微信小程序框架文档

书栈(BookStack.CN)

目　录

致谢

当前文档 《微信小程序框架文档》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-06-26。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：http://www.bookstack.cn/books/mp-framework

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

介绍

框架

小程序开发框架的目标是通过尽可能简单、高效的方式让开发者可以在微信中开发具有原生 APP 体验的服务。

框架提供了自己的视图层描述语言 WXML 和 WXSS，以及基于 JavaScript 的逻辑层框架，并在视图层与逻辑层间提供了数据传输和事件系统，可以让开发者可以方便的聚焦于数据与逻辑上。

响应的数据绑定

框架的核心是一个响应的数据绑定系统。

整个系统分为两块视图层（View）和逻辑层（App Service）

框架可以让数据与视图非常简单地保持同步。当做数据修改的时候，只需要在逻辑层修改数据，视图层就会做相应的更新。

通过这个简单的例子来看：

在开发者工具中预览效果

```
<!-- This is our View -->
<view> Hello {{name}}! </view>
<button bindtap="changeName"> Click me! </button>
```

```
// This is our App Service.
// This is our data.
var helloData = {
  name: 'WeChat'
}

// Register a Page.
Page({
  data: helloData,
  changeName: function(e) {
    // sent data change to view
    this.setData({
      name: 'MINA'
    })
  }
})
```

- 开发者通过框架将逻辑层数据中的 name 与视图层的 name 进行了绑定，所以在页面一打开的时候会显示 Hello WeChat!
- 当点击按钮的时候，视图层会发送 changeName 的事件给逻辑层，逻辑层找到对应的事件处理函数

- 逻辑层执行了 setData 的操作，将 name 从 WeChat 变为 MINA，因为该数据和视图层已经绑定了，从而视图层会自动改变为 Hello MINA! 。

页面管理

框架 管理了整个小程序的页面路由，可以做到页面间的无缝切换，并给以页面完整的生命周期。开发者需要做的只是将页面的数据，方法，生命周期函数注册进 框架 中，其他的一切复杂的操作都交由 框架 处理。

基础组件

框架 提供了一套基础的组件，这些组件自带微信风格的样式以及特殊的逻辑，开发者可以通过组合基础组件，创建出强大的微信小程序 。

丰富的 API

框架 提供丰富的微信原生 API，可以方便的调起微信提供的能力，如获取用户信息，本地存储，支付功能等。

原文:

> *https://developers.weixin.qq.com/miniprogram/dev/framework/MINA.html*

目录结构

文件结构

小程序包含一个描述整体程序的 app 和多个描述各自页面的 page。

一个小程序主体部分由三个文件组成，必须放在项目的根目录，如下：

文件	必填	作用
app.js	是	小程序逻辑
app.json	是	小程序公共设置
app.wxss	否	小程序公共样式表

一个小程序页面由四个文件组成，分别是：

文件类型	必填	作用
js	是	页面逻辑
wxml	是	页面结构
wxss	否	页面样式表
json	否	页面配置

注意：为了方便开发者减少配置项，描述页面的四个文件必须具有相同的路径与文件名。

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/structure.html*

配置

配置

`app.json` 文件用来对微信小程序进行全局配置，决定页面文件的路径、窗口表现、设置网络超时时间、设置多 tab 等。

以下是一个包含了所有配置选项的 `app.json` ：

```
{
  "pages": [
    "pages/index/index",
    "pages/logs/index"
  ],
  "window": {
    "navigationBarTitleText": "Demo"
  },
  "tabBar": {
    "list": [{
      "pagePath": "pages/index/index",
      "text": "首页"
    }, {
      "pagePath": "pages/logs/logs",
      "text": "日志"
    }]
  },
  "networkTimeout": {
    "request": 10000,
    "downloadFile": 10000
  },
  "debug": true
}
```

app.json 配置项列表

属性	类型	必填	描述
pages	String Array	是	设置页面路径
window	Object	否	设置默认页面的窗口表现
tabBar	Object	否	设置底部 tab 的表现
networkTimeout	Object	否	设置网络超时时间
debug	Boolean	否	设置是否开启 debug 模式

pages

接受一个数组，每一项都是字符串，来指定小程序由哪些页面组成。每一项代表对应页面的【路径+文件名】信息，数组的第一项代表小程序的初始页面。小程序中新增/减少页面，都需要对 **pages** 数组进行修改。

文件名不需要写文件后缀，因为框架会自动去寻找路径下 `.json` , `.js` , `.wxml` , `.wxss` 四个文件进行整合。

如开发目录为:

> *pages/pages/index/index.wxmlpages/index/index.jspages/index/index.wxsspages/logs/logs.wxmlpages/logs/logs.jsapp.jsapp.jsonapp.wxss*

则需要在 app.json 中写

```
{
  "pages":[
    "pages/index/index",
    "pages/logs/logs"
  ]
}
```

window

用于设置小程序的状态栏、导航条、标题、窗口背景色。

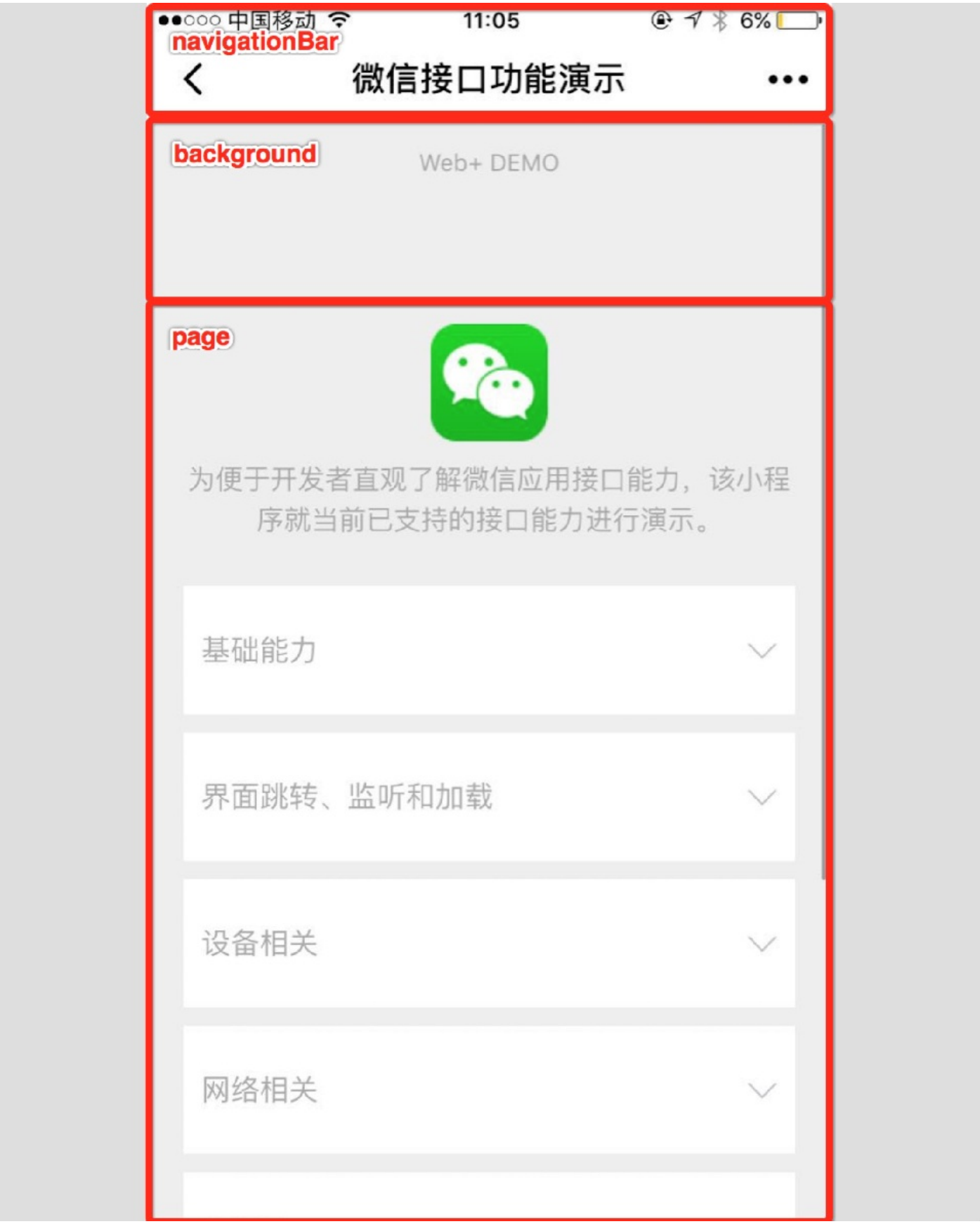
属性	类型	默认值	描述	最低版本
navigationBarBackgroundColor	HexColor	#000000	导航栏背景颜色，如"#000000"	
navigationBarTextStyle	String	white	导航栏标题颜色，仅支持 black/white	
navigationBarTitleText	String		导航栏标题文字内容	
navigationStyle	String	default	导航栏样式，仅支持 default/custom。custom 模式可自定义导航栏，只保留右上角胶囊状的按钮	微信版本 6.6.0
backgroundColor	HexColor	#ffffff	窗口的背景色	
backgroundTextStyle	String	dark	下拉 loading 的样式，仅支持 dark/light	
backgroundColorTop	String	#ffffff	顶部窗口的背景色，仅 iOS 支持	微信版本 6.5.16
backgroundColorBottom	String	#ffffff	底部窗口的背景色，仅 iOS 支持	微信版本 6.5.16
enablePullDownRefresh	Boolean	false	是否开启下拉刷新，详见页面相关事件处理函数	
onReachBottomDistance	Number	50	页面上拉触底事件触发时距页面底部距离，单位为px	

注: **HexColor**（十六进制颜色值），如**"#ff00ff"**

注: **navigationStyle** 只在 **app.json** 中生效。开启 **custom** 后，低版本客户端需要做好兼容。开发者工具基础库版本切到 **1.7.0**（不代表最低版本，只供调试用） 可方便切到旧视觉

如 app.json ：

```
{
  "window":{
    "navigationBarBackgroundColor": "#ffffff",
    "navigationBarTextStyle": "black",
    "navigationBarTitleText": "微信接口功能演示",
    "backgroundColor": "#eeeeee",
    "backgroundTextStyle": "light"
  }
}
```



tabBar

如果小程序是一个多 tab 应用（客户端窗口的底部或顶部有 tab 栏可以切换页面），可以通过 tabBar 配置项指定 tab 栏的表现，以及 tab 切换时显示的对应页面。

Tip:

- 当设置 position 为 top 时，将不会显示 icon
- tabBar 中的 list 是一个数组，只能配置最少**2**个、最多**5**个 **tab**，tab 按数组的顺序排序。

属性说明:

属性	类型	必填	默认值	描述
color	HexColor	是		tab 上的文字默认颜色
selectedColor	HexColor	是		tab 上的文字选中时的颜色
backgroundColor	HexColor	是		tab 的背景色
borderStyle	String	否	black	tabbar上边框的颜色， 仅支持 black/white
list	Array	是		tab 的列表，详见 list 属性说明，最少2个、最多5个 tab
position	String	否	bottom	可选值 bottom、top

其中 list 接受一个数组，数组中的每个项都是一个对象，其属性值如下:

属性	类型	必填	说明
pagePath	String	是	页面路径，必须在 pages 中先定义
text	String	是	tab 上按钮文字
iconPath	String	否	图片路径，icon 大小限制为40kb，建议尺寸为 81px * 81px，当 postion 为 top 时，此参数无效，不支持网络图片
selectedIconPath	String	否	选中时的图片路径，icon 大小限制为40kb，建议尺寸为 81px * 81px ，当 postion 为 top 时，此参数无效

networkTimeout

可以设置各种网络请求的超时时间。

属性说明:

属性	类型	必填	说明
request	Number	否	wx.request的超时时间，单位毫秒，默认为: 60000
connectSocket	Number	否	wx.connectSocket的超时时间，单位毫秒，默认为: 60000
uploadFile	Number	否	wx.uploadFile的超时时间，单位毫秒，默认为: 60000
downloadFile	Number	否	wx.downloadFile的超时时间，单位毫秒，默认为: 60000

debug

可以在开发者工具中开启 debug 模式，在开发者工具的控制台面板，调试信息以 info 的形式给出，其信息有 `Page的注册` ， `页面路由` ， `数据更新` ， `事件触发` 。 可以帮助开发者快速定位一些常见的问题。

page.json

每一个小程序页面也可以使用 `.json` 文件来对本页面的窗口表现进行配置。 页面的配置比 `app.json` 全局配置简单得多，只是设置 app.json 中的 window 配置项的内容，页面中配置项会覆盖 app.json 的 window 中相同的配置项。

页面的 `.json` 只能设置 `window` 相关的配置项，以决定本页面的窗口表现，所以无需写 `window` 这个键，如:

属性	类型	默认值	描述
navigationBarBackgroundColor	HexColor	#000000	导航栏背景颜色，如"#000000"
navigationBarTextStyle	String	white	导航栏标题颜色，仅支持 black/white
navigationBarTitleText	String		导航栏标题文字内容
backgroundColor	HexColor	#ffffff	窗口的背景色
backgroundTextStyle	String	dark	下拉 loading 的样式，仅支持 dark/light
enablePullDownRefresh	Boolean	false	是否开启下拉刷新，详见页面相关事件处理函数。
disableScroll	Boolean	false	设置为 true 则页面整体不能上下滚动；只在 page.json 中有效，无法在 app.json 中设置该项
onReachBottomDistance	Number	50	页面上拉触底事件触发时距页面底部距离，单位为px

```
{
  "navigationBarBackgroundColor": "#ffffff",
  "navigationBarTextStyle": "black",
  "navigationBarTitleText": "微信接口功能演示",
  "backgroundColor": "#eeeeee",
  "backgroundTextStyle": "light"
}
```

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/config.html*

逻辑层

逻辑层(App Service)

小程序开发框架的逻辑层由 JavaScript 编写。

逻辑层将数据进行处理后发送给视图层，同时接受视图层的事件反馈。

在 JavaScript 的基础上，我们做了一些修改，以方便地开发小程序。

- 增加 App 和 Page 方法，进行程序和页面的注册。
- 增加 getApp 和 getCurrentPages 方法，分别用来获取 App 实例和当前页面栈。
- 提供丰富的 API，如微信用户数据，扫一扫，支付等微信特有能力。
- 每个页面有独立的作用域，并提供模块化能力。
- 由于框架并非运行在浏览器中，所以 JavaScript 在 web 中一些能力都无法使用，如 document，window 等。
- 开发者写的所有代码最终将会打包成一份 JavaScript，并在小程序启动的时候运行，直到小程序销毁。类似 ServiceWorker，所以逻辑层也称之为 App Service。

原文:

> *https://developers.weixin.qq.com/miniprogram/dev/framework/app-service/*

注册程序

App

App()

`App()` 函数用来注册一个小程序。接受一个 object 参数，其指定小程序的生命周期函数等。

object参数说明：

属性	类型	描述	触发时机
onLaunch	Function	生命周期函数--监听小程序初始化	当小程序初始化完成时，会触发 onLaunch（全局只触发一次）
onShow	Function	生命周期函数--监听小程序显示	当小程序启动，或从后台进入前台显示，会触发 onShow
onHide	Function	生命周期函数--监听小程序隐藏	当小程序从前台进入后台，会触发 onHide
onError	Function	错误监听函数	当小程序发生脚本错误，或者 api 调用失败时，会触发 onError 并带上错误信息
onPageNotFound	Function	页面不存在监听函数	当小程序出现要打开的页面不存在的情况，会带上页面信息回调该函数，详见下文
其他	Any		开发者可以添加任意的函数或数据到 Object 参数中，用 `this` 可以访问

前台、后台定义： 当用户点击左上角关闭，或者按了设备 Home 键离开微信，小程序并没有直接销毁，而是进入了后台；当再次进入微信或再次打开小程序，又会从后台进入前台。需要注意的是：只有当小程序进入后台一定时间，或者系统资源占用过高，才会被真正的销毁。

关闭小程序（基础库版本1.1.0开始支持）： 当用户从扫一扫、转发等入口(场景值为1007, 1008, 1011, 1025)进入小程序，且没有置顶小程序的情况下退出，小程序会被销毁。

小程序运行机制在基础库版本 **1.4.0** 有所改变： 上一条关闭逻辑在新版本已不适用。详情

示例代码：

```
App({
  onLaunch: function(options) {
    // Do something initial when launch.
  },
  onShow: function(options) {
      // Do something when show.
  },
  onHide: function() {
      // Do something when hide.
  },
  onError: function(msg) {
```

```
    console.log(msg)
  },
  globalData: 'I am global data'
})
```

onLaunch, onShow 参数

字段	类型	说明
path	String	打开小程序的路径
query	Object	打开小程序的query
scene	Number	打开小程序的场景值
shareTicket	String	shareTicket，详见 获取更多转发信息
referrerInfo	Object	当场景为由从另一个小程序或公众号或App打开时，返回此字段
referrerInfo.appId	String	来源小程序或公众号或App的 appId，详见下方说明
referrerInfo.extraData	Object	来源小程序传过来的数据，scene=1037或1038时支持

场景值 详见。

以下场景支持返回 referrerInfo.appId:

场景值	场景	appId 信息含义
1020	公众号 profile 页相关小程序列表	返回来源公众号 appId
1035	公众号自定义菜单	返回来源公众号 appId
1036	App 分享消息卡片	返回来源应用 appId
1037	小程序打开小程序	返回来源小程序 appId
1038	从另一个小程序返回	返回来源小程序 appId
1043	公众号模板消息	返回来源公众号 appId

onPageNotFound

> 基础库 *1.9.90* 开始支持，低版本需做兼容处理

当要打开的页面并不存在时，会回调这个监听器，并带上以下信息:

字段	类型	说明
path	String	不存在页面的路径
query	Object	打开不存在页面的 query
isEntryPage	Boolean	是否本次启动的首个页面（例如从分享等入口进来，首个页面是开发者配置的分享页面）

开发者可以在 `onPageNotFound` 回调中进行重定向处理，但必须在回调中同步处理，异步处理（例如 setTimeout 异步执行）无效。

示例代码:

```
App({
  onPageNotFound(res) {
    wx.redirectTo({
      url: 'pages/...'
    }) // 如果是 tabbar 页面，请使用 wx.switchTab
  }
})
```

注意：

- 如果开发者没有添加 onPageNotFound 监听，当跳转页面不存在时，将推入微信客户端原生的页面不存在提示页面
- 如果 onPageNotFound 回调中又重定向到另一个不存在的页面，将推入微信客户端原生的页面不存在提示页面，并且不在回调 onPageNotFound

getApp()

全局的 `getApp()` 函数可以用来获取到小程序实例。

```
// other.js
var appInstance = getApp()
console.log(appInstance.globalData) // I am global data
```

注意：

- App() 必须在 app.js 中注册，且不能注册多个。
- 不要在定义于 App() 内的函数中调用 getApp() ，使用 this 就可以拿到 app 实例。
- 不要在 onLaunch 的时候调用 getCurrentPages()，此时 page 还没有生成。
- 通过 getApp() 获取实例之后，不要私自调用生命周期函数。

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/app-service/app.html*

场景值

场景值

> 基础库 *1.1.0* 开始支持，低版本需做兼容处理

当前支持的场景值有：

场景值ID	说明
1001	发现栏小程序主入口
1005	顶部搜索框的搜索结果页
1006	发现栏小程序主入口搜索框的搜索结果页
1007	单人聊天会话中的小程序消息卡片
1008	群聊会话中的小程序消息卡片
1011	扫描二维码
1012	长按图片识别二维码
1013	手机相册选取二维码
1014	小程序模版消息
1017	前往体验版的入口页
1019	微信钱包
1020	公众号 profile 页相关小程序列表
1022	聊天顶部置顶小程序入口
1023	安卓系统桌面图标
1024	小程序 profile 页
1025	扫描一维码
1026	附近小程序列表
1027	顶部搜索框搜索结果页“使用过的小程序”列表
1028	我的卡包
1029	卡券详情页
1030	自动化测试下打开小程序
1031	长按图片识别一维码
1032	手机相册选取一维码
1034	微信支付完成页
1035	公众号自定义菜单
1036	App 分享消息卡片
1037	小程序打开小程序
1038	从另一个小程序返回

