第 5 章

文本显示 Text 组件

在 Flutter 中，Text 组件用于显示文本，类似 Android 里的 TextView，以及 iOS 里的 UILabel，最简单的使用方式可以直接用 new Text 这样的代码来展示文本，代码如下：

```
new Text('这里是文本')
```

5.1 文本显示组件的基本使用

在实际应用项目开发中，文本显示是视图内容的主要表现方式之一，Flutter 中使用 Text 组件来显示文本，如图 5-1 所示，页面的 AppBar 使用了 Text，页面的主体内容也使用了 Text，代码如下：

```
//5.2 /lib/code3/main_data66.dart
//文本显示组件 Text
Widget buildBodyFunction() {
  return Scaffold(
appBar: AppBar(
      title: Text("这是一个标题"),
    ),
    body: Center(child: Text("内容主体"),),);
}
```

对于文本组件 Text 来讲，创建及使用比较简单，直接使用 new Text()就可创建，在实际项目开发中，每一行显示的文本都有对应的样式来配置显示，在这里，通过 new Text()来创建文本组件，Text 的属性 style 用来配置当前文本的样式，style 属性配置的是样式组件 TextStyle。

图 5-1　Text 的基本使用

5.2　样式组件 TextStyle 的使用分析

在文本显示组件 Text 中，TextStyle 用来配置显示的文本样式，如图 5-2 所示 AppBar 中 title 配置的文本 Text 的字体颜色为红色，实现代码如下：

```
//5.2.2 /lib/code3/main_data67.dart
//文本显示组件 Text
Widget buildBodyFunction() {
  return Scaffold(
appBar: AppBar(
     title: Text(
       "这是一个标题",
         style: TextStyle(
            //用来配置文字的颜色
            color: Colors.red,
             ),
        ),
    ),
    body: Center(child: Text("内容主体"),),);
}
```

图 5-2　Text 文本的颜色配置使用

通过 TextStyle，可以配置文本的颜色，同样也可以设置文字大小、斜体、粗细等一系列样式，如表 5-1 所示，列出了 TextStyle 属性配置说明。

表 5-1　TextStyle 属性配置说明

取　值	类　型	说　明
inherit	bool	为 true 时，表示把当前样式合并 defaultStyle 样式一起使用
color	Color	文本的颜色
backgroundColor	Color	Text 文本的背景色
fontSize	double	字体大小配置
fontWeight	FontWeight	字体粗细配置
fontStyle	FontStyle	字体样式配置，如常规体、斜体
letterSpacing	double	字符之间的间隔
wordSpacing	double	单词之间的间隔
textBaseline	TextBaseline	文本绘制时对齐的基线
height	double	文本的高度
foreground	Paint	通过 Paint 来设置文本的颜色，不能与 color 属性同时配置，属于互斥关系
background	Paint	用来绘制 Text 文本的背景颜色，不能与 backgroundColor 同时配置
shadows	List < ui. Shadow >	字体阴影设置
decoration	TextDecoration	Text 的装饰线配置，如下画线、删除线
decorationColor	Color	装饰线的颜色配置
decorationStyle	TextDecorationStyle	装饰线的样式，如波浪式、实线、虚线等
decorationThickness	double	装饰线的粗细配置
debugLabel	String	此属性仅在调试构建中维护，配置对文本样式可读的描述，用来辅助理解，目前在开发中无实际用处
fontFamily	String	字体设置
fontFamilyFallback	fontFamilyFallback	fontFamily 字体加载后，优先选择从这里配置的字体

5.2.1　样式组件 TextStyle 的 inherit

在英文的意思中，inherit 为继承，在这里可理解为是否继承父组件的属性样式，默认为 true，在 Text 里使用时，默认会加载一个默认样式 DefaultTextStyle，文本样式为白色，

10 像素，sans-serif 字体，如果设置 inherit 为 true，程序在执行时会把默认样式 DefaultTextStyle 与当前设置的文本样式进行合并，Text 使用样式的部分源码如下：

```
... //省略
if (style == null || style.inherit)
effectiveTextStyle = defaultTextStyle.style.merge(style);
... //省略
```

如图 5-3 所示 Text 设置了 inherit 一个为 true，另一个为 false 的效果图，对应的代码如下：

```
//5.2.2 /lib/code3/main_data67.dart
//文本显示组件 Text
Widget buildBodyFunction3() {
  return Scaffold(
appBar: AppBar(
      title: Text(
          "这是一个标题"
      ),
    ),
    body: Container(child: Center(
      child: Column(children: [
        Text("inherit: true", style: TextStyle(
//文字的大小
fontSize: 16,
          inherit: true,),),
        Text("inherit: false ", style: TextStyle(
//文字的大小
fontSize: 16,
          inherit: false,),),

      ],),),color: Colors.grey,));
}
```

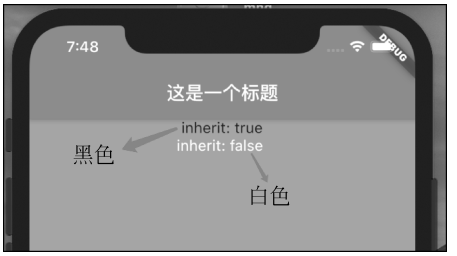

图 5-3 inherit 属性的配置

5.2.2　样式组件 TextStyle 的颜色配置

属性 color 用来设置文字的颜色，backgroundColor 用来设置 Text 的背景，是一个 Color 类型的配置。background 也是用来设置 Text 的背景，是一个 Paint 类型的配置，在 TextStyle 中，两者是互斥的，不可同时使用。

对于 background 与 backgroundColor 来讲，在 CSS 中，background 可以设置背景颜色、背景图片等，而 backgroundColor 只能设置一种颜色，同理 TextStyle 中的 backgroundColor 只能配置一种 Color，而 background 可以通过 Paint 绘制比较复杂的背景样式。

5.2.3　文字大小 fontSize

TextStyle 的 fontSize 是用来配置文字显示大小的，是一个 double 类型的配置，一般使用时的代码如下：

```
Text("设置文字大小", style: TextStyle(fontSize: 16.0,),
```

这里设置的 16.0 是逻辑像素大小，当 TextStyle 没有指定 fontSize 时，默认值是 14.0。

设备独立像素，device independent pixel，简称 dip，在 CSS 中，有

```
CSS 像素 = 设备独立像素 = 逻辑像素
```

在 Flutter 中有

```
Flutter 像素 = 设备独立像素 = 逻辑像素
```

在分析尺寸时，首先需要明确一些概念，如表 5-2 所示。

表 5-2　像素概念简述

类　别	简称	全　称
物理像素	dp	physical pixel
逻辑像素/设备独立像素	dip	device independent pixels
设备像素比	dpr	device pixel ratio
每英寸的像素	dpi	dots per inch
屏幕像素密度	ppi	pixel per inch

物理像素又被称为设备像素，单位就是用户常说的 pixel，简写成 PX，物理像素也就是实际手机中所描述的屏幕分辨率，如图 5-4 所示，一个设备的物理像素在手机出厂时就固定了。

逻辑像素或者称为设备独立像素，通常也被称设备无关像素，单位是 PT，可以认为逻

辑像素是程序运行平台坐标系统的一个点，这个点代表一个可以由程序使用的虚拟像素，在实际开发中，所提到的一倍屏、二倍屏、三倍屏，指的是一个虚拟像素可以由多少个物理像素来表示，一倍屏就是用一个物理像素来表示一个虚拟像素，多倍屏就是由多个物理像素来表示一个虚拟像素。

图 5-4　手机中的屏幕尺寸效果图

设备像素比，在手机平台中，Android 程序在开启第 1 个 Activity 前，或者 Android 程序在启动时会先创建一个 window，然后在 window 中渲染 Activity。在 iOS 中，在启动的时候开发者常常也会创建一个自定义的 window 来渲染 view，对于 window，有一个 devicePixelRatio 属性，官方的解释为：设备物理像素和设备独立像素的比例，也就是说设备像素比，devicePixelRatio＝物理像素/独立像素。

