wx.showLoading()

# 文件结构

创建一个微信小程序

pages

index

index.js

index.json

index.wxss

index.wxml

logs

logs.js

logs.json

logs.wxss

logs.wxml

utils

util.js

app.js

app.json

app.wxss

project.config.json

sitemap.json

微信客户端给小程序所提供的环境为**宿主环境**

小程序的运行环境分成渲染层和逻辑层，WXML和WXSS工作在**渲染层**，JS脚本工作在**逻辑层**。

小程序的渲染层和逻辑层分别由2个线程管理：渲染层使用**WebView** 进行渲染；逻辑层采用**JsCore**线程运行JS脚本。

一个小程序存在多个界面，所以渲染层存在多个WebView线程，这两个线程的通信经由微信客户端（**Native**）做中转，逻辑层发送网络请求也经由Native转发

## app.js

App() 构造器, 是单例对象，在其他JS脚本中可以使用getApp() 来获取实例。

小程序启动之后，onLaunch回调执行

App({

**onLaunch**: function () {}

})

## app.json

**app.json**是当前小程序的**全局配置**

{

"pages":[

"pages/index/index",

"pages/logs/logs"

],

"window":{

"backgroundTextStyle":"light",

"navigationBarBackgroundColor": "#fff",

"navigationBarTitleText": "Weixin",

"navigationBarTextStyle":"black"

},

"style": "v2",

"sitemapLocation": "sitemap.json"

}

**pages**描述所有页面路径

写在 pages 字段的第一个页面就是这个小程序的首页。

**window**全局的默认窗口表现

navigationBarTitleText 导航栏标题文字内容

backgroundTextStyle 下拉 loading 的样式，dark / light

navigationBarBackgroundColor 导航栏背景颜色，#000000

navigationBarTextStyle 导航栏标题颜色，black / white

## project.config.json

开发者工具配置

微信小程序工具的个性化配置，用于换电脑时不用重新修改设置

比如编辑器的颜色、代码上传时自动压缩

## page.json

页面配置

可以在app.json里边声明小程序顶部颜色是蓝色，在index.json文件为index页面独立定义顶部颜色为红色

# WXML

WeiXin Markup Language微信标记语言

<**text**>你好</text>

在app.js中pages属性添加"pages/newpage/newpage"就会在文件目录自动生成newpage目录和newpage.js, newpage.json, newpage,wxml, newpage.wxss4个文本文件

<**image** src="./image/a.png"></image>

<text>你好{{time}}</text>

.js文件

Page({

**data**: {

time: (new Date()).toString()

},

})

**数据绑定**

支持{{"hello " +time}}

{{ }}中也可以直接放置数组

## wx:if

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

<view wx:if="{{length > 5}}"> 1 </view>

<view wx:**elif**="{{length > 2}}"> 2 </view>

<view wx:**else**> 3 </view>

如果要一次性判断多个组件标签，可以使用<block/>标签将多个组件包装起来，并使用wx:if控制属性。

<**block** wx:if="{{true}}">

<view> view1 </view>

<view> view2 </view>

</block>

## wx:for

<view **wx:for**="{{array}}">

{{**index**}}: {{**item**}}

</view>

data: {

array: ['foo', 'bar']

