本章目录

布局类Widget

简介

布局类Widget都会包含一个或多个子widget，不同的布局类Widget对子widget排版(layout)方式不同。我们在前面说过Element树才是最终的绘制树，Element树是通过widget树来创建的（通过`Widget.createElement()`），widget其实就是Element的配置数据。Flutter中，根据Widget是否需要包含子节点将Widget分为了三类，分别对应三种Element，如下表：

Widget	对应的Element	用途
LeafRenderObjectWidget	LeafRenderObjectElement	Widget树的叶子节点，用于没有子节点的widget，通常基础widget都属于这一类，如Text、Image。
SingleChildRenderObjectWidget	SingleChildRenderObjectElement	包含一个子Widget，如：ConstrainedBox、DecoratedBox等
MultiChildRenderObjectWidget	MultiChildRenderObjectElement	包含多个子Widget，一般都有一个children参数，接受一个Widget数组。如Row、Column、Stack等

> 注意，Flutter中的很多Widget是直接继承自StatelessWidget或StatefulWidget，然后在`build()`方法中构建真正的RenderObjectWidget，如Text，它其实是继承自StatelessWidget，然后在`build()`方法中通过RichText来构建其子树，而RichText才是继承自LeafRenderObjectWidget。所以为了方便叙述，我们也可以直接说Text属于LeafRenderObjectWidget（其它widget也可以这么描述），这才是本质。读到这里我们也会发现，其实**StatelessWidget和StatefulWidget就是两个用于组合Widget的基类，它们本身并不关联最终的渲染对象（RenderObjectWidget）**。

布局类Widget就是指直接或间接继承(包含)MultiChildRenderObjectWidget的Widget，它们一般都会有一个children属性用于接收子Widget。我们看一下继承关系 Widget > RenderObjectWidget > (Leaf/SingleChild/MultiChild)RenderObjectWidget 。RenderObjectWidget类中定义了创建、更新RenderObject的方法，子类必须实现他们，关于RenderObject我们现在只需要知道它是最终布局、渲染UI界面的对象即可，也就是说，对于布局类Widget来说，其布局算法都是通过对应的RenderObject对象来实现的，所以读者如果对接下来介绍的某个布局类Widget原理感兴趣，可以查看其RenderObject的实现，而在本章中，为了让读者对布局类Widget有个快速的认识，所以我们不会深入到RenderObject的细节中区。在学习本章时，读者的重点是掌握不同布局类Widget的布局特点，具体原理和细节我们会在后面高级部分介绍。

线性布局Row和Column

所谓线性布局，即指沿水平或垂直方向排布子Widght。Flutter中通过Row和Column来实现线性布局，类似于Android中的LinearLayout控件。Row和Column都继承自Flex，我们将在弹性布局一节中详细介绍Flex。

主轴和纵轴

对于线性布局，有主轴和纵轴之分，如果布局是沿水平方，那么主轴就指是水平方向，而纵轴即垂直方向；如果布局沿垂直方向，那么主轴就是指垂直方向，而纵轴就是水平方向。在线性布局中，有两个定义对齐方式的枚举类MainAxisAlignment和CrossAxisAlignment，分别代表主轴对齐和纵轴对齐。

Row

Row可以在水平方向排列其子widget。定义如下：

```
Row({
  ...
  TextDirection textDirection,
  MainAxisSize mainAxisSize = MainAxisSize.max,
  MainAxisAlignment mainAxisAlignment = MainAxisAlignment.start,
  VerticalDirection verticalDirection = VerticalDirection.down,
  CrossAxisAlignment crossAxisAlignment =
CrossAxisAlignment.center,
  List<Widget> children = const <Widget>[],
})
```

