本节目标:

```dart
[1]. 了解手势组件在画布中的使用方式。
[2]. 练习绘制，并根据手指滑动完成 [控制杆] 绘制。
[3]. 练习绘制，并根据手指滑动完成 [刻度尺] 绘制。
[4]. 了解如何限制绘制区域。
```

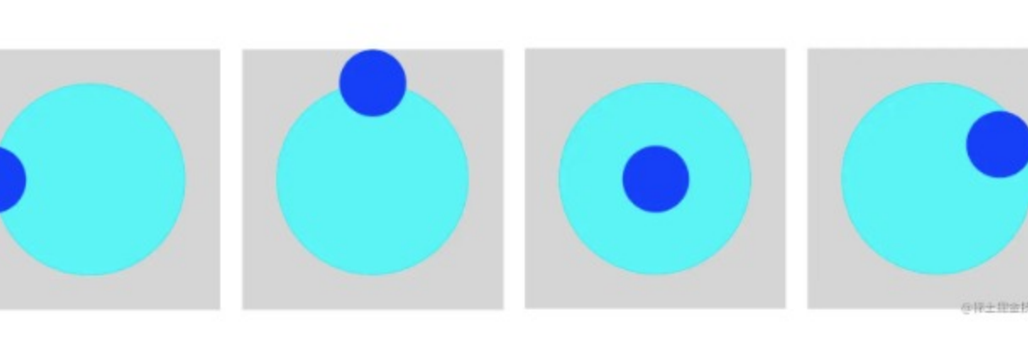
一、控制柄组件

> 下面的控制器很能体现出 手势操作在画布中的使用 ，其中小球的 圆心 只能在 大圈内移动 。可以回调出移动的方向和距离，是一个非常好的控制案例。其中包含一些 角度计算 、 边界控制 的技巧也很值得去体会。

---

1. 基本思路

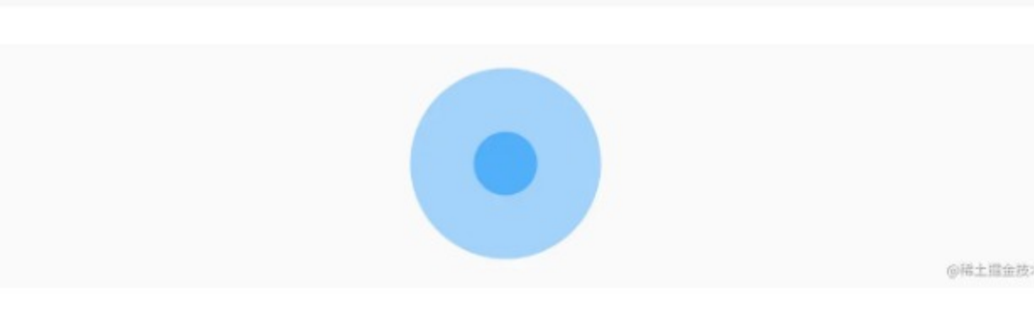
> 如下图所示，是控制柄的四个瞬间，灰色区域是组件的占位空间。小圆的圆心 只能在大圆的区域 内运动。 所以组件尺寸 `size` 和小圆半径 `handleRadius` 可以确定大圆半径 `bgR = size/2 - handleRadius` 在移动过程中，获取到触点位置，计算小圆偏移即可。



---

2. 静态效果

> 先完成下面的静态效果，这对于现在的我们来说可谓轻而易举。



```dart
---->[p12_gesture/s01_ctrl1/handle_widget.dart]----
class HandleWidget extends StatefulWidget {
  final double size;
  final double handleRadius;

  HandleWidget({Key? key, this.size = 60.0, this.handleRadius = 30.0})
      : super(key: key);

  @override
  _HandleWidgetState createState() => _HandleWidgetState();
}

class _HandleWidgetState extends State<HandleWidget> {
  @override
  Widget build(BuildContext context) {
    return CustomPaint(
        size: Size(widget.size, widget.size),
        painter: _HandlePainter(handleR: widget.handleRadius));
  }
}

class _HandlePainter extends CustomPainter {
  var _paint = Paint();
  var handleR;

  _HandlePainter({this.handleR}) {
    _paint
      ..color = Colors.blue
      ..style = PaintingStyle.fill
      ..isAntiAlias = true;
  }

  @override
  void paint(Canvas canvas, Size size) {
    canvas.clipRect(Offset.zero & size);
    final bgR = size.width / 2 - handleR;
    canvas.translate(size.width / 2, size.height / 2);
    _paint.color = _paint.color.withAlpha(100);
    canvas.drawCircle(Offset(0, 0), bgR, _paint);
    _paint.color = _paint.color.withAlpha(150);
    canvas.drawCircle(Offset(0, 0), handleR, _paint);
  }

  @override
  bool shouldRepaint(_HandlePainter oldDelegate) =>
      oldDelegate.handleR != handleR;
}
```

