链滴

PWA 之 Service Worker 从介绍到实战再到爬坑

作者：Vanessa

原文链接：https://ld246.com/article/1520483961387

来源网站：链滴

继 现在你可以在电脑Chrome上使用PWA啦！ 等一系列文章发出后，PWA 又要开始火一波了。

概念

PWA(Progressive Web Apps)

我们访问互联网的方式已经改变。随着智能手机和移动设备的增长，我们看到全球数百万人首次在移设备上使用互联网。 Google 支持渐进式 Web 应用程序（PWA），以帮助开发人员在 Web 应用程中能快速、可靠、高质量的提供和本机应用相媲美的程序。

Service Worker

什么是 Service Worker

Service Worker 是一种 Web Worker。它本质上是一个与主浏览器线程分开运行的 JavaScript 文件可以拦截网络请求、缓存资源或从缓存中检索资源、传递推送消息。

由于 Workers 与主线程分开运行，因此 Service Worker 独立于与其关联的应用程序。这将导致一结果：

- 由于 Service Worker 没有阻塞（它被设计为完全异步），同步 XHR 和 localStorage 不能在 Service Worker 中使用。
- 当应用程序处于没有活动状态时，Service Worker 可以从服务器接收推送消息。这可以让您的应程序向用户显示推送通知，即使它未在浏览器中打开。

注意 浏览器在没有运行的情况下是否能收到通知取决于浏览器如何与操作系统集成。例如，在桌面操系统上，Chrome 浏览器和 Firefox 只会在浏览器运行时收到通知。然而，Android 是在接收到推送息时唤醒任何浏览器，并且无论浏览器状态如何都将始终接收推送消息。有关更多信息，请参阅 Matt Gaunt 的 Web Push Book 中的常见问题解答。

● Service Worker 不能直接访问 DOM。为了与页面通信，需使用 postMessage() 方法发送数据并使用 message 事件侦听器来接收数据。

Service Worker 注意事项：

● Service Worker 是一个可编程的网络代理，可以控制如何处理来自页面的网络请求。

● Service Worker 只能通过 HTTPS 运行。由于 Service Worker 可以拦截网络请求并修改响应，因会带来非常糟糕的 "man-in-the-middle" 攻击。

注意 像 Letsencrypt 这样的服务可让您免费获取 SSL 证书以安装到您的服务器上。

● Service Worker 在不使用时变为空闲状态，并在下次需要时重新启动。你不能依赖事件之间持续在的全局状态。如果存在需要在重新启动时保留和重用的信息，则可以使用 IndexedDB 数据库。

● Service Worker 广泛使用 Promises，所以如果你对 Promises 不熟悉，那么你应该停止阅读并开学习 Promises 的介绍。

教程

Service Worker 生命周期

Registration

要**安装** Service Worker，您需要在 JavaScript 主进程中进行**注册**。注册时需要告诉浏览器您的 Service Worker 所在的位置，然后在后台安装它。如：

main.js

```
if ('serviceWorker' in navigator) {
    navigator.serviceWorker.register('/service-worker.js').then(function (registration) {
    console.log('Registration successful, scope is:', registration.scope);
  }).catch(function (error) {
    console.log('Service Worker registration failed, error:', error);
  });
}
```

registration.scope 决定 Service Worker 可以控制哪些文件，换句话说Service Worker 将从哪个路拦截请求。默认的范围是 Service Worker 文件的位置，并扩展到以下所有目录。因此，如果 service worker.js 位于根目录中，则服务工作人员将控制来自该域中所有文件的请求。

当然您还可以在注册时通过传入附加参数来设置任意范围。例如：

main.js

```
navigator.serviceWorker.register('/service-worker.js',  {
  scope:  '/app/'
});
```

Installation

一旦浏览器注册了 Service Worker，**Installation**就会被触发。以下情况都会触发 Installation：

● Service Worker 被浏览器认为是新的

- 该站点当前没有注册过 Service Worker
- 新的 Service Worker 和先前安装的 Service Worker 之间存在字节差异

Service Worker Installation 会在 Service Worker installing 过程中触发 install 事件。 我们可以在 ervice Worker 监听 install 事件，以便在 Service Worker 安装时执行一些任务。例如，在安装过程，Service Worker 可以预先缓存 Web 应用程序的某些部分，以便在用户下次打开应用程序时立即加它（请参阅

caching the application shell）。所以，在第一次加载之后，后面的重复加载都会被缓存，这样，互性上的时间将会缩短。监听示例如下：

service-worker.js

```
self.addEventListener('install', function(event) {
  // Perform some task
});
```