}

不是，这样的机制就断绝了双重循环的可能性了啊。当然不会

wx:for-item 指定数组元素的变量名，wx:for-index指定下标的变量名：

<view wx:for="{{array}}" wx:**for-index**="idx" wx:**for-item**="itemName">

{{idx}}: {{itemName}}

</view>

类似block wx:if，也可以将wx:for用在<block/>标签上，以渲染包含多节点的结构块

<block wx:for="{{[1, 2, 3]}}">

<view> {{index}}: </view>

<view> {{item}} </view>

</block>

如果item的位置会动态改变，或者有新的item添加到列表中，需要使用 wx:key 来指定item的唯一的标识符。

wx:key的值以两种形式提供：

* 字符串。item 的某个property，需要是列表中唯一的字符串或数字，且不能动态改变。
* this。item 本身是一个唯一的字符串或者数字时使用

当数据改变时，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，以提高列表渲染时的效率。

<**switch** wx:for="{{objectArray}}" **wx:key**="unique" > {{item.id}} </switch>

objectArray: [

{id: 1, unique: 'unique\_1'},

{id: 0, unique: 'unique\_0'},

],

key里没有{{}}

switch是一个单选按钮

<switch wx:for="{{numberArray}}" wx:key="**\*this**" > {{item}} </switch>

## 模板

定义模板

<**template** **name**="msgItem">

<view>

<text> {{index}}: {{msg}} </text>

<text> Time: {{time}} </text>

</view>

</template>

使用模板

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

使用模板时需要为模板中的引用的变量提供参数，注意{{}}符号和…符号，还有引号。。。

item: {

index: 0,

msg: 'this is a template',

time: '2016-06-18'

}

可以动态切换模板

<template is="{{item % 2 == 0 ? 'even' : 'odd'}}"/>

## 引用

WXML 提供两种文件引用方式import和include。

import可以使用目标文件定义的 template

item.wxml

<template name="item">

<text>{{text}}</text>

</template>

index.wxml

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

<template is="item" data="{{text: 'forbar'}}"/>

import 不具有递归的特性，不会 import 目标文件中import的template

include将目标文件整个代码引入，相当于是拷贝到 include 位置

index.wxml

<view> body </view>

<**include** src="header.wxml"/>

header.wxml

<view> header </view>

# WXSS

WeiXin Style Sheets 微信样式表

* 项目公共样式：根目录中的app.wxss
* 页面样式：与页面同名同级的WXSS文件
* 其它样式：可以被项目公共样式和页面样式引用

**@import** './test\_0.wxss'

WXSS引入了rpx（responsive pixel响应式像素）尺寸单位，适配不同宽度的屏幕

在一个宽度为375**物理像素**的屏幕下，1rpx = 1px。

以此类推，在宽度为750物理像素屏幕下，1rpx=0.5px。。。不太懂

内联样式

<view **style=**"color: red; font-size: 48rpx"></view>

支持动态更新内联样式：

{

eleColor: 'red',

eleFontsize: '48rpx'

}

<view style="color: {{eleColor}}; font-size: {{eleFontsize}}"></view>

## Flex布局

默认情况下，**主轴**（main axis）水平方向向右，**交叉轴**（cross axis）垂直方向向下

项目在主轴上排列，排满后在交叉轴方向换行。

flex**容器**包含着**项目**

### 容器属性

.container{

display:**flex**;

}

设置项目排列方向

.container{

**flex-direction**: row

}

row（默认值）：主轴向右。项目沿主轴从左到右排列。

row-reverse：主轴左。项目沿主轴从右到左排列。

column：主轴纵向，方向从上向下。项目沿主轴从上到下排列。

column-reverse：主轴纵向，方向从下向上。项目沿主轴从下到上排列。

**flex-wrap**:nowrap

nowrap（默认值）：不换行。如果单行内容过多，则溢出容器。

wrap：容器单行容不下所有项目时，换行排列。

wrap-reverse：换行排列。换行方向为wrap时的反方向。

设置项目在主轴方向上**对齐方式**

**justify-content**: flex-start

* flex-start（默认值）：项目间不留空隙。
* center：项目在主轴上居中排列，项目间不留空隙
* flex-end：项目对齐主轴终点，项目间不留空隙。
* space-between：项目间间距相等，首项目离起点距离，尾项目离终点距离为0。
* space-around：项目间间距相等，首项目离主轴起点距离和尾项目离终点距离为中间项目间间距的一半。
* space-evenly：项目间间距、首项目离主轴起点和尾项目离终点距等于项目间间距。

