第 5 章

分 页 呈 现

显示数据是大部分应用程序的主要功能。当业务数据量较大时如何高效地加载与渲染页面，以及当用户滑动屏幕翻页时怎样平滑且快速地加载新的数据是本章重点讨论的内容。

5.1 列表和网格

5.1.1 ListView

ListView 组件是 Flutter 框架中最常用的支持滚动的列表组件。它的基础用法非常简单，只需通过 children 参数传入一系列组件，ListView 就能将它们依次摆放，并支持用户滑动屏幕的手势，或在有鼠标的设备上支持鼠标滚轮，自动实现滚动功能，代码如下：

```
ListView(
    children: [
        Text("1"),
        Text("2"),
        Text("3"),
    ],
)
```

运行效果与使用 Column 组件类似，垂直方向依次摆放 3 个 Text 组件，但支持滚动。

1. 动态加载

在安卓原生开发中，RecyclerView 是一个可以高效地展示大量数据的列表控件。在 iOS 原生开发中，UITableView 控件也能起到同样的作用。它们的共同特点是在程序运行时不会立即加载列表中的全部数据，而是通过测量屏幕高度及需要显示的数据的高度，计算出满满一屏幕究竟可以显示几行数据，并只加载那部分数据。当用户向下滚动屏幕时，它会动态加载新的数据，并同时将移出屏幕的数据回收，以节约计算机资源。例如，某通讯录页面需要显示共 200 条通信记录，但测量后得出当前设备的屏幕高度最多只能显示 12 条，这些有动态加载机制的控件会选择只加载屏幕上必须显示的 12 条及若干额外条目作为缓冲（如额外加载 3 条），这样一共只需加载 15 条记录，其余的 185 条记录暂不作处理。随着用

户向下滚动屏幕，最顶部的记录会逐渐移出屏幕的可见区域，这时用来显示它们的控件会被回收(recycle)。系统可将回收后的控件翻新，将旧内容改成即将出现的新内容，再从屏幕底部插入。这样程序运行时，用户可以自由滚动到列表的任意位置，但其背后始终只需大约15个控件互相交替，回收及循环，动态实现全部资料的显示。安卓原生开发中的RecyclerView也因此得名，其中recycler就是回收者的意思。

在Flutter框架中，使用ListView组件的默认构造函数会使其立即初始化children列表，从而无法发挥动态加载的全部优势，因此，ListView组件的默认构造函数只建议在children数量较少时使用。

一般情况下，长列表推荐使用ListView.builder()命名构造函数。此时，children参数将不可使用，取而代之的是itemBuilder回传函数。该回传函数有上下文(context)和位置索引(index)参数，开发者需要根据这2个参数，尤其是位置索引，返回一个供ListView渲染的组件。

例如，可对列表中的每个位置生成一个Text组件，文本内容为位置索引，代码如下：

```
ListView.builder(
    itemBuilder: (context, index) {
        return Text("这是一个Text组件,索引: $index");
    },
)
```

运行效果如图5-1所示。

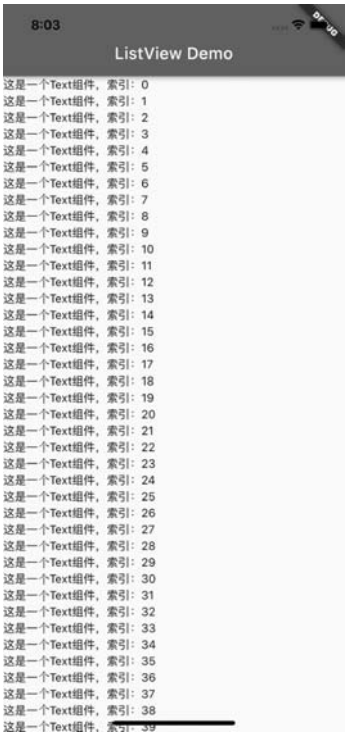

图5-1　利用ListView动态生成子组件

1）列表长度

默认情况下，动态创建的ListView列表的内容长度无限，即用户可以一直向下滑动屏幕，永远不会触碰到底边。例如当用户即将浏览到第1000个元素的时候，ListView便会自动开始加载第1001个元素，即调用itemBuilder函数并将1001个元素传入index，再将该函数返回的那个组件作为第1001个元素渲染到列表的合适位置。

如需限制列表长度，则可在使用ListView.builder()时通过itemCount参数传入一个不小于0的整数。如传入itemCount:20，则ListView只会渲染20个元素。它在调用itemBuilder回传函数时，传入的位置索引只可能是0～19。

当用户已经浏览到列表末尾但依然向下滚动屏幕时，ListView会自动产生触边反馈效果。在iOS上呈现的是过度滚动后自动弹回的动画效果，在安卓上呈现的是波形色块的效果，这些都是相应平台中非常自然且常见的效果，用户应该不会感到陌生。

2）分割线

ListView 列表中元素之间若需要分割线，则可以借助 ListView.separated()构造函数轻松实现。这个构造函数的用法与 ListView.builder()大同小异，主要区别有 2 点：首先是除了 itemBuilder 回传函数外还多加了一个 separatorBuilder 回传，用于在元素之间插入分割线，其次是 itemCount 不可为空，代码如下：

```
ListView.separated(
  itemCount: 3,
  separatorBuilder: (context, index) {
    return Center(
      child: Text("--- 这是索引为 $ index 的分割线 ---"),
    );
  },
  itemBuilder: (context, index) {
    return Container(
      height: 50,
      color: Colors.grey,
      alignment: Alignment.center,
      child: Text("这是索引为 $ index 的元素"),
    );
  },
)
```

ListView 会首先根据 itemCount 属性确定该列表共 3 个元素，并在调用 itemBuilder 函数构建 3 个元素的基础上，额外调用 separatorBuilder 函数在元素之间插入 2 次分割线，如图 5-2 所示。

图 5-2 在 ListView 元素之间插入分割线

Flutter 框架中有一个名为 Divider 的组件，可用于渲染 Material 风格的分割线。若需使用，则可以直接在 separatorBuilder 方法中回传“return Divider();”完成分割线的建造。如需自定义样式，读者也可参考第 11 章“风格组件”中有关 Divider 的介绍。

2. 子组件尺寸

一般情况下 ListView 列表中的元素（子组件）的尺寸都是交给每个子组件自行决定的，因此不同元素之间尺寸也可相差甚远。例如，可构建一个长度为 10 的列表，其中每当索引为奇数时，返回一个灰色且较高的 Container 容器，其余索引则直接返回 Text 组件，代码如下：

```
ListView.builder(
  itemCount: 10,
  itemBuilder: (context, index) {
    if (index.isOdd) {
      return Container(
```

```
            height: 60,
            color: Colors.grey,
            alignment: Alignment.center,
            child: Text("索引: $ index"),
          );
        } else {
          return Text("这是索引为 $ index 的元素");
        }
      },
    )
```

运行效果如图 5-3 所示。

图 5-3 ListView 中子组件元素可以不等高

这种不约束子组件尺寸的布局思路在大部分情况下是一种既灵活又方便的设计方案。实战中 Flutter 的性能也足以轻松应付用户滑动屏幕时连续地调用 itemBuilder 实时创建并渲染组件，但这么做也有缺点：例如 ListView 无法预知未加载的元素尺寸，继而无法确定所有元素的总尺寸，这样有时会导致右侧的滚动条(稍后介绍)无法准确显示滚动进度。又或当程序支持大幅跳转时，不固定每个元素的高度可能会导致性能问题。

在这些特殊情形下，选择牺牲元素尺寸的多样性可换来性能的提升。实战中可通过 ListView 的 itemExtent 属性强制固定每个元素的尺寸，如在上例中增加 itemExtent：80 即可保证列表中的每个元素在列表滚动的主轴方向必须占用 80 逻辑像素。换言之，如果 ListView 是竖着滚动的，则每个元素的高度必为 80 单位，如果 ListView 是横着滚动的，则每个元素的宽度均为 80 单位。

固定子组件的主轴尺寸可在大幅跳转时提升性能。例如某程序可通过滚动控制器(稍后介绍)实现一键跳转 10000 逻辑像素的功能。若 ListView 无法提前确定每个元素的高度，则跳转时它必须依次加载这些元素并完成布局测量，这样才可得知跳转 10000 逻辑像素后应该落在第几个元素上，然而，若每个元素的高度是固定的，如 80 逻辑像素，则只需简单计算便可得知一共需跳过 10000÷80＝125 个元素，跳转后应显示第 126 个元素，这样就不必逐个加载被跳过的元素了。

3. 空白页面

空白页面是当下非常流行的一种界面设计。例如，当通讯录列表为空时，程序可显示“暂时还没有联系人，单击此处添加”，或者当垃圾邮件列表为空时，程序可显示“太棒了，没有垃圾邮件!”等特制图形或文案。这么做通常会比直接渲染一个空空的列表更友好些。

ListView 组件本身没有直接支持空白页的功能，但借助 Flutter 灵活的框架，只需要在

合适的时机将整个 ListView 组件替换成另一套显示空白页面的组件(如 Image 或者 Text 组件),就可以轻松实现这个需求了。

4. 内部状态保持

在 ListView 组件对元素的动态加载与资源回收机制的作用下,移出屏幕的元素会被摧毁。当用户往回翻页并再次浏览到该元素时,这个元素又会被重新加载。如果元素内部存有状态,并且在摧毁时没有注意保存,则重新加载时可能会丢失之前的状态。

这里通过一个实例来举例说明。首先定义一个含有 30 个元素的 ListView 列表,每个元素都是自定义的 Counter 组件,即一个简单的计数器,拥有内部状态。完整代码如下:

```
//第 5 章/list_view_state.dart

import 'package:flutter/material.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text("ListView 演示"),
        ),
        body: ListView.builder(
          itemCount: 30,
          itemExtent: 80,
          itemBuilder: (context, index) {
            return Center(child: Counter(index));
          },
        ),
      ),
    );
  }
}

class Counter extends StatefulWidget {
  final int index;

  Counter(this.index);

  @override
  _CounterState createState() => _CounterState();
}
```

```
class _CounterState extends State<Counter> {
  int _count = 0;

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      child: Text("第${widget.index + 1}个计数器: $_count"),
      onPressed: () => setState(() => _count++),
    );
  }
}
```

当用户单击按钮时，对应的计数器就会自增。例如程序运行后，第 1 个按钮被单击了 3 下，它就会显示数字 3，其余按钮保持显示 0 的状态，如图 5-4 所示。

图 5-4 ListView 元素拥有内部状态

由于一般手机屏幕并不能同时显示全部 30 个元素，用户此时可滑动屏幕查看更多内容。当第 1 个显示着数字 3 的计数器被移出屏幕时，对应的 Counter 组件会被摧毁，因此其内部状态(count 变量)也会丢失。具体表现为当用户先向下滚动屏幕，再向上滚动屏幕回来时，会意外发现第 1 个计数器已被重置，显示着数字 0 而不是之前的数字 3 了。

遇到类似情况时，就有必要对 ListView 内部元素进行状态保持。

1) 状态提升

当列表的子组件的内部状态会意外丢失时，最直接的解决办法是采纳前端网页 React 框架中著名的 Lift State Up(状态提升)思路，把列表中每个子组件的状态都提升到列表之上，这样当子组件被摧毁重制时状态就不会丢失了。

例如之前的计数器例子，可以把计数器的内部状态提取到外部，将 30 个计数器的数值一并保存为一个外部数组，这样 Counter 组件需要显示的数字由外部数组传入，就不会发生数值丢失的情况了。

2) KeepAlive

实战中并不是所有情况都适合采用状态提升的思路。当程序的设计架构导致确实有保留元素的内部状态的必要时，也可采用 KeepAlive(保持活跃)的方式使内部状态不丢失。

ListView 组件中的 addAutomaticKeepAlives 参数可自动为子组件添加 KeepAlive 功能，而且默认值已经是 true，因此不需要额外设置，但开发者应确保子组件本身必须同时支持该功能。上例中，计数器组件的状态类需要做一定改写。

比较简单的改写方法是先将 CounterState 类添加 AutomaticKeepAliveClientMixin 融

合类，接着在其 build 方法中调用父级方法，最后添加继承 wantKeepAlive(是否需要保持活跃)返回值为 true。修改后的代码如下：

```
class _CounterState extends State<Counter>
    with AutomaticKeepAliveClientMixin              //1. 添加融合类
{
  int _count = 0;

  @override
  Widget build(BuildContext context) {
    super.build(context);                           //2. 调用父级方法
    return ElevatedButton(
      child: Text("第${widget.index + 1}个计数器: $_count"),
      onPressed: () => setState(() => _count++),
    );
  }

  @override
  bool get wantKeepAlive => true;                   //3. 声明需要保持状态
}
```

这样修改后的 Counter 组件就不会在移出屏幕时丢失内部数据了，即实现了保持内部状态的效果。如需进一步优化，则可在声明是否需要保持状态时，仅保持非 0 状态的计数器，代码如下：

```
@override
bool get wantKeepAlive => _count != 0;        //3. 仅当计数器不为 0 时需保持状态
```

这样若用户尚未开始使用某个计数器，则它显示的数字本来就是 0，也就没有必要保存了。

5. 滚动控制器

ListView 组件可以通过 controller 参数接收一个 ScrollController 类的滚动控制器。开发者可以通过它更直接地掌控当前列表的状态，以及控制列表滚动。例如当用户单击程序顶部的导航栏时，可以触发跳转至列表顶部的操作，或者在程序刚开始运行时，设置列表默认的起始位置等。使用完毕后，应在适宜的时机调用 ScrollController 的 dispose 方法释放资源。

1) 初始化

滚动控制器的初始化代码不建议放在 build 方法中，否则 Flutter 每次重绘都会初始化一个新的控制器。实战中一般在 State(组件状态类)中定义私有变量_controller，并初始化为一个新的 ScrollController 控制器，之后再将其通过 controller 属性传给 ListView 组件，代码如下：

```
class _MyHomePageState extends State<MyHomePage> {
  ScrollController _controller = ScrollController();        //初始化

  @override
  void dispose() {
    super.dispose();
    _controller.dispose();                                  //回收资源
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(),
      body: ListView.builder(
        controller: _controller,                            //在 ListView 组件中使用控制器
        itemBuilder: (_, index) => Text("$index"),
      ),
    );
  }
}
```

2) jumpTo

当滚动控制器通过 controller 参数传给 ListView 组件后，开发者就可以通过它直接操作列表的滚动状态。其中最简单的一种操作就是 jumpTo(跳转至)方法：传入一个小数类型的值，列表就会跳转到这个位置。例如可制作一个按钮，当用户单击按钮时，通过调用 jumpTo(0.0) 将列表跳转至 0.0 逻辑像素的位置，即回到列表顶部，代码如下：

```
ElevatedButton(
  child: Text("跳转至顶部"),
  onPressed: () => _controller.jumpTo(0.0),
)
```

值得一提的是，如果这里故意传入负数，如－5.0 或者－10.0 等，列表则会跳转至顶部后过量滚动，产生触顶动画并自动纠正至 0.0。这与用户平时用手指迅速滚动列表并触边时的视觉效果一致，在 iOS 上呈现的是过量滚动后自动弹回的动画效果，而在安卓上呈现的是波形色块的效果，因此实战中传入负数会使整个“快速跳转至顶部”的功能看上去更加自然，读者不妨一试。

