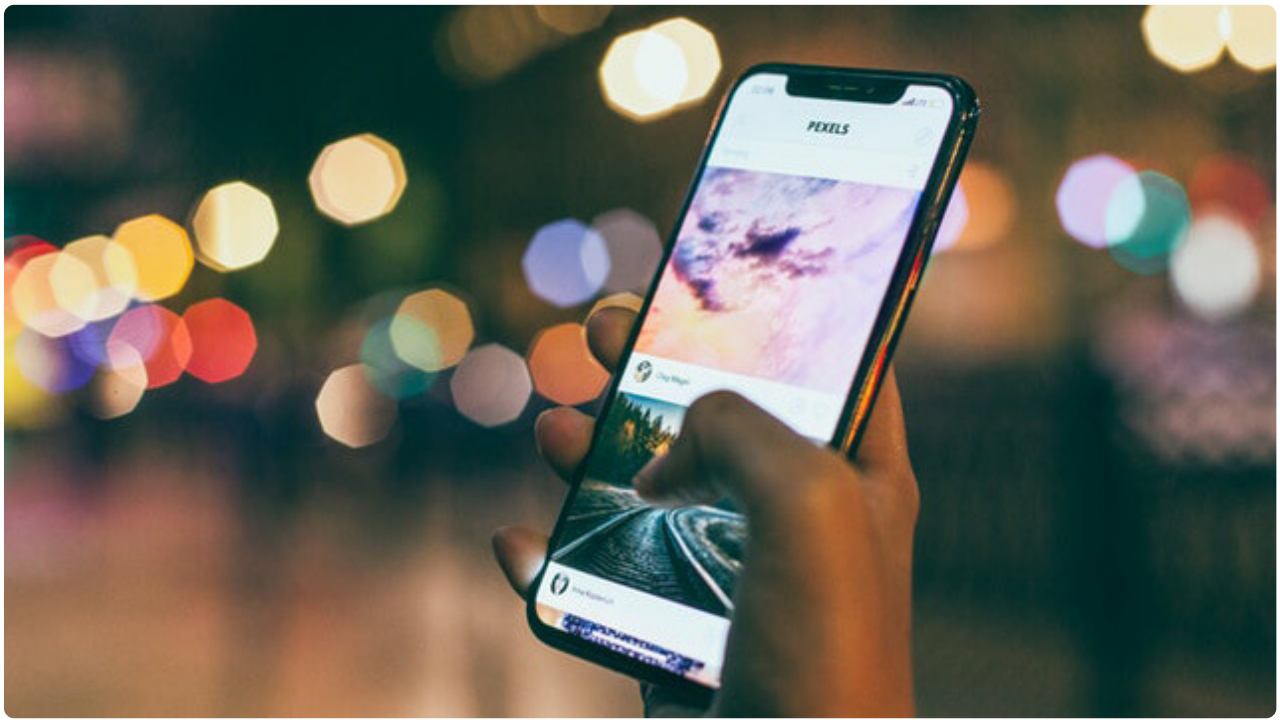
31 使用 PWA 改造真正的 webApp

更新时间：2019-09-30 09:48:17



> 只有在那崎岖的小路上不畏艰险奋勇攀登的人,才有希望达到光辉的顶点。
>
> ——马克思

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！

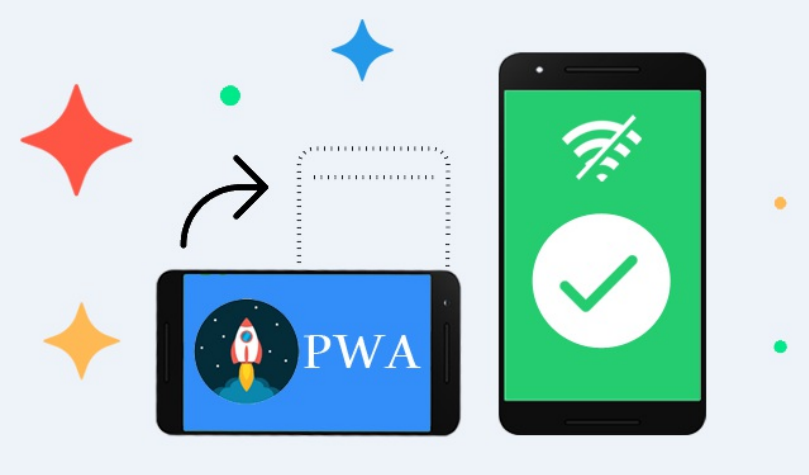
本章节完整源代码地址，大家可以事先浏览一下：

Github-registerServiceWorker.js

Github-vue.config.js

Github-sw-my.js

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

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：

