如上表所示，脚本执行次数、打开标签页数和运行的线程数是对等关系。下面再来看看这个简单的脚本，每次执行它都会创建或者连接到共享线程：

```
new SharedWorker('./sharedWorker.js');
```

下表列出了当三个包含此脚本的标签页按顺序打开和关闭时会发生什么。

事　件	结　果	事件发生后的线程数
标签页 1 执行 main.js	创建共享线程 1	1
标签页 2 执行 main.js	连接共享线程 1	1
标签页 3 执行 main.js	连接共享线程 1	1
标签页 1 关闭	断开与共享线程 1 的连接	1
标签页 2 关闭	断开与共享线程 1 的连接	1
标签页 3 关闭	断开与共享线程 1 的连接。没有连接了，因此终止共享线程 1	0

如上表所示，标签页 2 和标签页 3 再次调用 `new SharedWorker()`会连接到已有线程。随着连接的增加和移除，浏览器会记录连接总数。在连接数为 0 时，线程被终止。

关键在于，没有办法以编程方式终止共享线程。前面已经交代过，`SharedWorker` 对象上没有 `terminate()`方法。在共享线程端口（稍后讨论）上调用 `close()`时，只要还有一个端口连接到该线程就不会真的终止线程。

`SharedWorker` 的“连接”与关联 `MessagePort` 或 `MessageChannel` 的状态无关。只要建立了连接，浏览器会负责管理该连接。建立的连接会在页面的生命周期内持续存在，只有当页面销毁且没有连接时，浏览器才会终止共享线程。

27.3.3　连接到共享工作者线程

每次调用 `SharedWorker()`构造函数，无论是否创建了工作者线程，都会在共享线程内部触发 `connect` 事件。下面的例子演示了这一点，在循环中调用 `SharedWorker()`构造函数：

sharedWorker.js

```
let i = 0;
self.onconnect = () => console.log(`connected ${++i} times`);
```

main.js

```
for (let i = 0; i < 5; ++i) {
  new SharedWorker('./sharedWorker.js');
}

// connected 1 times
// connected 2 times
// connected 3 times
// connected 4 times
// connected 5 times
```

发生 `connect` 事件时，`SharedWorker()`构造函数会隐式创建 `MessageChannel` 实例，并把 `MessagePort` 实例的所有权唯一地转移给该 `SharedWorker` 的实例。这个 `MessagePort` 实例会保存在 `connect` 事件对象的 `ports` 数组中。一个连接事件只能代表一个连接，因此可以假定 `ports` 数组的长度等于 1。

图灵社区会员 aSINKz(1561821892@qq.com) 专享 尊重版权

下面的代码演示了访问事件对象的 ports 数组。这里使用了 Set 来保证只跟踪唯一的对象实例：

sharedWorker.js

```
const connectedPorts = new Set();

self.onconnect = ({ports}) => {
  connectedPorts.add(ports[0]);

  console.log(`${connectedPorts.size} unique connected ports`);
};
```

main.js

```
for (let i = 0; i < 5; ++i) {
  new SharedWorker('./sharedWorker.js');
}

// 1 unique connected ports
// 2 unique connected ports
// 3 unique connected ports
// 4 unique connected ports
// 5 unique connected ports
```

关键在于，共享线程与父上下文的启动和关闭不是对称的。每个新 SharedWorker 连接都会触发一个事件，但没有事件对应断开 SharedWorker 实例的连接（如页面关闭）。

在前面的例子中，随着与相同共享线程连接和断开连接的页面越来越多，connectedPorts 集合中会受到死端口的污染，没有办法识别它们。一个解决方案是在 beforeunload 事件即将销毁页面时，明确发送卸载消息，让共享线程有机会清除死端口。

27.4 服务工作者线程

服务工作者线程（service worker）是一种类似浏览器中代理服务器的线程，可以拦截外出请求和缓存响应。这可以让网页在没有网络连接的情况下正常使用，因为部分或全部页面可以从服务工作者线程缓存中提供服务。服务工作者线程也可以使用 Notifications API、Push API、Background Sync API 和 Channel Messaging API。

与共享工作者线程类似，来自一个域的多个页面共享一个服务工作者线程。不过，为了使用 Push API 等特性，服务工作者线程也可以在相关的标签页或浏览器关闭后继续等待到来的推送事件。