3) animateTo

除了跳转外，滚动控制器还支持 animateTo(动画至)方法。这与跳转的最终效果类似，但不是瞬间完成，代码如下：

```
ElevatedButton(
  child: Text("回到顶部"),
  onPressed: () => _controller.animateTo(
```

```
        0.0,
        duration: Duration(milliseconds: 300),
        curve: Curves.easeOut,
      ),
    )
```

以上代码定义了一个凸起按钮，单击之后 ListView 将开始动画滚动至 0.0 的位置，滚动速度逐渐缓慢，总耗时 300ms。其中 duration(时长)和 curve(动画曲线)都是 Flutter 框架中与动画相关的常见属性，对此不熟悉的读者可参考第 7 章“过渡动画”中的相关内容。

4) offset

开发者若需要得到当前列表的位置，则可以通过访问 controller.offset 属性获取。这里的返回值是逻辑像素，例如列表中每个元素的高度均为 100 单位，那么当 offset 返回值 50 时，则表示现在列表刚开始滚动，它的第 1 个元素有一半已经移出屏幕了。再例如，若 offset 返回值 2048，则表示列表的顶部显示的是第 21～22 个元素的位置。

除了 offset 外，控制器还支持 controller.position 属性，用来提供更详细的信息，如滚动物理等，有兴趣的读者可自行查阅相关文档。其实上文介绍的 controller.offset 属性就是 controller.position.pixels 的语法糖，用于方便大家查询最常用的列表位置信息。

5) 事件监听

滚动控制器可以通过 addListener 方法添加一个或多个回传函数，并在滚动值发生变化时调用。例如可在列表发生滚动时，打印出当前位置，代码如下：

```
_controller.addListener(() {
  print("现在的位置: ${_controller.offset}");
});
```

这样每当列表滚动时就会源源不断地打印出当前 offset 的值，直到滚动完毕，整个列表缓缓停下，并最终彻底静止后才会停止打印。

6. 其他属性

1) scrollDirection

一般情况下 ListView 组件以垂直方向进行滚动，视觉效果类似于一个可支持滚动的 Column 组件。通过 scrollDirection 属性，ListView 也可以变成水平方向滚动，就如同 Row 组件一样。

若想改为水平方向滚动，则只需传入 scrollDirection: Axis.horizontal。对应的垂直方向值为 Axis.vertical，或者直接删掉该参数，即可采用默认的垂直方向。

2) reverse

ListView 组件可通过 reverse: true 开启“倒序”模式。在默认垂直方向滚动的列表中开启倒序模式会使最后一个元素显示在列表的最顶部，而第 1 个元素显示在最底部。当使

用 jumpTo 等方法跳转至 0.0，即列表初始位置时，列表也会跳转至底部，因此仍然会跳转到第 1 个元素。若元素不够占满整个视窗的高度，倒序的列表则由下自上摆放完全部元素后，会在上方留白，这也与正序列表恰好相反。

若 ListView 组件被 scrollDirection 属性设置为水平方向滚动，则默认顺序是由设备的阅读顺序决定的。例如，在阿拉伯文的系统上，默认水平方向是由右到左滚动，这与汉语或英文的设备默认方向不同。无论默认方向如何，这里的“倒序”都会将其反转。

3）padding

列表外部的留白可以通过在 ListView 组件的父级添加 Padding 组件实现，而列表内部留白则可以用 padding 属性设置。两者的主要区别在于，当列表的元素比较多（超出视窗范围）时，外部留白会将屏幕上的 ListView 组件当作一个整体，始终保持它与其他组件（或与屏幕边框）的留白，但内部留白则是将 ListView 组件内的所有元素当作一个整体，始终保持元素与 ListView 的边框的距离。

为了演示，这里将一个 ListView 组件嵌入一个灰色 Container 中，以方便观察它的尺寸和位置。接着借助 Padding 组件将 ListView 的四周外部留白 48 单位，再同时使用 padding 属性将其四周内部留白 72 单位，代码如下：

```
Padding(
  padding: EdgeInsets.all(48),                    //外部留白 48 单位
  child: Container(
    color: Colors.grey,                           //灰色背景
    child: ListView.builder(
      padding: EdgeInsets.all(72),                //四周内部留白 72 单位
      itemCount: 100,
      itemBuilder: (_, index) {
        return Text("这是列表的第 $ index 项");
      },
    ),
  ),
)
```

运行效果如图 5-5 所示。首先灰色的 ListView 与屏幕边的 48 单位留白是由父级 Padding 组件造成的。其次 ListView 与其内部元素之间的 72 单位留白则是由 padding 属性导致的，这里可以观察到左边、右边及上边的留白，却没有底部留白。这是由于 padding 属性设置的内部留白会将全部元素当作一个整体。这里由于屏幕高度的限制，共 100 项的 ListView 目前只显示了 31 项，因此底部没有出现留白。可想而知，当用户滚动到最后一个元素时便可观察到底部的留白了。

值得一提的是，ListView 组件的 padding 属性默认值不是 0，而是会根据设备的不同，自动避让当前设备屏幕上的缺陷区域（如某款苹果手机的“刘海儿”等位置）。若对此默认行为不满意，则可以通过手动传入 EdgeInsets.zero 直接将四周设置为固定的 0 留白。

这里还有一个小技巧：在实战中，若程序布局采用了 FloatingActionButton 设计，即右

下角出现一个悬浮的按钮，则很可能会导致列表的最后一个元素被悬浮按钮遮住。这时可利用 padding 属性为底部增加留白，这样列表中的最后一个元素会与 ListView 的底边保持距离，空出悬浮按钮的位置，方便用户浏览最后一个元素，运行效果如图 5-6 所示。

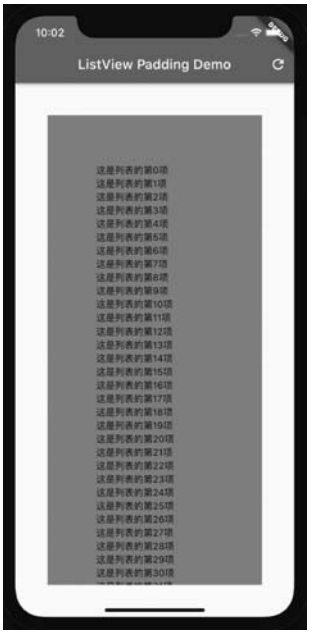

图 5-5 ListView 组件内部与外部留白的对比

图 5-6 ListView 组件的内部底边留白

对渲染悬浮按钮不熟悉的读者可参考第 11 章“风格组件”中有关 FloatingActionButton 组件和 Scaffold 组件的简介。另外，对 Padding 组件不熟悉的读者也可以参考第 6 章“进阶布局”中关于 Padding 组件的详细介绍。

4）shrinkWrap

这里 shrinkWrap 在英文里是“真空包装”的意思，指的是用塑料薄膜将物体严实包裹后再把其中的空气抽掉，一般用于为物体运输途中减小体积或为食品保鲜。在 ListView 组件中，shrinkWrap 是指将列表主轴方向的尺寸压缩至全部子组件尺寸的总和，使其尽量少占空间。

默认情况下 ListView 组件不采用 shrinkWrap，因此在主轴方向会占满父级组件允许的最大尺寸，例如竖着滚动的列表如果没有其他约束，就会占满屏幕高度。若传入 true 启用真空包装，则 ListView 的高度会变为 children 高度之和，因此当元素较少时可被方便地嵌套在 Column 组件中，代码如下：

```
Column(
  children: [
    Text("列表之前"),
```

```
      Container(
        color: Colors.grey[400],
        child: ListView(
          padding: EdgeInsets.all(8.0),
          shrinkWrap: true,
          children: [
            Text("#1. 达拉崩吧斑得贝迪卜多比鲁翁"),
            Text("#2. 昆图库塔卡提考特苏瓦西拉松"),
          ],
        ),
      ),
      Text("列表之后"),
    ],
  )
```

运行效果如图 5-7 所示。

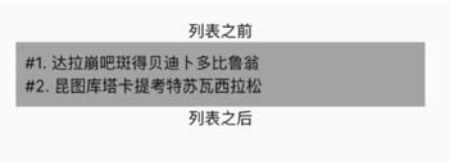

图 5-7　被 shrinkWrap 后的列表

启用 shrinkWrap 的 ListView 列表不得不放弃动态加载，即便使用了 ListView.builder 构造函数，它也会立刻将所有的元素全部加载，以计算尺寸总和，因此启用 shrinkWrap 是一项非常消耗计算资源的操作。若列表中的元素较多（甚至无限多），则真空包装需要非常多的时间才能将列表中的每个元素加载并布局，这样会使程序出现卡顿甚至无响应。

实战中很少需要启用 shrinkWrap。如上例 ListView 中元素较少（只有 2 项），可考虑直接将它们并入 Column 组件的 children 中。若 ListView 元素较多，则更常见的办法是借助 Expanded 组件嵌入 Column 中。这样可保持 ListView 的动态加载和滚动，最终运行效果也与真空包装不同。关于 Column 等 Flex 组件中的 Expanded 的用法和原理，读者可以参考第 6 章“进阶布局”中的相关内容。

5）physics

滚动物理，可以设置列表滚动时的物理样式。例如，当用户将列表滚动至边界后继续滚动，在 iOS 上会出现过量滚动后自动弹回的动画效果，而在安卓上呈现的是波形色块的效果。这些行为属于 physics 的一部分。

由于 Flutter 引擎是通过直接在设备进行像素级别的绘制来渲染画面的，所以这些列表滚动及触边的动画效果等并不是调用系统的 API 完成的，因此均不受设备的原生操作系统局限。例如，传入 physics：BouncingScrollPhysics()可使所有设备表现出 iOS 默认的触边回弹效果，而传入 ClampingScrollPhysics()则可以使所有设备表现出安卓的效果。

另外，传入 NeverScrollableScrollPhysics()可禁止列表的滚动，这样即便列表元素超过视窗范围，用户也不能滑动手指滚动屏幕，但开发者仍可使用 controller 读取和控制列表的位置，以及完成跳转等操作。与之相对的 AlwaysScrollableScrollPhysics()则确保列表永远可以滚动。

如有必要，读者还可以继承 ScrollPhysics 类，创建符合项目需求的滚动物理。例如这里新建一个滚动物理，叫作 AutoScrollPhysics，其功能是自动使列表向下滚动，速度为固定的每秒 200 逻辑像素，且不受用户的手势影响，永不停止。完整代码如下：

```
//第 5 章/list_view_physics_auto.dart

import 'package:flutter/material.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text("Auto Scroll Physics"),
        ),
        body: ListView.builder(
          physics: AutoScrollPhysics(),
          itemBuilder: (_, index) => Text("$index"),
        ),
      ),
    );
  }
}

class AutoScrollPhysics extends ScrollPhysics {
  @override
  ScrollPhysics applyTo(ScrollPhysics? ancestor) => AutoScrollPhysics();

  @override
  bool shouldAcceptUserOffset(ScrollMetrics position) => false;

  @override
  Simulation createBallisticSimulation(position, velocity) =>
      AutoScrollSimulation();
}

class AutoScrollSimulation extends Simulation {
  static const velocity = 200.0;

  @override
  double x(double time) => velocity * time;

  @override
```

```
    double dx(double time) => velocity;

    @override
    bool isDone(double time) => false;
  }
```

6）cacheExtent

在ListView动态加载与回收元素时，除了屏幕上可见的子组件外，ListView还会在视窗范围外（如垂直滚动列表的上方和下方）额外加载几个元素作为缓冲。这里cacheExtent属性定义了缓冲区的长度。例如设置cacheExtent：1000，则表示在当前屏幕可见的第1个元素之前，以及可见的最后一个元素之后，分别插入长度为1000逻辑像素的缓冲区。例如当前视窗高度为800逻辑像素，这样就一共产生了1000＋800＋1000共计2800逻辑像素的缓冲区，所有缓冲区内的组件都会被加载。这就意味着，若每个元素的高度为100单位，虽然手机屏幕一次只能显示8～9个元素，但始终都有近30个元素被加载到内存中，时刻准备着被渲染至屏幕上。

实战中很少需要手动设置这个属性，一般直接使用默认值250即可。

7）semanticChildCount

semanticChildCount是语义标签属性，属于辅助功能的一部分，用来协助第三方软件为有障碍人士提供无障碍功能。例如盲人一般会通过某些软件将屏幕上的内容朗读出来，而这里的语义标签就可以帮助屏幕朗读软件，以提供更友好的用户体验。

当朗读软件遇到列表时可能会朗读“列表共12项内容”等语句，但列表中有些元素也许是装饰性的分割线或者标题等，不应该被认为是列表内容，因此可以用semanticChildCount提供实际有意义的元素的数量，如semanticChildCount：8表示列表里只有8项真正有意义的内容。使用该属性时需注意列表元素总量不能为无限，且semanticChildCount不能超过itemCount的数量。

7. 扩展到Sliver

ListView的本质是一个只有一个SliverList的CustomScrollView，因此当ListView无法满足某些复杂的需求时，例如当需要列表与网格（稍后介绍）联合滚动，或当程序顶部的导航条也需要参与滚动时，可考虑直接使用CustomScrollView。如有需求，读者可参考本书第13章“滚动布局”的内容。

5.1.2 ListWheelScrollView

转轮列表，ListWheelScrollView是一个将子组件放在一个转轮上并可以呈现三维显示效果的列表组件。其基础用法与ListView组件相似，通过children参数接收一系列组件，将它们依次三维变换后摆放，并支持用户滑动屏幕手势。不同的是这个组件的itemExtent参数必传，因此所有列表中的元素在主轴方向必须是统一尺寸，不支持大小不一的子组件，代码如下：

```
ListWheelScrollView(
  itemExtent: 100,
  children: [
    for (int i = 0; i < 5; i++) Container(color: Colors.grey),
  ],
)
```

这里通过 for 循环，向 children 传入了 5 个灰色的 Container 组件，并通过 itemExtent 属性将列表内每个元素的高度都设置为 100 单位。程序运行时，默认显示第 1 个元素，且将其放大并居中，效果如图 5-8(左图)所示。当用户开始滚动列表，如滚动至第 3 个元素时，被“选中”的第 3 个元素将被放大且居中，效果如图 5-8(右图)所示。

图 5-8 ListWheelScrollView 的显示效果

Dart Tips 语法小贴士

列表中的 if 和 for 循环

上例代码利用 for 循环，方便地向 ListWheelScrollView 组件的 children 属性传入了 5 个组件。这是 2019 年 5 月发布的 Dart 2.3 的新语法，允许列表中使用 if 和 for 关键字，代码如下：

```
Column(
  children: [
    if (showHeader)                    //根据变量判断是否在列表中插入 Header 组件
      Header(),
```

```
      Item1(),                              //固定插入 Item1 组件
      Item2(),                              //固定插入 Item2 组件
      for(int i = 0; i < 10; i++)
        FlutterLogo(),                      //循环插入 10 遍 FlutterLogo 组件
      Item3(),                              //固定插入 Item3 组件
    ],
  )
```

