浏览器会在每次导航事件中检查新服务脚本，但有时候这样也太不够了。ServiceWorkerRegistration 对象为此提供了一个 update()方法，可以用来告诉浏览器去重新获取服务脚本，与现有的比较，然后必要时安装更新的服务工作者线程。可以这样来实现：

```
navigator.serviceWorker.register('./serviceWorker.js')
.then((registration) => {
  // 每 17 分钟左右检查一个更新版本
  setInterval(() => registration.update(), 1E6);
});
```

27.4.9　服务工作者线程消息

与专用工作者线程和共享工作者线程一样，服务工作者线程也能与客户端通过 postMessage()交换消息。实现通信的最简单方式是向活动工作者线程发送一条消息，然后使用事件对象发送回应。发送给服务工作者线程的消息可以在全局作用域处理，而发送回客户端的消息则可以在 ServiceWorker-Context 对象上处理：

ServiceWorker.js

```
self.onmessage = ({data, source}) => {
  console.log('service worker heard:', data);

  source.postMessage('bar');
};
```

main.js

```
navigator.serviceWorker.onmessage = ({data}) => {
  console.log('client heard:', data);
};

navigator.serviceWorker.register('./serviceWorker.js')
.then((registration) => {
  if (registration.active) {
    registration.active.postMessage('foo');
  }
});

// service worker heard: foo
// client heard: bar
```

也可以简单地使用 serviceWorker.controller 属性：

ServiceWorker.js

```
self.onmessage = ({data, source}) => {
  console.log('service worker heard:', data);

  source.postMessage('bar');
};
```

main.js

```
navigator.serviceWorker.onmessage = ({data}) => {
  console.log('client heard:', data);
};

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

```
.then(() => {
  if (navigator.serviceWorker.controller) {
    navigator.serviceWorker.controller.postMessage('foo');
  }
});

// service worker heard: foo
// client heard: bar
```

前面的例子在每次页面重新加载时都会运行。这是因为服务工作者线程会回应每次刷新后客户端脚本发送的消息。在通过新标签页打开这个页面时也一样。

如果服务工作者线程需要率先发送消息，可以像下面这样获得客户端的引用：

ServiceWorker.js

```
self.onmessage = ({data}) => {
  console.log('service worker heard:', data);
};

self.onactivate = () => {
  self.clients.matchAll({includeUncontrolled: true})
  .then((clientMatches) => clientMatches[0].postMessage('foo'));
};
```

main.js

```
navigator.serviceWorker.onmessage = ({data, source}) => {
  console.log('client heard:', data);

  source.postMessage('bar');
};

navigator.serviceWorker.register('./serviceWorker.js')

// client heard: foo
// service worker heard: bar
```

前面的例子只会运行一次，因为活动事件在每个服务工作者线程上只会触发一次。

因为客户端和服务工作者线程可以相互之间发送消息，所以通过 MessageChannel 或 BroadcastChannel 实现通信也是可能的。

27.4.10 拦截 fetch 事件

服务工作者线程最重要的一个特性就是拦截网络请求。服务工作者线程作用域中的网络请求会注册为 fetch 事件。这种拦截能力不限于 fetch() 方法发送的请求，也能拦截对 JavaScript、CSS、图片和 HTML（包括对主 HTML 文档本身）等资源发送的请求。这些请求可以来自 JavaScript，也可以通过 <script>、<link>或<img>标签创建。直观地说，这样是合理的：如果想让服务工作者线程模拟离线应用程序，它就必须能够把控页面正常运行所需的所有请求资源。

FetchEvent 继承自 ExtendableEvent。让服务工作者线程能够决定如何处理 fetch 事件的方法是 event.respondWith()。该方法接收期约，该期约会解决为一个 Response 对象。当然，该 Response 对象实际上来自哪里完全由服务工作者线程决定。可以来自网络，来自缓存，或者动态创建。下面几节将介绍几种网络/缓存策略，可以在服务工作者线程中使用。

1. 从网络返回

这个策略就是简单地转发 fetch 事件。那些绝对需要发送到服务器的请求例如 POST 请求就适合该策略。可以像下面实现这一策略：

