

## 6. 商品列表

### 6.0 创建 goodslist 分支

运行如下的命令，基于 master 分支在本地创建 goodslist 子分支，用来开发商品列表相关的功能：

```
1 git checkout -b search
```

### 6.1 定义请求参数对象

1. 为了方便发起请求获取商品列表的数据，我们要根据接口的要求，事先定义一个**请求参数对象**：

```
1 data() {  
2     return {  
3         // 请求参数对象  
4         queryObj: {  
5             // 查询关键词  
6             query: '',  
7             // 商品分类Id  
8             cid: '',  
9             // 页码值  
10            pagenum: 1,  
11            // 每页显示多少条数据  
12            pagesize: 10  
13        }  
14    }  
15 }
```

2. 将页面跳转时携带的参数，转存到 `queryObj` 对象中：

```
1 onLoad(options) {  
2     // 将页面参数转存到 this.queryObj 对象中  
3     this.queryObj.query = options.query || ''  
4     this.queryObj.cid = options.cid || ''  
5 }
```

3. 为了方便开发商品分类页面，建议大家通过**微信开发者工具**，新建商品列表页面的**编译模式**：

自定义编译条件

模式名称	subpkg/goods_list/goods_list
启动页面	subpkg/goods_list/goods_list
启动参数	cid=5
进入场景	默认
编译设置	<input type="checkbox"/> 下次编译时模拟更新 (需 1.9.90 及以上基础库版本)

删除该模式 取消 确定

## 6.2 获取商品列表数据

1. 在 data 中新增如下的数据节点:

```
1  data() {  
2    return {  
3      // 商品列表的数据  
4      goodsList: [],  
5      // 总数量, 用来实现分页  
6      total: 0  
7    }  
8  }
```

2. 在 onLoad 生命周期函数中, 调用 getGoodsList 方法获取商品列表数据:

```
1  onLoad(options) {  
2    // 调用获取商品列表数据的方法  
3    this.getGoodsList()  
4  }
```

3. 在 methods 节点中, 声明 getGoodsList 方法如下:

```
1  methods: {  
2    // 获取商品列表数据的方法  
3    async getGoodsList() {  
4      // 发起请求  
5      const { data: res } = await  
6      uni.$http.get('/api/public/v1/goods/search', this.queryObj)  
7      if (res.meta.status !== 200) return uni.$showMsg()  
8      // 为数据赋值  
9      this.goodsList = res.message.goods  
10     this.total = res.message.total  
11   }  
12 }
```

## 6.3 渲染商品列表结构

1. 在页面中, 通过 v-for 指令, 循环渲染出商品的 UI 结构:

```

1   <template>
2     <view>
3       <view class="goods-list">
4         <block v-for="(goods, i) in goodsList" :key="i">
5           <view class="goods-item">
6             <!-- 商品左侧图片区域 -->
7             <view class="goods-item-left">
8               <image :src="goods.goods_small_logo || defaultPic"
class="goods-pic"></image>
9             </view>
10            <!-- 商品右侧信息区域 -->
11            <view class="goods-item-right">
12              <!-- 商品标题 -->
13              <view class="goods-name">{{goods.goods_name}}</view>
14              <view class="goods-info-box">
15                <!-- 商品价格 -->
16                <view class="goods-price">¥{{goods.goods_price}}</view>
17              </view>
18            </view>
19          </view>
20        </block>
21      </view>
22    </view>
23  </template>

```

2. 为了防止某些商品的图片不存在，需要在 data 中定义一个默认的图片：

```

1   data() {
2     return {
3       // 默认的空图片
4       defaultPic:
'https://img3.doubanio.com/f/movie/8dd0c794499fe925ae2ae89ee30cd225750457b4/pics/movie/celebrity-default-medium.png'
5     }
6   }

```

并在页面渲染时按需使用：

```

1   <image :src="goods.goods_small_logo || defaultPic" class="goods-pic">
</image>

```

3. 美化商品列表的 UI 结构：

```

1   .goods-item {
2     display: flex;
3     padding: 10px 5px;
4     border-bottom: 1px solid #f0f0f0;
5
6     .goods-item-left {
7       margin-right: 5px;
8
9       .goods-pic {
10        width: 100px;
11        height: 100px;
12        display: block;
13      }
14    }
15

```

```

16     .goods-item-right {
17         display: flex;
18         flex-direction: column;
19         justify-content: space-between;
20
21         .goods-name {
22             font-size: 13px;
23         }
24
25         .goods-price {
26             font-size: 16px;
27             color: #c00000;
28         }
29     }
30 }

```

## 6.4 把商品 item 项封装为自定义组件

1. 在 `components` 目录上鼠标右键，选择 **新建组件**：



2. 将 `goods_list` 页面中，关于商品 item 项相关的 UI 结构、样式、data 数据，封装到 `my-goods` 组件中：

```

1     <template>
2         <view class="goods-item">
3             <!-- 商品左侧图片区域 -->
4             <view class="goods-item-left">
5                 <image :src="goods.goods_small_logo || defaultPic" class="goods-
pic"></image>
6             </view>
7             <!-- 商品右侧信息区域 -->
8             <view class="goods-item-right">
9                 <!-- 商品标题 -->
10                <view class="goods-name">{{goods.goods_name}}</view>

