Web Worker

概述

JavaScript 语言采用的是单线程模型，也就是说，所有任务只能在一个线程上完成，一次只能做一件事。前面的任务没做完，后面的任务只能等着。随着电脑计算能力的增强，尤其是多核 CPU 的出现，单线程带来很大的不便，无法充分发挥计算机的计算能力。

Web Worker 的作用，就是为 JavaScript 创造多线程环境，允许主线程创建 Worker 线程，将一些任务分配给后者运行。在主线程运行的同时，Worker 线程在后台运行，两者互不干扰。等到 Worker 线程完成计算任务，再把结果返回给主线程。这样的好处是，一些计算密集型或高延迟的任务可以交由 Worker 线程执行，主线程（通常负责 UI 交互）能够保持流畅，不会被阻塞或拖慢。

Worker 线程一旦新建成功，就会始终运行，不会被主线程上的活动（比如用户点击按钮、提交表单）打断。这样有利于随时响应主线程的通信。但是，这也造成了 Worker 比较耗费资源，不应该过度使用，而且一旦使用完毕，就应该关闭。

Web Worker 有以下几个使用注意点。

（1）**同源限制**

分配给 Worker 线程运行的脚本文件，必须与主线程的脚本文件同源。

（2）**DOM 限制**

Worker 线程所在的全局对象，与主线程不一样，无法读取主线程所在网页的 DOM 对象，也无法使用`document`、`window`、`parent`这些对象。但是，Worker 线程可以使用`navigator`对象和`location`对象。

（3）**全局对象限制**

Worker 的全局对象`WorkerGlobalScope`，不同于网页的全局对象`Window`，很多接口拿不到。比如，理论上 Worker 线程不能使用`console.log`，因为标准里面没有提到 Worker 的全局对象存在`console`接口，只定义了`Navigator`接口和`Location`接口。不过，浏览器实际上支持 Worker 线程使用`console.log`，保险的做法还是不使用这个方法。

（4）**通信联系**

Worker 线程和主线程不在同一个上下文环境，它们不能直接通信，必须通过消息完成。

（5）**脚本限制**

Worker 线程不能执行`alert()`方法和`confirm()`方法，但可以使用 XMLHttpRequest 对象发出 AJAX 请求。

（6）**文件限制**

Worker 线程无法读取本地文件，即不能打开本机的文件系统（`file://`），它所加载的脚本，必须来自网络。

基本用法

主线程

主线程采用`new`命令，调用`Worker()`构造函数，新建一个 Worker 线程。

```javascript
var worker = new Worker('work.js');
```

`Worker()`构造函数的参数是一个脚本文件，该文件就是 Worker 线程所要执行的任务。由于 Worker 不能读取本地文件，所以这个脚本必须来自网络。如果下载没有成功（比如404错误），Worker 就会默默地失败。

然后，主线程调用`worker.postMessage()`方法，向 Worker 发消息。

```javascript
worker.postMessage('Hello World');
worker.postMessage({method: 'echo', args: ['Work']});
```

`worker.postMessage()`方法的参数，就是主线程传给 Worker 的数据。它可以是各种数据类型，包括二进制数据。

接着，主线程通过`worker.onmessage`指定监听函数，接收子线程发回来的消息。

```javascript
worker.onmessage = function (event) {
  doSomething(event.data);
}

function doSomething() {
  // 执行任务
  worker.postMessage('Work done!');
}
```

上面代码中，事件对象的`data`属性可以获取 Worker 发来的数据。

Worker 完成任务以后，主线程就可以把它关掉。

```javascript
worker.terminate();
```

Worker 线程

Worker 线程内部需要有一个监听函数，监听`message`事件。

```javascript
self.addEventListener('message', function (e) {
  self.postMessage('You said: ' + e.data);
}, false);
```

上面代码中，`self`代表子线程自身，即子线程的全局对象。因此，等同于下面两种写法。

```javascript
// 写法一
this.addEventListener('message', function (e) {
  this.postMessage('You said: ' + e.data);
}, false);

// 写法二
addEventListener('message', function (e) {
  postMessage('You said: ' + e.data);
}, false);
```

除了使用`self.addEventListener()`指定监听函数，也可以使用`self.onmessage`指定。监听函数的参数是一个事件对象，它的`data`属性包含主线程发来的数据。`self.postMessage()`方法用来向主线程发送消息。

