

4. 分类

4.0 创建 cate 分支

运行如下的命令，基于 master 分支在本地创建 cate 子分支，用来开发分类页面相关的功能：

```
1 git checkout -b cate
```

4.1 渲染分类页面的基本结构

1. 定义页面结构如下：

```
1 <template>
2   <view>
3     <view class="scroll-view-container">
4       <!-- 左侧的滚动视图区域 -->
5       <scroll-view class="left-scroll-view" scroll-y :style="{height: wh
+ 'px'}">
6         <view class="left-scroll-view-item active">xxx</view>
7         <view class="left-scroll-view-item">xxx</view>
8         <view class="left-scroll-view-item">xxx</view>
9         <view class="left-scroll-view-item">xxx</view>
10        <view class="left-scroll-view-item">xxx</view>
11        <view class="left-scroll-view-item">多复制一些节点，演示纵向滚动效
果...</view>
12      </scroll-view>
13      <!-- 右侧的滚动视图区域 -->
14      <scroll-view class="right-scroll-view" scroll-y :style="{height: wh
+ 'px'}">
15        <view class="left-scroll-view-item">zzz</view>
16        <view class="left-scroll-view-item">zzz</view>
17        <view class="left-scroll-view-item">zzz</view>
18        <view class="left-scroll-view-item">zzz</view>
19        <view class="left-scroll-view-item">多复制一些节点，演示纵向滚动效果
</view>
20      </scroll-view>
21    </view>
22  </view>
23 </template>
```

2. 动态计算窗口的剩余高度：

```
1 <script>
2   export default {
3     data() {
4       return {
5         // 窗口的可用高度 = 屏幕高度 - navigationBar高度 - tabBar 高度
6         wh: 0
7       };
8     },
```

```

9      onLoad() {
10         // 获取当前系统的信息
11         const sysInfo = uni.getSystemInfoSync()
12         // 为 wh 窗口可用高度动态赋值
13         this.wh = sysInfo.windowHeight
14     }
15 }
16 </script>

```

3. 美化页面结构:

```

1  .scroll-view-container {
2      display: flex;
3
4      .left-scroll-view {
5          width: 120px;
6
7          .left-scroll-view-item {
8              line-height: 60px;
9              background-color: #f7f7f7;
10             text-align: center;
11             font-size: 12px;
12
13             // 激活项的样式
14             &.active {
15                 background-color: #ffffff;
16                 position: relative;
17
18                 // 渲染激活项左侧的红色指示边线
19                 &::before {
20                     content: ' ';
21                     display: block;
22                     width: 3px;
23                     height: 30px;
24                     background-color: #c00000;
25                     position: absolute;
26                     left: 0;
27                     top: 50%;
28                     transform: translateY(-50%);
29                 }
30             }
31         }
32     }
33 }

```

4.2 获取分类数据

1. 在 data 中定义分类数据节点:

```

1  data() {
2      return {
3          // 分类数据列表
4          cateList: []
5      }
6  }

```

2. 调用获取分类列表数据的方法：

```
1  onLoad() {  
2    // 调用获取分类列表数据的方法  
3    this.getCateList()  
4  }
```

3. 定义获取分类列表数据的方法：

```
1  methods: {  
2    async getCateList() {  
3      // 发起请求  
4      const { data: res } = await  
uni.$http.get('/api/public/v1/categories')  
5      // 判断是否获取失败  
6      if (res.meta.status !== 200) return uni.$showMsg()  
7      // 转存数据  
8      this.cateList = res.message  
9    }  
10 }
```

4.3 动态渲染左侧的一级分类列表

1. 循环渲染列表结构：

```
1  <!-- 左侧的滚动视图区域 -->  
2  <scroll-view class="left-scroll-view" scroll-y :style="{height: wh +  
'px'}">  
3    <block v-for="(item, i) in cateList" :key="i">  
4      <view class="left-scroll-view-item">{{item.cat_name}}</view>  
5    </block>  
6  </scroll-view>
```

