Service Worker

贾正权 2018/05/04

Contents

- ✓Web offline caching
- ✓AppCache
- ✓Service Worker introduction
- ✓Service Worker model
- ✓Client context
- ✓Execution context
- ✓CacheStorage & Cache
- ✓Security & Privacy
- ✓Service Worker Lifecycle
- ✓Update Service Worker
- ✓Performance & Gotchas
- ✓Best Practice

Offline caching

传统 webapp 都假设网络是可以被访问

通过 HTTP 访问 HTML 文档，后续 HTTP 请求下载子资源

这使得 webapp 相对于其它技术栈如 native app 处于劣势

离线缓存优势：

✓离线使用提升体验

✓数据在本地速度更快

✓减轻服务器负载

AppCache

SW 之前的一种解决方案是 HTML5 (Section 7.9) 中引入的 AppCache。W3C 决定 AppCache 仍然保留在 HTML 5.0 Recommendation 中，在 HTML 后续版本中移除。WHATWG HTML5 作为 Live Standard，也将 AppCache 标注为 Discouraged 并引导至 Service Worker。

AppCache 存在的问题：

✓文件总是直接从缓存中加载无论是否离线（Firefox 的实现可以先从网络取）

✓依靠 manifest 文件变化更新，一旦变化全量更新（也会走 HTTP 缓存）

✓不够灵活，更新资源需要二次刷新才被采用

✓对多页应用不够友好

✓...

https://alistapart.com/article/application-cache-is-a-douchebag

SW introduction

SW 是什么？

Web worker 中的一种，浏览器后台运行的脚本

Network ⇔ Service worker(proxy server?) ⇔ Document renderer

SW 能做什么？

- ✓Offline caching
- ✓Handling push notifications
- ✓Background data synchronization(only Chrome)
- ✓Responding to resource requests from other origins
- ✓Receiving centralized updates to expensive-to-calculate data (e.g., geolocation or gyroscope)

SW introduction

SW VS AppCache

AppCache 能解决掉离线缓存问题但是不够灵活，SW 是高度程序化的，它给 Web 开发者提供了很多接口

之前提到 AppCache 在设计上存在的缺陷可能导致无法被修复，SW 可以避免这些问题，W3C 标准推荐使用 SW 代替掉 AppCache

SW 特点

- ✓Event-driven background processing
- ✓Time-limited script contexts that run at an origin.
- ✓Extensible

SW introduction

SW 当前状态

标准制定 W3C Service Workers Working Group

Live document Service Workers Nightly

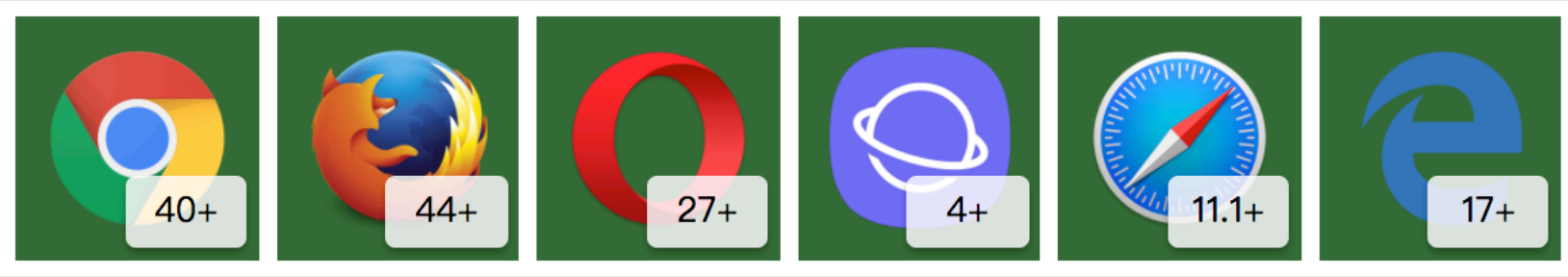
Working Draft Service Workers 1 (20171102)

working draft(WD) -> last call working draft(LCWD) -> candidate recommendation(CR) -> proposed recommendation(PR) -> recommendation(REC)

