- **学习视频**
  - **黑马2021微信小程序开发教程**
- **官方文档**

第一章 起步

一、小程序简介与开始

1 - 小程序与普通网页开发的区别

- 运行环境不同
  - 网页 - 浏览器环境
  - 小程序 - 微信环境
- API 不同
  - 由于运行环境的不同， 小程序 无法调用 DOM 和 BOM 的 API
  - 小程序 可以调用 微信环境 提供的各种 API
    - 地理定位
    - 扫码
    - 支付
- 开发模式不同
  - 网页 - 浏览器 + 代码编辑器
  - 小程序
    - 申请小程序开发账号
    - 安装小程序开发工具
    - 创建和配置小程序项目

2 - 注册小程序开发账号

小程序注册页

- 注册小程序开发账号
- 登陆 小程序后台， 获取小程序的 AppID
  - 开发管理 -> 开发设置 -> AppID(小程序ID)

3 - 安装微信开发者工具

微信开发者工具 是官方推荐使用的小程序开发工具，提供的主要功能：

- 快速创建 小程序项目

- 查看和编辑 代码
- 调试 小程序功能
- 预览和发布 小程序

下载安装 最新的稳定版 的微信开发者工具

下载地址

4 - 创建小程序项目

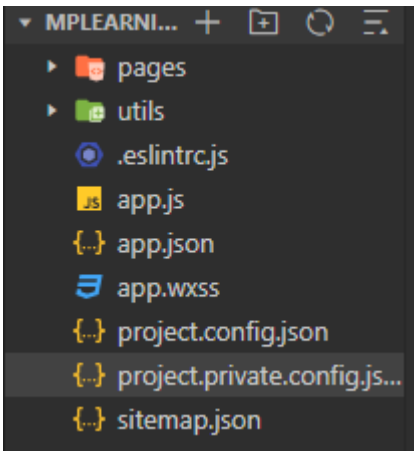
- 新建项目选择小程序项目
- 选择代码存放的硬盘路径，填入刚刚申请到的小程序的 AppID
- 给你项目起名，勾选 "不使用云服务" （注意: 你要选择一个空的目录才可以创建项目）
- 点击新建，你就得到了你的第一个小程序了
- 点击顶部菜单编译就可以在微信开发者工具中预览你的第一个小程序。

5 - 编译预览

- 点击工具上的编译按钮，可以在工具的左侧模拟器界面看到这个小程序的表现
- 也可以点击预览按钮，通过微信的扫一扫在手机上体验你的第一个小程序

二、小程序代码构成

1 - 项目的基本组成结构

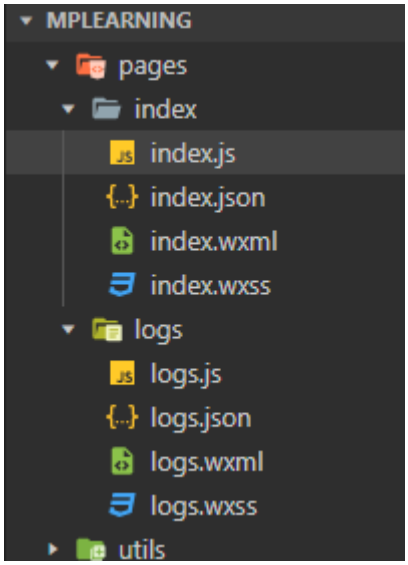


- pages - 用来存放 所有小程序的页面
- utils - 用来存放 工具性质的模块 （如：格式化时间的自定义模块）
- app.js - 小程序项目的 入口文件
- app.json - 小程序项目的 全局配置文件
    - 包括了小程序的所有页面路径、界面表现、网络超时时间、底部 tab 等
    - 各配置项的含义：

        - pages 字段 - 描述当前小程序所有页面路径
        - window 字段 - 定义小程序所有页面的顶部背景颜色，文字颜色定义等
- app.wxss - 小程序项目的 全局样式文件
- project.config.json - 小程序项目的 配置文件
    - 当前开发项目的个性化配置，其中会包括编辑器的颜色、代码上传时自动压缩等等
    - 重新安装工具或者换电脑工作时，只要载入同一个项目的代码包，开发者工具就自动会帮你恢复
- sitemap.json - 配置 小程序及其页面 是否允许 被微信索引

2 - 小程序页面的组成部分

小程序官网建议把所有小程序页面，都存放在 pages 目录 中，以 单独的文件夹存在



每个页面 **由 4 个基本文件** 组成：

- .js 文件 - 页面的 脚本文件 ，存放页面的数据、事件处理函数等
- .json 文件 - 页面的 配置文件 ，配置窗口的外观、表现等
- .wxml 文件 - 页面的 模版结构文件
- .wxss 文件 - 页面的 样式表文件

2.1 - JSON 配置文件的作用

JSON 是一种数据格式，并不是编程语言，在小程序中，JSON扮演的静态配置的角色

通过 .json 配置文件，可对小程序项目进行不同级别的配置

小程序项目中有 4 种 json 配置文件：

- 项目根目录中的 app.json 配置文件
- 项目根目录中的 project.json 配置文件
- 项目根目录中的 sitemap.json 配置文件
- 每个页面文件夹 中的 .json 配置文件

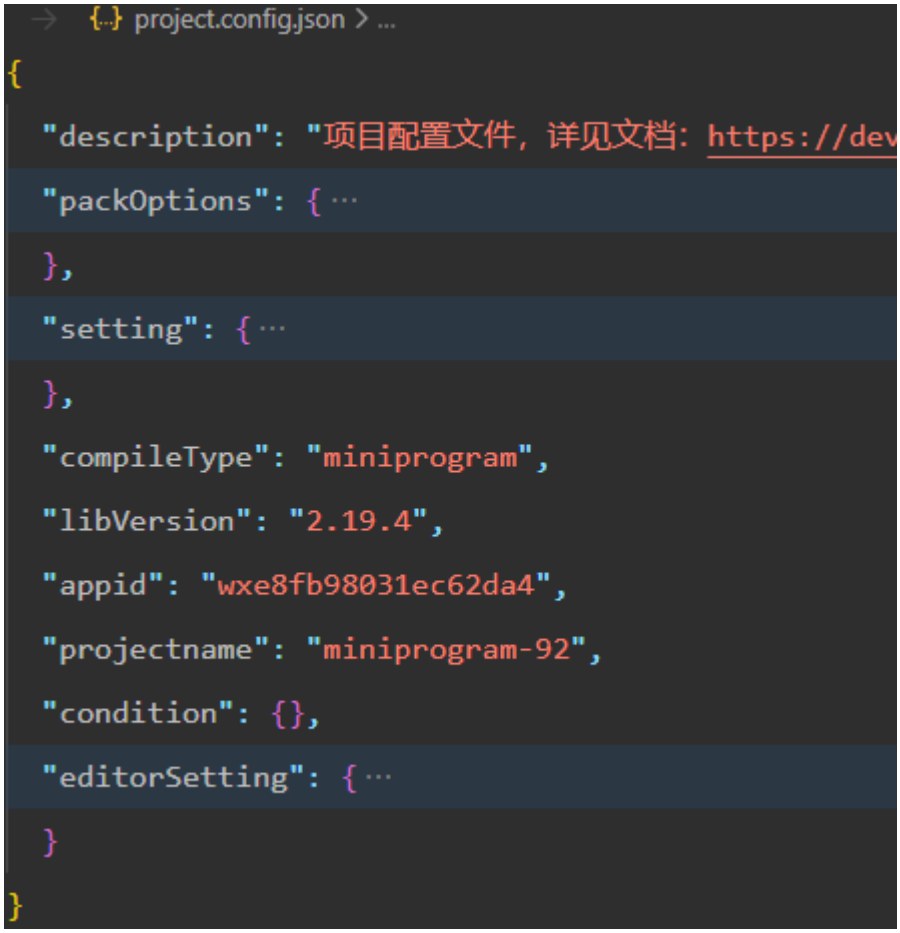
2.1.1 - app.json

`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

- pages - 记录当前小程序 所有页面的路径
  - 文件名不需要写文件后缀，框架会自动去寻找对应位置的 `.json`，`.js`，`.wxml`，`.wxss` 四个文件进行处理
  - 未指定 `entryPagePath` 时，数组的第一项代表小程序的初始页面（首页）
  - **小程序中新增/减少页面，都需要对 pages 数组进行修改**
- window - 设置小程序 状态栏、导航条、标题、窗口背景色
- style - 全局定义小程序 组件使用的样式版本
  - app.json 中配置 `"style": "v2"` 可表明启用新版的组件样式
- sitemapLocation - 指明 sitemap.json 的位置
  - 默认为 app.json 同级目录下名字的 `sitemap.json` 文件

2.1.2 - project.config.json

小程序项目的 配置文件



- setting - 保存了 编译相关的配置
- projectname - 保存了 项目名称
  - 小程序名称 != 项目名称 （这里 MPlearing != miniprogram-92）
- appid - 保存了 小程序的账号 ID

2.1.3 - sitemap.json

微信现已开发 小程序内搜索，效果类似于 PC 网页的 SEO

sitemap.json 文件用来 配置小程序页面是否允许微信索引

2.1.4 - 页面的 .json 配置文件

小程序的每个页面。可使用 .json 文件对 当前页面的窗口外观进行配置

> **页面中的配置项会 覆盖 app.json 的 window 中相同的配置项**

2.2 - 新建小程序页面

只需在 `app.json` -> `pages` 中新增页面的存放路径，小程序开发者工具会 自动创建对应的页面文件

2.3 - 修改项目首页

只需调整 `app.json` -> `pages` 数组中页面路径的前后顺序，即可修改首页

小程序会把排在第一位的页面，当做项目首页进行渲染

3 - WXML 模版

WXML（WeiXin Markup Language）是小程序框架设计中的一套 标签语言 ， 用来构建小程序页面的结构 ，其作用类似于网页开发中的 HTML，结合基础组件、事件系统，可以构建出页面的结构

3.1 - WXML 和 HTML 区别

- 标签名字不同
  - WXML - view、text、image、navigator
  - HTML - div、span、img、a
- 属性节点不同
  - WXML - `<navigator url="/pages/home/home"></navigator>`
  - HTML - `<a href="#">超链接</a>`
- 提供了类似于 Vue 中的模版语法
  - 数据绑定
  - 列表渲染
  - 条件渲染

4 - WXSS

WXSS（WeiXin Style Sheets）是小程序框架设计中的一套 样式语言 ，用于描述 WXML 的组件样式，类似于网页开发中的 CSS

4.1 - WXSS 和 CSS 的区别

- 新增了 rpx 尺寸单位
  - `WXSS` 在底层支持新的尺寸单位 `rpx`，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
    - **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**
  - `CSS` 中需手动进行像素单位的换算，例如 rem
- 提供了全局的样式和局部样式
  - 项目根目录中的 app.wxss 会作用于所有小程序页面
  - 局部页面的 .wxss 样式仅对当前页面生效，并会覆盖 app.wxss 中相同的选择器。
- `WXSS` 仅支持部分 `CSS` 选择器
  - .class
  - #id

- element
- element, element
- ::after
- ::before

5 - JS 逻辑交互

在小程序里边，可通过编写 `JS` 脚本文件来处理用户的操作。例如：响应用户的点击、获取用户的位置

小程序中的 JS 文件分为 3 大类：

- app.js - 整个小程序项目的入口文件，通过调用 APP() 函数来 启动 整个小程序
- 页面的的 .js 文件 - 页面的入口文件，通过调用 Page() 函数来 创建并运行 页面
- 普通的 .js 文件 - 普通的功能模块文件，用来 封装 公共的函数或属性 供页面使用

三、小程序的宿主环境

宿主环境（host environment）指 程序运行所必须的依赖环境

- Android 系统 和 iOS 系统 是两个不同的宿主环境
- 安卓版的 微信App 不能在 iOS 环境下运行，Android 是安卓软件的宿主环境
- 脱离宿主环境的软件是没有任何意义的

1 - 小程序的宿主环境

手机微信 是 小程序的宿主环境

小程序 借助宿主环境提供的能力，可以完成许多普通网页无法完成的功能

如：微信登陆、微信扫码、微信支付、地理定位、etc.....

2 - 小程序宿主环境包含的内容

- **通信模型**
- **程序与页面**
- **组件**
- **API**

2.1 - 通信模型

2.1.1 - 通信主体

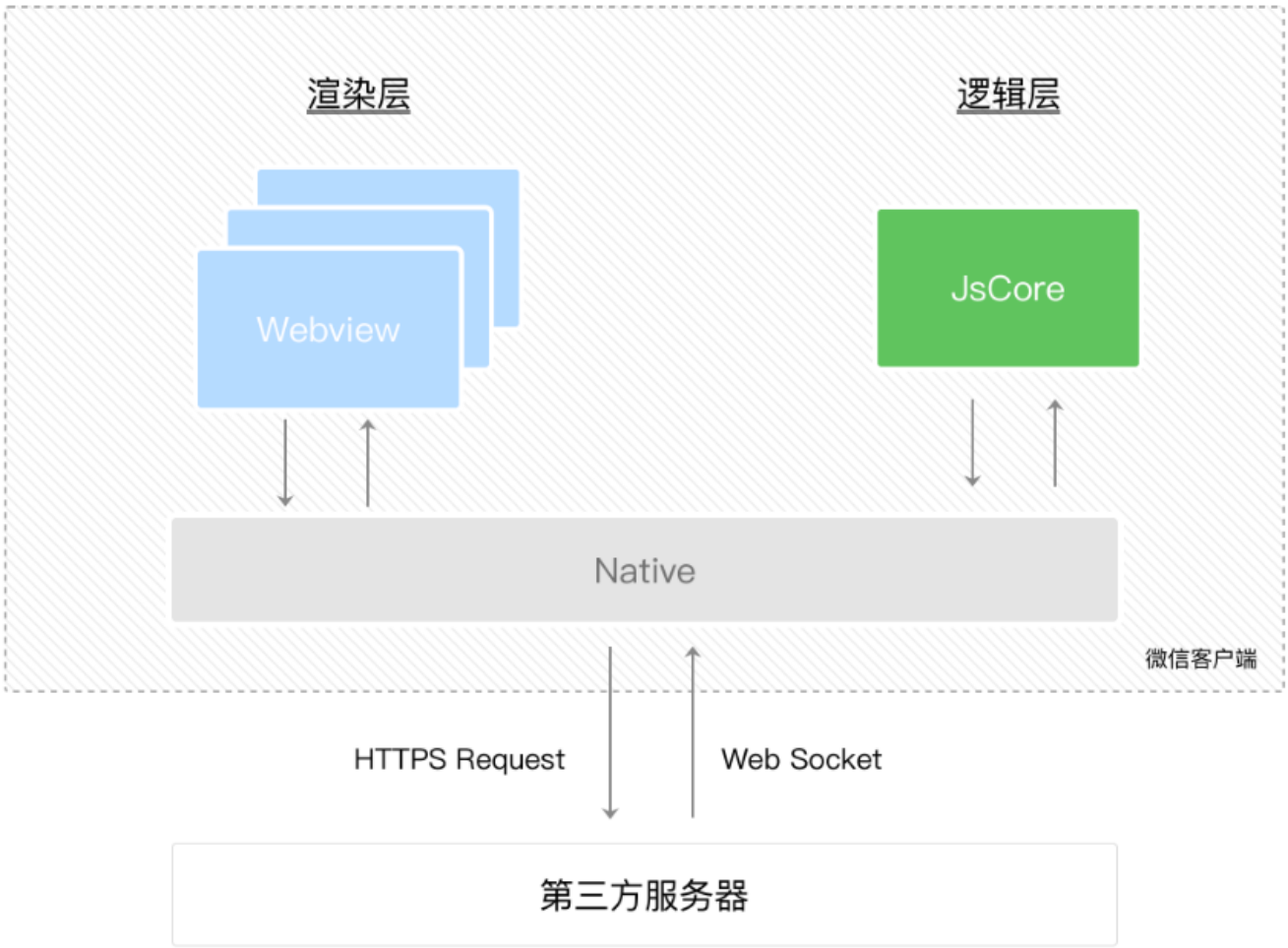
小程序中通信的主体是 渲染层 和 逻辑层：

- WXML 模版 和 WXSS 样式 工作在 渲染层
  - 渲染层的界面使用了WebView 进行渲染
- JS 脚本 工作在 逻辑层
  - 逻辑层采用 JsCore 线程运行 JS 脚本

