第 3 章

拥有用户界面的 Feature Ability

在众多类型的 Ability 中，Feature Ability(FA)是最为核心的。这是因为只有 FA 才可以拥有用户界面，是实实在在可以被用户视觉所感知的。对于许多简单的用户需求来讲，只通过 FA 就可以构建出一个鸿蒙应用程序了，因此，学习 FA 是学习鸿蒙应用程序开发的第一步。

本章将在介绍 FA、Page、AbilitySlice 等基本概念的基础上，详细分析 Page、AbilitySlice 的具体使用方法，包括用户界面的构建、生命周期回调、Page 的基本选项、用户界面的跳转、工程资源的使用等。这些内容非常重要，属于鸿蒙应用程序开发中必备的基础知识。通过本章的学习，读者应该可以开发出一个具有简单用户界面的鸿蒙应用程序了，希望大家能够学有所获。

38min

3.1 Page 和 AbilitySlice

FA 只包含 Page Ability(以下简称 Page)这一类模板，因此，从目前来讲 FA 和 Page 没有什么概念上的差异，FA 就是 Page，Page 就是 FA。在本章中，统一使用 Page 来指代 FA。

本节首先介绍 AbilitySlice 的基本概念、Page 与 AbilitySlice 之间的关系及用户界面的两个必备要素：组件和布局。然后，介绍 AbilitySlice 如何承载用户界面，并通过 XML 文件和 Java 代码两种基本方式构建简单的用户界面。最后，介绍像素、虚拟像素和字体像素的相关概念，用以定义组件和文字的大小。

3.1.1 Page 的好伙伴 AbilitySlice

1. AbiltySlice 是用户界面的直接承载者

1 个 Page 可以包含 1 个或 1 组与功能相关的用户界面，其中每个独立的用户界面都被 1 个 AbilitySlice 所管理，Page 与 AbilitySlice 的关系如图 3-1 所示。虽然在 Page 中能够直接进行用户界面编程，但是一般情况下 AbilitySlice 才是用户界面的直接承载者。建议将与功能相关或相似的 AbilitySlice 由统一的 Page 对象进行管理。例如，当我们需要设计一个视频的播放功能时，可以将一个 AbilitySlice 设计为播放界面，将另一个 AbilitySlice 设计为

视频的评论界面，而这两个 AbilitySlice 均由同一个 Ability 进行管理。

注意：这里的 AbilitySlice 的概念和 Android 中的 Fragment 颇有些相似，但是，目前 1 个 Page 上只能显示 1 个 AbilitySlice，而 Android 中的 Activity 能够同时承载并显示多个 Fragment。

在通过 DevEco Studio 创建 Empty Feature Ability(Java)类型的鸿蒙应用程序工程时，默认会自动创建 1 个 Page 及与此关联的 1 个 AbilitySlice。在第 2 章所创建的 HelloWorld 工程中，自动创建的 MainAbility 类就是一个典型的 Page。除此之外，在 com. example. helloworld. slice 包中还自动创建了 1 个 MainAbilitySlice 类，而这个类就是 1 个 AbilitySlice。MainAbility 和 MainAbilitySlice 类文件的所在位置如图 3-2 所示。

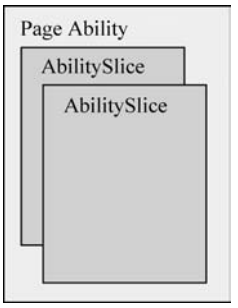

图 3-1　Feature Ability(FA)与 AbilitySlice 的关系

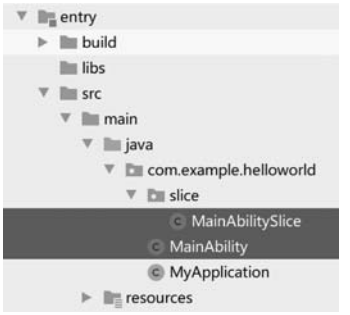

图 3-2　MainAbility 和 MainAbilitySlice 类文件所在位置

总结一下，使用 AbilitySlice 时开发者需要时刻注意以下 3 个核心要点：

(1) 虽然 Page 也能够承载用户界面，但是仍然建议开发者使用 AbilitySlice。

(2) 被同一个 Page 管理的 AbilitySlice 需要具有高度相关性。

(3) 在一个应用程序运行的过程中，同一时刻有且只能有一个 AbilitySlice 处于前台状态。

2. AbilitySlice 主路由

AbilibySlice 之间的跳转关系被称为 AbilitySlice 路由。其中，打开 Page 时默认启动的 AbilitySlice 称为该 Page 的 AbilitySlice 主路由(Main Route)。关于 Page 和 Ability 的跳转将在 3.3 节中进行详细介绍，这里仅介绍 AbilitySlice 的主路由设置方法。

大家回顾一下 MainAbility 的代码，其中只包含了一个 onStart 方法。这个 onStart 方法是每次加载 MainAbility 时所必须调用(且只调用一次)的生命周期方法。关于生命周期方法将会在 3.1.3 节中进行详细介绍，这里只需知道 onStart 方法是 MainAbility 的“入口”方法。

在该方法中，调用了父类的 setMainRoute 方法，即为 AbilitySlice 的主路由设置方法，用于定义该 Page 默认启动的 AbilitySlice，代码如下：

```
super.setMainRoute(MainAbilitySlice.class.getName());
```

这种方法需要传递一个参数，即目标 AbilitySlice 的全类名字符串（即包名＋类名字符串）。AbilitySlice 类的全类名字符串可以通过其 class 对象的 getName()方法获得。

MainAbilitySlice 的全类名字符串为 com. example. helloworld. slice. MainAbilitySlice。通过将该字符串传递到 setMainRoute 方法用于指定 MainAbility 的 AbilitySlice 主路由为 MainAbilitySlice。此时，启动 MainAbility 时就会默认启动 MainAbilitySlice。

3.1.2　初探布局和组件

构建用户界面（User Interface，UI）是 Page 的基本任务之一。优秀的 UI 是成就一个应用程序的关键因素之一。不过，再优秀的 UI 也是由一个个“零件”所组成的。这些“零件”被称为组件，包括文本、按钮等。有了这些组件还不够，还需要通过一定的规则将这些组件排列组合在一起。这样的规则被称为布局。UI 设计不过就是在“摆弄”这些组件和布局。如果说组件是优美的旋律，布局是悠扬的节奏，内容是动听的歌声，那么这三者组合在一起就能够谱写美妙的乐章了。

布局和组件是一个用户界面的必备要素，前者是骨架，后者为血肉。

（1）组件（Component）：是指具有某一特定显示、交互或布局功能的可视化物件，分为显示类组件、交互类组件和布局类组件，所有的组件都继承于基类 Component。

（2）布局（Layout）：即布局类组件，也被称为组件容器，所有的布局继承于基类 ComponentContainer。

注意：由于布局类组件属于组件，因此 ComponentContainer 也继承于 Component。

布局之间可以嵌套，即布局之中还可以包含布局，且处于最为顶层的布局称为根布局。组件不能嵌套，并且必须放入布局中才能够显示在用户界面中，不能够单独显示。典型的布局与组件关系如图 3-3 所示。

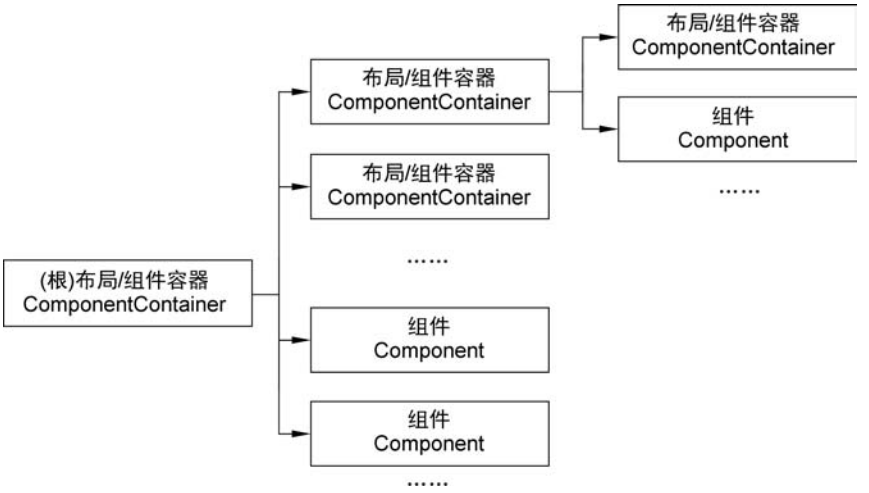

图 3-3　布局（ComponentContainer）和组件（Component）之间的关系

AbilitySlice 是用户界面的直接承载者，因此，接下来分析一下 MainAbilitySlice 类的默认创建代码，并介绍 MainAbilitySlice 是如何构建和加载用户界面的，代码如下：

```
//chapter3/LayoutXML/entry/src/main/java/com/example/layoutxml/slice/MainAblilitySlice.java
public class MainAbilitySlice extends AbilitySlice {
    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
        super.setUIContent(ResourceTable.Layout_ability_main);
    }

    @Override
    public void onActive() {
        super.onActive();
    }

    @Override
    public void onForeground(Intent intent) {
        super.onForeground(intent);
    }
}
```

MainAbilitySlice 类继承于 AbilitySlice 类，包含了 onStart、onActive 和 onForeground 方法，这 3 种方法都是生命周期方法。关于这些生命周期方法将在 3.2.1 节进行详细介绍。

在 onStart 方法中，通过 setUIContent 方法设置该 MainAbilitySlice 的 UI 界面。setUIContent 方法存在两种重载方法：

- setUIContent(int layoutRes)
- setUIContent(ComponentContainer componentContainer)

这两种重载方法对应了鸿蒙操作系统中的两种用户界面构建方法：通过 XML 文件构建用户界面和通过 Java 代码构建用户界面。通过 XML 代码可以以对象的方式声明布局和组件，可以显示布局和组件的层级结构，更加直观。通过 Java 代码可以直接在 AbilitySlice 中添加布局和组件，是最原始且运行效率最高的方法。

在 3.1.3 节和 3.1.4 节中将分别介绍用 XML 文件和 Java 代码设计用户界面的方法。

3.1.3 通过 XML 文件构建用户界面

1. 一个简单的 XML 布局文件

通过 XML 文件可以构建一个完整的用户界面，这个 XML 文件通常被称为布局文件。布局文件也属于应用资源的一类（详情可参见 3.4.1 节的相关内容），默认在 HAP 目录的 /src/main/resources/base/layout 中。例如，在 HelloWorld 工程中 MainAbilitySlice 的默认布局文件为 ability_main.xml，如图 3-4 所示。

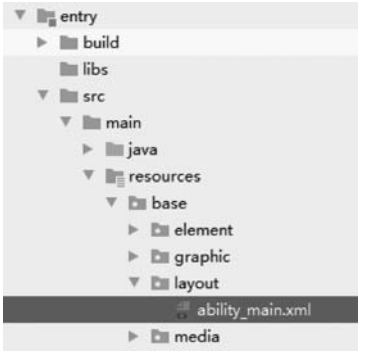

图 3-4　布局文件 ability_main. xml

接下来，让我们仔细看一看 ability_main. xml 文件，代码如下：

```
//chapter3/LayoutXML/entry/src/main/resources/base/layout/ability_main.xml
<?xml version = "1.0" encoding = "utf - 8"?>
< DirectionalLayout
    xmlns:ohos = "http://schemas.huawei.com/res/ohos"
    ohos:height = "match_parent"
    ohos:width = "match_parent"
    ohos:orientation = "vertical">

    < Text
        ohos:id = " $ + id:text_helloworld"
        ohos:height = "match_parent"
        ohos:width = "match_content"
        ohos:background_element = " $ graphic:background_ability_main"
        ohos:layout_alignment = "horizontal_center"
        ohos:text = "Hello World"
        ohos:text_size = "50"
    />

</DirectionalLayout >
```

在上述代码中，根元素< DirectionalLayout >声明了一个定向布局。定向布局是将其内部的组件沿着一个方向(横向或纵向)依次排列的一种布局方式。这个定向布局处于根元素的位置，而根元素有一个重要的任务：定义命名空间，因此，该定向布局的 xmlns:ohos 属性定义了 ohos 的命名空间 http://schemas. huawei. com/res/ohos。在各种布局和组件中，都应当使用 ohos 命令空间所定义的属性。例如，< DirectionalLayout >定向布局包含以下 3 个属性：

(1) ohos:height：组件(或布局)的高度。

(2) ohos:width：组件(或布局)的宽度。

(3) ohos:orientation：定向布局的组件排列方向：vertical 表示纵向排列；horizontal 表示横向排列。

组件(或布局)的高度和宽度属性可以通过以下几种类型进行定义,如图 3-5 所示。

(1) match_parent:由父布局或窗口对象决定组件的大小。通常,这个组件会填充整个父布局或整个窗口的大小。

(2) match_content:由组件的内容决定组件的大小。通常,这个组件会刚好包含组件中的内容。

(3) 数值+px:通过像素值(pixel,px)规定组件大小。

(4) 数值+vp:通过虚拟像素值(virtual pixel,vp)规定组件大小。关于像素与虚拟像素的概念和关系将在 3.1.5 节进行详细探讨。

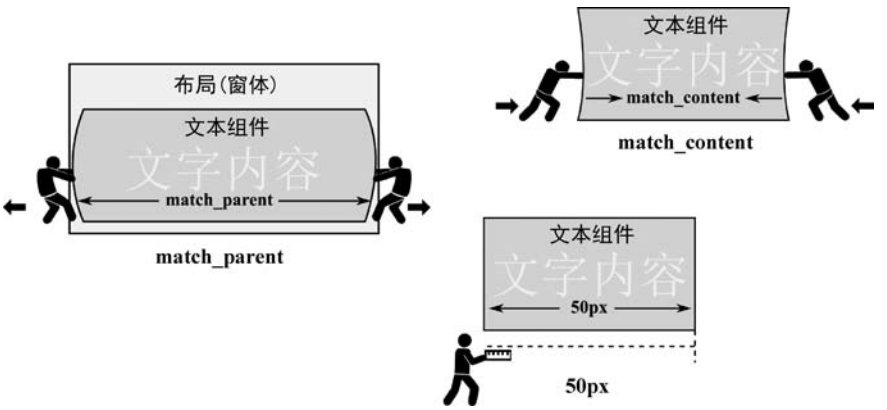

图 3-5 match_parent 与 match_content

在上面的定向布局中,由于< DirectionalLayout >是根元素,因此该布局为根布局,并且这个布局由应用程序的窗口对象管理。将定向布局的高度和宽度都设置为 match_parent 表示这个定向布局填充整个窗口对象。

