Weex入坑

基本信息

- 官方支持iOS、Android、HTML5.
- Write Once, Run Everywhere。一次编写可生成三平台代码。
- DSL模板学习超简单，直接写HTML、CSS、JS。这意味着可以直接用现有编辑器和IDE的代码补全、提示、检查等功能。
- 轻量级、可扩展、高性能。
- 集成花样多，可在HTML5页面嵌入，也可嵌在原生UI中。

历史和背景

Weex的前身是WeApp，一个用JSON配置原生UI组件来实现动态化的框架，Weex是WeApp的进化版本，加上ex去掉App，就成了现在这个名字。

他们还编了个段子：

> You give us a few weeks, so we bring you a little weex.

动态化的一些解决方案

- 传统 Hybrid：基于嵌入 native app 的 webview，通过对 API 的扩展达到解决问题的目的。
- 为 Hybrid 引入 native 界面：大概做法是以类似 plugin 的方式把一块 native 界面引入到 webview 当中，这样在整体还是 webview 的情况下，可以局部达到特殊的 native 效果。
- WeApp：这是阿里巴巴无线事业部2年前开启的一个项目，设计一套 JSON 格式，可以描述界面的结构和样式，然后各端实现对这套 JSON 格式的解析和渲染。这样只要每次请求不一样的 JSON 数据，就可以展示出不一样的界面效果，和之前的 Hybrid 方案不同的是，在 native app 里的展示效果是 100% native 的，并且还做了 HTML5 版本的“降级”渲染方案，用户在浏览器里同样有机会把整个界面展示出来。
- React Native：它毫无疑问是目前全球最火热的移动技术方案之一，界面是 100% 的 native，彻底颠覆了移动应用的技术栈和开发体验。

最终Weex诞生

- 致力于移动端
- 能够充分调度 native 的能力
- 能够充分解决和回避性能瓶颈
- 能够灵活扩展
- 能够多端统一
- 能够优雅“降级”到 HTML5
- 能够保持较低的开发成本
- 能够快速迭代
- 能够轻量实时发布
- 能够融入现有的 native 技术体系
- 能够工程化管理和监控

特点

轻量

体积小巧，语法简单，方便上手

可扩展

业务方可自行横向定制 native 组件和 API

高性能

快速加载，快速渲染，体验流畅

其它

- 拥抱标准：基于 Web 标准设计语法。
- 响应式界面： 通过简单的**模板**和**数据绑定**轻松解决数据和视图的同步关联问题。
- 多端统一：iOS、Android、HTML5 多端效果一致，撰写一次就可以轻松达到跨平台的一致性，无需针对多套平台单独开发，省时省力。
- 复杂逻辑描述：动态性不只体现在展示效果的动态性上，更体现在可以实时调整复杂的数据处理方式和逻辑控制方式。
- 组件化：组件之间通过 webcomponents 的设计完美的隔离，并可以通过特定的方式进行数据和事件的传递。
- 生态&链路：Weex 为开发者和使用者在不同维度上提供了各式各样的工具和平台，包括代码打包工具、开发者调试工具、部署平台、Playground、经典案例、入门指南和详尽的文档等。你不是从零开始，你也不是一个人在战斗！

技术相关

本地组件开发

首先，我们像开发 webcomponents 一样，把一个组件分成 `<template>`、`<style>`、`<script>` 三部分，刚好对应一个组件的界面结构、界面样式、数据&逻辑。

