能力介绍 & API

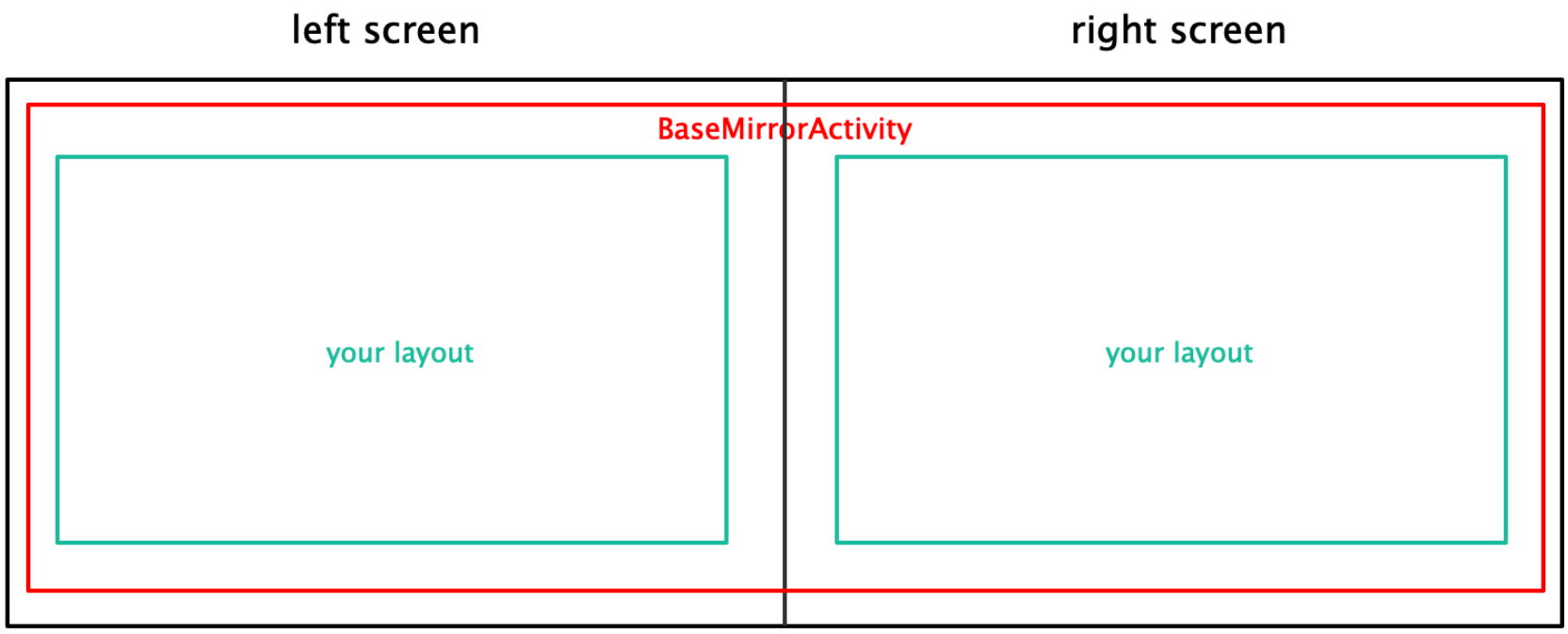
1. 合目

为了实现合目，除了需要相同的UI布局左右各放置一份外，每次对左边的view进行修改，同样的操作也需要在右边的布局也重新设置一遍，这两点都会引入不少繁琐的模版代码。为了解决这个问题，基于ViewBinding，SDK中提供了**BindingPair**工具类，将左右布局对应的ViewBinding对象传入它的构造函数后，通过**updateView**方法同时操作左右的布局，通过**setLeft**方法只操作左边的布局，通过checkIsLeft方法，在updateView的block中判断当前是左边的布局还是右边的布局。

