```
.then(() => caches.delete('v1'))
.then(() => caches.has('v1'))
.then(console.log);  // false

// 打开缓存 v1、v3 和 v2
// 检查当前缓存的键
// 注意：缓存键按创建顺序输出

caches.open('v1')
.then(() => caches.open('v3'))
.then(() => caches.open('v2'))
.then(() => caches.keys())
.then(console.log); // ["v1", "v3", "v2"]
```

CacheStorage 接口还有一个 match()方法，可以根据 Request 对象搜索 CacheStorage 中的所有 Cache 对象。搜索顺序是 CacheStorage.keys()的顺序，返回匹配的第一个响应：

```
// 创建一个请求键和两个响应值
const request = new Request('');
const response1 = new Response('v1');
const response2 = new Response('v2');

// 用同一个键创建两个缓存对象，最终会先找到 v1
// 因为它排在 caches.keys()输出的前面
caches.open('v1')
.then((v1cache) => v1cache.put(request, response1))
.then(() => caches.open('v2'))
.then((v2cache) => v2cache.put(request, response2))
.then(() => caches.match(request))
.then((response) => response.text())
.then(console.log); // v1
```

CacheStorage.match()可以接收一个 options 配置对象。下一节会介绍该对象。

2. Cache 对象

CacheStorage 通过字符串映射到 Cache 对象。Cache 对象跟 CacheStorage 一样，类似于异步的 Map。Cache 键可以是 URL 字符串，也可以是 Request 对象。这些键会映射到 Response 对象。

服务工作者线程缓存只考虑缓存 HTTP 的 GET 请求。这样是合理的，因为 GET 请求的响应通常不会随时间而改变。另一方面，默认情况下，Cache 不允许使用 POST、PUT 和 DELETE 等请求方法。这些方法意味着与服务器动态交换信息，因此不适合客户端缓存。

为填充 Cache，可能使用以下三个方法。

- put(request, response)：在键（Request 对象或 URL 字符串）和值（Response 对象）同时存在时用于添加缓存项。该方法返回期约，在添加成功后会解决。
- add(request)：在只有 Request 对象或 URL 时使用此方法发送 fetch()请求，并缓存响应。该方法返回期约，期约在添加成功后会解决。
- addAll(requests)：在希望填充全部缓存时使用，比如在服务工作者线程初始化时也初始化缓存。该方法接收 URL 或 Request 对象的数组。addAll()会对请求数组中的每一项分别调用 add()。该方法返回期约，期约在所有缓存内容添加成功后会解决。

与 Map 类似，Cache 也有 delete()和 keys()方法。这些方法与 Map 上对应方法类似，但都基于期约。

```
const request1 = new Request('https://www.foo.com');
const response1 = new Response('fooResponse');

caches.open('v1')
.then((cache) => {
  cache.put(request1, response1)
  .then(() => cache.keys())
  .then(console.log)  // [Request]
  .then(() => cache.delete(request1))
  .then(() => cache.keys())
  .then(console.log); // []
});
```

要检索 Cache，可以使用下面的两个方法。

- matchAll(request, options)：返回期约，期约解决为匹配缓存中 Response 对象的数组。
  - 此方法对结构类似的缓存执行批量操作，比如删除所有缓存在/images 目录下的值。
  - 可以通过 options 对象配置请求匹配方式，本节稍后会介绍。
- match(request, options)：返回期约，期约解决为匹配缓存中的 Response 对象；如果没命中缓存则返回 undefined。
  - 本质上相当于 matchAll(request, options)[0]。
  - 可以通过 options 对象配置请求匹配方式，本节稍后会介绍。

缓存是否命中取决于 URL 字符串和/或 Request 对象 URL 是否匹配。URL 字符串和 Request 对象是可互换的，因为匹配时会提取 Request 对象的 URL。下面的例子演示了这种互换性：

```
const request1 = 'https://www.foo.com';
const request2 = new Request('https://www.bar.com');

const response1 = new Response('fooResponse');
const response2 = new Response('barResponse');

caches.open('v1').then((cache) => {
  cache.put(request1, response1)
  .then(() => cache.put(request2, response2))
  .then(() => cache.match(new Request('https://www.foo.com')))
  .then((response) => response.text())
  .then(console.log)  // fooResponse
  .then(() => cache.match('https://www.bar.com'))
  .then((response) => response.text())
  .then(console.log); // barResponse
});
```

