第5章

CHAPTER 5

滑动视图

在Flutter中，当内容超过显示视图时，如果没有特殊处理，Flutter则会提示Overflow错误，一般当内容体超过显示视图时，采用滑动（滚动视图）来处理。

在Flutter中通过ScrollView组件实现滑动视图效果，当ScrollView的内容大于它本身的大小时，ScrollView会自动添加滚动条，并可以竖直滑动，如Android中的ScrollView、iOS中的UIScrollView。

在Flutter中常用的滚动视图有SingleChildScrollView、NestedScrollView、CustomScrollView、Scrollable、ListView、GridView、TabBarView、PageView。

5.1 长页面滑动视图

Flutter中SingleChildScrollView、NestedScrollView、CustomScrollView用来处理长页面滑动效果。

默认情况下滑动视图可在竖直方向上下滑动，在Android平台中只是一个滑动，当滑动到边缘时会有浅水波纹拉伸效果，在iOS平台中，有回弹效果，可通过physics属性修改为指定的效果，physics属性的取值如表5-1所示。

表5-1 滑动回弹效果概述

类　别	描　述
BouncingScrollPhysics	可滑动，当滑动到边界时有回弹效果，iOS平台默认使用
ClampingScrollPhysics	可滑动，当滑动到边界时有水波回弹效果，Android平台默认使用
AlwaysScrollableScrollPhysics	列表总是可滚动的，在iOS上会有回弹效果，在Android上不会回弹
PageScrollPhysics	一般用于PageView的滑动效果，如果将ListView设置为滑动到末尾，则会有个比较大的弹起和回弹
FixedExtentScrollPhysics	一般用于ListWheelScrollViews
NeverScrollableScrollPhysics	不可滑动

scrollDirection属性用来设置滚动方向（滑动组件的通用配置），默认为垂直，也就是在

竖直方向上下滚动，可取值包括 Axis. vertical(竖直方向)、Axis. horizontal(水平方向)。

5.1.1 滑动组件 SingleChildScrollView

SingleChildScrollView 适用于简单滑动视图处理，如 App 中常见的商品详情页面、订单详情页面(笔者建议在实际应用开发中常用的详情页面使用这个组件)，代码如下：

```
//代码清单 5-1-1 SingleChildScrollView 的基本使用
//lib/code5/example_501_SingleChildScrollView.dart
class Example501 extends StatefulWidget {
  const Example501({Key? key}) : super(key: key);
  @override
  State<StatefulWidget> createState() {
    return _ExampleState();
  }
}

class _ExampleState extends State<Example501> {
  //滑动控制器
  final ScrollController _scrollController = ScrollController();
  //文本的关键 Key,用于获取文本的位置信息
  final GlobalKey _globalKey = GlobalKey();

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("滑动")),
      body: SingleChildScrollView(
        //设置内边距
        padding: const EdgeInsets.all(20),
        physics: const BouncingScrollPhysics(),
        controller: _scrollController,  //配置滑动控制器
        //子 Widget,通常是 UI 布局系列的 Column、Stack、Row
        child: Container(
          alignment: Alignment.center,
          color: Colors.grey,
          height: 1600,
          child: Text("测试数据",key: _globalKey,),
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          //scrollToTop();
          scrollToWidgetPostion(_globalKey);
        },
```

```
          child: Icon(Icons.arrow_upward),
        ),
      );
    }

    …

  }
```

ScrollController是滑动视图使用的控制器，在ScrollController中可添加监听，实时获取滚动的距离，代码如下：

```
//代码清单 5-1-2 ScrollController 的监听
//lib/code5/example_501_SingleChildScrollView.dart
@override
void initState() {
  super.initState();
  //添加滚动监听
  _scrollController.addListener(() {
    //滚动时会实时回调这里
    //获取滚动的距离
    double offsetValue = _scrollController.offset;
    //ScrollView 最大可滑动的距离
    double max = _scrollController.position.maxScrollExtent;
    if (offsetValue <= 0) {
      //如果有回弹效果，则 offsetValue 会出现负值
      print("滚动到了顶部");
    } else if (offsetValue >= max) {
      //如果有回弹效果，则 offsetValue 的值可能大于 max
      print("滚动到了底部");
    } else {
      print("滑动的距离 offsetValue $offsetValue max $max");
    }
  });
}
```

可通过ScrollController的animateTo()方法滑动到指定位置，如这里滑动到视图中的文本位置，通过绑定的GlobalKey获取对应文本的位置，代码如下：

```
//代码清单 5-1-3 ScrollController 的监听
//lib/code5/example_501_SingleChildScrollView.dart
//通过 Widget 绑定的 GlobalKey 获取位置信息
void scrollToWidgetPostion(GlobalKey key) {
  //根据 key 获取上下文对象，即获取 Element 信息
```

```
  BuildContext ? stackContext = key.currentContext;
  if (stackContext != null) {
    //获取对应的 RenderObj 对象
    RenderObject? renderObject = stackContext.findRenderObject();
    if (renderObject != null) {
      //获取指定的 Widget 的位置信息
      Size size = renderObject.paintBounds.size;
      //获取矩阵
      Matrix4 matrix4 = renderObject.getTransformTo(null);
      var translation = matrix4.getTranslation();
      //距离手机屏幕顶部的距离
      double? top = translation.y;
      //获取状态栏高度
      final double statusBarHeight = MediaQuery.of(context).padding.top;
      //获取 AppBar 的高度
      final double appBarHeight = 76;
      //滑动到这个 Widget 的位置
      double tagHeight = top - statusBarHeight - appBarHeight;
scrollOffset(tagHeight);
    }
  }
}
```

scrollOffset 对应的滑动方法如下：

```
//代码清单 5-1-4
//lib/code5/example_501_SingleChildScrollView.dart
//滚动到指定的位置
void scrollOffset(double offset) {
  //返回顶部指定位置
  _scrollController.animateTo(
    offset,
    //返回顶部的过程中执行一个滚动动画,动画持续时间是 200ms
    duration: const Duration(milliseconds: 200),
    //动画曲线是 Curves.ease
    curve: Curves.ease,
  );
}
```

