2.4 JavaScript 脚本

小程序的主要开发语言是 JavaScript ，开发者使用 JavaScript 来开发业务逻辑以及调用小程序的 API 来完成业务需求。

2.4.1 ECMAScript

在大部分开发者看来，ECMAScript 和 JavaScript 表达的是同一种含义，但是严格的说，两者的意义是不同的。ECMAScript 是一种由 Ecma 国际通过 ECMA-262 标准化的脚本程序设计语言， JavaScript 是 ECMAScript 的一种实现。理解 JavaScript 是 ECMAScript 一种实现后，可以帮助开发者理解小程序中的 JavaScript 同浏览器中的 JavaScript 以及 NodeJS 中的 JavaScript 是不相同的。

ECMA-262 规定了 ECMAScript 语言的几个重要组成部分：

1. 语法
2. 类型
3. 语句
4. 关键字
5. 操作符
6. 对象

浏览器中 JavaScript 构成如下图：

图 2-15 浏览器中的 JavaScript

浏览器中的 JavaScript 是由 ECMAScript 和 BOM（浏览器对象模型）以及 DOM（文档对象模型）组成的，Web 前端开发者会很熟悉这两个对象模型，它使得开

发者可以去操作浏览器的一些表现，比如修改 URL、修改页面呈现、记录数据等等。

NodeJS 中 JavaScript 构成如下图：

图 2-16 NodeJS 中的 JavaScript

NodeJS 中的 JavaScript 是由 ECMAScript 和 NPM 以及 Native 模块组成，NodeJS 的开发者会非常熟悉 NPM 的包管理系统，通过各种拓展包来快速的实现一些功能，同时通过使用一些原生的模块例如 FS、HTTP、OS 等等来拥有一些语言本身所不具有的能力。

那么，同开发者所熟悉的这两个环境是不同的，小程序中 JavaScript 构成如图 2-17 所示。

图 2-17 小程序中的 JavaScript

小程序中的 JavaScript 是由 ECMAScript 以及小程序框架和小程序 API 来实现的。同浏览器中的 JavaScript 相比没有 BOM 以及 DOM 对象，所以类似 JQuery、Zepto 这种浏览器类库是无法在小程序中运行起来的，同样的缺少 Native 模块和 NPM 包管理的机制，小程序中无法加载原生库，也无法直接使用大部分的 NPM 包。

2.4.2 小程序的执行环境

明白了小程序中的 JavaScript 同浏览器以及 NodeJS 有所不同后，开发者还需要注意到另外一个问题，不同的平台的小程序的脚本执行环境也是有所区别的。

小程序目前可以运行在三大平台：

1. iOS 平台，包括 iOS9、iOS10、iOS11
2. Android 平台
3. 小程序 IDE

这种区别主要是体现三大平台实现的 ECMAScript 的标准有所不同。截止到当前一共有七个版本的 ECMAScript 标准，目前开发者大部分使用的是 ECMAScript 5 和 ECMAScript 6 的标准，但是在小程序中，iOS9 和 iOS10 所使用的运行环境并没有完全的兼容到 ECMAScript 6 标准，一些 ECMAScript 6 中规定的语法和关键字是没有的或者同标准是有所不同的，例如：

1. 箭头函数
2. let const
3. 模板字符串
4. …

所以一些开发者会发现有些代码在旧的手机操作系统上出现一些语法错误。为了帮助开发者解决这类问题，小程序 IDE 提供语法转码工具帮助开发者，将 ECMAScript 6 代码转为 ECMAScript 5 代码，从而在所有的环境都能得到很好的执行。

开发者需要在项目设置中，勾选 ES6 转 ES5 开启此功能。

图 2-18 勾选 ES6 转 ES5

2.4.3 模块化

浏览器中，所有 JavaScript 是在运行在同一个作用域下的，定义的参数或者方法可以被后续加载的脚本访问或者改写。同浏览器不同，小程序中可以将任何一

个 JavaScript 文件作为一个模块，通过 module.exports 或者 exports 对外暴露接口。

请看是一个简单模块示例，B.js 引用模块 A，并使用 A 暴露的 multiplyBy2 方法完成一个变量乘以 2 的操作。

代码清单 2-26 模块示例

```
// moduleA.js
module.exports = function( value ){
  return value * 2;
}
```

代码清单 2-27 引用模块 A

```
// B.js

// 在 B.js 中引用模块 A
var multiplyBy2 = require('./moduleA')
var result = multiplyBy2(4)
```

