width;
10. }
11. };
12. **function** mixin(...mixins) {
13. **var** base = **function**() {};
14. Object.assign(base.prototype, ...mixins);
15. **return** base;
16. }
17. **class** Square **extends** mixin(AreaMixin, SerializableMixin) {
18. constructor(length) {
19. **super**();
20. **this**.length = length;
21. **this**.width = length;
22. }
23. }
24. **var** x = **new** Square(3);
25. console.log(x.getArea()); *// 9*
26. console.log(x.serialize()); *// "{"length":3,"width":3}"*

96.继承内置对象

Es5中无法完美继承内置对象，es6中可以：

**class** MyArray **extends** Array {

*// empty*

}

**var** colors = **new** MyArray();

colors[0] = "red";

console.log(colors.length); *// 1*

colors.length = 0;

console.log(colors[0]); *// undefined*

97. Symbol.species

用Symbol.species可以指定对象**instanceof** 。不常用，详细见原著。

98，new.target 抽象类

new.target可以用在类中，其中最常见的用法是创建抽象类：

*// abstract base class*

**class** Shape {

constructor() {

**if** (**new**.target === Shape) {

**throw** **new** Error("This class cannot be instantiated directly.")

}

}

}

**class** Rectangle **extends** Shape {

constructor(length, width) {

**super**();

**this**.length = length;

**this**.width = width;

}

}

**var** x = **new** Shape(); *// throws error*

**var** y = **new** Rectangle(3, 4); *// no error*

console.log(y **instanceof** Shape); *// true*

# 十、增强数组功能

99. Array.of()

在使用构造函数创建数组，传入单个数字时，会把这个数字当作长度使用，而不是第一项。这个行为比较怪异：

**let** items = **new** Array(2);

console.log(items.length); *// 2*

console.log(items[0]); *// undefined*

console.log(items[1]); *// undefined*

用Array.of()可以解决这个问题：

100 .Array.from()

这个方法用于将累数组对象转化为数组。Es5可以使用循环赋值等方法，es6做了简化：

例子：将传入的参数转为数组：

**function** doSomething() {

**var** args = Array.from(arguments);

*// use args*

}

Array.from方法可以接受第二个参数，作为map操作的方法：

**function** translate() {

**return** Array.from(arguments, (value) => value + 1);

}

**let** numbers = translate(1, 2, 3);

console.log(numbers); *// 2,3,4*

Array.from方法可以接受第三个参数，作为map操作的方法中的this：

**let** helper = {

diff: 1,

add(value) {

**return** value + **this**.diff;

}

};

**function** translate() {

**return** Array.from(arguments, helper.add, helper);

}

**let** numbers = translate(1, 2, 3);

console.log(numbers); *// 2,3,4*

Array.from方法可以接收类数组对象，或者有迭代器的对象。如果都有，以迭代器为准：

**let** numbers = {

\*[Symbol.iterator]() {

**yield** 1;

**yield** 2;

**yield** 3;

}

};

**let** numbers2 = Array.from(numbers, (value) => value + 1);

console.log(numbers2); *// 2,3,4*

101. find() and findIndex()

Es5中添加了indexOf和lastIndexOf两个方法，但是这两个方法都必须知道具体元素才能查询。、

Es6中添加了新的方法，可以根据条件查询：

**let** numbers = [25, 30, 35, 40, 45];

console.log(numbers.find(n => n > 33)); *// 35*

console.log(numbers.findIndex(n => n > 33)); *// 2*

当回调函数返回true时查询结束，这俩方法分别返回元素和索引。

方法还可以接收第二个参数作为回调函数的this

102. fill()

Fill方法填充数组元素，第一个参数是填充元素，第二个参数是起始位，第三个参数是结束位：

**let** numbers = [1, 2, 3, 4];

numbers.fill(1, 2);

console.log(numbers.toString()); *// 1,2,1,1*

numbers.fill(0, 1, 3);

console.log(numbers.toString()); *// 1,0,0,1*

103. copyWithin()

