# ArkTS开发基础

## 1、环境：

DevEco软件：

工程级目录



## 2、UI开发框架：

方舟开发框架针对不同目的和技术背景的开发者提供了两种开发范式

1. 声明式开发范式 ArkTs语言->UI更新->数据驱动更新
2. 类Web开发范式 JavaScript语言-> UI更新->数据驱动更新

3、应用模型：

FA(feature Ability)模型：HarmonyOS API 7开始支持的模型，已经不再主推

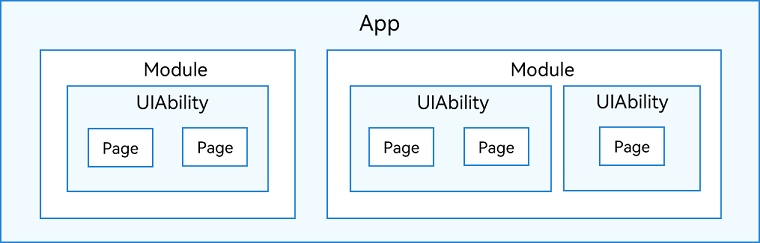
Stage模型：HarmonyOS API 9开始新增的模型，是目前主推且会长期演进的模型。在该模型中，提供了AbilityStage、WindowStage等类作为应用组件和Window窗口的“舞台”，因此称这种应用模型为Stage模型

4、应用程序包：

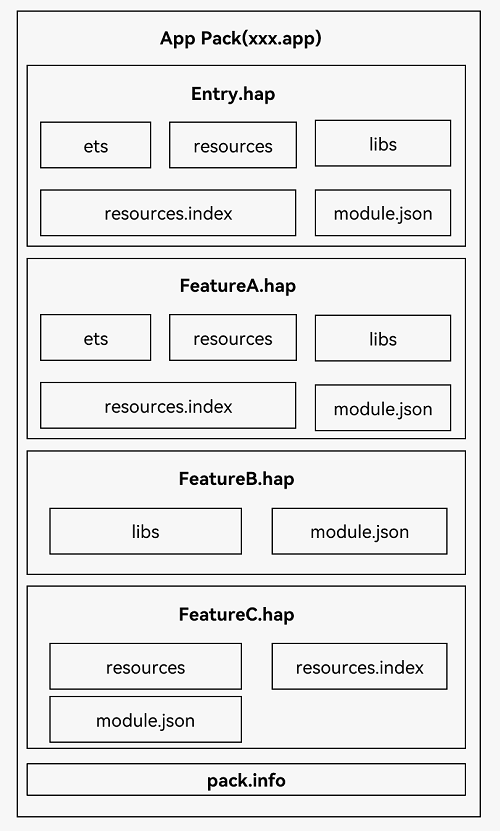
用户应用程序泛指运行在设备的操作系统之上，为用户提供特定服务的程序，简称“应用”。一个应用所对应的软件包文件，称为“应用程序包”。

在开发态，一个应用包含一个或者多个Module，可以在DevEco Studio工程中创建一个或者多个Module。Module是HarmonyOS应用/服务的基本功能单元，包含了源代码、资源文件、第三方库及应用/服务配置文件，每一个Module都可以独立进行编译和运行。Module分为“Ability”和“Library”两种类型，“Ability”类型的Module对应于编译后的HAP（Harmony Ability Package）；“Library”类型的Module对应于HAR（Harmony Archive），或者HSP（Harmony Shared Package）。

Module与UIAbility组件：



Stage模型应用程序包结构：



Entry类型的HAP：是应用的主模块

Feature类型的HAP：是应用的动态特性模块

每个HarmonyOS应用可以包含多个.hap文件，一个应用中的.hap文件合在一起称为一个Bundle，而bundleName就是应用的唯一标识

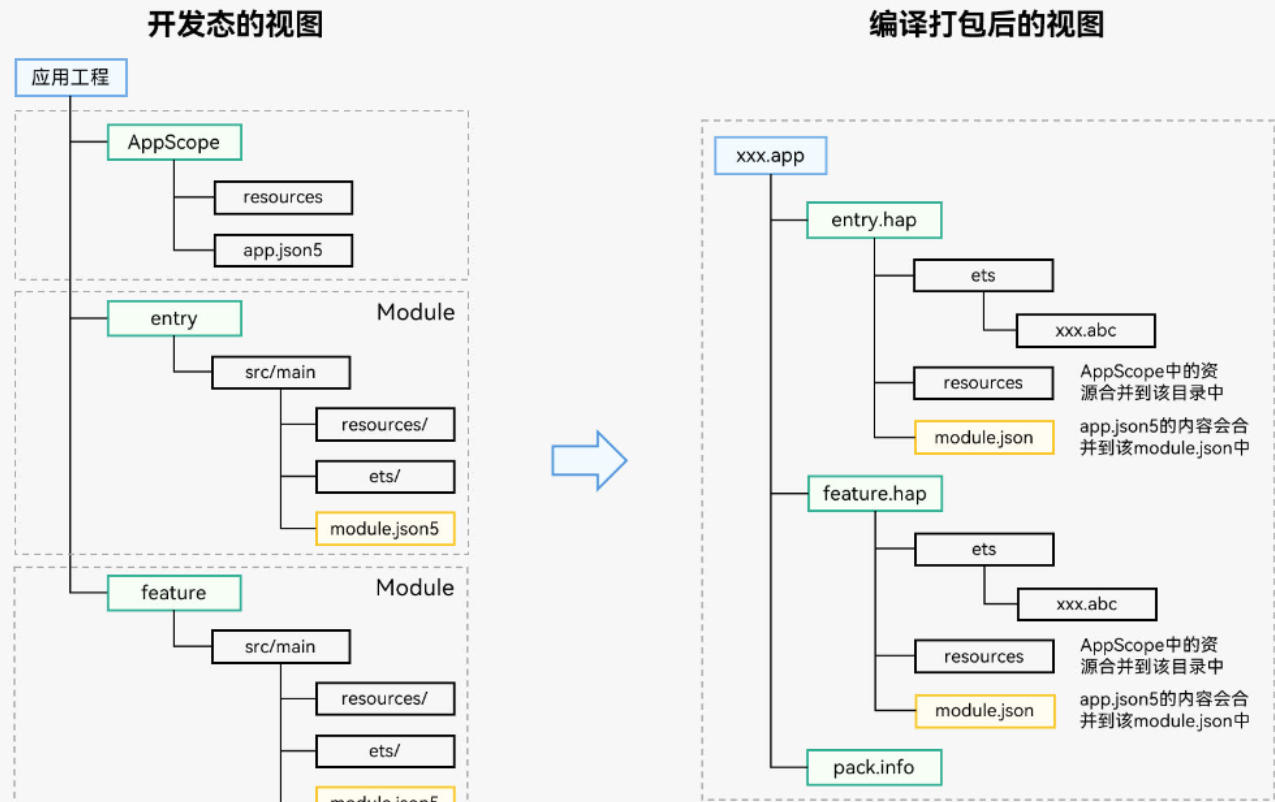
5、应用程序包多HAP机制

-设计目标

方便开发者模块化的管理应用，一般都是模块化管理，模块之间属于松耦合关系。多HAP方便了开发者将业务划分成多个模块，每个模块放到独立的HAP中。

例如支付类应用，有统一的主界面，主界面管理“扫一扫”、“收付款”、“消息”、“理财”等各个模块。其中主界面管理其他模块的逻辑在Entry包中实现，而“扫一扫”、“收付款”、“消息”和“理财”等模块在不同的Feature包中实现。可以同时开发多个Feature包，能够实现Feature包单独的开发测试，最终由Entry包统一集成Feature包的特性

-构建视图



--开发态视图：

AppScope-->app.json5:配置应用全局描述信息

resource目录-->存放应用的图标资源和应用名称的字符串资源

AppScope目录下面的文件名与Entry、Feature模块下面的文件名不能重复，否则IDE会报错

entry和feature为Module，实现应用的业务逻辑

resource:存放Module的所有资源

ets:实现业务逻辑

module.json5:Module的配置描述信息，如module名称、入口路径、组件信息

--编译后视图：

ets目录：ArkTS源码编译生成.abc文件。

resources目录：AppScope目录下的资源文件会合入到Module下面资源目录中

module配置文件：AppScope目录下的app.json5文件字段会合入到Module下面的module.json5文件之中，编译后生成HAP或HSP最终的module.json文件

## ArkTS语言

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

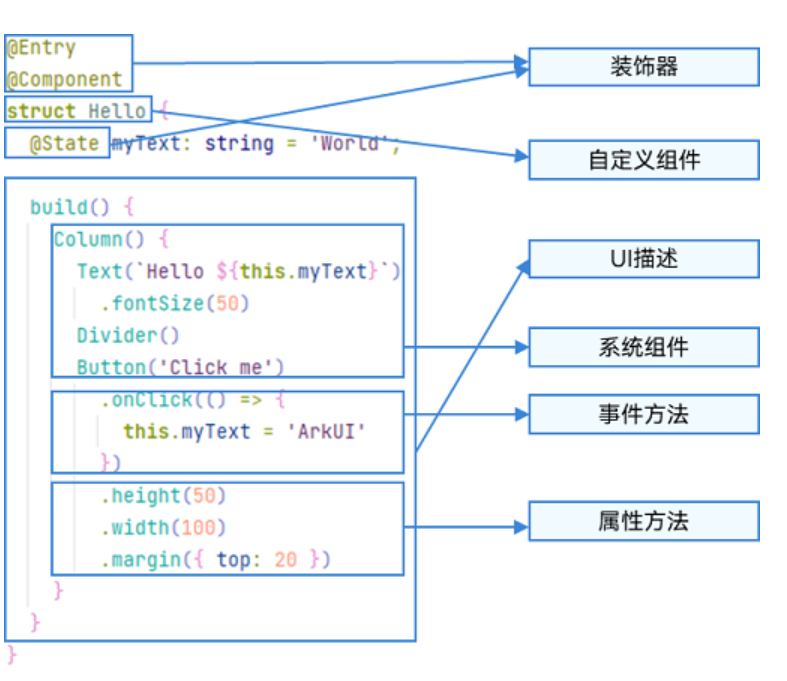
ArkTS的三个重要知识点：

--声明式UI基础语法

--状态管理

--渲染控制

### ---基础语法：



装饰器：用于修饰类、结构、方法及变量，并赋予其特殊的含义。如上图中，@Entry、@Component和@State都是装饰器。@Component表示自定义组件，@Entry表示该自定义组件的入口，@State表示组件中的状态变量，状态变量变化会触发UI刷新

UI描述：以声明式的方式来描述UI的结构，如build()方法中的代码块

自定义组件：可复用的UI单元，可组合其他组件，如上图中的@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，如上图中的Column

属性方法：组件可以通过链式调用配置多项属性

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

系统组件、属性方法、事件方法可参考：ArkTS的声明式开发范式

扩展语法：

@Builder/@BuilderParam：特殊的封装UI描述的方法，细粒度的封装和复用UI描述

@Extend/@Styles：扩展内置组件和封装属性样式，更领会地组合内置组件

@stateStyle：多态样式，可以根据组件的内部状态的不同，设置不同样式。

### ---声明式UI描述

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：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

1. 如果组件的接口定义没有包含必选构造参数，则()内不需要配置任何内容
2. 如果组件的接口定义包含必选构造参数，则应该在()配置相应参数

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，建议每个属性方法写一行

属性方法除了可以传递常量参数外，还可以传递变量或者表达式；对于系统组件，ArkUI还为其预定义了一些枚举类型供开发者使用。

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

1. 使用箭头函数配置组件的事件方法

.onclick(()=>{})

1. 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。

.onclick(function(){}.bind(this))

1. 使用组件的成员函数配置组件的事件方法

myClickHandler(): void{}

.onclick(this.myClickHandler.bind(this))

1. 使用声明的箭头函数，可以直接调用，不需要bind this

fn = () => {

console.info(`${this.counter}`)

this.counter++

}

.onclick(this.fn)

配置子组件

如果组件支持子组件配置，则需要在尾随闭包{…}中为组件添加子组件的UI描述

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

### ---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，由开发者自定义的称为自定义组件。UI界面开发需要考虑代码可复用性、业务逻辑与UI分离，将UI和部分业务逻辑封装成自定义组件时不可或缺的能力

特点：

可复用：可以组合使用系统组件、及其属性和方法

可重用：自定义组件可以被其他组件重用，并作为不同的实力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

@Component：该装饰器仅能装饰struct关键字声明的数据结构，struct被其修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

build()函数：build函数用于定义自定义组件的声明式UI描述

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

成员函数/变量：

自定义组件的成员函数为私有的，且不建议声明成静态函数

自定义组件的成员变量为私有的，且不建议声明成静态变量

自定义组件的参数规定：

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

build()函数：

@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。

Ps:

1. 不允许声明本地变量
2. 不允许直接使用console.info()
3. 不允许创建本地作用域
4. 不允许调用没有用@Builder装饰的方法
5. 不允许switch语法
6. 不允许使用表达式，如三目运算表达式

自定义组件通用样式：



### ---页面定义和自定义组件声明周期：

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，可以调用组件的声明周期。

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装饰的自定义组件作为页面的入口组件，也是页面的根节点，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装饰的组件的生命周期：

--onPageShow: 页面每次显示时触发一次，包括路由过程、应用进入前台等场景

--onPageHide: 页面每次隐藏时触发一次，包括路由过程、应用进入后台等场景

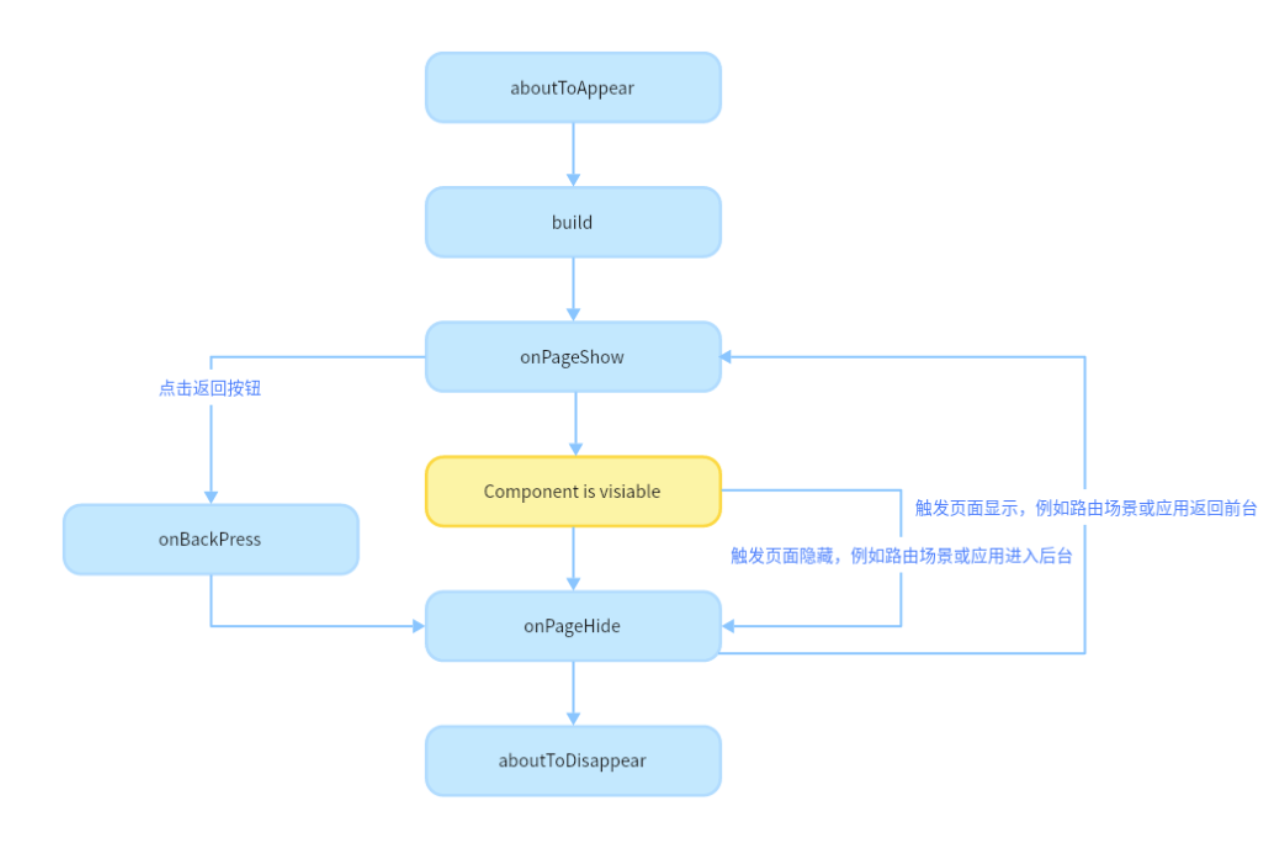
--onBackPress: 当用户点击返回按钮时触发

--组件生命周期，即一般用@Component装饰的自定义组件的生命周期

--aboutToAppear: 组件即将出现时回调该接口，具体时机为在创建自定义组件的新实例后，在执行build函数之前执行

--aboutToDisappear: 在自定义组件析构(在对象销毁前，执行一些相应的处理操作)销毁之前执行。不允许在aboutToDisappear函数中改变状态变量，特别是@Link变量的修改可能会导致应用程序行为不稳定

被@Entry装饰的组件（页面）生命周期：



--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。在首次渲染的过程中，框架会记录状态变量和组件的映射关系，当状态变量改变时，驱动器相关的组件刷新

当应用在后台启动时，此时应用进程并没有被销毁，所以仅需执行onPageShow

--自定义组件重新渲染：

当事件句柄被触发（点击事件触发）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-->键值对集合(组件和状态变量的映射关系)，框架可以知道该状态变量管理类哪些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不变，组件将被删除：

