Harmony OS 4.0 应用开发

讲师：木子

目 录

1 第一章 Harmony OS 概述

1.1 系统定义

1.1.1 系统定位

HarmonyOS 是一款面向万物互联时代的、全新的分布式操作系统。

在传统的单设备系统能力基础上，HarmonyOS 提出了基于同一套系统能力、适配多种终端形态的分布式理念，能够支持手机、平板、智能穿戴、智慧屏、车机、PC、智能音箱、耳机、AR/VR 眼镜等多种终端设备，提供全场景（移动办公、运动健康、社交通信、媒体娱乐等）业务能力。

HarmonyOS 有三大特征：

- **搭载该操作系统的设备在系统层面融为一体、形成超级终端，让设备的硬件能力可以弹性扩展，实现设备之间硬件互助，资源共享。**

对消费者而言，HarmonyOS 能够将生活场景中的各类终端进行能力整合，实现不同终端设备之间的快速连接、能力互助、资源共享，匹配合适的设备、提供流畅的全场景体验。

- **面向开发者，实现一次开发，多端部署。**

对应用开发者而言，HarmonyOS 采用了多种分布式技术，使应用开发与不同终端设备的形态差异无关，从而让开发者能够聚焦上层业务逻辑，更加便捷、高效地开发应用。

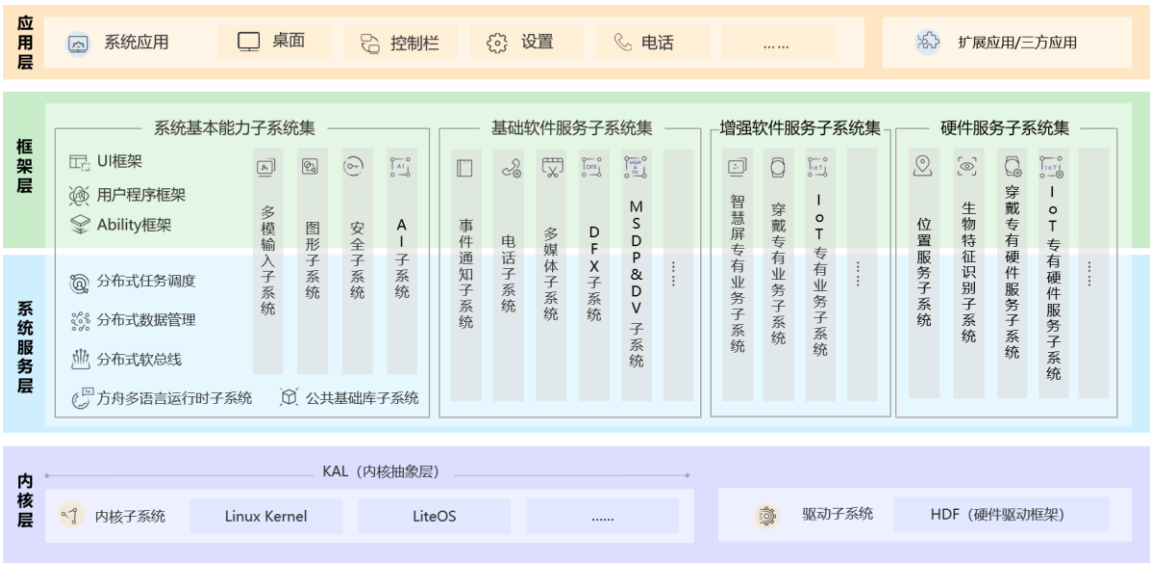
- **一套操作系统可以满足不同能力的设备需求，实现统一 OS，弹性部署。**

对设备开发者而言，HarmonyOS 采用了组件化的设计方案，可根据设备的资源能力和业务特征灵活裁剪，满足不同形态终端设备对操作系统的要求。

HarmonyOS 提供了支持多种开发语言的 API，供开发者进行应用开发。支持的开发语言包括 ArkTS、JS（JavaScript）、C/C++、Java。

1.1.2 技术架构

HarmonyOS 整体遵从分层设计，从下向上依次为：内核层、系统服务层、框架层和应用层。系统功能按照“系统 > 子系统 > 功能/模块”逐级展开，在多设备部署场景下，支持根据实际需求裁剪某些非必要的子系统或功能/模块。HarmonyOS 技术架构如下所示。



1.1.2.1 内核层

- 内核子系统：HarmonyOS 采用多内核设计，支持针对不同资源受限设备选用适合的 OS 内核。内核抽象层（KAL，Kernel Abstract Layer）通过屏蔽多内核差异，对上层提供基础的内核能力，包括进程/线程管理、内存管理、文件系统、网络管理和外设管理等。

- 驱动子系统：硬件驱动框架（HDF）是 HarmonyOS 硬件生态开放的基础，提供统一外设访问能力和驱动开发、管理框架。

1.1.2.2 系统服务层

系统服务层是 HarmonyOS 的核心能力集合，通过框架层对应用程序提供服务。该层包含以下几个部分：

- 系统基本能力子系统集：为分布式应用在 HarmonyOS 多设备上的运行、调度、迁移等操作提供了基础能力，由分布式软总线、分布式数据管理、分布式任务调度、方舟多语言运行时、公共基础库、多模输入、图形、安全、AI 等子系统组成。其中，方舟运行时提供了 C/C++/JS 多语言运行时和基础的系统类库，也为使用方舟编译器静态化的 Java 程序（即应用程序或框架层中使用 Java 语言开发的部分）提供运行时。

- 基础软件服务子系统集：为 HarmonyOS 提供公共的、通用的软件服务，由事件通知、电话、多媒体、DFX（Design For X）、MSDP&DV 等子系统组成。

- 增强软件服务子系统集：为 HarmonyOS 提供针对不同设备的、差异化的能力增强型软件服务，由智慧屏专有业务、穿戴专有业务、IoT 专有业务等子系统组成。

- 硬件服务子系统集：为 HarmonyOS 提供硬件服务，由位置服务、生物特征识别、穿戴专有硬件服务、IoT 专有硬件服务等子系统组成。

根据不同设备形态的部署环境，基础软件服务子系统集、增强软件服务子系统集、硬件服务子系统集内部可以按子系统粒度裁剪，每个子系统内部又可以按功能粒度裁剪。

1.1.2.3 框架层

框架层为 HarmonyOS 应用开发提供了 ArkTS/JS/C/C++/Java 等多语言的用户程序框架，两种 UI 框架（包括适用于 ArkTS/JS 语言的方舟开发框架即 ArkUI、适用于 Java 语言的 Java UI 框架），以及各种软硬件服务对外开放的多语言框架 API。根据系统的组件化裁剪程度，HarmonyOS 设备支持的 API 也会有所不同。

1.1.2.4 应用层

应用层包括系统应用和第三方非系统应用。HarmonyOS 的应用由一个或多个 FA（Feature Ability）或 PA（Particle Ability）组成。其中，FA 有 UI 界面，提供与用户交互的能力；而 PA 无 UI 界面，提供后台运行任务的能力以及统一的数据访问抽象。FA 在进行用户交互时所需的后台数据访问也需要由对应的 PA 提供支撑。基于 FA/PA 开发的应用，能够实现特定的业务功能，支持跨设备调度与分发，为用户提供一致、高效的应用体验。

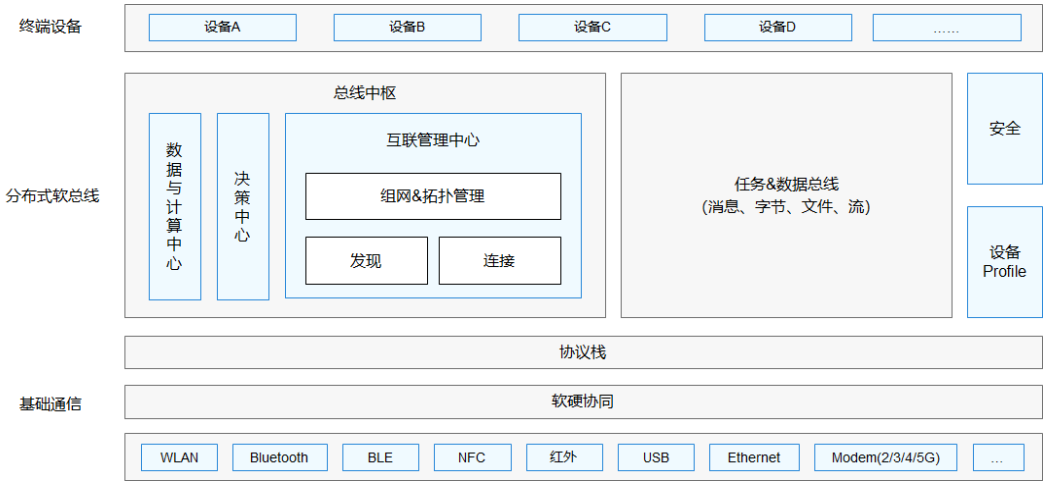
1.2 技术特性

1.2.1 硬件互助，资源共享

多种设备之间能够实现硬件互助、资源共享，依赖的关键技术包括分布式软总线、分布式设备虚拟化、分布式数据管理、分布式任务调度等。

1.2.1.1 分布式软总线

分布式软总线是手机、平板、智能穿戴、智慧屏、车机等分布式设备的通信基座，为设备之间的互联互通提供了统一的分布式通信能力，为设备之间的无感发现和零等待传输创造了条件。开发者只需聚焦于业务逻辑的实现，无需关注组网方式与底层协议。分布式软总线示意图如下。



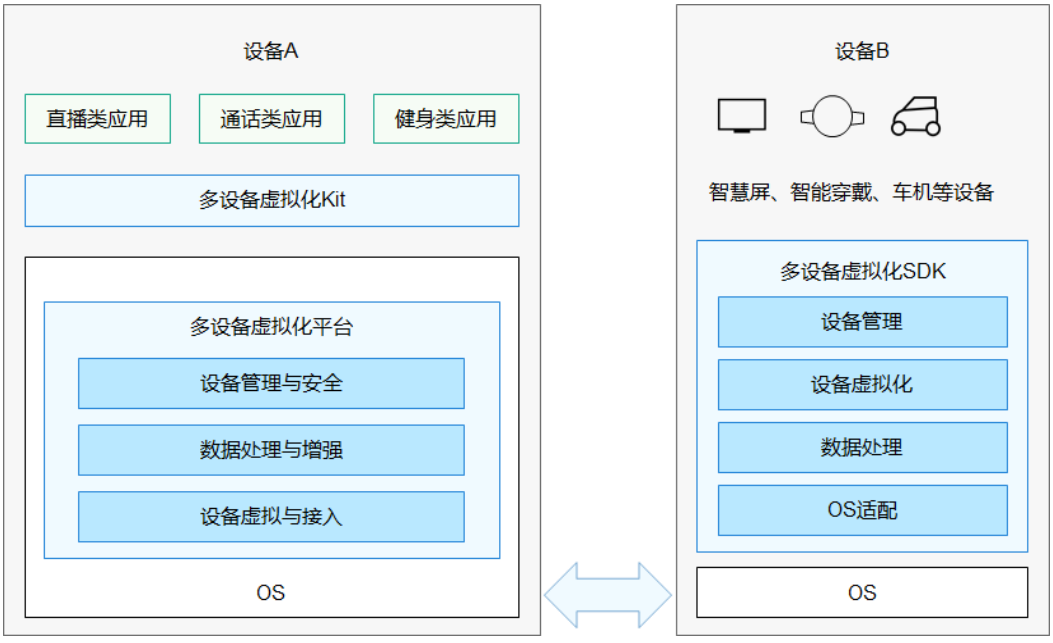
典型应用场景举例：

- 智能家居场景：在烹饪时，手机可以通过碰一碰和烤箱连接，并将自动按照菜谱设置烹调参数，控制烤箱来制作菜肴。与此类似，料理机、油烟机、空气净化器、空调、灯、窗帘等都可以在手机端显示并通过手机控制。设备之间即连即用，无需繁琐的配置。

- 多屏联动课堂：老师通过智慧屏授课，与学生开展互动，营造课堂氛围；学生通过平板完成课程学习和随堂问答。统一、全连接的逻辑网络确保了传输通道的高带宽、低时延、高可靠。

1.2.1.2 分布式设备虚拟化

分布式设备虚拟化平台可以实现不同设备的资源融合、设备管理、数据处理，多种设备共同形成一个超级虚拟终端。针对不同类型的任务，为用户匹配并选择能力合适的执行硬件，让业务连续地在不同设备间流转，充分发挥不同设备的能力优势，如显示能力、摄像能力、音频能力、交互能力以及传感器能力等。分布式设备虚拟化示意图如下。



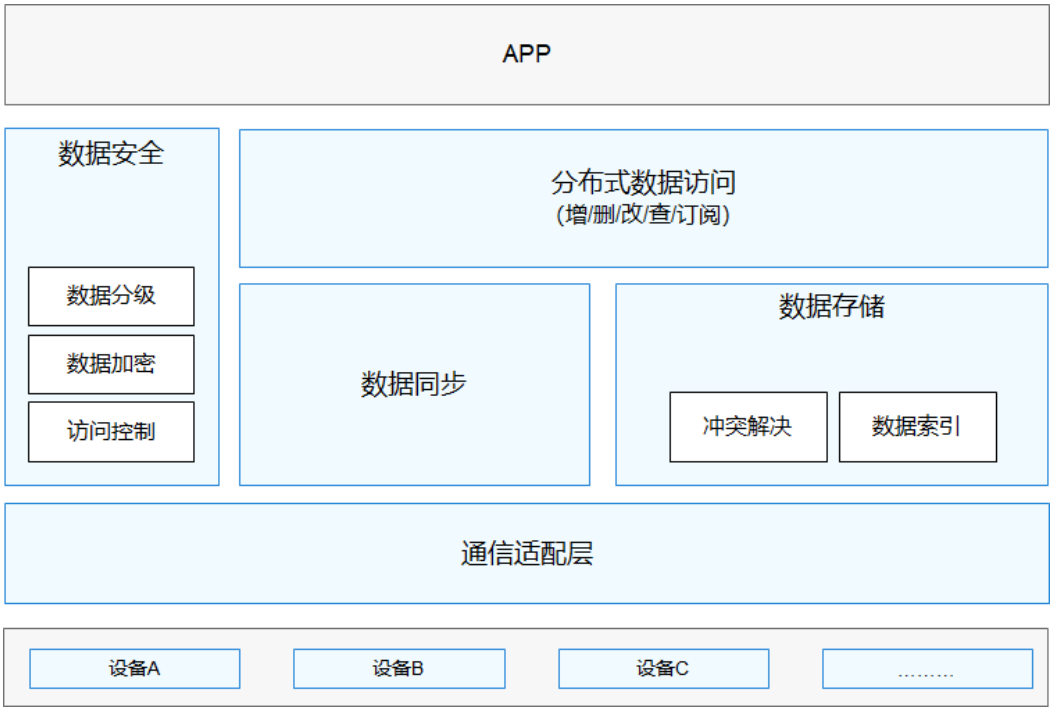
典型应用场景举例：

- 视频通话场景：在做家务时接听视频电话，可以将手机与智慧屏连接，并将智慧屏的屏幕、摄像头与音箱虚拟化为本地资源，替代手机自身的屏幕、摄像头、听筒与扬声器，实现一边做家务、一边通过智慧屏和音箱来视频通话。

- 游戏场景：在智慧屏上玩游戏时，可以将手机虚拟化为遥控器，借助手机的重力传感器、加速度传感器、触控能力，为玩家提供更便捷、更流畅的游戏体验。

1.2.1.3 分布式数据管理

分布式数据管理基于分布式软总线的能力，实现应用程序数据和用户数据的分布式管理。用户数据不再与单一物理设备绑定，业务逻辑与数据存储分离，跨设备的数据处理如同本地数据处理一样方便快捷，让开发者能够轻松实现全场景、多设备下的数据存储、共享和访问，为打造一致、流畅的用户体验创造了基础条件。分布式数据管理示意图如下。



典型应用场景举例：

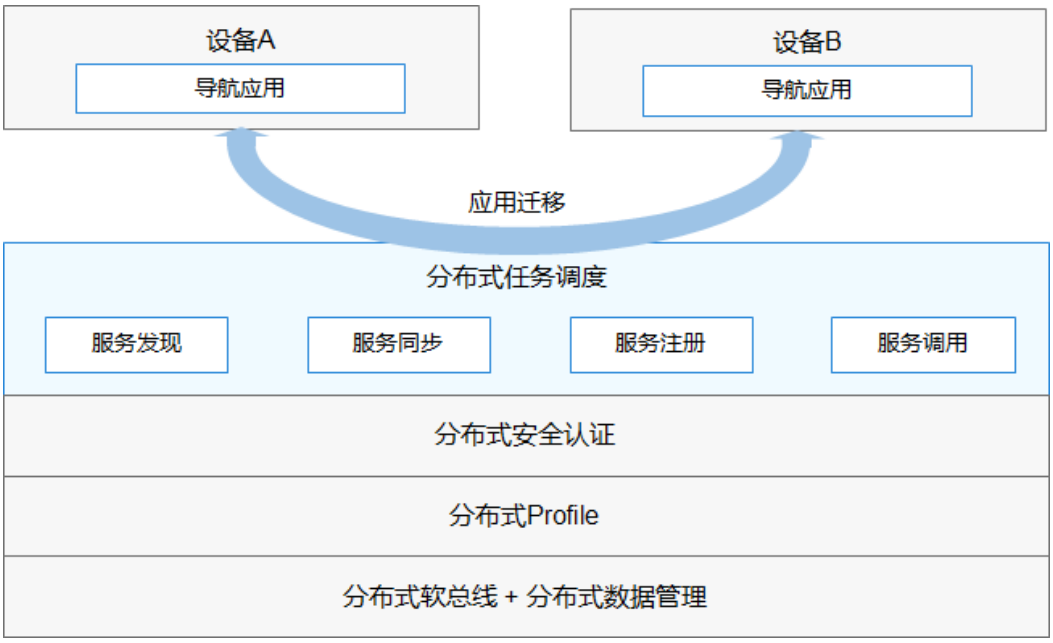
- 协同办公场景：将手机上的文档投屏到智慧屏，在智慧屏上对文档执行翻页、缩放、涂鸦等操作，文档的最新状态可以在手机上同步显示。

- 照片分享场景：出游时，使用手机拍摄的照片，可以在登录了同帐号的其他设备，比如平板上更方便地浏览、收藏、保存或编辑，也可以通过家中的智慧屏上同家人一起分享记录下的快乐瞬间。

1.2.1.4 分布式任务调度

分布式任务调度基于分布式软总线、分布式数据管理、分布式 Profile 等技术特性，构建统一的分布式服务管理（发现、同步、注册、调用）机制，支持对跨设备的应用进行远程启动、远程调用、远程连接以及迁移等操作，能够根据不同设备的能力、位置、业务运行状态、资源使用情况，以及用户的习惯和意图，选择合适的设备运行分布式任务。

下图以应用迁移为例，简要地展示了分布式任务调度能力。



典型应用场景举例：

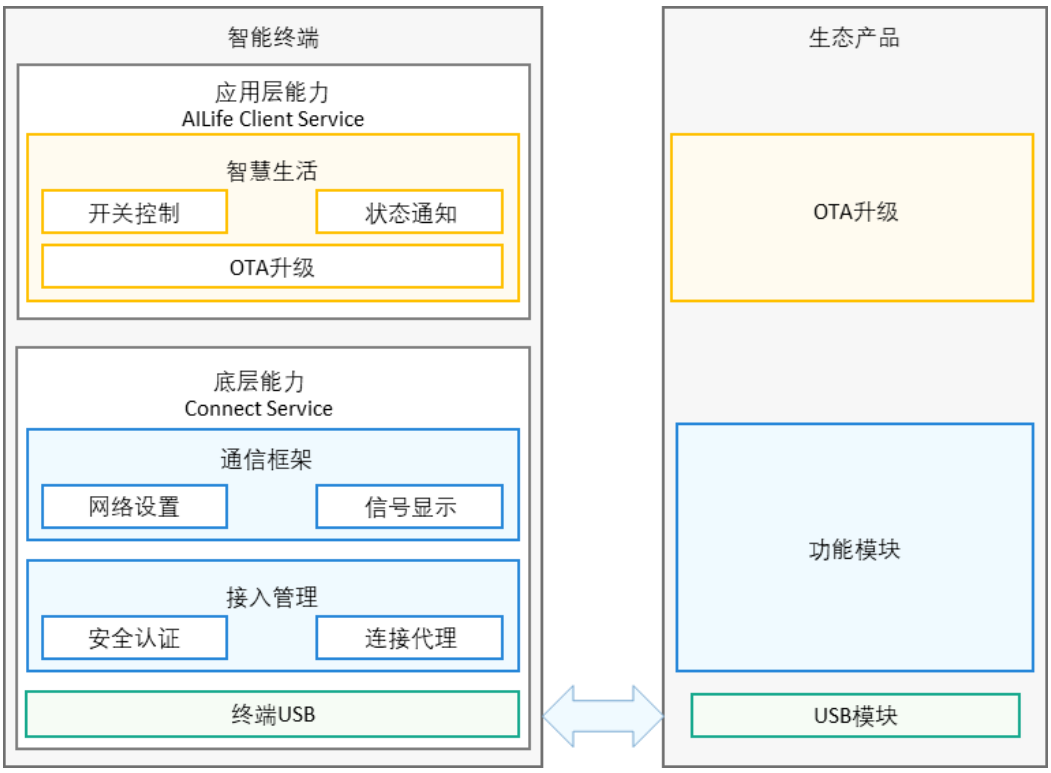
- 导航场景：如果用户驾车出行，上车前，在手机上规划好导航路线；上车后，导航自动迁移到车机和车载音箱；下车后，导航自动迁移回手机。如果用户骑车出行，在手机上规划好导航路线，骑行时手表可以接续导航。

- 外卖场景：在手机上点外卖后，可以将订单信息迁移到手表上，随时查看外卖的配送状态。

1.2.1.5 分布式连接能力

分布式连接能力提供了智能终端底层和应用层的连接能力，通过 USB 接口共享终端部分硬件资源和软件能力。开发者基于分布式连接能力，可以开发相应形态的生态产品为消费者提供更丰富的连接体验。分布式连接能力示

意图如下。



分布式连接能力包含底层能力（Connect Service）和应用层能力（AILife Client Service）。

底层能力（Connect Service）涉及如下模块：

- 终端 USB：智能终端侧 USB 模块，可对 USB 生态产品供电，是连接智能终端和生态产品的物理接口。
- 接入管理：智能终端统一对外提供的接口，用于和生态产品进行通信。
- 通信框架：统一管理搜网、信号显示，通过接入管理模块对外提供接口。
- 应用层能力（AILife Client Service）涉及如下模块：
- 智慧生活：生态产品的公共开发平台，能够接入 USB 生态设备并创建接入卡片。

典型应用场景举例：

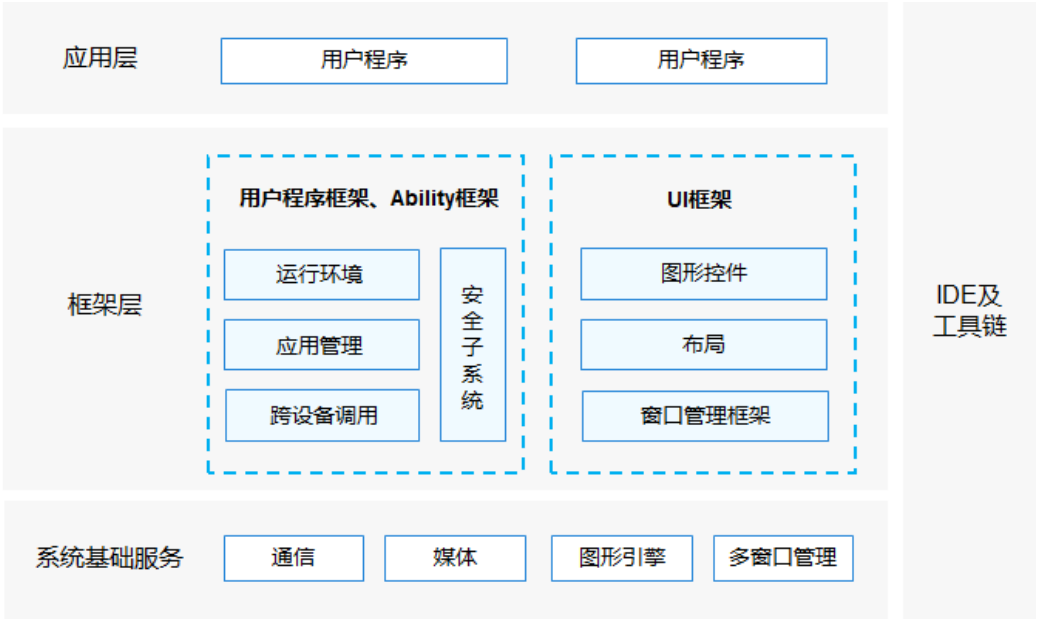
基于分布式连接能力，可以通过开发生态配件拓展智能终端的通信能力：

- USB 模块：生态配件侧 USB 模块，用于和智能终端 USB 建立物理连接。
- 功能模块：生态合作伙伴根据需求开发设备系统和功能。
- 配件插件：生态合作伙伴基于 AILife Client Service 能力开发生态配件功能。

1.2.2 一次开发，多端部署

HarmonyOS 提供了用户程序框架、Ability 框架以及 UI 框架，支持应用开发过程中多终端的业务逻辑和界面逻辑进行复用，能够实现应用的一次开发、多端部署，提升了跨设备应用的开发效率。一次开发、多端部署示意图见图 6。

其中，UI 框架支持使用 ArkTS、JS、Java 语言进行开发，并提供了丰富的多态控件，可以在手机、平板、智能穿戴、智慧屏、车机上显示不同的 UI 效果。采用业界主流设计方式，提供多种响应式布局方案，支持栅格化布局，满足不同屏幕的界面适配能力。



1.2.3 统一 OS，弹性部署

HarmonyOS 通过组件化和小型化等设计方法，支持多种终端设备按需弹性部署，能够适配不同类别的硬件资源和功能需求。支撑通过编译链关系去自动生成组件化的依赖关系，形成组件树依赖图，支撑产品系统的便捷开发，降低硬件设备的开发门槛。

- 支持各组件的选择（组件可有可无）：根据硬件的形态和需求，可以选择所需的组件。
- 支持组件内功能集的配置（组件可大可小）：根据硬件的资源情况和功能需求，可以选择配置组件中的功能集。例如，选择配置图形框架组件中的部分控件。
- 支持组件间依赖的关联（平台可大可小）：根据编译链关系，可以自动生成组件化的依赖关系。例如，选择图形框架组件，将会自动选择依赖的图形引擎组件等。

2 第二章 Harmony OS 快速上手

本节课将学习 HarmonyOS 应用开发领域，我们将逐步学习通过 DeEco Studio 开发工具创建并运行一个 Hello World 的工程。

2.1 下载与安装 DevEco Studio

俗话说，“工欲善其事，必先利其器”，为了进行 HarmonyOS 应用开发，需要完成一些准备工作，确保准备好了必备的 DevEco Studio 开发工具，即 HarmonyOS 的一站式集成开发环境（IDE）。

下面以 window 中安装 DevEco Studio 开发工具为例，介绍如何下载、安装并配置开发环境。

为保证 DevEco Studio 正常运行，建议 Window 电脑配置满足如下要求：

- 操作系统：Windows10 64 位
- 内存：8GB 及以上
- 硬盘：100GB 及以上
- 分辨率：1280*800 像素及以上

进入 DevEco Studio 下载官网：https://developer.harmonyos.com/cn/develop/deveco-studio，单击“立即下载”进入下载页面。

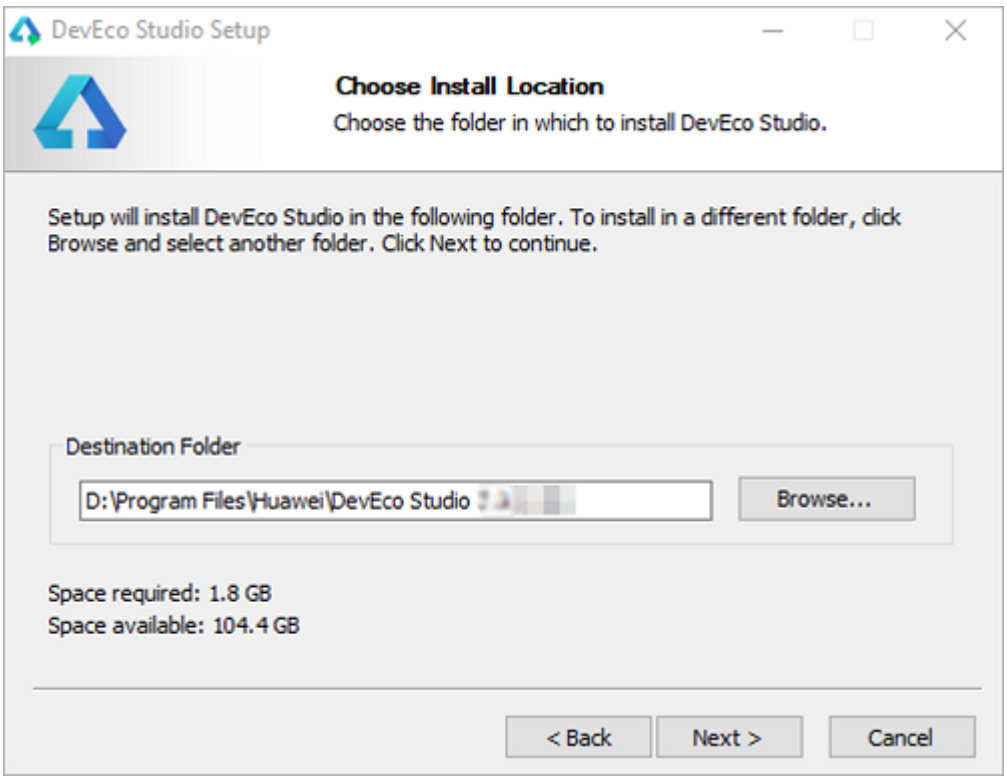
DevEco Studio 提供了 Windows 版本和 Mac 版本选择，可以根据操作系统选择对应的版本进行下载。

DevEco Studio 3.1.1 Release

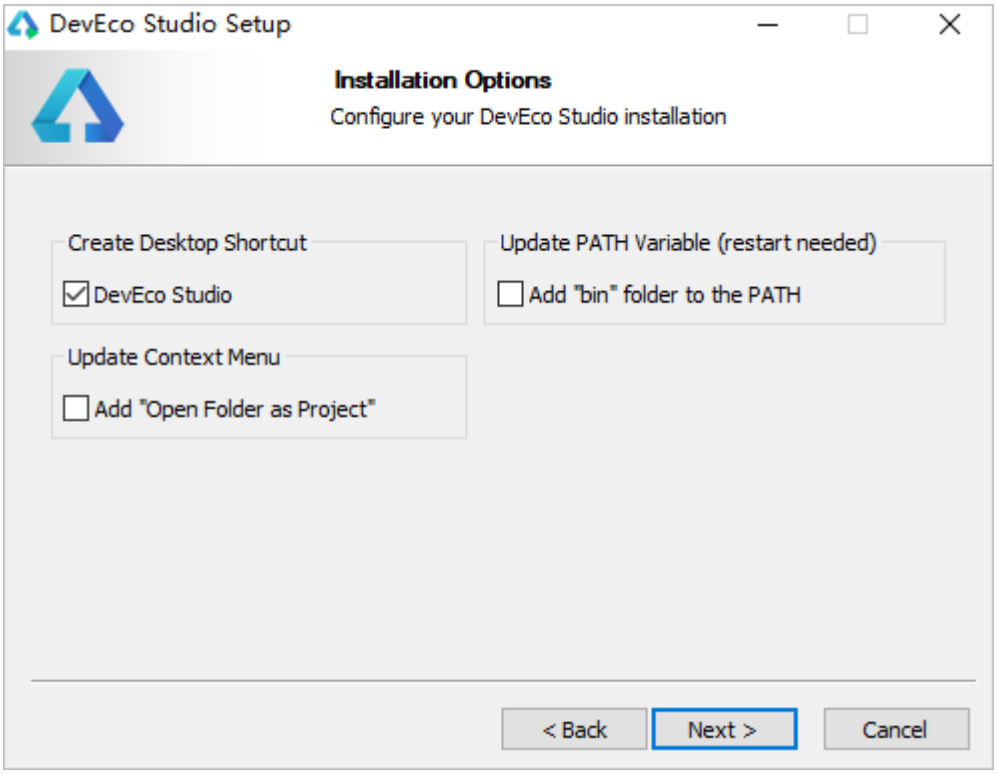
platform	DevEco Studio Package	Size	SHA-256 checksum	Download
Windows(64-bit)	devecostudio-windows-3.1.0.501.zip	843.6M	fbe79d92017d642ee91b2471b36c3e22ff3c186a0df36f3ae683129cfd445d9c	
Mac(X86)	devecostudio-mac-3.1.0.501.zip	942.9M	1a380b8b4a172b0f00af476b3bdcd83ee2dab24937c00b72d20d9121db99f5b7	
Mac(ARM)	devecostudio-mac-arm-3.1.0.501.zip	934.8M	f3e77ba60e596c9e49cd5fc3ab67f3f944efd235f2fa7b298b501abcfc668f04	

该版本适用HarmonyOS和OpenHarmony应用及服务开发，您可体验HarmonyOS 3.1 版本及以上的开发能力，在使用过程中如遇到问题请积极反馈，我们将在后续版本中进行优化，点击查看版本说明。

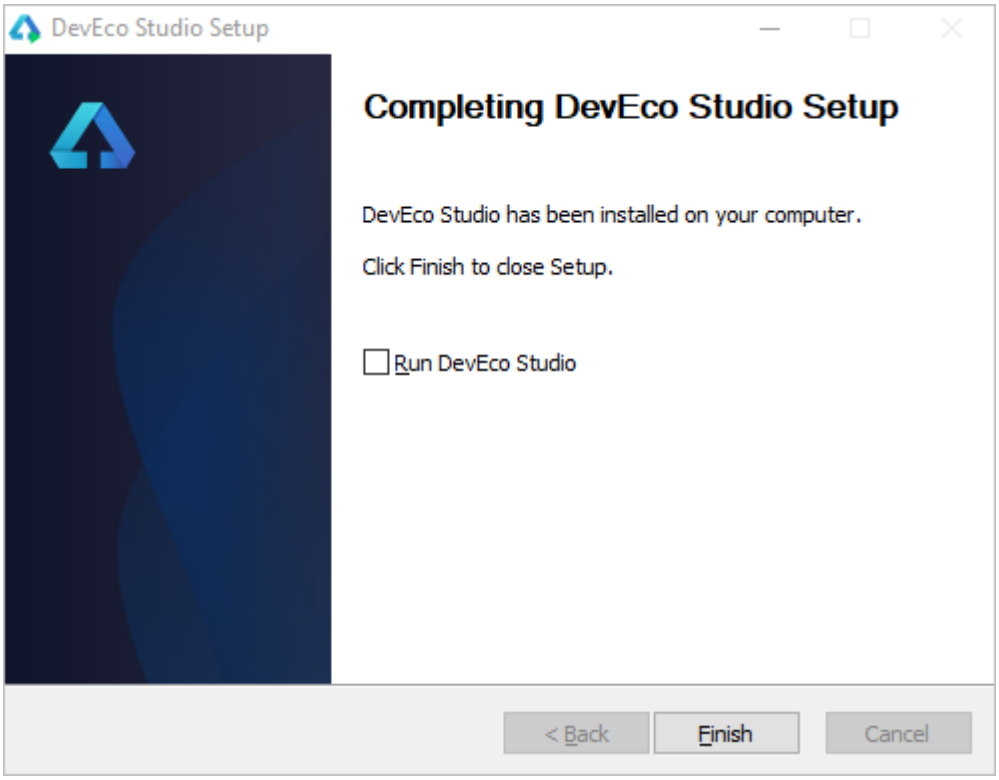
下载完成后，解压下载的压缩包并进入到其中，双击下载的“deveco-studio-xxxx.exe”，进入 DevEco Studio 安装向导，在如下界面选择安装路径，默认安装于“C:\Program Files”下，也可以单击“Browse...”指定其他安装路径，然后单击“Next”。



如下安装选项界面勾选 DevEco Studio 后，单击“Next”，直至安装完成。

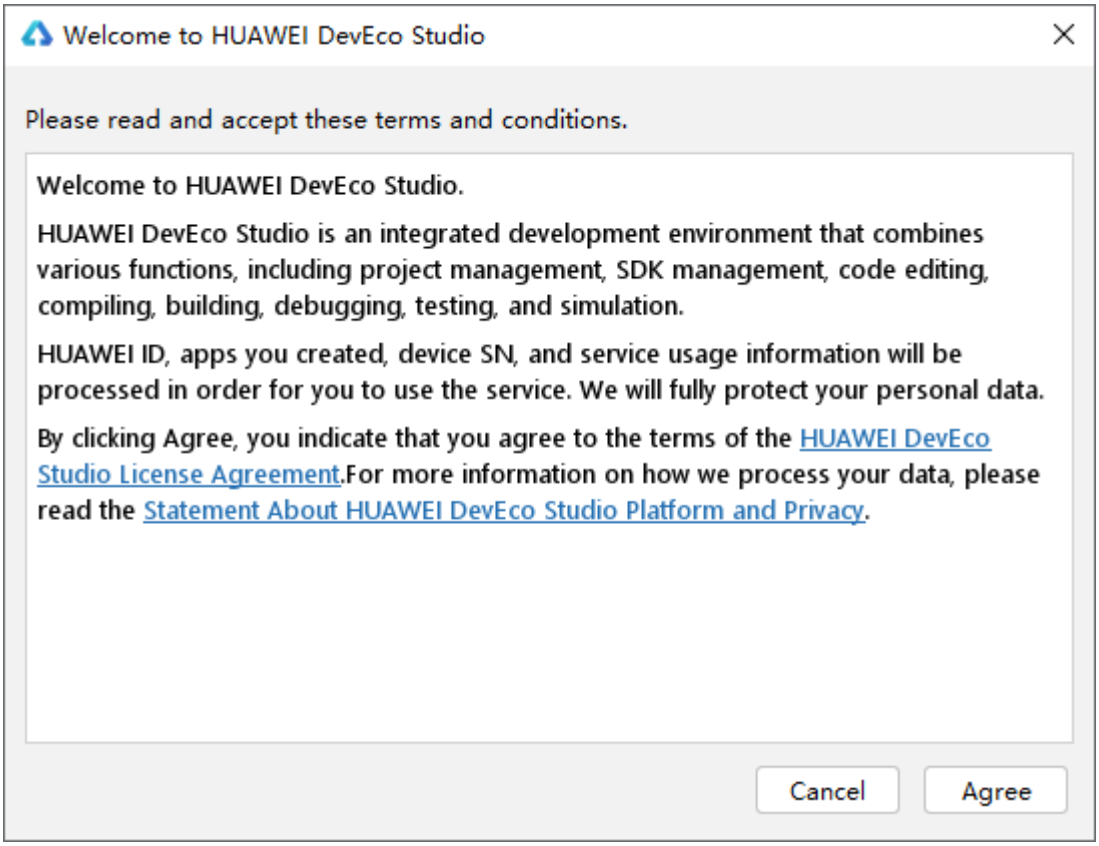


安装完成后，单击“Finish”完成安装。

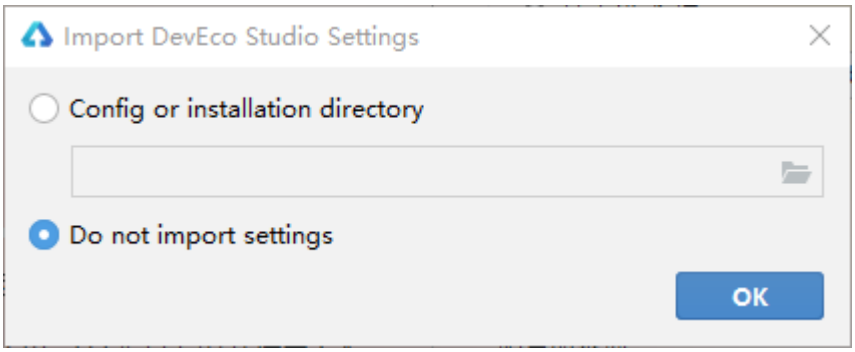


2.2 配置环境

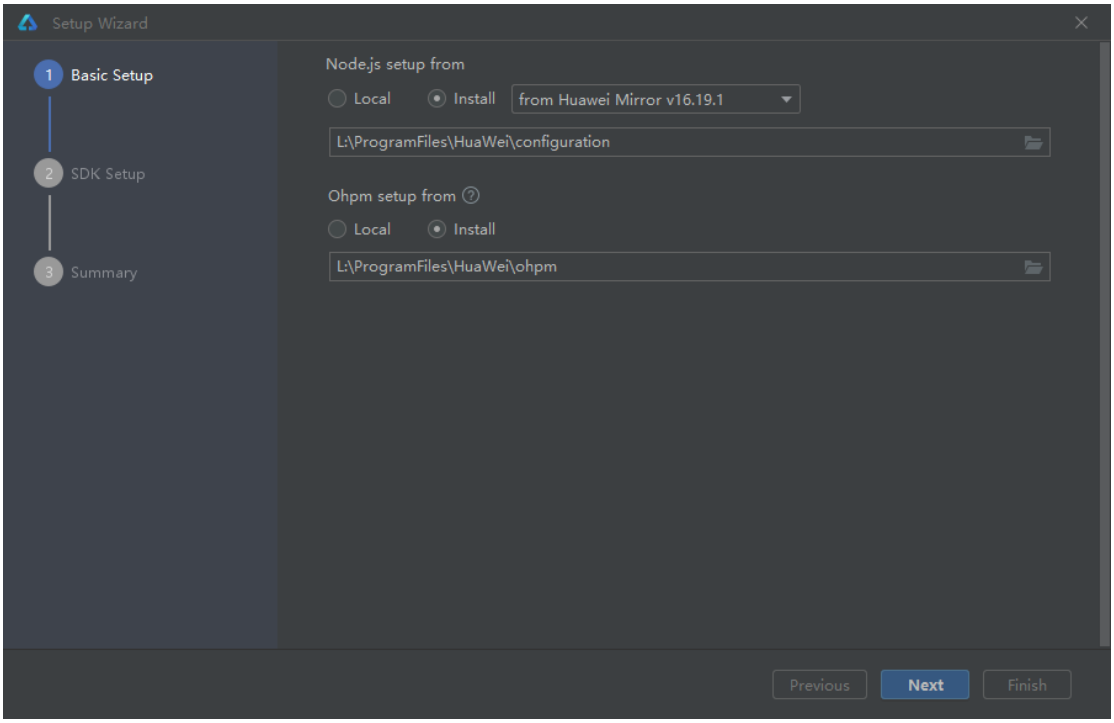
双击已安装的 DevEco Studio 快捷方式进入配置页面，IDE 会进入配置向导，选择 Agree，同意相应的条款，进入配置页。



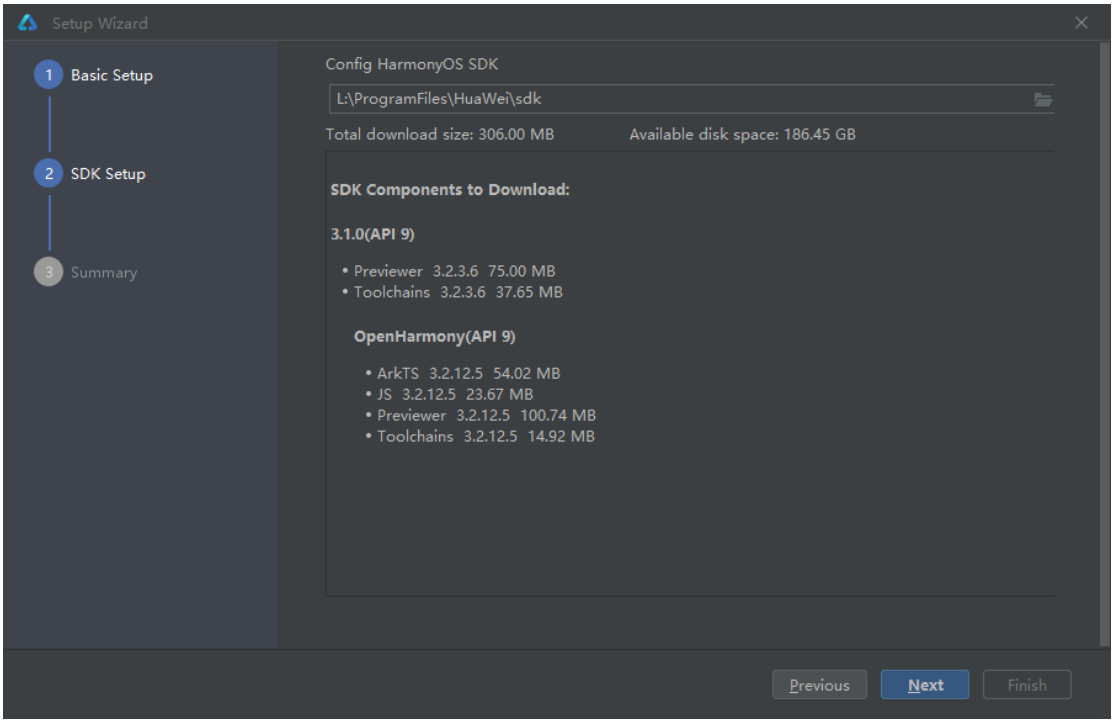
点击“OK”跳过导入设置：



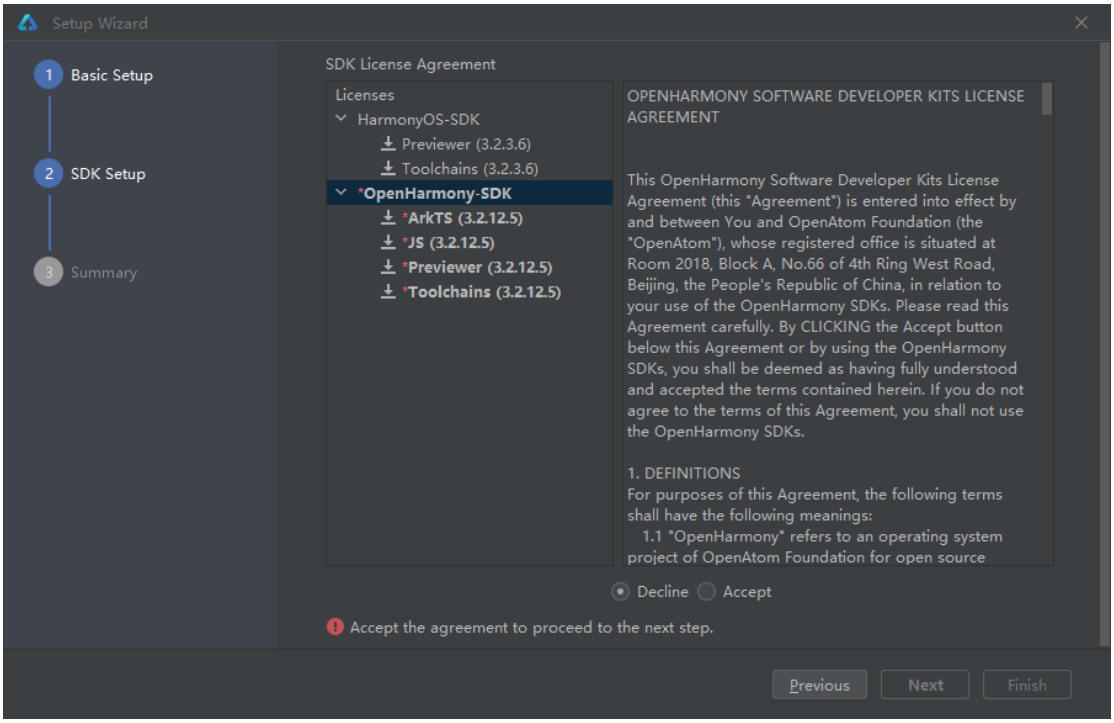
进入 DevEco Studio 配置页面，首先需要进行基础配置，包括 Node.js 与 Ohpm 的安装路径设置，选择从华为镜像下载至合适的路径。



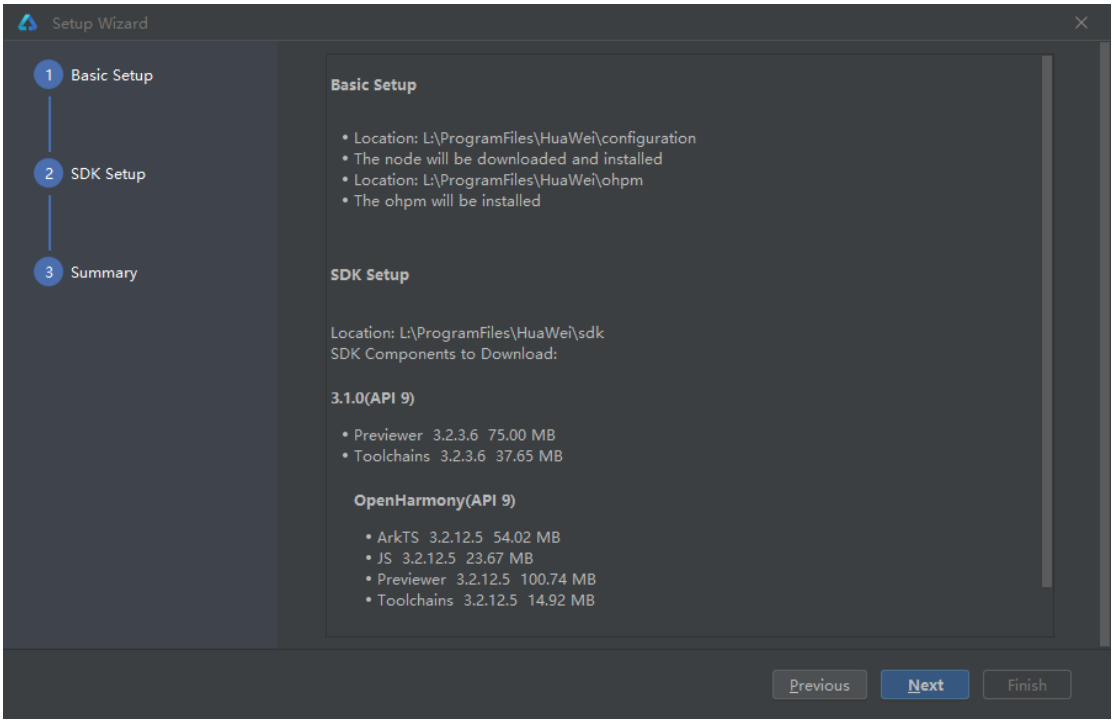
单击'Next'进入 SDK 配置，设置为合适的路径，



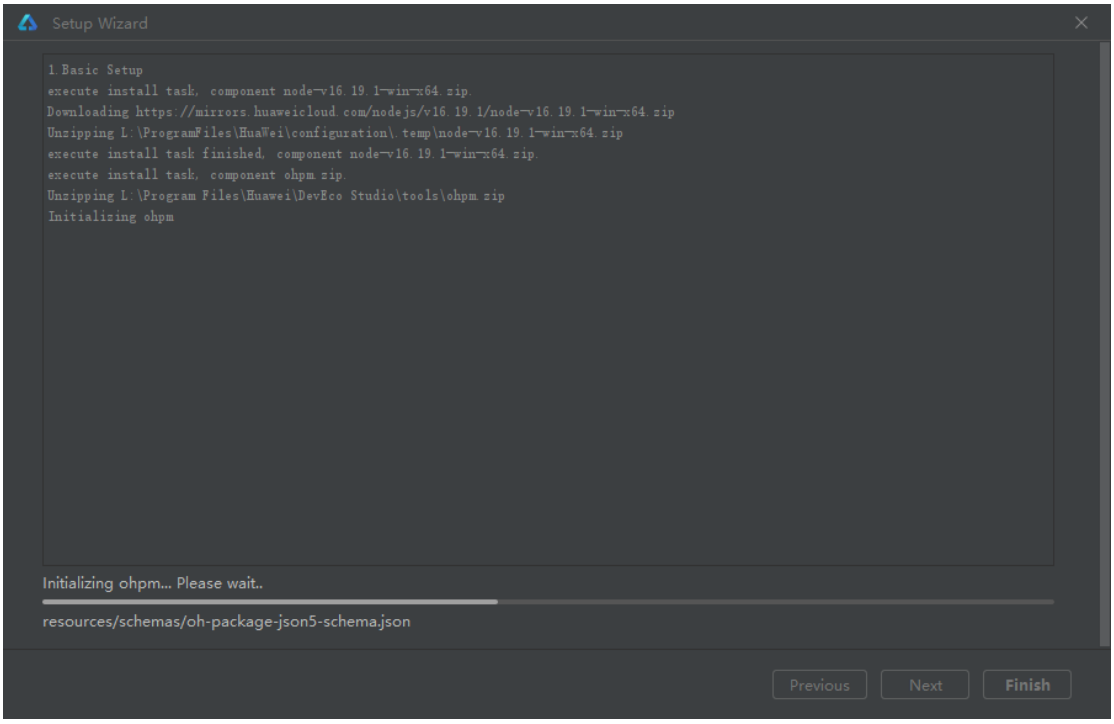
点击'Next'后会显示'SDK License Agreement'，阅读相关协议后，勾选'Accept'。

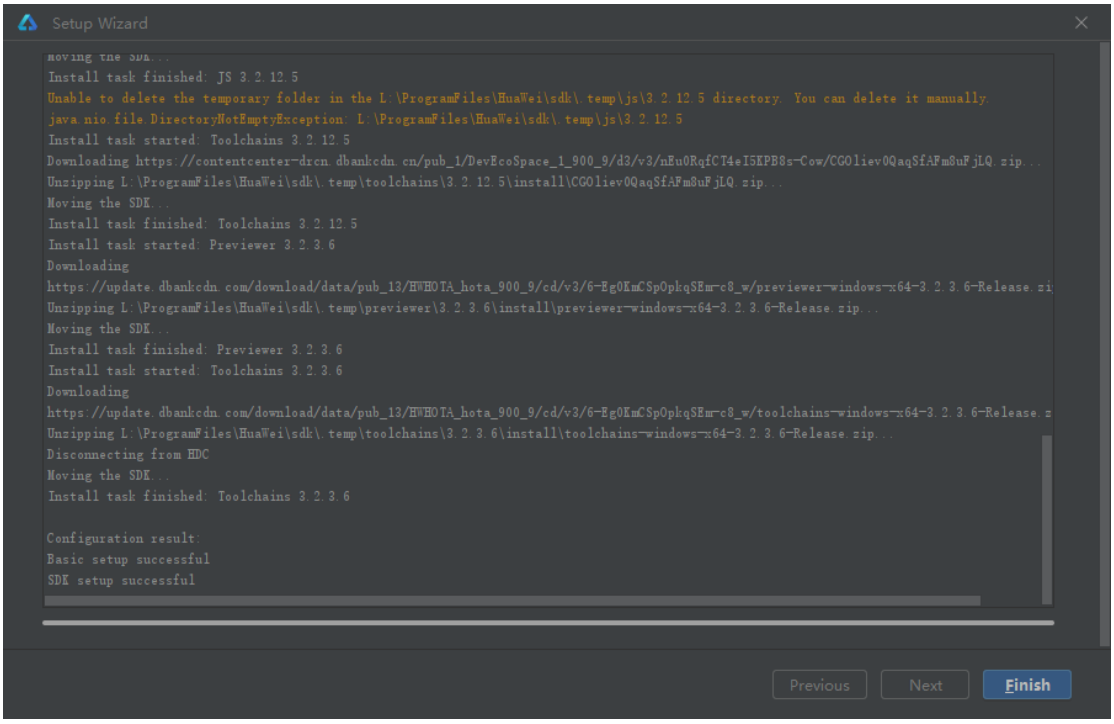


单击‘Next’进入配置预览页，在这里进行配置项的确认。

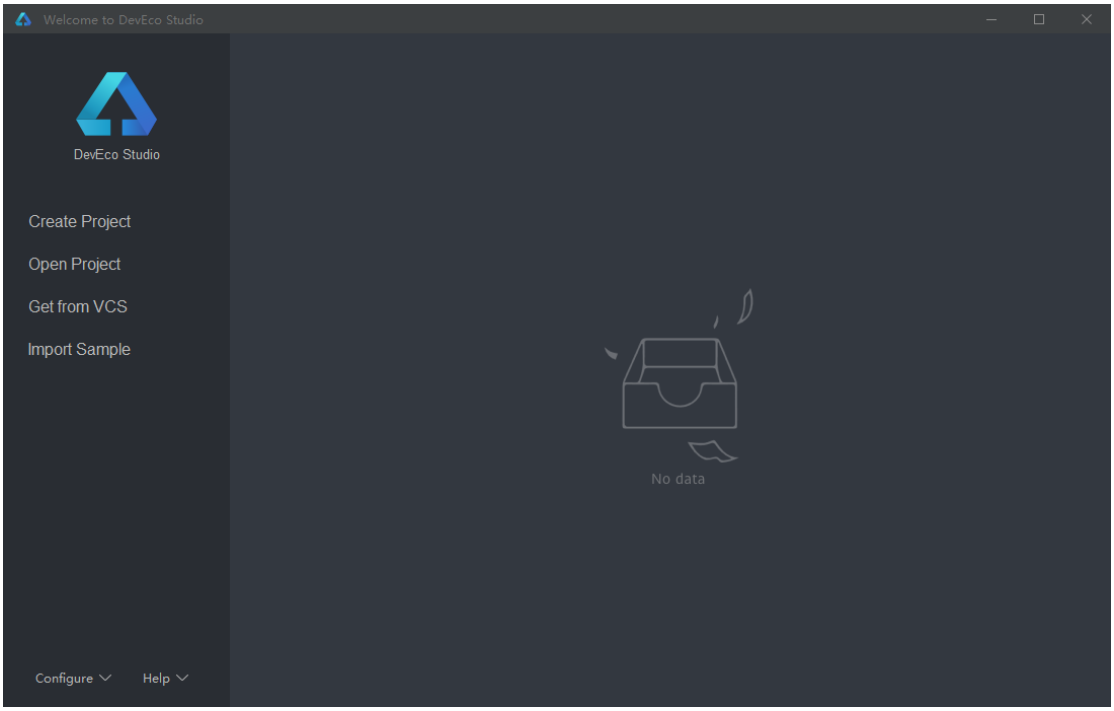


确认完成后，单击'Next'，进入下一步。





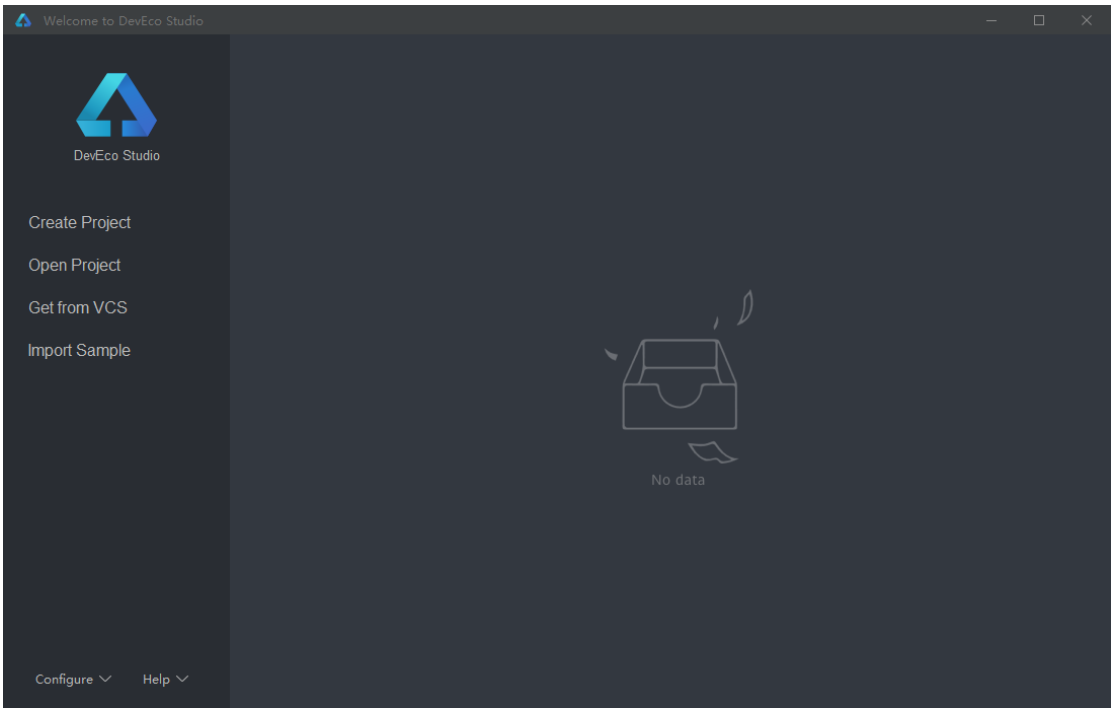
等待配置自动下载完成，完成后，单击'Finish'，IDE 会进入欢迎页，我们也就成功配置好了开发环境。