无论如何，对于大多数开发者而言，服务工作者线程在两个主要任务上最有用：充当网络请求的缓存层和启用推送通知。在这个意义上，服务工作者线程就是用于把网页变成像原生应用程序一样的工具。

> **注意** 服务工作者线程涉及的内容极其广泛，几乎可以单独写一本书。为了更好地理解这一话题，推荐有条件的读者学一下 Udacity 的课程“Offline Web Applications”。除此之外，也可以参考 Mozilla 维护的 Service Worker Cookbook 网站，其中包含了常见的服务工作者线程模式。

> **注意**　服务工作者线程的生命周期取决于打开的同源标签页（称为“客户端”）数量、页面是否发生导航，以及服务脚本是否改变（以及其他一些因素）。如果对服务工作者线程的生命周期认识不够，本节的一些例子可能会让人觉得出乎意料。27.4.5 节详细解释了服务工作者线程的生命周期。
>
> 另外，在调试服务工作者线程时，要谨慎使用浏览器的强制刷新功能（Ctrl+Shift+R）。强制刷新会强制浏览器忽略所有网络缓存，而服务工作者线程对大多数主流浏览器而言就是网络缓存。

27.4.1　服务工作者线程基础

作为一种工作者线程，服务工作者线程与专用工作者线程和共享工作者线程拥有很多共性。比如，在独立上下文中运行，只能通过异步消息通信。不过，服务工作者线程与专用工作者线程和共享工作者线程还是有很多本质区别的。

1. ServiceWorkerContainer

服务工作者线程与专用工作者线程或共享工作者线程的一个区别是没有全局构造函数。服务工作者线程是通过 `ServiceWorkerContainer` 来管理的，它的实例保存在 `navigator.serviceWorker` 属性中。该对象是个顶级接口，通过它可以让浏览器创建、更新、销毁或者与服务工作者线程交互。

```
console.log(navigator.serviceWorker);
// ServiceWorkerContainer { ... }
```

2. 创建服务工作者线程

与共享工作者线程类似，服务工作者线程同样是在还不存在时创建新实例，在存在时连接到已有实例。`ServiceWorkerContainer` 没有通过全局构造函数创建，而是暴露了 `register()`方法，该方法以与 `Worker()`或 `SharedWorker()`构造函数相同的方式传递脚本 URL：

emptyServiceWorker.js

```
// 空服务脚本
```

main.js

```
navigator.serviceWorker.register('./emptyServiceWorker.js');
```

`register()`方法返回一个期约，该期约解决为 `ServiceWorkerRegistration` 对象，或在注册失败时拒绝。

emptyServiceWorker.js

```
// 空服务脚本
```

main.js

```
// 注册成功，成功回调（解决）
navigator.serviceWorker.register('./emptyServiceWorker.js')
  .then(console.log, console.error);

// ServiceWorkerRegistration { ... }

// 使用不存在的文件注册，失败回调（拒绝）
navigator.serviceWorker.register('./doesNotExist.js')
  .then(console.log, console.error);
```

```
// TypeError: Failed to register a ServiceWorker:
// A bad HTTP response code (404) was received when fetching the script.
```

服务工作者线程对于何时注册是比较灵活的。在第一次调用 register()激活服务工作者线程后，后续在同一个页面使用相同 URL 对 register()的调用实际上什么也不会执行。此外，即使浏览器未全局支持服务工作者线程，服务工作者线程本身对页面也应该是不可见的。这是因为它的行为类似代理，就算有需要它处理的操作，也仅仅是发送常规的网络请求。

考虑到上述情况，注册服务工作者线程的一种非常常见的模式是基于特性检测，并在页面的 load 事件中操作。比如：

```
if ('serviceWorker' in navigator) {
  window.addEventListener('load', () => {
    navigator.serviceWorker.register('./serviceWorker.js');
  });
}
```

如果没有 load 事件这个门槛，服务工作者线程的注册就会与页面资源的加载重叠，进而拖慢初始页面渲染的过程。除非该服务工作者线程负责管理缓存（这样的话就需要尽早注册，比如使用本章稍后会讨论的 clients.claim()），否则等待 load 事件是个明智的选择，这样同样可以发挥服务工作者线程的价值。

