**了解PWA**

1. **概念**

PWA (Progressive Web Apps) 是一种 Web App 新模型，并不是具体指某一种前沿的技术或者某一个单一的知识点，我们从英文缩写来看就能看出来，这是一个渐进式的 Web App，是通过一系列新的 Web 特性，配合优秀的 UI 交互设计，逐步的增强 Web App 的用户体验。

用户的手机现在几乎被各种大大小小形形色色的 App 给攻占了，手机的容量是有上限的，用户的时间成本也是有上限的，我们如何让 App 做到用户需要就能立马得到，用户不需要不占用手机资源呢？显然 Native App 是根本做不到这点的，用户能做的只会在抱怨和留舍纠结中一遍一遍的重复着安装和卸载。而另一方面 Native App 没法做到平台兼容，对于企业和开发者来说相对于 Web App 的平台兼容性以及可维护行来说 Native App 的开发运维成本太高。

PWA 工程的解决方案中借助了 [service worker](https://lavas.baidu.com/doc/offline-and-cache-loading/service-worker/service-worker-introduction) 的离线存储能力，消息推送能力以及系统的添加桌面能力，从而形成一个完善的 Web App 解决方案，帮助我们在 Web 端低成本的开发和维护一个逐步类 Native App 化的 Web App

**1.1 核心功能**

Web App Manifest – 在主屏幕添加app图标，定义手机标题栏颜色之类

Service Worker – 缓存，离线开发，以及地理位置信息处理等

App Shell – 先显示APP的主结构，再填充主数据，更快显示更好体验

Push Notification – 消息推送

**2. Web App Manifest**

通过添加到主屏幕功能可以在用户桌面(主屏幕)上创建一个站点的快捷图标，实现和 Native APP 近似的使用体验。通过该图标打开的页面和浏览器中直接访问的体验有些不同，它在体验上更像 Native 应用

和 APP 一样在用户桌面(主屏幕)上存在图标

有 splash screen 启动屏

可以控制控制屏幕方向

可以设置全屏显示，没有浏览器的搜索框、导航栏

**2.2 创建manifest.json**

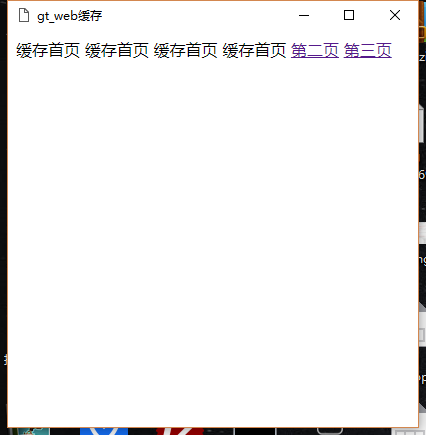
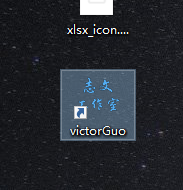
{  
 "scope": "/",  
 "name": "victorGuo",  
 "short\_name": "victorGuo",  
 "start\_url": "./index.html",  
 "display": "standalone", //使打开没有地址栏像app一样  
 "description": "victorGuo",  
 "dir": "rtl",  
 "lang": "cn",  
 "orientation": "portrait",  
 "theme\_color": "#3f51b5",  
 "background\_color": "#fff",  
 "icons": [  
 {  
 "src": "image/lzwme-36x36.png",  
 "sizes": "36x36",  
 "type": "image/png"  
 }, {  
 "src": "image/lzwme-48x48.png",  
 "sizes": "48x48",  
 "type": "image/png"  
 }, {  
 "src": "image/lzwme-72x72.png",  
 "sizes": "72x72",  
 "type": "image/png"  
 }, {  
 "src": "image/lzwme-96x96.png",  
 "sizes": "96x96",  
 "type": "image/png"  
 }, {  
 "src": "image/lzwme-144x144.png",  
 "sizes": "144x144",  
 "type": "image/png"  
 }, {  
 "src": "image/lzwme-192x192.png",  
 "sizes": "192x192",  
 "type": "image/png"  
 }, {  
 "src": "image/lzwme-256x256.png",  
 "sizes": "256x256",  
 "type": "image/png"  
 }  
 ]  
}