根据主线程发来的数据，Worker 线程可以调用不同的方法，下面是一个例子。

```javascript
self.addEventListener('message', function (e) {
  var data = e.data;
  switch (data.cmd) {
    case 'start':
      self.postMessage('WORKER STARTED: ' + data.msg);
      break;
    case 'stop':
      self.postMessage('WORKER STOPPED: ' + data.msg);
      self.close(); // Terminates the worker.
      break;
    default:
      self.postMessage('Unknown command: ' + data.msg);
  };
```

```
}, false);
```

上面代码中，`self.close()`用于在 Worker 内部关闭自身。

Worker 加载脚本

Worker 内部如果要加载其他脚本，有一个专门的方法`importScripts()`。

```javascript
importScripts('script1.js');
```

该方法可以同时加载多个脚本。

```javascript
importScripts('script1.js', 'script2.js');
```

错误处理

主线程可以监听 Worker 是否发生错误。如果发生错误，Worker 会触发主线程的`error`事件。

```javascript
worker.onerror(function (event) {
  console.log([
    'ERROR: Line ', event.lineno, ' in ', event.filename, ': ', event.message
  ].join(''));
});

// 或者
worker.addEventListener('error', function (event) {
  // ...
});
```

Worker 内部也可以监听`error`事件。

关闭 Worker

使用完毕，为了节省系统资源，必须关闭 Worker。

```javascript
// 主线程
worker.terminate();

// Worker 线程
self.close();
```

数据通信

前面说过，主线程与 Worker 之间的通信内容，可以是文本，也可以是对象。需要注意的是，这种通信是拷贝关系，即是传值而不是传址，Worker 对通信内容的修改，不会影响到主线程。事实上，浏览器内部的运行机制是，先将通信内容串行化，然后把串行化后的字符串发给 Worker，后者再将它还原。

主线程与 Worker 之间也可以交换二进制数据，比如 File、Blob、ArrayBuffer 等类型，也可以在线程之间发送。下面是一个例子。

```javascript
// 主线程
var uInt8Array = new Uint8Array(new ArrayBuffer(10));
for (var i = 0; i < uInt8Array.length; ++i) {
  uInt8Array[i] = i * 2; // [0, 2, 4, 6, 8,...]
}
worker.postMessage(uInt8Array);

// Worker 线程
self.onmessage = function (e) {
  var uInt8Array = e.data;
  postMessage('Inside worker.js: uInt8Array.toString() = ' + uInt8Array.toString());
  postMessage('Inside worker.js: uInt8Array.byteLength = ' + uInt8Array.byteLength);
};
```

但是，拷贝方式发送二进制数据，会造成性能问题。比如，主线程向 Worker 发送一个 500MB 文件，默认情况下浏览器会生成一个原文件的拷贝。为了解决这个问题，JavaScript 允许主线程把二进制数据直接转移给子线程，但是一旦转移，主线程就无法再使用这些二进制数据了，这是为了防止出现多个线程同时修改数据的麻烦局面。这种转移数据的方法，叫做[Transferable Objects](http://www.w3.org/html/wg/drafts/html/master/infrastructure.html#transferable-objects)。这使得主线程可以快速把数据交给 Worker，对于影像处理、声音处理、3D 运算等就非常方便了，不会产生性能负担。

如果要直接转移数据的控制权，就要使用下面的写法。

```javascript
// Transferable Objects 格式
worker.postMessage(arrayBuffer, [arrayBuffer]);

// 例子
var ab = new ArrayBuffer(1);
worker.postMessage(ab, [ab]);
```

同页面的 Web Worker

通常情况下，Worker 载入的是一个单独的 JavaScript 脚本文件，但是也可以载入与主线程在同一个网页的代码。

```html
<!DOCTYPE html>
  <body>
    <script id="worker" type="app/worker">
      addEventListener('message', function () {
        postMessage('some message');
      }, false);
    </script>
  </body>
</html>
```

上面是一段嵌入网页的脚本，注意必须指定`<script>`标签的`type`属性是一个浏览器不认识的值，上例是`app/worker`。

然后，读取这一段嵌入页面的脚本，用 Worker 来处理。

```javascript
var blob = new Blob([document.querySelector('#worker').textContent]);
var url = window.URL.createObjectURL(blob);
var worker = new Worker(url);

worker.onmessage = function (e) {
  // e.data === 'some message'
};
```

上面代码中，先将嵌入网页的脚本代码，转成一个二进制对象，然后为这个二进制对象生成 URL，再让 Worker 加载这个 URL。这样就做到了，主线程和 Worker 的代码都在同一个网页上面。

