本章目录

可滚动Widget简介

当内容超过显示视口(ViewPort)时，如果没有特殊处理，Flutter则会提示Overflow错误。为此，Flutter提供了多种可滚动widget（Scrollable Widget）用于显示列表和长布局。在本章中，我们先介绍一下常用的可滚动widget（如ListView、GridView等），然后介绍一下Scrollable与可滚动widget的原理。可滚动Widget都直接或间接包含一个Scrollable widget，因此它们包括一些共同的属性，为了避免重复介绍，我们在此统一介绍一下：

```
Scrollable({
  ...
  this.axisDirection = AxisDirection.down,
  this.controller,
  this.physics,
  @required this.viewportBuilder, //后面介绍
})
```

- axisDirection：滚动方向。
- physics：此属性接受一个ScrollPhysics对象，它决定可滚动Widget如何响应用户操作，比如用户滑动完抬起手指后，继续执行动画；或者滑动到边界时，如何显示。默认情况下，Flutter会根据具体平台分别使用不同的ScrollPhysics对象，应用不同的显示效果，如当滑动到边界时，继续拖动的话，在iOS上会出现弹性效果，而在Android上会出现微光效果。如果你想在所有平台下使用同一种效果，可以显式指定，Flutter SDK中包含了两个ScrollPhysics的子类可以直接使用：
  - ClampingScrollPhysics：Android下微光效果。
  - BouncingScrollPhysics：iOS下弹性效果。
- controller：此属性接受一个ScrollController对象。ScrollController的主要作用是控制滚动位置和监听滚动事件。默认情况下，widget树中会有一个默认的PrimaryScrollController，如果子树中的可滚动widget没有显式的指定`controller`并且`primary`属性值为`true`时（默认就为`true`），可滚动widget会使用这个默认的PrimaryScrollController，这种机制带来的好处是父widget可以控制子树中可滚动widget的滚动，例如，Scaffold使用这种机制在iOS中实现了"回到顶部"的手势。我们将在本章后面“滚动控制”一节详细介绍ScrollController。

Scrollbar

Scrollbar是一个Material风格的滚动指示器（滚动条），如果要给可滚动widget添加滚动条，只需将Scrollbar作为可滚动widget的父widget即可，如：

```
Scrollbar(
  child: SingleChildScrollView(
    ...
  ),
);
```

Scrollbar和CupertinoScrollbar都是通过ScrollController来监听滚动事件来确定滚动条位置，关于ScrollController详细的内容我们将在后面专门一节介绍。

CupertinoScrollbar

CupertinoScrollbar是iOS风格的滚动条，如果你使用的是Scrollbar，那么在iOS平台它会自动切换为CupertinoScrollbar。

ViewPort视口

在很多布局系统中都有ViewPort的概念，在Flutter中，术语ViewPort（视口），如无特别说明，则是指一个Widget的实际显示区域。例如，一个ListView的显示区域高度是800像素，虽然其列表项总高度可能远远超过800像素，但是其ViewPort仍然是800像素。

主轴和纵轴

在可滚动widget的坐标描述中，通常将滚动方向称为主轴，非滚动方向称为纵轴。由于可滚动widget的默认方向一般都是沿垂直方向，所以默认情况下主轴就是指垂直方向，水平方向同理。

SingleChildScrollView

SingleChildScrollView类似于Android中的ScrollView，它只能接收一个子Widget。定义如下：

```
SingleChildScrollView({
  this.scrollDirection = Axis.vertical, //滚动方向，默认是垂直方向
  this.reverse = false,
  this.padding,
  bool primary,
  this.physics,
  this.controller,
  this.child,
})
```

除了通用属性，我们重点看一下`reverse`和`primary`两个属性：

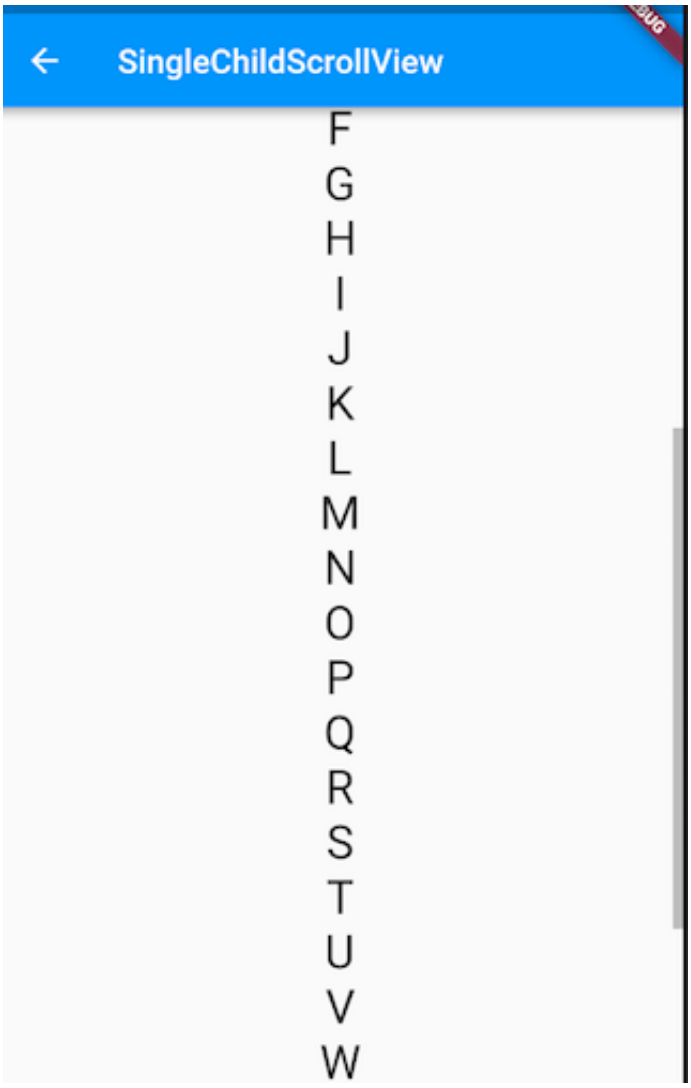
- reverse：该属性API文档解释是：是否按照阅读方向相反的方向滑动，如：`scrollDirection`值为`Axis.horizontal`，如果阅读方向是从左到右(取决于语言环境，阿拉伯语就是从右到左)，reverse为`true`时，那么滑动方向就是从右往左。其实此属性本质上是决定可滚动widget的初始滚动位置是在“头”还是“尾”，取`false`时，初始滚动位置在“头”，反之则在“尾”，读者可以自己试验。
- primary：指是否使用widget树中默认的PrimaryScrollController；当滑动方向为垂直方向（`scrollDirection`值为`Axis.vertical`）并且`controller`没有指定时，`primary`默认为`true`.