实战中除了在列表中直接使用 for 之外，也可以使用 List.generate 功能，代码如下：

```
ListWheelScrollView(
  itemExtent: 100,
  children: List.generate(5, (index) => FlutterLogo()),
)
```

上述代码使用 List.generate()传入 5，表示需要生成一个长度为 5 的列表，接着传入一个回传函数。运行时，Flutter 会根据所设置的长度，连续调用回传函数，并依次提供递增的索引(index)。本例中的回传函数会被调用 5 次，其中 index 变量将分别对应 0、1、2、3、4。每次返回的值会被最终合并到一个列表中，因此，上述代码最终会生成 5 个 FlutterLogo 组件。

此外，实战中也常常会根据业务逻辑，直接借助数据集的 map 等功能直接将数据转换为组件。很多其他编程语言都有类似的方法，5.1.3 节也会简单介绍。

1. 渲染三维效果

1) offAxisFraction

该属性用于控制转轮中的 children 远离中心轴的偏差值，默认为 0，即无左右偏离。当传入一个正数时，转轮会向观测者角度的右侧偏移，负数则向左侧偏移。数值的绝对值越大，偏离得越多。例如传入 offAxisFraction：－1.2，会产生如图 5-9 所示的偏移效果。

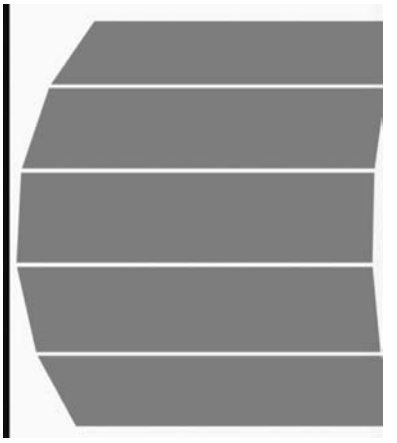

图 5-9 使用 offAxisFraction 设定中心轴的偏差值

2) 放大中心元素

如有必要，则可以通过 useMagnifier(启用放大镜)属性传入 true 开启中心元素的放大功能。开启后，还需要通过 magnification 属性传入放大倍数，默认 1.0 倍无效果，例如传入 2.0 表示放大 2 倍，传入 0.5 表示缩小至原来一半的尺寸，代码如下：

```
ListWheelScrollView(
  itemExtent: 80,
  useMagnifier: true,                   //启用放大镜
```

```
    magnification: 1.5,                    //放大 1.5 倍
    children: List.generate(
      5,
      (index) => Container(
        color: Colors.grey,
        alignment: Alignment.center,
        child: Text("这是第 $ index 项"),
      ),
    ),
  )
```

运行效果如图 5-10 所示。

3）overAndUnderCenterOpacity

除了可以放大中心元素外，ListWheelScrollView 组件还支持将不在中心位置的其他元素添加半透明的效果。这个名字比较长的属性 overAndUnderCenterOpacity（中心上面和下面不透明度），主要就是做这个用途的。它可以接收数值 0.0～1.0，默认为 1.0，即没有特殊的半透明效果。

例如，可以传入 overAndUnderCenterOpacity：0.5，呈现 50%透明的效果，如图 5-11 所示。

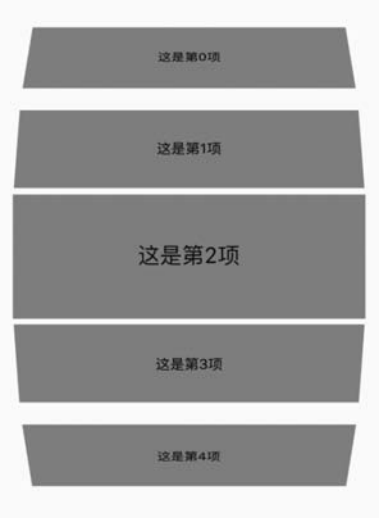

图 5-10　中心元素放大 1.5 倍的效果

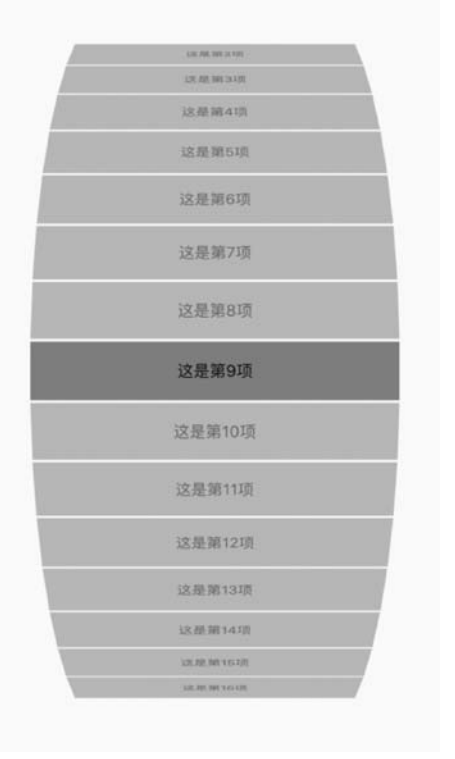

图 5-11　非中心元素半透明的效果

值得一提的是，当使用这个属性时，ListWheelScrollView 组件会自动打开 useMagnifier（启用放大镜）属性，不需要特意设置，因此若需要同时放大中心元素的尺寸，则可直接额外通过 magnification 属性传入放大倍数实现。

4）diameterRatio

转轮直径可以通过 diameterRatio 参数设置，默认为 2.0。该组件的开发者在源码注释中提到，默认选择 2.0 并无特殊含义，就是觉得渲染出的视觉效果看起来还不错。

图 5-12 从左到右依次展示了转轮直径 0.5（较小）、2.0（默认）和 3.0（较大）的效果，读者可自行体会。

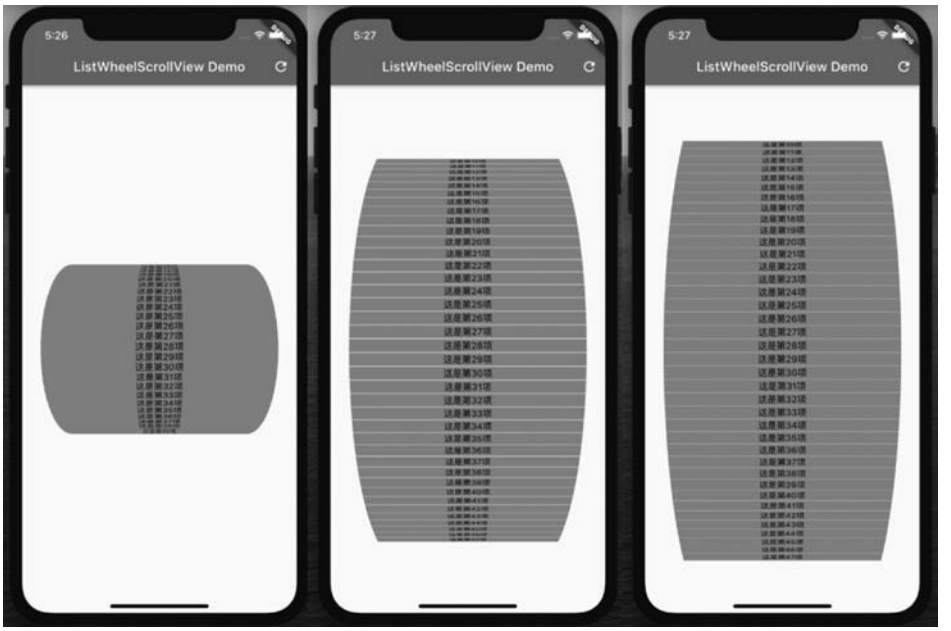

图 5-12　转轮直径设置不同值的效果

用于实现图 5-12 所示效果的代码如下，读者也可自行修改代码，尝试其他的值：

```
ListWheelScrollView(
  itemExtent: 20,
  diameterRatio: 2.0,                          //转轮直径，默认为 2.0
  children: List.generate(
    100,
    (index) => Container(
      color: Colors.grey,
      alignment: Alignment.center,
      child: Text("这是第 $ index 项"),
    ),
  ),
)
```

5）perspective

这个属性定义了将转轮的三维圆柱体投影到二维的屏幕时的视角。视角 0 表示从无限远的距离观察这个圆柱体，而视角 1 则表示从无限近的地方观测，但 0 和 1 这两个表示“无限”的值都不可被实际渲染，因此，该组件的开发者设置了允许的最大值为 0.01，并选择默认值 0.003，该默认值并无特殊含义，只是组件的开发者觉得渲染出的视觉效果看起来还不错。

图 5-13 从左到右依次展示了视角为 0.00001（较远）、0.003（默认）和 0.01（允许范围内最近）的效果，读者可自行体会。本例用到的代码与上例 diameterRatio（转轮直径）的代码类似，只是将 diameterRatio 属性改写为 perspective 属性，代码略。

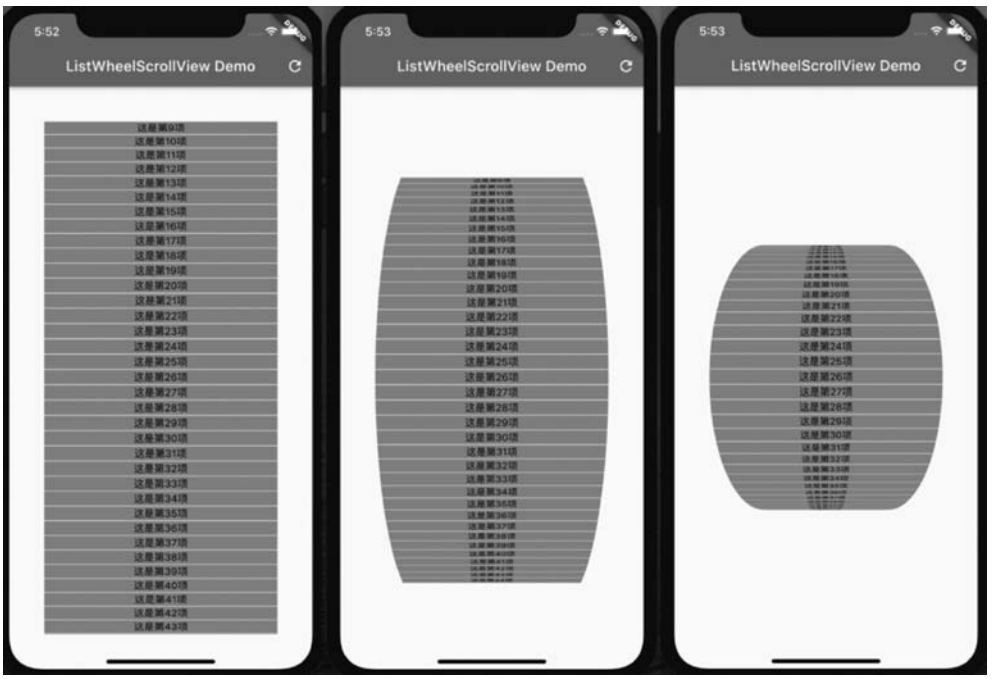

图 5-13　视角属性设置不同值的效果

6）squeeze

这个属性用于控制每个子组件在转盘上插入的位置，或密集程度，默认为 1。若传入 0.5，则转盘会将元素的密度减半，即原先每个可以插入 children 的位置，此时选择插一个空一个，做出一半留白的效果。相反，若传入 2，则转盘会在原先每个元素中间的位置再挤入一个新元素，使密度变成双倍，并可能导致每个元素下半部分被新增的元素所遮挡。

改变转盘的密度会导致屏幕可显示的元素的数量有所增减，因此也会影响程序在运行时需要同时构建的子组件的数量。例如当传入 squeeze：1.5 使密度变为原来的 1.5 倍时，原本一屏幕只能显示 20 个元素的列表，此时可以显示 30 个元素。

2. 精确选择

由于ListWheelScrollView组件的itemExtent属性不能为空，故其所有子组件的高度必须一致，因此列表中的每个元素所对应的子组件在没有加载完成时就已经可以预先确定尺寸了，所以这类列表可以做到对列表内容的精确定位。

1）physics

该组件的滚动物理physics属性与ListView组件的同名属性类似，对此属性不熟悉的读者可以先阅读5.1.1节ListView组件的相关内容。例如设置BouncingScrollPhysics()可使所有设备表现出iOS默认的触边回弹效果，而传入ClampingScrollPhysics()则可以使所有设备表现出安卓的色块效果。此外，还可以传入NeverScrollableScrollPhysics()禁止列表的滚动，以及设置AlwaysScrollableScrollPhysics()可使列表永远可以滚动。

除了上述这些普通的physics值之外，由于ListWheelScrollView支持精确选择，这里还可以传入一个特殊的值：FixedExtentScrollPhysics()，即固定范围的滚动物理。使用这个值可以保证列表滚动停止后最终会稳稳地停在一个元素上，而不是停在两个元素之间的任意位置。

2）onSelectedItemChanged

这是ListWheelScrollView所支持的回传函数，每当用户选择的值发生变化时该函数会被调用，并且会将用户当前选择的元素的索引作为参数传入，以方便开发者实现业务逻辑所需要的功能。

例如可用ListWheelScrollView组件及各项属性，实现一个三维滚动的日期选择器，代码如下：

```
//第5章/list_wheel_date_selector.dart

ListWheelScrollView(
  perspective: 0.005,                          //定义“视角”
  itemExtent: 48,                              //固定元素高度
  magnification: 1.2,                          //中心元素放大1.2倍
  overAndUnderCenterOpacity: 0.5,              //非中心元素半透明
  physics: FixedExtentScrollPhysics(),         //固定范围的滚动物理
  onSelectedItemChanged: (index) {             //选择值改变时的回传
    print("选择了${index + 1}日");
  },
  children: List.generate(                     //生成子组件列表
    31,                                        //共有31个元素
    (index) => Container(
      color: Colors.grey,                      //灰色
      alignment: Alignment.center,             //居中
      child: Text("${index + 1}日"),           //显示日期，如28日
    ),
  ),
)
```

运行效果如图 5-14 所示。

图 5-14 立体的日期选择器

3. 控制器

与 ListView 组件类似，ListWheelScrollView 也可通过 controller 参数接收一个 ScrollController 类的控制器，但由于这个组件支持精确控制，实战中一般会选择传一个更具体的子类，FixedExtentScrollController，即固定范围的滚动控制器。这是一种专门为已经固定了子组件尺寸的列表（目前 Flutter 框架的内置组件中只有 ListWheelScrollView 组件符合这一要求）订制的、更方便好用的滚动控制器。它除了支持普通 ScrollController 所支持的全部功能，如 jumpTo（跳转至）或 offset（读取当前位置）等以逻辑像素为单位的操作外，FixedExtentScrollController 还支持从逻辑像素到“元素索引”的转换。

例如之前介绍过的 jumpTo（100）可以跳转至列表第 100 逻辑像素的位置，但 jumpToItem（100）就可以直接跳转到列表的第 100 个元素开始的位置。同样地，animateToItem（）和 animateTo（）用法类似，但数量单位从逻辑像素变为了元素索引。另外，FixedExtentScrollController 提供 selectedItem 属性，可以获得当前选中（中心位置）的元素索引，非常方便。

