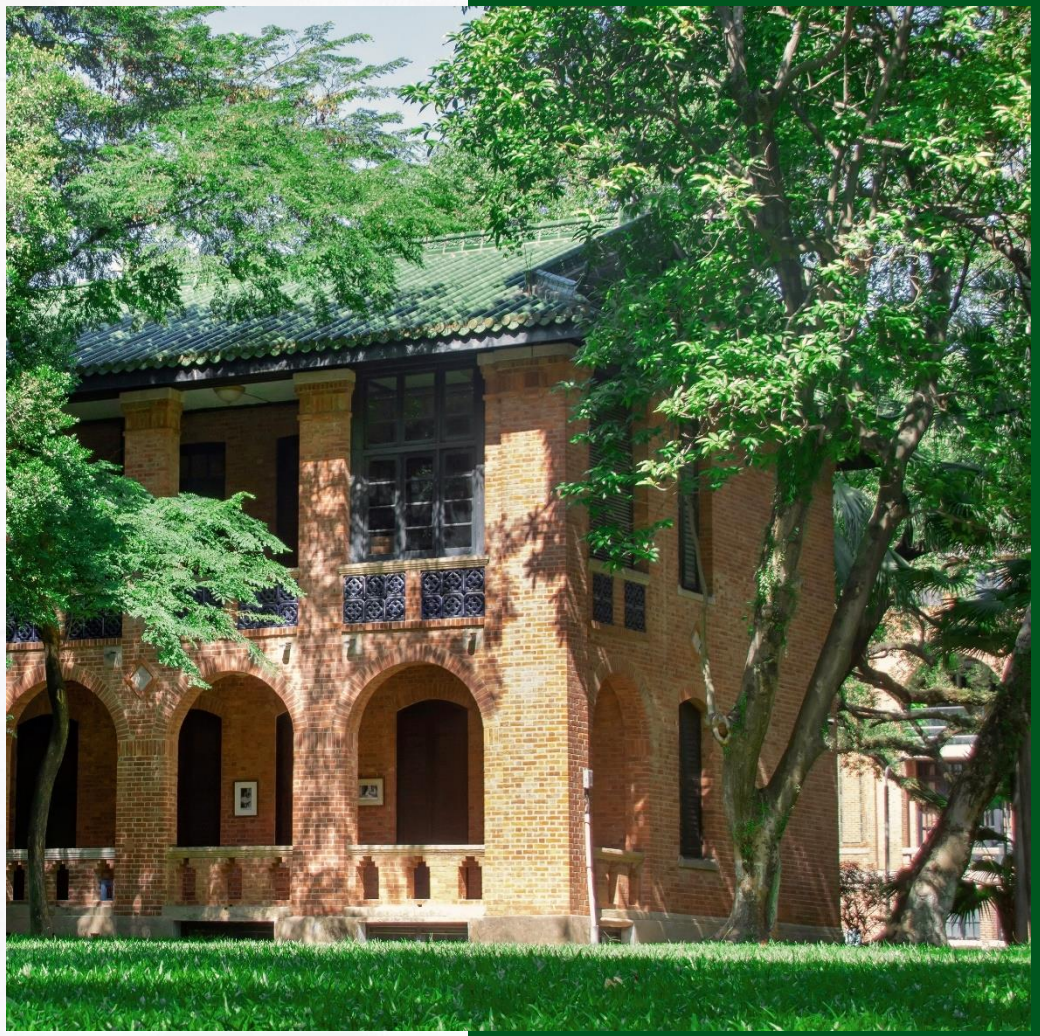
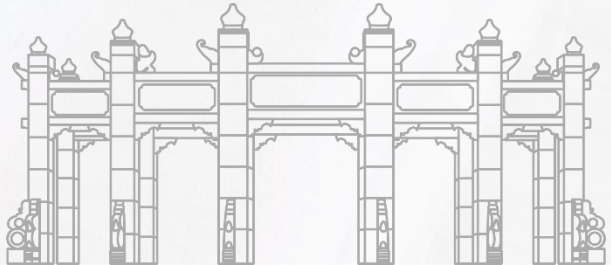
中山大學
SUN YAT-SEN UNIVERSITY

鸿蒙操作系统应用开发基础





应用程序框架

1. 应用程序框架是一种编程框架，它用于简化应用程序的开发过程，提高代码的可重用性和可维护性。可以帮助开发人员更快速、更高效的开发应用程序。
2. 应用程序框架连接开发者和用户的桥梁。
   - 它为用户提供应用内的交互、应用间的交互以及应用的跨设备流转。
   - 它为开发者提供应用进程管理、应用生命周期调度、应用组件生命周期调度、应用上下文环境、系统环境监听等相关能力。

图10-1 应用程序框架

HarmonyOS应用模型

1. 应用模型是系统为开发者提供的应用程序所需能力的抽象提炼，它提供了应用程序必备的组件和运行机制。有了应用模型，开发者可以基于一套统一的模型进行应用开发，使应用开发更简单、高效。
2. 应用模型的构成要素包括：
   - 应用组件：是应用的基本组成单位，是应用的运行入口。用户启动、使用和退出应用过程中，应用组件会在不同的状态间切换，这些状态称为应用组件的生命周期。
   - 应用进程模型：定义应用进程的创建和销毁方式，以及进程间的通信方式。
   - 应用线程模型：定义应用进程内线程的创建和销毁方式、主线程和UI线程的创建方式、线程间的通信方式。
   - 应用任务管理模型（仅对系统应用开放）：定义任务的创建和销毁方式，以及任务与组件间的关系。
   - 应用配置文件：包含应用配置信息、应用组件信息、权限信息、开发者自定义信息等。
3. OpenHarmony先后提供了两种应用模型：
   - **FA模型**：API 7开始支持的模型，已经不再主推。
   - **Stage模型**：API 9开始新增的模型，是目前主推且会长期演进的模型。在该模型中，由于提供了AbilityStage、WindowStage等类作为应用组件和Window窗口的“舞台”，因此称这种应用模型为Stage模型。

Ability

1. **Ability**，官方中文翻译为“元能力”。在HarmonyOS中，它是系统调度应用的最小单元（包括本地调度和分布式调度），是能够完成一个独立功能的实体对象，一个APP由一个或多个Ability组成。
2. Ability与Task（线程）之间的关系：
   - Ability的生命周期与Task的生命周期非常相似。
   - 在OS层面，调度的对象就是Task；而在应用开发领域，调度的对象变成了Ability。
   - 一般情况下，一个Ability实例都对应一个Task。
   - Ability是操作系统提供的更高一层的封装，借助这层封装，可以简化应用的开发过程；另一方面保证了应用的规范化。
   - 在系统服务层面，不存在Ability，都是以Task的方式提供各种服务，但在应用层面，一切都是通过Ability进行。

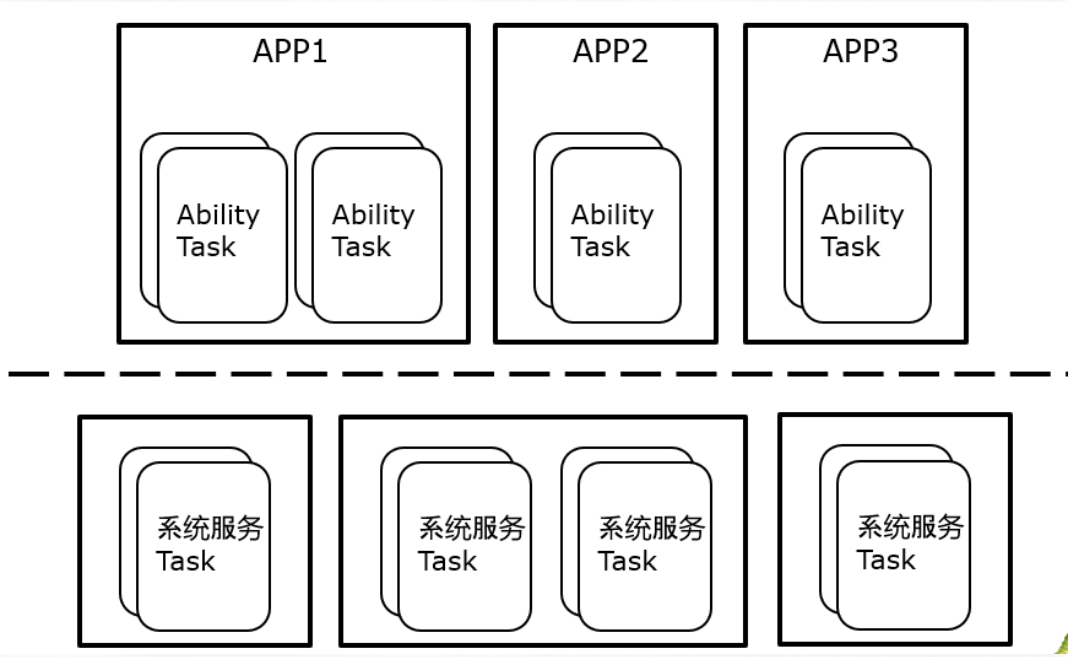


图10-2 Ability与Task

FA模型

1. 在OpenHarmony中，Ability框架模型结构具有两种形态，FA和Stage。
2. FA模型的应用组件有：
   - **PageAbility**：包含UI，提供展示UI的能力，主要用于与用户交互。
   - **ServiceAbility**：主要用于后台运行任务（如执行音乐播放、文件下载等），不提供用户交互界面。ServiceAbility可由其他应用或PageAbility启动，即使用户切换到其他应用，ServiceAbility仍将在后台继续运行。
   - **DataAbility**：提供数据分享的能力，提供其他Ability进行数据的增删查服务，无UI。使用DataAbility有助于应用管理其自身和其他应用存储数据的访问，并提供与其他应用共享数据的方法。DataAbility既可用于同设备不同应用的数据共享，也支持跨设备不同应用的数据共享。

