HarmonyOS 入门宝典

注：本文档主要内容来自：
https://developer.harmonyos.com

版本：2.0

1.HarmonyOS 概述

1.1 系统定义

HarmonyOS 是一款“面向未来”、面向全场景（移动办公、运动健康、社交通信、媒体娱乐等）的分布式操作系统。在传统的单设备系统能力的基础上，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，能够支持多种终端设备。

- 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，形成一个“超级虚拟终端”，可以实现不同的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。
- 对应用开发者而言，HarmonyOS 采用了多种分布式技术，使得应用程序的开发实现与不同终端设备的形态差异无关，降低了开发难度和成本。这能够让开发者聚焦上层业务逻辑，更加便捷、高效地开发应用。
- 对设备开发者而言，HarmonyOS 采用了组件化的设计方案，可以根据设备的资源能力和业务特征进行灵活裁剪，满足不同形态的终端设备对于操作系统的要求。

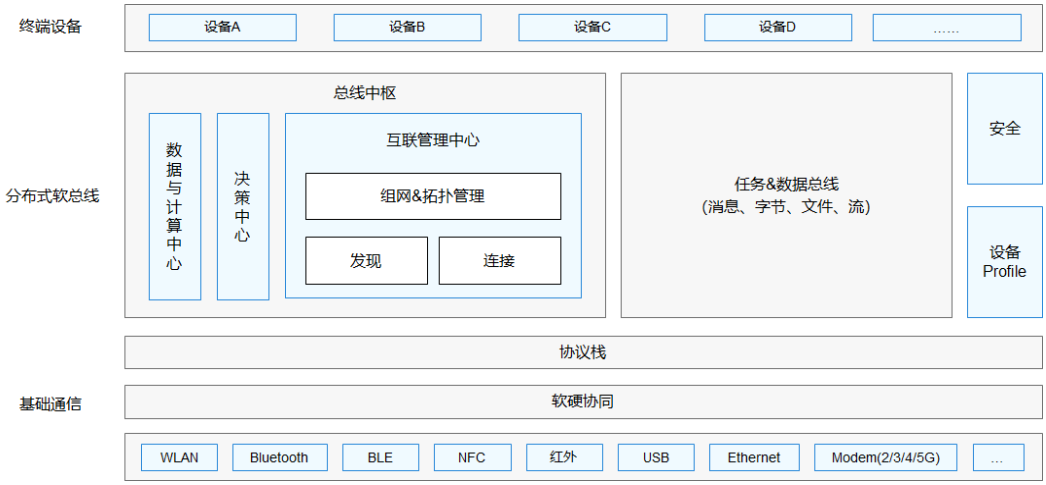
1.2 技术特性

硬件互助，资源共享

分布式软总线

分布式软总线是多种终端设备的统一基座，为设备之间的互联互通提供了统一的分布式通信能力，能够快速发现并连接设备，高效地分发任务和传输数据。分布式软总线示意图见图 1。

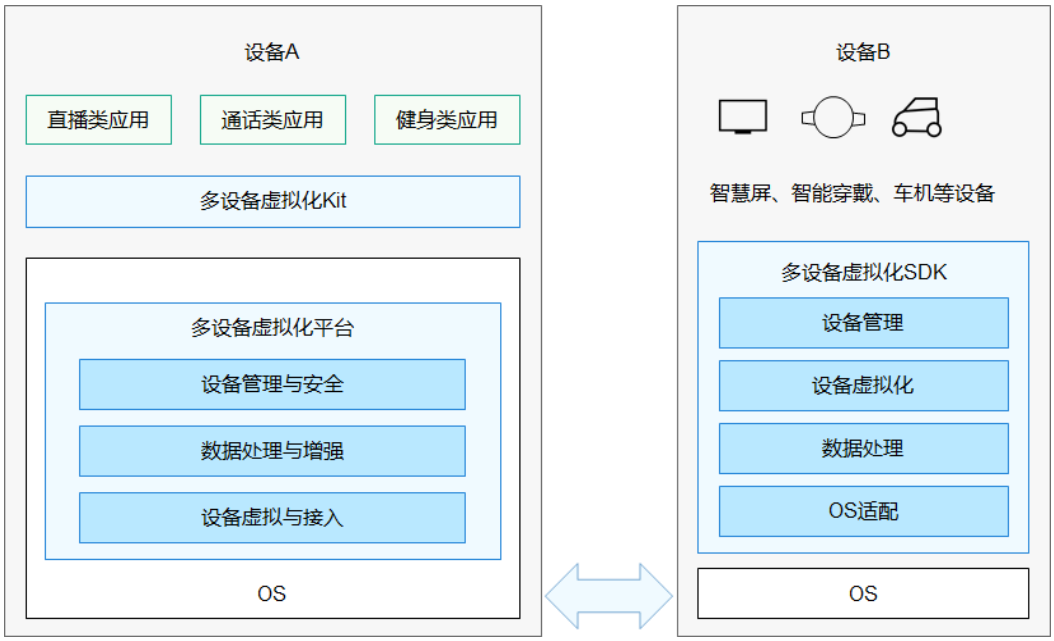
图 1　分布式软总线示意图



分布式设备虚拟化

分布式设备虚拟化平台可以实现不同设备的资源融合、设备管理、数据处理，多种设备共同形成一个超级虚拟终端。针对不同类型的任务，为用户匹配并选择能力合适的执行硬件，让业务连续地在不同设备间流转，充分发挥不同设备的资源优势。分布式设备虚拟化示意图见图 2。

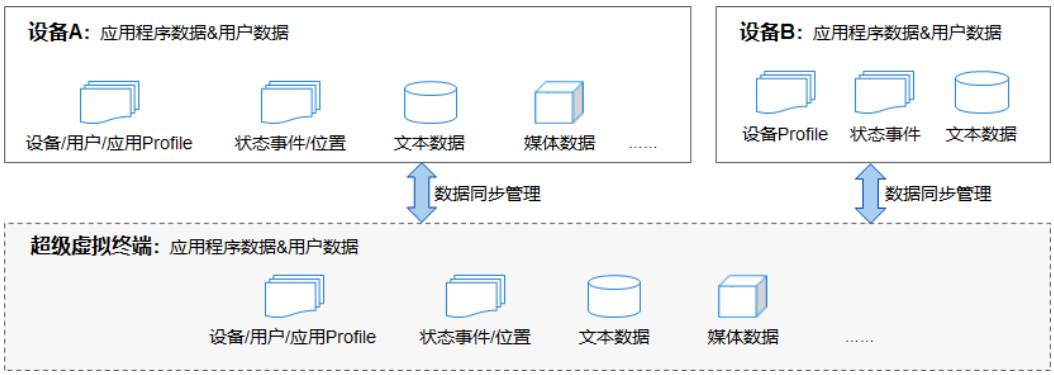
图 2　分布式设备虚拟化示意图



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，应用跨设备运行时数据无缝衔接，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见图 3。

图 3　分布式数据管理示意图



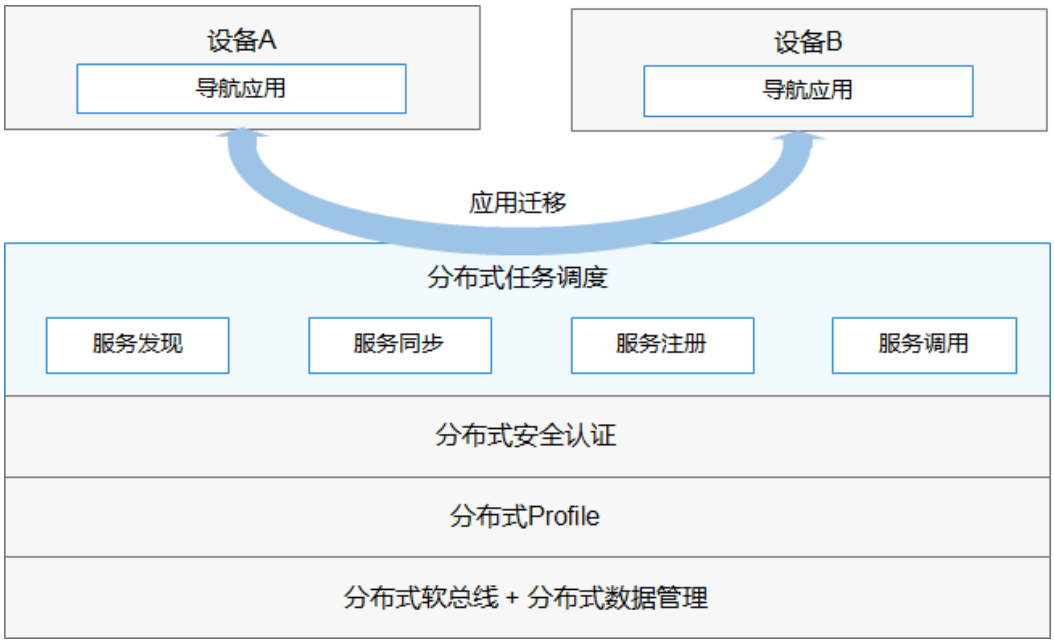
分布式任务调度

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。

图 4 以应用迁移为例，简要地展示了分布式任务调度能力。

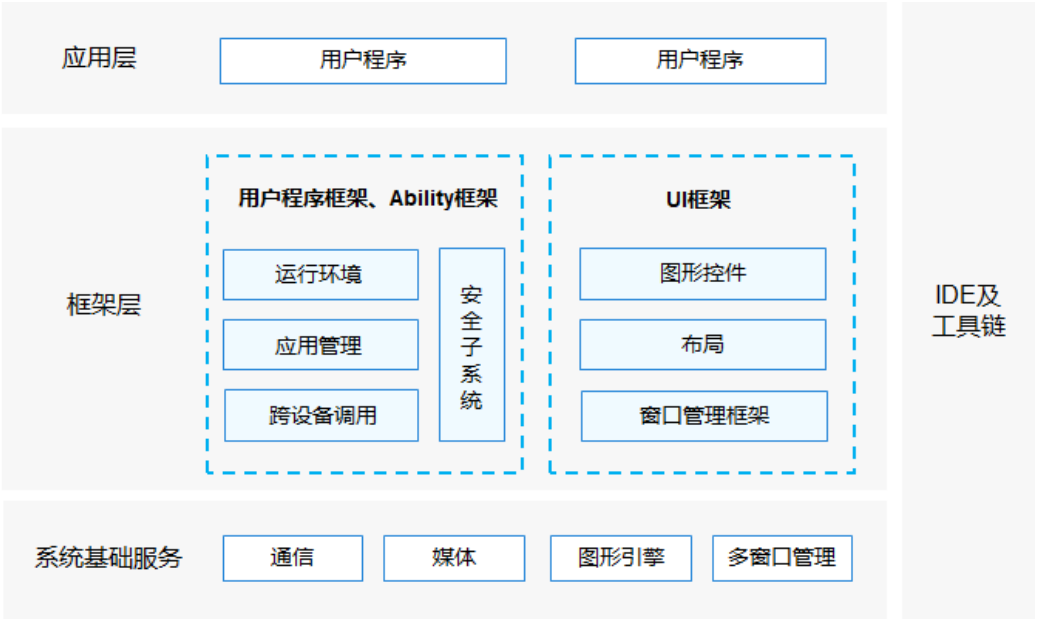
图 4　分布式任务调度示意图



一次开发，多端部署

HarmonyOS 提供了用户程序框架、Ability 框架以及 UI 框架，支持应用开发过程中多终端的业务逻辑和界面逻辑进行复用，能够实现应用的一次开发、多端部署，提升了跨设备应用的开发效率。一次开发、多端部署示意图见图 5。

图 5　一次开发、多端部署示意图



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。

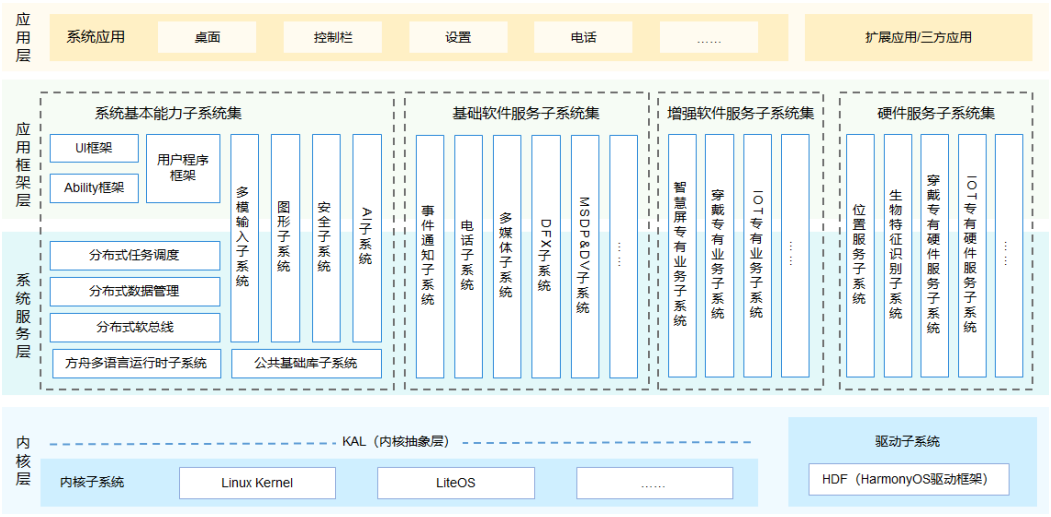
1.3 技术架构

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 技术架构如图 1 所示。

图 1　技术架构



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

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、MSDP&DV 等子系统组成。
- **增强软件服务子系统集：**为 HarmonyOS 提供针对不同设备的、差异化的能力增强型软件服务，由智慧屏专有业务、穿戴专有业务、IoT 专有业务等子系统组成。
- **硬件服务子系统集：**为 HarmonyOS 提供硬件服务，由位置服务、生物特征识别、穿戴专有硬件服务、IoT 专有硬件服务等子系统组成。

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

框架层

框架层为 HarmonyOS 的应用程序提供了 Java/C/C++/JS 等多语言的用户程序框架和 Ability 框架，以及各种软硬件服务对外开放的多语言框架 API；同时为采用 HarmonyOS 的设备提供了 C/C++/JS 等多语言的框架 API，不同设备支持的 API 与系统的组件化裁剪程度相关。

应用层

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

1.4 系统安全

在搭载 HarmonyOS 的分布式终端上，可以保证“**正确的人，通过正确的设备，正确地使用数据**”。

- 通过“分布式多端协同身份认证”来保证“正确的人”。
- 通过“在分布式终端上构筑可信运行环境”来保证“正确的设备”。
- 通过“分布式数据在跨终端流动的过程中，对数据进行分类分级管理”来保证“正确地使用数据”。

正确的人

在分布式终端场景下，“正确的人”指通过身份认证的数据访问者和业务操作者。“正确的人”是确保用户数据不被非法访问、用户隐私不泄露的前提条件。HarmonyOS 通过以下三个方面来实现协同身份认证：

- **零信任模型**：HarmonyOS 基于零信任模型，实现对用户的认证和对数据的访问控制。当用户需要跨设备访问数据资源或者发起高安全等级的业务操作（例如，对安防设备的操作）时，HarmonyOS 会对用户进行身份认证，确保其身份的可靠性。
- **多因素融合认证**：HarmonyOS 通过用户身份管理，将不同设备上标识同一用户的认证凭据关联起来，用于标识一个用户，来提高认证的准确度。
- **协同互助认证**：HarmonyOS 通过将硬件和认证能力解耦（即信息采集和认证可以在不同的设备上完成），来实现不同设备的资源池化以及能力的互助与共享，让高安全等级的设备协助低安全等级的设备完成用户身份认证。

正确的设备

在分布式终端场景下，只有保证用户使用的设备是安全可靠的，才能保证用户数据在虚拟终端上得到有效保护，避免用户隐私泄露。

- **安全启动**

确保源头每个虚拟设备运行的系统固件和应用程序是完整的、未经篡改的。通过安全启动，各个设备厂商的镜像包就不易被非法替换为恶意程序，从而保护用户的数据和隐私安全。

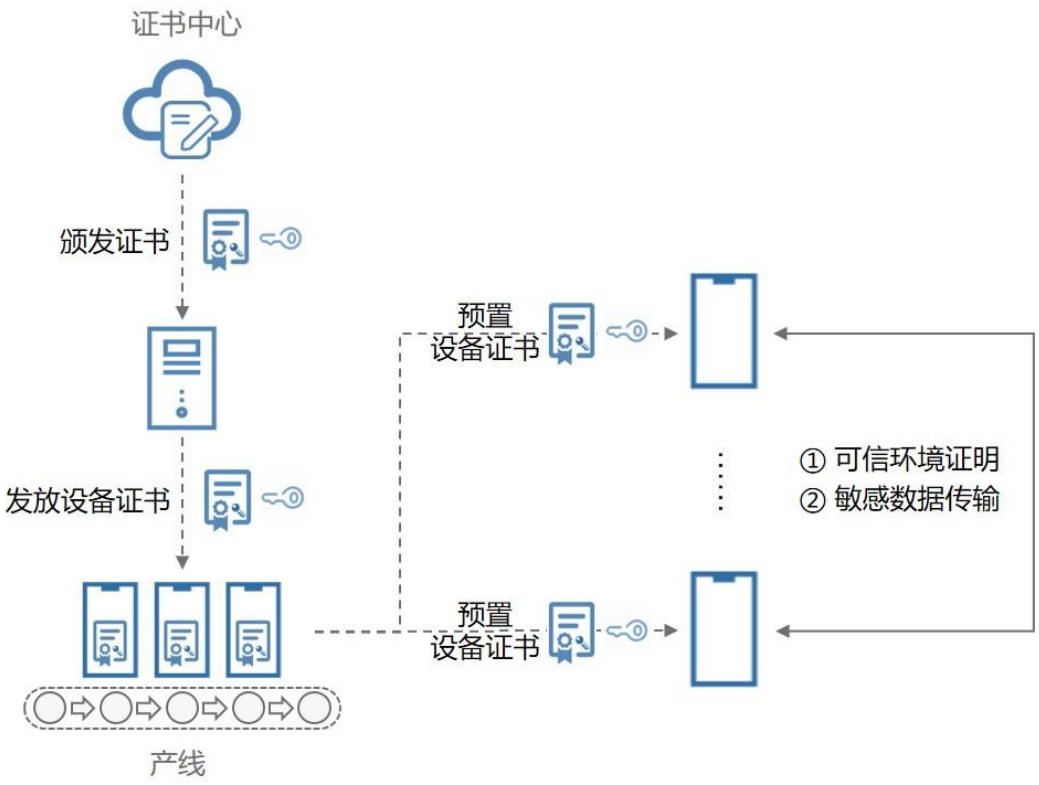
- **可信执行环境**

提供了基于硬件的可信执行环境（TEE，Trusted Execution Environment）来保护用户的个人敏感数据的存储和处理，确保数据不泄露。由于分布式终端硬件的安全能力不同，对于用户的敏感个人数据，需要使用高安全等级的设备进行存储和处理。HarmonyOS 使用基于数学可证明的形式化开发和验证的 TEE 微内核，获得了商用 OS 内核 CC EAL5+的认证评级。

- **设备证书认证**

支持为具备可信执行环境的设备预置设备证书，用于向其他虚拟终端证明自己的安全能力。对于有 TEE 环境的设备，通过预置 PKI（Public Key Infrastructure）设备证书给设备身份提供证明，确保设备是合法制造生产的。设备证书在产线进行预置，设备证书的私钥写入并安全保存在设备的 TEE 环境中，且只在 TEE 内进行使用。在必须传输用户的敏感数据（例如密钥、加密的生物特征等信息）时，会在使用设备证书进行安全环境验证后，建立从一个设备的 TEE 到另一设备的 TEE 之间的安全通道，实现安全传输。如图 1 所示。

图 1　设备证书使用示意图



正确地使用数据

在分布式终端场景下，需要确保用户能够正确地使用数据。HarmonyOS 围绕数据的生成、存储、使用、传输以及销毁过程进行全生命周期的保护，从而保证个人数据与隐私、以及系统的机密数据（如密钥）不泄漏。

- **数据生成：**根据数据所在的国家或组织的法律法规与标准规范，对数据进行分类分级，并且根据分类设置相应的保护等级。每个保护等级的数据从生成开始，在其存储、使用、传输的整个生命周期都需要根据对应的安全策略提供不同强度的安全防护。虚拟超级终端的访问控制系统支持依据标签的访问控制策略，保证数据只能在可以提供足够安全防护的虚拟终端之间存储、使用和传输。

- **数据存储：**HarmonyOS 通过区分数据的安全等级，存储到不同安全防护能力的分区，对数据进行安全保护，并提供密钥全生命周期的跨设备无缝流动和跨设备密钥访问控制能力，支撑分布式身份认证协同、分布式数据共享等业务。

- **数据使用：**HarmonyOS 通过硬件为设备提供可信执行环境。用户的个人敏感数据仅在分布式虚拟终端的可信执行环境中进行使用，确保用户数据的安全和隐私不泄露。

- **数据传输：**为了保证数据在虚拟超级终端之间安全流转，需要各设备是正确可信的，建立了信任关系（多个设备通过华为帐号建立配对关系），并能够在验证信任关系后，建立安全的连接通道，按照数据流动的规则，安全地传输数据。当设备之间进行通信时，需要基于设备的身份凭据对设备进行身份认证，并在此基础上，建立安全的加密传输通道。

- **数据销毁：**销毁密钥即销毁数据。数据在虚拟终端的存储，都建立在密钥的基础上。当销毁数据时，只需要销毁对应的密钥即完成了数据的销毁。

2.开发基础知识

2.1 应用基础知识

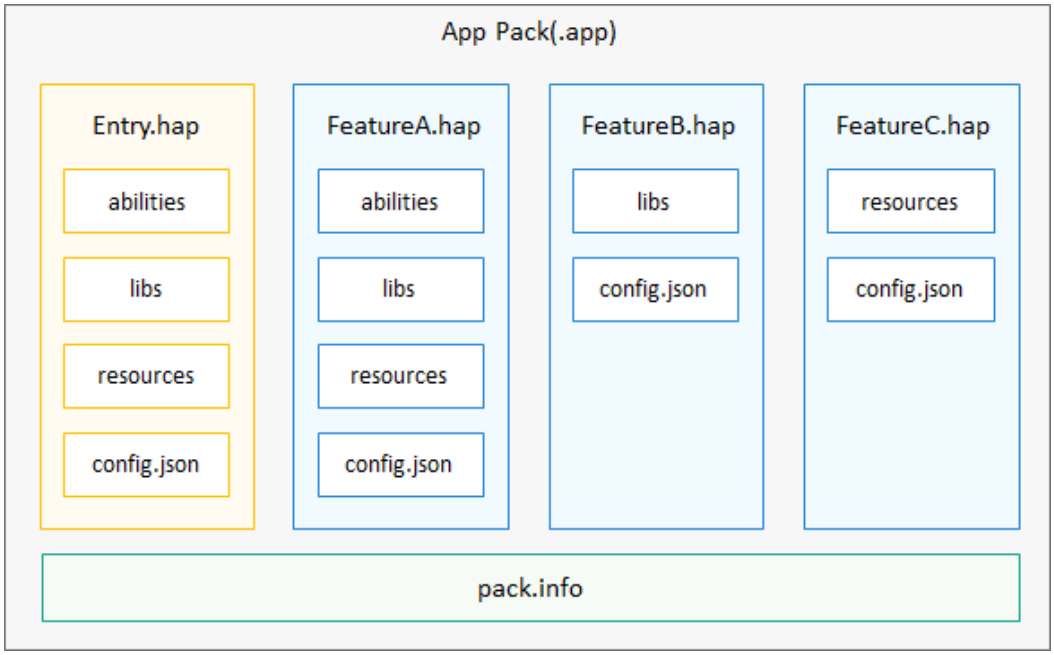
APP

HarmonyOS 的应用软件包以 **APP Pack**（Application Package）形式发布，它是由一个或多个 **HAP**（HarmonyOS Ability Package）以及描述每个 HAP 属性的 pack.info 组成。HAP 是 Ability 的部署包，HarmonyOS 应用代码围绕 Ability 组件展开。

一个 HAP 是由代码、资源、第三方库及应用配置文件组成的模块包，可分为 entry 和 feature 两种模块类型，如图 1 所示。

- **entry**：应用的主模块。一个 APP 中，对于同一设备类型必须有且只有一个 entry 类型的 HAP，可独立安装运行。
- **feature**：应用的动态特性模块。一个 APP 可以包含一个或多个 feature 类型的 HAP，也可以不含。只有包含 Ability 的 HAP 才能够独立运行。

图 1　APP 逻辑视图



Ability

Ability 是应用所具备的能力的抽象，一个应用可以包含一个或多个 Ability。Ability 分为两种类型：FA（Feature Ability）和 PA（Particle Ability）。FA/PA 是应用的基本组成单元，能够实现特定的业务功能。FA 有 UI 界面，而 PA 无 UI 界面。

库文件

库文件是应用依赖的第三方代码形式，存放在 libs 目录，是.so 文件。

资源文件

应用的资源文件（字符串、图片、音频等）存放于 resources 目录下，便于开发者使用和维护，详见资源文件分类。

配置文件

配置文件 (config.json) 是应用的 Ability 信息，用于声明应用的 Ability，以及应用所需权限等信息，详见应用配置文件。

pack. info

描述应用软件包中每个 HAP 的属性，由 IDE 编译生成，应用市场根据该文件进行拆包和 HAP 的分类存储。HAP 的具体属性包括：

- delivery-with-install：表示该 HAP 是否支持随应用安装。
- name：HAP 文件名。
- module-type：模块类型，entry 或 feature。
- device-type：表示支持该 HAP 运行的设备类型。

2.2 应用配置文件

2.2.1 简介

应用的每个 HAP 的根目录下都存在一个“config.json”配置文件，主要涵盖以下三个方面：

- 应用的全局配置信息，包含应用的包名、生产厂商、版本号等基本信息。
- 应用在具体设备上的配置信息。
- HAP 包的配置信息，包含每个 Ability 必须定义的基本属性（如包名、类名、类型以及 Ability 提供的能力），以及应用访问系统或其他应用受保护部分所需的权限等。

文件约定

配置文件“config.json”采用 JSON 文件格式，由属性和值两部分构成：

- **属性**

属性出现顺序不分先后，且每个属性最多只允许出现一次。

- **值**

每个属性的值为 JSON 的基本数据类型（数值、字符串、布尔值、数组、对象或者 null 类型）。如果属性值需要引用资源文件，可参见资源文件。

2.2.2 配置文件的元素

此部分提供“config.json”文件中所有属性的详细解释。

配置文件的内部结构

应用的配置文件“config.json”中由“app”、“deviceConfig”和“module”三个部分组成，缺一不可。配置文件的内部结构说明参见表 1。

表 1 配置文件的内部结构说明

属性名称	含义	数据类型	是否可缺省
app	表示应用的全局配置信息。同一个应用的不同 HAP 包的“app”配置必须保持一致。	对象	否
deviceConfig	表示应用在具体设备上的配置信息。	对象	否
module	表示 HAP 包的配置信息。该标签下的配置只对当前 HAP 包生效。	对象	否

app 对象的内部结构

app 对象包含应用的全局配置信息，内部结构说明参见表 2。

表 2 app 对象的内部结构说明

属性名称	子属性名称	含义	数据类型	是否可缺省
bundleName	-	表示应用的包名，用于标识应用的唯一性。 采用反域名形式的字符串表示（例如，com.huawei.himusic）。建议第一级为域名后缀“com”，第二级为厂商/个人名，第三级为应用名，也可以采用多级。 支持的字符串长度为 7~127 字节。	字符串	否
vendor	-	表示对应用开发厂商的描述。 字符串长度不超过 255 字节。	字符串	可缺省，缺省值为空。
version	-	表示应用的版本信息。	对象	否
	code	表示应用的版本号，仅用于 HarmonyOS 管理该应用，对用户不可见。取值为大于零的整数。	数值	否
	name	表示应用的版本号，用于向用户呈现。取值可以自定义。	字符串	否
apiVersion	-	表示应用依赖的 HarmonyOS 的 API 版本。	对象	否

表 2 app 对象的内部结构说明

属性名称	子属性名称	含义	数据类型	是否可缺省
	compatible	表示应用运行需要的 API 最小版本。取值为大于零的整数。	数值	否
	target	表示应用运行需要的 API 目标版本。取值为大于零的整数。	数值	可缺省，缺省值为应用所在设备的当前 API 版本。

app 示例：

```
1."app": {
2.     "bundleName": "com.huawei.hiworld.example",
3.     "vendor": "huawei",
4.     "version": {
5.         "code": 2,
6.         "name": "2.0"
7.     }
8.     "apiVersion": {
9.         "compatible": 3,
```

```
10.         "target": 3
11.     }
12.}
```

deviceConfig 对象的内部结构

deviceConfig 包含在具体设备上的应用配置信息，可以包含 default、car、tv、wearable、liteWearable、smartVision 等属性。default 标签内的配置是适用于所有设备通用，其他设备类型如果有特殊的需求，则需要在该设备类型的标签下进行配置。内部结构说明参见表 3。

表 3 deviceConfig 对象的内部结构说明

属性名称	含义	数据类型	是否可缺省
default	表示所有设备通用的应用配置信息。	对象	否
car	表示车机特有的应用配置信息。	对象	可缺省，缺省为空。
tv	表示智慧屏特有的应用配置信息。	对象	可缺省，缺省为空。
wearable	表示智能穿戴特有的应用配置信息。	对象	可缺省，缺省为空。
liteWearable	表示轻量级智能穿戴特有的应用配置信息。	对象	可缺省，缺省为空。
smartVision	表示智能摄像头特有的应用配置信息。	对象	可缺省，缺省为空。

default、car、tv、wearable、liteWearable、smartVision 等对象的内部结构说明，可参见表 4。

表 4 default/car/tv/wearable 等对象的内部结构说明

属性名称	含义	数据类型	是否可缺省
process	表示应用或者 Ability 的进程名。 如果在“deviceConfig”标签下配置了“process”标签，则该应用的所有 Ability 都运行在这个进程中。 如果在“abilities”标签下也为某个 Ability 配置了“process”标签，则该 Ability 就运行在这个进程中。 该标签仅适用于智慧屏、智能穿戴、车机。	字符串	可缺省，缺省为应用的软件包名。
directLaunch	表示应用是否支持在设备未解锁状态直接启动。如果配置为“true”，则表示应用支持在设备未解锁状态下启动。使用场景举例：应用支持在设备未解锁情况下接听来电。 该标签仅适用于智慧屏、智能穿戴、车机。	布尔类型	可缺省，缺省为 false。
supportBackup	表示应用是否支持备份和恢复。 如果配置为“false”，则不支持为该应用执行备份或恢复操作。 该标签仅适用于智慧屏、智能穿戴、车机。	布尔类型	可缺省，缺省为 false。

表 4 default/car/tv/wearable 等对象的内部结构说明

属性名称	含义	数据类型	是否可缺省
compressNativeLibs	表示 libs 库是否以压缩存储的方式打包到 HAP 包。如果配置为“false”，则 libs 库以不压缩的方式存储，HAP 包在安装时无需解压 libs，运行时会直接从 HAP 内加载 libs 库。 该标签仅适用于智慧屏、智能穿戴、车机。	布尔类型	可缺省，缺省为 true。
network	表示网络安全性配置。该标签允许应用通过配置文件的安全声明来自定义其网络安全，无需修改应用代码。	对象	可缺省，缺省为空。

表 5 network 对象的内部结构说明

属性名称	含义	数据类型	是否可缺省
usesCleartext	表示是否允许应用使用明文网络流量（例如，明文 HTTP）。默认值为“false”。 true：允许应用使用明文流量的请求。 false：拒绝应用使用明文流量的请求。	布尔类型	可缺省，缺省为空。
securityConfig	表示应用的网络安全配置信息。	对象	可缺省，缺省为空。

表 6 securityConfig 对象的内部结构说明

属性名称	子属性名称	含义	数据类型	是否可缺省
domainSettings	-	表示自定义的网域范围的安全配置，支持多层嵌套，即一个 domainSettings 对象中允许嵌套更小网域范围的 domainSettings 对象。	对象	可缺省，缺省为空。
	cleartextPermitted	表示自定义的网域范围内是否允许明文流量传输。当 useCleartext 和 securityConfig 同时存在时，自定义网域是否允许明文流量传输以 cleartextPermitted 的取值为准。 true：允许明文流量传输。 false：拒绝明文流量传输。	布尔类型	否
	domains	表示域名配置信息，包含两个参数：subDomains 和 name。 subDomains（布尔类型）：表示是否包含子域名。如果为 “true”，此网域规则将与相应网域及所有子网域（包括子网域的子网	对象数组	否

表 6 securityConfig 对象的内部结构说明

属性名称	子属性名称	含义	数据类型	是否可缺省
		域）匹配。否则，该规则仅适用于精确匹配项。 name（字符串）：表示域名名称。		

deviceConfig 示例：

```
1. "deviceConfig": {
2.     "default": {
3.         "process": "com.huawei.hiworld.example",
4.         "directLaunch": false,
5.         "supportBackup": false,
6.         "network": {
7.             "usesCleartext": true,
8.             "securityConfig": {
9.                 "domainSettings": {
10.                     "cleartextPermitted": true,
11.                     "domains": [
12.                         {
13.                             "subDomains": true,
14.                             "name": "example.ohos.com"
15.                         }
16.                     ]
```

```
17.                     }
18.                 }
19.             }
20.         }
21.}
```

module 对象的内部结构

module 对象包含 HAP 包的配置信息，内部结构说明参见表 7。

表 7 module 对象的内部结构说明

属性名称	含义	数据类型	是否可缺省
package	表示 HAP 的包结构名称，在应用内应保证唯一性。采用反向域名格式（建议与 HAP 的工程目录保持一致）。字符串长度不超过 127 字节。 该标签仅适用于智慧屏、智能穿戴、车机。	字符串	否
name	表示 HAP 的类名。采用反向域名方式表示，前缀需要与同级的 package 标签指	字符串	否

表 7 module 对象的内部结构说明

属性名称	含义	数据类型	是否可缺省
	定的包名一致，也可采用“.”开头的命名方式。字符串长度不超过 255 字节。 该标签仅适用于智慧屏、智能穿戴、车机。		
description	表示 HAP 的描述信息。字符串长度不超过 255 字节。如果字符串超出长度或者需要支持多语言，可以采用资源索引的方式添加描述内容。 该标签仅适用于智慧屏、智能穿戴、车机。	字符串	可缺省，缺省值为空。
supportedModes	表示应用支持的运行模式。当前只定义了驾驶模式（drive）。 该标签仅适用于车机。	字符串数组	可缺省，缺省值为空。
deviceType	表示允许 Ability 运行的设备类型。系统预定义的设备类型包	字符串数组	否

表 7 module 对象的内部结构说明

属性名称	含义	数据类型	是否可缺省
	括：tv（智慧屏）、car（车机）、wearable（智能穿戴）、liteWearable（轻量级智能穿戴）等。		
distro	表示 HAP 发布的具体描述。 该标签仅适用于智慧屏、智能穿戴、车机。	对象	否
abilities	表示当前模块内的所有 Ability。采用对象数组格式，其中每个元素表示一个 Ability 对象。	对象数组	可缺省，缺省值为空。
js	表示基于 JS UI 框架开发的 JS 模块集合，其中的每个元素代表一个 JS 模块的信息。	对象	可缺省，缺省值为空。
shortcuts	表示应用的快捷方式信息。采用对象数组格式，其中的每个元素表示一个快捷方式对象。	对象数组	可缺省，缺省值为空。

表 7 module 对象的内部结构说明

属性名称	含义	数据类型	是否可缺省
defPermissions	表示应用定义的权限。应用调用者必须申请这些权限，才能正常调用该应用。	对象数组	可缺省，缺省值为空。
reqPermissions	表示应用运行时向系统申请的权限。	对象数组	可缺省，缺省值为空。

module 示例：

```
1."module": {
2.     "package": "com.example.myapplication.entry",
3.     "name": ".MyOHOSAbilityPackage",
4.     "description": "$string:description_application",
5.     "supportedModes": [
6.         "drive"
7.     ],
8.     "deviceType": [
9.         "car"
10.     ],
11.     "distro": {
12.         "deliveryWithInstall": true,
13.         "moduleName": "ohos_entry",
```

```
        "moduleType": "entry"
    },
    "abilities": [
        ...
    ],
    "shortcuts": [
        ...
    ],
    "js": [
        ...
    ],
    "reqPermissions": [
        ...
    ],
    "defPermissions": [
        ...
    ]
}
```

表 8 distro 对象的内部结构说明

属性名称	含义	数据类型	是否可缺省
deliveryWithInstall	表示当前 HAP 是否支持随应用安装。 true：支持随应用安装。 false：不支持随应用安装。	布尔类型	否
moduleName	表示当前 HAP 的名称。	字符串	否
moduleType	表示当前 HAP 的类型，包括两种类型：entry 和 feature。	字符串	否

distro 示例：

```
1."distro": {
2.    "deliveryWithInstall": true,
3.    "moduleName": "ohos_entry",
4.    "moduleType": "entry"
5.}
```

表 9 abilities 对象的内部结构说明

属性名称	含义	数据类型	是否可缺省
name	表示 Ability 名称。取值可采用反向域名方式表示，由包名和类名组成，如“com.example.myapplication.MainAbility”；也可采用“.”开头的类名方式表示，如“.MainAbility”。 该标签仅适用于智慧屏、智能穿戴、车	字符串	否

表 9 abilities 对象的内部结构说明

属性名称	含义	数据类型	是否可缺省
	机。		
description	表示对 Ability 的描述。取值可以是描述性内容，也可以是对描述性内容的资源索引，以支持多语言。	字符串	可缺省，缺省值为空。
icon	表示 Ability 图标资源文件的索引。取值示例：$media:ability_icon。 如果在该 Ability 的“skills”属性中，“actions”的取值包 含 “action.system.home”，“entities”取值中包含“entity.system.home”，则该 Ability 的 icon 将同时作为应用的 icon。 如果存在多个符合条件的 Ability，则取位置靠前的 Ability 的 icon 作为应用的 icon。	字符串	可缺省，缺省值为空。
label	表示 Ability 对用户显示的名称。取值可以是 Ability 名称，也可以是对该名称的资源索引，以支持多语言。 如果在该 Ability 的“skills”属性中，“actions”的取值包 含 “action.system.home”，“entities”取值中包含“entity.system.home”，则该 Ability 的 label 将同时作为应用的 label。 如果存在多个符合条件的 Ability，则取位置靠前的 Ability 的 label 作为应用的 label。	字符串	可缺省，缺省值为空。
uri	表示 Ability 的统一资源标识符。格式为[scheme:][//authority][path][?query][#fr	字符	可缺省，对于 data 类

表 9 abilities 对象的内部结构说明

属性名称	含义	数据类型	是否可缺省
	agment]。	串	型的 Ability 不可缺省。
launchType	表示 Ability 的启动模式，支持“standard”和“singleton”两种模式： standard：表示该 Ability 可以有多实例。“standard”模式适用于大多数应用场景。 singleton：表示该 Ability 只可以有一个实例。例如，具有全局唯一性的呼叫来电界面即采用“singleton”模式。 该标签仅适用于智慧屏、智能穿戴、车机。	字符串	可缺省，缺省值为 standard。
visible	表示 Ability 是否可以被其他应用调用。 true：可以被其他应用调用。 false：不能被其他应用调用。	布尔类型	可缺省，缺省值为 false。
permissions	表示其他应用的 Ability 调用此 Ability 时需要申请的权限。通常采用反向域名格式，取值可以是系统预定义的权限，也可以是开发者自定义的权限。如果是自定义权限，取值必须与“defPermissions”标签中定义的某个权限的“name”标签值一致。	字符串数组	可缺省，缺省值为空。
skills	表示 Ability 能够接收的 Intent 的特征。	对象数	可缺省，缺省值为空。

表 9 abilities 对象的内部结构说明

属性名称	含义	数据类型	是否可缺省
		组	
deviceCapability	表示 Ability 运行时要求设备具有的能力，采用字符串数组的格式表示。	字符串数组	可缺省，缺省值为空。
type	表示 Ability 的类型。取值范围如下： page：表示基于 Page 模板开发的 FA，用于提供与用户交互的能力。 service：表示基于 Service 模板开发的 PA，用于提供后台运行任务的能力。 data：表示基于 Data 模板开发的 PA，用于对外部提供统一的数据访问抽象。	字符串	否
formEnabled	表示 FA 类型的 Ability 是否提供卡片（form）能力。该标签仅适用于 page 类型的 Ability。 true：提供卡片能力。 false：不提供卡片能力。	布尔类型	可缺省，缺省值为 false。
form	表示 AbilityForm 的属性。该标签仅当“formEnabled”为“true”时，才能生效。	对象	可缺省，缺省值为空。
orientation	表示该 Ability 的显示模式。该标签仅适用于 page 类型的 Ability。取值范围如下： unspecified：由系统自动判断显示方	字符串	可缺省，缺省值为 unspecified。

表 9 abilities 对象的内部结构说明

属性名称	含义	数据类型	是否可缺省
	向。 landscape：横屏模式。 portrait：竖屏模式。 followRecent：跟随栈中最近的应用。		
backgroundModes	表示后台服务的类型，可以为一个服务配置多个后台服务类型。该标签仅适用于 service 类型的 Ability。取值范围如下： dataTransfer：通过网络/对端设备进行数据下载、备份、分享、传输等业务。 audioPlayback：音频输出业务。 audioRecording：音频输入业务。 pictureInPicture：画中画、小窗口播放视频业务。 voip：音视频电话、VOIP 业务。 location：定位、导航业务。 bluetoothInteraction：蓝牙扫描、连接、传输业务。 wifiInteraction：WLAN 扫描、连接、传输业务。 screenFetch：录屏、截屏业务。	字符串数组	可缺省，缺省值为空。
readPermission	表示读取 Ability 的数据所需的权限。该标签仅适用于 data 类型的 Ability。取值为长度不超过 255 字节的字符串。	字符串	可缺省，缺省为空。

表 9 abilities 对象的内部结构说明

属性名称	含义	数据类型	是否可缺省
	该标签仅适用于智慧屏、智能穿戴、车机。		
writePermission	表示向 Ability 写数据所需的权限。该标签仅适用于 data 类型的 Ability。取值为长度不超过 255 字节的字符串。 该标签仅适用于智慧屏、智能穿戴、车机。	字符串	可缺省，缺省为空。
directLaunch	表示 Ability 是否支持在设备未解锁状态直接启动。如果配置为“true”，则表示 Ability 支持在设备未解锁状态下启动。 如果“deviceConfig”和“abilities”中同时配置了“directLaunch”，则采用 Ability 对应的取值；如果同时未配置，则采用系统默认值。	布尔值	可缺省，缺省为 false。
configChanges	表示 Ability 关注的系统配置集合。当已关注的配置发生变更后，Ability 会收到 onConfigurationUpdated 回调。取值范围： locale：表示语言区域发生变更。 layout：表示屏幕布局发生变更。 fontSize：表示字号发生变更。 orientation：表示屏幕方向发生变更。 density：表示显示密度发生变更。	字符串数组	可缺省，缺省为空。
mission	表示 Ability 指定的任务栈。该标签仅适	字	可缺省，缺

表 9 abilities 对象的内部结构说明

属性名称	含义	数据类型	是否可缺省
	用于 page 类型的 Ability。默认情况下应用中所有 Ability 同属一个任务栈。 该标签仅适用于智慧屏、智能穿戴、车机。	符串	省为应用的包名。
targetAbility	表示当前 Ability 重用的目标 Ability。该标签仅适用于 page 类型的 Ability。如果配置了 targetAbility 属性，则当前 Ability（即别名 Ability）的属性中仅“name”、“icon”、“label”、“visible”、“permissions”、“skills”生效，其它属性均沿用 targetAbility 中的属性值。目标 Ability 必须与别名 Ability 在同一应用中，且在配置文件中目标 Ability 必须在别名之前进行声明。 该标签仅适用于智慧屏、智能穿戴、车机。	字符串	可缺省，缺省值为空。表示当前 Ability 不是一个别名 Ability。
multiUserShared	表示 Ability 是否支持多用户状态进行共享，该标签仅适用于 data 类型的 Ability。 配置为“true”时，表示在多用户下只有一份存储数据。需要注意的是，该属性会使 visible 属性失效。 该标签仅适用于智慧屏、智能穿戴、车机。	布尔类型	可缺省，缺省值为 false。
supportPipMode	表示 Ability 是否支持用户进入 PIP 模式（用于在在页面最上层悬浮小窗口，俗称“画中画”，常见于视频播放等场景）。	布尔类	可缺省，缺省值为 false。

表 9 abilities 对象的内部结构说明

属性名称	含义	数据类型	是否可缺省
	该标签仅适用于 page 类型的 Ability。该标签仅适用于智慧屏、智能穿戴、车机。	型	

abilities 示例：

```
1."abilities": [
2.    {
3.        "name": ".MainAbility",
4.        "description": "$string:description_main_ability",
5.        "icon": "$media:hiworld.png",
6.        "label": "HiMusic",
7.        "type": "page",
8.        "formEnabled": false,
9.        "launchType": "standard",
10.        "orientation": "unspecified",
11.        "permissions": [
12.        ],
13.        "visible": false,
14.        "skills": [
15.            {
16.                "actions": [
17.                    "action.system.home"
18.                ],
```

```
                "entities": [
                    "entity.system.home"
                ]
            }
        ],
        "configChanges": [
            "locale",
            "layout",
            "fontSize",
            "orientation"
        ],
        "directLaunch": false,
        "process": "string",
        "backgroundModes": [
            "dataTransfer",
            "audioPlayback",
            "audioRecording",
            "pictureInPicture",
            "voip",
            "location",
            "bluetoothInteraction",
            "wifiInteraction",
            "screenFetch"
        ],
    }
]
```

表 10 skills 对象的内部结构说明

属性名称	子属性名称	含义	数据类型	是否可缺省
actions	-	表示能够接收的 Intent 的 action 值，可以包含一个或多个 action。 取值通常为系统预定义的 action 值，详见《API 参考》中的 ohos.aafwk.content.Intent 类。	字符串数组	可缺省，缺省值为空。
entities	-	表示能够接收的 Intent 的 Ability 的类别（如视频、桌面应用等），可以包含一个或多个 entity。 取值通常为系统预定义的类别，详见《API 参考》中的 ohos.aafwk.content.Intent 类，也可以自定义。	字符串数组	可缺省，缺省值为空。
uris	-	表示能够接收的 Intent 的 uri，可以包含一个或者多个 uri。	对象数组	可缺省，缺省值为空。
	scheme	表示 uri 的 scheme 值。	字符串	不可缺省。
	host	表示 uri 的 host 值。	字符串	可缺省，缺省值为空。
	port	表示 uri 的 port 值。	字符	可缺省，缺

表 10 skills 对象的内部结构说明

属性名称	子属性名称	含义	数据类型	是否可缺省
			串	省值为空。
	path	表示 uri 的 path 值。	字符串	可缺省，缺省值为空。
	type	表示 uri 的 type 值。	字符串	可缺省，缺省值为空。

skills 示例：

```
1."skills": [
2.    {
3.        "actions": [
4.            "action.system.home"
5.        ],
6.        "entities": [
7.            "entity.system.home"
8.        ],
9.        "uris": [
10.            {
11.                "scheme": "http",
```

```
12.                 "host": "www.xxx.com",
13.                 "port": "8080",
14.                 "path": "query/student/name",
15.                 "type": "text"
16.             }
17.         ]
18.     }
19.]
```

表 11 form 对象的内部结构说明

属性名称	含义	数据类型	是否可缺省
formEntity	表示 AbilityForm 支持的显示方式，当前支持的位置包括： homeScreen：以桌面图标显示。 searchbox：在全局搜索显示。	字符串数组	可缺省，缺省值为空。
minHeight	表示 AbilityForm 缩放时能达到的最小高度，单位：像素。	数值	可缺省，缺省值为 0。
defaultHeight	表示 AbilityForm 的默认高度，单位：像素。Form 使用方应当根据该值为 Form 申请相应高度的容器布局。	数值	可缺省，缺省值为 0。
minWidth	表示 AbilityForm 缩放时能达到的最小宽度，单位：像素。	数值	可缺省，缺省值为 0。
defaultWidth	表示 AbilityForm 的默认宽度，单位：	数	可缺省，

表 11 form 对象的内部结构说明

属性名称	含义	数据类型	是否可缺省
	像素。Form 使用方应当根据该值为 Form 申请相应宽度的容器布局。	值	缺省值为 0。

form 示例：

```
1."form": {
2.    "formEntity": [
3.        "homeScreen",
4.        "searchbox"
5.    ],
6.    "minHeight": 100,
7.    "maxHeight": 200,
8.    "minWidth": 100,
9.    "maxWidth": 200
10.}
```

表 12 js 对象的内部结构说明

属性名称	子属性名称	含义	数据类型	是否可缺省
name	-	表示 JS Module 的名字。该标签不可缺省，默认值为 default。	字符串	否

表 12 js 对象的内部结构说明

属性名称	子属性名称	含义	数据类型	是否可缺省
pages	-	表示 JS Module 的页面用于列举 JS Module 中每个页面的路由信息[页面路径+页面名称]。该标签不可缺省，取值为数组，数组第一个元素代表 JS FA 首页。	数组	否
window	-	用于定义与显示窗口相关的配置。 该标签仅适用于智慧屏、智能穿戴、车机。	对象	可缺省。
	designWidth	表示页面设计基准宽度。以此为基准，根据实际设备宽度来缩放元素大小。	数值	可缺省，缺省值为 750px。
	autoDesignWidth	表示页面设计基准宽度是否自动计算。当配置为 true 时，designWidth 将会被忽略，设计基准宽度由设备宽度与屏幕密度计算得出。	布尔类型	可缺省，缺省值为 false。

js 示例：

```
"js": [
    {
        "name": "default",
        "pages": [
            "pages/index/index",
            "pages/detail/detail"
        ],
        "window": {
            "designWidth": 750,
             "autoDesignWidth": false
         }
    }
]
```

表 13 shortcuts 对象的内部结构说明

属性名称	子属性名称	含义	数据类型	是否可缺省
shortcutId	-	表示快捷方式的 ID。字符串的最大长度为 63 字节。	字符串	否
label	-	表示快捷方式的标签信息，即快捷方式对外显示的文字描	字符	可缺省，缺

表 13 shortcuts 对象的内部结构说明

属性名称	子属性名称	含义	数据类型	是否可缺省
		述信息。取值可以是描述性内容，也可以是标识 label 的资源索引。字符串最大长度为 63 字节。	串	省为空。
intents	-	表示快捷方式内定义的目标 intent 信息集合，每个 intent 可配置两个子标签，targetClass, targetBundle。	-	可缺省，缺省为空。
	targetClass	表示快捷方式目标类名。	字符串	可缺省，缺省值为空。
	targetBundle	表示快捷方式目标 Ability 所在应用的包名。	字符串	可缺省，缺省值为空。

示例：

```
"shortcuts": [
    {
        "shortcutId": "id",
        "label": "$string:shortcut",
        "intents": [
            {
                "targetBundle": "com.huawei.hiworld.himusic",
                "targetClass":
"com.huawei.hiworld.himusic.entry.MainAbility"
            }
        ]
    }
]
```

2.2.3 配置文件示例

以 JSON 文件为 config.json 的一个简单示例，该示例的应用声明为三个 Ability。

```
{
    "app": {
        "bundleName": "com.huawei.hiworld.himusic",
        "vendor": "huawei",
        "version": {
            "code": 2,
            "name": "2.0"
        }
        "apiVersion": {
             "compatible": 3,
             "target": 3
         }
    },
    "deviceConfig": {
        "default": {
        }
    },
    "module": {
        "package": "com.huawei.hiworld.himusic.entry",
        "name": ".MainApplication",
        "supportedModes": [
            "drive"
        ],
        "distro": {
```

```
            "moduleType": "entry",
            "deliveryWithInstall": true,
            "moduleName": "hap-car"
        },
        "deviceType": [
            "car"
        ],

        "abilities": [
            {
                "name": ".MainAbility",
                "description": "himusic main ability",
                "icon": "$media:ic_launcher",
                "label": "HiMusic",
                "launchType": "standard",
                "orientation": "unspecified",
                "visible": true,
                "skills": [
                    {
                        "actions": [
                            "action.system.home"
                        ],
                        "entities": [
                            "entity.system.home"
                        ]
                    }
                ],
                "type": "page",
```

```
53.                 "formEnabled": false
54.             },
55.             {
56.                 "name": ".PlayService",
57.                 "description": "himusic play ability",
58.                 "icon": "$media:ic_launcher",
59.                 "label": "HiMusic",
60.                 "launchType": "standard",
61.                 "orientation": "unspecified",
62.                 "visible": false,
63.                 "skills": [
64.                     {
65.                         "actions": [
66.                             "action.play.music",
67.                             "action.stop.music"
68.                         ],
69.                         "entities": [
70.                             "entity.audio"
71.                         ]
72.                     }
73.                 ],
74.                 "type": "service",
75.                 "formEnabled": false,
76.                 "backgroundModes": [
77.                     "audioPlayback"
78.                 ]
79.             },
80.             {
```

```
81.                 "name": ".UserADataAbility",
82.                 "type": "data",
83.                 "uri":
"dataability://com.huawei.hiworld.himusic.UserADataAbility",
84.                 "visible": true
85.             }
86.         ],
87.         "reqPermissions": [{
88.                 "name": "ohos.permission.DISTRIBUTED_DATASYNC",
89.                 "reason": "",
90.                 "usedScene": {
91.                     "ability": [
92.
"com.huawei.hiworld.himusic.entry.MainAbility",
93.
"com.huawei.hiworld.himusic.entry.PlayService"
94.                     ],
95.                     "when": "inuse"
96.                 }
97.             }
98.         ]
99.     }
100.}
```

2.3 资源文件

2.3.1 资源文件分类

resources 目录

应用的资源文件（字符串、图片、音频等）统一存放于 resources 目录下，便于开发者使用和维护。resources 目录包括两大类目录，一类为 base 目录与限定词目录，另一类为 rawfile 目录，详见表 1。

资源目录示例：

```
resources
|---base  // 默认存在的目录
|   |---element
|   |   |---string.json
|   |---media
|   |   |---icon.png
|---en_GB-vertical-car-mdpi // 限定词目录示例，需要开发者自行创建
|   |---element
|   |   |---string.json
|   |---media
|   |   |---icon.png
|---rawfile  // 默认存在的目录
```

表 1 resources 目录分类

分类	base 目录与限定词目录	rawfile 目录
组织形式	按照两级目录形式来组织，目录命名必须符合规范，以便根据设备状态去匹配相应目录下的资源文件。 一级子目录为 **base 目录**和 **限定词目录**。 base 目录是默认存在的目录。当应用的 resources 资源目录中没有与设备状态匹配的限定词目录时，会自动引用该目录中的资源文件。 限定词目录需要开发者自行创建。目录名称由一个或多个表征应用场景或设备特征的限定词组合而成，具体要求参见限定词目录。 二级子目录为**资源目录**，用于存放字符串、颜色、布尔值等基础元素，以及媒体、动画、布局等资源文件，具体要求参见资源组目录。	支持创建多层子目录，目录名称可以自定义，文件夹内可以自由放置各类资源文件。rawfile 目录的文件不会根据设备状态去匹配不同的资源。
编译方式	目录中的资源文件会被编译成二进制文件，并赋予资源文件 ID。	目录中的资源文件会被直接打包进应用，不经过编译，也不会被赋予资源文件 ID。

引用方式	通过文件类型（type）和资源名称（name）的组合引用。 Java 文件采用：**ResourceTable.*type_name***。 特别地，如果引用的是系统资源，则采用：**ohos.global.systemres.ResourceTable.*type_name***。 XML 文件采用：**$*type:name***。 特别地，如果引用的是系统资源，则采用：**$ohos:*type:name***。	通过指定文件路径和文件名来引用。 例如：“resources/rawfile/example.js”。

限定词目录

限定词目录可以由一个或多个表征应用场景或设备特征的限定词组合而成，包括语言、文字、国家或地区、横竖屏、设备类型和屏幕密度等六个维度，限定词之间通过下划线（_）或者中划线（-）连接。开发者在创建限定词目录时，需要掌握限定词目录的命名要求以及与限定词目录与设备状态的匹配规则。

限定词目录的命名要求

- 限定词的组合顺序：*语言_文字_国家或地区-横竖屏-设备类型-屏幕密度*。开发者可以根据应用的使用场景和设备特征，选择其中的一类或几类限定词组成目录名称。

- 限定词的连接方式：语言、文字、国家或地区之间采用下划线（_）连接，除此之外的其他限定词之间均采用中划线（-）连接。例如：**zh_Hant_CN**、**zh_CN-car-ldpi**。
- 限定词的取值范围：每类限定词的取值必须符合表 2 中的条件，否则，将无法匹配目录中的资源文件。
-

表 2 限定词取值要求

限定词类型	含义与取值说明
语言	表示设备使用的语言类型，由 2 个小写字母组成。例如：zh 表示中文，en 表示英语。 详细取值范围，参见 **ISO 639-1**（ISO 制定的语言编码标准）。
文字	表示设备使用的文字类型，由 1 个大写字母（首字母）和 3 个小写字母组成。例如：Hans 表示简体中文，Hant 表示繁体中文。 详细取值范围，参见 **ISO 15924**（ISO 制定的文字编码标准）。
国家或地区	表示用户所在的国家或地区，由 2~3 个大写字母或者 3 个数字组成。例如：CN 表示中国，GB 表示英国。 详细取值范围，参见 **ISO 3166-1**（ISO 制定的国家和地区编码标准）。
横竖屏	表示设备的屏幕方向，取值如下： vertical：竖屏 horizontal：横屏
设备类型	表示设备的类型，取值如下： car：车机 tv：智慧屏 wearable：智能穿戴
屏幕密度	表示设备的屏幕密度（单位为 dpi），取值如下： sdpi：表示小规模的屏幕密度（Small-scale Dots Per Inch），

表 2 限定词取值要求

限定词类型	含义与取值说明
	适用于 120dpi 及以下的设备。 mdpi：表示中规模的屏幕密度（Medium-scale Dots Per Inch），适用于 120dpi~160dpi 的设备。 ldpi：表示大规模的屏幕密度（Large-scale Dots Per Inch），适用于 160dpi~240dpi 的设备。 xldpi：表示特大规模的屏幕密度（Extra Large-scale Dots Per Inch），适用于 240dpi~320dpi 的设备。 xxldpi：表示超大规模的屏幕密度（Extra Extra Large-scale Dots Per Inch），适用于 320dpi~480dpi 的设备。 xxxldpi：表示超特大规模的屏幕密度（Extra Extra Extra Large-scale Dots Per Inch），适用于 480dpi~640dpi 的设备。

限定词目录与设备状态的匹配规则

- 在为设备匹配对应的资源文件时，限定词目录匹配的优先级从高到低依次为：区域（语言_文字_国家或地区）> 横竖屏 > 设备类型 > 屏幕密度。
- 如果限定词目录中包含**语言、文字、横竖屏、设备类型**限定词，则对应限定词的取值必须与当前的设备状态完全一致，该目录才能够参与设备的资源匹配。例如，限定词目录“zh_CN-car-ldpi”不能参与“en_US”设备的资源匹配。

资源组目录

base 目录与限定词目录下面可以创建资源组目录（包括 element、media、animation、layout、graphic、profile），用于存放特定类型的资源文件，详见表3。

表 3 资源组目录 说明

资源组目录	目录说明	资源文件
element	表示元素资源，以下每一类数据都采用相应的 JSON 文件来表征。 boolean，布尔型 color，颜色 float，浮点型 intarray，整型数组 integer，整型 pattern，样式 plural，复数形式 strarray，字符串数组 string，字符串	element 目录中的文件名称建议与下面的文件名保持一致。每个文件中只能包含同一类型的数据。 boolean.json color.json float.json intarray.json integer.json pattern.json plural.json strarray.json string.json
media	表示媒体资源，包括图片、音频、视频等非文本格式的文件。	文件名可自定义，例如：icon.png。
animation	表示动画资源，采用 XML 文件格式。	文件名可自定义，例如：zoom_in.xml。
layout	表示布局资源，采用 XML 文件格式。	文件名可自定义，例如：home_layout.xml。
graphic	表示可绘制资源，采用 XML 文件格式。	文件名可自定义，例如：notifications_dark.xml。

表 3 资源组目录 说明

资源组目录	目录说明	资源文件
profile	表示其他类型文件，以原始文件形式保存。	文件名可自定义。

系统资源文件

目前支持的系统资源文件详见表 4。

表 4 系统资源文件说明

系统资源名称	含义	类型
ic_app	表示 HarmonyOS 应用的默认图标。	媒体
request_location_reminder_title	表示“请求使用设备定位功能”的提示标题。	字符串
request_location_reminder_content	表示“请求使用设备定位功能”的提示内容，即：请在下拉快捷栏打开"位置信息"开关。	字符串

2.3.2 资源文件示例

boolean. json 示例

```
{
    "boolean":[
        {
            "name":"boolean_1",
            "value":true
        },
        {
            "name":"boolean_ref",
            "value":"$boolean:boolean_1"
        }
    ]
}
```

color.json 示例

```
{
    "color":[
        {
            "name":"red",
            "value":"#ff0000"
        },
        {
            "name":"red_ref",
            "value":"$color:red"
        }
```

```
    ]
}
```

float.json 示例

```
{
    "float":[
        {
            "name":"float_1",
            "value":"30.6"
        },
        {
            "name":"float_ref",
            "value":"$float:float_1"
        },
        {
            "name":"float_px",
            "value":"100px"
        }
    ]
}
```

intarray.json 示例

```
{
    "intarray":[
        {
            "name":"intarray_1",
            "value":[
                100,
                200,
                "$integer:integer_1"
            ]
        }
    ]
}
```

integer.json 示例

```
{
    "integer":[
        {
            "name":"integer_1",
            "value":100
        },
        {
            "name":"integer_ref",
            "value":"$integer:integer_1"
        }
    ]
}
```

pattern.json 示例

```
{
    "pattern":[
        {
            "name":"base",
            "value":[
                {
                    "name":"width",
                    "value":"100vp"
                },
                 {
                    "name":"height",
                    "value":"100vp"
                },
                 {
                    "name":"size",
                    "value":"25px"
                }
            ]
        },
        {
            "name":"child",
            "parent":"base",
            "value":[
                {
```

```
                    "name":"noTitile",
                    "value":"Yes"
                }
            ]
        }
    ]
}
```

plural.json 示例

```
{
    "plural":[
        {
            "name":"eat_apple",
            "value":[
                {
                    "quantity":"one",
                    "value":"%d apple"
                },
                 {
                     "quantity":"other",
                     "value":"%d apples"
                 }
             ]
        }
    ]
```

```
17.}
```

strarray.json 示例

```
1.{
2.    "strarray":[
3.        {
4.            "name":"size",
5.            "value":[
6.                {
7.                    "value":"small"
8.                },
9.                {
10.                    "value":"$string:hello"
11.                },
12.                {
13.                    "value":"large"
14.                },
15.                {
16.                    "value":"extra large"
17.                }
18.            ]
19.        }
20.    ]
21.}
```

string.json 示例

```
{
    "string":[
        {
            "name":"hello",
            "value":"hello base"
        },
        {
            "name":"app_name",
            "value":"my application"
        },
        {
            "name":"app_name_ref",
            "value":"$string:app_name"
        },
        {
            "name":"app_sys_ref",
            "value":"$ohos:string:request_location_reminder_title"
        }
    ]
}
```

2.4 应用数据管理

HarmonyOS 应用数据管理支撑单设备的各种结构化数据的持久化，以及跨设备之间数据的同步、共享以及搜索功能。开发者通过应用数据管理，能够方便地完成应用程序数据在不同终端设备间的无缝衔接，满足用户跨设备使用数据的一致性体验。

本地应用数据管理

提供单设备上结构化数据的存储和访问能力。使用 SQLite 作为持久化存储引擎，提供了多种类型的本地数据库，分别是关系型数据库（Relational Database）和对象关系映射数据库（Object Relational Mapping Database），此外还提供一种轻量级偏好数据库（Light Weight Preference Database），用以满足开发人员使用不同数据模型对应用数据进行持久化和访问的需求。

有关于本地应用数据管理的详细信息，请参阅关系型数据库、对象关系映射数据库和轻量级偏好数据库。

分布式数据服务

分布式数据库支持用户数据跨设备相互同步，为用户提供在多种终端设备上一致的数据访问体验。通过调用分布式数据接口，应用可以将数据保存到分布式数据库中。通过结合帐号、应用唯一标识和数据库三元组，分布式数据库对属于不同应用的数据进行隔离。

有关于分布式数据库的详细信息，请参阅分布式数据服务。

分布式文件服务

在多个终端设备间为单个设备上应用程序创建的文件提供多终端的分布式共享能力。每台设备上都存储一份全量的文件元数据，应用程序通过文件元数据中的路径，可以实现同一应用文件的跨设备访问。

有关于分布式文件的详细信息，请参阅分布式文件服务。

数据搜索服务

在单个设备上，为应用程序提供搜索引擎级的全文索引管理、建立索引和搜索功能。

有关于数据搜索的详细信息，请参阅融合搜索。

数据存储管理

为应用开发者提供系统存储路径、存储设备列表，存储设备属性的查询和管理功能。

有关于数据存储的详细信息，请参阅数据存储管理。

2.5 应用权限管理

HarmonyOS 中所有的应用均在应用沙盒内运行。默认情况下，应用只能访问有限的系统资源，系统负责管理应用对资源的访问权限。

应用权限管理是由接口提供方（Ability）、接口使用方（应用）、系统（包括云侧和端侧）以及用户等多方共同参与的整个流程，保证受限接口是在约定好的规则下被正常使用，避免接口被滥用而导致用户、应用和设备受损。

权限声明

- 应用需要在 config.json 中使用“reqPermissions”属性对需要的权限逐个进行声明。
- 若使用到的三方库也涉及权限使用，也需统一在应用的 config.json 中逐个声明。
- 没有在 config.json 中声明的权限，应用就无法获得此权限的授权。

动态申请敏感权限

动态申请敏感权限基于用户可知可控的原则，需要应用在运行时主动调用系统动态申请权限的接口，系统弹框由用户授权，用户结合应用运行场景的上下文，识别出应用申请相应敏感权限的合理性，从而做出正确的选择。

即使用户向应用授予了请求的权限，应用在调用受此权限管控的接口前，也应该先检查自己有无此权限，而不能把之前授予的状态持久化，因为用户在动态授予后还可以通过设置取消应用的权限。

有关于应用动态申请敏感权限的详细信息，请参阅动态申请权限。

自定义权限

HarmonyOS 为了保证应用对外提供的接口不被恶意调用，需要对调用接口的调用者进行鉴权。

大多情况下，系统已定义的权限满足了应用的基本需要，若有特殊的访问控制需要，应用可在 config.json 中以"defPermissions": []属性来定义新的权限，并通过“availableScope”和“grantMode”两个属性分别确定权限的开放范围和授权方式，使得权限定义更加灵活且易于理解。有关 HarmonyOS 权限开放范围和授权方式详细的描述，请参阅权限授予方式字段说明和权限限制范围字段说明。

为了避免应用自定义新权限出现重名的情况，建议应用对新权限的命名以包名的前两个字段开头，这样可以防止不同开发者的应用间出现自定义权限重名的情况。

权限保护方法

- 保护 Ability：通过在 config.json 里对应的 Ability 中配置"permissions": ["*权限名*"]属性，即可实现保护整个 Ability 的目的，无指定权限的应用不能访问此 Ability。
- 保护 API：若 Ability 对外提供的数据或能力有多种，且开放范围或保护级别也不同，可以针对不同的数据或能力在接口代码实现中通过 verifyPermission(String permissionName, int pid, int uid)来对 uid 标识的调用者进行鉴权。

权限使用原则

- 权限申请最小化。跟用户提供的功能无关的权限，不要申请；尽量采用其他无需权限的操作来实现相应功能（如：通过 intent 拉起系统 UI 界面由用户交互、应用自己生成 uuid 代替设备 ID 等）。
- 权限申请完整。应用所需权限（包括应用调用到的三方库依赖的权限）都要逐个在应用的 config.json 中按格式声明。
- 满足用户可知。应用申请的敏感权限的目的需要真实准确告知用户。
- 权限就近申请。应用在用户触发相关业务功能时，就近提示用户授予实现此功能所需的权限。
- 权限不扩散。在用户未授权的情况下，不允许提供给其他应用使用。
- 应用自定义权限防止重名。建议以包名为前缀来命名权限，防止跟系统定义的权限重名。

2.6 应用隐私保护

随着移动终端及其相关业务（如移动支付、终端云等）的普及，用户隐私保护的重要性愈发突出。应用开发者在产品设计阶段就需要考虑保护的用户隐私，提高应用的安全性。HarmonyOS 应用开发需要遵从其隐私保护规则，在应用上架应用市场时，应用市场会根据规则进行校验，如不满足条件则无法上架。

数据收集及使用公开透明

应用采集个人数据时，应清晰、明确地告知用户，并确保告知用户的个人信息将被如何使用。

- 应用申请操作系统受限权限和敏感权限时，需要明确告知用户权限申请的目的和用途，并获取用户的同意。受限权限 API 使用方案请参考权限章节。详细的 UX 设计方案请参考 UX 设计隐私方案。

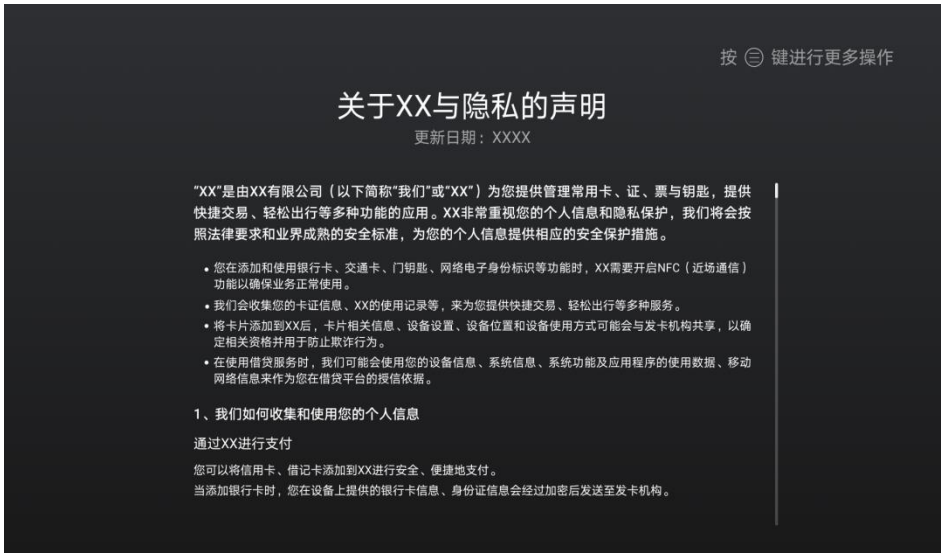
图 1　敏感权限获取弹框示例

- 开发者应制定并遵从适当的隐私政策，在收集、使用留存和第三方分享用户数据时需要符合所有适用法律、政策和规定。需充分告知用户处理个人数据的种类、目的、处理方式、保留期限等，满足数据主体权利等要求。

根据以上原则，我们设计了示例以供参考。隐私通知/声明的参考示例如下：

图 2　应用隐私通知示例图

图 3　应用隐私声明示例图



● 个人数据应当基于具体、明确、合法的目的收集，不应以与此目的不相符的方式作进一步处理。对于收集目的变更和用户撤销同意后再次使用的场景都需要用户重新同意。隐私声明变更示例图，隐私声明撤销同意示例图所示。

图 4　隐私声明变更示例图

图 5　撤销同意示例图

- 应用的隐私声明应覆盖本应用所有收集的个人数据。
- 有 UI 的 Ability 运行时需要在明显位置展示 Ability 的功能名称及开发者名称/logo。
- 应用的隐私声明应在应用首次启动时通过弹框等明显的方式展示给用户，并提供用户查看隐私声明的入口。
- 调用第三方 Ability 时，需要明确调用方与被调用方履行的隐私责任，并在声明弹框中告知数据主体相关隐私权责。
- 调用第三方 Ability 时，如涉及个人数据的分享，调用方需在隐私声明中说明分享的数据类型和数据接收者的类型。

数据收集及使用最小化

应用个人数据收集应与数据处理目的相关，且是适当、必要的。开发者应尽可能对个人数据进行匿名或化名，降低数据主体的风险。仅可收集和处理与特定目的相关且必需的个人数据，不能对数据做出与特定目的不相关的进一步处理。

- 敏感权限申请的时候要满足权限最小化的要求，在进行权限申请时，只申请获取必需的信息或资源所需要的权限。
- 应用针对数据的收集要满足最小化要求，不收集与应用提供服务无关联的数据。
- 数据使用的功能要求能够使用户受益，收集的数据不能用于与用户正常使用无关的功能。

数据处理选择和控制

对个人数据处理必须要征得用户的同意，用户对其个人数据要有充分的控制权。

- 应用申请使用系统权限：应用弹窗提醒，向用户呈现应用需要获取的权限和权限使用目的、应用需要收集的数据和使用目的等，通过用户点击“确认”的方式完成用户授权，让用户对应用权限的授予和使用透明、可知、可控。
- 用户可以修改、取消授予应用的权限：当用户不同意某一权限或者数据收集时，应当允许用户使用与这部分权限和数据收集不相关的功能。
- 在进入应用的主界面之前不建议直接弹窗申请敏感权限，仅在用户使用功能时才请求对应的权限。
- 系统对于用户的敏感数据和系统关键资源的获取设置了对应的权限，应用访问这些数据时需要申请对应的权限。相关权限列表请参考应用权限列表章节。

数据安全

从技术上保证数据处理活动的安全性，包括个人数据的加密存储、安全传输等安全机制，系统应默认开启或采取安全保护措施。

- **数据存储**

. 应用产生的密钥以及用户的敏感数据需要存储在应用的私有目录下，敏感数据定义可参考数据分类分级标准。

. 应用可以调用系统提供的本地数据库 RdbStore 的加密接口对敏感数据进行加密存储。接口详见关系型数据库章节。

. 应用产生的分布式数据可以调用系统的分布式数据库进行存储，对于敏感数据需要采用分布式数据库提供的加密接口进行加密，接口详见分布式数据服务章节。

- **安全传输**

需要分别针对本地传输和远程传输采取不同的安全保护措施。

本地传输：

- 应用通过 intent 跨应用传输数据时避免包含敏感数据，**intent scheme url 协议**使用过程中加入安全限制，防止 UXSS 等安全问题。
- 应用内组件调用应采用安全方式，避免通过隐式方式进行调用组件，防止组件劫持。
- 避免使用 socket 方式进行本地通信，如需使用，localhost 端口号随机生成，并对端口连接对象进行身份认证和鉴权。
- 本地 IPC 通信安全：作为服务提供方需要校验服务使用方的身份和访问权限，防止服务使用方进行身份仿冒或者权限绕过。
-

远程传输：

- 使用 https 代替 http 进行通信，并对 https 证书进行严格校验。
- 避免进行远程端口进行通信，如需使用，需要对端口连接对象进行身份认证和鉴权。
- 应用进行跨设备通信时，需要校验被访问设备和应用的身份信息，防止被访问方的设备和应用进行身份仿冒。
- 应用进行跨设备通信时，作为服务提供方需要校验服务使用方的身份和权限，防止服务使用方进行身份仿冒或者权限绕过。

本地化处理

应用开发的数据优先在本地进行处理，对于本地无法处理的数据上传云服务要满足最小化的原则，不能默认选择上传云服务。

未成年人数据保护要求

如果应用是针对未成年人设计的，或者应用通过收集的用户年龄数据识别出用户是未成年人，开发者应该结合目标市场国家的相关法律，专门分析未成年人个人数据保护的问题。收集未成年人数据前需要征得监护人的同意。

3.快速入门

3.1 简介

本文档适用于 HarmonyOS 应用开发的初学者。编写两个简单的页面，实现在第一个页面点击按钮跳转到第二个页面。

说明

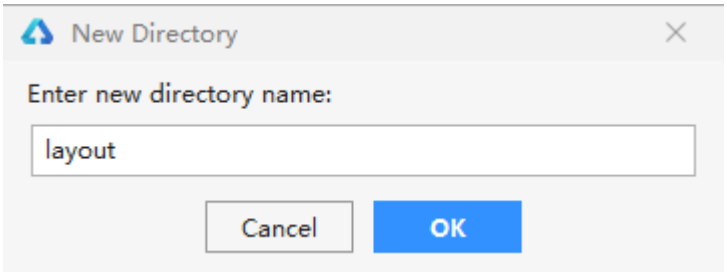
开始前，请参考 DevEco Studio 快速开始完成环境搭建、创建并运行一个项目。

3.2 编写第一个页面

在 Java UI 框架中，提供了两种编写布局的方式：在 XML 中声明 UI 布局和在代码中创建布局。这两种方式创建出的布局没有本质差别，为了熟悉两种方式，我们将通过 XML 的方式编写第一个页面，通过代码的方式编写第二个页面。

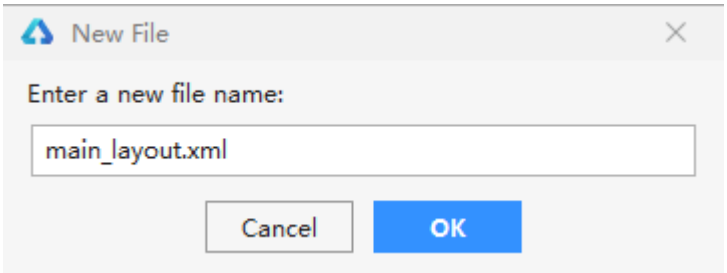
XML 编写页面

1. 在“Project”窗口，打开“entry > src > main > resources > base”，右键点击“base”文件夹，选择“New > Directory”，命名为“layout”。

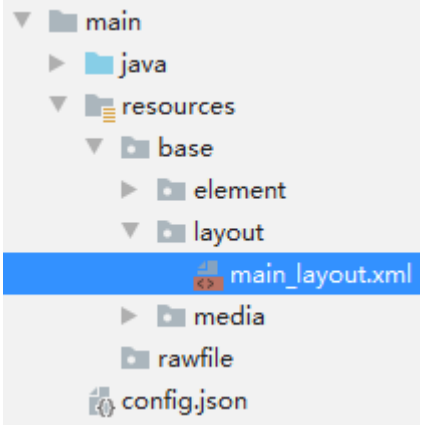


1.

2. 右键点击“layout”文件夹，选择“New > File”，命名为“main_layout.xml”。



在“layout”文件夹下可以看到新增了“main_layout.xml”文件。



1.

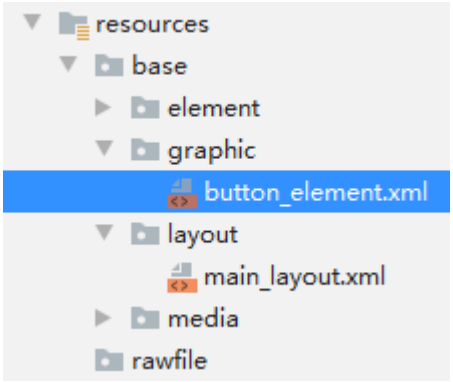
2. 打开“main_layout.xml”文件，添加一个文本和一个按钮，示例代码如下：

3.

```
.<?xml version="1.0" encoding="utf-8"?>
.<DependentLayout
.        xmlns:ohos="http://schemas.huawei.com/res/ohos"
.        ohos:width="match_parent"
.        ohos:height="match_parent"
.        ohos:background_element="#000000">
.    <Text
.            ohos:id="$+id:text"
.            ohos:width="match_content"
.            ohos:height="match_content"
```

```
.            ohos:center_in_parent="true"
.            ohos:text="Hello World"
.            ohos:text_color="white"
.            ohos:text_size="32fp"/>
.    <Button
.            ohos:id="$+id:button"
.            ohos:width="match_content"
.            ohos:height="match_content"
.            ohos:text_size="19fp"
.            ohos:text="Next"
.            ohos:top_padding="8vp"
.            ohos:bottom_padding="8vp"
.            ohos:right_padding="80vp"
.            ohos:left_padding="80vp"
.            ohos:text_color="white"
.            ohos:background_element="$graphic:button_element"
.            ohos:center_in_parent="true"
.            ohos:align_parent_bottom="true"/>
.</DependentLayout>
```

1. 上述按钮的背景是通过“button_element”来显示的，需要在“base”目录下创建“graphic”文件夹，在“graphic”文件夹中新建一个“button_element.xml”文件。



“button_element.xml”的示例代码如下：

```
.<?xml version="1.0" encoding="utf-8"?>
.<shape
.        xmlns:ohos="http://schemas.huawei.com/res/ohos"
.        ohos:shape="oval">
.    <solid
.            ohos:color="#007DFF"/>
.</shape>
```

加载 XML 布局

1. 在“Project”窗口中，选择“entry > src > main > java > com.example.helloworld > slice”，打开“MainAbilitySlice.java”文件。
2. 重写 onStart()方法加载 XML 布局，示例代码如下：

```
.package com.example.myapplication.slice;
.
.import com.example.myapplication.ResourceTable;
.import ohos.aafwk.ability.AbilitySlice;
.import ohos.aafwk.content.Intent;
.
.public class MainAbilitySlice extends AbilitySlice {
.
.    @Override
.    public void onStart(Intent intent) {
```

```
.         super.onStart(intent);
.         super.setUIContent(ResourceTable.Layout_main_layout); // 加载 XML 布局
.     }
.
.     @Override
.     public void onActive() {
.         super.onActive();
.     }
.
.     @Override
.     public void onForeground(Intent intent) {
.         super.onForeground(intent);
.     }
.}
```

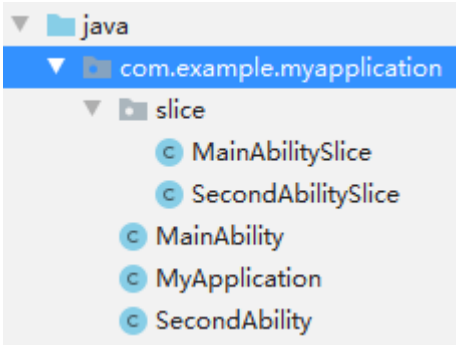
3. 请参考应用运行，效果如图所示：

3.3 创建另一个页面

创建 Feature Ability

1. 在“Project”窗口，打开“entry > src > main > java”，右键点击“com.example.myapplication”文件夹，选择“New > Ability > Empty Feature Ability(Java)”。

2. 配置 Ability 时，将“Page Name”设置为“SecondAbility”，点击“Finish”。创建完成后，可以看到新增了“SecondAbility”和“SecondAbilitySlice”文件。



代码编写界面

在上一节中，我们用 XML 的方式编写了一个包含文本和按钮的页面。为了帮助开发者熟悉在代码中创建布局的方式，接下来我们使用此方式编写第二个页面。

打开 “SecondAbilitySlice.java”文件，添加一个文本，示例代码如下：

```
1.package com.example.myapplication.slice;
2.
3.import ohos.aafwk.ability.AbilitySlice;
4.import ohos.aafwk.content.Intent;
5.import ohos.agp.colors.RgbColor;
```

```
6. import ohos.agp.components.DependentLayout;
7. import ohos.agp.components.DependentLayout.LayoutConfig;
8. import ohos.agp.components.Text;
9. import ohos.agp.components.element.ShapeElement;
10. import ohos.agp.utils.Color;
11.
12. import static
ohos.agp.components.ComponentContainer.LayoutConfig.MATCH_PARENT;
13. import static
ohos.agp.components.ComponentContainer.LayoutConfig.MATCH_CONTENT;
14.
15. public class SecondAbilitySlice extends AbilitySlice {
16.
17.     @Override
18.     public void onStart(Intent intent) {
19.         super.onStart(intent);
20.         // 声明布局
21.         DependentLayout myLayout = new DependentLayout(this);
22.         // 设置布局大小
23.         myLayout.setWidth(MATCH_PARENT);
24.         myLayout.setHeight(MATCH_PARENT);
25.         ShapeElement element = new ShapeElement();
26.         element.setRgbColor(new RgbColor(0, 0, 0));
27.         myLayout.setBackground(element);
28.
29.         // 创建一个文本
30.         Text text = new Text(this);
31.         text.setText("Nice to meet you.");
32.         text.setWidth(MATCH_PARENT);
```

```
33.         text.setTextSize(55);
34.         text.setTextColor(Color.WHITE);
35.         // 设置文本的布局
36.         DependentLayout.LayoutConfig textConfig = new
DependentLayout.LayoutConfig(MATCH_CONTENT,MATCH_CONTENT);
37.         textConfig.addRule(LayoutConfig.CENTER_IN_PARENT);
38.         text.setLayoutConfig(textConfig);
39.         myLayout.addComponent(text);
40.
41.         super.setUIContent(myLayout);
42.     }
43.
44.     @Override
45.     public void onActive() {
46.         super.onActive();
47.     }
48.
49.     @Override
50.     public void onForeground(Intent intent) {
51.         super.onForeground(intent);
52.     }
53.}
```

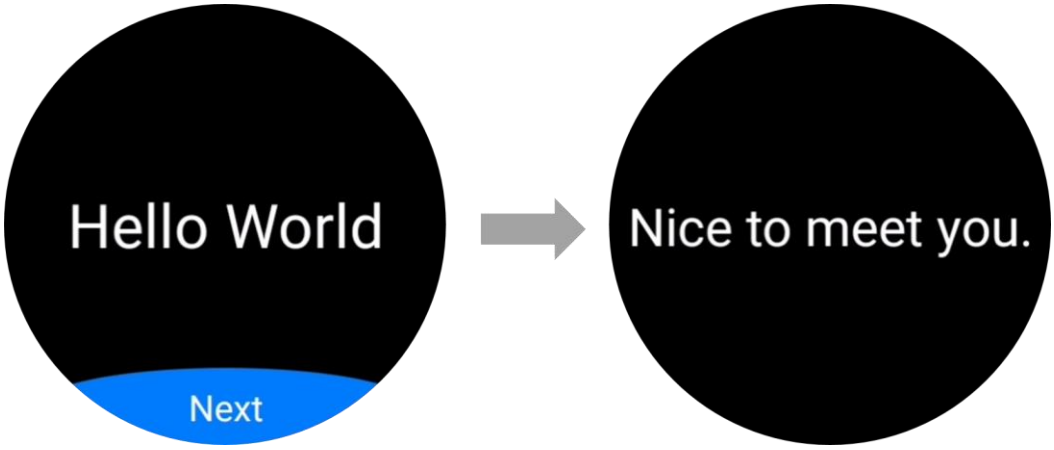
3.4 实现页面跳转

打开第一个页面的“MainAbilitySlice.java”文件，重写 onStart()方法添加按钮的响应逻辑，实现点击按钮跳转到下一页，示例代码如下：

```
.package com.example.myapplication.slice;

.

.import com.example.myapplication.ResourceTable;

.import ohos.aafwk.ability.AbilitySlice;

.import ohos.aafwk.content.Intent;

.import ohos.aafwk.content.Operation;

.import ohos.agp.components.*;

.

.public class MainAbilitySlice extends AbilitySlice {

.

.    @Override

.    public void onStart(Intent intent) {

.        super.onStart(intent);

.        super.setUIContent(ResourceTable.Layout_main_layout);

.        Button button = (Button)
findComponentById(ResourceTable.Id_button);

.

.        if (button != null) {

.            // 为按钮设置点击回调

.            button.setClickedListener(new
Component.ClickedListener() {

.                @Override

.                public void onClick(Component component) {

.                Intent secondIntent = new Intent();
```

```
.                // 指定待启动 FA 的 bundleName 和 abilityName
.                Operation operation = new Intent.OperationBuilder()
.                        .withDeviceId("")
.                        .withBundleName("com.example.myapplication")
.                        .withAbilityName("com.example.myapplication.
SecondAbility")
.                        .build();
.                secondIntent.setOperation(operation);
.                startAbility(secondIntent); // 通过 AbilitySlice 的
startAbility 接口实现启动另一个页面
.                }
.            });
.        }
.    }
.
.    @Override
.    public void onActive() {
.        super.onActive();
.    }
.
.    @Override
.    public void onForeground(Intent intent) {
.        super.onForeground(intent);
.    }
.}
```

再次运行项目，效果如图所示：



开发

开发指导

Ability

概述

Ability 是应用所具备能力的抽象，也是应用程序的重要组成部分。一个应用可以具备多种能力（即可以包含多个 Ability），HarmonyOS 支持应用以 Ability 为单位进行部署。Ability 可以分为 FA（Feature Ability）和 PA（Particle Ability）两种类型，每种类型为开发者提供了不同的模板，以便实现不同的业务功能。

- FA 支持 Page Ability：
  Page 模板是 FA 唯一支持的模板，用于提供与用户交互的能力。一个 Page 实例可以包含一组相关页面，每个页面用一个 AbilitySlice 实例表示。

- PA 支持 Service Ability 和 Data Ability：

  Service 模板：用于提供后台运行任务的能力。

  Data 模板：用于对外部提供统一的数据访问抽象。

在配置文件（config.json）中注册 Ability 时，可以通过配置 Ability 元素中的“type”属性来指定 Ability 模板类型，示例如下。

其中，“type”的取值可以为“page”、“service”或“data”，分别代表 Page 模板、Service 模板、Data 模板。为了便于表述，后文中我们将基于 Page 模板、Service 模板、Data 模板实现的 Ability 分别简称为 Page、Service、Data。

```
{
    "module": {
        ...
```

```
        "abilities": [
            {
                ...
                "type": "page"
                ...
            }
        ]
        ...
    }
    ...
}
```

Page Ability

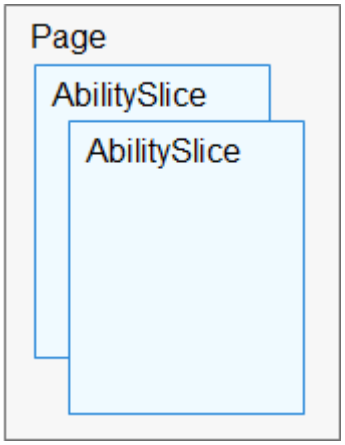
基本概念

Page 与 AbilitySlice

Page 模板（以下简称“Page”）是 FA 唯一支持的模板，用于提供与用户交互的能力。一个 Page 可以由一个或多个 AbilitySlice 构成，AbilitySlice 是指应用的单个页面及其控制逻辑的总和。

当一个 Page 由多个 AbilitySlice 共同构成时，这些 AbilitySlice 页面提供的业务能力应具有高度相关性。例如，新闻浏览功能可以通过一个 Page 来实现，其中包含了两个 AbilitySlice：一个 AbilitySlice 用于展示新闻列表，另一个 AbilitySlice 用于展示新闻详情。Page 和 AbilitySlice 的关系如图 1 所示。

图 1　Page 与 AbilitySlice



相比于桌面场景，移动场景下应用之间的交互更为频繁。通常，单个应用专注于某个方面的能力开发，当它需要其他能力辅助时，会调用其他应用提供的能力。例如，外卖应用提供了联系商家的业务功能入口，当用户在使用该功能时，会跳转到通话应用的拨号页面。与此类似，HarmonyOS 支持不同 Page 之间的跳转，并可以指定跳转到目标 Page 中某个具体的 AbilitySlice。

AbilitySlice 路由配置

虽然一个 Page 可以包含多个 AbilitySlice，但是 Page 进入前台时界面默认只展示一个 AbilitySlice。默认展示的 AbilitySlice 是通过 **setMainRoute()**方法来指定的。如果需要更改默认展示的 AbilitySlice，可以通过 **addActionRoute()**方法为此 AbilitySlice 配置一条路由规则。此时，当其他 Page 实例期望导航到此 AbilitySlice 时，可以在 Intent 中指定 Action，详见不同 Page 间导航。

setMainRoute()方法与 addActionRoute()方法的使用示例如下：

```
public class MyAbility extends Ability {
    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
        // set the main route
        setMainRoute(MainSlice.class.getName());

        // set the action route
        addActionRoute("action.pay", PaySlice.class.getName());
        addActionRoute("action.scan", ScanSlice.class.getName());
    }
}
```

addActionRoute()方法中使用的动作命名，需要在应用配置文件（config.json）中注册：

```
{
    "module": {
        "abilities": [
            {
                "skills":[
                    {
```

```
                        "actions":[
                            "action.pay",
                            "action.scan"
                        ]
                    }
                ]
                ...
            }
        ]
        ...
    }
    ...
}
```

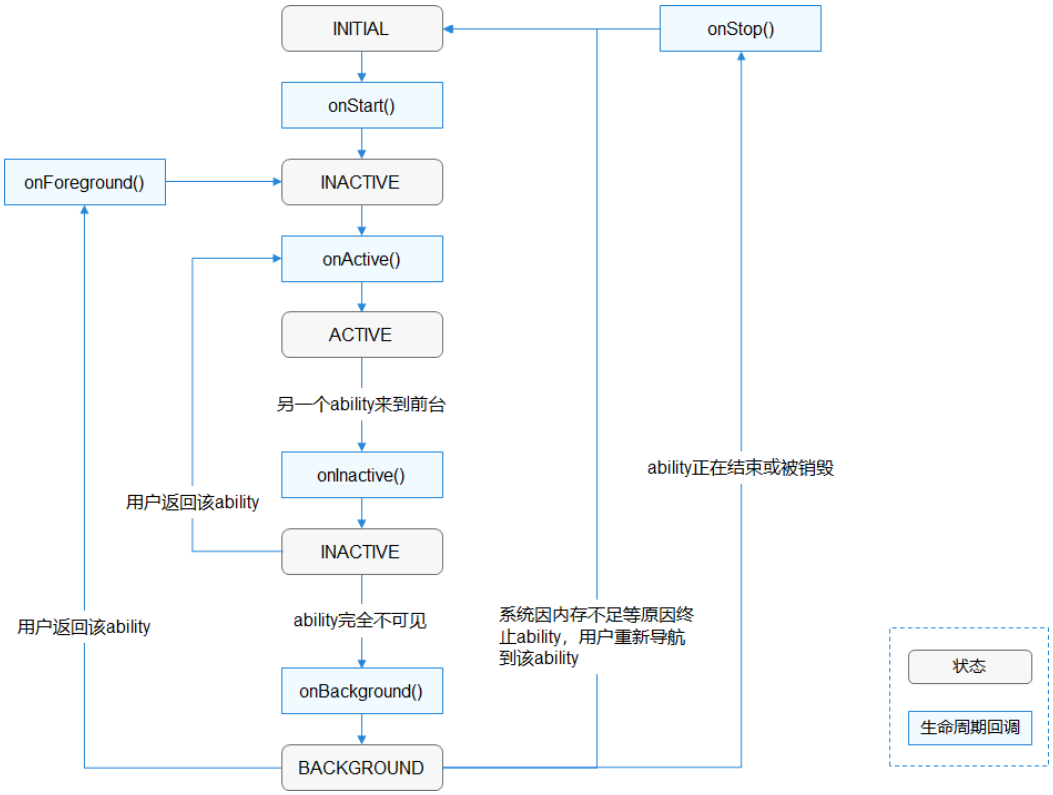
生命周期

系统管理或用户操作等行为均会引起 Page 实例在其生命周期的不同状态之间进行转换。Ability 类提供的回调机制能够让 Page 及时感知外界变化，从而正确地应对状态变化（比如释放资源），这有助于提升应用的性能和稳健性。

Page 生命周期回调

Page 生命周期的不同状态转换及其对应的回调，如图 1 所示。

图 1　Page 生命周期



- **onStart()**

当系统首次创建 Page 实例时，触发该回调。对于一个 Page 实例，该回调在其生命周期过程中仅触发一次，Page 在该逻辑后将进入 INACTIVE 状态。开发者必须重写该方法，并在此配置默认展示的 AbilitySlice。

```
    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
        super.setMainRoute(FooSlice.class.getName());
    }
```

