事件处理与通知

Flutter中的手势系统有两个独立的层。第一层为原始指针(pointer)事件，它描述了屏幕上指针（例如，触摸、鼠标和触控笔）的位置和移动。 第二层为手势，描述由一个或多个指针移动组成的语义动作，如拖动、缩放、双击等。本章将先分别介绍如何处理这两种事件，最后再介绍一下Flutter中重要的Notification机制。

本章目录

Pointer事件处理

本节先来介绍一下原始指针事件(Pointer Event，在移动设备上通常为触摸事件)，下一节再介绍手势处理。

在移动端，各个平台或UI系统的原始指针事件模型基本都是一致，即：一次完整的事件分为三个阶段：手指按下、手指移动、和手指抬起，而更高级别的手势（如点击、双击、拖动等）都是基于这些原始事件的。

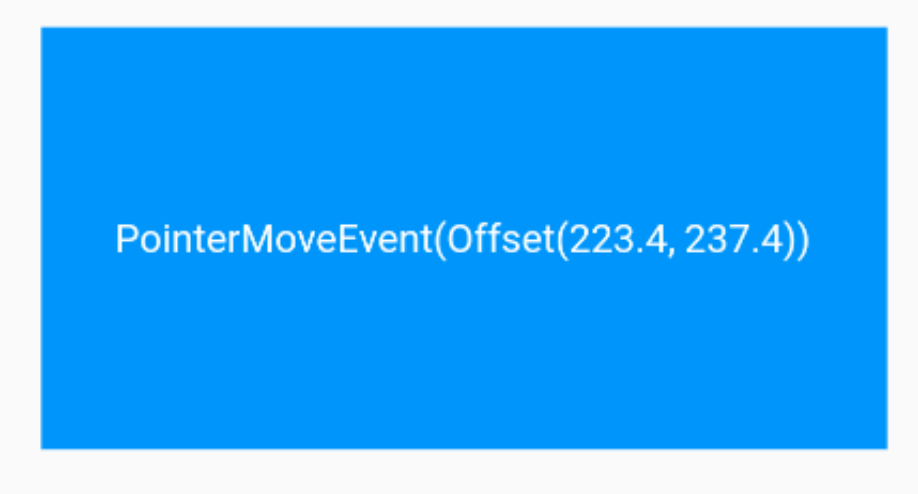
当指针按下时，Flutter会对应用程序执行**命中测试(Hit Test)**，以确定指针与屏幕接触的位置存在哪些widget， 指针按下事件（以及该指针的后续事件）然后被分发到由命中测试发现的最内部的widget。 从那里开始，这些事件会冒泡在widget树中向上冒泡，这些事件会从最内部的widget被分发到到widget根的路径上的所有Widget，这和Web开发中浏览器的**事件冒泡**机制相似， 但是Flutter中没有机制取消或停止冒泡过程，而浏览器的冒泡是可以停止的。注意，只有通过命中测试的Widget才能触发事件。

Flutter中可以使用Listener widget来监听原始触摸事件，它也是一个功能性widget。

```
Listener({
  Key key,
  this.onPointerDown, //手指按下回调
  this.onPointerMove, //手指移动回调
  this.onPointerUp,//手指抬起回调
  this.onPointerCancel,//触摸事件取消回调
  this.behavior = HitTestBehavior.deferToChild, //在命中测试期间如何表
现
  Widget child
})
```

我们先看一个示例，后面再单独讨论一下`behavior`属性。

```
...
//定义一个状态，保存当前指针位置
PointerEvent _event;
...
Listener(
  child: Container(
    alignment: Alignment.center,
    color: Colors.blue,
    width: 300.0,
    height: 150.0,
    child: Text(_event?.toString()??"",style: TextStyle(color:
Colors.white)),
  ),
  onPointerDown: (PointerDownEvent event) =>
setState(()=>_event=event),
  onPointerMove: (PointerMoveEvent event) =>
setState(()=>_event=event),
  onPointerUp: (PointerUpEvent event) =>
setState(()=>_event=event),
),
```



