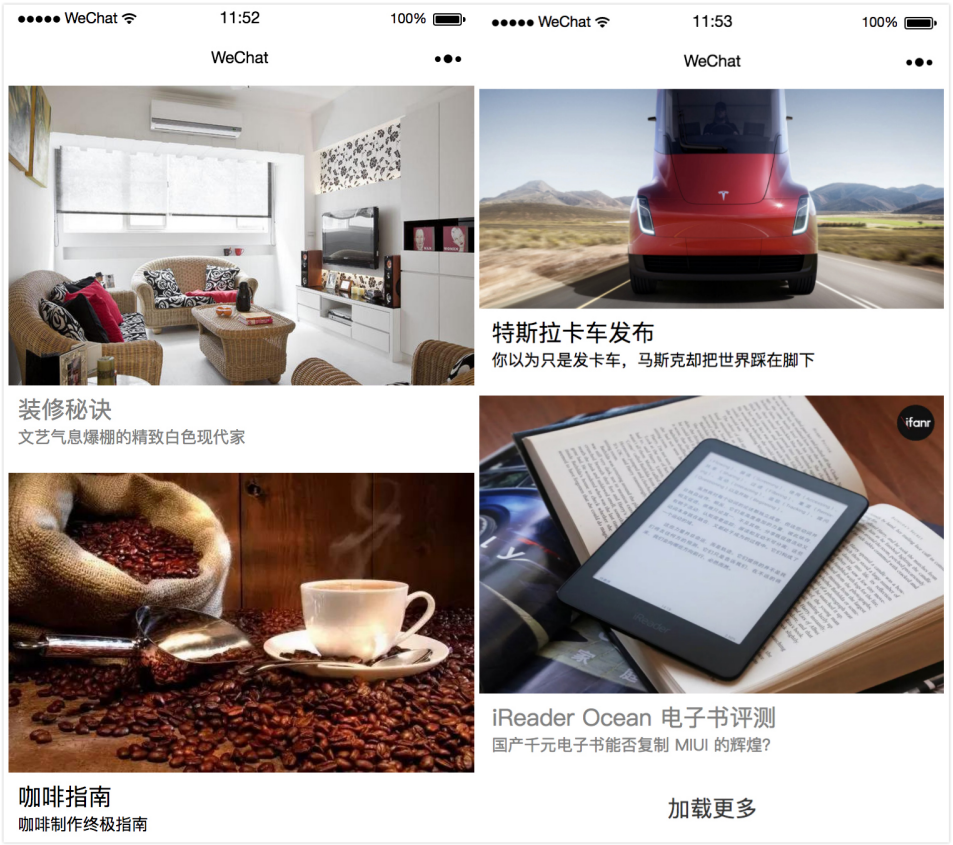
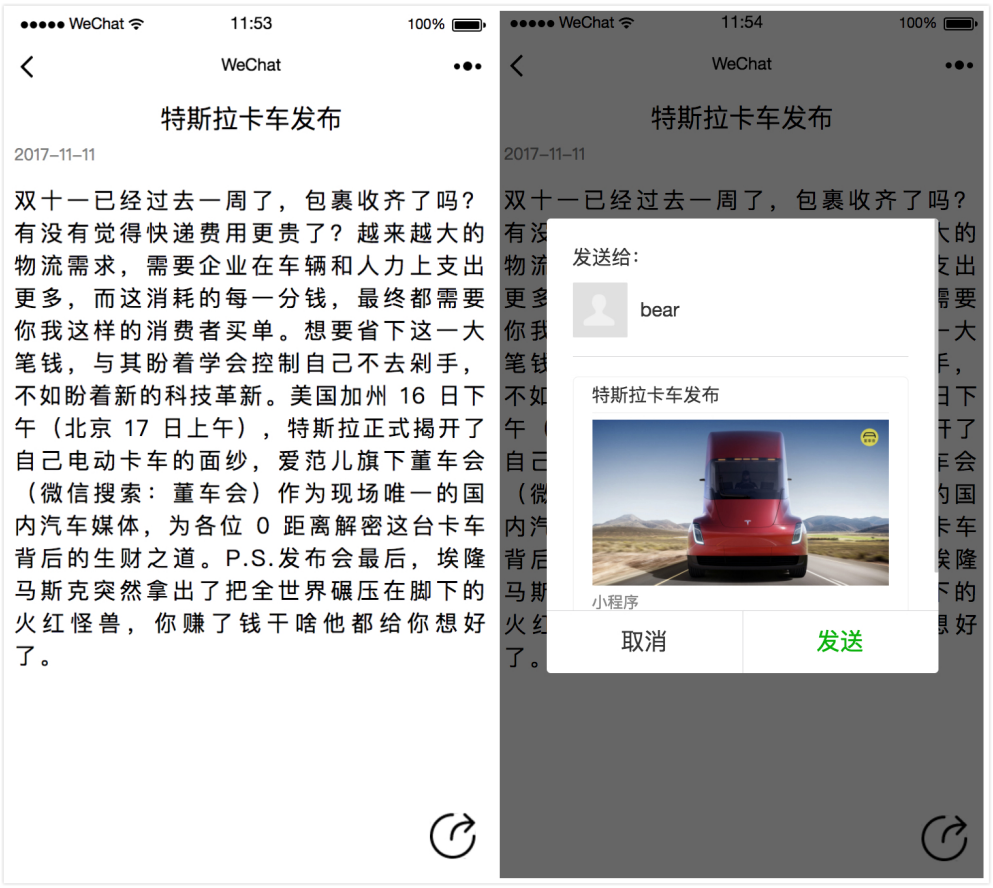
第四章 在小程序里阅读文章（上）