**let** numbers = [1, 2, 3, 4];

*// paste values into array starting at index 2*

*// copy values from array starting at index 0*

*// stop copying values when you hit index 1*

numbers.copyWithin(2, 0, 1);

console.log(numbers.toString()); *// 1,2,1,4*

第三个参数可选

104. Typed Arrays

包含一些Array Buffers之类的东西，可能解析视频的时候会用到。这里不详细写了，详见原著。

# 十一、promise

105.promise的生命周期

在异步函数执行完之前，promise是pending 状态。称为Unsettled

执行完毕之后，称为settled，如果执行成功，为Fulfilled状态。如果失败，为Rejected状态。

状态改变后，会调用then() ,中的处理方法。

注意：实现了then方法的对象叫做thenable，所有promise都是thenable的，但不是所有thenable的都是promise。

**let** promise = readFile("example.txt");

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

*// fulfillment*

console.log(contents);

}, **function**(err) {

*// rejection*

console.error(err.message);

});

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

*// fulfillment*

console.log(contents);

});

promise.then(**null**, **function**(err) {

*// rejection*

console.error(err.message);

});

也可以使用catch来捕捉错误，以下两种写法是相同的：

promise.**catch**(**function**(err) {

*// rejection*

console.error(err.message);

});

*// is the same as:*

promise.then(**null**, **function**(err) {

*// rejection*

console.error(err.message);

});

即使处理函数是异步函数早已执行结束之后添加的，也依然会执行：

**let** promise = readFile("example.txt");

*// original fulfillment handler*

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

console.log(contents);

*// now add another*

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

console.log(contents);

});

});

106. Creating Unsettle promise

*// Node.js example*

**let** fs = require("fs");

**function** readFile(filename) {

**return** **new** Promise(**function**(resolve, reject) {

*// trigger the asynchronous operation*

fs.readFile(filename, { encoding: "utf8" }, **function**(err, contents) {

*// check for errors*

**if** (err) {

reject(err);

**return**;

}

*// the read succeeded*

resolve(contents);

});

});

}

**let** promise = readFile("example.txt");

*// listen for both fulfillment and rejection*

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

*// fulfillment*

console.log(contents);

}, **function**(err) {

*// rejection*

console.error(err.message);

});

Promise中的resolve和reject，原理比较像setTimeout，他们会把相关方法放入执行计划，但不会马上执行，而是等待触发。

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

console.log("Promise");

resolve();

});

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

console.log("Resolved.");

});

console.log("Hi!");

The output for this example is:

Promise

Hi!

Resolved

107. Creating Settled Promises

**let** promise = Promise.resolve(42);

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

console.log(value); *// 42*

});

Promise.resolve() 可以接受non-promise thenables 作为参数。

这地儿没看懂，用的时候在研究吧

**let** thenable = {

then: **function**(resolve, reject) {

resolve(42);

}

};

**let** p1 = Promise.resolve(thenable);

p1.then(**function**(value) {

console.log(value); *// 42*

});

108.执行时错误捕捉

Promise中隐性的包含了一个try-catch，如果在执行时出现了错误，则会自动调用rejection 。

以下两种写法是完全一样的：

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

**throw** **new** Error("Explosion!");

});

promise.**catch**(**function**(error) {

console.log(error.message); *// "Explosion!"*

});

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

**try** {

**throw** **new** Error("Explosion!");

} **catch** (ex) {

reject(ex);

}

});

promise.**catch**(**function**(error) {

console.log(error.message); *// "Explosion!"*

});

注意：如果reject没有传入处理函数，那么这个错误是不会报出来的。

109. Global Promise Rejection Handling

前面说reject没有处理函数的话，promise中的报错不会显示出来。为了解决这个问题，node和浏览器都添加了一些事件来处理这个问题。该方法不太常用，这里不写了，详情见原著。

110.链式promise

Then方法依然会返回一个promise，所以可以链式调用：

**let** p1 = **new** Promise(**function**(resolve, reject) {

resolve(42);

});

p1.then(**function**(value) {

console.log(value);

}).then(**function**() {

console.log("Finished");

});

拆开了长这样：

**let** p1 = **new** Promise(**function**(resolve, reject) {

resolve(42);

});

**let** p2 = p1.then(**function**(value) {

console.log(value);

})

p2.then(**function**() {

console.log("Finished");

});

111.错误捕捉

**let** p1 = **new** Promise(**function**(resolve, reject) {

resolve(42);

});

p1.then(**function**(value) {

**throw** **new** Error("Boom!");

}).**catch**(**function**(error) {

console.log(error.message); *// "Boom!"*

});

放一个错误捕捉在链式调用的最后 是比较推荐的

112.在链式调用中传值

用 return：

**let** p1 = **new** Promise(**function**(resolve, reject) {

resolve(42);

