

容器类Widget

容器类Widget和布局类Widget都作用于其子Widget，不同的是：

- 布局类widget一般都需要接收一个widget数组（children），他们直接或间接继承自（或包含）MultiChildRenderObjectWidget；而容器类Widget一般只需要接受一个子Widget（child），他们直接或间接继承自（或包含）SingleChildRenderObjectWidget。
- 布局类Widget是按照一定的排列方式来对其子widget进行排列；而容器类Widget一般只是包装其子Widget，对其添加一些修饰（补白或背景色等）、变换(旋转或剪裁等)、或限制(大小等)。

注意，Flutter官方并没有对Widget进行官方分类，我们对其分类主要是为了方便讨论和对widget功能的区分记忆。

本章目录

- [Padding](#)
- [布局限制类容器ConstrainedBox、SizeBox](#)
- [装饰容器DecoratedBox](#)
- [变换Transform](#)
- [Container容器](#)

Padding

Padding可以给其子节点添加补白（填充），我们在前面很多示例中都已经使用过它了，现在来看看它的定义：

```
Padding({  
  ...  
  EdgeInsetsGeometry padding,  
  Widget child,  
})
```

EdgeInsetsGeometry是一个抽象类，开发中，我们一般都使用EdgeInsets，它是EdgeInsetsGeometry的一个子类，定义了一些设置补白的便捷方法。

EdgeInsets

我们看看EdgeInsets提供的便捷方法：

- `fromLTRB(double left, double top, double right, double bottom)`：分别指定四个方向的补白。
- `all(double value)`：所有方向均使用相同数值的补白。
- `only({left, top, right, bottom})`：可以设置具体某个方向的补白(可以同时指定多个方向)。
- `symmetric({vertical, horizontal})`：用于设置对称方向的补白，vertical指top和bottom，horizontal指left和right。

示例

```

class PaddingTestRoute extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Padding(
      // 上下左右各添加16像素补白
      padding: EdgeInsets.all(16.0),
      child: Column(
        // 显式指定对齐方式为左对齐, 排除对齐干扰
        crossAxisAlignment: CrossAxisAlignment.start,
        children: <Widget>[
          Padding(
            // 左边添加8像素补白
            padding: const EdgeInsets.only(left: 8.0),
            child: Text("Hello world"),
          ),
          Padding(
            // 上下各添加8像素补白
            padding: const EdgeInsets.symmetric(vertical: 8.0),
            child: Text("I am Jack"),
          ),
          Padding(
            // 分别指定四个方向的补白
            padding: const EdgeInsets.fromLTRB(20.0, .0, 20.0, 20.0),
            child: Text("Your friend"),
          )
        ],
      ),
    );
  }
}

```

ConstrainedBox和SizedBox

ConstrainedBox和SizedBox都是通过RenderConstrainedBox来渲染的。SizedBox只是ConstrainedBox一个定制，本节把他们放在一起讨论。

ConstrainedBox

ConstrainedBox用于对齐子widget添加额外的约束。例如，如果你想让子widget的最小高度是80像素，你可以使用 `const BoxConstraints(minHeight: 80.0)` 作为子widget的约束。

示例

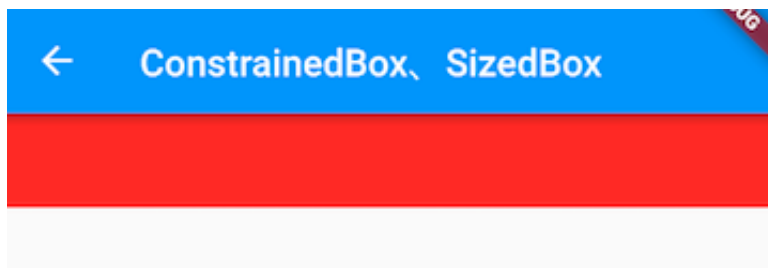
我们先定义一个redBox，它是一个背景颜色为红色的盒子，不指定它的宽度和高度：

```
Widget redBox=DecoratedBox(  
  decoration: BoxDecoration(color: Colors.red),  
);
```