该组件的 controller 属性也可以接收一个普通的 ScrollController 控制器，但这样做就会失去精确选择的功能，包括失去这里基于元素索引的操作，以及本节之前提到的 FixedExtentScrollPhysics 和 onSelectedItemChanged 回传函数功能。当不传入任何控制器时，Flutter 会自动为 ListWheelScrollView 创建一个 FixedExtentScrollController 控制器，因此不会失去精确选择的功能。

对普通滚动控制器仍不熟悉的读者可参考本章 5.1.1 节 ListView 组件介绍中的相关内容。

5.1.3 ReorderableListView

ReorderableListView（可排序的列表）顾名思义，是一个支持用户拖动列表中的元素并

任意改变它们的顺序的组件。该组件基本用法比较简单，可利用 builder 方法或直接通过 children 参数传入需要渲染的子组件，再通过 onReorder 参数设置一个回传函数，用来处理当列表组件的顺序发生改变时的业务逻辑。这里唯一需要注意的是，ReorderableListView 组件要求所有子组件的 key 属性不为空，否则在改变顺序时会出现混淆。实战中一般使用 ValueKey 作为标识，代码如下：

```
ReorderableListView(
  children: [
    Text("在汗水中奋斗之后终将绽放", key: ValueKey(1)),
    Text("梦想就如同花朵一样", key: ValueKey(2)),
    Text("努力地前往也永远不曾改变方向", key: ValueKey(3)),
  ],
  onReorder: (int oldIndex, int newIndex) {
    print("用户把位于 $ oldIndex 的元素移动到了 $ newIndex 的位置");
  },
)
```

运行时，列表会先顺位显示 3 个 Text 组件，并在用户长按其中任意组件后进入排序模式。排序时，被拖动的元素会有阴影边框，以增加立体感，如图 5-15 所示。当用户松开手指后即退出排序模式，此时 onReorder 函数会被调用，输出如“用户把位于 1 的元素移动到了 0 的位置”的字样。

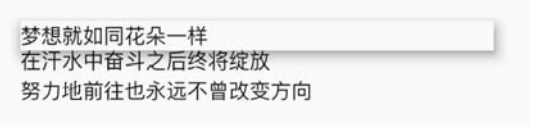

图 5-15　原本列表中第 2 项内容正被拖至第 1 项


(11min)

Flutter 框架小知识

组件中常见的 key 属性是什么

在 Flutter 框架中，组件(Widget)本身是不可变(immutable)的，即组件一旦被创建后，就不可以再改变它的值了，因此当程序的界面需要发生改变时，例如一个 Text 组件中的文字从“张三”换成了“李四”时，Flutter 需要将旧的 Text 组件摧毁，再重新创建一个新的 Text 组件。

由于组件在程序运行的过程中经常会被摧毁重制，它们并不适合保存程序运行时的状态，否则每当组件被摧毁时程序的运行状态就会丢失。Flutter 的 StatefulWidget(有状态的组件)就是通过其附属的 State 类存储状态信息。当某个组件在某一帧被摧毁时，对应的 State 会被暂时保留，并试图在同一帧找到新创建的组件，重新建立对应关系，以达到保存状态的目的。具体寻找的办法就是在组件树(Widget Tree)的相同位置查找相同类型的组件。

于是，当树中某一级有不止一个同样类型的组件（如 ListView 的 children 一般都是同一类型的），且其中部分组件被添加、删除或调整了顺序，Flutter 在寻找 State 与 Widget 对应的过程中就会出现混淆，而 key（键）可以帮助避免混淆。设置了 key 以后，寻找对应关系时 Flutter 不但会检查组件在树中的位置和类型是否相同，还需要再检查 key 的值是否相等。只有在满足了这 3 个条件后，State 才会与新的 Widget 建立对应关系。

Flutter 中的局部键共有 3 种，分别是 UniqueKey、ValueKey 和 ObjectKey。其中，UniqueKey 可以直接使用，并且只与自身相等，其余 2 种在使用时需要传入一个捆绑的值，类型不限。ValueKey 是否相等取决于捆绑的值是否相等（调用具体类的 == 方法对比），而 ObjectKey 是否相等则是根据捆绑的值是否为同一个实例（指针指向相同内存区域）。

1. 属性

1）children

实战中若列表元素较少，可以直接使用 children 参数传入一个子组件列表。若元素较多，则推荐使用 builder 方法实现动态加载，这些都与 ListView 组件无异。唯一不同的是，这里的每个子组件都必须有 key，这样在用户拖动组件改变元素顺序时 Flutter 才能跟踪每个组件的新位置及组件与状态之间的对应关系。

2）onReorder

每当用户完成拖动操作，以及手指离开屏幕时，如果用户的操作确实有将任何元素改变位置，则回传函数就会被调用。如果用户的拖动操作最终并没有将任何元素改变位置，则这里的回传函数不会被调用。

回传函数被调用时会将 oldIndex（旧索引）与 newIndex（新索引）一并作为参数传入。开发者应在该回传函数中处理相关的业务逻辑，例如将列表背后真正的数据中的元素也调整顺序，并通过调用 setState 方法让 Flutter 重新渲染界面。

在用户拖动的过程中，ReorderableListView 会自动处理被拖动的组件的位置（跟随手指移动），以及列表中的其他组件的相应位置（自动避让，为正在移动的组件腾出空间），但当用户完成拖动后，这些自动处理的临时效果也会随即消失，因此，如果这里的回传函数没有及时处理业务逻辑，用户在完成拖动操作后，列表中的元素则依然会按照原来的顺序排列。

3）header

表头，即列表正式开始之前的额外组件。实战中一般可以用这个属性做一些标题之类的用途，若不需要也可以留为空值。这个 header 组件不能被拖动，也不会被改变顺序，永远都是列表的第一项，但它并不是“钉”在屏幕上的：当列表过长而发生滚动时，它也会随着滚动，渐渐移出屏幕可见范围。

4）其他属性

除了上面介绍的这些属性外，ReorderableListView 还有不少与 ListView 组件相同或相似的属性，在此不逐一列举。值得一提的是，负责关联滚动控制器的属性在该组件中被称作 scrollController，但其用法仍和 ListView 的 controller 属性一致。

2. 实例

这里提供一个利用 ReorderableListView 组件完成的色彩排序的小游戏实例，完整代码如下：

```
//第 5 章/reorderable_list_view_example.dart

import 'package:flutter/material.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: MyHomePage(),
    );
  }
}

class MyHomePage extends StatefulWidget {
  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  final shades = [700, 200, 600, 500, 900, 800];

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("ReorderableListView"),
      ),
      body: ReorderableListView(
        children: shades
            .map((shade) => Container(
                  key: ValueKey(shade),
                  height: 50,
                  margin: EdgeInsets.all(4.0),
                  color: Colors.grey[shade],
                ))
            .toList(),
        onReorder: (int oldIndex, int newIndex) {
          if (newIndex > oldIndex) newIndex--;
          setState(() {
            final shade = shades.removeAt(oldIndex);
```

```
                shades.insert(newIndex, shade);
              });
            },
          ),
        );
      }
    }
```

程序最初的运行效果如图 5-16 所示。在运行的过程中，用户可以通过长按并拖动任意色块，从而改变它在列表中的位置。

该例比较简单，首先定义了 shades = [700, 200, 600, 500, 900, 800]变量，用一个数组保存若干色彩深度的信息。接着在 ReorderableListView 中，传入 children 子组件。这里通过 map 方法将数组中的色彩深度信息转换为相应的 Container 组件，并设置了宽度、高度、留白及对应的颜色。接着设置 onReorder 回传函数，即当用户改变列表中的元素顺序时，将相应改动应用到之前定义的 shades 变量，并借助 setState 要求 Flutter 重绘整个组件。

这个例子中值得注意的是，onReorder 回传函数提供的旧索引变量(oldIndex)和新索引变量(newIndex)都以旧列表为参照系。这个设计不一定是最妥当的，但由于历史版本的兼容问题，这个小缺陷应该不会在未来版本中被修复。具体表现为，当用户由下至上拖动元素(如将第 3 个元素拖到第 2 个元素的位置)时，onReorder 回传函数可以正确汇报“将 3 移动至 2”，但当用户由上至下拖动元素(如将第 2 个元素拖到第 3 个元素的位置)时，onReorder 回传函数则会汇报“将 2 移动至 4”，这背后的逻辑如图 5-17 所示。

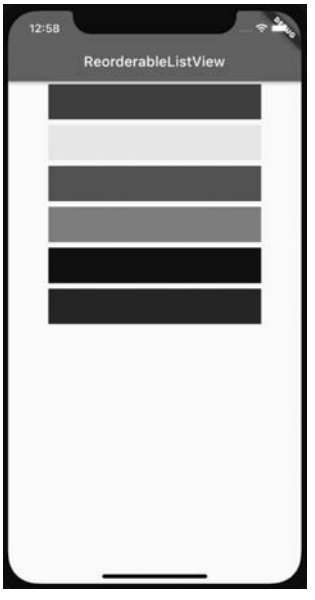

图 5-16　颜色排序游戏的运行效果

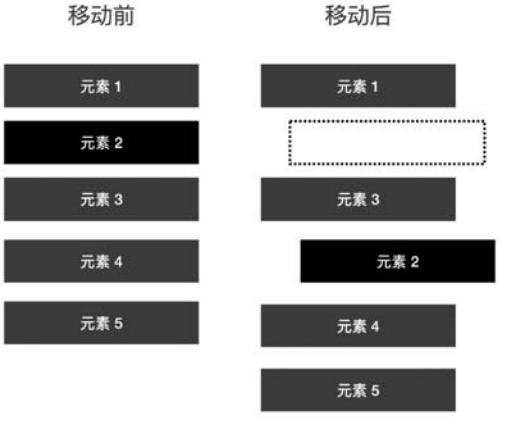

图 5-17　元素 2 与元素 3 调换位置的前后对比

但一般来讲，当元素 2 拖动到元素 3 的下方（同时元素 3 会自动避让，移动到元素 2 的位置，实则两者对调）时，开发者希望得到的数据是“将 2 移动至 3”而不是“将 2 移动至 4”。为了修正这个问题，使元素 2 与元素 3 调换位置时可以获得正确的新旧索引，上例使用了这句代码补丁：

```
if (newIndex > oldIndex) newIndex--;
```

使用了上述代码后，当用户由下至上拖动元素时新索引会减 1，这样可以得到正确的新索引，方便编写业务逻辑代码。

3. 扩展

默认情况下 ReorderableListView 组件要求用户长按后才可以触发拖动模式，然而在上述小游戏实例中，若允许用户直接轻触拖动，而不必长按，就可以显著提升游戏体验。

事实上，ReorderableListView 组件背后调用的是一个更基础的 ReorderableList 组件，开发者也可以直接使用后者。ReorderableList 组件不自动处理拖放手势，因此开发者可根据实际需求，通过向列表内的元素插入 ReorderableDragStartListener 或 ReorderableDelayedDragStartListener 组件之一，自行决定应在轻触后开始拖动，或在长按后开始拖动。

例如可将上例中的 ReorderableListView 组件替换为 ReorderableList 组件，代码如下：

```
//第 5 章/reorderable_list_demo.dart

ReorderableList(
  itemCount: shades.length,
  itemBuilder: (BuildContext context, int index) {
    return ReorderableDragStartListener(
      key: ValueKey(shades[index]),
      index: index,
      child: Container(
        height: 50,
        margin: EdgeInsets.all(4.0),
        color: Colors.grey[shades[index]],
      ),
    );
  },
  onReorder: (int oldIndex, int newIndex) {
    if (newIndex > oldIndex) newIndex--;
    setState(() {
      final shade = shades.removeAt(oldIndex);
      shades.insert(newIndex, shade);
    });
  },
  physics: NeverScrollableScrollPhysics(),
)
```

这里配合 ReorderableDragStartListener 组件，允许用户轻触后直接拖动。另外为了避免整个列表滚动，上述代码还传入了 NeverScrollableScrollPhysics 禁用列表滚动。修正这两个小问题后，用户体验得到显著提升，读者不妨亲自动手试一试。

5.1.4　GridView


(10min)

GridView 组件是一个可将元素显示为二维网格状的列表组件，并支持主轴方向滚动。网格与普通列表 ListView 组件十分相似，建议对 ListView 组件不熟悉的读者先阅读 5.1.1 节的内容。

网格列表最简单的用法是直接将不可滚动的交叉轴的元素数量固定，例如每行固定显示 4 个元素，这样可滚动的主轴就会根据元素的总数量自动确定总行数，代码如下：

```
GridView.count(
  crossAxisCount: 4,
  children: List.generate(
    23,
    (index) => Container(
      color: Colors.grey[index % 6 * 100],
      child: Text("$index"),
    ),
  ),
)
```

这样可实现每行显示 4 个元素，一共包含 23 个元素的网格列表。运行时每个元素的背景都为不同色度的灰色，且显示编号 0～22 的索引，在某款苹果手机的竖屏状态下，运行效果如图 5-18 所示。

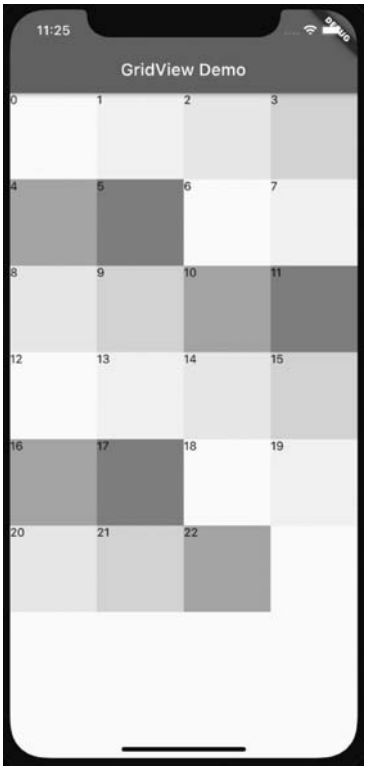

图 5-18　GridView 固定每行 4 个元素的运行效果

1. 构造函数

1) GridView.count()

这是 GridView 组件最易用的构造函数，只需通过 crossAxisCount 传入交叉轴方向的元素数量，并通过 children 传入全部元素，即可实现一个滚动的二维网格状列表。例如，一个默认情况下主轴为垂直方向(竖着滚动)的网格，传入 crossAxisCount:4 就可以使每行固定显示 4 个元素，不因设备屏幕的尺寸而改变。在较窄的屏幕上显示 4 个元素就会使每个元素较小，而在较宽的屏幕上(如平板计算机或者横屏模式下的手机)，则每个元素会比较大。将上例中的同样代码，运行在横屏模式下的某款苹果手机上的效果如图 5-19 所示。

2) GridView.extent()

除了固定交叉轴方向的元素数量(如每行 4 个)这种方式外，开发者也可以选择固定每个元素在交叉轴方向的最大尺寸(例如单个元素宽度不可超过 200 逻辑像素)。这往往是种更好的思路，首先因为用户在较大的屏