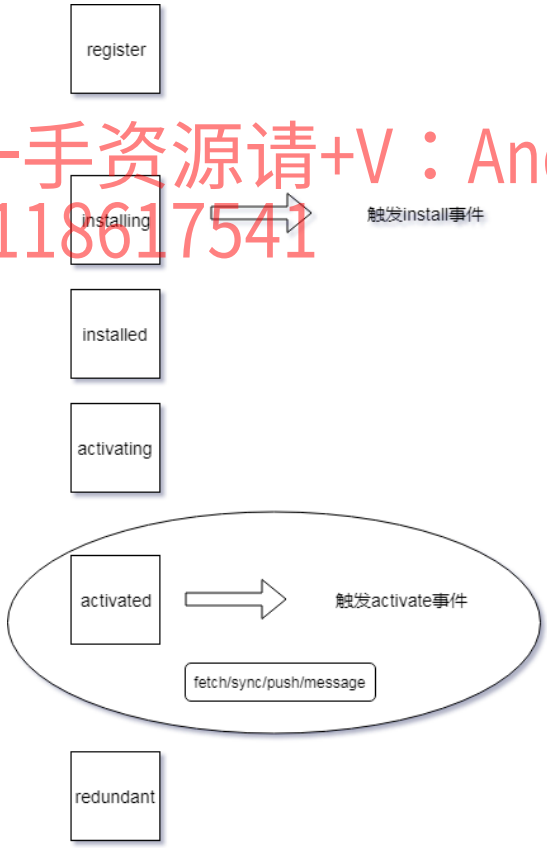
- 一个独立的执行线程，单独的作用域范围，单独的运行环境，有自己独立的context上下文。
- 一旦被 install，就永远存在，除非被手动 unregister。即使Chrome（浏览器）关闭也会在后台运行。利用这个特性可以实现离线消息推送功能。
- 处于安全性考虑，必须在 HTTPS 环境下才能工作。当然在本地调试时，使用localhost则不受HTTPS限制。
- 提供拦截浏览器请求的接口，可以控制打开的作用域范围下所有的页面请求。需要注意的是一旦请求被Service Worker接管，意味着任何请求都由你来控制，一定要做好容错机制，保证页面的正常运行。
- 由于是独立线程，Service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