2.1.2 - 通信模型

小程序中的通信模型分为两部分

- 渲染层 和 逻辑层 之间的通信
  - 经由微信客户端做中转
- 逻辑层 和 第三方服务器 之间的通信
  - 经由微信客户端做中转



2.2 - 程序与页面

2.2.1 - 小程序启动的过程

- 把小程序的代码包下载到本地
- 解析 app.json 全局配置文件
- 执行 app.js 小程序入口文件， 调用 App() 创建小程序实例
- 渲染小程序首页
- 小程序启动完成（可调用 `onLaunch` 回调）

2.2.2 - 小程序渲染的过程

- 加载解析页面的 .json 配置文件
- 加载页面的 .wxml 模版和 .wxss 样式
- 执行页面的 .js 文件， 调用 Page() 创建页面实例
- 页面渲染完成（可调用 `onLoad` 回调）

2.3 - 组件

小程序中的组件 也是 由宿主环境提供， 开发者可基于组件快速搭建除漂亮的页面结构

- 只需要在 `WXML` 写上对应的组件标签名字就可以把该组件显示在界面上

```
<!-- 在界面上显示地图 -->
<map></map>
```

- 使用组件的时候，还可以通过属性传递值给组件，让组件可以以不同的状态去展现

```
<!-- 设置地图一开始的中心的经纬度是广州 -->
<map longitude="广州经度" latitude="广州纬度"></map>
```

- 组件的内部行为也会通过事件的形式让开发者可以感知

```
<!-- 用户点击了地图上的某个标记，在 js 编写 markertap 函数来处理 -->
<map bindmarkertap="markertap" longitude="广州经度" latitude="广州纬度"></map>
```

官方把小程序的组件氛围了 9 大类：

- 视图容器
- 基础组件
- 表单组件
- 导航组件
- 媒体组件
- map 地图组件
- canvas 画布组件
- 开放能力
- 无障碍访问

2.4 - API

小程序中的 API 是由宿主环境提供的，通过这些丰富的小程序 API，开发者可以方便地调用微信提供的能力

例如：获取用户信息、本地存储、支付功能等

官方把 API 分为了 3 大类：

- 事件监听 API
  - **特点**：以 on 开头，用来 监听某些事件的触发
  - **示例**：

```
wx.onCompassChange(function (res) {
  console.log(res.direction)
})
```

- 同步 API
  - **特点**：以 sync 结尾
  - **特点**：同步 API 的执行结果，可以通过函数返回值直接获取，如果执行出错会抛出异常
  - **示例**：

```
try {
  wx.setStorageSync('key', 'value')
} catch (e) {
  console.error(e)
}
```

- 异步 API
  - **特点**：大多数 API 都是异步 API，这类 API 接口通常都接受一个 `Object` 类型的参数，需要通过 success、fall、complate 接收调用的结果
  - **示例**：

```
wx.login({
  success(res) {
    console.log(res.code)
  }
})
```

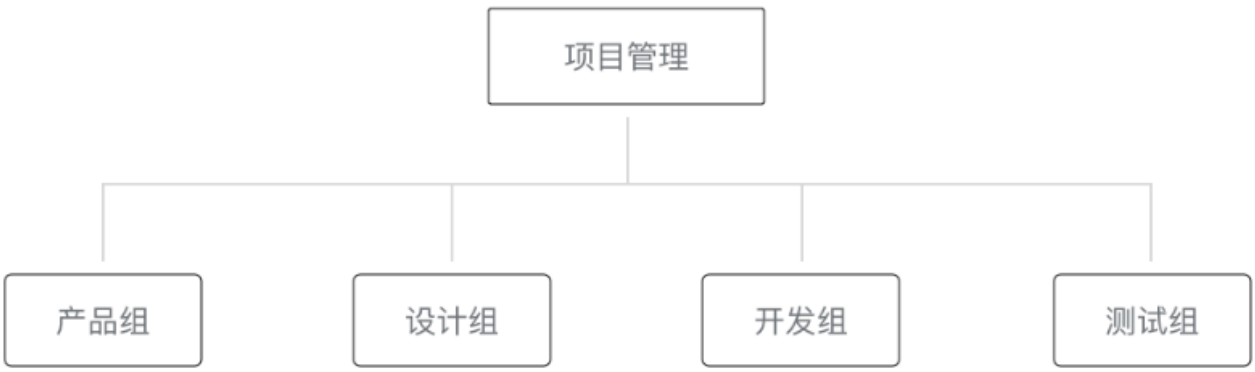
四、小程序协同工作和发布

中大型的公司里，人员的分工非常仔细：同一小程序项目。一般会有不同岗位，不同角色的员工同时参与设计与开发

此时出于管理需要， 迫切需要 对不同岗位、不同角色的 员工的权限进行边界的划分 ，使他们能够高效地进行协同工作

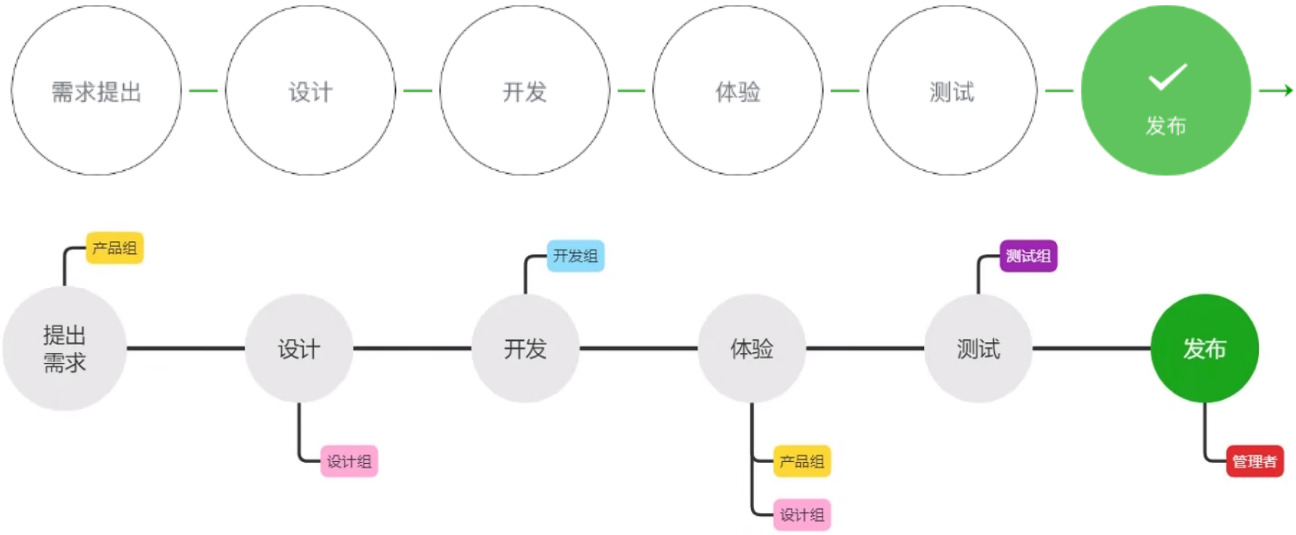
1 - 协同工作

1.1 - 项目成员组织结构



- 项目管理成员 - 负责统筹整个项目的进展和风险、把控小程序对外发布的节奏
- 产品组 - 提出需求
- 设计组与产品组 - 讨论并对需求进行抽象，设计出可视化流程与图形，输出设计方案
- 开发组 - 依据设计方案，进行程序代码的编写
- 设计组与产品组 - 代码编写完成后 ，产品组与设计组体验小程序的整体流程
- 测试组 - 编写测试用例并对小程序进行各种边界测试

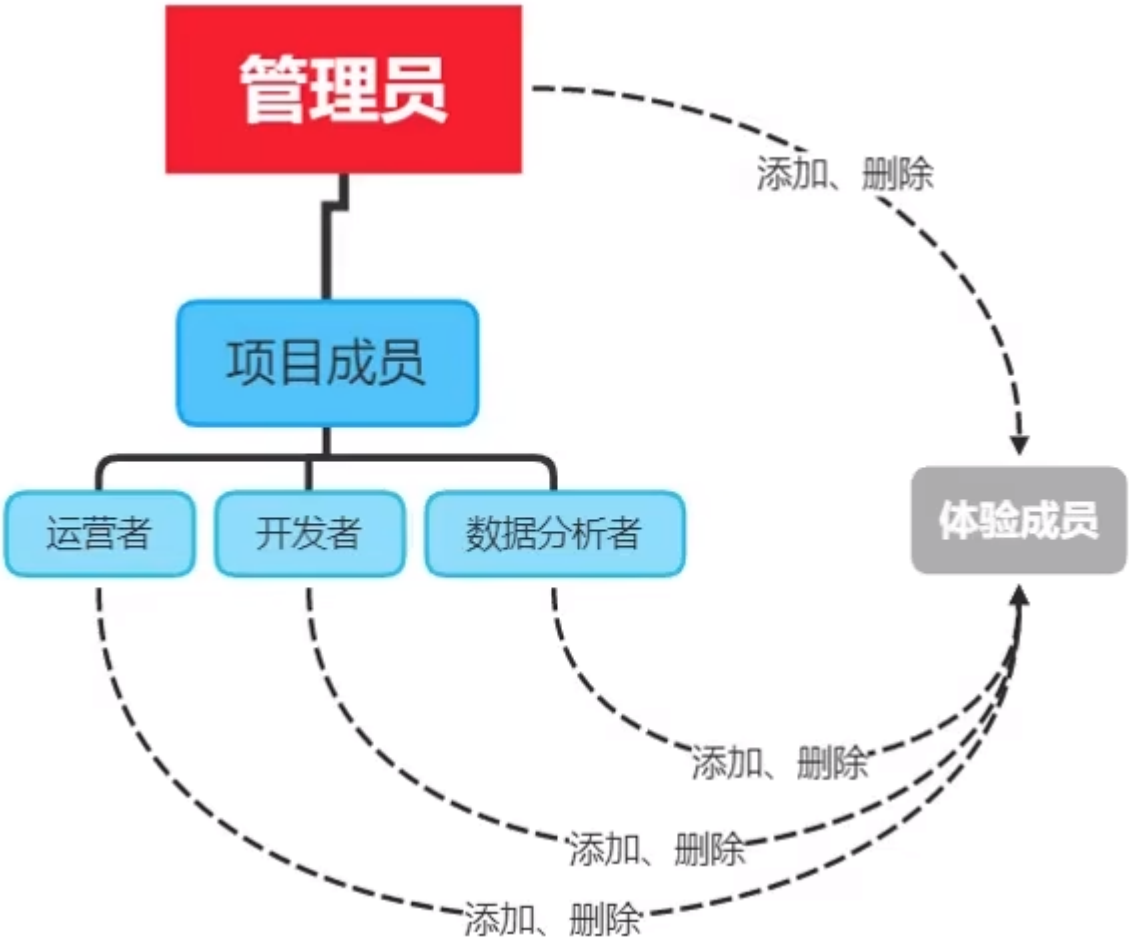
1.2 - 项目工作流程



1.3 - 小程序成员管理

小程序成员管理包括对小程序项目成员及体验成员的管理

- 项目成员 - 参与小程序开发、运营的成员，可登录小程序管理后台，包括运营者、开发者及数据分析者
  - 管理员 可在“成员管理”中添加、删除项目成员，并设置项目成员的角色
- 体验成员 - 参与小程序内测体验的成员，可使用体验版小程序，但不属于项目成员
  - 管理员 及 项目成员 均可添加、删除体验成员



1.4 - 不同项目成员对应的权限

不同项目成员拥有不同的权限，从而保证小程序开发安全有序

权限	运营者	开发者	数据分析者
开发者权限		√	
体验者权限	√	√	√
登录	√	√	√
数据分析			√
微信支付	√		
推广	√		
开发管理	√		
开发设置		√	
暂停服务	√		
解除关联公众号	√		
腾讯云管理		√	
小程序插件	√		
游戏运营管理	√		

- 开发者权限 - 可使用小程序开发者工具及开发版小程序进行开发
- 体验者权限 - 可使用体验版小程序
- 登陆权限 - 可登录小程序管理后台，无需管理员确认
- 开发设置 - 设置小程序服务器域名、消息推送及扫描普通链接二维码打开小程序
- 腾讯云管理 - 云开发相关设置

1.5 - 添加项目成员和体验成员

`登陆微信小程序后台` -> `管理` -> `成员管理`

2 - 小程序的版本

2.1 - 软件开发过程中的版本

在软件开发过程中，根据时间节点的不同，会产出不同的软件版本

- 开发版本 - 开发者编写代码的同时，对项目代码进行 自测
- 体验版本 - 程序达到稳定可体验的状态，开发者把体验版本给到产品经理和测试人员进行 体验测试
- 正式版本 - 修复完程序 bug 后，发布正式版供外部用户使用

2.2 - 小程序的版本

小程序的版本根据这个流程设计了小程序版本的概念

权限	说明
开发版本	使用开发者工具，可将代码上传到开发版本中 开发版本只保留每人最新的一份上传的代码 点击提交审核，可将代码提交审核 开发版本可删除，不影响线上版本和审核中版本的代码
体验版本	可以选择某个开发版本作为体验版，并且选取一份体验版
审核中版本	只能有一份代码处于审核中 有审核结果后可以发布到线上，也可直接重新提交审核，覆盖原审核版本
线上版本	线上所有用户使用的代码版本，该版本代码在新版本代码发布后被覆盖更新

3 - 发布上线

一个小程序从开发完到上线一般要经过 预览-> 上传代码 -> 提交审核 -> 发布等步骤

3.1 - 预览

使用开发者工具可以预览小程序，帮助开发者检查小程序在移动客户端上的真实表现

3.2 - 上传代码

上传代码是用于提交体验或者审核使用

- 点击开发者工具顶部操作栏的 `上传` 按钮
- 填写 版本号 以及 项目备注
    - 此处版本号以及项目备注是为了方便管理员检查版本使用的
    - 开发者可以根据自己的实际要求来填写这两个字段

3.2.1 - 后台查看上传结果

上传成功之后，登录小程序管理后台 - `版本管理` - `开发版本` ，就可以找到刚提交上传的版本了

可以将这个版本设置 体验版 或者是 提交审核

3.3 - 提交审核

为保证小程序的质量，以及符合相关的规范，小程序的发布，是需要经过腾讯官方审核的

提交审核的方式:

- 开发者工具中上传了小程序代码之后
- 登录 小程序管理后台 - 版本管理 - 开发版本
- 找到提交上传的版本，在开发版本的列表中，点击 **提交审核**

- 按照页面提示，填写相关的信息，即可以将小程序提交审核

> **开发者严格测试了版本之后，再提交审核， 过多的审核不通过，可能会影响后续的时间**

3.4 - 发布

- 审核通过之后，管理员的微信中会收到小程序通过审核的通知
- 登录 小程序管理后台 - 开发管理 - 审核版本中，可以看到通过审核的版本
- 点击发布后，即可发布小程序

3.4.1 - 发布模式

小程序提供了两种发布模式

- 全量发布 - 当点击发布之后，所有用户访问小程序时都会使用当前最新的发布版本
- 分阶段发布（灰度发布） - 分不同时间段来控制部分用户使用最新的发布版本

4 - 基于小程序码进行推广

在小程序设计的初期，小程序平台提供的二维码的形式，用户并不知道当前这次扫码会出现什么样的服务

因此微信设计了小程序码，让用户在扫码之前就有一个明确的预期

相比普通二维码， 小程序码 的优势：

- 在样式上更具 辨识度 和 视觉冲击力
- 能够更加清晰地树立小程序的 品牌形象
- 可以帮助开发者 更好地推广小程序

获取小程序码的 5 个步骤：

登陆小程序管理后台 -> 设置 -> 基本设置 -> 基本信息 -> 小程序码及线下物料下载

4 - 运营数据

查看小程序运营数据的 两种 方式：

- 在 小程序后台 查看
  - 登录 小程序管理后台
  - 统计
  - 点击相应的 tab 可以看到相关的数据
- 使用 小程序数据助手 查看
  - 打开微信
  - 搜索“小程序数据助手”