```

```

11     <view class="goods-info-box">
12         <!-- 商品价格 -->
13         <view class="goods-price">¥{{goods.goods_price}}</view>
14     </view>
15 </view>
16 </view>
17 </template>
18
19 <script>
20     export default {
21         // 定义 props 属性，用来接收外界传递到当前组件的数据
22         props: {
23             // 商品的信息对象
24             goods: {
25                 type: Object,
26                 default: {},
27             },
28         },
29         data() {
30             return {
31                 // 默认的空图片
32                 defaultPic:
33                 'https://img3.doubanio.com/f/movie/8dd0c794499fe925ae2ae89ee30cd225750457b
34                 4/pics/movie/celebrity-default-medium.png',
35             }
36         },
37     }
38 </script>
39
40 <style lang="scss">
41     .goods-item {
42         display: flex;
43         padding: 10px 5px;
44         border-bottom: 1px solid #f0f0f0;
45
46         .goods-item-left {
47             margin-right: 5px;
48
49             .goods-pic {
50                 width: 100px;
51                 height: 100px;
52                 display: block;
53             }
54
55             .goods-item-right {
56                 display: flex;
57                 flex-direction: column;
58                 justify-content: space-between;
59
60                 .goods-name {
61                     font-size: 13px;
62                 }
63
64                 .goods-price {
65                     font-size: 16px;
66                     color: #c00000;
67                 }
68             }
69         }
70     }
71 </style>

```

```

67     }
68     }
69 </style>

```

3. 在 `goods_list` 组件中，循环渲染 `my-goods` 组件即可：

```

1 <view class="goods-list">
2   <block v-for="(item, i) in goodsList" :key="i">
3     <!-- 为 my-goods 组件动态绑定 goods 属性的值 -->
4     <my-goods :goods="item"></my-goods>
5   </block>
6 </view>

```

## 6.5 使用过滤器处理价格

1. 在 `my-goods` 组件中，和 `data` 节点同级，声明 `filters` 过滤器节点如下：

```

1 filters: {
2   // 把数字处理为带两位小数点的数字
3   tofixed(num) {
4     return Number(num).toFixed(2)
5   }
6 }

```

2. 在渲染商品价格的时候，通过管道符 `|` 调用过滤器：

```

1 <!-- 商品价格 -->
2 <view class="goods-price">¥{{goods.goods_price | tofixed}}</view>

```

## 6.6 上拉加载更多

### 6.6.1 初步实现上拉加载更多

1. 打开项目根目录中的 `pages.json` 配置文件，为 `subPackages` 分包中的 `goods_list` 页面配置上拉触底的距离：

```

1 "subPackages": [
2   {
3     "root": "subpkg",
4     "pages": [
5       {
6         "path": "goods_detail/goods_detail",
7         "style": {}
8       },
9       {
10        "path": "goods_list/goods_list",
11        "style": {
12          "onReachBottomDistance": 150
13        }
14      },
15      {
16        "path": "search/search",
17        "style": {}
18      }
19    ]
20   }
21 ]

```

```

18     }
19   ]
20 }
21 ]

```

2. 在 `goods_list` 页面中, 和 `methods` 节点同级, 声明 `onReachBottom` 事件处理函数, 用来监听页面上拉触底行为:

```

1  // 触底的事件
2  onReachBottom() {
3    // 让页码值自增 +1
4    this.queryObj.pagenum += 1
5    // 重新获取列表数据
6    this.getGoodsList()
7  }

```

3. 改造 `methods` 中的 `getGoodsList` 函数, 当列表数据请求成功之后, 进行新旧数据的拼接处理:

```

1  // 获取商品列表数据的方法
2  async getGoodsList() {
3    // 发起请求
4    const { data: res } = await
uni.$http.get('/api/public/v1/goods/search', this.queryObj)
5    if (res.meta.status !== 200) return uni.$showMsg()
6
7    // 为数据赋值: 通过展开运算符的形式, 进行新旧数据的拼接
8    this.goodsList = [...this.goodsList, ...res.message.goods]
9    this.total = res.message.total
10  }

```

## 6.6.2 通过节流阀防止发起额外的请求

1. 在 `data` 中定义 `isLoading` 节流阀如下:

```

1  data() {
2    return {
3      // 是否正在请求数据
4      isLoading: false
5    }
6  }

```

2. 修改 `getGoodsList` 方法, 在请求数据前后, 分别打开和关闭节流阀:

```

1  // 获取商品列表数据的方法
2  async getGoodsList() {
3    // ** 打开节流阀
4    this.isLoading = true
5    // 发起请求
6    const { data: res } = await
uni.$http.get('/api/public/v1/goods/search', this.queryObj)
7    // ** 关闭节流阀
8    this.isLoading = false
9
10   // 省略其它代码...
11  }

```

3. 在 `onReachBottom` 触底事件处理函数中，根据节流阀的状态，来决定是否发起请求：

```
1 // 触底的事件
2 onReachBottom() {
3     // 判断是否正在请求其它数据，如果是，则不发起额外的请求
4     if (this.isloading) return
5
6     this.queryObj.pagenum += 1
7     this.getGoodsList()
8 }
```

### 6.6.3 判断数据是否加载完毕

1. 如果下面的公式成立，则证明没有下一页数据了：

```
1 当前的页码值 * 每页显示多少条数据 >= 总数条数
2  pagenum * pagesize >= total
```