注意:窗口对象由 ohos.agp.window.service.Window 类定义。每个应用程序都包括了单例的窗口对象。通常情况下,应用程序的窗口是固定且占满整个屏幕的,但是,当设备处在分屏模式或者悬浮窗模式时,窗口的大小就不是全屏大小了,甚至是可以移动的。窗口对象可以通过 Ability 或 AbilitySlice 的 getWindow()方法获取。

根元素< DirectionalLayout >包含了子元素< Text >,这说明这个定向布局包含了 1 个文本组件。在 MainAbilitySlice 中,窗口、定向布局和这个文本组件之间的关系如图 3-6 所示。

这个文本组件包含了以下属性:

- ohos:id:ID 属性,用于唯一性的标识组件。
- ohos:height:组件高度,match_parent 表示高度刚好填充整个父布局(定向布局)。
- ohos:width:组件宽度,match_content 表示宽度刚好包含文字内容。

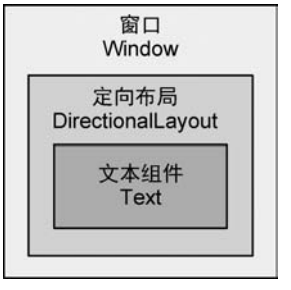

图 3-6 窗口、定向布局和文本组件之间的关系

- ohos:background_element：背景元素。
- ohos:layout_alignment：布局对齐方式，horizontal_center 表示水平居中。
- ohos:text：文本内容，设置为 Hello World。
- ohos:text_size：文本大小，设置为 50px。

注意：如果读者查看过组件的说明文档，可以发现每个组件都含有包括 AttrSet 参数的构造方法。事实上，在应用程序运行时将 XML 布局中定义的组件转换为 Java 对象，而 AttrSet 参数用于接收 XML 定义组件时的各类属性。

文本组件的 ID 属性和背景要素都引用了工程的资源。资源通过资源引用字符串进行引用。通常，资源引用字符串的格式为 $type:name，其中 type 表示资源类型，name 表示资源名称。资源类型包括 ID 资源(id)、媒体资源(media)、布局资源(layout)、可绘制资源(graphic)等，其各类资源的详细说明详见 3.4 节。例如，在上面的文本组件中，背景元素属性为 $graphic:background_ability_main，说明引用了名为 background_ability_main 的可绘制资源。

这些资源都会在 ResourceTable 类中自动生成一个静态类型常量的唯一标识符。通过这些标识符就可以在 Java 代码中获取相应的资源对象了。

唯一不同的是，ID 资源需要在资源类型前加入“+”用以在 ResourceTable 类中自动生成该 ID 资源的唯一标识符。例如，在上面的文本组件中，ID 属性为“$+id:text_helloworld”，此时就会在 ResourceTable 类中自动生成唯一标识符 Id_text_helloworld 常量。在引用这个 ID 属性时，就不需要“+”号了，使用“$id:text_helloworld”进行引用即可。

相应地，ability_main.xml 这个文件作为布局资源，也在 ResourceTable 类中生成了对应的常量 Layout_ability_main，因此，在 AbilitySlice 的 onStart 方法中，将这个常量作为参数传入 setUIContent(int layoutRes)重载方法中即可实现布局资源(也即用户界面)的加载，即

```
super.setUIContent(ResourceTable.Layout_ability_main);
```

注意：如果在编程中提示没有找到 ResourceTable 类错误，或者该类中没有生成 Layout_layout 常量错误，则可以在 Gradle 工具窗体中执行 entry→Tasks→ohos→generateDebugResources 工具，此时可以重新生成 ResourceTable 类。

2. 创建一个新的 XML 布局文件

在 DevEco Studio 的 Project 工具窗体中，定位到 HAP 目录的/src/main/resources/base 中，然后在 base 目录上右击，在弹出的菜单中选择 New→Layout Resource File 菜单，弹出创建布局资源对话框，如图 3-7 所示。

在 File name 选项中输入需要创建的布局名称 layout；在 Layout Type 中选择布局的模板类型 DirectionalLayout，即定向布局。单击 Finish 按钮，DevEco Studio 会在 layout 目录中创建一个名为 layout.xml 的布局文件，并自动生成定向布局的基础代码，

图 3-7　创建布局资源文件

代码如下：

```
//chapter3/LayoutXML/entry/src/main/resources/base/layout/layout.xml
<?xml version = "1.0" encoding = "utf - 8"?>
< DirectionalLayout xmlns:ohos = "http://schemas.huawei.com/res/ohos"
                    ohos:width = "match_parent"
                    ohos:height = "match_parent"
                    ohos:orientation = "vertical">

</DirectionalLayout >
```

此时，开发者就可以根据需求和设计方案自定义布局中的内容了。由于这个布局文件属于应用资源，因此在 ResourceTable 类中会自动生成一个名为 Layout_layout 的标识符常量。在相应的 AbilitySlice 中，通过以下代码就可以使用该布局文件了。

```
//chapter3/LayoutXML/entry/src/main/java/com/example/layoutxml/slice/MainAbilitySlice.java
@Override
public void onStart(Intent intent) {
    super.onStart(intent);
    //super.setUIContent(ResourceTable.Layout_ability_main);
    //设置布局
    super.setUIContent(ResourceTable.Layout_layout);
}
```

3. 预览 XML 布局

通过 XML 文件构建用户界面有一个好处就是可以使用预览器(Previewer)实时预览用户界面的效果。

在代码编辑窗口中打开布局文件、Page 源代码文件或 AbilitySlice 源代码文件的情况下，在 DevEco Studio 菜单栏中选择 View→Tool Windows→Previewer 菜单即可打开 Previewer 工具窗体，如图 3-8 所示。此时，这个窗体中显示了当前 Page 或当前 AbilitySlice 的预览界面。

注意：通过这种方法也可以浏览 JS UI 中的 HML 界面。

在预览器的上方，有以下几个按钮和选项：

- 刷新(Refresh)：刷新当前预览界面。
- 热刷新(ChangeHot)：当打开该选项后，进行修改代码时会实时刷新预览界面。

- 多设备预览(Multi-device preview)：当打开该选项后，可以同时显示该布局在多个设备上的预览界面。
- 缩小(Zoom Out)：缩小预览界面。
- 放大(Zoom In)：放大预览界面。

另外，当单击预览界面下方的【...】按钮后，在弹出的Debugging下拉列表框中选中Screen coordinate system即可显示屏幕坐标系，便于分析各个组件的位置和大小是否符合设计规范。

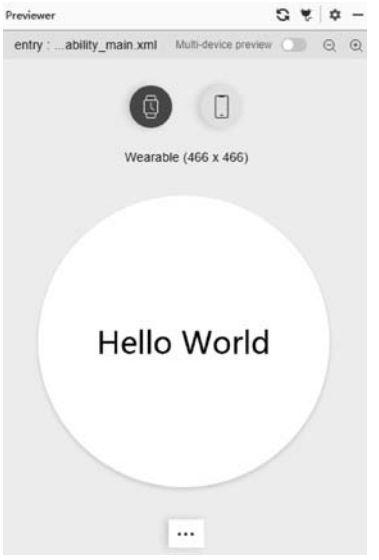

图 3-8　通过预览器(Previewer)预览XML布局文件

3.1.4　通过Java代码构建用户界面

通过Java代码构建用户界面相对枯燥一些，主要分为3个步骤：

(1) 创建布局，并设置其相关的属性。如果需要嵌套布局，可将被嵌套的子布局通过addComponent方法添加到父布局之中。

(2) 创建组件，并通过布局的addComponent方法将组件添加到布局之中。

(3) 通过setUIContent方法将UI内容设置为根布局对象。

为了能够对比XML文件和Java代码构建用户界面的区别，本节通过Java代码实现了与3.1.3节同样的用户界面，代码如下：

```
//chapter3/LayoutJava/entry/src/main/java/com/example/layoutjava/slice/MainAbilitySlice.java
public class MainAbilitySlice extends AbilitySlice {
@Override
    public void onStart(Intent intent) {
        super.onStart(intent);
        //super.setUIContent(ResourceTable.Layout_ability_main);

        //1. 创建定向布局,并设置相关的属性
        //创建定向布局的布局配置对象
        ComponentContainer.LayoutConfig configForLayout = new ComponentContainer.LayoutConfig(
                ComponentContainer.LayoutConfig.MATCH_PARENT,      //宽度为 match_parent
                ComponentContainer.LayoutConfig.MATCH_PARENT);     //高度为 match_parent
        //创建定向布局对象,并传入布局配置
        DirectionalLayout layout = new DirectionalLayout(this);
        layout.setLayoutConfig(configForLayout);           //设置定向布局的布局配置选项
        layout.setOrientation(DirectionalLayout.VERTICAL);//设置定向布局的纵向排列方式

        //2. 创建文本组件,并添加到定向布局中
```

```
        //创建文本组件的布局配置对象
        DirectionalLayout.LayoutConfig configForText = new DirectionalLayout.LayoutConfig(
                ComponentContainer.LayoutConfig.MATCH_CONTENT,   //宽度为 match_content
                ComponentContainer.LayoutConfig.MATCH_PARENT);   //高度为 match_parent
        configForText.alignment = LayoutAlignment.HORIZONTAL_CENTER;
                                                  //设置对齐方式为水平居中
        Text text = new Text(this);               //创建文本组件
        text.setLayoutConfig(configForText);      //设置文本组件的布局配置选项
        text.setBackground(new ShapeElement(
                getContext(),
                ResourceTable.Graphic_background_ability_main)); //设置文本组件的背景
        text.setText("Hello World");                  //设置文本内容为"Hello World"
        text.setTextSize(50);                         //设置文本大小为 50
        layout.addComponent(text);                    //将文本组件加入定向布局中

        //3. 将 UI 内容设置为定向布局
        super.setUIContent(layout);
    }
```

首先，创建了定向布局对象（DirectionalLayout 对象）layout。然后，定义了定向布局的配置选项，接着创建一个文本组件并加入定向布局中，最后设置 MainAbilitySlice 的 UI 内容为定向布局 layout 对象。

在这段代码中，需要开发者注意以下要点：

（1）每个布局和组件都需要设置相应的布局配置对象（LayoutConfig）。通过 LayoutConfig 可用于设置组件的高度、宽度、外边距（Margin）等与所在布局强相关的属性。值得注意的是，不同类型的布局都定义了与其相关的 LayoutConfig 子类，继承于 ComponentContainer 中的 LayoutConfig 父类。在开发时，设置某个组件的 LayoutConfig 对象时，其 LayoutConfig 类型必须与其所在布局的类型相同，开发者一定不要混淆。例如，在上例中，文本组件对象 text 所在的父布局为定向布局（DirectionalLayout），因此这个组件所使用的 LayoutConfig 对象是通过 DirectionalLayout.LayoutConfig 类所定义的。

（2）文本组件 text 的背景是通过 ShapeElement 定义的。ShapeElement 对象可以用于定义不同类型的形状元素。在上例中，通过 ResourceTable 中的 Graphic_background_ability_main 唯一标识符常量引用了相应的可绘制资源。关于可绘制资源可详见 3.4.1 节的相关内容。

（3）使用 Text 组件时要注意包名。此处应使用 ohos.agp.components 包下的 Text 类，而不是 ohos.ai.cv.text 包下的 Text 类，一定不要混淆。

3.1.3 节和 3.1.4 节介绍的这两种用户界面的构建方法都非常实用。相对来讲，XML 文件更直观具体，而 Java 代码效率更高。在运行时，XML 文件定义的各种组件会被实时地转换为 Java 对象，然后渲染到屏幕窗口中，因此 XML 文件构建用户界面的效率相对较低，但是一般来讲用户很难感知这种性能影响。在实际开发中，通常会结合这两种方法来完成

复杂的用户界面设计。

3.1.5　关于像素和虚拟像素的关系

在构建鸿蒙应用程序中，经常需要定义组件或字体的尺寸，而这些组件和字体最终会呈现在屏幕上，因此设计时需要考虑屏幕的尺寸和分辨率。开发者和设计师需要掌握一些关于像素的基本概念。

1. 像素与分辨率

无论设备屏幕显示技术采用的是 LCD(Liquid Crystal Display)还是 OLED(Organic Light-Emitting Diode)，五彩缤纷的画面都是通过能够表达颜色的发光点阵实现的，这其中的每个点都是屏幕中能够显示的最小且不可分割的单元，称为物理像素，简称像素(pixel，px)。

在之前，消费者经常通过分辨率评判一个屏幕的优劣，而分辨率就是指屏幕在横向和纵向上像素的数量。例如，华为 P40 手机的屏幕分辨率为 2340×1080。这说明，华为 P40 的屏幕在纵向上最多排列了 2340 像素，在横向上最多排列了 1080 像素，而像素总数约为 2340 和 1080 的乘积，但是，目前绝大多数的手机和平板计算机采用了异形屏幕设计(例如，屏幕的 4 个角具有弧度、挖孔屏、刘海屏、水滴屏等)，实际的像素要略少于分辨率的数值乘积。

长期以来，设计师和开发者经常以像素为单位设计组件的尺寸。例如，定义某个文本组件宽度为 300px，高度为 50px，那么可以直接在代码中指定其像素数值，代码如下：

```
<Text
    ohos:height = "48px"
    ohos:width = "300px"
    …
/>
```

如果不带单位，则默认的单位也是像素数值，代码如下：

```
<Text
    ohos:height = "48"
    ohos:width = "300"
    …
/>
```

这段代码的含义与上段代码相同。这个组件在屏幕中在横向上会占据 300 像素宽度，在纵向上会占据 48 像素高度，最终会以 300×48 像素展现这个组件。

2. 像素密度和虚拟像素

在过去的很长一段时间内，用户只要细心观察，就能够从屏幕上分辨出像素阵的存在，即存在“颗粒感”。事实上，导致屏幕颗粒感的主要因素并不是分辨率，而是像素密度。像素密度(Pixels Per Inch，PPI)是指屏幕上每英寸距离上的像素数量。