- 查看已发布的小程序相关的数据

第二章、模版与配置

一、WXML 模版语法

1 - 数据绑定

WXML 中的动态数据均来自对应 Page 的 data

```
Page({
    data: {
        info: 'init data',
        msgList: [{msg: 'hello'}, {msg: 'world'}]
    }
})
```

1.1 - 简单绑定

数据绑定使用 Mustache 语法（双大括号）将变量包起来，可作用于:

1.1.1 - 内容

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

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

1.1.2 - 组件属性

需要在双引号之内

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

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

1.1.3 - 控制属性

需要在双引号之内

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

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

1.1.4 - 关键字

需要在双引号之内

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

> **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**

1.2 - 运算

可以在 `{{}}` 内进行简单的运算，支持的有如下几种方式

1.2.1 - 三元运算

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

1.2.2 - 算术运算

```
<!-- view中的内容为 3 + 3 + d -->
<view> {{a + b}} + {{c}} + d </view>
```

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

1.2.3 - 逻辑判断

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

1.2.4 - 字符串运算

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

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

1.2.5 - 数据路径运算

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

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

1.3 - 组合

也可以在 Mustache 内直接进行组合，构成新的对象或者数组

1.3.1 - 数组

```
<!-- 最终组合成数组[0, 1, 2, 3, 4] -->
<view wx:for="{{[zero, 1, 2, 3, 4]}}"> {{item}} </view>
```

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

1.3.2 - 对象

```
<!-- 最终组合成的对象是 {for: 1, bar: 2} -->
<template is="objectCombine" data="{{for: a, bar: b}}"></template>
```

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

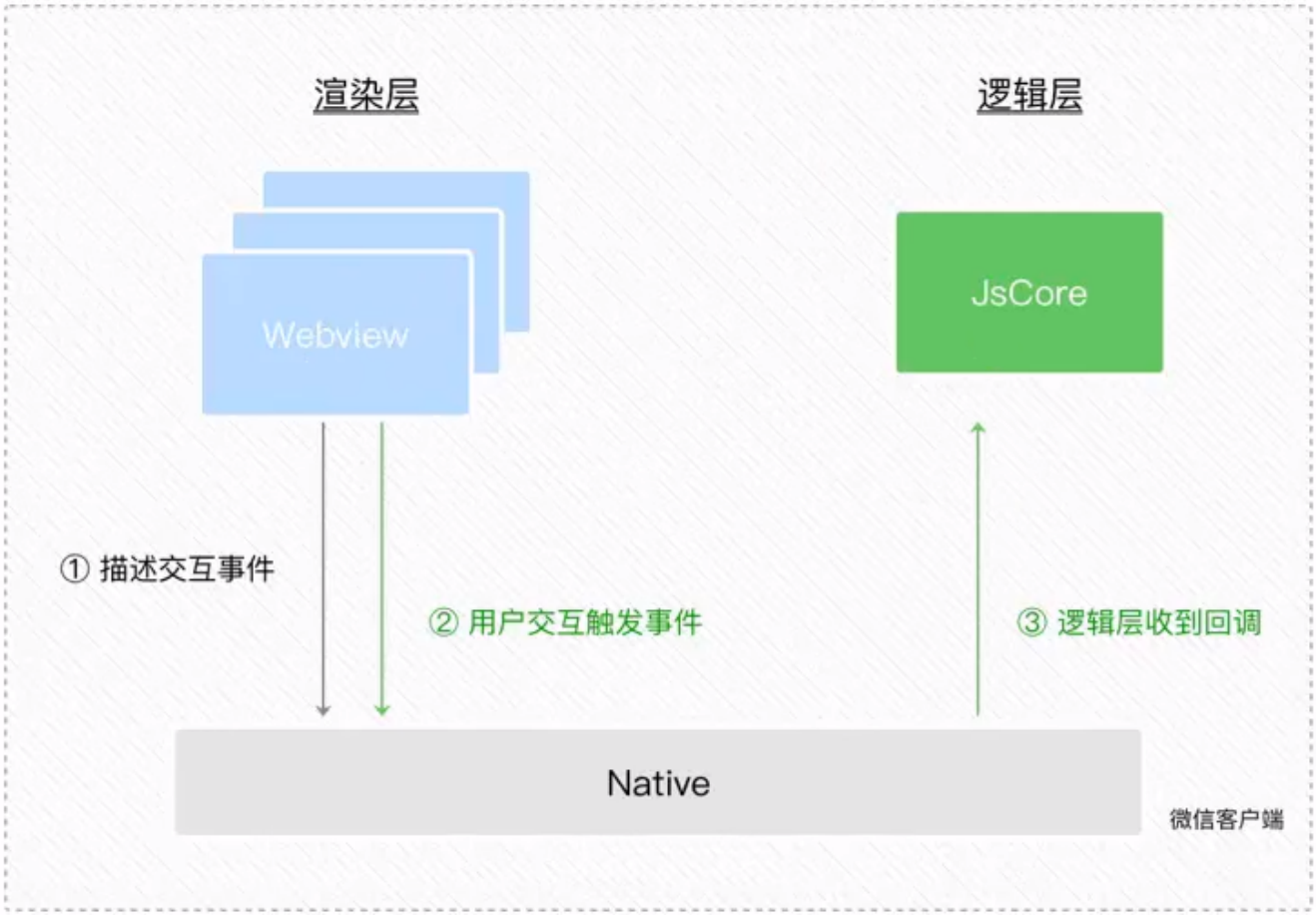
> **1、也可以用扩展运算符 `...` 来将一个对象展开**
>
> **2、如果对象的 key 和 value 相同，也可以间接地表达**
>
> **3、方式可以随意组合，但是如有存在变量名相同的情况，后边的会覆盖前面**
>
> **4、花括号和引号之间如果有空格，将最终被解析成为字符串**

2 - 事件系统

2.1 - 事件介绍

事件 是 渲染层到逻辑层的通信方式 。通过事件可以将用户在渲染层产生的行为，反馈到逻辑层进行处理

- 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

2.2 - 小程序中常用事件

类型	绑定方式	事件描述
tap	bindtap 或 bind:tap	手指触摸后马上离开，类似于 HTML 中的 click 事件
input	bindput 或 bind:input	文本框的输入事件
change	bindchange 或 bind:change	状态改变时触发

2.3 - 事件对象的属性列表

当事件回调触发时，会收到一个事件对象 event ， 它的详细属性如下表所示

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
mark	Object	事件标记数据

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

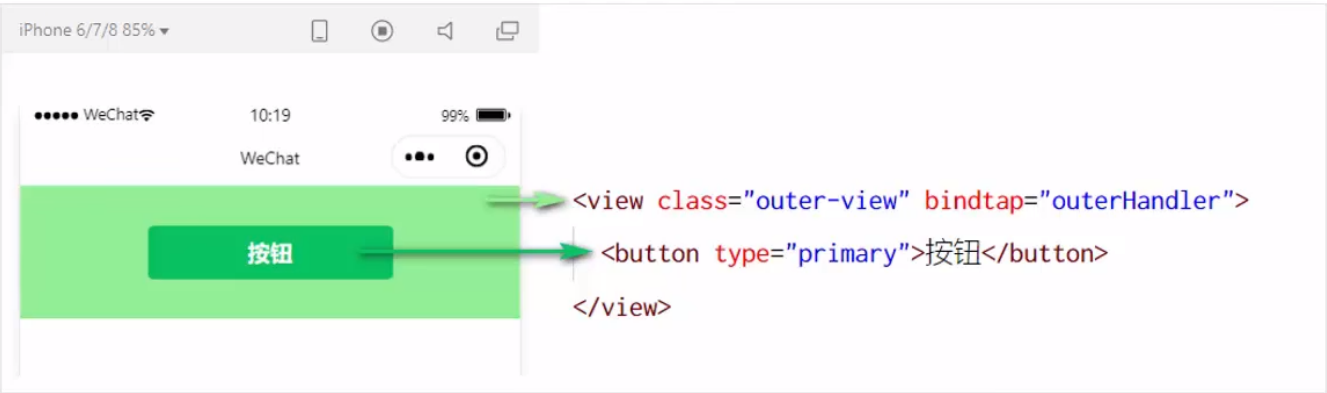
TouchEvent 触摸事件对象属性列表（继承 BaseEvent）

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

2.4 - target 和 currentTarget 的区别

- target - 触发该事件的源头组件
- currentTarget - 当前事件所绑定的组件

示例



点击内部按钮时，点击事件以 冒泡 的方式向外扩散，也会触发外层 view 的 tap 事件处理函数

此时，对于外层的 view 来说

- e.target 是内部的按钮组件
- e.currentTarget 是外部的 view 组件

2.5 - 普通事件绑定

```
<view bindtap="handleTap">
    Click here!
</view>
```

```
page({
    handleTap(e) {
        console.log(e)
    }
})
```

> **1、以字符串指定事件处理函数名**
>
> **2、如果它是个空字符串，则这个绑定会失效（可以利用这个特性来暂时禁用一些事件）**

2.6 - 在事件处理函数中为 data 中的数据赋值

通过调用 this.setData(dataObject) 方法，可给页面 data 中的数据重新赋值

```
// 页面的 .js 文件
page({
    data: {
        count: 0
    },

    // 修改 count 的值
    changeCount({
        this.setData({
            count: this.data.count + 1
        })
    })
})
```

2.7 - 事件传参

小程序中， 不能在绑定事件的同时为事件处理函数传递参数

```
<!-- 在小程序中，不起作用 ↓↓↓ -->
<button type="primary" bindtap="btnHandler(123)">事件传参</button>
```

小程序会将 bindtap 的属性值，统一当做 事件名称来处理 ， 此处相当于调用一个名称为 btnHandler(123) 的实践处理函数

2.7.1 - 小程序中的事件传参

可以为组件提供 data-* 自定义属性传参，其中 * 代表的是参数的名字

```
<button type="primary" bindtap="btnHandler" data-info="{{2}}">事件传参</button>
```

最终：

- info - 解析为参数的名字
- 数值2 - 解析为参数的值

在事件处理函数中，通过 **event.target.dataset.参数名** 即可获取 具体参数的值

```
btnHandler (event) {
    console.log(event.target.dataset.info)
}
```

> **dataset 是一个 对象 ， 包含了所有通过 data-* 传递过来的参数项**
>
> **通过 dataset，可以访问到具体参数的值**

2.8 - input 文本框和 data 之间的数据同步

在 WXML 中，普通的属性的绑定是 单向 的

```
<input value="{{value}}" />
```

- 如果使用 `this.setData({ value: 'leaf' })` 来更新 `value` ， `this.data.value` 和输入框的中显示的值都会被更新为 `leaf`
- 但如果用户修改了输入框里的值，却不会同时改变 `this.data.value`

如果需要在用户输入的同时改变 `this.data.value` ，需要借助简易双向绑定机制

此时，可以在对应项目之前加入 `model:` 前缀

```
<input model:value="{{value}}" />
```

这样，如果输入框的值被改变了， `this.data.value` 也会同时改变

也可以使用 `e.detail.value`

```
<input value="{{msg}}" bindinput="inputHandler"/>
```

```
data: {
    msg: 123
}

inputHandler(e) {
  this.setData({
    msg: e.detail.value
  })
}
```

---

3 - 列表渲染

3.1 - 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

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

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

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

3.2 - 手动指定当前项的变量名和索引

- 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>
```

> **可用于嵌套**

```
<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>
```

3.3 - wx:key

类似于 Vue 列表渲染中的 :key，小程序在实现列表渲染时，也建议为渲染出来的列表项指定唯一的 key 值，从而提高渲染的效率

```
// data 数据
data: {
    userList: [
        { id: 1, name: 'Alice' },
        { id: 2, name: 'bob' },
        { id: 3, name: 'coco'}
    ]
}
```

```
<!-- wxml 结构 -->
<view wx:for="{{ userList}}" wx:key="id">{{ item.name }}</view>
```

绑定 key 值时，不用插值表达式

4 - 条件渲染

4.1 - wx:if

在框架中，使用 `wx:if=""` 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>
```

4.2 - 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>
```

> `<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**

4.3 - hidden

在小程序中，直接使用 `hidden="{{ condition }}"` 也能控制元素的显示和隐藏

- 条件为 true 时 隐藏 元素
- 条件为 false 时 显示 元素

4.4 - wx:if 对比 hidden

- **运行方式**
  - wx:if - 以 动态创建和移除元素 的方式，控制元素的展示和隐藏
  - hidden - 以 切换样式 的方式（ display: none/block; ）控制元素的显示和隐藏
- **使用建议**
  - 频繁切换 时 - hidden
  - 控制条件复杂 时 - wx:if 搭配 wx:elif 、wx:else 进行展示与隐藏的切换

二、WXSS 模版样式

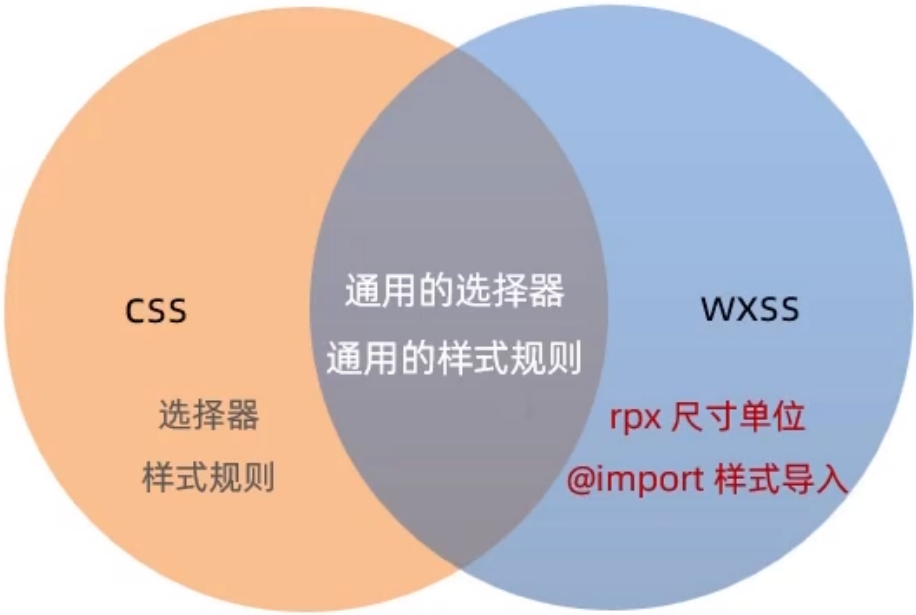
WXSS (WeiXin Style Sheets)是一套样式语言，用于描述 WXML 的组件样式。

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



1 - rpx 尺寸单位

rpx （responsive pixel）是微信小程序独有的，用来 解决屏适配的尺寸单位

1.1 - rpx 实现原理

rpx 把所有设备的屏幕，在**宽度**上 等分 750 份 （即： 当前屏幕的总宽度为 750 rpx ）

- 在 较小 的设备上， 1 rpx 所代表的宽度较小
- 在 较大 的设备上， 1 rpx 所代表的宽度较大

小程序在不同设备上运行时，自动把 rpx 的样式单位换算成对应的像素单位来渲染，从而实现屏幕适配

> **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**

2 - 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

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