1. 在删除组件之前，将调用其aboutToDisapear生命周期函数，标记这该节点将要被销毁。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（获取一个变量的地址或对象的引用），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，如果有其同步的变量，比如@Link、@Prop、@StorageLink，将从同步源（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）上取消注册。

不建议在生命周期aboutToDisappear内使用async await异步操作，避免自定义组件被保留在Promise的闭包中，知道回调方法被执行完，这个行为阻止了自定义组件的垃圾回收

Ps:页面路由间跳转，需要在main\_pages.json中配置路由

### --@Builder装饰器：自定义构建函数

@Builder所装饰的函数遵循build()函数语法，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

1、自定义组件内自定义构建函数

语法：@Builder MyBuilderFunction(){...}

使用方法this.MyBuilderFunction()

---允许在自定义组件内定义一个或多个@Builder方法，该方法被认为是该组件的私有、特殊类型的成员函数

---自定义构建函数可以在所属组件的build方法和其他自定义构建函数中调用，但不允许在组件外调用

---在自定义函数体中，this指代当前所属组件，组件的状态变量可以在自定义构建函数内访问。建议通过this访问自定义组件的状态变量而不是参数传递

2、全局自定义构建函数

语法：@Builder function MyGlobalBuildFunciton(){...}

使用：MyGlobalBuidlerFunction()

---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

---如果不设计组件状态变化，建议使用全局的自定义构建方法

3、参数传递规则

自定义构建函数的参数传递由“按值传递”、“按引用传递”两种，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undfinded、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，建议使用@Link

-@Builder内UI语法遵循UI语法规则

只有传入一个参数，且参数需要直接传入对象字面量才会按引用传递该参数，其余方式均为按值传递。

按引用传递：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

overbuilder($$ : {paramAa1: string,paramB1: string});

按值传递：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。

---@BuilderParam装饰器：引用@Builder函数

若直接在组件内嵌入事件方法，将会导致所有引入该自定义组件的地方均增加了该功能。为此，ArkUI引入了@BuilderParam装饰器，@BuilderParam用来装饰指向@Builder方法的变量，开发者可在初始化自定义组件时对此属性进行赋值，为自定义组件增加特定的功能。该装饰器用于声明任意UI描述的一个元素。

--装饰器使用

-初始化@BuilderParam装饰的方法

@BuilderParam装饰的方法只能被自定义构建函数(@Builder装饰的方法)初始化

-1、使用所属自定义组件的自定义构建函数或者全局的自定义构建函数，在本地初始化@BuilderParam：

@Builder function overbuilder(){}

@Component

Struct Child{

@Builder doNothingBuilder(){};

//使用自定义组件的自定义构建函数初始化@BuilderParam

@BuilderParam customBuilderParam: () => void = this.doNothingBuilder

//使用全局自定义构建函数初始化@BuilderParam

@BuilderParam customOverBuilderParam: ()=>void = overBuilder

build(){}

}

-2、用父组件自定义构建函数初始化子组件@BuilderParam装饰的方法：

@Component

Struct Child{

//使用父组件@Builder装饰的方法初始化子组件@BuilderParam

@BuilderParam customBuilderParam: ()=> void;

build(){

Column(){

This.customBuilderParam()

}

}

}

@Entry

@Component

Struct Parent{

@Builder componentBuilder(){

Text(`Parent builder`)

}

build(){

Column(){

Child({customBuilderParam: this.componentBuilder})

}

}

}

--使用场景

-参数初始化组件

@BuilderParam装饰的方法可以是有参数和无参数的两种形式，需与指向的@Builder方法类型匹配。@BuilderParam装饰的方法类型需要和@Builder方法类型一致。

-尾随闭包初始化组件

在自定义组件中使用@BuilderParam装饰的属性时也可通过尾随闭包进行初始化。在初始化自定义组件时，组件后紧跟一个大括号’’{}’’形成尾随闭包场景。（在此场景下自定义组件内有且仅有一个使用@BuilderParam装饰的属性）

开发者可以将尾随闭包内的内容看作@Builder装饰的函数传给@BuilderParam

### --Styles装饰器：定义组件重用样式

@Styles可以提炼公共样式成一个方法，直接在组件声明的位置调用。通过@Styles装饰器可以快速定义并复用自定义样式。

---当前@Styles仅支持通用属性和通用事件

---@Styles方法不支持参数

---@Styles可以定义在组件内或全局，在全局定义时需要在方法名前面加上function关键字，组件内定义时则不需要。

---@Style定义的样式只能在当前文件内使用，不支持export

---定义在组件内的@Styles可以通过this访问组件的常量和状态变量，并可以在@Styles里通过事件来改变状态变量的值。

---组件内@Styles的优先级高于全局@Styles，如果找不到，则会全局查找

--@Extend装饰器：定义扩展组件样式

在@Styles（仅支持通用属性和通用事件）的基础上，提供了@Extend，用于扩展原生组件样式。

语法：@Extend(UIComponentName) function functionName{...}

使用规则：

---和@Styles不同，@Extend仅支持定义在全局，不支持在组件内部定义

---@Extend定义的样式只能在当前文件内使用，不支持export

---和@Styles不同，@Extend支持封装指定的组件的私有属性和私有事件和预定义相同组件的@Extend的方法

---和@Styles不同，@Extend装饰的方法支持参数，开发者可以在调用函数时传递参数，调用遵循TS方法传值调用

---@Extend装饰的方法的参数可以为function，作为Event事件的句柄

---@Extend的参数可以为状态变量，当状态变量改变时，UI可以正常的被刷新渲染

--stateStyles：多态样式

@Styles和@Extend仅仅应用与静态页面的样式复用，stateStyles可以依据组件的内部状态的不同，快速设置不同样式。

stateStyles是属性方法，可以根据UI内部状态来设置样式，类似于css伪类，但语法不同。ArkUI提供以下四种状态：

--1、focused：获焦态

--2、normal：正常态

--3、pressed：按压态

--4、disabled：不可用态

使用场景

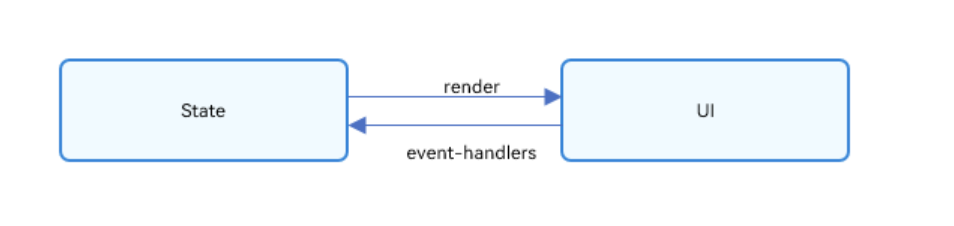
--基础场景

### ---状态管理

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

状态管理机制：在声明式UI编程框架中，UI是程序状态的允许结果，用户构建了一个UI模型，其中应用的运行时的状态时参数。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，叫做状态管理机制。

状态变量的改变会引起UI的渲染刷新：



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。

-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。

-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。

-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。

-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。

-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。

装饰器总览

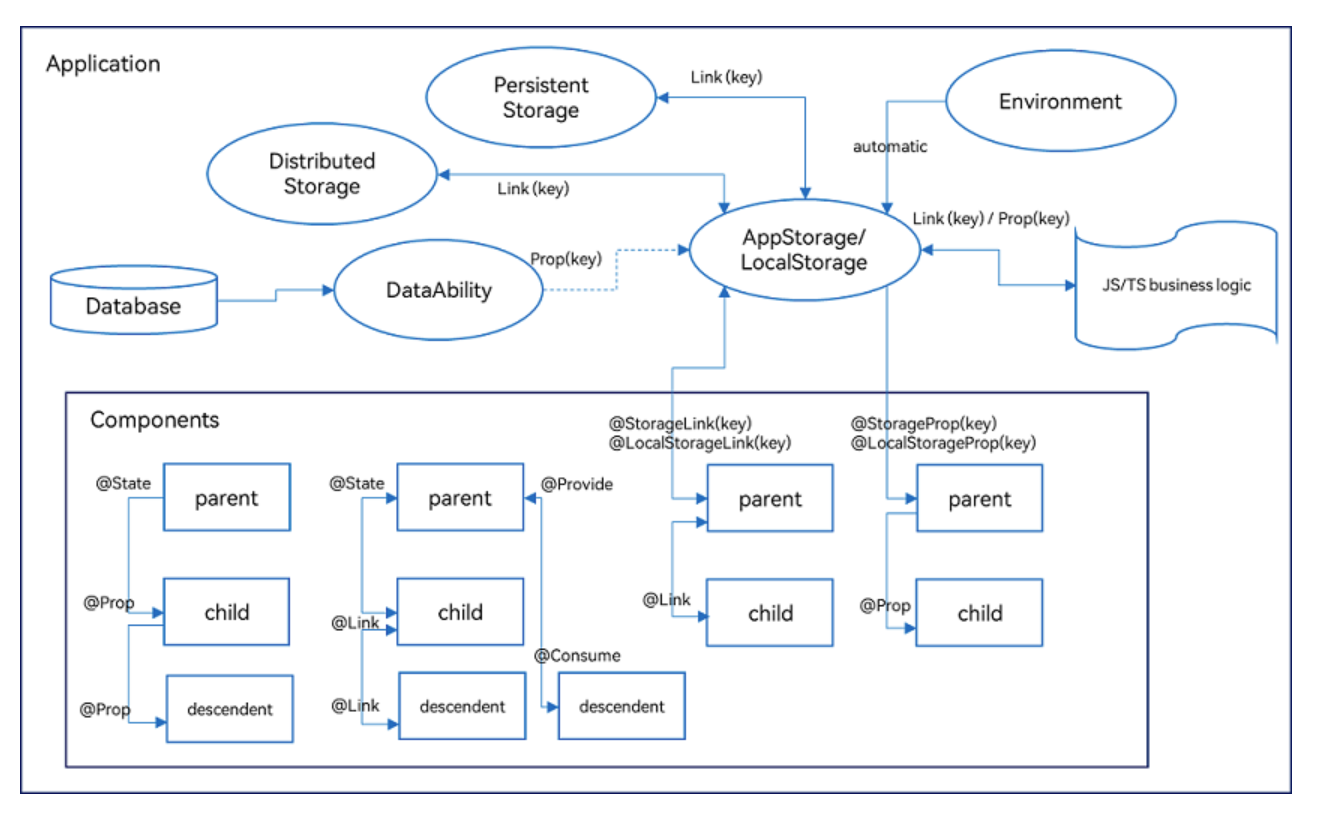
ArkUI装饰器分为：

1. 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
2. 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

从数据的传递形式和同步类型，装饰器分为：

--只读的单向传递

--可变更的双向传递



Components部分的装饰器为组件级别的状态管理，Application部分为应用的状态管理。开发者可以通过@StorageLink/@LocalStorageLink实现应用和组件状态的双向同步，通过@StorageProp/@LocalStorageProp实现应用和组件状态的单向同步。

管理组件拥有的状态：

@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。

1、@Link：@Link装饰的变量和父组件构建双向同步关系的状态变量，父组件会接受来自@Link装饰的变量的修改的同步，父组件的更新也会同步给@Link装饰的变量。

2、@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。

3、@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连用。

4、@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。

Ps: 仅@Observed/@ObjectLink可以观察嵌套场景，其他的状态变量仅能观察第一层

管理应用拥有的状态：

1、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。

2、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。

3、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。

其他状态管理功能：

@Watch用于监听状态变量的变化。

$$运算符：给内置组件提供TS变量的引用，使得TS变量和内置组件的内部状态保持同步。

--管理组件拥有的状态

#### @State装饰器：组件状态

-@State装饰的变量，称为状态变量，一旦拥有了状态属性，就和自定义组件的渲染绑定起来。当组件改变时，UI会发生相应的渲染改变。@State是最基础的，使变量拥有状态属性的装饰器，它是大部分状态变量的数据源。私有，只能从组件内部访问，声明时必须指定其类型和本地初始化。

@State变量特点：

--@State装饰的变量与子组件中的@Prop装饰变量之间建立单向数据同步，与@Link、@ObjectLink装饰变量之间建立双向数据同步

--@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

@State使用规则：

--装饰器参数无

--不与父组件中任何类型的变量同步

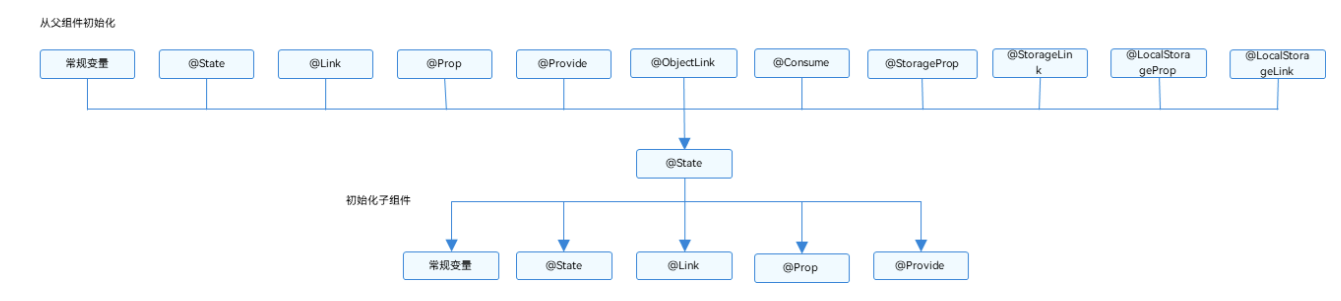
--允许装饰的变量类型：Object、class、string、number、boolean、enum类型，以及这些类型的数组。嵌套类型的场景参考“观察变化”。类型必须被指定。不支持any，不支持简单类型和赋值类型的联合类型，不允许使用undefinded和null。

--被装饰变量的初始值：必须本地初始化。

Ps:建议不要装饰Date类型，不支持Length、ResourceStr、ResourceColor类型，这些类型为简单类型和复杂类型的联合类型。

@State变量的传递/访问规则说明

|  |  |
| --- | --- |
| 传递/访问 | 说明 |
| 从父组件初始化 | 可选，从父组件初始化或者本地初始化。如果从父组件初始化将会覆盖本地初始化。  支持父组件中常规变量、@State、@Link、@Prop、@Provide、@Consume、@ObjectLink、@StorageLink、@StorageProp、@LocalStorageLink和@LocalStorageProp装饰的变量，初始化子组件的@State。 |
| 用于初始化子组件 | @State装饰的变量支持初始化子组件的常规变量、@State、@Link、@Prop、@Provide |
| 是否支持组件外访问 | 不支持，只能在组件内访问 |



@State观察变化和行为表现

--并不是状态变量的所有更改都会引起UI的刷新，只有可以被框架观察到的修改才会引起UI刷新。

-观察变化：

--当装饰的数据类型为boolean、string、number类型时，可以观察到数值的变化

--当装饰的数据类型为class或者object时，可以观察到自身的赋值的变化，和其属性赋值的变化，即Object.keys(obsercedObject)返回的所有属性，嵌套属性的赋值观察不到

--当装饰的对象是array时，可以观察到数组本身的赋值和添加、删除、更新数组的变化。数组项中属性的赋值观察不到

框架行为

---当状态变量被改变时，查询依赖该状态变量的组件

---执行依赖该组件状态变量的组件的更新方法，组件更新渲染

---和状态变量不相关的组件或者UI描述不会发生重新渲染，从而实现页面渲染的按需更新。

@State使用场景：

-装饰简单类型的变量

-装饰class对象类型的变量

@State变量首次渲染的初始化流程：

1. 使用默认的本地初始化
2. 如果没有命名参数传值，则使用本地初始化的默认值

在父组件与子组件之间，发生数据同步，需要使用@Prop和@Link单向或双向同步

#### @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。但变化不会同步回其父组件

-@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。

限制：@Prop装饰器不能再@Entry装饰的自定义组件中使用

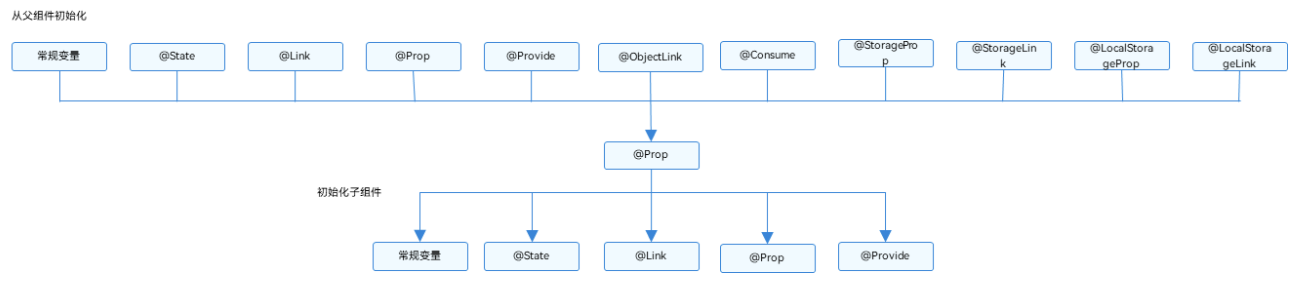
装饰器使用规则：

|  |  |
| --- | --- |
| @Prop | 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变量的值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及使用场景 | String,number,Boolean,enum类型。  不支持any,不允许使用undefined和null。  必须指定类型  @Prop和数据源类型相同，有一下三种情况：  -@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和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 |

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 | 组件私有，不能 |

初始化规则：



观察变化：

-当装饰的类型是允许的类型，string,number,Boolean,enum都可以观察

-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变量变化时变量值会同步更新至@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的同步机制时相同的