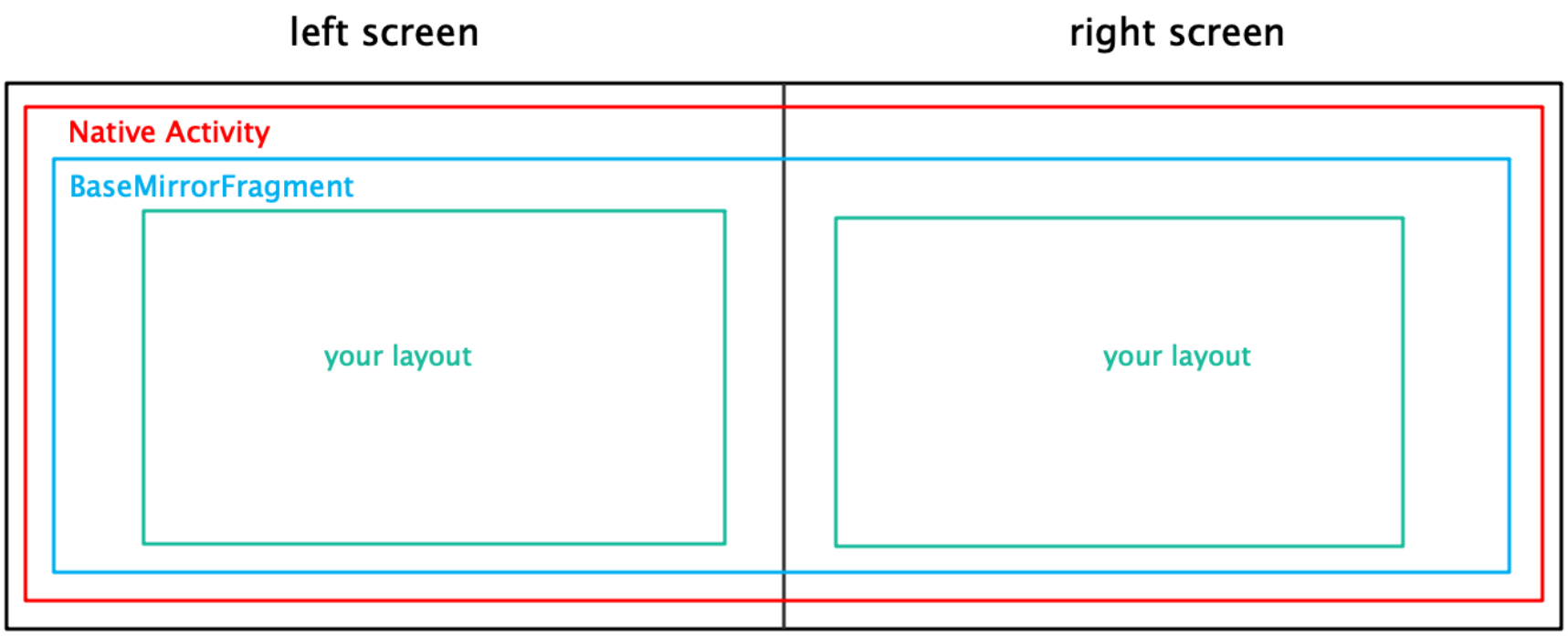
1.1 合目组件

合目镜像操作主要由基于**ViewBinding**的**BindingPair**工具类完成，基于BindingPair封装了以下UI组件：

- Activity级别：**BaseMirrorActivity**，实现了Activity自动镜像布局视图的逻辑，开发者只用关注业务的UI需求，按照原生的xml布局方式开发布局文件就能实现合目的效果，具体可以参考sample中的FusionVisionActivity或者所有的BaseMirrorActivity子类



- Fragment级别：**BaseMirrorFragment**，实现了Fragment级别的合目封装，添加方式与原生的Fragment一致，但是注意不能添加到BaseMirrorActivity中去,具体实现可参考sample中的FragmentDemoActivity



- View级别：基于组合的**MirrorContainerView** 和基于继承的**BaseMirrorContainerView**

MirrorContainerView的使用可参考MirrorContainerViewActivity，注意一定要实现**mirrorContainer.bindTo**方法，否则合目不生效；

BaseMirrorContainerView的使用可参考com.ffalcon.mercury.android.sdk.demo.ui.wedget.TitleView,需要注意的是MirrorView也同样不能添加到BaseMirrorActivity中

- Toast: **FToast**，支持合目的通用toast封装，具体使用可参考FusionVisionHomeActivity
- Dialog：**FDialog**，支持合目的通用Dialog封装，具体使用可参考DialogActivity

上述的合目组件除了FToast和FDialog是独立于页面的组件外，其他的合目组件可以根据业务需求合理的进行组装，以达到完美的双屏合目效果

1.2 updateView & setLeft

mBindingPair.updateView{}:

上述的合目组件均由**BindingPair**工具类实现了布局左右映射的效果，一份布局文件同时映射到左右两个屏幕，所以实质上在内存中还是同时存在两份布局的，为了处理繁琐的镜像操作，SDK提供了**mBindingPair.updateView**方法来同时处理左右布局同一个组件的更新操作。

eg:

代码块

```
//在updateView block中的所有操作均会映射到左右两个布局文件中
mBindingPair.updateView {
    tvTitle.text = "my title"
}
//等价于
mBindingPair.left.tvTitle.text = "my title"
mBindingPair.right.tvTitle.text = "my title"
```

需要注意的是，由于该特性，推荐的做法是只在updateView block中做UI层面的更新操作，**如果在该方法中进行了对外部源数据的修改或者绑定了事件，可以预知的行为是源数据的修改和绑定的事件均会执行两次**，造成不可预知的后果！！！如果有外部数据更新的操作或者对具体组件进行事件绑定的逻辑，推荐的做法是使用**mBindingPair.setLeft** 方法

mBindingPair.setLeft{}:

```
mBindingPair.setLeft {
    focusHolder.addFocusTarget(
    //一般的做法是将事件绑定在左边的视图组件上面
        FocusInfo(
            btnEvent,
            eventHandler = { action -> handleAction(action) },
            focusChangeHandler = { hasFocus ->
            //如果响应某个事件需要同步更新两边的组件，这个时候可以再次调用updateView的方法
                mBindingPair.updateView {
                    triggerFocus(hasFocus, btnEvent, mBindingPair.checkIsLeft(this))
                }
            }
        )
    )
    focusHolder.currentFocus(mBindingPair.left.btnEvent)
}
```

当然如果复杂业务需要**updateView**中进行逻辑整合又不期望被调用两次的话，也可以在updateView的block中对当前执行区域进行判断，具体的实现如下：

