Atitit atit it pwa Service Worke重要api

目录

[1.1. 注册 1](#_Toc7487)

[1.2. 安装 1](#_Toc21752)

[1.3. Fetch事件 api 1](#_Toc4333)

[1.4. 更新 Service Worker 2](#_Toc539)

[2. Other 3](#_Toc5145)

[2.1.1. fetch() 默认值 3](#_Toc28536)

[2.1.2. 处理响应式图像 3](#_Toc9090)

[3. 常遵循以下基本步骤来使用 service workers： 5](#_Toc1636)

[3.1. Promises 5](#_Toc11375)

## 注册

## 安装

在受控页面启动注册流程后，我们来看看处理 install 事件的 Service Worker 脚本。

最基本的例子是，您需要为安装事件定义回调，并决定想要缓存的文件。

self.addEventListener('install', function(event) {  
  // Perform install steps  
});

## Fetch事件 api

缓存和返回请求

您已安装 Service Worker，现在可能会想要返回一个缓存的响应，对吧？

在安装 Service Worker 且用户转至其他页面或刷新当前页面后，Service Worker 将开始接收 fetch 事件。下面提供了一个示例。

## 更新 Service Worker

在某个时间点，您的 Service Worker 需要更新。 此时，您需要遵循以下步骤:

1. 更新您的服务工作线程 JavaScript 文件。 用户导航至您的站点时，浏览器会尝试在后台重新下载定义 Service Worker 的脚本文件。 如果 Service Worker 文件与其当前所用文件存在字节差异，则将其视为*新 Service Worker*。
2. 新 Service Worker 将会启动，且将会触发 install 事件。
3. 此时，旧 Service Worker 仍控制着当前页面，因此新 Service Worker 将进入 waiting 状态。
4. 当网站上当前打开的页面关闭时，旧 Service Worker 将会被终止，新 Service Worker 将会取得控制权。
5. 新 Service Worker 取得控制权后，将会触发其 activate 事件。

出现在 activate 回调中的一个常见任务是缓存管理。 您希望在 activate 回调中执行此任务的原因在于，如果您在安装步骤中清除了任何旧缓存，则继续控制所有当前页面的任何旧 Service Worker 将突然无法从缓存中提供文件。

比如说我们有一个名为 'my-site-cache-v1' 的缓存，我们想要将该缓存拆分为一个页面缓存和一个博文缓存。 这就意味着在安装步骤中我们创建了两个缓存: 'pages-cache-v1' 和 'blog-posts-cache-v1'，且在激活步骤中我们要删除旧的 'my-site-cache-v1'。

# Other

### fetch() 默认值

#### 默认情况下没有凭据

使用 fetch 时，默认情况下请求中不包含 Cookie 等凭据。 如需凭据，改为调用:

fetch(url, {  
  credentials: 'include'  
})

这一行为是有意为之，可以说比 XHR 更复杂的以下默认行为更好: 如果网址具有相同来源，则默认发送凭据，否则忽略。 提取的行为更接近于其他 CORS 请求，如 <img crossorigin>，它将决不会发送 Cookie，除非您使用 <img crossorigin="use-credentials"> 选择加入。

#### 非 CORS 默认失败

默认情况下，从不支持 CORS 的第三方网址中提取资源将会失败。 您可以向请求中添加 no-CORS 选项来克服此问题，不过这可能会导致“不透明”的响应，这意味着您无法辨别响应是否成功。

cache.addAll(urlsToPrefetch.map(function(urlToPrefetch) {  
  return new Request(urlToPrefetch, { mode: 'no-cors' });  
})).then(function() {  
  console.log('All resources have been fetched and cached.');  
});

### 处理响应式图像

srcset 属性或 <picture> 元素将在运行期间选择最适当的图像资产，并发出网络请求。

对于 Service Worker，如果您想要在安装过程中缓存图像，您有下列几种选择:

1. 安装 <picture> 元素和 srcset 属性将请求的所有图像。
2. 安装一个低分辨率版本的图像。
3. 安装一个高分辨率版本的图像。

实际上，您应该选择 2 或 3，因为下载所有图像会浪费存储空间。

假定您在安装期间选择安装低分辨率版本的图像，在页面加载时您想要尝试从网络中检索高分辨率的图像，但是如果检索高分辨率版本失败，则回退到低分辨率版本。 这没有问题，而且这种做法很好，但是有另外一个问题。

如果我们有以下两张图像:

| **屏幕密度** | **宽度** | **高度** |
| --- | --- | --- |
| 1x | 400 | 400 |
| 2x | 800 | 800 |

在 srcset 图像中，我们有一些像这样的标记:

<img src="image-src.png" srcset="image-src.png 1x, image-2x.png 2x" />

如果我们使用的是 2x 显示屏，浏览器将会选择下载 image-2x.png。如果我们处于离线状态，您可以对请求执行 .catch() 并返回 image-src.png（如已缓存）。但是，浏览器会期望 2x 屏幕上的图像有额外的像素，这样图像将显示为 200x200 CSS 像素而不是 400x400 CSS 像素。 解决该问题的唯一办法是设定固定的图像高度和宽度。

<img src="image-src.png" srcset="image-src.png 1x, image-2x.png 2x"  
 style="width:400px; height: 400px;" />

对于要用于艺术指导的 <picture> 元素，这会变得相当困难，而且很大程度上取决于图像的创建和使用方式，但是您可以使用类似于 srcset 的方法。

# 常遵循以下基本步骤来使用 service workers：

1. service worker URL 通过 [serviceWorkerContainer.register()](https://developer.mozilla.org/zh-CN/docs/Web/API/ServiceWorkerContainer/register) 来获取和注册。
2. 如果注册成功，service worker 就在 [ServiceWorkerGlobalScope](https://developer.mozilla.org/zh-CN/docs/Web/API/ServiceWorkerGlobalScope) 环境中运行； 这是一个特殊类型的 worker 上下文运行环境，与主运行线程（执行脚本）相独立，同时也没有访问 DOM 的能力。
3. service worker 现在可以处理事件了。
4. 受 service worker 控制的页面打开后会尝试去安装 service worker。最先发送给 service worker 的事件是安装事件(在这个事件里可以开始进行填充 IndexDB和缓存站点资源)。这个流程同原生 APP 或者 Firefox OS APP 是一样的 — 让所有资源可离线访问。
5. 当 oninstall 事件的处理程序执行完毕后，可以认为 service worker 安装完成了。
6. 下一步是激活。当 service worker 安装完成后，会接收到一个激活事件(activate event)。 onactivate 主要用途是清理先前版本的 service worker 脚本中使用的资源。
7. Service Worker 现在可以控制页面了，但仅是在 register()  成功后的打开的页面。也就是说，页面起始于有没有 service worker ，且在页面的接下来生命周期内维持这个状态。所以，页面不得不重新加载以让 service worker 获得完全的控制。

## [Promises](https://developer.mozilla.org/zh-CN/docs/Web/API/Service_Worker_API/Using_Service_Workers" \l "promises" \o "Permalink to Promises)

[Promises](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) 是一种非常适用于异步操作的机制，一个操作依赖于另一个操作的成功执行。这是 service worker 的核心工作机制。  
  
Promises 可以做很多事情。但现在，你只需要知道，如果有什么返回了一个promise，你可以在后面加上 .then() 来传入成功和失败的回调函数。或者，你可以在后面加上 .catch() 如果你想添加一个操作失败的回调函数。

接下来，让我们对比一下传统的同步回调结构，和异步promise结构，两者在功能上是等效的：