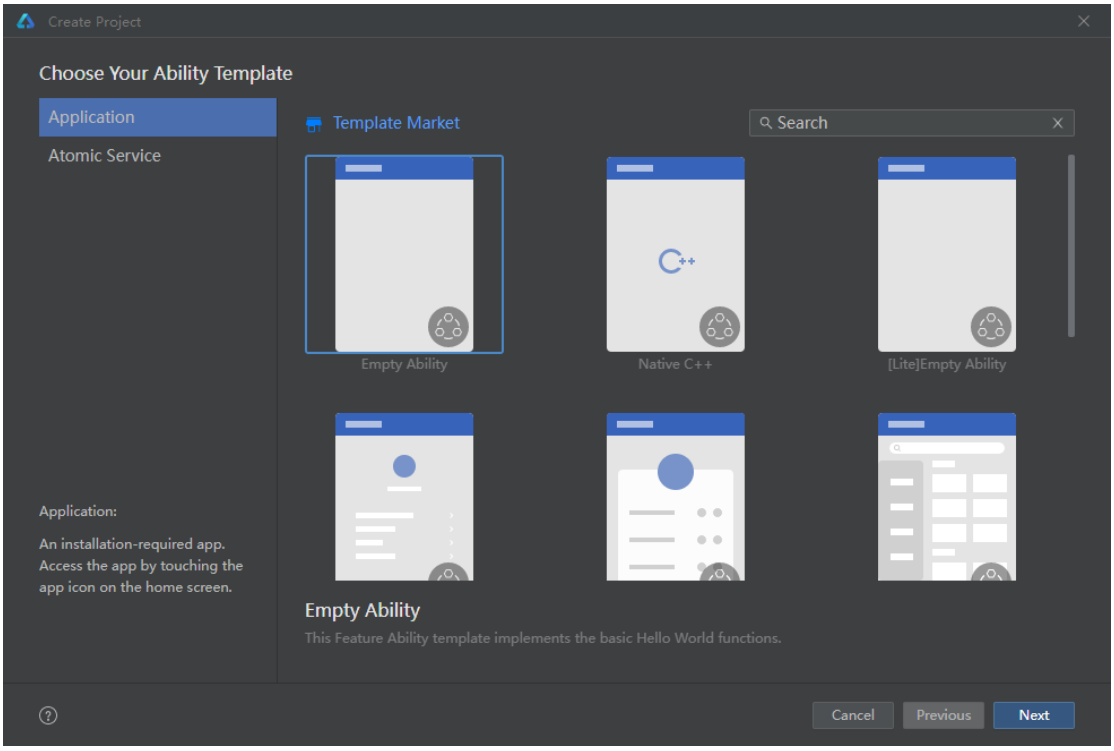
准备工作完成后，接下来将进入 DevEco Studio 进行工程创建和运行。

2.3 创建项目

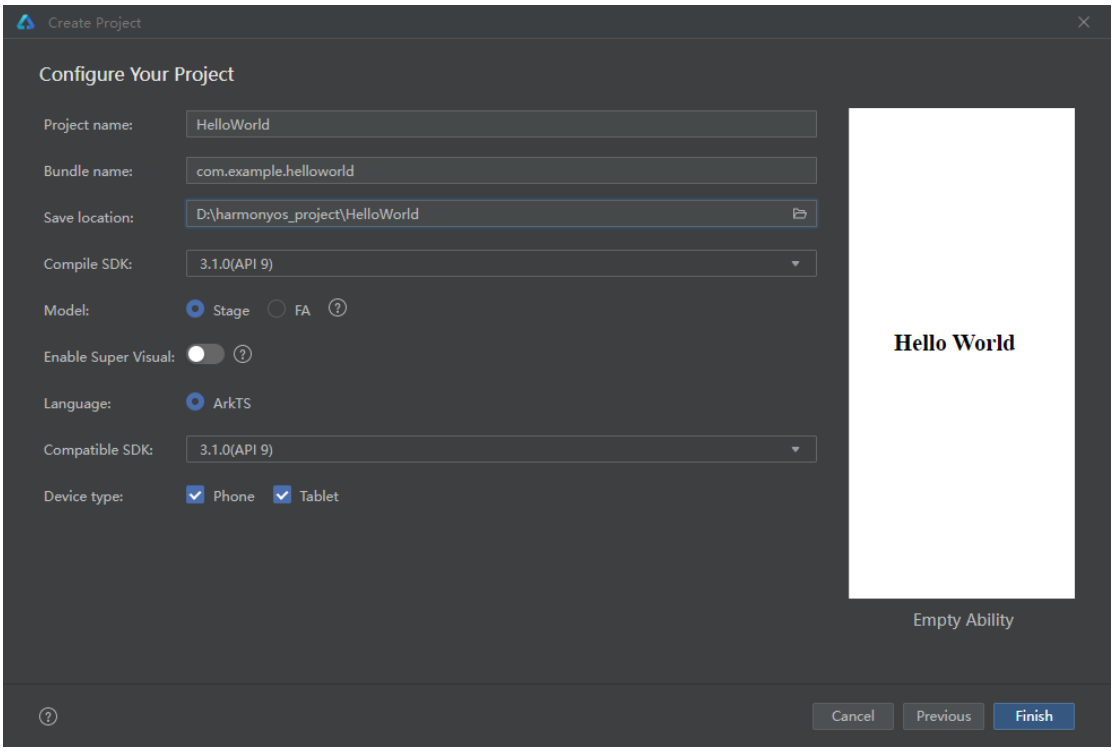
如果你是首次打开 DevEco Studio，那么首先会进入欢迎页。



在欢迎页中单击 Create Project，进入项目创建页面。



选择‘Application’，然后选择‘Empty Ability’，单击‘Next’进入工程配置页。



配置页中，详细信息如下：

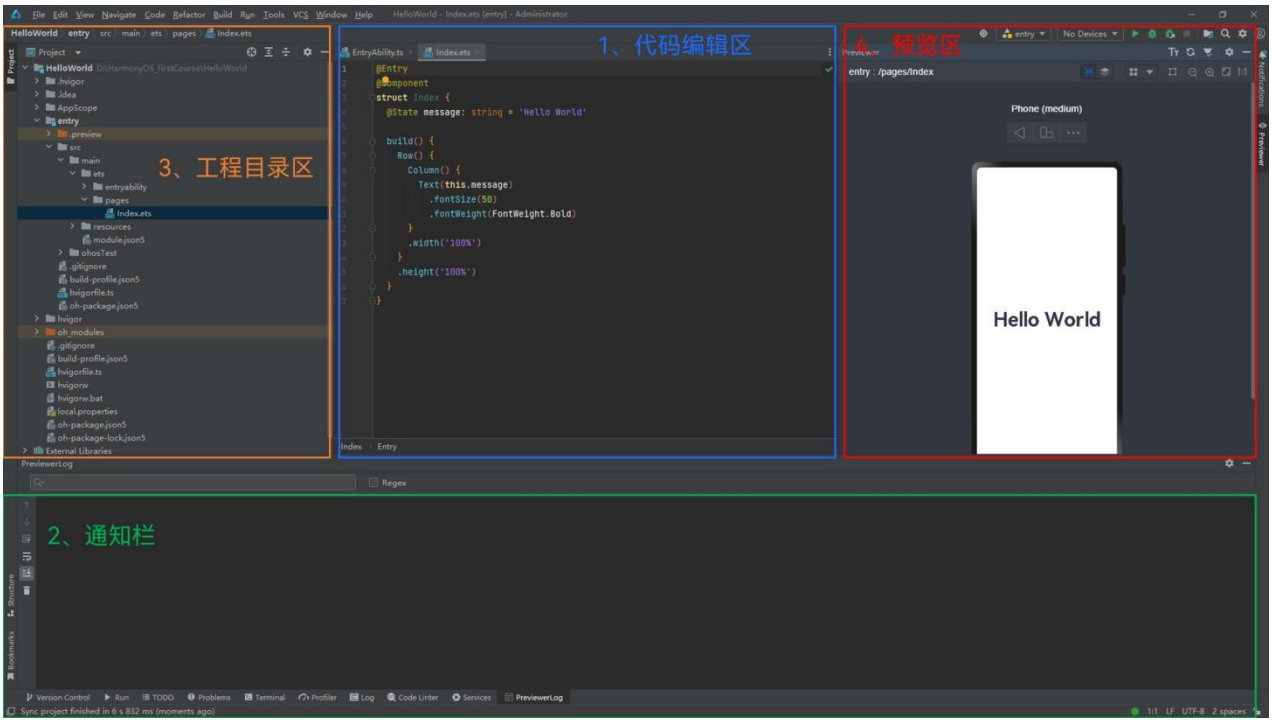
- Project name 是开发者可以自行设置的项目名称，这里根据自己选择修改为自己项目名称。
- Bundle name 是包名称，默认情况下应用 ID 也会使用该名称，应用发布时对应的 ID 需要保持一致。
- Save location 为工程保存路径，建议用户自行设置相应位置。

- Compile SDK 是编译的 API 版本，这里默认选择 API9。
- Model 选择 Stage 模型，其他保持默认即可。

然后单击“Finish”完成工程创建，等待工程同步完成。

2.4 认识 DevEco Studio 界面

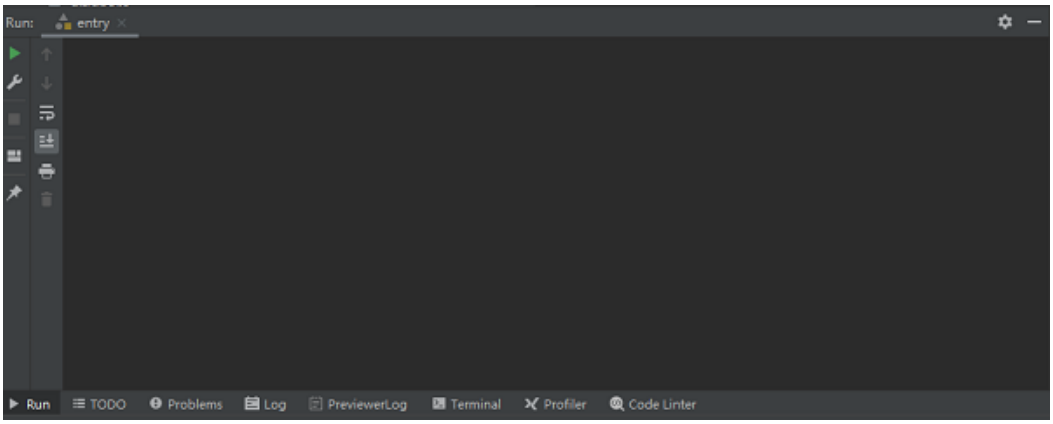
进入 IDE 后，我们首先了解一下基础的界面。整个 IDE 的界面大致上可以分为四个部分，分别是代码编辑区、通知栏、工程目录区以及预览区。



2.4.1 代码编辑区

中间的是代码编辑区，你可以在这里修改你的代码，以及切换显示的文件。通过按住 Ctrl 加鼠标滚轮，可以实现界面的放大与缩小。

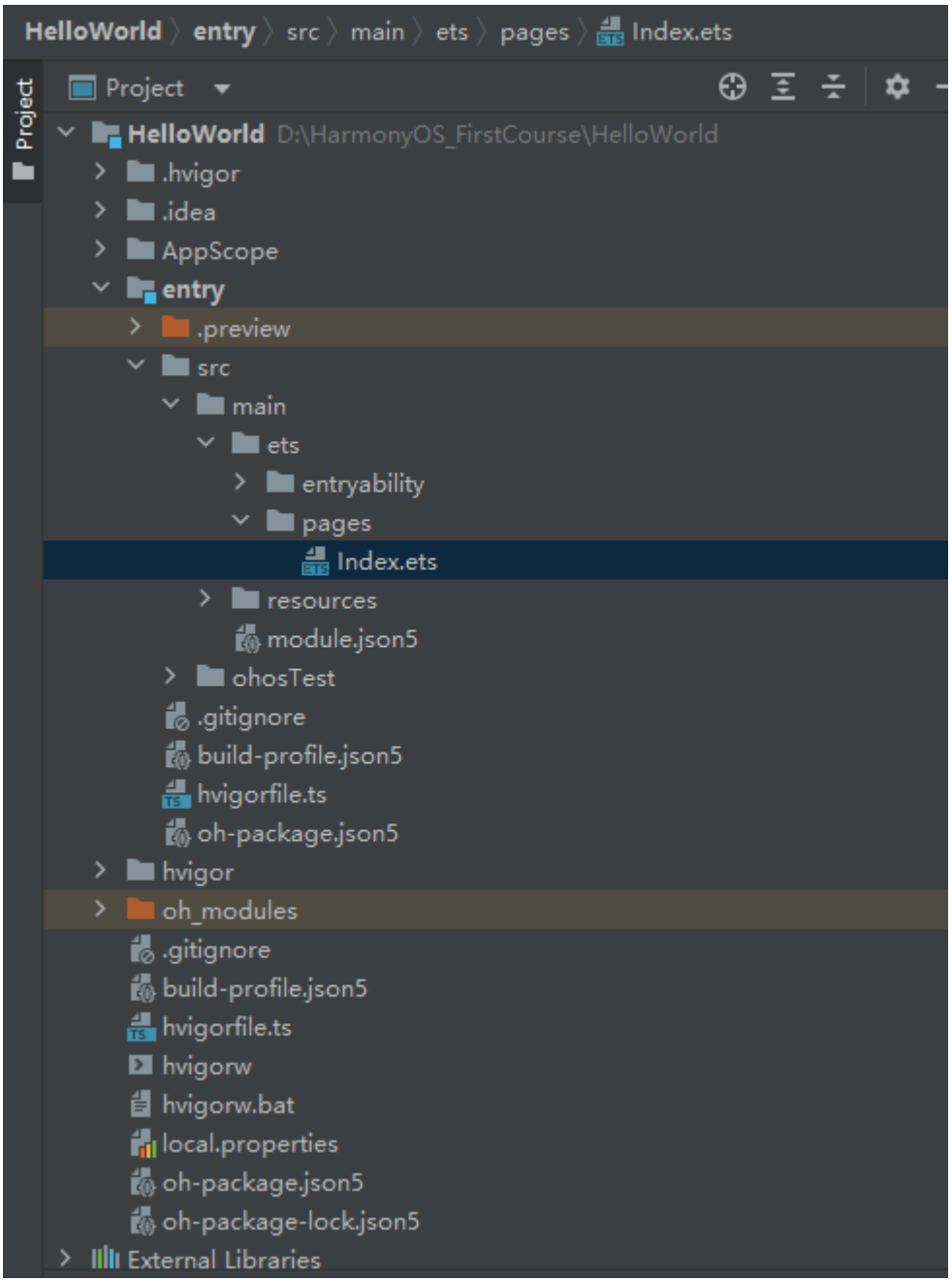
2.4.2 通知栏



在编辑器底部有一行工具栏，主要介绍常用信息栏，其中 Run 是项目运行时的信息栏，Problems 是当前工程错误与提醒信息栏，Terminal 是命令行终端，在这里执行命令行操作，PreviewerLog 是预览器日志输出栏，Log 是模拟器和真机运行时的日志输出栏。在后续使用中会陆续接触。

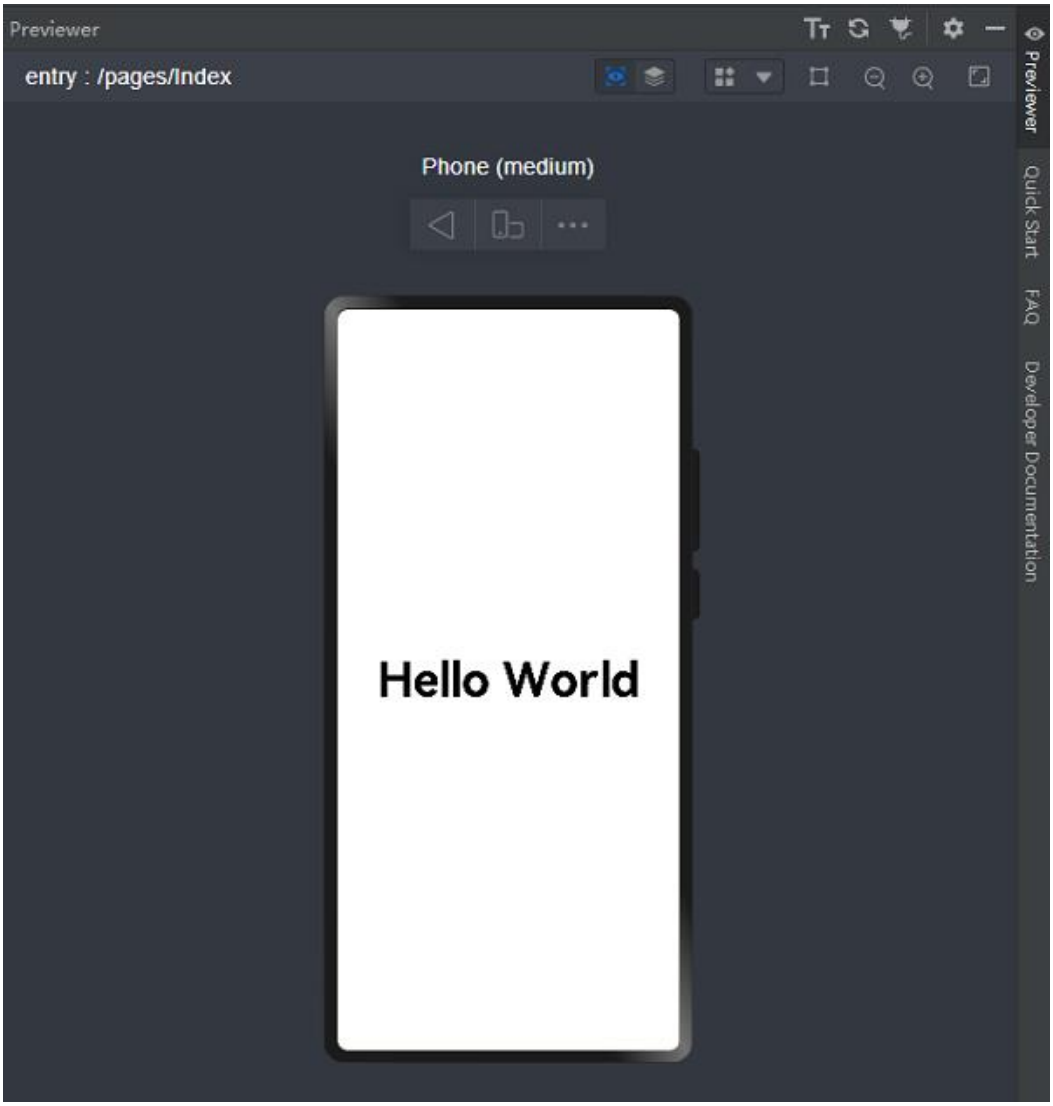
2.4.3 工程目录区

左侧为工程目录区，后续章节会详细介绍。

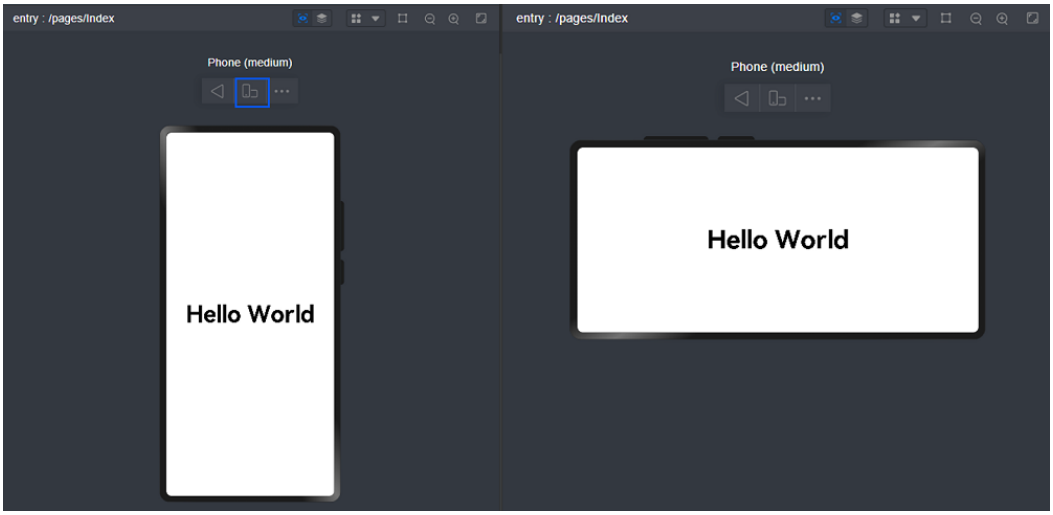


2.4.4 预览区

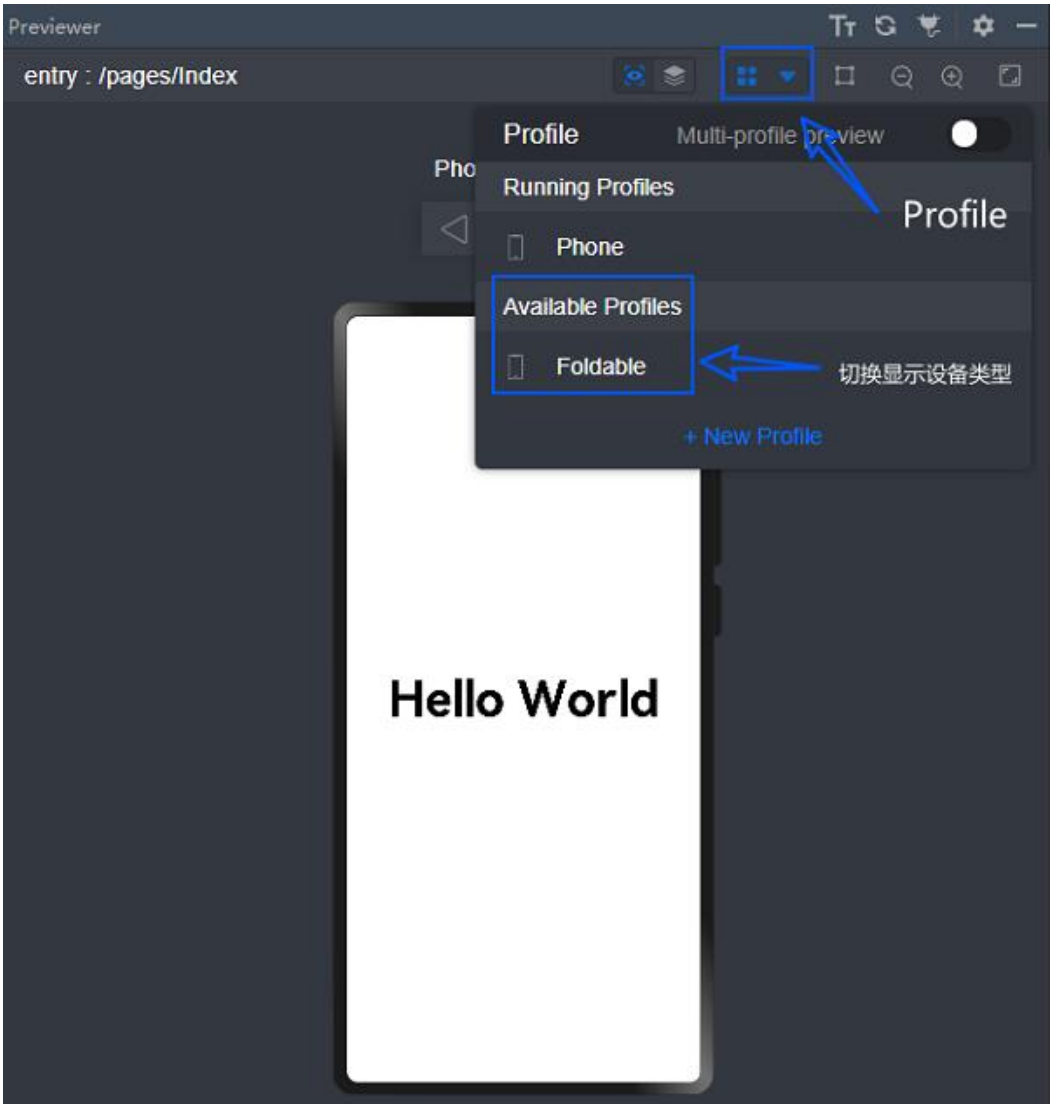
单击右上角 Previewer，可以预览相应的文件 UI 展示效果。



预览器提供了一些基本功能，包括旋转屏幕，切换显示设备及多设备预览等。单击旋转按钮，可以切换竖屏和横屏显示的效果。

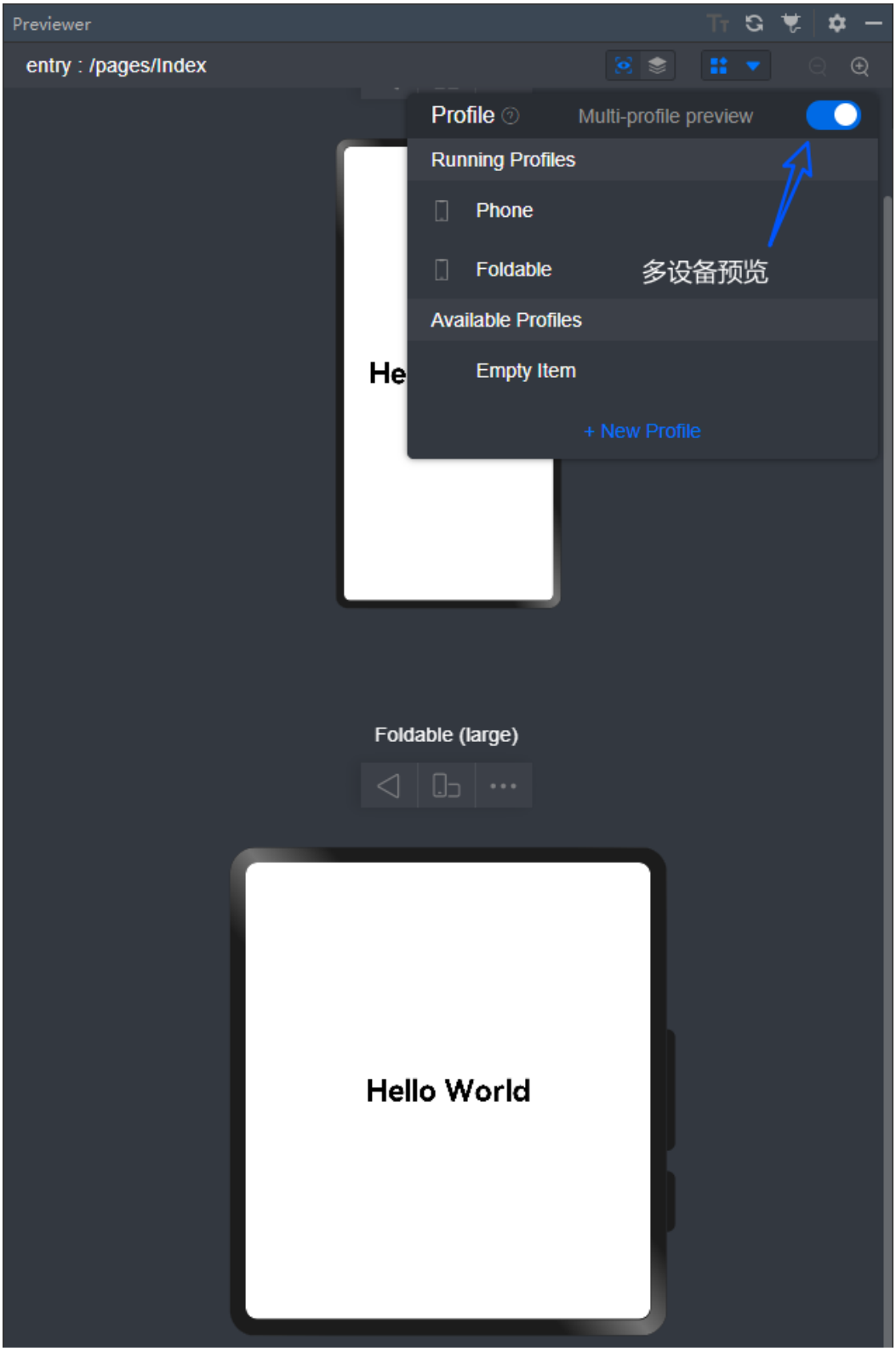


也可以单击如下列表按钮，切换显示的设备类型。弹出框内会显示 Available Profiles，即可用的设备类型。

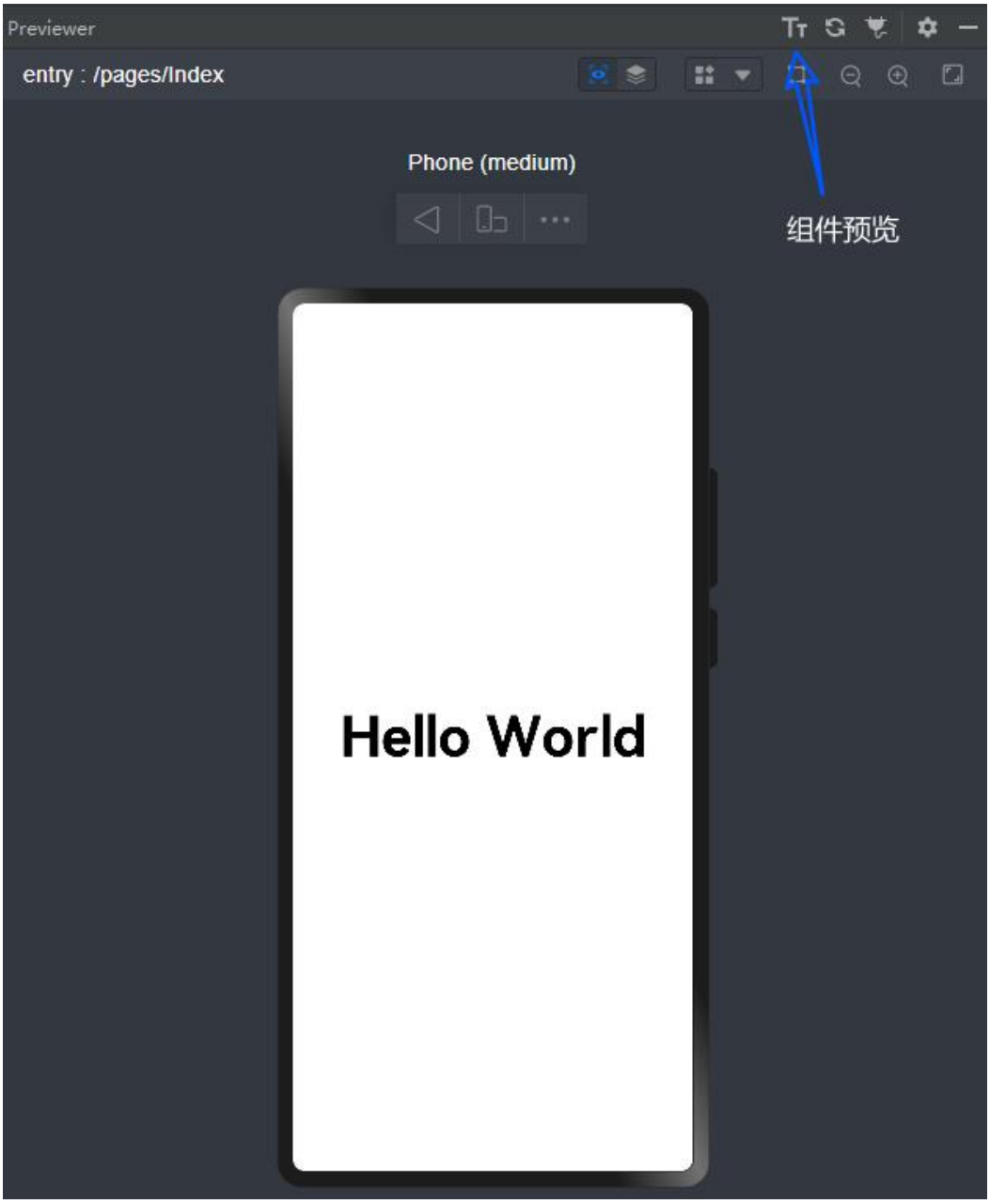


如单击 Foldable 切换设备，也可以单击旋转按钮切换 Foldable 的横竖屏显示模式。

打开 Muti-profile preview 开关，可以实现多个尺寸设备的实时预览。

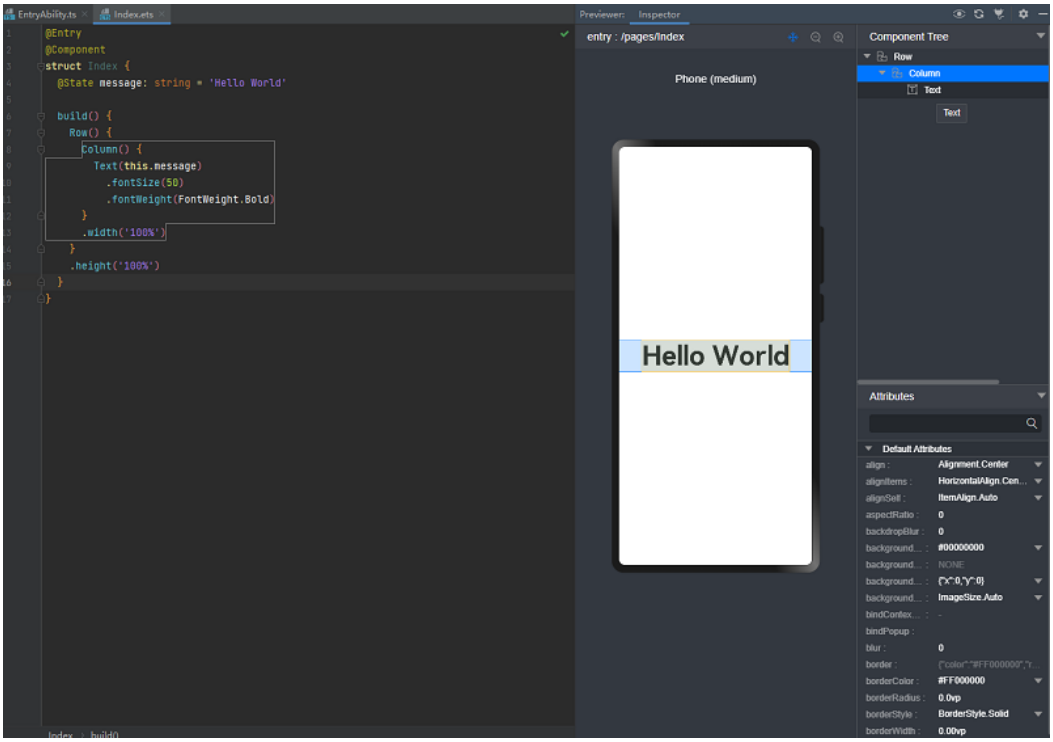


单击预览器右上角组件预览按钮，可以进入组件预览界面。



组件预览模式可以预览当前组件对应的代码块。

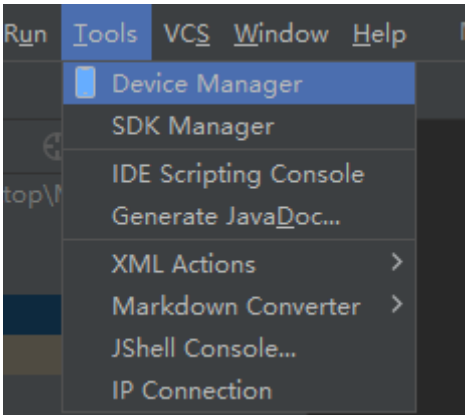
点击相应组件，代码文件中会框选对应的组件代码部分，下方则对应当前组件的基本属性。



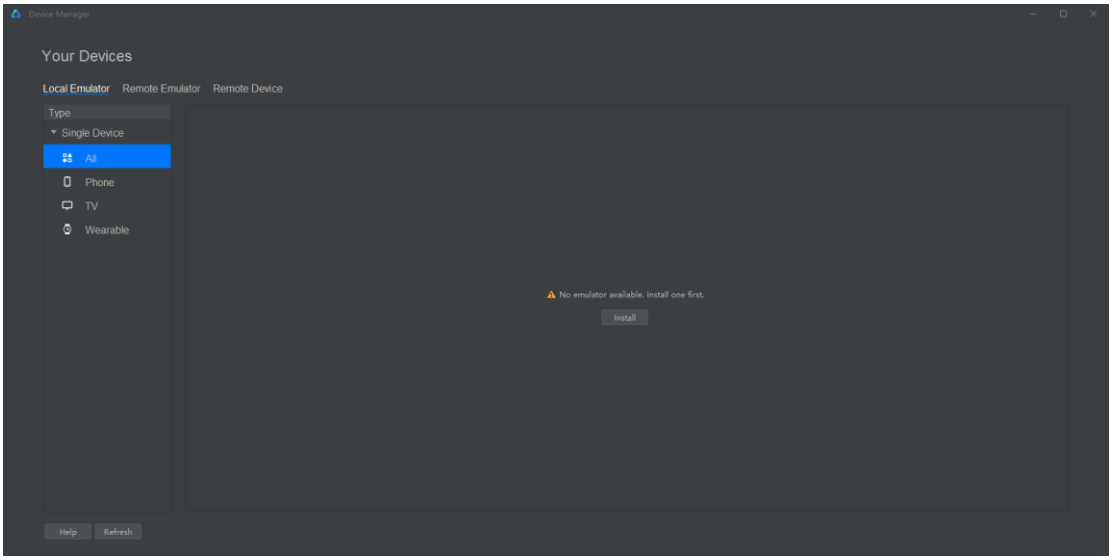
2.5 运行 Hello World

IDE 提供了本地模拟器供开发者使用，我们首先需要下载安装本地模拟器，然后进行运行工程。

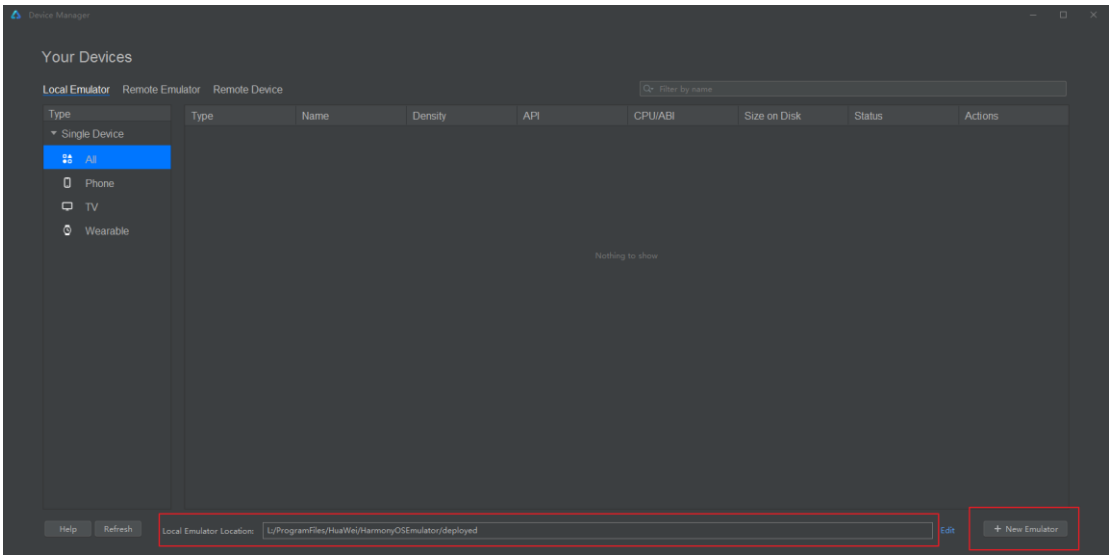
1) 单击顶部工具栏 Tools>Device Manager。



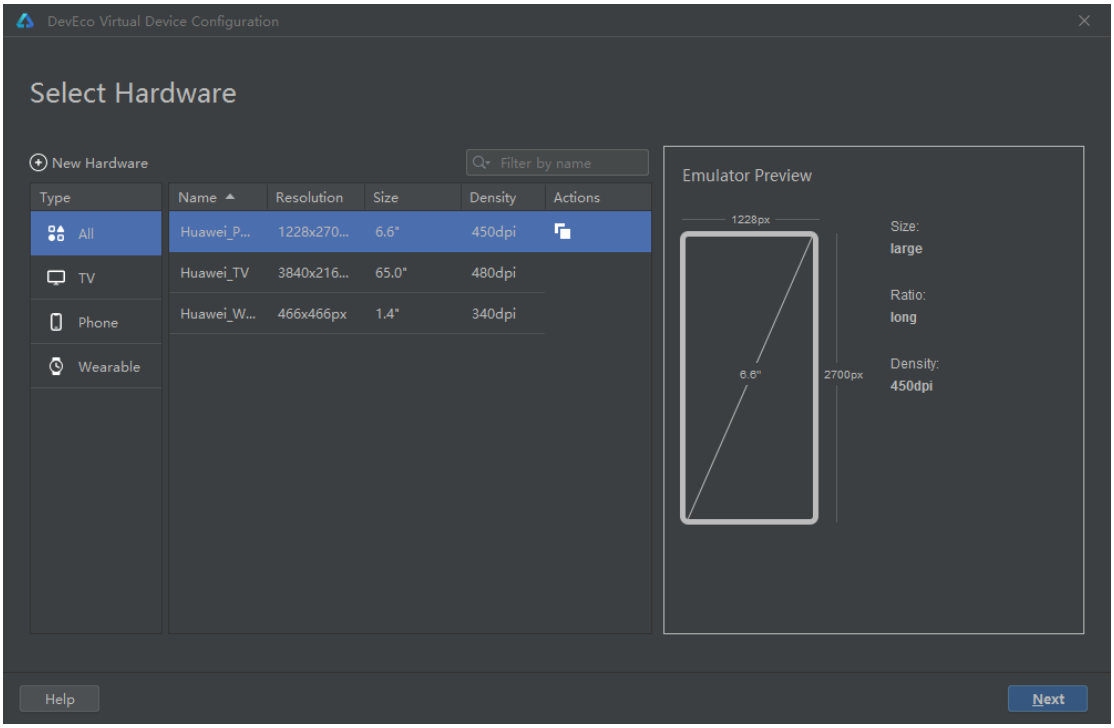
2) 选择 Local Emulator，安装模拟器



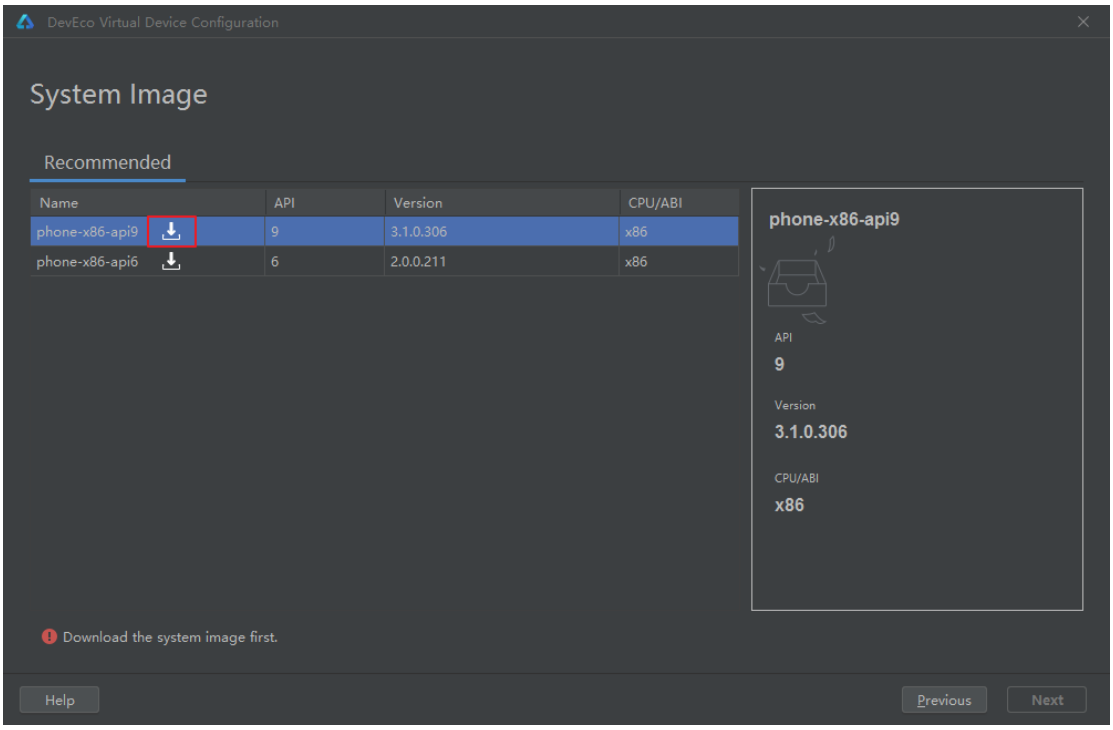
设置合适的 Local Emulator Location 存储地址，然后单击’ +New Emulator’ 。

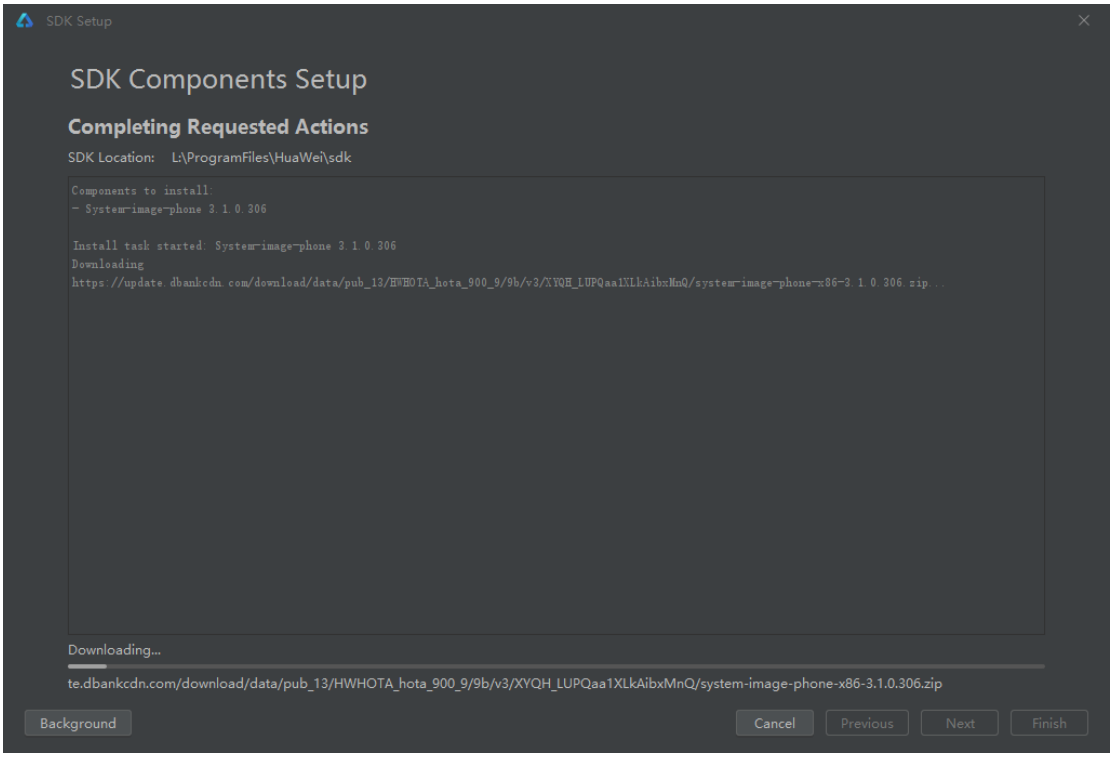


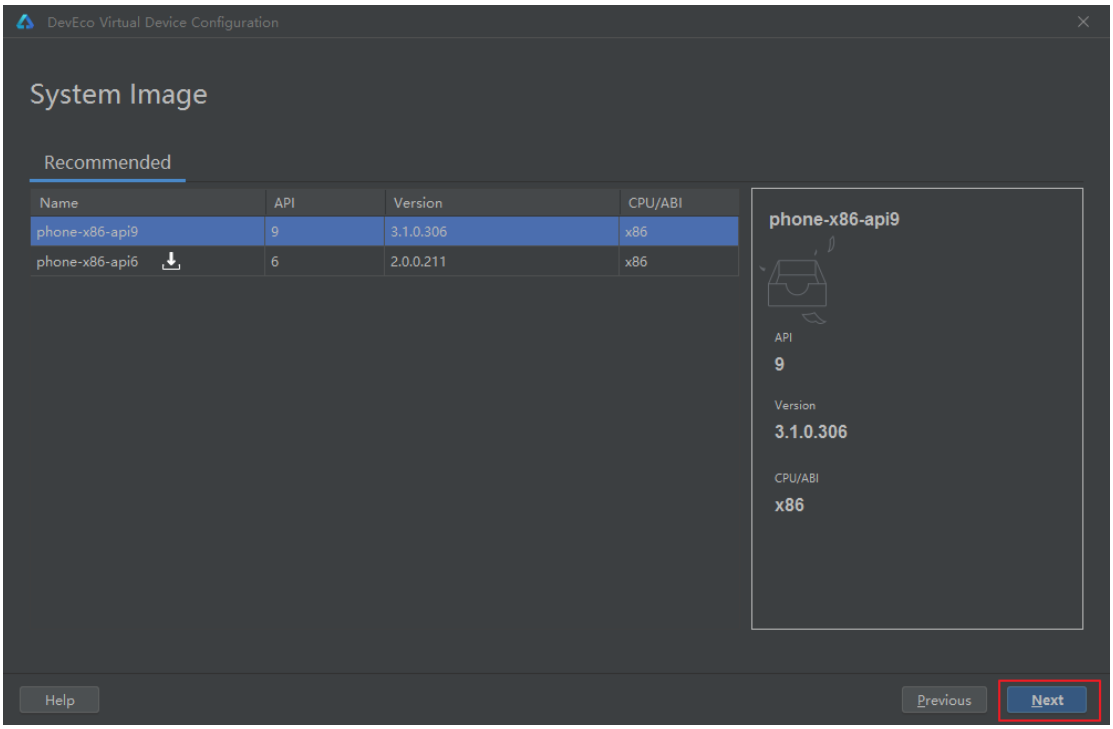
选择 Huawei_Phone 手机模拟器，单击'Next'，进入模拟器系统下载页。



