# 第一章 现代前端开发

## 1.1Es6—新一代的javascirpt

### 1.1.1语言特性

#### const let关键字（关键字什么环境下用）

**Const定义常量不可以修改**。

**Let定义**，

If（true）{

Let a = “name”

}

Console.log(a);

#### 函数

**箭头函数**：（箭头函数什么环境下用）

**This在箭头函数的使用**：

**Rest参数数组**，函数方法参数添加…，argument参数集合

function test2(name,…args){//函数方法中，参数加…

Console.log(args);

}

Test2(“Peter”,2,3);//执行方法时参数展开

#### 展示开操作符…

（不会用）

**用户函数调用…**

function test(X,Y,Z){console.log(X+"-"+Y+"-"+Z)} var arg=[1,2,3]; test.apply(null,arg);

function test(X,Y,Z){console.log(X+"-"+Y+"-"+Z)} var arg=[1,2,3]; test(...arg);

**用于数组字面量…**

Splice concat push

var arr1=[1,2,3];var arr2=[4,5,6]; var arr3=arr1.concat(arr2);console.log(arr3)

var arr1=[1,2,3];var arr2=[4,5,6]; var arr3=[...arr1,...arr2];console.log(arr3)

**对像展开运算符**

var mike={name:'mike',age:50};mike={...mike,sex:'male'};console.log(mike);

#### 模板字符串

var name='mike';var a=`My name is ${name} !`;console.log(a);

var name='mike'; var age=20; var a=`My name is ${name} !

My age is ${age}`;console.log(a);

#### 结构赋值

**结构语法可以快速从数组或者对像中提取变量，可以用一个表达式读取整个结构。**

**解构数组**

var foo=['one','two','three']; var [one,two,three] = foo; console.log(`${one},${two},${three}`)

解构对像

var person={name:'mike',age:'20'};var {name,age}=person;console.log(`${name},${age}`);

#### 类

class animal(){//不可用

constructor(name,age){

this.name=name; this.age=age;

}

shout(){

return `My name is ${this.name}, age is ${this.age}`;

}

state foo(){

return `Here is state method`;

}

}

var con=new animal('betty',2); con.shout();

class Dog extends animal{

constructor(name,age=2,color='black'){

super(name,age);//构造函数可用直接用super

this.color=color;

}

shout(){

return super.shout()+`,color is ${this.color}`;

}

}

var jackTheDog = new Dog('jack');

jackTheDog.shout();

#### 模块

1、Export 和Import需处于模块顶层，import语句不能有表达式和变量，且导出方法应加大括号｛｝

**2、import** 'lodash'; 会执行所加载的模块，若重负测只加载一遍。

**3**、利用import模块整体加载所在的那个对象，应该是可以静态分析的，所以不允许运行时改变。

**4、export default命令，import可用任意名称指向default，且不需要带大括号。如：**

Export hello as HI;

**export** {firstName, lastName, year};

import customName from './export-default';//导出defaul方法

import \_, { each } from 'lodash';导出default方法和each方法。

Import {PI ,hello ,person} from ‘./hello’;

Import \* as util from ‘./hello’;

**5、export 与 import 的复合写法**

**export** { foo, bar } **from** 'my\_module';

*// 等同于*

**import** { foo, bar } **from** 'my\_module';

**export** { foo, bar };

**export** { **default** } **from** 'foo';

**export** { es6 **as** **default** } **from** './someModule';

**export** \* **from** 'my\_module';

#### set和map

**Set**

set实例的方法分为两大类：操作方法（用于操作数据）和遍历方法（用于遍历成员）。

set四个操作方法:

add(value)：添加某个值，返回Set结构本身。

delete(value)：删除某个值，返回一个布尔值，表示删除是否成功。

has(value)：返回一个布尔值，表示该值是否为Set的成员。

clear()：清除所有成员，没有返回值

set四个遍历方法：

keys()：返回键名的遍历器

values()：返回键值的遍历器

entries()：返回键值对的遍历器

forEach()：使用回调函数遍历每个成员

**let** **set** = **new** Set(['red', 'green', 'blue']);

**for** (**let** item of **set**.keys()) {

console.log(item);

}

*// red// green// blue*

**for** (**let** item of **set**.values()) {

console.log(item);

}

*// red// green// blue*

**for** (**let** item of **set**.entries()) {

console.log(item);

}

*// ["red", "red"]// ["green", "green"]// ["blue", "blue"]*

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

**set**.forEach((value, key) => console.log(value \* 2) )

*// 2// 4// 6*

**map**

map类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。也就是说，Object结构提供了“字符串—值”的对应，Map结构提供了“值—值”的对应，是一种更完善的Hash结构实现。如果你需要“键值对”的数据结构，Map比Object更合适。

属性和方法：size, set(key,value), get(key), has(key), delete(key), clear();

eys()：返回键名的遍历器。

values()：返回键值的遍历器。

entries()：返回所有成员的遍历器。

forEach()：遍历Map的所有成员

let map = new Map([

['F', 'no'],

['T', 'yes'],

]);

for (let key of map.keys()) {

console.log(key);

}

// "F"// "T"

for (let value of map.values()) {

console.log(value);

}

// "no"// "yes"

for (let item of map.entries()) {

console.log(item[0], item[1]);

}

// "F" "no"// "T" "yes"

// 或者

for (let [key, value] of map.entries()) {

console.log(key, value);

}

// 等同于使用map.entries()

for (let [key, value] of map) {

console.log(key, value);

}

#### Promise对像

##### Promise的含义

Promise是异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理和更强大。

两个特点：

1. 对像的状态不受外界影响。Promise对像代表一个异步操作，有三种状态：pending(进行中)、fulfilled(已成功)和rejeccted(已失败)。
2. 一旦状态改变，就不会再变，任何时候都可以得到这个结果。Promise对像的状态改变，只有两种可能：从pending变为fulfilled和从pending变为rejected。

只要两种情况发生，状态就凝固了，不会再变了，这时就称为resolved（已定型）。如果改变已经发生了，你再对Promise对像添加回调函数，也会产即得到这个结果。

优点：可以将异步操作以同步操作的流程表达出来 ，避免了层层嵌套的回调函数。引外，Promise对像提供统一的接口，使得控制异步操作更加容易。

缺点：首先，无法取消Promise，一旦新建它就会立即执行，无法中途取消。其次，如果不设置回调函数，Promise内部抛出的错误，不会反应到外部。第三当处于pending状态时，无法得知目前进展到哪一个阶段。（刚刚开始还是即将完成）。

##### 基本用法

用法一：ES6规定，Promise对像是一个构造函数，用来生成Promise实例。下面代码创造了一个Promise实例。

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

if(/\*异步操作成功\*/){

resolve(value);

}else{

reject(error);

}

});

Promise构造函数接受一个函数作为参数，该函数的两个参数分别是resolve和reject。它们是两个函数，由JavaScript引擎提供。

resolse函数的作用是，将Promise对像的状态从“未完成”变为“成功”（即从pending变为resolved），在异步操作成功时调用，并将异步操作的结果，作为参数传递出去;reject函数的作用是，将Promise对像的状态从“未完成”变为“失败”（即pending变为rejected），在异步操作失败时调用，并将异步操作报出的错误，作为参数传递出去。

用法二：Promise实例生成以后，可以用then方法分别指定resolved状态和rejected状态的回调函数。

promise.then(function(valuve){

//success

},function(error){

//failure

});

then方法可以接受两个回调函数作为参数。第一个回调函数是Promise对像的状态变为resolved时调用，第二个回调函数是Promise对像的状态变为rejected时调用。其中，第二个函数是可选的，不一定要提供。这两个函数都接受Promise对像传出的值作为参数。

##### Promise.prototype.then()

Promise 实例具有then方法，也就是说，then方法是定义在原型对象Promise.prototype上的。它的作用是为 Promise 实例添加状态改变时的回调函数。前面说过，then方法的第一个参数是resolved状态的回调函数，第二个参数（可选）是rejected状态的回调函数。

then方法返回的是一个新的Promise实例（注意，不是原来那个Promise实例）。因此可以采用链式写法，即then方法后面再调用另一个then方法

采用链式的then，可以指定一组按照次序调用的回调函数。这时，前一个回调函数，有可能返回的还是一个Promise对象（即有异步操作），这时后一个回调函数，就会等待该Promise对象的状态发生变化，才会被调用。

getJSON("/post/1.json").then(function(post) {

return getJSON(post.commentURL);

}).then(function funcA(comments) {

console.log("resolved: ", comments);

}, function funcB(err){

console.log("rejected: ", err);

});

上面代码中，第一个then方法指定的回调函数，返回的是另一个Promise对象。这时，第二个then方法指定的回调函数，就会等待这个新的Promise对象状态发生变化。如果变为resolved，就调用funcA，如果状态变为rejected，就调用funcB。

如果采用箭头函数，上面的代码可以写得更简洁。

getJSON("/post/1.json").then(

post => getJSON(post.commentURL)

).then(

comments => console.log("resolved: ", comments),

err => console.log("rejected: ", err)

);

##### Promise.prototype.catch()

Promise.prototype.catch方法是.then(null, rejection)的别名，用于指定发生错误时的回调函数。

**getJSON('/posts.json').then(function(posts) {**

**// ...**

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

**// 处理 getJSON 和 前一个回调函数运行时发生的错误**

**console.log('发生错误！', error);**

**});**

then方法指定的回调函数，如果运行中抛出错误，也会被catch方法捕获。

**p.then((val) => console.log('fulfilled:', val))**

**.catch((err) => console.log('rejected', err));**

**// 等同于**

**p.then((val) => console.log('fulfilled:', val))**

**.then(null, (err) => console.log("rejected:", err));**

下面是一个例子。

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

**throw new Error('test');**

**});**

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

**console.log(error);**

**});**

上面代码中，promise抛出一个错误，就被catch方法指定的回调函数捕获。注意，上面的写法与下面两种写法是等价的。

**// 写法一**

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

**try {**

**throw new Error('test');**

**} catch(e) {**

**reject(e);**

**}**

**});**

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

**console.log(error);**

**});**

**// 写法二**

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

**reject(new Error('test'));**

**});**

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

**console.log(error);**

**});**

比较上面两种写法，可以发现reject方法的作用，等同于抛出错误。

一般来说，不要在then方法里面定义 Reject 状态的回调函数（即then的第二个参数），总是使用catch方法。

##### Promise.prototype.finally() [§](http://es6.ruanyifeng.com/#docs/promise#Promise-prototype-finally) [⇧](http://es6.ruanyifeng.com/#docs/promise)

finally方法用于指定不管 Promise 对象最后状态如何，都会执行的操作。该方法是 ES2018 引入标准的。

**promise**

**.then(result => {···})**

**.catch(error => {···})**

**.finally(() => {···});**

上面代码中，不管promise最后的状态，在执行完then或catch指定的回调函数以后，都会执行finally方法指定的回调函数。

##### Promise.all()

Promise.all方法用于将多个 Promise 实例，包装成一个新的 Promise 实例。

**const p = Promise.all([p1, p2, p3]);**

上面代码中，Promise.all方法接受一个数组作为参数，p1、p2、p3都是 Promise 实例，如果不是，就会先调用下面讲到的Promise.resolve方法，将参数转为 Promise 实例，再进一步处理。（Promise.all方法的参数可以不是数组，但必须具有 Iterator 接口，且返回的每个成员都是 Promise 实例。）

p的状态由p1、p2、p3决定，分成两种情况。

（1）只有p1、p2、p3的状态都变成fulfilled，p的状态才会变成fulfilled，此时p1、p2、p3的返回值组成一个数组，传递给p的回调函数。

（2）只要p1、p2、p3之中有一个被rejected，p的状态就变成rejected，此时第一个被reject的实例的返回值，会传递给p的回调函数。

Const p1 = new Promise((resolve, reject) => {

resolve('hello');

})

.then(result => result)

.catch(e => e);

const p2 = new Promise((resolve, reject) => {

throw new Error('报错了');

})

.then(result => result)

.catch(e => e);

Promise.all([p1, p2])

.then(result => console.log(result))

.catch(e => console.log(e));

// ["hello", Error: 报错了]

如果p2没有自己的catch方法，就会调用Promise.all()的catch方法。

const p1 = new Promise((resolve, reject) => {

resolve('hello');

})

.then(result => result);

const p2 = new Promise((resolve, reject) => {

throw new Error('报错了');

})

.then(result => result);

Promise.all([p1, p2])

.then(result => console.log(result))

.catch(e => console.log(e));

// Error: 报错了

##### Promise.race() [§](http://es6.ruanyifeng.com/#docs/promise#Promise-race) [⇧](http://es6.ruanyifeng.com/#docs/promise)

Promise.race方法同样是将多个 Promise 实例，包装成一个新的 Promise 实例。

上面代码中，只要p1、p2、p3之中有一个实例率先改变状态，p的状态就跟着改变。那个率先改变的 Promise 实例的返回值，就传递给p的回调函数。

下面是一个例子，如果指定时间内没有获得结果，就将 Promise 的状态变为reject，否则变为resolve。

const p = Promise.race([

fetch('/resource-that-may-take-a-while'),

new Promise(function (resolve, reject) {

setTimeout(() => reject(new Error('request timeout')), 5000)

})

]);

p

.then(console.log)

.catch(console.error);

上面代码中，如果 5 秒之内fetch方法无法返回结果，变量p的状态就会变为rejected，从而触发catch方法指定的回调函数。

##### Promise.resolve()

有时需要将现有对象转为 Promise 对象，Promise.resolve方法就起到这个作用。

Promise.resolve('foo')

// 等价于

new Promise(resolve => resolve('foo'))

参数分为四种情况：

**（1）参数是一个 Promise 实例**

如果参数是 Promise 实例，那么Promise.resolve将不做任何修改、原封不动地返回这个实例

**（2）参数是一个thenable对象**

thenable对象指的是具有then方法的对象，比如下面这个对象。

let thenable = {

then: function(resolve, reject) {

resolve(42);

}

};

Promise.resolve方法会将这个对象转为 Promise 对象，然后就立即执行thenable对象的then方法。

let thenable = {

then: function(resolve, reject) {

resolve(42);

}

};

let p1 = Promise.resolve(thenable);

p1.then(function(value) {

console.log(value); // 42

});

上面代码中，thenable对象的then方法执行后，对象p1的状态就变为resolved，从而立即执行最后那个then方法指定的回调函数，输出 42。

**（3）参数不是具有then方法的对象，或根本就不是对象**

如果参数是一个原始值，或者是一个不具有then方法的对象，则Promise.resolve方法返回一个新的 Promise 对象，状态为resolved。

const p = Promise.resolve('Hello');

p.then(function (s){

console.log(s)

}); // Hello

上面代码生成一个新的 Promise 对象的实例p。由于字符串Hello不属于异步操作（判断方法是字符串对象不具有 then 方法），返回 Promise 实例的状态从一生成就是resolved，所以回调函数会立即执行。Promise.resolve方法的参数，会同时传给回调函数。

**（4）不带有任何参数**

Promise.resolve方法允许调用时不带参数，直接返回一个resolved状态的 Promise 对象。

所以，如果希望得到一个 Promise 对象，比较方便的方法就是直接调用Promise.resolve方法。

const p = Promise.resolve();

p.then(function () {

// ...

});

上面代码的变量p就是一个 Promise 对象。

##### Promise.reject() [§](http://es6.ruanyifeng.com/#docs/promise#Promise-reject) [⇧](http://es6.ruanyifeng.com/#docs/promise)

Promise.reject(reason)方法也会返回一个新的 Promise 实例，该实例的状态为rejected。

const p = Promise.reject('出错了');

// 等同于

const p = new Promise((resolve, reject) => reject('出错了'))

p.then(null, function (s) {

console.log(s)

});

上面代码生成一个 Promise 对象的实例p，状态为rejected，回调函数会立即执行。

注意，Promise.reject()方法的参数，会原封不动地作为reject的理由，变成后续方法的参数。这一点与Promise.resolve方法不一致。

const thenable = {

then(resolve, reject) {

reject('出错了');

}

};

Promise.reject(thenable)

.catch(e => {

console.log(e === thenable)

}) // true

上面代码中，Promise.reject方法的参数是一个thenable对象，执行以后，后面catch方法的参数不是reject抛出的“出错了”这个字符串，而是thenable对象。

##### 应用 [§](http://es6.ruanyifeng.com/#docs/promise#应用) [⇧](http://es6.ruanyifeng.com/#docs/promise)

**加载图片**

我们可以将图片的加载写成一个Promise，一旦加载完成，Promise的状态就发生变化。

const preloadImage = function (path) {

return new Promise(function (resolve, reject) {

const image = new Image();

image.onload = resolve;

image.onerror = reject;

image.src = path;

});

};

**Generator 函数与 Promise 的结合** [**§**](http://es6.ruanyifeng.com/#docs/promise#Generator-函数与-Promise-的结合)[**⇧**](http://es6.ruanyifeng.com/#docs/promise)

使用 Generator 函数管理流程，遇到异步操作的时候，通常返回一个Promise对象。

function getFoo () {

return new Promise(function (resolve, reject){

resolve('foo');

});

}

const g = function\* () {

try {

const foo = yield getFoo();

console.log(foo);

} catch (e) {

console.log(e);

}

};

function run (generator) {

const it = generator();

function go(result) {

if (result.done) return result.value;

return result.value.then(function (value) {

return go(it.next(value));

}, function (error) {

return go(it.throw(error));

});

}

go(it.next());

}

run(g);

上面代码的 Generator 函数g之中，有一个异步操作getFoo，它返回的就是一个Promise对象。函数run用来处理这个Promise对象，并调用下一个next方法。

##### Promise.try()

实际开发中，经常遇到一种情况：不知道或者不想区分，函数f是同步函数还是异步操作，但是想用 Promise 来处理它。因为这样就可以不管f是否包含异步操作，都用then方法指定下一步流程，用catch方法处理f抛出的错误。一般就会采用下面的写法。

Promise.resolve().then(f)

上面的写法有一个缺点，就是如果f是同步函数，那么它会在本轮事件循环的末尾执行。

const f = () => console.log('now');

Promise.resolve().then(f);

console.log('next');

// next

// now

上面代码中，函数f是同步的，但是用 Promise 包装了以后，就变成异步执行了。

那么有没有一种方法，让同步函数同步执行，异步函数异步执行，并且让它们具有统一的 API 呢？回答是可以的，并且还有两种写法。第一种写法是用async函数来写。

const f = () => console.log('now');

(async () => f())();

console.log('next');

// now

// next

上面代码中，第二行是一个立即执行的匿名函数，会立即执行里面的async函数，因此如果f是同步的，就会得到同步的结果；如果f是异步的，就可以用then指定下一步，就像下面的写法。

(async () => f())()

.then(...)

需要注意的是，async () => f()会吃掉f()抛出的错误。所以，如果想捕获错误，要使用promise.catch方法。

(async () => f())()

.then(...)

.catch(...)

第二种写法是使用new Promise()。

const f = () => console.log('now');

(

() => new Promise(

resolve => resolve(f())

)

)();

console.log('next');

// now

// next

上面代码也是使用立即执行的匿名函数，执行new Promise()。这种情况下，同步函数也是同步执行的。

鉴于这是一个很常见的需求，所以现在有一个[提案](https://github.com/ljharb/proposal-promise-try)，提供Promise.try方法替代上面的写法。

const f = () => console.log('now');

Promise.try(f);

console.log('next');

// now

// next

事实上，Promise.try存在已久，Promise 库[Bluebird](http://bluebirdjs.com/docs/api/promise.try.html)、[Q](https://github.com/kriskowal/q/wiki/API-Reference#promisefcallargs)和[when](https://github.com/cujojs/when/blob/master/docs/api.md#whentry)，早就提供了这个方法。

由于Promise.try为所有操作提供了统一的处理机制，所以如果想用then方法管理流程，最好都用Promise.try包装一下。这样有[许多好处](http://cryto.net/~joepie91/blog/2016/05/11/what-is-promise-try-and-why-does-it-matter/)，其中一点就是可以更好地管理异常。

1. function getUsername(userId) {
2. return database.users.get({id: userId})
3. .then(function(user) {
4. return user.name;
5. });
6. }

上面代码中，database.users.get()返回一个 Promise 对象，如果抛出异步错误，可以用catch方法捕获，就像下面这样写。

database.users.get({id: userId})

.then(...)

.catch(...)

但是database.users.get()可能还会抛出同步错误（比如数据库连接错误，具体要看实现方法），这时你就不得不用try...catch去捕获。

try {

database.users.get({id: userId})

.then(...)

.catch(...)

} catch (e) {

// ...

}

上面这样的写法就很笨拙了，这时就可以统一用promise.catch()捕获所有同步和异步的错误。

Promise.try(database.users.get({id: userId}))

.then(...)

.catch(...)

事实上，Promise.try就是模拟try代码块，就像promise.catch模拟的是catch代码块。

#### Iterator（遍历器）

##### Iterator（遍历器）的概念

JavaScript原有的表示“集合”的数据结构，主要是数组（Array）和对像（Object），ES6以添加了Map和Set。这样有了四种数据集合，用户还可以组合使用它们，定义自己的数据结构，比如数组的成员是Map，Map成员是对像。这样就需要一种统一的接口机制，来处理所有不同的数据结构。

遍历器（Iterator）就是这样一种机制。它是一种接口，为各种不同的数据结构提供统一的访问机制。任何数据结构只要部署Iterator接口，就可以完成遍历操作

Ietrator的作用有三个：一是为各种数据结构，提供一个统一的、简便的访问接口；二是使得数据结构的成员能够按某种次序排列；三是ES6创造了一种新的遍历命令for…of循环，Iterator接口主要供for…of消费。

Iterator的遍历过程是第一次调用next方法，都会返回数据结构的当前成员信息。具体来说，就是返回一个包含value和done两个属性的对象。其中，value属性是当前成员的值，done属性是一个布尔值，表示遍历是否结束。

##### 默认Iterator接口

ES6 规定，默认的 Iterator 接口部署在数据结构的Symbol.iterator属性，或者说，一个数据结构只要具有Symbol.iterator属性，就可以认为是“可遍历的”（iterable）。Symbol.iterator属性本身是一个函数，就是当前数据结构默认的遍历器生成函数。执行这个函数，就会返回一个遍历器。至于属性名Symbol.iterator，它是一个表达式，返回Symbol对象的iterator属性，这是一个预定义好的、类型为 Symbol 的特殊值，所以要放在方括号内（参见《Symbol》一章）。

原生具备 Iterator 接口的数据结构如下。

Array

Map

Set

String

TypedArray

函数的 arguments 对象

NodeList 对象

let iterable = {

0: 'a',

1: 'b',

2: 'c',

length: 3,

[Symbol.iterator]: Array.prototype[Symbol.iterator]

};

for (let item of iterable) {

console.log(item); // 'a', 'b', 'c'

}

##### 调用Iterator接口的场合

有一些场合会默认调用 Iterator 接口（即Symbol.iterator方法），除了下文会介绍的for...of循环，还有几个别的场合。

1. **解构赋值**

let set = new Set().add('a').add('b').add('c');

let [x,y] = set; // x='a'; y='b'

let [first, ...rest] = set;

// first='a'; rest=['b','c'];

1. **扩展运算**

var str = 'hello';

[...str] // ['h','e','l','l','o']

// 例二

let arr = ['b', 'c'];

['a', ...arr, 'd']

// ['a', 'b', 'c', 'd']

1. **yield\***

yield\*后面跟的是一个可遍历的结构，它会调用该结构的遍历器接口。

let generator = function\* () {

yield 1;

yield\* [2,3,4];

yield 5;

};

var iterator = generator();

iterator.next() // { value: 1, done: false }

iterator.next() // { value: 2, done: false }

iterator.next() // { value: 3, done: false }

iterator.next() // { value: 4, done: false }

iterator.next() // { value: 5, done: false }

iterator.next() // { value: undefined, done: true }

1. **其他场合**

由于数组的遍历会调用遍历器接口，所以任何接受数组作为参数的场合，其实都调用了遍历器接口。下面是一些例子。

for...of

Array.from()

Map(), Set(), WeakMap(), WeakSet()（比如new Map([['a',1],['b',2]])）

Promise.all()

Promise.race()

##### 字符串的 Iterator 接口 [§](http://es6.ruanyifeng.com/#docs/iterator#字符串的-Iterator-接口) [⇧](http://es6.ruanyifeng.com/#docs/iterator)

字符串是一个类似数组的对象，也原生具有 Iterator 接口。

var someString = "hi";

typeof someString[Symbol.iterator]

// "function"

var iterator = someString[Symbol.iterator]();

iterator.next() // { value: "h", done: false }

iterator.next() // { value: "i", done: false }

iterator.next() // { value: undefined, done: true }

上面代码中，调用Symbol.iterator方法返回一个遍历器对象，在这个遍历器上可以调用 next 方法，实现对于字符串的遍历。

可以覆盖原生的Symbol.iterator方法，达到修改遍历器行为的目的。

var str = new String("hi");