手指在蓝色矩形区域内移动即可看到当前指针偏移，当触发指针事件时，参数PointerDownEvent、PointerMoveEvent、PointerUpEvent都是PointerEvent的一个子类，PointerEvent类中包括当前指针的一些信息，如：

- `position`：它是鼠标相对于当对于全局坐标的偏移。
- `delta`：两次指针移动事件（PointerMoveEvent）的距离。
- `pressure`：按压力度，如果手机屏幕支持压力传感器(如iPhone的3D Touch)，此属性会更有意义，如果手机不支持，则始终为1。
- `orientation`：指针移动方向，是一个角度值。

上面只是PointerEvent一些常用属性，除了这些它还有很多属性，读者可以查看API文档。

现在，我们重点来介绍一下behavior属性，它决定子Widget如何响应命中测试，它的值类型为HitTestBehavior，这是一个枚举类，有三个枚举值：

- `deferToChild`：子widget会一个接一个的进行命中测试，如果子Widget中有测试通过的，则当前Widget通过，这就意味着，如果指针事件作用于子Widget上时，其父(祖先)Widget也肯定可以收到该事件。

- `opaque`：在命中测试时，将当前Widget当成不透明处理(即使本身是透明的)，最终的效果相当于当前Widget的整个区域都是点击区域。举个例子：

```
Listener(
    child: ConstrainedBox(
        constraints: BoxConstraints.tight(Size(300.0, 150.0)),
        child: Center(child: Text("Box A")),
        //behavior: HitTestBehavior.opaque
    ),
    onPointerDown: (event) => print("down A")
),
```

上例中，只有点击文本内容区域才会触发点击事件，如果我们想让整个300×150的矩形区域都能点击我们可以将`behavior`设为`HitTestBehavior.opaque`。注意，该属性并不能用于在Widget树中拦截（忽略）事件，它只是决定命中测试时的Widget大小。

- `translucent`：当点击透明区域时，可以对底部widget进行命中测试，这意味着底部widget也可以接收事件。`translucent`可以在Stack中实现"点透"的效果。例如：

```
Stack(
  children: <Widget>[
    Listener(
      child: ConstrainedBox(
        constraints: BoxConstraints.tight(Size(300.0, 200.0)),
        child: DecoratedBox(
            decoration: BoxDecoration(color: Colors.blue)),
      ),
      onPointerDown: (event) => print("down0"),
    ),
    Listener(
      child: ConstrainedBox(
        constraints: BoxConstraints.tight(Size(200.0, 100.0)),
        child: Center(child: Text("左上角200*100范围内非文本区域点
击")),
      ),
      onPointerDown: (event) => print("down1"),
      //behavior: HitTestBehavior.translucent, //放开此行注释后可
以"点透"
    )
  ],
)
```

上例中，当注释掉最后一行代码后，在左上角200*100范围内非文本区域点击

时，控制台只会打印“down1”，当放开注释后，再点击时就会打印：

```
I/flutter ( 3039): down1
I/flutter ( 3039): down0
```

忽略PointerEvent

假如我们不想让某个子树响应PointerEvent的话，我们可以使用IgnorePointer和AbsorbPointer，这两个Widget都能阻止子树接收指针事件，不同之处在于AbsorbPointer本身会参与命中测试，而IgnorePointer本身不会参与，这就意味着AbsorbPointer本身是可以接收指针事件的(但其子树不行)，而IgnorePointer不可以。一个简单的例子如下：

```
Listener(
  child: AbsorbPointer(
    child: Listener(
      child: Container(
        color: Colors.red,
        width: 200.0,
        height: 100.0,
      ),
      onPointerDown: (event)=>print("in"),
    ),
  ),
  onPointerDown: (event)=>print("up"),
)
```

点击Container时，由于它在`AbsorbPointer`的子树上，所以不会响应指针事件，所以日志不会输出"in"，但`AbsorbPointer`本身是可以接收指针事件的，所以会输出"up"。如果将`AbsorbPointer`换成`IgnorePointer`，那么两个都不会输出。