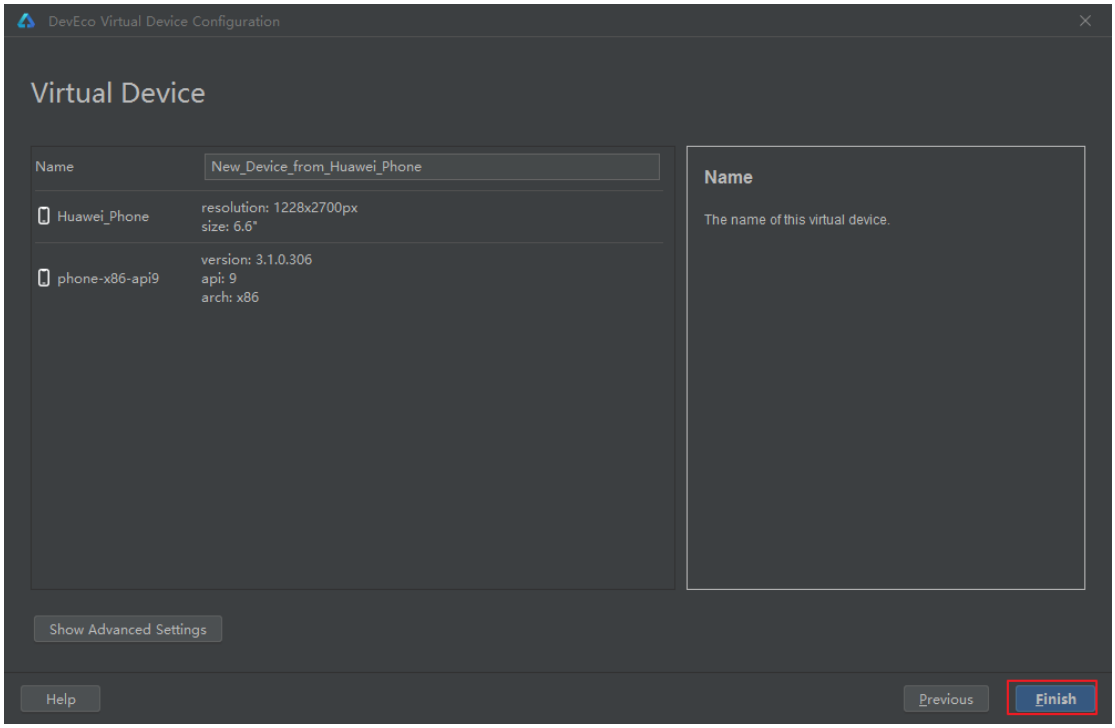
选择下载 api9 的系统镜像，然后单击'Next' ，等待下载完成。



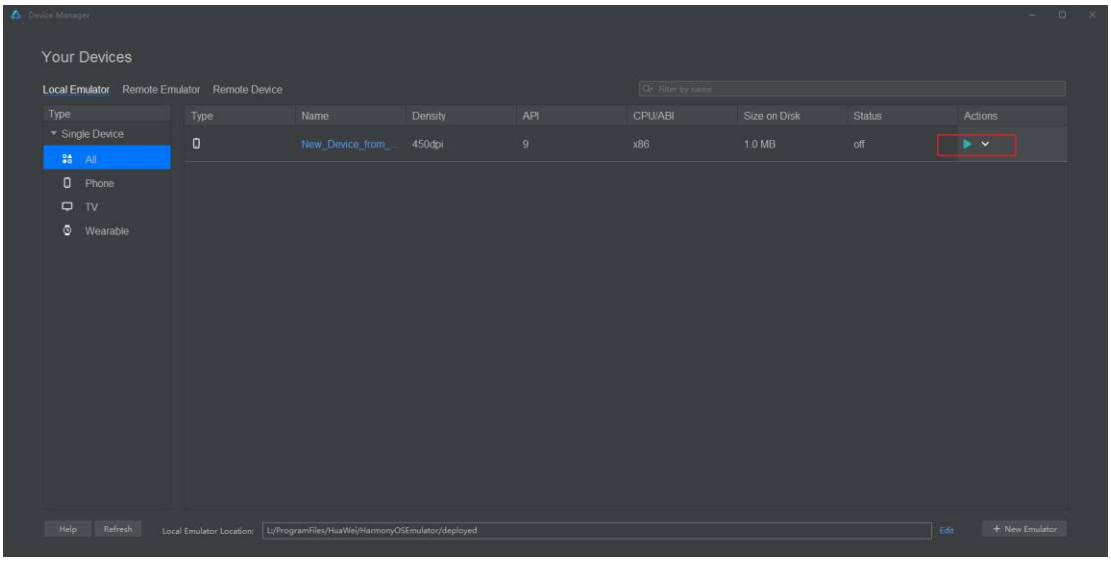




下载完成后，进行创建相应的手机模拟器，单击 Finish 完成创建。

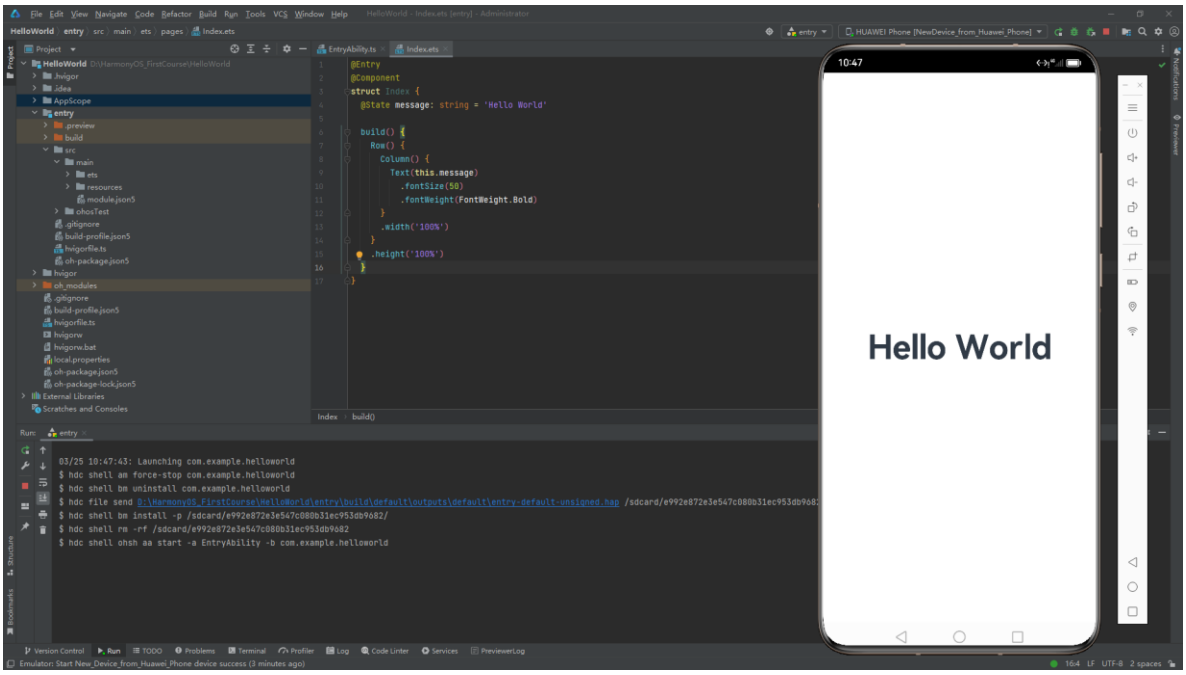


下载完成后，在 Local Emulator 页面中会出现创建的手机模拟器，点击 Actions 按钮，就能够启动模拟器。



模拟器启动后，点击上方启动按钮，将 Hello World 工程运行到模拟器上。

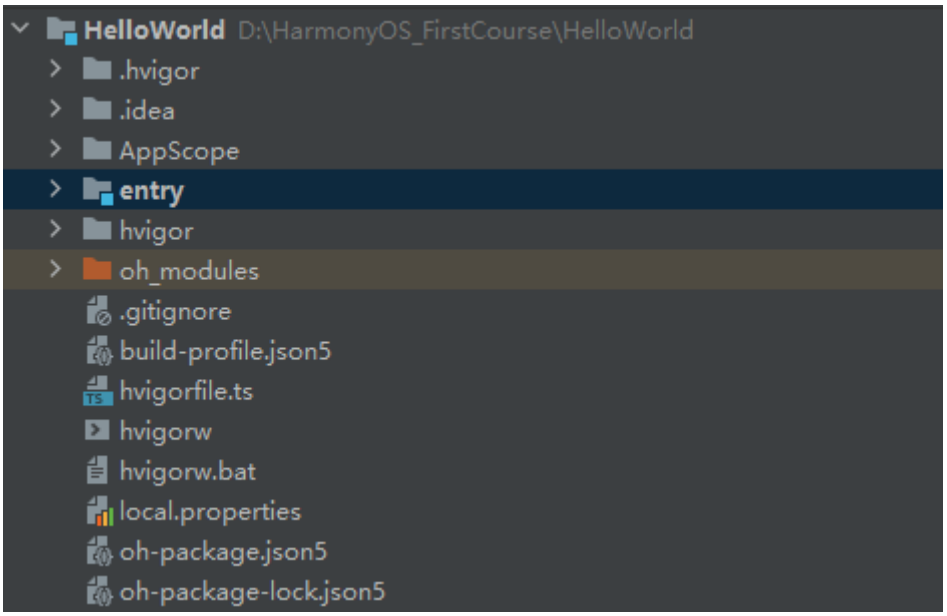
IDE 构建完成后，即可在模拟器上看到运行效果，我们也就完成了 Hello World 工程在模拟器上的运行。



2.6 了解基本工程目录

2.6.1 工程级目录

工程的目录结构如下。



其中详细如下：

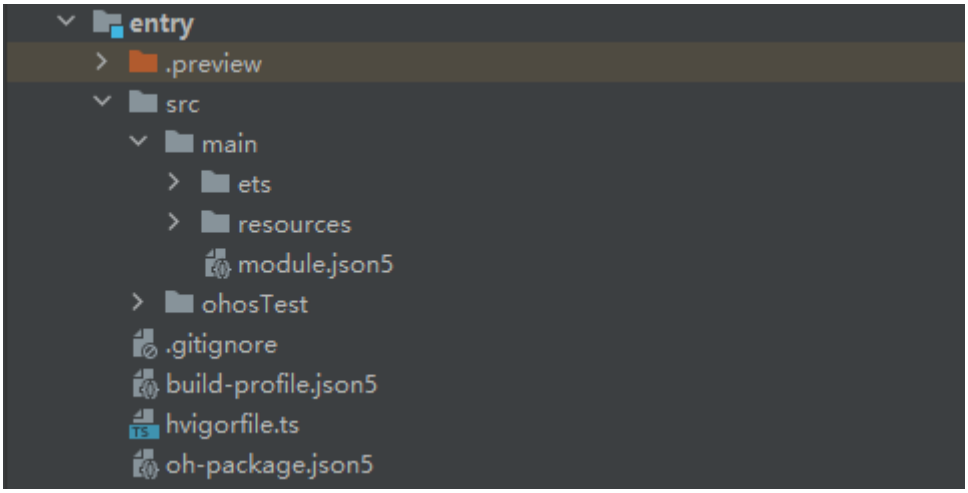
- AppScope 中存放应用全局所需要的资源文件。

- entry 是应用的主模块，存放 HarmonyOS 应用的代码、资源等。
- oh_modules 是工程的依赖包，存放工程依赖的源文件。
- build-profile.json5 是工程级配置信息，包括签名、产品配置等。
- hvigorfile.ts 是工程级编译构建任务脚本，hvigor 是基于任务管理机制实现的一款全新的自动化构建工具，主要提供任务注册编排，工程模型管理、配置管理等核心能力。
- oh-package.json5 是工程级依赖配置文件，用于记录引入包的配置信息。

在 AppScope，其中有 resources 文件夹和配置文件 app.json5。AppScope>resources>base 中包含 element 和 media 两个文件夹，

- 其中 element 文件夹主要存放公共的字符串、布局文件等资源。
- media 存放全局公共的多媒体资源文件。

2.6.2 模块级目录



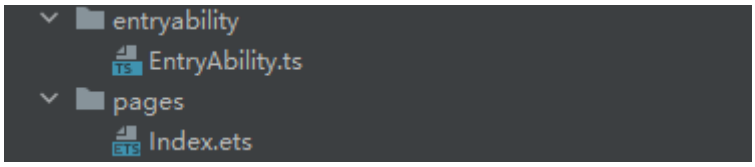
entry>src 目录中主要包含总的 main 文件夹，单元测试目录 ohosTest，以及模块级的配置文件。

- main 文件夹中，ets 文件夹用于存放 ets 代码，resources 文件存放模块内的多媒体及布局文件等，module.json5 文件为模块的配置文件。
- ohosTest 是单元测试目录。

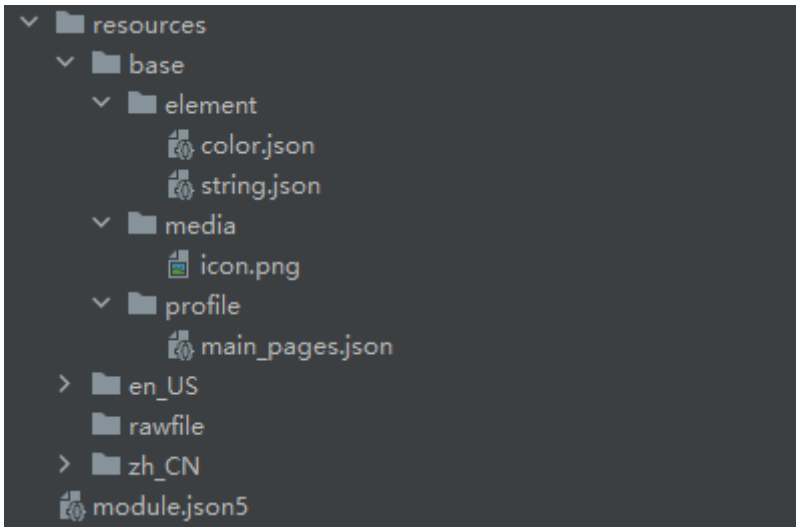
- build-profile.json5 是模块级配置信息，包括编译构建配置项。
- hvigorfile.ts 文件是模块级构建脚本。
- oh-package.json5 是模块级依赖配置信息文件。

进入 src>main>ets 目录中，其分为 entryability、pages 两个文件夹。

- entryability 存放 ability 文件，用于当前 ability 应用逻辑和生命周期管理。
- pages 存放 UI 界面相关代码文件，初始会生成一个 Index 页面。



resources 目录下存放模块公共的多媒体、字符串及布局文件等资源，分别存放在 element、media 文件夹中。



2.6.3 app.json5

AppScope>app.json5 是应用的全局的配置文件，用于存放应用公共的配置信息。

```
"app": {
  "bundleName": "com.example.helloworld",
  "vendor": "example",
  "versionCode": 1000000,
  "versionName": "1.0.0",
  "icon": "$media:app_icon",
  "label": HelloWorld
}
```

其中配置信息如下：

- bundleName 是包名。
- vendor 是应用程序供应商。
- versionCode 是用于区分应用版本。
- versionName 是版本号。

2.6.4 module.json5

entry>src>main>module.json5 是模块的配置文件，包含当前模块的配置信息。

```
"module": {
  "name": "entry",
  "type": "entry",
  "description": module description,
  "mainElement": "EntryAbility",
  "deviceTypes": [
    "phone"
  ],
  "deliveryWithInstall": true,
  "installationFree": false,
  "pages": "$profile:main_pages",
  "abilities": [
    {
      "name": "EntryAbility",
      "srcEntry": "./ets/entryability/EntryAbility.ts",
      "description": description,
      "icon": "$media:icon",
      "label": label,
      "startWindowIcon": "$media:icon",
      "startWindowBackground": #FFFFFF,
      "exported": true,
      "skills": [
        {
          "entities": [
            "entity.system.home"
          ],
          "actions": [
            "action.system.home"
          ]
        }
      ]
    }
  ]
}
```

其中 module 对应的是模块的配置信息，一个模块对应一个打包后的 hap 包，hap 包全称是 HarmonyOS Ability Package，其中包含了 ability、第三方库、资源和配置文件。其具体属性及其描述可以参照下表 1。

表 1 module.json5 默认配置属性及描述

属性	描述

属性	描述
name	该标签标识当前 module 的名字，module 打包成 hap 后，表示 hap 的名称，标签值采用字符串表示（最大长度 31 个字节），该名称在整个应用要唯一。
type	表示模块的类型，类型有三种，分别是 entry、feature 和 har。
srcEntry	当前模块的入口文件路径。
description	当前模块的描述信息。
mainElement	该标签标识 hap 的入口 ability 名称或者 extension 名称。只有配置为 mainElement 的 ability 或者 extension 才允许在服务中心露出。
deviceTypes	该标签标识 hap 可以运行在哪类设备上，标签值采用字符串数组的表示。
deliveryWithInstall	标识当前 Module 是否在用户主动安装的时候安装，表示该 Module 对应的 HAP 是否跟随应用一起安装。- true：主动安装时安装。- false：主动安装时不安装。
installationFree	标识当前 Module 是否支持免安装特性。- true：表示支持免安装特性，且符合免安装约束。- false：表示不支持免安装特性。
pages	对应的是 main_pages.json 文件，用于配置 ability 中用到的 page 信息。
abilities	是一个数组，存放当前模块中所有的 ability 元能力的配置信息，其中可以有多个 ability。

对于 abilities 中每一个 ability 的属性项，其描述信息如下表 2。

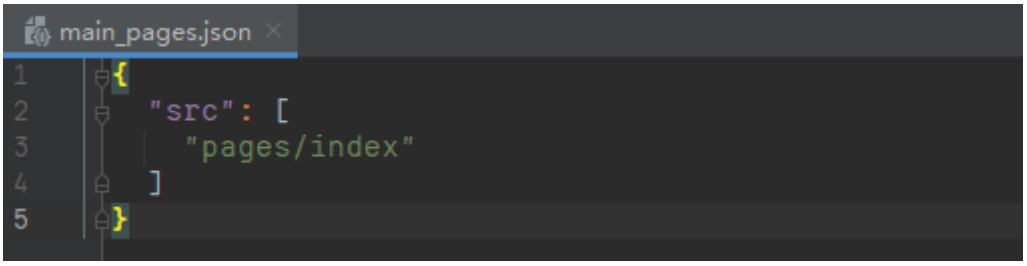
表 2 abilities 中对象的默认配置属性及描述

属性	描述

属性	描述
name	该标签标识当前 ability 的逻辑名，该名称在整个应用要唯一，标签值采用字符串表示（最大长度 127 个字节）。
srcEntry	ability 的入口代码路径。
description	ability 的描述信息。
icon	ability 的图标。该标签标识 ability 图标，标签值为资源文件的索引。该标签可缺省，缺省值为空。如果 ability 被配置为 MainElement，该标签必须配置。
label	ability 的标签名。
startWindowIcon	启动页面的图标。
startWindowBackgrou	启动页面的背景色。
visible	ability 是否可以被其他应用程序调用，true 表示可以被其它应用调用， false 表示不可以被其它应用调用。
skills	标识能够接收的意图的 action 值的集合，取值通常为系统预定义的 action 值，也允许自定义。
entities	标识能够接收的 Want 的 Action 值的集合，取值通常为系统预定义的 action 值，也允许自定义。
actions	标识能够接收 Want 的 Entity 值的集合。

2.6.5 main_pages.json

src/main/resources/base/profile/main_pages.json 文件保存的是页面 page 的路径配置信息，所有需要进行路由跳转的 page 页面都要在这里进行配置。



2.7 章节习题

1) DevEco Studio 是开发 HarmonyOS 应用的一站式集成开发环境。正确(True)

2) main_pages.json 存放页面 page 路径配置信息。正确(True)

3) 在 stage 模型中，下列配置文件属于 AppScope 文件夹的是？C

A. main_pages.json

B. module.json5

C. app.json5

D. package.json

4) 如何在 DevEco Studio 中创建新项目？BC

A. 在计算机上创建一个新文件，并将其命名为“new harmonyOS 项目”

B. 如果已打开项目，从 DevEco Studio 菜单选择'file>new>Create Project'

C. 如果第一次打开 DevEco Studio，在欢迎页点击“Create new Project”

5) module.json5 配置文件中，包含了以下哪些信息？ABD

A. ability 的相关配置信息

B. 模块名

C. 应用的版本号

D. 模块类型

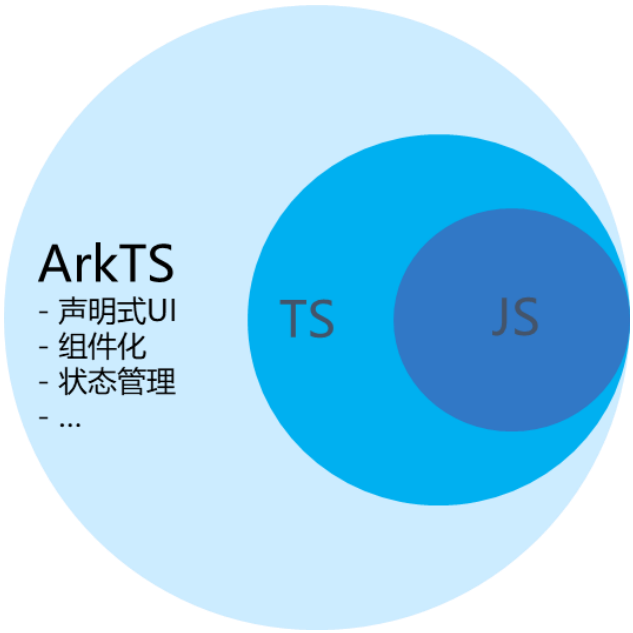
3 第三章 ArkTS 开发语言介绍

3.1 TypeScrip 快速入门

学习 TypeScript 对于 HarmonyOS 应用开发至关重要。在 HarmonyOS 中，主力编程语言为 ArKTS，它是基于 TypeScript 的一种语言，其通过与 ArkUI 框架的匹配，拓展了声明式 UI 和状态管理等能力，使开发者能够以更简洁自然的方式开发跨端应用。TypeScript 本身是 JavaScript 的超集，通过引入静态类型定义等特性，提高了代码的可维护性和可读性，有助于在编码阶段检测潜在错误，提高开发效率另外，学习 TypeScript 还为处理 HarmonyOS 应用中的 UI 和应用状态提供了更强大的支持，在并发任务方面也有相应的扩展。为了更好地对 HarmonyOS 进行开发需要掌握 TypeScript 语言，本接我们重点介绍 TypeScript 语言。

3.1.1 编程语言介绍

ArkTS 是 HarmonyOS 优选的主力应用开发语言。它在 TypeScript（简称 TS）的基础上，匹配 ArkUI 框架，扩展了声明式 UI、状态管理等相应的能力，让开发者以更简洁、更自然的方式开发跨端应用。要了解什么是 ArkTS，我们首先要了解下 ArkTS、TypeScript 和 JavaScript 之间的关系：



- JavaScript 是一种属于网络的高级脚本语言，已经被广泛用于 Web 应用开发，常用来为网页添加各式各样的动态功能，为用户提供更流畅美观的浏览效果。
- TypeScript 是 JavaScript 的一个超集，它扩展了 JavaScript 的语法，通过在 JavaScript 的基础上添加静态类型定义构建而成，是一个开源的编程语言。
- ArkTS 兼容 TypeScript 语言，拓展了声明式 UI、状态管理、并发任务等能力。

在学习 ArkTS 声明式的相关语法之前，我们首先学习下 TypeScript 的基础语法。

3.1.2 基础类型

TypeScript 支持一些基础的数据类型，如布尔型、数组、字符串等，下文举例几个较为常用的数据类型，我们来了解下他们的基本使用。

- **布尔值**

TypeScript 中可以使用 boolean 来表示这个变量是布尔值，可以赋值为 true 或者 false。例如我们这里可以设置 IsDone 为 False 来表示未完成。

```
let isDone: boolean = false;
```

- **数字**

TypeScript 里的所有数字都是浮点数，这些浮点数的类型是 number。除了支持十进制，还支持二进制、八进制、十六进制。如下我们用十进制、二进制、八进制和十六进制分别定义了 2023，当把数据通过日志方式打印出来，结果都会转换为十进制，也都是 2023。

```
let decLiteral: number = 2023;
let binaryLiteral: number = 0b11111100111;
let octalLiteral: number = 0o3747;
let hexLiteral: number = 0x7e7;
```

- **字符串**

TypeScript 里使用 string 表示文本数据类型， 可以使用双引号（ "）或单引号（'）表示字符串。例如我们这里定义 Name 是一个字符串类型，其数值我们可以用双引号或者单引号包裹起来。

```
let name: string = "Jacky";
name = "Tom";
name = 'Mick';
```

- **数组**

TypeScrip 有两种方式可以定义数组。 第一种，可以在元素类型后面接上 []，表示由此类型元素组成的一个数组。

```
let list: number[] = [1, 2, 3];
```

第二种方式是使用数组泛型，Array<元素类型>。

```
let list: Array<number> = [1, 2, 3];
```

- **元组**

元组类型允许表示一个已知元素数量和类型的数组，各元素的类型不必相同。 比如，你可以定义一对值分别为 string 和 number 类型的元组。

例如这里我们定义了一个 X 元组，类型为 String 和 Number。第一行的赋值和我们元组定义的顺序是一致的，这种是正确的。第二行先赋值 Number 后赋值 String，这种赋值的顺序和我们定义的不一致，所以是错误的。

```
let x: [string, number];
x = ['hello', 10]; // OK
x = [10, 'hello']; // Error
```

- **枚举**

enum 类型是对 JavaScript 标准数据类型的一个补充，使用枚举类型可以为一组数值赋予友好的名字。例如我们这里定义 Color 为 Red, Green 和 Blue，到时候就可以使用 Color.Green 来定义颜色。

```
enum Color {Red, Green, Blue};
let c: Color = Color.Green;
```

- **Unknown**

有时候，我们会想要为那些在编程阶段还不清楚类型的变量指定一个类型。这种情况下，我们不希望类型检查器对这些值进行检查而是直接让它们通过编译阶段的检查。那么我们可以使用 unknown 类型来标记这些变量。

如下案例中，这里的 Not Sure 定义为 Unknown 后，我们可以赋值为 Number 类型，也可以赋值为 String 类型，还可以赋值为 false 类型。

```
let notSure: unknown = 4;
notSure = 'maybe a string instead';
notSure = false;
```

- **Void**

当一个函数没有返回值时，你通常会见到其返回值类型是 void。如下的 test 方法，其返回类型就是 Void。

```
function test(): void {
```

```
    console.log('This is function is void');
}
```

- **Null 和 Undefined**

TypeScript 里，undefined 和 null 两者各自有自己的类型分别叫做 undefined 和 null。

```
let u: undefined = undefined;
let n: null = null;
```

- **联合类型**

联合类型（Union Types）表示取值可以为多种类型中的一种。例如我们这里定义 MyFavoriteNumber 为联合类型，其取值可以是 String 或者 Number，我们可以给其赋值为字符串 7，也可以给其赋值为 Number 类型 7。联合类型在日常的使用过程中用的比较多，大家要掌握这种定义方式。

```
let myFavoriteNumber: string | number;
myFavoriteNumber = 'seven';
myFavoriteNumber = 7;
```

3.1.3 条件语句

条件语句用于基于不同的条件来执行不同的动作。TypeScript 条件语句是通过一条或多条语句的执行结果（True 或 False）来决定执行的代码块。

- **if 语句**

TypeScript if 语句由一个布尔表达式后跟一个或多个语句组成。例如，如下代码中是一个 If 语句，定义的 Number 为 5，判断的条件是 Number 大于 0，程序满足这个条件会输出数字为正数。

```
var num:number = 5
if (num > 0) {
  console.log('数字是正数')
}
```

- **if...else 语句**

一个 if 语句后可跟一个可选的 else 语句，else 语句在布尔表达式为 false 时执行。如下代码中声明一个 If-else 语句，定义的 Number 是 12，符合 Number 对 2 取余等于 0 的条件，所以输出为偶数。

```
var num:number = 12;
if (num % 2==0) {
    console.log('偶数');
} else {
    console.log('奇数');
}
```

- **if...else if....else 语句**

if...else if....else 语句在执行多个判断条件的时候很有用。如下代码中是一个 If-else 语句，定义的 Number 为 0，满足最后的 else 条件，所以输出为 0。

```
var num:number = 0
if(num > 0) {
    console.log(num+' 是正数')
} else if(num < 0) {
    console.log(num+' 是负数')
} else {
    console.log(num+' 为 0')
}
```

- **switch...case 语句**

除了可以通过 If-else 语句进行条件判断外，还可以通过 Switch-case 语句进行条件判断。一个 switch 语句允许测试一个变量等于多个值时的情况。每个值称为一个 case，且被测试的变量会对每个 switch case 进行检查。

如下代码中我们有 4 个 Case 条件，分别是 A 输出日志优，B 输出日志良，C 输出日志及格，D 输出日志不及格，最后还有一个 default 条件，当输入的字符不在 ABCD 中则表示非法输入，最后我们定义的 Grade 为 A，所以这个代码打印的日志为优。

```
var grade:string = 'A';
switch(grade) {
    case 'A': {
        console.log('优');
        break;
    }
```

```
        case 'B': {
            console.log('良');
            break;
        }
        case 'C': {
            console.log('及格');
            break;
        }
        case 'D': {
            console.log('不及格');
            break;
        }
        default: {
            console.log('非法输入');
            break;
        }
    }
```

3.1.4 函数

函数是一组一起执行一个任务的语句，函数声明要告诉编译器函数的名称、返回类型和参数。TypeScript 可以创建有名字的函数和匿名函数，其创建方法如下：

```
// 有名函数
function add(x, y) {
  return x + y;
}

// 匿名函数
let myAdd = function (x, y) {
  return x + y;
};
```

- **为函数定义类型**

为了确保输入输出的准确性，我们可以为上面那个函数添加类型：

```
// 有名函数：给变量设置为 number 类型
function add(x: number, y: number): number {
  return x + y;
}

// 匿名函数：给变量设置为 number 类型
let myAdd = function (x: number, y: number): number {
  return x + y;
};
```

以上函数的名称叫做 Add，实现的是两个数值的累加，参数是 X 和 Y 两个 number 类型的数字，返回值是 X+Y 的结果。其返回的类型也是 number 类型，上面一个函数是有名函数，下面一个函数是匿名函数，匿名函数没有函数名，但其作用是一样的。

- **可选参数**

在 TypeScript 里我们可以在参数名旁使用 ?实现可选参数的功能。 比如，我们想让 lastName 是可选的。使用了可选参数后，我们在调用函数的时候就可以传入一个参数或者两个参数，如 Result1 和 Result2 中的代码所示：

```
function buildName(firstName: string, lastName?: string) {
    if (lastName)
        return firstName + ' ' + lastName;
    else
        return firstName;
}

let result1 = buildName('Bob');
let result2 = buildName('Bob', 'Adams');
```

- **剩余参数**

函数的入参除了可以使用可选参数外，还可以使用剩余参数。剩余参数会被当做个数不限的可选参数，可以一个都没有，同样也可以有任意个。 可以使用省略号（ ...）进行定义。如下代码中，我们调用

getEmployeeName 方法时，可以只传入 firstName，也就是 Joseph，不传入剩余参数，也可以传入多个剩余参数：Samuel,Lucas,MacKinzie 等。

```
function getEmployeeName(firstName: string, ...restOfName: string[]) {
  return firstName + ' ' + restOfName.join(' ');
}

let employeeName = getEmployeeName('Joseph', 'Samuel', 'Lucas', 'MacKinzie');
```

- **箭头函数**

ES6 版本的 TypeScript 提供了一个箭头函数，它是定义匿名函数的简写语法，用于函数表达式，它省略了 function 关键字。箭头函数的定义如下，其函数是一个语句块：

```
( [param1, parma2,...param n] )=> {
   // 代码块
}
```

其中，括号内是函数的入参，可以有 0 到多个参数，箭头后是函数的代码块。我们可以将这个箭头函数赋值给一个变量，如下所示：

```
let arrowFun = ( [param1, parma2,...param n] )=> {
   // 代码块
}
```

如何要主动调用这个箭头函数，可以按如下方法去调用：

```
arrowFun(param1, parma2,...param n)
```

接下来我们看看如何将我们熟悉的函数定义方式转换为箭头函数。我们可以定义一个判断正负数的函数，如下：

```
function testNumber(num: number) {
  if (num > 0) {
    console.log(num + ' 是正数');
  } else if (num < 0) {
    console.log(num + ' 是负数');
  } else {
```

```
    console.log(num + ' 为 0');
  }
}
```

其调用方法如下：

```
testNumber(1)  //输出日志：1 是正数
```

如果将这个函数定义为箭头函数，定义如下所示：

```
let testArrowFun = (num: number) => {
  if (num > 0) {
    console.log(num + ' 是正数');
  } else if (num < 0) {
    console.log(num + ' 是负数');
  } else {
    console.log(num + ' 为 0');
  }
}
```

其调用方法如下：

```
testArrowFun(-1)  //输出日志：-1 是负数
```

后面，我们在学习 HarmonyOS 应用开发时会经常用到箭头函数。例如，给一个按钮添加点击事件，其中 onClick 事件中的函数就是箭头函数。

```
Button("Click Now")
  .onClick(() => {
    console.info("Button is click")
  })
```

3.1.5 类

TypeScript 支持基于类的面向对象的编程方式，定义类的关键字为 class，后面紧跟类名。类描述了所创建的对象共同的属性和方法。

- **类的定义**

例如，我们可以声明一个 Person 类，这个类有 3 个成员：一个是属性（包含 name 和 age），一个是构造函数，一个是 getPersonInfo 方法，其定义如下所示。

```
class Person {
  private name: string
  private age: number

  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }

  public getPersonInfo(): string {
    return `My name is ${this.name} and age is ${this.age}`;
  }
}
```

通过上面的 Person 类，我们可以定义一个人物 Jacky 并获取他的基本信息，其定义如下，我们可以使用 new 方法，传入 person 的姓名和年龄，创建为 person1 对象，person1 可以调用其中的公有属性的方法，也就是 getpersoninfo 方法，这个是一个最简单的类的定义和调用。当然类里面还有很多的知识，比如我们可以通过修改修饰符，private public 等来控制属性和方法的访问权限，这些知识大家可以参考 TS 的相关文档进行自行学习。

```
let person1 = new Person('Jacky', 18);
person1.getPersonInfo();
```

- **继承**

继承就是子类继承父类的特征和行为，使得子类具有父类相同的行为。TypeScript 中允许使用继承来扩展现有的类，对应的关键字为 extends。如下案例中，我们定义 employee 是继承于 person 的 employee 叫 person 新增了一个属性 department，我们可以这样去定义它的构造方法，通过 super 关键字实际上就调用了 person 中的构造方法，初始化 name 和 age，并在构造方法中初始化好了 department，employee 有个公有方法，getemployeeinfo 获取雇员的信息，其中调用 getpersoninfo 来获取雇员的姓名、年龄信息。

```
class Employee extends Person {
```

```
    private department: string


    constructor(name: string, age: number, department: string) {
      super(name, age);
      this.department = department;
    }


    public getEmployeeInfo(): string {
      return this.getPersonInfo() + ` and work in ${this.department}`;
    }
  }
```

通过上面的 Employee 类，我们可以定义一个人物 Tom，这里可以获取他的基本信息，也可以获取他的雇员信息，其定义如下：

```
let person2 = new Employee('Tom', 28, 'HuaWei');
person2.getPersonInfo();
person2.getEmployeeInfo();
```

在 TypeScript 中，有 public、private、protected 修饰符，其功能和具体使用场景大家可以参考 TypeScript 的相关学习资料，进行拓展学习。

3.1.6 模块

随着应用越来越大，通常要将代码拆分成多个文件，即所谓的模块（module）。模块可以相互加载，并可以使用特殊的指令 export 和 import 来交换功能，从另一个模块调用一个模块的函数。

两个模块之间的关系是通过在文件级别上使用 import 和 export 建立的。模块里面的变量、函数和类等在模块外部是不可见的，除非明确地使用 export 导出它们。类似地，我们必须通过 import 导入其他模块导出的变量、函数、类等。

- **导出**

任何声明（比如变量，函数，类，类型别名或接口）都能够通过添加 export 关键字来导出，例如我们要把 NewsData 这个类导出，代码示意如下：

```
export class NewsData {
```

```
    title: string;
    content: string;
    imagesUrl: Array<NewsFile>;
    source: string;

    constructor(title: string, content: string, imagesUrl: Array<NewsFile>, source: string) {
      this.title = title;
      this.content = content;
      this.imagesUrl = imagesUrl;
      this.source = source;
    }
  }
```

- **导入**

模块的导入操作与导出一样简单。 可以使用以下 import 形式之一来导入其它模块中的导出内容。

```
import { NewsData } from '../common/bean/NewsData';
```

以上案例中，我们在一个文件中定义了一个类 news data，我们要在其他文件中引用这个类，首先就需要在这个类的前面加一个修饰符 export，之后我们可以利用 import 来导入这个类,这个类的具体路径是填写在 form 后面的。export 进来后，我们就可以在这个模块中引用其他模块中定义的 NewsData。

3.1.7 可迭代对象

当一个对象实现了 Symbol.iterator 属性时，我们认为它是可迭代的。一些内置的类型如 Array，Map，Set，String，Int32Array，Uint32Array 等都具有可迭代性。

- **for..of 语句**

for..of 会遍历可迭代的对象，调用对象上的 Symbol.iterator 方法。 下面是在数组上使用 for..of 的简单例子，如这里定了一个 someArray 数组，使用 for-of 语句进行循环遍历，可以打印这个数组中的元素。

```
let someArray = [1, "string", false];

for (let entry of someArray) {
  console.log(entry); // 1, "string", false
}
```

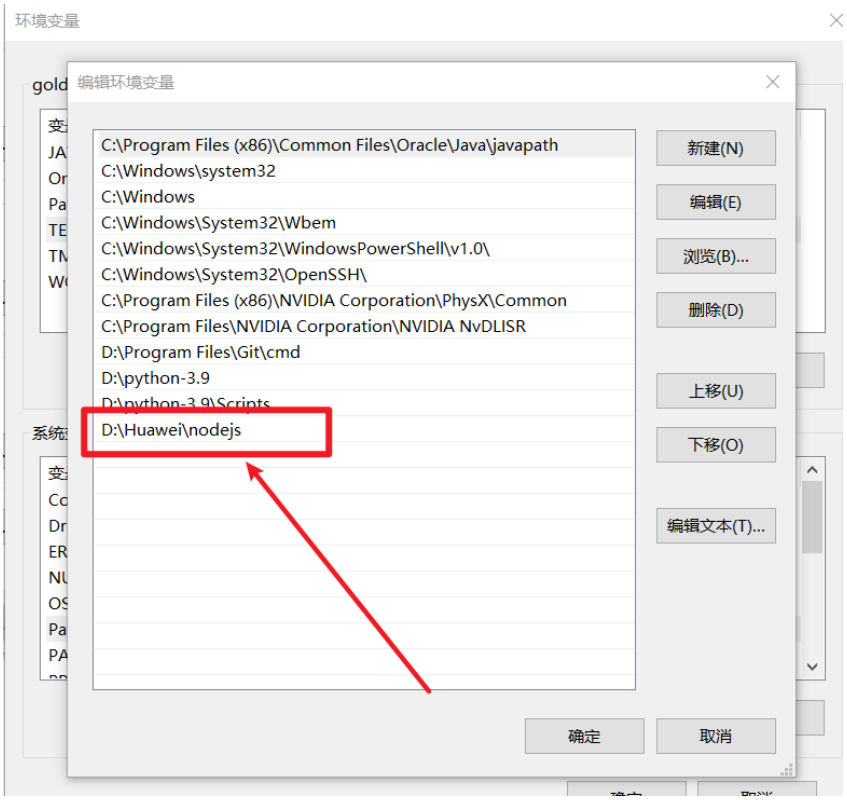
- **for..of vs. for..in 语句**

for..of 和 for..in 均可迭代一个列表，但是用于迭代的值却不同：for..in 迭代的是对象的键，而 for..of 则迭代的是对象的值。如下，for...in 打印的是数组的下标。

```
let list = [4, 5, 6];

for (let i in list) {
    console.log(i); // "0", "1", "2",
}

for (let i of list) {
    console.log(i); // "4", "5", "6"
}
```

3.1.8 DevEco Studio 中配置 TypeScript

配置 node.js 的环境变量：



安装 typescript：

```
npm install -g typescript
```

安装完成后我们可以使用 **tsc** 命令来执行 TypeScript 的相关代码，以下是查看版本号：

```
$ tsc -v
Version 5.3.2
```

3.2 初识 ArkTs 语言

ArkTS 是 HarmonyOS 优选的主力应用开发语言。ArkTS 围绕应用开发在 TypeScript（简称 TS）生态基础上做了进一步扩展，继承了 TS 的所有特性，是 TS 的超集。因此，在学习 ArkTS 语言之前，建议开发者具备 TS 语言开发能力。

当前，ArkTS 在 TS 的基础上主要扩展了如下能力：

- 基本语法：ArkTS 定义了声明式 UI 描述、自定义组件和动态扩展 UI 元素的能力，再配合 ArkUI 开发框架中的系统组件及其相关的事件方法、属性方法等共同构成了 UI 开发的主体。
- 状态管理：ArkTS 提供了多维度的状态管理机制。在 UI 开发框架中，与 UI 相关联的数据可以在组件内使用，也可以在不同组件层级间传递，比如父子组件之间、爷孙组件之间，还可以在应用全局范围内传递或跨设备传递。另外，从数据的传递形式来看，可分为只读的单向传递和可变更的双向传递。开发者可以灵活的利用这些能力来实现数据和 UI 的联动。
- 渲染控制：ArkTS 提供了渲染控制的能力。条件渲染可根据应用的不同状态，渲染对应状态下的 UI 内容。循环渲染可从数据源中迭代获取数据，并在每次迭代过程中创建相应的组件。数据懒加载从数据源中按需迭代数据，并在每次迭代过程中创建相应的组件。

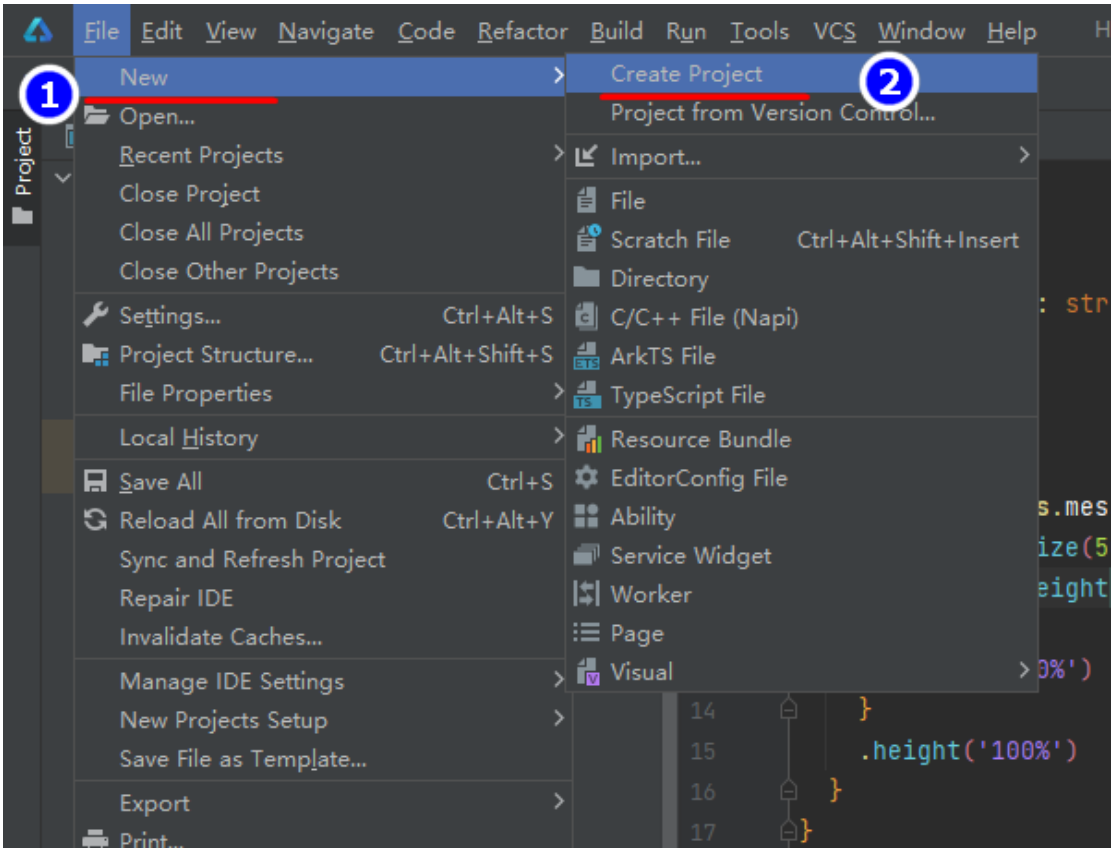
未来，ArkTS 会结合应用开发/运行的需求持续演进，逐步提供并行和并发能力增强、系统类型增强、分布式开发范式等更多特性。

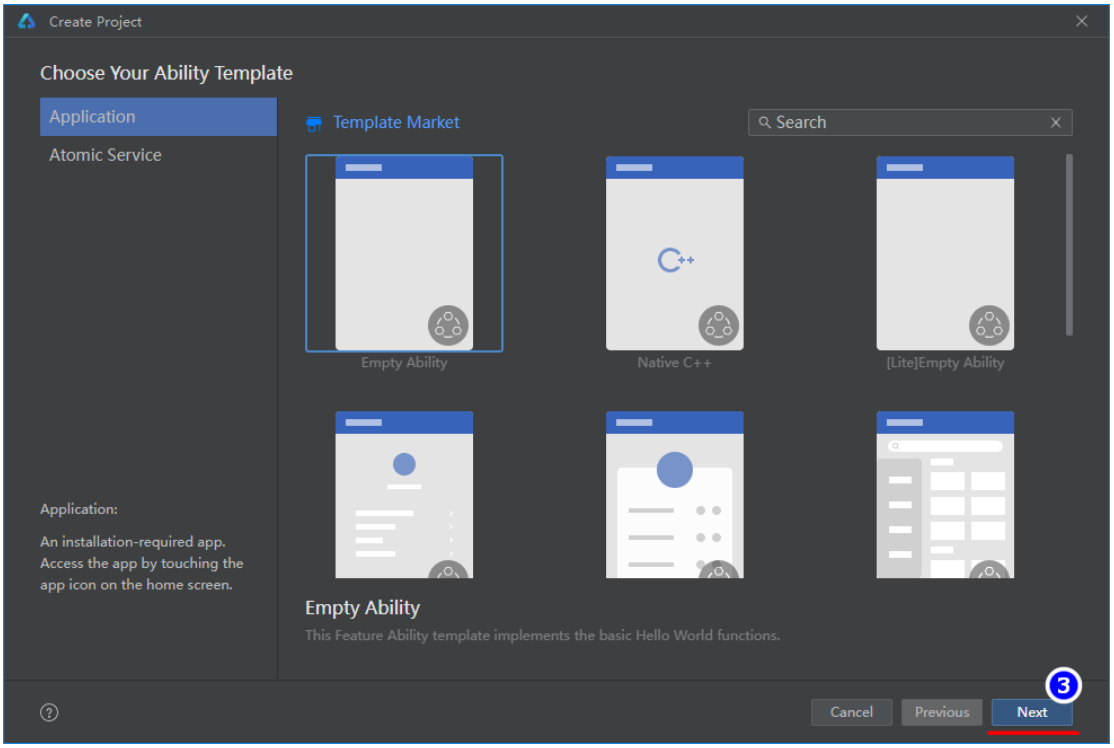
3.3 ArkTS 基本语法

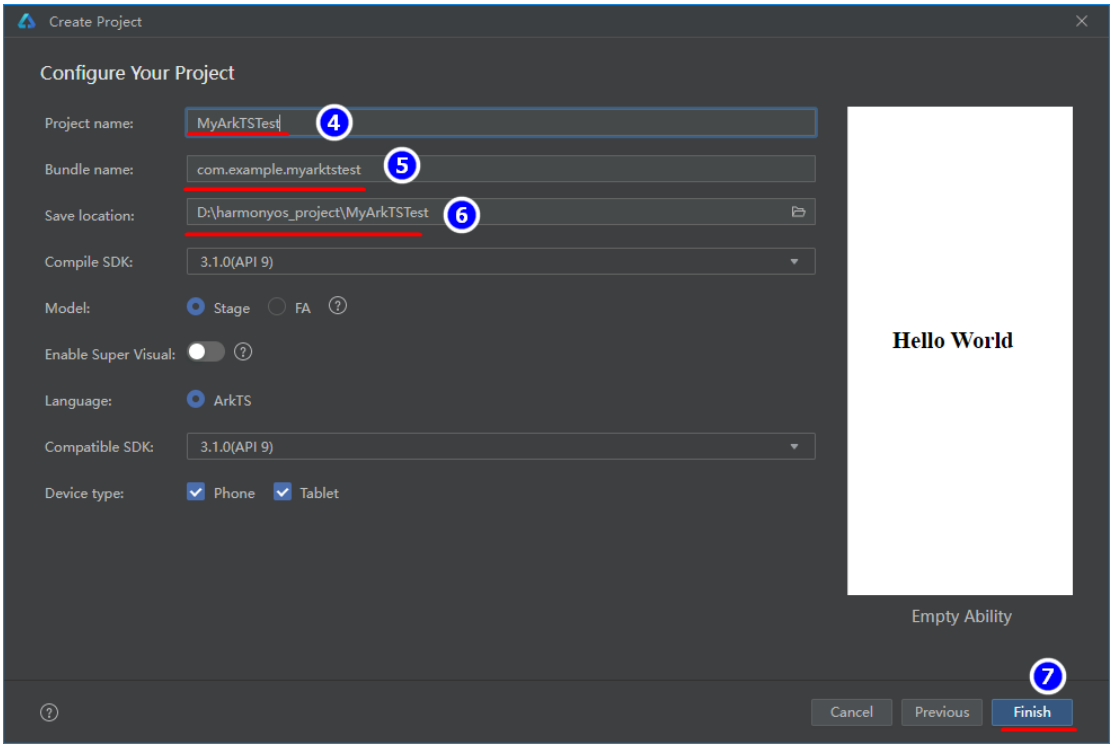
3.3.1 基本语法概述

在初步了解了 ArkTS 语言之后，我们以一个具体的示例来说明 ArkTS 的基本组成。该案例中当开发者点击按钮时，文本内容从“Hello World”变为“Hello ArkUI”,创建步骤如下。

1) 打开 DevEvo Studio 开发工具，新建项目


File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
New
Open...
Recent Projects
Close Project
Close All Projects
Close Other Projects
Settings... Ctrl+Alt+S
Project Structure... Ctrl+Alt+Shift+S
File Properties
Local History
Save All Ctrl+S
Reload All from Disk Ctrl+Alt+Y
Sync and Refresh Project
Repair IDE
Invalidate Caches...
Manage IDE Settings
New Projects Setup
Save File as Template...
Export
Print...
Create Project
Project from Version Control...
Import...
File
Scratch File Ctrl+Alt+Shift+Insert
Directory
C/C++ File (Napi)
ArkTS File
TypeScript File
Resource Bundle
EditorConfig File
Ability
Service Widget
Worker
Page
Visual
1
2
Project
.height('100%')


Create Project
Choose Your Ability Template
Application
Atomic Service
Template Market
Search
Empty Ability
Native C++
[Lite]Empty Ability
Application:
An installation-required app.
Access the app by touching the app icon on the home screen.
Empty Ability
This Feature Ability template implements the basic Hello World functions.
Cancel
Previous
Next
3



项目创建完成，进入该项目等待项目初始化完成即可：

2) 在 Index.ets 中写入如下代码

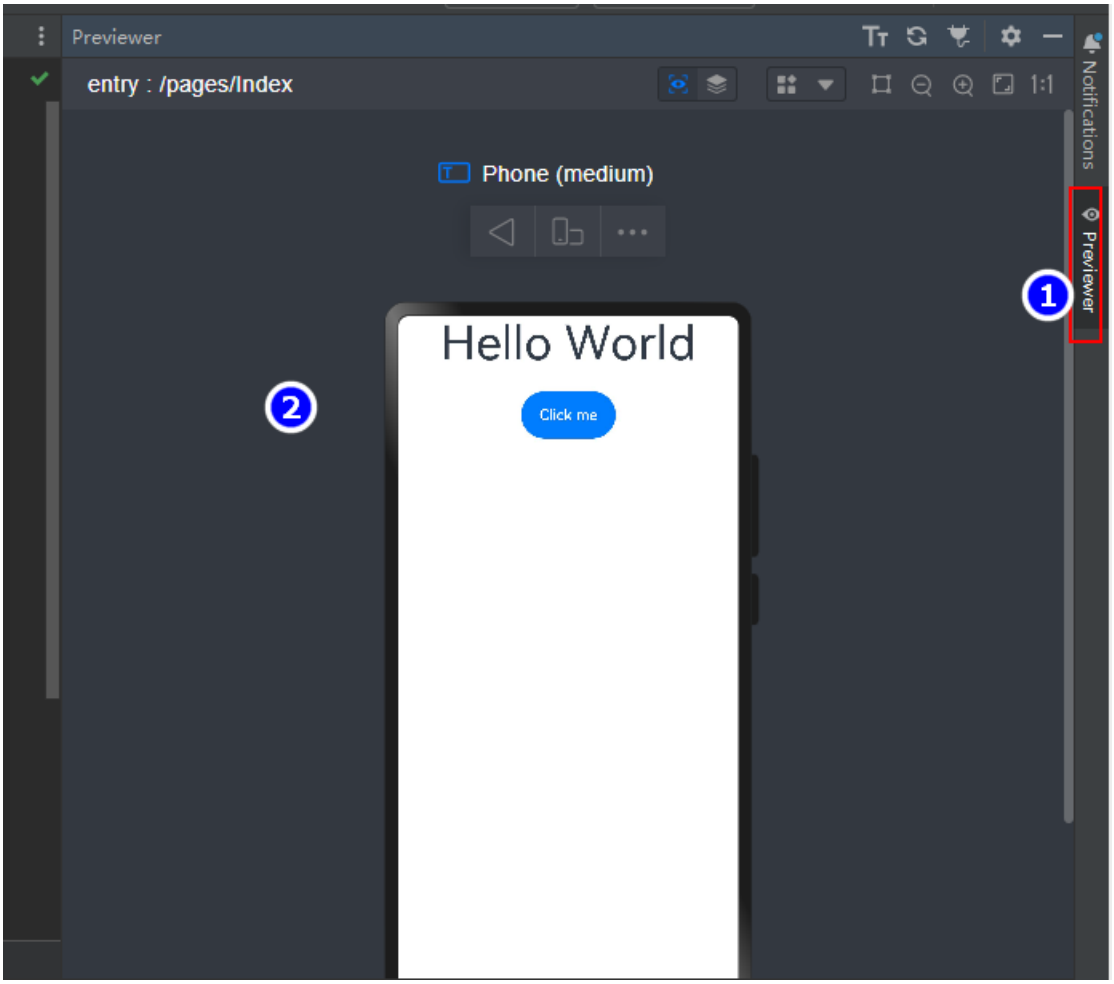
```
@Entry
@Component
struct Hello {
  @State myText: string = 'World'

  build() {
    Column(){
      Text(`Hello ${this.myText}`)
        .fontSize(50)
      Divider()
      Button('Click me')
        .onClick(()=>{
          this.myText='ArkUI'
        })
        .height(50)
        .width(100)
        .margin({top:20})
```

```
  }
 }
}
```

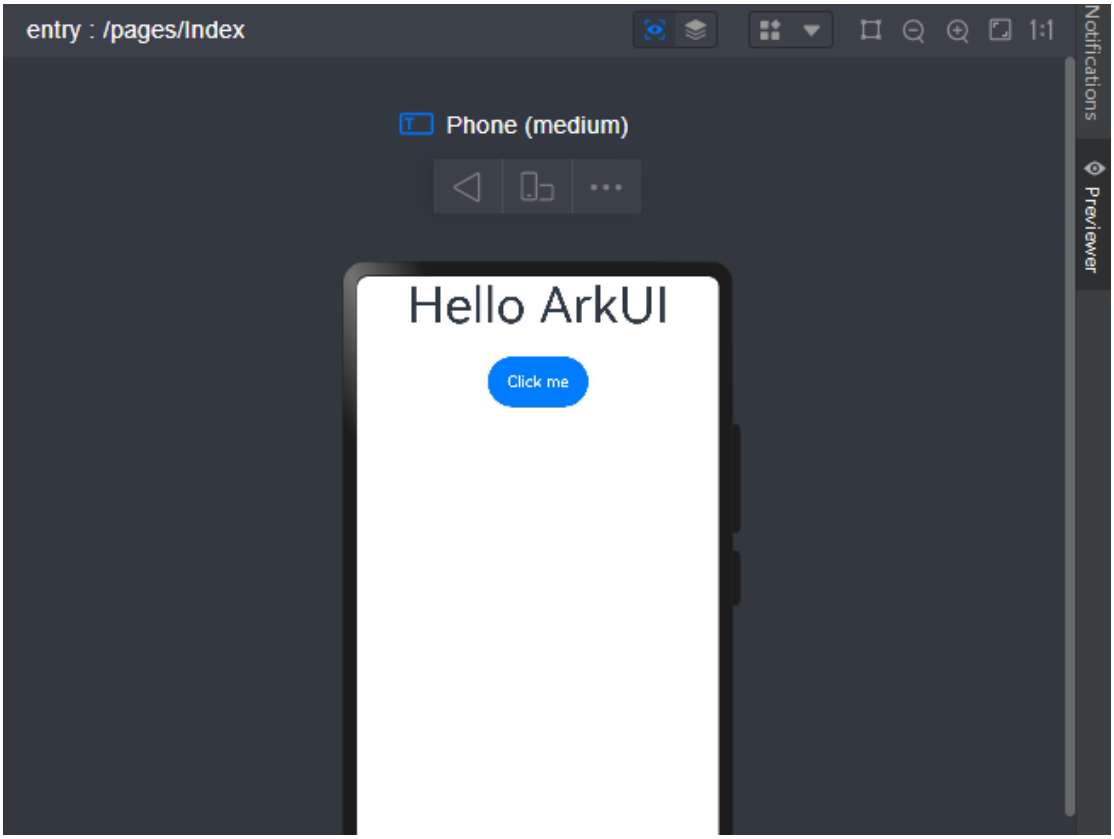
对以上代码详细解释如下：

```
    //@Entry 装饰的自定义组件将作为 UI 页面的入口。在单个 UI 页面中，最多可以使用@Entry 装饰一个自定义组件。
@Entry
/**
 * @Component 是一种装饰器，代表自定义组件，用@Component 装饰的 struct Hello 代表一个自定义的结构体，名字是 Hello，是可重用的 UI 单元，可以与其他组件组合。
 */
@Component
struct Hello {
 //@State 是一种装饰器，被它装饰的变量 myText 值发生改变时，会触发该变量所对应的自定义组件 Hello 的 UI 界面进行自动刷新。
 @State myText: string = 'World'

 //build 方法中的代码块表示 UI 描述，以声明式的方式描述 UI 结构。
 build() {
  //Column 是内置组件，表示设置一列
  Column(){
   //设置文本及内容
   Text(`Hello ${this.myText}`)
    .fontSize(50)//设置文本大小
   Divider() //Divider 提供分隔器组件，分隔不同内容块/内容元素。
   //设置按钮
   Button('Click me')
    //设置按钮点击事件，点击按钮时将 myText 由 World 改变成 ArkUI
    .onClick(()=>{
     this.myText='ArkUI'
    })
    .height(50) //设置按钮高度
    .width(100) //设置按钮宽度
    .margin({top:20}) //设置按钮外边距
  }
 }
}
```

