Exploring ES2016 and ES2017

中文译本

Dr. Axel Rauschmayer

ECMANAUTEN

翻译免责声明

本书的翻译**未经**作者本人授权，翻译结果仅用于学习交流目的，不得用于任何商业目的。 以任何形式传播、 复制该翻译结果导致的后果，本人不承担任何责任。

DLR. 2017. 07. 29

I. 背景

第1章． TC39 关于 ECMAScritp 新特性的接纳过程

TC39:ECMAScript新特性的接纳过程

本章解释了所谓的TC39过程，它以ECMAScript 2016（ES7）为起点，管理ECMAScript新特性的设计。

1.1、谁设计ECMAScript？

TC39是开发JavaScript的委员会。其成员是由各个公司（其中包括所有主要浏览器厂商）组成的。TC39会定期开会，会议由成员代表和受邀专家出席。会议纪要可以在网上找到，可以让您更加详细地了解TC39的工作机制。

偶尔（甚至在这本书中），你会看到TC39成员指的是一个人。那么这意味着：它是TC39成员公司派出的代表。

有趣的是，TC39以协商一致方式运作：决议需要大多数人同意并且没有人强烈反对。对于许多成员来说，协议意味着真正的责任（他们必须实现协议中的特性）。

1.2、ECMAScript是如何设计的？

1.2.1问题：ECMAScript 2015（ES6）的版本太大

最新版本的ECMAScript ES6很大，在ES5（2009年12月和2015年6月）之后差不多6年才被标准化。有两个主要问题导致它们间隔这么久才发布：

1. 在新版本发布之前就就绪的新特性必须等到新版本发布之后才可以使用
2. 需要耗费较长时间实现的新特性将会感受到压力。因为如果将其推迟到下一个版本发布，那意味着需要等很久很久。

因此，从ECMAScript 2016（ES7）开始，新版本发布将更频繁地发生，因此也会更小。每年将有一个新版本，它将包含所有在截止日期之前完成的新特性。

1.2.2解决方案：TC39过程

ECMAScript新特性的每一个提案都将从第0阶段开始经过以下几个阶段达到成熟阶。从一个阶段到下一个阶段的进展必须得到TC39的批准。

1.2.2.1 Stage 0: 原型

它是什么？是一种以自由形式提交关于ECMAScript演变的想法。提交的材料必须来自TC39成员或注册为TC39贡献者的非成员。

需要些什么？该文件必须在TC39会议进行审查，然后将其添加到第0阶段提案的页面中。

1.2.2.2 Stage 1：提案

它是什么？它是该特性的正式提案。提案解决的问题必须在文中加以说明。必须通过示例，API和语义和算法的讨论来描述该解决方案。 最后，必须确定提案的潜在障碍，比如与其他特性的相互作用和实施挑战。需要采用实施方式，进行多重展示和演示。

下一步是什么？ TC39通过接受第一阶段的建议，声明会对该提案进行审议，讨论和贡献。接下来，预计将对该提案进行重大修改。

1.2.2.3 Stage 2：草案

它是什么？它是将会成为标准的该特性的第一个版本。到这一阶段，说明该特性可能会包含在最终的标准中。

需要做什么？

该提议现在还必须具有对该特征的语法和语义的正式描述（使用ECMAScript规范的形式语言）。描述应尽可能完整，但可以包含todos和占位符。需要两个实现该功能的试验，但是其中一个可以在诸如Babel的转换器中做。

下一步是什么？ 下一步还需要对其进行改进。

1.2.2.4 Stage 3：候选者

它是什么？它表明该草案的大部分内容都已经完成了，现在需要来自实施方和用户的反馈来进一步改进。

需要什么？该规范的书面文字必须完成。指定的评审员（由TC39指定）和ECMAScript规范编辑者必须在规范文本上签名。必须至少有两个规范兼容的实现（默认情况下不必启用）。

下一步是什么？ 从此以后，只有在实施及其使用过程中遇到重大问题时，才需要做出改变。

1.2.2.5 Stage 4：完成

它是什么？ 该提案已准备好纳入标准。

需要什么？ 在提案达到这个阶段之前，需要做以下事情：

1. 测试262验收测试（大体而言是语言特征的单元测试，用JavaScript编写）。
2. 两个通过测试的符合规范的实现。
3. 具有实践经验。
4. ECMAScript编辑必须在规范文本上签名。 下一步是什么？ 该提案将尽快包含在ECMAScript规范中。在每年发布新版标准的时候，这个提案就被批准为一部分。

1.3、不要称它们为ECMAScript 20xx功能

您可以看到，只有当提案达到第4阶段时，才能确定标准中将包含该功能。然后将其包含在下一个ECMAScript版本中是可能的，但不能100％确定（可能需要更长时间）。 因此，您无法再将它们成为“ES7功能”或“ES2016功能”。 因此，我写文章和博客时最喜欢采用的两种方式是：

- “ECMAScript提案：foo功能”。在文章开头指出该提案的阶段。
- “ES.stage2：foo功能” 如果一个提案处于第4阶段，我可以称它是一个ES20xx功能，但最安全的是等到规范编辑确认它将被包含在那个版本中。Object.observe是一个ECMAScript提案的例子， 它进展到了第二阶段，但最终被撤回。

1.4、进一步阅读

下面是本章节内容的主要来源：

- The ecma262 (ECMA-262 is the ID of the ECMAScript standard) GitHub repository, 它主要包含：
- A readme file with all proposals at stage 1 or higher
- A list of stage 0 proposals
- ECMA-262 frequently asked questions

- The TC39 process document

其它资料：

- “Kangax’ ES7 compatibility table” 显示了现在有哪提案，都到了哪一阶段
- 更多关于ES6是如何设计的请参考“How ECMAScript 6 was designed”

第2章. ES2016 和 ES2017 的常见问题

ECMAScript2016和ECMAScript2017常见问题

一、ECMAScript2016是不是内容太少了？

ECMAScript2016这么小正好说明了新的发布过程（正如上一章所讲）起到了作用：

1. 新特性只有在完全准备就绪并且在至少有两个经过充分现场测试的实现之后才被包含进来。
2. 发布频率会更高（每年一次）并且将会更加渐进式增加

ES2016让所有人（TC39,引擎开发者以及JS开发者）都可以喘一口气，它是发布体积庞大的ES6版本之后的一次小小的休息。

II. ECMAScript 2016

第3章. Array.prototype.includes

ECMAScript2016之includes方法

一、概述

本章将讲述由Domenic Denicola和Rick Waldron提出的ECMAScript2016新特性——Array.prototype.includes.

二、Array.prototype.includes

2.1、基本用法

```
> ['a','b','c'].includes('a')
< true
> ['a','b','c'].includes('d')
< false
```

2.2、includes方法详解

数值的includes方法的签名如下：

```
Array.prototype.includes(value: any) : boolean
```

意思是说，参数值是任意类型，返回值是布尔型。

如果数组中存在includes传入的元素，该方法返回true,如果不存在则返回false.

includes和indexOf很相似，下面这两种用法**几乎是**等价的:

```
arr.includes(x)
arr.indexOf(x) > 0
```

这里为什么说几乎呢？主要是因为includes可以找到NaN,而indexOf做不到。如下：

```
> [NaN].includes(NaN)
< true
> [NaN].indexOf(NaN)
< -1
```

includes不区分+0和-0，这是几乎所有JS的工作方式相同。另外Typed Array也有includes方法：

```
> [+0].includes(-0)
< true
> let arr = Uint8Array.of(1,2,3);
< undefined
> arr.includes(3)
< true
```

2.3、常见问题

2.3.1、为什么这个方法叫做includes而不是contains?

本来最开始是打算叫做contains的，但是由于网上的一些其它工具(MooTools)在Array.prototype上实现了contains，所以为了不破坏代码，改为叫做includes.