```

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

3 - 全局样式和局部样式

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

> **当局部样式的 权重大于或等于 全局样式的权重时，才会覆盖全局的样式**

三、全局配置和页面配置

1 - 全局配置文件及常用的配置项

小程序目录下的 app.json 文件时小程序的 全局配置文件

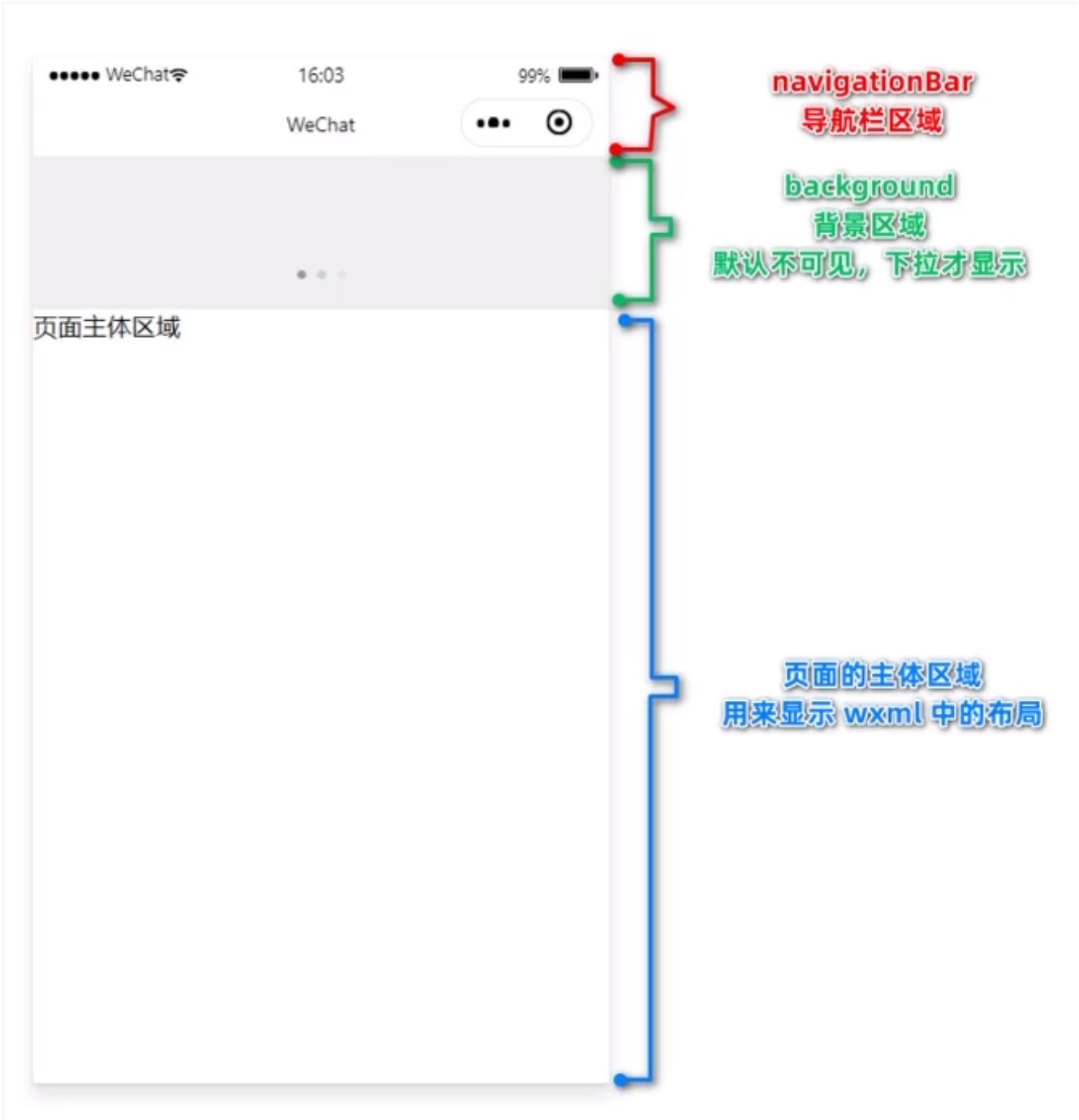
常用的配置项如下：

- pages
  - 记录当前小程序所有页面的存放路径
- window
  - 全局设置小程序窗口的外观
- tabBar
  - 设置小程序底部的 tabBar 效果
- style
  - 是否启用新版的组件样式

1.1 - window

1.1.1 - 小程序窗口的组成部分

导航栏区域 和 背景区域 可通过 window 节点 进行全局配置



1.1.2 - window 节点常用的配置项

属性名	类型	默认值	说明
navigationBarTitleText	string		导航栏标题文字内容
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`
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	是否开启全局的下拉刷新。
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

1.2 - tabBar

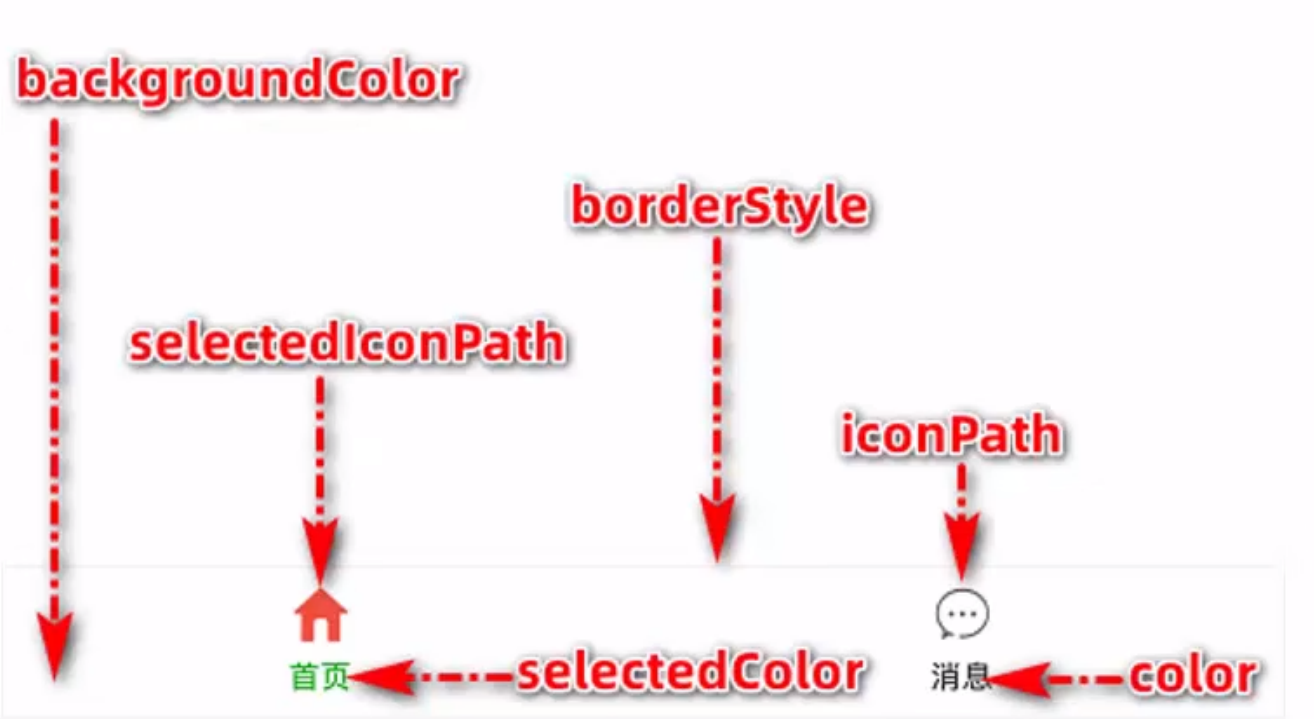
tabBar 是移动端应用常见的页面效果， 用于实现多页面的快速切换

小程序中通常将其分为

- 底部 tabBar
- 顶部 tabBar

> **1、tabBar 中只能配置 最少 2 个、最多 5 个 tab 页签**
>
> **2、当渲染 顶部 tabBar 时， 不显示 icon ， 只显示文本**

1.2.1 - tabBar 的组成部分



属性名	类型	默认值	说明
backgroundColor	HexColor		tab 的背景色，仅支持十六进制颜色
selectedIconPath	string		选中时的图片路径
selectedColor	HexColor		tab 上的文字选中时的颜色，仅支持十六进制颜色
iconPath	string		图片路径
color	HexColor		tab 上的文字默认颜色，仅支持十六进制颜色
borderStyle	string	black	tabbar 上边框的颜色， 仅支持 `black` / `white`

1.2.2 - tabBar 节点的配置项

属性名	类型	必填	默认值	说明
position	string	否	bottom	tabBar 的位置，仅支持 `bottom` / `top`
borderStyle	string	否	black	tabbar 上边框的颜色， 仅支持 `black` / `white`
color	HexColor	是		tab 上的文字默认颜色，仅支持十六进制颜色
selectedColor	HexColor	是		tab 上的文字选中时的颜色，仅支持十六进制颜色
backgroundColor	HexColor	是		tab 的背景色，仅支持十六进制颜色
list	Array	是		tab 的列表，最少 2 个、最多 5 个 tab

`list` 属性说明

list 接受一个 数组，**只能配置最少 2 个、最多 5 个 tab**。tab 按数组的顺序排序，每个项都是一个对象

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，**当 position 为 top 时，不显示 icon**
selectedIconPath	string	否	选中时的图片路径，**当 position 为 top 时，不显示 icon**

2 - 页面配置

小程序中，每个页面都由自己的 .json 配置文件，用来对 当前页面 的窗口外观、页面效果等进行配置

2.1 - 页面配置和全局配置的关系

小程序中

- app.json 中的window 节点 - 全局配置 每个页面的窗口表现
- 页面的 .json 配置文件 - 为指定页面 配置 特殊的窗口表现

> **当页面配置与全局配置 冲突 时，根据 就近原则 ，最终效果 以页面配置为准**

2.2 - 页面配置中常用的配置项

属性	类型	默认值	说明
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	是否开启当前页面下拉刷新
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

四、网络数据请求

1 - 小程序中网络数据请求的限制

出于 安全性 方面的考虑，小程序官方对 数据接口的请求 做出了如两个限制

- 只能请求 HTTPS 类型的接口
- 必须将 接口的域名 添加到 信任列表 中

2 - 服务器域名配置流程

- 登陆微信小程序管理后台
- 开发 -> 开发设置 -> 服务器域名
- 开始配置

2.1 - 配置注意事项

- 域名只支持 `https` 协议
- 域名不能使用 IP 地址（小程序的局域网IP 除外）或 localhost

- 端口配置可选
  - 配置端口 - 配置后只能向 指定端口 发起请求
  - 不配置端口 - 请求的 URL 中不能包含端口
- 域名必须经过 ICP 备案
- **出于安全考虑，api.weixin.qq.com 不能被配置为服务器域名，相关 API 也不能在小程序内调用**
- 不支持配置父域名，使用子域名
- 服务器域名一个月内最多可申请 5 次修改

2.2 - 发起 GET 请求

调用微信小程序提供的 **wx.request()** 方法，可以发起 GET 数据请求

```
wx.request({
    // 请求的接口地址，必须基于 https 协议
    url: 'https://www.chuxiu.cpm/api/get',
    // 请求的方式
    method: 'GET',
    // 发送到服务器的数据
    data: {
        name: 'coco',
        age: 18
    },
    // 请求成功之后的回调函数
    success: (res) => {
        console.log(res)
    }
})
```

2.3 - 发起 POST 请求

调用微信小程序提供的 **wx.request()** 方法，可以发起 GET 数据请求

```
wx.request({
    // 请求的接口地址，必须基于 https 协议
    url: 'https://www.chuxiu.cpm/api/post',
    // 请求的方式
    method: 'POST',
    // 发送到服务器的数据
    data: {
        name: 'coco',
        age: 18
    },
    // 请求成功之后的回调函数
    success: (res) => {
        console.log(res.data)
    }
})
```

2.4 - 在页面刚加载时请求数据

在页面的 onLoad 事件中调用获取数据的函数

```
/**
 * 生命周期函数--监听页面加载
 */
onLoad(options) {
  this.getRequest()
  this.postRequest()
},
```

2.5 - 跳过 request 合法域名校验

若 后端 仅 提供了 http 协议的接口、暂时没提供 https 协议的接口

可在微信开发者工具，临时开启 **开发环境不校验请求域名、TLS版本及 HTTPS 证书** 选项，跳过 request 合法域名的校验

> **仅限 在 开发与调试阶段 使用**

2.6 - 关于跨域和 Ajax 的说明

跨域

跨域问题 只存在于 基于浏览器的 Web 开发中

由于 小程序的宿主环境 不是浏览器，而是微信客户端，所以 小程序中不存在跨域的问题

Ajax

Ajax 技术的核心是依赖于浏览器中的 XMLHttpRequest 对象，由于 小程序的宿主环境是微信客户端，所以小程序中 不能叫做 “发起 Ajax 请求”，而是叫做 “ 发起网络数据请求 ”

五、WXS 脚本

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

应用场景： WXML 中无法调用在页面的 .js 中定义的函数，但 **WXML 中可以调用 WXS 中定义的函数**，因此，小程序中 WXS 的 典型应用场景 就是 “过滤器”

1 - WXS 和 JavaScript 的关系

虽然 WXS 的语法类似于 JavaScript，但 WXS 和 JavaScript 时完全不同的两种语言：

- WXS 有自己的数据类型
  - Number 、 String 、 Boolean 、 Object
  - Function 、 Array 、 Date 、 Regexp
- WXS 不支持类似于 ES6 及以上的语法形式
  - 支持 - var 定义变量、普通 function 函数等类似于 ES5 的语法
  - 不支持 - let、const、结构赋值、展开运算符、箭头函数、对象属性简写、etc
- WXS 遵循 CommonJs 规范
  - module 对象
  - require() 函数
  - module.exports 对象

2 - WXS 的特点

为降低 WXS （WeiXin Script）的学习成本，WXS 语言在设计时借鉴了大量 JavaScript 的语法

2.1 - WXS 中的方法不能作为组件的事件回调

下面的用法是 错误 的

```
<button bindtap="m2.toLower">按钮</button>
```

2.2 - 隔离性

隔离性 指 WXS 的运行环境和其他 JavaScript 代码是隔离的

- WXS 不能调用 JavaScript 中定义的函数
- WXS 不能调用 小程序 提供的 API

2.3 - 性能好

- 在 iOS 设备 上，小程序内的 WXS 比 JavaScript 块 2~20 倍
- 在 Android 设备 上，二者的运行效率 无差异

3 - 使用 WXS

3.1 - 内嵌 WXS 脚本

WXS 代码可编写在 WXML 文件中的 `<wxs>` 标签内，与 JavaScript 类似

WXML 文件中的每个 `<wxs></wsx>` 标签， 必须提供 module 属性，用来指定 当前 wxs 的模块名称，方便在 wxml 中访问模块中的成员

```
<view>{{m1.toUpper(username)}}</view>

<wxs module="m1">
    module.exports.toUpper = function(str) {
        return str.toUpperCase()
    }
</wxs>
```

3.2 - 外联 WXS 脚本

WXS 代码还可编写在 以 .wxs 为后缀名的文件内，与 JavaScript 类似

```
// tols.wxs
function toLower(str) {
    return str.toLowerCase()
}

module.exports = {
    // 小程序内不支持对象简写形式
    toLower: toLower
}
```

使用外联 WXS 脚本

在 WXML 中引入外联的 WXS 脚本时， 必须 为 `<wxs>` 标签添加 module 和 src 属性

- module - 用来指定 模块的名称
- src - 用来指定要引入的 脚本的路径，且 必须是绝对路径

```
<!-- 调用 m2 模块中的方法 -->
<view>{{m2.toLower(country)}}</view>
<!-- 引用外联的 wxs 脚本，并命名为 m2 -->
<wxs src="/utils/tools.wxs" module="m2"></wxs>
```

第三章、视图与逻辑

一、页面导航

页面导航 指 页面之间的相互跳转

例如，浏览器中实现页面导航的方式有如下两张

- `<a>` 链接
- `location.href`

1 - 小程序中实现页面导航的两种方式

- **声明式导航**
  - 在页面上 声明 一个 `<navigator>` 导航组件
  - 通过 点击 `<navigator>` 导航组件实现页面跳转
- **编程式导航**
  - 调用 小程序的 导航 API，实现页面的跳转

2 - 声明式导航

2.1 - 导航到 tabBar 页面

使用 `<navigator>` 组件跳转到指定的 **tabBar 页面**，需要指定 url 属性和 open-type 属性

- **url** 表示要跳转的 页面的地址，必须以 / 开头
- **open-type** 表示 跳转的方式，必须为 switchTab

```
<navigator url="/pages/index/index" open-type="switchTab">跳转到首页</navigator>
```

