6.2 组件系统

小程序的视图是在 WebView 里渲染的，那搭建视图的方式自然就需要用到 HTML 语言。如果我们直接提供 HTML 的能力，那前面章节所介绍的为解决管控与安全而建立的双线程模型就成为摆设了。开发者可以利用 A 标签实现跳转到其它在线网页，也可以动态执行 JavaScript 等。除管控与安全外，还有一些的不足之处：

1 标签众多，增加理解成本；

1 接口底层，不利于快速开发；

1 能力有限，会限制小程序的表现形式。

因此，我们设计一套组件框架——Exparser。基于这个框架，内置了一套组件，以涵盖小程序的基础功能，便于开发者快速搭建出任何界面。同时也提供了自定义组件的能力，开发者可以自行扩展更多的组件，以实现代码复用。

6.2.1 Exparser 框架

Exparser 是微信小程序的组件组织框架，内置在小程序基础库中，为小程序的各种组件提供基础的支持。小程序内的所有组件，包括内置组件和自定义组件，都由 Exparser 组织管理。

Exparser 的组件模型与 WebComponents 标准中的 ShadowDOM 高度相似。Exparser 会维护整个页面的节点树相关信息，包括节点的属性、事件绑定等，相当于一个简化版的 Shadow DOM 实现。Exparser 的主要特点包括以下几点：

1. 基于 Shadow DOM 模型：模型上与 WebComponents 的 ShadowDOM 高度相似，但不依赖浏览器的原生支持，也没有其他依赖库；实现时，还针对性地增加了其他 API 以支持小程序组件编程。
2. 可在纯 JS 环境中运行：这意味着逻辑层也具有一定的组件树组织能力。
3. 高效轻量：性能表现好，在组件实例极多的环境下表现尤其优异，同时代码尺寸也较小。

小程序中，所有节点树相关的操作都依赖于 Exparser，包括 WXML 到页面最终节点树的构建、createSelectorQuery 调用和自定义组件特性等。

6.2.2 内置组件

我们基于 Exparser 框架，内置了一套组件，提供了视图容器类、表单类、导航类、媒体类、开放类等几十种组件。有了这么丰富的组件，再配合 WXSS，我们可以搭建出任何效果的界面。在功能层面上，也满足绝大部分需求。

一般而言，我们会把一个组件内置到小程序框架里的一个重要原则是：这个组件是基础的。换句话说，没有这个组件的话，在小程序架构里无法实现或者实现不好某类功能。比如像一些开放类组件，有 open-data 组件提供展示群名称、用户信息等微信体系下的隐私信息，有 button 组件里 open-type 属性所提供分享、跳转 App 等敏感操作的能力。还有比如像视图容器类组件 movable-view 这种因双线程模型导致手势识别不好实现的组件，这是因为手势识别需要高频率捕捉手指的触摸事件，而在双线程模型中，触摸事件从渲染层发出，派发到逻辑层，这中间是有一定的延时而导致视图跟随手指运动这类交互变得有些卡顿。

6.2.3 自定义组件

自定义组件是开发者可以自行扩充的组件。开发者可以将常用的节点树结构提取成自定义组件，实现代码复用。

1. ShadowTree 的概念

我们以下面的代码为例来阐述 Shadow Tree 的概念。

代码清单 6-2 页面节点树(Composed Tree)

```
<view>
  <input-with-label>
    <label>
      TEXT
    </label>
    <input />
  </input-with-label>
</view>
```

这里如果将 input-with-label 抽象成一个组件，那么可以将整个节点树拆分成两部分。

代码清单 6-3 组件节点树(Shadow Tree)

```
<label><slot/></label>
<input />
```

代码清单 6-4 调用组件的节点树

```
<view>
    <input-with-label>
        TEXT
    </input-with-label>
</view>
```

在 Exparser 的组件模型中，这两个节点树可以被拼接成上方的页面节点树。其中，组件的节点树称为“ShadowTree”，即组件内部的实现；最终拼接成的页面节点树被称为“Composed Tree”，即将页面所有组件节点树合成之后的树。在进行了这样的组件分离之后，整个页面节点树实质上被拆分成了若干个 ShadowTree（页面的 body 实质上也是一个组件，因而也是一个 ShadowTree）。

