微信小程序基础Day03

今日目标:

- 能够知道如何实现页面之间的导航跳转
- 能够知道如何实现下拉刷新效果
- 能够知道如何实现上拉加载更多效果
- 能够知道小程序中常用的生命周期函数

讲解目录:

- 1.页面导航
- 2.页面事件
- 3.生命周期
- 4.`WXS` 脚本
- 5.案例 - 本地生活（列表页面
- 6.总结

1. 页面导航

> **1. 什么是页面导航**

页面导航指的是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种：

a 链接

`location.href`

a链接导航称之为声明式导航，`location.href`赋值的方式称之为编程式导航

> **2. 小程序中实现页面导航的两种方式**

① 声明式导航

- 在页面上声明一个 导航组件
- 通过点击 组件实现页面跳转

② 编程式导航

- 调用小程序的导航 `API`，实现页面的跳转

1.1. 声明式导航

> **1. 导航到 `tabBar` 页面**

`tabBar` 页面指的是被配置为 `tabBar` 的页面。

在使用 组件跳转到指定的 `tabBar` 页面时，需要指定 `url`属性和 `open-type` 属性，其中：

- `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`

示例代码如下：

```
<!-- 声明式导航 -->
<!--
  导航到 tabBar页面的方法：
  url必须以“/”根路径开头
  并且必须配置open-type属性值为switchTab
  -->
<navigator url="/pages/message/message" open-type="switchTab">导航到消息页面
</navigator>
```

2. 导航到非 tabBar 页面

非 tabBar 页面指的是没有被配置为 tabBar 的页面。

在使用 组件跳转到普通的非 tabBar 页面时，则需要指定 url 属性和 open-type 属性，其中：

- 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 navigate
- 为了方便, 非 tabBar 页码的跳转时 open-type 也可以省略

示例代码如下：

```
<!--
    导航到非tabBar页面
    也就是没有被配置为tabBar的页面
    url必须以“/”根路径开头
    如果配置了open-type属性，值为navigate,也可省略不写
 -->
 <navigator url="/pages/info/info" open-type="navigate">跳转到info页面
</navigator>
 <navigator url="/pages/info/info" >跳转到info页面</navigator>
```

3. 后退导航

如果要后退到上一页面或多级页面，则需要指定 open-type 属性和 delta 属性，其中：

- open-type 的值必须是 navigateBack，表示要进行后退导航
- delta 的值必须是数字，表示要后退的层级

示例代码如下：

```
<!--
    后退导航
    open-type属性值为：navigateBack
    结合delta属性：表示后退的层级，默认为1，为1时该属性可省略不写
 -->
 <navigator open-type="navigateBack" delta="1">后退</navigator>
 <navigator open-type="navigateBack">后退</navigator>
```

注意：

- 为了简便，如果只是后退到上一页面，则可以省略 delta 属性，因为其默认值就是 1
- tabBar 页面是不能实现后退的效果的. 因为, 当我们跳转到 tabBar 页面，会关闭其他所有非 tabBar 页面,所以当处于 tabBar 页面时, 无页面可退

1.2. 编程式导航

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` 参数对象的属性列表如下

属性	类型	是否必选	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）

示例代码如下：

```
<!--
  编程式导航
  跳转至tabBar页面
 -->
 <button bindtap="gotoMessage">跳转到messae页面</button>



  /*
  通过编程式导航
  跳转至tabBar页面
  message页面
  */
  gotoMessage () {
    wx.switchTab({
      // 代表要跳转的路径
      url: '/pages/message/message',
    })
  },
```

2. 导航到非 `tabBar` 页面

调用 `wx.navigateTo(Object object)` 方法，可以跳转到非 `tabBar` 的页面。

其中 `Object` 参数对象的属性列表, 如下：

属性	类型	是否必选	说明
`url`	`string`	是	需要跳转到的非 `tabBar` 页面的路径，路径后可以带参数
`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）

示例代码如下：

```
<!--
 编程式导航
 跳转至非 tabBar页面
-->
<button bindtap="gotoInfo">跳转到Info页面</button>

/*
  通过编程式导航
  跳转至非tabBar页面
  Info页面
  */
  gotoInfo () {
    wx.navigateTo({
      url: '/pages/info/info',
    })
  },
```