手势识别GestureDetector

GestureDetector是一个用于手势识别的功能性Widget，我们通过它可以来识别各种手势，它是指针事件的语义化封装，接下来我们详细介绍一下各种手势识别：

点击、双击、长按

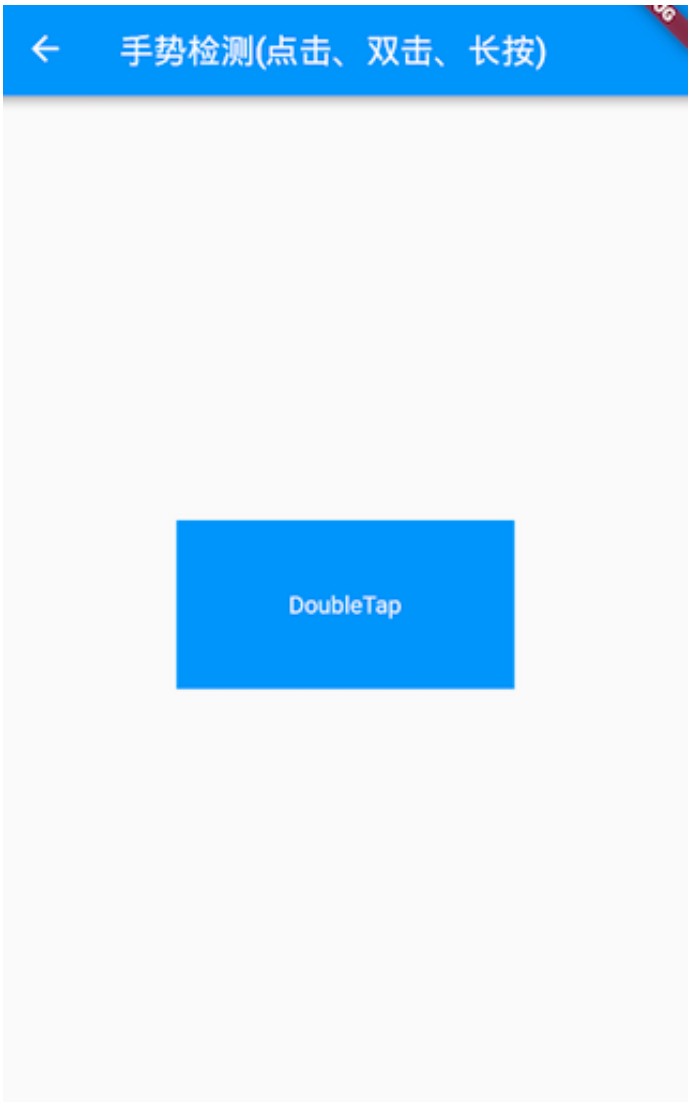
我们通过GestureDetector对Container进行手势识别，触发相应事件后，在Container上显示事件名，为了增大点击区域，将Container设置为200×100，代码如下：

```
class GestureDetectorTestRoute extends StatefulWidget {
  @override
  _GestureDetectorTestRouteState createState() =>
      new _GestureDetectorTestRouteState();
}

class _GestureDetectorTestRouteState extends
State<GestureDetectorTestRoute> {
  String _operation = "No Gesture detected!"; //保存事件名
  @override
  Widget build(BuildContext context) {
    return Center(
      child: GestureDetector(
        child: Container(
          alignment: Alignment.center,
          color: Colors.blue,
          width: 200.0,
          height: 100.0,
          child: Text(_operation,
            style: TextStyle(color: Colors.white),
          ),
        ),
        onTap: () => updateText("Tap"),//点击
        onDoubleTap: () => updateText("DoubleTap"), //双击
        onLongPress: () => updateText("LongPress"), //长按
      ),
    );
  }

  void updateText(String text) {
    //更新显示的事件名
    setState(() {
      _operation = text;
    });
  }
}
```

运行效果：



注意： 当同时监听`onTap`和`onDoubleTap`事件时，当用户触发tap事件时，会有200毫秒左右的延时，这是因为当用户点击完之后很可能会再次点击以触发双击事件，所以GestureDetector会等一断时间来确定是否为双击事件。如果用户只监听了`onTap`（没有监听`onDoubleTap`）事件时，则没有延时。

