**Javascript 中的神器——Promise**

**Promise in js**

回调函数真正的问题在于他剥夺了我们使用 return 和 throw 这些关键字的能力。而 Promise 很好地解决了这一切。

2015 年 6 月，[ECMAScript 6 的正式版](http://www.ecma-international.org/publications/standards/Ecma-262.htm) 终于发布了。

ECMAScript 是 JavaScript 语言的国际标准，JavaScript 是 ECMAScript 的实现。ES6 的目标，是使得 JavaScript 语言可以用来编写大型的复杂的应用程序，成为企业级开发语言。

**概念**

ES6 原生提供了 Promise 对象。

所谓 Promise，就是一个对象，用来传递异步操作的消息。它代表了某个未来才会知道结果的事件（通常是一个异步操作），并且这个事件提供统一的 API，可供进一步处理。

Promise 对象有以下两个特点。

（1）对象的状态不受外界影响。Promise 对象代表一个异步操作，有三种状态：Pending（进行中）、Resolved（已完成，又称 Fulfilled）和 Rejected（已失败）。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。这也是 Promise 这个名字的由来，它的英语意思就是「承诺」，表示其他手段无法改变。

（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 对象的状态改变，只有两种可能：从 Pending 变为 Resolved 和从 Pending 变为 Rejected。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果。就算改变已经发生了，你再对 Promise 对象添加回调函数，也会立即得到这个结果。这与事件（Event）完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。

有了 Promise 对象，就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。此外，Promise 对象提供统一的接口，使得控制异步操作更加容易。

Promise 也有一些缺点。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）。

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

if (/\* 异步操作成功 \*/){

resolve(value);

} else {

reject(error);

}

});

promise.then(function(value) {

// success

}, function(value) {

// failure

});

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 方法和 reject 方法。

如果异步操作成功，则用 resolve 方法将 Promise 对象的状态，从「未完成」变为「成功」（即从 pending 变为 resolved）；

如果异步操作失败，则用 reject 方法将 Promise 对象的状态，从「未完成」变为「失败」（即从 pending 变为 rejected）。

**基本的 api**

1. Promise.resolve()
2. Promise.reject()
3. Promise.prototype.then()
4. Promise.prototype.catch()
5. Promise.all() // 所有的完成

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

1. Promise.race() // 竞速，完成一个即可

**进阶**

promises 的奇妙在于给予我们以前的 return 与 throw，每个 Promise 都会提供一个 then() 函数，和一个 catch()，实际上是 then(null, ...) 函数，

somePromise().then(functoin(){

// do something

});

我们可以做三件事，

1. return 另一个 promise

2. return 一个同步的值 (或者 undefined)

3. throw 一个同步异常 ` throw new Eror('');`

**1. 封装同步与异步代码**

```

new Promise(function (resolve, reject) {

resolve(someValue);

});

```

写成

```

Promise.resolve(someValue);

```

**2. 捕获同步异常**

new Promise(function (resolve, reject) {

throw new Error('悲剧了，又出 bug 了');

}).catch(function(err){

console.log(err);

});

如果是同步代码，可以写成

Promise.reject(new Error("什么鬼"));

**3. 多个异常捕获，更加精准的捕获**

somePromise.then(function() {

return a.b.c.d();

}).catch(TypeError, function(e) {

//If a is defined, will end up here because

//it is a type error to reference property of undefined

}).catch(ReferenceError, function(e) {

//Will end up here if a wasn't defined at all

}).catch(function(e) {

//Generic catch-the rest, error wasn't TypeError nor

//ReferenceError

});

**4. 获取两个 Promise 的返回值**

1. .then 方式顺序调用

2. 设定更高层的作用域

3. spread

**5. finally**

任何情况下都会执行的，一般写在 catch 之后

**6. bind**

somethingAsync().bind({})

.spread(function (aValue, bValue) {

this.aValue = aValue;

this.bValue = bValue;

return somethingElseAsync(aValue, bValue);

})

.then(function (cValue) {

return this.aValue + this.bValue + cValue;

});

或者 你也可以这样

var scope = {};

somethingAsync()

.spread(function (aValue, bValue) {

scope.aValue = aValue;

scope.bValue = bValue;

return somethingElseAsync(aValue, bValue);

})

.then(function (cValue) {

return scope.aValue + scope.bValue + cValue;

});

然而，这有非常多的区别，

1. 你必须先声明，有浪费资源和内存泄露的风险
2. 不能用于放在一个表达式的上下文中
3. 效率更低

**7. all。非常用于于处理一个动态大小均匀的 Promise 列表**

**8. join。非常适用于处理多个分离的 Promise**

```

var join = Promise.join;

join(getPictures(), getComments(), getTweets(),

function(pictures, comments, tweets) {

console.log("in total: " + pictures.length + comments.length + tweets.length);

});

```

**9. props。处理一个 promise 的 map 集合。只有有一个失败，所有的执行都结束**

```

Promise.props({

pictures: getPictures(),

comments: getComments(),

tweets: getTweets()

}).then(function(result) {

console.log(result.tweets, result.pictures, result.comments);

});

```

**10. any 、some、race**

```

Promise.some([

ping("ns1.example.com"),

ping("ns2.example.com"),

ping("ns3.example.com"),

ping("ns4.example.com")

], 2).spread(function(first, second) {

console.log(first, second);

}).catch(AggregateError, function(err) {

err.forEach(function(e) {  
console.error(e.stack);  
});  
});;

```

有可能，失败的 promise 比较多，导致，Promsie 永远不会 fulfilled

**11. .map(Function mapper [, Object options])**

用于处理一个数组，或者 promise 数组，

Option: concurrency 并发现

map(..., {concurrency: 1});

以下为不限制并发数量，读书文件信息

var Promise = require("bluebird");

var join = Promise.join;

var fs = Promise.promisifyAll(require("fs"));

var concurrency = parseFloat(process.argv[2] || "Infinity");

var fileNames = ["file1.json", "file2.json"];

Promise.map(fileNames, function(fileName) {

return fs.readFileAsync(fileName)

.then(JSON.parse)

.catch(SyntaxError, function(e) {

e.fileName = fileName;

throw e;

})

}, {concurrency: concurrency}).then(function(parsedJSONs) {

console.log(parsedJSONs);

}).catch(SyntaxError, function(e) {

console.log("Invalid JSON in file " + e.fileName + ": " + e.message);

});

结果

$ sync && echo 3 > /proc/sys/vm/drop\_caches

$ node test.js 1

reading files 35ms

$ sync && echo 3 > /proc/sys/vm/drop\_caches

$ node test.js Infinity

reading files: 9ms

**11. .reduce(Function reducer [, dynamic initialValue]) -> Promise**

Promise.reduce(["file1.txt", "file2.txt", "file3.txt"], function(total, fileName) {

return fs.readFileAsync(fileName, "utf8").then(function(contents) {

return total + parseInt(contents, 10);

});

}, 0).then(function(total) {

//Total is 30

});

**12. Time**

1. .delay(int ms) -> Promise
2. .timeout(int ms [, String message]) -> Promise

**Promise 的实现**

1. [q](https://github.com/kriskowal/q)
2. [bluebird](https://github.com/petkaantonov/bluebird)
3. [co](https://github.com/tj/co)
4. [when](https://github.com/cujojs/when)

**ASYNC**

async 函数与 Promise、Generator 函数一样，是用来取代回调函数、解决异步操作的一种方法。它本质上是 Generator 函数的语法糖。async 函数并不属于 ES6，而是被列入了 ES7。