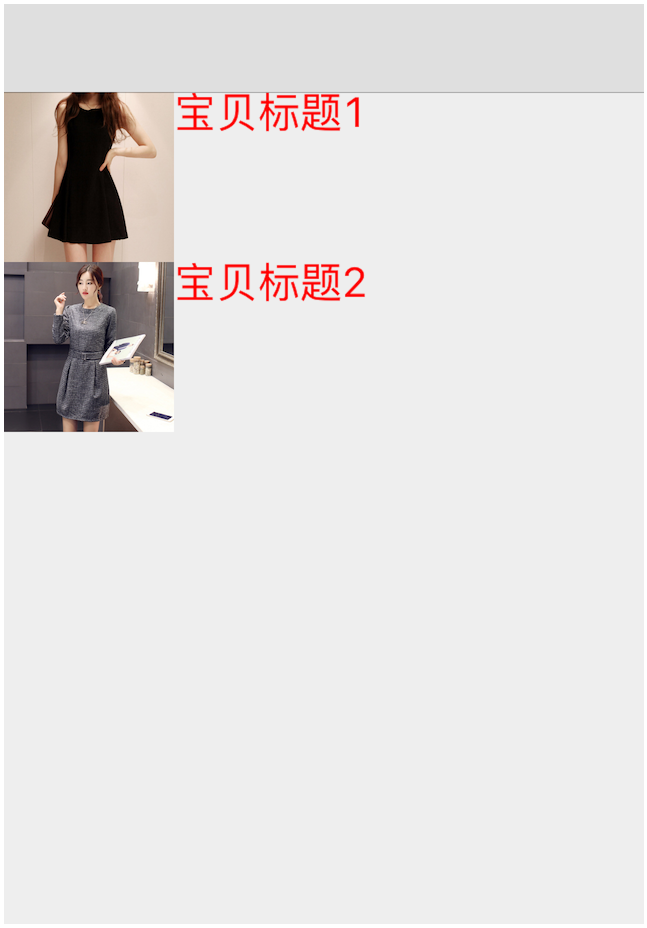
```
<template>
  <container style="flex-direction: column;">
    <container repeat="{{itemList}}" onclick="gotoDetail">
      <image class="thumb" src="{{pictureUrl}}"></image>
      <text class="title">{{title}}</text>
    </container>
  </container>
</template>

<style>
  .thumb {width: 200; height: 200;}
  .title {flex: 1; color: #ff0000; font-size: 48; font-weight: bold;
background-color: #eeeeee;}
</style>

<script>
  module.exports = {
    data: {
      itemList: [
        {itemId: '520421163634', title: '1', pictureUrl:
'https://gd2.alicdn.com/bao/uploaded/i2/T14H1LFwBcXXXXXXXX_!!0-item_pic.jp
g'},
        {itemId: '522076777462', title: '2', pictureUrl:
'https://gd1.alicdn.com/bao/uploaded/i1/TB1PXJCJFXXXXciXFXXXXXXXXXX_!!0-it
em_pic.jpg'}
      ]
    },
    methods: {
      gotoDetail: function () {
        this.$openURL('https://item.taobao.com/item.htm?id=' + this.itemId)
      }
    }
  }
</script>
```

1. 在本地用一个叫做 transformer 的工具把这套代码转成纯 JavaScript 代码。
2. 在客户端运行一个 JavaScript 引擎，随时接收 JavaScript 代码。
3. 在客户端设计一套 JS Bridge，让 native 代码可以和 JavaScript 引擎相互通信。

客户端渲染



native 渲染和 JavaScript 引擎之间，主要进行三类通信：

1. 界面渲染，单向 (JS -> native)：这毫无疑问，JavaScript 引擎需要把界面的结构和样式告诉 native 端，这样才能得到 native 级别的终极界面效果。
2. 事件绑定与触发，双向：native 端只负责界面渲染和非常薄的事件触发层，事件的逻辑处理都会放在 JavaScript，这样就具备了复杂数据处理和逻辑控制的动态性可能。JS 告诉 native 需要监听的交互行为，而当用户产生对应的交互行为时，native 端会把交互信息回传给 JS。
3. 对外的数据/信息请求与响应，双向：JS 引擎在处理特殊逻辑时，难免需要向服务器请求数据、或请求本地的系统信息和用户信息、或调用 native 的某个功能，这个时候也会通过 JS Bridge 进行请求，native 收到这些请求之后，也会在必要的情况下通过 JS Bridge 把信息回传给 JS 引擎。

