什么是PWA

PWA，即Progressive Web App, 是提升 Web App 的体验的一种新方法，能给用户原生应用的体验。

PWA并不是某一项特定的技术，而是一系列Web新技术与新标准的集合。通过灵活运用这些标准与技术，可以让我们的用户获得渐进增强的体验。所以，其实PWA本质上依然是一个Web App。

有哪些优点

Reliable：加载非常迅速，即使在各种不确定的网络情况下也不会展示断网错误页

Fast：响应用户交互非常迅速，动画平滑，操作不会有卡顿

Engaging：拥有类似原生应用的体验

PWA都有哪用到了哪些技术

Web App Manifest：是用来定义一些与web相关的东西，例如：名称，icon，启动地址等。它是一个JSON格式，它有以下几个属性配置：

background_color

为web应用程序预定义的背景颜色。此值在应用程序样式表中可以再次声明。它主要用于在样式表加载之前，当加载manifest，浏览器可以用来绘制web应用程序的背景颜色。在启动web应用程序和加载应用程序的内容之间创建了一个平滑的过渡。

```
"background_color": "red"
```

注意：background_color只是在web应用程序加载时提高用户体验，而当web应用程序的样式表可用时，不能替代作为背景颜色使用。

description

提供有关Web应用程序的一般描述。

```
"description": "The app that helps you find the best food in town!"
```

dir

指定名称、短名称和描述成员的主文本方向。与lang一起配置，可以帮助正确显示右到左文本。

```
"dir": "rtl",
"lang": "ar",
"short_name": "أنا من التطبيق!"
```

可选值:

- `ltr` (由左到右)
- `rtl` (由右到左)
- `auto` (由浏览器自动判断)。

注意：当省略时，默认为auto

display

定义开发人员对Web应用程序的首选显示模式。

```
"display": "standalone"
```

有效值:

显示模式	描述	后备显示模式
`fullscreen`	全屏显示, 所有可用的显示区域都被使用, 并且不显示状态栏chrome。	`standalone`
`standalone`	让这个应用看起来像一个独立的应用程序，包括具有不同的窗口，在应用程序启动器中拥有自己的图标等。这个模式中，用户代理将移除用于控制导航的UI元素，但是可以包括其他UI元素，例如状态栏。	`minimal-ui`
`minimal-ui`	该应用程序将看起来像一个独立的应用程序，但会有浏览器地址栏。 样式因浏览器而异。	`browser`
`browser`	该应用程序在传统的浏览器标签或新窗口中打开，具体实现取决于浏览器和平台。 这是默认的设置。	(None)

Note: 您可以使用显示模式媒体功能，根据显示模式选择性地将CSS应用到您的应用程序。 这可用于在从URL启动网站和从桌面图标启动网站之间提供一致的用户体验。

icons

指定可在各种环境中用作应用程序图标的图像对象数组。 例如，它们可以用来在其他应用程序列表中表示Web应用程序，或者将Web应用程序与OS的任务切换器和/或系统偏好集成在一起。

```
"icons": [
  {
    "src": "icon/lowres.webp",
    "sizes": "48x48",
    "type": "image/webp"
  },
  {
    "src": "icon/lowres",
    "sizes": "48x48"
  },
  {
    "src": "icon/hd_hi.ico",
    "sizes": "72x72 96x96 128x128 256x256"
  },
  {
    "src": "icon/hd_hi.svg",
    "sizes": "72x72"
```

```
  }
]
```

图像对象可能包含以下值：

字段	描述
`sizes`	包含空格分隔的图像尺寸的字符串。
`src`	图像文件的路径。 如果 `src` 是一个相对URL，则基本URL将是manifest的URL。
`type`	提示图像的媒体类型。此字段的目的是允许用户代理快速忽略不支持的媒体类型的图像。

lang

指定`name`和`short_name`成员中的值的主要语言。 该值是包含单个语言标记的字符串。

```
"lang": "en-US"
```

name

为应用程序提供一个人类可读的名称，例如在其他应用程序的列表中或作为图标的标签显示给用户。

```
"name": "Google I/O 2017"
```

orientation

定义所有Web应用程序顶级的默认方向 browsing contexts.

```
"orientation": "portrait-primary"
```

方向可以是以下值之一：

- `any`
- `natural`
- `landscape`
- `landscape-primary`
- `landscape-secondary`
- `portrait`
- `portrait-primary`
- `portrait-secondary`

prefer_related_applications

指定一个布尔值，提示用户代理向用户指示指定的相关应用程序（请参见下文）可用，并建议通过Web应用程序。 只有当相关的本地应用程序确实提供了某些Web应用程序无法做到的事情时，才应该使用它。

```
"prefer_related_applications": false
```