[...str] // ["h", "i"]

str[Symbol.iterator] = function() {

return {

next: function() {

if (this.\_first) {

this.\_first = false;

return { value: "bye", done: false };

} else {

return { done: true };

}

},

\_first: true

};

};

[...str] // ["bye"]

str // "hi"

上面代码中，字符串 str 的Symbol.iterator方法被修改了，所以扩展运算符（...）返回的值变成了bye，而字符串本身还是hi。

#### Generator 函数的语法

##### 简介

Generator 函数是 ES6 提供的一种异步编程解决方案，语法行为与传统函数完全不同。本章详细介绍 Generator 函数的语法和 API，它的异步编程应用请看《Generator 函数的异步应用》一章。

Generator 函数有多种理解角度。语法上，首先可以把它理解成，Generator 函数是一个状态机，封装了多个内部状态。

执行 Generator 函数会返回一个遍历器对象，也就是说，Generator 函数除了状态机，还是一个遍历器对象生成函数。返回的遍历器对象，可以依次遍历 Generator 函数内部的每一个状态。

形式上，Generator 函数是一个普通函数，但是有两个特征。一是，function关键字与函数名之间有一个星号；二是，函数体内部使用yield表达式，定义不同的内部状态（yield在英语里的意思就是“产出”）。

**yield 表达式**

于 Generator 函数返回的遍历器对象，只有调用next方法才会遍历下一个内部状态，所以其实提供了一种可以暂停执行的函数。yield表达式就是暂停标志。

forEach方法的参数是一个普通函数，不支持yield，需要改成for循环

var arr = [1, [[2, 3], 4], [5, 6]];

var flat = function\* (a) {

a.forEach(function (item) {

if (typeof item !== 'number') {

yield\* flat(item);

} else {

yield item;

}

});

};

for (var f of flat(arr)){

console.log(f);

}

上面代码也会产生句法错误，因为forEach方法的参数是一个普通函数，但是在里面使用了yield表达式（这个函数里面还使用了yield\*表达式，详细介绍见后文）。一种修改方法是改用for循环。

var arr = [1, [[2, 3], 4], [5, 6]];

var flat = function\* (a) {

var length = a.length;

for (var i = 0; i < length; i++) {

var item = a[i];

if (typeof item !== 'number') {

yield\* flat(item);

} else {

yield item;

}

}

};

for (var f of flat(arr)) {

console.log(f);

}

// 1, 2, 3, 4, 5, 6

另外，yield表达式如果用在另一个表达式之中，必须放在圆括号里面。

function\* demo() {

console.log('Hello' + yield); // SyntaxError

console.log('Hello' + yield 123); // SyntaxError

console.log('Hello' + (yield)); // OK

console.log('Hello' + (yield 123)); // OK

}

yield表达式用作函数参数或放在赋值表达式的右边，可以不加括号。

function\* demo() {

foo(yield 'a', yield 'b'); // OK

let input = yield; // OK

}

**与 Iterator 接口的关系** [**§**](http://es6.ruanyifeng.com/#docs/generator#与-Iterator-接口的关系)[**⇧**](http://es6.ruanyifeng.com/#docs/generator)

上一章说过，任意一个对象的Symbol.iterator方法，等于该对象的遍历器生成函数，调用该函数会返回该对象的一个遍历器对象。

由于 Generator 函数就是遍历器生成函数，因此可以把 Generator 赋值给对象的Symbol.iterator属性，从而使得该对象具有 Iterator 接口。

var myIterable = {};

myIterable[Symbol.iterator] = function\* () {

yield 1;

yield 2;

yield 3;

};

[...myIterable] // [1, 2, 3]

上面代码中，Generator 函数赋值给Symbol.iterator属性，从而使得myIterable对象具有了 Iterator 接口，可以被...运算符遍历了。

Generator 函数执行后，返回一个遍历器对象。该对象本身也具有Symbol.iterator属性，执行后返回自身。

function\* gen(){

// some code

}

var g = gen();

g[Symbol.iterator]() === g

// true

上面代码中，gen是一个 Generator 函数，调用它会生成一个遍历器对象g。它的Symbol.iterator属性，也是一个遍历器对象生成函数，执行后返回它自己。

##### next 方法的参数

yield表达式本身没有返回值，或者说总是返回undefined。next方法可以带一个参数，该参数就会被当作上一个yield表达式的返回值。

function\* f() {

for(var i = 0; true; i++) {

var reset = yield i;

if(reset) { i = -1; }

}

}

var g = f();

g.next() // { value: 0, done: false }

g.next() // { value: 1, done: false }

g.next(true) // { value: 0, done: false }

上面代码先定义了一个可以无限运行的 Generator 函数f，如果next方法没有参数，每次运行到yield表达式，变量reset的值总是undefined。当next方法带一个参数true时，变量reset就被重置为这个参数（即true），因此i会等于-1，下一轮循环就会从-1开始递增。

function\* foo(x) {

var y = 2 \* (yield (x + 1));

var z = yield (y / 3);

return (x + y + z);

}

var a = foo(5);

a.next() // Object{value:6, done:false}

a.next() // Object{value:NaN, done:false}

a.next() // Object{value:NaN, done:true}

var b = foo(5);

b.next() // { value:6, done:false }

b.next(12) // { value:8, done:false }

b.next(13) // { value:42, done:true }

上面代码中，第二次运行next方法的时候不带参数，导致 y 的值等于2 \* undefined（即NaN），除以 3 以后还是NaN，因此返回对象的value属性也等于NaN。第三次运行Next方法的时候不带参数，所以z等于undefined，返回对象的value属性等于5 + NaN + undefined，即NaN。

如果向next方法提供参数，返回结果就完全不一样了。上面代码第一次调用b的next方法时，返回x+1的值6；第二次调用next方法，将上一次yield表达式的值设为12，因此y等于24，返回y / 3的值8；第三次调用next方法，将上一次yield表达式的值设为13，因此z等于13，这时x等于5，y等于24，所以return语句的值等于42。

注意，由于next方法的参数表示上一个yield表达式的返回值，所以在第一次使用next方法时，传递参数是无效的。V8 引擎直接忽略第一次使用next方法时的参数，只有从第二次使用next方法开始，参数才是有效的。从语义上讲，第一个next方法用来启动遍历器对象，所以不用带有参数。

再看一个通过next方法的参数，向 Generator 函数内部输入值的例子。

function\* dataConsumer() {

console.log('Started');

console.log(`1. ${yield}`);

console.log(`2. ${yield}`);

return 'result';

}

let genObj = dataConsumer();

genObj.next();

// Started

genObj.next('a')

// 1. a

genObj.next('b')

// 2. b

上面代码是一个很直观的例子，每次通过next方法向 Generator 函数输入值，然后打印出来。

##### for...of 循环 [§](http://es6.ruanyifeng.com/#docs/generator#for---of-循环) [⇧](http://es6.ruanyifeng.com/#docs/generator)

for...of循环可以自动遍历 Generator 函数时生成的Iterator对象，且此时不再需要调用next方法。

function\* foo() {

yield 1;

yield 2;

yield 3;

yield 4;

yield 5;

return 6;

}

for (let v of foo()) {

console.log(v);

}

// 1 2 3 4 5

上面代码使用for...of循环，依次显示 5 个yield表达式的值。这里需要注意，一旦next方法的返回对象的done属性为true，for...of循环就会中止，且不包含该返回对象，所以上面代码的return语句返回的6，不包括在for...of循环之中。

利用for...of循环，可以写出遍历任意对象（object）的方法。原生的 JavaScript 对象没有遍历接口，无法使用for...of循环，通过 Generator 函数为它加上这个接口，就可以用了。

function\* objectEntries(obj) {

let propKeys = Reflect.ownKeys(obj);

for (let propKey of propKeys) {

yield [propKey, obj[propKey]];

}

}

let jane = { first: 'Jane', last: 'Doe' };

for (let [key, value] of objectEntries(jane)) {

console.log(`${key}: ${value}`);

}

// first: Jane

// last: Doe

上面代码中，对象jane原生不具备 Iterator 接口，无法用for...of遍历。这时，我们通过 Generator 函数objectEntries为它加上遍历器接口，就可以用for...of遍历了。

加上遍历器接口的另一种写法是，将 Generator 函数加到对象的Symbol.iterator属性上面。

function\* objectEntries() {

let propKeys = Object.keys(this);

for (let propKey of propKeys) {

yield [propKey, this[propKey]];

}

}

let jane = { first: 'Jane', last: 'Doe' };

jane[Symbol.iterator] = objectEntries;

for (let [key, value] of jane) {

console.log(`${key}: ${value}`);

}

// first: Jane

// last: Doe

除了for...of循环以外，扩展运算符（...）、解构赋值和Array.from方法内部调用的，都是遍历器接口。这意味着，它们都可以将 Generator 函数返回的 Iterator 对象，作为参数。

function\* numbers () {

yield 1

yield 2

return 3

yield 4

}

// 扩展运算符

[...numbers()] // [1, 2]

// Array.from 方法

Array.from(numbers()) // [1, 2]

// 解构赋值

let [x, y] = numbers();

x // 1

y // 2

// for...of 循环

for (let n of numbers()) {

console.log(n)

}

// 1

// 2

##### Generator.prototype.throw()

Generator 函数返回的遍历器对象，都有一个throw方法，可以在函数体外抛出错误，然后在 Generator 函数体内捕获。

var g = function\* () {

try {

yield;

} catch (e) {

console.log('内部捕获', e);

}

};

var i = g();

i.next();

try {

i.throw('a');

i.throw('b');

} catch (e) {

console.log('外部捕获', e);

}

// 内部捕获 a

// 外部捕获 b

上面代码中，遍历器对象i连续抛出两个错误。第一个错误被 Generator 函数体内的catch语句捕获。i第二次抛出错误，由于 Generator 函数内部的catch语句已经执行过了，不会再捕捉到这个错误了，所以这个错误就被抛出了 Generator 函数体，被函数体外的catch语句捕获。

##### Generator.prototype.return() [§](http://es6.ruanyifeng.com/#docs/generator#Generator-prototype-return) [⇧](http://es6.ruanyifeng.com/#docs/generator)

Generator 函数返回的遍历器对象，还有一个return方法，可以返回给定的值，并且终结遍历 Generator 函数。

function\* gen() {

yield 1;

yield 2;

yield 3;

}

var g = gen();

g.next() // { value: 1, done: false }

g.return('foo') // { value: "foo", done: true }

g.next() // { value: undefined, done: true }

如果return方法调用时，不提供参数，则返回值的value属性为undefined。

如果 Generator 函数内部有try...finally代码块，那么return方法会推迟到finally代码块执行完再执行。

function\* numbers () {

yield 1;

try {

yield 2;

yield 3;

} finally {

yield 4;

yield 5;

}

yield 6;

}

var g = numbers();

g.next() // { value: 1, done: false }

g.next() // { value: 2, done: false }

g.return(7) // { value: 4, done: false }

g.next() // { value: 5, done: false }

g.next() // { value: 7, done: true }

上面代码中，调用return方法后，就开始执行finally代码块，然后等到finally代码块执行完，再执行return方法。

##### next()、throw()、return() 的共同点 [§](http://es6.ruanyifeng.com/#docs/generator#next、throw、return-的共同点) [⇧](http://es6.ruanyifeng.com/#docs/generator)

网友 vision57 提出，next()、throw()、return()这三个方法本质上是同一件事，可以放在一起理解。它们的作用都是让 Generator 函数恢复执行，并且使用不同的语句替换yield表达式。

next()是将yield表达式替换成一个值。

const g = function\* (x, y) {

let result = yield x + y;

return result;

};

const gen = g(1, 2);

gen.next(); // Object {value: 3, done: false}

gen.next(1); // Object {value: 1, done: true}

// 相当于将 let result = yield x + y

// 替换成 let result = 1;

上面代码中，第二个next(1)方法就相当于将yield表达式替换成一个值1。如果next方法没有参数，就相当于替换成undefined。

throw()是将yield表达式替换成一个throw语句。

gen.throw(new Error('出错了')); // Uncaught Error: 出错了

// 相当于将 let result = yield x + y

// 替换成 let result = throw(new Error('出错了'));

return()是将yield表达式替换成一个return语句。

gen.return(2); // Object {value: 2, done: true}

// 相当于将 let result = yield x + y

// 替换成 let result = return 2;

##### yield\* 表达式

这个就需要用到yield\*表达式，用来在一个 Generator 函数里面执行另一个 Generator 函数。

function\* bar() {

yield 'x';

yield\* foo();

yield 'y';

}

// 等同于

function\* bar() {

yield 'x';

yield 'a';

yield 'b';

yield 'y';

}

// 等同于

function\* bar() {

yield 'x';

for (let v of foo()) {

yield v;

}

yield 'y';

}

for (let v of bar()){

console.log(v);

}

// "x"

// "a"

// "b"

// "y"

下面是一个稍微复杂的例子，使用yield\*语句遍历完全二叉树。

// 下面是二叉树的构造函数，

// 三个参数分别是左树、当前节点和右树

function Tree(left, label, right) {

this.left = left;

this.label = label;

this.right = right;

}

// 下面是中序（inorder）遍历函数。

// 由于返回的是一个遍历器，所以要用generator函数。

// 函数体内采用递归算法，所以左树和右树要用yield\*遍历

function\* inorder(t) {

if (t) {

yield\* inorder(t.left);

yield t.label;

yield\* inorder(t.right);

}

}

// 下面生成二叉树

function make(array) {

// 判断是否为叶节点

if (array.length == 1) return new Tree(null, array[0], null);

return new Tree(make(array[0]), array[1], make(array[2]));

}

let tree = make([[['a'], 'b', ['c']], 'd', [['e'], 'f', ['g']]]);

// 遍历二叉树

var result = [];

for (let node of inorder(tree)) {

result.push(node);

}

result

// ['a', 'b', 'c', 'd', 'e', 'f', 'g']

##### 作为对象属性的 Generator 函数

如果一个对象的属性是 Generator 函数，可以简写成下面的形式。

let obj = {

\* myGeneratorMethod() {

···

}

};

上面代码中，myGeneratorMethod属性前面有一个星号，表示这个属性是一个 Generator 函数。

它的完整形式如下，与上面的写法是等价的。

let obj = {

myGeneratorMethod: function\* () {

// ···

}

};

##### Generator 函数的this

Generator 函数总是返回一个遍历器，ES6 规定这个遍历器是 Generator 函数的实例，也继承了 Generator 函数的prototype对象上的方法。

function\* g() {}

g.prototype.hello = function () {

return 'hi!';

};

let obj = g();

obj instanceof g // true

obj.hello() // 'hi!'

上面代码表明，Generator 函数g返回的遍历器obj，是g的实例，而且继承了g.prototype。但是，如果把g当作普通的构造函数，并不会生效，因为g返回的总是遍历器对象，而不是this对象。

function\* g() {

this.a = 11;

}

let obj = g();

obj.next();

obj.a // undefined

上面代码中，Generator 函数g在this对象上面添加了一个属性a，但是obj对象拿不到这个属性。

Generator 函数也不能跟new命令一起用，会报错。

function\* F() {

yield this.x = 2;

yield this.y = 3;

}

new F()

// TypeError: F is not a constructor

下面是一个变通方法。首先，生成一个空对象，使用call方法绑定 Generator 函数内部的this。这样，构造函数调用以后，这个空对象就是 Generator 函数的实例对象了。

function\* F() {

this.a = 1;

yield this.b = 2;

yield this.c = 3;

}

var obj = {};

var f = F.call(obj);

f.next(); // Object {value: 2, done: false}

f.next(); // Object {value: 3, done: false}

f.next(); // Object {value: undefined, done: true}

obj.a // 1

obj.b // 2

obj.c // 3

那么，有没有办法让 Generator 函数返回一个正常的对象实例，既可以用next方法，又可以获得正常的this？

下面是一个变通方法。首先，生成一个空对象，使用call方法绑定 Generator 函数内部的this。这样，构造函数调用以后，这个空对象就是 Generator 函数的实例对象了。

function\* F() {

this.a = 1;

yield this.b = 2;

yield this.c = 3;

}

var obj = {};

var f = F.call(obj);

f.next(); // Object {value: 2, done: false}

f.next(); // Object {value: 3, done: false}

f.next(); // Object {value: undefined, done: true}

obj.a // 1

obj.b // 2

obj.c // 3

上面代码中，执行的是遍历器对象f，但是生成的对象实例是obj，有没有办法将这两个对象统一呢？

一个办法就是将obj换成F.prototype。

function\* F() {

this.a = 1;

yield this.b = 2;

yield this.c = 3;

}

var f = F.call(F.prototype);

f.next(); // Object {value: 2, done: false}

f.next(); // Object {value: 3, done: false}

f.next(); // Object {value: undefined, done: true}

f.a // 1

f.b // 2

f.c // 3

再将F改成构造函数，就可以对它执行new命令了。

function\* gen() {

this.a = 1;

yield this.b = 2;

yield this.c = 3;

}

function F() {

return gen.call(gen.prototype);

}

var f = new F();

f.next(); // Object {value: 2, done: false}

f.next(); // Object {value: 3, done: false}

f.next(); // Object {value: undefined, done: true}

f.a // 1

f.b // 2

f.c // 3

##### 含义

**Generator 与状态机**

Generator 是实现状态机的最佳结构。比如，下面的clock函数就是一个状态机。

var ticking = true;

var clock = function() {

if (ticking)

console.log('Tick!');

else

console.log('Tock!');

ticking = !ticking;

}

上面代码的clock函数一共有两种状态（Tick和Tock），每运行一次，就改变一次状态。这个函数如果用 Generator 实现，就是下面这样。

var clock = function\* () {

while (true) {

console.log('Tick!');

yield;

console.log('Tock!');

yield;

}

};

上面的 Generator 实现与 ES5 实现对比，可以看到少了用来保存状态的外部变量ticking，这样就更简洁，更安全（状态不会被非法篡改）、更符合函数式编程的思想，在写法上也更优雅。Generator 之所以可以不用外部变量保存状态，是因为它本身就包含了一个状态信息，即目前是否处于暂停态。

**Generator 与协程**

协程（coroutine）是一种程序运行的方式，可以理解成“协作的线程”或“协作的函数”。协程既可以用单线程实现，也可以用多线程实现。前者是一种特殊的子例程，后者是一种特殊的线程。

**（1）协程与子例程的差异**

从实现上看，在内存中，子例程只使用一个栈（stack），而协程是同时存在多个栈，但只有一个栈是在运行状态，也就是说，协程是以多占用内存为代价，实现多任务的并行。

**（2）协程与普通线程的差异**

不难看出，协程适合用于多任务运行的环境。在这个意义上，它与普通的线程很相似，都有自己的执行上下文、可以分享全局变量。它们的不同之处在于，同一时间可以有多个线程处于运行状态，但是运行的协程只能有一个，其他协程都处于暂停状态。此外，普通的线程是抢先式的，到底哪个线程优先得到资源，必须由运行环境决定，但是协程是合作式的，执行权由协程自己分配。

由于 JavaScript 是单线程语言，只能保持一个调用栈。引入协程以后，每个任务可以保持自己的调用栈。这样做的最大好处，就是抛出错误的时候，可以找到原始的调用栈。不至于像异步操作的回调函数那样，一旦出错，原始的调用栈早就结束。

Generator 函数是 ES6 对协程的实现，但属于不完全实现。Generator 函数被称为“半协程”（semi-coroutine），意思是只有 Generator 函数的调用者，才能将程序的执行权还给 Generator 函数。如果是完全执行的协程，任何函数都可以让暂停的协程继续执行。

如果将 Generator 函数当作协程，完全可以将多个需要互相协作的任务写成 Generator 函数，它们之间使用yield表示式交换控制权。

**Generator 与上下文**

JavaScript 代码运行时，会产生一个全局的上下文环境（context，又称运行环境），包含了当前所有的变量和对象。然后，执行函数（或块级代码）的时候，又会在当前上下文环境的上层，产生一个函数运行的上下文，变成当前（active）的上下文，由此形成一个上下文环境的堆栈（context stack）。

这个堆栈是“后进先出”的数据结构，最后产生的上下文环境首先执行完成，退出堆栈，然后再执行完成它下层的上下文，直至所有代码执行完成，堆栈清空。

Generator 函数不是这样，它执行产生的上下文环境，一旦遇到yield命令，就会暂时退出堆栈，但是并不消失，里面的所有变量和对象会冻结在当前状态。等到对它执行next命令时，这个上下文环境又会重新加入调用栈，冻结的变量和对象恢复执行。

function\* gen() {

yield 1;

return 2;

}

let g = gen();

console.log(

g.next().value,

g.next().value,

);

上面代码中，第一次执行g.next()时，Generator 函数gen的上下文会加入堆栈，即开始运行gen内部的代码。等遇到yield 1时，gen上下文退出堆栈，内部状态冻结。第二次执行g.next()时，gen上下文重新加入堆栈，变成当前的上下文，重新恢复执行。

##### 应用 [§](http://es6.ruanyifeng.com/#docs/generator#应用) [⇧](http://es6.ruanyifeng.com/#docs/generator)

Generator 可以暂停函数执行，返回任意表达式的值。这种特点使得 Generator 有多种应用场景。

###### 异步操作的同步化表达

Generator 函数的暂停执行的效果，意味着可以把异步操作写在yield表达式里面，等到调用next方法时再往后执行。这实际上等同于不需要写回调函数了，因为异步操作的后续操作可以放在yield表达式下面，反正要等到调用next方法时再执行。所以，Generator 函数的一个重要实际意义就是用来处理异步操作，改写回调函数。

function\* loadUI() {

showLoadingScreen();

yield loadUIDataAsynchronously();

hideLoadingScreen();

}

var loader = loadUI();

// 加载UI

loader.next()

// 卸载UI

loader.next()

上面代码中，第一次调用loadUI函数时，该函数不会执行，仅返回一个遍历器。下一次对该遍历器调用next方法，则会显示Loading界面（showLoadingScreen），并且异步加载数据（loadUIDataAsynchronously）。等到数据加载完成，再一次使用next方法，则会隐藏Loading界面。可以看到，这种写法的好处是所有Loading界面的逻辑，都被封装在一个函数，按部就班非常清晰。

Ajax 是典型的异步操作，通过 Generator 函数部署 Ajax 操作，可以用同步的方式表达。

function\* main() {

var result = yield request("http://some.url");

var resp = JSON.parse(result);

console.log(resp.value);

}

function request(url) {

makeAjaxCall(url, function(response){

it.next(response);

});

}

var it = main();

it.next();

上面代码的main函数，就是通过 Ajax 操作获取数据。可以看到，除了多了一个yield，它几乎与同步操作的写法完全一样。注意，makeAjaxCall函数中的next方法，必须加上response参数，因为yield表达式，本身是没有值的，总是等于undefined。

下面是另一个例子，通过 Generator 函数逐行读取文本文件。

function\* numbers() {

let file = new FileReader("numbers.txt");

try {

while(!file.eof) {

yield parseInt(file.readLine(), 10);

}

} finally {

file.close();

}

}

上面代码打开文本文件，使用yield表达式可以手动逐行读取文件。

###### 控制流管理

Generator 函数可以进一步改善代码运行流程。

function\* longRunningTask(value1) {

try {

var value2 = yield step1(value1);

var value3 = yield step2(value2);

var value4 = yield step3(value3);

var value5 = yield step4(value4);

// Do something with value4

} catch (e) {

// Handle any error from step1 through step4

}

}

然后，使用一个函数，按次序自动执行所有步骤。

scheduler(longRunningTask(initialValue));

function scheduler(task) {

var taskObj = task.next(task.value);

// 如果Generator函数未结束，就继续调用

if (!taskObj.done) {

task.value = taskObj.value

scheduler(task);

}

}

注意，上面这种做法，只适合同步操作，即所有的task都必须是同步的，不能有异步操作。因为这里的代码一得到返回值，就继续往下执行，没有判断异步操作何时完成。如果要控制异步的操作流程，详见后面的《异步操作》一章。

下面，利用for...of循环会自动依次执行yield命令的特性，提供一种更一般的控制流管理的方法。

let steps = [step1Func, step2Func, step3Func];

function\* iterateSteps(steps){

for (var i=0; i< steps.length; i++){

var step = steps[i];

yield step();

}

}

上面代码中，数组steps封装了一个任务的多个步骤，Generator 函数iterateSteps则是依次为这些步骤加上yield命令。

将任务分解成步骤之后，还可以将项目分解成多个依次执行的任务。

let jobs = [job1, job2, job3];

function\* iterateJobs(jobs){

for (var i=0; i< jobs.length; i++){

var job = jobs[i];

yield\* iterateSteps(job.steps);