2.3.2、为什么这个方法叫includes而不是has?

has通常用于键（Map.prototype.has）,includes通常用于元素（String.prototype.includes）.由于Set的元素既可以被视为键也可以被视为值，所以它有一个has方法，但是要注意，它并没有includes方法。

第4章. 幂操作符（**）

ECMAScript2016之幂操作符

一、概述

本章将讲述由由Rick Waldron提出的ECMAScript2016新特性——幂操作符

二、幂操作符

幂操作符（**）是一个中缀表达式，基本用法如下：

```
x**y
```

它其实和Math.pow(x,y)的结果是一样的。例子如下：

```
let squared = 3 ** 2; // 9

let num = 3;
num **= 2;
console.log(num); // 9
```

III. ECMAScript 2017

第5章. 异步函数（Async function）

ECMAScript2017之异步函数（async function）

本章将讲述由Brian Terlson提出的ECMAScript2017新特性——异步函数（async function）。

一、概述

1.1、变体

存在下面几种异步函数（async function）的变体，注意`async`关键字无处不在：

1. 异步函数声明：`async function foo () {}`
2. 异步函数表达式：`const foo = async function () {}`
3. 异步方法定义：`let obj = { async foo () {} }`
4. 异步箭头函数：`const foo = async () => {}`

1.2、异步函数总是返回promise

通过return来fullfill异步函数的promise：

```
> async function foo () {
    return 123
  }
  foo().then(x => {
    console.log(x)
  })
  123
  ◂ ▼ Promise {[[PromiseStatus]]: "resolved", [[PromiseValue]]:
      ▶ __proto__: Promise
        [[PromiseStatus]]: "resolved"
        [[PromiseValue]]: undefined
```

通过throw来reject异步函数的promise:

```
> async function foo () {
    throw new Error('错误')
  }
  foo().then(x => {
    console.log(x)
  })
  ◂ ▼ Promise {[[PromiseStatus]]: "rejected", [[PromiseValue]]: Error: 错误 at foo
      ▶ __proto__: Promise
        [[PromiseStatus]]: "rejected"
      ▶ [[PromiseValue]]: Error: 错误 at foo (<anonymous>:2:9) at <anonymous>:4:1
  ⊗ ▶Uncaught (in promise) Error: 错误
        at foo (<anonymous>:2:9)
        at <anonymous>:4:1
```

1.3、通过await处理异步函数返回的结果和错误

操作符await只能用在异步函数里面，它等待它的操作数（某promise）被设置：

- 如果该promise被fullfilled，那么await得到的是fullfilled的结果
- 如果该promise被rejected，那么await抛出rejected的结果

处理单个异步函数的返回结果

```
async function bar () {
  return 123
}
async function foo () {
  const result = await bar()
  console.log(result)
  return result
}
foo().then(x => {
  console.log(x)
})
```

```
123
123
▼ Promise {[[PromiseStatus]]: "resolved", [
  ▶ __proto__: Promise
    [[PromiseStatus]]: "resolved"
    [[PromiseValue]]: undefined
```

依次处理多个异步函数的返回结果

```
async function bar1 () {
  return 123
}
async function bar2 () {
  return 456
}
async function foo () {
  const result1 = await bar1()
  console.log(result1)
  const result2 = await bar2()
  console.log(result2)
}
foo()
```

```
123
456
▶ Promise {[[PromiseStatus]]: "resolved", [[PromiseValue]]: undefined}
```

上述代码等效于下面：

```
async function bar1 () {
  return 123
}
async function bar2 () {
  return 456
}
function foo () {
  return bar1()
  .then( result1 => {
        console.log(result1)
        return bar2()
  })
  .then( result2 => {
        console.log(result2)
  })
}
foo()
```

```
123
456
▶ Promise {[[PromiseStatus]]: "resolved", [[PromiseValue]]: undefined}
```

并行处理多个异步函数的返回结果

```
async function bar1 () {
  return 123
}
async function bar2 () {
  return 456
}
async function foo () {
  const [result1, result2] = await Promise.all([
      bar1(),
        bar2()
  ])
  console.log(result1, result2)
}
foo()
```

```
123 456
▼ Promise {[[PromiseStatus]]: "resolved", [[PromiseValue]]: undefined}
  ▶ __proto__: Promise
    [[PromiseStatus]]: "resolved"
    [[PromiseValue]]: undefined
```

上述代码等价于：

```
async function bar1 () {
  return 123
}
async function bar2 () {
  return 456
}
function foo () {
  return Promise.all([
    bar1(),
    bar2()
  ]).then(([result1, result2]) => {
    console.log(result1, result2)
  })
}
foo()
```

```
123 456
▶ Promise {[[PromiseStatus]]: "resolved", [[PromiseValue]]: undefined}
```

处理错误

```
async function bar () {
  throw new Error('错误')
}
async function foo () {
  try{
    await bar()
    }catch(err){
    console.log(err)
  }
}
foo()
```

```
Error: 错误
    at bar (<anonymous>:2:9)
    at foo (<anonymous>:6:11)
    at <anonymous>:11:1
▶ Promise {[[PromiseStatus]]: "resolved", [[PromiseValue]]:
```

上述代码等价于：

```
async function bar () {
  throw new Error('错误')
}
function foo () {
  return bar().catch(err => {
    console.log(err)
  })
}
foo()
```

```
Error: 错误
    at bar (<anonymous>:2:9)
    at foo (<anonymous>:5:10)
    at <anonymous>:9:1
```

```
▶ Promise {[[PromiseStatus]]: "resolved", [[PromiseValue]]: undefined}
```

二、理解异步函数

在我解释异步函数（async function）之前，我需要先解释一下如何将Promise和Generator结合来以同步式的代码书写形式实现异步操作。

对于那些需要得到一次性异步计算结果的函数，ES6提供的Promise大受欢迎。一个典型的例子是客户端的Fetch API。它是除XMLHttpRequest方法之外的另外一种获取资源的方式。它的基本用法如下：

```
function fetchJson(url) {
    return fetch(url)
    .then(request => request.text())
    .then(text => {
        return JSON.parse(text);
    })
    .catch(error => {
        console.log(`ERROR: ${error.stack}`);
    });
}
fetchJson('http://example.com/some_file.json')
.then(obj => console.log(obj));
```

2.1、通过Generator书写异步代码

co是一个使用Promise和Genrator让编程者可以使用同步式的代码风格来实现异步代码的库，它的基本使用例子如下：

```
const fetchJson = co.wrap(function* (url) {
    try {
        let request = yield fetch(url);
        let text = yield request.text();
        return JSON.parse(text);
    }
    catch (error) {
        console.log(`ERROR: ${error.stack}`);
    }
});
```

每当回调函数（实示例中的Generator函数）产生一个Promise给co时，回调函数都被暂停。一旦Promise被设定，co将恢复回调函数。如果Promise被fullfilled，yeild输出fillfilled的值，如果Promise被rejected,yeild将抛出rejected返回的错误。除此之外，co将回调函数返回的结果promise化（类似于then所做的事情）：

2.2、通过异步函数书写异步代码

异步函数（async function）基本上就是为co所做事情量身打造的语法：

```
async function fetchJson(url) {
    try {
        let request = await fetch(url);
        let text = await request.text();
        return JSON.parse(text);
    }
    catch (error) {
        console.log(`ERROR: ${error.stack}`);
    }
}
```

异步函数的内部实现和Generator很像。

2.3、异步函数同步启动，异步设定

下面是异步函数是如何被执行的：

1. 异步函数的返回结果总是Promise.这个Promise在开始执行异步函数的时候就被创建。
2. 函数体被执行。执行可以被return 或 throw永久性的结束。或者被await暂时性的结束。在后面这种情况下，函数体的执行可以在随后继续。
3. 该promise被返回。

当执行函数体的时候，return x将会用x值resolve该promise,而throw err将会用err值reject该promise。“异步函数被设定”这个通知是异步的。换句话说，只有在then和catch都只有在当前代码执行完成后才会执行（也就是被异步通知）。

下面的代码解释了上面所讲的,注意输出顺序：

```
async function asyncFunc() {
    console.log('asyncFunc()'); // (A)
    return 'abc';
}
asyncFunc().
then(x => console.log(`Resolved: ${x}`)); // (B)
console.log('main'); // (C)
```

```
asyncFunc()
main
Resolved: abc
```

下面讲解一下输出顺序：

1. A行。异步函数同步启动。它的promise是通过return给resolved的
2. C行。执行继续
3. B行。“promise被resolved”这个消息是异步的，只有收到了这个消息then才会执行

2.4、返回的promise没有被包裹

resolved一个promise是一个标准的操作，通过return来resolve一个promise(比如叫p),这意味着：

1. 直接return一个非promise值的话将会使用该值resolve该promise(即p)
2. 如果return一个promise的话意味着p将拷贝那个promise的状态，也就是说不会将这个返回的promise作为值resolve p，而是会拿它的状态给自己使用。