随着技术的进步，屏幕的像素密度正在不断升高，肉眼看上去也就越来越清晰。2010年苹果公司推出了视网膜屏幕(Retina Display)，PPI达到了326。这样的屏幕在一定的视距上肉眼就完全感知不到“颗粒感”了。事实上，PPI在300以上，人眼在使用移动设备时就几乎无法感知像素的存在了。

各种设备的PPI差别很大。手机屏幕的PPI非常敏感，绝大多数的手机PPI大于300，例如华为P40手机的PPI达到了480。有些手机的PPI甚至超过了500。对于可穿戴设备来讲，受限于续航能力其PPI往往较低。对于车机和智慧屏来讲，由于其观看的视距较远，用户对于PPI的敏感度较低，因此通常其也就在300左右。

注意：还存在与PPI类似的概念：DPI。DPI(Dots Per Inch)通常是针对打印机等输出设备而言的，是指每英寸距离上的墨点数量，但是，有些开发者也将DPI的概念用在屏幕上，此时PPI与DPI所表达的意义是相同的。

设备的PPI不同会导致一个问题：同样像素大小的组件显示在不同PPI的屏幕上其大小也不同。如果两块屏幕的PPI相差一倍，则显示在这两块屏幕上的组件大小也会缩放一倍，如图3-9所示。

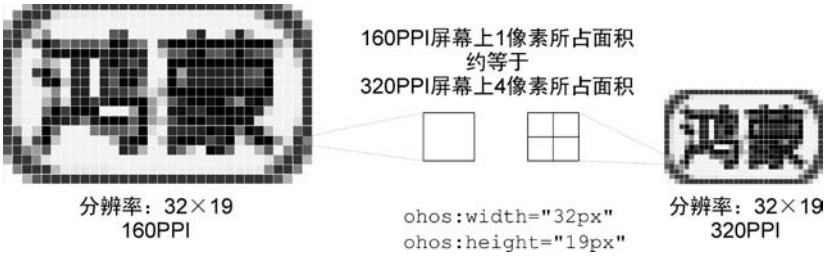

图3-9　通过像素(单位：px)定义组件大小会在不同PPI屏幕上呈现出不同的效果

这显然不是开发者和用户所希望的。为了解决这个问题，这里引出了一个新的概念：虚拟像素。

虚拟像素(virtual pixel，vp)是指与设备屏幕PPI无关的抽象像素。通过虚拟像素定义的组件，在不同PPI屏幕上显示时其实际的显示大小是相同的。那么这是怎样实现的呢？首先，定义在160PPI屏幕密度设备上，一个虚拟像素约等于一个物理像素。由此，在其他PPI设备上通过以下公式将虚拟像素的大小实时转换为物理像素的大小即可：

物理像素(px)＝虚拟像素(vp)×屏幕密度(PPI)/160

例如，30vp在华为P40(PPI为480)上的物理像素大小约为30×480/160＝90(px)，而在PPI为320的设备上的物理像素约为30×320/160＝60(px)。由此可见，以vp为单位定义的长度在高PPI设备上的实际物理像素大小要高于低PPI设备上的实际物理像素大小，并且成正比例关系。最终，以vp为单位的长度与屏幕密度无关，在不同屏幕上所显示出的物理长度是相同的。

再如，定义某个文本组件宽度为 100vp，高度为 16vp，代码如下：

```
<Text
    ohos:height = "16vp"
    ohos:width = "100vp"
    …
/>
```

这个文本组件在 160PPI 和 320PPI 的屏幕上的显示效果趋于一致，唯一不同的是后者屏幕上显示的文字会更加清晰，如图 3-10 所示。

图 3-10　通过虚拟像素(单位：vp)定义组件大小会在不同 PPI 屏幕上呈现出类似的效果

3. 字体像素

字体像素(font-size pixel，fp)的概念与虚拟像素类似，定义如下：

物理像素(px)＝字体像素(fp)×屏幕密度(PPI)/160

但是，字体像素通常应用在文本的字号上。例如，可以通过 text_size 属性定义文本的字号为 16fp，代码如下：

```
<Text
    ohos:text_size = "16fp"
    …
/>
```

这个组件中的文本内容在屏幕的纵向上占据了 16 个字体像素长度。通过上面的公式可以换算出在 480PPI 的设备(例如华为 P40 手机)上，其实际的物理像素长度约为 16×480/160＝48(px)，因此，以下代码的显示效果与上面的代码相同：

```
<Text
    ohos:text_size = "48px"
    …
/>
```

这个组件中的文本内容在屏幕的纵向上占据了 48 个像素长度。

使用字体像素还有一个优势，应用程序中的字体大小可以跟随系统显示大小。用户可以在鸿蒙操作系统的【设置】→【显示与亮度】→【字体与显示大小】→【显示大小】选项中改变应用程序中由字体像素定义的字体大小。

4. 像素和虚拟像素之间的转换

在 Java 代码中，通过布局配置（LayoutConfig）对象设置组件的宽度和高度时，所传入的数值为物理像素。如果开发者希望设置虚拟像素，那么就涉及像素转换问题了。

为了可以使用虚拟像素（字体像素）与物理像素之间的计算公式，首先需要通过 Java 代码获得当前设备的屏幕密度（PPI），代码如下：

```
int ppi = getContext().getResourceManager().getDeviceCapability().screenDensity;
```

然后，就可以实现将虚拟像素和字体像素转换为物理像素了，代码如下：

```
//chapter3/ScreenPixel/entry/src/main/java/com/example/screenpixel/slice/MainAbilitySlice.java
/**
 * 将虚拟像素(字体像素)转换为物理像素
 * @param value 虚拟像素或字体像素
 * @param context 上下文对象
 * @return 物理像素
 */
public static int toPixels(int value, Context context) {
    return value * context.getResourceManager().getDeviceCapability().screenDensity / 160;
}
```

随后，就可以应用 toPixels 方法为组件设置以虚拟像素（vp）为单位的高度和宽度，以及以字体像素（fp）为单位的字号了，代码如下：

```
//chapter3/ScreenPixel/entry/src/main/java/com/example/screenpixel/slice/MainAbilitySlice.java
Text text = new Text(this);
DirectionalLayout.LayoutConfig configForText = new DirectionalLayout.LayoutConfig(
        toPixels(100, getContext()),                //设置宽度为 100vp
        toPixels(16, getContext()));                //设置高度为 16vp
text.setLayoutConfig(configForText);                //设置文本组件的布局配置
text.setTextSize(toPixels(16, getContext()));       //设置文本大小为 16fp
```

5. 获取设备屏幕的宽度和高度

有时，为了能够更加精准地布局组件，需要获取设备屏幕的宽度和高度，代码如下：

```
//chapter3/ScreenPixel/entry/src/main/java/com/example/screenpixel/MainAbility.java
HiLog.info(loglabel, "设备宽度：" + getResourceManager().getDeviceCapability().width);
HiLog.info(loglabel, "设备高度：" + getResourceManager().getDeviceCapability().height);
```

需要注意的是，通过这种方法获取的设备屏幕的宽度和高度单位为虚拟像素单位。

3.2 Page 的生命周期和配置选项

45min

本节介绍 Page 和 AbilitySlice 的生命周期及 Page 的一些常用的属性设置，并详细介绍当设备屏幕改变时开发者所需要注意的问题。

3.2.1　Page 与 AbilitySlice 的生命周期

1. 什么是生命周期

任何事物都有其产生、发展和灭亡的过程。Page 和 AbilitySlice 也不例外。以 Page 为例，用户可以启动一个 Page，也可以关闭一个 Page。被打开的 Page 可能会被另一个 Page 全部或部分遮挡，此时被遮挡的 Page 就不能响应 UI 事件。当用户进入桌面时，被打开的应用程序的 Page 会进入后台。总之，一个 Page 被启动后可能会被用户各种“折腾”，并最终被关闭。从一个 Page(AbilitySlice)启动到关闭的全部过程就是一个 Page(AbilitySlice)的生命周期(Lifecycle)。由于 Page 和 AbilitySlice 都具有承载用户界面的功能，并且其生命周期的方法非常类似，因此下文一并进行介绍。

Page(AbilitySlice)包括 4 种生命周期状态：

(1) 初始态(INITAL)：当 Page(AbilitySlice)还没有被启动时，以及 Page 被关闭后就会处于初始态。

(2) 非活跃态(INACTIVE)是指 Page(AbilitySlice)已经启动，但是此时可能因为被对话框遮挡一部分界面等情况，无法进行用户交互，此时为非活跃态。

(3) 活跃态(ACTIVE)是指在 Page(AbilitySlice)处于界面的最前台，正在与用户进行交互，此时为活跃态。

(4) 后台态(BACKGROUND)是指 Page(AbilitySlice)完全不可见的状态。此时，可能被其他的 Page(AbilitySlice)完全遮挡，或者应用程序已经进入后台(用户按下 Home 键进入桌面或者正在熄屏)。

当生命周期状态被切换时，系统会回调到生命周期方法中以便处理一些必要的事务。例如，当 AbiltySlice 从初始态切换到非活跃态时，需要进行 UI 界面的初始化；当 AbilitySlice 进入后台态时，如果此时正在播放视频，可能需要将视频暂停。准确地应用生命周期方法进行界面和业务逻辑的控制有助于提高应用程序的设计感、稳健性和流畅性。

Page(AbilitySlice)的生命周期方法如下：

(1) onStart(Intent intent)：当 Page(AbilitySlice)从初始态进入非活跃态时，即启动 Page(AbilitySlice)时触发，在整个生命周期中仅被触发 1 次。

(2) onActive()：当 Page(AbilitySlice)从非活跃态进入活跃态时触发。

(3) onInActive()：当 Page(AbilitySlice)从活跃态进入非活跃态时触发。

(4) onBackground()：当 Page(AbilitySlice)从非活跃态进入后台态，即完全不可见时触发。

(5) onForeground(Intent intent)：当 Page(AbilitySlice)从后台态进入非活跃态，即重新可见时触发。

(6) onStop()：当 Page(AbilitySlice)从后台态进入初始态时，即结束 Page(AbilitySlice)时触发，在整个生命周期中仅被触发 1 次。

Page(AbilitySlice)的整个生命周期状态及状态切换时所调用的生命周期方法如图 3-11 所示。

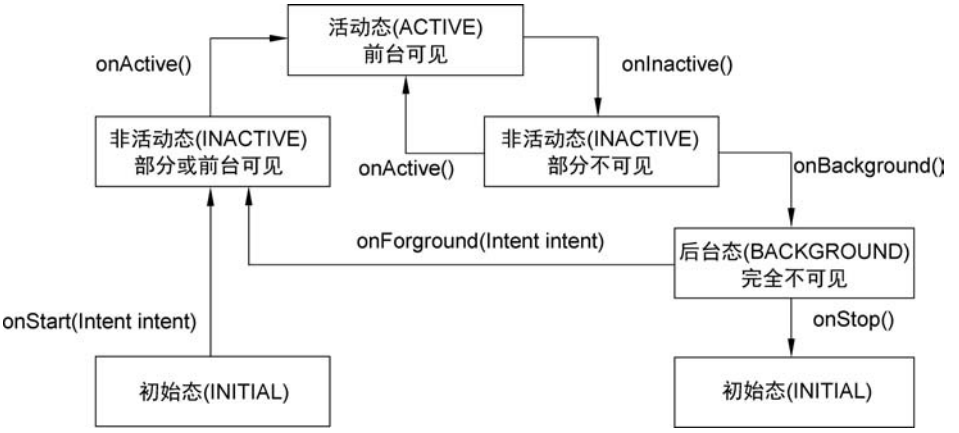

图 3-11　Page(AbilitySlice)的生命周期

使用生命周期方法时需要注意以下几个方面：

(1) 在一个 Page(AbilitySlice)的整个生命周期中，一定会回调 onStart、onActive、onInactive、onBackgroud 和 onStop 方法，而只有 onForeground 方法并不一定被回调。

(2) 在 Page(AbilitySlice)处在后台态时(即完全不可见时)，并且出现了内存不足等情况，Page(AbilitySlice)可能会被系统直接回收。

(3) 开发者需要把握各个业务逻辑的正确时机。例如，在 onStart 方法中需要进行 UI 界面的初始化；在 onStop 方法中需要检查并关闭所有由本 Page(AbilitySlice)打开的数据库连接等。一些常用业务逻辑的调用时机会在今后的学习中逐步介绍，当然对于特殊的业务逻辑则需要开发者自行设计。

接下来，我们通过实例深入体验一下 Page 和 AbilitySlice 的生命周期。

2. 深入体验 Page 与 AbilitySlice 的生命周期

首先，创建一个新的名为 Lifecycle 的应用程序，专门对生命周期方法进行学习及调试使用。与第 2 章所创建的 HelloWorld 类似，这个 Lifecycle 工程选择目标设备仍然为 Wearable，并使用 Empty Feature Ability(Java)模板。

然后，修改 MainAbility 类，实现所有的 6 个生命周期方法，并在每个生命周期方法被调用时打印其生命周期方法的名称，代码如下：

```
//chapter3/Lifecycle/entry/src/main/java/com/example/lifecycle/MainAbility.java
public class MainAbility extends Ability {
    static final HiLogLabel loglabel = new HiLogLabel(HiLog.LOG_APP, 0x00101, "MainAbility");
    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
        super.setMainRoute(MainAbilitySlice.class.getName());
        HiLog.info(loglabel, "onStart");
    }
```

```
    @Override
    protected void onActive() {
        super.onActive();
        HiLog.info(loglabel, "onActive");
    }

    @Override
    protected void onForeground(Intent intent) {
        super.onForeground(intent);
        HiLog.info(loglabel, "onForeground");
    }

    @Override
    protected void onBackground() {
        super.onBackground();
        HiLog.info(loglabel, "onBackground");
    }

    @Override
    protected void onInactive() {
        super.onInactive();
        HiLog.info(loglabel, "onInactive");
    }

    @Override
    protected void onStop() {
        super.onStop();
        HiLog.info(loglabel, "onStop");
    }
}
```