1. 项目简介

这一章节，我将一步一步教大家如何搭建一个文章展示小程序，从最开始使用静态数据，逐渐添加加载更多数据、分享、记录阅读信息等功能，一点点解答开发一个内容展示类小程序所需的基本功能的实现方法。





2. 搭建简单的页面

这里，我们先从最简单的搭建页面框架开始。

首先，我们先理清我们所需要的页面。我们这里只需要用到一个文章列表展示页，和一个文章详情页。在列表页选择一篇文章，点击便进入详情页。

理清了我们需要的页面后，在项目的 pages 文件夹中创建相应的页面，并且在 app.json 文件中进行声明。

```
// app.json
{
  "pages": [
```

```
    "pages/index/index",
    "pages/detail/index"
  ],
}
```

文章列表

列表循环

我们需要在文章列表展示页将我们文章的简要信息，以列表的形式展示出来。这里主要会用到小程序的 wx:for 属性。在小程序组件上使用 wx:for 属性绑定一个数组，即可使用数组中各项的数据对该组件进行重复渲染，配合 wx:for 一起用的属性如下：

- wx:for： 指定需要遍历的数组
- wx:for-item：遍历数组时，指定当前项的别名
- wx:for-index：当前遍历到数组的第几个
- wx:key：主要用于性能优化

除了 wx:for 属性，其它并非必须要使用的。我们通过一个例子，来看看它是如何使用的：

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

页面生命周期

一般情况下，我们都会从服务器获取文章列表数据，这就涉及到了在页面的哪个生命周期去获取数据。

一个页面的生命周期如下表所示：

生命周期	触发时机	说明
onLoad	页面加载	一个页面只会调用一次
onShow	页面显示	每次打开页面都会调用一次
onReady	页面初次渲染完成	视图层渲染完毕，一个页面只会调用一次
oHide	页面隐藏	当执行 navigateTo 或者切换底部 tab 时会被调用
onUnload	页面卸载	当执行 redirectTo 或者 navigateBack 时会被调用

我们可以通过一个简单的试验来了解这些生命周期的触发时机。

首先新建三个页面 index 页面，test 页面和 another 页面。在 index 页面添加一个 navigator 组件，url 指向 test 页面；在 test 页面添加一个 navigator 组件，url 指向 another 页面，同时在其 js 文件中，添加以下代码：

```
Page({
  onLoad: function (options) {
    console.log('onLoad')
  },

  onReady: function () {
    console.log('onReady')
  },

  onShow: function () {
    console.log('onShow')
```

```
  },

  onHide: function () {
    console.log('onHide')
  },

  onUnload: function () {
    console.log('onUnload')
  },
})
```