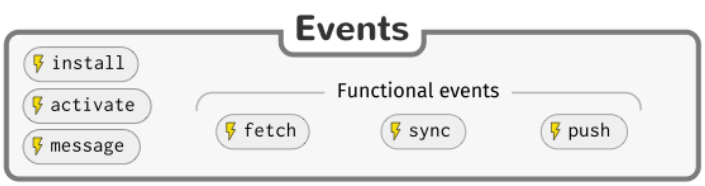
Service Worker生命周期：

- 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。

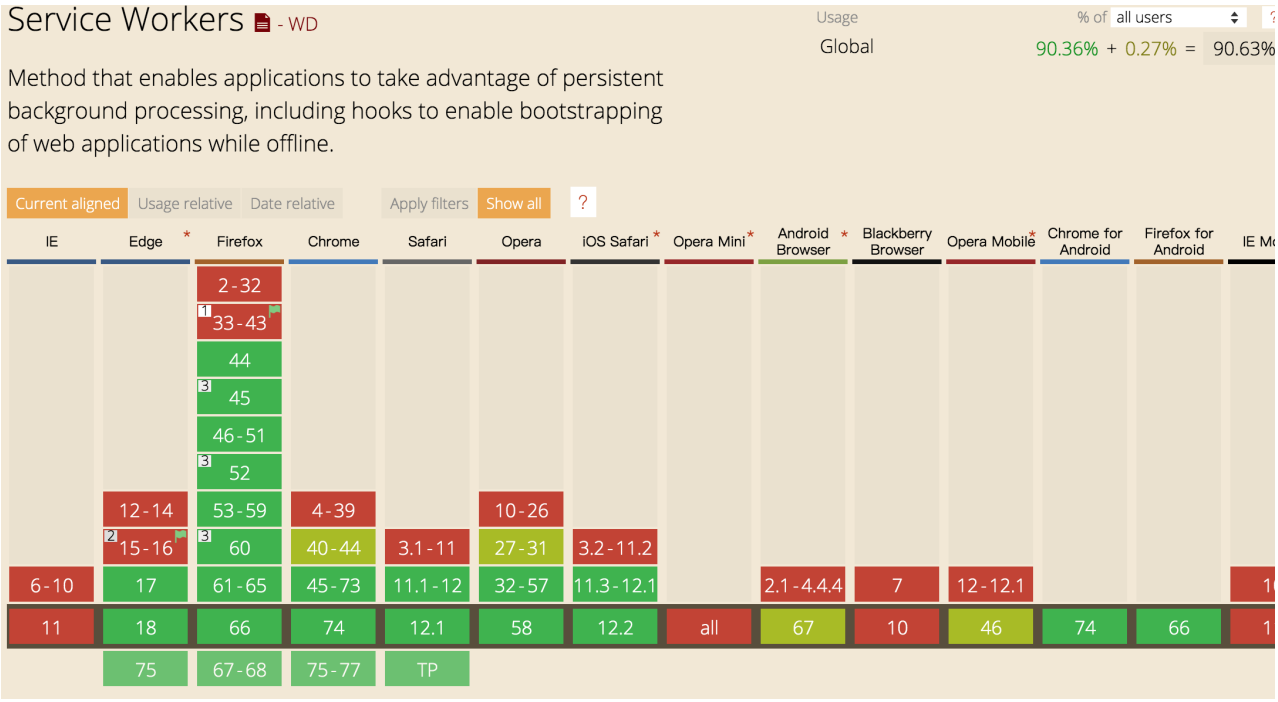
整个流程可以用下图解释：



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。

开始改造我们的**wecircle**应用：

先看下我们的改造效果：

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。