--数据源和@Prop变量的类型需要相同

框架行为：

1. 初始渲染：

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

初始化子组件@Prop装饰的变量

1. 更新：

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

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。

说明：@Prop装饰的数据更新依赖其所属自定义组件的重新渲染，所以在应用后台，@Prop无法刷新，推荐使用@Link代替。

#### @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

概述：@Link装饰的变量与其父组件中的数据共享相同的值

限制：@Link装饰器不能在@Entry装饰的自定义组件中使用

装饰器使用说明：

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和复合类型。

禁止本地初始化。

变量的传递/访问：

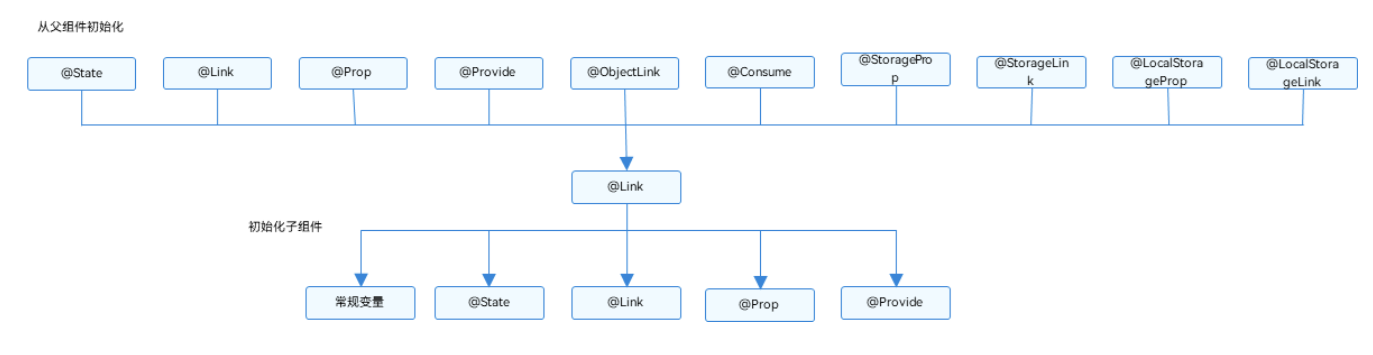
从父组件初始化和更新：必选。与父组件@State,@Link,@StorageLink建立双向数据绑定。允许父组件中@State、@Link、@Prop、@Provide、@Consume、@ObjectLink、@StorageLink、@StorageProp、@LocalStorageLink和@LocalStorageProp装饰变量初始化子组件@Link。

从API 9开始，@Link子组件从父组件初始化@State语法为Comp({aLink:this.aState})。同样Comp({aLink:$aState})也支持。

允许用于初始化常规变量，@State,@Link,@Prop,@Provide

私有，只能在所属组件访问

初始化规则：



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。

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：

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）。

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

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

处理步骤：

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）。

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

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

@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。

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。

#### @Provide装饰器和@Comsume装饰器：与后代组件双向同步

-@Provide和@Consume，应用于与后代组件的双向数据同步，应用于状态数据在多个层级之间传递的场景。不同于父子组件之间通过命名参数机制传递，@Provide和@Consume摆脱参数传递机制的束缚，实现跨层级传递。

其中@Provide装饰的变量是在祖先节点中，可以理解为被“提供”给后代的状态变量。@Consume装饰的变量是在后代组件中，去“消费（绑定）”祖先节点提供的变量。

-特性：

--@Provide装饰的状态变量自动对其所有后代组件可用。

--后代通过使用@Consume去获取@Provide提供的变量，建立在@Provide和@Consume之间的双向数据同步，与@State和@Link不同的是，前者可以在多层级的父子组件之间传递。

--@Provide和@Consume可以通过相同的变量名或者相同的变量别名绑定，变量类型必须相同。

// 通过相同的变量名绑定

@Provide a: number = 0;

@Consume a: number;

// 通过相同的变量别名绑定

@Provide('a') b: number = 0;

@Consume('a') c: number;

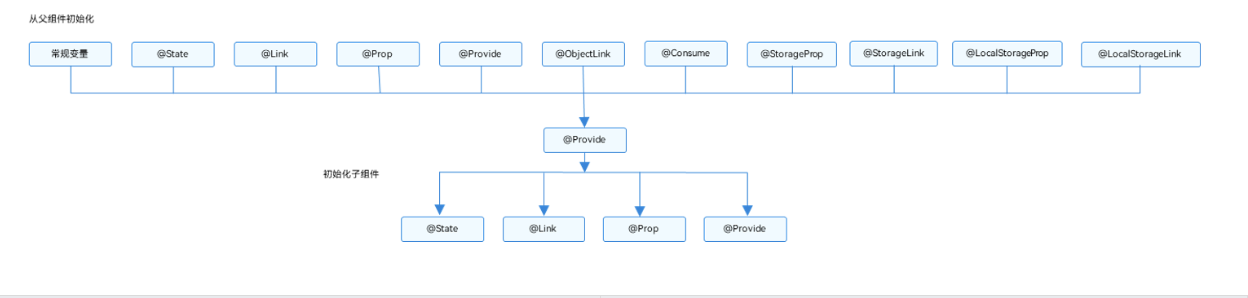
注意：@Provide和@Consume通过相同的变量名或者相同的变量别名绑定时，@Provide修饰的变量和@Consume修饰的变量是一对多的关系。不允许在同一个自定义舰内，包括其子组件中声明多个同名或者同别名的@Provide装饰的变量。

装饰器说明：

|  |  |
| --- | --- |
| @Provide变量装饰器 | 说明 |
| 装饰器参数 | 别名：字符串常量，可选  如果指定了别名，则通过别名来绑定变量；如果未指定别名，则通过变量名绑定变量。 |
| 同步类型 | 双向同步  从@Provide变量到所有@Consume变量以及相反的方向的数据同步。双向同步的操作与@State和@Link的组合相同。 |
| 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。  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。  必须指定类型。@Provide变量的@Consume变量的类型必须相同。 |
| 被装饰变量的初始值 | 必须指定。 |

|  |  |
| --- | --- |
| @Consume装饰器 | 说明 |
| 装饰器参数 | 别名：常量字符串，可选。  如果提供了别名，则必须有@Provide的变量和其有相同的别名才可以匹配成功；否则，则需要变量名相同才能匹配成功。 |
| 同步类型 | 双向：从@Provide变量到所有@Consume变量，以及相反的方向。双向同步操作与@State和@Link的组合相同 |
| 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。  不支持any，不允许使用undefined和null。  必须指定类型。@Provide变量的@Consume变量的类型必须相同。 |
| 被装饰变量的初始值 | 无，禁止本地初始化。 |

@Provide初始化规则：



@Consume初始化规则：



观察变化：

-当装饰的数据类型为boolean、string、number类型时，可以观察到数值的变化。

-当装饰的数据类型为class或者Object的时候，可以观察到赋值和属性赋值的变化（属性为Object.keys(observedObject)返回的所有属性）。

-当装饰的对象是array的时候，可以观察到数组的添加、删除、更新数组单元。

框架行为：

-初始渲染：

1.@Provide装饰的变量会以map的形式，传递给当前@Provide所属组件的所有子组件；

2.子组件中如果使用@Consume变量，则会在map中查找是否有该变量名/alias（别名）对应的@Provide的变量，如果查找不到，框架会抛出JS ERROR;

3.在初始化@Consume变量时，和@State/@Link的流程类似，@Consume变量会保存在map中查找到的@Provide变量，并把自己注册给@Provide。

-当@Provide装饰的数据变化时：

1.通过初始渲染的步骤可知，子组件@Consume已把自己注册给父组件。父组件@Provide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（@Consume）；

2.通知@Consume更新后，子组件所有依赖@Consume的系统组件（elementId）都会被通知更新。以此实现@Provide对@Consume状态数据同步。

-当@Consume装饰的数据变化时：

1.通过初始渲染的步骤可知，子组件@Consume持有@Provide的实例。在@Consume更新后调用@Provide的更新方法，将更新的数值同步回@Provide，以此实现@Consume向@Provide的同步更新。

#### @Observed装饰器和@ObjectLink装饰器：嵌套类和对象属性变化

对于多层嵌套的情况，比如二维数组，或者数组项class,或者class的属性为class,他们的第二层的属性变化是无法观察到的，而@Observed/@ObjectLink可以解决。

概述：

@Observed和@ObjectLink装饰器用于在设计嵌套对象或对象数组的场景中进行双向数据同步。

-被@Observed装饰的类，可以被观察到属性的变化；

-子组件中@ObjectLink装饰器的状态变量用于接收@Obserced装饰的类的实例，和父组件中对应的状态变量建立双向数据绑定。这个实例可以是数组中的被@Observed装饰的项，或者是class object中的属性，这个属性同样也需要被@Observed装饰。

-单独使用@Observed是没有任何作用的，需要搭配@ObjectLink或者Prop使用。

限制：

-使用@Observed装饰class会改变class原始的原型链，@Observed和其他类装饰器装饰同一个class可能会带来问题

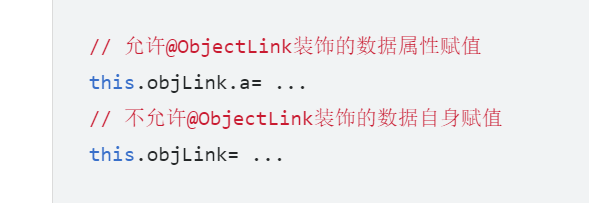
-@ObjectLink装饰器不能@Entry装饰的自定义组件中使用。

装饰器说明：

|  |  |
| --- | --- |
| @Observed类装饰器 | 说明 |
| 装饰器参数 | 无 |
| 类装饰器 | 装饰class.需要放在class的定义前，使用new创建类对象 |

|  |  |
| --- | --- |
| @ObjectLink变量装饰器 | 说明 |
| 装饰器参数 | 无 |
| 同步类型 | 不与父组件中的任何类型同步变量 |
| 允许装饰的变量类型 | 必须为被@Observed装饰的class实例，必须指定类型。  不支持简单类型，可以使用@Prop。  @ObjectLink的属性是可以改变的，但是变量的分配是不允许的，也就是说这个装饰器装饰变量是只读的，不能被改变。 |
| 被装饰变量的初始值 | 不允许 |

@ObjectLink装饰的数据为可读示例



Ps:

-@ObjectLink装饰的变量不能被赋值，如果要使用赋值操作，请使用@Prop。

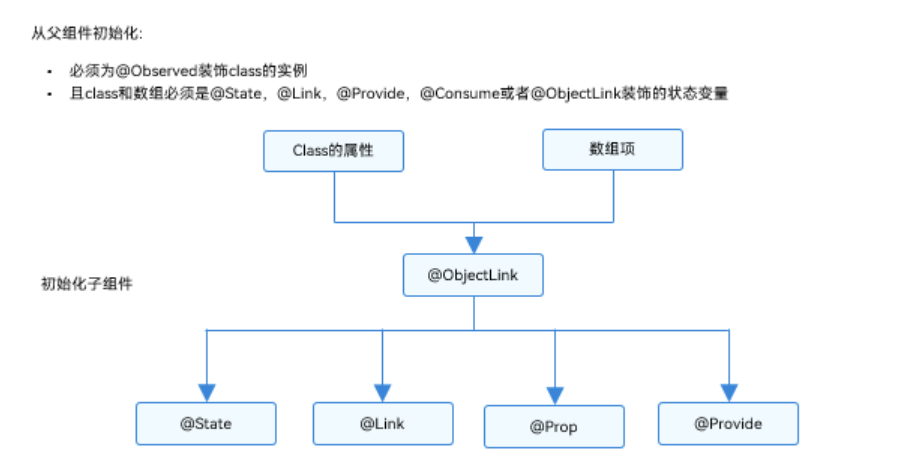
-@Prop装饰的变量和数据源的关系是是单向同步，@Prop装饰的变量在本地拷贝了数据源，所以它允许本地更改，如果父组件中的数据源有更新，@Prop装饰的变量本地的修改将被覆盖；

-@ObjectLink装饰的变量和数据源的关系是双向同步，@ObjectLink装饰的变量相当于指向数据源的指针。如果一旦发生@ObjectLink装饰的变量的赋值，则同步链将被打断。

变量的传递/访问规则

|  |  |
| --- | --- |
| @ObjectLink传递/访问 | 说明 |
| 从父组件初始化 | 必须指定。  初始化@ObjectLink装饰的变量必须同时满足以下场景;  -类型必须是@Observed装饰的class。  -初始化的数值需要是数组项，或者class的属性  -同步源的class或者数组必须是@State,@Link,@Provide,@Consume或者@ObjectLink装饰的数据 |
| 与源对象同步 | 双向 |
| 可以初始化子组件 | 允许，可用于初始化常规变量，@State,@Link,@Prop,@Provide |

初始化规则：



观察的变化：

- @Observed装饰的类，如果其属性为非简单类型，比如class、Object或者数组，也需要被@Observed装饰，否则将观察不到其属性的变化

-其属性的数值的变化，其中属性是指Object.keys(observedObject)返回的所有属性，示例请参考嵌套对象。

-如果数据源是数组，则可以观察到数组item的替换，如果数据源是class，可观察到class的属性的变化，示例请参考对象数组。

框架行为：

-1、初始渲染：

--@Observed装饰的class的实例会被不透明的代理对象包装，代理了class上的属性setter和getter方法

--子组件中@ObjectLink装饰的从父组件初始化，接收被@Observed装饰的class的实例，@ObjectLink的包装类会将自己注册给@Observed class

-2、属性更新：当@Observed装饰的class属性改变时，会走到代理的setter和getter，然后遍历依赖它的@ObjectLink包装类，通知数据更新。

使用场景：

嵌套对象、对象数组、二维数组。

### 管理应用拥有的状态概述

如果开发者要实现应用级的，或者多个页面的状态数据共享，就需要用到应用级别的状态管理的概念。ArkTS根据不同特性，提供了多种应用状态管理的能力：

-LocalStorage：页面级UI状态存储，通常用于UIAbility内、页面间的状态共享

-AppStorage：特殊的单例LocalStorage对象，由UI框架在应用程序启动时创建，为应用程序UI状态属性提供中央存储

-PersistentStorage：持久化存储UI状态，通常和AppStorage配合使用，选择AppStorage存储的数据写入磁盘，以确保这些属性在应用程序重新启动时的值与应用程序关闭时的值相同。

-Environment：应用程序允许的设备的环境参数，环境参数会同步到AppStorage中，可以和AppStorage搭配使用。

LocalStorage：页面级UI状态存储

LocalStorage是叶面级的UI状态存储，通过@Entry装饰器接收的参数可以在页面内共享同一个LocalStorage实例。LocalStorage也可以在UIAbility实例内，在页面间共享状态。

Environment：应用程序运行的设备的环境参数，环境参数会同步到AppStorage中，可以和AppStroage搭配使用

#### -LocalStorage：页面级UI状态存储

该存储是页面级的UI状态存储，通过@Entry装饰器接收的参数可以在页面内共享同一个LocalStorage实例。LocalStorage也可以在UIAbility实例内，在页面间共享状态。

概述：LocalStorage是ArkTS为构建页面级别状态遍历提供存储的内存内“数据库”

1、应用程序可以创建多个LocalStorage实例，LocalStorage实例可以在页面内共享，也可以通过GetShared接口，实现跨页面、UIAbility实例内共享。

2、组件树的根节点，即被@Entry装饰的Component，可以被分配一个LocalStorage实例，此组件的所有子组件实例将自动获得对该LoocalStorage实例的访问权限。

3、被@Component装饰的组件最多可以访问一个LocalStorage实例和AppStorage，未被@Entry装饰额组件不可被独立分配LocalStorage实例，只能接收父组件通过@Entry传递的LocalStorage实例。一个LocalStorage实例在组件树上可以被分配给多个组件。

4、LocalStorage中的所有属性都是可变的。

应用程序决定LocalStorage对象的生命周期。当应用释放最后一个指向LocalStorage的引用时，LocalStorage将被JS Engine垃圾回收

LocalStorage根据与@Component装饰的组件的同步类型不同，提供了两个装饰器：

@LocalStorageProp：@LocalStorageProp装饰的变量和与LocalStorage中给定属性建立单向同步关系。

@LocalStorageLink：@LocalStorageLink装饰的变量和在@Component中创建与LocalStorage中给定属性建立双向同步关系。

限制：

LocalStorage创建后，命名属性的类型不可更改。后续调用Set时必须使用相同类型的值。

LocalStorage是页面级存储，GetShared接口仅能获取当前Stage通过windowStage.loadContent传入的LocalStorage实例，否则返回undefined.