SW introduction

SW 能用了么？

navigator.serviceWorker



Edge 17+, 2018/04/30

Safari 11.1+, 2018/02/07

Other APIs

Promise / Fetch / Request & Response / Cache ...

https://jakearchibald.github.io/isserviceworkerready/index.html

SW model

- ✓Web Worker & Service Worker
- ✓Service Worker Registration
- ✓Service Worker Client
- ✓Task Source
- ✓Lifecycle

SW model – Web Worker

Web Worker

Web Worker 为 Web 内容在后台线程中运行脚本提供了一种简单的方法，线程可以执行任务而不干扰用户界面，workers 运行在另一个全局上下文中，不同于当前的window. 因此，使用 window快捷方式获取当前全局的范围 (而不是self) 在一个 Worker 内将返回错误

Web Worker 只能使用 JavaScript 功能子集：

- ✓navigator object
- ✓location object (read only)
- ✓XMLHttpRequest setTimeout()/clearTimeout() 和 setInterval()/clearInterval()
- ✓应用缓存
- ✓使用 importScripts() 方法导入外部脚本
- ✓生成其他 Web Worker

SW model – Web Worker

Worker 无法使用：

✓DOM（非线程安全）

✓window object

✓document object

✓parent object

Service Worker

Web Worker => [dedicated worker, shared worker, service worker]

Dedicated Worker =>只能被创建它的 JS 访问，创建它的页面关闭，它的生命周期就结束

Shared Worker =>可以被同一域名下的 JS 访问，关联的页面都关闭时，它的生命周期就结束

SW model - SW

SW 主要关联的属性和事件

state => Which is one of parsed, installing, installed, activating, activated, and redundant. It is initially parsed.
script url => A url
type => "classic" or "module". Unless stated otherwise, it is "classic".
containing service worker registration => A service worker registration which contains itself.
lifecycle events => install and activate
functional events => fetch

SW 的生命周期和它的生命周期事件有关，UA 可能随时终止 SW：

✓SW 没有事件可以处理

✓检测到异常操作：例如无限循环以及处理事件时超出时间限制（关于时限规范没有做具体要求）

SW model – SW Registration

SW Registration 主要关联的属性

scope url => A url

installing worker => SW state is installing, It is initially set to null.

waiting worker => SW state is installed, It is initially set to null.

active worker => SW state is activating / activated, It is initially set to null.

last update check time => It is initially set to null.

update via cache mode => imports / all / none, It is initially set to imports(Firefox 貌似现在还没有实现)

