如此例所示，顶级脚本和工作者线程中的 console 对象都将写入浏览器控制台，这对于调试非常有用。因为工作者线程具有不可忽略的启动延迟，所以即使 Worker 对象存在，工作者线程的日志也会在主线程的日志之后打印出来。

> **注意**　这里两个独立的 JavaScript 线程都在向一个 console 对象发消息，该对象随后将消息序列化并在浏览器控制台打印出来。浏览器从两个不同的 JavaScript 线程收到消息，并按照自己认为合适的顺序输出这些消息。为此，在多线程应用程序中使用日志确定操作顺序时必须要当心。

DedicatedWorkerGlobalScope 在 WorkerGlobalScope 基础上增加了以下属性和方法。

- name：可以提供给 Worker 构造函数的一个可选的字符串标识符。
- postMessage()：与 worker.postMessage()对应的方法，用于从工作者线程内部向父上下文发送消息。
- close()：与 worker.terminate()对应的方法，用于立即终止工作者线程。没有为工作者线程提供清理的机会，脚本会突然停止。
- importScripts()：用于向工作者线程中导入任意数量的脚本。

27.2.2　专用工作者线程与隐式 MessagePorts

专用工作者线程的 Worker 对象和 DedicatedWorkerGlobalScope 与 MessagePorts 有一些相同接口处理程序和方法：onmessage、onmessageerror、close()和 postMessage()。这不是偶然的，因为专用工作者线程隐式使用了 MessagePorts 在两个上下文之间通信。

父上下文中的 Worker 对象和 DedicatedWorkerGlobalScope 实际上融合了 MessagePort，并在自己的接口中分别暴露了相应的处理程序和方法。换句话说，消息还是通过 MessagePort 发送，只是没有直接使用 MessagePort 而已。

也有不一致的地方，比如 start()和 close()约定。专用工作者线程会自动发送排队的消息，因此 start()也就没有必要了。另外，close()在专用工作者线程的上下文中没有意义，因为这样关闭 MessagePort 会使工作者线程孤立。因此，在工作者线程内部调用 close()（或在外部调用 terminate()）不仅会关闭 MessagePort，也会终止线程。

27.2.3　专用工作者线程的生命周期

调用 Worker()构造函数是一个专用工作者线程生命的起点。调用之后，它会初始化对工作者线程脚本的请求，并把 Worker 对象返回给父上下文。虽然父上下文中可以立即使用这个 Worker 对象，但与之关联的工作者线程可能还没有创建，因为存在请求脚本的网格延迟和初始化延迟。

一般来说，专用工作者线程可以非正式区分为处于下列三个状态：**初始化**（initializing）、**活动**（active）和**终止**（terminated）。这几个状态对其他上下文是不可见的。虽然 Worker 对象可能会存在于父上下文中，但也无法通过它确定工作者线程当前是处理初始化、活动还是终止状态。换句话说，与活动的专用工作者线程关联的 Worker 对象和与终止的专用工作者线程关联的 Worker 对象无法分别。

初始化时，虽然工作者线程脚本尚未执行，但可以先把要发送给工作者线程的消息加入队列。这些消息会等待工作者线程的状态变为活动，再把消息添加到它的消息队列。下面的代码演示了这个过程。

initializingWorker.js

```
self.addEventListener('message', ({data}) => console.log(data));
```

main.js

```
const worker = new Worker('./initializingWorker.js');

// Worker 可能仍处于初始化状态
// 但 postMessage()数据可以正常处理
worker.postMessage('foo');
worker.postMessage('bar');
worker.postMessage('baz');

// foo
// bar
// baz
```

创建之后，专用工作者线程就会伴随页面的整个生命期而存在，除非自我终止（self.close()）或通过外部终止（worker.terminate()）。即使线程脚本已运行完成，线程的环境仍会存在。只要工作者线程仍存在，与之关联的 Worker 对象就不会被当成垃圾收集掉。

自我终止和外部终止最终都会执行相同的工作者线程终止例程。来看下面的例子，其中工作者线程在发送两条消息中间执行了自我终止：

closeWorker.js

```
self.postMessage('foo');
self.close();
self.postMessage('bar');
setTimeout(() => self.postMessage('baz'), 0);
```

main.js

```
const worker = new Worker('./closeWorker.js');
worker.onmessage = ({data}) => console.log(data);

// foo
// bar
```