@LocalStorageProp

使用@LocalStorageProp(key)/@LocalStorageLink(key)装饰组件内的变量，key标识了LocalStorage的属性。

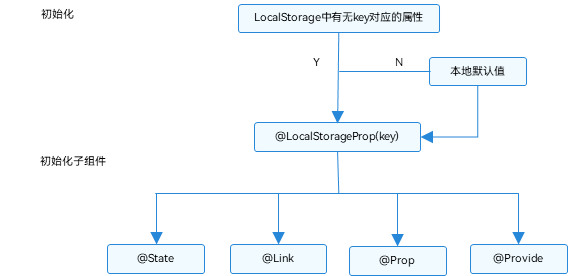
当自定义组件初始化的时候，@LocalStorageProp(key) /@LocalStorageLink(key)装饰的变量会通过给定的key，绑定LocalStorage对应的属性，完成初始化。本地初始化是必要的，因为无法保证LocalStorage一定存在给定的key(取决于应用逻辑是否在组件初始化之前在LocalStorage实例中存入相应的属性)

如果LocalStorage给定key的属性发生变化，改变会被同步给@LocalStorageProp，并覆盖掉本地的修改。

装饰器使用规则：

|  |  |
| --- | --- |
| @LocalStorageProp变量装饰器 | 说明 |
| 装饰器参数 | Key：常量字符串，必填 |
| 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，且必须和LocalStorage中对应属性相同。 |
| 同步类型 | 单向同步：从LocalStorage的对应属性到组件的状态变量。组件本地的修改是允许的，但是LocalStorage中给定的属性一旦发生变化，将覆盖本地的修改 |
| 被装饰变量的初始值 | 必须指定，如果LocalStorage实例中不存在属性，则作为初始化默认值，并存入LocalStorage |

变量的传递/访问规则：



观察变化和行为表现：

观察变化：

-当装饰的数据类型为boolean,string,number类型时，可以观察到数值的变化。

-当装饰的数据类型为class或Object时，可以观察到赋值和属性赋值的变化

-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。

框架行为

-当@LocalStorageProp(key)装饰的数值改变被观察到时，修改不会被同步回LocalStorage对应属性键值key的属性中。

-当前@LocalStorageProp(key)单向绑定的数据会被修改

-当@LocalStorageProp(key)装饰的数据本身是状态变量，它的改变虽然不hi同步回LocalStorage中，但是会引起所属的自定义组件的重新渲染

当LocalStorage中key对象的属性发生改变时，会同步给所有@LocalStorageProp(key)装饰的数据，@LocalStorageProp(key)本地的修改将被覆盖。

@LocalStorageLink

@LocalStorageLink(key)是和LocalStorage中key对应的属性建立双向数据同步：

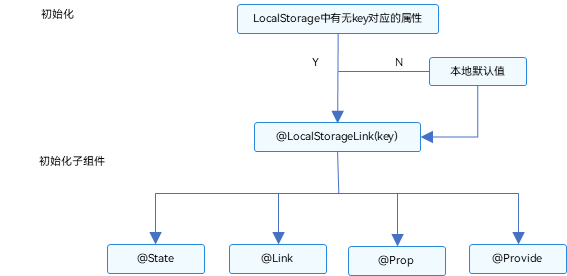
本地修改发生，该修改会被写回LocalStorage中；

LocalStorage中的修改发生后，该修改会被同步到所有绑定LocalStorage对应key的属性上，包括单向（@LocalStorageProp和通过prop创建的单向绑定变量）、双向（@LocalStorageLink和通过link创建的双向绑定变量）变量。

装饰器使用规则：

|  |  |
| --- | --- |
| @LocalStorageLink变量装饰器 | 说明 |
| 装饰器参数 | Key：常量字符串，必填 |
| 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，且必须和LocalStorage中对应属性相同。 |
| 同步类型 | 双向同步：从LocalStorage的对应属性到自定义组件，从自定义组件到LocalStorage对应属性。 |
| 被装饰变量的初始值 | 必须指定，如果LocalStorage实例中不存在属性，则作为初始化默认值，并存入LocalStorage |

变量的传递/访问：



观察变化和行为表现：

观察变化：

-当装饰的数据类型为boolean、string、number类型时，可以观察到数值的变化。

-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。

框架行为：

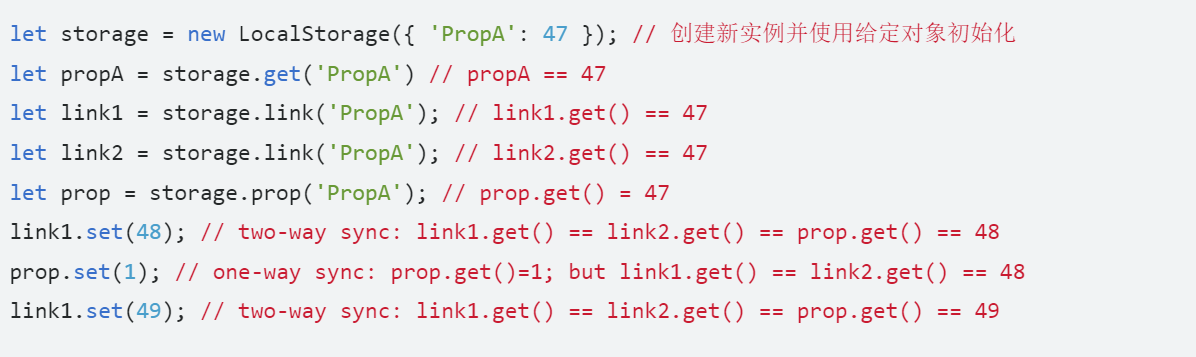
-当@LocalStorageLink(key)装饰的数值改变被观察到时，修改将被同步回LocalStorage对应属性键值key的属性中。

-LocalStorage中属性键值key对应的数据一旦改变，属性键值key绑定的所有的数据（包括双向@LocalStorageLink和单向@LocalStorageProp）都将同步修改。

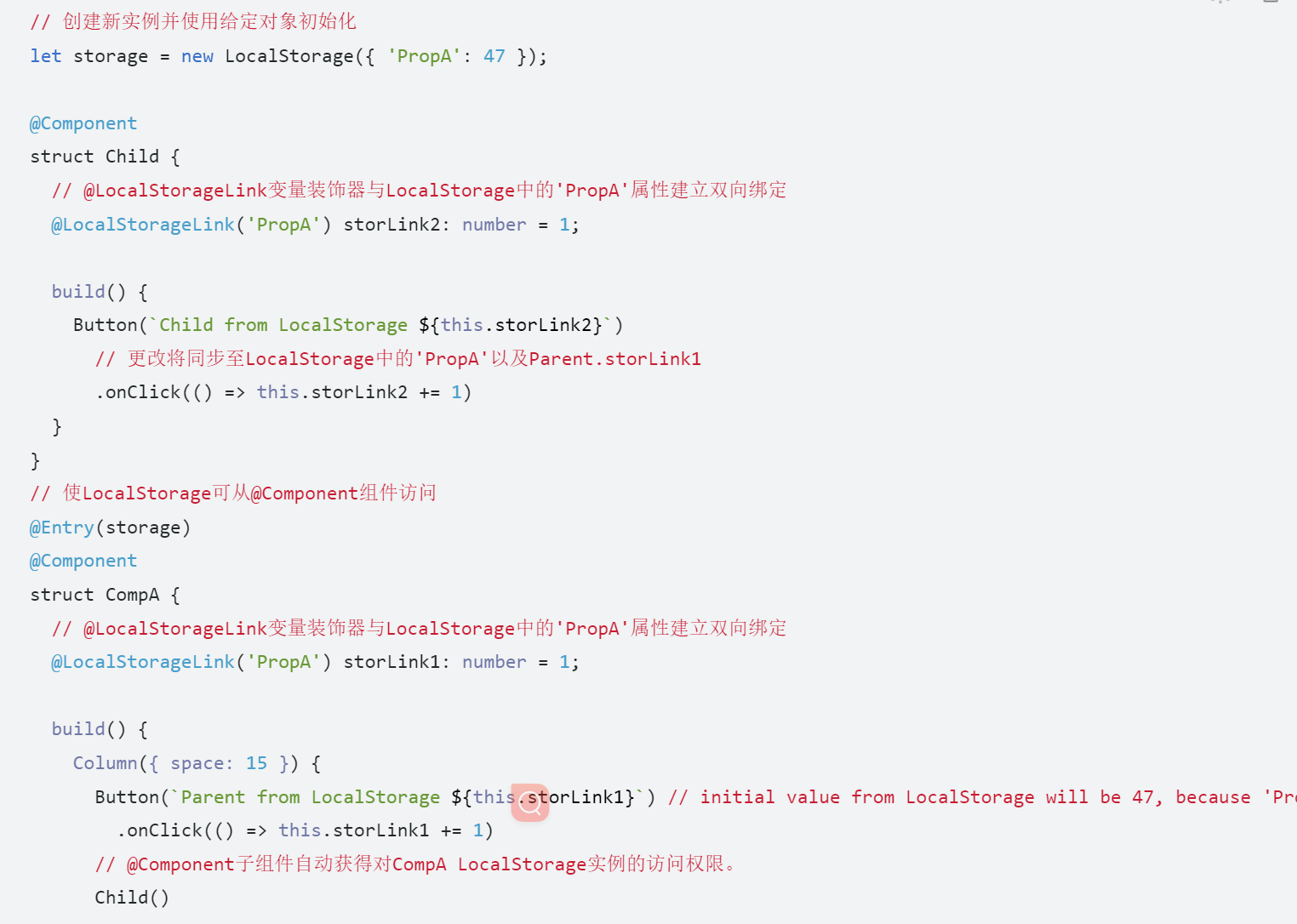
-当@LocalStorageLink(key)装饰的数据本身是状态变量，它的改变不仅仅会同步回LocalStorage中，还会引起所属的自定义组件的重新渲染。

使用场景：

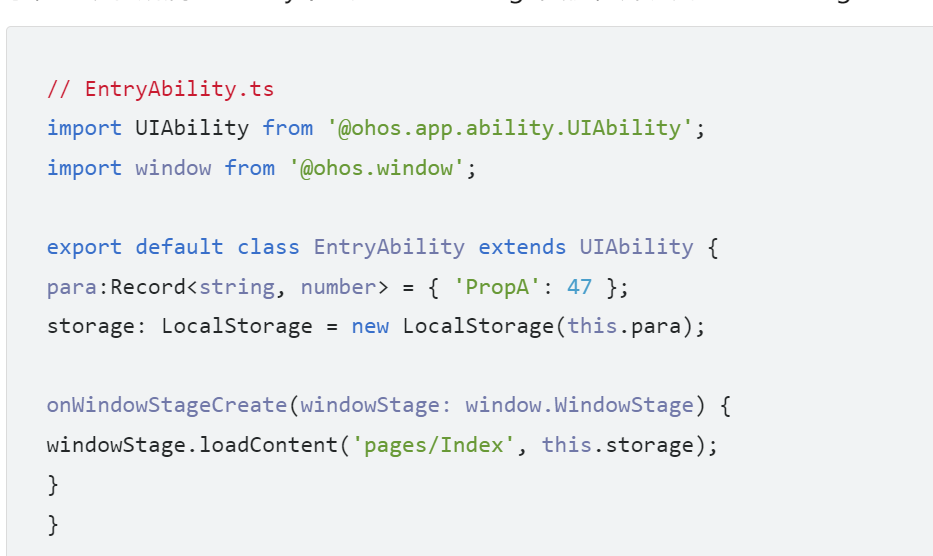
1. 应用逻辑使用LocalStorage

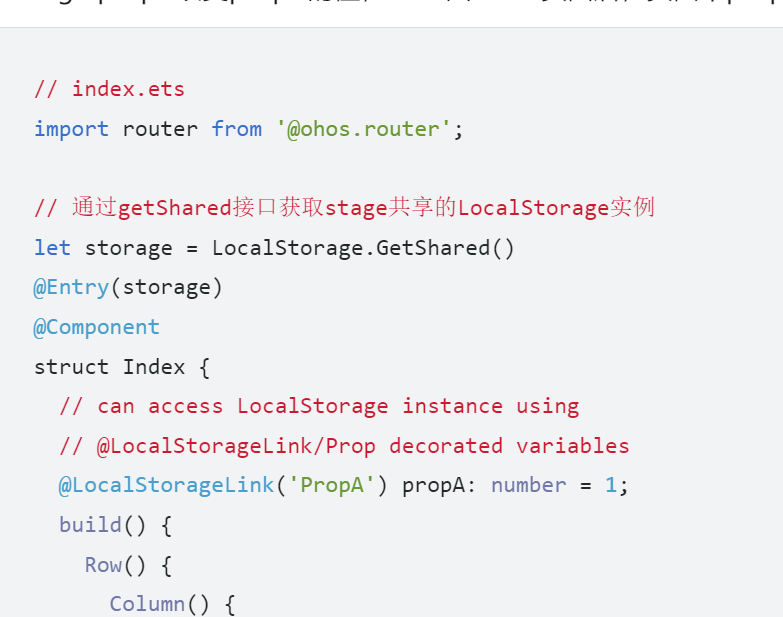


1. 从UI内部使用LocalStorage



1. @LocalStorageProp和LocalStorage单向同步的简单场景
2. @LocalStorageLink和LocalStorage双向同步的简单场景
3. 兄弟节点之间同步状态变量
4. 将LocalStorage实例从UIAbility共享到一个或多个视图。





#### AppStorage：应用全局的UI状态存储

AppStorage是应用全局的UI状态存储，是和应用的进程绑定的，由UI框架在应用程序启动时创建，为应用程序UI状态属性提供中央存储。

和AppStorage不同的是，LocalStorage是页面级的，通常用于页面内的数据共享。而AppStorage是应用级的全局状态共享，还相当于整个应用的中枢，持久化数据PersistentStorage和环境变量Environment都是通过和AppStorage中转，才可以和UI交互。

概述：AppStorage是在应用启动的时候会被创建的单例。它的目的是为了提供应用状态数据的中心存储，这些状态数据在应用级别都是可访问的。AppStorage将在应用运行过程保留其属性。属性通过唯一的键字符串值访问。AppStorage可以和UI组件同步，且可以在应用业务逻辑中被访问。AppStorage中的属性可以被双向同步，数据可以是存在于本地或远程设备上，并具有不同的功能。

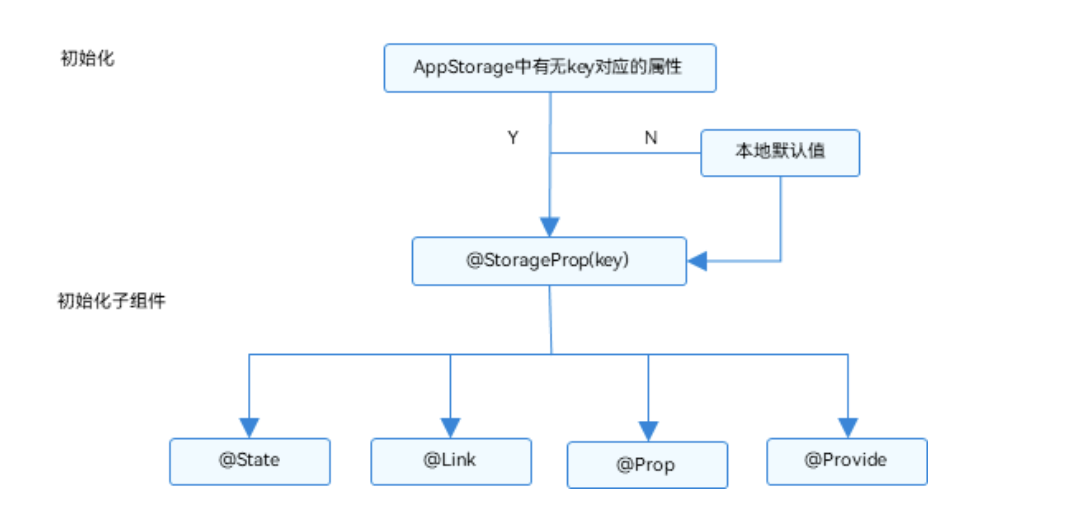
@StorageProp

如果要建立AppStorage和自定义组件的联系，需要使用@StorageProp和@StorageLink装饰器。@StorageProp(key)/@StorageLink(key)装饰组件内的变量，key标识了AppStorage的属性。当自定义组件初始化的时候，@StorageProp(key)/@StorageLink(key)装饰的变量会通过给定的key，绑定再AppStorage对应的属性，完成初始化。本地初始化是必要的，因为无法保证AppStorage一定存在给定的key，这取决于应用逻辑，是否在组件初始化之前在AppStorage实例中存入对应的属性。

装饰器规则：

|  |  |
| --- | --- |
| @StorageProp | 说明 |
| 装饰器参数 | key：字符串常量 |
| 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 |
| 同步类型 | 单向同步：从AppStorage的对应属性到组件的状态变量，允许本地修改，但是会被AppStorage给定属性变化将覆盖本地 |
| 被装饰变量的初始值 | 必须指定，如果AppStorage实例中不存在属性，则作为初始默认值，并存入AppStorage |

传递/访问规则：不允许从父节点初始化和更新



观察变化和行为表现

观察变化：

-当装饰器的数据类型为boolean、string、number类型时，可以观察到数值的变化

-当装饰器的数据类型为class或者Object时，可以观察到赋值和属性赋值的变化

-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

框架行为：

-当@StorageProp(key)装饰的数值改变被观察到时，修改不会被同步回AppStorage对应属性键值key的属性中。

-当前@StorageProp(key)单向绑定的数据会被修改，即仅限于当前组件的私有成员变量，其他绑定该key的数据不会同步改变

-当@StorageProp(key)装饰的参数本身时状态变量，它的改变虽然不会同步回AppStorage中，但是会引起所属的自定义组件的重新渲染。

-当AppStorage中key对应的属性发生改变时，会同步给所有@StorageProp(key)装饰的数据，@StorageProp(key)本地的修改将被覆盖。