在 Flutter 中，这里设置的 fontSize 大小为 16.0，那么它实际在不同平台不同手机上显示出来的物理像素应该是 16.0 与 devicePixelRatio 的乘积，在 Flutter 中，可通过 MediaQuery 来获取 devicePixelRatio 的值，以及屏幕尺寸相关的值，代码如下：

```
//5.2.2 /lib/code3/main_data67.dart
//文本显示组件 Text
Widget buildBodyFunction5() {
//获取 devicePixelRatio
  double devicePixelRatio = MediaQuery.of(context).devicePixelRatio;
  //获取当前 Widget 的宽度,因为是填充,所以可理解为是屏幕宽度
  double width = MediaQuery.of(context).size.width;
  //屏幕高度
  double height = MediaQuery.of(context).size.height;
//日志输出
  print("当前设备的 width 为 $width height 为 $height devicePixelRatio 为 $devicePixelRatio");

//定义
  double fontSize =16.0;
//设置中的文字的实际大小计算
  double practicalFontSize = fontSize * devicePixelRatio;
  print("设备中的实际文字大小为 $practicalFontSize");

... //省略

}
```

运行程序，输出的日志信息如下：

```
Performing hot reload...
Syncing files to device iPhone 11 Pro Max...
flutter: 当前设备的 width 为 414.0 height 为 896.0 devicePixelRatio 为 3.0
flutter: 设备中的实际文字大小为 48.0
Reloaded 1 of 499 libraries in 300ms.
```

5.2.4　文字粗细设置 fontWeight

属性 fontWeight 用来设置文字的粗细，接收的值是 FontWeight 枚举类型，基本使用代码如下：

```
Text("设置文字的字体规格",
  style: TextStyle(
//文字的大小
fontSize: 16,
//设置文字的粗细规则为常规体
fontWeight: FontWeight.normal
  ),),
```

在这里调用了 FontWeight. normal，直接取用的值为 FontWeight. w400 的值，FontWeight. w400 是程序默认使用的常规字体，代码如下：

```
//The default font weight.
static const FontWeight normal = w400;
```

FontWeight 所定义的取值范围如下：

```
//A list of all the font weights.
static const List<FontWeight> values = <FontWeight>[
  w100, w200, w300, w400, w500, w600, w700, w800, w900
];
```

其中 w100 为最细的字体，w400 为常规的字体，w900 为粗体。

5.2.5　文字斜体设置

在 TextStyle 中，属性 fontStyle 用来设置文字是否为斜体，取值为 FontStyle. normal 常规体或者 FontStyle. italic 斜体，基本使用代码如下：

```
Text("文字斜体设置",
  style: TextStyle(
```

```
//文字的大小
fontSize: 16,
//设置文字的粗细规则为常规体
fontWeight: FontWeight.normal,
//设置文字为斜体
fontStyle: FontStyle.italic
  ),),
```

5.2.6 文字间距设置

使用 Text 显示文本，文本中包含中文与英文，如图 5-5 所示，代码如下：

```
//5.2.7 /lib/code3/main_data68.dart
//文字间距设置
Widget buildBodyFunction4() {
  return Scaffold(
appBar: AppBar(
      title: Text("Text 组件使用分析"),),
    body: Container(child: Center(
      child: Column(children: [

        Text("天高水长,小编与你奋斗每一天,
              Small make up and you struggle every day",
          style: TextStyle(
letterSpacing: 12.0,
wordSpacing: 20.0,
          ),),
      ]
      ),),),);
}
```

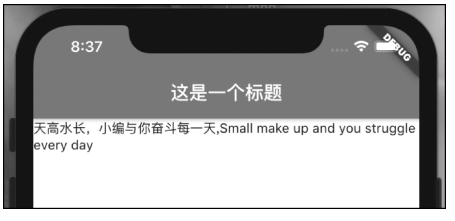

图 5-5 图文混合显示效果图

文本样式 TextStyle 中的 letterSpacing 属性用来配置字符与字符之间的间隔，wordSpacing 用来配置单词之间的间隔，如图 5-6 所示效果是指定这两个属性后运行程序结果，代码如下：

```
//5.2.7 /lib/code3/main_data68.dart
//文字间距设置
Widget buildBodyFunction4() {
  return Scaffold(
appBar: AppBar(
      title: Text("Text 组件使用分析"),),
    body: Container(child: Center(
      child: Column(children: [
        Text("天高水长,小编与你奋斗每一天,
              Small make up and you struggle every day",
          style: TextStyle(
//字符之间的间隔
letterSpacing: 12.0,
//单词之间的间隔
wordSpacing: 32.0,
          ),),
      ]
      ),),),);
}
```

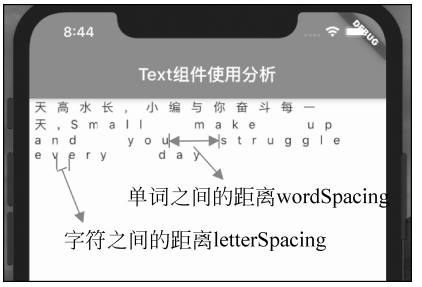

图 5-6　设置字符和单词间隔显示效果图

5.2.7　文字基线 textBaseline 分析

文本样式 TextStyle 中的 textBaseline 用来设置绘制文本时的当前文本基线，关于文本基线的分析如图 5-7 所示，在 Flutter 中使用代码如下：

```
Text("天高水长,小编与你奋斗每一天,Small make up and you struggle every day",
  style: TextStyle(
//设置文字绘制对齐的基线
textBaseline: TextBaseline.alphabetic
  ),),
```

图 5-7　文本绘制基线效果图

在 Flutter 的取值与描述代码如下：

```
//A horizontal line used for aligning text.
enumTextBaseline {
//The horizontal line used to align the bottom of glyphs for alphabetic characters.
  alphabetic,

//The horizontal line used to align ideographic characters.
  ideographic,
}
```

所以在实际开发中，这里的 textBaseline 作用不大。

5.2.8　装饰 decoration 分析

在英文中 decoration 的意思为装饰，在 Text 的 TextStyle 中，其实就是在文字上绘制一些线，例如下画线和删除线，默认的 TextStyle 是不使用装饰的，代码如下：

```
//5.2.2 /lib/code3/main_data70.dart
//文本显示组件 Text 样式 TextStyle 中 decoration
Widget buildBodyFunction() {
  return Scaffold(
appBar: AppBar(
        title: Text(
            "Text 文本样式"
        ),),
      body: Container(
        margin: EdgeInsets.all(30.0),
        child: Column(children: [
          Text("inherit: true",
            style: TextStyle(
//文字的大小
fontSize: 22,
//设置无装饰样式
                decoration: TextDecoration.none
            ),),
```

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

这里 decoration 配置的是枚举 TextDecoration 类型，取值效果如图 5-8 所示，源码说明如下：

```
    //无样式
    static const TextDecoration none = TextDecoration._(0x0);

    //在每行文本下面画一条线
    static const TextDecoration underline = TextDecoration._(0x1);

    //在每行文本上面画一条线
    static const TextDecoration overline = TextDecoration._(0x2);

    //在每行文本上画一条线
    static const TextDecoration lineThrough = TextDecoration._(0x4);
```