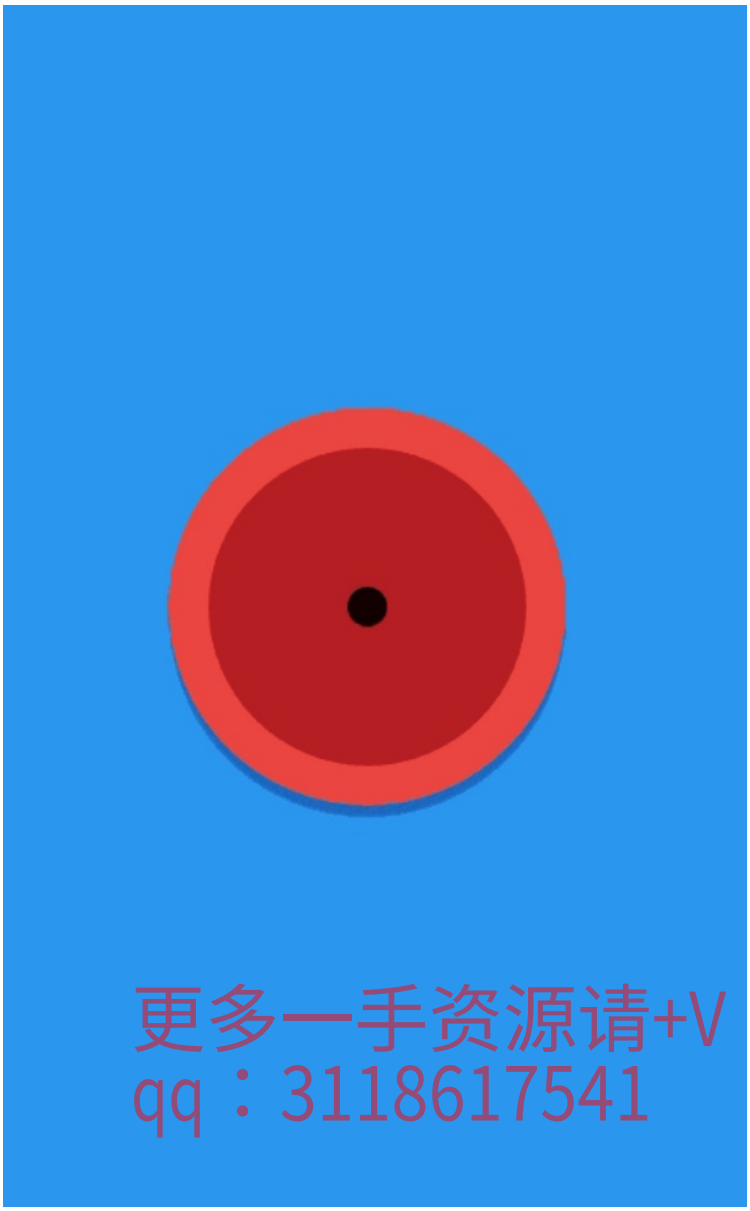
   这里需要说明一下的是当一些系统的浏览器不支持fullscreen时将会显示成 standalone 的效果，当不支持 standalone 属性时，将会显示成 minimal-ui 的效果，以此类推。
5. **icons**：指定了应用的桌面图标和启动页图像，用数组表示：

   `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http://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：