}

}

上面代码中，数组jobs封装了一个项目的多个任务，Generator 函数iterateJobs则是依次为这些任务加上yield\*命令。

最后，就可以用for...of循环一次性依次执行所有任务的所有步骤。

for (var step of iterateJobs(jobs)){

console.log(step.id);

}

再次提醒，上面的做法只能用于所有步骤都是同步操作的情况，不能有异步操作的步骤。如果想要依次执行异步的步骤，必须使用后面的《异步操作》一章介绍的方法。

for...of的本质是一个while循环，所以上面的代码实质上执行的是下面的逻辑。

var it = iterateJobs(jobs);

var res = it.next();

while (!res.done){

var result = res.value;

// ...

res = it.next();

}

###### 部署 Iterator 接口

function\* iterEntries(obj) {

let keys = Object.keys(obj);

for (let i=0; i < keys.length; i++) {

let key = keys[i];

yield [key, obj[key]];

}

}

let myObj = { foo: 3, bar: 7 };

for (let [key, value] of iterEntries(myObj)) {

console.log(key, value);

}

// foo 3

// bar 7

上述代码中，myObj是一个普通对象，通过iterEntries函数，就有了 Iterator 接口。也就是说，可以在任意对象上部署next方法。

下面是一个对数组部署 Iterator 接口的例子，尽管数组原生具有这个接口。

function\* makeSimpleGenerator(array){

var nextIndex = 0;

while(nextIndex < array.length){

yield array[nextIndex++];

}

}

var gen = makeSimpleGenerator(['yo', 'ya']);

gen.next().value // 'yo'

gen.next().value // 'ya'

gen.next().done // true

###### 作为数据结构 [§](http://es6.ruanyifeng.com/#docs/generator#（4）作为数据结构) [⇧](http://es6.ruanyifeng.com/#docs/generator)

Generator 可以看作是数据结构，更确切地说，可以看作是一个数组结构，因为 Generator 函数可以返回一系列的值，这意味着它可以对任意表达式，提供类似数组的接口。

function\* doStuff() {

yield fs.readFile.bind(null, 'hello.txt');

yield fs.readFile.bind(null, 'world.txt');

yield fs.readFile.bind(null, 'and-such.txt');

}

上面代码就是依次返回三个函数，但是由于使用了 Generator 函数，导致可以像处理数组那样，处理这三个返回的函数。

for (task of doStuff()) {

// task是一个函数，可以像回调函数那样使用它

}

实际上，如果用 ES5 表达，完全可以用数组模拟 Generator 的这种用法。

function doStuff() {

return [

fs.readFile.bind(null, 'hello.txt'),

fs.readFile.bind(null, 'world.txt'),

fs.readFile.bind(null, 'and-such.txt')

];

}

上面的函数，可以用一模一样的for...of循环处理！两相一比较，就不难看出 Generator 使得数据或者操作，具备了类似数组的接口。

#### Generator 函数的异步应用

##### 传统方法

ES6 诞生以前，异步编程的方法，大概有下面四种。

* 回调函数
* 事件监听
* 发布/订阅
* Promise 对象

Generator 函数将 JavaScript 异步编程带入了一个全新的阶段。

##### 基本概念

**异步：**

所谓"异步"，简单说就是一个任务不是连续完成的，可以理解成该任务被人为分成两段，先执行第一段，然后转而执行其他任务，等做好了准备，再回过头执行第二段。

比如，有一个任务是读取文件进行处理，任务的第一段是向操作系统发出请求，要求读取文件。然后，程序执行其他任务，等到操作系统返回文件，再接着执行任务的第二段（处理文件）。这种不连续的执行，就叫做异步。

**同步：**

相应地，连续的执行就叫做同步。由于是连续执行，不能插入其他任务，所以操作系统从硬盘读取文件的这段时间，程序只能干等着。

**回调函数 ：**

JavaScript 语言对异步编程的实现，就是回调函数。所谓回调函数，就是把任务的第二段单独写在一个函数里面，等到重新执行这个任务的时候，就直接调用这个函数。回调函数的英语名字callback，直译过来就是"重新调用"。

读取文件进行处理，是这样写的。

fs.readFile('/etc/passwd', 'utf-8', function (err, data) {

if (err) throw err;

console.log(data);

});

上面代码中，readFile函数的第三个参数，就是回调函数，也就是任务的第二段。等到操作系统返回了/etc/passwd这个文件以后，回调函数才会执行。

一个有趣的问题是，为什么 Node 约定，回调函数的第一个参数，必须是错误对象err（如果没有错误，该参数就是null）？

原因是执行分成两段，第一段执行完以后，任务所在的上下文环境就已经结束了。在这以后抛出的错误，原来的上下文环境已经无法捕捉，只能当作参数，传入第二段。

**Promise**

Promise 对象就是为了解决这个问题而提出的。它不是新的语法功能，而是一种新的写法，允许将回调函数的嵌套，改成链式调用。采用 Promise，连续读取多个文件，写法如下。

var readFile = require('fs-readfile-promise');

readFile(fileA)

.then(function (data) {

console.log(data.toString());

})

.then(function () {

return readFile(fileB);

})

.then(function (data) {

console.log(data.toString());

})

.catch(function (err) {

console.log(err);

});

上面代码中，我使用了fs-readfile-promise模块，它的作用就是返回一个 Promise 版本的readFile函数。Promise 提供then方法加载回调函数，catch方法捕捉执行过程中抛出的错误。

可以看到，Promise 的写法只是回调函数的改进，使用then方法以后，异步任务的两段执行看得更清楚了，除此以外，并无新意。

##### Generator 函数

**协成**

传统的编程语言，早有异步编程的解决方案（其实是多任务的解决方案）。其中有一种叫做"协程"（coroutine），意思是多个线程互相协作，完成异步任务。

举例来说，读取文件的协程写法如下。

function\* asyncJob() {

// ...其他代码

var f = yield readFile(fileA);

// ...其他代码

}

上面代码的函数asyncJob是一个协程，它的奥妙就在其中的yield命令。它表示执行到此处，执行权将交给其他协程。也就是说，yield命令是异步两个阶段的分界线。

**协程的 Generator 函数实现**

整个 Generator 函数就是一个封装的异步任务，或者说是异步任务的容器。异步操作需要暂停的地方，都用yield语句注明。Generator 函数的执行方法如下。

function\* gen(x) {

var y = yield x + 2;

return y;

}

var g = gen(1);

g.next() // { value: 3, done: false }

g.next() // { value: undefined, done: true }

**异步任务的封装**

下面看看如何使用 Generator 函数，执行一个真实的异步任务。

var fetch = require('node-fetch');

function\* gen(){

var url = 'https://api.github.com/users/github';

var result = yield fetch(url);

console.log(result.bio);

}

上面代码中，Generator 函数封装了一个异步操作，该操作先读取一个远程接口，然后从 JSON 格式的数据解析信息。就像前面说过的，这段代码非常像同步操作，除了加上了yield命令。

执行这段代码的方法如下。

var g = gen();

var result = g.next();

result.value.then(function(data){

return data.json();

}).then(function(data){

g.next(data);

});

上面代码中，首先执行 Generator 函数，获取遍历器对象，然后使用next方法（第二行），执行异步任务的第一阶段。由于Fetch模块返回的是一个 Promise 对象，因此要用then方法调用下一个next方法。

可以看到，虽然 Generator 函数将异步操作表示得很简洁，但是流程管理却不方便（即何时执行第一阶段、何时执行第二阶段）。

##### Thunk 函数

Thunk 函数是自动执行 Generator 函数的一种方法。

**Thunk 函数的含义**

编译器的“传名调用”实现，往往是将参数放到一个临时函数之中，再将这个临时函数传入函数体。这个临时函数就叫做 Thunk 函数。

function f(m) {

return m \* 2;

}

f(x + 5);

// 等同于

var thunk = function () {

return x + 5;

};

function f(thunk) {

return thunk() \* 2;

}

上面代码中，函数 f 的参数x + 5被一个函数替换了。凡是用到原参数的地方，对Thunk函数求值即可。

这就是 Thunk 函数的定义，它是“传名调用”的一种实现策略，用来替换某个表达式。

**JavaScript 语言的 Thunk 函数**

JavaScript 语言是传值调用，它的 Thunk 函数含义有所不同。在 JavaScript 语言中，Thunk 函数替换的不是表达式，而是多参数函数，将其替换成一个只接受回调函数作为参数的单参数函数。

// 正常版本的readFile（多参数版本）

fs.readFile(fileName, callback);

// Thunk版本的readFile（单参数版本）

var Thunk = function (fileName) {

return function (callback) {

return fs.readFile(fileName, callback);

};

};

var readFileThunk = Thunk(fileName);

readFileThunk(callback);

JavaScript 语言是传值调用，它的 Thunk 函数含义有所不同。在 JavaScript 语言中，Thunk 函数替换的不是表达式，而是多参数函数，将其替换成一个只接受回调函数作为参数的单参数函数。

// 正常版本的readFile（多参数版本）

fs.readFile(fileName, callback);

// Thunk版本的readFile（单参数版本）

var Thunk = function (fileName) {

return function (callback) {

return fs.readFile(fileName, callback);

};

};

var readFileThunk = Thunk(fileName);

readFileThunk(callback);

上面代码中，fs模块的readFile方法是一个多参数函数，两个参数分别为文件名和回调函数。经过转换器处理，它变成了一个单参数函数，只接受回调函数作为参数。这个单参数版本，就叫做 Thunk 函数。

任何函数，只要参数有回调函数，就能写成 Thunk 函数的形式。下面是一个简单的 Thunk 函数转换器。

// ES5版本

var Thunk = function(fn){

return function (){

var args = Array.prototype.slice.call(arguments);

return function (callback){

args.push(callback);

return fn.apply(this, args);

}

};

};

// ES6版本

const Thunk = function(fn) {

return function (...args) {

return function (callback) {

return fn.call(this, ...args, callback);

}

};

};

使用上面的转换器，生成fs.readFile的 Thunk 函数。

var readFileThunk = Thunk(fs.readFile);

readFileThunk(fileA)(callback);

下面是另一个完整的例子。

function f(a, cb) {

cb(a);

}

const ft = Thunk(f);

ft(1)(console.log) // 1

**Thunkify 模块**

生产环境的转换器，建议使用 Thunkify 模块。

首先是安装。

$ npm install thunkify

使用方式如下。

var thunkify = require('thunkify');

var fs = require('fs');

var read = thunkify(fs.readFile);

read('package.json')(function(err, str){

// ...

});

Thunkify 的源码与上一节那个简单的转换器非常像。

function thunkify(fn) {

return function() {

var args = new Array(arguments.length);

var ctx = this;

for (var i = 0; i < args.length; ++i) {

args[i] = arguments[i];

}

return function (done) {

var called;

args.push(function () {

if (called) return;

called = true;

done.apply(null, arguments);

});

try {

fn.apply(ctx, args);

} catch (err) {

done(err);

}

}

}

};

它的源码主要多了一个检查机制，变量called确保回调函数只运行一次。这样的设计与下文的 Generator 函数相关。请看下面的例子。

function f(a, b, callback){

var sum = a + b;

callback(sum);

callback(sum);

}

var ft = thunkify(f);

var print = console.log.bind(console);

ft(1, 2)(print);

// 3

上面代码中，由于thunkify只允许回调函数执行一次，所以只输出一行结果。

**Generator 函数的流程管理**

你可能会问， Thunk 函数有什么用？回答是以前确实没什么用，但是 ES6 有了 Generator 函数，Thunk 函数现在可以用于 Generator 函数的自动流程管理。

Generator 函数可以自动执行。

function\* gen() {

// ...

}

var g = gen();

var res = g.next();

while(!res.done){

console.log(res.value);

res = g.next();

}

上面代码中，Generator 函数gen会自动执行完所有步骤。

但是，这不适合异步操作。如果必须保证前一步执行完，才能执行后一步，上面的自动执行就不可行。这时，Thunk 函数就能派上用处。以读取文件为例。下面的 Generator 函数封装了两个异步操作。

var fs = require('fs');

var thunkify = require('thunkify');

var readFileThunk = thunkify(fs.readFile);

var gen = function\* (){

var r1 = yield readFileThunk('/etc/fstab');

console.log(r1.toString());

var r2 = yield readFileThunk('/etc/shells');

console.log(r2.toString());

};

上面代码中，yield命令用于将程序的执行权移出 Generator 函数，那么就需要一种方法，将执行权再交还给 Generator 函数。

这种方法就是 Thunk 函数，因为它可以在回调函数里，将执行权交还给 Generator 函数。为了便于理解，我们先看如何手动执行上面这个 Generator 函数。

var g = gen();

var r1 = g.next();

r1.value(function (err, data) {

if (err) throw err;

var r2 = g.next(data);

r2.value(function (err, data) {

if (err) throw err;

g.next(data);

});

});

上面代码中，变量g是 Generator 函数的内部指针，表示目前执行到哪一步。next方法负责将指针移动到下一步，并返回该步的信息（value属性和done属性）。

仔细查看上面的代码，可以发现 Generator 函数的执行过程，其实是将同一个回调函数，反复传入next方法的value属性。这使得我们可以用递归来自动完成这个过程。

**Thunk 函数的自动流程管理**

Thunk 函数真正的威力，在于可以自动执行 Generator 函数。下面就是一个基于 Thunk 函数的 Generator 执行器。

function run(fn) {

var gen = fn();

function next(err, data) {

var result = gen.next(data);

if (result.done) return;

result.value(next);

}

next();

}

function\* g() {

// ...

}

run(g);

上面代码的run函数，就是一个 Generator 函数的自动执行器。内部的next函数就是 Thunk 的回调函数。next函数先将指针移到 Generator 函数的下一步（gen.next方法），然后判断 Generator 函数是否结束（result.done属性），如果没结束，就将next函数再传入 Thunk 函数（result.value属性），否则就直接退出。

有了这个执行器，执行 Generator 函数方便多了。不管内部有多少个异步操作，直接把 Generator 函数传入run函数即可。当然，前提是每一个异步操作，都要是 Thunk 函数，也就是说，跟在yield命令后面的必须是 Thunk 函数。

var g = function\* (){

var f1 = yield readFileThunk('fileA');

var f2 = yield readFileThunk('fileB');

// ...

var fn = yield readFileThunk('fileN');

};

run(g);

上面代码中，函数g封装了n个异步的读取文件操作，只要执行run函数，这些操作就会自动完成。这样一来，异步操作不仅可以写得像同步操作，而且一行代码就可以执行。

Thunk 函数并不是 Generator 函数自动执行的唯一方案。因为自动执行的关键是，必须有一种机制，自动控制 Generator 函数的流程，接收和交还程序的执行权。回调函数可以做到这一点，Promise 对象也可以做到这一点。

##### co 模块

**基本用法**

[co 模块](https://github.com/tj/co)是著名程序员 TJ Holowaychuk 于 2013 年 6 月发布的一个小工具，用于 Generator 函数的自动执行。

下面是一个 Generator 函数，用于依次读取两个文件。

var gen = function\* () {

var f1 = yield readFile('/etc/fstab');

var f2 = yield readFile('/etc/shells');

console.log(f1.toString());

console.log(f2.toString());

};

co 模块可以让你不用编写 Generator 函数的执行器。

var co = require('co');

co(gen);

上面代码中，Generator 函数只要传入co函数，就会自动执行。

co函数返回一个Promise对象，因此可以用then方法添加回调函数。

co(gen).then(function (){

console.log('Generator 函数执行完成');

});

上面代码中，等到 Generator 函数执行结束，就会输出一行提示。

**co 模块的原理**

为什么 co 可以自动执行 Generator 函数？

前面说过，Generator 就是一个异步操作的容器。它的自动执行需要一种机制，当异步操作有了结果，能够自动交回执行权。

两种方法可以做到这一点。

（1）回调函数。将异步操作包装成 Thunk 函数，在回调函数里面交回执行权。

（2）Promise 对象。将异步操作包装成 Promise 对象，用then方法交回执行权。

co 模块其实就是将两种自动执行器（Thunk 函数和 Promise 对象），包装成一个模块。使用 co 的前提条件是，Generator 函数的yield命令后面，只能是 Thunk 函数或 Promise 对象。如果数组或对象的成员，全部都是 Promise 对象，也可以使用 co，详见后文的例子。

上一节已经介绍了基于 Thunk 函数的自动执行器。下面来看，基于 Promise 对象的自动执行器。这是理解 co 模块必须的。

**基于 Promise 对象的自动执行**

还是沿用上面的例子。首先，把fs模块的readFile方法包装成一个 Promise 对象。

var fs = require('fs');

var readFile = function (fileName){

return new Promise(function (resolve, reject){

fs.readFile(fileName, function(error, data){

if (error) return reject(error);

resolve(data);

});

});

};

var gen = function\* (){

var f1 = yield readFile('/etc/fstab');

var f2 = yield readFile('/etc/shells');

console.log(f1.toString());

console.log(f2.toString());

};

然后，手动执行上面的 Generator 函数。

var g = gen();

g.next().value.then(function(data){

g.next(data).value.then(function(data){

g.next(data);

});

});

手动执行其实就是用then方法，层层添加回调函数。理解了这一点，就可以写出一个自动执行器。

function run(gen){

var g = gen();

function next(data){

var result = g.next(data);

if (result.done) return result.value;

result.value.then(function(data){

next(data);

});

}

next();

}

run(gen);

上面代码中，只要 Generator 函数还没执行到最后一步，next函数就调用自身，以此实现自动执行。

**co 模块的源码**

co 就是上面那个自动执行器的扩展，它的源码只有几十行，非常简单。

首先，co 函数接受 Generator 函数作为参数，返回一个 Promise 对象。

function co(gen) {

var ctx = this;

return new Promise(function(resolve, reject) {

});

}

在返回的 Promise 对象里面，co 先检查参数gen是否为 Generator 函数。如果是，就执行该函数，得到一个内部指针对象；如果不是就返回，并将 Promise 对象的状态改为resolved。

function co(gen) {

var ctx = this;

return new Promise(function(resolve, reject) {

if (typeof gen === 'function') gen = gen.call(ctx);

if (!gen || typeof gen.next !== 'function') return resolve(gen);

});

}

接着，co 将 Generator 函数的内部指针对象的next方法，包装成onFulfilled函数。这主要是为了能够捕捉抛出的错误。

function co(gen) {

var ctx = this;

return new Promise(function(resolve, reject) {

if (typeof gen === 'function') gen = gen.call(ctx);

if (!gen || typeof gen.next !== 'function') return resolve(gen);

onFulfilled();

function onFulfilled(res) {

var ret;

try {

ret = gen.next(res);

} catch (e) {

return reject(e);

}

next(ret);

}

});

}

最后，就是关键的next函数，它会反复调用自身。

function next(ret) {

if (ret.done) return resolve(ret.value);

var value = toPromise.call(ctx, ret.value);

if (value && isPromise(value)) return value.then(onFulfilled, onRejected);

return onRejected(

new TypeError(

'You may only yield a function, promise, generator, array, or object, '

+ 'but the following object was passed: "'

+ String(ret.value)

+ '"'

)

);

}

上面代码中，next函数的内部代码，一共只有四行命令。

第一行，检查当前是否为 Generator 函数的最后一步，如果是就返回。

第二行，确保每一步的返回值，是 Promise 对象。

第三行，使用then方法，为返回值加上回调函数，然后通过onFulfilled函数再次调用next函数。

第四行，在参数不符合要求的情况下（参数非 Thunk 函数和 Promise 对象），将 Promise 对象的状态改为rejected，从而终止执行。

**处理并发的异步操作**

co 支持并发的异步操作，即允许某些操作同时进行，等到它们全部完成，才进行下一步。

这时，要把并发的操作都放在数组或对象里面，跟在yield语句后面。

// 数组的写法

co(function\* () {

var res = yield [

Promise.resolve(1),

Promise.resolve(2)

];

console.log(res);

}).catch(onerror);

// 对象的写法

co(function\* () {

var res = yield {

1: Promise.resolve(1),

2: Promise.resolve(2),

};

console.log(res);

}).catch(onerror);

下面是另一个例子。

co(function\* () {

var values = [n1, n2, n3];

yield values.map(somethingAsync);

});

function\* somethingAsync(x) {

// do something async

return y

}

上面的代码允许并发三个somethingAsync异步操作，等到它们全部完成，才会进行下一步。

**实例：处理 Stream**

Node 提供 Stream 模式读写数据，特点是一次只处理数据的一部分，数据分成一块块依次处理，就好像“数据流”一样。这对于处理大规模数据非常有利。Stream 模式使用 EventEmitter API，会释放三个事件。

* data事件：下一块数据块已经准备好了。
* end事件：整个“数据流”处理“完了。
* error事件：发生错误。

使用Promise.race()函数，可以判断这三个事件之中哪一个最先发生，只有当data事件最先发生时，才进入下一个数据块的处理。从而，我们可以通过一个while循环，完成所有数据的读取。

const co = require('co');

const fs = require('fs');

const stream = fs.createReadStream('./les\_miserables.txt');

let valjeanCount = 0;

co(function\*() {

while(true) {

const res = yield Promise.race([

new Promise(resolve => stream.once('data', resolve)),

new Promise(resolve => stream.once('end', resolve)),

new Promise((resolve, reject) => stream.once('error', reject))

]);

if (!res) {

break;

}

stream.removeAllListeners('data');

stream.removeAllListeners('end');

stream.removeAllListeners('error');

valjeanCount += (res.toString().match(/valjean/ig) || []).length;

}

console.log('count:', valjeanCount); // count: 1120

});

上面代码采用 Stream 模式读取《悲惨世界》的文本文件，对于每个数据块都使用stream.once方法，在data、end、error三个事件上添加一次性回调函数。变量res只有在data事件发生时才有值，然后累加每个数据块之中valjean这个词出现的次数。

#### Proxy

##### 概念

Proxy 用于修改某些操作的默认行为，等同于在语言层面做出修改，所以属于一种“元编程”（meta programming），即对编程语言进行编程。

Proxy 可以理解成，在目标对象之前架设一层“拦截”，外界对该对象的访问，都必须先通过这层拦截，因此提供了一种机制，可以对外界的访问进行过滤和改写。Proxy 这个词的原意是代理，用在这里表示由它来“代理”某些操作，可以译为“代理器”。

var obj = new Proxy({}, {

get: function (target, key, receiver) {

console.log(`getting ${key}!`);

return Reflect.get(target, key, receiver);

},

set: function (target, key, value, receiver) {

console.log(`setting ${key}!`);

return Reflect.set(target, key, value, receiver);

}

});

上面代码对一个空对象架设了一层拦截，重定义了属性的读取（get）和设置（set）行为。这里暂时先不解释具体的语法，只看运行结果。对设置了拦截行为的对象obj，去读写它的属性，就会得到下面的结果。

obj.count = 1

// setting count!

++obj.count

// getting count!

// setting count!

