微信小程序基础Day05

今日目标:

- 能够知道如何安装和配置 `vant-weapp` 组件库
- 能够知道如何使用 `MobX` 实现全局数据共享
- 能够知道如何对小程序的 `API` 进行 `Promise` 化
- 能够知道如何实现自定义 `tabBar` 的效果

讲解目录:

- 1.使用 `npm` 包
- 2.全局数据共享
- 3.分包
- 4.案例 - 自定义 `tabBar`
- 5.总结

1. 使用 `npm` 包

1.1. 小程序对 `npm` 的支持与限制

目前，小程序中已经支持使用 `npm` 安装第三方包，从而来提高小程序的开发效率。

但是，在小程序中使用`npm` 包有如下 3 个限制：

① 不支持依赖于 `Node.js` 内置库的包

② 不支持依赖于浏览器内置对象的包

③ 不支持依赖于 `C++` 插件的包

总结：虽然 `npm` 上的包有千千万，但是能供小程序使用的包却“为数不多”。

1.2. `Vant Weapp`

> **1. 什么是 `Vant Weapp`**

`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。它所使用的是`MIT` 开源许可协议，对商业使用比较友好。

官方文档地址 https://youzan.github.io/vant-weapp

扫描下方的小程序二维码，体验组件库示例：

> **2. 安装 `Vant` 组件库**

在小程序项目中，安装 `Vant` 组件库主要分为如下几 步：

① 首先要初始化 `package.json` 文件

在项目根目录执行 `npm init -y`

`-y` 意思是快速下载，不必询问，都是 `yes`

② 通过 `npm` 安装（建议指定版本为@1.3.3），下载到了项目根目录下的 `node-modules`

```
npm i @vant/weapp@1.3.3 -S --production
```

③ 构建 `npm` 包，使用 `npm` 模块

④ 修改 `app.json`

将 `app.json` 中的 `"style": "v2"` 去除，小程序的新版基础组件强行加上了许多样式，难以去除，不关闭将造成部分组件样式混乱。

详细的操作步骤，大家可以参考 `Vant` 官方提供的快速上手教程：

- https://youzan.github.io/vant-weapp/#/quickstart#an-zhuang

> **3. 使用 Vant 组件**

安装完 Vant 组件库之后，可以在 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

示例代码如下：

```
// app.json
"usingComponents": {
    "van-button": "@vant/weapp/bitton/index"
}

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

> **4. 定制全局主题样式**

Vant Weapp 使用 CSS 变量来实现定制主题。 关于 CSS 变量的基本用法，请参考 MDN 文档：

https://developer.mozilla.org/zh-CN/docs/Web/CSS/Using_CSS_custom_properties

> **5. 定制全局主题样式**

在 app.wxss 中，写入 CSS 变量，即可对全局生效：

```
/* app.wxss */
page: {
    /* 定制警告按钮的背景颜色 和边框颜色*/
    --button-danger-background-color: #c00000;
    --button-danger-border-color: #d60000;
}
```

所有可用的颜色变量，请参考 Vant 官方提供的配置文件：

https://github.com/youzan/vant-weapp/blob/dev/packages/common/style/var.less

1.3. 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，例如，网络请求的 API 需要按照如下的方式调用：

```
wx.request({
    method: '',
    url: '',
    data: {},
    success: () => {}, // 成功的回调
    complate: () => {}, // 无论成功与否都会执行的回调
    fail: () => {}  // 失败的回调
})

```

这种代码的缺点是显而易见的，容易造成回调地狱的问题，代码的可读性、维护性差！而我们就想将这种类型的代码使用 API Promise化进行改造

> 2. 什么是 `API` `Promise` 化

`API` `Promise`化，指的是通过额外的配置，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

在小程序中，实现 `API` `Promise` 化主要依赖于 `miniprogram-api-promise`这个第三方的 `npm` 包。

它的安装和使用步骤如下:

- 安装构建

```
npm i --save miniprogram-api-promise@1.0.4

- 下载完成，我们不能直接使用，而是需要再次重新构建npm包
- 建议在构建前先删除原有的miniprogram_npm
- 然后再点击工具，构建npm
```

- 导入并执行代码