同时，各个组件也将具有各自独立的逻辑空间。每个组件都分别拥有自己的独立的数据、setData 调用，createSelectorQuery 也将运行在 Shadow Tree 的层面上。关于具体如何使用自定义组件特性，这里不再详细讨论，请参阅小程序开发文档。

2. 运行原理

在使用自定义组件的小程序页面中，Exparser 将接管所有的自定义组件注册与实例化。从外部接口上看，小程序基础库提供有 Page 和 Component 两个构造器。以 Component 为例，在小程序启动时，构造器会将开发者设置的 properties、data、methods 等定义段，写入 Exparser 的组件注册表中。这个组件在被其它组件引用时，就可以根据这些注册信息来创建自定义组件的实例。Page 构造器的大体运行流程与之相仿，只是参数形式不一样。这样每个页面就有一个与之对应的组件，称为“页面根组件”。

在初始化页面时，Exparser 会创建出页面根组件的一个实例，用到的其他组件也会响应创建组件实例（这是一个递归的过程）。组件创建的过程大致有以下几个要点：

1. 根据组件注册信息，从组件原型上创建出组件节点的 JS 对象，即组件的 this；
2. 将组件注册信息中的 data 复制一份，作为组件数据，即 this.data；
3. 将这份数据结合组件 WXML，据此创建出 Shadow Tree，由于 Shadow Tree 中可能引用有其他组件，因而这会递归触发其他组件创建过程；
4. 将 ShadowTree 拼接到 Composed Tree 上，并生成一些缓存数据用于优化组件更新性能；
5. 触发组件的 created 生命周期函数；
6. 如果不是页面根组件，需要根据组件节点上的属性定义，来设置组件的属性值；
7. 当组件实例被展示在页面上时，触发组件的 attached 生命周期函数，如果 Shadw Tree 中有其他组件，也逐个触发它们的生命周期函数。

3. 组件间通信

不同组件实例间的通信有 WXML 属性值传递、事件系统、selectComponent 和 relations 等方式。其中，WXML 属性值传递是从父组件向子组件的基本通信方式，而事件系统是从子组件向父组件的基本通信方式。

Exparser 的事件系统完全模仿 Shadow DOM 的事件系统。在通常的理解中，事件可以分为冒泡事件和非冒泡事件，但在 ShadowDOM 体系中，冒泡事件还可以划分为在 Shadow Tree 上冒泡的事件和在 Composed Tree 上冒泡的事件。如果在 Shadow Tree 上冒泡，则冒泡只会经过这个组件 Shadow Tree 上的节点，这样可以有效控制事件冒泡经过的范围。

代码清单 6-5 input-with-label 的 WXML

```
<label>
  <input />
  <slot />
</label>
```

代码清单 6-6 页面 WXML

```
<view>
```

```
  <input-with-label>
    <button />
  </input-with-label>
</view>
```

用上面的例子来说，当在 button 上触发一个事件时：

1 如果事件是非冒泡的，那只能在 button 上监听到事件；

1 如果事件是在 Shadow Tree 上冒泡的，那 button 、 input-with-label 、 view 可以依次监听到事件；

1 如果事件是在 Composed Tree 上冒泡的，那 button 、 slot 、label 、 input-with-label 、 view 可以依次监听到事件。

在自定义组件中使用 triggerEvent 触发事件时，可以指定事件的 bubbles、composed 和 capturePhase 属性，用于标注事件的冒泡性质。

代码清单 6-7 triggerEvent 事例

```
Component({
  methods: {
    helloEvent: function() {
      this.triggerEvent('hello', {}, {
        bubbles: true,      // 这是一个冒泡事件
        composed: true,     // 这个事件在 Composed Tree 上冒泡
        capturePhase: false // 这个事件没有捕获阶段
      })
```

```
    }
  }
})
```

小程序基础库自身也会通过这套事件系统提供一些用户事件，如 tap、touchstart 和 form 组件的 submit 等。其中，tap 等用户触摸引发的事件是在 ComposedTree 上的冒泡事件，其他事件大多是非冒泡事件。