因此，你可以返回一个promise,这个promise不会被另一个promise包裹,下面两个结果是一样的：

```
async function asyncFunc() {
    return Promise.resolve(123);
}
asyncFunc()
.then(x => console.log(x)) // 123
```

```
123
```

```
▼ Promise {[[PromiseStatus]]: "resolved", [[PromiseV
  ▶ __proto__: Promise
    [[PromiseStatus]]: "resolved"
    [[PromiseValue]]: undefined
```

```
async function asyncFunc() {
    return 123;
}
asyncFunc()
.then(x => console.log(x)) // 123
```

```
123
```

```
▼ Promise {[[PromiseStatus]]: "resolved", [[
  ▶ __proto__: Promise
    [[PromiseStatus]]: "resolved"
    [[PromiseValue]]: undefined
```

有趣的是，返回一个rejected的promise将导致异步函数被rejected(正常情况下，我们是用throw来rejected外面的异步函数)：

```
async function asyncFunc() {
    return Promise.reject(new Error('Problem!'));
}
asyncFunc()
.catch(err => console.error(err)); // Error: Problem!
```

```
▶ Error: Problem!
    at asyncFunc (<anonymous>:2:27)
    at <anonymous>:4:1
```

```
▶ Promise {[[PromiseStatus]]: "resolved", [[PromiseValue]]:
```

```
async function asyncFunc() {
    throw new Error('Problem!')
}
asyncFunc()
.catch(err => console.error(err)); // Error: Problem!
```

```
▶ Error: Problem!
    at asyncFunc (<anonymous>:2:11)
    at <anonymous>:4:1
```

```
▶ Promise {[[PromiseStatus]]: "resolved", [[PromiseValue]]:
```

这与Promise解决方案的工作机制是一致的：它允许你不使用await转发另一个异步计算的结果，不论它是fullfilled还是rejected.

```
async function asyncFunc() {
    return anotherAsyncFunc();
```

```
}
```

上面的代码和下面的基本上一样，但是比下面代码更有效率，因为下面的代码使用await将promise unrapped(提取成功的值或失败的error)然后又用return将提取的值warp成promise了。

```
async function asyncFunc() {
    return await anotherAsyncFunc();
}
```

三、使用await的技巧

3.1、不要忘记使用await

一个非常容易犯的错误是在异步函数里调用另外一个异步函数的时候忘记使用await：

```
async function asyncFunc() {
    const value = otherAsyncFunc(); // missing `await`!
    ...
}
```

在这个例子里面value将会是一个promise,这通常不是你想要的结果。await在异步函数没有返回值的时候也有重要的意义。它的promise被用作信号告诉调用者里面的异步函数已经结束了。比如：

```
async function foo() {
    await step1(); // (A)
    ...
}
```

(A)处的await可以确保只有当step1执行结束之后，foo才能得到该通知。

3.2、当你只是想启动异步函数，但是不关心它啥时候结束的时候，可以不使用await

有些时候，我们只是想触发异步函数，但是不关心它啥时候结束，下面是一个例子：

```
async function asyncFunc() {
    const writer = openFile('someFile.txt');
    writer.write('hello'); // don’t wait
    writer.write('world'); // don’t wait
    await writer.close(); // wait for file to close
}
```

这里我们并不关心单个的write啥时候结束，它们只需要保证按顺序执行就可以了。最后一行的await确保只有当文件被成功关闭之后asyncFunc函数才被fullfilled。

考虑到返回的promise不会被promise化，你也可以直接返回writer.close()：

```
async function asyncFunc() {
    const writer = openFile('someFile.txt');
    writer.write('hello');
    writer.write('world');
```

```
    return writer.close();
}
```

两种版本都有各自的支持者和反对者。前面一种await版本可能更好理解。

3.3、await是串行的，Promise.all是并行的

下面这个例子进行了两个异步函数调用，asyncFunc1() 和 asyncFunc2().

```
async function foo() {
    const result1 = await asyncFunc1();
    const result2 = await asyncFunc2();
}
```

但是它俩是串行的，如果并行的执行它们可以加快执行速度，你可以使用Promise.all做到：

```
async function foo() {
    const [result1, result2] = await Promise.all([
        asyncFunc1(),
        asyncFunc2(),
    ]);
}
```

我们现在不需要await两个promise,只需要await有两个元素的Promise数组。

四、异步函数和回调

异步函数的一个限制是它只会影响它直接外围的异步函数。因此，异步函数不能再回调函数中await，然而回调函数本身可以是一个异步函数，这个我们稍后会看到。这使得基于回调的实体函数和方法难以使用。这方面的例子包括数组方法map和forEach

4.1、Array.prototype.map()

让我们先看看Array.prototype.map()。在下面的例子中，我们想下载数组中的URL链接的文件并且以数组的形式返回结果。

```
async function downloadContent(urls) {
    return urls.map(url => {
        // Wrong syntax!
        const content = await httpGet(url);
        return content;
    });
}
```

上面会存在语法错误，因为在普通的函数（包括普通箭头函数）中是不能够使用await的，只有在异步函数（async function）中才可以使用await。但是如果我们使用异步箭头函数呢？

```
async function downloadContent(urls) {
    return urls.map(async (url) => {
        const content = await httpGet(url);
        return content;
```

```
    });
}
```

语法上是没错误了，但是上述代码存在两个问题：

1. 返回结果将是promise组成的数组，而不是普通的数值或字符串等等
2. 当map结束时，map里的回调函数所做的工作并没有结束。因为await只是暂停它直接外围的异步箭头函数并且httpGet(url)的resolved是异步的。这意味着直到downloadContent完成你才能使用await去等待。

我们可以通过Promise.all解决上述两个问题。用Promise.all将两个Promise组成的数组转换成一个promise，这个promise包含两个的fullfilled的值：

```
async function downloadContent(urls) {
    const promiseArray = urls.map(async (url) => {
        const content = await httpGet(url);
        return content;
    });
    return await Promise.all(promiseArray);
}
```

map的回调函数并不对httpGet做太多事，它只是传递它。因此，这里我们没有必要再用异步箭头函数，普通箭头函数就OK了：

```
async function downloadContent(urls) {
    const promiseArray = urls.map(
        url => httpGet(url));
    return await Promise.all(promiseArray);
}
```

现在我们还可以做小小的改进。这个异步函数有点低调，它先用await将Promise.all返回的promise unwrap, 然后又将unwrap后的结果用return wrap为promise。也就是说先反promise化，又promise化。考虑到return并不会将promise再promise化，我们可以不用await，直接return。

```
async function downloadContent(urls) {
    const promiseArray = urls.map(
        url => httpGet(url));
    return Promise.all(promiseArray);
}
```

4.2、 Array.prototype.forEach()

让我们用forEach来输出多个URL指向的文件内容：

```
async function logContent(urls) {
    urls.forEach(url => {
        // Wrong syntax
        const content = await httpGet(url);
        console.log(content);
    });
}
```

同样，这段代码将报语法错误，因为你不能在普通箭头函数中使用await.让我们使用异步箭头函数：

```
async function logContent(urls) {
    urls.forEach(async url => {
        const content = await httpGet(url);
        console.log(content);
    });
    // Not finished here
}
```

这个确实能工作，但是有个警告：httpGet返回的peomise将会被异步的resolved，这意味着当forEach返回时，回调还没有结束。因此，你不能在logContent末尾await。

如果上面的结果不是你想要的，你可以改用for...of：

```
async function logContent(urls) {
    for (const url of urls) {
        const content = await httpGet(url);
        console.log(content);
    }
}
```

现在，当for...of结束时，所有的promise都resolved了。但是，这会导致httpGet被串行地调用。当第一个httpGet调用结束后，才会进行第二个调用。如果你想他们并行的调用，你必须使用Promise.all：

```
async function logContent(urls) {
    await Promise.all(urls.map(
        async url => {
            const content = await httpGet(url);
            console.log(content);
        }));
}
```

map()用来创建由promise组成的数组，我们并不关心他们fullfilled的结果，我们只等待所有promise都被fullfilled了。这意味着在logContent函数末尾他们都已经结束了。我们也可以只返回Promise.all()，但此时函数的结果将是undefined组成的数组。

测试一下：

```
> async function asyncFunc() {
      throw new Error('Problem!')
  }
  asyncFunc()
  .catch(err => console.error(err)); // Error: Problem!
▶ Error: Problem!
      at asyncFunc (<anonymous>:2:11)
      at <anonymous>:4:1
◂ ▶ Promise {[[PromiseStatus]]: "resolved", [[PromiseValue]]:
```

return的话返回undefined组成的数组：

```
async function httpGet (url) {
    return 1
}
async function logContent(urls) {
        return Promise.all(urls.map(
            async url => {
                const content = await httpGet(url);
                console.log(content);
        }));
}
logContent(['url1','url2','url3']).then(x => {console.log(x)})
```

```
3 1
▶ (3) [undefined, undefined, undefined]
▶ Promise {[[PromiseStatus]]: "resolved", [[PromiseValue]]: undefined}
```

五、异步函数使用技巧

5.1、理解你的异步函数

异步函数的基础是Promise，这就是为什么理解Promise对于理解异步函数具有至关重要的作用。尤其是当需要连接一些不是基于Promise的异步函数的老代码的时候，你常常别无选择只能直接使用Promise.

