目的:同异步编程,解决JS单线程阻塞

功能: Service Worker 可以拦截网络请求进行本地缓存或请求转发，相当于充当服务端与浏览器、浏览器与 Web 应用程序之间的代理服务器

优点: Service Worker 带来了速度，极大的提高了用户体验

1. Service Worker 可有效加快重复访问网络应用的速度。
2. 拥有拦截请求、消息推送、静默更新、地理围栏等服务。
3. 可以在客户端通过 indexedDB API 保存持久化信息。

要求:

1. 浏览器支持Service Worker
2. 在 localhost 域或 HTTPS 域下运行：介于我们能够通过使用 Service Worker 劫持连接、编撰以及过滤响应来进行权限较高的操作
3. 注册：注册过程独立于网页，先在页面执行注册，之后在浏览器后台启动安装步骤。
4. 安装：通常需要缓存某些静态资源。当所有文件已成功缓存，则安装完毕。如果任何文件下载失败或缓存失败，则安装失败，无法激活。
5. 激活：管理就缓存的绝佳机会。激活后它将会对作用域页面实时控制，不过首次注册该服务工作线程的页面需要再次加载才会受其控制。
6. 控制时：处于两种状态之一：

①、终止以节省内存；

②、监听获取 fetch 和消息 message 事件。

1. 销毁：由浏览器决定，因此尽量不要留存全局变量。

注册 Service Worker: 查看”注册Service Worker.html”

安装 Service Worker: 以逐步打开缓存、缓存文件和确认所有需要的资产是否 缓存为例.查看”ServiceWorker.js”

激活 Service Worker: 当 Service Worker 安装成功后，便被激活，这时可实 时控制作用域中的所有网站，进行缓存文件等操作。不过首次使用 Service Worker 的页面需要再次加载才会受其控制

控制 Service Worker: 例”文件缓存.js”

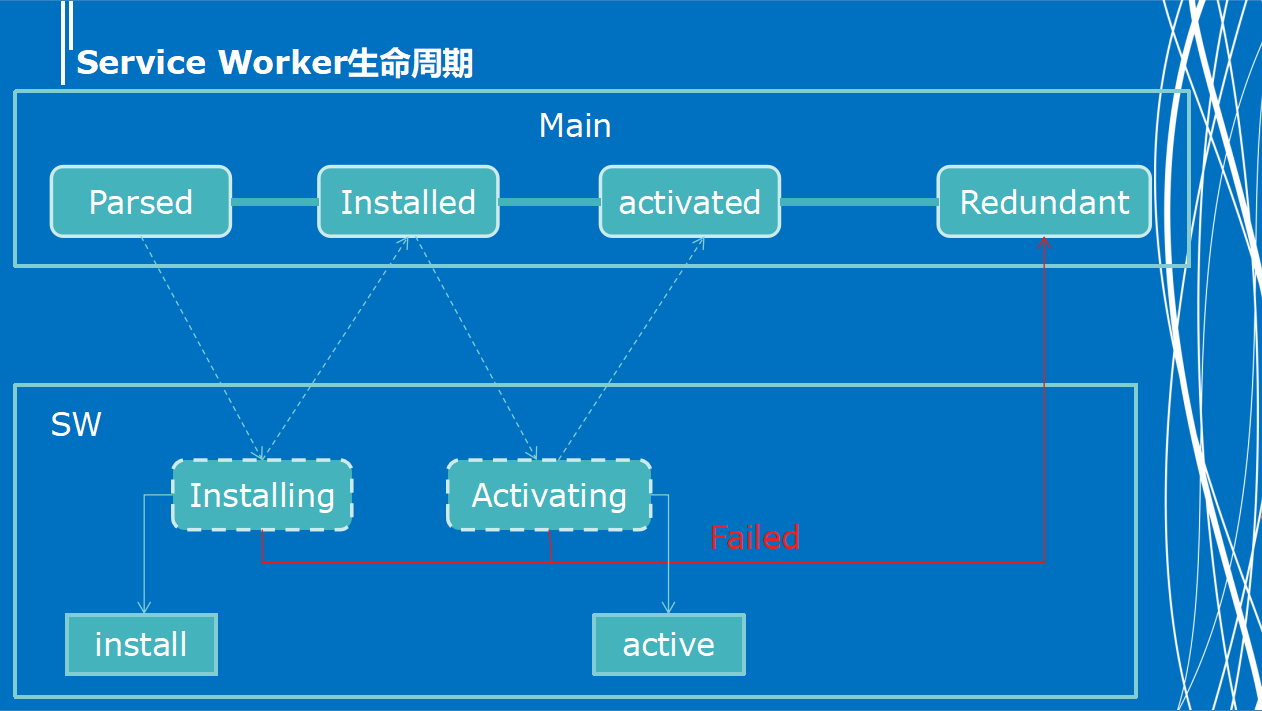
更新 Service Worker: “更新 .js”

