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：

```
git checkout -b search
```

6.1 定义请求参数对象

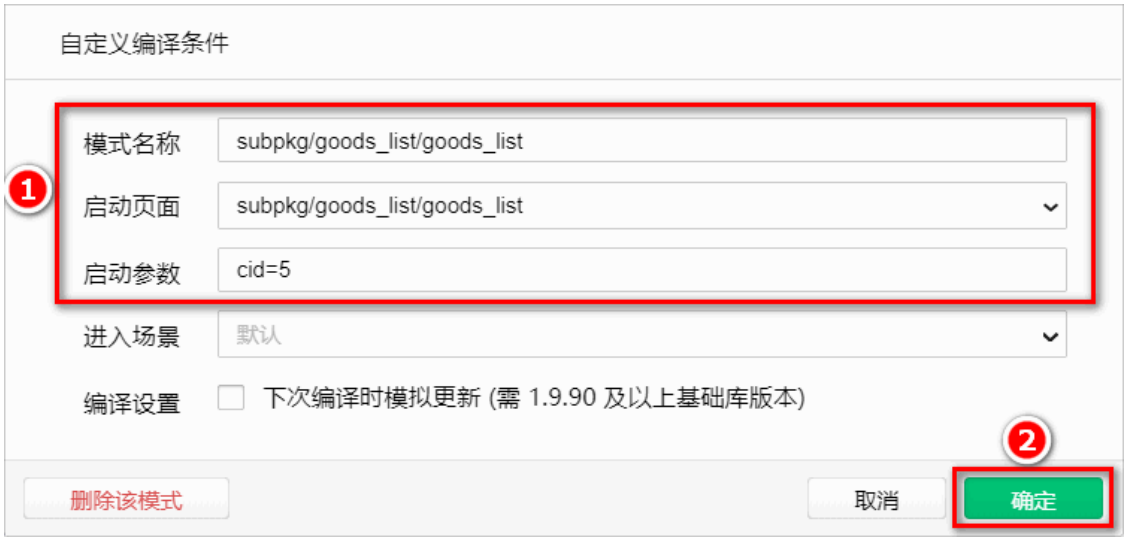
1. 为了方便发起请求获取商品列表的数据，我们要根据接口的要求，事先定义一个**请求参数对象**：

```
data() {
  return {
    // 请求参数对象
    queryObj: {
      // 查询关键词
      query: '',
      // 商品分类Id
      cid: '',
      // 页码值
      pagenum: 1,
      // 每页显示多少条数据
      pagesize: 10
    }
  }
}
```

2. 将页面跳转时**携带的参数**，转存到 `queryObj` 对象中：

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

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**：



6.2 获取商品列表数据

1. 在 data 中新增如下的数据节点：

```
data() {
  return {
    // 商品列表的数据
    goodsList: [],
    // 总数量，用来实现分页
    total: 0
  }
}
```

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

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

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

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

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：

```
<template>
  <view>
    <view class="goods-list">
      <block v-for="(goods, i) in goodsList" :key="i">
        <view class="goods-item">
          <!-- 商品左侧图片区域 -->
          <view class="goods-item-left">
            <image :src="goods.goods_small_logo || defaultPic" class="goods-pic"></image>
          </view>
          <!-- 商品右侧信息区域 -->
          <view class="goods-item-right">
            <!-- 商品标题 -->
            <view class="goods-name">{{goods.goods_name}}</view>
            <view class="goods-info-box">
              <!-- 商品价格 -->
              <view class="goods-price">¥{{goods.goods_price}}</view>
            </view>
          </view>
        </view>
      </block>
    </view>
  </view>
</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：

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

并在页面渲染时按需使用：

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

3. 美化商品列表的 UI 结构：

```
.goods-item {
  display: flex;
  padding: 10px 5px;
  border-bottom: 1px solid #f0f0f0;

  .goods-item-left {
    margin-right: 5px;

    .goods-pic {
      width: 100px;
      height: 100px;
      display: block;
    }
  }

```

```
    .goods-item-right {
      display: flex;
      flex-direction: column;
      justify-content: space-between;