比如说，下面是一个基于Promised版本的XMLHttpRequest：

```
function httpGet(url, responseType="") {
    return new Promise(
        function (resolve, reject) {
            const request = new XMLHttpRequest();
            request.onload = function () {
                if (this.status === 200) {
                    // Success
                    resolve(this.response);
                } else {
                    // Something went wrong (404 etc.)
                    reject(new Error(this.statusText));
                }
            };
            request.onerror = function () {
                reject(new Error(
                    'XMLHttpRequest Error: '+this.statusText));
            };
            request.open('GET', url);
            xhr.responseType = responseType;
            request.send();
        });
}
```

XMLHttpRequest的API是基于回调的，通过异步函数来promise化它意味着你需要fullfilled或者rejected我们在回调函数中返回的Promise，那是不可能的，因为在异步函数中你只能使用return和throw来fullfilled和resolve一

个promise。但是你又没法在回调函数中返回一个函数的结果，throw也有类似的限制。

因此，异步函数普遍的编程风格是这样的：

- 直接使用Promises构建异步原语。
- 通过异步函数使用这些原语。

5.2、立即调用异步箭头函数表达式

有时候，如果你可以在模块或者脚本的开始使用await，那是挺好的。可惜的是，await只能在异步函数中使用，所以你可以有几个选择。你可以创建一个异步函数main，然后立刻调用它：

```
async function main() {
    console.log(await asyncFunction());
}
main();
```

或者是你可以使用立即调用异步箭头函数表达式：

```
(async function () {
    console.log(await asyncFunction());
})();
```

另一种是立即调用异步箭头函数：

```
(async () => {
    console.log(await asyncFunction());
})();
```

5.3、异步函数的单元测试

下面的代码使用mocha测试框架来对异步函数 asyncFunc1() 和 asyncFunc2()进行单元测试:

```
import assert from 'assert';

// Bug: the following test always succeeds
test('Testing async code', function () {
    asyncFunc1() // (A)
    .then(result1 => {
        assert.strictEqual(result1, 'a'); // (B)
        return asyncFunc2();
    })
    .then(result2 => {
        assert.strictEqual(result2, 'b'); // (C)
    });
});
```

然而上面的测试永远会成功，因为mocha不会等待（B）和（C）处的断言语句被执行。

你可以通过返回promise链的结果来修正这个bug，因为mocha可以知道，当某个测试返回的是promise的时候，mocha将会等待那个promise被设定（除非超时了）。

```
return asyncFunc1() // (A)
```

因为异步函数总是返回promise，这使得它们对这种单元测试表现完美：

```
import assert from 'assert';
test('Testing async code', async function () {
    const result1 = await asyncFunc1();
    assert.strictEqual(result1, 'a');
    const result2 = await asyncFunc2();
    assert.strictEqual(result2, 'b');
});
```

现在，对于单元测试而言，异步函数有两大好处，一个是代码更加简练，另一个是返回的promise可以被正确处理。

5.4、无需担心未处理的rejected

JavaScript引擎越来越善于对未被处理的rejected发出警告。 例如，以下代码通常会在以前默默地失败，但大多数现代JavaScript引擎现在都报告了一个未处理的rejected：

```
async function foo() {
    throw new Error('Problem!');
}
foo();
```

第6章. 共享内存和原子操作

ECMAScript2017之共享内存和原子

本章将讲述由Lars T. Hansen提出的ECMAScript 2017的新功能——共享内存和原子。它引入了一个新的构造函数SharedArrayBuffer和具有辅助函数的命名空间对象Atomics。本章将介绍其细节。

一、并行性（parallelism） vs 并发性（concurrency）

在我们开始之前，让我们来澄清两个相似但不同的术语："并行"和"并发"。存在许多对于它俩的定义，我像下面这样使用它们：

- 并行（并行 vs 串行）：同时执行多个任务
- 并发（并发 vs 顺序）：在重复的时间段（而不是一个接一个）执行几个任务。

两者密切相关，但不一样：

- 并行但不并发：单指令，多数据（SIMD）。多次计算并行发生，但在任何给定的时刻只执行一个任务（指令）。
- 并发但不并行：通过单核CPU上的时间分配进行多任务处理。

但是，很难准确地使用这些术语，这就是为什么互换它们通常也没有什么大问题。

1.1、并行模式

有两种并行模式：

- 数据并行性：同一段代码并行执行多次。实例对同一数据集的不同元素进行操作。例如：MapReduce是一种数据并行编程模型。
- 任务并行性：并行执行不同的代码段。示例：web workers(web多线程)和Unix子进程模型

二、JavaScript并行的发展历史

（1）JavaScript在单线程中执行。某些任务可以异步执行：浏览器通常会在单独的线程中运行这些任务，然后通过回调将其结果返回到单个线程中。

（2）web Workers给JavaScript带来了多线程，它们是重量级的进程。每一个worker都有它自己的全局环境，默认情况下，任何东西都不共享。worker与worker（或子worker与父worker）之间的通信不断演进：

- 起初，你只能发送和接收字符串。
- 然后，引入了结构化克隆：可以发送和接收数据副本。结构化克隆适用于大多数数据（JSON数据，类型数组，正则表达式，Blob对象，ImageData对象等）。它甚至可以正确处理对象之间的循环引用。但是，不能克隆error对象，function对象和DOM节点。
- workers之间传递数据。当接收方收到数据后，发送方失去其访问权限。

（3）在GPU（非常适合并行计算）上做WebGL计算：这是一个hack，它像下面这样工作：

- 输入：您的数据被转换为图像（逐个像素）。
- 处理：OpenGL像素着色器可以在GPU上执行任意计算。您的像素着色器会转换输入图像。
- 输出：图像。你可以将它转换成其它数据类型

（4）SIMD（低级数据并行性）：通过ECMAScript提案SIMD.js支持。它允许您同时对多个整数或浮点执行操作（如加法和平方根操作）。

（4）PJS（代号River Trail）：这个最终被遗弃的项目的计划是将高级别的数据并行（想象一下map-reduce）转换为JavaScript。但是，开发人员和引擎执行者的兴趣还不够。没有实现，开发人员就不能尝试这个API，因为它不能被polyfill。在2015-01-05，Lars T. Hansen宣布将从Firefox中删除实验性实施。

2.1、下一步：SharedArrayBuffer

下一步是什么？对于低级并行性，方向相当明确：尽可能地支持SIMD和GPU。然而，对于高层次的并行性，方向并不明朗，尤其是在PJS失败之后。

需要尝试许多不同的实时方法，以了解如何最好地将高级别并行性提供给JavaScript。遵循可扩展网络宣言的原则，“共享内存和原子”（也叫作“Shared Array Buffers”）的提案应运而生。它提供可用于实现更高级别构造的低级原语。

三、Shared Array Buffers

Shared Array Buffers是用于实现更高级别并发的原始构建块。它允许你在工作者线程和主线程之间共享SharedArrayBuffer，这种共享有两个好处：

- 你可以更快地在工作者线程之间共享数据
- 工作者线程之间的协作将会更简单、更快速（相较postMessage）

3.1、创建和发送一个Shared Array Buffers

```
// main.js

const worker = new Worker('worker.js');

// To be shared
const sharedBuffer = new SharedArrayBuffer( // (A)
    10 * Int32Array.BYTES_PER_ELEMENT); // 10 elements

// Share sharedBuffer with the worker
worker.postMessage({sharedBuffer}); // clone

// Local only
const sharedArray = new Int32Array(sharedBuffer); // (B)
```

创建一个Shared Array Buffers的方式与创建普通Array Buffer相同：通过调用构造函数并以字节（A）指定缓冲区的大小。与工作者线程共享的是缓冲区，对于自己本地使用的话，通常将Shared Array Buffers用Typed Arrays （B）包装起来。