---

3. 添加手势控制

> 通过 `GestureDetector` 可以监听到手指在组件上的触碰信息。
> 在 `onPanUpdate` 时可以获取 触点信息 ，onPanEnd 是手指离开时，可以重置偏移。
> 将 `ValueNotifier<Offset>` 对象传入画板中用来触发画布更新。

```dart
---->[p12_gesture/s01_ctrl2/handle_widget.dart]----
class _HandleWidgetState extends State<HandleWidget> {
  ValueNotifier<Offset> _offset = ValueNotifier(Offset.zero);

  @override
  Widget build(BuildContext context) {
    return GestureDetector(
        onPanEnd: reset,
        onPanUpdate: parser,
        child: CustomPaint(
            size: Size(widget.size, widget.size),
            painter: _HandlePainter(
                color: Colors.green,
                handleR: widget.handleRadius,
                offset: _offset)));
  }

  reset(DragEndDetails details) {
    _offset.value = Offset.zero;
  }

  parser(DragUpdateDetails details) {
    // 见下...
  }
}
```

> 主要的核心逻辑就是 更新时的解析方法 ，下面着重看一下它:
> 通过触点减去尺寸的一半，可以得到触点 相对于中心点 的位置。

> 可以算出与 x 轴夹角值，当 x 为负时，角度计算为负值，需要校正一下 如下图。
> 如何通过 `thta = rad - pi / 2` 让坐标旋转 90°，成为正常坐标系，即 x 水平方向为 0°。
> 通过判断 到中心点的距离 是否大于 `bgR` 可以用来限定移动区域。


```dart
parser(DragUpdateDetails details) {
  final offset = details.localPosition;
  double dx = 0.0;
  double dy = 0.0;
  dx = offset.dx - widget.size / 2;
  dy = offset.dy - widget.size / 2;
  var rad = atan2(dx, dy);
  if (dx < 0) {
    rad += 2 * pi;
  }
  var bgR = widget.size / 2 - widget.handleRadius;
  var thta = rad - pi / 2; //旋转坐标系90度
  var d = sqrt(dx * dx + dy * dy);
  if (d > bgR) {
    dx = bgR * cos(thta);
    dy = -bgR * sin(thta);
  }
  _offset.value = Offset(dx, dy);
}
```

---

4. 绘制代码

> 绘制相对而言比较简单，小圆的圆心使用偏移坐标即可。

```dart
class _HandlePainter extends CustomPainter {
  var _paint = Paint();
  final ValueNotifier<Offset> offset;
  final Color color;

  var handleR;

  _HandlePainter({this.handleR, required this.offset, this.color = Colors.blue})
      : super(repaint: offset) ;

  @override
  void paint(Canvas canvas, Size size) {
    canvas.clipRect(Offset.zero & size);
    final bgR = size.width / 2 - handleR;
    canvas.translate(size.width / 2, size.height / 2);
    _paint.style = PaintingStyle.fill;
    _paint.color = color.withAlpha(100);
    canvas.drawCircle(Offset(0, 0), bgR, _paint);

    _paint.color = color.withAlpha(150);
    canvas.drawCircle(
        Offset(offset.value.dx, offset.value.dy), handleR, _paint);

    _paint.color = color;
    _paint.style = PaintingStyle.stroke;
    canvas.drawLine(Offset.zero, offset.value, _paint);
  }

  @override
  bool shouldRepaint(_HandlePainter oldDelegate) =>
      oldDelegate.offset != offset ||
      oldDelegate.color != color ||
      oldDelegate.handleR != handleR;
}
```