- **onActive()**

Page 会在进入 INACTIVE 状态后来到前台，然后系统调用此回调。Page 在此之后进入 ACTIVE 状态，该状态是应用与用户交互的状态。Page 将保持在此状态，除非某类事件发生导致 Page 失去焦点，比如用户点击返回键或导航到其他 Page。当此类事件发生时，会触发 Page 回到 INACTIVE 状态，系统将调用 onInactive()回调。此后，Page 可能重新回到 ACTIVE 状态，系统将再次调用 onActive()回调。因此，开发者通常需要成对实现 onActive()和 onInactive()，并在 onActive()中获取在 onInactive()中被释放的资源。

- **onInactive()**

当 Page 失去焦点时，系统将调用此回调，此后 Page 进入 INACTIVE 状态。开发者可以在此回调中实现 Page 失去焦点时应表现的恰当行为。

- **onBackground()**

如果 Page 不再对用户可见，系统将调用此回调通知开发者用户进行相应的资源释放，此后 Page 进入 BACKGROUND 状态。开发者应该在此回调中释放 Page 不可见时无用的资源，或在此回调中执行较为耗时的状态保存操作。

- **onForeground()**

处于 BACKGROUND 状态的 Page 仍然驻留在内存中，当重新回到前台时（比如用户重新导航到此 Page），系统将先调用 onForeground()回调通知开发者，而后 Page 的生命周期状态回到 INACTIVE 状态。开发者应当在此回调中重新申请在 onBackground()中释放的资源，最后 Page 的生命周期状态进一步回到 ACTIVE 状态，系统将通过 onActive()回调通知开发者用户。

- **onStop()**

系统将要销毁 Page 时，将会触发此回调函数，通知用户进行系统资源的释放。销毁 Page 的可能原因包括以下几个方面：

- 用户通过系统管理能力关闭指定 Page，例如使用任务管理器关闭 Page。
- 用户行为触发 Page 的 terminateAbility()方法调用，例如使用应用的退出功能。
- 配置变更导致系统暂时销毁 Page 并重建。
- 系统出于资源管理目的，自动触发对处于 BACKGROUND 状态 Page 的销毁。

AbilitySlice 生命周期

AbilitySlice 作为 Page 的组成单元，其生命周期是依托于其所属 Page 生命周期的。AbilitySlice 和 Page 具有相同的生命周期状态和同名的回调，当 Page 生命周期发生变化时，它的 AbilitySlice 也会发生相同的生命周期变化。此外，AbilitySlice 还具有独立于 Page 的生命周期变化，这发生在同一 Page 中的 AbilitySlice 之间导航时，此时 Page 的生命周期状态不会改变。

AbilitySlice 生命周期回调与 Page 的相应回调类似，因此不再赘述。由于 AbilitySlice 承载具体的页面，开发者必须重写 AbilitySlice 的 onStart()回调，并在此方法中通过 setUIContent()方法设置页面，如下所示：

```
@Override
protected void onStart(Intent intent) {
    super.onStart(intent);

    setUIContent(ResourceTable.Layout_main_layout);
}
```

AbilitySlice 实例创建和管理通常由应用负责，系统仅在特定情况下会创建 AbilitySlice 实例。例如，通过导航启动某个 AbilitySlice 时，是由系统负责实例化;但是在同一个 Page 中不同的 AbilitySlice 间导航时则由应用负责实例化。

Page 与 AbilitySlice 生命周期关联

当 AbilitySlice 处于前台且具有焦点时，其生命周期状态随着所属 Page 的生命周期状态的变化而变化。当一个 Page 拥有多个 AbilitySlice 时，例如：MyAbility 下有 FooAbilitySlice 和 BarAbilitySlice，当前 FooAbilitySlice 处于前台并获得焦点，并即将导航到 BarAbilitySlice，在此期间的生命周期状态变化顺序为：

1. FooAbilitySlice 从 ACTIVE 状态变为 INACTIVE 状态。
2. BarAbilitySlice 则从 INITIAL 状态首先变为 INACTIVE 状态，然后变为 ACTIVE 状态（假定此前 BarAbilitySlice 未曾启动）。
3. FooAbilitySlice 从 INACTIVE 状态变为 BACKGROUND 状态。

对应两个 slice 的生命周期方法回调顺序为：

FooAbilitySlice.onInactive() --> BarAbilitySlice.onStart() --> BarAbilitySlice.onActive() --> FooAbilitySlice.onBackground()

在整个流程中，MyAbility 始终处于 ACTIVE 状态。但是，当 Page 被系统销毁时，其所有已实例化的 AbilitySlice 将联动销毁，而不仅是处于前台的 AbilitySlice。

AbilitySlice 间导航

同一 Page 内导航

当发起导航的 AbilitySlice 和导航目标的 AbilitySlice 处于同一个 Page 时，您可以通过 present()方法实现导航。如下代码片段展示通过点击按钮导航到其他 AbilitySlice 的方法：

```
@Override
protected void onStart(Intent intent) {


    ...
    Button button = ...;
    button.setClickedListener(listener -> present(new TargetSlice(),
new Intent()));
    ...


}
```

如果开发者希望在用户从导航目标 AbilitySlice 返回时，能够获得其返回结果，则应当使用 presentForResult()实现导航。用户从导航目标 AbilitySlice 返回时，系统将回调 onResult()来接收和处理返回结果，开发者需要重写该方法。返回结果由导航目标 AbilitySlice 在其生命周期内通过 setResult()进行设置。

```
@Override
protected void onStart(Intent intent) {
```

```
    ...
    Button button = ...;
    button.setClickedListener(listener -> presentForResult(new
TargetSlice(), new Intent(), 0));
    ...

}

@Override
protected void onResult(int requestCode, Intent resultIntent) {
    if (requestCode == 0) {
        // Process resultIntent here.
    }
}
```

系统为每个 Page 维护了一个 AbilitySlice 实例的栈，每个进入前台的 AbilitySlice 实例均会入栈。当开发者在调用 present()或 presentForResult()时指定的 AbilitySlice 实例已经在栈中存在时，则栈中位于此实例之上的 AbilitySlice 均会出栈并终止其生命周期。前面的示例代码中，导航时指定的 AbilitySlice 实例均是新建的，即便重复执行此代码（此时作为导航目标的这些实例是同一个类），也不会导致任何 AbilitySlice 出栈。

不同 Page 间导航

不同 Page 中的 AbilitySlice 相互不可见，因此无法通过 present()或 presentForResult()方法直接导航到其他 Page 的 AbilitySlice。AbilitySlice 作

为 Page 的内部单元，以 Action 的形式对外暴露，因此可以通过配置 Intent 的 Action 导航到目标 AbilitySlice。Page 间的导航可以使用 startAbility()或 startAbilityForResult()方法，获得返回结果的回调为 onAbilityResult()。在 Ability 中调用 setResult()可以设置返回结果。详细用法可参考根据 Operation 的其他属性启动应用中的示例。

跨设备迁移

跨设备迁移（下文简称“迁移”）支持将 Page 在同一用户的不同设备间迁移，以便支持用户无缝切换的诉求。以 Page 从设备 A 迁移到设备 B 为例，迁移动作主要步骤如下：

1.设备 A 上的 Page 请求迁移。

2.HarmonyOS 处理迁移任务，并回调设备 A 上 Page 的保存数据方法，用于保存迁移必须的数据。

3.HarmonyOS 在设备 B 上启动同一个 Page，并回调其恢复数据方法。

开发者可以参考以下详细步骤开发具有迁移功能的 Page。

实现 IAbilityContinuation 接口

- **onStartContinuation()**

Page 请求迁移后，系统首先回调此方法，开发者可以在此回调中决策当前是否可以执行迁移，比如，弹框让用户确认是否开始迁移。

- **onSaveData()**

如果 onStartContinuation()返回 true，则系统回调此方法，开发者在此回调中保存必须传递到另外设备上以便恢复 Page 状态的数据。

- **onRestoreData()**

源侧设备上 Page 完成保存数据后，系统在目标侧设备上回调此方法，开发者在此回调中接受用于恢复 Page 状态的数据。注意，在目标侧设备上的 Page 会重新启动其生命周期，无论其启动模式如何配置。且系统回调此方法的时机在 onStart()之前。

- **onCompleteContinuation()**

目标侧设备上恢复数据一旦完成，系统就会在源侧设备上回调 Page 的此方法，以便通知应用迁移流程已结束。开发者可以在此检查迁移结果是否成功，并在此处理迁移结束的动作，例如，应用可以在迁移完成后终止自身生命周期。

- **onRemoteTerminated()**

如果开发者使用 continueAbilityReversibly()而不是 continueAbility()，则此后可以在源侧设备上使用 reverseContinueAbility()进行回迁。这种场景下，相当于同一个 Page（的两个实例）同时在两个设备上运行，迁移完成后，如果目标侧设备上 Page 因任何原因终止，则源侧 Page 通过此回调接收终止通知。

说明

一个应用可能包含多个 Page，仅支持迁移的 Page 需要实现 IAbilityContinuation 接口。同时，此 Page 所包含的所有 AbilitySlice 也需要实现此接口。

请求迁移

实现 IAbilityContinuation 的 Page 可以在其生命周期内，调用 continueAbility()或 continueAbilityReversibly()请求迁移。两者的区别是，通过后者发起的迁移此后可以进行回迁。

```
try {
    continueAbility();
} catch (IllegalStateException e) {
    // Maybe another continuation in progress.
    ...
}
```

以 Page 从设备 A 迁移到设备 B 为例，详细的流程如下：

1. 设备 A 上的 Page 请求迁移。
2. 系统回调设备 A 上 Page 及其 AbilitySlice 栈中所有 AbilitySlice 实例的 IAbilityContinuation.onStartContinuation()方法，以确认当前是否可以立即迁移。

3. 如果可以立即迁移，则系统回调设备 A 上 Page 及其 AbilitySlice 栈中所有 AbilitySlice 实例的 IAbilityContinuation.onSaveData()方法，以便保存迁移后恢复状态必须的数据。

4. 如果保存数据成功，则系统在设备 B 上启动同一个 Page，并恢复 AbilitySlice 栈，然后回调 IAbilityContinuation.onRestoreData()方法，传递此前保存的数据；此后设备 B 上此 Page 从 onStart()开始其生命周期回调。

5. 系统回调设备 A 上 Page 及其 AbilitySlice 栈中所有 AbilitySlice 实例的 IAbilityContinuation.onCompleteContinuation()方法，通知数据恢复成功与否。

请求回迁

使用 continueAbilityReversibly()请求迁移并完成后，源侧设备上已迁移的 Page 可以发起回迁，以便使用户活动重新回到此设备。

```
try {
    reverseContinueAbility();
} catch (IllegalStateException e) {
    // Maybe another continuation in progress.
    ...
}
```

以 Page 从设备 A 迁移到设备 B 后并请求回迁为例，详细的流程如下：

(1) 设备 A 上的 Page 请求回迁。

(2) 系统回调设备 B 上 Page 及其 AbilitySlice 栈中所有 AbilitySlice 实例的 IAbilityContinuation.onStartContinuation()方法，以确认当前是否可以立即迁移。

(3) 如果可以立即迁移，则系统回调设备 B 上 Page 及其 AbilitySlice 栈中所有 AbilitySlice 实例的 IAbilityContinuation.onSaveData()方法，以便保存回迁后恢复状态必须的数据。

(4) 如果保存数据成功，则系统在设备 A 上 Page 恢复 AbilitySlice 栈，然后回调 IAbilityContinuation.onRestoreData()方法，传递此前保存的数据。

(5) 如果数据恢复成功，则系统终止设备 B 上 Page 的生命周期。

Service Ability

基本概念

基于 Service 模板的 Ability（以下简称“Service”）主要用于后台运行任务（如执行音乐播放、文件下载等），但不提供用户交互界面。Service 可由其他应用或 Ability 启动，即使用户切换到其他应用，Service 仍将在后台继续运行。

Service 是单实例的。在一个设备上，相同的 Service 只会存在一个实例。如果多个 Ability 共用这个实例，只有当与 Service 绑定的所有 Ability 都退出后，Service 才能够退出。由于 Service 是在主线程里执行的，因此，如果在 Service 里面的操作时间过长，开发者必须在 Service 里创建新的线程来处理（详见线程间通信），防止造成主线程阻塞，应用程序无响应。

创建 Service

介绍如何创建一个 Service。

1.创建 Ability 的子类，实现 Service 相关的生命周期方法。Service 也是一种 Ability，Ability 为 Service 提供了以下生命周期方法，用户可以重写这些方法来添加自己的处理。

- onStart()

该方法在创建 Service 的时候调用，用于 Service 的初始化，在 Service 的整个生命周期只会调用一次。

- onCommand()

在 Service 创建完成之后调用，该方法在客户端每次启动该 Service 时都会调用，用户可以在该方法中做一些调用统计、初始化类的操作。

- onConnect()

在 Ability 和 Service 连接时调用，该方法返回 IRemoteObject 对象，用户可以在该回调函数中生成对应 Service 的 IPC 通信通道，以便 Ability 与 Service 交互。Ability 可以多次连接同一个 Service，系统会缓存该 Service 的 IPC 通信对象，只有第一个客户端连接 Service 时，系统才会调用 Service 的 onConnect 方法来生成 IRemoteObject 对象，而后系统会将同一个 RemoteObject 对象传递至其他连接同一个 Service 的所有客户端，而无需再次调用 onConnect 方法。

- onDisconnect()

在 Ability 与绑定的 Service 断开连接时调用。

- onStop()

在 Service 销毁时调用。Service 应通过实现此方法来清理任何资源，如关闭线程、注册的侦听器等。

创建 Service 的代码示例如下：

```
public class ServiceAbility extends Ability {
    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
    }

    @Override
    public void onCommand(Intent intent, boolean restart, int startId)
{
        super.onCommand(intent, restart, startId);
    }

    @Override
    public IRemoteObject onConnect(Intent intent) {
        super.onConnect(intent);
        return null;
    }

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

    @Override
    public void onStop() {
        super.onStop();
    }
```

```
}
```

2.注册 Service。

Service 也需要在应用配置文件中进行注册,注册类型 type 需要设置为 service。

```
{
    "module": {
        "abilities": [
            {
                "name": ".ServiceAbility",
                "type": "service",
                "visible": true
                ...
            }
        ]
        ...
    }
    ...
}
```

启动 Service

介绍通过 startAbility()启动 Service 以及对应的停止方法。

- 启动 Service

Ability 为开发者提供了 startAbility()方法来启动另外一个 Ability。因为 Service 也是 Ability 的一种，开发者同样可以通过将 Intent 传递给该方法来启动 Service。不仅支持启动本地 Service，还支持启动远程 Service。

开发者可以通过构造包含 DeviceId、BundleName 与 AbilityName 的 Operation 对象来设置目标 Service 信息。这三个参数的含义如下：

- DeviceId：表示设备 ID。如果是本地设备，则可以直接留空；如果是远程设备，可以通过 ohos.distributedschedule.interwork.DeviceManager 提供的 getDeviceList 获取设备列表，详见《API 参考》。
- BundleName：表示包名称。
- AbilityName：表示待启动的 Ability 名称。

启动本地设备 Service 的代码示例如下：

```
Intent intent = new Intent();
Operation operation = new Intent.OperationBuilder()
        .withDeviceId("")
        .withBundleName("com.huawei.hiworld.himusic")
        .withAbilityName("com.huawei.hiworld.himusic.entry.ServiceAbility")
        .build();
intent.setOperation(operation);
startAbility(intent);
```

启动远程设备 Service 的代码示例如下：

```
Operation operation = new Intent.OperationBuilder()
```

```
        .withDeviceId("deviceId")
        .withBundleName("com.huawei.hiworld.himusic")
        .withAbilityName("com.huawei.hiworld.himusic.entry.ServiceAbili
ty")
        .withFlags(Intent.FLAG_ABILITYSLICE_MULTI_DEVICE) // 设置支持分布
式调度系统多设备启动的标识
        .build();
Intent intent = new Intent();
intent.setOperation(operation);
startAbility(intent);
```

执行上述代码后，Ability 将通过 startAbility() 方法来启动 Service。

- 如果 Service 尚未运行，则系统会先调用 onStart()来初始化 Service，再回调 Service 的 onCommand()方法来启动 Service。
- 如果 Service 正在运行，则系统会直接回调 Service 的 onCommand()方法来启动 Service。
- 停止 Service

Service 一旦创建就会一直保持在后台运行，除非必须回收内存资源，否则系统不会停止或销毁 Service。开发者可以在 Service 中通过 terminateAbility()停止本 Service 或在其他 Ability 调用 stopAbility()来停止 Service。

停止 Service 同样支持停止本地设备 Service 和停止远程设备 Service，使用方法与启动 Service 一样。一旦调用停止 Service 的方法，系统便会尽快销毁 Service。

连接 Service

如果 Service 需要与 Page Ability 或其他应用的 Service Ability 进行交互，则应创建用于连接的 Connection。Service 支持其他 Ability 通过 connectAbility()方法与其进行连接。

在使用 connectAbility()处理回调时，需要传入目标 Service 的 Intent 与 IAbilityConnection 的实例。IAbilityConnection 提供了两个方法供开发者实现：onAbilityConnectDone()用来处理连接的回调，onAbilityDisconnectDone()用来处理断开连接的回调。

连接 Service 的代码示例如下：

```
// 创建连接回调实例
private IAbilityConnection connection = new IAbilityConnection() {
    // 连接到 Service 的回调
    @Override
    public void onAbilityConnectDone(ElementName elementName,
IRemoteObject iRemoteObject, int resultCode) {
        // 在这里开发者可以拿到服务端传过来 IRemoteObject 对象，从中解
析出服务端传过来的信息
    }

    // 断开与连接的回调
    @Override
    public void onAbilityDisconnectDone(ElementName elementName, int
resultCode) {
    }
};
// 连接 Service
connectAbility(intent, connection);
```

同时，Service 侧也需要在 onConnect()时返回 IRemoteObject，从而定义与 Service 进行通信的接口。onConnect()需要返回一个 IRemoteObject 对象，HarmonyOS 提供了 IRemoteObject 的默认实现，用户可以通过继承 RemoteObject 来创建自定义的实现类。Service 侧把自身的实例返回给调用侧的代码示例如下：

```
// 创建自定义 IRemoteObject 实现类
private class MyRemoteObject extends RemoteObject {
    public MyRemoteObject() {
        super("MyRemoteObject");
    }
}

// 把 IRemoteObject 返回给客户端
@Override
protected IRemoteObject onConnect(Intent intent) {
    return new MyRemoteObject();
}
```

生命周期

与 Page 类似，Service 也拥有生命周期，如图 1 所示。根据调用方法的不同，其生命周期有以下两种路径：

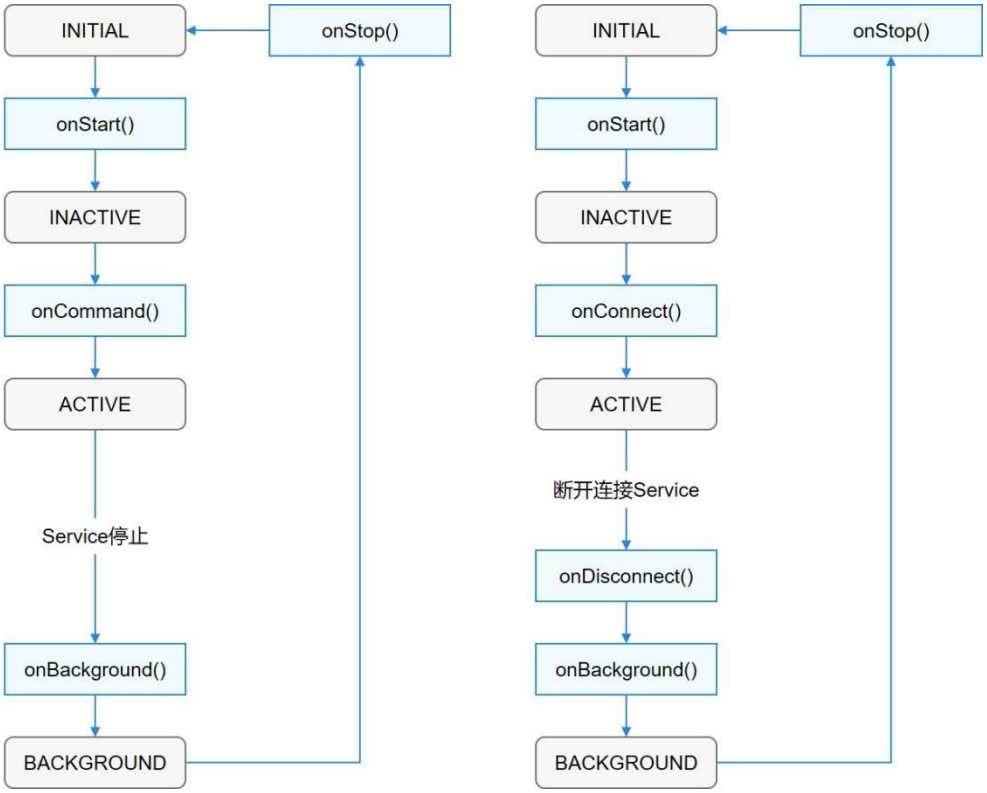
- 启动 Service

该 Service 在其他 Ability 调用 startAbility()时创建，然后保持运行。其他 Ability 通过调用 stopAbility()来停止 Service，Service 停止后，系统会将其销毁。

- 连接 Service

该 Service 在其他 Ability 调用 connectAbility()时创建，客户端可通过调用 disconnectAbility()断开连接。多个客户端可以绑定到相同 Service，而且当所有绑定全部取消后，系统即会销毁该 Service。

图 1 Service 生命周期



前台 Service

一般情况下，Service 都是在后台运行的，后台 Service 的优先级都是比较低的，当资源不足时，系统有可能回收正在运行的后台 Service。

在一些场景下（如播放音乐），用户希望应用能够一直保持运行，此时就需要使用前台 Service。前台 Service 会始终保持正在运行的图标在系统状态栏显示。

使用前台 Service 并不复杂，开发者只需在 Service 创建的方法里，调用 keepBackgroundRunning()将 Service 与通知绑定。调用 keepBackgroundRunning()方法前需要在配置文件中声明 ohos.permission.KEEP_BACKGROUND_RUNNING 权限，该权限是 normal 级别，同时还需要在配置文件中添加对应的 backgroundModes 参数。在 onStop()方法中调用 cancelBackgroundRunning()方法可停止前台 Service。

使用前台 Service 的 onStart()代码示例如下：

```
// 创建通知，其中 1005 为 notificationId
NotificationRequest request = new NotificationRequest(1005);
NotificationRequest.NotificationNormalContent content = new
NotificationRequest.NotificationNormalContent();
content.setTitle("title").setText("text");
NotificationRequest.NotificationContent notificationContent = new
NotificationRequest.NotificationContent(content);
request.setContent(notificationContent);


// 绑定通知，1005 为创建通知时传入的 notificationId
keepBackgroundRunning(1005, request);
```

在配置文件中配置如下：

```
{
    "name": ".ServiceAbility",
    "type": "service",
    "visible": true,
    "backgroundModes": ["dataTransfer","location"]
}
```

Data Ability

基本概念

使用 Data 模板的 Ability（以下简称“Data”）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。Data 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。

数据的存放形式多样，可以是数据库，也可以是磁盘上的文件。Data 对外提供对数据的增、删、改、查，以及打开文件等接口，这些接口的具体实现由开发者提供。

URI 介绍

Data 的提供方和使用方都通过 URI（Uniform Resource Identifier）来标识一个具体的数据，例如数据库中的某个表或磁盘上的某个文件。HarmonyOS 的 URI 仍基于 URI 通用标准，格式如下：

- scheme：协议方案名，固定为“dataability”，代表 Data Ability 所使用的协议类型。
- authority：设备 ID，如果为跨设备场景，则为目的设备的 IP 地址；如果为本地设备场景，则不需要填写。
- path：资源的路径信息，代表特定资源的位置信息。
- query：查询参数。
- fragment：可以用于指示要访问的子资源。

URI 示例：

- 跨设备场景：

dataability://*device_id*/com.huawei.dataability.persondata/person/10

- 本地设备：dataability:///com.huawei.dataability.persondata/person/10

访问 Data

开发者可以通过 DataAbilityHelper 类来访问当前应用或其他应用提供的共享数据。DataAbilityHelper 作为客户端，与提供方的 Data 进行通信。Data 接收到请求后，执行相应的处理，并返回结果。DataAbilityHelper 提供了一系列与 Data Ability 对应的方法。

下面介绍 DataAbilityHelper 具体的使用步骤。

声明使用权限

如果待访问的 Data 声明了访问需要权限，则访问此 Data 需要在配置文件中声明需要此权限。声明请参考权限申请字段说明。

```
"reqPermissions": [
    {
        "name": "com.example.myapplication5.DataAbility.DATA"
    }
]
```

创建 DataAbilityHelper

DataAbilityHelper 为开发者提供了 creator()方法来创建 DataAbilityHelper 实例。该方法为静态方法，有多个重载。最常见的方法是通过传入一个 context 对象来创建 DataAbilityHelper 对象。

获取 helper 对象示例：

```
DataAbilityHelper helper = DataAbilityHelper.creator(this);
```

访问 Data Ability

DataAbilityHelper为开发者提供了一系列的接口来访问不同类型的数据(文件、数据库等）。

- **访问文件**

DataAbilityHelper 为开发者提供了 FileDescriptor openFile(Uri uri, String mode)方法来操作文件。此方法需要传入两个参数，其中 uri 用来确定目标资源路径，mode 用来指定打开文件的方式，可选方式包含“r”(读)，“w”(写)，“rw”(读写)，“wt”(覆盖写)，“wa”(追加写)，“rwt”(覆盖写且可读)。

该方法返回一个目标文件的 FD（文件描述符），把文件描述符封装成流，开发者就可以对文件流进行自定义处理。

访问文件示例：

```
// 读取文件描述符
FileDescriptor fd = helper.openFile(uri, "r");
FileInputStream fis = new FileInputStream(fd);
```

访问数据库

DataAbilityHelper 为开发者提供了增、删、改、查以及批量处理等方法来操作数据库。

方法	描述
ResultSet query(Uri uri, String[] columns, DataAbilityPredicates predicates)	查询数据库
int insert(Uri uri, ValuesBucket value)	向数据库中插入单条数据
int batchInsert(Uri uri, ValuesBucket[] values)	向数据库中插入多条数据
int delete(Uri uri, DataAbilityPredicates predicates)	删除一条或多条数据
int update(Uri uri, ValuesBucket value, DataAbilityPredicates predicates)	更新数据库
DataAbilityResult[] executeBatch (ArrayList<DataAbilityOperation> operations)	批量操作数据库

这些方法的使用说明如下：

- query()

查询方法，其中 uri 为目标资源路径，columns 为想要查询的字段。开发者的查询条件可以通过 DataAbilityPredicates 来构建。查询用户表中 id 在 101-103 之间的用户，并把结果打印出来，代码示例如下：

```
DataAbilityHelper helper = DataAbilityHelper.creator(this);

// 构造查询条件
DataAbilityPredicates predicates = new DataAbilityPredicates();
predicates.between("userId", 101, 103);

// 进行查询
ResultSet resultSet = helper.query(uri,columns,predicates);

// 处理结果
resultSet.goToFirstRow();
do{
    // 在此处理 ResultSet 中的记录;
}while(resultSet.goToNextRow());
```

insert()

新增方法，其中 uri 为目标资源路径，ValuesBucket 为要新增的对象。插入一条用户信息的代码示例如下：

```
DataAbilityHelper helper = DataAbilityHelper.creator(this);


// 构造插入数据
ValuesBucket valuesBucket = new ValuesBucket();
valuesBucket.putString("name", "Tom");
valuesBucket.putInteger("age", 12);
helper.insert(uri, valuesBucket);
```

batchInsert()

批量插入方法，和 insert()类似。批量插入用户信息的代码示例如下：

```
DataAbilityHelper helper = DataAbilityHelper.creator(this);


// 构造插入数据
ValuesBucket[] values = new ValuesBucket[2];
value[0] = new ValuesBucket();
value[0].putString("name", "Tom");
value[0].putInteger("age", 12);
value[1] = new ValuesBucket();
value[1].putString("name", "Tom1");
value[1].putInteger("age", 16);
helper.batchInsert(uri, values);
```

update()

更新方法，更新数据由 ValuesBucket 传入，更新条件由 DataAbilityPredicates 来构建。更新 id 为 102 的用户，代码示例如下：

```
DataAbilityHelper helper = DataAbilityHelper.creator(this);


// 构造更新条件
DataAbilityPredicates predicates = new DataAbilityPredicates();
predicates.equalTo("userId",102);


// 构造更新数据
ValuesBucket valuesBucket = new ValuesBucket();
valuesBucket.putString("name", "Tom");
valuesBucket.putInteger("age", 12);
helper.update(uri, valuesBucket, predicates);
```

executeBatch()

此方法用来执行批量操作。DataAbilityOperation 中提供了设置操作类型、数据和操作条件的方法，开发者可自行设置自己要执行的数据库操作。插入多条数据的代码示例如下：

```
DataAbilityHelper helper = DataAbilityHelper.creator(abilityObj,
insertUri);


// 构造批量操作
ValuesBucket value1 = initSingleValue();
DataAbilityOperation opt1 =
DataAbilityOperation.newInsertBuilder(insertUri).withValuesBucket(val
ue1).build();
ValuesBucket value2 = initSingleValue2();
DataAbilityOperation opt2 =
DataAbilityOperation.newInsertBuilder(insertUri).withValuesBucket(val
ue2).build();
```

```
ArrayList<DataAbilityOperation> operations = new
ArrayList<DataAbilityOperation>();

operations.add(opt1);

operations.add(opt2);

DataAbilityResult[] result = helper.executeBatch(insertUri,
operations);
```

创建 Data

使用 Data 模板的 Ability 形式仍然是 Ability，因此，开发者需要为应用添加一个或多个 Ability 的子类，来提供程序与其他应用之间的接口。Data 为结构化数据和文件提供了不同 API 接口供用户使用，因此，开发者需要首先确定好使用何种类型的数据。本章节主要讲述了创建 Data 的基本步骤和需要使用的接口。

确定数据存储方式

确定数据的存储方式，Data 支持以下两种数据形式：

- 文件数据：如文本、图片、音乐等。
- 结构化数据：如数据库等。

实现 UserDataAbility

UserDataAbility 接收其他应用发送的请求，提供外部程序访问的入口，从而实现应用间的数据访问。Data 提供了文件存储和数据库存储两组接口供用户使用。

文件存储

开发者需要在 Data 中重写 FileDescriptor openFile(Uri uri, String mode)方法来操作文件：uri 为客户端传入的请求目标路径；mode 为开发者对文件的操作选项，可选方式包含“r”(读), “w”(写), “rw”(读写)等。

MessageParcel 类提供了一个静态方法，用于获取 MessageParcel 实例。通过 dupFileDescriptor()函数复制待操作文件流的文件描述符，并将其返回，供远端应用使用。

示例：根据传入的 uri 打开对应的文件

```
// 创建 messageParcel
MessageParcel messageParcel = MessageParcel.obtain();
```

```
File file = new File(uri.getDecodedPathList().get(1));
if (mode == null || !"rw".equals(mode)) {
    file.setReadOnly();
}
FileInputStream fileIs = new FileInputStream(file);
FileDescriptor fd = fileIs.getFD();

// 绑定文件描述符
return messageParcel.dupFileDescriptor(fd);
```

数据库存储

1. 初始化数据库连接。

系统会在应用启动时调用 onStart()方法创建 Data 实例。在此方法中，开发者应该创建数据库连接，并获取连接对象，以便后续和数据库进行操作。为了避免影响应用启动速度，开发者应当尽可能将非必要的耗时任务推迟到使用时执行，而不是在此方法中执行所有初始化。

示例：初始化的时候连接数据库

```
private static final String DATABASE_NAME ="UserDataAbility.db";
private static final String DATABASE_NAME_ALIAS = "UserDataAbility";
private OrmContext ormContext = null;

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

```
    DatabaseHelper manager = new DatabaseHelper(this);
    ormContext = manager.getOrmContext(DATABASE_NAME_ALIAS,
DATABASE_NAME, BookStore.class);
}
```

编写数据库操作方法。

Ability 定义了 6 个方法供用户处理对数据库表数据的增删改查。这 6 个方法在 Ability 中已默认实现，开发者可按需重写。

方法	描述
ResultSet query(Uri uri, String[] columns, DataAbilityPredicates predicates)	查询数据库
int insert(Uri uri, ValuesBucket value)	向数据库中插入单条数据
int batchInsert(Uri uri, ValuesBucket[] values)	向数据库中插入多条数据
int delete(Uri uri, DataAbilityPredicates predicates)	删除一条或多条数据
int update(Uri uri, ValuesBucket value, DataAbilityPredicates predicates)	更新数据库
DataAbilityResult[] executeBatch(ArrayList<DataAbilityOperation> operations)	批量操作数据库

这些方法的使用说明如下：

- query()

该方法接收三个参数，分别是查询的目标路径，查询的列名，以及查询条件，查询条件由类 DataAbilityPredicates 构建。根据传入的列名和查询条件查询用户表的代码示例如下：

```
public ResultSet query(Uri uri, String[] columns, DataAbilityPredicates predicates) {
    if (ormContext == null) {
        HiLog.error(this.getClass().getSimpleName(), "failed to query, ormContext is null");
        return null;
    }

    // 查询数据库
    OrmPredicates ormPredicates = DataAbilityUtils.createOrmPredicates(predicates,User.class);
    ResultSet resultSet = ormContext.query(ormPredicates, columns);
    if (resultSet == null) {
        HiLog.info(this.getClass(), "resultSet is null");
    }

    // 返回结果
    return resultSet;
}
```

insert()

该方法接收两个参数，分别是插入的目标路径和插入的数据值。其中，插入的数据由 ValuesBucket 封装，服务端可以从该参数中解析出对应的属性，然后插入

到数据库中。此方法返回一个 int 类型的值用于标识结果。接收到传过来的用户信息并把它保存到数据库中的代码示例如下：

```
public int insert(Uri uri, ValuesBucket value) {
    // 参数校验
    if (ormContext == null) {
        HiLog.error(this.getClass().getSimpleName(), "failed to insert, ormContext is null");
        return -1;
    }
    String path = uri.getPath();

    if (buildPathMatcher().getPathId(path) != PathId) {
        HiLog.info(this.getClass(), "UserDataAbility insert path is not matched");
        return -1;
    }

    // 构造插入数据
    User user = new User();
    user.setUserId(value.getInteger("userId"));
    user.setFirstName(value.getString("firstName"));
    user.setLastName(value.getString("lastName"));
    user.setAge(value.getInteger("age"));
    user.setBalance(value.getDouble("balance"));

    // 插入数据库
    boolean isSuccessed = true;
```

```
    try {
        isSuccessed = ormContext.insert(user);
    } catch (DataAbilityRemoteException e) {
        HiLog.error(TAG, "insert fail: " + e.getMessage());
        throw new RuntimeException(e);
    }
    if (!isSuccessed) {
        HiLog.error(this.getClass().getSimpleName(), "failed to
insert");
        return -1;
    }
    isSuccessed = ormContext.flush();
    if (!isSuccessed) {
        HiLog.error(this.getClass().getSimpleName(), "failed to insert
flush");
        return -1;
    }
    DataAbilityHelper.creator(this, uri).notifyChange(uri);
    int id = Math.toIntExact(user.getRowId());
    return id;
}
```

- batchInsert()

该方法为批量插入方法，接收一个 ValuesBucket 数组用于单次插入一组对象。它的作用是提高插入多条重复数据的效率。该方法系统已实现，开发者可以直接调用。

- delete()

该方法用来执行删除操作。删除条件由类 DataAbilityPredicates 构建，服务端在接收到该参数之后可以从中解析出要删除的数据，然后到数据库中执行。根据传入的条件删除用户表数据的代码示例如下：

```
public int delete(Uri uri, DataAbilityPredicates predicates) {
    if (ormContext == null) {
        HiLog.error(this.getClass().getSimpleName(), "failed to
delete, ormContext is null");
        return -1;
    }

    OrmPredicates ormPredicates =
DataAbilityUtils.createOrmPredicates(predicates,User.class);
    int value = ormContext.delete(ormPredicates);
    DataAbilityHelper.creator(this, uri).notifyChange(uri);
    return value;
}
```

update()

此方法用来执行更新操作。用户可以在 ValuesBucket 参数中指定要更新的数据，在 DataAbilityPredicates 中构建更新的条件等。更新用户表的数据的代码示例如下：

```
public int update(Uri uri, ValuesBucket value, DataAbilityPredicates
predicates) {
    if (ormContext == null) {
        HiLog.error(this.getClass().getSimpleName(), "failed to update,
ormContext is null");
        return -1;
    }

    OrmPredicates ormPredicates =
DataAbilityUtils.createOrmPredicates(predicates,User.class);
    int index = ormContext.update(ormPredicates, value);
    HiLog.info(this.getClass(), "UserDataAbility update value:" +
index);
    DataAbilityHelper.creator(this, uri).notifyChange(uri);
    return index;
}
```

- executeBatch()

此方法用来批量执行操作。DataAbilityOperation 中提供了设置操作类型、数据和操作条件的方法，用户可自行设置自己要执行的数据库操作。该方法系统已实现，开发者可以直接调用。

注册 UserDataAbility

和 Service 类似，开发者必须在配置配置文件中注册 Data。并且配置以下属性：

- type: 类型设置为 data
- uri: 对外提供的访问路径，全局唯一
- permissions: 访问该 data ability 时需要申请的访问权限

```
{
    "name": ".UserDataAbility",
     "type": "data",
     "visible": true,
     "uri":
"dataability://com.example.myapplication5.DataAbilityTest",
     "permissions": [
        "com.example.myapplication5.DataAbility.DATA"
     ]
}
```

Intent

基本概念

Intent 是对象之间传递信息的载体。例如，当一个 Ability 需要启动另一个 Ability 时，或者一个 AbilitySlice 需要导航到另一个 AbilitySlice 时，可以通过 Intent 指定启动的目标同时携带相关数据。Intent 的构成元素包括 Operation 与 Parameters，具体描述参见表 1。

表 1 Intent 的构成元素

属性	子属性	描述
Operation	Action	表示动作，通常使用系统预置 Action，应用也可以自定义 Action。例如 IntentConstants.ACTION_HOME 表示返回桌面动作。
	Entity	表示类别，通常使用系统预置 Entity，应用也可以自定义 Entity。例如 Intent.ENTITY_HOME 表示在桌面显示图标。
	Uri	表示 Uri 描述。如果在 Intent 中指定了 Uri，则 Intent 将匹配指定的 Uri 信息，包括 scheme, schemeSpecificPart, authority 和 path 信息。
	Flags	表示处理 Intent 的方式。例如 Intent.FLAG_ABILITY_CONTINUATION 标记在本地的一个 Ability 是否可以迁移到远端设备继续运行。
	BundleName	表示包描述。如果在 Intent 中同时指定了 BundleName 和 AbilityName，则 Intent 可以直接匹配到指定的 Ability。

表 1 Intent 的构成元素

属性	子属性	描述
	AbilityName	表示待启动的 Ability 名称。如果在 Intent 中同时指定了 BundleName 和 AbilityName，则 Intent 可以直接匹配到指定的 Ability。
	DeviceId	表示运行指定 Ability 的设备 ID。
Parameters	-	Parameters 是一种支持自定义的数据结构，开发者可以通过 Parameters 传递某些请求所需的额外信息。

当 Intent 用于发起请求时，根据指定元素的不同，分为两种类型：

- 如果同时指定了 BundleName 与 AbilityName，则根据 Ability 的全称（例如，“com.demoapp.FooAbility”）来直接启动应用。
- 如果未同时指定 BundleName 和 AbilityName，则根据 Operation 中的其他属性来启动应用。

根据 Ability 的全称启动应用

通过构造包含 BundleName 与 AbilityName 的 Operation 对象，可以启动一个 Ability、并导航到该 Ability。示例代码如下：

```
Intent intent = new Intent();


// 通过 Intent 中的 OperationBuilder 类构造 operation 对象，指定设备标识（空串表示当前设备）、应用包名、Ability 名称
Operation operation = new Intent.OperationBuilder()
        .withDeviceId("")
        .withBundleName("com.demoapp")
        .withAbilityName("com.demoapp.FooAbility")
```

```
        .build();

// 把 operation 设置到 intent 中
intent.setOperation(operation);
startAbility(intent);
```

作为处理请求的对象，会在相应的回调方法中接收请求方传递的 Intent 对象。以导航到另一个 Ability 为例，导航的目标 Ability 可以在其 onStart()回调的参数中获得 Intent 对象。

根据 Operation 的其他属性启动应用

有些场景下，开发者需要在应用中使用其他应用提供的某种能力，而不感知提供该能力的具体是哪一个应用。例如开发者需要通过浏览器打开一个链接，而不关心用户最终选择哪一个浏览器应用，则可以通过 Operation 的其他属性（除 BundleName 与 AbilityName 之外的属性）描述需要的能力。如果设备上存在多个应用提供同种能力，系统则弹出候选列表，由用户选择由哪个应用处理请求。以下示例展示使用 Intent 跨 Ability 查询天气信息。

请求方

在 Ability 中构造 Intent 以及包含 Action 的 Operation 对象，并调用 startAbilityForResult()方法发起请求。然后重写 onAbilityResult()回调方法，对请求结果进行处理。

```
private void queryWeather() {
    Intent intent = new Intent();
    Operation operation = new Intent.OperationBuilder()
            .withAction(Intent.ACTION_QUERY_WEATHER)
            .build();
    intent.setOperation(operation);
    startAbilityForResult(intent, REQ_CODE_QUERY_WEATHER);
}


@Override
protected void onAbilityResult(int requestCode, int resultCode, Intent
resultData) {
    switch (requestCode) {
        case REQ_CODE_QUERY_WEATHER:
            // Do something with result.
            ...
            return;
        default:
            ...
    }
}
```

处理方

1. 作为处理请求的对象，首先需要在配置文件中声明对外提供的能力，以便系统据此找到自身并作为候选的请求处理者。

```
{
    "module": {
        ...
        "abilities": [
            {
                ...
                "skills":[
                    {
                        "actions":[
                            "ability.intent.QUERY_WEATHER"
                        ]
                    }
                ]
                ...
            }
        ]
        ...
    }
    ...
}
```

在 Ability 中配置路由以便支持以此 action 导航到对应的 AbilitySlice。

```
@Override
protected void onStart(Intent intent) {
    ...
    addActionRoute(Intent.ACTION_QUERY_WEATHER,
DemoSlice.class.getName());
    ...
}
```

在 Ability 中处理请求，并调用 setResult()方法暂存返回结果。

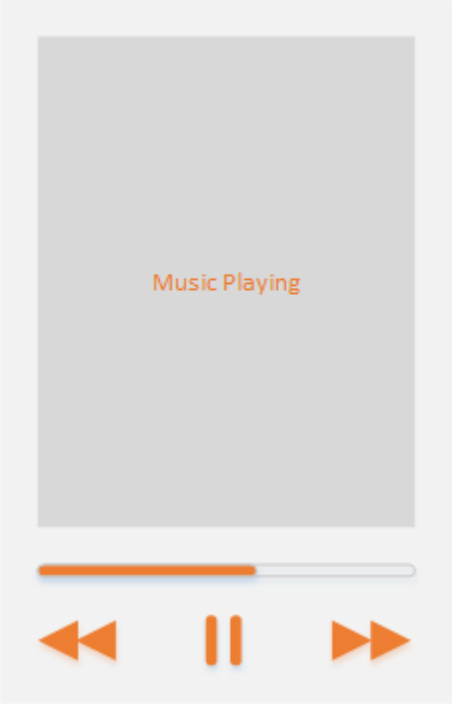
```
@Override
protected void onActive() {
    ...
    Intent resultIntent = new Intent();
    setResult(0, resultIntent);
    ...
}
```

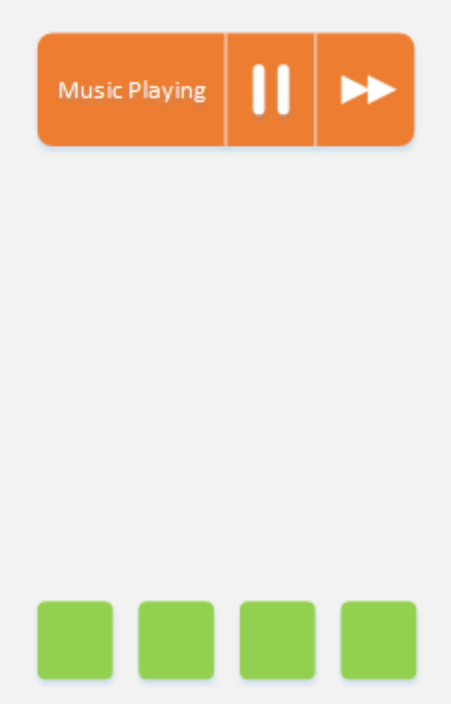
Ability Form

基本概念

Ability Form，即表单，是 Page 形态的 Ability 的一种界面展示形式，用于嵌入到其他应用中作为其界面的一部分显示，并支持基础的交互功能。表单使用方作为表单展示的宿主负责显示表单，表单使用方的典型应用就是桌面。下图展示一种音乐播放应用 Page 的完整显示及其微缩展示效果。

图 1　Page 及其表单





表单提供方

表单提供方是一个 Page 形态的 Ability，需要实现 onCreateForm()方法，并返回一个 AbilityForm 对象。创建 AbilityForm 对象时需要指定表单布局文件。

1. 为表单定义布局文件。创建布局文件 *form_layout.xml*：

```
<?xml version="1.0" encoding="utf-8"?>
<DirectionLayout xmlns:ohos="http://schemas.huawei.com/res/ohos"
                ohos:width="match_parent"
                ohos:height="match_parent"
                ohos:orientation="horizontal">
    <Text
            ohos:id="$+id:text01"
            ohos:width="match_content"
            ohos:height="match_content"
            ohos:text="Counter"
            ohos:text_color="#FF555555"
            ohos:text_size="20fp"/>
    <Text
            ohos:id="$+id:text02"
            ohos:width="match_content"
            ohos:height="match_content"
            ohos:text_color="#FF0000FF"
            ohos:text_size="20fp"/>
</DirectionLayout>
```

实现 onCreateForm()方法，并为表单视图控件注册回调。

```
public class FormAbility extends Ability {
    private static AbilityForm abilityForm;
    private static int clickTimes = 0;
```

```
    ...
    @Override
    public AbilityForm onCreateForm() {
        abilityForm = new AbilityForm(ResourceTable.Layout_form_layout,
this);

        abilityForm.setText(ResourceTable.Id_text02,
generateFormText());
        abilityForm.registerViewListener(ResourceTable.Id_text02, new
OnClickListener() {
            @Override
            public void onClick(int viewId, AbilityForm form, ViewsStatus
viewsStatus) {
                clickTimes++;
                form.setText(viewId, generateFormText());
                if (FormSlice.text != null) {
                    FormSlice.text.setText("Client.Counter: " +
clickTimes);
                }
            }
        });
        return abilityForm;
    }

    private static String generateFormText() {
        return "total: " + clickTimes;
    }
}
```

在 FormSlice 实现 text，用于展示 Form 点击效果（用例展示效果，非实现 Form 必须步骤）。

```
public class FormSlice extends AbilitySlice {
    public static Text text;

    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);

        PositionLayout positionLayout = new PositionLayout(this);

        ShapeElement background = new ShapeElement();
        background.setShape(ShapeElement.RECTANGLE);
        background.setRgbColor(new RgbColor(0xFFFFFFFF));
        positionLayout.setBackground(background);

        text = new Text(this);
        text.setTextSize(30);
        text.setTop(400);
        text.setLeft(200);
        text.setText(assembleText(FormAbility.getClickTimes()));
        positionLayout.addComponent(text);

        super.setUIContent(positionLayout);
    }
    ...
```

```
}
```

表单提供方在配置文件中声明，将 formEnabled 设置为 true，并提供表单尺寸信息。

```
{
    "module": {
        ...
        "abilities": [
            {
                ...
                "formEnabled": true,
                "form": {
                    "defaultHeight": 200,
                    "defaultWidth": 300
                },
                ...
            }
        ]
        ...
    }
    ...
}
```

表单使用方

表单使用方通常是桌面类应用。以下示例展示如何获取并展示表单。

1. 获取表单需要具有 ohos.permission.REQUIRE_FORM 权限，注意该权限仅限系统应用

获取。在配置文件中声明需要此权限。

```
"reqPermissions": [
    {
        "name": "ohos.permission.REQUIRE_FORM"
    }
]
```

调用 AbilitySlice 类的 acquireAbilityFormAsync()方法异步获取表单，该方法需要通过 Intent 指定获取的目标表单，并提供一个回调用于接收表单。注意，获取到 AbilityForm 实例并未立即显示到当前页面布局中，开发者需要继续后续步骤把表单添加到视图中才会显示。

```
    private void acquireForm() throws RemoteException {
        Intent intent = new Intent();
        Operation operation = new Intent.OperationBuilder()
                .withDeviceId("")
                .withBundleName("ohos.formsupplier.ability")
                .withAbilityName("ohos.formsupplier.ability.FormAbili
ty")
                .build();
        intent.setOperation(operation);

        // 获取 Form 的尺寸
        formAbilityInfo =
getBundleManager().queryAbilityByIntent(intent,IBundleManager.GET_ABI
LITY_INFO_WITH_PERMISSION,userId).get(0); // userId 的获取方式可以参见
AccountAbility.getOsAccountLocalIdFromProcess 等接口
```

```
        this.acquireAbilityFormAsync(intent, new
AbilityForm.OnAcquiredCallback() {
            @Override
            public void onAcquired(AbilityForm abilityForm) {
                form = abilityForm;
            }

            @Override
            public void onDestroyed(AbilityForm abilityForm) {
                form = null;
            }
        });
    }
```

创建布局，作为表单的视图容器。

```
    private PositionLayout createFormHostLayout() {
        PositionLayout formLayout = ...;

        ...
        // 将已获取的 AbilityInfo 的默认尺寸设置为 Form 尺寸
        formLayout.setHeight(formAbilityInfo.getDefaultFormHeight());
        formLayout.setWidth(formAbilityInfo.getDefaultFormWidth());
        ...

        return formLayout;
    }
```

将表单的视图容器添加到当前视图中，以便显示表单。

```
    private void showForm() {
        PositionLayout hostLayout = createFormHostLayout();

        // 将 Form 视图添加到视图容器
        hostLayout.addComponent(form.getView());

        // 将包含 Form 的视图容器添加到当前视图
        rootLayout.addComponent(hostLayout);
    }
```

如果开发者需要移除表单，调用 AbilitySlice 类的 releaseAbilityForm()方法，并以此前获取的 AbilityForm 对象作为参数。

分布式任务调度

概述

在 HarmonyOS 中，分布式任务调度平台对搭载 HarmonyOS 的多设备构筑的“超级虚拟终端”提供统一的组件管理能力，为应用定义统一的能力基线、接口形式、数据结构、服务描述语言，屏蔽硬件差异；支持远程启动、远程调用、业务无缝迁移等分布式任务。

分布式任务调度平台在底层实现 Ability（分布式任务调度的基本组件）跨设备的启动/关闭、连接及断开连接以及迁移等能力，实现跨设备的组件管理：

- 启动和关闭：向开发者提供管理远程 Ability 的能力，即支持启动 Page 模板的 Ability，以及启动、关闭 Service 和 Data 模板的 Ability。
- 连接和断开连接：向开发者提供跨设备控制服务（Service 和 Data 模板的 Ability）的能力，开发者可以通过与远程服务连接及断开连接实现获取或注销跨设备管理服务的对象，达到和本地一致的服务调度。
- 迁移能力：向开发者提供跨设备业务的无缝迁移能力，开发者可以通过调用 Page 模板 Ability 的迁移接口，将本地业务无缝迁移到指定设备中，打通设备间壁垒。

约束与限制

- 开发者需要在 Intent 中设置支持分布式的标记（例如：Intent.FLAG_ABILITYSLICE_MULTI_DEVICE 表示该应用支持分布式调度），否则将无法获得分布式能力。
- 开发者通过在 config.json 中添加分布式数据传输的权限申请：{"name": "ohos.permission.servicebus.ACCESS_SERVICE"}，获取跨设备连接的能力。
- PA（Particle Ability，Service 和 Data 模板的 Ability）的调用支持连接及断开连接、启动及关闭这四类行为，在进行调度时：
- 开发者必须在 Intent 中指定 PA 对应的 bundleName 和 abilityName。
- 当开发者需要跨设备启动、关闭或连接 PA 时，需要在 Intent 中指定对端设备的 deviceId。开发者可通过如设备管理类 DeviceManager 提供的 getDeviceList 获取指定条件下匿名化处理的设备列表，实现对指定设备 PA 的启动/关闭以及连接管理。
- FA（Feature Ability，Page 模板的 Ability）的调用支持启动和迁移行为，在进行调度时：
- 当启动 FA 时，需要开发者在 Intent 中指定对端设备的 deviceId、bundleName 和 abilityName。
- FA 的迁移实现相同 bundleName 和 abilityName 的 FA 跨设备迁移，因此需要指定迁移设备的 deviceId。

开发指导

场景介绍

开发者在应用中集成分布式调度能力，通过调用指定能力的分布式接口，实现跨设备能力调度。根据 Ability 模板及意图的不同，分布式任务调度向开发者提供以下六种能力：启动远程 FA、启动远程 PA、关闭远程 PA、连接远程 PA、断开连接远程 PA 和 FA 跨设备迁移。下面以设备 A（本地设备）和设备 B（远端设备）为例，进行场景介绍：

1. 设备 A 启动设备 B 的 FA：在设备 A 上通过本地应用提供的启动按钮，启动设备 B 上对应的 FA。例如：设备 A 控制设备 B 打开相册，只需开发者在启动 FA 时指定打开相册的意图即可。
2. 设备 A 启动设备 B 的 PA：在设备 A 上通过本地应用提供的启动按钮，启动设备 B 上指定的 PA。例如：开发者在启动远程服务时通过意图指定音乐播放服务，即可实现设备 A 启动设备 B 音乐播放的能力。
3. 设备 A 关闭设备 B 的 PA：在设备 A 上通过本地应用提供的关闭按钮，关闭设备 B 上指定的 PA。类似启动的过程，开发者在关闭远程服务时通过意图指定音乐播放服务，即可实现关闭设备 B 上该服务的能力。
4. 设备 A 连接设备 B 的 PA：在设备 A 上通过本地应用提供的连接按钮，连接设备 B 上指定的 PA。连接后，通过其他功能相关按钮实现控制对端 PA 的能力。通过连接关系，开发者可以实现跨设备的同步服务调度，实现如大型计算任务互助等价值场景。
5. 设备 A 与设备 B 的 PA 断开连接：在设备 A 上通过本地应用提供断开连接的按钮，将之前已连接的 PA 断开连接。
6. 设备 A 的 FA 迁移至设备 B：设备 A 上通过本地应用提供的迁移按钮，将设备 A 的业务无缝迁移到设备 B 中。通过业务迁移能力，打通设备 A 和设备 B 间的壁垒，实现如文档跨设备编辑、视频从客厅到房间跨设备接续播放等场景。

接口说明

分布式调度平台提供的连接和断开连接 PA、启动远程 FA、启动和关闭 PA 以及迁移 FA 的能力，是实现更多价值性场景的基础。

连接远程 PA

connectAbility(Intent intent, IAbilityConnection conn)接口提供连接指定设备上 PA 的能力，Intent 中指定待连接 PA 的设备 deviceId、bundleName 和 abilityName。当连接成功后，通过在 conn 定义的 onAbilityConnectDone 回调

中获取对端 PA 的服务代理，两者的连接关系则由 conn 维护。具体的参数定义如下表所示：

参数名	类型	说明
intent	ohos.aafwk.content.Intent	开发者需在 intent 对应的 Operation 中指定待连接 PA 的设备 deviceId、bundleName 和 abilityName。
conn	ohos.aafwk.ability.IAbilityConnection	当连接成功或失败时，作为连接关系的回调接口。该接口提供连接完成和断开连接完成时的处理逻辑，开发者可根据具体的场景进行定义。

启动远程 FA/PA

startAbility(Intent intent)接口提供启动指定设备上 FA 和 PA 的能力，Intent 中指定待启动 FA/PA 的设备 deviceId、bundleName 和 abilityName。具体参数定义如下表所示：

参数名	类型	说明
intent	ohos.aafwk.content.Intent	当开发者需要调用该接口启动远程 PA 时，需要指定待启动 PA 的设备 deviceId、bundleName 和 abilityName。若不指定设备 deviceId，则无法跨设备调用 PA。 类似地，在启动 FA 时，也需要开发者指定启动 FA 的设备 deviceId、bundleName 和 abilityName。

分布式调度平台还会提供与上述功能相对应的断开远程 PA 的连接和关闭远程 PA 的接口，相关的参数与连接、启动的接口类似。

- 断开远程 PA 连接：disconnectAbility(IAbilityConnection conn)。
- 关闭远程 PA：boolean stopAbility(Intent intent)。

迁移 FA

continueAbility(String deviceId)接口提供将本地 FA 迁移到指定设备上的能力，需要开发者在调用时指定目标设备的 deviceId。具体参数定义如下表所示：

说明

Ability 和 AbilitySlice 类均需要实现 IAbilityContinuation 及其方法，才可以实现 FA 迁移。

参数名	类型	说明
deviceId	String	当开发者需要调用该接口将本地 FA 迁移时，需要指定目标设备的 deviceId。

开发步骤

1. 导入功能依赖的包。

```
// 以下依赖包含分布式调度平台开放的接口，用于：连接/断开连接远程 PA、启动远程 FA、通过连接关系注册的回调函数 onAbilityConnectDone 中返回的对端 PA 的代理，实现对 PA 的控制
import ohos.aafwk.ability.AbilitySlice;
import ohos.aafwk.ability.IAbilityConnection;
import ohos.aafwk.content.Intent;
import ohos.aafwk.content.Operation;
import ohos.bundle.ElementName;
// 为了实现迁移能力，需要引入传递迁移所需数据的包以及实现迁移能力的接口。
import ohos.aafwk.ability.IAbilityContinuation;
import ohos.aafwk.content.IntentParams;
// 为了实现跨设备指令及数据通信，需要集成 HarmonyOS 提供的 RPC 接口
import ohos.rpc.IRemoteObject;
import ohos.rpc.IRemoteBroker;
```

```
import ohos.rpc.MessageParcel;
import ohos.rpc.MessageOption;
import ohos.rpc.RemoteException;
import ohos.rpc.RemoteObject;
//（可选）多设备场景下涉及设备选择，为此需要引入组网设备发现的能力
import ohos.distributedschedule.interwork.DeviceInfo;
import ohos.distributedschedule.interwork.DeviceManager;
//（可选）设计界面相关的包函数，对 FA 界面及按钮进行绘制
import ohos.agp.components.Button;
import ohos.agp.components.Component;
import ohos.agp.components.Component.ClickedListener;
import ohos.agp.components.ComponentContainer.LayoutConfig;
import ohos.agp.components.element.ShapeElement;
import ohos.agp.components.PositionLayout;
```

（可选）编写一个基本的 FA 用于使用分布式能力。

```
// 调用 AbilitySlice 模板实现一个用于控制基础功能的 FA
// Ability 和 AbilitySlice 类均需要实现 IAbilityContinuation 及其方法，才可以实现 FA 迁移。AbilitySlice 的代码示例如下
public class SampleSlice extends AbilitySlice implements IAbilityContinuation {
    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
        // 开发者可以自行进行界面设计
        // 为按钮设置统一的背景色
        // 例如通过 PositionLayout 指定大小可以实现简单界面
        PositionLayout layout = new PositionLayout(this);
```

```
        LayoutConfig config = new LayoutConfig(LayoutConfig.MATCH_PARENT,
LayoutConfig.MATCH_PARENT);
        layout.setLayoutConfig(config);
        ShapeElement buttonBg = new ShapeElement();
        buttonBg.setRgbColor(new RgbColor(0,125,255));
        addComponents(layout, buttonBg, config);
        super.setUIContent(layout);
    }

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

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

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

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

```
    @Override
    public void onStop() {
        super.onStop();
    }
}
```

说明

此步骤展示了一个简单 FA 的实现过程，实际开发中请开发者根据需要进行设计。

(可选）为不同的能力设置相应的控制按钮。

```
// 建议开发者按照自己的界面进行按钮设计
// 开发者可以自行实现如 createButton 的方法，新建一个显示文字 text，背景色为 buttonBg 以及大小尺寸位置符合 config 设置的按钮，用来与用户发生交互
// private Button createButton(String text, ShapeElement buttonBg, LayoutConfig config)
// 按照顺序在 PositionLayout 中依次添加按钮的示例
private void addComponents(PositionLayout linear, ShapeElement buttonBg, LayoutConfig config) {
    // 构建远程启动 FA 的按钮
    btnStartRemoteFA = createButton("StartRemoteFA", buttonBg, config);
    btnStartRemoteFA.setClickedListener(mStartRemoteFAListener);
    linear.addComponent(btnStartRemoteFA);
    // 构建远程启动 PA 的按钮
    btnStartRemotePA = createButton("StartRemotePA", buttonBg, config);
    btnStartRemotePA.setClickedListener(mStartRemotePAListener);
```

```
    linear.addComponent(btnStartRemotePA);
    // 构建远程关闭 PA 的按钮
    btnStopRemotePA = createButton("StopRemotePA", buttonBg, config);
    btnStopRemotePA.setClickedListener(mStopRemotePAListener);
    linear.addComponent(btnStopRemotePA);
    // 构建连接远程 PA 的按钮
    btnConnectRemotePA = createButton("ConnectRemotePA", buttonBg,
config);
    btnConnectRemotePA.setClickedListener(mConnectRemotePAListener);
    linear.addComponent(btnConnectRemotePA);
    // 构建控制连接 PA 的按钮
    btnControlRemotePA = createButton("ControlRemotePA", buttonBg,
config);
    btnControlRemotePA.setClickedListener(mControlPAListener);
    linear.addComponent(btnControlRemotePA);
    // 构建与远程 PA 断开连接的按钮
    btnDisconnectRemotePA = createButton("DisconnectRemotePA",
buttonBg, config);

btnDisconnectRemotePA.setClickedListener(mDisconnectRemotePAListener)
;
    linear.addComponent(btnDisconnectRemotePA);
    // 构建迁移 FA 的按钮
    btnContinueRemoteFA = createButton("ContinueRemoteFA", buttonBg,
config);
    btnContinueRemoteFA.setClickedListener(mContinueAbilityListener);
    linear.addComponent(btnContinueRemoteFA);
}
```

说明

此处只展示了基于按钮控制的能力调度方法，实际开发中请开发者根据需要选择能力调度方式。代码示例中未体现按钮如位置、样式等具体的设置方法，详请参考 JAVA UI 框架。

1. 通过设备管理 DeviceManager 提供的 getDeviceList 接口获取设备列表，用于指定目标设备。

```
// ISelectResult 是一个自定义接口，用来处理指定设备 deviceId 后执行的行为
 interface ISelectResult {
     void onSelectResult(String deviceId);
 }

// 获得设备列表，开发者可在得到的在线设备列表中选择目标设备执行操作
private void scheduleRemoteAbility(ISelectResult listener) {
    // 调用 DeviceManager 的 getDeviceList 接口，通过 FLAG_GET_ONLINE_DEVICE 标记获得在线设备列表
    List<DeviceInfo> onlineDevices = DeviceManager.getDeviceList(DeviceInfo.FLAG_GET_ONLINE_DEVICE);
    // 判断组网设备是否为空
    if (onlineDevices.isEmpty()) {
        listener.onSelectResult(null);
        return;
    }
    int numDevices = onlineDevices.size();
    ArrayList<String> deviceIds = new ArrayList<>(numDevices);
    ArrayList<String> deviceNames = new ArrayList<>(numDevices);
    onlineDevices.forEach((device) -> {
```

```
            deviceIds.add(device.getDeviceId());
            deviceNames.add(device.getDeviceName());
        });
        // 以选择首个设备作为目标设备为例
        // 开发者也可按照具体场景，通过别的方式进行设备选择
        String selectDeviceId = deviceIds.get(0);
        listener.onSelectResult(selectDeviceId);
    }
```

上述实例中涉及对在线组网设备的查询，该项能力需要开发者在对应的 config.json 中声明获取设备列表及设备信息的权限，如下所示：

```
{
    "reqPermissions": [
        {
            "name": "ohos.permission.DISTRIBUTED_DEVICE_STATE_CHANGE"
        },
        {
            "name": "ohos.permission.GET_DISTRIBUTED_DEVICE_INFO"
        },
        {
            "name": "ohos.permission.GET_BUNDLE_INFO"
        }
    ]
}
```

为启动远程 FA 的按钮设置点击回调，实现启动远程 FA 的能力。

```
// 启动一个指定 bundleName 和 abilityName 的 FA
private ClickedListener mStartRemoteFAListener = new ClickedListener() {
    @Override
    public void onClick(Component arg0) {
        // 启动远程 PA
        scheduleRemoteAbility(new ISelectResult() {
            @Override
            void onSelectResult(String deviceId) {
                if (deviceId != null) {
                    Intent intent = new Intent();
                    // 通过 scheduleRemoteAbility 指定目标设备 deviceId
                    // 指定待启动 FA 的 bundleName 和 abilityName
                    // 例如：bundleName = "com.huawei.helloworld"
                    //       abilityName =
"com.huawei.helloworld.SampleFeatureAbility"
                    // 设置分布式标记，表明当前涉及分布式能力
                    Operation operation = new Intent.OperationBuilder()
                            .withDeviceId(deviceId)
                            .withBundleName(bundleName)
                            .withAbilityName(abilityName)
                            .withFlags(Intent.FLAG_ABILITYSLICE_MULTI_D
EVICE)
                            .build();
                    intent.setOperation(operation);
                    // 通过 AbilitySlice 包含的 startAbility 接口实现跨设备启
动 FA
                    startAbility(intent);
                }
            }
```

```
        });
    }
};
```

为启动和关闭 PA 定义回调，实现启动和关闭 PA 的能力。

对于 PA 的启动、关闭、连接等操作都需要开发者提供目标设备的 deviceId。开发者可以通过 DeviceManager 相关接口得到当前组网下的设备列表，并以弹窗的形式供用户选择，也可以按照实际需要实现其他个性化的处理方式。在点击事件回调函数中，需要开发者指定得到 deviceId 后的处理逻辑，即实现类似上例中 listener.onSelectResult(String deviceId)的方法，代码示例如下：

```
// 启动远程 PA
private ClickedListener mStartRemotePAListener = new ClickedListener()
{
    @Override
    public void onClick(Component arg0) {
        // 启动远程 PA
        scheduleRemoteAbility(new ISelectResult() {
            @Override
            void onSelectResult(String deviceId) {
                if (deviceId != null) {
                    Intent intentToStartPA = new Intent();
                    // bundleName 和 abilityName 与待启动 PA 对应
                    // 例如：bundleName = "com.huawei.helloworld"
                    //       abilityName =
"com.huawei.helloworld.SampleParticleAbility"
                    Operation operation = new Intent.OperationBuilder()
                            .withDeviceId(deviceId)
```

```
                            .withBundleName(bundleName)
                            .withAbilityName(abilityName)
                            .withFlags(Intent.FLAG_ABILITYSLICE_MULTI
_DEVICE)
                            .build();
                    intentToStartPA.setOperation(operation);
                    startAbility(intentToStartPA);
                }
            }
        });
    }
};

// 关闭远程 PA，和启动类似开发者需要指定待关闭 PA 对应的 bundleName 和
abilityName
private ClickedListener mStopRemotePAListener = new ClickedListener() {
    @Override
    public void onClick(Component arg0) {
        scheduleRemoteAbility(new ISelectResult() {
            @Override
            void onSelectResult(String deviceId) {
                if (deviceId != null) {
                    Intent intentToStopPA = new Intent();
                    // bundleName 和 abilityName 与待关闭 PA 对应
                    // 例如：bundleName = "com.huawei.helloworld"
                    //       abilityName =
"com.huawei.helloworld.SampleParticleAbility"
                    Operation operation = new Intent.OperationBuilder()
                            .withDeviceId(deviceId)
```

```
                                .withBundleName(bundleName)
                                .withAbilityName(abilityName)
                                .withFlags(Intent.FLAG_ABILITYSLICE_MULTI
_DEVICE)
                                .build();
                        intentToStopPA.setOperation(operation);
                        stopAbility(intentToStopPA);
                    }
                }
            });
    }
};
```

说明

启动和关闭的行为类似,开发者只需在 Intent 中指定待调度 PA 的 deviceId、bundleName 和 abilityName，并以 operation 的形式封装到 Intent 内。通过 AbilitySlice（Ability）包含的 startAbility()和 stopAbility()接口即可实现相应功能。

1. 设备 A 连接设备 B 侧的 PA，利用连接关系调用该 PA 执行特定任务，以及断开连接。

```
// 当连接完成时，用来提供管理已连接 PA 的能力
private MyRemoteProxy mProxy = null;
// 用于管理连接关系
private IAbilityConnection conn = new IAbilityConnection() {
    @Override
```

```
    public void onAbilityConnectDone(ElementName element, IRemoteObject
remote, int resultCode) {
        // 跨设备 PA 连接完成后，会返回一个序列化的 IRemoteObject 对象
        // 通过该对象得到控制远端服务的代理
        mProxy = new MyRemoteProxy(remote);
        btnConnectRemotePA.setText("connectRemoteAbility done");
     }

    @Override
    public void onAbilityDisconnectDone(ElementName element, int
resultCode) {
        // 当已连接的远端 PA 关闭时，会触发该回调
        // 支持开发者按照返回的错误信息进行 PA 生命周期管理
        disconnectAbility(conn);
    }
};
```

仅通过启动/关闭两种方式对 PA 进行调度无法应对需长期交互的场景，因此，分布式任务调度平台向开发者提供了跨设备 PA 连接及断开连接的能力。为了对已连接 PA 进行管理，开发者需要实现一个满足 IAbilityConnection 接口的连接状态检测实例，通过该实例可以对连接及断开连接完成时设置具体的处理逻辑，例如：获取控制对端 PA 的代理等。进一步为了使用该代理跨设备调度 PA，开发者需要在本地及对端分别实现对外接口一致的代理。一个具备加法能力的代理示例如下：

```
// 以连接提供加法计算能力的 PA 为例。为了提供跨设备连接能力，需要在本地发起连接侧和对端被连接侧分别实现代理。
```

```
// 发起连接侧的代理示例如下
public class MyRemoteProxy implements IRemoteBroker{
    private static final int ERR_OK = 0;
    private static final int COMMAND_PLUS =
IRemoteObject.MIN_TRANSACTION_ID;
    private final IRemoteObject remote;

    public MyRemoteProxy(
        /* [in] */ IRemoteObject remote) {
        this.remote = remote;
    }

    @Override
    public IRemoteObject asObject() {
        return remote;
    }

    public int plus(
        /* [in] */ int a,
        /* [in] */ int b) throws RemoteException {
        MessageParcel data = MessageParcel.obtain();
        MessageParcel reply = MessageParcel.obtain();
        // option 不同的取值，决定采用同步或异步方式跨设备控制 PA
        // 本例需要同步获取对端 PA 执行加法的结果，因此采用同步的方式，
即 MessageOption.TF_SYNC
        // 具体 MessageOption 的设置，可参考相关 API 文档
        MessageOption option = new
MessageOption(MessageOption.TF_SYNC);
        data.writeInt(a);
```

```
        data.writeInt(b);

        try {
            remote.sendRequest(COMMAND_PLUS, data, reply, option);
            int ec = reply.readInt();
            if (ec != ERR_OK) {
                throw new RemoteException();
            }
            int result = reply.readInt();
            return result;
        } catch (RemoteException e) {
            throw new RemoteException();
        } finally {
            data.reclaim();
            reply.reclaim();
        }
    }
}
```

此外，对端待连接的 PA 需要实现对应的客户端，代码示例如下所示：

```
// 以计算加法为例，对端实现的客户端如下
public class MyRemote extends RemoteObject implements IRemoteBroker{
    private static final int ERR_OK = 0;
    private static final int ERROR = -1;
    private static final int COMMAND_PLUS =
IRemoteObject.MIN_TRANSACTION_ID;

    public MyRemote() {
```

```
        super("MyService_Remote");
    }

    @Override
    public IRemoteObject asObject() {
        return this;
    }

    @Override
    public boolean onRemoteRequest(int code, MessageParcel data,
MessageParcel reply, MessageOption option) {
        if (code != COMMAND_PLUS) {
            reply.writeInt(ERROR);
            return false;
        }
        int value1 = data.readInt();
        int value2 = data.readInt();
        int sum = value1 + value2;
        reply.writeInt(ERR_OK);
        reply.writeInt(sum);
        return true;
    }
}
```

对端除了要实现如上所述的客户端外，待连接的 PA 还需要作如下修改：

```
// 为了返回给连接方可调用的代理，需要在该 PA 中实例化客户端，例如作为该 PA 的成员变量
private MyRemote remote = new MyRemote();
// 当该 PA 接收到连接请求时，即将该客户端转化为代理返回给连接发起侧
```

```
@Override
protected IRemoteObject onConnect(Intent intent) {
    super.onConnect(intent);
    return remote.asObject();
}
```

完成上述步骤后，可以通过点击事件实现连接、利用连接关系控制 PA 以及断开连接等行为，代码示例如下：

```
// 连接远程 PA
private ClickedListener mConnectRemotePAListener = new ClickedListener()
{
    @Override
    public void onClick(Component arg0) {
        scheduleRemoteAbility(new ISelectResult() {
            @Override
            void onSelectResult(String deviceId) {
                if (deviceId != null) {
                    Intent connectPAIntent = new Intent();
                    // bundleName 和 abilityName 与待连接的 PA 一一对应
                    // 例如：bundleName = "com.huawei.helloworld"
                    //       abilityName =
"com.huawei.helloworld.SampleParticleAbility"
                    Operation operation = new Intent.OperationBuilder()
                            .withDeviceId(deviceId)
                            .withBundleName(bundleName)
                            .withAbilityName(abilityName)
                            .withFlags(Intent.FLAG_ABILITYSLICE_MULTI_D
EVICE)
                            .build();
```

```
                    connectPAIntent.setOperation(operation);
                    connectAbility(connectPAIntent, conn);
                }
            }
        });
    }
};
// 控制已连接 PA 执行加法
private ClickedListener mControlPAListener = new ClickedListener() {
    @Override
    public void onClick(Component arg0) {
        if (mProxy != null) {
            int ret = -1;
            try {
                ret = mProxy.plus(10, 20);
            } catch (RemoteException e) {
                e.printStackTrace();
            }
            btnControlRemotePA.setText("ControlRemotePA result = " + ret);
        }
    }
};
// 与远程 PA 断开连接
private ClickedListener mDisconnectRemotePAListener = new
ClickedListener() {
    @Override
    public void onClick(Component arg0) {
        // 按钮复位
```

```
        btnConnectRemotePA.setText("ConnectRemotePA");
        btnControlRemotePA.setText("ControlRemotePA");
        disconnectAbility(conn);
    }
};
```

说明

通过连接/断开连接远程 PA，与跨设备 PA 建立长期的管理关系。例如在本例中，通过连接关系得到远程 PA 的控制代理后，实现跨设备计算加法并将结果返回到本地显示。在实际开发中，开发者可以根据需要实现多种分布式场景，例如：跨设备位置/电量等信息的采集、跨设备计算资源互助等。

设备 A 将运行时的 FA 迁移到设备 B，实现业务在设备间无缝迁移。

```
// 跨设备迁移 FA
// 本地 FA 设置当前运行任务
private ClickedListener mContinueAbilityListener = new ClickedListener()
{
    @Override
    public void onClick(Component arg0) {
        // 用户选择设备后实现业务迁移
        scheduleRemoteAbility(new ISelectResult() {
            @Override
            public void onSelectResult(String deviceId) {
                continueAbility(deviceId);
            }
        });
    }
};
```

此外，不同于启动行为，FA 的迁移还涉及到状态数据的传递。为此，继承的 IAbilityContinuation 接口为开发者提供迁移过程中特定事件的管理能力。通过自定义迁移事件相关的行为，最终实现对 Ability 的迁移。具体的定义可以参考相关的 API 文档，此处主要以较为常用的两个事件，包括迁移发起端完成迁移的回调 onCompleteContinuation(int result)以及接收到远端迁移行为传递数据的回调 onRestoreData(IntentParams restoreData)。其他还包括迁移到远端设备的 FA 关闭的回调 onRemoteTerminated()、用于本地迁移发起时保存状态数据的回调 onSaveData(IntentParams saveData)和本地发起迁移的回调 onStartContinuation()。按照实际应用自定义特定场景对应的回调，可以完成多种场景下 FA 的迁移任务。

```
@Override
public boolean onSaveData(IntentParams saveData) {
    String exampleData = String.valueOf(System.currentTimeMillis());
    saveData.setParam("continueParam", exampleData);
    return true;
}

@Override
public boolean onRestoreData(IntentParams restoreData) {
    // 远端 FA 迁移传来的状态数据，开发者可以按照特定的场景对这些数据进行处理
    Object data = restoreData.getParam("continueParam");
    return true;
}
```

```
@Override
public void onCompleteContinuation(int result) {
    btnContinueRemoteFA.setText("ContinueAbility Done");
}
```

说明

- FA 迁移可以打通设备间的壁垒，有助于不同能力的设备进行互助。前文以一个简单的例子介绍如何通过分布式任务调度提供的能力，实现 FA 跨设备的迁移（包括 FA 启动及状态数据的同步）。
- FA 迁移过程中，远端 FA 首先接收到发起端 FA 传输的数据，再执行启动，即 onRestoreData()发生在 onStart()之前，详见 API 参考。

公共事件与通知

概述

HarmonyOS 通过 CES（Common Event Service，公共事件服务）为应用程序提供订阅、发布、退订公共事件的能力，通过 ANS（Advanced Notification Service，即高级通知服务）系统服务来为应用程序提供发布通知的能力。

- 公共事件可分为系统公共事件和自定义公共事件。
- 系统公共事件：系统将收集到的事件信息，根据系统策略发送给订阅该事件的用户程序。 例如：用户可感知亮灭屏事件，系统关键服务发送的系统事件（例如：USB 插拔，网络连接，系统升级等）。
- 自定义公共事件：应用自定义一些公共事件用来处理业务逻辑。
- 通知提供应用的即时消息或通信消息，用户可以直接删除或点击通知触发进一步的操作。
- IntentAgent 封装了一个指定行为的 Intent，可以通过 IntentAgent 启动 Ability 和发送公共事件。

应用如果需要接收公共事件，需要订阅相应的事件。

约束与限制

公共事件的约束与限制

- 目前公共事件仅支持动态订阅。部分系统事件需要具有指定的权限，具体的权限见 API 参考。
- 目前公共事件订阅不支持多用户。
- ThreadMode 表示线程模型，目前仅支持 HANDLER 模式，即在当前 UI 线程上执行回调函数。
- deviceId 用来指定订阅本地公共事件还是远端公共事件。deviceId 为 null、空字符串或本地设备 deviceId 时，表示订阅本地公共事件，否则表示订阅远端公共事件。

通知的约束与限制

- 通知目前支持六种样式：普通文本、长文本、图片、社交、多行文本和媒体样式。创建通知时必须包含一种样式。
- 通知支持快捷回复。
- 目前通知订阅不支持多用户。
- 通知的订阅目前仅支持系统应用，不支持第三方应用。

IntentAgent 的限制

使用 IntentAgent 启动 Ability 时，Intent 必须指定 Ability 的包名和类名。

公共事件开发指导

场景介绍

每个应用都可以订阅自己感兴趣的公共事件，订阅成功后且公共事件发布后，系统会把其发送给应用。这些公共事件可能来自系统、其他应用和应用自身。HarmonyOS 提供了一套完整的 API，支持用户订阅、发送和接收公共事件。发送公共事件需要借助 CommonEventData 对象，接收公共事件需要继承 CommonEventSubscriber 类并实现 onReceiveEvent 回调函数。

接口说明

公共事件相关基础类包含 CommonEventData、CommonEventPublishInfo、CommonEventSubscribeInfo、CommonEventSubscriber 和 CommonEventManager。基础类之间的关系如下图所示：

图 1　公共事件基础类关系图

- CommonEventData
- CommonEventData 封装公共事件相关信息。用于在发布、分发和接收时处理数据。在构造 CommonEventData 对象时，相关参数需要注意以下事项：
-
- code 为有序公共事件的结果码，data 为有序公共事件的结果数据，仅用于有序公共事件场景。
- intent 不允许为空，否则发布公共事件失败。

表 1 CommonEventData 主要接口

接口名	描述
CommonEventData()	创建公共事件数据。
CommonEventData(Intent intent)	创建公共事件数据指定 Intent。
CommonEventData(Intent intent, int code, String data)	创建公共事件数据，指定 Intent、code 和 data。
getIntent()	获取公共事件 intent。
setCode(int code)	设置有序公共事件的结果码。
getCode()	获取有序公共事件的结果码。
setData(String data)	设置有序公共事件的详细结果数据。
getData()	获取有序公共事件的详细结果数据。

CommonEventPublishInfo

CommonEventPublishInfo 封装公共事件发布相关属性、限制等信息，包括公共事件类型（有序或粘性）、接收者权限等。

- 有序公共事件：主要场景是多个订阅者有依赖关系或者对处理顺序有要求，例如：高优先级订阅者可修改公共事件内容或处理结果，包括终止公共事件处理；或者低优先级订阅者依赖高优先级的处理结果等。

有序公共事件的订阅者可以通过 CommonEventSubscribeInfo.setPriority()方法指定优先级，缺省为 0，优先级范围[-1000, 1000]，值越大优先级越高。

- 粘性公共事件：指公共事件的订阅动作是在公共事件发布之后进行，订阅者也能收到的公共事件类型。主要场景是由公共事件服务记录某些系统状态，如蓝牙、WLAN、充电等事件和状态。不使用粘性公共事件机制时，应用可以通过直接访问系统服务获取该状态；在状态变化时，系统服务、硬件需要提供类似 observer 等方式通知应用。

发布粘性公共事件可以通过 setSticky()方法设置， 发布粘性公共事件需要申请如下权限。声明请参考表 1。

```
"reqPermissions": [{
    "name": "ohos.permission.COMMONEVENT_STICKY",
    "reason": "get right",
    "usedScene": {
       "ability": [
        ".MainAbility"
       ],
       "when": "inuse"
    }
}, {
...
}]
```

表 2 CommonEventPublishInfo 主要接口

接口名	描述
CommonEventPublishInfo()	创建公共事件发送信息。
CommonEventPublishInfo(CommonEventPublishInfo publishInfo)	拷贝一个公共事件发送信息。
setSticky(boolean sticky）	设置公共事件的粘性属性。
setOrdered(boolean ordered)	设置公共事件的有

表 2 CommonEventPublishInfo 主要接口

接口名	描述
	序属性。
setSubscriberPermissions(String[] subscriberPermissions)	设置公共事件订阅者的权限，多参数仅第一个生效。

- **CommonEventSubscribeInfo**

CommonEventSubscribeInfo 封装公共事件订阅相关信息，比如优先级、线程模式、事件范围等。

线程模式（ThreadMode）：设置订阅者的回调方法执行的线程模式。

ThreadMode 有 HANDLER，POST，ASYNC，BACKGROUND 四种模式，目前只支持 HANDLER 模式。

- HANDLER：在 Ability 的主线程上执行。
- POST：在事件分发线程执行。
- ASYNC：在一个新创建的异步线程执行。
- BACKGROUND：在后台线程执行。

表 3 CommonEventSubscribeInfo 主要接口

接口名	描述
CommonEventSubscribeInfo(MatchingSkills matchingSkills)	创建公共事件订阅器指定 matchingSkills。

表 3 CommonEventSubscribeInfo 主要接口

接口名	描述
CommonEventSubscribeInfo(CommonEventSubscribeInfo)	拷贝公共事件订阅器对象。
setPriority(int priority)	设置优先级，用于有序公共事件。
setThreadMode(ThreadMode threadMode)	指定订阅者的回调函数运行在哪个线程上。
setPermission(String permission)	设置订阅者的权限。
setDeviceId(String deviceId)	指定订阅哪台设备的公共事件。

- **CommonEventSubscriber**

CommonEventSubscriber 封装公共事件订阅者及相关参数。

- CommonEventSubscriber.AsyncCommonEventResult 类处理有序公共事件异步执行，详见 API 参考。
- 目前只能通过调用 CommonEventManager 的 subscribeCommonEvent() 进行订阅。

表 4 CommonEventSubscriber 主要接口

接口名	描述

表 4 CommonEventSubscriber 主要接口

接口名	描述
CommonEventSubscriber(CommonEventSubscribeInfo subscribeInfo)	构造公共事件订阅者实例。
onReceiveEvent(CommonEventData data)	由开发者实现，在接收到公共事件时被调用。
AsyncCommonEventResult goAsyncCommonEvent()	设置有序公共事件异步执行。
setCodeAndData(int code, String data)	设置有序公共事件的异步结果。
setData(String data)	设置有序公共事件的异步结果数据。
setCode(int code)	设置有序公共事件的异步结果码。
getData()	获取有序公共事件的异步结果数据。
getCode()	获取有序公共事件的异步结果码。
abortComonEvent()	取消当前的公共事件，仅对有序公共事件有效，取消后，公共事件不再向下一个订阅者传递。
getAbortCommonEvent()	获取当前有序公共事件是否取消的状态。

表 4 CommonEventSubscriber 主要接口

接口名	描述
clearAbortCommonEvent()	清除当前有序公共事件 abort 状态。
isOrderedCommonEvent()	查询当前公共事件的是否为有序公共事件。
isStickyCommonEvent()	查询当前公共事件是为否粘性公共事件。

- **CommonEventManager**

CommonEventManager 是为应用提供订阅、退订和发布公共事件的静态接口类。

表 5 CommonEventManager 主要接口

方法	描述
publishCommonEvent(CommonEventData event)	发布公共事件。
publishCommonEvent(CommonEventData event, CommonEventPublishInfo publishinfo)	发布公共事件指定发布信息。
publishCommonEvent(CommonEventData event, CommonEventPublishInfo publishinfo, CommonEventSubscriber resultSubscriber)	发布有序公共事件，指定发布信息和最后一个

表 5 CommonEventManager 主要接口

方法	描述
	接收者。
subscribeCommonEvent(CommonEventSubscriber subscriber)	订阅公共事件。
unsubscribeCommonEvent(CommonEventSubscriber subscriber)	退订公共事件。

发布公共事件

开发者可以发布四种公共事件：无序的公共事件、带权限的公共事件、有序的公共事件、粘性的公共事件。

发布无序的公共事件：构造 CommonEventData 对象，设置 Intent，通过构造 operation 对象把需要发布的公共事件信息传入 intent 对象。然后调用 CommonEventManager.publishCommonEvent(CommonEventData) 接口发布公共事件。

```
try {
    Intent intent = new Intent();
    Operation operation = new Intent.OperationBuilder()
            .withAction("com.my.test")
            .build();
    intent.setOperation(operation);
```

```
        CommonEventData eventData = new CommonEventData(intent);
        CommonEventManager.publishCommonEvent(eventData);
    } catch (RemoteException e) {
        HiLog.info(LABEL, "publishCommonEvent occur exception.");
    }
```

发布携带权限的公共事件：构造 CommonEventPublishInfo 对象，设置订阅者的权限。

- 订阅者在 config.json 中申请所需的权限，各字段含义详见权限定义字段说明。

```
{
    "reqPermissions": [{
        "name": "com.example.MyApplication.permission",
        "reason": "get right",
        "usedScene": {
           "ability": [
            ".MainAbility"
           ],
           "when": "inuse"
        }
    }, {
    ...
    }]
}
```

发布带权限的公共事件示例代码如下：

```
Intent intent = new Intent();
Operation operation = new Intent.OperationBuilder()
        .withAction("com.my.test")
        .build();
intent.setOperation(operation);
CommonEventData eventData = new CommonEventData(intent);
CommonEventPublishInfo publishInfo = new CommonEventPublishInfo();
String[] permissions = {"com.example.MyApplication.permission" };
publishInfo.setSubscriberPermissions(permissions); // 设置权限
try {
     CommonEventManager.publishCommonEvent(eventData, publishInfo);
} catch (RemoteException e) {
     HiLog.info(LABEL, "publishCommoneEvent occur exception.");
}
```

发布有序的公共事件：构造 CommonEventPublishInfo 对象，通过 setOrdered(true)指定公共事件属性为有序公共事件，也可以指定一个最后的公共事件接收者。

```
CommonEventSubscriber resultSubscriber = new MyCommonEventSubscriber();
CommonEventPublishInfo publishInfo = new CommonEventPublishInfo();
publishInfo.setOrdered(true); // 设置属性为有序公共事件
try {
    CommonEventManager.publishCommonEvent(eventData, publishInfo,
resultSubscriber); // 指定 resultSubscriber 为有序公共事件最后一个接收者。
} catch (RemoteException e) {
    HiLog.info(LABEL, "publishCommoneEvent occur exception.");
}
```

发布粘性公共事件：构造CommonEventPublishInfo对象，通过setSticky(true)指定公共事件属性为粘性公共事件。

1. 发布者首先在 config.json 中申请发布粘性公共事件所需的权限，各字段含义详见权限申请字段说明。

```
{
    "reqPermissions": [{
        "name": "ohos.permission.COMMONEVENT_STICKY",
        "reason": "get right",
        "usedScene": {
           "ability": [
            ".MainAbility"
           ],
           "when": "inuse"
        }
    }, {
    ...
    }]
}
```

发布粘性公共事件。

```
CommonEventPublishInfo publishInfo = new CommonEventPublishInfo();
publishInfo.setSticky(true); // 设置属性为粘性公共事件
try {
    CommonEventManager.publishCommonEvent(eventData, publishInfo);
} catch (RemoteException e) {
```

```
    HiLog.info(LABEL, "publishCommoneEvent occur exception.");
}
```

订阅公共事件

1. 创建 CommonEventSubscriber 派生类，在 onReceiveEvent()回调函数中处理公共事件。

说明

此处不能执行耗时操作，否则会阻塞 UI 线程，产生用户点击没有反应等异常。

```
class MyCommonEventSubscriber extends CommonEventSubscriber {
    MyCommonEventSubscriber(CommonEventSubscribeInfo info) {
        super(info);
    }
    @Override
    public void onReceiveEvent(CommonEventData commonEventData) {
    }
}
```

构造 MyCommonEventSubscriber 对象，调用 CommonEventManager.subscribeCommonEvent()接口进行订阅。

```
String event = "com.my.test";
MatchingSkills matchingSkills = new MatchingSkills();
filter.addEvent(event); // 自定义事件
filter.addEvent(CommonEventSupport.COMMON_EVENT_SCREEN_ON); // 亮屏事件
CommonEventSubscribeInfo subscribeInfo = new
CommonEventSubscribeInfo(matchingSkills);
```

```
MyCommonEventSubscriber subscriber = new
MyCommonEventSubscriber(subscribeInfo);
try {
    CommonEventManager.subscribeCommonEvent(subscriber);
} catch (RemoteException e) {
    HiLog.info(LABEL, "subscribeCommonEvent occur exception.");
}
```

如果订阅拥有指定权限应用发布的公共事件，发布者需要在 config.json 中申请权限，各字段含义详见 权限申请字段说明。

```
"reqPermissions": [
    {
        "name": "ohos.abilitydemo.permission.PROVIDER",
        "reason": "get right",
        "usedScene": {
            "ability": ["com.huawei.hmi.ivi.systemsetting.MainAbility"],
            "when": "inuse"
        }
    }
]
```

如果订阅的公共事件是有序的，可以调用 setPriority() 指定优先级。

```
String event = "com.my.test";
MatchingSkills matchingSkills = new MatchingSkills();
matchingSkills.addEvent(event ); // 自定义事件

CommonEventSubscribeInfo subscribeInfo = new
CommonEventSubscribeInfo(matchingSkills);
```

```
subscribeInfo.setPriority(100); // 设置优先级，优先级取值范围[-1000，1000]，值默认为 0。

MyCommonEventSubscriber subscriber = new MyCommonEventSubscriber(subscribeInfo);

try {

     CommonEventManager.subscribeCommonEvent(subscriber);

} catch (RemoteException e) {

     HiLog.info(LABEL, "subscribeCommonEvent occur exception.");

}
```

针对在 onReceiveEvent 中不能执行耗时操作的限制，可以使用 CommonEventSubscriber 的 goAsyncCommonEvent()来实现异步操作，函数返回后仍保持该公共事件活跃，且执行完成后必须调用 AsyncCommonEventResult .finishCommonEvent()来结束。

```
EventRunner runner = EventRunner.create(); //EventRunner 创建新线程，将耗时的操作放到新的线程上执行

MyEventHandler myHandler = new MyEventHandler(runner); //MyEventHandler 为 EventHandler 的派生类，在不同线程间分发和处理事件和 Runnable 任务

@Override

public void onReceiveEvent(CommonEventData commonEventData){

    final AsyncCommonEventResult result = goAsyncCommonEvent();


    Runnable task = new Runnable() {

        @Override

        public void run() {

            ........          // 待执行的操作，由开发者定义

            result.finishCommonEvent(); // 调用 finish 结束异步操作

        }
```

```
    };
    myHandler.postTask(task);
}
```

退订公共事件

在 Ability 的 onStop()中调用 CommonEventManager.unsubscribeCommonEvent()方法来退订公共事件。调用后，之前订阅的所有公共事件均被退订。

```
try {
    CommonEventManager.unsubscribeCommonEvent(subscriber);
 } catch (RemoteException e) {
    HiLog.info(LABEL, "unsubscribeCommonEvent occur exception.");
 }
```

通知开发指导

场景介绍

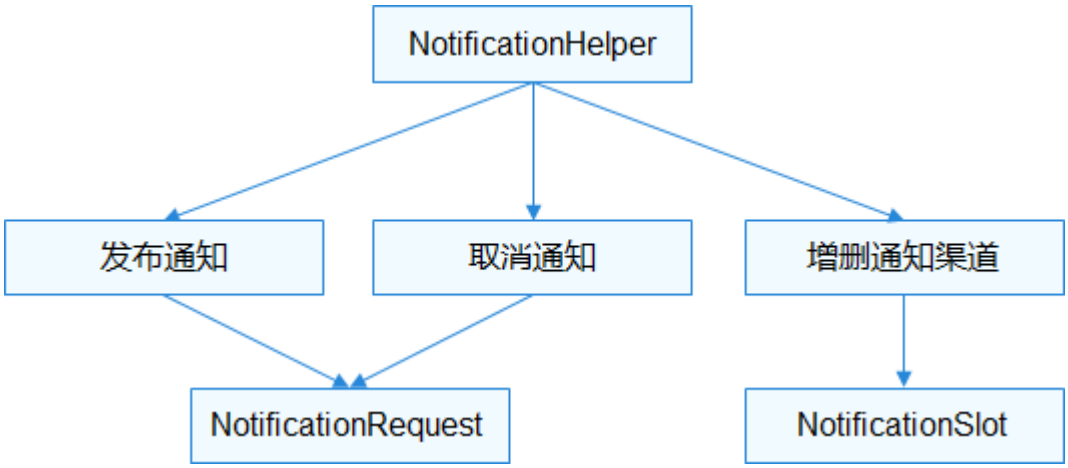
HarmonyOS 提供了通知功能，即在一个应用的 UI 界面之外显示的消息，主要用来提醒用户有来自该应用中的信息。当应用向系统发出通知时，它将先以图标的形式显示在通知栏中，用户可以下拉通知栏查看通知的详细信息。常见的使用场景：

- 显示接收到短消息、即时消息等。
- 显示应用的推送消息，如广告、版本更新等。
- 显示当前正在进行的事件，如播放音乐、导航、下载等。

接口说明

通知相关基础类包含 NotificationSlot、NotificationRequest 和 NotificationHelper。基础类之间的关系如下所示：