设置项目在交叉轴方向上每行的对齐方式

**align-items**:stretch

* stretch（默认值）：项目拉伸至填满行高。
* flex-start：项目顶部与行起点对齐。
* center：项目在行中居中对齐。
* flex-end：项目底部与行终点对齐。
* baseline：项目的第一行文字的基线对齐？？？？

多行排列时，设置行在交叉轴方向上的对齐方式

感觉是竖直方向的justify-content

**align-content**: stretch

* stretch（默认值）：当未设置项目尺寸，将各行中的项目拉伸至填满交叉轴。当设置了项目尺寸，项目尺寸不变，项目行拉伸至填满交叉轴。总之就是填满交叉轴
* flex-start：交叉轴起点开始排列，行间不留间距。
* center：行在交叉轴**中点**排列，行间不留间距
* flex-end：尾行在交叉轴终点开始排列，行间不留间距。
* space-between：行与行间距相等，首行离交叉轴起点和尾行离终点距离为0。
* space-around：行与行间距相等，首行离交叉轴起点和尾行离终点距离为行间距的一半
* space-evenly：行间间距、以及首行离交叉轴起点和尾行离终点距离相等。

用这个属性时，估计就是设置了容器的尺寸，才能使行与行之间有间距。自行撑开应该不会有间距的

### 项目属性

沿主轴方向上的排列顺序，数值越小，排列越靠前。

整数

.item{

**order**: 0

}

这需要给每个item都单独设置order属性，所以应该是在js中完成

当项目在主轴方向上溢出时，通过设置项目**收缩因子**来压缩项目适应容器。

**flex-shrink**: 3

未设置flex-shrink，默认值为1

最好同时规定width

一个宽度为400px的容器，里面的三个项目width分别为120px，150px，180px。分别对这项目1和项目2设置flex-shrink值为2和3。

项目溢出 400 - (120 + 150 + 180) = -50px。计算压缩量时总权重为各个项目的**宽度乘以flex-shrink的总和**，这个例子压缩总权重为120 \* 2 + 150 \* 3+ 180 \* 1 = 870。各个项目压缩空间大小为总溢出空间乘以项目宽度乘以flex-shrink除以总权重：

item1的最终宽度为：120 - 50 \* 120 \* 2 / 870 ≈ 106px

item2的最终宽度为：150 - 50 \* 150 \* 3 / 870 ≈ 124px

item3的最终宽度为：180 - 50 \* 180 \* 1 / 870 ≈ 169px

其中计算时候值如果为小数，则向下取整。

默认都为1时，每个项目需要压缩本身占项目宽度总和的比例，更宽的项目压缩的宽度相比于自身的比例更大。当然压缩的宽度更大。

当项目在主轴方向上还有剩余空间时，通过设置项目**扩张因子**进行剩余空间的分配。

flex-grow: 3

默认0

原理和上面那个差不多

当容器设置flex-direction为row或row-reverse时，flex-basis和width同时存在，flex-basis优先级高于width，也就是此时flex-basis代替项目的width属性。

当容器设置flex-direction为column或column-reverse时，flex-basis和height同时存在，flex-basis优先级高于height，也就是此时flex-basis代替项目的height属性。

当flex-basis和width（或height），其中一个属性值为auto时，非auto的优先级更高。

**flex-basis**: auto（默认值） | <number>px

flex是flex-grow，flex-shrink，flex-basis的简写方式。值设置为none，等价于00 auto。值设置为auto，等价于1 1 auto。

flex: none | auto | @flex-grow @flex-shrink@flex-basis

覆盖容器的align-items，可以对项目的对齐方式做特殊处理。

默认属性值为auto，继承容器的align-items值

当容器没有设置align-items时，属性值为stretch

**align-self**: auto（默认值） | flex-start | center | flex-end | baseline |stretch

# JS

* 项目默认勾选ES6转ES5，这样我们就可以使用箭头函数和let/const了嘻嘻嘻
* 所有页面的脚本逻辑都跑在同一个JsCore线程，页面使用setTimeout或者setInterval的定时器，然后跳转到其他页面时，这些定时器并没有被清除，需要开发者自己在页面离开的时候进行清理。