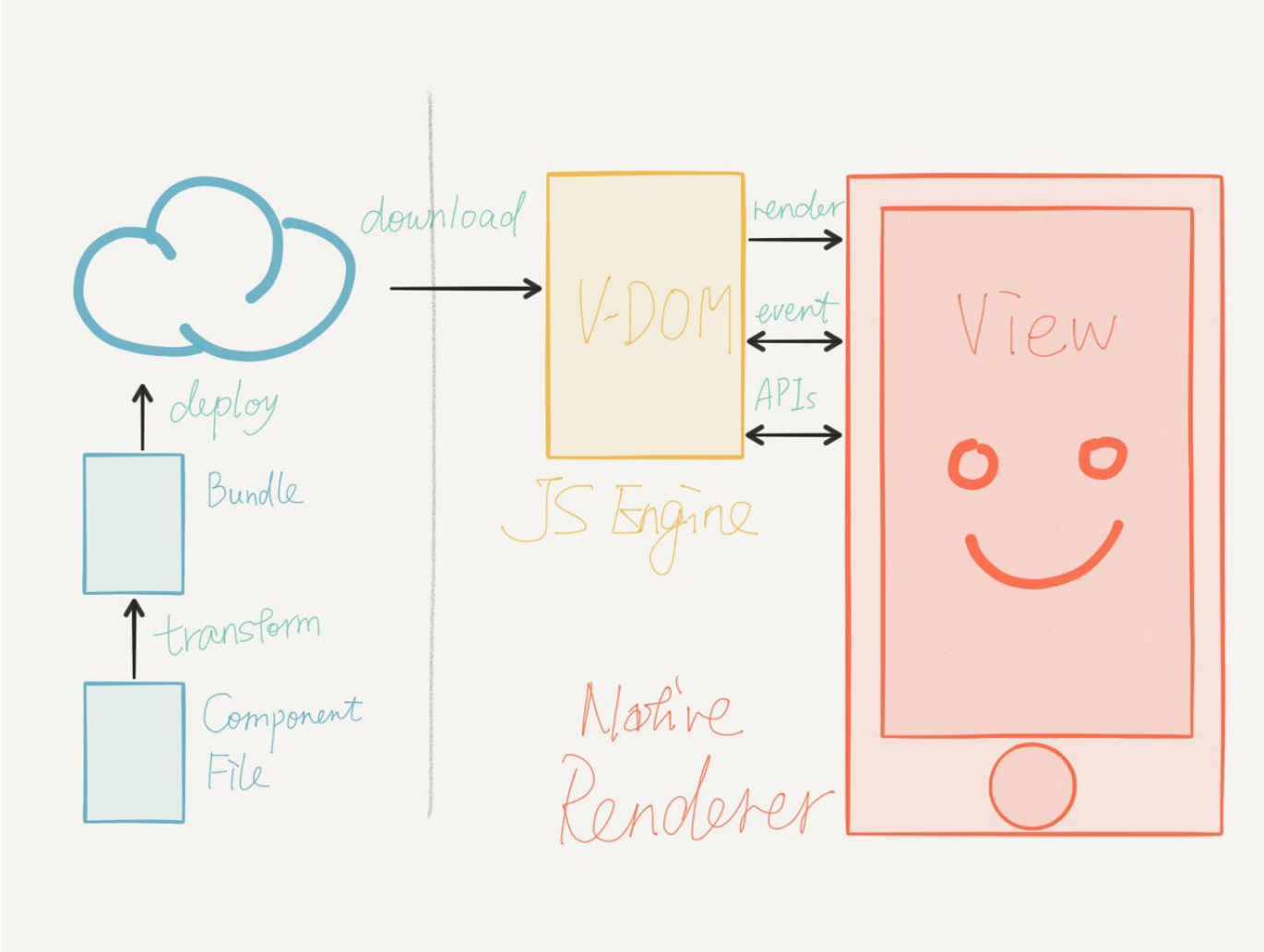
再加上外层对 Weex 实例的管理，整套机制就可以顺畅的工作起来了。

浏览器端渲染

同一份 JavaScript 程序包，可以同时通过客户端的 JS Bridge 渲染成为 native 界面，也可以通过浏览器渲染成为 web 界面。Weex的做法就是把 JS Bridge 背后的 native 处理逻辑同构成了 HTML5 版本。然后发布这样的一个页面。