Note: 如果省略,默认设置为 `false`.

related_applications

指定一个“应用程序对象”数组，代表可由底层平台安装或可访问的本机应用程序 - 例如可通过Google Play Store获取的原生Android应用程序。 这样的应用程序旨在替代提供类似或等同功能的Web应用程序 - 就像Web应用程序的本机应用程序版本一样。

```
"related_applications": [
  {
    "platform": "play",
    "url": "https://play.google.com/store/apps/details?id=com.example.app1",
    "id": "com.example.app1"
  }, {
    "platform": "itunes",
    "url": "https://itunes.apple.com/app/example-app1/id123456789"
  }]
```

应用程序对象可能包含以下值：

Member	Description
`platform`	可以找到应用程序的平台。
`url`	可以找到应用程序的URL。
`id`	用于表示指定平台上的应用程序的ID。

scope

定义此Web应用程序的应用程序上下文的导航范围。 这基本上限制了manifest可以查看的网页。 如果用户在范围之外浏览应用程序，则返回到正常的网页。

如果 `scope` 是相对URL，则基本URL将是manifest的URL。

```
"scope": "/myapp/"
```

Copy to Clipboard

short_name

为应用程序提供简短易读的名称。 这是为了在没有足够空间显示Web应用程序的全名时使用。

```
"short_name": "I/O 2017"
```

Copy to Clipboard

start_url

指定用户从设备启动应用程序时加载的URL。 如果以相对URL的形式给出，则基本URL将是manifest的URL。

```
"start_url": "./?utm_source=web_app_manifest"
```

Copy to Clipboard

theme_color

定义应用程序的默认主题颜色。 这有时会影响操作系统显示应用程序的方式（例如，在Android的任务切换器上，主题颜色包围应用程序）。

```
"theme_color": "aliceblue"
```

Service Worker

Service workers 本质上充当 Web 应用程序、浏览器与网络（可用时）之间的代理服务器。这个 API 旨在创建有效的离线体验，它会拦截网络请求并根据网络是否可用采取来适当的动作、更新来自服务器的的资源。它还提供入口以推送通知和访问后台同步 API。