UA 必须让 SW Registration 一直存在，除非它明确的被注销

SW model – SW Client

一个 service worker client 表示了一个环境

SW client可以是：

✓window client => 全局对象是 window

✓dedicated worker client => 全局对象是 DedicatedWorkerGlobalScope

✓shared worker client => 全局对象是 SharedWorkerGlobalScope

一个 worker client 可以是 dedicated worker client 或者 shared worker client

SW model – Task Source

以下 task source 是在 SW 中新增的

The handle fetch task source

在 SW 分发 fetch 事件的时候会使用该 task source

The handle functional event task source

在 SW 分发功能事件的时候会用到该 task source，比如 push 事件

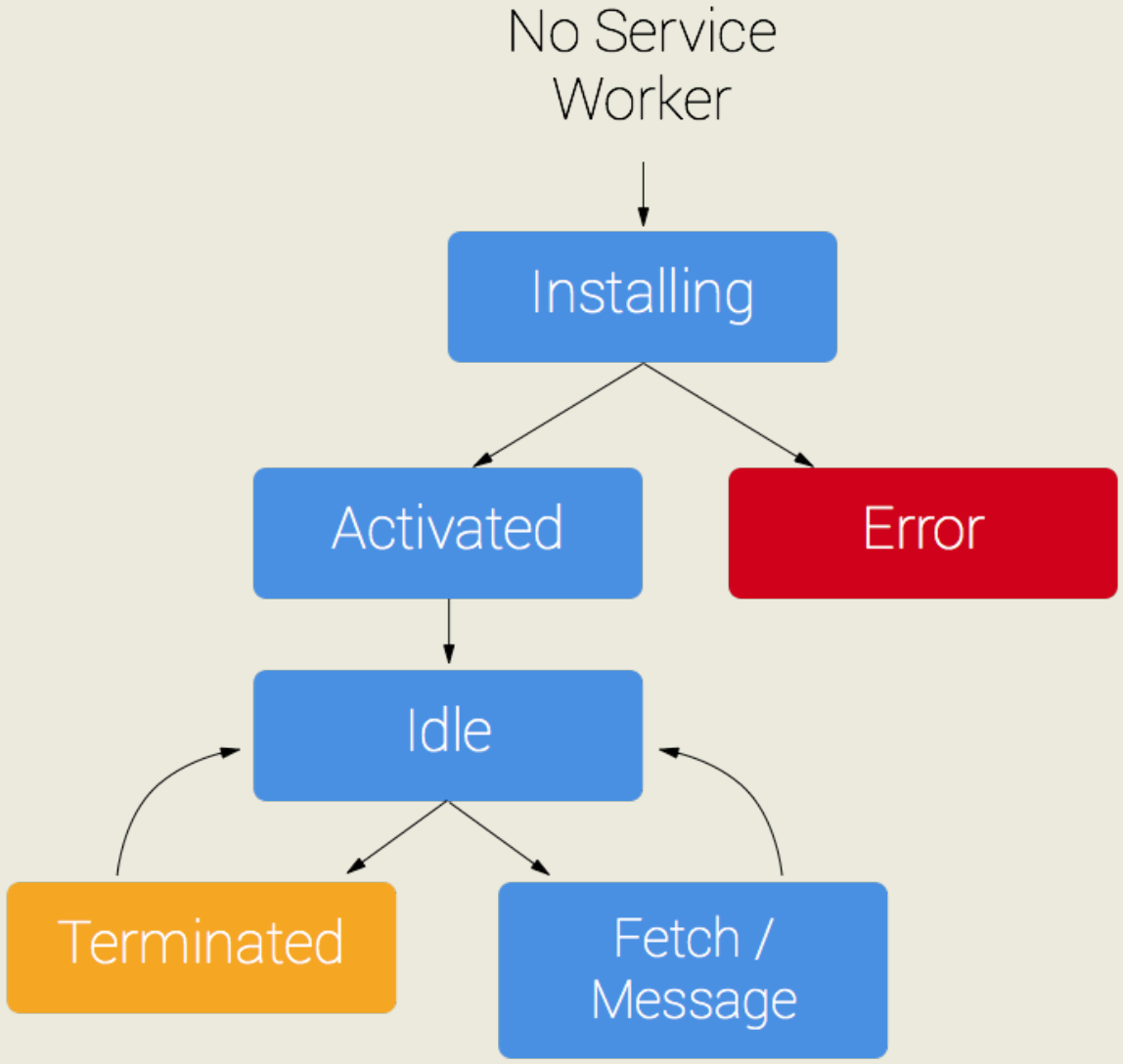
SW model – Task Source

SW Registration 拥有一个或者多个 task queue，它们来自 active worker event loop 中对应 task queue 的备份（备份的 task source 包括 handle fetch task source 以及 handle functional event task source）

当 SW 终止时，UA 将会复制 active worker 的 task 到 SW Registration 的 task queue，以及 re-queue 当时 active worker 对应的 task 到 active worker event loop 当 worker 再次激活时