代码清单 2-28 在需要使用这些模块的文件中，使用 require(path) 将公共代码引入

```
var common = require('common.js')
Page({
  helloMINA: function() {
    common.sayHello('MINA')
  },
  goodbyeMINA: function() {
    common.sayGoodbye('MINA')
  }
})
```

2.4.4 脚本的执行顺序

浏览器中，脚本严格按照加载的顺序执行，如代码 2-29 所示。

代码清单 2-29 浏览器中的脚本

```
<html>
<head>
  <!-- a.js
  console.log('a.js')
   -->
  <script src ="a.js">&lt;/script&gt;
  &lt;script&gt;
    console.log('inline script')
  &lt;/script&gt;

  <!-- b.js
  console.log('b.js')
   -->
  <script src ="b.js">&lt;/script&gt;
</head>
</html>
```

以上代码的输出是：

```
a.js

inline script

b.js
```

而在小程序中的脚本执行顺序有所不同。小程序的执行的入口文件是 app.js 。

并且会根据其中 require 的模块顺序决定文件的运行顺序，代码 2-30 是一个 app.js 示例。

代码清单 2-30 app.js

```
/* a.js
console.log('a.js')
*/
var a = require('./a.js')
console.log('app.js')

/* b.js
console.log('b.js')
*/
```

```
var b = require('./b.js')
```

以上代码的输出顺序是：

```
a.js

app.js

b.js
```

当 app.js 执行结束后，小程序会按照开发者在 app.json 中定义的 pages 的顺序，逐一执行。如代码 2-31 所示。

代码清单 2-31 app.json 文件

```
{
  "pages": [
    "pages/index/index",
    "pages/log/log",
    "pages/result/result"
  ],
  "window": {}
}
```

代码清单 2-32 app.js 文件

```
// app.js
console.log('app.js')
```

代码清单 2-33 pages/index/index.js 文件

```
// pages/index/index
console.log('pages/index/index')
```

代码清单 2-34 page/log/log.js 文件

```
// pages/log/log
console.log('pages/log/log')
```

代码清单 2-35 page/result/result.js 文件

```
// pages/result/result
console.log('pages/result/result')
```

以上文件执行后输出的结果如下：

```
app.js

pages/index/index

pages/log/log

pages/result/result
```

2.4.5 作用域

同浏览器中运行的脚本文件有所不同，小程序的脚本的作用域同 NodeJS 更为相似。

在文件中声明的变量和函数只在该文件中有效，不同的文件中可以声明相同名字的变量和函数，不会互相影响，如代码 2-36、代码 2-37 所示。

代码清单 2-36 在脚本 a.js 中定义局部变量

```
// a.js
// 定义局部变量
var localValue = 'a'
```

代码清单 2-37 在脚本 b.js 中无法访问 a.js 定义的变量

```
// b.js
// 定义局部变量
console.log(localValue) // 触发一个错误 b.js 中无法访问 a.js 中定义的变量
```

当需要使用全局变量的时，通过使用全局函数 getApp() 获取全局的实例，并设置相关属性值，来达到设置全局变量的目的，如代码 2-38、代码 2-39 所示。

代码清单 2-38 在脚本 a.js 中设置全局变量

```
// a.js
// 获取全局变量
var global = getApp()
global.globalValue = 'globalValue'
```

代码清单 2-39 在脚本 b.js 中访问 a.js 定义的全局变量

```
// b.js
// 访问全局变量
var global = getApp()
console.log(global.globalValue) // 输出 globalValue
```

需要注意的是，上述示例只有在 a.js 比 b.js 先执行才有效，当需要保证全局的数据可以在任何文件中安全的被使用到，那么可以在 App() 中进行设置，如代码 2-40、代码 2-41、代码 2-42 所示。

代码清单 2-40 定义全局变量

```
// app.js
App({
  globalData: 1
})
```

代码清单 2-41 获取以及修改 global 变量的方法

```
// a.js
// 局部变量
var localValue = 'a'

// 获取 global 变量
var app = getApp()

// 修改 global 变量
app.globalData++  // 执行后 globalData 数值为 2
```

代码清单 2-42 获取 global 变量

```
// b.js
// 定义另外的局部变量，并不会影响 a.js 中文件变量
var localValue = 'b'

// 如果先执行了 a.js 这里的输出应该是 2
console.log(getApp().globalData)
```