11.3.3　异步函数策略

因为简单实用，所以异步函数很快成为 JavaScript 项目使用最广泛的特性之一。不过，在使用异步函数时，还是有些问题要注意。

1. 实现 sleep()

很多人在刚开始学习 JavaScript 时，想找到一个类似 Java 中 `Thread.sleep()` 之类的函数，好在程序中加入非阻塞的暂停。以前，这个需求基本上都通过 `setTimeout()` 利用 JavaScript 运行时的行为来实现的。

有了异步函数之后，就不一样了。一个简单的箭头函数就可以实现 `sleep()`：

```
async function sleep(delay) {
  return new Promise((resolve) => setTimeout(resolve, delay));
}

async function foo() {
  const t0 = Date.now();
  await sleep(1500); // 暂停约 1500 毫秒
  console.log(Date.now() - t0);
}
foo();
// 1502
```

2. 利用平行执行

如果使用 `await` 时不留心，则很可能错过平行加速的机会。来看下面的例子，其中顺序等待了 5 个随机的超时：

```
async function randomDelay(id) {
  // 延迟 0~1000 毫秒
  const delay = Math.random() * 1000;
  return new Promise((resolve) => setTimeout(() => {
    console.log(`${id} finished`);
    resolve();
  }, delay));
}

async function foo() {
  const t0 = Date.now();
  await randomDelay(0);
  await randomDelay(1);
  await randomDelay(2);
  await randomDelay(3);
  await randomDelay(4);
  console.log(`${Date.now() - t0}ms elapsed`);
}
foo();

// 0 finished
// 1 finished
// 2 finished
// 3 finished
// 4 finished
// 877ms elapsed
```

用一个 `for` 循环重写，就是：

```
async function randomDelay(id) {
  // 延迟 0~1000 毫秒
  const delay = Math.random() * 1000;
  return new Promise((resolve) => setTimeout(() => {
    console.log(`${id} finished`);
    resolve();
  }, delay));
}

async function foo() {
  const t0 = Date.now();
  for (let i = 0; i < 5; ++i) {
    await randomDelay(i);
  }

  console.log(`${Date.now() - t0}ms elapsed`);
}
foo();

// 0 finished
// 1 finished
// 2 finished
// 3 finished
// 4 finished
// 877ms elapsed
```

就算这些期约之间没有依赖，异步函数也会依次暂停，等待每个超时完成。这样可以保证执行顺序，但总执行时间会变长。

如果顺序不是必需保证的，那么可以先一次性初始化所有期约，然后再分别等待它们的结果。比如：

```
async function randomDelay(id) {
  // 延迟 0~1000 毫秒
  const delay = Math.random() * 1000;
  return new Promise((resolve) => setTimeout(() => {
    setTimeout(console.log, 0, `${id} finished`);
    resolve();
  }, delay));
}

async function foo() {
  const t0 = Date.now();

  const p0 = randomDelay(0);
  const p1 = randomDelay(1);
  const p2 = randomDelay(2);
  const p3 = randomDelay(3);
  const p4 = randomDelay(4);

  await p0;
  await p1;
  await p2;
  await p3;
  await p4;

  setTimeout(console.log, 0, `${Date.now() - t0}ms elapsed`);
}
foo();

// 1 finished
```

```
// 4 finished
// 3 finished
// 0 finished
// 2 finished
// 877ms elapsed
```

用数组和 for 循环再包装一下就是：

```
async function randomDelay(id) {
  // 延迟 0~1000 毫秒
  const delay = Math.random() * 1000;
  return new Promise((resolve) => setTimeout(() => {
    console.log(`${id} finished`);
    resolve();
  }, delay));
}

async function foo() {
  const t0 = Date.now();

  const promises = Array(5).fill(null).map((_, i) => randomDelay(i));

  for (const p of promises) {
    await p;
  }

  console.log(`${Date.now() - t0}ms elapsed`);
}
foo();

// 4 finished
// 2 finished
// 1 finished
// 0 finished
// 3 finished
// 877ms elapsed
```

