对于 web Worker，你了解多少？

众所周知，Javascript 是运行在单线程环境中，也就是说无法同时运行多个脚本。假设用户点击一个按钮，触发了一段用于计算的 Javascript 代码，那么在这段代码执行完毕之前，页面是无法响应用户操作的。但是，如果将这段代码交给 Web Worker 去运行的话，那么情况就不一样了：浏览器会在后台启动一个独立的 worker 线程来专门负责这段代码的运行，因此，页面在这段 Javascript 代码运行期间依然可以响应用户的其他操作。

Web Worker 是 HTML5 标准的一部分，这一规范定义了一套 API，它允许一段 JavaScript 程序运行在主线程之外的另外一个线程中。
值得注意的是， Web Worker 规范中定义了两类工作线程，分别是专用线程 Dedicated Worker 和共享线程 Shared Worker，其中，Dedicated Worker 只能为一个页面所使用，而 Shared Worker 则可以被多个页面所共享。

用途

Web Worker 的意义在于可以将一些耗时的数据处理操作从主线程中剥离，使主线程更加专注于页面渲染和交互。

- 懒加载
- 文本分析
- 流媒体数据处理
- canvas 图形绘制
- 图像处理
- …

需要注意的点

- 有同源限制
- 无法访问 DOM 节点
- 运行在另一个上下文中，无法使用 Window 对象
- Web Worker 的运行不会影响主线程，但与主线程交互时仍受到主线程单线程的瓶颈制约。换言之，如果 Worker 线程频繁与主线程进行交互，主线程由于需要处理交互，仍有可能使页面发生阻塞
- 共享线程可以被多个浏览上下文（Browsing context）调用，但所有这些浏览上下文必须同源（相同的协议，主机和端口号）

线程创建

专用线程由 Worker()方法创建，可以接收两个参数，第一个参数是必填的脚本的位置，第二个参数是可选的配置对象，可以指定 type、credentials、name 三个属性。

```
var worker = new Worker('worker.js')
// var worker = new Worker('worker.js', { name: 'dedicatedWorker'})
```

共享线程使用 SharedWorker() 方法创建，同样支持两个参数，用法与 Worker() 一致。

```
var sharedWorker = new SharedWorker('shared-worker.js')
```

值得注意的是，因为 Web Worker 有同源限制，所以在本地调试的时候也需要通过启动本地服务器的方式访问，使用 file:// 协议直接打开的话将会抛出异常。

数据传递

Worker 线程和主线程都通过 postMessage() 方法发送消息，通过 onmessage 事件接收消息。在这个过程中数据并不是被共享的，而是被复制的。值得注意的是 Error 和 Function 对象不能被结构化克隆算法复制，如果尝试这么做的话会导致抛出 DATA_CLONE_ERR 的异常。另外，postMessage() 一次只能发送一个对象， 如果需要发送多个参数可以将参数包装为数组或对象再进行传递。

关于 postMessage() 和结构化克隆算法（The structured clone algorithm）将在本文最后进行阐述。

下面是专用线程数据传递的示例。

```
// 主线程
var worker = new Worker('worker.js')
worker.postMessage([10, 24])
worker.onmessage = function(e) {
    console.log(e.data)
}

// Worker 线程
onmessage = function (e) {
    if (e.data.length > 1) {
        postMessage(e.data[1] - e.data[0])
    }
}
```

在 Worker 线程中，self 和 this 都代表子线程的全局对象。对于监听 message 事件，以下的四种写法是等同的。

```
// 写法 1
self.addEventListener('message', function (e) {
    // ...
})

// 写法 2
this.addEventListener('message', function (e) {
    // ...
})

// 写法 3
addEventListener('message', function (e) {
    // ...
})

// 写法 4
onmessage = function (e) {
    // ...
}
```

主线程通过 MessagePort 访问专用线程和共享线程。专用线程的 port 会在线程创建时自动设置，并且不会暴露出来。与专用线程不同的是，共享线程在传递消息之前，端口必须处于打开状态。MDN 上的 MessagePort 关于 start() 方法的描述是：

> Starts the sending of messages queued on the port (only needed when using EventTarget.addEventListener; it is implied when using MessagePort.onmessage.)