模块化

可以将任何一个JavaScript 文件作为一个模块，通过module.exports 或者 exports 对外暴露接口。

B.js引用模块A

moduleA.js

**module.exports** = function( value ){

return value \* 2;

}

B.js

var multiplyBy2 = **require**('./moduleA')

var result = multiplyBy2(4)

当需要使用**全局变量**时，通过使用全局函数 getApp() 获取全局的实例，并设置相关属性值，来达到设置全局变量的目的

a.js

var global = **getApp()**

global.globalValue = 'globalValue'

b.js

var global = getApp()

console.log(global.globalValue)

只有在 a.js 比 b.js 先执行才有效

在app.js中设置，以保证全局的数据可以在任何文件中被使用到

App({

globalData: {

nihao: "111"

}

})

引用

console.log(getApp().globalData.nihao)

# 生命周期

app.js中可以规定的函数：

onLaunch 初始化完成时触发，全局只触发一次

onShow 启动，或从后台进入前台显示时触发

onHide 从前台进入后台时触发

onError 发生脚本错误，或者 API 调用失败时时触发。并带上错误信息

其他字段 　 可以添加任意的函数或数据到 Object 参数中，在App实例回调用 this 可以访问

App({

onLaunch: function (**options**) {

console.log(options.path);

}

})

onLaunch和onShow可以带参数

options有以下字段：

* path 页面路径
* query 页面参数query
* scene 场景值
* referrerInfo 特定场景下存在
* referrerInfo.appId 来源小程序或公众号或App的 appId

场景值 场景 appId

1020 公众号 profile 来源公众号 appId

1035 公众号自定义菜单 来源公众号 appId

1036 App 分享消息卡片 来源应用 appId

1037 小程序打开小程序 来源小程序 appId

1038 从另一个小程序返回 来源小程序 appId

1043 公众号模板消息 来源公众号 appId

# 页面

## 生命周期

页面构造器Page()

Page构造器接受一个Object参数

**Page**({

data: { text: "This is page data." },

onLoad: function(options) { },

})

* 生命周期函数

onLoad

onShow

onReady

onHide 监听页面隐藏

onUnload 监听页面卸载

onLoad在页面没被销毁之前只会触发1次

onReady在页面没被销毁前只会触发1次，onReady触发时，表示页面已经准备妥当，在逻辑层就可以和视图层进行交互了。

wx.navigateTo切换到其他页面、底部tab切换时触发onHide。

wx.redirectTo或wx.navigateBack返回到其他页，当前页面会被销毁，触发onUnload

* 监听页面的用户行为

onPullDownRefresh 监听下拉刷新事件

onReachBottom 监听触底事件

onShareAppMessage 监听点击右上角转发事件

onPageScroll 监听页面滚动事件

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page.json中设置

"window":{

"**enablePullDownRefresh**": true

},

wx.stopPullDownRefresh可以停止当前页面的下拉刷新。

onPullDownRefresh: function () {

setTimeout(() => {

**wx.stopPullDownRefresh()**

}, 500);

},

onReachBottom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page.json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。

页面滚动 onPageScroll

onPageScroll: function (option) {

console.log(option.**scrollTop**)

}

scrollTop字段表示页面在垂直方向已滚动的距离，单位px

只有定义了onShareAppMessage事件处理函数，右上角菜单才会显示“转发”按钮

此事件需要return一个Object，包含title和path两个字段，用于自定义转发内容

onShareAppMessage: function () {

return {

**title**: '自定义转发标题',

**path**: '/page/user?id=123'

}

}

## 路由参数

wx**.navigateTo**({ url: 'pages/detail/detail**?**id**=**1**&**other=abc' })

onLoad: function(option) {

console.log(option.id)