图 1　通知基础类关系图



- **NotificationSlot**

NotificationSlot 可以对提示音、振动、锁屏显示和重要级别等进行设置。一个应用可以创建一个或多个 NotificationSlot，在发送通知时，通过绑定不同的 NotificationSlot，实现不同用途。

说明

NotificationSlot 需要先通过 NotificationHelper 的 addNotificationSlot(NotificationSlot)方法发布后，通知才能绑定使用；所有绑定该 NotificationSlot 的通知在发布后都具备相应的特性，对象在创建后，将无法更改这些设置，对于是否启动相应设置，用户有最终控制权。

不指定 NotificationSlot 时，当前通知会使用默认的 NotificationSlot，默认的 NotificationSlot 优先级为 LEVEL_DEFAULT。

表 1 NotificationSlot 主要接口

接口名	描述
NotificationSlot(String id, String name, int level)	构造 NotificationSlot。
setLevel(int level)	设置 NotificationSlot 的级别。
setName(String name)	设置 NotificationSlot 的命名。
setDescription(String description)	设置 NotificationSlot 的描述信息。
enableBypassDnd(boolean bypassDnd)	设置是否绕过系统的免打扰模式。
setEnableVibration(boolean vibration)	设置收到通知时是否使能振动。
setLockscreenVisibleness(int visibleness)	设置在锁屏场景下，收到通知后是否显示，以及显示的效果。
setEnableLight(boolean isLightEnabled)	设置收到通知时是否开启呼吸灯，前提是当前硬件支持呼吸灯。
setLedLightColor(int color)	设置收到通知时的呼吸灯颜色。

表 1 NotificationSlot 主要接口

接口名	描述
setSlotGroup(String groupId)	绑定当前 NotificationSlot 到一个 NotificationSlot 组。

NotificationSlot 的级别目前支持如下几种， 由低到高：

- LEVEL_NONE： 表示通知不发布。
- LEVEL_MIN：表示通知可以发布，但是不显示在通知栏，不自动弹出，无提示音；该级别不适用于前台服务的场景。
- LEVEL_LOW：表示通知可以发布且显示在通知栏，不自动弹出，无提示音。
- LEVEL_DEFAULT：表示通知发布后可在通知栏显示，不自动弹出，触发提示音。
- LEVEL_HIGH：表示通知发布后可在通知栏显示，自动弹出，触发提示。
- **NotificationRequest**

NotificationRequest 用于设置具体的通知对象，包括设置通知的属性，如：通知的分发时间、小图标、大图标、自动删除等参数，以及设置具体的通知类型，如普通文本、长文本等。

表 2 NotificationRequest 主要接口

接口名	描述
NotificationRequest()	构建一个通知。
NotificationRequest(int notificationId)	构建一个通知，指定通知的 id。通知的 Id 在应用内容具有唯一性，如果不指定，默认为 0。
setNotificationId(int notificationId)	设置当前通知 id。

表 2 NotificationRequest 主要接口

接口名	描述
setAutoDeletedTime(long time)	设置通知自动取消的时间戳。
setContent (NotificationRequest.NotificationContent content)	设置通知的具体类型。
setCreateTime(long createTime)	设置通知的创建的时间戳。
setDeliveryTime(long deliveryTime)	设置通知分发的时间戳。
setSlotId(String slotId)	设置通知的 NotificationSlot id。
setTapDismissed(boolean tapDismissed)	设置通知在用户点击后是否自动取消。
setLittleIcon(PixelMap smallIcon)	设置通知的小图标，在通知左上角显示。
setBigIcon(PixelMap bigIcon)	设置通知的大图标，在通知的右边显示。
setGroupValue(String groupValue)	设置分组通知，相同分组的通知在通知栏显示时，将会折叠在一组应用中显示。
addActionButton(NotificationActionButton actionButton)	设置通知添加通知 ActionButton。
setIntentAgent(IntentAgent agent)	设置通知承载指定的 IntentAgent，在通知中实现即将触发的事件。

具体的通知类型：目前支持六种类型，包括普通文本 NotificationNormalContent、长文本 NotificationLongTextContent、图片 NotificationPictureContent、多行 NotificationMultiLineContent、社交 NotificationConversationalContent、媒体 NotificationMediaContent。

表 3 通知类型的主要接口

类名	接口名	描述
NotificationNormalContent	setTitle(String title)	设置通知标题。
NotificationNormalContent	setText(String text)	设置通知内容。
NotificationNormalContent	setAdditionalText(String additionalText)	设置通知次要内容，是对通知内容的补充。
NotificationPictureContent	setBriefText(String briefText)	设置通知概要内容，是对通知内容的总结。
NotificationPictureContent	setExpandedTitle(String expandedTitle)	设置通知一旦设置了，折叠时显示 setTitle(String)的值，展开时当前设置的展开标题。

表 3 通知类型的主要接口

类名	接口名	描述
NotificationPictureContent	setBigPicture(PixelMap bigPicture)	设置通知的图片内容，附加在 setText(String text)下方。
NotificationLongTextContent	setLongText(String longText)	设置通知的长文本。
NotificationConversationalContent	setConversationTitle(String conversationTitle)	设置社交通知的标题。
NotificationConversationalContent	addConversationalMessage(ConversationalMessage message)	通知添加一条消息。
NotificationMultiLineContent	addSingleLine(String line)	在当前通知中添加一行文本。
NotificationMediaContent	setAVToken(AVToken avToken)	将媒体通知绑定指定的 AVToken。
NotificationMediaContent	setShownActions(int[] actions)	设置媒体通知待展示的按钮。

说明

通知发布后，通知的设置不可修改。如果下次发布通知使用相同的 id，就会更新之前发布的通知。

- **NotificationHelper**

NotificationHelper 封装了发布、更新、订阅、删除通知等静态方法。订阅通知、退订通知和查询系统中所有处于活跃状态的通知，有权限要求需为系统应用或具有订阅者权限。

表 4 NotificationHelper 主要接口

接口名	描述
publishNotification(NotificationRequest request)	发布一条通知。
publishNotification(String tag, NotificationRequest)	发布一条带 TAG 的通知。
cancelNotification(int notificationId)	取消指定的通知。
cancelNotification(String tag, int notificationId)	取消指定的带 TAG 的通知。
cancelAllNotifications()	取消之前发布的所有通知。
addNotificationSlot(NotificationSlot slot)	创建一个 NotificationSlot。
getNotificationSlot(String slotId)	获取 NotificationSlot。
removeNotificationSlot(String slotId)	删除一个 NotificationSlot。
getActiveNotifications()	获取当前应用发的活跃通知。

表 4 NotificationHelper 主要接口

接口名	描述
getActiveNotificationNums()	获取系统中当前应用发的活跃通知的数量。
setNotificationBadgeNum(int num)	设置通知的角标。
setNotificationBadgeNum()	设置当前应用中活跃状态通知的数量在角标显示。

开发步骤

通知的开发指导分为创建 NotificationSlot、发布通知和取消通知等开发场景。

创建 NotificationSlot

NotificationSlot 可以设置公共通知的震动，锁屏模式，重要级别等，并通过调用 NotificationHelper.addNotificationSlot()发布 NotificationSlot 对象。

```
NotificationSlot slot = new NotificationSlot("slot_001","slot_default", NotificationSlot.LEVEL_MIN); // 创建 notificationSlot 对象

slot.setDescription("NotificationSlotDescription");

slot.setEnableVibration(true); // 设置振动提醒

slot.setLockscreenVisibleness(NotificationRequest.VISIBLENESS_TYPE_PUBLIC);//设置锁屏模式

slot.setEnableLight(true); // 设置开启呼吸灯提醒

slot.setLedLightColor(Color.RED.getValue());// 设置呼吸灯的提醒颜色

try {

   NotificationHelper.addNotificationSlot(slot);
```

```
} catch (RemoteException ex) {
    HiLog.warn(LABEL, "addNotificationSlot occur exception.");
}
```

发布通知

1. 构建 NotificationRequest 对象，应用发布通知前，通过 NotificationRequest 的 setSlotId() 方法与 NotificationSlot 绑定，使该通知在发布后都具备该对象的特征。

```
int notificationId = 1;
NotificationRequest request = new NotificationRequest(notificationId);
request.setSlotId(slot.getId());
```

调用 setContent()设置通知的内容。

```
String title = "title";
String text = "There is a normal notification content.";
NotificationNormalContent content = new NotificationNormalContent();
content.setTitle(title)
       .setText(text);
NotificationContent notificationContent = new
NotificationContent(content);
request.setContent(notificationContent); // 设置通知的内容
```

调用 publishNotification()发送通知。

```
try {
    NotificationHelper.publishNotification(request);
} catch (RemoteException ex) {
    HiLog.warn(LABEL, "publishNotification occur exception.");
```

```
}
```

取消通知

取消通知分为取消指定单条通知和取消所有通知，应用只能取消自己发布的通知。

- 调用 cancelNotification()取消指定的单条通知。

```
int notificationId = 1;
try {
    NotificationHelper.cancelNotification(notificationId);
} catch (RemoteException ex) {
    HiLog.warn(LABEL, "cancelNotification occur exception.");
}
```

调用 cancelAllNotifications()取消所有通知。

```
try {
    NotificationHelper.cancelAllNotifications();
} catch (RemoteException ex) {
    HiLog.warn(LABEL, "cancelAllNotifications occur exception.");
}
```

IntentAgent 开发指导

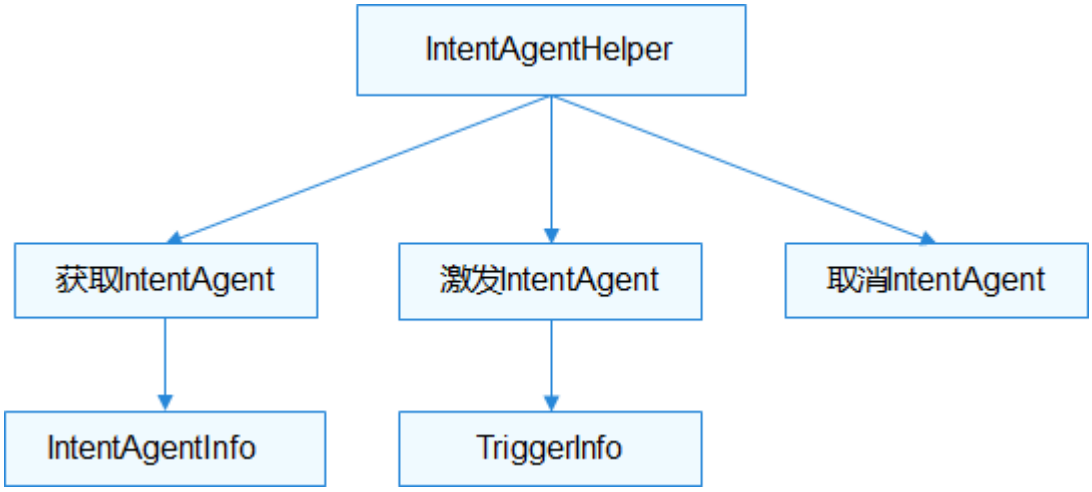
场景介绍

IntentAgent 封装了一个指定行为的 Intent，可以通过 triggerIntentAgent 接口主动触发，也可以与通知绑定被动触发。具体的行为包括：启动 Ability 和发送公共事件。例如：收到通知后，在点击通知后跳转到一个新的 Ability，不点击则不会触发。

接口说明

IntentAgent 相关基础类包括 IntentAgentHelper、IntentAgentInfo、IntentAgentConstant 和 TriggerInfo，基础类之间的关系如下图所示：

图 1 IntentAgent 基础类关系图



- **IntentAgentHelper**

IntentAgentHelper 封装了获取、激发、取消 IntentAgent 等静态方法。

表 1 IntentAgentHelper 主要接口

接口名	描述
getIntentAgent(Context context, IntentAgentInfo paramsInfo)	获取一个 IntentAgent 实例。
triggerIntentAgent(Context context, IntentAgent agent, IntentAgent.Oncompleted onCompleted, EventHandler handler, TriggerInfo paramsInfo)	主动激发一个 IntentAgent 实例。
cancel(IntentAgent agent)	取消一个 IntentAgent 实例。
judgeEquality(IntentAgent agent, IntentAgent otherAgent)	判断两个 IntentAgent 实例是否相等。
getHashCode(IntentAgent agent)	获取一个 IntentAgent 实例的哈希码。
getBundleName(IntentAgent agent)	获取一个 IntentAgent 实例的包名。
getUid(IntentAgent agent)	获取一个 IntentAgent 实例的用户 ID。

- **IntentAgentInfo**

IntentAgentInfo 类封装了获取一个 IntentAgent 实例所需的数据。使用构造函数 IntentAgentInfo(int requestCode, OperationType operationType, List<Flags> flags, List<Intent> intents, IntentParams extraInfo)获取 IntentAgentInfo 对象。

- requestCode：使用者定义的一个私有值。
- operationType：为 IntentAgentConstant.OperationType 枚举中的值。
- flags：为 IntentAgentConstant.Flags 枚举中的值。
- intents：将被执行的意图列表。operationType 的值为 START_ABILITY，START_SERVICE 和 SEND_COMMON_EVENT 时，intents 列表只允许包含一个 Intent；operationType 的值为 START_ABILITIES 时，intents 列表允许包含多个 Intent
- extraInfo：表明如何启动一个有页面的 ability，可以为 null，只在 operationType 的值为 START_ABILITY 和 START_ABILITIES 时有意义。
- **IntentAgentConstant**

IntentAgentConstant 类中包含 OperationType 和 Flags 两个枚举类：

类名	枚举值
IntentAgentConstant.OperationType	UNKNOWN_TYPE：不识别的类型。 START_ABILITY：开启一个有页面的 Ability。 START_ABILITIES：开启多个有页面的 Ability。 START_SERVICE：开启一个无页面的 ability。 SEND_COMMON_EVENT：发送一个公共事件。
IntentAgentConstant.Flags	ONE_TIME_FLAG：IntentAgent 仅能使用一次。只在 operationType 的值为 START_ABILITY，START_SERVICE 和 SEND_COMMON_EVENT 时有意义。 NO_BUILD_FLAG：如果描述 IntentAgent 对象不存在，则不创建它，直接返回 null。只在 operationType 的值为 START_ABILITY，START_SERVICE 和 SEND_COMMON_EVENT 时有

类名	枚举值
	意义。 CANCEL_PRESENT_FLAG：在生成一个新的 IntentAgent 对象前取消已存在的一个 IntentAgent 对象。只在 operationType 的值为 START_ABILITY，START_SERVICE 和 SEND_COMMON_EVENT 时有意义。 UPDATE_PRESENT_FLAG：使用新的 IntentAgent 的额外数据替换已存在的 IntentAgent 中的额外数据。只在 operationType 的值为 START_ABILITY，START_SERVICE 和 SEND_COMMON_EVENT 时有意义。 CONSTANT_FLAG：IntentAgent 是不可变的。 REPLACE_ELEMENT：当前 Intent 中的 element 属性可被 IntentAgentHelper.triggerIntentAgent()中 Intent 的 element 属性取代。 REPLACE_ACTION: 当前 Intent 中的 action 属性可被 IntentAgentHelper.triggerIntentAgent()中 Intent 的 action 属性取代。 REPLACE_URI：当前 Intent 中的 uri 属性可被 IntentAgentHelper.triggerIntentAgent()中 Intent 的 uri 属性取代。 REPLACE_ENTITIES：当前 Intent 中的 entities 属性可被 IntentAgentHelper.triggerIntentAgent()中 Intent 的 entities 属性取代。 REPLACE_BUNDLE：当前 Intent 中的 bundleName 属性可被 IntentAgentHelper.triggerIntentAgent()中 Intent 的

类名	枚举值
	bundleName 属性取代。

- **TriggerInfo**

TriggerInfo 类封装了主动激发一个 IntentAgent 实例所需的数据，使用构造函数 TriggerInfo(String permission, IntentParams extraInfo, Intent intent, int code)获取 TriggerInfo 对象。

- permission：IntentAgent 的接收者的权限名称，只在 operationType 的值为 SEND_COMMON_EVENT 时，该参数才有意义。
- extraInfo：激发 IntentAgent 时用户自定义的额外数据。
- intent：额外的 Intent。如果 IntentAgentInfo 成员变量 flags 包含 CONSTANT_FLAG，则忽略该参数；如果 flags 包含 REPLACE_ELEMENT，REPLACE_ACTION，REPLACE_URI，REPLACE_ENTITIES 或 REPLACE_BUNDLE，则使用额外 Intent 的 element，action，uri，entities 或 bundleName 属性替换原始 Intent 中对应的属性。如果 intent 是空，则不替换原始 Intent 的属性。
- code：提供给 IntentAgent 目标的结果码。

开发步骤

获取 IntentAgent 的代码示例如下：

```
// 指定要启动的 Ability 的 BundleName 和 AbilityName 字段
// 将 Operation 对象设置到 Intent 中
Operation operation = new Intent.OperationBuilder()
        .withDeviceId("")
        .withBundleName("com.huawei.testintentagent")
        .withAbilityName("com.huawei.testintentagent.entry.IntentAgentAbility")
```

```
        .build();
intent.setOperation(operation);
List<Intent> intentList = new ArrayList<>();
intentList.add(intent);
// 定义请求码
int requestCode = 200;
// 设置 flags
List<Flags> flags = new ArrayList<>();
flags.add(Flags.UPDATE_PRESENT_FLAG);
// 指定启动一个有页面的 Ability
IntentAgentInfo paramsInfo = new IntentAgentInfo(requestCode,
IntentAgentConstant.OperationType.START_ABILITY, flags, intentList,
null);
// 获取 IntentAgent 实例
IntentAgent agent = IntentAgentHelper.getIntentAgent(this, paramsInfo);
```

通知中添加 IntentAgent 的代码示例如下:

```
int notificationId = 1;
NotificationRequest request = new NotificationRequest(notificationId);
String title = "title";
String text = "There is a normal notification content.";
NotificationNormalContent content = new NotificationNormalContent();
content.setTitle(title)
       .setText(text);
NotificationContent notificationContent = new
NotificationContent(content);
request.setContent(notificationContent); // 设置通知的内容
```

```
request.setIntentAgent(agent); // 设置通知的 IntentAgent
```

主动激发 IntentAgent 的代码示例如下：

```
int code = 100;

IntentAgentHelper.triggerIntentAgent(this, agent, null, null, new
TriggerInfo(null, null, null, code ));
```

剪贴板

概述

用户通过系统剪贴板服务，可实现应用之间的简单数据传递。例如：在应用 A 中复制的数据，可以在应用 B 中粘贴，反之亦可。

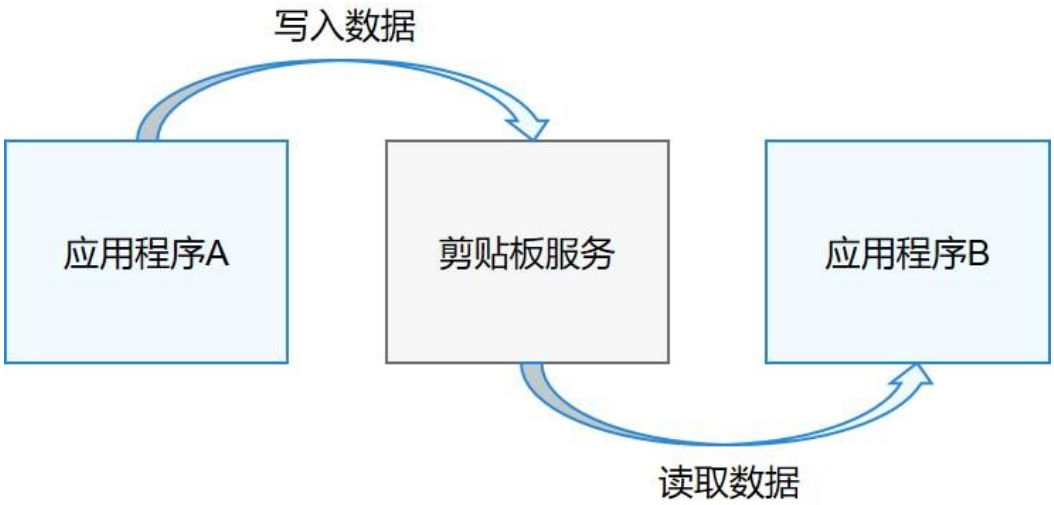
- HarmonyOS 提供系统剪贴板服务的操作接口，支持用户程序从系统剪贴板中读取、写入和查询剪贴板数据，以及添加、移除系统剪贴板数据变化的回调。
- HarmonyOS 提供剪贴板数据的对象定义，包含内容对象和属性对象。
- HarmonyOS 支持跨设备分布式剪贴板服务。

开发指导

场景介绍

同一设备的应用程序 A、B 之间可以借助系统剪贴板服务完成简单数据的传递，即应用程序 A 向剪贴板服务写入数据后，应用程序 B 可以从中读取出数据。在满足分布式剪贴板服务的使用条件时，应用程序 A、B 也可以来自组网内的不同设备。

图 1　剪贴板服务示意图



在使用剪贴板服务时，需要注意以下几点：

- 只有在前台获取到焦点的应用才有读取系统剪贴板的权限（系统默认输入法应用除外）。
- 写入到剪贴板服务中的剪贴板数据不会随应用程序结束而销毁。
- 对同一用户而言，写入剪贴板服务的数据会被下一次写入的剪贴板数据所覆盖。
- 如果设备满足分布式组网条件，且进行复制操作的设备打开了剪贴板分布式开关，未配置“仅在本地”标志位的剪贴板数据里的 MIME 类型为纯文本和 HTML 的内容可以被组网内其他打开了剪贴板分布式开关的设备粘贴出来。
- 在同一设备内，剪贴板单次传递内容不应超过 800KB。在分布式场景下多设备间传递时，每次传递内容不应超过 64KB。

接口说明

SystemPasteboard 提供系统剪贴板操作的相关接口，比如复制、粘贴、配置回调等。PasteData 是剪贴板服务操作的数据对象，一个 PasteData 由若干个内容节点（PasteData.Record）和一个属性集合对象（PasteData.DataProperty）组成。Record 是存放剪贴板数据内容信息的最小单位，每个 Record 都有其特定的 MIME 类型，如纯文本、HTML、URI、Intent。剪贴板数据的属性信息存在放 DataProperty 中，包括标签、时间戳、“仅在本地”标记位等。

SystemPasteboard

SystemPasteboard 提供系统剪贴板服务的操作接口，比如复制、粘贴、配置回调等。

表 1 SystemPasteboard 的主要接口

接口名	描述
getSystemPasteboard(Context context)	获取系统剪切板服务的对象实例。
getPasteData()	读取当前系统剪贴板中的数据。
hasPasteData()	判断当前系统剪贴板中是否有内容。
setPasteData(PasteData data)	将剪贴板数据写入到系统剪贴板。
clear()	清空系统剪贴板数据。
addPasteDataChangedListener(IPasteDataChangedListener	用户程序添

表 1 SystemPasteboard 的主要接口

接口名	描述
listener)	加系统剪贴板数据变化的回调，当系统剪贴板数据发生变化时，会触发用户程序的回调实现。
removePasteDataChangedListener(IPasteDataChangedListener listener)	用户程序移除系统剪贴板数据变化的回调。

PasteData

PasteData 是剪贴板服务操作的数据对象，其中内容节点定义为 PasteData.Record，属性集合定义为 PasteData.DataProperty。

表 2 PasteData 的主要接口

接口名	描述
PasteData()	构造器，创建一个空内容数据对象。
createPlainTextData(CharSequence text)	构建一个包含纯文本内容节点的数据对象。
creatHtmlData(String htmlText)	构建一个包含 HTML 内容节点的数据对象。
creatUriData(Uri uri)	构建一个包含 URI 内容节点的数据对象。

表 2 PasteData 的主要接口

接口名	描述
creatIntentData(Intent intent)	构建一个包含 Intent 内容节点的数据对象。
getPrimaryMimeType()	获取数据对象中首个内容节点的 MIME 类型，如果没有查询到内容，将返回一个空字符串。
getPrimaryText()	获取数据对象中首个内容节点的纯文本内容，如果没有查询到内容，将返回一个空对象。
addTextRecord(CharSequence text)	向数据对象中添加一个纯文本内容节点，该方法会自动更新数据属性中的 MIME 类型集合，最多只能添加 128 个内容节点。
addRecord(Record record)	向数据对象中添加一个内容节点，该方法会自动更新数据属性中的 MIME 类型集合，最多只能添加 128 个内容节点。
getRecordCount()	获取数据对象中内容节点的数量。
getRecordAt(int index)	获取数据对象在指定下标处的内容节点，如果操作失败会返回空对象。
removeRecordAt(int index)	移除数据对象在指定下标处的内容节点，如果操作成功会返回 true，操作失败会返回 false。
getMimeTypes()	获取数据对象中上所有内容节点的 MIME 类型列表，当内容节点为空时，返回列表为空对象。

表 2 PasteData 的主要接口

接口名	描述
getProperty()	获取该数据对象的属性集合成员。

表 3 PasteData 中定义的常量

常量名	描述
MIMETYPE_TEXT_PLAIN= "text/plain"	纯文本的 MIME 类型定义。
MIMETYPE_TEXT_HTML= "text/html"	HTML 的 MIME 类型定义。
MIMETYPE_TEXT_URI= "text/uri"	URI 的 MIME 类型定义。
MIMETYPE_TEXT_INTENT= "text/ohos.intent"	Intent 的 MIME 类型定义。
MAX_RECORD_NUM=128	单个 PasteData 中所能包含的 Record 的数量上限。

PasteData.Record

一个 PasteData 中包含若干个特定 MIME 类型的 PasteData.Record，每个 Record 是存放剪贴板数据内容信息的最小单位。

表 4 PasteData.Record 的主要接口

接口名	描述
createPlainTextRecord(CharSequence text)	构造一个 MIME 类型为纯文本的内容节点。
createHtmlTextRecord(String htmlText)	构造一个 MIME 类型为 HTML 的内容节点。
createUriRecord(Uri uri)	构造一个 MIME 类型为 URI 的内容节点。

表 4 PasteData.Record 的主要接口

接口名	描述
createIntentRecord(Intent intent)	构造一个 MIME 类型为 Intent 的内容节点。
getPlainText()	获取该内容节点中的文本内容，如果没有内容将返回空对象。
getHtmlText()	获取该内容节点中的 HTML 内容，如果没有内容将返回空对象。
getUri()	获取该内容节点中的 URI 内容，如果没有内容将返回空对象。
getIntent()	获取该内容节点中的 Intent 内容，如果没有内容将返回空对象。
getMimeType()	获取该内容节点的 MIME 类型。
convertToText(Context context)	将该内容节点的内容转为文本形式。

PasteData.DataProperty

每个 PasteData 中都有一个 PasteData.DataProperty 成员，其中存放着该数据对象的属性集合，例如自定义标签、MIME 类型集合列表，“仅在本地”标记位等。

表 5 PasteData.DataProperty 的主要接口

接口名	描述
getMimeTypes()	获取所属数据对象的 MIME 类型集合列表，当内容节点为空时，返回列表为空对象。
hasMimeType(String mimeType)	判断所属数据对象中是否包含特定 MIME 类型的内容。

表 5 PasteData.DataProperty 的主要接口

接口名	描述
getTimestamp()	获取所属数据对象被写入系统剪贴板时的时间戳，如果该数据对象尚未被写入，则返回 0。
setTag(CharSequence tag)	设置自定义标签。
getTag()	获取自定义标签。
setAdditions(PacMap extraProps)	设置一些附加键值对信息。
getAdditions()	获取附加键值对信息。
setLocalOnly(boolean isLocalonly)	配置“仅在本地”标志位，默认配置为 false，表示此数据对象能在分布式剪贴板场景下跨设备传递，否则只在本地设备使用。
isLocalOnly()	查询“仅在本地”标志位。

IPasteDataChangedListener

IPasteDataChangedListener 是定义剪贴板数据变化回调的接口类，开发者需要实现此接口来编码触发回调时的处理逻辑。

表 6 IPasteDataChangedListener 的主要接口

接口名	描述
onChanged()	当系统剪贴板数据发生变化时的回调接口。

开发步骤

1. 应用 A 获取系统剪贴板服务。

```
SystemPasteboard pasteboard =
SystemPasteboard.getSystemPasteboard(appContext);
```

应用 A 向系统剪贴板中写入一条纯文本数据。

```
if (pasteboard != null) {
    pasteboard.setPasteData(PasteData.creatPlainTextData("Hello,
world!"));
}
```

应用 B 从系统剪贴板读取数据，将数据对象中的首个文本类型（纯文本/HTML）内容信息在控件中显示，忽略其他类型内容。

```
PasteData pasteData = pasteboard.getPasteData();
if (pasteData == null) {
    return;
}
DataProperty dataProperty = pasteData.getProperty();
boolean hasHtml = dataProperty.hasMimeType(PasteData.MIMETYPE_TEXT_HTML);
boolean hasText =
dataProperty.hasMimeType(PasteData.MIMETYPE_TEXT_PLAIN);
if (hasHtml || hasText) {
    for (int i = 0; i < pasteData.getRecordCount(); i++) {
        Record record = pasteData.getRecordAt(i);
        String mimeType = record.getMimeType();
        if (mimeType.equals(PasteData.MIMETYPE_TEXT_HTML)) {
            text.setText(record.getHtmlText());
            break;
        } else if (mimeType.equals(PasteData.MIMETYPE_TEXT_PLAIN)) {
            text.setText(record.getPlainText().toString());
```

```
            break;
        }
    }
}
```

应用 C 注册添加系统剪贴板数据变化回调，当系统剪贴板数据发生变化时触发处理逻辑。

```
IPasteDataChangedListener listener = new IPasteDataChangedListener() {
    @Override
    public void onChanged() {
        PasteData pasteData = pasteboard.getPasteData();
        if (pasteData == null) {
            return;
        }
        // Operations to handle data change on the system pasteboard
    }
};
pasteboard.addPasteDataChangedListener(listener);
```

线程

线程管理

概述

不同应用在各自独立的进程中运行。当应用以任何形式启动时，系统为其创建进程，该进程将持续运行。当进程完成当前任务处于等待状态，且系统资源不足时，系统自动回收。

在启动应用时，系统会为该应用创建一个称为“主线程”的执行线程。该线程随着应用创建或消失，是应用的核心线程。UI 界面的显示和更新等操作，都是在主线程上进行。主线程又称 UI 线程，默认情况下，所有的操作都是在主线程上执行。如果需要执行比较耗时的任务（如下载文件、查询数据库），可创建其他线程来处理。

开发指导

场景介绍

如果应用的业务逻辑比较复杂，可能需要创建多个线程来执行多个任务。这种情况下，代码复杂难以维护，任务与线程的交互也会更加繁杂。要解决此问题，开发者可以使用“TaskDispatcher”来分发不同的任务。

接口说明

TaskDispatcher 是一个任务分发器，它是 Ability 分发任务的基本接口，隐藏任务所在线程的实现细节。

为保证应用有更好的响应性，我们需要设计任务的优先级。在 UI 线程上运行的任务默认以高优先级运行，如果某个任务无需等待结果，则可以用低优先级。

表 1 线程优先级介绍

优先级	详细描述
HIGH	最高任务优先级，比默认优先级、低优先级的任务有更高的几率得到执行。
DEFAULT	默认任务优先级， 比低优先级的任务有更高的几率得到执行。
LOW	低任务优先级，比高优先级、默认优先级的任务有更低的几率得到执行。

TaskDispatcher 具有多种实现，每种实现对应不同的任务分发器。在分发任务时可以指定任务的优先级，由同一个任务分发器分发出的任务具有相同的优先级。系统提供的任务分发器有 GlobalTaskDispatcher、ParallelTaskDispatcher、SerialTaskDispatcher 、SpecTaskDispatcher。

- **GlobalTaskDispatcher**

全局并发任务分发器，由 Ability 执行 getGlobalTaskDispatcher()获取。适用于任务之间没有联系的情况。一个应用只有一个 GlobalTaskDispatcher，它在程序结束时才被销毁。

```
TaskDispatcher globalTaskDispatcher =
getGlobalTaskDispatcher(TaskPriority.DEFAULT);
```

ParallelTaskDispatcher

并发任务分发器，由 Ability 执行 createParallelTaskDispatcher()创建并返回。与 GlobalTaskDispatcher 不同的是，ParallelTaskDispatcher 不具有全局唯一性，可以创建多个。开发者在创建或销毁 dispatcher 时，需要持有对应的对象引用 。

```
String dispatcherName = "parallelTaskDispatcher";
TaskDispatcher parallelTaskDispatcher =
createParallelTaskDispatcher(dispatcherName, TaskPriority.DEFAULT);
```

SerialTaskDispatcher

串行任务分发器，由 Ability 执行 createSerialTaskDispatcher()创建并返回。由该分发器分发的所有的任务都是按顺序执行，但是执行这些任务的线程并不是固定的。如果要执行并行任务，应使用 ParallelTaskDispatcher 或者 GlobalTaskDispatcher，而不是创建多个 SerialTaskDispatcher。如果任务之间没有依赖，应使用 GlobalTaskDispatcher 来实现。它的创建和销毁由开发者自己管理，开发者在使用期间需要持有该对象引用。

```
String dispatcherName = "parallelTaskDispatcher";
```

```
TaskDispatcher parallelTaskDispatcher =
createParallelTaskDispatcher(dispatcherName, TaskPriority.DEFAULT);
```

SerialTaskDispatcher

串行任务分发器，由 Ability 执行 createSerialTaskDispatcher()创建并返回。由该分发器分发的所有的任务都是按顺序执行，但是执行这些任务的线程并不是固定的。如果要执行并行任务，应使用 ParallelTaskDispatcher 或者 GlobalTaskDispatcher，而不是创建多个 SerialTaskDispatcher。如果任务之间没有依赖，应使用 GlobalTaskDispatcher 来实现。它的创建和销毁由开发者自己管理，开发者在使用期间需要持有该对象引用。

```
String dispatcherName = "serialTaskDispatcher";
TaskDispatcher serialTaskDispatcher =
createSerialTaskDispatcher(dispatcherName, TaskPriority.DEFAULT);
```

SpecTaskDispatcher

专有任务分发器，绑定到专有线程上的任务分发器。目前已有的专有线程是主线程。UITaskDispatcher 和 MainTaskDispatcher 都属于 SpecTaskDispatcher。建议使用 UITaskDispatcher。

UITaskDispatcher：绑定到应用主线程的专有任务分发器， 由 Ability 执行 getUITaskDispatcher()创建并返回。由该分发器分发的所有的任务都是在主线程上按顺序执行，它在应用程序结束时被销毁。

```
TaskDispatcher uiTaskDispatcher = getUITaskDispatcher();
```

MainTaskDispatcher：由 Ability 执行 getMainTaskDispatcher()创建并返回。

```
TaskDispatcher mainTaskDispatcher= getMainTaskDispatcher()
```

开发步骤

- **syncDispatch**

同步派发任务：派发任务并在当前线程等待任务执行完成。在返回前，当前线程会被阻塞。

如下代码示例展示了如何使用 GlobalTaskDispatcher 派发同步任务：

```
globalTaskDispatcher.syncDispatch(new Runnable() {
            @Override
            public void run() {
                HiLog.info(label, "sync task1 run");
            }
        });
HiLog.info(label, "after sync task1");

globalTaskDispatcher.syncDispatch(new Runnable() {
            @Override
            public void run() {
                HiLog.info(label, "sync task2 run");
            }
        });
HiLog.info(label, "after sync task2");
```

```
globalTaskDispatcher.syncDispatch(new Runnable() {
            @Override
            public void run() {
                HiLog.info(label, "sync task3 run");
            }
        });
HiLog.info(label, "after sync task3");

// 执行结果如下：
// sync task1 run
// after sync task1
// sync task2 run
// after sync task2
// sync task3 run
// after sync task3
```

说明

如果对 syncDispatch 使用不当，将会导致死锁。如下情形可能导致死锁发生：

- 在专有线程上，利用该专有任务分发器进行 syncDispatch。
- 在被某个串行任务分发器（dispatcher_a）派发的任务中，再次利用同一个串行任务分发器（dispatcher_a）对象派发任务。

- 在被某个串行任务分发器（dispatcher_a）派发的任务中，经过数次派发任务，最终又利用该（dispatcher_a）串行任务分发器派发任务。例如：dispatcher_a 派发的任务使用 dispatcher_b 进行任务的派发，在 dispatcher_b 派发的任务中又利用 dispatcher_a 进行派发任务。

- 串行任务分发器（dispatcher_a）派发的任务中利用串行任务分发器（dispatcher_b）进行同步派发任务，同时 dispatcher_b 派发的任务中利用串行任务分发器（dispatcher_a）进行同步派发任务。在特定的线程执行顺序下将导致死锁。

asyncDispatch

异步派发任务：派发任务，并立即返回，返回值是一个可用于取消任务的接口。

如下代码示例展示了如何使用 GlobalTaskDispatcher 派发异步任务：

```
Revocable revocable =  globalTaskDispatcher.asyncDispatch(new
Runnable() {
        @Override
            public void run() {
                HiLog.info(label, "async task1 run");
            }
        });
HiLog.info(label, "after async task1");

// 执行结果可能如下：
// after async task1
// async task1 run
```

delayDispatch

异步延迟派发任务：异步执行，函数立即返回，内部会在延时指定时间后将任务派发到相应队列中。延时时间参数仅代表在这段时间以后任务分发器会将任务加入到队列中，任务的实际执行时间可能晚于这个时间。具体比这个数值晚多久，取决于队列及内部线程池的繁忙情况。

如下代码示例展示了如何使用 GlobalTaskDispatcher 延迟派发任务：

```
 final long callTime = System.currentTimeMillis();
 final long delayTime = 50;
 Revocable revocable =  globalTaskDispatcher.delayDispatch(new
Runnable() {
            @Override
            public void run() {
                HiLog.info(label, "delayDispatch task1 run");
                final long actualDelayMs = System.currentTimeMillis() -
callTime;
                HiLog.info(label, "actualDelayTime >=
delayTime : %{public}b" + (actualDelayMs >= delayTime));
            }
        }, delayTime );
 HiLog.info(label, "after delayDispatch task1");


// 执行结果可能如下：
// after delayDispatch task1
// delayDispatch task1 run
```

```
// actualDelayTime >= delayTime : true
```

Group

任务组：表示一组任务，且该组任务之间有一定的联系，由 TaskDispatcher 执行 createDispatchGroup 创建并返回。将任务加入任务组，返回一个用于取消任务的接口。

如下代码示例展示了任务组的使用方式：将一系列相关联的下载任务放入一个任务组，执行完下载任务后关闭应用。

```
void groupTest(Context context) {
    TaskDispatcher dispatcher =
context.createParallelTaskDispatcher(dispatcherName,
TaskPriority.DEFAULT);
    // 创建任务组。
    Group group = dispatcher.createDispatchGroup();
    // 将任务 1 加入任务组，返回一个用于取消任务的接口。
    dispatcher.asyncGroupDispatch(group, new Runnable(){
        public void run() {
            HiLog.info(label, "download task1 is running");
            downLoadRes(url1);
        }
    });
   // 将与任务 1 相关联的任务 2 加入任务组。
    dispatcher.asyncGroupDispatch(group, new Runnable(){
        public void run() {
            HiLog.info(label, "download task2 is running");
            downLoadRes(url2);
```

```
        }
    });
    // 在任务组中的所有任务执行完成后执行指定任务。
    dispatcher.groupDispatchNotify(group, new Runnable(){
        public void run() {
            HiLog.info(label, "the close task is running after all tasks in the group are completed");
            closeApp();
        }
    });
}

// 可能的执行结果:
// download task1 is running
// download task2 is running
// the close task is running after all tasks in the group are completed

// 另外一种可能的执行结果:
// download task2 is running
// download task1 is running
// the close task is running after all tasks in the group are completed
```

Revocable

取消任务：Revocable 是取消一个异步任务的接口。异步任务包括通过 asyncDispatch、delayDispatch、asyncGroupDispatch 派发的任务。如果任务已经在执行中或执行完成，则会返回取消失败。

如下代码示例展示了如何取消一个异步延时任务：

```
void postTaskAndRevoke(Context context) {
    TaskDispatcher dispatcher = context.getUITaskDispatcher();
    Revocable revocable = dispatcher.delayDispatch(new Runnable(){
         HiLog.info(label, "delay dispatch");
    }, 10);
    boolean revoked = revocable.revoke();
    HiLog.info(label, "%{public}b", revoked);
}

// 一种可能的结果如下 :
// true
```

syncDispatchBarrier

同步设置屏障任务：在任务组上设立任务执行屏障，同步等待任务组中的所有任务执行完成，再执行指定任务。

说明

在全局并发任务分发器（GlobalTaskDispatcher）上同步设置任务屏障，将不会起到屏障作用。

如下代码示例展示了如何同步设置屏障：

```
    TaskDispatcher dispatcher =
context.createParallelTaskDispatcher(dispatcherName,
TaskPriority.DEFAULT);
    // 创建任务组。
    Group group = dispatcher.createDispatchGroup();
    // 将任务加入任务组，返回一个用于取消任务的接口。
    dispatcher.asyncGroupDispatch(group, new Runnable(){
        public void run() {
            HiLog.info(label, "task1 is running");  // 1
        }
    });
    dispatcher.asyncGroupDispatch(group, new Runnable(){
        public void run() {
```

```
            HiLog.info(label, "task2 is running");  // 2
        }
    });

    dispatcher.syncDispatchBarrier(new Runnable() {
    public void run() {
        HiLog.info(label, "barrier");  // 3
    }});
    HiLog.info(label, "after syncDispatchBarrier");  // 4
}

// 1 和 2 的执行顺序不定；3 和 4 总是在 1 和 2 之后按顺序执行。

// 可能的执行结果:
// task1 is running
// task2 is running
// barrier
// after syncDispatchBarrier

// 另外一种执行结果：
// task2 is running
// task1 is running
// barrier
// after syncDispatchBarrier
```

asyncDispatchBarrier

异步设置屏障任务：在任务组上设立任务执行屏障后直接返回，指定任务将在任务组中的所有任务执行完成后再执行。

说明

在全局并发任务分发器（GlobalTaskDispatcher）上异步设置任务屏障，将不会起到屏障作用。可以使用并发任务分发器（ParallelTaskDispatcher）分离不同的任务组，达到微观并行、宏观串行的行为。

如下代码示例展示了如何异步设置屏障：

```
    TaskDispatcher dispatcher =
context.createParallelTaskDispatcher(dispatcherName,
TaskPriority.DEFAULT);
    // 创建任务组。
    Group group = dispatcher.createDispatchGroup();
    // 将任务加入任务组，返回一个用于取消任务的接口。
    dispatcher.asyncGroupDispatch(group, new Runnable(){
        public void run() {
            HiLog.info(label, "task1 is running");  // 1
        }
    });
```

```
    dispatcher.asyncGroupDispatch(group, new Runnable(){
        public void run() {
            HiLog.info(label, "task2 is running");  // 2
        }
    });

    dispatcher.asyncDispatchBarrier(new Runnable() {
        public void run() {
            HiLog.info(label, "barrier");  // 3
        }
});
    HiLog.info(label, "after syncDispatchBarrier");  // 4
}

// 1 和 2 的执行顺序不定，但总在 3 和 4 之前执行；4 可能在 3 之前执行

// 可能的执行结果:
// task1 is running
// task2 is running
// after syncDispatchBarrier
// barrier
```

applyDispatch

执行多次任务：对指定任务执行多次。

如下代码示例展示了如何执行多次任务：

```
 final int total = 10;
 final CountDownLatch latch = new CountDownLatch(total);
```

```
final ArrayList<Long> indexList = new ArrayList<>(total);


// 执行任务 total 次
dispatcher.applyDispatch((index) -> {
    indexList.add(index);
    latch.countDown();
}, total);


// 设置任务超时
try {
    latch.await();
} catch (InterruptedException exception) {
   HiLog.info(label, "latch exception");
}
HiLog.info(label, "list size matches, %{public}b", (total ==
indexList.size()));


// 执行结果：
// list size matches, true
```

线程间通信

概述

在开发过程中，开发者经常需要在当前线程中处理下载任务等较为耗时的操作，但是又不希望当前的线程受到阻塞。此时，就可以使用 EventHandler 机制。EventHandler 是 HarmonyOS 用于处理线程间通信的一种机制，可以通过 EventRunner 创建新线程，将耗时的操作放到新线程上执行。这样既不阻塞原来的线程，任务又可以得到合理的处理。比如：主线程使用 EventHandler 创建子线程，子线程做耗时的下载图片操作，下载完成后，子线程通过 EventHandler 通知主线程，主线程再更新 UI。

基本概念

EventRunner 是一种事件循环器，循环处理从该 EventRunner 创建的新线程的事件队列中获取 InnerEvent 事件或者 Runnable 任务。InnerEvent 是 EventHandler 投递的事件。

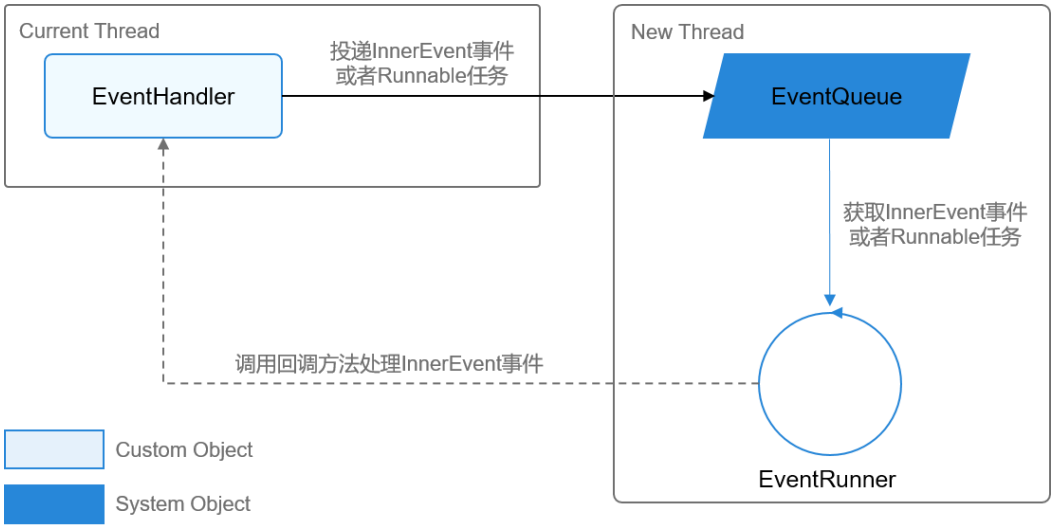
EventHandler 是一种用户在当前线程上投递 InnerEvent 事件或者 Runnable 任务到异步线程上处理的机制。每一个 EventHandler 和指定的 EventRunner 所创建的新线程绑定，并且该新线程内部有一个事件队列。EventHandler 可以投递指定的 InnerEvent 事件或 Runnable 任务到这个事件队列。EventRunner 从事件队列里循环地取出事件，如果取出的事件是 InnerEvent 事件，将在 EventRunner 所在线程执行 processEvent 回调；如果取出的事件是 Runnable 任务，将在 EventRunner 所在线程执行 Runnable 的 run 回调。一般，EventHandler 有两个主要作用：

- 在不同线程间分发和处理 InnerEvent 事件或 Runnable 任务。
- 延迟处理 InnerEvent 事件或 Runnable 任务。

运作机制

EventHandler 的运作机制如下图所示：

图 1　EventHandler 的运作机制



使用 EventHandler 实现线程间通信的主要流程：

1. EventHandler 投递具体的 InnerEvent 事件或者 Runnable 任务到 EventRunner 所创建的线程的事件队列。
2. EventRunner 循环从事件队列中获取 InnerEvent 事件或者 Runnable 任务。
3. 处理事件或任务：

- 如果 EventRunner 取出的事件为 InnerEvent 事件，则触发 EventHandler 的回调方法并触发 EventHandler 的处理方法，在新线程上处理该事件。
- 如果 EventRunner 取出的事件为 Runnable 任务，则 EventRunner 直接在新线程上处理 Runnable 任务。

约束限制

- 在进行线程间通信的时候，EventHandler 只能和 EventRunner 所创建的线程进行绑定，EventRunner 创建时需要判断是否创建成功，只有确保获取的 EventRunner 实例非空时，才可以使用 EventHandler 绑定 EventRunner。
- 一个 EventHandler 只能同时与一个 EventRunner 绑定，一个 EventRunner 上可以创建多个 EventHandler。

开发指导

场景介绍

EventHandler 开发场景

EventHandler 的主要功能是将 InnerEvent 事件或者 Runnable 任务投递到其他的线程进行处理，其使用的场景包括：

- 开发者需要将 InnerEvent 事件投递到新的线程，按照优先级和延时进行处理。投递时，EventHandler 的优先级可在 IMMEDIATE、HIGH、LOW、IDLE 中选择，并设置合适的 delayTime。
- 开发者需要将 Runnable 任务投递到新的线程，并按照优先级和延时进行处理。投递时，EventHandler 的优先级可在 IMMEDIATE、HIGH、LOW、IDLE 中选择，并设置合适的 delayTime。
- 开发者需要在新创建的线程里投递事件到原线程进行处理。

EventRunner 工作模式

EventRunner 的工作模式可以分为托管模式和手动模式。两种模式是在调用 EventRunner 的 create()方法时，通过选择不同的参数来实现的，详见 API 参考。默认为托管模式。

- 托管模式：不需要开发者调用 run()和 stop()方法去启动和停止 EventRunner。当 EventRunner 实例化时，系统调用 run()来启动 EventRunner；当 EventRunner 不被引用时，系统调用 stop()来停止 EventRunner。
- 手动模式：需要开发者自行调用 EventRunner 的 run()方法和 stop()方法来确保线程的启动和停止。

接口说明

EventHandler

- EventHandler 的属性 Priority(优先级)介绍：

EventRunner 将根据优先级的高低从事件队列中获取事件或者 Runnable 任务进行处理。

表 1 EventHandler 的属性

属性	描述
Priority.IMMEDIATE	表示事件被立即投递
Priority.HIGH	表示事件先于 LOW 优先级投递
Priority.LOW	表示事件优于 IDLE 优先级投递，事件的默认优先级是 LOW
Priority.IDLE	表示在没有其他事件的情况下，才投递该事件

- EventHandler 的主要接口介绍：

表 2 EventHandler 的主要接口

接口名	描述
EventHandler(EventRunner runner)	利用已有的 EventRunner 来创建 EventHandler
current()	在 processEvent 回调中，获取当前的 EventHandler
processEvent(InnerEvent event)	回调处理事件，由开发者实现
sendEvent(InnerEvent event)	发送一个事件到事件队列，延时为 0ms，优先级为 LOW
sendEvent(InnerEvent event, long delayTime)	发送一个延时事件到事件队列，优先级为 LOW

表 2 EventHandler 的主要接口

接口名	描述
sendEvent(InnerEvent event, long delayTime, EventHandler.Priority priority)	发送一个指定优先级的延时事件到事件队列
sendEvent(InnerEvent event, EventHandler.Priority priority)	发送一个指定优先级的事件到事件队列，延时为 0ms
sendSyncEvent(InnerEvent event)	发送一个同步事件到事件队列，延时为 0ms，优先级为 LOW
sendSyncEvent(InnerEvent event, EventHandler.Priority priority)	发送一个指定优先级的同步事件到事件队列，延时为 0ms，优先级不可以是 IDLE
postSyncTask(Runnable task)	发送一个 Runnable 同步任务到事件队列，延时为 0ms，优先级为 LOW
postSyncTask(Runnable task, EventHandler.Priority priority)	发送一个指定优先级的 Runnable 同步任务到事件队列，延时为 0ms
postTask(Runnable task)	发送一个 Runnable 任务到事件队列，延时为 0ms，优先级为 LOW
postTask(Runnable task, long delayTime)	发送一个 Runnable 延时任务到事件队列，优先级为 LOW
postTask(Runnable task, long delayTime, EventHandler.Priority priority)	发送一个指定优先级的 Runnable 延时任务到事件队列

表 2 EventHandler 的主要接口

接口名	描述
postTask(Runnable task, EventHandler.Priority priority)	发送一个指定优先级的 Runnable 任务到事件队列，延时为 0ms
sendTimingEvent(InnerEvent event, long taskTime)	发送一个定时事件到队列，在 taskTime 时间执行，如果 taskTime 小于当前时间，立即执行，优先级为 LOW
sendTimingEvent(InnerEvent event, long taskTime, EventHandler.Priority priority)	发送一个带优先级的事件到队列，在 taskTime 时间执行，如果 taskTime 小于当前时间，立即执行
postTimingTask(Runnable task, long taskTime)	发送一个 Runnable 任务到队列，在 taskTime 时间执行，如果 taskTime 小于当前时间，立即执行，优先级为 LOW
postTimingTask(Runnable task, long taskTime, EventHandler.Priority priority)	发送一个带优先级的 Runnable 任务到队列，在 taskTime 时间执行，如果 taskTime 小于当前时间，立即执行
removeEvent(int eventId)	删除指定 id 的事件
removeEvent(int eventId, long param)	删除指定 id 和 param 的事件
removeEvent(int eventId, long param, Object object)	删除指定 id、param 和 object 的事件
removeAllEvent()	删除该 EventHandler 的所有

表 2 EventHandler 的主要接口

接口名	描述
	事件
getEventName(InnerEvent event)	获取事件的名字
getEventRunner()	获取该 EventHandler 绑定的 EventRunner
isIdle()	判断队列是否为空
hasInnerEvent(Runnable runnable）	是否有还未被处理的这个任务

EventRunner

- EventRunner 的主要接口介绍：

表 3 EventRunner 主要接口

接口名	描述
create()	创建一个拥有新线程的 EventRunner
create(boolean isDeposited)	创建一个拥有新线程的 EventRunner，isDeposited 为 true 时，EventRunner 为托管模式，系统将自动管理该 EventRunner；isDeposited 为 false 时，EventRunner 为手动模式。
create (String newThreadName)	创建一个拥有新线程的 EventRunner， 新线程的名字是 newThreadName
current()	获取当前线程的 EventRunner
run()	EventRunner 为手动模式时，调用该方法启动新的线程
stop()	EventRunner 为手动模式时，调用该方法停止新的线程

InnerEvent

- InnerEvent 的属性介绍：

表 4 InnerEvent 的属性

属性	描述
eventId	事件的 ID，由开发者定义用来辨别事件
object	事件携带的 Object 信息
param	事件携带的 long 型数据

- InnerEvent 的主要接口介绍：

表 5 InnerEvent 的接口

接口名	描述
drop()	释放一个事件实例
get()	获得一个事件实例
get(int eventId)	获得一个指定的 eventId 的事件实例
get(int eventId, long param)	获得一个指定的 eventId 和 param 的事件实例
get(int eventId, long param, Object object)	获得一个指定的 eventId，param 和 object 的事件实例
get(int eventId, Object object)	获得一个指定的 eventId 和 object 的事件实例
PacMap getPacMap()	获取 PacMap，如果没有，会新建一个
Runnable getTask()	获取 Runnable 任务

表 5 InnerEvent 的接口

接口名	描述
PacMap peekPacMap()	获取 PacMap
void setPacMap(PacMap pacMap)	设置 PacMap

开发步骤

EventHandler 投递 InnerEvent 事件

EventHandler 投递 InnerEvent 事件，并按照优先级和延时进行处理，开发步骤如下：

1. 创建 EventHandler 的子类，在子类中重写实现方法 processEvent()来处理事件。

```
private class MyEventHandler extends EventHandler {
    private MyEventHandler(EventRunner runner) {
    super(runner);
    }
    // 重写实现 processEvent 方法
    @Override
    public void processEvent(InnerEvent event) {
    super.processEvent(event);
    if (event == null) {
            return;
    }
    int eventId = event.eventId;
```

```
    long param = event.param;
    switch (eventId | param) {
            case CASE1:
                // 待执行的操作，由开发者定义
            break;
        default:
        break;
    }
    }
}
```

创建 EventRunner，以手动模式为例。

```
EventRunner runner = EventRunner.create(false);// create()的参数是 true 时，则为托管模式

// 需要对 EventRunner 的实例进行校验，因为创建 EventRunner 可能失败，如创建线程失败时，创建 EventRunner 失败。

if (runner == null) {

    return;

}
```

创建 EventHandler 子类的实例。

```
MyEventHandler myHandler = new MyEventHandler(runner);
```

获取 InnerEvent 事件。

```
// 获取事件实例，其属性 eventId, param, object 由开发者确定，代码中只是示例。

int eventId1 = 0;

int eventId2 = 1;

long param = 0;
```

```
Object object = null;

InnerEvent event1 = InnerEvent.get(eventId1, param, object);

InnerEvent event2 = InnerEvent.get(eventId2, param, object);
```

投递事件，投递的优先级以 IMMEDIATE 为例，延时选择 0ms 和 2ms。

```
// 优先级 immediate，投递之后立即处理，延时为 0ms，该语句等价于同步投递
sendSyncEvent(event1，EventHandler.Priority.immediate);

myHandler.sendEvent(event1, 0, EventHandler.Priority.IMMEDIATE);

myHandler.sendEvent(event2, 2, EventHandler.Priority.IMMEDIATE); // 延时 2ms 后立即处理
```

启动和停止 EventRunner，如果为托管模式，则不需要此步骤。

```
runner.run();

//待执行操作

runner.stop();// 开发者根据业务需要在适当时机停止 EventRunner
```

EventHandler 投递 Runnable 任务

EventHandler 投递 Runnable 任务，并按照优先级和延时进行处理，开发步骤如下：

1. 创建 EventHandler 的子类，创建 EventRunner，并创建 EventHandler 子类的实例，步骤与 EventHandler 投递 InnerEvent 场景的步骤 1-3 相同。
2. 创建 Runnable 任务。

```
Runnable task1 = new Runnable() {
    @Override
    public void run() {
```

```
        // 待执行的操作，由开发者定义
    }
};
Runnable task2 = new Runnable() {
    @Override
    public void run() {
        // 待执行的操作，由开发者定义
    }
};
```

投递Runnable任务,投递的优先级以IMMEDIATE为例,延时选择0ms和2ms。

```
//优先级为 immediate，延时 0ms，该语句等价于同步投递
myHandler.postSyncTask(task1，EventHandler.Priority.immediate);

myHandler.postTask(task1，0，  EventHandler.Priority.IMMEDIATE);


myHandler.postTask(task2，2，  EventHandler.Priority.IMMEDIATE);// 延时 2ms 后立即执行
```

启动和停止 EventRunner，如果是托管模式，则不需要此步骤。

```
runner.run();

//待执行操作


runner.stop();// 停止 EventRunner
```

在新创建的线程里投递事件到原线程

EventHandler 从新创建的线程投递事件到原线程并进行处理，开发步骤如下：

1. 创建 EventHandler 的子类，在子类中重写实现方法 processEvent()来处理事件。

```
private class MyEventHandler extends EventHandler {
    private MyEventHandler(EventRunner runner) {
    super(runner);
    }
    // 重写实现 processEvent 方法
    @Override
    public void processEvent(InnerEvent event) {
    super.processEvent(event);
    if (event == null) {
            return;
    }
    int eventId = event.eventId;
    long param = event.param;
        Object object = event.object;
    switch (eventId | param) {
            case CASE1:
                // 待执行的操作，由开发者定义
            break;
            case CASE2:
                // 将原先线程的 EventRunner 实例投递给新创建的线程
                if (object instanceof EventRunner) {
                EventRunner runner2 = (EventRunner)object;
                }
                // 将原先线程的 EventRunner 实例与新创建的线程的
EventHandler 绑定
                EventHandler myHandler2 = new EventHandler(runner2) {
                @Override
                public void processEvent(InnerEvent event) {
```

```
                //需要在原先线程执行的操作
                }
                };
                int eventId = 1;
                long param = 0;
                Object object = null;
                InnerEvent event2 = InnerEvent.get(eventId, param,
object);
                myHandler2.sendEvent(event2); // 投递事件到原先的线程
                break;
        default:
        break;
    }
    }
}
```

创建 EventRunner，以手动模式为例。

```
EventRunner runner1 = EventRunner.create(false);// create()的参数是 true
时，则为托管模式。

// 需要对 EventRunner 的实例进行校验，不是任何线程都可以通过 create 创建，例如：
当线程池已满时，不能再创建线程。

if (runner1 == null) {

    return;

}
```

创建 EventHandler 子类的实例。

```
MyEventHandler myHandler1 = new MyEventHandler(runner1);
```

获取 InnerEvent 事件。

```
// 获取事件实例，其属性 eventId，param，object 由开发者确定，代码中只是示例。
int eventId1 = 0;
long param = 0;
Object object = (Object) EventRunner.current();
InnerEvent event1 = InnerEvent.get(eventId1, param, object);
```

投递事件，在新线程上直接处理。

```
// 将与当前线程绑定的 EventRunner 投递到与 runner1 创建的新线程中
 myHandler.sendEvent(event1);
```

启动和停止 EventRunner，如果是托管模式，则不需要此步骤。

```
runner.run();
//待执行操作

runner.stop();// 停止 EventRunner
```

完整代码示例

- 非托管情况：

```
//全局：
EventRunner runnerA

//线程 A：
runnerA = EventRunner.create(false);
runnerA.run(); // run 之后一直循环卡在这里，所以需要新建一个线程 run

//线程 B：
```

```
//1.创建类继承 EventHandler
public class MyEventHandler extends EventHandler {
    public static int CODE_DOWNLOAD_FILE1;
    public static int CODE_DOWNLOAD_FILE2;
    public static int CODE_DOWNLOAD_FILE3;
    private MyEventHandler(EventRunner runner) {
        super(runner);
    }

    @Override
    public void processEvent(InnerEvent event) {
        super.processEvent(event);
        if (event == null) {
            return;
        }

        int eventId = event.eventId;
        if (STOP_EVENT_ID != eventId) {
            resultEventIdList.add(eventId);
        }

        switch (eventId) {
            case CODE_DOWNLOAD_FILE1: {
                ... // your process
                break;
            }
            case CODE_DOWNLOAD_FILE1: {
                ... // your process
```

```
                break;
            }
            case CODE_DOWNLOAD_FILE1: {
                ... // your process
                break;
            }
            default:
                break;
        }
    }
}

//2.创建 MyEventHandler 实例
MyEventHandler handler = new MyEventHandler(runnerA);

// 3.向线程 A 发送事件
handler.sendEvent(CODE_DOWNLOAD_FILE1);
handler.sendEvent(CODE_DOWNLOAD_FILE2);
handler.sendEvent(CODE_DOWNLOAD_FILE3);
......

// 4.runnerA 不再使用后，退出
runnerA.stop();
```

托管情况:

```
//1.创建 EventRunner A:
EventRunner runnerA = EventRunner.create("downloadRunner"); // 内部会新建一个线程
```

```
//2.创建类继承 EventHandler
public class MyEventHandler extends EventHandler {
    public static int CODE_DOWNLOAD_FILE1;
    public static int CODE_DOWNLOAD_FILE2;
    public static int CODE_DOWNLOAD_FILE3;
    private MyEventHandler(EventRunner runner) {
        super(runner);
    }

    @Override
    public void processEvent(InnerEvent event) {
        super.processEvent(event);
        if (event == null) {
            return;
        }

        int eventId = event.eventId;
        if (STOP_EVENT_ID != eventId) {
            resultEventIdList.add(eventId);
        }

        switch (eventId) {
            case CODE_DOWNLOAD_FILE1: {
                ... // your process
                break;
            }
            case CODE_DOWNLOAD_FILE1: {
```

```
                ... // your process
                break;
            }
            case CODE_DOWNLOAD_FILE1: {
                ... // your process
                break;
            }
            default:
                break;
        }
    }
}

//3.创建 MyEventHandler 实例
MyEventHandler handler = new MyEventHandler(runnerA);

//4.向线程 A 发送事件
handler.sendEvent(CODE_DOWNLOAD_FILE1);
handler.sendEvent(CODE_DOWNLOAD_FILE2);
handler.sendEvent(CODE_DOWNLOAD_FILE3);
......

//5.runnerA 没有任何对象引入时，线程会自动回收
runnerA = null;
```

UI

Java UI 框架

概述

应用的 Ability 在屏幕上将显示一个用户界面，该界面用来显示所有可被用户查看和交互的内容。应用中所有的用户界面元素都是由 Component 和 ComponentContainer 对象构成。Component 是绘制在屏幕上的一个对象，用户能与之交互。ComponentContainer 是一个用于容纳其他 Component 和 ComponentContainer 对象的容器。

Java UI 框架提供了一部分 Component 和 ComponentContainer 的具体子类，即创建用户界面（UI）的各类组件，包括一些常用的组件（比如：文本、按钮、图片、列表等）和常用的布局（比如：DirectionalLayout 和 DependentLayout）。用户可通过组件进行交互操作，并获得响应。所有的 UI 操作都应该在主线程进行设置。

组件和布局

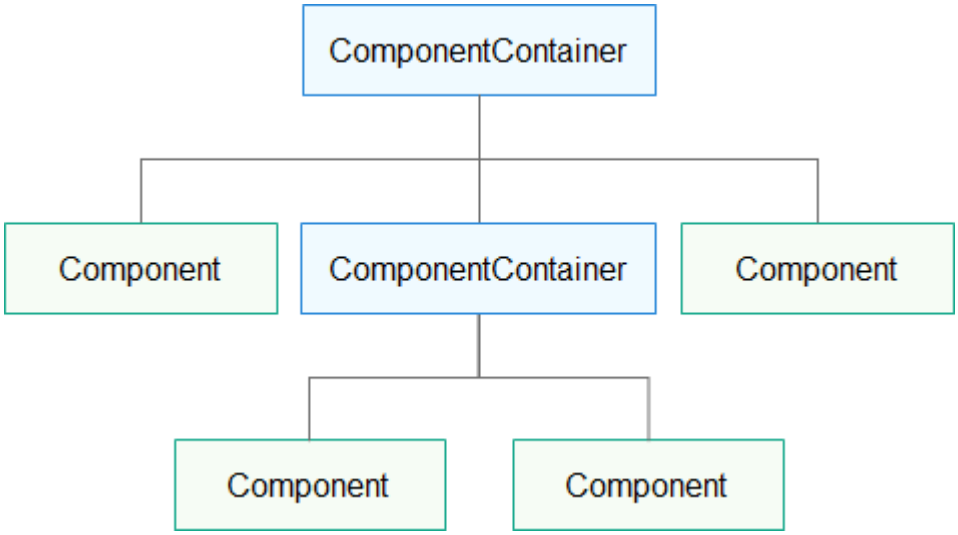
用户界面元素统称为组件，组件根据一定的层级结构进行组合形成布局。组件在未被添加到布局中时，既无法显示也无法交互，因此一个用户界面至少包含一个布局。在 UI 框架中，具体的布局类通常以 XXLayout 命名，完整的用户界面是一个布局，用户界面中的一部分也可以是一个布局。布局中容纳 Component 与 ComponentContainer 对象。

Component 和 ComponentContainer

- Component：提供内容显示，是界面中所有组件的基类，开发者可以给 Component 设置事件处理回调来创建一个可交互的组件。Java UI 框架提供了一些常用的界面元素，也可称之为组件，组件一般直接继承 Component 或它的子类，如 Text、Image 等。
- ComponentContainer：作为容器容纳 Component 或 ComponentContainer 对象，并对它们进行布局。Java UI 框架提供了一些标准布局功能的容器，它们继承自

ComponentContainer，一般以“Layout”结尾，如 DirectionalLayout、DependentLayout 等。

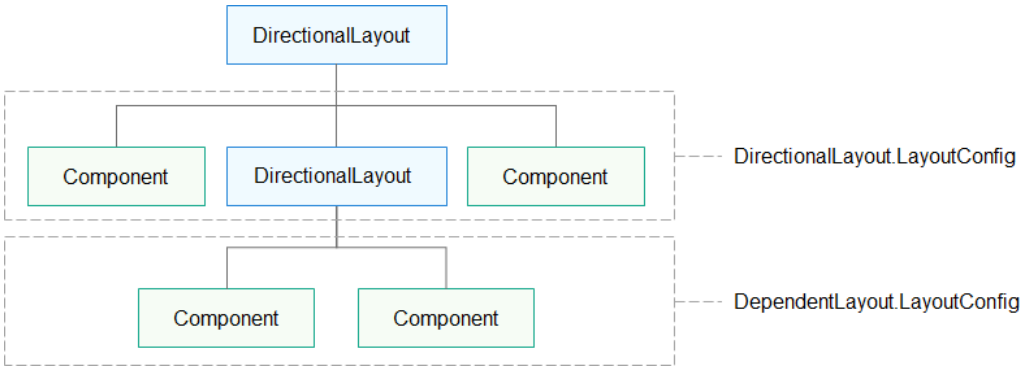
图 1　Component 结构



LayoutConfig

每种布局都根据自身特点提供 LayoutConfig 供子 Component 设定布局属性和参数，通过指定布局属性可以对子 Component 在布局中的显示效果进行约束。例如：“width”、“height”是最基本的布局属性，它们指定了组件的大小。

图 2　LayoutConfig



组件树

布局把 Component 和 ComponentContainer 以树状的层级结构进行组织，这样的一个布局就称为组件树。组件树的特点是仅有一个根组件，其他组件有且仅有一个父节点，组件之间的关系受到父节点的规则约束。

组件与布局开发指导

开发说明

HarmonyOS 提供了 Ability 和 AbilitySlice 两个基础类。有界面的 Ability 绑定了系统的 Window 进行 UI 展示，且具有生命周期。AbilitySlice 主要用于承载 Ability 的具体逻辑实现和界面 UI，是应用显示、运行和跳转的最小单元。AbilitySlice 通过 setUIContent()为界面设置布局。

表 1 AbilitySlice 的 UI 接口

接口声明	接口描述
setUIContent(ComponentContainer root)	设置界面入口，root 为界面组件树根节点。

组件需要进行组合，并添加到界面的布局中。在 Java UI 框架中，提供了两种编写布局的方式：

- 在代码中创建布局：用代码创建 Component 和 ComponentContainer 对象，为这些对象设置合适的布局参数和属性值，并将 Component 添加到 ComponentContainer 中，从而创建出完整界面。
- 在 XML 中声明 UI 布局：按层级结构来描述 Component 和 ComponentContainer 的关系，给组件节点设定合适的布局参数和属性值，代码中可直接加载生成此布局。

这两种方式创建出的布局没有本质差别，在 XML 中声明布局，在加载后同样可在代码中对该布局进行修改。

组件分类

根据组件的功能，可以将组件分为布局类、显示类、交互类三类：

表 2 组件分类

组件类别	组件名称	功能描述
布局类	PositionLayout、DirectionalLayout、StackLayout、DependentLayout、TableLayout、AdaptiveBoxLayout	提供了不同布局规范的组件容器，例如以单一方向排列的 DirectionalLayout、以相对位置排列的 DependentLayout、以确切位置排列的 PositionLayout 等。
显示类	Text、Image、Clock、TickTimer、ProgressBar	提供了单纯的内容显示，例如用于文本显示的 Text，用于图像显示的 Image 等。
交互类	TextField、Button、Checkbox、RadioButton/RadioContainer、Switch、ToggleButton、Slider、Rating、ScrollView、TabList、ListContainer、PageSlider、PageFlipper、PageSliderIndicator、Picker、TimePicker、DatePicker、SurfaceProvider、ComponentProvider	提供了具体场景下与用户交互响应的功能，例如 Button 提供了点击响应功能，Slider 提供了进度选择功能等。

框架提供的组件使应用界面开发更加便利，这些组件的具体功能说明及属性设置详见 API 参考。

组件与布局开发指导

开发说明

HarmonyOS 提供了 Ability 和 AbilitySlice 两个基础类。有界面的 Ability 绑定了系统的 Window 进行 UI 展示，且具有生命周期。AbilitySlice 主要用于承载 Ability 的具体逻辑实现和界面 UI，是应用显示、运行和跳转的最小单元。AbilitySlice 通过 setUIContent()为界面设置布局。

表 1 AbilitySlice 的 UI 接口

接口声明	接口描述
setUIContent(ComponentContainer root)	设置界面入口，root 为界面组件树根节点。

组件需要进行组合，并添加到界面的布局中。在 Java UI 框架中，提供了两种编写布局的方式：

- 在代码中创建布局：用代码创建 Component 和 ComponentContainer 对象，为这些对象设置合适的布局参数和属性值，并将 Component 添加到 ComponentContainer 中，从而创建出完整界面。
- 在 XML 中声明 UI 布局：按层级结构来描述 Component 和 ComponentContainer 的关系，给组件节点设定合适的布局参数和属性值，代码中可直接加载生成此布局。

这两种方式创建出的布局没有本质差别，在 XML 中声明布局，在加载后同样可在代码中对该布局进行修改。

组件分类

根据组件的功能，可以将组件分为布局类、显示类、交互类三类：

表 2 组件分类

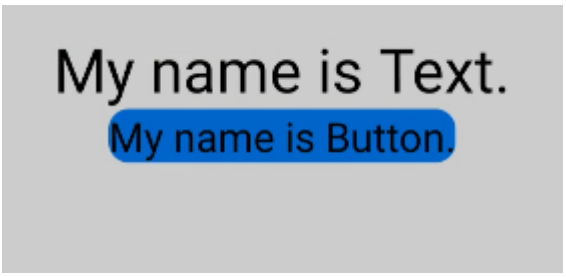
组件类别	组件名称	功能描述
布局类	PositionLayout、DirectionalLayout、StackLayout、DependentLayout、TableLayout、AdaptiveBoxLayout	提供了不同布局规范的组件容器，例如以单一方向排列的 DirectionalLayout、以相对位置排列的 DependentLayout、以确切位置排列的 PositionLayout 等。
显示类	Text、Image、Clock、TickTimer、ProgressBar	提供了单纯的内容显示，例如用于文本显示的 Text，用于图像显示的 Image 等。
交互类	TextField、Button、Checkbox、RadioButton/RadioContainer、Switch、ToggleButton、Slider、Rating、ScrollView、TabList、ListContainer、PageSlider、PageFlipper、PageSliderIndicator、Picker、TimePicker、DatePicker、SurfaceProvider、ComponentProvider	提供了具体场景下与用户交互响应的功能，例如 Button 提供了点击响应功能，Slider 提供了进度选择功能等。

框架提供的组件使应用界面开发更加便利，这些组件的具体功能说明及属性设置详见 API 参考。

代码创建布局

开发如下图所示界面，需要添加一个 Text 组件和 Button 组件。由于两个组件从上到下依次居中排列，可以选择使用竖向的 DirectionalLayout 布局来放置组件。

图 1　开发样例图



代码创建布局需要分别创建组件和布局，并将它们进行组织关联。

创建组件

1. 声明组件

```
Button button = new Button(context); // 参数 context 表示 AbilitySlice 的 Context 对象
```

设置组件大小

```
button.setWidth(ComponentContainer.LayoutConfig.MATCH_CONTENT);

button.setHeight(ComponentContainer.LayoutConfig.MATCH_CONTENT);
```

设置组件属性及 ID

```
button.setText("My name is Button.");

button.setTextSize(25);

button.setId(ID_BUTTON);
```

创建布局并使用

1. 声明布局

```
DirectionalLayout directionalLayout = new DirectionalLayout(context);
```

设置布局大小

```
directionalLayout.setWidth(ComponentContainer.LayoutConfig.MATCH_PARENT
);

directionalLayout.setHeight(ComponentContainer.LayoutConfig.MATCH_PAREN
T);
```

设置布局属性及 ID

```
directionalLayout.setOrientation(Component.VERTICAL);
```

将组件添加到布局中（视布局需要对组件设置布局属性进行约束）

```
directionalLayout.addComponent(button);
```

将布局添加到组件树中

```
setUIContent(directionalLayout);
```

完整代码示例：

```
@Override

public void onStart(Intent intent) {

    super.onStart(intent);

    // 步骤 1 声明布局

    DirectionalLayout directionalLayout = new DirectionalLayout(context);

    // 步骤 2 设置布局大小
```

```
directionalLayout.setWidth(ComponentContainer.LayoutConfig.MATCH_PARENT
);


directionalLayout.setHeight(ComponentContainer.LayoutConfig.MATCH_PAREN
T);
    // 步骤 3 设置布局属性及 ID（ID 视需要设置即可）
    directionalLayout.setOrientation(Component.VERTICAL);
    directionalLayout.setPadding(32, 32, 32, 32);


    Text text = new Text(context);
    text.setText("My name is Text.");
    text.setTextSize(50);
    text.setId(100);
    // 步骤 4.1 为组件添加对应布局的布局属性
    DirectionalLayout.LayoutConfig layoutConfig = new
DirectionalLayout.LayoutConfig(LayoutConfig.MATCH_CONTENT,
        LayoutConfig.MATCH_CONTENT);
    layoutConfig.alignment = LayoutAlignment.HORIZONTAL_CENTER;
    text.setLayoutConfig(layoutConfig);


    // 步骤 4.2 将 Text 添加到布局中
    directionalLayout.addComponent(text);


    // 类似的添加一个 Button
    Button button = new Button(context);
    layoutConfig.setMargins(0, 50, 0, 0);
    button.setLayoutConfig(layoutConfig);
    button.setText("My name is Button.");
    button.setTextSize(50);
```

```
    button.setId(100);
    ShapeElement background = new ShapeElement();
    background.setRgbColor(new RgbColor(0, 125, 255));
    background.setCornerRadius(25);
    button.setBackground(background);
    button.setPadding(10, 10, 10, 10);
    button.setClickedListener(new Component.ClickedListener() {
        @Override
        // 在组件中增加对点击事件的检测
        public void onClick(Component Component) {
            // 此处添加按钮被点击需要执行的操作
        }
    });
    directionalLayout.addComponent(button);

    // 步骤 5 将布局作为根布局添加到视图树中
    super.setUIContent(directionalLayout);
}
```

根据以上步骤创建组件和布局后的界面显示效果如图 1 所示。其中，代码示例中为组件设置了一个按键回调，在按键被按下后，应用会执行自定义的操作。

在代码示例中，可以看到设置组件大小的方法有两种：

- 通过 setWidth/setHeight 直接设置宽高。
- 通过 setLayoutConfig 方法设置布局属性来设定宽高。

这两种方法的区别是后者还可以增加更多的布局属性设置，例如：使用“alignment”设置水平居中的约束。另外，这两种方法设置的宽高以最后设置的作为最终结果。它们的取值一致，可以是以下取值：

- 具体以像素为单位的数值。
- MATCH_PARENT：表示组件大小将扩展为父组件允许的最大值，它将占据父组件方向上的剩余大小。
- MATCH_CONTENT：表示组件大小与它内容占据的大小范围相适应。

XML 创建布局

XML 声明布局的方式更加简便直观。每一个 Component 和 ComponentContainer 对象大部分属性都支持在 XML 中进行设置，它们都有各自的 XML 属性列表。某些属性仅适用于特定的组件，例如：只有 Text 支持“text_color”属性，但不支持该属性的组件如果添加了该属性，该属性则会被忽略。具有继承关系的组件子类将继承父类的属性列表，Component 作为组件的基类，拥有各个组件常用的属性，比如：ID、布局参数等。

ID

```
ohos:id="$+id:text"
```

在 XML 中使用此格式声明一个对开发者友好的 ID，它会在编译过程中转换成一个常量。尤其在 DependentLayout 布局中，组件之间需要描述相对位置关系，描述时要通过 ID 来指定对应组件。

布局中的组件通常要设置独立的 ID，以便在程序中查找该组件。如果布局中有不同组件设置了相同的 ID，在通过 ID 查找组件时会返回查找到的第一个组件，因此尽量保证在所要查找的布局中为组件设置独立的 ID 值，避免出现与预期不符合的问题。

布局参数

```
ohos:width="20vp"
ohos:height="10vp"
```

与代码中设置组件的宽度和高度类似，在 XML 中它们的取值可以是：

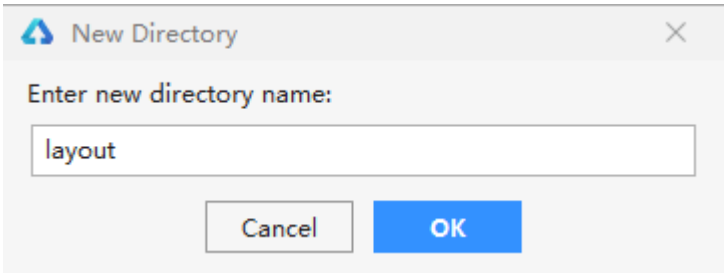
- 具体的数值：10（以像素为单位）、10vp（以屏幕相对像素为单位）。
- MATCH_PARENT：表示组件大小将扩展为父组件允许的最大值，它将占据父组件方向上的剩余大小，在 XML 中用“match_parent”表示。
- MATCH_CONTENT：表示组件大小与它的内容占据的大小范围相适应，在 XML 中用数值“match_content”表示。

更多的组件属性列表可参考组件的 XML 属性文档。

创建 XML 布局文件

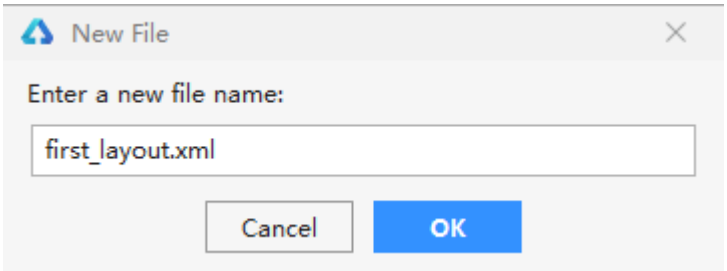
1. 在 DevEco Studio 的“Project”窗口，打开“entry > src > main > resources > base”，右键点击“base”文件夹，选择“New > Directory”，命名为“layout”。

图 1 设置 Directory 名称



2. 右键点击“layout”文件夹，选择“New > File”，命名为“first_layout.xml”。

图 2 设置 File 名称



修改 XML 布局文件

打开新创建的 first_layout.xml 布局文件，修改其中的内容，对布局和组件的属性和层级进行描述。

```
<?xml version="1.0" encoding="utf-8"?>
<DirectionalLayout
    xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:width="match_parent"
    ohos:height="match_parent"
    ohos:orientation="vertical"
    ohos:padding="32">
    <Text
        ohos:id="$+id:text"
        ohos:width="match_content"
        ohos:height="match_content"
        ohos:layout_alignment="horizontal_center"
        ohos:text="My name is Text."
        ohos:text_size="25vp"/>
    <Button
        ohos:id="$+id:button"
        ohos:width="match_content"
        ohos:height="match_content"
        ohos:layout_alignment="horizontal_center"
        ohos:text="My name is Button."
        ohos:text_size="50"/>
</DirectionalLayout>
```

加载 XML 布局

在代码中需要加载 XML 布局，并添加为根布局或作为其他布局的子 Component。

```
@Override
public void onStart(Intent intent) {
    super.onStart(intent);
    // 加载 XML 布局作为根布局
    super.setUIContent(ResourceTable.Layout_first_layout);
    // 查找布局中组件
    Button button = (Button)
findComponentById(ResourceTable.Id_button);
    if (button != null) {
        // 设置组件的属性
        ShapeElement background = new ShapeElement();
        background.setRgbColor(new RgbColor(0,125,255));
        background.setCornerRadius(25);
        button.setBackground(background);

        button.setClickedListener(new Component.ClickedListener() {
            @Override
            // 在组件中增加对点击事件的检测
            public void onClick(Component Component) {
                // 此处添加按钮被点击需要执行的操作
            }
        });
    }
}
```

常用组件开发指导

Text

文本（Text）是用来显示字符串的组件，在界面上显示为一块文本区域。Text 作为一个基本组件，有很多扩展，常见的有按钮组件 Button，文本编辑组件 TextField。

使用 Text

创建 Text

```
<Text
    ohos:id="$+id:text"
    ohos:width="match_content"
    ohos:height="match_content"
    ohos:text="Text"
    ohos:background_element="$graphic:color_gray_element"/>
```

color_gray_element.xml：

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:shape="rectangle">
    <solid
        ohos:color="#ff888888"/>
</shape>
```

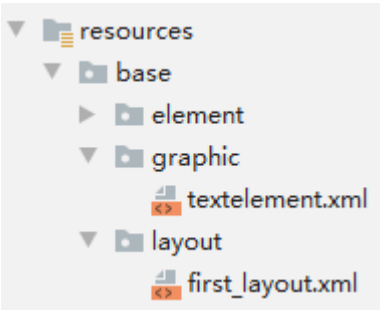
图 1 创建一个 Text

- 设置背景

常用的背景如常见的文本背景、按钮背景，可以采用 XML 格式放置在 graphic 目录下。

在“Project”窗口，打开“entry > src > main > resources > base”，右键点击“base”文件夹，选择“New > Directory”，命名为“graphic”。右键点击“graphic”文件夹，选择“New > File”，命名为“textelement.xml”。

图 2 创建 textelement.xml 文件后的 resources 目录结构



在 textelement.xml 中定义文本的背景:

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<shape xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:shape="rectangle">
    <corners
        ohos:radius="20"/>
    <solid
        ohos:color="#ff888888"/>
</shape>
```

在 first_layout.xml 中引用上面定义的文本背景：

```
<Text
    ohos:id="$+id:text"
    ohos:width="match_content"
    ohos:height="match_content"
    ohos:text="Text"
    ohos:background_element="$graphic:textelement"/>
```

设置字体大小和颜色

```
<Text
    ohos:id="$+id:text"
    ohos:width="match_content"
    ohos:height="match_content"
    ohos:text="Text"
    ohos:text_size="28fp"
    ohos:text_color="blue"
    ohos:left_margin="15vp"
    ohos:bottom_margin="15vp"
    ohos:right_padding="15vp"
```

```
    ohos:left_padding="15vp"
    ohos:background_element="$graphic:textelement"/>
```

图 3 设置字体大小和颜色的效果

- 设置字体风格和字重

```
<Text
    ohos:id="$+id:text"
    ohos:width="match_content"
    ohos:height="match_content"
    ohos:text="Text"
    ohos:text_size="28fp"
    ohos:text_color="blue"
    ohos:italic="true"
    ohos:text_weight="700"
    ohos:text_font="serif"
    ohos:left_margin="15vp"
    ohos:bottom_margin="15vp"
    ohos:right_padding="15vp"
    ohos:left_padding="15vp"
    ohos:background_element="$graphic:textelement"/>
```

图 4 设置字体风格和字重的效果

- 设置文本对齐方式

```
<Text
    ohos:id="$+id:text"
    ohos:width="300vp"
    ohos:height="100vp"
    ohos:text="Text"
    ohos:text_size="28fp"
    ohos:text_color="blue"
    ohos:italic="true"
    ohos:text_weight="700"
    ohos:text_font="serif"
    ohos:left_margin="15vp"
    ohos:bottom_margin="15vp"
    ohos:right_padding="15vp"
    ohos:left_padding="15vp"
    ohos:text_alignment="horizontal_center|bottom"
    ohos:background_element="$graphic:textelement"/>
```

图 5 设置文本对齐方式的效果

- 设置文本换行和最大显示行数

```
<Text
    ohos:id="$+id:text"
    ohos:width="75vp"
    ohos:height="match_content"
    ohos:text="TextText"
    ohos:text_size="28fp"
    ohos:text_color="blue"
    ohos:italic="true"
    ohos:text_weight="700"
    ohos:text_font="serif"
    ohos:multiple_lines="true"
    ohos:max_text_lines="2"
    ohos:background_element="$graphic:textelement"/>
```

图 6 设置文本换行和最大显示行数的效果

自动调节字体大小

Text 对象支持根据文本长度自动调整文本的字体大小和换行。

1. 设置自动换行、最大显示行数和自动调节字体大小。

```
<Text
    ohos:id="$+id:text1"
    ohos:width="90vp"
    ohos:height="match_content"
    ohos:min_height="30vp"
    ohos:text="T"
    ohos:text_color="blue"
    ohos:italic="true"
    ohos:text_weight="700"
    ohos:text_font="serif"
    ohos:multiple_lines="true"
    ohos:max_text_lines="1"
```

```
    ohos:auto_font_size="true"
    ohos:right_padding="8vp"
    ohos:left_padding="8vp"
    ohos:background_element="$graphic:textelement"/>
```

通过 setAutoFontSizeRule 设置自动调整规则，三个入参分别是最小的字体大小、最大的字体大小、每次调整文本字体大小的步长。

```
// 设置自动调整规则
text.setAutoFontSizeRule(30, 100, 1);
// 设置点击一次增多一个"T"
text.setClickedListener(new Component.ClickedListener() {
    @Override
    public void onClick(Component Component) {
        text.setText(text.getText() + "T");
    }
});
```

图 7 自动调节字体大小

跑马灯效果

当文本过长时，可以设置跑马灯效果，实现文本滚动显示。前提是文本换行关闭且最大显示行数为 1，默认情况下即可满足前提要求。

```
<Text
    ohos:id="$+id:text"
    ohos:width="75vp"
    ohos:height="match_content"
    ohos:text="TextText"
    ohos:text_size="28fp"
    ohos:text_color="blue"
    ohos:italic="true"
    ohos:text_weight="700"
    ohos:text_font="serif"
    ohos:background_element="$graphic:textelement"/>
```

```
// 跑马灯效果
text.setTruncationMode(Text.TruncationMode.AUTO_SCROLLING);
// 启动跑马灯效果
text.startAutoScrolling();
```

图 8 跑马灯效果

场景示例

利用文本组件实现一个标题栏和详细内容的界面。

图 9 界面效果

源码示例：

```
<?xml version="1.0" encoding="utf-8"?>
<DependentLayout
    xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:width="match_parent"
    ohos:height="match_content"
    ohos:background_element="$graphic:color_light_gray_element">
```

```
    <Text
        ohos:id="$+id:text1"
        ohos:width="match_parent"
        ohos:height="match_content"
        ohos:text_size="25fp"
        ohos:top_margin="15vp"
        ohos:left_margin="15vp"
        ohos:right_margin="15vp"
        ohos:background_element="$graphic:textelement"
        ohos:text="Title"
        ohos:text_weight="1000"
        ohos:text_alignment="horizontal_center"/>
    <Text
        ohos:id="$+id:text3"
        ohos:width="match_parent"
        ohos:height="120vp"
        ohos:text_size="25fp"
        ohos:background_element="$graphic:textelement"
        ohos:text="Content"
        ohos:top_margin="15vp"
        ohos:left_margin="15vp"
        ohos:right_margin="15vp"
        ohos:bottom_margin="15vp"
        ohos:text_alignment="center"
        ohos:below="$id:text1"
        ohos:text_font="serif"/>
    <Button
        ohos:id="$+id:button1"
```

```
        ohos:width="75vp"
        ohos:height="match_content"
        ohos:text_size="15fp"
        ohos:background_element="$graphic:textelement"
        ohos:text="Previous"
        ohos:right_margin="15vp"
        ohos:bottom_margin="15vp"
        ohos:left_padding="5vp"
        ohos:right_padding="5vp"
        ohos:below="$id:text3"
        ohos:left_of="$id:button2"
        ohos:text_font="serif"/>
    <Button
        ohos:id="$+id:button2"
        ohos:width="75vp"
        ohos:height="match_content"
        ohos:text_size="15fp"
        ohos:background_element="$graphic:textelement"
        ohos:text="Next"
        ohos:right_margin="15vp"
        ohos:bottom_margin="15vp"
        ohos:left_padding="5vp"
        ohos:right_padding="5vp"
        ohos:align_parent_end="true"
        ohos:below="$id:text3"
        ohos:text_font="serif"/>
</DependentLayout>
```

color_light_gray_element.xml：

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

textelement.xml：

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:shape="rectangle">
    <corners
        ohos:radius="20"/>
    <solid
        ohos:color="#ff888888"/>
</shape>
```

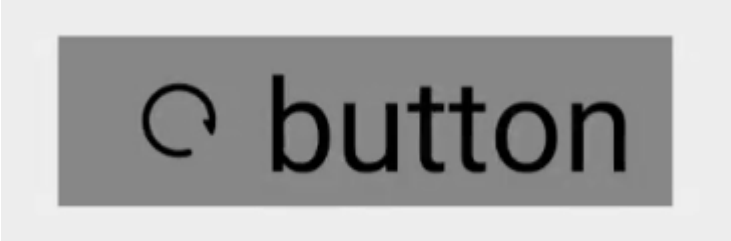
Button

按钮（Button）是一种常见的组件，点击可以触发对应的操作，通常由文本或图标组成，也可以由图标和文本共同组成。

图 1　文本按钮

图 2　图标按钮

图 3　图标和文本共同组成的按钮



创建 Button

使用 Button 组件，可以生成形状、颜色丰富的按钮。

```
<Button
    ohos:id="$+id:button_sample"
    ohos:width="match_content"
    ohos:height="match_content"
    ohos:text_size="27fp"
    ohos:text="button"
    ohos:background_element="$graphic:button_element"
```

```
    ohos:left_margin="15vp"
    ohos:bottom_margin="15vp"
    ohos:right_padding="8vp"
    ohos:left_padding="8vp"
    ohos:element_left="$graphic:ic_btn_reload"
/>
```

button_element.xml：

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:shape="rectangle">
    <corners
        ohos:radius="10"/>
    <solid
        ohos:color="#FF007DFF"/>
</shape>
```

响应点击事件

按钮的重要作用是当用户单击按钮时，会执行相应的操作或者界面出现相应的变化。实际上用户点击按钮时，Button 对象将收到一个点击事件。 开发者可以自

定义响应点击事件的方法。例如，通过创建一个 Component.ClickedListener 对象，然后通过调用 setClickedListener 将其分配给按钮。

```
//从定义的 xml 中获取 Button 对象
Button button = (Button)
rootLayout.findComponentById(ResourceTable.Id_button_sample);
// 为按钮设置点击事件回调
button.setClickedListener(new Component.ClickedListener() {
    public void onClick(Component v) {
        // 此处添加点击按钮后的事件处理逻辑
    }
});
```

不同类型的按钮

按照按钮的形状，按钮可以分为：普通按钮，椭圆按钮，胶囊按钮，圆形按钮等。

- **普通按钮**

普通按钮和其他按钮的区别在于不需要设置任何形状，只设置文本和背景颜色即可，例如：

```
<Button
    ohos:width="150vp"
    ohos:height="50vp"
```

```
    ohos:text_size="27fp"
    ohos:text="button"
    ohos:background_element="$graphic:color_blue_element"
    ohos:left_margin="15vp"
    ohos:bottom_margin="15vp"
    ohos:right_padding="8vp"
    ohos:left_padding="8vp"
/>
```

color_blue_element.xml：

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:shape="rectangle">
    <solid
        ohos:color="#FF007DFF"/>
</shape>
```

椭圆按钮

椭圆按钮是通过设置 background_element 的来实现的，

background_element 的 shape 设置为椭圆（oval），例如：

```
<Button

    ohos:width="150vp"

    ohos:height="50vp"

    ohos:text_size="27fp"

    ohos:text="button"

    ohos:background_element="$graphic:oval_button_element"

    ohos:left_margin="15vp"

    ohos:bottom_margin="15vp"

    ohos:right_padding="8vp"

    ohos:left_padding="8vp"

    ohos:element_left="$graphic:ic_btn_reload"

/>
```

oval_button_element.xml:

```
<?xml version="1.0" encoding="utf-8"?>

<shape xmlns:ohos="http://schemas.huawei.com/res/ohos"

    ohos:shape="oval">

    <solid

        ohos:color="#FF007DFF"/>

</shape>
```

胶囊按钮

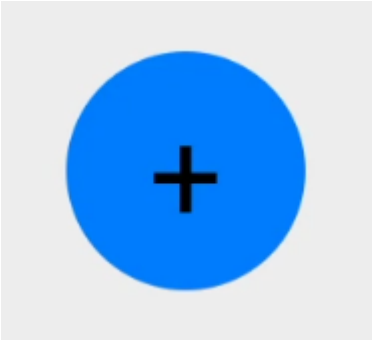
胶囊按钮是一种常见的按钮，设置按钮背景时将背景设置为矩形形状，并且设置 ShapeElement 的 radius 的半径，例如：