// 2

上面代码说明，Proxy 实际上重载（overload）了点运算符，即用自己的定义覆盖了语言的原始定义。

ES6 原生提供 Proxy 构造函数，用来生成 Proxy 实例。

var proxy = new Proxy(target, handler);

Proxy 对象的所有用法，都是上面这种形式，不同的只是handler参数的写法。其中，new Proxy()表示生成一个Proxy实例，target参数表示所要拦截的目标对象，handler参数也是一个对象，用来定制拦截行为。

下面是另一个拦截读取属性行为的例子。

var proxy = new Proxy({}, {

get: function(target, property) {

return 35;

}

});

proxy.time // 35

proxy.name // 35

proxy.title // 35

上面代码中，作为构造函数，Proxy接受两个参数。第一个参数是所要代理的目标对象（上例是一个空对象），即如果没有Proxy的介入，操作原来要访问的就是这个对象；第二个参数是一个配置对象，对于每一个被代理的操作，需要提供一个对应的处理函数，该函数将拦截对应的操作。比如，上面代码中，配置对象有一个get方法，用来拦截对目标对象属性的访问请求。get方法的两个参数分别是目标对象和所要访问的属性。可以看到，由于拦截函数总是返回35，所以访问任何属性都得到35。

注意，要使得Proxy起作用，必须针对Proxy实例（上例是proxy对象）进行操作，而不是针对目标对象（上例是空对象）进行操作。

如果handler没有设置任何拦截，那就等同于直接通向原对象。

var target = {};

var handler = {};

var proxy = new Proxy(target, handler);

proxy.a = 'b';

target.a // "b"

上面代码中，handler是一个空对象，没有任何拦截效果，访问proxy就等同于访问target。

一个技巧是将 Proxy 对象，设置到object.proxy属性，从而可以在object对象上调用。

var object = { proxy: new Proxy(target, handler) };

Proxy 实例也可以作为其他对象的原型对象。

var proxy = new Proxy({}, {

get: function(target, property) {

return 35;

}

});

let obj = Object.create(proxy);

obj.time // 35

上面代码中，proxy对象是obj对象的原型，obj对象本身并没有time属性，所以根据原型链，会在proxy对象上读取该属性，导致被拦截。

同一个拦截器函数，可以设置拦截多个操作。

var handler = {

get: function(target, name) {

if (name === 'prototype') {

return Object.prototype;

}

return 'Hello, ' + name;

},

apply: function(target, thisBinding, args) {

return args[0];

},

construct: function(target, args) {

return {value: args[1]};

}

};

var fproxy = new Proxy(function(x, y) {

return x + y;

}, handler);

fproxy(1, 2) // 1

new fproxy(1, 2) // {value: 2}

fproxy.prototype === Object.prototype // true

fproxy.foo === "Hello, foo" // true

对于可以设置、但没有设置拦截的操作，则直接落在目标对象上，按照原先的方式产生结果。

下面是 Proxy 支持的拦截操作一览，一共 13 种。

* get(target, propKey, receiver)：拦截对象属性的读取，比如proxy.foo和proxy['foo']。
* set(target, propKey, value, receiver)：拦截对象属性的设置，比如proxy.foo = v或proxy['foo'] = v，返回一个布尔值。
* has(target, propKey)：拦截propKey in proxy的操作，返回一个布尔值。
* deleteProperty(target, propKey)：拦截delete proxy[propKey]的操作，返回一个布尔值。
* ownKeys(target)：拦截Object.getOwnPropertyNames(proxy)、Object.getOwnPropertySymbols(proxy)、Object.keys(proxy)，返回一个数组。该方法返回目标对象所有自身的属性的属性名，而Object.keys()的返回结果仅包括目标对象自身的可遍历属性。
* getOwnPropertyDescriptor(target, propKey)：拦截Object.getOwnPropertyDescriptor(proxy, propKey)，返回属性的描述对象。
* defineProperty(target, propKey, propDesc)：拦截Object.defineProperty(proxy, propKey, propDesc）、Object.defineProperties(proxy, propDescs)，返回一个布尔值。
* preventExtensions(target)：拦截Object.preventExtensions(proxy)，返回一个布尔值。
* getPrototypeOf(target)：拦截Object.getPrototypeOf(proxy)，返回一个对象。
* isExtensible(target)：拦截Object.isExtensible(proxy)，返回一个布尔值。
* setPrototypeOf(target, proto)：拦截Object.setPrototypeOf(proxy, proto)，返回一个布尔值。如果目标对象是函数，那么还有两种额外操作可以拦截。
* apply(target, object, args)：拦截 Proxy 实例作为函数调用的操作，比如proxy(...args)、proxy.call(object, ...args)、proxy.apply(...)。
* construct(target, args)：拦截 Proxy 实例作为构造函数调用的操作，比如new proxy(...args)。

##### Proxy 实例的方法

###### get()

get方法用于拦截某个属性的读取操作，可以接受三个参数，依次为目标对象、属性名和 proxy 实例本身（严格地说，是操作行为所针对的对象），其中最后一个参数可选。

get方法的用法，上文已经有一个例子，下面是另一个拦截读取操作的例子。

var person = {

name: "张三"

};

var proxy = new Proxy(person, {

get: function(target, property) {

if (property in target) {

return target[property];

} else {

throw new ReferenceError("Property \"" + property + "\" does not exist.");

}

}

});

proxy.name // "张三"

proxy.age // 抛出一个错误

上面代码表示，如果访问目标对象不存在的属性，会抛出一个错误。如果没有这个拦截函数，访问不存在的属性，只会返回undefined。

get方法可以继承。

let proto = new Proxy({}, {

get(target, propertyKey, receiver) {

console.log('GET ' + propertyKey);

return target[propertyKey];

}

});

let obj = Object.create(proto);

obj.foo // "GET foo"

上面代码中，拦截操作定义在Prototype对象上面，所以如果读取obj对象继承的属性时，拦截会生效。

下面的例子使用get拦截，实现数组读取负数的索引。

function createArray(...elements) {

let handler = {

get(target, propKey, receiver) {

let index = Number(propKey);

if (index < 0) {

propKey = String(target.length + index);

}

return Reflect.get(target, propKey, receiver);

}

};

let target = [];

target.push(...elements);

return new Proxy(target, handler);

}

let arr = createArray('a', 'b', 'c');

arr[-1] // c

上面代码中，数组的位置参数是-1，就会输出数组的倒数第一个成员。

利用 Proxy，可以将读取属性的操作（get），转变为执行某个函数，从而实现属性的链式操作。

var pipe = (function () {

return function (value) {

var funcStack = [];

var oproxy = new Proxy({} , {

get : function (pipeObject, fnName) {

if (fnName === 'get') {

return funcStack.reduce(function (val, fn) {

return fn(val);

},value);

}

funcStack.push(window[fnName]);

return oproxy;

}

});

return oproxy;

}

}());

var double = n => n \* 2;

var pow = n => n \* n;

var reverseInt = n => n.toString().split("").reverse().join("") | 0;

pipe(3).double.pow.reverseInt.get; // 63

上面代码设置 Proxy 以后，达到了将函数名链式使用的效果。

下面的例子则是利用get拦截，实现一个生成各种 DOM 节点的通用函数dom。

const dom = new Proxy({}, {

get(target, property) {

return function(attrs = {}, ...children) {

const el = document.createElement(property);

for (let prop of Object.keys(attrs)) {

el.setAttribute(prop, attrs[prop]);

}

for (let child of children) {

if (typeof child === 'string') {

child = document.createTextNode(child);

}

el.appendChild(child);

}

return el;

}

}

});

const el = dom.div({},

'Hello, my name is ',

dom.a({href: '//example.com'}, 'Mark'),

'. I like:',

dom.ul({},

dom.li({}, 'The web'),

dom.li({}, 'Food'),

dom.li({}, '…actually that\'s it')

)

);

document.body.appendChild(el);

下面是一个get方法的第三个参数的例子。

const proxy = new Proxy({}, {

get: function(target, property, receiver) {

return receiver;

}

});

proxy.getReceiver === proxy // true

上面代码中，get方法的第三个参数receiver，总是为当前的 Proxy 实例。

如果一个属性不可配置（configurable）和不可写（writable），则该属性不能被代理，通过 Proxy 对象访问该属性会报错。

const target = Object.defineProperties({}, {

foo: {

value: 123,

writable: false,

configurable: false

},

});

const handler = {

get(target, propKey) {

return 'abc';

}

};

const proxy = new Proxy(target, handler);

proxy.foo

// TypeError: Invariant check failed

###### set()

set方法用来拦截某个属性的赋值操作，可以接受四个参数，依次为目标对象、属性名、属性值和 Proxy 实例本身，其中最后一个参数可选。

假定Person对象有一个age属性，该属性应该是一个不大于 200 的整数，那么可以使用Proxy保证age的属性值符合要求。

let validator = {

set: function(obj, prop, value) {

if (prop === 'age') {

if (!Number.isInteger(value)) {

throw new TypeError('The age is not an integer');

}

if (value > 200) {

throw new RangeError('The age seems invalid');

}

}

// 对于满足条件的 age 属性以及其他属性，直接保存

obj[prop] = value;

}

};

let person = new Proxy({}, validator);

person.age = 100;

person.age // 100

person.age = 'young' // 报错

person.age = 300 // 报错

上面代码中，由于设置了存值函数set，任何不符合要求的age属性赋值，都会抛出一个错误，这是数据验证的一种实现方法。利用set方法，还可以数据绑定，即每当对象发生变化时，会自动更新 DOM。

有时，我们会在对象上面设置内部属性，属性名的第一个字符使用下划线开头，表示这些属性不应该被外部使用。结合get和set方法，就可以做到防止这些内部属性被外部读写。

const handler = {

get (target, key) {

invariant(key, 'get');

return target[key];

},

set (target, key, value) {

invariant(key, 'set');

target[key] = value;

return true;

}

};

function invariant (key, action) {

if (key[0] === '\_') {

throw new Error(`Invalid attempt to ${action} private "${key}" property`);

}

}

const target = {};

const proxy = new Proxy(target, handler);

proxy.\_prop

// Error: Invalid attempt to get private "\_prop" property

proxy.\_prop = 'c'

// Error: Invalid attempt to set private "\_prop" property

上面代码中，只要读写的属性名的第一个字符是下划线，一律抛错，从而达到禁止读写内部属性的目的。

下面是set方法第四个参数的例子。

const handler = {

set: function(obj, prop, value, receiver) {

obj[prop] = receiver;

}

};

const proxy = new Proxy({}, handler);

proxy.foo = 'bar';

proxy.foo === proxy // true

上面代码中，set方法的第四个参数receiver，指的是操作行为所在的那个对象，一般情况下是proxy实例本身，请看下面的例子。

const handler = {

set: function(obj, prop, value, receiver) {

obj[prop] = receiver;

}

};

const proxy = new Proxy({}, handler);

const myObj = {};

Object.setPrototypeOf(myObj, proxy);

myObj.foo = 'bar';

myObj.foo === myObj // true

上面代码中，设置myObj.foo属性的值时，myObj并没有foo属性，因此引擎会到myObj的原型链去找foo属性。myObj的原型对象proxy是一个 Proxy 实例，设置它的foo属性会触发set方法。这时，第四个参数receiver就指向原始赋值行为所在的对象myObj。

注意，如果目标对象自身的某个属性，不可写或不可配置，那么set方法将不起作用。

const obj = {};

Object.defineProperty(obj, 'foo', {

value: 'bar',

writable: false,

});

const handler = {

set: function(obj, prop, value, receiver) {

obj[prop] = 'baz';

}

};

const proxy = new Proxy(obj, handler);

proxy.foo = 'baz';

proxy.foo // "bar"

上面代码中，obj.foo属性不可写，Proxy 对这个属性的set代理将不会生效。

###### apply()

apply方法拦截函数的调用、call和apply操作。

apply方法可以接受三个参数，分别是目标对象、目标对象的上下文对象（this）和目标对象的参数数组。

var handler = {

apply (target, ctx, args) {

return Reflect.apply(...arguments);

}

};

下面是一个例子。

var target = function () { return 'I am the target'; };

var handler = {

apply: function () {

return 'I am the proxy';

}

};

var p = new Proxy(target, handler);

p()

// "I am the proxy"

上面代码中，变量p是 Proxy 的实例，当它作为函数调用时（p()），就会被apply方法拦截，返回一个字符串。

下面是另外一个例子。

var twice = {

apply (target, ctx, args) {

return Reflect.apply(...arguments) \* 2;

}

};

function sum (left, right) {

return left + right;

};

var proxy = new Proxy(sum, twice);

proxy(1, 2) // 6

proxy.call(null, 5, 6) // 22

proxy.apply(null, [7, 8]) // 30

上面代码中，每当执行proxy函数（直接调用或call和apply调用），就会被apply方法拦截。

另外，直接调用Reflect.apply方法，也会被拦截。

Reflect.apply(proxy, null, [9, 10]) // 38

###### has()

has方法用来拦截HasProperty操作，即判断对象是否具有某个属性时，这个方法会生效。典型的操作就是in运算符。

下面的例子使用has方法隐藏某些属性，不被in运算符发现。

var handler = {

has (target, key) {

if (key[0] === '\_') {

return false;

}

return key in target;

}

};

var target = { \_prop: 'foo', prop: 'foo' };

var proxy = new Proxy(target, handler);

'\_prop' in proxy // false

上面代码中，如果原对象的属性名的第一个字符是下划线，proxy.has就会返回false，从而不会被in运算符发现。

如果原对象不可配置或者禁止扩展，这时has拦截会报错。

var obj = { a: 10 };

Object.preventExtensions(obj);

var p = new Proxy(obj, {

has: function(target, prop) {

return false;

}

});

'a' in p // TypeError is thrown

上面代码中，obj对象禁止扩展，结果使用has拦截就会报错。也就是说，如果某个属性不可配置（或者目标对象不可扩展），则has方法就不得“隐藏”（即返回false）目标对象的该属性。

值得注意的是，has方法拦截的是HasProperty操作，而不是HasOwnProperty操作，即has方法不判断一个属性是对象自身的属性，还是继承的属性。

另外，虽然for...in循环也用到了in运算符，但是has拦截对for...in循环不生效。

let stu1 = {name: '张三', score: 59};

let stu2 = {name: '李四', score: 99};

let handler = {

has(target, prop) {

if (prop === 'score' && target[prop] < 60) {

console.log(`${target.name} 不及格`);

return false;

}

return prop in target;

}

}

let oproxy1 = new Proxy(stu1, handler);

let oproxy2 = new Proxy(stu2, handler);

'score' in oproxy1

// 张三 不及格

// false

'score' in oproxy2

// true

for (let a in oproxy1) {

console.log(oproxy1[a]);

}

// 张三

// 59

for (let b in oproxy2) {

console.log(oproxy2[b]);

}

// 李四

// 99

上面代码中，has拦截只对in运算符生效，对for...in循环不生效，导致不符合要求的属性没有被排除在for...in循环之外。

###### construct()

construct方法用于拦截new命令，下面是拦截对象的写法。

var handler = {

construct (target, args, newTarget) {

return new target(...args);

}

};

construct方法可以接受两个参数。

* target: 目标对象
* args：构建函数的参数对象

下面是一个例子。

var p = new Proxy(function () {}, {

construct: function(target, args) {

console.log('called: ' + args.join(', '));

return { value: args[0] \* 10 };

}

});

(new p(1)).value

// "called: 1"

// 10

construct方法返回的必须是一个对象，否则会报错。

var p = new Proxy(function() {}, {

construct: function(target, argumentsList) {

return 1;

}

});

new p() // 报错

###### deleteProperty()

deleteProperty方法用于拦截delete操作，如果这个方法抛出错误或者返回false，当前属性就无法被delete命令删除。

var handler = {

deleteProperty (target, key) {

invariant(key, 'delete');

return true;

}

};

function invariant (key, action) {

if (key[0] === '\_') {

throw new Error(`Invalid attempt to ${action} private "${key}" property`);

}

}

var target = { \_prop: 'foo' };

var proxy = new Proxy(target, handler);

delete proxy.\_prop

// Error: Invalid attempt to delete private "\_prop" property

上面代码中，deleteProperty方法拦截了delete操作符，删除第一个字符为下划线的属性会报错。

注意，目标对象自身的不可配置（configurable）的属性，不能被deleteProperty方法删除，否则报错。

###### defineProperty()

defineProperty方法拦截了Object.defineProperty操作。

var handler = {

defineProperty (target, key, descriptor) {

return false;

}

};

var target = {};

var proxy = new Proxy(target, handler);

proxy.foo = 'bar'

// TypeError: proxy defineProperty handler returned false for property '"foo"'

上面代码中，defineProperty方法返回false，导致添加新属性会抛出错误。

注意，如果目标对象不可扩展（extensible），则defineProperty不能增加目标对象上不存在的属性，否则会报错。另外，如果目标对象的某个属性不可写（writable）或不可配置（configurable），则defineProperty方法不得改变这两个设置。

###### getOwnPropertyDescriptor()

getOwnPropertyDescriptor方法拦截Object.getOwnPropertyDescriptor()，返回一个属性描述对象或者undefined。

var handler = {

getOwnPropertyDescriptor (target, key) {

if (key[0] === '\_') {

return;

}

return Object.getOwnPropertyDescriptor(target, key);

}

};

var target = { \_foo: 'bar', baz: 'tar' };

var proxy = new Proxy(target, handler);

Object.getOwnPropertyDescriptor(proxy, 'wat')

// undefined

Object.getOwnPropertyDescriptor(proxy, '\_foo')

// undefined

Object.getOwnPropertyDescriptor(proxy, 'baz')

// { value: 'tar', writable: true, enumerable: true, configurable: true }

上面代码中，handler.getOwnPropertyDescriptor方法对于第一个字符为下划线的属性名会返回undefined。

###### getPrototypeOf()

getPrototypeOf方法主要用来拦截获取对象原型。具体来说，拦截下面这些操作。

* Object.prototype.\_\_proto\_\_
* Object.prototype.isPrototypeOf()
* Object.getPrototypeOf()
* Reflect.getPrototypeOf()
* instanceof

下面是一个例子。

var proto = {};

var p = new Proxy({}, {

getPrototypeOf(target) {

return proto;

}

});

Object.getPrototypeOf(p) === proto // true

上面代码中，getPrototypeOf方法拦截Object.getPrototypeOf()，返回proto对象。

注意，getPrototypeOf方法的返回值必须是对象或者null，否则报错。另外，如果目标对象不可扩展（extensible）， getPrototypeOf方法必须返回目标对象的原型对象。

###### isExtensible()

isExtensible方法拦截Object.isExtensible操作。

var p = new Proxy({}, {

isExtensible: function(target) {

console.log("called");

return true;

}

});

Object.isExtensible(p)

// "called"

// true

上面代码设置了isExtensible方法，在调用Object.isExtensible时会输出called。

注意，该方法只能返回布尔值，否则返回值会被自动转为布尔值。

这个方法有一个强限制，它的返回值必须与目标对象的isExtensible属性保持一致，否则就会抛出错误。

Object.isExtensible(proxy) === Object.isExtensible(target)

下面是一个例子。

var p = new Proxy({}, {

isExtensible: function(target) {

return false;

}

});

Object.isExtensible(p) // 报错

###### ownKeys()

ownKeys方法用来拦截对象自身属性的读取操作。具体来说，拦截以下操作。

* Object.getOwnPropertyNames()
* Object.getOwnPropertySymbols()
* Object.keys()

下面是拦截Object.keys()的例子。

let target = {

a: 1,

b: 2,

c: 3

};

let handler = {

ownKeys(target) {

return ['a'];

}

};

let proxy = new Proxy(target, handler);

Object.keys(proxy)

// [ 'a' ]

上面代码拦截了对于target对象的Object.keys()操作，只返回a、b、c三个属性之中的a属性。

下面的例子是拦截第一个字符为下划线的属性名。

let target = {

\_bar: 'foo',

\_prop: 'bar',

prop: 'baz'

};

let handler = {

ownKeys (target) {

return Reflect.ownKeys(target).filter(key => key[0] !== '\_');

}

};

let proxy = new Proxy(target, handler);

for (let key of Object.keys(proxy)) {

console.log(target[key]);

}

// "baz"

注意，使用Object.keys方法时，有三类属性会被ownKeys方法自动过滤，不会返回。

* 目标对象上不存在的属性
* 属性名为 Symbol 值
* 不可遍历（enumerable）的属性

let target = {

a: 1,

b: 2,

c: 3,

[Symbol.for('secret')]: '4',

};

Object.defineProperty(target, 'key', {

enumerable: false,

configurable: true,

writable: true,

value: 'static'

});

let handler = {

ownKeys(target) {

return ['a', 'd', Symbol.for('secret'), 'key'];

}

};

let proxy = new Proxy(target, handler);

Object.keys(proxy)

// ['a']

上面代码中，ownKeys方法之中，显式返回不存在的属性（d）、Symbol 值（Symbol.for('secret')）、不可遍历的属性（key），结果都被自动过滤掉。

ownKeys方法还可以拦截Object.getOwnPropertyNames()。

var p = new Proxy({}, {

ownKeys: function(target) {

return ['a', 'b', 'c'];

}

});

Object.getOwnPropertyNames(p)

// [ 'a', 'b', 'c' ]

ownKeys方法返回的数组成员，只能是字符串或 Symbol 值。如果有其他类型的值，或者返回的根本不是数组，就会报错。

var obj = {};

var p = new Proxy(obj, {

ownKeys: function(target) {

return [123, true, undefined, null, {}, []];

}

});

Object.getOwnPropertyNames(p)

// Uncaught TypeError: 123 is not a valid property name

上面代码中，ownKeys方法虽然返回一个数组，但是每一个数组成员都不是字符串或 Symbol 值，因此就报错了。

如果目标对象自身包含不可配置的属性，则该属性必须被ownKeys方法返回，否则报错。

var obj = {};

Object.defineProperty(obj, 'a', {

configurable: false,

enumerable: true,

value: 10 }

);

var p = new Proxy(obj, {

ownKeys: function(target) {

return ['b'];

}

});

Object.getOwnPropertyNames(p)

// Uncaught TypeError: 'ownKeys' on proxy: trap result did not include 'a'

上面代码中，obj对象的a属性是不可配置的，这时ownKeys方法返回的数组之中，必须包含a，否则会报错。

另外，如果目标对象是不可扩展的（non-extensition），这时ownKeys方法返回的数组之中，必须包含原对象的所有属性，且不能包含多余的属性，否则报错。

var obj = {

a: 1

};

Object.preventExtensions(obj);

var p = new Proxy(obj, {

ownKeys: function(target) {

return ['a', 'b'];

}

});

Object.getOwnPropertyNames(p)

// Uncaught TypeError: 'ownKeys' on proxy: trap returned extra keys but proxy target is non-extensible

上面代码中，obj对象是不可扩展的，这时ownKeys方法返回的数组之中，包含了obj对象的多余属性b，所以导致了报错。

###### preventExtensions()

preventExtensions方法拦截Object.preventExtensions()。该方法必须返回一个布尔值，否则会被自动转为布尔值。

这个方法有一个限制，只有目标对象不可扩展时（即Object.isExtensible(proxy)为false），proxy.preventExtensions才能返回true，否则会报错。

var p = new Proxy({}, {

preventExtensions: function(target) {

return true;

}

});

Object.preventExtensions(p) // 报错

上面代码中，proxy.preventExtensions方法返回true，但这时Object.isExtensible(proxy)会返回true，因此报错。

为了防止出现这个问题，通常要在proxy.preventExtensions方法里面，调用一次Object.preventExtensions。

var p = new Proxy({}, {

preventExtensions: function(target) {

console.log('called');

Object.preventExtensions(target);

return true;

}

});

Object.preventExtensions(p)

// "called"

// true

###### setPrototypeOf()

setPrototypeOf方法主要用来拦截Object.setPrototypeOf方法。

下面是一个例子。

var handler = {

setPrototypeOf (target, proto) {

throw new Error('Changing the prototype is forbidden');

}

};

var proto = {};

var target = function () {};

var proxy = new Proxy(target, handler);

Object.setPrototypeOf(proxy, proto);

// Error: Changing the prototype is forbidden

上面代码中，只要修改target的原型对象，就会报错。

注意，该方法只能返回布尔值，否则会被自动转为布尔值。另外，如果目标对象不可扩展（extensible），setPrototypeOf方法不得改变目标对象的原型。

##### Proxy.revocable()

Proxy.revocable方法返回一个可取消的 Proxy 实例。

let target = {};

let handler = {};

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

proxy.foo = 123;

proxy.foo // 123

revoke();

proxy.foo // TypeError: Revoked

Proxy.revocable方法返回一个对象，该对象的proxy属性是Proxy实例，revoke属性是一个函数，可以取消Proxy实例。上面代码中，当执行revoke函数之后，再访问Proxy实例，就会抛出一个错误。

Proxy.revocable的一个使用场景是，目标对象不允许直接访问，必须通过代理访问，一旦访问结束，就收回代理权，不允许再次访问。

##### this 问题

虽然 Proxy 可以代理针对目标对象的访问，但它不是目标对象的透明代理，即不做任何拦截的情况下，也无法保证与目标对象的行为一致。主要原因就是在 Proxy 代理的情况下，目标对象内部的this关键字会指向 Proxy 代理。

const target = {

m: function () {

console.log(this === proxy);

}

};

const handler = {};

const proxy = new Proxy(target, handler);

target.m() // false

proxy.m() // true

上面代码中，一旦proxy代理target.m，后者内部的this就是指向proxy，而不是target。

下面是一个例子，由于this指向的变化，导致 Proxy 无法代理目标对象。

const \_name = new WeakMap();

class Person {

constructor(name) {

\_name.set(this, name);

}

get name() {

return \_name.get(this);

}

}

const jane = new Person('Jane');

jane.name // 'Jane'

const proxy = new Proxy(jane, {});

proxy.name // undefined

上面代码中，目标对象jane的name属性，实际保存在外部WeakMap对象\_name上面，通过this键区分。由于通过proxy.name访问时，this指向proxy，导致无法取到值，所以返回undefined。

此外，有些原生对象的内部属性，只有通过正确的this才能拿到，所以 Proxy 也无法代理这些原生对象的属性。

const target = new Date();

const handler = {};

const proxy = new Proxy(target, handler);

proxy.getDate();

// TypeError: this is not a Date object.