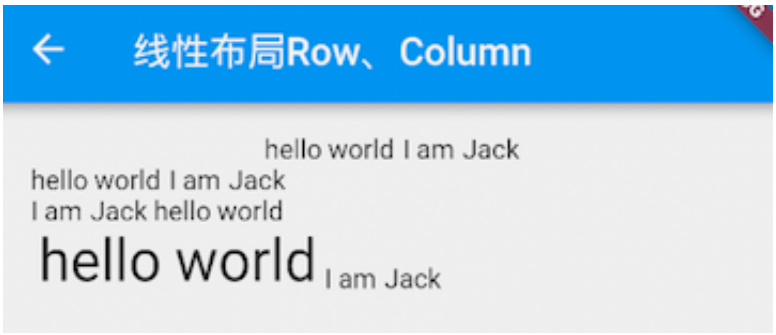
- textDirection：表示水平方向子widget的布局顺序(是从左往右还是从右往左)，默认为系统当前Locale环境的文本方向(如中文、英语都是从左往右，而阿拉伯语是从右往左)。
- mainAxisSize：表示Row在主轴(水平)方向占用的空间，默认是`MainAxisSize.max`，表示尽可能多的占用水平方向的空间，此时无论子widgets实际占用多少水平空间，Row的宽度始终等于水平方向的最大宽度；而`MainAxisSize.min`表示尽可能少的占用水平空间，当子widgets没有占满水平剩余空间，则Row的实际宽度等于所有子widgets占用的的水平空间；
- mainAxisAlignment：表示子Widgets在Row所占用的水平空间内对齐方式，如果mainAxisSize值为`MainAxisSize.min`，则此属性无意义，因为子widgets的宽度等于Row的宽度。只有当mainAxisSize的值为`MainAxisSize.max`时，此属性才有意义，`MainAxisAlignment.start`表示沿textDirection的初始方向对齐，如textDirection取值为`TextDirection.ltr`时，则`MainAxisAlignment.start`表示左对齐，textDirection取值为`TextDirection.rtl`时表示从右对齐。而`MainAxisAlignment.end`和`MainAxisAlignment.start`正好相反；`MainAxisAlignment.center`表示居中对齐。读者可以这么理解：textDirection是mainAxisAlignment的参考系。
- verticalDirection：表示Row纵轴（垂直）的对齐方向，默认是`VerticalDirection.down`，表示从上到下。
- crossAxisAlignment：表示子Widgets在纵轴方向的对齐方式，Row的高度等于子Widgets中最高的子元素高度，它的取值和MainAxisAlignment一样(包含`start`、`end`、`center`三个值)，不同的是crossAxisAlignment的参考系是verticalDirection，即verticalDirection值为`VerticalDirection.down`时`crossAxisAlignment.start`指顶部对齐，verticalDirection值为`VerticalDirection.up`时，`crossAxisAlignment.start`指底部对齐；而`MainAxisAlignment.end`和`MainAxisAlignment.start`正好相反；
- children： 子Widgets数组。

示例

请阅读下面代码，想象一下运行的结果：

```
Column(
  //测试Row对齐方式，排除Column默认居中对齐的干扰
  crossAxisAlignment: CrossAxisAlignment.start,
  children: <Widget>[
    Row(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        Text(" hello world "),
        Text(" I am Jack "),
      ],
    ),
    Row(
      mainAxisSize: MainAxisSize.min,
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        Text(" hello world "),
        Text(" I am Jack "),
      ],
    ),
    Row(
      mainAxisAlignment: MainAxisAlignment.end,
      textDirection: TextDirection.rtl,
      children: <Widget>[
        Text(" hello world "),
        Text(" I am Jack "),
      ],
    ),
    Row(
      crossAxisAlignment: CrossAxisAlignment.start,
      verticalDirection: VerticalDirection.up,
      children: <Widget>[
        Text(" hello world ", style: TextStyle(fontSize: 30.0),),
        Text(" I am Jack "),
      ],
    ),
  ],
);
```

运行结果：



解释：第一个Row很简单，默认为居中对齐；第二个Row，由于mainAxisSize值为`MainAxisSize.min`，Row的宽度等于两个Text的宽度和，所以对齐是无意义的，所以会从左往右显示；第三个Row设置textDirection值为`TextDirection.rtl`，所以子widget会从右向左的顺序排列，而此时`MainAxisAlignment.end`表示左对齐，所以最终显示结果就是图中第三行的样子；第四个Row测试的是纵轴的对齐方式，由于两个子Text字体不一样，所以其高度也不同，我们指定了verticalDirection值为`VerticalDirection.up`，即从低向顶排列，而此时crossAxisAlignment值为CrossAxisAlignment.start表示底对齐。