与 MainAbility 类似，实现 MainAbilitySlice 的 6 个生命周期方法，并在调用时打印其生命周期方法的名称，代码如下：

```
//chapter3/Lifecycle/entry/src/main/java/com/example/lifecycle/slice/MainAbilitySlice.java
public class MainAbilitySlice extends AbilitySlice {
    static final HiLogLabel loglabel = new HiLogLabel(HiLog.LOG_APP, 0x00101,
"MainAbilitySlice");
    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
        super.setUIContent(ResourceTable.Layout_ability_main);
        HiLog.info(loglabel, "onStart");
    }
```

```
    @Override
    public void onActive() {
        super.onActive();
        HiLog.info(loglabel, "onActive");
    }

    @Override
    public void onForeground(Intent intent) {
        super.onForeground(intent);
        HiLog.info(loglabel, "onForeground");
    }

    @Override
    protected void onBackground() {
        super.onBackground();
        HiLog.info(loglabel, "onBackground");
    }

    @Override
    protected void onInactive() {
        super.onInactive();
        HiLog.info(loglabel, "onInactive");
    }

    @Override
    protected void onStop() {
        super.onStop();
        HiLog.info(loglabel, "onStop");
    }
}
```

注意，为了区分打印输出的来源，在 MainAbility 类和 MainAbilitySlice 类中，HiLogLabel 的 tag 参数不同，前者为 MainAbility，而后者为 MainAbilitySlice。

编译并在虚拟机中运行 Lifecycle 应用程序，设备出现 MainAbilitySlice 界面。在这个过程中，HiLog 工具窗体依次显示以下提示（此处略去了提示时间，下同）：

```
20852 - 20852/com.example.lifecycle I 00101/MainAbility: onStart
20852 - 20852/com.example.lifecycle I 00101/MainAbilitySlice: onStart
20852 - 20852/com.example.lifecycle I 00101/MainAbility: onActive
20852 - 20852/com.example.lifecycle I 00101/MainAbilitySlice: onActive
```

这说明在进入 MainAbilitySlice 界面的过程中，MainAbility 和 MainAbilitySlice 从初始态进入了非活动态，紧接着又从非活动态进入了活动态。并且，在每次的状态变化时，MainAbility 都要先于 MainAbilitySlice 一步。

接下来，在虚拟机中单击○（Home）按钮进入桌面，同时应用程序进入后台，MainAbility

和 MainAbilitySlice 不可见。在这个过程中，HiLog 工具窗体依次显示以下提示：

```
20852 - 20852/com.example.lifecycle I 00101/MainAbility: onInactive
20852 - 20852/com.example.lifecycle I 00101/MainAbilitySlice: onInactive
20852 - 20852/com.example.lifecycle I 00101/MainAbility: onBackground
20852 - 20852/com.example.lifecycle I 00101/MainAbilitySlice: onBackground
```

这说明，MainAbility 和 MainAbilitySlice 从活动态进入了非活动态，紧接着又从非活动态进入了后台态。

然后，再次返回到该应用程序，HiLog 工具窗体依次显示以下提示：

```
20852 - 20852/com.example.lifecycle I 00101/MainAbility: onForeground
20852 - 20852/com.example.lifecycle I 00101/MainAbilitySlice: onForeground
20852 - 20852/com.example.lifecycle I 00101/MainAbility: onActive
20852 - 20852/com.example.lifecycle I 00101/MainAbilitySlice: onActive
```

这说明，MainAbility 和 MainAbilitySlice 从后台态进入了非活动态，紧接着又从非活动态进入了活动态。

如果此时单击模拟器的◁（Back）按钮退出应用程序，HiLog 工具窗体依次显示以下提示：

```
20852 - 20852/com.example.lifecycle I 00101/MainAbility: onInactive
20852 - 20852/com.example.lifecycle I 00101/MainAbilitySlice: onInactive
20852 - 20852/com.example.lifecycle I 00101/MainAbility: onBackground
20852 - 20852/com.example.lifecycle I 00101/MainAbilitySlice: onBackground
20852 - 20852/com.example.lifecycle I 00101/MainAbility: onStop
20852 - 20852/com.example.lifecycle I 00101/MainAbilitySlice: onStop
```

这说明，MainAbility 和 MainAbilitySlice 从活动态进入了非活动态，紧接着又从非活动态进入了后台态，最后又从后台态进入了初始态。

这些调用过程非常重要，开发者一定要时刻注意其状态的变化。接下来，总结一下常见的生命周期状态的变化过程，如表 3-1 所示。

表 3-1　常见的生命周期状态变化过程

常见操作	状态变化	回调方法
进入 Page(AbilitySlice)	初始态→非活动态→活动态	onStart→onActive
返回到桌面或熄屏，应用程序进入后台	活动态→非活动态→后台态	onInactive→onBackground
应用程序在后台时，重新进入前台	后台态→非活动态→活动态	onForeground→onActive
退出应用程序，或仅退出 Page(AbilitySlice)	活动态→非活动态→后台态→初始态	onInactive→onBackground→onStop

另外，因为 Page 是 AbilitySlice 的载体，所以 Page 的生命周期总是先于 AbilitySlice 一步。

3. 同一 Page 内部的 AbilitySlice 在跳转时的生命周期变化

上面演示的仅为单个的 Page，而且 Page 中仅有单个的 AbilitySlice 时的生命周期变化情况。事实上，不仅 Page 之间可以跳转，同一个 Page 内的 AbilitySlice 也可以跳转。当同一个 Page 内的 AbilitySlice 跳转时，Page 的状态一直处于活动态，而 AbilitySlice 的生命周期却在发生变化。

例如，某一个 Page 中存在两个 AbilitySlice，分别为 SliceA 和 SliceB。当从 SliceA 跳转到 SliceB 时，两者的生命周期状态变化过程如下：SliceA 从活动态转换为非活动态→SliceB 从初始态转换为非活动态→SliceB 从非活动态转换为活动态→SliceA 从非活动态转换为后台态。生命周期的调用顺序如下：SliceA. onInactive()→SliceB. onStart()→SliceB. onActive()→SliceA. onBackground()。

4. 知晓当前的生命周期状态

在 Page 或 AbilitySlice 中，通过 getLifecycle(). getLifecycleState()方法即可获得当前的生命周期状态。生命周期状态通过 Lifecycle. Event 枚举类型定义，其所有枚举值包括 UNDEFINED、ON_START、ON_INACTIVE、ON_ACTIVE、ON_BACKGROUND、ON_FOREGROUND、ON_STOP。

例如，在 Page 或 AbilitySlice 中通过这种方法即可打印出当前的生命周期状态，代码如下：

```
HiLog.info(logLabel, "生命周期:" + getLifecycle().getLifecycleState().name());
```

3.2.2 Page 常用配置选项

本节介绍 Page 类型的 Ability 的常用配置选项，以及如何在程序中获得这些配置信息。在 config. json 中，module 对象的 abilities 数据包含了各个 Ability 的配置选项。对于新创建的鸿蒙应用程序工程，典型的 abilities 配置信息如下：

```
//chapter3/Lifecycle/entry/src/main/config.json
"abilities": [
{
    "skills": [
      {
        "entities": [
          "entity.system.home"
        ],
        "actions": [
          "action.system.home"
        ]
      }
    ],
```

```
        "orientation": "landscape",
        "formEnabled": false,
        "name": "com.example.lifecycle.MainAbility",
        "icon": "$media:icon",
        "description": "$string:mainability_description",
        "label": "PageNavigation",
        "type": "page",
        "launchType": "standard"
      }]
```

这个数组仅包含了1个Ability对象，为Page类型。Ability的类型通过type属性定义，包括page、service和data，分别代表Ability的三类模板Page Ability、Service Ability和Data Ability。在上面的Ability中，type属性为page，因此属于Page类型的Ability。

下面分析一下这个Ability对象中各个属性的含义：

(1) name：Ability的名称，通常采用全类名(包名+类名)的方式定义。

(2) description：Ability的描述信息。

(3) icon：Ability的图标。

(4) label：Ability的显示名称，默认会显示在手机、车机等设备应用程序的标题栏中。

(5) formEnabled：是否支持卡片能力。支持卡片的Page可以微缩化显示在其他应用中，例如显示在桌面上。

(6) orientation：Page的屏幕方向，包括unspecified(未指定，由系统决定)、landscape(横向显示)、portrait(纵向显示)和followRecent(跟随最近使用的Ability一致)等选项。如果指定屏幕方向为横向显示或纵向显示，则在运行时，Ability无法随着设备的物理旋转而自动改变屏幕方向。对于可穿戴设备、智慧屏、车机来讲，默认的Page屏幕方向为横向显示。

(7) launchType：Page的启动模式，包括standard(标准模式)和singleton(单例模式)两类。

(8) skills数组：表示能够接收Intent的请求。这里有一个默认的skill对象“{"entities":["entity.system.home"],"actions":["action.system.home"]}”，表示该HAP的入口Ability。

(9) configChanges：表示Ability所关注的系统配置集合。当指定的系统配置发生变化后，则会调用Ability的onConfigurationUpdated回调，方便开发者进行处理。支持的系统配置包括语言区域配置(locate)、屏幕布局配置(layout)、字体显示大小配置(fontSize)、屏幕方向配置(orientation)、显示密度配置(density)。

注意：应用程序的图标和标题是通过Entry HAP的入口Page的图标(icon)和标题(label)进行定义的。如果存在多个入口Page，则以abilities数组中第一个出现的入口Page为准。另外，应用程序的图标可以被鸿蒙操作系统自动圆角化，不需要开发者主动制作圆角化图标。

以上仅介绍了涉及Page且最为常用的属性。更多的更加全面的属性配置读者可详见

官方文档。

在 Ability(及 AbilitySlice)内部可通过 AbilityInfo 对象获取上述绝大多数信息,代码如下:

```
HiLog.info(loglabel, "描述 : " + getAbilityInfo().getDescription());
HiLog.info(loglabel, "显示名称 : " + getAbilityInfo().getLabel());
HiLog.info(loglabel, "图标路径 : " + getAbilityInfo().getIconPath());
HiLog.info(loglabel, "启动模式 : " + (getAbilityInfo().getLaunchMode() == LaunchMode.
SINGLETON ? "单例模式" : "普通模式"));
```

此时,在 HiLog 工具窗体中可输出以下信息:

```
4264 - 4264/? I 00101/MainAbility: 描述 : MainAbility
4264 - 4264/? I 00101/MainAbility: 显示名称 : Lifecycle
4264 - 4264/? I 00101/MainAbility: 图标路径 : $media:icon
4264 - 4264/? I 00101/MainAbility: 启动模式 : 普通模式
```

3.2.3 屏幕方向与设备配置改变

对于可移动设备(手机、平板计算机等)来讲,用户可以根据实际的应用场景改变设备屏幕的方向。例如,当用户准备用手机看电影时,查找、浏览电影的信息通常使用纵向的屏幕方向,而观看电影时通常使用横向的屏幕方向。这时就需要通过代码控制屏幕的方向了。

1. 通过代码改变屏幕方向

通过 setDisplayOrientation(DisplayOrientation orientaion)方法即可改变屏幕的方向,代码如下:

```
//chapter3/DisplayOrientation/entry/src/main/java/com/example/displayorientation/slice/
MainAbilitySlice.java
//将屏幕方向强制改变为横向
setDisplayOrientation(AbilityInfo.DisplayOrientation.LANDSCAPE);
//将屏幕方向强制改变为纵向
setDisplayOrientation(AbilityInfo.DisplayOrientation.PORTRAIT);
```

2. 固定屏幕方向

永久性地固定 Page 的屏幕方向非常简单,只需要在 config.json 中配置 Page 的 orientation 属性为 landscape(横向显示)或 portrait(纵向显示),但是,很多情况下,一个 Page 并不是在所有的情况下都需要保持一个方向。例如,播放视频时在未锁定屏幕的情况下可以改变屏幕方向,在锁定屏幕的情况下固定屏幕方向,那么在 config.json 中配置固定屏幕方向就不合适了。这种需求可以采用复写 setDisplayOrientation(DisplayOrientation orientaion)方法实现,代码如下:

```
//chapter3/DisplayOrientation/entry/src/main/java/com/example/displayorientation/slice/
//MainAbilitySlice.java
//是否固定屏幕方向
private boolean isOrientationFixed = true;

//复写 setDisplayOrientation 方法
@Override
public void setDisplayOrientation(DisplayOrientation requestedOrientation) {
    //当 isOrientationFixed 为 true 时保持纵向(或横向)屏幕方向
    if (isOrientationFixed) {
        super.setDisplayOrientation(DisplayOrientation.PORTRAIT);
        return;
    }
    super.setDisplayOrientation(requestedOrientation);
}
```

当 isOrientationFixed 变量为 false 时，屏幕方向可以随意改变；但是当 isOrientationFixed 变量为 true 时，屏幕方向将被固定为纵向。

3. 屏幕方向变化时的设备配置改变

屏幕方向变化时会引起设备配置的改变(Device Config Change)，而设备配置的改变会引起 Page 的重建。设备配置包括屏幕密度、屏幕方向、屏幕尺寸、语言区域、字体显示大小等。显然，屏幕密度和屏幕尺寸在运行时几乎不会被改变，但是屏幕方向、语言区域等设备配置在运行时是可以被改变的。

设备配置改变后，当前 Page 就无法适应新的设备配置了，因此，在默认情况下应用程序可以将 Page 销毁并重建。这种方法显然最为简单，但是问题也最大：当前 Page 展示的数据和状态信息也通通被销毁了。

接下来用实例向大家解释一下。

在 3.2.1 节中的 Lifecycle 应用程序中，修改 config.json 中 module 对象下的 deviceType，加入 phone 类型，使其支持手机设备，便于测试屏幕方向变化，代码如下：

```
//chapter3/Lifecycle/entry/src/main/config.json
"module": {
  "package": "com.example.test1",
  "name": ".MyApplication",
  "deviceType": [
    "wearable", "phone"
  ],
  …
}
```

在手机设备上运行应用程序，然后改变屏幕方向，此时在 HiLog 工具窗体中提示以下信息：

```
20852 - 20852/com.example.lifecycle I 00101/MainAbility: onInactive
20852 - 20852/com.example.lifecycle I 00101/MainAbilitySlice: onInactive
20852 - 20852/com.example.lifecycle I 00101/MainAbility: onBackground
20852 - 20852/com.example.lifecycle I 00101/MainAbilitySlice: onBackground
20852 - 20852/com.example.lifecycle I 00101/MainAbility: onStop
20852 - 20852/com.example.lifecycle I 00101/MainAbilitySlice: onStop
20852 - 20852/com.example.lifecycle I 00101/MainAbility: onStart
20852 - 20852/com.example.lifecycle I 00101/MainAbilitySlice: onStart
20852 - 20852/com.example.lifecycle I 00101/MainAbility: onActive
20852 - 20852/com.example.lifecycle I 00101/MainAbilitySlice: onActive
```