综上所述

整个 Weex 的工作原理大致可以用一张图来表述：



一些实现细节

data-binding

Weex 对上层撰写的代码遵循了传统的 mustache 书写习惯。即：

```
<template>
  <text style="color: {{mainColor}}">{{title}}</text>
</template>
<script>
  module.exports = {
    mainColor: '#FF9900',
    title: 'Hello Weex!'
  }
</script>
```

Weex 的 transformer 对这样的语法的解析思路非常简单直接，即：

- 不含 mustache 语法的值，直接做为最终返回值
- 含 mustache 语法的值 (如 `{{xxx}}`)，转换为：`function () {return xxx}`

所以上面的模板最终会被转换成：

```
{
  type: 'text',
  style: {
    color: function () {
      return this.mainColor
    }
  },
  content: function () {
    return this.title
  }
}
```

在客户端的 JavaScript 引擎中，会进行这样的判断：

```
// parent, key, value, updater, data
if (typeof value === 'function') {
  updater = value
  parent._bindKey(data, null, key, updater)
}
else {
  data[key] = value
}
```

transformer

transformer 主要的工作就是对 HTML、CSS、JavaScript 代码进行解析和重组。这里用到了三个非常重要的库：

- HTML 解析工具：htmlparser
- CSS 解析工具：cssom
- JavaScript 解析工具：uglify-js

用 htmlparser 可以把 HTML 转换成 JSON。

```
var fragment
var handler = new htmlparser.DefaultHandler(function (err, data){
  fragment = data
})
var parser = new htmlparser.Parser(handler)
parser.parseComplete(content)
...
```

用 cssom 可以把 CSS 转换成对象供二次处理。

```
var css = cssom.parse(content)
var rules = css.cssRules

rules.forEach(function (rule) {
  rule.selectorText
  rule.style
  ...
})
```

用 uglify-js 可以把 JavaScript 代码进行细节解析处理。

```
//
new uglifyjs.TreeWalker(function(node, descend) {...})
...
//
uglifyjs.parse(code)
...
```

而且因为有了 transformer，可以把传统 mvvm 需要在客户端甚至 DOM
上完成的模板解析、数据绑定语法解析等工作提前处理完毕。所以免去了客户端运行时现解析模板源文件的负担。更酷的是，因为模板解析是不依赖真实 DOM 的，所以可以大大方方的把语法设计成 `<img src="{{xxx}}">` 而不必担心任何副作用。而高级的表达式、过滤器等数据绑定语法也都可以在 transformer 这一层提前处理好，这样在撰写体验持续增强的情况下，运行时并不会产生额外的负担——这都要归功于引入了这一层 transformer。

debugger tools

三个功能：

1. 调试 JavaScript 代码，任意设置断点 debug
2. 运行命令行代码 (Console) 对程序做实时的判断
3. 渲染结构的树形审查

实现原理：

1. 客户端设置一个开关，可以把 JS Bridge 对接到一个远程的 websocket 服务器，而不是对接到本地的 JavaScript 引擎。
2. 本地准备一个网页，其中运行了完整的 JavaScript 引擎的代码，并且也可以链接到一个远程的 websocket 服务器，这样客户端的 native 层和本地网页里的 JavaScript 引擎就串联起来了。
3. 原本通过 JS Bridge 的双向通信内容可以被 websocket 连接记录下来。
4. JavaScript 引擎里的所有代码都可以通过本地浏览器的开发者工具进行 debug 和 console 控制。
5. 开发一个简易的 Chrome Devtools Extension，可以得到 Weex 实例的界面结构，并以目录树的方式呈现出来。

这样，一个客户端开关，一个 websocket 服务器，一个本地的 JavaScript 引擎页面，一个开发者工具扩展，我们就实现了 Weex 的远程调试。