不同于其他 task queue 属于一个 event loop，SW Registration task queue 不会被任何 event loop 处理

SW model – SW Lifecycle



Client Context

注册 service worker

```
// scope defaults to the path the script sits in
// "/" in this example
navigator.serviceWorker.register("/serviceworker.js").then(registration => {
    console.log("success!");
    if (registration.installing) {
        registration.installing.postMessage("Howdy from your installing page.");
    }
}, err => {
    console.error("Installing the worker failed!", err);
});
```

Client Context

ServiceWorker 接口

```
[SecureContext, Exposed = (Window, Worker)]
interface ServiceWorker : EventTarget {
  readonly attribute USVString scriptURL;
  readonly attribute ServiceWorkerState state;
    void postMessage(any message, optional sequence < object > transfer =[]);

    // event
    attribute EventHandler onstatechange;
};
ServiceWorker implements AbstractWorker;

enum ServiceWorkerState {
    "installing",
    "installed",
    "activating",
    "activated",
    "redundant"
};
```

https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorker

Client Context

ServiceWorkerRegistration 接口

```
interface ServiceWorkerRegistration : EventTarget {
    readonly attribute ServiceWorker ? installing;
    readonly attribute ServiceWorker ? waiting;
    readonly attribute ServiceWorker ? active;

    readonly attribute USVString scope;
    readonly attribute ServiceWorkerUpdateViaCache updateViaCache;

    [NewObject] Promise < void> update();
    [NewObject] Promise < boolean > unregister();

    // event
    attribute EventHandler onupdatefound;
};

enum ServiceWorkerUpdateViaCache {
    "imports",
    "all",
    "none"
};
```

https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerRegistration

Client Context

ServiceWorkerContainer 接口

```
[SecureContext, Exposed = (Window, Worker)]
interface ServiceWorkerContainer : EventTarget {
  readonly attribute ServiceWorker ? controller;
  readonly attribute Promise < ServiceWorkerRegistration > ready;

    [NewObject] Promise < ServiceWorkerRegistration > register(USVString
scriptURL, optional RegistrationOptions options);

    [NewObject] Promise < any > getRegistration(optional USVString clientURL =
"");
    [NewObject] Promise < sequence < ServiceWorkerRegistration >>
getRegistrations();

    void startMessages();


    // events
    attribute EventHandler oncontrollerchange;
    attribute EventHandler onmessage; // event.source of message events is
ServiceWorker object
    attribute EventHandler onmessageerror;
};
```

https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerContainer

Execution Context

ServiceWorkerGlobalScope 对象关联了一个 SW 的执行上下文，它提供了一个通用、事件驱动、限时执行的 SW 执行环境，如果一个 SW 在上下文中启动，启动或杀掉 SW 是由与他们关联的事件决定的而不是 SW client，任何同步的请求都不能在 SW 的执行上下文中发起

```
[Global = (Worker, ServiceWorker), Exposed = ServiceWorker]
interface ServiceWorkerGlobalScope : WorkerGlobalScope {
    [SameObject] readonly attribute Clients clients;
    [SameObject] readonly attribute ServiceWorkerRegistration registration;

    [NewObject] Promise < void> skipWaiting(); // 直接跳过 waiting 状态

    attribute EventHandler oninstall;
    attribute EventHandler onactivate;
    attribute EventHandler onfetch;

    // event
    attribute EventHandler onmessage; // event.source of the message events is
Client object
    attribute EventHandler onmessageerror;
};
```

https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerGlobalScope

CacheStorage & Cache

由于 Application Cache 和 LocalStorage 在设计上先天性不能满足 SW 缓存的需求，W3C 规范为 SW 定义新的缓存接口 CacheStorage & Cache，规范未明确对 Cache 容量大小限制，浏览器实现可能存在差异，前端开发中有必要对冗余 Cache 做处理