}

使用英文 ? 分隔path和query部分，query部分的多个参数使用 & 进行分隔

onLoad可以拿到当前页面的打开参数

URL上的value如果涉及特殊字符（例如：&字符、?字符、中文字符等），需要采用UrlEncode后再拼接到页面URL上。

## data

用this.setData把数据传递给渲染层，从而更新界面。

setData的第二个参数是一个callback回调，在setData对界面渲染完毕后触发。

onLoad: function(){

**this.setData**({

text: 'change data'

}, function(){})

}

不需要每次都将整个data字段重新设置一遍，只需要把改变的值进行设置即可

data中的key非常灵活，可以以**数据路径**的形式给出。如果data的属性是一个数组，则

Page({

data: {

d: [1, {text: 'Hello'}, 3, 4]

}

onLoad: function(){

this.setData({**"d[0]"**: 100})

}

})

## 页面栈

pageA通过wx.navigateTo推入一个新页面pageB，再次使用wx.navigateTo推入pageC，这时页面层级有三层，这样的一个页面层级称为页面栈，记作[pageA, pageB, pageC]

**wx.navigateBack**() 退出当前页面栈的顶页面，此时页面栈变成 [ pageA, pageB]

**wx.redirectTo**({ url: 'pageE' })替换当前页变成pageE，此时页面栈变成 [ pageA, pageE ]

app.json定义小程序底部tab

{

"**tabBar**": {

"list": [

{ "text": "Tab1", "pagePath": "pageA" },

{ "text": "Tab1", "pagePath": "pageF" },

{ "text": "Tab1", "pagePath": "pageG" }

]

}

}

在刚刚的例子所在的页面栈中使用wx.switchTab({ url: 'pageF' })，此时原来的页面栈会被清空为，然后会切到pageF所在的tab页面，页面栈变成 [ pageF ]，此时点击Tab1切回到pageA时，pageA不会再触发onLoad，因为pageA没有被销毁。

wx.navigateTo和wx.redirectTo只能打开非TabBar页面，wx.switchTab只能打开Tabbar页面。

可以使用wx. reLaunch({ url: 'pageH' }) 重启小程序，并且打开pageH，此时页面栈为 [ pageH ]。

Tabbar页面初始化之后不会被销毁。

# 组件

* 视图容器

cover-image

cover-view

match-media

movable-area

movable-view

scroll-view

swiper

swiper-item

view

* 基本

icon 图标

progress 进度条

rich-text 富文本

text 文本

* 表单组件

button

checkbox

checkbox-group

editor

form

input

label

picker

picker-view

picker-view-column

radio

radio-group

slider

switch

textarea

* 导航

functional-page-navigator

navigator

* 媒体组件

audio

camera

image

live-player

live-pusher

video

voip-room

等

# 事件

<view **bindtap**="tapName"> Click me! </view>

Page({

tapName: function(**event**) {

console.log(event)

}

})

* 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

event对象的属性

* type 事件类型
* timeStamp 页面打开到触发事件所经过的毫秒数
* target 触发事件的组件的一些属性值集合
* currentTarget 当前组件的一些属性值集合
* detail 点击位置的横纵坐标
* touches 触摸事件，当前停留在屏幕中的触摸点信息的数组
* changedTouches 触摸事件，当前变化的触摸点信息的数组

currentTarget为当前事件所绑定的父组件，而target则是触发该事件的源头子组件。

<view id="outer" catchtap="handleTap">

<view id="inner">点击我</view>

</view>

点击inner节点时，evt.target 是inner view组件，evt.currentTarget 是绑定了handleTap的outer view组件

touch和changedTouches对象的详细参数

identifier 触摸点的标识符

pageX, pageY 距离文档左上角的距离，文档的左上角为原点

clientX, clientY 距离页面可显示区域左上角距离

## 冒泡/捕获

事件绑定的形式为key="value"

key形式为"bind/catch+时间类型"