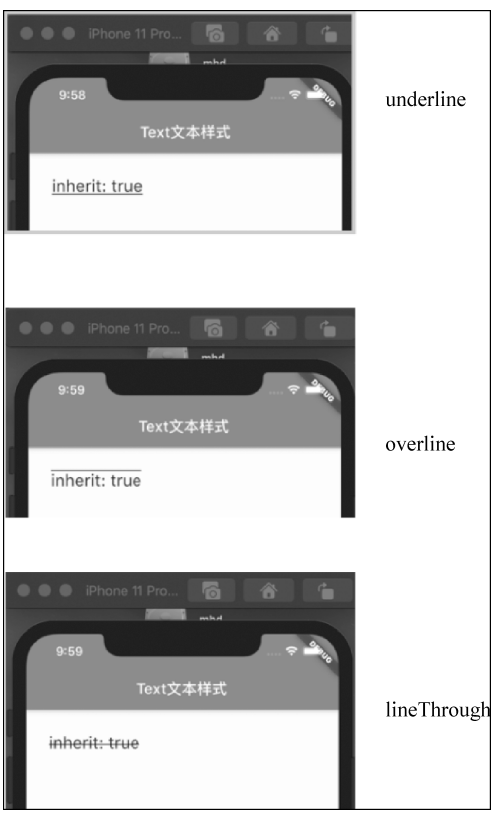

图 5-8　TextDecoration 各取值效果图

如图 5-9 所示效果，可以通过 TextDecoration 提供的方法 combine 同时设置文本的上画线 overline 与下画线 underline，或者如图 5-8 所示的任意的组合效果，代码如下：

```
//5.2.2 /lib/code3/main_data70.dart
//文本显示组件 Text 样式 TextStyle 中 decoration 组合多组装饰样式
Widget buildBodyFunction2() {
  return Scaffold(
appBar: AppBar(
        title: Text(
            "Text 文本样式"
        ),),
      body: Container(
        margin: EdgeInsets.all(30.0),
        child: Column(children: [
          Text("inherit: true",
            style: TextStyle(
//文字的大小
fontSize: 22,
//设置文本的装饰样式
//TextDecoration.combine 接收的是一个 List 参数，用来组合多个样式
                decoration: TextDecoration.combine([
TextDecoration.overline,
TextDecoration.underline,
                ]),
            ),),
        ]
        ),));
}
```

图 5-9　TextDecoration 组合使用效果图

属性 TextDecoration 设置的装饰线还可以通过各种 decoration 相关属性来配置颜色等样式，如图 5-10 所示中文字中间的删除线为红色的双线样式，代码如下：

```
//5.2.2 /lib/code3/main_data70.dart
//文本显示组件 Text 样式 TextStyle 中 decoration 的样式
```

```
Widget buildBodyFunction() {
  return Scaffold(
appBar: AppBar(title: Text("Text 文本样式"),),
      body: Container(
        margin: EdgeInsets.all(30.0),
        child: Column(children: [
          Text("inherit: true",
            style: TextStyle(
//文字的大小
fontSize: 25,
//设置中间通过装饰样式
                decoration: TextDecoration.lineThrough,
//配置中间通过线为红色
decorationColor: Colors.red,
//双线样式
decorationStyle: TextDecorationStyle.double,
            ),),
        ]
        ),));
}
```

图 5-10　TextDecoration 设置样式效果图

TextDecorationStyle 用来配置装饰线的样式，可取值如表 5-3 所示，效果对比如图 5-11 所示。

表 5-3　TextDecorationStyle 属性取值说明

取　值	说　明
TextDecorationStyle. solid	单线，实心线
TextDecorationStyle. double	双线，实心线
TextDecorationStyle. dotted	单线，点点虚线
TextDecorationStyle. dashed	单线，短横线，虚线
TextDecorationStyle. wavy	单线，连续波浪线

如图 5-12 所示效果，可通过 TextStyle 的属性 decorationThickness 来设置装饰线的粗细，它接收的是 double 类型，使用代码如下：

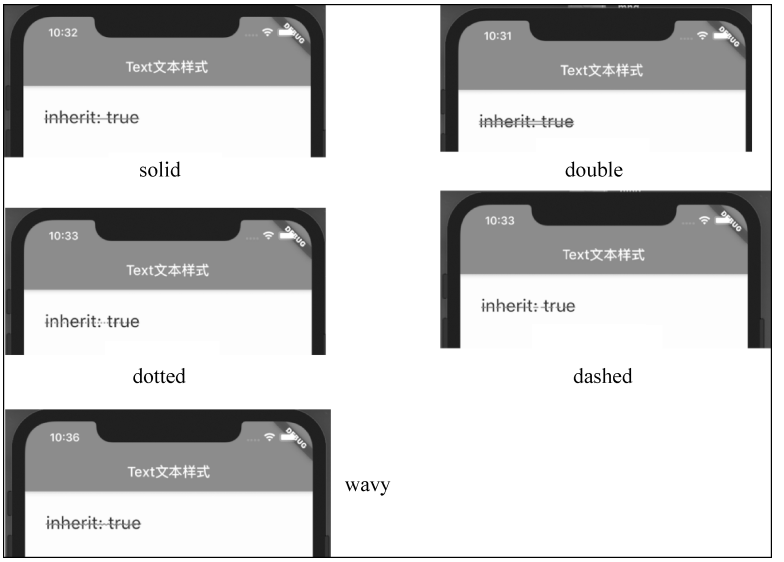

图 5-11　TextDecorationStyle 取值对比

```
//5.2.9 /lib/code3/main_data70.dart
//文本显示组件 Text 样式 TextStyle 中 decoration 的粗线
Widget buildBodyFunction3() {
  return Scaffold(
appBar: AppBar(
        title: Text(
            "Text 文本样式"
        ),),
      body: Container(
        margin: EdgeInsets.all(30.0),
        child: Column(children: [
          Text("inherit: true",
            style: TextStyle(
//文字的大小
fontSize: 25,
//设置中间通过装饰样式
              decoration: TextDecoration.lineThrough,
//配置中间通过线为红色
decorationColor: Colors.red,
//双线样式波浪线
decorationStyle: TextDecorationStyle.wavy,
//加粗
decorationThickness: 3,
            ),),
        ]
        ),));}
```

图 5-12　TextDecoration 的粗细设置

5.2.9　自定义字体 fontFamily 配置

在实际项目开发中，使用字体配置，可实现多样式美化效果，如图 5-13 所示，使用圆滑的字体样式，一般这样的字体在手机中是不支持的，所以需要在应用内配置字体。

在 Flutter 项目中，使用自定义的字体，首先需要配置自定义字体，第 1 步，导入字体样式文件 ttf 或者 otf 文件，在 Flutter 项目的根目录下，创建 fonts 文件夹，然后将字体文件复制到 fonts 文件夹，如图 5-14 所示。

图 5-13　自定义字体使用效果

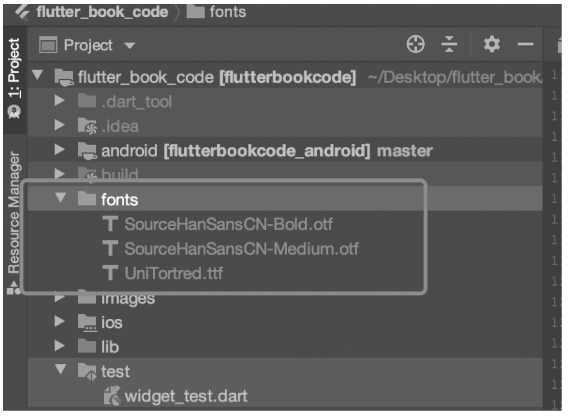

图 5-14　配置自定义字体项目目录结构图

第 2 步，在 Flutter 项目配置文件 pubspec.yaml 中配置 fonts 文件夹中已导入的字体，代码如下：

```
//pubspec.yaml 配置文件
#配置图片的目录结构
  assets:
    - images/2.0/
    - images/3.0/
#配置字体的目录结构
  fonts:
```

```
    - family: Schyler
      fonts:
        - asset: fonts/SourceHanSansCN-Bold.otf
    - family: Medium
      fonts:
        - asset: fonts/SourceHanSansCN-Medium.otf

    - family: UniTortred
      fonts:
        - asset: fonts/UniTortred.ttf
```

在 pubspec.yaml 文件中，fonts 节点用来配置自定义字体文件，-family 指定的是在 TextStyle 中 fontFamily 引用的名字，-asset 配置的是对应字体文件的路径与文件名称，如果需要配置多个自定义字体，可以依次向下排列。