示例

下面是一个将大写字母A-Z沿垂直方向显示的例子，由于垂直方向空间不够，所以使用SingleChildScrollView。：

```
class SingleChildScrollViewTestRoute extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    String str = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    return Scrollbar(
      child: SingleChildScrollView(
        padding: EdgeInsets.all(16.0),
        child: Center(
          child: Column(
            //动态创建一个List<Widget>
            children: str.split("")
                //每一个字母都用一个Text显示,字体为原来的两倍
                .map((c) => Text(c, textScaleFactor: 2.0,))
                .toList(),
          ),
        ),
      ),
    );
  }
}
```

效果：



ListView

ListView是最常用的可滚动widget，它可以沿一个方向线性排布所有子widget。我们看看ListView的默认构造函数定义：

```
ListView({
  ...
  //可滚动widget公共参数
  Axis scrollDirection = Axis.vertical,
  bool reverse = false,
  ScrollController controller,
  bool primary,
  ScrollPhysics physics,
  EdgeInsetsGeometry padding,

  //ListView各个构造函数的共同参数
  double itemExtent,
  bool shrinkWrap = false,
  bool addAutomaticKeepAlives = true,
  bool addRepaintBoundaries = true,
  double cacheExtent,

  //子widget列表
  List<Widget> children = const <Widget>[],
})
```

上面参数分为两组：第一组是可滚动widget公共参数，前面已经介绍过，不再赘述；第二组是ListView各个构造函数（ListView有多个构造函数）的共同参数，我们重点来看看这些参数，：

- itemExtent：该参数如果不为null，则会强制children的"长度"为itemExtent的值；这里的"长度"是指滚动方向上子widget的长度，即如果滚动方向是垂直方向，则itemExtent代表子widget的高度，如果滚动方向为水平方向，则itemExtent代表子widget的长度。在ListView中，指定itemExtent比让子widget自己决定自身长度会更高效，这是因为指定itemExtent后，滚动系统可以提前知道列表的长度，而不是总是动态去计算，尤其是在滚动位置频繁变化时（滚动系统需要频繁去计算列表高度）。
- shrinkWrap：该属性表示是否根据子widget的总长度来设置ListView的长度，默认值为`false`。默认情况下，ListView的会在滚动方向尽可能多的占用空间。当ListView在一个无边界(滚动方向上)的容器中时，shrinkWrap必须为`true`。
- addAutomaticKeepAlives：该属性表示是否将列表项（子widget）包裹在AutomaticKeepAlive widget中；典型地，在一个懒加载列表中，如果将列表项包裹在AutomaticKeepAlive中，在该列表项滑出视口时该列表项不会被GC，它会使用KeepAliveNotification来保存其状态。如果列表项自己维护其KeepAlive状态，那么此参数必须置为`false`。
- addRepaintBoundaries：该属性表示是否将列表项（子widget）包裹在RepaintBoundary中。当可滚动widget滚动时，将列表项包裹在RepaintBoundary中可以避免列表项重绘，但是当列表项重绘的开销非常小（如一个颜色块，或者一个较短的文本）时，不添加RepaintBoundary反而会更高效。和addAutomaticKeepAlive一样，如果列表项自己维护其KeepAlive状态，那么此参数必须置为`false`。

> 注意：上面这些参数并非ListView特有，在本章后面介绍的其它可滚动widget也可能会拥有这些参数，它们的含义是相同的。

默认构造函数

默认构造函数有一个`children`参数，它接受一个Widget列表（List<Widget>）。这种方式适合只有少量的子widget的情况，因为这种方式需要将所有`children`都提前创建好（这需要做大量工作），而不是等到子widget真正显示的时候再创建。实际上通过此方式创建的ListView和使用SingleChildScrollView+Column的方式没有本质的区别。下面是一个例子：

```
ListView(
  shrinkWrap: true,
  padding: const EdgeInsets.all(20.0),
  children: <Widget>[
    const Text('I\'m dedicating every day to you'),
    const Text('Domestic life was never quite my style'),
    const Text('When you smile, you knock me out, I fall apart'),
    const Text('And I thought I was so smart'),
  ],
);
```

> 注意：可滚动widget通过一个List<Widget>来作为其children属性时，只适用于子widget较少的情况，这是一个通用规律，并非ListView自己的特性，像GridView也是如此。

ListView.builder

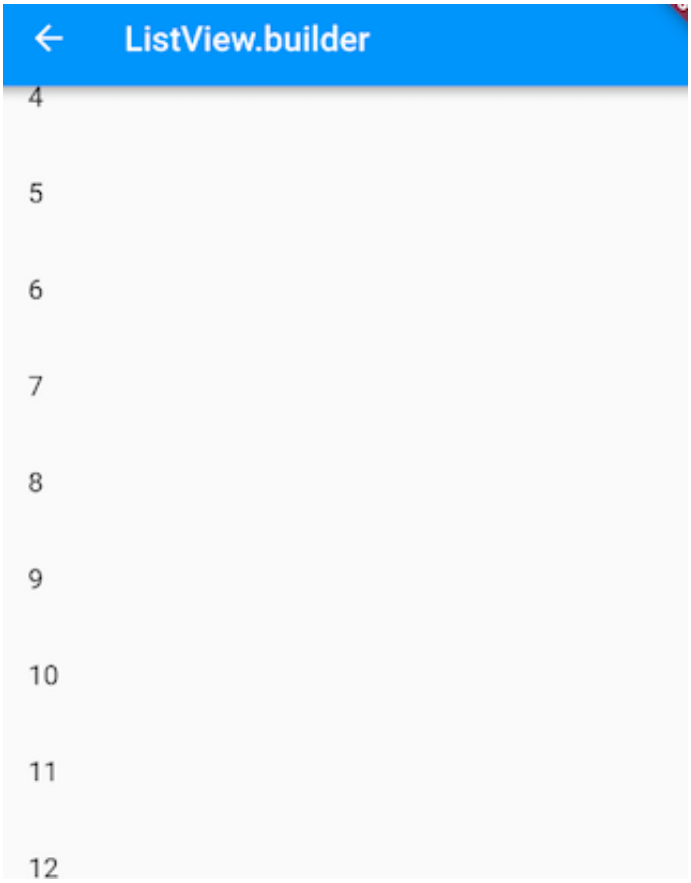
`ListView.builder`适合列表项比较多（或者无限）的情况，因为只有当子Widget真正显示的时候才会被创建。下面看一下ListView.builder的核心参数列表：

```
ListView.builder({
  // ListView公共参数已省略
  ...
  @required IndexedWidgetBuilder itemBuilder,
  int itemCount,
  ...
})
```

- itemBuilder：它是列表项的构建器，类型为IndexedWidgetBuilder，返回值为一个widget。当列表滚动到具体的index位置时，会调用该构建器构建列表项。
- itemCount：列表项的数量，如果为null，则为无限列表。

看一个例子：

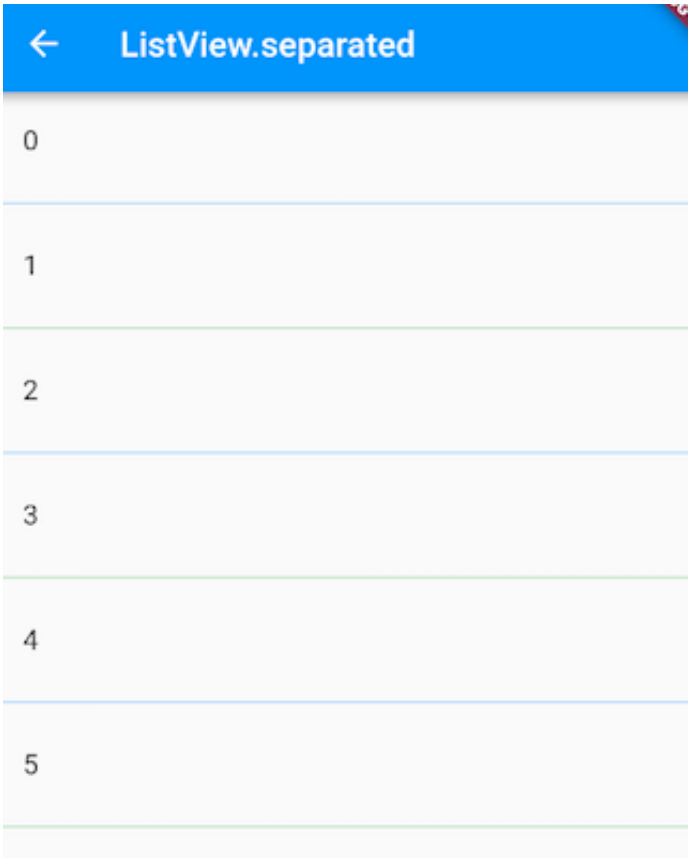
```
ListView.builder(
    itemCount: 100,
    itemExtent: 50.0, //强制高度为50.0
    itemBuilder: (BuildContext context, int index) {
      return ListTile(title: Text("$index"));
    }
);
```