3. 后退导航

调用 `wx.navigateBack(Object object)` 方法，可以返回上一页面或多级页面。

其中 `Object` 参数对象可选的, 属性列表如下：

属性	类型	是否必选	说明	默认值
`delta`	`number`	否	返回的页面数，如果 `delta` 大于现有页面数，则返回到首页	1
`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）	

示例代码如下：

```
  <!--
  通过编程式导航实现后退导航
 -->
 <button bindtap="goBack">编程式导航实现后退</button>


/*
    通过编程式导航实现后退导航
```

```
  */
  goBack () {
    // 如果不传递参数就是返回上一页
    // 如果要传递参数则是传递 delta数字型， 代表返回的层级。
    wx.navigateBack()
  },
```

注意：

- `tabBar`页面是不能实现后退的效果的. 因为, 当我们跳转到`tabBar`页面，会关闭其他所有非`tabBar`页面,所以当处于`tabBar`页面时, 无页面可退

1.3. 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的路径。同时，路径的后面还可以携带参数：

- 参数与路径之间使用？分隔
- 参数键与参数值用 = 相连
- 不同参数用 & 分隔

代码示例如下：

```
<!--
    声明式导航传参
    参数与路径之间使用 ？ 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=zs&age=20">跳转至info页面</navigator>
```

> 2. 编程式导航传参

调用`wx.navigateTo(Object object)`方法跳转页面时，也可以携带参数，代码示例如下：

```
<!--
  编程式导航传参
 -->
  <button bindtap="gotoInfo2">跳转到info页面</button>

  /*
    编程式导航传递参数
  */
  gotoInfo2 () {
    wx.navigateTo({
      url: '/pages/info/info?name=李&gender=男',
    })
  },
```

> 3. 在`onLoad`中接收导航参数

通过**声明式导航传参**或**编程式导航传参**所携带的参数，可以直接在`onLoad`事件中直接获取到，示例代码如下：

```
  /**
   * 页面的初始数据
```

```
   */
  data: {
    // 导航传递的参数
    query: {}
  },


/**
   * 生命周期函数--监听页面加载
   */
  onLoad: function (options) {
    // 通过声明式导航和编程式导航 都可以
    // 在onLoad声明周期函数中获取传递的参数
    console.log(options);
    // 将导航传递的参数转存在data中
    this.setData({
      query: options
    })
  },
```

2. 页面事件

2.1. 下拉刷新事件

> **1. 什么是下拉刷新**

下拉刷新是移动端的专有名词，指的是通过手指在屏幕上的下拉滑动操作，从而**重新加载页面数据**的行为。

> **2. 启用下拉刷新**

启用下拉刷新有两种方式：

① 全局开启下拉刷新

- 在 `app.json` 的 window 节点中，将 `enablePullDownRefresh` 设置为 true

② 局部开启下拉刷新

- 在页面的 `.json` 配置文件中，将 `enablePullDownRefresh` 设置为 true

在实际开发中，推荐使用第 2 种方式，为需要的页面单独开启下拉刷新的效果。

> **3. 配置下拉刷新窗口的样式**

在全局或页面的 `.json` 配置文件中，通过 `backgroundColor` 和 `backgroundTextStyle` 来配置下拉刷新窗口

的样式，其中：

- `backgroundColor` 用来配置下拉刷新窗口的背景颜色，仅支持16 进制的颜色值
- `backgroundTextStyle` 用来配置下拉刷新 `loading` 的样式，仅支持 `dark` 和 `light`

> **4. 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。

例如，在页面的 `wxml` 中有如下的 `UI` 结构，点击按钮可以让 `count` 值自增 +1：

```
<view>
  count的值为：  {{count}}
</view>

<button bindtap="addCount">+1</button>

```

```

Page({
      /**
   * 页面的初始数据
   */
  data: {
    count: 0
  },
  // 点击按钮  count自增加1事件
  addCount () {
    this.setData({
      count: this.data.count +1
    })
  }
})
```