```
self.onfetch = (fetchEvent) => {
  fetchEvent.respondWith(fetch(fetchEvent.request));
};
```

> **注意**　前面的代码只演示了如何使用 event.respondWith()。如果 event.respondWith() 没有被调用，浏览器也会通过网络发送请求。

2. 从缓存返回

这个策略其实就是缓存检查。对于任何肯定有缓存的资源（如在安装阶段缓存的资源），可以采用该策略：

```
self.onfetch = (fetchEvent) => {
  fetchEvent.respondWith(caches.match(fetchEvent.request));
};
```

3. 从网络返回，缓存作后备

这个策略把从网络获取最新的数据作为首选，但如果缓存中有值也会返回缓存的值。如果应用程序需要尽可能展示最新数据，但在离线的情况下仍要展示一些信息，就可以采用该策略：

```
self.onfetch = (fetchEvent) => {
  fetchEvent.respondWith(
    fetch(fetchEvent.request)
    .catch(() => caches.match(fetchEvent.request))
  );
};
```

4. 从缓存返回，网络作后备

这个策略优先考虑响应速度，但仍会在没有缓存的情况下发送网络请求。这是大多数渐进式 Web 应用程序（PWA，Progressive Web Application）采取的首选策略：

```
self.onfetch = (fetchEvent) => {
  fetchEvent.respondWith(
    caches.match(fetchEvent.request)
    .then((response) => response || fetch(fetchEvent.request))
  );
};
```

5. 通用后备

应用程序需要考虑缓存和网络都不可用的情况。服务工作者线程可以在安装时缓存后备资源，然后在缓存和网络都失败时返回它们：

```
self.onfetch = (fetchEvent) => {
  fetchEvent.respondWith(
    // 开始执行“从缓存返回，以网络为后备”策略
    caches.match(fetchEvent.request)
    .then((response) => response || fetch(fetchEvent.request))
    .catch(() => caches.match('/fallback.html'))
  );
};
```

这里的 catch()子句可以扩展为支持不同类型的后备，例如点位图、哑数据，等等。

> **注意** Jake Archibald 在 Google Developers 网站有一篇关于网络/缓存策略的好文章《离线指南》。

27.4.11 推送通知

对于模拟原生应用程序的 Web 应用程序而言，必须支持推送消息。这意味着网页必须能够接收服务器的推送事件，然后在设备上显示通知（即使应用程序没有运行）。当然，这在常规网页中肯定是不可能的。不过，有了服务工作者线程就可以实现该行为。

为了在 PWA 应用程序中支持推送通知，必须支持以下 4 种行为。

- 服务工作者线程必须能够显示通知。
- 服务工作者线程必须能够处理与这些通知的交互。
- 服务工作者线程必须能够订阅服务器发送的推送通知。
- 服务工作者线程必须能够处理推送消息，即使应用程序没在前台运行或者根本没打开。

1. 显示通知

服务工作者线程可以通过它们的注册对象使用 Notification API。这样做有很好的理由：与服务工作者线程关联的通知也会触发服务工作者线程内部的交互事件。

显示通知要求向用户明确地请求授权。授权完成后，可以通过 ServiceWorkerRegistration.showNotification()显示通知。下面是示例实现：

```
navigator.serviceWorker.register('./serviceWorker.js')
.then((registration) => {
  Notification.requestPermission()
  .then((status) => {
    if (status === 'granted') {
      registration.showNotification('foo');
    }
  });
});
```

类似地，在服务工作者线程内部可以使用全局 registration 属性触发通知：