2. 修改 `onReachBottom` 事件处理函数如下：

```
1 // 触底的事件
2 onReachBottom() {
3     // 判断是否还有下一页数据
4     if (this.queryObj.pagenum * this.queryObj.pagesize >= this.total)
5         return uni.$showMsg('数据加载完毕!')
6
7     // 判断是否正在请求其它数据，如果是，则不发起额外的请求
8     if (this.isloading) return
9
10    this.queryObj.pagenum += 1
11    this.getGoodsList()
12 }
```

## 6.7 下拉刷新

1. 在 `pages.json` 配置文件中，为当前的 `goods_list` 页面单独开启下拉刷新效果：

```
1 "subPackages": [{
2   "root": "subpkg",
3   "pages": [{
4     "path": "goods_detail/goods_detail",
5     "style": {}
6   }, {
7     "path": "goods_list/goods_list",
8     "style": {
9       "onReachBottomDistance": 150,
10      "enablePullDownRefresh": true,
11      "backgroundColor": "#F8F8F8"
12    }
13   }, {
14     "path": "search/search",
15     "style": {}
16   }
17 ]}]
```



2. 监听页面的 `onPullDownRefresh` 事件处理函数:

```
1 // 下拉刷新事件
2 onPullDownRefresh() {
3   // 1. 重置关键数据
4   this.queryObj.pagenum = 1
5   this.total = 0
6   this.isLoading = false
7   this.goodsList = []
8
9   // 2. 重新发起请求
10  this.getGoodsList(() => uni.stopPullDownRefresh())
11 }
```

3. 修改 `getGoodsList` 函数, 接收 `cb` 回调函数并按需进行调用:

```
1 // 获取商品列表数据的方法
2 async getGoodsList(cb) {
3   this.isLoading = true
4   const { data: res } = await
    uni.$http.get('/api/public/v1/goods/search', this.queryObj)
5   this.isLoading = false
6   // 只要数据请求完毕, 就立即按需调用 cb 回调函数
7   cb && cb()
8
9   if (res.meta.status !== 200) return uni.$showMsg()
10  this.goodsList = [...this.goodsList, ...res.message.goods]
11  this.total = res.message.total
12 }
```

## 6.8 点击商品 item 项跳转到详情页面

1. 将循环时的 `block` 组件修改为 `view` 组件, 并绑定 `click` 点击事件处理函数:

```
1 <view class="goods-list">
2   <view v-for="(item, i) in goodsList" :key="i" @click="gotoDetail(item)">
3     <!-- 为 my-goods 组件动态绑定 goods 属性的值 -->
4     <my-goods :goods="item"></my-goods>
5   </view>
6 </view>
```

2. 在 `methods` 节点中, 定义 `gotoDetail` 事件处理函数:

```
1 // 点击跳转到商品详情页面
2 gotoDetail(item) {
3   uni.navigateTo({
4     url: '/subpkg/goods_detail/goods_detail?goods_id=' + item.goods_id
5   })
6 }
```

## 6.9 分支的合并与提交

1. 将 `goodslist` 分支进行本地提交:

```
1 git add .
2 git commit -m "完成了商品列表页面的开发"
```

2. 将本地的 `goodslist` 分支推送到码云:

```
1 git push -u origin goodslist
```

3. 将本地 `goodslist` 分支中的代码合并到 `master` 分支:

```
1 git checkout master
2 git merge goodslist
3 git push
```

4. 删除本地的 `goodslist` 分支:

```
1 git branch -d goodslist
```

## 7. 商品详情

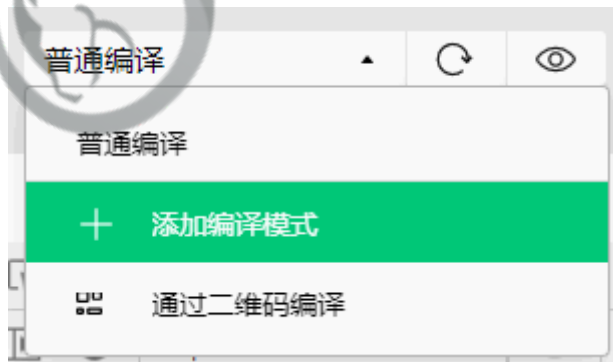
### 7.0 创建 goodsdetail 分支

运行如下的命令，基于 `master` 分支在本地创建 `goodsdetail` 子分支，用来开发商品详情页相关的功能:

```
1 git checkout -b goodsdetail
```

### 7.1 添加商品详情页的编译模式

1. 在微信开发者工具中，点击工具栏上的编译模式下拉菜单，选择 `添加编译模式` 选项:



2. 勾选 `启动页面` 的路径，并填写了 `启动参数` 之后，点击 `确定` 按钮，添加详情页面的编译模式:

自定义编译条件

模式名称	subpkg/goods_detail/goods_detail
启动页面	subpkg/goods_detail/goods_detail
启动参数	goods_id=395
进入场景	默认
编译设置	<input type="checkbox"/> 下次编译时模拟更新 (需 1.9.90 及以上基础库版本)

删除该模式 取消 确定

## 7.2 获取商品详情数据

1. 在 `data` 中定义商品详情的数据节点：

```
1 data() {  
2   return {  
3     // 商品详情对象  
4     goods_info: {}  
5   }  
6 }
```