2. 在 data 中定义默认选中项的索引：

```
1  data() {  
2    return {  
3      // 当前选中项的索引，默认让第一项被选中  
4      active: 0  
5    }  
6  }
```

3. 循环渲染结构时，为选中项动态添加 `.active` 类名：

```
1  <block v-for="(item, i) in cateList" :key="i">  
2    <view :class="['left-scroll-view-item', i === active ? 'active' : '']">  
    {{item.cat_name}}</view>  
3  </block>
```

4. 为一级分类的 Item 项绑定点击事件处理函数 `activeChanged`：

```
1  <block v-for="(item, i) in cateList" :key="i">  
2    <view :class="['left-scroll-view-item', i === active ? 'active' : '']"  
    @click="activeChanged(i)">{{item.cat_name}}</view>  
3  </block>
```

5. 定义 `activeChanged` 事件处理函数，动态修改选中项的索引：

```
1  methods: {  
2    // 选中项改变的事件处理函数  
3    activeChanged(i) {  
4      this.active = i  
5    }  
6  }
```

4.4 动态渲染右侧的二级分类列表

1. 在 `data` 中定义二级分类列表的数据节点：

```
1  data() {  
2    return {  
3      // 二级分类列表  
4      cateLevel2: []  
5    }  
6  }
```

2. 修改 `getCateList` 方法，在请求到数据之后，为二级分类列表数据赋值：

```
1  async getCateList() {  
2    const { data: res } = await uni.$http.get('/api/public/v1/categories')  
3    if (res.meta.status !== 200) return uni.$showMsg()  
4    this.cateList = res.message  
5    // 为二级分类赋值  
6    this.cateLevel2 = res.message[0].children  
7  }
```

3. 修改 `activeChanged` 方法，在一级分类选中项改变之后，为二级分类列表数据重新赋值：

```
1  activeChanged(i) {  
2    this.active = i  
3    // 为二级分类列表重新赋值  
4    this.cateLevel2 = this.cateList[i].children  
5  }
```

4. 循环渲染右侧二级分类列表的 UI 结构：

```
1  <!-- 右侧的滚动视图区域 -->  
2  <scroll-view class="right-scroll-view" scroll-y :style="{height: wh +  
3    'px'}">  
4    <view class="cate-lv2" v-for="(item2, i2) in cateLevel2" :key="i2">  
5      <view class="cate-lv2-title"> {{item2.cat_name}} </view>  
6    </scroll-view>
```

5. 美化二级分类的标题样式：

```
1  .cate-lv2-title {  
2    font-size: 12px;  
3    font-weight: bold;  
4    text-align: center;  
5    padding: 15px 0;  
6  }
```

4.5 动态渲染右侧的三级分类列表

1. 在二级分类的 `<view>` 组件中，循环渲染三级分类的列表结构：

```
1      <!-- 右侧的滚动视图区域 -->
2      <scroll-view class="right-scroll-view" scroll-y :style="{height: wh +
'px'}">
3          <view class="cate-lv2" v-for="(item2, i2) in cateLevel2" :key="i2">
4              <view class="cate-lv2-title"> {{item2.cat_name}} </view>
5              <!-- 动态渲染三级分类的列表数据 -->
6              <view class="cate-lv3-list">
7                  <!-- 三级分类 Item 项 -->
8                  <view class="cate-lv3-item" v-for="(item3, i3) in item2.children"
:key="i3">
9                      <!-- 图片 -->
10                     <image :src="item3.cat_icon"></image>
11                     <!-- 文本 -->
12                     <text>{{item3.cat_name}}</text>
13                 </view>
14             </view>
15         </view>
16     </scroll-view>
```

2. 美化三级分类的样式：

```
1      .cate-lv3-list {
2          display: flex;
3          flex-wrap: wrap;
4
5          .cate-lv3-item {
6              width: 33.33%;
7              margin-bottom: 10px;
8              display: flex;
9              flex-direction: column;
10             align-items: center;
11
12             image {
13                 width: 60px;
14                 height: 60px;
15             }
16
17             text {
18                 font-size: 12px;
19             }
20         }
21     }
```