1039	摇电视
1042	添加好友搜索框的搜索结果页
1043	公众号模板消息
1044	带 shareTicket 的小程序消息卡片（详情）
1045	朋友圈广告
1047	扫描小程序码
1048	长按图片识别小程序码
1049	手机相册选取小程序码
1052	卡券的适用门店列表
1053	搜一搜的结果页
1054	顶部搜索框小程序快捷入口
1056	音乐播放器菜单
1057	钱包中的银行卡详情页
1058	公众号文章
1059	体验版小程序绑定邀请页
1064	微信连Wi-Fi状态栏
1067	公众号文章广告
1068	附近小程序列表广告
1069	移动应用
1071	钱包中的银行卡列表页
1072	二维码收款页面
1073	客服消息列表下发的小程序消息卡片
1074	公众号会话下发的小程序消息卡片
1077	摇周边
1078	连Wi-Fi成功页
1079	微信游戏中心
1081	客服消息下发的文字链
1082	公众号会话下发的文字链
1089	微信聊天主界面下拉
1090	长按小程序右上角菜单唤出最近使用历史
1091	公众号文章商品卡片
1092	城市服务入口
1095	小程序广告组件
1096	聊天记录
1097	微信支付签约页
1102	服务号 profile 页服务预览

可以在 App 的 onlaunch 和 onshow 中获取上述场景值，部分场景值下还可以获取来源应用、公众号或小程序的 appId。详见

Tip: 由于Android系统限制，目前还无法获取到按 Home 键退出到桌面，然后从桌面再次进小程序的场景值，对于这种情况，会保留上一次的场景值。

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/app-service/scene.html*

注册页面

Page

`Page()` 函数用来注册一个页面。接受一个 object 参数，其指定页面的初始数据、生命周期函数、事件处理函数等。

object 参数说明：

属性	类型	描述
data	Object	页面的初始数据
onLoad	Function	生命周期函数—监听页面加载
onReady	Function	生命周期函数—监听页面初次渲染完成
onShow	Function	生命周期函数—监听页面显示
onHide	Function	生命周期函数—监听页面隐藏
onUnload	Function	生命周期函数—监听页面卸载
onPullDownRefresh	Function	页面相关事件处理函数—监听用户下拉动作
onReachBottom	Function	页面上拉触底事件的处理函数
onShareAppMessage	Function	用户点击右上角转发
onPageScroll	Function	页面滚动触发事件的处理函数
onTabItemTap	Function	当前是 tab 页时，点击 tab 时触发
其他	Any	开发者可以添加任意的函数或数据到 object 参数中，在页面的函数中用 `this` 可以访问

object 内容在页面加载时会进行一次深拷贝，需考虑数据大小对页面加载的开销

示例代码：

```
//index.js
Page({
  data: {
    text: "This is page data."
  },
  onLoad: function(options) {
    // Do some initialize when page load.
  },
  onReady: function() {
    // Do something when page ready.
  },
  onShow: function() {
    // Do something when page show.
  },
  onHide: function() {
    // Do something when page hide.
```

```
  },
  onUnload: function() {
    // Do something when page close.
  },
  onPullDownRefresh: function() {
    // Do something when pull down.
  },
  onReachBottom: function() {
    // Do something when page reach bottom.
  },
  onShareAppMessage: function () {
   // return custom share data when user share.
  },
  onPageScroll: function() {
    // Do something when page scroll
  },
  onTabItemTap(item) {
    console.log(item.index)
    console.log(item.pagePath)
    console.log(item.text)
  },
  // Event handler.
  viewTap: function() {
    this.setData({
      text: 'Set some data for updating view.'
    }, function() {
      // this is setData callback
    })
  },
  customData: {
    hi: 'MINA'
  }
})
```

初始化数据

初始化数据将作为页面的第一次渲染。data 将会以 JSON 的形式由逻辑层传至渲染层，所以其数据必须是可以转成 JSON 的格式：字符串，数字，布尔值，对象，数组。

渲染层可以通过 WXML 对数据进行绑定。

示例代码：

在开发者工具中预览效果

```
<view>{{text}}</view>
<view>{{array[0].msg}}</view>
```

```
Page({
  data: {
    text: 'init data',
```

```
    array: [{msg: '1'}, {msg: '2'}]
  }
})
```

生命周期函数

- onLoad: 页面加载
    - 一个页面只会调用一次，可以在 onLoad 中获取打开当前页面所调用的 query 参数。
- onShow: 页面显示
    - 每次打开页面都会调用一次。
- onReady: 页面初次渲染完成
    - 一个页面只会调用一次，代表页面已经准备妥当，可以和视图层进行交互。
    - 对界面的设置如wx.setNavigationBarTitle请在onReady之后设置。详见生命周期
- onHide: 页面隐藏
    - 当navigateTo或底部tab切换时调用。
- onUnload: 页面卸载
    - 当redirectTo或navigateBack的时候调用。
      生命周期的调用以及页面的路由方式详见

onLoad参数

类型	说明
Object	其他页面打开当前页面所调用的 query 参数

页面相关事件处理函数

- onPullDownRefresh: 下拉刷新
    - 监听用户下拉刷新事件。
    - 需要在app.json的window选项中或页面配置中开启enablePullDownRefresh。
    - 当处理完数据刷新后，wx.stopPullDownRefresh可以停止当前页面的下拉刷新。
- onReachBottom: 上拉触底
    - 监听用户上拉触底事件。
    - 可以在app.json的window选项中或页面配置中设置触发距离onReachBottomDistance。
    - 在触发距离内滑动期间，本事件只会被触发一次。
- onPageScroll: 页面滚动
    - 监听用户滑动页面事件。

    - 参数为 Object，包含以下字段:
      |字段|类型|说明
      |—-
      |scrollTop|Number|页面在垂直方向已滚动的距离（单位px）

- onShareAppMessage: 用户转发
    - 只有定义了此事件处理函数，右上角菜单才会显示“转发”按钮
    - 用户点击转发按钮的时候会调用
    - 此事件需要 return 一个 Object，用于自定义转发内容
      自定义转发字段

字段	说明	默认值
title	转发标题	当前小程序名称
path	转发路径	当前页面 path ，必须是以 / 开头的完整路径

示例代码

在开发者工具中预览效果

```
Page({
  onShareAppMessage: function () {
    return {
      title: '自定义转发标题',
      path: '/page/user?id=123'
    }
  }
})
```

事件处理函数

除了初始化数据和生命周期函数，Page 中还可以定义一些特殊的函数：事件处理函数。在渲染层可以在组件中加入事件绑定，当达到触发事件时，就会执行 Page 中定义的事件处理函数。

示例代码:

在开发者工具中预览效果

```
<view bindtap="viewTap"> click me </view>
```

```
Page({
  viewTap: function() {
    console.log('view tap')
  }
})
```

Page.prototype.route

> 基础库 *1.2.0* 开始支持，低版本需做兼容处理

`route` 字段可以获取到当前页面的路径。

Page.prototype.setData()

`setData` 函数用于将数据从逻辑层发送到视图层（异步），同时改变对应的 `this.data` 的值（同步）。

setData() 参数格式

字段	类型	必填	描述	最低版本
data	Object	是	这次要改变的数据	
callback	Function	否	回调函数	1.5.0

object 以 key，value 的形式表示将 this.data 中的 key 对应的值改变成 value。callback 是一个回调函数，在这次setData对界面渲染完毕后调用。

其中 key 可以非常灵活，以数据路径的形式给出，如 `array[2].message`，`a.b.c.d`，并且不需要在 this.data 中预先定义。

注意：

- 直接修改 **this.data** 而不调用 **this.setData** 是无法改变页面的状态的，还会造成数据不一致。
- 仅支持设置可 JSON 化的数据。
- 单次设置的数据不能超过1024kB，请尽量避免一次设置过多的数据。
- 请不要把 data 中任何一项的 value 设为 undefined ，否则这一项将不被设置并可能遗留一些潜在问题。

示例代码：
在开发者工具中预览效果

```
<!--index.wxml-->
<view>{{text}}</view>
<button bindtap="changeText"> Change normal data </button>
<view>{{num}}</view>
<button bindtap="changeNum"> Change normal num </button>
<view>{{array[0].text}}</view>
<button bindtap="changeItemInArray"> Change Array data </button>
<view>{{object.text}}</view>
<button bindtap="changeItemInObject"> Change Object data </button>
<view>{{newField.text}}</view>
<button bindtap="addNewField"> Add new data </button>
```

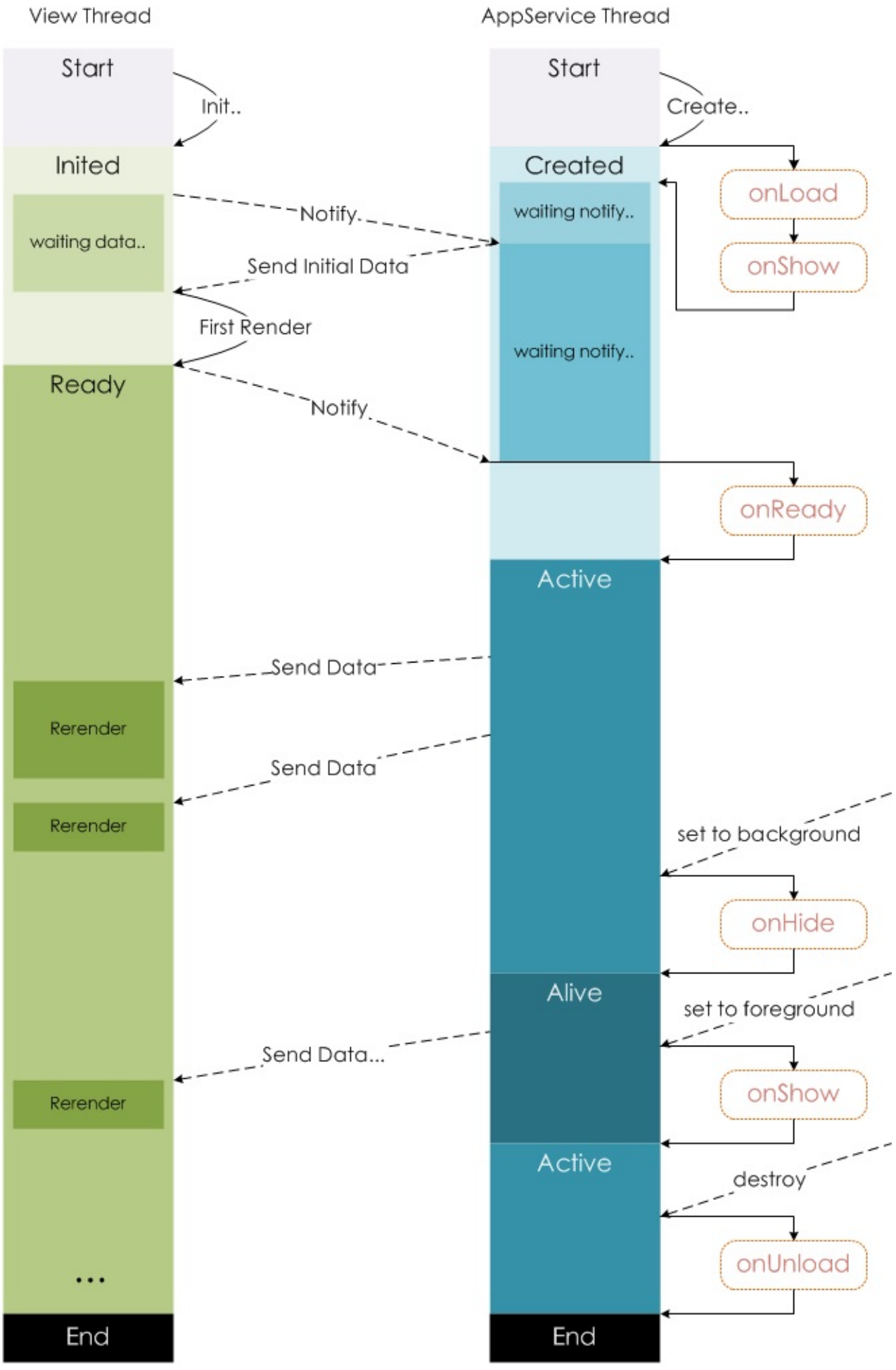
```
//index.js
Page({
  data: {
    text: 'init data',
    num: 0,
```

```
    array: [{text: 'init data'}],
    object: {
      text: 'init data'
    }
  },
  changeText: function() {
    // this.data.text = 'changed data'  // bad, it can not work
    this.setData({
      text: 'changed data'
    })
  },
  changeNum: function() {
    this.data.num = 1
    this.setData({
      num: this.data.num
    })
  },
  changeItemInArray: function() {
    // you can use this way to modify a danamic data path
    this.setData({
      'array[0].text':'changed data'
    })
  },
  changeItemInObject: function(){
    this.setData({
      'object.text': 'changed data'
    });
  },
  addNewField: function() {
    this.setData({
      'newField.text': 'new data'
    })
  }
})
```

以下内容你不需要立马完全弄明白，不过以后它会有帮助。

生命周期

下图说明了 Page 实例的生命周期。


View Thread
AppService Thread
Start
Init..
Inited
waiting data..
Notify
Send Initial Data
First Render
Ready
Notify
Rerender
Send Data
Rerender
Send Data
Rerender
Send Data...
...
End
Start
Create..
Created
waiting notify..
onLoad
onShow
waiting notify..
onReady
Active
set to background
onHide
Alive
set to foreground
onShow
Active
destroy
onUnload
End

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/app-service/page.html*

路由

页面路由

在小程序中所有页面的路由全部由框架进行管理。

页面栈

框架以栈的形式维护了当前的所有页面。当发生路由切换的时候，页面栈的表现如下：

路由方式	页面栈表现
初始化	新页面入栈
打开新页面	新页面入栈
页面重定向	当前页面出栈，新页面入栈
页面返回	页面不断出栈，直到目标返回页
Tab 切换	页面全部出栈，只留下新的 Tab 页面
重加载	页面全部出栈，只留下新的页面

getCurrentPages()

`getCurrentPages()` 函数用于获取当前页面栈的实例，以数组形式按栈的顺序给出，第一个元素为首页，最后一个元素为当前页面。

Tip: 不要尝试修改页面栈，会导致路由以及页面状态错误。

路由方式

对于路由的触发方式以及页面生命周期函数如下：

路由方式	触发时机	路由前页面	路由后页面
初始化	小程序打开的第一个页面		onLoad, onShow
打开新页面	调用 API `wx.navigateTo` 或使用组件 `<navigator open-type="navigateTo"/>`	onHide	onLoad, onShow
页面重定向	调用 API `wx.redirectTo` 或使用组件 `<navigator open-type="redirectTo"/>`	onUnload	onLoad, onShow
页面返回	调用 API `wx.navigateBack` 或使用组件 `<navigator open-type="navigateBack">` 或用户按左上角返回按钮	onUnload	onShow
Tab 切换	调用 API `wx.switchTab` 或使用组件 `<navigator open-type="switchTab"/>` 或用户切换 Tab		各种情况请参考下表

重启动	调用 API `wx.reLaunch` 或使用组件 `<navigator open-type="reLaunch"/>`	onUnload	onLoad, onShow

Tab 切换对应的生命周期（以 A、B 页面为 Tabbar 页面，C 是从 A 页面打开的页面，D 页面是从 C 页面打开的页面为例）：

当前页面	路由后页面	触发的生命周期（按顺序）
A	A	Nothing happend
A	B	A.onHide(), B.onLoad(), B.onShow()
A	B（再次打开）	A.onHide(), B.onShow()
C	A	C.onUnload(), A.onShow()
C	B	C.onUnload(), B.onLoad(), B.onShow()
D	B	D.onUnload(), C.onUnload(), B.onLoad(), B.onShow()
D（从转发进入）	A	D.onUnload(), A.onLoad(), A.onShow()
D（从转发进入）	B	D.onUnload(), B.onLoad(), B.onShow()

Tips:

- navigateTo, redirectTo 只能打开非 tabBar 页面。
- switchTab 只能打开 tabBar 页面。
- reLaunch 可以打开任意页面。
- 页面底部的 tabBar 由页面决定，即只要是定义为 tabBar 的页面，底部都有 tabBar。
- 调用页面路由带的参数可以在目标页面的onLoad中获取。

原文:

> *https://developers.weixin.qq.com/miniprogram/dev/framework/app-service/route.html*

模块化

文件作用域

在 JavaScript 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。

通过全局函数 `getApp()` 可以获取全局的应用实例，如果需要全局的数据可以在 `App()` 中设置，如：

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

```
// a.js
// The localValue can only be used in file a.js.
var localValue = 'a'
// Get the app instance.
var app = getApp()
// Get the global data and change it.
app.globalData++
```

```
// b.js
// You can redefine localValue in file b.js, without interference with the localValue in a.js.
var localValue = 'b'
// If a.js it run before b.js, now the globalData shoule be 2.
console.log(getApp().globalData)
```

模块化

可以将一些公共的代码抽离成为一个单独的 js 文件，作为一个模块。模块只有通过 `module.exports` 或者 `exports` 才能对外暴露接口。

需要注意的是：

- exports 是 module.exports 的一个引用，因此在模块里边随意更改 exports 的指向会造成未知的错误。所以更推荐开发者采用 module.exports 来暴露模块接口，除非你已经清晰知道这两者的关系。
- 小程序目前不支持直接引入 node_modules , 开发者需要使用到 node_modules 时候建议拷贝出相关的代码到小程序的目录中。

```
// common.js
function sayHello(name) {
console.log(`Hello - {name} !`)
}
function sayGoodbye(name) {
console.log(`Goodbye - {name} !`)
```

```
}
module.exports.sayHello = sayHello
exports.sayGoodbye = sayGoodbye
```

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')
  }
})
```

Tips

- tip: require 暂时不支持绝对路径

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/app-service/module.html*

API

API

小程序开发框架提供丰富的微信原生 API，可以方便的调起微信提供的能力，如获取用户信息，本地存储，支付功能等。

详细介绍请参考 API 文档

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/app-service/api.html*

视图层

视图层

框架的视图层由 WXML 与 WXSS 编写，由组件来进行展示。

将逻辑层的数据反应成视图，同时将视图层的事件发送给逻辑层。

WXML(WeiXin Markup language) 用于描述页面的结构。

WXS(WeiXin Script) 是小程序的一套脚本语言，结合 `WXML` ，可以构建出页面的结构。

WXSS(WeiXin Style Sheet) 用于描述页面的样式。

组件(Component)是视图的基本组成单元。

原文:

> *https://developers.weixin.qq.com/miniprogram/dev/framework/view/*

WXML

WXML

WXML（WeiXin Markup Language）是框架设计的一套标签语言，结合基础组件、事件系统，可以构建出页面的结构。

用以下一些简单的例子来看看 WXML 具有什么能力：

数据绑定

```
<!--wxml-->
<view> {{message}} </view>
```

```
// page.js
Page({
  data: {
    message: 'Hello MINA!'
  }
})
```

列表渲染

```
<!--wxml-->
<view wx:for="{{array}}"> {{item}} </view>
```

```
// page.js
Page({
  data: {
    array: [1, 2, 3, 4, 5]
  }
})
```

条件渲染

```
<!--wxml-->
<view wx:if="{{view == 'WEBVIEW'}}"> WEBVIEW </view>
<view wx:elif="{{view == 'APP'}}"> APP </view>
<view wx:else="{{view == 'MINA'}}"> MINA </view>
```