Column

Column可以在垂直方向排列其子widget。参数和Row一样，不同的是布局方向为垂直，主轴纵轴正好相反，读者可类比Row来理解，在此不再赘述。

特殊情况

如果Row里面嵌套Row，或者Column里面再嵌套Column，那么只有对最外面的Row或Column会占用尽可能大的空间，里面Row或Column所占用的空间为实际大小，下面以Column为例说明：

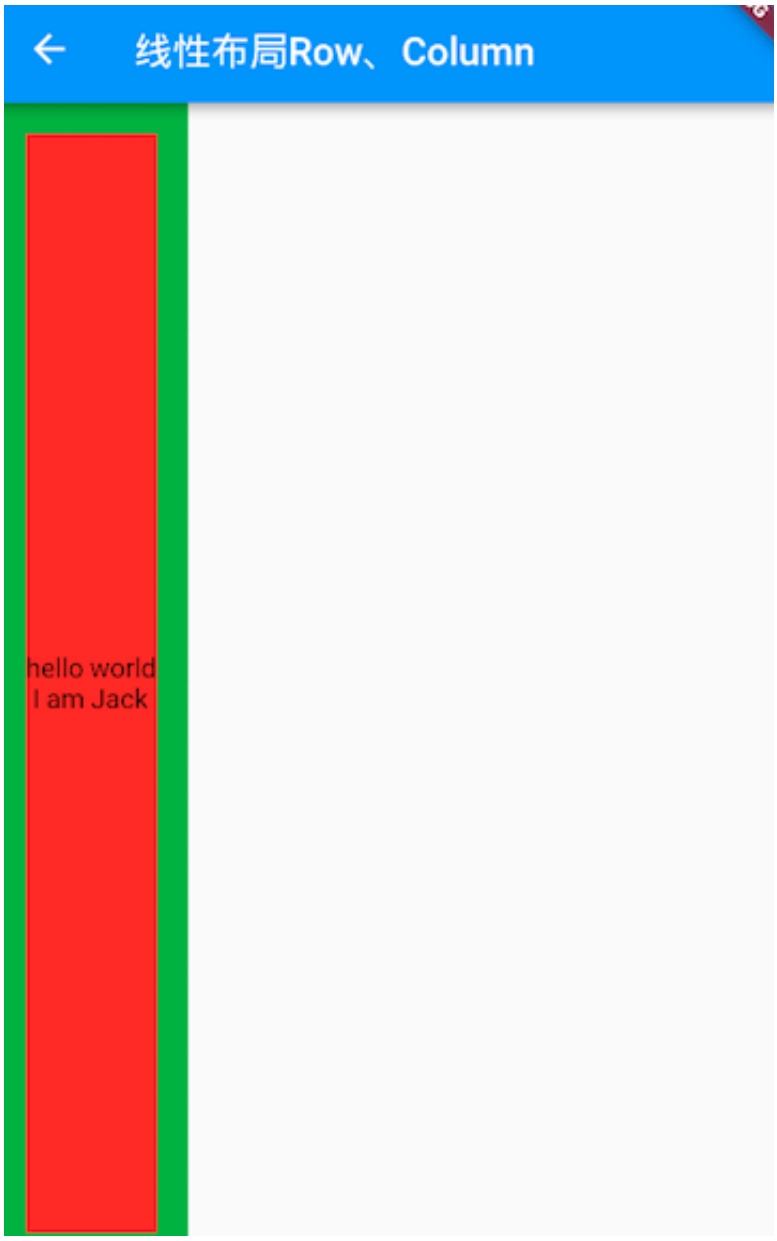
```
Container(
  color: Colors.green,
  child: Padding(
    padding: const EdgeInsets.all(16.0),
    child: Column(
      crossAxisAlignment: CrossAxisAlignment.start,
      mainAxisSize: MainAxisSize.max, //有效，外层Colum高度为整个屏幕
      children: <Widget>[
        Container(
          color: Colors.red,
          child: Column(
            mainAxisSize: MainAxisSize.max,//无效，内层Colum高度为实际
高度
            children: <Widget>[
              Text("hello world "),
              Text("I am Jack "),
            ],
          ),
        )
      ],
    ),
  ),
);
```