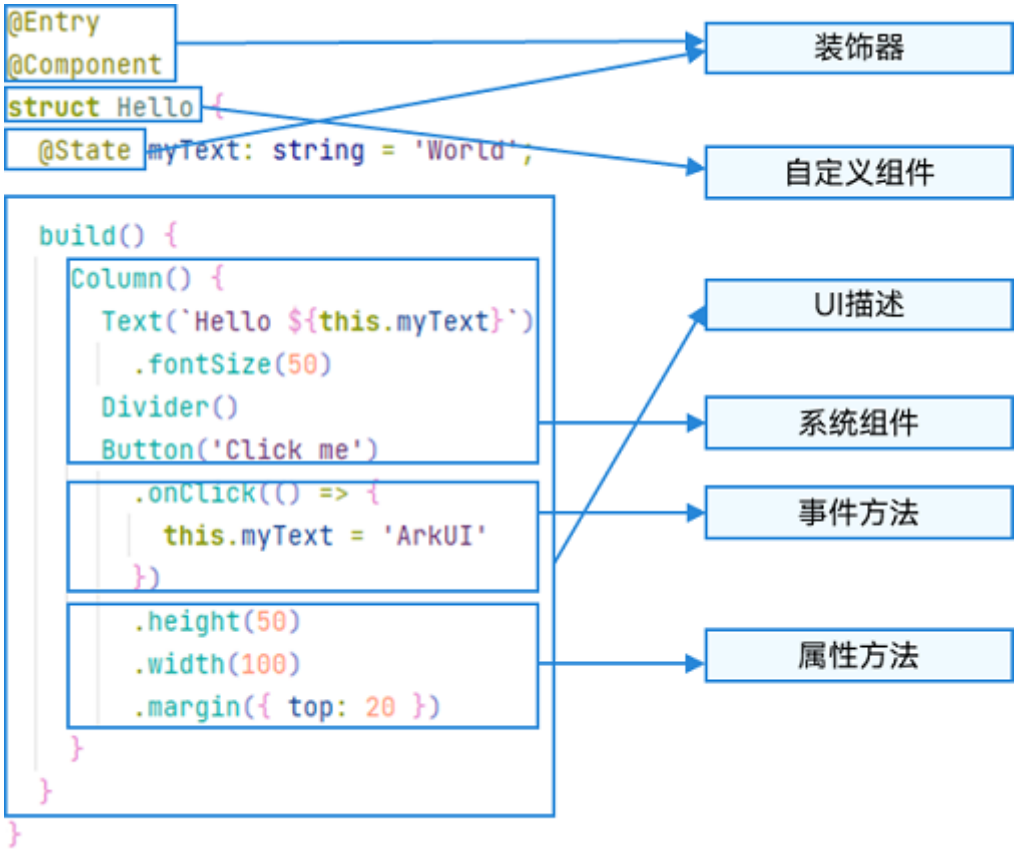
3) 打开预览，验证组件功能



当点击“Click me”按钮时，“Hello World”变换成“Hello ArkUI”。



在以上示例中，ArkTS 的基本组成如下所示。



- 装饰器： 用于装饰类、结构、方法以及变量，并赋予其特殊的含义。如上述示例中@Entry、@Component 和@State 都是装饰器，@Component 表示自定义组件，@Entry 表示该自定义组件为入口组件，@State 表示组件中的状态变量，状态变量变化会触发 UI 刷新。
- UI 描述：以声明式的方式来描述 UI 的结构，例如 build()方法中的代码块。
- 自定义组件：可复用的 UI 单元，可组合其他组件，如上述被@Component 装饰的 struct Hello。
- 系统组件：ArkUI 框架中默认内置的基础和容器组件，可直接被开发者调用，比如示例中的 Column、Text、Divider、Button。
- 属性方法：组件可以通过链式调用配置多项属性，如 fontSize()、width()、height()、backgroundColor()等。
- 事件方法：组件可以通过链式调用设置多个事件的响应逻辑，如跟随在 Button 后面的 onClick()。

系统组件、属性方法、事件方法具体使用可参考基于 ArkTS 的声明式开发范式。除此之外，ArkTS 扩展了多种语法范式来使开发更加便捷：

- @Builder/@BuilderParam：特殊的封装 UI 描述的方法，细粒度的封装和复用 UI 描述。
- @Extend/@Style：扩展内置组件和封装属性样式，更灵活地组合内置组件。

- stateStyles：多态样式，可以依据组件的内部状态的不同，设置不同样式。

3.3.2 声明式 UI 概述

ArkTS 以声明方式组合和扩展组件来描述应用程序的 UI，同时还提供了基本的属性、事件和子组件配置方法，帮助开发者实现应用交互逻辑。

3.3.2.1 创建组件

根据组件构造方法的不同，创建组件包含有参数和无参数两种方式。创建组件时不需要 new 运算符。

- **无参数**

如果组件的接口定义没有包含必选构造参数，则组件后面的“()”不需要配置任何内容。例如，Divider 组件不包含构造参数：

```
Column() {
  Text('item 1')
  Divider()
  Text('item 2')
}
```

示例演示：

```
@Entry
@Component
struct UITest {
  build() {
    Column() {
      Text('item 1')
      Divider()
      Text('item 2')
    }
  }
```

预览如下：

- **有参数**

如果组件的接口定义包含构造参数，则在组件后面的“()”配置相应参数。

a. Image 组件的必选参数 src。

```
Image('https://xyz/test.jpg')
```

b. Text 组件的非必选参数 content。

```
// string 类型的参数
Text('test')
// $r 形式引入应用资源，可应用于多语言场景
Text($r('app.string.title_value'))
// 无参数形式
Text()
```

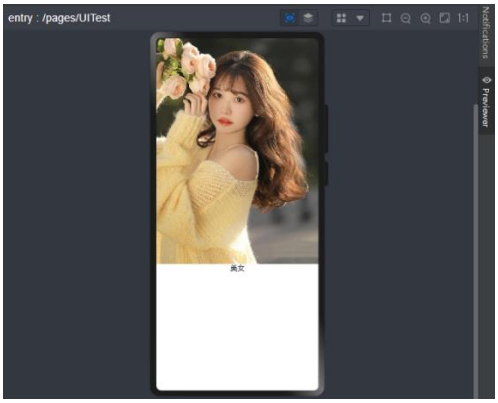
c. 变量或表达式也可以用于参数赋值，其中表达式返回的结果类型必须满足参数类型要求。

```
Image(this.imagePath)
Image('https://' + this.imageUrl)
Text(`count: ${this.count}`)
```

示例演示：

```
@Entry
@Component
struct UITest {
  build() {
    Column() {

Image('https://img0.baidu.com/it/u=110176915,621401482&fm=253&fmt=auto&app=138&f=JPEG?w=500&h=665')
        .height(500)
      Text('美女')
    }
  }
}
```

预览如下：



3.3.2.2 配置属性

属性方法以“.”链式调用的方式配置系统组件的样式和其他属性，建议每个属性方法单独写一行。

- **配置 Text 组件的字体大小。**

```
Text('test')
  .fontSize(12)
```

- **配置组件的多个属性。**

```
Image('test.jpg')
  .alt('error.jpg')
  .width(100)
  .height(100)
```

- **除了直接传递常量参数外，还可以传递变量或表达式。**

```
Text('hello')
  .fontSize(this.size)
Image('test.jpg')
  .width(this.count % 2 === 0 ? 100 : 200)
  .height(this.offset + 100)
```

- **对于系统组件，ArkUI 还为其属性预定义了一些枚举类型供开发者调用，枚举类型可以作为参数传递，但必须满足参数类型要求。**

例如，可以按以下方式配置 Text 组件的颜色和字体样式。

```
    Text('hello')
      .fontSize(20)
      .fontColor(Color.Red)
      .fontWeight(FontWeight.Bold)
```

示例：

```
    @Entry
@Component
struct UITest {
  textSize: number = 50;
  count:number = 2;
  imageOffset:number = 700;

  build() {
    Column(){
      Text("Hello ArkTS")
        .fontSize(this.textSize)
        .fontColor(Color.Red)
        .fontWeight(FontWeight.Bold)


Image('https://img0.baidu.com/it/u=110176915,621401482&fm=253&fmt=auto&app=138&f=JPEG?w=500&h=665')
        .width(this.count%2 === 0?500:200)
        .height(this.imageOffset + 100)
    }
      }
}
```

预览如下：

3.3.2.3 配置事件

事件方法以“.”链式调用的方式配置系统组件支持的事件，建议每个事件方法单独写一行。

- 使用箭头函数配置组件的事件方法。

```
Button('Click me')
  .onClick(() => {
    this.myText = 'ArkUI';
  })
```

- 使用匿名函数表达式配置组件的事件方法，要求使用 bind，以确保函数体中的 this 指向当前组件。

```
Button('add counter')
  .onClick(function(){
    this.counter += 2;
  }.bind(this))
```

- 使用组件的成员函数配置组件的事件方法。

```
myClickHandler(): void {
  this.counter += 2;
}
...
Button('add counter')
  .onClick(this.myClickHandler.bind(this))
```

示例：

```
    @Entry
@Component
struct UITest {
  @State textSize: number = 20;

  myClickHandler(): void {
    this.textSize += 10;
  }

  build() {
```

```
    Column() {
      Text("Hello ArkTS")
        .fontSize(this.textSize)
        .fontColor(Color.Red)
        .fontWeight(FontWeight.Bold)
      Divider()
      Button("增大字体")
        .height(50)
        .width(100)
        .margin(20)
        .onClick(this.myClickHandler.bind(this))
    }
  }
}
```

预览如下，每次点击“增大字体”按钮后，“Hello ArkTS”都会变大。

3.3.2.4 配置子组件

如果组件支持子组件配置，则需在尾随闭包"{...}"中为组件添加子组件的 UI 描述。Column、Row、Stack、Grid、List 等组件都是容器组件。

- 以下是简单的 Column 组件配置子组件的示例。

```
Column() {
  Text('Hello')
    .fontSize(100)
```

```
    Divider()
    Text(this.myText)
      .fontSize(100)
      .fontColor(Color.Red)
  }
```

- 容器组件均支持子组件配置，可以实现相对复杂的多级嵌套。

```
  Column() {
    Text('Hello')
      .fontSize(100)
    Divider()
    Text(this.myText)
      .fontSize(100)
      .fontColor(Color.Red)
  }
```

3.3.3 基础组件-Text

Text 组件是可以显示一段文本的组件。该组件从 API Version 7 开始支持，从 API version 9 开始，该接口支持在 ArkTS 卡片中使用。

3.3.3.1 用法

该组件使用方式如下：

```
Text(content?: string | Resource)
```

以上参数解释如下:

参数名	参数类型	必填	参数描述
content	string | Resource	否	文本内容。包含子组件 Span 时不生效，显示 Span 内容，并且此时 text 组件的样式不生效。

参数名	参数类型	必填	参数描述
			默认值：' '

Text 组件支持很多通用属性，如：width、height 等，还支持如下属性：

名称	参数类型	描述
textAlign	TextAlign	设置文本段落在水平方向的对齐方式 默认值：TextAlign.Start 说明： 文本段落宽度占满 Text 组件宽度。 可通过 align 属性控制文本段落在垂直方向上的位置，此组件中不可通过 align 属性控制文本段落在水平方向上的位置，即 align 属性中 Alignment.TopStart、Alignment.Top、Alignment.TopEnd 效果相同，控制内容在顶部。Alignment.Start、Alignment.Center、Alignment.End 效果相同，控制内容垂直居中。Alignment.BottomStart、Alignment.Bottom、Alignment.BottomEnd 效果相同，控制内容在底部。结合 TextAlign 属性可控制内容在水平方向的位置。 从 API version 9 开始，该接口支持在 ArkTS 卡片中使用。
textOverflow	{overflow: TextOverflow}	设置文本超长时的显示方式。 默认值：{overflow: TextOverflow.Clip} 说明： 文本截断是按字截断。例如，英文以单词为最小单位进行截断，若需要以字母为单位进行截断，可在字母间添加零宽空格：\u200B。 需配合 maxLines 使用，单独设置不生效。 从 API version 9 开始，该接口支持在 ArkTS 卡片中使用。
maxLines	number	设置文本的最大行数。 默认值：Infinity

名称	参数类型	描述
		说明： 默认情况下，文本是自动折行的，如果指定此参数，则文本最多不会超过指定的行。如果有多余的文本，可以通过 textOverflow 来指定截断方式。 从 API version 9 开始，该接口支持在 ArkTS 卡片中使用。
lineHeight	string | number | Resource	设置文本的文本行高，设置值不大于 0 时，不限制文本行高，自适应字体大小，Length 为 number 类型时单位为 fp。 从 API version 9 开始，该接口支持在 ArkTS 卡片中使用。

decoration	{
	type: TextDecorationType,
	color?: ResourceColor
	}
		设置文本装饰线样式及其颜色。
		默认值：{
		type: TextDecorationType.None,
		color：Color.Black
		}
		从 API version 9 开始，该接口支持在 ArkTS 卡片中使用。

baselineOffset	number | string	设置文本基线的偏移量，默认值 0。 从 API version 9 开始，该接口支持在 ArkTS 卡片中使用。 说明： 设置该值为百分比时，按默认值显示。
letterSpacing	number | string	设置文本字符间距。 从 API version 9 开始，该接口支持在 ArkTS 卡片中使用。 说明： 设置该值为百分比时，按默认值显示。
minFontSize	number | string | Resource	设置文本最小显示字号。 需配合 maxFontSize 以及 maxline 或布局

名称	参数类型	描述
		大小限制使用，单独设置不生效。 从 API version 9 开始，该接口支持在 ArkTS 卡片中使用。
maxFontSize	number | string | Resource	设置文本最大显示字号。 需配合 minFontSize 以及 maxline 或布局大小限制使用，单独设置不生效。 从 API version 9 开始，该接口支持在 ArkTS 卡片中使用。
textCase	TextCase	设置文本大小写。 默认值：TextCase.Normal 从 API version 9 开始，该接口支持在 ArkTS 卡片中使用。
copyOption9+	CopyOptions	组件支持设置文本是否可复制粘贴。 默认值：CopyOptions.None 该接口支持在 ArkTS 卡片中使用。 说明： 设置 copyOptions 为 CopyOptions.InApp 或者 CopyOptions.LocalDevice，长按文本，会弹出文本选择菜单，可选中文本并进行复制、全选操作。

3.3.3.2 示例

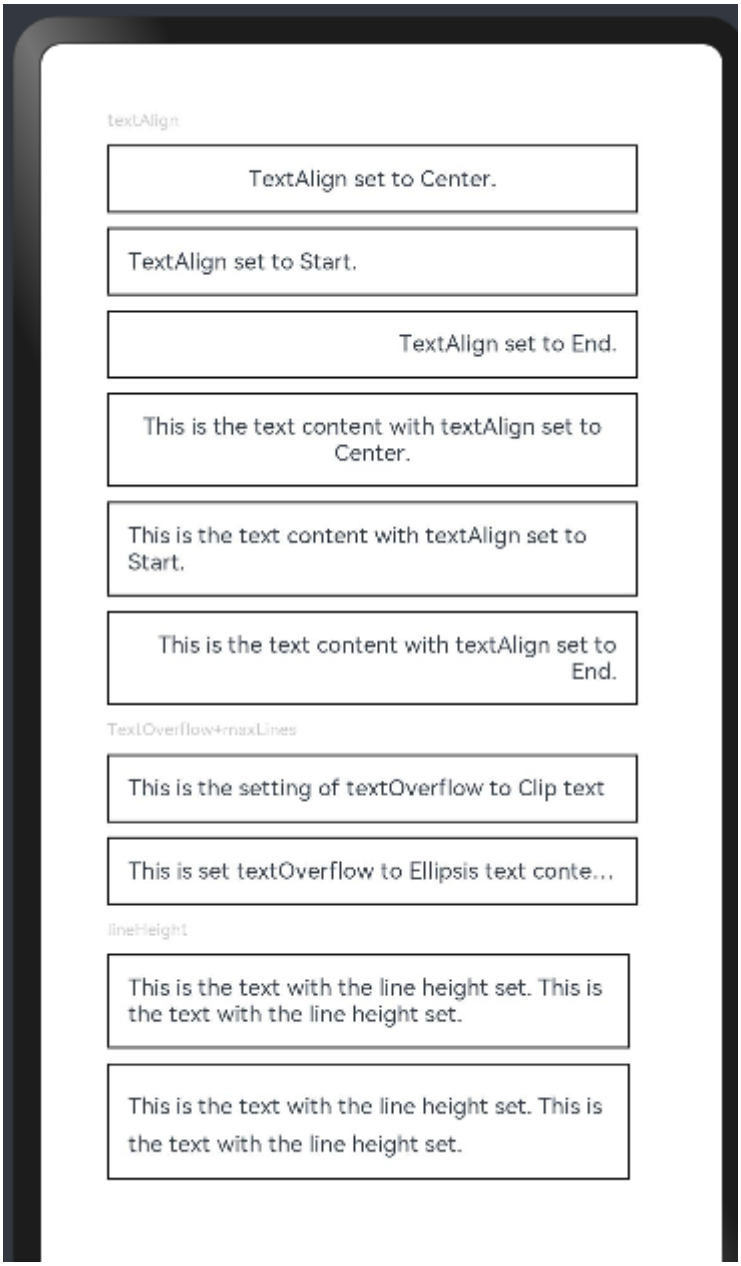
以下代码定义了一个名为 TextExample1 的组件，用于展示不同文本样式的效果，包括文本对齐、文本溢出处理和行高设置。

```
    @Entry  // 使用 @Entry 装饰器标识这是一个入口组件。
@Component  // 使用 @Component 装饰器定义一个新组件。
struct TextExample1 {  // 定义名为 TextExample1 的结构体，代表这个组件。

 build() {  // 定义 build 方法来构建 UI。
   Flex({  // 创建一个弹性布局容器。
     direction: FlexDirection.Column,  // 设置布局方向为垂直列。
     alignItems: ItemAlign.Start,  // 设置子项沿主轴的起始位置对齐。
```

```
    justifyContent: FlexAlign.SpaceBetween  // 设置子项间距均匀分布。
  }) {
    // 文本水平方向对齐方式设置
    // 单行文本
    Text('textAlign').fontSize(9).fontColor(0xCCCCCC)  // 创建一个文本组件，说明接下来的文本对齐设置。

    Text('TextAlign set to Center.')  // 创建一个文本组件，文本居中对齐。
      .textAlign(TextAlign.Center)  // 设置文本对齐方式为居中。
      .fontSize(12)  // 设置字体大小为 12。
      .border({ width: 1 })  // 设置边框宽度为 1。
      .padding(10)  // 设置内边距为 10。
      .width('100%')  // 设置宽度为 100%。

    Text('TextAlign set to Start.')  // 创建一个文本组件，文本起始对齐。
      .textAlign(TextAlign.Start)  // 设置文本对齐方式为起始对齐。
      .fontSize(12)  // 设置字体大小为 12。
      .border({ width: 1 })  // 设置边框宽度为 1。
      .padding(10)  // 设置内边距为 10。
      .width('100%')  // 设置宽度为 100%。

    Text('TextAlign set to End.')  // 创建一个文本组件，文本结束对齐。
      .textAlign(TextAlign.End)  // 设置文本对齐方式为结束对齐。
      .fontSize(12)  // 设置字体大小为 12。
      .border({ width: 1 })  // 设置边框宽度为 1。
      .padding(10)  // 设置内边距为 10。
      .width('100%')  // 设置宽度为 100%。

    // 多行文本
    Text('This is the text content with textAlign set to Center.')  // 创建一个多行文本组件，文本居中对齐。
      .textAlign(TextAlign.Center)  // 设置文本对齐方式为居中。
      .fontSize(12)  // 设置字体大小为 12。
      .border({ width: 1 })  // 设置边框宽度为 1。
      .padding(10)  // 设置内边距为 10。
      .width('100%')  // 设置宽度为 100%。

    Text('This is the text content with textAlign set to Start.')  // 创建一个多行文本组件，文本起始对齐。
      .textAlign(TextAlign.Start)  // 设置文本对齐方式为起始对齐。
      .fontSize(12)  // 设置字体大小为 12。
      .border({ width: 1 })  // 设置边框宽度为 1。
      .padding(10)  // 设置内边距为 10。
      .width('100%')  // 设置宽度为 100%。

    Text('This is the text content with textAlign set to End.')  // 创建一个多行文本组件，文本结束对齐。
      .textAlign(TextAlign.End)  // 设置文本对齐方式为结束对齐。
```

```
      .fontSize(12)  // 设置字体大小为 12。
      .border({ width: 1 })  // 设置边框宽度为 1。
      .padding(10)  // 设置内边距为 10。
      .width('100%')  // 设置宽度为 100%。

    // 文本超长时显示方式
    Text('TextOverflow+maxLines').fontSize(9).fontColor(0xCCCCCC)  // 创建一个文本组件，说明接下来的文本溢出设置。

    // 超出 maxLines 截断内容展示
    Text('This is the setting of textOverflow to Clip text content This is the setting of textOverflow to None text content. This is the setting of textOverflow to Clip text content This is the setting of textOverflow to None text content.')
      .textOverflow({ overflow: TextOverflow.Clip })  // 设置文本溢出方式为剪裁（Clip）。
      .maxLines(1)  // 设置最大行数为 1。
      .fontSize(12)  // 设置字体大小为 12。
      .border({ width: 1 })  // 设置边框宽度为 1。
      .padding(10)  // 设置内边距为 10。

    // 超出 maxLines 展示省略号
    Text('This is set textOverflow to Ellipsis text content This is set textOverflow to Ellipsis text content.'.split('')
      .join('\u200B'))
      .textOverflow({ overflow: TextOverflow.Ellipsis })  // 设置文本溢出方式为省略号（Ellipsis）。
      .maxLines(1)  // 设置最大行数为 1。
      .fontSize(12)  // 设置字体大小为 12。
      .border({ width: 1 })  // 设置边框宽度为 1。
      .padding(10)  // 设置内边距为 10。

    Text('lineHeight').fontSize(9).fontColor(0xCCCCCC)  // 创建一个文本组件，说明接下来的行高设置。

    // 设置文本的行高
    Text('This is the text with the line height set. This is the text with the line height set.')
      .fontSize(12)  // 设置字体大小为 12。
      .border({ width: 1 })  // 设置边框宽度为 1。
      .padding(10)  // 设置内边距为 10。
    Text('This is the text with the line height set. This is the text with the line height set.')
      .fontSize(12)  // 设置字体大小为 12。
      .border({ width: 1 })  // 设置边框宽度为 1。
      .padding(10)  // 设置内边距为 10。
      .lineHeight(20)  // 设置行高为 20。
  }.height(600).width(350).padding({ left: 35, right: 35, top: 35 })  // 设置容器的高度、宽度和内边距。
  }
}
```

以上代码预览如下：



3.3.4 容器组件-Column

Column 容器组件是沿垂直方向布局的容器。该组件从 API Version 7 开始支持，从 API version 9 开始，该接口支持在 ArkTS 卡片中使用。其可以包含子组件。

3.3.4.1 用法

Column 组件用法如下：

```
Column(value?: {space?: string | number})
```

以上参数解释如下：

参数名	参数类型	必填	参数描述
space	string | number	否	纵向布局元素垂直方向间距。 从 API version 9 开始，space 为负数或者 justifyContent 设置为 FlexAlign.SpaceBetween、FlexAlign.SpaceAround、FlexAlign.SpaceEvenly 时不生效。 默认值：0 说明： 可选值为大于等于 0 的数字，或者可以转换为数字的字符串。

Column 组件支持很多通用属性，如：width、height 等，还支持如下属性：

名称	参数类型	描述
alignItems	HorizontalAlign	设置子组件在水平方向上的对齐格式。 默认值：HorizontalAlign.Center 从 API version 9 开始，该接口支持在 ArkTS 卡片中使用。
justifyContent8+	FlexAlign	设置子组件在垂直方向上的对齐格式。 默认值：FlexAlign.Start 从 API version 9 开始，该接口支持在 ArkTS 卡片中使用。

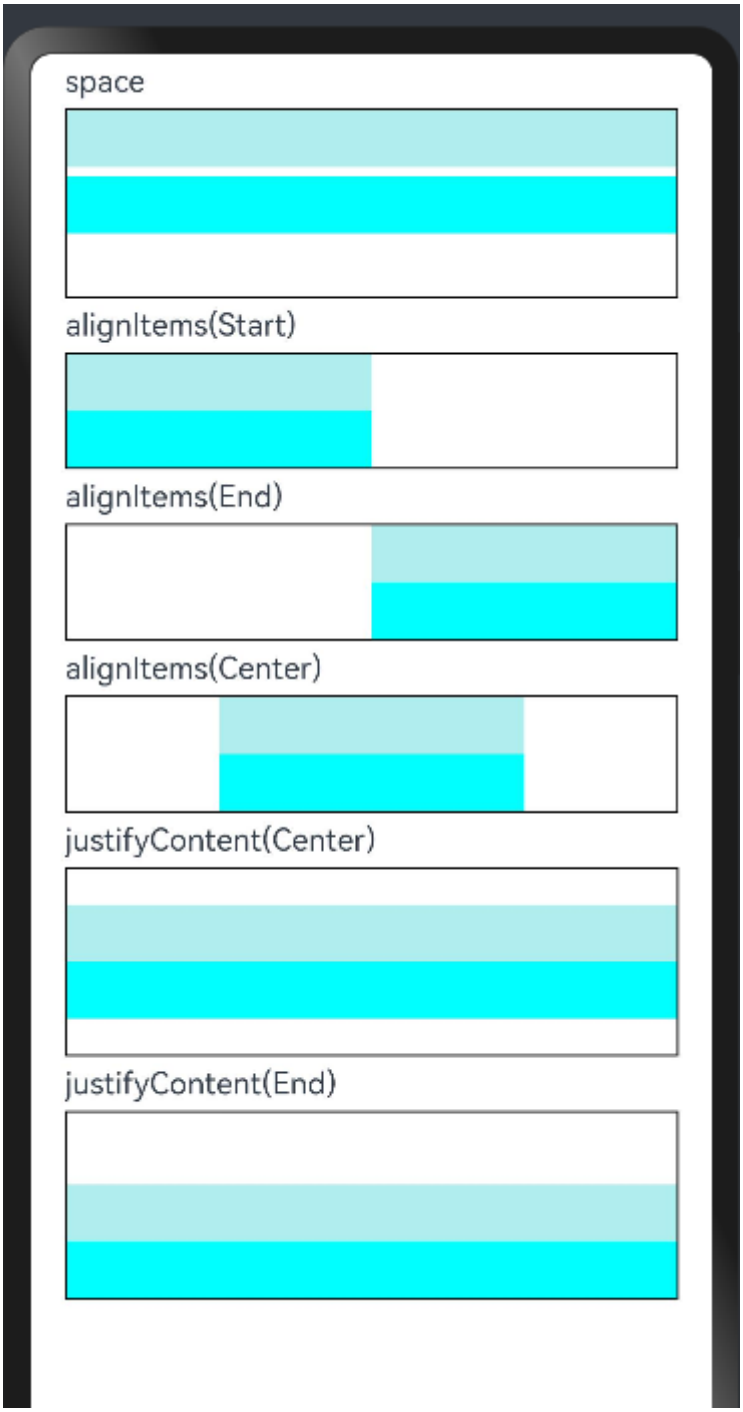
3.3.4.2 示例

以下代码定义了一个名为 ColumnExample 的组件，用于展示 Column 布局的不同特性，包括子元素间距、对齐方式和背景颜色。

```
@Entry  // 使用 @Entry 装饰器标识这是一个入口组件。
@Component  // 使用 @Component 装饰器定义一个新组件。
struct ColumnExample {  // 定义名为 ColumnExample 的结构体，代表这个组件。

  build() {  // 定义 build 方法来构建 UI。
    Column({ space: 5 }) {  // 创建一个 Column 组件，设置子元素间的垂直间距为 5。
      Text('space').width('90%')  // 创建一个 Text 组件，说明接下来的内容与 space 属性相关。

      Column({ space: 5 }) {  // 创建一个内部 Column 组件，再次设置子元素间的垂直间距为 5。
        Column().width('100%').height(30).backgroundColor(0xAFEEEE)  // 创建一个 Column 子组件，设置宽度、高度和背景颜色为浅蓝色。
        Column().width('100%').height(30).backgroundColor(0x00FFFF)  // 创建另一个 Column 子组件，设置宽度、高度和背景颜色为青色。
      }.width('90%').height(100).border({ width: 1 })  // 为这个内部 Column 设置宽度、高度和边框。

      // 设置子元素水平方向对齐方式
      Text('alignItems(Start)').width('90%')  // 创建一个 Text 组件，说明接下来的内容与水平起始对齐相关。
      Column() {  // 创建一个 Column 组件。
        Column().width('50%').height(30).backgroundColor(0xAFEEEE)  // 创建一个子 Column，设置宽度、高度和背景颜色为浅蓝色。
        Column().width('50%').height(30).backgroundColor(0x00FFFF)  // 创建另一个子 Column，设置宽度、高度和背景颜色为青色。
      }.alignItems(HorizontalAlign.Start).width('90%').border({ width: 1 })  // 为这个 Column 设置子元素水平起始对齐、宽度和边框。

      Text('alignItems(End)').width('90%')  // 创建一个 Text 组件，说明接下来的内容与水平结束对齐相关。
      Column() {  // 创建一个 Column 组件。
        Column().width('50%').height(30).backgroundColor(0xAFEEEE)  // 创建子 Column 组件，设置同上。
        Column().width('50%').height(30).backgroundColor(0x00FFFF)  // 创建另一个子 Column 组件，设置同上。
      }.alignItems(HorizontalAlign.End).width('90%').border({ width: 1 })  // 为这个 Column 设置子元素水平结束对齐、宽度和边框。

      Text('alignItems(Center)').width('90%')  // 创建一个 Text 组件，说明接下来的内容与水平居中对齐相关。
      Column() {  // 创建一个 Column 组件。
        Column().width('50%').height(30).backgroundColor(0xAFEEEE)  // 创建子 Column 组件，设置同上。
        Column().width('50%').height(30).backgroundColor(0x00FFFF)  // 创建另一个子 Column 组件，设置同上。
```

```
    }.alignItems(HorizontalAlign.Center).width('90%').border({ width: 1 })  // 为这个 Column 设置子元素水平居中对齐、宽度和边框。

    // 设置子元素垂直方向的对齐方式
    Text('justifyContent(Center)').width('90%')  // 创建一个 Text 组件，说明接下来的内容与垂直居中对齐相关。
    Column() {  // 创建一个 Column 组件。
      Column().width('90%').height(30).backgroundColor(0xAFEEEE)  // 创建子 Column 组件，设置宽度、高度和背景颜色为浅蓝色。
      Column().width('90%').height(30).backgroundColor(0x00FFFF)  // 创建另一个子 Column 组件，设置宽度、高度和背景颜色为青色。
    }.height(100).border({ width: 1 }).justifyContent(FlexAlign.Center)  // 为这个 Column 设置高度、边框和子元素垂直居中对齐。

    Text('justifyContent(End)').width('90%')  // 创建一个 Text 组件，说明接下来的内容与垂直结束对齐相关。
    Column() {  // 创建一个 Column 组件。
      Column().width('90%').height(30).backgroundColor(0xAFEEEE)  // 创建子 Column 组件，设置同上。
      Column().width('90%').height(30).backgroundColor(0x00FFFF)  // 创建另一个子 Column 组件，设置同上。
    }.height(100).border({ width: 1 }).justifyContent(FlexAlign.End)  // 为这个 Column 设置高度、边框和子元素垂直结束对齐。
  }.width('100%').padding({ top: 5 })  // 为最外层 Column 设置宽度和顶部内边距。
 }
}
```

以上代码预览如下：



3.3.5 组件组件-Row

Row 容器组件是沿水平方向布局容器。该组件从 API Version 7 开始支持，从 API version 9 开始，该接口支持在 ArkTS 卡片中使用。可以包含子组件。

3.3.5.1 用法

Row 用法如下：

```
Row(value?:{space?: number | string })
```

以上参数解释如下：

参数名	参数类型	必填	参数描述
space	string | number	否	横向布局元素间距。 从 API version 9 开始，space 为负数或者 justifyContent 设置为 FlexAlign.SpaceBetween、FlexAlign.SpaceAround、FlexAlign.SpaceEvenly 时不生效。 默认值：0，单位 vp 说明： 可选值为大于等于 0 的数字，或者可以转换为数字的字符串。

Row 支持的属性如下：

名称	参数类型	描述
alignItems	VerticalAlign	设置子组件在垂直方向上的对齐格式。 默认值：VerticalAlign.Center 从 API version 9 开始，该接口支持在 ArkTS 卡片中使用。
justifyContent8+	FlexAlign	设置子组件在水平方向上的对齐格式。 默认值：FlexAlign.Start 从 API version 9 开始，该接口支持在 ArkTS 卡片中使用。

3.3.5.2 示例

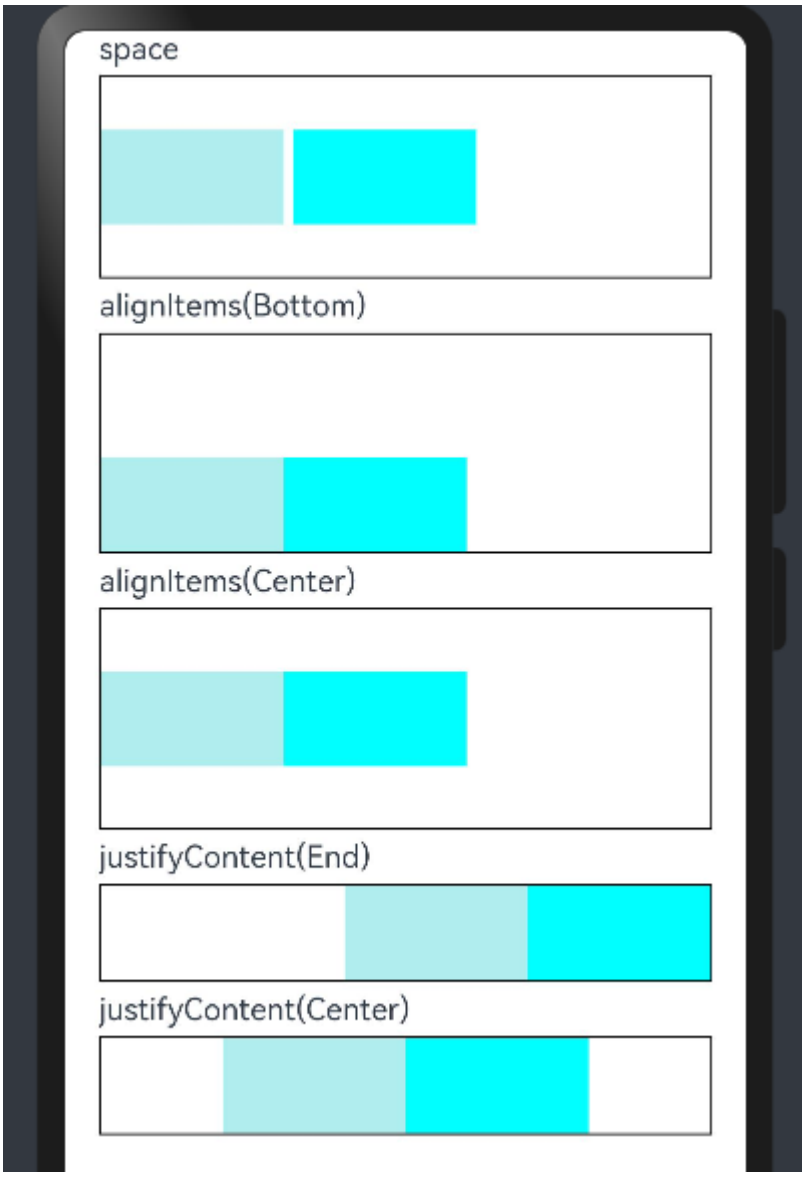
如下代码定义了一个名为 RowExample 的组件，用于展示 Row 布局的不同特性，包括子元素间距、垂直对齐方式和水平对齐方式。

```
    @Entry  // 使用 @Entry 装饰器标识这是一个入口组件。
@Component  // 使用 @Component 装饰器定义一个新组件。
```

```
struct RowExample {  // 定义名为 RowExample 的结构体，代表这个组件。

  build() {  // 定义 build 方法来构建 UI。
    Column({ space: 5 }) {  // 创建一个 Column 组件，设置子元素间的垂直间距为 5。
      Text('space').width('90%')  // 创建一个 Text 组件，说明接下来的内容与 space 属性相关。

      Row({ space: 5 }) {  // 创建一个 Row 组件，设置子元素间的水平间距为 5。
        Row().width('30%').height(50).backgroundColor(0xAFEEEE)  // 创建一个子 Row 组件，设置宽度、高度和背景颜色为浅蓝色。
        Row().width('30%').height(50).backgroundColor(0x00FFFF)  // 创建另一个子 Row 组件，设置宽度、高度和背景颜色为青色。
      }.width('90%').height(107).border({ width: 1 })  // 为这个 Row 设置宽度、高度和边框。

      // 设置子元素垂直方向对齐方式
      Text('alignItems(Bottom)').width('90%')  // 创建一个 Text 组件，说明接下来的内容与垂直底部对齐相关。
      Row() {  // 创建一个 Row 组件。
        Row().width('30%').height(50).backgroundColor(0xAFEEEE)  // 创建子 Row 组件，设置同上。
        Row().width('30%').height(50).backgroundColor(0x00FFFF)  // 创建另一个子 Row 组件，设置同上。
      }.width('90%').alignItems(VerticalAlign.Bottom).height('15%').border({ width: 1 })  // 为这个 Row 设置垂直底部对齐、宽度、高度和边框。

      Text('alignItems(Center)').width('90%')  // 创建一个 Text 组件，说明接下来的内容与垂直居中对齐相关。
      Row() {  // 创建一个 Row 组件。
        Row().width('30%').height(50).backgroundColor(0xAFEEEE)  // 创建子 Row 组件，设置同上。
        Row().width('30%').height(50).backgroundColor(0x00FFFF)  // 创建另一个子 Row 组件，设置同上。
      }.width('90%').alignItems(VerticalAlign.Center).height('15%').border({ width: 1 })  // 为这个 Row 设置垂直居中对齐、宽度、高度和边框。

      // 设置子元素水平方向对齐方式
      Text('justifyContent(End)').width('90%')  // 创建一个 Text 组件，说明接下来的内容与水平结束对齐相关。
      Row() {  // 创建一个 Row 组件。
        Row().width('30%').height(50).backgroundColor(0xAFEEEE)  // 创建子 Row 组件，设置同上。
        Row().width('30%').height(50).backgroundColor(0x00FFFF)  // 创建另一个子 Row 组件，设置同上。
      }.width('90%').border({ width: 1 }).justifyContent(FlexAlign.End)  // 为这个 Row 设置水平结束对齐、宽度和边框。

      Text('justifyContent(Center)').width('90%')  // 创建一个 Text 组件，说明接下来的内容与水平居中对齐相关。
      Row() {  // 创建一个 Row 组件。
        Row().width('30%').height(50).backgroundColor(0xAFEEEE)  // 创建子 Row 组件，设置同上。
        Row().width('30%').height(50).backgroundColor(0x00FFFF)  // 创建另一个子 Row 组件，设置同上。
```

```
    }.width('90%').border({ width: 1 }).justifyContent(FlexAlign.Center)  // 为这个 Row 设置水平居中对齐、宽度和边框。
  }.width('100%')  // 为最外层 Column 设置宽度为 100%。
 }
}
```

以上代码预览如下：



3.3.6 自定义组件

在 ArkUI 中，UI 显示的内容均为组件，由框架直接提供的称为系统组件，由开发者定义的称为自定义组件。在进行 UI 界面开发时，通常不是简单的将系统组件进行组合使用，而是需要考虑代码可复用性、业务逻辑与 UI 分离，后续版本演进等因素。因此，将 UI 和部分业务逻辑封装成自定义组件是不可或缺的能力。

自定义组件具有以下特点：

- 可组合：允许开发者组合使用系统组件、及其属性和方法。

- 可重用：自定义组件可以被其他组件重用，并作为不同的实例在不同的父组件或容器中使用。
- 数据驱动 UI 更新：通过状态变量的改变，来驱动 UI 的刷新。

3.3.6.1 自定义组件的基本用法

以下示例展示了自定义组件的基本用法。

```
@Component
struct HelloComponent {
  @State message: string = 'Hello, World!';

  build() {
    // HelloComponent 自定义组件组合系统组件 Row 和 Text
    Row() {
      Text(this.message)
        .onClick(() => {
          // 状态变量 message 的改变驱动 UI 刷新，UI 从'Hello, World!'刷新为'Hello, ArkUI!'
          this.message = 'Hello, ArkUI!';
        })
    }
  }
}
```

HelloComponent 可以在其他自定义组件中的 build()函数中多次创建，实现自定义组件的重用。

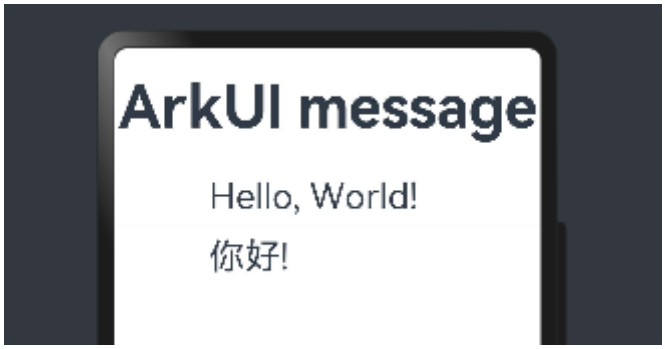
```
@Entry
@Component
struct ParentComponent {
  build() {
    Column() {
      Text('ArkUI message')
      HelloComponent({ message: 'Hello, World!' });
      Divider()
      HelloComponent({ message: '你好!' });
    }
  }
```

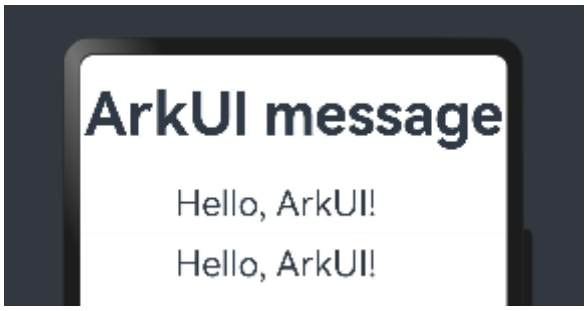
```
    }
```

以上代码写入一个 arkts 文件中，并设置对应的字体样式，如下：

```
    @Component
struct HelloComponent {
  @State message: string = 'Hello, World!';

  build() {
    // HelloComponent 自定义组件组合系统组件 Row 和 Text
    Row() {
      Text(this.message)
        .height(50)
        .width(200)
        .fontSize(30)
        .onClick(() => {
          // 状态变量 message 的改变驱动 UI 刷新，UI 从'Hello, World!'刷新为'Hello, ArkUI!'
          this.message = 'Hello, ArkUI!';
        })
    }
  }
}

@Entry
@Component
struct ParentComponent {
  build() {
    Column() {
      Text('ArkUI message')
        .height(100)
        .fontSize(50)
        .fontWeight(FontWeight.Bold)
      HelloComponent({ message: 'Hello, World!' });
      Divider()
      HelloComponent({ message: '你好!' });
    }
  }
}
```

预览如下，点击“Hello,World!”或者“你好”都会改变为“Hello, ArkUI!”做到了自定义组件复用。





3.3.6.2 自定义组件的基本结构

- **struct**

自定义组件基于 struct 实现，struct + 自定义组件名 + {...}的组合构成自定义组件，不能有继承关系。对于 struct 的实例化，可以省略 new。

注意：自定义组件名、类名、函数名不能和系统组件名相同。

- **@Component**

@Component 装饰器仅能装饰 struct 关键字声明的数据结构。struct 被@Component 装饰后具备组件化的能力，需要实现 build 方法描述 UI，一个 struct 只能被一个@Component 装饰。

```
@Component
struct MyComponent {
}
```

注意：从 API version 9 开始，该装饰器支持在 ArkTS 卡片中使用。

- **build()函数**

build()函数用于定义自定义组件的声明式 UI 描述，自定义组件必须定义 build()函数。

```
@Component
struct MyComponent {
  build() {
  }
}
```

- **@Entry**

@Entry 装饰的自定义组件将作为 UI 页面的入口。在单个 UI 页面中，最多可以使用@Entry 装饰一个自定义组件。@Entry 可以接受一个可选的 LocalStorage 的参数。

```
@Entry
@Component
struct MyComponent {
}
```

注意：从 API version 9 开始，该装饰器支持在 ArkTS 卡片中使用。

3.3.6.3 成员函数/变量

自定义组件除了必须要实现 build()函数外，还可以实现其他成员函数，成员函数具有以下约束：

- 不支持静态函数。
- 成员函数的访问是私有的。

自定义组件可以包含成员变量，成员变量具有以下约束：

- 不支持静态成员变量。
- 所有成员变量都是私有的，变量的访问规则与成员函数的访问规则相同。
- 自定义组件的成员变量本地初始化有些是可选的，有些是必选的。具体是否需要本地初始化，是否需要从父组件通过参数传递初始化子组件的成员变量，请参考状态管理。

3.3.6.4 自定义组件的参数规定

从上文的示例中，我们已经了解到，可以在 build 方法或者@Builder 装饰的函数里创建自定义组件，在创建自定义组件的过程中，根据装饰器的规则来初始化自定义组件的参数。

```
@Component
```

```
struct MyComponent {
  private countDownFrom: number = 0;
  private color: Color = Color.Blue;

  build() {
  }
}

@Entry
@Component
struct ParentComponent {
  private someColor: Color = Color.Pink;

  build() {
    Column() {
      // 创建 MyComponent 实例，并将创建 MyComponent 成员变量 countDownFrom 初始化为10，将成员变量 color 初始化为 this.someColor
      MyComponent({ countDownFrom: 10, color: this.someColor })
    }
  }
}
```

3.3.6.5 build()函数

所有声明在 build()函数的语言，我们统称为 UI 描述，UI 描述需要遵循以下规则：

- **@Entry 装饰的自定义组件，其 build()函数下的根节点唯一且必要，且必须为容器组件，其中 ForEach 禁止作为根节点。@Component 装饰的自定义组件，其 build()函数下的根节点唯一且必要，可以为非容器组件，其中 ForEach 禁止作为根节点。**

```
@Entry
@Component
struct MyComponent {
  build() {
    // 根节点唯一且必要，必须为容器组件
```

```
    Row() {
      ChildComponent()
    }
  }
}

@Component
struct ChildComponent {
  build() {
    // 根节点唯一且必要，可为非容器组件
    Image('test.jpg')
  }
}
```

- **不允许声明本地变量，反例如下。**

```
build() {
  // 反例：不允许声明本地变量
  let a: number = 1;
}
```

- **不允许在 UI 描述里直接使用 console.info，但允许在方法或者函数里使用，反例如下。**

```
build() {
  // 反例：不允许 console.info
  console.info('print debug log');
}
```

- **不允许创建本地的作用域，反例如下。**

```
build() {
  // 反例：不允许本地作用域
  {
    ...
  }
}
```

- **不允许调用没有用@Builder 装饰的方法，允许系统组件的参数是 TS 方法的返回值。**

```
@Component
struct ParentComponent {
  doSomeCalculations() {
  }

  calcTextValue(): string {
    return 'Hello World';
  }

  @Builder doSomeRender() {
    Text(`Hello World`)
  }

  build() {
    Column() {
      // 反例：不能调用没有用@Builder 装饰的方法
      this.doSomeCalculations();
      // 正例：可以调用
      this.doSomeRender();
      // 正例：参数可以为调用 TS 方法的返回值
      Text(this.calcTextValue())
    }
  }
}
```

- **不允许 switch 语法，如果需要使用条件判断，请使用 if。反例如下。**

```
build() {
  Column() {
    // 反例：不允许使用 switch 语法
    switch (expression) {
      case 1:
        Text('...')
        break;
```

```
      case 2:
        Image('...')
        break;
      default:
        Text('...')
        break;
    }
  }
}
```

- **不允许使用表达式，反例如下。**

```
build() {
  Column() {
    // 反例：不允许使用表达式
    (this.aVar > 10) ? Text('...') : Image('...')
  }
}
```

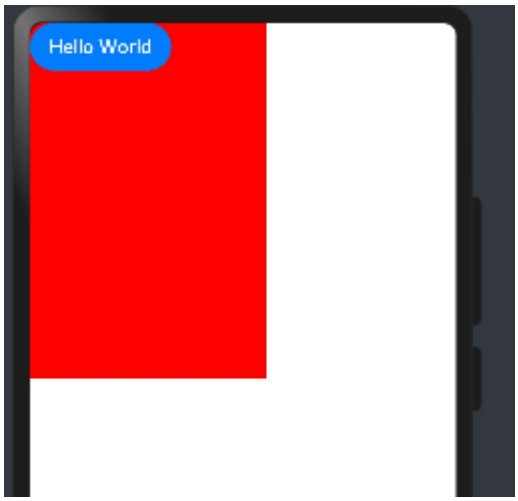
3.3.6.6 自定义组件通用样式

自定义组件通过“.”链式调用的形式设置通用样式。

```
@Component
struct MyComponent2 {
  build() {
    Button(`Hello World`)
  }
}

@Entry
@Component
struct MyComponent {
  build() {
    Row() {
      MyComponent2()
```

```
        .width(200)
        .height(300)
        .backgroundColor(Color.Red)
    }
  }
}
```

以上自定义组件预览效果如下，可以看到 ArkUI 给自定义组件设置样式时，相当于给 MyComponent2 套了一个不可见的容器组件，而这些样式是设置在容器组件上的，而非直接设置给 MyComponent2 的 Button 组件。通过渲染结果我们可以很清楚的看到，背景颜色红色并没有直接生效在 Button 上，而是生效在 Button 所处的开发者不可见的容器组件上。



3.3.7 页面和自定义组件生命周期

先明确自定义组件和页面的关系：

- 自定义组件：@Component 装饰的 UI 单元，可以组合多个系统组件实现 UI 的复用。
- 页面：即应用的 UI 页面。可以由一个或者多个自定义组件组成，@Entry 装饰的自定义组件为页面的入口组件，即页面的根节点，一个页面有且仅能有一个@Entry。只有被@Entry 装饰的组件才可以调用页面的生命周期。

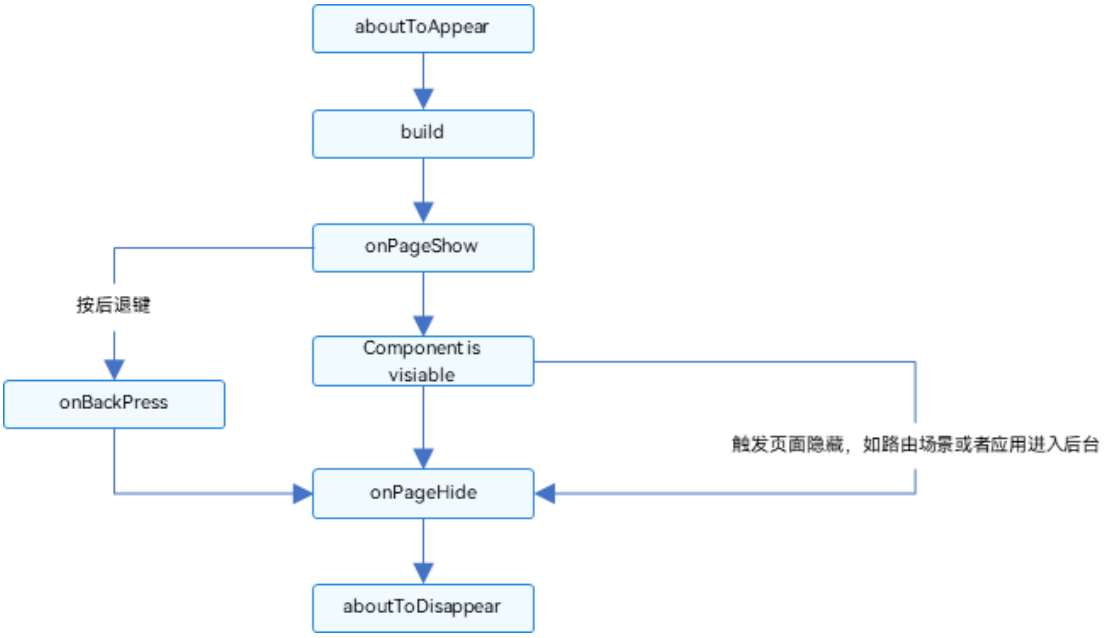
页面生命周期，即被@Entry 装饰的组件生命周期，提供以下生命周期接口：

- onPageShow：页面每次显示时触发。
- onPageHide：页面每次隐藏时触发一次。
- onBackPress：当用户点击返回按钮时触发。

组件生命周期，即一般用@Component 装饰的自定义组件的生命周期，提供以下生命周期接口：

- aboutToAppear：组件即将出现时回调该接口，具体时机为在创建自定义组件的新实例后，在执行其 build()函数之前执行。
- aboutToDisappear：在自定义组件即将析构销毁时执行。

生命周期流程如下图所示，下图展示的是被@Entry 装饰的组件（首页）生命周期。



需要注意的是，部分生命周期回调函数仅对@Entry 修饰的自定义组件生效，它们分别是：onPageShow、onPageHide、onBackPress。根据上面的流程图，我们从自定义组件的初始创建、重新渲染和删除来详细解释。

3.3.7.1 自定义组件的创建和渲染流程

1) 自定义组件的创建：自定义组件的实例由 ArkUI 框架创建。

2) 初始化自定义组件的成员变量：通过本地默认值或者构造方法传递参数来初始化自定义组件的成员变量，初始化顺序为成员变量的定义顺序。

3) 如果开发者定义了 aboutToAppear，则执行 aboutToAppear 方法。

4) 在首次渲染的时候，执行 build 方法渲染系统组件，如果子组件为自定义组件，则创建自定义组件的实例。在执行 build()函数的过程中，框架会观察每个状态变量的读取状态，将保存两个 map：

a.状态变量 -> UI 组件（包括 ForEach 和 if）。

b. UI 组件 -> 此组件的更新函数，即一个 lambda 方法，作为 build()函数的子集，创建对应的 UI 组件并执行其属性方法，示意如下。

```
build() {
  ...
  this.observeComponentCreation(() => {
    Button.create();
  })

  this.observeComponentCreation(() => {
    Text.create();
  })
  ...
}
```

当应用在后台启动时，此时应用进程并没有销毁，所以仅需要执行 onPageShow。

3.3.7.2 自定义组件重新渲染

当事件句柄被触发（比如设置了点击事件，即触发点击事件）改变了状态变量时，或者 LocalStorage / AppStorage 中的属性更改，并导致绑定的状态变量更改其值时：

1) 框架观察到了变化，将启动重新渲染。

2) 根据框架持有的两个 map（自定义组件的创建和渲染流程中第 4 步），框架可以知道该状态变量管理了哪些 UI 组件，以及这些 UI 组件对应的更新函数。执行这些 UI 组件的更新函数，实现最小化更新。

3.3.7.3 自定义组件的删除

如果 if 组件的分支改变，或者 ForEach 循环渲染中数组的个数改变，组件将被删除：

1) 在删除组件之前，将调用其 aboutToDisappear 生命周期函数，标记着该节点将要被销毁。ArkUI 的节点删除机制是：后端节点直接从组件树上摘下，后端节点被销毁，对前端节点解引用，当前端节点已经没有引用时，将被 JS 虚拟机垃圾回收。

2) 自定义组件和它的变量将被删除，如果其有同步的变量，比如@Link、@Prop、@StorageLink，将从同步源上取消注册。

不建议在生命周期 aboutToDisappear 内使用 async await，如果在生命周期的 aboutToDisappear 使用异步操作（Promise 或者回调方法），自定义组件将被保留在 Promise 的闭包中，直到回调方法被执行完，这个行为阻止了自定义组件的垃圾回收。

以下示例展示了生命周期的调用时机：

```
// Index.ets
import router from '@ohos.router';

@Entry
@Component
struct MyComponent {
  @State showChild: boolean = true;

  // 只有被@Entry 装饰的组件才可以调用页面的生命周期
  onPageShow() {
    console.info('Index onPageShow');
  }
  // 只有被@Entry 装饰的组件才可以调用页面的生命周期
  onPageHide() {
    console.info('Index onPageHide');
  }

  // 只有被@Entry 装饰的组件才可以调用页面的生命周期
  onBackPress() {
    console.info('Index onBackPress');
  }

  // 组件生命周期
  aboutToAppear() {
    console.info('MyComponent aboutToAppear');
  }

  // 组件生命周期
  aboutToDisappear() {
```

```
    console.info('MyComponent aboutToDisappear');
  }

  build() {
    Column() {
      // this.showChild 为 true，创建 Child 子组件，执行 Child aboutToAppear
      if (this.showChild) {
        Child()
      }
      // this.showChild 为 false，删除 Child 子组件，执行 Child aboutToDisappear
      Button('create or delete Child').onClick(() => {
        this.showChild = false;
      })
      // push 到 Page2 页面，执行 onPageHide
      Button('push to next page')
        .onClick(() => {
          router.pushUrl({ url: 'pages/Page2' });
        })
    }

  }
}

@Component
struct Child {
  @State title: string = 'Hello World';
  // 组件生命周期
  aboutToDisappear() {
    console.info('[lifeCycle] Child aboutToDisappear')
  }
  // 组件生命周期
  aboutToAppear() {
    console.info('[lifeCycle] Child aboutToAppear')
  }
```

```
    build() {
      Text(this.title).fontSize(50).onClick(() => {
        this.title = 'Hello ArkUI';
      })
    }
  }
```

以上示例中，Index 页面包含两个自定义组件，一个是被@Entry 装饰的 MyComponent，也是页面的入口组件，即页面的根节点；一个是 Child，是 MyComponent 的子组件。只有@Entry 装饰的节点才可以使页面级别的生命周期方法生效，所以 MyComponent 中声明了当前 Index 页面的页面生命周期函数。MyComponent 和其子组件 Child 也同时也声明了组件的生命周期函数。

- 应用冷启动的初始化流程为：MyComponent aboutToAppear --> MyComponent build --> Child aboutToAppear --> Child build --> Child build 执行完毕 --> MyComponent build 执行完毕 --> Index onPageShow。

- 点击“delete Child”，if 绑定的 this.showChild 变成 false，删除 Child 组件，会执行 Child aboutToDisappear 方法。

- 点击“push to next page”，调用 router.pushUrl 接口，跳转到另外一个页面，当前 Index 页面隐藏，执行页面生命周期 Index onPageHide。此处调用的是 router.pushUrl 接口，Index 页面被隐藏，并没有销毁，所以只调用 onPageHide。跳转到新页面后，执行初始化新页面的生命周期的流程。

- 如果调用的是 router.replaceUrl，则当前 Index 页面被销毁，执行的生命周期流程将变为：Index onPageHide --> MyComponent aboutToDisappear --> Child aboutToDisappear。上文已经提到，组件的销毁是从组件树上直接摘下子树，所以先调用父组件的 aboutToDisappear，再调用子组件的 aboutToDisappear，然后执行初始化新页面的生命周期流程。

- 点击返回按钮，触发页面生命周期 Index onBackPress，且触发返回一个页面后会导致当前 Index 页面被销毁。

- 最小化应用或者应用进入后台，触发 Index onPageHide。当前 Index 页面没有被销毁，所以并不会执行组件的 aboutToDisappear。应用回到前台，执行 Index onPageShow。

