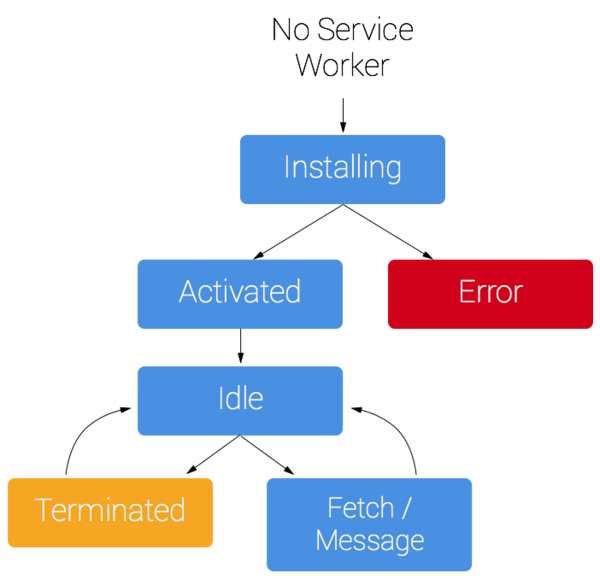
service worker运行在独立的浏览器线程上，无法操作window和dom。用self来指代service worker的上下文。在对应的service worker文件里，this指代的是文件本身。



serviceWorker的生命周期

代码示例

// 用于标注创建的缓存，也可以根据它来建立版本规范

const CACHE\_NAME = "local\_cache\_v1.2.0"; //缓存版本由gulp动态生成

// 列举要默认缓存的静态资源，一般用于离线使用

const urlsToCache = [

//这里缓存的必须是不会有变化的文件，如果文件内的变量影响其他文件加载就不能放在这里

"/apps/shared/version.js"

];

//不需要缓存的静态资源

const excludeFiles = [

"/api/"

]

const nerverToCache = [

"/api/",

]

//WorkerLocation, service worker专有对象,表明serviceworker文件绝对地址

const location = self.location;

const origin = location.origin;

// self 为当前 scope 内的上下文

//脚本安装时触发install事件

self.addEventListener('install', event => {

// event.waitUtil 用于在安装成功之前执行一些预装逻辑

// 但是建议只做一些轻量级和非常重要资源的缓存，减少安装失败的概率

// 安装成功后 ServiceWorker 状态会从 installing 变为 installed

// 使用 cache API 打开指定的 cache 文件

// event.waitUntil(

// // 使用 cache API 打开指定的 cache 文件

// caches.open(CACHE\_NAME).then(cache => {

// console.log(cache);

// // 添加要缓存的资源列表

// return cache.addAll(urlsToCache);

// })

// );

//人为控制cache更新方式2：SW安装时跳过安装阶段，先删除本地cache，直接加载线上最新资源。

//跳过安装阶段，直接进入active阶段

event.waitUntil(self.skipWaiting())

});

//安装完成，触发激活状态

self.addEventListener('activate', event => event.waitUntil(

Promise.all([

// 更新客户端

clients.claim(),

// 清理旧版本

caches.keys().then(cacheList => Promise.all(

cacheList.map(cacheName => {

if (cacheName !== CACHE\_NAME) {

caches.delete(cacheName);

}

})

))

])

.then(

()=>{

caches.open(CACHE\_NAME).then(cache => {

// 添加要缓存的资源列表

return cache.addAll(urlsToCache);

})

}

)

.then(

() =>{

PostMessageToClient({type:"reload", targetWindow: "top"})

}

)

.catch(

(err) =>{

console.error(err);

}

)

));

//通过fetch事件拦截浏览器的HTTP/HTTPS请求

// 联网状态下执行

function onlineRequest(fetchRequest) {

//console.log("获取在线资源", fetchRequest.url)

// 使用 fecth API 获取资源，以实现对资源请求控制

//Fix chrominu bug;

if (fetchRequest.cache === 'only-if-cached' && fetchRequest.mode !== 'same-origin') {

return;

}

return fetch(fetchRequest).then(response => {

// 在资源请求成功后，将 image、js、css 资源加入缓存列表

//对于非正常返回值和非资源请求，全部按原始值返回，不做缓存。!response.headers.get('Content-type') 是为了排除api接口请求

if (!response || response.status !== 200 || !response.headers.get('Content-type') || !response.headers.get('Content-type').match(/image|javascript|text\/css|text\/html/i)) {

return response;

}

for(let i in nerverToCache){

let files = nerverToCache[i];

//所有接口请求都走线上

if (fetchRequest.url.match( new RegExp(files,"i"))) {

return response;

}

}

for(let i in excludeFiles){

let fileName = excludeFiles[i]

if (fetchRequest.url.match( new RegExp(fileName,"i"))) {

return response;

}

}

//非本域名下的请求不缓存

if (!fetchRequest.url.match( new RegExp(origin,"i"))) {

return response;

}

const responseToCache = response.clone();

caches.open(CACHE\_NAME)

.then(function (cache) {

//console.log("缓存在线资源", fetchRequest.url)

cache.put(fetchRequest, responseToCache);

})

return response;

}).catch((err) => {

console.error("online", err)

// 获取失败，离线资源降级替换

offlineRequest(fetchRequest);

});

}

// 离线状态下执行，降级替换

function offlineRequest(request) {

//console.log("获取离线资源", request.url)

// 使用离线图片,静态页面样式表，js

if (request.url.match(/\.(png|gif|jpg|html|css|js)/i)) {

let filePath = request.url.replace(origin, "")

return caches.match(filePath);