运行结果：

如果要让里面的Colum占满外部Colum，可以使用Expanded widget：

```
Expanded(
  child: Container(
    color: Colors.red,
    child: Column(
      mainAxisAlignment: MainAxisAlignment.center, //垂直方向居中对齐
      children: <Widget>[
        Text("hello world "),
        Text("I am Jack "),
      ],
    ),
  ),
)
```

运行效果：



我们将在介绍弹性布局时详细介绍Expanded。

弹性布局

弹性布局允许子widget按照一定比例来分配父容器空间，弹性布局的概念在其UI系统中也都存在，如H5中的弹性盒子布局，Android中的FlexboxLayout。Flutter中的弹性布局主要通过Flex和Expanded来配合实现。

Flex

Flex可以沿着水平或垂直方向排列子widget，如果你知道主轴方向，使用Row或Column会方便一些，因为Row和Column都继承自Flex，参数基本相同，所以能使用Flex的地方一定可以使用Row或Column。Flex本身功能是很强大的，它也可以和Expanded配合实现弹性布局，接下来我们只讨论Flex和弹性布局相关的属性(其它属性已经在介绍Row和Column时介绍过了)。

```
Flex({
  ...
  @required this.direction, //弹性布局的方向, Row默认为水平方向, Column默认为垂直方向
  List<Widget> children = const <Widget>[],
})
```

Flex继承自MultiChildRenderObjectWidget，对应的RenderObject为RenderFlex，RenderFlex中实现了其布局算法。

Expanded

可以按比例“扩伸”Row、Column和Flex子widget所占用的空间。

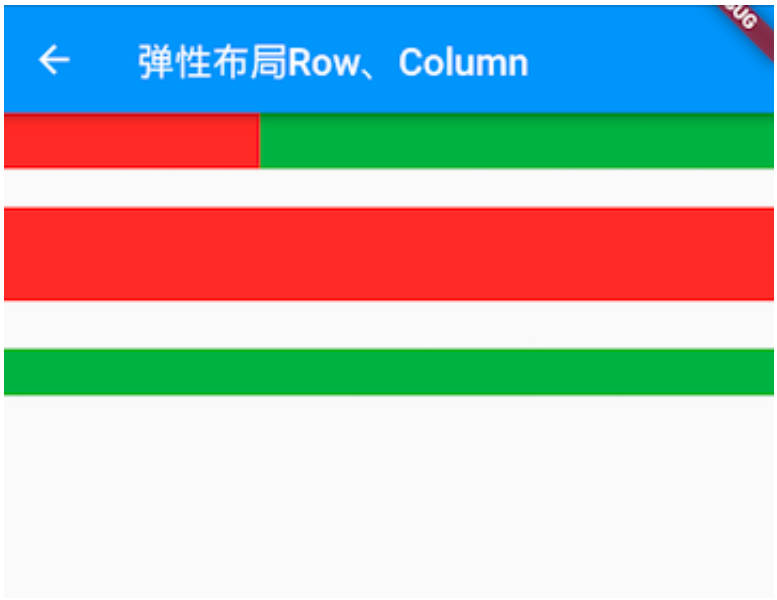
```
const Expanded({
  int flex = 1,
  @required Widget child,
})
```

flex为弹性系数，如果为0或null，则child是没有弹性的，即不会被扩伸占用的空间。如果大于0，所有的Expanded按照其flex的比例来分割主轴的全部空闲空间。下面我们看一个例子：

```
class FlexLayoutTestRoute extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Column(
      children: <Widget>[
        //Flex的两个子widget按1：2来占据水平空间
        Flex(
          direction: Axis.horizontal,
          children: <Widget>[
            Expanded(
              flex: 1,
```

```
              child: Container(
                height: 30.0,
                color: Colors.red,
              ),
            ),
            Expanded(
              flex: 2,
              child: Container(
                height: 30.0,
                color: Colors.green,
              ),
            ),
          ],
        ),
        Padding(
          padding: const EdgeInsets.only(top: 20.0),
          child: SizedBox(
            height: 100.0,
            //Flex的三个子widget，在垂直方向按2：1：1来占用100像素的空间
            child: Flex(
              direction: Axis.vertical,
              children: <Widget>[
                Expanded(
                  flex: 2,
                  child: Container(
                    height: 30.0,
                    color: Colors.red,
                  ),
                ),
                Spacer(
                  flex: 1,
                ),
                Expanded(
                  flex: 1,
                  child: Container(
                    height: 30.0,
                    color: Colors.green,
                  ),
                ),
              ],
            ),
          ),
        ),
      ],
    );
  }
}
```