可以发现,在屏幕方向改变的过程中,MainAbility 被销毁后重建了。

那么,如果开发者希望在屏幕方向改变后保留当前的 Page 数据和状态该怎么办呢? 有两种方法:

- 不销毁重建 Page。
- 销毁 Page 时保留临时数据,重建 Page 时读取临时数据。

接下来分别介绍这两种方法的实现方式:

1) 不销毁重建 Page

这种方法其实很简单,只需要在 config.json 中在当前的 Ability 的配置选项加入 configChanges 属性,并在其数组中加入 orientation,代码如下:

```
{
  "orientation": "unspecified",
  "name": "com.example.lifecycle.MainAbility"
  "configChanges": ["orientation"],
  …
}
```

重新运行程序,旋转设备并观察 HiLog 工具窗体,可以看出 MainAbility 不会被销毁后重建了。

2) 销毁 Page 时保留临时数据,重建 Page 时读取临时数据

在 Ability 中,通过重写 onStoreDataWhenConfigChange()方法存储临时数据,代码如下:

```
//chapter3/Lifecycle/entry/src/main/java/com/example/lifecycle/MainAbility.java
@Override
public Object onStoreDataWhenConfigChange() {
    return "需要存储的数据,转换为 Object 对象";
}
```

在 Ability 或 AbilitySlice 中,通过 getLastStoredDataWhenConfigChanged()方法读取临时数据,代码如下:

```
//chapter3/Lifecycle/entry/src/main/java/com/example/lifecycle/MainAbility.java
if (getLastStoredDataWhenConfigChanged() != null) {
    //获取存储的数据对象
    String data = getLastStoredDataWhenConfigChanged().toString();
    HiLog.info(loglabel, data);
}
```

3.3　用户界面的跳转

50min

在绝大多数的应用程序中，存在着许多不同功能的用户界面。例如，在社交应用中，包括了好友列表界面、聊天界面、个人资料界面等。在电商应用中，包括了商品列表、商品详情、购物车、订单浏览等界面。一般来讲，一个界面完成一项特定的功能即可，而用户会在不同的界面中不断跳转，去完成各种各样的操作。在鸿蒙应用程序中，一个 Page 中的多个 AbilitySlice 是具有功能相关性的一系列界面，而不同 Page 往往实现的是独立的功能界面，因此，用户界面跳转包含了 AbilitySlice 之间的跳转（即 AbilitySlice 路由），以及 Page 之间的跳转。

用户界面的跳转涉及数据的传递。例如，在商品列表中选择商品后弹出商品详情界面，那么商品详情界面的首要任务就是要知道用户选择的是哪个商品。这样在弹出商品详情界面时，就需要商品列表界面将商品的信息传递给商品详情界面。

本节介绍 Page 之间和 AbilitySlice 之间的跳转方法和数据传递方法。

3.3.1　AbilitySlice 的跳转

在介绍 AbilitySlice 的跳转方法之前，需要先创建一个名为 AbilitySliceNavigation 的新工程，选择目标设备为 Wearable，并使用 Empty Feature Ability(Java)模板。接下来，将介绍如何创建一个新的名为 SecondAbilitySlice 的 AbilitySlice，并实现 MainAbilitySlice 与 SecondAbilitySlice 之间的跳转。

1. 创建 SecondAbilitySlice

首先，在 Project 工具窗体中，在 slice 包上右击，选择 New→Java Class 菜单，弹出如图 3-12 所示的对话框。

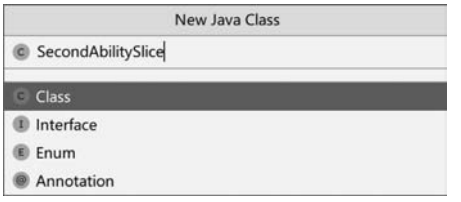

图 3-12　新建 SecondAbilitySlice 类

输入类名为 SecondAbilitySlice 后，按下回车键即可创建一个名为 SecondAbilitySlice 的 Java 类。

此时，SecondAbilitySlice 还没有继承 AbilitySlice 父类。打开 SecondAbilitySlice 类，在类名后添加 extends AbilitySlice 代码，使 SecondAbilitySlice 继承于 AbilitySlice。

```
public class SecondAbilitySlice extends AbilitySlice {
}
```

然后，复写 AbilitySlice 的生命周期方法 onStart。读者可以通过代码提示的方法加入 onStart 复写方法。当输入 onStart 的前面几个字符后，就会出现相应的代码提示，如图 3-13 所示。

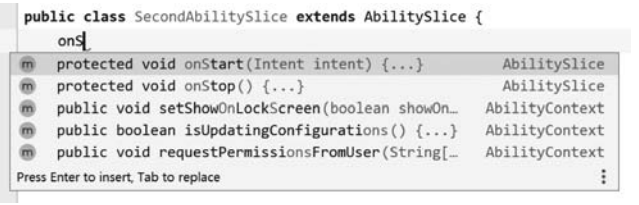

图 3-13　通过代码提示的方法复写 onStart 方法

此时，单击代码提示中的 onStart 方法即可自动生成 onStart 复写方法，代码如下：

```
//chapter3/AbilitySliceNavigation/entry/src/main/java/com/example/abilityslicenavigation/
slice/SecondAbilitySlice.java
public class SecondAbilitySlice extends AbilitySlice {
    @Override
    protected void onStart(Intent intent) {
        super.onStart(intent);
    }
}
```

当然，也可以选择直接手动键入上述所有的代码。这样，该工程中就有了两个 AbilitySlice：MainAbilitySlice 和 SecondAbilitySlice。

2. 修改 MainAbilitySlice 和 SecondAbilitySlice 的用户界面

为了实验方便，在 MainAbilitySlice 显示一个内容为 MainAbilitySlice 的文本视图，在 SecondAbilitySlice 显示一个内容为 SecondAbilitySlice 的文本视图。

首先，修改 MainAbilitySlice 类，让其显示一个内容为 MainAbilitySlice 的文本视图，代码如下：

```
//chapter3/AbilitySliceNavigation/entry/src/main/java/com/example/abilityslicenavigation/
//slice/MainAbilitySlice.java
public class MainAbilitySlice extends AbilitySlice {
    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
        //super.setUIContent(ResourceTable.Layout_ability_main);
        //创建布局配置对象
         LayoutConfig config = new LayoutConfig(ComponentContainer.LayoutConfig.MATCH_
PARENT, ComponentContainer.LayoutConfig.MATCH_PARENT);
        //创建定向布局对象,并传入布局配置
        DirectionalLayout layout = new DirectionalLayout(this);
```

```
        //将定向布局的背景颜色设置为白色
        ShapeElement element = new ShapeElement();          //创建形状元素 element 对象
        element.setRgbColor(new RgbColor(255, 255, 255));//将 element 颜色设置为白色
        layout.setBackground(element);                      //将背景设置为 element
        layout.setLayoutConfig(config);                     //设置定向布局的布局配置选项
        //创建文本组件对象,并传入布局配置,设置文本内容
        Text text = new Text(this);                         //创建文本组件
        text.setLayoutConfig(config);                       //设置文本组件的布局配置选项
        text.setTextAlignment(TextAlignment.CENTER);        //将文本对齐方式设置为居中
        text.setText("MainAbilitySlice");          //将文本内容设置为"MainAbilitySlice"
        text.setTextSize(50);                       //将文本大小设置为 50
        //将文本组件加入定向布局中
        layout.addComponent(text);
        //将 UI 内容设置为定向布局
        super.setUIContent(layout);
    }
}
```

类似地,修改 SecondAbilitySlice 文件,显示一个内容为 SecondAbilitySlice 的文本视图,代码如下:

```
//chapter3/AbilitySliceNavigation/entry/src/main/java/com/example/abilityslicenavigation/
//slice/SecondAbilitySlice.java
public class SecondAbilitySlice extends AbilitySlice {
    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
        //创建布局配置对象
         LayoutConfig config = new LayoutConfig(ComponentContainer.LayoutConfig.MATCH_
PARENT, ComponentContainer.LayoutConfig.MATCH_PARENT);
        //创建定向布局对象,并传入布局配置
        DirectionalLayout layout = new DirectionalLayout(this);
        //将定向布局的背景颜色设置为白色
        ShapeElement element = new ShapeElement();          //创建形状元素 element 对象
        element.setRgbColor(new RgbColor(100, 100, 255));  //将 element 颜色设置为浅蓝色
        layout.setBackground(element);                  //将背景设置为 element
        layout.setLayoutConfig(config);                 //设置定向布局的布局配置选项
        //创建文本组件对象,并传入布局配置,设置文本内容
        Text text = new Text(this);                     //创建文本组件
        text.setLayoutConfig(config);                   //设置文本组件的布局配置选项
        text.setTextAlignment(TextAlignment.CENTER);   //将文本对齐方式设置为居中
        text.setText("SecondAbilitySlice");     //设置文本内容为"SecondAbilitySlice"
        text.setTextSize(50);                   //设置文本大小为 50
        //将文本组件加入定向布局中
        layout.addComponent(text);
        //将 UI 内容设置为定向布局
        super.setUIContent(layout);
    }
}
```

MainAbilitySlice 与 SecondAbilitySlice 的界面仅有以下不同：

(1) 文本内容不同，前者显示文本为 MainAbilitySlice，后者显示为 SecondAbilitySlice。

(2) 布局背景不同，前者背景为白色，后者背景为蓝色。

在 MainAbility 中，找到该 Page 的默认 AbilitySlice 主路由配置代码，默认代码如下：

```
super.setMainRoute(MainAbilitySlice.class.getName());
```

此时，编译并运行程序，会默认显示 MainAbilitySlice 的内容，如图 3-14 所示。将该主路由配置代码进行修改，修改后代码如下：

```
super.setMainRoute(SecondAbilitySlice.class.getName());
```

此时编译并运行程序，就可以在应用程序中默认显示刚创建的 SecondAbilitySlice 的界面内容了，如图 3-15 所示。

图 3-14　MainAbilitySlice 的用户界面

图 3-15　SecondAbilitySlice 的用户界面

主路由的设置非常简单。接下来，还是把 MainAbility 中默认主路由改回 MainAbilitySlice，重点介绍 AbilitySlice 的跳转方法。

3. AbilitySlice 的跳转

在同一个 Page 中 AbilitySlice 的跳转非常简单，只需通过 present 方法传入需要跳转的 AbilitySlice 对象和一个空的 Intent 对象。例如，从 MainAbilitySlice 跳转到 SecondAbilitySlice 可通过以下代码实现：

```
present(new SecondAbilitySlice(), new Intent());
```

其中，第 1 个参数为即将跳转的 AbilitySlice 对象；第 2 个参数为空的 Intent 对象。

首先，在 MainAbilitySlice 的 onStart 方法的最后加入以下代码：

```
text.setClickedListener(new Component.ClickedListener() {
    @Override
    public void onClick(Component component) {
        present(new SecondAbilitySlice(), new Intent());
    }
});
```

这段代码为 text 文本组件添加了单击事件的回调。通过 setClickedListener 方法即可设置该回调。该回调需要传入 ClickedListener 回调接口，并实现其 onClick 回调方法。此时，当用户单击 text 文本后，即可在 onClick 方法中处理这一事件。此处，通过 present 方法跳转到 SecondAbilitySlice。

在 SecondAbilitySlice 的 onStart 方法的最后加入以下代码：

```
text.setClickedListener(new Component.ClickedListener() {
    @Override
    public void onClick(Component component) {
        present(new MainAbilitySlice(), new Intent());
    }
});
```

这段代码与 MainAbilitySlice 中添加的代码类似，为 text 文本组件添加了单击事件：单击后，通过 present 方法跳转到 SecondAbilitySlice。

此时，编译并运行 AbilitySliceRoute 程序，即可实现单击屏幕任意位置实现两个 AbilitySlice 的相互跳转，如图 3-16 所示。

图 3-16　AbilitySlice 的跳转

在上例中实现了两个 AbilitySlice 的跳转。由于在跳转过程中，这两个 AbilitySlice 之间是平级的，因此这种路由模式被称为平级路由，但是每次跳转都会创建一个全新的 AbilitySlice 的实例，这么做浪费大量资源。在实际应用中，这些 AbilitySlice 往往存在关联，因此常常只需要传递一些数据，并且也不需要每次跳转都创建一个新的实例。

实际上，AbilitySlice 可以层叠，即新创建的 AbilitySlice 可以叠加在原先的 AbilitySlice 之上，并且原先的 AbilitySlice 并不需要被销毁。当叠在最上层的 AbilitySlice 失效以后，就可以露出原先的 AbilitySlice 了。这种路由模式被称为层级路由。

接下来，将平级路由模式更改为层级路由模式，并实现 AbilitySlice 的数据传递。

4. AbilitySlice 的数据传递

AbilitySlice 之间通过 Intent 对象传递信息。细心的读者可能已经发现了，在刚才的实例中，present 方法已经传递了一个 Intent 对象。目前，还没有在这个 Intent 对象中设置任何参数。接下来，就需要对这个 Intent 对象做些“手脚”了，让它成为沟通 AbilitySlice 的桥梁。

接下来，对上面的程序进行一些修改，实现以下功能：默认 AbilitySlice 仍然是 MainAbilitySlice，并且显示计数 1。单击 MainAbilitySlice 上的文本组件后进入 SecondAbilitySlice，并将计数传递到 SecondAbilitySlice，通过逻辑代码使其加 1，显示计数为 2。单击 SecondAbilitySlice 的文本组件后退出 SecondAbilitySlice 并返回 MainAbilitySlice。在 MainAbilitySlice 中，通过逻辑代码使计数加 1，变为 3。如此循环，每次单击屏幕文本框都会经历 AbilitySlice 的跳转，且显示的计数依次增加，如图 3-17 所示。