预缓存: Service Worker 也可以在后台主动发送请求，优化用户体验

特点:

1. Https环境部署
2. 响应式设计，一次部署，可以在移动设备和 PC 设备上运行 在不同浏览器下可正常访问。
3. 浏览器离线和弱网环境可极速访问。
4. 可以把 App Icon 入口添加到桌面。
5. 点击 Icon 入口有类似 Native App 的动画效果。
6. 灵活的热更新
7. 后台消息传递
8. 网络代理，转发请求，伪造响应
9. 离线缓存
10. 消息推送

旨在（除其他之外）使得能够创建有效的离线体验，拦截网络请求并基于网络是否可用以及更新的资源是否驻留在服务器上来采取适当的动作



主线程中的状态和ServiceWorker子线程中的状态图:

依赖项：

依赖Cache API

依赖Fetch API Promise API

Https环境

错误排查：

install或active事件失败

非Https环境

sw.js安装路径问题

scope设置

当心边缘问题

之前提到过，这段代码 一定不要用在生产环境下 ，因为有很多没有处理的边界情况。

缓存依赖于每次修改内容后更新缓存名称

比如缓存方法需要你在每次改变内容后更新缓存的名字。否则，缓存不会被更新，旧的内容会一直被缓存返回。 所以，请确保每次修改你的项目后更新缓存名称。

每次修改后所有资源都需要被重新下载

另一个缺点是当一个文件被修改后，整个缓存都需要被重新下载。这意味着即使你修改了一个简单的拼写错误 也会让整个缓存重新下载。这不太高效。

浏览器的缓存可能阻碍 service worker 的缓存的更新

另外一个重要的警告。首次安装时请求的资源是直接经由 HTTPS 的，这个时候浏览器不会返回缓存的资源， 除此之外，浏览器可能返回旧的缓存资源，这导致 service worker 的缓存不会得到 更新。

在生产环境中当下 cache-first 策略

我们的应用使用了优先缓存的策略，这导致所有后续请求都会从缓存中返回而不询问网络。优先缓存的策略是 很容易实现的，但也会为未来带来诸多挑战。一旦主页和注册的 service worker 被缓存下来，将会很难 去修改 service worker 的配置（因为配置依赖于它的位置），你会发现你部署的站点很难被升级。

我该如何避免这些边缘问题

我们该如何避免这些边缘问题呢？ 使用一个库，比如 sw-precache, 它对资源何时过期提供了 精细的控制，能够确保请求直接经由网络，并且帮你处理了所有棘手的问题。

实时测试 service workers 提示

调试 service workers 是一件有调整性的事情，当涉及到缓存后，当你期望缓存更新，但实际上它并没有的时候，事情更是变得像一场恶梦。在 service worker 典型的生命周期和你的代码之间，你很快就会受挫。但幸运的是，这里有一些工具可以让你的生活更加简单。

其他的提示：

一旦 service worker 被注销（unregistered）。它会继续作用直到浏览器关闭。

如果你的应用打开了多个窗口，新的 service worker 不会工作，直到所有的窗口都进行了刷新，使用了 新的 service worker。

注销一个 service worker 不会清空缓存，所以如果缓存名没有修改，你可能继续获得到旧的数据。

如果一个 service worker 已经存在，而且另外一个新的 service worker 已经注册了，这个新的 service worker 不会接管控制权，知道该页面重新刷新后，除非你使用立刻控制的方式。

注：使用例程final或者其他service worker会出现serviceworker failed to install的错误，是因为路径原因导致缓存文件无法加载，请修改js中的文件路径或者将images 、scripts、styles三个文件夹复制到网站根目录下。

(安装是指首次运行之后,在浏览器永久留下该域名下service worker,只是在关闭网页后停止service worker进程)