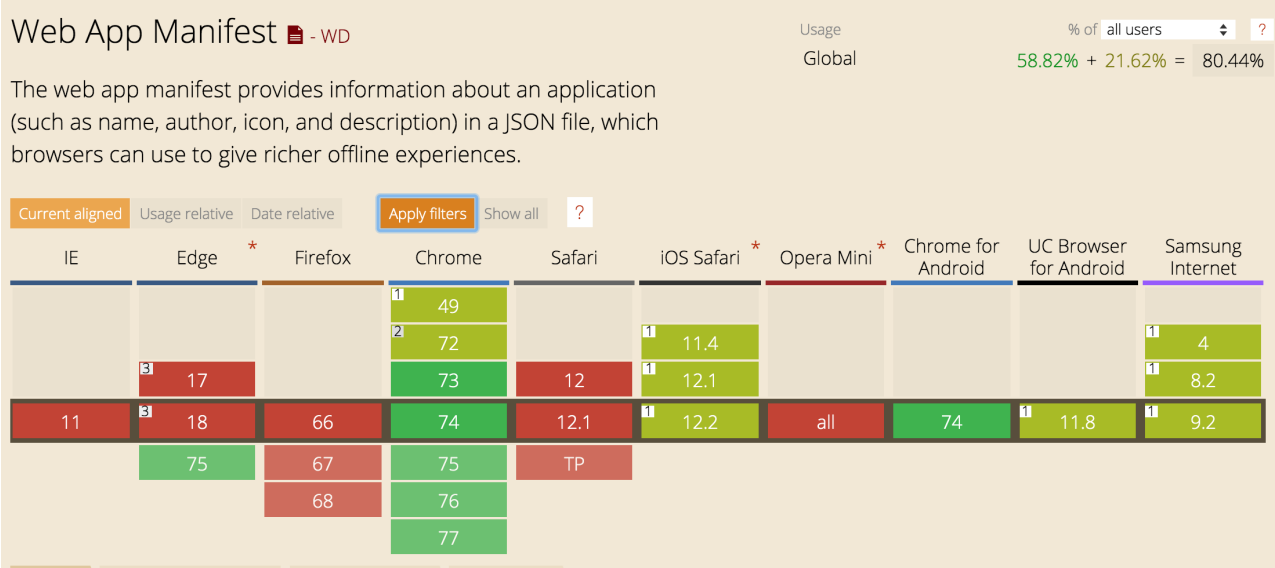


7. **theme_color**：指定了 Web App 的主题颜色。可以通过该属性来控制浏览器 UI 的颜色。比如状态栏、内容页中状态栏、地址栏的颜色。

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：

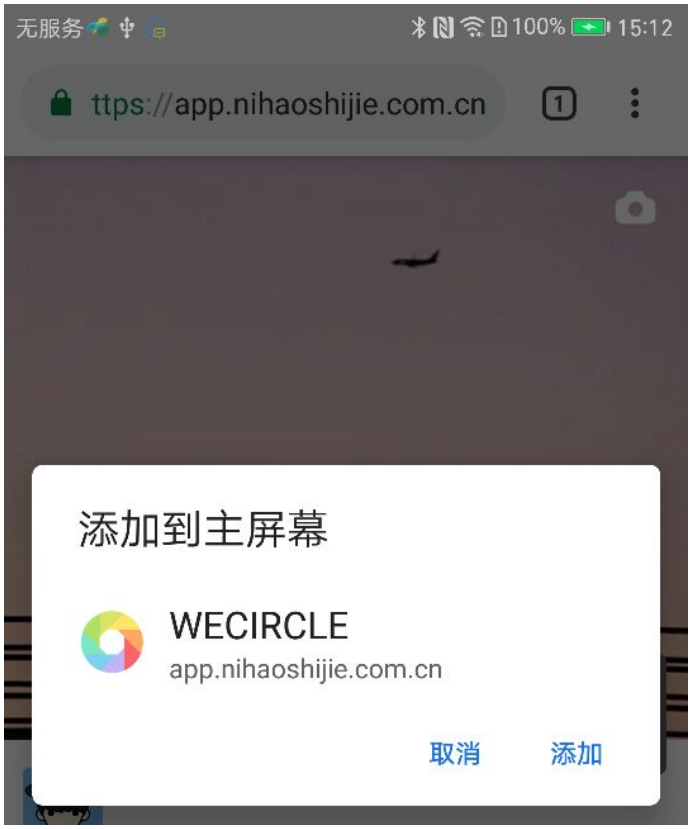
```
<meta name="apple-mobile-web-app-capable" content="yes">
<meta name="apple-mobile-web-app-title" content="WECIRCLE">
<link rel="apple-touch-icon" sizes="76x76" href="./img/icons/apple-touch-icon-76x76-1.png" />
<link rel="apple-touch-icon" sizes="152x152" href="./img/icons/apple-touch-icon-152x152.png-1"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


- **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：

此网站。

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：

第一步，我们需要将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.scope)
      })
      .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。

第二步，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'
```

