前言

在现代化的前端开发中，webpack已经成为了标配。与此同时，随着Google等一线互联网公司的大力推行，PWA(Progress Web Application，渐进式网络应用)的概念也逐渐进入了人们的视野。本文的内容，将会从搭建一个基于webpack的工程开始，介绍webpack相关的知识，同时引入PWA的概念，逐步把这个工程打造成一个完整的PWA应用，并最终部署上线。

在阅读这篇文章之前，要求读者了解webpack的基本概念，了解npm或者yarn的相关操作，这样在阅读的过程中才能把精力聚焦在重点的内容上。

项目构思

在大多数的前端应用当中，都喜欢拿TodoMVC（不是Hello World）作为例子。在这篇文章中，我们也通过构建一个TodoMVC来作为学习的例子。

todomvc.com这个网站提供了几乎所有框架/原生JS制作TodoMVC的例子，不同阵营的开发者都可以在这里找到自己想要的答案。为了简单起见，我将使用 VueJS 来构建这一个TodoMVC。当然，本文并不针对某个具体框架进行详细讨论，仅仅用其作为一个趁手的工具去完成我们的目的。

这个TodoMVC具有以下几个功能：

1. 能够添加一个todo；
2. 能够修改这个todo的状态为“完成”或者“激活”；
3. 页面能够按照“全部”、“完成”、“激活”来切换todo list的视图；
4. 提供删除、清空todo的操作。

通过分析上面的几个功能，我们不难发现，这个项目理应以“模块化”的形式去组织代码，因此也顺理成章地需要引入webpack来帮我们处理模块化加载的问题。

目录结构

项目已经部署到了https://jrainlau.github.io/TODOMVC，其中项目的关键目录结构如下：


```
├── package.json
├── favicons              # PWA专用的静态资源
├── index.html            # 开发模式下的html文件
├── index.tpl             # 生产模式下的html模板
├── src
|   ├── App.vue           # 根.vue文件
|   ├── assets            # 静态资源目录
|   ├── components        # 组件目录
|   └── main.js           # 入口js文件
├── webpack.config.js     # webpack配置文件
```

整个项目的核心代码均在 /src 目录下，后面使用webpack所处理的文件也基本都是在这个目录里面。

进入 main.js 文件，我们可以看到下列的基本代码：

```
import Vue from 'vue'
import App from './App.vue'

new Vue({
  el: '.todoapp',
  render: h => h(App)
})
```

这一段的代码的含义，就是注册一个 Vue实例 ，然后把根组件 App.vue 挂载在class名为 .todoapp 的html节点上。在接下来的章节里，对于这个应用的PWA改造，主要将会在这个关键的入口文件中进行。

Webpack配置

对于webpack的配置，有着多种多样的办法，比较主流的是把不同的功能区分成不同的配置文件，在运行的时候按条件指定。还有一种做法，就是在同一个配置文件内，按照不同的条件，通过 module.exports 把对应的配置暴露出去。在本文的例子中，由于代码逻辑并不复杂，为了方便展示，使用的是第二种办法。

作为一个相对完整的项目，本文所示例子的webpack配置也区分了**开发模式**与**生产模式**，环境的区分通过往 npm script 输入不同的参数实现。具体的webpack配置代码如下：

```
const path = require('path')
const webpack = require('webpack')
const cleanWebpackPlugin = require('clean-webpack-plugin')
const htmlWebpackPlugin = require('html-webpack-plugin')
const copyWebpackPlugin = require('copy-webpack-plugin')

module.exports = {
  entry: './src/main.js',
  output: {
    path: path.resolve(__dirname, './dist'),
    filename: 'build.js'
  },
  module: {
    rules: [
      // ...
    ]
  },
  devServer: {
    host: '0.0.0.0',
    historyApiFallback: true,
    noInfo: true
  },
  performance: {
    hints: false
  },
  devtool: '#eval-source-map'
}

if (process.env.NODE_ENV === 'production') {
  module.exports.devtool = '#source-map'
  module.exports.output = {
    path: path.resolve(__dirname, './docs'),
```

```
    filename: '[name].[hash].js'
  }
  module.exports.plugins = (module.exports.plugins ||
[]).concat([
    new webpack.DefinePlugin({
      'process.env': {
        NODE_ENV: '"production"'
      }
    }),
    // ...
    new htmlWebpackPlugin({
      filename: 'index.html',
      template: 'index.tpl',
      inject: true,
      minify: {
        removeComments: true,
        collapseWhitespace: true,
        removeAttributeQuotes: true
      },
      chunks: ['main'],
      chunksSortMode: 'dependency'
    }),
    new cleanWebpackPlugin(['docs']),
    new copyWebpackPlugin([
      { from: 'favicons' }
    ])
}
```


通过上述的配置项我们可以看到，生产环境和开发环境的配置大同小异，较大的区别有两处。

1. 开发模式所指定的输出文件，文件名并没有加入 hash 。
2. 在开发模式下，文件会被构建到 dist 目录下，通过根目录下的 index.html 运行；而生产模式，会使用 html-webpack-plugin ，根据模板 index.tpl 去**动态生成**html文件，然后被构建到 docs 目录下。

为什么要这么做呢？其实是跟浏览器的缓存策略有关。为了保证每一次的构建都能及时生效，我们希望浏览器能够拉取最新的资源。而为文件名添加 hash 值是一个行之有效的办法。然而，如果直接使用 index.html ，需要**手动**指定所需加载的资源，而借助 html-webpack-plugin ，我们就可以动态地注入**带有hash**的资源，省去了手动维护资源的麻烦。

在默认情况下，webpack会读取 webpack.config.js 作为它的配置文件，所以我们可以打开 package.json ，查看里面的 npm script 命令，看看我们应该如何对这个项目进行构建：

```
"scripts": {
  "dev": "cross-env NODE_ENV=development webpack-dev-server --
```

```
open --hot",
  "build": "cross-env NODE_ENV=production webpack --progress --
hide-modules"
},
```

构建指令非常简单，通过 `cross-env` 工具指定 `NODE_ENV` 的值，即可区分不同的环境。

什么是PWA

在搞定一个“普通项目”之后，我们就可以对其进行PWA升级了。在此之前，我们有必要了解一下到底什么是PWA。

PWA全称Progress Web Application，也就是渐进式web应用，是由Google在2015年就提出的概念。它本质上是一个普通的web页面，但是由于使用了Service Worker、Notification/Push API、Manifest等新的技术，使得这个web页面能够拥有媲美原生APP的用户体验，比如离线访问、消息推送等。

PWA具有以下几个优点：

1. **渐进式**：能运行在所有的浏览器中，只要支持PWA的浏览器都能获得性能和体验上的提升，不支持的浏览器也能正常访问。
2. **响应式**：能够运行在所有的设备当中，响应式适配不同大小的屏幕。
3. **允许离线访问**：得益于Service Worker的缓存机制，PWA允许离线使用。
4. **媲美原生APP**：拥有和原生APP相似的图标、启动画面、全屏体验等。
5. **更新及时**：Service Worker的更新机制让PWA永远处于最新版而无需用户重新手动更新。
6. **搜索引擎友好**：由于符合W3C标准的Manifest的存在，使得搜索引擎可以方便地收录PWA。
7. **交互性强**：通过Notification API等消息推送功能，使得PWA与用户的交互性更强。
8. **安装简单**：只要通过浏览器“添加到主屏”的功能即可安装PWA，省去在App Store寻找APP的麻烦。
9. **分享容易**：由于PWA只是一个web网站，所以只需要分享一条URL就可以把这个PWA分享出去。

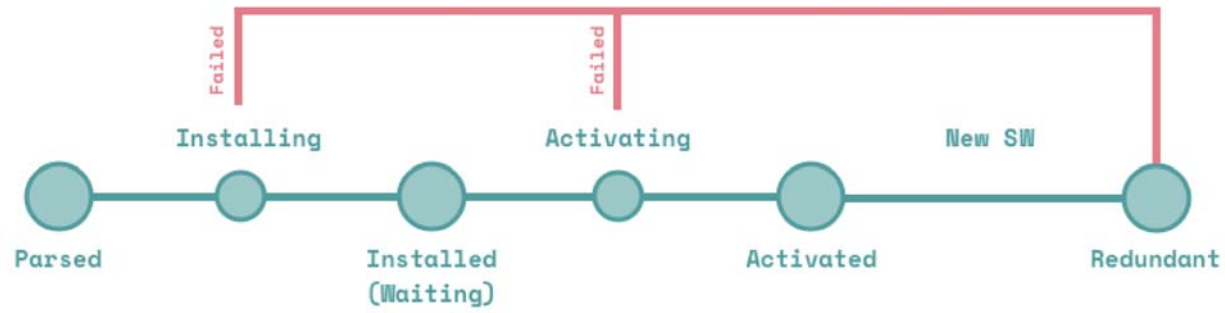
是不是觉得PWA非常强大，非常有研究价值呢？那么接下来，我们一起来看看它的几个核心知识点。

Service Worker

Service Worker，简单来说，是一个架设在“浏览器”与“服务器”中间的“代理”，它可以拦截浏览器与服务器的资源请求，同时对资源进行缓存。与Web Worker类似，它也是一个独立于浏览器的进程，能够在后台运行。

由于安全策略的问题，Service Worker只能运行在https协议，或者本地localhost/127.0.0.1开发域名下，所以要发布PWA的读者，需要把自己的网站升级成https。

学习Service Worker，最重要的一点是理解它的“生命周期”。我们可以通过两张图，直观地感受Service Worker的生命周期：



图片来源：

https://bitsofco.de/the-service-worker-lifecycle/?utm_source=javascriptweekly&utm_medium=email

enter image description here

图片来源：

https://github.com/delapuente/service-workers-101

INSTALLING 正在安装阶段

此阶段处于Service Worker被注册的时候。在这个阶段，Service Worker的 `install()` 方法将会被执行，声明的资源即将被加入缓存。

INSTALLED 安装完成阶段

当Service Worker完成其初始化安装之后就会进入到这个阶段。在这个阶段，它是一个“有效但尚未激活的worker”，等待着客户端对其进行激活使用。我们可以在这个阶段告知用户，PWA已经可以进行升级了。

ACTIVATING 正在激活阶段

当客户端已无其它激活状态的Service Worker、或者脚本中的 `skipWaiting()` 方法被调用、或者用户关闭了该Service Worker作用域下的所有页面时，就会触发ACTIVATING阶段。在这个阶段中，Service Worker脚本中的 `activate()` 事件会被执行，此时可以清除掉过期的资源，并将进入下一个“激活完成阶段”。

ACTIVETED 激活完成阶段

此时Service Worker已经被激活并生效，可以开始控制网站的资源请求了。此时Service Worker内的 `fetch` 和 `message` 事件已经可以被监听。

REDUNDANT 废弃阶段

当安装失败，激活失败或者当前Service Worker被其他Service Worker替换时，就会进入这个阶段，此时Service Worker将失去对页面的控制。

> 值得注意的是，上面说的“生命周期”，并非指某个具体的事件，而是“在这个阶段有什么作用，可以调用什么方法”的意思。初学者很有可能会弄混“周期”与“事件”之间的关系。
>
> 更多关于Service Worker生命周期的内容，可以参考下面这篇文章：
>
> https://bitsofco.de/the-service-worker-lifecycle/?utm_source=javascriptweekly&utm_medium=email

Manifest文件

前面提到，PWA具有和原生APP相似的用户体验，其中“添加到主屏”可以说是最关键的部分。和普通网页以书签形式添加到主屏不同，PWA在添加到主屏以后，可以做到和原生APP一样全屏展示，在任务管理器里面也是独立于浏览器进程的存在。同时它还允许开发者指定图标、指定APP名和指定启动画面——这一切都需要Manifest文件的支持。

Manifest是一个普通的 `json` 文件，它位于网站的根目录下，里面声明了这个PWA的“基本信息”，一个典型的 `manifest.json` 文件如下：

```
{
  "name": "Todo",
  "icons": [
    {
      "src": "android-chrome-192x192.png",
      "sizes": "192x192",
```

```
      "type": "image/png"
    },
    {
      "src": "android-chrome-512x512.png",
      "sizes": "512x512",
      "type": "image/png"
    }
  ],
  "theme_color": "#ffffff",
  "background_color": "#ffffff",
  "start_url": "/TodoMVC-PWA/",
  "display": "standalone"
}
```

在这个 `manifest.json` 文件中，我们可以轻松得到这个PWA的信息：

- name：定义此PWA的名称。
- icons：定义一系列的图标以适应不同型号的设备。
- theme_color：主题颜色（影响手机状态栏颜色）。
- background_color：背景颜色。
- start_url：启动地址。由于PWA实际上是一个web页面，所以需要定义PWA在启动时应该访问哪个地址。
- display：“standalone”表示其以类似原生APP的全屏方式启动。

在定义好 `manifest.json` 文件后，我们可以通过Chrome的开发者工具看到详细的内容：

下面几个网站都可以在线帮我们生成想要的 `manifest.json` 文件，非常方便：

- https://app-manifest.firebaseapp.com/
- https://tomitm.github.io/appmanifest/
- https://brucelawson.github.io/manifest/

到目前为止，iOS并未支持PWA，但我们可以通过几个由苹果公司提供的html标签使web页面获得和原生应用类似的效果。

应用图标：

```
<link rel="apple-touch-icon" href=“apple-touch-icon.png">
```

启动画面：

```
<link rel="apple-touch-startup-image" href="launch.png">
```

应用名称：

```
<meta name="apple-mobile-web-app-title" content="Todo-PWA">
```

全屏效果：

```
<meta name="apple-mobile-web-app-capable" content="yes">
```

设置状态栏颜色：

```
<meta name="apple-mobile-web-app-status-bar-style"
content="#fff">
```

至此，PWA已经适配了Android和iOS系统，分别使用Chrome和Safari浏览器的“添加到主屏”功能即可。感兴趣的读者可以访问下面这个地址进行体验：

https://jrainlau.github.io/TodoMVC-PWA/

App Shell

如果打开一个大型的web页面，“白屏”的体验是非常糟糕的。在PWA中，我们可以通过App Shell的办法优化这个问题。

App Shell，顾名思义，就是“壳”的意思，也可以理解为“骨架屏”，说白了就是在内容尚未加载完全的时候，优先展示页面的结构、占位图、主题和背景颜色等，它们都是一些被强缓存的html，css和javascript。

要用好App Shell，就必须保证这部分的资源被Service Worker缓存起来。我们在组织代码的时候，可以优先完成App Shell的部分，然后把这部分代码分别打包构建出来。

使用Offine-Plugin把网站升级成PWA

前面铺垫了那么多，终于要进行实际的操作了。由于是借助webpack构建的项目，因此我们可以很方便地使用这个名叫 `Offine-Plugin` 的webpack插件帮助我们升级PWA。

PWA的核心可谓是Service Worker，任何一个PWA都有且只有一个 `service-worker.js` 文件，用于为Service Worker添加资源列表，进行注册、激活等生命周期操作。但是在

webpack构建的项目中，生成一个 service-worker.js 可能会面临两个较大的问题：

1. webpack生成的资源多会生成一串hash，Service Worker的资源列表里面需要同步更新这些带hash的资源；
2. 每次更新代码，都需要通过更新 service-worker.js 文件版本号来通知客户端对所缓存的资源进行更新。（其实只要这一次的 service-worker.js 代码和上一次的 service-worker.js 代码不一样即可触发更新，但使用明确的版本号会更加合适）。

相比与 Service Worker-precache-webpack-plugin ，个人认为 offline-plugin 具有如下优点：

1. 更多的可选配置项，满足更加细致的配置要求；
2. 更为详细的文档和例子；
3. 更新频率相对更高，star数更多；
4. 自动处理生命周期，用户无需纠结生命周期的坑；

接下来将会介绍这个插件的用法。

基本使用

安装

```
npm install offline-plugin [--save-dev]
```

初始化

第一步，进入webpack.config:

```
// webpack.config.js example

var OfflinePlugin = require('offline-plugin');

module.exports = {
  // ...

  plugins: [
    // ... other plugins
    // it's always better if OfflinePlugin is the last plugin
added
    new OfflinePlugin()
  ]
```

```
  // ...
}
```