警告：克隆共享阵列缓冲区是正确的共享方式，但是一些引擎仍然会实现旧版本的API，并要求您进行传输：

```
worker.postMessage({sharedBuffer}, [sharedBuffer]); // transfer (deprecated)
```

在API的最终版本中，传输Shared Array Buffers意味着您将失去对访它的访问权。

3.2、接收Shared Array Buffers

工作者线程的实现类似如下：

```
// worker.js

self.addEventListener('message', function (event) {
    const {sharedBuffer} = event.data;
    const sharedArray = new Int32Array(sharedBuffer); // (A)

    // ...
});
```

我们首先提取发送给我们的Shared Array Buffers，然后将其包装在Typed Arrays（行A）中，以便我们可以在本地使用它。

四、原子：安全的访问共享数据

4.1问题：优化使得代码在工作者线程之间无法预测

在单线程中，编译器的优化可能破坏多线程代码。

以下列代码为例：

```
while (sharedArray[0] === 123) ;
```

在单个线程中，当循环运行时，sharedArray [0]的值不会更改。因此，代码可以优化如下：

```
const tmp = sharedArray[0];
while (tmp === 123) ;
```

但是，在多线程中，此优化会阻止我们使用此模式来等待另一个线程所做的更改。

另一个例子是以下代码：

```
// main.js
sharedArray[1] = 11;
sharedArray[2] = 22;
```

在单线程中，您可以重新排列这些写入操作的顺序，因为在这两次写入操作之间没有读取它们内容的操作。对于多线程，只要您期望以特定顺序完成写入操作，就会遇到麻烦：

```
// worker.js
while (sharedArray[2] !== 22) ;
console.log(sharedArray[1]); // 0 or 11
```

这些优化使得实际上不可能同步多个工作者线程在同一个Shared Array Buffers上的行为。

4.2、解决方案：原子

该提案提供了全局变量Atomics，其方法有三个主要用例：

4.2.1、案例： 同步

Atomics可以用来与其他工作者线程进行同步。例如，以下两个操作可以让您读取和写入数据，并且不会被编译器重新排列：

- Atomics.load(ta : TypedArray, index) : T
- Atomics.store(ta : TypedArray, index, value : T) : T

这个想法是使用普通操作来读取和写入大部分数据，而Atomics操作（加载，存储等）用来确保读写安全。通常，您需要自定义同步机制，例如锁，其实现是基于Atomics的。

这是一个非常简单的例子，感谢Atomics（我省略了设置sharedArray）：

```
// main.js
console.log('notifying...');
Atomics.store(sharedArray, 0, 123);

// worker.js
while (Atomics.load(sharedArray, 0) !== 123) ;
console.log('notified');
```

4.2.2、用例：等待通知

使用while循环来等待通知不是很有效率，这就是为什么Atomics有操作帮助：

- Atomics.wait（ta：Int32Array，index，value，timeout）。在ta [index]处等待通知，但只有当ta [index]为值时才起作用。
- Atomics.wake（ta：Int32Array，index，count）。唤醒在ta [index]处等待的工作者线程。

4.2.3、用例：原子操作

有几个Atomics操作用于执行算术运算，并且在执行此操作时不能中断，这有助于同步。例如：

- Atomics.add（ta：TypedArray ，index，value）：T

大致来说，此操作执行：

```
ta [index] + = value;
```

4.2.3、问题：破坏值

共享内存的一个有问题是破坏的值（垃圾）：当读取时，您可能会看到一个中间值，该值既不是新值写入内存之前的值，也不是新值。

规范中的“无撕裂读取”表示，当且仅当以下情况下，没有撕裂：

- 读写都通过Typed Arrays (而不是 DataViews)进行。
- 两个Typed Arrays与它们的Shared Array Buffers对齐：sharedArray.byteOffset%sharedArray.BYTES*PER*ELEMENT === 0
- 两个Typed Arrays的每个元素的字节数相同。

换句话说，每当通过以下方式访问相同的Shared Array Buffers时，会出现破坏值的问题：

- 一个或多个DateView
- 一个或多个未对齐的Typed Arrays

- 具有不同元素大小的Typed Arrays

为避免这些情况下的破坏值，请使用Atomics或同步。

五、使用Shared Array Buffers

5.1、Shared Array Buffer和JavaScript的运行到完成（run-to-completion）语义

JavaScript具有所谓的运行到完成语义：每个函数都可以直到完成为止也不会被另一个线程中断。函数成为了事务，可以执行完整的算法，而任何人都不能看到他们操作的数据的中间状态。

Shared Array Buffer可以中断运行到完成（RTC）：函数正在处理的数据可以在函数运行时由另一个线程更改。但是，是否允许这样的事情发生完全由程序控制：如果不使用Shared Array Buffer，则是安全的。

这与异步功能如何违反RTC类似。在那里，您可以通过await关键字阻止函数操作。

5.2、Shared Array Buffer和asm.js 和 WebAssembly

共享阵列缓冲区使emscripten可以将pthreads编译成asm.js.引用emscripten文档页面说明emscripten文档页面说明：

> [Shared Array Buffer允许] Emscripten应用程序共享工作者线程之间的主内存堆。这与低级原子和futex支持的原语一起使Emscripten能够实现对Pthreads（POSIX线程）API的支持。

也就是说，您可以将多线程C和C ++代码编译为asm.js.

讨论如何最好地将多线程引入WebAssembly正在进行中。鉴于web workers相对而言偏重量级，WebAssembly可能会引入轻量级线程。您可以看到线程在WebAssembly的未来路线图上。

5.3、共享整型以外的数据

目前，只能使用整数（最多32位长）的数组。这意味着共享其他类型数据的唯一方式是将其编码为整数。可能有助于包括的工具：

- TextEncoder和TextDecoder：前者将字符串转换为Uint8Array的实例。后者则相反。
- stringview.js：将字符串作为字符数组处理的库。使用阵列缓冲区
- FlatJS：通过在扁平化内存（ArrayBuffer和SharedArrayBuffer）中存储复杂数据结构（结构体，类和数组）的方式来增强JavaScript。JavaScript + FlatJS被编译成纯JavaScript。支持JavaScript方言（TypeScript等）。
- TurboScript：是一种用于快速并行编程的JavaScript方言。它编译为asm.js和WebAssembly。

最终，可能会有更多的更高级别的共享数据机制。有一个实验将继续弄明白这些机制应该如何更合理。

5.4、使用Shared Array Buffer的代码多快？

Lars T. Hansen编写了Mandelbrot算法的两个实现（如他的文章“A Taste of JavaScript's New Parallel Primitives”中所述，您可以在线尝试）：串行版本和使用多个工作者线程的并行版本。 对于最多4个工作者线程（处理器核心个数），加速几乎线性地提高，从每秒6.9帧（1个工作者线程）到每秒25.4帧（4个工作者线程）。更多的工作者线程带来了更多的性能改进，但适中更好。

Hansen指出，加速令人印象深刻，但是并行代码的代价更为复杂。

六、示例

我们来看一个更全面的例子。它的代码在GitHub上，在 shared-array-buffer-demo中。 你可以在线运行它 repository shared-array-buffer-demo。

6.1使用共享锁

在主线程中，我们设置好共享内存以便它编码一个关闭锁并将其发送给一个工作者线程（A行）。 用户点击后，我们打开锁（B行）。

```
// main.js

// Set up the shared memory
const sharedBuffer = new SharedArrayBuffer(
    1 * Int32Array.BYTES_PER_ELEMENT);
const sharedArray = new Int32Array(sharedBuffer);

// Set up the lock
Lock.initialize(sharedArray, 0);
const lock = new Lock(sharedArray, 0);
lock.lock(); // writes to sharedBuffer

worker.postMessage({sharedBuffer}); // (A)

document.getElementById('unlock').addEventListener(
    'click', event => {
        event.preventDefault();
        lock.unlock(); // (B)
    });
```

在工作者线程中，我们设置了一个本地版本的锁（其状态通过Shared Array Buffer与主线程共享）。 在B行中，我们一直等待知道锁被解开。在行A和C中，我们发送文本到主线程，它显示在我们的页面上（在以前的代码片段中未显示它是如何做到的）。 也就是说，我们在这两行中使用的self.postMessage（ ）非常类似于console.log（ ）。

```
// worker.js

self.addEventListener('message', function (event) {
    const {sharedBuffer} = event.data;
    const lock = new Lock(new Int32Array(sharedBuffer), 0);

    self.postMessage('Waiting for lock...'); // (A)
    lock.lock(); // (B) blocks!
    self.postMessage('Unlocked'); // (C)
});
```