运行效果如下：



示例中的Spacer的功能是占用指定比例的空间，实际上它只是Expanded的一个包装：

```
new Expanded(
  flex: flex,
  child: const SizedBox(
    height: 0.0,
    width: 0.0,
  ),
);
```

流式布局

Wrap

在介绍Row和Colum时，如果子widget超出屏幕范围，则会报溢出错误，如：

```
Row(
  children: <Widget>[
    Text("xxx"*100)
  ],
);
```

运行：

可以看到，右边溢出部分报错。这是因为Row默认只有一行，如果超出屏幕不会折行。我们把超出屏幕显示范围会自动折行的布局称为流式布局。Flutter中通过Wrap和Flow来支持流式布局，将上例中的Row换成Wrap后溢出部分则会自动折行。下面是Wrap的定义：

```
Wrap({
  ...
  this.direction = Axis.horizontal,
  this.alignment = WrapAlignment.start,
  this.spacing = 0.0,
  this.runAlignment = WrapAlignment.start,
  this.runSpacing = 0.0,
  this.crossAxisAlignment = WrapCrossAlignment.start,
  this.textDirection,
  this.verticalDirection = VerticalDirection.down,
  List<Widget> children = const <Widget>[],
})
```

我们可以看到Wrap的很多属性在Row（包括Flex和Column）中也有，如direction、crossAxisAlignment、textDirection、verticalDirection等，这些参数意义是相同的，我们不再重复介绍，读者可以查阅前面介绍Row的部分。读者可以认为Wrap和Flex（包括Row和Column）除了超出显示范围后Wrap会折行外，其它行为基本相同。下面我们看一下Wrap特有的几个属性：

- spacing：主轴方向子widget的间距
- runSpacing：纵轴方向的间距
- runAlignment：纵轴方向的对齐方式

下面看一个示例子：

```
Wrap(
  spacing: 8.0, // 主轴(水平)方向间距
  runSpacing: 4.0, // 纵轴（垂直）方向间距
  alignment: WrapAlignment.center, //沿主轴方向居中
  children: <Widget>[
    new Chip(
      avatar: new CircleAvatar(backgroundColor: Colors.blue, child:
Text('A')),
      label: new Text('Hamilton'),
    ),
    new Chip(
      avatar: new CircleAvatar(backgroundColor: Colors.blue, child:
Text('M')),
      label: new Text('Lafayette'),
    ),
    new Chip(
      avatar: new CircleAvatar(backgroundColor: Colors.blue, child:
Text('H')),
      label: new Text('Mulligan'),
    ),
    new Chip(
      avatar: new CircleAvatar(backgroundColor: Colors.blue, child:
Text('J')),
      label: new Text('Laurens'),
    ),
  ],
)
```

运行效果：

Flow

我们一般很少会使用Flow，因为其过于复杂，需要自己实现子widget的位置转换，在很多场景下首先要考虑的是Wrap是否满足需求。Flow主要用于一些需要自定义布局策略或性能要求较高(如动画中)的场景。Flow有如下优点：

- 性能好；Flow是一个对child尺寸以及位置调整非常高效的控件，Flow用转换矩阵（transformation matrices）在对child进行位置调整的时候进行了优化：在Flow定位过后，如果child的尺寸或者位置发生了变化，在FlowDelegate中的`paintChildren()`方法中调用`context.paintChild`进行重绘，而`context.paintChild`在重绘时使用了转换矩阵（transformation matrices），并没有实际调整Widget位置。
- 灵活；由于我们需要自己实现FlowDelegate的`paintChildren()`方法，所以我们需要自己计算每一个widget的位置，因此，可以自定义布局策略。