Cache 对象使用 Request 和 Response 对象的 clone()方法创建副本，并把它们存储为键/值对。下面的例子演示了这一点，因为从缓存中取得的实例并不等于原始的键/值对：

```
const request1 = new Request('https://www.foo.com');
const response1 = new Response('fooResponse');

caches.open('v1')
.then((cache) => {
  cache.put(request1, response1)
  .then(() => cache.keys())
  .then((keys) => console.log(keys[0] === request1))        // false
  .then(() => cache.match(request1))
  .then((response) => console.log(response === response1)); // false
});
```

Cache.match()、Cache.matchAll()和 CacheStorage.match()都支持可选的 options 对象，它允许通过设置以下属性来配置 URL 匹配的行为。

- cacheName：只有 CacheStorage.matchAll()支持。设置为字符串时，只会匹配 Cache 键为指定字符串的缓存值。
- ignoreSearch：设置为 true 时，在匹配 URL 时忽略查询字符串，包括请求查询和缓存键。例如，https://example.com?foo=bar 会匹配 https://example.com。
- ignoreMethod：设置为 true 时，在匹配 URL 时忽略请求查询的 HTTP 方法。比如下面的例子展示了 POST 请求匹配 GET 请求：

```
const request1 = new Request('https://www.foo.com');
const response1 = new Response('fooResponse');

const postRequest1 = new Request('https://www.foo.com',
                                 { method: 'POST' });

caches.open('v1')
.then((cache) => {
  cache.put(request1, response1)
  .then(() => cache.match(postRequest1))
  .then(console.log)  // undefined
  .then(() => cache.match(postRequest1, { ignoreMethod: true }))
  .then(console.log); // Response {}
});
```

- ignoreVary：匹配的时候考虑 HTTP 的 Vary 头部，该头部指定哪个请求头部导致服务器响应不同的值。ignoreVary 设置为 true 时，在匹配 URL 时忽略 Vary 头部。

```
const request1 = new Request('https://www.foo.com');
const response1 = new Response('fooResponse',
                               { headers: {'Vary': 'Accept' }});

const acceptRequest1 = new Request('https://www.foo.com',
                                   { headers: { 'Accept': 'text/json' } });

caches.open('v1')
.then((cache) => {
  cache.put(request1, response1)
  .then(() => cache.match(acceptRequest1))
  .then(console.log)  // undefined
  .then(() => cache.match(acceptRequest1, { ignoreVary: true }))
  .then(console.log); // Response {}
});
```

3. 最大存储空间

浏览器需要限制缓存占用的磁盘空间，否则无限制存储势必会造成滥用。该存储空间的限制没有任何规范定义，完全由浏览器供应商的个人喜好决定。

使用 StorageEstimate API 可以近似地获悉有多少空间可用（以字节为单位），以及当前使用了多少空间。此方法只在安全上下文中可用：

```
navigator.storage.estimate()
.then(console.log);

// 不同浏览器的输出可能不同：
// { quota: 2147483648, usage: 590845 }
```

根据 Service Worker 规范：

> 这些并不是确切的数值，考虑到压缩、去重和混淆等安全原因，该数字并不精确。

27.4.3 服务工作者线程客户端

服务工作者线程会使用 Client 对象跟踪关联的窗口、工作线程或服务工作者线程。服务工作者线程可以通过 Clients 接口访问这些 Client 对象。该接口暴露在全局上下文的 self.clients 属性上。

Client 对象支持以下属性和方法。

- id：返回客户端的全局唯一标识符，例如 7e4248ec-b25e-4b33-b15f-4af8bb0a3ac4。id 可用于通过 Client.get()获取客户端的引用。
- type：返回表示客户端类型的字符串。type 可能的值是 window、worker 或 sharedworker。
- url：返回客户端的 URL。
- postMessage()：用于向单个客户端发送消息。

Clients 接口支持通过 get()或 matchAll()访问 Client 对象。这两个方法都通过期约返回结果。matchAll()也可以接收 options 对象，该对象支持以下属性。

- includeUncontrolled：在设置为 true 时，返回结果包含不受当前服务工作者线程控制的客户端。默认为 false。
- type：可以设置为 window、worker 或 sharedworker，对返回结果进行过滤。默认为 all，返回所有类型的客户端。