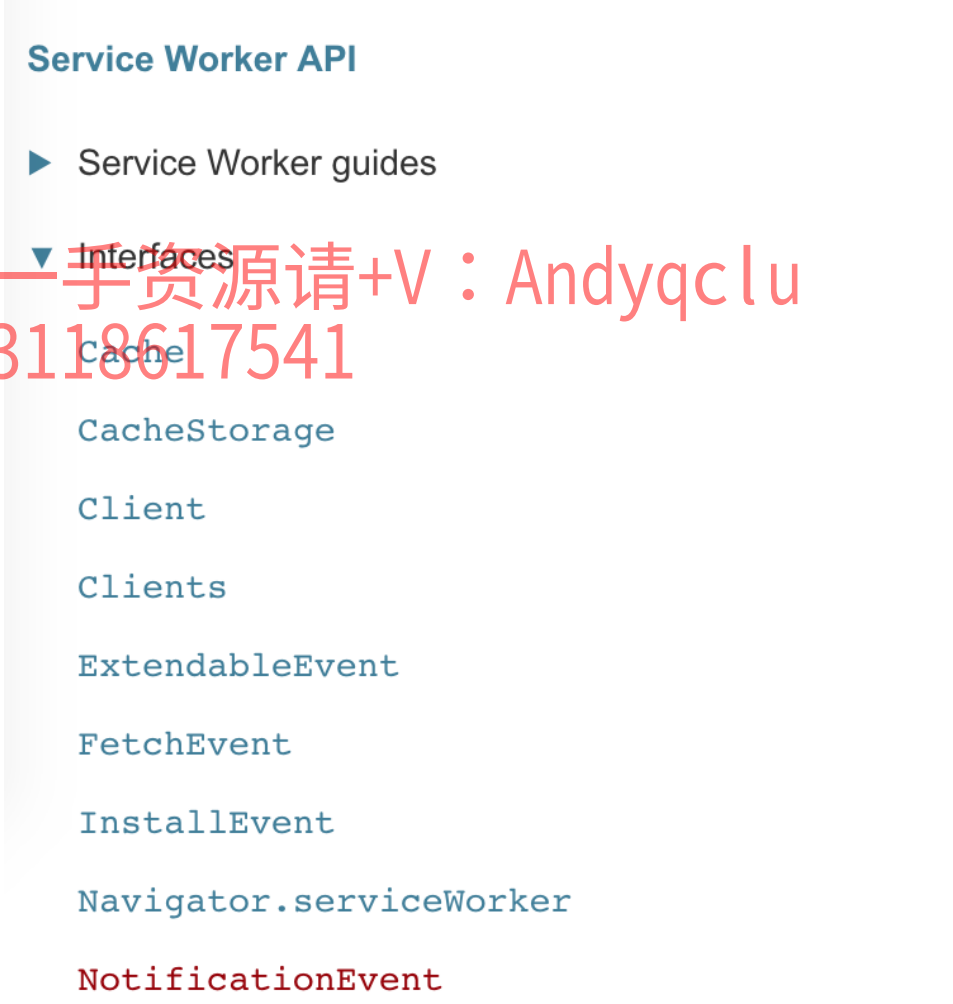
第三步，监听install事件，来进行相关文件的缓存操作：

```
// 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 对象匹配的第一个已经缓存的请求。
Cache.matchAll(request, options) 返回一个Promise 对象，resolve的结果是跟Cache对象匹配的所有请求组成的数组。
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，并且返回一个resolve为true的Promise对象；如果未找到，则返回一个resolve为false的Promise对象。
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。

第三步，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，于是来到第四步。

第四步，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：

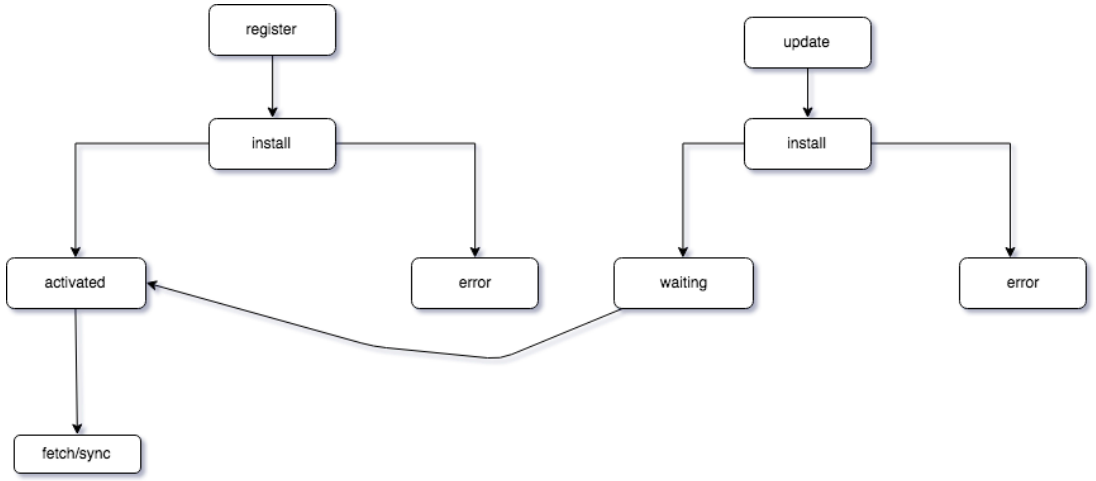