代码块

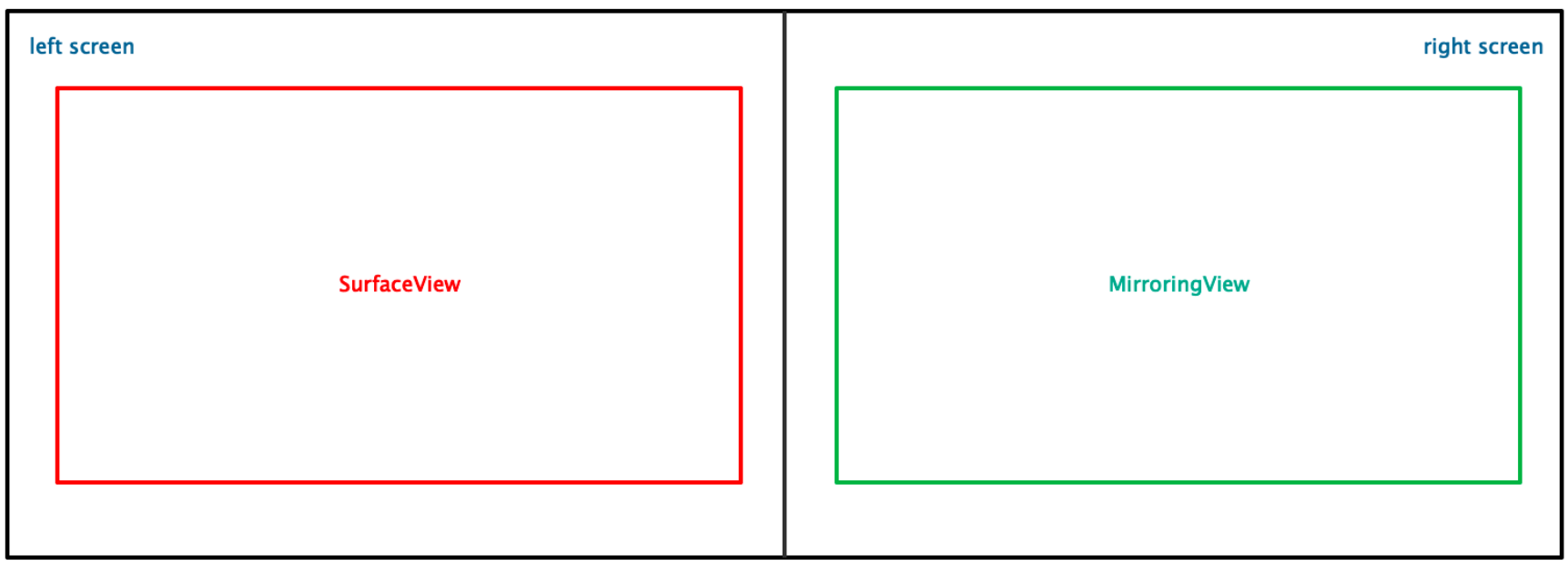
```
mBindingPair.updateView {
    //do something.......

    //check isLeft
    if (mBindingPair.checkIsLeft(this)){
        // only left do something.......
    }
}
```

更多具体的场景实现可以参考sample中的代码

1.3 视频合目（SurfaceView）

上文中的合目组件只能对于常规的UI组件进行镜像合目的操作，但对于视频播放需要用到SurfaceView这种类型的场景，还需要进行其他的特殊处理。rayneo对于这种合目场景提供了**MirroringView**组件进行支持，具体使用详情可参考sample中的VideoPlayActivity，主要的步骤如下：

1.确保布局中视频播放组件（SurfaceView）与镜像组件（MirroringView）分别处于不同的屏幕



2.将SurfaceView设置到MirroringView中，并开启Mirroring，此时surface开始渲染的同时也会将画面镜像到MirroringView中去

代码块

```
    binding.textureView.let { leftTexture ->
        binding.mirrorView.setSource(leftTexture)
        binding.mirrorView.startMirroring()
    }

    ........

    //在必要的地方停止
    override fun onDestroy() {
        super.onDestroy()
        binding.mirrorView.stopMirroring()
        mPlayer?.release()
    }
```

2. 焦点管理

2.1 焦点说明

眼镜上的屏幕无法直接触控，无法像手机可以直接选中目标View，所以需要**自己维护焦点**的切换，一般**前滑**和**后滑**手势用来切换焦点，**单击**手势用来触发焦点View的事件响应，**双击**手势退出焦点返回上一级。

焦点管理最简单的实现，可以通过维护一个索引值index，不同的索引值对应不同的能获取焦点的view，前滑和后滑手势触发焦点位的切换，单击响应对应view的点击事件，双击释放焦点给上一级或退出页面。但对于复杂页面，这样简单粗暴的实现方式，并不容易维护。

代码块

```
    //1.构建FocusHolder对象
    val focusHolder = FocusHolder(true)
    ....
    // 2. 以左边布局中的view，作为focus对象的target，
    mBindingPair.setLeft {
        //2.1 构建focusInfo对象
        val btn1Info = FocusInfo(
            btn1,
            eventHandler = { action ->
            // 处理事件响应
                when (action) {
                    is TempleAction.Click -> {
                        FToast.show("bt1 click")
                    }
                    else -> Unit

                }
            },
            focusChangeHandler = { hasFocus ->
            // 处理焦点切换
                mBindingPair.updateView {
                    btn1.setBackgroundColor(getColor(if(hasFocus) R.color.purple_200 else R.color.black))
                }
            }
        )
        //2.2. 添加同一层级的可以获取焦点的多个FocusInfo对象
        focusHolder.addFocusTarget(
             // 第一个焦点位
            btn1Info,
            // 第二个焦点位
            FocusInfo(
                btn2,
```

```
            eventHandler = {action ->
                when(action) {
                    is TempleAction.Click -> {
                        FToast.show("bt2 click")
                    }
                    else -> Unit
                }
            },
            focusChangeHandler = { hasFocus ->
                mBindingPair.updateView {
                    btn2.setBackgroundColor(getColor(if(hasFocus) R.color.purple_200 else R.color.black))
                }
            }
        ),
        // 第xxx个焦点位
        ......
    )
    // 2.3 设置默认的焦点位
    focusHolder.currentFocus(mBindingPair.left.btn1)
}
// 3. 最终构建FixPosFocusTracker对象
fixPosFocusTracker = FixPosFocusTracker(focusHolder).apply {
    //3.1 设置当前获取到焦点，如果没有获取焦点，则不会响应任何事件
    focusObj.hasFocus = true
}
......
//4. 与TempleAction事件流对接
lifecycleScope.launch {
    repeatOnLifecycle(Lifecycle.State.RESUMED) {
        templeActionViewModel.state.collect {
            FLogger.i("DemoActivity", "action = $it")
            when (it) {
                is TempleAction.DoubleClick -> {
                    // 统一处理双击退出逻辑
                    finish()
                }
                else -> fixPosFocusTracker?.handleFocusTargetEvent(it)
            }
        }
    }
}
```