Demo 运行调试图如图 5-1 所示，SingleChildScrollView 的顶部是状态栏与标题栏的高度，此高度为 79 像素，SingleChildScrollView 中的子组件 TextView 距离滑动组件的顶部为 790 像素，在代码中通过 GlobalKey 获取的 top 值为 866＝76＋790。

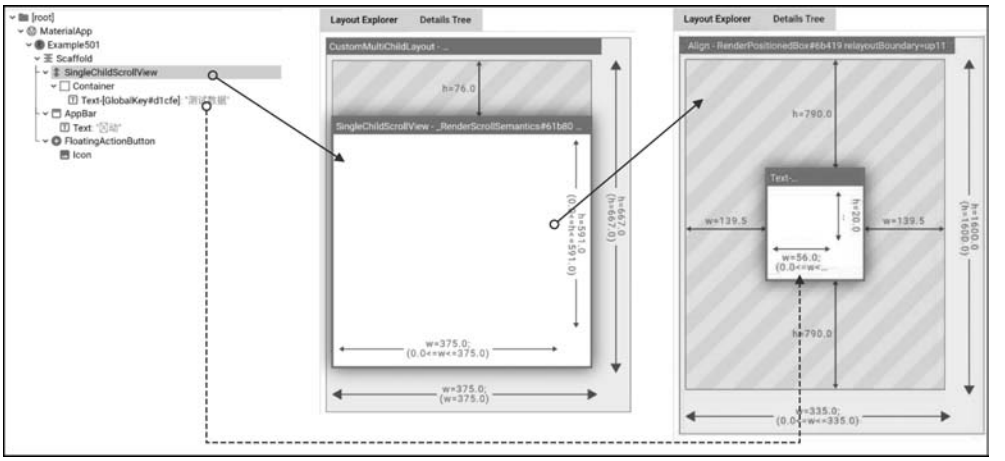

图 5-1　滑动视图构建效果图

5.1.2　滑动布局 NestedScrollView 与 SliverAppBar

NestedScrollView 继承于 CustomScrollView，它比 SingleChildScrollView 更强大，可以用来实现诸如滑动折叠头部的功能，如图 5-2 所示。读者可观看视频讲解【5.1.2 滑动布局 NestedScrollView 与 SliverAppBar】。

图 5-2　滑动折叠页面显示效果图

页面主视图通过 NestedScrollView 来搭建,代码如下:

```
//代码清单 5-2-1 NestedScrollView 的基本使用
//lib/code/code5/example_503_NestedScrollView.dart
class Example503 extends StatefulWidget {
  @override
  State<StatefulWidget> createState() {
    return _ExampleState();
  }
}
class _ExampleState extends State<Example503>
    with SingleTickerProviderStateMixin {

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: NestedScrollView(
        //配置可折叠的头布局
headerSliverBuilder: (BuildContext context, bool innerBoxIsScrolled) {
          //回调参数 innerBoxIsScrolled,当折叠头部隐藏时为 true
          //当折叠头部显示时为 false
          print("innerBoxIsScrolled $ innerBoxIsScrolled");
          return [buildSliverAppBar()];
        },
        //超出显示内容区域的 Widget 可以是一个列表、一个滑动视图
        //也可以是一个 TabBarView 来结合 TabBar 使用
        body: buildBodyWidget(),
      ),
    );
  } //Sl
…
}
```

在构建函数 headerSliverBuilder 中可使用 Sliver 家族的组件,在这里使用 SliverAppBar 组件,SliverAppBar 与 AppBar 类似,代码如下:

```
//代码清单 5-2-2 SliverAppBar 的详细配置
//lib/code/code5/example_503_NestedScrollView.dart
buildSliverAppBar() {
  return SliverAppBar(
    backgroundColor: Colors.white,
    title: const Text("这里是标题",style: TextStyle(color: Colors.blue),),
    //标题居中
    centerTitle: true,
    //当此值为 true 时 SliverAppBar title 会固定在页面顶部
```

```
      //当此值为 false 时 SliverAppBar title 会随着滑动向上滑动
      pinned: true,

      //当 pinned 属性值为 true 时才会起作用
      //当 floating 为 true 且滑动到顶部时 title 会隐藏
      //当为 false 时 title 不会隐藏
      floating: false,

      //当 snap 配置为 true 时,向下滑动页面,SliverAppBar(
      //及其中配置的 flexibleSpace 内容)会立即显示出来

      //反之当 snap 配置为 false 且向下滑动时,
      //只有当 ListView 的数据滑动到顶部时,SliverAppBar 才会被下拉显示出来
      snap: false,
      //展开的高度
      expandedHeight: 200,
      //AppBar 下的内容区域
      flexibleSpace: FlexibleSpaceBar(
        //背景
        //在这里直接使用的是一张图片
        background: Image.asset(
          "assets/images/banner_icon.jpg",
          height: 200,
          width: MediaQuery.of(context).size.width,
          fit: BoxFit.fill,
        ),
      ),
      bottom: TabBar(
        labelColor: Colors.blue,
        unselectedLabelColor: Colors.grey,
        controller: tabController,
        tabs: const <Widget>[
          Tab(text: "标签一"), Tab(text: "标签二"), Tab(text: "标签三"),
        ],
      ),
    );
  }
```

页面的主体部分通过 Scaffold 的 body 属性来配置,可以通过 TabBarView 结合上述 SliverAppBar 中的 bottom 属性配置的 TabBar 实现标签页面的切换,代码如下:

```
  //因为页面主体使用了 TabBar,所以用到了控制器
  late TabController tabController;
  @override
  void initState() {
```

```
    super.initState();
    //初始化控制器
    tabController = TabController(length: 3, vsync: this);
  }
  buildBodyWidget() {
    return TabBarView(
      controller: tabController,
      children: [
        ItemPage(),
        ItemPage(),
        ItemPage(),
      ],
    );
  }
```

5.1.3 滑动组件 CustomScrollView

当一个页面中，既有九宫格布局 GridView，又有列表 ListView，由于二者有各自的滑动区域，所以不能进行统一滑动，可通过 CustomScrollView 将二者结合起来，也就是可将 CustomScrollView 理解为滑动容器。