2.2 - 导航到非 tabBar 页面

使用 `<navigator>` 组件跳转到指定的 **非 tabBar 页面**，需要指定 url 属性和 open-type 属性

- **url** 表示要跳转的 页面的地址，必须以 / 开头
- **open-type** 表示 跳转的方式，为 navigate

> **navigate 可省略不写，因为 open-type 默认值为 navigate**

```
<navigator url="/pages/info/info" open-type="navigate"></navigator>
```

2.3 - 后退导航

使用 `<navigator>` 组件 后退 到 **上一页面或多级页面** ，， 则需指定 open-type 属性和 delta 属性

- **delta** 表示 回退的层数， 默认为 1， Number 类型

  > **delta 可省略不写**

- **open-type** 表示 跳转的方式，为 navigateBack

> **只能在 非tabBar页面 后退**

```
<navigator open-type="navigateBack">后退 1 页</navigator>
```

3 - 编程式导航

3.1 - 导航到 tabBar 页面

调用 wx.switchTab(Object object) 方法，可以跳转到 **tabBar 页面**

其中 Object object 参数对象的属性列表如下

属性	类型	必填	说明
url	string	是	需要跳转的 tabBar 页面的路径 路径后不能带参数
success	function	否	接口调用成功的回调函数
fail	function	否	接口调用失败的回调函数
complete	function	否	接口调用结束的回调函数（调用成功、失败都会执行）

```
<button bindtap="navigateToIndex">跳转至首页 - 编程式</button>
```

```
navigateToIndex() {
  wx.switchTab({
    url: '/pages/index/index',
  })
}
```

3.2 - 导航到非 tabBar 页面

调用 wx.navigateTo(Object object) 方法，可以跳转到 **非 tabBar 页面**

其中 Object object 参数对象的属性列表如下（ 与 wx.switchTab() 同 ）

属性	类型	必填	说明
url	string	是	需要跳转的 非 tabBar 页面的路径 路径后不能带参数
success	function	否	接口调用成功的回调函数
fail	function	否	接口调用失败的回调函数
complete	function	否	接口调用结束的回调函数（调用成功、失败都会执行）

```
<button bindtap="navigateToInfo">跳转到非 tabBar 页 - info</button>
```

```
navigateToInfo() {
  wx.navigateTo({
    url: '/pages/info/info',
  })
}
```

3.3 - 后退导航

调用 wx.navigateBack(Object object) 方法，可以 返回 到 **上一页面 或 多级页面**

其中 Object object 参数对象的属性列表如下

属性	类型	默认值	必填	说明
delta	number	1	是	需要跳转的 tabBar 页面的路径 路径后不能带参数
success	function		否	接口调用成功的回调函数
fail	function		否	接口调用失败的回调函数
complete	function		否	接口调用结束的回调函数（调用成功、失败都会执行）

```
<button bindtap="navigateBackTo">后退 - 编程式</button>
```

```
navigateBackTo() {
  wx.navigateBack()
}
```

二、导航传参

1 - 声明式导航传参

navigator 组件 的 url 属性用来指定将要跳转到的页面路径。

同时， **路径的后面还可携带参数**

- 参数 和 路径 之间使用 ? 分隔
- 参数键 与 参数值 用 = 相连
- 不同参数 用 & 分隔

```
<navigator url="/pages/info/info?name=coco&age=18">跳转到 info 页面</navigator>
```

2 - 编程时导航传参

调用 wx.navigateTo() 方法跳转到页面时， **路径的后面也可携带参数**

- 参数 和 路径 之间使用 ? 分隔
- 参数键 与 参数值 用 = 相连
- 不同参数 用 & 分隔

```
gotoInfo() {
    wx.navigateTo({
        url: '/pages/info/info?name=coco%age=19'
    })
}
```

3 - 参数获取

调用页面路由带的参数可以在目标页面的 `onLoad` 中通过 **形参** 获取

```
data: {
  // 导航传来的参数对象
  query: {}
},

onLoad(options) {
  console.log(options)
  this.setData({
    query: options
  })
}
```

三、页面事件

1 - 下拉刷新事件

下拉刷新 是移动端的专有名词，指得是通过手指在屏幕上的下拉的滑动操作，从而 重新加载页面数据 的行为

1.1 - 启动下拉刷新事件

- 全局开启下拉刷新
  - 在 app.json 的 window 节点，将 enablePullDownRefresh 设置为 true
- **局部开启下拉刷新**
  - 在页面的 .json 配置文件中，将 enablePullDownRefresh 设置为 true

> **实际开发中，推荐为 需要的页面单独开启下拉刷新的效果**

1.2 - 配置下拉刷新窗口的样式

属性	类型	默认值	说明
backgroundColor	HexColor	#ffffff	下拉刷新窗口的背景色
backgroundTextStyle	string	dark	下拉 loading 的样式，仅支持 `dark` / `light`

1.3 - 监听页面的下拉刷新事件

在页面的 .js 文件中，通过 onPullDownRefresh() 函数即可监听当前页面的下拉刷新事件

1.4 - 停止下拉刷新效果

当处理完下拉刷新后，下拉刷新的 loading 效果 不会主动消失

可调用 wx.stopPullDownRefresh() 停止当前页面的下拉刷新

```
onPullDownRefresh() {
  this.setData({
    count: 0
  })
  wx.stopPullDownRefresh()
}
```

2 - 上拉触底事件

上拉触底 是移动端的专有名词，通过手指在屏幕上的上拉滑动操作，从而 加载更多数据 的行为

2.1 - 配置上拉触底距离

上拉触底距离 指 触发上拉触底事件时，滚动条距离页面底部的距离

可在全局或页面的 .json 配置文件中，通过 onReachBottomDistance 属性配置上拉触底的距离

小程序默认的触底距离是 50 px，实际开发中，可根据需求修改这个默认值

2.1 - 监听页面的上拉触底事件

在页面的 .js 文件中，通过 onReachBottom() 函数即可监听当前页面的上拉触底事件

四、生命周期

1 - 生命周期

生命周期 （Life Cycle）指一个对象从 创建 -> 运行 -> 销毁 的整个阶段， 强调的是一个时间段

小程序中， **生命周期** 分为两类

- **应用生命周期**
  - 特指 **小程序** 从 启动 -> 运行 -> 销毁 的过程
- **页面生命周期**
  - 特指小程序中，每个 **页面** 从 加载 -> 渲染 -> 销毁 的过程

其中， 页面 的生命周期 范围较小 ， 应用程序 的生命周期 范围较大

2 - 生命周期函数

生命周期函数 是由小程序框架提供的 内置函数 ，会伴随着生命周期， 自动按次序执行

生命周期函数的作用： 允许程序员 在特定的时间点，执行某些特定的操作

> **生命周期 强调的是 时间段； 生命周期函数 强调的是 时间点**

小程序中， **生命周期函数** 分为两类

- **应用生命周期**
  - 特指 **小程序** 从 启动 -> 运行 -> 销毁 期间依次调用的那些 函数
- **页面生命周期**

- 特指小程序中，每个 **页面** 从 加载 -> 渲染 -> 销毁 期间依次调用的那些 函数

3 - 应用的生命周期函数

小程序的 **应用生命周期函数** 需要在 app.js 中进行声明

```
// app.js
App({
    // 小程序初始化完成时，执行此函数，全局只触发一次，可以做些初始化的工作
    onLaunch() {

    },
    // 小程序启动，或从后台进入前台显示时触发
    onShow() {

    },
    // 小程序从前台进入后台时触发
    onHide() {

    }
})
```

4 - 页面的生命周期函数

小程序的 **页面生命周期函数** 需要在页面的 .js文件 中进行声明

```
// 页面的 .js 文件
Page({
    // 监听页面加载，一个页面只调用 1 次
    onLoad(options) {

    },
    // 监听页面显示
    onShow() {

    },
    // 监听页面初次渲染完成，一个页面只调用 1 次
    onReady() {

    },
    // 监听页面隐藏
    onHide() {

    },
    // 监听页面卸载，一个页面只调用 1 次
    onUnload() {

    }
})
```

5 - 组件的生命周期函数

组件的生命周期，指的是组件自身的一些函数，这些函数在特殊的时间点或遇到一些特殊的框架事件时被自动触发

5.1 - 组件全部的生命周期函数

小程序组件可用的全部生命周期函数

生命周期	参数	描述
created	无	在 组件实例刚刚被创建时 执行
attached	无	在 组件实例进入页面节点树时 执行
ready	无	在组件在视图层布局完成后执行
moved	无	在组件实例被移动到节点树另一个位置时执行
detached	无	在 组件实例被从页面节点树移除时 执行
error	`Object Error`	每当组件方法抛出错误时执行

5.2 - 组件主要的生命周期函数

其中，最重要的生命周期是 `created` `attached` `detached`，包含一个组件实例生命流程的最主要时间点

- created - 组件实例刚刚被创建好时， `created` 生命周期被触发
  - 此时，组件数据 `this.data` 就是在 `Component` 构造器中定义的数据 `data`
  - **此时还不能调用 setData**
  - 通常情况下，这个生命周期只应该用于给组件 `this` 添加一些自定义属性字段
- attached - 在组件完全初始化完毕、进入页面节点树后， `attached` 生命周期被触发
  - 此时， `this.data` 已被初始化为组件的当前值
  - 这个生命周期很有用，绝大多数初始化工作可以在这个时机进行
- detached - 在组件离开页面节点树后， `detached` 生命周期被触发
  - 退出一个页面时，如果组件还在页面节点树中，则 `detached` 会被触发
  - 此时，适合做些清理性质的工作

5.3 - 定义组件生命周期函数

小程序中，组件的的生命周期也可以在 `lifetimes` 字段内进行声明（这是 推荐的方式，其 优先级最高 ）

也可以直接定义在 `Component` 构造器的第一级参数中（旧的定义方式）

```
Component({
  lifetimes: {
    attached: function() {
      // 在组件实例进入页面节点树时执行
```

```
    },
    detached: function() {
      // 在组件实例被从页面节点树移除时执行
    },
  },
  // 以下是旧式的定义方式，可以保持对 <2.2.3 版本基础库的兼容
  attached: function() {
    // 在组件实例进入页面节点树时执行
  },
  detached: function() {
    // 在组件实例被从页面节点树移除时执行
  },
  // ...
})
```

在 behaviors 中也可以编写生命周期方法，同时不会与其他 behaviors 中的同名生命周期相互覆盖

6 - 组件所在页面的生命周期

有时， 自定义组件的行为依赖于页面状态的变化 ，此时，就需用到 组件所在页面的生命周期

在 `pageLifetimes` 定义段中定义

其中可用的生命周期包括

生命周期	参数	描述
show	无	组件所在的页面 被展示 时执行
hide	无	组件所在的页面 被隐藏 时执行
resize	`Object Size`	组件所在的页面 尺寸变化 时执行

示例代码:

```
Component({
  pageLifetimes: {
    show: function() {
      // 页面被展示
    },
    hide: function() {
      // 页面被隐藏
    },
    resize: function(size) {
      // 页面尺寸变化
    }
  }
})
```

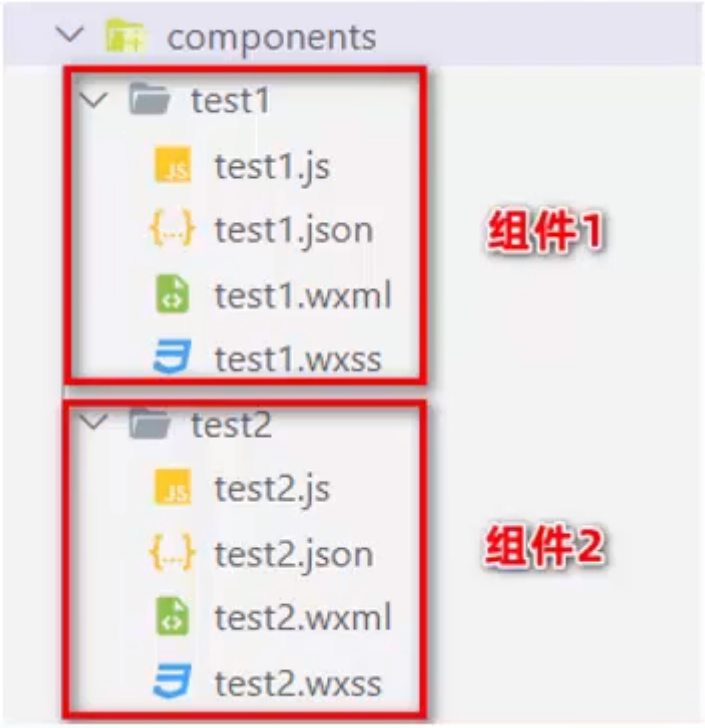
第四章、自定义组件

一、组件的创建与引用

1 - 创建组件

- 在项目的根目录中，创建 components -> test 文件夹
- 在 test 文件夹，点击“新建 Component”
- 输入组件名回车，会自动生成组件对应的 4 个文件，一个自定义组件由 `json` `wxml` `wxss` `js` 4个文件组成

> **为保证目录结构的清晰，可把不同的组件，存放到单独的目录中**



2 - 编写组件

- 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` 可将这一组文件设为自定义组件）
  - 自动生成组件时，默认为 true

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

- 同时，还要在 `wxml` 文件中编写组件模板，在 `wxss` 文件中加入组件样式
  - > **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

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

3 - 引用组件

组件的引用方式分为 “ 局部引用 ” 和 “ 全局引用 ”

- 局部引用 - 组件只能在当前被引用的页面内使用
- 局部引用 - 组件可以在每个小程序页面中使用

3.1 - 局部引用

在页面的 .json 配置文件中引用组件的方式

```
// 页面的 .json 文件中
{
    "usingComponents": {
        "my-test": "/components/test/test"
    }
}
```

```
<!-- 页面的 .wxml 文件中 -->
<my-test></my-test>
```

3.2 - 全局引用

在 app.json 全局配置文件中引用组件的方式

```
// 页面的 .json 文件中
{
    "pages": [],
    "window": {}
    "usingComponents": {
        "my-test2": "/components/test2/test2"
    }
}
```

```
<!-- 页面的 .wxml 文件中 -->
<my-test2></my-test2>
```

3.3 - 全局引用 vs 局部引用

根据组件的 使用频率 和 范围 ， 选择合适的使用方式

- 某组件在 多个页面中经常被用到 ， 建议进行” 全局引用 “
- 某组件 只 在 特定的页面中被用到 ， 建议进行” 局部引用 “

4 - 组件和页面的区别

组件和页面的 .js 与 .json 文件有明显的不同：

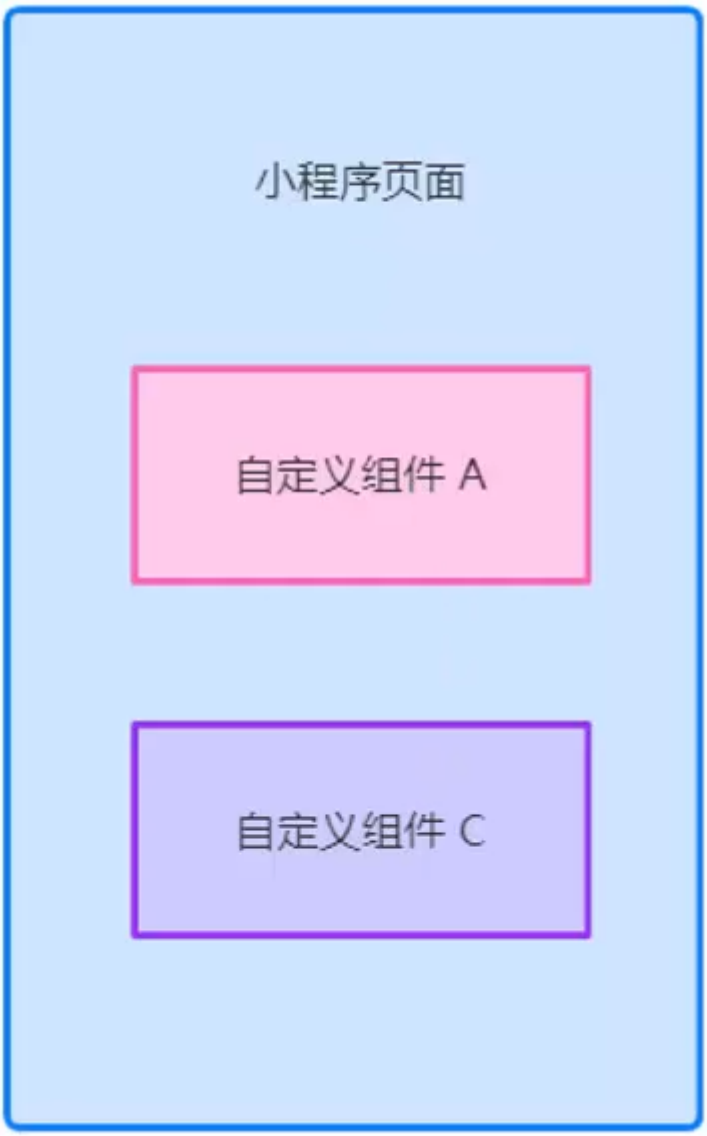
- 组件的 .json 文件需要声明 "component": true 属性
- 组件的 .js 文件中调用的是 Component() 函数
- 组件的事件处理函数需要定义到 methods 节点中