虽然调用了 close()，但显然工作者线程的执行并没有立即终止。close()在这里会通知工作者线程取消事件循环中的所有任务，并阻止继续添加新任务。这也是为什么"baz"没有打印出来的原因。工作者线程不需要执行同步停止，因此在父上下文的事件循环中处理的"bar"仍会打印出来。

下面来看外部终止的例子。

terminateWorker.js

```
self.onmessage = ({data}) => console.log(data);
```

main.js

```
const worker = new Worker('./terminateWorker.js');

// 给 1000 毫秒让工作者线程初始化
setTimeout(() => {
  worker.postMessage('foo');
  worker.terminate();
  worker.postMessage('bar');
  setTimeout(() => worker.postMessage('baz'), 0);
}, 1000);

// foo
```

这里，外部先给工作者线程发送了带"foo"的 postMessage，这条消息可以在外部终止之前处理。一旦调用了 terminate()，工作者线程的消息队列就会被清理并锁住，这也是只是打印"foo"的原因。

> **注意**　close()和 terminate()是幂等操作，多次调用没有问题。这两个方法仅仅是将 Worker 标记为 teardown，因此多次调用不会有不好的影响。

在整个生命周期中，一个专用工作者线程只会关联一个网页（Web 工作者线程规范称其为一个**文档**）。除非明确终止，否则只要关联文档存在，专用工作者线程就会存在。如果浏览器离开网页（通过导航或关闭标签页或关闭窗口），它会将与其关联的工作者线程标记为终止，它们的执行也会立即停止。

27.2.4　配置 Worker 选项

Worker()构造函数允许将可选的配置对象作为第二个参数。该配置对象支持下列属性。

- name：可以在工作者线程中通过 self.name 读取到的字符串标识符。
- type：表示加载脚本的运行方式，可以是"classic"或"module"。"classic"将脚本作为常规脚本来执行，"module"将脚本作为模块来执行。
- credentials：在 type 为"module"时，指定如何获取与传输凭证数据相关的工作者线程模块脚本。值可以是"omit"、"same-orign"或"include"。这些选项与 fetch()的凭证选项相同。在 type 为"classic"时，默认为"omit"。

> **注意**　有的现代浏览器还不完全支持模块工作者线程或可能需要修改标志才能支持。

27.2.5　在 JavaScript 行内创建工作者线程

工作者线程需要基于脚本文件来创建，但这并不意味着该脚本必须是远程资源。专用工作者线程也可以通过 Blob 对象 URL 在行内脚本创建。这样可以更快速地初始化工作者线程，因为没有网络延迟。

下面展示了一个在行内创建工作者线程的例子。

```
// 创建要执行的 JavaScript 代码字符串
const workerScript = `
  self.onmessage = ({data}) => console.log(data);
`;

// 基于脚本字符串生成 Blob 对象
const workerScriptBlob = new Blob([workerScript]);

// 基于 Blob 实例创建对象 URL
const workerScriptBlobUrl = URL.createObjectURL(workerScriptBlob);

// 基于对象 URL 创建专用工作者线程
const worker = new Worker(workerScriptBlobUrl);

worker.postMessage('blob worker script');
// blob worker script
```

在这个例子中，通过脚本字符串创建了 Blob，然后又通过 Blob 创建了对象 URL，最后把对象 URL 传给了 Worker()构造函数。该构造函数同样创建了专用工作者线程。

如果把所有代码写在一块，可以浓缩为这样：

```
const worker = new Worker(URL.createObjectURL(new Blob([`self.onmessage =
({data}) => console.log(data);`])));

worker.postMessage('blob worker script');
// blob worker script
```

工作者线程也可以利用函数序列化来初始化行内脚本。这是因为函数的 `toString()`方法返回函数代码的字符串，而函数可以在父上下文中定义但在子上下文中执行。来看下面这个简单的例子：

```
function fibonacci(n) {
  return n < 1 ? 0
      : n <= 2 ? 1
      : fibonacci(n - 1) + fibonacci(n - 2);
}

const workerScript = `
  self.postMessage(
    (${fibonacci.toString()})(9)
  );
`;

const worker = new Worker(URL.createObjectURL(new Blob([workerScript])));

worker.onmessage = ({data}) => console.log(data);

// 34
```