当点击 index 页面的 navigator 组件时，跳转到 test 页面，此时会依次触发 onLoad, onShow, onReady 方法，再点击 test 页面上的 navigator 组件时，跳转到 another 页面，则会触发 onHide 方法；点击返回，页面回到 test 页面，此时会触发 onShow 方法；再次点击返回，回到 index 页面，则会触发 onUnload 方法。

了解了不同生命周期的触发时机后，我们就可以很好地选择一个适合的时机发起我们的数据请求了。如果你只想在页面载入的时候拉取一次数据，你可以选择 onLoad；如果你的数据会在其它页面被改到，你需要更频繁地更新数据，你可以选 onShow；如果你想在页面渲染完成后加载获取数据，那就选择 onReady。后面拉取文章列表信息，我是选择在 onLoad 的时候加载的。

3. 功能实现

为了更清晰地讲解如何开发文章列表展示功能，我们会使用一些静态数据来替代真实的数据库数据。

这里先定义第一页的数据，为了方便，我们规定一页就只有四条数据。

```
// pages/index/index.js
const firstPage = [{
  id: '1',
  title: '装修秘诀',
  description: '文艺气息爆棚的精致白色现代家',
  cover: 'http://xxx.xxx/xxx.jpg',
},
 ...
{
  id: '4',
  title: '咖啡指南',
  description: '咖啡制作终极指南',
  cover: 'http://xxx.xxx/xxx.jpg',
}]
```

我们在 data 中加入 articles 属性，用于存放获取到的文章列表数据，并且在 onLoad 方法中调用获取数据的方法，通过使用 setTimeout 来模拟请求数据的效果。

```
// pages/index/index.js
Page({
  data: {
    articles: [],
  },
  onLoad: function() {
    this.getArticles()
  },
  getArticles: function() {
    setTimeout(() => {
```

```
      this.setData({
        articles: firstPage,
      })
    }, 1000)
  },
)}
```

编写好 javascript 逻辑，获取到我们需要的数据后，我们就可以在 wxml 里将数据赋给 wx:for ，渲染文章列表。

```
// pages/index/index.wxml
<view wx:for="{{articles}}" wx:for-
item="article" wx:key="id">
  <image  src="{{article.cover}}" data-
id="{{article.id}}" />
  <view>
     <view data-
id="{{article.id}}">{{article.title}}</view>
    <view>{{article.description}}</view>
  </view>
</view>
```

上面除了 wx:for 属性，其它并非必须的，可以按照我们的需要使用。这里我们重点讲一下 wx:key。

当我们做列表渲染时，如果不提供 wx:key 的话，控制台会抛出如下警告：

```
Now you can provide attr "wx:key" for a "wx:for" to impro
ve performance.
```

警告并非错误，因此不修改也是没关系的，但是，我们最好了解一下为什么小程序想要我们添加 wx:key 属性，了解了这个，我们就可以更好地决定是否需要加上这个属性了。

我们来看看，小程序官方文档是如何解释的：

使用 wx:key 可以指定列表中项目的唯一的标识，如果列表中项目的位置动态改变或者有新的项目添加到列表中，原来的列表中的项目可以保持自己的特征和状态（如 <input/> 中的输入内容，<switch/> 的选中状态）不变，即他们可以只进行重新排序而不是重新创建。

可以看出，wx:key 其实是起到性能优化的作用，当我们的列表数据发生变化时，相应的视图也会发生变化，从而导致 DOM 的重新创建或重新排序，很明显，重新排序比重新创建性能要好，因此我们要避免不必要的重新创建。

假设我们对数组 [A, B, C, D] 进行列表渲染， 现在要在 B 和 C 之间加入 F，如果不使用 wx:key 的话，进行 diff 后，发现排在第一和第二位的 A 和 B 没有发生改变，而第三为和第四位从 C 和 D 分别变为了 F 和 C，因此这里需要创建的节点就包括 F ，C，D。

再来看看加了 wx:key 的情况，尽管 C 和 D 的位置都向下挪了一位，但他们的 key 是没有发生变化的，因此判定为重新排序而不是重新创建。

关于列表渲染，还有一个小技巧想要和各位分享一下，那就是使用 <block> 组件。前面我们将 wx:for 放在了 view 组件上，如下：

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

这里渲染出来的每个列表项都将会被 <view> 所包含，但有时候，我们就是需要渲染一个包含多节点的结构块，这是 <block> 组件就派上用场了，<block> 并不会真实的被渲染出来：