在 pubspec.yaml 文件中，一定要注意空格的缩进，不能多也不能少，如图 5-15 所示。在 Text 组件中通过 fontFamily 属性引用 pubspec.yaml 文件中配置的 family 所对应的别名就可使用自定义字体，代码如下：

```
//5.2.10 /lib/code3/main_data70.dart
//文本显示组件 Text 样式 TextStyle
//字体 fontFamily 配置
Widget buildBodyFunction4() {
  return Scaffold(
appBar: AppBar(
        title: Text(
            "Text 文本样式"
        ),),
      body: Container(
        margin: EdgeInsets.all(30.0),
        child: Column(children: [
          Text("UniTortred 字体",
            style: TextStyle(
//文字的大小
fontSize: 25,
//引用圆滑的自定义字体
fontFamily: "UniTortred"
            ),),
        ]
        ),));
}
```

图 5-15　配置文件自定义字体配置分析说明

5.2.10　字体列表 fontFamilyFallback 配置

对于 fontFamilyFallback 属性来说，它接收的配置是一个 List < String >，这个 List 定义的是应用程序内支持的字体配置，配置代码如下：

```
Text("Regular 字体",
  style: TextStyle(
//文字的大小
fontSize: 25,
//引用圆滑的自定义字体
fontFamily: "Regular",
//支持的字体列表
fontFamilyFallback: < String >['Ubuntu', 'Cantarell', 'DejaVu Sans', 'Liberation Sans', 'Arial
'],
  ),),
```

也就是说当 fontFamily 中配置了指定加载的字体时，应用程序会优先加载这个字体，如果找不到或者加载失败，那么系统会加载 fontFamilyFallback 列表中配置的字体，如果还是找不到对应的字体文件或者加载失败，就使用当前系统默认的字体样式。

对 fontFamilyFallback 的官方源码解析如下：

```
//The ordered list of font families to fall back on when a glyph cannot be
//found in a higher priority font family.
//The value provided in [fontFamily] will act as the preferred/first font
//family that glyphs are looked for in, followed in order by the font families
//in [fontFamilyFallback]. If all font families are exhausted and no match
//was found, the default platform font family will be used instead.
//
//When [fontFamily] is null or not provided, the first value in [fontFamilyFallback]
...  //省略
//大致的意思就是如果描述系统在优先级高的 family 配置的字体中找不到时，会优先加载
//fontFamilyFallback 这个列表中配置的字体
```

5.2.11 字体阴影 shadows 配置

在 TextStyle 中，通过 shadows 配置字体文本的阴影，如图 5-16 所示，代码如下：

```
//5.2.11 /lib/code3/main_data70.dart
//文本显示组件 Text 样式 TextStyle
//阴影样式配置
Widget buildBodyFunction5() {
  return Scaffold(
appBar: AppBar(title: Text("Text 文本样式"),),
      body: Container(
        margin: EdgeInsets.all(30.0),
        child: Column(children: [
          Text("Regular 字体",
            style: TextStyle(
//文字的大小
fontSize: 25,
//引用圆滑的自定义字体
fontFamily: "Regular",
//配置字体阴影
             shadows:[
BoxShadow(
//阴影的颜色
                color: Color(0xffff0000),
//偏移量
                offset: Offset(2,1),
//模糊度
spreadRadius: 5.0,
               )
             ],
            ),),
        ]
        ),));
}
```

图 5-16 字体阴影配置效果图

在这里的 shadows 接收的是一个 List < ui. Shadow >集合，对于图 5-16 所示效果的代码片段 buildBodyFunction5 中使用了 BoxShadow，它继承于 ui. Shadow，对于阴影偏移量 offset，构造 Offset(x，y)，x 设置的是阴影相对于当前位置在 x 轴上的偏移量，y 设置的是阴影相对于当前位置在 y 轴上的偏移量。

5.2.12　文本高度 height 配置

对于 TextStyle 中的 height 属性来说，可以简单理解为文本高度，这里设置的 height 可影响实际中 Text 文本的高度，实际中 Text 文本的高度等于 height 与 fontSize 的乘积。不指定时，Text 在显示多行文本时，有默认的文本高度，当指定为 1.0 时，文本的高度就与 fontSize 的大小一致，下面会详细分析。

在源码中描述如下：

```
//The height of this text span, as a multiple of the font size.
//
//When [height] is null or omitted, the line height will be determined
//by the font's metrics directly, which may differ from the fontSize.
//When [height] is non - null, the line height of the span of text will be a
...  //省略
//![Text height comparison diagram](https://flutter.github.io/assets - for - api - docs/
assets/painting/text_height_comparison_diagram.png)
//
//See [StrutStyle] for further control of line height at the paragraph level.
final double height;
```

读者可以注意到，在案例中使用的 Text 的内容文本大部分是单行的。如图 5-17 所示，是 Text 显示多行文本的效果，对应代码如下：

```
//5.2.11 /lib/code3/main_data70.dart
//文本显示组件 Text 样式 TextStyle
//多行文本显示
Widget buildBodyFunction6() {
  return Scaffold(
appBar: AppBar(
        title: Text(
            "Text 文本样式"
        ),),
      body: Container(
        margin: EdgeInsets.all(30.0),
        child: Column(children: [
          Text("人生就有许多这样的奇迹，看似比登天还难的事，
                        有时轻而易举就可以做到，其中的差别就在于非凡的信念",
            style: TextStyle(
```

```
    //文字的大小
    fontSize: 25,
                ),),
            ]
            ),));
    }
```

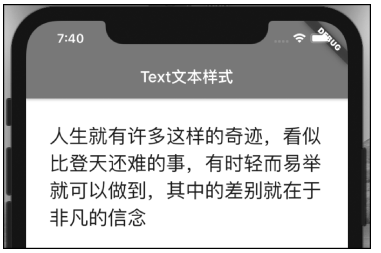

图 5-17　多行文本显示视图效果

在图 5-17 的效果中，没有指定 TextStyle 中的 height。当指定 height 为 1.0 时，效果如图 5-18 右图所示。

图 5-18　height 为 null 与 1.0 的效果对比图

如图 5-19 所示，当 height 没有指定时，默认值为 null，实际文本的高度为左侧的 Font metrics。当指定 height 时，实际文本的高度为右侧的 EM-square 标准，也就是指定 height 的值后，文本的高度为 height 与 FontSize 的乘积。

图 5-19　字体阴影配置效果图

如图 5-20 所示分析效果，是源自于谷歌官方提供的文字 height 距离分析图。在使用 TextStyle 时，当指定的字体大小为 50 并且指定 height 为 1.0 时，当前文本的高度为

50.0px。当不指定 height 时，当前文本高度为 59.0px，结合图 5-8 中提到文本绘制的各种基线，在这里前两者相差 9.0px 是 top 基线到文本之间的间隔。指定 height 后，文本实际的高度为 height 与 fontSize 的乘积。

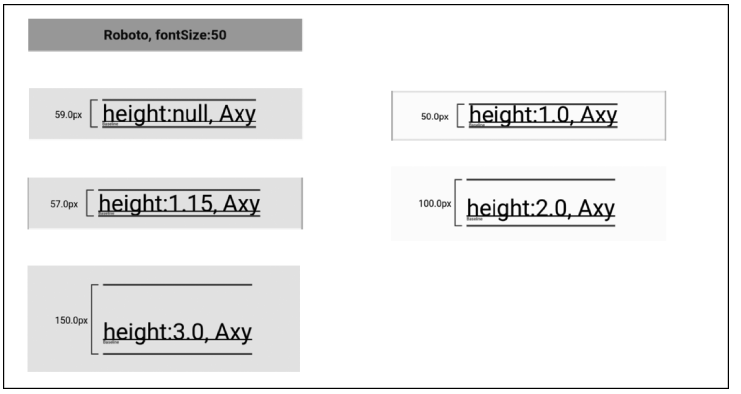

图 5-20　指定不同 height 值显示文本高度效果对比图

5.3　Text 中文字对齐方式

在 Text 组件中通过 textAlign 配置文字的对齐方式，取值类型为 TextAlign 枚举类型，可取值如表 5-4 所示。

表 5-4　文本对齐方式取值简述

类　别	全　称
TextAlign.left	左对齐
TextAlign.right	右对齐
TextAlign.center	居中对齐
TextAlign.justify	拉伸结束文本行以填充容器的宽度，即使用 decorationStyle 才生效
TextAlign.start	开始方向对齐
TextAlign.end	结束方向对齐