```
// page.js
Page({
```

```
  data: {
    view: 'MINA'
  }
})
```

模板

```
<!--wxml-->
<template name="staffName">
  <view>
    FirstName: {{firstName}}, LastName: {{lastName}}
  </view>
</template>

<template is="staffName" data="{{...staffA}}"></template>
<template is="staffName" data="{{...staffB}}"></template>
<template is="staffName" data="{{...staffC}}"></template>
```

```
// page.js
Page({
  data: {
    staffA: {firstName: 'Hulk', lastName: 'Hu'},
    staffB: {firstName: 'Shang', lastName: 'You'},
    staffC: {firstName: 'Gideon', lastName: 'Lin'}
  }
})
```

事件

```
<view bindtap="add"> {{count}} </view>
```

```
Page({
  data: {
    count: 1
  },
  add: function(e) {
    this.setData({
      count: this.data.count + 1
    })
  }
})
```

具体的能力以及使用方式在以下章节查看：

数据绑定、列表渲染、条件渲染、模板、事件、引用

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/view/wxml/*

数据绑定

数据绑定

WXML 中的动态数据均来自对应 Page 的 data。

简单绑定

数据绑定使用 Mustache 语法（双大括号）将变量包起来，可以作用于：

内容

```
<view> {{ message }} </view>
```

```
Page({
  data: {
    message: 'Hello MINA!'
  }
})
```

组件属性(需要在双引号之内)

```
<view id="item-{{id}}"> </view>
```

```
Page({
  data: {
    id: 0
  }
})
```

控制属性(需要在双引号之内)

```
<view wx:if="{{condition}}"> </view>
```

```
Page({
  data: {
    condition: true
  }
})
```

关键字(需要在双引号之内)

`true` : boolean 类型的 true，代表真值。

`false` : boolean 类型的 false，代表假值。

```
<checkbox checked="{{false}}"> </checkbox>
```

特别注意：不要直接写 `checked="false"` ，其计算结果是一个字符串，转成 ***boolean*** 类型后代表真值。

运算

可以在 `{{}}` 内进行简单的运算，支持的有如下几种方式：

三元运算

```
<view hidden="{{flag ? true : false}}"> Hidden </view>
```

算数运算

```
<view> {{a + b}} + {{c}} + d </view>
```

```
Page({
  data: {
    a: 1,
    b: 2,
    c: 3
  }
})
```

view中的内容为 `3 + 3 + d` 。

逻辑判断

```
<view wx:if="{{length > 5}}"> </view>
```

字符串运算

```
<view>{{"hello" + name}}</view>
```

```
Page({
  data:{
    name: 'MINA'
```

```
  }
})
```

数据路径运算

```
<view>{{object.key}} {{array[0]}}</view>
```

```
Page({
  data: {
    object: {
      key: 'Hello '
    },
    array: ['MINA']
  }
})
```

组合

也可以在 Mustache 内直接进行组合，构成新的对象或者数组。

数组

```
<view wx:for="{{[zero, 1, 2, 3, 4]}}"> {{item}} </view>
```

```
Page({
  data: {
    zero: 0
  }
})
```

最终组合成数组 `[0, 1, 2, 3, 4]` 。

对象

```
<template is="objectCombine" data="{{for: a, bar: b}}"></template>
```

```
Page({
  data: {
    a: 1,
    b: 2
  }
})
```

最终组合成的对象是 `{for: 1, bar: 2}`

也可以用扩展运算符 `...` 来将一个对象展开

```
<template is="objectCombine" data="{{...obj1, ...obj2, e: 5}}"></template>
```

```
Page({
  data: {
    obj1: {
      a: 1,
      b: 2
    },
    obj2: {
      c: 3,
      d: 4
    }
  }
})
```

最终组合成的对象是 `{a: 1, b: 2, c: 3, d: 4, e: 5}` 。

如果对象的 key 和 value 相同，也可以间接地表达。

```
<template is="objectCombine" data="{{foo, bar}}"></template>
```

```
Page({
  data: {
    foo: 'my-foo',
    bar: 'my-bar'
  }
})
```

最终组合成的对象是 `{foo: 'my-foo', bar:'my-bar'}` 。

注意：上述方式可以随意组合，但是如有存在变量名相同的情况，后边的会覆盖前面，如：

```
<template is="objectCombine" data="{{...obj1, ...obj2, a, c: 6}}"></template>
```

```
Page({
  data: {
    obj1: {
      a: 1,
      b: 2
    },
    obj2: {
      b: 3,
      c: 4
    },
    a: 5
  }
```

```
13. })
```

最终组合成的对象是 `{a: 5, b: 3, c: 6}` 。

注意： 花括号和引号之间如果有空格，将最终被解析成为字符串

```
1. <view wx:for="{{[1,2,3]}} ">
2.   {{item}}
3. </view>
```

等同于

```
1. <view wx:for="{{[1,2,3] + ' '}}">
2.   {{item}}
3. </view>
```

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/view/wxml/data.html*

列表渲染

列表渲染

wx:for

在组件上使用 `wx:for` 控制属性绑定一个数组，即可使用数组中各项的数据重复渲染该组件。

默认数组的当前项的下标变量名默认为 `index` ，数组当前项的变量名默认为 `item`

```
<view wx:for="{{array}}">
  {{index}}: {{item.message}}
</view>
```

```
Page({
  data: {
    array: [{
      message: 'foo',
    }, {
      message: 'bar'
    }]
  }
})
```

使用 `wx:for-item` 可以指定数组当前元素的变量名，

使用 `wx:for-index` 可以指定数组当前下标的变量名：

```
<view wx:for="{{array}}" wx:for-index="idx" wx:for-item="itemName">
  {{idx}}: {{itemName.message}}
</view>
```

`wx:for` 也可以嵌套，下边是一个九九乘法表

```
<view wx:for="{{[1, 2, 3, 4, 5, 6, 7, 8, 9]}}" wx:for-item="i">
  <view wx:for="{{[1, 2, 3, 4, 5, 6, 7, 8, 9]}}" wx:for-item="j">
    <view wx:if="{{i <= j}}">
      {{i}} * {{j}} = {{i * j}}
    </view>
  </view>
</view>
```

block wx:for

类似 `block wx:if` ，也可以将 `wx:for` 用在 `<block/>` 标签上，以渲染一个包含多节点的结构块。例如：

```
<block wx:for="{{[1, 2, 3]}}">
  <view> {{index}}: </view>
  <view> {{item}} </view>
</block>
```

wx:key

如果列表中项目的位置会动态改变或者有新的项目添加到列表中，并且希望列表中的项目保持自己的特征和状态（如 `<input/>` 中的输入内容， `<switch/>` 的选中状态），需要使用 `wx:key` 来指定列表中项目的唯一的标识符。

`wx:key` 的值以两种形式提供

- 字符串，代表在 for 循环的 array 中 item 的某个 property，该 property 的值需要是列表中唯一的字符串或数字，且不能动态改变。
- 保留关键字 *this 代表在 for 循环中的 item 本身，这种表示需要 item 本身是一个唯一的字符串或者数字，如：

当数据改变触发渲染层重新渲染的时候，会校正带有 key 的组件，框架会确保他们被重新排序，而不是重新创建，以确保使组件保持自身的状态，并且提高列表渲染时的效率。
如不提供 `wx:key` ，会报一个 `warning` ， 如果明确知道该列表是静态，或者不必关注其顺序，可以选择忽略。

示例代码：

在开发者工具中预览效果

```
<switch wx:for="{{objectArray}}" wx:key="unique" style="display: block;"> {{item.id}} </switch>
<button bindtap="switch"> Switch </button>
<button bindtap="addToFront"> Add to the front </button>

<switch wx:for="{{numberArray}}" wx:key="*this" style="display: block;"> {{item}} </switch>
<button bindtap="addNumberToFront"> Add to the front </button>
```

```
Page({
  data: {
    objectArray: [
      {id: 5, unique: 'unique_5'},
      {id: 4, unique: 'unique_4'},
      {id: 3, unique: 'unique_3'},
      {id: 2, unique: 'unique_2'},
      {id: 1, unique: 'unique_1'},
      {id: 0, unique: 'unique_0'},
    ],
    numberArray: [1, 2, 3, 4]
  },
  switch: function(e) {
    const length = this.data.objectArray.length
```

```
    for (let i = 0; i < length; ++i) {
      const x = Math.floor(Math.random() * length)
      const y = Math.floor(Math.random() * length)
      const temp = this.data.objectArray[x]
      this.data.objectArray[x] = this.data.objectArray[y]
      this.data.objectArray[y] = temp
    }
    this.setData({
      objectArray: this.data.objectArray
    })
  },
  addToFront: function(e) {
    const length = this.data.objectArray.length
    this.data.objectArray = [{id: length, unique: 'unique_' + length}].concat(this.data.objectArray)
    this.setData({
      objectArray: this.data.objectArray
    })
  },
  addNumberToFront: function(e){
    this.data.numberArray = [ this.data.numberArray.length + 1 ].concat(this.data.numberArray)
    this.setData({
      numberArray: this.data.numberArray
    })
  }
})
```

注意：

当 `wx:for` 的值为字符串时，会将字符串解析成字符串数组

```
<view wx:for="array">
  {{item}}
</view>
```

等同于

```
<view wx:for="{{['a','r','r','a','y']}}">
  {{item}}
</view>
```

注意： 花括号和引号之间如果有空格，将最终被解析成为字符串

```
<view wx:for="{{[1,2,3]}} ">
  {{item}}
</view>
```

等同于

```
<view wx:for="{{[1,2,3] + ' '}}" >
  {{item}}
```

```
3. </view>
```

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/view/wxml/list.html*

条件渲染

条件渲染

wx:if

在框架中，使用 `wx:if="{{condition}}"` 来判断是否需要渲染该代码块：

```
<view wx:if="{{condition}}"> True </view>
```

也可以用 `wx:elif` 和 `wx:else` 来添加一个 else 块：

```
<view wx:if="{{length > 5}}"> 1 </view>
<view wx:elif="{{length > 2}}"> 2 </view>
<view wx:else> 3 </view>
```

block wx:if

因为 `wx:if` 是一个控制属性，需要将它添加到一个标签上。如果要一次性判断多个组件标签，可以使用一个 `<block/>` 标签将多个组件包装起来，并在上边使用 `wx:if` 控制属性。

```
<block wx:if="{{true}}">
  <view> view1 </view>
  <view> view2 </view>
</block>
```

注意： `<block/>` 并不是一个组件，它仅仅是一个包装元素，不会在页面中做任何渲染，只接受控制属性。

wx:if vs hidden

因为 `wx:if` 之中的模板也可能包含数据绑定，所有当 `wx:if` 的条件值切换时，框架有一个局部渲染的过程，因为它会确保条件块在切换时销毁或重新渲染。

同时 `wx:if` 也是惰性的，如果在初始渲染条件为 `false` ，框架什么也不做，在条件第一次变成真的时候才开始局部渲染。

相比之下， `hidden` 就简单的多，组件始终会被渲染，只是简单的控制显示与隐藏。

一般来说， `wx:if` 有更高的切换消耗而 `hidden` 有更高的初始渲染消耗。因此，如果需要频繁切换的情景下，用 `hidden` 更好，如果在运行时条件不大可能改变则 `wx:if` 较好。

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/view/wxml/conditional.html*

模板

模板

WXML提供模板（template），可以在模板中定义代码片段，然后在不同的地方调用。

定义模板

使用 name 属性，作为模板的名字。然后在 `<template/>` 内定义代码片段，如：

```
<!--
  index: int
  msg: string
  time: string
-->
<template name="msgItem">
  <view>
    <text> {{index}}: {{msg}} </text>
    <text> Time: {{time}} </text>
  </view>
</template>
```

使用模板

使用 is 属性，声明需要的使用的模板，然后将模板所需要的 data 传入，如：

```
<template is="msgItem" data="{{...item}}"/>
```

```
Page({
  data: {
    item: {
      index: 0,
      msg: 'this is a template',
      time: '2016-09-15'
    }
  }
})
```

is 属性可以使用 Mustache 语法，来动态决定具体需要渲染哪个模板：

```
<template name="odd">
  <view> odd </view>
</template>
<template name="even">
  <view> even </view>
```

```
</template>

<block wx:for="{{[1, 2, 3, 4, 5]}}">
    <template is="{{item % 2 == 0 ? 'even' : 'odd'}}"/>
</block>
```

模板的作用域

模板拥有自己的作用域，只能使用 `data` 传入的数据以及模版定义文件中定义的 `<wxs />` 模块。

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/view/wxml/template.html*

事件

事件

什么是事件

- 事件是视图层到逻辑层的通讯方式。
- 事件可以将用户的行为反馈到逻辑层进行处理。
- 事件可以绑定在组件上，当达到触发事件，就会执行逻辑层中对应的事件处理函数。
- 事件对象可以携带额外信息，如 id, dataset, touches。

事件的使用方式

- 在组件中绑定一个事件处理函数。
  如 `bindtap` ，当用户点击该组件的时候会在该页面对应的Page中找到相应的事件处理函数。

```
1. <view id="tapTest" data-hi="WeChat" bindtap="tapName"> Click me! </view>
```

- 在相应的Page定义中写上相应的事件处理函数，参数是event。

```
1. Page({
2. tapName: function(event) {
3.   console.log(event)
4. }
5. })
```

- 可以看到log出来的信息大致如下:

```
{
"type":"tap",
"timeStamp":895,
"target": {
"id": "tapTest",
"dataset": {
"hi":"WeChat"
}
},
"currentTarget": {
"id": "tapTest",
"dataset": {
"hi":"WeChat"
}
```

```
},
"detail": {
"x":53,
"y":14
},
"touches":[{
"identifier":0,
"pageX":53,
"pageY":14,
"clientX":53,
"clientY":14
}],
"changedTouches":[{
"identifier":0,
"pageX":53,
"pageY":14,
"clientX":53,
"clientY":14
}]
}
```

事件详解

事件分类

事件分为冒泡事件和非冒泡事件：

- 冒泡事件：当一个组件上的事件被触发后，该事件会向父节点传递。
- 非冒泡事件：当一个组件上的事件被触发后，该事件不会向父节点传递。

WXML的冒泡事件列表：

类型	触发条件	最低版本
touchstart	手指触摸动作开始	
touchmove	手指触摸后移动	
touchcancel	手指触摸动作被打断，如来电提醒，弹窗	
touchend	手指触摸动作结束	
tap	手指触摸后马上离开	
longpress	手指触摸后，超过350ms再离开，如果指定了事件回调函数并触发了这个事件，tap事件将不被触发	1.5.0
longtap	手指触摸后，超过350ms再离开（推荐使用longpress事件代替）	
transitionend	会在 WXSS transition 或 wx.createAnimation 动画结束后触发	

animationstart	会在一个 WXSS animation 动画开始时触发	
animationiteration	会在一个 WXSS animation 一次迭代结束时触发	
animationend	会在一个 WXSS animation 动画完成时触发	
touchforcechange	在支持 3D Touch 的 iPhone 设备，重按时会触发	1.9.90

注：除上表之外的其他组件自定义事件如无特殊声明都是非冒泡事件，如 `<form/>` 的 `submit` 事件， `<input/>` 的 `input` 事件， `<scroll-view/>` 的 `scroll` 事件，(详见各个组件)

事件绑定和冒泡

事件绑定的写法同组件的属性，以 key、value 的形式。

- key 以bind或catch开头，然后跟上事件的类型，如bindtap、catchtouchstart。自基础库版本 1.5.0 起，bind和catch后可以紧跟一个冒号，其含义不变，如bind:tap、、catch:touchstart。
- value 是一个字符串，需要在对应的 Page 中定义同名的函数。不然当触发事件的时候会报错。

`bind` 事件绑定不会阻止冒泡事件向上冒泡， `catch` 事件绑定可以阻止冒泡事件向上冒泡。

如在下边这个例子中，点击 inner view 会先后调用 `handleTap3` 和 `handleTap2` (因为tap事件会冒泡到 middle view，而 middle view 阻止了 tap 事件冒泡，不再向父节点传递)，点击 middle view 会触发 `handleTap2` ，点击 outer view 会触发 `handleTap1` 。

```
<view id="outer" bindtap="handleTap1">
  outer view
  <view id="middle" catchtap="handleTap2">
    middle view
    <view id="inner" bindtap="handleTap3">
      inner view
    </view>
  </view>
</view>
```

事件的捕获阶段

自基础库版本 1.5.0 起，触摸类事件支持捕获阶段。捕获阶段位于冒泡阶段之前，且在捕获阶段中，事件到达节点的顺序与冒泡阶段恰好相反。需要在捕获阶段监听事件时，可以采用 `capture-bind` 、 `capture-catch` 关键字，后者将中断捕获阶段和取消冒泡阶段。

在下面的代码中，点击 inner view 会先后调用 `handleTap2` 、 `handleTap4` 、 `handleTap3` 、 `handleTap1` 。

```
<view id="outer" bind:touchstart="handleTap1" capture-bind:touchstart="handleTap2">
  outer view
  <view id="inner" bind:touchstart="handleTap3" capture-bind:touchstart="handleTap4">
    inner view
  </view>
</view>
```

如果将上面代码中的第一个 `capture-bind` 改为 `capture-catch` ，将只触发 `handleTap2` 。

```
<view id="outer" bind:touchstart="handleTap1" capture-catch:touchstart="handleTap2">
  outer view
  <view id="inner" bind:touchstart="handleTap3" capture-bind:touchstart="handleTap4">
    inner view
  </view>
</view>
```

事件对象

如无特殊说明，当组件触发事件时，逻辑层绑定该事件的处理函数会收到一个事件对象。

BaseEvent 基础事件对象属性列表：

属性	类型	说明
type	String	事件类型
timeStamp	Integer	事件生成时的时间戳
target	Object	触发事件的组件的一些属性值集合
currentTarget	Object	当前组件的一些属性值集合

CustomEvent 自定义事件对象属性列表（继承 **BaseEvent**）：

属性	类型	说明
detail	Object	额外的信息

TouchEvent 触摸事件对象属性列表（继承 **BaseEvent**）：

属性	类型	说明
touches	Array	触摸事件，当前停留在屏幕中的触摸点信息的数组
changedTouches	Array	触摸事件，当前变化的触摸点信息的数组

特殊事件：`<canvas/>` 中的触摸事件不可冒泡，所以没有 **currentTarget**。

type

代表事件的类型。

timeStamp

页面打开到触发事件所经过的毫秒数。

target

触发事件的源组件。

属性	类型	说明

id	String	事件源组件的id
tagName	String	当前组件的类型
dataset	Object	事件源组件上由 `data-` 开头的自定义属性组成的集合

currentTarget

事件绑定的当前组件。

属性	类型	说明
id	String	当前组件的id
tagName	String	当前组件的类型
dataset	Object	当前组件上由 `data-` 开头的自定义属性组成的集合

说明： **target** 和 **currentTarget** 可以参考上例中，点击 **inner view** 时， `handleTap3` 收到的事件对象 **target** 和 **currentTarget** 都是 **inner**，而 `handleTap2` 收到的事件对象 **target** 就是 **inner**，**currentTarget** 就是 **middle**。

dataset

在组件中可以定义数据，这些数据将会通过事件传递给 SERVICE。书写方式：以 `data-` 开头，多个单词由连字符 `-` 链接，不能有大写(大写会自动转成小写)如 `data-element-type` ，最终在 `event.currentTarget.dataset` 中会将连字符转成驼峰 `elementType` 。

示例：

```
<view data-alpha-beta="1" data-alphaBeta="2" bindtap="bindViewTap"> DataSet Test </view>
```