实例：Worker 线程完成轮询

有时，浏览器需要轮询服务器状态，以便第一时间得知状态改变。这个工作可以放在 Worker 里面。

```javascript
function createWorker(f) {
  var blob = new Blob(['(' + f.toString() + ')()']);
  var url = window.URL.createObjectURL(blob);
  var worker = new Worker(url);
  return worker;
}

var pollingWorker = createWorker(function (e) {
  var cache;
```

```
  function compare(new, old) { ... };

  setInterval(function () {
    fetch('/my-api-endpoint').then(function (res) {
      var data = res.json();

      if (!compare(data, cache)) {
        cache = data;
        self.postMessage(data);
      }
    })
  }, 1000)
});

pollingWorker.onmessage = function () {
  // render data
}

pollingWorker.postMessage('init');
```

上面代码中，Worker 每秒钟轮询一次数据，然后跟缓存做比较。如果不一致，就说明服务端有了新的变化，因此就要通知主线程。

实例： Worker 新建 Worker

Worker 线程内部还能再新建 Worker 线程（目前只有 Firefox 浏览器支持）。下面的例子是将一个计算密集的任务，分配到10个 Worker。

主线程代码如下。

```javascript
var worker = new Worker('worker.js');
worker.onmessage = function (event) {
  document.getElementById('result').textContent = event.data;
};
```

Worker 线程代码如下。

```javascript
// worker.js

// settings
var num_workers = 10;
var items_per_worker = 1000000;

// start the workers
var result = 0;
var pending_workers = num_workers;
```

```
for (var i = 0; i < num_workers; i += 1) {
  var worker = new Worker('core.js');
  worker.postMessage(i * items_per_worker);
  worker.postMessage((i + 1) * items_per_worker);
  worker.onmessage = storeResult;
}

// handle the results
function storeResult(event) {
  result += event.data;
  pending_workers -= 1;
  if (pending_workers <= 0)
    postMessage(result); // finished!
}
```

上面代码中，Worker 线程内部新建了10个 Worker 线程，并且依次向这10个 Worker 发送消息，告知了计算的起点和终点。计算任务脚本的代码如下。

```javascript
// core.js
var start;
onmessage = getStart;
function getStart(event) {
  start = event.data;
  onmessage = getEnd;
}

var end;
function getEnd(event) {
  end = event.data;
  onmessage = null;
  work();
}

function work() {
  var result = 0;
  for (var i = start; i < end; i += 1) {
    // perform some complex calculation here
    result += 1;
  }
  postMessage(result);
  close();
}
```

API

主线程

浏览器原生提供`Worker()`构造函数，用来供主线程生成 Worker 线程。

```javascript
var myWorker = new Worker(jsUrl, options);
```

`Worker()`构造函数，可以接受两个参数。第一个参数是脚本的网址（必须遵守同源政策），该参数是必需的，且只能加载 JS 脚本，否则会报错。第二个参数是配置对象，该对象可选。它的一个作用就是指定 Worker 的名称，用来区分多个 Worker 线程。

```javascript
// 主线程
var myWorker = new Worker('worker.js', { name : 'myWorker' });

// Worker 线程
self.name // myWorker
```

`Worker()`构造函数返回一个 Worker 线程对象，用来供主线程操作 Worker。Worker 线程对象的属性和方法如下。

- Worker.onerror：指定 error 事件的监听函数。
- Worker.onmessage：指定 message 事件的监听函数，发送过来的数据在`Event.data`属性中。
- Worker.onmessageerror：指定 messageerror 事件的监听函数。发送的数据无法序列化成字符串时，会触发这个事件。
- Worker.postMessage()：向 Worker 线程发送消息。
- Worker.terminate()：立即终止 Worker 线程。

Worker 线程

Web Worker 有自己的全局对象，不是主线程的`window`，而是一个专门为 Worker 定制的全局对象。因此定义在`window`上面的对象和方法不是全部都可以使用。

Worker 线程有一些自己的全局属性和方法。

- self.name： Worker 的名字。该属性只读，由构造函数指定。
- self.onmessage：指定`message`事件的监听函数。
- self.onmessageerror：指定 messageerror 事件的监听函数。发送的数据无法序列化成字符串时，会触发这个事件。
- self.close()：关闭 Worker 线程。
- self.postMessage()：向产生这个 Worker 线程发送消息。
- self.importScripts()：加载 JS 脚本。

（完）