---

5. 监听器

> 像 `Switch` 、 `Slider` 这样的组件都会向外界提供回调方法，我们也可以将 距离 和 角度 回调给使用者。
> 比如下面通过控制器旋转来对蓝色的 Container 进行操作。

```dart
---->[p12_gesture/s01_ctrl3/handle_widget.dart]----
class HandleWidget extends StatefulWidget {
  final double size;
  final double handleRadius;
  final void Function(double rotate, double distance) onMove; // 定义回调参数
    HandleWidget( {Key? key, this.size = 160, this.handleRadius = 20.0,
      required this.onMove}) : super(key: key);
// 略同...

parser(DragUpdateDetails details) {
  final offset = details.localPosition;
  double dx = 0.0;
  double dy = 0.0;
  dx = offset.dx - widget.size / 2;
  dy = offset.dy - widget.size / 2;
  var rad = atan2(dx, dy);
  if (dx < 0) {
    rad += 2 * pi;
  }
  var bgR = widget.size / 2 - widget.handleRadius;
  var thta = rad - pi / 2; //旋转坐标系90度
  var d = sqrt(dx * dx + dy * dy);
  if (d > bgR) {
    dx = bgR * cos(thta);
    dy = -bgR * sin(thta);
  }
   widget.onMove(thta, d); // <--- 控制器回调
  _offset.value = Offset(dx, dy);
}

reset(DragEndDetails details) {
  _offset.value = Offset.zero;
  widget.onMove(0, 0); // <--- 控制器回调
}
```

> 使用方式和 Slider 等控制组件基本一致。

```dart
---->[p12_gesture/s01_ctrl3/main.dart]----
class HomePage extends StatefulWidget {
  @override
  _HomePageState createState() => _HomePageState();
}

class _HomePageState extends State<HomePage> {
  double _rotate = 0;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Row(
          mainAxisSize: MainAxisSize.min,
          children: [
            Transform.rotate(
              angle: _rotate,
              child: Container( color: Colors.blue, width: 100, height: 100 ),
            ),
            HandleWidget(  onMove: _onMove ),
          ],
        ),
    );
  }

  void _onMove(double rotate, double distance) {
    setState(() {
      _rotate= rotate;
    });
  }
}
```

二、绘制刻度尺

> 通过绘制如下的滑动刻度尺，来练习一下 水平滑动的操作 以及刻度的绘制。 其中包含对 边界值的限值 、 一些数值计算 等实用的技巧。

1.组件介绍

> 需要指定的属性有，最大值 max 、最小值 min、刻度尺将可以在之间滑动。
> 并且给出一个 `onChanged` 的回调方法，将当前滑到的值传给用户。在画板中最重要的数据是水平滑动的距离，所以在组件中使用一个 `ValueNotifier<double>` 的对象传给画板，用于更新重绘。
> 通过 `GestureDetector` 在水平方向进行监听，核心是进行移动过程中的点位计算，逻辑我放在 `_parser` 方法中。

> 通过 `GestureDetector` 在水平方向进行监听，核心是进行移动过程中的点位计算，逻辑我放在 `_parser` 方法中。


```dart
---->[p12_gesture/s04_ruler/rouler_chooser.dart]----
class RulerChooser extends StatefulWidget {
  final Size size;
  final void Function(double) onChanged;
  final int min;
  final int max;

  RulerChooser(
      {Key? key,
      required this.onChanged,
      this.max = 200,
      this.min = 100,
      this.size = const Size(240.0, 60)})
      : super(key: key);

  @override
  _RulerChooserState createState() => _RulerChooserState();
}

class _RulerChooserState extends State<RulerChooser> {
  ValueNotifier<double> _dx = ValueNotifier(0.0);

  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onPanUpdate: _parser,
      child: CustomPaint(
          size: widget.size,
          painter: _HandlePainter(dx: _dx, max: widget.max, min: widget.min)),
    );
  }

  double dx = 0;
  void _parser(DragUpdateDetails details) {
    // 见下
  }
}
```

> 在看 `_parser` 方法之前，先看一下定义的一些常量。其中刻度宽和刻度间距都现在常量里，当然你可以将它们暴露出去，让用户去指定，这里为了方便点，就做成常量了。
> Dart 源码中私有常量命名使用 `_kXXXX` 。

```dart
const double _kHeightLevel1 = 20; // 短线长
const double _kHeightLevel2 = 25; // 5 线长
const double _kHeightLevel3 = 30; //10 线长
const double _kPrefixOffSet = 5; // 左侧偏移
const double _kVerticalOffSet = 12; // 线顶部偏移
const double _kStrokeWidth = 2; // 刻度宽
const double _kSpacer = 4; // 刻度间隙
const List<Color> _kRulerColors = [  // 渐变色
  Color(0xFF1426FB),
  Color(0xFF6080FB),
  Color(0xFFBEE0FB),
];
const List<double> _kRulerColorStops = [0.0, 0.2, 0.8];
```