浏览器一般有三类 web Worker：

- Worker：专用的 worker，只能被创建它的 JS 访问，生命周期到创建它的页面关闭时结束。

- SharedWorker：共享的 worker，可以被好几个 JS 访问，生命周期到关联的页面都关闭时结束。

ServiceWorker：一个特殊的 worker，生命周期与页面无关，关联页面未关闭时，它也可以退出，没有关联页面时，它也可以启动，嗯总之就是一个神奇的worker

对于workers来说，它运行的上下文不同于当前的window对象所在的上下文，在专用worker的情况下，DedicatedWorkerGlobalScope 对象代表了worker的上下文；在共享worker的情况下，SharedWorkerGlobalScope对象代表了共享worker的上下文。

Service Worker

前两种worker主要是为了解决js执行耗时操作时影响UI响应的问题，而之所以说service worker是一种特殊的worker，是因为它想要把一个web APP变得更像native APP，可以支持离线访问。在service worker之前，离线缓存使用AppCache来做，从Firefox44起，当使用 AppCache 来提供离线页面支持时，会提示建议开发者使用 service workers 来实现离线页面。

它是一种 JavaScript 工作线程，无法直接访问 DOM。 service worker通过响应 postMessage 接口发送的消息来与其控制的页面通信，页面可在必要时对 DOM 执行操作。

它是一种可编程网络代理，能够控制并处理页面所发送的网络请求。

它在不用时会被中止，并在下次有需要时重启，因此不能依赖于service worker的

onfetch事件 和 onmessage事件 处理程序中的全局状态。如果存在需要持续保存并在重启后加以重用的信息，service worker可以访问 IndexedDB API以及FireFox OS专用的Data Store API等数据存储机制。

service worker 广泛利用了 promise

service Workers 要求要在必须在 HTTPS 下才能运行,为了便于本地开发，localhost 也被浏览器认为是安全源。

生命周期

Service worker的生命周期完全独立于网页

如果要使用service worker，首先要在js中进行注册，注册的动作会让浏览器在后台启动service worker的安装步骤

service worker获取的第一个事件为 install。该事件在工作线程执行时立即触发，并且只能被每个service worker调用一次。 如果更改service worker的代码，则浏览器将其视为一个不同的service worker，并且它将获得自己的 install 事件。在安装的过程中，如果所有需要离线缓存的静态资源都已经成功缓存，那么service worker就安装完成进入激活步骤，如果有文件下载失败或缓存失败，service worker就无法完成安装过程。

安装之后进入激活步骤，可以对旧的缓存进行管理

激活之后，service worker开始施展身手，对它作用域内的所有页面进行控制，首次注册service worker的页面需要再次加载时才会受控制。激活之后，service worker将处于以下两种状态之一：终止或处理onfetch和onmessage事件，从页面发出网络请求或消息后将会出现后一种状态。

如果 service worker 脚本版本处于 ACTIVATED 状态，功能事件处理完之后，service worker 线程会被终止，当再次有功能事件时，service worker 线程又会被启动，启动完成后 service worker 就可以立即进入 ACTIVATED 状态。

生命周期

浏览器内核会管理三种 service worker 脚本版本：

installing\_version：处于 INSTALLING 状态的版本

waiting\_version：处于 INSTALLED 状态的版本

active\_version：处于 ACTIVATED 状态的版本

installing\_version 一般是在 service worker 线程启动后的版本，这是一个中间版本，在正确安装完成后会转入 waiting\_version。

waiting\_version 一般在注册信息已被存储的版本状态，或者在再次打开 service worker 页面时，检查到 service worker 脚本版本的状态为 INSTALLED，也会进入此版本状态。waiting\_version 的存在确保了当前 scope 下只有一个生效的 service worker。

active\_version 一般在 activate 事件处理完成后，就会处于此版本状态，同一 scope 下只有一个 active Service Worker。需要特别注意的是，当前页面已有 active worker 控制，刷新页面时，新版本 Waiting(Installed) 状态的 service worker 并不能转入 active 状态。

Service worker 可以从 waiting\_version 转入 active\_version 的条件:

当前 scope 下没有 active service worker 在运行。

页面 JS 调用 self.skipWaiting 跳过 waiting 状态。

用户关闭页面，释放了当前处于 active 状态的 service worker。

浏览器周期性检测，发现 active service worker 处于 idle 状态，就会释放当前处于 active 状态的 service worker。

其中，INSTALLED和ACTIVATED是稳定状态，可以对缓存之类的资源进行管理。