ListView.separated

`ListView.separated`可以生成列表项之间的分割器，它除了比`ListView.builder`多了一个`separatorBuilder`参数，该参数是一个分割器生成器。下面我们看一个例子：奇数行添加一条蓝色下划线，偶数行添加一条绿色下划线。

```dart
class ListView3 extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    //下划线widget预定义以供复用。
    Widget divider1=Divider(color: Colors.blue,);
    Widget divider2=Divider(color: Colors.green);
    return ListView.separated(
        itemCount: 100,
        //列表项构造器
        itemBuilder: (BuildContext context, int index) {
          return ListTile(title: Text("$index"));
        },
        //分割器构造器
        separatorBuilder: (BuildContext context, int index) {
          return index%2==0?divider1:divider2;
        },
    );
  }
}
```

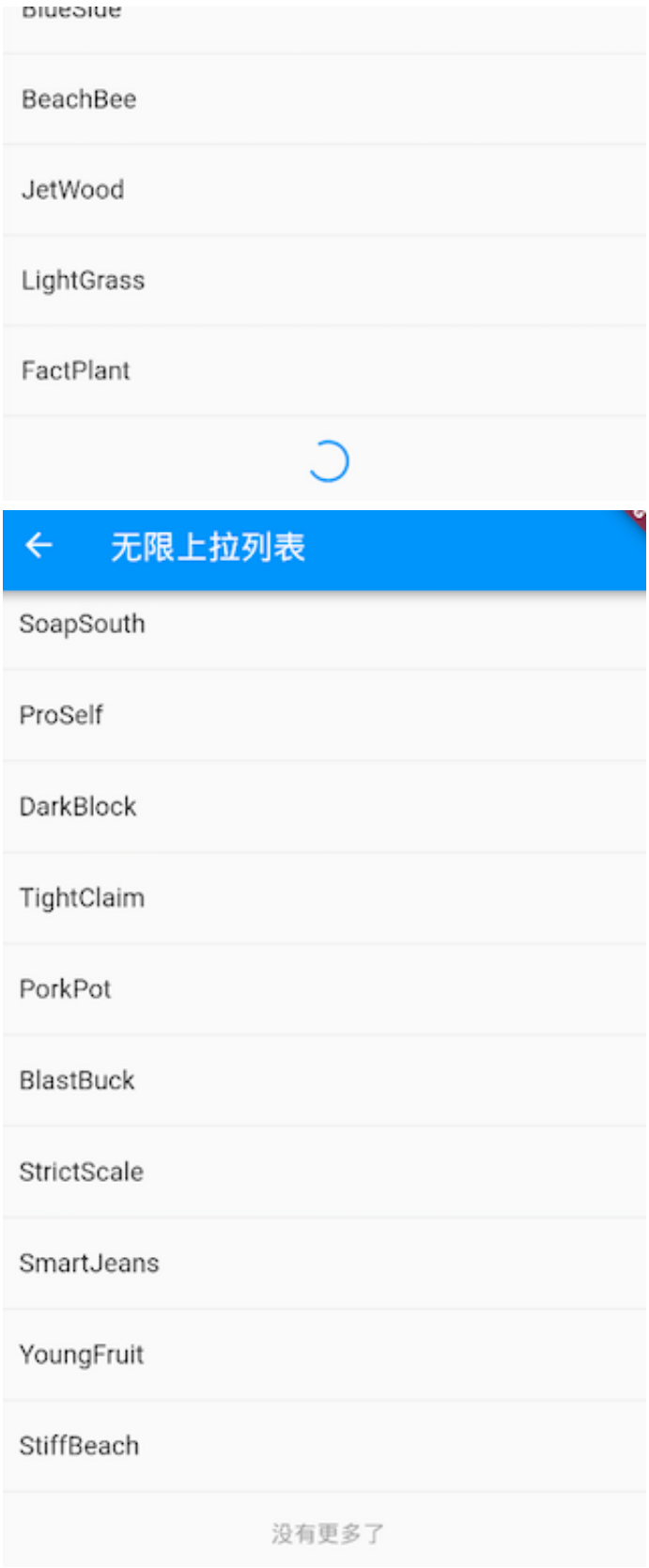


实例：无限加载列表

假设我们要从数据源异步分批拉取一些数据，然后用ListView显示，当我们滑动到列表末尾时，判断是否需要再去拉取数据，如果是，则去拉取，拉取过程中在表尾显示一个loading，拉取成功后将数据插入列表；如果不需要再去拉取，则在表尾提示"没有更多"。代码如下：

```
class InfiniteListView extends StatefulWidget {
  @override
  _InfiniteListViewState createState() => new
_InfiniteListViewState();
}

class _InfiniteListViewState extends State<InfiniteListView> {
  static const loadingTag = "##loading##"; //表尾标记
  var _words = <String>[loadingTag];

  @override
  void initState() {
    _retrieveData();
  }

  @override
  Widget build(BuildContext context) {
    return ListView.separated(
      itemCount: _words.length,
      itemBuilder: (context, index) {
        //如果到了表尾
        if (_words[index] == loadingTag) {
          //不足100条，继续获取数据
          if (_words.length - 1 < 100) {
            //获取数据
            _retrieveData();
            //加载时显示loading
            return Container(
              padding: const EdgeInsets.all(16.0),
              alignment: Alignment.center,
              child: SizedBox(
                  width: 24.0,
                  height: 24.0,
                  child: CircularProgressIndicator(strokeWidth:
2.0)
              ),
            );
          } else {
```

```
            //已经加载了100条数据，不再获取数据。
            return Container(
                alignment: Alignment.center,
                padding: EdgeInsets.all(16.0),
                child: Text("没有更多了", style: TextStyle(color:
Colors.grey),)
            );
          }
        }
        //显示单词列表项
        return ListTile(title: Text(_words[index]));
      },
      separatorBuilder: (context, index) => Divider(height: .0),
    );
  }

  void _retrieveData() {
    Future.delayed(Duration(seconds: 2)).then((e) {
      _words.insertAll(_words.length - 1,
          //每次生成20个单词
          generateWordPairs().take(20).map((e) =>
e.asPascalCase).toList()
      );
      setState(() {
        //重新构建列表
      });
    });
  }

}
```


BeachBee
JetWood
LightGrass
FactPlant
无限上拉列表
SoapSouth
ProSelf
DarkBlock
TightClaim
PorkPot
BlastBuck
StrictScale
SmartJeans
YoungFruit
StiffBeach
没有更多了

代码比较简单，读者可以参照代码中的注释理解，故不再赘述。需要说明的是，`_retrieveData()`的功能是模拟从数据源异步获取数据，我们使用english_words包的`generateWordPairs()`方法每次生成20个单词。

总结

本节主要介绍了ListView的一些公共参数以及常用的构造函数。不同的构造函数对应了不同的列表项生成模型，如果需要自定义列表项生成模型，可以通过`ListView.custom`来自定义，它需要实现一个SliverChildDelegate用来给ListView生成列表项widget，更多详情请参考API文档。

GridView

GridView可以构建一个二维网格列表，其默认构造函数定义如下：

```
GridView({
  Axis scrollDirection = Axis.vertical,
  bool reverse = false,
  ScrollController controller,
  bool primary,
  ScrollPhysics physics,
  bool shrinkWrap = false,
  EdgeInsetsGeometry padding,
  @required SliverGridDelegate gridDelegate, //控制子widget layout的
委托
  bool addAutomaticKeepAlives = true,
  bool addRepaintBoundaries = true,
  double cacheExtent,
  List<Widget> children = const <Widget>[],
})
```

