



Service Worker 实践指南

[@hijiangtao](#)



前言

当我们谈到 Service Worker 的时候，往往是和 Progressive Web Apps 绑在一起说出来的，但即便不用来做渐进式增强 Web 应用，我们也可以利用 Service Worker 的全局拦截能力，来自定义一个满足我们需要的全局代理。



TOC

1 / 概览

2 / 安装与控制

注册与注销

受控状态检查

3 / 通信与存储

线程间通信

全局变量

持久化存储

4 / 调试与其他

环境判断

运行与调试

5 / 总结



1 / 概览

1. Service Worker 定义
2. Service Worker 特点
3. Can I Use Service Worker



概览

Service Worker 定义



未连接到互联网

请试试以下办法：

- 关闭飞行模式
- 开启移动数据网络或 WLAN
- 检查您所在区域的网络信号

ERR_INTERNET_DISCONNECTED

它是一个服务器与浏览器之间的中间人角色，如果网站中注册了 Service Worker 那么它可以拦截当前网站所有的请求，进行判断（需要编写相应的判断程序）。

如果需要向服务器发起请求的就转给服务器，如果可以直接使用缓存的就直接返回，通过它从而大大提高浏览体验。



概览

Service Worker 特点

必须运行在 HTTPS 协议下

有自己完全独立的执行上下文。一旦被安装成功就永远存在，除非线程被程序主动解除

无法直接操作 DOM, UI 的渲染工作必须只能在主线程完成

基于 Web Worker 并拥有离线存储能力

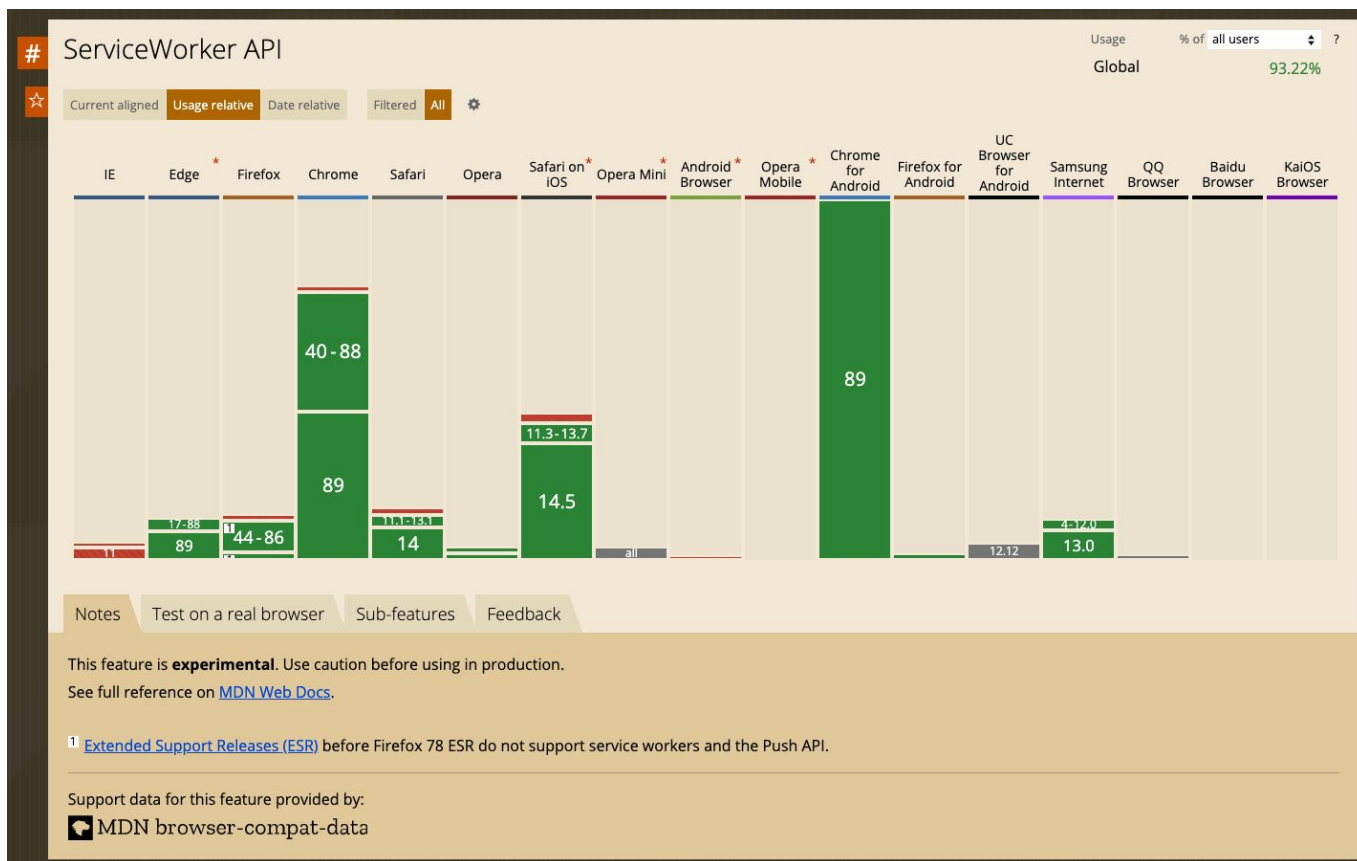
全局拦截，可以拦截并代理全局请求，可以处理请求的返回内容，开发者可控

事件驱动并拥有生命周期

支持推送与版本控制

概览

Can I use



<https://caniuse.com/?search=ServiceWorker>

Google Developers

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





安装与控制

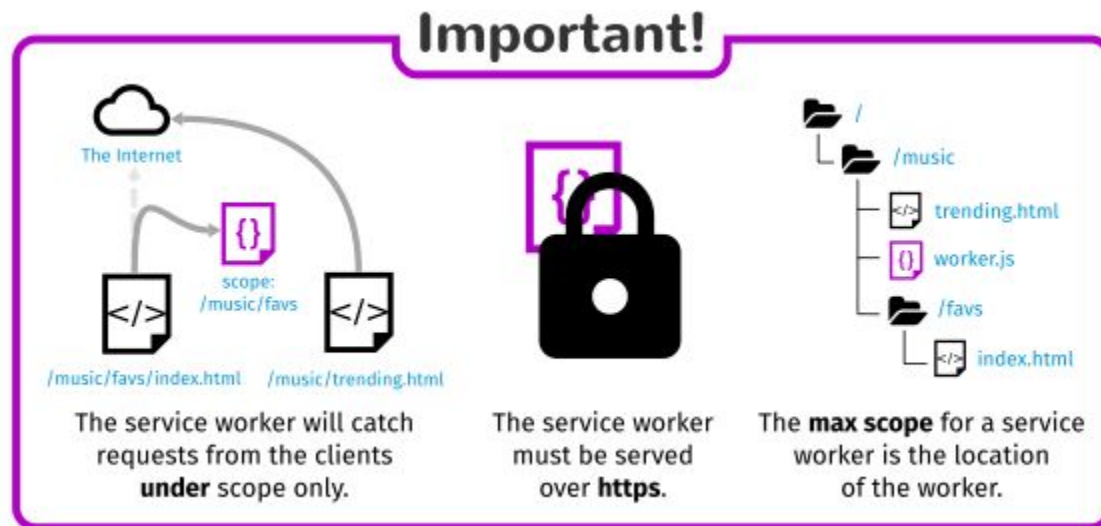
2.1 / Service Worker 的注册与注销



Service Worker 注册

```
navigator.serviceWorker
  .register(swFilePath as string)
  .then((reg) => {
    tlog.info('Registration succeeded. Scope is ' + reg.scope)