```
self.onactivate = () => self.registration.showNotification('bar');
```

在上面的例子中，获得显示通知的授权后，会把 foo 通知显示在浏览器中。该通知与使用 new Notification()创建的通知看不出有任何差别。此外，显示该通知不需要服务工作者线程额外做任何事情。服务工作者线程只在需要处理通知事件时才会发挥作用。

2. 处理通知事件

通过 ServiceWorkerRegistration 对象创建的通知会向服务工作者线程发送 notificationclick 和 notificationclose 事件。假设前面例子中的服务脚本定义了如下事件处理程序：

```
self.onnotificationclick = ({notification}) => {
  console.log('notification click', notification);
};

self.onnotificationclose = ({notification}) => {
```

```
  console.log('notification close', notification);
};
```

在这个例子中，与通知的两种交互操作都在服务工作者线程中注册了处理程序。这里的 notification 事件对象暴露了 notification 属性，其中包含着生成该事件 Notification 对象。这些处理程序可以决定交互操作之后的响应方式。

一般来说，单击通知意味着用户希望转到某个具体的页面。在服务工作者线程处理程序中，可以通过 clients.openWindow()打开相应的 URL，例如：

```
self.onnotificationclick = ({notification}) => {
  clients.openWindow('https://foo.com');
};
```

3. 订阅推送事件

对于发送给服务工作者线程的推送消息，必须通过服务工作者线程的 PushManager 来订阅。这样服务工作者线程就可以在 push 事件处理程序中处理推送消息。

下面展示了使用 ServiceWorkerRegistration.pushManager 订阅推送消息的例子：

```
navigator.serviceWorker.register('./serviceWorker.js')
.then((registration) => {
  registration.pushManager.subscribe({
    applicationServerKey: key, // 来自服务器的公钥
    userVisibleOnly: true
  });
});
```

另外，服务工作者线程也可以使用全局的 registration 属性自己订阅：

```
self.onactivate = () => {
  self.registration.pushManager.subscribe({
    applicationServerKey: key, // 来自服务器的公钥
    userVisibleOnly: true
  });
};
```

4. 处理推送事件

订阅之后，服务工作者线程会在每次服务器推送消息时收到 push 事件。这时候它可以这样来处理：

```
self.onpush = (pushEvent) => {
  console.log('Service worker was pushed data:', pushEvent.data.text());
};
```

为实现真正的推送通知，这个处理程序只需要通过注册对象创建一个通知即可。不过，完善的推送通知需要创建它的服务工作者线程保持活动足够长时间，以便处理后续的交互事件。

要实现这一点，push 事件继承了 ExtendableEvent。可以把 showNotification()返回的期约传给 waitUntil()，这样就会让服务工作者线程一直活动到通知的期约解决。

下面展示了实现上述逻辑的简单框架：

main.js

```
navigator.serviceWorker.register('./serviceWorker.js')
.then((registration) => {
  // 请求显示通知的授权
  Notification.requestPermission()
  .then((status) => {
```

```
    if (status === 'granted') {
      // 如果获得授权，只订阅推送消息
      registration.pushManager.subscribe({
        applicationServerKey: key, // 来自服务器的公钥
        userVisibleOnly: true
      });
    }
  });
});
```

ServiceWorker.js

```
// 收到推送事件后，在通知中以文本形式显示数据
self.onpush = (pushEvent) => {
  // 保持服务工作者线程活动到通知期约解决
  pushEvent.waitUntil(
    self.registration.showNotification(pushEvent.data.text())
  );
};