注意，虽然期约没有按照顺序执行，但 await **按顺序**收到了每个期约的值：

```
async function randomDelay(id) {
  // 延迟 0~1000 毫秒
  const delay = Math.random() * 1000;
  return new Promise((resolve) => setTimeout(() => {
    console.log(`${id} finished`);
    resolve(id);
  }, delay));
}

async function foo() {
  const t0 = Date.now();

  const promises = Array(5).fill(null).map((_, i) => randomDelay(i));

  for (const p of promises) {
    console.log(`awaited ${await p}`);
  }

  console.log(`${Date.now() - t0}ms elapsed`);
}
foo();
```

```
// 1 finished
// 2 finished
// 4 finished
// 3 finished
// 0 finished
// awaited 0
// awaited 1
// awaited 2
// awaited 3
// awaited 4
// 645ms elapsed
```

3. 串行执行期约

在 11.2 节，我们讨论过如何串行执行期约并把值传给后续的期约。使用 async/await，期约连锁会变得很简单：

```
function addTwo(x) {return x + 2;}
function addThree(x) {return x + 3;}
function addFive(x) {return x + 5;}

async function addTen(x) {
  for (const fn of [addTwo, addThree, addFive]) {
    x = await fn(x);
  }
  return x;
}

addTen(9).then(console.log); // 19
```

这里，await 直接传递了每个函数的返回值，结果通过迭代产生。当然，这个例子并没有使用期约，如果要使用期约，则可以把所有函数都改成异步函数。这样它们就都返回期约了：

```
async function addTwo(x) {return x + 2;}
async function addThree(x) {return x + 3;}
async function addFive(x) {return x + 5;}

async function addTen(x) {
  for (const fn of [addTwo, addThree, addFive]) {
    x = await fn(x);
  }
  return x;
}

addTen(9).then(console.log); // 19
```


4. 栈追踪与内存管理

期约与异步函数的功能有相当程度的重叠，但它们在内存中的表示则差别很大。看看下面的例子，它展示了拒绝期约的栈追踪信息：

```
function fooPromiseExecutor(resolve, reject) {
  setTimeout(reject, 1000, 'bar');
}

function foo() {
  new Promise(fooPromiseExecutor);
}
```

```
foo();
// Uncaught (in promise) bar
//   setTimeout
//   setTimeout (async)
//   fooPromiseExecutor
//   foo
```

根据对期约的不同理解程度，以上栈追踪信息可能会让某些读者不解。栈追踪信息应该相当直接地表现 JavaScript 引擎当前栈内存中函数调用之间的嵌套关系。在超时处理程序执行时和拒绝期约时，我们看到的错误信息包含嵌套函数的标识符，那是被调用以创建最初期约实例的函数。可是，我们知道这些函数**已经返回**了，因此栈追踪信息中不应该看到它们。

答案很简单，这是因为 JavaScript 引擎会在创建期约时尽可能保留完整的调用栈。在抛出错误时，调用栈可以由运行时的错误处理逻辑获取，因而就会出现在栈追踪信息中。当然，这意味着栈追踪信息会占用内存，从而带来一些计算和存储成本。

如果在前面的例子中使用的是异步函数，那又会怎样呢？比如：

```
function fooPromiseExecutor(resolve, reject) {
  setTimeout(reject, 1000, 'bar');
}

async function foo() {
  await new Promise(fooPromiseExecutor);
}
foo();

// Uncaught (in promise) bar
//   foo
//   async function (async)
//   foo
```

这样一改，栈追踪信息就准确地反映了当前的调用栈。`fooPromiseExecutor()`已经返回，所以它不在错误信息中。但 `foo()`此时被挂起了，并没有退出。JavaScript 运行时可以简单地在嵌套函数中存储指向包含函数的指针，就跟对待同步函数调用栈一样。这个指针实际上存储在内存中，可用于在出错时生成栈追踪信息。这样就不会像之前的例子那样带来额外的消耗，因此在重视性能的应用中是可以优先考虑的。

11.4　小结

长期以来，掌握单线程 JavaScript 运行时的异步行为一直都是个艰巨的任务。随着 ES6 新增了期约和 ES8 新增了异步函数，ECMAScript 的异步编程特性有了长足的进步。通过期约和 async/await，不仅可以实现之前难以实现或不可能实现的任务，而且也能写出更清晰、简洁，并且容易理解、调试的代码。

期约的主要功能是为异步代码提供了清晰的抽象。可以用期约表示异步执行的代码块，也可以用期约表示异步计算的值。在需要串行异步代码时，期约的价值最为突出。作为可塑性极强的一种结构，期约可以被序列化、连锁使用、复合、扩展和重组。

异步函数是将期约应用于 JavaScript 函数的结果。异步函数可以暂停执行，而不阻塞主线程。无论是编写基于期约的代码，还是组织串行或平行执行的异步代码，使用异步函数都非常得心应手。异步函数可以说是现代 JavaScript 工具箱中最重要的工具之一。

第 12 章

BOM

本章内容

- ❑ 理解 BOM 的核心——window 对象
- ❑ 控制窗口及弹窗
- ❑ 通过 location 对象获取页面信息
- ❑ 使用 navigator 对象了解浏览器
- ❑ 通过 history 对象操作浏览器历史

虽然 ECMAScript 把浏览器对象模型（BOM，Browser Object Model）描述为 JavaScript 的核心，但实际上 BOM 是使用 JavaScript 开发 Web 应用程序的核心。BOM 提供了与网页无关的浏览器功能对象。多年来，BOM 是在缺乏规范的背景下发展起来的，因此既充满乐趣又问题多多。毕竟，浏览器开发商都按照自己的意愿来为它添砖加瓦。最终，浏览器实现之间共通的部分成为了事实标准，为 Web 开发提供了浏览器间互操作的基础。HTML5 规范中有一部分涵盖了 BOM 的主要内容，因为 W3C 希望将 JavaScript 在浏览器中最基础的部分标准化。

12.1 window 对象

BOM 的核心是 window 对象，表示浏览器的实例。window 对象在浏览器中有两重身份，一个是 ECMAScript 中的 Global 对象，另一个就是浏览器窗口的 JavaScript 接口。这意味着网页中定义的所有对象、变量和函数都以 window 作为其 Global 对象，都可以访问其上定义的 parseInt() 等全局方法。

> **注意** 因为 window 对象的属性在全局作用域中有效，所以很多浏览器 API 及相关构造函数都以 window 对象属性的形式暴露出来。这些 API 将在全书各章中介绍，特别是第 20 章。
>
> 另外，由于实现不同，某些 window 对象的属性在不同浏览器间可能差异很大。本章不会介绍已经废弃的、非标准化或特定于浏览器的 window 属性。

12.1.1 Global 作用域

因为 window 对象被复用为 ECMAScript 的 Global 对象，所以通过 var 声明的所有全局变量和函数都会变成 window 对象的属性和方法。比如：

```
var age = 29;
var sayAge = () => alert(this.age);

alert(window.age); // 29
```

```
sayAge();          // 29
window.sayAge();   // 29
```

这里，变量 age 和函数 sayAge()被定义在全局作用域中，它们自动成为了 window 对象的成员。因此，变量 age 可以通过 window.age 来访问，而函数 sayAge()也可以通过 window.sayAge()来访问。因为 sayAge()存在于全局作用域，this.age 映射到 window.age，所以就可以显示正确的结果了。

如果在这里使用 let 或 const 替代 var，则不会把变量添加给全局对象：

```
let age = 29;
const sayAge = () => alert(this.age);

alert(window.age);  // undefined
sayAge();           // undefined
window.sayAge();    // TypeError: window.sayAge is not a function
```

另外，访问未声明的变量会抛出错误，但是可以在 window 对象上查询是否存在可能未声明的变量。比如：

```
// 这会导致抛出错误，因为 oldValue 没有声明
var newValue = oldValue;
// 这不会抛出错误，因为这里是属性查询
// newValue 会被设置为 undefined
var newValue = window.oldValue;
```

记住，JavaScript 中有很多对象都暴露在全局作用域中，比如 location 和 navigator（本章后面都会讨论），因而它们也是 window 对象的属性。

12.1.2　窗口关系

top 对象始终指向最上层（最外层）窗口，即浏览器窗口本身。而 parent 对象则始终指向当前窗口的父窗口。如果当前窗口是最上层窗口，则 parent 等于 top（都等于 window）。最上层的 window 如果不是通过 window.open()打开的，那么其 name 属性就不会包含值，本章后面会讨论。

还有一个 self 对象，它是终极 window 属性，始终会指向 window。实际上，self 和 window 就是同一个对象。之所以还要暴露 self，就是为了和 top、parent 保持一致。

这些属性都是 window 对象的属性，因此访问 window.parent、window.top 和 window.self 都可以。这意味着可以把访问多个窗口的 window 对象串联起来，比如 window.parent.parent。

12.1.3　窗口位置与像素比

window 对象的位置可以通过不同的属性和方法来确定。现代浏览器提供了 screenLeft 和 screenTop 属性，用于表示窗口相对于屏幕左侧和顶部的位置，返回值的单位是 CSS 像素。

可以使用 moveTo()和 moveBy()方法移动窗口。这两个方法都接收两个参数，其中 moveTo()接收要移动到的新位置的绝对坐标 x 和 y；而 moveBy()则接收相对当前位置在两个方向上移动的像素数。比如：

```
// 把窗口移动到左上角
window.moveTo(0,0);

// 把窗口向下移动 100 像素
window.moveBy(0, 100);
```

```
// 把窗口移动到坐标位置(200, 300)
window.moveTo(200, 300);

// 把窗口向左移动 50 像素
window.moveBy(-50, 0);
```

依浏览器而定，以上方法可能会被部分或全部禁用。

像素比

CSS 像素是 Web 开发中使用的统一像素单位。这个单位的背后其实是一个角度：0.0213°。如果屏幕距离人眼是一臂长，则以这个角度计算的 CSS 像素大小约为 1/96 英寸。这样定义像素大小是为了在不同设备上统一标准。比如，低分辨率平板设备上 12 像素（CSS 像素）的文字应该与高清 4K 屏幕下 12 像素（CSS 像素）的文字具有相同大小。这就带来了一个问题，不同像素密度的屏幕下就会有不同的缩放系数，以便把物理像素（屏幕实际的分辨率）转换为 CSS 像素（浏览器报告的虚拟分辨率）。

举个例子，手机屏幕的**物理**分辨率可能是 1920×1080，但因为其像素可能非常小，所以浏览器就需要将其分辨率降为较低的**逻辑**分辨率，比如 640×320。这个物理像素与 CSS 像素之间的转换比率由 `window.devicePixelRatio` 属性提供。对于分辨率从 1920×1080 转换为 640×320 的设备，`window.devicePixelRatio` 的值就是 3。这样一来，12 像素（CSS 像素）的文字实际上就会用 36 像素的物理像素来显示。

`window.devicePixelRatio` 实际上与每英寸像素数（DPI，dots per inch）是对应的。DPI 表示单位像素密度，而 `window.devicePixelRatio` 表示物理像素与逻辑像素之间的缩放系数。

12.1.4 窗口大小

在不同浏览器中确定浏览器窗口大小没有想象中那么容易。所有现代浏览器都支持 4 个属性：`innerWidth`、`innerHeight`、`outerWidth` 和 `outerHeight`。`outerWidth` 和 `outerHeight` 返回浏览器窗口自身的大小（不管是在最外层 `window` 上使用，还是在窗格`<frame>`中使用）。`innerWidth` 和 `innerHeight` 返回浏览器窗口中页面视口的大小（不包含浏览器边框和工具栏）。

`document.documentElement.clientWidth` 和 `document.documentElement.clientHeight` 返回页面视口的宽度和高度。

浏览器窗口自身的精确尺寸不好确定，但可以确定页面视口的大小，如下所示：

```
let pageWidth = window.innerWidth,
    pageHeight = window.innerHeight;

if (typeof pageWidth != "number") {
  if (document.compatMode == "CSS1Compat"){
    pageWidth = document.documentElement.clientWidth;
    pageHeight = document.documentElement.clientHeight;
  } else {
    pageWidth = document.body.clientWidth;
    pageHeight = document.body.clientHeight;
  }
}
```

这里，先将 `pageWidth` 和 `pageHeight` 的值分别设置为 `window.innerWidth` 和 `window.innerHeight`。然后，检查 `pageWidth` 是不是一个数值，如果不是则通过 `document.compatMode` 来检查页面是否处于标准模式。如果是，则使用 `document.documentElement.clientWidth` 和

document.documentElement.clientHeight；否则，就使用 document.body.clientWidth 和 document.body.clientHeight。

在移动设备上，window.innerWidth 和 window.innerHeight 返回视口的大小，也就是屏幕上页面可视区域的大小。Mobile Internet Explorer 支持这些属性，但在 document.documentElement.clientWidth 和 document.documentElement.clientHeight 中提供了相同的信息。在放大或缩小页面时，这些值也会相应变化。

在其他移动浏览器中，document.documentElement.clientWidth 和 document.documentElement.clientHeight 返回的布局视口的大小，即渲染页面的实际大小。布局视口是相对于可见视口的概念，可见视口只能显示整个页面的一小部分。Mobile Internet Explorer 把布局视口的信息保存在 document.body.clientWidth 和 document.body.clientHeight 中。在放大或缩小页面时，这些值也会相应变化。

因为桌面浏览器的差异，所以需要先确定用户是不是在使用移动设备，然后再决定使用哪个属性。

> **注意**　手机视口的概念比较复杂，有各种各样的问题。如果读者在做移动开发，推荐阅读 Peter-Paul Koch 发表在 QuirksMode 网站上的文章“A Tale of Two Viewports— Part Two”。

可以使用 resizeTo()和 resizeBy()方法调整窗口大小。这两个方法都接收两个参数，resizeTo()接收新的宽度和高度值，而 resizeBy()接收宽度和高度各要缩放多少。下面看个例子：

```
// 缩放到 100×100
window.resizeTo(100, 100);

// 缩放到 200×150
window.resizeBy(100, 50);

// 缩放到 300×300
window.resizeTo(300, 300);
```

与移动窗口的方法一样，缩放窗口的方法可能会被浏览器禁用，而且在某些浏览器中默认是禁用的。同样，缩放窗口的方法只能应用到最上层的 window 对象。

12.1.5　视口位置

浏览器窗口尺寸通常无法满足完整显示整个页面，为此用户可以通过滚动在有限的视口中查看文档。度量文档相对于视口滚动距离的属性有两对，返回相等的值：window.pageXoffset/window.scrollX 和 window.pageYoffset/window.scrollY。

可以使用 scroll()、scrollTo()和 scrollBy()方法滚动页面。这 3 个方法都接收表示相对视口距离的 x 和 y 坐标，这两个参数在前两个方法中表示要滚动到的坐标，在最后一个方法中表示滚动的距离。

```
// 相对于当前视口向下滚动 100 像素
window.scrollBy(0, 100);

// 相对于当前视口向右滚动 40 像素
window.scrollBy(40, 0);

// 滚动到页面左上角
window.scrollTo(0, 0);
```

```
// 滚动到距离屏幕左边及顶边各 100 像素的位置
window.scrollTo(100, 100);
```

这几个方法也都接收一个 ScrollToOptions 字典，除了提供偏移值，还可以通过 behavior 属性告诉浏览器是否平滑滚动。

```
// 正常滚动
window.scrollTo({
  left: 100,
  top: 100,
  behavior: 'auto'
});

// 平滑滚动
window.scrollTo({
  left: 100,
  top: 100,
  behavior: 'smooth'
});
```

12.1.6 导航与打开新窗口

window.open()方法可以用于导航到指定 URL，也可以用于打开新浏览器窗口。这个方法接收 4 个参数：要加载的 URL、目标窗口、特性字符串和表示新窗口在浏览器历史记录中是否替代当前加载页面的布尔值。通常，调用这个方法时只传前 3 个参数，最后一个参数只有在不打开新窗口时才会使用。

如果 window.open()的第二个参数是一个已经存在的窗口或窗格（frame）的名字，则会在对应的窗口或窗格中打开 URL。下面是一个例子：

```
// 与<a href="http://www.wrox.com" target="topFrame"/>相同
window.open("http://www.wrox.com/", "topFrame");
```

执行这行代码的结果就如同用户点击了一个 href 属性为"http://www.wrox.com"，target 属性为"topFrame"的链接。如果有一个窗口名叫"topFrame"，则这个窗口就会打开这个 URL；否则就会打开一个新窗口并将其命名为"topFrame"。第二个参数也可以是一个特殊的窗口名，比如_self、_parent、_top 或_blank。

1. 弹出窗口

如果 window.open()的第二个参数不是已有窗口，则会打开一个新窗口或标签页。第三个参数，即特性字符串，用于指定新窗口的配置。如果没有传第三个参数，则新窗口（或标签页）会带有所有默认的浏览器特性（工具栏、地址栏、状态栏等都是默认配置）。如果打开的不是新窗口，则忽略第三个参数。

特性字符串是一个逗号分隔的设置字符串，用于指定新窗口包含的特性。下表列出了一些选项。

设　置	值	说　明
fullscreen	"yes"或"no"	表示新窗口是否最大化。仅限 IE 支持
height	数值	新窗口高度。这个值不能小于 100
left	数值	新窗口的 *x* 轴坐标。这个值不能是负值
location	"yes"或"no"	表示是否显示地址栏。不同浏览器的默认值也不一样。在设置为"no"时，地址栏可能隐藏或禁用（取决于浏览器）