Clients 接口也支持以下方法。

- openWindow(url)：在新窗口中打开指定 URL，实际上会给当前服务工作者线程添加一个新 Client。这个新 Client 对象以解决的期约形式返回。该方法可用于回应点击通知的操作，此时服务工作者线程可以检测单击事件并作为响应打开一个窗口。
- claim()：强制性设置当前服务工作者线程以控制其作用域中的所有客户端。claim()可用于不希望等待页面重新加载而让服务工作者线程开始管理页面。

27.4.4 服务工作者线程与一致性

理解服务工作者线程最终用途十分重要：让网页能够模拟原生应用程序。要像原生应用程序一样，服务工作者线程必须支持**版本控制**（versioning）。

从全局角度说，服务工作者线程的版本控制可以确保任何时候两个网页的操作都有**一致性**。该一致性可以表现为如下两种形式。

- **代码一致性**。网页不是像原生应用程序那样基于一个二进制文件创建，而是由很多 HTML、CSS、JavaScript、图片、JSON，以及页面可能加载的任何类型的文件创建。网页经常会递增更新，即版本升级，以增加或修改行为。如果网页总共加载了 100 个文件，而加载的资源同时来自第 1 版和第 2 版，那么就会导致完全无法预测，而且很可能出错。服务工作者线程为此提供了一种强制机制，确保来自同源的所有并存页面始终会使用来自相同版本的资源。
- **数据一致性**。网页并非与外界隔绝的应用程序。它们会通过各种浏览器 API 如 LocalStorage 或 IndexedDB 在本地读取并写入数据；也会向远程 API 发送请求并获取数据。这些获取和写入数据的格式在不同版本中可能也会变化。如果一个页面以第 1 版中的格式写入了数据，第二个页

面以第 2 版中的格式读取该数据就会导致无法预测的结果甚至出错。服务工作者线程的资源一致性机制可以保证网页输入/输出行为对同源的所有并存网页都相同。

为确保一致性，服务工作者线程的生命周期不遗余力地避免出现有损一致性的现象。比如下面这些可能。

- **服务工作者线程提早失败**。在安装服务工作者线程时，任何预料之外的问题都可能阻止服务工作者线程成功安装。包括服务脚本加载失败、服务脚本中存在语法或运行时错误、无法通过 `importScripts()`加载工作者线程依赖，甚至加载某个缓存资源失败。
- **服务工作者线程激进更新**。浏览器再次加载服务脚本时（无论通过 `register()`手动加载还是基于页面重载），服务脚本或通过 `importScripts()`加载的依赖中哪怕有一个字节的差异，也会启动安装新版本的服务工作者线程。
- **未激活服务工作者线程消极活动**。当页面上第一次调用 `register()`时，服务工作者线程会被安装，但不会被激活，并且在导航事件发生前不会控制页面。这应该是合理的：可以认为当前页面已加载了资源，因此服务工作者线程不应该被激活，否则就会加载不一致的资源。
- **活动的服务工作者线程粘连**。只要至少有一个客户端与关联到活动的服务工作者线程，浏览器就会在该源的所有页面中使用它。浏览器可以安装新服务工作者线程实例以替代这个活动的实例，但浏览器在与活动实例关联的客户端为 0（或强制更新服务工作者线程）之前不会切换到新工作者线程。这个服务工作者线程逐出策略能够防止两个客户端同时运行两个不同版本的服务工作者线程。

27.4.5　理解服务工作者线程的生命周期

Service Worker 规范定义了 6 种服务工作者线程可能存在的状态：**已解析**（parsed）、**安装中**（installing）、**已安装**（installed）、**激活中**（activating）、**已激活**（activated）和**已失效**（redundant）。完整的服务工作者线程生命周期会以该顺序进入相应状态，尽管有可能不会进入每个状态。安装或激活服务工作者线程时遇到错误会跳到**已失效**状态。

上述状态的每次变化都会在 `ServiceWorker` 对象上触发 `statechange` 事件，可以像下面这样为它添加一个事件处理程序：

```
navigator.serviceWorker.register('./serviceWorker.js')
.then((registration) => {
  registration.installing.onstatechange = ({ target: { state } }) => {
    console.log('state changed to', state);
  };
});
```

1. 已解析状态

调用 `navigator.serviceWorker.register()`会启动创建服务工作者线程实例的过程。刚创建的服务工作者线程实例会进入**已解析**状态。该状态没有事件，也没有与之相关的 `ServiceWorker.state` 值。