```
// 在小程序入口文件中（app.js),只需要调用一次 promisifyAll()方法
// 即可实现异步API 的Promise化

// 按需导入一个方法
import { promisifyAll } from 'miniprogram-api-promise'

// 声明一个常量，为一个空对象，
// 并在wx顶级对象下添加一个属性p也指向该空对象，使所有成员都可以使用该对象
const wxp = wx.p = {}
// promisify all wx's api
// 参数1： wx顶级对象
// 参数2： wxp指向一个空对象
promisifyAll(wx, wxp)
```

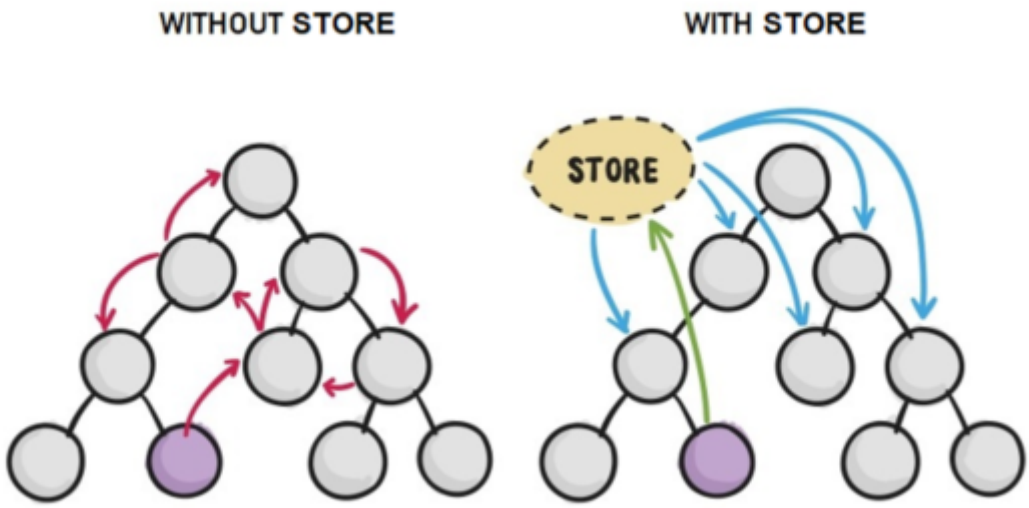
解释上述代码:

- `promisifyAll`:做的事就是将`wx` 拥有的属性方法都copy并改造了一份给了`wxp`这个对象
- 然而,`wxp`只是当前`js`文件的一个常量，只能在当前文件使用
- 因此：我们在`wx`上挂载一个属性`p` 让他和`wxp`指向同一个空对象
- 在其他页面或者组件就可以通过全局对象`wx`点出`p` 来访问到`wxp`
- 此时`wx.p`发起异步的请求时，得到的是一个`promise`对象
- 那么我们就可以使用 `async/await`简化Promise语法

2. 全局数据共享

> 1. 什么是全局数据共享

全局数据共享（又叫做：状态管理）是为了**解决组件之间数据共享**的问题。



开发中常用的全局数据共享方案有：`Vuex`、`Redux`、`MobX`等。

> **2. 小程序中的全局数据共享方案**

在小程序中，可使用`mobx-miniprogram`配合`mobx-miniprogram-bindings`实现全局数据共享。其中：

- `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

2.1. 安装`MobX`

在项目中运行如下的命令，安装`MobX`相关的包：

```
npm install --save mobx-miniprogram@4.13.2 mobx-miniprogram-bindings@1.2.1
```

注意：`MobX`相关的包安装完毕之后，记得删除`miniprogram_npm`目录后，**重新构建`npm`**。

2.2. 创建`MobX`的`Store`实例

在项目根目录下新建store文件夹，并且新建`store.js`文件

```
// 按需导入第三方包的方法observable, action
import { observable, action } from 'mobx-miniprogram'