这样同一层级的焦点，就可以按照以上流程来处理。而FixPosFocusTracker本身又可以作为FocusInfo的target，层层嵌套后，即可应对复杂页面的焦点切换场景。

对于RecyclerView来说，焦点一般在某个Item View上，焦点的切换又是另外一种情况。根据焦点位是否固定，可以分为**列表+固定焦点位** 和 **列表+移动焦点位**两种情况，SDK分别提供了**RecyclerViewSlidingTracker** 和**RecyclerViewFocusTracker**两个工具类来处理，具体代码参考SDKDemo中的FixedFocusPosRVActivity和MovedFocusPosRVActivity两个页面。

针对其他情况，比如自定义View中的整体的焦点切换，可以通过实现IFocusable接口来统一处理，只有在hasFocus属性为true时，才响应事件。

同时RecyclerViewSlidingTracker、RecyclerViewFocusTracker、IFocusable对象都可以作为FocusInfo的target。

2.2 动态焦点

> 根据用户交互、视线追踪、场景变化或设备姿态，**实时调整视图中 “焦点视图” 的位置、状态、显示内容或可交互优先级** —— 简单说就是让 “用户当前需要关注 / 可操作的核心视图” 随场景动态变化，而非固定在某个位置或状态
>
> 1. 用户转头看左侧菜单 → 焦点自动从右侧弹窗切换到左侧菜单；
> 2. 场景中出现新的交互目标（如虚拟按钮）→ 动态添加该按钮为新的焦点视图；

在眼镜合目的视图中动态调整焦点视图，需要强依赖视图组件，动态添加焦点视图示例代码如下（详情请参考sample中的DynamicFocusTargetActivity）：

代码块

```
private fun addDynamicFocus() {
    // 扩展函数API动态添加焦点View
    // 使用可变引用，因为handle需要在eventHandler中使用
    var handle: FocusViewHandle<View>? = null
    handle = mBindingPair.addFocusView(
        parent = mBindingPair.left.llParent,
        viewFactory = {
        //这里动态添加一个button，实际以需求为准
            Button(this@DynamicFocusTargetActivity).apply {
                text = "dynamic focus target"
                background = null
            }
        },
        focusHolder = focusHolder,
        focusConfig = {
            // 设置布局参数（eg: LinearLayout，以实际视图为准）
            layoutParamsFactory = { parent, view ->
                LinearLayout.LayoutParams(
                    150.dp,
                    ViewGroup.LayoutParams.WRAP_CONTENT
                ).apply {
                    setMargins(
                        20.dp,
                        5.dp,
                        20.dp,
                        5.dp
                    )
                }
            }

            //这里处理视图的响应事件
            eventHandler = { action ->
                when (action) {
                    is TempleAction.Click -> {
                        FToast.show("dynamic focus target click!!")
                    }
```

```
37
38                      else -> Unit
39                  }
40              }
41              //这里处理视图焦点切换
42              onFocusChange = { view, hasFocus, isLeft ->
43                  triggerFocus(hasFocus, view, isLeft)
44              }
45          }
46      )
47
48      // focusHandles是一个管理动态添加焦点视图的可变列表，可酌情添加
49      focusHandles.add(handle)
50
51      currentDynamicFocus = handle
52  }
```

移除动态焦点视图如下：

代码块

```
1   currentDynamicFocus.clearFocusView()
```

3. TP管理

目前**右镜腿**可以触发MotionEvent TP事件，相对于手机的二维TP事件，左右镜腿产生的TP事件，只有一维，即X坐标动态变化，Y坐标为固定值。除了左右镜腿外，还有**戒指**配件，配对连接后也会产生TP事件。

事件响应：TP事件分发至Activity或View级别后，在分发流程中会将原始MotionEvent传递给**TouchDispatcher**，TouchDispatcher会将原生TP事件识别为**单击、长按、双击、三击、前滑&后滑**等手势（部分手势带有系统默认音效），并以**CommonTouchCallback**接口暴露出去，调用方只需根据对应手势完成自己业务逻辑即可。

基于**TouchDispatcher**和**CommonTouchCallback**，针对Activity封装了BaseTouchActivity和BaseEventActivity。BaseTouchActivity自动注册了手势监听。BaseEventActivity继承自BaseTouchActivity，并将对应的手势转成了kotlin Flow事件流（BaseMirrorActivity继承自BaseEventActivity），将事件映射为了TempleAction子类对象，事件监听示例代码如下:

代码块

```
1   class TPEventActivity : BaseMirrorActivity<ActivityTpEventBinding>() {
2       private var fixPosFocusTracker: FixPosFocusTracker? = null
3       override fun onCreate(savedInstanceState: Bundle?) {
4           super.onCreate(savedInstanceState)
5           initFocusTarget()
6           initEvent()
7       }
8
9       private fun initFocusTarget() {
10          val focusHolder = FocusHolder(false)
11          mBindingPair.setLeft {
12              focusHolder.addFocusTarget(
13                  FocusInfo(
14                      btnEvent,
15                      eventHandler = { action -> handleAction(action) },
16                      focusChangeHandler = { hasFocus ->
17                          mBindingPair.updateView {
18                              triggerFocus(hasFocus, btnEvent, mBindingPair.checkIsLeft(this))
```

```
                        }
                    }
                )
            )
            focusHolder.currentFocus(mBindingPair.left.btnEvent)
        }
        fixPosFocusTracker = FixPosFocusTracker(focusHolder).apply {
            focusObj.reqFocus()
        }
    }

    private fun handleAction(action: TempleAction) {
        when (action) {
            is TempleAction.LongClick -> {
                FToast.show("LongClick")
            }

            is TempleAction.Click -> {
                FToast.show("Click")
            }

            is TempleAction.DoubleClick -> {
                FToast.show("DoubleClick")
                finish()
            }

            is TempleAction.TripleClick -> {
                FToast.show("TripleClick")
            }

            is TempleAction.SlideBackward -> {
                FToast.show("SlideBackward")
            }

            is TempleAction.SlideForward -> {
                FToast.show("SlideForward")
            }

            is TempleAction.SlideUpwards -> {
                FToast.show("SlideUpwards")
            }

            is TempleAction.SlideDownwards -> {
                FToast.show("SlideDownwards")
            }


            else -> Unit
        }
    }

    private fun initEvent() {
        lifecycleScope.launch {
            repeatOnLifecycle(Lifecycle.State.RESUMED) {
                templeActionViewModel.state.collect {
                    fixPosFocusTracker?.handleFocusTargetEvent(it)
                }
            }
        }
    }

    private fun triggerFocus(hasFocus: Boolean, view: View, isLeft: Boolean) {
```

```
81          view.setBackgroundColor(getColor(if (hasFocus)
    com.ffalcon.mercury.android.sdk.R.color.color_rayneo_theme_0 else R.color.black))
82          // 3D效果
83          make3DEffectForSide(view, isLeft, hasFocus)
84      }
85  }
```