我们可以看到，GridView和ListView的大多数参数都是相同的，它们的含义也都相同，如有疑惑读者可以翻阅ListView一节，在此不再赘述。我们唯一需要关注的是`gridDelegate`参数，类型是SliverGridDelegate，它的作用是控制GridView子widget如何排列(layout)，SliverGridDelegate是一个抽象类，定义了GridView Layout相关接口，子类需要通过实现它们来实现具体的布局算法，Flutter中提供了两个SliverGridDelegate的子类SliverGridDelegateWithFixedCrossAxisCount和SliverGridDelegateWithMaxCrossAxisExtent，下面我们分别介绍：

SliverGridDelegateWithFixedCrossAxisCount

该子类实现了一个纵轴为固定数量子元素的layout算法，其构造函数为：

```
SliverGridDelegateWithFixedCrossAxisCount({
  @required double crossAxisCount,
  double mainAxisSpacing = 0.0,
  double crossAxisSpacing = 0.0,
  double childAspectRatio = 1.0,
})
```

- crossAxisCount：纵轴子元素的数量。此属性值确定后子元素在纵轴的长度就确定了,即ViewPort纵轴长度/crossAxisCount。
- mainAxisSpacing：主轴方向的间距。
- crossAxisSpacing：纵轴方向子元素的间距。
- childAspectRatio：子元素在纵轴长度和主轴长度的比例。由于crossAxisCount指定后子元素纵轴长度就确定了，然后通过此参数值就可以确定子元素在主轴的长度。

可以发现，子元素的大小是通过crossAxisCount和childAspectRatio两个参数共同决定的。注意，这里的子元素指的是子widget的最大显示空间，注意确保子widget的实际大小不要超出子元素的空间。

下面看一个例子：

```
GridView(
  gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(
      crossAxisCount: 3, //纵轴三个子widget
      childAspectRatio: 1.0 //宽高比为1时，子widget
  ),
  children:<Widget>[
    Icon(Icons.ac_unit),
    Icon(Icons.airport_shuttle),
    Icon(Icons.all_inclusive),
    Icon(Icons.beach_access),
    Icon(Icons.cake),
    Icon(Icons.free_breakfast)
  ]
);
```

GridView.count

GridView.count构造函数内部使用了SliverGridDelegateWithFixedCrossAxisCount，我们通过它可以快速的创建纵轴固定数量子元素的GridView，上面的示例代码等价于：

```
GridView.count(
  crossAxisCount: 3,
  childAspectRatio: 1.0,
  children: <Widget>[
    Icon(Icons.ac_unit),
    Icon(Icons.airport_shuttle),
    Icon(Icons.all_inclusive),
    Icon(Icons.beach_access),
    Icon(Icons.cake),
    Icon(Icons.free_breakfast),
  ],
);
```

SliverGridDelegateWithMaxCrossAxisExtent

该子类实现了一个纵轴子元素为固定最大长度的layout算法，其构造函数为：

```
SliverGridDelegateWithMaxCrossAxisExtent({
  double maxCrossAxisExtent,
  double mainAxisSpacing = 0.0,
  double crossAxisSpacing = 0.0,
  double childAspectRatio = 1.0,
})
```

maxCrossAxisExtent为子元素在纵轴上的最大长度，之所以是“最大”长度，是**因为纵轴方向每个子元素的长度仍然是等分的**，举个例子，如果ViewPort的纵轴长度是450，那么当maxCrossAxisExtent的值在区间(450/4，450/3]内的话，子元素最终实际长度都为150，而`childAspectRatio`所指的子元素纵轴和主轴的长度比为**最终的长度比**。其它参数和SliverGridDelegateWithFixedCrossAxisCount相同。

下面我们看一个例子：

```
GridView(
  padding: EdgeInsets.zero,
  gridDelegate: SliverGridDelegateWithMaxCrossAxisExtent(
      maxCrossAxisExtent: 120.0,
      childAspectRatio: 2.0 //宽高比为2
  ),
  children: <Widget>[
    Icon(Icons.ac_unit),
    Icon(Icons.airport_shuttle),
    Icon(Icons.all_inclusive),
    Icon(Icons.beach_access),
    Icon(Icons.cake),
    Icon(Icons.free_breakfast),
  ],
);
```

GridView.extent

GridView.extent构造函数内部使用了SliverGridDelegateWithMaxCrossAxisExtent，我们通过它可以快速的创建纵轴子元素为固定最大长度的的GridView，上面的示例代码等价于：

```
GridView.extent(
   maxCrossAxisExtent: 120.0,
   childAspectRatio: 2.0,
   children: <Widget>[
     Icon(Icons.ac_unit),
     Icon(Icons.airport_shuttle),
     Icon(Icons.all_inclusive),
     Icon(Icons.beach_access),
     Icon(Icons.cake),
     Icon(Icons.free_breakfast),
   ],
 );
```

GridView.builder

上面我们介绍的GridView都需要一个Widget数组作为其子元素，这些方式都会提前将所有子widget都构建好，所以只适用于子Widget数量比较少时，当子widget比较多时，我们可以通过`GridView.builder`来动态创建子Widget。`GridView.builder`必须指定的参数有两个：

```
GridView.builder(
 ...
 @required SliverGridDelegate gridDelegate,
 @required IndexedWidgetBuilder itemBuilder,
)
```

其中itemBuilder为子widget构建器。

示例

假设我们需要从一个异步数据源（如网络）分批获取一些Icon，然后用GridView来展示：

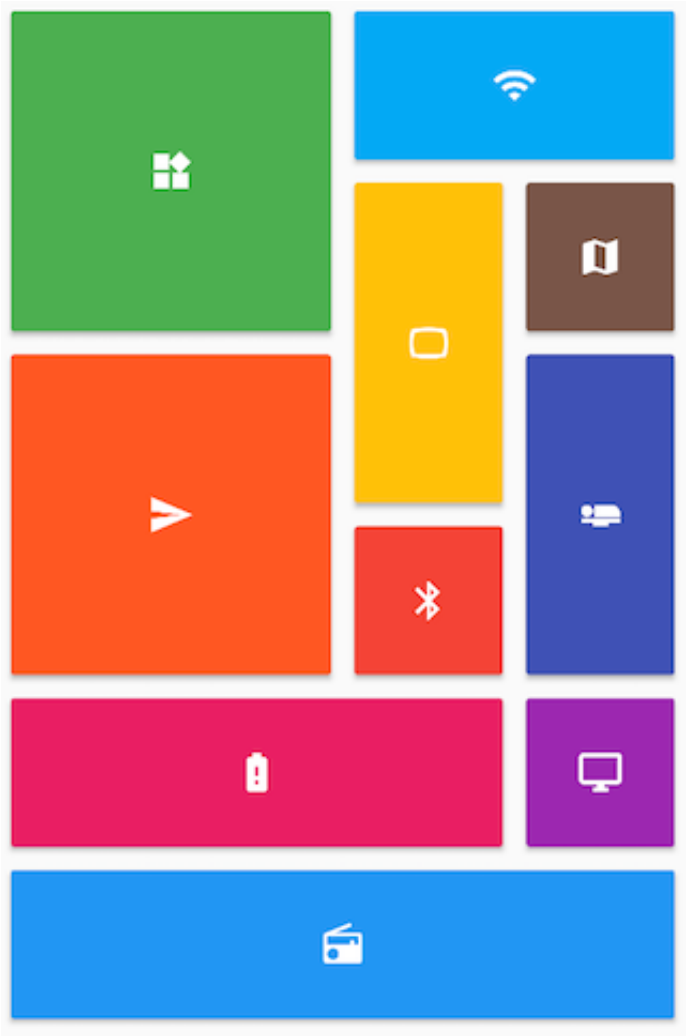
```
class InfiniteGridView extends StatefulWidget {
  @override
  _InfiniteGridViewState createState() => new
```

```
_InfiniteGridViewState();
}

class _InfiniteGridViewState extends State<InfiniteGridView> {

  List<IconData> _icons = []; //保存Icon数据

  @override
  void initState() {
    // 初始化数据
    _retrieveIcons();
  }

  @override
  Widget build(BuildContext context) {
    return GridView.builder(
        gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(
            crossAxisCount: 3, //每行三列
            childAspectRatio: 1.0 //显示区域宽高相等
        ),
        itemCount: _icons.length,
        itemBuilder: (context, index) {
          //如果显示到最后一个并且Icon总数小于200时继续获取数据
          if (index == _icons.length - 1 && _icons.length < 200) {
            _retrieveIcons();
          }
          return Icon(_icons[index]);
        }
    );
  }

  //模拟异步获取数据
  void _retrieveIcons() {
    Future.delayed(Duration(milliseconds: 200)).then((e) {
      setState(() {
        _icons.addAll([
          Icons.ac_unit,
          Icons.airport_shuttle,
          Icons.all_inclusive,
          Icons.beach_access, Icons.cake,
          Icons.free_breakfast
        ]);
      });
    });
  }
}
```