拖动、滑动

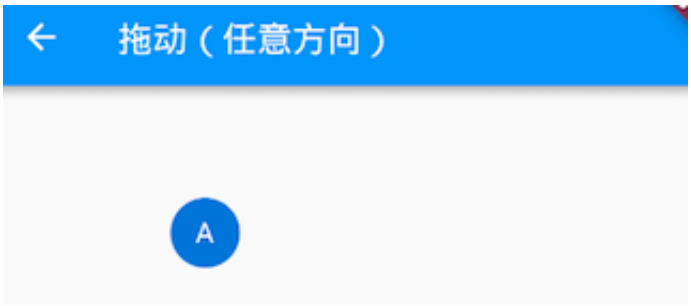
一次完整的手势过程是指用户手指按下到抬起的整个过程，期间，用户按下手指后可能会移动，也可能不会移动。GestureDetector对于拖动和滑动事件是没有区分的，他们本质上是一样的。GestureDetector会将要监听的widget的原点（左上角）作为本次手势的原点，当用户在监听的widget上按下手指时，手势识别就会开始。下面我们看一个拖动圆形字母A的示例：

```dart
class _Drag extends StatefulWidget {
  @override
  _DragState createState() => new _DragState();
}

class _DragState extends State<_Drag> with
SingleTickerProviderStateMixin {
  double _top = 0.0; //距顶部的便宜
  double _left = 0.0;//距左边的偏移

  @override
  Widget build(BuildContext context) {
    return Stack(
      children: <Widget>[
        Positioned(
          top: _top,
          left: _left,
          child: GestureDetector(
            child: CircleAvatar(child: Text("A")),
            //手指按下时会触发此回调
            onPanDown: (DragDownDetails e) {
              //打印手指按下的位置(相对于屏幕)
              print("用户手指按下：${e.globalPosition}");
            },
            //手指滑动时会触发此回调
            onPanUpdate: (DragUpdateDetails e) {
              //用户手指滑动时，更新偏移，重新构建
              setState(() {
                _left += e.delta.dx;
                _top += e.delta.dy;
              });
            },
            onPanEnd: (DragEndDetails e){
              //打印滑动结束时在x、y轴上的速度
              print(e.velocity);
            },
          ),
        )
      ],
    );
  }
}
```

运行后，就可以在任意方向拖动了：



日志：

```
I/flutter ( 8513): 用户手指按下：Offset(26.3, 101.8)
I/flutter ( 8513): Velocity(235.5, 125.8)
```

代码解释：

- `DragDownDetails.globalPosition`：当用户按下时，此属性为用户按下的位置相对于**屏幕**(而非父widget)原点(左上角)的偏移。
- `DragUpdateDetails.delta`：当用户在屏幕上滑动时，会触发多次Update事件，`delta`指一次Update事件的滑动的偏移量。
- `DragEndDetails.velocity`：该属性代表用户抬起手指时的滑动速度(包含x、y两个轴的），示例中并没有处理手指抬起时的速度，常见的效果是根据用户抬起手指时的速度做一个减速动画。

单一方向拖动

在本示例中，是可以朝任意方向拖动的，但是在很多场景，我们只需要沿一个方向来拖动，如一个垂直方向的列表，GestureDetector可以只识别特定方向的手势事件，我们将上面的例子改为只能沿垂直方向拖动：

```
class _DragVertical extends StatefulWidget {
  @override
  _DragVerticalState createState() => new _DragVerticalState();
}

class _DragVerticalState extends State<_DragVertical> {
  double _top = 0.0;

  @override
  Widget build(BuildContext context) {
    return Stack(
      children: <Widget>[
        Positioned(
          top: _top,
          child: GestureDetector(
            child: CircleAvatar(child: Text("A")),
            //垂直方向拖动事件
            onVerticalDragUpdate: (DragUpdateDetails details) {
              setState(() {
                _top += details.delta.dy;
              });
            }
          ),
        )
      ],
    );
  }
}
```