在 CustomScrollView 中需要结合 Sliver 家族的组件来使用，包括 SliverToBoxAdapter、SliverPersistentHeader、SliverFixedExtentList、SliverList、SliverGrid、SliverAppBar、SliverPadding 等。

SliverAppBar 用来处理折叠标题，在 CustomScrollView 中配置也可实现 5.1.2 节中的折叠视图效果，SliverToBoxAdapter 用于实现非 Sliver 家族的组件也可在 CustomScrollView 中使用，宫格布局 SliverGrid 用来实现二维滑动视图，代码如下：

```
//代码清单 5-3-1 九宫格通过构造函数来创建
//lib/code5/example_505_CustomScrollView.dart
class Example505 extends StatefulWidget {

  …

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("CustomScrollView"),
      ),
      body: CustomScrollView(
        //滑动控制器
        controller: _scrollController,
        slivers: [buildSliverGrid()],
```

```
        ),
      );
    }

    SliverGrid buildSliverGrid() {
      //使用构建方法来构建
      return SliverGrid(
        //用来配置每个子 Item 之间的关系
        gridDelegate: const SliverGridDelegateWithFixedCrossAxisCount(
          //Grid 按 2 列显示,也就是列数
          crossAxisCount: 2,
          //主方向每个 Item 之间的间隔
          mainAxisSpacing: 10.0,
          //次方向每个 Item 之间的间隔
          crossAxisSpacing: 10.0,
          //Item 的宽与高的比例
          childAspectRatio: 3.0,
        ),

        //用来配置每个子 Item 的具体构建
        delegate: SliverChildBuilderDelegate(
          //构建每个 Item 的具体显示 UI
          (BuildContext context, int index) {
            //创建子 Widget
            return Container(
              alignment: Alignment.center,
              //根据角标来动态计算生成不同的背景颜色
              color: Colors.cyan[100 * (index % 9)],
              child: new Text('grid item $ index'),
            );
          }, //Grid 的个数
          childCount: 20,
        ),
      );
    }
  }
```

在上述代码中使用了 SliverGridDelegateWithFixedCrossAxisCount,这个 delegate 用来根据指定的每行显示多少列 Item,而依次换行显示,不同屏幕分辨率下的手机显示的列数是一样的。

还可以使用 SliverGridDelegateWithMaxCrossAxisExtent,这个 delegate 根据每个 Item 允许的最大宽度依次排列每个 Item,也就是不同屏幕分辨率下的手机显示的列数不一样,代码如下:

```
//代码清单 5-3-2 九宫格通过构造函数来创建
//lib/code5/example_505_CustomScrollView.dart
SliverGrid buildSliverGrid2() {
  //使用构建方法来构建
  return SliverGrid(
    //用来配置每个子 Item 之间的关系
    gridDelegate: const SliverGridDelegateWithMaxCrossAxisExtent(
      //主方向每个 Item 之间的间隔
      mainAxisSpacing: 10.0,
      //次方向每个 Item 之间的间隔
      crossAxisSpacing: 10.0,
      //Item 的宽与高的比例
      childAspectRatio: 3.0,
      //每个 Item 的最大宽度
      maxCrossAxisExtent: 200,
    ),
    delegate: SliverChildListDelegate([
      Container(
        color: Colors.redAccent,
        child: new Text('grid item'),
      ),
      Container(
        color: Colors.black,
        child: new Text('grid item'),
      )
    ]),
  );
}
```

在上述代码中分别使用了 SliverChildListDelegate 和 SliverChildBuilderDelegate，这两个 delegate 的区别如下：

(1) SliverChildListDelegate 用来构建少量的 Item 应用场景，在使用这个 delegate 时，会将用到的 Item 一次性构建出来。

(2) SliverChildBuilderDelegate 用来构建大量的 Item 应用场景，这个 delegate 不会将超出屏幕以外未显示的 Item 构建出来，会在滑动到时且需要显示时才会去构建。

通过 SliverGrid.count()来创建，会指定一行展示多少个 Item，实现的效果与使用 SliverGrid 构造函数中 SliverGridDelegateWithFixedCrossAxisCount 的 delegate 创建的效果一致。

通过 SliverGrid.extent()来创建，实现的效果与使用 SliverGrid 构造函数中 SliverGridDelegateWithMaxCrossAxisExtent 的 delegate 创建的效果一致，会指定每个 Item 允许展示的最大宽度来依次排列 Item。

SliverList 只有一个 delegate 属性，可以用 SliverChildListDelegate 或 SliverChildBuilderDelegate

这两个类实现,在上述也描述及对比过这两个 delegate 的区别,前者将会一次性全部渲染子组件,后者将会根据视窗渲染当前出现的元素,在实际开发中,SliverChildBuilderDelegate 使用得比较多,代码如下:

```
//代码清单 5-3-3 SliverList 列表
//lib/code5/example_505_CustomScrollView.dart
Widget buildSliverList() {
  return SliverList(
    delegate: SliverChildBuilderDelegate(
      //构建每个 Item 的具体显示 UI
      (BuildContext context, int index) {
        //创建子 Widget
        return new Container(
          height: 44,
          alignment: Alignment.center,
          //根据角标来动态计算生成不同的背景颜色
          color: Colors.cyan[100 * (index % 9)],
          child: new Text('grid item $index'),
        );
      },
      //列表的条目个数
      childCount: 100,
    ),
  );
}
```

5.2 列表数据展示

ListView 是最常用的可滚动列表,GridView 用来构建二维网格列表,PageView 可用于 Widget 的整屏滑动切换。

5.2.1 ListView 用来构建常用的列表数据页面

ListView 有 4 种创建方式,描述如下:

(1) 默认构造函数(传入 List children)。

(2) 通过 ListView.builder 方式来创建,适用于有大量数据的情况。

(3) 通过 ListView.custom 方式来创建,提供了自定义子 Widget 的能力。

(4) 通过 ListView.separated 方式来创建,可以配置分割线,适用于具有固定数量列表项的 ListView。

通过 ListView 的构造函数来创建,适用于构建少量数据时的场景,如图 5-3 所示,代码如下:

```
ListView(
  //子 Item
  children: const [
    Text("测试数据 1"),
    Text("测试数据 2"),
    Text("测试数据 3"),
    Text("测试数据 4"),
  ],
)
```

