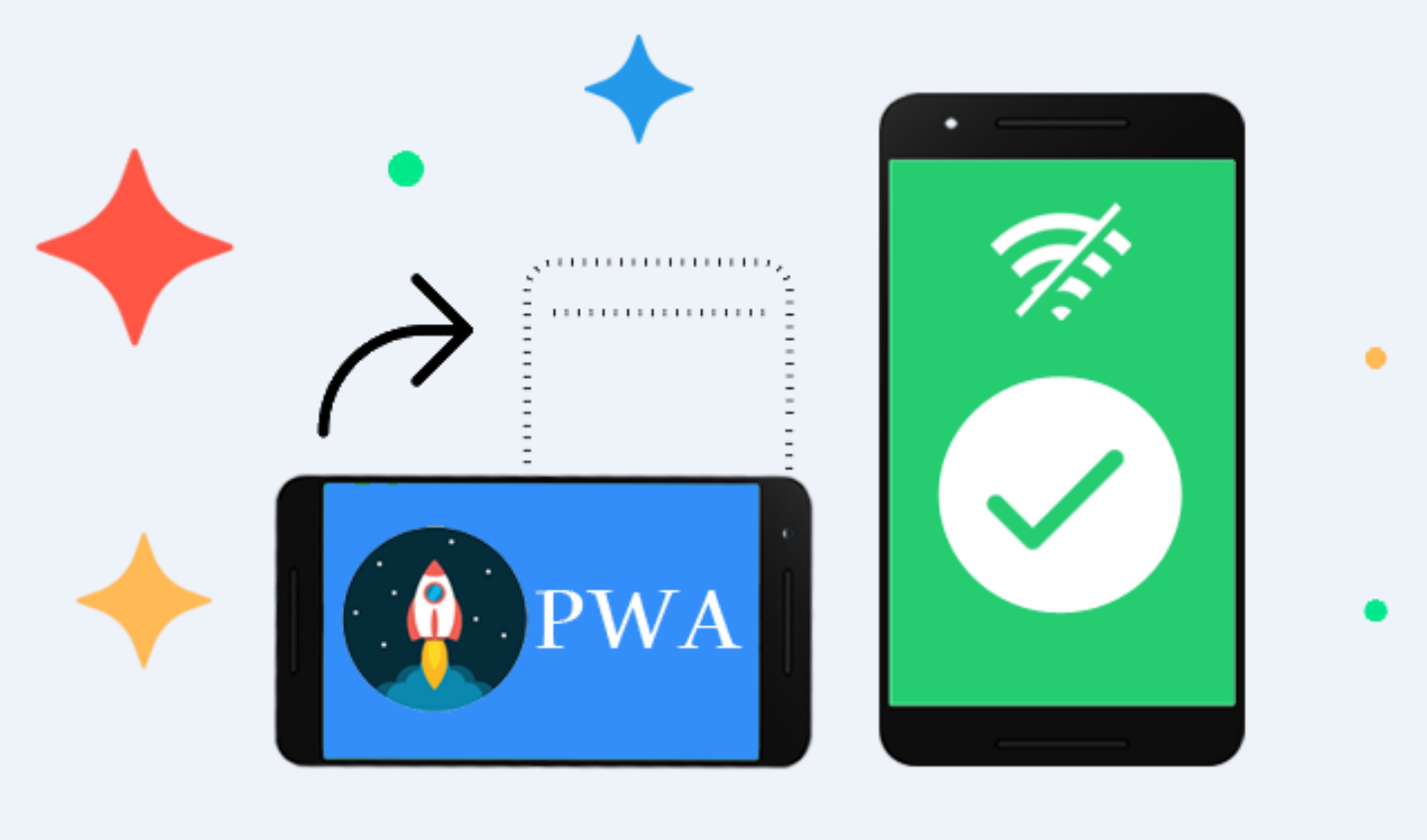


PWA技术理论+实战全解析

吕小鸣
实践永远比理论重要

9 人赞同了该文章

导读

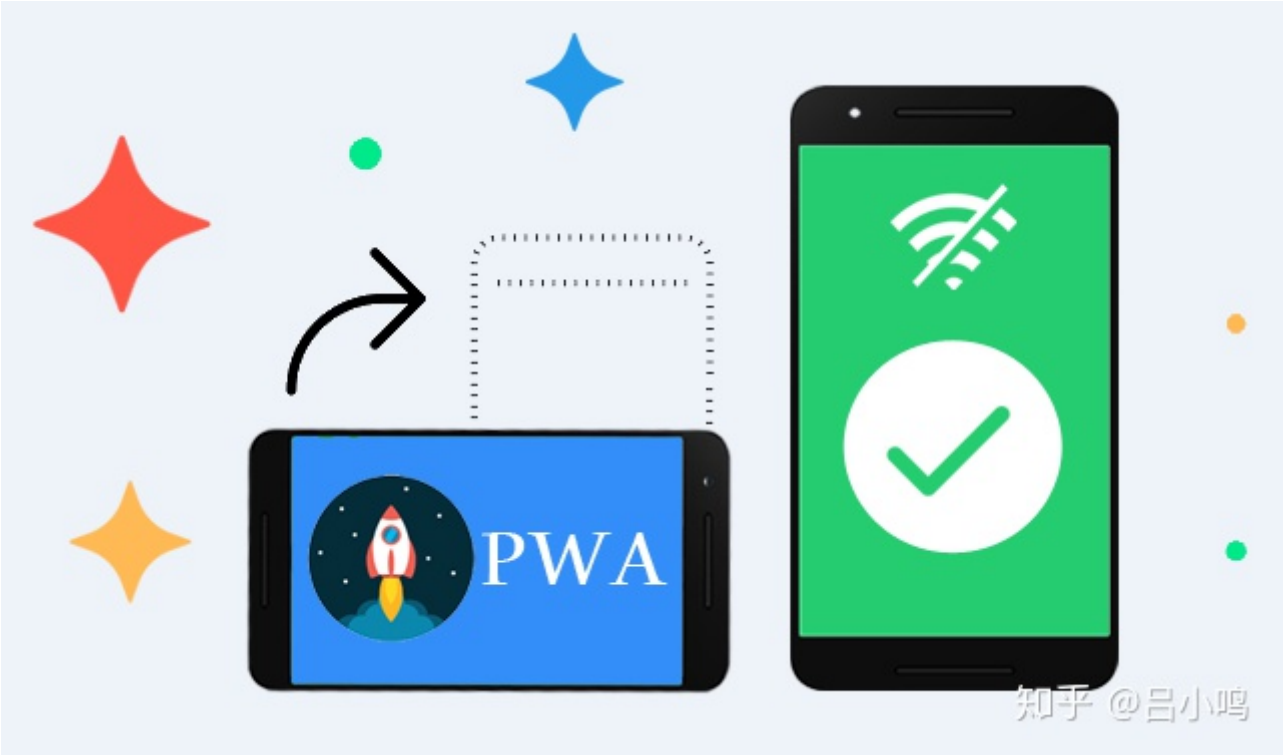
随着互联网技术的发展，web应用已经越来越流行，技术的发展越来越迅速，尤其是移动互联网的到来使得HTML5技术，Hybrid混合开发，更加火爆起来，但是web应用没能摆脱PC时代的一些根本性的问题，所需的资源依赖网络下载，用户体验始终要依赖浏览器，这让web应用和Native应用相比尤其在移动手机端的体验，总让人感觉"不正规"，而PWA技术的到来，让下一代web应用终于步入正轨！

基于此，本文主要有以下几部分内容：

- PWA基本概念讲解
- Service Worker原理讲解
- Web Push协议讲解
- 将一个SPA项目改造为PWA
  - manifest.json配置解析
  - Service Worker资源缓存
  - 添加保存到桌面功能
  - 接收消息推送

- 总结

什么是PWA?