缺点：

- 使用复杂.
- 不能自适应子widget大小，必须通过指定父容器大小或实现TestFlowDelegate的`getSize`返回固定大小。

示例：

我们对六个色块进行自定义流式布局：

```
Flow(
  delegate: TestFlowDelegate(margin: EdgeInsets.all(10.0)),
  children: <Widget>[
    new Container(width: 80.0, height:80.0, color: Colors.red,),
    new Container(width: 80.0, height:80.0, color: Colors.green,),
    new Container(width: 80.0, height:80.0, color: Colors.blue,),
    new Container(width: 80.0, height:80.0,  color:
Colors.yellow,),
    new Container(width: 80.0, height:80.0, color: Colors.brown,),
    new Container(width: 80.0, height:80.0,  color:
Colors.purple,),
  ],
)
```

实现TestFlowDelegate:

```
class TestFlowDelegate extends FlowDelegate {
  EdgeInsets margin = EdgeInsets.zero;
  TestFlowDelegate({this.margin});
  @override
  void paintChildren(FlowPaintingContext context) {
    var x = margin.left;
    var y = margin.top;
    //计算每一个子widget的位置
    for (int i = 0; i < context.childCount; i++) {
      var w = context.getChildSize(i).width + x + margin.right;
      if (w < context.size.width) {
        context.paintChild(i,
            transform: new Matrix4.translationValues(
                x, y, 0.0));
        x = w + margin.left;
      } else {
        x = margin.left;
        y += context.getChildSize(i).height + margin.top +
margin.bottom;
        //绘制子widget(有优化)
        context.paintChild(i,
            transform: new Matrix4.translationValues(
                x, y, 0.0));
        x += context.getChildSize(i).width + margin.left +
margin.right;
      }
    }
  }

  getSize(BoxConstraints constraints){
    //指定Flow的大小
    return Size(double.infinity,200.0);
  }

  @override
  bool shouldRepaint(FlowDelegate oldDelegate) {
    return oldDelegate != this;
  }
}
```

效果：

可以看到我们主要的任务就是实现 `paintChildren`，它的主要任务是确定每个子widget位置。由于Flow不能自适应子widget的大小，我们通过在 `getSize` 返回一个固定大小来指定Flow的大小。

层叠布局

层叠布局和Web中的绝对定位、Android中的Frame布局是相似的，子widget可以根据到父容器四个角的位置来确定本身的位置。绝对定位允许子widget堆叠（按照代码中声明的顺序）。Flutter中使用Stack和Positioned来实现绝对定位，Stack允许子widget堆叠，而Positioned可以给子widget定位（根据Stack的四个角）。

Stack

```
Stack({
  this.alignment = AlignmentDirectional.topStart,
  this.textDirection,
  this.fit = StackFit.loose,
  this.overflow = Overflow.clip,
  List<Widget> children = const <Widget>[],
})
```

- alignment：此参数决定如何去对齐没有定位（没有使用Positioned）或部分定位的子widget。所谓部分定位，在这里**特指没有在某一个轴上定位：**left、right为横轴，top、bottom为纵轴，只要包含某个轴上的一个定位属性就算在该轴上有定位。
- textDirection：和Row、Wrap的textDirection功能一样，都用于决定alignment对齐的参考系即：textDirection的值为`TextDirection.ltr`，则alignment的`start`代表左，`end`代表右；textDirection的值为`TextDirection.rtl`，则alignment的`start`代表右，`end`代表左。
- fit：此参数用于决定**没有定位**的子widget如何去适应Stack的大小。`StackFit.loose`表示使用子widget的大小，`StackFit.expand`表示扩伸到Stack的大小。
- overflow：此属性决定如何显示超出Stack显示空间的子widget，值为`Overflow.clip`时，超出部分会被剪裁（隐藏），值为`Overflow.visible`时则不会。