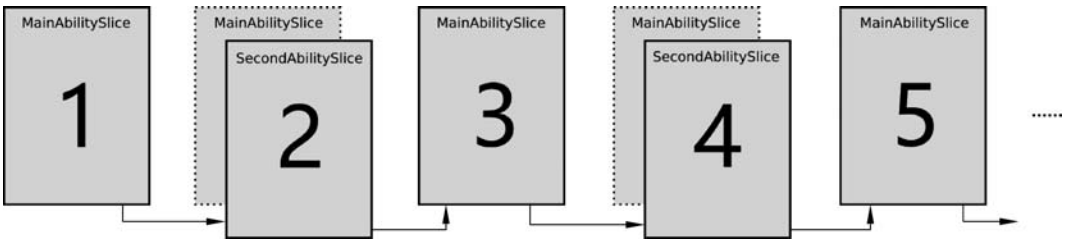

图 3-17　AbilitySlice 的层级跳转和数据传递

下面介绍这个功能的实现过程。

首先，在 MainAbilitySlice 中进行一些修改，代码如下：

```
//chapter3/AbilitySliceNavigation2/entry/src/main/java/com/example/abilityslicenavigation/
//slice/MainAbilitySlice.java
//文本组件
private Text text;
//计数变量
private int count = 1;
@Override
public void onStart(Intent intent) {
    super.onStart(intent);
    ……
    //Text text = new Text(this);                    //创建文本对象 text
    text = new Text(this);                           //初始化文本对象 text
    text.setLayoutConfig(config);                    //设置布局配置对象
    //text.setText("MainAbilitySlice");              //将内容字符串设置为"MainAbilitySlice"
    text.setText("" + count);                        //将内容字符串设置为计数
    ……
    text.setClickedListener(new Component.ClickedListener() {
        @Override
```

```
        public void onClick(Component component) {
            //present(new SecondAbilitySlice(), new Intent());
            //创建 Intent 对象
            Intent _intent = new Intent();
            //设置 intent 对象的计数参数
            _intent.setParam("count", count);
            //启动 SecondAbilitySlice
            presentForResult(new SecondAbilitySlice(), _intent, 0x00101);
        }
    });
}
@Override
protected void onResult(int requestCode, Intent resultIntent) {
    if (requestCode == 0x00101) {
        //获取返回的计数值,其中第 1 个参数为键,第 2 个参数为默认值
        count = resultIntent.getIntParam("count", 1);
        //count 自增 1 后显示在 text 文本组件上
        text.setText("" + ++count);
    }
}
```

在这段代码中,主要包括以下几个方面的修改:

(1) 将文本组件对象 text 改为 MainAbilitySlice 的私有成员变量,方便其他方法的调用。

(2) 增加计数变量 count,用于表示当前的计数情况。

(3) 在单击文本框后,创建了 Intent 对象_intent。通过该对象的 setParam 方法设置了一个计数参数。Intent 对象的各类参数均通过键值对的方式进行设置,在本例中将字符串 count 作为键,将计数变量 count 作为值传入_intent 对象中。最后,通过 presentForResult 方法启动 SecondAbilitySlice。presentForResult 方法与 present 方法类似,只是增加一个整型类型的 requestCode 参数。requestCode 表示请求代码,用于接收新启动的 SecondAbilitySlice 所返回的数据,并标识返回的结果。

(4) 实现 MainAbilitySlice 的 onResult 方法。该方法用户接收新启动的 AbilitySlice 所返回的数据。onResult 方法包含两个参数,分别为整型的 requestCode 请求代码和 resultIntent 对象。当 SecondAbilitySlice 退出并返回数据时,会调用 MainAbilitySlice 的 onResult 方法,且其 requestCode 请求代码与启动该 SecondAbilitySlice 时所设置的 requestCode 请求代码相同。resultIntent 对象用于存储 SecondAbilitySlice 所返回的具体数据,包括 getIntParam、getFloatParam、getDoubleParam 等多种方法,分别用于获取不同类型的参数数据。这些方法都包含两个参数,其中第 1 个参数为获取参数的键,第 2 个参数为默认值(当没有找到键值对时返回的数值)。

然后，对 SecondAbilitySlice 进行一些修改，代码如下：

```
//chapter3/AbilitySliceNavigation2/entry/src/main/java/com/example/abilityslicenavigation/
//slice/SecondAbilitySlice.java
//计数变量
private int count;
@Override
protected void onStart(Intent intent) {
    super.onStart(intent);

    count = intent.getIntParam("count", 1);        //获取传递的 count 计数值
    count++;                                       //count 计数值自增 1
    ……
    //text.setText("SecondAbilitySlice");
    text.setText("" + count);                      //显示计数值
    ……
    text.setClickedListener(new Component.ClickedListener() {
        @Override
        public void onClick(Component component) {
            //present(new MainAbilitySlice(), new Intent());
            //创建返回 MainAbilitySlice 的 Intent 对象
            Intent resultintent = new Intent();
            //设置计数值参数
            resultintent.setParam("count", count);
            //将返回的 Intent 对象设置为 resultintent
            setResult(resultintent);
            //结束当前的 SecondAbilitySlice
            terminate();
        }
    });
}
```

在这段代码中，主要包括以下几个方面的修改：

（1）增加计数变量 count，用于表示当前的计数情况。

（2）通过 onStart 方法的 intent 对象获取传入的计数值，并将该计数值显示在 text 文本组件中。

（3）在单击 text 文本组件时，通过 setResult 方法设置返回的上一层 AbilitySlice（MainAbilitySlice）的结果 Intent 对象 resultintent。通过 terminate 方法结束当前的 AbilitySlice。

编译并运行程序，就会达到预期的效果：单击屏幕时，会不断地跳入和跳出 SecondAbilitySlice，同时屏幕上的数字依次增加，如图 3-18 所示。

5. AbilitySlice 栈

层级跳转实际上是 AbilitySlice 的叠加。这个叠加过程是发生在被称为 AbilitySlice 栈上的。每个 Page 中都存在一个 AbilitySlice 栈。AbilitySlice 栈遵循着后入先出（LIFO）的原则，处在栈顶的 AbilitySlice 永远会先于底层的 AbilitySlice 出栈。

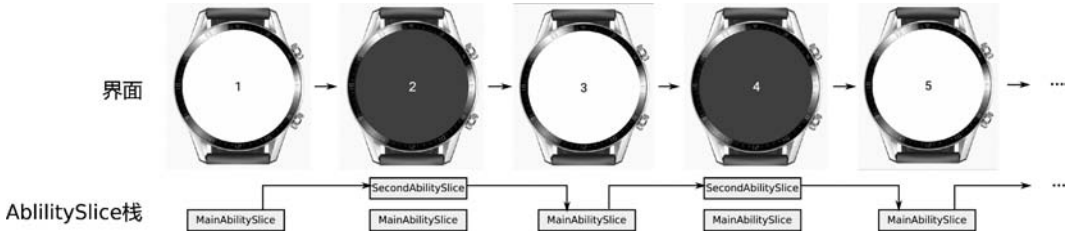

图 3-18　AbilitySlice 的层级跳转效果

在上例中，SecondAbilitySlice 实际上是叠加在 MainAbilitySlice 上的。MainAbility 作为 1 个 Page 类型的 Ability，存在 1 个 AbilitySlice 栈。在一开始，栈中仅有 1 个 MainAbilitySlice(主路由)。单击 MainAbilitySlice 界面中的文本组件后，SecondAbilitySlice 入栈。这时，实际上 MainAbilitySlice 仍然存在于界面中，只是完全被 SecondAbilitySlice 遮挡，因此 MainAbilitySlice 的生命周期会进入后台态。当用户单击 SecondAbilitySlice 界面中的文本组件时，SecondAbilitySlice 会被出栈销毁。此时，MainAbilitySlice 重见天日，再次回到前台。在随后的操作中会循环这一过程。

在开发过程中，一定要随时注意 AbilitySlice 栈中所包含的 AbilitySlice。如果栈中的 AbilitySlice 过多，会大量占据设备内存，影响用户体验。

3.3.2　Page 的显式跳转

Page 的跳转与 AbilitySlice 路由非常类似，传递数据同样采用 Intent 类进行传递。不过跳转的“目的地”就需要 Operation 类来帮忙设置了。

由于 Intent 分为显式(Explicit)Intent 和隐式(Implicit)Intent 两类，因此这里将 Page 的跳转也分为 Page 的显式跳转和隐式跳转。顾名思义，显式 Intent 更加直白，直接指定被跳转的目标位置。

隐式 Intent 则指定 Action 等方式进行跳转。跳转的能力需要被跳转的目标 Ability 所定义，并暴露出 Action 等接口，因此显得“含蓄”很多。通常，隐式 Intent 能够实现更加复杂的跳转功能，将在 3.3.3 节中进行详细介绍。

本节主要介绍 Page 的显式跳转。

在介绍具体的内容之前，需要先创建一个名为 PageNavigation 的新工程，选择目标设备为 Wearable，并使用 Empty Feature Ability(Java)模板。接下来，将介绍如何创建一个 SecondAbility，并实现 MainAbility 与 SecondAbility 之间的跳转。

1. 创建 SecondAbility

首先，在 Project 工具窗体中，在 entry 目录上右击，选择 New→Ability→Empty Page Ability(Java)菜单，弹出如图 3-19 所示的对话框。

在该对话框中，在 Page Name 选项中输入 Page 的名称 SecondAbility；在 Package name 中选择该类所在的包 com.example.pagenavigation；在 Layout Name 选项中输入布

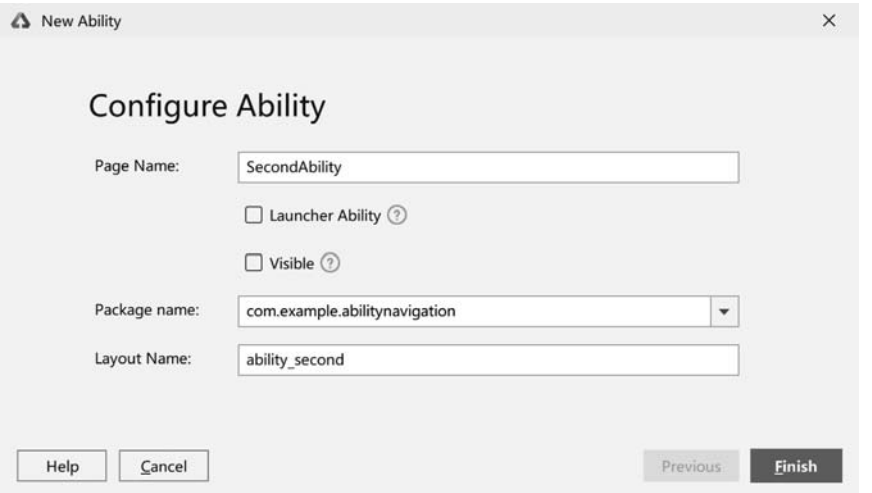

图 3-19　新建 SecondAbility 类

局文件的名称 ability_second。单击 Finish 按钮即可创建 SecondAbility 类。另外，在创建 SecondAbility 类的同时还创建了其主路由 SecondAbilitySlice，以及其布局文件 ability_second. xml。

这样，该工程中就有了两个 Page：MainAbility 和 SecondAbility。

2. 设置布局文件内容

在默认情况下，AbilitySlice 采用 xml 的方式定义用户界面，这种方式非常简单易用。目前，MainAbility 的主路由 AbilitySlice 为 MainAbilitySlice，其布局文件为 ability_main. xml；SecondAbility 的主路由 AbilitySlice 为 SecondAbilitySlice，其布局文件为 ability_second. xml。

为了能够区分这两个 Page，现在修改两个 xml 布局文件，以便显示不同的文本内容。首先，在 Project 工具窗体中定位并打开 ability_main. xml 文件，代码如下：

```
//chapter3/PageNavigation/entry/src/main/resources/base/layout/ability_main.xml
<?xml version = "1.0" encoding = "utf - 8"?>
< DirectionalLayout
    xmlns:ohos = "http://schemas.huawei.com/res/ohos"
    ohos:height = "match_parent"
    ohos:width = "match_parent"
    ohos:orientation = "vertical">

    < Text
        ohos:id = " $ + id:text_main"
        ohos:height = "match_parent"
        ohos:width = "match_content"
        ohos:layout_alignment = "horizontal_center"
        ohos:text = "MainAbility"
```

```
        ohos:text_size = "50"
    />

</DirectionalLayout >
```

在这个布局文件中，通过 DirectionalLayout 标签定义了一个定向布局（占据整个屏幕大小）。该定向布局中仅包含 1 个 Text 文本组件，其中 ohos:id 属性定义了其标识 ID，随后即可在 Java 代码中获取这个对象；ohos:text 属性定义了其文本内容 MainAbility。

类似地，修改 ability_second.xml 文件（与 ability_main.xml 在同一目录），代码如下：

```
//chapter3/PageNavigation/entry/src/main/resources/base/layout/ability_second.xml
<?xml version = "1.0" encoding = "utf - 8"?>
< DirectionalLayout
    xmlns:ohos = "http://schemas.huawei.com/res/ohos"
    ohos:height = "match_parent"
    ohos:width = "match_parent"
    ohos:orientation = "vertical"
    ohos:background_element = "#9090FF"> <!-- 浅蓝色背景 -->

    < Text
        ohos:id = " $ + id:text_second"
        ohos:height = "match_parent"
        ohos:width = "match_content"
        ohos:layout_alignment = "horizontal_center"
        ohos:text = "SecondAbility"
        ohos:text_size = "50"
    />

</DirectionalLayout >
```

此时，读者可以在 config.json 中切换默认启动的 FA（具体的方法可参见 3.2.2 节），MainAbility 和 SecondAbility 的显示效果分别如图 3-20 和图 3-21 所示。

图 3-20　MainAbility 的用户界面

图 3-21　SecondAbility 的用户界面

3. 实现从 MainAbility 跳转到 SecondAbility

由于在 ability_main.xml 中设置了文本 ID 为 text_main，此时会在 ResourceTable 类

中自动生成一个 Id_text_main 常量，通过这个常量和 findComponentById 方法就可以获取其 Java 对象，典型的代码如下：

```
Text text = (Text)findComponentById(ResourceTable.Id_text_main);
```

然后，为该对象设置一个单击监听器，在单击该文本后创建一个 Intent 对象和一个 Operation 对象。通过 Operation 对象指定跳转目标 Ability，并将 Operation 对象传递给 Intent 对象。最后，通过 startAbility 方法跳转目标的 Ability。

```
//chapter3/PageNavigation/entry/src/main/java/com/example/pagenavigation/slice/
//MainAbilitySlice.java
public class MainAbilitySlice extends AbilitySlice {
    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
        super.setUIContent(ResourceTable.Layout_ability_main);
        //获取文本组件对象
        Text text = (Text)findComponentById(ResourceTable.Id_text_main);
        //设置单击监听器
        text.setClickedListener(new Component.ClickedListener() {
            @Override
            public void onClick(Component component) {
                //创建 Intent 对象
                Intent _intent = new Intent();
                //创建 Operation 对象
                Operation operation = new Intent.OperationBuilder() //创建 Operation 对象
                        .withDeviceId("")           //目标设备,空字符串代表本设备
                        .withBundleName("com.example.pagenavigation")
                                                    //通过 BundleName 指定应用程序
                        .withAbilityName("com.example.pagenavigation.SecondAbility")
                                    //通过 Ability 的全名称(包名 + 类名)指定启动的 Ability
                        .build();
                //设置 Intent 对象的 operation 属性
                _intent.setOperation(operation);
                //启动 Ability
                startAbility(_intent);
            }
        });
    }
}
```