```
<Button
    ohos:id="$+id:button"
    ohos:width="match_content"
    ohos:height="match_content"
    ohos:text_size="27fp"
    ohos:text="button"
    ohos:background_element="$graphic:capsule_button_element"
    ohos:left_margin="15vp"
    ohos:bottom_margin="15vp"
    ohos:right_padding="15vp"
    ohos:left_padding="15vp"
/>
```

capsule_button_element.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:shape="rectangle">
    <corners
        ohos:radius="100"/>
    <solid
        ohos:color="#FF007DFF"/>
</shape>
```

圆形按钮



圆形按钮和椭圆按钮的区别在于组件本身的宽度和高度需要相同，例如：

```
<Button
    ohos:id="$+id:button4"
    ohos:width="50vp"
    ohos:height="50vp"
    ohos:text_size="27fp"
    ohos:background_element="$graphic:circle_button_element"
    ohos:text="+"
    ohos:left_margin="15vp"
    ohos:bottom_margin="15vp"
    ohos:right_padding="15vp"
    ohos:left_padding="15vp"
/>
```

circle_button_element.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:shape="oval">
    <solid
        ohos:color="#FF007DFF"/>
</shape>
```

场景示例

利用圆形按钮，胶囊按钮，文本组件可以绘制出如下拨号盘的 UI 界面。

图 4 界面效果

源码示例：

```
<?xml version="1.0" encoding="utf-8"?>
<DirectionalLayout
    xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:width="match_parent"
    ohos:height="match_parent"
    ohos:background_element="$graphic:color_light_gray_element"
    ohos:orientation="vertical">
    <Text
        ohos:width="match_content"
        ohos:height="match_content"
        ohos:text_size="20fp"
```

```
        ohos:text="0123456789"
        ohos:background_element="$graphic:green_text_element"
        ohos:text_alignment="center"
        ohos:layout_alignment="horizontal_center"
    />
    <DirectionalLayout
        ohos:width="match_parent"
        ohos:height="match_content"
        ohos:alignment="horizontal_center"
        ohos:orientation="horizontal"
        ohos:top_margin="5vp"
        ohos:bottom_margin="5vp">
        <Button
            ohos:width="40vp"
            ohos:height="40vp"
            ohos:text_size="15fp"

ohos:background_element="$graphic:green_circle_button_element"
            ohos:text="1"
            ohos:text_alignment="center"
        />
        <Button
            ohos:width="40vp"
            ohos:height="40vp"
            ohos:text_size="15fp"

ohos:background_element="$graphic:green_circle_button_element"
            ohos:text="2"
            ohos:left_margin="5vp"
```

```
            ohos:right_margin="5vp"
            ohos:text_alignment="center"
        />
        <Button
            ohos:width="40vp"
            ohos:height="40vp"
            ohos:text_size="15fp"
ohos:background_element="$graphic:green_circle_button_element"
            ohos:text="3"
            ohos:text_alignment="center"
        />
    </DirectionalLayout>
    <DirectionalLayout
        ohos:width="match_parent"
        ohos:height="match_content"
        ohos:alignment="horizontal_center"
        ohos:orientation="horizontal"
        ohos:bottom_margin="5vp">
        <Button
            ohos:width="40vp"
            ohos:height="40vp"
            ohos:text_size="15fp"
ohos:background_element="$graphic:green_circle_button_element"
            ohos:text="4"
            ohos:text_alignment="center"
        />
        <Button
```

```
            ohos:width="40vp"
            ohos:height="40vp"
            ohos:text_size="15fp"
            ohos:left_margin="5vp"
            ohos:right_margin="5vp"

ohos:background_element="$graphic:green_circle_button_element"
            ohos:text="5"
            ohos:text_alignment="center"
        />
        <Button
            ohos:width="40vp"
            ohos:height="40vp"
            ohos:text_size="15fp"

ohos:background_element="$graphic:green_circle_button_element"
            ohos:text="6"
            ohos:text_alignment="center"
        />
    </DirectionalLayout>
    <DirectionalLayout
        ohos:width="match_parent"
        ohos:height="match_content"
        ohos:alignment="horizontal_center"
        ohos:orientation="horizontal"
        ohos:bottom_margin="5vp">
        <Button
            ohos:width="40vp"
            ohos:height="40vp"
```

```
            ohos:text_size="15fp"

ohos:background_element="$graphic:green_circle_button_element"
            ohos:text="7"
            ohos:text_alignment="center"
        />
        <Button
            ohos:width="40vp"
            ohos:height="40vp"
            ohos:text_size="15fp"
            ohos:left_margin="5vp"
            ohos:right_margin="5vp"

ohos:background_element="$graphic:green_circle_button_element"
            ohos:text="8"
            ohos:text_alignment="center"
        />
        <Button
            ohos:width="40vp"
            ohos:height="40vp"
            ohos:text_size="15fp"

ohos:background_element="$graphic:green_circle_button_element"
            ohos:text="9"
            ohos:text_alignment="center"
        />
    </DirectionalLayout>
    <DirectionalLayout
        ohos:width="match_parent"
```

```
        ohos:height="match_content"
        ohos:alignment="horizontal_center"
        ohos:orientation="horizontal"
        ohos:bottom_margin="5vp">
        <Button
            ohos:width="40vp"
            ohos:height="40vp"
            ohos:text_size="15fp"
ohos:background_element="$graphic:green_circle_button_element"
            ohos:text="*"
            ohos:text_alignment="center"
        />
        <Button
            ohos:width="40vp"
            ohos:height="40vp"
            ohos:text_size="15fp"
            ohos:left_margin="5vp"
            ohos:right_margin="5vp"
ohos:background_element="$graphic:green_circle_button_element"
            ohos:text="0"
            ohos:text_alignment="center"
        />
        <Button
            ohos:width="40vp"
            ohos:height="40vp"
            ohos:text_size="15fp"
```

```
ohos:background_element="$graphic:green_circle_button_element"
            ohos:text="#"
            ohos:text_alignment="center"
        />
    </DirectionalLayout>
    <Button
        ohos:width="match_content"
        ohos:height="match_content"
        ohos:text_size="15fp"
        ohos:text="CALL"
        ohos:background_element="$graphic:green_capsule_button_element"
        ohos:bottom_margin="5vp"
        ohos:text_alignment="center"
        ohos:layout_alignment="horizontal_center"
        ohos:left_padding="10vp"
        ohos:right_padding="10vp"
        ohos:top_padding="2vp"
        ohos:bottom_padding="2vp"
    />
</DirectionalLayout>
```

color_light_gray_element.xml：

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

```
        ohos:color="#ffeeeeee"/>
</shape>
```

green_text_element.xml：

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:shape="rectangle">
    <corners
        ohos:radius="20"/>
    <stroke
        ohos:width="2"
        ohos:color="#ff008B00"/>
    <solid
        ohos:color="#ffeeeeee"/>
</shape>
```

green_circle_button_element.xml：

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:shape="oval">
    <stroke
        ohos:width="5"
        ohos:color="#ff008B00"/>
    <solid
        ohos:color="#ffeeeeee"/>
</shape>
```

green_capsule_button_element.xml：

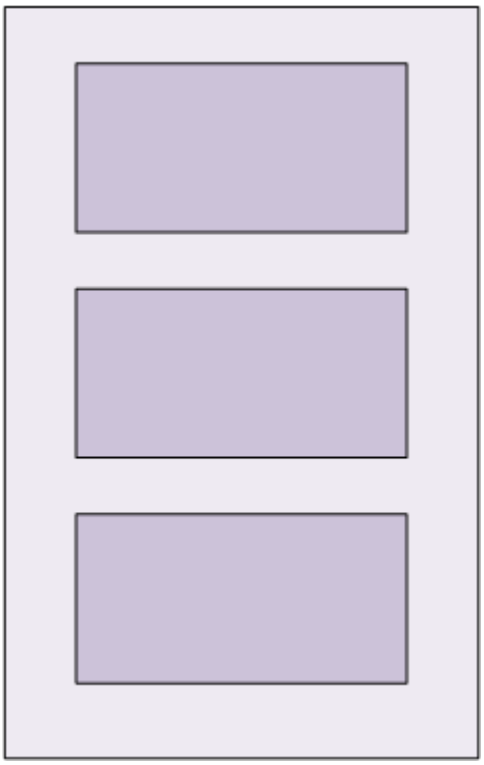
```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:shape="rectangle">
    <corners
        ohos:radius="100"/>
    <solid
        ohos:color="#ff008B00"/>
</shape>
```

常用布局开发指导

DirectionalLayout

DirectionalLayout 是 Java UI 中的一种重要组件布局，用于将一组组件(Component)按照水平或者垂直方向排布，能够方便地对齐布局内的组件。该布局和其他布局的组合，可以实现更加丰富的布局方式。

图 1　DirectionalLayout 示意图



排列方式

DirectionalLayout 的排列方向（orientation）分为水平（horizontal）或者垂直（vertical）方向。使用 orientation 设置布局内组件的排列方式，默认为垂直排列。

- **垂直排列**

垂直方向排列三个按钮，效果如下：

图 2　三个垂直排列的按钮


```
<?xml version="1.0" encoding="utf-8"?>
<DirectionalLayout
    xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:width="match_parent"
    ohos:height="match_content"
    ohos:orientation="vertical">
    <Button
        ohos:width="33vp"
        ohos:height="20vp"
        ohos:bottom_margin="3vp"
        ohos:left_margin="13vp"
        ohos:background_element="$graphic:color_cyan_element"
        ohos:text="Button 1"/>
    <Button
        ohos:width="33vp"
        ohos:height="20vp"
        ohos:bottom_margin="3vp"
        ohos:left_margin="13vp"
        ohos:background_element="$graphic:color_cyan_element"
        ohos:text="Button 2"/>
    <Button
        ohos:width="33vp"
```

```
        ohos:height="20vp"
        ohos:bottom_margin="3vp"
        ohos:left_margin="13vp"
        ohos:background_element="$graphic:color_cyan_element"
        ohos:text="Button 3"/>
</DirectionalLayout>
```

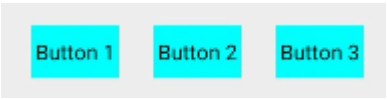
color_cyan_element.xml：

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:shape="rectangle">
    <solid
        ohos:color="#ff00ffff"/>
</shape>
```

水平排列

水平方向排列三个按钮，效果如下：

图 3 三个水平排列的按钮




```
<?xml version="1.0" encoding="utf-8"?>
<DirectionalLayout
    xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:width="match_parent"
    ohos:height="match_content"
    ohos:orientation="horizontal">
    <Button
```

```
        ohos:width="33vp"
        ohos:height="20vp"
        ohos:left_margin="13vp"
        ohos:background_element="$graphic:color_cyan_element"
        ohos:text="Button 1"/>
    <Button
        ohos:width="33vp"
        ohos:height="20vp"
        ohos:left_margin="13vp"
        ohos:background_element="$graphic:color_cyan_element"
        ohos:text="Button 2"/>
    <Button
        ohos:width="33vp"
        ohos:height="20vp"
        ohos:left_margin="13vp"
        ohos:background_element="$graphic:color_cyan_element"
        ohos:text="Button 3"/>
</DirectionalLayout>
```

color_cyan_element.xml：

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:shape="rectangle">
    <solid
        ohos:color="#ff00ffff"/>
</shape>
```

DirectionalLayout 不会自动换行，其子组件会按照设定的方向依次排列，若超过布局本身的大小，超出布局大小的部分将不会被显示，例如：

```
<?xml version="1.0" encoding="utf-8"?>
<DirectionalLayout
    xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:width="match_parent"
    ohos:height="20vp"
    ohos:orientation="horizontal">
    <Button
        ohos:width="166vp"
        ohos:height="match_content"
        ohos:left_margin="13vp"
        ohos:background_element="$graphic:color_cyan_element"
        ohos:text="Button 1"/>
    <Button
        ohos:width="166vp"
        ohos:height="match_content"
        ohos:left_margin="13vp"
        ohos:background_element="$graphic:color_cyan_element"
        ohos:text="Button 2"/>
    <Button
        ohos:width="166vp"
        ohos:height="match_content"
        ohos:left_margin="13vp"
        ohos:background_element="$graphic:color_cyan_element"
        ohos:text="Button 3"/>
```

```
</DirectionalLayout>
```

color_cyan_element.xml：

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:shape="rectangle">
    <solid
        ohos:color="#ff00ffff"/>
</shape>
```

此布局包含了三个按钮，但由于 DirectionalLayout 不会自动换行，超出布局大小的组件部分无法显示。界面显示如下：

图 4 DirectionalLayout 不自动换行示例

对齐方式

DirectionalLayout 中的组件使用 layout_alignment 控制自身在布局中的对齐方式，当对齐方式与排列方式方向一致时，对齐方式不会生效，如设置了水平方向的排列方式，则左对齐、右对齐将不会生效。常用的对齐参数见表 1。

表 1 常用的对齐参数

参数	作用	可搭配排列方式
left	左对齐	垂直排列

表 1 常用的对齐参数

参数	作用	可搭配排列方式
top	顶部对齐	水平排列
right	右对齐	垂直排列
bottom	底部对齐	水平排列
horizontal_center	水平方向居中	垂直排列
vertical_center	垂直方向居中	水平排列
center	垂直与水平方向都居中	水平/垂直排列

三种对齐方式的示例代码：

```
<?xml version="1.0" encoding="utf-8"?>
<DirectionalLayout
    xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:width="match_parent"
    ohos:height="60vp">
    <Button
        ohos:width="50vp"
        ohos:height="20vp"
        ohos:background_element="$graphic:color_cyan_element"
        ohos:layout_alignment="left"
        ohos:text="Button 1"/>
    <Button
        ohos:width="50vp"
        ohos:height="20vp"
```

```
        ohos:background_element="$graphic:color_cyan_element"
        ohos:layout_alignment="horizontal_center"
        ohos:text="Button 2"/>
    <Button
        ohos:width="50vp"
        ohos:height="20vp"
        ohos:background_element="$graphic:color_cyan_element"
        ohos:layout_alignment="right"
        ohos:text="Button 3"/>
</DirectionalLayout>
```

color_cyan_element.xml：

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:shape="rectangle">
    <solid
        ohos:color="#ff00ffff"/>
</shape>
```

图 5 三种对齐方式效果示例

权重

权重（weight）就是按比例来分配组件占用父组件的大小，在水平布局下计算公式为：

父布局可分配宽度=父布局宽度-所有子组件 width 之和；

组件宽度=组件 weight/所有组件 weight 之和*父布局可分配宽度；

实际使用过程中，建议使用 width=0 来按比例分配父布局的宽度，1:1:1 效果如下：


```
<?xml version="1.0" encoding="utf-8"?>
<DirectionalLayout
    xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:width="match_parent"
    ohos:height="match_content"
    ohos:orientation="horizontal">
    <Button
        ohos:width="0vp"
        ohos:height="20vp"
        ohos:weight="1"
        ohos:background_element="$graphic:color_cyan_element"
        ohos:text="Button 1"/>
    <Button
        ohos:width="0vp"
        ohos:height="20vp"
```

```
        ohos:weight="1"
        ohos:background_element="$graphic:color_gray_element"
        ohos:text="Button 2"/>
    <Button
        ohos:width="0vp"
        ohos:height="20vp"
        ohos:weight="1"
        ohos:background_element="$graphic:color_cyan_element"
        ohos:text="Button 3"/>
</DirectionalLayout>
```

color_cyan_element.xml：

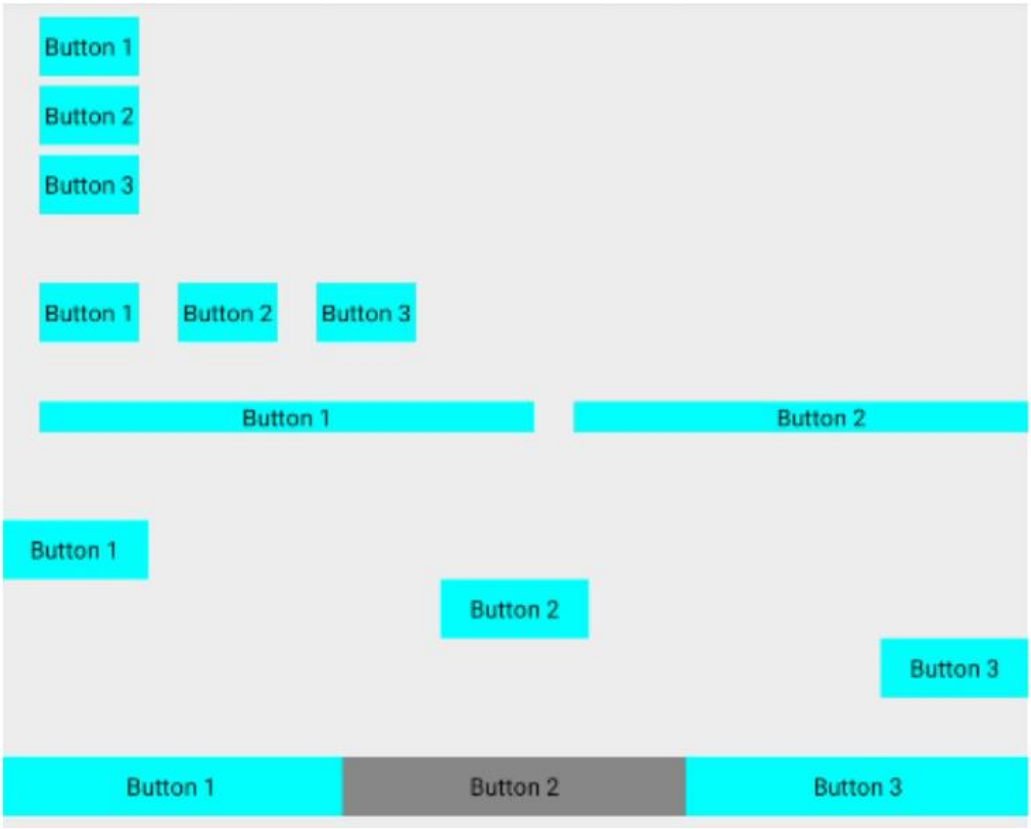
```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:shape="rectangle">
    <solid
        ohos:color="#ff00ffff"/>
</shape>
```

color_gray_element.xml：

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:shape="rectangle">
    <solid
        ohos:color="#ff888888"/>
```

```
</shape>
```

场景示例



源码示例：

```
<?xml version="1.0" encoding="utf-8"?>
<DirectionalLayout xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:width="match_parent"
    ohos:height="match_parent"
    ohos:background_element="$graphic:color_light_gray_element">
```

```
    <DirectionalLayout
        ohos:width="match_parent"
        ohos:height="match_content"
        ohos:orientation="vertical">
        <Button
            ohos:width="33vp"
            ohos:height="20vp"
            ohos:bottom_margin="3vp"
            ohos:left_margin="13vp"
            ohos:background_element="$graphic:color_cyan_element"
            ohos:text="Button 1"/>
        <Button
            ohos:width="33vp"
            ohos:height="20vp"
            ohos:bottom_margin="3vp"
            ohos:left_margin="13vp"
            ohos:background_element="$graphic:color_cyan_element"
            ohos:text="Button 2"/>
        <Button
            ohos:width="33vp"
            ohos:height="20vp"
            ohos:bottom_margin="3vp"
            ohos:left_margin="13vp"
            ohos:background_element="$graphic:color_cyan_element"
            ohos:text="Button 3"/>
    </DirectionalLayout>
    <Component ohos:height="20vp"/>
    <DirectionalLayout
```

```
        ohos:width="match_parent"
        ohos:height="match_content"
        ohos:orientation="horizontal">
        <Button
            ohos:width="33vp"
            ohos:height="20vp"
            ohos:left_margin="13vp"
            ohos:background_element="$graphic:color_cyan_element"
            ohos:text="Button 1"/>
        <Button
            ohos:width="33vp"
            ohos:height="20vp"
            ohos:left_margin="13vp"
            ohos:background_element="$graphic:color_cyan_element"
            ohos:text="Button 2"/>
        <Button
            ohos:width="33vp"
            ohos:height="20vp"
            ohos:left_margin="13vp"
            ohos:background_element="$graphic:color_cyan_element"
            ohos:text="Button 3"/>
    </DirectionalLayout>
    <Component ohos:height="20vp"/>
    <DirectionalLayout
        ohos:width="match_parent"
        ohos:height="20vp"
        ohos:orientation="horizontal">
        <Button
```

```
            ohos:width="166vp"
            ohos:height="match_content"
            ohos:left_margin="13vp"
            ohos:background_element="$graphic:color_cyan_element"
            ohos:text="Button 1"/>
        <Button
            ohos:width="166vp"
            ohos:height="match_content"
            ohos:left_margin="13vp"
            ohos:background_element="$graphic:color_cyan_element"
            ohos:text="Button 2"/>
        <Button
            ohos:width="166vp"
            ohos:height="match_content"
            ohos:left_margin="13vp"
            ohos:background_element="$graphic:color_cyan_element"
            ohos:text="Button 3"/>
    </DirectionalLayout>
    <Component ohos:height="20vp"/>
    <DirectionalLayout
        ohos:width="match_parent"
        ohos:height="60vp">
        <Button
            ohos:width="50vp"
            ohos:height="20vp"
            ohos:background_element="$graphic:color_cyan_element"
            ohos:layout_alignment="left"
            ohos:text="Button 1"/>
```

```
        <Button
            ohos:width="50vp"
            ohos:height="20vp"
            ohos:background_element="$graphic:color_cyan_element"
            ohos:layout_alignment="horizontal_center"
            ohos:text="Button 2"/>
        <Button
            ohos:width="50vp"
            ohos:height="20vp"
            ohos:background_element="$graphic:color_cyan_element"
            ohos:layout_alignment="right"
            ohos:text="Button 3"/>
    </DirectionalLayout>
    <Component ohos:height="20vp"/>
    <DirectionalLayout
        ohos:width="match_parent"
        ohos:height="match_content"
        ohos:orientation="horizontal">
        <Button
            ohos:width="0vp"
            ohos:height="20vp"
            ohos:weight="1"
            ohos:background_element="$graphic:color_cyan_element"
            ohos:text="Button 1"/>
        <Button
            ohos:width="0vp"
            ohos:height="20vp"
            ohos:weight="1"
```

```
            ohos:background_element="$graphic:color_gray_element"
            ohos:text="Button 2"/>
        <Button
            ohos:width="0vp"
            ohos:height="20vp"
            ohos:weight="1"
            ohos:background_element="$graphic:color_cyan_element"
            ohos:text="Button 3"/>
    </DirectionalLayout>
</DirectionalLayout>
```

color_light_gray_element.xml：

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

color_cyan_element.xml：

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:shape="rectangle">
    <solid
        ohos:color="#ff00ffff"/>
```

```
</shape>
```

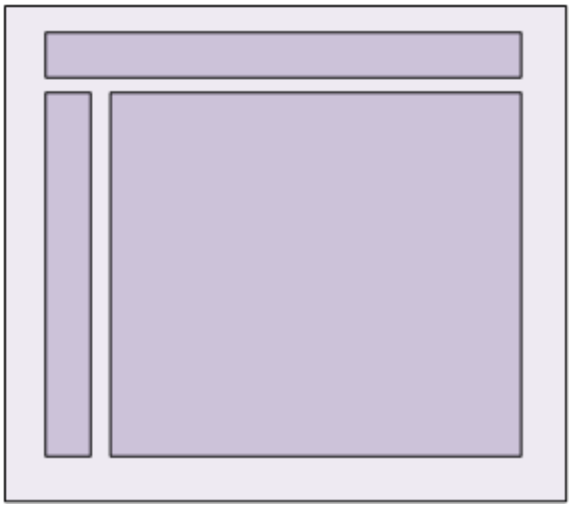
color_gray_element.xml：

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:shape="rectangle">
    <solid
        ohos:color="#ff888888"/>
</shape>
```

DependentLayout

DependentLayout 是 Java UI 系统里的一种常见布局。与 DirectionalLayout 相比，拥有更多的排布方式，每个组件可以指定相对于其他同级元素的位置，或者指定相对于父组件的位置。

图 1 DependentLayout 示意图



排列方式

DependentLayout 的排列方式是相对于其他同级组件或者父组件的位置进行布局。

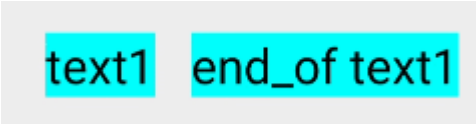
- **相对于同级组件**

相对于同级组件的位置布局见表 1。

表 1 相对于同级组件的位置布局

位置布局	描述
above	处于同级组件的上侧。
below	处于同级组件的下侧。
start_of	处于同级组件的起始侧。
end_of	处于同级组件的结束侧。
left_of	处于同级组件的左侧。
right_of	处于同级组件的右侧。

end_of:




```
<?xml version="1.0" encoding="utf-8"?>
<DependentLayout
    xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:width="match_content"
    ohos:height="match_content"
    ohos:background_element="$graphic:color_light_gray_element">
    <Text
        ohos:id="$+id:text1"
        ohos:width="match_content"
```

```
        ohos:height="match_content"
        ohos:left_margin="15vp"
        ohos:top_margin="15vp"
        ohos:bottom_margin="15vp"
        ohos:text="text1"
        ohos:text_size="20fp"
        ohos:background_element="$graphic:color_cyan_element"/>
    <Text
        ohos:id="$+id:text2"
        ohos:width="match_content"
        ohos:height="match_content"
        ohos:left_margin="15vp"
        ohos:top_margin="15vp"
        ohos:right_margin="15vp"
        ohos:bottom_margin="15vp"
        ohos:text="end_of text1"
        ohos:text_size="20fp"
        ohos:background_element="$graphic:color_cyan_element"
        ohos:end_of="$id:text1"/>
</DependentLayout>
```

color_light_gray_element.xml：

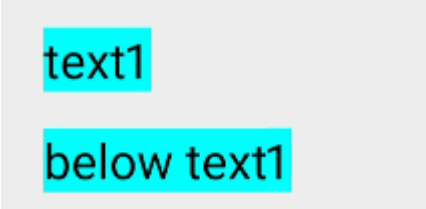
```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:shape="rectangle">
    <solid
        ohos:color="#ffeeeeee"/>
```

```
</shape>
```

color_cyan_element.xml：

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:shape="rectangle">
    <solid
        ohos:color="#ff00ffff"/>
</shape>
```

below：




```
<?xml version="1.0" encoding="utf-8"?>
<DependentLayout
    xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:width="match_content"
    ohos:height="match_content"
    ohos:background_element="$graphic:color_light_gray_element">
    <Text
        ohos:id="$+id:text1"
        ohos:width="match_content"
        ohos:height="match_content"
        ohos:left_margin="15vp"
        ohos:top_margin="15vp"
        ohos:right_margin="40vp"
```

```
        ohos:text="text1"
        ohos:text_size="20fp"
        ohos:background_element="$graphic:color_cyan_element"/>
    <Text
        ohos:id="$+id:text3"
        ohos:width="match_content"
        ohos:height="match_content"
        ohos:left_margin="15vp"
        ohos:top_margin="15vp"
        ohos:right_margin="40vp"
        ohos:bottom_margin="15vp"
        ohos:text="below text1"
        ohos:text_size="20fp"
        ohos:background_element="$graphic:color_cyan_element"
        ohos:below="$id:text1"/>
</DependentLayout>
```

color_light_gray_element.xml：

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

color_cyan_element.xml：

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<shape xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:shape="rectangle">
    <solid
        ohos:color="#ff00ffff"/>
</shape>
```

其他的 above、start_of、left_of、right_of 等参数可分别实现类似的布局。

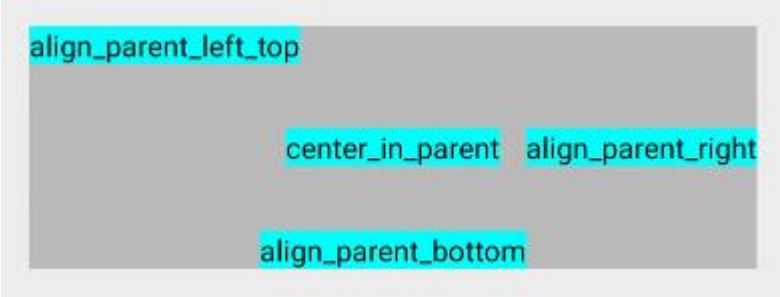
- **相对于父组件**

相对于父组件的位置布局见表 2。

表 2 相对于父组件的位置布局

位置布局	描述
align_parent_left	处于父组件的左侧。
align_parent_right	处于父组件的右侧。
align_parent_start	处于父组件的起始侧。
align_parent_end	处于父组件的结束侧。
align_parent_top	处于父组件的上侧。
align_parent_bottom	处于父组件的下侧。
center_in_parent	处于父组件的中间。

- 以上位置布局可以组合，形成处于左上角、左下角、右上角、右下角的布局。

- 


```
<?xml version="1.0" encoding="utf-8"?>
<DependentLayout
    xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:width="300vp"
    ohos:height="100vp"
    ohos:background_element="$graphic:color_background_gray_element">
    <Text
        ohos:id="$+id:text6"
        ohos:width="match_content"
        ohos:height="match_content"
        ohos:text="align_parent_right"
        ohos:text_size="12fp"
        ohos:background_element="$graphic:color_cyan_element"
        ohos:align_parent_right="true"
        ohos:center_in_parent="true"/>
    <Text
        ohos:id="$+id:text7"
        ohos:width="match_content"
        ohos:height="match_content"
        ohos:text="align_parent_bottom"
```

```
        ohos:text_size="12fp"
        ohos:background_element="$graphic:color_cyan_element"
        ohos:align_parent_bottom="true"
        ohos:center_in_parent="true"/>
    <Text
        ohos:id="$+id:text8"
        ohos:width="match_content"
        ohos:height="match_content"
        ohos:text="center_in_parent"
        ohos:text_size="12fp"
        ohos:background_element="$graphic:color_cyan_element"
        ohos:center_in_parent="true"/>
    <Text
        ohos:id="$+id:text9"
        ohos:width="match_content"
        ohos:height="match_content"
        ohos:text="align_parent_left_top"
        ohos:text_size="12fp"
        ohos:background_element="$graphic:color_cyan_element"
        ohos:align_parent_left="true"
        ohos:align_parent_top="true"/>
</DependentLayout>
```

color_background_gray_element.xml：

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

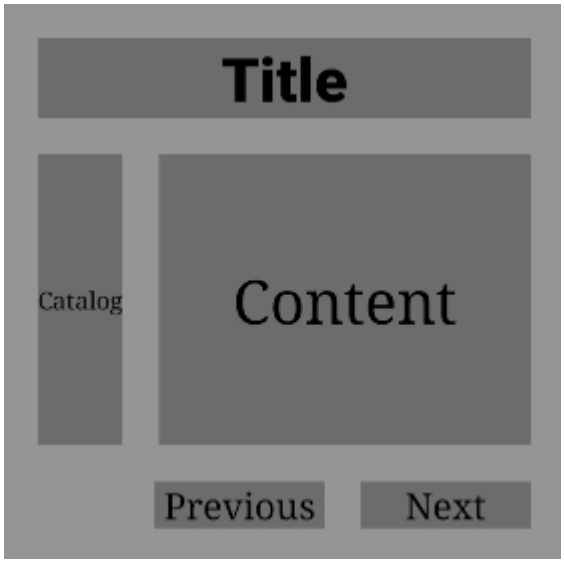
```
    <solid
        ohos:color="#ffbbbbbb"/>
</shape>
```

color_cyan_element.xml：

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:shape="rectangle">
    <solid
        ohos:color="#ff00ffff"/>
</shape>
```

场景示例

使用 DependentLayout 可以轻松实现内容丰富的布局。




```
<?xml version="1.0" encoding="utf-8"?>
<DependentLayout
```

```
    xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:width="match_parent"
    ohos:height="match_content"
    ohos:background_element="$graphic:color_background_gray_element">
    <Text
        ohos:id="$+id:text1"
        ohos:width="match_parent"
        ohos:height="match_content"
        ohos:text_size="25fp"
        ohos:top_margin="15vp"
        ohos:left_margin="15vp"
        ohos:right_margin="15vp"
        ohos:background_element="$graphic:color_gray_element"
        ohos:text="Title"
        ohos:text_weight="1000"
        ohos:text_alignment="horizontal_center"
    />
    <Text
        ohos:id="$+id:text2"
        ohos:width="match_content"
        ohos:height="120vp"
        ohos:text_size="10vp"
        ohos:background_element="$graphic:color_gray_element"
        ohos:text="Catalog"
        ohos:top_margin="15vp"
        ohos:left_margin="15vp"
        ohos:right_margin="15vp"
        ohos:bottom_margin="15vp"
```

```
        ohos:align_parent_left="true"
        ohos:text_alignment="center"
        ohos:multiple_lines="true"
        ohos:below="$id:text1"
        ohos:text_font="serif"/>
    <Text
        ohos:id="$+id:text3"
        ohos:width="match_parent"
        ohos:height="120vp"
        ohos:text_size="25fp"
        ohos:background_element="$graphic:color_gray_element"
        ohos:text="Content"
        ohos:top_margin="15vp"
        ohos:right_margin="15vp"
        ohos:bottom_margin="15vp"
        ohos:text_alignment="center"
        ohos:below="$id:text1"
        ohos:end_of="$id:text2"
        ohos:text_font="serif"/>
    <Button
        ohos:id="$+id:button1"
        ohos:width="70vp"
        ohos:height="match_content"
        ohos:text_size="15fp"
        ohos:background_element="$graphic:color_gray_element"
        ohos:text="Previous"
        ohos:right_margin="15vp"
        ohos:bottom_margin="15vp"
```

```
        ohos:below="$id:text3"
        ohos:left_of="$id:button2"
        ohos:italic="false"
        ohos:text_weight="5"
        ohos:text_font="serif"/>
    <Button
        ohos:id="$+id:button2"
        ohos:width="70vp"
        ohos:height="match_content"
        ohos:text_size="15fp"
        ohos:background_element="$graphic:color_gray_element"
        ohos:text="Next"
        ohos:right_margin="15vp"
        ohos:bottom_margin="15vp"
        ohos:align_parent_end="true"
        ohos:below="$id:text3"
        ohos:italic="false"
        ohos:text_weight="5"
        ohos:text_font="serif"/>
</DependentLayout>
```

color_background_gray_element.xml：

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

```
        ohos:color="#ffbbbbbb"/>
</shape>
```

color_gray_element.xml：

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:shape="rectangle">
    <solid
        ohos:color="#ff888888"/>
</shape>
```

动画开发指导

动画是组件的基础特性之一，精心设计的动画使 UI 变化更直观，有助于改进应用程序的外观并改善用户体验。Java UI 框架提供了数值动画（AnimatorValue）和属性动画（AnimatorProperty），并提供了将多个动画同时操作的动画集合（AnimatorGroup）。

数值动画（AnimatorValue）

AnimatorValue 数值从 0 到 1 变化，本身与 Component 无关。开发者可以设置 0 到 1 变化过程的属性，例如：时长、变化曲线、重复次数等，并通过值的变化改变组件的属性，实现组件的动画效果。

1.声明 AnimatorValue。

```
AnimatorValue animator = new AnimatorValue();
```

设置变化属性。

```
animator.setDuration(2000);

animator.setDelay(1000);

animator.setLoopedCount(2);

animator.setCurveType(Animator.CurveType.BOUNCE);
```

添加回调事件。

```
animator.setValueUpdateListener(new AnimatorValue.ValueUpdateListener() {

    @Override

    public void onUpdate(AnimatorValue animatorValue, float value) {

        button.setContentPosition((int) (800 * value),
button.getContentPositionY());

    }
```

```
});
```

启动动画或对动画做其他操作。

```
animator.start();
```

AnimatorGroup 动画效果如图所示：

图 1 数值动画效果

属性动画（AnimatorProperty）

为 Component 的属性设置动画是非常常见的需求，Java UI 框架可以为 Component 设置某个属性或多个属性的动画。

1. 声明 AnimatorProperty。

```
AnimatorProperty animator = button.createAnimatorProperty();
```

设置变化属性，可链式调用。

```
animator.moveFromX(50).moveToX(1000).rotate(180).alpha(0).setDuration(2500).setDelay(500).setLoopedCount(5);
```

启动动画或对动画做其他操作。

```
animator.start();
```

可以使用 setTarget()改变关联的 Component 对象。

```
animator.setTarget(button2);
```

动画效果如图所示：

图 2 属性动画效果

动画集合（AnimatorGroup）

如果需要使用一个组合动画，可以把多个动画对象进行组合，并添加到使用 AnimatorGroup 中。AnimatorGroup 提供了两个方法：runSerially() 和 runParallel()，分别表示动画按顺序开始和动画同时开始。

1. 声明 AnimatorGroup。

```
AnimatorGroup animatorGroup = new AnimatorGroup();
```

添加要按顺序或同时开始的动画。

```
// 4 个动画按顺序播放
animatorGroup.runSerially(am1, am2, am3, am4);
// 4 个动画同时播放
animatorGroup.runParallel(am1, am2, am3, am4);
```

启动动画或对动画做其他操作。

```
animatorGroup.start();
```

为了更加灵活处理多个动画的播放顺序，例如一些动画顺序播放，一些动画同时播放，Java UI 框架提供了更方便的动画 Builder 接口：

1. 声明 AnimatorGroup Builder。

```
AnimatorGroup.Builder animatorGroupBuilder = animatorGroup.build();
```

按播放顺序添加多个动画。

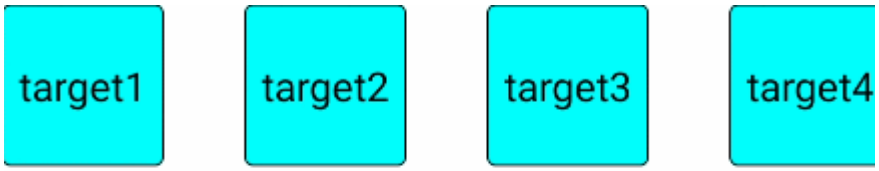
```
// 4 个动画的顺序为: am1 -> am2/am3 -> am4
animatorGroupBuilder.addAnimators(am1).addAnimators(am2,
am3).addAnimators(am4)
```

启动动画或对动画做其他操作。

```
animatorGroup.start();
```

动画集合的动画效果如下：

图 3 动画集合效果



可见即可说开发指导

可见即可说是要求 Component 中通过与热词关联，从而达到指定的效果。例如：在浏览图片时，说出图片的名字或角标序号，从而实现打开图片的效果。

说明

该功能目前仅在智慧屏产品上支持。

热词注册

开发者首先需要进行 Component 的热词注册，即告诉设备，哪些热词是这个 Component 所需要响应的。

构建 Component.VoiceEvent 对象，需要设置热词，中英文都可以。

```
Component.VoiceEvent eventKeys = new Component.VoiceEvent("ok");
```

如果一个 Component 的同一 VoiceEvent 存在多个热词匹配，可以通过 addSynonyms 方法增加 eventKeys 的热词。

```
eventKeys.addSynonyms("确定");
```

当 Component.VoiceEvent 对象操作完成后，使用 Component 的 subscribeVoiceEvents 方法来发起注册。

```
Component.subscribeVoiceEvents(eventKeys);
```

如果一个 Component 有多个事件需要响应，需要创建不同的事件来进行注册。

事件响应

开发者完成热词注册后，需要关注的是对应于不同热词所需要处理的事件。事件响应回调的 SpeechEvent 对象仅包含一个热词。

1. 首先需要实现 SpeechEventListener 接口。

```
private Component.SpeechEventListener speechEventListener = new
Component.SpeechEventListener(

    @Override

    public boolean onSpeechEvent(Component v, SpeechEvent event) {

        if (event.getActionProperty().equals("ok")) {

            ... // 检测注册的热词，进行相应的处理

        }

});
```

通过 setSpeechEventListener 方法实现回调注册。

```
Component.setSpeechEventListener(speechEventListener);
```

JS UI 框架

概述

JS UI 框架是一种跨设备的高性能 UI 开发框架，支持声明式编程和跨设备多态 UI。

阅读本开发指南前，开发者需要掌握以下基础知识：

- HTML5
- CSS
- JavaScript

说明

本文档适用于智慧屏（TV）和智能穿戴（Wearable）应用开发，针对轻量级智能穿戴（Lite Wearable）请参考轻量级智能穿戴开发。

基础能力

- **声明式编程**

JS UI 框架采用类 HTML 和 CSS 声明式编程语言作为页面布局和页面样式的开发语言，页面业务逻辑则支持 ECMAScript 规范的 JavaScript 语言。JS UI 框架提供的声明式编程，可以让开发者避免编写 UI 状态切换的代码，视图配置信息更加直观。

- **跨设备**

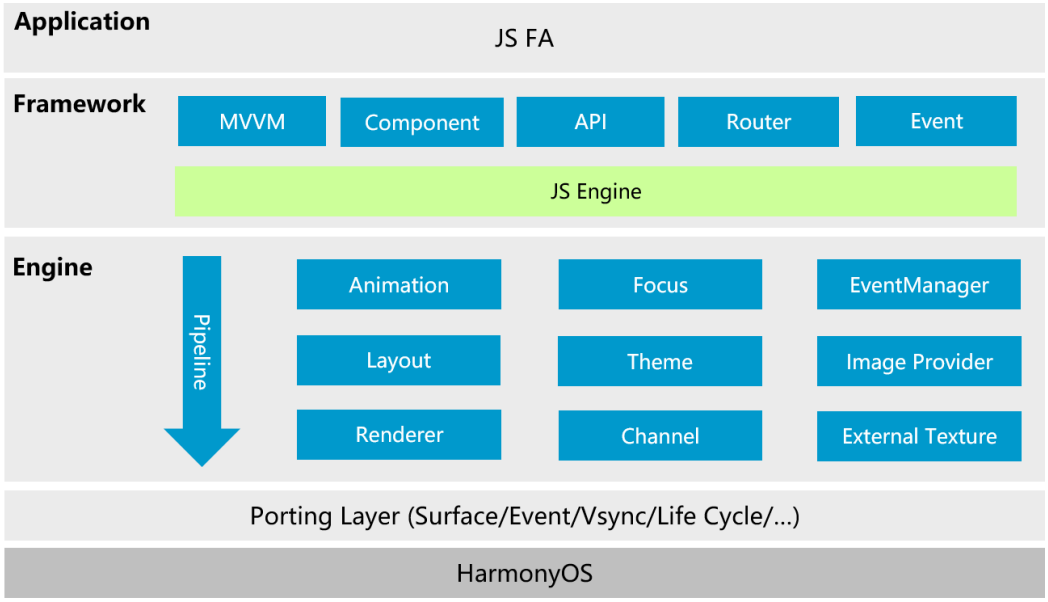
开发框架架构上支持 UI 跨设备显示能力，运行时自动映射到不同设备类型，开发者无感知，降低开发者多设备适配成本。

- **高性能**

开发框架包含了许多核心的控件，如列表、图片和各类容器组件等，针对声明式语法进行了渲染流程的优化。

整体架构

JS UI 框架包括应用层（Application）、前端框架层（Framework）、引擎层（Engine）和平台适配层（Porting Layer）。



- **Application**

应用层表示开发者使用 JS UI 框架开发的 FA 应用，这里的 FA 应用特指 JS FA 应用。使用 Java 开发 FA 应用请参考 Java UI 框架。

- **Framework**

前端框架层主要完成前端页面解析，以及提供 MVVM（Model-View-ViewModel）开发模式、页面路由机制和自定义组件等能力。

- **Engine**

引擎层主要提供动画解析、DOM（Document Object Model）树构建、布局计算、渲染命令构建与绘制、事件管理等能力。

- **Porting Layer**

适配层主要完成对平台层进行抽象，提供抽象接口，可以对接到系统平台。比如：事件对接、渲染管线对接和系统生命周期对接等。

初步体验 JS FA 应用

JS FA 概述

JS UI 框架支持纯 JavaScript、JavaScript 和 Java 混合语言开发。JS FA 指基于 JavaScript 或 JavaScript 和 Java 混合开发的 FA，下面主要介绍：JS FA 在 HarmonyOS 上运行时需要的基类 AceAbility、加载 JS FA 主体的方法、JS FA 开发目录。

AceAbility

AceAbility 类是 JS FA 在 HarmonyOS 上运行环境的基类，继承自 Ability。开发者的应用运行入口类应该从该类派生，代码示例如下：

```
public class MainAbility extends AceAbility {

    @Override

    public void onStart(Intent intent) {

        super.onStart(intent);

    }


    @Override

    public void onStop() {

        super.onStop();

    }

}
```

如何加载 JS FA

JS FA 生命周期事件分为应用生命周期和页面生命周期，应用通过 AceAbility 类中 setInstanceName()接口设置该 Ability 的实例资源，并通过 AceAbility 窗口进行显示以及全局应用生命周期管理。

setInstanceName(String name)的参数“name”指实例名称，实例名称与 config.json 文件中 profile.application.js.name 的值对应。若开发者未修改实例名，而使用了缺省值 default，则无需调用此接口。若开发者修改了实例名，则需在应用 Ability 实例的 onStart()中调用此接口，并将参数“name”设置为修改后的实例名称。

说明

多实例应用的 profile.application.js 字段中有多个实例项，使用时请选择相应的实例名称。

setInstanceName()接口使用方法：在 MainAbility 的 onStart()中的 super.onStart()前调用此接口。以 JSComponentName 作为实例名称，代码示例如下：

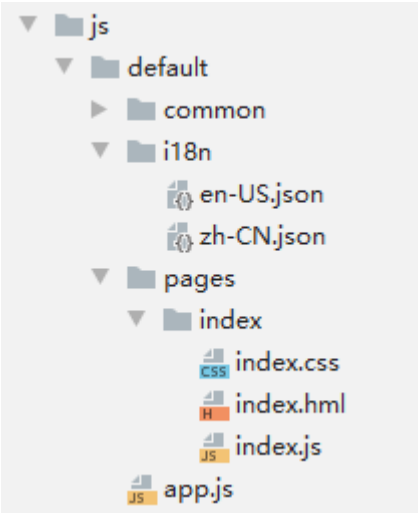
```
public class MainAbility extends AceAbility {

    @Override

    public void onStart(Intent intent) {

        setInstanceName("JSComponentName");  // config.json 配置文件中 ability.js.name 的标签值。

        super.onStart(intent);

    }

}
```

说明

需在 super.onStart(Intent)前调用此接口。

JS FA 开发目录

新建工程的 JS 目录如下图所示。



在工程目录中：common 文件夹主要存放公共资源，如图片、视频等；i18n 下存放多语言的 json 文件；pages 文件夹下存放多个页面，每个页面由 hml、css 和 js 文件组成。

- **main > js > default > i18n > en-US.json**：此文件定义了在英文模式下页面显示的变量内容。同理，zh-CN.json 中定义了中文模式下的页面内容。

```
{
```

```
  "strings": {
    "hello": "Hello",
    "world": "World"
  },
  "files": {
  }
}
```

main > js > default > pages > index > index.hml：此文件定义了 index 页面的布局、index 页面中用到的组件，以及这些组件的层级关系。例如：index.hml 文件中包含了一个 text 组件，内容为“Hello World”文本。

```
<div class ="container">
  <text class ="title">
    {{ $t('strings.hello') }} {{title}}
  </text>
</div>
```

main > js > default > pages > index > index.css：此文件定义了 index 页面的样式。例如：index.css 文件定义了“container”和“title”的样式。

```
.container {
  flex-direction: column;
  justify-content: center;
  align-items: center;
}
.title {
  font-size: 100px;
```

```
}
```

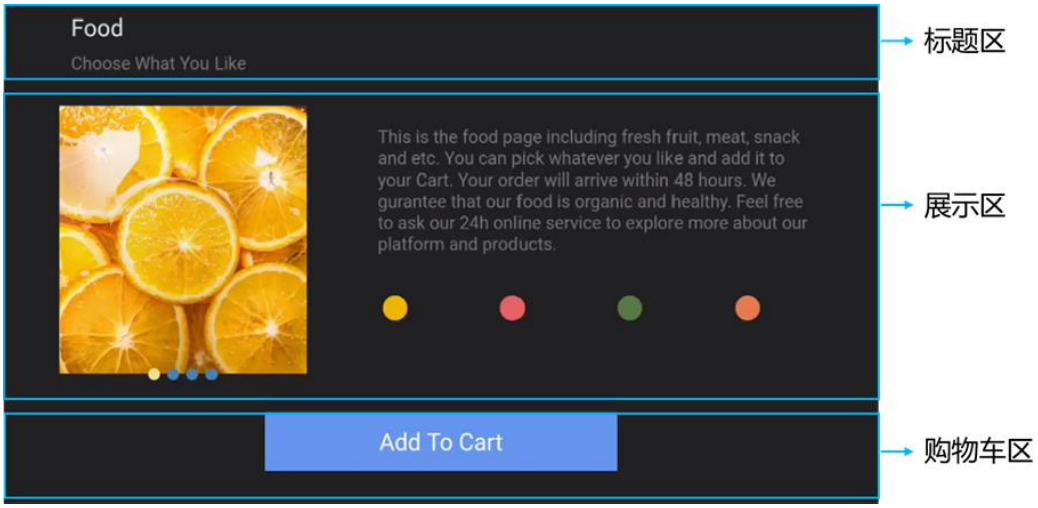
main > js > default > pages > index > index.js：此文件定义了 index 页面的业务逻辑，比如数据绑定、事件处理等。例如：变量“title”赋值为字符串“World”。

```
export default {
  data: {
    title: '',
  },
  onInit() {
    this.title = this.$t('strings.world');
  },
}
```

开发一个 JS FA 应用

本章节主要介绍如何开发一个 JS FA 应用。此应用相对于 Hello World 应用模板具备更复杂的页面布局、页面样式和页面逻辑。该页面可以通过将焦点移动到不同颜色的圆形来选择不同的食物图片，也可以进行添加到购物车操作，应用效果图如下。

图 1　JS FA 应用效果图



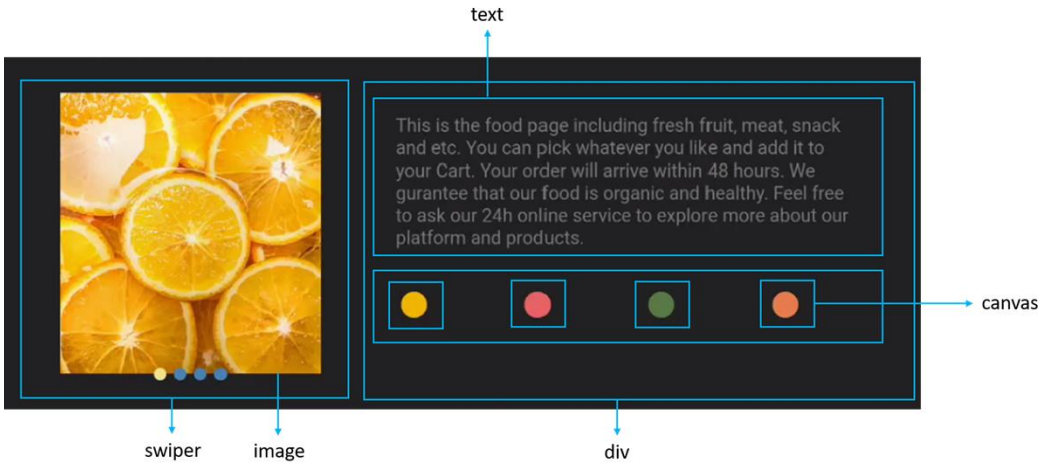
构建页面布局

开发者在 index.hml 文件中构建页面布局。在进行代码开发之前，首先要对页面布局进行分析，将页面分解为不同的部分，用容器组件来承载。根据 JS FA 应用效果图，此页面一共分成三个部分：标题区、展示区和购物车区。根据此分区，可以确定根节点的子节点应按列排列。

标题区是由两个按列排列的 text 组件实现，购物车区由一个 text 组件构成。展示区由按行排列的 swiper 组件和 div 组件组成，如下图所示：

- 第一部分是由一个容器组件 swiper，包含了四个 image 组件构成；
- 第二部分是由一个容器组件 div，包含了一个 text 组件和四个画布组件 canvas 绘制的圆形构成。

图 2　展示区布局



根据布局结构的分析，实现页面基础布局的代码示例如下（其中四个 image 组件和 canvas 组件通过 for 指令来循环创建）：

```
<!-- index.hml -->
<div class="container">
  <div class="title-section">
    <div class="title">
      <text class="name">Food</text>
      <text class="sub-title">Choose What You Like</text>
    </div>
  </div>
  <div>
    <swiper id="swiperImage" class="swiper-style">
      <image src="{{$item}}" class="image-mode" focusable="true"
for="{{imageList}}"></image>
    </swiper>
    <div class="container">
      <div class="description-first-paragraph">
        <text class="description">{{descriptionFirstParagraph}}</text>
      </div>
```

```
      <div class="color-column">
        <canvas id="{{$item.id}}" onfocus="swipeToIndex({{$item.index}})"
class="color-item" focusable="true"
            for="{{canvasList}}"></canvas>
      </div>
    </div>
  </div>
  <div class="cart">
    <text class="{{cartStyle}}" onclick="addCart" onfocus="getFocus"
onblur="lostFocus" focusable="true">
      {{cartText}}</text>
  </div>
</div>
```

说明

common 目录用于存放公共资源，swiper 组件里展示的图片需要放在 common 目录下。

构建页面样式

开发者在 index.css 文件中需要设定的样式主要有 flex-direction（主轴方向），padding（内边距），font-size（字体大小）等。在构建页面样式中，还采用了 css 伪类的写法，当焦点移动到 canvas 组件上时，背景颜色变成白色，也可以在 js 中通过 focus 和 blur 事件动态修改 css 样式来实现同样的效果。

```
/* index.css */
.container {
  flex-direction: column;
```

```
}

.title-section {
  flex-direction: row;
  height: 60px;
  margin-bottom: 5px;
  margin-top: 10px;
}

.title {
  align-items: flex-start;
  flex-direction: column;
  padding-left: 60px;
  padding-right: 160px;
}

.name {
  font-size: 20px;
}

.sub-title {
  font-size: 15px;
  color: #7a787d;
  margin-top: 10px;
}

.swiper-style {
  height: 250px;
```

```
  width: 350px;
  indicator-color: #4682b4;
  indicator-selected-color: #f0e68c;
  indicator-size: 10px;
  margin-left: 50px;
}

.image-mode {
  object-fit: contain;
}

.color-column {
  flex-direction: row;
  align-content: center;
  margin-top: 20px;
}

.color-item {
  height: 50px;
  width: 50px;
  margin-left: 50px;
  padding-left: 10px;
}

.color-item:focus {
  background-color: white;
}
```

```
.description-first-paragraph {
  padding-left: 60px;
  padding-right: 60px;
  padding-top: 30px;
}

.description {
  color: #7a787d;
  font-size: 15px;
}

.cart {
  justify-content: center;
  margin-top: 20px;
}

.cart-text {
  font-size: 20px;
  text-align: center;
  width: 300px;
  height: 50px;
  background-color: #6495ed;
  color: white;
}

.cart-text-focus {
  font-size: 20px;
  text-align: center;
```

```
  width: 300px;
  height: 50px;
  background-color: #4169e1;
  color: white;
}

.add-cart-text {
  font-size: 20px;
  text-align: center;
  width: 300px;
  height: 50px;
  background-color: #ffd700;
  color: white;
}
```

构建页面逻辑

开发者在 index.js 文件中构建页面逻辑，主要实现的是两个逻辑功能：

- 当焦点移动到不同颜色的圆形，swiper 滑动到不同的图片；
- 当焦点移动到购物车区时，“Add To Cart”背景颜色从浅蓝变成深蓝，点击后文字变化为“Cart + 1”，背景颜色由深蓝色变成黄色，添加购物车不可重复操作。

逻辑页面代码示例如下：

```
// index.js

export default {

  data: {

    cartText: 'Add To Cart',

    cartStyle: 'cart-text',

    isCartEmpty: true,

    descriptionFirstParagraph: 'This is the food page including fresh fruit, meat, snack and etc. You can pick whatever you like and add it to your Cart. Your order will arrive within 48 hours. We gurantee that our food is organic and healthy. Feel free to ask our 24h online service to explore more about our platform and products.',

    imageList: ['/common/food_000.JPG', '/common/food_001.JPG', '/common/food_002.JPG', '/common/food_003.JPG'],

    canvasList: [{

      id: 'cycle0',

      index: 0,

      color: '#f0b400',

    }, {

      id: 'cycle1',

      index: 1,

      color: '#e86063',

    }, {

      id: 'cycle2',

      index: 2,

      color: '#597a43',

    }, {

      id: 'cycle3',

      index: 3,

      color: '#e97d4c',
```

```
  }],
},

onShow() {
  this.canvasList.forEach(element => {
    this.drawCycle(element.id, element.color);
  });
},

swipeToIndex(index) {
  this.$element('swiperImage').swipeTo({index: index});
},

drawCycle(id, color) {
  var greenCycle = this.$element(id);
  var ctx = greenCycle.getContext("2d");
  ctx.strokeStyle = color;
  ctx.fillStyle = color;
  ctx.beginPath();
  ctx.arc(15, 25, 10, 0, 2 * 3.14);
  ctx.closePath();
  ctx.stroke();
  ctx.fill();
},

addCart() {
  if (this.isCartEmpty) {
    this.cartText = 'Cart + 1';
```

```
      this.cartStyle = 'add-cart-text';
      this.isCartEmpty = false;
    }
  },

  getFocus() {
    if (this.isCartEmpty) {
      this.cartStyle = 'cart-text-focus';
    }
  },

  lostFocus() {
    if (this.isCartEmpty) {
      this.cartStyle = 'cart-text';
    }
  },
}
```

构建用户界面

组件介绍

组件（Component）是构建页面的核心，每个组件通过对数据和方法的简单封装，实现独立的可视、可交互功能单元。组件之间相互独立，随取随用，也可以在需求相同的地方重复使用。开发者还可以通过组件间合理的搭配定义满足业务需求的新组件，减少开发量，自定义组件的开发方法详见自定义组件。

组件分类

根据组件的功能，可以将组件分为以下四大类：

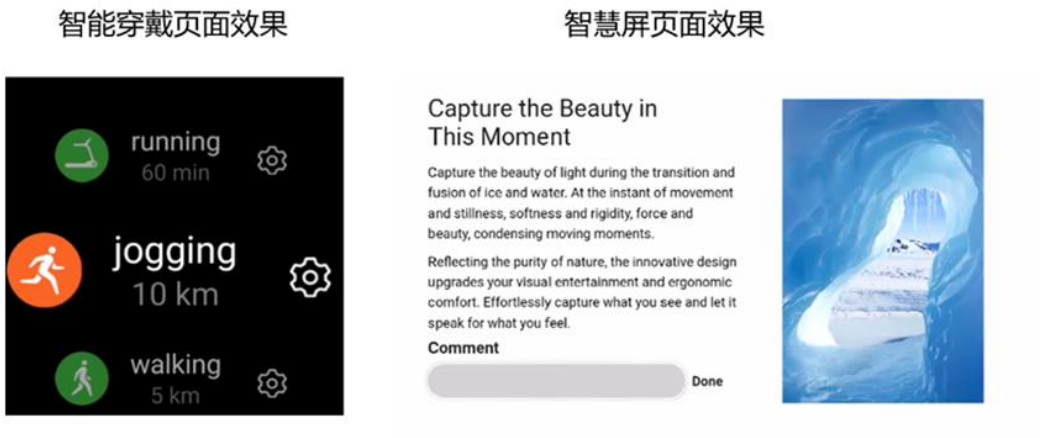
组件类型	主要组件
基础组件	text、image、progress、rating、span、marquee、image-animator、divider、search、menu、chart
容器组件	div、list、list-item、stack、swiper、tabs、tab-bar、tab-content、popup、list-item-group、refresh、dialog
媒体组件	video
画布组件	canvas

构建用户界面

布局说明

JS UI 框架在不同设备的布局示例如下：

图 1　不同设备的布局示例



JS UI 框架中智慧屏以 720px（px 指逻辑像素，非物理像素）为基准宽度，根据实际屏幕宽度进行缩放，例如当 width 设为 100px 时，在宽度为 1440 物理像素的屏幕上，实际显示的宽度为 200 物理像素。智能穿戴的基准宽度为 454px，换算逻辑同理。

一个页面的基本元素包含标题区域、文本区域、图片区域等，每个基本元素内还可以包含多个子元素，开发者根据需求还可以添加按钮、开关、进度条等组件。在构建页面布局时，需要对每个基本元素思考以下几个问题：

- 该元素的尺寸和排列位置
- 是否有重叠的元素
- 是否需要设置对齐、内间距或者边界
- 是否包含子元素及其排列位置
- 是否需要容器组件及其类型

将页面中的元素分解之后再对每个基本元素按顺序实现，可以减少多层嵌套造成的视觉混乱和逻辑混乱，提高代码的可读性，方便对页面做后续的调整。以下图为例进行分解：

图 2　页面布局分解

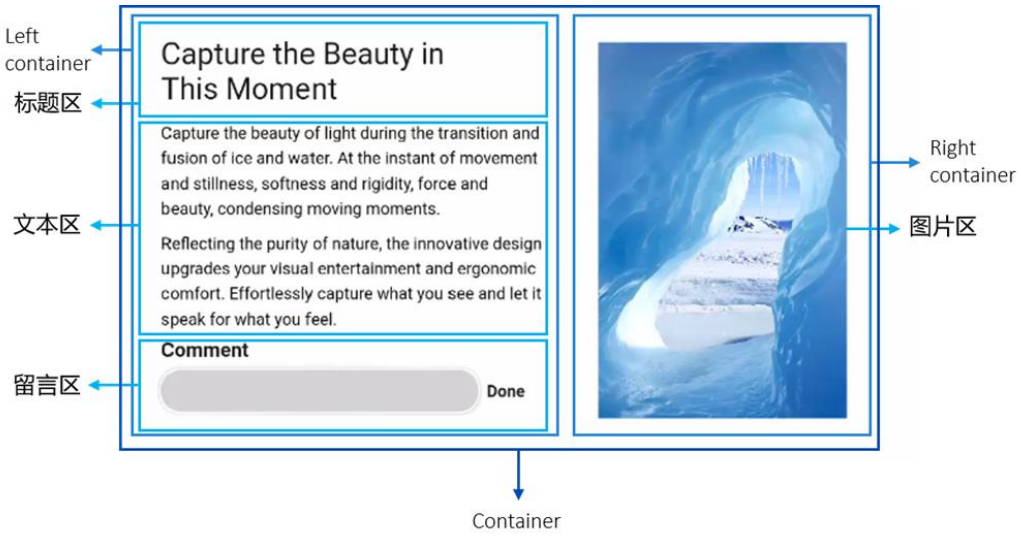
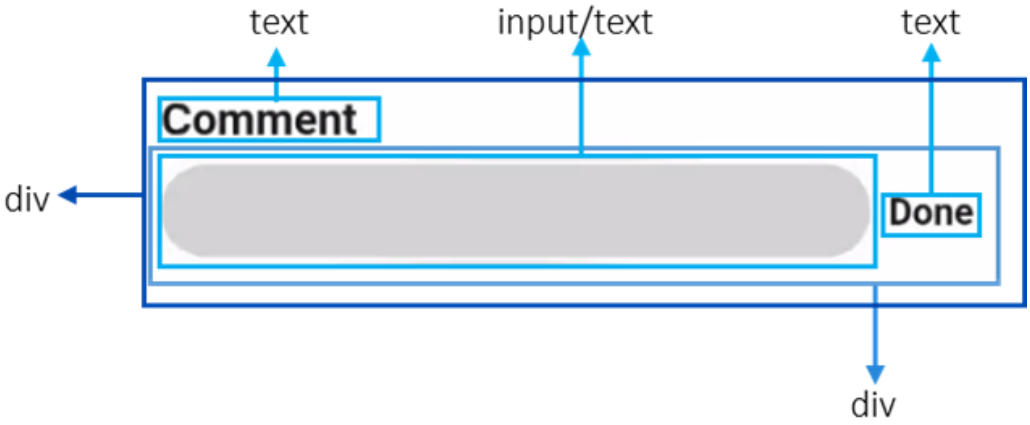

Left
container
标题区
文本区
留言区
Capture the Beauty in
This Moment
Capture the beauty of light during the transition and fusion of ice and water. At the instant of movement and stillness, softness and rigidity, force and beauty, condensing moving moments.
Reflecting the purity of nature, the innovative design upgrades your visual entertainment and ergonomic comfort. Effortlessly capture what you see and let it speak for what you feel.
Comment
Done
Right
container
图片区
Container

图 3　留言区布局分解


text
input/text
text
Comment
div
Done
div

添加标题行和文本区域

实现标题和文本区域最常用的是基础组件 text。text 组件用于展示文本，可以设置不同的属性和样式，文本内容需要写在标签内容区，完整属性和样式信息请参考 text。在页面中插入标题和文本区域的示例如下：

```
<!-- xxx.hml -->
<div class="container">
  <div class="left-container">
    <text class="title-text">{{headTitle}}</text>
    <text class="paragraph-text">{{paragraphFirst}}</text>
    <text class="paragraph-text">{{paragraphSecond}}</text>
  </div>
</div>
```

```
/* xxx.css */
.container {
  margin-top: 24px;
  background-color: #ffffff;
}
.left-container {
  flex-direction: column;
  margin-left: 48px;
  width: 460px;
}
.title-text {
  color: #1a1a1a;
  font-size: 36px;
  height: 90px;
```

```
  width: 400px;
}
.paragraph-text {
  color: #000000;
  margin-top: 12px;
  font-size: 20px;
  line-height: 30px;
}
```

```
// xxx.js
export default {
  data: {
    headTitle: 'Capture the Beauty in This Moment',
    paragraphFirst: 'Capture the beauty of light during the transition and fusion of ice and water. At the instant of movement and stillness, softness and rigidity, force and beauty, condensing moving moments.',
    paragraphSecond: 'Reflecting the purity of nature, the innovative design upgrades your visual entertainment and ergonomic comfort. Effortlessly capture what you see and let it speak for what you feel.',
  },
}
```

添加图片区域

实现图片区域通常用 image 组件来实现，使用的方法和 text 组件类似。图片资源放在 common 目录下，图片的路径要与图片实际所在的目录一致。具体示例如下：

```
<!-- xxx.hml -->
<!-- 插入图片 -->
<div class="right-container">
  <image class="img" src="{{middleImage}}"></image>
</div>
```

```
/* xxx.css */
.right-container {
  width: 432px;
  justify-content: center;
}
.img {
  margin-top: 10px;
  object-fit: contain;
  height: 450px;
}
```

```
// xxx.js
export default {
  data: {
    middleImage: '/common/ice.png',
  },
}
```

添加留言区域

留言框的功能为：用户输入留言后点击完成，留言区域即显示留言内容；用户点击右侧的删除按钮可删除当前留言内容重新输入。

留言区域由 div、text、input 关联 click 事件实现。开发者可以使用 input 组件实现输入留言的部分，使用 text 组件实现留言完成部分，使用 commentText 的状态标记此时显示的组件（通过 if 属性控制）。在包含文本“完成”和“删除”的 text 组件中关联 click 事件，更新 commentText 状态和 inputValue 的内容。具体的实现示例如下：

```
<!-- xxx.hml -->
<div class="container">
  <div class="left-container">
    <text class="comment-title">Comment</text>
    <div if="{{!commentText}}">
      <input class="comment" value="{{inputValue}}"
onchange="updateValue()"></input>
      <text class="comment-key" onclick="update"
focusable="true">Done</text>
    </div>
    <div if="{{commentText}}">
      <text class="comment-text" focusable="true">{{inputValue}}</text>
      <text class="comment-key" onclick="update"
focusable="true">Delete</text>
    </div>
  </div>
</div>
```

```
/* xxx.css */
.container {
```

```
  margin-top: 24px;
  background-color: #ffffff;
}
.left-container {
  flex-direction: column;
  margin-left: 48px;
  width: 460px;
}
.comment-title {
  font-size: 24px;
  color: #1a1a1a;
  font-weight: bold;
  margin-top: 10px;
  margin-bottom: 10px;
}
.comment-key {
  width: 74px;
  height: 50px;
  margin-left: 10px;
  font-size: 20px;
  color: #1a1a1a;
  font-weight: bold;
}
.comment-key:focus {
  color: #007dff;
}
.comment-text {
  width: 386px;
```

```
  height: 50px;
  text-align: left;
  line-height: 35px;
  font-size: 20px;
  color: #000000;
  border-bottom-color: #bcbcbc;
  border-bottom-width: 0.5px;
}
.comment {
  width: 386px;
  height: 50px;
  background-color: lightgrey;
}
```

```
// xxx.js
export default {
  data: {
    inputValue: '',
    commentText: false,
  },
  update() {
    this.commentText = !this.commentText;
  },
  updateValue(e) {
    this.inputValue = e.text;
  },
}
```

添加外部容器

要将页面的基本元素组装在一起，需要使用容器组件。在页面布局中常用到三种容器组件，分别是 div、list 和 tabs。在页面结构相对简单时，可以直接用 div 作为容器，因为 div 作为单纯的布局容器，使用起来更为方便，可以支持多种子组件。

List 组件

当页面结构较为复杂时，如果使用 div 循环渲染，容易出现卡顿，因此推荐使用 list 组件代替 div 组件实现长列表布局，从而实现更加流畅的列表滚动体验。但是，list 组件仅支持 list-item 作为子组件，因此使用 list 时需要留意 list-item 的注意事项。具体的使用示例如下：

```
<!-- xxx.hml -->
<list class="list">
  <list-item type="listItem" for="{{textList}}">
    <text class="desc-text">{{$item.value}}</text>
  </list-item>
</list>
```

```
/* xxx.css */
.desc-text {
  width: 683.3px;
  font-size: 35.4px;
  color: #000000;
}
```

```
// xxx.js
export default {
  data: {
    textList:  [{value: 'JS FA'}],
  },
}
```

为避免示例代码过长，以上示例的 list 中只包含一个 list-item，list-item 中只有一个 text 组件。在实际应用中可以在 list 中加入多个 list-item，同时 list-item 下可以包含多个其他子组件。

Tabs 组件

当页面经常需要动态加载时，推荐使用 tabs 组件。tabs 组件支持 change 事件，在页签切换后触发。tabs 组件仅支持一个 tab-bar 和一个 tab-content。具体的使用示例如下：

```
<!-- xxx.hml -->
<tabs>
  <tab-bar class="tab-bar">
    <text style="color: #000000">tab-bar</text>
  </tab-bar>
  <tab-content>
```

```
    <image src="{{tabImage}}"></image>
  </tab-content>
</tabs>
```

```
/* xxx.css */
.tab-bar {
  background-color: #f2f2f2;
  width: 720px;
}
```

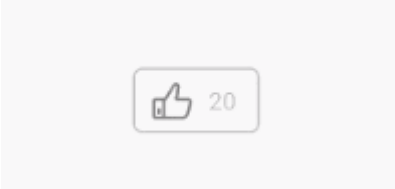
```
// xxx.js
export default {
  data: {
    tabImage: '/common/image.png',
  },
}
```

tab-content 组件用来展示页签的内容区，高度默认充满 tabs 剩余空间。
tab-content 支持 scrollable 属性，详见 tab-content。

添加交互

添加交互通过在组件上关联事件实现。本节将介绍如何用 div、text、image 组件关联 click 事件，构建一个如下图所示的点赞按钮。

图 1　点赞按钮效果



点赞按钮通过一个 div 组件关联 click 事件实现。div 组件包含一个 image 组件和一个 text 组件：

- image 组件用于显示未点赞和点赞的效果。click 事件函数会交替更新点赞和未点赞图片的路径。
- text 组件用于显示点赞数，点赞数会在 click 事件的函数中同步更新。

click 事件作为一个函数定义在 js 文件中，可以更改 isPressed 的状态，从而更新显示的 image 组件。如果 isPressed 为真，则点赞数加 1。该函数在 hml 文件中对应的 div 组件上生效，点赞按钮各子组件的样式设置在 css 文件当中。具体的实现示例如下：

```
<!-- xxx.hml -->

<!--  点赞按钮 -->

<div>

  <div class="like" onclick="likeClick">

    <image class="like-img" src="{{likeImage}}" focusable="true"></image>

    <text class="like-num" focusable="true">{{total}}</text>

  </div>

</div>

/* xxx.css */
```

```
.like {
  width: 104px;
  height: 54px;
  border: 2px solid #bcbcbc;
  justify-content: space-between;
  align-items: center;
  margin-left: 72px;
  border-radius: 8px;
}
.like-img {
  width: 33px;
  height: 33px;
  margin-left: 14px;
}
.like-num {
  color: #bcbcbc;
  font-size: 20px;
  margin-right: 17px;
}
```

```
// xxx.js
export default {
  data: {
    likeImage: '/common/unLike.png',
    isPressed: false,
    total: 20,
  },
```

```
  likeClick() {
    var temp;
    if (!this.isPressed) {
      temp = this.total + 1;
      this.likeImage = '/common/like.png';
    } else {
      temp = this.total - 1;
      this.likeImage = '/common/unLike.png';
    }
    this.total = temp;
    this.isPressed = !this.isPressed;
  },
}
```

JS UI 框架还提供了很多表单组件，例如开关、标签、滑动选择器等，以便于开发者在页面布局时灵活使用和提高交互性，详见容器组件。

动画

动画分为静态动画和连续动画。

静态动画

静态动画的核心是 transform 样式，主要可以实现以下三种变换类型，一次样式设置只能实现一种类型变换。

- translate：沿水平或垂直方向将指定组件移动所需距离。
- scale：横向或纵向将指定组件缩小或放大到所需比例。
- rotate：将指定组件沿横轴或纵轴或中心点旋转指定的角度。

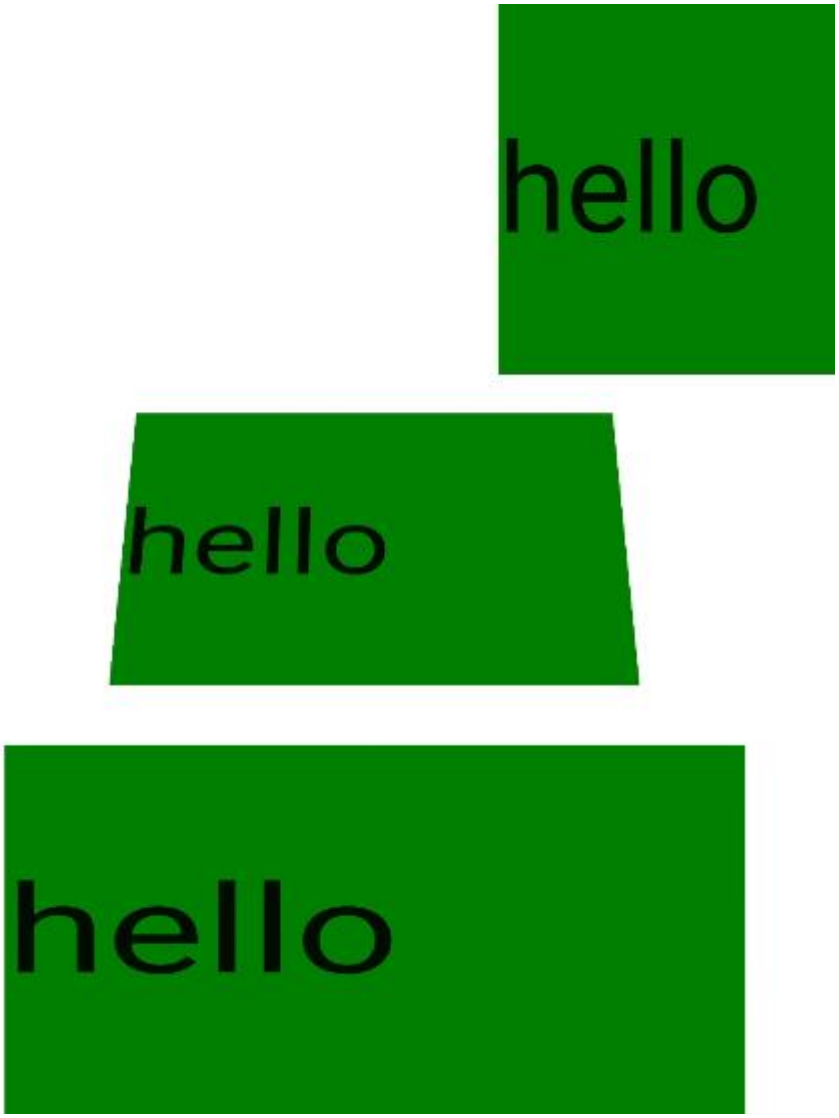
静态动画只有开始状态和结束状态，没有中间状态，如果需要设置中间的过渡状态和转换效果，需要使用连续动画实现。具体的使用示例如下，更多信息请参考组件。

```
<!-- xxx.hml -->
<div class="container">
  <text class="translate">hello</text>
  <text class="rotate">hello</text>
  <text class="scale">hello</text>
</div>
```

```
/* xxx.css */
.container {
  flex-direction: column;
  align-items: center;
}
.translate {
  height: 300px;
```

```
  width: 400px;
  font-size: 100px;
  background-color: #008000;
  transform: translate(300px);
}
.rotate {
  height: 300px;
  width: 400px;
  font-size: 100px;
  background-color: #008000;
  transform-origin: 200px 100px;
  transform: rotateX(45deg);
}
.scale {
  height: 300px;
  width: 400px;
  font-size: 100px;
  background-color: #008000;
  transform: scaleX(1.5);
}
```

图 1 静态动画效果图



连续动画

连续动画的核心是 animation 样式，它定义了动画的开始状态、结束状态以及时间和速度的变化曲线。通过 animation 样式可以实现的效果有：

- animation-name：设置动画执行后应用到组件上的背景颜色、透明度、宽高和变换类型。
- animation-delay 和 animation-duration：分别设置动画执行后元素延迟和持续的时间。
- animation-timing-function：描述动画执行的速度曲线，使动画更加平滑。
- animation-iteration-count：定义动画播放的次数。

- animation-fill-mode：指定动画执行结束后是否恢复初始状态。

animation 样式需要在 css 文件中先定义 keyframe，在 keyframe 中设置动画的过渡效果，并通过一个样式类型在 hml 文件中调用。animation-name 的使用示例如下：

```
<!-- xxx.hml -->
<div class="item-container">
  <div class="group">
    <text class="header">animation-name</text>
    <div class="item {{colorParam}}">
      <text class="txt">color</text>
    </div>
    <div class="item {{opacityParam}}">
      <text class="txt">opacity</text>
    </div>
    <input class="button" type="button" name="" value="show"
onclick="showAnimation"/>
  </div>
</div>
```

```
/* xxx.css */
.item-container {
  margin-bottom: 50px;
  margin-right: 60px;
  margin-left: 60px;
  flex-direction: column;
  align-items: flex-start;
```

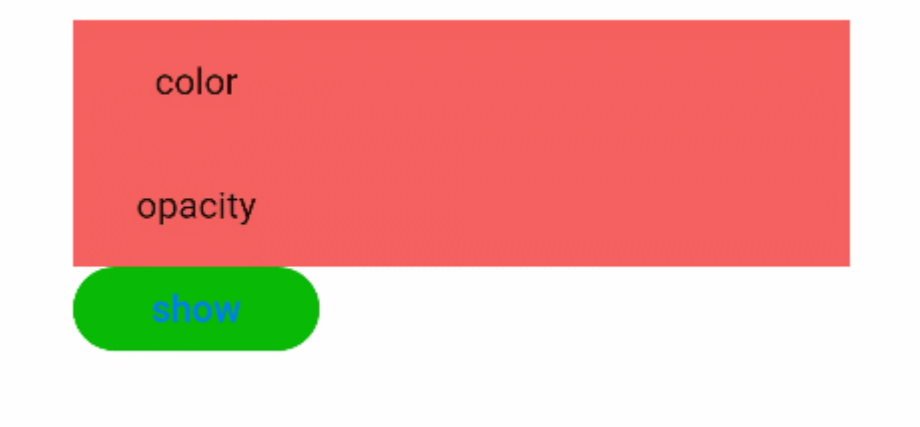
```
}
.group {
  margin-bottom: 150px;
  flex-direction: column;
  align-items: flex-start;
}
.header {
  margin-bottom: 20px;
}
.item {
  background-color: #f76160;
}
.txt {
  text-align: center;
  width: 200px;
  height: 100px;
}
.button {
  width: 200px;
  font-size: 30px;
  color: #ffffff;
  background-color: #09ba07;
}
.color {
  animation-name: Color;
  animation-duration: 8000ms;
}
.opacity {
```

```
  animation-name: Opacity;
  animation-duration: 8000ms;
}
@keyframes Color {
  from {
    background-color: #f76160;
  }
  to {
    background-color: #09ba07;
  }
}
@keyframes Opacity {
  from {
    opacity: 0.9;
  }
  to {
    opacity: 0.1;
  }
}
```

```
// xxx.js
export default {
  data: {
    colorParam: '',
    opacityParam: '',
  },
  showAnimation: function () {
```

```
    this.colorParam = '';
    this.opacityParam = '';
    this.colorParam = 'color';
    this.opacityParam = 'opacity';
  },
}
```

图 2 连续动画效果图



事件

事件主要包括手势事件和按键事件。手势事件主要用于智能穿戴等具有触摸屏的设备，按键事件主要用于智慧屏设备。

手势事件

手势表示由单个或多个事件识别的语义动作（例如：点击、拖动和长按）。一个完整的手势也可能由多个事件组成，对应手势的生命周期。JS UI 框架支持的手势事件有：

触摸

- touchstart：手指触摸动作开始。
- touchmove：手指触摸后移动。
- touchcancel：手指触摸动作被打断，如来电提醒、弹窗。
- touchend：手指触摸动作结束。

点击

click：用户快速轻敲屏幕。

长按

longpress：用户在相同位置长时间保持与屏幕接触。

具体的使用示例如下：

```
<!-- xxx.hml -->
<div class="container">
  <div class="text-container" onclick="click">
    <text class="text-style">{{onclick}}</text>
  </div>
  <div class="text-container" ontouchstart="touchStart">
    <text class="text-style">{{touchStart}}</text>
```

```
  </div>
  <div class="text-container" ontouchmove="touchMove">
    <text class="text-style">{{touchMove}}</text>
  </div>
  <div class="text-container" ontouchend="touchEnd">
    <text class="text-style">{{touchEnd}}</text>
  </div>
  <div class="text-container" ontouchcancel="touchCancel">
    <text class="text-style">{{touchCancel}}</text>
  </div>
  <div class="text-container" onlongpress="longPress">
    <text class="text-style">{{onLongPress}}</text>
  </div>
</div>
```

```
/* xxx.css */
.container {
  flex-direction: column;
  justify-content: center;
  align-items: center;
}
.text-container {
  padding-top: 10px;
  flex-direction: column;
}
.text-style {
  padding-top: 20px;
```

```
  padding-left: 100px;
  width: 750px;
  height: 100px;
  text-align: center;
  font-size: 50px;
  color: #ffffff;
  background-color: #09ba07;
}
```

```
// xxx.js
export default {
  data: {
    textData: '',
    touchStart: 'touchstart',
    touchMove: 'touchmove',
    touchEnd: 'touchend',
    touchCancel: 'touchcancel',
    onClick: 'onclick',
    onLongPress: 'onlongpress',
  },
  onInit() {
    this.textData = 'initdata';
  },
  onReady: function () {},
  onShow: function () {},
  onHide: function () {},
  onDestroy: function () {},
```

```
  touchCancel: function (event) {
    this.touchCancel = 'canceled';
  },
  touchEnd: function(event) {
    this.touchEnd = 'ended';
  },
  touchMove: function(event) {
    this.touchMove = 'moved';
  },
  touchStart: function(event) {
    this.touchStart = 'touched';
  },
  longPress: function() {
    this.onLongPress = 'longpressed';
  },
  click: function() {
    this.onClick = 'clicked';
  },
}
```

按键事件

按键事件是智慧屏上特有的手势事件，当用户操作遥控器按键时触发。用户点击一个遥控器按键，通常会触发两次 key 事件：先触发 action 为 0，再触发 action 为 1，即先触发按下事件，再触发抬起事件。action 为 2 的场景比较少见，一般为用户按下按键且不松开，此时 repeatCount 将返回次数。每个物理按键对

应各自的按键值（keycode）以实现不同的功能，常用的按键值请参考组件通用事件。具体的使用示例如下：

```
<!-- xxx.hml -->
<div class="card-box">
  <div class="content-box">
    <text class="content-text" onkey="keyUp" onfocus="focusUp"
onblur="blurUp">{{up}}</text>
  </div>
  <div class="content-box">
    <text class="content-text" onkey="keyDown" onfocus="focusDown"
onblur="blurDown">{{down}}</text>
  </div>
</div>
```

```
/* xxx.css */
.card-box {
  flex-direction: column;
  justify-content: center;
}
.content-box {
  align-items: center;
  height: 200px;
  flex-direction: column;
  margin-left: 200px;
  margin-right: 200px;
}
.content-text {
```

```
  font-size: 40px;
  text-align: center;
}
```

```
// xxx.js
export default {
  data: {
    up: 'up',
    down: 'down',
  },
  focusUp: function() {
    this.up = 'up focused';
  },
  blurUp: function() {
    this.up = 'up';
  },
  keyUp: function() {
    this.up = 'up keyed';
  },
  focusDown: function() {
    this.down = 'down focused';
  },
  blurDown: function() {
    this.down = 'down';
  },
  keyDown: function() {
    this.down = 'down keyed';
```

```
  },
}
```

按键事件通过获焦事件向下分发，因此示例中使用了 focus 事件和 blur 事件明确当前焦点的位置。点按上下键选中 up 或 down 按键，即相应的 focused 状态，失去焦点的按键恢复正常的 up 或 down 按键文本。按确认键后该按键变为 keyed 状态。

页面路由

很多应用由多个页面组成，比如用户可以从音乐列表页面点击歌曲，跳转到该歌曲的播放界面。开发者需要通过页面路由将这些页面串联起来，按需实现跳转。

页面路由 router 根据页面的 uri 来找到目标页面，从而实现跳转。以最基础的两个页面之间的跳转为例，具体实现步骤如下：

1. 创建两个页面。
2. 修改配置文件 config.json。
3. 调用 router.push()路由到详情页。
4. 调用 router.back()回到首页。

创建两个页面

创建 index 和 detail 页面，这两个页面均包含一个 text 组件和 button 组件：text 组件用来指明当前页面，button 组件用来实现两个页面之间的相互跳转。hml 文件代码示例如下：

```
<!-- index.hml -->
<div class="container">
  <div class="text-div">
    <text class="title">This is the index page.</text>
  </div>
  <div class="button-div">
    <button type="capsule" value="Go to the second page"
onclick="launch"></button>
  </div>
</div>
```

```
<!-- detail.hml -->
<div class="container">
  <div class="text-div">
    <text class="title">This is the detail page.</text>
  </div>
  <div class="button-div">
    <button type="capsule" value="Go back" onclick="launch"></button>
  </div>
</div>
```

修改配置文件

config.json 文件是配置文件，主要包含了 JS FA 页面路由信息。开发者新创建的页面都要在配置文件的 pages 标签中进行注册，处于第一位的页面为首页，即点击图标后的主页面。

```
{
...
  "pages": [
    "pages/index/index",
    "pages/detail/detail"
  ],
...
}
```

实现跳转

为了使 button 组件的 launch 方法生效，需要在页面的 js 文件中实现跳转逻辑。调用 router.push()接口将 uri 指定的页面添加到路由栈中，即跳转到 uri 指定的页面。在调用 router 方法之前，需要导入 router 模块。代码示例如下：

```
// index.js
import router from '@system.router';
export default {
  launch: function() {
    router.push ({
      uri: 'pages/detail/detail',
    });
  },
}
```

```
// detail.js
import router from '@system.router';
export default {
  launch: function() {
    router.back();
  },
}
```

焦点逻辑

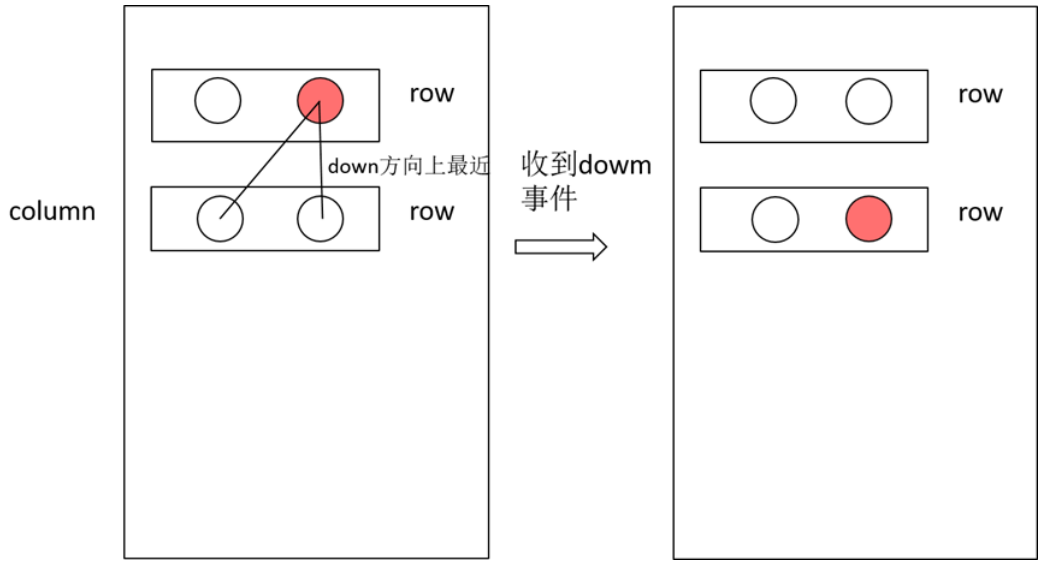
焦点移动是智慧屏的主要交互方式，本节将介绍焦点逻辑的主要规则。

- **容器组件焦点分发逻辑：**

容器组件在第一次获焦时焦点一般都落在第一个可获焦的子组件上，再次获焦时焦点落在上一次失去焦点时获焦的子组件上。容器组件一般都有特定的焦点分发逻辑，以下分别说明常用容器组件的焦点分发逻辑。

div 组件通过按键移动获焦时，焦点会移动到在移动方向上与当前获焦组件布局中心距离最近的可获焦叶子节点上。如图 1 中焦点在上方的横向 div 的第二个子组件上，当点击 down 按键时，焦点要移动到下方的横向 div 中。这时下方的横向 div 中的子组件会与当前焦点所在的组件进行布局中心距离的计算，其中距离最近的子组件获焦。

图 1　div 焦点移动时距离计算示例



- list 组件包含 list-item 与 list-item-group，list 组件每次获焦时会使第一个可获焦的 item 获焦。list-item-group 为特殊的 list-item，且两者都与 div 的焦点逻辑相同。
- stack 组件只能由自顶而下的第一个可获焦的子组件获焦。
- swiper 的每个页面和 refresh 的页面的焦点逻辑都与 div 的相同。
- tabs 组件包含 tab-bar 与 tab-content，tab-bar 中的子组件默认都能获焦，与是否有可获焦的叶子结点无关。tab-bar 与 tab-content 的每个页面都与 div 的焦点逻辑相同。

- dialog 的 button 可获焦，若有多个 button，默认初始焦点落在第二个 button 上。
- popup 无法获焦。
- **focusable 属性使用**

通用属性 focusable 主要用于控制组件能否获焦，本身不支持焦点的组件在设置此属性后可以拥有获取焦点的能力。如 text 组件本身不能获焦，焦点无法移动到它上面，设置 text 的 focusable 属性为 true 后，text 组件便可以获焦。特别的是，如果在没有使用 focusable 属性的情况下，使用了 focus，blur 或 key 事件，会默认添加 focusable 属性为 true。

容器组件是否可获焦依赖于是否拥有可获焦的子组件。如果容器组件内没有可以获焦的子组件，即使设置了 focusable 为 true，依然不能获焦。当容器组件 focusable 属性设置为 false，则它本身和它所包含的所有组件都不可获焦。

自定义组件

JS UI 框架支持自定义组件，用户可根据业务需求将已有的组件进行扩展，增加自定义的私有属性和事件，封装成新的组件，方便在工程中多次调用，提高页面布局代码的可读性。具体的封装方法示例如下：

- **构建自定义组件**

```
<!-- comp.hml -->
 <div class="item">
   <text class="title-style">{{title}}</text>
   <text class="text-style" onclick="childClicked" focusable="true">点击这里查看隐藏文本</text>
   <text class="text-style" if="{{show}}">hello world</text>
 </div>
```

```
/* comp.css */
 .item {
   width: 700px;
   flex-direction: column;
   height: 300px;
   align-items: center;
   margin-top: 100px;
 }
 .text-style {
   width: 100%;
   text-align: center;
   font-weight: 500;
   font-family: Courier;
   font-size: 36px;
```

```
}
.title-style {
  font-weight: 500;
  font-family: Courier;
  font-size: 50px;
  color: #483d8b;
}
```

```
// comp.js
export default {
  props: {
    title: {
      default: 'title',
    },
    showObject: {},
  },
  data() {
    return {
      show: this.showObject,
    };
  },
  childClicked () {
    this.$emit('eventType1', {text: '收到子组件参数'});
    this.show = !this.show;
  },
}
```

引入自定义组件

```
<!-- xxx.hml -->
 <element name='comp' src='../../common/component/comp.hml'></element>
 <div class="container">
   <text>父组件：{{text}}</text>
   <comp title="自定义组件" show-object="{{show}}"
@event-type1="textClicked"></comp>
 </div>
```

```
/* xxx.css */
 .container {
   background-color: #f8f8ff;
   flex: 1;
   flex-direction: column;
   align-content: center;
 }
```

```
// xxx.js
 export default {
   data: {
     text: '开始',
     show: false,
   },
   textClicked (e) {
     this.text = e.detail.text;
   },
 }
```

本示例中父组件通过添加自定义属性向子组件传递了名称为 title 的参数，子组件在 props 中接收，同时子组件也通过事件绑定向上传递了参数 text，接收时通过 e.detail 获取，要绑定子组件事件，父组件事件命名必须遵循事件绑定规则，详见 自定义组件开发规范。自定义组件效果如下图所示：

图 1 自定义组件静态效果

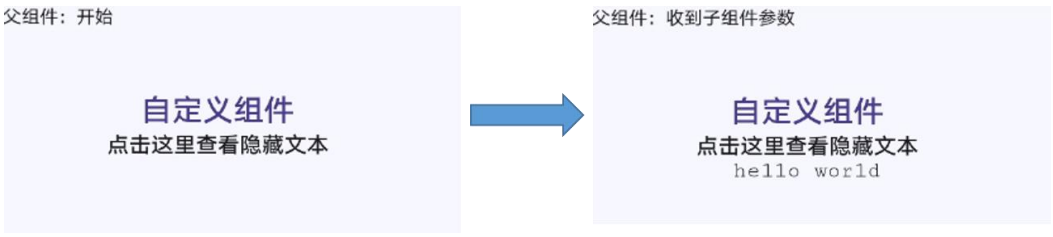


图 2 自定义组件动态效果

JS FA 如何调用 PA

JS UI 框架提供了 JS FA（Feature Ability）调用 Java PA（Particle Ability）的机制，该机制提供了一种通道来传递方法调用、数据返回以及订阅事件上报。

当前提供 Ability 和 Internal Ability 两种调用方式，开发者可以根据业务场景选择合适的调用方式进行开发。

- Ability：拥有独立的 Ability 生命周期，FA 使用远端进程通信拉起并请求 PA 服务，适用于基本服务供多 FA 调用或者服务在后台独立运行的场景。
- Internal Ability：与 FA 共进程，采用内部函数调用的方式和 FA 进行通信，适用于对服务响应时延要求较高的场景。该方式下 PA 不支持其他 FA 访问调用。

JS 端与 Java 端通过 bundleName 和 abilityName 来进行关联。在系统收到 JS 调用请求后，根据开发者在 JS 接口中设置的参数来选择对应的处理方式。开发者在 onRemoteRequest()中实现 PA 提供的业务逻辑。详细信息请参考 JS FA 调用 Java PA 机制。

FA 调用 PA 接口

FA 端提供以下三个 JS 接口：

- FeatureAbility.callAbility(OBJECT)：调用 PA 能力。
- FeatureAbility.subscribeAbilityEvent(OBJECT, Function)：订阅 PA 能力。
- FeatureAbility.unsubscribeAbilityEvent(OBJECT)：取消订阅 PA 能力。

PA 端提供以下两类接口：

- boolean IRemoteObject.onRemoteRequest(int code, MessageParcel data, MessageParcel reply, MessageOption option)：Ability 调用方式，FA 使用远端进程通信拉起并请求 PA 服务。
- boolean AceInternalAbility.AceInternalAbilityHandler.onRemoteRequest(int code, MessageParcel data, MessageParcel reply, MessageOption option)：Internal Ability 调用方式，采用内部函数调用的方式和 FA 进行通信。

FA 调用 PA 常见问题

- callAbility 返回报错："Internal ability not register."