2. 在 `onLoad` 中获取商品的 Id，并调用请求商品详情的方法：

```
1 onLoad(options) {  
2   // 获取商品 Id  
3   const goods_id = options.goods_id  
4   // 调用请求商品详情数据的方法  
5   this.getGoodsDetail(goods_id)  
6 }
```

3. 在 `methods` 中声明 `getGoodsDetail` 方法：

```
1 methods: {  
2   // 定义请求商品详情数据的方法  
3   async getGoodsDetail(goods_id) {  
4     const { data: res } = await  
uni.$http.get('/api/public/v1/goods/detail', { goods_id })  
5     if (res.meta.status !== 200) return uni.$showMsg()  
6     // 为 data 中的数据赋值  
7     this.goods_info = res.message  
8   }  
9 }
```

## 7.3 渲染商品详情页的 UI 结构

### 7.3.1 渲染轮播图区域

1. 使用 `v-for` 指令，循环渲染如下的轮播图 UI 结构：

```
1      <!-- 轮播图区域 -->
2      <swiper :indicator-dots="true" :autoplay="true" :interval="3000"
      :duration="1000" :circular="true">
3          <swiper-item v-for="(item, i) in goods_info.pics" :key="i">
4              <image :src="item.pics_big"></image>
5          </swiper-item>
6      </swiper>
```

2. 美化轮播图的样式：

```
1      swiper {
2          height: 750rpx;
3
4          image {
5              width: 100%;
6              height: 100%;
7          }
8      }
```

### 7.3.2 实现轮播图预览效果

1. 为轮播图中的 `image` 图片绑定 `click` 事件处理函数：

```
1      <swiper-item v-for="(item, i) in goods_info.pics" :key="i">
2          <!-- 把当前点击的图片的索引，传递到 preview() 处理函数中 -->
3          <image :src="item.pics_big" @click="preview(i)"></image>
4      </swiper-item>
```

2. 在 `methods` 中定义 `preview` 事件处理函数：

```
1      // 实现轮播图的预览效果
2      preview(i) {
3          // 调用 uni.previewImage() 方法预览图片
4          uni.previewImage({
5              // 预览时，默认显示图片的索引
6              current: i,
7              // 所有图片 url 地址的数组
8              urls: this.goods_info.pics.map(x => x.pics_big)
9          })
10     }
```

### 7.3.3 渲染商品信息区域

1. 定义商品信息区域的 UI 结构如下：

```
1      <!-- 商品信息区域 -->
2      <view class="goods-info-box">
3          <!-- 商品价格 -->
4          <view class="price">¥{{goods_info.goods_price}}</view>
5          <!-- 信息主体区域 -->
6          <view class="goods-info-body">
7              <!-- 商品名称 -->
8              <view class="goods-name">{{goods_info.goods_name}}</view>
9              <!-- 收藏 -->
10             <view class="favi">
```

```

11     <uni-icons type="star" size="18" color="gray"></uni-icons>
12     <text>收藏</text>
13 </view>
14 </view>
15 <!-- 运费 -->
16 <view class="yf">快递: 免运费</view>
17 </view>

```

## 2. 美化商品信息区域的样式:

```

1  // 商品信息区域的样式
2  .goods-info-box {
3    padding: 10px;
4    padding-right: 0;
5
6    .price {
7      color: #c00000;
8      font-size: 18px;
9      margin: 10px 0;
10 }
11
12 .goods-info-body {
13   display: flex;
14   justify-content: space-between;
15
16   .goods-name {
17     font-size: 13px;
18     padding-right: 10px;
19   }
20   // 收藏区域
21   .favi {
22     width: 120px;
23     font-size: 12px;
24     display: flex;
25     flex-direction: column;
26     justify-content: center;
27     align-items: center;
28     border-left: 1px solid #efefef;
29     color: gray;
30   }
31 }
32
33 // 运费
34 .yf {
35   margin: 10px 0;
36   font-size: 12px;
37   color: gray;
38 }
39 }

```

### 7.3.4 渲染商品详情信息

1. 在页面结构中, 使用 `rich-text` 组件, 将带有 HTML 标签的内容, 渲染为小程序的页面结构:

```

1  <!-- 商品详情信息 -->
2  <rich-text :nodes="goods_info.goods_introduce"></rich-text>

```

2. 修改 `getGoodsDetail` 方法，从而解决图片底部 空白间隙 的问题：

```
1 // 定义请求商品详情数据的方法
2 async getGoodsDetail(goods_id) {
3   const { data: res } = await uni.$http.get('/api/public/v1/goods/detail',
4     { goods_id })
5   if (res.meta.status !== 200) return uni.$showMsg()
6   // 使用字符串的 replace() 方法，为 img 标签添加行内的 style 样式，从而解决图片底部
  空白间隙的问题
7   res.message.goods_introduce = res.message.goods_introduce.replace(/<img
  /g, '<img style="display:block;" ')
8   this.goods_info = res.message
9 }
```

3. 解决 `.webp` 格式图片在 `ios` 设备上无法正常显示的问题：

```
1 // 定义请求商品详情数据的方法
2 async getGoodsDetail(goods_id) {
3   const { data: res } = await uni.$http.get('/api/public/v1/goods/detail',
4     { goods_id })
5   if (res.meta.status !== 200) return uni.$showMsg()
6   // 使用字符串的 replace() 方法，将 webp 的后缀名替换为 jpg 的后缀名
7   res.message.goods_introduce = res.message.goods_introduce.replace(/<img
  /g, '<img style="display:block;" ').replace(/webp/g, 'jpg')
8   this.goods_info = res.message
9 }
```

### 7.3.5 解决商品价格闪烁的问题

1. 导致问题的原因：在商品详情数据请求回来之前，data 中 `goods_info` 的值为 `{}`，因此初次渲染页面时，会导致 商品价格、商品名称 等闪烁的问题。
2. 解决方案：判断 `goods_info.goods_name` 属性的值是否存在，从而使用 `v-if` 指令控制页面的显示与隐藏：

```
1 <template>
2   <view v-if="goods_info.goods_name">
3     <!-- 省略其它代码 -->
4   </view>
5 </template>
```

## 7.4 渲染详情页底部的商品导航区域

### 7.4.1 渲染商品导航区域的 UI 结构

基于 uni-ui 提供的 `GoodsNav` 组件来实现商品导航区域的效果

1. 在 data 中，通过 `options` 和 `buttonGroup` 两个数组，来声明商品导航组件的按钮配置对象：

```
1 data() {
2   return {
```

```

3      // 商品详情对象
4      goods_info: {},
5      // 左侧按钮组的配置对象
6      options: [{
7          icon: 'shop',
8          text: '店铺'
9      }, {
10         icon: 'cart',
11         text: '购物车',
12         info: 2
13     }],
14     // 右侧按钮组的配置对象
15     buttonGroup: [{
16         text: '加入购物车',
17         backgroundColor: '#ff0000',
18         color: '#fff'
19     },
20     {
21         text: '立即购买',
22         backgroundColor: '#ffa200',
23         color: '#fff'
24     }
25 ]
26 }
27 }

```

## 2. 在页面中使用 `uni-goods-nav` 商品导航组件:

```

1      <!-- 商品导航组件 -->
2      <view class="goods_nav">
3          <!-- fill 控制右侧按钮的样式 -->
4          <!-- options 左侧按钮的配置项 -->
5          <!-- buttonGroup 右侧按钮的配置项 -->
6          <!-- click 左侧按钮的点击事件处理函数 -->
7          <!-- buttonClick 右侧按钮的点击事件处理函数 -->
8          <uni-goods-nav :fill="true" :options="options"
:buttonGroup="buttonGroup" @click="onClick" @buttonClick="buttonClick" />
9      </view>

```

## 3. 美化商品导航组件，使之固定在页面最底部:

```

1      .goods-detail-container {
2          // 给页面外层的容器，添加 50px 的内padding，
3          // 防止页面内容被底部的商品导航组件遮盖
4          padding-bottom: 50px;
5      }
6
7      .goods_nav {
8          // 为商品导航组件添加固定定位
9          position: fixed;
10         bottom: 0;
11         left: 0;
12         width: 100%;
13     }

```

## 7.4.2 点击跳转到购物车页面

1. 点击商品导航组件左侧的按钮，会触发 `uni-goods-nav` 的 `@click` 事件处理函数，事件对象 `e` 中会包含当前点击的按钮相关的信息：

```
1 // 左侧按钮的点击事件处理函数
2 onClick(e) {
3   console.log(e)
4 }
```

打印的按钮信息对象如下：

```
▼ {index: 1, content: {...}} ⓘ
  ► content: {icon: "cart", text: "购物车", info: 2}
  index: 1
  ► __proto__: Object
```

2. 根据 `e.content.text` 的值，来决定进一步的操作：

```
1 // 左侧按钮的点击事件处理函数
2 onClick(e) {
3   if (e.content.text === '购物车') {
4     // 切换到购物车页面
5     uni.switchTab({
6       url: '/pages/cart/cart'
7     })
8   }
9 }
```

## 7.5 分支的合并与提交

1. 将 `goodsdetail` 分支进行本地提交：

```
1 git add .
2 git commit -m "完成了商品详情页面的开发"
```

2. 将本地的 `goodsdetail` 分支推送到码云：

```
1 git push -u origin goodsdetail
```

3. 将本地 `goodsdetail` 分支中的代码合并到 `master` 分支：

```
1 git checkout master
2 git merge goodsdetail
3 git push
```

4. 删除本地的 `goodsdetail` 分支：

```
1 git branch -d goodsdetail
```

## 8. 加入购物车

### 8.0 创建 cart 分支

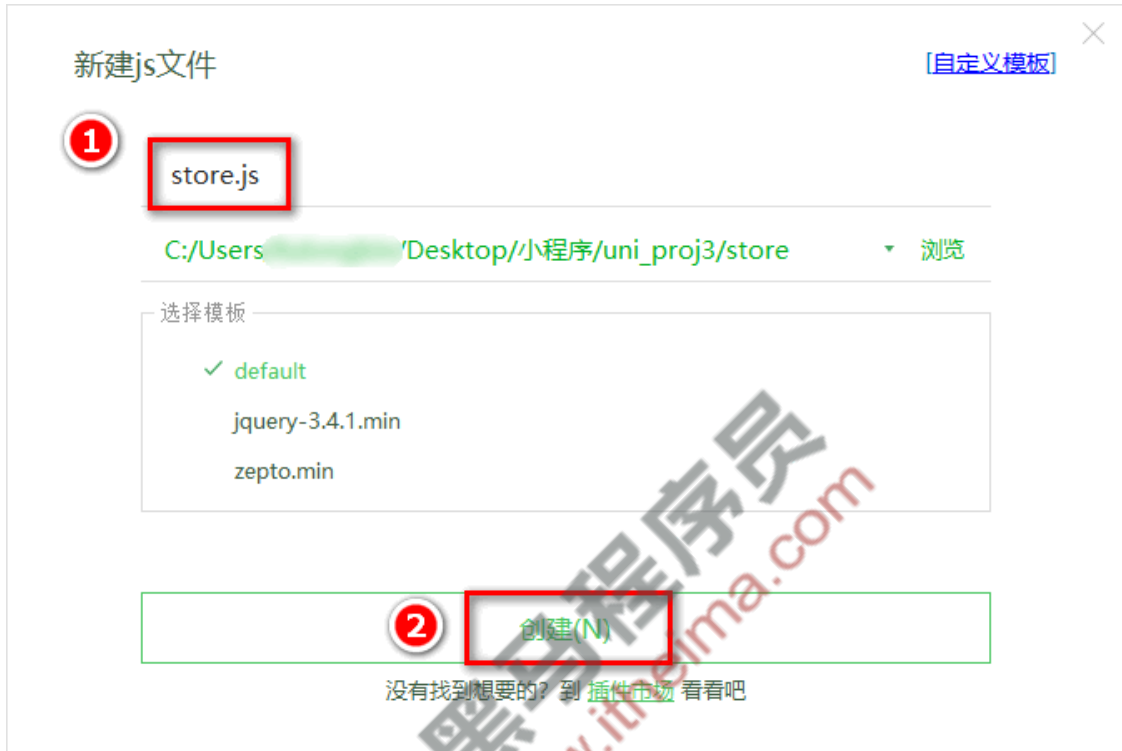
运行如下的命令，基于 `master` 分支在本地创建 `cart` 子分支，用来开发购物车相关的功能：



```
1 git checkout -b cart
```

## 8.1 配置 vuex

1. 在项目根目录中创建 `store` 文件夹，专门用来存放 vuex 相关的模块
2. 在 `store` 目录上鼠标右键，选择 `新建 -> js文件`，新建 `store.js` 文件：



3. 在 `store.js` 中按照如下 4 个步骤初始化 Store 的实例对象：

```
1 // 1. 导入 Vue 和 Vuex
2 import Vue from 'vue'
3 import Vuex from 'vuex'
4
5 // 2. 将 Vuex 安装为 Vue 的插件
6 Vue.use(Vuex)
7
8 // 3. 创建 Store 的实例对象
9 const store = new Vuex.Store({
10   // TODO: 挂载 store 模块
11   modules: {},
12 })
13
14 // 4. 向外共享 Store 的实例对象
15 export default store
```

4. 在 `main.js` 中导入 `store` 实例对象并挂载到 Vue 的实例上：

```

1 // 1. 导入 store 的实例对象
2 import store from './store/store.js'
3
4 // 省略其它代码...
5
6 const app = new Vue({
7   ...App,
8   // 2. 将 store 挂载到 Vue 实例上
9   store,
10 })
11 app.$mount()

```

## 8.2 创建购物车的 store 模块

1. 在 `store` 目录上鼠标右键，选择 `新建 -> js文件`，创建购物车的 store 模块，命名为 `cart.js`：



2. 在 `cart.js` 中，初始化如下的 vuex 模块：

```

1 export default {
2   // 为当前模块开启命名空间
3   namespaced: true,
4
5   // 模块的 state 数据
6   state: () => ({
7     // 购物车的数组，用来存储购物车中每个商品的信息对象
8     // 每个商品的信息对象，都包含如下 6 个属性：
9     // { goods_id, goods_name, goods_price, goods_count,
10    goods_small_logo, goods_state }
11     cart: [],
12   }),
13
14   // 模块的 mutations 方法
15   mutations: {},

```

```

15
16     // 模块的 getters 属性
17     getters: {},
18 }

```

3. 在 `store/store.js` 模块中，导入并挂载购物车的 `vuex` 模块，示例代码如下：

```

1  import Vue from 'vue'
2  import Vuex from 'vuex'
3  // 1. 导入购物车的 vuex 模块
4  import moduleCart from './cart.js'
5
6  Vue.use(Vuex)
7
8  const store = new Vuex.Store({
9    // TODO: 挂载 store 模块
10   modules: {
11     // 2. 挂载购物车的 vuex 模块，模块内成员的访问路径被调整为 m_cart，例如：
12     //   购物车模块中 cart 数组的访问路径是 m_cart/cart
13     m_cart: moduleCart,
14   },
15 })
16
17 export default store

```

## 8.3 在商品详情页中使用 Store 中的数据

1. 在 `goods_detail.vue` 页面中，修改 `<script></script>` 标签中的代码如下：

```

1  // 从 vuex 中按需导出 mapState 辅助方法
2  import { mapState } from 'vuex'
3
4  export default {
5    computed: {
6      // 调用 mapState 方法，把 m_cart 模块中的 cart 数组映射到当前页面中，作为计算
        // 属性来使用
7      // ...mapState('模块的名称', ['要映射的数据名称1', '要映射的数据名称2'])
8      ...mapState('m_cart', ['cart']),
9    },
10    // 省略其它代码...
11  }

```

注意：今后无论映射 `mutations` 方法，还是 `getters` 属性，还是 `state` 中的数据，都需要指定模块的名称，才能进行映射。

2. 在页面渲染时，可以直接使用映射过来的数据，例如：

```

1  <!-- 运费 -->
2  <view class="yf">快递: 免运费 -- {{cart.length}}</view>

```

## 8.4 实现加入购物车的功能

1. 在 store 目录下的 `cart.js` 模块中，封装一个将商品信息加入购物车的 mutations 方法，命名为 `addToCart`。示例代码如下：

```
1   export default {
2     // 为当前模块开启命名空间
3     namespaced: true,
4
5     // 模块的 state 数据
6     state: () => ({
7       // 购物车的数组，用来存储购物车中每个商品的信息对象
8       // 每个商品的信息对象，都包含如下 6 个属性：
9       // { goods_id, goods_name, goods_price, goods_count,
10      goods_small_logo, goods_state }
11      cart: [],
12    }),
13
14    // 模块的 mutations 方法
15    mutations: {
16      addToCart(state, goods) {
17        // 根据提交的商品的Id，查询购物车中是否存在这件商品
18        // 如果不存在，则 findResult 为 undefined；否则，为查找到的商品信息对象
19        const findResult = state.cart.find((x) => x.goods_id ===
20        goods.goods_id)
21
22        if (!findResult) {
23          // 如果购物车中没有这件商品，则直接 push
24          state.cart.push(goods)
25        } else {
26          // 如果购物车中有这件商品，则只更新数量即可
27          findResult.goods_count++
28        }
29      },
30    },
31
32    // 模块的 getters 属性
33    getters: {},
34  }
```

2. 在商品详情页面中，通过 `mapMutations` 这个辅助方法，把 vuex 中 `m_cart` 模块下的 `addToCart` 方法映射到当前页面：

```
1   // 按需导入 mapMutations 这个辅助方法
2   import { mapMutations } from 'vuex'
3
4   export default {
5     methods: {
6       // 把 m_cart 模块中的 addToCart 方法映射到当前页面使用
7       ...mapMutations('m_cart', ['addToCart']),
8     },
9   }
```

3. 为商品导航组件 `uni-goods-nav` 绑定 `@buttonClick="buttonClick"` 事件处理函数：

```
1   // 右侧按钮的点击事件处理函数
2   buttonClick(e) {
3     // 1. 判断是否点击了 加入购物车 按钮
4     if (e.content.text === '加入购物车') {
5
```

```

6      // 2. 组织一个商品的信息对象
7      const goods = {
8          goods_id: this.goods_info.goods_id,      // 商品的Id
9          goods_name: this.goods_info.goods_name,  // 商品的名称
10         goods_price: this.goods_info.goods_price, // 商品的价格
11         goods_count: 1,                          // 商品的数量
12         goods_small_logo: this.goods_info.goods_small_logo, // 商品的图片
13         goods_state: true                        // 商品的勾选状态
14     }
15
16     // 3. 通过 this 调用映射过来的 addToCart 方法，把商品信息对象存储到购物车中
17     this.addToCart(goods)
18
19 }
20 }

```

## 8.5 动态统计购物车中商品的总数量

1. 在 `cart.js` 模块中，在 `getters` 节点下定义一个 `total` 方法，用来统计购物车中商品的总数量：

```

1      // 模块的 getters 属性
2      getters: {
3          // 统计购物车中商品的总数量
4          total(state) {
5              let c = 0
6              // 循环统计商品的数量，累加到变量 c 中
7              state.cart.forEach(goods => c += goods.goods_count)
8              return c
9          }
10     }

```

2. 在商品详情页面的 `script` 标签中，按需导入 `mapGetters` 方法并进行使用：

```

1      // 按需导入 mapGetters 这个辅助方法
2      import { mapGetters } from 'vuex'
3
4      export default {
5          computed: {
6              // 把 m_cart 模块中名称为 total 的 getter 映射到当前页面中使用
7              ...mapGetters('m_cart', ['total']),
8          },
9      }

```

3. 通过 `watch` 侦听器，监听计算属性 `total` 值的变化，从而动态为购物车按钮的徽标赋值：

```

1      export default {
2          watch: {
3              // 1. 监听 total 值的变化，通过第一个形参得到变化后的新值
4              total(newVal) {
5                  // 2. 通过数组的 find() 方法，找到购物车按钮的配置对象
6                  const findResult = this.options.find((x) => x.text === '购物车')
7
8                  if (findResult) {
9                      // 3. 动态为购物车按钮的 info 属性赋值
10                     findResult.info = newVal

```

```

11     }
12     },
13     },
14 }

```

## 8.6 持久化存储购物车中的商品

1. 在 `cart.js` 模块中，声明一个叫做 `saveToStorage` 的 mutations 方法，此方法负责将购物车中的数据持久化存储到本地：

```

1  // 将购物车中的数据持久化存储到本地
2  saveToStorage(state) {
3      uni.setStorageSync('cart', JSON.stringify(state.cart))
4  }

```

2. 修改 `mutations` 节点中的 `addToCart` 方法，在处理完商品信息后，调用步骤 1 中定义的 `saveToStorage` 方法：

```

1  addToCart(state, goods) {
2      // 根据提交的商品的Id，查询购物车中是否存在这件商品
3      // 如果不存在，则 findResult 为 undefined；否则，为查找到的商品信息对象
4      const findResult = state.cart.find(x => x.goods_id === goods.goods_id)
5
6      if (!findResult) {
7          // 如果购物车中没有这件商品，则直接 push
8          state.cart.push(goods)
9      } else {
10         // 如果购物车中有这件商品，则只更新数量即可
11         findResult.goods_count++
12     }
13
14     // 通过 commit 方法，调用 m_cart 命名空间下的 saveToStorage 方法
15     this.commit('m_cart/saveToStorage')
16 }

```

3. 修改 `cart.js` 模块中的 `state` 函数，读取本地存储的购物车数据，对 `cart` 数组进行初始化：

```

1  // 模块的 state 数据
2  state: () => ({
3      // 购物车的数组，用来存储购物车中每个商品的信息对象
4      // 每个商品的信息对象，都包含如下 6 个属性：
5      // { goods_id, goods_name, goods_price, goods_count, goods_small_logo,
6      goods_state }
7      cart: JSON.parse(uni.getStorageSync('cart')) || '[]')
8  })

```

## 8.7 优化商品详情页的 total 侦听器

1. 使用普通函数的形式定义的 watch 侦听器，在页面首次加载后不会被调用。因此导致了商品详情页在首次加载完毕之后，不会将商品的总数量显示到商品导航区域：

```

1   watch: {
2     // 页面首次加载完毕后，不会调用这个侦听器
3     total(newVal) {
4       const findResult = this.options.find(x => x.text === '购物车')
5       if (findResult) {
6         findResult.info = newVal
7       }
8     }
9   }

```

2. 为了防止这个上述问题，可以使用**对象的形式**来定义 watch 侦听器（详细文档请参考 Vue 官方的 **watch 侦听器** 教程），示例代码如下：

```

1   watch: {
2     // 定义 total 侦听器，指向一个配置对象
3     total: {
4       // handler 属性用来定义侦听器的 function 处理函数
5       handler(newVal) {
6         const findResult = this.options.find(x => x.text === '购物车')
7         if (findResult) {
8           findResult.info = newVal
9         }
10      },
11      // immediate 属性用来声明此侦听器，是否在页面初次加载完毕后立即调用
12      immediate: true
13    }
14  }

```

## 8.8 动态为 tabBar 页面设置数字徽标

需求描述：从商品详情页面导航到购物车页面之后，需要为 tabBar 中的购物车动态设置数字徽标。

1. 把 Store 中的 total 映射到 `cart.vue` 中使用：

```

1   // 按需导入 mapGetters 这个辅助方法
2   import { mapGetters } from 'vuex'
3
4   export default {
5     data() {
6       return {}
7     },
8     computed: {
9       // 将 m_cart 模块中的 total 映射为当前页面的计算属性
10      ...mapGetters('m_cart', ['total']),
11    },
12  }

```

2. 在页面刚显示出来的时候，立即调用 `setBadge` 方法，为 tabBar 设置数字徽标：

```

1   onShow() {
2     // 在页面刚展示的时候，设置数字徽标
3     this.setBadge()
4   }

```

3. 在 `methods` 节点中，声明 `setBadge` 方法如下，通过 `uni.setTabBarBadge()` 为 tabBar 设置数字徽标：

```
1  methods: {
2    setBadge() {
3      // 调用 uni.setTabBarBadge() 方法，为购物车设置右上角的徽标
4      uni.setTabBarBadge({
5        index: 2, // 索引
6        text: this.total + '' // 注意: text 的值必须是字符串，不能是数字
7      })
8    }
9  }
```

## 8.9 将设置 tabBar 徽标的代码抽离为 mixins

注意：除了要在 `cart.vue` 页面中设置购物车的数字徽标，还需要在其它 3 个 tabBar 页面中，为购物车设置数字徽标。

此时可以使用 Vue 提供的 `mixins` 特性，提高代码的可维护性。

1. 在项目根目录中新建 `mixins` 文件夹，并在 `mixins` 文件夹之下新建 `tabbar-badge.js` 文件，用来把设置 tabBar 徽标的代码封装为一个 mixin 文件：

```
1  import { mapGetters } from 'vuex'
2
3  // 导出一个 mixin 对象
4  export default {
5    computed: {
6      ...mapGetters('m_cart', ['total']),
7    },
8    onShow() {
9      // 在页面刚展示的时候，设置数字徽标
10     this.setBadge()
11   },
12   methods: {
13     setBadge() {
14       // 调用 uni.setTabBarBadge() 方法，为购物车设置右上角的徽标
15       uni.setTabBarBadge({
16         index: 2,
17         text: this.total + '', // 注意: text 的值必须是字符串，不能是数字
18       })
19     },
20   },
21 }
```

2. 修改 `home.vue` , `cate.vue` , `cart.vue` , `my.vue` 这 4 个 tabBar 页面的源代码，分别导入 `@/mixins/tabbar-badge.js` 模块并进行使用：



```
1 // 导入自己封装的 mixin 模块
2 import badgeMix from '@mixins/tabbar-badge.js'
3
4 export default {
5   // 将 badgeMix 混入到当前的页面中进行使用
6   mixins: [badgeMix],
7   // 省略其它代码...
8 }
```



黑马程序员  
www.itheima.com