4.6 切换一级分类后重置滚动条的位置

1. 在 data 中定义 滚动条距离顶部的距离：

```

1  data() {
2    return {
3      // 滚动条距离顶部的距离
4      scrollTop: 0
5    }
6  }

```

2. 动态为右侧的 `<scroll-view>` 组件绑定 `scroll-top` 属性的值:

```

1  <!-- 右侧的滚动视图区域 -->
2  <scroll-view class="right-scroll-view" scroll-y :style="{height: wh +
'px'}" :scroll-top="scrollTop"></scroll-view>

```

3. 切换一级分类时, 动态设置 `scrollTop` 的值:

```

1  // 选中项改变的事件处理函数
2  activeChanged(i) {
3    this.active = i
4    this.cateLevel2 = this.cateList[i].children
5
6    // 让 scrollTop 的值在 0 与 1 之间切换
7    this.scrollTop = this.scrollTop === 0 ? 1 : 0
8
9    // 可以简化为如下的代码:
10   // this.scrollTop = this.scrollTop ? 0 : 1
11  }

```

4.7 点击三级分类跳转到商品列表页面

1. 为三级分类的 Item 项绑定点击事件处理函数如下:

```

1  <view class="cate-lv3-item" v-for="(item3, i3) in item2.children"
:key="i3" @click="gotoGoodsList(item3)">
2    <image :src="item3.cat_icon"></image>
3    <text>{{item3.cat_name}}</text>
4  </view>

```

2. 定义事件处理函数如下:

```

1  // 点击三级分类项跳转到商品列表页面
2  gotoGoodsList(item3) {
3    uni.navigateTo({
4      url: '/subpkg/goods_list/goods_list?cid=' + item3.cat_id
5    })
6  }

```

4.8 分支的合并与提交

1. 将 `cate` 分支进行本地提交:

```

1  git add .
2  git commit -m "完成了分类页面的开发"

```

2. 将本地的 `cate` 分支推送到码云:

```
1 git push -u origin cate
```

3. 将本地 `cate` 分支中的代码合并到 `master` 分支:

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

4. 删除本地的 `cate` 分支:

```
1 git branch -d cate
```

5. 搜索

5.0 创建 search 分支

运行如下的命令，基于 `master` 分支在本地创建 `search` 子分支，用来开发搜索相关的功能:

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

5.1 自定义搜索组件

5.1.1 自定义 my-search 组件

1. 在项目根目录的 `components` 目录下，鼠标右键，选择 `新建组件`，填写组件信息后，最后点击 `创建` 按钮:



2. 在分类页面的 UI 结构中，直接以标签的形式使用 `my-search` 自定义组件：

```
1 <!-- 使用自定义的搜索组件 -->
2 <my-search></my-search>
```

3. 定义 `my-search` 组件的 UI 结构如下：

```
1 <template>
2   <view class="my-search-container">
3     <!-- 使用 view 组件模拟 input 输入框的样式 -->
4     <view class="my-search-box">
5       <uni-icons type="search" size="17"></uni-icons>
6       <text class="placeholder">搜索</text>
7     </view>
8   </view>
9 </template>
```

注意：在当前组件中，我们使用 view 组件模拟 input 输入框的效果；并不会在页面上渲染真正的 input 输入框

4. 美化自定义 search 组件的样式：

```
1 .my-search-container {
2   background-color: #c00000;
3   height: 50px;
4   padding: 0 10px;
5   display: flex;
6   align-items: center;
7 }
8
9 .my-search-box {
10  height: 36px;
11  background-color: #ffffff;
12  border-radius: 15px;
13  width: 100%;
14  display: flex;
15  align-items: center;
16  justify-content: center;
17
18  .placeholder {
19    font-size: 15px;
20    margin-left: 5px;
21  }
22 }
```