**2.3 引入**

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

**2.4效果**





**3 Service Worker**

**3.1生命周期**

install -> installed -> actvating -> Active -> Activated -> Redundant

**3. 2 service worker注册**

if ('serviceWorker' in navigator) {  
 navigator.serviceWorker  
 .register('./sw.js', {scope: './'})  
 .then(*registration* => console.log('ServiceWorker 注册成功！作用域为: ', *registration*.scope))  
 .catch(*err* => console.log('ServiceWorker 注册失败: ', *err*));  
} else {

Console.log('浏览器不支持serviceWorker!')  
}

### **3.3 Service Worker 主要事件**

### **3.3.1 install 事件**

当前脚本被安装时，会触发 install 事件

const *offlineResources* = [  
 './',  
 './offline.html',  
 './jquery.min.js',  
 './all.png'  
];

self.addEventListener('install', onInstall);  
function onInstall(*event*) {  
 log('install event in progress.');  
  
 *event*.waitUntil(  
 caches.open(cacheKey('offline'))  
 .then(*cache* => *cache*.addAll(*offlineResources*)) //添加需要缓存的静态资源  
 .then(() => log('installation complete! version: ' + *version*))  
 .then(() => self.skipWaiting())  
 );  
}

### **3.3.2 activate事件**

当安装完成后并进入激活状态，会触发 activate 事件。通过监听 activate 事件你可以做一些预处理，如对于旧版本的更新、对于无用缓存的清理等。

function removeOldCache() {  
 return caches  
 .keys()  
 .then(*keys* =>  
 Promise.all( // 等待所有旧的资源都清理完成  
 *keys* .filter(*key* => !*key*.startsWith(*version*)) // 过滤不需要删除的资源  
 .map(*key* => {  
 console.log('缓存', caches)  
 caches.delete(*key*)  
 }) // 删除旧版本资源，返回为 Promise 对象  
 )  
 )  
 .then(() => {  
 log('removeOldCache completed.');  
 });  
}  
  
function onActivate(*event*) {  
 log('activate event in progress.');  
 *event*.waitUntil(Promise.all([  
 // 更新客户端  
 self.clients.claim(),  
 removeOldCache()  
 ]))  
}

### **3.3.3 fetch 事件**

监控所有 http 请求，当请求资源已经在缓存里了，直接返回缓存里的内容；否则使用 fetch API 继续请求，如果是 图片或 css、js 资源，请求成功后将他们加入缓存中；如果是离线状态或请求出错，则降级返回预缓存的离线内容。

function onFetch(*event*) {  
 const request = *event*.request;  
 // 应当永远从网络请求的资源  
 // 如果请求失败，则使用离线资源替代  
 if (shouldAlwaysFetch(request)) {  
 console.log('fetch2', request)  
 log('AlwaysFetch request: ', *event*.request.url);  
 *event*.respondWith(networkedOrOffline(request));  
 return;  
 }  
 // 应当从网络请求并缓存的资源  
 // 如果请求失败，则尝试从缓存读取，读取失败则使用离线资源替代  
 if (shouldFetchAndCache(request)) {  
 *event*.respondWith(  
 networkedAndCache(request).catch(() => cachedOrOffline(request))  
 );  
 return;  
 }  
 *event*.respondWith(cachedOrNetworked(request));  
}

function onFetch(*event*) {  
 const request = *event*.request;  
 // 应当永远从网络请求的资源  
 // 如果请求失败，则使用离线资源替代  
 if (shouldAlwaysFetch(request)) {  
 console.log('fetch2', request)  
 log('AlwaysFetch request: ', *event*.request.url);  
 *event*.respondWith(networkedOrOffline(request));  
 return;  
 }  
 // 应当从网络请求并缓存的资源  
 // 如果请求失败，则尝试从缓存读取，读取失败则使用离线资源替代  
 if (shouldFetchAndCache(request)) {  
 *event*.respondWith(  
 networkedAndCache(request).catch(() => cachedOrOffline(request))  
 );  
 return;  
 }  
 *event*.respondWith(cachedOrNetworked(request));  
}