- 退出应用，执行 Index onPageHide --> MyComponent aboutToDisappear --> Child aboutToDisappear。

3.3.8 @Builder 装饰器-自定义构建函数

前面介绍了如何创建一个自定义组件。该自定义组件内部 UI 结构固定，仅与使用方进行数据传递。ArkUI 还提供了一种更轻量的 UI 元素复用机制@Builder，@Builder 所装饰的函数遵循 build()函数语法规则，开发者可以将重复使用的 UI 元素抽象成一个方法，在 build 方法里调用。

为了简化语言，我们将@Builder 装饰的函数也称为“自定义构建函数”。从 API version 9 开始，该装饰器支持在 ArkTS 卡片中使用。

3.3.8.1 装饰器使用说明

1. 自定义组件内自定义构建函数

定义的语法：

```
@Builder MyBuilderFunction(){ ... }
```

使用方法：

```
this.MyBuilderFunction(){ ... }
```

- 允许在自定义组件内定义一个或多个@Builder 方法，该方法被认为是该组件的私有、特殊类型的成员函数。
- 自定义构建函数可以在所属组件的 build 方法和其他自定义构建函数中调用，但不允许在组件外调用。
- 在自定义函数体中，this 指代当前所属组件，组件的状态变量可以在自定义构建函数内访问。建议通过 this 访问自定义组件的状态变量而不是参数传递。

2. 全局自定义构建函数

定义的语法：

```
@Builder function MyGlobalBuilderFunction(){ ... }
```

使用方法：

```
MyGlobalBuilderFunction()
```

- 全局的自定义构建函数可以被整个应用获取，不允许使用 this 和 bind 方法。
- 如果不涉及组件状态变化，建议使用全局的自定义构建方法。

3.3.8.2 参数传递规则

自定义构建函数的参数传递有按值传递和按引用传递两种，均需遵守以下规则：

- 参数的类型必须与参数声明的类型一致，不允许 undefined、null 和返回 undefined、null 的表达式。
- 在自定义构建函数内部，不允许改变参数值。如果需要改变参数值，且同步回调用点，建议使用@Link。
- @Builder 内 UI 语法遵循 UI 语法规则。

1. 按引用传递参数

按引用传递参数时，传递的参数可为状态变量，且状态变量的改变会引起@Builder 方法内的 UI 刷新。ArkUI 提供$$作为按引用传递参数的范式。

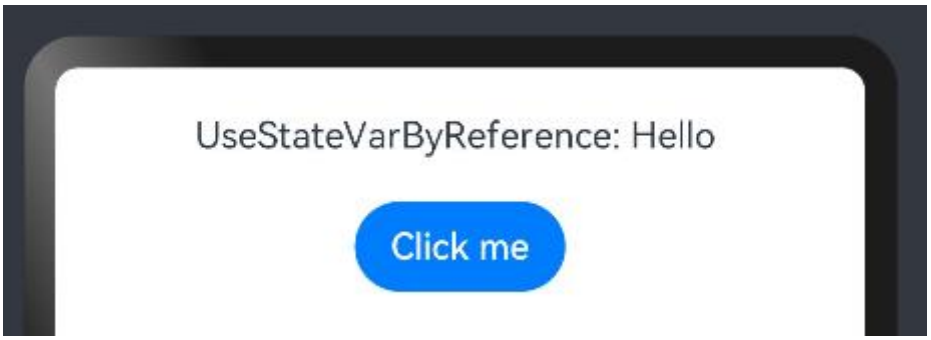
```
ABuilder( $$ : { paramA1: string, paramB1 : string } );
```

如下案例：

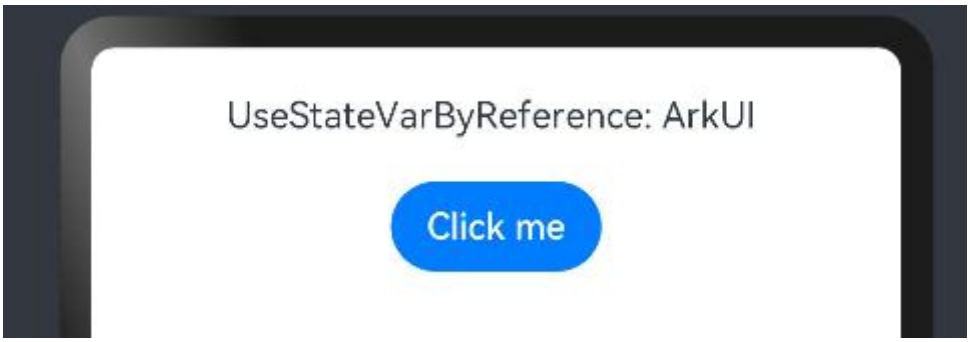
```
@Builder function ABuilder($$: { paramA1: string }) {
  Row() {
    Text(`UseStateVarByReference: ${$$.paramA1} `)
      .margin(20)
  }
}
@Entry
@Component
struct Parent {
  @State label: string = 'Hello';
  build() {
    Column() {
      Divider()
      // 在 Parent 组件中调用 ABuilder 的时候，将 this.label 引用传递给 ABuilder
      ABuilder({ paramA1: this.label })
      Button('Click me').onClick(() => {
        // 点击“Click me”后，UI 从“Hello”刷新为“ArkUI”
```

```
      this.label = 'ArkUI';
    })
  }
 }
}
```

以上代码对应的预览如下：



当点击“Click me”按钮时，预览如下：



2. 按值传递参数

调用@Builder 装饰的函数默认按值传递。当传递的参数为状态变量时，状态变量的改变不会引起@Builder 方法内的 UI 刷新。所以当使用状态变量的时候，推荐使用按引用传递。

改写如上案例：

```
    @Builder function ABuilder(paramA1: string) {
 Row() {
  Text(`UseStateVarByReference: ${paramA1} `)
    .margin(20)
 }
}
@Entry
@Component
struct Parent {
 @State label: string = 'Hello';
 build() {
  Column() {
```

```
    Divider()
    // 在 Parent 组件中调用 ABuilder 的时候，将 this.label 引用传递给 ABuilder
    ABuilder(this.label)
    Button('Click me').onClick(() => {
      // 点击“Click me”后，UI 从“Hello”刷新为“ArkUI”
      this.label = 'ArkUI';
    })
  }
 }
}
```

就算一直点击按钮，也不会看到对应的组件发生变化，说明按照值传递不会引起@Builder 方法内的 UI 刷新。

3.4 状态管理

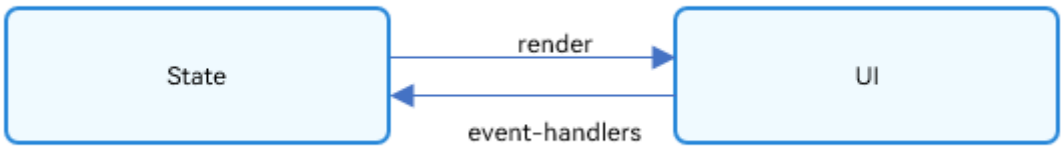
3.4.1 状态管理概述

在前文的描述中，我们构建的页面多为静态界面。如果希望构建一个动态的、有交互的界面，就需要引入“状态”的概念。

在本章节开始的案例中，用户与应用程序的交互触发了文本状态变更，状态变更引起了 UI 渲染，UI 从“Hello World”变更为“Hello ArkUI”，这个过程就用到了状态。

在声明式 UI 编程框架中，UI 是程序状态的运行结果，用户构建了一个 UI 模型，其中应用的运行时的状态是参数。当参数改变时，UI 作为返回结果，也将进行对应的改变。这些运行时的状态变化所带来的 UI 的重新渲染，在 ArkUI 中统称为状态管理机制。

自定义组件拥有变量，变量必须被装饰器装饰才可以成为状态变量，状态变量的改变会引起 UI 的渲染刷新。如果不使用状态变量，UI 只能在初始化时渲染，后续将不会再刷新。 下图展示了 State 和 View（UI）之间的关系。



- View(UI)：UI 渲染，指将 build 方法内的 UI 描述和@Builder 装饰的方法内的 UI 描述映射到界面。
- State：状态，指驱动 UI 更新的数据。用户通过触发组件的事件方法，改变状态数据。状态数据的改变，引起 UI 的重新渲染。

3.4.1.1 基本概念

- 状态变量：被状态装饰器装饰的变量，状态变量值的改变会引起 UI 的渲染更新。示例：@State num: number = 1,其中，@State 是状态装饰器，num 是状态变量。
- 常规变量：没有被状态装饰器装饰的变量，通常应用于辅助计算。它的改变永远不会引起 UI 的刷新。以下示例中 increaseBy 变量为常规变量。
- 数据源/同步源：状态变量的原始来源，可以同步给不同的状态数据。通常意义为父组件传给子组件的数据。以下示例中数据源为 count: 1。
- 命名参数机制：父组件通过指定参数传递给子组件的状态变量，为父子传递同步参数的主要手段。示例：CompA: ({ aProp: this.aProp })。
- 从父组件初始化：父组件使用命名参数机制，将指定参数传递给子组件。子组件初始化的默认值在有父组件传值的情况下，会被覆盖。示例：

```
@Component
struct MyComponent {
  @State count: number = 0;
  private increaseBy: number = 1;

  build() {
  }
}

@Component
struct Parent {
  build() {
    Column() {
      // 从父组件初始化，覆盖本地定义的默认值
      MyComponent({ count: 1, increaseBy: 2 })
    }
  }
}
```

- 初始化子节点：父组件中状态变量可以传递给子组件，初始化子组件对应的状态变量。示例同上。
- 本地初始化：在变量声明的时候赋值，作为变量的默认值。示例：@State count: number = 0。

3.4.2 管理组件拥有的状态

3.4.2.1 @State 装饰器-组件内状态

@State 装饰的变量，或称为状态变量，一旦变量拥有了状态属性，就和自定义组件的渲染绑定起来。当状态改变时，UI 会发生对应的渲染改变。

在状态变量相关装饰器中，@State 是最基础的，使变量拥有状态属性的装饰器，它也是大部分状态变量的数据源。从 API version 9 开始，该装饰器支持在 ArkTS 卡片中使用。

1. 概述

@State 装饰的变量，与声明式范式中的其他被装饰变量一样，是私有的，只能从组件内部访问，在声明时必须指定其类型和本地初始化。初始化也可选择使用命名参数机制从父组件完成初始化。

@State 装饰的变量拥有以下特点：

- @State 装饰的变量与子组件中的@Prop、@Link 或@ObjectLink 装饰变量之间建立单向或双向数据同步。
- @State 装饰的变量生命周期与其所属自定义组件的生命周期相同。

2. 装饰器使用规则说明

@State 变量装饰器	说明
装饰器参数	无
同步类型	不与父组件中任何类型的变量同步。

@State 变量装饰器	说明
允许装饰的变量类型	Object、class、string、number、boolean、enum 类型，以及这些类型的数组。 类型必须被指定。 不支持 any，不支持简单类型和复杂类型的联合类型，不允许使用 undefined 和 null。 说明：建议不要装饰 Date 类型，应用可能会产生异常行为。不支持 Length、ResourceStr、ResourceColor 类型，Length、ResourceStr、ResourceColor 为简单类型和复杂类型的联合类型。
被装饰变量的初始值	必须本地初始化。

3. 使用场景

A. 装饰简单类型的变量

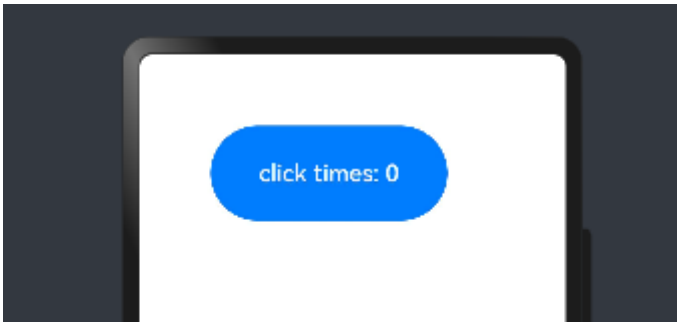
以下示例为@State 装饰的简单类型，count 被@State 装饰成为状态变量，count 的改变引起 Button 组件的刷新：

- 当状态变量 count 改变时，查询到只有 Button 组件关联了它；
- 执行 Button 组件的更新方法，实现按需刷新。

```
    @Entry
@Component
struct MyComponent {
  @State count: number = 0;

  build() {
    Button(`click times: ${this.count}`)
      .width(200)
      .height(80)
      .fontSize(20)
      .margin(60)
      .onClick(() => {
        this.count += 1;
      })
```

```
  }
}
```

以上代码预览如下，每当点击按钮，都会自动加 1。



B. 装饰 Class 对象类型的变量

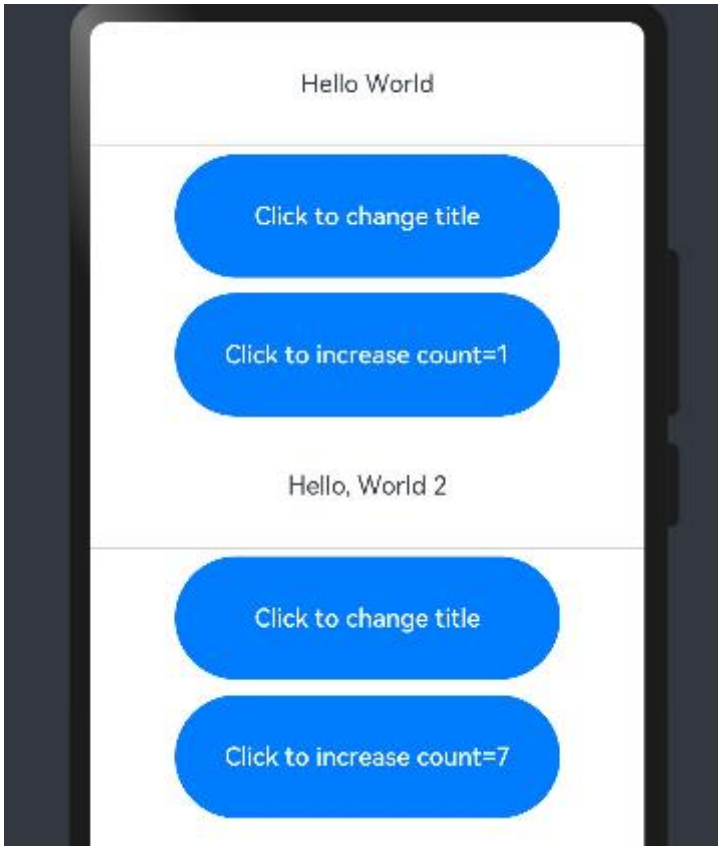
- 自定义组件 MyComponent 定义了被@State 装饰的状态变量 count 和 title，其中 title 的类型为自定义类 Model。如果 count 或 title 的值发生变化，则查询 MyComponent 中使用该状态变量的 UI 组件，并进行重新渲染。

- EntryComponent 中有多个 MyComponent 组件实例，第一个 MyComponent 内部状态的更改不会影响第二个 MyComponent。

```
    class Model {
  public value: string;

  constructor(value: string) {
    this.value = value;
  }
}

@Entry
@Component
struct EntryComponent {
  build() {
    Column() {
      // 此处指定的参数都将在初始渲染时覆盖本地定义的默认值，并不是所有的参数都需要从父组件初始化
      MyComponent({ count: 1, increaseBy: 2 })
      MyComponent({ title: new Model('Hello, World 2'), count: 7 })
    }
  }
}
```

```
@Component
struct MyComponent {
  @State title: Model = new Model('Hello World');
  @State count: number = 0;
  private increaseBy: number = 1;

  build() {
    Column() {
      Text(`${this.title.value}`)
        .height(80)
      Divider()
      Button(`Click to change title`).onClick(() => {
        // @State 变量的更新将触发上面的 Text 组件内容更新
        this.title.value = this.title.value === 'Hello ArkUI' ? 'Hello World' : 'Hello ArkUI';
      })
        .height(80)
        .width(250)
        .margin(5)

      Button(`Click to increase count=${this.count}`).onClick(() => {
        // @State 变量的更新将触发该 Button 组件的内容更新
        this.count += this.increaseBy;
      })
        .height(80)
        .width(250)
        .margin(5)
    }
  }
}
```

以上案例预览图如下,当点击两个“Click to change title”按钮时，对应的标题会在“Hello World”和“Hello ArkUI”之间进行切换。在第一个 MyComponent 中点击第二个按钮时会自动加 2；在第二个 MyComponent 中点击第二个按钮时会自动加 1。



从该示例中，我们可以了解到@State 变量首次渲染的初始化流程：

1) 使用默认的本地初始化：

```
@State title: Model = new Model('Hello World');
@State count: number = 0;
```

2) 对于@State 来说，命名参数机制传递的值并不是必选的，如果没有命名参数传值，则使用本地初始化的默认值：

```
MyComponent({ count: 1, increaseBy: 2 })
```

3.4.2.2 @Prop 装饰器-父子单向同步

@Prop 装饰的变量可以和父组件建立单向的同步关系。@Prop 装饰的变量是可变的，但是变化不会同步回其父组件。从 API version 9 开始，该装饰器支持在 ArkTS 卡片中使用。

1. 概述

@Prop 装饰的变量和父组件建立单向的同步关系：

- @Prop 变量允许在本地修改，但修改后的变化不会同步回父组件。

- 当父组件中的数据源更改时，与之相关的@Prop 装饰的变量都会自动更新。如果子组件已经在本地修改了@Prop 装饰的相关变量值，而在父组件中对应的@State 装饰的变量被修改后，子组件本地修改的@Prop 装饰的相关变量值将被覆盖。

2. 限制条件

- @Prop 修饰复杂类型时是深拷贝，在拷贝的过程中除了基本类型、Map、Set、Date、Array 外，都会丢失类型。

- @Prop 装饰器不能在@Entry 装饰的自定义组件中使用。

3. 装饰器使用规则说明

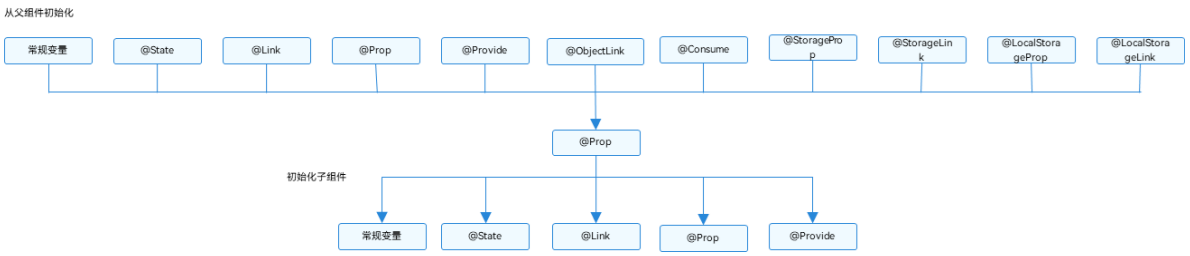
@Prop 变量装饰器	说明
装饰器参数	无
同步类型	单向同步：对父组件状态变量值的修改，将同步给子组件@Prop 装饰的变量，子组件@Prop 变量的修改不会同步到父组件的状态变量上
允许装饰的变量类型	string、number、boolean、enum 类型。 不支持 any，不允许使用 undefined 和 null。 必须指定类型。 在父组件中，传递给@Prop 装饰的值不能为 undefined 或者 null，反例如下所示。 CompA ({ aProp: undefined }) CompA ({ aProp: null }) @Prop 和数据源类型需要相同，有以下三种情况（数据源以@State 为例）： ● @Prop 装饰的变量和父组件状态变量类型相同，即@Prop : S 和@State : S； ● 当父组件的状态变量为数组时，@Prop 装饰的变量和父组件状态变量的数组项类型相同，即@Prop : S 和@State : Array<S>； ● 当父组件状态变量为 Object 或者 class 时，@Prop 装饰的变量和父组件状态变量的属性类型相同，即@Prop : S 和@State :

@Prop 变量装饰器	说明
	{ propA: S }。
被装饰变量的初始值	允许本地初始化。

4. 变量的传递/访问规则说明

传递/访问	说明
从父组件初始化	如果本地有初始化，则是可选的。没有的话，则必选，支持父组件中的常规变量、@State、@Link、@Prop、@Provide、@Consume、@ObjectLink、@StorageLink、@StorageProp、@LocalStorageLink 和@LocalStorageProp 去初始化子组件中的@Prop 变量。
用于初始化子组件	@Prop 支持去初始化子组件中的常规变量、@State、@Link、@Prop、@Provide。
是否支持组件外访问	@Prop 装饰的变量是私有的，只能在组件内访问。

初始化规则图示：



5. 观察变化和行为表现

A. 观察变化

@Prop 装饰的数据可以观察到以下变化。

- 当装饰的类型是允许的类型，即 string、number、boolean、enum 类型都可以观察到的赋值变化；

```
// 简单类型
@Prop count: number;
// 赋值的变化可以被观察到
this.count = 1;
```

对于@State 和@Prop 的同步场景：

- 使用父组件中@State 变量的值初始化子组件中的@Prop 变量。当@State 变量变化时，该变量值也会同步更新至@Prop 变量。
- @Prop 装饰的变量的修改不会影响其数据源@State 装饰变量的值。
- 除了@State，数据源也可以用@Link 或@Prop 装饰，对@Prop 的同步机制是相同的。
- 数据源和@Prop 变量的类型需要相同。

B. 框架行为

要理解@Prop 变量值初始化和更新机制，有必要了解父组件和拥有@Prop 变量的子组件初始渲染和更新流程。

1) 初始渲染：

a. 执行父组件的 build()函数将创建子组件的新实例，将数据源传递给子组件；

b. 初始化子组件@Prop 装饰的变量。

2) 更新：

a. 子组件@Prop 更新时，更新仅停留在当前子组件，不会同步回父组件；

b. 当父组件的数据源更新时，子组件的@Prop 装饰的变量将被来自父组件的数据源重置，所有@Prop 装饰的本地的修改将被父组件的更新覆盖。

6. 使用场景

A. 父组件@State 到子组件@Prop 简单数据类型同步

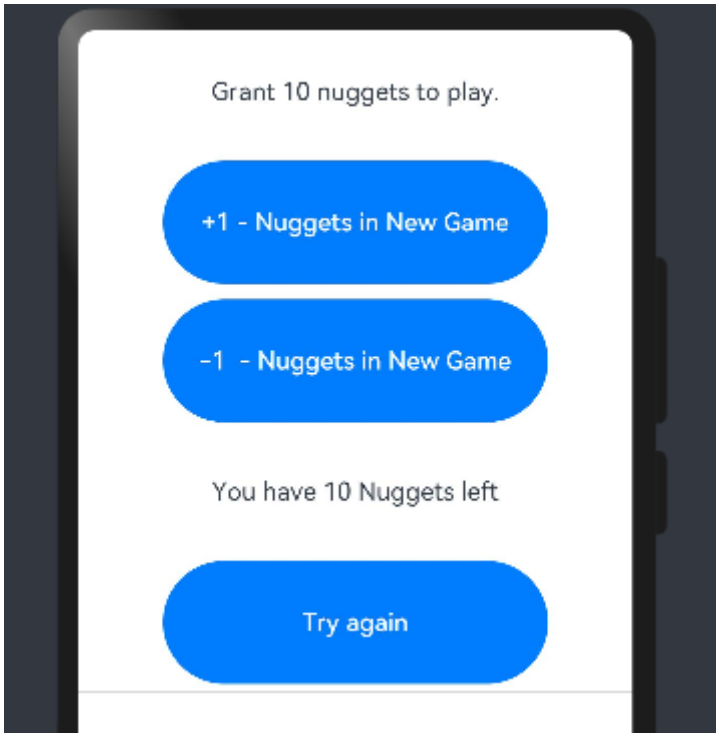
以下示例是@State 到子组件@Prop 简单数据同步，父组件 ParentComponent 的状态变量 countDownStartValue 初始化子组件 CountDownComponent 中@Prop 装饰的 count，点击“Try again”，count 的修改仅保留在 CountDownComponent，不会同步给父组件 ParentComponent。

ParentComponent 的状态变量 countDownStartValue 的变化将重置 CountDownComponent 的 count。

```
    @Component
struct CountDownComponent {
  @Prop count: number;
  costOfOneAttempt: number = 1;

  build() {
    Column() {
      if (this.count > 0) {
        Text(`You have ${this.count} Nuggets left`).height(80)
      } else {
        Text('Game over!').height(80)
      }
      // @Prop 装饰的变量不会同步给父组件
      Button(`Try again`).onClick(() => {
        this.count -= this.costOfOneAttempt;
      }).height(80)
        .width(250)
        .margin(5)
    }
  }
}

@Entry
@Component
struct ParentComponent {
  @State countDownStartValue: number = 10;

  build() {
    Column() {
      Text(`Grant ${this.countDownStartValue} nuggets to play.`)
        .height(80)
      // 父组件的数据源的修改会同步给子组件
      Button(`+1 - Nuggets in New Game`).onClick(() => {
        this.countDownStartValue += 1;
      }).height(80)
        .width(250)
        .margin(5)

      // 父组件的修改会同步给子组件
      Button(`-1  - Nuggets in New Game`).onClick(() => {
```

```
      this.countDownStartValue -= 1;
    }).height(80)
      .width(250)
      .margin(5)

    CountDownComponent({ count: this.countDownStartValue, costOfOneAttempt: 2 })
    Divider()
  }
 }
}
```

在上面的示例中，预览图如下：



1) CountDownComponent 子组件首次创建时其@Prop 装饰的 count 变量将从父组件@State 装饰的 countDownStartValue 变量初始化；

2) 按“+1”或“-1”按钮时，父组件的@State 装饰的 countDownStartValue 值会变化，这将触发父组件重新渲染，在父组件重新渲染过程中会刷新使用 countDownStartValue 状态变量的 UI 组件并单向同步更新 CountDownComponent 子组件中的 count 值；

3) 更新 count 状态变量值也会触发 CountDownComponent 的重新渲染，在重新渲染过程中，评估使用 count 状态变量的 if 语句条件（this.count > 0），并执行 true 分支中的使用 count 状态变量的 UI 组件相关描述来更新 Text 组件的 UI 显示；

4) 当按下子组件 CountDownComponent 的“Try again”按钮时，其@Prop 变量 count 将被更改，但是 count 值的更改不会影响父组件的 countDownStartValue 值；

5) 父组件的 countDownStartValue 值会变化时，父组件的修改将覆盖掉子组件 CountDownComponent 中 count 本地的修改。

B. 父组件@State 数组项到子组件@Prop 简单数据类型同步

父组件中@State 如果装饰的数组，其数组项也可以初始化@Prop。以下示例中父组件 Index 中@State 装饰的数组 arr，将其数组项初始化子组件 Child 中@Prop 装饰的 value。

```
    @Component
struct Child {
  @Prop value: number;

  build() {
    Text(`${this.value}`)
      .fontSize(50)
      .onClick(()=>{this.value++})
  }
}

@Entry
@Component
struct Index {
  @State arr: number[] = [1,2,3];

  build() {
    Row() {
      Column() {
        Child({value: this.arr[0]})
        Child({value: this.arr[1]})
        Child({value: this.arr[2]})

        Divider().height(5)

        ForEach(this.arr,
          item => {
            Child({value: item})
          },
          item => item.toString()
        )
        Text('replace entire arr')
```

```
        .fontSize(50)
        .onClick(()=>{
          // 两个数组都包含项“3”。
          this.arr = this.arr[0] == 1 ? [3,4,5] : [1,2,3];
        })
    }
  }
 }
}
```

以上代码运行预览如下：

当点击“replace entire arr”文本时，会判断 数组第一位是否是 1，同时给父组件中的 arr 赋值，并重新触发 ForEach 给子组件中的 value 赋值。父组件中的值传递给了子组件的@prop value。点击“replace entire arr”文本，预览如下：

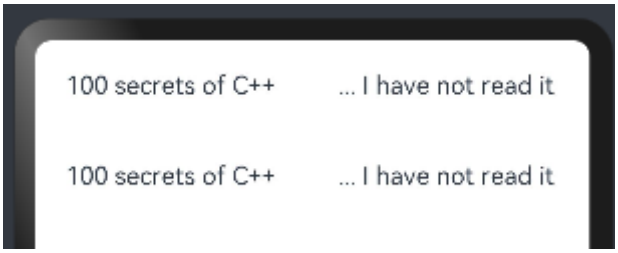
C. 从父组件中的@State 类对象属性到@Prop 简单类型的同步

如果图书馆有一本图书和两位用户，每位用户都可以将图书标记为已读，此标记行为不会影响其它读者用户。从代码角度讲，对@Prop 图书对象的本地更改不会同步给图书馆组件中的@State 图书对象。

```
    class Book {
  public title: string;
  public pages: number;
  public readIt: boolean = false;

  constructor(title: string, pages: number) {
    this.title = title;
    this.pages = pages;
  }
}

@Component
struct ReaderComp {
  @Prop title: string;
  @Prop readIt: boolean;

  build() {
    Row() {
      Text(this.title).margin(20)
      Text(`... ${this.readIt ? 'I have read' : 'I have not read it'}`)
        .margin(20)
        .onClick(() => this.readIt = true)
    }
```

```
  }
}

@Entry
@Component
struct Library {
  @State book: Book = new Book('100 secrets of C++', 765);

  build() {
    Column() {
      ReaderComp({ title: this.book.title, readIt: this.book.readIt })
      ReaderComp({ title: this.book.title, readIt: this.book.readIt })
    }
  }
}
```

以上案例预览如下：



当点击“I have not read it”后，自动更改为“I have read”，完成了父组件和子组件简单类型同步并且每个子组件互不影响。

D. @Prop 本地初始化不和父组件同步

为了支持@Component 装饰的组件复用场景，@Prop 支持本地初始化，这样可以让@Prop 是否与父组件建立同步关系变得可选。当且仅当@Prop 有本地初始化时，从父组件向子组件传递@Prop 的数据源才是可选的。

下面的示例中，子组件包含两个@Prop 变量：

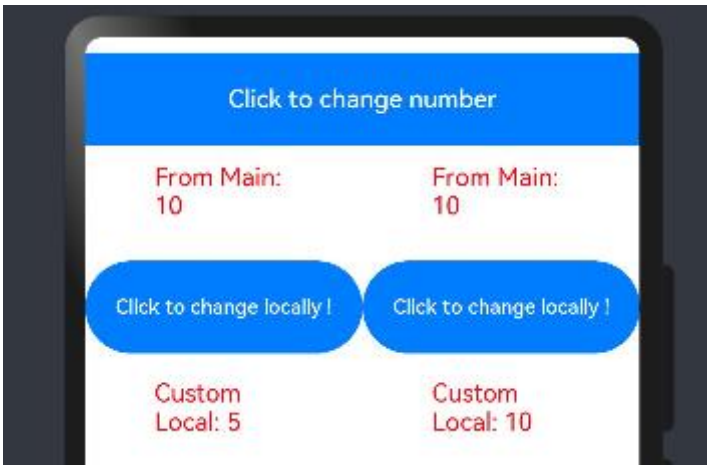
- @Prop customCounter 没有本地初始化，所以需要父组件提供数据源去初始化@Prop，并当父组件的数据源变化时，@Prop 也将被更新；
- @Prop customCounter2 有本地初始化，在这种情况下，@Prop 依旧允许但非强制父组件同步数据源给@Prop。

```
    @Component
struct MyComponent {
  @Prop customCounter: number;
```

```
  @Prop customCounter2: number = 5;

  build() {
    Column() {
      Row() {
        Text(`From Main: ${this.customCounter}`).width(90).height(40).fontColor('#FF0010')
      }

      Row() {
        Button('Click to change locally !').width(180).height(60).margin({ top: 10 })
          .onClick(() => {
            this.customCounter2++
          })
      }.height(100).width(180)

      Row() {
        Text(`Custom Local: ${this.customCounter2}`).width(90).height(40).fontColor('#FF0010')
      }
    }
  }
}

@Entry
@Component
struct MainProgram {
  @State mainCounter: number = 10;

  build() {
    Column() {
      Row() {
        Column() {
          Button('Click to change number').width(480).height(60).margin({ top: 10, bottom: 10 })
            .onClick(() => {
              this.mainCounter++
            })
        }
      }

      Row() {
        // customCounter 必须从父组件初始化，因为 MyComponent 的 customCounter 成员变量缺少本地初始化；此处，customCounter2 可以不做初始化。
        MyComponent({ customCounter: this.mainCounter })
        // customCounter2 也可以从父组件初始化，父组件初始化的值会覆盖子组件 customCounter2 的本地初始化的值
        MyComponent({ customCounter: this.mainCounter, customCounter2: this.mainCounter })

      }
    }
```

```
  }
}
```

以上代码预览如下：



当点击第 2 行两个“Click to change locally !”时，“Custorm Local：x”都会自动加 1。当点击“Click to change number”时，第 2 行中两个“From Main”和第 2 个“Click to change number”自动加 1，第一个“Click to change number”保持不变，不和父组件同步。

3.4.2.3 @Link 装饰器-父子双向同步

子组件中被@Link 装饰的变量与其父组件中对应的数据源建立双向数据绑定。从 API version 9 开始，该装饰器支持在 ArkTS 卡片中使用。

需要注意：@Link 装饰的变量与其父组件中的数据源共享相同的值。@Link 装饰器不能在@Entry 装饰的自定义组件中使用。

1. 装饰器使用规则说明

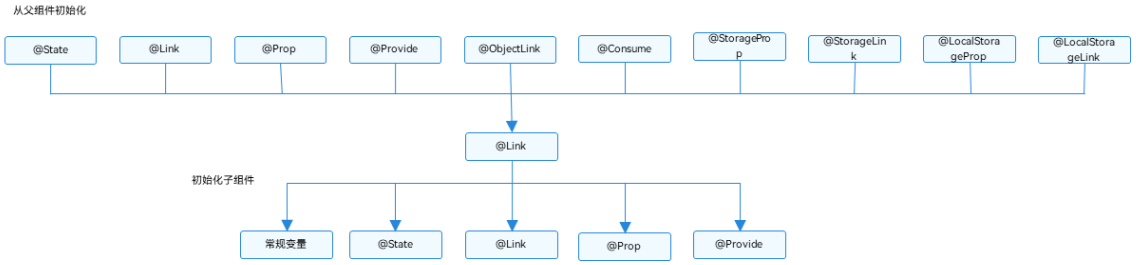
@Link 变量装饰器	说明
装饰器参数	无
同步类型	双向同步。 父组件中@State, @StorageLink 和@Link 和子组件@Link 可以建立双向数据同步，反之亦然。
允许装饰的变量类型	Object、class、string、number、boolean、enum 类型，以及这些类型的数组。

@Link 变量装饰器	说明
	类型必须被指定，且和双向绑定状态变量的类型相同。 不支持 any，不支持简单类型和复杂类型的联合类型，不允许使用 undefined 和 null。 说明：不支持 Length、ResourceStr、ResourceColor 类型，Length、ResourceStr、ResourceColor 为简单类型和复杂类型的联合类型。
被装饰变量的初始值	无，禁止本地初始化。

2. 变量的传递/访问规则说明

传递/访问	说明
从父组件初始化和更新	必选。与父组件@State, @StorageLink 和@Link 建立双向绑定。允许父组件中@State、@Link、@Prop、@Provide、@Consume、@ObjectLink、@StorageLink、@StorageProp、@LocalStorageLink 和@LocalStorageProp 装饰变量初始化子组件@Link。 从 API version 9 开始，@Link 子组件从父组件初始化@State 的语法为 Comp({ aLink: this.aState })。同样 Comp({aLink: $aState})也支持。
用于初始化子组件	允许，可用于初始化常规变量、@State、@Link、@Prop、@Provide。
是否支持组件外访问	私有，只能在所属组件内访问。

初始化规则图示如下：



3. 观察变化和行为表现

A. 观察变化

- 当装饰的数据类型为 boolean、string、number 类型时，可以同步观察到数值的变化。
- 当装饰的数据类型为 class 或者 Object 时，可以观察到赋值和属性赋值的变化，即 Object.keys(observedObject)返回的所有属性。
- 当装饰的对象是 array 时，可以观察到数组添加、删除、更新数组单元的变化。

B. 框架行为

@Link 装饰的变量和其所属的自定义组件共享生命周期。为了了解@Link 变量初始化和更新机制，有必要先了解父组件和拥有@Link 变量的子组件的关系，初始渲染和双向更新的流程（以父组件为@State 为例）。

1) 初始渲染：执行父组件的 build()函数后将创建子组件的新实例。初始化过程如下：

a. 必须指定父组件中的@State 变量，用于初始化子组件的@Link 变量。子组件的@Link 变量值与其父组件的数据源变量保持同步（双向数据同步）。

b. 父组件的@State 状态变量包装类通过构造函数传给子组件，子组件的@Link 包装类拿到父组件的@State 的状态变量后，将当前@Link 包装类 this 指针注册给父组件的@State 变量。

2) @Link 的数据源的更新：即父组件中状态变量更新，引起相关子组件的@Link 的更新。处理步骤：

a. 通过初始渲染的步骤可知，子组件@Link 包装类把当前 this 指针注册给父组件。父组件@State 变量变更后，会遍历更新所有依赖它的系统组件（elementid）和状态变量（比如@Link 包装类）。

b. 通知@Link 包装类更新后，子组件中所有依赖@Link 状态变量的系统组件（elementId）都会被通知更新。以此实现父组件对子组件的状态数据同步。

3) @Link 的更新：当子组件中@Link 更新后，处理步骤如下（以父组件为@State 为例）：

a. @Link 更新后，调用父组件的@State 包装类的 set 方法，将更新后的数值同步回父组件。

b. 子组件@Link 和父组件@State 分别遍历依赖的系统组件，进行对应的 UI 的更新。以此实现子组件@Link 同步回父组件@State。

4. 使用场景

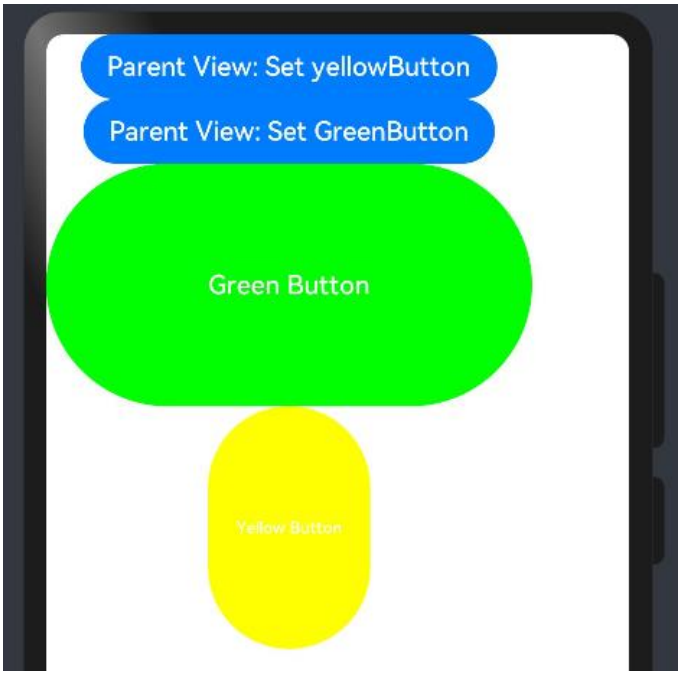
A. 简单类型和类对象类型的@Link

以下示例中，点击父组件 ShufflingContainer 中的“Parent View: Set yellowButton”和“Parent View: Set GreenButton”，可以从父组件将变化同步给子组件，子组件 GreenButton 和 YellowButton 中@Link 装饰变量的变化也会同步给其父组件。

```
    class GreenButtonState {
  width: number = 0;
  //构造方法
  constructor(width: number) {
    this.width = width;
  }
}
@Component
struct GreenButton {
  @Link greenButtonState: GreenButtonState;
  build() {
    Button('Green Button')
      .width(this.greenButtonState.width)
      .height(150.0)
      .backgroundColor('#00ff00')
      .onClick(() => {
        if (this.greenButtonState.width < 700) {
          // 更新 class 的属性，变化可以被观察到同步回父组件
          this.greenButtonState.width += 125;
        } else {
          // 更新 class，变化可以被观察到同步回父组件
          this.greenButtonState = new GreenButtonState(100);
        }
      })
  }
}
@Component
struct YellowButton {
  @Link yellowButtonState: number;
  build() {
    Button('Yellow Button')
      .width(this.yellowButtonState)
```

```
      .height(150.0)
      .backgroundColor('#ffff00')
      .onClick(() => {
        // 子组件的简单类型可以同步回父组件
        this.yellowButtonState += 50.0;
      })
  }
}
@Entry
@Component
struct ShufflingContainer {
  @State greenButtonState: GreenButtonState = new GreenButtonState(300);
  @State yellowButtonProp: number = 100;
  build() {
    Column() {
      // 简单类型从父组件@State 向子组件@Link 数据同步
      Button('Parent View: Set yellowButton')
        .onClick(() => {
          this.yellowButtonProp = (this.yellowButtonProp < 700) ? this.yellowButtonProp + 100 :
100;
        })
      // class 类型从父组件@State 向子组件@Link 数据同步
      Button('Parent View: Set GreenButton')
        .onClick(() => {
          this.greenButtonState.width = (this.greenButtonState.width < 700) ?
this.greenButtonState.width + 100 : 100;
        })
      // class 类型初始化@Link
      GreenButton({ greenButtonState: $greenButtonState })
      // 简单类型初始化@Link
      YellowButton({ yellowButtonState: $yellowButtonProp })
    }
  }
}
```

以上代码运行预览如下，点击四个按钮可以看到相应变化，父子组件进行双向同步数据。

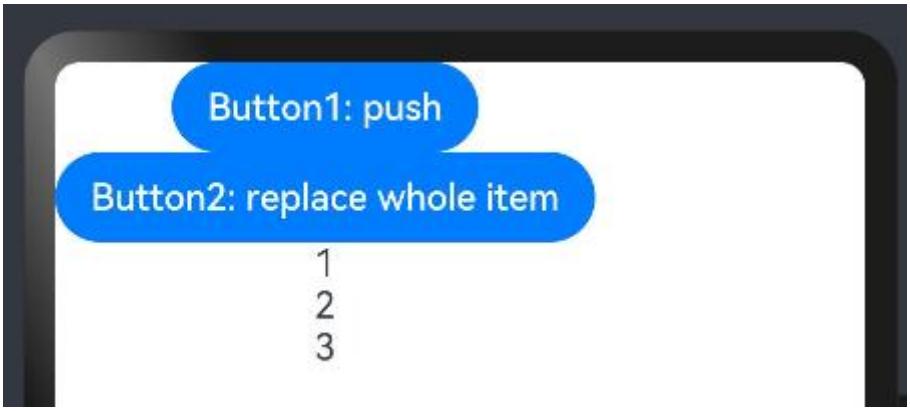


B. 数组类型的@Link

```
    @Component
struct Child {
  @Link items: number[];

  build() {
    Column() {
      Button(`Button1: push`).onClick(() => {
        this.items.push(this.items.length + 1);
      })
      Button(`Button2: replace whole item`).onClick(() => {
        this.items = [100, 200, 300];
      })
    }
  }
}

@Entry
@Component
struct Parent {
  @State arr: number[] = [1, 2, 3];

  build() {
    Column() {
      Child({ items: $arr })
      ForEach(this.arr,
        item => {
          Text(`${item}`)
        },
        item => item.toString()
```

```
    )
  }
 }
}
```

以上代码运行后预览如下，ArkUI 框架可以观察到数组元素的添加，删除和替换。在该示例中@State 和@Link 的类型是相同的 number[]，不允许将@Link 定义成 number 类型（@Link item : number），并在父组件中用@State 数组中每个数据项创建子组件。



每次点击“push”按钮数组长度都会增加，并且点击第二个按钮时会初始化数组。

3.5 if/else 条件渲染

ArkTS 提供了渲染控制的能力。条件渲染可根据应用的不同状态，使用 if、else 和 else if 渲染对应状态下的 UI 内容。从 API version 9 开始，该接口支持在 ArkTS 卡片中使用。

3.5.1 使用规则

- 支持 if、else 和 else if 语句。
- if、else if 后跟随的条件语句可以使用状态变量。
- 允许在容器组件内使用，通过条件渲染语句构建不同的子组件。
- 条件渲染语句在涉及到组件的父子关系时是“透明”的，当父组件和子组件之间存在一个或多个 if 语句时，必须遵守父组件关于子组件使用的规则。
- 每个分支内部的构建函数必须遵循构建函数的规则，并创建一个或多个组件。无法创建组件的空构建函数会产生语法错误。

- 某些容器组件限制子组件的类型或数量，将条件渲染语句用于这些组件内时，这些限制将同样应用于条件渲染语句内创建的组件。例如，Grid 容器组件的子组件仅支持 GridItem 组件，在 Grid 内使用条件渲染语句时，条件渲染语句内仅允许使用 GridItem 组件。

3.5.2 更新机制

当 if、else if 后跟随的状态判断中使用的状态变量值变化时，条件渲染语句会进行更新，更新步骤如下：

1) 评估 if 和 else if 的状态判断条件，如果分支没有变化，请无需执行以下步骤。如果分支有变化，则执行 2、3 步骤：

2) 删除此前构建的所有子组件。

3) 执行新分支的构造函数，将获取到的组件添加到 if 父容器中。如果缺少适用的 else 分支，则不构建任何内容。

条件可以包括 Typescript 表达式。对于构造函数中的表达式，此类表达式不得更改应用程序状态。

3.5.3 使用场景

3.5.3.1 使用 if 进行条件渲染

if 语句的每个分支都包含一个构建函数。此类构建函数必须创建一个或多个子组件。在初始渲染时，if 语句会执行构建函数，并将生成的子组件添加到其父组件中。

每当 if 或 else if 条件语句中使用的状态变量发生变化时，条件语句都会更新并重新评估新的条件值。如果条件值评估发生了变化，这意味着需要构建另一个条件分支。此时 ArkUI 框架将：

1) 删除所有以前渲染的（早期分支的）组件。

2) 执行新分支的构造函数，将生成的子组件添加到其父组件中。

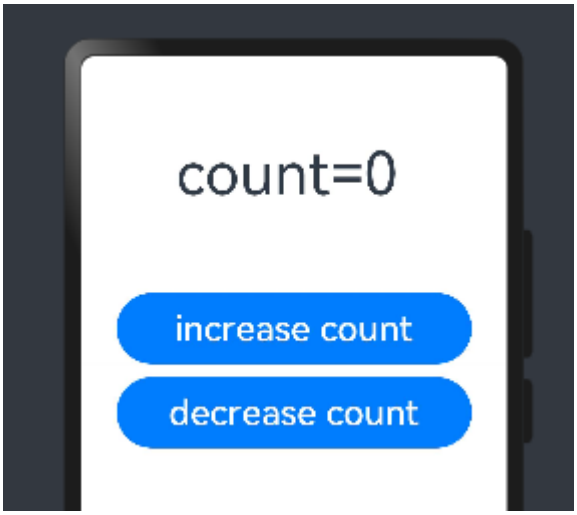
如下示例：

```
    @Entry
@Component
struct ViewA {
  @State count: number = 0;

  build() {
    Column() {
```

```
      Text(`count=${this.count}`)
      .width(200)
      .height(200)
      .fontSize(50)

      if (this.count > 0) {
        Text(`count is positive`)
          .width(300)
          .height(100)
          .fontSize(30)
          .fontColor(Color.Green)
      }

      Button('increase count')
        .width(300)
        .height(60)
        .fontSize(30)
        .onClick(() => {
          this.count++;
        })

      Divider()

      Button('decrease count')
        .width(300)
        .height(60)
        .fontSize(30)
        .margin(10)
        .onClick(() => {
          this.count--;
        })
    }
  }
}
```

在以上示例中，如果 count 从 0 增加到 1，那么 if 语句更新，条件 count > 0 将重新评估，评估结果将从 false 更改为 true。因此，将执行条件为真分支的构造函数，创建一个 Text 组件，并将它添加到父组件 Column 中。如果后续 count 更改为 0，则 Text 组件将从 Column 组件中删除。由于没有 else 分支，因此不会执行新的构造函数。

以上示例预览如下：



3.5.3.2 if...else...语句和子组件状态

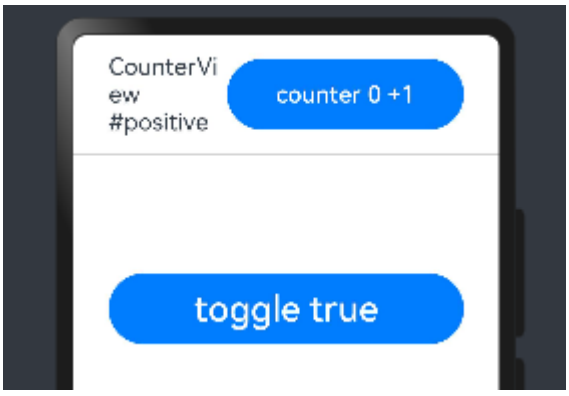
以下示例包含 if ... else ...语句与拥有@State 装饰变量的子组件。

```
    @Component
struct CounterView {
  @State counter: number = 0;
  label: string = 'unknown';

  build() {
    Row() {
      Text(`${this.label}`)
        .width(100)
        .height(100)
        .fontSize(20)

      Button(`counter ${this.counter} +1`)
        .width(200)
        .height(60)
        .fontSize(20)
        .onClick(() => {
          this.counter += 1;
        })
    }
  }
}

@Entry
@Component
struct MainView {
  @State toggle: boolean = true;

  build() {
    Column() {
      if (this.toggle) {
```

```
      CounterView({ label: 'CounterView #positive' })
    } else {
      CounterView({ label: 'CounterView #negative' })
    }
    Divider()
    Button(`toggle ${this.toggle}`)
      .width(300)
      .height(60)
      .fontSize(30)
      .margin(100)
      .onClick(() => {
        this.toggle = !this.toggle;
      })
  }
 }
}
```

CounterView（label 为 'CounterView #positive'）子组件在初次渲染时创建。此子组件携带名为 counter 的状态变量。当修改 CounterView.counter 状态变量时，CounterView（label 为 'CounterView #positive'）子组件重新渲染时并保留状态变量值。当 MainView.toggle 状态变量的值更改为 false 时，MainView 父组件内的 if 语句将更新，随后将删除 CounterView（label 为 'CounterView #positive'）子组件。与此同时，将创建新的 CounterView（label 为 'CounterView #negative'）实例。而它自己的 counter 状态变量设置为初始值 0。

以上代码预览如下：



CounterView（label 为 'CounterView #positive'）和 CounterView（label 为 'CounterView #negative'）是同一自定义组件的两个不同实例。if 分支的更改，不会更新现有子组件，也不会保留状态。

以下示例展示了条件更改时，若需要保留 counter 值所做的修改。

```
    @Component
struct CounterView {
  @Link counter: number;
  label: string = 'unknown';
```

```
  build() {
    Row() {
      Text(`${this.label}`)
        .width(100)
        .height(100)
        .fontSize(20)

      Button(`counter ${this.counter} +1`)
        .width(200)
        .height(60)
        .fontSize(20)
        .onClick(() => {
          this.counter += 1;
        })
    }
  }
}

@Entry
@Component
struct MainView {
  @State toggle: boolean = true;
  @State counter: number = 0;

  build() {
    Column() {
      if (this.toggle) {
        CounterView({ counter: $counter, label: 'CounterView #positive' })
      } else {
        CounterView({ counter: $counter, label: 'CounterView #negative' })
      }
      Button(`toggle ${this.toggle}`)
        .width(300)
        .height(60)
        .fontSize(30)
        .margin(100)
        .onClick(() => {
          this.toggle = !this.toggle;
        })
    }
  }
}
```

此处，@State counter 变量归父组件所有。因此，当 CounterView 组件实例被删除时，该变量不会被销毁。CounterView 组件通过@Link 装饰器引用状态。状态必须从子级移动到其父级（或父级的父级），以避免在条件内容或重复内容被销毁时丢失状态。

3.5.3.3 嵌套 if 语句

条件语句的嵌套对父组件的相关规则没有影响。

3.6 ForEach 循环渲染

ForEach 接口基于数组类型数据来进行循环渲染，需要与容器组件配合使用，且接口返回的组件应当是允许包含在 ForEach 父容器组件中的子组件。例如，ListItem 组件要求 ForEach 的父容器组件必须为 List 组件。从 API version 9 开始，该接口支持在 ArkTS 卡片中使用。

3.6.1 接口描述

```
ForEach(
arr: Array,
itemGenerator: (item: Array, index?: number) => void,
keyGenerator?: (item: Array, index?: number): string => string
)
```

关于 ForEach 参数如下：

参数名	参数类型	必填	参数描述
arr	Array	是	数据源，为 Array 类型的数组。 **说明**： - 可以设置为空数组，此时不会创建子组件。 - 可以设置返回值为数组类型的函数，例如 arr.slice(1, 3)，但设置的函数不应改变包括数组本身在内的任何状态变量，例如不应使用 Array.splice(),Array.sort()或 Array.reverse()这些会改变原数组的函数。
itemGenerator	(item: any, index?: number) => void	是	组件生成函数。 - 为数组中的每个元素创建对应的组件。 - item 参数：arr 数组中的数据项。 - index 参数（可选）：arr 数组中的数据项索引。 **说明**：

参数名	参数类型	必填	参数描述
			- 组件的类型必须是 ForEach 的父容器所允许的。例如，ListItem 组件要求 ForEach 的父容器组件必须为 List 组件。
keyGenerator	(item: any, index?: number) => string	否	键值生成函数。 - 为数据源 arr 的每个数组项生成唯一且持久的键值。函数返回值为开发者自定义的键值生成规则。 - item 参数：arr 数组中的数据项。- index 参数（可选）：arr 数组中的数据项索引。 **说明**: - 如果函数缺省，框架默认的键值生成函数为(item: T, index: number) => { return index + '__' + JSON.stringify(item); } - 键值生成函数不应改变任何组件状态。

- ForEach 的 itemGenerator 函数可以包含 if/else 条件渲染逻辑。另外，也可以在 if/else 条件渲染语句中使用 ForEach 组件。

- 在初始化渲染时，ForEach 会加载数据源的所有数据，并为每个数据项创建对应的组件，然后将其挂载到渲染树上。如果数据源非常大或有特定的性能需求，建议使用 LazyForEach 组件。

3.6.2 键值生成规则

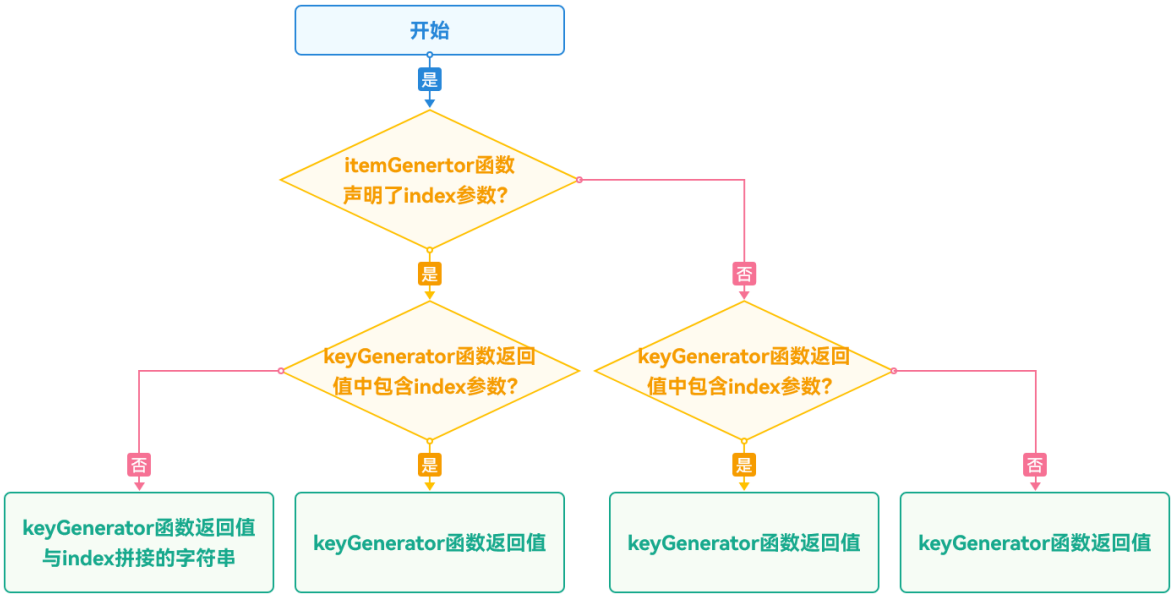
在 ForEach 循环渲染过程中，系统会为每个数组元素生成一个唯一且持久的键值，用于标识对应的组件。当这个键值变化时，ArkUI 框架将视为该数组元素已被替换或修改，并会基于新的键值创建一个新的组件。

ForEach 提供了一个名为 keyGenerator 的参数，这是一个函数，开发者可以通过它自定义键值的生成规则。如果开发者没有定义 keyGenerator 函数，则 ArkUI 框架会使用默认的键值生成函数，即(item: any, index: number) => { return index + '__' + JSON.stringify(item); }。

ArkUI 框架对于 ForEach 的键值生成有一套特定的判断规则，这主要与 itemGenerator 函数的第二个参数 index 以及 keyGenerator 函数的返回值有关。总的来说，只有当开发者在 itemGenerator 函数中声明了 index

参数，并且自定义的 keyGenerator 函数返回值中不包含 index 参数时，ArkUI 框架才会在开发者自定义的 keyGenerator 函数返回值前添加 index 参数，作为最终的键值。在其他情况下，系统将直接使用开发者自定义的 keyGenerator 函数返回值作为最终的键值。如果 keyGenerator 函数未定义，系统将使用上述默认的键值生成函数。具体的键值生成规则判断逻辑如下图所示。

