02

OPEN ORIENTED

凹 凸 实 验 室

ES6(三)

暖暖

目录

02

目录

02

Promise对象

Promise对象

02

简介：Promise对象是一个构造函数，用来生成Promise实例。

作用：将异步操作以同步操作的流程表达出来，避免层层嵌套的回调函数

Promise对象

02

基本用法

new Promise(function(参数1，参数2){
})

参数1：resolve
参数2：reject

```
var promise = new Promise(function(resolve, reject) {
  // ... some code

  if (/* 异步操作成功 */){
    // 从Pending变为Resolved，将value作为参数传递出去
    resolve(value);
  } else {
    // 从Pending变为Rejected，将error作为参数传递出去
    reject(error);
  }
});
```

Promise对象

02

	Promise.prototype.then()	Promise.prototype.catch
作用	then方法分别指定Resolved状态和Reject状态的回调函数。	别名：Promise.prototype.then(null, rejection)。 用于指定发生错误时的回调函数。

栗子	// promise是一个Promise实例
	promise.then(function(value) {
	  // Promise对象的状态变为Resolved时调用
	}, function(error) {
	  // Promise对象的状态变为Reject时调用，可选的。
	});
		var promise = new Promise(function(resolve, reject) {
		  reject(new Error('test'));
		});
		promise.catch(function(error) {
		  console.log(error);
		});
		// Error: test

共同点	返回的是一个新的Promise实例，因此可采用链式promise.then(…).then(…)	

Promise对象

02

	Promise.all()	Promise.race()
作用	参数数组的每个成员（Promise实例），都变成fulfilled，或者其中有一个变为rejected，才会调用Promise.all方法后面的回调函数	参数数组的每个成员（Promise实例），都变成fulfilled，或者其中有一个变为rejected，才会调用Promise.all方法后面的回调函数

栗子	// promises数组的每个成员是Promise实例
	var promises = [2, 3, 5].map(function (id) {
	  return new Promise((resolve,reject) => { resolve(id)});
	});
	
	Promise.all(promises).then(function (resultArr) {
	  // 将打印 [2, 3, 5]
	  console.log(resultArr);
	}).catch(function(result){
	  // result为第一个被reject的实例的返回值
	});
		// promises数组的每个成员是Promise实例
		var promises = [2, 3, 5].map(function (id) {
		  return new Promise((resolve,reject) => { reject(id)});
		});
		
		Promise.race(promises).then(function (result) {
		  console.log(result);
		}).catch(function(result){
		  // 将打印catch: 2
		  console.log('catch: ' + result);
		});

	接受一个数组作为参数，或者具有Iterator接口的对象；数组或对象的每个成员都必须是Promise实例。用于将多个Promise实例，包装成一个新的Promise实例。	

Promise对象

02

	Promise.resolve()	Promise.reject()
作用	返回一个新的Promise实例，该实例的状态为resolved。	返回一个新的Promise实例，该实例的状态为rejected。
参数用法	* 参数是一个Promise实例，则返回这个实例。 * 参数是一个具有then方法的对象，将这个对象转为Promise对象，然后就立即执行thenable对象的then方法。 * 参数不是具有then方法的对象，或根本就不是对象，则返回一个新的Promise对象，状态为resolved或者rejected。 * 不带有任何参数，则直接返回一个resolved或者rejected状态的Promise对象。	

共同点	var p1 = Promise.resolve('foo')
	// 等价于
	// var p1 = new Promise(resolve => resolve('foo'))
	
	p1.then(function(result){console.log(result);});
	// 将打印： foo
		var p2 = Promise.reject('出错了');
		// 等同于
		// var p2 = new Promise((resolve, reject) => reject('出错了'))
		
		p2.catch(function(result){console.log(result);})
		// 将打印： 出错了

目录

02

Iterator和for...of循环

Iterator和for...of循环

02

作用：

1、为各种数据结构，提供一个统一的、简便的访问接口；

2、使得数据结构的成员能够按某种次序排列；

3、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

基本用法

默认的Iterator接口部署在数据结构的Symbol.iterator属性

每一次调用next方法，返回一个包含value和done两个属性的对象，done表示遍历是否结束。

```
let arr = ['a', 'b', 'c'];
let iter = arr[Symbol.iterator]();

iter.next() // { value: 'a', done: false }
iter.next() // { value: 'b', done: false }
iter.next() // { value: 'c', done: false }
iter.next() // { value: undefined, done: true }
```