5. 由于自定义的 `my-search` 组件高度为 `50px`，因此，需要重新计算分类页面窗口的可用高度：

```
1 onLoad() {
2   const sysInfo = uni.getSystemInfoSync()
3   // 可用高度 = 屏幕高度 - navigationBar高度 - tabBar高度 - 自定义的search组件高度
4   this.wh = sysInfo.windowHeight - 50
5 }
```

5.1.2 通过自定义属性增强组件的通用性

为了增强组件的通用性，我们允许使用者自定义搜索组件的 背景颜色 和 圆角尺寸 。

1. 通过 `props` 定义 `bgcolor` 和 `radius` 两个属性，并指定值类型和属性默认值：

```
1  props: {  
2    // 背景颜色  
3    bgcolor: {  
4      type: String,  
5      default: '#C00000'  
6    },  
7    // 圆角尺寸  
8    radius: {  
9      type: Number,  
10     // 单位是 px  
11     default: 18  
12   }  
13 }
```

2. 通过属性绑定的形式，为 `.my-search-container` 盒子和 `.my-search-box` 盒子动态绑定 `style` 属性：

```
1  <view class="my-search-container" :style="{ 'background-color': bgcolor}">  
2    <view class="my-search-box" :style="{ 'border-radius': radius + 'px'}">  
3      <uni-icons type="search" size="17"></uni-icons>  
4      <text class="placeholder">搜索</text>  
5    </view>  
6  </view>
```

3. 移除对应 `scss` 样式中的 背景颜色 和 圆角尺寸 ：

```
1  .my-search-container {  
2    // 移除背景颜色，改由 props 属性控制  
3    // background-color: #C00000;  
4    height: 50px;  
5    padding: 0 10px;  
6    display: flex;  
7    align-items: center;  
8  }  
9  
10 .my-search-box {  
11   height: 36px;  
12   background-color: #ffffff;  
13   // 移除圆角尺寸，改由 props 属性控制  
14   // border-radius: 15px;  
15   width: 100%;  
16   display: flex;  
17   align-items: center;  
18   justify-content: center;  
19  
20   .placeholder {  
21     font-size: 15px;  
22     margin-left: 5px;  
23   }  
24 }
```

5.1.3 为自定义组件封装 click 事件

1. 在 `my-search` 自定义组件内部，给类名为 `.my-search-box` 的 `view` 绑定 `click` 事件处理函数：

```
1 <view class="my-search-box" :style="{ 'border-radius': radius + 'px' }"
  @click="searchBoxHandler">
2   <uni-icons type="search" size="17"></uni-icons>
3   <text class="placeholder">搜索</text>
4 </view>
```

2. 在 `my-search` 自定义组件的 `methods` 节点中，声明事件处理函数如下：

```
1 methods: {
2   // 点击了模拟的 input 输入框
3   searchBoxHandler() {
4     // 触发外界通过 @click 绑定的 click 事件处理函数
5     this.$emit('click')
6   }
7 }
```

3. 在分类页面中使用 `my-search` 自定义组件时，即可通过 `@click` 为其绑定点击事件处理函数：

```
1 <!-- 使用自定义的搜索组件 -->
2 <my-search @click="gotoSearch"></my-search>
```

同时在分类页面中，定义 `gotoSearch` 事件处理函数如下：

```
1 methods: {
2   // 跳转到分包中的搜索页面
3   gotoSearch() {
4     uni.navigateTo({
5       url: '/subpkg/search/search'
6     })
7   }
8 }
```

5.1.4 实现首页搜索组件的吸顶效果

1. 在 home 首页定义如下的 UI 结构：

```
1 <!-- 使用自定义的搜索组件 -->
2 <view class="search-box">
3   <my-search @click="gotoSearch"></my-search>
4 </view>
```

2. 在 home 首页定义如下的事件处理函数：

```
1 gotoSearch() {
2   uni.navigateTo({
3     url: '/subpkg/search/search'
4   })
5 }
```