// 创建 Store实例对象,并将其导出
export const store = observable({
    // 数据字段
    numA: 1,
    numB: 2,
    // 计算属性
    // 在计算属性的方法前，必须加 get修饰符，代表sum的值是只读的，无法进行修改
    // 计算属性sum  依赖于numA和numB的值，因此sum函数的返回值就是最终的值
    get sum() {
        return this.numA + this.numB
    },
    // 定义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. 使用将 Store 中的成员

2.3.1.在页面中使用

- 页面的`js`文件中

```
// 1.首先  导入第三方包，将数据绑定到页面
import { createStoreBindings } from 'mobx-miniprogram-bindings'
// 2.其次  在页面的js文件的头部区域导入容器的数据
import { store } from '../../store/store'

// 3. 绑定操作：将仓库的东西绑定到当前的页面中，在页面的js文件的Page方法中
Page({
    // 上面周期函数--监听页面的加载
    onLoad: function() {
        // 调用createStoreBindings方法
        // 参数1：  绑定给谁：当前页面this
        // 参数2：  对象{ store(容器)，  fields(数据)，  actions(修改方法)
        this.storeBindings = createStoreBindings(this, {
            // 映射容器的实例
            store,
            // 映射容器的数据字段
            fields: ['numA', 'numB', 'sum'],
            // 映射容器修改的方法
            actions: ['updateNum1']
        })
    },
    // 生命周期函数--监听页面的卸载
    onUnload: function () {
        // 使用this.storeBindings，得到调用createStoreBindings方法的返回值
        //调用destroyStoreBindings  方法，进行清理销毁的工作
        this.storeBindings.destroyStoreBindings()
    }
})
```

- 页面的`wxml`文件中

```
<!-- 使用仓库中的数据 -->
<view>{{numA}} + {{numB}} = {{sum}}</view>
<van-button type="primary" bindtap="btnHnadler1">numA+1</van-button>
```

监听函数btnHandler1的代码

```
// 页面的js文件中的 tap事件处理函数
btnHnadler1 (e) {
    console.log(e)
    // 使用仓库中的方法，并传递数据
    this.updateNum1(100)
}
```

2.3.2. 在组件中使用

- 准备工作
    - 新建组件文件夹以及组件文件`numbers`
    2. 全局注册这个组件
    3. 在`message`页面中使用子组件
- 组件的`js`文件

```
1. 按需导入容器成员
2. 在组件的behaviors节点 实现自动绑定
3. 在storeBindings节点指定要绑定的store和要绑定的数据以及方法

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

Component({
    // 通过storeBindingsBehavior 来实现自动绑定
    behaviors: [storeBindingsBehavior],
    storeBindings: {
        store, // 指定要绑定的store
        fields: {   // 指定要绑定的数据字段或计算属性
            numA: () => store.numA, // 绑定字段的方式1:
            numB: store => store.numB,  // 绑定字段的方式2
            sum: 'sum'  // 绑定字段的方式3
        },
        actions: {  // 指定要绑定的方法
            updateNum2: 'updateNum2'
        }
    }
})

注意： fields中前面是在组件中的名称，可自定义，后面是容器中的名称，必须和仓库一致
```

- 组件的`wxml`文件

```
<!-- 组件的 .wxml结构 -->
<view>{{numA}} + {{numB}} = {{sum}}</view>

<van-button type="primary" bindtap="btnHnadler2">numB+1</van-button>
```

事件处理函数`btnHandler2`

```
Compoonent({
    methods: {
        btnHnadler2(e) {
           // 直接使用this调用仓库中的方法
           this.updateNum2(20)
        }
    }
})
```