PWA(progressing web app)，渐进式网页应用程序，是Google在2016年GoogleI/O大会上提出的下一代web应用模型，并在随后的日子里迅速发展。 一个 PWA 应用首先是一个网页, 可以通过 Web 技术编写出一个网页应用. 随后借助于 App Manifest 和 Service Worker 来实现 PWA 的安装和离线等功能。

PWA的特点

- **渐进式**：适用于选用任何浏览器的所有用户，因为它是以渐进式增强作为核心宗旨来开发的。
- **自适应**：适合任何机型：桌面设备、移动设备、平板电脑或任何未来设备。
- **连接无关性**：能够借助于服务工作线程在离线或低质量网络状况下工作。
- **离线推送**：使用推送消息通知，能够让我们的应用像 Native App 一样，提升用户体验。
- **及时更新**：在服务工作线程更新进程的作用下时刻保持最新状态。
- **安全性**：通过 HTTPS 提供，以防止窥探和确保内容不被篡改。

对于我们移动端来讲，用简单的一句话来概况一个PWA应用就是，我们开发的H5页面增加可以添加至屏幕的功能，点击主屏幕图标可以实现启动动画以及隐藏地址栏实现离线缓存功能，即使用户手机没有网络，依然可以使用一些离线功能。 这些特点和功能不正是我们目前针对移动web的优化方向吗，有了这些特性将使得 Web 应用渐进式接近原生 App，真正实现秒开优化。

Service Worker是什么

Service Worker 是一个 基于HTML5 API ，也是PWA技术栈中最重要的特性， 它在 Web Worker 的基础上加上了持久离线缓存和网络代理能力，结合Cache API面向提供了JavaScript来操作浏览器缓存的能力，这使得Service Worker和PWA密不可分。

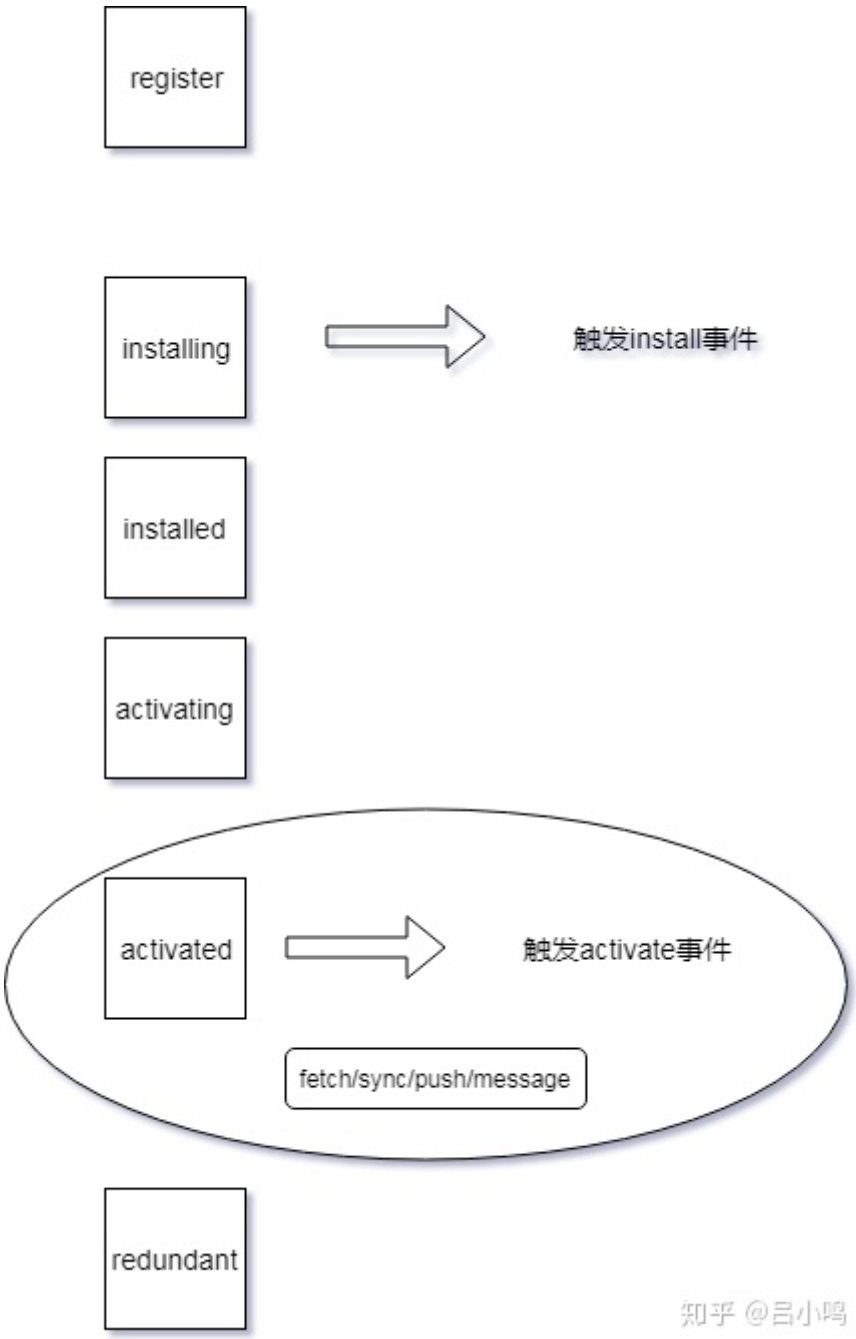
Service Worker概述：

- 一个独立的执行线程，单独的作用域范围，单独的运行环境，有自己独立的context上下文。
- 一旦被 install，就永远存在，除非被手动 unregister。即使Chrome（浏览器）关闭也会在后台运行。利用这个特性可以实现离线消息推送功能。
- 处于安全性考虑，必须在 HTTPS 环境下才能工作。当然在本地调试时，使用localhost则不受HTTPS限制。
- 提供拦截浏览器请求的接口，可以控制打开的作用域范围下所有的页面请求。需要注意的是一旦请求被Service Worker接管，意味着任何请求都由你来控制，一定要做好容错机制，保证页面的正常运行。
- 由于是独立线程，Service Worker不能直接操作页面 DOM。但可以通过事件机制来处理。例如使用postMessage。

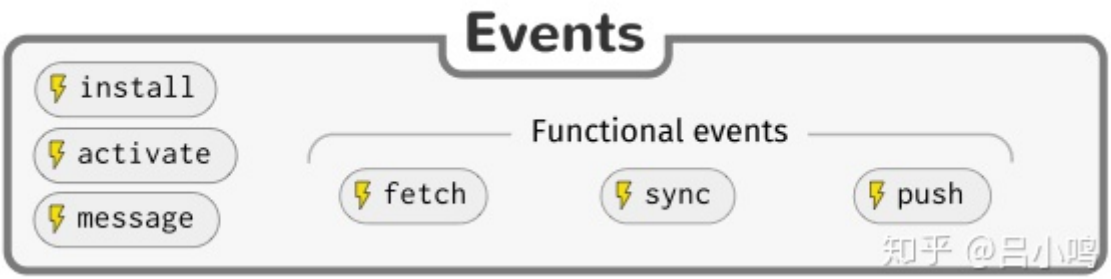
Service Worker生命周期：

- 注册（register）：这里一般指在浏览器解析到JavaScript有注册Service Worker时的逻辑，即调用navigator.serviceWorker.register()时所处理的事情。
- 安装中(installing)：这个状态发生在 Service Worker 注册之后，表示开始安装。
- 安装后(installed/waiting)：Service Worker 已经完成了安装，这时会触发install事件，在这里一般会做一些静态资源的离线缓存。如果还有旧的Service Worker正在运行，会进入waiting状态，如果你关闭当前浏览器，或者调用self.skipWaiting()，方法表示强制当前处在 waiting 状态的 Service Worker 进入 activate 状态。
- 激活(activating)：表示正在进入activate状态，调用self.clients.claim())会来强制控制未受控制的客户端，例如你的浏览器开了多个含有Service Worker的窗口，会在不切的情况下，替换旧的 Service Worker 脚本不再控制着这些页面，之后会被停止。此时会触发activate事件。
- 激活后(activated)：在这个状态表示Service Worker激活成功，在activate事件回调中，一般会清除上一个版本的静态资源缓存，或者其他更新缓存的策略。这代表Service Worker已经可以处理功能性的事件fetch (请求)、sync (后台同步)、push (推送)，message（操作dom）。
- 废弃状态 (redundant)：这个状态表示一个 Service Worker 的生命周期结束。

整个流程可以用下图解释：



Service Worker支持的事件：



Service Worker浏览器兼容性：

Service Worker作为一个新的技术，那么就必然会有浏览器兼容性问题，从图上可以看到对于大部分的Android来说支持性还是很不错的，尤其是Chrome for Android，但是对于iOS系统而言11.3之前是不支持Service Worker的，这可能也是Service Worker没能普及开来的一个原因，但是好消息是苹果宣布后续会持续更新对Service Worker的支持，那么前景还是很值得期待的。

消息推送

消息推送，顾名思义就是你在手机上收到的某个 APP 的消息推送，相较于移动端 Native 应用，web 应用是缺少这一项常用的功能。而借助 PWA 的 Push 特性，就是用户在打开浏览器时，不需要进入特定的网站，就能收到该网站推送而来的消息，例如：新评论，新动态等等，而借助于 Android 的 Chrome，我们可以实现在用户不打开任何浏览器或者应用的情况下，收到我们项目的推送，就像一个真实的手机推送。

