WeChat-小程序

指令

wx:for-item

wx:for-index

block wx:for

`<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。

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

`*this`

保留关键字 `*this` 代表在 for 循环中的 item 本身

block wx:if

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

template

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

事件

data-hi="WeChat"

> 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`。

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

当点击控件的时候，事件对象中包含：

```
"target": {
    "id": "tapTest",
    "dataset":  {
      "hi":"WeChat"
    }
  },
  "currentTarget":  {
    "id": "tapTest",
    "dataset": {
      "hi":"WeChat"
    }
  }
```

冒泡事件列表 - 传送门

- `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

捕获阶段 - 传送门

在捕获阶段中，事件到达节点的顺序与冒泡阶段恰好相反

```
<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>
```

事件对象

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

wxml

引用

WXML 提供两种文件引用方式`import`和`include`

import

```
<import src="item.wxml"/>
```

- 继承性

> 只会 import 目标文件中定义的 template，而不会 import 目标文件 import 的 template

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：

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

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

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

wxss

尺寸单位 - rpx

规定屏幕宽为750rpx，即：屏幕宽度被平分750等分

- eg：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

外联样式表

```
/** app.wxss **/
@import "common.wxss";
.middle-p {
  padding:15px;
}
```

内联样式

style

> 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

class

存疑：

> 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="class-name" />
```

选择器

- **::after**

  eg：`view::after` - 在 view 组件后边插入内容

- **::before**

  类似 `::after`

组件

所有组件与属性都是小写

公共属性

hidden/data-/*bind* / catch*

- *<text/>** 组件内只支持 `<text/>` 嵌套。
- 除了文本节点以外的其他节点都无法长按选中

自定义组件

- 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
- 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

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

Component

```
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(){}
  }
})
```

slot

自定义组件默认只能包含一个slot，如果想定义多个slot，需要在js文件中做以下设置：

```
options: {
   multipleSlots: true // 在组件定义时的选项中启用多slot支持
 }
```

多个slot使用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>
```

样式

组件对应 `wxss` 文件的样式，只对组件wxml内的节点生效，但是需要注意：

- 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；

`:host` 选择器 - 组件可以指定它所在节点的默认样式：

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

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

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：

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

```
/* 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>
```

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

开放全局样式类：

> 当开放了全局样式类，存在外部样式污染组件样式的风险，请谨慎选择

```
/* 组件 custom-component.js */
Component({
  options: {
    addGlobalClass: true,
  }
})
```

behaviors

> 每个组件可以引用多个 `behavior` 。 `behavior` 也可以引用其他 `behavior`

构造器

`Behavior()`

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

使用：`behaviors: [myBehavior]`

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

字段的覆盖和组合规则

- 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
  - 数据是对象类型，会进行对象合并
  - 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

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

自动为组件添加2个字段：

- name

  在表单中的字段名

- value

  在表单中的字段值

wx://component-export

使自定义组件支持 `export` 定义段。这个定义段可以用于指定组件被 `selectComponent` 调用时的返回值

> 未使用这个定义段时， `selectComponent` 将返回自定义组件的 `this` （插件的自定义组件将返回 `null` ）。使用这个定义段时，将以这个定义段的函数返回值代替

```
// 自定义组件 my-component 内部
Component({
  behaviors: ['wx://component-export'],
  export() {
    return { myField: 'myValue' }
```

```
  }
})
```

使用：

```
<!-- 使用自定义组件时 -->
<my-component id="the-id" />
```

```
this.selectComponent('#the-id') // 等于 { myField: 'myValue' }
```

relations

处理复杂的组件关系，多复杂呐？像这样：

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

`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。

```
// 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

抽象节点

```
<!-- selectable-group.wxml -->
```

```
<view wx:for="{{labels}}">
  <label>
    <selectable disabled="{{false}}"></selectable>
    {{item}}
  </label>
</view>
```

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

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

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

> **注意**：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

默认组件

当具体组件未被指定时，将创建默认组件的实例：

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

自定义组件扩展 - 传送门

`Behavior()` 构造器提供了新的定义段 `definitionFilter` ，用于支持自定义组件扩展：

- definitionFilter包含2个参数：
  - 第一个参数是使用该 behavior 的 component/behavior 的定义对象
  - 第二个参数是该 behavior 所使用的 behavior 的 `definitionFilter` 函数列表