我们实现一个最小高度为50，宽度尽可能大的红色容器。

```
ConstrainedBox(  
  constraints: BoxConstraints(  
    minWidth: double.infinity, // 宽度尽可能大  
    minHeight: 50.0 // 最小高度为50像素  
  ),  
  child: Container(  
    height: 5.0,  
    child: redBox  
  ),  
)
```

显示效果：



可以看到，我们虽然将Container的高度设置为5像素，但是最终却是50像素，这正是ConstrainedBox的最小高度限制生效了。如果将Container的高度设置为80像素，那么最终红色区域的高度也会是80像素，因为在此示例中，ConstrainedBox只限制了最小高度，并未限制最大高度。

BoxConstraints

BoxConstraints用于设置限制条件，它的定义如下：

```
const BoxConstraints({
  this.minWidth = 0.0, // 最小宽度
  this.maxWidth = double.infinity, // 最大宽度
  this.minHeight = 0.0, // 最小高度
  this.maxHeight = double.infinity // 最大高度
})
```

BoxConstraints还定义了一些便捷的构造函数，用于快速生成特定限制规则的BoxConstraints，如 `BoxConstraints.tight(Size size)`，它可以生成给定大小的限制；`const BoxConstraints.expand()` 可以生成一个尽可能大的用以填充另一个容器的BoxConstraints。除此之外还有一些其它的便捷函数，读者可以查看API文档。

SizedBox

SizedBox用于给子widget指定固定的宽高，如：

```
SizedBox(
  width: 80.0,
  height: 80.0,
  child: redBox
)
```

运行效果如下：



实际上SizedBox和只是ConstrainedBox一个定制，上面代码等价于：

```
ConstrainedBox(
  constraints: BoxConstraints.tightFor(width: 80.0,height: 80.0),
  child: redBox,
)
```

而 `BoxConstraints.tightFor(width: 80.0,height: 80.0)` 等价于：

```
BoxConstraints(minHeight: 80.0,maxHeight: 80.0,minWidth:
80.0,maxWidth: 80.0)
```

而实际上ConstrainedBox和SizedBox都是通过RenderConstrainedBox来渲染的，我们可以看到ConstrainedBox和SizedBox的 `createRenderObject()` 方法都返回的是一个RenderConstrainedBox对象：

```
@override
RenderConstrainedBox createRenderObject(BuildContext context) {
  return new RenderConstrainedBox(
    additionalConstraints: ...,
  );
}
```

多重限制

如果某一个widget有多个父ConstrainedBox限制，那么最终会是哪个生效？我们看一个例子：

```
ConstrainedBox(
  constraints: BoxConstraints(minWidth: 60.0, minHeight: 60.0),
  //父
  child: ConstrainedBox(
    constraints: BoxConstraints(minWidth: 90.0, minHeight:
20.0), //子
    child: redBox,
  )
)
```

上面我们有父子两个ConstrainedBox，他们的限制条件不同，运行后效果如下：



最终显示效果是宽90，高60，也就是说子ConstrainedBox的minWidth生效，而minHeight是父ConstrainedBox生效。单凭这个例子，我们还总结不出什么规律，我们将上例中父子限制条件换一下：

```
ConstrainedBox(  
  constraints: BoxConstraints(minWidth: 90.0, minHeight: 20.0),  
  child: ConstrainedBox(  
    constraints: BoxConstraints(minWidth: 60.0, minHeight: 60.0),  
    child: redBox,  
  )  
)
```



最终的显示效果仍然是90，高60，效果相同，但意义不同，因为此时minWidth生效的是父ConstrainedBox，而minHeight是子ConstrainedBox生效。