返回该错误说明 JS 接口调用请求未在系统中找到对应的 InternalAbilityHandler 进行处理，因此需要检查以下几点是否正确执行：

- 在 AceAbility 继承类中对 AceInternalAbility 继承类执行了 register 方法，具体注册可参考 Internal Ability 的示例代码。
- JS 侧填写的 bundleName 和 abilityName 与 AceInternalAbility 继承类构造函数中填写的名称保持相同，大小写敏感。
- 检查 JS 端填写的 abilityType（0：Ability; 1：Internal Ability），确保没有将 Ability 误填写为 Internal Ability 方式。

Ability 和 Internal Ability 是两种不同的 FA 调用 PA 的方式。 表 1 列举了在开发时各方面的差异，供开发者参考，避免开发时将两者混淆使用：

表 1 Ability 和 InternalAbility 差异项

差异项	Ability	InternalAbility
JS 端（abilityType）	0	1
是否需要在 config.json 的 abilities 中为 PA 添加声明	需要（有独立的生命周期）	不需要（和 FA 共生命周期）
是否需要在 FA 中注册	不需要	需要
继承的类	ohos.aafwk.ability.Ability	ohos.ace.ability.AceInternalAbility
是否允许被其他 FA 访问调用	是	否

- FeatureAbility.callAbility 中 syncOption 参数说明：

- 对于 JS FA 侧，返回的结果都是 Promise 对象，因此无论该参数取何值，都采用异步方式等待 PA 侧响应。
- 对于 JAVA PA 侧，在 Internal Ability 方式下收到 FA 的请求后，根据该参数的取值来选择：通过同步的方式获取结果后返回；或者异步执行 PA 逻辑，获取结果后使用 remoteObject.sendRequest 的方式将结果返回 FA。
- 使用 await 方式调用时 IDE 编译报错，需引入 babel-runtime/regenerator，具体请参见接口通用规则。

示例参考

- FA JavaScript 端

```
// abilityType: 0-Ability; 1-Internal Ability
const ABILITY_TYPE_EXTERNAL = 0;
const ABILITY_TYPE_INTERNAL = 1;
// syncOption(Optional, default sync): 0-Sync; 1-Async
const ACTION_SYNC = 0;
const ACTION_ASYNC = 1;
const ACTION_MESSAGE_CODE_PLUS = 1001;
export default {
  plus: async function() {
    var actionData = {};
    actionData.firstNum = 1024;
    actionData.secondNum = 2048;

    var action = {};
    action.bundleName = 'com.huawei.hiaceservice';
    action.abilityName = 'com.huawei.hiaceservice.CalcServiceAbility';
    action.messageCode = ACTION_MESSAGE_CODE_PLUS;
    action.data = actionData;
```

```
    action.abilityType = ABILITY_TYPE_EXTERNAL;
    action.syncOption = ACTION_SYNC;

    var result = await FeatureAbility.callAbility(action);
    var ret = JSON.parse(result);
    if (ret.code == 0) {
      console.info('plus result is:' + JSON.stringify(ret.abilityResult));
    } else {
      console.error('plus error code:' + JSON.stringify(ret.code));
    }
  }
}
```

多模输入

概述

HarmonyOS 旨在为开发者提供 NUI（Natural User Interface）的交互方式，有别于传统操作系统的输入划分方式，在 HarmonyOS 上，我们将多种维度的输入整合在一起，开发者可以借助应用程序框架、系统自带的 UI 控件或 API 接口轻松地实现具有多维、自然交互特点的应用程序。

具体来说，HarmonyOS 目前不仅支持传统的输入交互方式，例如按键、触控、键盘、鼠标等，同时也支持语音等新型的输入交互方式。

约束与限制

- 多模输入事件在不同形态产品支持的情况如下表。

表 1 多模输入事件在不同形态产品支持的情况

多模输入事件	智慧屏	车机	智能穿戴
按键输入事件	支持	支持	支持
触屏输入事件	支持	支持	支持
鼠标事件	部分支持	不支持	不支持
语音事件	支持	不支持	不支持

说明

智慧屏产品对鼠标事件只支持鼠标左键事件，鼠标右键以及滚轮等事件暂不支持。

- 目前多模输入不支持生成事件（即开发者无法创建事件）和注入事件（即开发者无法模拟注入事件验证应用程序功能）。
- 使用多模输入相关功能需要获取多模输入权限：ohos.permission.MULTIMODAL_INTERACTIVE。

开发指导

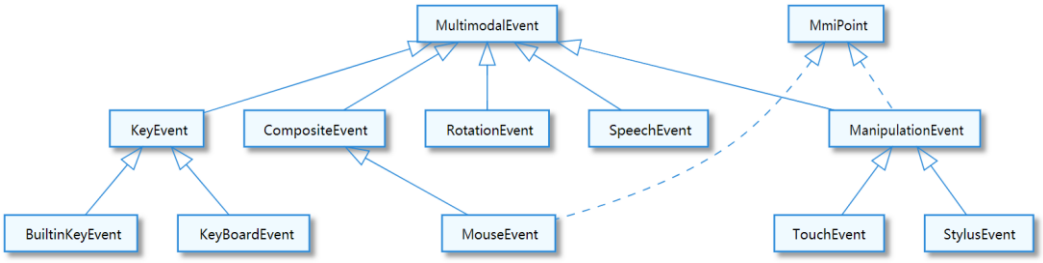
场景介绍

多模输入使 HarmonyOS 的 UI 控件能够响应多种输入事件，事件来源于用户的按键、点击、触屏、语音等。例如用户希望通过语音操作 UI 控件，那么开发者可以通过多模输入在智慧屏产品上提供的语音事件达到“可见即可说”的效果。

接口说明

多模输入的接口设计是基于多模事件基类（MultimodalEvent），派生出操作事件类（ManipulationEvent）、按键事件类（KeyEvent）、语音事件（SpeechEvent）等，详细见图 1。

图 1　多模输入事件类的派生关系



- MultimodalEvent 是所有事件的基类，该类中定义了一系列高级事件类型，这些事件类型通常是对某种行为或意图的抽象。

表 1 MultimodalEvent 的主要接口

功能分类	接口名	描述
设备信息相关	getDeviceId()	获取输入设备所在的承载设备 id，如当同时有两个鼠标连接到一个机器上，该机器为这两个鼠标的承载设备。
	getInputDeviceId()	获取产生当前事件的输入设备 id，该 id 是该输入设备的唯一标识，如两个鼠标同时输入时，它们会分别产生输入事件，且从事件中获取到的 deviceid 是

表 1 MultimodalEvent 的主要接口

功能分类	接口名	描述
		不同的，开发者可以将此 id 用来区分实际的输入设备源。
	getSourceDevice()	获取产生当前事件的输入设备类型。
时间	getOccurredTime()	获取产生当前事件的时间。
事件	getUuid()	获取事件的 UUID。
	isSameEvent(UUID id)	判断当前事件与传入 id 的事件是否为同一事件。

- CompositeEvent 处理常用设备对应的事件，目前暂时只有 MouseEvent 事件继承该类。
- RotationEvent 处理由旋转器件产生的事件，比如智能穿戴上的数字表冠。

表 2 RotationEvent 的主要接口

功能分类	接口名	描述
旋转器信息	getRotationValue()	获取旋转器件旋转产生的值。

- SpeechEvent 处理语音事件，开发者可以通过该类获取语音识别结果。

表 3 SpeechEvent 的主要接口

功能分类	接口名	描述
构造函数	public static Optional<SpeechEvent> createEvent(long	SpeechEvent 创建函数。

表 3 SpeechEvent 的主要接口

功能分类	接口名	描述
	occurTime, int action, String value)	
获取语音事件参数值	public int getAction()	获取当前动作的类型，如打开、关闭、命中热词。
	public int getScene()	获取当前动作时的场景。
	public String getActionProperty()	获取动作所携带的属性值。
	public int getMatchMode()	获取识别结果的匹配模式。

- ManipulationEvent 操作类事件主要包括手指触摸事件等事件，是对这些事件的一个抽象。该事件会持有事件发生的位置信息和发生的阶段等信息。通常情况下，该事件主要是作为操作回调接口的入参，开发者通过回调接口捕获及处理事件。回调接口将操作分为开始、操作过程中、结束。例如对于一次手指触控，手指接触屏幕作为操作开始，手指在屏幕上移动作为操作过程，手指抬起作为操作结束。

表 4 ManipulationEvent 的主要接口

功能分类	接口名	描述
手指信息	getPointerCount()	获取一次事件中触控或轨迹追踪的指针数量。
	getPointerId(int index)	获取一次事件中，指针的唯一标识 Id。
	setScreenOffset(float offsetX, float offsetY)	设置相对屏幕坐标原点的偏移位置信息。

表 4 ManipulationEvent 的主要接口

功能分类	接口名	描述
	getPointerPosition(int index)	获取一次事件中触控或轨迹追踪的某个指针相对于偏移位置的坐标信息。
	getPointerScreenPosition(int index）	获取一次事件中触控或轨迹追踪的某个指针相对屏幕坐标原点的坐标信息。
	getRadius(int index)	返回给定 index 手指与屏幕接触的半径值。
	getForce(int index)	获取给定 index 手指触控的压力值。
时间	getStartTime()	获取操作开始阶段时间。
阶段	getPhase()	事件所属阶段。

- KeyEvent 对所有按键类事件的定义，该类继承 MultimodalEvent 类，并对按键类事件做了专属的 Keycode 定义以及方法封装。

表 5 KeyEvent 的主要接口

功能分类	接口名	描述
KeyCode	getKeyCode()	获取当前按键类事件的 keycode 值。
	getMaxKeyCode()	获取当前定义的按键类事件的最大 keycode 值。
按键按下状态	getKeyDownDuration()	获取当前按键截止该接口被调用时被按下的时长。
	isKeyDown()	获取当前按键事件的按下状态。

- TouchEvent 处理手指触控相关事件。

表 6 TouchEvent 的主要接口

功能分类	接口名	描述
触控行为	getAction()	获取当前触摸行为。
	getIndex()	获取发生行为的对应指针。

- KeyBoardEvent 处理键盘类设备的事件。

表 7 KeyBoardEvent 的主要接口

功能分类	接口名	描述
输入法编辑器	enableIme()	启动输入法编辑器。
	disableIme()	关闭输入法编辑器。
	isHandledByIme()	判断输入法编辑器是否在使用。
NoncharacterKey 行为	isNoncharacterKeyPressed(int keycode)	判定输入的单个 NoncharacterKey 是否处于按下状态。
	isNoncharacterKeyPressed(int keycode1, int keycode2)	判定输入的两个 NoncharacterKey 是否都处于按下状态。
	isNoncharacterKeyPressed(int keycode1, int keycode2, int keycode3)	判定输入的三个 NoncharacterKey 是否都处于按下状态。
按键 Unicode 码	getUnicode()	获取按键对应的 Unicode 码。

说明

NoncharacterKey 为除了文本可见字符（A-Z，0-9，空格，逗号，句号等）以外的按键码，例如：Ctrl，Alt，Shift 等。

MouseEvent 处理鼠标的事件。

表 8 MouseEvent 的主要接口

功能分类	接口名	描述
鼠标行为	getAction()	获取鼠标设备产生事件的行为。
鼠标按键	getActionButton()	获取状态发生变化的鼠标按键。
	getPressedButtons()	获取所有按下状态的鼠标按键。
鼠标指针/位置	getCursor()	获取鼠标指针的位置。
	getCursorDelta(int axis)	获取鼠标指针位置相对上次的变化值。
	setCursorOffset(float offsetX, float offsetY)	设置相对屏幕的偏移位置信息。
鼠标滚轮	getScrollingDelta(int axis)	获取滚轮的滚动值。

- MmiPoint 处理在指定给定的坐标系中的 x,y 和 z 坐标。

表 9 MmiPoint 的主要接口

功能分类	接口名	描述
构造函数	MmiPoint(float px, float	创建一个只包含 x 和 y 坐

表 9 MmiPoint 的主要接口

功能分类	接口名	描述
	py)	标的 MmiPoint 对象。
	MmiPoint(float px, float py, float pz)	创建一个包含 x，y 和 z 坐标的 MmiPoint 对象。
坐标值	getX()	获取 x 坐标值。
	getY()	获取 y 坐标值。
	getZ()	获取 z 坐标值。
	toString()	返回包含 x、y、z 坐标值信息的字符串

开发步骤

处理按钮事件

1. 参考 HarmonyOS 的 Component 的 API 创建 KeyEventListener；
2. 重写实现 KeyEventListener 类中的 onKeyEvent(Component component, KeyEvent event) 方法；
3. 开发者根据自身需求处理存在按键被按下以及 KEY_DPAD_CENTER、KEY_DPAD_LEFT 等按键被按下后的具体实现。

```
private Component.KeyEventListener onKeyEvent = new
Component.KeyEventListener()

{

    @Override

    public boolean onKeyEvent(Component component, KeyEvent keyEvent) {

        if (keyEvent.isKeyDown()) {

            ... // 检测到按键被按下，开发者根据自身需求进行实现
```

```
        }
        int keycode = keyEvent.getKeyCode();
        switch (keycode) {
            case KeyEvent.KEY_DPAD_CENTER:
                ... // 检测到 KEY_DPAD_CENTER 被按下，开发者根据自身需求进行实现
                break;
            case KeyEvent.KEY_DPAD_LEFT:
                ... // 检测到 KEY_DPAD_LEFT 被按下，开发者根据自身需求进行实现
                break;
            case KeyEvent.KEY_DPAD_UP:
                ... // 检测到 KEY_DPAD_UP 被按下，开发者根据自身需求进行实现
                break;
            case KeyEvent.KEY_DPAD_RIGHT:
                ... // 检测到 KEY_DPAD_RIGHT 被按下，开发者根据自身需求进行实现
                break;
            case KeyEvent.KEY_DPAD_DOWN:
                ... // 检测到 KEY_DPAD_DOWN 被按下，开发者根据自身需求进行实现
                break;
            default:
                break;
        }
        ...
    }
};
```

处理语音事件

使用多模输入的语音事件实现“可见即可说”的效果简易开发样例参考可见即可说开发指导。

媒体

视频

概述

HarmonyOS 视频模块支持视频业务的开发和生态开放，开发者可以通过已开放的接口很容易地实现视频媒体的播放、操作和新功能开发。视频媒体的常见操作有视频编解码、视频合成、视频提取、视频播放以及视频录制等。

基本概念

- **编码**

编码是信息从一种形式或格式转换为另一种形式的过程。用预先规定的方法将文字、数字或其他对象编成数码，或将信息、数据转换成规定的电脉冲信号。在本模块中，编码是指编码器将原始的视频信息压缩为另一种格式的过程。

- **解码**

解码是一种用特定方法，把数码还原成它所代表的内容或将电脉冲信号、光信号、无线电波等转换成它所代表的信息、数据等的过程。在本模块中，解码是指解码器将接收到的数据还原为视频信息的过程，与编码过程相对应。

- **帧率**

帧率是以帧称为单位的位图图像连续出现在显示器上的频率（速率），以赫兹（Hz）为单位。该术语同样适用于胶片、摄像机、计算机图形和动作捕捉系统。

媒体编解码能力查询开发指导

场景介绍

媒体编解码能力查询主要指查询设备所支持的编解码器的 MIME（Multipurpose Internet Mail Extensions，媒体类型）列表，并判断设备是否支持指定 MIME 对应的编码器/解码器。

接口说明

表 1 媒体编解码能力查询类 CodecDescriptionList 的主要接口

接口名	功能描述
getSupportedMimes()	获取某设备所支持的编解码器的 MIME 列表。
isDecodeSupportedByMime(String mime)	判断某设备是否支持指定 MIME 对应的解码器。
isEncodeSupportedByMime(String mime)	判断某设备是否支持指定 MIME 对应的编码器。
isDecoderSupportedByFormat(Format format)	判断某设备是否支持指定媒体格式对应的解码器。
isEncoderSupportedByFormat(Format format)	判断某设备是否支持指定媒体格式对应的编码器。

开发步骤

1. 调用 CodecDescriptionList 类的静态 getSupportedMimes()方法，获取某设备所支持的编解码器的 MIME 列表。代码示例如下：

```
List<String> mimes = CodecDescriptionList.getSupportedMimes();
```

调用 CodecDescriptionList 类的静态 isDecodeSupportedByMime 方法，判断某设备是否支持指定 MIME 对应的解码器，支持返回 true，否则返回 false。代码示例如下：

```
boolean result =
CodecDescriptionList.isDecodeSupportedByMime(Format.VIDEO_VP9);
```

调用 CodecDescriptionList 类的静态 isEncodeSupportedByMime 方法，判断某设备是否支持指定 MIME 对应的编码器，支持返回 true，否则返回 false。代码示例如下：

```
boolean result =
CodecDescriptionList.isEncodeSupportedByMime(Format.AUDIO_FLAC);
```

调用 CodecDescriptionList 类的静态 isDecoderSupportedByFormat/isEncoderSupportedByFormat 方法，判断某设备是否支持指定 Format 的编解码器，支持返回 true，否则返回 false。代码示例如下：

```
Format format = new Format();

format.putStringValue(Format.MIME, Format.VIDEO_AVC);

format.putIntValue(Format.WIDTH, 2560);

format.putIntValue(Format.HEIGHT, 1440);

format.putIntValue(Format.FRAME_RATE, 30);

format.putIntValue(Format.FRAME_INTERVAL, 1);

boolean result = CodecDescriptionList.isDecoderSupportedByFormat(format);

result = CodecDescriptionList.isEncoderSupportedByFormat(format);
```

视频编解码开发指导

场景介绍

视频编解码的主要工作是将视频进行编码和解码。

接口说明

表 1 视频编解码类 Codec 的主要接口

接口名	功能描述
createDecoder()	创建解码器 Codec 实例。
createEncoder()	创建编码器 Codec 实例。
registerCodecListener(ICodecListener listener)	注册侦听器用来异步接收编码或解码后的数据。
setSource(Source source, TrackInfo trackInfo)	根据解码器的源轨道信息设置数据源，对于编码器 trackInfo 无效。
setSourceFormat(Format format)	编码器的管道模式下，设置编码器编码格式。
setCodecFormat(Format format)	普通模式设置编/解码器参数。
setVideoSurface(Surface surface)	设置解码器的 Surface。
getAvailableBuffer(long timeout)	普通模式获取可用 ByteBuffer。
writeBuffer(ByteBuffer buffer, BufferInfo info)	推送源数据给 Codec。

表 1 视频编解码类 Codec 的主要接口

接口名	功能描述
getBufferFormat(ByteBuffer buffer)	获取输出 Buffer 数据格式。
start()	启动编/解码。
stop()	停止编/解码。
release()	释放所有资源。

普通模式开发步骤

在普通模式下进行编解码，应用必须持续地传输数据到 Codec 实例。

编码的具体开发步骤如下：

1. 创建编码 Codec 实例，可调用 createEncoder()创建。

```
final Codec encoder = Codec.createEncoder();
```

构造数据源格式，并设置给 Codec 实例，调用 setCodecFormat()，代码示例如下：

```
Format fmt = new Format();
fmt.putStringValue(Format.MIME, Format.VIDEO_AVC);
fmt.putIntValue(Format.WIDTH, 1920);
fmt.putIntValue(Format.HEIGHT, 1080);
fmt.putIntValue(Format.BIT_RATE, 392000);
fmt.putIntValue(Format.FRAME_RATE, 30);
fmt.putIntValue(Format.FRAME_INTERVAL, -1);
```

```
codec.setCodecFormat(fmt);
```

如果需要编码过程中，检测是否读取到 Buffer 数据以及是否发生异常，可以构造 ICodecListener，ICodecListener 需要实现两个方法，实现读到 Buffer 数据时、编码发生异常时做相应的操作。举例中读到 buffer 时，获取 buffer 的 format 格式，异常时抛出运行时异常，代码示例如下：

```
Codec.ICodecListener listener = new Codec.ICodecListener() {
    @Override
    public void onReadBuffer(ByteBuffer byteBuffer, BufferInfo bufferInfo,
int trackId) {
        Format fmt = codec.getBufferFormat(byteBuffer);
    }

    @Override
    public void onError(int errorCode, int act, int trackId) {
        throw new RuntimeException();
    }
};
```

1. 调用 start()方法开始编码。
2. 调用 getAvailableBuffer()取到一个可用的 ByteBuffer，把数据填入 ByteBuffer 里，然后再调用 writeBuffer()把 ByteBuffer 写入编码器实例。
3. 调用 stop()方法停止编码。
4. 编码任务结束后，调用 release()释放资源。

解码的具体开发步骤如下：

1. 创建解码 Codec 实例，可调用 createDecoder()创建。
2. 构造数据源格式，并设置给 Codec 实例，调用 setCodecFormat()，代码示例如下：

```
Format fmt = new Format();

fmt.putStringValue(Format.MIME, Format.VIDEO_AVC);

fmt.putIntValue(Format.WIDTH, 1920);

fmt.putIntValue(Format.HEIGHT, 1080);

fmt.putIntValue(Format.BIT_RATE, 392000);

fmt.putIntValue(Format.FRAME_RATE, 30);

fmt.putIntValue(Format.FRAME_INTERVAL, -1);

codec.setCodecFormat(fmt);
```

（可选）如果需要解码过程中，检测是否读取到 Buffer 数据以及是否发生异常，可以构造 ICodecListener，ICodecListener 需要实现两个方法，实现读到 Buffer 数据时、解码发生异常时做相应的操作。举例中读到 buffer 时，获取 buffer 的 format 格式，异常时抛出运行时异常，代码示例如下：

```
Codec.ICodecListener listener = new Codec.ICodecListener() {

    @Override

    public void onReadBuffer(ByteBuffer byteBuffer, BufferInfo bufferInfo,
int trackId) {

        Format fmt = codec.getBufferFormat(byteBuffer);

    }


    @Override

    public void onError(int errorCode, int act, int trackId) {

        throw new RuntimeException();

    }
```

```
};
```

1. 调用 start()方法开始解码。
2. 调用 getAvailableBuffer 取到一个可用的 ByteBuffer，把数据填入 ByteBuffer 里，然后再调用 writeBuffer 把 ByteBuffer 写入解码器实例。
3. 调用 stop()方法停止解码。
4. 解码任务结束后，调用 release()释放资源。

管道模式开发步骤

管道模式下应用只需要调用 Source 类的 setSource()方法，数据会自动解析并传输给 Codec 实例。管道模式编码支持视频流编码和音频流编码。

编码的具体开发步骤如下：

1. 调用 createEncoder()创建编码 Codec 实例。
2. 调用 setSource()设置数据源，支持设定文件路径或者文件 File Descriptor。
3. 构造数据源格式或者从 Extractor 中读取数据源格式，并设置给 Codec 实例，调用 setSourceFormat()，构造数据原格式代码示例如下：

```
Format fmt = new Format();

fmt.putStringValue(Format.MIME, Format.VIDEO_AVC);

fmt.putIntValue(Format.WIDTH, 1920);

fmt.putIntValue(Format.HEIGHT, 1080);

fmt.putIntValue(Format.BIT_RATE, 392000);

fmt.putIntValue(Format.FRAME_RATE, 30);

fmt.putIntValue(Format.FRAME_INTERVAL, -1);

codec.setSourceFormat(fmt);
```

（可选）如果需要编码过程中，检测是否读取到 Buffer 数据以及是否发生异常，可以构造 ICodecListener，ICodecListener 需要实现两个方法，实现读到 Buffer

数据时、编码发生异常时做相应的操作。举例中读到 buffer 时，获取 buffer 的 format 格式，异常时抛出运行时异常，代码示例如下：

```
Codec.ICodecListener listener = new Codec.ICodecListener() {
    @Override
    public void onReadBuffer(ByteBuffer byteBuffer, BufferInfo bufferInfo, int trackId) {
        Format fmt = codec.getBufferFormat(byteBuffer);
    }

    @Override
    public void onError(int errorCode, int act, int trackId) {
        throw new RuntimeException();
    }
};
```

1. 调用 start()方法开始编码。
2. 调用 stop()方法停止编码。
3. 编码任务结束后，调用 release()释放资源。

解码的具体开发步骤如下：

1. 调用 createDecoder()创建解码 Codec 实例。
2. 调用 setSource()设置数据源，支持设定文件路径或者文件 File Descriptor。
3. （可选）如果需要解码过程中，检测是否读取到 Buffer 数据以及是否发生异常，可以构造 ICodecListener，ICodecListener 需要实现两个方法，实现读到 Buffer 数据时、解码发生异常时做相应的操作。举例中读到 buffer 时，获取 buffer 的 format 格式，异常时抛出运行时异常，代码示例如下：

```
Codec.ICodecListener listener = new Codec.ICodecListener() {

    @Override

    public void onReadBuffer(ByteBuffer byteBuffer, BufferInfo bufferInfo,
int trackId) {

        Format fmt = codec.getBufferFormat(byteBuffer);

    }



    @Override

    public void onError(int errorCode, int act, int trackId) {

        throw new RuntimeException();

    }

};
```

1. 调用 start()方法开始解码。
2. 调用 stop()方法停止解码。
3. 解码任务结束后，调用 release()释放资源。

视频播放开发指导

场景介绍

视频播放包括播放控制、播放设置和播放查询，如播放的开始/停止、播放速度设置和是否循环播放等。

接口说明

表 1 视频播放类 Player 的主要接口

接口名	功能描述
Player(Context context)	创建 Player 实例。
setSource(Source source)	设置媒体源。
prepare()	准备播放。
play()	开始播放。
pause()	暂停播放。
stop()	停止播放。
rewindTo(long microseconds)	拖拽播放。
setVolume(float volume)	调节播放音量。
setVideoSurface(Surface surface)	设置视频播放的窗口。
enableSingleLooping(boolean looping)	设置为单曲循环。
isSingleLooping()	检查是否单曲循环播放。

表 1 视频播放类 Player 的主要接口

接口名	功能描述
isNowPlaying()	检查是否播放。
getCurrentTime()	获取当前播放位置。
getDuration()	获取媒体文件总时长。
getVideoWidth()	获取视频宽度。
getVideoHeight()	获取视频高度。
setPlaybackSpeed(float speed)	设置播放速度。
getPlaybackSpeed()	获取播放速度。
setAudioStreamType(int type)	设置音频类型。
getAudioStreamType()	获取音频类型。
setNextPlayer(Player next)	设置当前播放结束后的下一个播放器。
reset()	重置播放器。
release()	释放播放资源。
setPlayerCallback(IPlayerCallback callback)	注册回调，接收播放器的事件通知或异常通知。

开发步骤

1. 创建 Player 实例，可调用 Player(Context context)，创建本地播放器，用于在本设备播放。
2. 构造数据源对象，并调用 Player 实例的 setSource(Source source)方法，设置媒体源，代码示例如下：

```
Player impl = new Player(context);

File file = new File("/path/test_audio.aac");

in = new FileInputStream(file);

FileDescriptor fd = in.getFD(); // 从输入流获取 FD 对象

Source source = new Source(fd);

impl.setSource(source);
```

1. 调用 prepare()，准备播放。
2. （可选）构造 IPlayerCallback，IPlayerCallback 需要实现 onPlayBackComplete 和 onError(int errorType, int errorCode)两个方法，实现播放完成和播放异常时做相应的操作。代码示例如下：

```
@Override

public void onPlayBackComplete() {

    HiLog.info("[PlayerCallback]", "onPlayBackComplete");


    if (impl != null) {

        impl.stop();

        impl = null;

    }

}


@Override

public void onError(int errorType, int errorCode) {

    HiLog.error("[PlayerCallback]", "onError");

}
```

1. 调用 play()方法，开始播放。
2. （可选）调用 pause()方法和 resume()方法，可以实现暂停和恢复播放。
3. （可选）调用 rewindTo(long microseconds)方法实现播放中的拖拽功能。
4. （可选）调用 getDuration()方法和 getCurrentTime()方法，可以实现获取总播放时长

以及当前播放位置功能。

5. 调用 stop()方法停止播放。
6. 播放结束后，调用 release()释放资源。

视频录制开发指导

场景介绍

视频录制的主要工作是选择视频/音频来源后，录制并生成视频/音频文件。

接口说明

表 1 视频录制类 Recorder 的主要接口

接口名	功能描述
Recorder()	创建 Recorder 实例。
setSource(Source source)	设置音视频源。
setAudioProperty(AudioProperty property)	设置音频属性。
setVideoProperty(VideoProperty property)	设置视频属性。
setStorageProperty(StorageProperty property)	设置音视频存储属性。
prepare()	准备录制资源。
start()	开始录制。
stop()	停止录制。
pause()	暂停录制。
resume()	恢复录制。
reset()	重置录制。

表 1 视频录制类 Recorder 的主要接口

接口名	功能描述
setRecorderLocation(float latitude, float longitude)	设置视频的经纬度。
setOutputFormat(int outputFormat)	设置输出文件格式。
getVideoSurface()	获取视频窗口。
setRecorderProfile(RecorderProfile profile)	设置媒体录制配置信息。
registerRecorderListener(IRecorderListener listener)	注册媒体录制回调。
release()	释放媒体录制资源。

开发步骤

1. 调用 Recorder()方法，创建 Recorder 实例。
2. 调用 setOutputFormat(int outputFormat)方法，设置录制文件存储格式。
3. 构造数据源对象，并调用 Recorder 实例的 setSource(Source source)方法，设置媒体源，代码示例如下：

```
Recorder recorder = new Recorder();

Source source = new Source();

source.setRecorderAudioSource(Recorder.AudioSource.DEFAULT);

recorder.setSource(source);
```

（可选）构造音频属性 AudioProperty 对象（不设置音频则是只录视频），并调用 Recorder 实例的 setAudioProperty(AudioProperty property)方法，设置录制的音频属性，代码示例如下：

```
Recorder recorder = new Recorder();
Source source = new Source();
source.setRecorderAudioSource(Recorder.AudioSource.DEFAULT);
recorder.setSource(source);
```

（可选）构造音频属性 AudioProperty 对象（不设置音频则是只录视频），并调用 Recorder 实例的 setAudioProperty(AudioProperty property)方法，设置录制的音频属性，代码示例如下：

```
final int AUDIO_NUM_CHANNELS_STEREO = 2;
final int AUDIO_SAMPLE_RATE_HZ = 8000;
AudioProperty audioProperty = new AudioProperty.Builder()
                    .setRecorderNumChannels(AUDIO_NUM_CHANNELS_STEREO)
                    .setRecorderSamplingRate(AUDIO_SAMPLE_RATE_HZ)
                    .setRecorderAudioEncoder(Recorder.AudioEncoder.DEFAUL
T)
                    .build();
recorder.setAudioProperty(audioProperty);
```

构造存储属性 StorageProperty 对象，并调用 Recorder 实例的 setStorageProperty(StorageProperty property)方法，设置录制的存储属性，代码示例如下：

```
String path = "/path/audiotestRecord.mp4";
StorageProperty storageProperty = new StorageProperty.Builder()
                    .setRecorderPath(path)
                    .setRecorderMaxDurationMs(-1)
                    .setRecorderMaxFileSizeBytes(-1)
                    .build();
```

```
recorder.setStorageProperty(storageProperty);
```

（可选）构造视频属性 VideoProperty 对象，并调用 Recorder 实例的 setVideoProperty(VideoProperty property)方法，设置录制的视频属性，代码示例如下：

```
VideoProperty videoProperty = new VideoProperty.Builder()
                    .setRecorderVideoEncoder(Recorder.VideoEncoder.DEFAUL
T)
                    .setRecorderWidth(1080)
                    .setRecorderDegrees(0)
                    .setRecorderHeight(800)
                    .setRecorderBitRate(10000000)
                    .setRecorderRate(30)
                    .build();
recorder.setVideoProperty(videoProperty);
```

1. 调用 prepare()，准备录制。
2. （可选）构造录制回调，首先构造对象 IRecorderListener，IRecorderListener 需要实现 onError(int what, int extra)，实现录制过程收到错误信息时做相应的操作。下面的代码例子中录制异常时，打印了相关的日志信息，代码示例如下：

```
IRecorderListener listener = new RecorderErrorAndInfoListener() {
    @Override
    public void onError(int what, int extra) {
        HiLog.error("EncodeWriteFileListener onError what:%{public}d,
extra:%{public}d", what, extra);
    }
}
```

1. 调用 start()方法，开始录制。
2. （可选）调用 pause()方法和 resume()方法，可以实现暂停和恢复录制。
3. 调用 stop()方法停止录制。

4. 录制结束后，调用 release()释放资源。

视频提取开发指导

场景介绍

视频提取主要工作是将多媒体文件中的音视频数据进行分离，提取出音频、视频数据源。

接口说明

表 1 视频提取类 Extractor 的主要接口

接口名	功能描述
Extractor()	创建 Extractor 实例。
setSource(Source source)	设置媒体播放源。
getStreamFormat(int id)	获取对应索引的轨道数据的格式。
getTotalStreams()	获取媒体文件中总轨道数。
selectStream(int id)	根据轨道号选择媒体文件中对应的轨道。
unselectStream(int id)	取消轨道选择。
rewindTo(long microseconds, int mode)	根据时间和 mode 跳转到指定帧。
next()	跳转到下一帧。
readBuffer(ByteBuffer buf, int offset)	读取解复用后的数据。
getStreamId()	获取当前轨道号。
getFrameTimestamp()	获取当前媒体数据帧的时间戳。

表 1 视频提取类 Extractor 的主要接口

接口名	功能描述
getFrameSize()	获取当前媒体数据帧的数据大小。
getFrameType()	获取当前媒体数据帧的 flags。
release()	释放资源。

开发步骤

1. 调用 Extractor()方法创建 Extractor 实例。
2. 构造数据源对象，并调用 Extractor 实例的 setSource(Source source)方法，设置媒体源，代码示例如下：

```
Extractor extractor = new Extractor();

FileDescriptor fd = in.getFD();

Source source = new Source(fd);

extractor.setSource(source);
```

1. 调用 getTotalStreams()方法获取媒体的轨道数量。
2. 调用 selectStream(int id)方法选择特定轨道的数据，进行提取。
3. （可选）调用 unselectStream(int id)方法取消选择轨道。
4. （可选）调用 rewindTo(long microseconds, int mode)方法实现提取过程中的跳转指定位置。
5. 调用 readBuffer(ByteBuffer buf, int offset)方法，可以实现获取提取出来的 Buffer 数据功能。
6. 调用 next()方法，实现提取下一帧的功能。
7. （可选）调用 getMediaStreamId()方法，可以实现获取当前选择的轨道编号的功能。
8. （可选）调用 getFrameTimestamp()方法，可以实现获取当前轨道内媒体数据帧时间戳的功能。
9. （可选）调用 getFrameSize()方法，可以实现获取当前轨道的媒体数据帧大小的功能。
10. （可选）调用 getFrameType()方法，可以实现获取当前轨道的媒体数据帧 flags 的功能。
11. 提取结束后，调用 release()释放资源。

媒体描述信息开发指导

场景介绍

媒体描述信息主要工作是支持多媒体的相关描述信息的存取。

接口说明

表 1 媒体描述信息类 AVDescription 的主要接口

接口名	功能描述
getMediaId()	获取媒体标识。
getTitle()	获取媒体标题。
getSubTitle()	获取媒体副标题。
getDescription()	获取媒体描述信息。
getIcon()	获取媒体图标。
getIconUri()	获取媒体图标的 Uri。
getExtras()	获取媒体添加的额外信息，例如应用和系统使用的内部信息。
getMediaUri()	获取媒体内容的 Uri。
marshalling(Parcel parcel)	将一个 AVDescription 对象写入到 Parcel 对象。
unmarshalling(Parcel parcel)	将一个 Parcel 对象写入到 AVDescription 对象。

表 2 媒体描述信息内部静态类 AVDescription.Builder 的主要接口

接口名	功能描述
setMediaId(String mediaId)	设置媒体标识。
setTitle(CharSequence title)	设置媒体标题。
setSubTitle(CharSequence subTitle)	设置媒体副标题。
setDescription(String description)	设置媒体描述信息。
setIcon(PixelMap icon)	设置媒体图标。
setIconUri(Uri iconUri)	设置媒体图标的 Uri。
setExtras(PacMap extras)	设置媒体的额外信息，例如应用和系统使用的内部信息。
setIMediaUri(Uri mediaUri)	设置媒体的 Uri。
build()	构造方法。

开发步骤

1. 调用 AVDescription.Builder 类的 build 方法创建 AVDescription 实例。代码示例如下：

```
AVDescription avDescription = new AVDescription.Builder().setExtras(null)
            .setMediaId("1")
            .setDescription("Description")
            .setIconUri(iconUri)
            .setIMediaUri(mediaUri)
            .setExtras(pacMap)
            .setIcon(pixelMap)
```

```
            .setTitle("title")
            .setSubTitle("subTitle")
            .build();
```

（可选）根据已有的 AVDescription 对象，可以获取媒体的描述信息，如获取媒体 Uri，代码示例如下：

```
Uri uri = avDescription.getMediaUri();
```

（可选）根据已有的 AVDescription 对象，可以将媒体的描述信息写入 Parcel 对象，代码示例如下：

```
Parcel parcel = Parcel.create();
boolean result = avDescription.marshalling(parcel);
```

（可选）根据已有的 Parcel 对象，可以读取到 AVDescription 对象，实现媒体描述信息的写入，代码示例如下：

```
boolean result = avDescription.unmarshalling(parcel);
```

媒体元数据开发指导

场景介绍

媒体元数据主要用于媒体数据的存放和读取，包含诸如媒体资源的描述、创建日期、作者、封面图片等等。

接口说明

表 1 媒体元数据存放类 AVMetadata.Builder 的主要接口

接口名	功能描述
Builder()	媒体元数据构造器的构造函数。
Builder(AVMetadata source)	媒体元数据构造器的带参构造函数。
setText(String key, CharSequence value)	用于存储媒体标题等信息。
setString(String key, String value)	用于存储媒体作者、艺术家、描述等。
setLong(String key, long value)	用于存储媒体 ID、媒体时长等信息。
setPixelMap(String key, PixelMap value)	用于存储媒体元数据相关的图片资源。
build()	媒体元数据生成函数。

表 2 媒体元数据类 AVMetadata 的主要接口

接口名	功能描述
hasKey(String key)	媒体元数据中是否包含某一个 key 的数据。
getText(String key)	获取 text 类型的 key 的数据，比如获取媒体标题等信息。

表 2 媒体元数据类 AVMetadata 的主要接口

接口名	功能描述
getString(String key)	获取 String 类型 key 的数据，比如获取媒体作者、艺术家、描述等。
getLong(String key)	获取 Long 类型 key 数据，比如获取媒体 ID、媒体时长等信息。
getKeysSet()	获取媒体元数据的集合。
getPixelMap(String key)	获取 PixelMap 类型 key 数据，获取媒体元数据相关的图片资源。
marshalling(Parcel in)	将一个 AVMetadata 对象写入到 Parcel 对象。
getAVDescription()	获取媒体的简要描述信息。

开发步骤

1. 调用 AVMetadata.Builder 类的 build 方法创建 AVMetadata 实例。代码示例如下：

```
AVMetadata avMetadata = new
AVMetadata.Builder().setString(AVMetadata.AVTextKey.MEDIA_ID,
"illuminate.mp3")
                .setString(AVMetadata.AVTextKey.TITLE, "title")
                .setString(AVMetadata.AVTextKey.ARTIST, "artist")
                .setString(AVMetadata.AVTextKey.ALBUM, "album")
                .setString(AVMetadata.AVTextKey.DISPLAY_SUBTITLE,
"display_subtitle")
                .setPixelMap(AVMetadata.AVPixelMapKey.DISPLAY_ICON_URI,
pixelmap)
                .build();
```

(可选)根据已有的 AVMetadata 对象，可以获取媒体元数据信息，如获取媒体标题等，代码示例如下：

```
String title = avMetadata.getString(AVMetadata.AVTextKey.TITLE);
```

我们需要结合 AVSession 使用，将已有的媒体元数据 AVMetadata 对象下发给应用，具体参考 AVSession 使用，示例如下：

```
mediaSession.setAVMetadata(avMetadata);
```

应用获取媒体元数据一般结合 AVControllerCallback 相关类使用，通过 onAVMetadataChanged 回调获取媒体元数据。

```
public class Callback extends AVControllerCallback {
    @Override
    public void onAVMetadataChanged(AVMetadata metadata) {
        // 歌曲信息回调
        AVDescription description = metadata.getAVDescription();
        // 获取标题
        String title = description.getTitle().toString();
        CharSequence sequence =
metadata.getText(AVMetadata.AVTextKey.TITLE);
        if (sequence != null) {
            title =
metadata.getText(AVMetadata.AVTextKey.TITLE).toString();
        }
        // 设置媒体 title
        musicTitle.setText(title);
        // 获取曲目专封面
```

```
        PixelMap iconPixelMap = description.getIcon();
        // 设置歌曲封面图
        musicCover.setPixelMap(iconPixelMap);
    }
}
```

图像

概述

HarmonyOS 图像模块支持图像业务的开发，常见功能如图像解码、图像编码、基本的位图操作、图像编辑等。当然，也支持通过接口组合来实现更复杂的图像处理逻辑。

基本概念

- **图像解码**

图像解码就是不同的存档格式图片（如 JPEG、PNG 等）解码为无压缩的位图格式，以方便在应用或者系统中进行相应的处理。

- **PixelMap**

PixelMap 是图像解码后无压缩的位图格式，用于图像显示或者进一步的处理。

- **渐进式解码**

渐进式解码是在无法一次性提供完整图像文件数据的场景下，随着图像文件数据的逐步增加，通过多次增量解码逐步完成图像解码的模式。

- **预乘**

预乘时，RGB 各通道的值被替换为原始值乘以 Alpha 通道不透明的比例（0~1）后的值，方便后期直接合成叠加；不预乘指 RGB 各通道的数值是图像的原始值，与 Alpha 通道的值无关。

- **图像编码**

图像编码就是将无压缩的位图格式，编码成不同格式的存档格式图片（JPEG、PNG 等），以方便在应用或者系统中进行相应的处理。

约束与限制

为及时释放本地资源，建议在图像解码的 ImageSource 对象、位图图像 PixelMap 对象或图像编码的 ImagePacker 对象使用完成后，主动调用 release()方法。

图像解码开发指导

场景介绍

图像解码就是将所支持格式的存档图片解码成统一的 PixelMap 图像，用于后续图像显示或其他处理，比如旋转、缩放、裁剪等。当前支持格式包括 JPEG、PNG、GIF、HEIF、WebP、BMP。

接口说明

ImageSource 主要用于图像解码。

表 1 ImageSource 的主要接口

接口名	描述
create(String pathName, SourceOptions opts)	从图像文件路径创建图像数据源。
create(InputStream is, SourceOptions opts)	从输入流创建图像数据源。
create(byte[] data, SourceOptions opts)	从字节数组创建图像源。
create(byte[] data, int offset, int length, SourceOptions opts)	从字节数组指定范围创建图像源。
create(File file, SourceOptions opts)	从文件对象创建图像数据源。
create(FileDescriptor fd, SourceOptions opts)	从文件描述符创建图像数据源。
createIncrementalSource(SourceOptions opts)	创建渐进式图像数据源。
createIncrementalSource(IncrementalSourceOptions opts)	创建渐进式图像数据源，支持设置渐进式数据更新模式。

表 1 ImageSource 的主要接口

接口名	描述
createPixelmap(DecodingOptions opts)	从图像数据源解码并创建 PixelMap 图像。
createPixelmap(int index, DecodingOptions opts)	从图像数据源解码并创建 PixelMap 图像，如果图像数据源支持多张图片的话，支持指定图像索引。
updateData(byte[] data, boolean isFinal)	更新渐进式图像源数据。
updateData(byte[] data, int offset, int length, boolean isFinal)	更新渐进式图像源数据，支持设置输入数据的有效数据范围。
getImageInfo()	获取图像基本信息。
getImageInfo(int index)	根据特定的索引获取图像基本信息。
getSourceInfo()	获取图像源信息。
release()	释放对象关联的本地资源。

普通解码开发步骤

创建图像数据源 ImageSource 对象，可以通过 SourceOptions 指定数据源的格式信息，此格式信息仅为给解码器的提示，正确提供能帮助提高解码效率，如果不设置或设置不正确，会自动检测正确的图像格式。不使用该选项时，可以将 create 接口传入的 SourceOptions 设置为 null。

```
ImageSource.SourceOptions srcOpts = new ImageSource.SourceOptions();

srcOpts.formatHint = "image/png";

String pathName = "/path/to/image.png";

ImageSource imageSource = ImageSource.create(pathName, srcOpts);
```

```
ImageSource imageSourceNoOptions = ImageSource.create(pathName, null);
```

设置解码参数，解码获取 PixelMap 图像对象，解码过程中同时支持图像处理操作。设置 desiredRegion 支持按矩形区域裁剪，如果设置为全 0，则不进行裁剪。设置 desiredSize 支持按尺寸缩放，如果设置为全 0，则不进行缩放。设置 rotateDegrees 支持旋转角度，以图像中心点顺时针旋转。如果只需要解码原始图像，不使用该选项时，可将给 createPixelMap 传入的 DecodingOptions 设置为 null。

```
// 普通解码叠加旋转、缩放、裁剪
ImageSource.DecodingOptions decodingOpts = new
ImageSource.DecodingOptions();
decodingOpts.desiredSize = new Size(100, 2000);
decodingOpts.desiredRegion = new Rect(0, 0, 100, 100);
decodingOpts.rotateDegrees = 90;
PixelMap pixelMap = imageSource.createPixelmap(decodingOpts);

// 普通解码
PixelMap pixelMapNoOptions = imageSource.createPixelmap(null);
```

解码完成获取到 PixelMap 对象后，可以进行后续处理，比如渲染显示等。

渐进式解码开发步骤

创建渐进式图像数据源 ImageSource 对象，可以通过 SourceOptions 指定数据源的格式信息，此格式信息仅为提示，如果填写不正确，会自动检测正确的图像格式，使用 IncrementalSourceOptions 指定图像数据的更新方式为渐进式更新。

```
ImageSource.SourceOptions srcOpts = new ImageSource.SourceOptions();

srcOpts.formatHint = "image/jpeg";

ImageSource.IncrementalSourceOptions incOpts = new
ImageSource.IncrementalSourceOptions();

incOpts.opts = srcOpts;

incOpts.mode = ImageSource.UpdateMode.INCREMENTAL_DATA;

imageSource = ImageSource.createIncrementalSource(incOpts);
```

渐进式更新数据，在未获取到全部图像时，支持先更新部分数据来尝试解码，更新数据时设置 isFinal 为 false，当获取到全部数据后，最后一次更新数据时设置 isFinal 为 true，表示数据更新完毕。设置解码参数同普通解码。

```
// 获取到一定的数据时尝试解码

imageSource.updateData(data, 0, bytes, false);

ImageSource.DecodingOptions decodingOpts = new
ImageSource.DecodingOptions();
```

```
PixelMap pixelMap = imageSource.createPixelmap(decodingOpts);


// 更新数据再次解码，重复调用直到数据全部更新完成
imageSource.updateData(data, 0, bytes, false);
PixelMap pixelMap = imageSource.createPixelmap(decodingOpts);


// 数据全部更新完成时需要传入 isFinal 为 true
imageSource.updateData(data, 0, bytes, true);
PixelMap pixelMap = imageSource.createPixelmap(decodingOpts);
```

解码完成获取到 PixelMap 对象后，可以进行后续处理，比如渲染显示等。

图像编码开发指导

场景介绍

图像编码就是将 PixelMap 图像编码成不同存档格式图片，用于后续其他处理，比如保存、传输等。当前仅支持 JPEG 格式。

接口说明

ImagePacker 主要用于图像编码。

表 1 图像编码类 ImagePacker 的主要接口

接口名	描述
create()	创建图像打包器实例。
initializePacking(byte[] data, PackingOptions opts)	初始化打包任务，将字节数组设置为打包后输出目的。
initializePacking(byte[] data, int offset, PackingOptions opts)	初始化打包任务，将带偏移量的字节数组设置为打包后输出目的。
initializePacking(OutputStream outputStream, PackingOptions opts)	初始化打包任务，将输出流设置为打包后输出目的。
addImage(PixelMap pixelmap)	将 PixelMap 对象添加到图像打包器中。
addImage(ImageSource source)	将图像数据源 ImageSource 中图像添加到图像打包器中。
addImage(ImageSource source, int index)	将图像数据源 ImageSource 中指定图像添加到图像打包器中。

表 1 图像编码类 ImagePacker 的主要接口

接口名	描述
finalizePacking()	完成图像打包任务。
release()	释放对象关联的本地资源。

开发步骤

创建图像编码 ImagePacker 对象。

```
ImagePacker imagePacker = ImagePacker.create();
```

设置编码输出流和编码参数。设置 format 为编码的图像格式，当前支持 jpeg 格式。设置 quality 为图像质量，范围从 0-100，100 为最佳质量。

```
FileOutputStream outputStream = new
FileOutputStream("/path/to/packed.file");

ImagePacker.PackingOptions packingOptions = new
ImagePacker.PackingOptions();

packingOptions.format = "image/jpeg";

packingOptions.quality = 90;

boolean result = imagePacker.initializePacking(outputStream,
packingOptions);
```

添加需要编码的 PixelMap 对象，进行编码操作。

```
result = imagePacker.addImage(pixelMap);

long dataSize = imagePacker.finalizePacking();
```

编码输出完成后，可以进行后续处理，比如保存、传输等。

位图操作开发指导

场景介绍

位图操作就是指对 PixelMap 图像进行相关的操作，比如创建、查询信息、读写像素数据等。

接口说明

表 1 位图操作类 PixelMap 的主要接口

接口名	描述
create(InitializationOptions opts)	根据图像大小、像素格式、alpha 类型等初始化选项创建 PixelMap。
create(int[] colors, InitializationOptions opts)	根据图像大小、像素格式、alpha 类型等初始化选项,以像素颜色数组为数据源创建 PixelMap。
create(int[] colors, int offset, int stride, InitializationOptions opts)	根据图像大小、像素格式、alpha 类型等初始化选项,以像素颜色数组、起始偏移量、行像素大小描述的数据源创建 PixelMap。
create(PixelMap source, InitializationOptions opts)	根据图像大小、像素格式、alpha 类型等初始化选项,以源 PixelMap 为数据源创建 PixelMap。
create(PixelMap source, Rect srcRegion, InitializationOptions opts)	根据图像大小、像素格式、alpha 类型等初始化选项,以源 PixelMap、源裁剪区域描述的数据源创建 PixelMap。

表 1 位图操作类 PixelMap 的主要接口

接口名	描述
getBytesNumberPerRow()	获取每行像素数据占用的字节数。
getPixelBytesCapacity()	获取存储 Pixelmap 像素数据的内存容量。
isEditable()	判断 PixelMap 是否允许修改。
isSameImage(PixelMap other)	判断两个图像是否相同，包括 ImageInfo 属性信息和像素数据。
readPixel(Position pos)	读取指定位置像素的颜色值,返回的颜色格式为 PixelFormat.ARGB_8888。
readPixels(int[] pixels, int offset, int stride, Rect region)	读取指定区域像素的颜色值,输出到以起始偏移量、行像素大小描述的像素数组，返回的颜色格式为 PixelFormat.ARGB_8888。
readPixels(Buffer dst)	读取像素的颜色值到缓冲区,返回的数据是 PixelMap 中像素数据的原样拷贝，即返回的颜色数据格式与 PixelMap 中像素格式一致。
resetConfig(Size size, PixelFormat pixelFormat)	重置 PixelMap 的大小和像素格式配置，但不会改变原有的像素数据也不会重新分配像素数据的内存，重置后图像数据的字节数不能超过 PixelMap 的内存容量。
setAlphaType(AlphaType alphaType)	设置 PixelMap 的 Alpha 类型。
writePixel(Position pos, int color)	向指定位置像素写入颜色值,写入颜色格式为 PixelFormat.ARGB_8888。

表 1 位图操作类 PixelMap 的主要接口

接口名	描述
writePixels(int[] pixels, int offset, int stride, Rect region)	将像素颜色数组、起始偏移量、行像素的个数描述的源像素数据写入 PixelMap 的指定区域,写入颜色格式为 PixelFormat.ARGB_8888。
writePixels(Buffer src)	将缓冲区描述的源像素数据写入 PixelMap,写入的数据将原样覆盖 PixelMap 中的像素数据，即写入数据的颜色格式应与 PixelMap 的配置兼容。
writePixels(int color)	将所有像素都填充为指定的颜色值,写入颜色格式为 PixelFormat.ARGB_8888。
getPixelBytesNumber()	获取全部像素数据包含的字节数。
setBaseDensity(int baseDensity)	设置 PixelMap 的基础像素密度值。
getBaseDensity()	获取 PixelMap 的基础像素密度值。
setUseMipmap(boolean useMipmap)	设置 PixelMap 渲染是否使用 mipmap。
useMipmap()	获取 PixelMap 渲染是否使用 mipmap。
getNinePatchChunk()	获取图像的 NinePatchChunk 数据。
getFitDensitySize(int targetDensity)	获取适应目标像素密度的图像缩放的尺寸。
getImageInfo()	获取图像基本信息。

表 1 位图操作类 PixelMap 的主要接口

接口名	描述
release()	释放对象关联的本地资源。

开发步骤

创建位图对象 PixelMap。

```
// 指定初始化选项创建
PixelMap pixelMap2 = PixelMap.create(initializationOptions);


// 从像素颜色数组创建
int[] defaultColors = new int[] {5, 5, 5, 5, 6, 6, 3, 3, 3, 0};
PixelMap.InitializationOptions initializationOptions = new
PixelMap.InitializationOptions();
initializationOptions.size = new Size(3, 2);
initializationOptions.pixelFormat = PixelFormat.ARGB_8888;
PixelMap pixelMap1 = PixelMap.create(defaultColors,
initializationOptions);


// 以另外一个 PixelMap 作为数据源创建
PixelMap pixelMap3 = PixelMap.create(pixelMap2, initializationOptions);
```

从位图对象中获取信息。

```
long capacity = pixelMap.getPixelBytesCapacity();
long bytesNumber = pixelMap.getPixelBytesNumber();
int rowBytes = pixelMap.getBytesNumberPerRow();
```

```
byte[] ninePatchData = pixelMap.getNinePatchChunk();
```

读写位图像素数据。

```
// 读取指定位置像素
int color = pixelMap.readPixel(new Position(1, 1));

// 读取指定区域像素
int[] pixelArray = new int[50];
Rect region = new Rect(0, 0, 10, 5);
pixelMap.readPixels(pixelArray, 0, 10, region);

// 读取像素到 Buffer
IntBuffer pixelBuf = IntBuffer.allocate(50);
pixelMap.readPixels(pixelBuf);

// 在指定位置写入像素
pixelMap.writePixel(new Position(1, 1), 0xFF112233);

// 在指定区域写入像素
pixelMap.writePixels(pixelArray, 0, 10, region);

// 写入 Buffer 中的像素
pixelMap.writePixels(intBuf);
```

图像属性解码开发指导

场景介绍

图像属性解码就是获取图像中包含的属性信息，比如 EXIF 属性。

接口说明

图像属性解码的功能主要由 ImageSource 和 ExifUtils 提供。

表 1 ImageSource 的主要接口

接口名	描述
getThumbnailInfo()	获取嵌入图像文件的缩略图的基本信息。
getImageThumbnailBytes()	获取嵌入图像文件缩略图的原始数据。
getThumbnailFormat()	获取嵌入图像文件缩略图的格式。

表 2 ExifUtils 的主要接口

接口名	描述
getLatLong(ImageSource imageSource)	获取嵌入图像文件的经纬度信息。
getAltitude(ImageSource imageSource, double defaultValue)	获取嵌入图像文件的海拔信息。

开发步骤

创建图像数据源 ImageSource 对象，可以通过 SourceOptions 指定数据源的格式信息，此格式信息仅为给解码器的提示，正确提供能帮助提高解码效率，如果不设置或设置不正确，会自动检测正确的图像格式。

```
ImageSource.SourceOptions srcOpts = new ImageSource.SourceOptions();

srcOpts.formatHint = "image/jpeg";

String pathName = "/path/to/image.jpg";

ImageSource imageSource = ImageSource.create(pathName, srcOpts);
```

获取缩略图信息。

```
int format = imageSource.getThumbnailFormat();

byte[] thumbnailBytes = imageSource.getImageThumbnailBytes();


// 将缩略图解码为 PixelMap 对象

ImageSource.DecodingOptions decodingOpts = new
ImageSource.DecodingOptions();

PixelMap thumbnailPixelmap =
imageSource.createThumbnailPixelmap(decodingOpts, false);
```

相机

概述

HarmonyOS 相机模块支持相机业务的开发，开发者可以通过已开放的接口实现相机硬件的访问、操作和新功能开发，最常见的操作如：预览、拍照、连拍和录像等。

基本概念

- **相机静态能力**

用于描述相机的固有能力的一系列参数，比如朝向、支持的分辨率等信息。

- **物理相机**

物理相机就是独立的实体摄像头设备。物理相机 ID 是用于标志每个物理摄像头的唯一字串。

- **逻辑相机**

逻辑相机是多个物理相机组合出来的抽象设备，逻辑相机通过同时控制多个物理相机设备来完成相机某些功能，如大光圈、变焦等功能。逻辑摄像机 ID 是一个唯一的字符串，标识多个物理摄像机的抽象能力。

- **帧捕获**

相机启动后对帧的捕获动作统称为帧捕获。主要包含单帧捕获、多帧捕获、循环帧捕获。

- **单帧捕获**

指的是相机启动后，在帧数据流中捕获一帧数据，常用于普通拍照。

- **多帧捕获**

指的是相机启动后，在帧数据流中连续捕获多帧数据，常用于连拍。

- **循环帧捕获**

指的是相机启动后，在帧数据流中一直捕获帧数据，常用于预览和录像。

约束与限制

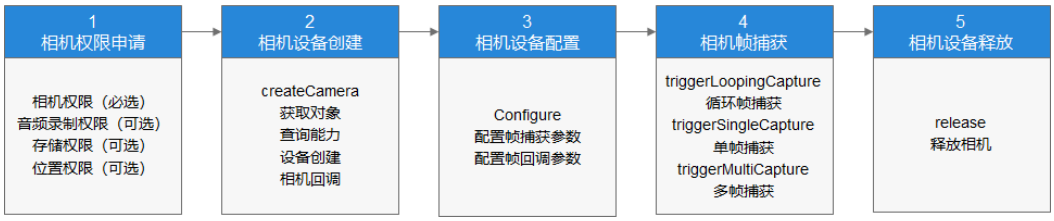
- 在同一时刻只能有一个相机应用在运行中。
- 相机模块内部有状态控制，开发者必须按照指导文档中的流程进行接口的顺序调用，否则可能会出现调用失败等问题。
- 为了开发的相机应用拥有更好的兼容性，在创建相机对象或者参数相关设置前请务必进行能力查询。

设备开发指导

相机开发流程

相机模块主要工作是给相机应用开发者提供基本的相机 API 接口，用于使用相机系统的功能，进行相机硬件的访问、操作和新功能开发。相机的开发流程如图所示：

图 1　相机开发流程



接口说明

相机模块为相机应用开发者提供了 3 个包的内容，包括方法、枚举、以及常量/变量，方便开发者更容易地实现相机功能。详情请查阅对应开发场景。

包名	功能
ohos.media.camera.CameraKit	相机功能入口类。获取当前支持的相机列表及其静态能力信息，创建相机对象。
ohos.media.camera.device	相机设备操作类。提供相机能力查询、相机配置、相机帧捕获、相机状态回调等功能。
ohos.media.camera.params	相机参数类。提供相机属性、参数和操作结果的定义。

相机权限申请

在使用相机之前，需要申请相机的相关权限，保证应用拥有相机硬件及其他功能权限，应用权限的介绍请参考 权限章节，相机涉及权限如下表。

表 1 相机权限列表

权限名称	权限属性值	是否必选
相机权限	ohos.permission.CAMERA	必选
录音权限	ohos.permission.MICROPHONE	可选（需要录像时申请）
存储权限	ohos.permission.WRITE_USER_STORAGE	可选（需要保存图像及视频到设备的外部存储时申请）
位置权限	ohos.permission.LOCATION	可选（需要保存图像及视频位置信息时申请）

相机设备创建

CameraKit 类是相机的入口 API 类，用于获取相机设备特性、打开相机，其接口如下表。

表 2 CameraKit 的主要接口

接口名	描述
createCamera(String cameraId, CameraStateCallback callback, EventHandler handler)	创建相机对象。
getCameraAbility(String cameraId)	获取指定逻辑相机或物理相机的静态能力。
getCameraIds()	获取当前逻辑相机列表。
getCameraInfo(String cameraId)	获取指定逻辑相机的信息。

表 2 CameraKit 的主要接口

接口名	描述
getInstance(Context context)	获取 CameraKit 实例。
registerCameraDeviceCallback (CameraDeviceCallback callback, EventHandler handler)	注册相机使用状态回调。
unregisterCameraDeviceCallback (CameraDeviceCallback callback)	注销相机使用状态回调。

基于 HarmonyOS 实现一个相机应用，无论将来想应用到哪个或者哪些设备上，都必须先创建一个独立的相机设备，然后才能继续相机的其他操作。相机设备创建的建议步骤如下：

1. 通过 CameraKit.getInstance(Context context)方法获取唯一的 CameraKit 对象是创建新的相机应用的第一步操作。

```
private void openCamera(){
    // 获取 CameraKit 对象
    CameraKit cameraKit = CameraKit.getInstance(context);
    if (cameraKit == null) {
        // 处理 cameraKit 获取失败的情况
    }
}
```

如果此步骤操作失败，相机可能被占用或无法使用。如果被占用，必须等到相机释放后才能重新获取 CameraKit 对象。

通过 getCameraIds()方法，获取当前使用的设备支持的逻辑相机列表。逻辑相机列表中存储了当前设备拥有的所有逻辑相机 ID，如果列表不为空，则列表中的每个 ID 都支持独立创建相机对象；否则，说明正在使用的设备无可用的相机，不能继续后续的操作。

```
try {
    // 获取当前设备的逻辑相机列表
    String[] cameraIds = cameraKit.getCameraIds();
    if (cameraIds.length <= 0) {
        HiLog.error("cameraIds size is 0");
    }
} catch (IllegalStateException e) {
    // 处理异常
}
```

还可以继续查询指定相机 ID 的静态信息：

1.调用 getDeviceLinkType(String physicalId)方法获取物理相机连接方式；

2.调用 getCameraInfo(String cameraId)方法查询相机硬件朝向等信息；

3.调用 getCameraAbility(String cameraId)方法查询相机能力信息（比如支持的分辨率列表等）。

表 3 CameraInfo 的主要接口

接口名	描述
getDeviceLinkType(String physicalId)	获取物理相机连接方式。
getFacingType()	获取相机朝向信息。
getLogicalId()	获取逻辑相机 ID。
getPhysicalIdList()	获取对应的物理相机 ID 列表。

表 4 CameraAbility 的主要接口

接口名	描述
getSupportedSizes(int format)	根据格式查询输出图像的分辨率列表。
getSupportedSizes(Class<T> clazz)	根据 Class 类型查询分辨率列表。
getParameterRange (ParameterKey.Key<T> parameter)	获取指定参数能够设置的值范围。
getPropertyValue (PropertyKey.Key<T> property)	获取指定属性对应的值。
getSupportedAeMode()	获取当前相机支持的自动曝光模式。

表 4 CameraAbility 的主要接口

接口名	描述
getSupportedAfMode()	获取当前相机支持的自动对焦模式。
getSupportedFaceDetection()	获取相机支持的人脸检测类型范围。
getSupportedFlashMode()	当前相机支持的闪光灯取值范围。
getSupportedParameters()	当前相机支持的参数设置。
getSupportedProperties()	获取当前相机的属性列表。
getSupportedResults()	获取当前相机支持的参数设置可返回的结果列表。
getSupportedZoom()	获取相机支持的变焦范围。

通过 createCamera(String cameraId, CameraStateCallback callback, EventHandler handler)方法，创建相机对象，此步骤执行成功意味着相机系统的硬件已经完成了上电。

```
// 创建相机设备
cameraKit.createCamera(cameraIds[0], cameraStateCallback, eventHandler);
```

第一个参数 cameraId 可以是上一步获取的逻辑相机列表中的任何一个相机 ID。第二和第三个参数负责相机创建和相机运行时的数据和状态检测，请务必保证在整个相机运行周期内有效。

```
private final class CameraStateCallbackImpl extends CameraStateCallback {
    @Override
```

```
    public void onCreated(Camera camera) {
         // 创建相机设备
    }

    @Override
    public void onConfigured(Camera camera) {
        // 配置相机设备
    }

    @Override
    public void onPartialConfigured(Camera camera) {
        // 当使用了 addDeferredSurfaceSize 配置了相机，会接到此回调
    }

    @Override
    public void onReleased(Camera camera) {
       // 释放相机设备
    }
}

// 相机创建和相机运行时的回调
CameraStateCallbackImpl cameraStateCallback = new
CameraStateCallbackImpl();
if(cameraStateCallback ==null) {
    HiLog.error("cameraStateCallback is null");
}
```

```
import ohos.eventhandler.EventHandler;

import ohos.eventhandler.EventRunner;


// 执行回调的 EventHandler

EventHandler eventHandler = new
EventHandler(EventRunner.create("CameraCb"));

if(eventHandler ==null) {

    HiLog.error("eventHandler is null");

}
```

至此，相机设备的创建已经完成。相机设备创建成功会在 CameraStateCallback 中触发 onCreated(Camera camera)回调。在进入相机设备配置前，请确保相机设备已经创建成功。否则会触发相机设备创建失败的回调，并返回错误码，需要进行错误处理后，重新执行相机设备的创建。

相机设备配置

创建相机设备成功后，在 CameraStateCallback 中会触发 onCreated(Camera camera)回调，并且带回 Camera 对象，用于执行相机设备的操作。

当一个新的相机设备成功创建后，首先需要对相机进行配置，调用 configure(CameraConfig)方法实现配置。相机配置主要是设置预览、拍照、录像用到的 Surface(详见 ohos.agp.graphics.Surface)，没有配置过 Surface，相应的功能不能使用。

为了进行相机帧捕获结果的数据和状态检测，还需要在相机配置时调用setFrameStateCallback(FrameStateCallback, EventHandler)方法设置帧回调。

```
private final class CameraStateCallbackImpl extends CameraStateCallback {
    @Override
    public void onCreated(Camera camera) {
        cameraConfigBuilder = camera.getCameraConfigBuilder();
        if (cameraConfigBuilder == null) {
            HiLog.error("onCreated cameraConfigBuilder is null");
            return;
        }
        // 配置预览的 Surface
        cameraConfigBuilder.addSurface(previewSurface);
        // 配置拍照的 Surface

cameraConfigBuilder.addSurface(imageReceiver.getRecevingSurface());
        // 配置帧结果的回调

cameraConfigBuilder.setFrameStateCallback(frameStateCallbackImpl,
handler);
        try {
            // 相机设备配置
            camera.configure(cameraConfigBuilder.build());
        } catch (IllegalArgumentException e) {
            HiLog.error("Argument Exception");
```

```
        } catch (IllegalStateException e) {
            HiLog.error("State Exception");
        }
    }
}
```

相机配置成功后，在 CameraStateCallback 中会触发 onConfigured(Camera camera)回调，然后才可以执行相机帧捕获相关的操作。

表 5 CameraConfig.Builder 的主要接口

接口名	描述
addSurface(Surface surface)	相机配置中增加 Surface。
build()	相机配置的构建类。
removeSurface(Surface surface)	移除先前添加的 Surface。
setFrameStateCallback (FrameStateCallback callback, EventHandler handler)	设置用于相机帧结果返回的 FrameStateCallback 和 Handler。
addDeferredSurfaceSize(Size surfaceSize, Class<T> clazz)	添加延迟 Surface 的尺寸、类型。
addDeferredSurface(Surface surface)	设置延迟的 Surface，此 Surface 的尺寸和类型必须和使用 addDeferredSurfaceSize 配置的一致。

相机帧捕获

Camera 操作类，包括相机预览、录像、拍照等功能接口。

表 6 Camera 的主要接口

接口名	描述
triggerSingleCapture (FrameConfig frameConfig)	启动相机帧的单帧捕获。
triggerMultiCapture (List<FrameConfig> frameConfigs)	启动相机帧的多帧捕获。
configure(CameraConfig config)	配置相机。
flushCaptures()	停止并清除相机帧的捕获，包括循环帧/单帧/多帧捕获。
getCameraConfigBuilder()	获取相机配置构造器对象。
getCameraId()	获取当前相机的 ID。
getFrameConfigBuilder(int type)	获取指定类型的相机帧配置构造器对象。
release()	释放相机对象及资源。
triggerLoopingCapture (FrameConfig frameConfig)	启动或者更新相机帧的循环捕获。
stopLoopingCapture()	停止当前相机帧的循环捕获。

启动预览（循环帧捕获）

用户一般都是先看见预览画面才执行拍照或者其他功能，所以对于一个普通的相机应用，预览是必不可少的。启动预览的建议步骤如下：

通过 getFrameConfigBuilder(FRAME_CONFIG_PREVIEW)方法获取预览配置模板，常用帧配置项见下表，更多的帧配置项以及详细使用方法请参考 API 接口说明的 FrameConfig.Builder 部分。

表 7 常用帧配置项

接口名	描述	是否必选
addSurface(Surface surface)	配置预览 surface 和帧的绑定。	是
setAfMode(int afMode, Rect rect)	配置对焦模式。	否
setAeMode(int aeMode, Rect rect)	配置曝光模式。	否
setZoom(float value)	配置变焦值。	否
setFlashMode(int flashMode)	配置闪光灯模式。	否
setFaceDetection(int type, boolean isEnable)	配置人脸检测或者笑脸检测。	否
setParameter(Key<T> key, T value)	配置其他属性（如自拍镜像等）。	否
setMark(Object mark)	配置一个标签，后续可以从 FrameConfig 中通过 Object getMark()拿到标签，判断两个是否相等，相等就说明是同一个配置。	否

表 7 常用帧配置项

接口名	描述	是否必选
setCoordinateSurface(Surface surface)	配置坐标系基准 Surface，后续计算 Ae/Af 等区域都会基于此 Surface 为基本的中心坐标系，不设置默认使用添加的第一个 Surface。	否

通过 triggerLoopingCapture(FrameConfig)方法实现循环帧捕获(如预览/录像)。

```
private final class CameraStateCallbackImpl extends CameraStateCallback {

    @Override

    public void onConfigured(Camera camera) {

        // 获取预览配置模板

        frameConfigBuilder =
camera.getFrameConfigBuilder(FRAME_CONFIG_PREVIEW);

        // 配置预览 Surface

        frameConfigBuilder.addSurface(previewSurface);

        previewFrameConfig = frameConfigBuilder.build();

        try {

            // 启动循环帧捕获

            int triggerId =
camera.triggerLoopingCapture(previewFrameConfig);

        } catch (IllegalArgumentException e) {

            HiLog.error("Argument Exception");

        } catch (IllegalStateException e) {
```

```
            HiLog.error("State Exception");
        }
    }
}
```

经过以上的操作，相机应用已经可以正常进行实时预览了。在预览状态下，开发者还可以执行其他操作，比如：

当预览帧配置更改时，可以通过 triggerLoopingCapture(FrameConfig)方法实现预览帧配置的更新；

```
// 预览帧变焦值变更
frameConfigBuilder.setZoom(1.2f);
// 调用 triggerLoopingCapture 方法实现预览帧配置更新
triggerLoopingCapture(frameConfigBuilder.build());
```

通过 stopLoopingCapture()方法停止循环帧捕获(停止预览)。

```
// 停止预览帧捕获
camera.stopLoopingCapture(frameConfigBuilder.build())
```

实现拍照（单帧捕获）

拍照功能属于相机应用的最重要功能之一，而且照片质量对用户至关重要。相机模块基于相机复杂的逻辑，从应用接口层到器件驱动层都已经默认的做好了最适合用户的配置，这些默认配置尽可能地保证用户拍出的每张照片的质量。发起拍照的建议步骤如下：

通过 getFrameConfigBuilder(FRAME_CONFIG_PICTURE)方法获取拍照配置模板，并且设置拍照帧配置，如下表：

表 8 常用拍照帧配置

接口名	描述	是否必选
FrameConfig.Builder addSurface(Surface)	实现拍照 Surface 和帧的绑定。	必选
FrameConfig.Builder setImageRotation(int)	设置图片旋转角度。	可选
FrameConfig.Builder setLocation(Location)	设置图片地理位置信息。	可选
FrameConfig.Builder setParameter(Key<T>, T)	配置其他属性（如自拍镜像等）。	可选

拍照前准备图像帧数据的接收实现。

```
// 图像帧数据接收处理对象
private ImageReceiver imageReceiver;
// 执行回调的 EventHandler
private EventHandler eventHandler = new
EventHandler(EventRunner.create("CameraCb"));
// 拍照支持分辨率
private Size pictureSize;


// 单帧捕获生成图像回调 Listener
private final ImageReceiver.IImageArrivalListener imageArrivalListener =
new ImageReceiver.IImageArrivalListener() {
```

```
    @Override
    public void onImageArrival(ImageReceiver imageReceiver) {
        StringBuffer fileName = new StringBuffer("picture_");
        fileName.append(UUID.randomUUID()).append(".jpg"); // 定义生成图片文件名
        File myFile = new File(dirFile, fileName.toString()); // 创建图片文件
        imageSaver = new ImageSaver(imageReceiver.readNextImage(), myFile); // 创建一个读写线程任务用于保存图片
        eventHandler.postTask(imageSaver); // 执行读写线程任务生成图片
    }
};

// 保存图片，图片数据读写，及图像生成见 run 方法
class ImageSaver implements Runnable {
    private final Image myImage;
    private final File myFile;

    ImageSaver(Image image, File file) {
        myImage = image;
        myFile = file;
    }

    @Override
    public void run() {
        Image.Component component = myImage.getComponent(ImageFormat.ComponentType.JPEG);
        byte[] bytes = new byte[component.remaining()];
        component.read(bytes);
```

```
        FileOutputStream output = null;
        try {
            output = new FileOutputStream(myFile);
            output.write(bytes); // 写图像数据
        } catch (IOException e) {
            HiLog.error("save picture occur exception!");
        } finally {
            myImage.release();
            if (output != null) {
                try {
                    output.close(); // 关闭流
                } catch (IOException e) {
                    HiLog.error("image release occur exception!");
                }
            }
        }
    }
}
private void takePictureInit() {
    List<Size> pictureSizes =
cameraAbility.getSupportedSizes(ImageFormat.JPEG); // 获取拍照支持分辨率列
表
    pictureSize = getpictureSize(pictureSizes) // 根据拍照要求选择合适的分
辨率
    imageReceiver = ImageReceiver.create(Math.max(pictureSize.width,
pictureSize.height),
        Math.min(pictureSize.width, pictureSize.height),
ImageFormat.JPEG, 5); // 创建 ImageReceiver 对象，注意 creat 函数中宽度要大于
高度；5 为最大支持的图像数，请根据实际设置。
    imageReceiver.setImageArrivalListener(imageArrivalListener);
```

```
}
```

通过 triggerSingleCapture(FrameConfig)方法实现单帧捕获(如拍照)。

```
private void capture() {
    // 获取拍照配置模板
    framePictureConfigBuilder =
cameraDevice.getFrameConfigBuilder(FRAME_CONFIG_PICTURE);
    // 配置拍照 Surface

framePictureConfigBuilder.addSurface(imageReceiver.getRecevingSurface()
);
    // 配置拍照其他参数
    framePictureConfigBuilder.setImageRotation(90);
    try {
        // 启动单帧捕获(拍照)
        camera.triggerSingleCapture(framePictureConfigBuilder.build());
    } catch (IllegalArgumentException e) {
        HiLog.error("Argument Exception");
    } catch (IllegalStateException e) {
        HiLog.error("State Exception");
    }
}
```

为了捕获到质量更高和效果更好的图片，还可以在帧结果中实时监测自动对焦和自动曝光的状态，一般而言，在自动对焦完成，自动曝光收敛后的瞬间是发起单帧捕获的最佳时机。

实现连拍（多帧捕获）

连拍功能方便用户一次拍照获取多张照片，用于捕捉精彩瞬间。同普通拍照的实现流程一致，但连拍需要使用 triggerMultiCapture(List<FrameConfig> frameConfigs)方法。

启动录像（循环帧捕获）

启动录像和启动预览类似，但需要另外配置录像 Surface 才能使用。

录像前需要进行音视频模块的配置。

```
private Source source; // 音视频源
private AudioProperty.Builder audioPropertyBuilder; // 音频属性
private VideoProperty.Builder videoPropertyBuilder; // 视频属性
private StorageProperty.Builder storagePropertyBuilder; // 音视频存储属性
private Recorder mediaRecorder; // 录像操作对象
private String recordName; // 音视频文件名

private void initMediaRecorder() {
    HiLog.info("initMediaRecorder begin");
    videoPropertyBuilder.setRecorderBitRate(10000000); // 设置录制比特率
    int rotation =
DisplayManager.getInstance().getDefaultDisplay(this).get().getRotation(
);
    videoPropertyBuilder.setRecorderDegrees(getOrientation(rotation));
// 设置录像方向
    videoPropertyBuilder.setRecorderFps(30); // 设置录制采样率
```

```
    videoPropertyBuilder.setRecorderHeight(Math.min(recordSize.height,
recordSize.width)); // 设置录像支持的分辨率，需保证 width > height

    videoPropertyBuilder.setRecorderWidth(Math.max(recordSize.height,
recordSize.width));


videoPropertyBuilder.setRecorderVideoEncoder(Recorder.VideoEncoder.H264
); // 设置视频编码方式

    videoPropertyBuilder.setRecorderRate(30); // 设置录制帧率

    source.setRecorderAudioSource(Recorder.AudioSource.MIC); // 设置录制
音频源

    source.setRecorderVideoSource(Recorder.VideoSource.SURFACE); // 设置
视频窗口

    mediaRecorder.setSource(source); // 设置音视频源

    mediaRecorder.setOutputFormat(Recorder.OutputFormat.MPEG_4); // 设置
音视频输出格式

    StringBuffer fileName = new StringBuffer("record_"); // 生成随机文件名

    fileName.append(UUID.randomUUID()).append(".mp4");

    recordName = fileName.toString();

    File file = new File(dirFile, fileName.toString()); // 创建录像文件对
象

    storagePropertyBuilder.setRecorderFile(file); // 设置存储音视频文件名

    mediaRecorder.setStorageProperty(storagePropertyBuilder.build());


audioPropertyBuilder.setRecorderAudioEncoder(Recorder.AudioEncoder.AAC)
; // 设置音频编码格式

    mediaRecorder.setAudioProperty(audioPropertyBuilder.build()); // 设置
音频属性

    mediaRecorder.setVideoProperty(videoPropertyBuilder.build()); // 设置
视频属性

    mediaRecorder.prepare(); // 准备录制

    HiLog.info("initMediaRecorder end");

}
```

配置录像帧，启动录像。

```
private final class CameraStateCallbackImpl extends CameraStateCallback {

    @Override

    public void onConfigured(Camera camera) {

        // 获取预览配置模板

        frameConfigBuilder =
camera.getFrameConfigBuilder(FRAME_CONFIG_PREVIEW);

        // 配置预览 Surface

        frameConfigBuilder.addSurface(previewSurface);

        // 配置录像的 Surface

        mRecorderSurface = mediaRecorder.getVideoSurface();

        cameraConfigBuilder.addSurface(mRecorderSurface);

        previewFrameConfig = frameConfigBuilder.build();

        try {

            // 启动循环帧捕获

            int triggerId =
camera.triggerLoopingCapture(previewFrameConfig);

        } catch (IllegalArgumentException e) {

            HiLog.error("Argument Exception");

        } catch (IllegalStateException e) {

            HiLog.error("State Exception");

        }

    }

}
```

通过 camera.stopLoopingCapture()方法停止循环帧捕获（录像）。

相机设备释放

使用完相机后，必须通过 release()来关闭相机和释放资源，否则可能导致其他相机应用无法启动。一旦相机被释放，它所提供的操作就不能再被调用，否则会导致不可预期的结果，或是会引发状态异常。

相机设备释放的示例代码如下：

```
private void releaseCamera() {
    if (camera != null) {
        // 关闭相机和释放资源
        camera.release();
        camera = null;
    }
    // 拍照配置模板置空
    framePictureConfigBuilder = null;
    // 预览配置模板置空
    previewFrameConfig = null;
}
```

概述

HarmonyOS 音频模块支持音频业务的开发，提供音频相关的功能，主要包括音频播放、音频采集、音量管理和短音播放等。

基本概念

- **采样**

采样是指将连续时域上的模拟信号按照一定的时间间隔采样，获取到离散时域上离散信号的过程。

- **采样率**

采样率为每秒从连续信号中提取并组成离散信号的采样次数，单位用赫兹（Hz）来表示。通常人耳能听到频率范围大约在 20Hz～20kHz 之间的声音。常用的音频采样频率有：8kHz、11.025kHz、22.05kHz、16kHz、37.8kHz、44.1kHz、48kHz、96kHz、192kHz 等。

- **声道**

声道是指声音在录制或播放时在不同空间位置采集或回放的相互独立的音频信号，所以声道数也就是声音录制时的音源数量或回放时相应的扬声器数量。

- **音频帧**

音频数据是流式的，本身没有明确的一帧帧的概念，在实际的应用中，为了音频算法处理/传输的方便，一般约定俗成取 2.5ms~60ms 为单位的数据量为一帧音频。这个时间被称之为“采样时间”，其长度没有特别的标准，它是根据编解码器和具体应用的需求来决定的。

- **PCM**

PCM（Pulse Code Modulation），即脉冲编码调制，是一种将模拟信号数字化的方法，是将时间连续、取值连续的模拟信号转换成时间离散、抽样值离散的数字信号的过程。

- **短音**

使用源于应用程序包内的资源或者是文件系统里的文件为样本，将其解码成一个 16bit 单声道或者立体声的 PCM 流并加载到内存中，这使得应用程序可以直接用压缩数据流同时摆脱 CPU 加载数据的压力和播放时重解压的延迟。

- **tone 音**

根据特定频率生成的波形，比如拨号盘的声音。

- **系统音**

系统预置的短音，比如按键音，删除音等。

约束与限制

- 在使用完 AudioRenderer 音频播放类和 AudioCapturer 音频采集类后，需要调用 release()方法进行资源释放。
- 音频采集所使用的最终采样率与采样格式取决于输入设备，不同设备支持的格式及采样率范围不同，可以通过 AudioManager 类的 getDevices 接口查询。
- 在进行音频采集之前，需要申请麦克风权限 ohos.permission.MICROPHONE。

音频播放开发指导

场景介绍

音频播放的主要工作是将音频数据转码为可听见的音频模拟信号并通过输出设备进行播放，同时对播放任务进行管理。

接口说明

表 1 音频播放类 AudioRenderer 的主要接口

接口名	描述
AudioRenderer(AudioRendererInfo audioRendererInfo, PlayMode pm) throws IllegalArgumentException	构造函数，设置播放相关音频参数和播放模式，使用默认播放设备。
AudioRenderer(AudioRendererInfo audioRendererInfo, PlayMode pm, AudioDeviceDescriptor outputDevice) throws IllegalArgumentException	构造函数，设置播放相关音频参数、播放模式和播放设备。
boolean start()	播放音频流。
boolean write(byte[] data, int offset, int size)	将音频数据以 byte 流写入音频接收器以进行播放。
boolean write(short[] data, int offset, int size)	将音频数据以 short 流写入音频接收器以进行播放。
boolean write(float[] data, int offset, int size)	将音频数据以 float 流写入音频接收器以进行播放。
boolean write(java.nio.ByteBuffer data, int size)	将音频数据以 ByteBuffer 流写入音

表 1 音频播放类 AudioRenderer 的主要接口

接口名	描述
	频接收器以进行播放。
boolean pause()	暂停播放音频流。
boolean stop()	停止播放音频流。
boolean release()	释放播放资源。
AudioDeviceDescriptor getCurrentDevice()	获取当前工作的音频播放设备。
boolean setPlaybackSpeed(float speed)	设置播放速度。
boolean setPlaybackSpeed (AudioRenderer.SpeedPara speedPara)	设置播放速度与音调。
boolean setVolume(ChannelVolume channelVolume)	设置指定声道上的输出音量。
boolean setVolume(float vol)	设置所有声道上的输出音量。
static int getMinBufferSize(int sampleRate, AudioStreamInfo.EncodingFormat format, AudioStreamInfo.ChannelMask channelMask)	获取 Stream 播放模式所需的 buffer 大小。
State getState()	获取音频播放的状态。
int getRendererSessionId()	获取音频播放的 session ID。
int getSampleRate()	获取采样率。
int getPosition()	获取音频播放的帧数位置。
boolean setPosition(int position)	设置起始播放帧位置。

表 1 音频播放类 AudioRenderer 的主要接口

接口名	描述
AudioRendererInfo getRendererInfo()	获取音频渲染信息。
boolean duckVolume()	降低音量并将音频与另一个拥有音频焦点的应用程序混合。
boolean unduckVolume()	恢复音量。
SpeedPara getPlaybackSpeed()	获取播放速度、音调参数。
boolean setSpeed(SpeedPara speedPara)	设置播放速度、音调参数。
Timestamp getAudioTime()	获取播放时间戳信息。
boolean flush()	刷新当前的播放流数据队列。
static float getMaxVolume()	获取播放流可设置的最大音量。
static float getMinVolume()	获取播放流可设置的最小音量。
StreamType getStreamType()	获取播放流的音频流类型。

开发步骤

构造音频流参数的数据结构 AudioStreamInfo，推荐使用 AudioStreamInfo.Builder 类来构造，模板如下，模板中设置的均为 AudioStreamInfo.Builder 类的默认值，根据音频流的具体规格来设置具体参数。

```
AudioStreamInfo audioStreamInfo = new
AudioStreamInfo.Builder().sampleRate(

    AudioStreamInfo.SAMPLE_RATE_UNSPECIFIED)

    .audioStreamFlag(AudioStreamInfo.AudioStreamFlag.AUDIO_STREAM_FLAG_N
ONE)
```

```
    .encodingFormat(AudioStreamInfo.EncodingFormat.ENCODING_INVALID)
    .channelMask(AudioStreamInfo.ChannelMask.CHANNEL_INVALID)
    .streamUsage(AudioStreamInfo.StreamUsage.STREAM_USAGE_UNKNOWN)
    .build();
```

以真实的播放 pcm 流为例：

```
AudioStreamInfo audioStreamInfo = new
AudioStreamInfo.Builder().sampleRate(44100) // 44.1kHz
    .audioStreamFlag(AudioStreamInfo.AudioStreamFlag.AUDIO_STREAM_FLAG_MAY_DUCK) // 混音
    .encodingFormat(AudioStreamInfo.EncodingFormat.ENCODING_PCM_16BIT)
// 16-bit PCM
    .channelMask(AudioStreamInfo.ChannelMask.CHANNEL_OUT_STEREO) // 双声道
    .streamUsage(AudioStreamInfo.StreamUsage.STREAM_USAGE_MEDIA) // 媒体类音频
    .build();
```

使用创建的音频流构建音频播放的参数结构 AudioRendererInfo，推荐使用 AudioRendererInfo.Builder 类来构造，模板如下，模板中设置的均为 AudioRendererInfo.Builder 类的默认值，根据音频播放的具体规格来设置具体参数。

```
AudioRendererInfo audioRendererInfo = new
AudioRendererInfo.Builder().audioStreamInfo(audioStreamInfo)
    .audioStreamOutputFlag(AudioRendererInfo.AudioStreamOutputFlag.AUDIO_STREAM_OUTPUT_FLAG_NONE)
    .bufferSizeInBytes(0)
    .distributedDeviceId("")
    .isOffload(false)
```

```
    .sessionID(AudioRendererInfo.SESSION_ID_UNSPECIFIED)
    .build();
```

以真实的播放 pcm 流为例：

```
AudioRendererInfo audioRendererInfo = new
AudioRendererInfo.Builder().audioStreamInfo(audioStreamInfo)
    .audioStreamOutputFlag(AudioRendererInfo.AudioStreamOutputFlag.AUDIO
_STREAM_OUTPUT_FLAG_DIRECT_PCM) // pcm 格式的输出流
    .bufferSizeInBytes(100)
    .distributedDeviceId("E54***5E8") // 使用分布式设备 E54***5E8 播放
    .isOffload(false) // false 表示分段传输 buffer 并播放，true 表示整个音频流
一次性传输到 HAL 层播放
    .build();
```

1. 根据要播放音频流指定 PlayMode，不同的 PlayMode 在写数据时存在差异，详情见步骤 7，其余播放流程是无区别的。并通过构造函数获取 AudioRenderer 类的实例化对象。
2. 使用构造函数获取 AudioRenderer 类的实例化对象，其中步骤 2，步骤 3 中的数据为构造函数的必选参数，指定播放设备为可选参数，根据使用场景选择不同的构造函数。

（可选）构造音频播放回调，首先构造对象 AudioInterrupt，其中 setInterruptListener 方法的入参需要实现接口类 InterruptListener，setStreamInfo 方法使用步骤 1 的 AudioStreamInfo 作为入参。然后调用 AudioManager 类的 activateAudioInterrupt(AudioInterrupt interrupt)方法进行音频播放回调注册。代码示例如下：

```
AudioInterrupt audioInterrupt = new AudioInterrupt();
AudioManager audioManager = new AudioManager();
audioInterrupt.setStreamInfo(streamInfo);
```

```
audioInterrupt.setInterruptListener(new
AudioInterrupt.InterruptListener() {

    @Override

    public void onInterrupt(int type, int hint) {

        if (type == AudioInterrupt.INTERRUPT_TYPE_BEGIN

                && hint == AudioInterrupt.INTERRUPT_HINT_PAUSE) {

            renderer.pause();

        } else if (type == AudioInterrupt.INTERRUPT_TYPE_BEGIN

                && hint == AudioInterrupt.INTERRUPT_HINT_NONE) {


        } else if (type == AudioInterrupt.INTERRUPT_TYPE_END && (

                hint == AudioInterrupt.INTERRUPT_HINT_NONE

                        || hint == AudioInterrupt.INTERRUPT_HINT_RESUME)) {

            renderer.play();

        } else {

        }

    }

});

audioManager.activateAudioInterrupt(audioInterrupt);
```

1. 调用 AudioRenderer 实例化对象的 start()方法启动播放任务。
2. 将要播放的音频数据读取为 byte 流或 short 流，对于选择 MODE_STREAM 模式的 PlayMode，需要循环调用 write 方法进行数据写入。对于选择 MODE_STATIC 模式的 PlayMode，只能通过调用一次 write 方法将要播放的音频数据全部写入，因此该模式限制在文件规格较小的音频数据播放场景下才能使用。
3. （可选）当需要对音频播放进行暂停或停止时，调用 AudioRenderer 实例化对象的 pause()或 stop()方法进行暂停或停止播放。
4. （可选）调用 AudioRenderer 实例化对象的 setSpeed 调节播放速度，setVolume 调节播放音量。
5. 播放任务结束后，调用 AudioRenderer 实例化对象的 release()释放资源。

音频采集开发指导

场景介绍

音频采集的主要工作是通过输入设备将声音采集并转码为音频数据，同时对采集任务进行管理。

接口说明

表 1 音频采集类 AudioCapturer 的主要接口

接口名	描述
AudioCapturer(AudioCapturerInfo audioCapturerInfo) throws IllegalArgumentException	构造函数，设置录音相关音频参数，使用默认录音设备。
AudioCapturer(AudioCapturerInfo audioCapturerInfo, AudioDeviceDescriptor devInfo) throws IllegalArgumentException	构造函数，设置录音相关音频参数并指定录音设备。
static int getMinBufferSize(int sampleRate, int channelCount, int audioFormat)	获取指定参数条件下所需的最小缓冲区大小。
boolean addSoundEffect(UUID type, String packageName)	增加录音的音频音效。
boolean start()	开始录音。
int read(byte[] data, int offset, int size)	读取音频数据。
int read(byte[] data, int offset, int size, boolean isBlocking)	读取音频数据并写入传入的 byte 数组中。

表 1 音频采集类 AudioCapturer 的主要接口

接口名	描述
int read(float[] data, int offsetInFloats, int sizeInFloats)	阻塞式读取音频数据并写入传入的 float 数组中。
int read(float[] data, int offsetInFloats, int sizeInFloats, boolean isBlocking)	读取音频数据并写入传入的 float 数组中。
int read(short[] data, int offsetInShorts, int sizeInShorts)	阻塞式读取音频数据并写入传入的 short 数组中。
int read(short[] data, int offsetInShorts, int sizeInShorts, boolean isBlocking)	读取音频数据并写入传入的 short 数组中。
int read(java.nio.ByteBuffer buffer, int sizeInBytes)	阻塞式读取音频数据并写入传入的 ByteBuffer 对象中。
int read(java.nio.ByteBuffer buffer, int sizeInBytes, boolean isBlocking)	读取音频数据并写入传入的 ByteBuffer 对象中。
boolean stop()	停止录音。
boolean release()	释放录音资源。
AudioDeviceDescriptor getSelectedDevice()	获取输入设备信息。
AudioDeviceDescriptor getCurrentDevice()	获取当前正在录制音频的设备信息。
int getCapturerSessionId()	获取录音的 session ID。
Set<SoundEffect> getSoundEffects()	获取已经激活的音频音效列表。
AudioCapturer.State getState()	获取音频采集状态。
int getSampleRate()	获取采样率。

表 1 音频采集类 AudioCapturer 的主要接口

接口名	描述
int getAudioInputSource()	获取录音的输入设备信息。
int getBufferFrameCount()	获取以帧为单位的缓冲区大小。
int getChannelCount()	获取音频采集通道数。
AudioStreamInfo.EncodingFormat getEncodingFormat()	获取音频采集的音频编码格式。
boolean getAudioTime(Timestamp timestamp, Timestamp.Timebase timebase)	获取一个即时的捕获时间戳。

开发步骤

构造音频流参数的数据结构 AudioStreamInfo，推荐使用 AudioStreamInfo.Builder 类来构造，模板如下，模板中设置的均为 AudioStreamInfo.Builder 类的默认值，根据音频流的具体规格来设置具体参数。

```
AudioStreamInfo audioStreamInfo = new
AudioStreamInfo.Builder().sampleRate(
    AudioStreamInfo.SAMPLE_RATE_UNSPECIFIED)
    .audioStreamFlag(AudioStreamInfo.AudioStreamFlag.AUDIO_STREAM_FLAG_N
ONE)
    .encodingFormat(AudioStreamInfo.EncodingFormat.ENCODING_INVALID)
    .channelMask(AudioStreamInfo.ChannelMask.CHANNEL_INVALID)
    .streamUsage(AudioStreamInfo.StreamUsage.STREAM_USAGE_UNKNOWN)
    .build();
```