@StorageLink

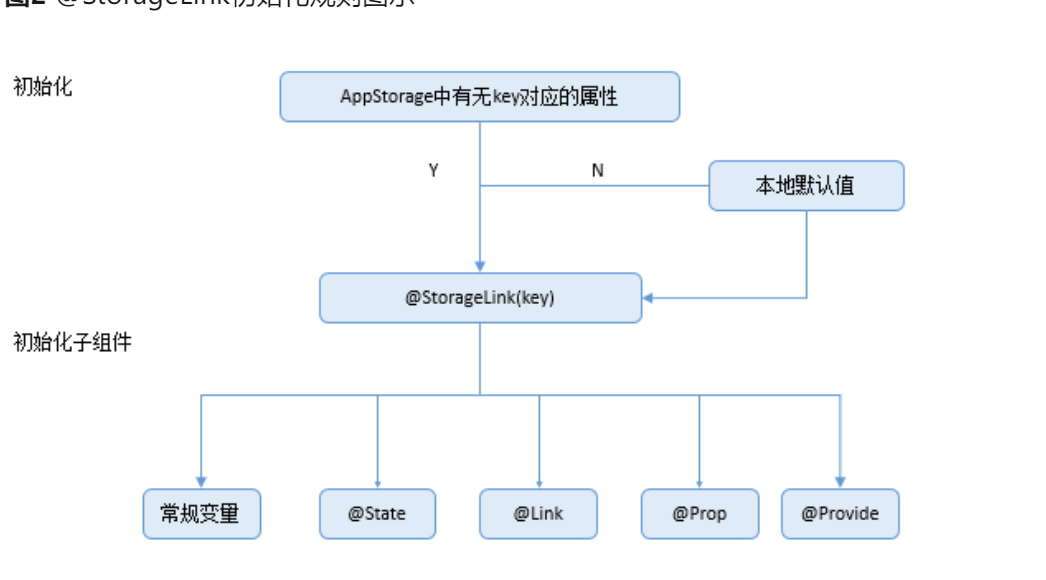
@StorageLink(key)是和AppStorage中key对应的属性建立双向数据同步。

1. 本地修改发生，该修改会被写回AppStorage中
2. AppStorage中的修改发生后，该修改会被同步到所有绑定AppStorage对应的key的属性上，包括单向(@StorageProp和通过Prop创建额单向绑定变量)、双向（@StorageLink、@Link）和其他实例

装饰器使用规则：

|  |  |
| --- | --- |
| @StorageLink变量装饰器 | 说明 |
| 装饰器参数 | Key：字符串常量 |
| 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 |
| 同步类型 | 双向同步：从AppStorage中对应属性到自定义组件，从自定义组件到AppStorage对应属性。 |
| 被装饰变量的初始值 | 必须指定，如果AppStorage实例中不存在属性，则作为初始化默认值，并存入AppStorage中。 |

变量传递/访问规则：禁止从父组件初始化和更新



观察变化和行为表现：

观察变化：

-当装饰器的数据类型为boolean、string、number类型时，可以观察到数值的变化

-当装饰器的数据类型为class或者Object时，可以观察到赋值和属性赋值的变化

-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

框架行为：

-当@StorageLink(key)装饰的数值改变被观察到时，修改不会被同步回AppStorage对应属性键值key的属性中。

-AppStorage中属性键值key对应的数据一旦改变，属性键值key绑定的所有的数据(包括双向@StorageLink和单向@StorageProp)都将同步修改。

-当@StorageLink(key)装饰的数据本身是状态变量，它的改变不仅仅会同步会AppStorage中，还会引起所属的自定组件的重新渲染。

使用场景：

--从应用逻辑使用AppStorage和LocalStorage



--从UI内部使用AppStorage和LocalStorage



不建议开发者使用@StorageLink和AppStorage的双向同步的机制来实现事件通知，AppStorage是和UI相关的数据存储，改变会带来UI的刷新，相对于一般的事件通知，UI刷新的成本较大。

限制条件

--在AppStorage中创建属性后，调用PersistentStorage.persistProp()接口时，会使用在AppStorage中已经存在的值，并覆盖PersistentStorage中的同名属性，所以建议要使用相反的调用顺序

--如果在AppStorage中已经创建属性后，再调用Environment.envProp()创建同名的属性，会调用失败。因为AppStorage已经有同名属性，Environment环境变量不会再写入AppStorage中，所以建议AppStorage中属性不要使用Environment预置环境变量名。

--状态装饰器装饰的变量，改变会引起UI的渲染更新，如果改变的变量不是用于UI更新，只是用于消息传递，推荐使用 emitter方式。

#### PersistentStorage:持久化存储UI状态

概述：PersistentStorage将选定的AppStorage属性保留在设备磁盘上。应用程序通过API，决定哪些AppStorage属性应借助PersistentStorage持久化。UI和业务逻辑不直接访问PersistentStorage中的属性，所有属性访问都是对AppStorage的访问，AppStorage中的更改会自动同步到PersistentStorage。PersistentStorage和AppStorage中的属性建立双向同步。应用开发通常通过AppStorage访问PersistenetStorage，另外还有一些接口可以用于管理持久化属性，但是业务逻辑始终是通过AppStorage获取和设置属性。

限制条件：

允许类型：

-number,string,Boolean,enum等简单类型

-可以被JSON.stringfy()和JSON.parse()重构的对象。例如内置Date,Map不可以

不允许类型：

-不支持嵌套对象（对象数组，对象的属性是对象等），不支持undefined和null

PersistentStorage的持久化变量最好是小于2kb的数据，不要大量的数据持久化，因为PersistentStorage写入磁盘的操作是同步的，大量的数据本地化读写会同步在UI线程中执行，影响UI渲染性能。如果开发者需要存储大量的数据，建议使用数据库api。

PersistentStorage只能在UI页面内使用，否则将无法持久化数据。

支持接口：

PersistProp

static PersistProp<T>(key:string,defaultValue:T):void

将AppStorage中key对应得属性持久化到文件中。该接口的调用通常在访问AppStorage之前。

确定属性的类型和值的顺序如下：

1. 如果PersistentStorage文件中存在key对应的属性，在AppStorage中创建对应的propName，并用在PersistentStorage中找到的key的属性初始化。
2. 如果PersistentStorage文件中没有查询到key对应的属性，则在AppStorage中查找key对应的属性。如果找到key对应的属性，则将该属性持久化。
3. 如果AppStorage也没查到key对应的属性，则在AppStorage中创建key对应的属性。用defaultValue初始化其值，并将该属性持久化。

-如果AppStorage中有该属性，则会使用其值，覆盖掉PersistentStorage文件中的值。由于AppStorage是内存内数据，该行为会导致数据丧失持久化的能力。



DeleteProp

static DeleteProp(key:string):void

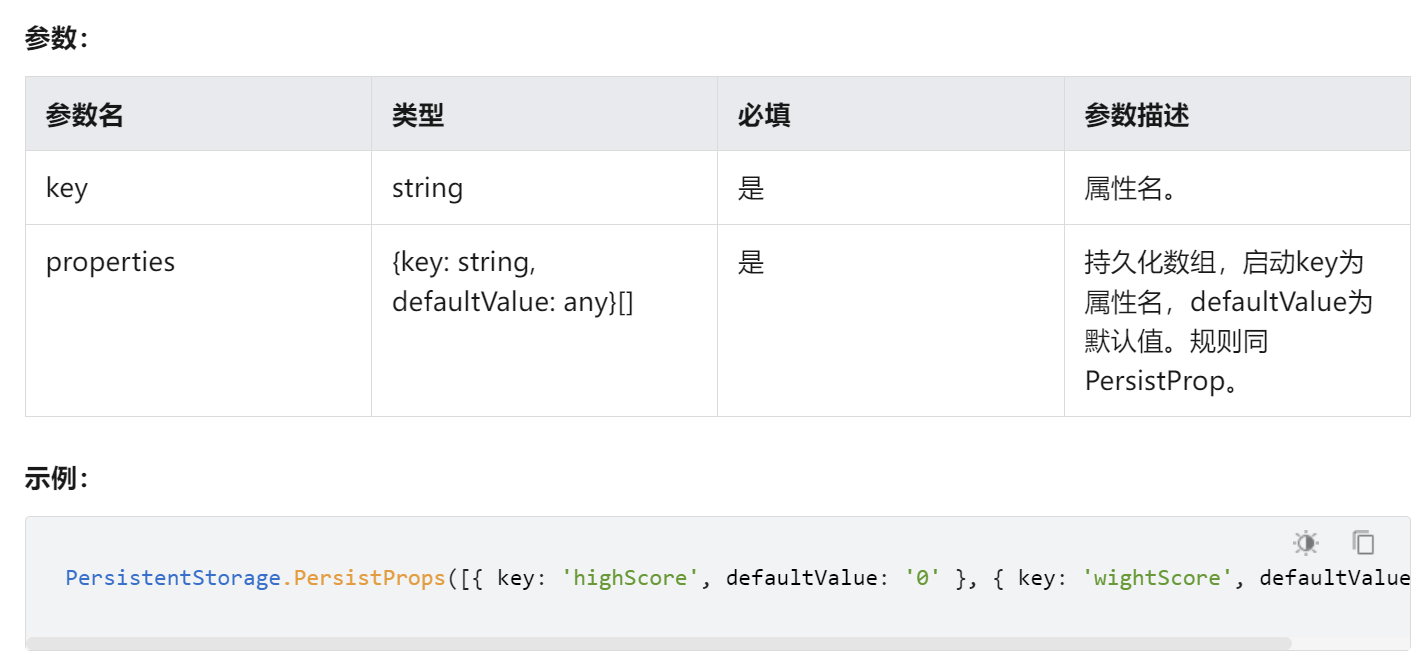
将key对应的属性从PersistentStorage删除，后续AppStorage的操作，对PersistentStorage不会再有影响。



PersistProp

static PersistProps(properties:{key:string,defaultValue:any;}[]):void

行为和PersistProp类似，不同在于可以一次性持久化多个数据，适合在应用启动的时候初始化。





使用场景：

1. 从AppStorage中访问PersistentStorage初始化的属性

-初始化PersistentStorage:

PersistentStorage.PersistProp(‘aProp’,47);

-从AppStorage获取对应属性：

AppStorage.Get(‘aProp’);// return 47

或在组件内部定义：

@StorageLink(‘aProp’) aProp:number = 48



1. 在PersistentStorage之前访问AppStorage中的属性



1. 在PersistentStorage之后访问AppStorage中的属性



#### Environment：设备环境查询

开发者如果需要应用程序运行的设备的环境参数，以此来作出不同的场景判断，比如多语言，黑暗模式，需要使用Environment设备环境查询。

Environment是ArkUI框架在应用程序启动时创建的单例对象。它为AppStorage提供了一系列描述应用程序状态的属性。Environment的所有属性都是不可变的，所有的属性都是简单类型。

Environment：参数

|  |  |  |
| --- | --- | --- |
| 键 | 数据类型 | 描述 |
| accessibilityEnabled | boolean | 获取无障碍屏幕是否启用 |
| colorMode | ColorMode | 色彩模型类型：选项为ColorMode.LIGHT：浅色，  ColorMode.DARK：深色 |
| fontScale | number | 字体大小比例，范围[0.85,1.45] |
| fontWeightScale | number | 字体粗细程度，范围[0.6,1.6] |
| layoutDirection | LayoutDirection | 布局方向类型：包括  LayoutDirection.LTR：从左到右  LayoutDireciton.RTL:从右到左 |
| languageCode | string | 当前系统语言值，取值必须为小写字母，例如zh |

使用场景：

从UI中访问Environment参数

-使用Environment.EnvProp将设备运行的环境变量存入AppStorage中:

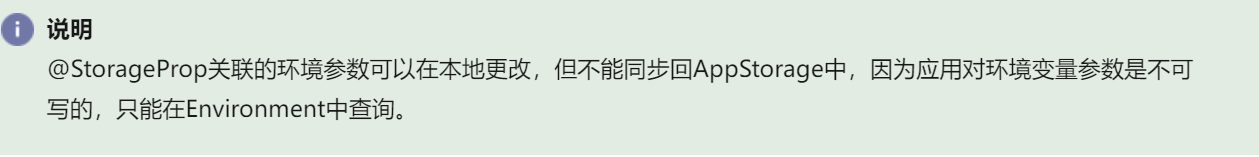
Environment.EnvProp(‘languageCode’,’en’);//将设备的语言code存入AppStorage，默认值为en

-可以使用@StorageProp链接到Component中

@StorageProp(‘languageCode’) lang:string = ‘en’;

设备环境到Component的更新链：Environment->AppStorage->Component





应用逻辑使用Environment

//使用Environment.EnvProp将设备运行languageCode存入AppStorage中

Environment.EnvProp(‘languageCode’,’en’);

//从AppStorage中获取单向绑定的languageCode的变量

Const lang: SubscribedAbstractProperty<string> = AppStorage.Prop(‘languageCode’);

If(lang.get()===’zh’){

Console.log(‘你好’)

} else {

Console.log(‘Hello’)

}

其他状态管理概述：

--@Watch用于监听状态变量的变化

--$$运算符:给内置组件提供TS变量的引用，使得TS变量和内置组件的内部状态保持同步

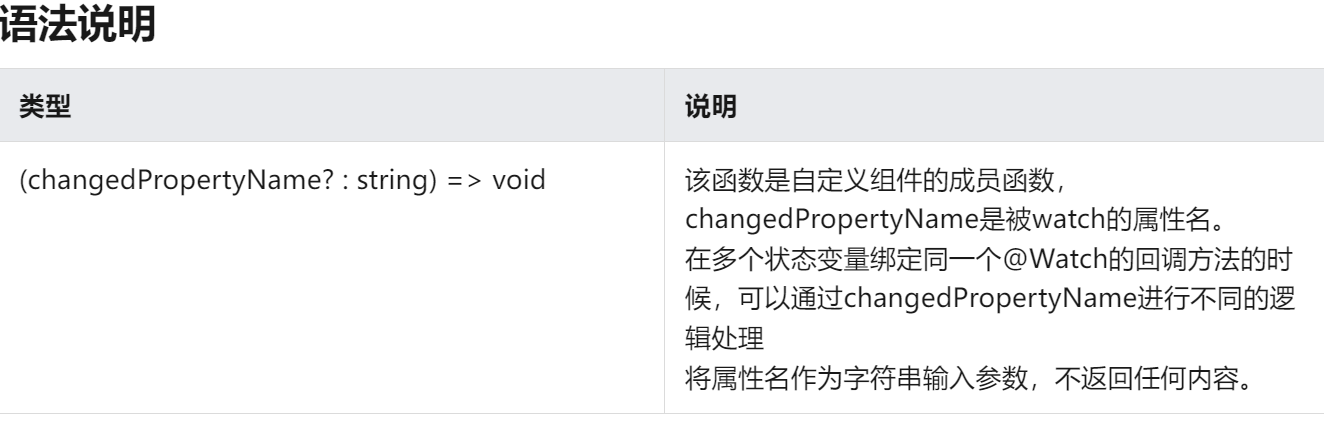
#### @Watch:

@Watch应用于对状态变量的监听。如果开发者需要关注某个状态变量的值是否改变，可以使用@Watch为状态变量设置回调函数

概述：@Watch用于监听状态变量的变化，当状态变量变化时，@Watch的回调方法将被调用。@Watch在ArkUI框架内部判断数值有无更新使用的是严格相等（===），遵循严格相等规范。当在严格相等为false的情况下，就会触发@Watch的回调。

装饰器说明：





观察变化和行为表现：

1. 当观察到状态变量的变化（包括双向绑定的AppStorage和LocalStorage中对应的key发生的变化）的时候，对应的@Watch的回调方法将被触发。
2. @Watch方法在自定义组件的属性变更之后同步执行
3. 如果在@Watch的方法里改变了其他的状态变量，也会引起状态变更和@Watch的执行。
4. 在第一次初始化的时候，@Watch装饰的方法不会被调用，只有在后续状态改变时，才会调用@Watch回调方法。

限制条件：

-建议开发者避免无限循环。循环可能是因为在@Watch的回调方法里直接或者间接地修改了同一个状态变量引起的。为了避免循环的产生，建议不要在@Watch的回调方法里修改当前装饰的状态变量；

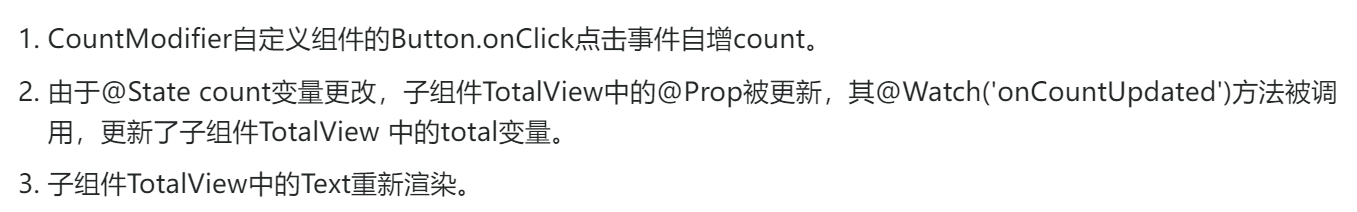
-开发者应关注性能，属性值更新函数会延迟组件的重新渲染（具体请见上面的行为表现），因此，回调函数应仅执行快速运算；

-不建议在@Watch函数中调用async await，因为@Watch设计的用途是为了快速的计算，异步行为可能会导致重新渲染速度的性能问题。

使用场景：

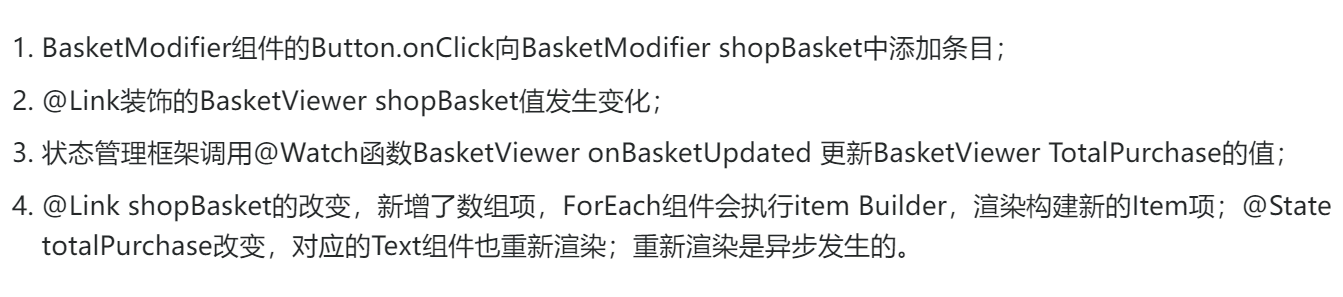
@Watch和自定义组件更新





@Watch与Link组合使用





#### $$运算符

$$运算符为系统内置组件提供TS变量的引用，使得TS变量和系统内置组件的内部状态保持同步。

使用规则：

-当前$$支持基础类型变量，以及@State、@Link、@Prop装饰的变量

-当前$$仅支持Refresh组件的refreshing参数

-$$绑定的变量变化时，会触发UI的同步刷新。



### 渲染控制概述：

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。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，这些渲染控制语句包括控制组件是否显示的条件渲染语句，基于数组数据快速生成组件的循环渲染语句以及针对大数据量场景的数据懒加载语句。