Cache 对象受到 CacheStorage 的管理，CacheStorage 对应到内核 ServiceWorkerCacheStorage 对象，它提供了很多 JS 接口来操作 cache

Cache 和 HTTP 缓存完全隔离，Cache 是持久化到文件系统中不会自动失效，只能显示删除，并且它不能被跨域共享

浏览器在实现 Cache API 上存在一些差异，Chromium 将其定位为 Application Cache，复用了大量的 Application Cache 代码，FireFox 基于 SQLite 为 Cache API 实现一套新的存储机制

CacheStorage

CacheStorage 用于对 Cache object 的存储，它提供了一个 SW 的所有命名缓存的主目录其他类型的 worker 或 window 范围可以访问（不必与 SW 一起使用它，即使这是定义它的规范），维护缓存名称到相应 Cache 对象的映射

CacheStorage 需要在 HTTPS origin 上使用，否则 promise reject SecurityError

APIs：

- ✓ CacheStorage.open() 获取一个 Cache 对象实例
- ✓ CacheStorage.match() 检查是否存在以 Request 为 key 的 Cache 对象
- ✓ CacheStorage.has() 是否存在指定名称的 Cache 对象
- ✓ CacheStorage.keys() 返回 Cache 对象的 cacheName 列表
- ✓ CacheStorage.delete() 删除指定 cacheName 的 Cache 对象

Cache

Cache 提供了已缓存的 Request / Response 对象的存储管理机制，以及一些列的 JS 接口

Cache.put() 将 Request / Response 放进指定的 Cache

Cache.add() 获取一个 Request 的 Response，并将 Request / Response 放进 Cache，add 可以看成 fetch(request) + Cache.put(request, response) 的语法糖

Cache.addAll() 用于获取一组 Request 的 Response，并将该组 Request / Response 对象体放进指定的Cache

Cache.keys() 用于获取 Cache 中所有 key。

Cache.match() 用于查找是否存在以 Request 为 key 的Cache 对象。

Cache.matchAll() 用于查找是否存在一组以 Request 为 key 的 Cache 对象组。

Cache.delete() 用于删除以 Request 为 key 的 Cache Entry，Cache 不会过期，只能显式删除

Security & Privacy

SW / SW Registration / Cache 都必须在一个安全的上下文中执行，否则抛出 DOMException

Chromium 的 security origin 包含以下模式（scheme，host，port）：

- ✓ (https, *, *)
- ✓ (wss, *, *)
- ✓ (*, localhost, *)
- ✓ (*, 127/8, *)
- ✓ (*, ::1/128, *)