这样就只能在垂直方向拖动了，如果只想在水平方向滑动同理。

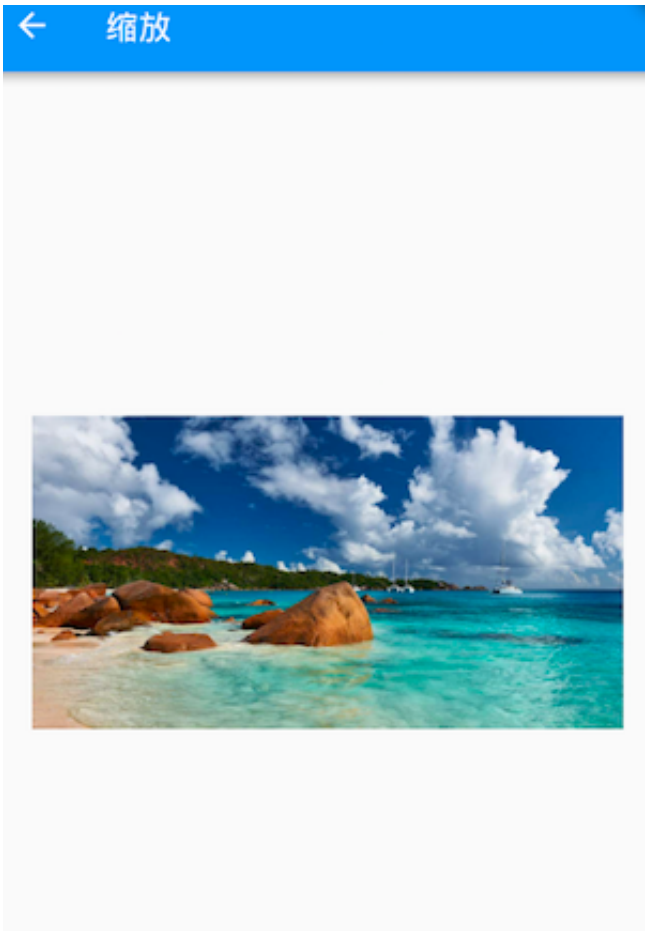
缩放

GestureDetector可以监听缩放事件，下面示例演示了一个简单的图片缩放效果：

```
class _ScaleTestRouteState extends State<_ScaleTestRoute> {
  double _width = 200.0; //通过修改图片宽度来达到缩放效果

  @override
  Widget build(BuildContext context) {
   return Center(
     child: GestureDetector(
        //指定宽度，高度自适应
        child: Image.asset("./images/sea.png", width: _width),
        onScaleUpdate: (ScaleUpdateDetails details) {
          setState(() {
            //缩放倍数在0.8到10倍之间
            _width=200*details.scale.clamp(.8, 10.0);
          });
        },
      ),
   );
  }
}
```

运行效果：



现在在图片上双指张开、收缩就可以放大、缩小图片。本示例比较简单，实际中我们通常还需要一些其它功能，如双击放大或缩小一定倍数、双指张开离开屏幕时执行一个减速放大动画等，我们将在后面“动画”一章中实现一个完整的缩放Widget。

GestureRecognizer

GestureDetector内部是使用一个或多个GestureRecognizer来识别各种手势的，而GestureRecognizer的作用就是通过Listener来将原始指针事件转换为语义手势，GestureDetector直接可以接收一个子Widget。GestureRecognizer是一个抽象类，一种手势的识别器对应一个GestureRecognizer的子类，Flutter实现了丰富的手势识别器，我们可以直接使用。

示例

假设我们要给一段富文本（RichText）的不同部分分别添加点击事件处理器，但是TextSpan并不是一个Widget，这时我们不能用GestureDetector，但TextSpan有一个`recognizer`属性，它可以接收一个GestureRecognizer，假设我们在点击时给文本变色:

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

class _GestureRecognizerTestRouteState
    extends State<_GestureRecognizerTestRoute> {
  TapGestureRecognizer _tapGestureRecognizer = new
TapGestureRecognizer();
  bool _toggle = false; //变色开关