```
// 监听activate事件，激活后通过cache的key来判断是否更新cache中的静态资源
self.addEventListener('activate', function (e) {
  console.log('Service Worker 状态：activate')
  var cachePromise = caches.keys().then(function (keys) {
    // 遍历当前scope使用的key值
    return Promise.all(keys.map(function (key) {
      // 如果新获取到的key和之前缓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

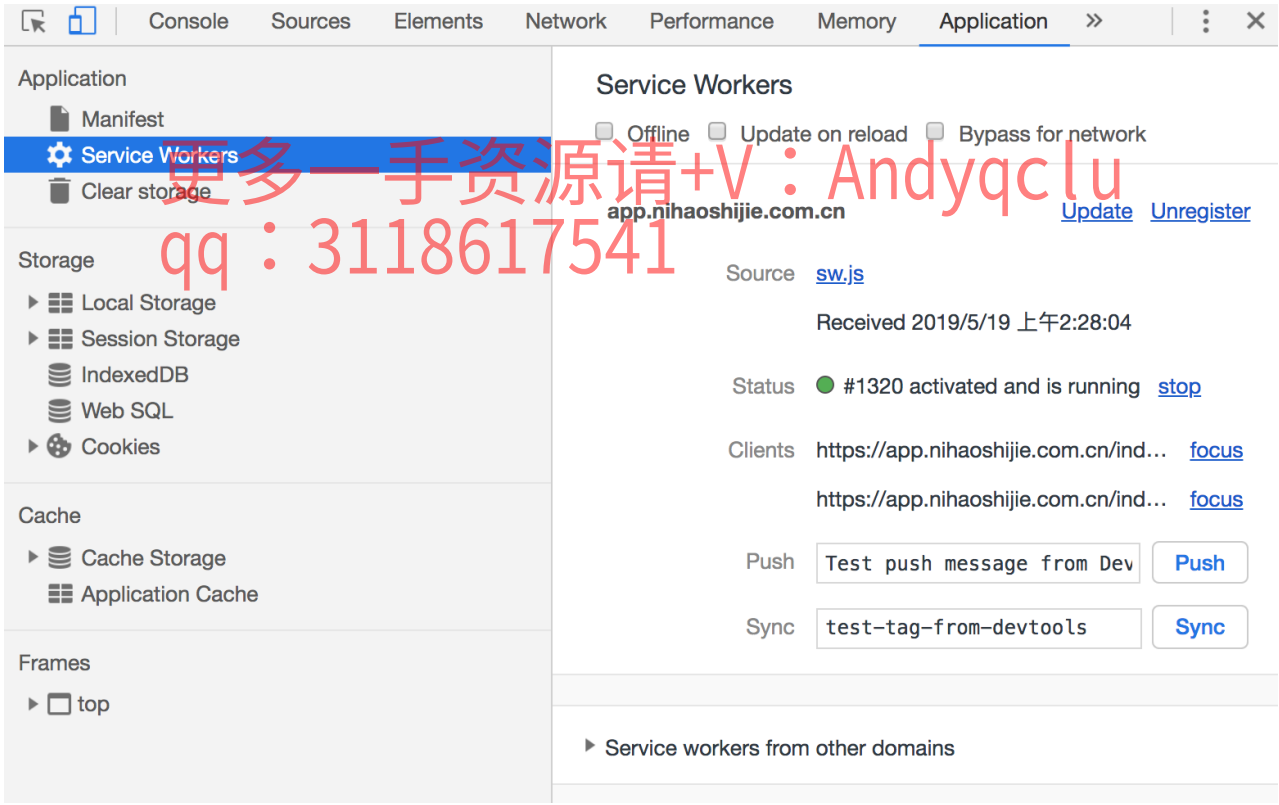
更新流程与安装类似，只是在更新安装成功后不会立即进入active状态，更新后的Service Worker会和原始的Service Worker共同存在，并运行它的install，一旦新Service Worker安装成功，它会进入wait状态，需要等待旧版本的Service Worker进/线程终止。