> `_parser` 方法中校验偏移范围，将移动的距离进行累加，设置给 `_dx` 后，就会触发画布的重绘。
> 在最后通过 `widget.onChanged` 将刻度值回调出去。

```dart
double dx = 0;

void _parser(DragUpdateDetails details) {
  dx += details.delta.dx;
  if (dx > 0) {
    dx = 0.0;
  }
  var limitMax = -(widget.max - widget.min) * (_kSpacer + _kStrokeWidth);
  if (dx < limitMax) {
    dx = limitMax;
  }
  _dx.value = dx;

  if (widget.onChanged != null) {
    widget.onChanged(details.delta.dx / (_kSpacer + _kStrokeWidth));
  }
}
```

2.画板绘制

> 画板的绘制也比较容易，值得注意的是一些小细节的处理:
> 比如画布开始要根据线宽和前部预留部分进行 x 偏移，小三角形的偏移，要对准第一格线。
> 绘制刻度时，根据 `%5` 和 `%10` 来进行线长的更改，每次画完一个刻度，画布移动 线宽+间距 的长度。

```dart
class _HandlePainter extends CustomPainter {
  var _paint = Paint();
  Paint _pointPaint = Paint();

  final ValueNotifier<double> dx;
  final int max;
  final int min;

  _HandlePainter({required this.dx, required this.max, required this.min}) : super(repaint:
    _paint
      ..strokeWidth = _kStrokeWidth
          ..shader = ui.Gradient.radial(
          Offset(0, 0), 25, _kRulerColors, _kRulerColorStops, TileMode.mirror);
    _pointPaint
      ..color = Colors.purple
      ..strokeWidth = 4
      ..strokeCap = StrokeCap.round;
  }

  @override
  void paint(Canvas canvas, Size size) {
    canvas.clipRect(Offset.zero & size);

    drawArrow(canvas);
    canvas.translate(_kStrokeWidth / 2 + _kPrefixOffSet, _kVerticalOffSet);
    canvas.translate(dx.value, 0);
    drawRuler(canvas);
  }

  // 绘制刻度
  void drawRuler(Canvas canvas) {
    double y = _kHeightLevel1;
    for (int i = min; i < max + 5; i++) {
      if (i % 5 == 0 && i % 10 != 0) {
        y = _kHeightLevel2;
      } else if (i % 10 == 0) {
        y = _kHeightLevel3;
        _simpleDrawText(canvas, i.toString(),
            offset: Offset(-3, _kHeightLevel3 + 5));
      } else {
        y = _kHeightLevel1;
      }
      canvas.drawLine(Offset.zero, Offset(0, y), _paint);
      canvas.translate(_kStrokeWidth + _kSpacer, 0);
    }
  }

  // 绘制三角形尖角
  void drawArrow(Canvas canvas) {
    var path = Path()
      ..moveTo(_kStrokeWidth / 2 + _kPrefixOffSet, 3)
      ..relativeLineTo(-3, 0)
      ..relativeLineTo(3, _kPrefixOffSet)
      ..relativeLineTo(3, -_kPrefixOffSet)
      ..close();
    canvas.drawPath(path, _pointPaint);
  }

  void _simpleDrawText(Canvas canvas, String str,
      {Offset offset = Offset.zero}) {
    var builder = ui.ParagraphBuilder(ui.ParagraphStyle())
      ..pushStyle(
        ui.TextStyle(
            color: Colors.black, textBaseline: ui.TextBaseline.alphabetic),
      )
      ..addText(str);
    canvas.drawParagraph(
        builder.build()
          ..layout(ui.ParagraphConstraints(width: 11.0 * str.length)),
        offset);
  }

  @override
  bool shouldRepaint(_HandlePainter oldDelegate) =>
      oldDelegate.dx != dx || oldDelegate.min != min || oldDelegate.max != max;
}
```

> 通过这两个案例，你应该了解如何在画板中使用手势监听，来进行交互，这样所有的拼图都已经在手。下面将进入 Path 的世界，着重看一下对于 Path 的使用技巧，这也是绘制的 最核心难点 。