在触发页面的下拉刷新事件的时候，如果要把 count 的值重置为 0，示例代码如下：

```
  /**
   * 页面相关事件处理函数--监听用户下拉动作
   */
  onPullDownRefresh: function () {
    // 触发了下拉刷新事件就会立即调用该方法
    console.log("触发了下拉刷新");
    // 触发了下拉刷洗就将data中的count重置为0
    this.setData({
      count: 0
    })
  },
```

5. 停止下拉刷新的效果

当处理完下拉刷新后，下拉刷新的 `loading` 效果会一直显示，不会主动消失，所以需要手动隐藏下拉刷新的

`loading`效果。此时，调用 `wx.stopPullDownRefresh()` 可以停止当前页面的下拉刷新。示例代码如下：

```
  /**
   * 页面相关事件处理函数--监听用户下拉动作
   */
  onPullDownRefresh: function () {
    wx.request({
        url: '',
        method: 'GET',
        success: (res) => {
            this.setData({
                XXX: res.data
```

```
                // 数值处理完毕，就可以关闭下拉刷新的事件了
                wx.stopPullDownRefresh()
            })
        }
    })
},
```

2.2. 上拉触底事件

> 1. 什么是上拉触底

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. 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。示例代码如下：

```
/**
 * 页面上拉触底事件的处理函数
 */
onReachBottom: function () {
  /*
  上拉触底事件不需要开启，直接监听就可以
  在全局配置的window节点中或者页面的配置文件中可设置触底距离：
  上拉触底的距离：默认50像素，单位省去,我们会在触发了上拉触底事件时获取下一页的数据
  "onReachBottomDistance": 50
  */
  console.log("触发了上拉触底事件");
  /*
    在上拉触底事件中，需要做节流处理
    防止频繁触发该事件导致频繁发起请求
  */
},
```

> 3. 配置上拉触底距离

上拉触底距离指的是触发上拉触底事件时，滚动条距离页面底部的距离。

可以在全局或页面的`.json`配置文件中，通过`onReachBottomDistance`节点来配置上拉触底的距离。

小程序默认的触底距离是`50px`，在实际开发中，可以根据自己的需求修改这个默认值。

2.3. 上拉触底案例

> 1. 案例效果展示

2. 案例的实现步骤

① 定义获取随机颜色的方法

② 在页面加载时获取初始数据

③ 渲染 `UI` 结构并美化页面效果

④ 在上拉触底时调用获取随机颜色的方法

⑤ 添加 `loading` 提示效果

⑥ 对上拉触底进行节流处理

3. 步骤1 - 定义获取随机颜色的方法

```
Page({
  data: {
    // 随机颜色数组
    colorList: []
  },
  // 获取随机颜色的方法
  getColors () {
    wx.request({
      url: 'https://www.escook.cn/api/color',
      method: 'GET',
      success: ({data: res}) => {
        console.log(res)
        this.setData({
          // 使用展开运算符获取到每一个数据，再将数据进行拼接
```

```
          colorList: [...this.data.colorList, ...res.data]
        })
      }
    })
  },
})
```

3. 步骤2 - 在页面加载时获取初始数据

```
Page({
      /**
   * 生命周期函数--监听页面加载
   */
  onLoad: function (options) {
    // 初始化页面
    this.getColors()
  }
})
```

3. 步骤3 - 渲染 `UI` 结构并美化页面效果

```
/* pages/contact/contact.wxss */
.num-item {
  border: 1rpx solid #efefef;
  border-radius: 8rpx;
  line-height: 200px;
  margin: 15rpx;
  text-align: center;
  text-shadow: 0rpx 0rpx 5rpx #fff;
  box-shadow: 1rpx 1rpx 5rpx #efefef;
}
```

3. 步骤4 - 上拉触底时获取随机颜色

```
  /**
   * 页面上拉触底事件的处理函数
   */
  onReachBottom: function () {
    // 在真实的项目中，我们请求数据一定是下一页数据，因此需要再data中存储一个当前页码的数据
    // 在触发上拉触底事件时，将代表页码的数据对象+1，然后再去调用获取数据的方法
    this.getColors()// 上拉触底的时候，再次调用获取随机颜色的方法
  },
```