出于开发角度也允许了在 localhost(127.0.0.1，::1) 中执行

SW 引入了新的持久性存储功能，为了保护用户免受任何未经批准的跟踪威胁，在用户打算清除这些数据的时候，应该给用户提供清除数据的功能

SW Lifecycle

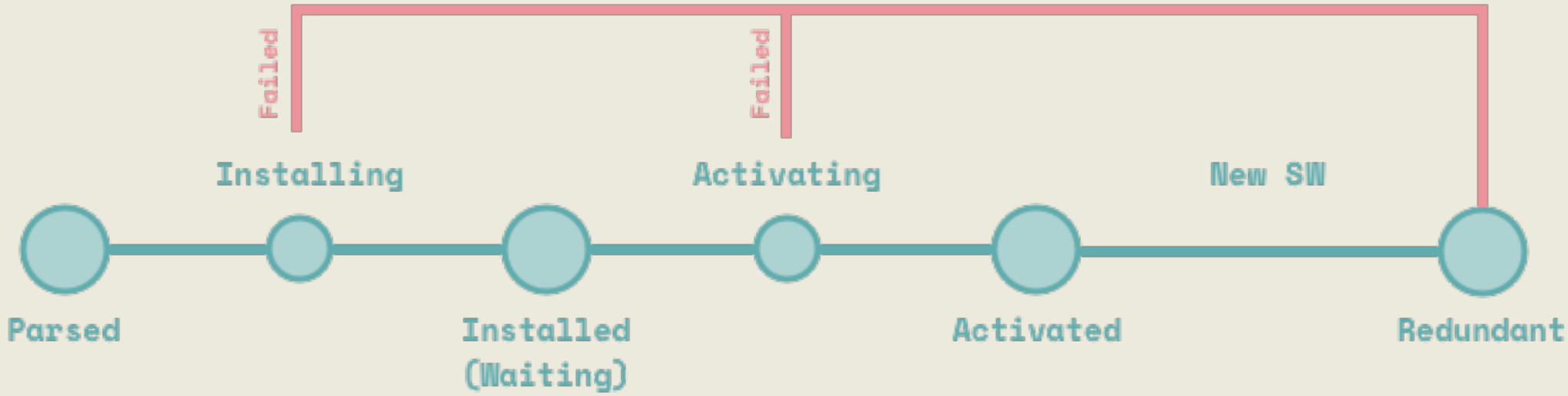
为什么需要 Lifecycle？

- ✓ 使离线缓存成为可能
- ✓ 在不中断当前 SW 工作的时候准备好新的 SW
- ✓ 确保在一个页面范围内始终由一个 SW 控制，或者没有 SW 控制
- ✓ 确保网站在一个时刻只有一个版本的 SW 在运行

最后一点非常重要，如果站点存在多个不同版本的 SW，并且在 SW 中有对数据的操作，SW 对数据操作的策略不同将会面临数据丢失或者损坏的风险

SW Lifecycle

SW Lifecycle 状态切换



SW Lifecycle - Install

Install

- ✓ install 事件是 SW 的第一个事件，并且只会被触发一次
- ✓ 会传递给 installEvent.waitUntil() 一个 promise，安装成功或者失败
- ✓ SW 将不会收到 fetch、push 事件直到 SW 安装成功状态变为 active
- ✓ 第一次安装 fetch 不会通过 SW同时页面也不会被 SW 控制，直到下次载入/刷新
- ✓ clients.claim() 可以覆盖默认行为（第一次不能控制 client），控制一个不受控制的页面

每当页面在调用 register 方法的时候会去下载 SW，每次当页面加载的时候，浏览器会知道 SW 是否已经被注册，如果下载或者在初始化执行的时候解析出错，都会 reject promise 并且丢弃 SW，错误会被报告在 chrome devtools 的 SW 面板

SW Lifecycle - Install

在页面注册完 SW 后，SW 会触发 install 事件，在 install 事件里面可以缓存 SW 想要缓存的文件

SW 会去下载缓存的文件，如果下载失败，整个 SW 安装就会失败，所以如果缓存文件列表里面的资源过多的话将会增加 SW 安装失败的概率

在 install 事件里面进行缓存只是常见的用法，也可以在回调中做其他事情，或者根本不为 SW 设置install 监听器

如果更新了 sw.js 浏览器将会认为这是一个全新的 SW，这个 SW 也有自己的 install 事件

SW Lifecycle - Activate

一旦 SW 已经准备好控制一个 client，以及可以开始处理 push、sync 事件，SW 将会得到一个 activate 事件，但是这并不代表调用 register() 的页面马上就会被 SW 所控制，需要刷新之后才会被控制（install 事件里面也是如此）

在 activate 事件里面也可以通过 clients.claim() 来主动控制不受控制的页面（无需刷新），一般情况下不需要这么做

SW 的生命周期事件 =>

Update SW

如果需要更新 SW，会经历以下步骤：

- ✓ 更新 sw.js 文件，UA 会在后台重新下载 sw.js 文件并和原来的 sw.js 进行逐字节比较来看是不是有更新
- ✓ 新的 SW 会开始安装并触发 SW 的 install 事件
- ✓ 这个时候上一个版本的 SW 对页面的控制仍然有效，页面的事件会被旧版本 SW 响应到，新版本的 SW 则会进入 waiting 的状态
- ✓ 当所有被旧版本 SW 控制的页面被关闭的时候，旧版本 SW 将会被杀掉新版本 SW 将会获得页面控制权
- ✓ 当新版本 SW 获得控制权之后，SW 的 activate 事件将会被触发