结论: 无论是在页面中使用仓库中的东西, 还是在组件中使用仓库中的东西, 在绑定成功之后, 就可以像使用自身数据和调用自身方法一样的对仓库的数据和方法进行操作

3. 分包

3.1. 分包相关概念

> 1. 什么是分包

分包指的是把一个**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。

> 2. 分包的好处

对小程序进行分包的好处主要有以下两点：

- 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的解耦协作

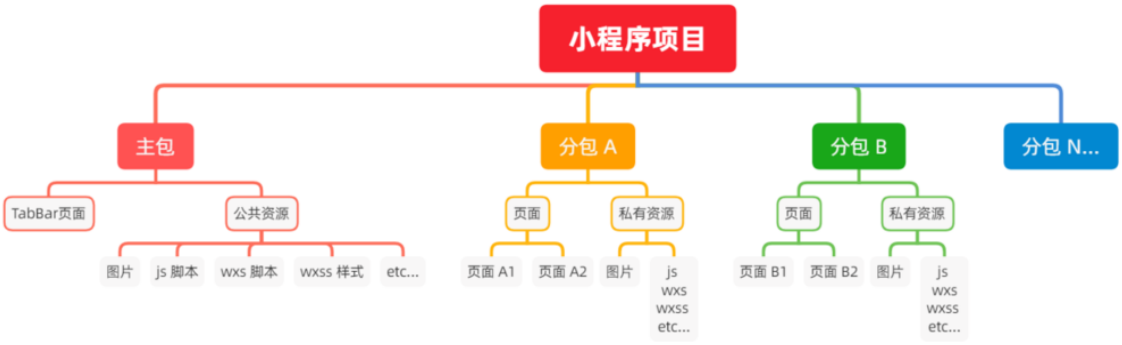
> 3. 分包前 项目的构成

分包前，小程序项目中所有的页面和资源都被打包到了一起，导致整个项目体积过大，影响小程序首次启动的下载时间。

> 4. 分包后 项目的构成

分包后，小程序项目由 **1 个主包 + 多个分包**组成：

- **主包**：一般只包含项目的**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**：只包含和当前分包有关的页面和**私有资源**



> 5. 分包的加载规则

① 在小程序启动时，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

② 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**，下载完成后再进行展示

- 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

6. 分包的体积限制

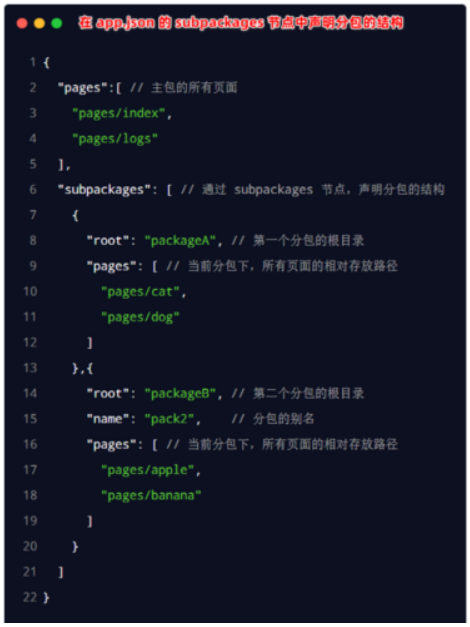
目前，小程序分包的大小有以下两个限制：

- 整个小程序所有分包大小不超过 `16M`（主包 + 所有分包）
- 单个分包/主包大小不能超过 `2M`

3.2. 使用分包

1. 配置方法

在 `app.json` 配置文件中, 新增 `subpackages` 的节点, 将希望放到分包的页面,写入 `subpackages` 数组的元素中.