值得注意的是，等待B行的锁导致整个工作者线程被停止。这是真正的阻塞，JavaScript直到现在还不存在（等待异步函数是一个近似）。

6.2、实现共享锁

接下来，我们将看看Lars T. Hansen实现的一个ES6版本、基于SharedArrayBuffer的Lock。 在本节中，我们需要以下（以及其他）Atomics功能：

- Atomics.compareExchange（ta：TypedArray ，index，expectedValue，replacementValue）：T

如果ta[index]处的值为我们期待的值，则将其替换为替代值。返回索引的前一个（或未更改）的元素。

从几个常量和构造函数开始实现：

```
const UNLOCKED = 0;
const LOCKED_NO_WAITERS = 1;
const LOCKED_POSSIBLE_WAITERS = 2;

// Number of shared Int32 locations needed by the lock.
const NUMINTS = 1;

class Lock {

    /**
     * @param iab an Int32Array wrapping a SharedArrayBuffer
     * @param ibase an index inside iab, leaving enough room for NUMINTS
     */
    constructor(iab, ibase) {
        // OMITTED: check parameters
        this.iab = iab;
        this.ibase = ibase;
    }
```

构造函数主要将其参数存储在实例属性中。

锁定方法如下:

```
/**
 * Acquire the lock, or block until we can. Locking is not recursive:
 * you must not hold the lock when calling this.
 */
lock() {
    const iab = this.iab;
    const stateIdx = this.ibase;
    var c;
    if ((c = Atomics.compareExchange(iab, stateIdx, // (A)
    UNLOCKED, LOCKED_NO_WAITERS)) !== UNLOCKED) {
        do {
            if (c === LOCKED_POSSIBLE_WAITERS // (B)
            || Atomics.compareExchange(iab, stateIdx,
            LOCKED_NO_WAITERS, LOCKED_POSSIBLE_WAITERS) !== UNLOCKED) {
                Atomics.wait(iab, stateIdx, // (C)
                    LOCKED_POSSIBLE_WAITERS, Number.POSITIVE_INFINITY);
            }
        } while ((c = Atomics.compareExchange(iab, stateIdx,
        UNLOCKED, LOCKED_POSSIBLE_WAITERS)) !== UNLOCKED);
    }
}
```

在A行中，如果当前值为UNLOCKED，则将锁更改为LOCKED*NO*WAITERS。 如果锁已经被锁定（在这种情况下，compareExchange（ ）没有改变任何东西），我们将进入then代码块。

在B行（在do-while循环内），我们检查锁是否与waiters锁定或未锁定。鉴于我们将要等待，如果当前值为LOCKED*NO*WAITERS，则compareExchange（ ）也将切换到LOCKED*POSSIBLE*WAITERS。

在C行中，如果锁值为LOCKED*POSSIBLE*WAITERS，我们就等待。 最后一个参数Number.POSITIVE_INFINITY意味着等待永不超时。

当醒来后，如果我们没有解锁，我们会继续循环。 如果锁为UNLOCKED，则compareExchange（ ）也会切换到LOCKED*POSSIBLE*WAITERS。 我们使用LOCKED*POSSIBLEWAITERS而不是LOCKEDNO*WAITERS，因为我们需要在unlock（ ）临时将其设置为UNLOCKED并唤醒我们后恢复该值。

解锁方法如下：

```
    /**
     * Unlock a lock that is held.  Anyone can unlock a lock that
     * is held; nobody can unlock a lock that is not held.
     */
    unlock() {
        const iab = this.iab;
        const stateIdx = this.ibase;
        var v0 = Atomics.sub(iab, stateIdx, 1); // A

        // Wake up a waiter if there are any
        if (v0 !== LOCKED_NO_WAITERS) {
            Atomics.store(iab, stateIdx, UNLOCKED);
            Atomics.wake(iab, stateIdx, 1);
        }
    }

    // ...
}
```

在行A中，v0得到iab[stateIdx]在从中减去1之前的值。减法表示我们从LOCKED*NO*WAITERS到UNLOCKED，从LOCKED*POSSIBLE*WAITERS到LOCKED。

如果该值以前是LOCKED*NO*WAITERS，那么它现在是UNLOCKED，一切都很好（没有人醒来）。

否则，该值为LOCKED*POSSIBLE*WAITERS或UNLOCKED。 在前一种情况下，我们现在解锁，并且必须唤醒某人（通常会再次锁住）。 在后一种情况下，我们必须修正由减法创建的非法值，并且wake（ ）根本不做任何事情。

6.3、例子总结

以上给出了一个基于SharedArrayBuffer的锁的工作原理。 请记住，多线程代码是非常难写的，因为事情可以随时改变。 例子：lock.js是基于一个记录Linux内核的futex实现的文件。 该文件的标题是“Futex很棘手”（PDF）。

如果要更深入地使用Shared Array Buffers进行并行编程，请查看synchronizeic.js和它所基于的PDFPDF文档。

七、共享内存和原子有关的API

7.1、SharedArrayBuffer

构造函数

- new SharedArrayBuffer(length)。创建一个length个字节的缓冲区

静态属性

- get SharedArrayBuffer[Symbol.species]。默认返回this,可通过控制slice的返回值复写

实例属性

- get SharedArrayBuffer.prototype.byteLength()。返回缓冲区字节数
- SharedArrayBuffer.prototype.slice(start, end)。返回原始缓冲区的某一片

7.2、原子

原子函数的操作数必须是Typed Array（Int8Array, Uint8Array, Int16Array, Uint16Array, Int32Array or Uint32Array）的实例，该实例必须包裹着SharedArrayBuffer。

所有的函数都原子的执行，store操作的顺序是固定的，不能被编译器或CPU改变。

7.2.1 Loading and storing

- Atomics.load(ta : TypedArray, index) : T。读取和返回ta[index]
- Atomics.store(ta : TypedArray, index, value : T) : T。写入一个值到ta[index]并返回写入的值
- Atomics.exchange(ta : TypedArray, index, value : T) : T。设置ta[index]处的值，并返回设置前的值
- Atomics.compareExchange(ta : TypedArray, index, expectedValue, replacementValue) : T。如果ta[index]处的值是期待的值，则用替代值替换它。如果原始值是期待值，则返回替换的值，否则返回原始值。

7.2.2 简单修改 Typed Array 元素

下面的每一个函数都会改变 Typed Array指定索引位置的元素。它对提供一个参数，对元素施加一种操作，然后写入结果。它返回原始值。

- Atomics.add(ta : TypedArray, index, value) : T。执行 ta[index] += value 并返回 ta[index]处的原始值.
- Atomics.sub(ta : TypedArray, index, value) : T。执行 ta[index] -= value 并返回 ta[index]处的原始值.
- Atomics.and(ta : TypedArray, index, value) : T。执行 ta[index] &= value 并返回 ta[index]处的原始值.
- Atomics.or(ta : TypedArray, index, value) : T。执行 ta[index] |= value 并返回 ta[index]处的原始值.
- Atomics.xor(ta : TypedArray, index, value) : T。执行 ta[index] ^= value 并返回 ta[index]处的原始值.

7.2.3、等待和唤醒

等待和唤醒操作需要参数ta是一个Int32Array的实例.

- Atomics.wait(ta: Int32Array, index, value, timeout=Number.POSITIVE_INFINITY) : ('not-equal' | 'ok' | 'timed-out')。如果ta[index]当前值不是一个数值，返回'not-equal'。否则休眠直到通过 Atomics.wake()唤

醒或者超时。前一种情况返回ok，后一种情况返回毫秒数'timed-out', 该函数的助记符是“wait if ta[index] is value”.

- Atomics.wake(ta : Int32Array, index, count)。唤醒在ta[index]处的工作者线程.