      .goods-name {
        font-size: 13px;
      }

      .goods-price {
        font-size: 16px;
        color: #c00000;
      }
    }
  }
```

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：

```
  <template>
    <view class="goods-item">
      <!-- 商品左侧图片区域 -->
      <view class="goods-item-left">
        <image :src="goods.goods_small_logo || defaultPic" class="goods-
pic"></image>
      </view>
      <!-- 商品右侧信息区域 -->
      <view class="goods-item-right">
        <!-- 商品标题 -->
        <view class="goods-name">{{goods.goods_name}}</view>
```

```
        <view class="goods-info-box">
          <!-- 商品价格 -->
          <view class="goods-price">￥{{goods.goods_price}}</view>
        </view>
      </view>
    </view>
  </template>

  <script>
    export default {
      // 定义 props 属性，用来接收外界传递到当前组件的数据
      props: {
        // 商品的信息对象
        goods: {
          type: Object,
          defaul: {},
        },
      },
      data() {
        return {
          // 默认的空图片
          defaultPic:
'https://img3.doubanio.com/f/movie/8dd0c794499fe925ae2ae89ee30cd225750457b4/pics/movie/celebrity-default-medium.png',
        }
      },
    }
  </script>

  <style lang="scss">
    .goods-item {
      display: flex;
      padding: 10px 5px;
      border-bottom: 1px solid #f0f0f0;

      .goods-item-left {
        margin-right: 5px;

        .goods-pic {
          width: 100px;
          height: 100px;
          display: block;
        }
      }

      .goods-item-right {
        display: flex;
        flex-direction: column;
        justify-content: space-between;

        .goods-name {
          font-size: 13px;
        }

        .goods-price {
          font-size: 16px;
          color: #c00000;
        }
```

```
      }
    }
  </style>
```

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

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

6.5 使用过滤器处理价格

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

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

2. 在渲染商品价格的时候，通过管道符 `|` 调用过滤器：

```
<!-- 商品价格 -->
<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：

```
  "subPackages": [
    {
      "root": "subpkg",
      "pages": [
        {
          "path": "goods_detail/goods_detail",
          "style": {}
        },
        {
          "path": "goods_list/goods_list",
          "style": {
            "onReachBottomDistance": 150
          }
        },
        {
          "path": "search/search",
          "style": {}
```

```
          }
        ]
      }
    ]
```

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

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

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：

```
// 获取商品列表数据的方法
async getGoodsList() {
  // 发起请求
  const { data: res } = await uni.$http.get('/api/public/v1/goods/search', this.queryObj)
  if (res.meta.status !== 200) return uni.$showMsg()

  // 为数据赋值：通过展开运算符的形式，进行新旧数据的拼接
  this.goodsList = [...this.goodsList, ...res.message.goods]
  this.total = res.message.total
}
```

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：

```
data() {
  return {
    // 是否正在请求数据
    isloading: false
  }
}
```

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

```
// 获取商品列表数据的方法
async getGoodsList() {
  // ** 打开节流阀
  this.isloading = true
  // 发起请求
  const { data: res } = await uni.$http.get('/api/public/v1/goods/search', this.queryObj)
  // ** 关闭节流阀
  this.isloading = false

  // 省略其它代码...
}
```

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：

```
// 触底的事件
onReachBottom() {
  // 判断是否正在请求其它数据，如果是，则不发起额外的请求
  if (this.isloading) return

  this.queryObj.pagenum += 1
  this.getGoodsList()
}
```

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：

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

2. 修改 `onReachBottom` 事件处理函数如下：

```
// 触底的事件
onReachBottom() {
  // 判断是否还有下一页数据
  if (this.queryObj.pagenum * this.queryObj.pagesize >= this.total) return uni.$showMsg('数据加载完毕！')

  // 判断是否正在请求其它数据，如果是，则不发起额外的请求
  if (this.isloading) return

  this.queryObj.pagenum += 1
  this.getGoodsList()
}
```

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：

```
"subPackages": [{
  "root": "subpkg",
  "pages": [{
    "path": "goods_detail/goods_detail",
    "style": {}
  }, {
    "path": "goods_list/goods_list",
    "style": {
      "onReachBottomDistance": 150,
      "enablePullDownRefresh": true,
      "backgroundColor": "#F8F8F8"
    }
  }, {
    "path": "search/search",
    "style": {}
  }]
}]
```