3. 使用 ServiceWorkerContainer 对象

ServiceWorkerContainer 接口是浏览器对服务工作者线程生态的顶部封装。它为管理服务工作者线程状态和生命周期提供了便利。

ServiceWorkerContainer 始终可以在客户端上下文中访问：

```
console.log(navigator.serviceWorker);

// ServiceWorkerContainer { ... }
```

ServiceWorkerContainer 支持以下事件处理程序。

- oncontrollerchange：在 ServiceWorkerContainer 触发 controllerchange 事件时会调用指定的事件处理程序。
  - 此事件在获得新激活的 ServiceWorkerRegistration 时触发。
  - 此事件也可以使用 navigator.serviceWorker.addEventListener('controllerchange', handler)处理。
- onerror：在关联的服务工作者线程触发 ErrorEvent 错误事件时会调用指定的事件处理程序。
  - 此事件在关联的服务工作者线程内部抛出错误时触发。
  - 此事件也可以使用 navigator.serviceWorker.addEventListener('error', handler)处理。
- onmessage：在服务工作者线程触发 MessageEvent 事件时会调用指定的事件处理程序。
  - 此事件在服务脚本向父上下文发送消息时触发。
  - 此事件也可以使用 navigator.serviceWorker.addEventListener('message', handler)处理。

ServiceWorkerContainer 支持下列属性。

- ready：返回期约，解决为激活的 ServiceWorkerRegistration 对象。该期约不会拒绝。

- controller：返回与当前页面关联的激活的 ServiceWorker 对象，如果没有激活的服务工作者线程则返回 null。

ServiceWorkerContainer 支持下列方法。

- register()：使用接收的 url 和 options 对象创建或更新 ServiceWorkerRegistration。
- getRegistration()：返回期约，解决为与提供的作用域匹配的 ServiceWorkerRegistration 对象；如果没有匹配的服务工作者线程则返回 undefined。
- getRegistrations()：返回期约，解决为与 ServiceWorkerContainer 关联的 ServiceWorkerRegistration 对象的数组；如果没有关联的服务工作者线程则返回空数组。
- startMessage()：开始传送通过 Client.postMessage()派发的消息。

4. 使用 ServiceWorkerRegistration 对象

ServiceWorkerRegistration 对象表示注册成功的服务工作者线程。该对象可以在 register()返回的解决期约的处理程序中访问到。通过它的一些属性可以确定关联服务工作者线程的生命周期状态。

调用 navigator.serviceWorker.register()之后返回的期约会将注册成功的 ServiceWorkerRegistration 对象（注册对象）发送给处理函数。在同一页面使用同一 URL 多次调用该方法会返回相同的注册对象。

```
navigator.serviceWorker.register('./serviceWorker.js')
.then((registrationA) => {
  console.log(registrationA);

  navigator.serviceWorker.register('./serviceWorker2.js')
    .then((registrationB) => {
      console.log(registrationA === registrationB);
    });
});
```

ServiceWorkerRegistration 支持以下事件处理程序。

- onupdatefound：在服务工作者线程触发 updatefound 事件时会调用指定的事件处理程序。
  - 此事件会在服务工作者线程开始安装新版本时触发，表现为 ServiceWorkerRegistration.installing 收到一个新的服务工作者线程。
  - 此事件也可以使用 serv serviceWorkerRegistration.addEventListener('updatefound', handler)处理。

ServiceWorkerRegistration 支持以下通用属性。

- scope：返回服务工作者线程作用域的完整 URL 路径。该值源自接收服务脚本的路径和在 register()中提供的作用域。
- navigationPreload：返回与注册对象关联的 NavigationPreloadManager 实例。
- pushManager：返回与注册对象关联的 pushManager 实例。

ServiceWorkerRegistration 还支持以下属性，可用于判断服务工作者线程处于生命周期的什么阶段。

- installing：如果有则返回状态为 installing（安装）的服务工作者线程，否则为 null。
- waiting：如果有则返回状态为 waiting（等待）的服务工作者线程，否则为 null。
- active：如果有则返回状态 activating 或 active（活动）的服务工作者线程，否则为 null。