7.2.4、杂项

- Atomics.isLockFree（大小）如果没有锁定，可以操作具有给定大小（以字节为单位）的操作数，这个函数可以让您询问JavaScript引擎。 这可以通知算法是否要依赖内置的基元（compareExchange（ ）等）或使用自己的锁。 Atomics.isLockFree（4）总是返回true，因为这是所有当前相关的支持。

八、常见问题

8.1、浏览器支持情况如何？

目前而言某些浏览器通过如下方式支持：

- Firefox (50.1.0+): go to `about:config` and set `javascript.options.shared_memory` to `true`
- Safari Technology Preview (Release 21+): enabled by default.
- Chrome Canary (58.0+): There are two ways to switch it on.
    - Via `chrome://flags/` (“Experimental enabled SharedArrayBuffer support in JavaScript”)
    - `--js-flags=--harmony-sharedarraybuffer --enable-blink-feature=SharedArrayBuffer`

第7章. Object.entries()和 Object.values()

ECMAScript2017（ES8）之对象迭代

一、概述

ES6中大部分数据结构都是可以迭代的，但是对象字面量却是不可迭代的，要想对对象进行迭代，我们只能借用Object.keys()或其它方法来进行一定程度的迭代，但远远没有其它数据结构的迭代那么方便。
ECMAScript2017中新增了两个方法**Object.entries()**和**Object.values()**用于增强对象的迭代。

本章将讲解由Jordan Harband提交的这两种**Object.entries()**和**Object.values()**方法。

二、基本用法

Object.values的基本用法如下：

```
let obj = {
  name: 'libai',
  age: 1
}
for (let value of Object.values(obj)){
  console.log(value)
}
```

```
libai
```

```
1
```

Object.entries的基本用法如下：

```
let obj = {
  name: 'libai',
  age: 1
}
for (let [key, value] of Object.entries(obj)){
  console.log(`${key} : ${value}`)
}
```

```
name : libai
```

```
age : 1
```

三、Object.values()详解

Object.values具有如下的语法签名：

```
Object.values(value: any): Array<any>
```

它用来枚举键值。返回的是一个数组。注意不是返回迭代器

注意，它只能得到字符串类型的可枚举自有属性的值。**不能得到Symbol类型的属性的值。**

四、Object.entries()详解

Object.entries具有如下的语法签名：

```
Object.values(value: any): Array<[string,any]>
```

Object.entries(x)将x隐式转换为对象，并且以[key,value]形式返回所有字符串类型的可枚举自有属性及其值。它的返回值是由[key,value]构成的数组。

```
> let obj = {
    name: 'libai',
    age: 1
  }
  Object.entries(obj)
<· ▼(2) [Array(2), Array(2)]
    ▼0: Array(2)
        0: "name"
        1: "libai"
        length: 2
      ▶__proto__: Array(0)
    ▼1: Array(2)
        0: "age"
        1: 1
        length: 2
```

对于Symbol类型的属性，将会忽略：

```
> let obj = {
    [Symbol()]: 'libai',
    age: 1
  }
  Object.entries(obj)
<- ▼[Array(2)] i
    ▼0: Array(2)
        0: "age"
        1: 1
        length: 2
      ▶__proto__: Array(0)
      length: 1
    ▶__proto__: Array(0)
>
```

Object.entries()的返回值可以很方便的用于构造Map:

```
> let obj = {
    name: 'libai',
    age: 1
  }
  new Map(Object.entries(obj))
<- ▼Map(2) {"name" => "libai", "age" => 1} i
      size: (...)
    ▶__proto__: Map
    ▼[[Entries]]: Array(2)
      ▶0: {"name" => "libai"}
      ▶1: {"age" => 1}
        length: 2
```

五、常见问题

5.1、为什么Object.entries()和Object.values()的返回值是数组而不是迭代器？

这是为了和Obejct.keys()一致

5.2、为什么Object.entries()和Object.values()只能枚举字符串类型的可枚举自有属性？

这也是为了和Obejct.keys()一致，如果你想枚举所有属性可以使用Reflect.ownEntries()

第8章. 新的字符串方法：padStart 和 padEnd

ECMAScript2017之字符串新特性

本章将讲述由`Jordan Harband`和`Rick Waldon`提交的ECMAScript2017字符串操作的两个新的方法：padStart和padEnd

一、概述

ECMAScript2017为字符串操作提供了两个新的方法：padStart和padEnd

```
> 'x'.padStart(5,'ay')
"ayayx"
> 'x'.padEnd(5,'abc')
"xabca"
```

二、为什么要填充字符串？

填充字符串的使用场合主要有一下几个：

1. 将表格数据显示为等宽
2. 给一个文件UTL添加数值或ID：'file001.txt'
3. 对齐输出
4. 打印拥有前缀的十六进制或二进制数据：'0x00FF'

三、padSatrt的用法

3.1、基本用法

padStart的语法签名如下：

```
String.prototype.padStart(maxLength, padString=' ')
```

这个方法将会在调用该方法的字符串前面填充padString字符串直到达到最大长度值maxLength。

如果一次填充不够达到maxLength它会持续填充：

```
> 'x'.padStart(5,'ay')//一次不够
"ayayx"
```

如果padString与调用该函数的字符串的总长度比maxLength还大的话，将会用padString的一部分进行填充：

```
> 'x'.padStart(5,'abcdefg')//一次够
<- "abcdx"
```

如果调用该函数的字符串本身的长度比maxLength还大的话，将不会进行填充操作：

```
> 'abcdefg'.padStart(5,'x')
<- "abcdefg"
```

如果你省略了padString参数，他将会用‘ ’字符串进行填充：

```
> 'x'.padStart(5)
<- "    x"
```

3.2、粗略实现

下面是padStart的一个粗略的实现思路，它对于某些边界条件还不完全兼容：

```
String.prototype.padStart =
function (maxLength, fillString=' ') {
    let str = String(this);
    if (str.length >= maxLength) {
        return str;
    }

    fillString = String(fillString);
    if (fillString.length === 0) {
        fillString = ' ';
    }

    let fillLen = maxLength - str.length;
    let timesToRepeat = Math.ceil(fillLen / fillString.length);
    let truncatedStringFiller = fillString
        .repeat(timesToRepeat)
        .slice(0, fillLen);
    return truncatedStringFiller + str;
};
```

四、padEnd的用法

3.1、基本用法

padEnd的语法签名如下：

```
String.prototype.padEnd(maxLength, padString=' ')
```

它与padStart的用法基本一样，只不过它是在调用者的尾部填充。

3.2、粗略实现

它的实现与padStart也基本一样，只不过最后返回值是这样的写法：

```
return str + truncatedStringFiller;
```

五、常见问题

5.1、为什么填充方法不叫padLeft和padRight?

一方面对于双向文字或者方向文字（比如说中国古代从右向左的书写形式），left和right的意义就混乱了 另一方面这和String现有的两个方法startWith和EndWith名称相一致。

第9章. Object.getOwnPropertyDescriptors()

ECMAScript2017之getOwnPropertyDescriptors

本章将讲述由`Jordan Harband`和`Andrea Giammarchi`提交的ECMAScript2017新特性——Object.getOwnPropertyDescriptors()

一、概述

Object.getOwnPropertyDescriptors(obj)以对象的形式返回obj对象所有自有属性（包括字符属性和Symbol属性）的属性描述。如下：

```
> let obj = {
      [Symbol('foo')]: 3,
      get bar() { return 'abc' },
      name: 'libai'
  }
  Object.getOwnPropertyDescriptors(obj)
< ▼Object {bar: Object, name: Object, Symbol(foo): Object}
    ▼bar: Object
       configurable: true
       enumerable: true
     ▶get: function bar()
       set: undefined
     ▶__proto__: Object
    ▶name: Object
    ▼Symbol(foo): Object
       configurable: true
       enumerable: true
       value: 3
       writable: true
     ▶__proto__: Object
    ▶__proto__: Object
```

二、Object.getOwnPropertyDescriptors()

Object.getOwnPropertyDescriptors()接收一个对象作为参数，然后以对象的形式返回obj对象所有属性的属性描述：

> 对于obj的每一个自有属性，该方法给它的返回结果对象添加一个属性，属性名就是obj的那个属性，属性值就是obj的那个属性的描述符

属性描述符描述了该属性的一些信息（它的值，它的可读、可写、可枚举、可配置等信息）。下面是个例子，和概述例子一样：

```
> let obj = {
      [Symbol('foo')]: 3,
      get bar() { return 'abc' },
      name: 'libai'
  }
  Object.getOwnPropertyDescriptors(obj)
< ▼Object {bar: Object, name: Object, Symbol(foo): Object}
    ▼bar: Object
        configurable: true
        enumerable: true
      ▶get: function bar()
        set: undefined
      ▶__proto__: Object
    ▶name: Object
    ▼Symbol(foo): Object
        configurable: true
        enumerable: true
        value: 3
        writable: true
      ▶__proto__: Object
    ▶__proto__: Object
```