3. 步骤5 - 添加 loading 提示效果

```
  // 获取随机颜色的方法
  getColors () {
    // 1.展示loading效果
    wx.showLoading({
      title: '数据加载中...',
    })
    // 2.发起网络请求
    wx.request({
      url: 'https://www.escook.cn/api/color',
```

```
      method: 'GET',
      success: ({data: res}) => {
        console.log(res)
        this.setData({
          // 使用展开运算符获取到每一个数据，再将数据进行拼接
          colorList: [...this.data.colorList, ...res.data]
        })
      },
      // 无论成功与否都会调用该方法
      complete: () => {
       // 3.隐藏loading效果
       wx.hideLoading()
      }
    })
  }
```

3. 步骤6 - 对上拉触底进行节流处理

① 在 `data` 中定义 `isLoading` 节流阀

- `false` 表示当前没有进行任何数据请求
- `true` 表示当前正在进行数据请求

```
  /**
  * 页面的初始数据
  */
 data: {
   // 随机颜色数组
   colorList: [],
   // 节流阀: false代表当前没有进行任何数据请求
   isLoading: false
 },
```

② 在 `getColors()` 方法中修改 `isloading` 节流阀的值

- 在刚调用 `getColors` 时将节流阀设置 `true`
- 在网络请求的 `complete` 回调函数中，将节流阀重置为 `false`

```
// 获取随机颜色的方法
  getColors () {
    // 1.展示loading效果
    wx.showLoading({
      title: '数据加载中...',
    })
    // 2.开启节流阀
    this.setData({
     isLoading: true
    })
    // 3.发起网络请求
    wx.request({
      url: 'https://www.escook.cn/api/color',
      method: 'GET',
      success: ({data: res}) => {
        // 修改data中的数据
        this.setData({
          // 使用展开运算符获取到每一个数据，再将数据进行拼接
          colorList: [...this.data.colorList, ...res.data]
```

```
        })
      },
      // 无论成功与否都会调用该方法
      complete: () => {
        // 4.隐藏loading效果
        wx.hideLoading()
        // 5.关闭节流阀
        this.setData({
          isLoading: false
       })
      }
    })
  },
```

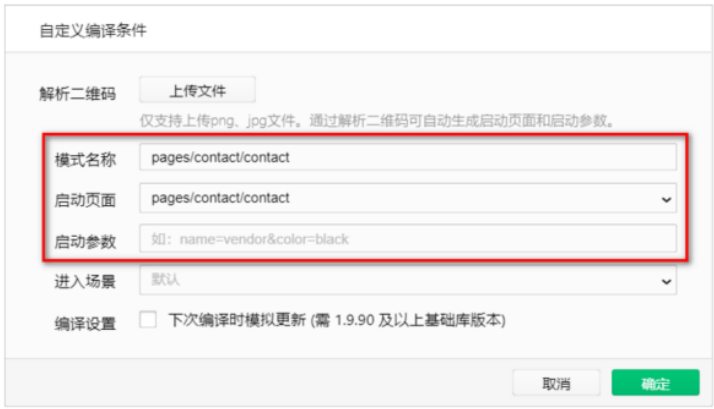
③ 在 `onReachBottom` 中判断节流阀的值，从而对数据请求进行节流控制

- 如果节流阀的值为 `true`，则阻止当前请求
- 如果节流阀的值为 `false`，则发起数据请求

```
/**
  * 页面上拉触底事件的处理函数
  */
 onReachBottom: function () {
  if (this.data.isLoading) return
  this.getColors()
 },
```