2. 打包原则

① 小程序会按 `subpackages` 的配置进行分包， `subpackages` 之外的目录将被打包到主包中

② 主包也可以有自己的 `pages`（即最外层的 pages 字段）

③ `tabBar` 页面必须在主包内

④ 分包之间不能互相嵌套

3. 引用原则

① 主包无法引用分包内的私有资源

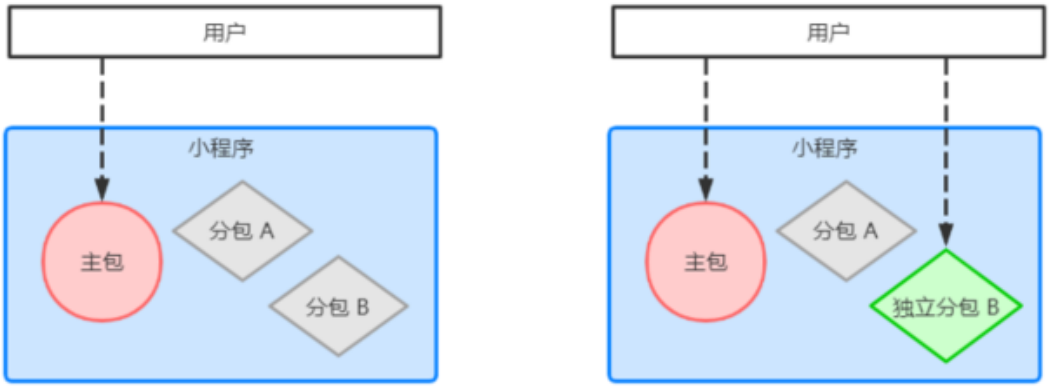
② 分包之间不能相互引用私有资源

③ 分包可以引用主包内的公共资源

3.3. 独立分包

1. 什么是独立分包

独立分包本质上也是分包，只不过它比较特殊，可以独立于主包和其他分包而**单独运行。**



2. 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. 独立分包的应用场景

开发者可以按需，将某些**具有一定功能独立性的页面配置到独立分包中**。原因如下：

- 当小程序从普通的分包页面启动时，需要首先下载主包
- 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**

注意：一个小程序中可以有多个独立分包。

4. 独立分包的配置方法

和普通分包对比，独立分包就是在`subpackages`数组的某个元素中，配置`independent`为true即可

5. 引用原则

独立分包和普通分包以及主包之间，是**相互隔绝**的，不能相互引用彼此的资源！例如：

① 主包无法引用独立分包内的私有资源

② 独立分包之间，不能相互引用私有资源

③ 独立分包和普通分包之间，不能相互引用私有资源

④ **特别注意：**独立分包中不能引用主包内的公共资源

3.4. 分包预下载

> 1. 什么是分包预下载

分包预下载指的是：在进入小程序的某个页面时，*由框架自动预下载可能需要的分包*，从而**提升**进入后续分包页面时的启动速度。

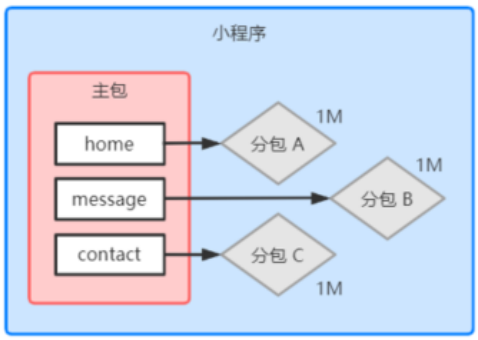
> 2. 配置分包的预下载

预下载分包的行为，会在进入指定的页面时触发。在 `app.json` 中，使用 `preloadRule` 节点定义分包的预下载规则，示例代码如下：

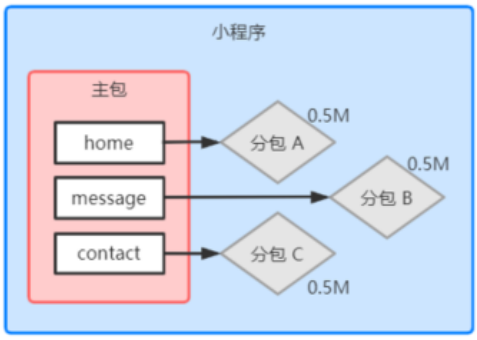
```
{
    分包预下载的规则
    "preloadRule": {
         触发分包预下载的页面路径
        "page/concact/concact": {
             network 表示在指定的网络模式下进行预下载
             可选值有： all(不限网络)和wifi(仅 wifi 模式下进行预下载)
             默认值为： wifi
            "network": "all",
             packages 表示进入页面后， 预下载哪些分包
             可以通过root 或name 指定预下载哪些分包
            "packages": ["pkgA"]
        }
    }
}
```