下图是 ForEach 键值生成规则：



注意：ArkUI 框架会对重复的键值发出警告。在 UI 更新的场景下，如果出现重复的键值，框架可能无法正常工作

3.6.3 组件创建规则

在确定键值生成规则后，ForEach 的第二个参数 itemGenerator 函数会根据键值生成规则为数据源的每个数组项创建组件。组件的创建包括两种情况：ForEach 首次渲染和 ForEach 非首次渲染。

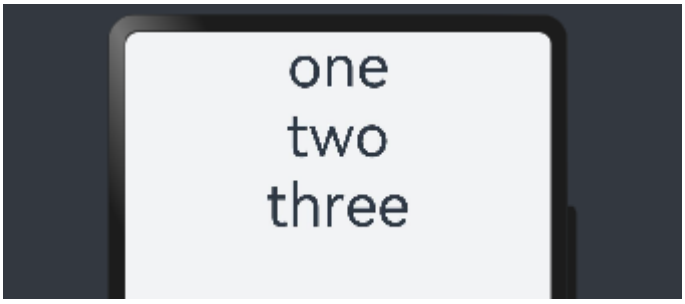
3.6.3.1 首次渲染

在 ForEach 首次渲染时，会根据前述键值生成规则为数据源的每个数组项生成唯一键值，并创建相应的组件。

```
    @Entry
@Component
struct Parent {
  @State simpleList: Array<string> = ['one', 'two', 'three'];

  build() {
    Row() {
```

```
      Column() {
        ForEach(this.simpleList, (item: string) => {
          ChildItem({ item: item })
        }, (item: string) => item)
      }
      .width('100%')
      .height('100%')
    }
    .height('100%')
    .backgroundColor(0xF1F3F5)
  }
}

@Component
struct ChildItem {
  item: string;

  build() {
    Text(this.item)
      .fontSize(50)
  }
}
```

以上代码运行效果如下图:



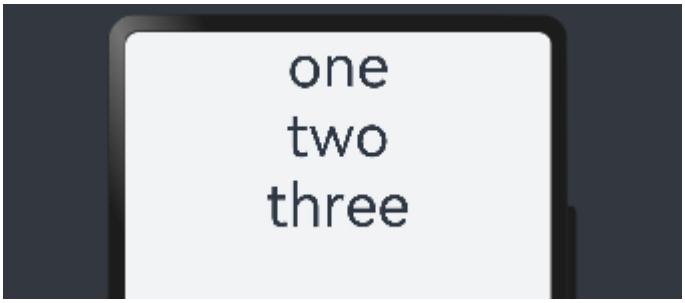
在上述代码中，键值生成规则是 keyGenerator 函数的返回值 item。在 ForEach 渲染循环时，为数据源数组项依次生成键值 one、two 和 three，并创建对应的 ChildItem 组件渲染到界面上。

当不同数组项按照键值生成规则生成的键值相同时，框架的行为是未定义的。例如，在以下代码中，ForEach 渲染相同的数据项 two 时，只创建了一个 ChildItem 组件，而没有创建多个具有相同键值的组件。

```
      @Entry
@Component
struct Parent {
  @State simpleList: Array<string> = ['one', 'two', 'three','two'];

  build() {
    Row() {
```

```
      Column() {
        ForEach(this.simpleList, (item: string) => {
          ChildItem({ item: item })
        }, (item: string) => item)
      }
      .width('100%')
      .height('100%')
    }
    .height('100%')
    .backgroundColor(0xF1F3F5)
  }
}

@Component
struct ChildItem {
  item: string;

  build() {
    Text(this.item)
      .fontSize(50)
  }
}
```

运行效果如下图所示，ForEach 数据源存在相同值案例首次渲染运行效果图。



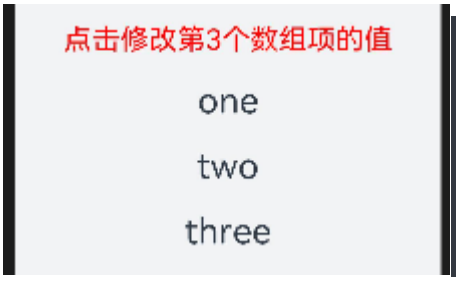
在该示例中，最终键值生成规则为 item。当 ForEach 遍历数据源 simpleList，遍历到索引为 1 的 two 时，按照最终键值生成规则生成键值为 two 的组件并进行标记。当遍历到索引为 3 的 two 时，按照最终键值生成规则当前项的键值也为 two，此时不再创建新的组件。

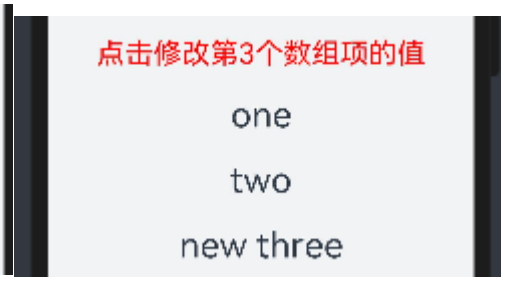
3.6.3.2 非首次渲染

在 ForEach 组件进行非首次渲染时，它会检查新生成的键值是否在上次渲染中已经存在。如果键值不存在，则会创建一个新的组件；如果键值存在，则不会创建新的组件，而是直接渲染该键值所对应的组件。例如，在以下的代码示例中，通过点击事件修改了数组的第三项值为"new three"，这将触发 ForEach 组件进行非首次渲染。

```
@Entry
```

```
@Component
struct Parent {
  @State simpleList: Array<string> = ['one', 'two', 'three'];

  build() {
    Row() {
      Column() {
        Text('点击修改第 3 个数组项的值')
          .fontSize(24)
          .fontColor(Color.Red)
          .onClick(() => {
            this.simpleList[2] = 'new three';
          })

        ForEach(this.simpleList, (item: string) => {
          ChildItem({ item: item })
            .margin({ top: 20 })
        }, (item: string) => item)
      }
      //设置子组件在垂直方向上的对齐格式，默认值：FlexAlign.Start
      .justifyContent(FlexAlign.Center)
      .width('100%')
      .height('100%')
    }
    .height('100%')
    .backgroundColor(0xF1F3F5)
  }
}

@Component
struct ChildItem {
  item: string;

  build() {
    Text(this.item)
      .fontSize(30)
  }
}
```





从本例可以看出@State 能够监听到简单数据类型数组数据源 simpleList 数组项的变化。

1) 当 simpleList 数组项发生变化时，会触发 ForEach 进行重新渲染。

2) ForEach 遍历新的数据源 ['one', 'two', 'new three']，并生成对应的键值 one、two 和 new three。

3) 其中，键值 one 和 two 在上次渲染中已经存在，所以 ForEach 复用了对应的组件并进行了渲染。对于第三个数组项 "new three"，由于其通过键值生成规则 item 生成的键值 new three 在上次渲染中不存在，因此 ForEach 为该数组项创建了一个新的组件。

3.6.4 使用场景

ForEach 组件在开发过程中的主要应用场景包括：数据源不变、数据源数组项发生变化（如插入、删除操作）、数据源数组项子属性变化。

3.6.4.1 数据源数组项发生变化

在数据源数组项发生变化的场景下，例如进行数组插入、删除操作或者数组项索引位置发生交换时，数据源应为对象数组类型，并使用对象的唯一 ID 作为最终键值。例如，当在页面上通过手势上滑加载下一页数据时，会在数据源数组尾部新增新获取的数据项，从而使得数据源数组长度增大。

```
    @Entry
@Component
struct ArticleListView {
  @State isListReachEnd: boolean = false;
  @State articleList: Array<Article> = [
    //创建对象，详见后面 Article 类
    new Article('001', '第 1 篇文章', '文章简介内容'),
    new Article('002', '第 2 篇文章', '文章简介内容'),
    new Article('003', '第 3 篇文章', '文章简介内容'),
    new Article('004', '第 4 篇文章', '文章简介内容'),
    new Article('005', '第 5 篇文章', '文章简介内容'),
    new Article('006', '第 6 篇文章', '文章简介内容')
  ]

  //自定义组件中的方法，用于添加一个文章
  loadMoreArticles() {
    this.articleList.push(new Article('007', '加载的新文章', '文章简介内容'));
  }

  build() {
    Column({ space: 5 }) {
```

```
      List() {
        ForEach(this.articleList, (item: Article) => {
          ListItem() {
            // ArticleCard 为自定义组件，用于构建单个文章卡片
            ArticleCard({ article: item })
              .margin({ top: 20 })
          }
        }, (item: Article) => item.id)
      }
      //列表滚动和触底加载更多功能
      .onReachEnd(() => {
        this.isListReachEnd = true;
      })
      // 定义了一个并行手势，这里用于检测向上的滑动手势
      .parallelGesture(
        PanGesture({ direction: PanDirection.Up, distance: 80 })
          .onActionStart(() => {
            //当手势开始且列表已到达底部时
            if (this.isListReachEnd) {
              //添加文章
              this.loadMoreArticles();
              this.isListReachEnd = false;
            }
          })
      )
      // 为列表添加 20 单位的内边距
      .padding(20)
      //表示关闭滚动条
      .scrollBar(BarState.Off)
    }
    .width('100%')
    .height('100%')
    .backgroundColor(0xF1F3F5)
  }
}

@Component
struct ArticleCard {
  article: Article;

  build() {
    Row() {
      Image($r('app.media.icon'))
        .width(80)
        .height(80)
        .margin({ right: 20 })

      Column() {
```

```
      Text(this.article.title)
        .fontSize(20)
        .margin({ bottom: 8 })
      Text(this.article.brief)
        .fontSize(16)
        .fontColor(Color.Gray)
        .margin({ bottom: 8 })
    }
    //设置子组件在水平方向上的对齐格式,默认值：HorizontalAlign.Center
    .alignItems(HorizontalAlign.Start)
    .width('80%')
    .height('100%')
  }
  //设置内边缘
  .padding(20)
  //是否圆角
  .borderRadius(12)
  .backgroundColor('#FFECECEC')
  .height(120)
  .width('100%')
  //设置子组件在垂直方向上的对齐格式。默认值：FlexAlign.Start
  //Flex 主轴方向均匀分配弹性元素，相邻元素之间距离相同。第一个元素与行首对齐，最后一个元素与行尾对齐。
  .justifyContent(FlexAlign.SpaceBetween)
 }
}

//自定义 Article 类
class Article {
 public id: string
 public title: string
 public brief: string

 constructor(id: string, title: string, brief: string) {
  this.id = id;
  this.title = title;
  this.brief = brief;
 }

}
```

初始运行效果（左图）和手势上滑加载后效果（右图）如下图所示。

在以上示例中，ArticleCard 组件作为 ArticleListView 组件的子组件，装饰器接收一个 Article 对象，用于渲染文章卡片。

1) 当列表滚动到底部时，如果手势滑动距离超过指定的 80，将触发 loadMoreArticle()函数。此函数会在 articleList 数据源的尾部添加一个新的数据项，从而增加数据源的长度。

2) 数据源被@State 装饰器修饰，ArkUI 框架能够感知到数据源长度的变化，并触发 ForEach 进行重新渲染。

3.6.5 使用建议

- 尽量避免在最终的键值生成规则中包含数据项索引 index，以防止出现渲染结果非预期和渲染性能降低。如果业务确实需要使用 index，例如列表需要通过 index 进行条件渲染，开发者需要接受 ForEach 在改变数据源后重新创建组件所带来的性能损耗。

- 为满足键值的唯一性，对于对象数据类型，建议使用对象数据中的唯一 id 作为键值。

- 基本数据类型的数据项没有唯一 ID 属性。如果使用基本数据类型本身作为键值，必须确保数组项无重复。因此，对于数据源会发生变化的场景，建议将基本数据类型数组转化为具备唯一 ID 属性的对象数据类型数组，再使用 ID 属性作为键值生成规则。

3.7 案例一-待办列表案例

本小节我们将通过以上学习的 ArkTs 内容来实现一个“待办”列表案例，该案例最终效果图如下：

当我们完成一条待办时，点击其中一条待办事项呈现效果如下：

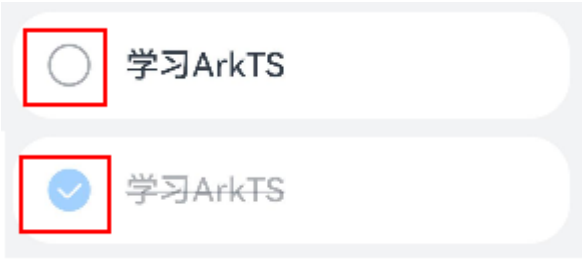
以上应用界面是由一个个页面组成，可以通过 ArkUI 框架声明式 UI 进行构建，声明式 UI 构建页面的过程，其实是组合组件的过程，声明式 UI 的思想，主要体现在两个方面：

- 描述 UI 的呈现结果，而不关心过程
- 状态驱动视图更新

ArkUI 作为 HarmonyOS 应用开发的 UI 开发框架，其使用 ArkTS 语言构建自定义组件，通过组合自定义组件完成页面的构建。下面我们一步步实现这个功能。

3.7.1 准备图片

首先我们准备案例所需要的图片。在项目 ets 目录下创建 images 目录，将“ic_default.png”和“ic_ok.png”放入该目录。两个图片分别用于一条待办未完成和完成后的状态，如下图所示。



3.7.2 自定义组件-ToDoListPage

我们定义“ToDoListPage”组件的主要目的是构建整个待办事项页面。ArkTS 通过 struct 声明组件名，并通过@Component 和@Entry 装饰器，来构成一个自定义组件。使用@Entry 和@Component 装饰的自定义组件作为页面的入口，会在页面加载时首先进行渲染。

整体代码如下：

```
    import DataModel from './DataModel'; // 从'DataModel'文件导入 DataModel 类
import ToDoItem from './ToDoItem'; // 从'ToDoItem'文件导入 ToDoItem 类

@Entry // 标记这是入口组件
@Component // 标记这是一个组件
struct ToDoListPage { // 定义一个名为 ToDoListPage 的结构体，代表待办列表页面
  private totalTasks: Array<string> = []; // 定义一个字符串数组，用于存储总任务

  aboutToAppear() { // 页面即将出现时调用的方法
    this.totalTasks = DataModel.getData(); // 从 DataModel 获取数据并赋值给 totalTasks
  }

  build() { // 构建 UI 界面的方法
    Column({ space: 16 }) { // 创建一个列布局，元素之间的间隔为 16
      Text("待办") // 显示文本“待办”
        .fontSize("28fp") // 字体大小为 28fp
        .fontWeight(FontWeight.Bold) // 字体加粗
        .lineHeight("33vp") // 行高为 33vp
        .width('80%') // 宽度占父容器的 80%
        .margin({ // 设置边距
          top: "24vp", // 顶部边距为 24vp
          bottom: "12vp" // 底部边距为 12vp
        })
        .textAlign(TextAlign.Start) // 文本对齐方式为左对齐

      ForEach(this.totalTasks, (item: string) => { // 遍历 totalTasks 数组
        ToDoItem({ content: item }) // 对于每个任务项，创建一个 ToDoItem 组件
```

```
    }, (item: string) => JSON.stringify(item)) // 使用 JSON.stringify 转换每个项目的键
   }
   .width('100%') // 列布局的宽度为 100%
   .height('100%') // 高度为 100%
   .backgroundColor("#F1F3F5") // 背景颜色设置为#F1F3F5
  }
}
```

当整个页面加载时，默认首先调用“aboutToAppear”方法，在该方法中我们给“totalTasks ”进行加载待办事项数据，这里使用到了“DataMode 类”，参考后续实现。

“build”方法中我们首先加载“Text("待办")”然后通过“ForEach”函数进行循环将各个待办事项进行顺序加载，这里使用到了“ToDoItem”组件用于展示每个待办项，关于该组件参考后续代码实现。

3.7.3 DataModel 类

DataModel 类实现代码如下：

```
    export class DataModel { // 定义并导出一个名为 DataModel 的类
  private tasks: Array<string> = [ // 定义一个私有数组 tasks，用于存储字符串类型的任务
    "早起晨练", // 数组第一个元素：早起晨练
    "准备早餐", // 数组第二个元素：准备早餐
    "阅读名著", // 数组第三个元素：阅读名著
    "学习 ArkTS", // 数组第四个元素：学习 ArkTS
    "看剧放松"  // 数组第五个元素：看剧放松
  ];

  getData(): Array<string> { // 定义一个公开方法 getData，返回一个字符串类型的数组
    return this.tasks; // 返回 tasks 数组
  }
}

export default new DataModel(); // 导出 DataModel 类的一个新实例
```

3.7.4 自定义组件-ToDoItem

ToDoItem 主要来展示每条待办事项，并且在每条待办未完成和完成展示效果不同，实现代码如下:

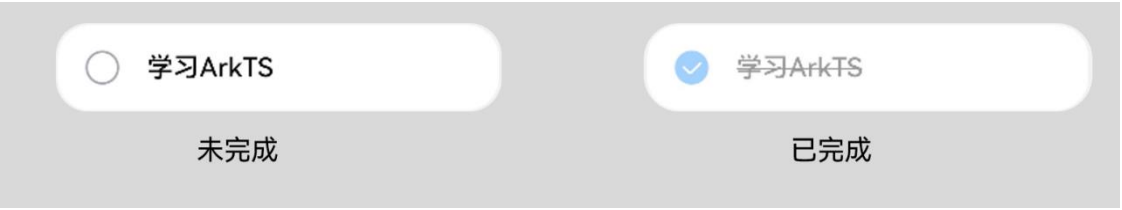
```
    @Component // 标记这是一个组件
export default struct ToDoItem { // 定义并导出名为 ToDoItem 的结构体
  private content?: string; // 定义一个私有属性 content，类型为 string，可选
  @State isComplete: boolean = false; // 使用@State 声明一个状态变量 isComplete，初始值为 false

  @Builder labelIcon(icon: string) { // 定义一个名为 labelIcon 的方法，用于构建标签图标
```

```
    Image(icon) // 创建一个 Image 组件，图标源为 icon 参数
      .objectFit(ImageFit.Contain) // 设置图片对象填充方式为 Contain
      .width("28vp") // 图片宽度为 28vp
      .height("28vp") // 图片高度为 28vp
      .margin("20vp") // 设置边距为 20vp
  }

  build() { // 构建 UI 界面的方法
    Row() { // 创建一个行布局
      if (this.isComplete) { // 如果任务已完成
        this.labelIcon("images/ic_ok.png"); // 显示完成的图标
      } else { // 否则
        this.labelIcon("images/ic_default.png"); // 显示默认图标
      }

      Text(this.content) // 显示文本内容
        .fontSize("20fp") // 字体大小为 20fp
        .fontWeight(500) // 字体粗细为 500
        .opacity(this.isComplete ? 0.4 : 1) // 根据是否完成调整透明度
        .decoration({ type: this.isComplete ? TextDecorationType.LineThrough :
TextDecorationType.None }) // 完成时添加删除线
    }
    .borderRadius(24) // 设置边框圆角为 24
    .backgroundColor("#FFFFFF") // 背景颜色为白色
    .width('93.3%') // 宽度为 93.3%
    .height("64vp") // 高度为 64vp
    .onClick(() => { // 设置点击事件
      this.isComplete = !this.isComplete; // 点击时切换 isComplete 的状态
    })
  }
}
```

以上代码注意如下几点：

1) 实际开发中由于交互，页面的内容可能需要产生变化，以每一个 ToDoItem 为例，其在完成时的状态与未完成时的展示效果是不一样的。声明式 UI 的特点就是 UI 是随数据更改而自动刷新的，我们这里定义了一个类型为 boolean 的变量 isComplete，其被@State 装饰后，框架内建立了数据和视图之间的绑定，其值的改变影响 UI 的显示。

2) 用圆圈和对勾这样两个图片，分别来表示该项是否完成，这部分涉及到内容的切换，需要使用条件渲染 if / else 语法来进行组件的显示与消失，当判断条件为真时，组件为已完成的状态，反之则为未完成。



3) 由于两个 Image 的实现具有大量重复代码，ArkTS 提供了@Builder 装饰器，来修饰一个函数，快速生成布局内容，从而可以避免重复的 UI 描述内容。这里使用@Bulider 声明了一个 labelIcon 的函数，参数为 url，对应要传给 Image 的图片路径。使用时只需要使用 this 关键字访问@Builder 装饰的函数名，即可快速创建布局。

4) 为了让待办项带给用户的体验更符合已完成的效果，给内容的字体也增加了相应的样式变化，这里使用了三目运算符来根据状态变化修改其透明度和文字样式，如 opacity 是控制透明度，decoration 是文字是否有划线。通过 isComplete 的值来控制其变化。

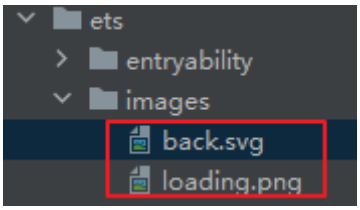
5) 为了实现与用户交互的效果，在组件上添加了 onClick 点击事件，当用户点击该待办项时，数据 isComplete 的更改就能够触发 UI 的更新。

3.8 案例二-水果排行榜案例

本案例使用声明式语法和组件化基础知识，搭建一个可刷新的排行榜页面。在排行榜页面中，使用循环渲染控制语法来实现列表数据渲染，使用@Builder 创建排行列表布局内容，使用装饰器@State、@Prop、@Link 来管理组件状态。最后我们点击系统返回按键，来学习自定义组件生命周期函数。完成效果如图所示：

3.8.1 准备图片

首先准备案例中所需要的图片。在项目 ets 目录中创建 images 目录，放入“back.svg”和“loading.png”两个图片。



3.8.2 RankData

创建水果排行的类。

```
    //RankData 类 - 水果对象
export class RankData {
  name: string;
  vote: string; // 投票数
  id: string;

  constructor(id: string, name: string, vote: string) {
    this.id = id;
    this.name = name;
```

```
    this.vote = vote;
  }
}
```

3.8.3 TitleComponent

创建“TitleComponent”文件,创建一个带有标题和两个交互元素（返回图标和加载图标）的用户界面。图示和代码如下：


```
import AppContext from '@ohos.app.ability.common';  // 从鸿蒙的应用能力通用库中导入AppContext。


@Component  // 使用@Component 装饰器标记这是一个组件。
```

```
export struct TitleComponent {  // 定义一个名为 TitleComponent 的结构体，作为组件的主体。

  @Link isRefreshData: boolean; // 定义一个名为 isRefreshData 的变量，类型为 boolean，用于控制数据是否刷新。
  @State title: string  = "";  // 定义一个名为 title 的状态变量，类型为 string，默认为空字符串。

  build() {  // 定义组件的构建方法。
    Row() {  // 创建一个行布局。
      Row() {  // 在行布局中嵌套另一个行布局。
        Image('images/back.svg')  // 插入一个图片元素，用于显示返回图标。
          .height(21)  // 设置图片的高度为 21。
          .width(21)  // 设置图片的宽度为 21。
          .margin({ right: 18 })  // 设置图片右侧的外边距为 18。
          .onClick(() => {  // 为图片添加点击事件。
            let handler = getContext(this) as AppContext.UIAbilityContext;  // 获取当前组件的上下文，并转换为 UIAbilityContext 类型。
            handler.terminateSelf();  // 调用 terminateSelf 方法结束当前界面。
          })
        Text(this.title)  // 插入一个文本元素，显示标题。
          .fontSize(20)  // 设置文本的字体大小为 20。
      }
      .width('50%')  // 设置内层行布局的宽度为父容器宽度的 50%。
      .height('100%')  // 设置内层行布局的高度为父容器高度的 100%。
      .justifyContent(FlexAlign.Start)  // 设置内层行布局的内容靠左对齐。

      Row() {  // 再创建一个行布局。
        Image('images/loading.png')  // 插入一个图片元素，用于显示加载图标。
          .height(22)  // 设置图片的高度为 22。
          .width(22)  // 设置图片的宽度为 22。
          .onClick(() => {  // 为图片添加点击事件。
```

```
            this.isRefreshData = !this.isRefreshData;  // 切换 isRefreshData 的值。
          })
        }
        .width('50%')  // 设置此行布局的宽度为父容器宽度的 50%。
        .height('100%')  // 设置此行布局的高度为父容器高度的 100%。
        .justifyContent(FlexAlign.End)  // 设置此行布局的内容靠右对齐。
      }
      .width('100%')  // 设置最外层行布局的宽度为 100%。
      .padding({ left: 26, right: 26 })  // 设置左右的内边距为 26。
      .margin({ top: 10 })  // 设置顶部的外边距为 10。
      .height(47)  // 设置最外层行布局的高度为 47。
      .justifyContent(FlexAlign.SpaceAround)  // 设置内容分布方式为平均分布。
    }
  }
```

3.8.4 ListHeaderComponent

创建“ListHeaderComponent”组件，用于创建一个列表头部，显示列标题。图示和代码如下：


```
    @Component  // 使用@Component 装饰器标记这是一个组件。
export struct ListHeaderComponent {  // 定义一个名为 ListHeaderComponent 的结构体，作为组件的主体。

  paddingValue: Padding | Length = 0;  // 定义一个名为 paddingValue 的变量，用于设置内边距，可
```

```
以是 Padding 类型或 Length 类型，默认值为 0。
  widthValue: Length = 0;  // 定义一个名为 widthValue 的变量，用于设置组件的宽度，类型为 Length，默认值为 0。

  build() {  // 定义组件的构建方法。
    Row() {  // 创建一个行布局。
      Text('排名')  // 插入一个文本元素，内容为“排名”。
        .fontSize(14)  // 设置字体大小为 14。
        .width('30%')  // 设置此文本元素的宽度占父容器的 30%。
        .fontWeight(400)  // 设置字体的粗细为 400（正常粗细）。
        .fontColor('#989A9C')  // 设置字体颜色为灰色（#989A9C）。
      Text('种类')  // 插入另一个文本元素，内容为“种类”。
        .fontSize(14)  // 设置字体大小为 14。
        .width('50%')  // 设置此文本元素的宽度占父容器的 50%。
        .fontWeight(400)  // 设置字体的粗细为 400。
        .fontColor('#989A9C')  // 设置字体颜色为灰色。
      Text('得票数')  // 插入第三个文本元素，内容为“得票数”。
        .fontSize(14)  // 设置字体大小为 14。
        .width('20%')  // 设置此文本元素的宽度占父容器的 20%。
        .fontWeight(400)  // 设置字体的粗细为 400。
        .fontColor('#989A9C')  // 设置字体颜色为灰色。
    }
    .width(this.widthValue)  // 设置行布局的宽度，使用之前定义的 widthValue 变量。
    .padding(this.paddingValue)  // 设置行布局的内边距，使用之前定义的 paddingValue 变量。
  }
}
```

3.8.5 ListItemComponent

创建“ListItemComponent”用于在 UI 界面上显示一行水果名称和得票数，同时包含交互效果。图示和代码如下：


```
    @Component  // 使用 @Component 装饰器定义一个组件。
```

```
export struct ListItemComponent {  // 定义一个名为 ListItemComponent 的结构体，用于表示列表项组件。
  index?: number;  // 可选属性：表示列表项的索引（位置）。
  private name?: string;  // 私有属性：存储列表项的名称。
  vote: string = '';  // 属性：存储列表项的得票数，默认为空字符串。
  isSwitchDataSource: boolean = false;  // 属性：标志是否切换数据源，默认为假（false）。

  @State isChange: boolean = false;  // 使用 @State 装饰器，当 isChange 的值改变时，会自动刷新 UI。

  build() {  // 定义 build 方法，用于构建组件的 UI 界面。
    Row() {  // 使用 Row 组件创建水平布局。
      Column() {  // 使用 Column 组件创建垂直布局。
        if (this.isRenderCircleText()) {  // 判断是否渲染圆形文本。
          if (this.index !== undefined) {  // 如果索引不为空，则渲染圆形文本。
            this.CircleText(this.index);  // 调用 CircleText 方法来渲染圆形文本。
          }
        } else {
          Text(this.index?.toString())  // 否则，渲染普通文本。
            .lineHeight(24)  // 设置文本的行高为 24。
            .textAlign(TextAlign.Center)  // 设置文本对齐方式为居中。
            .width(24)  // 设置文本宽度为 24。
            .fontWeight('400')  // 设置文本的字体粗细为 400。
            .fontSize(14)  // 设置文本字体大小为 14。
        }
      }
      .width('30%')  // 设置列的宽度为容器宽度的 30%。
      .alignItems(HorizontalAlign.Start)  // 设置水平对齐方式为开始。

      Text(this.name)  // 创建一个 Text 组件用于显示名称。
        .width('50%')  // 设置文本宽度为容器宽度的 50%。
        .fontWeight('500')  // 设置字体粗细为 500。
        .fontSize(16)  // 设置字体大小为 16。
        .fontColor(this.isChange ? "#007DFF" : "#182431")  // 设置字体颜色，当 isChange 为真时为蓝色，否则为深灰色。
      Text(this.vote)  // 创建一个 Text 组件用于显示得票数。
        .width('20%')  // 设置文本宽度为容器宽度的 20%。
        .fontWeight('400')  // 设置字体粗细为 400。
        .fontSize(14)  // 设置字体大小为 14。
        .fontColor(this.isChange ? "#007DFF" : "#182431")  // 设置字体颜色，同上。
    }
    .height(48)  // 设置行组件的高度为 48。
    .width('100%')  // 设置行组件的宽度为 100%。
    .onClick(() => {  // 设置点击事件处理器。
      this.isSwitchDataSource = !this.isSwitchDataSource;  // 切换数据源标志。
      this.isChange = !this.isChange;  // 切换改变状态标志。
    })
  }
```

```
  @Builder CircleText(index: number) {  // 定义一个名为 CircleText 的方法，用于创建圆形文本。
    Row() {  // 使用 Row 组件。
      Text(this.index?.toString())  // 创建 Text 组件显示索引。
        .fontWeight('400')  // 设置字体粗细为 400。
        .fontSize(14)  // 设置字体大小为 14。
        .fontColor(Color.White);  // 设置字体颜色为白色。
    }
    .justifyContent(FlexAlign.Center)  // 设置内容居中对齐。
    .borderRadius(24)  // 设置边框半径为 24，形成圆形。
    .size({ width: 24, height: 24})  // 设置尺寸为 24x24。
    .backgroundColor("#007dff")  // 设置背景颜色为蓝色。
  }

  isRenderCircleText(): boolean {  // 定义一个方法判断是否渲染圆形文本。
    return this.index === 1 || this.index === 2 || this.index === 3;  // 对于索引 1、2、3 的项返回
真，其他返回假。
  }
}
```

3.8.6 RankPage

定义“rankPage”的组件，用于显示一个排行榜页面，包括标题、列表头部和列表项。将以上所有组件组合形成最终排行榜效果。

```
    import { ListHeaderComponent } from './ListHeaderComponent';  // 导入
ListHeaderComponent 组件。
import { ListItemComponent } from './ListItemComponent';  // 导入 ListItemComponent 组件。
import { RankData } from './RankData';  // 导入 RankData 类。
import { TitleComponent } from './TitleComponent';  // 导入 TitleComponent 组件。

@Entry  // 使用 @Entry 装饰器标识入口组件。
@Component  // 使用 @Component 装饰器定义组件。
struct RankPage {  // 定义名为 RankPage 的结构体组件。

  // 使用 @State 装饰器定义状态，存储两个不同的排行数据源。
  // 初始化第一个数据源，包含苹果、葡萄等水果的排名和得票数。
  @State dataSource1: RankData[] = [
    new RankData('1', '苹果', '12080'),
    new RankData('2', '葡萄', '10320'),
    new RankData('3', '西瓜', '9801'),
    new RankData('4', '香蕉', '8431'),
    new RankData('5', '菠萝', '7546'),
    new RankData('6', '榴莲', '7431'),
    new RankData('7', '红葡萄', '7187'),
    new RankData('8', '梨子', '7003'),
    new RankData('9', '杨桃', '6794'),
```

```
  new RankData('10','番石榴', '6721')
];

// 初始化第二个数据源，包含另一组水果的排名和得票数。
@State dataSource2: RankData[] = [
  new RankData('11', '西瓜', '8836'),
  new RankData('12', '苹果', '8521'),
  new RankData('13', '香蕉', '8431'),
  new RankData('14', '葡萄', '7909'),
  new RankData('15', '红葡萄', '7547'),
  new RankData('16', '梨子', '7433'),
  new RankData('17', '菠萝', '7186'),
  new RankData('18', '榴莲', '7023'),
  new RankData('19', '番石榴', '6794'),
  new RankData('20', '杨桃', '6721')

];
@State isSwitchDataSource: boolean = true;  // 定义一个状态，用于控制是否切换数据源。

private clickBackTimeRecord: number = 0;  // 私有变量，记录点击系统导航返回按钮的时间。

build() {  // 定义 build 方法来构建 UI 界面。
  Column() {  // 使用 Column 组件创建垂直布局。
    TitleComponent({ isRefreshData: $isSwitchDataSource, title: "排行榜" })  // 添加标题组件。

    ListHeaderComponent({  // 添加列表头部组件。
      paddingValue: {  // 设置内边距。
        left: 15,
        right: 15
      },
      widthValue: '90%'  // 设置宽度值。
    })
      .margin({  // 设置外边距。
        top: 20,
        bottom: 15
      })

    this.RankList('90%')  // 调用 RankList 方法创建列表。
  }
  .backgroundColor("#F1F3F5")  // 设置背景颜色。
  .height('100%')  // 设置高度为 100%。
  .width('100%')  // 设置宽度为 100%。
}

@Builder RankList(widthValue: Length) {  // 使用 @Builder 装饰器定义 RankList 方法。
  Column() {  // 使用 Column 组件。
    List() {  // 创建 List 组件。
      ForEach(this.isSwitchDataSource ? this.dataSource1 : this.dataSource2,  // 根据状态选择数据
```

```
源。
        (item: RankData, index?: number) => {  // 遍历数据项。
          ListItem() {  // 创建列表项。
            ListItemComponent({  // 使用 ListItemComponent 组件。
              index: (Number(index) + 1),  // 设置索引。
              name: item.name,  // 设置名称。
              vote: item.vote,  // 设置得票数。
              isSwitchDataSource: this.isSwitchDataSource  // 传递数据源切换状态。
            })
          }
        }, (item: RankData) => JSON.stringify(item))  // 使用 JSON.stringify 作为键函数。
      }
      .width('100%')  // 设置列表宽度为 100%。
      .height('65%')  // 设置列表高度为 65%。
      .divider({ strokeWidth: 1 })  // 设置列表项之间的分割线。
    }
    .padding({  // 设置内边距。
      left: 15,
      right: 15
    })
    .borderRadius(20)  // 设置边框圆角。
    .width(widthValue)  // 设置组件宽度。
    .alignItems(HorizontalAlign.Center)  // 设置水平对齐方式为居中。
    .backgroundColor(Color.White)  // 设置背景颜色为白色。
  }
}
```

3.9 章节习题

1. **循环渲染 ForEach 可以从数据源中迭代获取数据，并为每个数组项创建相应的组件。正确(True)**

2. **@Link 变量不能在组件内部进行初始化。正确(True)**

3. **用哪一种装饰器修饰的 struct 表示该结构体具有组件化能力？A**

A.@Component B.@Entry C.@Builder D.@Preview

4. **用哪一种装饰器修饰的自定义组件可作为页面入口组件？B**

A.@Component B.@Entry C.@Builder D.@Preview

5. **下面哪些函数是自定义组件的生命周期函数？ABCDE**

A.aboutToAppear B.aboutToDisappear C.onPageShow D.onPageHide E.onBackPress

6. **下面哪些装饰器可以用于管理自定义组件中变量的状态？CD**

A.@Component B.@Entry C.@State D.@Link

4 第四章 应用程序框架

4.1 UIAbility

4.1.1 UIAbility 介绍

UIAbility 是一种包含用户界面的应用组件，主要用于和用户进行交互。与应用程序交互界面形式有如下三种。

1. 点击桌面图标进入应用

例如在桌面点击图库图标，拉起图库应用，看到的图库应用就是基于 UI Ability 实现的一个应用实例。

2. 一个应用拉起另外一个应用

我们也可以通过应用之间的互相跳转进入另外一个应用，例如在图库应用中通过图片分享进入备忘录应用，进行分享图片的保存操作，看到的图库应用和备忘录应用 均是基于 UI Ability 实现的一个应用实例。

3. 最近任务列表切回应用

我们还可以从最近任务列表中找到该应用，并再次进入应用，看到的任务列表中的应用任务，也都是基于 UIAbility 实现的一个应用实例。

以上每一个 UIAbility 实例，都对应于一个最近任务列表中的任务。UIAbility 作为系统调度的单元，提供窗口用于界面绘制。一个应用可以有一个 UIAbility，也可以有多个 UIAbility，如下图所示。例如浏览器应用可以通过一个 UIAbility 结合多页面的形式让用户进行的搜索和浏览内容；而聊天应用增加一个“外卖功能”的场景，则可以将聊天应用中“外卖功能”的内容独立为一个 UIAbility，当用户打开聊天应用的“外卖功能”，查看外卖订单详情，此时有新的聊天消息，即可以通过最近任务列表切换回到聊天窗口继续进行聊天对话。

一个 UIAbility 可以对应于多个页面，建议将一个独立的功能模块放到一个 UIAbility 中，以多页面的形式呈现。例如新闻应用在浏览内容的时候，可以进行多页面的跳转使用。

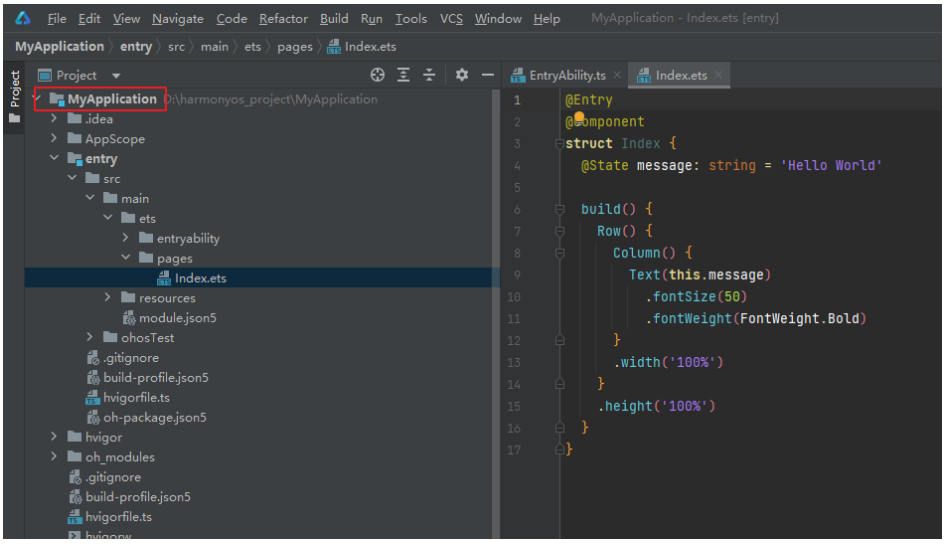
4.1.2 UIAbility 内页面创建

UIAbility 的数据传递包括有 UIAbility 内页面的跳转和数据传递、UIAbility 间的数据跳转和数据传递，本小节主要讲解 UIAbility 内页面的跳转和数据传递。

在一个应用包含一个 UIAbility 的场景下，可以通过新建多个页面来实现和丰富应用的内容。这会涉及到 UIAbility 内页面的新建以及 UIAbility 内页面的跳转和数据传递。下面创建一个新的项目学习在 UIAbility 中创建页面。

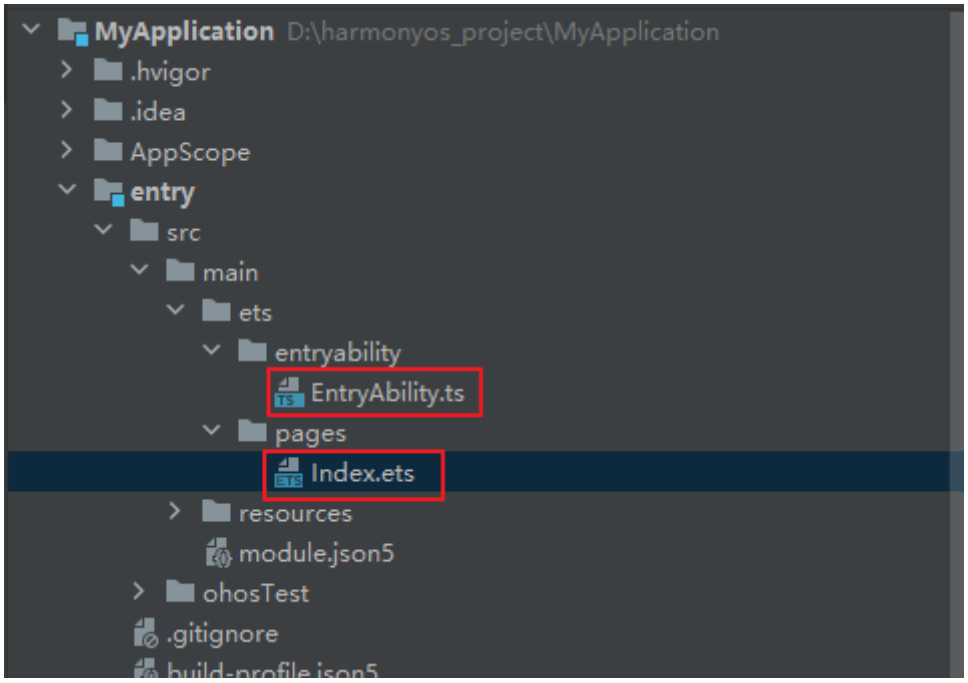
4.1.2.1 新建项目

打开 DevEco Studio，选择一个 Empty Ability 工程模板，创建一个工程，例如命名为 MyApplication。



在 src/main/ets/entryability 目录下，初始会生成一个 UIAbility 文件 EntryAbility.ts。可以在 EntryAbility.ts 文件中根据业务需要实现 UIAbility 的生命周期回调内容。

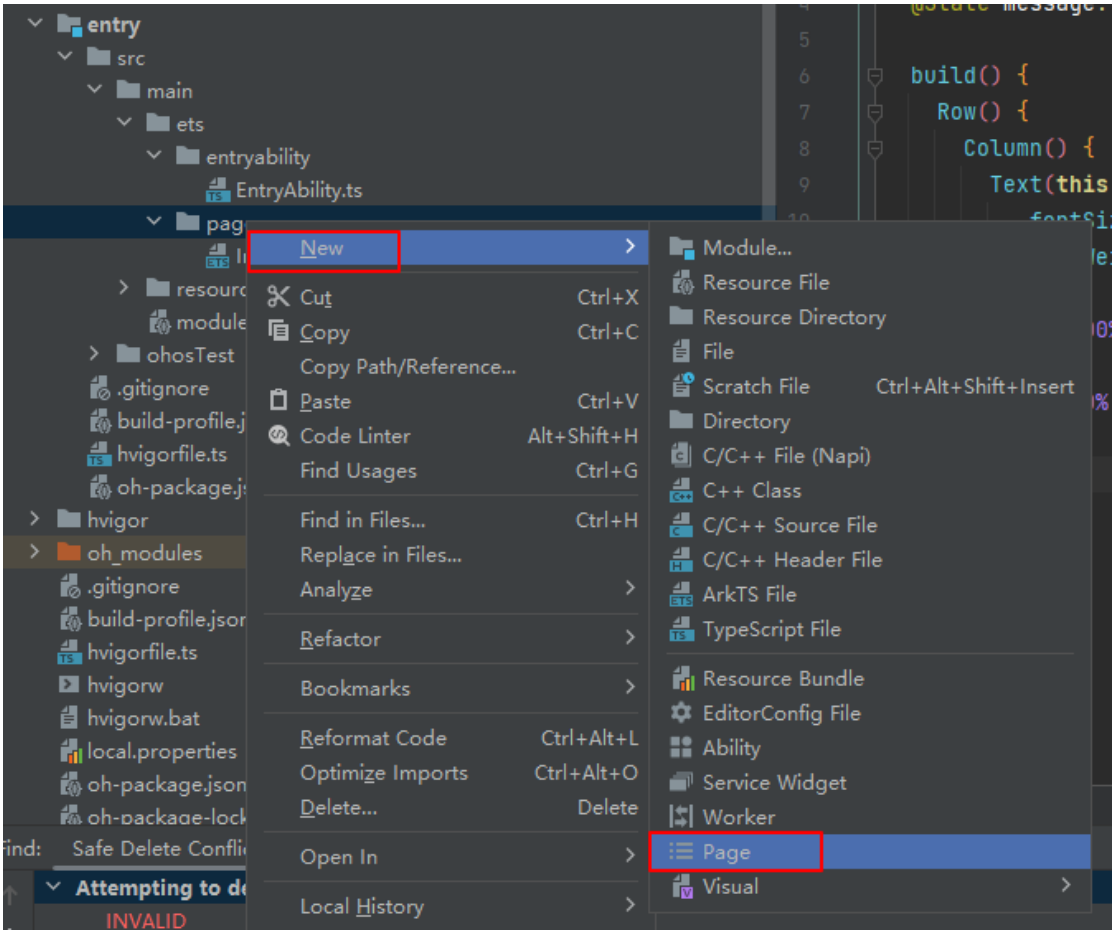
在 src/main/ets/pages 目录下，会生成一个 Index 页面。这也是基于 UIAbility 实现的应用的入口页面。可以在 Index 页面中根据业务需要实现入口页面的功能。

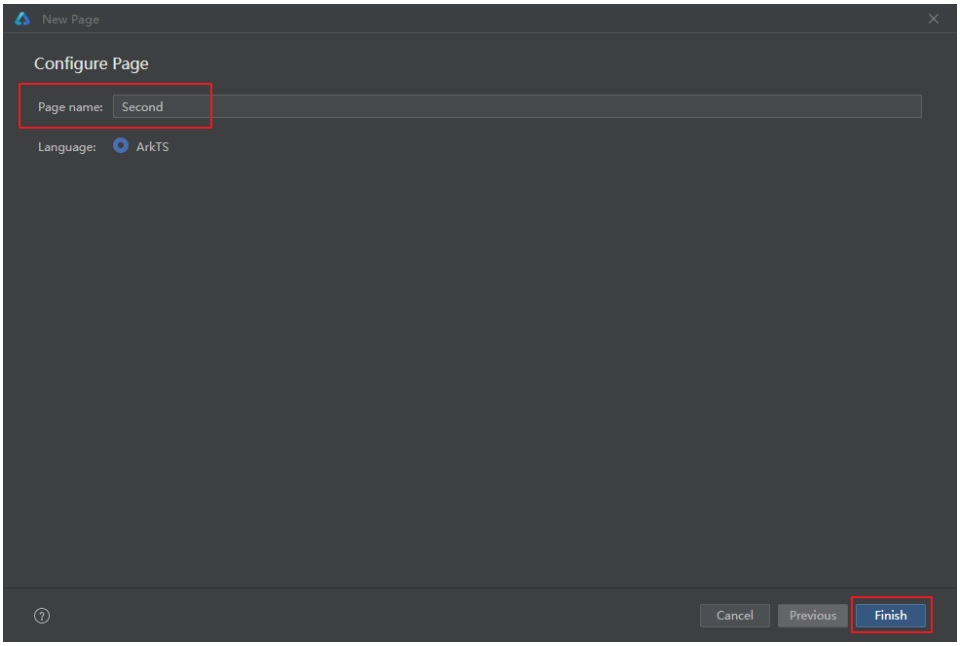


4.1.2.2 新建跳转页面

在 src/main/ets/pages 目录下，右键 New->Page，新建一个 Second 页面，用于实现页面间的跳转和数据传递。

一个应用一般会有多个页面，为了实现页面的跳转和数据传递，需要新建一个页面。在原有 Index 页面的基础上，新建一个页面，例如命名为 Second.ets。


entry
src
main
ets
entryability
EntryAbility.ts
New
Cut Ctrl+X
Copy Ctrl+C
Copy Path/Reference...
Paste Ctrl+V
Code Linter Alt+Shift+H
Find Usages Ctrl+G
Find in Files... Ctrl+H
Replace in Files...
Analyze
Refactor
Bookmarks
Reformat Code Ctrl+Alt+L
Optimize Imports Ctrl+Alt+O
Delete... Delete
Open In
Local History
Module...
Resource File
Resource Directory
File
Scratch File Ctrl+Alt+Shift+Insert
Directory
C/C++ File (Napi)
C++ Class
C/C++ Source File
C/C++ Header File
ArkTS File
TypeScript File
Resource Bundle
EditorConfig File
Ability
Service Widget
Worker
Page
Visual
build() {
Row() {
Column() {
Text(this


New Page
Configure Page
Page name: Second
Language: ArkTS
Cancel
Previous
Finish

4.1.2.3 页面跳转和参数接收

页面间的导航可以通过页面路由 router 模块来实现。页面路由模块根据页面 url 找到目标页面，从而实现跳转。通过页面路由模块，可以使用不同的 url 访问不同的页面，包括跳转到 UIAbility 内的指定页面、用 UIAbility 内的某个页面替换当前页面、返回上一页面或指定的页面等。

在使用页面路由之前，需要先导入 router 模块，如下代码所示。

```
import router from '@ohos.router';
```

页面跳转的几种方式，根据需要选择一种方式跳转即可。

方式一：API9 及以上，router.pushUrl()方法新增了 mode 参数，可以将 mode 参数配置为 router.RouterMode.Single 单实例模式和 router.RouterMode.Standard 多实例模式。

在单实例模式下：如果目标页面的 url 在页面栈中已经存在同 url 页面，离栈顶最近同 url 页面会被移动到栈顶，移动后的页面为新建页，原来的页面仍然存在栈中，页面栈的元素数量不变；如果目标页面的 url 在页面栈中不存在同 url 页面，按多实例模式跳转，页面栈的元素数量会加 1。

注意：当页面栈的元素数量较大或者超过 32 时，可以通过调用 router.clear()方法清除页面栈中的所有历史页面，仅保留当前页面作为栈顶页面。

```
router.pushUrl({
  url: 'pages/Second',
  params: {
    src: 'Index 页面传来的数据',
  }
}, router.RouterMode.Single)
```

方式二：API9 及以上，router.replaceUrl()方法新增了 mode 参数，可以将 mode 参数配置为 router.RouterMode.Single 单实例模式和 router.RouterMode.Standard 多实例模式。

在单实例模式下：如果目标页面的 url 在页面栈中已经存在同 url 页面，离栈顶最近同 url 页面会被移动到栈顶，替换当前页面，并销毁被替换的当前页面，移动后的页面为新建页，页面栈的元素数量会减 1；如果目标页面的 url 在页面栈中不存在同 url 页面，按多实例模式跳转，页面栈的元素数量不变。

```
router.replaceUrl({
```

```
      url: 'pages/Second',
      params: {
        src: 'Index 页面传来的数据',
      }
    }, router.RouterMode.Single)
```

在 Index 页面中首先导入 Router 路由模块并添加一些基础样式，以及增加一个 Button 按钮、添加一个 onclick 事件。在 onClick 事件中通过调用 Router.pushURL 方法将 URL 指定为需要跳转的 Second 页面路径即可。如下是 Index 页面代码：

```
    import router from '@ohos.router'; // 导入鸿蒙操作系统的路由模块，用于页面间的导航

@Entry // 标记这个组件为应用的入口点
@Component // 标记这个结构体为一个组件，用于 UI 界面构建
struct IndexPage {
  @State message: string = 'Index Page'; // 定义一个状态变量 message，初始值为 'Index Page'

  build() {
    Row() { // 创建一个水平布局的容器
      Column() { // 在水平布局内创建一个垂直布局的容器
        Text(this.message) // 创建一个文本组件，内容为 message 变量的值
          .fontSize('38') // 设置文本的字体大小为 38
          .fontWeight(FontWeight.Bold) // 设置文本为粗体
        Blank() // 插入一个空白组件，用于间隔
        Button('Next') // 创建一个标签为 'Next' 的按钮
          .fontSize(16) // 设置按钮的字体大小为 16
          .width(296) // 设置按钮的宽度为 296
          .height(40) // 设置按钮的高度为 40
          .backgroundColor("#007DFF") // 设置按钮的背景颜色
          .onClick(() => { // 为按钮添加点击事件处理函数
            router.pushUrl({ // 使用路由器的 pushUrl 方法进行页面跳转
              url: 'pages/Second', // 目标页面的 URL
              params: {
                src: 'Index 页面传来的数据' // 传递给目标页面的参数
              }
            });
          })
      }
      .width('100%') // 设置垂直布局容器的宽度为 100%
      .height(140) // 设置垂直布局容器的高度为 140
    }
    .height('100%') // 设置水平布局容器的高度为 100%
    .backgroundColor("#F1F3F5") // 设置水平布局容器的背景颜色
  }
```

```
}
```

接下来，在 Second 页面中如何进行自定义参数的接收呢？通过调用 router.getParams()方法获取 Index 页面传递过来的自定义参数。如下示例：

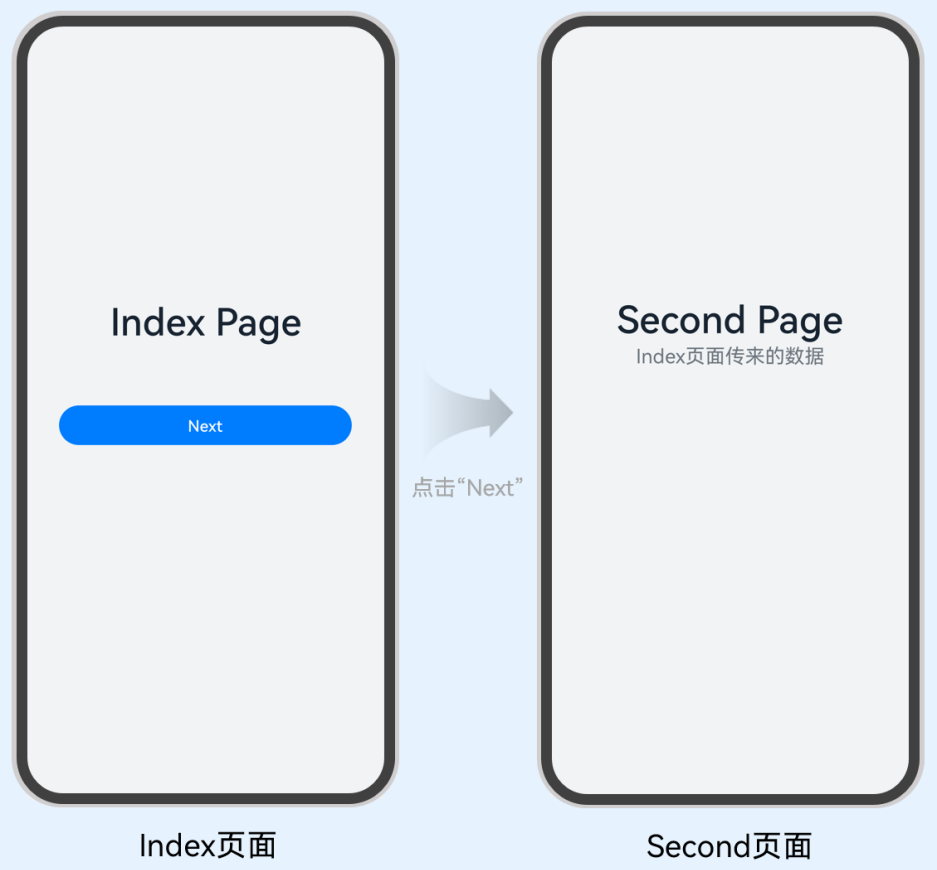
```
import router from '@ohos.router';


@Entry
@Component
struct Second {
  @State src: string = (router.getParams() as Record<string, string>)['src'];
  // 页面刷新展示
  ...
}
```

完整的 Second 页面代码如下：

```
import router from '@ohos.router'; // 导入鸿蒙操作系统的路由模块，用于页面间的导航

@Entry // 标记这个组件为应用的入口点
@Component // 标记这个结构体为一个组件，用于 UI 界面构建
struct SecondPage {
  @State message: string = 'Second Page'; // 定义一个状态变量 message，初始值为 'Second Page'
  @State src: string = (router.getParams() as Record<string, string>)['src']; // 从路由参数中获取 'src' 参数的值并存储在状态变量 src 中

  build() {
    Row() { // 创建一个水平布局的容器
      Column() { // 在水平布局内创建一个垂直布局的容器
        Text(this.message) // 创建一个文本组件，内容为 message 变量的值
          .fontSize(38) // 设置文本的字体大小为 38
          .fontWeight(FontWeight.Bold) // 设置文本为粗体
        Text(this.src) // 创建另一个文本组件，显示从上一个页面传递来的 src 参数的值
          .fontSize(20) // 设置文本的字体大小为 20
          .opacity(0.6) // 设置文本的透明度为 0.6
        Blank() // 插入一个空白组件，用于间隔
        Button("Back") // 创建一个标签为 'Back' 的按钮
          .fontSize(16) // 设置按钮的字体大小为 16
          .width(296) // 设置按钮的宽度为 296
          .height(40) // 设置按钮的高度为 40
          .backgroundColor("#007DFF") // 设置按钮的背景颜色
          .onClick(() => { // 为按钮添加点击事件处理函数
```

```
          router.back(); // 使用路由器的 back 方法返回上一个页面
        })
    }
    .width('100%') // 设置垂直布局容器的宽度为 100%
    .height(140) // 设置垂直布局容器的高度为 140
  }
  .height('100%') // 设置水平布局容器的高度为 100%
  .backgroundColor("#F1F3F5") // 设置水平布局容器的背景颜色
 }
}
```

以下是从 Index 页面跳转到 Seconds 页面的预览图：



4.1.2.4 页面返回和参数接收

在 Second 页面中，完成了一些功能操作之后，希望能返回到 Index 页面，那我们要如何实现呢?