> **注意**　虽然**已解析**（`parsed`）是 Service Worker 规范正式定义的一个状态，但 `ServiceWorker.prototype.state` 永远不会返回`"parsed"`。通过该属性能够返回的最早阶段是 `installing`。

浏览器获取脚本文件，然后执行一些初始化任务，服务工作者线程的生命周期就开始了。

(1) 确保服务脚本来自相同的源。

(2) 确保在安全上下文中注册服务工作者线程。

(3) 确保服务脚本可以被浏览器 JavaScript 解释器成功解析而不会抛出任何错误。

(4) 捕获服务脚本的快照。下一次浏览器下载到服务脚本，会与这个快照对比差异，并据此决定是否应该更新服务工作者线程。

所有这些任务全部成功，则 `register()` 返回的期约会解决为一个 `ServiceWorkerRegistration` 对象。新创建的服务工作者线程实例进入到安装中状态。

2. 安装中状态

安装中状态是执行所有服务工作者线程设置任务的状态。这些任务包括在服务工作者线程控制页面前必须完成的操作。

在客户端，这个阶段可以通过检查 `ServiceWorkerRegistration.installing` 是否被设置为 `ServiceWorker` 实例：

```
navigator.serviceWorker.register('./serviceWorker.js')
.then((registration) => {
  if (registration.installing) {
    console.log('Service worker is in the installing state');
  }
});
```

关联的 `ServiceWorkerRegistration` 对象也会在服务工作者线程到达该状态时触发 `updatefound` 事件：

```
navigator.serviceWorker.register('./serviceWorker.js')
.then((registration) => {
  registration.onupdatefound = () =>
    console.log('Service worker is in the installing state');
  };
});
```

在服务工作者线程中，这个阶段可以通过给 `install` 事件添加处理程序来确定：

```
self.oninstall = (installEvent) => {
  console.log('Service worker is in the installing state');
};
```

安装中状态频繁用于填充服务工作者线程的缓存。服务工作者线程在成功缓存指定资源之前可以一直处于该状态。如果任何资源缓存失败，服务工作者线程都会安装失败并跳至**已失效**状态。

服务工作者线程可以通过 `ExtendableEvent` 停留在安装中状态。`InstallEvent` 继承自 `ExtendableEvent`，因此暴露了一个 API，允许将状态过渡延迟到期约解决。为此要调用 `ExtendableEvent.waitUntil()` 方法，该方法接收一个期约参数，会将状态过渡延迟到这个期约解决。例如，下面的例子可以延迟 5 秒再将状态过渡到**已安装**状态：

```
self.oninstall = (installEvent) => {
  installEvent.waitUntil(
    new Promise((resolve, reject) => setTimeout(resolve, 5000))
  );
};
```

更接近实际的例子是通过 `Cache.addAll()` 缓存一组资源之后再过渡：

```
const CACHE_KEY = 'v1';

self.oninstall = (installEvent) => {
  installEvent.waitUntil(
    caches.open(CACHE_KEY)
    .then((cache) => cache.addAll([
      'foo.js',
      'bar.html',
      'baz.css',
    ]))
  );
};
```

如果没有错误发生或者没有拒绝，服务工作者线程就会前进到**已安装**状态。

3. 已安装状态

已安装状态也称为**等待中**（waiting）状态，意思是服务工作者线程此时没有别的事件要做，只是准备在得到许可的时候去控制客户端。如果没有活动的服务工作者线程，则新安装的服务工作者线程会跳到这个状态，并直接进入**激活中**状态，因为没有必要再等了。

在客户端，这个阶段可以通过检查 `ServiceWorkerRegistration.waiting` 是否被设置为一个 `ServiceWorker` 实例来确定：

```
navigator.serviceWorker.register('./serviceWorker.js')
.then((registration) => {
  if (registration.waiting) {
    console.log('Service worker is in the installing/waiting state');
  }
});
```

如果已有了一个活动的服务工作者线程，则**已安装**状态是触发逻辑的好时机，这样会把这个新服务工作者线程推进到**激活中**状态。可以通过 `self.skipWaiting()`强制推进服务工作者线程的状态，也可以通过提示用户重新加载应用程序，从而使浏览器可以按部就班地推进。

4. 激活中状态

激活中状态表示服务工作者线程已经被浏览器选中即将变成可以控制页面的服务工作者线程。如果浏览器中没有活动服务工作者线程，这个新服务工作者线程会自动到达激活中状态。如果有一个活动服务工作者线程，则这个作为替代的服务工作者线程可以通过如下方式进入激活中状态。