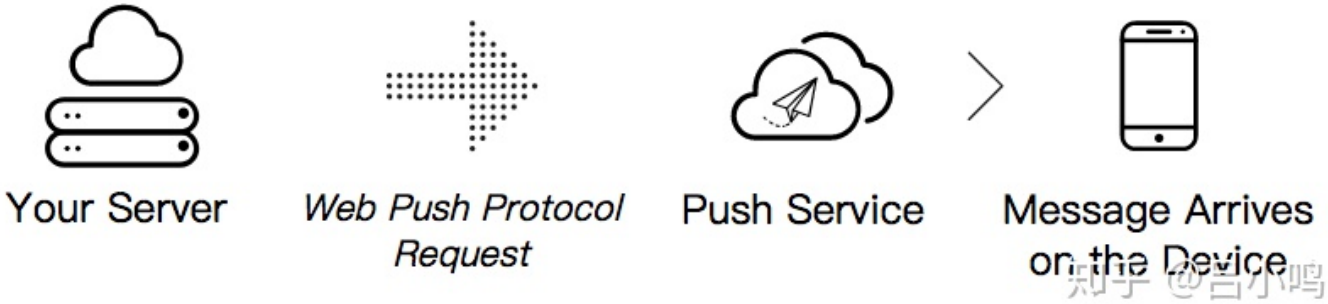
什么是Web Push

Web Push是一个基于客户端，服务端和推送服务器三者组成的一种流程规范，可以分为三个步骤： 1. 客户端完成请求订阅一个用户的逻辑。 2. 服务端调用遵从 web push 协议的接口，传送消息推送（push message）到推送服务器（该服务器由浏览器决定，开发者所能做的只有控制发送的数据）。 3. 推送服务器将该消息推送至对应的浏览器，用户收到该推送。

下图展示了一个用户订阅的过程：

所谓用户订阅，就是说我想要收到你的网站或者你的 APP 的推送通知，我就需要告诉你我是谁，我要把我的标识传给你，否则你怎么知道要给我推送。

下图展示了服务端收到用户订阅请求后如何推送：



1. 首先，在你项目的后台(Your Server)要存储一下用户订阅时传给你的标识。
2. 在后台需要给你推送的时候，找到这个标识，然后联系推送服务器(Push Service)将内容和标识传给推送服务，然后让推送服务将消息推送给用户端。（iOS和Android各自有自己的推送服务器，这个和操作系统相关）。
3. 这里就有一个约定，用户的标识，要和推送服务达成一致，例如使用Chrome浏览器，那么推送服务就是谷歌的推送服务(FCM)。

开始改造现有的SPA应用

本章节会将一个基于Vue.js2.6版本的SPA项目进行PWA改造，原有项目的开发过程就不再讲解，各位读者可以到Github来看源码，最终的体验地址：app.nihaoshijie.com.cn，请使用Safari或者Android Chrome打开体验。

添加manifest.json配置页面参数：

添加到桌面快捷方式功能本身是PWA应用的一部分，他让我们的应用看起来更像是一个Web App，我们在前端项目的 public 文件夹下新建 manifest.json 文件：

```
{
    "name": "WECIRCLE",
    "short_name": "WECIRCLE",
    "icons": [
        {
          "src": "./img/icons/android-chrome-192x192.png",
          "sizes": "192x192",
          "type": "image/png"
        },
        {
          "src": "./img/icons/android-chrome-512x512.png",
          "sizes": "512x512",
          "type": "image/png"
        }
    ],
    "start_url": "./index.html",
    "display": "standalone",
    "background_color": "#000000",
    "theme_color": "#181818"
}
```

其中：

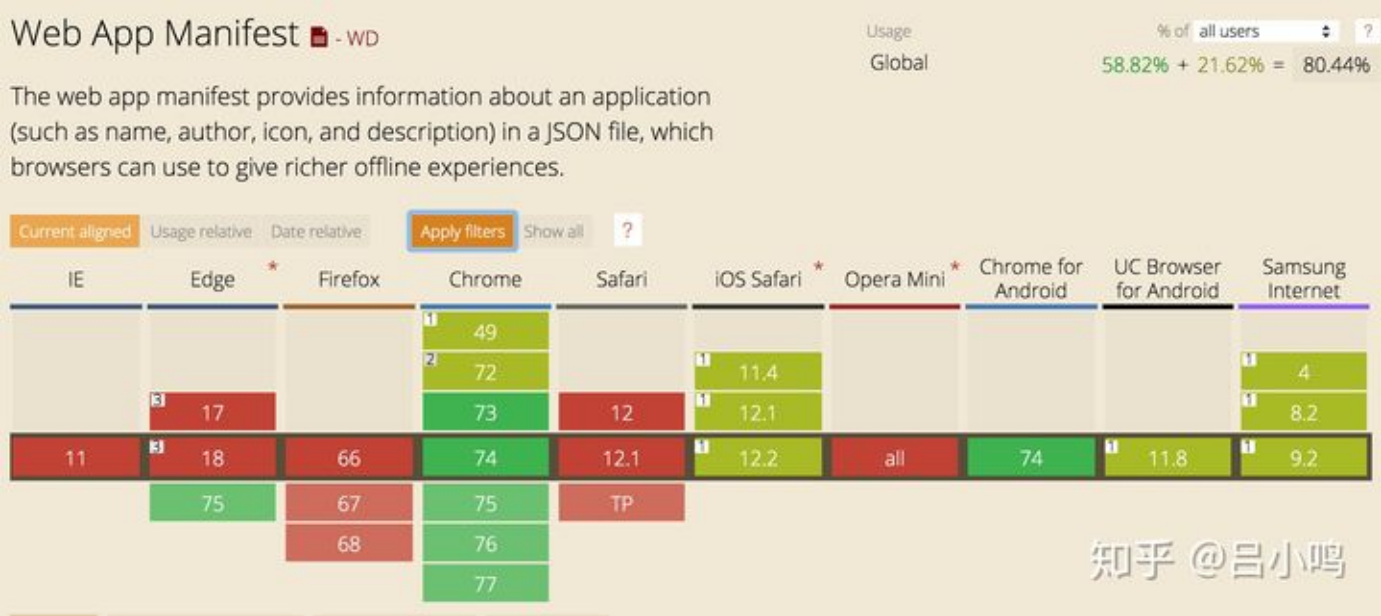
1. **name**：指定了 Web App 的名称，也就是保存到桌面图标的名称。
2. **short_name**：当 name 名称过长时，将会使用 short_name 来代替name显示，也就是 Web App 的简称。
3. **start_url**：指定了用户打开该 Web App 时加载的URL。相对URL会相对于 manifest.json 。这里我们指定了 index.html 作为 Web App 的启动页。
4. **display**：指定了应用的显示模式，它有四个值可以选择： `fullscreen` ：全屏显示，会尽可能将所有的显示区域都占满。 `standalone` ：浏览器相关UI（如导航栏、工具栏等）将会被隐藏，因此看起来更像一个Native App。 `minimal-ui` ：显示形式与standalone类似，浏览器相关UI会最小化为一个按钮，不同浏览器在实现上略有不同。 `browser` ：一般来说，会和正常使用浏览器打开样式一致。 这里需要说明一下的是当一些系统的浏览器不支持fullscreen时将会显示成 standalone 的效果，当不支持 standalone 属性时，将会显示成 minimal-ui 的效果，以此类推。
5. **icons**：指定了应用的桌面图标和启动页图像，用数组表示： `sizes` ：图标的大小。通过指定大小，系统会选取最合适的图标展示在相应位置上。 `src` ：图标的文件路径。相对路径是相对于 manifest.json 文件，也可以使用绝对路径例如xxx.png。 `type` ：图标的图片类型。 浏览器会从 icons 中选择最接近 128dp(px = dp * (dpi / 160)) 的图片作为启动画面图像。
6. **background_color**：指定了启动画面的背景颜色，采用相同的颜色可以实现从启动画面到首页的平稳过渡，也可以用来改善页面资源正在加载时的用户体验，结合icons属性，可以定义背景颜色+图片icon的启动页效果，类似与Native App的splash screen效果。

7. **theme_color**：指定了 Web App 的主题颜色。可以通过该属性来控制浏览器 UI 的颜色。比如状态栏、内容页中状态栏、地址栏的颜色。

当然，这里我们只是列举我我们项目中用到的 manifest.json 相关属性的讲解，更多的参数配置可以参考MDN，当然如果你觉得这些配置太过于繁琐，也可以用Web App Manifest Generator来实现可视化的配置。

配置iOS系统的页面参数：

理想很丰满，现实却很骨感，manifest.json 那么强大但是也逃不过浏览器兼容性问题，正如下图 manifest.json 的兼容性：



由于iOS系统对 manifest.json 是属于部分支持，所以我们需要在head里给配置而外的 meta 属性才能让iOS系统更加完善：

```
<meta name="apple-mobile-web-app-capable" content="yes">
 <meta name="apple-mobile-web-app-title" content="WECIRCLE">
 <link rel="apple-touch-icon" sizes="76x76" href="./img/icons/apple-touch-icon-76x76-1
 <link rel="apple-touch-icon" sizes="152x152" href="./img/icons/apple-touch-icon-152x1
 <link rel="apple-touch-icon" sizes="180x180" href="./img/icons/apple-touch-icon-180x1
```

- **apple-touch-icon**:指定了应用的图标，类似与manifest.json文件的icons配置，也是支持sizes属性，来供不同场景的选择。
- **apple-mobile-web-app-capable**：类似于 manifest.json 中的display的功能，通过设置为yes可以进入standalone模式，目前来说iOS系统还支持这个模式。
- **apple-mobile-web-app-title**：指定了应用的名称。
- **apple-mobile-web-app-status-bar-style**：指定了iOS移动设备的状态栏(status bar)的样