Activation

一旦 Service Worker 成功安装，它将转换到**Activation**阶段。如果以前的 Service Worker 还在服着任何打开的页面，则新的 Service Worker 进入 waiting 状态。新的 Service Worker 仅在旧的 Serv ce Worker 没有任何页面被加载时激活。这确保了在任何时间内只有一个版本的 Service Worker 正运行。

注意一般的页面刷新不会将控制权转移给新的 Service Worker，因为刷新之前新页面并不会被加载整个过程中旧的 Service Worker 将会一直被使用。

当新的 Service Worker 激活时，activate 事件将被触发。此事件侦听器可以用来清理过时的缓存（参阅 Offline Cookbook 中的示例）。

service-worker.js

```
self.addEventListener('activate', function(event) { // Perform some task
});
```

激活后，Service Worker 将控制加载在其范围内的所有页面，并开始监听来自这些页面的事件。但，在 Service Worker 激活之前加载的页面不在 Service Worker 控制之下。当您关闭并重新打开您应用程序时，或者 Service Worker 调用 clients.claim 时，新的 Service Worker 才会生效。在此之，来自该页面的请求将不会被新的 Service Worker 拦截。这是可以保证您网站的一致性。

HacPai 离线应用完整代码

common.js

```
/**
 * @description 注册 service worker
 */
_initServiceWorker: function () {
  var isWeChat = navigator.userAgent.toLowerCase().indexOf('micromessenger') > -1;
  if ('serviceWorker' in navigator && 'caches' in window && 'fetch' in window && Label.miniP
stfix !== '' &&
    !isWeChat) {
    navigator.serviceWorker.register('/sw.min.js?' + Label.staticResourceVersion, {scope: '/'});
  }
```

```
}
```