这句话经过试验，可以理解为 start() 方法是与 addEventListener 配套使用的。如果我们选择 onmessage 进行事件监听，那么将隐含调用 start() 方法。

```
// 主线程
var sharedWorker = new SharedWorker('shared-worker.js')
sharedWorker.port.onmessage = function(e) {
    // 业务逻辑
}
var sharedWorker = new SharedWorker('shared-worker.js')
sharedWorker.port.addEventListener('message', function(e) {
    // 业务逻辑
}, false)
sharedWorker.port.start() // 需要显式打开
```

在传递消息时，postMessage() 方法和 onmessage 事件必须通过端口对象调用。另外，在 Worker 线程中，需要使用 onconnect 事件监听端口的变化，并使用端口的消息处理函数进行响应。

```
// 主线程
sharedWorker.port.postMessage([10, 24])
sharedWorker.port.onmessage = function (e) {
    console.log(e.data)
}

// Worker 线程
onconnect = function (e) {
    let port = e.ports[0]

    port.onmessage = function (e) {
        if (e.data.length > 1) {
            port.postMessage(e.data[1] - e.data[0])
        }
    }
}
```

关闭 Worker

可以在主线程中使用 terminate() 方法或在 Worker 线程中使用 close() 方法关闭 worker。这两种方法是等效的，但比较推荐的用法是使用 close()，防止意外关闭正在运行的 Worker 线程。Worker 线程一旦关闭 Worker 后 Worker 将不再响应。

```
// 主线程
worker.terminate()

// Dedicated Worker 线程中
self.close()

// Shared Worker 线程中
self.port.close()
```

错误处理

可以通过在主线程或 Worker 线程中设置 onerror 和 onmessageerror 的回调函数对错误进行处理。其中，onerror 在 Worker 的 error 事件触发并冒泡时执行，onmessageerror 在 Worker 收到的消息不能进行反序列化时触发(本人经过尝试没有办法触发 onmessageerror 事件，如果在 worker 线程使用

postMessage 方法传递一个 Error 或 Function 对象会因为无法序列化优先被 onerror 方法捕获，而根本不会进入反序列化的过程)。

```
// 主线程
worker.onerror = function () {
    // ...
}

// 主线程使用专用线程
worker.onmessageerror = function () {
    // ...
}

// 主线程使用共享线程
worker.port.onmessageerror = function () {
    // ...
}

// worker 线程
onerror = function () {

}
```

加载外部脚本

Web Worker 提供了 importScripts() 方法，能够将外部脚本文件加载到 Worker 中。

```
importScripts('script1.js')
importScripts('script2.js')

// 以上写法等价于
importScripts('script1.js', 'script2.js')
```

子线程

Worker 可以生成子 Worker，但有两点需要注意。

- 子 Worker 必须与父网页同源
- 子 Worker 中的 URI 相对于父 Worker 所在的位置进行解析

嵌入式 Worker

目前没有一类标签可以使 Worker 的代码像 <script> 元素一样嵌入网页中，但我们可以通过 Blob() 将页面中的 Worker 代码进行解析。

```
<script id="worker" type="javascript/worker">
// 这段代码不会被 JS 引擎直接解析，因为类型是 'javascript/worker'

// 在这里写 Worker 线程的逻辑
</script>
<script>
    var workerScript = document.querySelector('#worker').textContent
    var blob = new Blob(workerScript, {type: "text/javascript"})
    var worker = new Worker(window.URL.createObjectURL(blob))
</script>
```

关于 postMessage

Web Worker 中，Worker 线程和主线程之间使用结构化克隆算法（The structured clone algorithm）进行数据通信。结构化克隆算法是一种通过递归输入对象构建克隆的算法，算法通过保存之前访问过的引用的映射，避免无限遍历循环。这一过程可以理解为，在发送方使用类似 JSON.stringfy() 的方法将参数序列化，在接收方采用类似 JSON.parse() 的方法反序列化。

但是，一次数据传输就需要同时经过序列化和反序列化，如果数据量大的话，这个过程本身也可能造成性能问题。因此， Worker 中提出了 Transferable Objects 的概念，当数据量较大时，我们可以选择在将主线程中的数据直接移交给 Worker 线程。值得注意的是，这种转移是彻底的，一旦数据成功转移，主线程将不能访问该数据。这个移交的过程仍然通过 postMessage 进行传递。

```
postMessage(message, transferList)
```

例如，传递一个 ArrayBuffer 对象

```
let aBuffer = new ArrayBuffer(1)
worker.postMessage({ data: aBuffer }, [aBuffer])
```

上下文

Worker 工作在一个 WorkerGlobalDataScope 的上下文中。每一个 WorkerGlobalDataScope 对象都有不同的 event loop。这个 event loop 没有关联浏览器上下文（browsing context），它的任务队列也只有事件（events）、回调（callbacks）和联网的活动（networking activity）。

每一个 WorkerGlobalDataScope 都有一个 closing 标志，当这个标志设为 true 时，任务队列将丢弃之后试图加入任务队列的任务，队列中已经存在的任务不受影响（除非另有指定）。同时，定时器将停止工作，所有挂起（pending）的后台任务将会被删除。

Worker 中可以使用的函数和类

由于 Worker 工作的上下文不同于普通的浏览器上下文，因此不能访问 window 以及 window 相关的 API，也不能直接操作 DOM。Worker 中提供了 WorkerNavigator 和 WorkerLocation 接口，它们分别是 window 中 Navigator 和 Location 的子集。除此之外，Worker 还提供了涉及时间、存储、网络、绘图等多个种类的接口，以下列举了其中的一部分，更多的接口可以参考文档。