  })
  .catch((error) => {
    tlog.error('Registration failed with ' + error)
  })
```

Service Worker 注册





Service Worker 注销

```
navigator.serviceWorker.getRegistrations().then((registrations) => {  
  for (let registration of registrations) {  
    if (  
      registration.active?.scriptURL ===  
      `${location.origin}/${DEFAULT_REPLAY_SW_FILENAME}`  
    ) {  
      registration.unregister()  
    }  
  }  
})
```



安装与控制

2.2 / Service Worker 受控状态检测

通过 `navigator.serviceWorker.controller` 可以获得当前控制页面的 Service Worker 实例，这个实例是一个 ServiceWorker 对象，通过这个对象，你可以读取 `scriptURL` 获得序列化脚本的实际 URL 等等。



Ready 并不是真正的 ready

```
navigator.serviceWorker?.ready
  .then(() => {
    navigator.serviceWorker.controller?.postMessage(
      {
        ...
      }
    )
  })
```



Workbox

```
// somewhere
import {Workbox} from 'workbox-window';
const workbox = new Workbox('/service-worker.js');

// somewhere else
await workbox.active
await workbox.controlling
```



Implement with Promise

```
const p = new Promise(r => {  
  if (navigator.serviceWorker.controller) return r();  
  navigator.serviceWorker.addEventListener('controllerchange', e => r());  
});
```