}

}

//Service Worker监听http/https，通过fetch加载文件

self.addEventListener('fetch', event => {

//console.log("fetch", event.request.url);

event.respondWith(

caches.match(event.request,{ignoreSearch:true})

.then(

(hit) => {

// 返回缓存中命中的文件,但是对于ajax请求不能从本地读取任何缓存

for(let i in nerverToCache){

let files = nerverToCache[i];

//所有接口请求都走线上

if (event.request.url.match( new RegExp(files,"i"))) {

break;

}else{

if (hit) {

return hit;

}

}

}

const fetchRequest = event.request.clone();

if (navigator.onLine) {

// 如果为联网状态

return onlineRequest(fetchRequest);

} else {

// 如果为离线状态

return offlineRequest(fetchRequest);

}

},

(err) =>{

console.log("caches.match -> hit err", err);

}

)

.catch(

(err) =>{

console.log("fetch err", err);

throw err;

}

)

);

});

//Notification -- 消息订阅和推送

//在 Service Worker 中通过监听 push 事件对推送的消息作处理

self.addEventListener('push', function(event) {

// 读取 event.data 获取传递过来的数据，根据该数据做进一步的逻辑处理

const obj = event.data.json();

// 逻辑处理示例

if(Notification.permission === 'granted' && obj.action === 'subscribe') {

self.registration.showNotification("Hi：", {

body: '订阅成功 ~',

icon: 'static/assets/logo.png',

tag: 'push'

});

}

});

//PostMessage -- 页面和service worker之间的数据传输

//向特定的页面传递数据

function PostMessageToClient(msg){

self.clients.matchAll().then(clientList => {

clientList.forEach(client => {

client.postMessage(msg);

})

});

}

//接收页面传来的数据

self.addEventListener('message', function(ev) {

//TODO -- 待补充需求

// PostMessageToClient("haha")

});

serviceWorker安装

/\*\*

\* 注册安装Service Worker

\*/

const Version = "1.0.3"; //当前版本号

//Service Worker 安装、注册

function install() {

//TOFix - xp 下不安装

var uaString = navigator.userAgent.toLowerCase();

// var osV = uaString.substr(uaString.indexOf("windows nt ")+11, 3);

if (uaString.indexOf('wisdomclass\_xp') != -1) {

return true;

}

if ('serviceWorker' in navigator) {

navigator.serviceWorker.register('/sw.js', {

scope: '/'

})

.then(

(registration) => {

//取消注册

console.log('ServiceWorker 注册成功！作用域为: ', registration.scope);

//人为控制cache更新方式1：安装时设置版本号，根据版本号做本地更新比对

if (localStorage.getItem("ws\_version") !== Version) {

//Note -- 注意 update是一个异步操作，不能立即得到serviceWorker.controller，注意回调的顺序

registration.update().then(

() => {

localStorage.setItem("ws\_version", Version);

bindAct();

}

)

} else {

bindAct();

}

}

)

.catch(err => console.log('ServiceWorker 注册失败: ', err));

// 取消所有serviceWorker的

// navigator.serviceWorker.getRegistrations()

// .then(function (registrations) {

// for (let registration of registrations) {

// registration.unregister();

// }

// });

}

}

function bindAct() {

//ServiceWorker注册成功后才能向ServiceWorker发送数据,但是此时ServiceWorker.controller不一定建立完毕，可能处于activing中

sendMessageToServiceWorker('hello sw!');

getPostMessage();

}

//PostMessage -- 向service worker发送数据

//与iframe传递数据的PostMessage方法的异同，同属于HTML5的PostMessage方法

function sendMessageToServiceWorker(msg) {

const controller = navigator.serviceWorker.controller;

if (!controller) {

return;

}

controller.postMessage(msg, []);

}

//接受PostMessage数据

//serviceWorker,worker,window 三种场景下的postMessage主体是不一样的，在不同场景下要取对应的对象窗体

function getPostMessage() {

navigator.serviceWorker.addEventListener('message', (e) => {

console.log("从serviceworker收到消息", e)

// 这里对收到的消息做origin验证，根据不同类型做不同的处理

if(e.data && e.data.targetWindow =="top" && e.data.type == "reload"){

console.log("reload page")

window.top.location.reload();

}

});

}

install();

存在的问题：

1、拦截fetch再去匹配本地缓存是否有缓存数据时，使用caches.match(event.request,{ignoreSearch:true})

导致serviceworker执行的时间太长，caches,match耗费了大量的时间

应该先找到缓存的cache在去match。

caches.open(CACHE\_NAME).then(function (cache) {

return cache.match(request).then(function (response) {

if (response) {

return response;

}

return fetch(request, option).then(function (response) {

if (response.ok && response.status != 206) {

//这里拿到chche，决定是否缓存

cacheResponse(cache, request, response);

}

return response;

});

});

})

//重新创建response是关键

function cacheResponse(cache, request, response) {

// Response objects are read-only, so to add our custom header, we need to

// recreate the object.

var init = {

status: response.status,

statusText: response.statusText,

headers: {

'X-Shaka-From-Cache': true

}

};

response.headers.forEach(function (value, key) {

init.headers[key] = value;

});

// Response objects are single use. This means we need to call clone() so

// we can both store the ArrayBuffer and give the response to the page.

return response.clone().arrayBuffer().then(function (ab) {

cache.put(request, new Response(ab, init));

});

}