这里通过建造者模式创建了 Operation 对象，主要包括 3 个参数：DeviceId、BundleName 和 AbilityName。通过 DeviceId 指定启动 Ability 的设备，通过 BundleName 指定启动的应用程序，通过 AbilityName 指定具体需要启动的 Ability。由此可见，

Operation 对象具备了分布式能力，可以跨设备、跨应用地启动 Ability。

然后，在 SecondAbilitySlice 实现退出当前 Ability 的功能，代码如下：

```
//chapter3/PageNavigation/entry/src/main/java/com/example/pagenavigation/slice/
//SecondAbilitySlice.java
public class SecondAbilitySlice extends AbilitySlice {
    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
        super.setUIContent(ResourceTable.Layout_ability_second);
        //获取文本组件对象
        Text text = (Text)findComponentById(ResourceTable.Id_text_second);
        //设置文本组件的单击监听器
        text.setClickedListener(new Component.ClickedListener() {
            @Override
            public void onClick(Component component) {
                //结束当前的 Ability
                terminateAbility();
            }
        });
    }
}
```

此时，编译并运行 PageNavigation 程序，即可单击屏幕任意位置实现两个 Ability 的相互跳转，如图 3-22 所示。

图 3-22　Ability 的跳转

4. Page Ability 栈与 Page 的启动模式

多个 Page 被 Page 栈进行管理。在一个鸿蒙应用程序中，一般仅存在一个 Page 栈。与 AbilitySlice 栈类似，Page 栈遵循着后入先出（LIFO）的原则，处在栈顶的 Page 永远会先于底层的 Page 出栈。

在上例中，Page 显式跳转实际上是 SecondAbility 的入栈和出栈过程，如图 3-23 所示。与 3.3.1 节中的 AbilitySlice 栈非常类似，这个过程很容易被理解，这里不再赘述。

在将 Page 的启动模式设置为标准模式的情况下，应用程序仅存在 1 个 Page 栈，但是，如果指定某个 Page 的启动模式为单例模式，则应用程序会另创建 1 个新的 Page 栈，用于管理这个单例模式的 Page。如此一来，就可以保证这个 Page 始终不会被其他 Page 所遮盖，

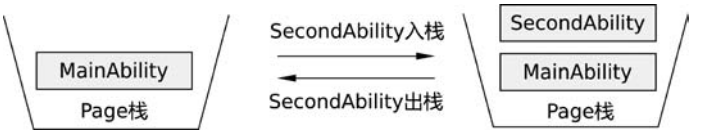

图 3-23　Ability 的入栈和出栈

从而方便开发者调用。Page 的单例模式类似于 Android 中 Activity 的 singleInstance。通常，账号登录注册界面、拍照录像界面等常用 Page 单例模式。

5. Ability 的数据传递

Ability 的数据传递与 AbilitySlice 的数据传递非常类似，只不过需要用 startAbilityForResult 方法代替 startAbility 方法跳转 Ability。至于其他传递数据的方法可参见 3.3.1 节的相关内容实现，这里不再详细介绍。

3.3.3　Page 的隐式跳转

Page 的显式跳转可以满足绝大多数的需求，但是 Page 的隐式跳转可以实现更加高级的功能。

例如，Page1 中包含了 Slice1 和 Slice2 两个 AbilitySlice，并且 Slice1 为主路由。如果希望从 Page2 直接跳转到 Page1 中的 Slice2 该怎么办呢？如果通过显式跳转，则首先需要从 Page1 跳转到 Page2，然后从 Slice1 跳转到 Slice2。这种方法需要经过两次跳转，并且必须经过 Slice1。通过隐式跳转就可以直接避免经过 Slice1，而直接从 Page2 跳转到 Page1 中的 Slice2。

另外，Page 的隐式跳转还可以轻松地实现跳转到桌面等场景。

下面以两个实例来介绍 Page 隐式跳转的用法。

1. 跳转到指定 Page 的指定 AbilitySlice

在开始介绍正式内容之前，先做一些准备工作：

首先，需要创建一个名为 PageNavigationImplicit 的新工程，选择目标设备为 Wearable，并使用 Empty Feature Ability(Java)模板。

然后，在 PageNavigationImplicit 工程中创建一个名为 SecondAbility 的新 Page，同时创建其主路由 SecondAbilitySlice。创建另外一个名为 TargetAbilitySlice 的 AbilitySlice，与 SecondAbilitySlice 一并被 SecondAbility 管理，如图 3-24 所示。

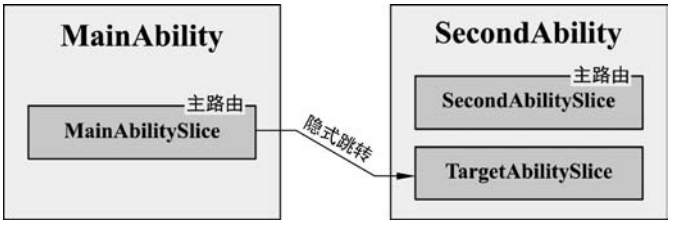

图 3-24　MainAbilitySlice、SecondAbilitySlice 和 TargetAbilitySlice 的关系

最后，让 MainAbilitySlice、SecondAbilitySlice 和 TargetAbilitySlice 的用户界面分别显示 MainAbilitySlice、SecondAbilitySlice 和 TargetAbilitySlice 的文本组件。

接下来，实现通过 Page 隐式跳转的方法从 MainAbility 的 MainAbilitySlice 直接跳转到 SecondAbility 的 TargetAbilitySlice。

（1）在 config.json 中的 SecondAbility 配置选项中声明 Action，代码如下：

```
//chapter3/PageNavigationImplicit/entry/src/main/config.json
{
  "skills": [
    {
      "actions": [
        "action.intent.targetabilityslice"
      ]
    }
  ],
  "name": "com.example.pagenavigationimplicit.SecondAbility",
  …
}
```

这里的 Action 名称可以任意起名，但一般以 action.intent. 开头。

（2）在 SecondAbility.java 中，添加 Action 路由，代码如下：

```
//chapter3/PageNavigationImplicit/entry/src/main/java/com/example/pagenavigationimplicit/
//SecondAbility.java
public class SecondAbility extends Ability {
    //声明 Action,需要与 config.json 中的 Action 声明字符串一致
    public static final String ACTION_TARGET = "action.intent.targetabilityslice";
    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
        super.setMainRoute(SecondAbilitySlice.class.getName());
        //增加 Action 路由
        super.addActionRoute(ACTION_TARGET, TargetAbilitySlice.class.getName());
    }
}
```

（3）在 MainAbilitySlice.java 中，实现单击文本跳转到 TargetAbilitySlice，代码如下：

```
//chapter3/PageNavigationImplicit/entry/src/main/java/com/example/pagenavigationimplicit/
slice/MainAbility.java
@Override
public void onStart(Intent intent) {
    super.onStart(intent);
    super.setUIContent(ResourceTable.Layout_ability_main);
```

```
        Text text = (Text) findComponentById(ResourceTable.Id_text_main);
        text.setClickedListener(new Component.ClickedListener() {
            @Override
            public void onClick(Component component) {
                Intent _intent = new Intent();
                Operation operation = new Intent.OperationBuilder()
                        .withAction(SecondAbility.ACTION_TARGET)
                        .build();
                _intent.setOperation(operation);
                startAbility(_intent);
            }
        });
    }
```

与显式跳转不同,隐式跳转仅设置了 Operation 的 action 属性。可以说,所有的跳转属性(如 Action 路由)都由被跳转的 Page 所管理。主动权交给了被跳转的 Page。

编译并运行程序,单击 MainAbilitySlice 按钮后即可直接跳转到 TargetAbilitySlice,然后,单击返回按钮,可以发现用户界面直接返回了 MainAbilitySlice。整个过程中并没有经过 SecondAbilitySlice。

2. 预置 Action

除了自定义的 Action 以外,鸿蒙 API 还定义了许多系统预置的 Action。这些 Action 被包含在 IntentConstants 类之中,例如 ACTION_HOME(系统桌面)、ACTION_DIAL(拨号界面)、ACTION_SEARCH(搜索界面)、ACTION_MANAGE_APPLICATIONS_SETTINGS(系统设置界面)等。通过这些 Action 可以进入相应的系统界面。

例如,进入拨号界面的代码如下:

```
Intent _intent = new Intent();
Operation operation = new Intent.OperationBuilder()
        .withAction(IntentConstants.ACTION_DIAL)
        .build();
_intent.setOperation(operation);
startAbility(_intent);
```

3.4 应用资源

24min

在应用程序开发过程中,一个非常重要的思想就是保持表现与数据的分离。绝大多数的开发者对这种思想应该并不陌生。例如,典型的 MVC 模式就是将数据置入模型层,将界面和界面行为置入表现层,并通过控制器进行两者的沟通和管理。

然而,数据的含义是非常广泛的,不仅包括关系型数据,还包括各种类型的视频、声频、图

像等以文件形式存储的数据，更包括了应用程序中所引用的字符串、整型数字等对象或数值。

在由初学者所开发的程序中，常常会将许多固定的字符串、固定的数值写入程序代码之中，例如文本组件前的“用户名：”“密码：”等提示性字符串等。这样会导致两个问题：一是加大了后期的维护成本，其他开发者需要通过上下文理解这些字符串或数值的含义；二是难以实现国际化。为此，建议开发者将与应用程序密切相关的各类字符串、数值、图形等放置到应用资源中。

本节介绍应用资源的基本使用方法。

3.4.1　应用资源的分类与引用

应用资源通常被放置在鸿蒙应用程序工程中 HAP 下的 src/main/resources 目录中。在该目录下包含了 base 和 rawfile 两个目录。这两个目录代表了应用资源的两种类型：base 资源具有更强的组织方式，在编译过程中会被编译成二进制文件，并赋予相应的资源标识符。

处理 rawfile 资源就非常简单了，其目录结构可以由开发者随意组织，并且在编译过程中不会被编译为二进制码。

注意：base 资源类似于 Android 中的 res 资源，rawfiles 资源类似于 Android 中的 assets 资源。

一般情况下，更加推荐使用 base 应用资源。

1. base 应用资源

在 base 目录中，包含了元素资源(element)、可绘制资源(graphic)、布局资源(layout)、媒体资源(media)、动画资源(animation)、其他资源(profile)等若干类型，如表 3-2 所示。

表 3-2　base 应用资源类型

资源类型	存储位置	Java 引用格式	XML/JSON 引用格式
颜色资源	./base/element/color.json	ResourceTable.Color_*	$color:*
布尔型资源	./base/element/boolean.json	ResourceTable.Boolean_*	$boolean:*
整型资源	./base/element/integer.json	ResourceTable.Integer_*	$integer:*
整型数组资源	./base/element/intarray.json	ResourceTable.Intarray_*	$intarray:*
浮点型资源	./base/element/float.json	ResourceTable.Float_*	$float:*
复数资源	./base/element/plural.json	ResourceTable.Plural_*	$plural:*
字符串资源	./base/element/string.json	ResourceTable.String_*	$string:*
字符串数组资源	./base/element/strarray.json	ResourceTable.Strarray_*	$strarray:*
样式资源	./base/element/pattern.json	ResourceTable.Pattern_*	$pattern:*
可绘制资源	./base/graphic/*	ResourceTable.Graphic_*	$graphic:*
布局资源	./base/layout/*	ResourceTable.Layout_*	$layout:*
媒体资源	./base/media/*	ResourceTable.Media_*	$media:*
动画资源	./base/animation/*	ResourceTable.Animation_	$animation:*
其他资源	./base/profile/*	ResourceTable.Profile_	$profile:*

其中，颜色资源（color）、布尔型资源（boolean）、整型资源（integer）、整型数组资源（intarray）、浮点型资源（float）、复数资源（plural）、字符串资源（string）、字符串数组资源（strarray）、样式资源（pattern）都属于元素资源。

除上述表格列出的 base 应用资源以外，还包括一种特殊的资源类型：ID 资源。ID 资源用于标识布局资源的各类组件。通过 ID 资源的唯一标识符，开发者可以在 Java 代码中获取相应的组件对象。

上面这些资源都可以在 Java 代码中或 XML/JSON 文件中引用。在 XML/JSON 文件中，基本的引用形式为“$type:name”。其中，type 为资源类型，name 为资源名称。在 Java 代码中，开发者可以使用 ResourceTable 自动生成的唯一标识符进行引用。在 3.1.3 节和 3.1.4 节中，读者已经学习了资源应用的基本使用方法，这里不再赘述。

值得注意的是，除了用户可以自定义资源以外，还可以使用全局资源。例如，鸿蒙 API 在全局资源中提供了默认的应用程序图标。在 XML/JSON 文件中引用这个默认图标的方法为“$ohos:media:ic_app”。可见，全局资源的应用格式只需加上“ohos:”标识，其基本引用格式为“$ohos:type:name”。在 Java 文件中引用这个默认图标的唯一标识符为 ohos.global.systemres.ResourceTable.Media_ic_app。注意，这里的 ResourceTable 的包名为 ohos.global.systemres，读者不要混淆。

注意：除了上述全局资源以外，还包括 request_location_reminder_title 和 request_location_reminder_content 这两个字符串资源，分别为请求使用设备定位功能的提示标题和提示内容。

2. rawfile 应用资源

rawfile 应用资源无法在 XML 和 JSON 中使用，只能通过 Java 代码获取其内容，代码如下：

```
RawFileEntry entry = getResourceManager()
        .getRawFileEntry("resources/rawfile/icon.png");
HiLog.info(loglabel, "文件类型:" + entry.getType().name());
```

通过 RawFileEntry 对象的 openRawFileDescriptor().getFileDescriptor()方法即可获得其 FileDescriptor 对象，代码如下：

```
FileDescriptor fd = entry.openRawFileDescriptor().getFileDescriptor()
```