如果当前视图没有具体的组件需要监听TP事件，只需要实现类似原生Activity返回的效果，那么只需要与TempleAction事件流对接，眼镜推荐的做法是双击退出页面，示例代码如下:

代码块

```
1   class DialogActivity : BaseMirrorActivity<LayoutDialogBinding>() {
2
3       override fun onCreate(savedInstanceState: Bundle?) {
4           super.onCreate(savedInstanceState)
5           initEvent()
6       }
7
8       private fun initEvent() {
9           lifecycleScope.launch {
10              repeatOnLifecycle(Lifecycle.State.RESUMED) {
11                  templeActionViewModel.state.collect {
12                      FLogger.i("DemoActivity", "action = $it")
13                      when (it) {
14                          is TempleAction.DoubleClick -> {
15                              finish()
16                          }
17                          is TempleAction.Click -> {
18                              showDialog()
19                          }
20                          else -> Unit
21                      }
22                  }
23              }
24          }
25      }
```

如果之前使用过X2的SDK，那么X2的应用可以迁移到X3上，X3 SDK相对X2 新增了DeviceUtil的isX3Device函数，用于判断是否是RayneoX3

代码块

```
1   if(DeviceUtil.isX3Device()){
2       // in Rayneo X3
3   }else{
4       // in Rayneo X2
5   }
```

3.1 Rayneo X3 SDK中CommonTouchBack接口改动

增加了如下函数，在X2系列产品中不包含下面的事件

代码块

```
1   //上滑
2   open fun onTPSlideUpwards(args: FlingArgs): Boolean {
3       return false
```

```
    }
    //下滑
    open fun onTPSlideDownwards(args: FlingArgs): Boolean {
        return false
    }
    //双指点击
    open fun onTPDoubleFingerClick() {
    }
    //双指长按
    open fun onTPDoubleFingerLongClick() {
    }

```

修改了如下函数，增加了vertical，表示是否是垂直方向上的滑动数据。

代码块

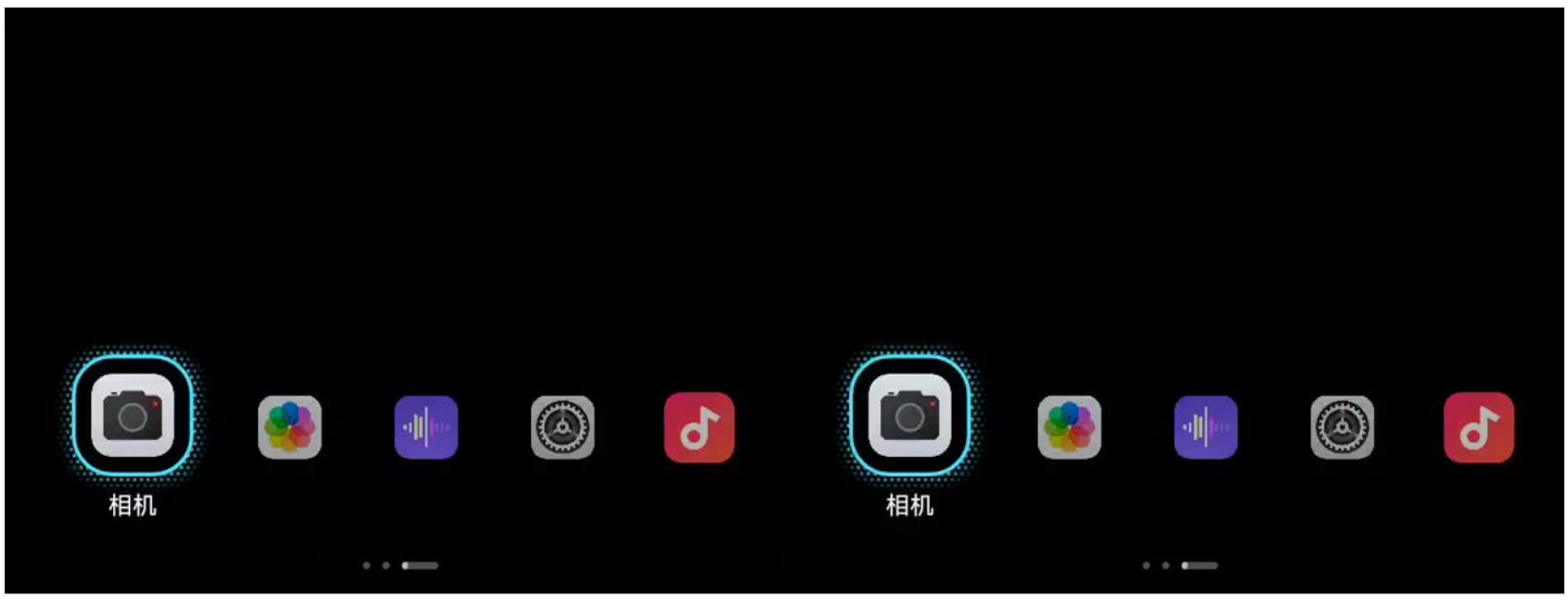
```
onTPSlideContinuous(delta: Float, longClick: Boolean = false,vertical:Boolean=false)
```