```
Page({
  bindViewTap:function(event){
    event.currentTarget.dataset.alphaBeta === 1 // - 会转为驼峰写法
    event.currentTarget.dataset.alphabeta === 2 // 大写会转为小写
  }
})
```

touches

touches 是一个数组，每个元素为一个 Touch 对象（canvas 触摸事件中携带的 touches 是 CanvasTouch 数组）。表示当前停留在屏幕上的触摸点。

Touch 对象

属性	类型	说明
identifier	Number	触摸点的标识符
pageX, pageY	Number	距离文档左上角的距离，文档的左上角为原点 ，横向为X轴，纵向为Y轴

clientX, clientY	Number	距离页面可显示区域（屏幕除去导航条）左上角距离，横向为X轴，纵向为Y轴

CanvasTouch 对象

属性	类型	说明	特殊说明
identifier	Number	触摸点的标识符	
x, y	Number	距离 Canvas 左上角的距离，Canvas 的左上角为原点 ，横向为X轴，纵向为Y轴	

changedTouches

changedTouches 数据格式同 touches。表示有变化的触摸点，如从无变有（touchstart），位置变化（touchmove），从有变无（touchend、touchcancel）。

detail

自定义事件所携带的数据，如表单组件的提交事件会携带用户的输入，媒体的错误事件会携带错误信息，详见组件定义中各个事件的定义。

点击事件的 `detail` 带有的 x, y 同 pageX, pageY 代表距离文档左上角的距离。

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/view/wxml/event.html*

引用

引用

WXML 提供两种文件引用方式 `import` 和 `include` 。

import

`import` 可以在该文件中使用目标文件定义的 `template` ，如:

在 item.wxml 中定义了一个叫 `item` 的 `template` :

```
<!-- item.wxml -->
<template name="item">
  <text>{{text}}</text>
</template>
```

在 index.wxml 中引用了 item.wxml，就可以使用 `item` 模板:

```
<import src="item.wxml"/>
<template is="item" data="{{text: 'forbar'}}"/>
```

import 的作用域

import 有作用域的概念，即只会 import 目标文件中定义的 template，而不会 import 目标文件 import 的 template。

如: **C import B，B import A**，在**C**中可以使用**B**定义的 `template` ，在**B**中可以使用**A**定义的 `template` ，但是**C**不能使用**A**定义的 `template` 。

```
<!-- A.wxml -->
<template name="A">
  <text> A template </text>
</template>
```

```
<!-- B.wxml -->
<import src="a.wxml"/>
<template name="B">
  <text> B template </text>
</template>
```

```
<!-- C.wxml -->
<import src="b.wxml"/>
<template is="A"/>  <!-- Error! Can not use tempalte when not import A. -->
```

```
4. <template is="B"/>
```

include

`include` 可以将目标文件除了 `<template/>` `<wxs/>` 外的整个代码引入，相当于是拷贝到 `include` 位置，如:

```
1. <!-- index.wxml -->
2. <include src="header.wxml"/>
3. <view> body </view>
4. <include src="footer.wxml"/>
```

```
1. <!-- header.wxml -->
2. <view> header </view>
```

```
1. <!-- footer.wxml -->
2. <view> footer </view>
```

原文:

> *https://developers.weixin.qq.com/miniprogram/dev/framework/view/wxml/import.html*

WXS

WXS

WXS（WeiXin Script）是小程序的一套脚本语言，结合 `WXML` ，可以构建出页面的结构。

注意

- wxs 不依赖于运行时的基础库版本，可以在所有版本的小程序中运行。
- wxs 与 javascript 是不同的语言，有自己的语法，并不和 javascript 一致。
- wxs 的运行环境和其他 javascript 代码是隔离的，wxs 中不能调用其他 javascript 文件中定义的函数，也不能调用小程序提供的API。
- wxs 函数不能作为组件的事件回调。
- 由于运行环境的差异，在 iOS 设备上小程序内的 wxs 会比 javascript 代码快 2 ~ 20 倍。在 android 设备上二者运行效率无差异。

以下是一些使用 WXS 的简单示例：

页面渲染

```
<!--wxml-->
<wxs module="m1">
var msg = "hello world";

module.exports.message = msg;
</wxs>

<view> {{m1.message}} </view>
```

页面输出：

```
hello world
```

数据处理

```
// page.js
Page({
  data: {
    array: [1, 2, 3, 4, 5, 1, 2, 3, 4]
  }
})
```

```
<!--wxml-->
<!-- 下面的 getMax 函数，接受一个数组，且返回数组中最大的元素的值 -->
<wxs module="m1">
var getMax = function(array) {
  var max = undefined;
  for (var i = 0; i < array.length; ++i) {
    max = max === undefined ?
      array[i] :
      (max >= array[i] ? max : array[i]);
  }
  return max;
}

module.exports.getMax = getMax;
</wxs>

<!-- 调用 wxs 里面的 getMax 函数，参数为 page.js 里面的 array -->
<view> {{m1.getMax(array)}} </view>
```

页面输出：

```
5
```

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/view/wxs/*

模块

WXS 模块

WXS 代码可以编写在 wxml 文件中的 `<wxs>` 标签内，或以 `.wxs` 为后缀名的文件内。

模块

每一个 `.wxs` 文件和 `<wxs>` 标签都是一个单独的模块。

每个模块都有自己独立的作用域。即在一个模块里面定义的变量与函数，默认为私有的，对其他模块不可见。

一个模块要想对外暴露其内部的私有变量与函数，只能通过 `module.exports` 实现。

.wxs 文件

在微信开发者工具里面，右键可以直接创建 `.wxs` 文件，在其中直接编写 WXS 脚本。

示例代码：

```
// /pages/comm.wxs

var foo = "'hello world' from comm.wxs";
var bar = function(d) {
  return d;
}
module.exports = {
  foo: foo,
  bar: bar
};
```

上述例子在 `/pages/comm.wxs` 的文件里面编写了 WXS 代码。该 `.wxs` 文件可以被其他的 `.wxs` 文件 或 WXML 中的 `<wxs>` 标签引用。

module 对象

每个 `wxs` 模块均有一个内置的 `module` 对象。

属性

- exports：通过该属性，可以对外共享本模块的私有变量与函数。
  示例代码：

在开发者工具中预览效果

```
// /pages/tools.wxs

var foo = "'hello world' from tools.wxs";
var bar = function (d) {
  return d;
}
module.exports = {
  FOO: foo,
  bar: bar,
};
module.exports.msg = "some msg";
```

```
<!-- page/index/index.wxml -->

<wxs src="./../tools.wxs" module="tools" />
<view> {{tools.msg}} </view>
<view> {{tools.bar(tools.FOO)}} </view>
```

页面输出：

```
some msg
'hello world' from tools.wxs
```

require函数

在 `.wxs` 模块中引用其他 `wxs` 文件模块，可以使用 `require` 函数。

引用的时候，要注意如下几点：

- 只能引用 .wxs 文件模块，且必须使用相对路径。
- wxs 模块均为单例，wxs 模块在第一次被引用时，会自动初始化为单例对象。多个页面，多个地方，多次引用，使用的都是同一个 wxs 模块对象。
- 如果一个 wxs 模块在定义之后，一直没有被引用，则该模块不会被解析与运行。
  示例代码：

在开发者工具中预览效果

```
// /pages/tools.wxs

var foo = "'hello world' from tools.wxs";
var bar = function (d) {
  return d;
}
module.exports = {
  FOO: foo,
  bar: bar,
};
module.exports.msg = "some msg";
```

```
// /pages/logic.wxs

var tools = require("./tools.wxs");

console.log(tools.FOO);
console.log(tools.bar("logic.wxs"));
console.log(tools.msg);
```

```
<!-- /page/index/index.wxml -->

<wxs src="./../logic.wxs" module="logic" />
```

控制台输出：

```
'hello world' from tools.wxs
logic.wxs
some msg
```

<wxs> 标签

属性名	类型	默认值	说明
module	String		当前 `<wxs>` 标签的模块名。必填字段。
src	String		引用 .wxs 文件的相对路径。仅当本标签为单闭合标签或标签的内容为空时有效。

module 属性

module 属性是当前 `<wxs>` 标签的模块名。在单个 wxml 文件内，建议其值唯一。有重复模块名则按照先后顺序覆盖（后者覆盖前者）。不同文件之间的 wxs 模块名不会相互覆盖。

module 属性值的命名必须符合下面两个规则：

- 首字符必须是：字母（a-zA-Z），下划线（_）
- 剩余字符可以是：字母（a-zA-Z），下划线（_）， 数字（0-9）
  示例代码：

在开发者工具中预览效果

```
<!--wxml-->

<wxs module="foo">
var some_msg = "hello world";
module.exports = {
    msg : some_msg,
}
</wxs>
```

```
<view> {{foo.msg}} </view>
```

页面输出：

```
hello world
```

上面例子声明了一个名字为 `foo` 的模块，将 `some_msg` 变量暴露出来，供当前页面使用。

src 属性

`src` 属性可以用来引用其他的 `wxs` 文件模块。

引用的时候，要注意如下几点：

- 只能引用 `.wxs` 文件模块，且必须使用相对路径。
- `wxs` 模块均为单例，`wxs` 模块在第一次被引用时，会自动初始化为单例对象。多个页面，多个地方，多次引用，使用的都是同一个 `wxs` 模块对象。
- 如果一个 `wxs` 模块在定义之后，一直没有被引用，则该模块不会被解析与运行。
  示例代码：

在开发者工具中预览效果

```
// /pages/index/index.js

Page({
  data: {
    msg: "'hello wrold' from js",
  }
})
```

```
<!-- /pages/index/index.wxml -->

<wxs src="./../comm.wxs" module="some_comms"></wxs>
<!-- 也可以直接使用单标签闭合的写法
<wxs src="./../comm.wxs" module="some_comms" />
-->

<!-- 调用 some_comms 模块里面的 bar 函数，且参数为 some_comms 模块里面的 foo -->
<view> {{some_comms.bar(some_comms.foo)}} </view>
<!-- 调用 some_comms 模块里面的 bar 函数，且参数为 page/index/index.js 里面的 msg -->
<view> {{some_comms.bar(msg)}} </view>
```

页面输出：

```
'hello world' from comm.wxs
'hello wrold' from js
```

上述例子在文件 `/page/index/index.wxml` 中通过 `<wxs>` 标签引用了 `/page/comm.wxs` 模块。

注意

- 模块只能在定义模块的 WXML 文件中被访问到。使用 或 时， 模块不会被引入到对应的 WXML 文件中。
-

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/view/wxs/01wxs-module.html*

变量

变量

概念

- WXS 中的变量均为值的引用。
- 没有声明的变量直接赋值使用，会被定义为全局变量。
- 如果只声明变量而不赋值，则默认值为 undefined。
- var表现与javascript一致，会有变量提升。

```
var foo = 1;
var bar = "hello world";
var i; // i === undefined
```

上面代码，分别声明了 `foo` 、 `bar` 、 `i` 三个变量。然后， `foo` 赋值为数值 `1` ， `bar` 赋值为字符串 `"hello world"` 。

变量名

变量命名必须符合下面两个规则：

- 首字符必须是：字母（a-zA-Z），下划线（_）
- 剩余字符可以是：字母（a-zA-Z），下划线（_）， 数字（0-9）

保留标识符

以下标识符不能作为变量名：

```
delete
void
typeof

null
undefined
NaN
Infinity
var

if
else

true
false

```

```
require

this
function
arguments
return

for
while
do
break
continue
switch
case
default
```

原文：

> https://developers.weixin.qq.com/miniprogram/dev/framework/view/wxs/02variate.html

注释

注释

WXS 主要有 3 种注释的方法。

示例代码：

```
<!-- wxml -->
<wxs module="sample">
// 方法一：单行注释

/*
方法二：多行注释
*/

/*
方法三：结尾注释。即从 /* 开始往后的所有 WXS 代码均被注释
var a = 1;
var b = 2;
var c = "fake";

</wxs>
```

上述例子中，所有 WXS 代码均被注释掉了。

> 方法三 和 方法二 的唯一区别是，没有 `*/` 结束符。

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/view/wxs/03annotation.html*

运算符

运算符

基本运算符

示例代码：

```
var a = 10, b = 20;

// 加法运算
console.log(30 === a + b);
// 减法运算
console.log(-10 === a - b);
// 乘法运算
console.log(200 === a * b);
// 除法运算
console.log(0.5 === a / b);
// 取余运算
console.log(10 === a % b);
```

- 加法运算（+）也可以用作字符串的拼接。

```
var a = '.w' , b = 'xs';
```

```
// 字符串拼接
console.log('.wxs' === a + b);
```

一元运算符

示例代码：

```
var a = 10, b = 20;

// 自增运算
console.log(10 === a++);
console.log(12 === ++a);
// 自减运算
console.log(12 === a--);
console.log(10 === --a);
// 正值运算
console.log(10 === +a);
// 负值运算
console.log(0-10 === -a);
```

```
// 否运算
console.log(-11 === ~a);
// 取反运算
console.log(false === !a);
// delete 运算
console.log(true === delete a.fake);
// void 运算
console.log(undefined === void a);
// typeof 运算
console.log("number" === typeof a);
```

位运算符

示例代码：

```
var a = 10, b = 20;

// 左移运算
console.log(80 === (a << 3));
// 无符号右移运算
console.log(2 === (a >> 2));
// 带符号右移运算
console.log(2 === (a >>> 2));
// 与运算
console.log(2 === (a & 3));
// 异或运算
console.log(9 === (a ^ 3));
// 或运算
console.log(11 === (a | 3));
```

比较运算符

示例代码：

```
var a = 10, b = 20;

// 小于
console.log(true === (a < b));
// 大于
console.log(false === (a > b));
// 小于等于
console.log(true === (a <= b));
// 大于等于
console.log(false === (a >= b));
```

等值运算符

示例代码：

```
var a = 10, b = 20;

// 等号
console.log(false === (a == b));
// 非等号
console.log(true === (a != b));
// 全等号
console.log(false === (a === b));
// 非全等号
console.log(true === (a !== b));
```

赋值运算符

示例代码：

```
var a = 10;

a = 10; a *= 10;
console.log(100 === a);
a = 10; a /= 5;
console.log(2 === a);
a = 10; a %= 7;
console.log(3 === a);
a = 10; a += 5;
console.log(15 === a);
a = 10; a -= 11;
console.log(-1 === a);
a = 10; a <<= 10;
console.log(10240 === a);
a = 10; a >>= 2;
console.log(2 === a);
a = 10; a >>>= 2;
console.log(2 === a);
a = 10; a &= 3;
console.log(2 === a);
a = 10; a ^= 3;
console.log(9 === a);
a = 10; a |= 3;
console.log(11 === a);
```

二元逻辑运算符

示例代码：

```
var a = 10, b = 20;

```

```
// 逻辑与
console.log(20 === (a && b));
// 逻辑或
console.log(10 === (a || b));
```

其他运算符

示例代码:

```
var a = 10, b = 20;

//条件运算符
console.log(20 === (a >= 10 ? a + 10 : b + 10));
//逗号运算符
console.log(20 === (a, b));
```

运算符优先级

优先级	运算符	说明	结合性
20	`(` … `)`	括号	n/a
19	… `.` …	成员访问	从左到右
	… `[` … `]`	成员访问	从左到右
	… `(` … `)`	函数调用	从左到右
17	… `++`	后置递增	n/a
	… `--`	后置递减	n/a
16	`!` …	逻辑非	从右到左
	`~` …	按位非	从右到左
	`+` …	一元加法	从右到左
	`-` …	一元减法	从右到左
	`++` …	前置递增	从右到左
	`--` …	前置递减	从右到左
	`typeof` …	typeof	从右到左
	`void` …	void	从右到左
	`delete` …	delete	从右到左
14	… `*` …	乘法	从左到右
	… `/` …	除法	从左到右
	… `%` …	取模	从左到右
13	… `+` …	加法	从左到右
	… `-` …	减法	从左到右
12	… `<<` …	按位左移	从左到右

	... `>>` ...	按位右移	从左到右
	... `>>>` ...	无符号右移	从左到右
11	... `<` ...	小于	从左到右
	... `<=` ...	小于等于	从左到右
	... `>` ...	大于	从左到右
	... `>=` ...	大于等于	从左到右
10	... `==` ...	等号	从左到右
	... `!=` ...	非等号	从左到右
	... `===` ...	全等号	从左到右
	... `!==` ...	非全等号	从左到右
9	... `&` ...	按位与	从左到右
8	... `^` ...	按位异或	从左到右
7	... `|` ...	按位或	从左到右
6	... `&&` ...	逻辑与	从左到右
5	... `||` ...	逻辑或	从左到右
4	... `?` ... `:` ...	条件运算符	从右到左
3	... `=` ...	赋值	从右到左
	... `+=` ...	赋值	从右到左
	... `-=` ...	赋值	从右到左
	... `*=` ...	赋值	从右到左
	... `/=` ...	赋值	从右到左
	... `%=` ...	赋值	从右到左
	... `<<=` ...	赋值	从右到左
	... `>>=` ...	赋值	从右到左
	... `>>>=` ...	赋值	从右到左
	... `&=` ...	赋值	从右到左
	... `^=` ...	赋值	从右到左
	... `|=` ...	赋值	从右到左
0	... `,` ...	逗号	从左到右

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/view/wxs/04operator.html*

语句

语句

if 语句

在 WXS 中，可以使用以下格式的 `if` 语句：

- if (expression) statement：当 expression 为 truthy 时，执行 statement。
- if (expression) statement1 else statement2：当 expression 为 truthy 时，执行 statement1。否则，执行 statement2
- if … else if … else statementN 通过该句型，可以在 statement1 ~ statementN 之间选其中一个执行。

示例语法：

```
// if ...

if (表达式) 语句;

if (表达式)
  语句;

if (表达式) {
  代码块;
}


// if ... else

if (表达式) 语句;
else 语句;

if (表达式)
  语句;
else
  语句;

if (表达式) {
  代码块;
} else {
  代码块;
}

// if ... else if ... else ...

```

```
if (表达式) {
  代码块;
} else if (表达式) {
  代码块;
} else if (表达式) {
  代码块;
} else {
  代码块;
}
```

switch 语句

示例语法：

```
switch (表达式) {
  case 变量:
    语句;
  case 数字:
    语句;
    break;
  case 字符串:
    语句;
  default:
    语句;
}
```

- default 分支可以省略不写。
- case 关键词后面只能使用：变量，数字，字符串。
  示例代码：

```
var exp = 10;

switch ( exp ) {
case "10":
  console.log("string 10");
  break;
case 10:
  console.log("number 10");
  break;
case exp:
  console.log("var exp");
  break;
default:
  console.log("default");
}
```

输出：

```
number 10
```

for 语句

示例语法：

```
for (语句; 语句; 语句)
  语句;

for (语句; 语句; 语句) {
  代码块;
}
```

- 支持使用 break，continue 关键词。
  示例代码：

```
for (var i = 0; i < 3; ++i) {
  console.log(i);
  if( i >= 1) break;
}
```

输出：

```
0
1
```

while 语句

示例语法：

```
while (表达式)
  语句;

while (表达式){
  代码块;
}

do {
  代码块;
} while (表达式)
```

- 当表达式为 true 时，循环执行语句或代码块。
- 支持使用 break，continue 关键词。

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/view/wxs/05statement.html*

数据类型

数据类型

WXS 语言目前共有以下几种数据类型：

- number ： 数值
- string : 字符串
- boolean: 布尔值
- object: 对象
- function: 函数
- array ： 数组
- date: 日期
- regexp: 正则

number

语法

number 包括两种数值：整数，小数。

```
var a = 10;
var PI = 3.141592653589793;
```

属性

- constructor: 返回字符串 "Number"。

方法

- toString
- toLocaleString
- valueOf
- toFixed
- toExponential
- toPrecision

> 以上方法的具体使用请参考 *ES5* 标准。

string

语法

string 有两种写法:

```
1. 'hello world';
2. "hello world";
```

属性

- constructor: 返回字符串 "String"。
- length

> 除*constructor*外属性的具体含义请参考 *ES5* 标准。

方法

- toString
- valueOf
- charAt
- charCodeAt
- concat
- indexOf
- lastIndexOf
- localeCompare
- match
- replace
- search
- slice
- split
- substring
- toLowerCase
- toLocaleLowerCase
- toUpperCase
- toLocaleUpperCase
- trim

> 以上方法的具体使用请参考 *ES5* 标准。

boolean

语法

布尔值只有两个特定的值: `true` 和 `false` 。

属性

- constructor: 返回字符串 "Boolean"。

方法

- toString
- valueOf

> 以上方法的具体使用请参考 *ES5* 标准。

object

语法

object 是一种无序的键值对。使用方法如下所示：

```
var o = {} //生成一个新的空对象

//生成一个新的非空对象
o = {
  'string'  : 1,  //object 的 key 可以是字符串
  const_var : 2,  //object 的 key 也可以是符合变量定义规则的标识符
  func      : {}, //object 的 value 可以是任何类型
};

//对象属性的读操作
console.log(1 === o['string']);
console.log(2 === o.const_var);

//对象属性的写操作
o['string']++;
o['string'] += 10;
o.const_var++;
o.const_var += 10;

//对象属性的读操作
console.log(12 === o['string']);
console.log(13 === o.const_var);
```

属性

- constructor: 返回字符串 "Object"。

```
console.log("Object" === {k:"1",v:"2"}.constructor)
```

方法

- toString: 返回字符串 "[object Object]"。

function

语法

function 支持以下的定义方式:

```
//方法 1
function a (x) {
  return x;
}

//方法 2
var b = function (x) {
  return x;
}
```

function 同时也支持以下的语法（匿名函数，闭包等）:

```
var a = function (x) {
  return function () { return x;}
}

var b = a(100);
console.log( 100 === b() );
```

arguments

function 里面可以使用 `arguments` 关键词。该关键词目前只支持以下的属性:

- length: 传递给函数的参数个数。
- [index]: 通过 index 下标可以遍历传递给函数的每个参数。
  示例代码:

```
var a = function(){
    console.log(3 === arguments.length);
    console.log(100 === arguments[0]);
    console.log(200 === arguments[1]);
    console.log(300 === arguments[2]);
};
a(100,200,300);
```

属性

- constructor: 返回字符串 "Function"。
- length: 返回函数的形参个数。

方法

- toString: 返回字符串 "[function Function]"。
  示例代码:

```
var func = function (a,b,c) { }

console.log("Function" === func.constructor);
console.log(3 === func.length);
console.log("[function Function]" === func.toString());
```

array

语法

array 支持以下的定义方式:

```
var a = [];      //生成一个新的空数组

a = [1,"2",{},function(){}];  //生成一个新的非空数组, 数组元素可以是任何类型
```

属性

- constructor: 返回字符串 "Array"。
- length

> 除*constructor*外属性的具体含义请参考 *ES5* 标准。

方法

- toString
- concat
- join
- pop
- push
- reverse
- shift
- slice
- sort
- splice
- unshift
- indexOf
- lastIndexOf
- every
- some

- forEach
- map
- filter
- reduce
- reduceRight

> 以上方法的具体使用请参考 *ES5* 标准。

date

语法

生成 date 对象需要使用 `getDate` 函数，返回一个当前时间的对象。

```
1. getDate()
2. getDate(milliseconds)
3. getDate(datestring)
4. getDate(year, month[, date[, hours[, minutes[, seconds[, milliseconds]]]]])
```

- 参数
  - milliseconds: 从1970年1月1日00:00:00 UTC开始计算的毫秒数
  - datestring: 日期字符串，其格式为: "month day, year hours seconds"
    示例代码:

```
1. var date = getDate(); //返回当前时间对象
2.
3. date = getDate(1500000000000);
4. // Fri Jul 14 2017 10:40:00 GMT+0800 (中国标准时间)
5. date = getDate('2017-7-14');
6. // Fri Jul 14 2017 00:00:00 GMT+0800 (中国标准时间)
7. date = getDate(2017, 6, 14, 10, 40, 0, 0);
8. // Fri Jul 14 2017 10:40:00 GMT+0800 (中国标准时间)
```

属性

- constructor: 返回字符串 “Date”。

方法

- toString
- toDateString
- toTimeString
- toLocaleString
- toLocaleDateString

- toLocaleTimeString
- valueOf
- getTime
- getFullYear
- getUTCFullYear
- getMonth
- getUTCMonth
- getDate
- getUTCDate
- getDay
- getUTCDay
- getHours
- getUTCHours
- getMinutes
- getUTCMinutes
- getSeconds
- getUTCSeconds
- getMilliseconds
- getUTCMilliseconds
- getTimezoneOffset
- setTime
- setMilliseconds
- setUTCMilliseconds
- setSeconds
- setUTCSeconds
- setMinutes
- setUTCMinutes
- setHours
- setUTCHours
- setDate
- setUTCDate
- setMonth
- setUTCMonth
- setFullYear
- setUTCFullYear
- toUTCString
- toISOString
- toJSON

> 以上方法的具体使用请参考 *ES5* 标准。

regexp

语法

生成 regexp 对象需要使用 `getRegExp` 函数。

```
getRegExp(pattern[, flags])
```

- 参数:
    - pattern: 正则表达式的内容。
    - flags: 修饰符。该字段只能包含以下字符:
        - g: global
        - i: ignoreCase
        - m: multiline。

          示例代码:

```
var a = getRegExp("x", "img");
console.log("x" === a.source);
console.log(true === a.global);
console.log(true === a.ignoreCase);
console.log(true === a.multiline);
```

属性

- constructor: 返回字符串 "RegExp"。
- source
- global
- ignoreCase
- multiline
- lastIndex

> 除*constructor*外属性的具体含义请参考 `ES5` 标准。

方法

- exec
- test
- toString

> 以上方法的具体使用请参考 `ES5` 标准。

数据类型判断

constructor 属性

数据类型的判断可以使用 `constructor` 属性。

示例代码:

```
var number = 10;
console.log( "Number" === number.constructor );

```

```
var string = "str";
console.log( "String" === string.constructor );

var boolean = true;
console.log( "Boolean" === boolean.constructor );

var object = {};
console.log( "Object" === object.constructor );

var func = function(){};
console.log( "Function" === func.constructor );

var array = [];
console.log( "Array" === array.constructor );

var date = getDate();
console.log( "Date" === date.constructor );

var regexp = getRegExp();
console.log( "RegExp" === regexp.constructor );
```

typeof

使用 `typeof` 也可以区分部分数据类型。

示例代码：

```
var number = 10;
var boolean = true;
var object = {};
var func = function(){};
var array = [];
var date = getDate();
var regexp = getRegExp();

console.log( 'number' === typeof number );
console.log( 'boolean' === typeof boolean );
console.log( 'object' === typeof object );
console.log( 'function' === typeof func );
console.log( 'object' === typeof array );
console.log( 'object' === typeof date );
console.log( 'object' === typeof regexp );

console.log( 'undefined' === typeof undefined );
console.log( 'object' === typeof null );
```

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/view/wxs/06datatype.html*

基础类库

基础类库

console

`console.log` 方法用于在 console 窗口输出信息。它可以接受多个参数，将它们的结果连接起来输出。

Math

属性

- E
- LN10
- LN2
- LOG2E
- LOG10E
- PI
- SQRT1_2
- SQRT2

> 以上属性的具体使用请参考 *ES5* 标准。

方法

- abs
- acos
- asin
- atan
- atan2
- ceil
- cos
- exp
- floor
- log
- max
- min
- pow
- random
- round
- sin
- sqrt

- tan

> 以上方法的具体使用请参考 *ES5* 标准。

JSON

方法

- stringify(object): 将 object 对象转换为 JSON 字符串，并返回该字符串。
- parse(string): 将 JSON 字符串转化成对象，并返回该对象。
  示例代码:

```
1.
2. console.log(undefined === JSON.stringify());
3. console.log(undefined === JSON.stringify(undefined));
4. console.log("null"===JSON.stringify(null));
5.
6. console.log("111"===JSON.stringify(111));
7. console.log('"111"'===JSON.stringify("111"));
8. console.log("true"===JSON.stringify(true));
9. console.log(undefined===JSON.stringify(function(){}));
10.
11.
12. console.log(undefined===JSON.parse(JSON.stringify()));
13. console.log(undefined===JSON.parse(JSON.stringify(undefined)));
14. console.log(null===JSON.parse(JSON.stringify(null)));
15.
16. console.log(111===JSON.parse(JSON.stringify(111)));
17. console.log("111"===JSON.parse(JSON.stringify("111")));
18. console.log(true===JSON.parse(JSON.stringify(true)));
19.
20. console.log(undefined===JSON.parse(JSON.stringify(function(){})));
```

Number

属性

- MAX_VALUE
- MIN_VALUE
- NEGATIVE_INFINITY
- POSITIVE_INFINITY

> 以上属性的具体使用请参考 *ES5* 标准。

Date

属性

- parse
- UTC
- now

> 以上属性的具体使用请参考 `ES5` 标准。

Global

属性

- NaN
- Infinity
- undefined

> 以上属性的具体使用请参考 `ES5` 标准。

方法

- parseInt
- parseFloat
- isNaN
- isFinite
- decodeURI
- decodeURIComponent
- encodeURI
- encodeURIComponent

> 以上方法的具体使用请参考 `ES5` 标准。

原文:

> *https://developers.weixin.qq.com/miniprogram/dev/framework/view/wxs/07basiclibrary.html*

WXSS

WXSS

WXSS(WeiXin Style Sheets)是一套样式语言，用于描述 WXML 的组件样式。

WXSS 用来决定 WXML 的组件应该怎么显示。

为了适应广大的前端开发者，WXSS 具有 CSS 大部分特性。同时为了更适合开发微信小程序，WXSS 对 CSS 进行了扩充以及修改。

与 CSS 相比，WXSS 扩展的特性有：

- 尺寸单位
- 样式导入

尺寸单位

- rpx（responsive pixel）：可以根据屏幕宽度进行自适应。规定屏幕宽为750rpx。如在 iPhone6 上，屏幕宽度为375px，共有750个物理像素，则750rpx = 375px = 750物理像素，1rpx = 0.5px = 1物理像素。
  |设备|rpx换算px (屏幕宽度/750)|px换算rpx (750/屏幕宽度)
  |——-
  |iPhone5|1rpx = 0.42px|1px = 2.34rpx
  |iPhone6|1rpx = 0.5px|1px = 2rpx
  |iPhone6 Plus|1rpx = 0.552px|1px = 1.81rpx

建议： 开发微信小程序时设计师可以用 iPhone6 作为视觉稿的标准。

注意： 在较小的屏幕上不可避免的会有一些毛刺，请在开发时尽量避免这种情况。

样式导入

使用 `@import` 语句可以导入外联样式表， `@import` 后跟需要导入的外联样式表的相对路径，用 `;` 表示语句结束。

示例代码：

```
/** common.wxss **/
.small-p {
  padding:5px;
}
```

```
/** app.wxss **/
@import "common.wxss";
.middle-p {
```

```
    padding:15px;
}
```

内联样式

框架组件上支持使用 style、class 属性来控制组件的样式。

- style：静态的样式统一写到 class 中。style 接收动态的样式，在运行时会进行解析，请尽量避免将静态的样式写进 style 中，以免影响渲染速度。

```
<view style="color:{{color}};" />
```

- class：用于指定样式规则，其属性值是样式规则中类选择器名(样式类名)的集合，样式类名不需要带上.，样式类名之间用空格分隔。

```
<view class="normal_view" />
```

选择器

目前支持的选择器有：

选择器	样例	样例描述
.class	`.intro`	选择所有拥有 class="intro" 的组件
#id	`#firstname`	选择拥有 id="firstname" 的组件
element	`view`	选择所有 view 组件
element, element	`view, checkbox`	选择所有文档的 view 组件和所有的 checkbox 组件
::after	`view::after`	在 view 组件后边插入内容
::before	`view::before`	在 view 组件前边插入内容

全局样式与局部样式

定义在 app.wxss 中的样式为全局样式，作用于每一个页面。在 page 的 wxss 文件中定义的样式为局部样式，只作用在对应的页面，并会覆盖 app.wxss 中相同的选择器。

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/view/wxss.html*

组件

基础组件

框架为开发者提供了一系列基础组件，开发者可以通过组合这些基础组件进行快速开发。

详细介绍请参考组件文档

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/view/component.html*

自定义组件

自定义组件

从小程序基础库版本 1.6.3 开始，小程序支持简洁的组件化编程。所有自定义组件相关特性都需要基础库版本 1.6.3 或更高。

开发者可以将页面内的功能模块抽象成自定义组件，以便在不同的页面中重复使用；也可以将复杂的页面拆分成多个低耦合的模块，有助于代码维护。自定义组件在使用时与基础组件非常相似。

创建自定义组件

类似于页面，一个自定义组件由 `json` `wxml` `wxss` `js` 4个文件组成。要编写一个自定义组件，首先需要在 `json` 文件中进行自定义组件声明（将 `component` 字段设为 `true` 可这一组文件设为自定义组件）：

```
{
  "component": true
}
```

同时，还要在 `wxml` 文件中编写组件模版，在 `wxss` 文件中加入组件样式，它们的写法与页面的写法类似。具体细节和注意事项参见 组件模版和样式 。

代码示例：

```
<!-- 这是自定义组件的内部WXML结构 -->
<view class="inner">
  {{innerText}}
</view>
<slot></slot>
```

```
/* 这里的样式只应用于这个自定义组件 */
.inner {
  color: red;
}
```

注意：在组件wxss中不应使用ID选择器、属性选择器和标签名选择器。

在自定义组件的 `js` 文件中，需要使用 `Component()` 来注册组件，并提供组件的属性定义、内部数据和自定义方法。

组件的属性值和内部数据将被用于组件 `wxml` 的渲染，其中，属性值是可由组件外部传入的。更多细节参见 Component构造器 。

代码示例：

```
Component({
  properties: {
    // 这里定义了innerText属性，属性值可以在组件使用时指定
    innerText: {
      type: String,
      value: 'default value',
    }
  },
  data: {
    // 这里是一些组件内部数据
    someData: {}
  },
  methods: {
    // 这里是一个自定义方法
    customMethod: function(){}
  }
})
```

使用自定义组件

使用已注册的自定义组件前，首先要在页面的 `json` 文件中进行引用声明。此时需要提供每个自定义组件的标签名和对应的自定义组件文件路径：

```
{
  "usingComponents": {
    "component-tag-name": "path/to/the/custom/component"
  }
}
```

这样，在页面的 `wxml` 中就可以像使用基础组件一样使用自定义组件。节点名即自定义组件的标签名，节点属性即传递给组件的属性值。

代码示例：

在开发者工具中预览效果

```
<view>
  <!-- 以下是对一个自定义组件的引用 -->
  <component-tag-name inner-text="Some text"></component-tag-name>
</view>
```

自定义组件的 `wxml` 节点结构在与数据结合之后，将被插入到引用位置内。

Tips：

- 对于基础库的1.5.x版本， 1.5.7 也有部分自定义组件支持。
- 因为WXML节点标签名只能是小写字母、中划线和下划线的组合，所以自定义组件的标签名也只能包含这些字符。
- 自定义组件也是可以引用自定义组件的，引用方法类似于页面引用自定义组件的方式（使用

usingComponents 字段）。

- 自定义组件和使用自定义组件的页面所在项目根目录名不能以“wx-”为前缀，否则会报错。
- 旧版本的基础库不支持自定义组件，此时，引用自定义组件的节点会变为默认的空节点。

原文：

> https://developers.weixin.qq.com/miniprogram/dev/framework/custom-component/

组件模版和样式

组件模版和样式

类似于页面，自定义组件拥有自己的 `wxml` 模版和 `wxss` 样式。

组件模版

组件模版的写法与页面模板相同。组件模版与组件数据结合后生成的节点树，将被插入到组件的引用位置上。

在组件模板中可以提供一个 `<slot>` 节点，用于承载组件引用时提供的子节点。

代码示例：

在开发者工具中预览效果

```
<!-- 组件模板 -->
<view class="wrapper">
  <view>这里是组件的内部节点</view>
  <slot></slot>
</view>
```

```
<!-- 引用组件的页面模版 -->
<view>
  <component-tag-name>
    <!-- 这部分内容将被放置在组件 <slot> 的位置上 -->
    <view>这里是插入到组件slot中的内容</view>
  </component-tag-name>
</view>
```

注意，在模版中引用到的自定义组件及其对应的节点名需要在 `json` 文件中显式定义，否则会被当作一个无意义的节点。除此以外，节点名也可以被声明为抽象节点。

模版数据绑定

与普通的 WXML 模版类似，可以使用数据绑定，这样就可以向子组件的属性传递动态数据。

代码示例：

```
<!-- 引用组件的页面模版 -->
<view>
  <component-tag-name prop-a="{{dataFieldA}}" prop-b="{{dataFieldB}}">
    <!-- 这部分内容将被放置在组件 <slot> 的位置上 -->
    <view>这里是插入到组件slot中的内容</view>
  </component-tag-name>
</view>
```

在以上例子中，组件的属性 `propA` 和 `propB` 将收到页面传递的数据。页面可以通过 `setData` 来改变绑定的数据字段。

注意：这样的数据绑定只能传递 JSON 兼容数据。自基础库版本 2.0.9 开始，还可以在数据中包含函数（但这些函数不能在 WXML 中直接调用，只能传递给子组件）。

组件wxml的slot

在组件的wxml中可以包含 `slot` 节点，用于承载组件使用者提供的wxml结构。

默认情况下，一个组件的wxml中只能有一个slot。需要使用多slot时，可以在组件js中声明启用。

```
Component({
  options: {
    multipleSlots: true // 在组件定义时的选项中启用多slot支持
  },
  properties: { /* ... */ },
  methods: { /* ... */ }
})
```

此时，可以在这个组件的wxml中使用多个slot，以不同的 `name` 来区分。

```
<!-- 组件模板 -->
<view class="wrapper">
  <slot name="before"></slot>
  <view>这里是组件的内部细节</view>
  <slot name="after"></slot>
</view>
```

使用时，用 `slot` 属性来将节点插入到不同的slot上。

```
<!-- 引用组件的页面模版 -->
<view>
  <component-tag-name>
    <!-- 这部分内容将被放置在组件 <slot name="before"> 的位置上 -->
    <view slot="before">这里是插入到组件slot name="before"中的内容</view>
    <!-- 这部分内容将被放置在组件 <slot name="after"> 的位置上 -->
    <view slot="after">这里是插入到组件slot name="after"中的内容</view>
  </component-tag-name>
</view>
```

组件样式

组件对应 `wxss` 文件的样式，只对组件wxml内的节点生效。编写组件样式时，需要注意以下几点：

- 组件和引用组件的页面不能使用id选择器（#a）、属性选择器（[a]）和标签名选择器，请改用class选择器。

- 组件和引用组件的页面中使用后代选择器（`.a .b`）在一些极端情况下会有非预期的表现，如遇，请避免使用。
- 子元素选择器（`.a>.b`）只能用于 `view` 组件与其子节点之间，用于其他组件可能导致非预期的情况。
- 继承样式，如 `font` 、 `color` ，会从组件外继承到组件内。
- 除继承样式外， `app.wxss` 中的样式、组件所在页面的的样式对自定义组件无效。

```
#a { } /* 在组件中不能使用 */
[a] { } /* 在组件中不能使用 */
button { } /* 在组件中不能使用 */
.a > .b { } /* 除非 .a 是 view 组件节点，否则不一定会生效 */
```

除此以外，组件可以指定它所在节点的默认样式，使用 `:host` 选择器（需要包含基础库 `1.7.2` 或更高版本的开发者工具支持）。

代码示例：

在开发者工具中预览效果