随后，即可通过 Java API 中的 FileReader 对 FileDescriptor 所指代的文件内容进行读取。

另外，还可以通过 openRawFile()方法打开资源文件，并获得其 Resource 资源对象。通过 Resource 资源对象即可读取其二进制数据。

```
RawFileEntry entry = getResourceManager()
        .getRawFileEntry("resources/rawfile/icon.png");
```

```
try {
    //打开资源文件
    Resource resource = entry.openRawFile();
    //通过 available()方法获得文件长度
    int length = resource.available();
    //创建 Byte[]对象保存文件内容数据
    Byte[] Bytes = new Byte[length];
    //读取文件内容数据
    resource.read(Bytes, 0, length);
} catch (IOException e) {
    e.printStackTrace();
}
```

3.4.2　常见应用资源的使用方法

本节介绍字符串资源、颜色资源和可绘制资源这 3 种常见应用资源的使用方法。

1. 字符串资源

字符串资源是最为常见的应用资源。字符串资源属于元素资源的一种。元素资源都是以键值对的方式存储在 JSON 文件中。

例如，在默认情况下，鸿蒙应用程序自动生成 resource/element/string.json 文件用于存储字符串资源，代码如下：

```
{
  "string": [
    {
      "name": "app_name",
      "value": "HelloWorld"
    },
    {
      "name": "mainability_description",
      "value": "Java_Phone_Empty Feature Ability"
    }
  ]
}
```

默认情况下，string.json 中包括了 app_name 和 mainability_description 两个字符串资源。键入 app_name 字符串资源的值为 HelloWorld。

随后，就可以在文本组件中使用这个字符串资源了，代码如下：

```
<Text
    ohos:text = " $ string:app_name"
    ...
/>
```

当然，也可以在 Java 代码中直接通过文本组件的 setText 方法设置字符串，代码如下：

```
Text text = new Text(getContext());
text.setText(ResourceTable.String_app_name);
```

如果仅希望获得这个字符串的值，则需要注意捕获异常，代码如下：

```
try {
    String strAppName = getResourceManager()
            .getElement(ResourceTable.String_app_name)
            .getString();
    HiLog.info(logLabel, strAppName);
} catch (Exception e) {
    HiLog.info(logLabel, e.toString());
}
```

所有的资源都是通过资源管理器（ResourceManager）获取的。在 Ability 或 AbilitySlice 中，通过 getResourceManager()方法即可获取资源管理器对象。

资源管理器对象的常用方法包括：

（1）getRawFileEntry(String path)：获取 rawFile 资源。

（2）getElement(int resid)：获取元素资源，随后可通过 Element 对象具体的 get 方法获取细分资源类型对象。

（3）getDeviceCapability()：获取设备能力(设备类型、屏幕密度、高度、宽度、是否为圆形屏幕等)。

（4）getResource(int resid)：获取资源对象，进而可以获得其二进制数据。

（5）getMediaPath(int resid)：获得媒体资源的路径。

2. 颜色资源

与字符串资源类似，但是需要在 resource/element 目录下手动创建一个名为 color.json 的颜色资源文件，然后添加一个名为热情粉色的颜色，代码如下：

```
{
  "color":[
    {
      "name":"hotpink",
      "value":"#FF69B4"
    }
  ]
}
```

颜色资源支持 4 种颜色值形式：

（1）#RGB，分别用 1 位十六进制数代表红(R)、绿(G)、蓝(B)的值。

（2）#ARGB 分别用 1 位十六进制数代表红(R)、绿(G)、蓝(B)的值，以及色彩的透明

度(A)。

(3) #RRGGBB 分别用 2 位十六进制数代表红(R)、绿(G)、蓝(B)的值。

(4) #AARRGGBB 分别用 2 位十六进制数代表红(R)、绿(G)、蓝(B)的值,以及色彩的透明度(A)。

注意:在颜色值中色彩的透明度在整个颜色值的最前端。在许多其他编程环境中,颜色值可能在最后端,需要注意区分。

随后,就可以在 Java 代码中使用这种颜色了,代码如下:

```
try {
    int hotPink = getResourceManager()
            .getElement(ResourceTable.Color_hotpink)
            .getColor();
    text.setTextColor(new Color(hotPink));
} catch (Exception e) {
    HiLog.info(label, e.toString());
}
```

注意,这里的 Color 类的包名为 ohos.agp.utils,不要与 ohos.agp.colors.Color 类相混淆。

另外,在这个 Color 类中还包含了许多预置颜色,如表 3-3 所示。

表 3-3　Color 预置颜色

常　量	名　称	颜　色　值
Color.BLACK	黑色	0xFF000000
Color.DKGRAY	暗灰色	0xFF444444
Color.GRAY	灰色	0xFF808080
Color.LTGRAY	亮灰色	0xFFCCCCCC
Color.RED	红色	0xFFFF0000
Color.MAGENTA	品红	0xFFFF00FF
Color.YELLOW	黄色	0xFFFFFF00
Color.GREEN	绿色	0xFF00FF00
Color.CYAN	青色	0xFF00FFFF
Color.BLUE	蓝色	0xFF0000FF
Color.WHITE	白色	0xFFFFFFFF
Color.TRANSPARENT	透明色	0x00000000

例如,可以通过这些常量设置状态栏和底部虚拟按键的背景颜色,代码如下:

```
//设置状态栏可见
getWindow().setStatusBarVisibility(Component.VISIBLE);
//将状态栏颜色设置为蓝色背景
getWindow().setStatusBarColor(Color.BLUE.getValue());
```

```
//将底部虚拟按键设置为红色背景
getWindow().setNavigationBarColor(Color.RED.getValue());
```

3. 可绘制资源

在默认的鸿蒙应用程序工程中，包含了一个默认的可绘制资源 background_ability_main，代码如下：

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
 < shape xmlns:ohos = "http://schemas.huawei.com/res/ohos"
        ohos:shape = "rectangle">
    < solid
        ohos:color = "＃FFFFFF"/>
</shape >
```

在 Java 代码中，应用这个可绘制资源的代码如下：

```
ShapeElement shapeElement = new ShapeElement(getContext(), ResourceTable.Graphic_background
_ability_main);
```

ShapeElement 为形状元素，继承于 ohos.agp.components.element.Element 类（注意不要和元素资源类 ohos.global.resource.Element 类混淆）。

1）形状元素的类型

通过形状元素可以定义 5 种类型的形状：Rectangle（矩形）、Oval（椭圆）、Line（直线）、Arc（弧线）和 Path（线段）。这些形状类型由 ShapeElement 的 5 个常量所定义。通过 ShapeElement 对象的 setShape 方法即可设置其形状元素类型。

例如，将一个组件的背景设置为红色椭圆的代码如下：

```
Component component = new Component(this);          //创建组件对象
ShapeElement element = new ShapeElement();          //创建形状元素对象
element.setShape(ShapeElement.OVAL);                //将形状元素设置为椭圆
element.setRgbColor(new RgbColor(255, 0, 0));       //将形状元素的颜色设置为红色
component.setBackground(element);                   //将背景设置为 element
```

上述代码的最终显示效果如图 3-25 所示。

另外，还可以在矩形的椭圆形状元素上设置圆角，代码如下：

```
element.setShape(ShapeElement.RECTANGLE);
element.setCornerRadius(50);
```

此时，将其设置为组件的背景，其显示效果如图 3-26 所示。

图 3-25　椭圆形状元素

图 3-26　带圆角的矩形形状元素

2）形状元素的渐变色

形状元素除了可以设置单一的色彩以外，还可以将其设置为渐变色。通过 setRgbColors 方法设置颜色数组；通过 setShaderType 方法设置渐变类型。

渐变类型包括 3 类，如图 3-27 所示。

- 线性渐变（LINEAR_GRADIENT_SHADER_TYPE）：沿着某个方向进行渐变。
- 辐射渐变（RADIAL_GRADIENT_SHADER_TYPE）：从中央向四周进行渐变。
- 梯度渐变（SWEEP_GRADIENT_SHADER_TYPE）：沿圆周进行渐变。

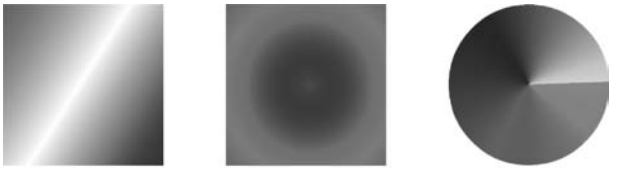

图 3-27　渐变类型

在线性渐变中，通过 setOrientation 方法可以设置线性渐变的方向。例如，一个典型的线性渐变矩形的代码如下：

```
ShapeElement element = new ShapeElement();                    //创建形状元素 element 对象
RgbColor[] colors = new RgbColor[3];
colors[0] = new RgbColor(255, 0, 0);                          //红色
colors[1] = new RgbColor(0, 255, 0);                          //绿色
colors[2] = new RgbColor(0, 0, 255);                          //蓝色
element.setRgbColors(colors);                                 //将 element 设置为渐变色
element.setShaderType(ShapeElement.LINEAR_GRADIENT_SHADER_TYPE);              //线性渐变
element.setOrientation(ShapeElement.Orientation.TOP_END_TO_BOTTOM_START);     //渐变方向
```

将该形状元素设置为组件的背景，显示效果如图 3-28 所示。

3）PixelMap 元素

通过 PixelMap 元素可以为组件设置图像背景。首先，需要获取 PixelMap 的 Resource 资源对象；然后创建 PixelMap 元素，并将 PixelMap 传入该对象；最后将该 PixelMap 元素设置为组件背景。

图 3-28　线性渐变形状元素

创建 PixelMap 元素的典型代码如下：

```
try {
    Resource pixmapRes = getResourceManager().getResource(ResourceTable.Media_icon);
    PixelMapElement element = new PixelMapElement(pixmapRes);
    component.setBackground(element);
} catch (Exception e) {
    e.printStackTrace();
}
```

3.4.3 限定词与国际化

1. 限定词

在之前的学习中，将图标、字符串等应用资源放置到 base 目录中。实际上，可以通过限定词的方式创建更多的资源目录，以适配不同的屏幕密度、不同的语言区域。

限定词可以包含以下几个部分：

(1) 语言：语言类型，用 2 个小写字母组成(采用 ISO 639-1 标准)。例如，zh 表示中文，en 表示英文。

(2) 文字：文字类型，由 1 个大写字母和 3 个小写字母组成(采用 ISO 15924 标准)。例如，Hans 表示简体中文，Hant 表示繁体中文。

(3) 国家或地区：国家或地区编码，由 2～3 个大写字母或者 3 个数字组成(采用 ISO 3166-1 标准)。例如，CN 表示中国，US 表示美国，JP 表示日本。

(4) 横竖屏：横屏(Horizontal)或竖屏(Vertical)。

(5) 设备类型：手机(Phone)、可穿戴设备(Wearable)、智慧屏(TV)等。

(6) 屏幕密度：包含 sdpi、mdpi、ldpi、xldpi、xxldpi、xxxldpi 等分级。每个分级都代表了一定的屏幕密度(DPI 或 PPI)的范围，各分级所代表的 DPI 范围如图 3-29 所示。

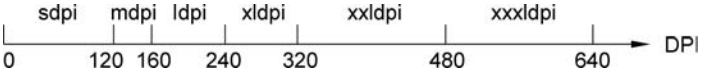

图 3-29 屏幕密度分级

限定词之间使用“-”或“_”连接。例如，可以创建限定词名为 zh_Hans_CN_vertical-phone-mdpi 目录。此时，该目录下的资源文件将在设备使用简体中文，所在国家为中国，设备类型为手机，屏幕为竖屏，且屏幕密度介于 120～160 时匹配使用。

注意：在匹配限定词时，各个限定词组成部分优先级从高到低依次为区域(语言、文字、国家或地区)> 横竖屏 > 设备类型 > 屏幕密度。

2. 国际化

接下来，以语言文字的国际化为例，介绍限定词的使用方法。

首先，在 resources 目录下，分别创建 en、zh_Hans 和 zh_Hant 限定词目录，然后分别在

这 3 个目录下创建 element 目录以及 element/string.json 文件，如图 3-30 所示。

图 3-30　国际化字符串资源文件

修改 en/element/string.json 文件，代码如下：

```
//chapter3/PageNavigationImplicit/entry/src/main/resources/en/element/string.json
{
  "string": [
    {
      "name": "harmonyos",
      "value": "HarmonyOS"
    }
  ]
}
```

修改 zh_Hans/element/string.json 文件，代码如下：

```
//chapter3/PageNavigationImplicit/entry/src/main/resources/zh_Hans/element/string.json
{
  "string": [
    {
      "name": "harmonyos",
      "value": "鸿蒙操作系统"
    }
  ]
}
```

修改 zh_Hant/element/string.json 文件，代码如下：

```
//chapter3/PageNavigationImplicit/entry/src/main/resources/zh_Hant/element/string.json
{
  "string": [
    {
      "name": "harmonyos",
      "value": "鴻濛作業系統"
    }
  ]
}
```

最后，将某个文本组件的内容设置为 harmonyos 字符串资源，代码如下：

```
text.setText(ResourceTable.String_harmonyos);
```

编译并运行程序，在鸿蒙操作系统中，进入【设置】→【系统与更新】→【语言与输入法】→【语言与地区】，切换【语言】选项为繁体中文、简体中文和英文。此时，上述文本组件中的显示内容会随着系统语言的变化而发生变化，如图 3-31 所示。

图 3-31　国际化显示效果

注意：上例中仅以语言文字的国际化为例介绍了限定词的使用方法。实际上，国际化的含义远超过翻译语言文字的范畴，还需要考虑到语言文字的方向(RTL、LTR)、布局和图标的方向、图片的禁忌等方面。绝大多数的国际化因素可以通过限定词的方式指定符合要求的资源文字。

3.5　本章小结

通过本章的学习，已经基本全面地了解了用户界面编程的基础知识。这主要包括：通过 XML 文件和 Java 代码的形式构建用户界面；使用 Page 和 AbilitySlice 的声明周期方法；了解 Page 栈和 AbilitySlice 栈及其作用；Page 和 AbilitySlice 的跳转；应用资源和限定词的使用方法。

通过这些知识，读者已经可以根据用户的需求构思和创建一个鸿蒙应用程序的框架了，但是，要做到无障碍开发还存在一定的距离。在用户界面方面，读者还需要掌握常见组件和布局的用法，希望读者继续学习下一章节的内容。为了达到炉火纯青、出神入化的开发水平，请大家继续加油学习吧！