  @override
  void dispose() {
    //用到GestureRecognizer的话一定要调用其dispose方法释放资源
    _tapGestureRecognizer.dispose();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Text.rich(
          TextSpan(
              children: [
                TextSpan(text: "你好世界"),
                TextSpan(
                  text: "点我变色",
                  style: TextStyle(
                      fontSize: 30.0,
                      color: _toggle ? Colors.blue : Colors.red
                  ),
                  recognizer: _tapGestureRecognizer
                    ..onTap = () {
                      setState(() {
                        _toggle = !_toggle;
                      });
                    },
                ),
                TextSpan(text: "你好世界"),
              ]
          )
      ),
    );
  }
}
```

运行效果：

注意：使用GestureRecognizer后一定要调用其`dispose()`方法来释放资源（主要是取消内部的计时器）。

手势竞争与冲突

竞争

如果在上例中我们同时监听水平和垂直方向的拖动事件，那么我们斜着拖动时哪个方向会生效？实际上取决于第一次移动时两个轴上的位移分量，哪个轴的大，哪个轴在本次滑动事件竞争中就胜出。实际上Flutter中的手势识别引入了一个Arena的概念，Arena直译为“竞技场”的意思，每一个手势识别器（GestureRecognizer）都是一个“竞争者”（GestureArenaMember），当发生滑动事件时，他们都要在“竞技场”去竞争本次事件的处理权，而最终只有一个“竞争者”会胜出(win)。例如，假设有一个ListView，它的第一个子Widget也是ListView，如果现在滑动这个子ListView，父ListView会动吗？答案是否定的，这时只有子Widget会动，因为这时子Widget会胜出而获得滑动事件的处理权。

示例

我们以拖动手势为例，同时识别水平和垂直方向的拖动手势，当用户按下手指时就会触发竞争（水平方向和垂直方向），一旦某个方向“获胜”，则直到当次拖动手势结束都会沿着该方向移动。代码如下：

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

class BothDirectionTestRoute extends StatefulWidget {
  @override
  BothDirectionTestRouteState createState() =>
      new BothDirectionTestRouteState();
}

class BothDirectionTestRouteState extends
State<BothDirectionTestRoute> {
  double _top = 0.0;
  double _left = 0.0;

  @override
  Widget build(BuildContext context) {
    return Stack(
      children: <Widget>[
        Positioned(
          top: _top,
          left: _left,
          child: GestureDetector(
            child: CircleAvatar(child: Text("A")),
            //垂直方向拖动事件
            onVerticalDragUpdate: (DragUpdateDetails details) {
              setState(() {
                _top += details.delta.dy;
              });
            },
            onHorizontalDragUpdate: (DragUpdateDetails details) {
              setState(() {
                _left += details.delta.dx;
              });
            },
          ),
        )
      ],
    );
  }
}
```

此示例运行后，每次拖动只会沿一个方向移动（水平或垂直），而竞争发生在手指按下后首次移动（move）时，此例中具体的“获胜”条件是：首次移动时的位移在水平和垂直方向上的分量大的一个获胜。

手势冲突

由于手势竞争最终只有一个胜出者，所以，当有多个手势识别器时，可以会产生冲突。假设有一个widget，它可以左右拖动，现在我们也想检测在它上面手指按下和抬起的事件，代码如下：

```
class GestureConflictTestRouteState extends
State<GestureConflictTestRoute> {
  double _left = 0.0;
  @override
  Widget build(BuildContext context) {
    return Stack(
      children: <Widget>[
        Positioned(
          left: _left,
          child: GestureDetector(
              child: CircleAvatar(child: Text("A")), //要拖动和点击的
widget
              onHorizontalDragUpdate: (DragUpdateDetails details) {
                setState(() {
                  _left += details.delta.dx;
                });
              },
              onHorizontalDragEnd: (details){
                print("onHorizontalDragEnd");
              },
              onTapDown: (details){
                print("down");
              },
              onTapUp: (details){
                print("up");
              },
          ),
        )
      ],
    );
  }
}
```