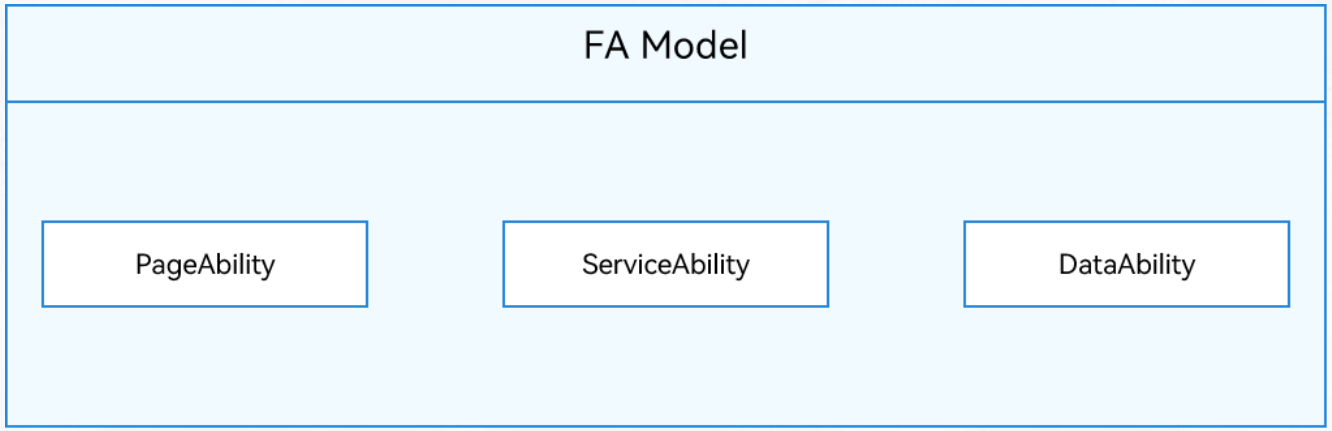


图10-3 FA模型

FA模型

PageAbility

1. PageAbility生命周期是PageAbility被调度到INACTIVE、ACTIVE、BACKGROUND等各个状态的统称。
   - **UNINITIALIZED**：未初始状态，为临时状态，PageAbility被创建后会由UNINITIALIZED状态进入INITIAL状态。
   - **INITIAL**：初始化状态，也表示停止状态，表示当前PageAbility未运行，PageAbility被启动后由INITIAL态进入INACTIVE状态。
   - **INACTIVE**：失去焦点状态，表示当前窗口已显示但是无焦点状态。
   - **ACTIVE**：前台激活状态，表示当前窗口已显示，并获取焦点。
   - **BACKGROUND**：后台状态，表示当前PageAbility退到后台，PageAbility在被销毁后由BACKGROUND状态进入INITIAL状态，或者重新被激活后由BACKGROUND状态进入ACTIVE状态。

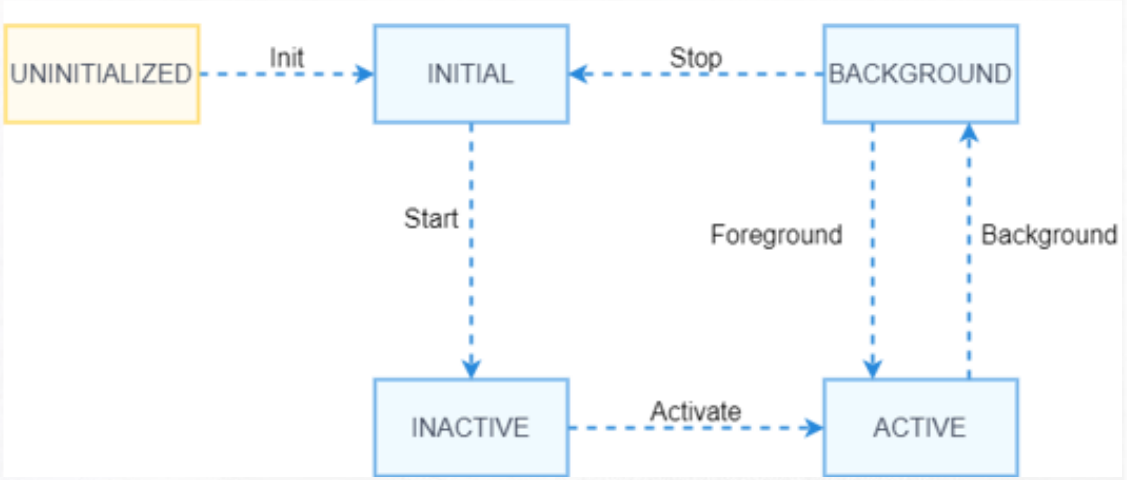


图10-4 PageAbility生命周期

FA模型

1. FA模型的进程模型中，主要有两类进程：主进程和渲染进程。
2. 应用中（同一包名）的所有PageAbility、ServiceAbility、DataAbility、FormAbility运行在同一个独立进程中，即图中绿色部分的“Main Process”。
3. WebView拥有独立的渲染进程，即图中黄色部分的“Render Process”。
4. 主要通过公共事件机制实现进程间的通信。公共事件机制多用于一对多的通信场景，公共事件发布者可能存在多个订阅者同时接收事件。

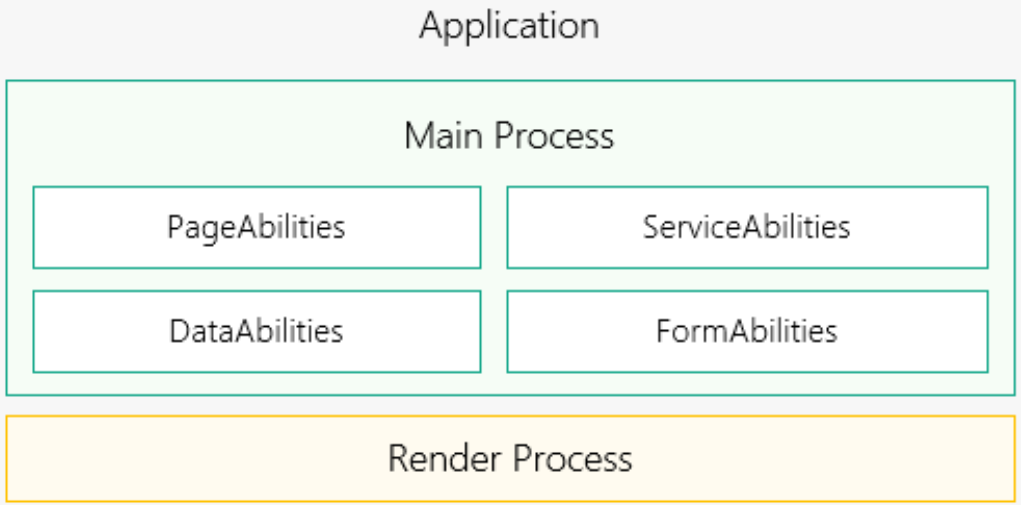


图10-5 FA模型的进程模型

FA模型

1. FA模型下的线程主要有如下三类：
   - **主线程**：负责管理其他线程。
   - **Ability线程**：每个Ability一个线程；输入事件分发；UI绘制；应用代码回调（事件处理，生命周期）；接收Worker发送的消息。接收Worker发送的消息。
   - **Worker线程**：执行耗时操作。
2. 不同的业务功能运行在不同的线程上，业务功能的交互就需要线程间通信。线程间通信目前主要有Emitter和Worker两种方式，其中Emitter主要适用于线程间的事件同步， Worker主要用于新开一个线程执行耗时任务。
3. FA模型下的任务管理
   - 每个PageAbility组件实例创建一个任务。
   - 任务会持久化存储，直到超过最大任务个数（根据产品配置自定义）或者用户主动删除任务。

Stage模型

Stage模型的设计思想

1. 为复杂应用而设计。
   - 多个应用组件共享同一个ArkTS引擎（运行ArkTS语言的虚拟机）实例，应用组件之间可以方便的共享对象和状态，同时减少复杂应用运行对内存的占用。
   - 采用面向对象的开发方式，使得复杂应用代码可读性高、易维护性好、可扩展性强。
2. 原生支持应用组件级的跨端迁移和多端协同。
   - Stage模型实现了应用组件与UI解耦：在跨端迁移场景下，系统在多设备的应用组件之间迁移数据/状态后，UI便可利用ArkUI的声明式特点，通过应用组件中保存的数据/状态恢复用户界面，便捷实现跨端迁移。在多端协同场景下，应用组件具备组件间通信的RPC调用能力，天然支持跨设备应用组件的交互。
3. 支持多设备和多窗口形态。
   - 应用组件管理和窗口管理在架构层面解耦，便于系统对应用组件进行裁剪（无屏设备可裁剪窗口）。便于系统扩展窗口形态。在多设备（如桌面设备和移动设备）上，应用组件可使用同一套生命周期。
4. 平衡应用能力和系统管控成本。
   - 重新定义应用能力的边界，平衡应用能力和系统管控成本。