上面代码中，getDate方法只能在Date对象实例上面拿到，如果this不是Date对象实例就会报错。这时，this绑定原始对象，就可以解决这个问题。

const target = new Date('2015-01-01');

const handler = {

get(target, prop) {

if (prop === 'getDate') {

return target.getDate.bind(target);

}

return Reflect.get(target, prop);

}

};

const proxy = new Proxy(target, handler);

proxy.getDate() // 1

##### 实例：Web 服务的客户端

Proxy 对象可以拦截目标对象的任意属性，这使得它很合适用来写 Web 服务的客户端。

const service = createWebService('http://example.com/data');

service.employees().then(json => {

const employees = JSON.parse(json);

// ···

});

上面代码新建了一个 Web 服务的接口，这个接口返回各种数据。Proxy 可以拦截这个对象的任意属性，所以不用为每一种数据写一个适配方法，只要写一个 Proxy 拦截就可以了。

function createWebService(baseUrl) {

return new Proxy({}, {

get(target, propKey, receiver) {

return () => httpGet(baseUrl+'/' + propKey);

}

});

}

同理，Proxy 也可以用来实现数据库的 ORM 层。

#### Reflect

##### 概述

Reflect对象与Proxy对象一样，也是 ES6 为了操作对象而提供的新 API。Reflect对象的设计目的有这样几个。

（1） 将Object对象的一些明显属于语言内部的方法（比如Object.defineProperty），放到Reflect对象上。现阶段，某些方法同时在Object和Reflect对象上部署，未来的新方法将只部署在Reflect对象上。也就是说，从Reflect对象上可以拿到语言内部的方法。

（2） 修改某些Object方法的返回结果，让其变得更合理。比如，Object.defineProperty(obj, name, desc)在无法定义属性时，会抛出一个错误，而Reflect.defineProperty(obj, name, desc)则会返回false。

// 老写法

try {

Object.defineProperty(target, property, attributes);

// success

} catch (e) {

// failure

}

// 新写法

if (Reflect.defineProperty(target, property, attributes)) {

// success

} else {

// failure

}

（3） 让Object操作都变成函数行为。某些Object操作是命令式，比如name in obj和delete obj[name]，而Reflect.has(obj, name)和Reflect.deleteProperty(obj, name)让它们变成了函数行为。

// 老写法

'assign' in Object // true

// 新写法

Reflect.has(Object, 'assign') // true

（4）Reflect对象的方法与Proxy对象的方法一一对应，只要是Proxy对象的方法，就能在Reflect对象上找到对应的方法。这就让Proxy对象可以方便地调用对应的Reflect方法，完成默认行为，作为修改行为的基础。也就是说，不管Proxy怎么修改默认行为，你总可以在Reflect上获取默认行为。

Proxy(target, {

set: function(target, name, value, receiver) {

var success = Reflect.set(target,name, value, receiver);

if (success) {

log('property ' + name + ' on ' + target + ' set to ' + value);

}

return success;

}

});

上面代码中，Proxy方法拦截target对象的属性赋值行为。它采用Reflect.set方法将值赋值给对象的属性，确保完成原有的行为，然后再部署额外的功能。

下面是另一个例子。

var loggedObj = new Proxy(obj, {

get(target, name) {

console.log('get', target, name);

return Reflect.get(target, name);

},

deleteProperty(target, name) {

console.log('delete' + name);

return Reflect.deleteProperty(target, name);

},

has(target, name) {

console.log('has' + name);

return Reflect.has(target, name);

}

});

上面代码中，每一个Proxy对象的拦截操作（get、delete、has），内部都调用对应的Reflect方法，保证原生行为能够正常执行。添加的工作，就是将每一个操作输出一行日志。

有了Reflect对象以后，很多操作会更易读。

// 老写法

Function.prototype.apply.call(Math.floor, undefined, [1.75]) // 1

// 新写法

Reflect.apply(Math.floor, undefined, [1.75]) // 1

##### 静态方法

Reflect对象一共有 13 个静态方法。

* 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)

上面这些方法的作用，大部分与Object对象的同名方法的作用都是相同的，而且它与Proxy对象的方法是一一对应的。下面是对它们的解释。

###### Reflect.get(target, name, receiver)

Reflect.get方法查找并返回target对象的name属性，如果没有该属性，则返回undefined。

var myObject = {

foo: 1,

bar: 2,

get baz() {

return this.foo + this.bar;

},

}

Reflect.get(myObject, 'foo') // 1

Reflect.get(myObject, 'bar') // 2

Reflect.get(myObject, 'baz') // 3

如果name属性部署了读取函数（getter），则读取函数的this绑定receiver。

var myObject = {

foo: 1,

bar: 2,

get baz() {

return this.foo + this.bar;

},

};

var myReceiverObject = {

foo: 4,

bar: 4,

};

Reflect.get(myObject, 'baz', myReceiverObject) // 8

如果第一个参数不是对象，Reflect.get方法会报错。

Reflect.get(1, 'foo') // 报错

Reflect.get(false, 'foo') // 报错

###### Reflect.set(target, name, value, receiver)

Reflect.set方法设置target对象的name属性等于value。

var myObject = {

foo: 1,

set bar(value) {

return this.foo = value;

},

}

myObject.foo // 1

Reflect.set(myObject, 'foo', 2);

myObject.foo // 2

Reflect.set(myObject, 'bar', 3)

myObject.foo // 3

如果name属性设置了赋值函数，则赋值函数的this绑定receiver。

var myObject = {

foo: 4,

set bar(value) {

return this.foo = value;

},

};

var myReceiverObject = {

foo: 0,

};

Reflect.set(myObject, 'bar', 1, myReceiverObject);

myObject.foo // 4

myReceiverObject.foo // 1

注意，如果 Proxy 对象和 Reflect 对象联合使用，前者拦截赋值操作，后者完成赋值的默认行为，而且传入了receiver，那么Reflect.set会触发Proxy.defineProperty拦截。

let p = {

a: 'a'

};

let handler = {

set(target, key, value, receiver) {

console.log('set');

Reflect.set(target, key, value, receiver)

},

defineProperty(target, key, attribute) {

console.log('defineProperty');

Reflect.defineProperty(target, key, attribute);

}

};

let obj = new Proxy(p, handler);

obj.a = 'A';

// set

// defineProperty

上面代码中，Proxy.set拦截里面使用了Reflect.set，而且传入了receiver，导致触发Proxy.defineProperty拦截。这是因为Proxy.set的receiver参数总是指向当前的 Proxy 实例（即上例的obj），而Reflect.set一旦传入receiver，就会将属性赋值到receiver上面（即obj），导致触发defineProperty拦截。如果Reflect.set没有传入receiver，那么就不会触发defineProperty拦截。

let p = {

a: 'a'

};

let handler = {

set(target, key, value, receiver) {

console.log('set');

Reflect.set(target, key, value)

},

defineProperty(target, key, attribute) {

console.log('defineProperty');

Reflect.defineProperty(target, key, attribute);

}

};

let obj = new Proxy(p, handler);

obj.a = 'A';

// set

如果第一个参数不是对象，Reflect.set会报错。

Reflect.set(1, 'foo', {}) // 报错

Reflect.set(false, 'foo', {}) // 报错

###### Reflect.has(obj, name)

Reflect.has方法对应name in obj里面的in运算符。

var myObject = {

foo: 1,

};

// 旧写法

'foo' in myObject // true

// 新写法

Reflect.has(myObject, 'foo') // true

如果第一个参数不是对象，Reflect.has和in运算符都会报错。

###### Reflect.deleteProperty(obj, name)

Reflect.deleteProperty方法等同于delete obj[name]，用于删除对象的属性。

const myObj = { foo: 'bar' };

// 旧写法

delete myObj.foo;

// 新写法

Reflect.deleteProperty(myObj, 'foo');

该方法返回一个布尔值。如果删除成功，或者被删除的属性不存在，返回true；删除失败，被删除的属性依然存在，返回false。

###### Reflect.construct(target, args)

Reflect.construct方法等同于new target(...args)，这提供了一种不使用new，来调用构造函数的方法。

function Greeting(name) {

this.name = name;

}

// new 的写法

const instance = new Greeting('张三');

// Reflect.construct 的写法

const instance = Reflect.construct(Greeting, ['张三']);

###### Reflect.getPrototypeOf(obj)

Reflect.getPrototypeOf方法用于读取对象的\_\_proto\_\_属性，对应Object.getPrototypeOf(obj)。

const myObj = new FancyThing();

// 旧写法

Object.getPrototypeOf(myObj) === FancyThing.prototype;

// 新写法

Reflect.getPrototypeOf(myObj) === FancyThing.prototype;

Reflect.getPrototypeOf和Object.getPrototypeOf的一个区别是，如果参数不是对象，Object.getPrototypeOf会将这个参数转为对象，然后再运行，而Reflect.getPrototypeOf会报错。

Object.getPrototypeOf(1) // Number {[[PrimitiveValue]]: 0}

Reflect.getPrototypeOf(1) // 报错

###### Reflect.setPrototypeOf(obj, newProto)

Reflect.setPrototypeOf方法用于设置对象的\_\_proto\_\_属性，返回第一个参数对象，对应Object.setPrototypeOf(obj, newProto)。

const myObj = new FancyThing();

// 旧写法

Object.setPrototypeOf(myObj, OtherThing.prototype);

// 新写法

Reflect.setPrototypeOf(myObj, OtherThing.prototype);

如果第一个参数不是对象，Object.setPrototypeOf会返回第一个参数本身，而Reflect.setPrototypeOf会报错。

Object.setPrototypeOf(1, {})

// 1

Reflect.setPrototypeOf(1, {})

// TypeError: Reflect.setPrototypeOf called on non-object

如果第一个参数是undefined或null，Object.setPrototypeOf和Reflect.setPrototypeOf都会报错。

Object.setPrototypeOf(null, {})

// TypeError: Object.setPrototypeOf called on null or undefined

Reflect.setPrototypeOf(null, {})

// TypeError: Reflect.setPrototypeOf called on non-object

###### Reflect.apply(func, thisArg, args)

Reflect.apply方法等同于Function.prototype.apply.call(func, thisArg, args)，用于绑定this对象后执行给定函数。

一般来说，如果要绑定一个函数的this对象，可以这样写fn.apply(obj, args)，但是如果函数定义了自己的apply方法，就只能写成Function.prototype.apply.call(fn, obj, args)，采用Reflect对象可以简化这种操作。

const ages = [11, 33, 12, 54, 18, 96];

// 旧写法

const youngest = Math.min.apply(Math, ages);

const oldest = Math.max.apply(Math, ages);

const type = Object.prototype.toString.call(youngest);

// 新写法

const youngest = Reflect.apply(Math.min, Math, ages);

const oldest = Reflect.apply(Math.max, Math, ages);

const type = Reflect.apply(Object.prototype.toString, youngest, []);

###### Reflect.defineProperty(target, propertyKey, attributes)

Reflect.defineProperty方法基本等同于Object.defineProperty，用来为对象定义属性。未来，后者会被逐渐废除，请从现在开始就使用Reflect.defineProperty代替它。

function MyDate() {

/\*…\*/

}

// 旧写法

Object.defineProperty(MyDate, 'now', {

value: () => Date.now()

});

// 新写法

Reflect.defineProperty(MyDate, 'now', {

value: () => Date.now()

});

如果Reflect.defineProperty的第一个参数不是对象，就会抛出错误，比如Reflect.defineProperty(1, 'foo')。

这个方法可以与Proxy.defineProperty配合使用。

const p = new Proxy({}, {

defineProperty(target, prop, descriptor) {

console.log(descriptor);

return Reflect.defineProperty(target, prop, descriptor);

}

});

p.foo = 'bar';

// {value: "bar", writable: true, enumerable: true, configurable: true}

p.foo // "bar"

上面代码中，Proxy.defineProperty对属性赋值设置了拦截，然后使用Reflect.defineProperty完成了赋值。

###### Reflect.getOwnPropertyDescriptor(target, propertyKey)

Reflect.getOwnPropertyDescriptor基本等同于Object.getOwnPropertyDescriptor，用于得到指定属性的描述对象，将来会替代掉后者。

var myObject = {};

Object.defineProperty(myObject, 'hidden', {

value: true,

enumerable: false,

});

// 旧写法

var theDescriptor = Object.getOwnPropertyDescriptor(myObject, 'hidden');

// 新写法

var theDescriptor = Reflect.getOwnPropertyDescriptor(myObject, 'hidden');

Reflect.getOwnPropertyDescriptor和Object.getOwnPropertyDescriptor的一个区别是，如果第一个参数不是对象，Object.getOwnPropertyDescriptor(1, 'foo')不报错，返回undefined，而Reflect.getOwnPropertyDescriptor(1, 'foo')会抛出错误，表示参数非法。

###### Reflect.isExtensible (target)

Reflect.isExtensible方法对应Object.isExtensible，返回一个布尔值，表示当前对象是否可扩展。

const myObject = {};

// 旧写法

Object.isExtensible(myObject) // true

// 新写法

Reflect.isExtensible(myObject) // true

如果参数不是对象，Object.isExtensible会返回false，因为非对象本来就是不可扩展的，而Reflect.isExtensible会报错。

Object.isExtensible(1) // false

Reflect.isExtensible(1) // 报错

###### Reflect.preventExtensions(target)

Reflect.preventExtensions对应Object.preventExtensions方法，用于让一个对象变为不可扩展。它返回一个布尔值，表示是否操作成功。

var myObject = {};

// 旧写法

Object.preventExtensions(myObject) // Object {}

// 新写法

Reflect.preventExtensions(myObject) // true

如果参数不是对象，Object.preventExtensions在 ES5 环境报错，在 ES6 环境返回传入的参数，而Reflect.preventExtensions会报错。

// ES5 环境

Object.preventExtensions(1) // 报错

// ES6 环境

Object.preventExtensions(1) // 1

// 新写法

Reflect.preventExtensions(1) // 报错

###### Reflect.ownKeys (target)

Reflect.ownKeys方法用于返回对象的所有属性，基本等同于Object.getOwnPropertyNames与Object.getOwnPropertySymbols之和。

var myObject = {

foo: 1,

bar: 2,

[Symbol.for('baz')]: 3,

[Symbol.for('bing')]: 4,

};

// 旧写法

Object.getOwnPropertyNames(myObject)

// ['foo', 'bar']

Object.getOwnPropertySymbols(myObject)

//[Symbol(baz), Symbol(bing)]

// 新写法

Reflect.ownKeys(myObject)

// ['foo', 'bar', Symbol(baz), Symbol(bing)]

##### 实例：使用 Proxy 实现观察者模式

观察者模式（Observer mode）指的是函数自动观察数据对象，一旦对象有变化，函数就会自动执行。

const person = observable({

name: '张三',

age: 20

});

function print() {

console.log(`${person.name}, ${person.age}`)

}

observe(print);

person.name = '李四';

// 输出

// 李四, 20

上面代码中，数据对象person是观察目标，函数print是观察者。一旦数据对象发生变化，print就会自动执行。

下面，使用 Proxy 写一个观察者模式的最简单实现，即实现observable和observe这两个函数。思路是observable函数返回一个原始对象的 Proxy 代理，拦截赋值操作，触发充当观察者的各个函数。

const queuedObservers = new Set();

const observe = fn => queuedObservers.add(fn);

const observable = obj => new Proxy(obj, {set});

function set(target, key, value, receiver) {

const result = Reflect.set(target, key, value, receiver);

queuedObservers.forEach(observer => observer());

return result;

}

上面代码中，先定义了一个Set集合，所有观察者函数都放进这个集合。然后，observable函数返回原始对象的代理，拦截赋值操作。拦截函数set之中，会自动执行所有观察者。

#### async 函数

##### 含义

ES2017 标准引入了 async 函数，使得异步操作变得更加方便。

async 函数是什么？一句话，它就是 Generator 函数的语法糖。

前有一个 Generator 函数，依次读取两个文件。

const fs = require('fs');

const readFile = function (fileName) {

return new Promise(function (resolve, reject) {

fs.readFile(fileName, function(error, data) {

if (error) return reject(error);

resolve(data);

});

});

};

const gen = function\* () {

const f1 = yield readFile('/etc/fstab');

const f2 = yield readFile('/etc/shells');

console.log(f1.toString());

console.log(f2.toString());

};

写成async函数，就是下面这样。

const asyncReadFile = async function () {

const f1 = await readFile('/etc/fstab');

const f2 = await readFile('/etc/shells');

console.log(f1.toString());

console.log(f2.toString());

};

一比较就会发现，async函数就是将 Generator 函数的星号（\*）替换成async，将yield替换成await，仅此而已。

async函数对 Generator 函数的改进，体现在以下四点。

（1）内置执行器。

Generator 函数的执行必须靠执行器，所以才有了co模块，而async函数自带执行器。也就是说，async函数的执行，与普通函数一模一样，只要一行。

asyncReadFile();

上面的代码调用了asyncReadFile函数，然后它就会自动执行，输出最后结果。这完全不像 Generator 函数，需要调用next方法，或者用co模块，才能真正执行，得到最后结果。

（2）更好的语义。

async和await，比起星号和yield，语义更清楚了。async表示函数里有异步操作，await表示紧跟在后面的表达式需要等待结果。

（3）更广的适用性。

co模块约定，yield命令后面只能是 Thunk 函数或 Promise 对象，而async函数的await命令后面，可以是 Promise 对象和原始类型的值（数值、字符串和布尔值，但这时等同于同步操作）。

（4）返回值是 Promise。

async函数的返回值是 Promise 对象，这比 Generator 函数的返回值是 Iterator 对象方便多了。你可以用then方法指定下一步的操作。

进一步说，async函数完全可以看作多个异步操作，包装成的一个 Promise 对象，而await命令就是内部then命令的语法糖。

##### 基本用法

async函数返回一个 Promise 对象，可以使用then方法添加回调函数。当函数执行的时候，一旦遇到await就会先返回，等到异步操作完成，再接着执行函数体内后面的语句。

下面是一个例子。

async function getStockPriceByName(name) {

const symbol = await getStockSymbol(name);

const stockPrice = await getStockPrice(symbol);

return stockPrice;

}

getStockPriceByName('goog').then(function (result) {

console.log(result);

});

上面代码是一个获取股票报价的函数，函数前面的async关键字，表明该函数内部有异步操作。调用该函数时，会立即返回一个Promise对象。

下面是另一个例子，指定多少毫秒后输出一个值。

function timeout(ms) {

return new Promise((resolve) => {

setTimeout(resolve, ms);

});

}

async function asyncPrint(value, ms) {

await timeout(ms);

console.log(value);

}

asyncPrint('hello world', 50);

上面代码指定 50 毫秒以后，输出hello world。

async 函数有多种使用形式。

// 函数声明

async function foo() {}

// 函数表达式

const foo = async function () {};

// 对象的方法

let obj = { async foo() {} };

obj.foo().then(...)

// Class 的方法

class Storage {

constructor() {

this.cachePromise = caches.open('avatars');

}

async getAvatar(name) {

const cache = await this.cachePromise;

return cache.match(`/avatars/${name}.jpg`);

}

}

const storage = new Storage();

storage.getAvatar('jake').then(…);

// 箭头函数

const foo = async () => {};

##### 语法

async函数的语法规则总体上比较简单，难点是错误处理机制。

###### 返回 Promise 对象

async函数返回一个 Promise 对象。

async函数内部return语句返回的值，会成为then方法回调函数的参数。

async function f() {

return 'hello world';

}

f().then(v => console.log(v))

// "hello world"

上面代码中，函数f内部return命令返回的值，会被then方法回调函数接收到。

async函数内部抛出错误，会导致返回的 Promise 对象变为reject状态。抛出的错误对象会被catch方法回调函数接收到。

async function f() {

throw new Error('出错了');

}

f().then(

v => console.log(v),

e => console.log(e)

)

// Error: 出错了

async函数返回的 Promise 对象，必须等到内部所有await命令后面的 Promise 对象执行完，才会发生状态改变，除非遇到return语句或者抛出错误。也就是说，只有async函数内部的异步操作执行完，才会执行then方法指定的回调函数。

下面是一个例子。

async function getTitle(url) {

let response = await fetch(url);

let html = await response.text();

return html.match(/<title>([\s\S]+)<\/title>/i)[1];

}

getTitle('https://tc39.github.io/ecma262/').then(console.log)

// "ECMAScript 2017 Language Specification"

上面代码中，函数getTitle内部有三个操作：抓取网页、取出文本、匹配页面标题。只有这三个操作全部完成，才会执行then方法里面的console.log。

###### Promise 对象的状态变化

async函数返回的 Promise 对象，必须等到内部所有await命令后面的 Promise 对象执行完，才会发生状态改变，除非遇到return语句或者抛出错误。也就是说，只有async函数内部的异步操作执行完，才会执行then方法指定的回调函数。

下面是一个例子。

async function getTitle(url) {

let response = await fetch(url);

let html = await response.text();

return html.match(/<title>([\s\S]+)<\/title>/i)[1];

}

getTitle('https://tc39.github.io/ecma262/').then(console.log)

// "ECMAScript 2017 Language Specification"

上面代码中，函数getTitle内部有三个操作：抓取网页、取出文本、匹配页面标题。只有这三个操作全部完成，才会执行then方法里面的console.log。

###### await 命令

正常情况下，await命令后面是一个 Promise 对象。如果不是，会被转成一个立即resolve的 Promise 对象。

async function f() {

return await 123;

}

f().then(v => console.log(v))

// 123

上面代码中，await命令的参数是数值123，它被转成 Promise 对象，并立即resolve。

await命令后面的 Promise 对象如果变为reject状态，则reject的参数会被catch方法的回调函数接收到。

async function f() {

await Promise.reject('出错了');

}

f()

.then(v => console.log(v))

.catch(e => console.log(e))

// 出错了

注意，上面代码中，await语句前面没有return，但是reject方法的参数依然传入了catch方法的回调函数。这里如果在await前面加上return，效果是一样的。

只要一个await语句后面的 Promise 变为reject，那么整个async函数都会中断执行。

async function f() {

await Promise.reject('出错了');

await Promise.resolve('hello world'); // 不会执行

}

上面代码中，第二个await语句是不会执行的，因为第一个await语句状态变成了reject。

有时，我们希望即使前一个异步操作失败，也不要中断后面的异步操作。这时可以将第一个await放在try...catch结构里面，这样不管这个异步操作是否成功，第二个await都会执行。

async function f() {

try {

await Promise.reject('出错了');

} catch(e) {

}

return await Promise.resolve('hello world');

}

f()

.then(v => console.log(v))

// hello world

另一种方法是await后面的 Promise 对象再跟一个catch方法，处理前面可能出现的错误。

async function f() {

await Promise.reject('出错了')

.catch(e => console.log(e));

return await Promise.resolve('hello world');

}

f()

.then(v => console.log(v))

// 出错了

// hello world

如果await后面的异步操作出错，那么等同于async函数返回的 Promise 对象被reject。

async function f() {

await new Promise(function (resolve, reject) {

throw new Error('出错了');

});

}

f()

.then(v => console.log(v))

.catch(e => console.log(e))

// Error：出错了

上面代码中，async函数f执行后，await后面的 Promise 对象会抛出一个错误对象，导致catch方法的回调函数被调用，它的参数就是抛出的错误对象。具体的执行机制，可以参考后文的“async 函数的实现原理”。

防止出错的方法，也是将其放在try...catch代码块之中。

async function f() {

try {

await new Promise(function (resolve, reject) {

throw new Error('出错了');

});

} catch(e) {

}

return await('hello world');

}

如果有多个await命令，可以统一放在try...catch结构中。

async function main() {

try {

const val1 = await firstStep();

const val2 = await secondStep(val1);

const val3 = await thirdStep(val1, val2);

console.log('Final: ', val3);

}

catch (err) {

console.error(err);

}

}

下面的例子使用try...catch结构，实现多次重复尝试。

const superagent = require('superagent');

const NUM\_RETRIES = 3;

async function test() {

let i;

for (i = 0; i < NUM\_RETRIES; ++i) {

try {

await superagent.get('http://google.com/this-throws-an-error');

break;

} catch(err) {}

}

console.log(i); // 3

}

test();

上面代码中，如果await操作成功，就会使用break语句退出循环；如果失败，会被catch语句捕捉，然后进入下一轮循环。

###### 使用注意点

第一点，前面已经说过，await命令后面的Promise对象，运行结果可能是rejected，所以最好把await命令放在try...catch代码块中。

async function myFunction() {

try {

await somethingThatReturnsAPromise();

} catch (err) {

console.log(err);

}

}

// 另一种写法

async function myFunction() {

await somethingThatReturnsAPromise()

.catch(function (err) {

console.log(err);

});

}

第二点，多个await命令后面的异步操作，如果不存在继发关系，最好让它们同时触发。

let foo = await getFoo();

let bar = await getBar();

上面代码中，getFoo和getBar是两个独立的异步操作（即互不依赖），被写成继发关系。这样比较耗时，因为只有getFoo完成以后，才会执行getBar，完全可以让它们同时触发。

// 写法一

let [foo, bar] = await Promise.all([getFoo(), getBar()]);

// 写法二

let fooPromise = getFoo();

let barPromise = getBar();

let foo = await fooPromise;

let bar = await barPromise;

上面两种写法，getFoo和getBar都是同时触发，这样就会缩短程序的执行时间。

第三点，await命令只能用在async函数之中，如果用在普通函数，就会报错。

async function dbFuc(db) {

let docs = [{}, {}, {}];

// 报错

docs.forEach(function (doc) {

await db.post(doc);

});

}

上面代码会报错，因为await用在普通函数之中了。但是，如果将forEach方法的参数改成async函数，也有问题。

function dbFuc(db) { //这里不需要 async

let docs = [{}, {}, {}];

// 可能得到错误结果

docs.forEach(async function (doc) {

await db.post(doc);

});

}

上面代码可能不会正常工作，原因是这时三个db.post操作将是并发执行，也就是同时执行，而不是继发执行。正确的写法是采用for循环。

async function dbFuc(db) {

let docs = [{}, {}, {}];

for (let doc of docs) {

await db.post(doc);

}

}

如果确实希望多个请求并发执行，可以使用Promise.all方法。当三个请求都会resolved时，下面两种写法效果相同。

async function dbFuc(db) {

let docs = [{}, {}, {}];

let promises = docs.map((doc) => db.post(doc));

let results = await Promise.all(promises);

console.log(results);

}

// 或者使用下面的写法

async function dbFuc(db) {

let docs = [{}, {}, {}];

let promises = docs.map((doc) => db.post(doc));

let results = [];

for (let promise of promises) {

results.push(await promise);

}

console.log(results);

}

目前，[@std/esm](https://www.npmjs.com/package/@std/esm)模块加载器支持顶层await，即await命令可以不放在 async 函数里面，直接使用。

// async 函数的写法

const start = async () => {

const res = await fetch('google.com');

return res.text();

};

start().then(console.log);

// 顶层 await 的写法

const res = await fetch('google.com');

console.log(await res.text());

上面代码中，第二种写法的脚本必须使用@std/esm加载器，才会生效。

##### async 函数的实现原理

async 函数的实现原理，就是将 Generator 函数和自动执行器，包装在一个函数里。

async function fn(args) {

// ...