- 原有服务工作者线程控制的客户端数量变为 0。这通常意味着所有受控的浏览器标签页都被关闭。在下一个导航事件时，新服务工作者线程会到达激活中状态。
- 已安装的服务工作者线程调用 `self.skipWaiting()`。这样可以立即生效，而不必等待一次导航事件。

在激活中状态下，不能像已激活状态中那样执行发送请求或推送事件的操作。

在客户端，这个阶段大致可以通过检查 `ServiceWorkerRegistration.active` 是否被设置为一个 `ServiceWorker` 实例来确定：

```
navigator.serviceWorker.register('./serviceWorker.js')
.then((registration) => {
  if (registration.active) {
    console.log('Service worker is in the activating/activated state');
  }
});
```

注意，ServiceWorkerRegistration.active 属性表示服务工作者线程可能在激活中状态，也可能在已激活状态。

在这个服务工作者线程内部，可以通过给 activate 事件添加处理程序来获悉：

```
self.oninstall = (activateEvent) => {
  console.log('Service worker is in the activating state');
};
```

activate 事件表示可以将老服务工作者线程清理掉了，该事件经常用于清除旧缓存数据和迁移数据库。例如，下面的代码清除了所有版本比较老的缓存：

```
const CACHE_KEY = 'v3';

self.oninstall = (activateEvent) => {
  caches.keys()
  .then((keys) => keys.filter((key) => key != CACHE_KEY))
  .then((oldKeys) => oldKeys.forEach((oldKey) => caches.delete(oldKey));
};
```

activate 事件也继承自 ExtendableEvent，因此也支持 waitUntil()方法，可以延迟过渡到**已激活**状态，或者基于期约拒绝过渡到已失效状态。

> **注意** 服务工作者线程中的 activate 事件并不代表服务工作者线程正在控制客户端。

5. 已激活状态

已激活状态表示服务工作者线程正在控制一个或多个客户端。在这个状态，服务工作者线程会捕获其作用域中的 fetch()事件、通知和推送事件。

在客户端，这个阶段大致可以通过检查 ServiceWorkerRegistration.active 是否被设置为一个 ServiceWorker 实例来确定：

```
navigator.serviceWorker.register('./serviceWorker.js')
.then((registration) => {
  if (registration.active) {
    console.log('Service worker is in the activating/activated state');
  }
});
```

注意，ServiceWorkerRegistration.active 属性表示服务工作者线程可能在**激活中**状态，也可能在**已激活**状态。

更可靠的确定服务工作者线程处于已激活状态一种方式是检查 ServiceWorkerRegistration 的 controller 属性。该属性会返回激活的 ServiceWorker 实例，即控制页面的实例：

```
navigator.serviceWorker.register('./serviceWorker.js')
.then((registration) => {
  if (registration.controller) {
    console.log('Service worker is in the activated state');
  }
});
```

在新服务工作者线程控制客户端时，该客户端中的 ServiceWorkerContainer 会触发 controllerchange 事件：

```
navigator.serviceWorker.oncontrollerchange = () => {
  console.log('A new service worker is controlling this client');
};
```

另外，也可以使用 ServiceWorkerContainer.ready 期约来检测活动服务工作者线程。该期约会在当前页面拥有活动工作者线程时立即解决：

```
navigator.serviceWorker.ready.then(() => {
  console.log('A new service worker is controlling this client');
});
```

6. 已失效状态

已失效状态表示服务工作者线程已被宣布死亡。不会再有事件发送给它，浏览器随时可能销毁它并回收它的资源。

7. 更新服务工作者线程

因为版本控制的概念根植于服务工作者线程的整个生命周期，所以服务工作者线程会随着版本变化。为此，服务工作者线程提供了稳健同时也复杂的流程，以安装替换过时的服务工作者线程。

这个更新流程的初始阶段是更新检查，也就是浏览器重新请求服务脚本。以下事件可以触发更新检查。

- 以创建当前活动服务工作者线程时不一样的URL 调用 navigator.serviceWorker.register()。
- 浏览器导航到服务工作者线程作用域中的一个页面。
- 发生了 fetch()或 push()等功能性事件，且至少 24 小时内没有发生更新检查。