如图 5-21 所示效果，为程序默认的文字对齐方式，在这里显示的是左对齐，其实默认的文字对齐方式是 TextAlign.start，如图 5-22 所示是设置文字居中对齐的效果，textAlign 的基本使用代码如下：

```
//5.3 /lib/code3/main_data71.dart
//文本对齐方式 textAlign
Widget buildBodyFunction() {
  return Scaffold(
```

```
appBar: AppBar(
        title: Text(
            "Text 文本样式"
        ),),
      body: Container(
        margin: EdgeInsets.all(30.0),
        child: Column(children: [
          Text("人生就有许多这样的奇迹,看似比登天还难的事,
                        有时轻而易举就可以做到,其中的差别就在于非凡的信念",
//居中对齐
textAlign: TextAlign.center,
            style: TextStyle(
//文字的颜色
foreground:Paint()..color = Colors.blue,
//文字的大小
fontSize: 16,
            ),),
        ]
        ),));
}
```

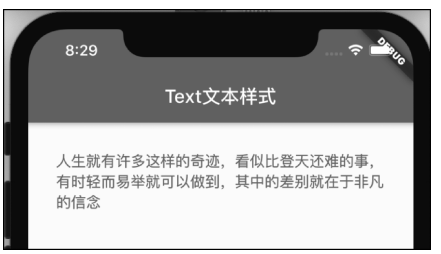

图 5-21　文本的默认对齐方式

图 5-22　文本的居中对齐方式

在对齐方式中，left、center、right 都容易理解，分别为左对齐、居中、右对齐。对于 TextAlign. start 与 TextAlign. end 来讲，需要结合文本的绘制方向 textDirection 来分析，textDirection 有两个取值，一个是 TextDirection. ltr 从左向右，另一个是 TextDirection. rtl 从右向左，在中文环境下，文本绘制方向是从左向右，此时 textDirection 的方向就是 ltr 值，这种情况与 left 左对齐效果是一致的，如图 5-23 所示。

图 5-23　默认情况下的文字方向与文本对齐

图 5-23 所示效果代码如下：

```
//5.3 /lib/code3/main_data71.dart
//文本对齐,文字方向从左向右以开始位置对齐
Widget buildBodyFunction3() {
  return Scaffold(
...  //省略
      body: Container(
        margin: EdgeInsets.all(30.0),
        child: Column(children: [
          Container(
//设置 Text 的宽度
            width: MediaQuery.of(context).size.width,
            child: Text("人生就有许多",
textDirection: TextDirection.ltr,
//开始位置对齐
textAlign: TextAlign.start,
              style: TextStyle(
//文字的颜色
                foreground: Paint()..color = Colors.blue,
//文字的大小
fontSize: 16,
              ),),),
        ]
        ),));
}
```

图 5-24　文字方向从左向右以 end 方式对齐

图 5-24 所示效果代码如下：

```
//5.3 /lib/code3/main_data71.dart
//文本对齐,文字方向从左向右以结束位置对齐
Widget buildBodyFunction4() {
  return Scaffold(
...  //省略
```

```
      body: Container(
        margin: EdgeInsets.all(30.0),
        child: Column(children: [
          Container(
//设置 Text 的宽度
            width: MediaQuery.of(context).size.width,
            child: Text("人生就有许多",
textDirection: TextDirection.ltr,
//结束位置对齐
textAlign: TextAlign.end,
              style: TextStyle(
//文字的颜色
                foreground: Paint()..color = Colors.blue,
//文字的大小
fontSize: 16,
              ),),),
        ]
        ),));
}
```

图 5-25 所示代码如下：

```
//5.3 /lib/code3/main_data71.dart
//文本对齐,文字方向从右向左以开始位置对齐
Widget buildBodyFunction5() {
  return Scaffold(
...  //省略
      body: Container(
        margin: EdgeInsets.all(30.0),
        child: Column(children: [
          Container(
//设置 Text 的宽度
            width: MediaQuery.of(context).size.width,
            child: Text("人生就有许多",
textDirection: TextDirection.rtl,
//开始位置对齐
textAlign: TextAlign.start,
              style: TextStyle(
//文字的颜色
                foreground: Paint()..color = Colors.blue,
//文字的大小
fontSize: 16,
              ),),),
        ]
        ),));
}
```

图 5-25　文字方向从右向左以 start 方式对齐

图 5-26 所示效果代码如下：

```
//5.3 /lib/code3/main_data71.dart
//文本对齐,文字方向从右向左以结束位置对齐
Widget buildBodyFunction6() {
  return Scaffold(
...  //省略
      body: Container(
        margin: EdgeInsets.all(30.0),
        child: Column(children: [
          Container(
//设置 Text 的宽度
            width: MediaQuery.of(context).size.width,
            child: Text("人生就有许多",
textDirection: TextDirection.rtl,
//结束位置对齐
textAlign: TextAlign.end,
             ,),),
        ]
        ),));
}
```

图 5-26　文字方向从右向左以 end 方式对齐

5.3.1　文字过长显示省略号

在实际项目开发中,常常会遇到在设计中最多只显示一行字体,当文字过长时,在末尾

显示省略号，这样在视觉上会达成一个好的效果。

在 Android 项目中文字显示 TextView 中添加两个配置就可以达到这样的设计效果，代码如下：

```
android:ellipsize = "end"
android:lines = "1"
```

在 iOS 中将文本显示 UILabel 设定显示的行数 numberOfLines 后，再设置 UILabel 的 lineBreakMode 属性为 NSLineBreakByTruncatingTail，也可以达到预期的效果。

在 Flutter 中，则可通过设置 Text 组件的 overflow 与 maxLines 来达到预期效果。如图 5-27 所示为在限定 Text 只显示一行时，配置 overflow 后的效果，对应代码如下：

```
//5.3 /lib/code3/main_data71.dart
//文本超出后显示省略号
  Widget buildBodyFunction7() {
    return Scaffold(
appBar: AppBar(
          title: Text(
              "Text 文本样式"
          ),),
        body: Container(
          margin: EdgeInsets.all(30.0),
          child: Column(children: [
            Container(
//设置 Text 的宽度
              width: MediaQuery.of(context).size.width,
              height: 20,
              child: Text("人生就有许多这样的奇迹，看似比登天还难的事，
有时轻而易举就可以做到，其中的差别就在于非凡的信念",
                //设置省略号
                overflow: TextOverflow.ellipsis,
                //设置最大显示行数
              maxLines: 1,
                style: TextStyle(
//文字的颜色
                  foreground: Paint()..color = Colors.blue,
//文字的大小
fontSize: 18,
                ),),),
          ]
    ),));
   }
```

图 5-27　文字超出字数后显示省略号效果图

TextOverflow 是一个枚举类型，其各取值类型效果如图 5-28 所示，代码如下：

```
enumTextOverflow {
//Clip the overflowing text to fix its container.
  //超出宽度的文本部分直接裁剪
  clip,
//Fade the overflowing text to transparent.
  //超出宽度的文本部分以透明渐变方式结尾
  fade,
//Use an ellipsis to indicate that the text has overflowed.
  //超出宽度的文本内容部分以省略号的方式结尾
  ellipsis,
//Render overflowing text outside of its container.
  //超出的部分照常显示，效果与 clip 的效果差不多，只不过 clip 直接裁剪掉了多余的部分
  visible,
}
```

图 5-28　TextOverflow 效果示意图

5.3.2　文字自动换行设置

如图 5-29 所示，在 Flutter 中可以通过设置 Text 组件的 softWrap 属性来控制是否自动换行。当 softWrap 为 false 时，Text 中的文本超出父组件配置的宽度时，不会自动换行，而会直接被裁剪掉。