}

// 等同于

function fn(args) {

return spawn(function\* () {

// ...

});

}

所有的async函数都可以写成上面的第二种形式，其中的spawn函数就是自动执行器。

下面给出spawn函数的实现，基本就是前文自动执行器的翻版。

function spawn(genF) {

return new Promise(function(resolve, reject) {

const gen = genF();

function step(nextF) {

let next;

try {

next = nextF();

} catch(e) {

return reject(e);

}

if(next.done) {

return resolve(next.value);

}

Promise.resolve(next.value).then(function(v) {

step(function() { return gen.next(v); });

}, function(e) {

step(function() { return gen.throw(e); });

});

}

step(function() { return gen.next(undefined); });

});

}

##### 与其他异步处理方法的比较

我们通过一个例子，来看 async 函数与 Promise、Generator 函数的比较。

假定某个 DOM 元素上面，部署了一系列的动画，前一个动画结束，才能开始后一个。如果当中有一个动画出错，就不再往下执行，返回上一个成功执行的动画的返回值。

首先是 Promise 的写法。

function chainAnimationsPromise(elem, animations) {

// 变量ret用来保存上一个动画的返回值

let ret = null;

// 新建一个空的Promise

let p = Promise.resolve();

// 使用then方法，添加所有动画

for(let anim of animations) {

p = p.then(function(val) {

ret = val;

return anim(elem);

});

}

// 返回一个部署了错误捕捉机制的Promise

return p.catch(function(e) {

/\* 忽略错误，继续执行 \*/

}).then(function() {

return ret;

});

}

虽然 Promise 的写法比回调函数的写法大大改进，但是一眼看上去，代码完全都是 Promise 的 API（then、catch等等），操作本身的语义反而不容易看出来。

接着是 Generator 函数的写法。

function chainAnimationsGenerator(elem, animations) {

return spawn(function\*() {

let ret = null;

try {

for(let anim of animations) {

ret = yield anim(elem);

}

} catch(e) {

/\* 忽略错误，继续执行 \*/

}

return ret;

});

}

上面代码使用 Generator 函数遍历了每个动画，语义比 Promise 写法更清晰，用户定义的操作全部都出现在spawn函数的内部。这个写法的问题在于，必须有一个任务运行器，自动执行 Generator 函数，上面代码的spawn函数就是自动执行器，它返回一个 Promise 对象，而且必须保证yield语句后面的表达式，必须返回一个 Promise。

最后是 async 函数的写法。

async function chainAnimationsAsync(elem, animations) {

let ret = null;

try {

for(let anim of animations) {

ret = await anim(elem);

}

} catch(e) {

/\* 忽略错误，继续执行 \*/

}

return ret;

}

可以看到 Async 函数的实现最简洁，最符合语义，几乎没有语义不相关的代码。它将 Generator 写法中的自动执行器，改在语言层面提供，不暴露给用户，因此代码量最少。如果使用 Generator 写法，自动执行器需要用户自己提供。

##### 实例：按顺序完成异步操作

实际开发中，经常遇到一组异步操作，需要按照顺序完成。比如，依次远程读取一组 URL，然后按照读取的顺序输出结果。

Promise 的写法如下。

function logInOrder(urls) {

// 远程读取所有URL

const textPromises = urls.map(url => {

return fetch(url).then(response => response.text());

});

// 按次序输出

textPromises.reduce((chain, textPromise) => {

return chain.then(() => textPromise)

.then(text => console.log(text));

}, Promise.resolve());

}

上面代码使用fetch方法，同时远程读取一组 URL。每个fetch操作都返回一个 Promise 对象，放入textPromises数组。然后，reduce方法依次处理每个 Promise 对象，然后使用then，将所有 Promise 对象连起来，因此就可以依次输出结果。

这种写法不太直观，可读性比较差。下面是 async 函数实现。

async function logInOrder(urls) {

for (const url of urls) {

const response = await fetch(url);

console.log(await response.text());

}

}

上面代码确实大大简化，问题是所有远程操作都是继发。只有前一个 URL 返回结果，才会去读取下一个 URL，这样做效率很差，非常浪费时间。我们需要的是并发发出远程请求。

async function logInOrder(urls) {

// 并发读取远程URL

const textPromises = urls.map(async url => {

const response = await fetch(url);

return response.text();

});

// 按次序输出

for (const textPromise of textPromises) {

console.log(await textPromise);

}

}

上面代码中，虽然map方法的参数是async函数，但它是并发执行的，因为只有async函数内部是继发执行，外部不受影响。后面的for..of循环内部使用了await，因此实现了按顺序输出。

##### 异步遍历器

《遍历器》一章说过，Iterator 接口是一种数据遍历的协议，只要调用遍历器对象的next方法，就会得到一个对象，表示当前遍历指针所在的那个位置的信息。next方法返回的对象的结构是{value, done}，其中value表示当前的数据的值，done是一个布尔值，表示遍历是否结束。

这里隐含着一个规定，next方法必须是同步的，只要调用就必须立刻返回值。也就是说，一旦执行next方法，就必须同步地得到value和done这两个属性。如果遍历指针正好指向同步操作，当然没有问题，但对于异步操作，就不太合适了。目前的解决方法是，Generator 函数里面的异步操作，返回一个 Thunk 函数或者 Promise 对象，即value属性是一个 Thunk 函数或者 Promise 对象，等待以后返回真正的值，而done属性则还是同步产生的。

ES2018 [引入](https://github.com/tc39/proposal-async-iteration)了”异步遍历器“（Async Iterator），为异步操作提供原生的遍历器接口，即value和done这两个属性都是异步产生。

###### 异步遍历的接口

异步遍历器的最大的语法特点，就是调用遍历器的next方法，返回的是一个 Promise 对象。

asyncIterator

.next()

.then(

({ value, done }) => /\* ... \*/

);

上面代码中，asyncIterator是一个异步遍历器，调用next方法以后，返回一个 Promise 对象。因此，可以使用then方法指定，这个 Promise 对象的状态变为resolve以后的回调函数。回调函数的参数，则是一个具有value和done两个属性的对象，这个跟同步遍历器是一样的。

我们知道，一个对象的同步遍历器的接口，部署在Symbol.iterator属性上面。同样地，对象的异步遍历器接口，部署在Symbol.asyncIterator属性上面。不管是什么样的对象，只要它的Symbol.asyncIterator属性有值，就表示应该对它进行异步遍历。

下面是一个异步遍历器的例子。

const asyncIterable = createAsyncIterable(['a', 'b']);

const asyncIterator = asyncIterable[Symbol.asyncIterator]();

asyncIterator

.next()

.then(iterResult1 => {

console.log(iterResult1); // { value: 'a', done: false }

return asyncIterator.next();

})

.then(iterResult2 => {

console.log(iterResult2); // { value: 'b', done: false }

return asyncIterator.next();

})

.then(iterResult3 => {

console.log(iterResult3); // { value: undefined, done: true }

});

上面代码中，异步遍历器其实返回了两次值。第一次调用的时候，返回一个 Promise 对象；等到 Promise 对象resolve了，再返回一个表示当前数据成员信息的对象。这就是说，异步遍历器与同步遍历器最终行为是一致的，只是会先返回 Promise 对象，作为中介。

由于异步遍历器的next方法，返回的是一个 Promise 对象。因此，可以把它放在await命令后面。

async function f() {

const asyncIterable = createAsyncIterable(['a', 'b']);

const asyncIterator = asyncIterable[Symbol.asyncIterator]();

console.log(await asyncIterator.next());

// { value: 'a', done: false }

console.log(await asyncIterator.next());

// { value: 'b', done: false }

console.log(await asyncIterator.next());

// { value: undefined, done: true }

}

上面代码中，next方法用await处理以后，就不必使用then方法了。整个流程已经很接近同步处理了。

注意，异步遍历器的next方法是可以连续调用的，不必等到上一步产生的 Promise 对象resolve以后再调用。这种情况下，next方法会累积起来，自动按照每一步的顺序运行下去。下面是一个例子，把所有的next方法放在Promise.all方法里面。

const asyncGenObj = createAsyncIterable(['a', 'b']);

const [{value: v1}, {value: v2}] = await Promise.all([

asyncGenObj.next(), asyncGenObj.next()

]);

console.log(v1, v2); // a b

另一种用法是一次性调用所有的next方法，然后await最后一步操作。

async function runner() {

const writer = openFile('someFile.txt');

writer.next('hello');

writer.next('world');

await writer.return();

}

runner();

###### for await...of

前面介绍过，for...of循环用于遍历同步的 Iterator 接口。新引入的for await...of循环，则是用于遍历异步的 Iterator 接口。

async function f() {

for await (const x of createAsyncIterable(['a', 'b'])) {

console.log(x);

}

}

// a

// b

上面代码中，createAsyncIterable()返回一个异步遍历器，for...of循环自动调用这个遍历器的next方法，会得到一个 Promise 对象。await用来处理这个 Promise 对象，一旦resolve，就把得到的值（x）传入for...of的循环体。

for await...of循环的一个用途，是部署了 asyncIterable 操作的异步接口，可以直接放入这个循环。

let body = '';

async function f() {

for await(const data of req) body += data;

const parsed = JSON.parse(body);

console.log('got', parsed);

}

上面代码中，req是一个 asyncIterable 对象，用来异步读取数据。可以看到，使用for await...of循环以后，代码会非常简洁。

如果next方法返回的 Promise 对象被reject，for await...of就会报错，要用try...catch捕捉。

async function () {

try {

for await (const x of createRejectingIterable()) {

console.log(x);

}

} catch (e) {

console.error(e);

}

}

注意，for await...of循环也可以用于同步遍历器。

(async function () {

for await (const x of ['a', 'b']) {

console.log(x);

}

})();

// a

// b

###### 异步 Generator 函数

就像 Generator 函数返回一个同步遍历器对象一样，异步 Generator 函数的作用，是返回一个异步遍历器对象。

在语法上，异步 Generator 函数就是async函数与 Generator 函数的结合。

async function\* gen() {

yield 'hello';

}

const genObj = gen();

genObj.next().then(x => console.log(x));

// { value: 'hello', done: false }

上面代码中，gen是一个异步 Generator 函数，执行后返回一个异步 Iterator 对象。对该对象调用next方法，返回一个 Promise 对象。

异步遍历器的设计目的之一，就是 Generator 函数处理同步操作和异步操作时，能够使用同一套接口。

// 同步 Generator 函数

function\* map(iterable, func) {

const iter = iterable[Symbol.iterator]();

while (true) {

const {value, done} = iter.next();

if (done) break;

yield func(value);

}

}

// 异步 Generator 函数

async function\* map(iterable, func) {

const iter = iterable[Symbol.asyncIterator]();

while (true) {

const {value, done} = await iter.next();

if (done) break;

yield func(value);

}

}

上面代码中，map是一个 Generator 函数，第一个参数是可遍历对象iterable，第二个参数是一个回调函数func。map的作用是将iterable每一步返回的值，使用func进行处理。上面有两个版本的map，前一个处理同步遍历器，后一个处理异步遍历器，可以看到两个版本的写法基本上是一致的。

下面是另一个异步 Generator 函数的例子。

async function\* readLines(path) {

let file = await fileOpen(path);

try {

while (!file.EOF) {

yield await file.readLine();

}

} finally {

await file.close();

}

}

上面代码中，异步操作前面使用await关键字标明，即await后面的操作，应该返回 Promise 对象。凡是使用yield关键字的地方，就是next方法停下来的地方，它后面的表达式的值（即await file.readLine()的值），会作为next()返回对象的value属性，这一点是与同步 Generator 函数一致的。

异步 Generator 函数内部，能够同时使用await和yield命令。可以这样理解，await命令用于将外部操作产生的值输入函数内部，yield命令用于将函数内部的值输出。

上面代码定义的异步 Generator 函数的用法如下。

(async function () {

for await (const line of readLines(filePath)) {

console.log(line);

}

})()

异步 Generator 函数可以与for await...of循环结合起来使用。

async function\* prefixLines(asyncIterable) {

for await (const line of asyncIterable) {

yield '> ' + line;

}

}

异步 Generator 函数的返回值是一个异步 Iterator，即每次调用它的next方法，会返回一个 Promise 对象，也就是说，跟在yield命令后面的，应该是一个 Promise 对象。

function fetchRandom() {

const url = 'https://www.random.org/decimal-fractions/'

+ '?num=1&dec=10&col=1&format=plain&rnd=new';

return fetch(url);

}

async function\* asyncGenerator() {

console.log('Start');

const result = await fetchRandom(); // (A)

yield 'Result: ' + await result.text(); // (B)

console.log('Done');

}

const ag = asyncGenerator();

ag.next().then(({value, done}) => {

console.log(value);

})

上面代码中，ag是asyncGenerator函数返回的异步遍历器对象。调用ag.next()以后，上面代码的执行顺序如下。

* 1. ag.next()立刻返回一个 Promise 对象。
  2. asyncGenerator函数开始执行，打印出Start。
  3. await命令返回一个 Promise 对象，asyncGenerator函数停在这里。
  4. A 处变成 fulfilled 状态，产生的值放入result变量，asyncGenerator函数继续往下执行。
  5. 函数在 B 处的yield暂停执行，一旦yield命令取到值，ag.next()返回的那个 Promise 对象变成 fulfilled 状态。
  6. ag.next()后面的then方法指定的回调函数开始执行。该回调函数的参数是一个对象{value, done}，其中value的值是yield命令后面的那个表达式的值，done的值是false。

A 和 B 两行的作用类似于下面的代码。

return new Promise((resolve, reject) => {

fetchRandom()

.then(result => result.text())

.then(result => {

resolve({

value: 'Result: ' + result,

done: false,

});

});

});

如果异步 Generator 函数抛出错误，会导致 Promise 对象的状态变为reject，然后抛出的错误被catch方法捕获。

async function\* asyncGenerator() {

throw new Error('Problem!');

}

asyncGenerator()

.next()

.catch(err => console.log(err)); // Error: Problem!

注意，普通的 async 函数返回的是一个 Promise 对象，而异步 Generator 函数返回的是一个异步 Iterator 对象。可以这样理解，async 函数和异步 Generator 函数，是封装异步操作的两种方法，都用来达到同一种目的。区别在于，前者自带执行器，后者通过for await...of执行，或者自己编写执行器。下面就是一个异步 Generator 函数的执行器。

async function takeAsync(asyncIterable, count = Infinity) {

const result = [];

const iterator = asyncIterable[Symbol.asyncIterator]();

while (result.length < count) {

const {value, done} = await iterator.next();

if (done) break;

result.push(value);

}

return result;

}

上面代码中，异步 Generator 函数产生的异步遍历器，会通过while循环自动执行，每当await iterator.next()完成，就会进入下一轮循环。一旦done属性变为true，就会跳出循环，异步遍历器执行结束。

下面是这个自动执行器的一个使用实例。

async function f() {

async function\* gen() {

yield 'a';

yield 'b';

yield 'c';

}

return await takeAsync(gen());

}

f().then(function (result) {

console.log(result); // ['a', 'b', 'c']

})

异步 Generator 函数出现以后，JavaScript 就有了四种函数形式：普通函数、async 函数、Generator 函数和异步 Generator 函数。请注意区分每种函数的不同之处。基本上，如果是一系列按照顺序执行的异步操作（比如读取文件，然后写入新内容，再存入硬盘），可以使用 async 函数；如果是一系列产生相同数据结构的异步操作（比如一行一行读取文件），可以使用异步 Generator 函数。

异步 Generator 函数也可以通过next方法的参数，接收外部传入的数据。

const writer = openFile('someFile.txt');

writer.next('hello'); // 立即执行

writer.next('world'); // 立即执行

await writer.return(); // 等待写入结束

上面代码中，openFile是一个异步 Generator 函数。next方法的参数，向该函数内部的操作传入数据。每次next方法都是同步执行的，最后的await命令用于等待整个写入操作结束。

最后，同步的数据结构，也可以使用异步 Generator 函数。

async function\* createAsyncIterable(syncIterable) {

for (const elem of syncIterable) {

yield elem;

}

}

上面代码中，由于没有异步操作，所以也就没有使用await关键字。

###### yield\* 语句

yield\*语句也可以跟一个异步遍历器。

async function\* gen1() {

yield 'a';

yield 'b';

return 2;

}

async function\* gen2() {

// result 最终会等于 2

const result = yield\* gen1();

}

上面代码中，gen2函数里面的result变量，最后的值是2。

与同步 Generator 函数一样，for await...of循环会展开yield\*。

(async function () {

for await (const x of gen2()) {

console.log(x);

}

})();

// a

// b

### 1.1.2使用Babel

#### 1、认识babel

利用babel就可以让我们在当前项目中随意使用这些最新的ES6语法特性。

在目录下添加package.json文件

命令：npm init

安装Babel CL1,这是一个在命令行中使用的babel编译命令。

npm install babel-cli -g

babel es6.js –o compiled.js//将es6.js代码编译到compiled.js中

1、run npm config get proxy看看拿到什么

2、如果上面的返回不为空，npm config set proxy null或者npm config delete proxy ,顺便把https的代理也关掉npm config set https-proxy null, 证书问题，把证书的校验关掉npm config set strict-ssl false, 当然上面的命令你不用全敲一遍，根据报错信息提示，选择对应的解决方式， 然后再试试

3、如果1的命令为空，检查你的环境变量里面是不是有HTTP\_PROXY

4、是不是开了全局的代理

#### 2、配置babel

Babel是通过安装插件（plugin）或者预设（preset,就是设定好的插件）来编译代码的

**预设preset**

创建一个配置文件.babelrc

{

“presets”:[],

“plugins”:[]

}

安装预设命令，它可以把ES6代码编译成ES5

npm install –save-dev babel-preset-es2015

将这个preset填充到配置文件中

{

“presets”:[‘es2015’],

“plugins”:[]

}

这时可以编译了。

**安装插件plugin**

编译ES7安装object-rest-spread插件

npm install babel-plugin-transform-object-rest-spread --save-dev

添加到配置文件：

｛

“presets”:[‘es2015’],

“plugins”:[‘transform-object-rest-spread ‘]

｝

ES7代码

let mike = {name:'nike',age:40};

mike = {...mike,sex:'male'};

console.log(mike);

babel核心概念就是利用一系列的plugin来管理编译规则，不仅ES6还可以编译React JSX，CoffeeScript，甚至ES7的一些特性。后面会介绍到webpact、React如何共同构建一个完美开发环境。

#### 3、小结

ES6可以通过babel适应各版本兼容浏览器的问题。

## 1.2、前端组件化方案

区分两个概念：模块（module）、与组件（component）。

模块是语言层面的，在前端领域我们说的module一般都是指JavaScript module，往往表为一个单独的JS文件，对外暴露一些属性或方法。

前端组件更多是业务层面的概念，可以看成一个独立使用的功能实现，往往表现为一个UI部件（并不绝对），比如一个下拉菜单，一个富文件编辑器或者一个路由系统。一个组件包含它所需要的所有资源，包括逻辑（javaScript）、样式（CSS）、模板（HTML/template）,甚至图片与文件。

因而，一个组件有时仅仅是一个JavaScript模块，而更多时候不仅是一个JavaScript模块。前端组件化方案都不可避免要以JavaScript的模块化方案为基础。

### 1.2.1 JavaScript 模块化方案

有ES6之前，JavaScirpt并没有原生的模块，JavaScript开发者通过各种约定或妥协实现了模块特征，如独立命名空间、暴露属性与方法的能力等。粗略分析的话，这一过程大致经历了案个阶段：全局变量+命名空间（namespace）、AMD&CommonJs、ES6模块。

#### 1、全局变量+命名空间（namespace）

第一阶段基于全局变量，各模块按照各自的命名空间时行挂载。典型例子如：jQuery，又如整个项目使用同一个全局变更window.foo，项目中对应的其他角色，如模块bar，其产出就挂载到foo.bar上。

const foo = window.foo;

const bar = ‘i\’m bar’;

foo.bar= bar;

模块内部一般通过简单的自执行函数实现局部作用域，避免污染全局作用域，因此一个模块外观往往如下。

var function1 = （function(){

//define & export…

}）()

这么做的问题很多，比较主要的问题如下。

1. 依赖全局变量，污染全局作用域的同时，安全性得不到保障。
2. 依赖约定命名空间来避免冲突，可靠性不高。
3. 需要依赖手动管理并控制执行顺序，容易出错。
4. 需要在最终上线前手动合并所有用到的模块。

#### 2、AMD&CommonJS

AMD将革命性的JavaScript模块化方案带到了前端开发中，它解决了前面方案的几乎所有问题。

1. 仅仅需要在全局环境下定义require和define,不需要其他的全局变量。
2. 通过文件路径或模块自己声明的模块名定位模块。
3. 模块实现中声明依赖，依赖的加载与执行均由加载器操作。
4. 提供了打包工具自动分析依赖并合并。

AMD模块一般如下：

define(function (require){

//通过相对路径 获得依赖模块

const bar = require(‘./bar’’);

//从模块产出

return function (){

//…

}

})

至于CommonJs规范，它本不适合浏览器环境，但依赖现代打包工具的能力，CommonJS规范的模块也可以经过转换后在浏览器中执行。相比AMD模块格式，CommonJS的模块格式更简洁，而且可以更方便的实现前后端代码共用（Node.js的模块正是用CommonJs规范），因而得到了广泛的欢迎。一个典型的CommonJs模块如下。

//通过相对路径获得依赖模块

const bar = require(‘./bar’);

//模块产出

module.exports = function(){

//……

};

#### 3、ES6模块

ES6，即ES2015，为JavaScript世界带来了规范的模块化方案，相比AMD/CommonJS，它更为强大，引用与暴露的方式更多样。而且它支持较复杂的静态分析，使构建工具更细粒度地移除模块实现中的无用代码成为可能（感兴趣的读者可以支了解一种叫“tree shaking”的技术）

基于ES6规范的模块是这样的：

//通过相对路径获得依赖模块

Import bar form ‘./bar’;

//模块产出

export default function(){

//……

}

### 1.2.2前端的模块化和组件化

前端的组件化方案在模块化发展的基础上也经历了漫长演变过程。大致可以分划为何个阶段：基于命名空间的多入口文件组件、基于模块的多入口文件组件、单JavaScript入口组件、Web Component。

#### 1、基于命名空间的多入口文件组件

这一方案特点如下：

1. 基于前面介绍的第一种模块化方案。
2. 不同资源分别手动引入（或手动合并）。

最典型的例子就是jQuery插件。首先需要通过手动插入<script>标签引入该插件对应的JavaScript代码，再通过插入<link>标签引入插件的样式内容，然后才可以在我们的代码中使用这个插件。在使用时，插件的实现会向全局的$中添加内容，直接使用$上的方法即可。

#### 2、基于模块的多入口文件组件

后来前端有了流行的模块化方案，这一时期的组件也趋于使用像AMD这样的规范来组织其JavaScript实现，把自己也暴露为一个模块。然而，样式内容及其他的依赖源（图片、字体等）还没能纳入整体的模块化方案里。因此这时的组件往往会呈现为：