**catch**事件绑定终止捕获，阻止冒泡事件向上冒泡。也就是阻止事件传递

默认是冒泡阶段触发事件。capture-来表示捕获阶段触发事件。

先捕获后冒泡

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。

点击 inner view 会先后调用handleTap2、handleTap4、handleTap3、handleTap1。

<view bind:touchstart="handleTap1" **capture-bind:**touchstart="handleTap2">

<view 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

# 兼容

可以使用 wx.getSystemInfo 或者 wx.getSystemInfoSync 来获取手机品牌、操作系统版本号、微信版本号以及小程序基础库版本号等，通过这个信息针对不同平台做差异化的服务。

wx.getSystemInfoSync()

{

brand: "iPhone", // 手机品牌

model: "iPhone 6", // 手机型号

platform: "ios", // 客户端平台

system: "iOS 9.3.4", // 操作系统版本

version: "6.5.23", // 微信版本号

SDKVersion: "1.7.0", // 小程序基础库版本

language: "zh\_CN", // 微信设置的语言

pixelRatio: 2, // 设备像素比

screenWidth: 667, // 屏幕宽度

screenHeight: 375, // 屏幕高度

windowWidth: 667, // 可使用窗口宽度

windowHeight: 375, // 可使用窗口高度

fontSizeSetting: 16 // 用户字体大小设置

}

# 交互反馈

小程序的view容器组件和button组件提供了hover-class属性，触摸时会往该组件加上对应的class改变组件的样式。

.hover{

background-color: gray;

}

<button **hover-class**="hover"> 点击button </button>

<view hover-class="hover"> 点击view</view>

显示按钮loading

<button **loading**="{{loading}}" bindtap="tap">操作</button>

data: { loading: false },

tap: function() {

this.setData({

loading: true

})

弹出式提示**Toast**

默认1.5秒后自动消失

**wx.showToast**({

title: '已发送',

icon: '**success**',

duration: 1500

})

// wx.**hideToast**() // 隐藏Toast

一般如果用户需要明确知晓操作结果状态的话，使用**模态对话框**来提示，同时附带下一步操作的指引。toast一闪而过，不适合错误处理

**wx.showModal**({

title: '标题',

content: '告知当前状态，信息和解决方法',

confirmText: '主操作',

cancelText: '次要操作',

success: function(res) {

if (**res.confirm**) {

console.log('用户点击主操作')

} else if (res.**cancel**) {

console.log('用户点击次要操作')

}

}

})

**scroll-view**可滚动视图组件可以使页面中某一小块区域滚动。

# HTTPS网络通信

**wx.request**({

**url**: 'https://test.com/getinfo',

**success**: function(res) {

console.log(res)

}

})

参数的属性：

url 唯一的必填值

data 请求参数

header 默认{'content-type': 'application/json'}

method 默认'GET'，有效值：OPTIONS, GET, HEAD, POST, PUT, DELETE等

dataType String 默认'json'，对返回的数据进行一次JSON解析

success 调用成功的回调函数

fail 调用失败的回调函数

complete 调用结束的回调函数（调用成功、失败都会执行）

两种方法传GET方法参数

url:'https://test.com/getinfo**?**id**=**1**&**version=1.0.0',

或

data: { id:1, version:'1.0.0' },

唯一区别是URL有长度限制，推荐使用data属性传递。

url中的域名只支持https和wss (wx.connectSocket)协议，不能使用 IP 地址或localhost；

在开发时可以勾选本地设置——不校验合法域名、web-view（业务域名）、TLS版本以及HTTPS证书，允许向localhost发送请求

success()方法接收一个res参数，有data, statusCode, header属性，得到服务器返回的数据、HTTP状态码和响应头

app.json文件指定request的超时时间。

"**networkTimeout**": {

"request": 3000

}

# 微信登录

通常我认为的过程：小程序调用wx.login()🡪微信服务器返回用户信息

实际上的过程：小程序调用wx.login()🡪小程序收到临时令牌code🡪小程序将code发送给开发者服务器🡪开发者服务器将code、AppID、Secret发送给微信服务器🡪开发者服务器收到用户信息。