现在我们按住圆形“A”拖动然后抬起手指，控制台日志如下：

```
I/flutter (17539): down
I/flutter (17539): onHorizontalDragEnd
```

我们发现没有打印"up"，这是因为在拖动时，刚开始按下手指时在没有移动时，拖动手势还没有完整的语义，此时TapDown手势胜出(win)，此时打印"down"，而拖动时，拖动手势会胜出，当手指抬起时，`onHorizontalDragEnd` 和 `onTapUp` 发生了冲突，但是因为是在拖动的语义中，所以 `onHorizontalDragEnd` 胜出，所以就会打印“onHorizontalDragEnd”。如果我们的代码逻辑中，对于手指按下和抬起是强依赖的，比如在一个轮播图组件中，我们希望手指按下时，暂停轮播，而抬起时恢复轮播，但是由于轮播图组件中本身可能已经处理了拖动手势（支持手动滑动切换），甚至可能也支持了缩放手势，这时我们如果在外部再用 `onTapDown` 、 `onTapUp` 来监听的话是不行的。这时我们应该怎么做？其实很简单，通过Listener监听原始指针事件就行：

```
Positioned(
  top:80.0,
  left: _leftB,
  child: Listener(
    onPointerDown: (details) {
      print("down");
    },
    onPointerUp: (details) {
      //会触发
      print("up");
    },
    child: GestureDetector(
      child: CircleAvatar(child: Text("B")),
      onHorizontalDragUpdate: (DragUpdateDetails details) {
        setState(() {
          _leftB += details.delta.dx;
        });
      },
      onHorizontalDragEnd: (details) {
        print("onHorizontalDragEnd");
      },
    ),
  ),
)
```

总结：

手势冲突只是手势级别的，而手势是对原始指针的语义化的识别，所以在遇到复杂的冲突场景时，都可以通过Listener直接识别原始指针事件来解决冲突。

事件总线

在APP中，我们经常会需要一个广播机制，用以跨页面事件通知，比如一个需要登录的APP中，页面会关注用户登录或注销事件，来进行一些状态更新。这时候，一个事件总线便会非常有用，事件总线通常实现了订阅者模式，订阅者模式包含发布者和订阅者两种角色，可以通过事件总线来触发事件和监听事件，本节我们实现一个简单的全局事件总线，我们使用单例模式，代码如下：

```
//订阅者回调签名
typedef void EventCallback(arg);

class EventBus {
  //私有构造函数
  EventBus._internal();

  //保存单例
  static EventBus _singleton = new EventBus._internal();

  //工厂构造函数
  factory EventBus()=> _singleton;

  //保存事件订阅者队列，key:事件名(id)，value: 对应事件的订阅者队列
  var _emap = new Map<Object, List<EventCallback>>();

  //添加订阅者
  void on(eventName, EventCallback f) {
    if (eventName == null || f == null) return;
    _emap[eventName] ??= new List<EventCallback>();
    _emap[eventName].add(f);
  }

  //移除订阅者
  void off(eventName, [EventCallback f]) {
    var list = _emap[eventName];
    if (eventName == null || list == null) return;
    if (f == null) {
      _emap[eventName] = null;
    } else {
      list.remove(f);
    }
  }

  //触发事件，事件触发后该事件所有订阅者会被调用
  void emit(eventName, [arg]) {
    var list = _emap[eventName];
    if (list == null) return;
    int len = list.length - 1;
    //反向遍历，防止在订阅者在回调中移除自身带来的下标错位
```

```
    for (var i = len; i > -1; --i) {
      list[i](arg);
    }
  }
}

//定义一个top-level变量，页面引入该文件后可以直接使用bus
var bus = new EventBus();
```

使用

```
//页面A中
...
 //监听登录事件
bus.on("login", (arg) {
  // do something
});

//登录页B中
...
//登录成功后触发登录事件，页面A中订阅者会被调用
bus.emit("login", userInfo);
```

> 注意：Dart中实现单例模式的标准做法就是使用static变量+工厂构造函数的方式，这样就可以保证`new EventBus()`始终返回都是同一个实例，读者应该理解并掌握这种方法。