注意，这些属性都是服务工作者线程状态的一次性快照。这在大多数情况下是没有问题的，因为活

图灵社区会员 aSINKz(1561821892@qq.com) 专享 尊重版权

动状态的服务工作者线程在页面的生命周期内不会改变状态，除非强制这样做（比如调用 ServiceWorkerGlobalScope.skipWaiting()）。

ServiceWorkerRegistration 支持下列方法。

- getNotifications()：返回期约，解决为 Notification 对象的数组。
- showNotifications()：显示通知，可以配置 title 和 options 参数。
- update()：直接从服务器重新请求服务脚本，如果新脚本不同，则重新初始化。
- unregister()：取消服务工作者线程的注册。该方法会在服务工作者线程执行完再取消注册。

5. 使用 ServiceWorker 对象

ServiceWorker 对象可以通过两种方式获得：通过 ServiceWorkerContainer 对象的 controller 属性和通过 ServiceWorkerRegistration 的 active 属性。该对象继承 Worker 原型，因此包括其所有属性和方法，但没有 terminate()方法。

ServiceWorker 支持以下事件处理程序。

- onstatechange：ServiceWorker 发生 statechange 事件时会调用指定的事件处理程序。
  - 此事件会在 ServiceWorker.state 变化时发生。
  - 此事件也可以使用 serviceWorker.addEventListener('statechange', handler)处理。

ServiceWorker 支持以下属性。

- scriptURL：解析后注册服务工作者线程的 URL。例如，如果服务工作者线程是通过相对路径 './serviceWorker.js'创建的，且注册在 https://www.example.com 上，则 scriptURL 属性将返回"https://www.example.com/serviceWorker.js"。
- state：表示服务工作者线程状态的字符串，可能的值如下。
  - installing
  - installed
  - activating
  - activated
  - redundant

6. 服务工作者线程的安全限制

与其他工作者线程一样，服务工作者线程也受加载脚本对应源的常规限制（更多信息参见 27.2.1 节）。此外，由于服务工作者线程几乎可以任意修改和重定向网络请求，以及加载静态资源，服务工作者线程 API 只能在安全上下文（HTTPS）下使用。在非安全上下文（HTTP）中，navigator.serviceWorker 是 undefined。为方便开发，浏览器豁免了通过 localhost 或 127.0.0.1 在本地加载的页面的安全上下文规则。

> **注意** 可以通过 window.isSecureContext 确定当前上下文是否安全。

7. ServiceWorkerGlobalScope

在服务工作者线程内部，全局上下文是 ServiceWorkerGlobalScope 的实例。ServiceWorkerGlobalScope 继承自 WorkerGlobalScope，因此拥有它的所有属性和方法。服务工作者线程可以通过 self 关键字访问该全局上下文。

ServiceWorkerGlobalScope 通过以下属性和方法扩展了 WorkerGlobalScope。

- caches：返回服务工作者线程的 CacheStorage 对象。
- clients：返回服务工作者线程的 Clients 接口，用于访问底层 Client 对象。
- registration：返回服务工作者线程的 ServiceWorkerRegistration 对象。
- skipWaiting()：强制服务工作者线程进入活动状态；需要跟 Clients.claim()一起使用。
- fetch()：在服务工作者线程内发送常规网络请求；用于在服务工作者线程确定有必要发送实际网络请求（而不是返回缓存值）时。

虽然专用工作者线程和共享工作者线程只有一个 message 事件作为输入，但服务工作者线程则可以接收很多事件，包括页面操作、通知操作触发的事件或推送事件。

> **注意**　根据浏览器实现，在 SeviceWorker 中把日志打印到控制台不一定能在浏览器默认控制台中看到。

服务工作者线程的全局作用域可以监听以下事件，这里进行了分类。

- **服务工作者线程状态**

- install：在服务工作者线程进入**安装**状态时触发（在客户端可以通过 ServiceWorkerRegistration.installing 判断）。也可以在 self.onintall 属性上指定该事件的处理程序。
  - 这是服务工作者线程接收的第一个事件，在线程一开始执行时就会触发。
  - 每个服务工作者线程只会调用一次。
- activate：在服务工作者线程进入**激活**或**已激活**状态时触发（在客户端可以通过 ServiceWorkerRegistration.active 判断）。也可以在 self.onactive 属性上指定该事件的处理程序。
  - 此事件在服务工作者线程准备好处理功能性事件和控制客户端时触发。
  - 此事件并不代表服务工作者线程在控制客户端，只表明具有控制客户端的条件。