二、组件的样式

1 - 组件样式隔离

默认情况下，自定义组件的样式只对当前组件生效，不会影响到组件之外的 UI 结构

- 组件 A 的样式 不会影响 组件 C 的样式
- 组件 A 的样式 不会影响 小程序页面 的样式
- 小程序页面 的样式 不会影响 组件 A 和 C 的样式



> **防止外界样式影响组件内部的样式**
>
> **防止组件的样式影响外界的样式**

1.1 - 组件样式隔离的注意点

- app.wxss 中的全局样式对组件 无效
  - 除非
    - `app.wxss` 或页面的 `wxss` 中使用了 标签名选择器 （或一些其他特殊选择器）来直接指定样式
    - 指定特殊的样式隔离选项 `styleIsolation`
- 只有 class 选择器 会有样式隔离效果，id 选择器、属性选择器、标签选择器不受样式隔离的影响

1.2 - 修改组件的样式隔离选项

默认情况下，自定义组件的 样式隔离特性 能够 防止组件内外样式互相干扰的问题

但有时，希望外界能够控制组件内部的样式，此时，可通过 styleIsolation 修改组件的样式隔离选项

```
// 在组件的 .js 文件中新增如下配置
Component({
  options: {
    styleIsolation: 'isolated'
  }
})

// 或在组件的 .json 文件中新增如下配置
{
    "styleIsolation": "isolated"
}
```

styleIsolation 的可选值

可选值	默认值	说明
isolated	是	表示 启用样式隔离，在自定义组件内外，使用 class 指定的样式将 不会相互影响
apply-shared	否	表示 页面 wxss 样式将影响到自定义组件，但自定义组件 wxss 中 指定的样式不会影响到页面
shared	否	表示 页面 wxss 样式将影响到自定义组件，自定义组件wxss 中 指定的样式会影响到页面和其他设置了 apply-shared 或 shared 的自定义组件

> **apply-shared - 单向影响**
>
> **shared - 双向影响**

三、组件的数据、方法和属性

1 - data 数据

自定义组件中， 用于组件模版渲染 的 私有数据 ， 需要定义到 data 节点 中

```
Component({
    // 组件的初始数据
    data: {
        count: 0
    }
})
```

2 - methods 方法

自定义组件中， 事件处理函数 和 自定义方法 ， 需要定义到 methods 节点 中

```
Component({
```

```
    // 组件的方法列表 【包含事件处理函数和自定义方法】
    methods:{
        // 事件处理函数
        addCount() {
            this.setData({
                count: this.data.count + 1
            })
            // 通过 this 直接调用自定义方法
            this._showCount()
        },
        // 自定义方法以 _ 开头
        _showCount() {
            wx.showToast({
                title: `count 的值为 ${this.data.count}`,
                icon: 'none'
            })
        }
    }
})
```

自定义方法可通过 _标识符 与事件处理函数区分

3 - properties 属性

自定义组件中， properties 是组件的对外属性， 用来接受外界传递到组件的数据

```
Component({
    // 属性定义
    properties: {
        // 完整定义属性的方式【当需要指定属性默认值时，建议使用此方式】
        max: {
            type: Number,
            value: 10
        },
        // 简化定义属性的方式【不需指定属性默认值时，可以使用简化方式】
        max: Number
    }
})
```

```
<my-test></my-test>
```

4 - data 和 properties 的区别

小程序中，properties 属性 和 data 数据 的用法相同，它们都是 可读可写 的

- data 更倾向于 存储组件的私有数据
- properties 更倾向于 存储外界传递到组件中的数据

```
properties: {max: Number},

data: {
    count: 0
},

methods: {
    showInfo() {
      console.log(this.data)            // => {count: 0, max: 9}
      console.log(this.properties)      // => {count: 0, max: 9}
      console.log(this.data === this.properties)    // => true
    }
}
```

> **Vue 中 props 仅可读**

5 - 使用 setData() 修改 properties 的值

由于 data 数据 properties 属性 在本质上无任何区别，因此 properties 属性的值也可用于页面渲染

或使用 setData() 修改 properties 的值

```
<view>max 的值是 {{max}}</view>
<button bindtap="addMax">add max</button>
```

```
Component({
    properties: {
        max: Number
    },
    methods: {
        addMax() {
            this.setData({
                max: this.properties.max + 1
          })
        },
    }
})
```

四、数据监听器

数据监听器 可以用于 监听 和 响应 任何 属性和数据字段的变化， 从而执行特定的操作

作用类似于 vue 中的 watch 侦听器

语法格式:

```
Component({
    observers: {
        '字段A, 字段B': function(字段A的新值, 字段B的新值) {
            // do something
        }
    }
})
```

1 - 数据监听器的基本用法

```
<view>{{num1}} + {{num2}} = {{sum}}</view>
<button type="primary" size="mini" bindtap="addNum1">num1 + 1</button>
<button type="primary" size="mini" bindtap="addNum2">num2 + 2</button>
```

```
Component({
    data: { num1: 0, num2: 0, sum: 0},

    methods: {
      addNum1() {
        this.setData({
          num1: this.data.num1 + 1
        })
      },

      addNum2() {
        this.setData({
          num2: this.data.num2 + 1
        })
      }
    },

    observers: {
        'num1, num2': function(num1, num2) {
            this.setData({
                sum: num1 + num2
            })
        }
    }
})
```

2 - 监听对象属性的变化

数据监听器支持监听 对象 中 单个 或 多个属性 的变化

语法格式:

```
Component({
    observer: {
        '对象.属性A，对象.属性B': function(属性A的新值，属性B的新值) {
            // do something
        }
    }
})
```

触发情况

- 为 属性A 赋值 - 使用 setData() 设置 this.data.对象.属性A 时触发
- 为 属性B 赋值 - 使用 setData() 设置 this.data.对象.属性B 时触发
- 为 对象 赋值 - 使用 setData() 设置 this.data.对象 时触发

3 - 监听对象中所有属性的变化

如果需要监听 所有子数据字段的变化，可以使用通配符 `**`

语法格式:

```
Component({
    observer: {
        '对象.***': function(对象) {
            // do something
        }
    }
})
```

特别地，仅使用通配符 `**` 可以监听全部 setData

```
Component({
  observers: {
    '**': function() {
      // 每次 setData 都触发
    },
  },
})
```

五、纯数据字段

纯数据字段 是一些 不用于界面渲染的 data 字段，可以用于提升页面更新性能

应用场景 - 既不会展示在界面上，也不会传递给其他组件，仅仅在当前组件内部使用，此时，可以指定这样的数据字段为“纯数据字段”

> **仅仅被记录在 `this.data` 中，而不参与任何界面渲染过程，这样有助于提升页面更新性能**

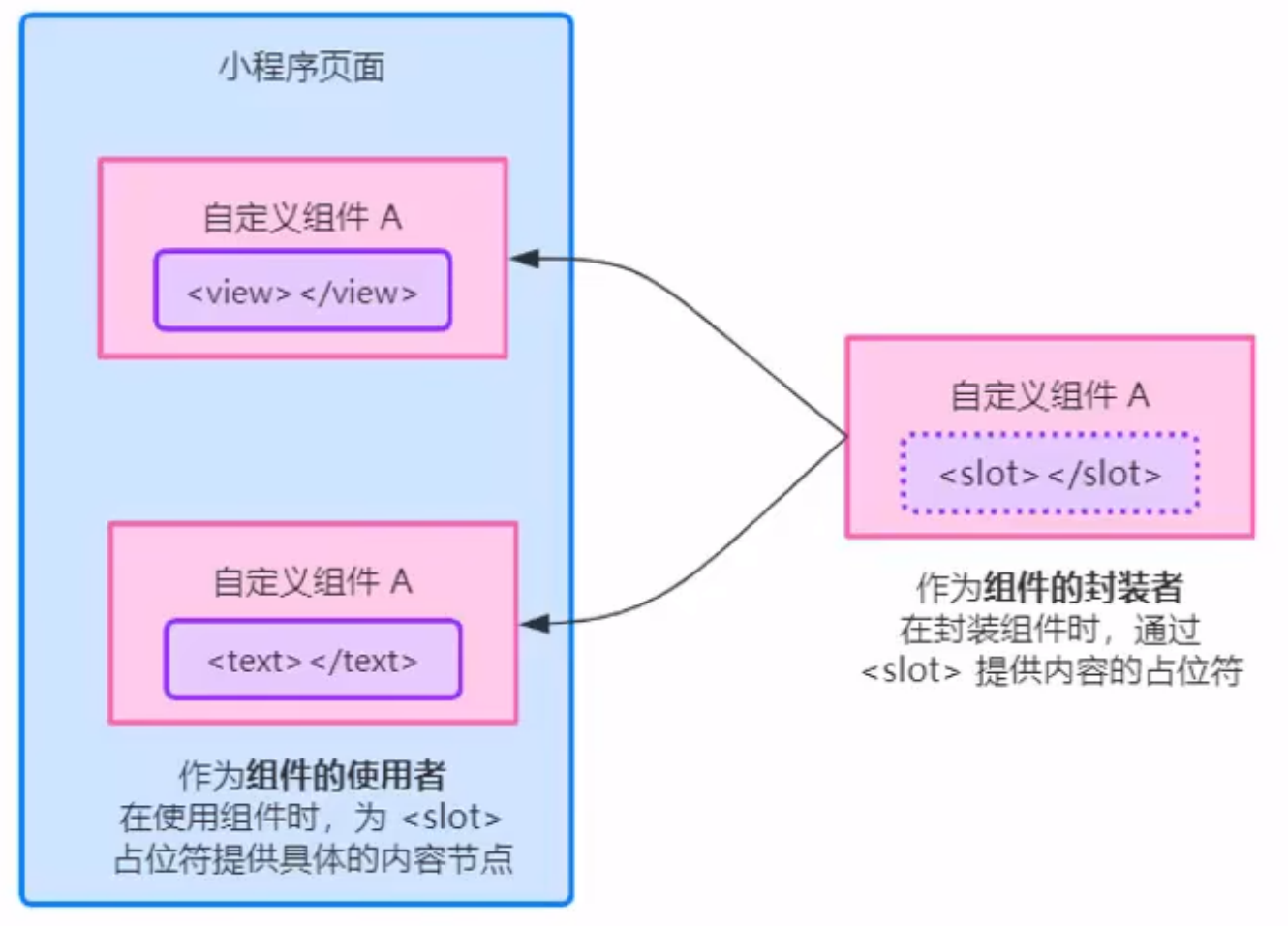
1 - 指定“纯数据字段”的方法

在 `Component` 构造器的 `options` 定义段中指定 `pureDataPattern` 为一个正则表达式，字段名符合这个正则表达式的字段将成为纯数据字段

```
Component({
    options: {
        // 指定所有 _ 开头的数据字段为纯数据字段
        pureDataPattern: /^_/
    }
    data: {
        a: true,    // 普通数据字段
        _b: true,   // 纯数据字段
    }
    methods: {
        myMethod() {
          this.data._b // 纯数据字段可以在 this.data 中获取
          this.setData({
            c: true, // 普通数据字段
            _d: true, // 纯数据字段
          })
        }
  }
})
```

六、插槽

在自定义组件的 wxml 结构中，可以提供一个 `<slot>` 节点（插槽），用来承载组件使用者提供的 wxml 结构



1 - 单个插槽

小程序中，默认每个自定义组件只允许使用一个 `<slot>` 进行占位

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

```
<!-- 组件的封装者 -->
<view class="wrapper">
  <view>这是组件的内部节点</view>
  <!-- 对于不确定的内容，可使用 <slot> 占位 -->
  <slot></slot>
</view>

<!-- 组件的使用者 -->
<component-tag-name>
  <!-- 这部分内容将被放置在组件 <slot> 的位置上 -->
  <view>这是插入到组件 slot 中的内容</view>
</component-tag-name>
```

2 - 多个插槽

小程序中，需要使用多个 slot 时，可以在组件 js 中声明启用

在 `options` 节点，通过 `multipleSlots: true` 启用多 slot 支持

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

2.1 - 多个插槽的使用

定义多插槽:

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>
```

使用多插槽:

使用时，用 `slot` 属性来将节点插入到不同的 slot 上

```
<!-- 引用组件的页面模板 -->
<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>
```

七、组件间通信

1 - 父子组件之间通信的 3 种方式

- 属性绑定
  - 父 -> 子，仅能设置 JSON 兼容的数据
- 事件绑定
  - 子 -> 父，可以传递任意数据
- 获取组件实例
  - 父组件可通过 this.selectComponent() 获取子组件实例对象
  - 这样就可以直接访问子组件的任意数据和方法

2 - 属性绑定

属性绑定 用于实现 父向子传值 ， 而且 只能传递普通类型的数据 ， 无法将方法传递给子组件

父组件在 子组件标签 通过 属性绑定 ， 向子组件的属性传递动态数据

```
// 父组件的 wxml 结构
<my-test count="{{count}}"></my-test>
<view> ~~~~~~ </view>
<view>父组件中，count的值为：{{count}}</view>

// 父组件的 data 节点
data: {
    count: 0
}

```

子组件在 properties 节点中 声明对应的属性并使用

```
// 子组件的 wxml 结构
<text>子组件中，count值为：{{count}}</text>

// 子组件的 properties 节点
properties: {
    count: Number
}
```

3 - 事件绑定

事件绑定 用于实现 子 -> 父 传值，可以传递 任何类型的数据

实现步骤：

- 在 **父组件** 的 .js 中 - 定义一个函数，这个函数 即将 通过自定义事件的形式，传递给子组件

  ```
  // 父组件.js
  // 将来，这个方法会被传递给子组件，供子组件进行调用
  syncCount(e) {
    e.detail // 自定义组件触发事件时提供的 detail 对象
  },
  ```

- 在 **父组件** 的 wxml 中 - 通过 自定义事件 的形式，将步骤 1 定义的函数引用，传递给子组件

```wxml
<!-- 使用 bind:自定义事件名称（推荐，结构清晰） -->
<my-test count="{{count}}" bind:sync="syncCount"></my-test>
<!-- 使用 bind自定义事件名称-->
<my-test count="{{count}}" bindsync="syncCount"></my-test>
```

- 在 **子组件** 的 .js 中 - 通过调用 this.triggerEvent('自定义事件名称', {/* 参数对象 */}) ，将数据发送到父组件

```wxml
<!-- 子组件的 wxml 结构 -->
<view>子组件中，count的值为：{{count}}</view>
<button type="primary" bindtap="addCount">+ 1</button>
```

```js
// 子组件的 js 代码
methods: {
  addCount() {
    this.setData({
      count: this.properties.count + 1
    })
    this.triggerEvent('sync', {value: this.properties.count})
  },
}
```

- 在 **父组件** 的 .js 中 - 通过 e.detail 获取到子组件传递过来的数据