```
/* 组件 custom-component.wxss */
:host {
  color: yellow;
}
```

```
<!-- 页面的 WXML -->
<custom-component>这段文本是黄色的</custom-component>
```

外部样式类

有时，组件希望接受外部传入的样式类（类似于 `view` 组件的 `hover-class` 属性）。此时可以在 `Component` 中用 `externalClasses` 定义段定义若干个外部样式类。这个特性从小程序基础库版本 `1.9.90` 开始支持。

注意：在同一个节点上使用普通样式类和外部样式类时，两个类的优先级是未定义的，因此最好避免这种情况。

代码示例：

```
/* 组件 custom-component.js */
Component({
  externalClasses: ['my-class']
})
```

```
<!-- 组件 custom-component.wxml -->
<custom-component class="my-class">这段文本的颜色由组件外的 class 决定</custom-component>
```

这样，组件的使用者可以指定这个样式类对应的 `class` ，就像使用普通属性一样。

代码示例：

在开发者工具中预览效果

```
<!-- 页面的 WXML -->
<custom-component my-class="red-text" />
```

```
.red-text {
  color: red;
}
```

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/custom-component/wxml-wxss.html*

Component构造器

Component构造器

定义段与示例方法

Component构造器可用于定义组件，调用Component构造器时可以指定组件的属性、数据、方法等。

定义段	类型	是否必填	描述
properties	Object Map	否	组件的对外属性，是属性名到属性设置的映射表，属性设置中可包含三个字段，`type` 表示属性类型、`value` 表示属性初始值、`observer` 表示属性值被更改时的响应函数
data	Object	否	组件的内部数据，和 `properties` 一同用于组件的模版渲染
methods	Object	否	组件的方法，包括事件响应函数和任意的自定义方法，关于事件响应函数的使用，参见 组件事件
behaviors	String Array	否	类似于mixins和traits的组件间代码复用机制，参见 behaviors
created	Function	否	组件生命周期函数，在组件实例进入页面节点树时执行，注意此时不能调用 `setData`
attached	Function	否	组件生命周期函数，在组件实例进入页面节点树时执行
ready	Function	否	组件生命周期函数，在组件布局完成后执行，此时可以获取节点信息（使用 SelectorQuery ）
moved	Function	否	组件生命周期函数，在组件实例被移动到节点树另一个位置时执行
detached	Function	否	组件生命周期函数，在组件实例被从页面节点树移除时执行
relations	Object	否	组件间关系定义，参见 组件间关系
externalClasses	String Array	否	组件接受的外部样式类，参见 外部样式类
options	Object Map	否	一些组件选项，请参见文档其他部分的说明

生成的组件实例可以在组件的方法、生命周期函数和属性 `observer` 中通过 `this` 访问。组件包含一些通用属性和方法。

属性名	类型	描述
is	String	组件的文件路径
id	String	节点id
dataset	String	节点dataset
data	Object	组件数据，包括内部数据和属性值
properties	Object	组件数据，包括内部数据和属性值（与 `data` 一致）

方法名	参数	描述
setData	Object `newData`	设置data并执行视图层渲染
hasBehavior	Object `behavior`	检查组件是否具有 `behavior` （检查时会递归检查被直接或间接引入的所有behavior）
triggerEvent	String `name`, Object `detail`, Object `options`	触发事件，参见 组件事件
createSelectorQuery		创建一个 SelectorQuery 对象，选择器选取范围为这个组件实例内
selectComponent	String `selector`	使用选择器选择组件实例节点，返回匹配到的第一个组件实例对象
selectAllComponents	String `selector`	使用选择器选择组件实例节点，返回匹配到的全部组件实例对象组成的数组
getRelationNodes	String `relationKey`	获取所有这个关系对应的所有关联节点，参见 组件间关系

代码示例：

在开发者工具中预览效果

```
Component({

  behaviors: [],

  properties: {
    myProperty: { // 属性名
      type: String, // 类型（必填），目前接受的类型包括：String, Number, Boolean, Object, Array, null（表示任意类型）
      value: '', // 属性初始值（可选），如果未指定则会根据类型选择一个
      observer: function(newVal, oldVal, changedPath) {
         // 属性被改变时执行的函数（可选），也可以写成在methods段中定义的方法名字符串，如：'_propertyChange'
         // 通常 newVal 就是新设置的数据， oldVal 是旧数据
      }
    },
    myProperty2: String // 简化的定义方式
  },
  data: {}, // 私有数据，可用于模版渲染

  // 生命周期函数，可以为函数，或一个在methods段中定义的方法名
  attached: function(){},
  moved: function(){},
  detached: function(){},

  methods: {
    onMyButtonTap: function(){
      this.setData({
        // 更新属性和数据的方法与更新页面数据的方法类似
      })
    },
    // 内部方法建议以下划线开头
    _myPrivateMethod: function(){
      // 这里将 data.A[0].B 设为 'myPrivateData'
```

```
      this.setData({
        'A[0].B': 'myPrivateData'
      })
    },
    _propertyChange: function(newVal, oldVal) {

    }
  }

})
```

注意：在 `properties` 定义段中，属性名采用驼峰写法（ `propertyName` ）；在 `wxml` 中，指定属性值时则对应使用连字符写法（ `component-tag-name property-name="attr value"` ），应用于数据绑定时采用驼峰写法（ `attr="{{propertyName}}"` ）。

使用 Component 构造器构造页面

事实上，小程序的页面也可以视为自定义组件。因而，页面也可以使用 `Component` 构造器构造，拥有与普通组件一样的定义段与实例方法。但此时要求对应 json 文件中包含 `usingComponents` 定义段。

此时，组件的属性可以用于接收页面的参数，如访问页面 `/pages/index/index?paramA=123&paramB=xyz` ，如果声明有属性 `paramA` 或 `paramB` ，则它们会被赋值为 `123` 或 `xyz` 。

代码示例：

```
{
  "usingComponents": {}
}
```

```
Component({

  properties: {
    paramA: Number,
    paramB: String,
  },

  methods: {
    onLoad: function() {
      this.data.paramA // 页面参数 paramA 的值
      this.data.paramB // 页面参数 paramA 的值
    }
  }

})
```

Bug & Tips:

- 使用 this.data 可以获取内部数据和属性值，但不要直接修改它们，应使用 setData 修改。
- 生命周期函数无法在组件方法中通过 this 访问到。

- 属性名应避免以 data 开头，即不要命名成 dataXyz 这样的形式，因为在 WXML 中， data-xyz="" 会被作为节点 dataset 来处理，而不是组件属性。
- 在一个组件的定义和使用时，组件的属性名和data字段相互间都不能冲突（尽管它们位于不同的定义段中）。
- bug ： 对于 type 为 Object 或 Array 的属性，如果通过该组件自身的 this.setData 来改变属性值的一个子字段，则依旧会触发属性 observer ，且 observer 接收到的 newVal 是变化的那个子字段的值， oldVal 为空， changedPath 包含子字段的字段名相关信息。

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/custom-component/component.html*

组件事件

组件间通信与事件

组件间通信

组件间的基本通信方式有一下几种。

- WXML 数据绑定：用于父组件向子组件的指定属性设置数据，仅能设置 JSON 兼容数据（自基础库版本 2.0.9 开始，还可以在数据中包含函数）。具体在 组件模版和样式 章节中介绍。
- 事件：用于子组件向父组件传递数据，可以传递任意数据。
- 如果以上两种方式不足以满足需要，父组件还可以通过 this.selectComponent 方法获取子组件实例对象，这样就可以直接访问组件的任意数据和方法。

监听事件

事件系统是组件间通信的主要方式之一。自定义组件可以触发任意的事件，引用组件的页面可以监听这些事件。关于事件的基本概念和用法，参见 事件 。

监听自定义组件事件的方法与监听基础组件事件的方法完全一致：

代码示例：

```
<!-- 当自定义组件触发“myevent”事件时，调用“onMyEvent”方法 -->
<component-tag-name bindmyevent="onMyEvent" />
<!-- 或者可以写成 -->
<component-tag-name bind:myevent="onMyEvent" />
```

```
Page({
  onMyEvent: function(e){
    e.detail // 自定义组件触发事件时提供的detail对象
  }
})
```

触发事件

自定义组件触发事件时，需要使用 `triggerEvent` 方法，指定事件名、detail对象和事件选项：

代码示例：

在开发者工具中预览效果

```
<!-- 在自定义组件中 -->
```

```
<button bindtap="onTap">点击这个按钮将触发“myevent”事件</button>
```

```
Component({
  properties: {}
  methods: {
    onTap: function(){
      var myEventDetail = {} // detail对象，提供给事件监听函数
      var myEventOption = {} // 触发事件的选项
      this.triggerEvent('myevent', myEventDetail, myEventOption)
    }
  }
})
```

触发事件的选项包括：

选项名	类型	是否必填	默认值	描述
bubbles	Boolean	否	false	事件是否冒泡
composed	Boolean	否	false	事件是否可以穿越组件边界，为false时，事件将只能在引用组件的节点树上触发，不进入其他任何组件内部
capturePhase	Boolean	否	false	事件是否拥有捕获阶段

关于冒泡和捕获阶段的概念，请阅读 事件 章节中的相关说明。

代码示例：

在开发者工具中预览效果

```
// 页面 page.wxml
<another-component bindcustomevent="pageEventListener1">
  <my-component bindcustomevent="pageEventListener2"></my-component>
</another-component>
```

```
// 组件 another-component.wxml
<view bindcustomevent="anotherEventListener">
  <slot />
</view>
```

```
// 组件 my-component.wxml
<view bindcustomevent="myEventListener">
  <slot />
</view>
```

```
// 组件 my-component.js
Component({
  methods: {
```

```
    onTap: function(){
      this.triggerEvent('customevent', {}) // 只会触发 pageEventListener2
      this.triggerEvent('customevent', {}, { bubbles: true }) // 会依次触发 pageEventListener2 、 pageEventListener1
      this.triggerEvent('customevent', {}, { bubbles: true, composed: true }) // 会依次触发 pageEventListener2 、 anotherEventListener 、 pageEventListener1
    }
  }
})
```

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/custom-component/events.html*

behaviors

behaviors

定义和使用 behaviors

`behaviors` 是用于组件间代码共享的特性，类似于一些编程语言中的“mixins”或“traits”。

每个 `behavior` 可以包含一组属性、数据、生命周期函数和方法，组件引用它时，它的属性、数据和方法会被合并到组件中，生命周期函数也会在对应时机被调用。每个组件可以引用多个 `behavior` 。 `behavior` 也可以引用其他 `behavior` 。

`behavior` 需要使用 `Behavior()` 构造器定义。

代码示例：

```
// my-behavior.js
module.exports = Behavior({
  behaviors: [],
  properties: {
    myBehaviorProperty: {
      type: String
    }
  },
  data: {
    myBehaviorData: {}
  },
  attached: function(){},
  methods: {
    myBehaviorMethod: function(){}
  }
})
```

组件引用时，在 `behaviors` 定义段中将它们逐个列出即可。

代码示例：

在开发者工具中预览效果

```
// my-component.js
var myBehavior = require('my-behavior')
Component({
  behaviors: [myBehavior],
  properties: {
    myProperty: {
      type: String
    }
```

```
  },
  data: {
    myData: {}
  },
  attached: function(){},
  methods: {
    myMethod: function(){}
  }
})
```

在上例中， `my-component` 组件定义中加入了 `my-behavior` ，而 `my-behavior` 中包含有 `myBehaviorProperty` 属性、 `myBehaviorData` 数据字段、 `myBehaviorMethod` 方法和一个 `attached` 生命周期函数。这将使得 `my-component` 中最终包含 `myBehaviorProperty` 、 `myProperty` 两个属性， `myBehaviorData` 、 `myData` 两个数据字段，和 `myBehaviorMethod` 、 `myMethod` 两个方法。当组件触发 `attached` 生命周期时，会依次触发 `my-behavior` 中的 `attached` 生命周期函数和 `my-component` 中的 `attached` 生命周期函数。

字段的覆盖和组合规则

组件和它引用的 `behavior` 中可以包含同名的字段，对这些字段的处理方法如下：

- 如果有同名的属性或方法，组件本身的属性或方法会覆盖 behavior 中的属性或方法，如果引用了多个 behavior ，在定义段中靠后 behavior 中的属性或方法会覆盖靠前的属性或方法；
- 如果有同名的数据字段，如果数据是对象类型，会进行对象合并，如果是非对象类型则会进行相互覆盖；
- 生命周期函数不会相互覆盖，而是在对应触发时机被逐个调用。如果同一个 behavior 被一个组件多次引用，它定义的生命周期函数只会被执行一次。

内置 behaviors

自定义组件可以通过引用内置的 `behavior` 来获得内置组件的一些行为。

代码示例：

在开发者工具中预览效果

```
Component({
  behaviors: ['wx://form-field']
})
```

在上例中， `wx://form-field` 代表一个内置 `behavior` ，它使得这个自定义组件有类似于表单控件的行为。

内置 `behavior` 往往会为组件添加一些属性。在没有特殊说明时，组件可以覆盖这些属性来改变它的 `type` 或添加 `observer` 。

wx://form-field

使自定义组件有类似于表单控件的行为。 form 组件可以识别这些自定义组件，并在 submit 事件中返回组件的字

段名及其对应字段值。这将为它添加以下两个属性。

属性名	类型	描述	最低版本
name	String	在表单中的字段名	1.6.7
value	任意	在表单中的字段值	1.6.7

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/custom-component/behaviors.html*

组件间关系

组件间关系

定义和使用组件间关系

有时需要实现这样的组件：

```
<custom-ul>
  <custom-li> item 1 </custom-li>
  <custom-li> item 2 </custom-li>
</custom-ul>
```

这个例子中， `custom-ul` 和 `custom-li` 都是自定义组件，它们有相互间的关系，相互间的通信往往比较复杂。此时在组件定义时加入 `relations` 定义段，可以解决这样的问题。示例：

在开发者工具中预览效果

```
// path/to/custom-ul.js
Component({
  relations: {
    './custom-li': {
      type: 'child', // 关联的目标节点应为子节点
      linked: function(target) {
        // 每次有custom-li被插入时执行，target是该节点实例对象，触发在该节点attached生命周期之后
      },
      linkChanged: function(target) {
        // 每次有custom-li被移动后执行，target是该节点实例对象，触发在该节点moved生命周期之后
      },
      unlinked: function(target) {
        // 每次有custom-li被移除时执行，target是该节点实例对象，触发在该节点detached生命周期之后
      }
    }
  },
  methods: {
    _getAllLi: function(){
      // 使用getRelationNodes可以获得nodes数组，包含所有已关联的custom-li，且是有序的
      var nodes = this.getRelationNodes('path/to/custom-li')
    }
  },
  ready: function(){
    this._getAllLi()
  }
})
```

```
// path/to/custom-li.js
```

```
Component({
  relations: {
    './custom-ul': {
      type: 'parent', // 关联的目标节点应为父节点
      linked: function(target) {
        // 每次被插入到custom-ul时执行，target是custom-ul节点实例对象，触发在attached生命周期之后
      },
      linkChanged: function(target) {
        // 每次被移动后执行，target是custom-ul节点实例对象，触发在moved生命周期之后
      },
      unlinked: function(target) {
        // 每次被移除时执行，target是custom-ul节点实例对象，触发在detached生命周期之后
      }
    }
  }
})
```

注意：必须在两个组件定义中都加入**relations**定义，否则不会生效。

关联一类组件

在开发者工具中预览效果

有时，需要关联的是一类组件，如：

```
<custom-form>
  <view>
    input
    <custom-input></custom-input>
  </view>
  <custom-submit> submit </custom-submit>
</custom-form>
```

`custom-form` 组件想要关联 `custom-input` 和 `custom-submit` 两个组件。此时，如果这两个组件都有同一个 behavior：

```
// path/to/custom-form-controls.js
module.exports = Behavior({
  // ...
})
```

```
// path/to/custom-input.js
var customFormControls = require('./custom-form-controls')
Component({
  behaviors: [customFormControls],
  relations: {
    './custom-form': {
      type: 'ancestor', // 关联的目标节点应为祖先节点
    }
```

```
  }
})
```

```
// path/to/custom-submit.js
var customFormControls = require('./custom-form-controls')
Component({
  behaviors: [customFormControls],
  relations: {
    './custom-form': {
      type: 'ancestor', // 关联的目标节点应为祖先节点
    }
  }
})
```

则在 `relations` 关系定义中，可使用这个behavior来代替组件路径作为关联的目标节点：

```
// path/to/custom-form.js
var customFormControls = require('./custom-form-controls')
Component({
  relations: {
    'customFormControls': {
      type: 'descendant', // 关联的目标节点应为子孙节点
      target: customFormControls
    }
  }
})
```

relations 定义段

`relations` 定义段包含目标组件路径及其对应选项，可包含的选项见下表。

选项	类型	是否必填	描述
type	String	是	目标组件的相对关系，可选的值为 `parent` 、 `child` 、 `ancestor` 、 `descendant`
linked	Function	否	关系生命周期函数，当关系被建立在页面节点树中时触发，触发时机在组件attached生命周期之后
linkChanged	Function	否	关系生命周期函数，当关系在页面节点树中发生改变时触发，触发时机在组件moved生命周期之后
unlinked	Function	否	关系生命周期函数，当关系脱离页面节点树时触发，触发时机在组件detached生命周期之后
target	String	否	如果这一项被设置，则它表示关联的目标节点所应具有的behavior，所有拥有这一behavior的组件节点都会被关联

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/custom-component/relations.html*

抽象节点

抽象节点

这个特性自小程序基础库版本 1.9.6 开始支持。

在组件中使用抽象节点

有时，自定义组件模版中的一些节点，其对应的自定义组件不是由自定义组件本身确定的，而是自定义组件的调用者确定的。这时可以把这个节点声明为“抽象节点”。

例如，我们现在来实现一个“选框组”（selectable-group）组件，它其中可以放置单选框（custom-radio）或者复选框（custom-checkbox）。这个组件的 wxml 可以这样编写：

```
<!-- selectable-group.wxml -->
<view wx:for="{{labels}}">
  <label>
    <selectable disabled="{{false}}"></selectable>
    {{item}}
  </label>
</view>
```

其中，“selectable”不是任何在 json 文件的 `usingComponents` 字段中声明的组件，而是一个抽象节点。它需要在 `componentGenerics` 字段中声明：

```
{
  "componentGenerics": {
    "selectable": true
  }
}
```

使用包含抽象节点的组件

在使用 selectable-group 组件时，必须指定“selectable”具体是哪个组件：

```
<selectable-group generic:selectable="custom-radio" />
```

这样，在生成这个 selectable-group 组件的实例时，“selectable”节点会生成“custom-radio”组件实例。类似地，如果这样使用：

```
<selectable-group generic:selectable="custom-checkbox" />
```

“selectable”节点则会生成“custom-checkbox”组件实例。

注意：上述的 `custom-radio` 和 `custom-checkbox` 需要包含在这个 `wxml` 对应 `json` 文件的 `usingComponents` 定义段中。

```
{
  "usingComponents": {
    "custom-radio": "path/to/custom/radio",
    "custom-checkbox": "path/to/custom/checkbox"
  }
}
```

抽象节点的默认组件

抽象节点可以指定一个默认组件，当具体组件未被指定时，将创建默认组件的实例。默认组件可以在 `componentGenerics` 字段中指定：