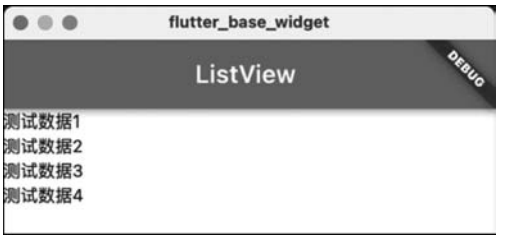

图 5-3 ListView 效果图

在实际项目业务开发中，以 ListView.builder 方式来创建使用得比较多，通常称为懒加载模式，适合列表项比较多的情况，因为只有当子组件真正显示时才会被创建，代码如下：

```
//代码清单 5-4-1 ListView 通过 builder 来构建
//lib/code5/example_510_ListView.dart
Widget buildListView1() {
  return ListView.builder(
    //列表子 Item 的个数
    itemCount: 10000,
    //每列表子 Item 的高度
    itemExtent: 100,
    //构建每个 ListView 的 Item
    itemBuilder: (BuildContext context, int index) {
      //子 Item 可单独封装成一个 StatefulWidget
      //也可以是一个 Widget
      return buildListViewItemWidget(index);
    },
  );
}
```

对于在 ListView 中构建的子 Item 的页面 UI，在实际很多应用场景中，代码量比较大，也要根据数据的不同来展示不同的效果，建议将子 Item 的构建放到单独的 StatelessWidget 或者 StatefulWidget 中，本节中显示的效果简单，所以构建了一个普通的 Widget，代码如下：

```
//代码清单 5-4-2 创建 ListView 使用的子布局
//lib/code5/example_510_ListView.dart
Widget buildListViewItemWidget(int index) {
  return Container(
    height: 84,//列表子 Item 的高度
    alignment: Alignment.center,  //内容居中
    //根据索引来动态计算生成不同的背景颜色
    color: Colors.cyan[100 * (index % 9)],
    child: Text('grid item $ index'),
  );
}
```

ListView.separated 可以在生成的列表项之间添加一个分割组件，它比 ListView.builder 多了一个 separatorBuilder 参数，该参数是一个分割组件生成器，常用于列表 Item 之间有分隔线的场景，代码如下：

```
//代码清单 5-4-3 通过 separated 来构建
//lib/code5/example_510_ListView.dart
Widget buildListView2() {
  return ListView.separated(
    //列表子 Item 的个数
    itemCount: 10000,
    //构建每个 ListView 的 Item
    itemBuilder: (BuildContext context, int index) {
      //ListView 的子 Item
      return buildListViewItemWidget(index);
    },
    //构建每个子 Item 之间的间隔 Widget
    separatorBuilder: (BuildContext context, int index) {
      //这里构建的是不同颜色的分隔线
      return Container(
        height: 4,
        //根据索引来动态计算生成不同的背景颜色
        color: Colors.cyan[100 * (index % 9)],
      );
    },
  );
}
```

ListView 的 custom()方法使用参数 childrenDelegate 来配置一个 SliverChildDelegate 代理构建子 Item，SliverChildDelegate 是抽象的，不可直接使用，一般在实际项目开发中使用它的两个子类 SliverChildListDelegate 和 SliverChildBuilderDelegate，SliverChildListDelegate 常用于构建少量数据 Item 的场景，它会一次性将所有的子 Item 绘制出来。

SliverChildBuilderDelegate 常用于构建大量的数据，采用懒加载的模式来加载数据与 ListView.builder 的加载原理一致。

一般在实际应用开发的特殊场景中，当上述 ListView 构建方式无法满足时，才使用 ListView.custom 方式，如这里需要获取页面上 ListView 屏幕中显示的第 1 个 Item 在 ListView 中对应的 postion 索引，代码如下：

```
//代码清单 5-4-4 通过 custom 来构建
//lib/code5/example_510_ListView.dart
buildListView6() {
  return ListView.custom(
    //缓存空间
    cacheExtent: 0.0,
    //自定义代理
    childrenDelegate: CustomScrollDelegate(
          (BuildContext context, int index) {
            //构建子 Item 显示布局
       return Container(
         height: 80,
         //根据索引来动态计算生成不同的背景颜色
         color: Colors.cyan[100 * (index % 9)],
       );
      },
      itemCount: 1000,//子 Item 的个数
      //滑动回调
      scrollCallBack: (int firstIndex, int lastIndex) {
        print("firstIndex $firstIndex lastIndex $lastIndex");
      },
    ),
  );
}
```

自定义滑动监听回调 CustomScrollDelegate，代码如下：

```
//代码清单 5-4-5 ListView 自定义滑动监听回调
//lib/code5/example_510_ListView.dart
class CustomScrollDelegate extends SliverChildBuilderDelegate {
  //定义滑动回调监听
  Function(int firstIndex, int lastIndex) scrollCallBack;
  //构造函数
  CustomScrollDelegate(builder,
      {required int itemCount, required this.scrollCallBack})
      : super(builder, childCount: itemCount);

  @override
  double? estimateMaxScrollOffset(int firstIndex, int lastIndex,
      double leadingScrollOffset, double trailingScrollOffset) {
      scrollCallBack(firstIndex, lastIndex);
```

```
        return super.estimateMaxScrollOffset(
            firstIndex, lastIndex, leadingScrollOffset, trailingScrollOffset);
      }
    }
```

5.2.2 GridView 用来构建二维宫格页面

GridView 用来构建二维网格列表，GridView 创建方法有 5 种，如图 5-4 所示，描述如下：

（1）GridView 的构造函数方法，一次性构建所有的子条目，适用于少量数据。

（2）通过 GridView.builder 方式来构建，懒加载模式，适用于大量数据的情况。

（3）通过 GridView.count 方式来构建，适用于固定列的情况，适用于少量数据。

（4）通过 GridView.extent 方式来构建，适用于条目有最大宽度的限制情况，适用于少量数据。

（5）通过 GridView.custom 方式来构建，可配置子条目的排列规则，也可配置子条目的渲染加载模式。

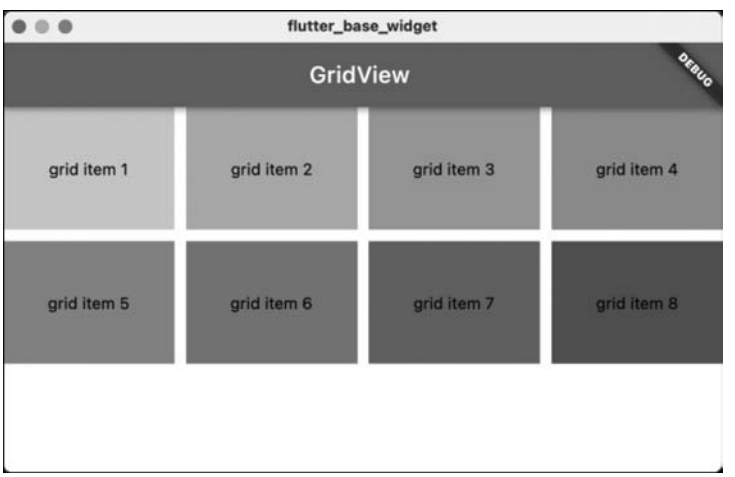

图 5-4 GridView 效果图

通过 GridView 构造函数、count 方法与 extent 方式来构建，都是一次性将所有的子 Item 构建出来，所以只适用于少量的数据，在实际业务开发中，如果数据少于一屏内容，则建议使用少数数据构建方式。

通过 GridView 的构造函数来构建，通过参数 children 来构建 GridView 中用到的所有的子条目，通过参数 gridDelegate 配置 SliverGridDelegate 来配置子条目的排列规则，可以使用 SliverGridDelegateWithFixedCrossAxisCount 和 SliverGridDelegateWithMaxCrossAxisExtent。

通过 SliverGridDelegateWithFixedCrossAxisCount 来构建一个横轴为固定数量的子条

目的 GridView，代码如下：

```
//代码清单 5-5-1 GridView 通过构造函数来创建
//llib/code5/example_511_GridView.dart
Widget buildGridView1() {
  return GridView(
    //子 Item 排列规则
    gridDelegate: const SliverGridDelegateWithFixedCrossAxisCount(
        crossAxisCount: 4,        //横轴元素的个数
        mainAxisSpacing: 10.0,    //纵轴间距
        crossAxisSpacing: 10.0,   //横轴间距
        //子组件宽和高的比值
        childAspectRatio: 1.4),
    //GridView 中使用的子 Widegt
    children: [ … ],
  );
}
```

通过 SliverGridDelegateWithMaxCrossAxisExtent 来构建横轴 Item 数量不固定的 GridView，其水平方向 Item 的个数由 maxCrossAxisExtent 和屏幕的宽度及 padding 和 mainAxisSpacing 来共同决定，代码如下：

```
//代码清单 5-5-2 GridView 通过构造函数来创建
//llib/code5/example_511_GridView.dart
Widget buildGridView2() {
  return GridView(
    //子 Item 排列规则
    gridDelegate: const SliverGridDelegateWithMaxCrossAxisExtent(
      maxCrossAxisExtent: 120,      //子 Item 的最大宽度
      mainAxisSpacing: 10.0,        //纵轴间距
      crossAxisSpacing: 10.0,       //横轴间距
      childAspectRatio: 1.4,        //子组件宽和高的比值
    ),
    //GridView 中使用的子 Widegt
    children: [ … ],
  );
}
```

GridView 的 count 用来构建每行有固定列数的宫格布局，参数 crossAxisCount 为必选参数，用来配置列数，与使用 GridView 通过 SliverGridDelegateWithFixedCrossAxisCount 方式来构建的效果一致，代码如下：

```
//代码清单 5-5-3 GridView 以 count 方式来创建，适用于少量数据
//llib/code5/example_511_GridView.dart
```

```
Widget buildGridView3() {
  return GridView.count(
    crossAxisCount: 4,          //每行的列数
    mainAxisSpacing: 10.0,      //纵轴间距
    crossAxisSpacing: 10.0,     //横轴间距
    //所有的子条目
    children: [ … ],
  );
}
```

GridView 的 extent 用来构建列数不固定，限制每列的最大宽度或者高度的宫格布局，参数 maxCrossAxisExtent 为必选参数，用来配置每列允许的最大宽度或者高度，与使用 GridView 通过 SliverGridDelegateWithMaxCrossAxisExtent 方式来构建的效果一致，代码如下：

```
//代码清单 5-5-4 GridView 以 extent 方式来创建适用于少量数据
//llib/code5/example_511_GridView.dart
Widget buildGridView4() {
  return GridView.extent(
    maxCrossAxisExtent: 120,  //每列 Item 的最大宽度
    mainAxisSpacing: 10.0,    //纵轴间距
    crossAxisSpacing: 10.0,   //横轴间距
    //所有的子条目
    children: [ … ],
  );
}
```

GridView 的 builder 方式来构建，是通过懒加载模式来构建的，参数 gridDelegate 用来配置子 Item 的排列规则，与 GridView 的构造函数中 gridDelegate 使用一致，可分别使用 SliverGridDelegateWithFixedCrossAxisCount 构建固定列数的宫格和 SliverGridDelegateWithMaxCrossAxisExtent 构建不固定列数与固定 Item 最大宽度或者高度的宫格，代码如下：

```
//代码清单 5-5-5 GridView 以 builder 方式来创建
//懒加载模式适用于大量数据
//llib/code5/example_511_GridView.dart
Widget buildGridView5() {
  return GridView.builder(
    cacheExtent: 120,                       //缓存区域

    padding: const EdgeInsets.all(8),       //内边距
    itemCount: 100,                         //条目的个数
    //子 Item 排列规则
```

```
    gridDelegate: const SliverGridDelegateWithMaxCrossAxisExtent(
      maxCrossAxisExtent: 100,       //子 Item 的最大宽度
      mainAxisSpacing: 10.0,         //纵轴间距
      crossAxisSpacing: 10.0,        //横轴间距
      childAspectRatio: 1.4,         //子组件宽和高的比值
    ),
    //懒加载构建子条目
    itemBuilder: (BuildContext context, int index) {
      //index 子 Item 对应的索引
      return buildListViewItemWidget(index);
    },
  );
}
```