> 3. 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`，例如：



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

4. 案例 - 自定义 `tabBar`

4.1. 案例效果

其中,底部的 `tabBar` 是通过 `vant` 中的组件而实现的

在此案例中，用到的主要知识点如下：

- 自定义组件
- `Vant` 组件库
- `MobX` 数据共享
- 组件样式隔离
- 组件数据监听器
- 组件的 `behaviors`
- `Vant` 样式覆盖

4.2. 自定义 `tabBar` 的一般实现步骤

自定义 `tabBar` 分为 3 大步骤，分别是：

① 配置信息

- 在 `app.json` 中的 `tabBar` 项指定 `custom` 字段，同时其余 `tabBar` 相关配置也补充完整。
- 所有 `tab` 页的 `json` 里需声明 `usingComponents` 项，也可以在 `app.json` 全局开启。

> 在自定义 `tabBar` 模式下

- 为了保证低版本兼容以及区分哪些页面是 `tab` 页，`tabBar` 的相关配置项需完整声明，但这些字段不会作用于自定义 `tabBar` 的渲染。
- 此时需要开发者提供一个自定义组件来渲染 `tabBar`，所有 `tabBar` 的样式都由该自定义组件渲染。

```
{
  "tabBar": {
    "custom": true,
```

```
    "color": "#000000",
    "selectedColor": "#000000",
    "backgroundColor": "#000000",
    "list": [{
      "pagePath": "page/component/index",
      "text": "组件"
    }, {
      "pagePath": "page/API/index",
      "text": "接口"
    }]
  },
  "usingComponents": {}
}

```

② 添加 `tabBar` 代码文件

在项目根目录下创建custom-tab-bar文件夹, 并创建index组件

```
custom-tab-bar/index.js
custom-tab-bar/index.json
custom-tab-bar/index.wxml
custom-tab-bar/index.wxss
```

注意: 文件夹的名称和组件的名称都不能改

③ 编写 `tabBar` 代码

> 用自定义组件的方式编写即可，该自定义组件完全接管 `tabBar` 的渲染。另外，自定义组件新增 `getTabBar` 接口，可获取当前页面下的自定义 `tabBar` 组件实例。

详细步骤，可以参考小程序官方给出的文档:

https://developers.weixin.qq.com/miniprogram/dev/framework/ability/custom-tabbar.html

4.3. 基于`vant-weapp`组件库来渲染底部`tabbar`

基于`vant-weapp`组件库来渲染底部`tabbar`

- 在`app.json/index.json`文件中，注册组件

```
"usingComponents": {
  "van-tabbar": "@vant/weapp/tabbar/index",
  "van-tabbar-item": "@vant/weapp/tabbar-item/index"
}
```

- 在组件的`wxml`文件中使用组件

```
<van-tabbar active="{{ active }}" bind:change="onChange">
  <van-tabbar-item info="3">
    <image
      slot="icon"
      src="/images/tabs/home.png"
      mode="aspectFit"
      style="width: 25px; height: 25px;"
    />

```

```
    <image
      slot="icon-active"
      src="/images/tabs/home-active.png"
      mode="aspectFit"
      style="width: 25px; height: 25px;"
    />
    自定义
  </van-tabbar-item>
</van-tabbar>
```

- 组件`js`文件中，书写逻辑

```
Page({
  data: {
    active: 0,
   将app.json的中的tabBar的list节点的内容粘贴过来
   循环遍历生成tabbar，也可以手动绘制tabbar,直接使用sum

  },
  onChange(event) {
    // event.detail 的值为当前选中项的索引
    this.setData({ active: event.detail });
  },
});
```

- 重置`vant`组件库的样式

```

.van-tabbar-item {
    重置css变量的值为0
    --tabbar-item-bottom(0)
}

// 样式不生效，需要在组件的js文件中，添加解除样式隔离的配置

Component({
    options: {
        // 解除样式隔离
        styleIsolation: 'shared'
    }
})
```