通过上面示例，我们发现有多重限制时，对于minWidth和minHeight来说，是取父子中相应数值较大的。实际上，只有这样才能保证父限制与子限制不冲突。

思考题：对于maxWidth和maxHeight，多重限制的策略是什么样的呢？

UnconstrainedBox

UnconstrainedBox不会对子Widget产生任何限制，它允许其子Widget按照其本身大小绘制。一般情况下，我们会很少直接使用此widget，但在"去除"多重限制的时候也许会有帮助，我们看一下下面的代码：

```

ConstrainedBox(
  constraints: BoxConstraints(minWidth: 60.0, minHeight: 100.0),
  //父
  child: UnconstrainedBox( //“去除”父级限制
    child: ConstrainedBox(
      constraints: BoxConstraints(minWidth: 90.0, minHeight:
20.0), //子
      child: redBox,
    ),
  )
)

```

上面代码中，如果没有中间的UnconstrainedBox，那么根据上面所述的多重限制规则，那么最终将显示一个90×100的红色框。但是由于 UnconstrainedBox “去除”了父ConstrainedBox的限制，则最终会按照子ConstrainedBox的限制来绘制redBox，即90×20：



但是，读者请注意，UnconstrainedBox对父限制的“去除”并非是真的去除，上面例子中虽然红色区域大小是90×20，但上方仍然有80的空白空间。也就是说父限制的minHeight(100.0)仍然是生效的，只不过它不影响最终子元素的大小，但仍然还是占有相应的空间，可以认为此时的父ConstrainedBox是作用于子ConstrainedBox上，而redBox只受子ConstrainedBox限制，这一点请读者务必注意。

那么有什么方法可以彻底去除父BoxConstraints的限制吗？答案是否定的！所以在此提示读者，在定义一个通用的widget时，如果对子widget指定限制时一定要注意，因为一旦指定限制条件，子widget如果要进行相关自定义大小时将可能非常困难，因为子widget在不更改父widget的代码的情况下无法彻底去除其限制条件。

DecoratedBox

DecoratedBox可以在其子widget绘制前(或后)绘制一个装饰Decoration（如背景、边框、渐变等）。DecoratedBox定义如下：


```
const DecoratedBox({
  Decoration decoration,
  DecorationPosition position = DecorationPosition.background,
  Widget child
})
```

- decoration: 代表将要绘制的装饰，它类型为Decoration，Decoration是一个抽象类，它定义了一个接口 `createBoxPainter()`，子类的主要职责是需要通过实现它来创建一个画笔，该画笔用于绘制装饰。
- position: 此属性决定在哪里绘制Decoration，它接收DecorationPosition的枚举类型，该枚举类两个值：
 - background: 在子widget之后绘制，即背景装饰。
 - foreground: 在子widget之上绘制，即前景。

BoxDecoration

我们通常会直接使用 `BoxDecoration`，它是一个Decoration的子类，实现了常用的装饰元素的绘制。

```
BoxDecoration({
  Color color, // 颜色
  DecorationImage image, // 图片
  BoxBorder border, // 边框
  BorderRadiusGeometry borderRadius, // 圆角
  List<BoxShadow> boxShadow, // 阴影, 可以指定多个
  Gradient gradient, // 渐变
  BlendMode backgroundBlendMode, // 背景混合模式
  BoxShape shape = BoxShape.rectangle, // 形状
})
```

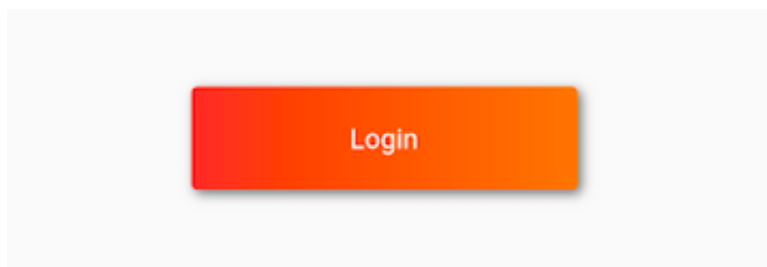
各个属性名都是自解释的，详情读者可以查看API文档，我们看一个示例：

```

DecoratedBox(
  decoration: BoxDecoration(
    gradient: LinearGradient(colors:
[Colors.red,Colors.orange[700]]), //背景渐变
    borderRadius: BorderRadius.circular(3.0), //3像素圆角
    boxShadow: [ //阴影
      BoxShadow(
        color:Colors.black54,
        offset: Offset(2.0,2.0),
        blurRadius: 4.0
      )
    ],
  ),
  child: Padding(padding: EdgeInsets.symmetric(horizontal: 80.0,
vertical: 18.0),
    child: Text("Login", style: TextStyle(color: Colors.white),),
  )
)