Implement with Promise

```
window._controlledPromise = new Promise(function(resolve) {  
  // Resolve with the registration, to match the .ready promise's behavior.  
  var resolveWithRegistration = function() {  
    navigator.serviceWorker.getRegistration().then(function(registration) {  
      resolve(registration);  
    });  
  };  
  
  if (navigator.serviceWorker.controller) {  
    resolveWithRegistration();  
  } else {  
    navigator.serviceWorker.addEventListener('controllerchange', resolveWithRegistration);  
  }  
});
```



通信与存储

3.1 / 线程间通信

postMessage, MessageChannel & MessagePort



按照使用场景分类

1. 客户端可能希望向 Service Worker 发送消息, 一对一(单播)场景
2. Service Worker 可能希望将一些信息发送给与他传递消息的客户端, 依旧是单播场景
3. Service Worker 可能希望向其控制下的每个客户端都发送信息, 一对多(广播)消息
4. Service Worker 可能希望向发起请求的客户端发送消息, 单播场景



场景一 / 一对一单播

发送方如下调用：

```
worker.postMessage(data)
```

而接收方如下监听即可：

```
self.addEventListener('message', function handler(event: MessageEvent<any>)  
    console.log(event.data)  
})
```



场景二 / Service Worker 消息回传客户端

Channel Messaging API的 `MessageChannel` 接口允许我们创建一个新的消息通道, 并通过它的两个 `MessagePort` 属性发送数据。



场景二 / Service Worker 消息回传客户端

客户端

```
const messageChannel = new MessageChannel();

messageChannel.port1.addEventListener('message', replyHandler);
navigator.serviceWorker.controller.postMessage(data, [messageChannel.port2]);

function replyHandler (event) {
  console.log(event.data); // this comes from the ServiceWorker
}
```



场景二 / Service Worker 消息回传客户端

Service Worker 侧

```
self.addEventListener('message', function handler(event) {
  self.messagePort = event.ports[0]

  postMessageToClientViaMessagePort({
    type: 'Test',
    data: JSON.stringify({}),
  })
})

export const postMessageToClientViaMessagePort = async (
  data: any,
) => {
  const port = self.messagePort

  if (!port) {
    console.error('Invalid MessagePort')
    return
  }

  port.postMessage(data)
}
```



场景二 / Service Worker 消息回传客户端

基于 MessagePort 通信

```
self.addEventListener('message', function handler(event) {  
  if (!self.messagePort) {  
    self.messagePort = event.ports[0]  
  
    self.messagePort.onmessage =  
      (e)=>console.log('Got message from MessagePort')  
  }  
  
  // ...  
})
```




场景三 / Service Worker 广播

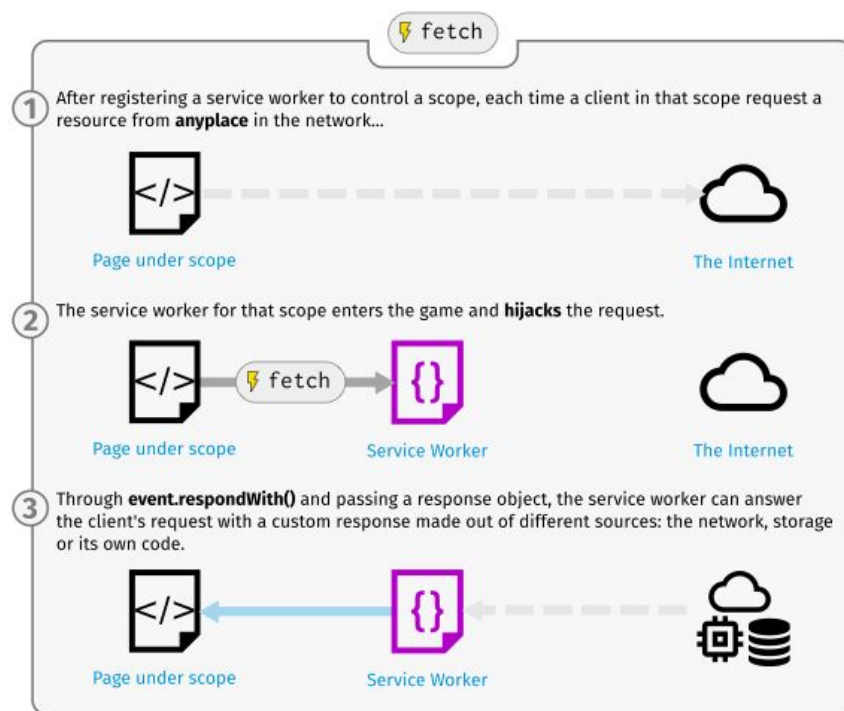
```
// 广播消息
self.clients.matchAll()
  .then(all =>
    all.map(
      client => client.postMessage(data)
    )
  );
```