-If/else条件渲染

-ForEach循环渲染

-LazyForEach数据懒加载

#### If/else条件渲染：

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：

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

--删除此前构建的所有子组件。

--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。

#### -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。

接口描述：

ForEach(

arr: Array,

itemGenerator: (item: any, index: number) => void,

keyGenerator?: (item: any, index: number) => string

)

参数：



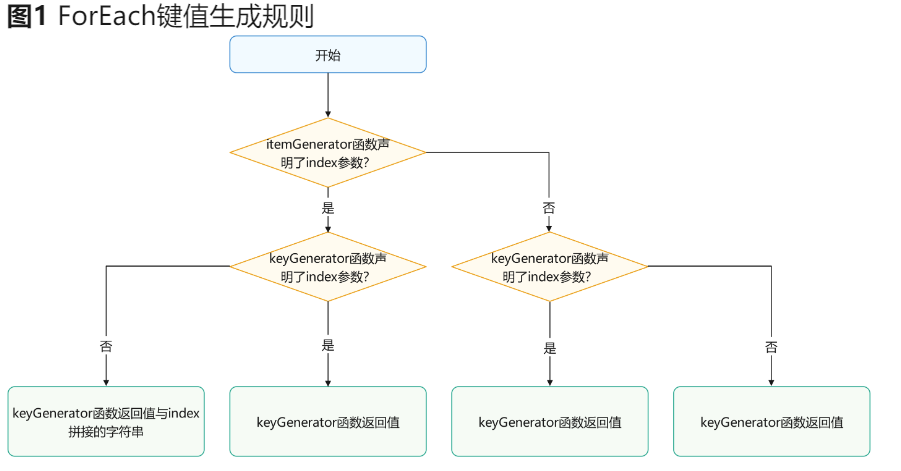
Ps:

1.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。

2.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。



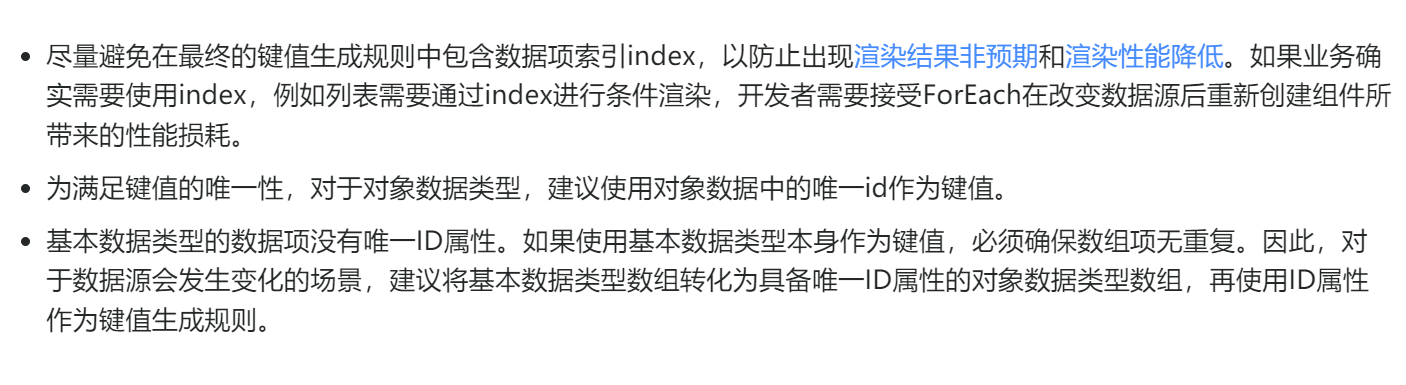
组件创建规则：

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

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。

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。

使用建议：



#### LazyForEach数据懒加载：

LazyForEach从提供的数据源中按需迭代数据，并在每次迭代的过程中创建相应的组件。当滚动容器中使用了LazyForeach，框架会根据滚动容器可视区域按需创建数组，当组件滑出可视区域外时，框架会进行组件销毁回收以降低内存占用。

接口描述：

LazyForEach(

dataSource:IDataSource, //需要进行数据迭代的数据源

itemGenerator: (item:any, index: number)=>void, //子组件生成函数

keyGenerator?: (item:any, index:number) => string // 键值生成函数

):void

参数：

|  |  |  |  |
| --- | --- | --- | --- |
| 参数名 | 参数类型 | 必填 | 参数描述 |
| dataSource | IDataSource | 是 | LazyForEach数据源，需要开发者实现相关接口 |
| itemGenerator | (item:any,  index:number)=>void | 是 | 子组件生成函数，为数组中的每一个数据项创建一个子组件。说明：item是当前数据项,index是数据项索引值。itemGenerator的函数体必须使用大括号{…}。ItemGenerator每次迭代只能并且必须生成一个子组件itemGenerator中可以使用if语句，但是必须保证if语句每个分支都会创建一个相同类型的子组件。itemGenerator中不允许使用ForEach和LazyForEach语句。 |
| keyGenerator | (item:any,  index:number)=>string | 否 | 键值生成函数，用于给数据源中的每一个数据项生成唯一且固定的键值。当数据项在数组中的位置更改时，其键值不得更改，当数组中的数据项被新项替换时，被替换项的键值和新项的键值必须不同。键值生成器的功能是可选的，但是，为了使开发框架能够更好的识别数组更改，提高性能，建议提供。如将数组反向时，如果没有提供键值生成器，则LazyForEach中的所有节点都将重建。说明：item是当前数据项，index是数据项索引值。数据源中的每一个数据项生成的键值不能重复。 |

IDataSource类型说明

Interface IDataSource {

totalCount():number;//获得数据总数

getData(index:number):Object;//获取索引值对应的数据

registerDataChangeListener(listener:DataChangeListener):void;//注册数据改变的监听器

unregisterDataChangeListener(listener:DataChangeListener):void;//注销数据改变的监听器。

}

|  |  |  |
| --- | --- | --- |
| 接口声明 | 参数类型 | 说明 |
| totalCount():number | - | 获得数据总数 |
| getData(index:number):any | number | 获取索引值index对应的数据。Index:获取数据对应的索引值 |
| registerDataChangeListener(  listener:DataChangeListener):void | DataChangeListener | 注册数据改变的监听器。listener:数据变化监听器 |
| unregisterDataChangeListener(  listener:DataChangeListener):void | DataChangeListener | 注销数据改变的监听器。listener:数据变化监听器 |

DataChangeListener类型说明

Interface DataChangeListener {

onDataReload():void;//重新加载数据完成后调用

onDataAdded(index:number):void;//添加数据完成后调用，废弃

onDataMoved(from:number,to:number):void;//数据移动起始位置与数据移动目标位置交换完成后调用，废弃

onDataDeleted(index:number):void;//删除数据完成后调用，废弃

onDataChanged(index:number):void;//改变数据完成后调用，废弃

onDataAdd(index:number):void;//添加数据完成后调用，建议使用

onDataMove(from:number,to:number):void;//数据移动起始位置与数据移动目标位置交换完成后调用，建议使用

onDataDelete(index:number):void;//删除数据完成后调用，建议使用

onDataChange(index:number):void;//改变数据完成后调用，建议使用

}

|  |  |  |
| --- | --- | --- |
| 接口声明 | 参数类型 | 说明 |
| onDataReload():void | - | 通知组件重新加载所有数据。键值没有变化的数据项会使用原先的子组件，键值发生变化的会重建子组件。 |
| onDataAdd(index:number):void | number | 通知组件index的位置有数据添加。Index：数据添加位置的索引值 |
| onDataMove(from:number,to:  number):void | from:number,to:number | 通知组件数据有移动。From：数据移动的起始位置，to：数据移动目标位置  说明：数据移动前后键值要保持不变，如果键值有变化，应使用删除数据和新增数据接口。 |
| onDataDelete(index:number):  void | number | 通知组件删除index位置的数据并刷新LazyForEach的展示内容。index：数据删除位置的索引值。 |
| onDataChange(index:number):  void | number | 通知组件index的位置有数据有变化。Index:数据变化位置的索引值。 |

使用限制

-LazyForEach必须在容器组件内使用，仅有List、Grid、Swiper以及WaterFlow组件支持数据懒加载（可配置cachedCount属性，即只加载可视部分以及前后少量数据用于缓存），其他组件仍然是一次性加载所有的数据。

-LacyForEach在每次迭代中，必须创建且只允许创建一个子组件。

-生成的子组件必须是允许包含在LazyForEach父容器组件中的子组件。

-允许LazyForEach包含在if/else条件渲染语句中，也允许LazyForEach中出现if/else条件渲染语句。

-键值生成器必须针对每个数据项生成唯一的值，如果键值相同，将导致键值相同的UI组件渲染出问题。

-LazyForEach必须使用DataChangeListener对象来进行更新，第一个参数dataSource使用状态变量时，状态变化改变不会触发LazyForEach的UI刷新。

-为了高性能渲染，通过DataChangeListener对象的onDataChange方法来更新UI时，需要生成不同于原来的键值来触发组件刷新。

键值生成规则：

在LazyForEach循环渲染过程中，系统会为每个item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。

LazyForEach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 viewId + '-' + index.toString(); }, viewId在编译器转换过程中生成，同一个LazyForEach组件内其viewId是一致的。

组件创建规则：在确定键值生成规则后，LazyForEach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：LazyForEach首次渲染和LazyForEach非首次渲染。

首次渲染：在LazyForEach首次渲染时，会根据键值首次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。

Class BasicDataSource implements IDataSource {

private listeners:DataChangeListener[] = []

private originDataArray:string[] = []

public totalCount():number {

return 0

}

public getData(index:number):string{

return this.originDataArray[index]

}

//为LazyForEach组件向数据源处添加listener监听

registerDataChangeListener(listener:DataChangeListener):void{

if(this.listeners.indexOf(listener)<0){

console.log(‘add listener’)

this.listeners.push(listener)

}

}

//为LazyForEach组件向数据源处取消listener监听

unregisterDataChangeListener(listener:DataChangeListener):void{

const pos = this.listenenrs.indexOf(listener)

if(pos>=0){

console.info(‘remove listener’)

this.listeners.splice(pos,1)

}

}

//通知LazyForEach组件需要重新加载所有子组件

notifyDataReload():void{

this.listeners.forEach(listener=>{

listener.onDataReaload();

})

}

//通知LazyForEach组件需要在index对应索引处添加子组件

notifyDataAdd(index:number):void{

this.listeners.forEach(listener=>{

listener.onDataAdd(index);

})

}

//通知LazyForEach组件在index对应索引处数据有变化，需要重建该子组件

notifyDataChange(index:number):void{

this.listeners.forEach(listener=>{

listener.onDataChange(index)

})

}

//通知LazyForEach组件需要在index对应索引处删除该子组件

notifyDataDelete(inde:number):void{

this.listeners.forEach(listener=>{

listener.onDataDelete(index)

})

}

//通知LazyForEach组件将from索引和to索引处的子组件进行交换

notifyDataMove(from:number,to:number):void{

this.listeners.forEach(listener=>{

listener.onDataMove(from,to)

})

}

}

Class MyDataSource extends BasicDataSource {

private dataArray:string[] = []

public totalCount():number{

return this.dataArray.length

}

public getData(index:number):string{

return this.dataArray[index]

}

public addData(index:number,data:string):void{

this.dataArray.splice(index,0,data)

this.notifyDataAdd(index)

}

public pushData(data:string):void{

this.dataArray.push(data)

this.notifyDataAdd(this.dataArray.length-1);

}

public deleteData(index: number): void {

this.dataArray.splice(index, 1);

this.notifyDataDelete(index);

}

public moveData(from: number, to: number): void {

let temp: string = this.dataArray[from];

this.dataArray[from] = this.dataArray[to];

this.dataArray[to] = temp;

this.notifyDataMove(from, to);

}

public changeData(index: number, data: string): void {

this.dataArray.splice(index, 1, data);

this.notifyDataChange(index);

}

public reloadData(): void {

this.notifyDataReload();

}

public modifyAllData(): void {

this.dataArray = this.dataArray.map((item: string) => {

return item + '0';

})

}

}

# ArkTS开发部分

### 应用模型

#### 应用模型概述

应用模型是HarmonyOS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。

应用模型构成要素：

1、应用组件

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。应用组件提供生命周期的回调函数，开发者通过应用组件的生命周期回调感知应用的状态变化。应用开发者在编写应用时，首先需要编写的就是应用组件，同时还需编写应用组件的生命周期回调函数，并在应用配置文件中配置相关信息。这样，操作系统在运行期间通过配置文件创建应用组件的实例，并调度它的生命周期回调函数，从而执行开发者的代码。

2、应用进程模型

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。

3、应用线程模型

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。

4、应用任务管理模型

应用任务管理模型定义任务(Mission)的创建和销毁方式，以及任务与组件间的关系。HarmonyOS应用任务管理由系统应用负责，三方应用无需关注。

5、应用配置文件

应用配置文件中包含应用配置信息、应用组件信息、权限信息、开发者自定义信息等，这些信息在编译构建、分发和运行阶段分别提供编译工具、应用市场和操作系统使用。

Stage模型：HarmonyOS 3.1 Developer Preview版本开始新增的模型，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。

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。

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。

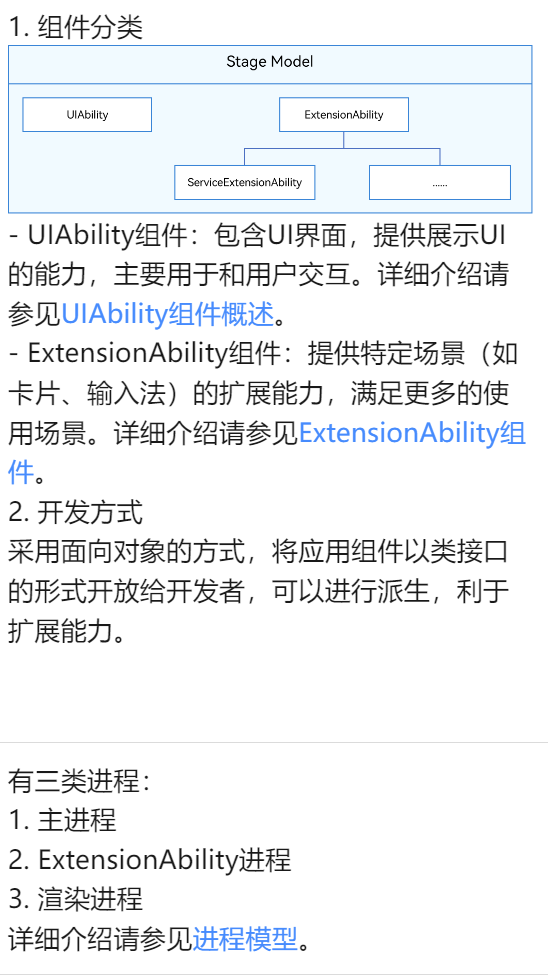
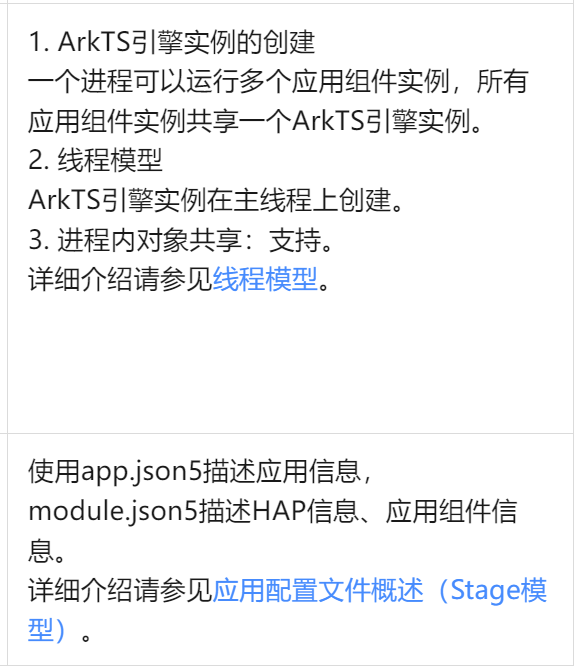
平衡应用能力和系统管控成本：