Stage模型

Stage模型基本概念

1. UIAbility组件和ExtensionAbility组件

- **UIAbility组件**是一种包含UI的应用组件，主要用于和用户交互。例如，图库类应用可以在UIAbility组件中展示图片瀑布流，在用户选择某个图片后，在新的页面中展示图片的详细内容。UIAbility组件的生命周期只包含创建/销毁/前台/后台等状态，与显示相关的状态通过WindowStage的事件暴露给开发者。
- **ExtensionAbility组件**是一种面向特定场景的应用组件。开发者并不直接从ExtensionAbility组件派生，而是需要使用ExtensionAbility组件的派生类。ExtensionAbility组件的派生类实例由用户触发创建，并由系统管理生命周期。

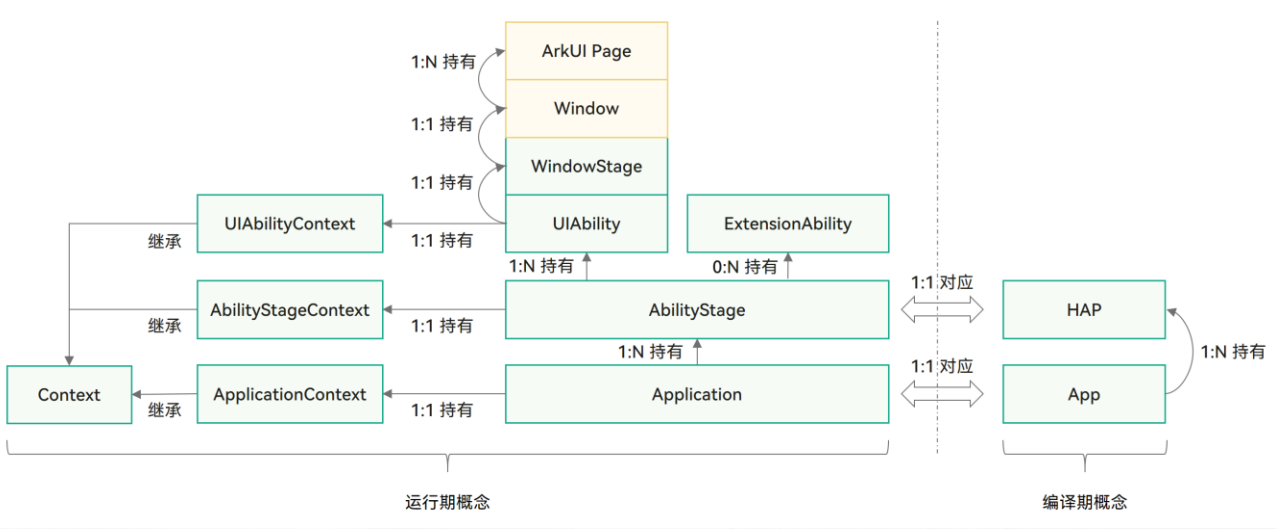


图10-6 Stage模型中的基本概念

Stage模型

2. WindowStage
   - 每个UIAbility实例都会与一个WindowStage类实例绑定，该类起到了应用进程内窗口管理器的作用。它包含一个主窗口。也就是说UIAbility实例通过WindowStage持有了一个主窗口，该主窗口为ArkUI提供了绘制区域。
3. Context
   - 在Stage模型上，Context及其派生类向开发者提供在运行期可以调用的各种资源和能力。UIAbility组件和ExtensionAbility组件的派生类都有各自不同的Context类，他们都继承自基类Context，但是各自根据所属组件提供不同的能力。
4. AbilityStage
   - 每个Entry类型或者Feature类型的HAP在运行期都有一个AbilityStage类实例，当HAP中的代码首次被加载到进程中的时候，系统会先创建AbilityStage实例。

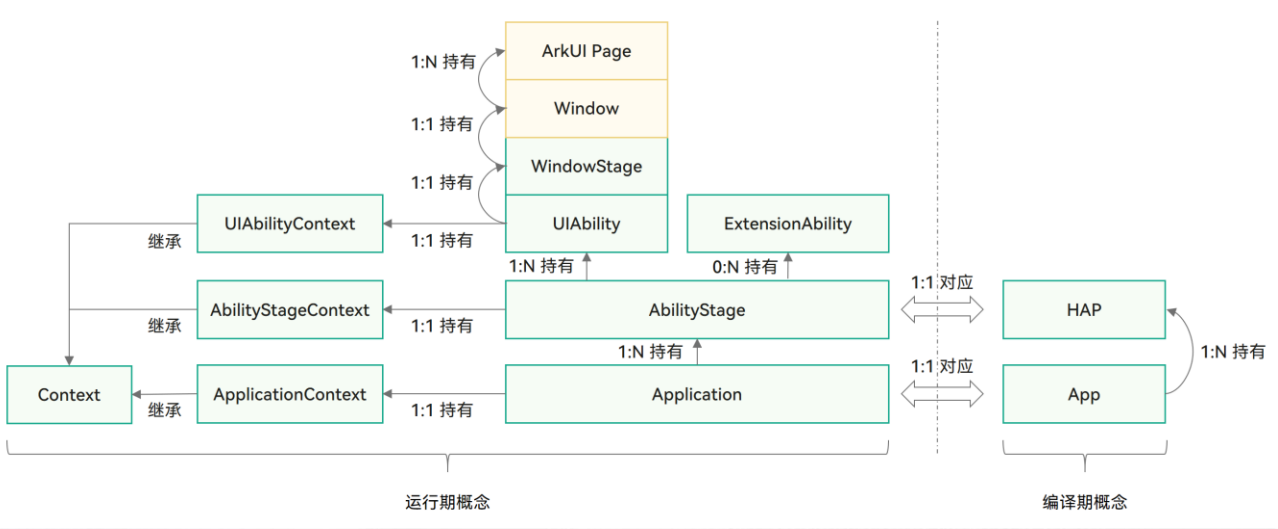


图10-6 Stage模型中的基本概念

Stage模型

UIAbility组件

1. UIAbility组件是系统调度的基本单元，为应用提供绘制界面的窗口。一个应用可以包含一个或多个UIAbility组件。例如，在支付应用中，可以将入口功能和收付款功能分别配置为独立的UIAbility。
2. 每一个UIAbility组件实例都会在最近任务列表中显示一个对应的任务。
3. 当用户打开、切换和返回到对应应用时，应用中的UIAbility实例会在其生命周期的不同状态之间转换。UIAbility的生命周期包括Create、Foreground、Background、Destroy四个状态：
   - **Create**：Create状态为在应用加载过程中，UIAbility实例创建完成时触发，系统会调用onCreate()回调。
   - **Foreground/ Background**：Foreground和Background状态分别在UIAbility实例切换至前台和切换至后台时触发，对应于onForeground()回调和onBackground()回调。
   - **Destroy**：Destroy状态在UIAbility实例销毁时触发。可以在onDestroy()回调中进行系统资源的释放、数据的保存等操作。

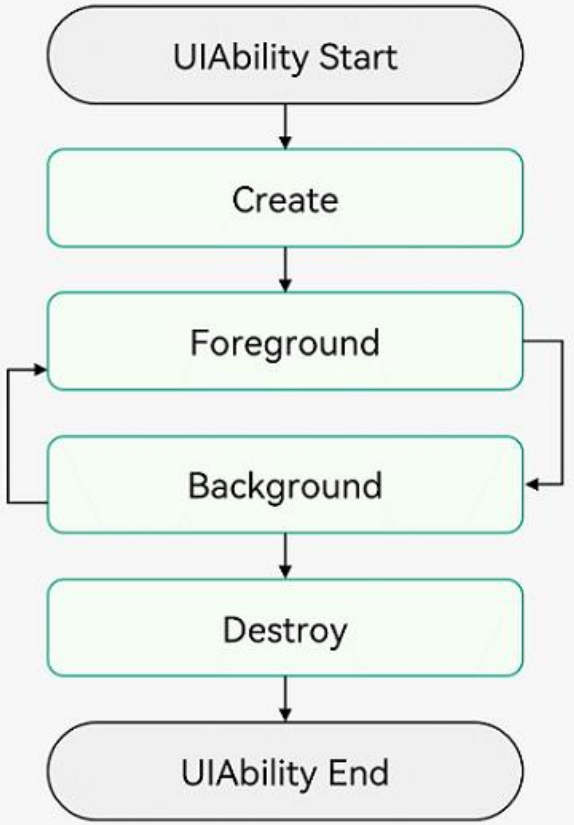


图10-7 UIAbility生命周期状态

Stage模型

UIAbility组件

1. UIAbility的启动模式是指UIAbility实例在启动时的不同呈现状态。针对不同的业务场景，系统提供了三种启动模式：
   - **singleton启动模式**：每次调用startAbility()方法时，如果应用进程中该类型的UIAbility实例已经存在，则复用系统中的UIAbility实例。系统中只存在唯一一个该UIAbility实例，即在最近任务列表中只存在一个该类型的UIAbility实例。
   - **multiton启动模式**：为多实例模式，每次调用startAbility()方法时，都会在应用进程中创建一个新的该类型UIAbility实例。即在最近任务列表中可以看到有多个该类型的UIAbility实例。这种情况下可以将UIAbility配置为multiton（多实例模式）。
   - **specified启动模式**：为指定实例模式，针对一些特殊场景使用（例如文档应用中每次新建文档希望都能新建一个文档实例，重复打开一个已保存的文档希望打开的都是同一个文档实例）。
2. UIAbility组件与UI的数据同步
   - 使**用EventHub**进行数据通信：在基类Context中提供了EventHub对象，可以通过发布订阅方式来实现事件的传递。在事件传递前，订阅者需要先进行订阅，当发布者发布事件时，订阅者将接收到事件并进行相应处理。
   - 使用**AppStorage/LocalStorage**进行数据同步：ArkUI提供了AppStorage和LocalStorage两种应用级别的状态管理方案，可用于实现应用级别和UIAbility级别的数据同步。

Stage模型

ExtensionAbility组件：ServiceExtensionAbility

1. ServiceExtensionAbility是SERVICE类型的ExtensionAbility组件，提供后台服务能力，通过ServiceExtensionContext提供了丰富的接口供外部使用。
2. ServiceExtensionAbility的生命周期。
   - **onCreate**：服务被首次创建时触发该回调，开发者可以在此进行一些初始化的操作，例如注册公共事件监听等。
   - **onRequest**：当另一个组件调用startServiceExtensionAbility()方法启动该服务组件时，触发该回调。
   - **onConnect**：当另一个组件调用connectServiceExtensionAbility()方法与该服务连接时，触发该回调。
   - **onDisconnect**：当最后一个连接断开时，将触发该回调。客户端死亡或者调用disconnectServiceExtensionAbility()方法使连接断开。
   - **onDestroy**：当不再使用服务且准备将其销毁该实例时，触发该回调。开发者可以在该回调中清理资源，如注销监听等

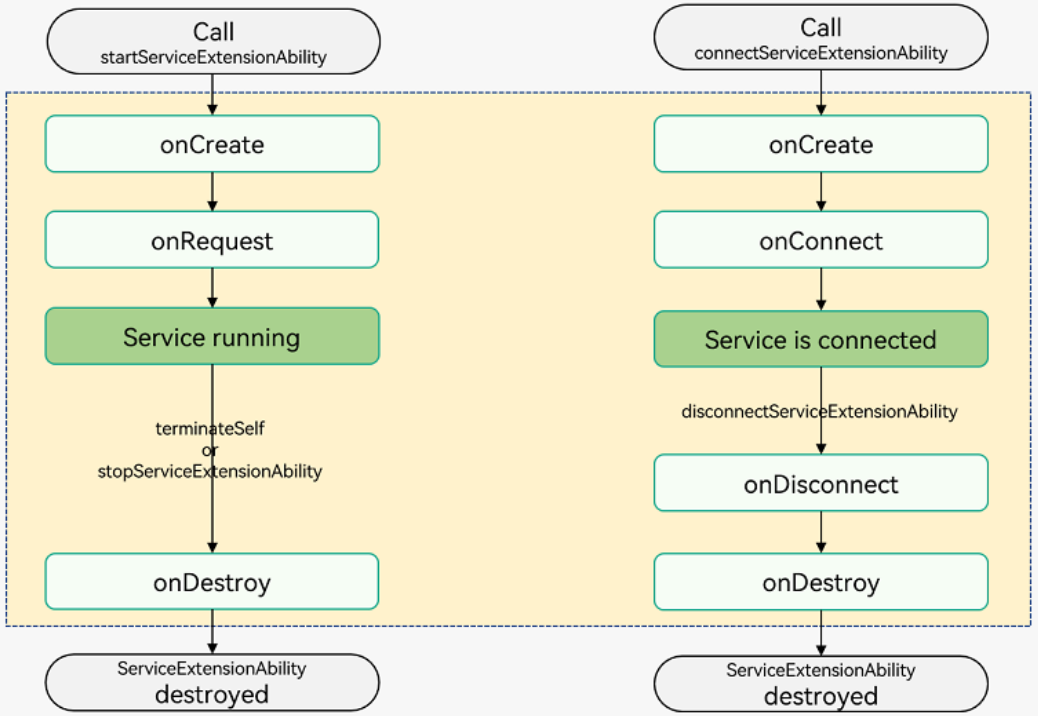


图10-8 ServiceExtensionAbility生命周期

Stage模型

ExtensionAbility组件：AccessibilityExtensionAbility

1. AccessibilityExtensionAbility为无障碍扩展服务框架，允许三方开发自己的扩展服务，提供在应用程序和扩展服务之间交换信息的标准机制，以便为各种障碍人群和障碍场景提供辅助能力，增强用户的无障碍使用体验。例如在使用微信时，利用辅助应用旁白，用户可以“听见”当前屏幕内容。
   - **Accessibility App**：开发者基于无障碍扩展服务框架扩展出来的辅助应用，如视障用户使用的读屏App。
   - **Tartget App**：被Accessibility App辅助的目标应用。
   - **AccessibilityAbilityManagerService（AAMS）**：无障碍扩展服务框架主服务，用于对Accessibility App生命周期进行管理，同时为Accessibility App和Target App提供信息交互的桥梁。
   - **AccessibilityAbility（AAkit）**：Accessibility App利用AAkit构建扩展服务Ability运行环境，并为Accessibility App提供可查询和操作Target App的接口，如查询节点信息、对节点执行点击/长按操作等。
   - **AccessibilitySystemAbilityClient（ASACkit）**：Target App通过ASACkit向AAMS发送无障碍事件，如内容变化事件等。

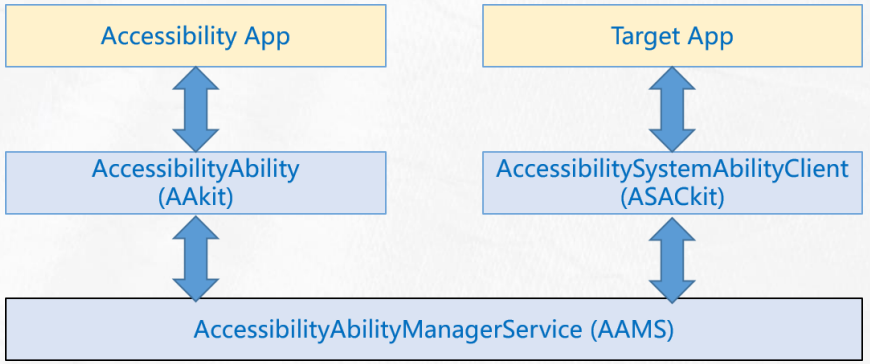


图10-9 AccessibilityExtensionAbility框架

Stage模型

AbilityStage组件容器

1. AbilityStage是一个Module级别的组件容器，应用的HAP在首次加载时会创建一个AbilityStage实例，可以对该Module进行初始化等操作。
2. AbilityStage与Module一一对应，即一个Module拥有一个AbilityStage。
3. AbilityStage拥有onCreate()生命周期回调和onAcceptWant()、onConfigurationUpdated()、onMemoryLevel()事件回调。
   - onCreate()生命周期回调：在开始加载对应Module的第一个UIAbility实例之前会先创建AbilityStage，并在AbilityStage创建完成之后执行其onCreate()生命周期回调。AbilityStage模块提供在Module加载的时候，通知开发者，可以在此进行该Module的初始化（如资源预加载，线程创建等）能力。
   - onAcceptWant()事件回调：UIAbility指定实例模式（specified）启动时候触发的事件回调，具体使用请参见UIAbility启动模式综述。
   - onConfigurationUpdated()事件回调：当系统全局配置发生变更时触发的事件，系统语言、深浅色等，配置项目前均定义在Configuration类中。
   - onMemoryLevel()事件回调：当系统调整内存时触发的事件。

Stage模型

应用上下文Context

1. Context是应用中对象的上下文，其提供了应用的一些基础信息，例如resourceManager（资源管理）、applicationInfo（当前应用信息）、dir（应用文件路径）、area（文件分区）等，以及应用的一些基本方法，例如createBundleContext()、getApplicationContext()等。
2. UIAbility组件和各种ExtensionAbility派生类组件都有各自不同的Context类。分别有基类Context、ApplicationContext、AbilityStageContext、UIAbilityContext、ExtensionContext、ServiceExtensionContext等Context

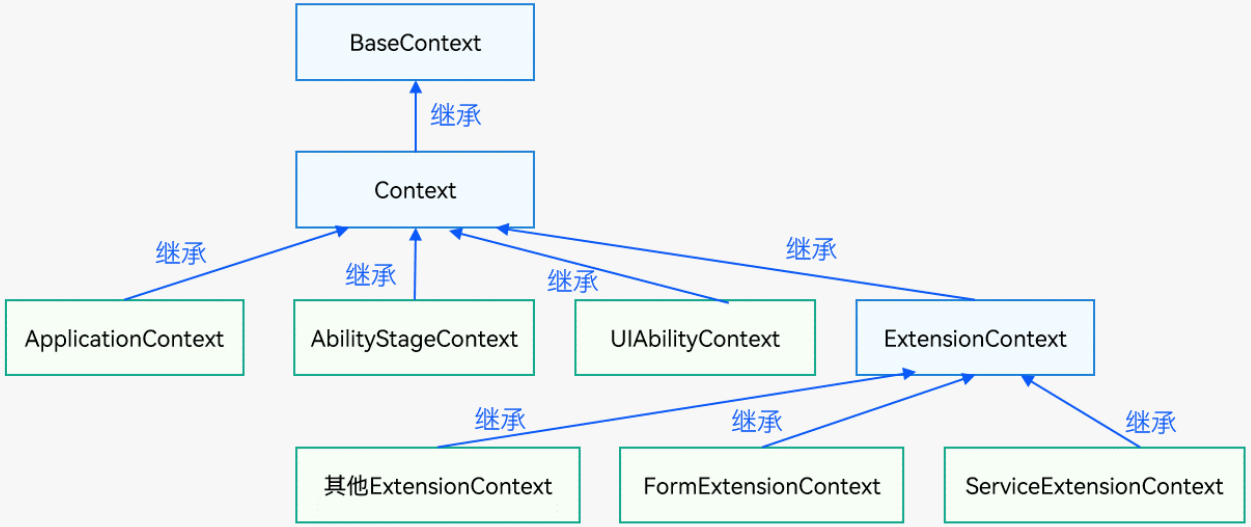


图10-10 各类Context的继承关系

Stage模型

Stage模型的进程模型

1. Stage模型的进程模型中，主要有三类进程：主进程、ExtensionAbility进程和渲染进程。
2. 应用中的所有UIAbility、ServiceExtensionAbility和DataShareExtensionAbility均是运行在同一个独立进程（主进程）中，如右图中绿色部分的“Main Process”。
3. 除ServiceExtensionAbility和DataShareExtensionAbility外，所有同一类型ExtensionAbility均是运行在一个独立进程中，如下图中蓝色部分。
4. 可以通过公共事件机制和后台服务机制来实现进程之间的通信。
   - 公共事件按发送方式可分为：无序公共事件、有序公共事件和粘性公共事件。
   - 后台服务机制：系统应用A实现了一个后台服务，三方应用B可以通过连接系统应用A的后台服务与其进行进程间通信。

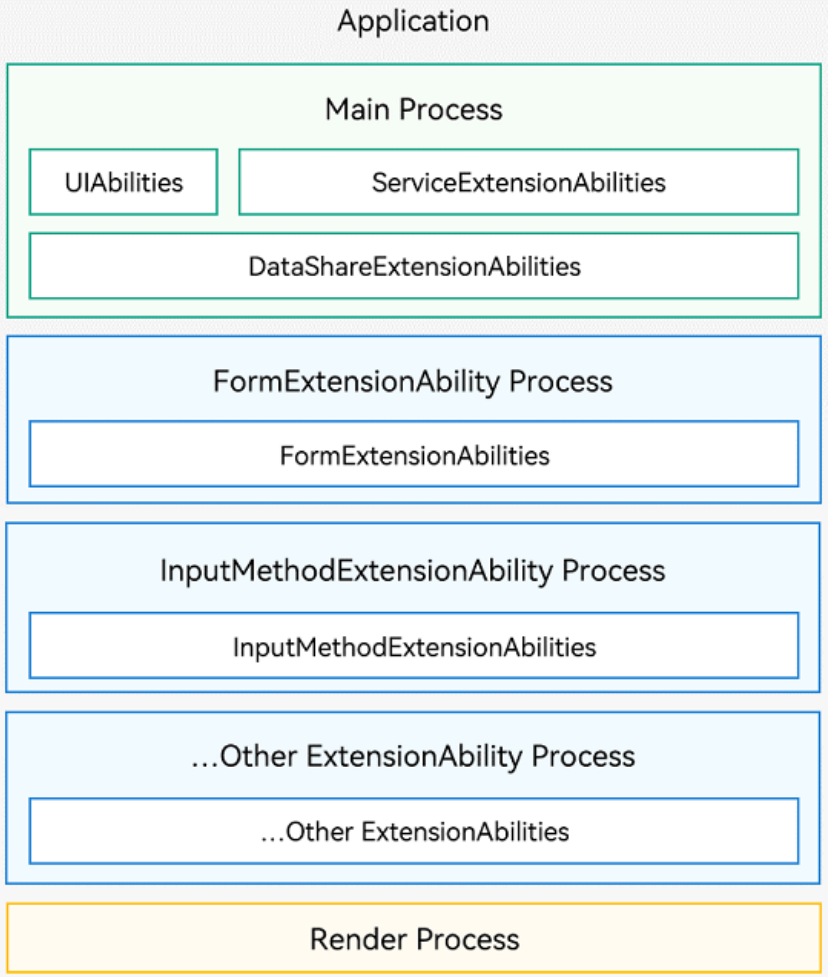


图10-11 Stage模型的进程模型

Stage模型

Stage模型的线程模型

1. Stage模型下的线程主要有如下三类：
   - **主线程**：管理主线程和其他线程的ArkTS引擎实例；输入事件分发；UI绘制；应用代码回调（事件处理，生命周期）；接收Worker发送的消息。
   - **TaskPool Worker线程**：用于执行耗时操作，支持设置调度优先级、负载均衡等功能。
   - **Worker线程**：用于执行耗时操作，支持线程间通信。
2. 同一线程中存在多个组件，例如UIAbility组件和UI组件都存在于主线程中。在Stage模型中目前主要使用EventHub进行数据通信。
   - EventHub提供了线程内发送和处理事件的能力，包括对事件订阅、取消订阅、触发事件等。

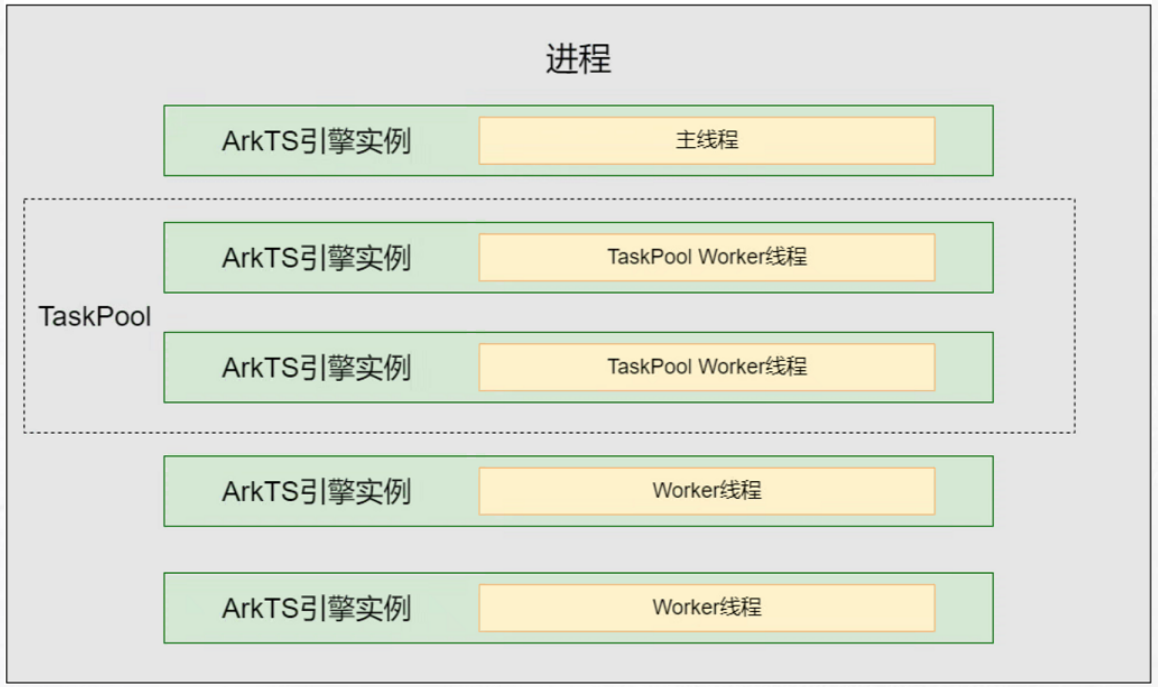


图10-12 Stage模型的线程模型

HarmonyOS应用开发的基本流程

1. 开发准备：
   - 下载 HUAWEI DevEco Studio，然后设置开发环境。
2. 开发应用：
   - DevEco Studio 集成了多个的典型场景模板，可以通过工程向导轻松的创建一个新工程。
   - 定义应用的UI、开发业务功能等编码工作。
   - 开发代码，通过查看文档找到调用的API接口。
3. 运行、调试和测试应用：
   - 应用开发完成后，您可以使用真机进行调试或者使用模拟器进行调试。
   - 还需要对应用进行测试，主要包括漏洞、隐私、兼容性、稳定性、性能等进行测试。
4. 发布应用：
   - HarmonyOS 应用开发一切就绪后，需要将应用发布至华为应用市场，以便应用市场对您的应用进行分发。

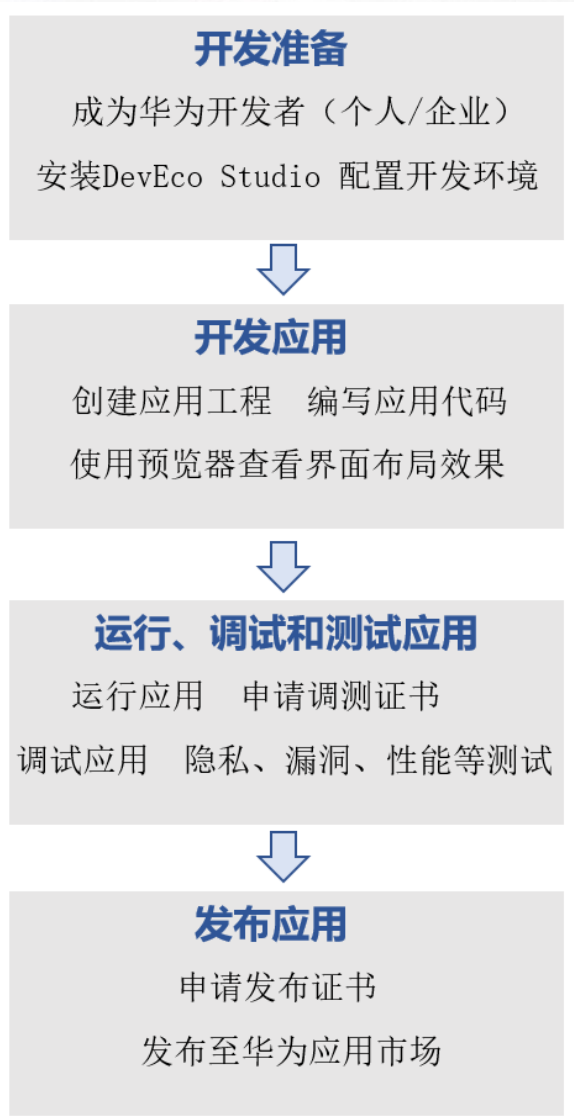


图10-13 鸿蒙应用开发流程

- **HarmonyOS应用开发工具**

1. HUAWEI DevEco Studio：是基于IntelliJ IDEA Community开源版本打造，为运行在HarmonyOS和OpenHarmony系统上的应用和服务提供一站式的开发平台。

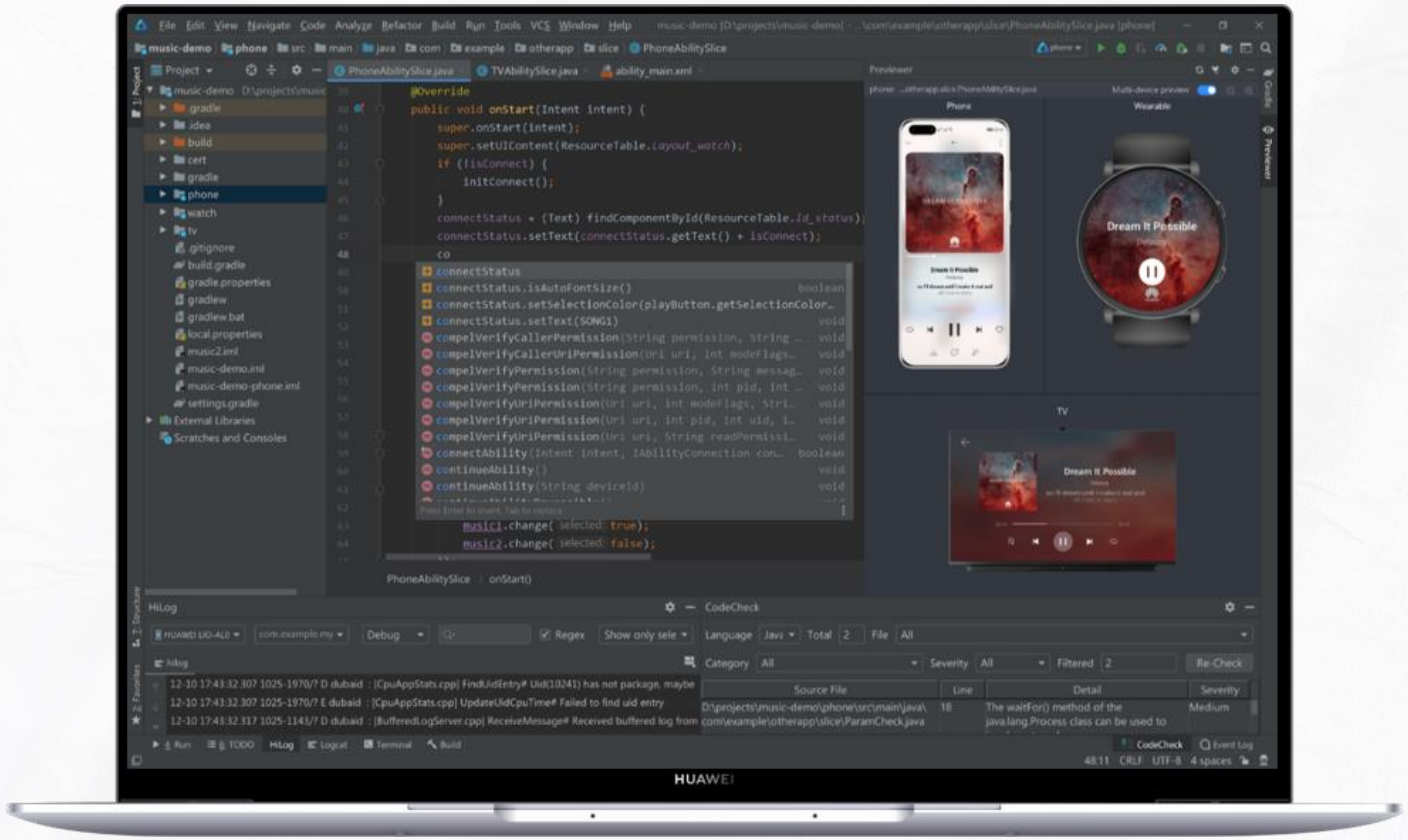


图10-14 HUAWEI DevEco Studio

HarmonyOS应用开发工具

1. HUAWEI DevEco Studio：作为一款开发工具，除了具有基本的代码开发、编译构建及调测等功能外，DevEco Studio还具有如下特点：

   - 高效智能代码编辑：支持ArkTS、JS、C/C++等语言的代码高亮、代码智能补齐、代码错误检查、代码自动跳转、代码格式化、代码查找等功能。
   - 低代码可视化开发：丰富的UI界面编辑能力，；同时支持卡片的零代码开发。
   - 多端双向实时预览：支持UI界面代码的双向预览、实时预览、动态预览等，便于快速查看代码运行效果。
   - 多端设备模拟仿真：提供HarmonyOS本地模拟器，支持手机等设备的模拟仿真，便捷获取调试环境。

应用开发语言ArkTS

1. ArkTS是OpenHarmony优选的应用高级开发语言。ArkTS提供了声明式UI范式、状态管理支持等相应的能力，让开发者可以以更简洁、更自然的方式开发应用。
2. 它在保持TypeScript基本语法风格的基础上，进一步通过规范强化静态检查和分析，使得在程序运行之前的开发期能检测更多错误，提升代码健壮性，并实现更好的运行性能。详见初识ArkTS语言。
3. 当前，在UI开发框架中，ArkTS主要扩展了如下能力：
   - **基本语法**：ArkTS定义了声明式UI描述、自定义组件和动态扩展UI元素的能力，再配合ArkUI开发框架中的系统组件及其相关的事件方法、属性方法等共同构成了UI开发的主体。
   - **状态管理**：ArkTS提供了多维度的状态管理机制。在UI开发框架中，与UI相关联的数据可以在组件内使用，也可以在不同组件层级间传递，比如父子组件之间、爷孙组件之间，还可以在应用全局范围内传递或跨设备传递。
   - **渲染控制**：ArkTS提供了渲染控制的能力。条件渲染可根据应用的不同状态，渲染对应状态下的UI内容。

应用开发语言ArkTS

4. ArkTS提供了标准内置对象，例如Array、Map、TypedArray、Math等，供开发者直接使用。另外，ArkTS也提供了语言基础类库，为应用开发者提供常用的基础能力。

   - 提供异步并发和多线程并发的能力。
   - 提供常见的容器类库增、删、改、查的能力。
   - 提供XML、URL、URI构造和解析的能力。
   - 提供常见的字符串和二进制数据处理的能力，以及控制台打印的相关能力。
   - 提供获取进程信息和操作进程的能力。

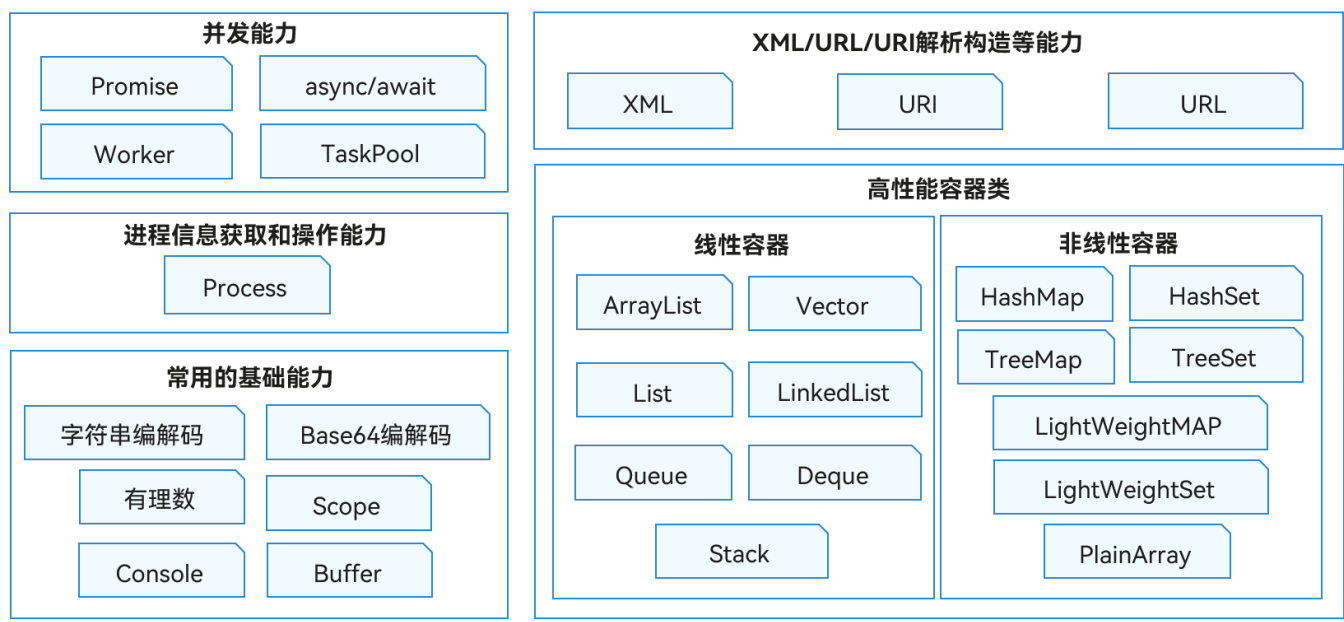


图10-15 ArkTS语言基础类库能力示意图

ArkTS的基本语法

示例：当开发者点击按钮时，文本内容从“Hello World”变为“Hello ArkUI”。

图10-16 示例效果图

```
@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 })
    }
  }
}
```

装饰器

自定义组件

UI描述

系统组件

事件方法

属性方法

图10-17 ArkTS的基本组成

ArkTS的基本语法

1. 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。如示例中入口组件@Entry、自定义组件@Component和组件中的状态变量@State。
2. UI描述：以声明式的方式来描述UI的结构，例如build()方法中的代码块。
3. 自定义组件：可复用的UI单元，可组合其他组件，如被@Component装饰的struct Hello。
4. 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。如示例中入口组件@Entry、自定义组件@Component和组件中的状态变量@State。
5. UI描述：以声明式的方式来描述UI的结构，例如build()方法中的代码块。
6. 自定义组件：可复用的UI单元，可组合其他组件，如被@Component装饰的struct Hello。

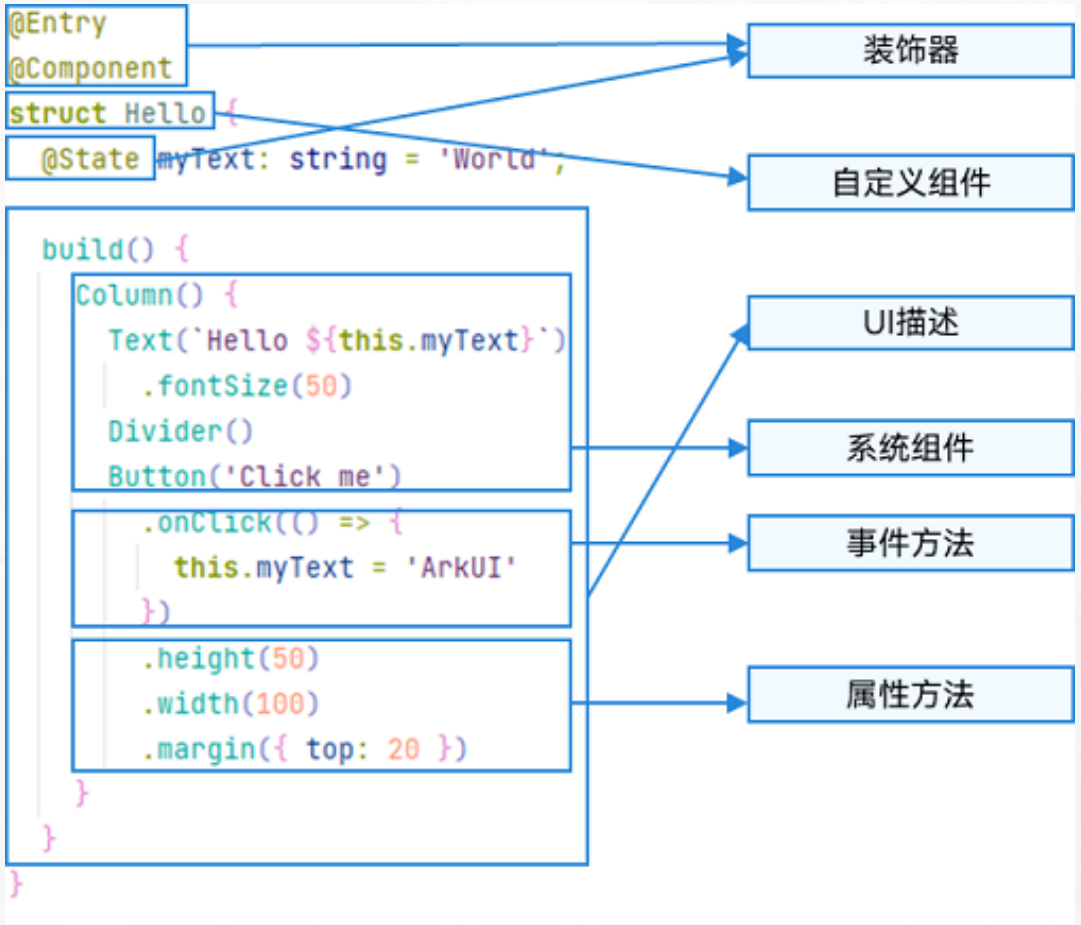


图10-17 ArkTS的基本组成

ArkTS的基本语法

声明式UI描述

1. 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。
2. 创建组件包含有参数和无参数两种方式。
3. 属性配置方法以“.”链式调用的方式配置系统组件的样式和其他属性。除了直接传递常量参数外，还可以传递变量或表达式。
4. 事件配置方法：使用箭头函数进行配置。如果使用组件的成员函数配置组件的事件方法，需要bind this。如果使用声明的箭头函数，则可以直接调用，不需要bind this。

```
// 有参形式
Text('test')
// 无参数形式
Text()

// 配置组件的多个属性
Image('test.jpg')
  .alt('error.jpg')
  .width(100)
  .height(100)

// 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))

// 使用声明的箭头函数配置组件的事件方法
fn = () => {
  console.info(`counter: ${this.counter}`)
  this.counter++
}
...
Button('add counter')
  .onClick(this.fn)
```

ArkTS的状态管理

1. 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。
2. 上面的示例中，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”。
3. 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。
4. 而常规变量是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。
5. 在上述例子中，当点击事件发生后，会引起状态变量myText的变化，进一步引发UI的渲染刷新。

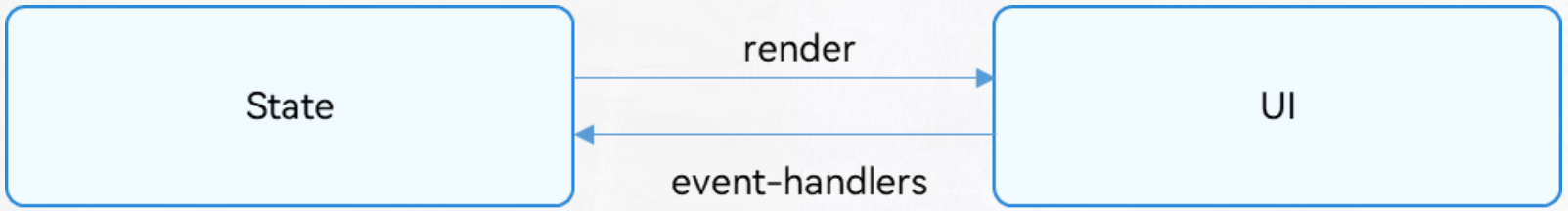


图10-18 状态与UI之间的关系

ArkTS的状态管理

1. 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。
2. 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。
3. 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。
4. 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。
5. 初始化：
   - 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。
   - 初始化子组件：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。
   - 本地初始化：在变量声明的时候赋值，作为变量的默认值。

```
@Component
struct MyComponent {
  // 状态变量
  @State count: number = 0;
  // 常规变量
  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
      // "count: 1, increaseBy: 2" 为数据源
      MyComponent({ count: 1, increaseBy: 2 })
    }
  }
}
```

ArkTS的状态管理

装饰器

1. ArkUI提供了多种装饰器，通过使用这些装饰器，状态变量不仅可以观察在组件内的改变，还可以在不同组件层级间传递，比如父子组件、跨组件层级，也可以观察全局范围内的变化。
2. 据状态变量的影响范围，将所有的装饰器可以分为：
   - **管理组件拥有状态的装饰器**：组件级别的状态管理，可以观察组件内变化，和不同组件层级的变化，但需要唯一观察同一个组件树上，即同一个页面内。
   - **管理应用拥有状态的装饰器**：应用级别的状态管理，可以观察不同页面，甚至不同UIAbility的状态变化，是应用内全局的状态管理。
3. 从数据的传递形式和同步类型层面看，装饰器可分为：
   - 只读的单向传递；
   - 可变更的双向传递。

ArkTS的状态管理

装饰器

4. 管理组件拥有的状态与管理应用拥有的状态的装饰器：

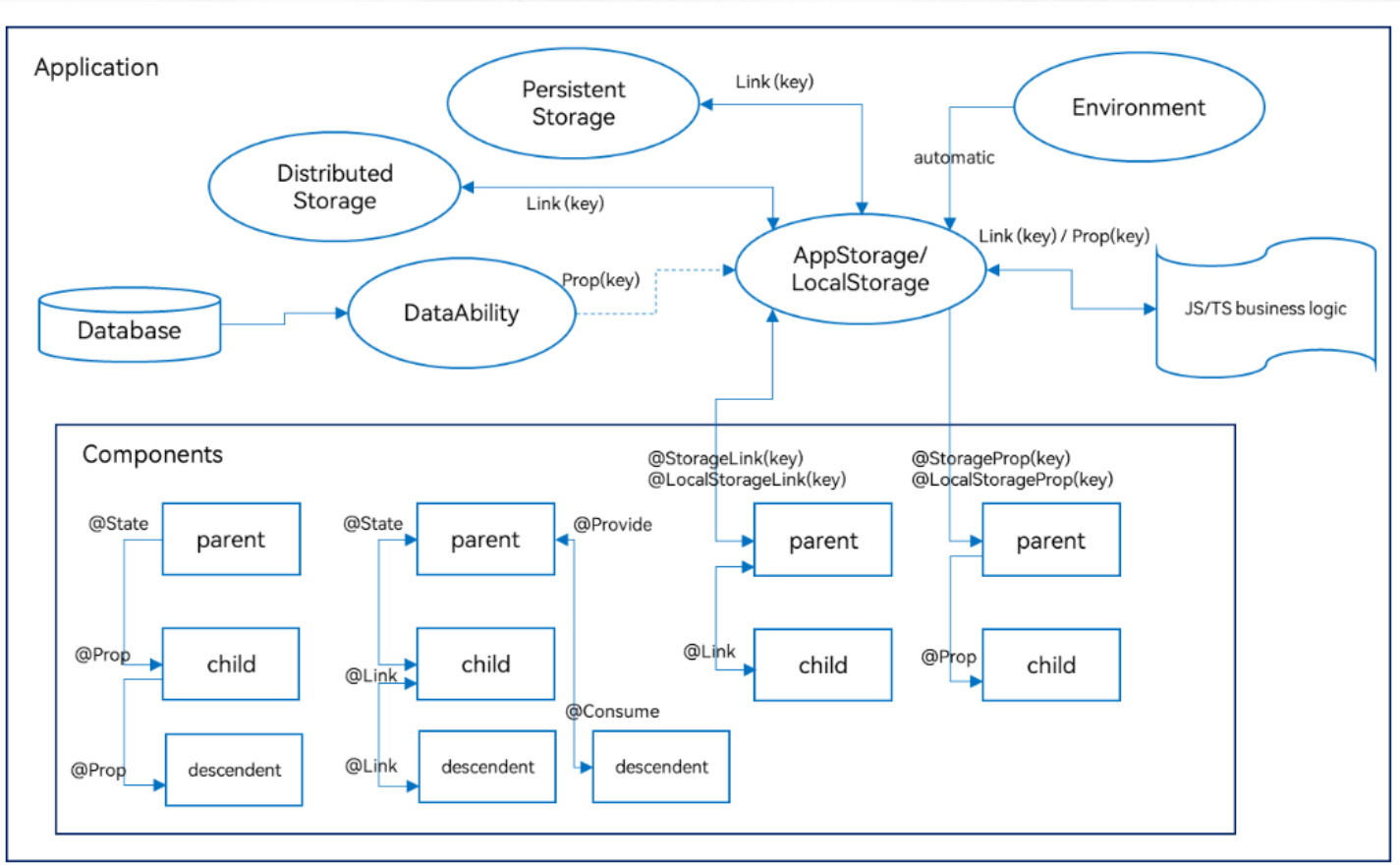


图10-19 装饰器类别

ArkTS的状态管理

装饰器

5. 组件级别状态管理的装饰器：

- **@State**：@State装饰的变量拥有其所属组件的状态，可以作为其子组件单向和双向同步的数据源。当其数值改变时，会引起相关组件的渲染刷新。
- **@Prop**：@Prop装饰的变量可以和父组件建立单向同步关系，@Prop装饰的变量是可变的，但修改不会同步回父组件。
- **@Link**：@Link装饰的变量可以和父组件建立双向同步关系，子组件中@Link装饰变量的修改会同步给父组件中建立双向数据绑定的数据源，父组件的更新也会同步给@Link装饰的变量。
- **@Provide/@Consume**：@Provide/@Consume装饰的变量用于跨组件层级（多层组件）同步状态变量，可以不需要通过参数命名机制传递，通过alias（别名）或者属性名绑定。
- **@Observed**：@Observed装饰class，需要观察多层嵌套场景的class需要被@Observed装饰。单独使用@Observed没有任何作用，需要和@ObjectLink、@Prop联用。
- **@ObjectLink**：@ObjectLink装饰的变量接收@Observed装饰的class的实例，应用于观察多层嵌套场景，和父组件的数据源构建双向同步。

ArkTS的状态管理

装饰器

6. 应用级别状态管理的装饰器：

- AppStorage是应用程序中的一个特殊的单例LocalStorage对象，是应用级的数据库，和进程绑定，通过@StorageProp和@StorageLink装饰器可以和组件联动。
- AppStorage是应用状态的“中枢”，将需要与组件（UI）交互的数据存入AppStorage，比如持久化数据PersistentStorage和环境变量Environment。UI再通过AppStorage提供的装饰器或者API接口，访问这些数据。
- 框架还提供了LocalStorage，AppStorage是LocalStorage特殊的单例。LocalStorage是应用程序声明的应用状态的内存“数据库”，通常用于页面级的状态共享，通过@LocalStorageProp和@LocalStorageLink装饰器可以和UI联动。

ArkTS的渲染控制

1. ArkUI通过自定义组件的build()函数和@builder装饰器中的声明式UI描述语句构建相应的UI。
2. 在声明式描述语句中开发者除了使用系统组件外，还可以使用渲染控制语句来辅助UI的构建。常用的渲染控制语句有条件渲染和循环渲染。
3. 条件渲染：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。
   - 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。

```
@Component
struct CounterView {
  @State counter: number = 0;
  label: string = 'unknown';
  build() {
    Row() {
      Text(`${this.label}`)
      Button(`counter ${this.counter} +1`)
        .onClick(() => {
          this.counter += 1;
        })
    }
  }
}
@Entry
@Component
struct MainView {
  @State toggle: boolean = true;
  build() {
    Column() {
      // 使用条件渲染，根据toggle的状态进行渲染
      if (this.toggle) {
        CounterView({ label: 'CounterView #positive' })
      } else {
        CounterView({ label: 'CounterView #negative' })
      }
      Button(`toggle ${this.toggle}`)
        .onClick(() => {
          this.toggle = !this.toggle;
        })
    }
  }
}
```

ArkTS的渲染控制

4. 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。ForEach包含arr、itemGenerator、keyGenerator三个接口。
   - **arr**：是数据源，为Array类型的数组。
   - **itemGenerator**：组件生成函数。为数组中的每个元素创建对应的组件。
   - **keyGenerator**：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。
   - 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。
   - 在确定键值生成规则后，ForEach的第二个参数itemGenerator函数会根据键值生成规则为数据源的每个数组项创建组件。组件的创建包括两种情况：ForEach首次渲染和ForEach非首次渲染。
   - 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。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。

```
// 接口描述
ForEach(
  arr: Array,
  itemGenerator: (item: Object, index: number) =>
void,
  keyGenerator?: (item: Object, index: number) =>
string
)

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

  build() {
    Row() {
      Column() {
        // 循环渲染，对应接口描述
        // this.simpleList为数据源。
        // (item: string) => item为键值生成器
        // (item: string) => { ChildItem({
            item: item })}为组件生成器。
        ForEach(this.simpleList, (item: string) => {
          ChildItem({ item: item })
        }, (item: string) => item)
      }
      .width('100%')
      .height('100%')
    }
    .height('100%')
    .backgroundColor(0xF1F3F5)
  }
}
```

中山大學
SUN YAT-SEN UNIVERSITY

谢谢观看

SUN YAT-SEN UNIVERSITY