- `_retrieveIcons()`：在此方法中我们通过`Future.delayed`来模拟从异步数据源获取数据，每次获取数据需要200毫秒，获取成功后将新数据添加到_icons，然后调用setState重新构建。
- 在itemBuilder中，如果显示到最后一个时，判断是否需要继续获取数据，然后返回一个Icon。

更多

Flutter的GridView默认子元素显示空间是相等的，但在实际开发中，你可能会遇到子元素大小不等的情况，如下面这样的布局：



Pub上有一个包“flutter_staggered_grid_view”，它实现了一个交错GridView的布局模型，可以很轻松的实现这种布局，详情读者可以自行了解。

CustomScrollView

CustomScrollView可以使用sliver自定义滚动模型（效果）的widget。它可以包含多种滚动模型，举个例子，假设有一个页面，顶部需要一个GridView，底部需要一个ListView，而要求整个页面的滑动效果是统一一致的，即它们看起来是一个整体，如果使用GridView+ListView来实现的话，就不能保证一致的滑动效果，因为它们的滚动效果是分离，所以这时就需要一个"胶水"，把这些彼此独立的可滚动widget（Sliver）"粘"起来，而CustomScrollView的功能就相当于“胶水”。

Sliver

Sliver有细片、小片之意，在Flutter中，Sliver通常指具有特定滚动效果的可滚动块。可滚动widget，如ListView、GridView等都有对应的Sliver实现如SliverList、SliverGrid等。对于大多数Sliver来说，它们和可滚动Widget最主要的区别是**Sliver不会包含Scrollable Widget，也就是说Sliver本身不包含滚动交互模型**，正因如此，CustomScrollView才可以将多个Sliver"粘"在一起，这些Sliver共用CustomScrollView的Scrollable，最终实现统一的滑动效果。

> Sliver系列Widget比较多，我们不会一一介绍，读者只需记住它的特点，需要时再去查看文档即可。上面之所以说“大多数“Sliver都和可滚动Widget对应，是由于还有一些如SliverPadding、SliverAppBar等是和可滚动Widget无关的，它们主要是为了结合CustomScrollView一起使用，这是因为**CustomScrollView的子widget必须都是Sliver**。

示例

```dart
import 'package:flutter/material.dart';

class CustomScrollViewTestRoute extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    //因为本路由没有使用Scaffold，为了让子级Widget(如Text)使用
    //Material Design 默认的样式风格,我们使用Material作为本路由的根。
    return Material(
      child: CustomScrollView(
        slivers: <Widget>[
          //AppBar，包含一个导航栏
          SliverAppBar(
            pinned: true,
            expandedHeight: 250.0,
```

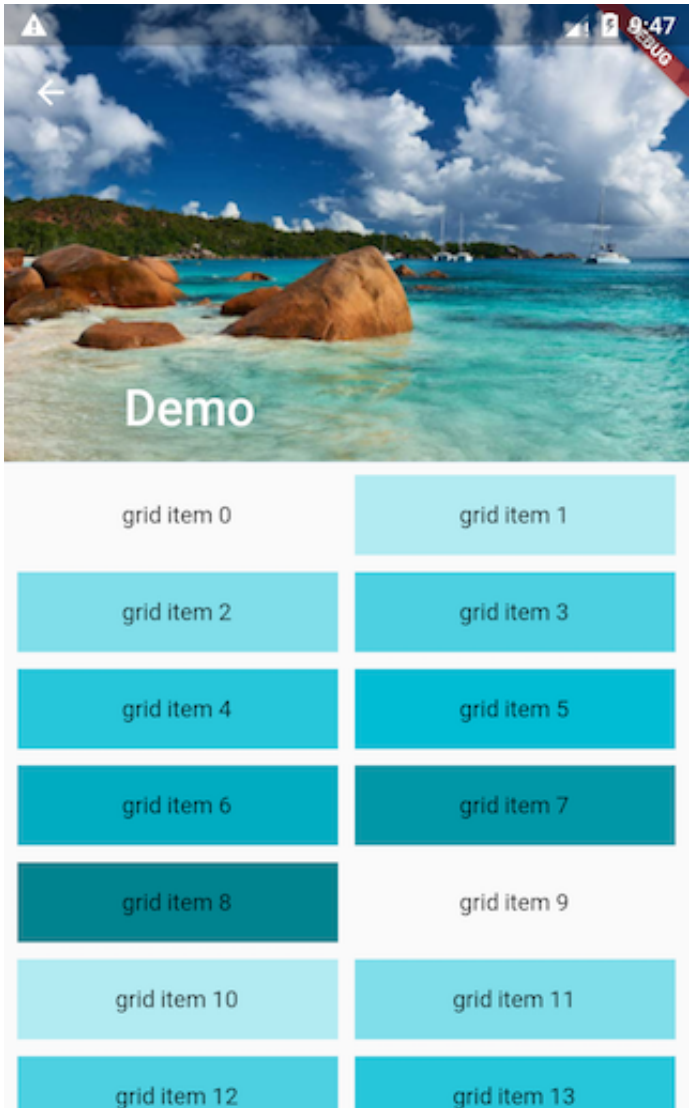
```
            flexibleSpace: FlexibleSpaceBar(
              title: const Text('Demo'),
              background: Image.asset(
                "./images/avatar.png", fit: BoxFit.cover,),
              ),
            ),

            SliverPadding(
              padding: const EdgeInsets.all(8.0),
              sliver: new SliverGrid( //Grid
                gridDelegate: new
SliverGridDelegateWithFixedCrossAxisCount(
                  crossAxisCount: 2, //Grid按两列显示
                  mainAxisSpacing: 10.0,
                  crossAxisSpacing: 10.0,
                  childAspectRatio: 4.0,
                ),
                delegate: new SliverChildBuilderDelegate(
                      (BuildContext context, int index) {
                    //创建子widget
                    return new Container(
                      alignment: Alignment.center,
                      color: Colors.cyan[100 * (index % 9)],
                      child: new Text('grid item $index'),
                    );
                  },
                  childCount: 20,
                ),
              ),
            ),
            //List
            new SliverFixedExtentList(
              itemExtent: 50.0,
              delegate: new SliverChildBuilderDelegate(
                      (BuildContext context, int index) {
                    //创建列表项
                    return new Container(
                      alignment: Alignment.center,
                      color: Colors.lightBlue[100 * (index % 9)],
                      child: new Text('list item $index'),
                    );
                  },
                  childCount: 50 //50个列表项
              ),
            ),
          ],
        ),
      );
```