3. 通过如下的样式实现吸顶的效果：

```

1  .search-box {
2    // 设置定位效果为“吸顶”
3    position: sticky;
4    // 吸顶的“位置”
5    top: 0;
6    // 提高层级，防止被轮播图覆盖
7    z-index: 999;
8  }

```

5.2 搜索建议

5.2.1 渲染搜索页面的基本结构

1. 定义如下的 UI 结构：

```

1  <view class="search-box">
2    <!-- 使用 uni-ui 提供的搜索组件 -->
3    <uni-search-bar @input="input" :radius="100" cancelButton="none"></uni-
search-bar>
4  </view>

```

2. 修改 `components -> uni-search-bar -> uni-search-bar.vue` 组件，将默认的白色搜索背景改为 `#C00000` 的红色背景：

```

1  .uni-searchbar {
2    /* #ifndef APP-NVUE */
3    display: flex;
4    /* #endif */
5    flex-direction: row;
6    position: relative;
7    padding: 16rpx;
8    /* 将默认的 #FFFFFF 改为 #C00000 */
9    background-color: #c00000;
10 }

```

3. 实现搜索框的吸顶效果：

```

1  .search-box {
2    position: sticky;
3    top: 0;
4    z-index: 999;
5  }

```

4. 定义如下的 input 事件处理函数：

```

1  methods: {
2    input(e) {
3      // e.value 是最新的搜索内容
4      console.log(e.value)
5    }
6  }

```

5.2.2 实现搜索框自动获取焦点

1. 修改 `components -> uni-search-bar -> uni-search-bar.vue` 组件，把 `data` 数据中的 `show` 和 `showSync` 的值，从默认的 `false` 改为 `true` 即可：

```
1  data() {
2    return {
3      show: true,
4      showSync: true,
5      searchVal: ""
6    }
7  }
```

2. 使用手机扫码预览，即可在真机上查看效果。

5.2.3 实现搜索框的防抖处理

1. 在 `data` 中定义防抖的延时器 `timerId` 如下：

```
1  data() {
2    return {
3      // 延时器的 timerId
4      timer: null,
5      // 搜索关键词
6      kw: ''
7    }
8  }
```

2. 修改 `input` 事件处理函数如下：

```
1  input(e) {
2    // 清除 timer 对应的延时器
3    clearTimeout(this.timer)
4    // 重新启动一个延时器，并把 timerId 赋值给 this.timer
5    this.timer = setTimeout(() => {
6      // 如果 500 毫秒内，没有触发新的输入事件，则为搜索关键词赋值
7      this.kw = e.value
8      console.log(this.kw)
9    }, 500)
10 }
```

5.2.4 根据关键词查询搜索建议列表

1. 在 `data` 中定义如下的数据节点，用来存放搜索建议的列表数据：

```
1  data() {
2    return {
3      // 搜索结果列表
4      searchResults: []
5    }
6  }
```

2. 在防抖的 `setTimeout` 中，调用 `getSearchList` 方法获取搜索建议列表：

```
1  this.timer = setTimeout(() => {
2    this.kw = e.value
3    // 根据关键词，查询搜索建议列表
4    this.getSearchList()
5  }, 500)
```

3. 在 `methods` 中定义 `getSearchList` 方法如下:

```
1 // 根据搜索关键词，搜索商品建议列表
2 async getSearchList() {
3   // 判断关键词是否为空
4   if (this.kw === '') {
5     this.searchResults = []
6     return
7   }
8   // 发起请求，获取搜索建议列表
9   const { data: res } = await
uni.$http.get('/api/public/v1/goods/qsearch', { query: this.kw })
10   if (res.meta.status !== 200) return uni.$showMsg()
11   this.searchResults = res.message
12 }
```

5.2.5 渲染搜索建议列表

1. 定义如下的 UI 结构:

```
1 <!-- 搜索建议列表 -->
2 <view class="sugg-list">
3   <view class="sugg-item" v-for="(item, i) in searchResults" :key="i"
@click="gotoDetail(item.goods_id)">
4     <view class="goods-name">{{item.goods_name}}</view>
5     <uni-icons type="arrowright" size="16"></uni-icons>
6   </view>
7 </view>
```