在 Second 页面中，可以通过调用 router.back()方法实现返回到上一个页面，或者在调用 router.back()方法时增加可选的 options 参数（增加 url 参数）返回到指定页面。

注意如下几点：

- 调用 router.back()返回的目标页面需要在页面栈中存在才能正常跳转。

- 例如调用 router.pushUrl()方法跳转到 Second 页面，在 Second 页面可以通过调用 router.back()方法返回到上一个页面。

- 例如调用 router.clear()方法清空了页面栈中所有历史页面，仅保留当前页面，此时则无法通过调用 router.back()方法返回到上一个页面。

返回上一个页面代码示例：

```
router.back();
```

返回到指定页面代码示例：

```
router.back({ url: 'pages/Index' });
```

效果示意如下图所示。在 Second 页面中，点击“Back”后，即可从 Second 页面返回到 Index 页面。

页面返回可以根据业务需要增加一个询问对话框。即在调用 router.back()方法之前，可以先调用 router.showAlertBeforeBackPage()方法开启页面返回询问对话框功能。

在 second 页面中修改部分代码如下：

```
    … …

    Button("Back") // 创建一个标签为 'Back' 的按钮
  .fontSize(16) // 设置按钮的字体大小为 16
  .width(296) // 设置按钮的宽度为 296
  .height(40) // 设置按钮的高度为 40
  .backgroundColor("#007DFF") // 设置按钮的背景颜色
  .onClick(() => { // 为按钮添加点击事件处理函数
    router.showAlertBeforeBackPage({message:'确定返回上个页面？'});
    router.back(); // 使用路由器的 back 方法返回上一个页面
  })
}

    … …
```

弹窗效果如下：

在 Second 页面中，调用 router.back()方法返回上一个页面或者返回指定页面时，还可以根据需要继续增加自定义参数，例如在返回时增加一个自定义参数 src。

Second 页面部分修改代码如下：

```
    … …

    Button("Back") // 创建一个标签为 'Back' 的按钮
  .fontSize(16) // 设置按钮的字体大小为 16
  .width(296) // 设置按钮的宽度为 296
  .height(40) // 设置按钮的高度为 40
  .backgroundColor("#007DFF") // 设置按钮的背景颜色
  .onClick(() => { // 为按钮添加点击事件处理函数
    router.showAlertBeforeBackPage({message:'确定返回上个页面？'});
    router.back({
      url: 'pages/Index',
      params: {
        src: 'Second 页面传来的数据',
```

```
    }
  }); // 使用路由器的 back 方法返回上一个页面
 })

    ... ...
```

特别注意：调用 router.back()方法，不会新建页面，返回的是原来的页面，在原来页面中@State 声明的变量不会重复声明，以及也不会触发页面的 aboutToAppear()生命周期回调，因此无法直接在变量声明以及页面的 aboutToAppear()生命周期回调中接收和解析 router.back()传递过来的自定义参数。

那么在 Index 页面中如何接收 Second 页面传递过来的参数呢？可以放在业务需要的位置进行参数解析。如下在 Index 代码中加入如下内容用于接收从 Second 页面中传递回来的参数。

```
    ... ...

    @State src: string = '';
onPageShow() {
  this.src = (router.getParams() as Record<string, string>)['src'];
}

    ... ...
```

同时在 Index 页面的 build 组件中加入如下展示 Second 页面返回的数据 Text 组件：

```
    ... ...

    Text(this.src) // 创建另一个文本组件，显示从上一个页面传递来的 src 参数的值
  .fontSize(20) // 设置文本的字体大小为 20
  .opacity(0.6) // 设置文本的透明度为 0.6

    ... ...
```

修改好代码后可以看到最终预览如下：

4.1.3 UIAbility 的生命周期

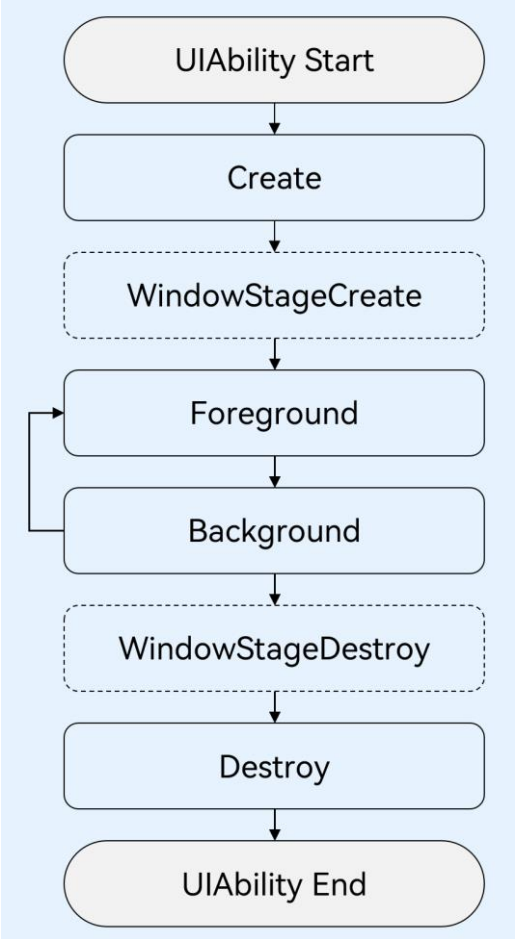
当用户浏览、切换和返回到对应应用的时候，应用中的 UIAbility 实例会在其生命周期的不同状态之间转换。UIAbility 类提供了很多回调，通过这些回调可以知晓当前 UIAbility 的某个状态已经发生改变：例如 UIAbility 的创建和销毁，或者 UIAbility 发生了前后台的状态切换。

例如从桌面点击图库应用图标，到启动图库应用，应用的状态经过了从创建到前台展示的状态变化。如下图所示。

回到桌面，从最近任务列表，切换回到图库应用，应用的状态经过了从后台到前台展示的状态变化。如下图所示。

在 UIAbility 的使用过程中，会有多种生命周期状态。掌握 UIAbility 的生命周期，对于应用的开发非常重要。

为了实现多设备形态上的裁剪和多窗口的可扩展性，系统对组件管理和窗口管理进行了解耦。UIAbility 的生命周期包括 Create、Foreground、Background、Destroy 四个状态，WindowStageCreate 和 WindowStageDestroy 为窗口管理器（WindowStage）在 UIAbility 中管理 UI 界面功能的两个生命周期回调，从而实现 UIAbility 与窗口之间的弱耦合。如下图所示。



- **Create 状态**

在 UIAbility 实例创建时触发，系统会调用 onCreate 回调。可以在 onCreate 回调中进行相关初始化操作。

```
import UIAbility from '@ohos.app.ability.UIAbility';
import window from '@ohos.window';

export default class EntryAbility extends UIAbility {
    onCreate(want: Want, launchParam: AbilityConstant.LaunchParam) {
        // 应用初始化
        ...
    }
    ...
}
```

例如用户打开电池管理应用，在应用加载过程中，在 UI 页面可见之前，可以在 onCreate 回调中读取当前系统的电量情况，用于后续的 UI 页面展示。

- **WindowStage**

UIAbility 实例创建完成之后，在进入 Foreground 之前，系统会创建一个 WindowStage。每一个 UIAbility 实例都对应持有一个 WindowStage 实例。

WindowStage 为本地窗口管理器，用于管理窗口相关的内容，例如与界面相关的获焦/失焦、可见/不可见。可以在 onWindowStageCreate 回调中，设置 UI 页面加载、设置 WindowStage 的事件订阅。在 onWindowStageCreate(windowStage)中通过 loadContent 接口设置应用要加载的页面。

```
import UIAbility from '@ohos.app.ability.UIAbility';
import window from '@ohos.window';

export default class EntryAbility extends UIAbility {
  ...

  onWindowStageCreate(windowStage: window.WindowStage) {
    // 设置 UI 页面加载
    // 设置 WindowStage 的事件订阅（获焦/失焦、可见/不可见）
    ...

    windowStage.loadContent('pages/Index', (err, data) => {
      ...
    });
  }
  ...
}
```

例如用户打开游戏应用，正在打游戏的时候，有一个消息通知，打开消息，消息会以弹窗的形式弹出在游戏应用的上方，此时，游戏应用就从获焦切换到了失焦状态，消息应用切换到了获焦状态。对于消息应用，在 onWindowStageCreate 回调中，会触发获焦的事件回调，可以进行设置消息应用的背景颜色、高亮等操作。

- **Foreground 和 Background 状态**

Foreground 和 Background 状态分别在 UIAbility 切换至前台或者切换至后台时触发。分别对应于 onForeground 回调和 onBackground 回调。

onForeground 回调，在 UIAbility 的 UI 页面可见之前，即 UIAbility 切换至前台时触发。可以在 onForeground 回调中申请系统需要的资源，或者重新申请在 onBackground 中释放的资源。

onBackground 回调，在 UIAbility 的 UI 页面完全不可见之后，即 UIAbility 切换至后台时候触发。可以在 onBackground 回调中释放 UI 页面不可见时无用的资源，或者在此回调中执行较为耗时的操作，例如状态保存等。

```
import UIAbility from '@ohos.app.ability.UIAbility';
import window from '@ohos.window';

export default class EntryAbility extends UIAbility {
    ...

    onForeground() {
        // 申请系统需要的资源，或者重新申请在 onBackground 中释放的资源
        ...
    }

    onBackground() {
        // 释放 UI 页面不可见时无用的资源，或者在此回调中执行较为耗时的操作
        // 例如状态保存等
        ...
    }
}
```

例如用户打开地图应用查看当前地理位置的时候，假设地图应用已获得用户的定位权限授权。在 UI 页面显示之前，可以在 onForeground 回调中打开定位功能，从而获取到当前的位置信息。

当地图应用切换到后台状态，可以在 onBackground 回调中停止定位功能，以节省系统的资源消耗。

- **onWindowStageDestroy**

前面我们了解了 UIAbility 实例创建时的 onWindowStageCreate 回调的相关作用。

对应于 onWindowStageCreate 回调。在 UIAbility 实例销毁之前，则会先进入 onWindowStageDestroy 回调，我们可以在该回调中释放 UI 页面资源。

```
import UIAbility from '@ohos.app.ability.UIAbility';
```

```
import window from '@ohos.window';

export default class EntryAbility extends UIAbility {
    ...

    onWindowStageDestroy() {
        // 释放 UI 页面资源
        ...
    }
}
```

例如在 onWindowStageCreate 中设置的获焦/失焦等 WindowStage 订阅事件。

- **Destroy 状态**

Destroy 状态在 UIAbility 销毁时触发。可以在 onDestroy 回调中进行系统资源的释放、数据的保存等操作。

```
import UIAbility from '@ohos.app.ability.UIAbility';
import window from '@ohos.window';

export default class EntryAbility extends UIAbility {
    ...

    onDestroy() {
        // 系统资源的释放、数据的保存等
        ...
    }
}
```

例如用户使用应用的程序退出功能，会调用 UIAbilityContext 的 terminalSelf()方法，从而完成 UIAbility 销毁。或者用户使用最近任务列表关闭该 UIAbility 实例时，也会完成 UIAbility 的销毁。

4.1.4 UIAbility 的启动模式

UIAbility 当前支持 singleton（单实例模式）、multiton（多实例模式）和 specified（指定实例模式）3 种启动模式。这 3 中启动模式分别对应如下三种场景：

- 对于浏览器或者新闻等应用，用户在打开该应用，并浏览访问相关内容后，回到桌面，再次打开该应用，显示的仍然是用户当前访问的界面。

- 对于应用的分屏操作，用户希望使用两个不同应用（例如备忘录应用和图库应用）之间进行分屏，也希望能使用同一个应用（例如备忘录应用自身）进行分屏。

- 对于文档应用，用户从文档应用中打开一个文档内容，回到文档应用，继续打开同一个文档，希望打开的还是同一个文档内容。

下面详细介绍这 3 中启动模式。

4.1.4.1 singleton-单实例模式

当用户打开浏览器或者新闻等应用，并浏览访问相关内容后，回到桌面，再次打开该应用，显示的仍然是用户当前访问的界面。

这种情况下可以将 UIAbility 配置为 singleton（单实例模式）。每次调用 startAbility()方法时，如果应用进程中该类型的 UIAbility 实例已经存在，则复用系统中的 UIAbility 实例，系统中只存在唯一一个该 UIAbility 实例。

即在最近任务列表中只存在一个该类型的 UIAbility 实例。

singleton 启动模式，也是默认情况下的启动模式。singleton 启动模式，每次调用 startAbility()启动 UIAbility 时，如果应用进程中该类型的 UIAbility 实例已经存在，则复用系统中的 UIAbility 实例，系统中只存在唯一一个该 UIAbility 实例。

singleton 启动模式的开发使用，在 module.json5 文件中的“launchType”字段配置为“singleton”即可。

```
{
```

```
    "module": {
      ...
      "abilities": [
        {
          "launchType": "singleton",
          ...
        }
      ]
    }
}
```

4.1.4.2 multiton-多实例模式

用户在使用分屏功能时，希望使用两个不同应用（例如备忘录应用和图库应用）之间进行分屏，也希望能使用同一个应用（例如备忘录应用自身）进行分屏。

这种情况下可以将 UIAbility 配置为 multiton（多实例模式）。每次调用 startAbility()方法时，都会在应用进程中创建一个该类型的 UIAbility 实例。

即在最近任务列表中可以看到有多个该类型的 UIAbility 实例。

multiton 启动模式，每次调用 startAbility()方法时，都会在应用进程中创建一个该类型的 UIAbility 实例。

multiton 启动模式的开发使用，在 module.json5 文件中的“launchType”字段配置为“multiton”即可。

```
{
  "module": {
    ...
    "abilities": [
      {
        "launchType": "multiton",
        ...
      }
    ]
  }
}
```

4.1.4.3 specified-指定实例模式

用户打开文档应用，从文档应用中打开一个文档内容，回到文档应用，继续打开同一个文档，希望打开的还是同一个文档内容；以及在文档应用中新建一个新的文档，每次新建文档，希望打开的都是一个新的空白文档内容。

这种情况下可以将 UIAbility 配置为 specified（指定实例模式）。在 UIAbility 实例新创建之前，允许开发者为该实例创建一个字符串 Key，新创建的 UIAbility 实例绑定 Key 之后，后续每次调用 startAbility 方法时，都会询问应用使用哪个 Key 对应的 UIAbility 实例来响应 startAbility 请求。如果匹配有该 UIAbility 实例的 Key，则直接拉起与之绑定的 UIAbility 实例，否则创建一个新的 UIAbility 实例。运行时由 UIAbility 内部业务决定是否创建多实例。

specified 启动模式，根据业务需要是否创建一个新的 UIAbility 实例。在 UIAbility 实例创建之前，会先进入 AbilityStage 的 onAcceptWant 回调，在 onAcceptWant 回调中为每一个 UIAbility 实例创建一个 Key，后续每次调用 startAbility()方法创建该类型的 UIAbility 实例都会询问使用哪个 Key 对应的 UIAbility 实例来响应 startAbility()请求。

specified 启动模式的开发使用的步骤如下所示。

1．在 module.json5 文件中的“launchType”字段配置为“specified”。

```
{
  "module": {
    ...
    "abilities": [
      {
        "launchType": "specified",
        ...
      }
    ]
  }
}
```

2．在调用 startAbility()方法的 want 参数中，增加一个自定义参数来区别 UIAbility 实例，例如增加一个“instanceKey”自定义参数。

```
// 在启动指定实例模式的 UIAbility 时，给每一个 UIAbility 实例配置一个独立的 Key 标识
function getInstance() {
  ...
}
let context:common.UIAbilityContext = ...; // context 为调用方 UIAbility 的 UIAbilityContext
let want: Want = {
  deviceId: '', // deviceId 为空表示本设备
  bundleName: 'com.example.myapplication',
  abilityName: 'SpecifiedAbility',
  moduleName: 'specified', // moduleName 非必选
  parameters: { // 自定义信息
    instanceKey: getInstance(),
```

```
    },
}
context.startAbility(want).then(() => {
    ...
}).catch((err: BusinessError) => {
    ...
})
```

3．在被拉起方 UIAbility 对应的 AbilityStage 的 onAcceptWant 生命周期回调中，解析传入的 want 参数，获取“instanceKey”自定义参数。根据业务需要返回一个该 UIAbility 实例的字符串 Key 标识。如果之前启动过此 Key 标识的 UIAbility，则会将之前的 UIAbility 拉回前台并获焦，而不创建新的实例，否则创建新的实例并启动。

```
onAcceptWant(want: want): string {
    // 在被启动方的 AbilityStage 中，针对启动模式为 specified 的 UIAbility 返回一个 UIAbility 实例对应的一个 Key 值
    // 当前示例指的是 device Module 的 EntryAbility
    if (want.abilityName === 'MainAbility') {
        return `DeviceModule_MainAbilityInstance_${want.parameters.instanceKey}`;
    }
    return '';
}
```

例如在文档应用中，可以对不同的文档实例内容绑定不同的 Key 值。当每次新建文档的时候，可以传入不同的新 Key 值（如可以将文件的路径作为一个 Key 标识），此时 AbilityStage 中启动 UIAbility 时都会创建一个新的 UIAbility 实例；当新建的文档保存之后，回到桌面，或者新打开一个已保存的文档，回到桌面，此时再次打开该已保存的文档，此时 AbilityStage 中再次启动该 UIAbility 时，打开的仍然是之前原来已保存的文档界面。

5 第五章 电商严选小项目

5.1 Column&Row 组件的使用

5.1.1 概述

一个丰富的页面需要很多组件组成，那么，我们如何才能让这些组件有条不紊地在页面上布局呢？这就需要借助容器组件来实现。

容器组件是一种比较特殊的组件，它可以包含其他的组件，而且按照一定的规律布局，帮助开发者生成精美的页面。容器组件除了放置基础组件外，也可以放置容器组件，通过多层布局的嵌套，可以布局出更丰富的页面。

ArkTS 为我们提供了丰富的容器组件来布局页面，本文将以构建登录页面为例，介绍 Column 和 Row 组件的属性与使用。

5.1.2 组件介绍

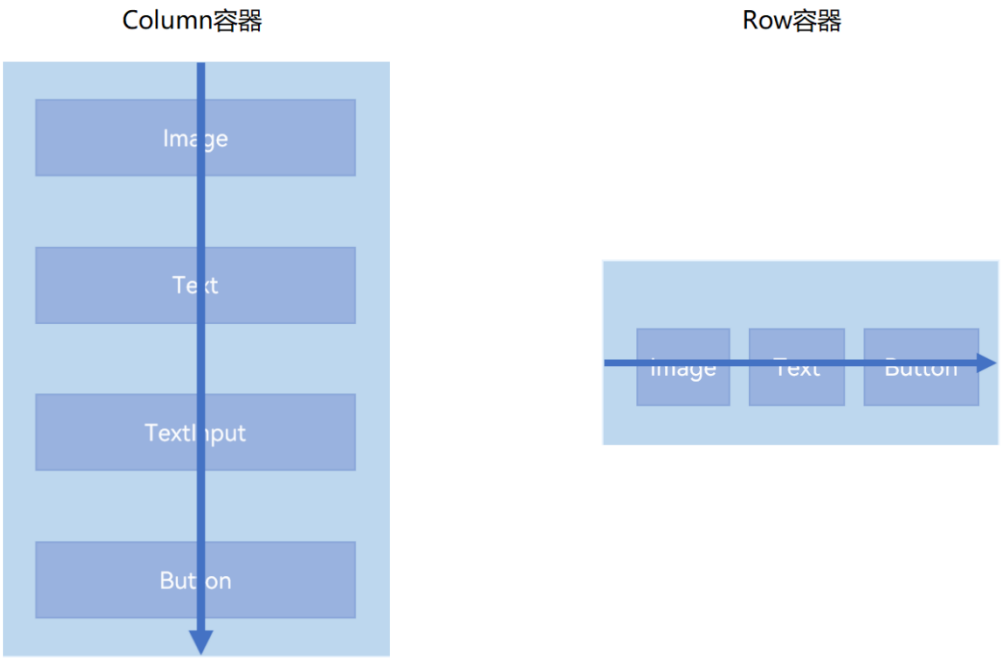
5.1.2.1 布局容器

线性布局容器表示按照垂直方向或者水平方向排列子组件的容器，ArkTS 提供了 Column 和 Row 容器来实现线性布局。

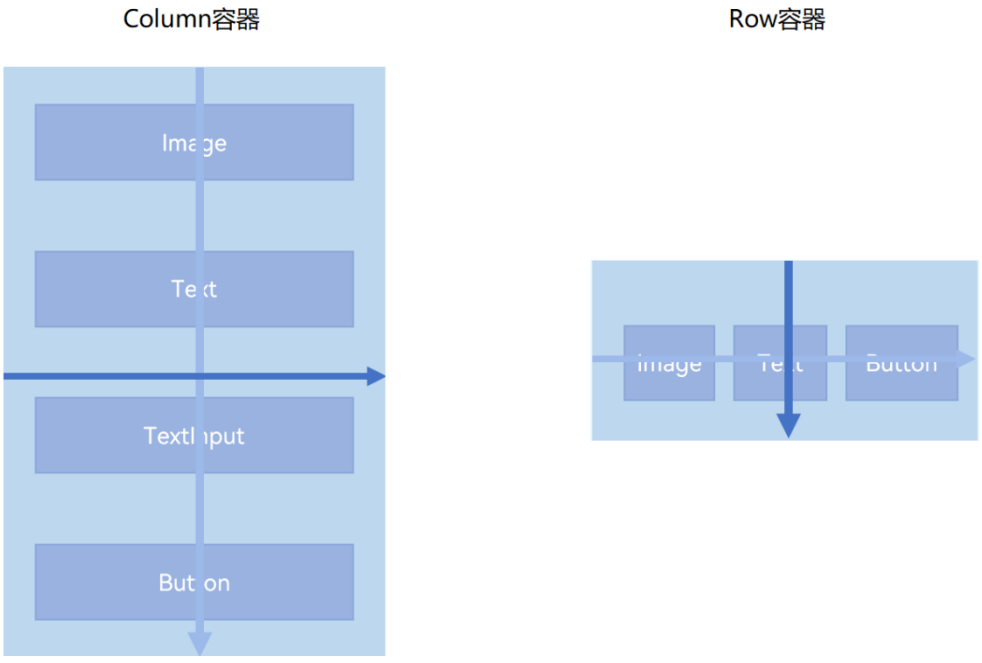
- Column 表示沿垂直方向布局的容器。
- Row 表示沿水平方向布局的容器。

在布局容器中，默认存在两根轴，分别是主轴和交叉轴，这两个轴始终是相互垂直的。不同的容器中主轴的方向不一样。

主轴：在 Column 容器中的子组件是按照从上到下的垂直方向布局的，其主轴的方向是垂直方向；在 Row 容器中的组件是按照从左到右的水平方向布局的，其主轴的方向是水平方向。下图为 Column 容器&Row 容器主轴。



交叉轴：与主轴垂直相交的轴线，如果主轴是垂直方向，则交叉轴就是水平方向；如果主轴是水平方向，则交叉轴是垂直方向。下图为 Column 容器&Row 容器交叉轴。



5.1.2.2 属性介绍

了解布局容器的主轴和交叉轴，主要是为了让大家更好地理解子组件在主轴和交叉轴的排列方式。接下来，我们将详细讲解 Column 和 Row 容器的两个属性 justifyContent 和 alignItems。

属性名称	描述
justifyContent	设置子组件在主轴方向上的对齐格式。
alignItems	设置子组件在交叉轴方向上的对齐格式。

1. 主轴方向的对齐（justifyContent）

子组件在主轴方向上的对齐使用 justifyContent 属性来设置，其参数类型是 FlexAlign。

Column 代码示例如下：

```
    @Entry
@Component
struct Demo {
  build() {
    Column(){
      Text("文本 1")
      Text("文本 2")
      Text("文本 3")
    }
    .height('100%')
    .width('100%')
    .justifyContent(FlexAlign.Start)
  }
}
```

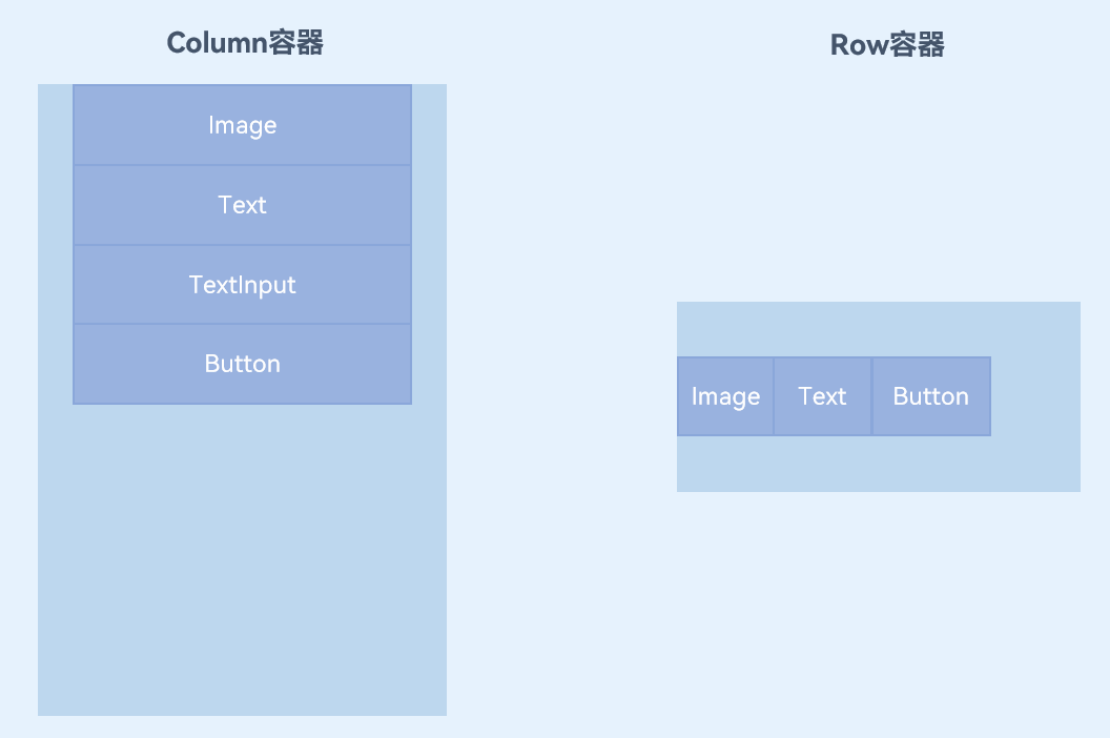
Row 代码示例如下：

```
    @Entry
@Component
struct Demo {
  build() {
    Row(){
      Text("文本 1")
      Text("文本 2")
      Text("文本 3")
    }
    .height('100%')
    .width('100%')
    .justifyContent(FlexAlign.SpaceEvenly)
  }
```

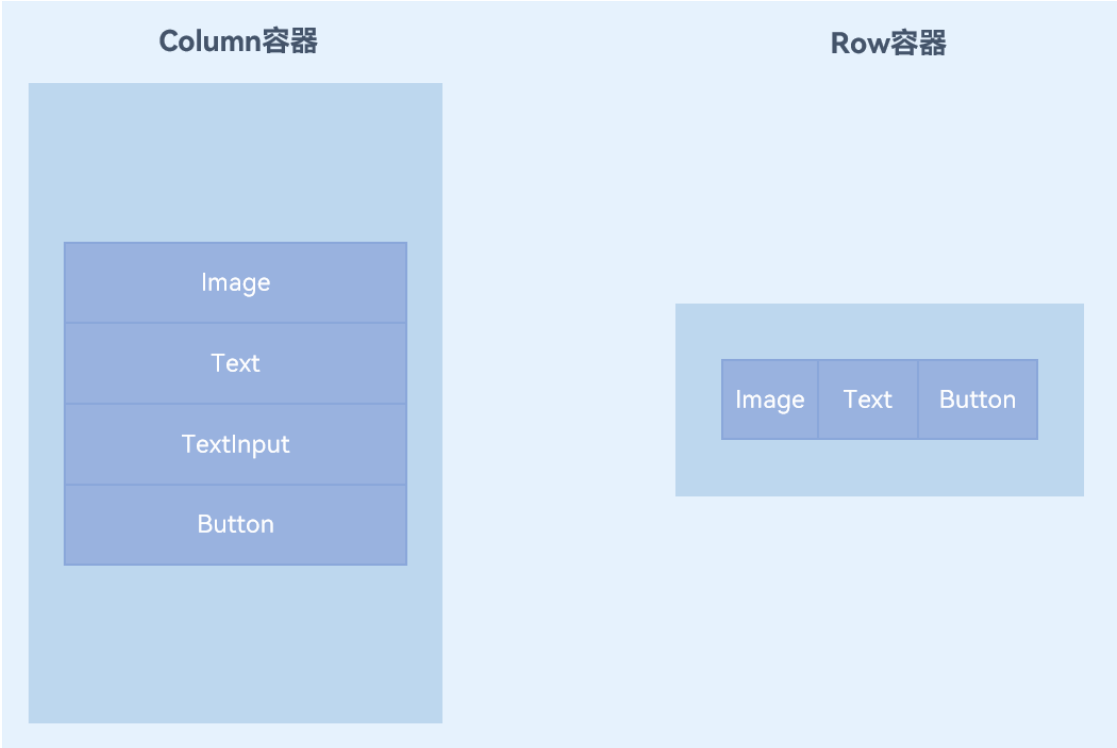
```
}
```

FlexAlign 定义了以下几种类型：

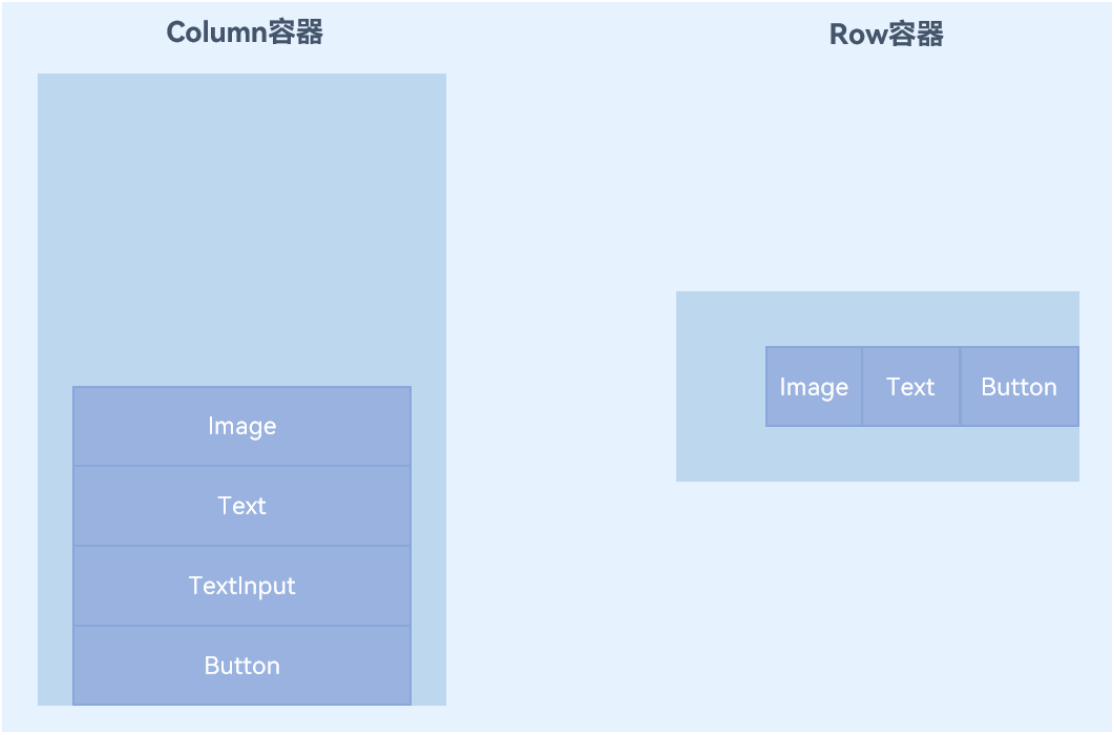
- Start：元素在主轴方向首端对齐，第一个元素与行首对齐，同时后续的元素与前一个对齐。



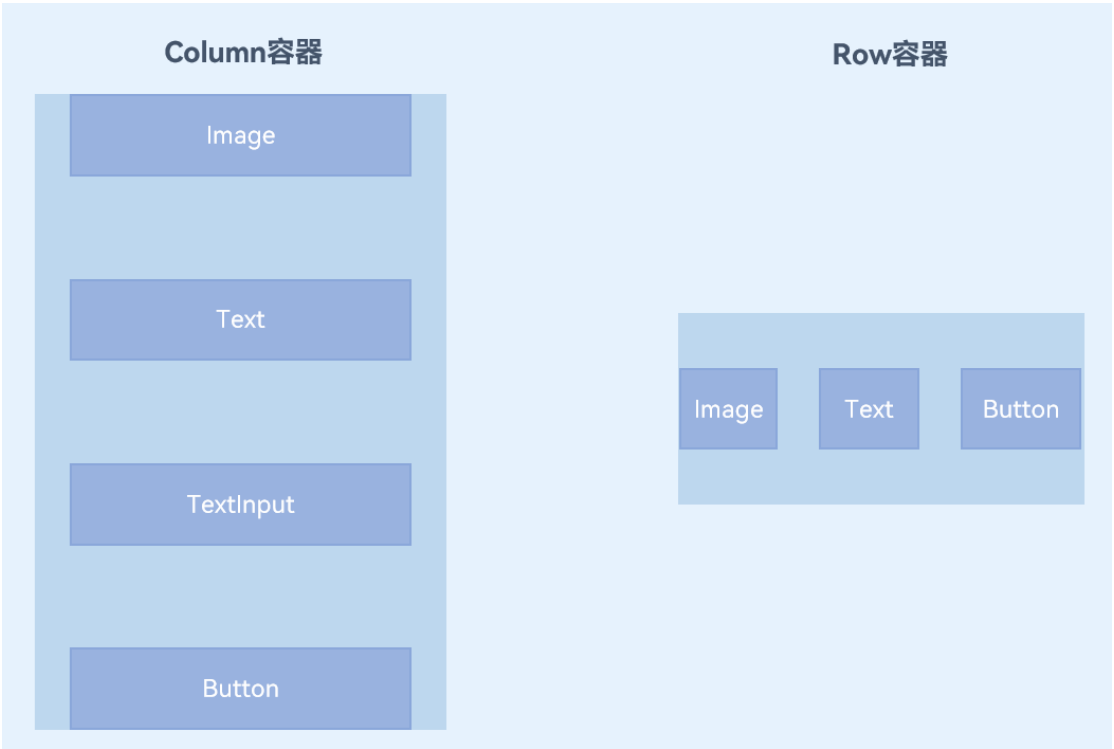
- Center：元素在主轴方向中心对齐，第一个元素与行首的距离以及最后一个元素与行尾距离相同。



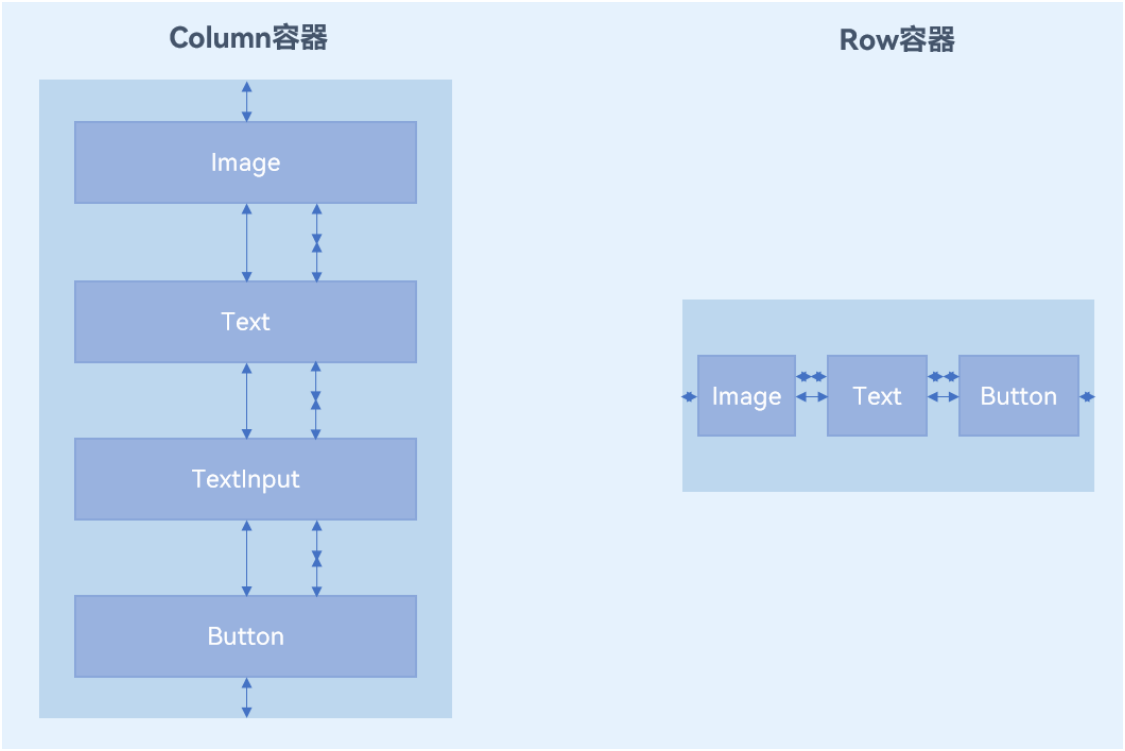
- End：元素在主轴方向尾部对齐，最后一个元素与行尾对齐，其他元素与后一个对齐。



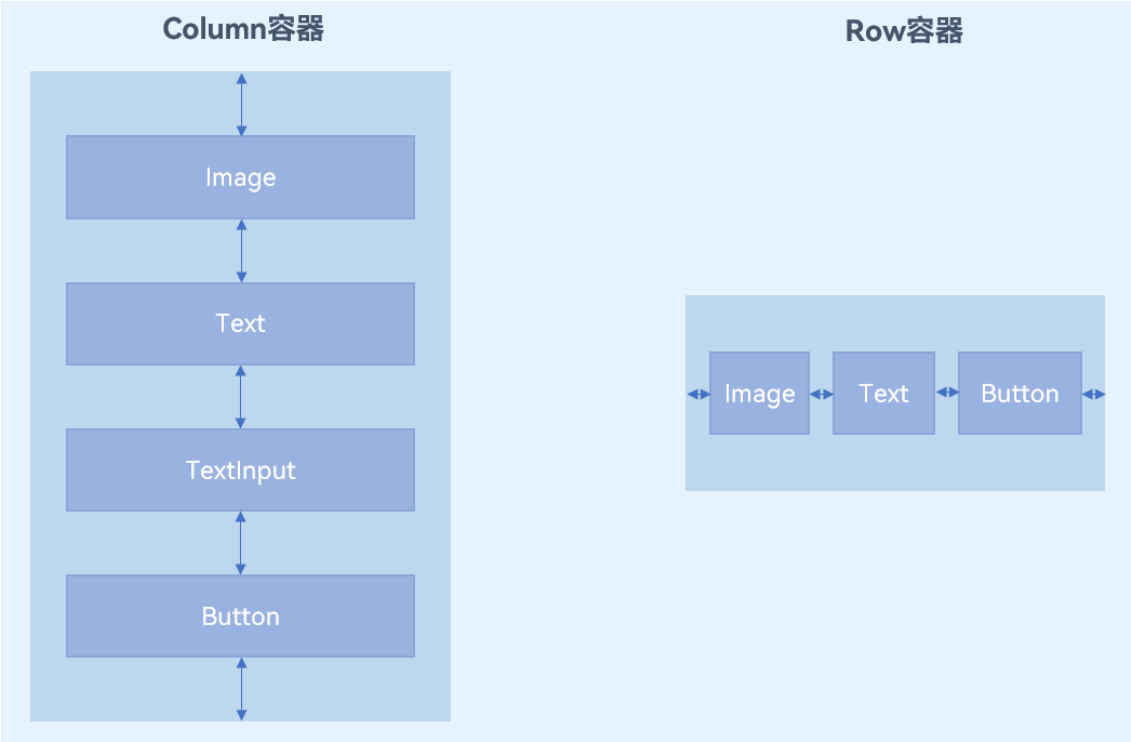
- SpaceBetween：元素在主轴方向均匀分配弹性元素，相邻元素之间距离相同。 第一个元素与行首对齐，最后一个元素与行尾对齐。



- SpaceAround：元素在主轴方向均匀分配弹性元素，相邻元素之间距离相同。 第一个元素到行首的距离和最后一个元素到行尾的距离是相邻元素之间距离的一半。



- SpaceEvenly：元素在主轴方向等间距布局，无论是相邻元素还是边界元素到容器的间距都一样。



2. 交叉轴方向的对齐（alignItems）

子组件在交叉轴方向上的对齐方式使用 alignItems 属性来设置。

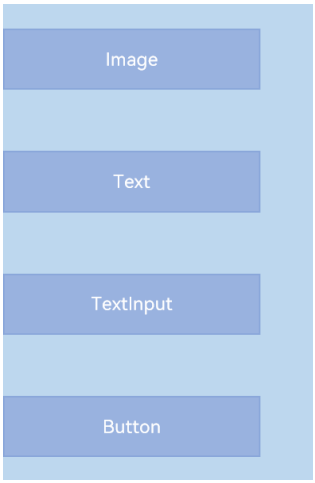
Column 容器的主轴是垂直方向，交叉轴是水平方向，其参数类型为 HorizontalAlign（水平对齐）。

Column 代码示例如下：

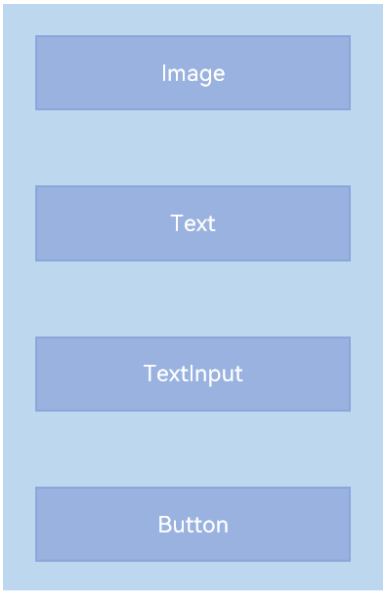
```
    @Entry
@Component
struct Demo {
  build() {
    Column(){
      Text("文本 1")
      Text("文本 2")
      Text("文本 3")
    }
    .height('100%')
    .width('100%')
    .justifyContent(FlexAlign.Start)
    .alignItems(HorizontalAlign.End)
  }
}
```

HorizontalAlign 定义了以下几种类型：

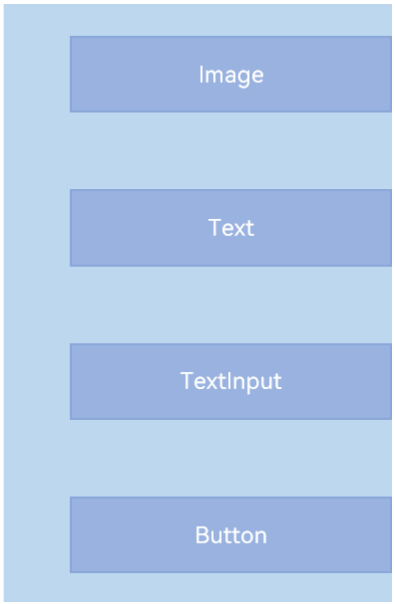
- Start：设置子组件在水平方向上按照起始端对齐。



- Center（默认值）：设置子组件在水平方向上居中对齐。



- End：设置子组件在水平方向上按照末端对齐。



Row 容器的主轴是水平方向，交叉轴是垂直方向，其参数类型为 VerticalAlign（垂直对齐）。

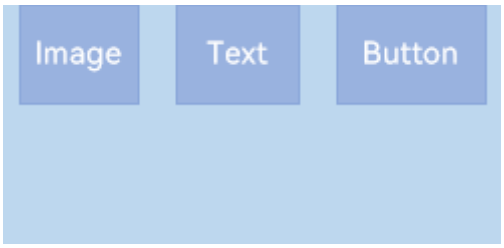
Row 代码示例如下：

```
    @Entry
@Component
struct Demo {
  build() {
    Row(){
      Text("文本 1")
      Text("文本 2")
      Text("文本 3")
    }
    .height('100%')
    .width('100%')
```

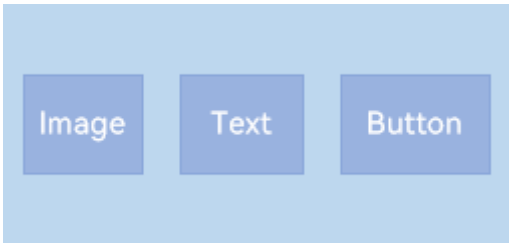
```
    .justifyContent(FlexAlign.SpaceEvenly)
    .alignItems(VerticalAlign.Bottom)
  }
}
```

VerticalAlign 定义了以下几种类型：

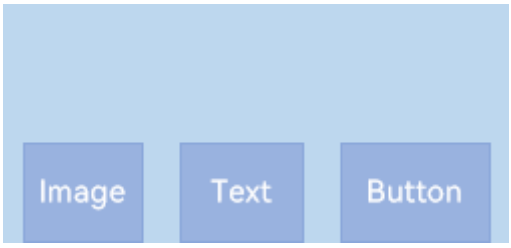
- Top：设置子组件在垂直方向上居顶部对齐。



- Center（默认值）：设置子组件在竖直方向上居中对齐。



- Bottom：设置子组件在竖直方向上居底部对齐。



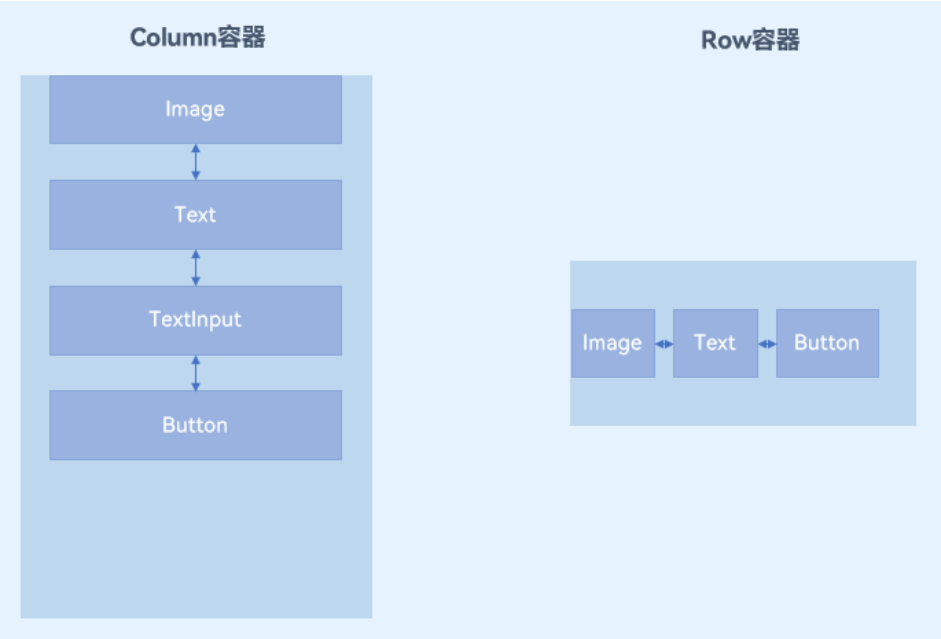
5.1.2.3 接口介绍

接下来，我们介绍 Column 和 Row 容器的接口。

容器组件	接口
Column	Column(value?:{space?: string | number})

Row	Row(value?:{space?: string | number})

Column 和 Row 容器的接口都有一个可选参数 space，表示子组件在主轴方向上的间距。效果如下：



Column 接口代码示例如下，随着 space 调节大小，可以看到各个文本间距变化。

```
    @Entry
@Component
struct Demo {
  build() {
    Column({space:100}){
      Text("文本 1")
      Text("文本 2")
      Text("文本 3")
    }
    .height('100%')
    .width('100%')
    .justifyContent(FlexAlign.Start)
    .alignItems(HorizontalAlign.End)
  }
}
```

Row 接口代码示例如下，随着 space 调节大小，可以看到各个文本间距变化。

```
    @Entry
@Component
struct ImgDemo {
  build() {
    Row({space:30}){
      Text("文本 1")
```

```
      Text("文本 2")
      Text("文本 3")
    }
    .height('100%')
    .width('100%')
    .justifyContent(FlexAlign.Start)
    .alignItems(VerticalAlign.Bottom)
  }
}
```

5.1.3 案例-登录页面实现

我们来具体讲解如何高效的使用 Column 和 Row 容器组件来构建如下登录页面。当我们从设计同学那拿到一个页面设计图时，我们需要对页面进行拆解，先确定页面的布局，再分析页面上的内容分别使用哪些组件来实现。

我们仔细分析这个登录页面。在静态布局中，组件整体是从上到下布局的，因此构建该页面可以使用 Column 来构建。在此基础上，我们可以看到有部分内容在水平方向上由几个基础组件构成，例如页面中间的短信验证码登录与忘记密码以及页面最下方的其他方式登录，那么构建这些内容的时候，可以在 Column 组件中嵌套 Row 组件，继而在 Row 组件中实现水平方向的布局。

根据上述页面拆解，在 Column 容器里，依次是 Image、Text、TextInput、Button 等基础组件，还有两组组件是使用 Row 容器组件来实现的。按照如下步骤来实现以上页面。

1. **准备图片**

将“logo.png”、“login_method1.png”、“login_method2.png”、“login_method3.png”图片放入项目 resource 下的 media 目录中。

2. 准备资源文件

在 resource/element 目录下准备 color.json、float.json、string.json 三个文件，同时确保 string.json 文件中一些配置项在 en_US、zh_CN 目录下的 element 中也存在。具体如下：

```
    … …
    {
  "name": "login_page",
  "value": "登录界面"
},{
  "name": "login_more",
  "value": "登录账号以使用更多服务"
},{
  "name": "account",
  "value": "请输入账号"
},{
  "name": "password",
  "value": "请输入密码"
},{
  "name": "message_login",
  "value": "短信验证码登录"
},{
  "name": "forgot_password",
  "value": "忘记密码"
},{
  "name": "login",
  "value": "登录"
},{
  "name": "register_account",
  "value": "注册账号"
},{
  "name": "other_login_method",
  "value": "其他方式登录"
}

    … …
```

3. 编写登录页面代码

```
    // TextInput 组件的自定义样式扩展
@Extend(TextInput)
function inputStyle() {
  .placeholderColor($r('app.color.placeholder_color')) // 占位符颜色
```

```
  .height($r('app.float.login_input_height')) // 输入框高度
  .fontSize($r('app.float.big_text_size')) // 字体大小
  .backgroundColor($r('app.color.background')) // 背景颜色
  .width(CommonConstants.FULL_PARENT) // 宽度为父组件的 100%
  .padding({ left: CommonConstants.INPUT_PADDING_LEFT }) // 左侧填充
  .margin({ top: $r('app.float.input_margin_top') }) // 上方边距
}

// Line 组件的自定义样式扩展
@Extend(Line)
function lineStyle() {
  .width(CommonConstants.FULL_PARENT) // 宽度为父组件的 100%
  .height($r('app.float.line_height')) // 高度
  .backgroundColor($r('app.color.line_color')) // 背景颜色
}

// Text 组件的蓝色文本样式
@Extend(Text)
function blueTextStyle() {
  .fontColor($r('app.color.login_blue_text_color')) // 字体颜色
  .fontSize($r('app.float.small_text_size')) // 字体大小
  .fontWeight(FontWeight.Medium) // 字体粗细
}

/**
 * 登录页面组件
 */
import CommonConstants from './CommonConstants';
import prompt from '@ohos.promptAction';

@Entry
@Component
struct LoginPage {
  @State account: string = ''; // 账号状态变量
  @State password: string = ''; // 密码状态变量
  @State isShowProgress: boolean = false; // 显示进度指示器的状态变量
  private timeOutId: number = -1; // 用于控制超时的变量

  // 构建图片按钮的函数
  @Builder
  imageButton(src: Resource) {
    Button({ type: ButtonType.Circle, stateEffect: true }) {
      Image(src)
    }
    .height($r('app.float.other_login_image_size')) // 图片按钮高度
    .width($r('app.float.other_login_image_size')) // 图片按钮宽度
    .backgroundColor($r('app.color.background')) // 背景颜色
  }
```

```
  // 页面消失时的处理函数
  aboutToDisappear() {
    clearTimeout(this.timeOutId); // 清除超时计时器
    this.timeOutId = -1;
  }

  // 构建页面布局的函数
  build() {
    Column() {
      Image($r('app.media.logo')) // Logo 图片
        .width($r('app.float.logo_image_size')) // Logo 宽度
        .height($r('app.float.logo_image_size')) // Logo 高度
        .margin({ top: $r('app.float.logo_margin_top'), bottom: $r('app.float.logo_margin_bottom') }) // Logo 边距
      Text($r('app.string.login_page')) // 登录页面标题
        .fontSize($r('app.float.page_title_text_size')) // 标题字体大小
        .fontWeight(FontWeight.Medium) // 标题字体粗细
        .fontColor($r('app.color.title_text_color')) // 标题字体颜色
      Text($r('app.string.login_more')) // “了解更多”文本
        .fontSize($r('app.float.normal_text_size')) // 字体大小
        .fontColor($r('app.color.login_more_text_color')) // 字体颜色
        .margin({ bottom: $r('app.float.login_more_margin_bottom'), top: $r('app.float.login_more_margin_top') }) // 边距

      // 账号输入框
      TextInput({ placeholder: $r('app.string.account') })
        .maxLength(CommonConstants.INPUT_ACCOUNT_LENGTH) // 最大长度
        .type(InputType.Number) // 输入类型为数字
        .inputStyle() // 应用自定义样式
        .onChange((value: string) => {
          this.account = value; // 更新账号状态
        })
      Line().lineStyle() // 应用自定义 Line 样式

      // 密码输入框
      TextInput({ placeholder: $r('app.string.password') })
        .maxLength(CommonConstants.INPUT_PASSWORD_LENGTH) // 最大长度
        .type(InputType.Password) // 输入类型为密码
        .inputStyle() // 应用自定义样式
        .onChange((value: string) => {
          this.password = value; // 更新密码状态
        })
      Line().lineStyle() // 应用自定义 Line 样式

      // 登录与忘记密码文本
      Row() {
        Text($r('app.string.message_login')).blueTextStyle() // “消息登录”文本
```

```
      Text($r('app.string.forgot_password')).blueTextStyle() //  “忘记密码”  文本
    }
    .justifyContent(FlexAlign.SpaceBetween) // 两端对齐
    .width(CommonConstants.FULL_PARENT) // 宽度为父组件的 100%
    .margin({ top: $r('app.float.forgot_margin_top') }) // 上方边距

    // 登录按钮
    Button($r('app.string.login'), { type: ButtonType.Capsule })
      .width(CommonConstants.BUTTON_WIDTH) // 按钮宽度
      .height($r('app.float.login_button_height')) // 按钮高度
      .fontSize($r('app.float.normal_text_size')) // 字体大小
      .fontWeight(FontWeight.Medium) // 字体粗细
      .backgroundColor($r('app.color.login_button_color')) // 背景颜色
      .margin({ top: $r('app.float.login_button_margin_top'), bottom:
$r('app.float.login_button_margin_bottom') }) // 边距
      .onClick(() => {
        prompt.showToast({
          message: "登录成功" // 登录成功提示
        })
      })

    // 注册账号文本
    Text($r('app.string.register_account'))
      .fontColor($r('app.color.login_blue_text_color')) // 字体颜色
      .fontSize($r('app.float.normal_text_size')) // 字体大小
      .fontWeight(FontWeight.Medium) // 字体粗细

    // 进度指示器
    if (this.isShowProgress) {
      LoadingProgress() // 加载进度组件
        .color($r('app.color.loading_color')) // 颜色
        .width($r('app.float.login_progress_size')) // 宽度
        .height($r('app.float.login_progress_size')) // 高度
        .margin({ top: $r('app.float.login_progress_margin_top') }) // 上方边距
    }

    Blank() // 空白组件

    // 其他登录方式文本
    Text($r('app.string.other_login_method'))
      .fontColor($r('app.color.other_login_text_color')) // 字体颜色
      .fontSize($r('app.float.little_text_size')) // 字体大小
      .fontWeight(FontWeight.Medium) // 字体粗细
      .margin({ top: $r('app.float.other_login_margin_top'), bottom:
$r('app.float.other_login_margin_bottom') }) // 边距

    // 其他登录方式的图片按钮
    Row({ space: CommonConstants.LOGIN_METHODS_SPACE }) {
```