其实Object.getOwnPropertyDescriptors()的实现很简单，就是遍历对象的属性然后调用之前就有的函数Object.getOwnPropertyDescriptor，需要注意的是要遍历所有自有属性：

```
> function getOwnPropertyDescriptors(obj) {
      const result = {}
      for (let key of Reflect.ownKeys(obj)) {
          result[key] = Object.getOwnPropertyDescriptor(obj, key)
      }
      return result
  }
```

三、Object.getOwnPropertyDescriptors()使用场合

3.1、场合1：复制对象属性

在ES6中提供了一个复制对象属性的方法Object.assign()。但是这个对象仅仅只是调用get和set操作来复制对象属性，所完成的不过是一下操作：

```
const value = source[key]
target[key] = value
```

这意味着它没法正确的复制非默认属性（getters、setters、non-writable属性等等），下面的例子说明了这一点：

```
> let obj = {
      set foo (value) {
          console.log(value)
      }
  }
  Object.getOwnPropertyDescriptors(obj)
◂ ▼ Object {foo: Object} i
    ▼ foo: Object
        configurable: true
        enumerable: true
        get: undefined
      ▶ set: function foo(value)
```

assign之后的失败结果：

```
> let target = {}
  Object.assign(target, obj)
  Object.getOwnPropertyDescriptors(target)
◂ ▼ Object {foo: Object} i
    ▼ foo: Object
        configurable: true
        enumerable: true
        value: undefined
        writable: true
      ▶ __proto__: Object
    ▶ __proto__: Object
```

可以看到assign并没有实现正确的复制（原属性的getter和setter都没复制过来）。

幸运的是，我们可以结合Object.definedProperties和Object.getOwnPropertyDescriptors()来实现预期的结果：

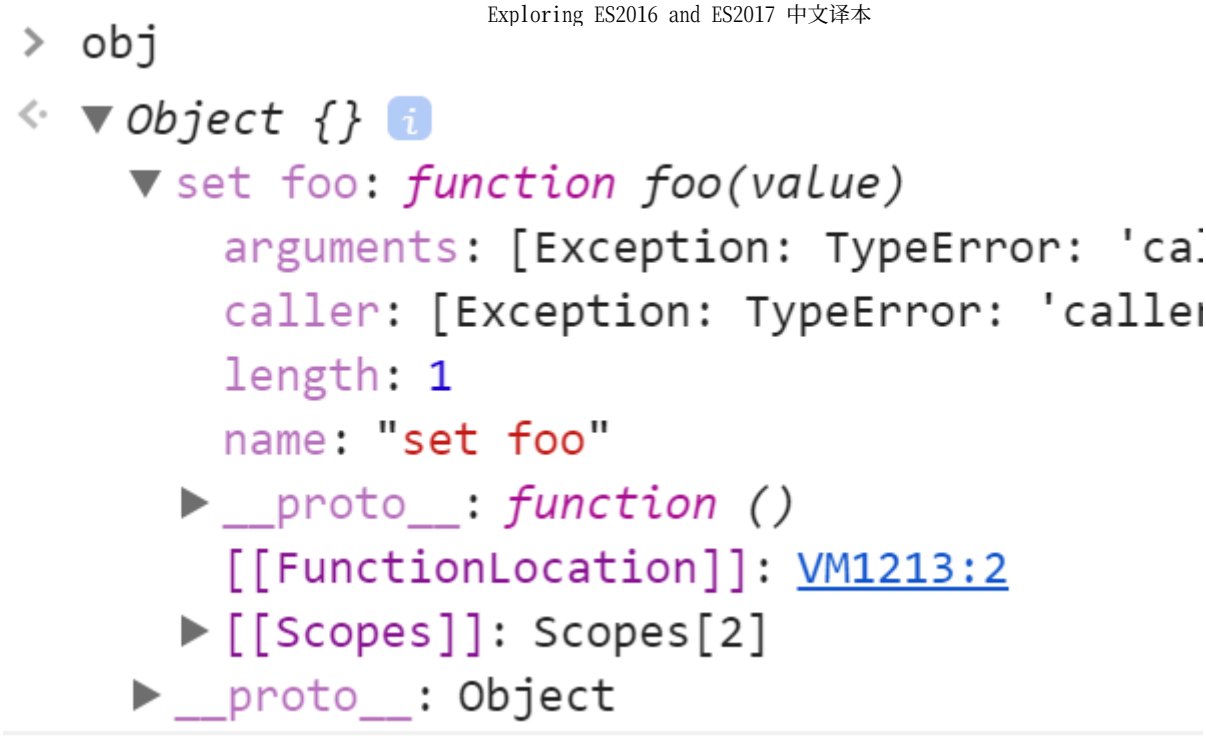
```
> let target2 = {}
  Object.defineProperties(target2, Object.getOwnPropertyDescriptors(obj))
  Object.getOwnPropertyDescriptors(target2)
◂ ▼ Object {foo: Object} i
    ▼ foo: Object
        configurable: true
        enumerable: true
        get: undefined
      ▶ set: function foo(value)
      ▶ __proto__: Object
    ▶ __proto__: Object
```

3.2、场合2：克隆对象

浅克隆就和复制对象属性一个道理，所以Object.getOwnPropertyDescriptors()非常适合用来做浅克隆。 现在我们使用Object.create()，并给他两个参数：

- 第一个参数是对象的原型
- 第二个可选参数是对象的属性吗，描述符集合，他是用Object.getOwnPropertyDescriptors()得到的

```
> const clone = Object.create(Object.getPrototypeOf(obj),
   Object.getOwnPropertyDescriptors(obj))
◂ undefined
> clone
◂ ▼ Object {} i
    ▼ set foo: function foo(value)
        arguments: [Exception: TypeError: 'caller' and 'ar
        caller: [Exception: TypeError: 'caller' and 'argum
        length: 1
        name: "set foo"
      ▶ __proto__: function ()
```



3.3、跨平台地指定对象字面量的任意原型

使用对象字面量创建具有任意原型对象的语法最好的方法是使用特殊属性`__proto__`：

```
let obj = {
    __proto__: prot,//这里prot表示某个原型对象
    foo: 123
}
```

但是`__proto__`只有某些浏览器（比如Chrome）支持,一种普遍的变通方案是用Object.create()方法：

```
const obj = Object.create(prot)//这里prot表示某个原型对象
obj.foo = 123
```

这是因为`A = Object.create(B)`实现的就是`A.__proto__ = B`这么一个功能。

你也可以结合Object.getOwnPropertyDescriptors()：

```
const obj = Object.create(prot,
    Object.getOwnPropertyDescriptors({
        foo: 123
    })
)//这里prot表示某个原型对象
```

另外一种变通方法是使用assign():

```
const obj = Object.assign(
    object.create(prot),
    {
        foo: 123
    }
)//这里prot表示某个原型对象
```

四、陷阱

对象A的某个方法B中使用了super方法的话，该方法(B)将与其home对象（对象A存储在其中）牢固连接。目前没有办法将方法B复制或移动到其它的对象。

第 10 章. 函数参数列表和调用时尾部的逗号

ECMAScript2017之参数列表尾部的逗号

本章讲述ECMAScript2017中由`Jeff Morrision`提交的“函数定义和调用时参数列表尾部的参数”。

一、概述

在ECMAScript2017中规定，函数定义时参数列表尾部的逗号是合法的：

```
> function test(arg1,arg2,){

  }
< undefined
> test
< function test(arg1,arg2,){

  }
```

同样，函数调用时，参数列表尾部的逗号也是合法的：

```
> function test(arg1,arg2,){
      console.log(arg1,arg2)
  }
  test(1,2,)
  1 2
```

二、对象和字面量和数组字面量尾部的逗号

对象字面量以及**数组字面量**尾部的逗号将会被忽略：

```
> let obj = {
      name: 'libai',
      age: 1,
  }
  let arr = ['1','2',]
<· undefined
> obj
<· ▶ Object {name: "libai", age: 1}
> arr
<· ▶ (2) ["1", "2"]
```

为什么这很有用？

1. 重新排列元素会很方便，因为如果最后元素改变的话你不需要去增加或删除逗号
2. 他可以帮助版本管理系统来判断什么东西被真正改变了

比如说，下面两个从A到B，版本管理器会认为foo和bar这两个元素都被改变了：

```
A:  ['foo']     B: ['foo','bar']
```

但实际上，这里面只有bar是新增的。

三、新特性：允许函数定义以及函数调用时参数列表尾部有逗号

这就是概述所说的两个场景，这里不再赘述。