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: ''
    });
  }

})
```

这里说明一下：

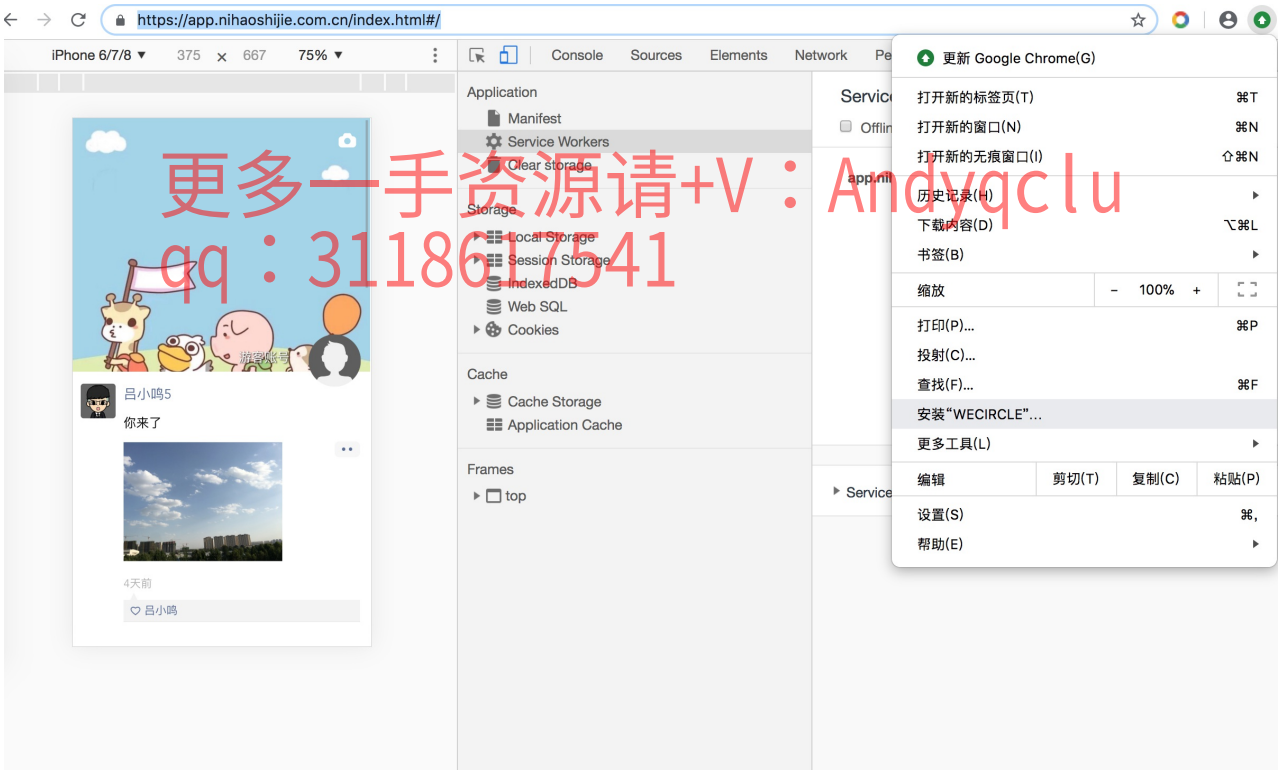
- 选择了offline-plugin插件之后呢，之前我们手写的注册Service Worker和Service Worker缓存相关逻辑都可以去掉

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。

OK，到这里我们的PWA改造就已经完成了，我们在执行npm run build命令之后，就会生成对应`的sw.js`文件，部署之后，我们就可以将页面保存到桌面，并且拥有了离线缓存，这看起来就像是一个Native App。这个文件名默认叫做`sw.js`，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：



小结

本章节主要讲解了PWA的相关知识，以及将我们的项目改成成一个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. manifest.json文件的各个配置项作用。
4. 拦截`fetch`事件，缓存前端静态资源文件的原理。
5. 结合offline-plugin插件，将项目改造成PWA应用。

本章节完整源代码地址：

Github-registerServiceWorker.js

Github-vue.config.js

Github-sw-my.js

}

30 页面转场动画

32 模拟真实APP完成消息推送