2. 监听页面的 `onPullDownRefresh` 事件处理函数：

```
// 下拉刷新的事件
onPullDownRefresh() {
  // 1. 重置关键数据
  this.queryObj.pagenum = 1
  this.total = 0
  this.isloading = false
  this.goodsList = []

  // 2. 重新发起请求
  this.getGoodsList(() => uni.stopPullDownRefresh())
}
```

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

```
// 获取商品列表数据的方法
async getGoodsList(cb) {
  this.isloading = true
  const { data: res } = await uni.$http.get('/api/public/v1/goods/search', this.queryObj)
  this.isloading = false
  // 只要数据请求完毕，就立即按需调用 cb 回调函数
  cb && cb()

  if (res.meta.status !== 200) return uni.$showMsg()
  this.goodsList = [...this.goodsList, ...res.message.goods]
  this.total = res.message.total
}
```

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：

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

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

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

6.9 分支的合并与提交

1. 将 `goodslist` 分支进行本地提交：

```
git add .
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：

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

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

```
git checkout master
git merge goodslist
git push
```

4. 删除本地的 `goodslist` 分支：

```
git branch -d goodslist
```

7. 商品详情

7.0 创建 goodsdetail 分支

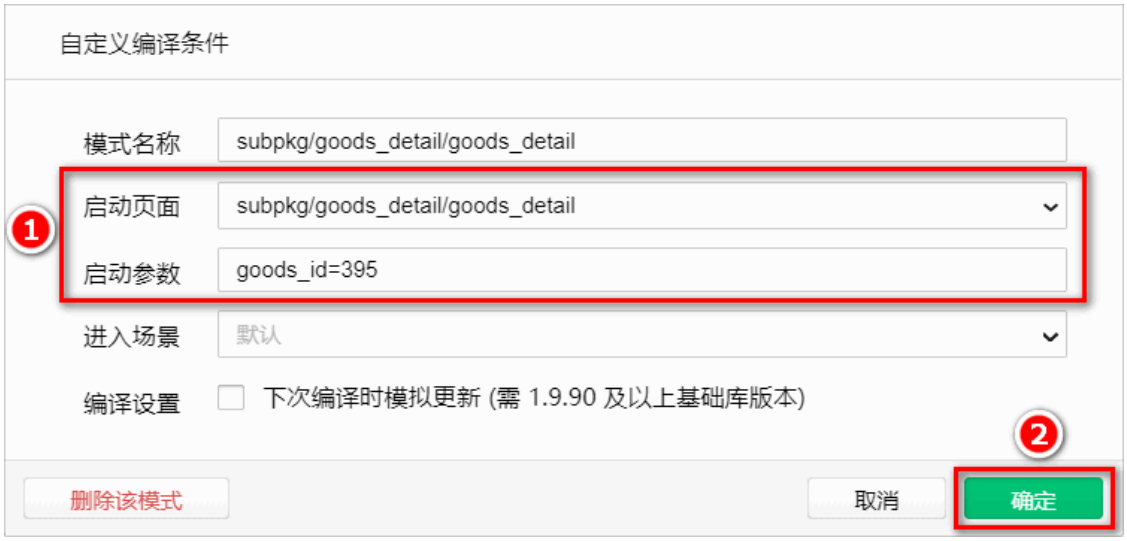
运行如下的命令，基于 master 分支在本地创建 goodsdetail 子分支，用来开发商品详情页相关的功能：

```
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：



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：

```
data() {
  return {
    // 商品详情对象
    goods_info: {}
  }
}
```

2. 在 `onLoad` 中获取商品的 Id，并调用请求商品详情的方法：

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

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

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

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：

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

2. 美化轮播图的样式：

```
swiper {
  height: 750rpx;

  image {
    width: 100%;
    height: 100%;
  }
}
```