// 如果用户单击通知，则打开相应的应用程序页面
self.onnotificationclick = ({notification}) => {
  clients.openWindow('https://example.com/clicked-notification');
};
```

27.5 小结

工作者线程可以运行异步 JavaScript 而不阻塞用户界面。这非常适合复杂计算和数据处理，特别是需要花较长时间因而会影响用户使用网页的处理任务。工作者线程有自己独立的环境，只能通过异步消息与外界通信。

工作者线程可以是专用线程、共享线程。专用线程只能由一个页面使用，而共享线程则可以由同源的任意页面共享。

服务工作者线程用于让网页模拟原生应用程序。服务工作者线程也是一种工作者线程，但它们更像是网络代理，而非独立的浏览器线程。可以把它们看成是高度定制化的网络缓存，它们也可以在 PWA 中支持推送通知。

第28章 最佳实践

本章内容

- 编写可维护的代码
- 保证代码性能
- 部署代码到线上环境

自 2000 年以来，Web 开发一直在以惊人的速度发展。从最初毫无章法可循的“野蛮生长”，到如今已发展出完整的规范体系，各种研究成果和最佳实践层出不穷。随着简单的网站变成复杂的 Web 应用程序，曾经的 Web 开发爱好者也变成了收入不菲的专业人士。Web 开发领域的最新技术和开发工具已经令人目不暇接。其中，JavaScript 尤其成为了研究和关注的焦点。JavaScript 的最佳实践可以分成几类，适用于开发流程的不同阶段。

28.1 可维护性

在早期网站中，JavaScript 主要用于实现一些小型动效或表单验证。今天的 Web 应用程序则动辄成千上万行 JavaScript 代码，用于完成各种各样的复杂处理。这些变化要求开发者把可维护能力放到重要位置上。正如更传统意义上的软件工程师一样，JavaScript 开发者受雇是要为公司创造价值的。他们不仅要保证产品如期上线，而且要随着时间推移为公司不断积累知识资产。

编写可维护的代码十分重要，因为大多数开发者会花大量时间去维护别人写的代码。实际开发中，从第一行代码开始写起的情况非常少，通常是要在别人的代码之上构建自己的工作。让自己的代码容易维护，可以保证其他开发者更好地完成自己的工作。

> **注意** 可维护代码的概念并不只适用于 JavaScript，其中很多概念适用于所有编程语言，尽管部分概念特定于 JavaScript。

28.1.1 什么是可维护的代码

通常，说代码“可维护”就意味着它具备如下特点。

- **容易理解**：无须求助原始开发者，任何人一看代码就知道它是干什么的，以及它是怎么实现的。
- **符合常识**：代码中的一切都显得顺理成章，无论操作有多么复杂。
- **容易适配**：即使数据发生变化也不用完全重写。
- **容易扩展**：代码架构经过认真设计，支持未来扩展核心功能。
- **容易调试**：出问题时，代码可以给出明确的信息，通过它能直接定位问题。

能够写出可维护的 JavaScript 代码是一项重要的专业技能。这就是业余爱好者和专业开发人员之间的区别，前者用一个周末就拼凑出一个网站，而后者真正了解自己的技术。

28.1.2 编码规范

编写可维护代码的第一步是认真考虑编码规范。大多数编程语言会涉及编码规范，简单上网一搜，就可以找到成千上万的相关文章。专业组织有为开发者建立的编码规范，旨在让人写出更容易维护的代码。优秀开源项目有严格的编码规范，可以让社区的所有人容易地理解代码是如何组织的。

编码规范对 JavaScript 而言非常重要，因为这门语言实在太灵活了。与大多数面向对象语言不同，JavaScript 并不强迫开发者把任何东西都定义为对象。它支持任何编程风格，包括传统的面向对象编程、声明式编程，以及函数式编程。简单看几个开源的 JavaScript 库，就会发现有很多方式可以创建对象、定义方法和管理环境。