2. 美化搜索建议列表:

```
1 .sugg-list {
2   padding: 0 5px;
3
4   .sugg-item {
5     font-size: 12px;
6     padding: 13px 0;
7     border-bottom: 1px solid #efefef;
8     display: flex;
9     align-items: center;
10    justify-content: space-between;
11
12    .goods-name {
13      // 文字不允许换行（单行文本）
14      white-space: nowrap;
15      // 溢出部分隐藏
16      overflow: hidden;
17      // 文本溢出后，使用 ... 代替
18      text-overflow: ellipsis;
19      margin-right: 3px;
20    }
21  }
22 }
```

3. 点击搜索建议的 Item 项，跳转到商品详情页面:

```

1 gotoDetail(goods_id) {
2   uni.navigateTo({
3     // 指定详情页面的 URL 地址，并传递 goods_id 参数
4     url: '/subpkg/goods_detail/goods_detail?goods_id=' + goods_id
5   })
6 }

```

5.3 搜索历史

5.3.1 渲染搜索历史记录的基本结构

1. 在 data 中定义搜索历史的 假数据：

```

1 data() {
2   return {
3     // 搜索关键词的历史记录
4     historyList: ['a', 'app', 'apple']
5   }
6 }

```

2. 渲染搜索历史区域的 UI 结构：

```

1 <!-- 搜索历史 -->
2 <view class="history-box">
3   <!-- 标题区域 -->
4   <view class="history-title">
5     <text>搜索历史</text>
6     <uni-icons type="trash" size="17"></uni-icons>
7   </view>
8   <!-- 列表区域 -->
9   <view class="history-list">
10    <uni-tag :text="item" v-for="(item, i) in historyList" :key="i">
11  </uni-tag>
12 </view>

```

3. 美化搜索历史区域的样式：

```

1 .history-box {
2   padding: 0 5px;
3
4   .history-title {
5     display: flex;
6     justify-content: space-between;
7     align-items: center;
8     height: 40px;
9     font-size: 13px;
10    border-bottom: 1px solid #efefef;
11  }
12
13  .history-list {
14    display: flex;
15    flex-wrap: wrap;
16
17    .uni-tag {

```

```

18         margin-top: 5px;
19         margin-right: 5px;
20     }
21 }
22 }

```

5.3.2 实现搜索建议和搜索历史的按需展示

1. 当搜索结果列表的长度 **不为 0** 的时候 (`searchResults.length !== 0`) , 需要展示搜索建议区域, 隐藏搜索历史区域
2. 当搜索结果列表的长度 **等于 0** 的时候 (`searchResults.length === 0`) , 需要隐藏搜索建议区域, 展示搜索历史区域
3. 使用 `v-if` 和 `v-else` 控制这两个区域的显示和隐藏, 示例代码如下:

```

1  <!-- 搜索建议列表 -->
2  <view class="sugg-list" v-if="searchResults.length !== 0">
3      <!-- 省略其它代码... -->
4  </view>
5
6  <!-- 搜索历史 -->
7  <view class="history-box" v-else>
8      <!-- 省略其它代码... -->
9  </view>

```

5.3.3 将搜索关键词存入 historyList

1. 直接将搜索关键词 `push` 到 `historyList` 数组中即可

```

1  methods: {
2      // 根据搜索关键词, 搜索商品建议列表
3      async getSearchList() {
4          // 省略其它不必要的代码...
5
6          // 1. 查询到搜索建议之后, 调用 saveSearchHistory() 方法保存搜索关键词
7          this.saveSearchHistory()
8      },
9      // 2. 保存搜索关键词的方法
10     saveSearchHistory() {
11         // 2.1 直接把搜索关键词 push 到 historyList 数组中
12         this.historyList.push(this.kw)
13     }
14 }