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：

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

2. 在 `methods` 中定义 `preview` 事件处理函数：

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

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：

```
<!-- 商品信息区域 -->
<view class="goods-info-box">
  <!-- 商品价格 -->
  <view class="price">￥{{goods_info.goods_price}}</view>
  <!-- 信息主体区域 -->
  <view class="goods-info-body">
    <!-- 商品名称 -->
    <view class="goods-name">{{goods_info.goods_name}}</view>
    <!-- 收藏 -->
    <view class="favi">
```

```
        <uni-icons type="star" size="18" color="gray"></uni-icons>
        <text>收藏</text>
      </view>
    </view>
    <!-- 运费 -->
    <view class="yf">快递：免运费</view>
  </view>
```

2. 美化商品信息区域的样式：

```
// 商品信息区域的样式
.goods-info-box {
  padding: 10px;
  padding-right: 0;

  .price {
    color: #c00000;
    font-size: 18px;
    margin: 10px 0;
  }

  .goods-info-body {
    display: flex;
    justify-content: space-between;

    .goods-name {
      font-size: 13px;
      padding-right: 10px;
    }
    // 收藏区域
    .favi {
      width: 120px;
      font-size: 12px;
      display: flex;
      flex-direction: column;
      justify-content: center;
      align-items: center;
      border-left: 1px solid #efefef;
      color: gray;
    }
  }

  // 运费
  .yf {
    margin: 10px 0;
    font-size: 12px;
    color: gray;
  }
}
```

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：

```
<!-- 商品详情信息 -->
<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：

```
// 定义请求商品详情数据的方法
async getGoodsDetail(goods_id) {
  const { data: res } = await uni.$http.get('/api/public/v1/goods/detail', { goods_id })
  if (res.meta.status !== 200) return uni.$showMsg()

  // 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
  res.message.goods_introduce = res.message.goods_introduce.replace(/<img/g, '<img style="display:block;" ')
  this.goods_info = res.message
}
```

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

```
// 定义请求商品详情数据的方法
async getGoodsDetail(goods_id) {
  const { data: res } = await uni.$http.get('/api/public/v1/goods/detail', { goods_id })
  if (res.meta.status !== 200) return uni.$showMsg()

  // 使用字符串的 replace() 方法，将 webp 的后缀名替换为 jpg 的后缀名
  res.message.goods_introduce = res.message.goods_introduce.replace(/<img/g, '<img style="display:block;" ').replace(/webp/g, 'jpg')
  this.goods_info = res.message
}
```

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：

```
<template>
  <view v-if="goods_info.goods_name">
    <!-- 省略其它代码 -->
  </view>
</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：

```
data() {
  return {
```

```
      // 商品详情对象
      goods_info: {},
      // 左侧按钮组的配置对象
      options: [{
        icon: 'shop',
        text: '店铺'
      }, {
        icon: 'cart',
        text: '购物车',
        info: 2
      }],
      // 右侧按钮组的配置对象
      buttonGroup: [{
          text: '加入购物车',
          backgroundColor: '#ff0000',
          color: '#fff'
        },
        {
          text: '立即购买',
          backgroundColor: '#ffa200',
          color: '#fff'
        }
      ]
    }
  }
```

2. 在页面中使用 `uni-goods-nav` 商品导航组件：

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

3. 美化商品导航组件，使之固定在页面最底部：

```
.goods-detail-container {
  // 给页面外层的容器，添加 50px 的内padding，
  // 防止页面内容被底部的商品导航组件遮盖
  padding-bottom: 50px;
}

.goods_nav {
  // 为商品导航组件添加固定定位
  position: fixed;
  bottom: 0;
  left: 0;
  width: 100%;
}
```

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：

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

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：

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

7.5 分支的合并与提交

1. 将 `goodsdetail` 分支进行本地提交：

```
git add .
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：

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

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

```
git checkout master
git merge goodsdetail
git push
```

4. 删除本地的 `goodsdetail` 分支：

```
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：

```
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. 导入 Vue 和 Vuex
import Vue from 'vue'
import Vuex from 'vuex'

// 2. 将 Vuex 安装为 Vue 的插件
Vue.use(Vuex)

// 3. 创建 Store 的实例对象
const store = new Vuex.Store({
  // TODO：挂载 store 模块
  modules: {},
})

// 4. 向外共享 Store 的实例对象
export default store
```