式，有Default，Black，Black-translucent可以设置。

采用iOS12.0测试下来看， `apple-touch-icon` ， `apple-mobile-web-app-status-bar-style` 是真实生效的，而 manifest.json 的 icons 则不会被iOS系统识别，下面是iOS系统safari保存到桌面操作的截图：

在Android的Chrome中：

最后，别忘了将manifest.json文件在html中进行引入：

```
<link rel="manifest" href="manifest.json">
```

注册和使用Service Worker的缓存功能：

1. 要将Service Worker进行注册：

在前端项目public文件夹下的index.html中添加如下代码：

```
if ('serviceWorker' in navigator) {
  window.addEventListener('load', function () {
    navigator.serviceWorker.register('/sw-my.js', {scope: '/'})
      .then(function (registration) {
        // 注册成功
        console.log('ServiceWorker registration successful with scope: ', registration
```

```
      })
      .catch(function (err) {
        // 注册失败:(
        console.log('ServiceWorker registration failed: ', err)
      })
  })
}
```

采用 serviceWorkerContainer.register() 来注册Service Worker，这里要做好容错判断，保证某些机型在不支持Service Worker的情况下可以正常运行，而不会报错。

另外需要注意的是只有在https下，navigator里才会有serviceWorker这个对象。

2. 在前端项目public文件夹下新建 sw-my.js ，并定义需要缓存的文件路径：

```
// 定义需要缓存的文件
var cacheFiles = [
  './lib/weui/weui.min.js',
  './lib/slider/slider.js',
  './lib/weui/weui.min.css'
]
// 定义缓存的key值
var cacheName = '20190301'
```

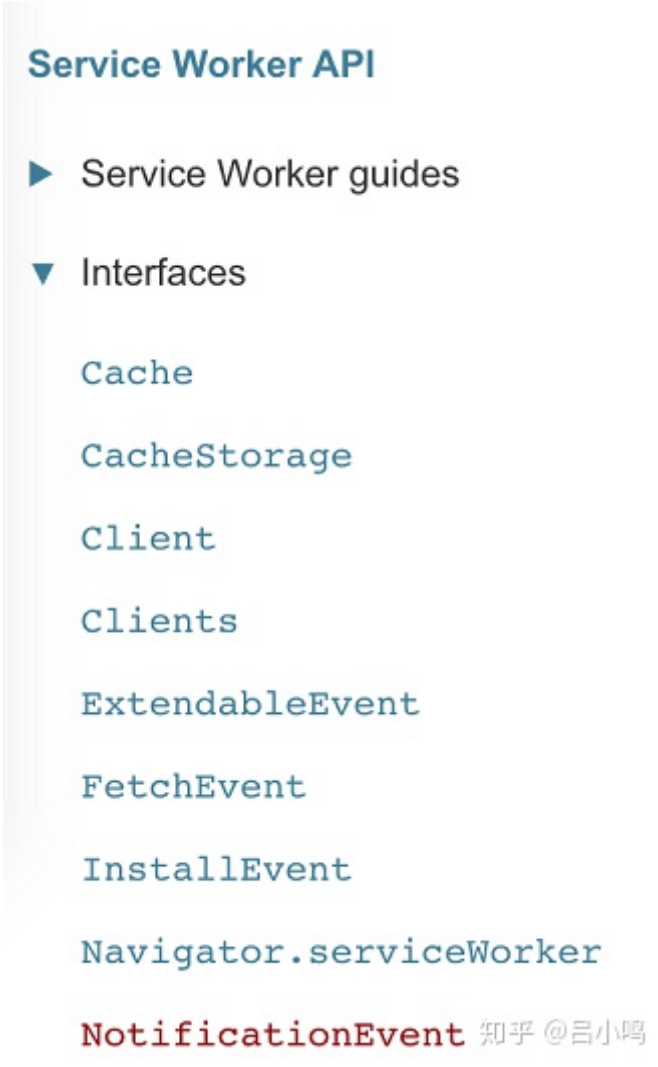
3. 监听install事件，来进行相关文件的缓存操作：

```
// 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')

  // 找到key对应的缓存并且获得可以操作的cache对象
  var cacheOpenPromise = caches.open(cacheName).then(function (cache) {
    // 将需要缓存的文件加进来
    return cache.addAll(cacheFiles)
  })
  // 将promise对象传给event
  e.waitUntil(cacheOpenPromise)
})
```

我们在 `sw-my.js` 里面采用的标准的web worker的编程方式，由于运行在另一个全局上下文中（ `self` ），这个全局上下文不同于window，所以我们采用 `self.addEventListener()` 。

Cache API是由Service Worker提供用来操作缓存的的接口，这些接口基于Promise来实现，包括了 `Cache` 和 `Cache Storage` ，Cache直接和请求打交道，为缓存的 Request / Response 对象对提供存储机制，CacheStorage 表示 Cache 对象的存储实例，我们可以直接使用全局的caches属性访问Cache API。



Cache相关API说明：

`Cache.match(request, options)` 返回一个 `Promise`对象，`resolve`的结果是跟 `Cache` 对象匹配的第一
`Cache.matchAll(request, options)` 返回一个`Promise` 对象，`resolve`的结果是跟`Cache`对象匹配的所有
`Cache.addAll(requests)`接收一个`URL`数组，检索并把返回的`response`对象添加到给定的`Cache`对象。
`Cache.delete(request, options)`搜索`key`值为`request`的`Cache` 条目。如果找到，则删除该`Cache` 条目
`Cache.keys(request, options)`返回一个`Promise`对象，`resolve`的结果是`Cache`对象`key`值组成的数组。

4. 监听fetch事件来使用缓存数据：

```
self.addEventListener('fetch', function (e) {
  console.log('现在正在请求：' + e.request.url)

  e.respondWith(
    // 判断当前请求是否需要缓存
    caches.match(e.request).then(function (cache) {
      // 有缓存就用缓存，没有就从新发请求获取
      return cache || fetch(e.request)
    }).catch(function (err) {
      console.log(err)
      // 缓存报错还直接从新发请求获取
      return fetch(e.request)
    })
  )
})
```

上一步我们将相关的资源进行了缓存，那么接下来就要使用这些缓存，这里同样要做好容错逻辑，记住一旦请求被Service Worker接管，浏览器的默认请求就不再生效了，意思就是请求的发与不发，出错与否全部由自己的代码控制，这里一定要做好兼容，当缓存失效或者发生内部错误时，及时调用fetch重新在发起请求。 正如上面提到的Service Worker的生命周期，fetch事件的触发，必须依赖Service Worker进入 `activated` 状态，于是来到第五步。

5. 监听activate事件来更新缓存数据：

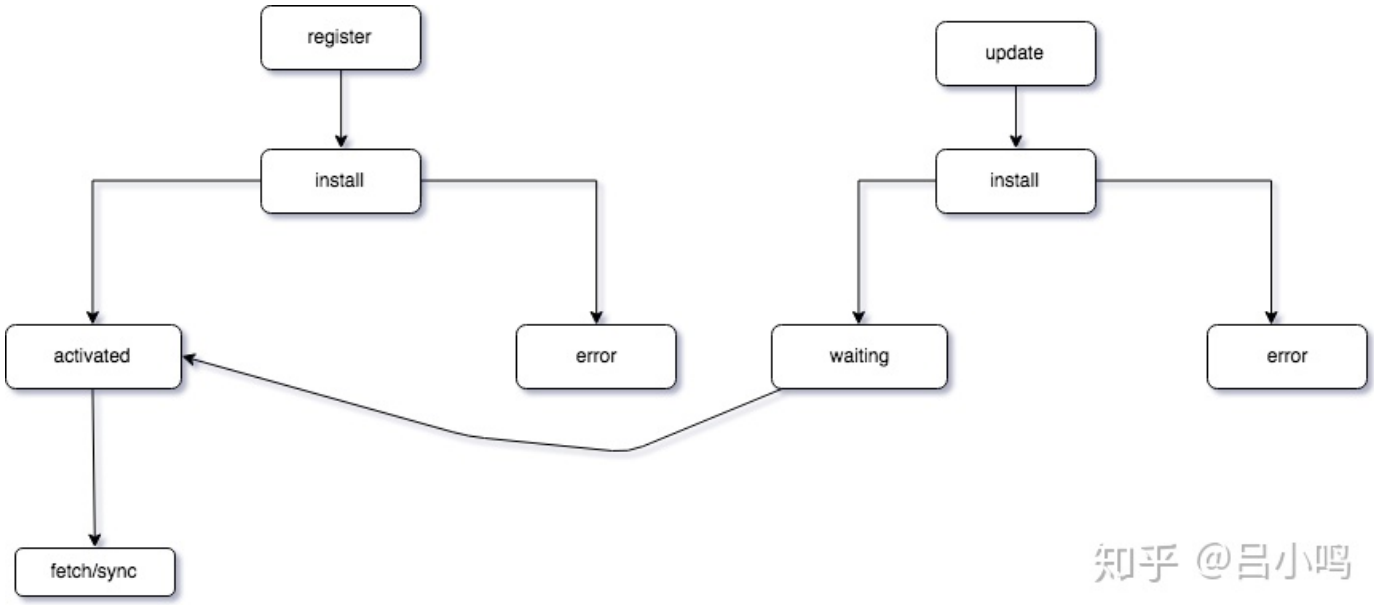
使用缓存一个必不可少的步骤就是更新缓存，如果缓存无法更新，那么将毫无意义。 我们在 `sw-my.js` 中添加如下代码：

```
// 监听activate事件，激活后通过cache的key来判断是否更新cache中的静态资源
self.addEventListener('activate', function (e) {
  console.log('Service Worker 状态： activate')
  var cachePromise = caches.keys().then(function (keys) {
    // 遍历当前scope使用的key值
    return Promise.all(keys.map(function (key) {
      // 如果新获取到的key和之前缓存的key不一致，就删除之前版本的缓存
      if (key !== cacheName) {
        return caches.delete(key)
      }
    }))
  })
  e.waitUntil(cachePromise)
```

