6.1 双线程模型

在前面第二章中，我们就有提到过小程序是基于双线程模型的，在这个模型中，小程序的逻辑层与渲染层分开在不同的线程运行，这跟传统的 Web 单线程模型有很大的不同，使得小程序架构上多了一些复杂度，也多了一些限制。至于为何选择基于双线程模型来搭建小程序，以及因此而产生的问题和解决方案，接下来我们将一一介绍。

6.1.1 技术选型

我们在对小程序的架构设计时的要求只有一个，就是要快，包括要渲染快、加载快等。当用户点开某个小程序时，我们期望体验到的是只有很短暂的加载界面，在一个过渡动画之后可以马上看到小程序的主界面。

我们首先需要确定用什么技术来渲染小程序界面，这是跟开发者的学习门槛息息相关的。

一般来说，渲染界面的技术有三种：

1. 用纯客户端原生技术来渲染
2. 用纯 Web 技术来渲染
3. 介于客户端原生技术与 Web 技术之间的，互相结合各自特点的技术（**下面统称 Hybrid 技术**）来渲染

由于小程序的宿主是微信，所以我们不太可能用纯客户端原生技术来编写小程序 。如果这么做，那小程序代码需要与微信代码一起编包，跟随微信发版本，这种方式跟开发节奏必然都是不对的。因此，我们需要像 Web 技术那样，有一份随时可更新的资源包放在云端，通过下载到本地，动态执行后即可渲染出界面。

但是，如果我们用纯 Web 技术来渲染小程序，在一些有复杂交互的页面上可能会面临一些性能问题，这是因为在 Web 技术中，UI 渲染跟 JavaScript 的脚本执行都在一个单线程中执行，这就容易导致一些逻辑任务抢占 UI 渲染的资源。

按照上面的讨论，使用纯客户端原生技术或纯 Web 技术都有各自的缺点，那如果使用两者结合起来的 Hybrid 技术来渲染小程序，能否优于各自独立渲染的技术方案呢？实际上，这种 Hybrid 技术在业界过去几年里演化过数种技术方案，典型的如早期的 PhoneGap[1]，还有近两年流行的 React Native[2]（下称 RN），还有像微信网页里的 JS-SDK[3] 这种轻量级的应用。

从渲染底层来看，PhoneGap 与微信 JS-SDK 是类似的，它们最终都还是使用浏览器内核来渲染界面。而 RN 则不同，虽然是用 Web 相关技术来编写，同样是利用了 JavaScript 解释执行的特性，但 RN 在渲染底层是用客户端原生渲染的。实际上，小程序最初选型时 RN 是候选之一，虽然说 RN 是结合了 React 框架的代码组成方式，但是我们完全可以剥离 React 框架这套写法，定义出更符合小程序特点的代码组成方式。不过，最终我们并没有选择这种类 RN 技术，原因有三：

1. RN 所支持的样式是 CSS 的子集，会满足不了 Web 开发者日渐增长的需求，而对 RN 的改造具有不小的成本和风险。
2. RN 现有能力下还存在的一些不稳定问题，比如性能、Bug 等。RN 是把渲染工作全都交由客户端原生渲染，实际上一些简单的界面元素使用 Web 技术渲染完全能胜任，并且非常稳定。
3. RN 存在一些不可预期的因素，比如近期就出现了许可协议问题。

最终，我们选择类似于微信 JSSDK 这样的 Hybrid 技术，即界面主要由成熟的 Web 技术渲染，辅之以大量的接口提供丰富的客户端原生能力。同时，每个小程序页面都是用不同的 WebView 去渲染，这样可以提供更好的交互体验，更贴近原

生体验，也避免了单个 WebView 的任务过于繁重。此外，界面渲染这一块我们定义了一套内置组件以统一体验，并且提供一些基础和通用的能力，进一步降低开发者的学习门槛。值得一提的是，内置组件有一部分较复杂组件是用客户端原生渲染的，以提供更好的性能，在后面的章节中将深入介绍。

6.1.2 管控与安全

基于 Web 技术来渲染小程序是存在一些不可控因素和安全风险的。这是因为 Web 技术是非常开放灵活的，我们可以利用 JavaScript 脚本随意地跳转网页或者改变界面上的任意内容。