- **Fetch API**

- fetch：在服务工作者线程截获来自主页面的 fetch()请求时触发。服务工作者线程的 fetch 事件处理程序可以访问 FetchEvent，可以根据需要调整输出。也可以在 self.onfetch 属性上指定该事件的处理程序。

- **Message API**

- message：在服务工作者线程通过 postMesssage()获取数据时触发。也可以在 self.onmessage 属性上指定该事件的处理程序。

- **Notification API**

- notificationclick：在系统告诉浏览器用户点击了 ServiceWorkerRegistration.showNotification()生成的通知时触发。也可以在 self.onnotificationclick 属性上指定该事件的处理程序。
- notificationclose：在系统告诉浏览器用户关闭或取消显示了 ServiceWorkerRegistration.showNotification()生成的通知时触发。也可以在 self.onnotificationclose 属性上指定该事件的处理程序。

- **Push API**

❑ push：在服务工作者线程接收到推送消息时触发。也可以在 self.onpush 属性上指定该事件的处理程序。

❑ pushsubscriptionchange：在应用控制外的因素（非 JavaScript 显式操作）导致推送订阅状态变化时触发。也可以在 self.onpushsubscriptionchange 属性上指定该事件的处理程序。

> **注意** 有些浏览器也支持 async 事件，该事件是在 Background Sync API 中定义的。Background Sync API 还没有标准化，目前只有 Chrome 和 Opera 支持，因此本书没介绍。

8. 服务工作者线程作用域限制

服务工作者线程只能拦截其作用域内的客户端发送的请求。作用域是相对于获取服务脚本的路径定义的。如果没有在 register()中指定，则作用域就是服务脚本的路径。

（本章中涉及注册服务工作者线程的例子都使用脚本绝对 URL，以避免混淆。）下面第一个例子演示通过根目录获取服务脚本对应的默认根作用域：

```
navigator.serviceWorker.register('/serviceWorker.js')
.then((serviceWorkerRegistration) => {
  console.log(serviceWorkerRegistration.scope);
  // https://example.com/
});

// 以下请求都会被拦截：
// fetch('/foo.js');
// fetch('/foo/fooScript.js');
// fetch('/baz/bazScript.js');
```

下面的例子演示了通过根目录获取服务脚本但指定了同一目录作用域：

```
navigator.serviceWorker.register('/serviceWorker.js', {scope: './'})
.then((serviceWorkerRegistration) => {
  console.log(serviceWorkerRegistration.scope);
  // https://example.com/
});

// 以下请求都会被拦截：
// fetch('/foo.js');
// fetch('/foo/fooScript.js');
// fetch('/baz/bazScript.js');
```

下面的例子演示了通过根目录获取服务脚本但限定了目录作用域：

```
navigator.serviceWorker.register('/serviceWorker.js', {scope: './foo'})
.then((serviceWorkerRegistration) => {
  console.log(serviceWorkerRegistration.scope);
  // https://example.com/foo/
});

// 以下请求都会被拦截：
// fetch('/foo/fooScript.js');

// 以下请求都不会被拦截：
// fetch('/foo.js');
// fetch('/baz/bazScript.js');
```

下面的例子演示了通过嵌套的二级目录获取服务脚本对应的同一目录作用域：

```
navigator.serviceWorker.register('/foo/serviceWorker.js')
.then((serviceWorkerRegistration) => {
  console.log(serviceWorkerRegistration.scope);
  // https://example.com/foo/
});

// 以下请求都会被拦截：
// fetch('/foo/fooScript.js');

// 以下请求都不会被拦截：
// fetch('/foo.js');
// fetch('/baz/bazScript.js');
```

服务工作者线程的作用域实际上遵循了目录权限模型，即只能相对于服务脚本所在路径缩小作用域。像下面这样扩展作用域会抛出错误：

```
navigator.serviceWorker.register('/foo/serviceWorker.js', {scope: '/'});

// Error: The path of the provided scope 'https://example.com/'
// is not under the max scope allowed 'https://example.com/foo/'
```

通常，服务工作者线程作用域会使用末尾带斜杠的绝对路径来定义，比如：

```
navigator.serviceWorker.register('/serviceWorker.js', {scope: '/foo/'})
```