场景四 / 响应发起请求的客户端

Service Worker 中的事件



场景四 / 响应发起请求的客户端



场景四 / 响应发起请求的客户端

Service Worker 侧

```
self.on('fetch', function handler (event: FetchEvent) {  
  fetch(event.request)  
    .then(response => response.json())  
    .then(function (data) {  
      self.clients  
        .match(event.clientId)  
        .then(client => client.postMessage(data));  
    });  
});
```



场景四 / 响应发起请求的客户端

客户端

```
window.navigator.serviceWorker.onmessage = (event) => {  
  // ...  
}
```

如上列出了线程间通信的所有枚举情况，但实际如何组合使用，还要看各自的业务场景特征，比如有些场景需要保证从导航开始的所有请求拦截与通信，而有些场景只需要保证用户交互产生的数据可以得到监控与传输即可。



通信与存储

3.2 / 全局变量

全局变量

Service Worker 在安装时会执行一遍 Service Worker 入口文件的所有逻辑, 而其中各类事件监听器是按需调用的。

如果为了保持一些状态而在 Service Worker 入口文件中定义了一些全局变量, 那么需要注意的是当你关闭页面或判断需要执行清理逻辑时, 对其进行重置。



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

Worker lifecycle

INSTALLING

This stage marks the beginning of registration. It's intended to allow to setup worker-specific resources such as offline caches.

 `install`

-  Use `event.waitUntil()` passing a promise to extend the installing stage until the promise is resolved.
-  Use `self.skipWaiting()` anytime before activation to skip installed stage and directly jump to activating stage without waiting for currently controlled clients to close.



INSTALLED



The service worker has finished its setup and it's waiting for clients using other service workers to be closed.



ACTIVATING

There are no clients controlled by other workers. This stage is intended to allow the worker to finish the setup or clean other worker's related resources like removing old caches.

 `activate`

-  Use `event.waitUntil()` passing a promise to extend the activating stage until the promise is resolved.
-  Use `self.clients.claim()` in the activate handler to start controlling all open clients without reloading them.



ACTIVATED

The service worker can now handle functional events.



REDUNDANT

This service worker is being replaced by another one.





全局变量

Tenet

```
useEffect(() => {
  TenetReplay.start().then(() => {
    navigator.serviceWorker.ready.then((reg) =>
      reg.active?.postMessage({
        type: ServiceWorkerMsgType.TenetSwitch,
        onTenet: true,
      }),
    )
  })
})

return () => {
  TenetReplay.end()

  navigator.serviceWorker.ready.then((reg) =>
    reg.active?.postMessage({
      type: ServiceWorkerMsgType.TenetSwitch,
      onTenet: false,
    }),
  )
}
}, [])
```



通信与存储

3.3 / 持久化存储



持久化存储

无法使用如下方案

1. XHR
2. localStorage



持久化存储

建议

1. IndexedDB
2. cache API



调试与其他

4.1 / 客户端环境判断



环境判断

```
export const postMessageToClient = async (
  event: FetchEvent
) => {
  if (!event.clientId) {
    console.error('No available clientId for postMessage')
    return
  }

  const client: Client = await (self as any).clients.get(event.clientId)

  let windowClientId = '';
  let iframeClientId = '';

  // 一级页面
  if (client.frameType === "top-level") {
    windowClientId = client.id

    // iframe
  } else if (client.frameType === "nested") {
    iframeClientId = client.id
  }

  client.postMessage({windowClientId, iframeClientId})
}
```