事件总线通常用于Widget之间状态共享，但关于Widget之间状态共享也有一些专门的Package如redux，这和web框架Vue/React是一致的。通常情况下事件总线是足以满足业务需求的，如果你决定使用redux的话，一定要想清楚业务是否真的有必要用它，防止“化简为繁”、过度设计。

Notification

Notification是Flutter中一个重要的机制，在Widget树中，每一个节点都可以分发通知，通知会沿着当前节点（context）向上传递，所有父节点都可以通过NotificationListener来监听通知，Flutter中称这种通知由子向父的传递为“通知冒泡”（Notification Bubbling），这个和用户触摸事件冒泡是相似的，但有一点不同：通知冒泡可以中止，但用户触摸事件不行。

Flutter中很多地方使用了通知，如可滚动(Scrollable) Widget中滑动时就会分发ScrollNotification，而Scrollbar正是通过监听ScrollNotification来确定滚动条位置的。除了ScrollNotification，Flutter中还有SizeChangedLayoutNotification、KeepAliveNotification 、LayoutChangedNotification等。下面是一个监听Scrollable Widget滚动通知的例子：

```
NotificationListener(
  onNotification: (notification){
    //print(notification);
    switch (notification.runtimeType){
      case ScrollStartNotification: print("开始滚动"); break;
      case ScrollUpdateNotification: print("正在滚动"); break;
      case ScrollEndNotification: print("滚动停止"); break;
      case OverscrollNotification: print("滚动到边界"); break;
    }
  },
  child: ListView.builder(
      itemCount: 100,
      itemBuilder: (context, index) {
        return ListTile(title: Text("$index"),);
      }
  ),
);
```

上例中的滚动通知如ScrollStartNotification、ScrollUpdateNotification等都是继承自ScrollNotification类，不同类型的通知子类会包含不同的信息，比如ScrollUpdateNotification有一个`scrollDelta`属性，它记录了移动的位移，其它通知属性读者可以自己查看SDK文档。

自定义通知

除了Flutter内部通知，我们也可以自定义通知，下面我们看看如何实现自定义通知：

1. 定义一个通知类，要继承自Notification类；

```
class MyNotification extends Notification {
  MyNotification(this.msg);
  final String msg;
}
```

2. 分发通知。

   Notification有一个`dispatch(context)`方法，它是用于分发通知的，我们说过context实际上就是操作Element的一个接口，它与Element树上的节点是对应的，通知会从context对应的Element节点向上冒泡。

下面我们看一个完整的例子：

```
class NotificationRoute extends StatefulWidget {
  @override
  NotificationRouteState createState() {
    return new NotificationRouteState();
  }
}

class NotificationRouteState extends State<NotificationRoute> {
  String _msg="";
  @override
  Widget build(BuildContext context) {
    //监听通知
    return NotificationListener<MyNotification>(
      onNotification: (notification) {
        setState(() {
          _msg+=notification.msg+"  ";
        });
      },
      child: Center(
        child: Column(
          mainAxisSize: MainAxisSize.min,
          children: <Widget>[
//          RaisedButton(
//           onPressed: () =>
MyNotification("Hi").dispatch(context),
//           child: Text("Send Notification"),
//          ),
            Builder(
              builder: (context) {
```

```
                return RaisedButton(
                  //按钮点击时分发通知
                  onPressed: () =>
MyNotification("Hi").dispatch(context),
                  child: Text("Send Notification"),
                );
              },
            ),
            Text(_msg)
          ],
        ),
      ),
    );
  }
}

class MyNotification extends Notification {
  MyNotification(this.msg);
  final String msg;
}
```

上面代码中，我们每点一次按钮就会分发一个`MyNotification`类型的通知，我们在Widget根上监听通知，收到通知后我们将通知通过Text显示在屏幕上。

> 注意：代码中注释的部分是不能正常工作的，因为这个`context`是根Context，而NotificationListener是监听的子树，所以我们通过`Builder`来构建RaisedButton，来获得按钮位置的context。

运行效果如下：


7:08
DEBUG
Widget树Notification
Send Notification
Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi
Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi
Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi
Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi
Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi