

Progressive Web App 教程

编者 xiedrsz



中科软信息系统有限公司

sinosoft

目录

[简介 2](#_Toc476041853)

[概述 2](#_Toc476041854)

[搭建开发环境 2](#_Toc476041855)

[Hello World 2](#_Toc476041856)

# 简介

## 概述

**Progressive Web Apps** 是结合了 Web 和 原生应用中最好功能的一种体验。对于首次访问的用户它是非常有利的, 用户可以直接在浏览器中进行访问，不需要安装应用。随着时间的推移当用户渐渐地和应用建立了联系，它将变得越来越强大。它能够快速地加载，即使在比较糟糕的网络环境下，能够推送相关消息, 也可以像原生应用那样添加至主屏，能够有全屏浏览的体验。

## 特性

**渐进增强** - 能够让每一位用户使用，无论用户使用什么浏览器，因为它是始终以渐进增强为原则。

**响应式用户界面** - 适应任何环境：桌面电脑，智能手机，笔记本电脑，或者其他设备。

**不依赖网络连接** - 通过 service workers 可以在离线或者网速极差的环境下工作。

**类原生应用** - 有像原生应用般的交互和导航给用户原生应用般的体验，因为它是建立在 app shell model 上的。

**持续更新** - 受益于 service worker 的更新进程，应用能够始终保持更新。

**安全** - 通过 HTTPS 来提供服务来防止网络窥探，保证内容不被篡改。

**可发现** - 得益于 W3C manifests 元数据和 service worker 的登记，让搜索引擎能够找到 web 应用。

**再次访问** - 通过消息推送等特性让用户再次访问变得容易。

**可安装** - 允许用户保留对他们有用的应用在主屏幕上，不需要通过应用商店。

**可连接性** - 通过 URL 可以轻松分享应用，不用复杂的安装即可运行。

## 通过本教程你能学到什么？

1. 如何使用 "app shell" 的方法来设计和构建应用程序。
2. 如何让你的应用程序能够离线工作。
3. 如何存储数据以在离线时使用。

## 软件环境

Chrome 52 或以上

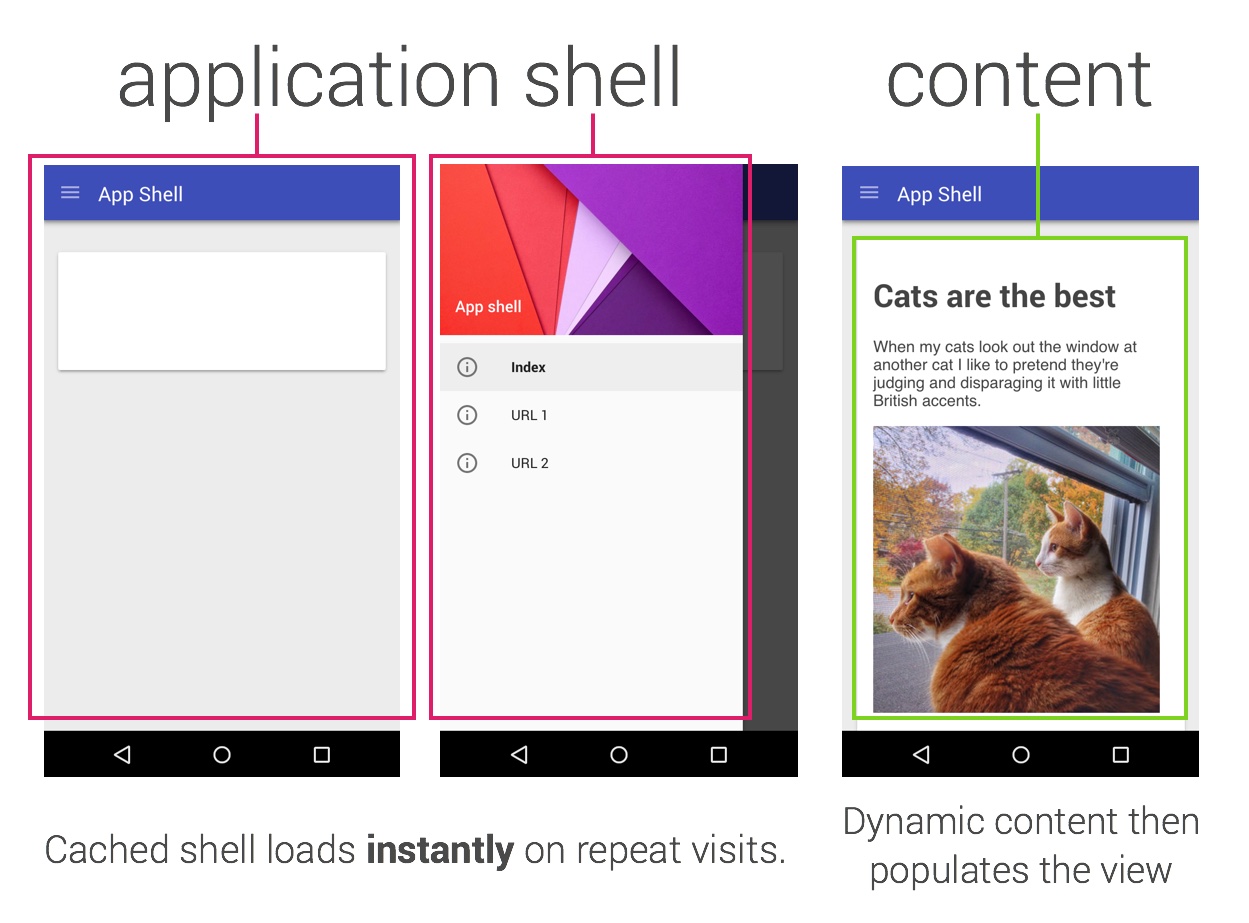
Web Server for Chrome 或其他的网络服务器

# 基于应用外壳的架构

## 应用外壳(App Shell)

**App Shell**是应用的用户界面所需的最基本的 HTML、CSS 和 JavaScript，也是一个用来确保应用有好多性能的组件。它的首次加载将会非常快，加载后立刻被缓存下来。这意味着应用的外壳不需要每次使用时都被下载，而是只加载需要的数据。

应用外壳的结构分为应用的核心基础组件和承载数据的 UI。所有的 UI 和基础组件都使用一个 service worker 缓存在本地，因此在后续的加载中 Progressive Web App 仅需要加载需要的数据，而不是加载所有的内容。



换句话说，应用的壳相当于那些发布到应用商店的原生应用中打包的代码。它是让你的应用能够运行的核心组件，只是没有包含数据。

## 为什么使用基于应用外壳的结构?

使用基于应用外壳的结构允许你专注于速度，给你的 Progressive Web App 和原生应用相似的属性：快速的加载和灵活的更新，所有这些都不需要用到应用商店。

## 设计应用外壳

在设计一个更加复杂的应用时，内容不需要在首次全部加载，可以在之后按需加载，然后缓存下来供下次使用。因此我们必须清楚：

1. 需要立刻显示什么在屏幕上？
2. 我们的应用需要那些关键的 UI 组件？
3. 应用外壳需要那些资源？比如图片，JavaScript，样式表等等。

## 为应用外壳编写 HTML 代码（以 天气预报 应用为例）

为了保证我们的起步代码尽可能清晰，我们将会开始于一个新的 index.html 文件并添加核心组件的代码。

核心组件包括：

1. 包含标题的头部，以及头部的 添加/刷新 按钮
2. 放置天气预报卡片的容器
3. 天气预报卡片的模板
4. 一个用来添加城市的对话框
5. 一个加载指示器

# service worker

Progressive Web Apps 是快速且可安装的，这意味着它能在在线、离线、断断续续或者缓慢的网络环境下使用。为了实现这个目标，我们需要使用一个 **service worker** 来缓存应用外壳，以保证它能始终迅速可用且可靠。

service workers 提供的是一种应该被理解为渐进增强的特性，这些特性仅仅作用于支持service workers 的浏览器。比如，使用 service workers 你可以缓存应用外壳和你的应用所需的数据，所以这些数据在离线的环境下依然可以获得。使用特性检测来渐渐增强有一些小的开销，它不会在老旧的不支持 service workers 的浏览器中产生破坏性影响。

## 注册 service worker

为了让应用离线工作，要做的第一件事是注册一个 service worker，一段允许在后台运行的脚本，不需要 用户打开 web 页面，也不需要其他交互。

这只需要简单两步：

1. 创建一个 JavaScript 文件作为 service worker
2. 告诉浏览器注册这个 JavaScript 文件为 service worker

第一步，在你的应用根目录下创建一个空文件叫做 service-worker.js 。这个 service-worker.js 文件必须放在跟目录，因为 service workers 的作用范围是根据其在目录结构中的位置决定的。

接下来，我们需要检查浏览器是否支持 service workers，如果支持，就注册 service worker，将下面代码添加至 app.js中。

if ('serviceWorker' in navigator) {

navigator.serviceWorker

.register('./service-worker.js')

.then(function () {

console.log('Service Worker Registered');

});

}

## 缓存站点的资源

当 service worker 被注册以后，当用户首次访问页面的时候一个 install 事件会被触发。在这个事件的回调函数中，我们能够缓存所有的应用需要再次用到的资源。

当 service worker 被激活后，它应该打开缓存对象并将应用外壳需要的资源存储进去。将下面这些代码加入你的 service-worker.js (你可以在your-first-pwapp-master/work中找到) ：

var cacheName = 'weatherPWA-final-1';

var filesToCache = [

'/',

'/index.html',

'/scripts/app.js',

'/styles/inline.css',

'/images/clear.png',

'/images/cloudy-scattered-showers.png',

'/images/cloudy.png',

'/images/fog.png',

'/images/ic\_add\_white\_24px.svg',

'/images/ic\_refresh\_white\_24px.svg',

'/images/partly-cloudy.png',

'/images/rain.png',

'/images/scattered-showers.png',

'/images/sleet.png',

'/images/snow.png',

'/images/thunderstorm.png',

'/images/wind.png'

];

self.addEventListener('install', function(e) {

console.log('[ServiceWorker] Install');

e.waitUntil(

caches.open(cacheName).then(function(cache) {

console.log('[ServiceWorker] Caching app shell');

return cache.addAll(filesToCache);

})

);

});

首先，我们需要提供一个缓存的名字并利用 **caches.open()**打开 cache 对象。提供的缓存名允许我们给 缓存的文件添加版本，或者将数据分开，以至于我们能够轻松地升级数据而不影响其他的缓存。

# 额外收获

1. **localStorage** 是阻塞型同步的存储机制，在某些设备上可能很缓慢。

# 参考文献

参考教程（天气预报）：<http://www.w3cplus.com/pwa/your-first-pwapp.html>

Service Workers简介：

<https://developers.google.com/web/fundamentals/getting-started/primers/service-workers>

# 搭建开发环境

windows Android react native 开发环境已搭建完成，详见《react native开发环境搭建》

# Hello World

程序代码：

import React, { Component } from 'react';

import { AppRegistry, Text } from 'react-native';

class HelloWorldApp extends Component {

render() {

return (

<Text>Hello world!</Text>

);

}

}

// 注意，这里用引号括起来的'HelloWorldApp'必须和你init创建的项目名一致

AppRegistry.registerComponent('HelloWorldApp', () => HelloWorldApp);

你可以新建一个项目，然后用上面的代码覆盖你的index.ios.js或是index.android.js 文件，然后运行看看。

React Native内置了对**ES2015**标准的支持，你可以放心使用而无需担心兼容性问题。上面的示例代码中的import、from、class、extends、以及() =>箭头函数等新语法都是ES2015中的特性。

上面的代码定义了一个名为HelloWorldApp的新的组件（**Component**），并且使用了名为**AppRegistry**的内置模块进行了“注册”操作。你在编写React Native应用时，肯定会写出很多新的组件。而一个App的最终界面，其实也就是各式各样的组件的组合。组件本身结构可以非常简单——唯一必须的就是在render方法中返回一些用于渲染结构的JSX语句。

AppRegistry模块则是用来告知React Native哪一个组件被注册为整个应用的根容器。你无需在此深究，因为**一般在整个应用里AppRegistry.registerComponent这个方法只会调用一次**。上面的代码里已经包含了具体的用法，你只需整个复制到index.ios.js或是index.android.js文件中即可运行。

# Props属性

大多数组件在创建时就可以使用各种参数来进行定制。用于定制的这些参数就称为props（属性）。

以常见的基础组件Image为例，在创建一个图片时，可以传入一个名为source的prop来指定要显示的图片的地址，以及使用名为style的prop来控制其尺寸。

代码如下：

import React, { Component } from 'react';

import { AppRegistry, Image } from 'react-native';

class Bananas extends Component {

render() {

let pic = {

uri: 'https://upload.wikimedia.org/wikipedia/commons/d/de/Bananavarieties.jpg'

};

return (

<Image source={pic} style={{width: 193, height: 110}} />

);

}

}

AppRegistry.registerComponent('Bananas', () => Bananas);

注：在iOS上使用http链接的图片地址可能不会显示，参见：

<https://segmentfault.com/a/1190000002933776>

请注意{pic}外围有一层括号，我们需要用括号来把pic这个变量嵌入到JSX语句中。括号的意思是括号内部为一个js变量或表达式，需要执行后取值。因此我们可以把任意合法的JavaScript表达式通过括号嵌入到JSX语句中。

自定义的组件也可以使用props。通过在不同的场景使用不同的属性定制，可以尽量提高自定义组件的复用范畴。只需在render函数中引用this.props，然后按需处理即可。下面是一个例子：

import React, { Component } from 'react';

import { AppRegistry, Text, View } from 'react-native';

class Greeting extends Component {

render() {

return (

<Text>Hello {this.props.name}!</Text>

);

}

}

class LotsOfGreetings extends Component {

render() {

return (

<View style={{alignItems: 'center'}}>

<Greeting name='Rexxar' />

<Greeting name='Jaina' />

<Greeting name='Valeera' />

</View>

);

}

}

AppRegistry.registerComponent('LotsOfGreetings', () => LotsOfGreetings);

上面的例子出现了一样新的名为View的组件。View 常用作其他组件的容器，来帮助控制布局和样式。

# State状态

我们使用两种数据来控制一个组件：**props**和**state**。props是在父组件中指定，而且一经指定，在被指定的组件的生命周期中则不再改变。 对于需要改变的数据，我们需要使用state。

一般来说，你需要在**constructor**中初始化state（译注：这是ES6的写法，早期的很多ES5的例子使用的是getInitialState方法来初始化state，这一做法会逐渐被淘汰），然后在需要修改时调用**setState**方法。

假如我们需要制作一段不停闪烁的文字。文字内容本身在组件创建时就已经指定好了，所以文字内容应该是一个prop。而文字的显示或隐藏的状态（快速的显隐切换就产生了闪烁的效果）则是随着时间变化的，因此这一状态应该写到state中：

import React, { Component } from 'react';

import { AppRegistry, Text, View } from 'react-native';

class Blink extends Component {

constructor(props) {

super(props);

this.state = { showText: true };

// 每1000毫秒对showText状态做一次取反操作

setInterval(() => {

this.setState({ showText: !this.state.showText });

}, 1000);

}

render() {

// 根据当前showText的值决定是否显示text内容

let display = this.state.showText ? this.props.text : ' ';

return (

<Text>{display}</Text>

);

}

}

class BlinkApp extends Component {

render() {

return (

<View>

<Blink text='I love to blink' />

<Blink text='Yes blinking is so great' />

<Blink text='Why did they ever take this out of HTML' />

<Blink text='Look at me look at me look at me' />

</View>

);

}

}

AppRegistry.registerComponent('BlinkApp', () => BlinkApp);

实际开发中，我们一般不会在定时器函数（setInterval、setTimeout等）中来操作state。典型的场景是在接收到服务器返回的新数据，或者在用户输入数据之后。你也可以使用一些“状态容器”比如Redux来统一管理数据流（译注：但我们不建议新手过早去学习redux <http://redux.js.org/index.html>）。

State的工作原理和React.js完全一致。