- 渲染数组徽标

```
可以将三个tabbar手动渲染出来，不使用循环遍历生成
```

- 在`tabbar`组件中使用`store`中的数据

```
1. 导入mobx 第三方包，导入store
2. 使用behavior挂载按需加载的第三方包的方法`storeBindingsBehavior`
3. 进行数据的绑定
// 添加数据绑定节点
storeBindings: {
    store,
        fields: {
```

```
            sum: 'sum'
        },
        actions: {

        }
},
4.监听sum的变化，当变化的时候为info赋值
// 添加侦听器节点
observers: {
    'sum': function(val) {
        this.setData({
            sum: val
        })
    }
}

5.在组件的js文件中处理tabbar的change事件
onChange(e) {
     // e.detail 的值为当前选中项的索引
    this.setData({ active: e.detail})
    wx.switchTab({
        // 这个url必须以/根路径开头
        url: this.data.list[e.detail].pagePath
    })
}
6. 将组件内部的数据声明移除掉，声明在store中
- 由于van-tabbar组件内部修改了active的值而我们通过change事件也去修改了active的值，使tabbar的展示出现了bug
- 解决办法： 将数据放在容器中，通过映射去使用容器的数据和修改数据的方法

// store.js中
export const store = observable({
    numA: 1,
    numB: 2,
    activeTabBarIndex: 0,
    get sum() {
        return this.numA + this.numB
    }
    // 定义actives方法， 用来修改 store 中的数据
    ...
    updateActiveTabBarIndex: active(function(index) {
    this.activeTabBarIndex = index
})
})
// 组件的index.js中
Component({
    // 将容器中的方法挂载在behaviors节点？？
    behaviors: ['storeBindingsBehavior'],
    storeBindings: {
        // 指定使用的容器
       store,
        // 容器中的数据
        fields: {
            sum: 'sum',
            active: 'activeTabBarIndex'
        },
        // 容器中修改数据的方法
        actions: {
```

```
            // 自定义方法：  '容器中的方法'
            updateActive: 'updateActiveTabBarIndex'
        }
    },
     methods: {
        onChange(e) {
             // e.detail 的值为当前选中项的索引
            // this.setData({ active: e.detail})
           // 使用容器中的方法，更新active的值
            this.updateActive(e.detail)
            // 实现tabbar的跳转
            wx.switchTab({
               // 这个url必须以/根路径开头
               url: this.data.list[e.detail].pagePath
           })
        }
     }
})
```

4.4.最终案例代码

- `store.js`

```
/**
* mobx 第三方包，解决组件之间数据共享
* 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
**/
// 1.按需导入包的方法
import { observable, action } from 'mobx-miniprogram'
// 2.创建store 实例对象，并导出该实例
export const store = observable({
    // 数据字段
    numA: 1,
    numB: 2,
    /*** 计算属性：
    * 在计算属性的方法名前 必须加 get修饰符，代表是计算属性，且当前的 数据是只读的，无法进行修改
    * 计算属性sum 依赖于numA 和numB 的值， 因此sum 函数的返回值 就是最终sum的值

    */
    get sum() {
        return this.numA + this.numB
    },
    // actions ：用来修改store中的数据
    updateNumA: action(function(step) {
        this.numA += step
    }),
    updateNumB: action(function(step) {
        this.numB += step
    })
})
```

- `custom-tab-bar/index.js`

```
/** 在该组件中使用仓库中的数据sum  */
import { storeBindingsBehavior } from 'mobx-miniprogram- bindings'
import { store } from '../store/store'
Component({
    /*** 组件的配置节点 */
    options: {
        /** 解除 * 组件样式隔离 */
        styleIsolation: 'shared',
        /*** 匹配纯数据字段的规则，以_开头 */
        pureDataPattern: /^_/
    },
    // 将导入的包 挂载在behaviors中
    behaviors: [storeBindingsBehavior],
    // 添加storeBindings节点
    storeBindings: {
        // 指定仓库
        store, fields: {
            sum: 'sum',
            // 当前激活的tabbar的索引
            active: 'activeIndex'
        },
        actions: {
            // 映射仓库修改数据的方法
            updateActive: 'updateActiveIndex'
        } },
    /*** 组件的属性列表 */
    properties: { },
    /*** 组件的初始数据 */
    data: {
        // 当前激活的tabbar的索引
        // active: 0,
        // 纯数字字段： 路径列表
        _list: [
            {"pagePath": "/pages/home/home"},
            {"pagePath": "/pages/message/message"},
            {"pagePath": "/pages/contact/contact"}
        ] },
    /*** 组件的方法列表 */
    methods: {
        onChange(e) {
            // 修改仓库active的数据
            this.updateActive(e.detail)
            // tabbar的切换导航
            wx.switchTab({
                url: this.data._list[e.detail].pagePath,
            })
        }}
})
```