```

效果如下：



怎么样，通过BoxDecoration，我们实现了一个渐变按钮的外观，但此示例还不是一个标准的按钮，因为它还不能响应点击事件，我们将在本章末尾来实现一个完整的GradientButton。

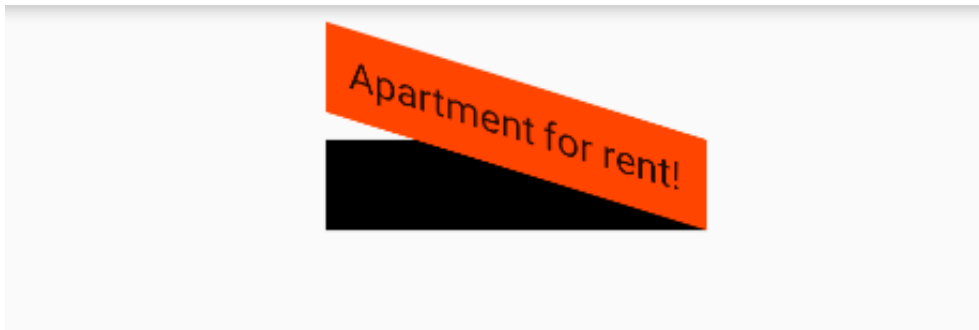
Transform变换

Transform可以在其子Widget绘制时对其应用一个矩阵变换（transformation），Matrix4是一个4D矩阵，通过它我们可以实现各种矩阵操作。下面是一个例子：

```

Container(
  color: Colors.black,
  child: new Transform(
    alignment: Alignment.topRight, // 相对于坐标系原点的对齐方式
    transform: new Matrix4.skewY(0.3), // 沿Y轴倾斜0.3弧度
    child: new Container(
      padding: const EdgeInsets.all(8.0),
      color: Colors.deepOrange,
      child: const Text('Apartment for rent!'),
    ),
  ),
);

```



关于矩阵变换的相关内容属于线性代数范畴，本书不做讨论，读者有兴趣可以自行了解。本书中，我们把焦点放在Flutter中一些常见的变换效果上。

平移

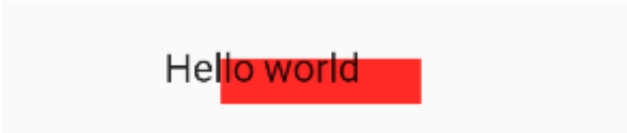
`Transform.translate`接收一个`offset`参数，可以在绘制时沿x、y轴对子widget平移指定的距离。

```

DecoratedBox(
  decoration: BoxDecoration(color: Colors.red),
  // 默认原点为左上角，左移20像素，向上平移5像素
  child: Transform.translate(offset: Offset(-20.0, -5.0),
    child: Text("Hello world"),
  ),
)

```

效果：



旋转

Transform.rotate可以对子widget进行旋转变换，如：

```
DecoratedBox(  
  decoration: BoxDecoration(color: Colors.red),  
  child: Transform.rotate(  
    // 旋转90度  
    angle: math.pi/2 ,  
    child: Text("Hello world"),  
  ),  
);
```