```
definitionFilter(defFields, definitionFilterArr) {},
```

例子：

```
// behavior.js
module.exports = Behavior({
  definitionFilter(defFields) {
    defFields.data.from = 'behavior'
  },
})

// component.js
Component({
  data: {
    from: 'component'
  },
  behaviors: [require('behavior.js')],
  ready() {
    console.log(this.data.from) // 此处会发现输出 behavior 而不是
component
  }
})
```

WXS

Careful

- 不依赖于运行时的基础库版本，可在所有版本的小程序中运行
- 有自己的语法，并不和 javascript 一致
- 和其他 javascript 代码是隔离的（包括：其他javascript文件、小程序提供的API）
- wxs 函数不能作为组件的事件回调
- 运行环境的差异：

- iOS中，wxs 会比 javascript 代码快 2 ~ 20 倍
- android中，滚粗，没你啥事（运行效率无差异）

模块

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

- 每一个 `.wxs` 文件和 `<wxs>` 标签都是一个单独的模块
- 只能通过 `module.exports` 暴露变量
- 可通过 `require` 引入 `.wxs` 文件，但是必须注意：
  - 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
- 标签引入：

```
<wxs src='common.wxs'>
```

  仅当本标签为**单闭合标签**或**标签的内容为空**时有效;

- **<wxs> - module**

  表示：`<wxs>` 标签的模块名。必填字段

  - 如果重复：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。
- `<template>` 标签中，只能使用定义该 `<template>` 的 WXML 文件中定义的 `<wxs>` 模块

数据类型

正则

语法：`getRegExp(pattern[, flags])`

- flags：修饰符。该字段只能包含以下字符:
  - `g` : global
  - `i` : ignoreCase
  - `m` : multiline。

```
var a = getRegExp("x", "img");
```

基础类库

JSON

- `JSON.stringify()`
- `JSON.parse(string)`

插件

引用插件中的组件

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

```
{
  "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：

- 默认情况下，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

跳转到插件的页面

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

js接口的调用

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

myPluginInterface.hello();
var myWorld = myPluginInterface.world;
```

分包

subPackages

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

- 首页的 TAB 页面必须在 app（主包）内

多线程Worker

文件系统

全局唯一

```
var fs = wx.getFileSystemManager()
```

用户文件目录

微信提供了一个用户文件目录给开发者，开发者对这个目录有完全自由的读写权限。通过 `wx.env.USER_DATA_PATH` 可以获取到这个目录的路径

```
// 在本地用户文件目录下创建一个文件 a.txt，写入内容 "hello, world"
const fs = wx.getFileSystemManager()
fs.writeFileSync(`${wx.env.USER_DATA_PATH}/hello.txt`, 'hello, world',
'utf8')
```

优化

`wx:if` VS `hidden`

1. 因为 `wx:if` 之中的模板也可能包含数据绑定，所以当 `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。
2. `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

总结：`wx:if` 有更高的切换消耗而 `hidden` 有更高的初始渲染消耗

官方计算属性 - computed

版本兼容

获取基础库版本

`wx.getSystemInfoSync` / `wx.getSystemInfo`

检查版本支持

`wx.canIUse(String)`

版本比较

版本号风格为 Major.Minor.Patch（主版本号.次版本号.修订号）

```
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
```

判断用户手机时否支持某api

```
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。'
  })
}
```

setData性能优化

1. 避免高频调用setData

2. 避免每次传输大量数据
3. 避免向后台页面执行setData

大图性能

1. 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
2. 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。

扩展

js基础库

- **Object.keys(obj)**
- 字符串中填充变量值：

```
console.log('name=${name}')
```

- JavaScript 之 Object.apply()与Object.call()和Object.bind()
-