```js
syncCount(e) {
  // console.log('syncCount')
  this.setData({
    count: e.detail.value
  })
},
```

4 - 获取组件实例

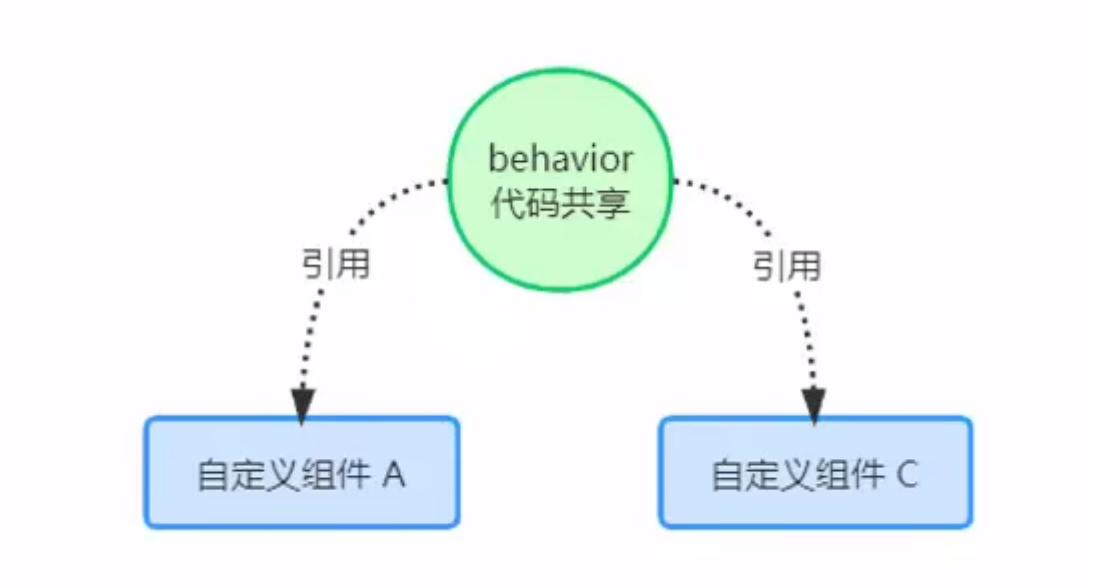
可在父组件里调用 this.selectComponent(' 选择器（id或class）') ，获取子组件的实例对象，从而直接访问子组件的任意数据和方法

```wxml
<!-- 父组件 wxml 结构 -->
<my-test count="{{count}}" bind:sync="syncCount" class="customA" id="cA"></my-test>
<button bindtap="getChild">获取子组件实例</button>
```

```
// 父组件的 js 文件
getChild() {
    const child = this.selectComponent('.customA')
    // 调用子组件的 setData()
    child.setData({
        count: child.properties.count + 1
    })
    // 调用子组件的 addCount()
    child.addCount()
}
```

八、behaviors

behaviors 是小程序中， 用于实现组件间代码共享 的特性，类似于 Vue.js 中的“mixins”



1 - behaviors 的工作方式

每个 `behavio` 可以包含一组 属性、数据、生命周期函数和方法

组件 引用 `behavior` 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 也可相互引用

2 - 创建 behavior

调用 Behavior(Object object) 方法，即可创建一个 共享的 behavior 实例对象， 供所有的组件使用

示例代码:

在 项目根目录 新建 behaviors 文件夹， 新建 my-behavior.js 文件

```
// my-behavior.js

module.exports = Behavior({
    // 属性节点
    properties: {},
    // data 节点
    data: {
        username: 'coco'
    },
    // methods 节点
    methods: {},
    // lifetime 节点
    lifetime: {}
})
```

3 - 导入并使用 behavior

在组件中，使用 require() 方法导入需要的 behavior， 挂载后即可访问 behavior 中的数据与方法

```
// my-component.js

// 使用 require() 导入需要的自定义 behavior 模块
const myBehavior = require('../../behaviors/my-behavior')

Component({
    // 将导入的 behavior 实例对象，挂载到 behaviors 数据节点中，即可生效
    behaviors: [myBehavior],
    // 组件其他节点……
})
```

组件中使用：

```
<!-- my-component.wxml -->

<!-- 输出 coco -->
<view>在 behavior 中定义的 username 是：{{username}}</view>
```

4 - behavior 中可用的节点

定义段	类型	是否必填	描述
properties	Object Map	否	同组件的属性
data	Object	否	同组件的数据
methods	Object	否	同自定义组件的方法
behaviors	String Array	否	引入其它的 `behavior`
created	Function	否	生命周期函数
attached	Function	否	生命周期函数
ready	Function	否	生命周期函数
moved	Function	否	生命周期函数
detached	Function	否	生命周期函数

5 - 同名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

- 同名的 数据字段 （ data ）
  - 若同名的数据字段都是对象类型，会进行对象合并
  - 其余情况会进行按 优先级 数据覆盖
    - `组件` > `引用者 behavior` > `被引用的 behavior` 、 `靠后的 behavior` > `靠前的 behavior`
- 同名的 属性和方法 （ properties 和 methods ）
  - 组件本身有，则覆盖 `behavior` 中的同名属性或方法
  - 组件本身无，则在组件的 `behaviors` 字段中定义靠后的 `behavior` 的属性或方法会覆盖靠前的同名属性或方法
  - 在 2 的基础上，若存在嵌套引用 `behavior` 的情况，则规则为： `引用者 behavior` 覆盖 `被引用的 behavior` 中的同名属性或方法
- 同名的 生命周期函数
  - `behavior` 优先于组件执行
  - `被引用的 behavior` 优先于 `引用者 behavior` 执行
  - `靠前的 behavior` 优先于 `靠后的 behavior` 执行

第五章、npm 包

一、使用 npm 包

1 - 小程序对 npm的支持和限制

目前，小程序中已经支持使用 npm 安装第三方包，从而提高小程序的开发效率

但是，小程序中使用 npm 包有如下 3 个限制：

- 不支持依赖于 Node.js 内置库 的包
- 不支持依赖于 浏览器 内置对象 的包
- 不支持依赖于 C++ 插件 的包

> **能供小程序使用的包 为数不多**

二、Vant Weapp UI组件库

Vant Weapp 是有赞前端团队开源的一套 小程序 UI 组件库，助力开发者快速搭建小程序应用

使用 MIT 开源许可协议，对商业使用比较友好

官方文档地址

1 - 安装 vant 组件库

- 初始化

```
$ npn init -y
```

- 通过 npm 安装

```
# 在小程序 package.json 所在的目录中执行命令
$ npm i @vant/weapp -S --production
```

- 修改 app.json
  - 将 app.json 中的 `"style": "v2"` 去除
- 修改 project.config.json

```
{
  ...
  "setting": {
    ...
    "packNpmManually": true,
    "packNpmRelationList": [
      {
        "packageJsonPath": "./package.json",
        "miniprogramNpmDistDir": "./"
      }
    ]
  }
}
```

- 构建 npm 包
  - 点击开发者工具中的菜单栏：工具 --> 构建 npm

2 - 使用 Vant 组件

安装完 Vant 组件库之后，可在 app.json 的 usingComponents 节点中引入需要的组件，即可在 wxml 中直接使用组件

示例代码：

```
// app.json

"usingComponents": {
    "van-button": "@vant/weapp/button/index"
}
```

```
<!-- 页面的 wxml 结构 -->
<van-button type="primary">按钮</van-button>
```

3 - 定制全局主题样式

Vant Weapp 使用 **CSS 变量** 来实现定制主题样式

3.1 - CSS 变量

基本用法：

- 声明一个自定义属性，属性名需要以两个减号（`--`）开始

```
element {
  --main-bg-color: brown;
}

/* 定义在根伪类 :root 下 , 全局访问 */

:root {
  --main-bg-color: brown;
}

html {
  --main-bg-color: brown;
}
```

- 使用一个局部变量时用 `var()` 函数包裹以表示一个合法的属性值
  - 用 `var()` 函数可以定义多个**备用值**(fallback value)
    - 第一个参数是自定义属性的名称

- 如果提供了第二个参数，则表示备用值

```
element {
    background-color: var(--main-bg-color);
}

/* 定义备用值 */
.two {
  color: var(--my-var, red); /* Red if --my-var is not defined */
}
```

3.2 - 定制 Vant 的全局主题样式

在 app.wxss 中，写入 CSS 变量，即可对全局生效

所有可用的颜色变量

示例代码：

```
/* app.wxss */
page {
    /* 定制警告按钮的背景色和边框色 */
    --button-danger-background-color: #C00000;
    --button-danger-border-color: #D60000;
}
```

三、API Promise 化

1 - 基于回调函数的异步 API 的缺点

默认情况下，小程序官方提供的 异步 API 都是 基于回调函数 实现的

```
wx.request({
    method: '',
    url: '',
    data: {},
    success: () => {},
    fail: () => {},
    complete: () =< {}
})
```

缺点：容易造成 回调地狱 的问题，代码的 可读性、维护性 差！

2 - 什么是 API Promise 化

API Promise 化，值 通过额外的配置，将官方提供的、基于回调函数的异步 API， 升级改造为 基于 Promise 的异步 API，从而提高代码的 可读性、维护性， 避免回调地狱 的问题

3 - 实现 API Promise 化

小程序中，实现 API Promise 化主要依赖于 miniprogram-api-promise 这个第三方得到 npm 包

安装使用步骤：

- 安装

```
$ npm install --save miniprogram-api-promise
```

- 构建 npm
- 在小程序入口文件 app.js 中，只需调用一次 promisifyAll() 方法，即可实现异步 API 的 Promise 化

```
// app.js

import { promisifyAll } from 'miniprogram-api-promise'

const wxp = wx.p = {}
// promisify all wx's apo
promisifyAll(wx, wxp)
```

4 - 调用 Promise 化之后的异步 API

示例代码：

```
<!-- 页面的 wxml 结构 -->
<vant-button type="danger" bindtap="getInfo">vant 按钮</vant-button>
```

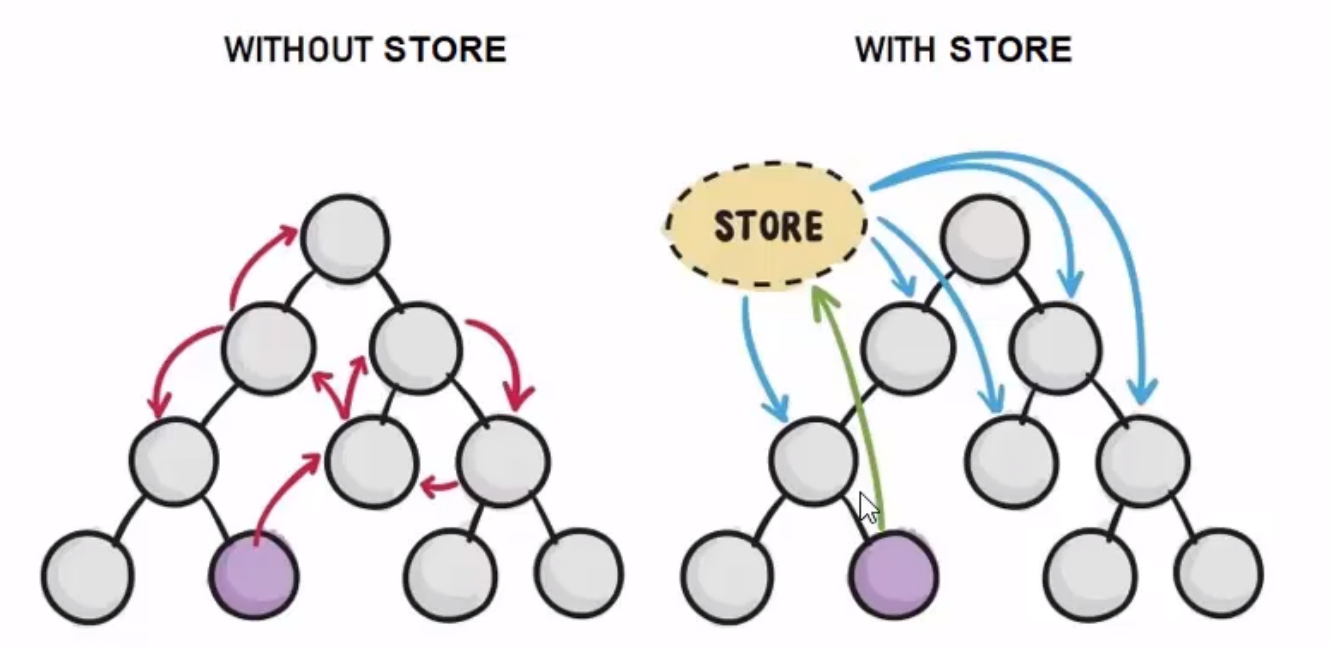
```
// 页面的 js 文件，定义对应的 tap 事件处理函数
async getInfo() {
    const { data: res } = await wx.p.request({
        method: 'GET',
        url: 'https://www.escook.cn/api/get',
        data: { name: 'coco', age: 20}
    })

    console.log(res)
}
```

第六章、全局数据共享

一、全局数据共享

全局数据共享（又称：状态管理）是为了解决 组件之间数据共享 的问题

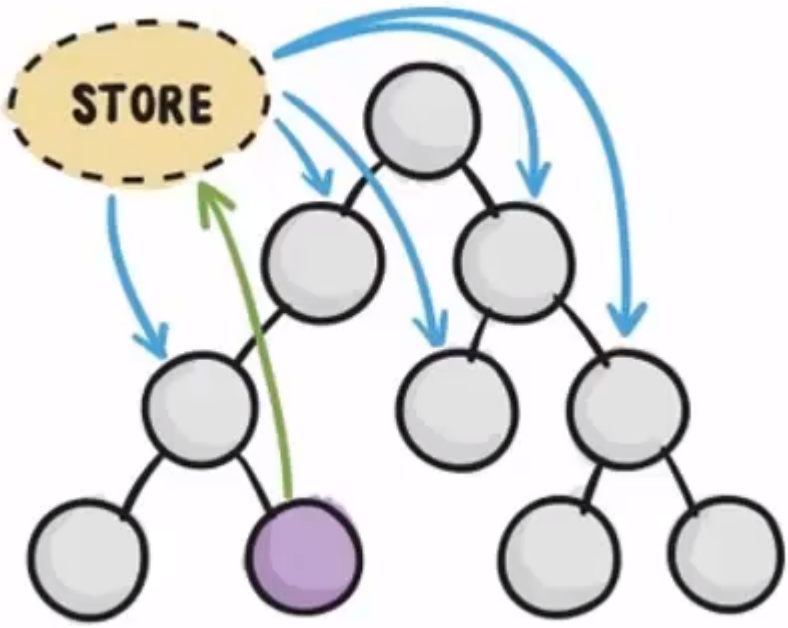
开发者常用的全局数据共享方案有：Vuex、Redux、Mobx 等



1 - 小程序中的全局数据共享方案

小程序中，可使用 mobx-miniprogram 配合 mobx-miniprogram-bindings 实现全局数据共享

- mobx-miniprogram - 用来 创建 Store 实例对象
- mobx-miniprogram-bindings - 用来 把 Store 中的共享数据或方法，绑定到组件或页面中使用



2 - MobX

2.1 - 安装 Mobx 相关的包

- 在项目中运行如下命令，安装 MobX 相关的包

```
$ npm i --save mobx-miniprogram mobx-miniprogram-bindings
```

- 构建 npm

2.2 - 创建 MobX 的 Store 实例

在 项目根目录 新建 store 文件夹，并在文件夹中新建 store.js 文件

在 store.js 中专门创建 Store 的实例对象

```
import { action, observable } from "mobx-miniprogram";

export const store = observable({
  // 挂载需要共享的数据字段
  // 数据字段
  numA: 1,
  numB: 2,
  // 计算属性 * get 指明只读 *
  get sum() {
    return this.numA + this.numB;
  },
  // actions 方法，用来修改 store 中的数据
  updateNum1: action(function (step) {
    this.numA += step;
  }),
  updateNum2: action(function (step) {
    this.numB += step;
  }),
});

```

2.3 - 将 Store 中的成员绑定到页面中

- 在页面 js 中导入需要的成员
- 在 onLoad() 中进行绑定
- 在 onUnload 中进行清理