5.2.3 PageView 实现页面整屏切换

PageView 可用于 Widget 的整屏滑动切换，如常用的短视频 App 中的上下滑动切换的功能，也可用于横向页面的切换，如 App 第一次安装时的引导页面，也可用于开发轮播图功能，代码如下：

```
//代码清单 5-6-1 PageView 以 builder 方式来创建
//lib/code5/example_508_PageView.dart
//懒加载模式适用于大量数据
PageView buildPageView() {
  return PageView.builder(
    //当页面选中后回调此方法
    //参数[index]是当前滑动到的页面角标索引,从 0 开始
    onPageChanged: (int index) {
      print("当前的页面是 $index");
      currentPage = index;
    },
    //当值为 false 时显示第 1 个页面,然后从左向右开始滑动
    //当值为 true 时显示最后一个页面,然后从右向左开始滑动
    reverse: false,
    //滑动到页面底部的回弹效果
    physics: const BouncingScrollPhysics(),
    scrollDirection: Axis.vertical,     //纵向滑动切换
    controller: pageController,         //页面控制器
    //所有的子 Widget
    itemBuilder: (BuildContext context, int index) {
      return SizedBox(
        width: MediaQuery.of(context).size.width,
        child: Image.asset(
          "assets/images/loginbg.png",
```

```
        ),
      );
    },
  );
}
```

PageController 可以监听 PageView 的滑动监听，通过 PageView 的属性 controller 来绑定，代码如下：

```
//代码清单 5-6-2 PageView 控制器
//lib/code5/example_508_PageView.dart
//初始化控制器
late PageController pageController;
//PageView 当前显示页面索引
int currentPage = 0;

@override
void initState() {
  super.initState();
  //创建控制器的实例
  pageController = PageController(
    //用来配置 PageView 中默认显示的页面，从 0 开始
    initialPage: 0,
    //当为 true 时保持加载的每个页面的状态
    keepPage: true,
  );

  //PageView 设置滑动监听
  pageController.addListener(() {
    //PageView 滑动的距离
    double offset = pageController.offset;
    //当前显示的页面的索引
    double? page = pageController.page;
    print("pageView 滑动的距离 $offset 索引 $page");
  });
}
```

通过控制器 PageController 可以将 PageView 主动滑动到指定的位置，核心代码如下：

```
//代码清单 5-6-3 控制器常用方法
//lib/code5/example_508_PageView.dart
void pageViewController() {
  //以动画的方式滚动到指定的页面
  pageController.animateToPage(
    0,//子 Widget 的索引
```

```
      curve: Curves.ease,//动画曲线
      //滚动时间
      duration: const Duration(milliseconds: 200),
    );

    //以动画的方式滚动到指定的位置
    pageController.animateTo(
      100,
      //动画曲线
      curve: Curves.ease,
      //滚动时间
      duration: Duration(milliseconds: 200),
    );

    //无动画切换到指定的页面
    pageController.jumpToPage(0);
    //无动画切换到指定的位置
    pageController.jumpTo(100);
  }
```

5.3 滑动视图的应用

RefreshIndicator 可结合滑动视图实现下拉刷新效果，ListView、GridView、PageView 在加载多数据时，时常会用到分页加载功能，本节通过滚动监听，可以实现当滑动列表视图时动态隐藏操作按钮功能。

5.3.1 ListView 下刷新与分页加载

RefreshIndicator 是 Material 风格的滑动刷新 Widget，可在 ListView 与 GridView 的外层直接嵌套使用，如图 5-5 所示。

图 5-5 RefreshIndicator 下拉刷新效果图

代码如下：

```
//代码清单 5-7-1 RefreshIndicator 基本使用
//lib/code5/example_513_ListView_RefreshIndicator.dart
buildRefreshIndicator() {
  return RefreshIndicator(
      color: Colors.blue,                  //圆圈进度颜色
      displacement: 44.0,                  //下拉停止的距离
      backgroundColor: Colors.grey[200],  //背景颜色
      //下拉刷新的回调
      onRefresh: () async {
        //模拟网络请求
        await Future.delayed(Duration(milliseconds: 2000));
        //清空数据业务操作

        //结束刷新
        return Future.value(true);
      },
      child: buildListView(),//ListView
  );
}
```

手动触发 RefreshIndicator 刷新，需要定义 GlobalKey，类型为 RefreshIndicatorState，然后通过 RefreshIndicator 的 key 绑定，代码如下：

```
final GlobalKey<RefreshIndicatorState>
      _indicatorKey = GlobalKey<RefreshIndicatorState>();

_refresh() {
  _indicatorKey.currentState?.show();
}
```

对于上滑加载更多数据，在这里通过 ListView 的滑动控制器来监听 ListView 当前滑动的距离，当滑动的距离超出总长的 2/3 时，静默加载更多数据，代码如下：

```
//代码清单 5-7-2 ListView 加载更多数据
//lib/code5/example_513_ListView_RefreshIndicator.dart
//ListView 中使用的滑动控制器
final ScrollController _scrollController = ScrollController();
//是否在加载更多
bool isLoadingMore = false;
//当前加载的页数
int pageIndex = 1;

@override
```

```
void initState() {
  super.initState();

  //添加滑动监听,在这里实现的是预一屏加载
  //也就是当数据滑动查看还只剩下一屏显示时
  //如果用户还在滑动就触发加载更多数据功能
  //在网络正常的情况下就可达成静默加载效果
  _scrollController.addListener(() {
    //获取滑动的距离
    double offset = _scrollController.offset;
    //ListView 可滑动的最大距离
    double maxOffset = _scrollController.position.maxScrollExtent;
    //当前视图的一屏高度
    double height = MediaQuery.of(context).size.height;
    if (offset >= maxOffset * 2/3 && !isLoadingMore) {
      print("上拉加载更多数据");
      //更新标识,防止重复调用,加载完成后更新标识
      isLoadingMore = true;
      //当前数据页数更新
      pageIndex++;
      //加载更多数据方法
      loadMoreData();
   }
  });
}
```