这里有意使用了斐波那契数列的实现，将其序列化之后传给了工作者线程。该函数作为 IIFE 调用并传递参数，结果则被发送回主线程。虽然计算斐波那契数列比较耗时，但所有计算都会委托到工作者线程，因此并不会影响父上下文的性能。

> **注意** 像这样序列化函数有个前提，就是函数体内不能使用通过闭包获得的引用，也包括全局变量，比如 window，因为这些引用在工作者线程中执行时会出错。

27.2.6 在工作者线程中动态执行脚本

工作者线程中的脚本并非铁板一块，而是可以使用 `importScripts()`方法通过编程方式加载和执行任意脚本。该方法可用于全局 `Worker` 对象。这个方法会加载脚本并按照加载顺序同步执行。比如，下面的例子加载并执行了两个脚本：

main.js

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

// importing scripts
// scriptA executes
// scriptB executes
// scripts imported
```

scriptA.js

```
console.log('scriptA executes');
```

scriptB.js

```
console.log('scriptB executes');
```

worker.js

```
console.log('importing scripts');

importScripts('./scriptA.js');
importScripts('./scriptB.js');

console.log('scripts imported');
```

importScripts()方法可以接收任意数量的脚本作为参数。浏览器下载它们的顺序没有限制，但执行则会严格按照它们在参数列表的顺序进行。因此，下面的代码与前面的效果一样：

```
console.log('importing scripts');

importScripts('./scriptA.js', './scriptB.js');

console.log('scripts imported');
```

脚本加载受到常规 CORS 的限制，但在工作者线程内部可以请求来自任何源的脚本。这里的脚本导入策略类似于使用生成的<script>标签动态加载脚本。在这种情况下，所有导入的脚本也会共享作用域。下面的代码演示了这个事实：

main.js

```
const worker = new Worker('./worker.js', {name: 'foo'});

// importing scripts in foo with bar
// scriptA executes in foo with bar
// scriptB executes in foo with bar
// scripts imported
```

scriptA.js

```
console.log(`scriptA executes in ${self.name} with ${globalToken}`);
```

scriptB.js

```
console.log(`scriptB executes in ${self.name} with ${globalToken}`);
```

worker.js

```
const globalToken = 'bar';

console.log(`importing scripts in ${self.name} with ${globalToken}`);

importScripts('./scriptA.js', './scriptB.js');

console.log('scripts imported');
```

27.2.7　委托任务到子工作者线程

有时候可能需要在工作者线程中再创建子工作者线程。在有多个 CPU 核心的时候，使用多个子工作者线程可以实现并行计算。使用多个子工作者线程前要考虑周全，确保并行计算的投入确实能够得到收益，毕竟同时运行多个子线程会有很大计算成本。

除了路径解析不同，创建子工作者线程与创建普通工作者线程是一样的。子工作者线程的脚本路径根据父工作者线程而不是相对于网页来解析。来看下面的例子（注意额外的 js 目录）：

main.js

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

// worker
// subworker
```

js/worker.js

```
console.log('worker');

const worker = new Worker('./subworker.js');
```

js/subworker.js

```
console.log('subworker');
```

> **注意** 顶级工作者线程的脚本和子工作者线程的脚本都必须从与主页相同的源加载。

27.2.8 处理工作者线程错误

如果工作者线程脚本抛出了错误，该工作者线程沙盒可以阻止它打断父线程的执行。如下例所示，其中的 `try/catch` 块不会捕获到错误：

main.js

```
try {
  const worker = new Worker('./worker.js');
  console.log('no error');
} catch(e) {
  console.log('caught error');
}

// no error
```

worker.js

```
throw Error('foo');
```

不过，相应的错误事件仍然会冒泡到工作者线程的全局上下文，因此可以通过在 `Worker` 对象上设置错误事件侦听器访问到。下面看这个例子：

main.js

```
const worker = new Worker('./worker.js');
worker.onerror = console.log;

// ErrorEvent {message: "Uncaught Error: foo"}
```

worker.js

```
throw Error('foo');
```

27.2.9 与专用工作者线程通信

与工作者线程的通信都是通过异步消息完成的，但这些消息可以有多种形式。

1. 使用 postMessage()