});

p1.then(**function**(value) {

console.log(value); *// "42"*

**return** value + 1;

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

console.log(value); *// "43"*

});

113.在链式调用中传promise

**let** p1 = **new** Promise(**function**(resolve, reject) {

resolve(42);

});

**let** p2 = **new** Promise(**function**(resolve, reject) {

resolve(43);

});

p1.then(**function**(value) {

*// first fulfillment handler*

console.log(value); *// 42*

**return** p2;

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

*// second fulfillment handler*

console.log(value); *// 43*

});

也可以在执行函数中返回promise：

**let** p1 = **new** Promise(**function**(resolve, reject) {

resolve(42);

});

p1.then(**function**(value) {

console.log(value); *// 42*

*// create a new promise*

**let** p2 = **new** Promise(**function**(resolve, reject) {

resolve(43);

});

**return** p2

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

console.log(value); *// 43*

});

114. The Promise.all() Method

在所有promise都成功之后触发：

**let** p1 = **new** Promise(**function**(resolve, reject) {

resolve(42);

});

**let** p2 = **new** Promise(**function**(resolve, reject) {

resolve(43);

});

**let** p3 = **new** Promise(**function**(resolve, reject) {

resolve(44);

});

**let** p4 = Promise.all([p1, p2, p3]);

p4.then(**function**(value) {

console.log(Array.isArray(value)); *// true*

console.log(value[0]); *// 42*

console.log(value[1]); *// 43*

console.log(value[2]); *// 44*

});

只要有一个失败了，就会立刻出发失败处理：

**let** p1 = **new** Promise(**function**(resolve, reject) {

resolve(42);

});

**let** p2 = **new** Promise(**function**(resolve, reject) {

reject(43);

});

**let** p3 = **new** Promise(**function**(resolve, reject) {

resolve(44);

});

**let** p4 = Promise.all([p1, p2, p3]);

p4.**catch**(**function**(value) {

console.log(Array.isArray(value)) *// false*

console.log(value); *// 43*

});

115. The Promise.race() Method

触发最先返回的promise：

**let** p1 = Promise.resolve(42);

**let** p2 = **new** Promise(**function**(resolve, reject) {

resolve(43);

});

**let** p3 = **new** Promise(**function**(resolve, reject) {

resolve(44);

});

**let** p4 = Promise.race([p1, p2, p3]);

p4.then(**function**(value) {

console.log(value); *// 42*

});

失败和all一样

116.继承promise

可以继承promise创建新的类，不知道有什么用，详见原著

117.promise改进版异步任务

**let** fs = require("fs");

**function** run(taskDef) {

*// create the iterator*

**let** task = taskDef();

*// start the task*

**let** result = task.next();

*// recursive function to iterate through*

(**function** step() {

*// if there's more to do*

**if** (!result.done) {

*// resolve to a promise to make it easy*

**let** promise = Promise.resolve(result.value);

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

result = task.next(value);

step();

}).**catch**(**function**(error) {

result = task.**throw**(error);

step();

});

}

}());

}

*// Define a function to use with the task runner*

**function** readFile(filename) {

**return** **new** Promise(**function**(resolve, reject) {

fs.readFile(filename, **function**(err, contents) {

**if** (err) {

reject(err);

} **else** {

resolve(contents);

}

});

});

}

*// Run a task*

run(**function**\*() {

**let** contents = **yield** readFile("config.json");

doSomethingWith(contents);

console.log("Done");

});

118.未来的异步任务语法

(async **function**() {

**let** contents = await readFile("config.json");

doSomethingWith(contents);

console.log("Done");

});

作者预计是es8出

# 十二. Proxies and the Reflection API

代理一个对象，监听其事件

可监听的事件列表：

| **Proxy Trap** | **Overrides the Behavior Of** | **Default Behavior** |
| --- | --- | --- |
| get | Reading a property value | Reflect.get() |
| set | Writing to a property | Reflect.set() |
| has | The in operator | Reflect.has() |
| deleteProperty | The delete operator | Reflect.deleteProperty() |
| getPrototypeOf | Object.getPrototypeOf() | Reflect.getPrototypeOf() |
| setPrototypeOf | Object.setPrototypeOf() | Reflect.setPrototypeOf() |
| isExtensible | Object.isExtensible() | Reflect.isExtensible() |
| preventExtensions | Object.preventExtensions() | Reflect.preventExtensions() |
| getOwnPropertyDescriptor | Object.getOwnPropertyDescriptor() | Reflect.getOwnPropertyDescriptor() |
| defineProperty | Object.defineProperty() | Reflect.defineProperty |
| ownKeys | Object.keys, Object.getOwnPropertyNames(), Object.getOwnPropertySymbols() | Reflect.ownKeys() |
| apply | Calling a function | Reflect.apply() |
| construct | Calling a function with new | Reflect.construct() |
| Table 11-1: Proxy traps in JavaScript | | |