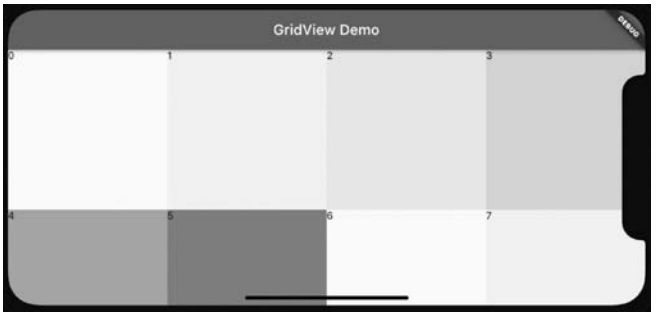

图 5-19　GridView 固定每行 4 个元素的横屏显示效果

幕上通常会期待看到较多的元素，而不只是少量元素的放大版，其次因为大部分情况下无论元素上用到的素材文件还是布局结构的设计，都可能会限制单个元素不宜太大，否则难免会遇到由放大而导致失真的素材图片，或在布局设计上出现大量空白。这里 GridView.extent()构造函数的 maxCrossAxisExtent 参数可以设置每个元素交叉轴方向允许的最大尺寸，这样 GridView 就能根据屏幕的尺寸宽度自动选择合适的每行数量。

例如某款苹果手机屏幕的尺寸为 414×896 逻辑像素，如果竖屏显示，且 GridView 的主轴方向也是垂直方向，则传入 maxCrossAxisExtent:100 就会保证网格的每个元素的最大宽度不超过 100 单位。因为这款手机的屏幕宽度是 414 单位，若每行显示 4 个元素则一共会占用 400 单位，不足以填满屏幕的宽度，因此 Flutter 会自动选择每行显示 5 个元素，这样当每个元素宽度为 82.8 单位时刚好可以填满屏幕，且符合“每个元素的宽度都不超过 100 单位”的要求。当该手机切换到横屏模式时，设备屏幕宽度变成了 896 逻辑像素，因此每行需要显示 9 个元素，才可保证填满屏幕时每个元素的宽度均不超过 100 单位。经计算可得，此时每个元素的宽度约为 99.5 单位。

3) GridView()

这个 GridView 组件的主构造函数没有命名。实际上 GridView.count()和 GridView.extent()这两个命名构造函数可以看作这个主构造函数的语法糖。不同于前 2 者，使用 GridView()时需要传入 children 和 gridDelegate 属性，其中 gridDelegate(网格委托)属性需要传入一个 SliverGridDelegate 类，说明网格该如何构建。Flutter 框架已经提供了该委托的两种实现方式，分别是 SliverGridDelegateWithFixedCrossAxisCount(交叉轴方向固定数量的委托)及 SliverGridDelegateWithMaxCrossAxisExtent(交叉轴方向限制最大长度的委托)，开发者可选择其中一种，直接传入。

当传入这两种委托之一时，实际效果等同于直接使用对应的命名参数。例如使用 SliverGridDelegateWithFixedCrossAxisCount 就与直接使用 GridView.count 等效，代码如下：

```
//使用主构造函数
GridView(
  gridDelegate:                    //传入委托
```

```
        SliverGridDelegateWithFixedCrossAxisCount(crossAxisCount: 4),
    children: children,
)

//直接使用命名参数
GridView.count(
    crossAxisCount: 4,
    children: children,
)
```

4) GridView.builder()

当网格列表中需要显示的元素数量较多(或甚至无限多)时,一般不适宜将全部数据同时加载。这时,使用 GridView.builder()构造函数就可以实现元素的动态加载与回收,以便高效且流畅地展示大量数据。该构造函数背后的动态加载原理与 ListView.builder()函数相同,对此不熟悉的读者可阅读 ListView 相关内容,本书在此不再赘述。

使用 builder 构造函数时 children 参数将不可使用,取而代之的是 itemBuilder 回传函数。该回传函数会提供上下文(context)和位置索引(index)参数,开发者需要根据这 2 个参数,尤其是位置索引,返回一个供 GridView 渲染的子组件。同时,另一个必传函数则是 gridDelegate(网格委托),这与主构造函数的同名属性一致,主要负责设置网格交叉轴方向的渲染方式,指明每行需有几个元素或每个元素的最大宽度。最后,若列表不是无限长,则在使用 builder 构造函数时还应通过 itemCount 参数传入元素的总量,代码如下:

```
GridView.builder(
    gridDelegate:                                    //传入委托
        SliverGridDelegateWithMaxCrossAxisExtent(maxCrossAxisExtent: 100),
    itemCount: 5000000,                              //共 500 万个元素
    itemBuilder: (context, index) {
      return Container(
        color: Colors.grey[index % 4 * 100],
        alignment: Alignment.center,
        child: Text("${index + 1}"),
      );
    },
)
```

这里利用 itemBuilder 回传函数,动态加载共 500 万个元素,并借助委托要求每个元素对应的子组件的最大宽度为 100 单位。例如某款苹果手机的横屏模式下屏幕宽度为 896 单位,则每行需要显示 9 个元素,效果如图 5-20 所示。

这个例子中的 500 万个元素如果不使用动态加载,而是直接全部通过 List.generate()等方法生成后再通过 children 参数传入,则在笔者的测试机上造成了近 3s 的卡顿。当使用上例示范的 builder 方法动态加载后,程序可在毫秒级完成。

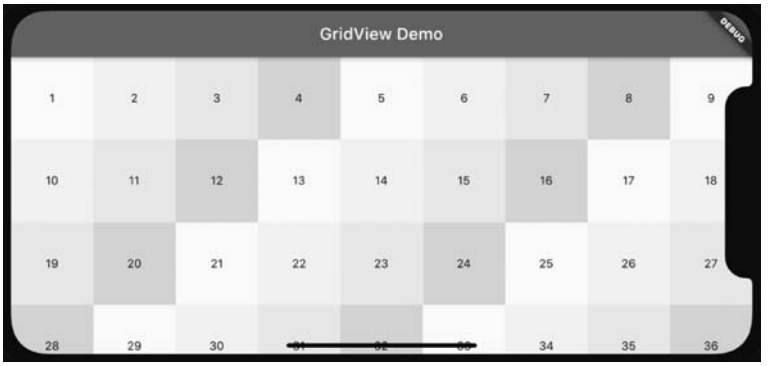

图 5-20　动态加载 500 万个元素也不会卡顿

2. 网格样式

除了上文介绍的交叉轴方向固定元素数量或最大宽度外，网格样式还有 3 个参数，分别是 mainAxisSpacing 属性、crossAxisSpacing 属性和 childAspectRatio 属性，用于设置元素间距和长宽比。当使用 GridView()或 GridView.builder()这 2 个构造函数时，这 3 个属性可在委托类中设置。若直接使用 GridView.count()或 GridView.extent()命名构造函数，因为不会用到委托，所以这 3 个属性应直接传给构造函数。

1）元素间距

元素之间默认不会留白，若需要设置元素间距，则可以通过 mainAxisSpacing 和 crossAxisSpacing 分别设置主轴与交叉轴方向的元素间距，代码如下：

```
GridView.count(
  crossAxisCount: 4,                    //每行 4 个元素
  mainAxisSpacing: 16,                  //主轴间距：16 逻辑像素
  crossAxisSpacing: 4,                  //交叉轴间距：4 逻辑像素
  children: List.filled(50, Container(color: Colors.grey)),
)
```

由于 GridView 的默认滚动方向是垂直方向，因此这里的主轴间距 16 单位会被运用到垂直方向的元素之间，而交叉轴间距 4 单位则会被插入元素的水平方向之间。具体运行效果如图 5-21 所示。

这里同时可以观察到，元素间距不同于列表内的 padding 属性，只会在元素之间插入空白，而不会在网格列表与屏幕边缘之间插入留白。如有必要，实战中也可配合 padding 属性一同使用。

2）元素比例

网格中的元素所对应的子组件默认为 1∶1 的正方形，且由于元素的宽度已被确定（无论是通过固定数量或限制最大宽度），元素的高度因此也只会有唯一的值，所以无论网格内的子组件怎样用 Container 或者 SizedBox 的 width 和 height 属性设置它们的尺寸，都不会有效果。

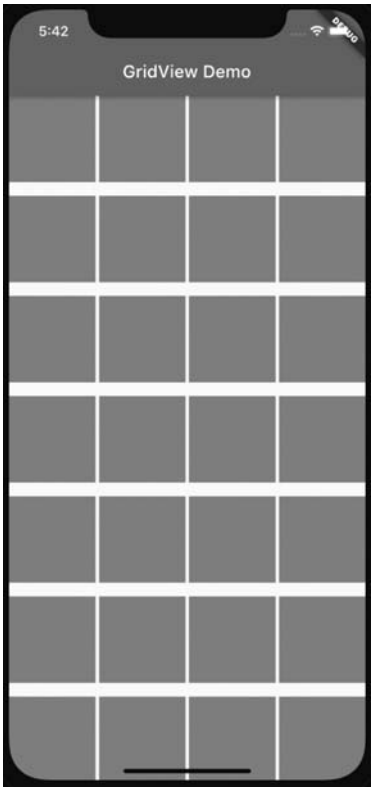

图 5-21　主轴与交叉轴方向的元素间距

如果需要修改网格的长宽比，则可以通过 childAspectRatio 属性传入一个小数，例如需要 3∶2 的长宽比，可以通过传入 1.5 实现，再例如需设置 16∶9 的长宽比，可传入 1.78（16÷9 的近似结果）。为了提高代码的可读性，这里鼓励直接传入“16/9”而不使用计算后的小数，代码如下：

```
GridView.builder(
  gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(
    crossAxisCount: 3,                          //每行 3 个元素
    mainAxisSpacing: 4,                         //主轴间距：4
    crossAxisSpacing: 4,                        //交叉轴间距：4
    childAspectRatio: 16 / 9,                   //长宽比例为 16∶9
  ),
  itemCount: 20,
  itemBuilder: (_, index) => Container(color: Colors.grey),
)
```

这里使用了 builder 构造函数，以展示有委托时如何在委托类中设置元素间距与比例。程序运行效果如图 5-22 所示。

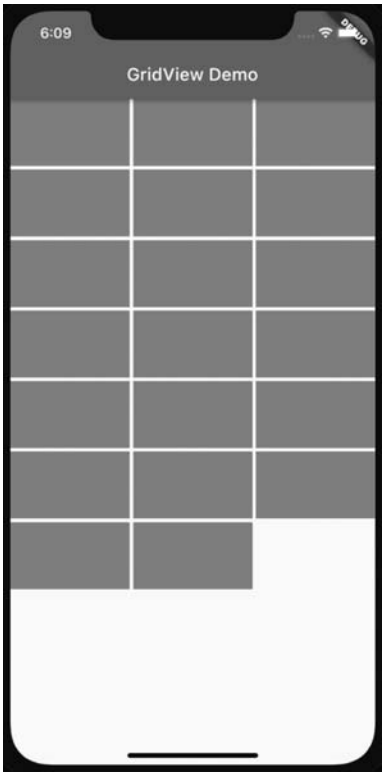

图 5-22　将元素长宽比设置为 16∶9

3. 其他属性

除了上面列举的这些属性外，GridView 还有一部分与 ListView 组件相似的属性，它们分别是 controller(滚动控制器)、scrollDirection(滚动方向)、reverse(倒序)、padding(内部留白)、shrinkWrap(真空包装)、physics(滚动物理)、cacheExtent(缓冲区的长度)及 semanticChildCount(语义元素数量)，这些属性的名称和用法均与 ListView 组件的同名属性一致，不熟悉的读者可查阅本章 ListView 小节的内容。

4. 多种列表样式混搭

实战中利用 shrinkWrap(真空包装)和 NeverScrollableScrollPhysics(禁止列表滚动的物理)，也可勉强实现 ListView 和 GridView 混搭的样式，代码如下：

```
//第 5 章/grid_view_shrink_wrap.dart

ListView(
  children: [
    Row(
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      children: [
```

```
          Container(width: 100, height: 100, color: Colors.grey),
          Container(width: 100, height: 100, color: Colors.grey),
        ],
      ),
      GridView.extent(
        shrinkWrap: true,
        physics: NeverScrollableScrollPhysics(),
        maxCrossAxisExtent: 80,
        padding: EdgeInsets.all(32),
        mainAxisSpacing: 32,
        crossAxisSpacing: 32,
        children: List.generate(
          100,
          (index) => Container(color: Colors.grey),
        ),
      ),
    ],
  )
```

这里主要利用 shrinkWrap 固定了 GridView 主轴方向的长度，再通过禁用滚动，将用户的滚动手势传导至外层的 ListView 上，运行效果如图 5-23 所示。

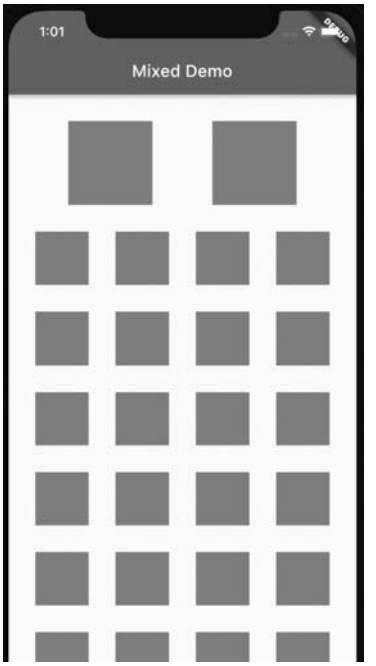

图 5-23 多种列表样式混搭

这个例子中的第一排元素虽然使用了 Row 容器，但也可以根据需要改成 shrinkWrap 的 ListView 或其他任意组件。需要注意的是，利用 shrinkWrap 的思路实现的混搭样式在程序运行时并不高效，因此只适用于列表元素极少的情况。这是由于启用 shrinkWrap 会导致 ListView 或 GridView 等列表放弃动态加载，即便使用了 builder 构造函数，它也会立刻将所有元素加载，以计算尺寸总和。如需高效地混搭有大量元素的列表，或需使程序顶部的导航栏也参与联合滚动，则应考虑使用 Sliver 方式。

5. 扩展到 Sliver

正如 ListView 组件一样，GridView 组件的本质也是 CustomScrollView 组件的简单应用，因此当普通 ListView 或 GridView 无法满足某些复杂的功能时，例如当网格列表需要与普通列表联合滚动，或当程序顶部的导航条也需要参与滚动时，可考虑直接使用 CustomScrollView 组件。对此不熟悉的读者可参考第 13 章“滚动布局”的内容。

5.1.5 PageView

PageView 组件是一个可以实现整屏页面滚动效果的组件，用户滑动一次手指就可以

直接翻动一整个屏幕的距离。页面滚动与普通列表 ListView 组件十分相似，建议对 ListView 组件不熟悉的读者先阅读本章 5.1.1 节的内容。

PageView 的基础用法简单易读，默认为水平方向翻页，代码如下：

```
PageView(
  children: [
    Container(
      color: Colors.grey,
      child: Center(
        child: Text("这是第一页"),
      ),
    ),
    Container(
      color: Colors.white,
      child: Text("第二页"),
    ),
  ],
)
```