5.3.2 苹果风格下拉刷新

一个苹果风格的下拉刷新效果,在 Flutter 中需要结合滑动视图 CustomScrollView 来实现,如图 5-6 所示。

图 5-6 CupertinoSliverRefreshControl 下拉刷新效果

代码如下：

```
//代码清单 5-8 ListView 加载更多数据
//lib/code4/example_403_progress_page.dart
buildCustomScrollView() {
  return CustomScrollView(
    slivers: <Widget>[
      //下拉刷新组件
CupertinoSliverRefreshControl(
        //下拉刷新回调
        onRefresh: () async {
          //模拟网络请求
          await Future.delayed(const Duration(milliseconds: 5000));

        },
      ),
      //列表
      SliverList(
        delegate: SliverChildBuilderDelegate((content, index) {
          return ListTile(
            title: Text('测试数据 $ index'),
          );
        }, childCount: 100),
      )
    ],
  );
}
```

5.3.3 PageView 实现轮播图特效

通过 PageView 实现水平轮播图效果，如图 5-7 所示。读者可观看视频讲解【5.3.3 PageView 实现轮播图特效】。

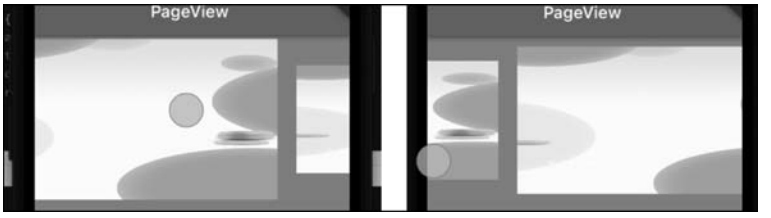

图 5-7 水平轮播图

代码如下：

```
//代码清单 5-9-1 PageView 构建
//lib/code5/example_509_PageView.dart
```

```
buildPageView() {
  return SizedBox(
    height: 200,
    child: PageView.custom(
      controller: _pageController,  //控制器
      //子 Item 的构建器,当前显示的和即将显示的子 Item 都会回调
      childrenDelegate: SliverChildBuilderDelegate(
        (BuildContext context, int index) {
          //子 Item 变换操作,移出和移进都会回调
          return buildTransform(index);
        },
        //普通数据集合
        childCount: _imageList.length,
      ),
    ),
  );
}
```

需要用到 PageView 滑动控制器来监听 PageView 的滑动,并获取当前页面中显示的 Page 的索引,代码如下:

```
//代码清单 5-9-2 PageView 控制器
//lib/code5/example_509_PageView.dart
//初始化控制器
late PageController _pageController;
//PageView 当前显示页面的索引
double? _currentPage = 0;
@override
void initState() {
  super.initState();

  //创建控制器的实例
  _pageController = PageController(
    //用来配置 PageView 中默认显示的页面,从 0 开始
    initialPage: 0,
    //当为 true 时保持加载的每个页面的状态
    keepPage: true,
  );
  //PageView 设置滑动监听
  _pageController.addListener(() {
    //PageView 滑动的距离
    setState(() {
      _currentPage = _pageController.page;
    });
  });
}
```

结合 Transform，实现 PageView 展示切换特效，代码如下：

```
//代码清单 5-9-3 Transform 构建
//lib/code5/example_509_PageView.dart
buildTransform(int index) {
  if (_currentPage != null) {
    //计算
    if (index == _currentPage!.floor()) {
      //出去的 Item
      return Transform(
          alignment: Alignment.center,
          transform: Matrix4.identity()
            ..rotateX(_currentPage! - index)
            ..scale(0.98, 0.98),
          child: buildItem(index));
    } else if (index == _currentPage!.floor() + 1) {
      //进来的 Item
      return Transform(
          alignment: Alignment.center,
          transform: Matrix4.identity()
            ..rotateX(_currentPage! - index)
            ..scale(0.9, 0.9),
          child: buildItem(index));
    } else {
      print("当前显示 $index");
      return buildItem(index);
    }
  } else {
    return buildItem(index);   //buildItem 用来构建具体的 UI 视图
  }
}
```

5.3.4 NestedScrollView 下拉刷新失效问题

在使用 NestedScrollView 结合 RefreshIndicator 实现下拉刷新功能时，会出现无法触发刷新问题，解决方法如下：

```
//代码清单 5-10 NestedScrollView 下拉刷新
//lib/code5/example_506_NestScrollView.dart
buildRefreshIndicator(){
  return RefreshIndicator(
    //可滚动组件在滚动时会发送 ScrollNotification 类型的通知
    notificationPredicate: (ScrollNotification notifation) {
      //该属性包含当前 ViewPort 及滚动位置等信息
```

```
        ScrollMetrics scrollMetrics = notifation.metrics;
        if (scrollMetrics.minScrollExtent == 0) {
          return true;
        } else {
          return false;
        }
      },
      //下拉刷新回调方法
      onRefresh: () async {
        //模拟网络刷新,等待 2s
        await Future.delayed(Duration(milliseconds: 2000));
      },
      //NestedScrollView
      child: buildNestedScrollView(),
    );
  }
```

5.3.5 滚动监听 NotificationListener

当滑动组件开始滑动时,实际上手指接触屏幕那一刻就会触发 ScrollStartNotification 通知消息,滑动组件结束滑动时会触发 ScrollEndNotification,滑动中会有 ScrollUpdateNotification,滑动越界时会触发 OverscrollNotification 通知,在 Flutter 中,可通过 NotificationListener 来捕捉这些滑动通知,代码如下:

```
//代码清单 5-11 滚动监听者
//lib/code5/example_503_NotificationListener.dart
buildNotificationListener() {
  return NotificationListener<ScrollNotification>(
    onNotification: (ScrollNotification notification) {
      //滚动信息封装对象
      ScrollMetrics metrics = notification.metrics;
      //可滑动的最大距离
      double max = metrics.maxScrollExtent;
      double extentBefore = metrics.extentBefore;
      //距离底部边距
      double extentAfter = metrics.extentAfter;
      if (_scrollController.hasClients) {
        //根据滑动组件绑定的控制器来判断监听的是哪个组件触发的滑动
        double maxScrollExtent = _scrollController.position.maxScrollExtent;
        if (max == maxScrollExtent) {
          print("netsScrollView 滑动 extentAfter $extentAfter");
        }
      }
```

```
        return false;
      },
      child: buildScrollView,
    );
  }
```