1. （可选）通过采集的采样率、声道数和数据格式，调用 getMinBufferSize 方法获取采

集任务所需的最小 buffer，参照该 buffer 值设置步骤 3 中 AudioCapturerInfo 的 bufferSizeInBytes。

2. 使用步骤 1 创建的音频流构建音频播放的参数结构 AudioCapturerInfo，推荐使用 AudioCapturerInfo.Builder 类来构造，根据音频采集的具体规格来设置具体参数。以真实的录制 pcm 流为例：

```
AudioStreamInfo audioStreamInfo = new
AudioStreamInfo.Builder().encodingFormat(

    AudioStreamInfo.EncodingFormat.ENCODING_PCM_16BIT) // 16-bit PCM

    .channelMask(AudioStreamInfo.ChannelMask.CHANNEL_IN_STEREO) // 双声道

    .sampleRate(44100) // 44.1kHz

    .build();

AudioCapturerInfo audioCapturerInfo = new
AudioCapturerInfo.Builder().audioStreamInfo(audioStreamInfo)

    .build();
```

1. （可选）设置采集设备，如麦克风、耳机等。通过 AudioManager.getDevices(AudioDeviceDescriptor.DeviceFlag.INPUT_DEVICES_FLAG) 获取到设备支持的输入设备，然后依照 AudioDeviceDescriptor.DeviceType 选择要选用的输入设备类型。
2. 通过构造方法获取 AudioCapturer 类的实例化对象，其中步骤 3 的参数为必选参数，通过步骤 4 获取的指定录音设备为可选参数。
3. （可选）设置采集音效，如降噪、回声消除等。使用 addSoundEffect(UUID type, String packageName)进行音效设置，其中 UUID 参考类 SoundEffect 中提供的静态变量。
4. （可选）构造音频采集回调，首先继承抽象类 AudioCapturerCallback，并实现抽象方法 onCapturerConfigChanged(List<AudioCapturerConfig> configs)，然后调用 AudioManager 类的 registerAudioCapturerCallback(AudioCapturerCallback cb)方法进行音频采集回调注册。代码示例如下：

```
private AudioManager audioManager = new AudioManager();


public void main() {

    AudioCapturerCallback cb = new AudioCapturerCallback() {

        @Override

        public void onCapturerConfigChanged(List<AudioCapturerConfig>
configs) {
```

```
            configs.forEach(config -> doSomething(config));
        }
    };
    audioManager.registerAudioCapturerCallback(cb);
}

private void doSomething(AudioCapturerConfig config) {
    ...
}
```

1. 调用 AudioCapturer 实例化对象的 start()方法启动采集任务。
2. 采集的音频数据读取为 byte 流，循环调用 read 方法进行数据读取。
3. 调用 AudioCapturer 实例化对象的 stop()方法停止采集。
4. 播放任务结束后，调用 AudioCapturer 实例化对象的 release()释放资源。

音量管理开发指导

场景介绍

音量管理的主要工作是音量调节，输入/输出设备管理，注册音频中断、音频采集中断的回调等。

接口说明

表 1 音量管理类 AudioManager 的主要接口

接口名	描述
AudioManager()	构造函数。
AudioManager(Context context)	构造函数，由使用者指定应用上下文 Context。
AudioManager(String packageName)	构造函数，由使用者指定包信息。
activateAudioInterrupt(AudioInterrupt interrupt)	激活音频中断状态检测。
deactivateAudioInterrupt(AudioInterrupt interrupt)	去激活音频中断状态检测。
getAudioParameter(String key)	获取音频硬件中指定参数 keys 所对应的参数值。
AudioDeviceDescriptor[] getDevices(DeviceFlag flag)	获取设备信息。

表 1 音量管理类 AudioManager 的主要接口

接口名	描述
int getMaxVolume(AudioVolumeType volumeType)	获取指定音频流音量最大档位。
int getMinVolume(AudioVolumeType volumeType)	获取指定音频流音量最小档位。
int getRingerMode()	获取铃声模式。
int getVersion()	获取音频套件版本。
int getVolume(AudioVolumeType volumeType)	获取指定音频流的音量档位。
boolean isDeviceActive(int deviceType)	判断设备的开关状态。
boolean isMute(AudioVolumeType volumeType)	特定的流是否处于静音状态。
boolean mute(AudioVolumeType volumeType)	将特定流设置为静音状态。
boolean setAudioParameter(String key, String value)	为音频硬件设置可变数量的参数值。
boolean setDeviceActive(int deviceType, boolean state)	设置设备的开关状态。
boolean setRingerMode(AudioRingMode mode)	设置铃声模式。
boolean setVolume(AudioVolumeType volumeType, int volume)	设置特定流的音量档位。
boolean unmute(AudioVolumeType	将特定流设置为非静音状态。

表 1 音量管理类 AudioManager 的主要接口

接口名	描述
volumeType)	
boolean setMasterMute(boolean isMute)	将主音频输出设备设置为静音或取消静音状态。
boolean setMicrophoneMute(boolean isMute)	将麦克风设置为静音或取消静音状态。
boolean isMicrophoneMute()	判断麦克风是否处于静音状态。
List<AudioCapturerConfig> getActiveCapturerConfigs()	获取设备当前激活的音频采集任务的配置信息。
registerAudioCapturerCallback(AudioCapturerCallback cb)	注册音频采集参数变更回调。
void unregisterAudioCapturerCallback (AudioCapturerCallback cb)	去注册音频采集参数变更回调。
Uri getRingerUri(Context context, RingToneType type)	获取指定铃声类型的 Uri。
void setRingerUri(Context context, RingToneType type, Uri uri)	设置指定铃声类型的 Uri。
AudioManager.CommunicationState getCommunicationState()	获取当前的通话模式。
void setCommunicationState (AudioManager.CommunicationState communicationState)	设置当前的通话模式。
boolean changeVolumeBy (AudioVolumeType volumeType, int index)	将当前音量增加或减少一定量。

表 1 音量管理类 AudioManager 的主要接口

接口名	描述
boolean connectBluetoothSco()	连接到蓝牙 SCO 通道。
boolean disconnectBluetoothSco()	断开与蓝牙 SCO 通道的连接。
java.util.List<AudioRendererInfo> getActiveRendererConfigs()	获取有关活动音频流信息，包括使用类型、内容类型和标志。
static int getMasterOutputFrameCount()	获取主输出设备缓冲区中的帧数。
static int getMasterOutputSampleRate()	获取主输出设备的采样率。
boolean isMasterMute()	检查音频流是否全局静音。
static boolean isStreamActive (AudioVolumeType volumeType)	检查指定类型的音频流是否处于活动状态。
static int makeSessionId()	创建一个会话 ID，AudioRendererInfo.Builder.sessionID(int)将使用该会话 ID 来设置音频播放参数，而 AudioCapturerInfo.Builder.sessionID(int)将使用该会话 ID 来设置记录参数。
void registerAudioRendererCallback (AudioRendererCallback cb)	注册音频播放参数变更回调。
void unregisterAudioRendererCallback (AudioRendererCallback cb)	去注册音频播放参数变更回调。

开发步骤

音量管理提供的都是独立的功能，一般作为音频播放和音频采集的功能补充来使用。开发者根据具体使用场景选择方法即可。

音频中断状态检测和音频采集中断状态检测的使用样例，请参考音频播放和音频采集的开发步骤。

短音播放开发指导

场景介绍

短音播放主要负责管理音频资源的加载与播放、tone 音的生成与播放以及系统音播放。

接口说明

短音播放开放能力分为音频资源、tone 音和系统音三部分，均定义在 SoundPlayer 类。

表 1 音频资源的加载与播放类 SoundPlayer 的主要接口

接口名	描述
SoundPlayer(int taskType)	构造函数，仅用于音频资源。
int createSound(String path)	从指定的路径加载音频数据生成短音资源。
int createSound(Context context, int resourceId)	根据应用程序上下文合音频资源 ID 加载音频数据生成短音资源。
int createSound(AssetFD assetFD)	从指定的 AssetFD 实例加载音频数据生成短音资源。
int createSound(java.io.FileDescriptor fd, long offset, long length)	根据文件描述符从文件加载音频数据生成音频资源。
int createSound(java.lang.String path, AudioRendererInfo rendererInfo)	根据从指定路径和播放信息加载音频数据生成短音资源。
boolean setOnCreateCompleteListener	设置声音创建完成的回调。

表 1 音频资源的加载与播放类 SoundPlayer 的主要接口

接口名	描述
(SoundPlayer.OnCreateCompleteListener listener)	
boolean setOnCreateCompleteListener (SoundPlayer.OnCreateCompleteListener listener, boolean isDiscarded)	设置用于声音创建完成的回调，并根据指定的 isDiscarded 标志位确定是否丢弃队列中的原始回调通知消息。
boolean deleteSound(int soundID)	删除短音同时释放短音所占资源。
boolean pause(int taskID)	根据播放任务 ID 暂停对应的短音播放。
int play(int soundID)	使用默认参数播放短音。
int play(int soundID, SoundPlayerParameters parameters)	使用指定参数播放短音。
boolean resume(int taskID)	恢复短音播放任务。
boolean setLoop(int taskID, int loopNum)	设置短音播放任务的循环次数。
boolean setPlaySpeedRate(int taskID, float speedRate)	设置短音播放任务的播放速度。
boolean setPriority(int taskID, int priority)	设置短音播放任务的优先级。
boolean setVolume(int taskID, AudioVolumes audioVolumes)	设置短音播放任务的播放音量。
boolean setVolume(int taskID, float volume)	设置短音播放任务的所有音频声道的播放音量。
boolean stop(int taskID)	停止短音播放任务。

表 1 音频资源的加载与播放类 SoundPlayer 的主要接口

接口名	描述
boolean pauseAll()	暂停所有正在播放的任务。
boolean resumeAll()	恢复虽有已暂停的播放任务。

表 2 tone 音的生成与播放 API 接口功能介绍

接口名	描述
SoundPlayer()	构造函数，仅用于 tone 音。
boolean createSound(ToneDescriptor.ToneType type, int durationMs)	创建具有音调频率描述和持续时间（毫秒）的 tone 音。
boolean createSound(AudioStreamInfo.StreamType streamType, float volume)	根据音量和音频流类型创建 tone 音。
boolean play(ToneDescriptor.ToneType toneType, int durationMs)	播放指定时长和 tone 音类型的 tone 音。
boolean pause()	暂停 tone 音播放。
boolean play()	播放创建好的 tone 音。
boolean release()	释放 tone 音资源。

表 3 系统音的播放 API 接口功能介绍

接口名	描述
SoundPlayer(String packageName)	构造函数，仅用于系统音。
boolean playSound(SoundType type)	播放系统音。
boolean playSound(SoundType type, float volume)	指定音量播放系统音。

音频资源的加载与播放

1. 通过 SoundPlayer(int taskType)构造方法获取 SoundPlayer 类的实例化对象，其中入参 taskType 的取值范围和含义参考枚举类 AudioManager.AudioStreamType 的定义。
2. 调用 createSound(String path)方法从指定路径加载音频资源，并生成短音 ID，后续可使用通过短音 ID 进行短音资源的播放和删除等操作。
3. （可选）提供单独对音量，循环次数，播放速度和优先级进行的设置的方法，支持在短音播放过程中进行实时调整。
4. 短音播放提供两种方法，一种是包含播放参数设置的 play(int soundID, SoundPlayerParameters parameters)方法，用户可以在 SoundPlayerParameters 数据结构中定义音量，循环次数，播放速度和优先级，另一种是使用默认播放参数的 play(int soundID)方法。短音播放成功后返回任务 ID，供后续对任务的管理。
5. 通过任务 ID，可以对短音播放任务进行暂停，恢复和停止。
6. 短音资源使用完毕需要调用 deleteSound(int soundID)完成对资源的释放。

下面的样例展示音频资源的加载与播放：

```
public void demo() {

    // 步骤 1：实例化对象

    SoundPlayer soundPlayer = new
SoundPlayer(AudioManager.AudioVolumeType.STREAM_MUSIC.getValue());

    // 步骤 2：指定音频资源加载并创建短音

    int soundId = soundPlayer.createSound("/system/xxx");

    // 步骤 3：指定音量，循环次数和播放速度

    SoundPlayerParameters parameters = new SoundPlayerParameters();

    parameers.setVolumes(new AudioVolumes());

    parameters.setLoop(10);

    parameters.setSpeed(1.0f);

    // 步骤 4：短音播放

    soundPlayer.play(soundId, parameters);

    // 步骤 5：停止播放

    soundPlayer.stop(soundId);

    // 步骤 6：释放短音资源
```

```
    soundPlayer.deleteSound(soundId);
}
```

tone 音的生成与播放

1. 通过 SoundPlayer()构造方法获取 SoundPlayer 类的实例化对象。
2. 使用 SoundPlayer 的实例化对象，通过 createSound(ToneDescriptor.ToneType type, int durationMs)方法，指定 tone 音类型和 tone 音播放时长来创建 tone 音资源。
3. 使用 SoundPlayer 的实例化对象，通过 play、pause、release 方法完成 tone 音播放，tone 音暂停和 tone 音资源释放。
4. 下面的样例展示 tone 音的生成与播放：

```
public void demo() {
    // 步骤 1：实例化对象
    SoundPlayer soundPlayer = new SoundPlayer();
    // 步骤 2：创建 DTMF_0（高频 1336Hz，低频 941Hz）持续时间 1000ms 的 tone 音
    soundPlayer.createSound(ToneDescriptor.ToneType.DTMF_0, 1000);
    // 步骤 3：tone 应播放，暂停和资源释放
    soundPlayer.play();
    soundPlayer.pause();
    soundPlayer.release();
}
```

系统音的播放

1. 通过 SoundPlayer(String packageName)构造方法获取 SoundPlayer 类的实例化对象。
2. 使用 SoundPlayer 的实例化对象，通过 playSound(SoundType type) 或 playSound(SoundType type, float volume)方法指定系统音类型和音量，并进行系统音播放。

下面的样例展示系统音的播放：

```
public void demo() {
    // 步骤 1：实例化对象
    SoundPlayer soundPlayer = new SoundPlayer("packageName");
    // 步骤 2：播放键盘敲击音，音量为 1.0
    soundPlayer.playSound(SoundType.KEY_CLICK, 1.0f);
}
```

媒体会话管理

概述

AVSession 是一套媒体播放控制框架，对媒体服务和界面进行解耦，并提供规范的通信接口，使应用可以自由、高效地在不同的媒体之间完成切换。

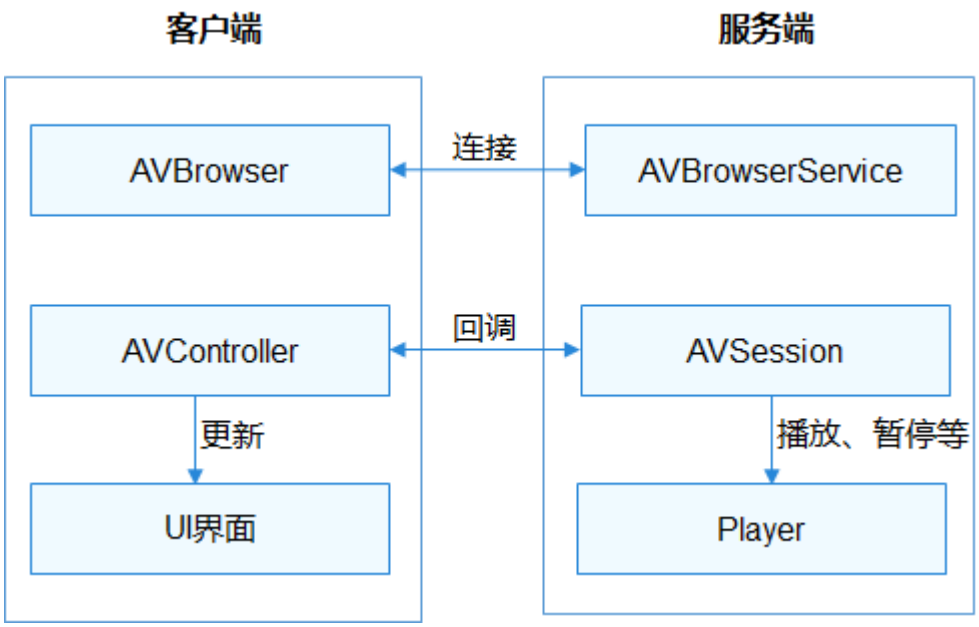
约束与限制

- 在使用完 AVSession 类后，需要及时进行资源释放。
- 调用 AVBrowser 的 subscribeByParentMediaId(String, AVSubscriptionCallback)之前，需要先执行 unsubscribeByParentMediaId(String)，防止重复订阅。
- 使用 AVBrowserService 的方法 onLoadAVElementList(String, AVBrowserResult)的 result 返回数据前，执行 detachForRetrieveAsync()。
- 播放器类需要使用 ohos.media.player.Player，否则无法正常接收按键事件。

媒体会话开发指导

场景介绍

AVSession 框架有四个主要的类，控制着整个框架的核心，下图简单的说明四个核心媒体框架控制类的关系。



- **AVBrowser**

媒体浏览器，通常在客户端创建，成功连接媒体服务后，通过媒体控制器 AVBrowser 向服务端发送播放控制指令。

其主要流程为，调用 connect 方法向 AVBrowserService 发起连接请求，连接成功后在回调方法 AVConnectionCallback.onConnected 中发起订阅数据请求，并在回调方法 AVSubscriptionCallback.onAVElementListLoaded 中保存请求的媒体播放数据。

- **AVController**

媒体控制器，在客户端 AVBrowser 连接服务成功后的回调方法 AVConnectionCallback.onConnected 中创建，用于向 Service 发送播放控制指令，并通过实现 AVControllerCallback 回调来响应服务端媒体状态变化，例如曲目信息变更、播放状态变更等，从而完成 UI 刷新。

- **AVBrowserService**

媒体浏览器服务，通常在服务端，通过媒体会话 AVSession 与媒体浏览器建立连接，并通过实现 Player 进行媒体播放。其中有两个重要的方法：

- onGetRoot，处理从媒体浏览器 AVBrowser 发来的连接请求，通过返回一个有效的 AVBrowserRoot 对象表示连接成功；
- onLoadAVElementList，处理从媒体浏览器 AVBrowser 发来的数据订阅请求，通过 AVBrowserResult.sendAVElementList(List<AVElement>) 方法返回媒体播放数据。

- **AVSession**

媒体会话，通常在 AVBrowserService 的 onStart 中创建，通过 setAVToken 方法设置到 AVBrowserService 中，并通过实现 AVSessionCallback 回调来接收和处理媒体控制器 AVController 发送的播放控制指令，如播放、暂停、跳转至上一曲、跳转至下一曲等。

除了上述四个类，AVSession 框架还有 AVElement。

- **AVElement**

媒体元素，用于将播放列表从 AVBrowserService 传递给 AVBrowser。

接口说明

表 1 AVBrowser 的主要接口

接口名	描述
AVBrowser(Context context, ElementName name, AVConnectionCallback callback, PacMap options)	构造 AVBrowser 实例，用于浏览 AVBrowserService 提供的媒体数据。
void connect()	连接 AVBrowserService。

表 1 AVBrowser 的主要接口

接口名	描述
void disconnect()	与 AVBrowserService 断开连接。
boolean isConnected()	判断当前是否已经与 AVBrowserService 连接。
ElementName getElementName()	获取 AVBrowserService 的 ohos.bundle.ElementName 实例。
String getRootMediaId()	获取默认媒体 id。
PacMap getOptions()	获取 AVBrowserService 提供的附加数据。
AVToken getAVToken()	获取媒体会话的令牌。
void getAVElement(String mediaId, AVElementCallback callback)	输入媒体的 id，查询对应的 ohos.media.common.sessioncore.AVElement 信息，查询结果会通过 callback 返回。
void subscribeByParentMediaId(String parentMediaId, AVSubscriptionCallback callback)	查询指定媒体 id 包含的所有媒体元素信息，并订阅它的媒体信息更新通知。
void subscribeByParentMediaId(String parentMediaId, PacMap options, AVSubscriptionCallback callback)	基于特定于服务的参数来查询指定媒体 id 中的媒体元素的信息，并订阅它的媒体信息更新通知。
void unsubscribeByParentMediaId(String parentMediaId)	取消订阅对应媒体 id 的信息更新通知。
void unsubscribeByParentMediaId(String parentMediaId, AVSubscriptionCallback callback)	取消订阅与指定 callback 相关的媒体 id 的信息更新通知。

表 2 AVBrowserService 的主要接口

接口名	描述
abstract AVBrowserRoot onGetRoot(String callerPackageName, int clientUid, PacMap options)	回调方法，用于返回应用程序的媒体内容的根信息，在AVBrowser.connect()后进行回调。
abstract void onLoadAVElementList(String parentMediaId, AVBrowserResult result)	回调方法，用于返回应用程序的媒体内容的结果信息 AVBrowserResult，其中包含了子节点的 AVElement 列表，在 AVBrowser 的方法 subscribeByParentMediaId 或 notifyAVElementListUpdated 执行后进行回调。
abstract void onLoadAVElement(String mediaId, AVBrowserResult result)	回调方法，用于获取特定的媒体项目 AVElement 结果信息，在 AVBrowser.getAVElement 方法执行后进行回调。
AVToken getAVToken()	获取 AVBrowser 与 AVBrowserService 之间的会话令牌。
void setAVToken(AVToken token)	设置 AVBrowser 与 AVBrowserService 之间的会话令牌。
final PacMap getBrowserOptions()	获取 AVBrowser 在连接 AVBrowserService 时设置的服务参数选项。
final AVRemoteUserInfo getCallerUserInfo()	获取当前发送请求的调用者信息。
void notifyAVElementListUpdated(String parentMediaId)	通知所有已连接的 AVBrowser 当前父节点的子节点已经发生改变。
void	通知所有已连接的 AVBrowser 当前

表 2 AVBrowserService 的主要接口

接口名	描述
notifyAVElementListUpdated(String parentId, PacMap options)	父节点的子节点已经发生改变，可设置服务参数。

表 3 AVController 的主要接口

接口名	描述
AVController(Context context, AVToken avToken)	构造 AVController 实例，用于应用程序与 AVSession 进行交互以控制媒体播放。
static boolean setControllerForAbility(Ability ability, AVController controller)	将媒体控制器注册到 ability 以接收按键事件。
boolean setAVControllerCallback(AVControllerCallback callback)	注册一个回调以接收来自 AVSession 的变更，例如元数据和播放状态变更。
boolean releaseAVControllerCallback(AVControllerCallback callback)	释放与 AVSession 之间的回调实例。
List<AVQueueElement> getAVQueueElement()	获取播放队列。
CharSequence getAVQueueTitle()	获取播放队列的标题。
AVPlaybackState getAVPlaybackState()	获取播放状态。
boolean dispatchAVKeyEvent(KeyEvent keyEvent）	应用分发媒体按键事件给会话以控制播放。

表 3 AVController 的主要接口

接口名	描述
void sendCustomCommand(String command, PacMap pacMap, GeneralReceiver receiverCb)	应用向 AVSession 发送自定义命令，参考 ohos.media.common.sessioncore.AVSessionCallback.onCommand。
IntentAgent getAVSessionAbility()	获取启动用户界面的 IntentAgent。
AVToken getAVToken()	获取应用连接到会话的令牌。此令牌用于创建媒体播放控制器。
void adjustAVPlaybackVolume(int direction, int flags)	调节播放音量。
void setAVPlaybackVolume(int value, int flags)	设置播放音量，要求支持绝对音量控制。
PacMap getOptions()	获取与此控制器连接的 AVSession 的附加数据。
long getFlags()	获取 AVSession 的附加标识，标记在 AVSession 中的定义。
AVMetadata getAVMetadata()	获取媒体资源的元数据 ohos.media.common.AVMetadata。
AVPlaybackInfo getAVPlaybackInfo()	获取播放信息。
String getSessionOwnerPackageName()	获得 AVSession 实例的应用程序的包名称。

表 3 AVController 的主要接口

接口名	描述
PacMap getAVSessionInfo()	获取会话的附加数据。
PlayControls getPlayControls()	获取一个 PlayControls 实例，将用于控制播放，比如控制媒体播放、停止、下一首等。

表 4 AVSession 的主要接口

接口名	描述
AVSession(Context context, String tag)	构造 AVSession 实例，用于控制媒体播放。
AVSession(Context context, String tag, PacMap sessionInfo)	构造带有附加会话信息的 AVSession 实例，用于控制媒体播放。
void setAVSessionCallback(AVSessionCallback callback)	设置回调函数来控制播放器，控制逻辑由应用实现。如果 callback 为 null 则取消控制。
boolean setAVSessionAbility(IntentAgent ia)	给 AVSession 设置一个 IntentAgent，用来启动用户界面。
boolean setAVButtonReceiver(IntentAgent ia)	为媒体按键接收器设置一个 IntentAgent，以便应用结束后，可以通过媒体按键重新拉起应用。
void enableAVSessionActive(boolean active)	设置是否激活媒体会话。当会话准备接收命令时，将输入参数设置为 true。如果会话停止接收命令，则设置为 false。
boolean isAVSessionActive()	查询会话是否激活。
void sendAVSessionEvent(String event, PacMap options)	向所有订阅此会话的控制器发送事件。

表 4 AVSession 的主要接口

接口名	描述
void release()	释放资源，应用播放完之后需调用。
AVToken getAVToken()	获取应用连接到会话的令牌。此令牌用于创建媒体播放控制器。
AVController getAVController()	获取会话构造时创建的控制器，方便应用使用。
void setAVPlaybackState(AVPlaybackState state)	设置当前播放状态。
void setAVMetadata(AVMetadata avMetadata)	设置媒体资源元数据 ohos.media.common.AVMetadata。
void setAVQueue(List<AVQueueElement> queue)	设置播放队列。
void setAVQueueTitle(CharSequence queueTitle)	设置播放队列的标题，UI 会显示此标题。
void setOptions(PacMap options)	设置此会话关联的附加数据。
AVCallerUserInfo getCurrentControllerInfo()	获取发送当前请求的媒体控制器信息。

表 5 AVElement 的主要接口

接口名	描述
AVElement(AVDescription description, int flags)	构造 AVElement 实例。
int getFlags()	获取 flags 的值。
boolean isScannable()	判断媒体是否可扫描，如：媒体有子节点，

表 5 AVElement 的主要接口

接口名	描述
	则可继续扫描获取子节点内容。
boolean isPlayable()	检查媒体是否可播放。
AVDescription getAVDescription()	获取媒体的详细信息。
String getMediaId()	获取媒体的 id。

开发步骤

使用 AVSession 媒体框架创建一个播放器示例，分为创建客户端和创建服务端。

创建客户端

在客户端 AVClientAbility 中声明 AVBrowser 和 AVController，并向服务端发送连接请求。

```
public class AVClientAbility extends Ability {
    // 媒体浏览器
    private AVBrowser avBrowser;
    // 媒体控制器
    private AVController avController;
    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
        // 用于指向媒体浏览器服务的包路径和类名
        ElementName elementName = new ElementName("",
"com.huawei.samples.audioplayer",
"com.huawei.samples.audioplayer.AVService");
```

```
        // connectionCallback 在调用 avBrowser.connect 方法后进行回调。
        avBrowser = new AVBrowser(context, elementName, connectionCallback,
null);
        // avBrowser 发送对媒体浏览器服务的连接请求。
        avBrowser.connect();
        // 将媒体控制器注册到 ability 以接收按键事件。
        AVController.setControllerForAbility(this, avController);
    }
}
```

AVConnectionCallback 回调接口中的方法为可选实现，通常需要会在 onConnected 中订阅媒体数据和创建媒体控制器 AVController。

```
// 发起连接（avBrowser.connect）后的回调方法实现
private AVConnectionCallback connectionCallback = new
AVConnectionCallback() {
    @Override
    public void onConnected() {
        // 成功连接媒体浏览器服务时回调该方法，否则回调 onConnectionFailed()。
        // 重复订阅会报错，所以先解除订阅。

avBrowser.unsubscribeByParentMediaId(avBrowser.getRootMediaId());
        // 第二个参数 AVSubscriptionCallback，用于处理订阅信息的回调。

avBrowser.subscribeByParentMediaId(AV_ROOTavBrowser.getRootMediaId(),
avSubscriptionCallback);
        AVToken token = avBrowser.getAVToken();
        avController = new AVController(AVClient.this, token); //
AVController 第一个参数为当前类的 context
        // 参数 AVControllerCallback，用于处理服务端播放状态及信息变化时回调。
```

```
        avController.setAVControllerCallback(avControllerCallback);
        // ...
    }
    // 其它回调方法（可选）
    // ...
};
```

通常在订阅成功时，在 AVSubscriptionCallback 回调接口 onAVElementListLoaded 中保存服务端回传的媒体列表。

```
// 发起订阅信息(avBrowser.subscribeByParentMediaId)后的回调方法实现
private AVSubscriptionCallback avSubscriptionCallback = new
AVSubscriptionCallback() {
    @Override
    public void onAVElementListLoaded(String parentId, List<AVElement>
children) {
        // 订阅成功时回调该方法，parentID 为标识，children 为服务端回传的媒体列
表
        super.onAVElementListLoaded(parentId, children);
        list.addAll(children);
        // ...
    }
};
```

AVControllerCallback 回调接口中的方法均为可选方法，主要用于服务端播放状态及信息的变化后对客户端的回调，客户端可在这些方法中实现 UI 的刷新。

```
// 服务对客户端的媒体数据或播放状态变更后的回调
private AVControllerCallback avControllerCallback = new
AVControllerCallback() {
```

```
    @Override
    public void onAVMetadataChanged(AVMetadata metadata) {
        // 当服务端调用 avSession.setAVMetadata(avMetadata)时，此方法会被回调。
        super.onAVMetadataChanged(metadata);
        AVDescription description = metadata.getAVDescription();
        String title = description.getTitle().toString();
        PixelMap pixelMap = description.getIcon();
        // ...
    }
    @Override
    public void onAVPlaybackStateChanged(AVPlaybackState playbackState) {
        // 当服务端调用 avSession.setAVPlaybackState(...)时，此方法会被回调。
        super.onAVPlaybackStateChanged(playbackState);
        long position = playbackState.getCurrentPosition();
        // ...
    }
    // 其它回调方法（可选）
    // ...
};
```

完成以上实现后，则应用可以在 UI 事件中调用 avController 的方法向服务端发送播放控制指令。

```
// 在 UI 播放与暂停按钮的点击事件中向服务端发送播放或暂停指令
public void toPlayOrPause() {
    switch (avController.getAVPlaybackState().getAVPlaybackState()) {
        case AVPlaybackState.PLAYBACK_STATE_NONE: {
```

```
            avController.getPlayControls().prepareToPlay();
            avController.getPlayControls().play();
            break;
        }
        case AVPlaybackState.PLAYBACK_STATE_PLAYING: {
            avController.getPlayControls().pause();
            break;
        }
        case AVPlaybackState.PLAYBACK_STATE_PAUSED: {
            avController.getPlayControls().play();
            break;
        }
        default: {
            // ...
        }
    }
}
```

其它播放控制根据业务是否需要实现，比如：

```
avController.getPlayControls().playNext();
avController.getPlayControls().playPrevious();
avController.getPlayControls().playFastForward();
avController.getPlayControls().rewind();
avController.getPlayControls().seekTo(1000);
avController.getPlayControls().stop();
// ...
```

也可以主动获取媒体信息、播放状态等数据：

```
AVMetadata avMetadata = avController.getAVMetadata();

AVPlaybackState avPlaybackState = avController.getAVPlaybackState();

// ...
```

创建服务端

在服务端 AVService 中声明 AVSession 和 Player。

```
public class AVService extends AVBrowserService {

    // 媒体会话

    private AVSession avSession;

    // 媒体播放器

    private Player player;


    @Override

    public void onStart(Intent intent) {

        super.onStart(intent);

        avSession = new AVSession(this, "AVService");

        setAVToken(avSession.getAVToken());

        // 设置 sessioncallback，用于响应客户端的媒体控制器发起的播放控制指令。

        avSession.setAVSessionCallback(avSessionCallback);

        // 设置播放状态初始状态为 AVPlaybackState.PLAYBACK_STATE_NONE。

        AVPlaybackState playbackState = new
AVPlaybackState.Builder().setAVPlaybackState(AVPlaybackState.PLAYBACK_S
TATE_NONE, 0, 1.0f).build();

        avSession.setAVPlaybackState(playbackState);

        // 完成播放器的初始化，如果使用多个 Player，也可以在执行播放时初始化。
```

```
        player = new Player(this);
    }
    @Override
    public AVBrowserRoot onGetRoot(String clientPackageName, int clientUid,
PacMap rootHints) {
        // 响应客户端 avBrowser.connect()方法。若同意连接，则返回有效的
AVBrowserRoot 实例，否则返回 null
        return new AVBrowserRoot(AV_ROOT, null);
    }
    @Override
    public void onLoadAVElementList(String parentId, AVBrowserResult
result) {
         LogUtil.info(TAG, "onLoadChildren");
         // 响应客户端 avBrowser.subscribeByParentMediaId(...)方法。
         // 先执行该方法 detachForRetrieveAsync()
         result.detachForRetrieveAsync();
         // externalAudioItems 缓存媒体文件，请开发者自行实现。
         result.sendAVElementList(externalAudioItems.getAudioItems());
    }
    @Override
    public void onLoadAVElementList(String s, AVBrowserResult
avBrowserResult, PacMap pacMap) {
         // 响应客户端 avBrowser.subscribeByParentMediaId(String, PacMap,
AVSubscriptionCallback)方法。
    }
    @Override
    public void onLoadAVElement(String s, AVBrowserResult avBrowserResult)
{
        // 响应客户端 avBrowser.getAVElement(String, AVElementCallback)方
法。
```

```
    }
}
```

响应客户端的媒体控制器发起的播放控制指令的回调实现。

```
private AVSessionCallback avSessionCallback = new AVSessionCallback() {
    @Override
    public void onPlay() {
        super.onPlay();
        // 当客户端调用 avController.getPlayControls().play()时，该方法会被回调。
        // 响应播放请求，开始播放。
        if (avSession.getAVController().getAVPlaybackState().getAVPlaybackState() == AVPlaybackState.PLAYBACK_STATE_PAUSED) {
            if (player.play()) {
                AVPlaybackState playbackState = new AVPlaybackState.Builder().setAVPlaybackState(
                    AVPlaybackState.PLAYBACK_STATE_PLAYING, player.getCurrentTime(),
                    player.getPlaybackSpeed()).build();
                avSession.setAVPlaybackState(playbackState);
            }
        }
    }
    @Override
    public void onPause() {
        super.onPause();
        // 当客户端调用 avController.getPlayControls().pause()时，该方法会被回调。
```

```
        // 响应暂停请求，暂停播放。
    }
    @Override
    public void onPlayNext() {
        super.onPlayNext();
        // 当客户端调用 avController.getPlayControls().playNext()时，该方法会被回调。
        // 响应播放下一曲请求，通过 avSession.setAVMetadata 设置下一曲曲目的信息。
        avSession.setAVMetadata(avNextMetadata);
    }
    // 重写以处理按键事件
    @Override
    public boolean onMediaButtonEvent(Intent mediaButtonIntent) {
        KeyEvent ke = mediaButtonIntent.getParcelableParam(AVSession.PARAM_KEY_EVENT);
        if (ke == null) {
            LogUtil.error("onMediaButtonEvent", "getParcelableParam failed");
            return false;
        }
        if (ke.isKeyDown()) {
            // 只处理按键抬起事件
            return true;
        }

        switch (ke.getKeyCode()) {
            case KeyEvent.KEY_MEDIA_PLAY_PAUSE: {
                if (playbackState.getAVPlaybackState() == AVPlaybackState.PLAYBACK_STATE_PAUSED) {
```

```
                    onPlay();
                    break;
                }
                if (playbackState.getAVPlaybackState() ==
AVPlaybackState.PLAYBACK_STATE_PLAYING) {
                    onPause();
                    break;
                }
                break;
            }
            case KeyEvent.KEY_MEDIA_PLAY: {
                onPlay();
                break;
            }
            case KeyEvent.KEY_MEDIA_PAUSE: {
                onPause();
                break;
            }
            case KeyEvent.KEY_MEDIA_STOP: {
                onStop();
                break;
            }
            case KeyEvent.KEY_MEDIA_NEXT: {
                onPlayNext();
                break;
            }
            case KeyEvent.KEY_MEDIA_PREVIOUS: {
                onPlayPrevious();
```

```
                break;
            }
            default: {
                break;
            }
        }
        return true;
    }
    // 其它回调方法（可选）
    // ...
}
```

安全

权限

概述

基本概念

- **应用沙盒**

系统利用内核保护机制来识别和隔离应用资源，可将不同的应用隔离开，保护应用自身和系统免受恶意应用的攻击。默认情况下，应用间不能彼此交互，而且对系统的访问会受到限制。例如，如果应用 A（一个单独的应用）尝试在没有权限的情况下读取应用 B 的数据或者调用系统的能力拨打电话，操作系统会阻止此类行为，因为应用 A 没有被授予相应的权限。

- **应用权限**

由于系统通过沙盒机制管理各个应用，在默认规则下，应用只能访问有限的系统资源。但应用为了扩展功能的需要，需要访问自身沙盒之外的系统或其他应用的数据（包括用户个人数据）或能力；系统或应用也必须以明确的方式对外提供接口来共享其数据或能力。为了保证这些数据或能力不会被不当或恶意使用，就需要有一种访问控制机制来保护，这就是应用权限。

应用权限是程序访问操作某种对象的许可。权限在应用层面要求明确定义且经用户授权，以便系统化地规范各类应用程序的行为准则与权限许可。

- **权限保护的对象**

权限保护的对象可以分为数据和能力。数据包含了个人数据（如照片、通讯录、日历、位置等）、设备数据（如设备标识、相机、麦克风等）、应用数据；能力包括了设备能力（如打电话、发短信、联网等）、应用能力（如弹出悬浮框、创建快捷方式等）等。

- **权限开放范围**

权限开放范围指一个权限能被哪些应用申请。按可信程度从高到低的顺序，不同权限开放范围对应的应用可分为：系统服务、系统应用、系统预置特权应用、同签名应用、系统预置普通应用、持有权限证书的后装应用、其他普通应用，开放范围依次扩大。

- **敏感权限**

涉及访问个人数据（如：照片、通讯录、日历、本机号码、短信等）和操作敏感能力（如：相机、麦克风、拨打电话、发送短信等）的权限。

- **应用核心功能**

一个应用可能提供了多种功能，其中应用为满足用户的关键需求而提供的功能，称为应用的核心功能。这是一个相对宽泛的概念，本规范用来辅助描述用户权限授权的预期。用户选择安装一个应用，通常是被应用的核心功能所吸引。比如导航类应用，定位导航就是这种应用的核心功能；比如媒体类应用，播放以及媒体资源管理就是核心功能，这些功能所需要的权限，用户在安装时内心已经倾向于授予（否则就不会去安装）。与核心功能相对应的是辅助功能，这些功能所需要的权限，需要向用户清晰说明目的、场景等信息，由用户授权。既不属于核心功能，也不是支撑核心功能的辅助功能，就是多余功能。不少应用存在并非为用户服务的功能，这些功能所需要的权限通常被用户禁止。

- **最小必要权限**

保障应用某一服务类型正常运行所需要的应用权限的最小集，一旦缺少将导致该类型服务无法实现或无法正常运行的应用权限。

运作机制

系统所有应用均在应用沙盒内运行。默认情况下，应用只能访问有限的系统资源。这些限制是通过 DAC（Discretionary Access Control）、MAC（Mandatory Access Control）以及本文描述的应用权限机制等多种不同的形式实现的。因应用需要实现其某些功能而必须访问系统或其他应用的数据或操作某些器件，此时就需要系统或其他应用能提供接口，考虑到安全，就需要对这些接口采用一种限制措施，这就是称为“应用权限”的安全机制。

接口的提供涉及到其权限的命名和分组、对外开放的范围、被授予的应用、以及用户的参与和体验。应用权限管理模块的目的就是负责管理由接口提供方（访问客体）、接口使用方（访问主体）、系统（包括云侧和端侧）和用户等共同参与的整个流程，保证受限接口是在约定好的规则下被正常使用，避免接口被滥用而导致用户、应用和设备受损。

约束与限制

- 同一应用自定义权限个数不能超过 1024 个。
- 同一应用申请权限个数不能超过 1024 个。
- 为避免与系统权限名冲突，应用自定义权限名不能以 ohos 开头，且权限名长度不能超过 256 字符。
- 自定义权限授予方式不能为 user_grant。
- 自定义权限开放范围不能为 restricted。含义见表 2。

开发指导

场景介绍

HarmonyOS 支持开发者自定义权限来保护能力或接口，同时开发者也可申请权限来访问受权限保护的对象。

权限申请

开发者需要在 config.json 文件中的“reqPermissions”字段中声明所需要的权限。

```
{
    "reqPermissions": [{
        "name": "ohos.permission.CAMERA",
        "reason": "$string:permreason_camera",
        "usedScene": {
            "ability": ["com.mycamera.Ability",
"com.mycamera.AbilityBackground"],
            "when": "always"
        }
    },{
    ...
    }]
}
```

权限申请格式采用数组格式，可支持同时申请多个权限，权限个数最多不能超过1024 个。

表 1 reqPermissions 权限申请字段说明

键	值说明	类型	取值范围	默认值	规则约束
name	必须，填写需要使用的权限名称。	字符串	自定义	无	未填写时，解析失败。
reason	可选，当申请的权限为 user_grant 权限时此字段必填。描述申请权限的原因。	字符串	显示文字长度不能超过 256 个字节。	空	user_grant 权限必填，否则不允许在应用市场上架。需做多语种适配。
usedScene	可选，当申请的权限为 user_grant 权限时此字段必填。描述权限使用的场景和时机。场景类型有：ability、when（调用时机）。可配置多个 ability。	ability：字符串数组 when：字符串	ability：ability 的名称 when：inuse（使用时）、always（始终）	ability：空 when：inuse	user_grant 权限必填 ability，可选填 when。

如果声明使用的权限的 grantMode 是 system_grant，则权限会在当应用安装的时候被自动授予。

如果声明使用的权限的 grantMode 是 user_grant，则必须经用户手动授权（用户在弹框中授权或进入权限设置界面授权）才可使用。用户会看到 reason 字段中填写的理由，来帮助用户决定是否给予授权。

说明

对于授权方式为 user_grant 的权限，每一次执行需要这一权限的操作时，都需要检查自身是否有该权限。当自身具有权限时，才可继续执行，否则应用需要请求用户授予权限。示例参见动态申请权限开发步骤。

自定义权限

开发者需要在 config.json 文件中的“defPermissions”字段中自定义所需的权限：

```
{
    "defPermissions": [{
        "name": "com.myability.permission.MYPERMISSION",
        "grantMode": "system_grant",
        "availableScope": ["signature"]
    }, {
    ...
    }]
}
```

权限定义格式采用数组格式，可支持同时定义多个权限，自定义的权限个数最多不能超过 1024 个。权限定义的字段描述详见表 2。

表 2 defPermissions 权限定义字段说明

键	值说明	类型	取值范围	默认值	规则约束
name	必填，权限名称。为最大可能避免重名，采用反向域公司名+应用名+权限名组合。	字符串	自定义	无	第三方应用不允许填写系统存在的权限，否则安装失败。未填写解析失败。权限名长度不能超过 256 个字符。
grantMode	必填，权限授予方式。	字符串	user_grant（用户授权） system_grant（系统授权） 取值含义参见：表 3。	system_grant	未填值或填写了取值范围以外的值时，自动赋予默认值；不允许第三方应用填写 user_grant，填写后会自动赋予默认值。
availableScope	选填，权限限制范围。不填则表示此权限对所有应用开放。	字符串数组	signature privileged restricte	空	填写取值范围以外的值时，权限限制范围不生效。 由于第三方

表 2 defPermissions 权限定义字段说明

键	值说明	类型	取值范围	默认值	规则约束
			d 可参见。取值含义请见：表 4。		应用并不在 restricted 的范围内，很少会出现权限定义者不能访问自身定义的权限的情况，所以不允许三方应用填写 restricted。
label	选填，权限的简短描述，若未填写，则使用到简短描述的地方由权限名取代。	字符串	自定义	空	需要多语种适配。
description	选填，权限的详细描述，若未填写，则使用到详细描述的地方由 label 取代。	字符串	自定义	空	需要多语种适配。

表 3 权限授予方式字段说明

授予方式(grantMode)	说明	自定义权限是否可指定该级别	取值样例

表 3 权限授予方式字段说明

授予方式(grantMode)	说明	自定义权限是否可指定该级别	取值样例
system_grant	在“config.json”里面声明，安装后系统自动授予。	是	GET_NETWORK_INFO 、GET_WIFI_INFO
user_grant	在“config.json”里面声明，并在使用时动态申请，用户授权后才可使用。	否，如自定义则强制修改为system_grant。	CAMERA、MICROPHONE

表 4 权限限制范围字段说明

权限范围(availableScope)	说明	自定义权限是否可指定该级别	取值样例
restricted	需要开发者向华为申请后才能被使用的特殊权限。	否	ANSWER_CALL、READ_CALL_LOG、RECEIVE_SMS
signature	权限定义方和使用方的签名一致。需在“config.json”里面声明后，由权限管理模块负责签名校验一致后，可使用。	是	对应用（或 Ability）操作的系统接口上由系统定义权限以及应用自定义的权限。 如：find 某 Ability，连接某 Ability。
privileged	预置在系统版本中的特权应用可申请的权限。	是	SET_TIME、MANAGE_USER_STORAGE

访问权限控制

- **Ability 的访问权限控制**

在 config.json 中填写“abilities”到“permissions”字段，即只有拥有该权限的应用可访问此 Ability。下面的例子表明只有拥有“ohos.permission.CAMERA”权限的应用可以访问此 ability。

```
"abilities": [{
       "name": ".MainAbility",
       "description": "$string:description_main_ability",
       "icon": "$media:hiworld.png",
       "label": "HiCamera",
       "launchType": "standard",
       "orientation": "portrait",
       "visible": false,
       "permissions": [
           "ohos.permission.CAMERA"
       ],
   }]
```

表 5 权限保护字段说明

键	值说明	类型	取值范围	默认值	规则约束
permissions	选填，权限名称。用以表示此 ability 受哪个权限保护，即只有拥有此权	字符串	自定义	无	目前仅支持填写一个权限名，若填写多个权限名，仅第一个权限名称

表 5 权限保护字段说明

键	值说明	类型	取值范围	默认值	规则约束
	限的应用可访问此ability。				有效。

- **Ability 接口的访问权限控制**

在 Ability 实现中，如需要对特定接口对调用者做访问控制，可在服务侧的接口实现中，主动通过 verifyCallingPermission、verifyCallingOrSelfPermission 来检查访问者是否拥有所需要的权限。

```
if (verifyCallingPermission("ohos.permission.CAMERA") !=
IBundleManager.PERMISSION_GRANTED) {
    // 调用者无权限，做错误处理
}
    // 调用者权限校验通过，开始提供服务
```

表 5 权限保护字段说明

键	值说明	类型	取值范围	默认值	规则约束
permissions	选填，权限名称。用以表示此ability 受哪个权限保护，即只有	字符串	自定义	无	目前仅支持填写一个权限名，若填写多个权限名，仅第一个

表 5 权限保护字段说明

键	值说明	类型	取值范围	默认值	规则约束
	拥有此权限的应用可访问此 ability。				权限名称有效。

- **Ability 接口的访问权限控制**

在 Ability 实现中，如需要对特定接口对调用者做访问控制，可在服务侧的接口实现中，主动通过 verifyCallingPermission、verifyCallingOrSelfPermission 来检查访问者是否拥有所需要的权限。

```
if (verifyCallingPermission("ohos.permission.CAMERA") != IBundleManager.PERMISSION_GRANTED) {
    // 调用者无权限，做错误处理
}
    // 调用者权限校验通过，开始提供服务
```

API 接口说明

表 6 应用权限接口说明

接口原型	接口详细描述
public int verifyPermission(String permissionName, int pid, int uid)	接口功能：查询指定 PID、UID 的应用是否已被授予某权限 输入参数：permissionName：权限名；pid：进程 id；uid：uid

表 6 应用权限接口说明

接口原型	接口详细描述
	输出参数：无 返回值: IBundleManager.PERMISSION_DENIED、 IBundleManager.PERMISSION_GRANTED
public int verifyCallingPermission(String permissionName)	接口功能：查询 IPC 跨进程调用方的进程是否已被授予某权限 输入参数：permissionName：权限名 输出参数：无 返回值: IBundleManager.PERMISSION_DENIED、 IBundleManager.PERMISSION_GRANTED
public int verifySelfPermission(String permissionName)	接口功能：查询自身进程是否已被授予某权限 输入参数：permissionName：权限名 输出参数：无 返回值: IBundleManager.PERMISSION_DENIED、 IBundleManager.PERMISSION_GRANTED
public int verifyCallingOrSelfPermission(String permissionName)	接口功能：当有远端调用检查远端是否有权限，否则检查自身是否拥有权限 输入参数：permissionName：权限名 输出参数：无 返回值: IBundleManager.PERMISSION_DENIED、 IBundleManager.PERMISSION_GRANTED
public boolean canRequestPermission(String permissionName)	接口功能：向系统权限管理模块查询某权限是否不再弹框授权了 输入参数：permissionName：权限名 输出参数：无 返回值：true 允许弹框，false 不允许弹框

表 6 应用权限接口说明

接口原型	接口详细描述
void requestPermissionsFromUser (String[] permissions, int requestCode)	接口功能：向系统权限管理模块申请权限（接口可支持一次申请多个。若下一步操作涉及到多个敏感权限，可以这么用，其他情况建议不要这么用。因为弹框还是按权限组一个个去弹框，耗时比较长。用到哪个权限就去申请哪个） 输入参数： permissionNames:权限名列表；requestCode: 请求应答会带回此编码以匹配本次申请的权限请求 输出参数：无 返回值：无
void onRequestPermissionsFromUserResult (int requestCode, String[] permissions, int[] grantResults)	接口功能：调用 requestPermissionsFromUser 后的应答接口 输入参数：requestCode：requestPermission 中传入的 requestCode；permissions：申请的权限名；grantResults：申请权限的结果 输出参数：无 返回值：无

动态申请权限开发步骤

```
{
    " reqPermissions": [{
        "name": "ohos.permission.CAMERA",
        "reason": "$string:permreason_camera",
        "usedScene": {
```

```
            "ability": ["com.mycamera.Ability",
"com.mycamera.AbilityBackground"],
            "when": "always"}
        }, {
        ...
        }]
    }]
}
```

1. 使用 ohos.app.Context.verifySelfPermission 接口查询应用是否已被授予该权限。

- 如果已被授予权限，可以结束权限申请流程。
- 如果未被授予权限，继续执行下一步。

2. 使用 canRequestPermission 查询是否可动态申请。

- 如果不可动态申请，说明已被用户或系统永久禁止授权，可以结束权限申请流程。
- 如果可动态申请，继续执行下一步。

3. 使用 requestPermissionFromUser 动态申请权限，通过回调函数接受授予结果。

样例代码如下：

```
if (verifySelfPermission("ohos.permission.CAMERA") !=
IBundleManager.PERMISSION_GRANTED) {
    // 应用未被授予权限
```

```
    if (canRequestPermission("ohos.permission.CAMERA")) {
        // 是否可以申请弹框授权(首次申请或者用户未选择禁止且不再提示)
        requestPermissionsFromUser(
                new String[] { "ohos.permission.CAMERA" } ,
MY_PERMISSIONS_REQUEST_CAMERA);
    } else {
        // 显示应用需要权限的理由，提示用户进入设置授权
    }
} else {
    // 权限已被授予
}


@override
public void onRequestPermissionsFromUserResult (int requestCode, String[]
permissions, int[] grantResults){
    switch (requestCode) {
        case MY_PERMISSIONS_REQUEST_CAMERA: {
            // 匹配 requestPermissions 的 requestCode
            if (grantResults.length > 0
                && grantResults[0] == IBundleManager.PERMISSION_GRANTED) {
                // 权限被授予
                // 注意：因时间差导致接口权限检查时有无权限，所以对那些因无权限
而抛异常的接口进行异常捕获处理
            } else {
                // 权限被拒绝
            }
            return;
        }
    }
```

```
}
```

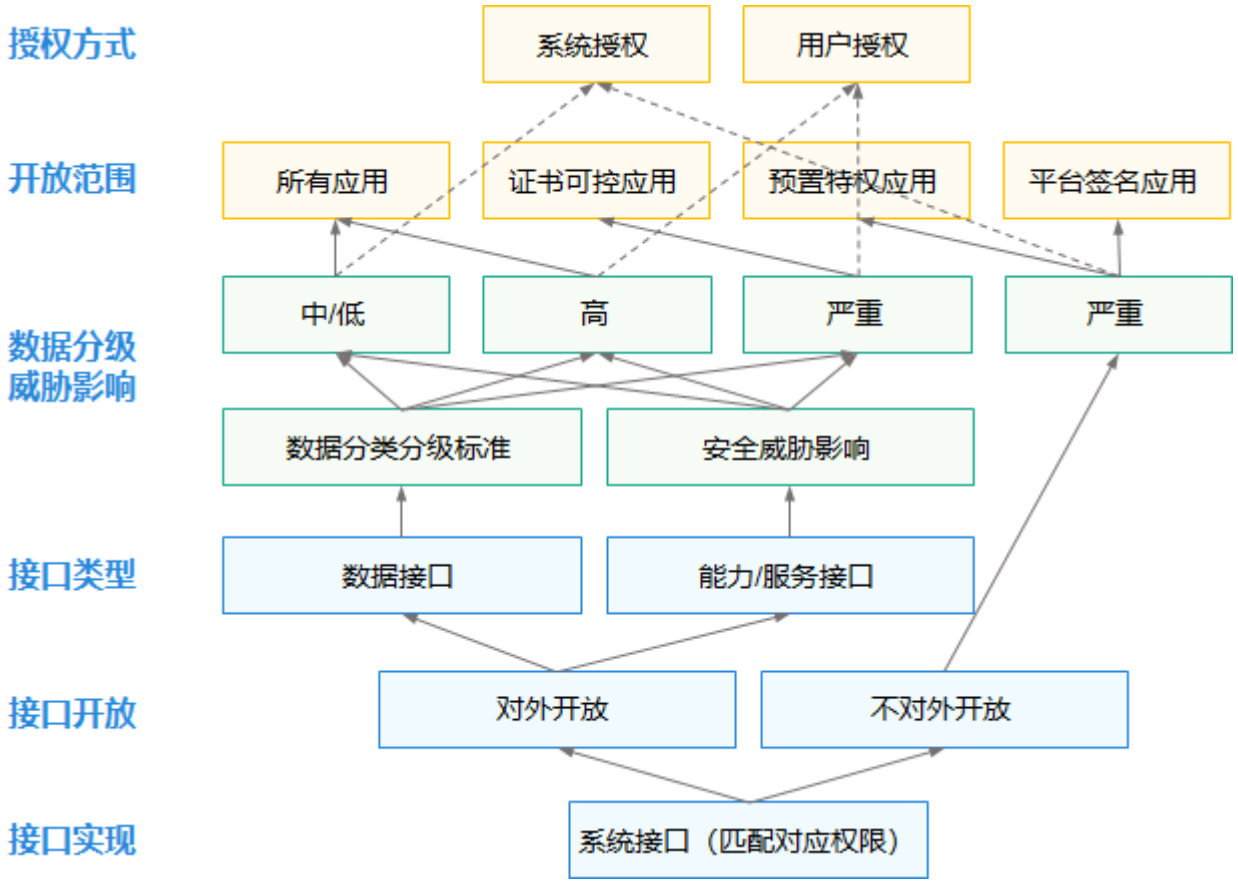
应用权限列表

权限分类分级模型

HarmonyOS 的应用权限严格按照权限分类分级模型进行定义，如图 1 所示，具体过程可分为三步：

1. 根据不同应用所需实现的功能，明确接口是否需要对外开放。
2. 根据接口所涉数据的敏感程度或所涉能力的安全威胁影响，对所有的开放接口进行分级（包括中、低、高、严重）。不对外开放的接口均为严重级别。
3. 根据不同的分级，确定权限的开放范围与授权方式。

图 1　权限分类分级模型



HarmonyOS 已定义的权限列表详见《API 参考》中的“ohos.security.SystemPermission”。下面重点介绍对所有应用开放的 HarmonyOS 的应用权限。

敏感权限

敏感权限的申请需要按照动态申请流程向用户申请授权。

表 1 敏感权限说明

权限分类名称	权限名	说明
位置	ohos.permission.LOCATION	允许应用在前台运行时获取位置信息。如果应用在后台运行时也要获取位置信息，则需要同时申请 ohos.permission.LOCATION_IN_BACKGROUND 权限。
	ohos.permission.LOCATION_IN_BACKGROUND	允许应用在后台运行时获取位置信息，需要同时申请 ohos.permission.LOCATION 权限。
相机	ohos.permission.CAMERA	允许应用使用相机拍摄照片和录制视频。
麦克风	ohos.permission.MICROPHONE	允许应用使用麦克风进行录音。
日历	ohos.permission.READ_CALENDAR	允许应用读取日历信息。
	ohos.permission.WRITE_CALENDAR	允许应用在设备上添加、移除或修改日历活动。
健身	ohos.permission.ACTIVITY_MOTION	允许应用读取用户当前的运动状态。

表 1 敏感权限说明

权限分类名称	权限名	说明
运动		
健康	ohos.permission.READ_HEALTH_DATA	允许应用读取用户的健康数据。
媒体	ohos.permission.MEDIA_LOCATION	允许应用访问用户媒体文件中的地理位置信息。
	ohos.permission.READ_MEDIA	允许应用读取用户外部存储中的媒体文件信息。
	ohos.permission.WRITE_MEDIA	允许应用读写用户外部存储中的媒体文件信息。
帐号	ohos.permission.GET_APP_ACCOUNTS	允许应用访问系统帐号的分布式信息权限。

非敏感权限

非敏感权限不涉及用户的敏感数据或危险操作，仅需在 config.json 中声明，应用安装后即被授权。

表 2 非敏感权限说明

权限名	说明
ohos.permission.GET_NETWORK_INFO	允许应用获取数据网络信息。
ohos.permission.GET_WIFI_INFO	允许获取 WLAN 信息。
ohos.permission.USE_BLUETOOTH	允许应用查看蓝牙的配置。
ohos.permission.DISCOVER_BLUETOOTH	允许应用配置本地蓝牙，并允许其查找远端设备且与之配对连接。
ohos.permission.SET_NETWORK_INFO	允许应用控制数据网络。
ohos.permission.SET_WIFI_INFO	允许配置 WLAN 设备。
ohos.permission.SPREAD_STATUS_BAR	允许应用以缩略图方式呈现在状态栏。
ohos.permission.INTERNET	允许使用网络 socket。
ohos.permission.MODIFY_AUDIO_SETTINGS	允许应用程序修改音频设置。
ohos.permission.RECEIVER_STARTUP_COMPLETED	允许应用接收设备启动完成广播。
ohos.permission.RUNNING_LOCK	允许申请休眠运行锁，并执行相关操作。
ohos.permission.ACCESS_BIOMETRIC	允许应用使用生物识别能力进行身份认证。
ohos.permission.RCV_NFC_TRANSACTION_EVENT	允许应用接收卡模拟交易事件。
ohos.permission.COMMONEVENT_STICKY	允许发布粘性公共事件的权限。
ohos.permission.SYSTEM_FLOAT_WINDOW	提供显示悬浮窗的能力。
ohos.permission.VIBRATE	允许应用程序使用马达。

表 2 非敏感权限说明

权限名	说明
ohos.permission.USE_TRUSTCIRCLE_MANAGER	允许调用设备间认证能力。
ohos.permission.USE_WHOLE_SCREEN	允许通知携带一个全屏 IntentAgent。
ohos.permission.SET_WALLPAPER	允许设置静态壁纸。
ohos.permission.SET_WALLPAPER_DIMENSION	允许设置壁纸尺寸。
ohos.permission.REARRANGE__MISSIONS	允许调整任务栈。
ohos.permission.CLEAN_BACKGROUND_PROCESSES	允许根据包名清理相关后台进程。
ohos.permission.KEEP_BACKGROUND_RUNNING	允许 Service Ability 在后台继续运行。
ohos.permission.GET_BUNDLE_INFO	查询其他应用的信息。
ohos.permission.ACCELEROMETER	允许应用程序读取加速度传感器的数据。
ohos.permission.GYROSCOPE	允许应用程序读取陀螺仪传感器的数据。
ohos.permission.MULTIMODAL_INTERACTIVE	允许应用订阅语音或手势事件。
ohos.permission.radio.ACCESS_FM_AM	允许用户获取收音机相关服务。
ohos.permission.NFC_TAG	允许应用读写 Tag 卡片。
ohos.permission.NFC_CARD_EMULATION	允许应用实现卡模拟功能。

受限开放的权限

受限开放的权限通常是不允许三方应用申请的。如果有特殊场景需要使用，请提供相关申请材料到应用市场申请相应权限证书。如果应用未申请相应的权限证书，却试图在 config.json 文件中声明此类权限，将会导致应用安装失败。另外，由于此类权限涉及到用户敏感数据或危险操作，当应用申请到权限证书后，还需按照动态申请权限的流程向用户申请授权。

表 3 受限开放权限说明

权限分类名称	权限名	说明
信息	ohos.permission.READ_MESSAGES	允许应用读取短信息。
	ohos.permission.RECEIVE_MMS	允许应用接收彩信。
	ohos.permission.RECEIVE_SMS	允许应用接收短信息。
	ohos.permission.RECEIVE_WAP_MESSAGES	允许应用接收 WAP 消息。
	ohos.permission.SEND_MESSAGES	允许应用发送短信。
	ohos.permission.READ_CELL_MESSAGES	允许应用读取小区广播消息。
通话记录	ohos.permission.READ_CALL_LOG	允许应用读取通话记录。
	ohos.permission.WRITE_CALL_LOG	允许应用在设备上添加、修改和删除通话记录。
通讯录	ohos.permission.READ_CONTACTS	允许应用读取联系人数据。
	ohos.permission.WRITE_CONTACTS	允许应用添加、移除和更改联系人数据。
电话	ohos.permission.ANSWER_CALL	允许应用接听来电。

生物特征识别

概述

提供生物特征识别认证能力，即基于人体固有的生理特征和行为特征来识别用户身份，供第三方应用调用，可应用于设备解锁、支付、应用登录等身份认证场景。

当前生物特征识别能力提供 2D 人脸识别、3D 人脸识别两种人脸识别能力，设备具备哪种识别能力，取决于设备的硬件能力和技术实现。3D 人脸识别技术识别率、防伪能力都优于 2D 人脸识别技术，但具有 3D 人脸能力（比如 3D 结构光、3D TOF 等）的设备才可以使用 3D 人脸识别技术。

基本概念

生物特征识别（又叫生物认证）：通过计算机与光学、声学、生物传感器和生物统计学原理等高科技手段密切结合，利用人体固有的生理特性（如指纹、面容、虹膜等）和行为特征（如笔迹、声音、步态等）来进行个人身份的鉴定。

人脸识别：基于人的脸部特征信息进行身份识别的一种生物特征识别技术，用摄像机或摄像头采集含有人脸的图像或视频流，并自动在图像中检测和跟踪人脸，进而对检测到的人脸进行脸部识别，通常也叫做人像识别、面部识别、人脸认证。

运作机制

人脸识别会在摄像头和 TEE（Trusted Execution Environment）之间建立安全通道，人脸图像信息通过安全通道传递到 TEE 中，由于人脸图像信息从 REE（Rich Execution Environment）侧无法获取，从而避免了恶意软件从 REE 侧进行攻击。对人脸图像采集、特征提取、活体检测、特征比对等处理完全在 TEE 中，基于 TrustZone 进行安全隔离，外部的人脸框架只负责人脸的认证发起和处理认证结果等数据，不涉及人脸数据本身。

人脸特征数据通过 TEE 的安全存储区进行存储，采用高强度的密码算法对人脸特征数据进行加密和完整性保护，外部无法获取到加密人脸特征数据的密钥，保证

用户的人脸特征数据不会泄露。本能力采集和存储的人脸特征数据不会在用户未授权的情况下被传出 TEE，这意味着，用户未授权时，无论是系统应用还是三方应用都无法获得人脸特征数据，也无法将人脸特征数据传送或备份到任何外部存储介质。

约束与限制

- 当前版本提供的生物特征识别能力只包含人脸识别，且只支持本地认证，不提供认证界面。
- 要求设备上具备摄像器件，且人脸图像像素大于 100*100。
- 要求设备上具有 TEE 安全环境，人脸特征信息高强度加密保存在 TEE 中。
- 对于面部特征相似的人（比如双胞胎、兄弟姐妹等）、面部特征不断发育的儿童，人脸特征匹配率有所不同。如果对此担忧，可考虑其他认证方式。

开发指导

场景介绍

当前生物特征识别支持 2D 人脸识别、3D 人脸识别，可应用于设备解锁、应用登录、支付等身份认证场景。

接口说明

BiometricAuthentication 类提供了生物认证的相关方法，包括检测认证能力、认证和取消认证等，用户可以通过人脸等生物特征信息进行认证操作。在执行认证前，需要检查设备是否支持该认证能力，具体指认证类型、安全级别和是否本地认证。如果不支持，需要考虑使用其他认证能力。

表 1 生物特征识别开放能力列表：

接口名	功能描述
getInstance(Ability ability)	获取 BiometricAuthentication 的单例对象。
checkAuthenticationAvailability(AuthType type, SecureLevel level, boolean isLocalAuth)	检测设备是否具有生物认证能力。
execAuthenticationAction(AuthType type, SecureLevel level, boolean isLocalAuth,boolean isAppAuthDialog, SystemAuthDialogInfo information)	调用者使用该方法进行生物认证。可以使用自定义的认证界面，也可以使用系统提供的认证界面。当使用系统认证界面时，调用者可以自定义提示语。该方法直到认证结束才返回认证结果。
getAuthenticationTips()	获取生物认证过程中的提示信息。
cancelAuthenticationAction()	取消生物认证操作。
setSecureObjectSignature(Signature sign)	设置需要关联认证结果的 Signature 对象，

表 1 生物特征识别开放能力列表：

接口名	功能描述
	在进行认证操作后，如果认证成功则 Signature 对象被授权可以使用。设置前 Signature 对象需要正确初始化，且配置为认证成功才能使用。
getSecureObjectSignature()	在认证成功后，可通过该方法获取已授权的 Signature 对象。如果未设置过 Signature 对象，则返回 null。
setSecureObjectCipher(Cipher cipher)	设置需要关联认证结果的 Cipher 对象，在进行认证操作后，如果认证成功则 Cipher 对象被授权可以使用。设置前 Cipher 对象需要正确初始化，且配置为认证成功才能使用。
getSecureObjectCipher()	在认证成功后，可通过该方法获取已授权的 Cipher 对象。如果未设置过 Cipher 对象，则返回 null。
setSecureObjectMac(Mac mac)	设置需要关联认证结果的 Mac 对象，在进行认证操作后，如果认证成功则 Mac 对象被授权可以使用。设置前 Mac 对象需要正确初始化，且配置为认证成功才能使用。
getSecureObjectMac()	在认证成功后，可通过该方法获取已授权的 Mac 对象。如果未设置过 Mac 对象，则返回 null。

开发步骤

开发前请完成以下准备工作：

1. 在应用配置权限文件中，增加 ohos.permission.ACCESS_BIOMETRIC 的权限声明。

2. 在使用生物特征识别认证能力的代码文件中增加 import ohos.biometrics.authentication.BiometricAuthentication。

开发过程：

1. 获取 BiometricAuthentication 的单例对象，代码示例如下：

```
BiometricAuthentication  mBiometricAuthentication =
BiometricAuthentication.getInstance(MainAbility.mAbility);
```

检测设备是否具有生物认证能力：

2D 人脸识别建议使用 SECURE_LEVEL_S2，3D 人脸识别建议使用

SECURE_LEVEL_S3。代码示例如下：

```
int retChkAuthAvb =
mBiometricAuthentication.checkAuthenticationAvailability(

BiometricAuthentication.AuthType.AUTH_TYPE_BIOMETRIC_FACE_ONLY,
BiometricAuthentication.SecureLevel.SECURE_LEVEL_S2, true);
```

（可选）设置需要关联认证结果的 Signature 对象或 Cipher 对象或 Mac 对象，

代码示例如下：

```
// 定义一个 Signature 对象 sign;

mBiometricAuthentication.setSecureObjectSignature(sign);


// 定义一个 Cipher 对象 cipher;

mBiometricAuthentication.setSecureObjectCipher(cipher);


// 定义一个 Mac 对象 mac;

mBiometricAuthentication.setSecureObjectMac(mac);
```

在新线程里面执行认证操作，避免阻塞其他操作，代码示例如下：

```
new Thread(new Runnable() {

    @Override

    public void run() {

        int retExcAuth;

        retExcAuth =
mBiometricAuthentication.execAuthenticationAction(       BiometricAuthe
ntication.AuthType.AUTH_TYPE_BIOMETRIC_FACE_ONLY,
BiometricAuthentication.SecureLevel.SECURE_LEVEL_S2, true, false, null);

    }

}).start();
```

获得认证过程中的提示信息，代码示例如下：

```
AuthenticationTips  mTips =
mBiometricAuthentication.getAuthenticationTips();
```

（可选）认证成功后获取已设置的 Signature 对象或 Cipher 对象或 Mac 对象，代码示例如下：

```
Signature  sign = mBiometricAuthentication.getSecureObjectSignature();


Cipher cipher = mBiometricAuthentication.getSecureObjectCipher();


Mac mac = mBiometricAuthentication.getSecureObjectMac();
```

认证过程中取消认证，代码示例如下：

```
int  ret = mBiometricAuthentication.cancelAuthenticationAction();
```

AI

码生成

概述

码生成能够根据开发者给定的字符串信息和二维码图片尺寸，返回相应的二维码图片字节流。调用方可以通过二维码字节流生成二维码图片。

约束与限制

- 当前仅支持生成 QR 二维码（Quick Response Code）。由于 QR 二维码算法的限制，字符串信息的长度不能超过 2953 个字符。
- 生成的二维码图片的宽度不能超过 1920 像素，高度不能超过 1680 像素。由于 QR 二维码是通过正方形阵列承载信息的，建议二维码图片采用正方形，当二维码图片采用长方形时，会在 QR 二维码信息的周边区域留白。

开发指导

场景介绍

码生成能够根据给定的字符串信息，生成相应的二维码图片。常见应用场景举例：

- 社交或通讯类应用：根据输入的联系人信息，生成联系人二维码。
- 购物或支付类应用：根据输入的支付链接，生成收款或付款二维码。

接口说明

码生成提供了的 IBarcodeDetector()接口，常用方法的功能描述如下：

接口名	方法	功能描述
IBarcodeDetector	int detect(String barcodeInput, byte[] bitmapOutput, int width, int height);	根据给定的信息和二维码图片尺寸，生成二维码图片字节流。
IBarcodeDetector	int release();	停止 QR 码生成服务，释放资源。

开发步骤

1. 在使用码生成 SDK 时，需要先将相关的类添加至工程。

```
import ohos.cvinterface.common.ConnectionCallback;

import ohos.cvinterface.common.VisionManager;

import ohos.cvinterface.qrcode.IBarcodeDetector;
```

定义 ConnectionCallback 回调，实现连接能力引擎成功与否后的操作。

```
ConnectionCallback connectionCallback = new ConnectionCallback() {
    @Override
    public void onServiceConnect() {
        // Do something when service connects successfully
    }

    @Override
    public void onServiceDisconnect() {
        // Do something when service connects unsuccessfully
    }
};
```

调用 VisionManager.init()方法，将此工程的 context 和 connectionCallback 作为入参，建立与能力引擎的连接，context 应为 ohos.aafwk.ability.Ability 或 ohos.aafwk.ability.AbilitySlice 的实例或子类实例。

```
int result = VisionManager.init(context, connectionCallback);
```

实例化 IBarcodeDetector 接口，将此工程的 context 作为入参。

```
IBarcodeDetector barcodeDetector =
VisionManager.getBarcodeDetector(context);
```

定义码生成图像的尺寸，并根据图像大小分配字节流数组空间。

```
final int SAMPLE_LENGTH = 152;
byte[] byteArray = new byte[SAMPLE_LENGTH * SAMPLE_LENGTH * 4];
```

调用 IBarcodeDetector 的 detect()方法，根据输入的字符串信息生成相应的二维码图片字节流。

```
int result = barcodeDetector.detect("This is a TestCase of
IBarcodeDetector", byteArray, SAMPLE_LENGTH, SAMPLE_LENGTH);
```

如果返回值为 0，表明调用成功。

当码生成能力使用完毕后，调用 IBarcodeDetector 的 release()方法，释放资源。

```
result = barcodeDetector.release();
```

调用 VisionManager.destroy()方法，断开与能力引擎的连接。

```
VisionManager.destroy();
```

网络与连接

NFC

概述

NFC（Near Field Communication，近距离无线通信技术) 是一种非接触式识别和互联技术，让移动设备、消费类电子产品、PC 和智能设备之间可以进行近距离无线通信。

HarmonyOS 提供了 NFC 基础控制、Tag 读写、安全单元访问、卡模拟以及 NFC 消息通知的功能。

NFC 基础控制

场景介绍

应用或者其他模块可以通过接口完成以下功能：

1. 查询本机是否支持 NFC 能力。
2. 开启或者关闭本机 NFC。

接口说明

表 1 NFC 开关控制功能的主要接口

类名	接口名	功能描述
NfcController	getInstance(Context context)	获得一个 NFC 控制类的单例。
	openNfc()	打开本机 NFC。
	closeNfc()	关闭本机 NFC。
	isNfcOpen()	查询本机 NFC 是否已打开。
	getNfcState()	获取本机 NFC 的开关状态。
	isNfcAvailable()	查询本机是否支持 NFC 功能。
NfcPermissionException	NfcPermissionException(String errorMessage)	构造一个 NFC 权限异常的实例。

开发步骤

1. 调用 NfcController 类的 getInstance()接口，获取 NfcController 实例，管理本机 NFC 操作。

2. 调用 isNfcOpen()接口，查询 NFC 是否打开。
3. 调用 openNfc()接口打开 NFC；或者调用 closeNfc()接口关闭 NFC。

```
// 查询本机是否支持 NFC
NfcController nfcController = NfcController.getInstance(context);
boolean isAvailable = nfcController.isNfcAvailable();
if (isAvailable) {
    // 调用查询 NFC 是否打开接口，返回值为 NFC 是否是打开的状态
    boolean isOpen = nfcController.isNfcOpen();

    if (!isOpen) {
        // 调用打开 NFC 接口,返回值为函数是否正常执行
        boolean isEnableSuccess = nfcController.openNfc();
    } else {
        // 调用关闭 NFC 接口,返回值为函数是否正常执行
        boolean isDisableSuccess = nfcController.closeNfc();
    }
}
```

Tag 读写

场景介绍

应用或其他模块可以通过接口访问多种协议或技术的 Tag 卡片。

接口说明

表 1 Tag 读写功能的主要接口

类名	接口名	功能描述
TagInfo	getTagId()	获取当前 Tag 的 ID。
	getTagSupportedProfiles()	获取当前 Tag 支持的协议或技术。
	isProfileSupported(int profile)	判断 Tag 是否支持指定的协议或技术。
TagManager	getTagInfo()	获取标签信息。
	connectTag()	建立与 Tag 设备的连接。
	reset()	重置与 Tag 设备的连接，同时会把写 Tag 的超时时间恢复为默认值。
	isTagConnected()	判断与 Tag 设备是否保持连接。
	setSendDataTimeout(int timeout)	设置发送数据到 Tag 的超时时间，单位是 ms。
	getSendDataTimeout()	查询发送数据到 Tag 设备的超

表 1 Tag 读写功能的主要接口

类名	接口名	功能描述
		时时间，单位是 ms。
	sendData(byte[] data)	写数据到 Tag 设备中。
	checkConnected()	检查 Tag 设备是否已连接。
	getMaxSendLength()	获取发送数据的最大长度。在发送数据到 Tag 设备时，用于查询最大可发送数据的长度。
IsoDepTag	IsoDepTag(TagInfo tagInfo)	根据分发 Tag 信息获取 IsoDep 类型 Tag 标签对象。
	getHiLayerResponse()	获取基于 NfcB 技术的 IsoDep 类型 Tag 的高层响应内容。
	getHistoricalBytes()	获取基于 NfcA 技术的 IsoDep 类型 Tag 的历史字节内容。
NfcATag	getInstance(TagInfo tagInfo)	根据分发 Tag 信息获取 NfcA 标签对象。
	getSak()	获取 NfcA 类型 Tag 的 SAK。
	getAtqa()	获取 NfcA 类型 Tag 的 ATQA。
NfcBTag	getInstance(TagInfo tagInfo)	根据分发 Tag 信息获取 NfcB 标签对象。
	getRespAppData()	获取 NfcB 标签对象的应用数据。
	getRespProtocol()	获取 NfcB 标签对象的协议信息。

表 1 Tag 读写功能的主要接口

类名	接口名	功能描述
NdefTag	getNdefMessage()	获取当前连接的 NDEF Tag 设备的 NDEF 信息。
	getNdefMaxSize()	获取最大的 NDEF 信息尺寸。
	getTagType()	获取 NDEF Tag 设备类型。
	readNdefMessage()	从当前连接的 NDEF Tag 设备读取 NDEF 信息。
	writeNdefMessage(NdefMessage msg)	写 NDEF 信息到当前连接的 NDEF Tag 设备。
	canSetReadOnly()	检查 NDEF Tag 设备是否可被设置为只读。
	setReadOnly()	设置 NDEF Tag 设备为只读。
	isNdefWritable()	判断 NDEF Tag 设备是否可写。
MifareClassicTag	getInstance(TagInfo tagInfo)	根据分发 Tag 信息获取 MifareClassic 标签对象。
	getMifareType()	获取 MifareClassic Tag 设备的 Mifare 类型。
	getTagSize()	获取 MifareClassic Tag 设备的尺寸。
	getSectorsNum()	获取 MifareClassic Tag 设备内所有扇区数。
	getBlocksNum()	获取 MifareClassic Tag 设备内所有块数。

表 1 Tag 读写功能的主要接口

类名	接口名	功能描述
	getBlocksNumForSector(int sectorId)	获取 MifareClassic Tag 设备内一个扇区的块数。
	getSectorId(int blockId)	用块 ID 获取 MifareClassic Tag 设备内扇区号。
	getFirstBlockId(int sectorId)	获取 MifareClassic Tag 设备内特定扇区的第一个块 ID。
	authenSectorUseKey(int sectorId, byte[] key, byte keyType)	用密钥鉴权 MifareClassic Tag 设备内特定扇区。
	readBlock(int blockId)	读取 MifareClassic Tag 设备内特定块内容。
	writeBlock(int blockId, byte[] data)	在 MifareClassic Tag 设备内特定块写内容。
	incBlock(int blockId, int value)	在 MifareClassic Tag 设备内特定块的内容加上一个值。
	decBlock(int blockId, int value)	从 MifareClassic Tag 设备内特定块的内容减去一个值。
	restoreBlock(int blockId)	把 MifareClassic Tag 设备内特定块的内容移动到一个内部的缓存区。
MifareUltralightTag	getInstance(TagInfo tagInfo)	根据分发 Tag 信息获取 MifareUltralight 标签对象。
	getMifareType()	获取 MifareUltralight Tag 设备类型。

表 1 Tag 读写功能的主要接口

类名	接口名	功能描述
	readFourPages(int pageOffset)	从 MifareUltralight Tag 设备的特定页数开始读四页。
	writeOnePage(int pageOffset, byte[] data)	在 MifareUltralight Tag 设备的特定页数写数据。

读取卡片类型

1. 从 Intent 中获取 TagInfo，初始化 TagInfo 实例。
2. TagInfo 实例调用 getTagSupportedProfiles()接口查询当前 Tag 支持的技术或协议类型。
3. 调用 isProfileSupported(int profile)接口查询是否支持 NfcA、IsoDep、MifareClassic 等类型。若支持，可使用 TagInfo 实例构造 NfcATag、IsoDep、MifareClassic 等实例。
4. 根据不同的 Tag 技术类型的实例，调用不同的 API 完成 Tag 的访问。

```
// 从 Intent 中获取 TagInfo，初始化 TagInfo 实例
TagInfo tagInfo =
getIntent().getParcelableExtra(NfcController.EXTRA_TAG_INFO);


// 查询 Tag 设备支持的技术或协议，返回值为支持的技术或协议列表
int[] profiles = tagInfo.getTagSupportedProfiles();


// 查询是否支持 NfcA，若支持，构造一个 NfcATag
boolean isSupportedNfcA = tagInfo.isProfileSupported(TagManager.NFC_A);
if (isSupportedNfcA) {
    NfcATag tagNfcA = NfcATag.getInstance(tagInfo);
}


// 查询是否支持 NfcB，若支持，构造一个 NfcBTag
```

```
boolean isSupportedNfcB = tagInfo.isProfileSupported(TagManager.NFC_B);

if (isSupportedNfcB) {

    NfcBTag tagNfcB = NfcBTag.getInstance(tagInfo);

}


// 查询是否支持 IsoDep，若支持，构造一个 IsoDepTag

boolean isSupportedIsoDep =
tagInfo.isProfileSupported(TagManager.ISO_DEP);

if (isSupportedIsoDep) {

    IsoDepTag tagIsoDep = new IsoDepTag(tagInfo);

}


// 查询是否支持 NDEF，若支持，构造一个 NdefTag

boolean isSupportedNdefDep = tagInfo.isProfileSupported(TagManager.NDEF);

if (isSupportedNdefDep) {

    NdefTag tagNdef = new NdefTag(tagInfo);

}


// 查询是否支持 MifareClassic，若支持，构造一个 MifareClassicTag

boolean isSupportedMifareClassic =
tagInfo.isProfileSupported(TagManager.MIFARE_CLASSIC);

if (isSupportedMifareClassic) {

    MifareClassicTag mifareClassicTag =
MifareClassicTag.getInstance(tagInfo);

}


// 查询是否支持 MifareUltralight，若支持，构造一个 MifareUltralightTag

boolean isSupportedMifareUltralight =
tagInfo.isProfileSupported(TagManager.MIFARE_ULTRALIGHT);
```

```
if (isSupportedMifareUltralight) {

    MifareUltralightTag mifareUltralightTag =
MifareUltralightTag.getInstance(tagInfo);

}
```

访问 NfcA 卡片

1. 调用 connectTag()接口连接 Tag 设备。
2. 调用 isTagConnected()接口查询 Tag 设备连接状态。
3. 调用 sendData(byte[] data)接口发送数据到 Tag。

```
// 连接 Tag 设备，返回值为是否连接成功

boolean connSuccess = tagNfcA.connectTag();


// 查询 Tag 连接状态

boolean isConnected= tagNfcA.isTagConnected();


// 发送数据到 Tag，返回值为 Tag 的响应数据

byte[] data = {0x13, 0x59, 0x22};

byte[] response = tagNfcA.sendData(data);
```

访问 NfcB 卡片

1. 调用 connectTag()接口连接 Tag 设备。
2. 调用 isTagConnected()接口查询 Tag 设备连接状态。
3. 调用 sendData(byte[] data)接口发送数据到 Tag。

```
// 连接 Tag 设备，返回值为是否连接成功

boolean connSuccess = tagNfcB.connectTag();
```

```
// 查询 Tag 连接状态
boolean isConnected= tagNfcB.isTagConnected();

// 发送数据到 Tag，返回值为 Tag 的响应数据
byte[] data = {0x13, 0x59, 0x22};
byte[] response = tagNfcB.sendData(data);
```

访问 IsoDep 卡片

1. 调用 getTagInfo()接口获取 TagInfo 对象。
2. 调用 connectTag()接口连接 Tag 设备。
3. 调用 isTagConnected()接口查询 Tag 设备连接状态。
4. 调用 sendData(byte[] data)接口发送数据到 Tag。

```
// 连接 Tag 设备，返回值为是否连接成功
boolean connSuccess = tagIsoDep.connectTag();

// 查询 Tag 连接状态
boolean isConnected= tagIsoDep.isTagConnected();

// 发送数据到 Tag，返回值为 Tag 的响应数据
byte[] data = {0x13, 0x59, 0x22};
byte[] response = tagIsoDep.sendData(data);
```

访问 Ndef 卡片

1. 调用 getTagInfo()接口获取 TagInfo 对象。

2. 调用 connectTag()接口连接 Tag 设备。
3. 调用 isTagConnected()接口查询 Tag 设备连接状态。
4. 调用 sendData(byte[] data)接口发送数据到 Tag。

```
// 连接 Tag 设备，返回值为是否连接成功
boolean connSuccess = tagNdef.connectTag();


// 查询 Tag 连接状态
boolean isConnected = tagNdef.isTagConnected();


// 发送数据到 Tag，返回值为 Tag 的响应数据
byte[] data = {0x13, 0x59, 0x22};
byte[] response = tagNdef.sendData(data);
```

访问 MifareClassic 卡片

1. 调用 getTagInfo()接口获取 TagInfo 对象。
2. 调用 connectTag()接口连接 Tag 设备。
3. 调用 isTagConnected()接口查询 Tag 设备连接状态。
4. 调用 sendData(byte[] data)接口发送数据到 Tag。

```
// 连接 Tag 设备，返回值为是否连接成功
boolean connSuccess = mifareClassicTag.connectTag();


// 查询 Tag 连接状态
boolean isConnected = mifareClassicTag.isTagConnected();


// 发送数据到 Tag，返回值为 Tag 的响应数据
byte[] data = {0x13, 0x59, 0x22};
byte[] response = mifareClassicTag.sendData(data);
```

访问 MifareUltralight 卡片

1. 调用 getTagInfo()接口获取 TagInfo 对象。
2. 调用 connectTag()接口连接 Tag 设备。
3. 调用 isTagConnected()接口查询 Tag 设备连接状态。
4. 调用 sendData(byte[] data)接口发送数据到 Tag。

```
// 连接 Tag 设备，返回值为是否连接成功
boolean connSuccess = mifareUltralightTag.connectTag();


// 查询 Tag 连接状态
boolean isConnected = mifareUltralightTag.isTagConnected();


// 发送数据到 Tag，返回值为 Tag 的响应数据
byte[] data = {0x13, 0x59, 0x22};
byte[] response = mifareUltralightTag.sendData(data);
```

访问安全单元

场景介绍

应用或者其他模块可以通过接口完成以下功能：

1. 获取安全单元（简称为 SE，Secure Element）的个数和名称。
2. 判断安全单元是否在位。
3. 在指定安全单元上打开基础通道。
4. 在指定安全单元上打开逻辑通道。
5. 发送 APDU（Application Protocol Data Unit）数据到安全单元上。

接口说明

表 1 NFC 访问安全单元功能的的主要接口

类名	接口名	功能描述
SEService	SEService()	创建一个安全单元服务的实例。
	isConnected()	查询安全单元服务是否已连接。
	shutdown()	关闭安全单元服务。
	getReaders()	获取全部安全单元。
	getVersion()	获得安全单元服务的版本。
	OnCallback	用于回调的内部类，用于定义回调接口。在服务连接成功后，回调该接口通知应用。
Reader	getName()	获取安全单元的名称。
	isSecureElementPresent()	检查安全单元是否在位。

表 1 NFC 访问安全单元功能的的主要接口

类名	接口名	功能描述
	openSession()	打开当前安全单元上的 session。
	closeSession()	关闭当前安全单元上的所有 session。
Session	openBasicChannel(Aid aid)	打开基础通道。
	openLogicalChannel(Aid aid)	创建逻辑通道。
	getATR()	获得重设安全单元指令的响应。
	closeSessionChannels()	关闭当前 session 的所有通道。
Channel	isClosed()	判断通道是否关闭。
	isBasicChannel()	判断是否是基础通道。
	transmit(byte[] command)	发送指令到安全单元。
	getSelectResponse()	获得应用程序选择指令的响应。
	closeChannel()	关闭通道。
Aid	Aid(byte[] aid, int offset, int length)	构造一个 AID 类的实例。
	isAidValid()	查询 AID 是否有效。
	getAidBytes()	获取 AID 的字节数组形式的值。

开发步骤

1. 调用 SEService 类的构造函数，创建一个安全单元服务的实例，用于访问安全单元。
2. 调用 isConnected()接口，查询安全单元服务的连接状态。
3. 调用 getReaders()接口，获取本机的全部安全单元。
4. 调用 Reader 类的 openSession()接口打开 Session，返回一个打开的 Session 实例。
5. 调用 Session 类的 openBasicChannel(Aid aid)接口打开基础通道，或者调用 openLogicalChannel(Aid aid)接口打开逻辑通道，返回一个打开通道 Channel 实例。
6. 调用 Channel 类的 transmit(byte[] command)，发送 APDU 到安全单元。
7. 调用 Channel 类的 closeChannel()接口关闭通道。
8. 调用 Session 类的 closeSessionChannels()接口关闭 Session 的所有通道。
9. 调用 Reader 类的 closeSessions()接口关闭安全单元的所有 Session。
10. 调用 SEService 类的 shutdown()接口关闭安全单元服务。

```
private class AppServiceConnectedCallback implements SEService.OnCallback
{
    @Override
    public void serviceConnected() {
        // 应用自实现
    }
}
// 创建安全单元服务实例
SEService sEService = new SEService(context, new
AppServiceConnectedCallback());
// 查询安全单元服务的连接状态
boolean isConnected = sEService.isConnected();

// 获取本机的全部安全单元，并获取指定的安全单元 eSE
Reader[] elements = sEService.getReaders();
Reader eSe = null;
for (int i = 0; i < elements.length; i++) {
    if ("eSE".equals(elements[i].getName())) {
        eSe = elements[i];
```

```
        break;
    }
}

// 查询安全单元是否在位
boolean isPresent = eSe.isSecureElementPresent();

// 打开 Session
Optional<Session> optionalSession = eSe.openSession();
Session session = optionalSession.orElse(null);

// 打开通道
if (eSe != null) {
    byte[] aidValue = new byte[]{(byte)0x01, (byte)0x02, (byte)0x03,
(byte)0x04, (byte)0x05};
    // 创建 Aid 实例
    Aid aid = new Aid(aidValue, 0, aidValue.length);
    // 打开基础通道
    Optional<Channel> optionalChannel = session.openBasicChannel(aid);
    Channel basicChannel = optionalChannel.orElse(null);
    // 打开逻辑通道
    optionalChannel = session.openLogicalChannel(aid);
    Channel logicalChannel = optionalChannel.orElse(null);

    // 发送指令给安全单元，返回值为安全单元对指令的响应
    byte[] resp = logicalChannel.transmit(new byte[]{(byte)0x00,
(byte)0xa4, (byte)0x00, (byte)0x00, (byte)0x02, (byte)0x00, (byte)0x00});

    // 关闭通道资源
```

```
    basicChannel.closeChannel()
    logicalChannel.closeChannel();
}

// 关闭 Session 资源
session.close();

// 关闭安全单元资源
eSe.closeSessions();

// 关闭安全单元服务资源 sEService.shutdown();
```

卡模拟功能

场景介绍

应用或者其他模块可以通过接口完成以下功能：

1. 查询是否支持指定安全单元的卡模拟功能，安全单元包括 HCE（Host Card Emulation）、ESE（Embedded Secure Element）和 SIM（Subscriber Identity Module）卡。
2. 开关卡模拟以及查询卡模拟状态，可以打开或关闭指定技术类型的卡模拟。
3. 获取 NFC 信息，信息包括当前激活的安全单元、Hisee 上电状态、是否支持 RSSI 查询等信息。
4. 根据 NFC 服务的类型获取刷卡时选择服务的方式，应用或者其它模块可以查询支付（Payment）类型和非支付（Other）类型业务选择服务的方式。
5. 动态设置和注销前台优先应用。
6. NFC 应用的 AID 相关操作，包括注册和删除应用的 AID、查询应用是否是指定 AID 的默认应用、获取应用的 AID 等。
7. 定义 Host 和 OffHost 服务的抽象类，三方应用通过继承抽象类来实现 NFC 卡模拟功能。

接口说明

表 1 NFC 卡模拟功能的主要接口

类名	接口名	功能描述
CardEmulation	getInstance(NfcController controller)	创建一个卡模拟类的实例。
	isSupported(int feature)	查询是否支持卡模拟功能。
	setListenMode(int mode)	设置卡模拟模式。
	isListenModeEnabled()	查询卡模拟功能是否打开。

表 1 NFC 卡模拟功能的主要接口

类名	接口名	功能描述
	getNfcInfo(String key)	获取 NFC 的信息。
	getSelectionType(String category)	根据 NFC 服务的类型获取刷卡时选择服务的方式。
	registerForegroundPreferred(Ability appAbility, ElementName appName)	动态设置前台优先应用。
	unregisterForegroundPreferred(Ability appAbility)	取消设置前台优先应用。
	isDefaultForAid(ElementName appName, String aid)	判断应用是否是指定 AID 的默认处理应用。
	registerAids(ElementName appName, String type, List<String> aids)	给应用注册指定类型的 AID。
	removeAids(ElementName appName, String type)	删除应用的指定类型的 AID。
	getAids(ElementName appName, String type)	获取应用中指定类型的 AID 列表。
HostService	sendResponse(byte[] response)	发送响应的数据到对端设备。
	handleRemoteCommand(byte[] cmd, IntentParams params)	处理对端设备发送的命令。
	disabledCallback(int errCode)	连接异常的回调。
OffHostService	onConnect(Intent intent)	连接服务并获取远程服务对象。

查询是否支持卡模拟功能

1. 调用 NfcController 类的 getInstance(Context context)接口，获取 NfcController 实例。
2. 调用 CardEmulation 类的 getInstance(NfcController controller)接口，获取 CardEmulation 实例，去管理本机卡模拟模块操作。
3. 调用 isSupported(int feature)接口去查询是否 HCE、UICC、ESE 卡模拟。

```
// 获取 NFC 控制对象

NfcController nfcController = NfcController.getInstance(context);

// 获取卡模拟控制对象

CardEmulation cardEmulation = CardEmulation.getInstance(nfcController);

// 查询是否支持 HCE、UICC、ESE 卡模拟，返回值表示是否支持对应安全单元的卡模拟

boolean isSupportedHce =
cardEmulation.isSupported(CardEmulation.FEATURE_HCE);

boolean isSupportedUicc =
cardEmulation.isSupported(CardEmulation.FEATURE_UICC);

boolean isSupportedEse =
cardEmulation.isSupported(CardEmulation.FEATURE_ESE);
```

开关卡模拟及查询卡模拟状态

1. 调用 NfcController 类的 getInstance(Context context)接口，获取 NfcController 实例。
2. 调用 CardEmulation 类的 getInstance(NfcController controller)接口，获取 CardEmulation 实例，去管理本机卡模拟模块操作。
3. 调用 setListenMode(int mode)接口去打开或者关闭卡模拟。
4. 调用 isListenModeEnabled()接口去查询卡模拟是否打开。

```
// 获取 NFC 控制对象

NfcController nfcController = NfcController.getInstance(context);

// 获取卡模拟控制对象

CardEmulation cardEmulation = CardEmulation.getInstance(nfcController);

// 打开卡模拟
```

```
cardEmulation.setListenMode(CardEmulation.ENABLE_MODE_ALL);

// 调用查询卡模拟开关状态的接口，返回值为卡模拟是否是打开的状态

boolean isEnabled = cardEmulation.isListenModeEnabled(); // true

// 关闭卡模拟

cardEmulation.setListenMode(CardEmulation.DISABLE_MODE_A_B);

// 调用查询卡模拟开关状态的接口，返回值为卡模拟是否是打开的状态

isEnabled = cardEmulation.isListenModeEnabled(); // false
```

获取 NFC 信息

1. 调用 NfcController 类的 getInstance(Context context)接口，获取 NfcController 实例。
2. 调用 CardEmulation 类的 getInstance(NfcController controller)接口，获取 CardEmulation 实例，去管理本机卡模拟模块操作。
3. 调用 getNfcInfo(String key)接口去获取 NFC 信息。

```
// 获取 NFC 控制对象

NfcController nfcController = NfcController.getInstance(context);

// 获取卡模拟控制对象

CardEmulation cardEmulation = CardEmulation.getInstance(nfcController);

// 查询本机当前使能的安全单元类型

String seType =
cardEmulation.getNfcInfo(CardEmulation.KEY_ENABLED_SE_TYPE); //
ENABLED_SE_TYPE_ESE

// 查询 Hisee 上电状态

String hiseeState =
cardEmulation.getNfcInfo(CardEmulation.KEY_HISEE_READY);

// 查询是否支持 rssi 的查询

String rssiAbility =
cardEmulation.getNfcInfo(CardEmulation.KEY_RSSI_SUPPORTED);
```