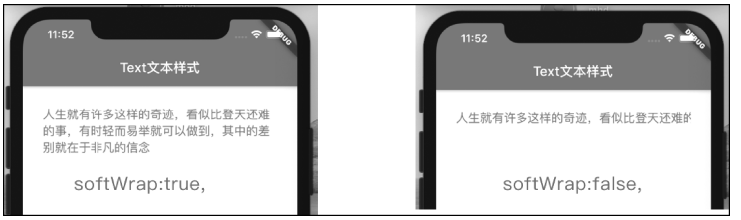

图 5-29　softWrap 效果示意图

自动换行属性 softWrap 配置基本使用代码如下：

```
Text("",
//true 为自动换行,默认为 true
 //false 为不自动换行
softWrap:true,
),
```

在实际项目开发中，常会以如图 5-30 所示的布局效果进行开发，代码如下：

```
//5.3 /lib/code3/main_data71.dart
//常用布局
Widget buildBodyFunction9() {
  return Scaffold(
appBar: AppBar(
        title: Text(
            "Text 文本样式"
        ),),
      body: Container(
        margin: EdgeInsets.all(30.0),
        child: Row(children: [
          Text("个性签名：",style: TextStyle(fontWeight: FontWeight.w500),),
SizedBox(width: 14,),
          Text("人生就有许多这样的奇迹")
        ]
        ),));
}
```

图 5-30　常用的文本布局排列

图 5-30 中右侧的文本显示得比较少时，可以正常显示。当把右侧的文本内容加多后，即使设置了 softWrap 为 true 自动换行，在程序实际运行中，依然没有换行。当实际绘制的文本宽度超出了父组件所允许的宽度时，则会出现如图 5-31 所示的异常效果，Android Studio 的控制台输出的异常日志如下：

```
═══════ Exception caught by rendering library ═══════
The following assertion was thrown during layout:

A RenderFlex overflowed by 388 pixels on the right.

The relevant error-causing widget was:
  Row file://Users/.../code/flutter_book_code/flutter_book_code/lib/code4/main_data71.
dart:267:18
The overflowing RenderFlex has an orientation of Axis.horizontal.
The edge of the RenderFlex that is overflowing has been marked in the rendering with a yellow
and black striped pattern. This is usually caused by the contents being too big for the
RenderFlex.


Consider applying a flex factor (e.g. using an Expanded widget) to force the children of the
RenderFlex to fit within the available space instead of being sized to their natural size.
This is considered an error condition because it indicates that there is content that cannot be
seen. If the content is legitimately bigger than the available space, consider clipping it with
a ClipRect widget before putting it in the flex, or using a scrollable container rather than a
Flex, like a ListView.

The specific RenderFlex in question is: RenderFlex#2a2a0 relayoutBoundary=up2 OVERFLOWING
...  parentData: offset=Offset(30.0, 30.0) (can use size)
...  constraints: BoxConstraints(0.0<=w<=354.0, 0.0<=h<=736.0)
...  size: Size(354.0, 20.0)
...  direction: horizontal
...  mainAxisAlignment: start
...  mainAxisSize: max
...  crossAxisAlignment: center
...  textDirection: ltr
...  verticalDirection: down
◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤
◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤◢◤
```

图 5-31　文本超出限定宽度后异常

在异常日志信息中，也提示出了异常的原因描述及解决方案，日志中提示，可以使用flex布局来解决这种情况。在CSS中，2009年W3C提出了弹性布局flex这一概念，在Flutter中，flex布局理念与W3C的flex理念一致，与Android中的线性布局使用weight属性进行权重布局排列的理念一致。

如图5-32所示效果，使用弹性布局Flexible或者Expanded来解决显示多余文本的超出限制问题。

图5-32　使用弹性布局的解决方案

图5-32所示效果对应的代码如下：

```
//5.3 /lib/code3/main_data71.dart
//弹性布局解决文本超出限制宽度问题
Widget buildBodyFunction10() {
  return Scaffold(
appBar: AppBar(
        title: Text(
            "Text 文本样式"
        ),),
      body: Container(
        margin: EdgeInsets.all(30.0),
        child: Row(
//内容顶部对齐
crossAxisAlignment: CrossAxisAlignment.start,
            children: [
              Text("个性签名：",
                        style: TextStyle(fontWeight: FontWeight.w500),),
SizedBox(width: 14,),
//使用弹性布局
                Flexible(child: Text("人生就有许多这样的奇迹,看似比登天还难的事,有时轻而易举就可以做到,其中的差别就在于非凡的信念"),)
            ]
        ),));
}
```

5.3.3　弹性布局综述

对于弹性布局来讲，在 Flutter 中有 Flexible、Expanded、Spacer 这 3 个 Widget，Spacer 常用于调节两个 Widget 之间的间距。

Expanded 用在 Row、Column、Flex，它会限定在这些 Widget 的主轴方向上占满剩余的可用空间，如图 5-33 所示效果，Expanded 所修饰的内容区域的宽度会自动占满在水平方向上除标题文字以外的所有空间，代码如下：

```
//5.3 /lib/code3/main_data72.dart
//弹性布局 Expanded
Widget buildBodyFunction() {
  return Scaffold(
appBar: AppBar(
        title: Text(
            "Text 文本样式"
        ),),
      body: Container(
        margin: EdgeInsets.all(30.0),
        child: Row(children: [
          Text("标题",),
          Expanded(child: Container(color: Colors.grey,height: 20,),)
        ]
        ),));
}
```

图 5-33　Expanded 权重适配

如图 5-34 中，通过 Expanded 进行权重适配，使用 Row 的两个子 Widget 在水平方向上平分 Row 的宽度，对应代码如下：

```
//5.3 /lib/code3/main_data72.dart
//弹性布局 Expanded
Widget buildBodyFunction2() {
  return Scaffold(
appBar: AppBar(
```

```
          title: Text(
              "Text 文本样式"
          ),),
        body: Container(
          margin: EdgeInsets.all(30.0),
          child: Row(children: [
            Expanded(child: Text("标题",),),
            Expanded(child: Container(color: Colors.grey,height: 20,),)
          ]
          ),));
  }
```

图 5-34　Expanded 权重适配平分宽度

如图 5-35 所示，通过设置 Expanded 的 flex 属性按照比例分配 Row 在水平方向上的宽度，在这里，配置左侧文本的 Expanded 的 flex 为 1，右侧 Container 的 Expanded 的 flex 为 2，那么整体来讲，应用程序会把 Row 主方向也就是水平方向的宽度平均分成 3 份，然后根据 flex 配置的值再分配给每个 Expanded 对应的宽度，代码如下：

```
//5.3 /lib/code3/main_data72.dart
//弹性布局 Expanded
Widget buildBodyFunction3() {
  return Scaffold(
appBar: AppBar(
          title: Text(
              "Text 文本样式"
          ),),
        body: Container(
          margin: EdgeInsets.all(30.0),
          child: Row(children: [
//水平方向上的空间占 1 份
            Expanded(flex: 1,child: Text("标题",),),
//水平方向上的空间占 2 份
            Expanded(flex: 2,child: Container(color: Colors.grey,height: 20,),)
          ]
          ),));
  }
```

图 5-35　Expanded 权重分配

Flexible 允许子 Widget 弹性地使用它的空间，不必填满可用空间，它的构造函数代码如下：

```
const Flexible({
  Key key,
this.flex = 1,
this.fit = FlexFit.loose,
  @required Widget child,
}) : super(key: key, child: child);
```

Flexible 比 Expanded 多了一个配置 fit 属性。

5.4　富文本 RichText 组件的使用分析

在 Flutter 中，使用 RichText 实现多种文本风格的功能，类似 Android 中的 SpannableString/SpannableStringBuilder 为不同的文本片段指定不同的 span；类似 iOS 中的 NSMutableAttributedString，通过向里面添加文字样式，然后通过控件的 AttributedText 赋值显示。