最简单也最常用的形式是使用 `postMessage()` 传递序列化的消息。下面来看一个计算阶乘的例子：

factorialWorker.js

```
function factorial(n) {
  let result = 1;
  while(n) { result *= n--; }
  return result;
}

self.onmessage = ({data}) => {
  self.postMessage(`${data}! = ${factorial(data)}`);
};
```

main.js

```
const factorialWorker = new Worker('./factorialWorker.js');

factorialWorker.onmessage = ({data}) => console.log(data);

factorialWorker.postMessage(5);
factorialWorker.postMessage(7);
factorialWorker.postMessage(10);

// 5! = 120
// 7! = 5040
// 10! = 3628800
```

对于传递简单的消息，使用 postMessage()在主线程和工作者线程之间传递消息，与在两个窗口间传递消息非常像。主要区别是没有 targetOrigin 的限制，该限制是针对 Window.prototype.postMessage 的，对 WorkerGlobalScope.prototype.postMessage 或 Worker.prototype.postMessage 没有影响。这样约定的原因很简单：工作者线程脚本的源被限制为主页的源，因此没有必要再去过滤了。

2. 使用 MessageChannel

无论主线程还是工作者线程，通过 postMessage()进行通信涉及调用全局对象上的方法，并定义一个临时的传输协议。这个过程可以被 Channel Messaging API 取代，基于该 API 可以在两个上下文间明确建立通信渠道。

MessageChannel 实例有两个端口，分别代表两个通信端点。要让父页面和工作线程通过 MessageChannel 通信，需要把一个端口传到工作者线程中，如下所示：

worker.js

```
// 在监听器中存储全局 messagePort
let messagePort = null;

function factorial(n) {
  let result = 1;
  while(n) { result *= n--; }
  return result;
}

// 在全局对象上添加消息处理程序
self.onmessage = ({ports}) => {
  // 只设置一次端口
  if (!messagePort) {
    // 初始化消息发送端口，
    // 给变量赋值并重置监听器
```

```
    messagePort = ports[0];
    self.onmessage = null;

    // 在全局对象上设置消息处理程序
    messagePort.onmessage = ({data}) => {
      // 收到消息后发送数据
      messagePort.postMessage(`${data}! = ${factorial(data)}`);
    };
  }
};
```

main.js

```
const channel = new MessageChannel();
const factorialWorker = new Worker('./worker.js');

// 把`MessagePort`对象发送到工作者线程
// 工作者线程负责处理初始化信道
factorialWorker.postMessage(null, [channel.port1]);

// 通过信道实际发送数据
channel.port2.onmessage = ({data}) => console.log(data);

// 工作者线程通过信道响应
channel.port2.postMessage(5);

// 5! = 120
```

在这个例子中，父页面通过 postMessage 与工作者线程共享 MessagePort。使用数组语法是为了在两个上下文间传递可转移对象。本章稍后会介绍可转移对象（Transferable）。工作者线程维护着对该端口的引用，并使用它代替通过全局对象传递消息。当然，消息的格式也需要临时约定：工作者线程收到的第一条消息包含端口，后续的消息才是数据。

使用 MessageChannel 实例与父页面通信很大程度上是多余的。这是因为全局 postMessage() 方法本质上与 channel.postMessage()执行的是同样的操作（不考虑 MessageChannel 接口的其他特性）。MessageChannel 真正有用的地方是让两个工作者线程之间直接通信。这可以通过把端口传给另一个工作者线程实现。下面的例子把一个数组传给了一个工作者线程，这个线程又把它传另一个工作者线程，然后再传回主页：

main.js

```
const channel = new MessageChannel();
const workerA = new Worker('./worker.js');
const workerB = new Worker('./worker.js');

workerA.postMessage('workerA', [channel.port1]);
workerB.postMessage('workerB', [channel.port2]);

workerA.onmessage = ({data}) => console.log(data);
workerB.onmessage = ({data}) => console.log(data);

workerA.postMessage(['page']);

// ['page', 'workerA', 'workerB']

workerB.postMessage(['page'])

// ['page', 'workerB', 'workerA']
```