注意：要使用math.pi需先进行如下导包。

```
import 'dart:math' as math;
```

效果：

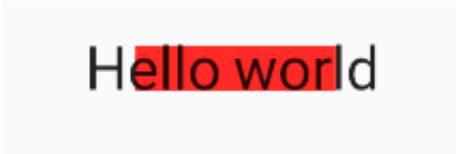


缩放

Transform.scale可以对子Widget进行缩小或放大，如：

```
DecoratedBox(  
  decoration: BoxDecoration(color: Colors.red),  
  child: Transform.scale(  
    scale: 1.5, // 放大到1.5倍  
    child: Text("Hello world")  
  )  
);
```

效果：

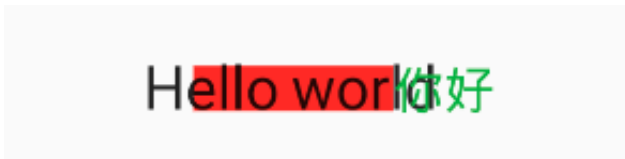


注意

- Transform的变换是应用在绘制阶段，而并不是应用在布局(layout)阶段，所以无论对子widget应用何种变化，其占用空间的大小和在屏幕上的位置都是固定不变的，因为这些是在布局阶段就确定的。下面我们具体说明：

```
Row(  
  mainAxisAlignment: MainAxisAlignment.center,  
  children: <Widget>[  
    DecoratedBox(  
      decoration: BoxDecoration(color: Colors.red),  
      child: Transform.scale(scale: 1.5,  
        child: Text("Hello world")  
      )  
    ),  
    Text("你好", style: TextStyle(color: Colors.green, fontSize:  
      18.0),)  
  ],  
)
```

显示效果：



由于第一个Text应用变换(放大)后，其在绘制时会放大，但其占用的空间依然为红色部分，所以第二个text会紧挨着红色部分，最终就会出现文字有重合部分。

- 由于矩阵变化只会作用在绘制阶段，所以在某些场景下，在UI需要变化时，可以直接通过矩阵变化来达到视觉上的UI改变，而不需要去重新触发build流程，这样会节省layout的开销，所以性能会比较好。如之前介绍的Flow widget，它内部就是用矩阵变换来更新UI，除此之外，Flutter的动画widget中也大量使用了Transform以提高性能。

RotatedBox

RotatedBox和Transform.rotate功能相似，它们都可以对子widget进行旋转变换，但是有一点不同：RotatedBox的变换是在layout阶段，会影响在子widget的位置和大小。我们将上面介绍Transform.rotate时的示例改一下：

```

Row(
  mainAxisAlignment: MainAxisAlignment.center,
  children: <Widget>[
    DecoratedBox(
      decoration: BoxDecoration(color: Colors.red),
      //将Transform.rotate换成RotatedBox
      child: RotatedBox(
        quarterTurns: 1, // 旋转90度(1/4圈)
        child: Text("Hello world"),
      ),
    ),
    Text("你好", style: TextStyle(color: Colors.green, fontSize:
18.0),)
  ],
),

```

效果：



由于RotatedBox是作用于layout阶段，所以widget会旋转90度（而不只是绘制的内容），decoration会作用到widget所占用的实际空间上，所以就是上图的效果。读者可以和前面Transform.rotate示例对比理解。

Container

Container是我们要介绍的最后一个容器类widget，它本身不对应具体的RenderObject，它是DecoratedBox、ConstrainedBox、Transform、Padding、Align等widget的一个组合widget。所以我们只需通过一个Container可以实现同时需要装饰、变换、限制的场景。下面是Container的定义：