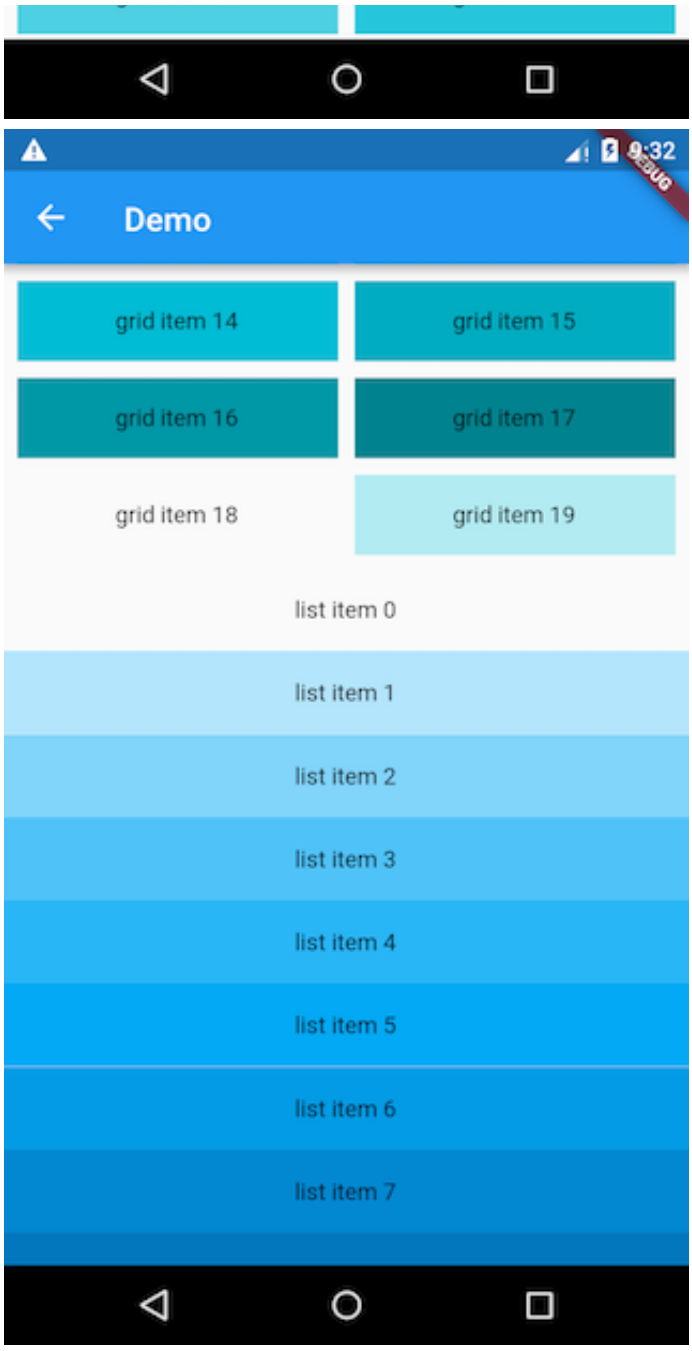
```
    }
  }
```

代码分为三部分：

- 头部SliverAppBar：SliverAppBar对应AppBar，两者不同之处在于SliverAppBar可以集成到CustomScrollView。SliverAppBar可以结合FlexibleSpaceBar实现Material Design中头部伸缩的模型，具体效果，读者可以运行该示例查看。
- 中间的SliverGrid：它用SliverPadding包裹以给SliverGrid添加补白。SliverGrid是一个两列，宽高比为4的网格，它有20个子widget。
- 底部SliverFixedExtentList：它是一个所有子元素高度都为50像素的列表。

运行效果：





滚动监听及控制

在前几节中，我们介绍了Flutter中常用的可滚动Widget，也说过可以用ScrollController来控制可滚动widget的滚动位置，本节先介绍一下ScrollController，然后以ListView为例，展示一下ScrollController的具体用法。最后，再介绍一下路由切换时如何来保存滚动位置。

ScrollController

构造函数：

```
ScrollController({
  double initialScrollOffset = 0.0, //初始滚动位置
  this.keepScrollOffset = true,//是否保存滚动位置
  ...
})
```

我们介绍一下ScrollController常用的属性和方法：

- `offset`：可滚动Widget当前滚动的位置。
- `jumpTo(double offset)`、`animateTo(double offset,...)`：这两个方法用于跳转到指定的位置，它们不同之处在于，后者在跳转时会执行一个动画，而前者不会。

ScrollController还有一些属性和方法，我们将在后面原理部分解释。

滚动监听

ScrollController间接继承自Listenable，我们可以根据ScrollController来监听滚动事件。如：

```
controller.addListener(()=>print(controller.offset))
```

示例

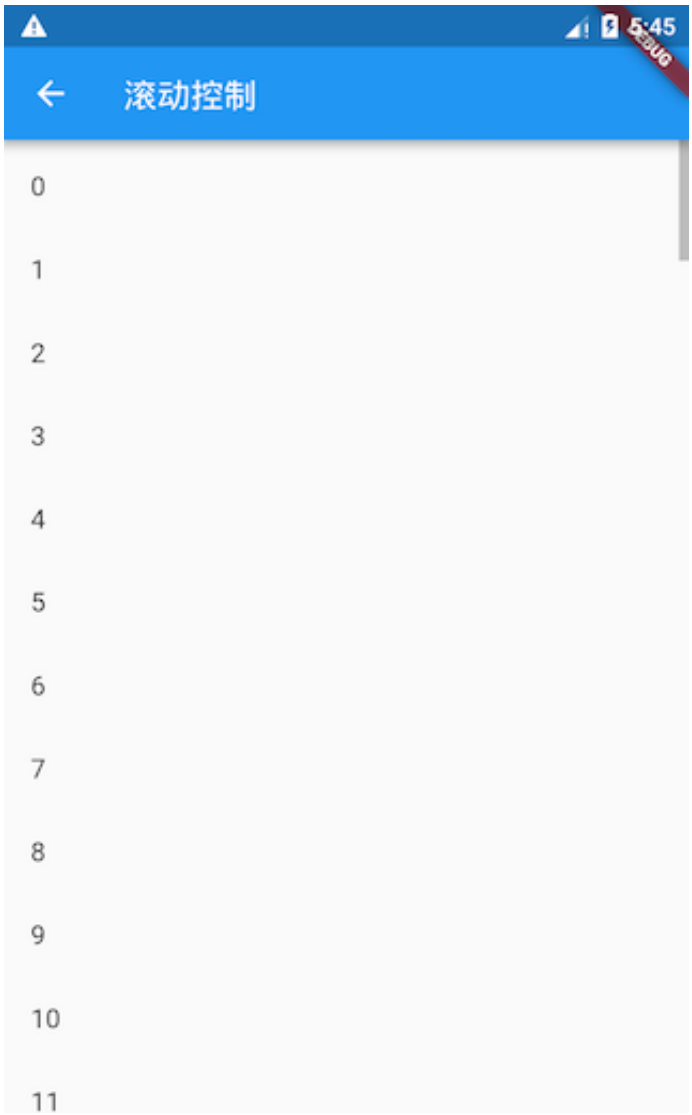
我们创建一个ListView，当滚动位置发生变化时，我们先打印出当前滚动位置，然后判断当前位置是否超过1000像素，如果超过则在屏幕右下角显示一个“返回顶部”的按钮，该按钮点击后可以使ListView恢复到初始位置；如果没有超过1000像素，则隐藏“返回顶部”按钮。代码如下：