并且在CommondTouchBack中，增加了属性filterMode,当CommonTouchCallback设置为*OnlyX的时候onTPSlideContinuous只返回X轴滑动数据*，当CommonTouchCallback设置为*OnlyY的时候，onTPSlideContinuous只返回Y轴滑动的数据。**只有RayneoX3支持此项设置**。*

并且TempleAction.TpSlideContinuous也增加了滑动的方向是否是X轴还是Y轴。

3.2 SlideBackward & SlideForward

需要特别说明一下前滑事件TempleAction.SlideForward与后滑事件TempleAction.SlideBackward的处理，在Rayneo眼镜中触控板的滑动与UI的映射关系分为两种，一种是自然模式，一种是非自然模式，用户可以在设置界面里面进行设置，设置路径参考下方视频：



自然模式(系统默认)

触控板与UI映射关系如下图：

该模式下从镜腿方向往镜片方向滑动会被系统识别为TempleAction.SlideBackward事件，反过来会被识别为TempleAction.SlideForward事件

非自然模式下

触控板与UI映射关系如下图：

该模式下从镜腿方向往镜片方向滑动会被系统识别为TempleAction.SlideForward事件，反过来会被识别为TempleAction.SlideBackward事件

4. 跟手效果

基于TouchDispatcher和CommonTouchCallback，对于一般场景来说已经够用，但如果想实现手机端列表跟手滑动的效果，则无法实现，原因有2个：

- TouchDispatcher其实是对原始事件的二次处理，而列表跟手效果，需要将原始事件直接喂给对应的列表;
- 原始事件的Y坐标为固定值，已经写死，而目标列表可横可竖，且只响应落在列表区域内的事件。这意味着对于横向列表，需要根据列表中心位置修改Y坐标值；对于竖向列表，需要将TP事件对X、Y坐标对调，然后再固定X坐标值。

目前跟手效果的整体实现思路是：原始事件仍旧喂给TouchDispatcher，以触发各类手势，但只响应**单击、双击、长按等**操作，**前滑&后滑**操作不响应。同时RecyclerView**固定焦点位**，截获原始事件流，根据原始事件生成新的MotionEvent，修改Y坐标后，将新TP事件喂给RecyclerView，之后回收新生成的MotionEvent；新生成的TP事件流，只实现跟手效果，不做点击等事件响应。

按场景，可以分为以下3种情况：

- **固定焦点位+ 列表滚动**跟手效果：核心工具类是**RecyclerViewSlidingTracker**
- **移动焦点位+ 列表** 跟手效果：按**滑动距离**方式，核心工具类是**RecyclerViewFocusTracker**

- **移动焦点位 + 固定位置View** 跟手效果：核心工具类是**FixPosFocusTracker**

其中，移动焦点位跟手效果的相关的核心逻辑是：滑动一定距离（比如50dp）后，根据滑动方向，触发焦点前后、上下切换逻辑。

固定焦点位时，需要将TP事件修改坐标后，传给RecycerView，示例代码如下：

代码块

```
// 监听原始事件，实现跟手效果
recyclerViewSlidingTracker.observeOriginMotionEventStream(
    motionEventDispatcher
) { event ->
    // 生产信的MotionEvent，并修改坐标，X与Y坐标是否互换位置，根据使用场景，动态调整
    MotionEvent.obtain(
        event.downTime,
        event.eventTime,
        event.action,
        320f,
        event.x,
        event.metaState
    ).apply {
        val e = this
        FLogger.d(
            "onReceiveEvent: x = ${e.x}, y = ${e.y},action = ${e.actionName()}, deviceId = ${e.deviceId}"
        )
    }
}

lifecycleScope.launchWhenResumed {
    val templeActionViewModel =
        ViewModelProvider(this@FixedFocusPosRVActivity).get<TempleActionViewModel>()
    templeActionViewModel.state.collect {
        if (!favoriteTracker.focusObj.hasFocus || !this.isActive || it.consumed) {
            return@collect
        }
        recyclerViewSlidingTracker.handleActionEvent(it) { action ->
            when (action) {
                is TempleAction.DoubleClick -> {
                    finish()
                }
                is TempleAction.Click -> {
                    if (!action.consumed) {
                        (mBindingPair.left.recyclerView.adapter as FixedFocusPosAdapter)
                            .getCurrentData()?.apply {
                                FToast.show(displayName)
                            }
                    }
                }
                else -> {}
            }
        }
    }
}
```

其中motionEventDispatcher是BaseTouchActivity成员变量，用于截获原始的事件流；observeOriginMotionEventStream函数，用于修改X、Y坐标，将原始事件流转为目标RecyclerView(左屏幕中)可以响应的事件流。完整代码，参考FixedFocusPosRVActivity。

5. 3D效果实现

目前在Android原生开发环境下，实现了基于**双目视差**的**伪3D效果**，基本原理是让左右布局中相同位置的View，**左边**布局的view**向右**偏移，**右边**布局的view**向左**偏移，从而产生视差，不同的偏移值会产生不同的景深感觉，合目后形成3D视觉效果。SDK提供了**make3DEffect**() 和**make3DEffectForSide**()，来实现该功能。

代码块

```
// 同时设置左右两个View
make3DEffect(left.slideView, right.slideView, true, 10f)
// 左右两边分别单独调用
make3DEffectForSide(this.ivSelectHover, isLeft, true)
```

6. 音频开发

在RayNeo眼镜上进行音频业务的开发，实现步骤与一般的android平台音频应用开发类似，如果需要调用MIC进行声音的收录，RayNeo眼镜上封装特有的MIC收音模式，在X2跟X3上略有不同：

X2支持的收音模式：

每个apk在调用mic的时候，去设置状态通知audioHAL; AudioManager.setparameter("audio_source=xxxx")；

模式	描述	示例
audio record:	**暂未配置**	setParameters("audio_source_record=sound")
camera record:	**调用镜腿2颗mic，声音全收进来，无降噪算法**	setParameters("audio_source_record=camcorder")
translation:	**调用3颗麦克风，不收佩戴者声音，收外部声音**	setParameters("audio_source_record=translation")
voice assistant:	**调用镜腿2颗mic，主要收佩戴者声音**	setParameters("audio_source_record=voice assistant")

为了确保各个应用都有自己的mic组合，请在退出record的时候，也设置下setParameters("audio_source_record=off");

X3支持的收音模式：

模式	前置条件(AudioRecord)	setParameters	描述
OFF	**退出录音**	**audioManager.setParameters("audio_source_record=off")**	**apk退出时记得释放麦克风，退出record的时候需要设置**
RECORD_TRANSLATION	device.id **== SPEAKER_MIC ; CHANNEL_IN_MONO; 送翻译引擎的采样率一般设置16k;**	ExampleRecordTranslation.java	调用前麦克风，录制周围人声，适合会议记录/现场翻译
CAMCORDER	device.id **== SPEAKER_MIC; CHANNEL_IN_STEREO; 采样率根据需要设定，最高不超出48kHz;**	ExampleRecordCamcorder.java	左右镜腿麦克风，立体声录音，全向收音，适合录像场景使用
VOICE_RECOGNITION	device.id **== SPEAKER_MIC; CHANNEL_IN_MONO; 识别引擎一般要求Sample rate为16kHz，单通道;**	ExampleRecordRecognition.java	调用三颗麦克风，只收佩戴者声音，送识别引擎识别，不考虑音质，适合于语音助手，导航场景等
VOICE_COMMUNICATION	device.id **== SPEAKER_MIC**	ExampleRecordCommunication.java	调用前麦和右方麦克风，只收佩戴者的声音，适用于网络通话、网络会议等

7. camera开发

在RayNeo眼镜上进行camera应用的开发与一般的android平台camera应用开发并无什么区别，详情请参考developer.android.com，同时也需要注意权限的申请以及资源的释放。

这里推荐使用camera2 API，sample中有详细的示例演示了如何在眼镜应用中调用camera，并在两个屏幕上进行预览以及调用camera获取一帧图片的逻辑。具体的实现方式可以参考CameraActivity。

如果需要查看眼镜camera支持的分辨率以及支持的预览参数，用户可自行枚举查看，示例中也有一个简单的camera枚举方法仅供参考：

代码块

```
private fun enumerateCameraResolutions() {
    val cameraManager = getSystemService(CAMERA_SERVICE) as CameraManager
    val cameraIdList = cameraManager.cameraIdList

    for (cameraId in cameraIdList) {
        val characteristics = cameraManager.getCameraCharacteristics(cameraId)
        val map = characteristics.get(CameraCharacteristics.SCALER_STREAM_CONFIGURATION_MAP)

        if (map != null) {
            val previewSizes = map.getOutputSizes(SurfaceTexture::class.java)
            val pictureSizes = map.getOutputSizes(ImageFormat.JPEG)

            FLogger.d("Camera ID: $cameraId")
            FLogger.d("Supported Preview Sizes:")
            for (size in previewSizes) {
                FLogger.d("  ${size.width}x${size.height}")
            }

            FLogger.d("Supported Picture Sizes:")
            for (size in pictureSizes) {
                FLogger.d("  ${size.width}x${size.height}")
            }
        }
    }
}
```

snapdragonCamera的源码示例（AE收敛状态=2则可以正常取图）：
https://github.com/Lh1600852534/snapdragonCamera/blob/master/src/com/android/camera/CaptureModule.java

VGA camera

Rayneo x3眼镜上还有一个独立的VGA摄像头，专门用于空间定位，支持的参数如下：

代码块

```
2025-09-25 18:13:32.490  1583-1583  camera                  com.ffalcon.mercury.android.sdk      D
Camera ID: 1
2025-09-25 18:13:32.490  1583-1583  camera                  com.ffalcon.mercury.android.sdk      D
Supported Preview Sizes:
2025-09-25 18:13:32.491  1583-1583  camera                  com.ffalcon.mercury.android.sdk      D
640x480
2025-09-25 18:13:32.491  1583-1583  camera                  com.ffalcon.mercury.android.sdk      D
640x400
2025-09-25 18:13:32.491  1583-1583  camera                  com.ffalcon.mercury.android.sdk      D
640x360
```

```
 6  2025-09-25 18:13:32.491  1583-1583  camera                com.ffalcon.mercury.android.sdk      D
    480x360
 7  2025-09-25 18:13:32.491  1583-1583  camera                com.ffalcon.mercury.android.sdk      D
    352x288
 8  2025-09-25 18:13:32.491  1583-1583  camera                com.ffalcon.mercury.android.sdk      D
    320x240
 9  2025-09-25 18:13:32.491  1583-1583  camera                com.ffalcon.mercury.android.sdk      D
    320x180
10  2025-09-25 18:13:32.491  1583-1583  camera                com.ffalcon.mercury.android.sdk      D
    176x144
11  2025-09-25 18:13:32.491  1583-1583  camera                com.ffalcon.mercury.android.sdk      D
    Supported Picture Sizes:
12  2025-09-25 18:13:32.491  1583-1583  camera                com.ffalcon.mercury.android.sdk      D
    640x480
13  2025-09-25 18:13:32.491  1583-1583  camera                com.ffalcon.mercury.android.sdk      D
    640x400
14  2025-09-25 18:13:32.491  1583-1583  camera                com.ffalcon.mercury.android.sdk      D
    640x360
15  2025-09-25 18:13:32.492  1583-1583  camera                com.ffalcon.mercury.android.sdk      D
    480x360
16  2025-09-25 18:13:32.492  1583-1583  camera                com.ffalcon.mercury.android.sdk      D
    352x288
17  2025-09-25 18:13:32.492  1583-1583  camera                com.ffalcon.mercury.android.sdk      D
    320x240
18  2025-09-25 18:13:32.492  1583-1583  camera                com.ffalcon.mercury.android.sdk      D
    320x180
19  2025-09-25 18:13:32.492  1583-1583  camera                com.ffalcon.mercury.android.sdk      D
    176x144
```