Positioned

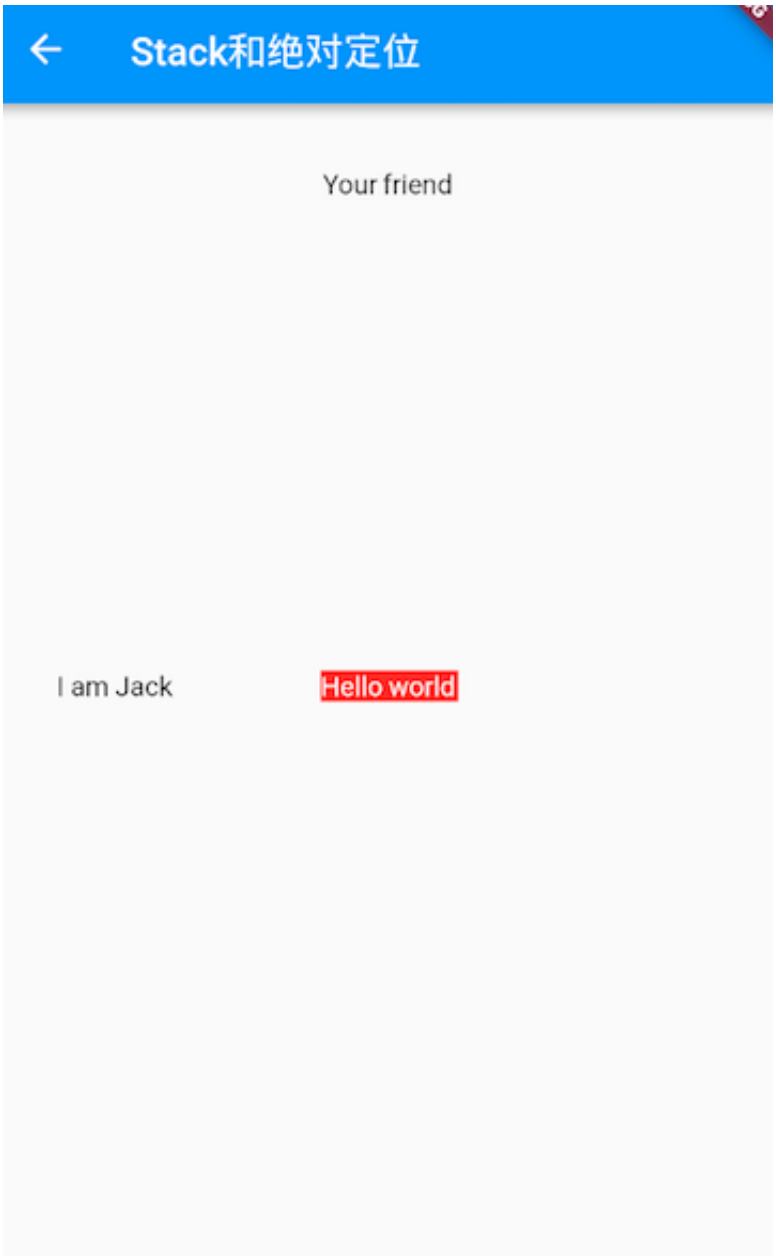
```
const Positioned({
  Key key,
  this.left,
  this.top,
  this.right,
  this.bottom,
  this.width,
  this.height,
  @required Widget child,
})
```

left、top 、right、 bottom分别代表离Stack左、上、右、底四边的距离。width和height用于指定定位元素的宽度和高度，注意，此处的width、height 和其它地方的意义稍微有点区别，此处用于配合left、top 、right、 bottom来定位widget，举个例子，在水平方向时，你只能指定left、right、width三个属性中的两个，如指定left和width后，right会自动算出(left+width)，如果同时指定三个属性则会报错，垂直方向同理。

示例

```
//通过ConstrainedBox来确保Stack占满屏幕
ConstrainedBox(
  constraints: BoxConstraints.expand(),
  child: Stack(
    alignment:Alignment.center , //指定未定位或部分定位widget的对齐方式
    children: <Widget>[
      Container(child: Text("Hello world",style: TextStyle(color:
Colors.white)),
        color: Colors.red,
      ),
      Positioned(
        left: 18.0,
        child: Text("I am Jack"),
      ),
      Positioned(
        top: 18.0,
        child: Text("Your friend"),
      )
    ],
  ),
);
```