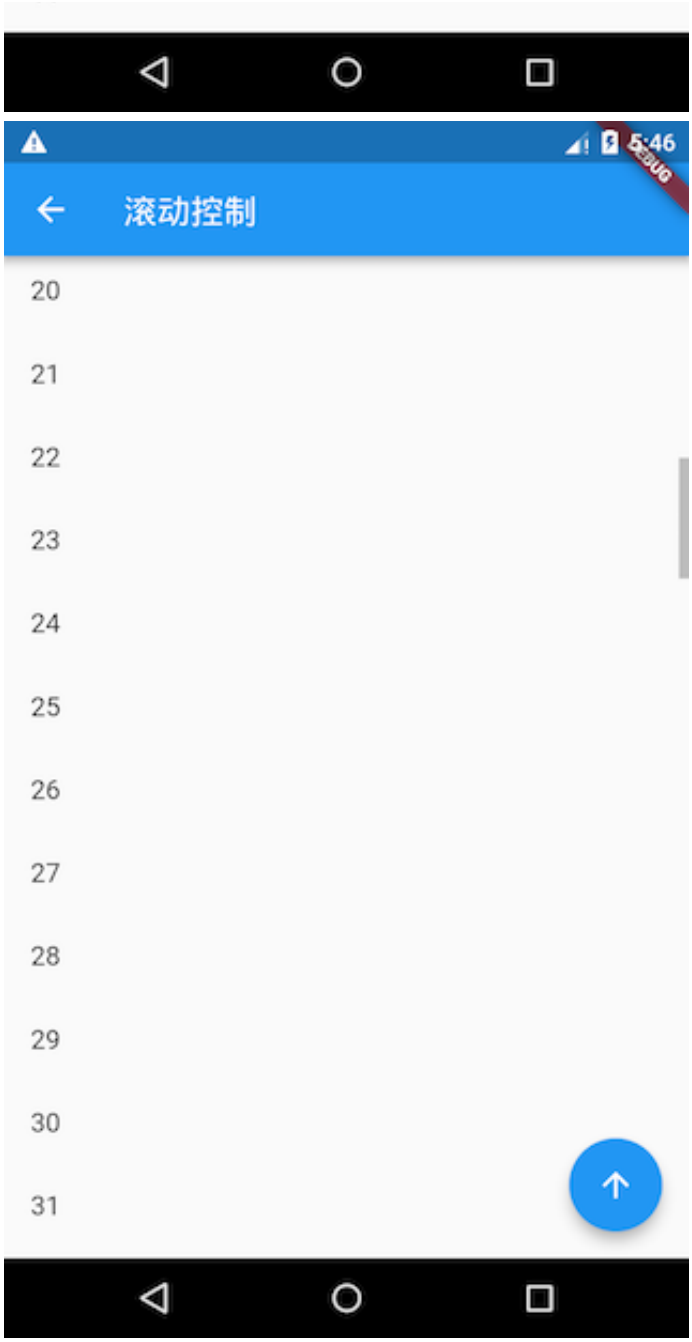
```
class ScrollControllerTestRoute extends StatefulWidget {
  @override
  ScrollControllerTestRouteState createState() {
    return new ScrollControllerTestRouteState();
```

```
  }
}

class ScrollControllerTestRouteState extends
State<ScrollControllerTestRoute> {
  ScrollController _controller = new ScrollController();
  bool showToTopBtn = false; //是否显示“返回到顶部”按钮

  @override
  void initState() {
    //监听滚动事件，打印滚动位置
    _controller.addListener(() {
      print(_controller.offset); //打印滚动位置
      if (_controller.offset < 1000 && showToTopBtn) {
        setState(() {
          showToTopBtn = false;
        });
      } else if (_controller.offset >= 1000 && showToTopBtn ==
false) {
        setState(() {
          showToTopBtn = true;
        });
      }
    });
  }

  @override
  void dispose() {
    //为了避免内存泄露，需要调用_controller.dispose
    _controller.dispose();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("滚动控制")),
      body: Scrollbar(
        child: ListView.builder(
            itemCount: 100,
            itemExtent: 50.0, //列表项高度固定时，显式指定高度是一个好习惯
(性能消耗小)
            controller: _controller,
            itemBuilder: (context, index) {
              return ListTile(title: Text("$index"),);
            }
        ),
      ),
```

```
      floatingActionButton: !showToTopBtn ? null :
FloatingActionButton(
          child: Icon(Icons.arrow_upward),
          onPressed: () {
            //返回到顶部时执行动画
            _controller.animateTo(.0,
                duration: Duration(milliseconds: 200),
                curve: Curves.ease
            );
          }
      ),
    );
  }
}
```

代码说明已经包含在注释里，下面我们看看运行效果：





由于列表项高度为50像素，当滑动到第20个列表项后，右下角“返回顶部”按钮会显示，点击该按钮，ListView会在返回顶部的过程中执行一个滚动动画，动画时间是200毫秒，动画曲线是Curves.ease，关于动画的详细内容我们将在后面“动画”一章中详细介绍。

滚动位置恢复

PageStorage是一个用于保存页面(路由)相关数据的Widget，它并不会影响子树的UI外观，其实，PageStorage是一个功能型Widget，它拥有一个存储桶（bucket），子树中的Widget可以通过指定不同的PageStorageKey来存储各自的数据或状态。

每次滚动结束，Scrollable Widget都会将滚动位置`offset`存储到PageStorage中，当Scrollable Widget 重新创建时再恢复。如果`ScrollController.keepScrollOffset`为`false`，则滚动位置将不会被存储，Scrollable Widget重新创建时会使用`ScrollController.initialScrollOffset`；`ScrollController.keepScrollOffset`为`true`时，Scrollable Widget在**第一次**创建时，会滚动到`initialScrollOffset`处，因为这时还没有存储过滚动位置。在接下来的滚动中就会存储、恢复滚动位置，而`initialScrollOffset`会被忽略。

当一个路由中包含多个Scrollable Widget时，如果你发现在进行一些跳转或切换操作后，滚动位置不能正确恢复，这时你可以通过显式指定PageStorageKey来分别跟踪不同Scrollable Widget的位置，如：

```
ListView(key: PageStorageKey(1), ... );
...
ListView(key: PageStorageKey(2), ... );
```

不同的PageStorageKey，需要不同的值，这样才可以区分为不同Scrollable Widget保存的滚动位置。

> 注意：一个路由中包含多个Scrollable Widget时，如果要分别跟踪它们的滚动位置，并非一定就得给他们分别提供PageStorageKey。这是因为Scrollable本身是一个StatefulWidget，它的状态中也会保存当前滚动位置，所以，只要Scrollable Widget本身没有被从树上detach掉，那么其State就不会销毁(dispose)，滚动位置就不会丢失。只有当Widget发生结构变化，导致Scrollable Widget的State销毁或重新构建时才会丢失状态，这种情况就需要显式指定PageStorageKey，通过PageStorage来存储滚动位置，一个典型的场景是在使用TabBarView时，在Tab发生切换时，Tab页中的Scrollable Widget的State就会销毁，这时如果想恢复滚动位置就需要指定PageStorageKey。

ScrollPosition

一个ScrollController可以同时被多个Scrollable Widget使用，ScrollController会为每一个Scrollable Widget创建一个ScrollPosition对象，这些ScrollPosition保存在ScrollController的`positions`属性中（List<ScrollPosition>）。ScrollPosition是真正保存滑动位置信息的对象，`offset`只是一个便捷属性：

```
double get offset => position.pixels;
```