Update SW

手动触发 SW 更新：

Service Worker 标准中给出了 ServiceWorkerRegistration.update() 方法，该方法将会让 UA 重新下载 Service Worker 如果 sw.js 发生了变化，如果 SW 距离的上一次更新超过了 86400s（一天），将会绕过 HTTP 缓存重新下载 SW

UA 检查更新：

- ✓ 第一个访问到 SW scope 控制的页面
- ✓ 距离上次更新检查超过 86400s
- ✓ push / sync 事件发生
- ✓ sw.js 文件缓存时间超出 header max-age 设置的值，如果大于一天按一天算

Update SW

Cache 管理

一般情况下我们会在 activate 事件回调中进行 cache 管理，这样做的原因是在新版本的 SW 中你可能想对旧版本 SW 缓存进行清除，如果在新版本 SW install 回调中做了这件事，新的 SW 还没有被激活，当前打开的页面还是被旧版本的 SW 所控制，如果 cache 被新的 SW 更改，旧版本 SW 可能不能正常的使用被更改后的 cache

Performance & Gotchas

SW 启动

1. 触发启动流程

    在访问一个有 SW 的文档时，dispatch fetch，触发 SW 启动流程

2. 分派线程

    SW 启动之前，向 UI 线程申请分配一个线程

3. 加载 sw.js

    执行 sw.js 文件加载流程

4. 启动 SW

    sw.js 加载完成，触发 SW 启动，包括创建 ServiceWorkerGlobalScope 初始化上下文和执行 JS

5. 回调启动完成

SW 启动过程比较复杂，线程之间频繁切换，存在一定的性能开销

Performance & Gotchas

引导启动 SW 可能对页面初次加载性能造成影响，尤其在低端移动设备

主要体现在：

SW 启动占用 CPU 和内存开辟新线程，可能会影响到页面渲染

SW 中缓存资源会抢占带宽，在移动端很多时候带宽是有限的，应该先让文档关键资源得到下载

所以在注册 SW 时可以考虑延迟注册，比如在页面 load 事件中注册，或者页面启动时有大量的动画，可以在动画结束时。总之考虑到不要影响主资源下载和渲染，SW 可以在闲时注册

Best Practice

✓ 使用 SW 进行简单的资源缓存

http://localhost:8000/cache/

http://localhost:8000/cache/scope.html

✓ 不影响安装的资源预缓存

在 install 过程中缓存大量数据可能导致 SW 安装失败，可以根据资源优先级先缓存重要资源，再缓存次要资源

http://localhost:8000/cache-priority/

✓ 渐进式缓存

在 install 事件中没有缓存，可以在 fetch 的时候缓存

http://localhost:8000/cache-progressive/index.html

Best Practice

- ✓ 缓存优先

  先匹配 Cache 中的内容，失败 fetch 线上资源

- ✓ 网络优先

  先 fetch 线上数据，fallback 缓存数据

- ✓ Service side worker render & compile

  http://www.zhangxinxu.com/wordpress/2018/04/service-worker-client-online-html-css-complie/

Reference

https://www.w3.org/TR/service-workers-1/

https://developers.google.com/web/fundamentals/primers/service-workers/lifecycle

https://developers.google.com/web/fundamentals/primers/service-workers/high-performance-loading

https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API

https://serviceworke.rs/

https://x5.tencent.com/tbs/guide/serviceworker.html

https://www.html5rocks.com/zh/tutorials/workers/basics/

https://alistapart.com/article/application-cache-is-a-douchebag

https://zhuanlan.zhihu.com/p/27264234

Thanks