在 Flutter 中，RichText 也有对应的 TextSpan，TextSpan 指定同一文本在不同区域的文字样式，如图 5-36 所示效果分析，对应的 RichText 基本使用代码如下：

```
//5.4 /lib/code3/main_data73.dart
//富文本的基本使用
RichText(
//文字区域
  text:
TextSpan(
      text: "登录即代表同意",style: TextStyle(color: Colors.grey),
      children: [
TextSpan(
              text: "《用户注册协议》", style: TextStyle(color: Colors.blue)
            ),
```

```
TextSpan(
                text: "与",style: TextStyle(color: Colors.grey),
            ),
TextSpan(
                text: "《隐私协议》", style: TextStyle(color: Colors.blue)
            )
      ]
  ),
),
```

图 5-36　RichText 富文本的基本使用

正如图 5-36 所示效果，一行文字通过一个 RichText 组件实现，这个 RichText 组件包含了 4 个 TextSpan 文本片段，TextSpan 的构造函数代码如下：

```
const TextSpan({
//显示的文本
this.text,
//可包含的子 Widget
this.children,
//当前文本片段 TextSpan 的片段
TextStyle style,
//手势识别
this.recognizer,
//标签内容
this.semanticsLabel,
}) : super(style: style,);
```

RichText 只有一个 text 属性配置它所需要显示的文本，而 TextSpan 的 text 属性用来配置当前文本片段显示的内容，它的属性 children 用来配置其他的 TextSpan 片段，可以理解为 TextSpan 可以无限地使用 TextSpan 嵌套下去。

TextSpan 的 recognizer 用来配置当前显示的文本片段的手势识别功能，在实际项目开发中，如图 5-36 的效果，常用于监听用户对文本片段的单击事件，代码如下：

```
//5.4 /lib/code3/main_data73.dart
//富文本单击事件添加
RichText(
//文字区域
  text:
TextSpan(
      text: "登录即代表同意",style: TextStyle(color: Colors.grey),
      children: [
TextSpan(
              text: "《用户注册协议》", style: TextStyle(color: Colors.blue),
              //单击事件
recognizer:TapGestureRecognizer()..onTap = (){
                print("单击用户协议");
              }
            ),
TextSpan(
                text: "与",style: TextStyle(color: Colors.grey),
            ),
TextSpan(
                text: "《隐私协议》", style: TextStyle(color: Colors.blue),
                //单击事件
recognizer:TapGestureRecognizer()..onTap = (){
                  print("单击隐私协议");
                }
            )
      ]
  ),
),
```

TapGestureRecognizer 与 iOS 中的 UITapGestureRecognizer 类似，与 Android 中的 OnGestureListener 手势监听类似，所以可以理解为在 Flutter 中 TapGestureRecognizer 也相当于设置了一个手势监听，因此建议在 Widget 页面创建时创建 TapGestureRecognizer 手势识别类，然后在页面销毁中注销 TapGestureRecognizer 手势监听，以此很好地控制资源的有效利用，代码如下：

```
//5.4 /lib/code3/main_data74.dart
//富文本中手势识别的综合使用
class _FirstPageState extends State<FirstPage> {

TapGestureRecognizer _registProtocolRecognizer;
TapGestureRecognizer _privacyProtocolRecognizer;
//生命周期函数页面创建时执行一次
  @override
```

```
  void initState() {
super.initState();
    //注册协议的手势
    _registProtocolRecognizer = TapGestureRecognizer();
    //隐私协议的手势
    _privacyProtocolRecognizer = TapGestureRecognizer();
  }
//生命周期函数页面销毁时执行一次
  @override
  void dispose() {
super.dispose();
//销毁
    _registProtocolRecognizer.dispose();
    _privacyProtocolRecognizer.dispose();
  }

  @override
  Widget build(BuildContext context) {
    return buildBodyFunction();
  }

//5.4 /lib/code3/main_data74.dart
//富文本中手势识别的综合使用
//在 TextSpan 中使用声明创建的手势识别监听
Widget buildBodyFunction() {
...  //省略

TextSpan(
       text: "登录即代表同意",style: TextStyle(color: Colors.grey),
       children: [
TextSpan(
             text: "《用户注册协议》", style: TextStyle(color: Colors.blue),
             //单击事件
             recognizer:_registProtocolRecognizer..onTap = (){
                       print("单击用户协议");
               }
             ),
TextSpan(
                  text: "与",style: TextStyle(color: Colors.grey),
             ),
TextSpan(
                   text: "《隐私协议》",
                       style: TextStyle(color: Colors.blue),
                    //单击事件
                   recognizer:_privacyProtocolRecognizer..onTap = (){
                              print("单击隐私协议");
```

```
                    }
                   )
                  ]
... //省略 }
}
```

5.5　富文本 RichText 使用案例

在实际项目开发中，一个常见的需求就是如果搜索框搜索出的列表显示内容中有与关键字相同的内容就使用高亮颜色显示，在 6.7.3 节搜索输入框使用案例会有详细的搜索框搜索逻辑使用分析，在这里实现将一段文本中的关键词筛选出来，然后高亮显示，代码如下：

```
//lib/code4/main_data401.dart
class RichTextTagPage extends StatefulWidget {
  @override
  _FirstPageStatecreateState() => _FirstPageState();
}
class _FirstPageState extends State<RichTextTagPage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
appBar: AppBar(
          title: Text("文本标签"),
        ),
        body: Container(
          margin: EdgeInsets.all(30.0),
//参数一为显示的文本段落
//参数二为筛选的关键词
          child:  RichTextTag("adadfafeaaefagcvae32455aa565eza","a"),
        ));
  }
}
```

运行效果如图 5-37 所示，搜索关键字为字符 a，在文本段落中检索出关键字，并使用高亮颜色蓝色标记。

图 5-37　RichText 富文本显示高亮关键字

这里将用于筛选高亮显示内容的功能封装到 RichTextTag 中，代码如下：

```
//lib/demo/rich_text_tag.dart
//富文本标签,关键词高亮显示
class RichTextTagextends StatelessWidget{

//显示的内容文本
  String contentText;
//关键词
  String keyWordText;

RichTextTag(this.contentText, this.keyWordText);

  @override
  Widget build(BuildContext context) {
    return buildKeyWordRichText(contentText,keyWordText);
  }

//包含关键词的内容高亮显示
//[title] 内容体
//[keyWord] 关键词
buildKeyWordRichText(String title, String keyWord) {
    List<TextSpan> spans = [];
spans.addAll(keyWordSpan(title, keyWord));
    return RichText(text: TextSpan(children: spans));
  }
keyWordSpan(String word, String keyWord) {
    List<TextSpan> spans = [];
    if (word == null || word.length == 0) return spans;
//按关键词分隔
    List<String> strings = word.split(keyWord);
    for (int i = 0; i<strings.length; i++) {
//关键词部分
      if ((i + 1) % 2 == 0) {
        spans
            .add(TextSpan(text: keyWord, style: TextStyle(color: Colors.blue)));
      }
//普通部分
      String value = strings[i];
      if (value != null &&value.length> 0) {
spans.add(TextSpan(text: value, style: TextStyle(color: Colors.grey)));
      }
    }
    return spans;
  }
}
```

笔者已将 RichTextTag 封装在依赖库 flutter_tag_layout 中，在实际项目开发中，读者可直接添加此依赖库并使用，代码如下：

```
flutter_tag_layout: ^0.0.3
```

在使用的地方导入依赖包的方式如下：

```
import 'package:flutter_tag_layout/flutter_tag_layout.dart';
```

然后就可以直接使用 RichTextTag 组件。

5.6　文本标签

在实际项目开发中，一个常见的需求就是在搜索框下方会以文本标签的方式展示最近搜索的关键词记录，文本标签如图 5-38 所示。

图 5-38　流式布局展示文本标签

5.6.1　文本标签构建

一个小的文本标签由两部分组成，一部分用于文本显示的 Text 组件，一部分用于装饰效果的 Container，在这里，笔者将这部分代码封装到 TextTagWidget 中，并且已添加至依赖库 flutter_tag_layout 中，读者在使用时可添加依赖库，如 5.6 节所讲的添加方法添加依赖库后即可直接使用。