8. IMU数据获取

可通过Android原生的api获取并显示加速度计、陀螺仪和磁力计的数据，详情请参考demo中的IMUActivity

对于需要设备姿态（如旋转角度）的应用，直接使用原始传感器数据非常复杂。Android 提供了软件传感器，它自动融合了多个硬件传感器的数据，结果更平滑、更准确。

游戏旋转矢量传感器 (Game Rotation Vector)

这是最适合游戏和实时姿态跟踪的传感器，它不使用磁力计，因此不受磁场干扰，但无法提供绝对的方向（如指北）。

代码块

```
// 在 onCreate 中获取传感器
private var gameRotationVectorSensor: Sensor? = null

// 在 onCreate 方法内添加:
gameRotationVectorSensor = sensorManager.getDefaultSensor(Sensor.TYPE_GAME_ROTATION_VECTOR)

// 在 onResume 中注册
gameRotationVectorSensor?.let {
    sensorManager.registerListener(this, it, SensorManager.SENSOR_DELAY_FASTEST)
}

// 在 onSensorEvent 中处理数据
Sensor.TYPE_GAME_ROTATION_VECTOR -> {
    // event.values 包含4个分量: [x*sin(θ/2), y*sin(θ/2), z*sin(θ/2), cos(θ/2)]
    // 这是一个四元数(Quaternion), 表示旋转
    val x = event.values[0]
    val y = event.values[1]
    val z = event.values[2]
    val w = event.values[3]
    // 可以将四元数转换为欧拉角（俯仰角、横滚角、偏航角）以便于理解
```

```
        // 注意：由于不使用磁力计，偏航角是相对值，会漂移。
    }
```

将四元数转换为欧拉角（简单示例）

代码块

```
private fun quaternionToEuler(x: Float, y: Float, z: Float, w: Float): FloatArray {
    val euler = FloatArray(3)

    // 俯仰角 (pitch, X轴旋转)
    val sinP = 2.0f * (w * x + y * z)
    val cosP = 1.0f - 2.0f * (x * x + y * y)
    euler[0] = Math.atan2(sinP.toDouble(), cosP.toDouble()).toFloat()

    // 横滚角 (roll, Y轴旋转)
    val sinR = 2.0f * (w * y - z * x)
    euler[1] = if (Math.abs(sinR) >= 1) {
        (Math.PI / 2).toFloat() * Math.signum(sinR)
    } else {
        Math.asin(sinR.toDouble()).toFloat()
    }

    // 偏航角 (yaw, Z轴旋转)
    val sinY = 2.0f * (w * z + x * y)
    val cosY = 1.0f - 2.0f * (y * y + z * z)
    euler[2] = Math.atan2(sinY.toDouble(), cosY.toDouble()).toFloat()

    // 将弧度转换为角度（可选）
    euler[0] = Math.toDegrees(euler[0].toDouble()).toFloat()
    euler[1] = Math.toDegrees(euler[1].toDouble()).toFloat()
    euler[2] = Math.toDegrees(euler[2].toDouble()).toFloat()

    return euler
}
```

采样率常量

在 `registerListener` 时，你可以选择不同的采样率：

- `SENSOR_DELAY_FASTEST`：最快，尽可能高的频率。
- `SENSOR_DELAY_GAME`：适合游戏的速率。
- `SENSOR_DELAY_NORMAL`：默认速率，适合屏幕方向改变。
- `SENSOR_DELAY_UI`：适合用户界面变化。

最佳实践：总是记得在 `onPause()` 中调用 `sensorManager.unregisterListener(this)` 来避免后台耗电。

9. 手机连接状态 & GPS推流

手机连接状态主要是通过MobileState类来获取，MobileState.isMobileConnected()得到手机连接状态的数据流，可以实时监听手机的连接状态，实现方式如下：

代码块

```
private fun collectBleStatus() {
    MobileState.isMobileConnected().onEach {
        FLogger.d("isMobileConnected:$it")
        mBindingPair.updateView {
```

```
            tvBleStatus.text = if (it) "connect" else "disconnect"
        }
    }.launchIn(lifecycleScope)
}
```

获取连接手机的GPS推流需要先集成IPC SDK，集成方式可参考IPCSDK for Android，具体实现方式可参考IPC SDK Sample或者上述Sample中的APIActivity，需要注意的是onResponse回调在异步线程，需要关注线程切换：

代码块

```
private val response =
    OnResponseListener { response ->
        if (response?.getData() == null) return@OnResponseListener
        try {
            val jo = JSONObject(response.getData())
            if (jo.has("mLatitude") && jo.has("mLongitude") && jo.has("mAltitude")) { //GPS数据
                val mProvider = jo.getString("mProvider")
                val mTime = jo.getLong("mTime")
                val mElapsedRealtimeNanos = jo.getLong("mElapsedRealtimeNanos")
                val mLatitude = jo.getDouble("mLatitude")
                val mLongitude = jo.getDouble("mLongitude")
                runOnUiThread {
                    mBindingPair.updateView {
                        tvTvLocationInfo.text = "$mLatitude,$mLongitude"
                    }
                }

            }
        } catch (e: JSONException) {
            e.printStackTrace()
        }
    }
```