在service worker的生命周期里，一共分为以下阶段：

install：安装阶段

activate： 激活阶段

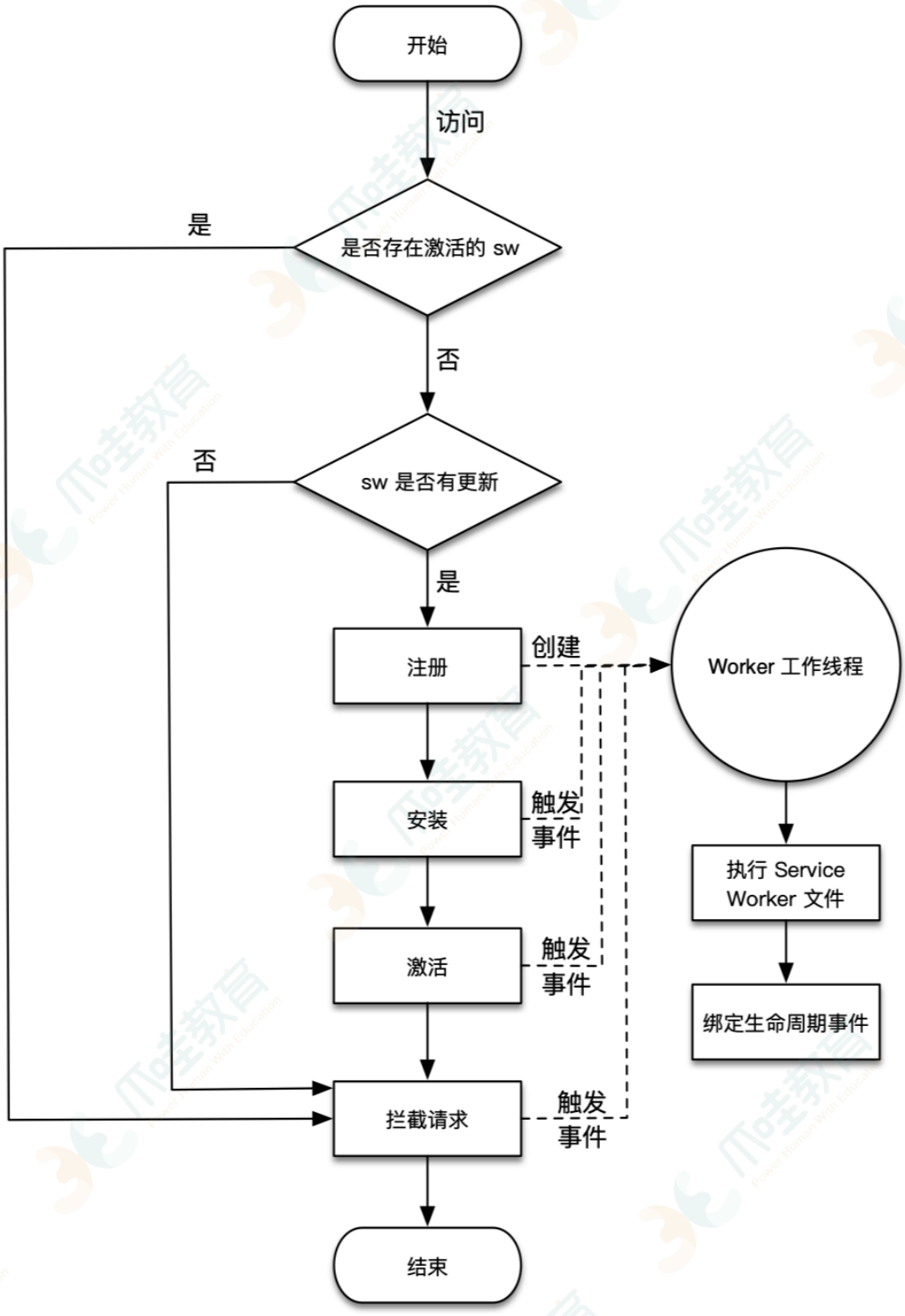
waiting：等待阶段（用于更新阶段）

> 需要注意的是，只有当文件发生更改的时候，才会重新出发注册。

```
// sw.js
console.log('service worker 注册成功')

self.addEventListener('install', () => {
  // 安装回调的逻辑处理
  console.log('service worker 安装成功')
})

self.addEventListener('activate', () => {
```

```
  // 激活回调的逻辑处理
  console.log('service worker 激活成功')
})

self.addEventListener('fetch', event => {
  console.log('service worker 抓取请求成功: ' + event.request.url)
})
```

演示第一次请求 VS 演示第二次请求，得出一个流程图：



> 思考一个问题：为什么，Service Worker 在首次安装之后，并不会启用代理呢?

原因就在于`Service Worker`的工作机制。Service Worker 的注册是一个异步的过程，在激活完成后当前页面的请求都已经发送完成，因为时机太晚，此时是拦截不到任何请求的，只能等待下次访问再进行。而第二次刷新页面，由于当前站点的 Service Worker 是处于激活状态，所以不会再次新建 worker 工作线程并执行 Service Worker。除非自己手动的更改sw的文件，或者手动的卸载掉当前的service worker。

上面说到service worker的注册是一个异步的过程，会出现什么问题呢?

```
// sw.js
console.log('service worker 注册成功')

self.addEventListener('install', () => {
  // 一段一定会报错的代码
  console.log(a.undefined)
  console.log('service worker 安装成功')
})

self.addEventListener('activate', () => {
  // 激活回调的逻辑处理
  console.log('service worker 激活成功')
})

self.addEventListener('fetch', event => {
  console.log('service worker 抓取请求成功: ' + event.request.url)
})
```

其实这就好比我们在try catch里使用prmoise一样的，由于是异步的问题，即使报错，也是在当前worker install完毕。

对于上面的问题如何改进?

使用waitUtils，waitUntil他接受一个promise，如果在install周期内，有任何一个资源报错调用了promise.reject的话，那么service worker就知道有资源获取失败了，他就会吧当前的 servrice worker中断，不会继续往下执行。这样的话就会中断他的生命周期

```
console.log('service worker 注册成功')

self.addEventListener('install', event => {
  // 引入 event.waitUntil 方法
  event.waitUntil(new Promise((resolve, reject) => {
    // 模拟 promise 返回错误结果的情况
    reject('安装出错')
    // resolve('安装成功')
  }))
})

self.addEventListener('activate', () => {
  // 激活回调的逻辑处理
  console.log('service worker 激活成功')
})

self.addEventListener('fetch', event => {
  console.log('service worker 抓取请求成功: ' + event.request.url)
})
```