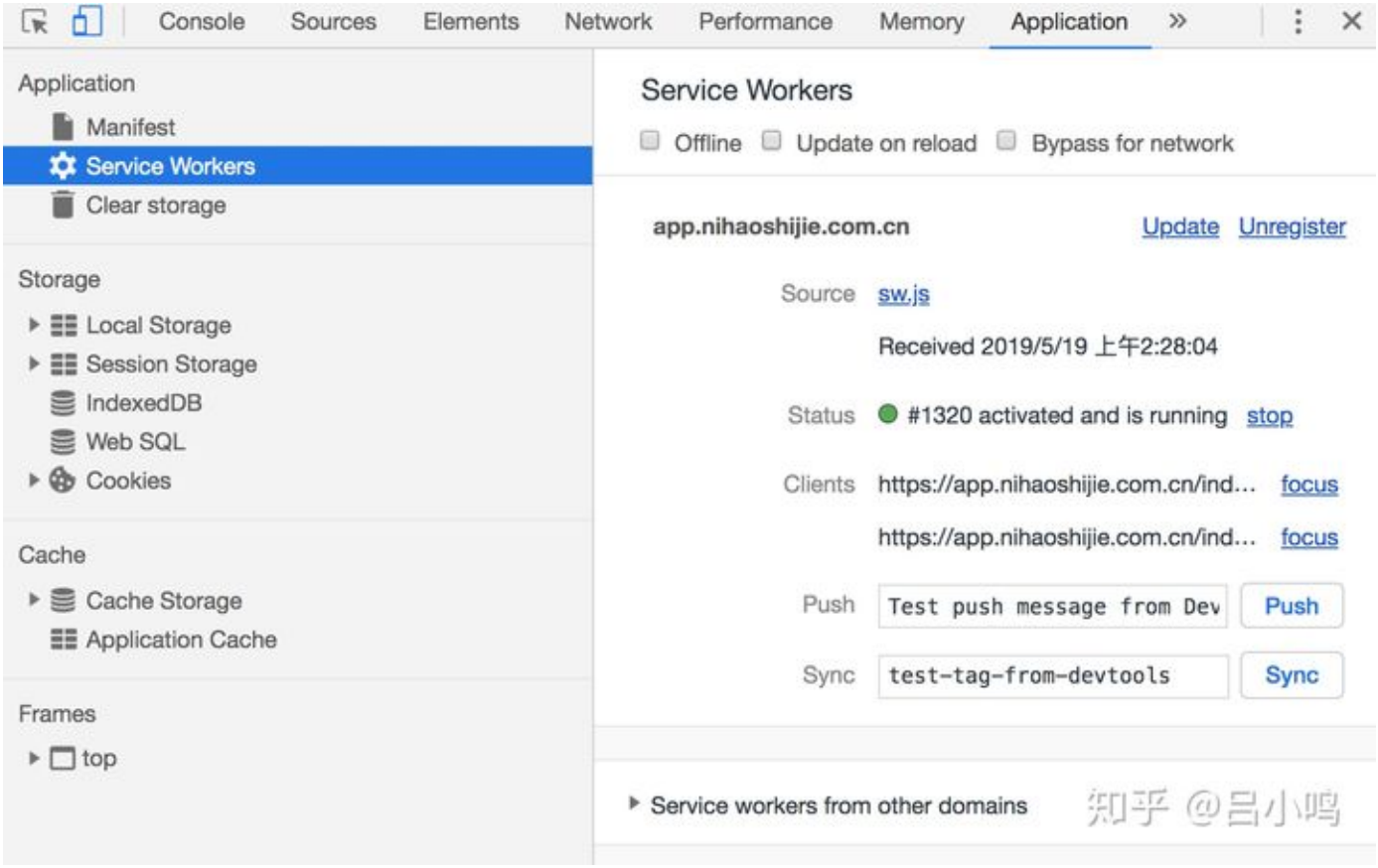
```
    // 保证第一次加载fetch触发
    return self.clients.claim()
  })
```

- 每当已安装的Service Worker页面被打开时，便会触发Service Worker脚本更新。
- 当上次脚本更新写入Service Worker数据库的时间戳与本次更新超过24小时，便会触发Service Worker脚本更新。
- 当sw-my.js文件改变时，便会触发Service Worker脚本更新。

更新流程与安装类似，只是在更新安装成功后不会立即进入 `active` 状态，更新后的Service Worker会和原始的Service Worker共同存在，并运行它的 `install` ，一旦新Service Worker安装成功，它会进入wait状态，需要等待旧版本的Service Worker进/线程终止。
`self.skipWaiting()` 可以阻止等待，让新Service Worker安装成功后立即激活。
`self.clients.claim()` 方法来让没被控制的 clients 受控，也就是设置本身为activate的Service Worker。



打开Chrome控制台，点击Application，查看Service Worker状态：



- status表示当前Service Worker的状态。
- clients表示当前几个窗口连接这个Service Worker。

这里需要说明是，如果你的浏览器开了多个窗口，那么如果在不调用 `self.skipWaiting()` 的情况下，必须将窗口关闭在打开才能使Service Worker更新成功。

采用offline-plugin插件完善Service Worker：

上面的我们写的Service Worker逻辑虽然已经完成，但是还有一些不完善的地方，比如，我们每次构建完之后，每个文件的md5都会改变，所以我们每次在写缓存文件列表时，都需要手动的的修改:

```
var cacheFiles = [
  './static/js/vendor.d70d8829.js'
  './static/js/app.d70d8869.js'
]
```

这带来的一定的复杂性，那么接下来就利用webpack的offline-plugin插件来帮助我们完善这些事情，自动生成sw-my.js。

1. 安装offline-plugin插件：

```
npm install offline-plugin --save
```

2. 在 vue.config.js 里配置：

```
configureWebpack: {
    plugins: [
      new OfflinePlugin({
        // 要求触发ServiceWorker事件回调
        ServiceWorker: {
          events: true
        },
        // 更更新策略选择全部更新
        updateStrategy: 'all',
        // 除去一些不需要缓存的文件
        excludes: ['**/.*', '**/*.map', '**/*.gz', '**/*.png', '**/*.jpg'],

        // 添加index.html的更新
        rewrites (asset) {
          if (asset.indexOf('index.html') > -1) {
            return './index.html'
          }

          return asset
        }
      })
    ]
  }
```

3. 在前端项目src目录新建 registerServiceWorker.js 里面对Service Worker进行注册：

```
import * as OfflinePluginRuntime from 'offline-plugin/runtime'
  OfflinePluginRuntime.install({

    onUpdateReady: () => {
      // 更新完成之后，调用applyUpdate即skipwaiting()方法
      OfflinePluginRuntime.applyUpdate()
    },
    onUpdated: () => {
      //弹一个确认框
      weui.confirm('发现新版本，是否更新？', ()=>{
```

```
        // 刷新一下页面
        window.location.reload()
      }, ()=>{

      }, {
          title: ''
      });
    }

  })
```