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

关于列表渲染，我就讲这么多了，更多细节可以参考小程序开发文档，框架-视图层-WXML-列表渲染这一小节。接下来我们讲一讲如何实现跳转到文章详情页。

添加跳转文章详情页功能

仅仅有文章列表还不够，我们还需要允许用户选择某篇文章，点击进入到它的详情页面。前面我们已经定义好了页面，现在只需要在点击封面和点击文章标题的时候，做下路由跳转，跳到详情页即可。

实现路由跳转的两种方式

在小程序中，你可以使用以下两种方式实现路由跳转。

1. 在 wxml 代码中添加跳转逻辑

通过使用 navigator 组件，并配上相应的属性，即可实现路由跳转。这种实现方式的优点是，逻辑清晰，通过浏览 wxml 即可了解到页面的调整逻辑，缺点是灵活性差。

```
<navigator url="{{pageUrl}}" open-type="navigator">跳转
</navigator>
```

navigator 组件支持以下几个重要属性：

属性名	类型	默认值	说明
url	String		应用内的跳转链接
open-type	String	navigate	跳转方式
hover-class	String	navigator-hover	点击时的样式类

其中，open-type 的有效值包括：

- navigate：保留当前页面，跳转到应用内的某个页面
- redirect：关闭当前页面，跳转到应用内的某个页面
- switchTab：跳转到 tabBar 页面，并关闭其他所有非 tabBar 页面
- navigateBack：关闭当前页面，返回上一页面或多级页面
- reLaunch：关闭所有页面，打开到应用内的某个页面

更多属性细节，可查看小程序文档，组件-导航 这一小节。

2. 用 javascript 代码中添加跳转逻辑

小程序也支持使用 API 来实现路由跳转，包括 wx.navigateTo, wx.redirectTo, wx.switchTab, wx.navigateBack, wx.reLaunch, 与 navigate 组件的 open-type 属性相对应。

这种实现方法的有点事灵活性强，大部分情况下，我们做路由调整的同时，还需要做其它的动作，用 javascript 代码来控制路由跳转将很容易做到这一点，因此，我们将采用这种方法来实现跳转文章详情页功能。

3. 在 JavaScript 中实现路由跳转

我们想要的效果是，点击文章封面和点击文章标题的时候，会跳转到文章详情页，所以我们需要给这两个节点绑上相应的事件。同时，为了告知点击的具体文章，我们要在这两个节点上加上 data-id="{{article.id}}" 属性。

```
// pages/index/index.wxml
<image src="{{article.cover}}" data-
id="{{article.id}}" bindtap="toDetailPage" />
  <view class="article-item__desc>
    <view data-
id="{{article.id}}" bindtap="toDetailPage">{{article.titl
e}}</view>
    <view>{{article.description}}</view>
  </view>
</view>
```

然后实现 toDetailPage 事件的逻辑：

```
// pages/index/index.js
toDetailPage: function(e) {
  let id = e.currentTarget.dataset.id
  wx.navigateTo({
    url: `../detail/index?id=${id}`
  })
}
```

跳转的功能已经实现了，现在需要解决的是，如何在详情页显示相应文章的详情。

在详情页面，我们需要知道到底哪篇文章需要被显示，前面，我们在跳转链接上加了 id=${id} 参数，因此，我们只需要在 onLoad 方法中获取到这个参数，并发送获取相应文章详情的请求即可。同样，我们使用静态数据，并模拟一下获取文章详情的请求。

```
// pages/detail/index.js
const articleInfo = {
  title: '特斯拉卡车发布',
  category: '科技',
  poster: 'https://xxx.xxx/xxx.jpg',
  content: '特斯拉卡车发布',
  created_at: '2017-11-11'
}

Page({
  data: {
    article: {},
  },

  onLoad: function(option) {
    this.getArticle(option.id)
  },

  getArticle: function(id) {
    this.setData({article: articleInfo})
  },
})
<view>{{article.title}}</view>
<view>{{article.created_at}}</view>
<view>{{article.content}}</view>
```