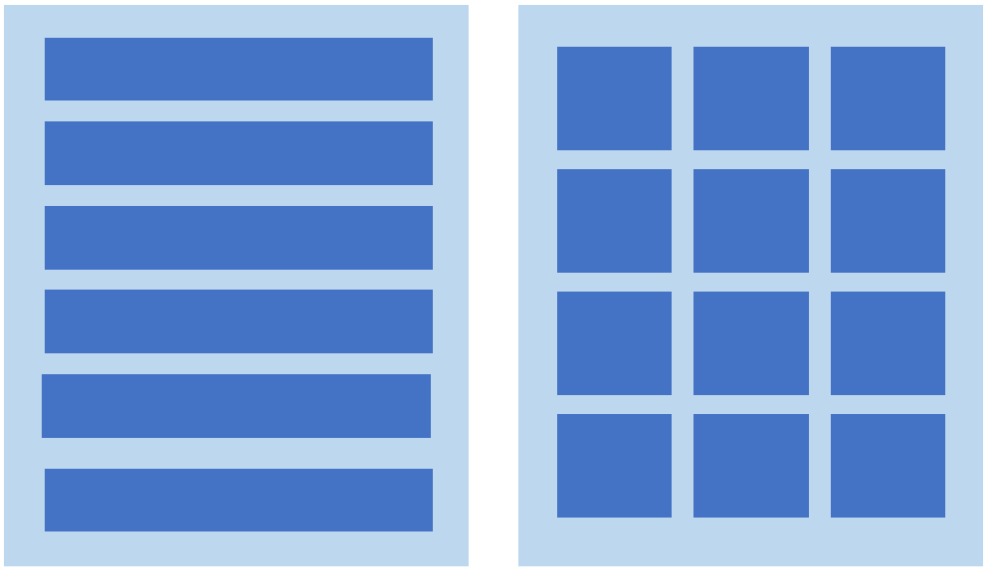
```
        this.imageButton($r('app.media.login_method1')) // 登录方式 1
        this.imageButton($r('app.media.login_method2')) // 登录方式 2
        this.imageButton($r('app.media.login_method3')) // 登录方式 3
      }
    }
    .backgroundColor($r('app.color.background')) // 背景颜色
    .height(CommonConstants.FULL_PARENT) // 高度为父组件的 100%
    .width(CommonConstants.FULL_PARENT) // 宽度为父组件的 100%
    .padding({
      left: $r('app.float.page_padding_hor'), // 左右填充
      right: $r('app.float.page_padding_hor'),
      bottom: $r('app.float.login_page_padding_bottom') // 底部填充
    })
  }
}
```

以上登录页面预览图如下：

5.2 List 组件和 Grid 组件

在我们常用的手机应用中，经常会见到一些数据列表，如设置页面、通讯录、商品列表等。下图中两个页面都包含列表，“首页”页面中包含两个网格布局，“商城”页面中包含一个商品列表。

上图中的列表中都包含一系列相同宽度的列表项，连续、多行呈现同类数据，例如图片和文本。常见的列表有线性列表（List 列表）和网格布局（Grid 列表）：



List列表　　Grid列表

为了帮助开发者构建包含列表的应用，ArkUI 提供了 List 组件和 Grid 组件，开发者使用 List 和 Grid 组件能够很轻松的完成一些列表页面。

5.2.1 List 组件

List 是很常用的滚动类容器组件，一般和子组件 ListItem 一起使用，List 列表中的每一个列表项对应一个 ListItem 组件。

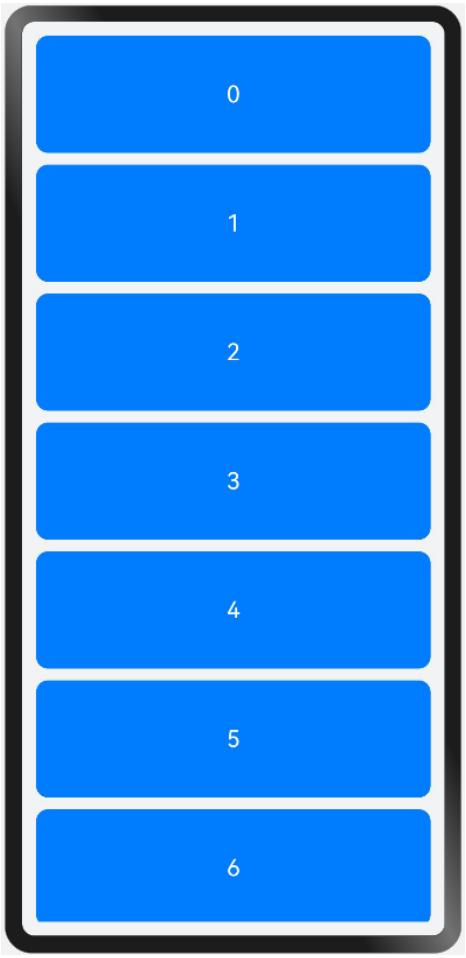
5.2.1.1 使用 ForEach 渲染列表

列表往往由多个列表项组成，所以我们需要在 List 组件中使用多个 ListItem 组件来构建列表，这就会导致代码的冗余。使用循环渲染（ForEach）遍历数组的方式构建列表，可以减少重复代码，示例代码如下：

```
    @Entry
@Component
struct ListDemo {
  private arr: number[] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

  build() {
    Column() {
      List({ space: 10 }) {
        ForEach(this.arr, (item: number) => {
          ListItem() {
            Text(`${item}`)
              .width('100%')
              .height(100)
              .fontSize(20)
              .fontColor(Color.White)
              .textAlign(TextAlign.Center)
              .borderRadius(10)
              .backgroundColor(0x007DFF)
```

```
        }
      }, item => item)
    }
  }
  .padding(12)
  .height('100%')
  .backgroundColor(0xF1F3F5)
 }
}
```

效果图如下：

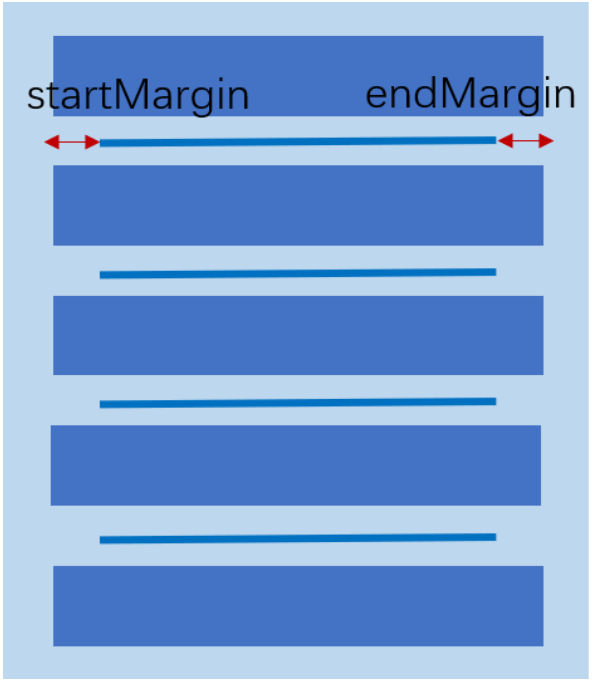


5.2.1.2 设置列表分割线

List 组件子组件 ListItem 之间默认是没有分割线的，部分场景子组件 ListItem 间需要设置分割线，这时候您可以使用 List 组件的 divider 属性。divider 属性包含四个参数：

- strokeWidth: 分割线的线宽。
- color: 分割线的颜色。
- startMargin：分割线距离列表侧边起始端的距离。

- endMargin: 分割线距离列表侧边结束端的距离。

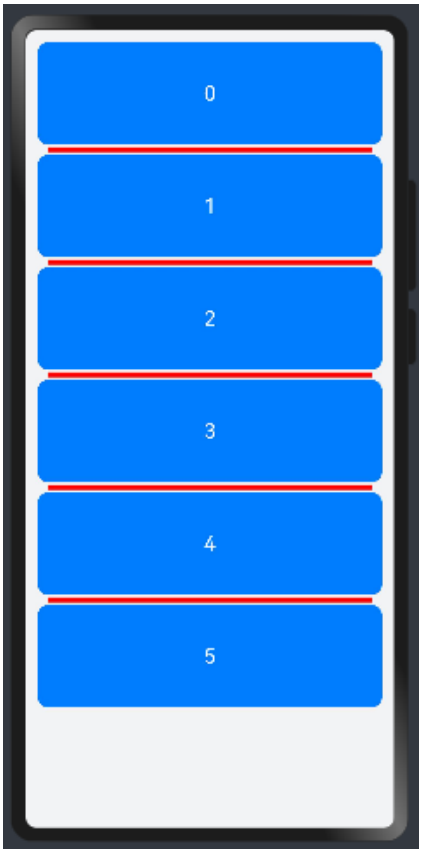


示例代码：

```
    @Entry
@Component
struct ListDemo {
  private arr: number[] = [0, 1, 2, 3, 4, 5]

  build() {
    Column() {
      List({ space: 10 }) {
        ForEach(this.arr, (item: number) => {
          ListItem() {
            Text(`${item}`)
              .width('100%')
              .height(100)
              .fontSize(20)
              .fontColor(Color.White)
              .textAlign(TextAlign.Center)
              .borderRadius(10)
              .backgroundColor(0x007DFF)
          }
        }, item => item)
      }
      .divider({strokeWidth:5,color:Color.Red,startMargin:10,endMargin:10})
    }
    .padding(12)
    .height('100%')
    .backgroundColor(0xF1F3F5)
  }
```

```
}
```

效果图如下：



5.2.1.3 List 列表滚动事件监听

List 组件提供了一系列事件方法用来监听列表的滚动，您可以根据需要，监听这些事件来做一些操作：

- onScroll：列表滑动时触发，返回值 scrollOffset 为滑动偏移量，scrollState 为当前滑动状态。
- onScrollIndex：列表滑动时触发，返回值分别为滑动起始位置索引值与滑动结束位置索引值。
- onReachStart：列表到达起始位置时触发。
- onReachEnd：列表到底末尾位置时触发。
- onScrollStop：列表滑动停止时触发。

使用示例代码如下：

```
    @Entry
@Component
struct ListDemo {
  private arr: number[] = [0, 1, 2, 3, 4, 5,6,7,8,9,10]

  build() {
```

```
  Column() {
    List({ space: 10 }) {
      ForEach(this.arr, (item) => {
        ListItem() {
          Text(`${item}`)
            .width('100%')
            .height(100)
            .fontSize(20)
            .fontColor(Color.White)
            .textAlign(TextAlign.Center)
            .borderRadius(10)
            .backgroundColor(0x007DFF)
        }
      }, item => item)
    }
    .onScrollIndex((firstIndex: number, lastIndex: number) => {
      console.info('滑动起始位置索引值' + firstIndex)
      console.info('滑动结束位置索引值' + lastIndex)
    })
    .onScroll((scrollOffset: number, scrollState: ScrollState) => {
      console.info('滑动偏移量' + scrollOffset)
      console.info('当前滑动状态' + scrollState)
    })
    .onReachStart(() => {
      console.info('列表起始位置到达')
    })
    .onReachEnd(() => {
      console.info('列表末尾位置到达')
    })
    .onScrollStop(() => {
      console.info('列表滑动停止')
    })
  }
  .padding(12)
  .height('100%')
  .backgroundColor(0xF1F3F5)
 }
}
```

以上代码编写完成后，可以使用鼠标上下拖动列表，在 log 标签下可以看到打印的不同结果。

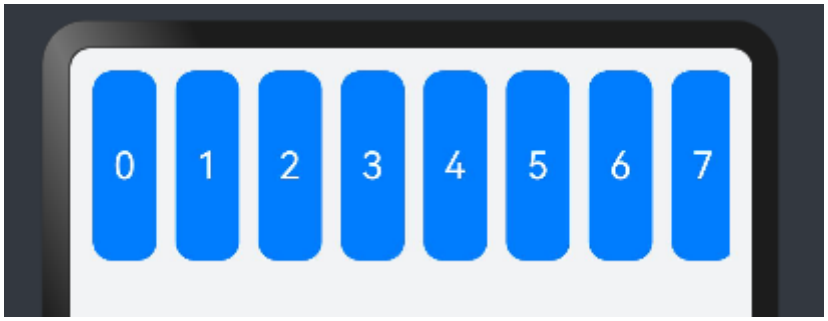
5.2.1.4 设置 List 排列方向

List 组件里面的列表项默认是按垂直方向排列的，如果您想让列表沿水平方向排列，您可以将 List 组件的 listDirection 属性设置为 Axis.Horizontal。

示例代码如下:

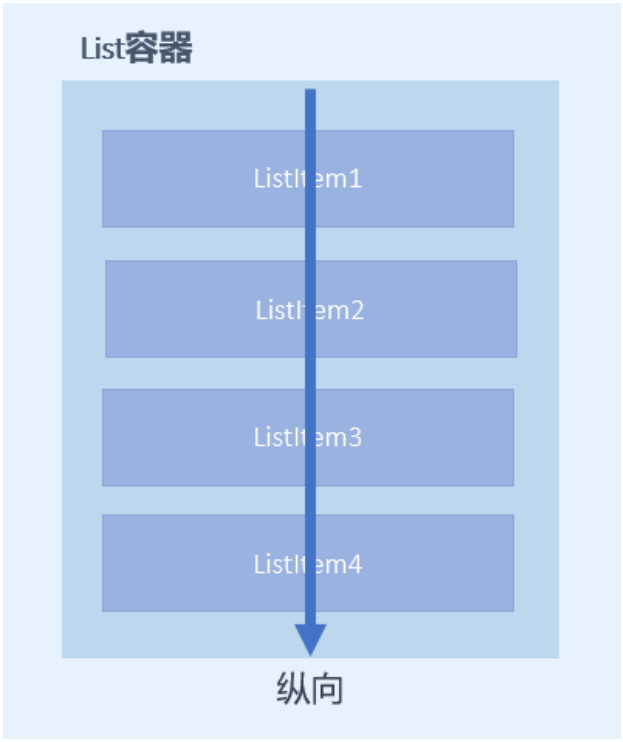
```
    @Entry
@Component
struct ListDemo {
  private arr: number[] = [0, 1, 2, 3, 4, 5,6,7,8,9,10]

  build() {
    Column() {
      List({ space: 10 }) {
        ForEach(this.arr, (item) => {
          ListItem() {
            Text(`${item}`)
              .width('10%')
              .height(100)
              .fontSize(20)
              .fontColor(Color.White)
              .textAlign(TextAlign.Center)
              .borderRadius(10)
              .backgroundColor(0x007DFF)
          }
        }, item => item)
      }
      .listDirection(Axis.Horizontal)//设置水平方向排布
      .onScrollIndex((firstIndex: number, lastIndex: number) => {
        console.info('滑动起始位置索引值' + firstIndex)
        console.info('滑动结束位置索引值' + lastIndex)
      })
      .onScroll((scrollOffset: number, scrollState: ScrollState) => {
        console.info('滑动偏移量' + scrollOffset)
        console.info('当前滑动状态' + scrollState)
      })
      .onReachStart(() => {
        console.info('列表起始位置到达')
      })
      .onReachEnd(() => {
        console.info('列表末尾位置到达')
      })
      .onScrollStop(() => {
        console.info('列表滑动停止')
      })
    }
    .padding(12)
    .height('100%')
    .backgroundColor(0xF1F3F5)
  }
}
```

效果图如下：

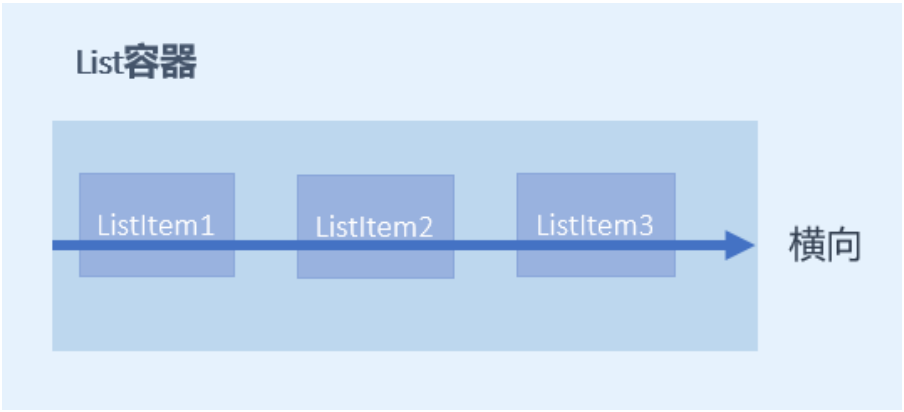


listDirection 参数类型是 Axis，定义了以下两种类型：

- Vertical（默认值）：子组件 ListItem 在 List 容器组件中呈纵向排列。



- Horizontal：子组件 ListItem 在 List 容器组件中呈横向排列。



5.2.2 Grid 组件

Grid 组件为网格容器，是一种网格列表，由“行”和“列”分割的单元格所组成，通过指定“项目”所在的单元格做出各种各样的布局。Grid 组件一般和子组件 GridItem 一起使用，Grid 列表中的每一个条目对应一个 GridItem 组件。

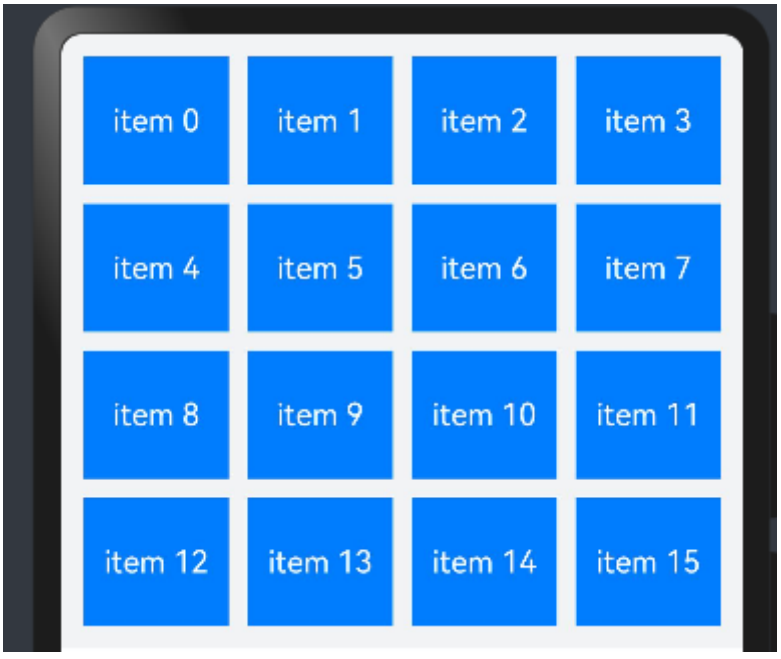
5.2.2.1 使用 Foreach 渲染网格布局

和 List 组件一样，Grid 组件也可以使用 ForEach 来渲染多个列表项 GridItem，我们通过下面的这段示例代码来介绍 Grid 组件的使用。

```
    @Entry // 表示入口组件
@Component // 表示这是一个组件
struct GridExample {
  // 定义一个长度为 16 的字符串数组，用于存放网格中的项
  private arr: string[] = new Array(16).fill('').map((_, index) => `item ${index}`);

  build() {
    Column() { // 创建一个列布局
      Grid() { // 创建一个网格布局
        // 遍历数组中的每个元素，为每个元素创建一个网格项
        ForEach(this.arr, (item: string) => {
          GridItem() { // 创建网格中的单个项
            Text(item) // 显示文本
              .fontSize(16) // 设置字体大小为 16
              .fontColor(Color.White) // 设置字体颜色为白色
              .backgroundColor(0x007DFF) // 设置背景颜色
              .width('100%') // 设置宽度为容器的 100%
              .height('100%') // 设置高度为容器的 100%
              .textAlign(TextAlign.Center) // 设置文本居中对齐
          }
```

```
      }, item => item)
    }
    .columnsTemplate('1fr 1fr 1fr 1fr') // 设置网格的列模板，4 列，每列比例相同
    .rowsTemplate('1fr 1fr 1fr 1fr') // 设置网格的行模板，4 行，每行比例相同
    .columnsGap(10) // 设置列之间的间隙为 10
    .rowsGap(10) // 设置行之间的间隙为 10
    .height(300) // 设置网格的高度为 300
  }
  .width('100%') // 设置列布局的宽度为 100%
  .padding(12) // 设置列布局的内边距为 12
  .backgroundColor(0xF1F3F5) // 设置列布局的背景颜色
 }
}
```

示例代码中创建了 16 个 GridItem 列表项。同时设置 columnsTemplate 的值为'1fr 1fr 1fr 1fr'，表示这个网格为 4 列，将 Grid 允许的宽分为 4 等分，每列占 1 份；rowsTemplate 的值为'1fr 1fr 1fr 1fr'，表示这个网格为 4 行，将 Grid 允许的高分为 4 等分，每行占 1 份。这样就构成了一个 4 行 4 列的网格列表，然后使用 columnsGap 设置列间距为 10vp，使用 rowsGap 设置行间距也为 10vp。示例代码效果图如下：



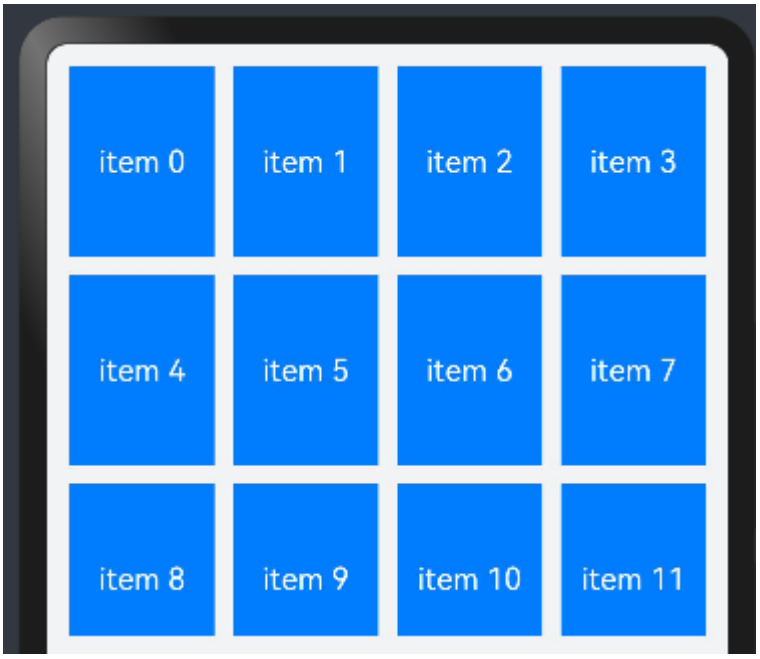
上面构建的网格布局使用了固定的行数和列数，所以构建出的网格是不可滚动的。然而有时候因为内容较多，我们通过滚动的方式来显示更多的内容，就需要一个可以滚动的网格布局。我们只需要设置 rowsTemplate 和 columnsTemplate 中的一个即可。

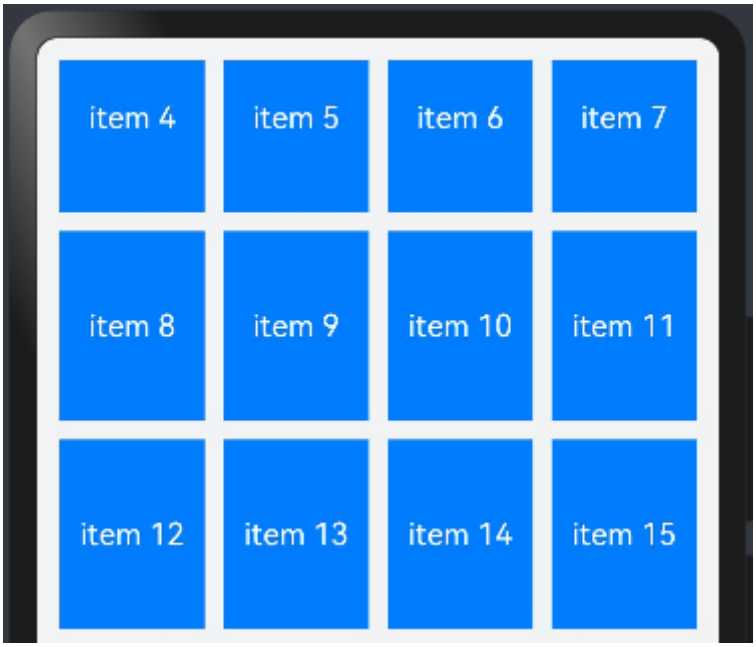
将示例代码中 GridItem 的高度设置为固定值，例如 100；仅设置 columnsTemplate 属性，不设置 rowsTemplate 属性，就可以实现 Grid 列表的滚动：

```
@Entry // 表示入口组件
```

```
@Component // 表示这是一个组件
struct GridExample {
  // 定义一个长度为 16 的字符串数组，用于存放网格中的项
  private arr: string[] = new Array(16).fill('').map((_, index) => `item ${index}`);

  build() {
    Column() { // 创建一个列布局
      Grid() { // 创建一个网格布局
        // 遍历数组中的每个元素，为每个元素创建一个网格项
        ForEach(this.arr, (item: string) => {
          GridItem() { // 创建网格中的单个项
            Text(item) // 显示文本
              .fontSize(16) // 设置字体大小为 16
              .fontColor(Color.White) // 设置字体颜色为白色
              .backgroundColor(0x007DFF) // 设置背景颜色
              .width('100%') // 设置宽度为容器的 100%
              .height('100') // 设置高度为 100
              .textAlign(TextAlign.Center) // 设置文本居中对齐
          }
        }, item => item)
      }
      .columnsTemplate('1fr 1fr 1fr 1fr') // 设置网格的列模板，4 列，每列比例相同
      .columnsGap(10) // 设置列之间的间隙为 10
      .rowsGap(10) // 设置行之间的间隙为 10
      .height(300) // 设置网格的高度为 300
    }
    .width('100%') // 设置列布局的宽度为 100%
    .padding(12) // 设置列布局的内边距为 12
    .backgroundColor(0xF1F3F5) // 设置列布局的背景颜色
  }
}
```

效果图如下：

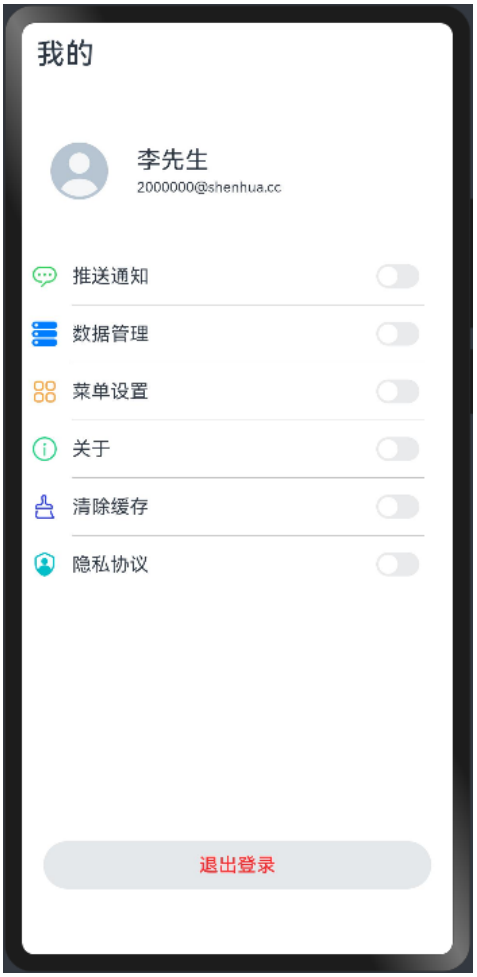




此外，Grid 像 List 一样也可以使用 onScrollIndex 来监听列表的滚动。

5.2.3 “我的”页面案例-List 组件实现

下面通过 List 组件来实现如下“我的”页面：



1. 准备图片和配置文件

这里需要在项目中 media 目录下放入对应使用到的图标和“string.json”配置文件。关于图片和文件详见资料。

2. 创建 ItemData 类

由于 “我的”页面中有多处图片和文字的组合，因此提取出 ItemData 类。

```
    /**
 * 列表项数据实体。
 */
export default class PageResource {
  /**
   * 列表项的文本。
   */
  title: Resource;
  /**
   * 列表项的图片。
   */
  img: Resource;
```

```
  /**
   * 列表项的其他资源。
   */
  others?: Resource;

  constructor(title: Resource, img: Resource, others?: Resource) {
    this.title = title;
    this.img = img;
    this.others = others;
  }
}
```

3. 创建 MainViewModel

以上页面由列表组成，这里准备 MainViewModel.ets 文件中对页面使用的资源进行定义，在 MainViewModel.ets 文件中定义数据。

```
    // 导入 ItemData 类
import ItemData from './ItemData';

/**
 * 绑定数据到组件并提供接口。
 */
export class MainViewModel {

  /**
   * 获取设置列表的数据。
   *
   * @return {Array<PageResource>} 返回设置列表数据数组。
   */
  getSettingListData(): Array<ItemData> {
    let settingListData: ItemData[] = [
      new ItemData($r('app.string.setting_list_news'), $r('app.media.news'),
$r("app.string.setting_toggle")),
      new ItemData($r('app.string.setting_list_data'), $r('app.media.data')),
      new ItemData($r('app.string.setting_list_menu'), $r('app.media.menu')),
      new ItemData($r('app.string.setting_list_about'), $r('app.media.about')),
      new ItemData($r('app.string.setting_list_storage'), $r('app.media.storage')),
      new ItemData($r('app.string.setting_list_privacy'), $r('app.media.privacy'))
    ];
    return settingListData;
  }

}

// 导出 MainViewModel 的一个新实例。
```

```
export default new MainViewModel();
```

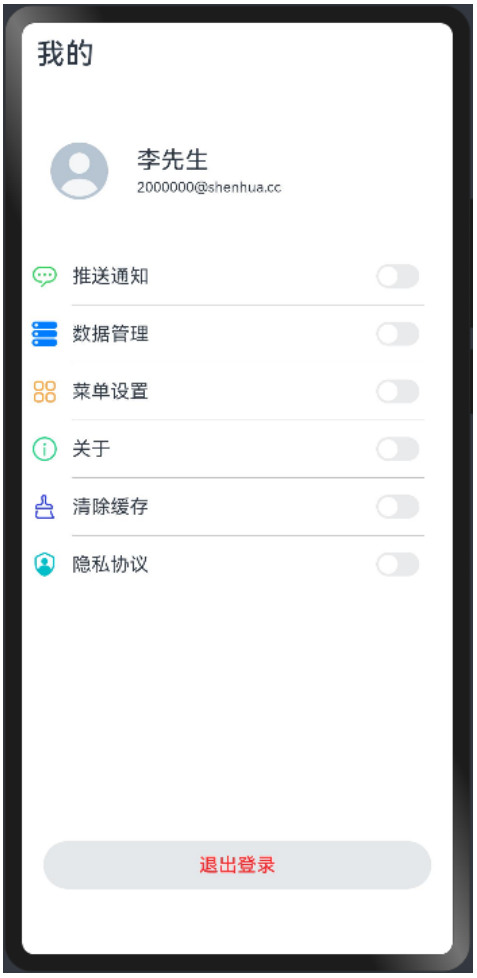
4. 编写页面代码

```
    // 导入公共常量和 ItemData 类，以及 mainViewModel 实例
import CommonConstants from './CommonConstants'
import ItemData from './ItemData'
import mainViewModel from './MainViewModel';

/**
 * 设置标签页的内容
 */
@Entry // 表示入口组件
@Component // 表示这是一个组件
export default struct Setting {
  // 构建设置单元格的函数，接收一个 ItemData 类型的 item 作为参数
  @Builder settingCell(item: ItemData) {
    Row() { // 创建一个行布局
      Row({ space: CommonConstants.COMMON_SPACE }) { // 创建一个有间隔的内嵌行布局
        Image(item.img) // 显示图像
          .width($r('app.float.setting_size')) // 设置图像的宽度
          .height($r('app.float.setting_size')) // 设置图像的高度
        Text(item.title) // 显示文本
          .fontSize($r('app.float.normal_text_size')) // 设置文本的字号
      }

      // 如果 item 的 others 属性为空，则显示一个图像，否则显示一个开关组件
      if (item.others === null) {
        Image($r('app.media.right_grey')) // 显示一个灰色的箭头图像
          .width($r('app.float.setting_jump_width')) // 设置图像宽度
          .height($r('app.float.setting_jump_height')) // 设置图像高度
      } else {
        Toggle({ type: ToggleType.Switch, isOn: false }) // 显示一个开关组件
      }
    }
    .justifyContent(FlexAlign.SpaceBetween) // 设置内容在主轴方向的对齐方式为两端对齐
    .width(CommonConstants.FULL_PARENT) // 设置宽度为父容器的 100%
    .padding({ // 设置内边距
      left: $r('app.float.setting_settingCell_left'),
      right: $r('app.float.setting_settingCell_right')
    })
  }

  // 组件的构建函数，定义了组件的布局和样式
  build() {
    Scroll() { // 创建一个可滚动的容器
      Column({ space: CommonConstants.COMMON_SPACE }) { // 创建一个有间隔的列布局
```

```
    Column(){ // 创建一个内嵌的列布局
      Text($r('app.string.mainPage_tabTitles_mine')) // 显示文本
        .fontWeight(FontWeight.Medium) // 设置字体权重
        .fontSize($r('app.float.page_title_text_size')) // 设置字号
        .margin({ top: $r('app.float.mainPage_tabTitles_margin') }) // 设置外边距
        .padding({ left: $r('app.float.mainPage_tabTitles_padding') }) // 设置内边距
    }
    .width(CommonConstants.FULL_PARENT) // 设置宽度为父容器的 100%
    .alignItems(HorizontalAlign.Start) // 设置子项在交叉轴上的对齐方式为开始对齐

    // 接下来是账户信息的布局
    Row() {
      Image($r('app.media.account')) // 显示账户图片
        .width($r('app.float.setting_account_size')) // 设置图片宽度
        .height($r('app.float.setting_account_size')) // 设置图片高度
      Column() {
        Text($r('app.string.setting_account_name')) // 显示账户名称
          .fontSize($r('app.float.setting_account_fontSize')) // 设置字号
        Text($r('app.string.setting_account_email')) // 显示账户邮箱
          .fontSize($r('app.float.little_text_size')) // 设置字号
          .margin({ top: $r('app.float.setting_name_margin') }) // 设置外边距
      }
      .alignItems(HorizontalAlign.Start) // 设置子项在交叉轴上的对齐方式为开始对齐
      .margin({ left: $r('app.float.setting_account_margin') }) // 设置外边距
    }
    .margin({ top: $r('app.float.setting_account_margin') }) // 设置外边距
    .alignItems(VerticalAlign.Center) // 设置子项在交叉轴上的对齐方式为居中对齐
    .width(CommonConstants.FULL_PARENT) // 设置宽度为父容器的 100%
    .height($r('app.float.setting_account_height')) // 设置高度
    .backgroundColor(Color.White) // 设置背景颜色为白色
    .padding({ left: $r('app.float.setting_account_padding') }) // 设置内边距
    .borderRadius($r('app.float.setting_account_borderRadius')) // 设置边角半径

    // 设置列表的布局
    List() {
      // 遍历设置列表数据，为每个数据项创建一个列表项
      ForEach(mainViewModel.getSettingListData(), (item: ItemData) => {
        ListItem() {
          this.settingCell(item) // 调用 settingCell 方法来构建每个列表项
        }
        .height($r('app.float.setting_list_height')) // 设置列表项高度
      }, (item:ItemData) => JSON.stringify(item)) // 使用 JSON.stringify 来生成列表项的 key
    }
    .backgroundColor(Color.White) // 设置背景颜色为白色
    .width(CommonConstants.FULL_PARENT) // 设置宽度为父容器的 100%
    .height(CommonConstants.SET_LIST_WIDTH) // 设置高度
    .divider({ // 设置列表项之间的分割线
      strokeWidth: $r('app.float.setting_list_strokeWidth'), // 设置分割线的宽度
```

```
        color: Color.Grey, // 设置分割线的颜色为灰色
        startMargin: $r('app.float.setting_list_startMargin'), // 设置分割线的起始边距
        endMargin: $r('app.float.setting_list_endMargin') // 设置分割线的结束边距
      })
      .borderRadius($r('app.float.setting_list_borderRadius')) // 设置边角半径
      .padding({ top: $r('app.float.setting_list_padding'), bottom: $r('app.float.setting_list_padding') }) // 设置内边距

      Blank() // 空白占位符

      // 退出按钮的布局
      Button($r('app.string.setting_button'), { type: ButtonType.Capsule }) // 创建一个按钮
        .width(CommonConstants.BUTTON_WIDTH) // 设置按钮宽度
        .height($r('app.float.login_button_height')) // 设置按钮高度
        .fontSize($r('app.float.normal_text_size')) // 设置字号
        .fontColor($r('app.color.setting_button_fontColor')) // 设置字体颜色
        .fontWeight(FontWeight.Medium) // 设置字体权重
        .backgroundColor($r('app.color.setting_button_backgroundColor')) // 设置背景颜色
        .margin({ bottom: $r('app.float.setting_button_bottom')}) // 设置外边距
    }
    .height(CommonConstants.FULL_PARENT) // 设置高度为父容器的 100%
  }
 }
}
```

预览结果如下：



5.2.4 “首页”页面案例-Grid 组件实现

下面通过 Grid 组件来实现如下“首页”页面：

1. 准备图片和配置文件

这里需要在项目中 media 目录下放入对应使用到的图标和“string.json”配置文件。关于图片和文件详见资料。

2. 创建 ItemData 类

由于“首页”和“我的”页面中有多处图片和文字的组合，因此提取出 ItemData 类。

```
    /**
 * 列表项数据实体。
 */
export default class PageResource {
  /**
   * 列表项的文本。
   */
  title: Resource;
  /**
   * 列表项的图片。
   */
  img: Resource;
  /**
   * 列表项的其他资源。
   */
```

```
  others?: Resource;

  constructor(title: Resource, img: Resource, others?: Resource) {
    this.title = title;
    this.img = img;
    this.others = others;
  }
}
```

3. 创建 MainViewModel

以上页面由列表组成，这里准备 MainViewModel.ets 文件中对页面使用的资源进行定义，在 MainViewModel.ets 文件中定义数据。

```
    // 导入 ItemData 类
import ItemData from './ItemData';

/**
 * 绑定数据到组件并提供接口。
 */
export class MainViewModel {

  /**
   * 获取设置列表的数据。
   *
   * @return {Array<PageResource>} 返回设置列表数据数组。
   */
  getSettingListData(): Array<ItemData> {
    let settingListData: ItemData[] = [
      new ItemData($r('app.string.setting_list_news'), $r('app.media.news'),
$r("app.string.setting_toggle")),
      new ItemData($r('app.string.setting_list_data'), $r('app.media.data')),
      new ItemData($r('app.string.setting_list_menu'), $r('app.media.menu')),
      new ItemData($r('app.string.setting_list_about'), $r('app.media.about')),
      new ItemData($r('app.string.setting_list_storage'), $r('app.media.storage')),
      new ItemData($r('app.string.setting_list_privacy'), $r('app.media.privacy'))
    ];
    return settingListData;
  }

}

// 导出 MainViewModel 的一个新实例。
export default new MainViewModel();
```

4. 编写页面代码

```
    // 导入公共常量、ItemData 类和 mainViewModel 实例
import CommonConstants from './CommonConstants';
import ItemData from './ItemData';
import mainViewModel from './MainViewModel';

/**
 * 首页标签的内容
 */
@Entry // 标记为入口组件
@Component // 标记为一个组件
export default struct Home {
  // 私有属性，轮播控制器的实例
  private swiperController: SwiperController = new SwiperController();

  // 构建函数，定义组件的布局和样式
  build() {
    Scroll() { // 创建一个可滚动的容器
      Column({ space: CommonConstants.COMMON_SPACE }) { // 创建一个带间隔的列布局
        Column() { // 创建一个子列布局
          Text($r('app.string.mainPage_tabTitles_home')) // 显示"首页"的文本
            .fontWeight(FontWeight.Medium) // 设置文本字重为中等
            .fontSize($r('app.float.page_title_text_size')) // 设置文本字号
            .margin({ top: $r('app.float.mainPage_tabTitles_margin') }) // 设置文本上边距
            .padding({ left: $r('app.float.mainPage_tabTitles_padding') }) // 设置文本左边距
        }
        .width(CommonConstants.FULL_PARENT) // 设置列宽度为父容器的 100%
        .alignItems(HorizontalAlign.Start) // 子元素的水平对齐方式为开始

        Swiper(this.swiperController) { // 创建轮播组件，控制器为 swiperController
          ForEach(mainViewModel.getSwiperImages(), (img: Resource) => { // 遍历轮播图图片资源
            Image(img) // 显示图片
              .borderRadius($r('app.float.home_swiper_borderRadius')) // 设置图片边角半径
          }, (img: Resource) => JSON.stringify(img.id)) // 使用图片 id 作为 key
        }
        .margin({ top: $r('app.float.home_swiper_margin') }) // 设置轮播组件的上边距
        .autoPlay(true) // 设置轮播组件自动播放

        Grid() { // 创建网格布局
          ForEach(mainViewModel.getFirstGridData(), (item: ItemData) => { // 遍历第一组网格数据
            GridItem() { // 创建网格项
              Column() { // 创建列布局
                Image(item.img) // 显示图片
                  .width($r('app.float.home_homeCell_size')) // 设置图片宽度
                  .height($r('app.float.home_homeCell_size')) // 设置图片高度
                Text(item.title) // 显示文本标题
                  .fontSize($r('app.float.little_text_size')) // 设置文本字号
```

```
            .margin({ top: $r('app.float.home_homeCell_margin') }) // 设置文本上边距
          }
        }
      }, (item: ItemData) => JSON.stringify(item)) // 使用 item 的字符串化作为 key
    }
    .columnsTemplate('1fr 1fr 1fr 1fr') // 设置网格的列模板，每列平分空间
    .rowsTemplate('1fr 1fr') // 设置网格的行模板，每行平分空间
    .columnsGap($r('app.float.home_grid_columnsGap')) // 设置列之间的间隔
    .rowsGap($r('app.float.home_grid_rowGap')) // 设置行之间的间隔
    .padding({ top: $r('app.float.home_grid_padding'), bottom: $r('app.float.home_grid_padding') }) // 设置网格内边距
    .height($r('app.float.home_grid_height')) // 设置网格高度
    .backgroundColor(Color.White) // 设置网格背景色为白色
    .borderRadius($r('app.float.home_grid_borderRadius')) // 设置网格边角半径

    Text($r('app.string.home_list')) // 显示"列表"的文本
      .fontSize($r('app.float.normal_text_size')) // 设置文本字号
      .fontWeight(FontWeight.Medium) // 设置文本字重
      .width(CommonConstants.FULL_PARENT) // 设置文本宽度为父容器的 100%
      .margin({ top: $r('app.float.home_text_margin') }) // 设置文本上边距

    Grid() { // 创建第二个网格布局
      ForEach(mainViewModel.getSecondGridData(), (secondItem: ItemData) => { // 遍历第二组网格数据
        GridItem() { // 创建网格项
          Column() { // 创建列布局
            Text(secondItem.title) // 显示标题文本
              .fontSize($r('app.float.normal_text_size')) // 设置文本字号
              .fontWeight(FontWeight.Medium) // 设置文本字重
            Text(secondItem.others) // 显示其他文本
              .margin({ top: $r('app.float.home_list_margin') }) // 设置文本上边距
              .fontSize($r('app.float.little_text_size')) // 设置文本字号
              .fontColor($r('app.color.home_grid_fontColor')) // 设置文本颜色
          }
          .alignItems(HorizontalAlign.Start) // 子元素的水平对齐方式为开始
        }
        .padding({ top: $r('app.float.home_list_padding'), left: $r('app.float.home_list_padding') }) // 设置网格项内边距
        .borderRadius($r('app.float.home_backgroundImage_borderRadius')) // 设置网格项边角半径
        .align(Alignment.TopStart) // 设置网格项对齐方式为顶部开始
        .backgroundImage(secondItem.img) // 设置网格项背景图像
        .backgroundImageSize(ImageSize.Cover) // 设置背景图像尺寸模式为覆盖
        .width(CommonConstants.FULL_PARENT) // 设置网格项宽度为父容器的 100%
        .height(CommonConstants.FULL_PARENT) // 设置网格项高度为父容器的 100%
      }, (secondItem: ItemData) => JSON.stringify(secondItem)) // 使用 item 的字符串化作为 key
    }
    .width(CommonConstants.FULL_PARENT) // 设置网格宽度为父容器的 100%
```

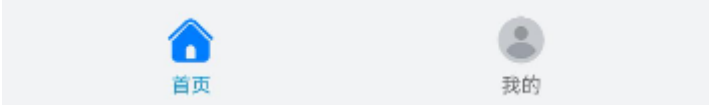
```
      .height($r('app.float.home_secondGrid_height')) // 设置网格高度
      .columnsTemplate('1fr 1fr') // 设置网格的列模板，每列平分空间
      .rowsTemplate('1fr 1fr') // 设置网格的行模板，每行平分空间
      .columnsGap($r('app.float.home_grid_columnsGap')) // 设置列之间的间隔
      .rowsGap($r('app.float.home_grid_rowGap')) // 设置行之间的间隔
      .margin({ bottom: $r('app.float.setting_button_bottom') }) // 设置网格底部外边距
    }
  }
  .height(CommonConstants.FULL_PARENT) // 设置滚动视图的高度为父容器的 100%
 }
}
```

预览结果如下：

5.3 页面切换

在我们常用的应用中，经常会有视图内容切换的场景，来展示更加丰富的内容。比如下面这个页面，点击底部的页签的选项，可以实现“首页”和“我的”两个内容视图的切换。

ArkUI 开发框架提供了一种页签容器组件 Tabs，开发者通过 Tabs 组件可以很容易的实现内容视图的切换。页签容器 Tabs 的形式多种多样，不同的页面设计页签不一样，可以把页签设置在底部、顶部或者侧边。



TabContent 的 tabBar 属性除了支持 string 类型，还支持使用@Builder 装饰器修饰的函数。我们可以使用@Builder 装饰器，构造一个生成自定义 TabBar 样式的函数，实现上面的底部页签效果，示例代码如下：

```
    @Entry
@Component
struct TabsExample {
  @State currentIndex: number = 0;
  private tabsController: TabsController = new TabsController();

  @Builder TabBuilder(title: string, targetIndex: number, selectedImg: Resource, normalImg: Resource) {
    Column() {
      Image(this.currentIndex === targetIndex ? selectedImg : normalImg)
        .size({ width: 25, height: 25 })
      Text(title)
        .fontColor(this.currentIndex === targetIndex ? '#1698CE' : '#6B6B6B')
    }
```

```
    .width('100%')
    .height(50)
    .justifyContent(FlexAlign.Center)
    .onClick(() => {
      this.currentIndex = targetIndex;
      this.tabsController.changeIndex(this.currentIndex);
    })
  }

  build() {
    Tabs({ barPosition: BarPosition.End, controller: this.tabsController }) {
      TabContent() {
        Column().width('100%').height('100%').backgroundColor('#00CB87')
      }
      .tabBar(this.TabBuilder('首 页', 0, $r('app.media.home_selected'),
$r('app.media.home_normal')))

      TabContent() {
        Column().width('100%').height('100%').backgroundColor('#007DFF')
      }
      .tabBar(this.TabBuilder('我的', 1, $r('app.media.mine_selected'), $r('app.media.mine_normal')))
    }
    .barWidth('100%')
    .barHeight(50)
    .onChange((index: number) => {
      this.currentIndex = index;
    })
  }
}
```

效果如下：

5.3.1 案例-实现“我的”和“首页”切换

下面将前几小节实现的“我的”和“首页”页面通过 Tabs 实现切换。步骤如下：

1. 将两个页面中的 Entry 入口去掉

一个项目中只能有一个入口，所以将“我的”和“首页”页面中的 @Entry

注解去掉。

2. 通过 Tabs 组件实现切换

创建 MainPage ets 文件，在该 ets 文件中通过 Tabs 完成两个页面切换。

```
import CommonConstants from './CommonConstants';
import Home from './Home';
import Setting from './Setting';
/**
 * 主页
 */
@Entry
@Component
struct MainPage {
  @State currentIndex: number = CommonConstants.HOME_TAB_INDEX;
  private tabsController: TabsController = new TabsController();

  @Builder TabBuilder(title: string, index: number, selectedImg: Resource, normalImg: Resource)
{
    Column() {
      Image(this.currentIndex === index ? selectedImg : normalImg)
        .width($r('app.float.mainPage_baseTab_size'))
        .height($r('app.float.mainPage_baseTab_size'))
      Text(title)
        .margin({ top: $r('app.float.mainPage_baseTab_top') })
        .fontSize($r('app.float.main_tab_fontSize'))
        .fontColor(this.currentIndex === index ? $r('app.color.mainPage_selected') :
$r('app.color.mainPage_normal'))
    }
    .justifyContent(FlexAlign.Center)
    .height($r('app.float.mainPage_barHeight'))
    .width(CommonConstants.FULL_PARENT)
    .onClick(() => {
      this.currentIndex = index;
      this.tabsController.changeIndex(this.currentIndex);
    })
  }

  build() {
    Tabs({
      barPosition: BarPosition.End,
      controller: this.tabsController
    }) {
      TabContent() {
        Home()
      }
      .padding({ left: $r('app.float.mainPage_padding'), right: $r('app.float.mainPage_padding') })
      .backgroundColor($r('app.color.mainPage_backgroundColor'))
```

```
      .tabBar(this.TabBuilder(CommonConstants.HOME_TITLE,
CommonConstants.HOME_TAB_INDEX,
        $r('app.media.home_selected'), $r('app.media.home_normal')))

      TabContent() {
        Setting()
      }
      .padding({ left: $r('app.float.mainPage_padding'), right: $r('app.float.mainPage_padding') })
      .backgroundColor($r('app.color.mainPage_backgroundColor'))
      .tabBar(this.TabBuilder(CommonConstants.MINE_TITLE,
CommonConstants.MINE_TAB_INDEX,
        $r('app.media.mine_selected'), $r('app.media.mine_normal')))
    }
    .width(CommonConstants.FULL_PARENT)
    .backgroundColor(Color.White)
    .barHeight($r('app.float.mainPage_barHeight'))
    .barMode(BarMode.Fixed)
    .onChange((index: number) => {
      this.currentIndex = index;
    })
  }
}
```

3. 在 login ets 文件中实现点击登录页面跳转功能

在 LoginPage.ets 文件中增加如下代码，并在登录按钮中实现 click 点击登录功能。

```
      ... ...

      login(): void {
  if (this.account === '' || this.password === '') {
    prompt.showToast({
      message: $r('app.string.input_empty_tips')
    })
  } else {
    this.isShowProgress = true;
    if (this.timeOutId === -1) {
      this.timeOutId = setTimeout(() => {
        this.isShowProgress = false;
        this.timeOutId = -1;
        router.replaceUrl({ url: 'pages/components/MainPage' });
      }, CommonConstants.LOGIN_DELAY_TIME);
    }
  }
}

      ... ...
```

完整代码如下：

```
    // TextInput 组件的自定义样式扩展
@Extend(TextInput)
function inputStyle() {
  .placeholderColor($r('app.color.placeholder_color')) // 占位符颜色
  .height($r('app.float.login_input_height')) // 输入框高度
  .fontSize($r('app.float.big_text_size')) // 字体大小
  .backgroundColor($r('app.color.background')) // 背景颜色
  .width(CommonConstants.FULL_PARENT) // 宽度为父组件的 100%
  .padding({ left: CommonConstants.INPUT_PADDING_LEFT }) // 左侧填充
  .margin({ top: $r('app.float.input_margin_top') }) // 上方边距
}

// Line 组件的自定义样式扩展
@Extend(Line)
function lineStyle() {
  .width(CommonConstants.FULL_PARENT) // 宽度为父组件的 100%
  .height($r('app.float.line_height')) // 高度
  .backgroundColor($r('app.color.line_color')) // 背景颜色
}

// Text 组件的蓝色文本样式
@Extend(Text)
function blueTextStyle() {
  .fontColor($r('app.color.login_blue_text_color')) // 字体颜色
  .fontSize($r('app.float.small_text_size')) // 字体大小
  .fontWeight(FontWeight.Medium) // 字体粗细
}

/**
 * 登录页面组件
 */
import CommonConstants from './CommonConstants';
import prompt from '@ohos.promptAction';
import router from '@ohos.router';

@Entry
@Component
struct LoginPage {
  @State account: string = ''; // 账号状态变量
  @State password: string = ''; // 密码状态变量
  @State isShowProgress: boolean = false; // 显示进度指示器的状态变量
  private timeOutId: number = -1; // 用于控制超时的变量

  // 构建图片按钮的函数
  @Builder
  imageButton(src: Resource) {
```

```
    Button({ type: ButtonType.Circle, stateEffect: true }) {
      Image(src)
    }
    .height($r('app.float.other_login_image_size')) // 图片按钮高度
    .width($r('app.float.other_login_image_size')) // 图片按钮宽度
    .backgroundColor($r('app.color.background')) // 背景颜色
  }


  login(): void {
    if (this.account === '' || this.password === '') {
      prompt.showToast({
        message: $r('app.string.input_empty_tips')
      })
    } else {
      this.isShowProgress = true;
      if (this.timeOutId === -1) {
        this.timeOutId = setTimeout(() => {
          this.isShowProgress = false;
          this.timeOutId = -1;
          router.replaceUrl({ url: 'pages/components/MainPage' });
        }, CommonConstants.LOGIN_DELAY_TIME);
      }
    }
  }

  // 页面消失时的处理函数
  aboutToDisappear() {
    clearTimeout(this.timeOutId); // 清除超时计时器
    this.timeOutId = -1;
  }

  // 构建页面布局的函数
  build() {
    Column() {
      Image($r('app.media.logo')) // Logo 图片
        .width($r('app.float.logo_image_size')) // Logo 宽度
        .height($r('app.float.logo_image_size')) // Logo 高度
        .margin({ top: $r('app.float.logo_margin_top'), bottom:
$r('app.float.logo_margin_bottom') }) // Logo 边距
      Text($r('app.string.login_page')) // 登录页面标题
        .fontSize($r('app.float.page_title_text_size')) // 标题字体大小
        .fontWeight(FontWeight.Medium) // 标题字体粗细
        .fontColor($r('app.color.title_text_color')) // 标题字体颜色
      Text($r('app.string.login_more')) // “了解更多”文本
        .fontSize($r('app.float.normal_text_size')) // 字体大小
        .fontColor($r('app.color.login_more_text_color')) // 字体颜色
        .margin({ bottom: $r('app.float.login_more_margin_bottom'), top:
```

```
$r('app.float.login_more_margin_top') }) // 边距

    // 账号输入框
    TextInput({ placeholder: $r('app.string.account') })
      .maxLength(CommonConstants.INPUT_ACCOUNT_LENGTH) // 最大长度
      .type(InputType.Number) // 输入类型为数字
      .inputStyle() // 应用自定义样式
      .onChange((value: string) => {
        this.account = value; // 更新账号状态
      })
    Line().lineStyle() // 应用自定义 Line 样式

    // 密码输入框
    TextInput({ placeholder: $r('app.string.password') })
      .maxLength(CommonConstants.INPUT_PASSWORD_LENGTH) // 最大长度
      .type(InputType.Password) // 输入类型为密码
      .inputStyle() // 应用自定义样式
      .onChange((value: string) => {
        this.password = value; // 更新密码状态
      })
    Line().lineStyle() // 应用自定义 Line 样式

    // 登录与忘记密码文本
    Row() {
      Text($r('app.string.message_login')).blueTextStyle() // “消息登录”文本
      Text($r('app.string.forgot_password')).blueTextStyle() // “忘记密码”文本
    }
    .justifyContent(FlexAlign.SpaceBetween) // 两端对齐
    .width(CommonConstants.FULL_PARENT) // 宽度为父组件的 100%
    .margin({ top: $r('app.float.forgot_margin_top') }) // 上方边距

    // 登录按钮
    Button($r('app.string.login'), { type: ButtonType.Capsule })
      .width(CommonConstants.BUTTON_WIDTH) // 按钮宽度
      .height($r('app.float.login_button_height')) // 按钮高度
      .fontSize($r('app.float.normal_text_size')) // 字体大小
      .fontWeight(FontWeight.Medium) // 字体粗细
      .backgroundColor($r('app.color.login_button_color')) // 背景颜色
      .margin({ top: $r('app.float.login_button_margin_top'), bottom:
$r('app.float.login_button_margin_bottom') }) // 边距
      .onClick(() => {
        this.login();
      })

    // 注册账号文本
    Text($r('app.string.register_account'))
      .fontColor($r('app.color.login_blue_text_color')) // 字体颜色
      .fontSize($r('app.float.normal_text_size')) // 字体大小
```

```
      .fontWeight(FontWeight.Medium) // 字体粗细

    // 进度指示器
    if (this.isShowProgress) {
      LoadingProgress() // 加载进度组件
        .color($r('app.color.loading_color')) // 颜色
        .width($r('app.float.login_progress_size')) // 宽度
        .height($r('app.float.login_progress_size')) // 高度
        .margin({ top: $r('app.float.login_progress_margin_top') }) // 上方边距
    }

    Blank() // 空白组件

    // 其他登录方式文本
    Text($r('app.string.other_login_method'))
      .fontColor($r('app.color.other_login_text_color')) // 字体颜色
      .fontSize($r('app.float.little_text_size')) // 字体大小
      .fontWeight(FontWeight.Medium) // 字体粗细
      .margin({ top: $r('app.float.other_login_margin_top'), bottom:
$r('app.float.other_login_margin_bottom') }) // 边距

    // 其他登录方式的图片按钮
    Row({ space: CommonConstants.LOGIN_METHODS_SPACE }) {
      this.imageButton($r('app.media.login_method1')) // 登录方式 1
      this.imageButton($r('app.media.login_method2')) // 登录方式 2
      this.imageButton($r('app.media.login_method3')) // 登录方式 3
    }
  }
  .backgroundColor($r('app.color.background')) // 背景颜色
  .height(CommonConstants.FULL_PARENT) // 高度为父组件的 100%
  .width(CommonConstants.FULL_PARENT) // 宽度为父组件的 100%
  .padding({
    left: $r('app.float.page_padding_hor'), // 左右填充
    right: $r('app.float.page_padding_hor'),
    bottom: $r('app.float.login_page_padding_bottom') // 底部填充
  })
 }
}
```

4. 添加 router 路由可转发页面

在 resource/base/profile/main_pages.json 中添加路由页面：

```
    {
  "src": [
    "pages/index",
    "pages/components/LoginPage",
    "pages/components/MainPage"
```

```
]
```

通过以上步骤，最终实现页面登录后，切换“我的”和“首页”页面功能。