新获取的服务脚本会与当前服务工作者线程的脚本比较差异。如果不相同，浏览器就会用新脚本初始化一个新的服务工作者线程。更新的服务工作者线程进入自己的生命周期，直至抵达**已安装**状态。到达**已安装**状态后，更新服务工作者线程会等待浏览器决定让它安全地获得页面的控制权（或用户强制它获得页面控制权）。

关键在于，刷新页面**不**会让更新服务工作者线程进入激活状态并取代已有的服务工作者线程。比如，有个打开的页面，其中有一个服务工作者线程正在控制它，而一个更新服务工作者线程正在**已安装**状态中等待。客户端在页面刷新期间会发生重叠，即旧页面还没有卸载，新页面已加载了。因此，现有的服务工作者线程永远不会让出控制权，毕竟至少还有一个客户端在它的控制之下。为此，取代现有服务工作者线程唯一的方式就是关闭所有受控页面。

27.4.6　控制反转与服务工作者线程持久化

虽然专用工作者线程和共享工作者线程是有状态的，但服务工作者线程是无状态的。更具体地说，服务工作者线程遵循控制反转（IoC，Inversion of Control）模式并且是事件驱动的。

这样就意味着服务工作者线程不应该依赖工作者线程的全局状态。服务工作者线程中的绝大多数代码应该在事件处理程序中定义。当然，服务工作者线程的版本作为全局常量是个显而易见的例外。服务脚本执行的次数变化很大，高度依赖浏览器状态，因此服务脚本的行为应该是幂等的。

理解服务工作者线程的生命周期与它所控制的客户端的生命周期无关非常重要。大多数浏览器将服务工作者线程实现为独立的进程，而该进程由浏览器单独控制。如果浏览器检测到某个服务工作者线程空闲了，就可以终止它并在需要时再重新启动。这意味着可以依赖服务工作者线程在激活后处理事件，但不能依赖它们的持久化全局状态。

27.4.7 通过 updateViaCache 管理服务文件缓存

正常情况下，浏览器加载的所有 JavaScript 资源会按照它们的 Cache-Control 头部纳入 HTTP 缓存管理。因为服务脚本没有优先权，所以浏览器不会在缓存文件失效前接收更新的服务脚本。

为了尽可能传播更新后的服务脚本，常见的解决方案是在响应服务脚本时设置 Cache-Control: max-age=0 头部。这样浏览器就能始终取得最新的脚本文件。

这个即时失效的方案能够满足需求，但仅仅依靠 HTTP 头部来决定是否更新意味着只能由服务器控制客户端。为了让客户端能控制自己的更新行为，可以通过 updateViaCache 属性设置客户端对待服务脚本的方式。该属性可以在注册服务工作者线程时定义，可以是如下三个字符串值。

- imports：默认值。顶级服务脚本永远不会被缓存，但通过 importScripts()在服务工作者线程内部导入的文件会按照 Cache-Control 头部设置纳入 HTTP 缓存管理。
- all：服务脚本没有任何特殊待遇。所有文件都会按照 Cache-Control 头部设置纳入 HTTP 缓存管理。
- none：顶级服务脚本和通过 importScripts()在服务工作者线程内部导入的文件永远都不会被缓存。

可以像下面这样使用 updateViaCache 属性：

```
navigator.serviceWorker.register('/serviceWorker.js', {
  updateViaCache: 'none'
});
```

浏览器仍在渐进地支持这个选项，因此强烈推荐读者同时使用 updateViaCache 和 CacheControl 头部指定客户端的缓存行为。

27.4.8 强制性服务工作者线程操作

某些情况下，有必要尽可能快地让服务工作者线程进入已激活状态，即使可能会造成资源版本控制不一致。该操作通常适合在安装事件中缓存资源，此时要强制服务工作者线程进入活动状态，然后再强制活动服务工作者线程去控制关联的客户端。

实现上述操作的基本代码如下。

```
const CACHE_KEY = 'v1';

self.oninstall = (installEvent) => {
  // 填充缓存，然后强制服务工作者线程进入已激活状态
  // 这样会触发 activate 事件
  installEvent.waitUntil(
    caches.open(CACHE_KEY)
    .then((cache) => cache.addAll([
      'foo.css',
      'bar.js',
    ]))
    .then(() => self.skipWaiting())
  );
};

// 强制服务工作者线程接管客户端
// 这会在每个客户端触发 controllerchange 事件
self.onactivate = (activateEvent) => clients.claim();
```