从代码中可以看到，这里第 1 个页面是灰色背景，且有一个居中的 Text 组件，写有“这是第一页”字样。第 2 个页面是白色背景，并在默认的左上角的位置有一个 Text 组件，写有“第二页”字样。运行时，程序会先打开第一页。当用户将屏幕滑向第二页时会出现一个滑动的视觉效果，如图 5-24 所示。滑动结束时若用户的滑动力度足够，则程序会翻页且稳稳地停留在第 2 个页面，若用户滑动的力度不够，则会自动弹回第 1 个页面。

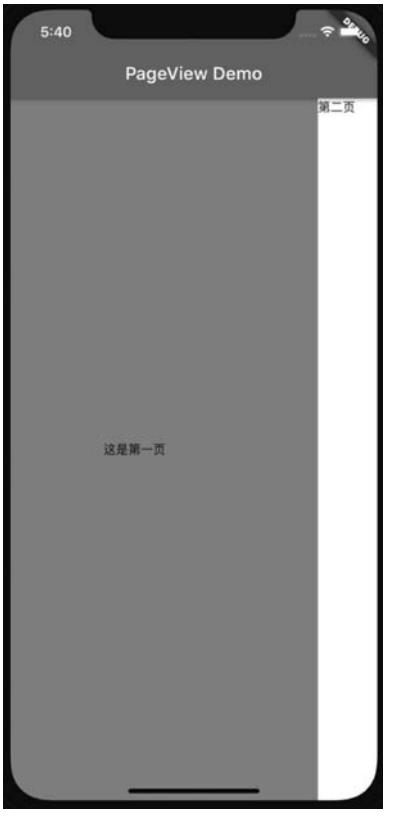

图 5-24　从第 1 个页面滑向第 2 个页面时的效果

由于 PageView 组件主要用于在多个整屏页面之间切换，这里它的每个子组件的尺寸都会被约束为父级允许的最大尺寸，如全屏。

1. 页面固定

默认情况下，PageView 总是会在滚动结束后稳稳地停留在某个页面上，而不会停在 2 个页面之间。当页面为水平方向滚动时，相邻页面之间的滚动幅度为 PageView 的宽度，而当页面为垂直方向滚动时（可通过传入 scrollDirection：Axis.vertical 实现），每次滚动的幅度则为 PageView 的高度。

1）pageSnapping

如需允许用户停留在相邻页面之间的任意位置，

则可传入 pageSnapping：false 实现。例如可先通过 scrollDirection 参数设置垂直滚动，再通过 pageSnapping 属性取消页面固定，代码如下：

```
PageView(
  scrollDirection: Axis.vertical,
  pageSnapping: false,
  children: [
    Container(
      color: Colors.grey,
      child: Center(
        child: Text("这是第一页"),
      ),
    ),
    Container(
      color: Colors.white,
      child: Text("第二页"),
    ),
  ],
)
```

当用户滑动屏幕由第 1 页翻至第 2 页的过程中突然停止，ListView 不会自动完成翻页操作，也不会跳转到第 1 页，而是直接停在两个页面之间，如图 5-25 所示。

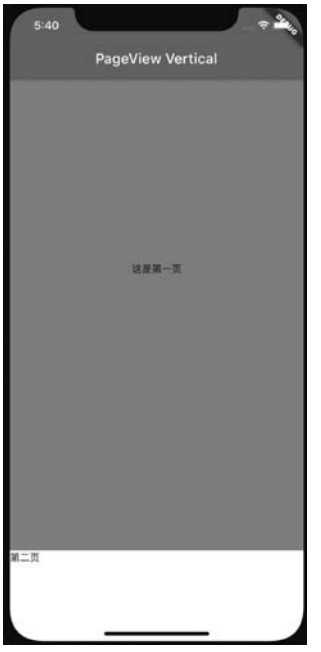

图 5-25 禁用页面固定可使 ListView 在任意位置停留

2）onPageChanged

另外，PageView 组件还支持 onPageChanged（页码变化）属性，可以设置一个回传函数。每当发生翻页时 Flutter 会调用这个函数，并提供当前页面的索引，以方便开发者处理相应的业务逻辑。无论 pageSnapping 是否开启，页码变化的回传函数都会在翻页过半时触发，而不是在动作完成后触发。若用户在两个页面之间反复翻动而不松开手指，这里的回传函数可触发多次。

2. 页面控制器

PageView 组件与之前介绍过的 ListView 及一些其他的类似组件不同，这里 controller 属性需要传入的控制器类型为 PageController（页面控制器），继承于 ScrollController（滚动控制器）。它除了支持普通 ScrollController 所支持的全部功能，如 jumpTo（跳转至）或 offset（读取当前位置）等以逻辑像素为单位的操作外，还可以额外支持从逻辑像素到“页面索引”之间的转换。

例如，jumpTo(300)可以跳转300个逻辑像素，但jumpToPage(3)则可以直接跳转到第4个页面(因为第1个页面的索引是0)。假设某款手机的屏幕尺寸是896×414逻辑像素，若PageView是水平滚动，则一般情况下jumpToPage(3)相当于jumpTo(414.0 * 3)，即3倍于屏幕宽度，但若PageView的页面为垂直方向滚动，则一般情况下还需要从屏幕高度中扣除常见的导航条等元素的高度，最终跳转的幅度为PageView组件的实际尺寸高度的3倍。

同样地，animateToPage()和animateTo()用法类似，但数量单位从逻辑像素变为了页面索引。例如可通过300ms的时间，将页面翻至第3页，代码如下：

```
_controller.animateToPage(
  2,                          //翻至第3个页面(因为第一页的索引是0)
  curve: Curves.linear,
  duration: Duration(milliseconds: 300),
);
```

页面控制器还支持previousPage(前一页)和nextPage(后一页)方法，使用方式与animateToPage()翻页方法类似，但不需要传入目标页码，直接实现由当前所在页面滚动至上一页或下一页的动画效果。

另外，开发者可通过PageController的page属性直接读取当前所在页面。例如屏幕正停留在从第4个页面翻到第5个页面一半的位置时，print(_controller.page)可以得到3.5的输出值。

3. 动态加载

除了通过children属性直接将全部子组件传入以外，PageView组件也支持PageView.builder()这个命名构造函数，通过builder方法动态加载页面列表中的元素。这与ListView组件的builder()用法相同，包含itemBuilder回传函数，并可用itemCount设置元素的总数量，代码如下：

```
PageView.builder(
  itemCount: 20,
  itemBuilder: (context, index) {
    return Center(
      child: Text("这是第${index + 1}页"),
    );
  },
)
```

这里定义了20个页面，每个页面的中心位置都由Text组件显示页码，运行效果略。

4. 其他属性

PageView组件还有几个与ListView组件相同的参数，它们分别是scrollDirection(滚动方向)、reverse(倒序)和physics(滚动物理)，这些属性的名称和用法均与ListView组件的同名属性一致，读者可翻阅5.1.1节关于ListView的介绍。

5.2 滚动监听和控制

5.2.1 Scrollbar

Scrollbar 组件可以为大部分滚动列表添加滚动条。使用时只需要在滚动列表组件(例如 ListView、GridView、ListWheelScrollView 甚至 PageView)的父级插入 Scrollbar 组件,代码如下:

```
Scrollbar(
  child: ListView.builder(
    itemCount: 200,
    itemBuilder: (context, index) {
      return Center(child: Text("这是第${index + 1}个元素"));
    },
  ),
)
```

这样可为 ListView 组件添加一个滚动条,运行在安卓设备时会呈现 Material 风格的滚动条效果,如图 5-26(左图)所示,而在 iOS 或 macOS 设备上则会自动切换为 Cupertino 风格的滚动条,如图 5-26(右图)所示。事实上 Cupertino 风格的滚动条背后是由 CupertinoScrollbar 组件实现的,Flutter 默认会根据程序运行时的当前设备自动适配。若需要在任何设备上都显示 iOS 风格的滚动条,则可以直接使用 CupertinoScrollbar 组件。

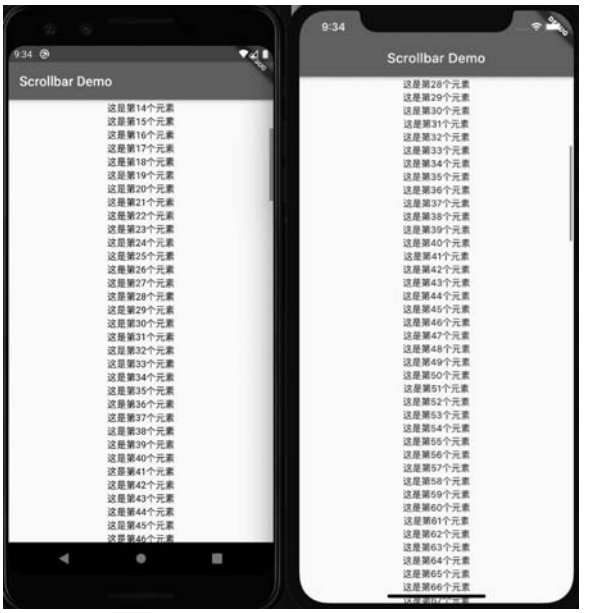

图 5-26 在安卓和 iOS 设备上的滚动条效果

滚动条会在用户开始滚动屏幕时出现，并在滚动完成不久后消失，因此，只有当列表的元素数量足够时才可能观察到滚动条效果。

1. 拖动跳转

自从 iOS 13 系统于 2019 年 9 月发布后，iOS 设备上的滚动条便开始支持用户手指直接拖动，以及跳转列表。在 Flutter 程序中，如果屏幕上只有一个 ListView 等支持滚动的组件，则默认情况下 CupertinoScrollbar 组件也会自动支持手指拖动跳转功能，无须编写任何代码。

如果程序的某个页面使用了多个可滚动的列表类组件，并且需要支持拖动滚动条跳转的功能，则需要借助 controller(控制器)来指定 Scrollbar 与列表的对应关系，代码如下：

```
//第 5 章/scroll_bar_controller.dart

import 'package:flutter/material.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: MyHomePage(),
    );
  }
}

class MyHomePage extends StatefulWidget {
  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  //定义两个 ScrollController 滚动控制器
  ScrollController _controller1 = ScrollController();
  ScrollController _controller2 = ScrollController();

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Scrollbar Demo"),
      ),
```

```
        body: Column(
          mainAxisAlignment: MainAxisAlignment.spaceEvenly,
          children: [
            Container(
              height: 280,
              child: Scrollbar(                                  //第 1 个滚动条
                controller: _controller1,                        //使用第 1 个控制器
                child: ListView.builder(                         //第 1 个列表(普通样式)
                  controller: _controller1,                      //也使用第 1 个控制器
                  itemCount: 2000,
                  itemBuilder: (context, index) =>
                      Center(child: Text("列表 1 的第 ${index + 1}个元素")),
                ),
              ),
            ),
            Container(
              height: 280,
              child: Scrollbar(                                  //第 2 个滚动条
                controller: _controller2,                        //使用第 2 个控制器
                child: GridView.builder(                         //第 2 个列表(网格样式)
                  gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(
                      crossAxisCount: 4),
                  controller: _controller2,                      //使用第 2 个控制器
                  itemCount: 2000,
                  itemBuilder: (context, index) =>
                      Center(child: Text("网格的\n 第 ${index + 1}个元素")),
                ),
              ),
            ),
          ],
        ),
      );
    }
  }
```

运行效果如图 5-27 所示。

同一个页面很少会出现多个滚动列表，即使出现了通常也不需要支持手指拖动滚动条时发生跳转，因此实战中很少需要使用这里的 controller 属性。

2. 保持可见

如需使滚动条在列表没有发生滚动时也保持可见，则可以使用 isAlwaysShown(永远显示)属性，并将此属性设置为 true 即可。唯一需要注意的是，当启用该特性时，controller(控制器)不能为空，否则运行时会出现错误。

图 5-27　用控制器支持两个滚动条的拖动跳转

5.2.2　RefreshIndicator

RefreshIndicator(刷新指示器)组件可为大部分滚动列表添加“下拉刷新”的功能,但它目前只支持垂直方向滚动的列表。使用时只需要在滚动列表(如 ListView)组件的父级插入 RefreshIndicator 组件,并通过 onRefresh 参数传入刷新时的业务逻辑,代码如下:

```
RefreshIndicator(
  onRefresh: () async {
    await Future.delayed(Duration(seconds: 2));
  },
  child: GridView.count(
    crossAxisCount: 4,
    children: List.filled(50, Text("列表的一格")),
  ),
)
```

上述代码为 GridView 组件添加了刷新指示器，在用户下拉时出现，效果如图 5-28 所示。

图 5-28　下拉刷新的效果

当用户成功完成下拉刷新的连贯手势后，Flutter 会调用 onRefresh 参数传入的回传函数进行刷新操作。在该刷新函数的执行过程中，刷新指示器的滚动进度条会保持可见，直到该异步函数执行完毕后刷新指示器才会消失。

这里值得注意的是，只有当列表确实可被滚动时才有可能出现下拉刷新的效果。如在实战中发现某列表无法滚动，则可以考虑将该列表的 physics（滚动物理）属性设置为 AlwaysScrollableScrollPhysics（永远可以滚动），详情可参考 5.1.1 节所介绍的 ListView 组件的相关内容。

1. 自定义样式

RefreshIndicator 提供了 4 个用于自定义下拉刷新的指示器（滚动进度条）样式的属性。由于这部分内容相对比较简单，本书先依次介绍这些属性，最后一并举例。

1）color

颜色，指的是刷新进度条的前景颜色，默认为程序主题中的强调色，即 ThemeData.colorScheme.secondary 属性的颜色。若没有单独设置过，则 Flutter 程序默认为 #2196f3 淡蓝色。

例如，传入 color：Colors.red 可将刷新进度条改为红色。笔者测试时发现改变这里的 color 属性后，热更新（Hot Reload）无效，需要彻底重新启动 Flutter 程序才能观察到新赋的值。

2）backgroundColor

背景颜色，顾名思义，指的是刷新进度条的背景色，默认为程序主题中的画布背景色，即 ThemeData.colorScheme.secondary 属性的颜色。若没有单独设置过，则 Flutter 程序默认为 #fafafa 近白色。

3）displacement

位移，指的是刷新时进度条与列表顶部的位置关系，默认为 40.0 逻辑像素。这里需要注意的是，该属性定义的是用户松开手指触发刷新操作后，刷新的等待过程中的进度条的位置。在用户下拉的过程中实际产生的位移可超过这个数值。

4）strokeWidth

刷新图标的粗细，默认为 2.0 逻辑像素。例如可传入 strokeWidth：4.0 将其加粗。

下例通过上述 4 个属性，同时设置 RefreshIndicator 组件的颜色、背景色、位移和图标的粗细，代码及详细注释如下：

```
//第 5 章/refresh_indicator_styles.dart

RefreshIndicator(
  onRefresh: () async {
    await Future.delayed(Duration(seconds: 2));
  },
```

```
    color: Colors.white,                                    //颜色：白色
    backgroundColor: Colors.black,                          //背景色：黑色
    strokeWidth: 4.0,                                       //粗细：4 单位
    displacement: 20,                                       //位移：20 单位
    child: GridView.count(
      physics: AlwaysScrollableScrollPhysics(),
      crossAxisCount: 4,
      children: List.filled(10, Text("列表的一格")),
    ),
  )
```

