**Harmony OS-4.0应用开发**

**讲师：子木**

# 第一章 Harmony OS概述

## 系统定义

### 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](https://developer.harmonyos.com/cn/docs/documentation/doc-guides/harmonyos-features-0000000000011907" \l "section3784192251716)。**

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](https://developer.harmonyos.com/cn/docs/documentation/doc-guides/harmonyos-features-0000000000011907" \l "section35241557442)。**

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。

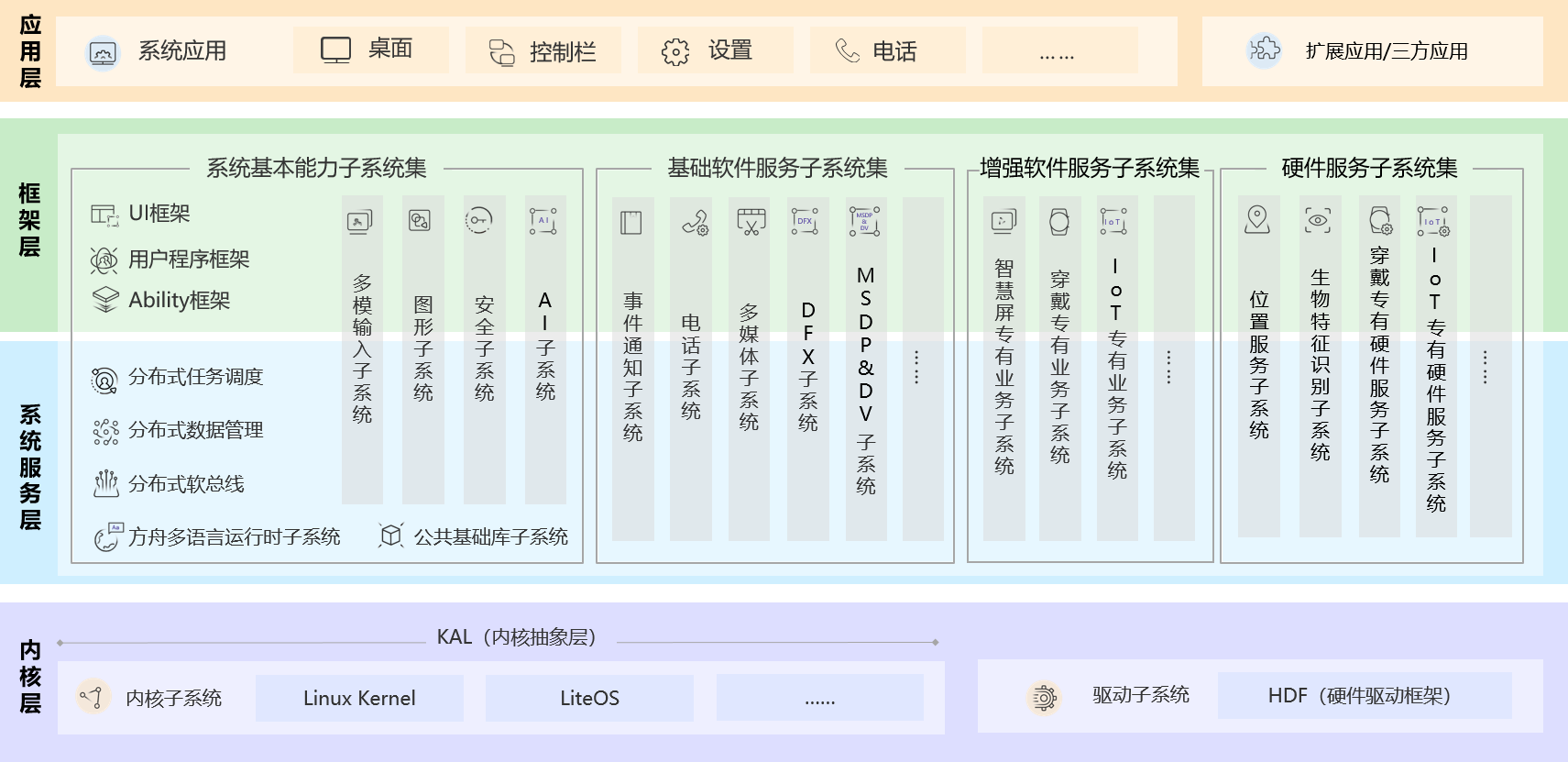
* **一套操作系统可以满足不同能力的设备需求，实现[统一OS，弹性部署](https://developer.harmonyos.com/cn/docs/documentation/doc-guides/harmonyos-features-0000000000011907" \l "section1169115112119)。**

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。

### 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。



#### 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）](https://developer.harmonyos.com/cn/docs/documentation/doc-guides/glossary-0000000000029587" \l "ZH-CN_TOPIC_0000001114162884__li1544183516475" \t "https://developer.harmonyos.com/cn/docs/documentation/doc-guides/_blank)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。

#### 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](https://developer.harmonyos.com/cn/docs/documentation/doc-guides/glossary-0000000000029587" \l "ZH-CN_TOPIC_0000001114162884__li1113671654618" \t "https://developer.harmonyos.com/cn/docs/documentation/doc-guides/_blank)&[DV](https://developer.harmonyos.com/cn/docs/documentation/doc-guides/glossary-0000000000029587" \l "ZH-CN_TOPIC_0000001114162884__li13399361415" \t "https://developer.harmonyos.com/cn/docs/documentation/doc-guides/_blank)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。

#### 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。

#### 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）](https://developer.harmonyos.com/cn/docs/documentation/doc-guides/glossary-0000000000029587" \l "ZH-CN_TOPIC_0000001114162884__li102311923104712" \t "https://developer.harmonyos.com/cn/docs/documentation/doc-guides/_blank)或[PA（Particle Ability）](https://developer.harmonyos.com/cn/docs/documentation/doc-guides/glossary-0000000000029587" \l "ZH-CN_TOPIC_0000001114162884__li11872193812460" \t "https://developer.harmonyos.com/cn/docs/documentation/doc-guides/_blank)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。

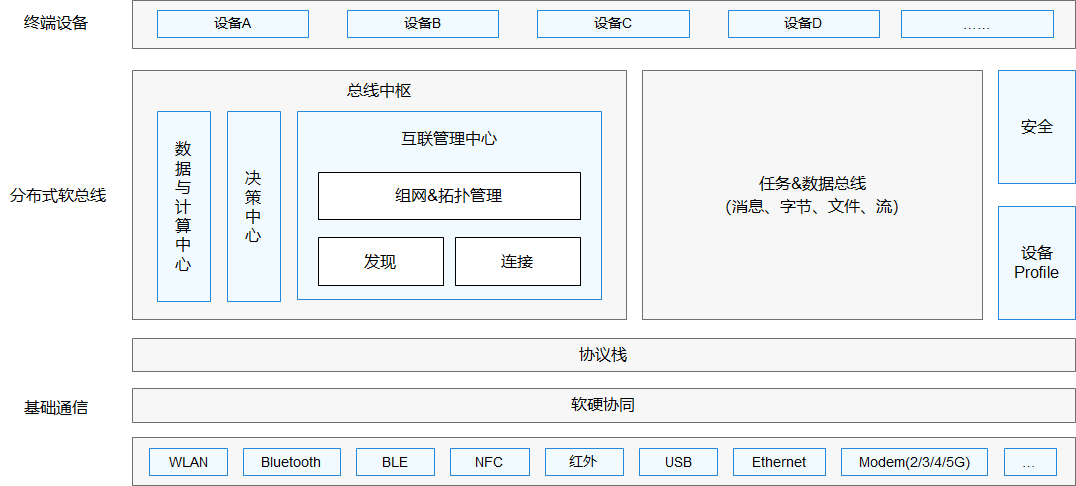
## 技术特性

### [硬件互助，资源共享](https://developer.harmonyos.com/cn/docs/" \l "section3784192251716" \o "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。

#### 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。

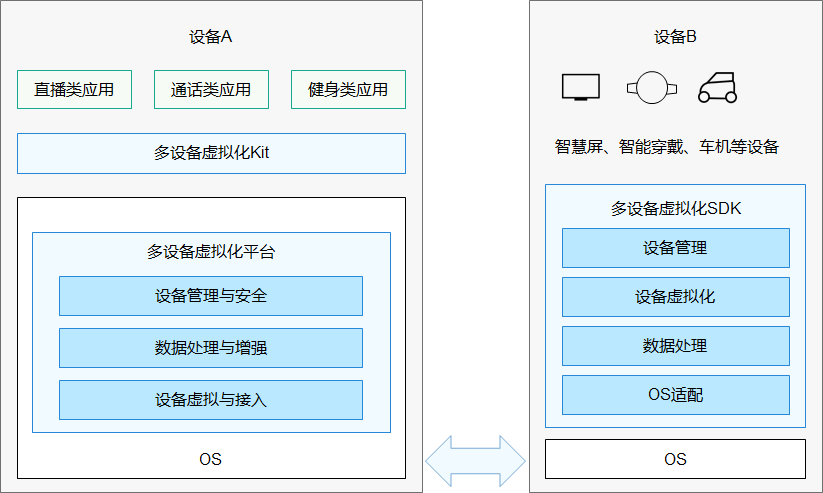


典型应用场景举例：

* 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。

#### 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](https://developer.harmonyos.com/cn/docs/documentation/doc-guides/glossary-0000000000029587" \l "ZH-CN_TOPIC_0000001114162884__li1133517458578" \t "https://developer.harmonyos.com/cn/docs/documentation/doc-guides/_blank)。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。

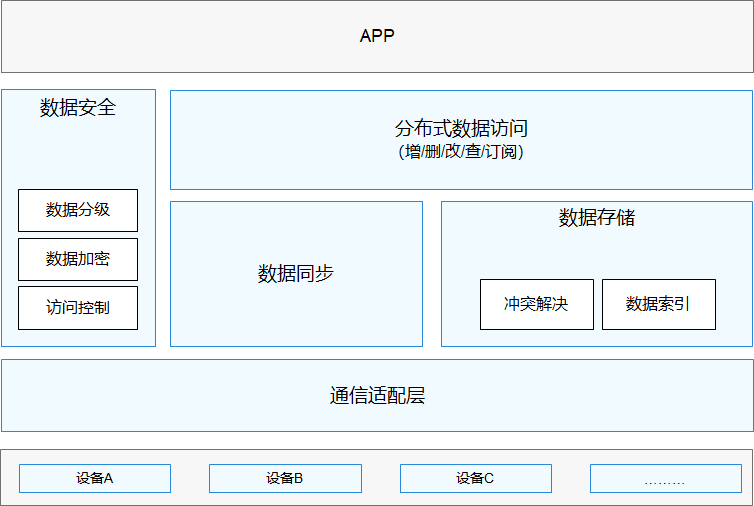


典型应用场景举例：

* 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。

#### 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。



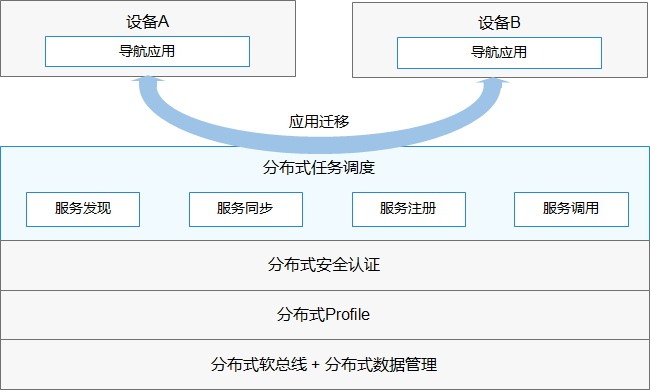
典型应用场景举例：

* 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。

#### 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。

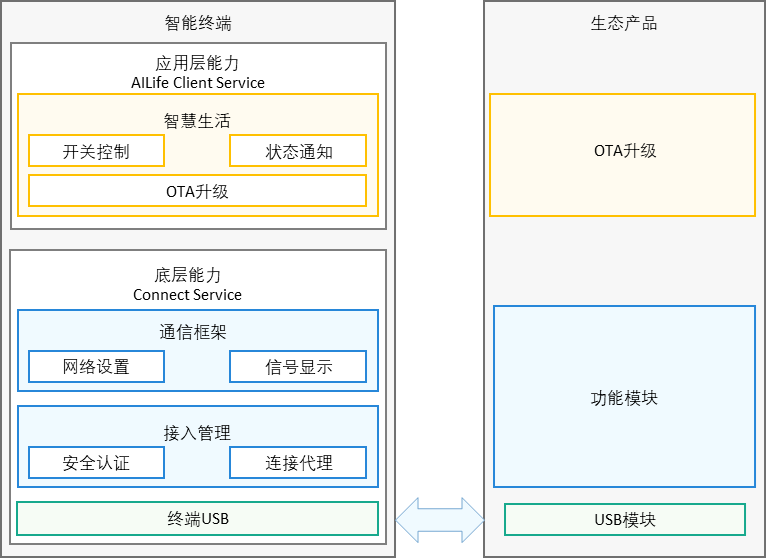
下图以应用迁移为例，简要地展示了分布式任务调度能力。



[典型应用场景](https://developer.harmonyos.com/cn/docs/documentation/doc-guides/hop-overview-0000001092995092" \t "https://developer.harmonyos.com/cn/docs/documentation/doc-guides/_blank)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。

#### 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：

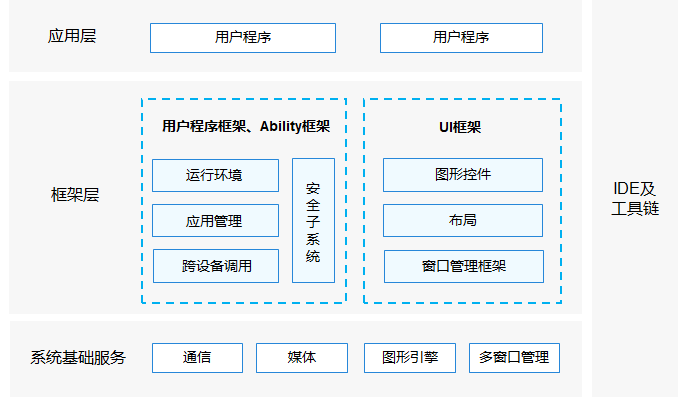
基于分布式连接能力，可以通过开发生态配件拓展智能终端的通信能力：

* 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。

### [一次开发，多端部署](https://developer.harmonyos.com/cn/docs/" \l "section35241557442" \o "一次开发，多端部署)

HarmonyOS提供了用户程序框架、[Ability](https://developer.harmonyos.com/cn/docs/documentation/doc-guides/glossary-0000000000029587" \l "ZH-CN_TOPIC_0000001114162884__li1373094219463" \t "https://developer.harmonyos.com/cn/docs/documentation/doc-guides/_blank)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](https://developer.harmonyos.com/cn/docs/documentation/doc-guides/harmonyos-features-0000000000011907" \l "ZH-CN_TOPIC_0000001175461719__fig087974703213)。

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。



### [统一OS，弹性部署](https://developer.harmonyos.com/cn/docs/" \l "section1169115112119" \o "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。

# 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。

## 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

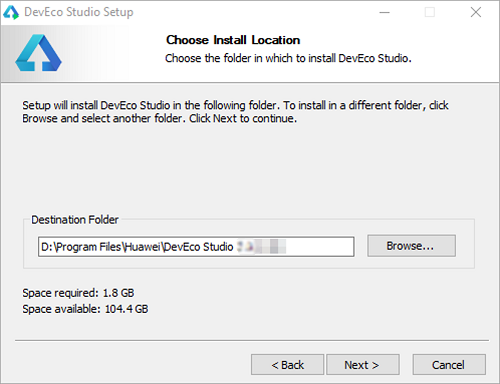
进入[DevEco Studio下载官网](https://developer.harmonyos.com/cn/develop/deveco-studio" \t "https://developer.huawei.com/consumer/cn/doc/html/doc-guides-V3/_blank)：https://developer.harmonyos.com/cn/develop/deveco-studio，单击“立即下载”进入下载页面。