我们原本定义了一套内置组件以提供统一的体验，用户进入小程序时，小程序代码包会被拉到本地（第七章将会详细介绍），这使得小程序可以离线浏览（只要小程序开发者把一些应用数据缓存到了本地），但要是开发者通过 JavaScript 把渲染小程序的 WebView 跳转到其他在线网页，这个体验就变得非常糟。

除此之外，我们也提供一种可以展示敏感数据的组件（这些数据只能被展示，开发者并不能拿到数据），若开发者可以通过 JavaScript 操作界面（DOM 树），从而直接获取这些敏感数据，那小程序毫无安全可言。

为了解决管控与安全问题，我们必须阻止开发者使用一些浏览器提供的，诸如跳转页面、操作 DOM、动态执行脚本的开放性接口。假设我们一个一个禁止，那势必会进入一个攻防战，这是因为 JavaScript 的灵活性以及浏览器接口的丰富性，我们很容易遗漏一些危险的接口，而且就算被我们找到所有危险的接口，也许在

下一次浏览器内核更新而新增了一个可能会在这套体系下产生漏洞的接口，这样还是无法完全避免。

因此，要彻底解决这个问题，我们必须提供一个沙箱环境来运行开发者的 JavaScript 代码。这个沙箱环境不能有任何浏览器相关接口，只提供纯 JavaScript 的解释执行环境，那么像 HTML5 中的 ServiceWorker、WebWorker 特性就符合这样的条件，这两者都是启用另一线程来执行 JavaScript。但是考虑到小程序是一个多 WebView 的架构，每一个小程序页面都是不同的 WebView 渲染后显示的，在这个架构下我们不好去用某个 WebView 中的 ServiceWorker 去管理所有的小程序页面。

得益于客户端系统有 JavaScript 的解释引擎（在 iOS 下是用内置的 JavaScriptCore 框架，在安卓则是用腾讯 x5 内核提供的 JsCore 环境），我们可以创建一个单独的线程去执行 JavaScript，在这个环境下执行的都是有关小程序业务逻辑的代码，也就是我们前面一直提到的逻辑层。而界面渲染相关的任务全都在 WebView 线程里执行，通过逻辑层代码去控制渲染哪些界面，那么这一层当然就是所谓的渲染层。这就是小程序双线程模型的由来。

6.1.3 天生的延时

既然小程序是基于双线程模型，那就意味着任何数据传递都是线程间的通信，也就是都会有一定的延时。这不像传统 Web 那样，当界面需要更新时，通过调用更新接口 UI 就会同步地渲染出来。在小程序架构里，这一切都会变成异步。

异步会使得各部分的运行时序变得复杂一些。比如在渲染首屏的时候，逻辑层与渲染层会同时开始初始化工作，但是渲染层需要有逻辑层的数据才能把界面渲染出来，如果渲染层初始化工作较快完成，就要等逻辑层的指令才能进行下一步工作。因此逻辑层与渲染层需要有一定的机制保证时序正确，这些工作在小程序框架里会处理好，开发者只需要理解生命周期，以及控制合适的时机更新 UI 即可。更多的运行流程细节会在第七章中详细介绍。

除了逻辑层与渲染层之间的通信有延时，各层与客户端原生交互同样是有延时的。以逻辑层为例，开发者的代码是跑在逻辑层这个线程之上，而客户端原生是跑在微信主线程（安卓上是线程）之上，所以注册给逻辑层有关客户端能力的接口，实际上也是跟微信主线程之间的通信，同样意味着有延时。这也是我们看到大部分提供的接口都是异步的原因。

在理解了小程序架构下很多天生的延时后，我们会更容易想到一些问题的解决方法。比如，我们在使用一块画布（canvas 组件）做图像处理并导出照片时，会习惯地在调用 draw 方法渲染后，立即调用 wx.canvasToTempFilePath 接口来导出图片，实际上很有可能调用的时刻画布还未完成渲染而使导出的图片不是期望的效果。

代码清单 6-1 使用 canvas 导出图片脚本

```
var ctx = wx.createCanvasContext('myCanvas');

ctx.fillRect(0, 0, 100, 100);

// ……

ctx.draw();
```

```
// 以下调用应该要在 ctx.draw 调用的回调函数里执行

wx.canvasToTempFilePath({

  canvasId: 'myCanvas',

  success(res) {

    console.log('canvasToTempFilePathresult: ' + res.tempFilePath);

  }

});
```