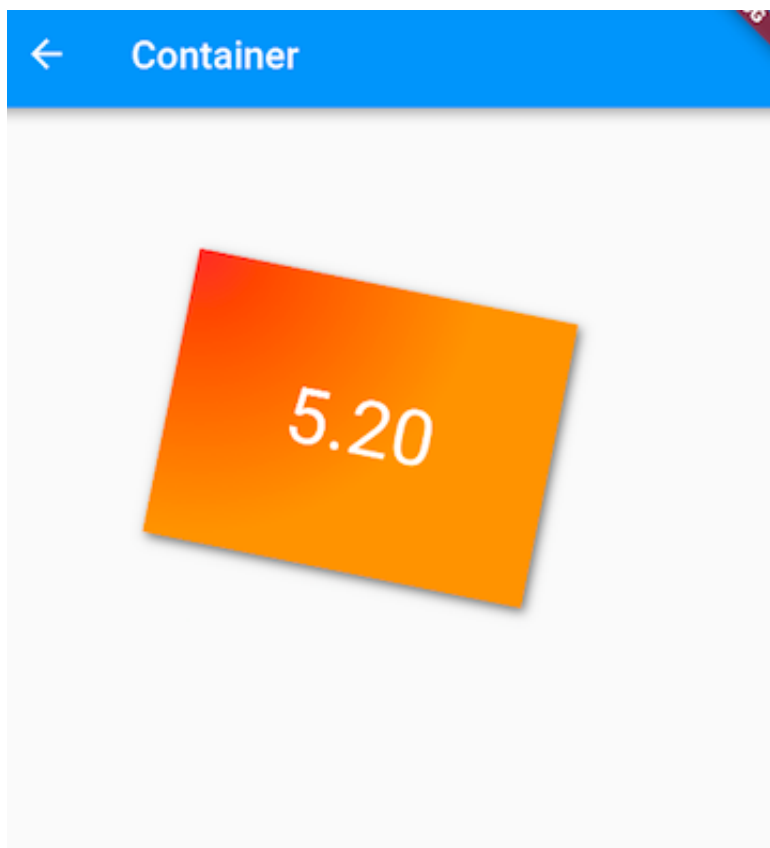
```
Container({
    this.alignment,
    this.padding, // 容器内补白, 属于decoration的装饰范围
    Color color, // 背景色
    Decoration decoration, // 背景装饰
    Decoration foregroundDecoration, // 前景装饰
    double width, // 容器的宽度
    double height, // 容器的高度
    BoxConstraints constraints, // 容器大小的限制条件
    this.margin, // 容器外补白, 不属于decoration的装饰范围
    this.transform, // 变换
    this.child,
})
```

大多说属性在介绍其它容器时都已经介绍过了，不再赘述，但有两点需要说明：

- 容器的大小可以通过 `width`、`height` 属性来指定，也可以通过 `constraints` 来指定，如果同时存在时，`width`、`height` 优先。实际上 `Container` 内部会根据 `width`、`height` 来生成一个 `constraints`。
- `color` 和 `decoration` 是互斥的，实际上，当指定 `color` 时，`Container` 内会自动创建一个 `decoration`。

实例

我们通过 `Container` 来实现如下的卡片：



代码：

```

Container(
  margin: EdgeInsets.only(top: 50.0, left: 120.0), // 容器外补白
  constraints: BoxConstraints.tightFor(width: 200.0, height:
150.0), // 卡片大小
  decoration: BoxDecoration(// 背景装饰
    gradient: RadialGradient( // 背景径向渐变
      colors: [Colors.red, Colors.orange],
      center: Alignment.topLeft,
      radius: .98
    ),
    boxShadow: [ // 卡片阴影
      BoxShadow(
        color: Colors.black54,
        offset: Offset(2.0, 2.0),
        blurRadius: 4.0
      )
    ]
  ),
  transform: Matrix4.rotationZ(.2), // 卡片倾斜变换
  alignment: Alignment.center, // 卡片内文字居中
  child: Text( // 卡片文字
    "5.20", style: TextStyle(color: Colors.white, fontSize: 40.0),
  ),
);

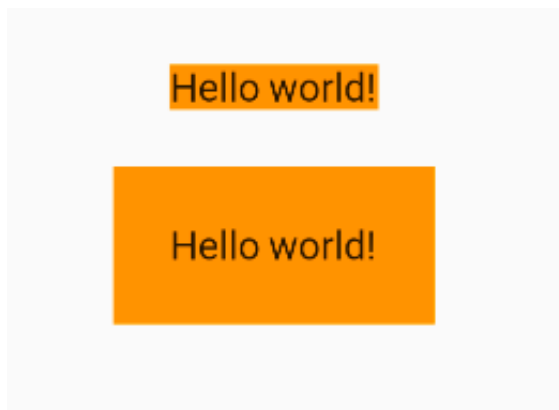
```