根据 NFC 服务的类型获取刷卡时选择服务的方式

1. 调用 NfcController 类的 getInstance(Context context)接口，获取 NfcController 实例。
2. 调用 CardEmulation 类的 getInstance(NfcController controller)接口，获取 CardEmulation 实例，去管理本机卡模拟模块操作。
3. 调用 getSelectionType(Sring category)接口去获取选择服务的方式。

```
// 获取 NFC 控制对象

NfcController nfcController = NfcController.getInstance(context);

// 获取卡模拟控制对象

CardEmulation cardEmulation = CardEmulation.getInstance(nfcController);

// 获取选择服务的方式

int result =
cardEmulation.getSelectionType(CardEmulation.CATEGORY_PAYMENT); //
SELECTION_TYPE_PREFER_DEFAULT

result = cardEmulation.getSelectionType(CardEmulation.CATEGORY_OTHER); //
SELECTION_TYPE_ASK_IF_CONFLICT
```

动态设置和注销前台优先应用

1. 调用 NfcController 类的 getInstance(Context context)接口，获取 NfcController 实例。
2. 调用 CardEmulation 类的 getInstance(NfcController controller)接口，获取 CardEmulation 实例，去管理本机卡模拟模块操作。
3. 调用 registerForegroundPreferred(Ability appAbility, ElementName appName)接口去动态设置前台优先应用。
4. 调用 unregisterForegroundPreferred(Ability appAbility)接口去取消设置前台优先应用。

```
// 获取 NFC 控制对象

NfcController nfcController = NfcController.getInstance(context);

// 获取卡模拟控制对象

CardEmulation cardEmulation = CardEmulation.getInstance(nfcController);

// 动态设置前台优先应用
```

```
cardEmulation.registerForegroundPreferred(new Ability(), new
ElementName());

// 注销前台优先应用

cardEmulation.unregisterForegroundPreferred(new Ability());
```

NFC 应用的 AID 相关操作

1. 调用 NfcController 类的 getInstance(Context context)接口，获取 NfcController 实例。
2. 调用 CardEmulation 类的 getInstance(NfcController controller)接口，获取 CardEmulation 实例，去管理本机卡模拟模块操作。
3. 调用 registerAids(ElementName appName, String type, List<String> aids)接口去给应用注册指定类型的 AID。
4. 调用 removeAids(ElementName appName, String type)接口去删除应用的指定类型的 AID。
5. 调用 isDefaultForAid(ElementName appName, String aid)接口去判断应用是否是指定 AID 的默认处理应用。
6. 调用 getAids(ElementName appName, String type)接口去获取应用中指定类型的 AID 列表。

```
// 获取 NFC 控制对象

NfcController nfcController = NfcController.getInstance(context);

// 获取卡模拟控制对象

CardEmulation cardEmulation = CardEmulation.getInstance(nfcController);

// 给应用注册指定类型的 AID

List<String> aids = new ArrayList<String>();

aids.add(0, "A0028321901280");

aids.add(1, "A0028321901281");

try {    cardEmulation.registerAids(new ElementName(),
CardEmulation.CATEGORY_PAYMENT, aids);

} catch (IllegalArgumentException e) {

    HiLog.error(LABEL, "IllegalArgumentException when registerAids");

}
```

```
// 删除应用的指定类型的 AID

cardEmulation.removeAids(new ElementName(),
CardEmulation.CATEGORY_PAYMENT);

cardEmulation.removeAids(new ElementName(),
CardEmulation.CATEGORY_OTHER);

// 判断应用是否是指定 AID 的默认处理应用

String aid = "A0028321901280";

cardEmulation.isDefaultForAid(new ElementName(), aid);

// 获取应用中指定类型的 AID 列表

try {

    cardEmulation.getAids(new ElementName(),
CardEmulation.CATEGORY_PAYMENT);

} catch (NullPointerException e) {

    HiLog.error(LABEL, "NullPointerException when getAids");

} catch (IllegalArgumentException e) {

    HiLog.error(LABEL, "IllegalArgumentException when getAids");

}
```

Host 服务的抽象类

1. 三方应用的服务继承 HostService，实现 HCE 卡模拟功能。
2. 三方应用自定义实现抽象方法 handleRemoteCommand(byte[] cmd, IntentParams params)和 disabledCallback()。
3. 三方应用自定义功能。

```
// 三方 HCE 应用的服务继承 HostService，实现 HCE 卡模拟功能

public class AppService extends HostService {
```

```
    @Override
    public byte[] handleRemoteCommand(byte[] cmd, IntentParams params) {
    // 三方应用自定义接口实现。
    }

    @Override
    public void disabledCallback(int errCode) {
    // 三方应用自定义接口实现。
    }

    // 三方应用自定义功能
}
```

NFC 消息通知

场景介绍

NFC 消息通知（Notification）是 HarmonyOS 内部或者与应用之间跨进程通讯的机制，注册者在注册消息通知后，一旦符合条件的消息被发出，注册者即可接收到该消息。

接口说明

表 1 NFC 消息通知的相关广播介绍

描述	通知名	附加参数
NFC 状态	usual.event.nfc.action.ADAPTER_STATE_CHANGED	extra_nfc_state
进场消息	usual.event.nfc.action.RF_FIELD_ON_DETECTED	extra_nfc_transaction
离场消息	usual.event.nfc.action.RF_FIELD_OFF_DETECTED	-

注册并获取 NFC 状态改变消息

1. 构建消息通知接收者 NfcStateEventSubscriber。

2. 注册 NFC 状态改变消息。
3. NfcStateEventSubscriber 接收并处理 NFC 状态改变消息。

```
// 构建消息接收者/注册者
class NfcStateEventSubscriber extends CommonEventSubscriber {
    NfcStateEventSubscriber (CommonEventSubscribeInfo info) {
        super(info);
    }

    @Override
    public void onReceiveEvent(CommonEventData commonEventData) {
        if (commonEventData == null || commonEventData.getIntent() == null)
{
            return;
        }
        if
(NfcController.STATE_CHANGED.equals(commonEventData.getIntent().getActi
on())) {
            IntentParams params = commonEventData.getIntent().getParams();
            if (params != null) {
                int currState =
commonEventData.getIntent().getIntParam(NfcController.EXTRA_NFC_STATE,
NfcController.STATE_OFF);
            }
        }
    }
}

// 注册消息
MatchingSkills matchingSkills = new MatchingSkills();
// 增加获取 NFC 状态改变消息
```

```
matchingSkills.addEvent(NfcController.STATE_CHANGED);

matchingSkills.addEvent(CommonEventSupport.COMMON_EVENT_NFC_ACTION_ADAP
TER_STATE_CHANGED);

CommonEventSubscribeInfo subscribeInfo = new
CommonEventSubscribeInfo(matchingSkills);

NfcStateEventSubscriber subscriber = new
NfcStateEventSubscriber(subscribeInfo);

try {

    CommonEventManager.subscribeCommonEvent(subscriber);

} catch (RemoteException e) {

    HiLog.e(TAG, "doSubscribe occur exception:" + e.toString());

}
```

注册并获取 NFC 场强消息

1. 构建消息通知接收者 NfcFieldOnAndOffEventSubscriber。
2. 注册 NFC 场强消息。
3. NfcFieldOnAndOffEventSubscriber 接收并处理 NFC 场强消息。

```
// 构建消息接收者/注册者

class NfcFieldOnAndOffEventSubscriber extends CommonEventSubscriber {

    NfcFieldOnAndOffEventSubscriber (CommonEventSubscribeInfo info) {

        super(info);

    }


    @Override

    public void onReceiveEvent(CommonEventData commonEventData) {

        if (commonEventData == null || commonEventData.getIntent() == null)
{

            return;
```

```
        }
        if
(NfcController.FIELD_ON_DETECTED.equals(commonEventData.getIntent().get
Action())) {
            IntentParams params = commonEventData.getIntent().getParams();
            if (params == null) {
                HiLog.i(TAG, "Pure FIELD_ON_DETECTED");
            } else {
                HiLog.i(TAG, "Transaction FIELD_ON_DETECTED");
                Intent transactionIntent = (Intent)
params.getParam("transactionIntent");
            }
        } else if
(NfcController.FIELD_OFF_DETECTED.equals(commonEventData.getIntent().ge
tAction())) {
            HiLog.i(TAG, "FIELD_OFF_DETECTED");
        }
        HiLog.i(TAG, "MyFieldOnAndOffEventSubscriber onReceiveEvent ....:"
+ commonEventData.getIntent().getAction());
    }
}


// 注册消息
MatchingSkills matchingSkills = new MatchingSkills();
// 增加获取 NFC 状态改变消息
matchingSkills.addEvent(NfcController.FIELD_ON_DETECTED);
matchingSkills.addEvent(NfcController.FIELD_OFF_DETECTED);
CommonEventSubscribeInfo subscribeInfo = new
CommonEventSubscribeInfo(DomainMode.BOTH, matchingSkills);
HiLog.i(TAG, "subscribeInfo permission: " +
subscribeInfo.getPermission());
```

```
MyFieldOnAndOffEventSubscriber subscriber = new
MyFieldOnAndOffEventSubscriber(subscribeInfo);

try {

    CommonEventManager.subscribeCommonEvent(subscriber);

} catch (RemoteException e) {

    HiLog.e(TAG, "doSubscribe occur exception:" + e.toString());

}
```

蓝牙

概述

蓝牙是短距离无线通信的一种方式，支持蓝牙的两个设备必须配对后才能通信。HarmonyOS 蓝牙主要分为传统蓝牙和低功耗蓝牙（通常称为 BLE，Bluetooth Low Energy）。传统蓝牙指的是蓝牙版本 3.0 以下的蓝牙，低功耗蓝牙指的是蓝牙版本 4.0 以上的蓝牙。

当前蓝牙的配对方式有两种：蓝牙协议 2.0 以下支持 PIN 码（Personal Identification Number，个人识别码）配对，蓝牙协议 2.1 以上支持简单配对。

传统蓝牙

HarmonyOS 传统蓝牙提供的功能有：

- 传统蓝牙本机管理：打开和关闭蓝牙、设置和获取本机蓝牙名称、扫描和取消扫描周边蓝牙设备、获取本机蓝牙 profile 对其他设备的连接状态、获取本机蓝牙已配对的蓝牙设备列表。
- 传统蓝牙远端设备操作：查询远端蓝牙设备名称和 MAC 地址、设备类型和配对状态，以及向远端蓝牙设备发起配对。

BLE

BLE 设备交互时会分为不同的角色：

- 中心设备和外围设备：中心设备负责扫描外围设备、发现广播。外围设备负责发送广播。
- GATT（Generic Attribute Profile，通用属性配置文件）服务端与 GATT 客户端：两台设备建立连接后，其中一台作为 GATT 服务端，另一台作为 GATT 客户端。

HarmonyOS 低功耗蓝牙提供的功能有：

- BLE 扫描和广播：根据指定状态获取外围设备、启动或停止 BLE 扫描、广播。

约束与限制

调用蓝牙的打开接口需要 ohos.permission.USE_BLUETOOTH 权限，调用蓝牙扫描接口需要 ohos.permission.LOCATION 权限和 ohos.permission.DISCOVER_BLUETOOTH 权限。

传统蓝牙本机管理

场景介绍

传统蓝牙本机管理主要是针对蓝牙本机的基本操作，包括打开和关闭蓝牙、设置和获取本机蓝牙名称、扫描和取消扫描周边蓝牙设备、获取本机蓝牙 profile 对其他设备的连接状态、获取本机蓝牙已配对的蓝牙设备列表。

接口说明

表 1 蓝牙本机管理类 BluetoothHost 的主要接口

接口名	功能描述
getDefaultHost(Context context)	获取 BluetoothHost 实例，去管理本机蓝牙操作。
enableBt()	打开本机蓝牙。
disableBt()	关闭本机蓝牙。
setLocalName(String name)	设置本机蓝牙名称。
getLocalName()	获取本机蓝牙名称。
getBtState()	获取本机蓝牙状态。
startBtDiscovery()	发起蓝牙设备扫描。
cancelBtDiscovery()	取消蓝牙设备扫描。
isBtDiscovering()	检查蓝牙是否在扫描设备中。
getProfileConnState(int profile)	获取本机蓝牙 profile 对其他设备的连接状

表 1 蓝牙本机管理类 BluetoothHost 的主要接口

接口名	功能描述
	态。
getPairedDevices()	获取本机蓝牙已配对的蓝牙设备列。

打开蓝牙

1. 调用 BluetoothHost 的 getDefaultHost(Context context)接口，获取 BluetoothHost 实例，管理本机蓝牙操作。
2. 调用 enableBt()接口，打开蓝牙。
3. 调用 getBtState()，查询蓝牙是否打开。

```
// 获取蓝牙本机管理对象
BluetoothHost bluetoothHost = BluetoothHost.getDefaultHost(context);
// 调用打开接口
bluetoothHost.enableBt();
// 调用获取蓝牙开关状态接口
int state = bluetoothHost.getBtState();
```

蓝牙扫描

1. 开始蓝牙扫描前要先注册广播 BluetoothRemoteDevice.EVENT_DEVICE_DISCOVERED。
2. 调用 startBtDiscovery()接口开始进行扫描外围设备。
3. 如果想要获取扫描到的设备，必须在注册广播时继承实现 CommonEventSubscriber 类的 onReceiveEvent(CommonEventData data)方法，并接收 EVENT_DEVICE_DISCOVERED 广播。

```
//开始扫描
mBluetoothHost.startBtDiscovery();
```

```
//接收系统广播
public class MyCommonEventSubscriber extends CommonEventSubscriber {
    @Override
    public void onReceiveEvent(CommonEventData var){
        Intent info = var.getIntent();
        if(info == null) return;
        //获取系统广播的 action
        String action = info.getAction();
        //判断是否为扫描到设备的广播
        if(action == BluetoothRemoteDevice.EVENT_DEVICE_DISCOVERED){
            IntentParams myParam = info.getParams();
            BluetoothRemoteDevice device =
(BluetoothRemoteDevice)myParam.getParam(BluetoothRemoteDevice.REMOTE_
DEVICE_PARAM_DEVICE);
        }
    }
}
```

传统蓝牙远端设备操作

场景介绍

传统蓝牙远端管理操作主要是针对远端蓝牙设备的基本操作，包括获取远端蓝牙设备地址、类型、名称和配对状态，以及向远端设备发起配对。

接口说明

表 1 蓝牙远端设备管理类 BluetoothRemoteDevice 的主要接口

接口名	功能描述
getDeviceAddr()	获取远端蓝牙设备地址。
getDeviceClass()	获取远端蓝牙设备类型。
getDeviceName()	获取远端蓝牙设备名称。
getPairState()	获取远端设备配对状态。
startPair()	向远端设备发起配对。

开发步骤

1. 调用 BluetoothHost 的 getDefaultHost(Context context)接口，获取 BluetoothHost 实例，管理本机蓝牙操作。
2. 调用 enableBt()接口，打开蓝牙。
3. 调用 startBtDiscovery()，扫描设备。
4. 调用 startPair()，发起配对。

```
// 获取蓝牙本机管理对象
BluetoothHost bluetoothHost = BluetoothHost.getDefaultHost(context);
```

```
// 调用打开接口
bluetoothHost.enableBt();
// 调用扫描接口
bluetoothHost.startBtDiscovery();
//设置界面会显示出扫描结果列表，点击蓝牙设备去配对
BluetoothRemoteDevice device =
bluetoothHost.getRemoteDev(TEST_ADDRESS);
device.startPair();
```

BLE 扫描和广播

场景介绍

通过 BLE 扫描和广播提供的开放能力，可以根据指定状态获取外围设备、启动或停止 BLE 扫描、广播。

接口说明

表 1 BLE 中心设备管理类 BleCentralManager 的主要接口

接口名	功能描述
startScan(List<BleScanFilter> filters)	进行 BLE 蓝牙扫描，并使用 filters 对结果进行过滤。
stopScan()	停止 BLE 蓝牙扫描。
getDevicesByStates(int[] states)	根据状态获取连接的外围设备。
BleCentralManager(BleCentralManagerCallback callback)	获取中心设备管理对象。

表 2 中心设备管理回调类 BleCentralManagerCallback 的主要接口

接口名	功能描述
onScanCallback(BleScanResult result)	扫描到 BLE 设备的结果回调。
onStartScanFailed(int resultCode)	启动扫描失败的回调。

表 3 BLE 广播相关的 BleAdvertiser 类和 BleAdvertiseCallback 类的主要接口

接口名	功能描述

表 3 BLE 广播相关的 BleAdvertiser 类和 BleAdvertiseCallback 类的主要接口

接口名	功能描述
BleAdvertiser(Context context, BleAdvertiseCallback callback)	用于获取广播操作对象。
startAdvertising(BleAdvertiseSettings settings, BleAdvertiseData advData, BleAdvertiseData scanResponse)	进行 BLE 广播，第一个参数为广播参数，第二个为广播数据，第三个参数是扫描和广播数据参数的响应。
stopAdvertising()	停止 BLE 广播。
startResultEvent(int result)	广播回调结果。

中心设备进行 BLE 扫描

1. 进行 BLE 扫描之前先要继承 BleCentralManagerCallback 类实现 onScanCallback 和 onStartScanFailed 回调函数，用于接收扫描结果。
2. 调用 BleCentralManager(BleCentralManagerCallback callback)接口获取中设备管理对象。
3. 获取扫描过滤器，过滤器为空时为不使用过滤器扫描，然后调用 startScan()开始扫描 BLE 设备，在回调中获取扫描到的 BLE 设备。

```
// 实现扫描回调
public class ScanCallback implements BleCentralManagerCallback{
    List<BleScanResult>results = new ArrayList<BleScanResult>();
    @Override
    public void onScanCallback(BleScanResult var1) {
        // 对扫描结果进行处理
        results.add(var1);
    }
    @Override
    public void onStartScanFailed(int var1) {
```

```
        HiLog.info(TAG,"Start Scan failed,Code:" + var1);
    }
}
// 获取中心设备管理对象
private ScanCallback centralManagerCallback = new ScanCallback();
private BleCentralManager centralManager = new
BleCentralManager(centralManagerCallback);
// 创建扫描过滤器然后开始扫描
List<BleScanFilter> filters = new ArrayList<BleScanFilter>();
centralManager.startScan(filters);
```

外围设备进行 BLE 广播

1. 进行 BLE 广播前需要先继承 advertiseCallback 类实现 startResultEvent 回调，用于获取广播结果。
2. 调用接口 BleAdvertiser(Context context, BleAdvertiseCallback callback)获取广播对象，构造广播参数和广播数据。
3. 调用 startAdvertising(BleAdvertiseSettings settings, BleAdvertiseData advData, BleAdvertiseData scanResponse)接口开始 BLE 广播。

```
// 实现 BLE 广播回调
private BleAdvertiseCallback advertiseCallback = new
BleAdvertiseCallback() {
    @Override
    public void startResultEvent(int result) {
        if(result == BleAdvertiseCallback.RESULT_SUCC){
            // 开始 BLE 广播成功
        }
        else {
            // 开始 BLE 广播失败
        }
```

```
    }
};
// 获取 BLE 广播对象
private BleAdvertiser advertiser = new
BleAdvertiser(this,advertiseCallback);
// 创建 BLE 广播参数和数据
private BleAdvertiseData data = new BleAdvertiseData.Builder()
                        .addServiceUuid(SequenceUuid.uuidFromString(S
erver_UUID))      // 添加服务的 UUID
                        .addServiceData(SequenceUuid.uuidFromString(S
erver_UUID),new byte[]{0x11})    // 添加广播数据内容
                        .build();
private BleAdvertiseSettings advertiseSettings = new
BleAdvertiseSettings.Builder()
                       .setConnectable(true)                  // 设置
是否可连接广播
                       .setInterval(BleAdvertiseSettings.INTERVAL_SLO
T_DEFAULT)     // 设置广播间隔
                       .setTxPower(BleAdvertiseSettings.TX_POWER_DEFA
ULT)        // 设置广播功率
                       .build();
// 开始广播
advertiser.startAdvertising(advertiseSettings,data,null);
```

WLAN

概述

无线局域网（Wireless Local Area Networks，WLAN），是通过无线电、红外光信号或者其他技术发送和接收数据的局域网，用户可以通过 WLAN 实现结点之间无物理连接的网络通讯。常用于用户携带可移动终端的办公、公众环境中。

HarmonyOS WLAN 服务系统为用户提供 WLAN 基础功能、P2P（peer-to-peer）功能和 WLAN 消息通知的相应服务，让应用可以通过 WLAN 和其他设备互联互通。

约束与限制

本开发指南提供多个开发场景的指导，涉及多个 API 接口的调用。在调用 API 前，应用需要先申请对应的访问权限，具体请参照对应场景的开放能力介绍。

WLAN 基础功能

场景介绍

应用或者其他模块可以通过接口完成以下功能：

1. 获取 WLAN 状态，查询 WLAN 是否打开。
2. 发起扫描并获取扫描结果。
3. 获取连接态详细信息，包括连接信息、IP 信息等。
4. 获取设备国家码。
5. 获取设备是否支持指定的能力。

接口说明

WifiDevice 提供 WLAN 的基本功能，其接口说明如下。

表 1 WifiDevice 的主要接口

接口名	描述	所需权限
getInstance(Context context)	获取 WLAN 功能管理对象实例，通过该实例调用 WLAN 基本功能 API。	NA
isWifiActive()	获取当前 WLAN 打开状态。	ohos.permission.GET_WIFI_INFO
scan()	发起 WLAN 扫描。	ohos.permission.SET_WIFI_INFO ohos.permission.LOCATION
getScanInfoList()	获取上次扫描结果。	ohos.permission.GET_WIFI_INFO ohos.permission.LOCATION
isConnected()	获取当前 WLAN 连接状态。	ohos.permission.GET_WIFI_INFO
getLinkedInfo()	获取当前的 WLAN 连接信	ohos.permission.GET_WIFI_INFO

表 1 WifiDevice 的主要接口

接口名	描述	所需权限
	息。	
getIpInfo()	获取当前连接的 WLAN IP 信息。	ohos.permission.GET_WIFI_INFO
getSignalLevel(int rssi, int band)	通过 RSSI 与频段计算信号格数。	NA
getCountryCode()	获取设备的国家码。	ohos.permission.LOCATION ohos.permission.GET_WIFI_INFO
isFeatureSupported(long featureId)	获取设备是否支持指定的特性。	ohos.permission.GET_WIFI_INFO

获取 WLAN 状态

1. 调用 WifiDevice 的 getInstance(Context context)接口，获取 WifiDevice 实例，用于管理本机 WLAN 操作。
2. 调用 isWifiActive()接口查询 WLAN 是否打开。

```
// 获取 WLAN 管理对象
WifiDevice wifiDevice = WifiDevice.getInstance(context);
// 调用获取 WLAN 开关状态接口
boolean isWifiActive  = wifiDevice.isWifiActive(); // 若 WLAN 打开，则返回 true，否则返回 false
```

发起扫描并获取结果

1. 调用 WifiDevice 的 getInstance(Context context)接口，获取 WifiDevice 实例，用于管

理本机 WLAN 操作。
2. 调用 scan()接口发起扫描。
3. 调用 getScanInfoList()接口获取扫描结果。

```
// 获取 WLAN 管理对象
WifiDevice wifiDevice = WifiDevice.getInstance(context);
// 调用 WLAN 扫描接口
boolean isScanSuccess = wifiDevice.scan(); // true
// 调用获取扫描结果
 List<WifiScanInfo> scanInfos = wifiDevice.getScanInfoList();
```

获取连接态详细信息

1. 调用 WifiDevice 的 getInstance(Context context)接口，获取 WifiDevice 实例，用于管理本机 WLAN 操作。
2. 调用 isConnected()接口获取当前连接状态。
3. 调用 getLinkedInfo()接口获取连接信息。
4. 调用 getIpInfo()接口获取 IP 信息。

```
// 获取 WLAN 管理对象
WifiDevice wifiDevice = WifiDevice.getInstance(context);
// 调用 WLAN 连接状态接口,确定当前设备是否连接 WLAN
boolean isConnected = wifiDevice.isConnected();
if (isConnected) {
    // 获取 WLAN 连接信息
    Optional<WifiLinkedInfo> linkedInfo = wifiDevice.getLinkedInfo();
    // 获取连接信息中的 SSID
    String ssid = linkedInfo.get().getSsid();
    // 获取 WLAN 的 IP 信息
    Optional<IpInfo> ipInfo = wifiDevice.getIpInfo();
    // 获取 IP 信息中的 IP 地址与网关
```

```
    int ipAddress = ipInfo.get().getIpAddress();
    int gateway = ipInfo.get().getGateway();
}
```

获取设备国家码

1. 调用 WifiDevice 的 getInstance(Context context)接口，获取 WifiDevice 实例，用于管理本机 WLAN 操作。
2. 调用 getCountryCode()接口获取设备的国家码。

```
// 获取 WLAN 管理对象
WifiDevice wifiDevice = WifiDevice.getInstance(context);
// 获取当前设备的国家码
String countryCode = wifiDevice.getCountryCode();
```

判断设备是否支持指定的能力

1. 调用 WifiDevice 的 getInstance(Context context)接口，获取 WifiDevice 实例，用于管理本机 WLAN 操作。
2. 调用 isFeatureSupported(long featureId)接口判断设备是否支持指定的能力。

```
// 获取 WLAN 管理对象
WifiDevice wifiDevice = WifiDevice.getInstance(context);
// 获取当前设备是否支持指定的能力
boolean isSupport =
wifiDevice.isFeatureSupported(WifiUtils.WIFI_FEATURE_INFRA);
isSupport =
wifiDevice.isFeatureSupported(WifiUtils.WIFI_FEATURE_INFRA_5G);
isSupport =
wifiDevice.isFeatureSupported(WifiUtils.WIFI_FEATURE_PASSPOINT);
isSupport = wifiDevice.isFeatureSupported(WifiUtils.WIFI_FEATURE_P2P);
```

```
isSupport =
wifiDevice.isFeatureSupported(WifiUtils.WIFI_FEATURE_MOBILE_HOTSPOT);

isSupport =
wifiDevice.isFeatureSupported(WifiUtils.WIFI_FEATURE_AWARE);

isSupport =
wifiDevice.isFeatureSupported(WifiUtils.WIFI_FEATURE_AP_STA);

isSupport =
wifiDevice.isFeatureSupported(WifiUtils.WIFI_FEATURE_WPA3_SAE);

isSupport =
wifiDevice.isFeatureSupported(WifiUtils.WIFI_FEATURE_WPA3_SUITE_B);

isSupport = wifiDevice.isFeatureSupported(WifiUtils.WIFI_FEATURE_OWE);
```

P2P 功能

场景介绍

WLAN P2P 功能用于设备与设备之间的点对点数据传输，应用可以通过接口完成以下功能：

1. 发现对端设备。
2. 建立与移除群组。
3. 向对端设备发起连接。
4. 获取 P2P 相关信息。

接口说明

WifiP2pController 提供 WLAN P2P 功能，接口说明如下。

表 1 WifiP2pController 的主要接口

接口名	描述	所需权限
init(EventRunner eventRunner, WifiP2pCallback callback)	初始化 P2P 的信使，当且仅当信使被成功初始化，P2P 的其他功能才可以正常使用。	ohos.permission.GET_WIFI_INFO ohos.permission.SET_WIFI_INFO
discoverDevices(WifiP2pCallback callback)	搜索附近可用的 P2P 设备。	ohos.permission.GET_WIFI_INFO
stopDeviceDiscovery(WifiP2pCallback callback)	停止搜索附近的 P2P 设备。	ohos.permission.GET_WIFI_INFO
createGroup(WifiP2pConfig wifiP2pConfig, WifiP2pCallback callback)	建立 P2P 群组。	ohos.permission.GET_WIFI_INFO

表 1 WifiP2pController 的主要接口

接口名	描述	所需权限
removeGroup(WifiP2pCallback callback)	移除 P2P 群组。	ohos.permission.GET_WIFI_INFO
requestP2pInfo(int requestType, WifiP2pCallback callback)	请求 P2P 相关信息，如群组信息、连接信息、设备信息等。	ohos.permission.GET_WIFI_INFO
connect(WifiP2pConfig wifiP2pConfig, WifiP2pCallback callback)	向指定设备发起连接。	ohos.permission.GET_WIFI_INFO
cancelConnect(WifiP2pCallback callback)	取消向指定设备发起的连接。	ohos.permission.GET_WIFI_INFO

启动与停止 P2P 搜索的开发步骤

1. 调用 WifiP2pController 的 getInstance(Context context)接口，获取 P2P 控制器实例，用于管理 P2P 操作。
2. 调用 init(EventRunner eventRunner, WifiP2pCallback callback)初始化 P2P 控制器实例。
3. 发起 P2P 搜索。
4. 获取 P2P 搜索回调信息。
5. 停止 P2P 搜索。

```
try {
    // 获取 P2P 管理对象
    WifiP2pController wifiP2pController =
WifiP2pController.getInstance(this);
    // 初始化 P2P 管理对象，用于建立 P2P 信使等行为
    wifiP2pController.init(EventRunner.create(true), null);
```

```
    // 创建 P2P 回调对象
    P2pDiscoverCallBack p2pDiscoverCallBack = new
P2pDiscoverCallBack();
    // 发起 P2P 搜索
    wifiP2pController.discoverDevices(p2pDiscoverCallBack);
    // 停止 P2P 搜索
    wifiP2pController.stopDeviceDiscovery(p2pDiscoverCallBack);
} catch (RemoteException re) {
    HiLog.warn(LABEL, "exception happened.");
}

// 获取 P2P 启动与停止搜索的回调信息（失败或者成功）
private class P2pDiscoverCallBack extends WifiP2pCallback {
    @Override
    public void eventExecFail(int reason) {
        HiLog.info(LABEL, "discoverDevices eventExecFail
reason : %{public}d", reason);
    }

    @Override
    public void eventExecOk() {
        HiLog.info(LABEL, "discoverDevices eventExecOk");
    }
}
```

创建与移除群组的开发步骤

1. 调用 WifiP2pController 的 getInstance(Context context)接口，获取 P2P 控制器实例，用于管理 P2P 操作。
2. 调用 init(EventRunner eventRunner, WifiP2pCallback callback)初始化 P2P 控制器实例。
3. 创建 P2P 群组。
4. 移除 P2P 群组。

```
try {
    // 获取 P2P 管理对象
    WifiP2pController wifiP2pController =
WifiP2pController.getInstance(this);
    // 初始化 P2P 管理对象，用于建立 P2P 信使等行为
    wifiP2pController.init(EventRunner.create(true), null);
    // 创建用于 P2P 建组需要的配置
    WifiP2pConfig wifiP2pConfig = new
WifiP2pConfig("DEFAULT_GROUP_NAME", "DEFAULT_PASSPHRASE");
    wifiP2pConfig.setDeviceAddress("02:02:02:02:03:04");
    wifiP2pConfig.setGroupOwnerBand(0);
    // 创建 P2P 回调对象
    P2pCreateGroupCallBack p2pCreateGroupCallBack = new
P2pCreateGroupCallBack();
    // 创建 P2P 群组
    wifiP2pController.createGroup(wifiP2pConfig,
p2pCreateGroupCallBack);
    // 移除 P2P 群组
    wifiP2pController.removeGroup(p2pCreateGroupCallBack);
} catch (RemoteException re) {
    HiLog.warn(LABEL, "exception happened.");
}

private class P2pCreateGroupCallBack extends WifiP2pCallback {
```

```
    @Override
    public void eventExecFail(int reason) {
        HiLog.info(LABEL, "CreateGroup eventExecFail
reason : %{public}d", reason);
    }

    @Override
    public void eventExecOk() {
        HiLog.info(LABEL, "CreateGroup eventExecOk");
    }
}
```

发起 P2P 连接的开发步骤

1. 调用 WifiP2pController 的 getInstance(Context context)接口，获取 P2P 控制器实例，用于管理 P2P 操作。
2. 调用 init(EventRunner eventRunner, WifiP2pCallback callback)初始化 P2P 控制器实例。
3. 调用 requestP2pInfo()查询 P2P 可用设备信息。
4. 根据场景不同，从可用设备信息中选择目标设备。
5. 调用 connect 接口发起连接。

```
try {
    // 获取 P2P 管理对象
    WifiP2pController wifiP2pController =
WifiP2pController.getInstance(this);
    // 初始化 P2P 管理对象，用于建立 P2P 信使等行为
    wifiP2pController.init(EventRunner.create(true), null);
    // 查询可用 P2P 设备信息，通过回调获取 P2P 设备信息
    P2pRequestPeersCallBack p2pRequestPeersCallBack = new
P2pRequestPeersCallBack();
```

```
wifiP2pController.requestP2pInfo(WifiP2pController.DEVICE_LIST_REQUEST, p2pRequestPeersCallBack);
} catch (RemoteException re) {
    HiLog.warn(LABEL, "exception happened.");
}


private class P2pRequestPeersCallBack extends WifiP2pCallback {
    @Override
    public void eventP2pDevicesList(List<WifiP2pDevice> devices) {
        HiLog.info(LABEL, "eventP2pDevicesList when start connect group");
        // 根据场景不同，选择不同的设备进行连接，这里，通过 MAC 地址搜索到指定设备
        WifiP2pConfig wifiP2pConfig = getSameP2pConfigFromDevices(devices);
        try {
            if (wifiP2pConfig != null) {
                // 向指定的设备发起连接
                wifiP2pController.connect(wifiP2pConfig, null);
                }
        } catch (RemoteException re) {
            HiLog.warn(LABEL, "exception happened in connect.");
        }
    }
}


private WifiP2pConfig getSameP2pConfigFromDevices(List<WifiP2pDevice> devices) {
    if (devices == null) {
```

```
        return null;
    }
    for (int i = 0; i < devices.size(); i++) {
        WifiP2pDevice p2pDevice = devices.get(i);
        HiLog.info(LABEL,
"p2pDevice.getDeviceAddress() : %{private}s",
p2pDevice.getDeviceAddress());
        if (p2pDevice.getDeviceAddress() != null
                &&
p2pDevice.getDeviceAddress().equals(TARGET_P2P_MAC_ADDRESS)) {
            HiLog.info(LABEL, "received same mac address");
            WifiP2pConfig wifiP2pConfig = new
WifiP2pConfig("DEFAULT_GROUP_NAME", "DEFAULT_PASSPHRASE");

wifiP2pConfig.setDeviceAddress(p2pDevice.getDeviceAddress());
            return wifiP2pConfig;
        }
    }
    return null;
}
```

请求 P2P 相关信息的开发步骤

1. 调用 WifiP2pController 的 getInstance()接口，获取 P2P 控制器实例，用于管理 P2P 操作。
2. 调用 init()初始化 P2P 控制器实例。
3. 调用 requestP2pInfo()查询 P2P 群组信息。
4. 调用 requestP2pInfo()查询 P2P 设备信息。
5. 根据场景不同，可以调用 requestP2pInfo 获取需要的信息。

```
try {
```

```
    // 获取 P2P 管理对象
    WifiP2pController wifiP2pController =
WifiP2pController.getInstance(this);
    // 初始化 P2P 管理对象，用于建立 P2P 信使等行为
    wifiP2pController.init(EventRunner.create(true), null);
    // 查询可用 P2P 群组信息，通过回调获取 P2P 群组信息
    P2pRequestGroupInfoCallBack p2pRequestGroupInfoCallBack = new
P2pRequestGroupInfoCallBack();

wifiP2pController.requestP2pInfo(WifiP2pController.GROUP_INFO_REQUEST
, p2pRequestGroupInfoCallBack);
    // 查询可用 P2P 设备信息，通过回调获取 P2P 设备信息
    P2pRequestDeviceInfoCallBack p2pRequestDeviceInfoCallBack = new
P2pRequestDeviceInfoCallBack();

wifiP2pController.requestP2pInfo(WifiP2pController.DEVICE_INFO_REQUES
T, p2pRequestDeviceInfoCallBack);
    // 通过调用 requestP2pInfo 接口，可以查询以下关键信息

wifiP2pController.requestP2pInfo(WifiP2pController.NETWORK_INFO_REQUE
ST, callback); // 网络信息

wifiP2pController.requestP2pInfo(WifiP2pController.DEVICE_LIST_REQUES
T, callback); // 设备列表信息
} catch (RemoteException re) {
    HiLog.warn(LABEL, "exception happened.");
}

// 群组信息回调
private class P2pRequestGroupInfoCallBack extends WifiP2pCallback {
    @Override
    public void eventP2pGroup(WifiP2pGroup group) {
```

```
        HiLog.info(LABEL, "P2pRequestGroupInfoCallBack
eventP2pGroup");
        doSthFor(group);
    }
}
// 设备信息回调
private class P2pRequestDeviceInfoCallBack extends WifiP2pCallback {
    @Override
    public void eventP2pGroup(WifiP2pDevice p2pDevice) {
        HiLog.info(LABEL, "eventP2pGroup");
            doSthFor(p2pDevice);
    }
}
```

WLAN 消息通知

场景介绍

WLAN 消息通知（Notification）是 HarmonyOS 内部或者与应用之间跨进程通讯的机制，注册者在注册消息通知后，一旦符合条件的消息被发出，注册者即可接收到该消息并获取消息中附带的信息。

接口说明

表 1 WLAN 消息通知的相关广播介绍

描述	通知名	附加参数
WLAN 状态	usual.event.wifi.POWER_STATE	active_state
WLAN 扫描	usual.event.wifi.SCAN_FINISHED	scan_state
WLAN RSSI 变化	usual.event.wifi.RSSI_VALUE	rssi_value
WLAN 连接状态	usual.event.wifi.CONN_STATE	conn_state
Hotspot 状态	usual.event.wifi.HOTSPOT_STATE	hotspot_active_state
Hotspot 连接状态	usual.event.wifi.WIFI_HS_STA_JOIN usual.event.wifi.WIFI_HS_STA_LEAVE	-
P2P 状态	usual.event.wifi.p2p.STATE_CHANGE	p2p_state

表 1 WLAN 消息通知的相关广播介绍

描述	通知名	附加参数
P2P 连接状态	usual.event.wifi.p2p.CONN_STATE_CHANGE	linked_info net_info group_info
P2P 设备列表变化	usual.event.wifi.p2p.PEERS_STATE_CHANGE	peers_list
P2P 搜索状态变化	usual.event.wifi.p2p.PEER_DISCOVERY_STATE_CHANGE	peers_discovery
P2P 当前设备变化	usual.event.wifi.p2p.CURRENT_DEVICE_CHANGE	p2p_device
P2P 群组信息变化	usual.event.wifi.p2p.GROUP_STATE_CHANGED	-

开发步骤

1. 构建消息通知接收者 WifiEventSubscriber。
2. 注册 WLAN 变化消息。
3. WifiEventSubscriber 接收并处理 WLAN 广播消息。

```
// 构建消息接收者/注册者
class WifiEventSubscriber extends CommonEventSubscriber {
    WifiEventSubscriber (CommonEventSubscribeInfo info) {
        super(info);
    }
```

```
    @Override
    public void onReceiveEvent(CommonEventData commonEventData) {
        if
(WifiEvents.EVENT_ACTIVE_STATE.equals(commonEventData.getIntent().get
Action())) {
            // 获取附带参数
            IntentParams params =
commonEventData.getIntent().getParams();
            if (params == null) {
                return;
            }
            int wifiState= (int)
params.getParam(WifiEvents.PARAM_ACTIVE_STATE);

            if (wifiState== WifiEvents.STATE_ACTIVE) { // 处理 WLAN 被
打开消息
                HiLog.info(LABEL, false, "Receive
WifiEvents.STATE_ACTIVE %{public}d", wifiState);
            } else if (wifiState == WifiEvents.STATE_INACTIVE) { // 处
理 WLAN 被关闭消息
                HiLog.info(LABEL, false, "Receive
WifiEvents.STATE_INACTIVE %{public}d", wifiState);
            } else { // 处理 WLAN 异常状态
                HiLog.info(LABEL, false, "Unknown wifi state");
            }
        }
    }
}

// 注册消息
```

```
MatchingSkills match = new MatchingSkills();
// 增加获取 WLAN 状态变化消息
filter.addEvent(WifiEvents.EVENT_ACTIVE_STATE);
CommonEventSubscribeInfo subscribeInfo = new
CommonEventSubscribeInfo(match);
subscribeInfo.setPriority(100);
WifiEventSubscriber subscriber = new
WifiEventSubscriber(subscribeInfo);


try {
    CommonEventManager.subscribeCommonEvent(subscriber);
} catch (RemoteException e) {
    HiLog.warn(LABEL, false, "subscribe in wifi events failed!");
}
```

网络管理

概述

HarmonyOS 网络管理模块主要提供以下功能：

- 数据连接管理：网卡绑定，打开 URL，数据链路参数查询。
- 数据网络管理：指定数据网络传输，获取数据网络状态变更，数据网络状态查询。
- 流量统计：获取蜂窝网络、所有网卡、指定应用或指定网卡的数据流量统计值。
- HTTP 缓存：有效管理 HTTP 缓存，减少数据流量。

约束与限制

使用网络管理模块的相关功能时，需要请求相应的权限。

权限名	权限描述
ohos.permission.GET_NETWORK_INFO	获取网络连接信息。
ohos.permission.SET_NETWORK_INFO	修改网络连接状态。
ohos.permission.INTERNET	允许程序打开网络套接字，进行网络连接。

使用当前网络打开一个 URL 链接

场景介绍

应用使用当前的数据网络打开一个 URL 链接。

接口说明

应用使用当前网络打开一个 URL 链接，所使用的接口说明如下。

表 1 网络管理功能的主要接口

类名	接口名	功能描述
NetManager	getInstance(Context context)	获取网络管理的实例对象。
	hasDefaultNet()	查询当前是否有默认可用的数据网络。
	getDefaultNet()	获取当前默认的数据网络句柄。
	addDefaultNetStatusCallback(NetStatusCallback callback)	获取当前默认的数据网络状态变化。
	setAppNet(NetHandle netHandle)	应用绑定该数据网络。
NetHandle	openConnection(URL url, Proxy proxy) throws IOException	使用该网络打开一个 URL 链接。

开发步骤

1. 调用 NetManager.getInstance(Context)获取网络管理的实例对象。
2. 调用 NetManager.getDefaultNet()获取默认的数据网络。
3. 调用 NetHandle.openConnection()打开一个 URL。
4. 通过 URL 链接实例访问网站。

```
NetManager netManager = NetManager.getInstance(null);

if (!netManager.hasDefaultNet()) {
    return;
}
NetHandle netHandle = netManager.getDefaultNet();

// 可以获取网络状态的变化
NetStatusCallback callback = new NetStatusCallback() {
    // 重写需要获取的网络状态变化的 override 函数
};
netManager.addDefaultNetStatusCallback(callback);

// 通过 openConnection 来获取 URLConnection
HttpURLConnection connection = null;
try {
    String urlString = "https://www.huawei.com/";
    URL url = new URL(urlString);

    URLConnection urlConnection = netHandle.openConnection(url,
                java.net.Proxy.NO_PROXY);
    if (urlConnection instanceof HttpURLConnection) {
        connection = (HttpURLConnection) urlConnection;
```

```
    }
    connection.setRequestMethod("GET");
    connection.connect();
    // 之后可进行 url 的其他操作
} catch(IOException e) {
} finally {
    connection.disconnect();
}
```

使用当前网络进行 Socket 数据传输

场景介绍

应用使用当前的数据网络进行 Socket 数据传输。

接口说明

应用使用当前网络进行 Socket 数据传输，所使用的接口说明如下。

表 1 网络管理功能的主要接口

类名	接口名	功能描述
NetManager	getByName(String host）	解析主机名，获取其 IP 地址。
	bindSocket(Socket socket)	绑定 Socket 到该数据网络。
NetHandle	bindSocket(DatagramSocket socket)	绑定 DatagramSocket 到该数据网络。

开发步骤

1. 调用 NetManager.getInstance(Context)获取网络管理的实例对象。
2. 调用 NetManager.getDefaultNet()获取默认的数据网络。
3. 调用 NetHandle.bindSocket()绑定网络。
4. 使用 socket 发送数据。

```
NetManager netManager = NetManager.getInstance(null);


if (!netManager.hasDefaultNet()) {
    return;
```

```
}
NetHandle netHandle = netManager.getDefaultNet();


// 通过 Socket 绑定来进行数据传输
try {
    InetAddress address = netHandle.getByName("www.huawei.com");
    DatagramSocket socket = new DatagramSocket();
    netHandle.bindSocket(socket);
    byte[] buffer = new byte[1024];
    DatagramPacket request = new DatagramPacket(buffer, buffer.length,
address, port);
    // buffer 赋值


    // 发送数据
    socket.send(request);
} catch(IOException e) {
    e.printStackTrace();
}
```

使用指定网络进行数据访问

场景介绍

应用可以调用 API 接口来使用指定网络进行数据传输。在进行数据传输前，需要先建立自定义的网络类型。

接口说明

应用使用指定网络进行数据访问，所使用的接口说明如下。

表 1 网络管理功能的主要接口

类名	接口名	功能描述
NetSpecifier	Builder()	创建一个指定网络实例。
NetManager	setupSpecificNet(NetSpecifier netSpecifier, NetStatusCallback callback)	建立指定的数据网络。
	removeNetStatusCallback(NetStatusCallback callback)	停止获取数据网络状态。

开发步骤

1. 调用 NetSpecifier.Builder()构建指定数据网络的实例。
2. 调用 NetManager.setupSpecificNet()建立数据网络，通过 callback 获取网络状态变化。
3. 进行数据发送。

```
NetManager netManager = NetManager.getInstance(null);


private class MmsCallback extends NetStatusCallback {
```

```
    @Override
    public void onAvailable(NetHandle netHandle) {
        // 通过 setAppNet 把后续应用所有的请求都通过该网络进行发送
        netManager.setAppNet(netHandle);
        HttpURLConnection connection = null;
        try {
            String urlString = "https://www.huawei.com/";
            URL url = new URL(urlString);
            URLConnection urlConnection =
netHandle.openConnection(url, java.net.Proxy.NO_PROXY);
            if (urlConnection instanceof HttpURLConnection) {
                connection = (HttpURLConnection) urlConnection;
            }
            connection.setRequestMethod("GET");
            connection.connect();
            // 之后可进行 url 的其他操作
        } finally {
            connection.disconnect();
        }

        // 如果业务执行完毕，可以停止获取
        netManager.removeNetStatusCallback(this);
    }
}

MmsCallback callback = new MmsCallback();

// 配置一个彩信类型的蜂窝网络
```

```
NetSpecifier req = new NetSpecifier.Builder()
    .addCapability(NetCapabilities.NET_CAPABILITY_MMS)
    .addBearer(NetCapabilities.BEARER_CELLULAR)
    .build();

// 建立数据网络，通过 callback 获取网络变更状态
netManager.setupSpecificNet(req, callback);
```

流量统计

场景介绍

应用通过调用 API 接口，可以获取蜂窝网络、所有网卡、指定应用或指定网卡的数据流量统计值。

接口说明

应用进行流量统计，所使用的接口主要由 DataFlowStatistics 提供。

表 1 DataFlowStatistics 的主要接口

接口名	功能描述
getCellularRxBytes()	获取蜂窝数据网络的下行流量。
getCellularTxBytes()	获取蜂窝数据网络的上行流量。
getAllRxBytes()	获取所有网卡的下行流量。
getAllTxBytes()	获取所有网卡的上行流量。
getUidRxBytes(int uid)	获取指定 UID 的下行流量。
getUidTxBytes(int uid)	获取指定 UID 的上行流量。
getIfaceRxBytes(String nic)	获取指定网卡的下行流量。
getIfaceTxBytes(String nic)	获取指定网卡的上行流量。

开发步骤

调用 DataFlowStatistics 的接口可进行流量统计，以统计指定应用进程的流量为例。

```
long rx = DataFlowStatistics.getUidRxBytes(uid);
long tx = DataFlowStatistics.getUidTxBytes(uid);

// 进行数据收发

// 统计流量
rx = DataFlowStatistics.getUidRxBytes(uid) - rx;
tx = DataFlowStatistics.getUidTxBytes(uid) - tx;
```

管理 HTTP 缓存

场景介绍

应用重复打开一个相同网页时，可以优先从缓存文件里读取内容，从而减少数据流量，降低设备功耗，提升应用性能。

接口说明

管理 HTTP 缓存的功能主要由 HttpResponseCache 类提供。

表 1 HttpResponseCache 的主要接口

接口名	功能描述
install(File directory, long size)	使能 HTTP 缓存，设置缓存保存目录及大小。
getInstalled()	获取缓存实例。
flush()	立即保存缓存信息到文件系统中。
close()	关闭缓存功能。
delete()	关闭并清除缓存内容。

开发步骤

1. 配置缓存目录及最大缓存空间。
2. 保存缓存。
3. 关闭缓存。

```
// 初始化时设置缓存目录 dir 及最大缓存空间
```

```
HttpResponseCache.install(dir, 10 * 1024 * 1024);

// 访问 URL

// 为确保缓存保存到文件系统可以执行 flush 操作
HttpResponseCache.getInstalled().flush();

// 结束时关闭缓存
HttpResponseCache.getInstalled().close();
```

电话服务

概述

电话服务系统，除了为用户提供拨打语音/视频呼叫以及发送标准短信的功能以外，还提供了一系列的 API 用于获取无线蜂窝网络和 SIM 卡相关的一些信息。

其中拨打电话相关功能由 DistributedCallManager 提供，短信服务能力由 ShortMessageManager 提供。

应用还可以通过调用 RadioInfoManager 中的 API，来获取当前注册网络名称、网络服务状态以及信号强度等信息；以及调用 SimInfoManager 中的 API，来获取 SIM 卡的相关信息。

约束与限制

1. 部分 API 接口需要一定访问权限才能调用，因此三方应用在调用有权限控制的 API 时，需要先申请对应权限，权限申请详见权限章节。
2. 语音呼叫和短信功能暂不支持传入卡槽编号(SlotId)，双卡场景下的使用规则详见发起一路呼叫和发送一条文本信息的场景介绍。
3. 注册获取 SIM 卡状态接口仅针对有 SIM 卡在位场景生效，若用户拔出 SIM 卡，则接收不到回调事件。应用可通过调用 hasSimCard 接口来确定当前卡槽是否有卡在位。

发起一路呼叫

场景介绍

当应用需要发起一路呼叫给一个指定的号码时，使用本业务。呼叫可以是音频呼叫，也可以是视频呼叫。

如果设备支持同时插入两张 SIM 卡，且拨打电话时两张 SIM 卡均在位，呼叫时会弹出弹框让用户选择从卡 1 还是卡 2 呼出。

接口说明

DistributedCallManager 为开发者提供呼叫管理功能，具体功能分类如下表。

表 1 DistributedCallManager 的主要接口

功能分类	接口名	描述	所需权限
能力获取	hasVoiceCapability()	检查当前设备是否支持语音呼叫。	无
获取管理对象	getInstance(Context context)	获取呼叫管理对象。	无
发	dial(String number, boolean	发起音频	ohos.permission.PLACE_CALL

表 1 DistributedCallManager 的主要接口

功能分类	接口名	描述	所需权限
起呼叫	isVideoCall)	或视频呼叫。	
观察通话业务状态变化	addObserver(CallStateObserver observer, int mask)	观察通话业务状态变化。	ohos.permission.READ_CALL_LOG（获取通话号码需要该权限）

开发步骤

1. 调用 DistributedCallManager 的 getInstance 接口，创建/获取呼叫管理对象。
2. 调用 hasVoiceCapability()接口获取当前设备呼叫能力，如果支持继续下一步；如果不支持则无法发起呼叫。
3. 发起一路呼叫。
4. 注册观察呼叫状态变化。

```
// 创建呼叫管理对象
DistributedCallManager dcManager =
DistributedCallManager.getInstance(context);


// 调用查询能力接口
if (!dcManager.hasVoiceCapability()) {
```

```
    return;
}

// 如果设备支持呼叫能力，则继续发起呼叫
dcManager.dial(destinationNum, isVideoCall);

// 创建继承 CallStateObserver 的类 MyCallStateObserver
class MyCallStateObserver extends CallStateObserver {
    // 构造方法，在当前线程的 runner 中执行回调，slotId 需要传入要观察的卡槽 ID（0 或 1）
    MyCallStateObserver(int slotId) {
        super(slotId);
    }

    // 构造方法，在执行 runner 中执行回调，slotId 需要传入要观察的卡槽 ID（0 或 1）
    MyCallStateObserver(int slotId, EventRunner runner) {
        super(slotId, runner);
    }

    // 通话状态变化的回调方法
    @Override
    public void onCallStateUpdated(int state, String number) {
        ...
    }
}

// 执行回调的 runner
EventRunner runner = EventRunner.create();
```

```
// 创建 MyCallStateObserver 的对象
MyCallStateObserver observer = new MyCallStateObserver(slotId, runner);

// 观察 OBSERVE_CALL_STATE 的变化
dcManager.addObserver(observer,
CallStateObserver.OBSERVE_CALL_STATE);
```

发送一条文本信息

场景介绍

应用需要发送一条短信给一个指定的号码时，使用本业务。发送信息需要经过短信中心，短信中心号码可以是运营商默认的，也可以由应用自己指定。

如果设备支持同时插入 2 张 SIM 卡，且 2 张 SIM 卡均在位时，短信会从默认 SIM 卡发出。应用可通过调用 getDefaultSmsSlotId 来获取当前发短信的默认 SIM 卡位置。目前 API 暂不支持短信发送结果通知和送达报告。

接口说明

ShortMessageManager 为开发者提供短信管理功能，具体功能分类如下表。

表 1 ShortMessageManager 的主要接口

功能分类	接口名	描述	所需权限
能力获取	hasSmsCapability()	检查当前设备是否支持短信收发。	无
获取管理对象	getInstance(Context context)	获取短信管理对象。	无
获取默认短信卡	getDefaultSmsSlotId()	获取默认短信卡对应卡槽 ID。	无

表 1 ShortMessageManager 的主要接口

功能分类	接口名	描述	所需权限
长短信转化	splitMessage(String content)	将超过 140 个字节的长短信（如中文 70 个字符，英文 160 个字符）拆分成多条短信。	ohos.permission.SEND_MESSAGES
发送短信	sendMessage(String destinationHost, String serviceCenter, String content)	发送单条短信。	ohos.permission.SEND_MESSAGES
	sendMultipartMessage(String destinationHost, String serviceCenter, ArrayList<String> parts)	发送拆分后的多条短信。	ohos.permission.SEND_MESSAGES

开发步骤

1. 调用 ShortMessageManager 的 getInstance 接口，创建/获取短信收发管理对象。
2. 调用 hasSmsCapability()接口获取当前设备短信收发能力，如果支持继续下一步；如果不支持则无法收发短信。
3. 发送短信。

```
// 创建短信收发管理对象
ShortMessageManager smManager =
ShortMessageManager.getInstance(context);


// 检查短信能力
```

```
if (!smManager.hasSmsCapability()) {
    return;
}

// 如果设备支持收发短信，则继续发送短信
// 发送短信前可先调用 splitMessage()接口判断拆分后的短信条数，然后决定调用长短信或普通短信发送接口
ArrayList<String> msgs = smManager.splitMessage(messageContent);
if (msgs.size() > 1) { // 长短信拆分发送
    smManager.sendMultipartMessage(destinationNumber, serviceCenter, msgs);
} else { // 一般文本短信发送
    smManager.sendMessage(destinationNumber, serviceCenter, messageContent);
}
```

获取当前蜂窝网络信号信息

场景介绍

应用通常需要获取用户所在蜂窝网络下信号信息，以便获取当前驻网质量。开发者可以通过本业务，获取到用户指定 SIM 卡当前所在网络下的信号信息。

接口说明

RadioInfoManager 类中提供了获取当前网络信号信息列表的方法。

表 1 RadioInfoManager 的主要接口

功能分类	接口名	描述	所需权限
获取管理对象	getInstance(Context context)	获取网络管理对象。	无
信号强度信息	getSignalInfoList(int slotId)	获取当前注册蜂窝网络信号强度信息。	无

开发步骤

1. 调用 RadioInfoManager 的 getInstance 接口，获取到 RadioInfoManager 实例。
2. 调用 getSignalInfoList(slotId)方法，返回所有 SignalInformation 列表。
3. 遍历 SignalInformation 列表，并分别根据 signalNetworkType 转换为对应制式的 SignalInformation 子类对象。
4. 调用子类中的方法，获取信号强度信息。

```
// 获取 RadioInfoManager 对象。
RadioInfoManager radioInfoManager =
RadioInfoManager.getInstance(context);
```

```
// 获取信号信息。
List<SignalInformation> signalList =
radioInfoManager.getSignalInfoList(slotId);


// 检查信号信息列表大小。
if (signalList.size() == 0) {
    return;
}
// 依次遍历 list 获取当前驻网 networkType 对应的信号信息。
LteSignalInformation lteSignal;
for (SignalInformation signal : signalList) {
    int signalNetworkType = signal.getSignalNetworkType();
    if (signalNetworkType == TelephonyConstants.NETWORK_TYPE_LTE) {
        lteSignal = (LteSignalInformation) signal;
    }
}
// 调用子类中相应方法，获取对应制式的信号强度信息。
int signalLevel = lteSignal.getSignalLevel();
```

观察蜂窝网络状态变化

场景介绍

应用可以通过观察蜂窝网络状态变化，来接收最新蜂窝网络服务状态信息、信号信息等。

接口说明

RadioStateObserver 类中提供了观察蜂窝网络状态变化的方法，为了能够实时观察蜂窝网络状态变化，应用必须包含以下权限。

表 1 观察蜂窝网络状态变化需要的权限

观察状态名称	权限名称
网络状态信息(NETWORK_STATE)	ohos.permission.GET_NETWORK_INFO
信号信息(SIGNAL_INFO)	NA

需要使用 RadioInfoManager 的如下接口将继承 RadioStateObserver 类的对象注册到系统服务：

表 2 添加观察和停止观察接口 API 介绍

接口名	观察事件的掩码	描述
addObserver	OBSERVE_MASK_NETWORK_STATE	观察蜂窝网络驻网状态信息。
	OBSERVE_MASK_SIGNAL_INFO	观察蜂窝网络信号信息。
removeObserver	N/A	停止观察所有状态的变化。

开发步骤

添加观察事件

1. 调用 RadioInfoManager 的 getInstance 接口，获取到 RadioInfoManager 实例。
2. 创建继承 RadioStateObserver 的类 MyRadioStateObserver，并覆写状态变化回调方法。
3. 创建 MyRadioStateObserver 的对象 observer。
4. 调用 RadioInfoManager 的 addObserver 方法，传入已创建的 MyRadioStateObserver 对象 observer 和需要观察的 mask。

```
// 获取 RadioInfoManager 对象。
RadioInfoManager radioInfoManager =
RadioInfoManager.getInstance(context);
// 创建继承 RadioStateObserver 的类 MyRadioStateObserver
class MyRadioStateObserver extends RadioStateObserver {
    // 构造方法，在当前线程的 runner 中执行回调，slotId 需要传入要观察的卡槽 ID（0 或 1）。
    MyRadioStateObserver(int slotId) {
        super(slotId);
    }

    // 构造方法，在执行 runner 中执行回调。
    MyRadioStateObserver(int slotId, EventRunner runner) {
        super(slotId, runner);
    }

    // 网络注册状态变化的回调方法。
    @Override
    public void onNetworkStateUpdated(NetworkState state) {
        ...
    }
```

```
    // 信号信息变化的回调方法。
    @Override
    public void onSignalInfoUpdated(List<SignalInformation>
signalInfos) {
        ...
    }
}

// 执行回调的 runner。
EventRunner runner = EventRunner.create();

// 创建 MyRadioStateObserver 的对象。
MyRadioStateObserver observer = new MyRadioStateObserver(slotId,
runner);

// 添加回调，以 NETWORK_STATE 和 SIGNAL_INFO 为例。
radioInfoManager.addObserver(observer,
RadioStateObserver.OBSERVE_MASK_NETWORK_STATE |
RadioStateObserver.OBSERVE_MASK_SIGNAL_INFO);
```

停止观察

1. 调用 RadioInfoManager 的 getInstance 接口，获取到 RadioInfoManager 实例。
2. 调用 RadioInfoManager 的 removeObserver 方法，传入添加观察事件时创建的 MyRadioStateObserver 对象 observer。

```
// 获取 RadioInfoManager 对象。
RadioInfoManager radioInfoManager =
RadioInfoManager.getInstance(context);
// 停止观察
radioInfoManager.removeObserver(observer);
```

设备管理

传感器

概述

基本概念

HarmonyOS 传感器是应用访问底层硬件传感器的一种设备抽象概念。开发者根据传感器提供的 Sensor API，可以查询设备上的传感器，订阅传感器的数据，并根据传感器数据定制相应的算法，开发各类应用，比如指南针、运动健康、游戏等。

根据传感器的用途，可以将传感器分为六大类：运动类传感器、环境类传感器、方向类传感器、光线类传感器、健康类传感器、其他类传感器（如霍尔传感器），每一大类传感器包含不同类型的传感器，某种类型的传感器可能是单一的物理传感器，也可能是由多个物理传感器复合而成。传感器列表如图 1 所示。

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
运动类	ohos.sensor.agent.CategoryMotionAgent	SENSOR_TYPE_ACCELEROMETER	加速度传感器	测量三个物理轴（x、y和	检测运动状态

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
				z）上，施加在设备上的加速度（包括重力加速度），单位：m/s2	
		SENSOR_TYPE_ACCELEROMETER_UNCALIBRATED	未校准	测量三	检测加

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
			加速度传感器	个物理轴（x、y和z）上，施加在设备上的未校准的加速度（包括重力加速	速度偏差估值

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
				度），单位：m/s2	
		SENSOR_TYPE_LINEAR_ACCELERATION	线性加速度传感器	测量三个物理轴（x、y和z）上，施加在设备上的线性加	检测每个单轴方向上的线性加速度

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
				速度（不包括重力加速度），单位：m/s2	
		SENSOR_TYPE_GRAVITY	重力传感器	测量三个物理轴（x、y和z）上，	测量重力大小

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
				施加在设备上的重力加速度，单位：m/s2	
		SENSOR_TYPE_GYROSCOPE	陀螺仪传感器	测量三个物理轴（x、y和z）上，	测量旋转的角速度

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
				设备的旋转角速度，单位：rad/s	
		SENSOR_TYPE_GYROSCOPE_UNCALIBRATED	未校准陀螺仪传感器	测量三个物理轴（x、y和z）上，设备的未	测量旋转的角速度及偏差估值

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
				校准旋转角速度，单位：rad/s	
		SENSOR_TYPE_SIGNIFICANT_MOTION	大幅度动作传感器	测量三个物理轴（x、y和z）上，设备是否存	用于检测设备是否存在大幅度运动

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
				在大幅度运动；如果取值为 1 则代表存在大幅度运动，取值为 0 则代表没	

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
				有大幅度运动	
		SENSOR_TYPE_DROP_DETECTION	跌落检测传感器	检测设备的跌落状态；如果取值为 1 则代表发生跌落，取	用于检测设备是否发生了跌落

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
				值为0则代表没有发生跌落	
		SENSOR_TYPE_PEDOMETER_DETECTION	计步器检测传感器	检测用户的计步动作；如果取值为1则代	用于检测用户是否有计步的动作

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
				表用户产生了计步行走的动作；取值为 0 则代表用户没有发生运动	
		SENSOR_TYPE_PEDOMETE	计	统	用

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
		R	步器传感器	计用户的行走步数	于提供用户行走的步数数据
环境类	ohos.sensor.agent.CategoryEnvironmentAgent	SENSOR_TYPE_AMBIENT_TEMPERATURE	环境温度传感器	测量环境温度，单位：摄氏度(°C)	测量环境温度
		SENSOR_TYPE_MAGNETIC_FIELD	磁场传	测量三	创建指

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
			感器	个物理轴向（x、y、z）上，环境地磁场，单位：μT	南针
		SENSOR_TYPE_MAGNETIC_FIELD_UNCALIBRATED	未校准磁场传感器	测量三个物理轴向（x、y、z）上，	测量地磁偏差估值

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
				未校准环境地磁场，单位：μT	
		SENSOR_TYPE_HUMIDITY	湿度传感器	测量环境的相对湿度，以百分比(%)表示	监测露点、绝对湿度和相对湿度
		SENSOR_TYPE_BAROMET	气	测	测

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
		ER	压计传感器	量环境气压，单位：hPa 或 mbar	量环境气压
		SENSOR_TYPE_SAR	比吸收率传感器	测量比吸收率，单位：W/kg	测量设备的电磁波能量吸收比值。
方	ohos.sensor.agent.Cate	SENSOR_TYPE_6DOF	6	测	检

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
向类	goryOrientationAgent		自由度传感器	量上下、前后、左右方向上的位移，单位：m 或 mm；测量俯仰、偏摆、翻滚的角度，	测设备的三个平移自由度以及旋转自由度，用于目标定位追踪，如：

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
				单位：rad	VR
		SENSOR_TYPE_SCREEN_ROTATION	屏幕旋转传感器	检测设备屏幕的旋转状态	用于检测设备屏幕是否发生了旋转
		SENSOR_TYPE_DEVICE_ORIENTATION	设备方向传感器	测量设备的旋转方向，单	用于检测设备旋转方向

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
				位：rad	的角度值
		SENSOR_TYPE_ORIENTATION	方向传感器	测量设备围绕所有三个物理轴（x、y、z）旋转的角度值，单位：rad	用于提供屏幕旋转的3个角度值

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
		SENSOR_TYPE_ROTATION_VECTOR	旋转矢量传感器	测量设备旋转矢量，复合传感器：由加速度传感器、磁场传感器、陀螺仪传感	检测设备相对于东北天坐标系的方向

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
				器合成	
		SENSOR_TYPE_GAME_ROTATION_VECTOR	游戏旋转矢量传感器	测量设备游戏旋转矢量，复合传感器：由加速度传感器、陀螺仪传	应用于游戏场景

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
				感器合成	
		SENSOR_TYPE_GEOMAGNETIC_ROTATION_VECTOR	地磁旋转矢量传感器	测量设备地磁旋转矢量，复合传感器：由加速度传感器、磁场传	用于测量地磁旋转矢量

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
				感器合成	
光线类	ohos.sensor.agent.CategoryLightAgent	SENSOR_TYPE_PROXIMITY	接近光传感器	测量可见物体相对于设备显示屏的接近或远离状态	通话中设备相对人的位置
		SENSOR_TYPE_TOF	To	测量	人脸

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
			F 传感器	光在介质中行进一段距离所需的时间	识别
		SENSOR_TYPE_AMBIENT_LIGHT	环境光传感器	测量设备周围光线强度，单位：lux	自动调节屏幕亮度，检测屏幕

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
					上方是否有遮挡
		SENSOR_TYPE_COLOR_TEMPERATURE	色温传感器	测量环境中的色温	应用于设备的影像处理
		SENSOR_TYPE_COLOR_RGB	RGB颜色传感器	测量环境中的RGB颜色值	通过三原色的反射比率实

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
					现颜色检测
		SENSOR_TYPE_COLOR_XYZ	XYZ 颜色传感器	测量环境中的 XYZ 颜色值	用于辨识真色色点，还原色彩更真实
健康类	ohos.sensor.agent.CategoryBodyAgent	SENSOR_TYPE_HEART_RATE	心率传感器	测量用户的心率	用于提供用户的

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
				数值	心率健康数据
		SENSOR_TYPE_WEAR_DETECTION	佩戴检测传感器	检测用户是否佩戴	用于检测用户是否佩戴智能穿戴
其他类	ohos.sensor.agent.CategoryOtherAgent	SENSOR_TYPE_HALL	霍尔传感器	测量设备周围是否	设备的皮套模式

表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
				存在磁力吸引	
		SENSOR_TYPE_GRIP_DETECTOR	手握检测传感器	检测设备是否有抓力施加	用于检查设备侧边是否被手握住
		SENSOR_TYPE_MAGNET_BRACKET	磁铁支架传感器	检测设备是否被磁	检测设备是否位于

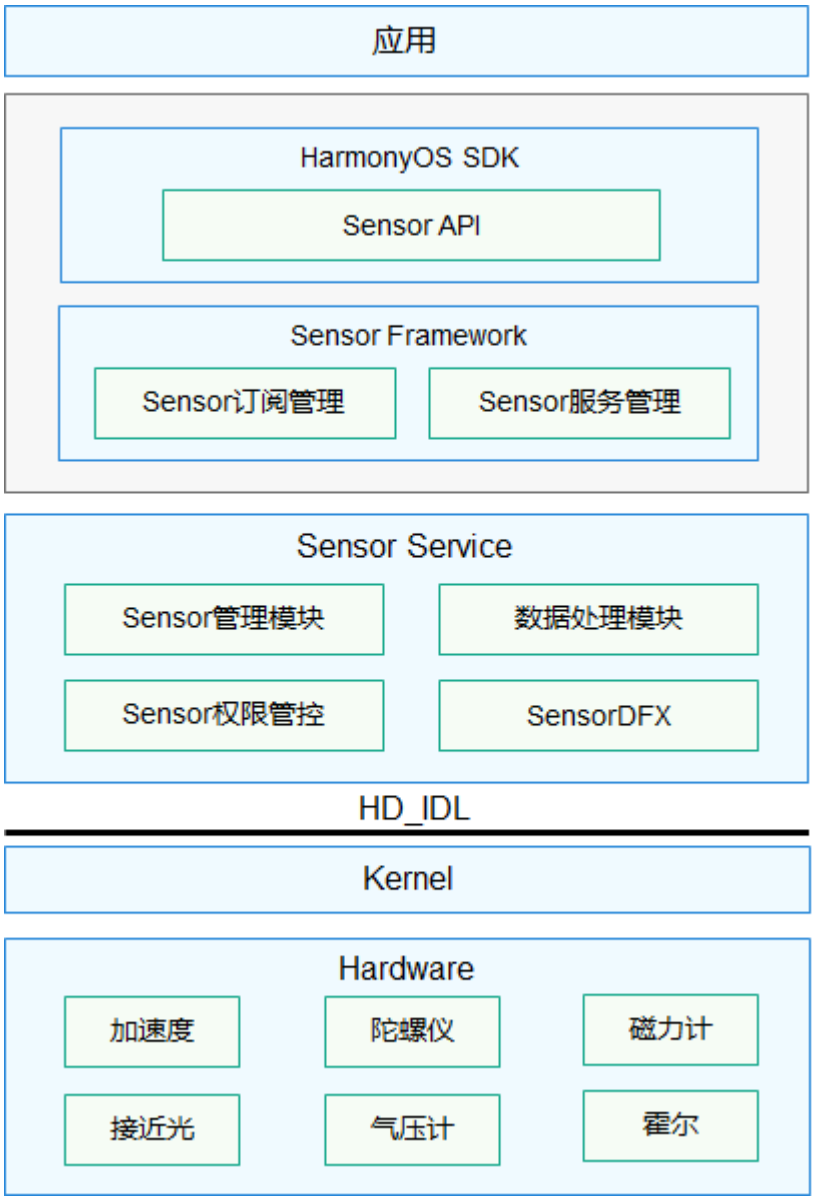
表 1 传感器列表

分类	API 类名	传感器类型	中文描述	说明	主要用途
				吸	车内或者室内
		SENSOR_TYPE_PRESSURE_DETECTOR	按压检测传感器	检测设备是否有压力施加	用于检测设备的正上方是否存在按压

运作机制

HarmonyOS 传感器包含如下四个模块：Sensor API、Sensor Framework、Sensor Service、HD_IDL 层。

图 1　HarmonyOS 传感器



- Sensor API：提供传感器的基础 API，主要包含查询传感器的列表、订阅/取消传感器的数据、执行控制命令等，简化应用开发。
- Sensor Framework：主要实现传感器的订阅管理，数据通道的创建、销毁、订阅与取消订阅，实现与 SensorService 的通信。
- Sensor Service：主要实现 HD_IDL 层数据接收、解析、分发，前后台的策略管控，对该设备 Sensor 的管理；Sensor 权限管控等。
- HD_IDL 层：对不同的 FIFO、频率进行策略选择；以及对不同设备（车机、智能穿戴、智慧屏等）的适配。

约束与限制

1. 针对某些传感器，开发者需要请求相应的权限，才能获取到相应传感器的数据。

表 2 HarmonyOS 传感器权限列表

传感器	HarmonyOS 权限名	敏感级别	权限描述
加速度传感器、加速度未校准传感器、线性加速度传感器	ohos.permission.ACCELEROMETER	system_grant	允许订阅 Motion 组对应的加速度传感器的数据
陀螺仪传感器、陀螺仪未校准传感器	ohos.permission.GYROSCOPE	system_grant	允许订阅 Motion 组对应的陀螺仪传感器的数据
计步器	ohos.permission.ACTIVITY_MOTION	user_grant	允许订阅运动状态
心率	ohos.permission.READ_HEALTH_DATA	user_grant	允许读取健康数据

2. 传感器数据订阅和取消订阅接口成对调用，当不再需要订阅传感器数据时，开发者需要调用取消订阅接口进行资源释放。

开发指导

场景介绍

- 通过方向传感器数据，可以感知用户设备当前的朝向，从而达到为用户指明方位的目的。
- 通过重力和陀螺仪传感器数据，能感知设备倾斜和旋转量，提高用户在游戏场景中的体验。
- 通过接近光传感器数据，感知距离遮挡物的距离，使设备能够自动亮灭屏，达到防误触目的。
- 通过气压计传感器数据，可以准确的判断设备当前所处的海拔。
- 通过环境光传感器数据，设备能够实现背光自动调节。

接口说明

HarmonyOS 传感器提供的功能有：查询传感器的列表、订阅/取消订阅传感器数据、查询传感器的最小采样时间间隔、执行控制命令。

以订阅方向类别的传感器数据为例，本节示例涉及的接口如下：

表 1 CategoryOrientationAgent 的主要接口

接口名	描述
getAllSensors()	获取属于方向类别的传感器列表。
getAllSensors(int)	获取属于方向类别中特定类型的传感器列表。
getSingleSensor(int)	查询方向类别中特定类型的默认 sensor（如果存在多个则返回第一个）。
setSensorDataCallback(ICategoryOrientationDataCallback, CategoryOrientation, long)	以设定的采样间隔订阅给定传感器的数据。

表 1 CategoryOrientationAgent 的主要接口

接口名	描述
setSensorDataCallback(ICategoryOrientationDataCallback, CategoryOrientation, long, long)	以设定的采样间隔和时延订阅给定传感器的数据。
releaseSensorDataCallback(ICategoryOrientationDataCallback, CategoryOrientation)	取消订阅指定传感器的数据。
releaseSensorDataCallback(ICategoryOrientationDataCallback)	取消订阅的所有传感器数据。

表 2 SensorAgent 的主要接口

接口名	描述
getSensorMinSampleInterval(int)	查询给定传感器的最小采样间隔。
runCommand(int, int, int)	针对某个传感器执行命令，刷新传感器的数据。

开发步骤

权限配置

如果设备上使用了表 2 中的传感器，需要请求相应的权限，开发者才能获取到传感器数据。

表 3 不同敏感级别的 HarmonyOS 传感器举例

敏感级别	传感器	HarmonyOS 权限名	权限描述
system_grant	加速度传感器、加速度未校准传感器、线	ohos.permission.ACCELEROMETER	允许订阅 Motion 组对应的加速度传感器的数据。

表 3 不同敏感级别的 HarmonyOS 传感器举例

敏感级别	传感器	HarmonyOS 权限名	权限描述
	性加速度传感器		
user_grant	计步器	ohos.permission.ACTIVITY_MOTION	允许订阅运动状态。

开发者需要在 config.json 里面配置权限：

- 开发者如果需要获取加速度的数据，需要进行如下权限配置。

```
"reqPermissions": [
    {
        "name": "ohos.permission.ACCELEROMETER",
        "reason": "",
        "usedScene": {
            "ability": [
                ".MainAbility"
            ],
            "when": "inuse"
        }
    }
]
```

对于需要用户授权的权限，如计步器传感器，需要进行如下权限配置。

```
"reqPermissions": [
    {
        "name": "ohos.permission.ACTIVITY_MOTION",
        "reason": "",
```

```
        "usedScene": {
            "ability": [
                ".MainAbility"
            ],
        "when": "inuse"
        }
    }
]
```

由于敏感权限需要用户授权，因此，开发者在应用启动时或者调用订阅数据接口前，需要调用权限检查和请求权限接口。

```
@Override
public void onStart(Intent intent) {
    super.onStart(intent);
    if (verifySelfPermission("ohos.permission.ACTIVITY_MOTION") != 0) {
        if (canRequestPermission("ohos.permission.ACTIVITY_MOTION")) {
            requestPermissionsFromUser(new String[]
{"ohos.permission.ACTIVITY_MOTION"}, 1);
        }
    }
    // ...
}


@Override
public void onRequestPermissionsFromUserResult(int requestCode, String[]
permissions,
        int[] grantResults) {
    switch (requestCode) {
```

```
        case 1: {
            // 匹配 requestPermissionsFromUser 的 requestCode
            if (grantResults.length > 0 && grantResults[0] == 0) {
                // 权限被授予
            } else {
                // 权限被拒绝
            }
            return;
        }
    }
}
```

使用传感器

以使用方向类别的传感器为例，运动类、环境类、健康类等类别的传感器使用方法类似。

1. 获取待订阅数据的传感器。
2. 创建传感器回调。
3. 订阅传感器数据。
4. 接收并处理传感器数据。
5. 取消订阅传感器数据。

```
private Button btnSubscribe;


private Button btnUnsubscribe;


private CategoryOrientationAgent categoryOrientationAgent = new
CategoryOrientationAgent();


private ICategoryOrientationDataCallback orientationDataCallback;
```

```
private CategoryOrientation orientationSensor;

private long interval = 100000000;

@Override
public void onStart(Intent intent) {
    super.onStart(intent);
    super.setUIContent(ResourceTable.Layout_sensor_layout);
    findComponent(rootComponent);

    // 创建传感器回调对象。
    orientationDataCallback = new ICategoryOrientationDataCallback() {
        @Override
        public void onSensorDataModified(CategoryOrientationData
categoryOrientationData) {
            // 对接收的 categoryOrientationData 传感器数据对象解析和使
用
            int dim = categoryOrientationData.getSensorDataDim(); //
获取传感器的维度信息
            float degree = categoryOrientationData.getValues()[0]; //
获取方向类传感器的第一维数据
        }

        @Override
        public void onAccuracyDataModified(CategoryOrientation
categoryOrientation, int i) {
            // 使用变化的精度
        }
```

```
        @Override
        public void onCommandCompleted(CategoryOrientation
categoryOrientation) {
            // 传感器执行命令回调
        }
    };

    btnSubscribe.setClickedListener(v -> {
        // 获取传感器对象，并订阅传感器数据
        orientationSensor = categoryOrientationAgent.getSingleSensor(
                CategoryOrientation.SENSOR_TYPE_ORIENTATION);
        if (orientationSensor != null) {
            categoryOrientationAgent.setSensorDataCallback(
                    orientationDataCallback, orientationSensor,
interval);
        }
    });
    // 取消订阅传感器数据
    btnUnsubscribe.setClickedListener(v -> {
        if (orientationSensor != null) {
            categoryOrientationAgent.releaseSensorDataCallback(
                    orientationDataCallback, orientationSensor);
        }
    });
}

private void findComponent(Component component) {
    btnSubscribe = (Button)
component.findComponentById(Resource.Id.btnSubscribe);
```

```
    btnUnsubscribe = (Button)
component.findComponentById(Resource.Id.btnUnsubscribe);

}
```

控制类小器件

概述

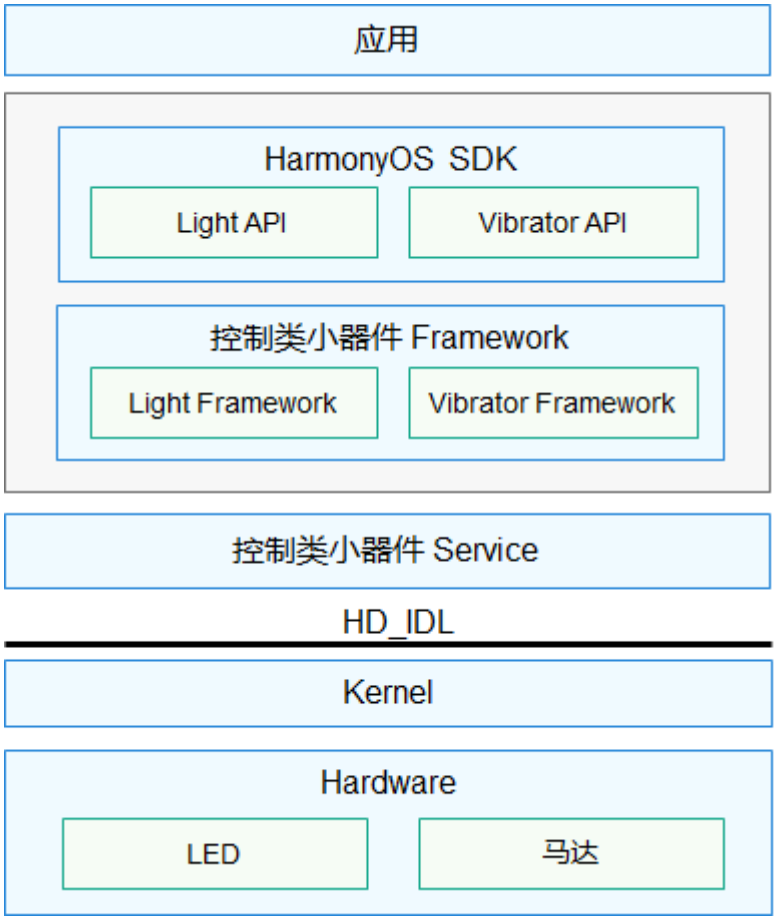
基本概念

控制类小器件指的是设备上的 LED 灯和振动器。其中，LED 灯主要用作指示（如充电状态）、闪烁功能（如三色灯）等；振动器主要用于闹钟、开关机振动、来电振动等场景。

运作机制

控制类小器件主要包含以下四个模块：控制类小器件 API、控制类小器件 Framework、控制类小器件 Service、HD_IDL 层。

图 1　控制类小器件



- 控制类小器件 API：提供灯和振动器基础的 API，主要包含灯的列表查询、打开灯、关闭灯等接口，振动器的列表查询、振动器的振动器效果查询、触发/关闭振动器等接口。
- 控制类小器件 Framework：主要实现灯和振动器的框架层管理，实现与控制类小器件 Service 的通信。
- 控制类小器件 Service：实现灯和振动器的服务管理。
- HD_IDL 层：对不同设备（车机、智能穿戴、智慧屏等）的适配。

约束与限制

- 在调用 Light API 时，请先通过 getLightIdList 接口查询设备所支持的灯的 ID 列表，以免调用打开接口异常。
- 在调用 Vibrator API 时，请先通过 getVibratorIdList 接口查询设备所支持的振动器的 ID 列表，以免调用振动接口异常。
- 在使用振动器时，开发者需要配置请求振动器的权限 ohos.permission.VIBRATE，才能控制振动器振动。

Light 开发指导

场景介绍

当设备需要设置不同的闪烁效果时，可以调用 Light 模块，例如，LED 灯能够设置灯颜色、灯亮和灯灭时长的闪烁效果。

说明

使用该功能依赖于硬件设备是否具有 LED 灯。

接口说明

灯模块主要提供的功能有：查询设备上灯的列表，查询某个灯设备支持的效果，打开和关闭灯设备。LightAgent 类开放能力如下，具体请查阅 API 参考文档。

表 1 LightAgent 的主要接口

接口名	描述
getLightIdList()	获取硬件设备上的灯列表。
isSupport(int)	根据指定灯 Id 查询硬件设备是否有该灯。
isEffectSupport(int, String)	查询指定的灯是否支持指定的闪烁效果。
turnOn(int, String)	对指定的灯创建指定效果的一次性闪烁。
turnOn(int, LightEffect)	对指定的灯创建自定义效果的一次性闪烁。
turnOn(String)	对指定的灯创建指定效果的一次性闪烁。

表 1 LightAgent 的主要接口

接口名	描述
turnOn(LightEffect)	对指定的灯创建自定义效果的一次性闪烁。
turnOff(int)	关闭指定的灯。
turnOff()	关闭指定的灯。

开发步骤

1. 查询硬件设备上灯的列表。
2. 查询指定的灯是否支持指定的闪烁效果。
3. 创建不同的闪烁效果。
4. 关闭指定的灯。

```
private LightAgent lightAgent = new LightAgent();


@Override
public void onStart(Intent intent) {
    super.onStart(intent);
    super.setUIContent(ResourceTable.Layout_light_layout);


    // ...


    // 查询硬件设备上的灯列表
    List<Integer> myLightList = lightAgent.getLightIdList();
    if (myLightList.isEmpty()) {
        return;
    }
```

```
    int lightId = myLightList.get(0);


    // 查询指定的灯是否支持指定的闪烁效果
    boolean isSupport = lightAgent.isEffectSupport(lightId,
LightEffect.LIGHT_ID_KEYBOARD);


    // 创建指定效果的一次性闪烁
    boolean turnOnResult = lightAgent.turnOn(lightId,
LightEffect.LIGHT_ID_KEYBOARD);


    // 创建自定义效果的一次性闪烁
    LightBrightness lightBrightness = new LightBrightness(255, 255,
255);
    LightEffect lightEffect = new LightEffect(lightBrightness, 1000,
1000);
    boolean turnOnEffectResult = lightAgent.turnOn(lightId,
lightEffect);


    // 关闭指定的灯
    boolean turnOffResult = lightAgent.turnOff(lightId);
}
```

Vibrator 开发指导

场景介绍

当设备需要设置不同的振动效果时，可以调用 Vibrator 模块，例如，设备的按键可以设置不同强度和时长的振动，闹钟和来电可以设置不同强度和时长的单次或周期性振动。

接口说明

振动器模块主要提供的功能有：查询设备上振动器的列表，查询某个振动器是否支持某种振动效果，触发和关闭振动器。VibratorAgent 类开放能力如下，具体请查阅 API 参考文档。

表 1 VibratorAgent 的主要接口

接口名	描述
getVibratorIdList()	获取硬件设备上的振动器列表。
isSupport(int)	根据指定的振动器 Id 查询硬件设备是否存在该振动器。
isEffectSupport(int, String)	查询指定的振动器是否支持指定的震动效果。
startOnce(int, String)	对指定的振动器创建指定效果的一次性振动。
startOnce(String)	对指定的振动器创建指定效果的一次性振动。
startOnce(int, int)	对指定的振动器创建指定振动时长的一次性振动。

表 1 VibratorAgent 的主要接口

接口名	描述
startOnce(int)	对指定的振动器创建指定振动时长的一次性振动。
start(int, VibrationPattern)	对指定的振动器创建自定义效果的波形或一次性振动。
start(VibrationPattern)	对指定的振动器创建自定义效果的波形或一次性振动。
stop(int, String)	关闭指定的振动器指定模式的振动。
stop(String)	关闭指定的振动器指定模式的振动。

开发步骤

1. 控制设备上的振动器，需要在“config.json”里面进行配置请求权限。具体如下：

```
"reqPermissions": [
    {
        "name": "ohos.permission.VIBRATE",
        "reason": "",
        "usedScene": {
            "ability": [
                ".MainAbility"
            ],
            "when": "inuse"
        }
```

```
    }
]
```

1. 查询硬件设备上的振动器列表。
2. 查询指定的振动器是否支持指定的震动效果。
3. 创建不同效果的振动。
4. 关闭指定的振动器指定模式的振动。

```
private VibratorAgent vibratorAgent = new VibratorAgent();

private int[] timing = {1000, 1000, 2000, 5000};

private int[] intensity = {50, 100, 200, 255};

@Override
public void onStart(Intent intent) {
    super.onStart(intent);
    super.setUIContent(ResourceTable.Layout_vibrator_layout);

    // ...

    // 查询硬件设备上的振动器列表
    List<Integer> vibratorList = vibratorAgent.getVibratorIdList();
    if (vibratorList.isEmpty()) {
        return;
    }
    int vibratorId = vibratorList.get(0);

    // 查询指定的振动器是否支持指定的振动效果
    boolean isSupport = vibratorAgent.isEffectSupport(vibratorId,
```

```
            VibrationPattern.VIBRATOR_TPYE_CAMERA_CLICK);


    // 创建指定效果的一次性振动
    boolean vibrateEffectResult = vibratorAgent.vibrate(vibratorId,
            VibrationPattern.VIBRATOR_TPYE_CAMERA_CLICK);


    // 创建指定振动时长的一次性振动
    int vibratorTiming = 1000;
    boolean vibrateResult = vibratorAgent.vibrate(vibratorId,
vibratorTiming);


    // 创建自定义效果的周期性波形振动
    int count = 5;
    VibrationPattern vibrationPeriodEffect =
VibrationPattern.createPeriod(timing, intensity, count);
    boolean vibratePeriodResult = vibratorAgent.vibrate(vibratorId,
vibrationPeriodEffect);


    // 创建自定义效果的一次性振动
    VibrationPattern vibrationOnceEffect =
VibrationPattern.createSingle(3000, 50);
    boolean vibrateSingleResult = vibratorAgent.vibrate(vibratorId,
vibrationOnceEffect);


    // 关闭指定的振动器自定义模式的振动
    boolean stopResult = vibratorAgent.stop(vibratorId,
            VibratorAgent.VIBRATOR_STOP_MODE_CUSTOMIZED);
}
```

位置

概述

移动终端设备已经深入人们日常生活的方方面面，如查看所在城市的天气、新闻轶事、出行打车、旅行导航、运动记录。这些习以为常的活动，都离不开定位用户终端设备的位置。

当用户处于这些丰富的使用场景中时，系统的位置能力可以提供实时准确的位置数据。对于开发者，设计基于位置体验的服务，也可以使应用的使用体验更贴近每个用户。

当应用在实现基于设备位置的功能时，如：驾车导航，记录运动轨迹等，可以调用该模块的 API 接口，完成位置信息的获取。

基本概念

位置能力用于确定用户设备在哪里，系统使用位置坐标标示设备的位置，并用多种定位技术提供服务，如 GNSS 定位、基站定位、WLAN/蓝牙定位（基站定位、WLAN/蓝牙定位后续统称“网络定位技术”）。通过这些定位技术，无论用户设备在室内或是户外，都可以准确地确定设备位置。

- **坐标**

系统以 1984 年世界大地坐标系统为参考，使用经度、纬度数据描述地球上的一个位置。

- **GNSS 定位**

基于全球导航卫星系统，包含：GPS、GLONASS、北斗、Galileo 等，通过导航卫星，设备芯片提供的定位算法，来确定设备准确位置。定位过程具体使用哪些定位系统，取决于用户设备的硬件能力。

- **基站定位**

根据设备当前驻网基站和相邻基站的位置，估算设备当前位置。此定位方式的定位结果精度相对较低，并且需要设备可以访问蜂窝网络。

- **WLAN、蓝牙定位**

根据设备可搜索到的周围 WLAN、蓝牙设备位置，估算设备当前位置。此定位方式的定位结果精度依赖设备周围可见的固定 WLAN、蓝牙设备的分布，密度较高时，精度也相较与基站定位方式更高，同时也需要设备可以访问网络。

运作机制

位置能力作为系统为应用提供的一种基础服务，需要应用在所使用的业务场景，向系统主动发起请求，并在业务场景结束时，主动结束此请求，在此过程中系统会将实时的定位结果上报给应用。

约束与限制

使用设备的位置能力，需要用户进行确认并主动开启位置开关。如果位置开关没有开启，系统不会向任何应用提供位置服务。

设备位置信息属于用户敏感数据，所以即使用户已经开启位置开关，应用在获取设备位置前仍需向用户申请位置访问权限。在用户确认允许后，系统才会向应用提供位置服务。

获取设备的位置信息

场景介绍

开发者可以调用 HarmonyOS 位置相关接口，获取设备实时位置，或者最近的历史位置。

对于位置敏感的应用业务，建议获取设备实时位置信息。如果不需要设备实时位置信息，并且希望尽可能的节省耗电，开发者可以考虑获取最近的历史位置。

接口说明

获取设备的位置信息，所使用的接口说明如下。

表 1 获取位置信息 API 功能介绍

接口名	功能描述
Locator(Context context)	创建 Locator 实例对象。
RequestParam(int scenario)	根据定位场景类型创建定位请求的 RequestParam 对象。
onLocationReport(Location location)	获取定位结果。
startLocating(RequestParam request, LocatorCallback callback)	向系统发起定位请求。
requestOnce(RequestParam request, LocatorCallback callback)	向系统发起单次定位请求。
stopLocating(LocatorCallback callback)	结束定位。
getCachedLocation()	获取系统缓存的位置信息。

开发步骤

1. 应用在使用系统能力前，需要检查是否已经获取用户授权访问设备位置信息。如未获得授权，可以向用户申请需要的位置权限。

系统提供的定位权限有：

- ohos.permission.LOCATION
- ohos.permission.LOCATION_IN_BACKGROUND

访问设备的位置信息，必须申请 ohos.permission.LOCATION 权限，并且获得用户授权。
如果应用在后台运行时也需要访问设备位置，除需要将应用声明为允许后台运行外，还必须申请 ohos.permission.LOCATION_IN_BACKGROUND 权限，这样应用在切入后台之后，系统依然可以继续上报位置信息。
开发者可以在应用 config.json 文件中声明所需要的权限，示例代码如下：

```
{
    "reqPermissions": [{
        "name": "ohos.permission.LOCATION",
        "reason": "$string:reason_description",
        "usedScene": {
            "ability": ["com.myapplication.LocationAbility"],
            "when": "inuse"
        }, {
        ...
        }]
    }]
}
```

说明

配置字段详细说明见权限开发指导。在使用系统位置能力时，向用户动态申请位置权限，申请方式请参考动态申请权限开发步骤。

实例化 Locator 对象，所有与基础定位能力相关的功能 API，都是通过 Locator 提供的。

```
Locator locator = new Locator(context);
```

其中入参需要提供当前应用程序的 AbilityInfo 信息，便于系统管理应用的定位请求。

实例化 RequestParam 对象，用于告知系统该向应用提供何种类型的位置服务，以及位置结果上报的频率。

方式一：

为了面向开发者提供贴近其使用场景的 API 使用方式，系统定义了几种常见的位置能力使用场景，并针对使用场景做了适当的优化处理，应用可以直接匹配使用，简化开发复杂度。系统当前支持场景如下表所示。

表 2 定位场景类型说明

场景名称	常量定义	说明
导航场景	SCENE_NAVIGATION	适用于在户外定位设备实时位置的场景，如车载、步行导航。在此场景下，为保证系统提供位置结果精度最优，主要使用 GNSS 定位技术提供定位服务，结合场景特点，在导航启动之初，用户很可能在室内、车库等遮蔽环境，GNSS 技术很难提供位置服务。为解决此问题，我们会在 GNSS 提供稳定位置结果之前，使用系统网络定位技术，向应用提供位置服务，以在导航初始阶段提升用户体验。 此场景默认以最小 1 秒间隔上报定位结果，使用此场景的应用必须申请 ohos.permission.LOCATION 权限，同时获得用户授权。
轨迹跟踪场景	SCENE_TRAJECTORY_TRACKING	适用于记录用户位置轨迹的场景，如运动类应用记录轨迹功能。主要使用 GNSS 定位技术提供定位服务。 此场景默认以最小 1 秒间隔上报定位结果，并且应用必须申请 ohos.permission.LOCATION 权限，同时获得用户授权。
出行约车场景	SCENE_CAR_HAILING	适用于用户出行打车时定位当前位置的场景，如网约车类应用。 此场景默认以最小 1 秒间隔上报定位结果，并且应用必须申请 ohos.permission.LOCATION 权限，同时获得用户授权。