当发现Service Worker更新后，弹窗来确认是否更新，如下图：

这里说明一下：

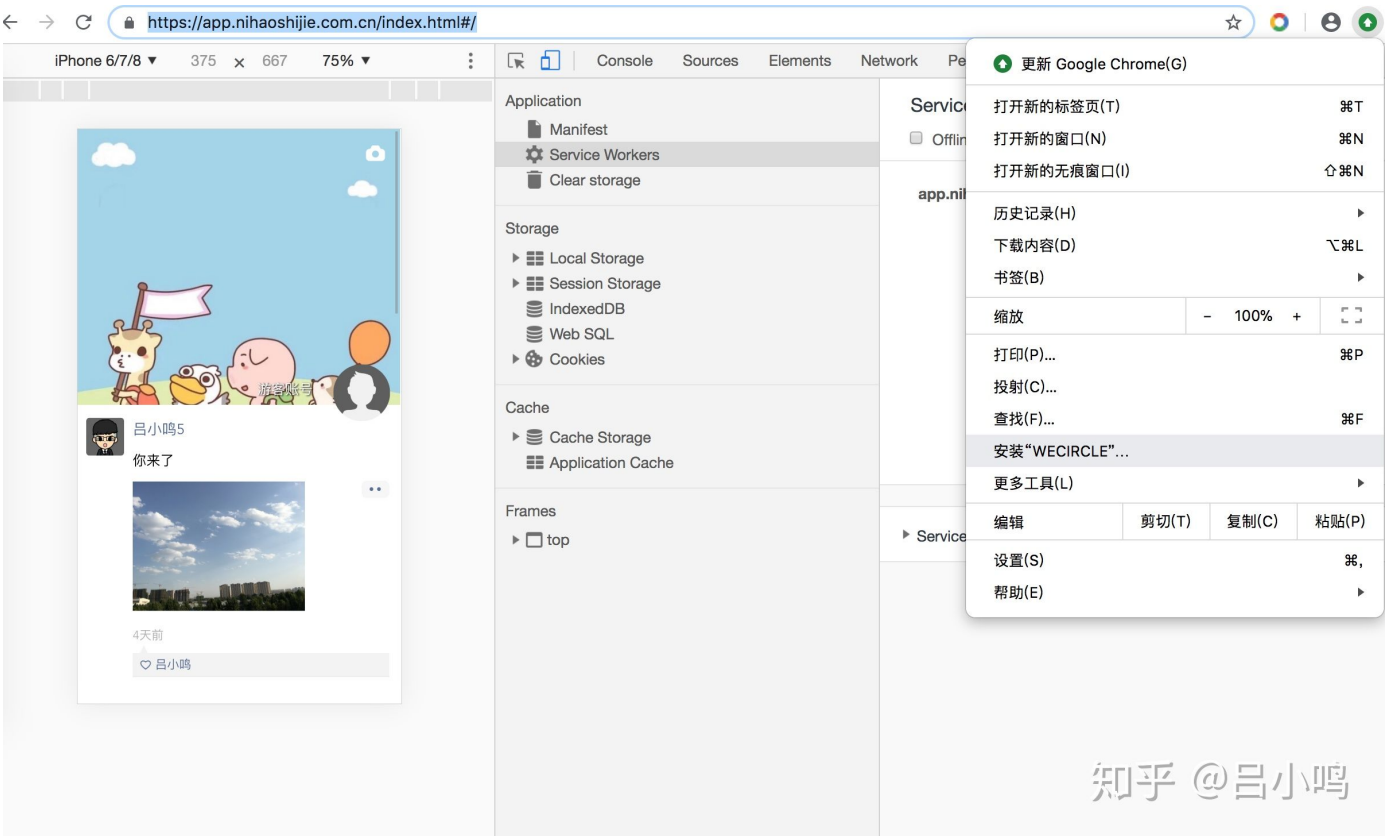
- 选择了offline-plugin插件之后呢，之前我们手写的注册Service Worker和Service Worker缓存相关逻辑都可以去掉了，因为offline-plugin会帮我们做这些事情。

- offline-plugin插件会自动扫描webpack构建出来的dist目录里的文件，对这些文件配置缓存列表，正如上面插件里面的配置。
- `excludes` ： 指定了一些不需要缓存的文件列表，例如我们不希望对图片资源进行缓存，并且支持正则表达式的方式。
- `updateStrategy` ： 指定了缓存策略选择全部更新，另外一种是增量更新 `changed` 。
- `event: true` 指定了要触发Service Worker事件的回调，这个 `main.js` 里的配置是相对应的，只有这里设置成true，那边的回调才会触发。
- 我们在 `main.js` 里的配置是为了，当Service Worker有更新时，立刻进行更新，而不让Service Worker进入wait状态，这和上面我们讲到的Service Worker更新流程相对应。

当让更多的offline-plugin相关配置，也可以去官网看文档。

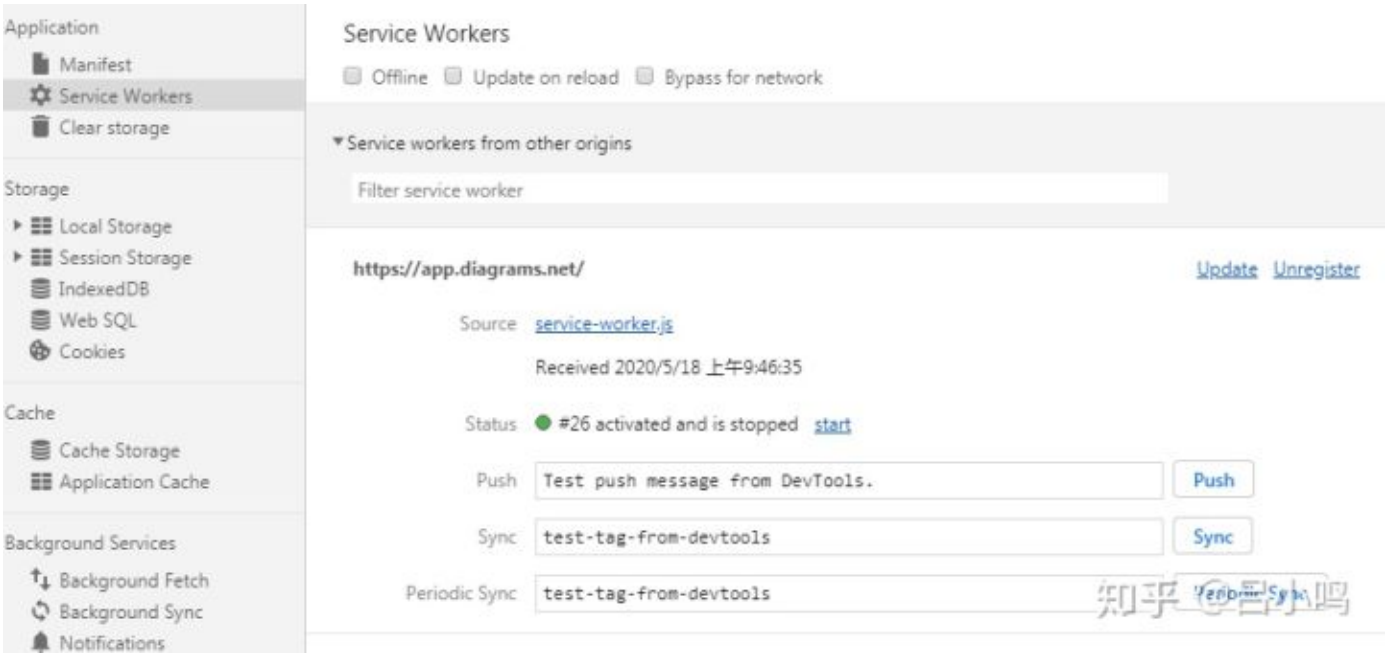
在执行npm run build命令之后，就会生成对应 的`sw.js` 文件，部署之后就可以替换我们之前手写的 `sw-my.js` 了。

除此之外，我们在PC端的Chrome也可以选择使用安装到桌面的功能，这让我们的程序应用看起来更像是一个桌面应用：



添加消息推送逻辑：

消息推送逻辑，主要分为两种方案，一种的非常简单的使用PC的Chrome的开发者工具自带的Push功能，可以通过Application->Service Worker面板->Push按钮来实现，如下图：



这种方式只能模拟简单的推送，并且依赖浏览器，大多数作为调试来使用，而真正的为APP添加消息推送，需要结合Web Push协议来实现，同时消息推送主要包括前端逻辑和后端逻辑，其中：