```

2. 上述实现思路存在的问题:
 - 关键词**前后顺序**的问题 (可以调用数组的 `reverse()` 方法 对数组进行反转)
 - 关键词**重复**的问题 (可以使用 `Set 对象` 进行去重操作)

5.3.4 解决关键字前后顺序的问题

1. data 中的 `historyList` 不做任何修改, 依然使用 `push` 进行**末尾追加**
2. 定义一个计算属性 `historys` , 将 `historyList` 数组 `reverse` 反转之后, 就是此计算属性的值:

```

1   computed: {
2     historys() {
3       // 注意：由于数组是引用类型，所以不要直接基于原数组调用 reverse 方法，以免修改原数
      组中元素的顺序
4       // 而是应该新建一个内存无关的数组，再进行 reverse 反转
5       return [...this.historyList].reverse()
6     }
7   }

```

3. 页面中渲染搜索关键词的时候，不再使用 data 中的 `historyList`，而是使用计算属性 `historys`：

```

1   <view class="history-list">
2     <uni-tag :text="item" v-for="(item, i) in historys" :key="i"></uni-tag>
3 </view>

```

5.3.5 解决关键词重复的问题

1. 修改 `saveSearchHistory` 方法如下：

```

1   // 保存搜索关键词为历史记录
2   saveSearchHistory() {
3     // this.historyList.push(this.kw)
4
5     // 1. 将 Array 数组转化为 Set 对象
6     const set = new Set(this.historyList)
7     // 2. 调用 Set 对象的 delete 方法，移除对应的元素
8     set.delete(this.kw)
9     // 3. 调用 Set 对象的 add 方法，向 Set 中添加元素
10    set.add(this.kw)
11    // 4. 将 Set 对象转化为 Array 数组
12    this.historyList = Array.from(set)
13  }

```

5.3.6 将搜索历史记录持久化存储到本地

1. 修改 `saveSearchHistory` 方法如下：

```

1   // 保存搜索关键词为历史记录
2   saveSearchHistory() {
3     const set = new Set(this.historyList)
4     set.delete(this.kw)
5     set.add(this.kw)
6     this.historyList = Array.from(set)
7     // 调用 uni.setStorageSync(key, value) 将搜索历史记录持久化存储到本地
8     uni.setStorageSync('kw', JSON.stringify(this.historyList))
9   }

```

2. 在 `onLoad` 生命周期函数中，加载本地存储的搜索历史记录：

```

1   onLoad() {
2     this.historyList = JSON.parse(uni.getStorageSync('kw') || '[]')
3   }

```

5.3.7 清空搜索历史记录

1. 为清空的图标按钮绑定 `click` 事件:

```
1 <uni-icons type="trash" size="17" @click="cleanHistory"></uni-icons>
```

2. 在 `methods` 中定义 `cleanHistory` 处理函数:

```
1 // 清空搜索历史记录
2 cleanHistory() {
3   // 清空 data 中保存的搜索历史
4   this.historyList = []
5   // 清空本地存储中记录的搜索历史
6   uni.setStorageSync('kw', '[]')
7 }
```

5.3.8 点击搜索历史跳转到商品列表页面

1. 为搜索历史的 Item 项绑定 `click` 事件处理函数:

```
1 <uni-tag :text="item" v-for="(item, i) in historys" :key="i"
  @click="gotoGoodsList(item)"></uni-tag>
```

2. 在 `methods` 中定义 `gotoGoodsList` 处理函数:

```
1 // 点击跳转到商品列表页面
2 gotoGoodsList(kw) {
3   uni.navigateTo({
4     url: '/subpkg/goods_list/goods_list?query=' + kw
5   })
6 }
```

5.4 分支的合并与提交

1. 将 `search` 分支进行本地提交:

```
1 git add .
2 git commit -m "完成了搜索功能的开发"
```

2. 将本地的 `search` 分支推送到码云:

```
1 git push -u origin search
```

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

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

4. 删除本地的 `search` 分支:

```
1 git branch -d search
```