sw.js

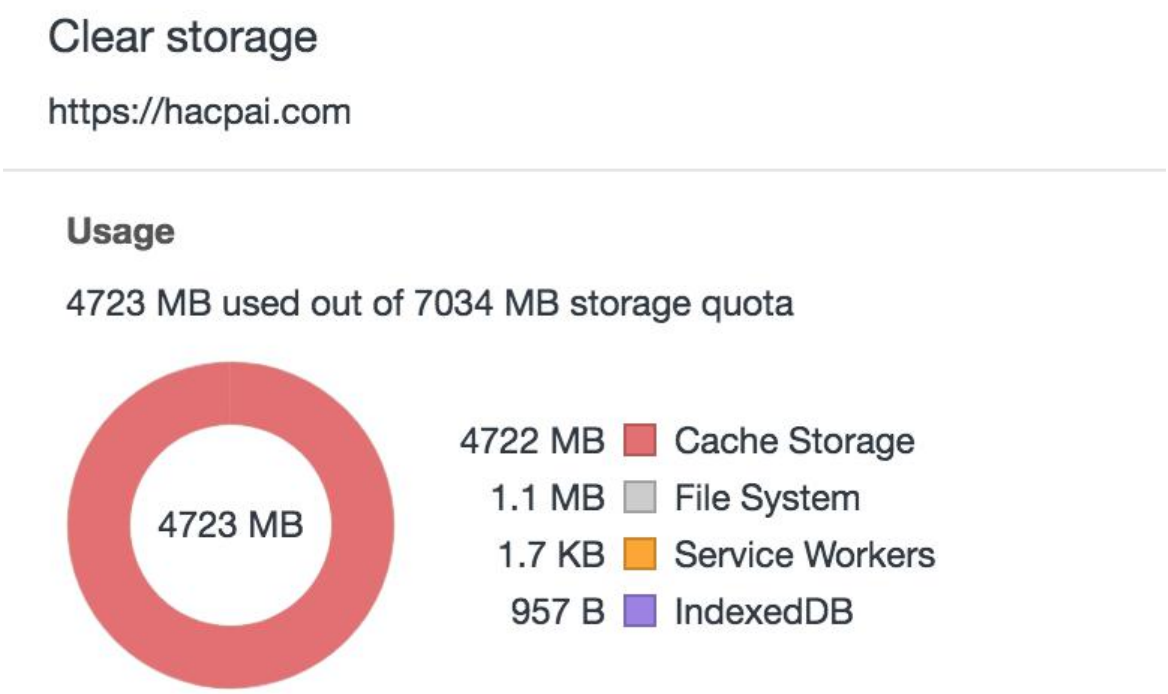
```
/*
 * Symphony - A modern community (forum/SNS/blog) platform written in Java.
 * Copyright (C) 2012-2017,  b3log.org & hacpai.com
 *
 * 本文件属于 Sym 商业版的一部分，请仔细阅读项目根文件夹的 LICENSE 并严格遵守相关约定
 */
/**
 * @fileoverview service work.
 *
 * @author <a href="http://vanessa.b3log.org">Liyuan Li</a>
 * @version 0.2.2.1, Jan 24, 2018
 * @since 2.2.0
 */

const version = '1520176925816';
const staticServePath = 'https://static.hacpai.com/';
const imgServePath = 'https://b3logfile.com/';
const servePath = 'https://hacpai.com/';
/**
 * @description add offline cache
 */
self.addEventListener('activate', event => {
  // delete all caches
  event.waitUntil(
    caches.keys().then(function (keyList) {
      return Promise.all(keyList.map(async function (key) {
        const storageStats = await navigator.storage.estimate();
        if (key !== 'hacpai-html' && key !== 'hacpai-image' &&
          key !== 'hacpai-static-' + version) {
          return caches.delete(key);
        } else if (storageStats.usage / storageStats.quota > 0.8 && (key === 'hacpai-html' || key
== 'hacpai-image')) {
          console.log(`clear ${key} cache`);
          return caches.delete(key);
        }
      }));
    })
  );
});

// 请求截取
self.addEventListener('fetch', event => {
  if (event.request.headers.get('accept').indexOf('text/html') === 0 || (
      event.request.headers.get('accept') === '*/*' &&
      event.request.url.indexOf('/js/') === -1 &&
      event.request.url.indexOf('/notification/unread/count') === -1
    )) {
    // 动态资源
    event.respondWith(
      // 动态资源需要每次进行更新
```

```
    fetch(event.request).then(function (response) {
      // 站点以外的需求不缓存
      if (event.request.url.indexOf(servePath) === -1) {
        return response;
      }
      const responseClone = response.clone();
      caches.open('hacpai-html').then(function (cache) {
        // 更新动态资源的缓存
        if (event.request.method !== 'POST' && event.request.method !== 'DELETE' &&
          event.request.method !== 'PUT') {
          // cache is unsupported POST and so on
          cache.put(event.request, responseClone);
        }
      });
      return response;
    }).catch(function () {
      // 动态资源需离线后从缓存中获取
      return caches.match(event.request);
    })
  );
} else {
  // 静态资源
  event.respondWith(
    caches.match(event.request).then(response => {
      // 指定的静态资源直接从缓存中获取
      return response ||
        // 没有指定的静态资源从服务器拉取
        fetch(event.request).then(function (fetchResponse) {
          if (event.request.url.indexOf(imgServePath) > -1 ||
            event.request.url.indexOf(servePath + 'porter') > -1 ||
            event.request.url.indexOf(staticServePath + 'emoji/') > -1 ||
            event.request.url.indexOf(staticServePath + 'images/emotions/') > -1) {
            // 对用户头像、图片、solo代理图片、emoji、solo emotion 进行缓存
            return caches.open('hacpai-image').then(function (cache) {
              cache.put(event.request, fetchResponse.clone());
              return fetchResponse;
            });
          } else if (event.request.url.indexOf(staticServePath + 'css/') > -1 ||
            event.request.url.indexOf(staticServePath + 'js/') > -1 ||
            event.request.url.indexOf(staticServePath + 'images/') > -1) {
            // 对 css, js, image(不含 emoji) 进行缓存
            return caches.open('hacpai-static-' + version).then(function (cache) {
              cache.put(event.request, fetchResponse.clone());
              return fetchResponse;
            });
          } else {
            return fetchResponse;
          }
        }).catch(function () {
          // 静态资源获取失败
          console.log(`fetch ${event.request.url} error`)
        });
    })
  )
```

```
  }
});
```

坑

缓存



缓存是有限的，所以要定期清理。超出的时候会出现 Uncaught (in promise) DOMException: Quota exceeded. 异常。调用清理后，必须要重启浏览器才生效。

微信

在服务端获取 useragent 浏览器信息的时候，会一下带微信标示 MicroMessenger，一下不带。最移除 Service Worker 就正常了。

不支持以下原生方法

- history.back()
- window.open()
- 文件选择