```
{
  "componentGenerics": {
    "selectable": {
      "default": "path/to/default/component"
    }
  }
}
```

Tips:

- 节点的 `generic` 引用 `generic:xxx="yyy"` 中，值 `yyy` 只能是静态值，不能包含数据绑定。因而抽象节点特性并不适用于动态决定节点名的场景。

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/custom-component/generics.html*

插件

插件

插件的开发和使用自小程序基础库版本 `1.9.6` 开始支持。

插件是对一组 `js` 接口、自定义组件或页面的封装，用于提供给第三方小程序调用。插件必须嵌入在其他小程序中才能被用户使用。

插件开发者可以像开发小程序一样编写一个插件并上传代码，在插件发布之后，其他小程序方可调用。小程序平台会托管插件代码，其他小程序调用时，上传的插件代码会随小程序一起下载运行。

相对于普通 `js` 文件或自定义组件，插件拥有更强的独立性，拥有独立的 `API` 接口、域名列表等，但同时会受到一些限制，如一些 `API` 无法调用或功能受限。对于一些特殊的接口，如 `wx.login` 和 `wx.requestPayment` ，虽然插件不能直接调用，但可以使用 插件功能页 来间接实现。

对于插件开发者，请阅读开发插件章节；对于插件使用者，请阅读使用插件章节。

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/plugin/*

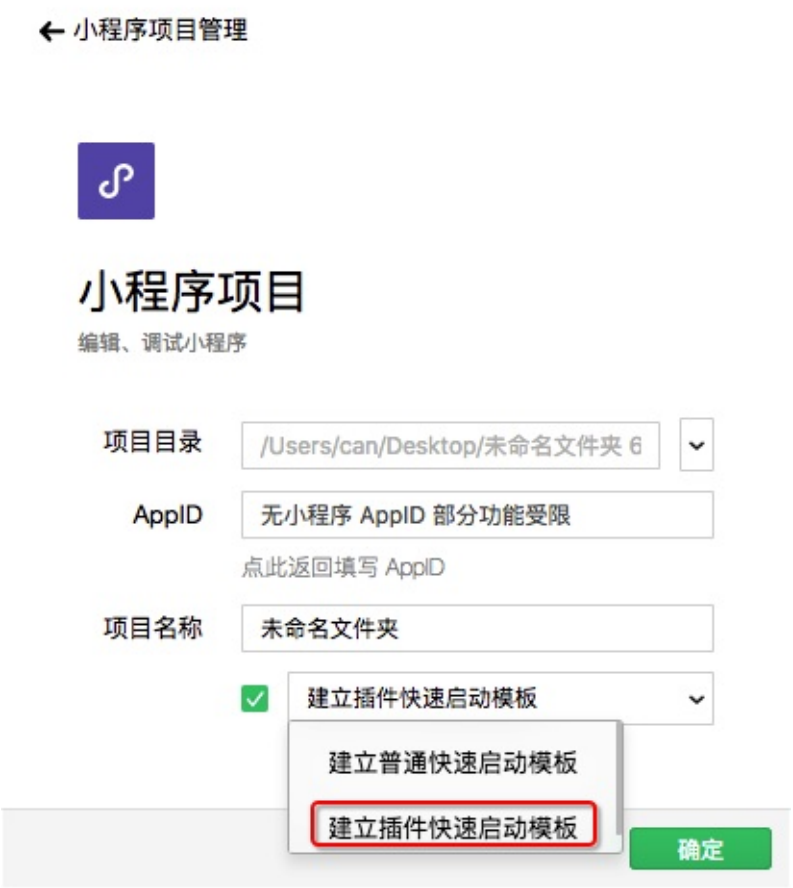
开发插件

开发插件

开发插件前，请阅读了解《小程序插件接入指南》了解开通流程及开放范围，并开通插件功能。如果未开通插件功能，将无法上传插件。

创建插件项目

插件类型的项目可以在开发者工具中直接创建。详情



新建插件类型的项目后，如果创建示例项目，则项目中将包含两个目录：

- `plugin` 目录：插件代码目录。
- `miniprogram` 目录：放置一个小程序，用于调试插件。
- 此外，还可以加入一个 `doc` 目录，用于放置插件开发文档。

`miniprogram` 目录内容可以当成普通小程序来编写，用于插件调试、预览和审核。下面的内容主要介绍 `plugin` 的编写方法。

插件目录结构

一个插件可以包含若干个自定义组件、页面，和一组js接口。插件的目录内容如下：

```
plugin
├── components
│   ├── hello-component.js   // 插件提供的自定义组件（可以有多个）
│   ├── hello-component.json
│   ├── hello-component.wxml
│   └── hello-component.wxss
├── pages
│   ├── hello-page.js        // 插件提供的页面（可以有多个，自小程序基础库版本 [2.1.0](/framework/compatibility.md "基础库 2.1.0 开始支持，低版本需做兼容处理。") 开始支持）
│   ├── hello-page.json
│   ├── hello-page.wxml
│   └── hello-page.wxss
├── index.js                 // 插件的 js 接口
└── plugin.json              // 插件配置文件
```

插件配置文件

插件配置文件 `plugin.json` 主要说明有哪些自定义组件可以供插件外部调用，并标识哪个js文件是插件的js接口文件，如：

代码示例：

```
{
  "publicComponents": {
    "hello-component": "components/hello-component"
  },
  "pages": {
    "hello-page": "pages/hello-page"
  },
  "main": "index.js"
}
```

插件页面跳转

插件的页面从小程序基础库版本 2.1.0 开始支持。

插件执行页面跳转的时候，只能使用 `navigator` 组件。当插件跳转到自身页面时， `url` 应设置为这样的形式： `plugin-private://PLUGIN_APPID/PATH/TO/PAGE` 。需要跳转到其他插件时，也可以这样设置 `url` 。

代码示例：

```
<navigator url="plugin-private://wxidxxxxxxxxxxxxx/pages/hello-page">
  Go to pages/hello-page!
</navigator>
```

插件对外接口

插件内的自定义组件与普通的自定义组件相仿。插件可以定义若干个自定义组件，这些自定义组件都可以在插件内相互引用。其中，提供给外部使用的自定义组件，必须在插件配置文件中显式声明。

插件的 js 接口文件 `index.js` 中可以 export 一些 js 接口，插件的使用者可以使用 `requirePlugin` 来获得这些接口。

代码示例：

```
module.exports = {
  hello: function() {
    console.log('Hello plugin!')
  }
}
```

预览、上传和发布

插件可以像小程序一样预览和上传，但插件没有体验版。

插件会同时有多个线上版本，由使用插件的小程序决定具体使用的版本号。

注意：目前，手机预览插件时将使用一个特殊分配的小程序（即“插件开发助手”）来套用这个插件，这个小程序的 appid 与插件的 appid 不同。服务器端处理插件的网络请求时请留心这个问题。

插件开发文档

除了插件代码本身，小程序开发者可以另外上传一份插件开发文档。这份文档必须放置在插件项目根目录中的 `doc` 目录下，目录结构如下：

```
doc
├── README.md   // 插件文档，应为 markdown 格式
└── picture.jpg // 其他资源文件，仅支持图片
```

其中，引用到的图片资源不能是网络图片，必须放在这个目录下。编辑 `README.md` 之后，可以使用开发者工具预览插件文档和单独上传插件文档。

上传之后，可以使用帐号和密码登录管理后台，在插件设置页面中找到发布插件的链接指引。

插件请求签名

插件在使用 `wx.request` 等 API 发送网络请求时，将会额外携带一个签名 `HostSign` ，用于验证请求来源于小程序插件。这个签名位于请求头中，形如：

```
X-WECHAT-HOSTSIGN: {"noncestr":"NONCESTR", "timestamp":"TIMESTAMP", "signature":"SIGNATURE"}
```

其中， `NONCESTR` 是一个随机字符串， `TIMESTAMP` 是生成这个随机字符串和 `SIGNATURE` 的 UNIX 时间戳。它们是用于计算签名 `SIGNATRUE` 的参数，签名算法为：

```
1. SIGNATURE = sha1([APPID, NONCESTR, TIMESTAMP, TOKEN].sort().join(''))
```

具体来说，这个算法分为几个步骤：

- sort 对 APPID NONCESTR TIMESTAMP TOKEN 四个值表示成字符串形式，按照字典序排序（同 JavaScript 数组的 sort 方法）；
- join 将排好序的四个字符串直接连接在一起；
- 对连接结果使用 sha1 算法，其结果即 SIGNATURE 。

插件开发者可以在服务器上使用这个算法校验签名。其中， `APPID` 是所在小程序的 AppId ； `TOKEN` 是插件 Token ，可以在小程序插件基本设置中找到。
自基础库版本 2.0.7 开始，在小程序运行期间，若网络状况正常， `NONCESTR` 和 `TIMESTAMP` 会每 10 分钟变更一次。如有必要，可以通过判断 `TIMESTAMP` 来确定当前签名是否依旧有效。

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/plugin/development.html*

使用插件

使用插件

申请使用插件

在使用插件前，首先要在小程序管理后台的“设置-第三方服务-插件管理”中添加插件。开发者可登录小程序管理后台，通过 appId 查找插件并添加。插件开发者通过申请后，方可在小程序中使用相应的插件。

引入插件代码包

对于插件的使用者，使用插件前要在 `app.json` 中声明需要使用的插件，例如：

```
{
  "plugins": {
    "myPlugin": {
      "version": "1.0.0",
      "provider": "wxxxxxxxxxxxxxxxx"
    }
  }
}
```

如上例所示， `plugins` 定义段中可以包含多个插件声明，每个插件声明中都必须指明插件的 appid 和需要使用的版本号。

使用插件的 js 接口

在引入插件代码包之后，就可以在这个小程序中使用插件提供的自定义组件或者 js 接口。

如果需要使用插件的 js 接口，可以使用 `requirePlugin` 方法：

```
var myPluginInterface = requirePlugin('myPlugin')

myPluginInterface.hello()
```

使用插件的自定义组件

使用插件提供的自定义组件，和使用普通自定义组件的方式相仿。在 `json` 文件定义需要引入的自定义组件时，使用 `plugin://` 协议即可，例如：

```
{
  "usingComponents": {
```

```
    "hello-component": "plugin://myPlugin/hello-component"
  }
}
```

出于对插件的保护，插件提供的自定义组件在使用上有一定的限制：

- 页面中的 `this.selectComponent` 接口无法获得插件的自定义组件实例对象；
- `wx.createSelectorQuery` 等接口的 `>>>` 选择器无法选入插件内部。

使用插件的页面

插件的页面从小程序基础库版本 `2.1.0` 开始支持。

需要跳转到插件页面时， `url` 应使用 `plugin://` 前缀。

代码示例：

```
<navigator url="plugin://myPlugin/hello-page">
  Go to pages/hello-page!
</navigator>
```

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/plugin/using.html*

插件的限制

插件调用 API 的限制

插件可以调用的 API 与小程序不同，主要有两个区别：

- 插件的请求域名列表与小程序相互独立；
- 一些 API 不允许插件调用。
  注意，虽然一些 API 无法调用，但可以用组件来实现相应的功能。典型的例子是 `wx.navigateTo` 。

目前，允许插件调用的 API 及其对应版本要求如下。

API	最低版本
`wx.addPhoneContact`	1.9.6
`wx.arrayBufferToBase64`	1.9.6
`wx.base64ToArrayBuffer`	1.9.6
`wx.canvasGetImageData`	1.9.6
`wx.canvasPutImageData`	1.9.6
`wx.canvasToTempFilePath`	1.9.6
`wx.chooseImage`	1.9.6
`wx.chooseLocation`	1.9.6
`wx.chooseVideo`	1.9.6
`wx.closeBLEConnection`	1.9.6
`wx.closeBluetoothAdapter`	1.9.6
`wx.connectSocket`	1.9.6
`wx.createAnimation`	1.9.6
`wx.createAudioContext`	1.9.6
`wx.createBLEConnection`	1.9.6
`wx.createCameraContext`	1.9.6
`wx.createCanvasContext`	1.9.6
`wx.createInnerAudioContext`	1.9.6
`wx.createIntersectionObserver`	1.9.6
`wx.createLivePlayerContext`	1.9.6
`wx.createLivePusherContext`	1.9.6
`wx.createMapContext`	1.9.6
`wx.createSelectorQuery`	1.9.6
`wx.createVideoContext`	1.9.6
`wx.downloadFile`	1.9.6

wx.getBLEDeviceCharacteristics	1.9.6
wx.getBLEDeviceServices	1.9.6
wx.getBackgroundAudioManager	1.9.6
wx.getBackgroundAudioPlayerState	1.9.6
wx.getBeacons	1.9.6
wx.getBluetoothAdapterState	1.9.6
wx.getBluetoothDevices	1.9.6
wx.getClipboardData	1.9.6
wx.getConnectedBluetoothDevices	1.9.6
wx.getHCEState	2.1.0
wx.getImageInfo	1.9.6
wx.getLocation	1.9.6
wx.getNetworkType	1.9.6
wx.getRecorderManager	1.9.94
wx.getScreenBrightness	1.9.6
wx.getStorage	1.9.6
wx.getStorageSync	1.9.6
wx.getSystemInfo	1.9.6
wx.getSystemInfoSync	1.9.6
wx.hideLoading	1.9.6
wx.hideToast	1.9.6
wx.makePhoneCall	1.9.6
wx.makeVoIPCall	1.9.6
wx.navigateBack	2.1.0
wx.notifyBLECharacteristicValueChange	1.9.6
wx.notifyBLECharacteristicValueChanged	1.9.6
wx.onAccelerometerChange	1.9.6
wx.onBLECharacteristicValueChange	1.9.6
wx.onBLEConnectionStateChange	1.9.6
wx.onBLEConnectionStateChanged	1.9.6
wx.onBackgroundAudioPause	1.9.6
wx.onBackgroundAudioPlay	1.9.6
wx.onBackgroundAudioStop	1.9.6
wx.onBeaconServiceChange	1.9.6
wx.onBeaconUpdate	1.9.6
wx.onBluetoothAdapterStateChange	1.9.6
wx.onBluetoothDeviceFound	1.9.6

wx.onCompassChange	1.9.6
wx.onHCEMessage	2.1.0
wx.onNetworkStatusChange	1.9.6
wx.onUserCaptureScreen	1.9.6
wx.openBluetoothAdapter	1.9.6
wx.openLocation	1.9.6
wx.pauseBackgroundAudio	1.9.6
wx.pauseVoice	1.9.6
wx.playBackgroundAudio	1.9.6
wx.playVoice	1.9.6
wx.previewImage	1.9.6
wx.readBLECharacteristicValue	1.9.6
wx.removeStorage	1.9.6
wx.removeStorageSync	1.9.6
wx.reportAnalytics	1.9.6
wx.request	1.9.6
wx.saveImageToPhotosAlbum	1.9.6
wx.saveVideoToPhotosAlbum	1.9.6
wx.scanCode	1.9.6
wx.seekBackgroundAudio	1.9.6
wx.sendHCEMessage	2.1.0
wx.setClipboardData	1.9.6
wx.setKeepScreenOn	1.9.6
wx.setScreenBrightness	1.9.6
wx.setStorage	1.9.6
wx.setStorageSync	1.9.6
wx.showActionSheet	1.9.6
wx.showLoading	1.9.6
wx.showModal	1.9.6
wx.showToast	1.9.6
wx.startAccelerometer	1.9.6
wx.startBeaconDiscovery	1.9.6
wx.startBluetoothDevicesDiscovery	1.9.6
wx.startCompass	1.9.6
wx.startHCE	2.1.0
wx.startRecord	1.9.6
wx.stopAccelerometer	1.9.6

`wx.stopBackgroundAudio`	1.9.6
`wx.stopBeaconDiscovery`	1.9.6
`wx.stopBluetoothDevicesDiscovery`	1.9.6
`wx.stopCompass`	1.9.6
`wx.stopHCE`	2.1.0
`wx.stopRecord`	1.9.6
`wx.stopVoice`	1.9.6
`wx.uploadFile`	1.9.6
`wx.vibrateLong`	1.9.6
`wx.vibrateShort`	1.9.6
`wx.writeBLECharacteristicValue`	1.9.6

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/plugin/api-limit.html*

插件功能页

插件功能页

插件功能页从小程序基础库版本 2.1.0 开始支持。

插件不能直接调用 `wx.login` 等较为敏感的接口。在需要访问一些敏感接口时，可以使用插件功能页的方式。使用插件功能页可以实现以下这些功能：

- 获取用户信息，包括 openid 和昵称等（相当于 wx.login 和 wx.getUserInfo 的功能）。
- 支付（相当于 wx.requestPayment ）。
  需要注意的是：插件使用支付功能，需要进行额外的权限申请，申请位置位于管理后台的“小程序插件 -> 基本设置 -> 支付能力”设置项中。另外，无论是否通过申请，主体为个人小程序在使用插件时，都无法正常使用插件里的支付功能。

在具体使用功能页时，插件可以在插件的自定义组件中放置一个 `<functional-page-navigator>` 组件，用户在点击这个组件区域时，可以跳转到一个固定的页面，允许用户执行登录或其他操作。

激活功能页特性

功能页是 插件所有者小程序 中的一个特殊页面。

插件所有者小程序，指的是与插件 AppID 相同的小程序。例如，“小程序示例”小程序开发了一个“小程序示例插件”，无论这个插件被哪个小程序使用，这个插件的插件所有者小程序都是“小程序示例”。

启用插件功能页时，需要在插件所有者小程序 `app.json` 文件中添加 `functionalPages` 定义段，其值为 `true` 。

```
{
  "functionalPages": true
}
```

注意，新增或改变这个字段时，需要这个小程序发布新版本，才能在正式环境中使用插件功能页。

跳转到功能页

在插件需要登录时，可以在插件的自定义组件中放置一个 `<functional-page-navigator>` 组件。

代码示例：

```
<functional-page-navigator name="loginAndGetUserInfo" args="" version="develop" bind:success="loginSuccess">
  <button>登录到插件</button>
</functional-page-navigator>
```

用户在点击这个区域时，会自动跳转到插件所有者小程序的功能页。功能页会提示用户进行登录或其他相应的操作。操作结果会以组件事件的方式返回。

具体用法和支持的功能页列表详见 组件说明 。

目前，功能页的跳转目前不支持在开发者工具中调试，请在真机上测试。

功能页函数

在使用支付功能页时，插件所有者小程序需要提供一个函数来响应支付请求。这个响应函数应当写在小程序根目录中的 `functional-pages/request-payment.js` 文件中，名为 `beforeRequestPayment` 。如果不提供这段代码，将通过 `fail` 事件返回失败。

注意：功能页函数不应 **`require`** 其他非 **`functional-pages`** 目录中的文件，其他非 **`functional-pages`** 目录中的文件也不应 **`require`** 这个目录中的文件。这样的 **`require`** 调用在未来将不被支持。

代码示例：

```
// functional-pages/request-payment.js
exports.beforeRequestPayment = function(paymentArgs, callback) {
  paymentArgs // 就是 functional-page-navigator 的 args 属性中 paymentArgs

  // 在这里可以执行一些支付前的参数处理逻辑，包括通知后台调用统一下单接口

  // 在 callback 中需要返回两个参数： err 和 requestPaymentArgs
  // err 应为 null （或者一些失败信息）
  // requestPaymentArgs 将被用于调用 wx.requestPayment
  callback(null, {
    // 这里的参数与 wx.requestPayment 相同，除了 success/fail/complete 不被支持
    timeStamp: timeStamp,
    nonceStr: nonceStr,
    package: package,
    signType: signType,
    paySign: paySign,
  })
}
```

这个目录和文件应当被放置在插件所有者小程序代码中（而非插件代码中），它是插件所有者小程序的一部分（而非插件的一部分）。 如果需要新增或更改这段代码，需要发布插件所有者小程序，才能在正式版中生效；需要重新预览插件所有者小程序，才能在开发版中生效。

Bugs & Tips

- Bug：在微信版本 6.6.7 中，功能页被拉起时会触发 App 的部分生命周期并使得功能页启动时间变得比较长。在后续的微信版本中这一行为会发生变更，使 App 生命周期不再被触发。

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/plugin/functional-pages.html*

分包加载

分包加载

> 微信 *6.6* 客户端，*1.7.3* 及以上基础库开始支持，请更新至最新客户端版本，开发者工具请使用 *1.01.1712150* 及以上版本，可点此下载

某些情况下，开发者需要将小程序划分成不同的子包，在构建时打包成不同的分包，用户在使用时按需进行加载。

在构建小程序分包项目时，构建会输出一个或多个功能的分包，其中每个分包小程序必定含有一个主包，所谓的主包，即放置默认启动页面/TabBar 页面，以及一些所有分包都需用到公共资源/JS 脚本，而分包则是根据开发者的配置进行划分。

在小程序启动时，默认会下载主包并启动主包内页面，如果用户需要打开分包内某个页面，客户端会把对应分包下载下来，下载完成后再进行展示。

目前小程序分包大小有以下限制：

- 整个小程序所有分包大小不超过 8M
- 单个分包/主包大小不能超过 2M
  对小程序进行分包，可以优化小程序首次启动的下载时间，以及在多团队共同开发时可以更好的解耦协作。

使用方法

假设支持分包的小程序目录结构如下：