一个ScrollController虽然可以对应多个Scrollable Widge，但是有一些操作，如读取滚动位置`offset`，则需要一对一，但是我们仍然可以在一对多的情况下，通过其它方法读取滚动位置，举个例子，假设一个ScrollController同时被两个Scrollable Widget使用，那么我们可以通过如下方式分别读取他们的滚动位置：

```
...
controller.positions.elementAt(0).pixels
controller.positions.elementAt(1).pixels
...
```

我们可以通过`controller.positions.length`来确定`controller`被几个Scrollable Widget使用。

方法

ScrollPosition有两个常用方法：`animateTo()`和`jumpTo()`，它们是真正来控制跳转滚动位置的方法，ScrollController的这两个同名方法，内部最终都会调用ScrollPosition的。

ScrollController控制原理

我们来介绍一下ScrollController的另外三个方法：

```
ScrollPosition createScrollPosition(
    ScrollPhysics physics,
    ScrollContext context,
    ScrollPosition oldPosition);
void attach(ScrollPosition position) ;
void detach(ScrollPosition position) ;
```

当ScrollController和Scrollable Widget关联时，Scrollable Widget首先会调用ScrollController的`createScrollPosition()`方法来创建一个ScrollPosition来存储滚动位置信息，接着，Scrollable Widget会调用`attach()`方法，将创建的ScrollPosition添加到ScrollController的`positions`属性中，这一步称为“注册位置”，只有注册后`animateTo()`和`jumpTo()`才可以被调用。当Scrollable Widget销毁时，会调用ScrollController的`detach()`方法，将其ScrollPosition对象从ScrollController的`positions`属性中移除，这一步称为“注销位置”，注销后`animateTo()`和`jumpTo()`将不能再被调用。

需要注意的是，ScrollController的`animateTo()`和`jumpTo()`内部会调用所有ScrollPosition的`animateTo()`和`jumpTo()`，以实现所有和该ScrollController关联的Scrollable Widget都滚动到指定的位置。

滚动监听

Flutter Widget树中子Widget可以通过发送通知（Notification）与父(包括祖先)Widget通信。父Widget可以通过NotificationListener Widget来监听自己关注的通知，这种通信方式类似于Web开发中浏览器的事件冒泡，我们在Flutter中沿用“冒泡”这个术语。Scrollable Widget在滚动时会发送ScrollNotification类型的通知，ScrollBar正是通过监听滚动通知来实现的。通过NotificationListener监听滚动事件和通过ScrollController有两个主要的不同：

1. 通过NotificationListener可以在从Scrollable Widget到Widget树根之间任意位置都能监听。而ScrollController只能和具体的Scrollable Widget关联后才可以。
2. 收到滚动事件后获得的信息不同；NotificationListener在收到滚动事件时，通知中会携带当前滚动位置和ViewPort的一些信息，而ScrollController只能获取当前滚动位置。

NotificationListener<T>

NotificationListener<T>是一个Widget，模板参数T是想监听的通知类型，如果省略，则所有类型通知都会被监听，如果指定特定类型，则只有该类型的通知会被监听。NotificationListener需要一个onNotification回调函数，用于实现监听处理逻辑，该回调可以返回一个布尔值，代表是否阻止该事件继续向上冒泡，如果为`true`时，则冒泡终止，事件停止向上传播，如果不返回或者返回值为`false`时，则冒泡继续。

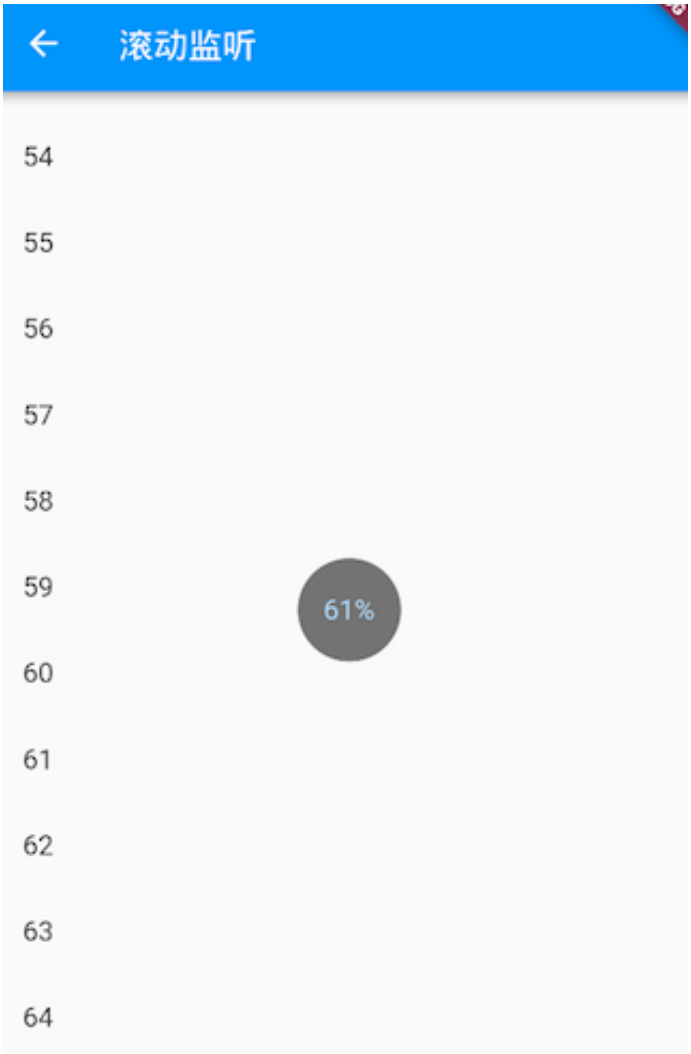
示例

下面，我们监听ListView的滚动通知，然后显示当前滚动进度百分比：

```
import 'package:flutter/material.dart';

class ScrollNotificationTestRoute extends StatefulWidget {
  @override
  _ScrollNotificationTestRouteState createState() =>
      new _ScrollNotificationTestRouteState();
}

class _ScrollNotificationTestRouteState
    extends State<ScrollNotificationTestRoute> {
  String _progress = "0%"; //保存进度百分比

  @override
  Widget build(BuildContext context) {
    return Scrollbar( //进度条
      // 监听滚动通知
      child: NotificationListener<ScrollNotification>(
        onNotification: (ScrollNotification notification) {
          double progress = notification.metrics.pixels /
              notification.metrics.maxScrollExtent;
          //重新构建
          setState(() {
            _progress = "${(progress * 100).toInt()}%";
          });
          print("BottomEdge: ${notification.metrics.extentAfter ==
0}");
          //return true; //放开此行注释后，进度条将失效
        },
        child: Stack(
          alignment: Alignment.center,
          children: <Widget>[
            ListView.builder(
                itemCount: 100,
                itemExtent: 50.0,
                itemBuilder: (context, index) {
                  return ListTile(title: Text("$index"));
                }
            ),
            CircleAvatar(  //显示进度百分比
              radius: 30.0,
              child: Text(_progress),
              backgroundColor: Colors.black54,
```

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

我们看一看运行结果：



在接收到滚动事件时，参数类型为ScrollNotification，它包括一个`metrics`属性，它的类型是ScrollMetrics，该属性包含当前ViewPort及滚动位置等信息：

- pixels：当前滚动位置。
- maxScrollExtent：最大可滚动长度。
- extentBefore：滑出ViewPort顶部的长度；此示例中相当于顶部滑出屏幕上方的列表长度。
- extentInside：ViewPort内部长度；此示例中屏幕显示的列表部分的长度。
- extentAfter：列表中未滑入ViewPort部分的长度；此示例中列表底部未显示到屏幕范围部分的长度。
- atEdge：是否滑到了Scrollable Widget的边界（此示例中相当于列表顶或底部）。

ScrollMetrics还有一些其它属性，读者可以自行查阅API文档。