Stage模型重新定义应用能力的边界，平衡应用能力和系统管控成本。

-提供特定场景（如卡片、输入法）的应用组件，以便满足更多的使用场景。

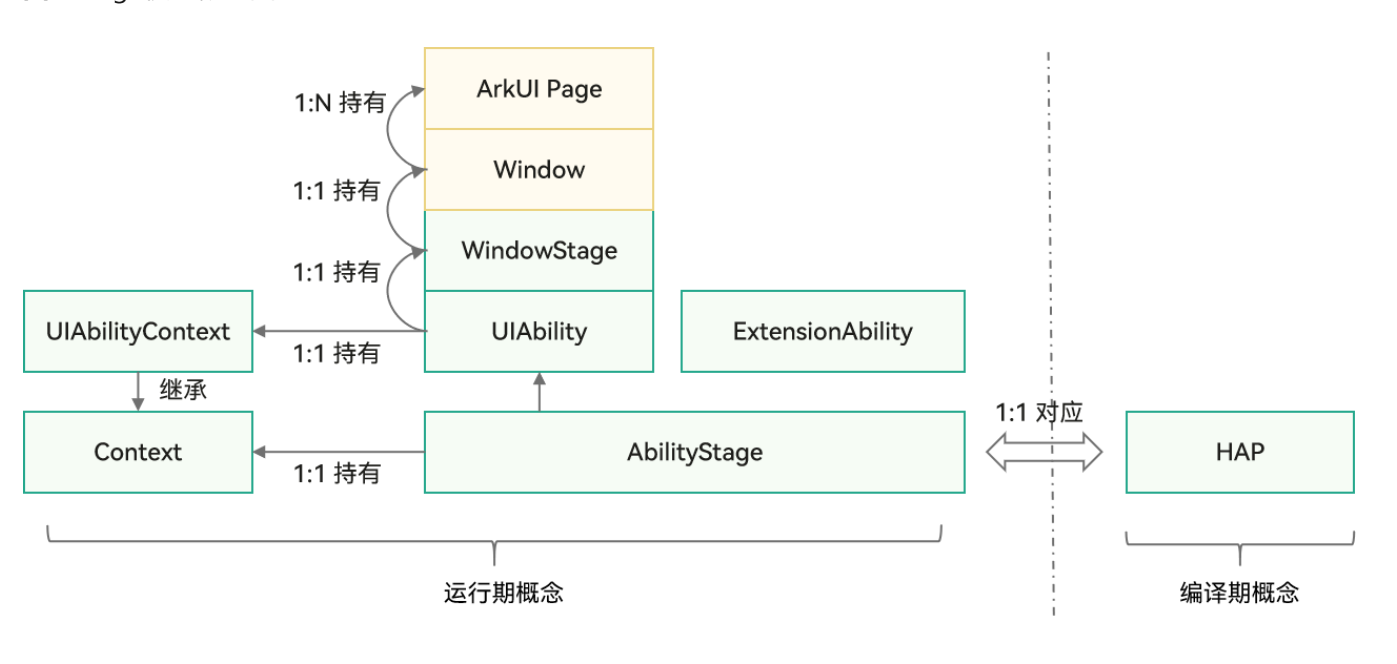
-规范化后台进程管理：为保障用户体验，Stage模型对后台应用进程进行了有序治理，应用程序不能随意驻留在后台，同时应用后台行为受到严格管理，防止恶意应用行为。

Stage模型与FA模型最大的区别在于：Stage模型中，多个应用组件共享同一个ArkTS引擎实例；而FA模型中，每个应用组件独享一个ArkTS引擎实例。因此在Stage模型中，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。Stage模型作为主推的应用模型，开发者通过它能够更加便利地开发出分布式场景下的复杂应用。



#### Stage模型开发

Stage模型概念图：



-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中的代码首次加载到进程中的时候，系统会先创建AbilityStage实例

-UIAbility组件和ExtensionAbility组件

Stage模型提供UIAbility和ExtensionAbility两种类型的组件，这两种组件都有具体的类承载，支持面向对象的开发方式。

--UIAbility组件是一种包含用户界面的应用组件，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。同时用户可以通过返回键返回到瀑布流页面。UIAbility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派生，而是需要使用ExtensionAbility的派生类。目前ExtensionAbility有用于卡片场景的FormExtensionAbility,用于闲时任务场景的WorkSchedulerExtensionAbility等多种派生类，这些派生类都是基于特定场景提供的。例如，用户在桌面创建应用的卡片，需要应用开发者从FormExtensionAbility派生，实现其中的回调函数，并在配置文件中配置该能力。ExtensionAbility派生类实例由用户触发创建，并由系统管理生命周期。在Stage模型上，普通应用开发者不能自定义服务，而需要根据自身的业务场景通过ExtensionAbility的派生类来实现。

-windowStage

每个UIAbility类实例都会与一个windowStage类实例绑定，WindowStage类起到了应用进程内窗口管理器的作用，它包含一个窗口。也就是是说UIAbility通过WindowStage持有了一个窗口，该窗口为ArkUI提供了绘制区域。

-Context

在Stage模型上，Context及其派生类向开发者提供在允许其可以调用的各种能力。UIAbility组件和各种ExtensionAbility派生类都有各自不同的Context类，他们都继承自基类Context，但是各自又根据所属组件，提供不同的能力。

Stage模型开发流程：



Stage模型支持对组件配置入口图标和入口标签。入口图标和入口标签会显示在桌面上。

入口图标需要在module.json5配置文件中配置，在abilities标签下面有icon标签。例如希望在桌面上显示该UIAbility的图标，则需要在skills标签下面的entities中添加"entity.system.home"、actions中添加"action.system.home"。同一个应用有多个UIAbility配置上述字段时，桌面上会显示出多个图标，分别对应各自的UIAbility。

#### UIAbility组件

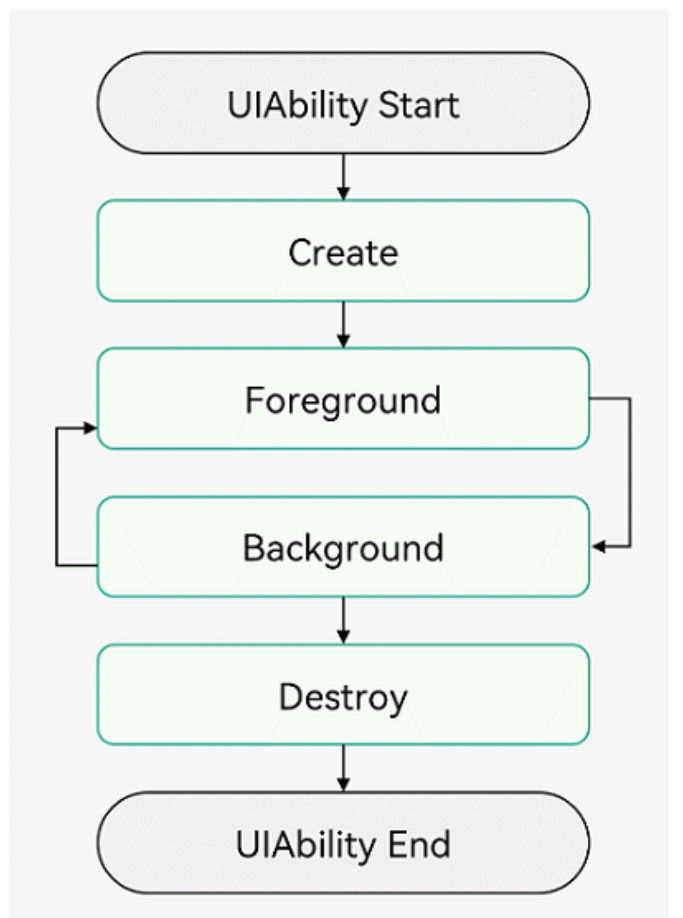
概述：UIAbility组件是一种包含UI界面的应用组件，主要用于和用户交互。

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UIAbility组件中可以通过多个页面来实现一个功能模块。每一个UIAbility组件实例，都对应于一个最近任务列表中的任务。

声明配置：为使应用能够正常使用UIAbility，需要在module.json5配置文件的abilities标签中声明UIAbility的名称、入口、标签等相关信息。

UIAbility生命周期

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状态：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。可以在该回调中进行页面初始化操作，例如变量定义资源加载等，用于后续的UI界面展示。

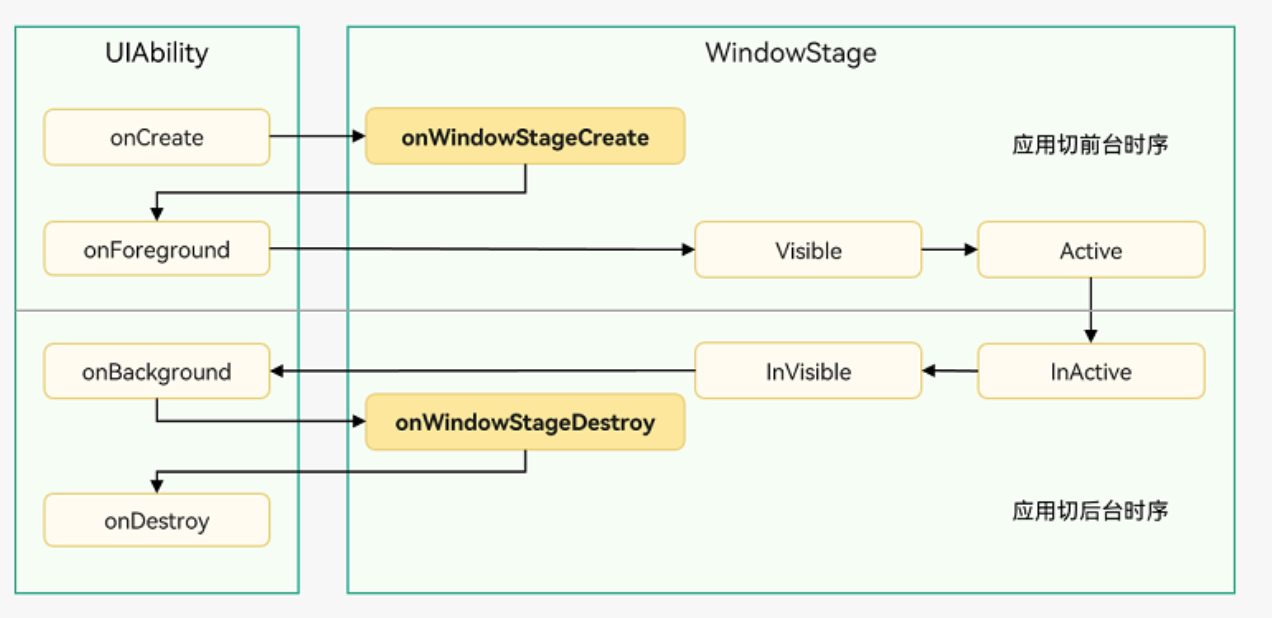
Foreground：onForeground()回调，在UIAbility的UI界面可见之前，如UIAbility切换至前台时触发。可以在onForeground()回调中申请系统需要的资源，或者重新申请在onBackground()中释放的资源。

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

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。

onWindowStageCreate：UIAbility实例创建完成之后，在进入Foreground之前，系统会创建一个WindowStage。WindowStage创建完成后会进入onWindowStageCreate()回调，可以在该回调中设置UI界面加载、设置WindowStage的事件订阅

onWindowStageDestroy：在UIAbility实例销毁之前，则会先进入onWindowStageDestroy()回调，可以在该回调中释放UI界面资源。例如在onWindowStageDestroy()中注销获焦/失焦等WindowStage事件。



UIAbility组件启动模式

UIAbility提供了四种启动模式：singleton、multiton、standard、specified

1、singleton启动模式为单实例模式，也是默认情况下的启动模式。

2、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,并销毁原来的UIAbility实例。

3、standard启动模式为标准实例模式，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,并保留原来的UIAbility实例。

4、specified启动模式为指定实例模式，在UIAbility实例创建之前，允许开发者为该实例创建一个唯一的字符串Key，创建的UIAbility实例绑定Key之后，后续每次调用startAbility()方法时，都会询问应用使用哪个Key对应的UIAbility实例来响应startAbility()请求。运行时由UIAbility内部业务决定是否创建多实例，如果匹配有该UIAbility实例的Key，则直接拉起与之绑定的UIAbility实例，否则创建一个新的UIAbility实例。

specified启动模式配置：

1、将目标Ability的module.json5配置文件的"launchType"字段配置为"specified"。

2、在EntryAbility中，调用startAbility()方法时，在want参数中，增加一个自定义参数来区别UIAbility实例，例如增加一个"instanceKey"自定义参数。

3、由于目标Ability的启动模式配置为了指定实例启动模式，在目标Ability启动之前，会先进入其对应的AbilityStage的onAcceptWant()生命周期回调中，解析传入的want参数，获取"instanceKey"自定义参数。根据业务需要通过AbilityStage的onAcceptWant()生命周期回调返回一个字符串Key标识。如果返回的Key对应一个已启动的UIAbility，则会将之前的UIAbility拉回前台并获焦，而不创建新的实例，否则创建新的实例并启动。



4、配置自定义abilityStage的路径。

UIAbility基本用法

-指定UIAbility的启动页面

可以在UIAbility的onWindowStageCreate()生命周期回调中，通过WindowStage对象的loadContent()方法设置启动页面。

-获取UIAbility的上下文信息

UIAbility类拥有自身的上下文信息，该信息为UIAbilityContext类的实例，UIAbilityContext类拥有abilityInfo、currentHapModuleInfo等属性。通过UIAbilityContext可以获取UIAbility的相关配置信息，如包代码路径、Bundle名称、Ability名称和应用程序需要的环境状态等属性信息，以及可以获取操作UIAbility实例的方法（如startAbility()、connectServiceExtensionAbility()、terminateSelf()等）。

1、在UIAbility中可以通过this.context获取UIAbility实例的上下文信息。

2、在页面中获取UIAbility实例的上下文信息，包括导入依赖资源context模块和在组件中定义一个context变量两个部分。



UIAbility组件与UI的数据同步

基于HarmonyOS的应用模型，可以通过以下两种方式来实现UIAbility组件与UI之间的数据同步。

-使用EventHub进行数据通信：基于发布订阅模式来实现，事件需要先订阅后发布，订阅者收到消息后进行处理。

-使用globalThis进行数据同步：ArkTS引擎实例内部的一个全局对象，在ArkTS引擎实例内部都能访问。

-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。

--使用EventHub进行数据通信

EventHub提供了UIAbility组件/ExtensionAbility组件级别的事件机制，以UIAbility组件/ExtensionAbility组件为中心提供了订阅、取消订阅和触发事件的数据通信能力。

1. 获取eventHub：let eventhub = this.context.eventHub;
2. 定义回调事件，订阅事件：eventHub.on(‘event’,func)
3. 在UI界面中通过eventHub.emit()方法触发该事件，在触发事件的同时，根据需要传入参数信息

--使用globalThis进行数据同步

globalThis是ArkTS引擎实例内部的一个全局对象，引擎内部的UIAbility/ExtensionAbility/Page都可以使用，因此可以使用globalThis全局对象进行数据同步。globalThis为ArkTS引擎实例下的全局对象，可以通过globalThis绑定属性/方法来进行UIAbility组件与UI的数据同步。

--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。使用这些方案可以方便地管理应用状态，提高应用性能和用户体验。其中，AppStorage是一个全局的状态管理器，适用于多个UIAbility共享同一状态数据的情况；而LocalStorage则是一个局部的状态管理器，适用于单个UIAbility内部使用的状态数据。

UIAbility组件间交互（设备）

UIAbility是系统调度的最小单元。在设备内的功能模块之间跳转时，会涉及到启动特定的UIAbility，该UIAbility可以是应用内的其他UIAbility，也可以是其他应用的UIAbility（例如启动三方支付UIAbility）。

设备内UIAbility交互方式：

-启动应用内的UIAbility

-启动应用内的UIAbility并获取返回结果

-启动其他应用的UIAbility

-启动其他应用的UIAbility并获取返回结果

-启动UIAbility的指定页面

启动应用内的UIAbility：

当一个应用内包含多个UIAbility时，存在应用内启动UIAbility的场景。例如在支付应用中从入口UIAbility启动收付款UIAbility。（UIAbility不在同一模块，需要指定moduleName）

启动应用内的UIAbility并获取返回结果:

在一个EntryAbility启动另外一个FuncAbility时，希望在被启动的FuncAbility完成相关业务后，能将结果返回给调用方。例如在应用中将入口功能和帐号登录功能分别设计为两个独立的UIAbility，在帐号登录UIAbility中完成登录操作后，需要将登录的结果返回给入口UIAbility。

启动其他应用的UIAbility:

启动其他应用的UIAbility，通常用户只需要完成一个通用的操作（例如需要选择一个文档应用来查看某个文档的内容信息），推荐使用隐式Want启动。系统会根据调用方的want参数来识别和启动匹配到的应用UIAbility。

隐式和显示Want启动：

显式Want启动：启动一个确定应用的UIAbility，在want参数中需要设置该应用bundleName和abilityName，当需要拉起某个明确的UIAbility时，通常使用显式Want启动方式。

隐式Want启动：根据匹配条件由用户选择启动哪一个UIAbility，即不明确指出要启动哪一个UIAbility（abilityName参数未设置），在调用startAbility()方法时，其入参want中指定了一系列的entities字段（表示目标UIAbility额外的类别信息，如浏览器、视频播放器）和actions字段（表示要执行的通用操作，如查看、分享、应用详情等）等参数信息，然后由系统去分析want，并帮助找到合适的UIAbility来启动。当需要拉起其他应用的UIAbility时，开发者通常不知道用户设备中应用的安装情况，也无法确定目标应用的bundleName和abilityName，通常使用隐式Want启动方式。

启动其他应用的UIAbility并获取返回结果：

当使用隐式Want启动其他应用的UIAbility并希望获取返回结果时，调用方需要使用startAbilityForResult()方法启动目标UIAbility。例如主应用中需要启动三方支付并获取支付结果。

启动UIAbility的指定页面：

一个UIAbility可以对应多个页面，在不同的场景下启动该UIAbility时需要展示不同的页面

-调用方指定UIAbility启动页面：调用方UIAbility启动另外一个UIAbility时，通常需要跳转到指定的页面。例如FuncAbility包含两个页面（Index对应首页，Second对应功能A页面），此时需要在传入的want参数中配置指定的页面路径信息，可以通过want中的parameters参数增加一个自定义参数传递页面跳转信息。

-目标UIAbility首次启动：目标UIAbility首次启动时，在目标UIAbility的onWindowStageCreate()生命周期回调中，解析EntryAbility传递过来的want参数，获取到需要加载的页面信息url，传入windowStage.loadContent()方法。

-目标UIAbility非首次启动：经常还会遇到一类场景，当应用A已经启动且处于主页面时，回到桌面，打开应用B，并从应用B再次启动应用A，且需要跳转到应用A的指定页面。例如联系人应用和短信应用配合使用的场景。

1. 在目标UIAbility中，默认加载的是Index页面。由于当前UIAbility实例之前已经创建完成，此时会进入UIAbility的onNewWant()回调中且不会进入onCreate()和onWindowStageCreate()生命周期回调，在onNewWant()回调中解析调用方传递过来的want参数，并挂载到全局变量globalThis中，以便于后续在页面中获取。

2. 在FuncAbility中，此时需要在Index页面中通过页面路由Router模块实现指定页面的跳转，由于此时FuncAbility对应的Index页面是处于激活状态，不会重新变量声明以及进入aboutToAppear()生命周期回调中。因此可以在Index页面的onPageShow()生命周期回调中实现页面路由跳转的功能。

#### ExtensionAbility组件

ExtensionAbility组件是基于特定场景提供的应用组件，以便满足更多的使用场景。

每一个具体场景对应一个ExtensionAbilityType，各类型的ExtensionAbility组件均由相应的系统服务统一管理，例如InputMethodExtensionAbility组件由输入法管理服务统一管理。当前支持的ExtensionAbility类型有：

FormExtensionAbility：FORM类型的ExtensionAbility组件，用于提供服务卡片场景相关能力。

WorkSchedulerExtensionAbility：WORK\_SCHEDULER类型的ExtensionAbility组件，用于提供延迟任务注册、取消、查询的能力。

#### AbilityStage组件容器

AbilityStage是一个Module级别的容器，应用的HAP在首次加载时会创建一个AbilityStage实例，可以对该Module进行初始化操作。

AbilityStage与Module一一对应，即一个Module拥有一个AbilityStage.