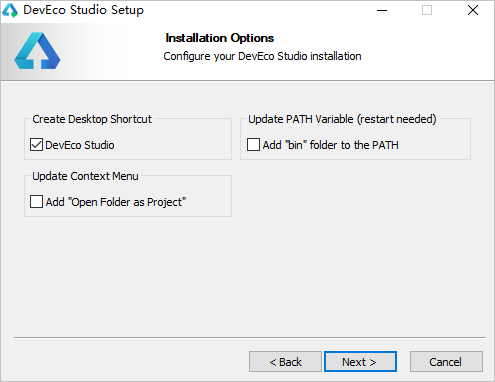
DevEco Studio提供了Windows版本和Mac版本选择，可以根据操作系统选择对应的版本进行下载。



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。

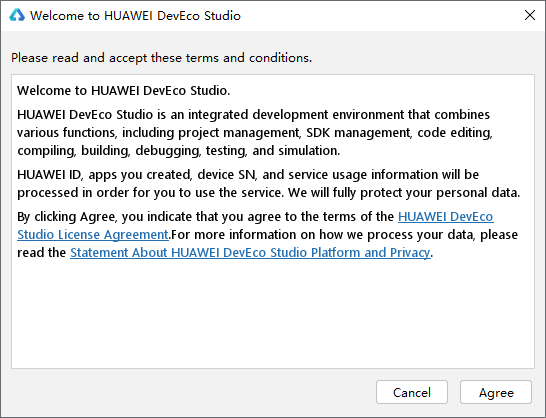


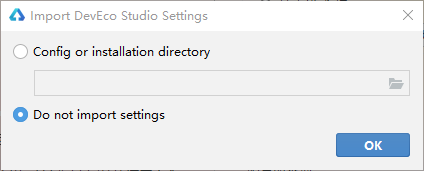
安装完成后，单击“Finish”完成安装。



## 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：  


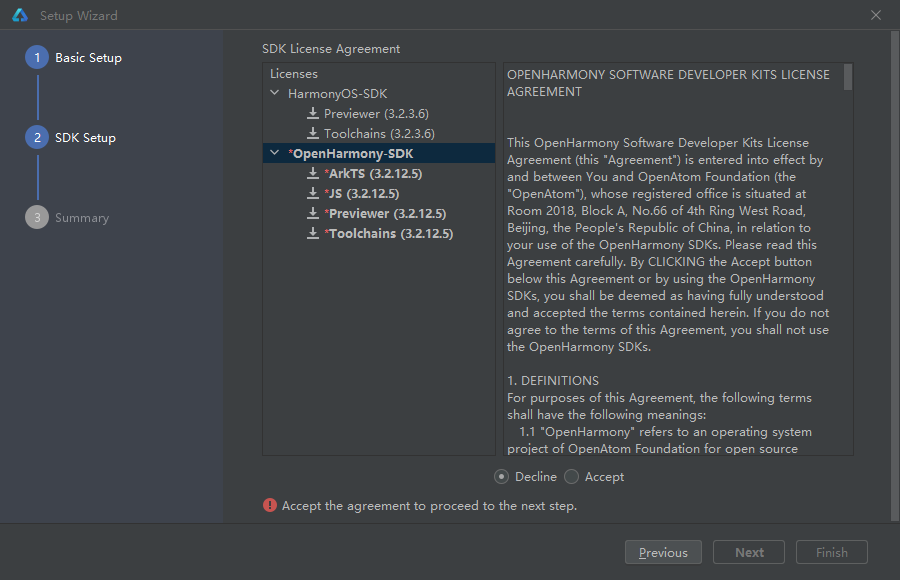
进入DevEco Studio配置页面，首先需要进行基础配置，包括Node.js与Ohpm的安装路径设置，选择从华为镜像下载至合适的路径。



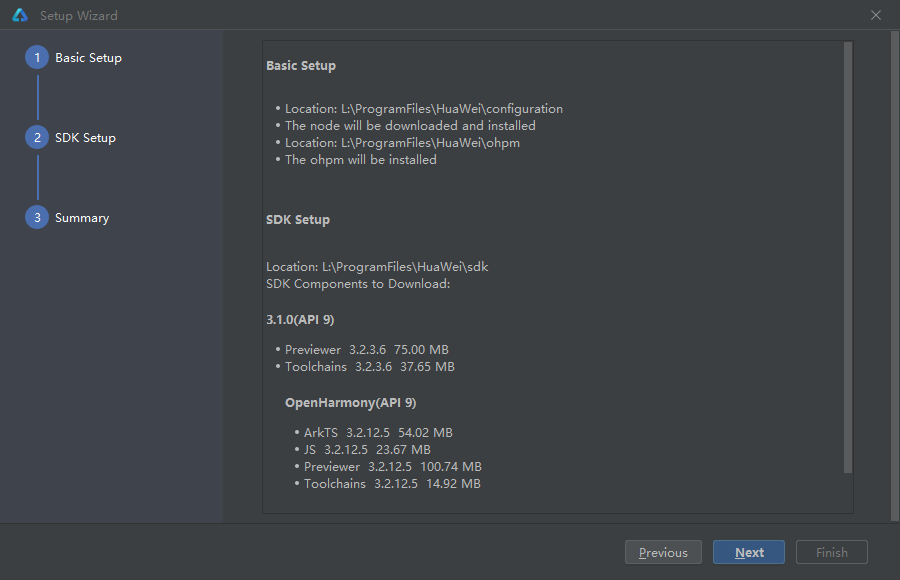
单击'Next'进入SDK配置，设置为合适的路径，



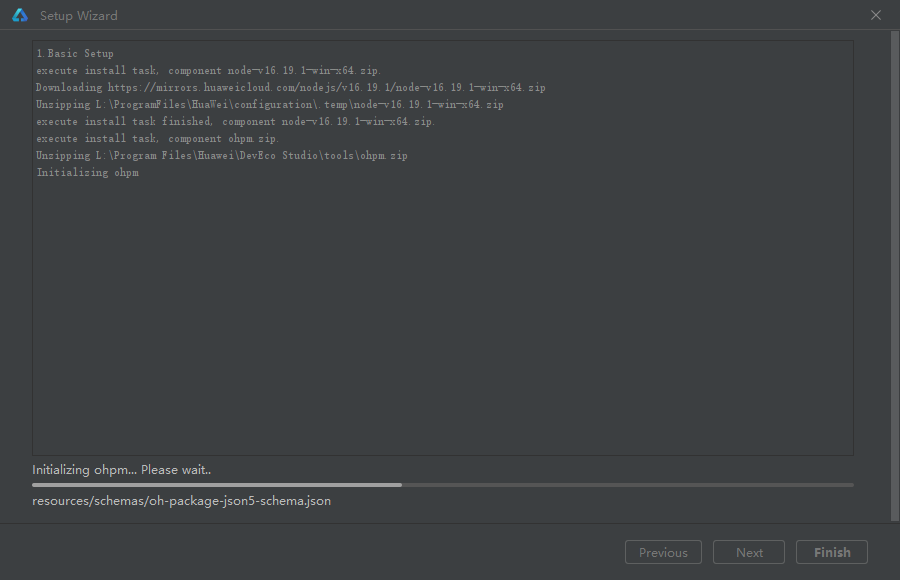
点击'Next'后会显示'SDK License Agreement'，阅读相关协议后，勾选'Accept'。

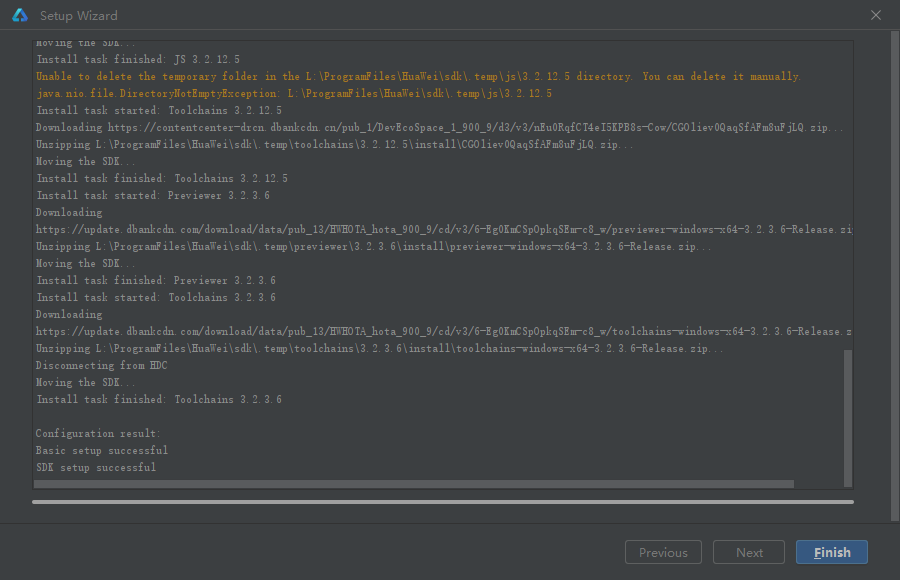


单击‘Next’进入配置预览页，在这里进行配置项的确认。

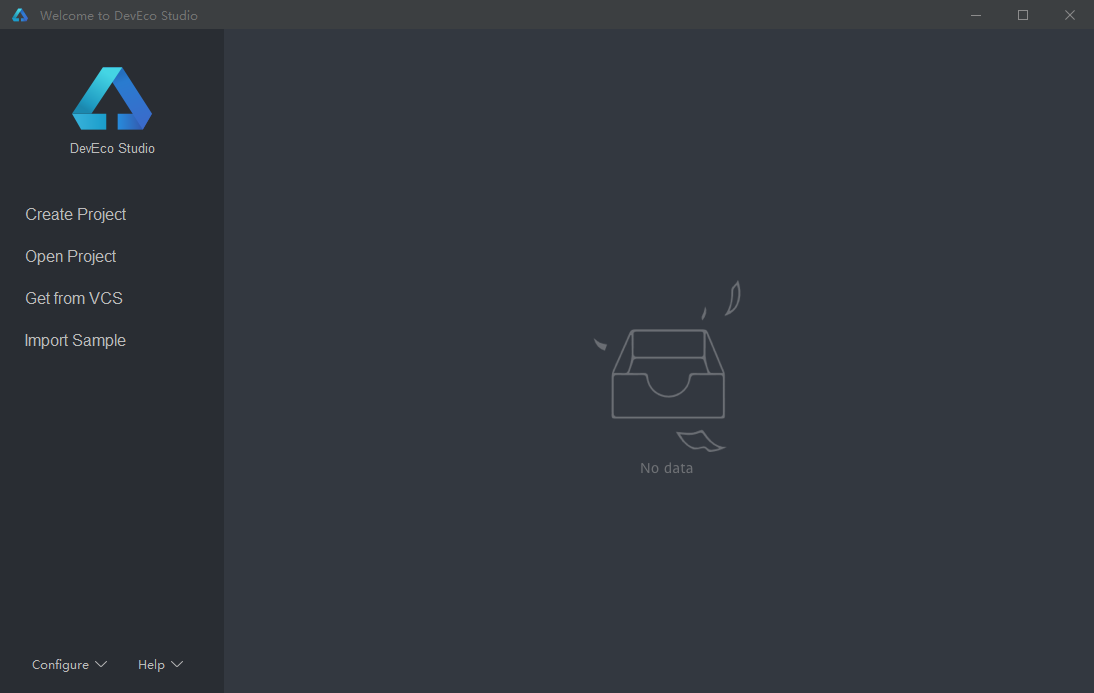


确认完成后，单击'Next'，进入下一步。





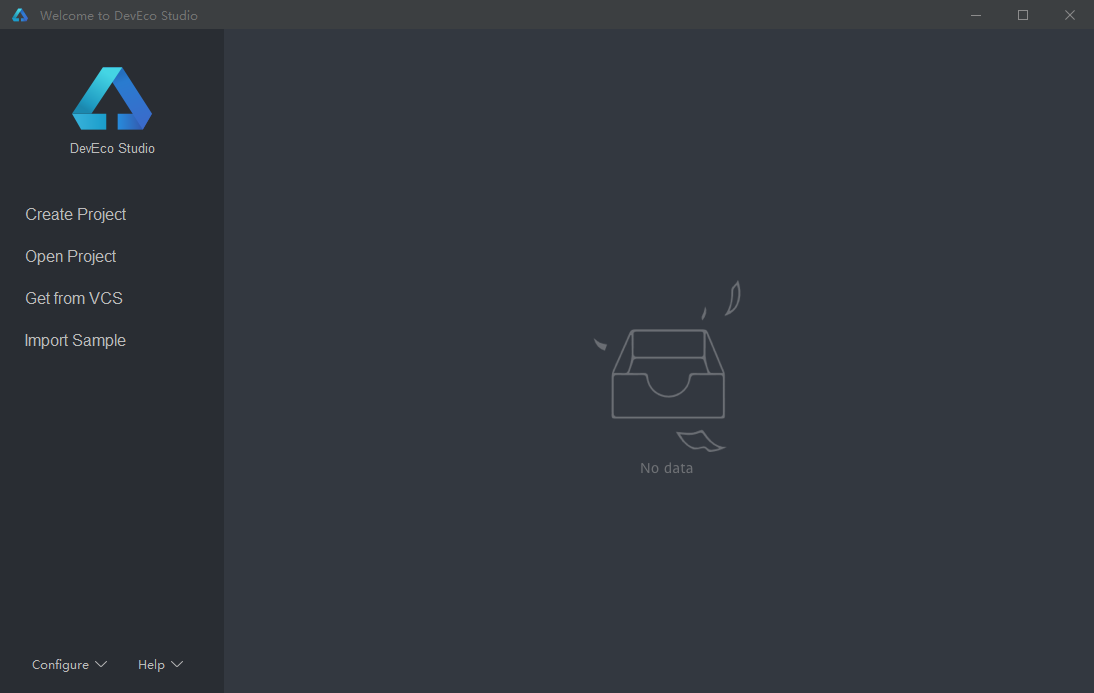
等待配置自动下载完成，完成后，单击'Finish'，IDE会进入欢迎页，我们也就成功配置好了开发环境。



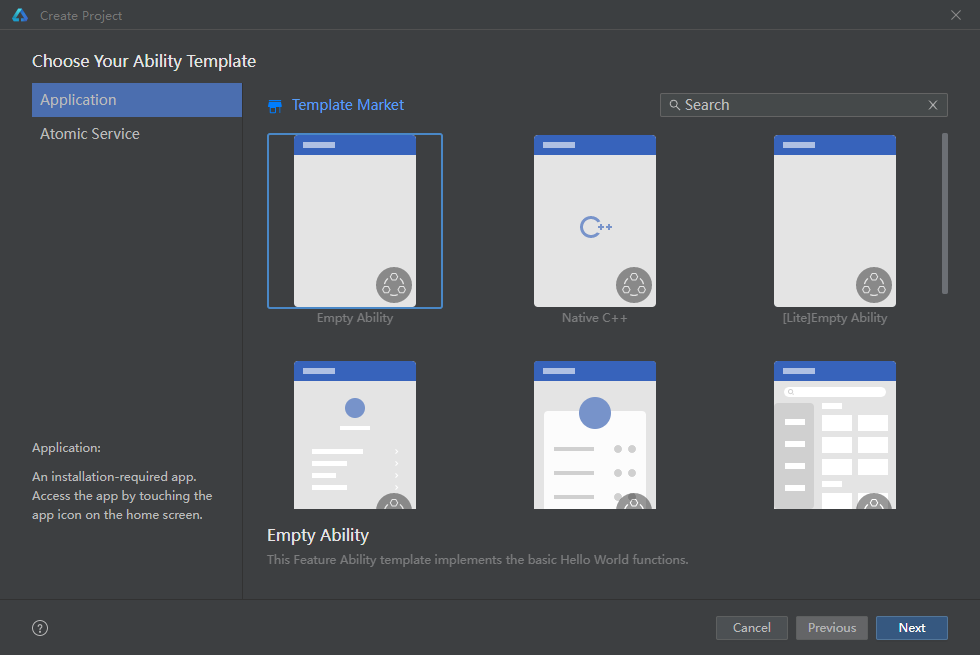
准备工作完成后，接下来将进入DevEco Studio进行工程创建和运行。

## 创建项目

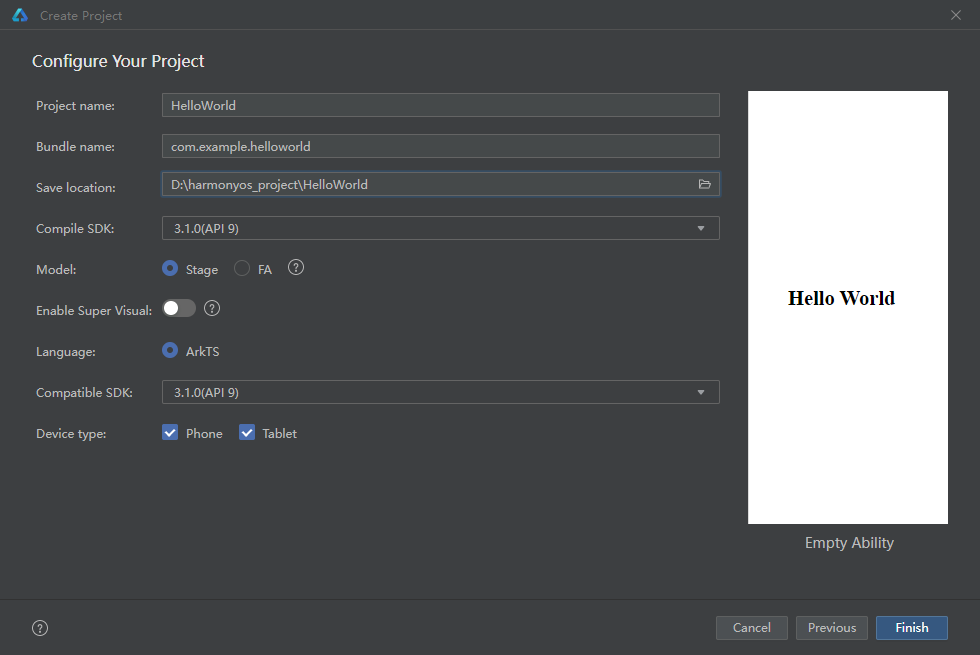
如果你是首次打开DevEco Studio，那么首先会进入欢迎页。



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。



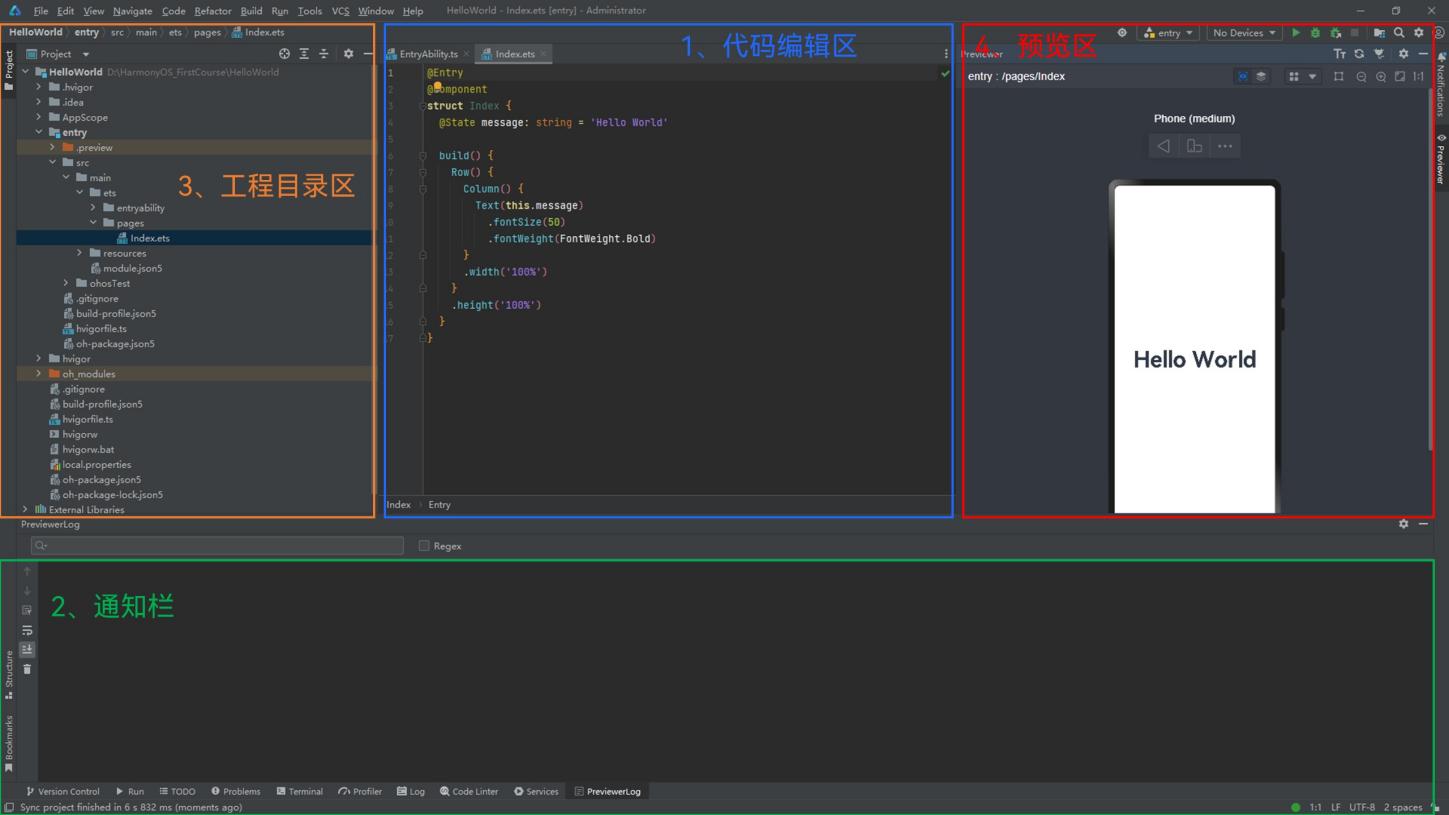
配置页中，详细信息如下：

* 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。

## 认识DevEco Studio 界面

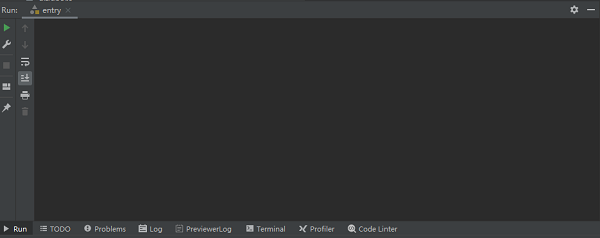
进入IDE后，我们首先了解一下基础的界面。整个IDE的界面大致上可以分为四个部分，分别是代码编辑区、通知栏、工程目录区以及预览区。



### 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。

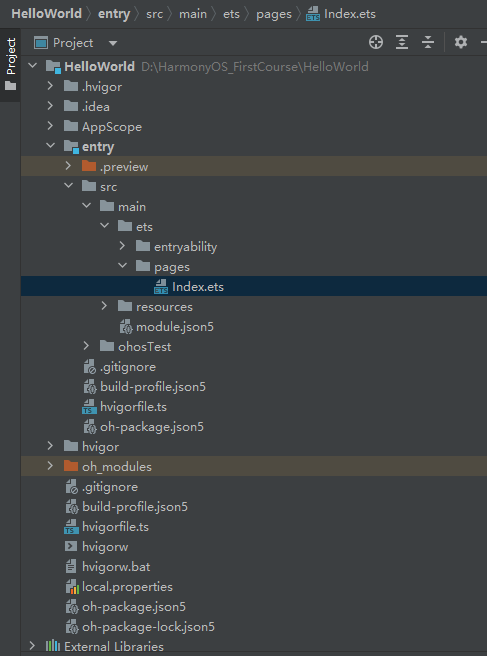
### 通知栏



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。

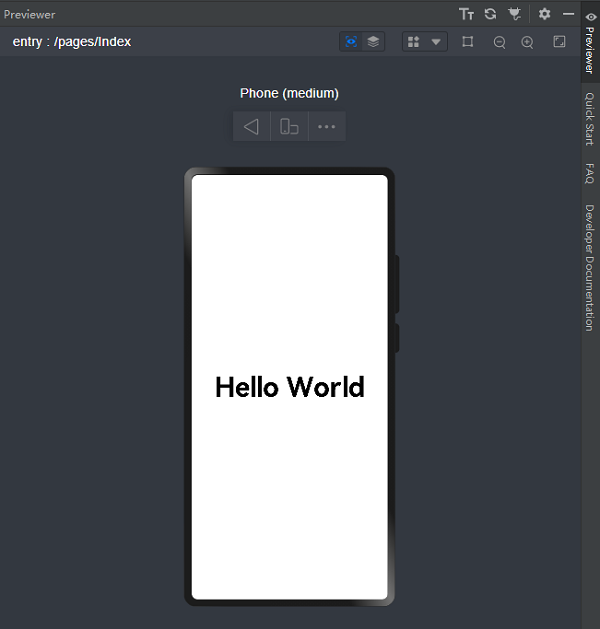
### 工程目录区

左侧为工程目录区，后续章节会详细介绍。

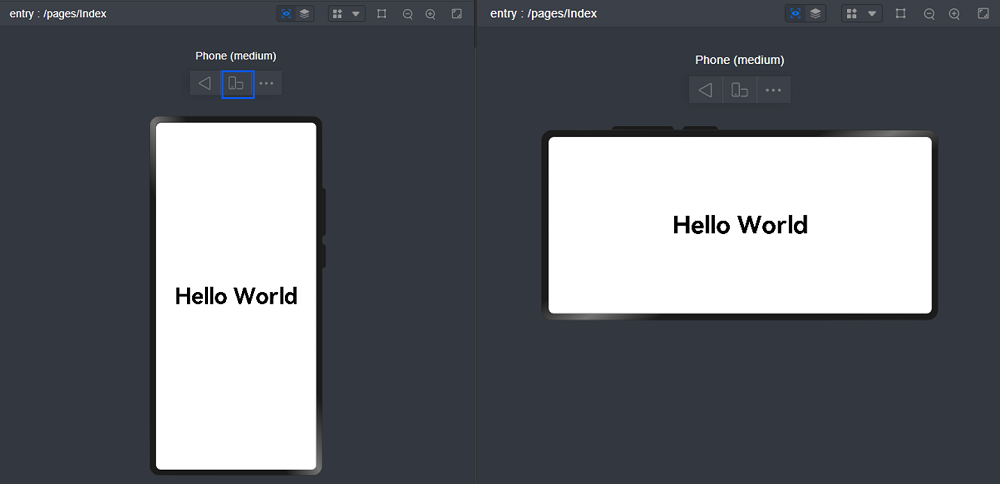


### 预览区

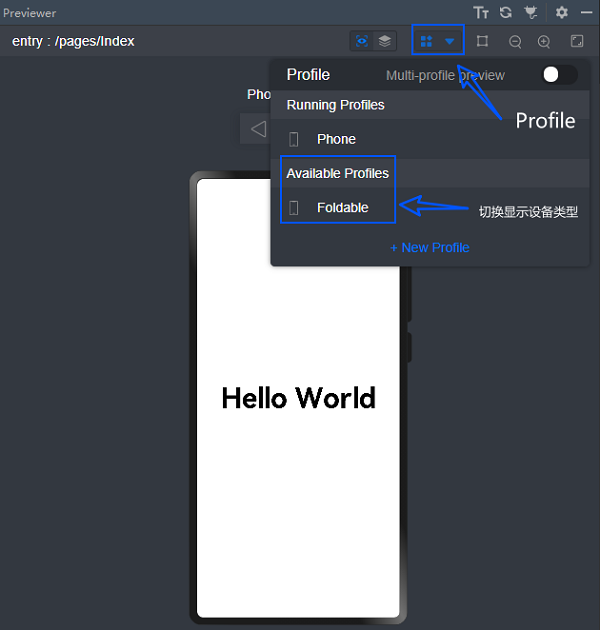
单击右上角Previewer，可以预览相应的文件UI展示效果。



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。

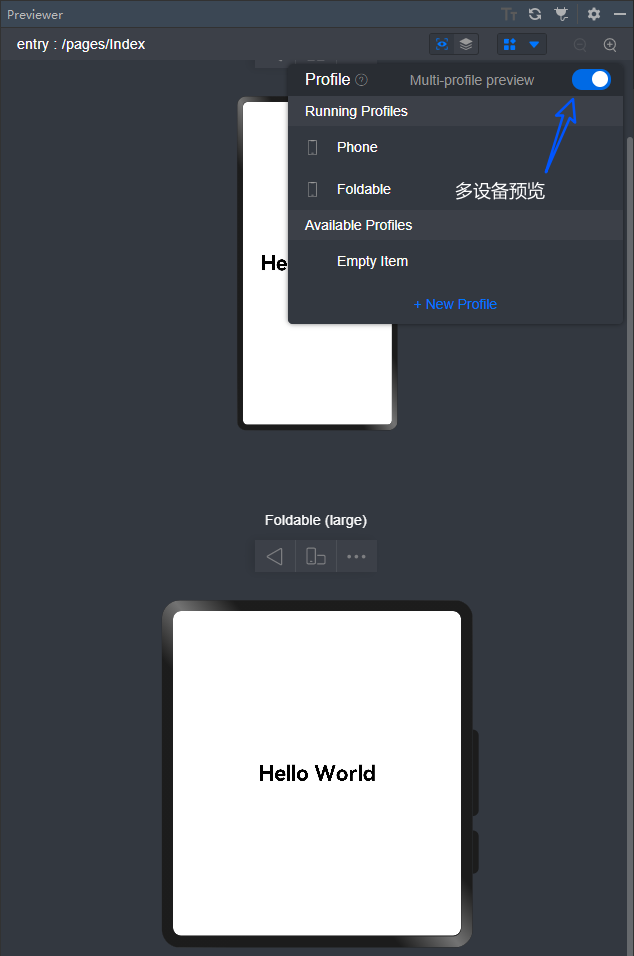


也可以单击如下列表按钮，切换显示的设备类型。弹出框内会显示Available Profiles，即可用的设备类型。

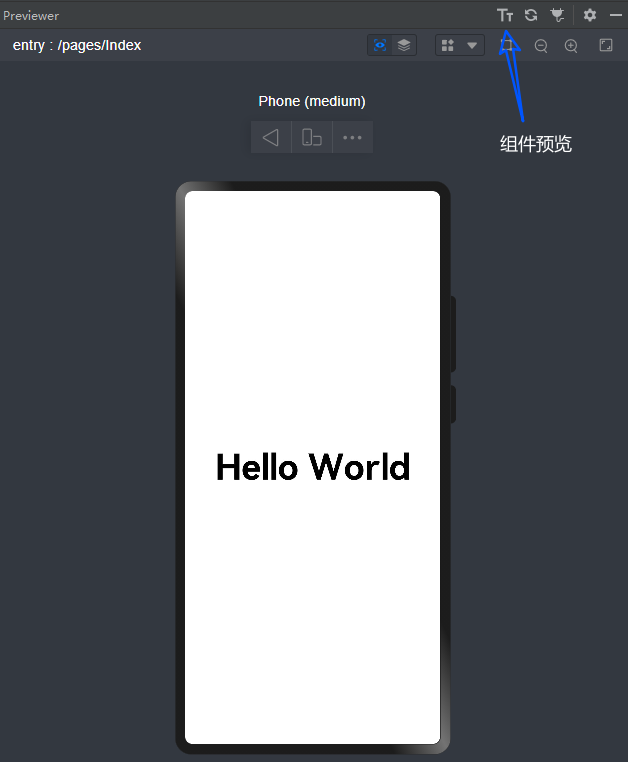


如单击Foldable切换设备，也可以单击旋转按钮切换Foldable的横竖屏显示模式。

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

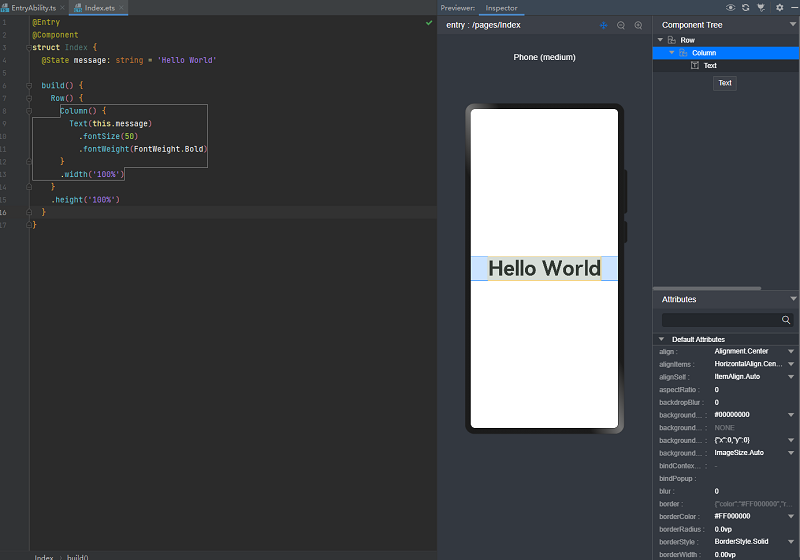


单击预览器右上角组件预览按钮，可以进入组件预览界面。



组件预览模式可以预览当前组件对应的代码块。

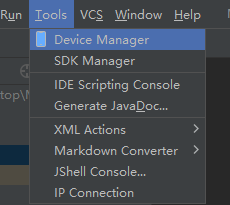
点击相应组件，代码文件中会框选对应的组件代码部分，下方则对应当前组件的基本属性。



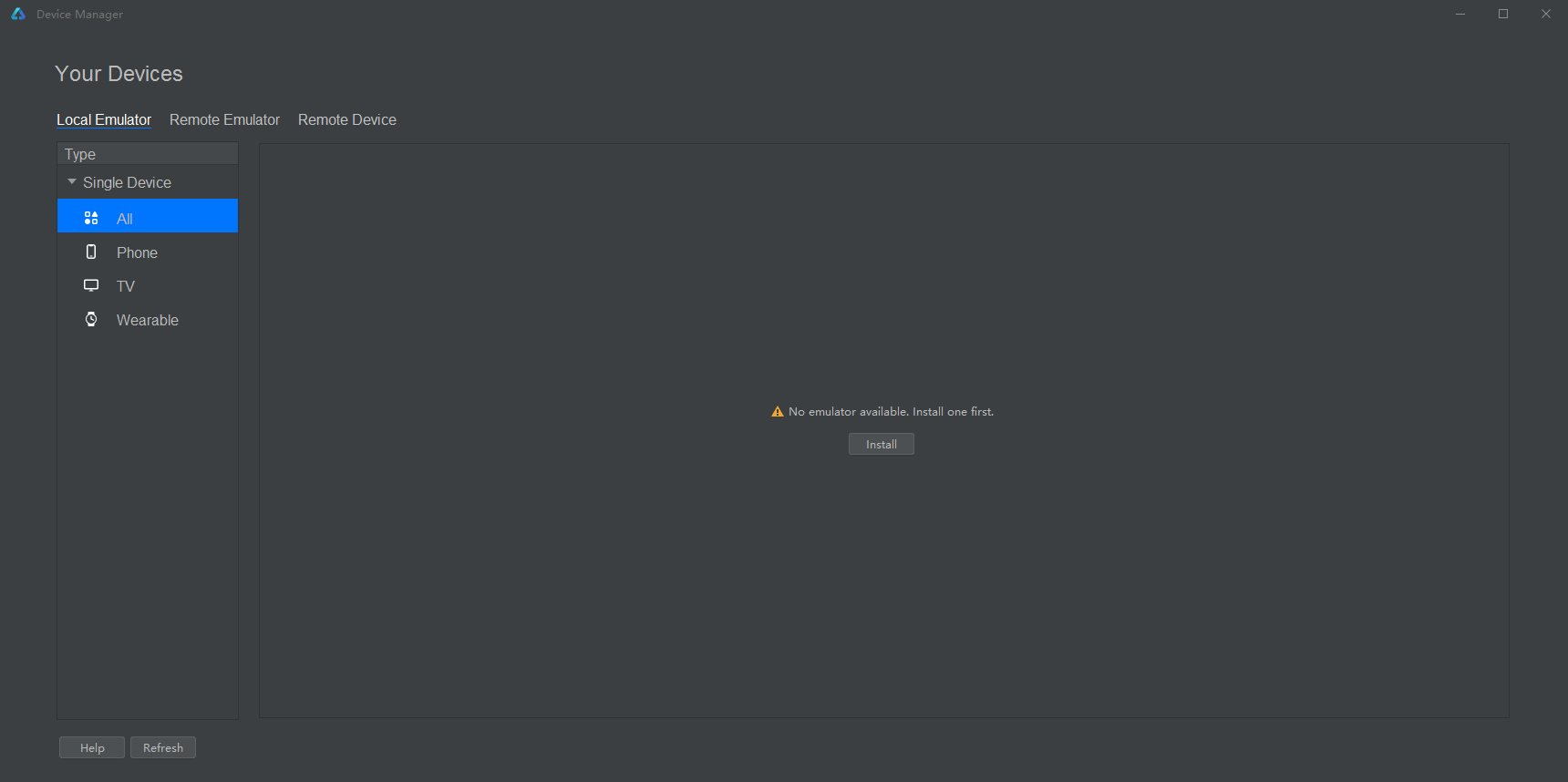
## 运行Hello World

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。

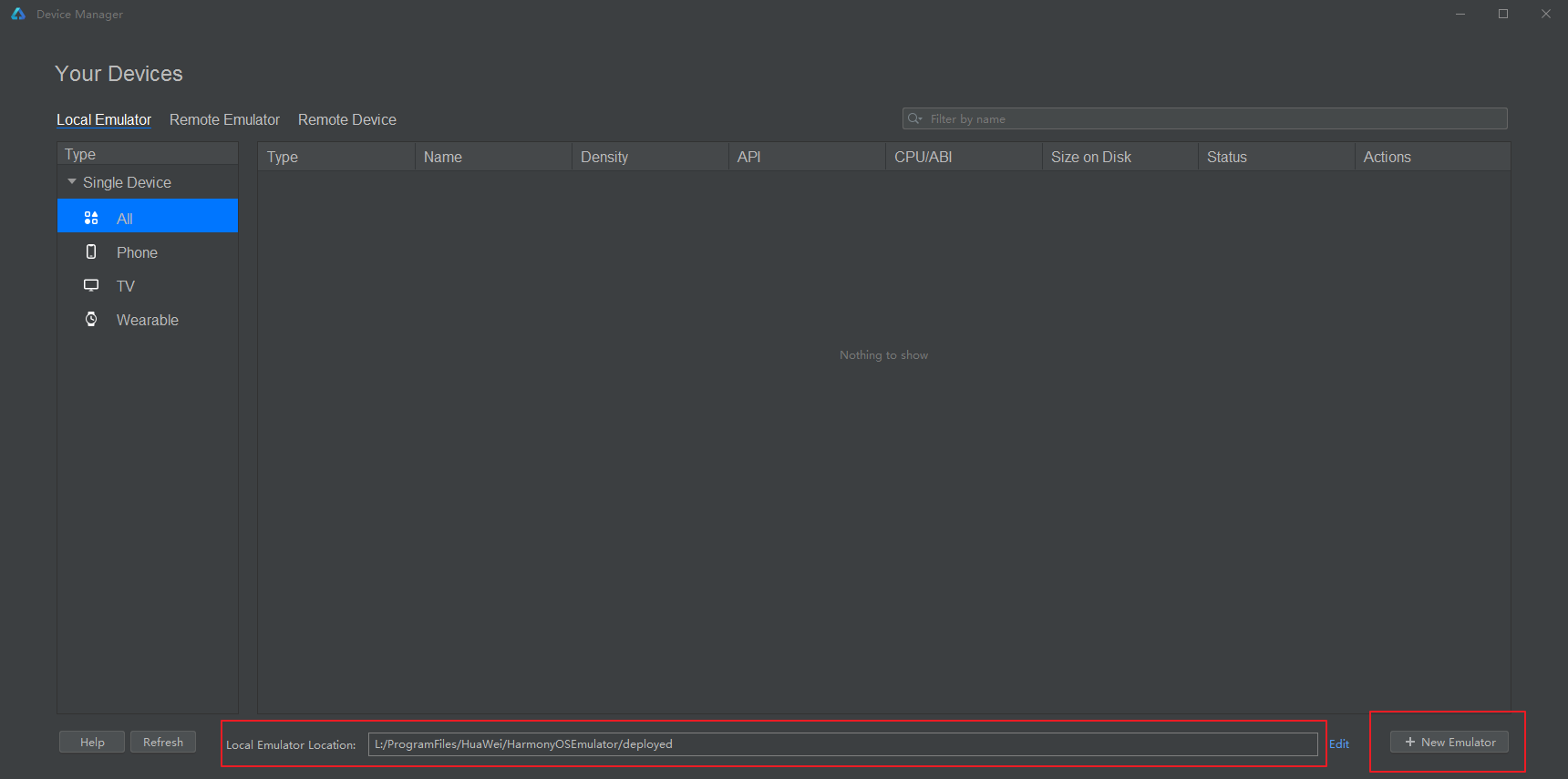
1. 单击顶部工具栏Tools>Device Manager。



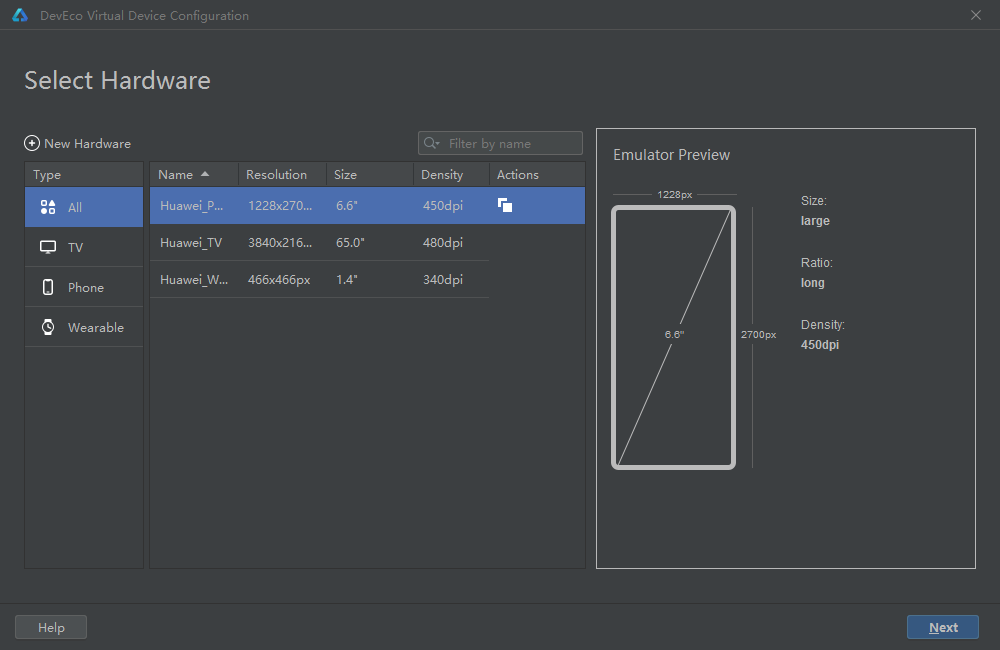
1. 选择Local Emulator，安装模拟器



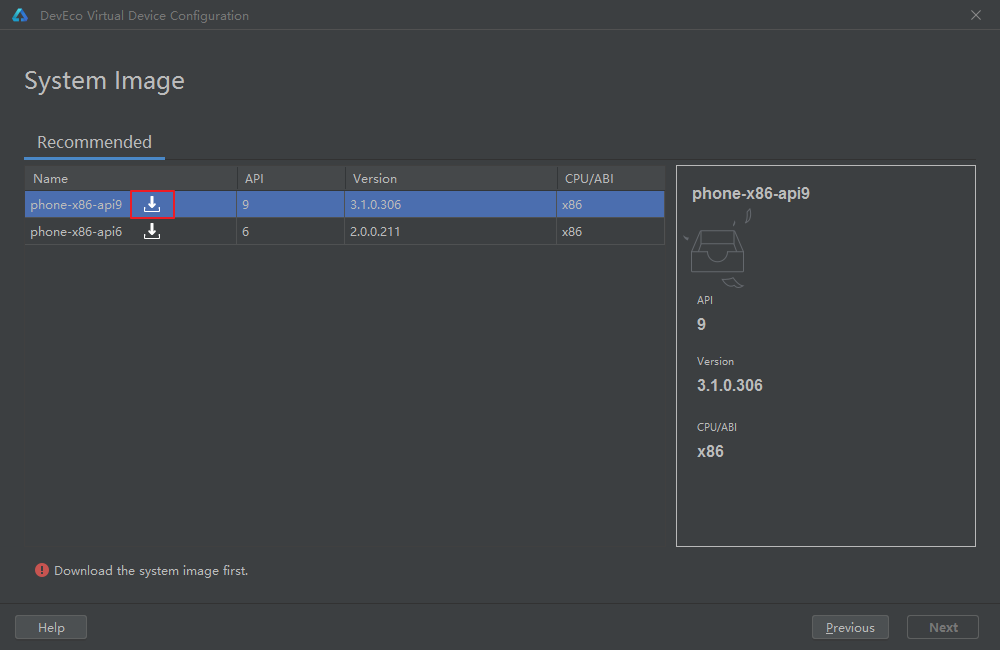
设置合适的Local Emulator Location存储地址，然后单击’+New Emulator’。

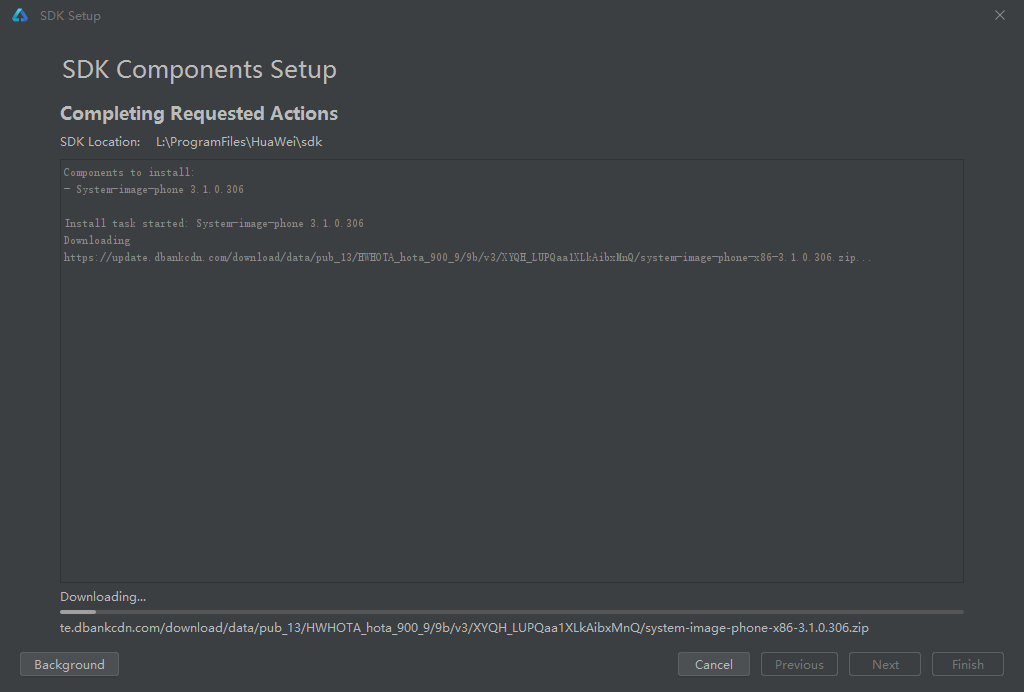


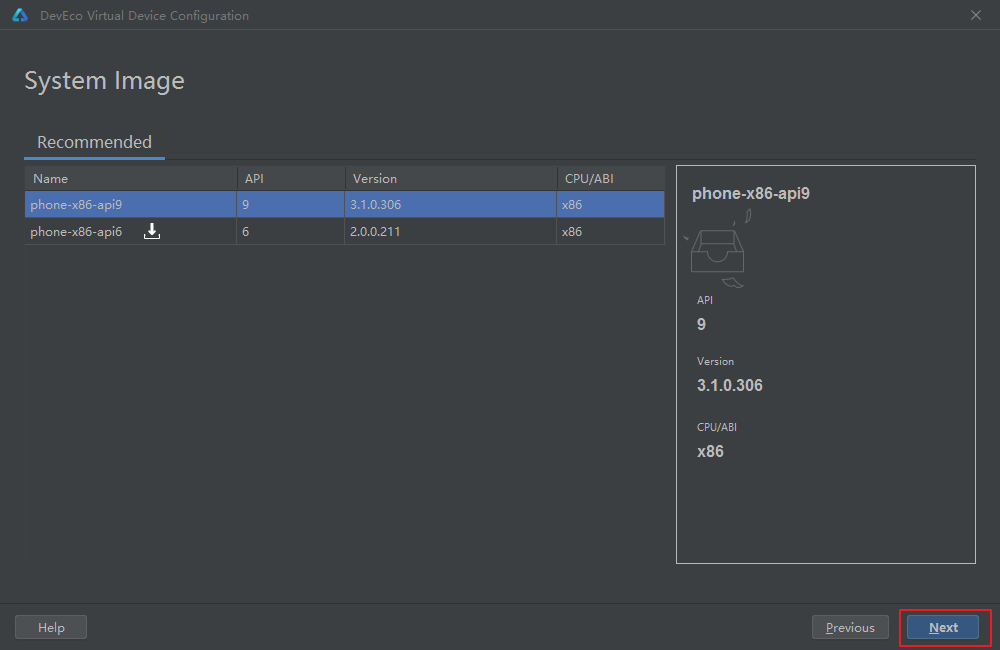
选择Huawei\_Phone手机模拟器，单击'Next'，进入模拟器系统下载页。



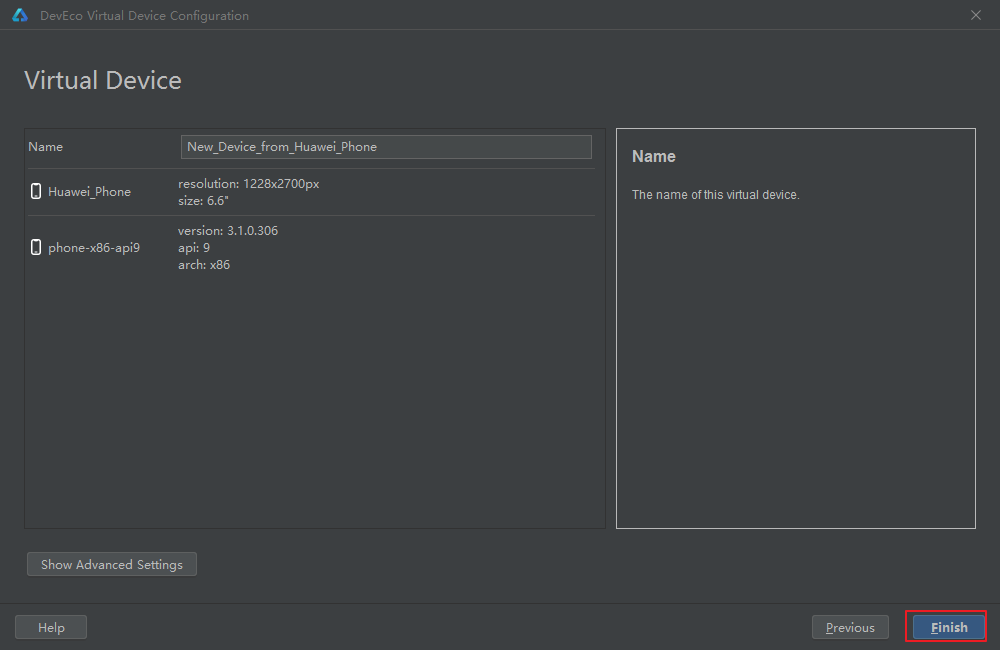
选择下载api9的系统镜像，然后单击'Next’，等待下载完成。



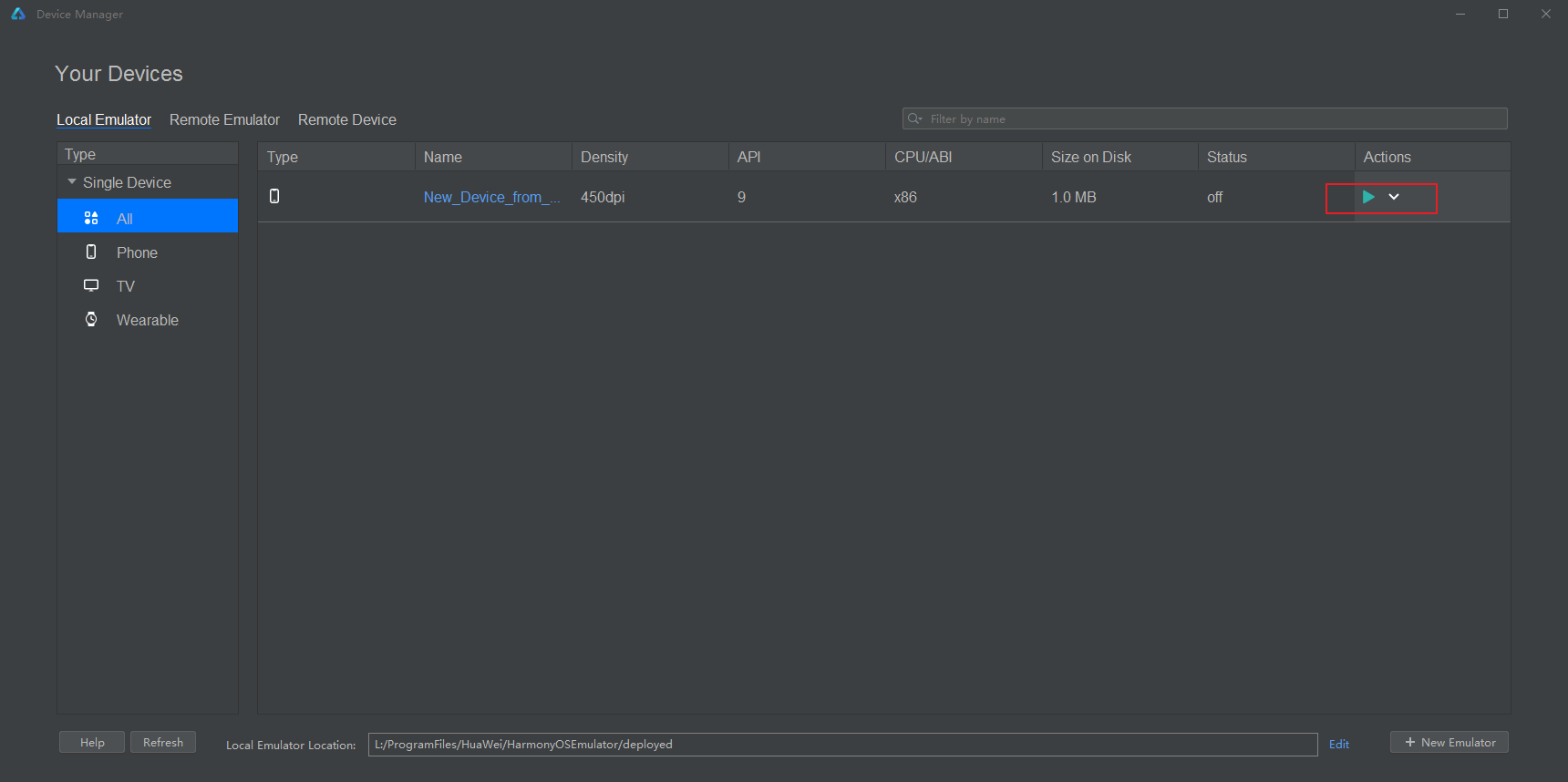


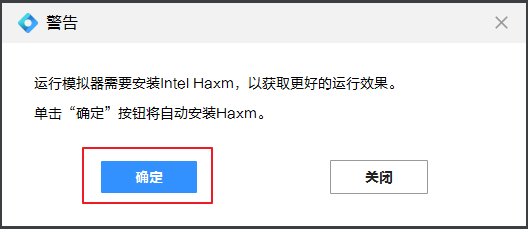


下载完成后，进行创建相应的手机模拟器，单击Finish完成创建。

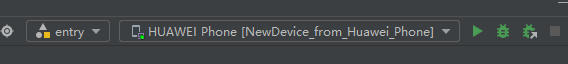


下载完成后，在Local Emulator页面中会出现创建的手机模拟器，点击Actions按钮，就能够启动模拟器。

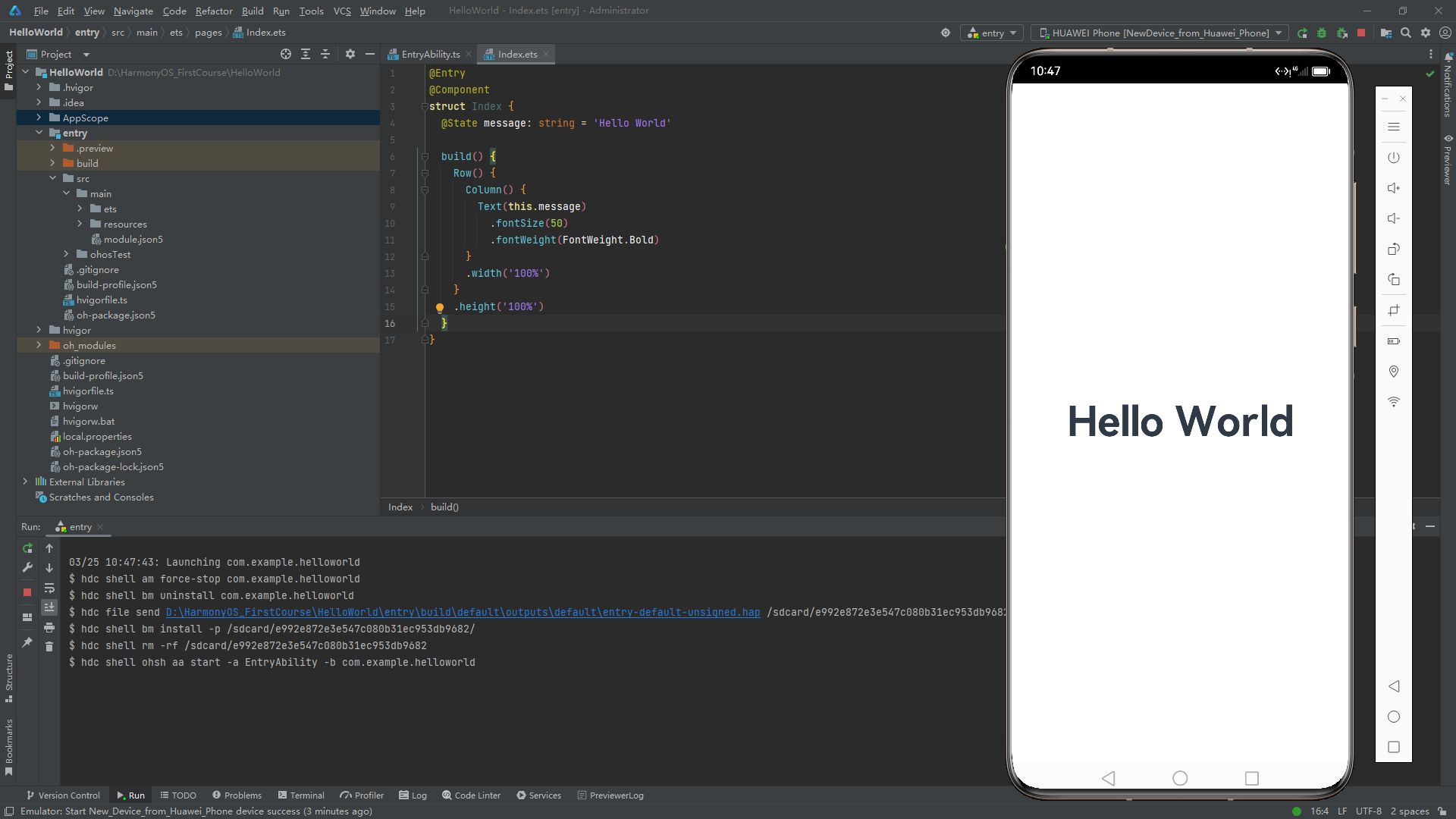




模拟器启动后，点击上方启动按钮，将Hello World工程运行到模拟器上。



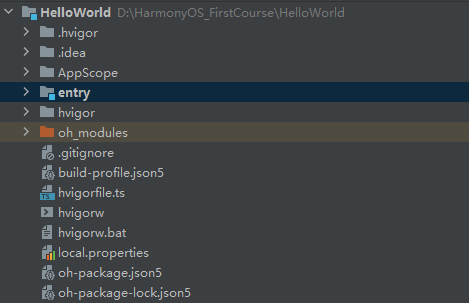
IDE构建完成后，即可在模拟器上看到运行效果，我们也就完成了Hello World工程在模拟器上的运行。



## 了解基本工程目录

### 工程级目录

工程的目录结构如下。



其中详细如下：

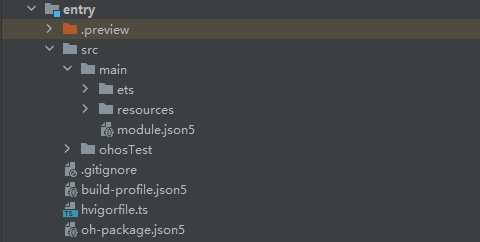
* AppScope中存放应用全局所需要的资源文件。
* entry是应用的主模块，存放HarmonyOS应用的代码、资源等。
* oh\_modules是工程的依赖包，存放工程依赖的源文件。
* build-profile.json5是工程级配置信息，包括签名、产品配置等。
* hvigorfile.ts是工程级编译构建任务脚本，hvigor是基于任务管理机制实现的一款全新的自动化构建工具，主要提供任务注册编排，工程模型管理、配置管理等核心能力。
* oh-package.json5是工程级依赖配置文件，用于记录引入包的配置信息。

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。



### 模块级目录

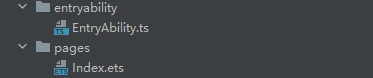


entry>src目录中主要包含总的main文件夹，单元测试目录ohosTest，以及模块级的配置文件。

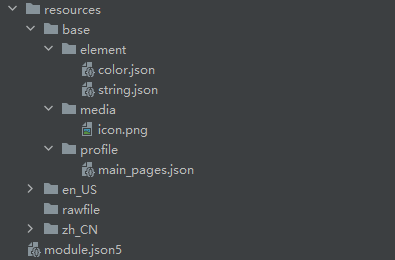
* main文件夹中，ets文件夹用于存放ets代码，resources文件存放模块内的多媒体及布局文件等，module.json5文件为模块的配置文件。
* ohosTest是单元测试目录。
* build-profile.json5是模块级配置信息，包括编译构建配置项。
* hvigorfile.ts文件是模块级构建脚本。
* oh-package.json5是模块级依赖配置信息文件。

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。



### 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。

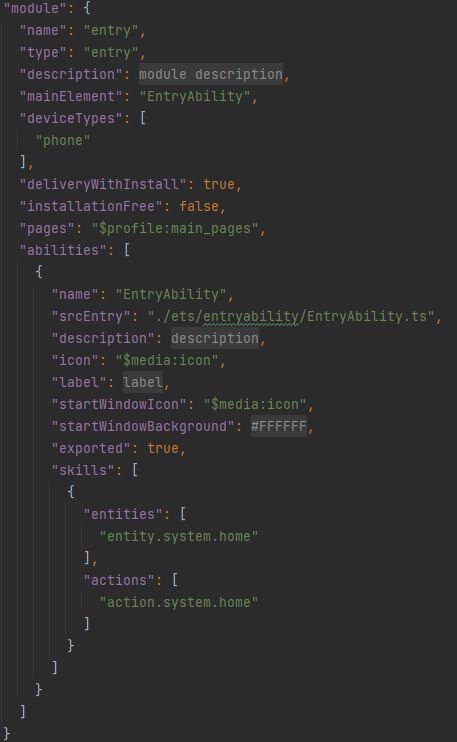


其中配置信息如下：

* 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。

### 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。



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** |
| --- | --- |
| 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。 |

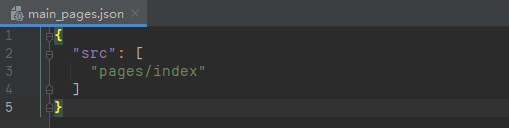
对于abilities中每一个ability的属性项，其描述信息如下表2。

表2 abilities中对象的默认配置属性及描述

| **属性** | **描述** |
| --- | --- |
| 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。 |
| startWindowBackground | 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。 |

### 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。



## 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**
4. main\_pages.json
5. module.json5
6. app.json5
7. package.json
8. **如何在DevEco Studio中创建新项目？BC**
9. 在计算机上创建一个新文件，并将其命名为“new harmonyOS项目”
10. 如果已打开项目，从DevEco Studio菜单选择'file>new>Create Project'
11. 如果第一次打开DevEco Studio，在欢迎页点击“Create new Project”
12. **module.json5配置文件中，包含了以下哪些信息？ABD**
13. ability的相关配置信息
14. 模块名
15. 应用的版本号
16. 模块类型

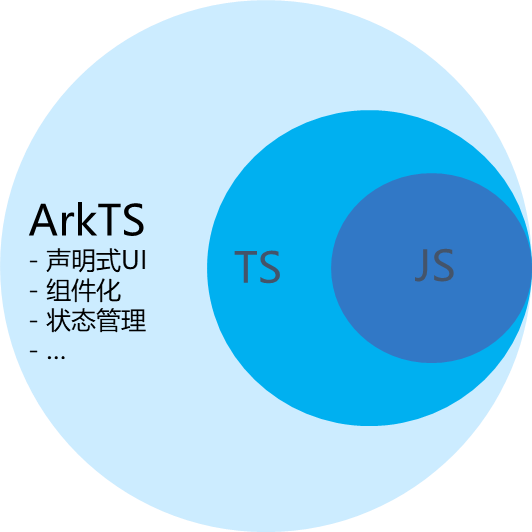
# 第三章 ArkTS开发语言介绍

## 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。

### 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。

### 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; |

### 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;  }  } |

### 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")  }) |

### 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。

### 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。

### 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  } |

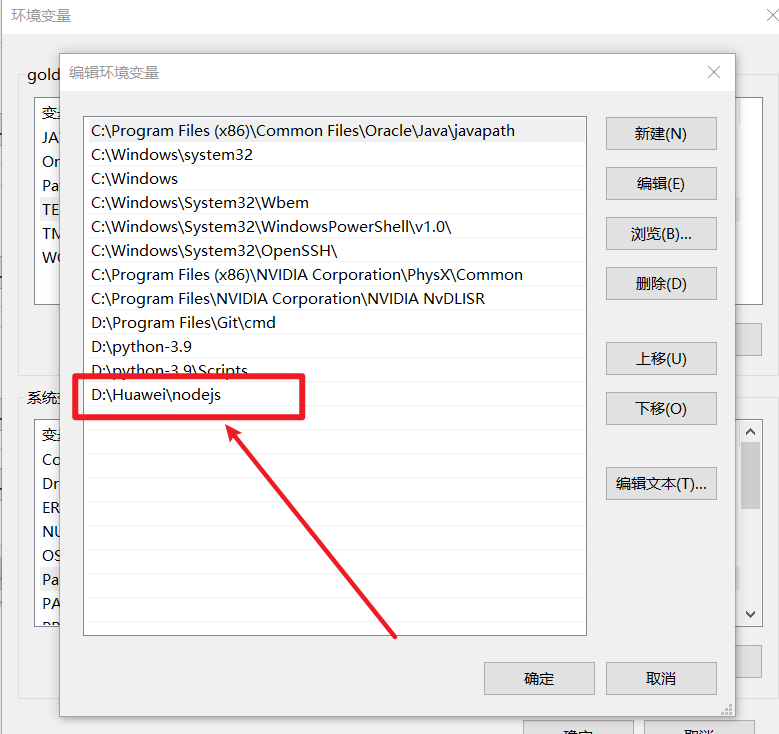
* **for..of vs. for..in 语句**

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"  } |

### 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

## 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](https://www.typescriptlang.org/" \t "https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/_blank)（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](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-basic-syntax-overview-0000001531611153-V3)：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](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-state-management-overview-0000001524537145-V3)：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](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-rendering-control-overview-0000001543911149-V3)：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。

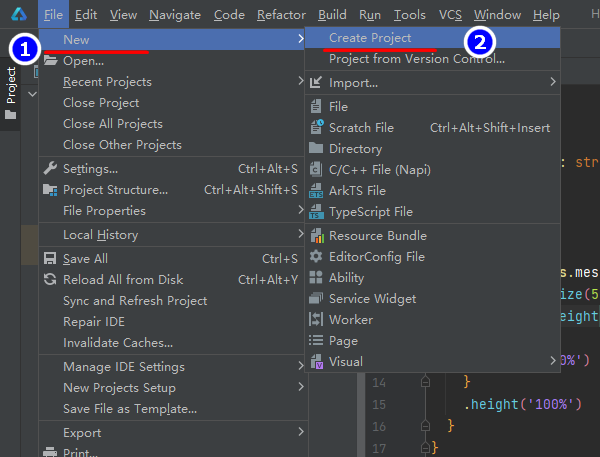
未来，ArkTS会结合应用开发/运行的需求持续演进，逐步提供并行和并发能力增强、系统类型增强、分布式开发范式等更多特性。

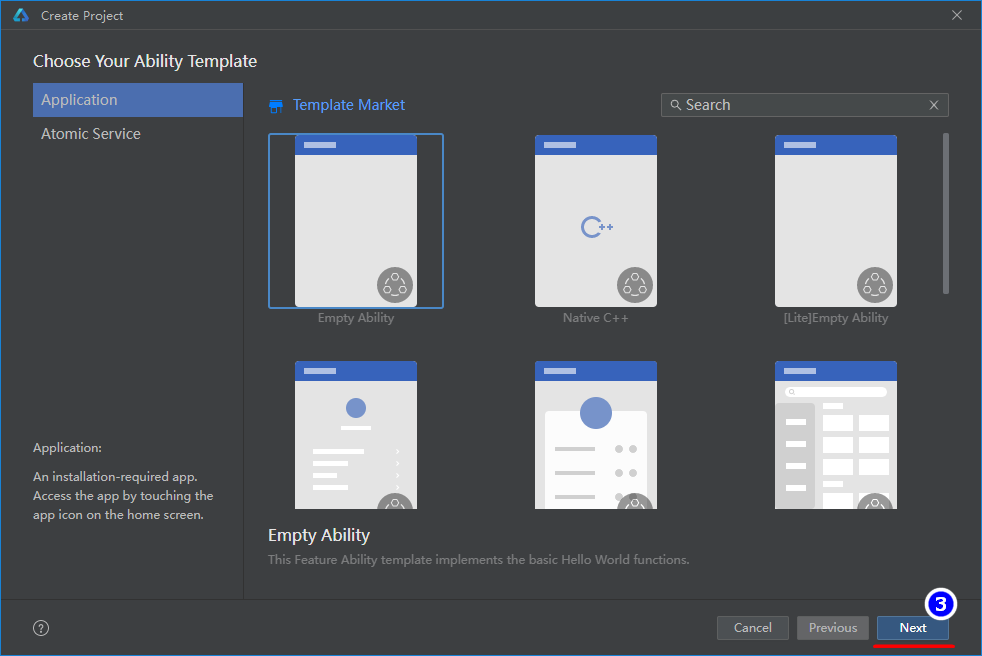
## ArkTS基本语法

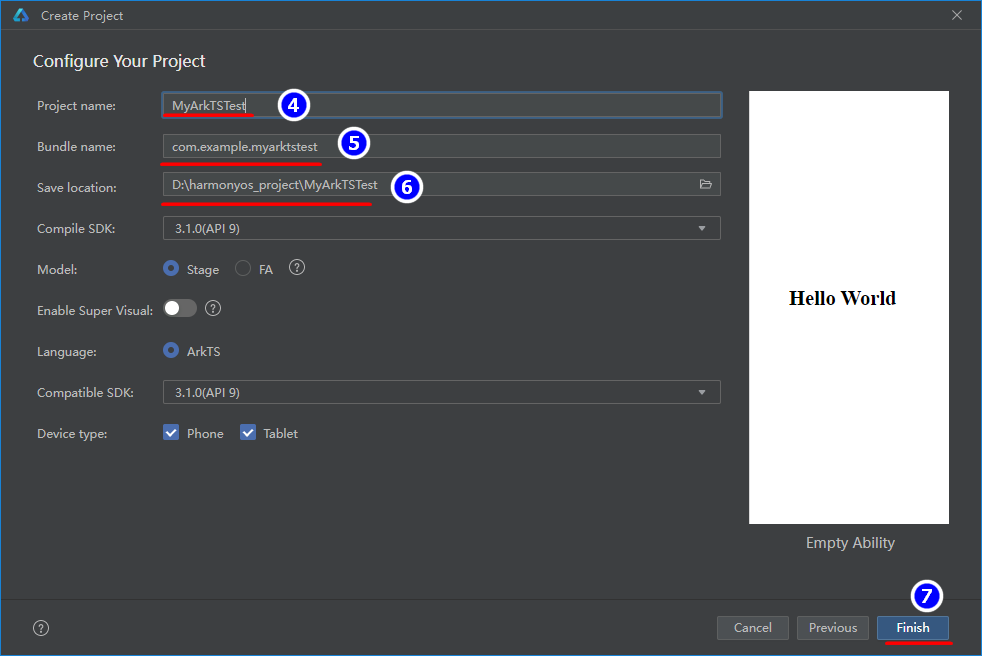
### 基本语法概述

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。

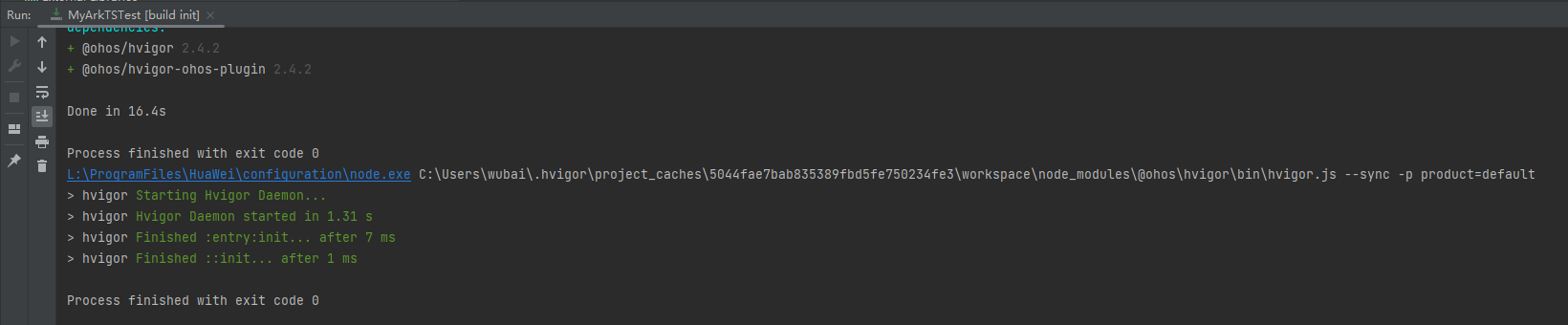
1. **打开 DevEvo Studio开发工具，新建项目**







项目创建完成，进入该项目等待项目初始化完成即可：



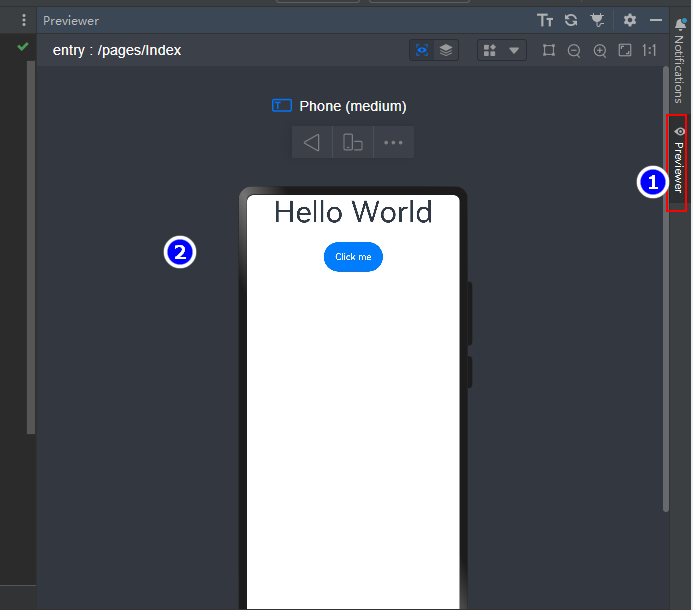
1. **在Index.ets中写入如下代码**

|  |
| --- |
| @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})  }  } } |

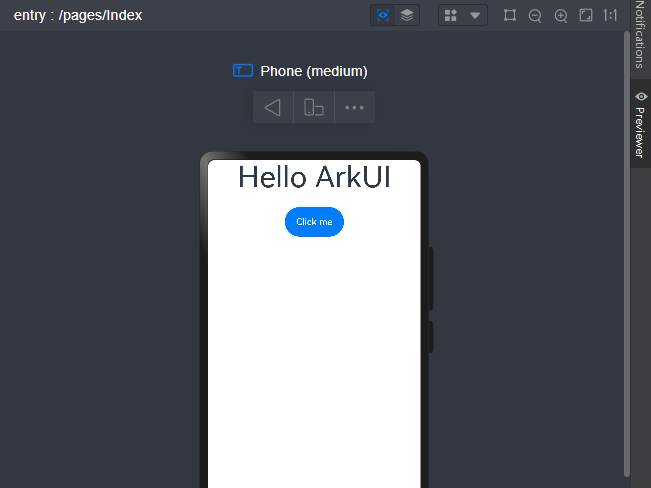
对以上代码详细解释如下：

|  |
| --- |
| //@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  }  } } |

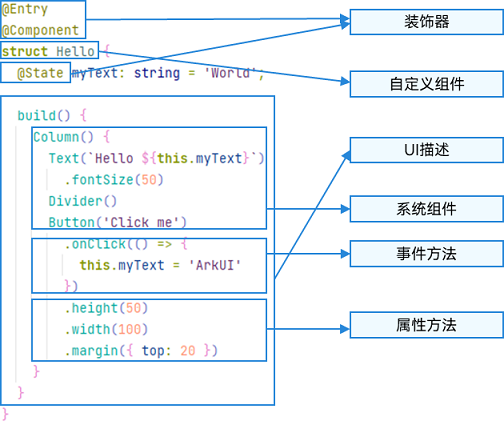
1. **打开预览，验证组件功能**



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](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-create-custom-components-0000001473537046-V3" \l "section1430055924816)表示自定义组件，[@Entry](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-create-custom-components-0000001473537046-V3" \l "section1430055924816)表示该自定义组件为入口组件，[@State](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-state-0000001474017162-V3)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](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-declarative-ui-description-0000001524416537-V3)：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](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-create-custom-components-0000001473537046-V3)：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](https://developer.harmonyos.com/cn/docs/documentation/doc-references-V3/ts-components-summary-0000001478181369-V3" \t "https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/_blank)。除此之外，ArkTS扩展了多种语法范式来使开发更加便捷：

* [@Builder](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-builder-0000001524176981-V3)/[@BuilderParam](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-builderparam-0000001524416541-V3)：特殊的封装UI描述的方法，细粒度的封装和复用UI描述。
* [@Extend](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-extend-0000001473696678-V3)/[@Style](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-style-0000001473856690-V3)：扩展内置组件和封装属性样式，更灵活地组合内置组件。
* [stateStyles](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-statestyles-0000001482592098-V3)：多态样式，可以依据组件的内部状态的不同，设置不同样式。

### 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。

#### 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**

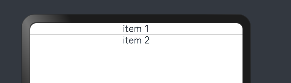
如果组件的接口定义没有包含必选构造参数，则组件后面的“()”不需要配置任何内容。例如，Divider组件不包含构造参数：

|  |
| --- |
| 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。

1. Image组件的必选参数src。

|  |
| --- |
| Image('https://xyz/test.jpg') |

1. 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() |

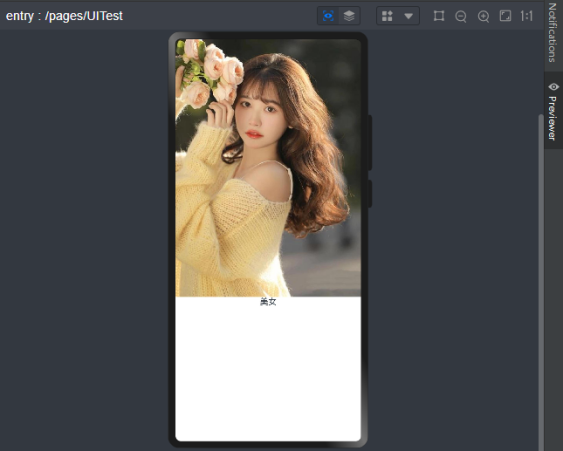
1. 变量或表达式也可以用于参数赋值，其中表达式返回的结果类型必须满足参数类型要求。

|  |
| --- |
| 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：



#### 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。**

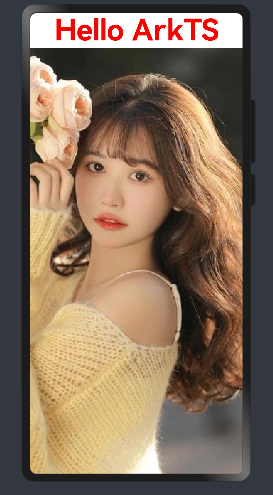
例如，可以按以下方式配置Text组件的颜色和字体样式。

|  |
| --- |
| 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：



#### 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)) |

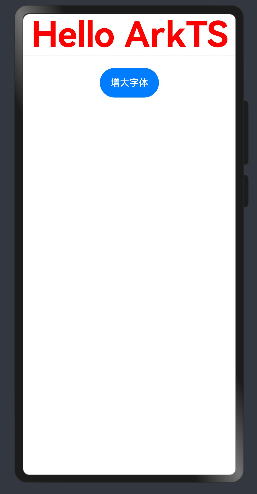
* 使用组件的成员函数配置组件的事件方法。

|  |
| --- |
| 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。



#### 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)  } |

### 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。

#### 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](https://developer.harmonyos.com/cn/docs/documentation/doc-references-V3/ts-types-0000001477981241-V3" \l "ZH-CN_TOPIC_0000001573928889__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。  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：

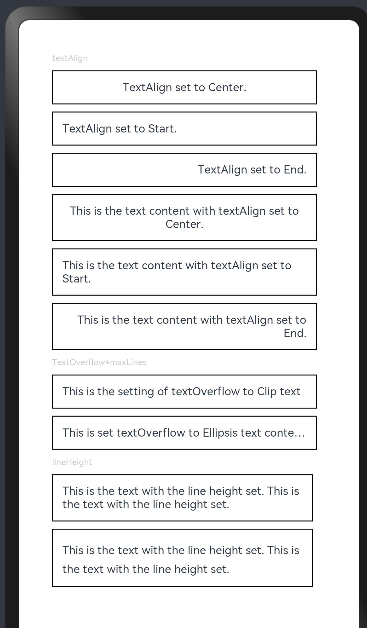
| **名称** | **参数类型** | **描述** |
| --- | --- | --- |
| textAlign | [TextAlign](https://developer.harmonyos.com/cn/docs/documentation/doc-references-V3/ts-appendix-enums-0000001478061741-V3" \l "ZH-CN_TOPIC_0000001574248789__textalign) | 设置文本段落在水平方向的对齐方式  默认值：TextAlign.Start  说明：  文本段落宽度占满Text组件宽度。  可通过[align](https://gitee.com/openharmony/docs/blob/f4a705fd02bb48eae9879ce16396442938779f7b/zh-cn/application-dev/reference/arkui-ts/ts-universal-attributes-location.md" \t "https://developer.harmonyos.com/cn/docs/documentation/doc-references-V3/_blank)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](https://developer.harmonyos.com/cn/docs/documentation/doc-references-V3/ts-appendix-enums-0000001478061741-V3" \l "ZH-CN_TOPIC_0000001574248789__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  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](https://developer.harmonyos.com/cn/docs/documentation/doc-references-V3/ts-types-0000001477981241-V3" \l "ZH-CN_TOPIC_0000001573928889__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](https://developer.harmonyos.com/cn/docs/documentation/doc-references-V3/ts-appendix-enums-0000001478061741-V3" \l "ZH-CN_TOPIC_0000001574248789__textdecorationtype),  color?: [ResourceColor](https://developer.harmonyos.com/cn/docs/documentation/doc-references-V3/ts-types-0000001477981241-V3" \l "ZH-CN_TOPIC_0000001573928889__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](https://developer.harmonyos.com/cn/docs/documentation/doc-references-V3/ts-types-0000001477981241-V3" \l "ZH-CN_TOPIC_0000001573928889__resource) | 设置文本最小显示字号。  需配合maxFontSize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。 |
| maxFontSize | number | string | [Resource](https://developer.harmonyos.com/cn/docs/documentation/doc-references-V3/ts-types-0000001477981241-V3" \l "ZH-CN_TOPIC_0000001573928889__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](https://developer.harmonyos.com/cn/docs/documentation/doc-references-V3/ts-appendix-enums-0000001478061741-V3" \l "ZH-CN_TOPIC_0000001574248789__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](https://developer.harmonyos.com/cn/docs/documentation/doc-references-V3/ts-appendix-enums-0000001478061741-V3" \l "ZH-CN_TOPIC_0000001574248789__copyoptions9) | 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。 |

#### 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：



### 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。

#### 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：

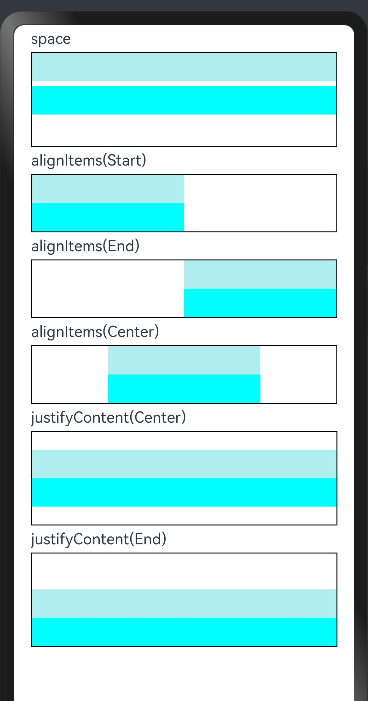
| 名称 | 参数类型 | 描述 |
| --- | --- | --- |
| alignItems | [HorizontalAlign](https://developer.harmonyos.com/cn/docs/documentation/doc-references-V3/ts-appendix-enums-0000001478061741-V3" \l "ZH-CN_TOPIC_0000001574248789__horizontalalign) | 设置子组件在水平方向上的对齐格式。  默认值：HorizontalAlign.Center  从API version 9开始，该接口支持在ArkTS卡片中使用。 |
| justifyContent8+ | [FlexAlign](https://developer.harmonyos.com/cn/docs/documentation/doc-references-V3/ts-appendix-enums-0000001478061741-V3" \l "ZH-CN_TOPIC_0000001574248789__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。 |

#### 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：



### 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。

#### 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：

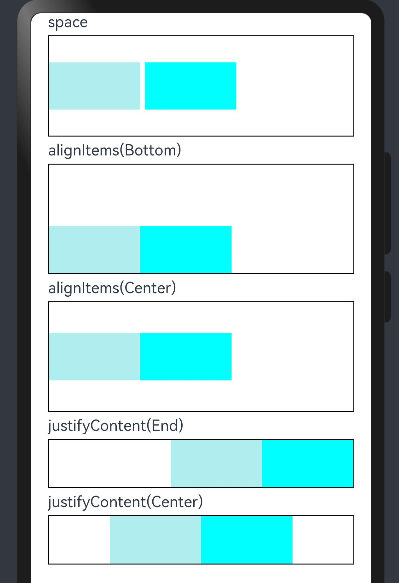
| **名称** | **参数类型** | **描述** |
| --- | --- | --- |
| alignItems | [VerticalAlign](https://developer.harmonyos.com/cn/docs/documentation/doc-references-V3/ts-appendix-enums-0000001478061741-V3" \l "ZH-CN_TOPIC_0000001574248789__verticalalign) | 设置子组件在垂直方向上的对齐格式。  默认值：VerticalAlign.Center  从API version 9开始，该接口支持在ArkTS卡片中使用。 |
| justifyContent8+ | [FlexAlign](https://developer.harmonyos.com/cn/docs/documentation/doc-references-V3/ts-appendix-enums-0000001478061741-V3" \l "ZH-CN_TOPIC_0000001574248789__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。 |

#### 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：



### 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。

#### [自定义组件的基本用法](https://developer.harmonyos.com/cn/docs/" \l "section1720153133218" \o "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!';  })  }  }  } |

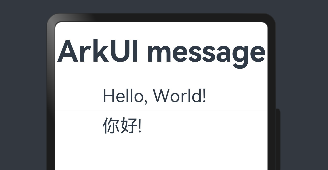
HelloComponent可以在其他自定义组件中的build()函数中多次创建，实现自定义组件的重用。

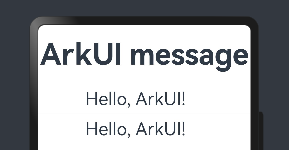
|  |
| --- |
| @Entry  @Component  struct ParentComponent {  build() {  Column() {  Text('ArkUI message')  HelloComponent({ message: 'Hello, World!' });  Divider()  HelloComponent({ message: '你好!' });  }  }  } |

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。





#### [自定义组件的基本结构](https://developer.harmonyos.com/cn/docs/" \l "section1430055924816" \o "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](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-localstorage-0000001524537149-V3)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。

#### [成员函数/变量](https://developer.harmonyos.com/cn/docs/" \l "section371262217494" \o "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](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-state-management-overview-0000001524537145-V3)。

#### [自定义组件的参数规定](https://developer.harmonyos.com/cn/docs/" \l "section4421142421915" \o "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](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-builder-0000001524176981-V3)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 })  }  }  } |

#### [build()函数](https://developer.harmonyos.com/cn/docs/" \l "section1150911733811" \o "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。**

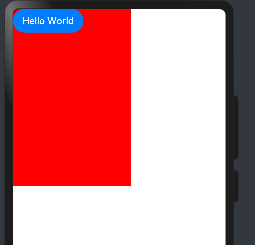
|  |
| --- |
| build() {  Column() {  // 反例：不允许使用表达式  (this.aVar > 10) ? Text('...') : Image('...')  }  } |

#### [自定义组件通用样式](https://developer.harmonyos.com/cn/docs/" \l "section1051122203016" \o "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。



### 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。

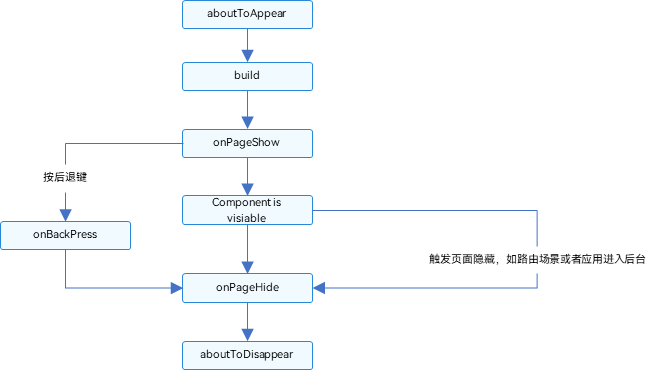
页面生命周期，即被@Entry装饰的组件生命周期，提供以下生命周期接口：

* [onPageShow](https://developer.harmonyos.com/cn/docs/documentation/doc-references-V3/arkts-custom-component-lifecycle-0000001482395076-V3" \l "ZH-CN_TOPIC_0000001523488850__onpageshow" \t "https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/_blank)：页面每次显示时触发。
* [onPageHide](https://developer.harmonyos.com/cn/docs/documentation/doc-references-V3/arkts-custom-component-lifecycle-0000001482395076-V3" \l "ZH-CN_TOPIC_0000001523488850__onpagehide" \t "https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/_blank)：页面每次隐藏时触发一次。
* [onBackPress](https://developer.harmonyos.com/cn/docs/documentation/doc-references-V3/arkts-custom-component-lifecycle-0000001482395076-V3" \l "ZH-CN_TOPIC_0000001523488850__onbackpress" \t "https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/_blank)：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](https://developer.harmonyos.com/cn/docs/documentation/doc-references-V3/arkts-custom-component-lifecycle-0000001482395076-V3" \l "ZH-CN_TOPIC_0000001523488850__abouttoappear" \t "https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/_blank)：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](https://developer.harmonyos.com/cn/docs/documentation/doc-references-V3/arkts-custom-component-lifecycle-0000001482395076-V3" \l "ZH-CN_TOPIC_0000001523488850__abouttodisappear" \t "https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/_blank)：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。

#### 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：
5. 状态变量 -> UI组件（包括ForEach和if）。
6. 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。

#### 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。

#### 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](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-link-0000001524297305-V3)、[@Prop](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-page-custom-components-lifecycle-0000001524296665-V3)、[@StorageLink](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-appstorage-0000001524417209-V3" \l "section84115526424)，将从[同步源](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-state-management-overview-0000001524537145-V3" \l "section127619262713)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。

### @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。

#### 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。

1. **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。

#### 参数传递规则

自定义构建函数的参数传递有[按值传递](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-builder-0000001524176981-V3" \l "section163841721135012)和[按引用传递](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-builder-0000001524176981-V3" \l "section1522464044212)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](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-link-0000001524297305-V3)。
* @Builder内UI语法遵循[UI语法规则](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-create-custom-components-0000001473537046-V3" \l "section1150911733811)。

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。

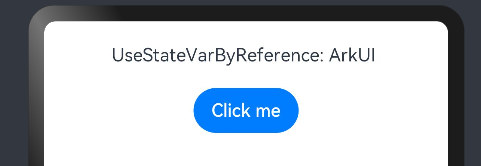
|  |
| --- |
| ABuilder( $$ : { paramA1: string, paramB1 : string } ); |

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：  


1. **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](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-builder-0000001524176981-V3" \l "section1522464044212)。

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。

## 状态管理

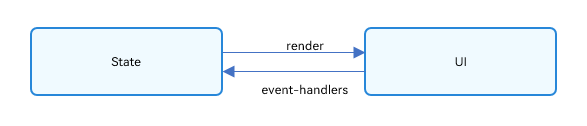
### 状态管理概述

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。

#### 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。

### 管理组件拥有的状态

#### @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。

1. **装饰器使用规则说明**

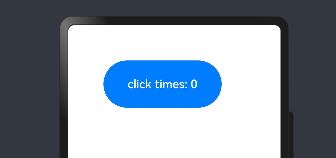
| **@State变量装饰器** | **说明** |
| --- | --- |
| 装饰器参数 | 无 |
| 同步类型 | 不与父组件中任何类型的变量同步。 |
| 允许装饰的变量类型 | Object、class、string、number、boolean、enum类型，以及这些类型的数组。  类型必须被指定。  不支持any，不支持简单类型和复杂类型的联合类型，不允许使用undefined和null。  说明：建议不要装饰Date类型，应用可能会产生异常行为。不支持Length、ResourceStr、ResourceColor类型，Length、ResourceStr、ResourceColor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。 |

1. **使用场景**
2. **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;  })  } } |

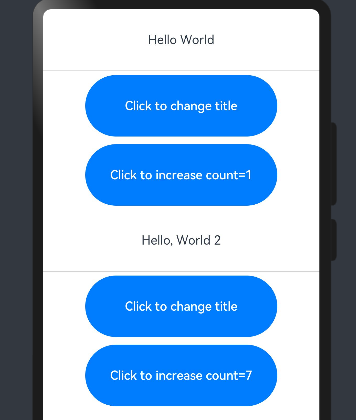
以上代码预览如下，每当点击按钮，都会自动加1。  


1. **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; |

1. 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 }) |

#### @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。

1. **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。

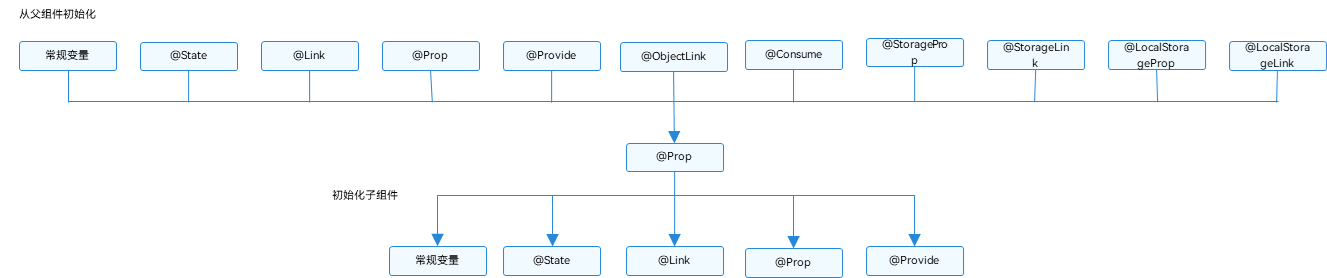
1. **装饰器使用规则说明**

| **@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](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-state-management-overview-0000001524537145-V3" \l "section127619262713)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 : { propA: S }。 |
| 被装饰变量的初始值 | 允许本地初始化。 |

1. **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：



1. **观察变化和行为表现**
2. **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。

1. **框架行为**

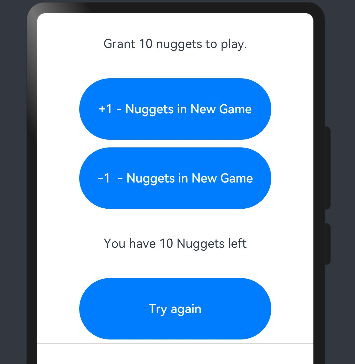
要理解@Prop变量值初始化和更新机制，有必要了解父组件和拥有@Prop变量的子组件初始渲染和更新流程。

1. 初始渲染：
2. 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；
3. 初始化子组件@Prop装饰的变量。
4. 更新：
5. 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；
6. 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。
7. **使用场景**
8. **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()  }  } } |

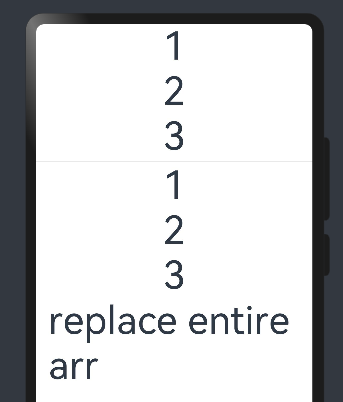
在上面的示例中，预览图如下：  


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。
6. **父组件@State数组项到子组件@Prop简单数据类型同步**

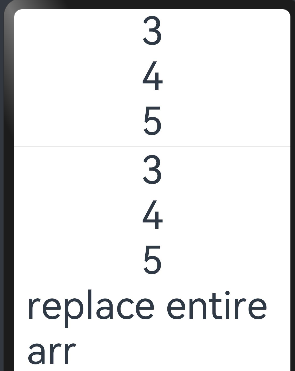
父组件中@State如果装饰的数组，其数组项也可以初始化@Prop。以下示例中父组件Index中@State装饰的数组arr，将其数组项初始化子组件Child中@Prop装饰的value。

|  |
| --- |
| @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：

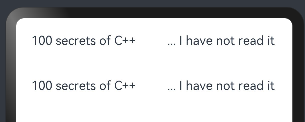


1. **从父组件中的@State类对象属性到@Prop简单类型的同步**

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。

1. **@Prop本地初始化不和父组件同步**

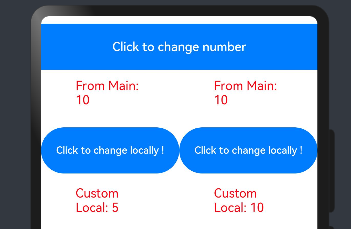
为了支持@Component装饰的组件复用场景，@Prop支持本地初始化，这样可以让@Prop是否与父组件建立同步关系变得可选。当且仅当@Prop有本地初始化时，从父组件向子组件传递@Prop的数据源才是可选的。

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。

#### @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。

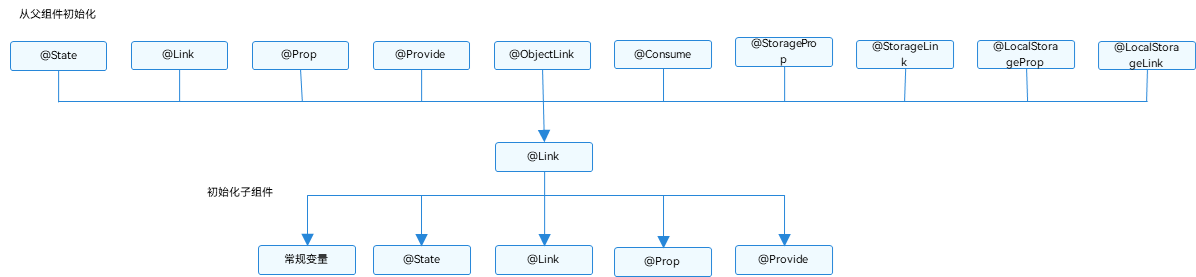
1. **装饰器使用规则说明**

| **@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。  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。 |

1. **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：



1. **观察变化和行为表现**
2. **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。

1. **框架行为**

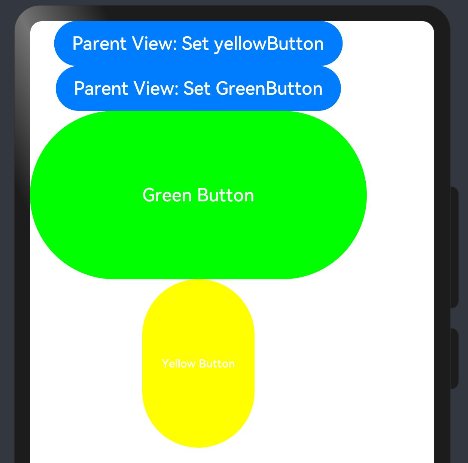
@Link装饰的变量和其所属的自定义组件共享生命周期。为了了解@Link变量初始化和更新机制，有必要先了解父组件和拥有@Link变量的子组件的关系，初始渲染和双向更新的流程（以父组件为@State为例）。

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：
2. 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）。
3. 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。
4. @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：
5. 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）。
6. 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。
7. @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）：
8. @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。
9. 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。
10. **使用场景**
11. **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。

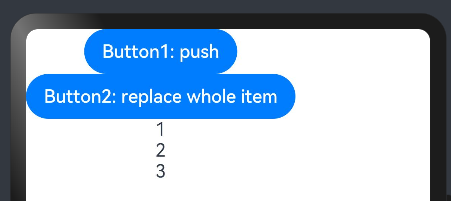
|  |
| --- |
| class GreenButtonState {  width: number = 0;  //构造方法  constructor(width: number) {  this.width = width;  } } @Component struct GreenButton {  @Link greenButtonState: GreenButtonState;  build() {  Button('Green Button')  .width(this.greenButtonState.width)  .height(150.0)  .backgroundColor('#00ff00')  .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。



1. **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。

## 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。

### 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。

### 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。

### 使用场景

#### 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：

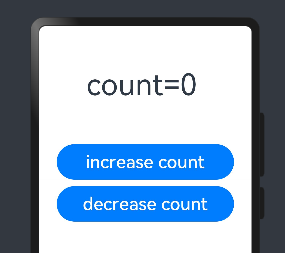
1. 删除所有以前渲染的（早期分支的）组件。
2. 执行新分支的构造函数，将生成的子组件添加到其父组件中。

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：



#### 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。

#### 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。

## 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](https://developer.harmonyos.com/cn/docs/documentation/doc-references-V3/ts-container-list-0000001477981213-V3" \t "https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/_blank)。从API version 9开始，该接口支持在ArkTS卡片中使用。

### 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**：  - 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。

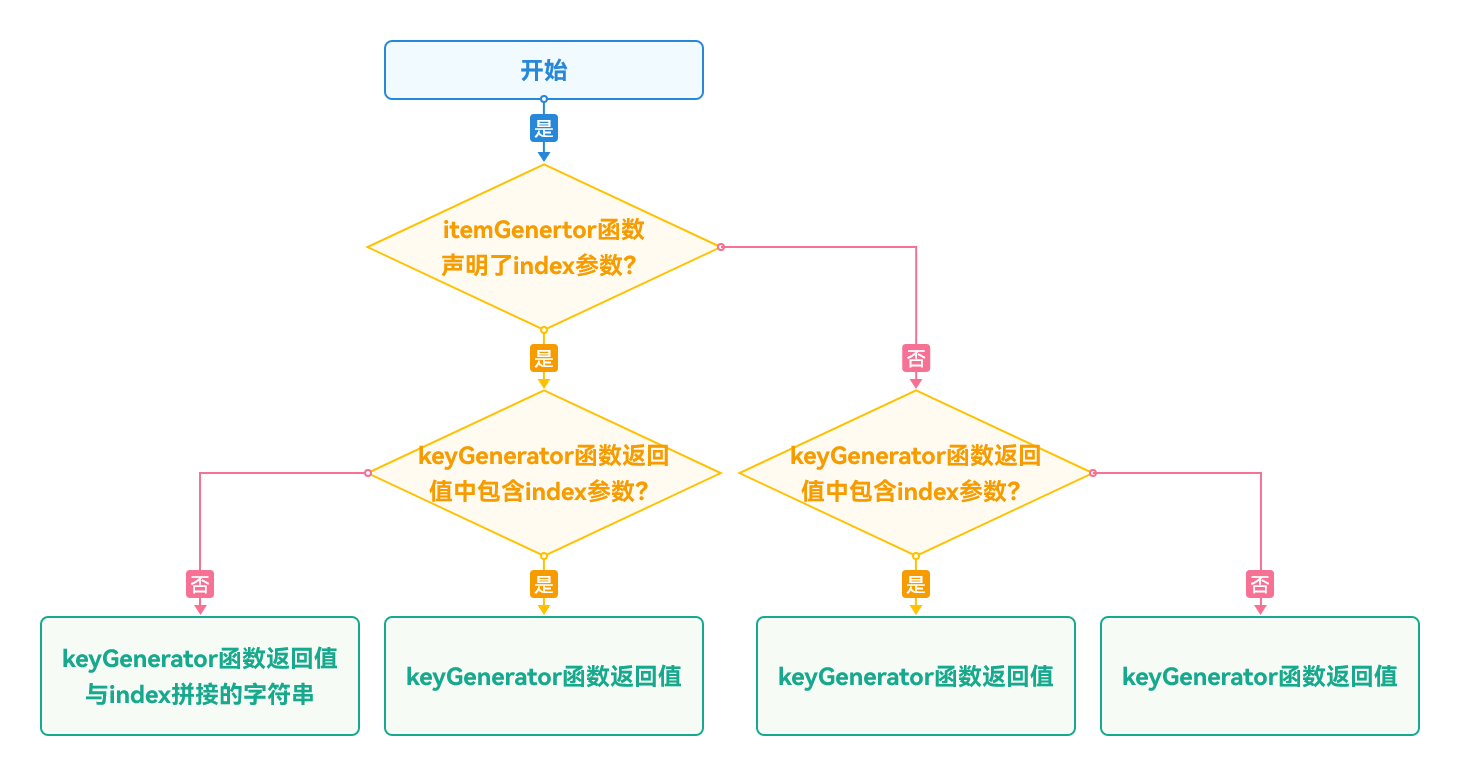
### 键值生成规则

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

### 组件创建规则

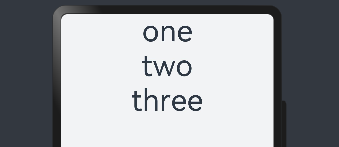
在确定键值生成规则后，ForEach的第二个参数itemGenerator函数会根据键值生成规则为数据源的每个数组项创建组件。组件的创建包括两种情况：[ForEach首次渲染](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-rendering-control-foreach-0000001524537153-V3" \l "section159mcpsimp)和[ForEach非首次渲染](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-rendering-control-foreach-0000001524537153-V3" \l "section169mcpsimp)。

#### 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:

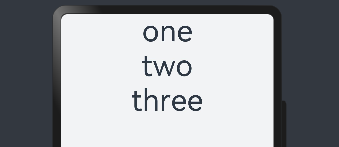


在上述代码中，键值生成规则是keyGenerator函数的返回值item。在ForEach渲染循环时，为数据源数组项依次生成键值one、two和three，并创建对应的ChildItem组件渲染到界面上。

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。

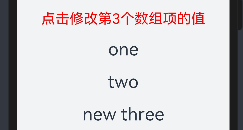
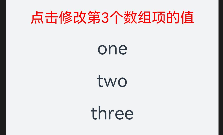


在该示例中，最终键值生成规则为item。当ForEach遍历数据源simpleList，遍历到索引为1的two时，按照最终键值生成规则生成键值为two的组件并进行标记。当遍历到索引为3的two时，按照最终键值生成规则当前项的键值也为two，此时不再创建新的组件。

#### 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。

### 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](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-rendering-control-foreach-0000001524537153-V3" \l "section182mcpsimp)、[数据源数组项发生变化](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-rendering-control-foreach-0000001524537153-V3" \l "section189mcpsimp)（如插入、删除操作）、[数据源数组项子属性变化](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-rendering-control-foreach-0000001524537153-V3" \l "section15662921144118)。

#### 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。

### 使用建议

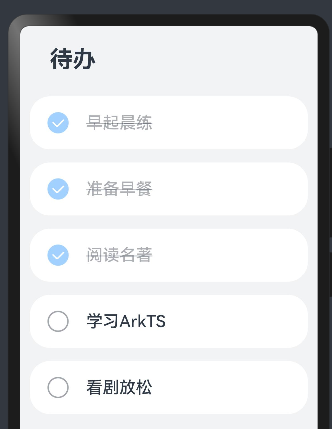
* 尽量避免在最终的键值生成规则中包含数据项索引index，以防止出现[渲染结果非预期](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-rendering-control-foreach-0000001524537153-V3" \l "section223mcpsimp)和[渲染性能降低](https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V3/arkts-rendering-control-foreach-0000001524537153-V3" \l "section234mcpsimp)。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。

## 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：



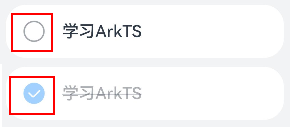
以上应用界面是由一个个页面组成，可以通过ArkUI框架声明式UI进行构建，声明式UI构建页面的过程，其实是组合组件的过程，声明式UI的思想，主要体现在两个方面：

* 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。

### 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。



### 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。

### 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 |

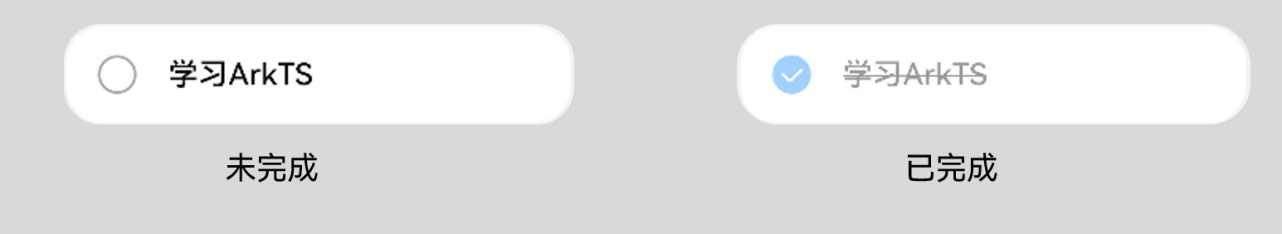
### 自定义组件-ToDoItem

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。



1. 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。
2. 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。
3. 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。

## 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：



### 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。



### 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;  } } |

### 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。  }  } |

### 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。  } } |

### 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。  } } |

### 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。  } } |

## 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

1. **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

1. **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

1. **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