2.4. 自定义编译模式

由于页码每次编译都会回到首页，如果我们想让页面编译时，直接展示我们调试的页面可以这样做



3. 生命周期

3.1. 什么是生命周期

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

- 张三出生，表示这个人生命周期的开始
- 张三离世，表示这个人生命周期的结束
- 中间张三的一生，就是张三的生命周期

我们可以把每个小程序运行的过程，也概括为生命周期：

- 小程序的启动，表示生命周期的开始
- 小程序的关闭，表示生命周期的结束
- 中间小程序运行的过程，就是小程序的生命周期

3.2. 生命周期的分类

在小程序中，生命周期分为两类，分别是：

① 应用生命周期

- 特指小程序从启动 -> 运行 -> 销毁的过程

② 页面生命周期

- 特指小程序中，每个页面的加载 -> 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，如图所示：

3.3. 什么是生命周期函数

- 生命周期函数

  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

  例如，页面刚加载的时候，可以在`onLoad`生命周期函数中初始化页面的数据。

注意：生命周期强调的是时间段，生命周期函数强调的是时间点。

3.4. 生命周期函数的分类

小程序中的生命周期函数分为两类，分别是：

① 应用的生命周期函数

- 特指小程序从启动 -> 运行 -> 销毁期间依次调用的那些函数

② 页面的生命周期函数

- 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

3.5. 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，示例代码如下：

```
/**
 * 当小程序初始化完成时，会触发 onLaunch（全局只触发一次）
 */
onLaunch: function () {
  console.log("onLaunch")
},
```

```
  /**
   * 当小程序启动，或从后台进入前台显示，会触发 onShow
   * 前台：手机打开该小程序就处于前台运行
   * 后台：手机回到主页，且未关闭小程序，此时小程序就处于后台运行
   */
  onShow: function (options) {
    console.log("onShow")
  },

  /**
   * 当小程序从前台进入后台，会触发 onHide
   */
  onHide: function () {
    console.log("onHide")
  },
```

3.6. 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，示例代码如下：

```
/**
  * 生命周期函数--监听页面加载,一个页面只调用一次
  */
 onLoad: function (options) {

 },

 /**
  * 生命周期函数--监听页面初次渲染完成，一个页面只调用一次
  */
 onReady: function () {

 },

 /**
  * 生命周期函数--监听页面显示
  */
 onShow: function () {

 },

 /**
  * 生命周期函数--监听页面隐藏
  */
 onHide: function () {

 },

 /**
  * 生命周期函数--监听页面卸载，一个页面只调用一次
  */
 onUnload: function () {

 },
```

4. WXS 脚本

4.1. 什么是 WXS

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

4.2. WXS 的应用场景

wxml 中无法调用在页面的 .js 中定义的函数（不包括事件绑定），但是，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**”。

4.3. WXS 和 JavaScript 的关系

虽然 wxs 的语法类似于 JavaScript，但是wxs 和 JavaScript 是完全不同的两种语言：

① wxs 有自己的数据类型

- number 数值类型、 string 字符串类型、 boolean 布尔类型、 object 对象类型、
- function 函数类型、 array 数组类型、 date 日期类型、 regexp 正则

② wxs 不支持类似于 ES6 及以上的语法形式

- 不支持： let 、 const 、解构赋值、展开运算符、箭头函数、对象属性简写、etc...
- 支持： var 定义变量、普通 function 函数等类似于 ES5 的语法

③ wxs 遵循 CommonJS 规范

- module 对象
- require() 函数
- module.exports 对象

4.4. 基础语法

> 1. 内嵌 wxs 脚本

wxs 代码可以编写在 wxml 文件中的 标签内，就像 Javascript 代码可以编写在 html 文件中的 标签内一样。

wxml 文件中的每个 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：

```
<!-- 定义一个文本，调用wxs中的方法 -->
<view>{{ m1.toUpper(username) }}</view>

<!-- 定义一个wxs的标签，并指定module模块名称 -->
<wxs module="m1">
<!-- 向外暴露一个方法 -->
  module.exports.toUpper = function(str) {
    return str.toUpperCase()
  }
</wxs>
```

> 2. 外联的 wxs 脚本

wxs 代码还可以编写在以 .wxs 为后缀名的文件内，就像 Javascript 代码可以编写在以 .js 为后缀名的文件中一样。示例代码如下：

```
// 1.定义方法
function toLower(str) {
  return str.toLowerCase()
}

// 2.暴露成员
module.exports = {
  toLower: toLower
}
```

在 `wxml` 中引入外联的 `wxs` 脚本时，必须为 标签添加 `module` 和 `src` 属性，其中：

- `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**相对路径**

示例代码如下：

```
<!-- 1.使用外联式引入外部的wxs文件 -->
<wxs module="m2" src="../../utils/tools.wxs"></wxs>
<!-- 2.调用 m2 模块的方法 -->
<view>{{ m2.toLower(country) }}</view>
```