119.监听set事件

**let** target = {

name: "target"

};

**let** proxy = **new** Proxy(target, {

set(trapTarget, key, value, receiver) {

*// ignore existing properties so as not to affect them*

**if** (!trapTarget.hasOwnProperty(key)) {

**if** (isNaN(value)) {

**throw** **new** TypeError("Property must be a number.");

}

}

*// add the property*

**return** Reflect.set(trapTarget, key, value, receiver);

}

});

*// adding a new property*

proxy.count = 1;

console.log(proxy.count); *// 1*

console.log(target.count); *// 1*

*// you can assign to name because it exists on target already*

proxy.name = "proxy";

console.log(proxy.name); *// "proxy"*

console.log(target.name); *// "proxy"*

*// throws an error*

proxy.anotherName = "proxy";

120.监听get事件

**let** proxy = **new** Proxy({}, {

get(trapTarget, key, receiver) {

**if** (!(key **in** receiver)) {

**throw** **new** TypeError("Property " + key + " doesn't exist.");

}

**return** Reflect.get(trapTarget, key, receiver);

}

});

*// adding a property still works*

proxy.name = "proxy";

console.log(proxy.name); *// "proxy"*

*// nonexistent properties throw an error*

console.log(proxy.nme); *// throws error*

121.监听has

**let** target = {

name: "target",

value: 42

};

**let** proxy = **new** Proxy(target, {

has(trapTarget, key) {

**if** (key === "value") {

**return** **false**;

} **else** {

**return** Reflect.has(trapTarget, key);

}

}

});

console.log("value" **in** proxy); *// false*

console.log("name" **in** proxy); *// true*

console.log("toString" **in** proxy); *// true*

122.监听删除属性事件

**let** target = {

name: "target",

value: 42

};

**let** proxy = **new** Proxy(target, {

deleteProperty(trapTarget, key) {

**if** (key === "value") {

**return** **false**;

} **else** {

**return** Reflect.deleteProperty(trapTarget, key);

}

}

});

*// Attempt to delete proxy.value*

console.log("value" **in** proxy); *// true*

**let** result1 = **delete** proxy.value;

console.log(result1); *// false*

console.log("value" **in** proxy); *// true*

*// Attempt to delete proxy.name*

console.log("name" **in** proxy); *// true*

**let** result2 = **delete** proxy.name;

console.log(result2); *// true*

console.log("name" **in** proxy); *// false*

123.监听原型操作

保护原型只读（只能通过代理，直接操作target依然可以）

. **let** target = {};

**let** proxy = **new** Proxy(target, {

getPrototypeOf(trapTarget) {

**return** Reflect.getPrototypeOf(trapTarget);

},

setPrototypeOf(trapTarget, proto) {

**return** **false**;

}

});

**let** targetProto = Object.getPrototypeOf(target);

**let** proxyProto = Object.getPrototypeOf(proxy);

console.log(targetProto === Object.prototype); *// true*

console.log(proxyProto === Object.prototype); *// false*

console.log(proxyProto); *// null*

*// succeeds*

Object.setPrototypeOf(target, {});

*// throws error*

Object.setPrototypeOf(proxy, {});

124.监听扩展性

阻止将对象设为不可扩展

**let** target = {};

**let** proxy = **new** Proxy(target, {

isExtensible(trapTarget) {

**return** Reflect.isExtensible(trapTarget);

},

preventExtensions(trapTarget) {

**return** **false**

}

});

console.log(Object.isExtensible(target)); *// true*

console.log(Object.isExtensible(proxy)); *// true*

Object.preventExtensions(proxy);

console.log(Object.isExtensible(target)); *// true*

console.log(Object.isExtensible(proxy)); *// true*

125.监听Property Descriptor

基本同上，这里简写：

defineProperty

getOwnPropertyDescriptor

126.监听ownKeys

以下方法会用到：

Object.keys()