接下来的几节会讨论制定编码规范的一些基础知识。这些话题很重要，当然每个人的需求不同，实现方式也可以不同。

1. 可读性

要想让代码容易维护，首先必须使其可读。可读性必须考虑代码是一种文本文件。为此，代码缩进是保证可读性的重要基础。如果所有人都使用相同的缩进，整个项目的代码就会更容易让人看懂。缩进通常要使用空格数而不是 Tab（制表符）来定义，因为后者在不同文本编辑器中的显示不同。一般来说，缩进是 4 个空格，当然具体多少个可以自己定。

可读性的另一方面是代码注释。在大多数编程语言中，广泛接受的做法是为每个方法都编写注释。因为 JavaScript 可以在代码中的任何地方创建函数，所以这一点经常被忽视。正因为如此，可能给 JavaScript 中的每个函数都写注释才更重要。一般来说，以下这些地方应该写注释。

- **函数和方法**。每个函数和方法都应该有注释来描述其用途，以及完成任务所用的算法。同时，也写清使用这个函数或方法的前提（假设）、每个参数的含义，以及函数是否返回值（因为通过函数定义看不出来）。
- **大型代码块**。多行代码但用于完成单一任务的，应该在前面给出注释，把要完成的任务写清楚。
- **复杂的算法**。如果使用了独特的方法解决问题，要通过注释解释明白。这样不仅可以帮助别人查看代码，也可以帮助自己今后查看代码。
- **使用黑科技**。由于浏览器之间的差异，JavaScript 代码中通常包含一些黑科技。不要假设其他人一看就能明白某个黑科技是为了解决某个浏览器的什么问题。如果某个浏览器不能使用正常方式达到目的，那要在注释里把黑科技的用途写出来。这样可以避免别人误以为黑科技没有用而把它“修复”掉，结果你已解决的问题又会出现。

缩进和注释可以让代码更容易理解，将来也更容易维护。

2. 变量和函数命名

代码中变量和函数的适当命名对于其可读性和可维护性至关重要。因为很多 JavaScript 开发者是业余爱好者出身，所以很容易用 `foo`、`bar` 命名变量，用 `doSomething` 来命名函数。专业 JavaScript 开发者必须改掉这些习惯，这样才能写出可维护的代码。以下是关于命名的通用规则。

- 变量名应该是名词，例如 `car` 或 `person`。
- 函数名应该以动词开始，例如 `getName()`。返回布尔值的函数通常以 `is` 开头，比如 `isEnabled()`。

- 对变量和函数都使用符合逻辑的名称，不用担心长度。长名字的问题可以通过后处理和压缩解决（本章稍后会讨论）。
- 变量、函数和方法应该以小写字母开头，使用驼峰大小写（camelCase）形式，如 `getName()`和 `isPerson`。类名应该首字母大写，如 `Person`、`RequestFactory`。常量值应该全部大写并以下划线相接，比如 `REQUEST_TIMEOUT`。
- 名称要尽量用描述性和直观的词汇，但不要过于冗长。`getName()`一看就知道会返回名称，而 `PersonFactory` 一看就知道会产生某个 `Person` 对象或实体。

要完全避免没有用的变量名，如不能表示所包含数据的类型的变量名。通过适当命名，代码读起来就会像故事，因此更容易理解。

3. 变量类型透明化

因为 JavaScript 是松散类型的语言，所以很容易忘记变量包含的数据类型。适当命名可以在某种程度上解决这个问题，但还不够。有三种方式可以标明变量的数据类型。

第一种标明变量类型的方式是通过初始化。定义变量时，应该立即将其初始化为一个将来要使用的类型值。例如，要保存布尔值的变量，可以将其初始化为 `true` 或 `false`；而要保存数值的变量，可以将其初始化为一个数值。再看几个例子：