// 优先从 cache 读取，读取失败则从网络请求并缓存。网络请求也失败，则使用离线资源替代  
function cachedOrNetworked(*request*) {  
 return caches.match(*request*)  
 .then((*response*) => {  
 log(*response* ? '(cached)' : '(network: cache miss)', *request*.method, *request*.url);  
 return *response* ||  
 networkedAndCache(*request*)  
 .catch(() => offlineResponse(*request*));  
 });  
}

function offlineResponse(*request*) {  
 log('(offline)', *request*.method, *request*.url);  
 if (*request*.url.match(/\.(jpg|png|gif|svg|jpeg)(\?.\*)?$/)) {  
 return caches.match('/wp-content/themes/Kratos/images/default.jpg');  
 } else {  
 return caches.match('./offline.html');  
 }  
}

**3.4 service worker 更新**

如果 sw.js 文件的内容有改动，当访问网站页面时浏览器获取了新的文件，它会认为有更新，于是会安装新的文件并触发 install 事件。但是此时已经处于激活状态的旧的 Service Worker 还在运行，新的 Service Worker 完成安装后会进入 waiting 状态。直到所有已打开的页面都关闭，旧的 Service Worker 自动停止，新的 Service Worker 才会在接下来打开的页面里生效。

如果希望在有了新版本时，所有的页面都得到及时更新怎么办呢？

可以在 install 事件中执行 skipWaiting 方法跳过 waiting 状态，然后会直接进入 activate 阶段。接着在 activate 事件发生时，通过执行 clients.claim 方法，更新所有客户端上

手动更新

const *version* = '1.0.1';  
navigator.serviceWorker.register('/sw.js').then(*reg* => {  
 if (localStorage.getItem('sw\_version') !== *version*) {  
 *reg*.update().then(() => localStorage.setItem('sw\_version', *version*));  
 }  
});

1. **App Shell**

App Shell 架构是构建 PWA 应用的一种方式，它通常提供了一个最基本的 Web App 框架，包括应用的头部、底部、菜单栏等结构。顾名思义，我们可以把它理解成应用的一个「空壳」，这个「空壳」仅包含页面框架所需的最基本的 HTML 片段，CSS 和 javaScript，这样一来，用户重复打开应用时就能迅速地看到 Web App 的基本界面，只需要从网络中请求、加载必要的内容。我们使用 Service Worker 对 App Shell 做离线缓存，以便它可以在离线时正常展现，达到类似 Native App 的体验

**5. Notification消息推送**

5.1 Notification.requestPermission() 通过此方法获取授权

5.2 Notification.permission

这是一个静态属性。表示是否允许通知，三个字

granted：允许

denied：已拒绝

default：默认状态，需要授权

5.3 使用方法：

new Notification(title, options)

Title: 必须参数

Options：一个被允许用来设置通知的对象。它包含以下属性

{

dir : 文字的方向；它的值可以是 auto（自动）, ltr（从左到右）, or rtl（从右到左）

lang: 指定通知中所使用的语言。这个字符串必须在 [BCP 47 language tag](http://tools.ietf.org/html/bcp47" \o "http://tools.ietf.org/html/bcp47) 文档中是有效的。

body: 通知中额外显示的字符串

tag: 赋予通知一个ID，以便在必要的时候对通知进行刷新、替换或移除。

icon: 一个图片的URL，将被用于显示通知的图标。

}

5.4 效果

<**body** onoffline=**"**offline()**"**>

function offline () {  
 console.log('offline')  
 Notification.requestPermission().then(*grant* => {  
 console.log(*grant*)  
 if (*grant* !== 'granted') {  
 return;  
 }  
 const notification = new Notification("Hi，网络不给力哟", {  
 body: '您的网络貌似离线了!!!!',  
 icon: 'all.png'  
 });  
 notification.onclick = function () {  
 notification.close();  
 };  
 });  
}