Object.getOwnPropertyNames()

Object.getOwnPropertySymbols()

Object.assign()

127.监听function的call和construct

*// adds together all arguments*

**function** sum(...values) {

**return** values.reduce((previous, current) => previous + current, 0);

}

**let** sumProxy = **new** Proxy(sum, {

apply: **function**(trapTarget, thisArg, argumentList) {

argumentList.forEach((arg) => {

**if** (**typeof** arg !== "number") {

**throw** **new** TypeError("All arguments must be numbers.");

}

});

**return** Reflect.apply(trapTarget, thisArg, argumentList);

},

construct: **function**(trapTarget, argumentList) {

**throw** **new** TypeError("This function can't be called with new.");

}

});

console.log(sumProxy(1, 2, 3, 4)); *// 10*

*// throws error*

console.log(sumProxy(1, "2", 3, 4));

*// also throws error*

**let** result = **new** sumProxy();

可以用来做参数验证，或者限制只能用new调用，或者不能用new调用等

128.解绑Proxy

**let** target = {

name: "target"

};

**let** { proxy, revoke } = Proxy.revocable(target, {});

console.log(proxy.name); *// "target"*

revoke();

*// throws error*

console.log(proxy.name);

129.以proxy作为原型

以proxy做原型之后，大部分监听会失效，只有get，set，has可以继承。以get为例：

**let** target = {};

**let** thing = Object.create(**new** Proxy(target, {

get(trapTarget, key, receiver) {

**throw** **new** ReferenceError(`**${**key**}** doesn't exist`);

}

}));

thing.name = "thing";

console.log(thing.name); *// "thing"*

*// throw an error*

**let** unknown = thing.unknown;

130.以proxy为原型的class

**function** NoSuchProperty() {

*// empty*

}

NoSuchProperty.prototype = **new** Proxy({}, {

get(trapTarget, key, receiver) {

**throw** **new** ReferenceError(`**${**key**}** doesn't exist`);

}

});

**class** Square **extends** NoSuchProperty {

constructor(length, width) {

**super**();

**this**.length = length;

**this**.width = width;

}

getArea() {

**return** **this**.length \* **this**.width;

}

}

**let** shape = **new** Square(2, 6);

**let** area1 = shape.length \* shape.width;

console.log(area1); *// 12*

**let** area2 = shape.getArea();

console.log(area2); *// 12*

*// throws an error because "wdth" doesn't exist*

**let** area3 = shape.length \* shape.wdth;

# 十三、用Modules封装代码

131. Modules的特性

1.modules自动为严格模式

2.modules中this为undefinded

132.基本导出

*// export data*

**export** **var** color = "red";

**export** **let** name = "Nicholas";

**export** **const** magicNumber = 7;

*// export function*

**export** **function** sum(num1, num2) {

**return** num1 + num1;

}

*// export class*

**export** **class** Rectangle {

constructor(length, width) {

**this**.length = length;

**this**.width = width;

}

}

*// this function is private to the module*

**function** subtract(num1, num2) {

**return** num1 - num2;

}

*// define a function...*

**function** multiply(num1, num2) {

**return** num1 \* num2;

}

*// ...and then export it later*

**export** { multiply };

可以导出声明或者引用，如果要导出匿名函数，则需要default关键字

133.基本导入

**import** { identifier1, identifier2 } from "./example.js";

区分库文件和本地文件的惯例：

example would be a package and ./example.js would be a local file.

注意：这个地方看起来像解构，但它不是

导入的变量相当于使用const赋值，不可修改，声明前不可调用。

确保在路径之前加 /、./、../，来兼容浏览器和nodejs。在浏览器中不加前缀会报错

134.导入全部

*// import everything*

**import** \* as example from "./example.js";

console.log(example.sum(1,

example.magicNumber)); *// 8*

console.log(example.multiply(1, 2)); *// 2*

即使一个文件被导入多次，它只会执行一次，结果会被缓存

**import** { sum } from "./example.js";

**import** { multiply } from "./example.js";

**import** { magicNumber } from "./example.js";

语法限制：导入语句只能放在顶级，不能放在代码中，比如if，function。也就是说不能动态导入模块。

135.导出默认模块

**export** **default** **function**(num1, num2) {

**return** num1 + num2;

}

**function** sum(num1, num2) {

**return** num1 + num2;

}

**export** **default** sum;

**function** sum(num1, num2) {

**return** num1 + num2;

}

**export** { sum as **default** };

136.导入默认模块

注意没有大括号

*// import the default*

**import** sum from "./example.js";

console.log(sum(1, 2)); *// 3*

模块也可以同时拥有默认值和非默认值

**export** **let** color = "red";

**export** **default** **function**(num1, num2) {

**return** num1 + num2;

}

You can import both color and the default function using the following import statement:

**import** sum, { color } from "./example.js";

console.log(sum(1, 2)); *// 3*

console.log(color); *// "red"*

也可以这样写：

*// equivalent to previous example*

**import** { **default** as sum, color } from "example";

console.log(sum(1, 2)); *// 3*

console.log(color); *// "red"*

137.将导入的模块导出

下面两段代码是相同的

**import** { sum } from "./example.js";

**export** { sum }

**export** { sum } from "./example.js";

导出并重命名

**export** { sum as add } from "./example.js";

导出全部：

**export** \* from "./example.js";

138.没有export的模块

模块中的变量不会影响全局，但是模块中的内置对象，比如array和object是全局共享的。所以一个模块即使不export任何东西，也可以对程序产生影响：

*/ module code without exports or imports*

Array.prototype.pushAll = **function**(items) {

*// items must be an array*

**if** (!Array.isArray(items)) {

**throw** **new** TypeError("Argument must be an array.");

}

*// use built-in push() and spread operator*

**return** **this**.push(...items);

};

这是个可用的模块，即使他没有导出任何值。这种模块可以当作模块来用，也可以当作script来用。因为它没有导出任何东西，所以可以简单的使用import来执行代码。

**import** "./example.js";

**let** colors = ["red", "green", "blue"];

**let** items = [];

items.pushAll(colors);

这种方法经常用来写pollyfill shims之类的东西

139.在浏览器中使用模块

1.用script，type=module

*<!-- load a module JavaScript file -->*

<**script** type="module" src="module.js"></**script**>

*<!-- include a module inline -->*

<**script** type="module">

**import** { sum } from "./example.js";

**let** result = sum(1, 2);

</**script**>

2.模块的加载和执行顺序：

Type=module的script是默认异步延迟加载的 相当于defer （写了也会被忽略掉）

这种模块的加载是按出现顺序加载，不区分inline或者src。如果模块里有import则直接去下载。

全部下载完之后，按照依赖关系（也是下载完成顺序）执行；

# 十四、附录

140.  \_\_proto\_\_

Es6建议使用Object.getPrototypeOf() and Object.setPrototypeOf()

而不建议使用\_\_proto\_\_

因为\_\_proto\_\_已经被比较广泛的应用，所以使用\_\_proto\_\_会导致一条警告，而不是禁用。

# 十五、es7

141.幂操作符，代替Math.pow

**let** result = 5 \*\* 2;

console.log(result); *// 25*

console.log(result === Math.pow(5, 2)); *// true*

幂操作符具有最高优先级

使用负数的时候要注意加括号，为了避免误会

*// syntax error*

**let** result = -5 \*\* 2;

*// ok*

**let** result1 = -(5 \*\* 2); *// equal to -25*

*// also ok*

**let** result2 = (-5) \*\* 2; *// equal to 25*

142. Array.prototype.includes() Method

Es6添加了string的includes方法，es7里把这个方法添加到了数组里：、

**let** values = [1, 2, 3];

console.log(values.includes(1)); *// true*

console.log(values.includes(0)); *// false*

*// start the search from index 2*

console.log(values.includes(1, 2)); *// false*

接受两个参数，查询项目，查询开始index

内部是用===做比较的

特殊处理了NAN：

**let** values = [1, **NaN**, 2];

console.log(values.indexOf(**NaN**)); *// -1*

console.log(values.includes(**NaN**)); *// true*

143.function作用域下的严格模式

问题原因：

**function** doSomething(first = **this**) {

"use strict";

**return** first;

}

参数this指向不明等问题

解决方案，在使用解构和默认参数的情况下，不允许添加function作用域严格模式：

*// okay - using simple parameter list*

**function** okay(first, second) {

"use strict";

**return** first;

}

*// syntax error*

**function** notOkay1(first, second=first) {

"use strict";

**return** first;

}

*// syntax error*

**function** notOkay2({ first, second }) {

"use strict";

**return** first;

}