运行效果如图 5-29 所示。

图 5-29　自定义下拉刷新的样式

2．实例

这里举一个利用 RefreshIndicator 组件实现为 ListView 列表添加内容的例子，完整代码如下：

```
//第 5 章/refresh_indicator_example.dart

import 'package:flutter/material.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      home: MyHomePage(),
    );
  }
}
```

```
class MyHomePage extends StatefulWidget {
  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  //列表的初始内容
  List<String> items = ["第 1 项", "第 2 项", "第 3 项"];

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("RefreshIndicator Demo"),
      ),
      body: RefreshIndicator(
        onRefresh: () async {
          //等待 2s,模拟网络延时
          await Future.delayed(Duration(seconds: 2));
          //添加新内容,并附加时间戳
          setState(() {
            items.add("新增内容: ${DateTime.now()}");
          });
        },
        child: ListView(
          //通过滚动列表将 items 的全部内容显示出来
          children: items.map((item) => Text("$item")).toList(),
        ),
      ),
    );
  }
}
```

程序刚开始运行时,ListView 列表中只有“第 1 项”“第 2 项”“第 3 项”这 3 个初始内容。当用户下拉触发刷新时,onRefresh 属性中的函数会被调用。刷新过程为先等待 2s,模拟网络延时,此时刷新进度条可见。2s 结束后添加新内容,附上时间戳,并利用 setState 使 Flutter 重绘。当 onRefresh 异步函数执行完毕后,刷新进度条会被自动隐藏,且新增内容会被显示到列表中。

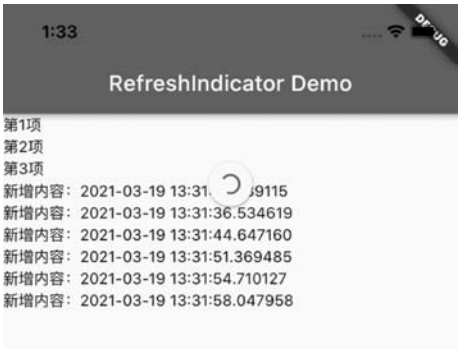

图 5-30 RefreshIndicator 实例运行效果

图 5-30 展示了本例运行时用户手动下拉刷新 6 次后,新增了 6 条带有时间戳的内容,并显示正在进行第 7 次刷新的效果。


(13min)

5.2.3 Dismissible

Dismissible 原意是“可被清除的”,因此这个 Flutter 组件主要用于帮助开发者实现看似复杂的“滑动清除”效果。例如,在电子邮箱管理软件中经常可以看到滑动即可删除某封电子邮件的功能。这个组件最常放在 ListView 之类的列表中,作为列表 children 的每个 Widget 的父级组件,为所有元素添加滑动清除功能,但 Dismissible 也可被用于其他任何接收 Widget 类型的场景。

使用时需要在可被清除的组件的父级插入 Dismissible,并传入一个 key(键)。如可在 ListView 组件的 itemBuilder 中返回 Dismissible 组件并利用 child 属性继续指定子组件,代码如下:

```
ListView.separated(
  itemCount: 20,
  separatorBuilder: (_, index) {
    return Divider();
  },
  itemBuilder: (_, index) {
    return Dismissible(                    //添加滑动清除功能
      key: ValueKey(index),                //传入 key 作为标识
      child: Container(
        height: 50,
        color: Colors.grey,
        alignment: Alignment.center,
        child: Text("这是第 ${index + 1}项"),
      ),
    );
  },
)
```

程序运行时可以得到一个包含 20 个元素的列表,并且每个元素都支持滑动清除。图 5-31 展示了当用户已经把第 3 项和第 4 项滑走,并正在滑动第 5 项时的效果。

1. 滑动时的背景

Dimissible 组件提供 background(背景)和 secondaryBackground(第二背景)属性,可用于设置向 2 个不同方向滑动的过程中的背景。若没有设置 secondaryBackground 属性,则无论用户向哪个方向滑动,都会采用 background 属性中的背景。这些参数支持 Widget 类型,因此开发者不仅可以修改背景颜色,还可以传入图标或文字等任意组件当作背景。

例如一款手机通讯录软件,从左向右滑动某位联系人可以打电话,从右向左滑动可以发短信。这样利用 background 和 secondaryBackground 属性,配合 Icon 组件,可在滑动时提供相应的图标以提示用户,代码如下:

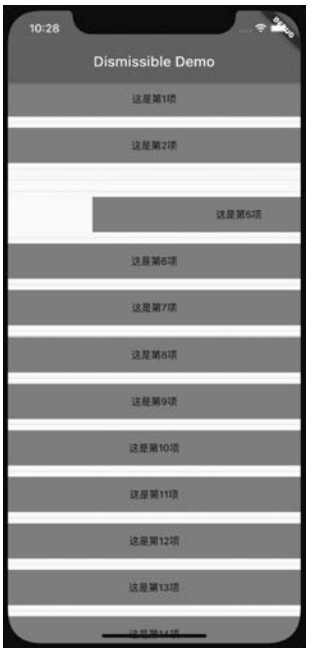

图 5-31 滑动清除的运行效果

```
//第 5 章/dismissible_example.dart

Dismissible(
  key: UniqueKey(),
  background: Container(
    padding: EdgeInsets.all(16),
    color: Colors.black,
    alignment: Alignment.centerLeft,
    child: Icon(
      Icons.phone,
      color: Colors.white,
    ),
  ),
  secondaryBackground: Container(
    padding: EdgeInsets.all(16),
    color: Colors.grey,
    alignment: Alignment.centerRight,
    child: Icon(Icons.sms),
  ),
  child: Container(
    height: 56,
    alignment: Alignment.center,
    child: Text("这是第${index + 1}项"),
  ),
)
```

当用户向左或向右滑动时可观察到不同的背景，如图 5-32 所示。

图 5-32　左划和右划时展示不同背景

2. 滑动行为

1）direction

方向属性用于设置 Dismissible 支持的滑动方向，需接收一个 DismissDirection 枚举类，有 6 种值，分别是 horizontal（水平方向）、vertical（垂直方向）、startToEnd(从起始到末尾)、endToStart(从末尾到起始)、up(上)和 down(下)。其中，前 2 种值表示支持水平或垂直维度的任意方向，例如设置 vertical 就表示同时支持上滑和下滑。后 4 种值表示唯一方向，例如设置 startToEnd 即表示只支持顺着阅读方向(在汉语或英语设备上即从左到右)滑动。

2）dismissThresholds

清除阈值，用于定义当用户的滑动手势完成到什么程度时可以视为成功的清除操作，低于该程度的滑动不会触发清除效果。默认为 0.4，即用户至少滑动 40%的位置后松手，该组件会自动帮其完成剩下的滑动动画，并顺利清除该元素。若用户滑动不足 40%时提前松手，则视为取消操作。

实战中很少需要改动这里的默认 40%阈值。如需改动，则可以传入一个 Map 数据类型，说明每种支持的滑动方向对应的有效阈值，例如可将从左向右滑动的阈值改为 20%，而从右向左滑动的阈值改为 99%难以滑动，代码如下：

```
dismissThresholds: {
  DismissDirection.startToEnd: 0.20,
  DismissDirection.endToStart: 0.99,
},
```

当用户手指较快速地做出“甩出”动作时，这里会收到一个非常接近但不足 1.0 的数值，因此当设置阈值 99%时，几乎只有此类大幅甩出动作才可能超过阈值，而当阈值被设置为大于或等于 1.0 时，用户将无法通过手势完成滑动清除的操作。

3）crossAxisEndOffset

交叉轴位移，用于定义当组件在滑动时向另一个维度的位移情况，默认为 0，不会发生交叉轴位移。例如取值 2.5 即为位移 2.5 倍于组件尺寸的距离，而取值－1.0 时则向反方向位移 1.0 倍。由于作用轴为交叉轴，在水平滑动的 Dismissible 中具体表现为 child 向上或向下飘走，如图 5-33 所示。

4）movementDuration 和 resizeDuration

移动时长是指当用户滑动超过阈值后松开手指，Flutter 自动帮其完成操作(补滑)的动画时长，或当用户滑动不足阈值时提前松开手指，Flutter 撤销该滑动操作，并将元素缓缓放

回原位的时长。

实际运行时，Dismissible 内部的动画控制器会处理整个动画，因此这里的时长是指从0%滑动到100%的总时长，开发者不必担心用户具体是在什么阶段松手，child 补滑的速度是恒定的。

缩放时长则是指当用户成功触发滑动清除手势，并且 Dismissible 已经将其补滑到位后，原组件逐渐变小并最终消失的动画时长。例如设置 resizeDuration：Duration(seconds：10)表示将时长改为 10s 的超慢速动画后，可以清晰地观察到元素逐渐消失的过程，如图 5-34 所示。

图 5-33 水平滑动时 child 向上方位移的效果

图 5-34 被清除的元素正在逐渐消失

3. 滑动事件

1）onResize 和 onDismissed

当用户完成滑动手势并成功触发清除操作，并且元素移动的动画播放完毕后，子组件将开始进行第二部分动画，逐渐缩小并最终消失。在尺寸缩小的动画进行过程中，Flutter 会反复多次调用 onResize 回传函数。

当移动和缩放动画均播放完毕后，Flutter 会调用一次 onDismissed 回传函数，并将滑动的方向提供给开发者，以便完成相应的业务逻辑。例如可将它们打印出来，代码如下：

```
onResize: () => print("resizing"),
onDismissed: (direction) => print(direction),
```

用户完成滑动操作的手势后，清除操作开始进行。此时终端会出现大量的 resizing 输出，并等到该组件最终完成缩放动画并消失后，终端会显示一行例如 DismissDirection.startToEnd 的字样。

2）confirmDismiss

当用户完成滑动手势后，在真正处理 onDismissed 事件之前，Flutter 还会调用 confirmDismiss 参数中的回传函数确认是否需要继续操作。如果函数的最终返回值为 true，则会继续清除操作。若函数返回值为 false，则会撤销操作，并开始逆向播放动画，将被清除的组件重新移动回原地。

这里用 AlertDialog 组件举例，弹出对话框后请用户确认是否删除，代码如下：

```
//第 5 章/dismissible_confirm.dart

import 'package:flutter/material.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: MyHomePage(),
    );
  }
}

class MyHomePage extends StatefulWidget {
  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Dismissible Demo"),
      ),
      body: ListView.separated(
        itemCount: 20,
        separatorBuilder: (_, __) => Divider(),
        itemBuilder: (_, index) {
          return Dismissible(
            key: UniqueKey(),
            confirmDismiss: (DismissDirection direction) async {
              return await showDialog(
                context: context,
                builder: (BuildContext context) {
                  return AlertDialog(
                    title: Text("确认"),
                    content: Text("确认要删除这一项吗?"),
                    actions: <Widget>[
                      TextButton(
```

```
                        onPressed: () => Navigator.of(context).pop(false),
                        child: Text("取消"),
                      ),
                      TextButton(
                          onPressed: () => Navigator.of(context).pop(true),
                          child: Text(
                            "删除",
                            style: TextStyle(color: Colors.red),
                          )),
                    ],
                  );
                },
              );
            },
            background: Container(
              padding: EdgeInsets.all(16),
              color: Colors.black,
              alignment: Alignment.centerLeft,
              child: Icon(
                Icons.delete_outline,
                color: Colors.white,
              ),
            ),
            child: Container(
              height: 56,
              alignment: Alignment.center,
              child: Text("这是第${index + 1}项"),
            ),
          );
        },
      ),
    );
  }
}
```

运行效果如图 5-35 所示。

对 AlertDialog 组件不熟悉的读者可翻阅第 9 章“悬浮与弹窗”中的相关内容与介绍。

5.2.4 ScrollConfiguration

如果需要改变一部分或全部列表的默认样式，则可以使用 ScrollConfiguration 组件。这种思路与同时设置所有子 Text 组件的 DefaultTextStyle 组件，或同时设置所有子 Icon 组件的 IconTheme 组件类似，所有列表类组件的默认样式是由最近上级的 ScrollConfiguration 组件提供的，因此，若需要全局设置整个应用程序的所有列表默认样式，则可将 ScrollConfiguration 组件插入接近组件树根部的位置。若只需设置某个列表的

图 5-35　滑动清除之前先弹出用户确认对话框

默认样式，则应把 ScrollConfiguration 组件直接插入该列表组件的父级，从而避免干扰到其他的列表。

使用时开发者必须通过 behavior 参数传入一个 ScrollBehavior 类的值，例如可传入 ScrollBehavior()，使用默认样式，代码如下：

```
ScrollConfiguration(
  behavior: ScrollBehavior(),
  child: ListView(
    children: [
      Text("1"),
      Text("2"),
    ],
  ),
)
```

1. ScrollBehavior

一般实战中开发者会创建一个新的继承 ScrollBehavior 的类，以实现自定义样式。继

承时，一般会重写 buildViewportChrome 和 getScrollPhysics 方法。

1）buildViewportChrome

这是用于在列表组件外部添加修饰的属性，若不想添加任何修饰，则可以直接回传 child 本身。默认情况下，Flutter 会为 Android 和 Fuchsia 这两个操作系统添加滚动过量时的波形色块效果，为其他系统（包括 iOS、Linux、Windows 和 macOS）不添加任何修饰。

2）getScrollPhysics

滚动物理，默认情况下在 iOS 和 macOS 这两个操作系统上呈现过量滚动后自动弹回的动画效果，而在其他操作系统（包括 Android、Fuchsia、Linux 和 Windows）则直接卡住，触碰到列表的边缘后不能过量滚动。

2. 实例

这里举个例子，不判断当前操作系统，当任何设备的列表滚动至边缘时均不能过量滚动，并且出现类似安卓的波形色块效果，但颜色改为灰色。为了实现这个效果，这里使用 ScrollConfiguration 组件，传入自制的 MyScrollBehavior 类，继承自 ScrollBehavior，并重写上述两种方法。

在 buildViewportChrome 方法中，这里在列表外部添加修饰过量滚动时的色块修饰，并定义为灰色。在 getScrollPhysics 方法中，则直接将滚动物理设置为 ClampingScrollPhysics，完整代码如下：

```
//第 5 章/scroll_configuration_example.dart

import 'package:flutter/material.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: MyHomePage(),
    );
  }
}

class MyHomePage extends StatefulWidget {
  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  @override
```

```
    Widget build(BuildContext context) {
      return Scaffold(
        appBar: AppBar(
          title: Text("ScrollConfiguration Demo"),
        ),
        body: ScrollConfiguration(
          behavior: MyScrollBehavior(),
          child: ListView.separated(
            itemCount: 20,
            separatorBuilder: (_, __) => Divider(),
            itemBuilder: (_, index) {
              return Container(
                height: 56,
                alignment: Alignment.center,
                child: Text("这是第${index + 1}项"),
              );
            },
          ),
        ),
      );
    }
  }

  class MyScrollBehavior extends ScrollBehavior {
    @override
    Widget buildViewportChrome(context, child, AxisDirection axisDirection) {
      return GlowingOverscrollIndicator(
        child: child,
        axisDirection: axisDirection,
        color: Colors.grey,
      );
    }

    @override
    ScrollPhysics getScrollPhysics(BuildContext context) {
      return ClampingScrollPhysics(parent: RangeMaintainingScrollPhysics());
    }
  }
```