可以看到Container通过组合多种widget来实现复杂强大的功能，在Flutter中，这也正是组合优先于继承的实例。

Padding和Margin

接下来我们看看Container的 `margin` 和 `padding` 属性的区别：

```
...
Container(
  margin: EdgeInsets.all(20.0), // 容器外补白
  color: Colors.orange,
  child: Text("Hello world!"),
),
Container(
  padding: EdgeInsets.all(20.0), // 容器内补白
  color: Colors.orange,
  child: Text("Hello world!"),
),
...
```



可以发现，直观的感觉就是margin的补白是在容器外部，而padding的补白是在容器内部，读者需要记住这个差异。事实上，Container内 `margin` 和 `padding` 都是通过Padding widget来实现的，上面的示例代码实际上等价于：

```

...
Padding(
  padding: EdgeInsets.all(20.0),
  child: DecoratedBox(
    decoration: BoxDecoration(color: Colors.orange),
    child: Text("Hello world!"),
  ),
),
DecoratedBox(
  decoration: BoxDecoration(color: Colors.orange),
  child: Padding(
    padding: const EdgeInsets.all(20.0),
    child: Text("Hello world!"),
  ),
),
...

```

剪裁

Flutter中提供了一些剪裁函数，用于对子widget的进行剪裁。

剪裁Widget	作用
ClipOval	子widget大小为正方形时剪裁为内贴圆形，为矩形时，剪裁为内贴椭圆
ClipRRect	将子widget剪裁为圆角矩形
ClipRect	剪裁子widget到实际占用的矩形大小（溢出部分剪裁）

下面我们看一个例子：

```

import 'package:flutter/material.dart';

class ClipTestRoute extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    // 头像
    Widget avatar = Image.asset("./images/avatar.png", width:
60.0);
    return Center(

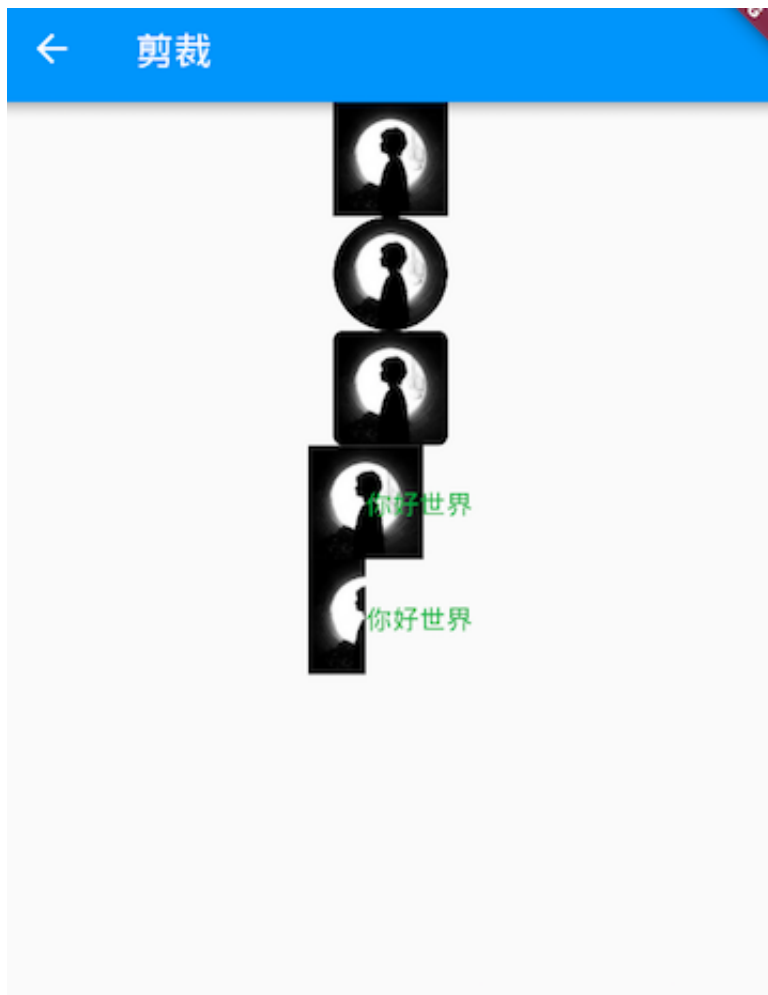
```

```

child: Column(
  children: <Widget>[
    avatar, // 不剪裁
    ClipOval(child: avatar), // 剪裁为圆形
    ClipRRect( // 剪裁为圆角矩形
      borderRadius: BorderRadius.circular(5.0),
      child: avatar,
    ),
    Row(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        Align(
          alignment: Alignment.topLeft,
          widthFactor: .5, // 宽度设为原来宽度一半, 另一半会溢出
          child: avatar,
        ),
        Text("你好世界", style: TextStyle(color:
Colors.green),)
      ],
    ),
    Row(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        ClipRect(// 将溢出部分剪裁
          child: Align(
            alignment: Alignment.topLeft,
            widthFactor: .5, // 宽度设为原来宽度一半
            child: avatar,
          ),
        ),
        Text("你好世界", style: TextStyle(color: Colors.green))
      ],
    ),
  ],
);
}
}