前端逻辑包括： - 用户授权订阅逻辑 - 收到推送后借助Notification通知逻辑

后端逻辑包括： - 存储用户授权标识 - 根据标识向推送服务器发送推送请求

1.前端订阅逻辑

获取到用户标识呢，要借助与Service Worker，基于Web Push的推送和通知相关全部要用到Service Worker。在之前创建的registerServiceWorker.js中，增加如下代码：

```
navigator.serviceWorker.ready.then((registration) => {
  //publicKey和后台的publicKey对应保持一致

  const publicKey = 'BAWz0cMW0hw4yYH-DwPrwyIVU0ee3f4oMrt6YLGPaDn3k5MNZtqjpYwUkD7nLz3AJ

  //获取订阅请求（浏览器会弹出一个确认框，用户是否同意消息推送）
  try {

      if (window.PushManager) {
          registration.pushManager.getSubscription().then(subscription => {

              // 如果用户没有订阅 并且是一个登录用户
              if (subscription && window.localStorage.getItem('cuser')) {

                  const subscription = registration.pushManager.subscribe({
```

```
            userVisibleOnly: true,//表明该推送是否需要显性地展示给用户，即推送时是否会
            applicationServerKey: urlBase64ToUint8Array(publicKey)//web-push定义
          })

          //用户同意
          .then(function(subscription) {
            console.log(subscription)
            alert(subscription)
            if (subscription && subscription.endpoint) {

              // 存入数据库
              let resp = service.post('users/addsubscription', {
                subscription: JSON.stringify(subscription)
              })
            }
          })

          //用户不同意或者生成失败
          .catch(function(err) {
              console.log("No it didn't. This happened: ", err)
            });

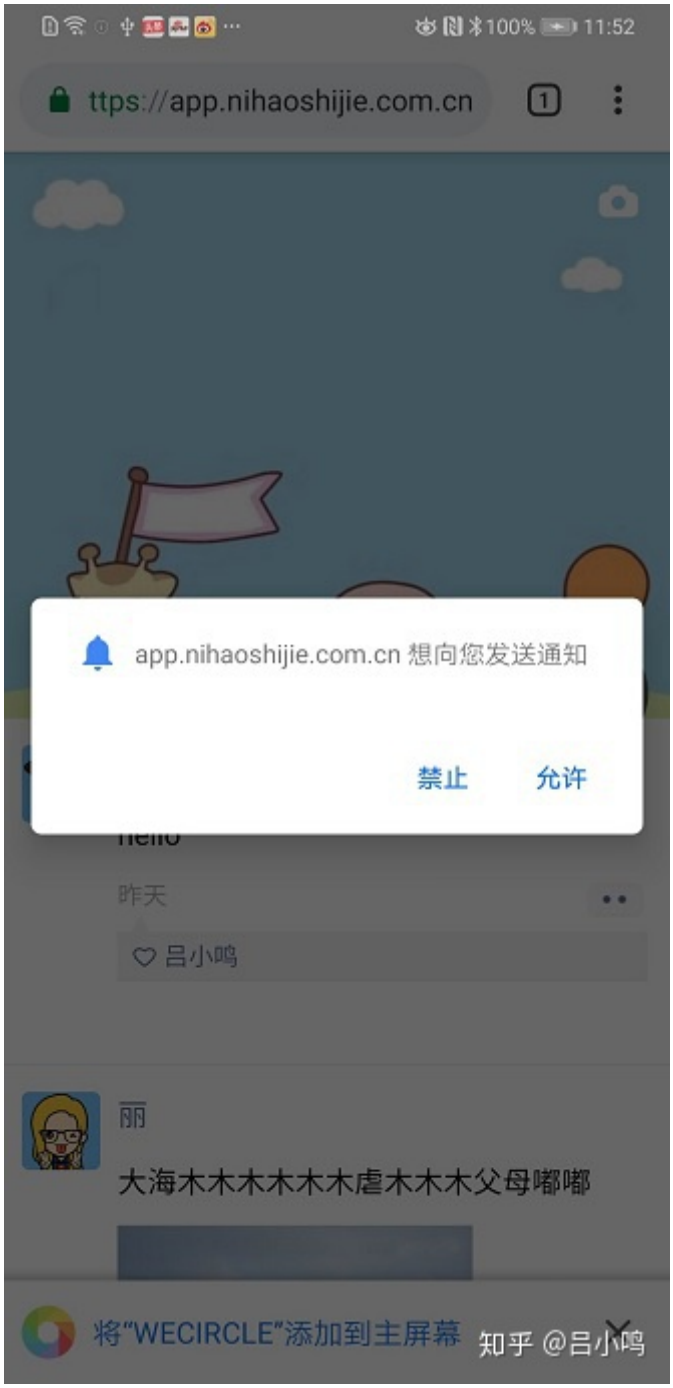
        } else {//用户已经订阅过
          console.log("You have subscribed our notification");
        }
      });
  }


}catch(e){
  console.log(e)
}

});
```

上面代码的逻辑主要要如下： 1. 通过 `registration.pushManager.getSubscription()` 先要确定用户是否已经订阅过，就是是否已经获取过标识，然后得到的 `subscription` 就是我们要的标识。（后面将 `subscription` 代替标识）。 2. 如果用户没有订阅过，通过 `registration.pushManager.subscribe()` 可以拿到 `subscription` ，在调用这个方法的时候，浏览器就会询问用户是否接受订阅，也就是会弹一个框：

如果是在手机端，前提是使用Android的Chrome，会收到这样的提示，如图：



3. 当我们点击同意，就会获取到 `subscription` ，然后通过 service 发请求到后台存储。这个 `subscription` 其实是一个对象，长这样：

```
{
    "endpoint": "https://fcm.googleapis.com/fcm/send/eekuJ6272vI:APA91bEdnUY1cpyTfRFVM
    "expirationTime": null,
    "keys": {
        "p256dh": "BLcOaaco6_dIjfIo3uiR6nDqERiCUwOuVT1mD5W45V99hvuYoqJxJZzKrKLsgE16zI_
        "auth": "vRqwuyij2AR9qkzUOwP3Pwx"
    }
}
```

对于每一个客户端来说 `subscription` 都是唯一的。

2.后端推送逻辑

后端是基于Node.js的Express框架开发的接口服务，其中推送逻辑包括在内，其他逻辑就不再文章里说明，各位可以直接浏览源码，这里只讲解Web Push的这块推送逻辑。 由于要存储前端发送的用户标识，所以需要采用数据库来持久化数据，这里使用MongoDB来存储，需要新建一张表 `Subscription` 。 在后端项目的models文件夹下新建 `Subscription.js` 代码如下：

```
var mongoose = require('mongoose');

var Schema = mongoose.Schema;

var SubscriptionSchema = new mongoose.Schema({
  subscription: { type: String,required:true },
  userid:{ type: String, unique: true },//注意这里不用ref外键
  update: { type: Date, default: Date.now },
  create: { type: Date, default: Date.now },
},{timestamps:{createdAt: 'create',updatedAt:'update'}});

module.exports = mongoose.model('Subscription', SubscriptionSchema);
```

- **subscription**:字段是一个字符串，我们会将前端传的对象 `JSON.stringify()` 一下。
- **userid**：这个字段是标识那个用户，采用 `unique:true` 表明唯一性，这里不用 ref 外键是为了该字段可空，为了后续可能会给没登录过的用户也推送一些消息。

然后在后端项目中，新增一个基于Express的路由方法：

```
/*
* 添加订阅信息
*/
router.post('/addsubscription', async (req, res, next)=> {
```

```
  var userid = req.user ? req.user._id : '';

  try {
    var result = await Subscription.create({
      subscription: req.body.subscription,
      userid:userid
    })

    res.json({
      code:0,
      data:result
    })
  }catch(e){
    // console.log(e)
    res.json({
      code:0,
      data: e.errmsg.indexOf('dup key') ? 'has scription' : e.errmsg // 说明用户已经订阅
    })
  }

});
```

上面代码通过 `Subscription.create()` 就完成了对一个 `subscription` 的存储。当存储时发现 userid 已经有过，就会抛出一个错误，就说明这个用户已经订阅过了。

然后，在后端项目的 utils 文件夹下新建 `push.js` 工具方法，来实现后台推送逻辑，首先安装 web-push，是一个基于Node.js的web-push封装，当然还有基于Java或者Php的：

```
npm install web-push --save
```

然后在 `push.js` 新增代码，首先需要生成 `vapidKeys` ，这个就是我们在前端用的那个key，要和这里保持一致，代码如下：

```
var vapidKeys = webpush.generateVAPIDKeys();
```

只需要生成一次，然后设置一下，得到之后后面一只用这个就可以，代码如下：

```
var vapidKeys = { publicKey:
   'BAWz0cMW0hw4yYH-DwPrwyIVU0ee3f4oMrt6YLGPaDn3k5MNZ1tqjpYwUkD7nLz3AJwtgo-kZhB_1pbcmz
  privateKey: 'BJ_V2wtPYaVCl7Ef2GAkVxXB2ft9cTgw-b5lM2ggc8lo' };
```

```
  webpush.setVapidDetails(
  'mailto:example@yourdomain.org',//不需要邮箱通知的话这里可以随意填
  vapidKeys.publicKey,
  vapidKeys.privateKey
);
  ```
```

完成这些设置之后，就可以和Push Service通信，来实现推送了，代码如下：

```javascript
module.exports = async function(userid,data){

  //国内使用的话，需要设置代理才行
  var option = {
    proxy: 'http://113.10.152.92:3128'   //http://www.freeproxylists.net/zh/hk.html
  }

  //从数据库中找到subscription
  var obj = await Subscription.findOne({
    userid: userid
  }).exec();
  console.log('检查是否有可推送的subscription')
  console.log(obj)
  if (obj && obj.subscription) {
    console.log('找到subscription 可以推送')
    // 调用webpush的sendNotification来发起推送通知
    webpush.sendNotification(JSON.parse(obj.subscription), JSON.stringify(data),option
      console.error(err)
    });
  }