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

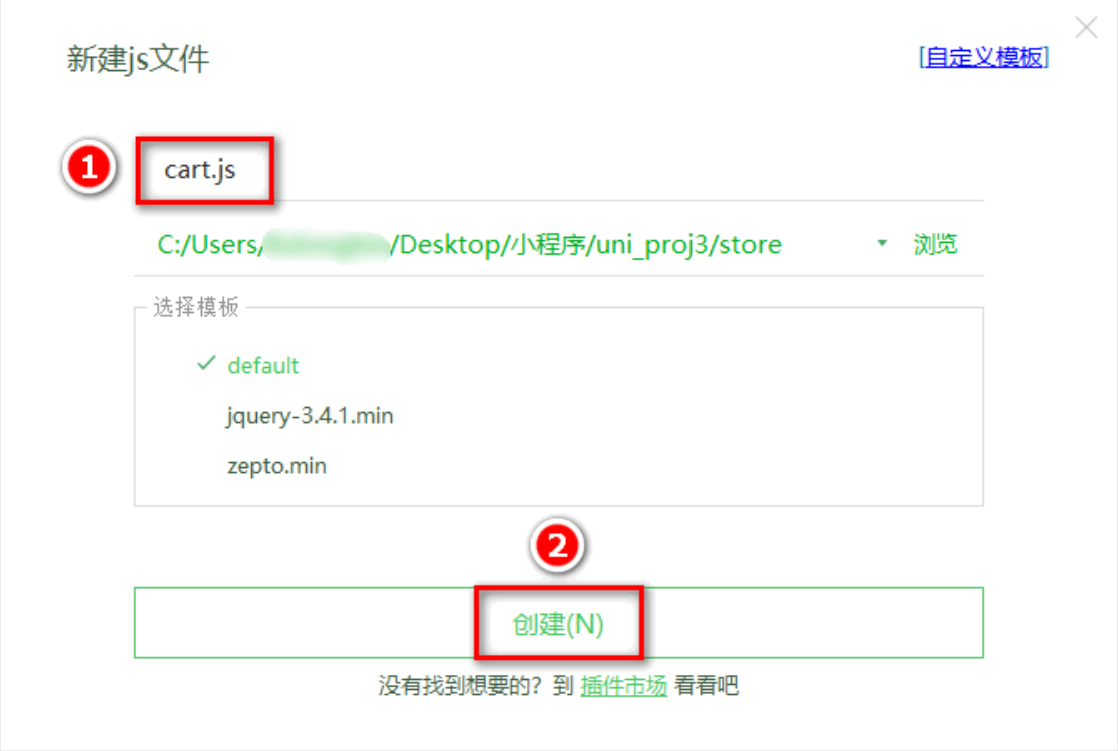
```
// 1. 导入 store 的实例对象
import store from './store/store.js'

// 省略其它代码...

const app = new Vue({
  ...App,
  // 2. 将 store 挂载到 Vue 实例上
  store,
})
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：

```
export default {
  // 为当前模块开启命名空间
  namespaced: true,

  // 模块的 state 数据
  state: () => ({
    // 购物车的数组，用来存储购物车中每个商品的信息对象
    // 每个商品的信息对象，都包含如下 6 个属性：
    // { goods_id, goods_name, goods_price, goods_count, goods_small_logo, goods_state }
    cart: [],
  }),

  // 模块的 mutations 方法
  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：

```
export default {
  // 为当前模块开启命名空间
  namespaced: true,

  // 模块的 state 数据
  state: () => ({
    // 购物车的数组，用来存储购物车中每个商品的信息对象
    // 每个商品的信息对象，都包含如下 6 个属性：
    // { goods_id, goods_name, goods_price, goods_count, goods_small_logo, goods_state }
    cart: [],
  }),

  // 模块的 mutations 方法
  mutations: {
    addToCart(state, goods) {
      // 根据提交的商品的Id，查询购物车中是否存在这件商品
      // 如果不存在，则 findResult 为 undefined；否则，为查找到的商品信息对象
      const findResult = state.cart.find((x) => x.goods_id === goods.goods_id)

      if (!findResult) {
        // 如果购物车中没有这件商品，则直接 push
        state.cart.push(goods)
      } else {
        // 如果购物车中有这件商品，则只更新数量即可
        findResult.goods_count++
      }
    },
  },

  // 模块的 getters 属性
  getters: {},
}
```