4.5. `WXS` 的特点

> **1. 与 `JavaScript` 不同**

为了降低 `wxs`（`WeiXin Script`）的学习成本， `wxs` 语言在设计时借大量鉴了`JavaScript` 的语法。但是本质上，`wxs` 和 `JavaScript` 是完全不同的两种语言！

> **2. 不能作为组件的事件回调**

`wxs` 典型的应用场景就是“**过滤器**”，经常配合 `Mustache` 语法进行使用，例如：

```
<view>{{ m2.toLower(country) }}</view>
```

但是，在 `wxs` 中定义的函数不能作为组件的事件回调函数。例如，下面的用法是错误的：

```
<button bindtap="m2.toLower(country)"></button>
```

> **3. 隔离性**

隔离性指的是 `wxs` 的运行环境和其他 `JavaScript` 代码是隔离的。体现在如下两方面：

① `wxs` 不能调用 `js` 中定义的函数

② `wxs` 不能调用小程序提供的 `API`

> **4. 性能好**

- 在 `iOS` 设备上，小程序内的 `WXS` 会比 `JavaScript` 代码快 2 ~ 20 倍
- 在 `Android` 设备上，二者的运行效率无差异

5. 案例 - 本地生活（列表页面）

> **1. 演示页面效果以及主要功能**

主要功能如下：

- 页面导航传参
- 上拉触底时加载下一页数据
- 下拉刷新列表数据

2. 列表页面的 `API` 接口

以分页的形式，加载指定分类下商铺列表的数据：

① 接口地址

- https://www.escook.cn/categories/:cate_id/shops
- `URL` 地址中的 `:cate_id` 是动态参数，表示分类的 `Id`

② 请求方式

- `GET` 请求

③ 请求参数

- `_page` 表示请求第几页的数据
- `_limit` 表示每页请求几条数据

3. 判断是否还有下一页数据

- 方式一：

  如果下面的公式成立，则证明没有下一页数据了：

  页码值 * 每页显示多少条数据 >= 总数据条数

`page` * `pageSize` >= `total`

举例1：假设总共有 77 条数据，如果每页显示 10 条数据，则总共分为 8 页，其中第 8 页只有 7 条数据

`page`（7）* `pageSize`（10） >= total（77）不成立, 所以有下一页数据

`page`（8）* `pageSize`（10） >= total（77）成立, 所以没有下一页数据

举例2：假设总共有 80 条数据，如果每页显示 10 条数据，则总共分为 8 页，其中第 8 页面有 10 条数据

`page`（7）* `pageSize`（10） >= total（80）不成立, 所以有下一页数据

`page`（8）* `pageSize`（10） >= total（80）成立, 所以没有下一页数据

- 方式二:

  可以将total的值, 和数组的长度进行对比

4. 具体代码如下

- 页面导航并传参

```
  <!-- 九宫格 -->
<view class="gird-list">
<!-- 循环生成多个gird-item -->
  <navigator url="/pages/shopList/shopList?id={{item.id}}&title={{item.name}}" class="gird-item" wx:for="{{ girdList }}" wx:key="id">
    <image src="{{item.icon}}"></image>
    <text>{{item.name}}</text>
  </navigator>
</view>

  <!-- 新建页面：shopList -->
  /**
   * 页面的初始数据
   */
  data: {
    // 当前被点击了的导航参数
    query: {}
  },

  /**
   * 生命周期函数--监听页面加载
   * 获取到上个页面传递的数据
   */
  onLoad: function (options) {
    this.setData({
      query: options
    })
  },

  /**
   * 生命周期函数--监听页面初次渲染完成
   * 此时可以动态设置导航标题
   */
  onReady: function () {
    wx.setNavigationBarTitle({
      title: this.data.query.title
    })
  },
```