}
```

这里解释一下，由于我们使用的推送服务器是基于谷歌的FCM，这个服务在国内是无法使用的（或者说有时可用有时不可用），所以我们需要设置一个代理，当然网上有很多免费的国外代理，可以在freeproxylists.net/zh/h...找找，如果想要稳定一点的可以掏钱买一个 VPN 服务。

3.收到推送后弹出通知

在收到Push之后如何通知提示呢，这就涉及到Notification相关的API了。

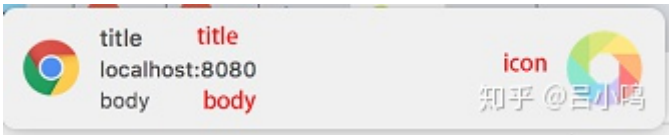
接下来，在Service Worker里注册 `push` 事件来接收 `push` 请求，在前端项目的public目录下新建一个 `sw-push.js`：
```

```
// 添加service worker对push的监听
self.addEventListener('push', function (e) {
  var data = e.data
  if (e.data) {
    data = data.json()
      e.waitUntil(
        self.registration.showNotification(data.title, {
          body: data.body || '',
          icon: data.img || "https://app.nihaoshijie.com.cn/img/icons/apple-touch-icon
          actions: [{
              action: 'go-in',
              title: '进入程序'
          }]
        })
      );
    ;
  } else {
    console.log('push没有任何数据')
  }
})
```

当浏览器收到推送通知时，就会进入这个事件里，我们通过 `self.registration.showNotification()`

- title：消息的标题，属于必传的值。
- body：消息的实体，可以不传。
- icon：配置消息的图片，会出现在消息里面。
- actions：配置消息的操作项，在结合 `notificationclick` 事件可以实现消息的点击交互。

就可以弹出一个通知框，前提是你在之前的通知允许中点击了确定：



在手机端是这个样子：

如果想要在手机端或者是PC端收到提示，前端必须满足下面条件： 1. PC端的Chrome要可以翻墙，也就是能够使用谷歌相关的服务。 2. 手机端的Chrome要内置了Chrome服务(GMS)，据笔者实验国内的华为，vivo，小米系列基本是没有内置Chrome服务的，而Nexus系列的手机则可以正常使用。能够连接Google Play则代表可以。

你要问我为什么这么多条件？原因是web-push是基于谷歌的FCM(云消息机制)实现的推送，而FCM包含在GMS里面。知道谷歌禁止国外的华为手机使用谷歌服务有多大影响了吧，连消息都收不到啊。

在 Notification 添加点击事件实现完整消息流程，在 `sw-push.js` 增加如下代码：

```
self.addEventListener('notificationclick', function (e) {
    var action = e.action;
    e.waitUntil(
        // 获取所有clients
        self.clients.matchAll().then(function (clientList) {
```

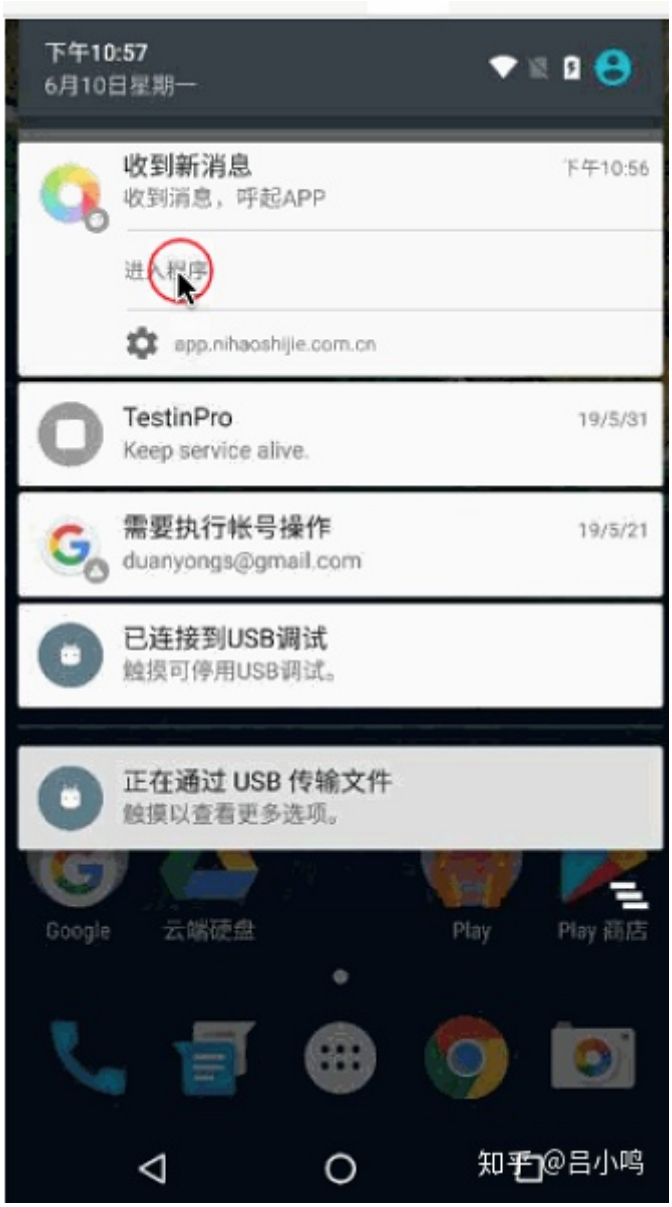
```
            if (clientList.length > 0) {
              return clientList[0].focus();
            }

            if (action === 'go-in') {
              return self.clients.openWindow('https://app.nihaoshijie.com.cn/index.html#
            }

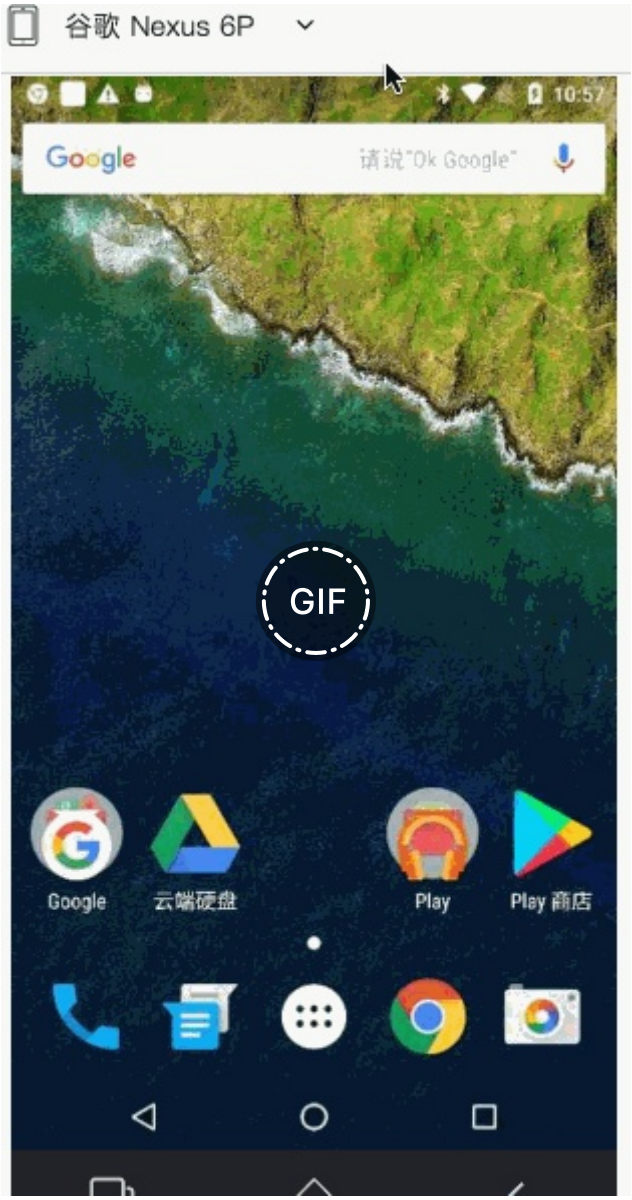
          })
      );
      e.notification.close();
  });
```

在 `self.registration.showNotification()` 中，我们传了一个action，这里就对应了消息弹出时，有选项可以选择： 在PC端：

在手机端：



根据上节我们讲的离线APP，收到消息通知的条件不限于你必须打开着APP，经过验证，即使APP已经关闭，同样可以收到推送消息，调用 `self.clients.openWindow()` 可以将APP呼起来，可以看下图的流程：



最后，我们的 `sw-push.js` 需要配置在 offline-plugin 插件里面进行合并，最终对于Service Worker 只有一个 `sw.js` ，在 `vue.config.js` 里修改代码，如下：

```
...
  ServiceWorker: {
    events: true,
    // push事件逻辑写在另外一个文件里面
    entry: './public/sw-push.js'
  },
...
```

总结

本文主要讨论了PWA的相关知识，以及如何将SPA项目改成成一个PWA应用。 相关知识点： 1. PWA应用的概念以及PWA应用的特性。 2. Service Worker的兼容性以及生命周期和事件等基本

概念。 3. Web Push的概念和基本流程。 4. manifest.json文件的各个配置项作用。 5. 拦截 `fetch` 事件，缓存前端静态资源文件的原理。 6. 结合offline-plugin插件，将项目改造成PWA应用。 7. 在Node.js里使用Web Push，并推送给前端。 8. Web Notification的API及相关的配置来提示消息。

本文改造的SPA项目完整源代码地址：Github

码字不易，如果本文对你有帮助，动手点个赞！

编辑于 2020-05-29

渐进式网络应用程序（PWA） HTML5 应用 网页应用