以上过程是第一次登录。第一次登陆在开发者服务器生成SessionID，第二次及以后的登录就可以不需要通过微信服务器中转，直接通过SessionID和开发者服务器交互即可。

wx.**login**({

success: function(res) {

if (res.**code**) {

console.log(res.code)

} else {

console.log('获取用户登录态失败！' + res.errMsg)

}

}

});

wx.login()给微信服务器发送请求获取code

code形如"013iEgll2B4N8643Elml2fgQIh4iEglU"，是一个有效期5分钟的令牌

wx.request({

url: 'https://api.weixin.qq.com/sns/jscode2session',

data: {

appid: 'wxb7d981af1e7ea708',

secret: 'edaf85a1ad223379de0cc6c5d1d33929',

js\_code: res.code

},

success: function(loginRes){

console.log(loginRes)

}

})

* 然后通过appid、secret和code获取用户信息
* 所谓的用户信息也只有一个用户在微信的唯一标识
* 这一步应该在开发者服务器端完成，这里为了演示用小程序的request方法演示
* loginRes.data包含以下属性

openid: "ofhUZ4\_nXLGjvx9ViiTi5eHA5hCI"

session\_key: "TZ9OziqWB0R3NDUZ+z6Trg=="

* openid 微信用户的唯一标识
* session\_key 会话密钥
* unionid 用户在微信开放平台的唯一标识符。本字段在满足一定条件的情况下才返回。

# 本地数据缓存

wx.getStorage/wx.getStorageSync读取本地缓存， wx.setStorage/wx.setStorageSync写数据到缓存，其中Sync后缀的接口表示是同步接口

wx.getStorage({

key: 'key1',

success: function(res) {

console.log(res.data)

}

})

try{

var value2 = wx.getStorageSync('key2')

}catch (e) {}

本地缓存的格式是key-value

通过getStorage(), getStorageSync()每次读取一个key对应的value

异步方法通过success方法的参数接收结果，同步方法数据直接是返回值

getStorage参数object同时也可以有fail、complete属性

wx.setStorage({

key:"key",

data:"value1"

})

try{

wx.setStorageSync('key', 'value2')

}catch (e) {}

wx.setStorage/wx.setStorageSync写入本地数据缓存

需要存储的data可以是Object/String

success, fail, complete方法同理

* 缓存限制和隔离

每个小程序的缓存空间上限为10MB，如果当前缓存已经达到10MB，再通过wx.setStorage写入缓存会触发fail回调。

由于本地缓存是存放在当前设备，用户换设备之后无法从另一个设备读取到当前设备数据，因此用户的关键信息不建议只存在本地缓存，应该把数据放到服务器端进行持久化存储。

* 用途：缓存用户登录态SessionId

wx.setStorageSync('SESSIONID',data.sessionId)

var expiredTime = +new Date() +1\*24\*60\*60\*1000

wx.setStorageSync('EXPIREDTIME',expiredTime)

将sessionID和自定义的expiredTime存储到本地，下一次进入小程序时

onLaunch: function(options) {

var sessionId =wx.getStorageSync('SESSIONID')

var expiredTime =wx.getStorageSync('EXPIREDTIME')

var now = +new Date()

if (now - expiredTime <=1\*24\*60\*60\*1000) {

this.globalData.sessionId = sessionId

this.globalData.expiredTime = expiredTime

}

检查本地是否有session信息以及是否过期，可以直接登录而不用再请求远程服务器

当然此处的过期期限是模拟的，真正的小程序本地数据有效期限谁也不知道