```
├── app.js
├── app.json
├── app.wxss
├── packageA
│   └── pages
│       ├── cat
│       └── dog
├── packageB
│   └── pages
│       ├── apple
│       └── banana
├── pages
│   ├── index
│   └── logs
└── utils
```

开发者通过在 app.json `subPackages` 字段声明项目分包结构：

```
{
  "pages":[
    "pages/index",
    "pages/logs"
```

```
  ],
  "subPackages": [
    {
      "root": "packageA",
      "pages": [
        "pages/cat",
        "pages/dog"
      ]
    }, {
      "root": "packageB",
      "pages": [
        "pages/apple",
        "pages/banana"
      ]
    }
  ]
}
```

打包原则

- 声明 subPackages 后，将按 subPackages 配置路径进行打包，subPackages 配置路径外的目录将被打包到 app（主包） 中
- app（主包）也可以有自己的 pages（即最外层的 pages 字段）
- subPackage 的根目录不能是另外一个 subPackage 内的子目录

- 首页的 TAB 页面必须在 app（主包）内

引用原则

- packageA 无法 require packageB JS 文件，但可以 require app、自己 package 内的 JS 文件

- packageA 无法 import packageB 的 template，但可以 require app、自己 package 内的 template
- packageA 无法使用 packageB 的资源，但可以使用 app、自己 package 内的资源

低版本兼容

由微信后台编译来处理旧版本客户端的兼容，后台会编译两份代码包，一份是分包后代码，另外一份是整包的兼容代码。新客户端用分包，老客户端还是用的整包，完整包会把各个 subpackage 里面的路径放到 pages 中。

示例项目

下载 小程序示例分包加载版源码

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/subpackages.html*

多线程

多线程 Worker

一些异步处理的任务，可以放置于 Worker 中运行，待运行结束后，再把结果返回到小程序主线程。Worker 运行于一个单独的全局上下文与线程中，不能直接调用主线程的方法。Worker 与主线程之间的数据传输，双方使用 Worker.postMessage() 来发送数据，Worker.onMessage() 来接收数据，传输的数据并不是直接共享，而是被复制的。

步骤

在开发者工具中预览效果

1. 配置 Worker 信息

在 app.json 中可配置 Worker 代码放置的目录，目录下的代码将被打包成一个文件：

配置示例：

```
{
  "workers": "workers"
}
```

2. 添加 Worker 代码文件

根据步骤 1 中的配置，在代码目录下新建以下两个入口文件：

```
workers/request/index.js
workers/request/utils.js
workers/response/index.js
```

添加后，目录结构如下：

```
├── app.js
├── app.json
├── project.config.json
└── workers
    ├── request
    |   ├── index.js
    |   └── utils.js
    └── response
        └── index.js
```

3. 编写 Worker 代码

在 workers/request/index.js 编写 Worker 响应代码

```
var utils = require('./utils')

// 在 Worker 线程执行上下文会全局暴露一个 `worker` 对象，直接调用 worker.onMeesage/postMessage 即可
worker.onMessage(function (res) {
  console.log(res)
})
```

4. 在主线程中初始化 Worker

在主线程的代码 app.js 中初始化 Worker

```
var worker = wx.createWorker('workers/request/index.js') // 文件名指定 worker 的入口文件路径，绝对路径
```

5. 主线程向 Worker 发送消息

```
worker.postMessage({
  msg: 'hello worker'
})
```

worker 对象的其它接口请看 worker接口说明

Tips

- Worker 最大并发数量限制为 1 个，创建下一个前请用 Worker.terminate() 结束当前 Worker
- Worker 内代码只能 require 指定 Worker 路径内的文件，无法引用其它路径
- Worker 的入口文件由 wx.createWorker() 时指定，开发者可动态指定 Worker 入口文件
- Worker 内不支持 wx 系列的 API
- Workers 之间不支持发送消息

原文:

> *https://developers.weixin.qq.com/miniprogram/dev/framework/workers.html*

基础库

基础库

基础库与客户端之间的关系

小程序的能力需要微信客户端来支撑，每一个基础库都只能在对应的客户端版本上运行，高版本的基础库无法兼容低版本的微信客户端。

关于基础库的兼容方法，可以查看「兼容处理」章节。

基础库更新时机

为了避免新版本的基础库给线上小程序带来未知的影响，微信客户端都是携带 上一个稳定版 的基础库发布的。

在新版本客户端发布后，再通过后台灰度新版本基础库，灰度时长一般为 12 小时，在灰度结束后，用户设备上才会有新版本的基础库。

以微信 6.5.8 为例，客户端在发布时携带的是 1.1.1 基础库(6.5.7 上已全量的稳定版)发布，在 6.5.8 发布后，我们再通过后台灰度 1.2.0 基础库。

基础库版本分布

全部

iOS

Android

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/client-lib.html*

兼容

兼容

小程序的功能不断的增加，但是旧版本的微信客户端并不支持新功能，所以在使用这些新能力的时候需要做兼容。

文档会在组件，API等页面描述中带上各个功能所支持的版本号。

可以通过 `wx.getSystemInfo` 或者 `wx.getSystemInfoSync` 获取到小程序的基础库版本号。

也可以通过 `wx.canIUse` 详情 来判断是否可以在该基础库版本下直接使用对应的API或者组件

兼容方式 - 版本比较

微信客户端和小程序基础库的版本号风格为 `Major.Minor.Patch`（主版本号.次版本号.修订号）。开发者可以根据版本号去做兼容，以下为参考代码：

```
function compareVersion(v1, v2) {
  v1 = v1.split('.')
  v2 = v2.split('.')
  var len = Math.max(v1.length, v2.length)

  while (v1.length < len) {
    v1.push('0')
  }
  while (v2.length < len) {
    v2.push('0')
  }

  for (var i = 0; i < len; i++) {
    var num1 = parseInt(v1[i])
    var num2 = parseInt(v2[i])

    if (num1 > num2) {
      return 1
    } else if (num1 < num2) {
      return -1
    }
  }

  return 0
}

compareVersion('1.11.0', '1.9.9')
// 1
```

兼容方式 - 接口

对于新增的 API，可以用以下代码来判断是否支持用户的手机。

```
if (wx.openBluetoothAdapter) {
  wx.openBluetoothAdapter()
} else {
  // 如果希望用户在最新版本的客户端上体验您的小程序，可以这样子提示
  wx.showModal({
    title: '提示',
    content: '当前微信版本过低，无法使用该功能，请升级到最新微信版本后重试。'
  })
}
```

兼容方式 - 参数

对于 API 的参数或者返回值有新增的参数，可以判断用以下代码判断。

```
wx.showModal({
  success: function(res) {
    if (wx.canIUse('showModal.cancel')) {
      console.log(res.cancel)
    }
  }
})
```

兼容方式 - 组件

对于组件，新增的组件或属性在旧版本上不会被处理，不过也不会报错。如果特殊场景需要对旧版本做一些降级处理，可以这样子做。

```
Page({
  data: {
    canIUse: wx.canIUse('cover-view')
  }
})
```

```
<video controls="{{!canIUse}}">
  <cover-view wx:if="{{canIUse}}">play</cover-view>
</video>
```

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/compatibility.html*

运行机制

运行机制

小程序启动会有两种情况，一种是「冷启动」，一种是「热启动」。假如用户已经打开过某小程序，然后在一定时间内再次打开该小程序，此时无需重新启动，只需将后台态的小程序切换到前台，这个过程就是热启动；冷启动指的是用户首次打开或小程序被微信主动销毁后再次打开的情况，此时小程序需要重新加载启动。

更新机制

小程序冷启动时如果发现有新版本，将会异步下载新版本的代码包，并同时用客户端本地的包进行启动，即新版本的小程序需要等下一次冷启动才会应用上。如果需要马上应用最新版本，可以使用 `wx.getUpdateManager` API 进行处理。

运行机制

- 小程序没有重启的概念
- 当小程序进入后台，客户端会维持一段时间的运行状态，超过一定时间后（目前是5分钟）会被微信主动销毁
- 当短时间内（5s）连续收到两次以上收到系统内存告警，会进行小程序的销毁

再次打开逻辑

> 基础库 *1.4.0* 开始支持，低版本需做兼容处理

用户打开小程序的预期有以下两类场景：

A. 打开首页： 场景值有 1001, 1019, 1022, 1023, 1038, 1056

B. 打开小程序指定的某个页面： 场景值为除 A 以外的其他

当再次打开一个小程序逻辑如下：

上一次的场景	当前打开的场景	效果
A	A	保留原来的状态
B	A	清空原来的页面栈，打开首页（相当于执行 wx.reLaunch 到首页）
A 或 B	B	清空原来的页面栈，打开指定页面（相当于执行 wx.reLaunch 到指定页）

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/operating-mechanism.html*

性能

性能

目前，我们提供了两种性能分析工具，和几个性能优化上的建议，开发者可以参考使用。

- 分析工具
- 优化建议

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/performance/*

优化建议

setData

`setData` 是小程序开发中使用最频繁的接口，也是最容易引发性能问题的接口。在介绍常见的错误用法前，先简单介绍一下 `setData` 背后的工作原理。

工作原理

小程序的视图层目前使用 WebView 作为渲染载体，而逻辑层是由独立的 JavascriptCore 作为运行环境。在架构上，WebView 和 JavascriptCore 都是独立的模块，并不具备数据直接共享的通道。当前，视图层和逻辑层的数据传输，实际上通过两边提供的 `evaluateJavascript` 所实现。即用户传输的数据，需要将其转换为字符串形式传递，同时把转换后的数据内容拼接成一份 JS 脚本，再通过执行 JS 脚本的形式传递到两边独立环境。

而 `evaluateJavascript` 的执行会受很多方面的影响，数据到达视图层并不是实时的。

常见的 setData 操作错误

1. 频繁的去 setData

在我们分析过的一些案例里，部分小程序会非常频繁（毫秒级）的去 `setData`，其导致了两个后果：

- Android 下用户在滑动时会感觉到卡顿，操作反馈延迟严重，因为 JS 线程一直在编译执行渲染，未能及时将用户操作事件传递到逻辑层，逻辑层亦无法及时将操作处理结果及时传递到视图层；
- 渲染有出现延时，由于 WebView 的 JS 线程一直处于忙碌状态，逻辑层到页面层的通信耗时上升，视图层收到的数据消息时距离发出时间已经过去了几百毫秒，渲染的结果并不实时；

2. 每次 setData 都传递大量新数据

由 `setData` 的底层实现可知，我们的数据传输实际是一次 `evaluateJavascript` 脚本过程，当数据量过大时会增加脚本的编译执行时间，占用 WebView JS 线程，

3. 后台态页面进行 setData

当页面进入后台态（用户不可见），不应该继续去进行 `setData`，后台态页面的渲染用户是无法感受的，另外后台态页面去 `setData` 也会抢占前台页面的执行。

图片资源

目前图片资源的主要性能问题在于大图片和长列表图片上，这两种情况都有可能导致 iOS 客户端内存占用上升，从而触发系统回收小程序页面。

图片对内存的影响

在 iOS 上，小程序的页面是由多个 WKWebView 组成的，在系统内存紧张时，会回收掉一部分 WKWebView。从过去我们分析的案例来看，大图片和长列表图片的使用会引起 WKWebView 的回收。

图片对页面切换的影响

除了内存问题外，大图片也会造成页面切换的卡顿。我们分析过的案例中，有一部分小程序会在页面中引用大图片，在页面后退切换中会出现掉帧卡顿的情况。

当前我们建议开发者尽量减少使用大图片资源。

代码包大小的优化

小程序一开始时代码包限制为 1MB，但我们收到了很多反馈说代码包大小不够用，经过评估后我们放开了这个限制，增加到 2MB 。代码包上限的增加对于开发者来说，能够实现更丰富的功能，但对于用户来说，也增加了下载流量和本地空间的占用。

开发者在实现业务逻辑同时也有必要尽量减少代码包的大小，因为代码包大小直接影响到下载速度，从而影响用户的首次打开体验。除了代码自身的重构优化外，还可以从这两方面着手优化代码大小：

控制代码包内图片资源

小程序代码包经过编译后，会放在微信的 CDN 上供用户下载，CDN 开启了 GZIP 压缩，所以用户下载的是压缩后的 GZIP 包，其大小比代码包原体积会更小。但我们分析数据发现，不同小程序之间的代码包压缩比差异也挺大的，部分可以达到 30%，而部分只有 80%，而造成这部分差异的一个原因，就是图片资源的使用。GZIP 对基于文本资源的压缩效果最好，在压缩较大文件时往往可高达 70%-80% 的压缩率，而如果对已经压缩的资源（例如大多数的图片格式）则效果甚微。

及时清理没有使用到的代码和资源

在日常开发的时候，我们可能引入了一些新的库文件，而过了一段时间后，由于各种原因又不再使用这个库了，我们常常会只是去掉了代码里的引用，而忘记删掉这类库文件了。目前小程序打包是会将工程下所有文件都打入代码包内，也就是说，这些没有被实际使用到的库文件和资源也会被打入到代码包里，从而影响到整体代码包的大小。

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/performance/tips.html*

分析工具

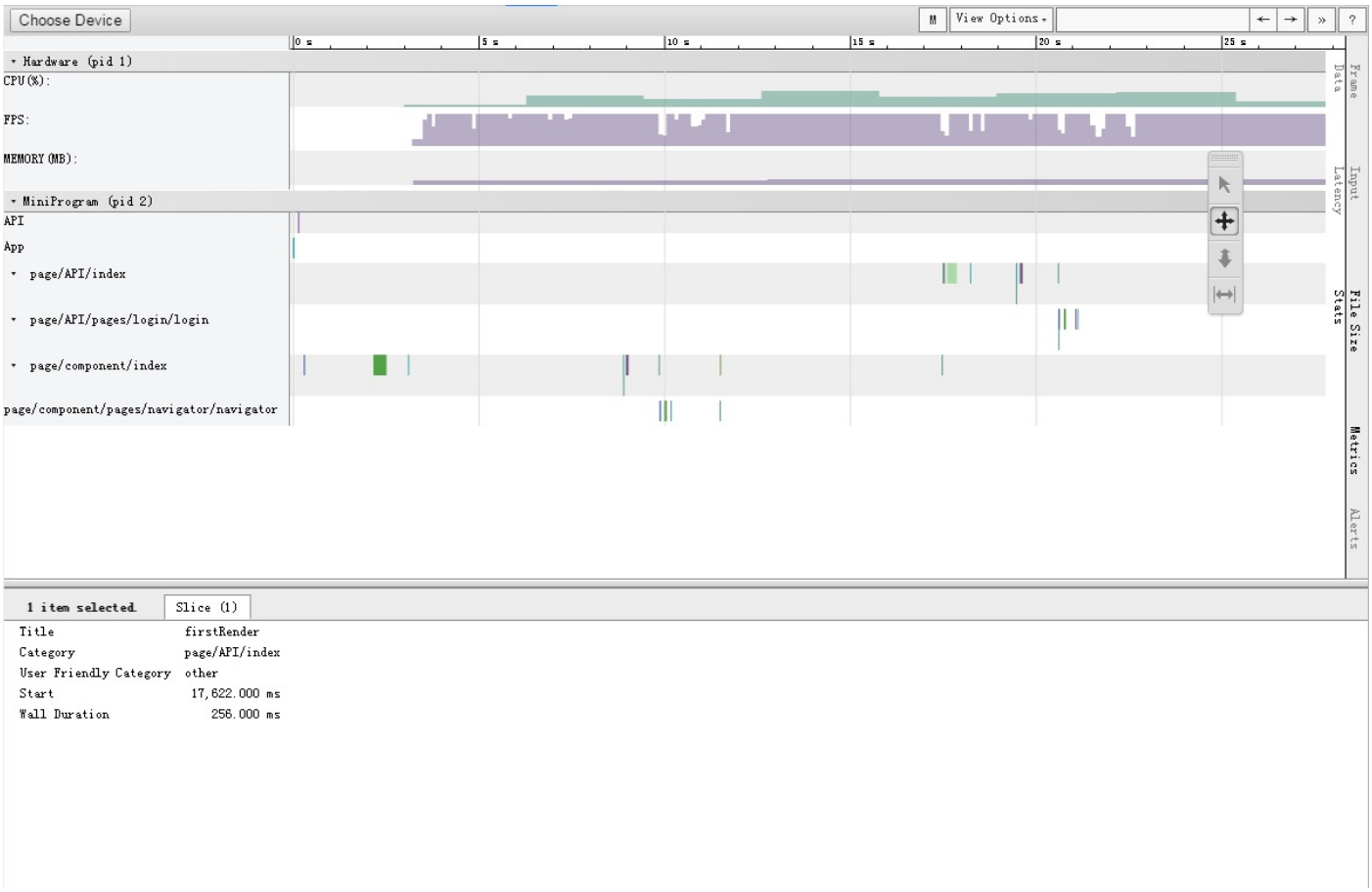
性能 Trace 工具

微信 Andoid 6.5.10 开始，我们提供了 Trace 导出工具，开发者可以在开发者工具 Trace Panel 中使用该功能。

使用方法

- PC 上需要先安装 adb 工具，可以参考一些主流教程进行安装，Mac 上可使用 brew 直接安装。
- 确定 adb 工具已成功安装后，在开发者工具上打开 Trace Panel，将 Android 手机通过 USB 连接上 PC，点击「Choose Devices」，此时手机上可能弹出连接授权框，请点击「允许」。
- 选择设备后，在手机上打开你需要调试的开发版小程序，通过右上角菜单，打开性能监控面板，重启小程序；
- 重启后，在小程序上进行操作，完成操作后，通过右上角菜单，导出 Trace 数据；
- 此时开发者工具 Trace Panel 上会自动拉取 Trace 文件，选择你要分析的 Trace 文件即可；

> 可以通过 `adb devices` 命令确定设备是否已和 PC 建立起连接



性能面板

从微信 6.5.8 开始，我们提供了性能面板让开发者了解小程序的性能。开发者可以在开发版小程序下打开性能面

板，打开方法：进入开发版小程序，进入右上角更多按钮，点击「显示性能窗口」。


中国移动
下午8:25
44%
小程序官方组件展示
性能数据
启动和切换
切页面耗时:0 ms
启动耗时:100 ms
包下载耗时:427 ms
渲染
初次渲染耗时:19 ms
再次渲染耗时:6 ms
帧率:60 fps
其他
数据缓存:0 B
视图容器
基础内容
表单组件
导航
媒体组件
组件
接口

性能面板指标说明

指标	说明
CPU	小程序进程的 CPU 占用率，仅 Android 下提供
内存	小程序进程的内存占用（Total Pss），仅 Android 下提供
启动耗时	小程序启动总耗时
下载耗时	小程序包下载耗时，首次打开或资源包需更新时会进行下载
页面切换耗时	小程序页面切换的耗时
帧率/FPS	
首次渲染耗时	页面首次渲染的耗时
再次渲染耗时	页面再次渲染的耗时（通常由开发者的 setData 操作触发）
数据缓存	小程序通过 Storage 接口储存的缓存大小

原文：

> *https://developers.weixin.qq.com/miniprogram/dev/framework/performance/tools.html*