for...of循环

02

	for…of循环	for...in
作用	直接获取对象的键值。	直接获得对象的键名。
遍历属性	只遍历具有数字索引的属性。	不仅遍历数字键名，还会遍历手动添加的其他键，甚至包括原型链上的键。
遍历顺序	严格按照某种顺序来遍历。如Set和Map结构遍历的顺序是按照各个成员被添加进数据结构的顺序。	某些情况下， for...in循环会以任意顺序遍历键名

栗子	let arr = [3, 5, 7];
	arr.foo = 'hello';
	
	for (let i of arr) {
	// 遍历器接口只返回具有数字索引的属性。
	console.log(i); // "3", "5", "7"
	}
		let arr = [3, 5, 7];
		arr.foo = 'hello';
		
		for (let i in arr) {
		// 数组的键名是数字，但是for...in循环是以字符串作为键名如“0”、“1”、“2”
		console.log(i); // "0", "1", "2", "foo"
		}

目录

02

Generator 函数

Generator 函数

基本用法

Generator函数是一个状态机，封装了多个内部状态。

调用Generator函数，总是返回一个遍历器对象。

```
function* helloWorldGenerator() {
  // yield控制着多个内部状态
  // yield表示函数暂停执行，
  // 下一次再从该位置继续向后执行
  yield 'hello';
  yield 'world';
  return 'ending';
}

var hw = helloWorldGenerator();

hw.next()
// { value: 'hello', done: false }

hw.next()
// { value: 'world', done: false }
```

Generator 函数

Generator.prototype.return()

return()可以返回给定的值，并且终结遍历Generator函数。
不提供参数，则返回值的value属性为undefined。

遇到finally，则运行后再结束。

```
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 }
```

Generator 函数

yield* 表达式

表明表达式返回的是一个遍历器对象

```
function* foo() {
  yield 'a';
  yield 'b';
}

function* bar() {
  yield 'x';
  // 在一个 Generator 函数里面执行另一个Generator 函数。
  yield* foo();
  yield 'y';
}

for (let v of bar()){
  console.log(v);
}
// x
// a
// b
// y
```

目录

02

Async函数

Async函数

02

基本用法

Generator函数的语法糖。
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。

但是：
* **async函数自带执行器**，直接与普通函数一样执行即可。
* async函数的await命令后面，可以是Promise对象和原始类型的值（此时相当于同步操作）。
*** 返回值是Promise，用then方法指定下一步的操作，比 Generator 函数的返回值是 Iterator 对象方便。**

Async函数

Async函数多种使用形式

```
// 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() {} };

// 箭头函数
const foo = async () => {};
```

```
// 获取股票报价的函数
async function getStockPriceByName(name) {
  // await表示等到触发的异步操作完成，
  // 再接着执行函数体内后面的语句
  var symbol = await getStockSymbol(name);
  var stockPrice = await getStockPrice(symbol);
  // return语句返回的值，会成为then方法回调函数的参数
  return stockPrice;
}

// 调用async函数，返回Promise对象，可使用then方法添加回调函数
getStockPriceByName('goog').then(function (result) {
  console.log(result);
});
```

Async函数

错误处理机制

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。

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
}

f()
.then(v => console.log(v))
.catch(e => console.log(e))
// 出错了
```

Async函数

错误处理机制

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。

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?
捕获错误：利用try catch 或者 await后面的Promise链式加上catch方法

```
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
```

Async函数

使用注意点

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，缩短程序的执行时间。

```
// 假设getFoo和getBar是asycn函数
let [foo, bar] = await Promise.all([getFoo(), getBar()]);
```

参考文献

http://es6.ruanyifeng.com/

02

OPEN ORIENTED

凹 凸 实 验 室

THANKS FOR LISTENING

王彩暖