TextTagWidget 继承于 StatefulWidget，并且会设置一些默认使用的样式，代码如下：

```
//lib/demo/text_tag_widget.dart
class TextTagWidgetextends StatefulWidget{
//显示的文本
  String text;
//外边距
```

```
EdgeInsets margin;
//内边距
EdgeInsets padding;
//文本显示样式
TextStyletextStyle;
//标签背景颜色
  Color backgroundColor;
//标签边框颜色
  Color borderColor;
//标签边框圆角
  double borderRadius;

TextTagWidget(this.text,
      {this.margin = const EdgeInsets.all(4),
this.padding =
          const EdgeInsets.only(left: 6, right: 6, top: 3, bottom: 3),
this.textStyle,
this.backgroundColor,
this.borderColor,
this.borderRadius = 20.0}) {
//定义边框的颜色
    if (this.borderColor == null) {
//当边框颜色没有指定时取用背景颜色
      if (this.backgroundColor != null) {
this.borderColor = this.backgroundColor;
      } else {
//当背景颜色没有指定时取随机生成颜色
this.borderColor = getRandomColor();
      }
    }
//当没有指定文本样式时
    if (this.textStyle == null) {
//默认文本颜色与边框颜色一致
      Color textColor = this.borderColor;
//当指定了背景颜色时
//使用文本颜色为白色
      if (backgroundColor != null) {
textColor = Colors.white;
      }
//创建使用的文本样式
this.textStyle = TextStyle(fontSize: 12, color: textColor);
    }
//当背景颜色未指定时
//设置为透明色
    if (this.backgroundColor == null) {
this.backgroundColor = Colors.transparent;
    }
  }
```

```
  @override
  _TestPageStatecreateState() => _TestPageState();
}
```

然后构建具体的文本标签，在这里使用容器 Container 将文本 Text 包裹起来，代码如下：

```
//lib/code/main_data.dart
class _TestPageState extends State<TextTagWidget> {
  @override
  Widget build(BuildContext context) {
//构建边框装饰
    return Container(
      margin: widget.margin,
      padding: widget.padding,
      decoration: BoxDecoration(
          color: widget.backgroundColor,
borderRadius: BorderRadius.all(Radius.circular(widget.borderRadius)),
          border: Border.all(color: widget.borderColor)),
      child: buildTextWidget(),
    );
  }

//构建文本
  Text buildTextWidget() {
    return Text(
widget.text,
      style: widget.textStyle,
textAlign: TextAlign.center,
    );
  }
}
```

最后直接使用 TextTagWidget 实现文本标签效果，如图 5-39 所示效果为默认生成的样式，文本颜色与边框颜色是随机生成的，当然也可以通过 borderColor 来指定边框的颜色。

图 5-39　默认样式的文本标签

图 5-39 对应的代码如下：

```
//lib/code4/main_data401.dart
//默认样式的文本标签
buildTextTagWidget() {
  return TextTagWidget("文本标签");
}
```

图 5-40 是设置了背景颜色的文本标签，此时默认的文本颜色为白色。

图 5-40　设置了背景颜色的文本标签

图 5-40 对应的代码如下：

```
//lib/code4/main_data401.dart
//指定了背景颜色的文本标签
buildTextTagWidget2() {
  return TextTagWidget(
    "文本标签",
backgroundColor: Colors.brown,
  );
}
```

当分别指定边框颜色 borderColor 与背景颜色 backgroundColor 时，标签同时显示背景与边框颜色。

当需要指定标签中显示文本的颜色时，只需要重新指定 textStyle 属性并重新配置 TextStyle 即可。

5.6.2　文本标签结合流式布局使用

如图 5-38 所示就是流式布局 Wrap 结合文本标签实现的效果，在实际项目开发中，只需要将这个效果放到搜索框的下方，然后把每次使用的关键词数据保存在本地磁盘中，这样就可以实现常见的搜索记录流式布局展示效果，代码如下：

```
//lib/code4/main_data402.dart
class TextWarpTagPage extends StatefulWidget {
  @override
  _FirstPageStatecreateState() => _FirstPageState();
}

class _FirstPageState extends State<TextWarpTagPage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
appBar: AppBar(
          title: Text("文本标签"),
        ),
        body: Container(
          margin: EdgeInsets.only(top: 30.0, left: 10, right: 10),
```

```
//流式布局
          child: Wrap(
//横向每个标签之间的距离
              spacing: 8.0,
//纵向每个标签之间的距离
runSpacing: 8.0,
//子标签
              children: buildItemWidgetList()),
        ),);
  }

//文本标签集合
  List < String > tagList = ["文本标签", "测试", "这是什么", "早上好","吃饭", "再来一次"];
//构建文本标签数据
buildItemWidgetList() {
    List < Widget > itemWidgetList = [];
    for (var i = 0; i < tagList.length; i++) {
      var str = tagList[i];
itemWidgetList.add(TextTagWidget(" $ str"));
    }
    return itemWidgetList;
  }
}
```

5.7　AnimatedDefaultTextStyle 的使用分析

AnimatedDefaultTextStyle 通过动画过渡的方式切换文本的显示样式，效果如图 5-41 所示，当单击切换样式按钮时，显示的文本样式会以动画过渡的方式来切换。

图 5-41　AnimatedDefaultTextStyle 的基本使用效果图

图 5-41 所示的效果对应的 AnimatedDefaultTextStyle 基本使用代码如下：

```
//5.7 /lib/code4/main_data404.dart
//构建动画样式组件
```

```
AnimatedDefaultTextStylebuildAnimatedDefaultTextStyle() {
  return AnimatedDefaultTextStyle(
//设置 Text 中的文本样式
//每当样式有改变时会以动画的方式过渡切换
    style: isSelected
        ? TextStyle(
fontSize: 50, color: Colors.red, fontWeight: FontWeight.bold)
        : TextStyle(
fontSize: 24.0, color: Colors.black, fontWeight: FontWeight.w100),
//动画切换的时间
    duration: const Duration(milliseconds: 200),
//动画执行插值器
    curve: Curves.bounceInOut,
//文本对齐方式
textAlign: TextAlign.start,
//文本是否应该在软换行符处换行
softWrap: true,
//超过文本行数区域的裁剪方式
//设置为省略号
    overflow: TextOverflow.ellipsis,
//最大显示行数
maxLines: 1,
//每当样式有修改而触发动画时
//动画执行结束的回调
onEnd: () {
      print("动画执行结束");
    },

//文本组件
    child: Text("Hello, Flutter"),
  );
}
```

通过一个按钮来动态修改 isSelected 的值，从而触发修改文本样式的切换动画过渡效果，完整代码如下：

```
//5.7 /lib/code4/main_data404.dart
//[AnimatedDefaultTextStyle] 的使用分析
class AnimatedTextStylePage extends StatefulWidget {
  @override
  _FirstPageStatecreateState() => _FirstPageState();
}

class _FirstPageState extends State<AnimatedTextStylePage> {
  @override
  Widget build(BuildContext context) {
```

```
    return buildBodyFunction();
  }

  bool isSelected = false;

//5.7 /lib/code4/main_data404.dart
//文本显示组件 Text
  Widget buildBodyFunction() {
    return Scaffold(
appBar: AppBar(
        title: Text("动画样式"),
      ),
      body: Container(
        padding: EdgeInsets.all(16),
        child: Column(
          children: <Widget>[
//动画样式组件
buildAnimatedDefaultTextStyle(),
SizedBox(
              height: 55,
            ),
FlatButton(
              child: Text("切换样式"),
onPressed: () {
setState(() {
isSelected = !isSelected;
                });
              },
            )
          ],
        ),
      ),
    );
  }
}
```

小结

在本章中详细分析了文本组件 Text 的各种属性配置与样式设置，在实际项目开发中经常会遇到 Text 文字的自适应包裹，时常会遇到超出边界的情况，需要组件 Column 或者 Row 来解决，灵活组合应用样式配置与交互设置，可开发出令人舒适的视觉效果。