```
import { createStoreBindings } from 'mobx-miniprogram-bindings'
import { store } from '../../store/store'

Page({
  onLoad: function () {
    this.storeBindings = createStoreBindings(this, {
```

```
      store,
      fields: ['numA', 'numB', 'sum'],
      actions: ['updateNum']
    })
  },
  onUnload: function () {
    this.storeBindings.destroyStoreBindings()
  }
})

```

2.4 - 在页面上使用 Store 中的成员

```
<!-- 页面的 wxml 结构 -->
<view>{{numA}} + {{numB}} = {{sum}}</view>
<van-button type="primary" bindtap="btnHandler1" data-step="{{1}}">numA + 1</van-button>
<van-button type="warning" bindtap="btnHandler1" data-step="{{-1}}">numA - 1</van-button>
```

```
// button tap事件的处理函数
btnHandler1(e) {
  this.updateNum1(e.target.dataset.step)
},
```

2.5 -将 Store 中的成员绑定到组件中

- 在组件 js 文件中按需导入需要的成员
- 在 behaviors 数组通过 storeBindingsBehavior 实现自动绑定
- 在 storeBindings 节点声明对象

```
// 组件的 js 文件

import { storeBindingsBehavior } from 'mobx-miniprogram-bindings'
import { store } from '../../store/store'

Component({
    behaviors: [storeBindingsBehavior],
    storeBindings: {
      store,  // 指定要绑定的 Store
      fields: { // 指定要绑定的数据字段
        numA: () => store.numA,  // 绑定字段的第 1 种方式
        numB: (store) => store.numB,  // 绑定字段的第 2 种方式
        sum: 'sum'  // 绑定字段的第 3 种方式
      },
      actions: {  // 指定要绑定的方法
        updateNum2: 'updateNum2'
      }
```

```
    },
})
```

2.6 - 在组件中使用 Store 中的成员

```
<!-- 组件的 wxml 结构 -->

<view>{{numA}} + {{numB}} = {{sum}}</view>
<van-button type="primary" bindtap="btnHandler2" data-step="{{1}}">numA + 1</van-button>
<van-button type="warning" bindtap="btnHandler2" data-step="{{-1}}">numA - 1</van-button>
```

```
// 组件的方法列表

methods: {
  btnHandler2(e) {
    this.updateNum2(e.target.dataset.step)
  }
}
```

第七章、分包

分包 是一个 完整的小程序项目，按照需求 划分为不同的子包，在构建时打包成 不同的分包，用户在使用时 按需进行加载

一、分包

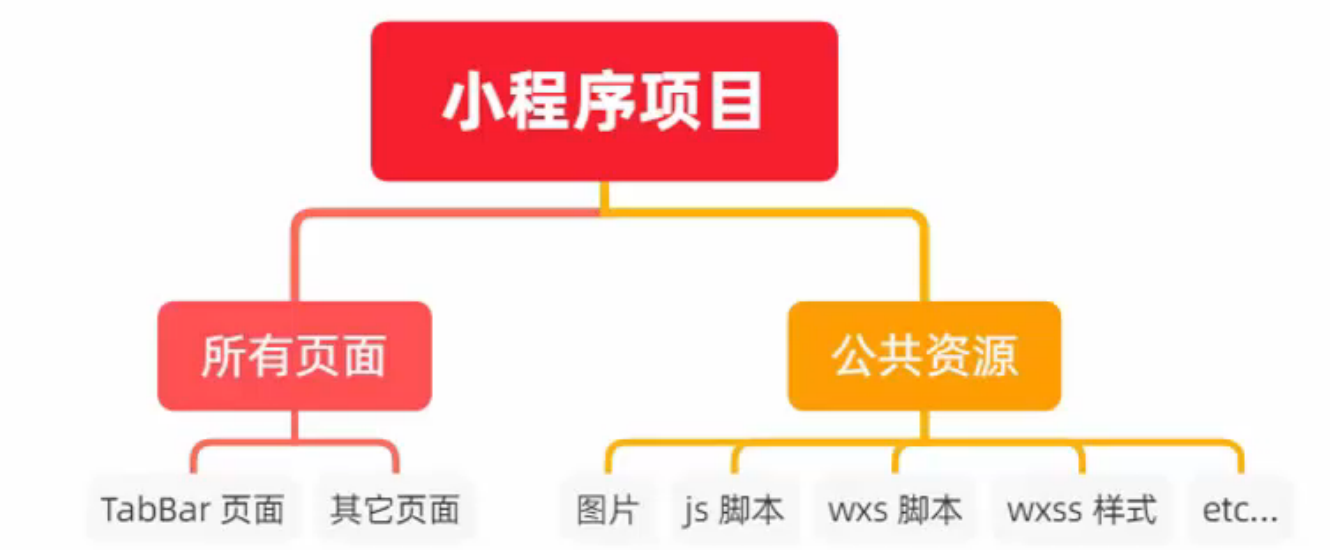
1 - 分包介绍

1.1 - 分包的好处

- 可以 优化小程序首次启动的下载时间
- 在 多团队共同开发 时可以更好地 解耦协同

1.2 - 分包前的项目构成

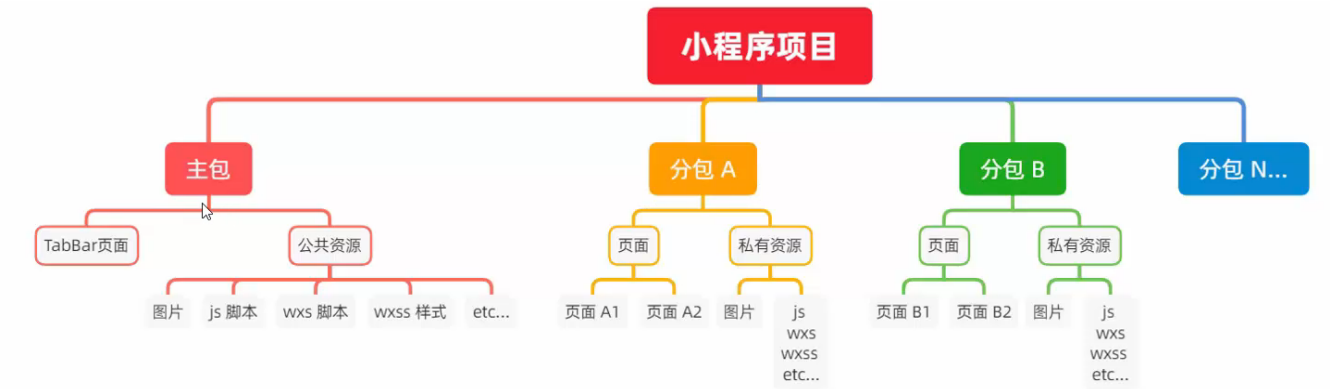
分包前，小程序项目中 所有的页面 和 资源 都被打包到一起，导致整个 项目体积过大，影响小程序 首次启动的下载时间



1.3 - 分包后的项目构成

分包后，小程序项目由 一个主包 + 多个分包 组成

- 主包 - 一般包含项目的 启动页面 或 Tabbar 页面，以及所有分包都需要用到的一些 公共资源
- 分包 - 只包含当前分包有关的页面和私有资源



1.4 - 分包的加载规则

- 小程序启动时，默认会 下载主包 并 启动主包内页面
  - tabBar 页面需要放到主包中
- 当用户进入分包内某个页面时， 客户端会把对应分包下载下来，下载后再进行展示
  - 非 tabBar 页面可以按照功能的不同，划分为不同的分包之后，进行按需下载

1.5 - 分包的体积限制

目前，小程序分包的大小有限制

- 整个小程序所有分包大小不超过 20M （主包 + 所有分包）
- 单个分包、主包大小不能超过 2M

2 - 使用分包

2.1 - 配置方法

假设支持分包的小程序目录结构如下

```
├── app.js
├── app.json
├── app.wxss
├── packageA         // 第一个分包
|   └── pages        // 第一个分包的所有页面
|       ├── cat
|       └── dog
├── packageB         // 第二个分包
|   └── pages        // 第二个分包的所有页面
|       ├── apple
|       └── banana
├── pages            // 主包的所有页面
|   ├── 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": [          // 声明项目分包结构
    {
      "root": "packageA",   // 分包根目录
      "pages": [            // 当前分包下，所有页面的相对存放路径
        "pages/cat/cat",
        "pages/dog/dog"
      ]
    }, {
      "root": "packageB",
      "name": "pack2",      // 分包别名
      "pages": [
        "pages/apple/apple",
        "pages/banana/banana"
      ]
    }
  ]
}
```

2.2 - 打包原则

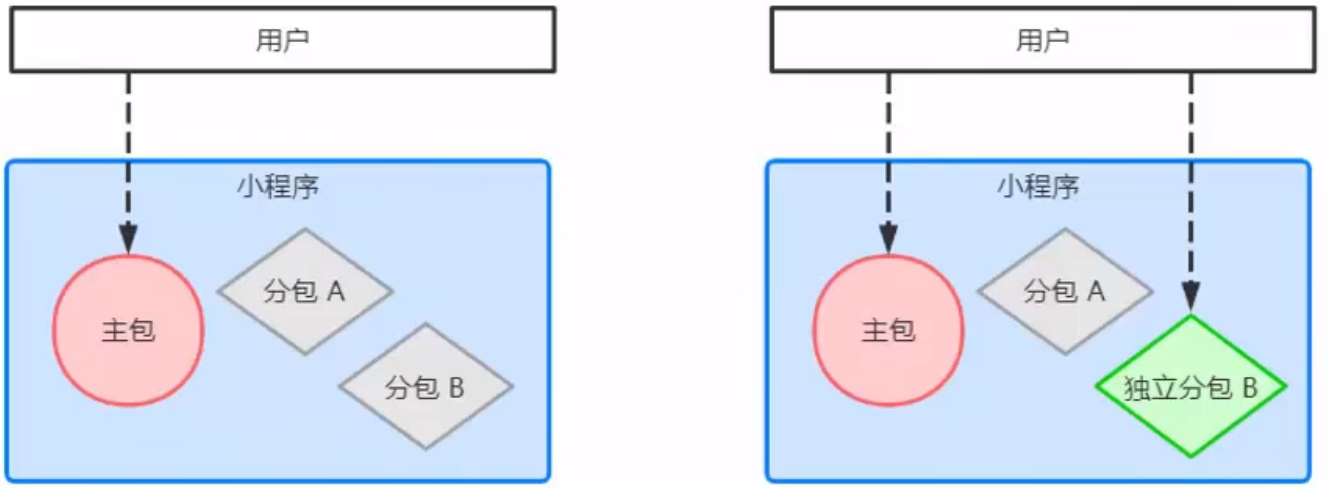
- 声明 `subpackages` 后，将按 `subpackages` 配置路径进行打包，`subpackages` 配置路径外的目录将被打包到主包中
- 主包也可以有自己的 pages，即最外层的 pages 字段。
- `subpackage` 的根目录不能是另外一个 `subpackage` 内的子目录
- `tabBar` 页面必须在主包内

2.3 - 引用原则

- 主包 无法引用 分包内的私有资源
- 分包之间 不能相互引用 私有资源
- 分包 可以 引用 主包内的公共资源

3 - 独立分包

独立分包 是小程序中一种特殊类型的分包， 可以独立于主包和其他分包运行



> **从独立分包中页面进入小程序时，不需要下载主包**
>
> **独立分包不依赖主包即可运行，可以很大程度上提升分包页面的启动速度**
>
> **一个小程序中可以有多个独立分包**

3.1 - 独立分包和普通分包的区别

最主要的区别： 是否依赖于主包才能运行

- 普通分包必须依赖于主包才能运行
- 独立分包可以在不下载主包的情况下，独立运行

3.2 - 独立分包的应用场景

可按需将 具有某些功能独立性的页面 配置到 独立分包 中

- 当小程序从普通的分包页面启动时，需首先下载主包
- 而独立分包 不依赖主包 即可运行，可以 很大程度上提升分包页面的启动速度

3.3 - 独立分包的配置方法

假设小程序目录结构如下

```
├── app.js
├── app.json
├── app.wxss
├── moduleA             // 普通分包
|   └── pages
|       ├── rabbit
|       └── squirrel
├── moduleB             // 独立分包
|   └── pages
|       ├── pear
|       └── pineapple
├── pages               // 主包所有页面
|   ├── index
|   └── logs
└── utils
```

开发者通过在 `app.json` 的 `subpackages` 字段中对应的分包配置项中定义 `independent` 字段声明对应分包为独立分包

```
{
  "pages": [
    "pages/index",
    "pages/logs"
  ],
  "subpackages": [
    {
      "root": "moduleA",
      "pages": [
        "pages/rabbit",
        "pages/squirrel"
      ]
    }, {
      "root": "moduleB",
      "pages": [
        "pages/pear",
        "pages/pineapple"
      ],
      "independent": true
    }
  ]
}
```

3.4 - 独立分包的引用原则

独立分包和普通分包及主包之间，是 相互隔绝 的， 不能相互引用彼此的资源

- 主包 无法引用 独立分包内的私有资源
- 独立分包之间， 不能相互引用 私有资源
- 独立分包和普通分包之间， 不能相互引用 私有资源
- **独立分包中也不能引用主包内的公共资源**

4 - 分包预下载

分包预下载 指在进入小程序某个页面时， 由框架自动预下载可能需要的分包 ， 提升进入后续分包页面时的启动速度

4.1 - 配置分包的预下载

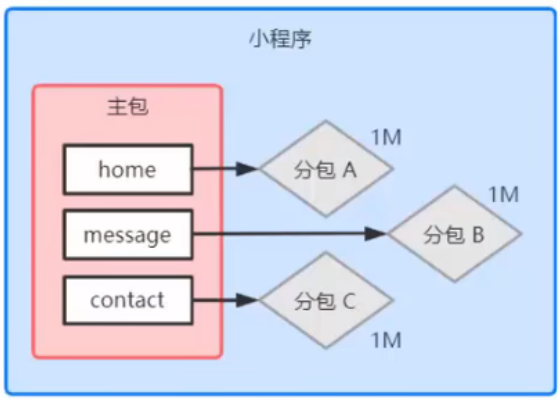
预下载分包行为在进入某个页面时触发，通过在 `app.json` 增加 `preloadRule` 配置来控制

```
{
  "pages": ["pages/index"],
  "subpackages": [
    {
      "root": "important",
      "pages": ["index"],
    },
    {
      "root": "sub1",
      "pages": ["index"],
    },
    {
      "name": "hello",
      "root": "path/to",
      "pages": ["index"]
    },
    {
      "root": "sub3",
      "pages": ["index"]
    },
    {
      "root": "indep",
      "pages": ["index"],
      "independent": true
    }
  ],
  "preloadRule": {              // 分包预下载规则
    "pages/index": {            // 触发分包预下载的页面路径
      "network": "all",         // 在指定网络模式下进行预下载，默认为 wifi
      "packages": ["important"] // 设置预下载的分包，可通过 root 或 name 指定
    },
    "sub1/index": {
      "packages": ["hello", "sub3"]
    },
    "sub3/index": {
```

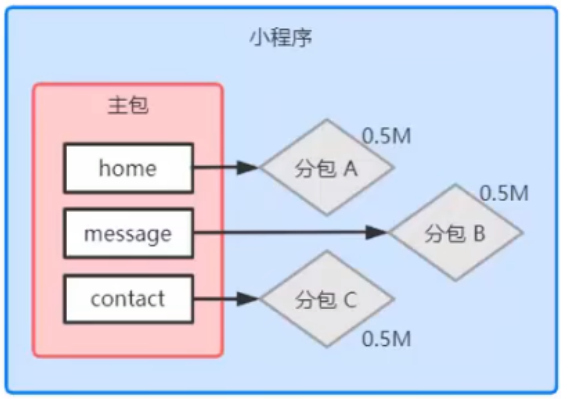
```
      "packages": ["path/to"]
    },
    "indep/index": {
      "packages": ["__APP__"]
    }
  }
}
```

4.2 - 分包预下载的限制

同一个分包中的页面享有 共同的预下载大小限额 2M ， 限额会在工具中打包时校验



不允许，分包 A+B+C 体积大于 2M



允许，分包 A+B+C 体积小于 2M