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：

```
// 按需导入 mapMutations 这个辅助方法
import { mapMutations } from 'vuex'

export default {
  methods: {
    // 把 m_cart 模块中的 addToCart 方法映射到当前页面使用
    ...mapMutations('m_cart', ['addToCart']),
  },
}
```

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

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

```

```
        // 2. 组织一个商品的信息对象
        const goods = {
            goods_id: this.goods_info.goods_id,       // 商品的Id
            goods_name: this.goods_info.goods_name,   // 商品的名称
            goods_price: this.goods_info.goods_price, // 商品的价格
            goods_count: 1,                           // 商品的数量
            goods_small_logo: this.goods_info.goods_small_logo, // 商品的图片
            goods_state: true                         // 商品的勾选状态
        }

        // 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
        this.addToCart(goods)

    }
}
```

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：

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

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：

```
// 按需导入 mapGetters 这个辅助方法
import { mapGetters } from 'vuex'

export default {
  computed: {
    // 把 m_cart 模块中名称为 total 的 getter 映射到当前页面中使用
    ...mapGetters('m_cart', ['total']),
  },
}
```

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**：

```
export default {
  watch: {
    // 1. 监听 total 值的变化，通过第一个形参得到变化后的新值
    total(newVal) {
      // 2. 通过数组的 find() 方法，找到购物车按钮的配置对象
      const findResult = this.options.find((x) => x.text === '购物车')

      if (findResult) {
        // 3. 动态为购物车按钮的 info 属性赋值
        findResult.info = newVal
```

```
        }
      },
    },
  }
```

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：

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

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：

```
addToCart(state, goods) {
  // 根据提交的商品的Id，查询购物车中是否存在这件商品
  // 如果不存在，则 findResult 为 undefined；否则，为查找到的商品信息对象
  const findResult = state.cart.find(x => x.goods_id === goods.goods_id)

  if (!findResult) {
    // 如果购物车中没有这件商品，则直接 push
    state.cart.push(goods)
  } else {
    // 如果购物车中有这件商品，则只更新数量即可
    findResult.goods_count++
  }

  // 通过 commit 方法，调用 m_cart 命名空间下的 saveToStorage 方法
  this.commit('m_cart/saveToStorage')
}
```

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：

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

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：

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

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：

```
watch: {
   // 定义 total 侦听器，指向一个配置对象
   total: {
      // handler 属性用来定义侦听器的 function 处理函数
      handler(newVal) {
         const findResult = this.options.find(x => x.text === '购物车')
         if (findResult) {
            findResult.info = newVal
         }
      },
      // 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
      immediate: true
   }
}
```

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：

```
// 按需导入 mapGetters 这个辅助方法
import { mapGetters } from 'vuex'

export default {
  data() {
    return {}
  },
  computed: {
    // 将 m_cart 模块中的 total 映射为当前页面的计算属性
    ...mapGetters('m_cart', ['total']),
  },
}
```

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：

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

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：

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

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：

```
import { mapGetters } from 'vuex'

// 导出一个 mixin 对象
export default {
  computed: {
    ...mapGetters('m_cart', ['total']),
  },
  onShow() {
    // 在页面刚展示的时候，设置数字徽标
    this.setBadge()
  },
  methods: {
    setBadge() {
      // 调用 uni.setTabBarBadge() 方法，为购物车设置右上角的徽标
      uni.setTabBarBadge({
        index: 2,
        text: this.total + '', // 注意：text 的值必须是字符串，不能是数字
      })
    },
  },
}
```

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：

```
// 导入自己封装的 mixin 模块
import badgeMix from '@/mixins/tabbar-badge.js'

export default {
  // 将 badgeMix 混入到当前的页面中进行使用
  mixins: [badgeMix],
  // 省略其它代码...
}
```