这样定义作用域有两个目的：将脚本文件的相对路径与作用域的相对路径分开，同时将该路径本身排除在作用域之外。例如，对于前面的代码片段而言，可能不需要在服务工作者线程的作用域中包含路径`/foo`。在末尾加上一个斜杠就可以明确排除`/foo`。当然，这要求绝对作用域路径不能扩展到服务工作者线程路径外。

如果想扩展服务工作者线程的作用域，主要有两种方式。

- 通过包含想要的作用域的路径提供（获取）服务脚本。
- 给服务脚本的响应添加 `Service-Worker-Allowed` 头部，把它的值设置为想要的作用域。该作用域值应该与 `register()`中的作用域值一致。

27.4.2　服务工作者线程缓存

在服务工作者线程之前，网页缺少缓存网络请求的稳健机制。浏览器一直使用 HTTP 缓存，但 HTTP 缓存并没有对 JavaScript 暴露编程接口，且其行为是受 JavaScript 运行时外部控制的。可以开发临时缓存机制，缓存响应字符串或 blob，但这种策略比较麻烦且效率低。

JavaScript 缓存的实现之前也有过尝试。MDN 文档也介绍了：

> 之前的尝试，即 AppCache，看起来是个不错的想法，因为它支持非常容易地指定要缓存的资源。可是，它对你想要做的事情做了很多假设，当应用程序没有完全遵循这些假设时，它就崩溃了。

服务工作者线程的一个主要能力是可以通过编程方式实现真正的网络请求缓存机制。与 HTTP 缓存或 CPU 缓存不同，服务工作者线程缓存非常简单。

- **服务工作者线程缓存不自动缓存任何请求**。所有缓存都必须明确指定。
- **服务工作者线程缓存没有到期失效的概念**。除非明确删除，否则缓存内容一直有效。

- **服务工作者线程缓存必须手动更新和删除**。
- **缓存版本必须手动管理**。每次服务工作者线程更新，新服务工作者线程负责提供新的缓存键以保存新缓存。
- **唯一的浏览器强制逐出策略基于服务工作者线程缓存占用的空间**。服务工作者线程负责管理自己缓存占用的空间。缓存超过浏览器限制时，浏览器会基于最近最少使用（LRU，Least Recently Used）原则为新缓存腾出空间。

本质上，服务工作者线程缓存机制是一个双层字典，其中顶级字典的条目映射到二级嵌套字典。顶级字典是 CacheStorage 对象，可以通过服务工作者线程全局作用域的 caches 属性访问。顶级字典中的每个值都是一个 Cache 对象，该对象也是个字典，是 Request 对象到 Response 对象的映射。

与 LocalStorage 一样，Cache 对象在 CacheStorage 字典中无限期存在，会超出浏览器会话的界限。此外，Cache 条目只能以源为基础存取。

> **注意** 虽然 CacheStorage 和 Cache 对象是在 Service Worker 规范中定义的，但它们也可以在主页面或其他工作者线程中使用。

1. CacheStorage 对象

CacheStorage 对象是映射到 Cache 对象的字符串键/值存储。CacheStorage 提供的 API 类似于异步 Map。CacheStorage 的接口通过全局对象的 caches 属性暴露出来。

```
console.log(caches); // CacheStorage {}
```

CacheStorage 中的每个缓存可以通过给 caches.open()传入相应字符串键取得。非字符串键会转换为字符串。如果缓存不存在，就会创建。

Cache 对象是通过期约返回的：

```
caches.open('v1').then(console.log);

// Cache {}
```

与 Map 类似，CacheStorage 也有 has()、delete()和 keys()方法。这些方法与 Map 上对应方法类似，但都基于期约。

```
// 打开新缓存 v1
// 检查缓存 v1 是否存在
// 检查不存在的缓存 v2

caches.open('v1')
.then(() => caches.has('v1'))
.then(console.log)   // true
.then(() => caches.has('v2'))
.then(console.log);  // false

// 打开新缓存 v1
// 检查缓存 v1 是否存在
// 删除缓存 v1
// 再次检查缓存 v1 是否存在

caches.open('v1')
.then(() => caches.has('v1'))
.then(console.log)   // true
```