1. 一个AMD模块，为JavaScript实现。
2. 一个CSS（或Less、Sass）文件，为样式内容。
3. 其他资源内容，往往不需要手动引入，组件会在其CSS实现中通过相对路径引入。

我们使用时需要：

1. 在JavaScript代码中require组件对应的模块。
2. 在样式代码中引入（CSS预处理器提供的import等方式）组件的样式内容。

不难发现，虽然JavaScript模块化了，但是组件的实现与使用依然不便利。

#### 3、单JavaScript入口组件

browserify、webpack等现代打包工具的出现为解决上一个方案遗留的问题带来了一线曙光。它们允许我们将一般的资源视作与JavaScript平等的模块，并以一致的方式加载进来，这样就可以按如下方式组织组件。

foo/

* img/
* index.js
* style.less

bar/

…

其中的index.js如下。

require(‘./style.less’);

const bar = require(‘./bar’);

module.exports = (){

//……

};

在style.less中以可以通过“./img/foo.png”这样的相对地址引用图片、字体这些依赖。

于是，我们组件的所有依赖都可以在自己实现中声明，而对外只暴露一个JavaScript模块作为入口。以优雅的方式解决已有方案的问题，借助JavaScript强大的表达能力与相关工具使该组件方案拥有了极大的可扩展性，“单JavaScript入口组件”自然成为目前较为主流的前端组件化方案。

#### 4、Web Component

Web Component 是前端组件化方案里的“国家队”，就像ES6 module对于JavaScript模块化方案一样。它于2011年就被提出，但遗憾的是，至今还处于不温不火的状态。先介绍下这个方案，主要包含4部分内容。

1. 自定义元素（Custom Element）
2. HTML模板（HTML Template）
3. Shadow Dom
4. HTML的引入（HTML Import）

拥有这四大本领的Web Component 为我们构造了一个美好的愿景——像使用普通HTML标签一样使用组件，组件的样式内外隔绝，通过简单的<link rel=”import” href=”bar.html”>就可以引入组件实现。

然而，浏览器支持程度迟迟不够，而且很难通过polyfill得以在旧版本浏览器上运行，该方案使用起来与已有的“单JavaScript入口组件”方案相比并无较大优势（样式内容隔绝算是一个，但后者也可以通过约定或工具变相实现样式的隔离），这些让Web Component 方案的前景蒙上了一层不确定性。

### 1.2.3小结

本书中，主要介绍了react提供的组件化方案，它提供了ES5与ES6两个版本，本书将以后者为主进行介绍。React推荐通过webpack或browserify进行应用的构建，搭配对应的loader或plugin可以实现通过JavaScript入口文件统一管理依赖资源。从整体上看，这是一个典型的“单JavaScript入口组件”方案。

# 第二章 webpack

webpack功能强大、配置灵活，特有的code spliting方案正戳中了大规模复杂Web应用的痛点，简单的loader/plugin 开发使它很快拥有了丰富的配套工具与生态。

## 2.1 webpack的特点与优势

### 2.1.1 webpack与RequireJs、browserify

### 2.1.2模块规范

### 2.1.3非javascrip模块支持

### 2.1.4构建产物

### 2.1.5使用

安装命令：npm install webpack -g

webpack main.js bundle.js

特点：支持部分命令行参数形式的配置项，但是其主要配置信息需要通过额外的文件（默认是webpack.config.js）进行配置。这个文件只需要是一个Node.js模块，且export一个JavaScript对象作为配置信息。相比命令行配置，更为灵活强大，因为配置会在Node.js环境中运行，甚至可以在其中require其他模块，这样对复杂项目中不同任务的配置信息进行组织变得更容易。例如，可以实现一个webpack.config.common.js，然后分别实现webpack.config.dev.js与webpack.config.prod.js，用于开发环境和生产环境构建（通过命令行参数指定配置文件），后两者可以直接通过require使用webpack.config.common.js中的公共配置信息，并在此基础上添加或修改以实现各自特有的部分。 webpack众多的配置项、强大的配置方式以及丰富的插件体系，大多数时候，我们仅仅书写配置文件，然后通过命令行工具就可以完成项目的构建工作。不过webpack也提供了Node.js的API，使用也很简单。

war webpack = require(“webpack”);

//返回一个Compiler实例

webpack({

//webpack配置

}，function(err,stats){

//……

});

### 2.1.6 webpack的特色

#### 1、代码拆分（code splitting）方案

单个文件体积过大会导致应用初始加载缓慢。尤其如果其中逻辑中在特定特定情况下需要执行，每次都完整加载所有模块就变得很浪费。webpack提供代码代码拆分的方案，可以将应用于代码拆分为我个志（chunk）,每个块包含一个或多个模块，块可以按需被异步加载。

#### 2、智能的静态分析

webpack支持简单的不含变量的表达式，如下：

require(expr?”a”:”b”);

require(“a”+”b”);

require(“not a”substr(4).replace(“a”,”b”));

其次，webpack还支持含变量的简单表达式，如下：

require(“./template/”+name+”.jade”);

这种情况下，webpack会从表达式”./template/”+name+”.jade”中提取出下信息。

1. 目录.、./template下。
2. 相对路径符合正则表达式：/^.\*\.jade$/。

#### 3、模块热替换（Hot Module Replacement）

webpack提供了模块热替换的能力，它使得在修改完某一模块后无须刷新页面，即可动态将受影响的模块替换为新模块，在后续的执行中使用新逻辑。

这一功能需要配合修改module本身，但一些第三方工具已经帮我们做了这些工作。如配合style-loader,新式模块可以被热替换；配合react-hot-loader,可以对React class模块进行热替换。

当然，配置webpack启动这一功能也相当简单，通过参数—hot启动webpack-dev-server即可。

webpack-dev-server –hot

### 2.1.7小结

## 2.2基于webpack进行开发

### 2.2.1安装

安装命令：

npm install webpack –g 或者指定版本号npm install web [pack@1.12.14](mailto:pack@1.12.14) -g

大部分情况需要以命令行工具的形式使用webpack,所以我们这里将它安装在全局，方便使用。有时候会希望编写自己的构建脚本，或是由项目指定需要依赖webpack，在这种情况下将webpack安装到本地会更合适。对前端webpack扮演的构建工具的角色，并不是代码依赖，应该被安装在dev-dependencies中，即：

npm install webpack --save-dev

本章最终完成示例代码都在https://github.com/vikingmute/webpack-react-codes/tree/master/chapter2,

### 2.2.2 Hello world

这个示例中，将使用webpack构建一个简单的Hello world应用。应用包括两个JavaScript模块（完整代码见chapter2/part1/）。

1.生成文本“Hello world！”的hello模块（hello.js）。

module.exports = ‘hello world’;

……

总结一下的话，webpack主要做了两部分工作，如下：

1. 分析得到所有必需模块并合并。
2. 提供了让这些模块有序、正常执行的环境。

### 2.2.3 使用loader

loaders are transformations that are applied on a resource file of your app. They are functions (running in node.js) that take the source of a resource file as theparameter and return the new source.

loader 是作用于应用中资源文件的转换行为。它们是函数（运行在Node.js环境中），接收资源文件的源代码作为参数，并返回新的代码。

例如：通过jax-loader将React的JSX代码转换为JS代码，从而被浏览器执行。

本节中，将以CSS为例，介绍一下loader的功能和使用（完整代码见chapter2/part2/）。

在webpack中，每个loader往往表现为一个命名为xxx-loader的npm包，针对特定的资源类型（xxx）进行转换。面为了CSS资源添加到项目中，下面介绍两个loader:style-loader与css-loader。前者将CSS代码以<style>标签的形式插入到页面上从而生效率；后者通过检查CSS代码中的import语句找到依赖合并。大部分情况下，我们将二者搭配使用。首先要安装这两个loader对应的npm包（你需要在目录下添加package.json文件或通过npm init自动生成。）

命令：npm install style-loader css-loader --save-dev

index.css

div {

width: 100px;

height: 100px;

background-color: red;

}

index.hmtl

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>loader</title>

</head>

<body>

<script src="./bundle.js"></script>

</body>

</html>

index.js

require('style!css!./index.css');

document.body.appendChild(document.createElement('div'));

注意这里的style!css!,类似xxx!这样写法是为了指定特定loader。这里是告诉webpack使用style-loader及css-loader这两个loader对index.css的内容进行整理。

2.2.4配置文件

配置文件可以在javaScript代码中require CSS模块的时候，不用每次都写一遍style!css!了

webpack.config.js

var path = require('path');

module.exports = {

entry:path.join(\_\_dirname,'index'),//项目入口文件

output:{//输出目录

path:\_\_dirname,

filename:'bundle.js',//输出文件名

publicPath:””//输出目录所对应的外部路径http://cdn.example.com/static/(bundle.js)

},

module:{

loaders:[

{

test:/\.css$/,

loaders:['style-loader','css-loader']

}

]

}

}

### 2.25作用plugin

plugin作用是为了完成loader完成不了的功能，如自动生项目的HTML页面在（HtmlWebpackPlugin）、向构建过程中注入环境变量（EnvironmentPlugin）、向块（chunk）的结果文件中添加注释信息（BannerPlugin）等。

#### 1、HtmlWebpackPlugin

npm i html-webpack-plugin --save-dev

上面例子中的构建生成结果文件bundle.js。一方面，budle.js是在webpack.config.js中配置的output.filename的值，在这里直接取固定值不方便后续维护者;另一方面，为了充分利用浏览器的缓存，提高页面的加载速度，在生产环境中常常会向静态文件的文件名添加MD5戳，即使用bundle\_[hash].js而不是bundle.js，这里的[hash]会在构建被该chunk内容的MD5结果替换，以实现内容不变则文件名不变，内容改变导致文件名改变。在这样的情况下，在HTML页面中给定结果文件的路径就变得不太现实。

而HtmlWebpackPlugin正是为了解决这一问题而生，它会自动生成一个几乎为空的HTML页面，并向其中注入构建的结果文件路径，即使路径中包含动态的内容，如MD5戳，也能够完美处理。

了解了HtmlWebpackPlugin的能力，下面来将它引入岐途到先前的项目中（完成代码见chapter2/part4）。

#### 2、安装plugin

前面介绍到，webpack会内置一部分plugin，用内置的plugin时，不需要额外安装，直接使用即可。

var webpack = require(‘webpack’);

webpack.BannerPlugin; //这样就可以直接获取BannerPlugin

而HtmlWebpackPluginua并不是内置plugin，它在npm包html-webpack-plugin中实现。注意对不同版本包html-webpack-plugin，其用法与配置格式可能会不一致）。

npm i [html-webpack-plugin@1.7.0](mailto:html-webpack-plugin@1.7.0) –save—dev

安装完后，在webpack.config.js中就可以获取这个插件了。

war HtmlWebpackPlugin = require(“html-webpack-plugin”);

#### 3、配置plugin

配置HtmlWebpackPlugin，plugin相关配置对应webpack配置信息中的plugins字段，它的值要求是一个数组，数组的第一荐为一个plugin实例。

var path = require(‘path’);

var HtmlWebpackPlugin = require(‘html-webpack-plugin’);

module.exports = {

entry:path.join(\_\_dirname,’index’),

ouput:{

path:\_\_dirname,

filename:’bundle.js’

},

nodule:{

loaders:[

{

test:/\.css$/,

loaders:[‘style-loader’,’css-loader’]

}

]

},

plugins:[

new HtmlWebpackPlugin({

title:’use plugin’

})

]

};

我们创造了一个HtmlWebpackPlugin实例，并将其添加进了配置信息的plugins字段。在实例化时，传入了｛title:’use plugin’｝，这是传递给HtmlWebpackPlugin的配置信息，它告诉HtmlWebpackPlugin给生的HTML页面设置设置<title>的内容为use plugin。这样原来的index.html就可以删除了。在构建完成后这个插件会自动在output目录（在这里即当前目录）下生成文件index.html。

再次执行构建命令webpack，便可以看到效果。

npm uninstall -g webpack npm install --save-dev html-webpack-plugin webpack webpack-dev-server

npm i webpack -g; npm link webpack --save-dev

### 2.2.6 实时构建

在webpack中，通过添加—watch选项即可开启监视功能，webpack会首先进行一次构建，然后依据依赖关系，对项目所依赖的所有有文件进行监听，一旦发生改动则触发重新构建。命令可简写成：

webpack –w

除了watch模式外，webpack 还提供了webpack-dev-server来辅助开发与调试。webpack-dev-server是一个基于Express框架的Node.js服务器。它还提供了一个客户端的运行环境，会被注入到页面代码中执行，并通过Socket.IO与服务器通信。这样服务器每次改动与重新构建都会被通知到页面上，页面可以随之做出反应。除了最基本的自动刷新，还提供有如模块热替换（Hot Module Replacement）这样强大的功能。

作用webpack-dev-server需要额外安装webpack-dev-server包。

npm install webpack-dev-server –g

然后启动webpack-dev-server即可。

webpack-dev-server

webpack-dev-server默认会监听8080端口，因此直接在浏览器里打开loalhost:8080,即可以看到结果页面。

webpack-dev-server的配置，既可以通过命令行参数的形式传递，也可以通过在webpack.config.js的export中添加字段devServer实现。详细的使用可以参考webpack的官方文档。

# 第三章 初始React

React是Facebook推出的一个JavaScript库，它的口号就是“用来创建用户界面的JavaScript库”，所怪它只是和用户的界面打交道，你可以把它看成MVC中的V（视图）这一层。

简单来说，React有三大颠覆性的特点。

## 3.0简述

#### 组件

React的一切都是基于组件的。

一个组件的例子如下。

//Profile.jsx

import React from ‘react’;

export default Class Profile extends React.Component{

render(){

return(

<div className = “profile-component”>

<h2>HI, I am {this.props.name}</h2>

</div>

)

}

}

用这种方式，就实现了一个React的组件，在其他的组件中，可以像HTML标签一样引用它。

import Profile from ‘./profile’;

export default function(){

return(

<Profile/>

)

}

#### 2、JSX

通过上面的例子可以看出，在render方法中有一种直接把HTML嵌套在JS中的写法，它被称为JSX。这种类似XML的写法，它可以定义类似HTML一样简洁的树状结构。这种语法结合了JavaScript和HTML的优点，既可以像平常一样使用HTML,也可以在里面嵌套JavaScript语法。这种友好的格式，让开发者易于阅读和开发。而且，对于组件来说，直接使用类似HTML的格式，也是非常合理的。但是，需要注意的是。JSX和HTML完成不是一回事，JSX只是作为编译器，把类似HTML的结构编译成JavaScript。

当然，在浏览器中不能直接使用这种格式工，需要添加JSX编译器来完成这项工作。

#### 3、Virtual DOM

在React的设计中，开发者不太需要操作真正的DOM节点，每个React组件都是用Virtual DOM渲染，它是一种对HTML DOM节点的抽象描述，你可以把它看成是种用JavaScript实现的结构，它不需要浏览器的DOM API支持，所以它在Node.js中也可以使用。它和DOM的一大区别就是它采用了更高效的渲染方式，组件的DOM结构映射到Virtual DOM上，当需要重新渲染组件的时，React在Virtual DOM上实现了一个Diff算法，通过这个算法寻找需要变更的节点，再把里面的修改更新到实际需要修改的DOM节点上，这样就避免了整个渲染DOM带来的巨大成本。

React改变了传统前端开发的固定模式，当然还有React Native，把它的魔力扩展到Web应用以外。在下面章节里，会根据这三大特性做详细的讲解。

## 3.1 使用React与传统前端开发的比较

React到底给前端带来了什么改变，是什么支撑我们接受这样一个新事物的学习成本，把它的这群引入到工作中。

3.1.1传统做法

下面使用传统的方式来实现这个列表的逻辑。

//列表容器

const wrapper = $(‘#list-wrapper’);

//列表所需数据

const data{

list:[1,2,3],

activeIndex:-1

};

//初始化行为

function init(){

wrapper.on(‘click’, ’li’, function(){

activate($(this).data(‘index’));

});

wrapper.html(template.render(data));

}

//选中某项的行为

function activate(index){

wrapper.find(‘li’).removeClass(‘active’);

wrapper.find(‘ li[data-index=’ + index + ’ ] ’).addClass(‘active’);

data.activeIndex = index;

}

//选中某项的行为

<ul>

{{each list item index}}

{{if index === activeIndex}}

<li data-index=”{{index}}” class=”active”>{{item}}</li>

{{else}}

<li data-index=”{{index}}”>{{item}}</li>

{{/if}}

{{/each}}

</ul>

列表组件的数据包含两部分：list及activeIndex。list是列表内容，activeIndex是当前选中项的索引。初始化（init）的时候，使用data渲染模板来产生原始DOM。在后续的交互中，每次单击列表项都触发一次选中（activate）,activate方法会移除其他项的active状态，并给本次选中的项添加active状态，它同时也会更新activeIndex信息，使其与DOM元素的表现相符。

我们习惯了以这种方式实现UI组件，而忽略了它严重的缺陷：需要同时维护数据及视图。模板引擎帮助我们解决了初始状态下二者的对应关系，即初始化时给出一个合理的数据就行了，视图会被（通过模板渲染）生成。然而，在组件状态变化时，它们依然是需要被各自维护的。

对此，一般的MVVM框架的做法是，通过对模板的分析获取数据与其视图元素（DOM节点）细致而具体的对应关系，然后对数据进行监控，在数据变化时更新对应的视图元素。

MVVM的门槛是独特的语法丰富的指令需要学习掌握，我们可不可以不学习这些，也能得到复用“数据生成视图”逻辑的效果呢？其实有一个简单（然后问题很大）的做法。

### 3.12 全量更新

下面对先前的实现稍做改动：

//列表容器

const wrapper = $(‘#list-wrapper’);

//列表所需数据

const data{

list:[1,2,3],

activeIndex:-1

};

function render(){

wrapper.html(template.render(data));

}

//初始化行为

function init(){

wrapper.on(‘click’, ’li’, function(){

activate($(this).data(‘index’));

});

render();

}

//选中某项的行为

function activate(index){

data.activeIndex = index;

render();

}

可以看到，把初始化时渲染的逻辑单独抽出为render方法，除在init时调用外，在选中某项时，更新数据后，两样可以通过直接调用render方法进行视图的更新。

这种做法，只需要要维护数据，每次数据发生变化的时，再渲染一下就能更新视图。逻辑清晰，更容易被理解和维护。

然而，在真实业务中，这么做的缺陷比第一种做法更突出：

1、每次数据变动都整体重新渲染，性能会非常差，尤其在数据变动频繁、界面复杂时间；

2、每次渲染都重新生成所有的DOM节点，那么在这些DOM节点上绑定的事件及外部持有的对这些DOM节点的引用都将失效。

而React可以有效避免这两种缺陷。

### 3.1.3 使用React

如果用React把这个例子重新实现一遍的话，它可能是如下这样的（为了清晰简单，这里把list也放在组件的state中，在实际的实现中，通过props将list传入组件会更合适）。

class List extends React.Component{

constructor(props){

super(props);

this.state = {

list:[1,2,3],

activeIndex:-1

};

}

activate(index){

this.setState({activeIndex:index});

}

render(){

const{ list,activeIndex } = this.state;

const lis = list.map(

(item,index) = > {

const cls = index === activeIndex ? ‘active’ : “”;

return(

<li

key = {index}

className = {cls}

onClick=>{()=>this.activate(index)}>

</li>

)

}

);

return(

<ul>{lis}</ul>

);

}

}

不难看出，除了React特有的Component接口、字段及JSX语法等，在基于React Componet的实现中，，整体逻辑组成与第二个例子是类似的：首次渲染通过render得到界面，每个li的点击触发activate方法，activate方法调用setState更新状态信息，setState会触发重新render（React提供了这一机制），从而使用界面科到更新。

具体分析之前，首选要说明一点：JS内容的渲染结果其实不是真实的DOM节点，本质上是JavaScript对象的虚拟DOM节点，它记录了这个节点的所有信息，可以依据一定的规则生成的真实DOM节点。

#### 1、性能问题

我们知道前端的性能瓶颈大多数时候都在于操作DOM，所以如果避开操作DOM，只是重新生成虚拟的DOM节点（JavaScript对像），本身是很快的。在将虚拟的DOM对应生成真实DOM节点之前，React会将虚拟的DOM树与先前的进行比较（Diff），计算出变化的部分，再将变化的部分作用到真实DOM上，实现最终界面的更新。得益于Diff算法的高效，整个过程的代价大致接近最终操作真实DOM的代价。与最初示例很接近了。区别在于React以额外的计算机量换取了对于更新点的自动定位，以框架本身复杂的代码实现换取了业务代码逻辑的清晰简单。

#### 2、DOM事件与引用失效

在React的哲学里，直接的操作DOM是典型的反模式。React对DOM事件进行了封装并提供了相应的接口。值得注意的是，React提供的事件绑定接口与其界面声明方式是一脉相承的，事件绑定表现为，值为回调函数的组件属性（props）。这样的好处是，绑定事件的过程自然地变成了界面渲染（render）的一部分，无须特别处理。

在事件绑定与读/写操作都被React通过抽象层屏蔽后，业务代码基本无须接触真实DOM，需要持有引用的场景自然也不复存在，引用句失效也就无从说起。

### 3.1.4 小结

最后总结一下，React的出现允许我们以简单粗暴的方式构建我们的界面：仅仅声明数据到视图瓣转换逻辑，然后维护数据的变动，自动更新视图，它看起来很像每次状态更新时，都需要整体地更新一次视图，但React的抽象层避免了辽一做法带来的弊端，让这一开发方式变得可行。

## 3.2 JSX

### 3.2.1 来历

We strongly believe that components are the right way to separate concerns rather than "templates" and "display" logic". We think that markup and code that generates it are intimately tied together.Additionally,display logic is often very complex and using template languages to express it becomes cumbersome.