对于 NotificationListener 的 onNotification()方法可以理解为事件分发,在这里如果返回值为 true,则代表消费事件,滑动通知不会再向上发送,也就是上级的 NotificationListener 无法捕捉到滑动通知,反之为 false 时,就会一级一级地向上传递。

5.3.6 ListView 实现自动滚动标签效果

如视频播放时,在视频播放小窗口下会有播放集的横向列表显示,单击对应的小集,标签会自动向前或者向后滚动,效果如图 5-8 所示。读者可观看视频讲解【5.3.6 ListView 实现自动滚动标签效果】。

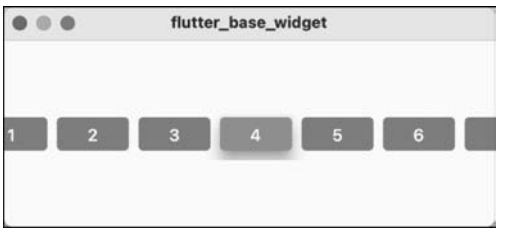

图 5-8 水平滚动标签

在页面中,组合页面可能非常复杂,本节使用 StreamBuilder 实现局部刷新功能,代码如下:

```
//代码清单 5-12 横向滚动标签
class _DemoListViewFlagPageState extends State<DemoListViewFlagPage> {

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      //层叠布局
      body: Center(
        child: Stack(children: [
          //页面中的其他布局

          //横向滚动的标签
          SizedBox(
            height: 44,
            child: buildStreamBuilder(),
```

```
            )
          ]),
        ),
      );
    }
    //以多订阅流的方式来创建
    final StreamController < int > _streamController
                                    = StreamController.broadcast();
    StreamBuilder < dynamic > buildStreamBuilder() {
      return StreamBuilder < int >(
        stream: _streamController.stream,
        builder: (BuildContext context, AsyncSnapshot < int > snapshot) = >
            buildListView(),
      );
    }

    @override
    void dispose() {
      _streamController.close();
      super.dispose();
    }
  }
```

横向滚动的标签通过 ListView 来构建，需要获取当前屏幕上显示的第 1 个 Item 与最后一个 Item 的位置，所以使用 5.2.1 节代码清单 5-4-5 中自定义的代理 CustomScrollDelegate，代码如下：

```
  //代码清单 5 - 12 - 1 横向 ListView 构建
    //滑动控制器
    final ScrollController _scrollController = ScrollController();
    int _firstIndex = 0;                  //屏幕上列表显示第 1 个标签位置
    int _lastIndex = 0;                   //屏幕上列表显示最后一个标签位置
    int _currentIndex = 0;                //当前选中的标签

    buildListView() {
      return ListView.custom(
        padding: const EdgeInsets.only(left: 5, right: 5),
        controller: _scrollController,
        cacheExtent: 0.0,                 //缓存空间
        scrollDirection: Axis.horizontal, //横向滑动
        //自定义代理
        childrenDelegate: CustomScrollDelegate(
          (BuildContext context, int index) {
            //ListView 的子条目
            return Row(
```

```
            mainAxisSize: MainAxisSize.min,
            children: [buildItemContainer(index)],
          );
        },
        //条目的个数
        itemCount: 1000,
        //滑动回调
        scrollCallBack: (int firstIndex, int lastIndex) {
          _firstIndex = firstIndex;
          _lastIndex = lastIndex;
        },
      ),
    );
  }
```

然后在子 Item 布局构建中，通过 Container 来限制标签的大小与外边距，通过 ElevatedButton 实现单击事件，代码如下：

```
//代码清单 5-12-2 标签构建
Container buildItemContainer(int index) {
  return Container(
      margin: EdgeInsets.only(left: 4, right: 4),
      width: 60,
      height: 28,
      child: ElevatedButton(
          style: ButtonStyle(
            elevation: MaterialStateProperty.all(
              index == _currentIndex ? 10 : 0,
            ),
            backgroundColor: MaterialStateProperty.resolveWith(
              (states) {
                if (index == _currentIndex) {
                  return Colors.orange;
                }
                //默认状态使用灰色
                return Colors.grey;
              },
            ),
          ),
          onPressed: () => clickAction(index),
          child: Text("$index")));
}
```

单击按钮时，调用 ListView 绑定的滑动控制器 scrollController 实现标签列表的自动滚动功能，代码如下：

```
//代码清单 5-12-3 标签单击事件自动滚动
  clickAction(int index) {
    _currentIndex = index;
    //计算当前屏幕中间显示的 Item 索引
    double mid = (_firstIndex + _lastIndex) / 2;
    //获取当前 ListView 滑动的距离
    double offset = _scrollController.offset;
    if (index > mid) {
      //向左
      _scrollController.animateTo(offset + 100,
          duration: Duration(milliseconds: 400), curve: Curves.easeInSine);
_streamController.add(_currentIndex);
    return;
    }
    //当前显示的是第 1 个 Item
    if (_firstIndex == 0) {
      //完全显示的第 1 个不滑动
      if (offset == 0) {
_streamController.add(_currentIndex);
        return;
      }
      //未完全显示的第 1 个滑动到顶部
      _scrollController.animateTo(0,
          duration: Duration(milliseconds: 400), curve: Curves.easeInSine);
_streamController.add(_currentIndex);
      return;
    }
    //向右
    _scrollController.animateTo(offset - 100,
        duration: Duration(milliseconds: 400), curve: Curves.easeInSine);
    //刷新页面显示
    _streamController.add(_currentIndex);
  }
```

5.4 小结

本章概述了 Flutter 项目中用来处理滑动视图的系列 Widget，App 运行在多种手机上，需要多种机型与屏幕尺寸适配，使用滑动视图是一个不错的选择，CustomScrollView 与 NestedScrollView 用来结合 Sliver 家族的 Widget 实现酷炫的滑动折叠图特效，ListView 用来处理列表视图，大部分 App 有这样的应用场景，GridView 用来处理宫格排版类，PageView 的合理应用，可以实现各种轮播图效果及页面的横向或者纵向的整屏内容切换。