worker.js

```
let messagePort = null;
let contextIdentifier = null;

function addContextAndSend(data, destination) {
  // 添加标识符以标识当前工作者线程
  data.push(contextIdentifier);

  // 把数据发送到下一个目标
  destination.postMessage(data);
}

self.onmessage = ({data, ports}) => {
  // 如果消息里存在端口 (ports)
  // 则初始化工作者线程
  if (ports.length) {
    // 记录标识符
    contextIdentifier = data;

    // 获取 MessagePort
    messagePort = ports[0];

    // 添加处理程序把接收的数据
    // 发回到父页面
    messagePort.onmessage = ({data}) => {
      addContextAndSend(data, self);
    }
  } else {
    addContextAndSend(data, messagePort);
  }
};
```

在这个例子中，数组的每一段旅程都会添加一个字符串，标识自己到过哪里。数组从父页面发送到工作者线程，工作者线程会加上自己的上下文标识符。然后，数组又从一个工作者线程发送到另一个工作者线程。第二个线程又加上自己的上下文标识符，随即将数组发回主页，主页把数组打印出来。这个例子中的两个工作者线程使用了同一个脚本，因此要注意数组可以双向传递。

3. 使用 BroadcastChannel

同源脚本能够通过 BroadcastChannel 相互之间发送和接收消息。这种通道类型的设置比较简单，不需要像 MessageChannel 那样转移乱糟糟的端口。这可以通过以下方式实现：

main.js

```
const channel = new BroadcastChannel('worker_channel');
const worker = new Worker('./worker.js');

channel.onmessage = ({data}) => {
  console.log(`heard ${data} on page`);
}

setTimeout(() => channel.postMessage('foo'), 1000);

// heard foo in worker
// heard bar on page
```

worker.js

```
const channel = new BroadcastChannel('worker_channel');
```

```
channel.onmessage = ({data}) => {
  console.log(`heard ${data} in worker`);
  channel.postMessage('bar');
}
```

这里，页面在通过 BroadcastChannel 发送消息之前会先等 1 秒钟。因为这种信道没有端口所有权的概念，所以如果没有实体监听这个信道，广播的消息就不会有人处理。在这种情况下，如果没有 setTimeout()，则由于初始化工作者线程的延迟，就会导致消息已经发送了，但工作者线程上的消息处理程序还没有就位。

27.2.10 工作者线程数据传输

使用工作者线程时，经常需要为它们提供某种形式的数据负载。工作者线程是独立的上下文，因此在上下文之间传输数据就会产生消耗。在支持传统多线程模型的语言中，可以使用锁、互斥量，以及 volatile 变量。在 JavaScript 中，有三种在上下文间转移信息的方式：**结构化克隆算法**（structured clone algorithm）、**可转移对象**（transferable objects）和**共享数组缓冲区**（shared array buffers）。

1. 结构化克隆算法

结构化克隆算法可用于在两个独立上下文间共享数据。该算法由浏览器在后台实现，不能直接调用。

在通过 postMessage()传递对象时，浏览器会遍历该对象，并在目标上下文中生成它的一个副本。下列类型是结构化克隆算法支持的类型。

- 除 Symbol 之外的所有原始类型
- Boolean 对象
- String 对象
- BDate
- RegExp
- Blob
- File
- FileList
- ArrayBuffer
- ArrayBufferView
- ImageData
- Array
- Object
- Map
- Set

关于结构化克隆算法，有以下几点需要注意。

- 复制之后，源上下文中对该对象的修改，不会传播到目标上下文中的对象。
- 结构化克隆算法可以识别对象中包含的循环引用，不会无穷遍历对象。
- 克隆 Error 对象、Function 对象或 DOM 节点会抛出错误。
- 结构化克隆算法并不总是创建完全一致的副本。
- 对象属性描述符、获取方法和设置方法不会克隆，必要时会使用默认值。
- 原型链不会克隆。
- RegExp.prototype.lastIndex 属性不会克隆。