第二步，把runtime添加到你的入口js文件当中：

```
require('offline-plugin/runtime').install();
ES6/Babel/TypeScript

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

经过上面的步骤，offline-plugin已经集成到项目之中，通过webpack构建即可。

配置

前面说过， `offline-plugin` 支持细致的配置，以满足不同的需求。下面将介绍几个比较常用的配置项，方便大家进一步使用。

- `Caches: 'all' | Object`

  告诉插件应该缓存什么东西，并以何种方式进行缓存。

  `all`：意味着所有webpack构建出来的资源，以及在 `externals` 选项中的资源都会被缓存。

  `Object`：包含三个数组或正则的配置对象（ `main`，`additional`，`optional` ），它们都是可选的，且默认为空。

  默认： `all`。

- `externals: Array<string>`

  允许开发者指定一些外部资源（比如CDN引用，或者不是通过webpack生成的资源）。配合 `Caches` 的 `additional` 项，能够实现缓存外部资源的功能。

  默认： `null`。

  举例： `['fonts/roboto.woff']`。

- `ServiceWorker: Object | null | false`

  该对象包含多个配置项，这里仅列举最常用的。

  `events` ：布尔值。允许runtime接受来自Service Worker的消息，默认值为false。

  `navigateFallbackURL` ：当一个URL请求从缓存或网络都无法被获取时，将会重定向到该选项所指向的URL。

- `AppCache: Object | null | false`

  `offline-plugin` 默认支持 `AppCache` ，但是 `AppCache` 草案已经被web标准所废弃，不建议使用。

  但是由于仍然有部分浏览器支持，所以插件默认提供这个功能。

runtime

上一节介绍了 `offline-plugin` 在webpack当中的配置，这一节将介绍runtime的一些用法。
若要使 `offline-plugin` 生效，用户必须在入口js文件中通过runtime进行初始化操作：

```
// 通过AMD方式
require('offline-plugin/runtime').install();

// 或者通过ES6/Babel/TypeScript方式

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

`OfflinePluginRuntime` 对象提供了下列三个方法：

- `install(options: Object)`

  开启ServiceWorker/AppCache的安装流程。这个方法是安全的，并且必须在页面初始化的时候就被调用。另外请勿把它放在任何的条件语句之内。（这句话不全对，在后面的降级方案里面会详细介绍）

- `applyUpdate()`

  接受当前所安装的Service Worker的更新信息。

- `update()`

  检查新版本的ServiceWorker/AppCache的更新信息。

  `runtime.install()` 方法接受一个配置对象参数，用于处理Service Worker各个生命周期里面的事件。

- `onInstalled`

  当ServiceWorker/AppCache被install时执行，可用于展示“APP已经支持离线访问”。

- `onUpdating`

  AppCache不支持该方法。

当更新信息被获取且浏览器正在进行资源更新时触发。在这个时刻，一些资源正在被下载。

- `onUpdateReady`

  当 `onUpdating` 事件完成时触发。这时，所有资源都已经下载完毕。

  通过调用 `runtime.applyUpdate()` 方法来触发更新。

- `onUpdateFailed`

  当 `onUpdating` 事件因为某些原因失败时触发。

  这时没有任何资源被下载，同时所有的资源更新进程都应该被取消或跳过。

- `onUpdated`

  当更新被接受时触发。

降级方案

当某些时候我们需要撤掉Service Worker进行降级的时候，我们需要主动注销Service Worker。然而 `offline-plugin` 默认没有提供注销Service Worker的 `unregister()` 方法，所以我们需要自己实现。

其实要主动注销Service Worker非常简单，我们可以直接调用 `ServiceWorkerContainer.getRegistrations()` 方法来拿到 `registration` 实例，然后调用 `registration.unregister()` 方法即可，具体代码如下：

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.getRegistration().then((registration)
=> {
    registration && registration.unregister().then((boolean) => {
      boolean ? alert('注销成功') : alert('注销失败')
    });
  })
}
```

在调用该方法后，Service Worker已经被注销，刷新一下页面就能看到资源是重新从网络获取的了。

在真实的生产环境中，我们可以通过调用接口，来决定是否使用降级方案：

```
fetch(URL).then((Service Workeritch) => {
  if (Service Workeritch) {
    OfflinePluginRuntime.install()
  } else {
    if ('serviceWorker' in navigator) {
```

```
navigator.serviceWorker.getRegistration().then((registration) =>
{
        registration && registration.unregister().then((boolean)
=> {
          boolean ? alert('注销成功') : alert('注销失败')
        })
      })
    }
  }
})
```

尾声

由于国内众所周知的原因，依赖Google服务的 `Notification API` 无法使用，所以关于PWA消息推送这一块暂时未付诸实践，待将来有时间再进行深入研究。

PWA是前端的未来，也是已经到达的未来。非常期待各位读者自行尝试，与我一起交流探讨，共同推进PWA在国内的发展。