```

效果如下:



上面示例代码中已添加注释，比较简单，值得一提的是最后的两个Row，通过Align设置widthFactor为0.5后，图片的实际宽度等于 60×0.5 ，即原宽度一半，但此时图片溢出部分依然会显示，所以第一个“你好世界”会和图片的另一部分重合，为了剪裁掉溢出部分，我们在第二个Row中通过ClipRect将溢出部分剪裁了。

CustomClipper

如果我们想剪裁子widget的特定区域，比如上面示例的图片中，我们只想截取图片中部 40×30 像素的范围，应该怎么做？这时我们可以使用CustomClipper来自定义剪裁区域，实现代码如下：

首先，自定义一个CustomClipper：

```
class myClipper extends CustomClipper<Rect> {
  @override
  Rect getClip(Size size) => Rect.fromLTWH(10.0, 15.0, 40.0, 30.0);

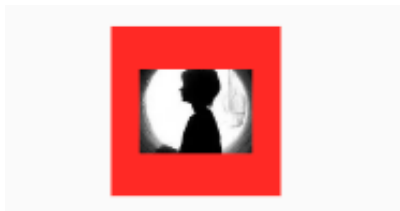
  @override
  bool shouldReclip(CustomClipper<Rect> oldClipper) => false;
}
```

- `getClip()` 是用于获取剪裁区域的接口，由于图片大小是60×60，我们返回剪裁区域为 `Rect.fromLTWH(10.0, 15.0, 40.0, 30.0)`，及图片中部40×30像素的范围。
- `shouldReclip()` 接口决定是否重新剪裁。如果在应用中，剪裁区域始终不会发生变化时应该返回 `false`，这样就不会触发重新剪裁，避免不必要的性能开销。如果剪裁区域会发生变化（比如在对剪裁区域执行一个动画），那么变化后应该返回 `true` 来重新执行剪裁。

然后，我们通过`ClipRect`来执行剪裁，为了看清图片实际所占用的位置，我们设置一个红色背景：

```
DecoratedBox(
  decoration: BoxDecoration(
    color: Colors.red
  ),
  child: ClipRect(
    clipper: myClipper(), // 使用自定义的clipper
    child: avatar
  ),
)
```

运行效果：



可以看到我们的剪裁成功了，但是图片所占用的空间大小仍然是60×60（红色区域），这是因为剪裁是在layout完成后的绘制阶段进行的，这和Transform是相似的。