在讲完如何处理异步错误之后，我们来看一个有趣的事情。

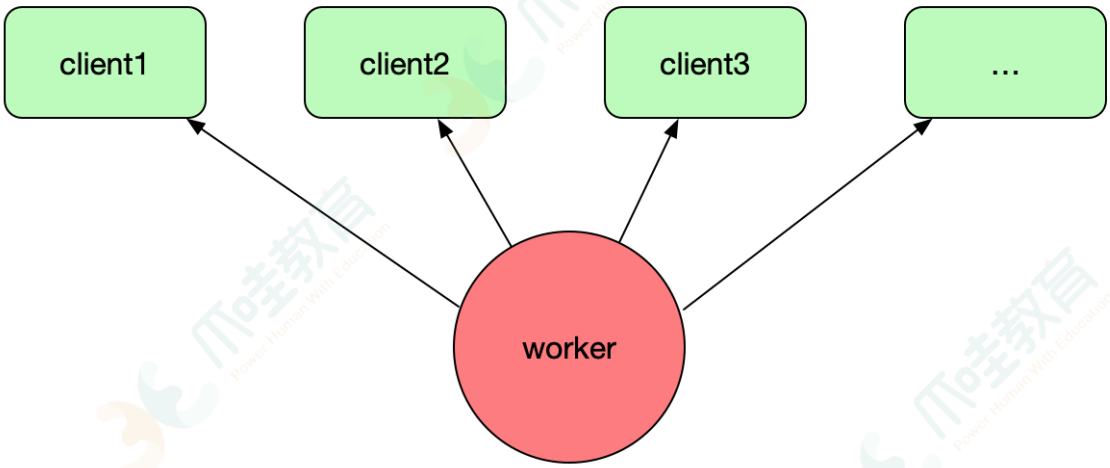
打开三个`http://localhost:8085/`页面，可以看到，我刷新了一个页面，为什么另外两个页面也会出现更新提示呢?

这是因为所有终端公用一个worker线程。这就会导致一个同步的问题。也就是worker其实已经准备好了，但是有些页面此时还在""安装"的声明周期。

如何同步所有worker:

```
self.addEventListener('activate', event => {
  // 通常在激活 Service Worker 后，通过在其中调用 self.clients.claim() 方法控制未受控制的客户端。self.clients.claim() 方法返回一个 Promise，可以直接在 waitUntil() 方法中调用，如下代码所示：
  event.waitUntil(
    self.clients.claim()
      .then(() => {
        // 返回处理缓存更新的相关事情的 Promise
      })
  )
})
```

说完注册安装完毕之后，接下来进入到缓存部分。



sw就像是一个中间件，他可以对资源进行缓存。在你下次访问的时候，会触发他的fetch事件。因此只需要在fetch里面对请求进行匹配即可。

如何缓存资源?

在开始之前呢，需要介绍一下全局的变量对象：

`Cache` 接口为缓存的 `Request` / `Response` 对象对提供存储机制。

cache.match（request， options）

cache.matchAll(request, options)

cache.addAll(request,): 抓取一个URL数组，检索并把返回的response对象添加到给定的Cache对象。

cache.put（request，response）:同时抓取一个请求及其响应，并将其添加到给定的

```
// 对于静态资源的缓存
// sw.js
// 在我们上一个例子中，我们可以知道的是:所有的页面公用一个sw,如果不设置合理的文件名，那么就可能会导致冲突。
var cacheName = 'bs-0-2-0';
var apiCacheName = 'api-0-1-1';
var cacheFiles = [
    '/',
    './index.html',
    './index.js',
    './style.css',
    './img/book.png',
    './img/loading.svg'
];

// 监听install事件，安装完成后，进行文件缓存
self.addEventListener('install', function (e) {
    console.log('Service Worker 状态： install');
    var cacheOpenPromise = caches.open(cacheName).then(function (cache) {
        return cache.addAll(cacheFiles);
    });
    e.waitUntil(cacheOpenPromise);
});
// 对于动态资源的缓存
self.addEventListener('fetch', function (e) {
    // 需要缓存的xhr请求
    var cacheRequestUrls = [
        '/test1?'
    ];
    console.log('现在正在请求：' + e.request.url);

    // 判断当前请求是否需要缓存
    var needCache = cacheRequestUrls.some(function (url) {
        return e.request.url.indexOf(url) > -1;
    });

    /**** 这里是对XHR数据缓存的相关操作 ****/
    if (needCache) {
        // 需要缓存
        // 使用fetch请求数据，并将请求结果clone一份缓存到cache
        // 此部分缓存后在browser中使用全局变量caches获取
        caches.open(apiCacheName).then(function (cache) {
            return fetch(e.request).then(function (response) {
                cache.put(e.request.url, response.clone());
```

```
                return response;
            });
        });
    }
    /* ****************************** */

    else {
        // 非api请求，直接查询cache
        // 如果有cache则直接返回，否则通过fetch请求
        e.respondWith(
            caches.match(e.request).then(function (cache) {
                return cache || fetch(e.request);
            }).catch(function (err) {
                console.log(err);
                return fetch(e.request);
            })
        );
    }
});
```

Push&Notification 推送与通知

草案中，大家下去了解。简单阐述原理

Background Sync 后台同步

草案中，大家下去了解。简单阐述原理