```
// 通过初始化标明变量类型
let found = false;       // 布尔值
let count = -1;          // 数值
let name = "";           // 字符串
let person = null;       // 对象
```

初始化为特定数据类型的值可以明确表示变量的类型。ES6 之前，初始化方式不适合函数声明中函数的参数；ES6 之后，可以在函数声明中为参数指定默认值来标明参数类型。

第二种标明变量类型的方式是使用匈牙利表示法。匈牙利表示法指的是在变量名前面前缀一个或多个字符表示数据类型。这种表示法曾在脚本语言中非常流行，很长时间以来也是 JavaScript 首选的格式。对于基本数据类型，JavaScript 传统的匈牙利表示法用 `o` 表示对象，`s` 表示字符串，`i` 表示整数，`f` 表示浮点数，`b` 表示布尔值。示例如下：

```
// 使用匈牙利表示法标明数据类型
let bFound;    // 布尔值
let iCount;    // 整数
let sName;     // 字符串
let oPerson;   // 对象
```

匈牙利表示法也可以很好地应用于函数参数。它的缺点是使代码可读性下降、不够直观，并破坏了类似句子的自然阅读流畅性。因此，匈牙利表示法在开发者中失宠了。

最后一种标明变量类型的方式是使用类型注释。类型注释放在变量名后面、初始化表达式的前面。基本思路是在变量旁边使用注释说明类型，比如：

```
// 使用类型注释表明数据类型
let found  /*:Boolean*/ = false;
let count  /*:int*/     = 10;
let name   /*:String*/  = "Nicholas";
let person /*:Object*/  = null;
```

类型注释在保持代码整体可读性的同时向其注释了类型信息。类型注释的缺点是不能再使用多行注释把大型代码块注释掉了。因为类型注释也是多行注释，所以会造成干扰，如下例所示：

```
// 这样多行注释不会生效
/*
let found  /*:Boolean*/ = false;
let count  /*:int*/     = 10;
let name   /*:String*/  = "Nicholas";
let person /*:Object*/  = null;
*/
```

这里本来是想使用多行注释把所有变量声明都注释掉。但类型注释产生了干扰，因为第一个/*（第 2 行）的实例会与第一个*/（第 3 行）的实例匹配，所以会导致语法错误。如果想注释掉使用类型注释的代码，则只能使用单行注释一行一行地注释掉每一行（很多编辑器可以自动完成）。

以上是最常用的三种标明变量数据类型方式。每种方式都有其优点和缺点，可以根据实际情况选用。关键要看哪一种最适合自己的项目，并保证一致性。

28.1.3 松散耦合

只要应用程序的某个部分对另一个部分依赖得过于紧密，代码就会变成紧密耦合，因而难以维护。典型的问题是在一个对象中直接引用另一个对象，这样，修改其中一个，可能必须还得修改另一个。紧密耦合的软件难于维护，肯定需要频繁地重写。

考虑到相关的技术，Web 应用程序在某些情况下可能变得过于紧密耦合。关键在于有这个意识，随时注意不要让代码产生紧密耦合。

1. 解耦 HTML/JavaScript

Web 开发中最常见的耦合是 HTML/JavaScript 耦合。在网页中，HTML 和 JavaScript 分别代表不同层面的解决方案。HTML 是数据，JavaScript 是行为。这是因为它们之间要交互操作，需要通过不同的方式将这两种技术联系起来。可惜的是，其中一些方式会导致 HTML 与 JavaScript 紧密耦合。

把 JavaScript 直接嵌入在 HTML 中，要么使用包含嵌入代码的<script>元素，要么使用 HTML 属性添加事件处理程序，这些都会造成紧密耦合。比如下面的例子：

```
<!-- 使用<script>造成 HTML/JavaScript 紧密耦合 -->
<script>
  document.write("Hello world!");
</script>

<!-- 使用事件处理程序属性造成 HTML/JavaScript 紧密耦合 -->
<input type="button" value="Click Me" onclick="doSomething()"/>
```

虽然技术上这样做没有问题，但实践中，这样会将表示数据的 HTML 与定义行为的 JavaScript 紧密耦合在一起。理想情况下，HTML 和 JavaScript 应该完全分开，通过外部文件引入 JavaScript，然后使用 DOM 添加行为。

HTML 与 JavaScript 紧密耦合的情况下，每次分析 JavaScript 的报错都要先确定错误来自 HTML 还是 JavaScript。这样也会引入代码可用性的新错误。在这个例子中，用户可能会在 doSomething()函数可用之前点击按钮，从而导致 JavaScript 报错。因为每次修改按钮的行为都需要既改 HTML 又改 JavaScript，而实际上只有后者才是有必要修改的，所以就会降低代码的可维护性。

在相反的情况下，HTML 和 JavaScript 也会变得紧密耦合：把 HTML 包含在 JavaScript 中。这种情况通常发生在把一段 HTML 通过 innerHTML 插入到页面中时，示例如下：