运行效果如下：

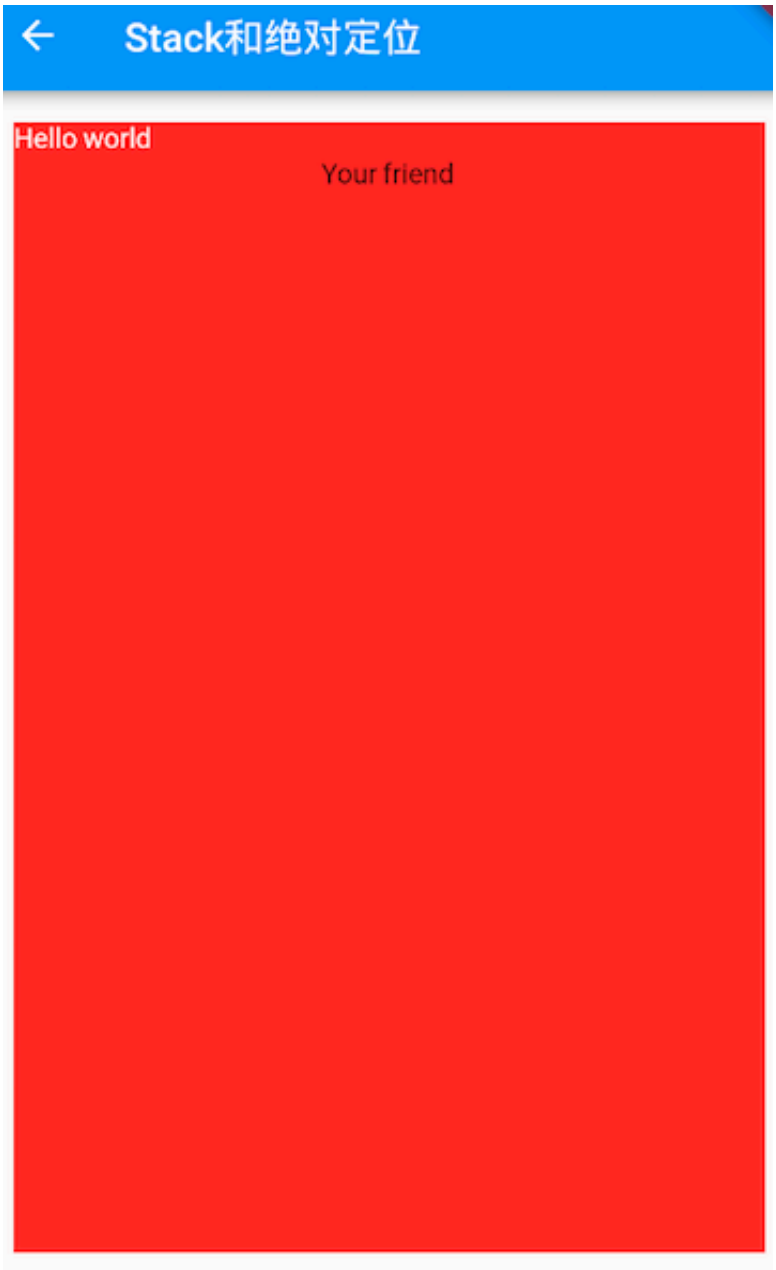


由于第一个子widget Text("Hello world")没有指定定位，并且alignment值为`Alignment.center`，所以，它会居中显示。第二个子widget Text("I am Jack")只指定了水平方向的定位(left)，所以属于部分定位，即垂直方向上没有定位，那么它在垂直方向对齐方式则会按照alignment指定的对齐方式对齐，即垂直方向居中。对于第三个子widget Text("Your friend")，和第二个Text原理一样，只不过是水平方向没有定位，则水平方向居中。

我们给上例中的Stack指定一个fit属性，然后将三个子widget的顺序调整一下：

```
Stack(
  alignment:Alignment.center ,
  fit: StackFit.expand, //未定位widget占满Stack整个空间
  children: <Widget>[
    Positioned(
      left: 18.0,
      child: Text("I am Jack"),
    ),
    Container(child: Text("Hello world",style: TextStyle(color:
Colors.white)),
      color: Colors.red,
    ),
    Positioned(
      top: 18.0,
      child: Text("Your friend"),
    )
  ],
),
```

显示效果如下：



可以看到，由于第二个子widget没有定位，所以 `fit` 属性会对它起作用，就会占满Stack。有Stack子元素是堆叠的，所以第一个子Widget被第二个遮住了，而第三个在最上层，所以可以正常显示。