- `custom-tab-bar/index.wxml`

```
<van-tabbar active="{{active}}" bind:change="onChange" active-
color="#13A7A0">
    <van-tabbar-item>
        <image slot="icon" src="/images/tabs/home.png" mode="aspectFit"
style="width: 25px; height: 25px;" />
```

```
        <image slot="icon-active" src="/images/tabs/home-active.png" mode="aspectFit" style="width: 25px; height: 25px;" />
        首页
    </van-tabbar-item>
    <!-- 将消息总数绑定为sum -->
    <van-tabbar-item info="{{sum}}">
        <image slot="icon" src="/images/tabs/message.png" mode="aspectFit" style="width: 25px; height: 25px;" />
        <image slot="icon-active" src="/images/tabs/message-active.png" mode="aspectFit" style="width: 25px; height: 25px;" />
        消息
    </van-tabbar-item>
    <van-tabbar-item>
        <image slot="icon" src="/images/tabs/contact.png" mode="aspectFit" style="width: 25px; height: 25px;" />
        <image slot="icon-active" src="/images/tabs/contact-active.png" mode="aspectFit" style="width: 25px; height: 25px;" />
        联系我们
    </van-tabbar-item>
</van-tabbar>
```

- `app.json`

```
{
    "pages":[
        "pages/home/home",
        "pages/message/message",
        "pages/contact/contact",
        "pages/info/info"
    ],
    "subpackages": [
        {
            "root": "packageA",
            "name": "p1",
            "pages": [
                "pages/cat/cat",
                "pages/dog/dog"
            ]
        },{
            "root": "packageB",
            "name": "p2",
            "pages": [
                "pages/apple/apple",
                "pages/banana/banana"
            ],
            "independent": true
        }
    ],
    "preloadRule": {
        "pages/contact/contact": {
            "network": "all",
            "packages": ["p1"]
        }
    },
    "window":{
        "backgroundTextStyle":"light",
        "navigationBarBackgroundColor": "#13A7A0",
```

```
        "navigationBarTitleText": "Weixin",
        "navigationBarTextStyle":"black"
    },
    "tabBar": {
        "custom": true,
        "list": [
            {
                "pagePath": "pages/home/home",
                "text": "首页",
                "iconPath": "/images/tabs/home.png",
                "selectedIconPath": "/images/tabs/home-active.png"
            },{
                "pagePath": "pages/message/message",
                "text": "消息", "iconPath":
                "/images/tabs/message.png",
                "selectedIconPath": "/images/tabs/message- active.png"
            },{
                "pagePath": "pages/contact/contact",
                "text": "联系我们",
                "iconPath": "/images/tabs/contact.png",
                "selectedIconPath": "/images/tabs/contact- active.png"
            }
        ]
    },
    "sitemapLocation": "sitemap.json",
    "usingComponents": {
        "van-button": "@vant/weapp/button/index",
        "my-numbers": "/components/numbers/numbers",
        "van-tabbar": "@vant/weapp/tabbar/index",
        "van-tabbar-item": "@vant/weapp/tabbar-item/index"
    }
}
```

5. 总结

① 能够知道如何安装和配置 `vant-weapp` 组件库

- 参考 `Vant` 的官方文档

② 能够知道如何使用 `MobX` 实现全局数据共享

- 安装包、创建 `Store`、参考官方文档进行使用

③ 能够知道如何对小程序的 `API` 进行 `Promise` 化

- 安装包、在 `app.js` 中进行配置

④能够知道如何实现自定义 `tabBar` 的效果

- `Vant` 组件库 + 自定义组件 + 全局数据共享