运行效果如图 5-36 所示。

5.2.5 NotificationListener

滚动类的列表组件如 ListView 或 GridView 等，在滚动的过程中会产生滚动通知事件。这类通知事件会沿着组件树向上冒泡(Bubble Up)，直到被某个监听该通知事件的组件拦截为止。

本章之前介绍的滚动条 Scrollbar 组件和下拉刷新 RefreshIndicator 组件就是通过监听

图 5-36　使用 ScrollConfiguration 设置子列表组件的样式

滚动通知事件获知滚动列表当前状态，以达到正确显示滚动进度或在恰当的时机触发刷新操作等功能，因此当使用它们时，只需简单地将 Scrollbar 或 RefreshIndicator 插入滚动列表组件的上级，并不需要编写过多的额外代码，它们就能自动获取列表的状态，非常方便。

开发者也可以使用 NotificationListener 组件直接监听此类通知事件。当下级组件发出事件时，onNotification 回传函数会被调用，同时该函数的返回值（布尔类型）决定了是否拦截事件，被拦截后的事件将不再继续向上级冒泡，代码如下：

```
NotificationListener(
  onNotification: (Notification notification) {
    print(notification);                    //输出通知内容
    return false;                           //不拦截(通知将继续冒泡)
  },
  child: ListView.builder(
    itemBuilder: (_, index) => Text("$index"),
  ),
)
```

在一个简单的 ListView 列表的父级插入 NotificationListener 组件后，一旦列表开始滚动，就可以在命令行观察到一系列事件。例如产生以下输出内容：

```
I/flutter (22142): ScrollStartNotification(depth: 0 (local), FixedScrollMetrics(669.6..
[657.5]..Infinity), DragStartDetails(Offset(165.8, 469.4)))
I/flutter (22142): UserScrollNotification(depth: 0 (local), FixedScrollMetrics(669.6..
[657.5]..Infinity), direction: ScrollDirection.reverse)
I/flutter (22142): ScrollUpdateNotification(depth: 0 (local), FixedScrollMetrics(670.8..[657.
5]..Infinity), scrollDelta: 1.1026278409090082, DragUpdateDetails(Offset(0.0, -1.1)))
I/flutter (22142): ScrollUpdateNotification(depth: 0 (local), FixedScrollMetrics(673.3..[657.
5]..Infinity), scrollDelta: 1.1026278409090082, DragUpdateDetails(Offset(0.0, -1.1)))
I/flutter (22142): ScrollUpdateNotification(depth: 0 (local), FixedScrollMetrics(691.8..[657.
5]..Infinity), scrollDelta: 1.1026278409090082, DragUpdateDetails(Offset(0.0, -1.1)))
I/flutter (22142): ScrollUpdateNotification(depth: 0 (local), FixedScrollMetrics(696.6..[657.
5]..Infinity), scrollDelta: 1.1026278409090082, DragUpdateDetails(Offset(0.0, -1.1)))
I/flutter (22142): ScrollEndNotification(depth: 0 (local), FixedScrollMetrics(696.6..[657.5]..
Infinity), DragEndDetails(Velocity(0.0, 0.0)))
I/flutter (22142): UserScrollNotification(depth: 0 (local), FixedScrollMetrics(696.6..[657.5]..
Infinity), direction: ScrollDirection.idle)
```

这里包括了 ScrollStartNotification(滚动开始通知)、UserScrollNotification(用户滚动通知,通常在用户改变滚动方向时触发)、ScrollUpdateNotification(滚动更新通知)、ScrollEndNotification(滚动终止通知)等。这些通知内含具体的事件细节,如滚动更新通知包括滚动了多少逻辑像素等信息。若安卓用户在列表触边后继续滚动,则还会触发 OverscrollNotification(过度滚动通知),表示列表已无法再继续滚动。在 iOS 系统上触边的列表会继续滚动,并在用户松开手指后弹回,因此不会触发这个通知。

1. 通知拦截

在 NotificationListener 的 onNotification 回传函数运行结束时,开发者可以选择回传一个布尔值,表明该通知事件是否有必要继续向上级冒泡。一般情况下,当需要处理的业务逻辑已经被处理完毕后可以选择回传 true 拦截该通知,阻止其继续通知上级的组件,以节约不必要的性能开支,但这么做时需注意确保父级没有依赖该通知的组件。

例如当 NotificationListener 组件选择拦截通知时,其父级的 Scrollbar 组件将无法获得滚动通知,因此无法显示列表的滚动进度。当 NotificationListener 不拦截通知时,Scrollbar 就可以收到通知,并利用通知内容,正确地显示滚动条,代码如下:

```
Scrollbar(
  child: NotificationListener(
    //拦截通知; 改为 false 后滚动条恢复正常
    onNotification: (_) => true,
    child: ListView.builder(
      itemCount: 200,
      itemBuilder: (_, index) => Text("$index"),
    ),
  ),
)
```

2. 自定义通知事件

在 Flutter 中，滚动列表在滚动时发出的通知事件只是众多通知事件之一。其他 Flutter 框架自带的通知事件还有 KeepAliveNotification、LayoutChangedNotification、OverscrollIndicatorNotification 等。除此之外开发者还可以通过继承 Notification 类，自定义通知事件。例如可定义 MyNotification 类，并支持在通知事件内部存储一个 dynamic 类型（支持任意数据类型）的细节信息，代码如下：

```
class MyNotification extends Notification {
  //自定义通知内部变量,用于存储通知细节信息
  final dynamic details;

  MyNotification(this.details);
}
```

当需要发送通知事件时，可通过调用 Notification 类的 dispatch 方法触发通知。例如当用户单击按钮时发出通知，并将 Colors.green 作为通知细节，传给该自定义通知的构造函数，代码如下：

```
ElevatedButton(
  child: Text("发送绿色通知"),
  onPressed: () => MyNotification(Colors.green).dispatch(context),
)
```

接着，若在组件树的上级插入 NotificationListener 组件，即可在 MyNotification 被用户触发时收到该通知。由于 NotificationListener 还可能会收到其他组件发出的其他通知，因此这里最好先判断通知类型是否为 MyNotification 类，以避免受到其他无关通知的干扰，代码如下：

```
NotificationListener(
  onNotification: (notification) {
    //判断通知是否为自定义的 MyNotification 类型
    if (notification is MyNotification) {
      //如果是,则打印出自定义通知中的细节内容
      print(notification.details);
      return true;                  //拦截该通知,不再冒泡
    }
    return false;                   //不拦截其他类型的通知
  },
  child: ...
)
```

本例的完整源代码如下：

```
//第 5 章/notification_listener.dart

import 'package:flutter/material.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(home: MyHomePage());
  }
}

class MyHomePage extends StatefulWidget {
  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Notification Demo")),
      body: NotificationListener(
        //监听通知
        onNotification: (notification) {
          //判断通知是否为自定义的 MyNotification 类型
          if (notification is MyNotification) {
            //打印出自定义通知中的细节内容
            print(notification.details);
            return true;                    //拦截,不再冒泡
          }
          return false;                     //不拦截其他类型的通知
        },
        child: Sender(),
      ),
    );
  }
}

class Sender extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
return Wrap(
  spacing: 20,
      children: [
        ElevatedButton(
```

```
            child: Text("发送字符串通知"),
            onPressed: () => MyNotification("hello world").dispatch(context),
          ),
          ElevatedButton(
            child: Text("发送颜色通知"),
            onPressed: () => MyNotification(Colors.blue).dispatch(context),
          ),
        ],
      );
    }
  }

  class MyNotification extends Notification {
    //自定义通知内部变量,用于存储通知细节信息,类型为 dynamic,即支持任意类型
    final dynamic details;

    MyNotification(this.details);
  }
```

程序运行效果如图 5-37 所示。

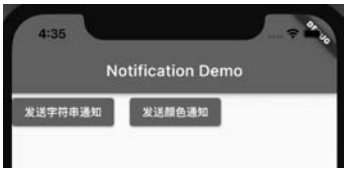

图 5-37　自定义通知的发送按钮运行效果

当用户依次单击 2 个按钮后,即可观察到程序输出的结果如下:

```
flutter: hello world
flutter: MaterialColor(primary value: Color(0xff2196f3))
```

这里需要注意的是,NotificationListener 只会监听子级(children)和其他下级(descendants)组件发出的通知事件,而无法监听父级(parents)和其他上级(ancestors)组件,也不会监听本身(或同级)发出的通知事件,因此在这个例子中,NotificationListener 的 child 属性传入的是自定义的 Sender 组件,而不是直接嵌套 Column 完成组件构造,以确保发送通知事件的是子组件而不是本身。

实战中除了可以将组件单独分离外,还可以通过 Builder 实现从属关系,直接在原地完成“匿名子组件”的构造,对此不熟悉的读者可参考本书第 9 章的 Flutter 框架小知识“什么时候需要使用 Builder 组件”。

5.2.6　SingleChildScrollView

本章在开头提到,数据显示通常是大部分应用程序界面的主要环节,之后本书也花费大量篇幅详细地介绍了 ListView、GridView 及其他支持动态加载元素的滚动列表类组件,但在实战

中，屏幕滚动的作用绝非仅限于高效地将成千上万条元素呈现给用户。例如有时只是担心较小屏幕的设备可能会显示不下某个用户界面而已。例如偏好设置页面，若选项较多时也应加上滚动效果，但可能选项也不会多到需要动态加载。这时也可以选用 SingleChildScrollView 组件，方便地为任何组件（尤其是 Column 组件）添加滚动功能，代码如下：

```
SingleChildScrollView(
  child: Column(
    mainAxisAlignment: MainAxisAlignment.spaceEvenly,
    children: [
        Container(height: 250, color: Colors.grey[200]),
        Container(height: 250, color: Colors.grey[400]),
        Container(height: 250, color: Colors.grey[600]),
        Container(height: 250, color: Colors.grey[800]),
    ],
  ),
)
```

这里利用 Column 组件垂直排列了 4 个高度为 250 单位的 Container 组件，总高度为 1000 单位。程序运行后，当可用高度不足 1000 逻辑像素时，Column 组件理应出现如图 5-38（左图）所示的溢出，但由于其父级插入了 SingleChildScrollView 组件，这里不会溢出，如图 5-38（右图）所示，并会自动开始允许用户滚动屏幕。

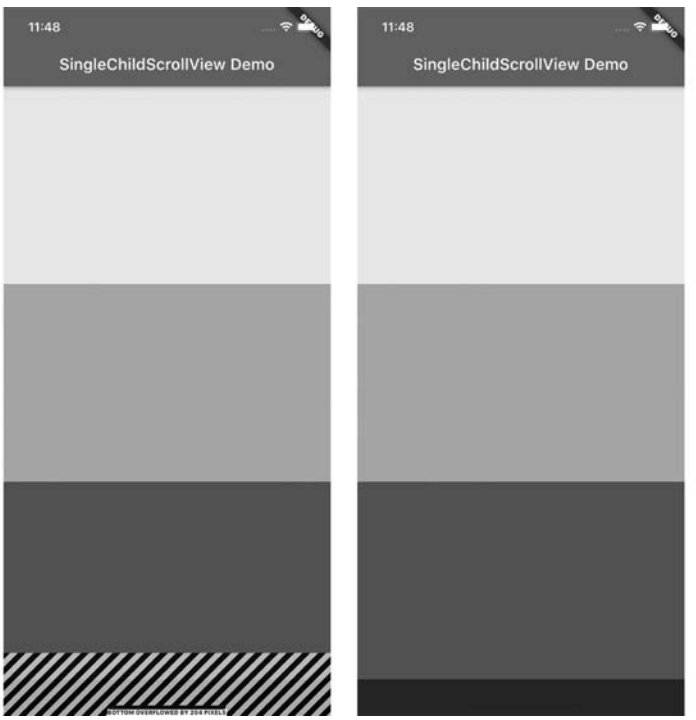

图 5-38　界面溢出时和允许滚动时的对比

但若本例中的 Column 原本就不足 4 个 Container 组件，不足以导致溢出，则即使插入了 SingleChildScrollView 也完全不会允许滚动。这与 ListView 的效果不同，读者不妨亲自动手试试。

1. 滚动少量 UI 元素

开发者难免会担心当屏幕尺寸不足时可能会导致一部分界面无法显示的情况，而屏幕尺寸不足的原因是多种多样的，例如可能是程序运行时的设备屏幕本身太小，或是用户将手机设备横屏使用，或安卓用户可能同时打开了多个程序分屏显示等，甚至是网页版的用户将浏览器窗口调整得太小。

SingleChildScrollView 组件就是为了解决这种大部分情况下一屏可以显示，但遇到特殊情况偶尔显示不下的问题。使用 SingleChildScrollView 会放弃一切动态加载的行为，因此它只适用于元素较少的滚动场景。

这里通过一个简单的登录页面举例，代码如下：

```
SingleChildScrollView(
  child: Column(
    children: [
      FlutterLogo(size: 400),
      Text("欢迎来到 Flutter 登录页面"),
      TextField(decoration: InputDecoration(labelText: "用户名")),
      TextField(
        decoration: InputDecoration(labelText: "密码"),
        obscureText: true,
      ),
      SizedBox(height: 32),
      ElevatedButton(
        child: Text("登录"),
        onPressed: () {},
      )
    ],
  ),
)
```

该页面在大部分情况下可在一屏内显示完，不需要滚动，程序运行效果如图 5-39 所示。

但当用户将设备横屏显示时，若不添加 SingleChildScrollView 开启滚动，则只会显示该页面上半部分内容，也就是巨大的徽标图案，而重要的输入框和按钮会超出屏幕的边界。添加 SingleChildScrollView 组件后，用户可以自由将页面滚动至页面下半部分，效果如图 5-40 所示。

图 5-39　登录页面竖屏显示效果

图 5-40　登录页面横屏后允许滚动的效果

这里值得一提的是，在 Column 的父级插入 SingleChildScrollView 会迫使 Column 进行真空包装（类似对 Column 传入了 mainAxisSize：MainAxisSize. min 参数），因此 Column 内的部分属性（如主轴尺寸和主轴方向的留白等）无法生效。如需要在主轴方向添加留白，则可考虑直接插入 SizedBox 组件固定留白，或考虑使用 LayoutBuild 或 MediaQuery 组件，根据屏幕尺寸动态计算留白尺寸。对这些组件不熟悉的读者可参考本书第 6 章“进阶布局”中关于尺寸与测量的相关内容。

2. 其他属性

SingleChildScrollView 还有一部分与 ListView 组件相同或相似的属性，它们分别是 controller（滚动控制器）、reverse（倒序）、padding（内部留白）、physics（滚动物理）及 scrollDirection（滚动方向）。其中滚动方向默认为垂直方向，使 SingleChildScrollView 适合在 Column 组件的父级位置插入，而修改为 Axis. horizontal 后则为水平方向滚动，适合作为 Row 组件的父级。其他属性的名称和用法均与 ListView 组件的同名属性一致，读者可查阅本章 ListView 小节的内容。

最后值得一提的是，当 SingleChildScrollView 确实有必要滚动时，也会发出滚动通知，因此 Scrollbar 等组件也可照常使用。