环境判断

应用场景

1. Service Worker 对不同环境下的请求做差别处理
2. 主线程中接收 Service Worker 传来的消息时, 也需要注意事件监听器挂载的 window

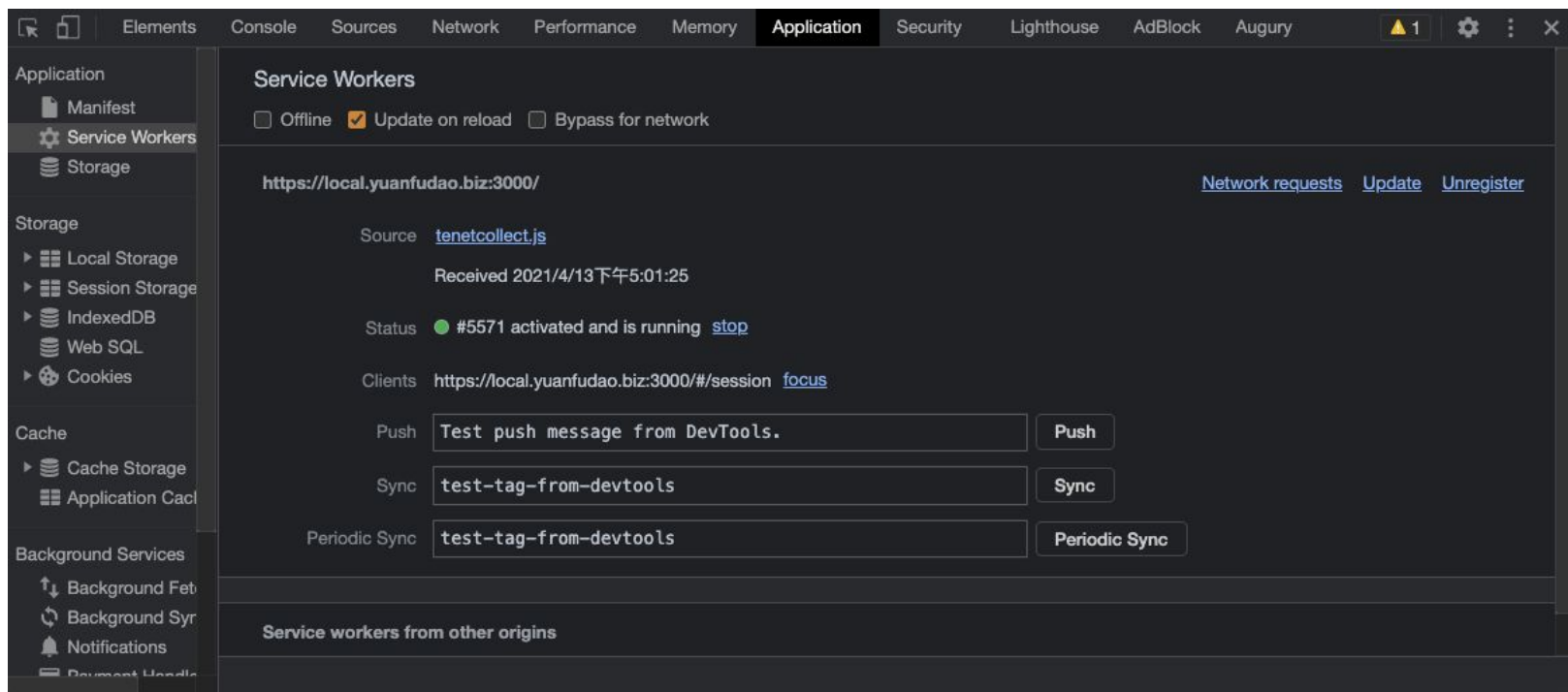


调试与其他

4.2 / 运行与调试



开发者调试工具





代码调试

<chrome://inspect/#service-workers>

DevTools

Devices

Pages

Extensions

Apps

Shared workers

Service workers

Other

Service workers

Service Worker <https://local.yuanfudao.biz:3000/tenetcollect.js> <https://local.yuanfudao.biz:3000/tenetcollect.js>

[inspect](#) [terminate](#)



Not work?

网络请求没有经过 Service Worker ?

首先, 可以确认下是否为首次加载 Service Worker

其次, 要看下 disable cache 是不是被你勾上了

最后, 检查下 Bypass for network 是不是被你勾上了

当然, 也可能是触发了 chrome 内部运行 bug 的边界条件



5 / 总结

1. 利用 Service Worker 的全局拦截能力，可以自定义一个满足需要的全局代理，这也是我们正在做的事情。
2. 在注册并使用 Service Worker 时，需要提前检查保证页面受控，行为符合预期。
3. 线程间通过 `postMessage` 通信，但要达到不同场景的需求，需要组合策略。
4. Service Worker 可以定义全局变量，但要注意及时清理，当然，可以使用 `cache API` 以及 `IndexedDB` 作为持久化存储方案。
5. 提前判断好客户端环境、利用好开发者调试工具，对 Service Worker 的编程体验会有很大帮助。

博客

<https://hijiangtao.github.io/2021/04/13/Service-Worker-Practical-Notes/>



Thank you.