- 渲染页面结构

```
<!--pages/shopList/shopList.wxml-->
<!-- 循环遍历生成商铺列表 -->
<view wx:for="{{shopList}}" wx:key="id" class="shop-item">
  <view class="thumb">
  <image src="{{item.images[0]}}"></image>
  </view>
  <view class="info">
    <text class="shop-title">{{item.name}}</text>
    <!-- 使用wxs 中的方法， 处理手机号码 -->
    <text>电话： {{tools.splitPhone(item.phone)}}</text>
    <text>地址： {{item.address}}</text>
    <text>营业时间： {{item.businessHours}}</text>
  </view>
</view>

<wxs module="tools" src="../../utils/tools.wxs"></wxs>

  /**
   * 页面的初始数据
   */
  data: {
    // 存储上一页传递的参数
    query: {},
    // 商铺列表数据
    shopList: [],
    // 页码
    page: 1,
    // 每页展示的条数
    pageSize: 10,
    // 数据列表总数
    total: 0,
    // 节流阀：false代表没有发起任何数据请求
    isLoading: false
  },
  /***
   * 获取商铺数据列表的方法
   */
  getShopList () {
    // 1. 开启loading
    wx.showLoading({
      title: '数据加载中...',
    })
    // 2.开启节流阀
    this.setData({
      isLoading: true
    })
    // 3.发起网络请求获取数据
    wx.request({
      url: `https://www.escook.cn/categories/${this.data.query.id}/shops`,
      method: 'GET',
      data: {
        _page: this.data.page,
        _limit: this.data.pageSize
      },
```

```
      success: res => {
        // 4.将数据存储在data中
        this.setData({
          shopList: [...this.data.shopList, ...res.data],
          total: res.header['X-Total-Count'] - 0
        })
      },
      complete: () => {
        // 5.关闭loading
        wx.hideLoading()
        // 6.关闭节流阀
        this.setData({
          isLoading: false
        })
      }
    })
  },
  /**
   * 生命周期函数--监听页面加载
   */
  onLoad: function (options) {
    // 将上个页面的参数保存起来
    this.setData({
      query: options
    })

    // 初始化页面，调用获取数据的方法
    this.getShopList()
  },

  /**
   * 生命周期函数--监听页面初次渲染完成
   */
  onReady: function () {
    // 动态设置导航标题
    wx.setNavigationBarTitle({
      title: this.data.query.title,
    })

  },
```

- 上拉触底时加载下一页数据

```
  /**
   * 页面上拉触底事件的处理函数
   */
  onReachBottom: function () {
    /**
     * 如果 total的值 >= shopList的length 代表 没有下一页数据了，无需再发起请求
     * 提示用户：wx.showToast()
     */
    if (this.data.shopList.length >= this.data.total) return wx.showToast({
      title: '没有更多了...',
      icon: 'none'
    })

    /**
```

```
      * 如果节流阀为开启状态，触发下拉触底事件，不需要再次重复发起请求
      */
    if (this.data.isLoading) return

    /**
     *  修改当前页码 :让页码+1  此处js不支持 递增/递减运算符
     */
    this.setData({
      page: this.data.page + 1
    })

    /**
     * 发起请求，获取下一页数据
     */
    this.getShopList()
  },
```

- 下拉刷新列表数据

  页面配置如下:

```
{
  "usingComponents": {},
      // 开启页面的下拉刷新
  "enablePullDownRefresh": true,
      // 下拉时loading的背景色
  "backgroundColor": "#efefef",
      // 下拉时的背景样式
  "backgroundTextStyle": "dark"
}
```

  `js`的代码如下:

```
/**
   * 页面相关事件处理函数--监听用户下拉动作
   */
  onPullDownRefresh: function () {
    // 1. 重置关键性的数据
    this.setData({
      // 重置页码为1
      page: 1,
      // 清空原有数据，否则我们获取的数据会被追加在末尾
      shopList: [],
      // 清空总数
      total: 0
    })
    // 2. 重新获取最新的数据,传递 关闭下拉刷新的回调回调函数
    this.getShopList(() => {
      wx.stopPullDownRefresh()
    })
  },
```

6. 总结

① 能够知道如何实现页面之间的导航跳转

- 声明式导航、编程式导航

② 能够知道如何实现下拉刷新效果

- `enablePullDownRefresh`、`onPullDownRefresh`

③ 能够知道如何实现上拉加载更多效果

- `onReachBottomDistance`、`onReachBottom`

④ 能够知道小程序中常用的生命周期函数

- 应用生命周期函数：`onLaunch`，`onShow`，`onHide`
- 页面生命周期函数：`onLoad`，`onShow`，`onReady`，`onHide`，`onUnload`