多年以来，在传统的开发中，把模板和功能分离看作是最佳事件的完美例子，翻阅形形色色的框架文档，总有一个模板文件里面放置了对应的模板文件，然后通过模板引擎处理这些字符串，来生成把数据和模板结合起来的字符。而React认为世界是基于组件的，组件自然而然和模板相连，把逻辑和模板分开放置是一种笨思路。所以React创造了一种名为JSX的语法格式来架起它们之间的桥梁。

### 3.2.2语法

JSX编译器把类似HTML的写法转换成原生的JavaScript方法，并且会将传入的属性转化为对应的对象。它就类似于一种语法糖，把标签类型的写法转换成React提供的一个用来创建ReactElement的方法。

const MyComponent

//input JSX，在JS中直接写类似HTML的内容，前所未有的感觉。其实它返回的是一个

//ReactElement

let app = <h1 title=”mytitle”>this is my title</h1>;

//JSK围捕的结果是

let app = React.createElement(‘h1,’{title:’my title’},’this is my title’);

#### HTML标签与React组件

React可以直接渲染HTML类型的标签，也可以渲染React的组件。

HTML类型的标签第一个字母用小写来表示。

import React from ‘react’;

//当一个标签里面为空的时候，可以直接使用自闭和标签

//注意class是一个JavaScript保留字，所以如果要写class应该替换成className

let divElement = <div className=”foo” />;

//等同于

let divElement = React.createElement(‘div’,{className:’foo’});

React 组件标签第一个字母用大写来表示

import React from ‘react’;

class Headline extends React.component{

…

render(){

//直接retrun JSX语法

return <h1>Hello React</h1>

}

}

let headline = <Headline />;

//等同于

let headline = React.createElement(headline);

JSX语法使用第一个字母大小写来区分是一个普通的HTML标签还是一个React组件

注意：因为JSX本身是JavaSript语法，所以一些JavaScript中的保留字要用其他的方式书写，比如第一例子中class要写成className。

#### JavaScript表达式

在给组件传入属性的时候，有一大部分的情况是要传入一个JavaScript对象的，那么基本的规则就是当遇到｛｝这个表达式的情况下，里面的代码会被当作JavaScript代码处理。

const MyComponent;

let isLoggedIn = true;

let app = <MyComponent name={isLoggedIn?’viking’:’please login’} />

子组表达式如下。

const MyComponent, LoginForm, Nav;

let isLoggedIn = true;

let app = <MyComponent>{isLoggedIn?<Nav/>:<LoginForm/>}</MyComponent>

由上面两个下馆子 可以得到一个基本规律。在JSX语法中，当遇到标签的时候就解释成组件或者HTML标签，当遇到｛｝包裹的时候就当成JavaScript代码来执行。

布尔类型属性如下。

当省略一个属性的值的时候，JSX会自动把它的值认为是true。

let myButton = <input type=”button” disabled />;

//等同于

let myButton = <input type=”button” disabled={true} />;

#### 4.注释

除字组件位置的注释需要用｛｝,其他延用JavaScript

let component = {

<div>

{/\*这里是不念旧恶注释\*/}

<Headline />

</div>

｝;

#### 4、JSX属性扩散

假如一个组件有很多属性，当然可以如下这样做。

const Profile;

let name = ‘viking’, age=10, gender=’Male’;

let component = <Profile name={name} age={age} gender={gender}/>;

如果属性过多的时候，可以用JSX提供的一个便利功能-----属性扩散。

const Profile;

let props = {

name:’viking’,

age:10,

gender:’Male’

};

//用这咱方式可以很方便地完成上一个例子里面的操作

let component = <Profile {…props} />;

//也可以和别的属性组合，后面的属性会覆盖前面的属性。

let component = <Profile {…props} name=’viking2’ />;

对像扩散是ES7的属性。

### 3.2.3 编译JSX

JSX不能直接在浏览器中使用，需要一种编译工具把它编译成React.CreateElement方法。

Facebook提供了一个简单的工具为JSXTransformer，它是面向浏览器的，你可以把它直接引入到HTML文档中。

<script src=<http://fb.me/JSXTransformer-0.14.7.js>></script>

<script type = ‘test/jsx’>

//jsx写在这里

</script>

另一种就是Babel，读者可以使用下面在线编译地址

<https://babeljs.io/repl/>。

### 3.2.4小结

JSX看起来就是HTML，每个前端开发者都可以很快地熟悉上手但是，请记住它不是真正的HTML，也和DOM没有关系。它像是一种React.createElement写法的语法糖，是快速高效书写这个函数的方法，它返回的是ReactElement，一种JavaScript的数据结构。

## 3.3React+webpact开发环境

webpackt易于配置，与React配合简单易用，同时引入具有强大魔力的Babel，让你可以轻松地用最新的标准来书写你的React代码。

本章完成的完整代码示例可以参考<https://github.com/vikingmute/webpack-react-codes/tree/master/chapter3/part3>。

创建一个项目文件夹，并且用npm init –yes命令生成一个package.json文件作为开始。确认是否安装了webpack和webpack-dev-server,使用npm install webpack webpack-dev-server-g进行全局安装，或者也可以使用npm install webpack webpack-dev-server --save-dev直接作为项目的依赖。

### 3.3.1安装配置babel

babel是一个JavaScript编译器，现在在这里使用babel有两个目的，一个是让代码支持ES6语法，一个是支持React的一些特性（例如JSX语法）。正好有两个presets可以完成这两个任务。

1. babel-preset-es2015 ES6语法包支持ES6新特性，const、箭头操作符等信手拈来。
2. babel-preset-react React 语法包支持React的优化，可以使用React ES6 classes，还支持JSX语法格式。

#### 1、安装Babel loader

在第二章已介绍过webpact中的loader,它作用于文件特定格式的转换，那么在这里需要安装Babel loader。

//安装babel-core核心模块和babel-loader

npm install babel-core babel-loader --save-dev

//安装ES6和React支持

npm install babel-preset-es2015 babel-preset-react --save-dev

#### 2、配置.babelrc

安装完Babel和它的插件，配置一下它的规则。在项目根目录下新建一上.babelrc空文件。用编辑器打开，输入如下内容。

//告诉Babel，编译JavaScript代码的时候要用这两个presets编译。

{

"presets":["es2015","react"]

}

### 3.3.2 安装配置ESLint

ESLint提供了一个完全可配置的检查规则，而且提供了非常多的第三方plugin适合不同的开发场景，输出的错误信息也非常明晰，同时最酷的一点是它有着ES6语法的最佳支持，还支持JSX语法，它简直就是React应用代码规范的绝配。

#### 1、安装ESLint loader

同样为webpack添加这个loader，其实更严格来说，它应该被称为preLoaders。preLoaders，顾名思议，就是在loader处理该资源之前，先用perLoaders进行处理，因为代码检查肯定是要在转换代码之前进行的。

npm install eslint eslint-loader --save-dev

刚才说过ESLint有很多第三方配置好的格式插件，那么在这里使用Airbnb开发配置合集eslint-config-airbnb，这个配置合集里面还包括如下三个插件。

npm install eslint-plugin-import eslint-plugin-react eslint-plugin-jsx-ally --save-dev

npm install eslint-config-airbnb --save-dev

2、配置.exlintrc

{

"extends":"airbnb",

"rules":{

"comma-dangle":["error","never"]

}

}

这个配置文件的意思就是直接继承eslint-config-airbnb的配置规则，同时也可以写入自己写的规则，后面内容覆盖默认的规则。例如"comma-dangle":["error","never"]，在定义这项之前，一个对像或者数组的最后一项是要加逗号的，要写成[1,2,3,4,]才可以，但是对于我个人来说，我喜欢最后一项不加逗号，所以我修改了这条规则，想了解ESLint的更多用法，可以去官方网站一探究竟。

### 3.3.3配置webpack

在项目根目录下新建一个webpack.config.js文件。

var path = require('path');

var webpack = require('webpack');

var HtmlwebpackPlugin = require('html-webpack-plugin');

//一些常用路径

var ROOT\_PATH = path.resolve(\_\_dirname);

var APP\_PATH = path.resolve(ROOT\_PATH,'app');

var BUILD\_PATH = path.resolve(ROOT\_PATH,'build');

module.exports = {

entry:{

app:path.resolve(APP\_PATH,'index.jsx')

},

output:{

path:BUILD\_PATH,

filename:'bundle.js'

},

//开启dev source map

devtool:'eval-source-map',

//开户webpack dev server

devServer:{

historyApiFallback:true,

hot:true,

inline:true,

progress:true

},

module:{

//配置preLoaders,将eslint添加进入

preLoaders:[

{

test:/\.jsx?$/,

loaders:['eslint'],

include:APP\_PATH

}

],

//配置loader,将Babel添加进去

loaders:[

{.

test:/\.jsx?$/,

loaders:['babel'],

include:APP\_PATH

}

]

},

//配置plugin

plugins:[

new HtmlwebpackPlugin({

title:'My first react app'

})

]

}

上面配置文件并不特殊，复习一下第2章的知识，把app文件夹的index.jsx作为入口，用配置好babel-loader处理它，在Babel处理之前先用ESLint检查代码的格式，最后使用HtmlwebpackPlugin在build文件夹中生成处理后的HTML文件。

这里还需要添加一个resolve的参数，把JSX扩展名添加进去，这样就可以在JS中import加载JSX扩展名的角本。

。。。

resolve:{

extension:[‘’,’.js’,’.jsx’]

},

下面在npm中添加webpack启动命令。

npm可以添加自定义命令，将两条命令添加到package.json里面，一个是运行webpack命令、build整个项目，一个是启动本地webpack-dev-server来进行开发：

…

“scripts”:{

“build”:”webpack”,

“div”:”webapack-dev-server --hot”,

},

…

### 3.3.4添加测试页面

把React库添加到项目中。

npm install react react-dom --save

新建app.jsx文件。

import React from 'react';

import ReactDOM from 'react-dom';

function App() {

return (

<div className="container">

<h1>Hello React!</h1>

</div>

);

}

const app = document.createElement('div');

document.body.appendChild(app);

ReactDOM.render(<App />, app);

在根目录下运行npm run dev,webpack-dev-server会新建一个基于Express的服务器，打开浏览器的http：//localhost：8080发现大标题出现，但打开控制台或者终端界面，会发现出出现了一些ESLint的错误，需要进行格式修改。

### 3.3.5 添加组件热加载（HMR）功能

如果更新代码以后可以只更新局部的组件，面对全局页面不要求直接强制刷新，那岂不是非常美妙。

在第二章讲到webpack支持HMR（Hot Module Replacement）,这里自然而然地会想到这种解决方案。不过早就有一些人帮我们做好了类似的工作，只要简单安装一个Babel的preset，就可以轻松地完成这项工作。

npm install babel-preset-react-hmre --save-dev

这个preset里面其实包括两个方面。

1. react-transform-hmr用来实现上面所说的热加载;
2. react-transform-catch-errors用来捕获render里面的方法，并且直接展示界面上。

配置一下.babelrc如下。

{

"presets":["react","es2015"],

"env":{

"development":{

"presets":["react-hmre"]

}

}

}

配置完毕，启动npm run dev。

看一下效果，然后随便改动h1标签里面的文字，发现页面没有被刷新，但是内容自动改动了。在render方法中故意设置一些错误，出现了红色错误提示，大功造成。

3.3.6 小结

本节的内容至此结束。已经设置好的React和webpack的开发环境，它支持的特性有以下几点。

**a、支持ES6语言特性。**

**b、支持JSX语法。**

**c、使用ESLint作为代码检查工具。**

**d、支持HMR热加载。**

有了这几个优点，在下面的React开发章节中，就可以把它作为通用的开发环境了。

其他需要安装的命令：npm install --save babel-polyfill //指的是“用于实现浏览器不支持原生功能的代码”以及API方法和自定义的全局方法

npm install babel-plugin-transform-object-rest-spread //试用于展开运算符

## 3.4组件

组件是React的基石，所有的React应用程序都是基于组件的。

在本节中我们会最终完成一个React组件的设计师，并通过这个例子慢慢展开描述各种关于React组件的概念。

之前的React组件，是使用Eeact.createClass来声明的。

var List = React.createClass({

getInitialState:function(){

return [‘a’,’b’,’c’];

},

render:function(){

return (…);

}

});

第一章描述过，本书代码全都使用了ES6的写法。当然React官方也在第一时间就支持了ES6 class的方法，这种写法可读性更强，一个直观的表现就是不用写getInitialState方法了，可以直接在constructor里面定义this.state的值。所以，以后的代码全部采用这样的格式。

class List extends React.Component {

constructor(){

super();

this.state = [‘a’,’b’,’c’];

}

render() {

return (…);

}

}

现在要建立的是一个个人的页面，称为Profile。最后完成的效果如图书-4所示。



这个例子几乎涵盖了关于React的所有概念。

完整代码参考https://github.com/vikingmute/webpack-react-codes/tree/master/chapter3/part4

### 3.4.1 props 属性

现在来新建一个组件，称为Pofile.jsx。

//Profile.jsx

import React from ‘react’;

export default Class Profile extends React.Component {

//render 是这个组件渲染的Vitrual DOM结构

render(){

return(

<div className=”profile-component”>

{/\*this.props就是传入的属性\*/}

<h1>我的名字叫{this.props.name}</h1>

<h2>我今年{this.props.age}岁</h2>

</div>

)

}

}

它只是简单地输出了一个标题，3.2.2节讲到JSX直接引入HTML标签也能完成相同的效果。

let profile = <div className=”profile-component”>

<h1>我的名字叫viking</h1>

<h2>我今年20岁</h2>

</div>

有了组件以后，可以使用React提供的另外一个库ReactDOM把这个组件挂载到DOM节点上。

//app.jsx

import { render } from ‘react-dom’;

import Profile from ‘./profile’;

render(<Profile name=”viking” age=20 />,document.getElementById(‘container’));

//或者可以使用“…”属性扩展

const props = {

name:’viking’,

age:20

};

render(<Profile {…props} />,document.getElementById(‘container’));

p实现后的截图如图书-5所示。



在JSX一节中已经提到过props就是传入组件的属性，由外部的JSX传入，在组件内部可以通过this.props来访问。在上面的例子中，name、age就是传入的属性，传入多个属性时可以使用“…”属性扩展。

下面来验证组件的属性。

当程序结构变得越来越复杂的时候，组件的复杂程度也会成倍地提高，所以一项很重要的工作就是验证组件传入的属性。比如说上面组件的age属性，应该传入数字类型，那么如果传入一个数组，肯定就会出现问题，所以React可以让用户定义组件属性的变量类型。

import { PropTypes } from ‘react’;

const = ｛

//验证不同类型的JavaScript变量

optionalArray:PropTypes.array,

optionalBool:PropTypes.bool,

optionalFunc:PropTypes.func,

optionalNumber:PropTypes.number,

optionalObject:PropTypes.object,

optionalString:PropTypes.string,

//可以是一个ReactElement类型

optionalElement:PropTypes.element,

//可以是别的组件的实例

optionalMessage:PropTypes.instanceOf(Message),

//可以规定为一组值其中的一个

optionalEnum:PropTypes.oneOf([‘News’,’Photos’]),

//可以规定是一组类型中的一个

optionalUnion:PropTypes.oneOfType([

PropTypes.string,

Proptypes.number,

Proptypes.instanceOf(Message)

]),

//可以在最后加一个isRequired,表明这个属性是必需的，否则就会返回一个错误

requiredFunc:React.PropTypes.func.isRequired

｝

了解了这么多种属性的验证，接下来给刚才简单的组件添加验证。

imort React, {PropTypes } from ‘react’;

//需要验证的属性

const propTypes = {

name:PropTypes.string.isRequired,

age:PropTypes.number.isRequired

};

class Profile extends React.Component {

//render 是这个组件渲染的Vitrual DOM结构

render(){

return(

<div className=”profile-component”>

{/\*this.props就是传入的属性\*/}

<h1>我的名字叫{this.props.name}</h1>

<h2>我今年{this.props.age}岁</h2>

</div>

)

}

}

//将验证赋值给这个组件的propTypes属性

Profile.propTypes = propTypes;

export default Profile;

3.4.2 state 状态

state是组件内部的属性，组件本身是一个状态机，它可以在constructor中通过this.state直接定义它的值，然后根据这些值来渲染不同的UI。当state的值发生改变时，可以通过this.setState方法让组件再次调用render方法，来渲染新的UI。

现在改造一个简单的组件，给它添加一个状态，一个“点赞”的按钮，每单击一次，就给赞的次数加1。

//Profile.jsx

export default class Profile extends React.Component {

constructor(props) {

super(props);

this.state = {

liked:0

};

this.likedCallback = this.likedCallback.bind(this);

}

likedCallback() {

let liked = this.state.liked;

liked++;

this.setState({

liked

});

}

reder() {

return(){

<div>

<h1>我的名字叫{this.props.name}</h1>

<h2>我今年{this.props.age}岁</h2>

<button onClick={this.likedCallback}>给我点赞</button>

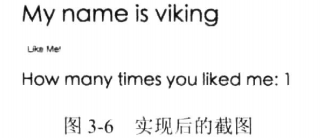
</div>

}

}

}

实现后截图如图3-6



和上面描述的一样，在constructor中添加this.state的定义，每次单击按钮以后调用回调函数，给当前liked值加1，然后更新this.setState，完成UI的重新渲染。因为在ES6 class类型的component组件声明方式中，不会把一些自定义的callback函数绑定到实例上，所以需要手动在constructor里面绑定。

this.likedCallback = this.likedCallback.bind(this);

React组件通过props和state的值，使用render方法生成一个组件的实例。

上面通过单击的event handler可以看到，这种写法和在普通的DOM元素上写事件回调没有任何差异，接受起来完成没有古本。其实这是React自己实现的合成事件，它完成符合W3C的标准，并且处理了不同浏览器之间的兼容性问题。React并未把事件绑定在特定的DOM节点上，实际上它是用事件代理的方式在最外层绑定了一个事件回调，当组件umounted的时候，这个事件回调会被自动删除。

3.4.3 组件生命周期

每个生物都有它自己的生命周期，从出生、少年、成年再到死亡。同理，组件也有它特定的生命周期，React用不同的方法来描述它的整个生命周期。现在，要稍微修改一个组件的代码，当组件加载完毕1秒以后，使liked的值自动加1。

…

componentDidMount(){

setTimeout(() =>{

this.likedCallback();

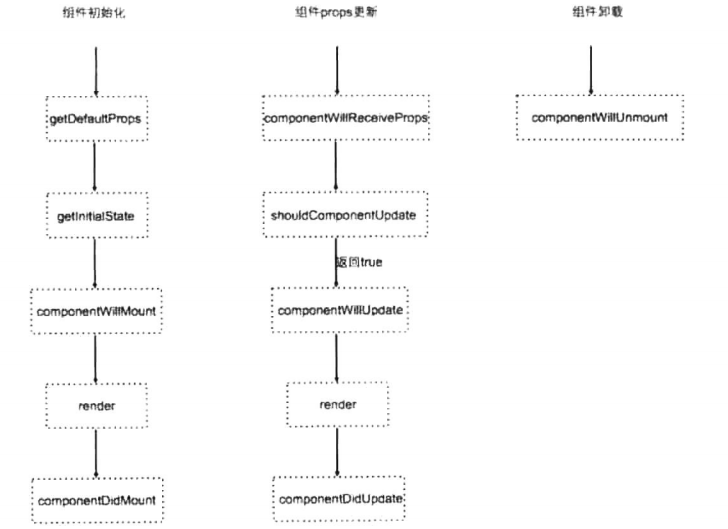
},1000);

}

…

componedtDidMount这个方法就是在render完成并且组件装载完成之后调用的方法，所以界面中先显示0，1秒以后此方法被调用，界面被重新渲染，liked值变成了1。

整个生命周期可以用图书-7描述。



如图3-7所示，生命周期可以分为以下部分。

#### 1、组件首次加载

**getDefaultProps**只会在装载之前调用一次，在组件中赋值的数据会被设置到this.props中。

**getInitalState** 只会装载之前调用一次，这个函数的返回值会被设置到this.state中。需要注意的是，ES6写法中，只需要在constructor中即可，如下。

**comPonentWillMount**在render之前被调用，可以在渲染之前做一些准备工作。

**render**这个方法是组件的一个必要方法。当这个方法被调用的时候，应该返回一个ReactElement对像，该对象在上3.2.2 节中被被提到过。render是一个纯函数，它的意义就是在给定相同的条件时，它的返回结果应该每次都完全一致的。不应该有任何修改组件state的代码或者是和浏览器交互的情况。

**componentDiMount**只会在装载完成之后调用一次，在render之后调用，从这里开始获取组件的DOM结构。如果想让组件加载完毕后做一些额外的操作（比如SJAX请求等），可以在这个方法中添加相应的代码。

#### 2、组件props更新

当组件接收到新的props的时候，会依次触发下列方法。

componentWillReceiveProps(object nextProps)，在组件接收到新的props的时候被触发，参数nextProps就会传入的新的props，你可以用它和this.props比较，来决定是否用this.setState实现UI重新渲染。

shouldComponentUpdate，在重新render之前被调用，可以返回一个布尔值来决定一个组件是否要更新，如果返回false，那么前面的流程都不会被触发。这个方法默认默认的返回值是true。

componentWillUpdate, 在render之前被调用，可以在渲染之前做一些准备工作，和componentWillMount类似。

render，和组件首次加载的方法相同。

componentDidUpdate,重新渲染完成以后立即调用，和componetDidMount类似。

#### 3、组件制裁

componentWillUnmount，在组件被卸载和销毁之前调用的方法，可以在这里做一些清理工作。

### 3.4.4组合组件

React应用建立在各种组件基础上，那么自然地，一个组件也可以包含多个其他组件。现在，扩展一下刚才的应用，显示一个爱好列表那么这个新的组件被称为Hobby.jsx。

Hobby.jsx

import React, {PropTypes} from ‘react’;

const propTypes = {

hobby: PropTypes.string.isRequired

};

class Hobby extends React.Component {

render(){

return <li>{this.props.hobby}</li>

}

}

Hobby.propTypes = propTypes;

export default Hobby;

上例只是输出列表中的一项。现在把它引入到Profile中，在3.2.2节讲过，组件可以直接用标签的形式放入JSX中。

//Profile.jsx

import Hobby from ‘./hobby’;

…

constructor(props){

super(props);

//在state中添加两个爱好

this.state = {

liked:0,

hobbies:[‘skateboarding’,’rock music’]

};

…

render(){

return(

<div>

<h1>我的名字叫{this.props.name}</h1>

<h2>我今年{this.props.age}岁</h2>

<button onClick={this.likedCallback}>给我点赞</button>

<h2>总点赞数{this.state.liked}</h2>

<h2>我的爱好：</h2>

<u1>

{this.state.hobbies.map((hobby, i)=><Hobby key={i} hobby={hobby} />)}

</u1>

</div>

)

}

}

…

只要将子组件看成自定义HTML标签就好了，然后传入想要的属性，特别注意要给每个循环组件添加一个唯一的key值。实现后的截图如图3-8所示。



3.4.5无状态函数组件

Hobby组件非常简单，没有内部state，不需要组件生命周期函数，那么，可以把这类组件写成一个纯函数的形式，称为stateless functional component（无状态函数组件）。它做的事情只是根据输入生成组件，没有其他的副作用，而且简洁明了。

//用一个纯函数表示组件

function Hobby(props){

function Hobby(props){

return <li>{props.hobby}</li>;

}

}

这种写法很简单，直接导出一个函数，它只有一个参数props，就是传入的属性。在实际的项目中，大部分的组件都是无状态函数式组件，所以这是React推荐的写法。在以后的版本中也会对这种形式进行优化。

### 3.4.6 sate设计原则

什么组件应该是state，而且应该遵循最小化state的准则？那就是习题让大多数的组件无状态的。为了实现这样的结构，应该尽量把状态分离在一些特定的组件中，来降低组件的复杂度。最常见的做法是创建尽量多的无状态组件，这些组件唯一关心的事情就是渲染数据。而在这些组件的外层，应该会有一个包含state的低级别的组件。这个组件用于处理各种事件、交流逻辑、修改state，对应的子组件要关心的只是传入的属性而已。

state应该包含什么数据？sate中应该包含组件的事件回调函数可能引发UI更新的这类数据。在实际的项目中，这些应该是轻量化的JSON数据，应该尽量把数据的表现设计到最小，而更多的数据可以在render方法中通过各种计算来得到。这里举一个例子，比如说现在有一个下例子，比如说现在有一个商品列表，还有一个用户已经选购的商品列表。最直观的设计方法是如下这样的。

goods:[{‘id’:1,’name’:’paper’},{‘id’:2,’name’:’pencil’}…],

selectedGoods:[{ ‘id’:1,’title’:’hello world’}]

sate不应该包函的数据

1. 可以由state计算得出的数据。就像刚才的selectedGoods一样，可以由goods列表计算得到。
2. 组件。组件不需要保存到state中，只需要在render方法中渲染。
3. props中的数据。props可以看作是组件的数据来源，它不需要保存在state中。

### 3.4.7 DOM操作

在大多数情况下，不需要通过操作DOM的方式去更新UI，应该使用setState来重新渲染UI。但是，有一些情况确实需要访问一些DOM结构（例如表单的值），那么可以采用refs这种方式来获得DOM节点，它的做法就是在要应用的节点上面设置一个ref属性，然后通过this.refs.name获得对应的DOM结构。

继续改造程序，添加一个输入框和一个按钮，能完成添加爱好的功能。

//Profile.jsx

render() {

return(

<div>

…

<input type=”test” ref=”hobby” />

<button onClick={this.addHobbyCallback}>添加爱好</button>

</div>

)

}

在button上添加事件———取得input的值，添加到state的值里面。下面来完成这个单击的回调。

//Profile.jsx

addHobbyCallback() {

//用this.refs.name来取得DOM节点

let hobbyInput = this.refs.hobby:

let val = hobbyInput.value;

if(val){

let hobbies = this.state.hobbies;

//添加值到数组

hobbies = […hobbies, val];

//更新state，刷新UI

this.setState({hobies}, () =>｛hobbyInput.value = ‘’;｝);

}

}

完成后截图如图书-9所示。



通过这个例子了解到了组件的声明、状态和属性、生命周期、与DOM的交互和事件，还有组合组件的用法。组件是React的核心所在，一个基于React的项目是由各种各样不同的组件构成的。

## 3.5 Virtual DOM

### 3.5.1 DOM

### 3.5.2虚拟元素

### 3.53比较差异

# 第四章 实践的React

## 4.1开发项目

开发项目时，我们要做的第一件事情不是写代码，而是要画一个简单的原型图，将不同的功能可视化地表现出来。

本章完整的代码示例可以参考<https://github.com/vikingmute/webpack-react-codes/tree/master/chapter4/part1>。

### 4.1.1将原型图分割成不同组件。

对于 项目的结构如下规定，所有的组件都放到新建好的components文件夹下，每下组件新建一个文件夹，组件的命名统一采用index.jsx的形式，同时样式文件命名为style.scss.

现在可以得出整个组件的简单结构：

components/

Deskmark(整个程序的框架)/

index.jsx

style.scss

CreateBar(新建按钮)/

List（左侧文章列表）/

ListItem(左侧列表中的每个条目)/

ItemEditor(右侧文章编辑器，包含保存和取消两个按钮)/

ItemShouwLayer(右侧文章展示，包含编辑和删除两个按钮)/

### 4.1.2 创建每个静态组件

当一个组件不需要内部的state、不需要组件的生命周期这些方法的时候，可以简写成一个纯函数。这种简便的写法，可以减少代码量。

从功能层面审视所有的组件，不难发现有一些组件只是传入属性、展示属性的值或者值或者对外输出方法。ListItem就是典型的例子，它什么都不关心只是接收一个属性，展示一条文章列表，在这个版本中，不会考虑任何的交互，组件只是静态静态地展示数据就可以了，那么很自然地，第一个无状态组件就出现了。