DevEco Studio默认工程中未自动生成AbilityStage，如需要使用AbilityStage的能力，可以手动新建一个AbilityStage文件，并在module.json5中配置srcEntry来指定模块对应代码的路径，作为HAP的入口。

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

#### 应用上下文Context

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（应用开发路径）、area（文件分区）等，以及应用的一些基本方法，例如createBundleContext()、getApplicationContext()等。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。

简单理解，context就是包含了当前应用的一些基础信息和资源的一个对象，提供了操作这些资源和信息的能力。

各类context获取方式：

1、let uiAbilityContext = this.context;

2、let abilityStageContext = this.context;

3、let applicationContext = this.context.getApplicationContext();//可以在UIAbility/abilityStage/ExtensionAbility中获取

Context使用场景：

获取应用文件路径

获取和修改加密分区

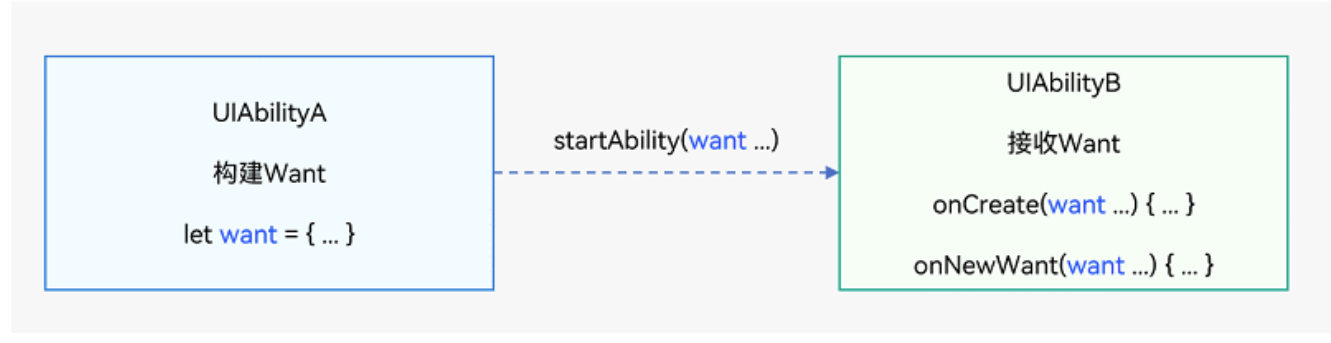
创建其他应用或其他Module的Context

订阅进程内Ability生命周期变化

#### 信息传输载体Want

概述：Want是对象间信息传递的载体，可以用于应用组件间的信息传递。其使用场景之一是作为startAbility()的参数，包含了指定的启动目标以及启动时需携带的相关数据

用法示意图：



显示Want:

在启动Ability时指定了abilityName和bundleName的Want称为显式Want。

当有明确处理请求的对象时，通过提供目标Ability所在应用的包名信息（bundleName），并在Want内指定abilityName便可启动目标Ability。显式Want通常用于在当前应用开发中启动某个已知的Ability。

隐式Want:

在启动UIAbility时未指定abilityName的Want称为隐式Want

当请求处理的对象不明确时，希望在当前应用中使用其他应用提供的某个能力（通过skills标签定义），而不关心提供该能力的具体应用，可以使用隐式Want。

隐式Want匹配：

未匹配到满足条件的Ability：启动失败。

匹配到一个满足条件的Ability：直接启动该Ability。

匹配到多个满足条件的Ability（UIAbility）：弹出选择框让用户选择。

action：表示调用方要执行的通用操作（如查看、分享、应用详情）。

常见Action:

ACTION\_HOME：启动应用入口组件的动作，需要和ENTITY\_HOME配合使用；系统桌 面应用图标就是显式的入口组件，点击也是启动入口组件；入口组件可以配置多个。

ACTION\_CHOOSE：选择本地资源数据，例如联系人、相册等；系统一般对不同类型的数据 有对应的Picker应用，例如联系人和图库。

ACTION\_VIEW\_DATA：查看数据，当使用网址uri时，则表示显示该网址对应的内容。

ACTION\_VIEW\_MULTIPLE\_DATA：发送多个数据记录的操作。

entities：表示目标Ability的类别信息（如浏览器、视频播放器），在隐式Want中是对action的补充。

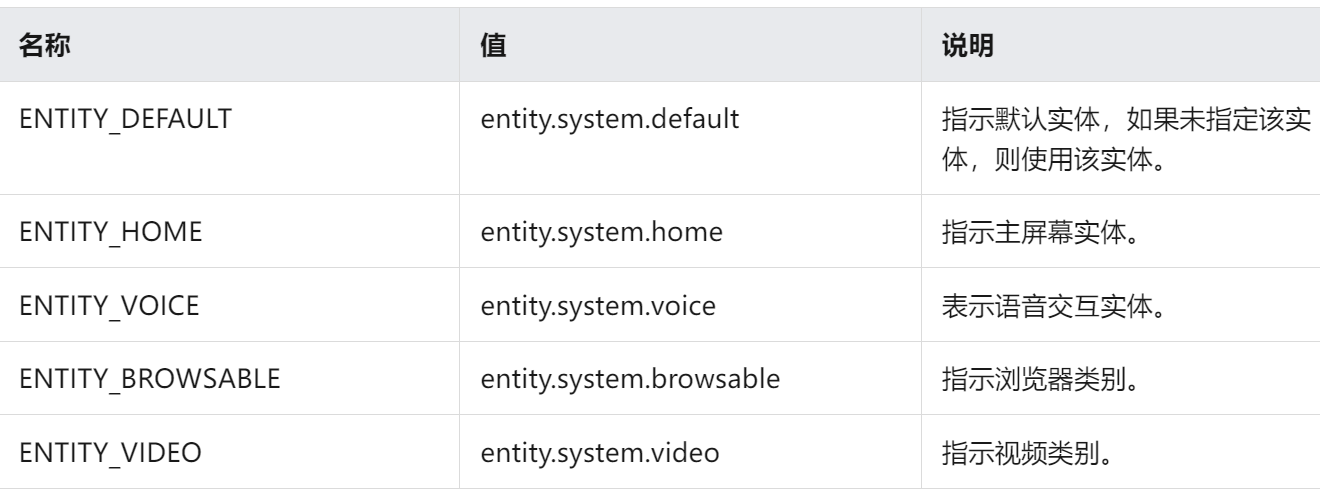
常用entities

ENTITY\_DEFAULT：默认类别无实际意义。

ENTITY\_HOME：主屏幕有图标点击入口类别。

ENTITY\_BROWSABLE：指示浏览器类别。





应用间使用Want分享数据：

数据分享需要使用两个UIAbility组件（分享方和被分享方）以及一个系统组件（应用分享框）。当分享方使用`startAbility()`方法发起数据分享时，系统会隐式匹配所有支持接收分享数据类型的应用，并将其展示给用户以供选择。用户选择应用后，系统将启动该应用来完成数据分享操作。

ohos.want.action.select：用于启动应用分享框。

ohos.want.action.sendData：用于发送单个数据记录。此Action用于将数据传递给分享方应用。

#### 进程模型

HarmonyOS的进程模型：

应用中（同一包名）的所有UIAbility运行在同一个独立进程中。

WebView拥有独立的渲染进程。

基于HarmonyOS的进程模型，系统提供了公共事件机制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。

#### 线程模型

HarmonyOS应用中每个进程都会有一个主线程，主线程有如下职责：

1.执行UI绘制；

2.管理主线程的ArkTS引擎实例，使多个UIAbility组件能够运行在其之上；

3.管理其他线程（例如Worker线程）的ArkTS引擎实例，例如启动和终止其他线程；

4.分发交互事件；

5.处理应用代码的回调，包括事件处理和生命周期管理；

6.接收Worker线程发送的消息；

除主线程外，还有一类与主线程并行的独立线程Worker，主要用于执行耗时操作，但不可以直接操作UI。Worker线程在主线程中创建，与主线程相互独立。最多可以创建8个Worker：

基于HarmonyOS的线程模型，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。

Stage模型只提供了主线程和Worker线程，Emitter主要用于主线程和Worker线程、Worker线程和Worker线程之间的事件同步。

UIAbility组件与UI均在主线程中，他们之间的数据同步请参见UIAbility组件与UI的数据同步。

# 媒体开发

## 音频开发

概述：在HarmonyOS系统中，多种API都提供了音频播放开发的支持，不同的API适用于不同音频数据格式、音频来源、音频使用场景，甚至是不同开发语言。

AVPlayer：功能较完善的音频、视频播放ArkTS/JS API,集成了流媒体和本地资源解析、媒体资源封装、音频解码和音频输出功能。可用于直接播放mp3、m4a等格式的音频文件，不支持直接播放PCM格式文件。

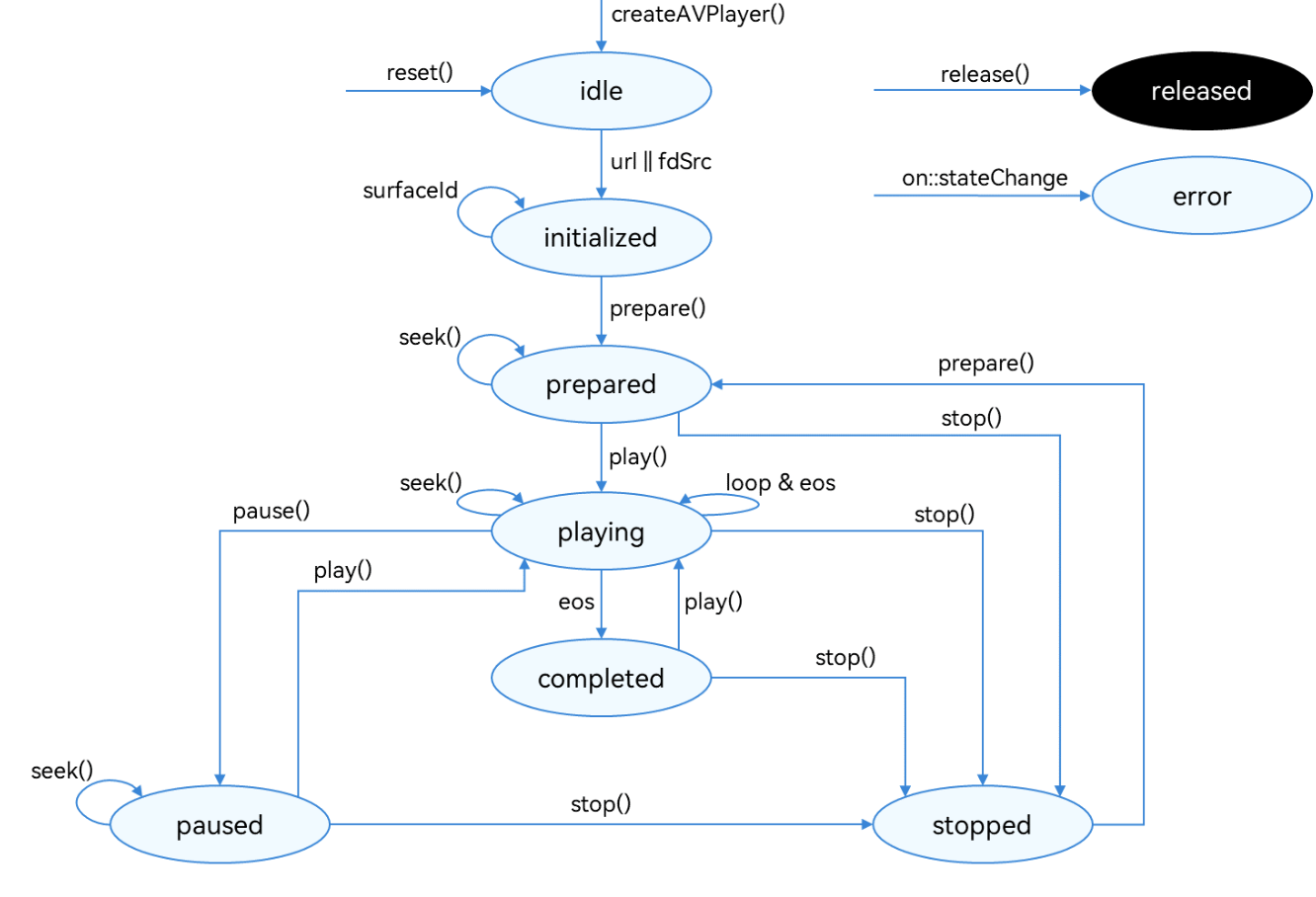
AudioRenderer：用于音频输出的ArkTS/JS API，仅支持PCM格式，需要应用持续写入音频数据进行工作。

OpenSL ES：一套跨平台标准的音频Native API，仅支持PCM格式。

使用AVPlayer开发音频播放功能：

播放的全流程包含：创建AVPlayer，设置播放源，设置播放参数（音量/倍速/焦点模式），播放控制（播放/暂停/跳转/停止），重置，销毁资源

在进行应用开发的过程中，开发者可以通过AVPlayer的state属性主动获取当前状态或使用on(stateChange’)方法监听状态变化。

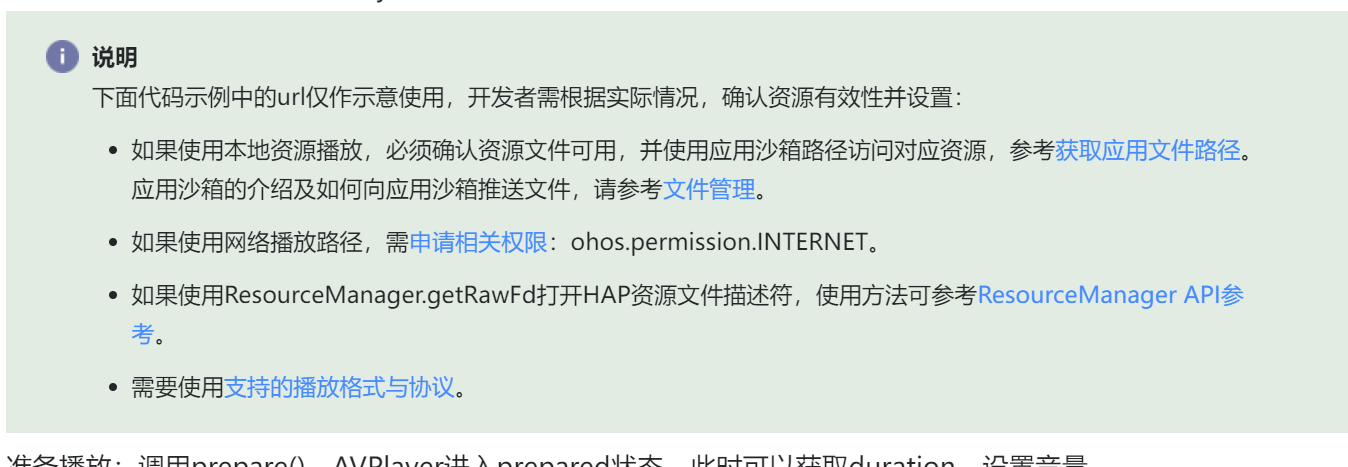


开发步骤：

1. 创建实例createAVPlayer(),AVPlayer初始化idle状态
2. 设置业务需要的监听事件，搭配全流程场景使用。支持监听事件：



1. 设置资源：设置属性url，AVPlayer进入initialized状态。



1. 准备播放：调用prepare()，AVPlayer进入prepared状态，此时可以获取duration，设置音量
2. 音频播控：播放play()，暂停pause()，跳转seek()，停止stop()等操作
3. （可选）更换资源：调用reset()重置资源，AVPlayer重新进入idle状态，允许更换url。
4. 退出播放：调用release()销毁实例，AVPlayer进入released状态，退出播放。