表 2 定位场景类型说明

场景名称	常量定义	说明
生活服务场景	SCENE_DAILY_LIFE_SERVICE	生活服务场景，适用于不需要定位用户精确位置的使用场景，如新闻资讯、网购、点餐类应用，做推荐、推送时定位用户大致位置即可。 此场景默认以最小 1 秒间隔上报定位结果，并且应用至少申请 ohos.permission.LOCATION 权限，同时获得用户授权。
无功耗场景	SCENE_NO_POWER	无功耗场景，适用于不需要主动启动定位业务。系统在响应其他应用启动定位业务并上报位置结果时，会同时向请求此场景的应用程序上报定位结果，当前的应用程序不产生定位功耗。 此场景默认以最小 1 秒间隔上报定位结果，并且应用需要申请 ohos.permission.LOCATION 权限，同时获得用户授权。

以导航场景为例，实例化方式如下：

```
.RequestParam requestParam = new
RequestParam(RequestParam.SCENE_NAVIGATION);
```

方式二：

如果定义的现有场景类型不能满足所需的开发场景，系统提供了基本的定位优先级策略类型。

表 3 定位优先级策略类型说明：

策略类型	常量定义	说明
定位精度优先策略	PRIORITY_ACCURACY	定位精度优先策略主要以 GNSS 定位技术为主，在开阔场景下可以提供米级的定位精度，具体性能指标依赖用户设备的定位硬件能力，但在室内等强遮蔽定位场景下，无法提供准确的位置服务。 应用必须申请 ohos.permission.LOCATION 权限，同时获得用户授权。
快速定位优先策略	PRIORITY_FAST_FIRST_FIX	快速定位优先策略会同时使用 GNSS 定位、基站定位和 WLAN、蓝牙定位技术，以便室内和户外场景下，通过此策略都可以获得位置结果，当各种定位技术都有提供位置结果时，系统会选择其中精度较好的结果返回给应用。 因为对各种定位技术同时使用，对设备的硬件资源消耗较大，功耗也较大。 应用必须申请 ohos.permission.LOCATION 权限，同时获得用户授权。
低功耗定位优先策略	PRIORITY_LOW_POWER	低功耗定位优先策略主要使用基站定位和 WLAN、蓝牙定位技术，也可以同时提供室内和户外场景下的位置服务，因为其依赖周边基站、可见 WLAN、蓝牙设备的分布情况，定位结果的精度波动

表 3 定位优先级策略类型说明：

策略类型	常量定义	说明
		范围较大，如果对定位结果精度要求不高，或者使用场景多在有基站、可见 WLAN、蓝牙设备高密度分布的情况下，推荐使用，可以有效节省设备功耗。 应用至少申请 ohos.permission.LOCATION 权限，同时获得用户授权。

以定位精度优先策略为例，实例化方式如下：

```
RequestParam requestParam = new
RequestParam(RequestParam.PRIORITY_ACCURACY,0,0);
```

后两个入参用于限定系统向应用上报定位结果的频率，分别为位置上报的最小时间间隔，和位置上报的最小距离间隔，开发者可以参考 API 具体说明进行开发。

实例化 LocatorCallback 对象，用于向系统提供位置上报的途径。

应用需要自行实现系统定义好的回调接口，并将其实例化。系统在定位成功确定设备的实时位置结果时，会通过 onLocationReport 接口上报给应用。应用程序可以在 onLocationReport 接口的实现中完成自己的业务逻辑。

```
MyLocatorCallback locatorCallback = new MyLocatorCallback();


public class MyLocatorCallback implements LocatorCallback {
    @Override
    public void onLocationReport(Location location) {
    }


    @Override
    public void onStatusChanged(int type) {
    }


    @Override
    public void onErrorReport(int type) {
    }
}
```

启动定位。

```
locator.startLocating(requestParam, locatorCallback);
```

如果应用不需要持续获取位置结果，可以使用如下方式启动定位，系统会上报一次实时定位结果后，自动结束应用的定位请求。应用不需要执行结束定位。

```
locator.requestOnce(requestParam, locatorCallback);
```

（可选）结束定位。

```
locator.stopLocating(locatorCallback);
```

如果应用使用场景不需要实时的设备位置，可以获取系统缓存的最近一次历史定位结果。

```
locator.getCachedLocation();
```

此接口的使用需要应用向用户申请 LOCATION 位置权限。

（逆）地理编码转化

场景介绍

使用坐标描述一个位置，非常准确，但是并不直观，面向用户表达并不友好。

系统向开发者提供了地理编码转化能力（将坐标转化为地理编码信息），以及逆地理编码转化能力（将地理描述转化为具体坐标）。其中地理编码包含多个属性来描述位置，包括国家、行政区划、街道、门牌号、地址描述等等，这样的信息更便于用户理解。

接口说明

进行坐标和地理编码信息的相互转化，所使用的接口说明如下。

表 1 地理编码转化能力和逆地理编码转化能力的 API 功能介绍

接口名	功能描述
GeoConvert()	创建 GeoConvert 实例对象。
GeoConvert(Locale locale)	根据自定义参数创建 GeoConvert 实例对象。
getAddressFromLocation(double latitude, double longitude, int maxItems)	根据指定的经纬度坐标获取地理位置信息。
getAddressFromLocationName(String description, int maxItems)	根据地理位置信息获取相匹配的包含坐标数据的地址列表。
getAddressFromLocationName(String description, double minLatitude, double minLongitude, double maxLatitude, double maxLongitude, int maxItems)	根据指定的位置信息和地理区域获取相匹配的包含坐标数据的地址列表。

开发步骤

1. 实例化 GeoConvert 对象，所有与(逆)地理编码转化能力相关的功能 API，都是通过 GeoConvert 提供的。

```
GeoConvert geoConvert = new GeoConvert();
```

获取转化结果。

坐标转化地理位置信息。

```
geoConvert.getAddressFromLocation(纬度值, 经度值, 1);
```

参考接口 API 说明，应用可以获得与此坐标匹配的 GeoAddress 列表，应用可以根据实际使用需求，读取相应的参数数据。

位置描述转化坐标。

```
geoConvert.getAddressFromLocationName("北京大兴国际机场", 1);
```

参考接口 API 说明，应用可以获得与位置描述相匹配的 GeoAddress 列表，其中包含对应的坐标数据，请参考 API 使用。

如果需要查询的位置描述可能出现多地重名的请求，可以同过设置一个经纬度范围，以便高效获取期望的准确结果。

```
geoConvert.getAddressFromLocationName("北京大兴国际机场", 纬度下限, 经度下限, 纬度上限, 经度上限, 1);
```

概述

应用程序可以对系统各类设置项进行增、删、改、查等操作。例如，三方应用提前注册飞行模式设置项的回调，当用户通过系统设置修改终端的飞行模式状态时，三方应用会检测到此设置项发生变化并进行适配。如检测到飞行模式开启，将进入离线状态；检测到飞行模式关闭，其将重新获取在线数据。

基本概念

系统设置数据项分为 TTS（Text To Speech）、Wireless、Network、Input、Sound、Display、Date、Call、General 九类，应用程序可以根据自身拥有的权限对其进行操作。

开发指导

场景介绍

TTS、Wireless、Network、Input、Sound、Display、Date、Call、General 九类定义了表征终端设备状态的相关字段，如屏幕亮度、日期格式、字体显示大小等，应用程序可以根据自身所拥有的权限对其进行增、删、改、查等操作，并进行相应的场景适配。

例如：TIME_FORMAT——表示日期格式，应用程序可进行读写。

图 1　数据表更新过程

接口说明

SystemSettings 提供系统设置的相关接口，包括 TTS、Wireless、Network、Input、Sound、Display、Date、Call、General 九类字段的存储和检索接口。应用程序通过 AppSettings 类提供的方法对其自身的能力进行查询。

表 1 AppSettings 的主要接口

接口名	描述
canShowOverlays(Context context)	检查指定应用程序是否可以显示在其他应用之上。
checkSetPermission(Context context)	通过应用上下文检查指定的应用是否具有修改系统设置的权限。

表 2 SystemSettings 的主要接口

接口名	描述
getUri(String name)	为特定的字段构造 URI，用于 DataAbility 的数据监视。
getValue(DataAbilityHelper dataAbilityHelper, String name)	获取指定字段的值。
setValue(DataAbilityHelper dataAbilityHelper, String name, String value)	设置指定字段的值。

表 3 SystemSettings.TTS 提供的典型字段

字段名	字段描述
DEFAULT_TTS_PITCH	文本转语音引擎的默认音调。
DEFAULT_TTS_RATE	文本转语音引擎的默认语速。

表 4 SystemSettings.Wireless 提供的典型字段

字段名	字段描述
BLUETOOTH_STATUS	蓝牙开启状态。
WIFI_STATUS	WLAN 是否启用。
WIFI_TO_MOBILE_DATA_AWAKE_TIMEOUT	从 WLAN 断开连接后等待建立移动数据连

表 4 SystemSettings.Wireless 提供的典型字段

字段名	字段描述
	接时保持唤醒锁的最长时间。

表 5 SystemSettings.Network 提供的典型字段

字段名	字段描述
DATA_ROAMING_STATUS	数据漫游开启状态。
NETWORK_PREFERENCE_USAGE	设置用户经常使用的网络。

表 6 SystemSettings.Input 提供的典型字段

字段名	字段描述
DEFAULT_INPUT_METHOD	设置默认的输入法，并记录此输入法的 ID。
ACTIVATED_INPUT_METHODS	已激活的输入法列表。
AUTO_CAPS_TEXT_INPUT	设置文本编辑器是否启用自动大写。

表 7 SystemSettings.Sound 提供的字段

字段名	字段描述
HAPTIC_FEEDBACK_STATUS	设置是否开启触摸反馈。
VIBRATE_WHILE_RINGING	设置来电响铃时是否震动。
DEFAULT_NOTIFICATION_SOUND	系统默认通知铃声的存储区。

表 8 SystemSettings.Display 提供的典型字段

字段名	字段描述
FONT_SCALE	设置字体大小因子。

表 8 SystemSettings.Display 提供的典型字段

字段名	字段描述
SCREEN_BRIGHTNESS_STATUS	设置屏幕亮度。
AUTO_SCREEN_BRIGHTNESS	设置是否打开屏幕亮度自动调节模式。
SCREEN_OFF_TIMEOUT	设置设备屏幕自动休眠时间。

表 9 SystemSettings.Date 提供的典型字段

字段名	字段描述
DATE_FORMAT	设置日期格式。
TIME_FORMAT	设置以 12 或 24 小时制显示时间。
AUTO_GAIN_TIME	是否从网络（NITZ）自动获取日期，时间和时区的值。
AUTO_GAIN_TIME_ZONE	是否从网络（NITZ）自动获取时区的值。

表 10 SystemSettings.General 提供的典型字段

字段名	字段描述
SETUP_WIZARD_FINISHED	识别开机向导是否已经运行过。
AIRPLANE_MODE_STATUS	飞行模式是否开启。
DEVICE_NAME	设备名称。
ACCESSIBILITY_STATUS	设置辅助功能是否可用。

表 11 SystemSettings.Call 提供的典型字段

字段名	字段描述
RTT_CALLING_STATUS	设置来去电是否启动 RTT 模式进行应答。

开发步骤

1. 应用程序打开某个 Slice 时，在 OnStart()时，注册相关设置项的回调，并读取一次该设置项的值，进行初始化适配。

```
@Override
public void onStart(Intent intent) {
    // ...
    dataAbilityHelper = DataAbilityHelper.creator(this);
    IDataAbilityObserver dataAbilityObserver = new
IDataAbilityObserver() {
        @Override
        public void onChange() {
            String timeFormat =
SystemSettings.getValue(dataAbilityHelper,
SystemSettings.Date.TIME_FORMAT);
            setTimeFormat(timeFormat);
        }
    };

dataAbilityHelper.registerObserver(SystemSettings.getUri(SystemSettin
gs.Date.TIME_FORMAT), dataAbilityObserver);
}

void setTimeFormat(String timeFormat) {
    if ("12".equals(timeFormat)) {
        // Display in 12-hour format
    } else {
        // Display in 24-hour format
    }
}
```

在 onStop()时，解除回调注册。

```
dataAbilityHelper.unregisterObserver(SystemSettings.getUri(SystemSettin
gs.Date.TIME_FORMAT), dataAbilityObserver);
```

数据管理

关系型数据库

概述

关系型数据库（Relational Database，RDB）是一种基于关系模型来管理数据的数据库。HarmonyOS 关系型数据库基于 SQLite 组件提供了一套完整的对本地数据库进行管理的机制，对外提供了一系列的增、删、改、查接口，也可以直接运行用户输入的 SQL 语句来满足复杂的场景需要。HarmonyOS 提供的关系型数据库功能更加完善，查询效率更高。

基本概念

- **关系型数据库**

创建在关系模型基础上的数据库，以行和列的形式存储数据。

- **谓词**

数据库中用来代表数据实体的性质、特征或者数据实体之间关系的词项，主要用来定义数据库的操作条件。

-
- **结果集**

指用户查询之后的结果集合，可以对数据进行访问。结果集提供了灵活的数据访问方式，可以更方便的拿到用户想要的数据。

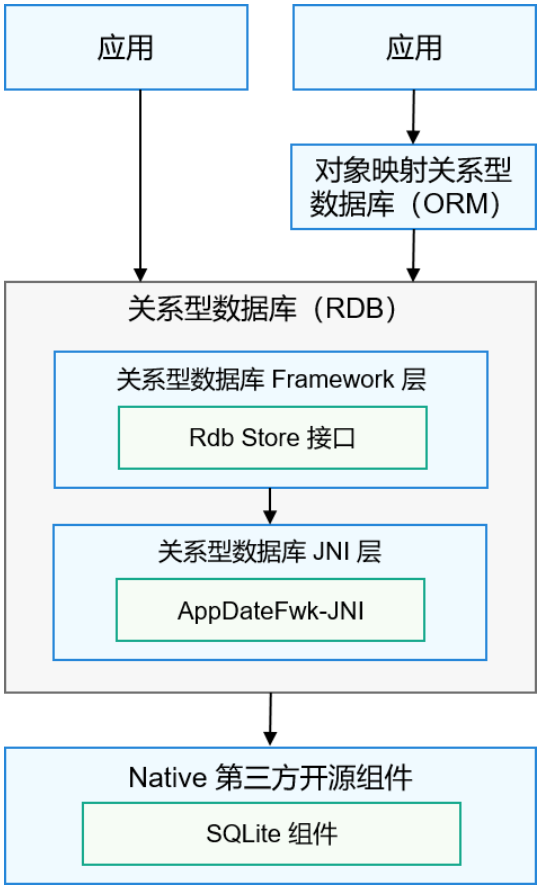
- **SQLite 数据库**

一款轻型的数据库，是遵守 ACID 的关系型数据库管理系统。它是一个开源的项目。

运作机制

HarmonyOS 关系型数据库对外提供通用的操作接口，底层使用 SQLite 作为持久化存储引擎，支持 SQLite 具有的所有数据库特性，包括但不限于事务、索引、视图、触发器、外键、参数化查询和预编译 SQL 语句。

图 1　关系型数据库运作机制



默认配置

- 如果不指定数据库的日志模式，那么系统默认日志方式是 WAL（Write Ahead Log）模式。
- 如果不指定数据库的落盘模式，那么系统默认落盘方式是 FULL 模式。
- HarmonyOS 数据库使用的共享内存默认大小是 2MB。

约束与限制

- 数据库中连接池的最大数量是 4 个，用以管理用户的读写操作。
- 为保证数据的准确性，数据库同一时间只能支持一个写操作。

开发指导

场景介绍

关系型数据库是在 SQLite 基础上实现的本地数据操作机制，提供给用户无需编写原生 SQL 语句就能进行数据增删改查的方法，同时也支持原生 SQL 操作。

接口说明

数据库的创建和删除

关系型数据库提供了数据库创建方式，以及对应的删除接口，涉及的 API 如下所示。

表 1 数据库创建和删除 API

类名	接口名	描述
StoreConfig.Builder	public builder()	对数据库进行配置，包括设置数据库名、存储模式、日志模式、同步模式，是否为只读，及对数据库加密。
RdbOpenCallback	public abstract void onCreate(RdbStore store)	数据库创建时被回调，开发者可以在该方法中初始化表结构，并添加一些应用使用到的初始化数据。
RdbOpenCallback	public abstract void onUpgrade(RdbStore store, int currentVersion, int targetVersion)	数据库升级时被回调。
DatabaseHelper	public RdbStore getRdbStore(StoreConfig config, int version,	根据配置创建或打开数据库。

表 1 数据库创建和删除 API

类名	接口名	描述
	RdbOpenCallback openCallback, ResultSetHook resultSetHook)	
DatabaseHelper	public boolean deleteRdbStore(String name)	删除指定的数据库。

数据库的加密

关系型数据库提供数据库加密的能力，创建数据库时传入指定密钥、创建加密数据库，后续打开加密数据库时，需要传入正确密钥。

表 2 数据库传入密钥接口

类名	接口名	描述
StoreConfig.Builder	Builder setEncryptKey(byte[] encryptKey)	为数据库配置类设置数据库加密密钥，创建或打开数据库时传入包含数据库加密密钥的配置类，即可创建或打开加密数据库。

数据库的增删改查

关系型数据库提供本地数据增删改查操作的能力，相关 API 如下所示。

- **新增**

关系型数据库提供了插入数据的接口，通过 ValuesBucket 输入要存储的数据，通过返回值判断是否插入成功，插入成功时返回最新插入数据所在的行号，失败则返回-1。

表 3 数据库插入 API

类名	接口名	描述
RdbStore	long insert(String table, ValuesBucket initialValues)	向数据库插入数据。 table：待添加数据的表名。 initialValues：以 ValuesBucket 存储的待插入的数据。它提供一系列 put 方法，如 putString(String columnName, String values)，putDouble(String columnName, double value)，用于向 ValuesBucket 中添加数据。

- **更新**

调用更新接口，传入要更新的数据，并通过 AbsRdbPredicates 指定更新条件。该接口的返回值表示更新操作影响的行数。如果更新失败，则返回 0。

表 4 数据库更新 API

类名	接口名	描述
RdbStore	int update(ValuesBucket values, AbsRdbPredicates predicates)	更新数据库表中符合谓词指定条件的数据。 values：以 ValuesBucket 存储的要更新的数据。 predicates：指定了更新操作的表名和条件。AbsRdbPredicates 的实现类有两个：RdbPredicates 和 RawRdbPredidates。 RdbPredicates：支持调用谓词提供的 equalTo 等接口，设置更新条件。 RawRdbPredidates：仅支持设置表名、where 条件子句、whereArgs 三个参数，不支持 equalTo 等接口调用。

- **删除**

调用删除接口，通过 AbsRdbPredicates 指定删除条件。该接口的返回值表示删除的数据行数，可根据此值判断是否删除成功。如果删除失败，则返回 0。

表 5 数据库删除 API

类名	接口名	描述
RdbStore	int delete(AbsRdbPredicates predicates）	删除数据。 predicates：Rdb 谓词，指定了删除操作的表名和条件。AbsRdbPredicates 的实现类有两个：RdbPredicates 和 RawRdbPredidates。 RdbPredicates：支持调用谓词提供的 equalTo 等接口，设置更新条件。 RawRdbPredidates：仅支持设置表名、where 条件子句、whereArgs 三个参数，不支持 equalTo 等接口调用。

- **查询**

关系型数据库提供了两种查询数据的方式：

- 直接调用查询接口。使用该接口，会将包含查询条件的谓词自动拼接成完整的 SQL 语句进行查询操作，无需用户传入原生的 SQL。
- 执行原生的用于查询的 SQL 语句。

表 6 数据库查询 API

类名	接口名	描述
RdbStore	ResultSet query(AbsRdbPredicates predicates, String[] columns)	查询数据。 predicates：谓词，可以设置查询条件。AbsRdbPredicates 的实现类有两个：RdbPredicates 和 RawRdbPredidates。 RdbPredicates：支持调用谓词提供的 equalTo 等接口，设置查询条件。 RawRdbPredidates：仅支持

表 6 数据库查询 API

类名	接口名	描述
		设置表名、where 条件子句、whereArgs 三个参数，不支持 equalTo 等接口调用。 columns：规定查询返回的列。
RdbStore	ResultSet querySql(String sql, String[] sqlArgs)	执行原生的用于查询操作的 SQL 语句。 sql：原生用于查询的 sql 语句。 sqlArgs：sql 语句中占位符参数的值，若 select 语句中没有使用占位符，该参数可以设置为 null。

数据库谓词的使用

关系型数据库提供了用于设置数据库操作条件的谓词 AbsRdbPredicates，其中包括两个实现子类 RdbPredicates 和 RawRdbPredicates：

- RdbPredicates：开发者无需编写复杂的 SQL 语句，仅通过调用该类中条件相关的方法，如 equalTo、notEqualTo、groupBy、orderByAsc、beginsWith 等，就可自动完成 SQL 语句拼接，方便用户聚焦业务操作。
- RawRdbPredicates：可满足复杂 SQL 语句的场景，支持开发者自己设置 where 条件子句和 whereArgs 参数。不支持 equalTo 等条件接口的使用。

表 7 数据库谓词 API

类名	接口名	描述
RdbPredicates	RdbPredicates equalTo(String field, String value)	设置谓词条件，满足 filed 字段与 value 值相等。
RdbPredicates	RdbPredicates notEqualTo(String field, String value)	设置谓词条件，满足 filed 字段与 value 值不相等。
RdbPredicates	RdbPredicates beginsWith(String field, String value)	设置谓词条件，满足 field 字段以 value 值开头。

表 7 数据库谓词 API

类名	接口名	描述
RdbPredicates	RdbPredicates between(String field, int low, int high)	设置谓词条件，满足 field 字段在最小值 low 和最大值 high 之间。
RdbPredicates	RdbPredicates orderByAsc(String field)	设置谓词条件，根据 field 字段升序排列。
RawRdbPredicates	void setWhereClause(String whereClause)	设置 where 条件子句。
RawRdbPredicates	void setWhereArgs(List<String> whereArgs)	设置 whereArgs 参数，该值表示 where 子句中占位符的值。

查询结果集的使用

关系型数据库提供了查询返回的结果集 ResultSet，他指向查询结果中的一行数据，供用户对查询结果进行遍历和访问。ReusltSet 的对外 API 如下表格。

表 8 结果集 API

类名	接口名	描述
ResultSet	boolean goTo(int offset)	从结果集当前位置移动指定偏移量。
ResultSet	boolean goToRow(int position)	将结果集移动到指定位置。
ResultSet	boolean goToNextRow()	将结果集向后移动一行。
ResultSet	boolean goToPreviousRow()	将结果集向前移动一行。
ResultSet	boolean isStarted()	判断结果集是否被移动过。
ResultSet	boolean isEnded()	判断结果集当前位置是否在最后一行之后。

表 8 结果集 API

类名	接口名	描述
ResultSet	boolean isAtFirstRow()	判断结果集当前位置是否在第一行。
ResultSet	boolean isAtLastRow()	判断结果集当前位置是否在最后一行。
ResultSet	int getRowCount()	获取当前结果集中的记录条数。
ResultSet	int getColumnCount()	获取结果集中的列数。
ResultSet	String getString(int columnIndex)	获取当前行指定索列的值，以 String 类型返回。
ResultSet	byte[] getBlob(int columnIndex)	获取当前行指定列的值，以字节数组形式返回。
ResultSet	double getDouble(int columnIndex)	获取当前行指定列的值，以 double 型返回。

事务

关系型数据库提供事务机制，来保证用户操作的原子性。对单条数据进行数据库操作时，无需开启事务；插入大量数据时，开启事务可以保证数据的准确性。如果中途操作出现失败，会执行回滚操作。

表 9 事务 API

类名	接口名	描述
RdbStore	beginTransaction()	开启事务。
RdbStore	markAsCommit()	设置事务的标记为成功。
RdbStore	endTransaction()	结束事务。

事务和结果集观察者

关系型数据库提供了事务和结果集观察者能力，当对应的事件被触发时，观察者会收到通知。

类名	接口名	描述
RdbStore	beginTransactionWithObserver(TransactionObserver transactionObserver)	开启事务，并观察事务的启动、提交和回滚。
ResultSet	void registerObserver(DataObserver observer)	注册结果集的观察者。
ResultSet	void unregisterObserver(DataObserver observer)	注销结果集的观察者。

数据库的备份和恢复

用户可以将当前数据库的数据进行保存进行备份，还可以在需要的时候进行数据恢复。

表 10 数据库备份和恢复

类名	接口名称	描述
RdbStore	boolean restore(String srcName）	数据库恢复接口，从指定的非加密数据库文件中恢复数据。
RdbStore	boolean restore(String srcName, byte[] srcEncryptKey, byte[] destEncryptKey）	数据库恢复接口，从指定的数据库文件（加密和非加密均可）中恢复数据。
RdbStore	boolean backup(String destName)	数据库备份接口，备份出的数据库文件是非加密的。
RdbStore	boolean backup(String destName, byte[] destEncryptKey)	数据库备份接口，此方法经常用在备份出加密数据库场景。

开发步骤

1. 创建数据库。

- 配置数据库相关信息，包括数据库的名称、存储模式、是否为只读模式等。
- 初始化数据库表结构和相关数据。
- 创建数据库。
- 示例代码如下：

```
StoreConfig config = StoreConfig.newDefaultConfig("RdbStoreTest.db");

private static final RdbOpenCallback callback = new RdbOpenCallback() {

    @Override

    public void onCreate(RdbStore store) {

        store.executeSql("CREATE TABLE IF NOT EXISTS test (id INTEGER
PRIMARY KEY AUTOINCREMENT, name TEXT NOT NULL, age INTEGER, salary REAL,
blobType BLOB)");

    }

    @Override

    public void onUpgrade(RdbStore store, int oldVersion, int
newVersion) {

    }

};

DatabaseHelper helper = new DatabaseHelper(context);

RdbStore store = helper.getRdbStore(config, 1, callback, null);
```

插入数据。

a. 构造要插入的数据，以 ValuesBucket 形式存储。
b. 调用关系型数据库提供的插入接口。

示例代码如下:

```
ValuesBucket values = new ValuesBucket();
values.putInteger("id", 1);
values.putString("name", "zhangsan");
values.putInteger("age", 18);
values.putDouble("salary", 100.5);
values.putByteArray("blobType", new byte[] {1, 2, 3});
long id = store.insert("test", values);
```

查询数据。

a. 构造用于查询的谓词对象，设置查询条件。
b. 指定查询返回的数据列。
c. 调用查询接口查询数据。
d. 调用结果集接口，遍历返回结果。

示例代码如下：

```
String[] columns = new String[] {"id", "name", "age", "salary"};
RdbPredicates rdbPredicates = new RdbPredicates("test").equalTo("age", 25).orderByAsc("salary");
ResultSet resultSet = store.query(rdbPredicates, columns);
resultSet.goToNextRow();
```

对象关系映射数据库

概述

HarmonyOS 对象关系映射（Object Relational Mapping，ORM）数据库是一款基于 SQLite 的数据库框架，屏蔽了底层 SQLite 数据库的 SQL 操作，针对实体和关系提供了增删改查等一系列的面向对象接口。应用开发者不必再去编写复杂的 SQL 语句， 以操作对象的形式来操作数据库，提升效率的同时也能聚焦于业务开发。

基本概念

- 对象关系映射数据库的三个主要组件：
- 数据库：被开发者用@Database 注解，且继承了 OrmDatabase 的类，对应关系型数据库。
- 实体对象：被开发者用@Entity 注解，且继承了 OrmObject 的类，对应关系型数据库中的表。
- 对象数据操作接口：包括数据库操作的入口 OrmContext 类和谓词接口（OrmPredicate）等。

谓词

数据库中是用来代表数据实体的性质、特征或者数据实体之间关系的词项，主要用来定义数据库的操作条件。对象关系映射数据库将 SQLite 数据库中的谓词封装成了接口方法供开发者调用。开发者通过对象数据操作接口，可以访问到应用持久化的关系型数据。

- **对象关系映射数据库**

通过将实例对象映射到关系上，实现使用操作实例对象的语法，来操作关系型数据库。它是在 SQLite 数据库的基础上提供的一个抽象层。

- **SQLite 数据库**

一款轻型的数据库，是遵守 ACID 的关系型数据库管理系统。

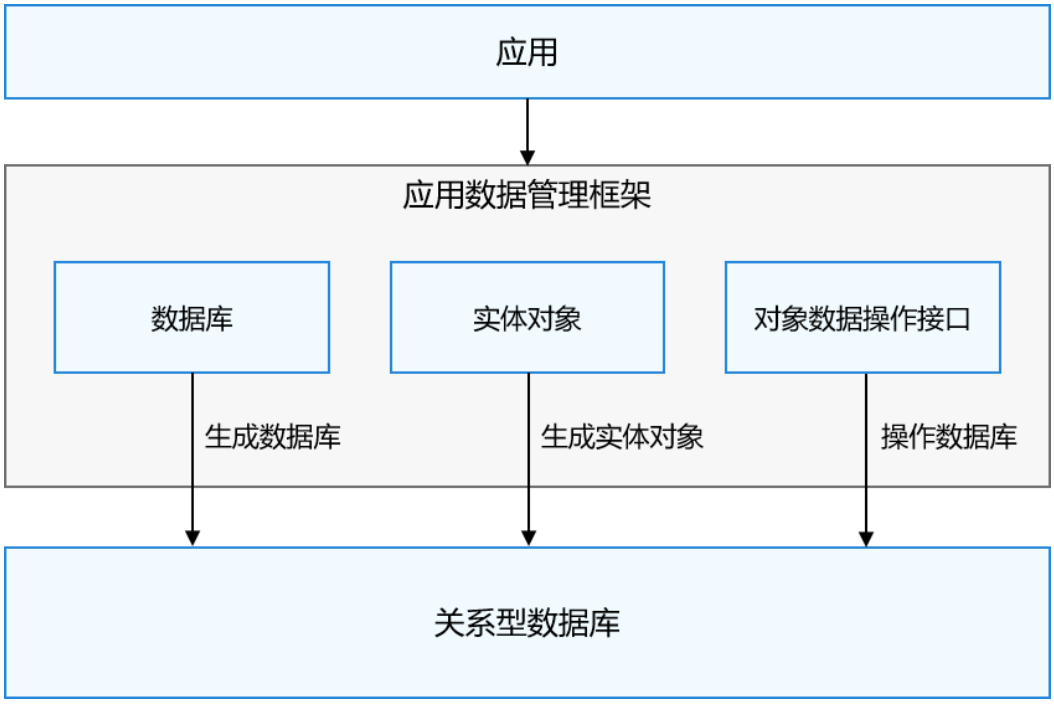
运作机制

对象关系映射数据库操作是基于关系型数据库操作接口完成的，实际是在关系型数据库操作的基础上又实现了对象关系映射等特性。因此对象关系映射数据库跟关系型数据库一样，都使用 SQLite 作为持久化引擎，底层使用的是同一套数据库连接池和数据库连接机制。

使用对象关系映射数据库的开发者需要先配置实体模型与关系映射文件。应用数据管理框架提供的类生成工具会解析这些文件，生成数据库帮助类，这样应用数据管理框架就能在运行时，根据开发者的配置创建好数据库，并在存储过程中自动完成对象关系映射。开发者再通过对象数据操作接口，如 OrmContext 接口和谓词接口等操作持久化数据库。

对象数据操作接口提供一组基于对象映射的数据操作接口，实现了基于 SQL 的关系模型数据到对象的映射，让用户不需要再和复杂的 SQL 语句打交道，只需简单地操作实体对象的属性和方法。对象数据操作接口支持对象的增删改查操作，同时支持事务操作等。

图 1　对象关系映射数据库运作机制



默认配置

- 如果不指定数据库的日志模式，那么系统默认日志方式是 WAL（Write Ahead Log）模式。
- 如果不指定数据库的落盘模式，那么系统默认落盘方式是 FULL 模式。
- HarmonyOS 数据库使用的共享内存默认大小是 2MB。

约束与限制

HarmonyOS 对象关系映射数据库是建立在 HarmonyOS 关系型数据库的基础之上的，所以关系型数据库的一些约束与限制请参考约束与限制。

此外当开发者建立实体对象类时，对象属性的类型可以在下表的类型中选择。不支持使用自定义类型。

表 1 实体对象属性支持的类型

类型名称	描述	初始值
Integer	封装整型	null
int	整型	0
Long	封装长整型	null
long	长整型	0L
Double	封装双精度浮点型	null
double	双精度浮点型	0
Float	封装单精度浮点型	null
float	单精度浮点型	0
Short	封装短整型	null
short	短整型	0
String	字符串型	null

表 1 实体对象属性支持的类型

类型名称	描述	初始值
Boolean	封装布尔型	null
boolean	布尔型	0
Byte	封装字节型	null
byte	字节型	0
Character	封装字符型	null
char	字符型	' '
Date	日期类	null
Time	时间类	null
Timestamp	时间戳类	null
Calendar	日历类	null
Blob	二进制大对象	null
Clob	字符大对象	null

开发指导

场景介绍

对象关系映射数据库适用于开发者使用的数据可以分解为一个或多个对象，且需要对数据进行增删改查等操作，但是不希望编写过于复杂的 SQL 语句的场景。

该对象关系映射数据库的实现是基于关系型数据库，除了数据库版本升降级等场景外，操作对象关系映射数据库一般不需要编写 SQL 语句，但是仍然要求使用者对于关系型数据库的基本概念有一定的了解。

开发能力介绍

对象关系映射数据库目前可以支持数据库和表的创建，对象数据的增删改查、对象数据变化回调、数据库升降级和备份等功能。

数据库和表的创建

1. 创建数据库。开发者需要定义一个表示数据库的类，继承 OrmDatabase，再通过@Database 注解内的 entities 属性指定哪些数据模型类属于这个数据库。

属性：

- version：数据库版本号。
- entities：数据库内包含的表。

2. 创建数据表。开发者可通过创建一个继承了 OrmObject 并用@Entity 注解的类，获取数据库实体对象，也就是表的对象。

属性：

- tableName：表名。
- primaryKeys：主键名，一个表里只能有一个主键，一个主键可以由多个字段组成。
- foreignKeys：外键列表。
- indices：索引列表。
-

表 1 注解对照表

接口名称	描述
@Database	被@Database 注解且继承了 OrmDatabase 的类对应数据库类。
@Entity	被@Entity 注解且继承了 OrmObject 的类对应数据表类。
@Column	被@Column 注解的变量对应数据表的字段。
@PrimaryKey	被@PrimaryKey 注解的变量对应数据表的主键。
@ForeignKey	被@ForeignKey 注解的变量对应数据表的外键。
@Index	被@Index 注解的内容对应数据表索引的属性。

数据库的加密

对象关系映射数据库提供数据库加密的能力，创建数据库时传入指定密钥、创建加密数据库，后续打开加密数据库时，需要传入正确密钥。

表 2 数据库传入密钥接口

类名	接口名	描述
OrmConfig.Builder	Builder setEncryptKey(byte[] encryptKey)	为数据库配置类设置数据库加密密钥，创建或打开数据库时传入包含数据库加密密钥的配置类，即可创建或打开加密数据库。

对象数据的增删改查

通过对象数据操作接口，开发者可以对对象数据进行增删改查操作。

表 3 对象数据操作接口

类名	接口名称	描述
OrmContext	<T extends OrmObject> boolean insert(T object)	添加方法。
OrmContext	<T extends OrmObject> boolean update(T object)	更新方法。
OrmContext	<T extends OrmObject> List<T> query(OrmPredicates predicates)	查询方法。
OrmContext	<T extends OrmObject> boolean delete(T object)	删除方法。
OrmContext	<T extends OrmObject> OrmPredicates where(Class<T> clz)	设置谓词方法。

对象数据的变化观察者设置

通过使用对象数据操作接口，开发者可以在某些数据上设置观察者，接收数据变化的通知。

表 4 数据变化观察者接口

类名	接口名称	描述
OrmContext	void registerStoreObserver(String alias, OrmObjectObserver observer)	注册数据库变化回调。
OrmContext	void registerContextObserver(OrmContext watchedContext, OrmObjectObserver observer)	注册上下文变化回调。
OrmContext	void registerEntityObserver(String entityName, OrmObjectObserver observer)	注册数据库实体变化回调。

表 4 数据变化观察者接口

类名	接口名称	描述
OrmContext	void registerObjectObserver(OrmObject ormObject, OrmObjectObserver observer)	注册对象变化回调。

数据库的升降级

通过调用数据库升降级接口，开发者可以将数据库切换到不同的版本。

表 5 数据库升降级接口

类名	接口名称	描述
OrmMigration	public void onMigrate(int beginVersion, int endVersion)	数据库版本升降级接口。

数据库的备份恢复

开发者可以将当前数据库的数据进行备份，在必要的时候进行数据恢复。

表 6 数据库备份与恢复接口

类名	接口名称	描述
OrmContext	boolean backup(String destPath)	数据库备份接口。
OrmContext	boolean restore(String srcPath);	数据库恢复备份接口。

开发步骤

配置“build.gradle”文件。

- 如果使用注解处理器的模块为“com.huawei.ohos.hap”模块，则需要在模块的“build.gradle”文件的“ohos”节点中添加以下配置：

```
compileOptions{
    annotationEnabled true
}
```

如果使用注解处理器的模块为“com.huawei.ohos.library”模块，则需要在模块的“build.gradle”文件的“dependencies”节点中配置注解处理器。查看“orm_annotations_java.jar”、“orm_annotations_processor_java.jar”、“javapoet_java.jar”这 3 个 jar 包在 HUAWEI SDK 中的对应目录，并将目录的这三个 jar 包导进来。

```
dependencies {
    compile files("orm_annotations_java.jar 的路径
","orm_annotations_processor_java.jar 的路径","javapoet_java.jar 的路径")
    annotationProcessor files("orm_annotations_java.jar 的路径
","orm_annotations_processor_java.jar 的路径","javapoet_java.jar 的路径")
}
```

如果使用注解处理器的模块为“java-library”模块，则需要在模块的“build.gradle”文件的“dependencies”节点中配置注解处理器，并导入“ohos.jar”。

```
dependencies {
    compile files("ohos.jar 的路径","orm_annotations_java.jar 的路径
","orm_annotations_processor_java.jar 的路径","javapoet_java.jar 的路径")
    annotationProcessor files("orm_annotations_java.jar 的路径
","orm_annotations_processor_java.jar 的路径","javapoet_java.jar 的路径")
}
```

构造数据库，即创建数据库类并配置对应的属性。

例如，定义了一个数据库类 BookStore.java，数据库包含了“User”，"Book"，"AllDataType"三个表，版本号为 “1” 。数据库类的 getVersion 方法和 getHelper 方法不需要实现，直接将数据库类设为虚类即可。

```
@Database(entities = {User.class, Book.class, AllDataType.class}, version = 1)
public abstract class BookStore extends OrmDatabase {
}
```

构造数据表,即创建数据库实体类并配置对应的属性(如对应表的主键,外键等)。数据表必须与其所在的数据库在同一个模块中。

例如，定义了一个实体类 User.java，对应数据库内的表名为“user”；indices 为“firstName”和“lastName”两个字段建立了复合索引“name_index”，并且索引值是唯一的；“ignoreColumns”表示该字段不需要添加到“user”表的属性中。

```
@Entity(tableName = "user", ignoredColumns = {"ignoreColumn1", "ignoreColumn2"},
    indices = {@Index(value = {"firstName", "lastName"}, name = "name_index", unique = true)})
public class User extends OrmObject {
    // 此处将 userId 设为了自增的主键。注意只有在数据类型为包装类型时，自增主键才能生效。
    @PrimaryKey(autoGenerate = true)
    private Integer userId;
    private String firstName;
    private String lastName;
    private int age;
```

```
    private double balance;
    private int ignoreColumn1;
    private int ignoreColumn2;

    // 开发者自行添加字段的 getter 和 setter 方法。
}
```

说明

示例中的 getter & setter 的方法名为小驼峰格式，除了手写方法，IDE 中包含自动生成 getter 和 setter 方法的 Generate 插件。

- 当变量名的格式类似“firstName”时，getter 和 setter 方法名应为“getFirstName”和“setFirstName”。
- 当变量名的格式类似“mAge”，即第一个字母小写，第二个字母大写的格式时，getter 和 setter 方法名应为“getmAge”和“setmAge”。
- 当变量名格式类似“x”，即只有一个字母时，getter 和 setter 方法名应为“getX”和“setX”。

变量为 boolean 类型时，上述规则仍然成立，即“isFirstName”，“ismAge”，“isX”。

使用对象数据操作接口 OrmContext 创建数据库。

例如，通过对象数据操作接口 OrmContext，创建一个别名为“BookStore”，数据库文件名为“BookStore.db”的数据库。如果数据库已经存在，执行以下代码不会重复创建。通过 context.getDatabaseDir()可以获取创建的数据库文件所在的目录。

```
DatabaseHelper helper = new DatabaseHelper(context); // context 入参类型为 ohos.app.Context，注意不要使用 slice.getContext()来获取 context，请直接传入 slice，否则会出现找不到类的报错。
```

```
OrmContext context = helper.getOrmContext("BookStore", "BookStore.db",
BookStore.class);
```

（可选）数据库升降级。如果开发者有多个版本的数据库，通过设置数据库版本迁移类可以实现数据库版本升降级。

数据库版本升降级的调用示例如下。其中 BookStoreUpgrade 类也是一个继承了 OrmDatabase 的数据库类，与 BookStore 类的区别在于配置的版本号不同。

```
OrmContext context = helper.getOrmContext("BookStore", "BookStore.db",
BookStoreUpgrade.class, new TestOrmMigration32(), new
TestOrmMigration23(), new TestOrmMigration12(), new
TestOrmMigration21());
```

TestOrmMigration12 的实现示例如下：

```
private static class TestOrmMigration12 extends OrmMigration {

    // 此处用于配置数据库版本迁移的开始版本和结束版本，super(startVersion,
endVersion)即数据库版本号从 1 升到 2。

    public TestOrmMigration12() {super(1, 2); }

    @Override

    public void onMigrate(RdbStore store) {

        store.executeSql("ALTER TABLE `Book` ADD COLUMN `addColumn12`
INTEGER");

    }

}
```

说明

数据库版本迁移类的起始版本和结束版本必须是连续的。

- 如果 BookStoreUpgrade 类的版本号配置为“2”，而当前 BookStore.db 的实际版本号为“1”时，TestOrmMigration12 类的 onMigrate 方法会自动被调用。开发者可在 onMigrate 方法中填写升级需要执行的 sql 语句。

- 如果 BookStoreUpgrade 的类版本号配置为“3”，而当前 BookStore.db 的实际版本号为“1”时，TestOrmMigration12，TestOrmMigration23 的 onMigrate 方法会自动被调用完成数据库升级。数据库版本降级同理。

使用对象数据操作接口 OrmContext 对数据库进行增删改查、注册观察者、备份数据库等。

- 增加数据。例如，在数据库的名为“user”的表中，新建一个 User 对象并设置对象的属性。直接传入 OrmObject 对象的增加接口，只有在 flush()接口被调用后才会持久化到数据库中。

```
// 更新数据
OrmPredicates predicates = context.where(User.class);
predicates.equalTo("age",29);
List<User> users = context.query(predicates);
User user = users.get(0);
user.setFirstName("Li");
context.update(user);
context.flush();

// 删除数据
OrmPredicates predicates = context.where(User.class);
predicates.equalTo("age",29);
List<User> users = context.query(predicates);
```

```
User user = users.get(0);
context.delete(user);
context.flush();
```

通过传入谓词的接口来更新和删除数据，方法与 OrmObject 对象的接口类似，只是无需 flush 就可以持久化到数据库中。

```
ValuesBucket valuesBucket = new ValuesBucket();
valuesBucket.putInteger("age", 31);
valuesBucket.putString("firstName", "ZhangU");
valuesBucket.putString("lastName", "SanU");
valuesBucket.putDouble("balance", 300.51);
OrmPredicates update = context.where(User.class).equalTo("userId", 1);
context.update(update, valuesBucket);
```

查询数据。在数据库的“user”表中查询 lastName 为“San”的 User 对象列表，示例如下：

```
OrmPredicates query = context.where(User.class).equalTo("lastName",
"San");
List<User> users = context.query(query);
```

注册观察者。

```
// 定义一个观察者类。
private class MyOrmObjectObserver implements OrmObjectObserver {
    @Override
    public void onChange(OrmContext changeContext,AllChangeToTarget
subAllChange {
```

```
        // 用户可以在此处定义观察者行为
    }
}


// 调用 registerEntityObserver 方法注册一个观察者 observer。
MyOrmObjectObserver observer = new MyOrmObjectObserver();
context.registerEntityObserver("user", observer);


// 当以下方法被调用，并 flush 成功时，观察者 observer 的 onChange 方法会被触发。其中，方法的入参必须为 User 类的对象。
public <T extends OrmObject> boolean insert(T object)
public <T extends OrmObject> boolean update(T object)
public <T extends OrmObject> boolean delete(T object)
```

备份数据库。其中原数据库名为“OrmBackUp.db”，备份数据库名为“OrmBackup001.db”。

```
OrmContext context = helper.getObjectContext("OrmBackup", "OrmBackup.db", BookStore.class);
context.backup("OrmBackup001.db");
context.close();
```

删除数据库，例如删除 OrmBackup.db。

```
helper.deleteRdbStore("OrmBackup.db");
```

轻量级偏好数据库

概述

轻量级偏好数据库主要提供轻量级 Key-Value 操作，支持本地应用存储少量数据，数据存储在本地文件中，同时也加载在内存中的，所以访问速度更快，效率更高。轻量级偏好数据库属于非关系型数据库，不宜存储大量数据，经常用于操作键值对形式数据的场景。

基本概念

- **Key-Value 数据库**

一种以键值对存储数据的一种数据库，类似 Java 中的 map。Key 是关键字，Value 是值。

- **非关系型数据库**

区别于关系数据库，不保证遵循 ACID（Atomic、Consistency、Isolation 及 Durability）特性，不采用关系模型来组织数据，数据之间无关系，扩展性好。

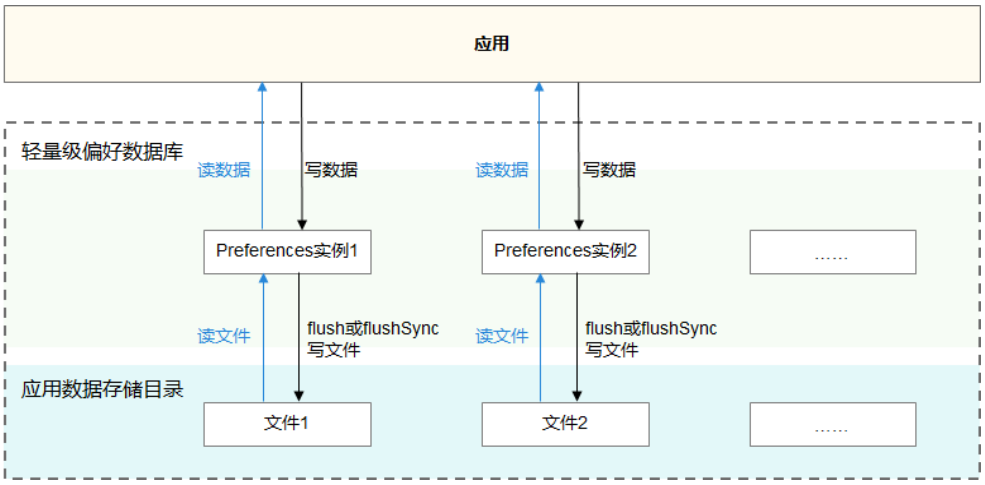
- **偏好数据**

用户经常访问和使用的数据。

运作机制

1. 本模块提供偏好型数据库的操作类，应用通过这些操作类完成数据库操作。
2. 借助 DatabaseHelper API，应用可以将指定文件的内容加载到 Preferences 实例，每个文件最多有一个 Preferences 实例，系统会通过静态容器将该实例存储在内存中，直到应用主动从内存中移除该实例或者删除该文件。
3. 获取到文件对应的 Preferences 实例后，应用可以借助 Preferences API，从 Preferences 实例中读取数据或者将数据写入 Preferences 实例，通过 flush 或者 flushSync 将 Preferences 实例持久化。

图 1　轻量级偏好数据库运作机制



约束与限制

- Key 键为 String 类型，要求非空且大小不超过 80 个字符。
- 如果 Value 值为 String 类型，可以为空但是长度不超过 8192 个字符。
- 存储的数据量应该是轻量级的，建议存储的数据不超过一万条，否则会在内存方面产生较大的开销。

开发指导

场景介绍

轻量级偏好数据库是轻量级存储，主要用于保存应用的一些常用配置，并不适合存储大量数据和频繁改变数据的场景。用户的数据保存在文件中，可以持久化的存储在设备上。需要注意的是用户访问的实例包含文件所有数据，并一直加载在设备的内存中，并通过轻量级偏好数据库的 API 完成数据操作。

接口说明

轻量级偏好数据库向本地应用提供了操作偏好型数据库的 API，支持本地应用读写少量数据及观察数据变化。数据存储形式为键值对，键的类型为字符串型，值的存储数据类型包括整型、字符串型、布尔型、浮点型、长整型、字符串型 Set 集合。

创建数据库

通过数据库操作的辅助类可以获取到要操作的 Preferences 实例，用于进行数据库的操作。

表 1 轻量级偏好数据库创建接口

类名	接口名	描述
DatabaseHelper	Preferences getPreferences(String name)	获取文件对应的 Preferences 单实例，用于数据操作。

查询数据

通过调用 Get 系列的方法，可以查询不同类型的数据。

表 2 轻量级偏好数据库查询接口

类名	接口名	描述
Preferences	int getInt(String key, int defValue)	获取键对应的 int 类型的值。
Preferences	float getFloat(String key, float defValue)	获取键对应的 float 类型的值。

插入数据

通过 Put 系列的方法可以修改 Preferences 实例中的数据，通过 flush 或者 flushSync 将 Preferences 实例持久化。

表 3 轻量级偏好数据库插入接口

类名	接口名	描述
Preferences	Preferences putInt(String key, int value)	设置 Preferences 实例中键对应的 int 类型的值。
Preferences	Preferences putString(String key, String value)	设置 Preferences 实例中键对应的 String 类型的值。
Preferences	void flush()	将 Preferences 实例异步写入文件。
Preferences	boolean flushSync()	将 Preferences 实例同步写入文件。

观察数据变化

轻量级偏好数据库还提供了一系列的接口变化回调，用于观察数据的变化。开发者可以通过重写 onChange 方法来定义观察者的行为。

表 4 轻量级偏好数据库接口变化回调

类名	接口名	描述
Preferences	void registerObserver(PreferencesObserver preferencesObserver)	注册观察者，用于观察数据变化。
Preferences	void unRegisterObserver(PreferencesObserver preferencesObserver)	注销观察者。
Preferences.PreferencesObserver	void onChange(Preferences preferences, String key)	观察者的回调方法，任意数据变化都会回调该方法。

删除数据文件

通过调用以下两种接口，可以删除数据文件。

表 5 轻量级偏好数据库删除接口

类名	接口名	描述
DatabaseHelper	boolean deletePreferences(String name)	删除文件和文件对应的 Preferences 单实例。

表 5 轻量级偏好数据库删除接口

类名	接口名	描述
DatabaseHelper	void removePreferencesFromCache(String name)	删除文件对应的 Preferences 单实例。

移动数据库文件

表 6 轻量级偏好数据库移动接口

类名	接口名	描述
DatabaseHelper	boolean movePreferences(Context sourceContext, String sourceName, String targetName)	移动数据库文件。

开发步骤

1. 准备工作，导入对轻量级偏好数据库 SDK 到开发环境。
2. 获取 Preferences 实例。
3. 读取指定文件，将数据加载到 Preferences 实例，用于数据操作。

```
DatabaseHelper databaseHelper = new DatabaseHelper(context); // context 入参类型为 ohos.app.Context。

String fileName = "name"; // fileName 表示文件名，其取值不能为空，也不能包含路径，默认存储目录可以通过 context.getPreferencesDir()获取。

Preferences preferences = databaseHelper.getPreferences(fileName);
```

从指定文件读取数据。

首先获取指定文件对应的 Preferences 实例，然后借助 Preferences API 读取数据。

java 接口 读取整型数据

```
int value = preferences.getInt("intKey", 0);
```

将数据写入指定文件。

首先获取指定文件对应的 Preferences 实例,然后借助 Preferences API 将数据写入 Preferences 实例,通过 flush 或者 flushSync 将 Preferences 实例持久化。

异步:

```
preferences.putInt("intKey", 3);
preferences.putString("StringKey", "String value");
preferences.flush();
```

同步:

```
preferences.putInt("intKey", 3);
preferences.putString("StringKey", "String value");
preferences.flushSync();
```

注册观察者。

开发者可以向 Preferences 实例注册观察者，观察者对象需实现 Preferences.PreferencesObserver 接口。flushSync()或 flush()执行后，该 Preferences 实例注册的所有观察者的 onChange()方法都会被回调。不再需要观察者时请注销。

```
private class PreferencesChangeCounter implements
Preferences.PreferencesObserver {
```

```
    final AtomicInteger notifyTimes = new AtomicInteger(0);
    @Override
    public void onChange(Preferences preferences, String key) {
        if ("intKey".equals(key)) {
            notifyTimes.incrementAndGet();
        }
    }
}
// 向 preferences 实例注册观察者
PreferencesChangeCounter counter = new PreferencesChangeCounter();
preferences.registerObserver(counter);
// 修改数据 preferences.putInt("intKey", 3);
boolean result = preferences.flushSync();
// 修改数据后，onChange 方法会被回调，notifyTimes == 1
int notifyTimes = counter.notifyTimes.intValue();
// 向 preferences 实例注销观察者
preferences.unRegisterObserver(counter);
```

移除 Preferences 实例。

从内存中移除指定文件对应的 Preferences 单实例。移除 Preferences 单实例时，应用不允许再使用该实例进行数据操作，否则会出现数据一致性问题。

```
DatabaseHelper databaseHelper = new DatabaseHelper(context);
String fileName = "name"; // fileName 表示文件名，其取值不能为空，也不能包含路径。
databaseHelper.removePreferencesFromCache(fileName);
```

删除指定文件。

从内存中移除指定文件对应的 Preferences 单实例,并删除指定文件及其备份文件、损坏文件。删除指定文件时，应用不允许再使用该实例进行数据操作，否则会出现数据一致性问题

```
DatabaseHelper databaseHelper = new DatabaseHelper(context);

String fileName = "name"; // fileName 表示文件名，其取值不能为空，也不能包含路径。

boolean result = databaseHelper.deletePreferences(fileName);
```

移动指定文件。

从源路径移动文件到目标路径。移动文件时，应用不允许再操作该文件数据，否则会出现数据一致性问题。

```
Context targetContext = XXX;

DatabaseHelper databaseHelper = new DatabaseHelper(targetContext);

String srcFile = "srcFile"; // srcFile 表示源文件名或者源文件的绝对路径,不能为相对路径,其取值不能为空。当 srcFile 只传入文件名时,srcContext 不能为空。

String targetFile = "targetFile"; // targetFile 表示目标文件名，其取值不能为空，也不能包含路径。

Context srcContext = XXX;

boolean result =
databaseHelper.movePreferences(srcContext,srcFile,targetFile);
```

分布式数据库服务

概述

轻量级偏好数据库主要提供轻量级 Key-Value 操作，支持本地应用存储少量数据，数据存储在本地文件中，同时也加载在内存中的，所以访问速度更快，效率更高。轻量级偏好数据库属于非关系型数据库，不宜存储大量数据，经常用于操作键值对形式数据的场景。

基本概念

- **Key-Value 数据库**

一种以键值对存储数据的一种数据库，类似 Java 中的 map。Key 是关键字，Value 是值。

- **非关系型数据库**

区别于关系数据库，不保证遵循 ACID（Atomic、Consistency、Isolation 及 Durability）特性，不采用关系模型来组织数据，数据之间无关系，扩展性好。

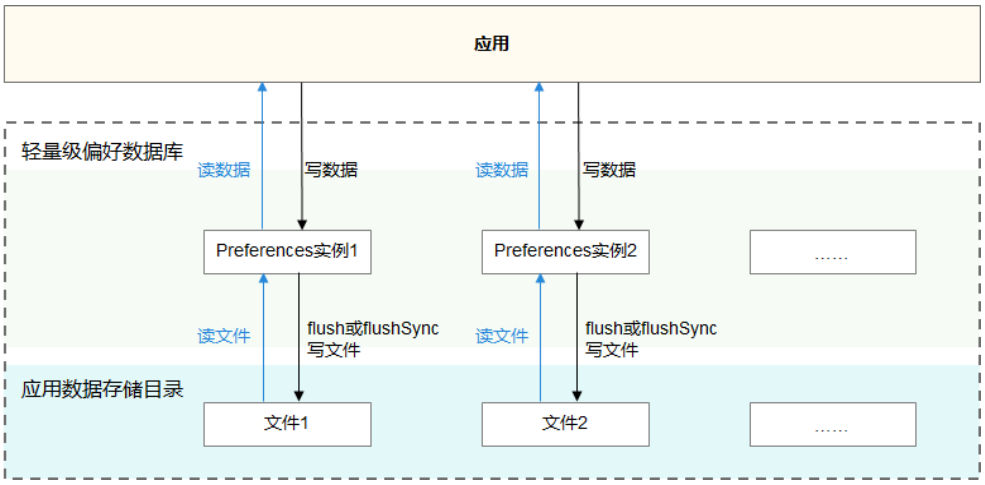
- **偏好数据**

用户经常访问和使用的数据。

运作机制

1. 本模块提供偏好型数据库的操作类，应用通过这些操作类完成数据库操作。
2. 借助 DatabaseHelper API，应用可以将指定文件的内容加载到 Preferences 实例，每个文件最多有一个 Preferences 实例，系统会通过静态容器将该实例存储在内存中，直到应用主动从内存中移除该实例或者删除该文件。
3. 获取到文件对应的 Preferences 实例后，应用可以借助 Preferences API，从 Preferences 实例中读取数据或者将数据写入 Preferences 实例，通过 flush 或者 flushSync 将 Preferences 实例持久化。

图 1　轻量级偏好数据库运作机制



约束与限制

- Key 键为 String 类型，要求非空且大小不超过 80 个字符。
- 如果 Value 值为 String 类型，可以为空但是长度不超过 8192 个字符。
- 存储的数据量应该是轻量级的，建议存储的数据不超过一万条，否则会在内存方面产生较大的开销。

开发指导

场景介绍

轻量级偏好数据库是轻量级存储，主要用于保存应用的一些常用配置，并不适合存储大量数据和频繁改变数据的场景。用户的数据保存在文件中，可以持久化的存储在设备上。需要注意的是用户访问的实例包含文件所有数据，并一直加载在设备的内存中，并通过轻量级偏好数据库的 API 完成数据操作。

接口说明

轻量级偏好数据库向本地应用提供了操作偏好型数据库的 API，支持本地应用读写少量数据及观察数据变化。数据存储形式为键值对，键的类型为字符串型，值的存储数据类型包括整型、字符串型、布尔型、浮点型、长整型、字符串型 Set 集合。

创建数据库

通过数据库操作的辅助类可以获取到要操作的 Preferences 实例，用于进行数据库的操作。

表 1 轻量级偏好数据库创建接口

类名	接口名	描述
DatabaseHelper	Preferences getPreferences(String name)	获取文件对应的 Preferences 单实例，用于数据操作。

查询数据

通过调用 Get 系列的方法，可以查询不同类型的数据。

表 2 轻量级偏好数据库查询接口

类名	接口名	描述
Preferences	int getInt(String key, int defValue)	获取键对应的 int 类型的值。
Preferences	float getFloat(String key, float defValue)	获取键对应的 float 类型的值。

插入数据

通过 Put 系列的方法可以修改 Preferences 实例中的数据，通过 flush 或者 flushSync 将 Preferences 实例持久化。

表 3 轻量级偏好数据库插入接口

类名	接口名	描述
Preferences	Preferences putInt(String key, int value)	设置 Preferences 实例中键对应的 int 类型的值。
Preferences	Preferences putString(String key, String value)	设置 Preferences 实例中键对应的 String 类型的值。
Preferences	void flush()	将 Preferences 实例异步写入文件。
Preferences	boolean flushSync()	将 Preferences 实例同步写入文件。

观察数据变化

轻量级偏好数据库还提供了一系列的接口变化回调，用于观察数据的变化。开发者可以通过重写 onChange 方法来定义观察者的行为。

表 4 轻量级偏好数据库接口变化回调

类名	接口名	描述
Preferences	void registerObserver(PreferencesObserver preferencesObserver)	注册观察者，用于观察数据变化。
Preferences	void unRegisterObserver(PreferencesObserver preferencesObserver)	注销观察者。
Preferences.PreferencesObserver	void onChange(Preferences preferences, String key)	观察者的回调方法，任意数据变化都会回调该方法。

删除数据文件

通过调用以下两种接口，可以删除数据文件。

表 5 轻量级偏好数据库删除接口

类名	接口名	描述
DatabaseHelper	boolean deletePreferences(String name)	删除文件和文件对应的 Preferences 单实例。

表 5 轻量级偏好数据库删除接口

类名	接口名	描述
DatabaseHelper	void removePreferencesFromCache(String name)	删除文件对应的 Preferences 单实例。

移动数据库文件

表 6 轻量级偏好数据库移动接口

类名	接口名	描述
DatabaseHelper	boolean movePreferences(Context sourceContext, String sourceName, String targetName)	移动数据库文件。

开发步骤

1. 准备工作，导入对轻量级偏好数据库 SDK 到开发环境。
2. 获取 Preferences 实例。
3. 读取指定文件，将数据加载到 Preferences 实例，用于数据操作。

```
DatabaseHelper databaseHelper = new DatabaseHelper(context); // context 入参类型为 ohos.app.Context。

String fileName = "name"; // fileName 表示文件名，其取值不能为空，也不能包含路径，默认存储目录可以通过 context.getPreferencesDir()获取。

Preferences preferences = databaseHelper.getPreferences(fileName);
```

从指定文件读取数据。

首先获取指定文件对应的 Preferences 实例，然后借助 Preferences API 读取数据。

java 接口 读取整型数据

```
int value = preferences.getInt("intKey", 0);
```

将数据写入指定文件。

首先获取指定文件对应的 Preferences 实例,然后借助 Preferences API 将数据写入 Preferences 实例,通过 flush 或者 flushSync 将 Preferences 实例持久化。

异步:

```
preferences.putInt("intKey", 3);
preferences.putString("StringKey", "String value");
preferences.flush();
```

同步:

```
preferences.putInt("intKey", 3);
preferences.putString("StringKey", "String value");
preferences.flushSync();
```

注册观察者。

开发者可以向 Preferences 实例注册观察者，观察者对象需实现 Preferences.PreferencesObserver 接口。flushSync()或 flush()执行后，该 Preferences 实例注册的所有观察者的 onChange()方法都会被回调。不再需要观察者时请注销。

```
private class PreferencesChangeCounter implements
Preferences.PreferencesObserver {
```

```
    final AtomicInteger notifyTimes = new AtomicInteger(0);
    @Override
    public void onChange(Preferences preferences, String key) {
        if ("intKey".equals(key)) {
            notifyTimes.incrementAndGet();
        }
    }
}
// 向 preferences 实例注册观察者
PreferencesChangeCounter counter = new PreferencesChangeCounter();
preferences.registerObserver(counter);
// 修改数据 preferences.putInt("intKey", 3);
boolean result = preferences.flushSync();
// 修改数据后，onChange 方法会被回调，notifyTimes == 1
int notifyTimes = counter.notifyTimes.intValue();
// 向 preferences 实例注销观察者
preferences.unRegisterObserver(counter);
```

移除 Preferences 实例。

从内存中移除指定文件对应的 Preferences 单实例。移除 Preferences 单实例时，应用不允许再使用该实例进行数据操作，否则会出现数据一致性问题。

```
DatabaseHelper databaseHelper = new DatabaseHelper(context);
String fileName = "name"; // fileName 表示文件名，其取值不能为空，也不能包含路径。
databaseHelper.removePreferencesFromCache(fileName);
```

删除指定文件。

从内存中移除指定文件对应的 Preferences 单实例,并删除指定文件及其备份文件、损坏文件。删除指定文件时，应用不允许再使用该实例进行数据操作，否则会出现数据一致性问题

```
DatabaseHelper databaseHelper = new DatabaseHelper(context);

String fileName = "name"; // fileName 表示文件名，其取值不能为空，也不能包含路径。

boolean result = databaseHelper.deletePreferences(fileName);
```

移动指定文件。

从源路径移动文件到目标路径。移动文件时，应用不允许再操作该文件数据，否则会出现数据一致性问题。

```
Context targetContext = XXX;

DatabaseHelper databaseHelper = new DatabaseHelper(targetContext);

String srcFile = "srcFile"; // srcFile 表示源文件名或者源文件的绝对路径，不能为相对路径，其取值不能为空。当 srcFile 只传入文件名时，srcContext 不能为空。

String targetFile = "targetFile"; // targetFile 表示目标文件名，其取值不能为空，也不能包含路径。

Context srcContext = XXX;

boolean result =
databaseHelper.movePreferences(srcContext,srcFile,targetFile);
```

分布式数据服务

概述

分布式数据服务（Distributed Data Service，DDS） 为应用程序提供不同设备间数据库数据分布式的能力。通过调用分布式数据接口，应用程序将数据保存到分布式数据库中。通过结合帐号、应用和数据库三元组，分布式数据服务对属于不同的应用的数据进行隔离，保证不同应用之间的数据不能通过分布式数据服务互相访问。在通过可信认证的设备间，分布式数据服务支持应用数据相互同步，为用户提供在多种终端设备上一致的数据访问体验。

基本概念

- **KV 数据模型**

“KV 数据模型”是“Key-Value 数据模型”的简称，“Key-Value”即“键-值”。它是一种 NoSQL 类型数据库，其数据以键值对的形式进行组织、索引和存储。

KV 数据模型适合不涉及过多数据关系和业务关系的业务数据存储，比 SQL 数据库存储拥有更好的读写性能，同时因在分布式场景中降低了数据库版本兼容和数据同步过程中冲突解决的复杂度而被广泛使用。分布式数据库也是基于 KV 数据模型，对外提供 KV 类型的访问接口。

- **分布式数据库事务性**

分布式数据库事务支持本地事务（和传统数据库的事务概念一致）和同步事务，同步事务是指在设备之间同步数据时，是以本地事务为单位进行同步，一次本地事务的修改要么都同步成功，要么都同步失败。

- **分布式数据库一致性**

在分布式场景中一般会涉及多个设备，组网内设备之间看到的数据是否一致称为分布式数据库的一致性。分布式数据库一致性可以分为**强一致性**、**弱一致性**和**最终一致性**。

- **强一致性**：是指某一设备成功增、删、改数据后，组网内设备对该数据的读取操作都将得到更新后的值。

- **弱一致性**：是指某一设备成功增、删、改数据后，组网内设备可能能读取到本次更新数据，也可能读取不到，不能保证在多长时间后每个设备的数据一定是一致的。

- **最终一致性**：是指某一设备成功增、删、改数据后，组网内设备可能读取不到本次更新数据，但在某个时间窗口之后组网内设备的数据能够达到一致状态。强一致性对分布式数据的管理要求非常高，在服务器的分布式场景可能会遇到。因为移动终端设备的不常在线、以及无中心的特性，分布式数据服务不支持强一致，只支持最终一致性。

- **分布式数据库同步**

底层通信组件完成设备发现和认证，会通知上层应用程序（包括分布式数据服务）设备上线。收到设备上线的消息后分布式数据服务可以在两个设备之间建立加密的数据传输通道，利用该通道在两个设备之间进行数据同步。

分布式数据服务提供了两种同步模式：**手动同步**和**自动同步模式**。**手动同步模式**完全由应用程序调用接口来触发，并且支持指定同步的设备列表和同步模式（**PULL、PUSH 和 PULL_PUSH 三种同步模式**）。**自动同步模式**由分布式数据库来完成数据同步（同步时机包括设备上线、应用程序修改数据等），业务不感知同步操作。

- **单版本分布式数据库**

单版本是指数据在本地保存是以单个 KV 条目为单位的方式保存，对每个 Key 最多只保存一个条目项，当数据在本地被用户修改时，不管它是否已经被同步出去，均直接在这个条目上进行修改。同步也以此为基础，按照它在本地被写入或更改的顺序将当前最新一次修改逐条同步至远端设备。

- **设备协同分布式数据库**

设备协同分布式数据库建立在单版本分布式数据库之上，对应用程序存入的 KV 数据中的 Key 前面拼接了本设备的 DeviceID 标识符，这样能保证每个设备产生的数据严格隔离，底层按照设备的维度管理这些数据，设备协同分布式数据库支持以设备的维度查询分布式数据，但是不支持修改远端设备同步过来的数据。

- **分布式数据库冲突解决策略**

分布式数据库多设备提交冲突场景，在给提交冲突做合并的过程中，如果多个设备同时修改了同一数据，则称这种场景为数据冲突。数据冲突采用默认冲突解决策略，基于提交时间戳，取时间戳较大的提交数据，当前不支持定制冲突解决策略。

- **数据库 Schema 化管理与谓词查询**

单版本数据库支持在创建和打开数据库时指定 Schema，数据库根据 Schema 定义感知 KV 记录的 Value 格式，以实现对 Value 值结构的检查，并基于 Value 中的字段实现索引建立和支持谓词查询。

- **分布式数据库备份能力**

提供分布式数据库备份能力，业务通过设置 backup 属性为 true，可以触发分布式数据服务每日备份。当分布式数据库发生损坏，分布式数据服务会删除损坏数据库，并且从备份数据库中恢复上次备份的数据。如果不存在备份数据库，则创建一个新的数据库。同时支持加密数据库的备份能力。

运作机制

分布式数据服务支撑 HarmonyOS 系统上应用程序数据库数据分布式管理，支持数据在相同帐号的多端设备之间相互同步，为用户在多端设备上提供一致的用户体验，分布式数据服务包含五部分：

- **服务接口**

分布式数据服务提供专门的数据库创建、数据访问、数据订阅等接口给应用程序调用，接口支持 KV 数据模型，支持常用的数据类型，同时确保接口的兼容性、易用性和可发布性。

- **服务组件**

服务组件负责服务内元数据管理、权限管理、加密管理、备份和恢复管理以及多用户管理等、同时负责初始化底层分布式 DB 的存储组件、同步组件和通信适配层。

- **存储组件**

存储组件负责数据的访问、数据的缩减、事务、快照、数据库加密，以及数据合并和冲突解决等特性。

- **同步组件**

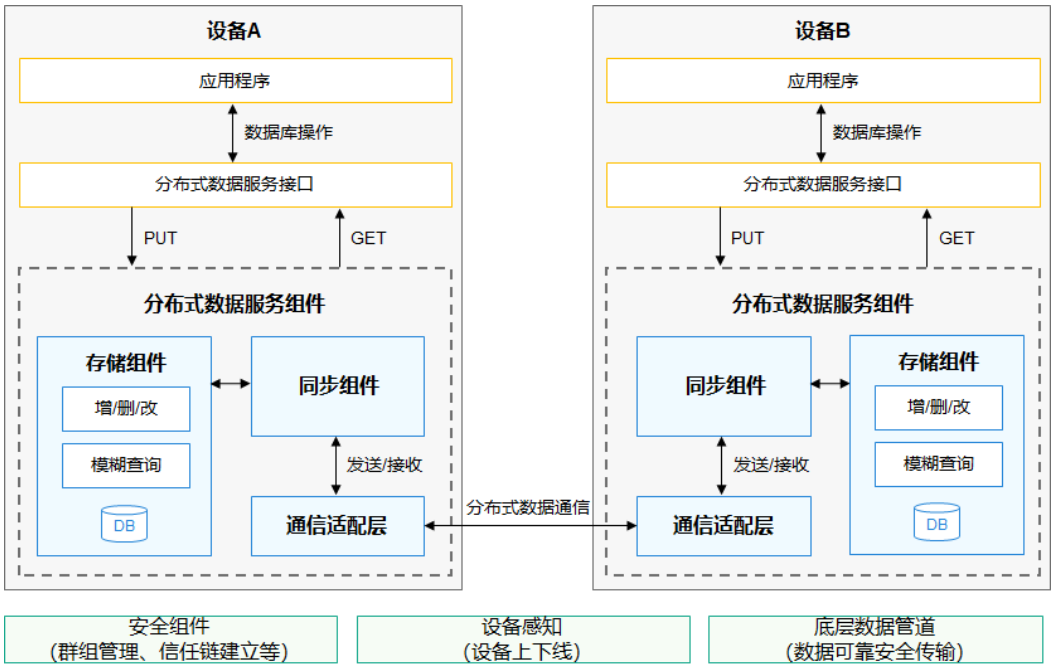
同步组件连结了存储组件与通信组件，其目标是保持在线设备间的数据库数据一致性，包括将本地产生的未同步数据同步给其他设备，接收来自其他设备发送过来的数据，并合并到本地设备中。

- **通信适配层**

通信适配层负责调用底层公共通信层的接口完成通信管道的创建、连接，接收设备上下线消息，维护已连接和断开设备列表的元数据，同时将设备上下线信息发送给上层同步组件，同步组件维护连接的设备列表，同步数据时根据该列表，调用通信适配层的接口将数据封装并发送给连接的设备。

应用程序通过调用分布式数据服务接口实现分布式数据库创建、访问、订阅功能，服务接口通过操作服务组件提供的能力，将数据存储至存储组件，存储组件调用同步组件实现将数据同步，同步组件使用通信适配层将数据同步至远端设备，远端设备通过同步组件接收数据，并更新至本端存储组件，通过服务接口提供给应用程序使用。

图 1　数据分布式运作示意图



约束与限制

- 应用程序如需使用分布式数据服务完整功能，需要申请ohos.permission.DISTRIBUTED_DATASYNC 权限。
- 分布式数据服务的数据模型仅支持 KV 数据模型，不支持外键、触发器等关系型数据库中的技术点。
- 分布式数据服务支持的 KV 数据模型规格：
- 设备协同数据库，Key 最大支持 896Byte，Value 最大支持 4MB - 1Byte。
- 单版本数据库，Key 最大支持 1KB，Value 最大支持 4MB - 1Byte。
- 每个应用程序最多支持同时打开 16 个 KvStore。
- 由于支持的存储类型不完全相同等原因，分布式数据服务无法完全代替业务沙箱内数据库数据的存储功能，开发人员需要确定要做分布式同步的数据，把这些数据保存到分布式数据服务中。
- 分布式数据服务当前不支持应用程序自定义冲突解决策略。

开发指导

场景介绍

分布式数据服务主要实现对用户设备中应用程序的数据内容的分布式同步。当设备 1 上的应用 A 在分布式数据库中增、删、改数据后，设备 2 上的应用 A 也可以获取到该数据库变化。可在分布式图库、信息、通讯录、文件管理器等场景中使用。

接口说明

HarmonyOS 系统中的分布式数据服务模块为开发者提供下面几种功能：

表 1 分布式数据服务关键 API 功能介绍

功能分类	接口名称	描述
分布式数据库创建、打开、关闭和删除。	isCreateIfMissing()	数据库不存在时是否创建。
	setCreateIfMissing(boolean isCreateIfMissing)	数据库不存在时是否创建。
	isEncrypt()	获取数据库是否加密。
	setEncrypt(boolean isEncrypt)	设置数据库是否加密。
	getStoreType()	获取分布式数据库的类型。
	setStoreType(KvStoreType storeType)	设置分布式数据库的类型。
	KvStoreType.DEVICE_COLLABORATION	设备协同分布式数据库类型。
	KvStoreType.SINGLE_VERSION	单版本分布式数据库类型。

表 1 分布式数据服务关键 API 功能介绍

功能分类	接口名称	描述
	getKvStore(Options options, String storeId)	根据 Options 配置创建和打开标识符为 storeId 的分布式数据库。
	closeKvStore(KvStore kvStore)	关闭分布式数据库。
	deleteKvStore(String storeId)	删除分布式数据库。
分布式数据增、删、改、查。	getStoreId()	根据配置构造帐号键值数据库管理类实例。
	putBoolean(String key, boolean value) putInt(String key, int value) putFloat(String key, float value) putDouble(String key, double value) putString(String key, String value) putByteArray(String key, byte[] value) putBatch(List<Entry> entries)	插入和更新数据。
	delete(String key) deleteBatch(List<String> keys)	删除数据。
	getInt(String key) getFloat(String key) getDouble(String key) getString(String key) getByteArray(String key) getEntries(String keyPrefix)	查询数据。
分布式数据谓词查询。	select() reset() equalTo(String field, int value) equalTo(String field, long value) equalTo(String field, double value)	对于 Schema 数据库谓词查询数据。

表 1 分布式数据服务关键 API 功能介绍

功能分类	接口名称	描述
	equalTo(String field, String value) equalTo(String field, boolean value) notEqualTo(String field, int value) notEqualTog(String field, long value) notEqualTo(String field, boolean value) notEqualTo(String field, String value) notEqualTo(String field, double value) greaterThan(String field, int value) greaterThan(String field, long value) greaterThan(String field, double value) greaterThan(String field, String value) lessThan(String field, int value) lessThan(String field, long value) lessThan(String field, double value) lessThan(String field, String value) greaterThanOrEqualTo(String field, int value) greaterThanOrEqualTo(String field, long value) greaterThanOrEqualTo(String field, double value) greaterThanOrEqualTo(String field, String value) lessThanOrEqualTo(String field, int value) lessThanOrEqualTo(String field, long value) lessThanOrEqualTo(String field, double value) lessThanOrEqualTo(String field, String value) isNull(String field) orderByDesc(String field) orderByAsc(String field)	

表 1 分布式数据服务关键 API 功能介绍

功能分类	接口名称	描述
	limit(int number, int offset) like(String field, String value) unlike(String field, String value) inInt(String field, List<Integer> valueList) inLong(String field, List<Long> valueList) inDouble(String field, List<Double> valueList) inString(String field, List<String> valueList) notInInt(String field, List<Integer> valueList) notInLong(String field, List<Long> valueList) notInDouble(String field, List<Double> valueList) notInString(String field, List<String> valueList) and() or()	
订阅分布式数据变化。	subscribe(SubscribeType subscribeType, KvStoreObserver observer)	订阅数据库中数据的变化。
分布式数据同步。	sync(List<String> deviceIdList, SyncMode mode)	在手动模式下，触发数据库同步。

开发步骤

以单版本分布式数据库为例，说明开发步骤。

1. 根据配置构造分布式数据库管理类实例。

- 根据应用上下文创建 KvManagerConfig 对象。
- 创建分布式数据库管理器实例。

以下为创建分布式数据库管理器的代码示例：

```
Context context;

...

KvManagerConfig config = new KvManagerConfig(context);

KvManager kvManager =
KvManagerFactory.getInstance().createKvManager(config);
```

获取/创建单版本分布式数据库。

a. 声明需要创建的单版本分布式数据库 ID 描述。
b. 创建单版本分布式数据库。

以下为创建单版本分布式数据库的代码示例：

```
Options CREATE = new Options();


CREATE.setCreateIfMissing(true).setEncrypt(false).setKvStoreType(KvSt
oreType.SINGLE_VERSION);

String storeID = "testApp";

SingleKvStore singleKvStore = kvManager.getKvStore(CREATE, storeID);
```

订阅分布式数据变化。

1. 客户端需要实现 KvStoreObserver 接口。

2. 构造并注册 KvStoreObserver 实例。

以下为订阅单版本分布式数据库所有（本地及远端）数据变化通知的代码示例：

```
class KvStoreObserverClient implements KvStoreObserver() {

    @Override

    public void onChange(ChangeNotification notification) {

        List<Entry> insertEntries = notification.getInsertEntries();
```

```
        List<Entry> updateEntries = notification.getUpdateEntries();
        List<Entry> deleteEntries = notification.getDeleteEntries();
    }
}


KvStoreObserver kvStoreObserverClient = new KvStoreObserverClient();
singleKvStore.subscribe(SubscribeType.SUBSCRIBE_TYPE_ALL,
kvStoreObserverClient);
```

将数据写入单版本分布式数据库。

a. 构造需要写入单版本分布式数据库的 Key(键)和 Value(值)。
b. 将键值数据写入单版本分布式数据库。

以下为将字符串类型键值数据写入单版本分布式数据库的代码示例：

```
String key = "todayWeather";
String value = "Sunny";
singleKvStore.putString(key, value);
```

查询单版本分布式数据库数据。

a. 构造需要从单版本分布式数据库快照中查询的 Key(键)。
b. 从单版本分布式数据库快照中获取数据。

以下为从单版本分布式数据库中查询字符串类型数据的代码示例：

```
String key = "todayWeather";
String value = singleKvStore.getString(key);
```

同步数据到其他设备。

1. 获取已连接的设备列表。

2. 选择同步方式进行数据同步。

以下为单版本分布式数据库进行数据同步的代码示例，其中同步方式为

PUSH_ONLY：

```
List<DeviceInfo> deviceInfoList =
kvManager.getConnectedDevicesInfo(DeviceFilterStrategy.NO_FILTER);
List<String> deviceIdList = new ArrayList<>();
for (DeviceInfo deviceInfo : deviceInfoList) {
    deviceIdList.add(deviceInfo.getId());
}
singleKvStore.sync(deviceIdList, SyncMode.PUSH_ONLY);
```

关闭单版本分布式数据库。以下为关闭单版本分布式数据库的代码示例：

```
kvManager.closeKvStore(singleKvStore);
```

删除单版本分布式数据库。以下为删除单版本分布式数据库的代码示例：

```
kvManager.deleteKvStore(storeID);
```

分布式文件服务

概述

分布式文件服务能够为用户设备中的应用程序提供多设备之间的文件共享能力，支持相同帐号下同一应用文件的跨设备访问，应用程序可以不感知文件所在的存储设备，能够在多个设备之间无缝获取文件。

基本概念

- **分布式文件**

分布式文件是指依赖于分布式文件系统，分散存储在多个用户设备上的文件，应用间的分布式文件目录互相隔离，不同应用的文件不能互相访问。

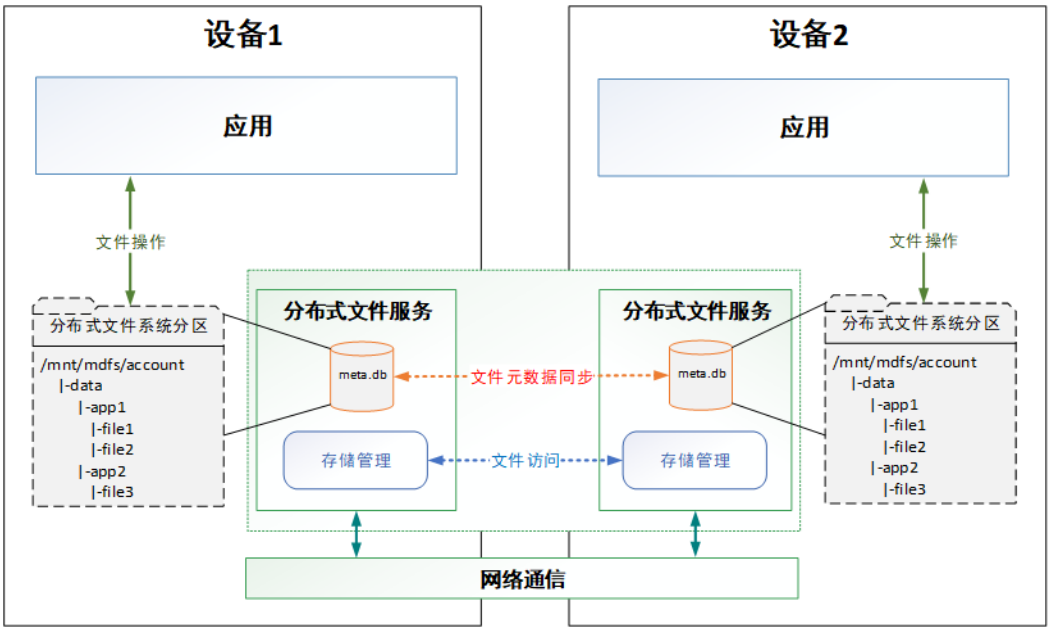
- **文件元数据**

文件元数据是用于描述文件特征的数据，包含文件名，文件大小，创建、访问、修改时间等信息。

运作机制

分布式文件服务采用无中心节点的设计，每个设备都存储一份全量的文件元数据和本设备上产生的分布式文件，元数据在多台设备间互相同步，当应用需要访问分布式文件时，分布式文件服务首先查询本设备上的文件元数据，获取文件所在的存储设备，然后对存储设备上的分布式文件服务发起文件访问请求，将文件内容读取到本地。

图 1　分布式文件服务运作示意图



约束与限制

- 应用程序如需使用分布式文件服务完整功能，需要申请ohos.permission.DISTRIBUTED_DATASYNC 权限。
- 多个设备需要打开蓝牙，连接同一 WLAN 局域网，登录相同华为帐号才能实现文件的分布式共享。
- 存在多设备并发写的场景下，为了保证文件独享，开发者需要对文件进行加锁保护。
- 应用访问分布式文件时，如果文件所在设备离线，文件不能访问。
- 非持锁情况下，并发写冲突时，后一次会覆盖前一次。
- 网络情况差时，访问存储在远端的分布式文件时，可能会长时间不返回或返回失败，应用需要考虑这种场景的处理。
- 当两台设备有同名文件时，同步元数据时会产生冲突，分布式文件服务根据时间戳将文件按创建的先后顺序重命名，为避免此场景，建议应用在文件名上做设备区分，例如，deviceID+时间戳。

开发指导

场景介绍

应用可以通过分布式文件服务实现多个设备间的文件共享，设备 1 上的应用 A 创建了分布式文件 a，设备 2 上的应用 A 能够通过分布式文件服务读写设备 1 上的文件 a。

接口说明

分布式文件兼容 POSIX 文件操作接口，应用使用 Context.getDistributedDir()接口获取目录后，可以直接使用 libc 或 JDK 访问分布式文件。

表 1 分布式文件服务 API 接口功能介绍

接口名	描述
Context.getDistributedDir()	获取文件的分布式目录

开发步骤

应用可以通过 Context.getDistributedDir()接口获取属于自己的分布式目录，然后通过 libc 或 JDK 接口，在该目录下创建、删除、读写文件或目录。

1. 设备 1 上的应用 A 创建文件 hello.txt，并写入内容"Hello World"。

```
Context context;
... // context 初始化
File distDir = context.getDistributedDir();
String filePath = distDir + File.separator + "hello.txt";
FileWriter fileWriter = new FileWriter(filePath,true);
fileWriter.write("Hello World");
```

```
fileWriter.close();
```

1. 设备 2 上的应用 A 通过 Context.getDistributedDir()接口获取分布式目录。
2. 设备 2 上的应用 A 读取文件 hello.txt。

```
FileReader fileReader = new FileReader(filePath);

char[] buffer = new char[1024];

fileReader.read(buffer);

fileReader.close();

System.out.println(buffer);
```

融合搜索

概述

HarmonyOS 融合搜索为开发者提供搜索引擎级的全文搜索能力，可支持应用内搜索和系统全局搜索，为用户提供更加准确、高效的搜索体验。

基本概念

- **全文索引**

记录字或词的位置和次数等属性，建立的倒排索引。

- **全文搜索**

通过全文索引进行匹配查找结果的一种搜索引擎技术。

- **全局搜索**

可以在系统全局统一的入口进行的搜索行为。

- **全局搜索应用**

HarmonyOS 上提供全局搜索入口的应用，一般为桌面下拉框或悬浮搜索框。

- **索引源应用**

通过融合搜索索引接口对其数据建立索引的应用。

- **可搜索配置**

每个索引源应用应该提供一个包括应用包名、是否支持全局搜索等信息的可搜索实体，以便全局搜索应用发起搜索。

- **群组**

经过认证的可信设备圈，可从账号模块获取群组 ID。

- **索引库**

一种搜索引擎的倒排索引库，包含多个索引文件的整个目录构成一个索引库。

- **索引域**

索引数据的字段名，比如一张图片有文件名、存储路径、大小、拍摄时间等，文件名就是其中的一个索引域。

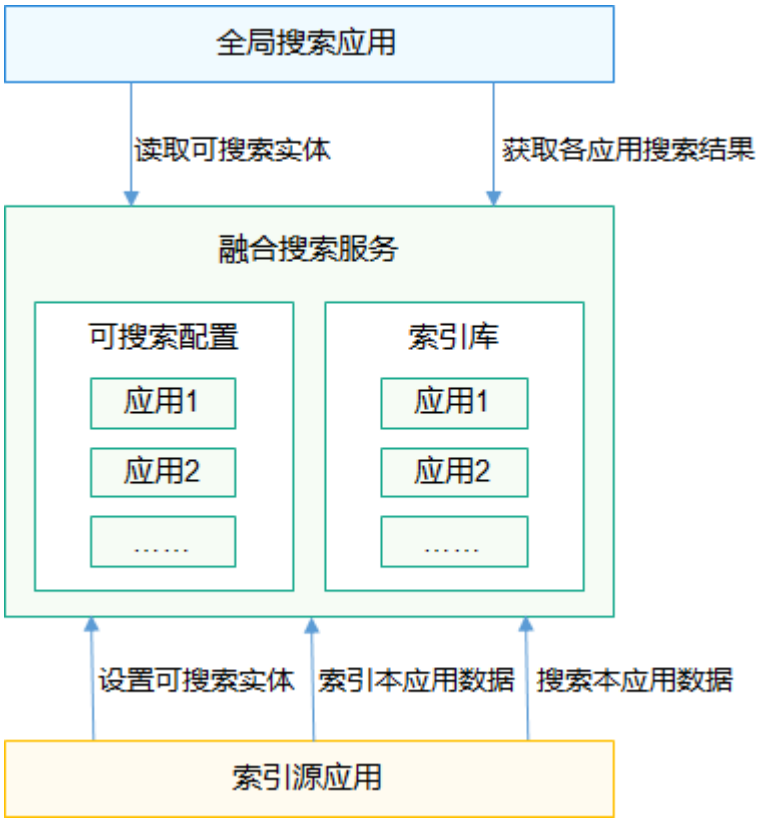
- **索引属性**

描述索引域的信息，包括索引类型、是否为主键、是否存储、是否支持分词等。

运作机制

索引源应用通过融合搜索接口设置可搜索实体，并为其数据内容构建全文索引。全局搜索应用接收用户发起的搜索请求，遍历支持全局搜索的可搜索实体，解析用户输入并构造查询条件，最后通过融合搜索接口获取各应用搜索结果。

图 1　融合搜索运作示意图



约束与限制

- 构建索引或者发起搜索前，索引源应用必须先设置索引属性，并且必须有且仅有一个索引域设置为主键，且主键索引域不能分词，索引和搜索都会使用到索引属性。
- 索引源应用的数据发生变动时，开发者应同步通过融合搜索索引接口更新索引，以保证索引和应用原始数据的一致性。
- 批量创建、更新、删除索引时，应控制单次待索引内容大小，建议分批创建索引，防止内存溢出。
- 分页搜索和分组搜索应控制每页返回结果数量，防止内存溢出。
- 构建和搜索本机索引时，应该使用提供的 SearchParameter.DEFAULT_GROUP 作为群组 ID，分布式索引使用通过账号模块获取的群组 ID。
- 搜索时需先创建搜索会话，并务必在搜索结束时关闭搜索会话，释放内存资源。
- 使用融合搜索服务接口需要在“config.json”配置文件中添加“ohos.permission.ACCESS_SEARCH_SERVICE”权限。
- 搜索时的 SearchParamter.DEVICE_ID_LIST 必须与创建索引时的 deviceId 一致。

开发指导

场景介绍

索引源应用，一般为有持久化数据的应用，可以通过融合搜索接口为其应用数据建立索引，并配置全局搜索可搜索实体，帮助用户通过全局搜索应用查找本应用内的数据。应用本身也提供搜索框时，也可直接在应用内部通过融合搜索接口实现全文搜索功能。

接口说明

HarmonyOS 中的融合搜索为开发者提供以下几种能力，详见 API 参考。

表 1 融合搜索接口功能介绍

类名	接口名	描述
SearchAbility	public List<IndexData> insert(String groupId, String bundleName, List<IndexData> indexDataList)	索引插入
	public List<IndexData> update(String groupId, String bundleName, List<IndexData> indexDataList)	索引更新
	public List<IndexData> delete(String groupId, String bundleName, List<IndexData> indexDataList)	索引删除
SearchSession	public int getSearchHitCount(String queryJsonStr)	搜索命中结果数量
	public List<IndexData> search(String queryJsonStr, int start, int limit)	分页搜索

表 1 融合搜索接口功能介绍

类名	接口名	描述
	public List<Recommendation> groupSearch(String queryJsonStr, int groupLimit)	分组搜索

开发步骤

1. 实例化 SearchAbility，连接融合搜索服务。

```
SearchAbility searchAbility = new SearchAbility(context);
CountDownLatch lock = new CountDownLatch(1);
// 连接服务
searchAbility.connect(new ServiceConnectCallback() {
    @Override
    public void onConnect() {
        lock.countDown();
    }

    @Override
    public void onDisconnect() {
    }
});
// 等待回调，最长等待时间可自定义
lock.await(3000, TimeUnit.MILLISECONDS);
// 连接失败可重试
```

设置索引属性。

```
// 构造索引属性

List<IndexForm> indexFormList = new ArrayList<>();

IndexForm primaryKey = new IndexForm("id", IndexType.NO_ANALYZED, true,
true, false); // 主键，不分词

indexFormList.add(primaryKey);

IndexForm title = new IndexForm("title", IndexType.ANALYZED, false, true,
true); // 分词

indexFormList.add(title);

IndexForm tagType = new IndexForm("tag_type", IndexType.SORTED, false,
true, false); // 分词，同时支持排序、分组

indexFormList.add(tagType);

IndexForm ocrText = new IndexForm("ocr_text",
IndexType.SORTED_NO_ANALYZED, false, true, false); // 支持排序、分组，不分
词，所以也支持范围搜索

indexFormList.add(ocrText);

IndexForm dateTaken = new IndexForm("datetaken", IndexType.LONG, false,
true, false); // 支持排序和范围查询

indexFormList.add(dateTaken);

IndexForm bucketId = new IndexForm("bucket_id", IndexType.INTEGER, false,
true, false); // 支持排序和范围查询

indexFormList.add(bucketId);

IndexForm latitude = new IndexForm("latitude", IndexType.FLOAT, false,
true, false); // 支持范围搜索

indexFormList.add(latitude);

IndexForm longitude = new IndexForm("longitude", IndexType.DOUBLE, false,
true, false); // 支持范围搜索

indexFormList.add(longitude);


// 设置索引属性

int result = searchAbility.setIndexForm(bundleName, 1, indexFormList);
```

```
// 设置失败可重试
```

插入索引。

```
// 构建索引数据
List<IndexData> indexDataList = new ArrayList<>();
for (int i = 0; i < 5; i++) {
    IndexData indexData = new IndexData();
    indexData.put("id", "id" + i);
    indexData.put("title", "title" + i);
    indexData.put("tag_type", "tag_type" + i);
    indexData.put("ocr_text", "ocr_text" + i);
    indexData.put("datetaken", System.currentTimeMillis());
    indexData.put("bucket_id", i);
    indexData.put("latitude", i / 5.0 * 180);
    indexData.put("longitude", i / 5.0 * 360);
    indexDataList.add(indexData);
}
// 插入索引
List<IndexData> failedList =
searchAbility.insert(SearchParameter.DEFAULT_GROUP, bundleName,
indexDataList);
// 失败的记录可以持久化，稍后重试
```

构建查询。

```
// 构建查询
JSONObject jsonObject = new JSONObject();


// SearchParameter.QUERY 对应用户输入，搜索域应该都是分词的
```

```
// 这里假设用户输入是“天空”，要在"title", "tag_type"这两个域上发起搜索

JSONObject query = new JSONObject();

query.put("天空", new JSONArray(Arrays.asList("title", "tag_type")));

jsonObject.put(SearchParameter.QUERY, query);


// SearchParameter.FILTER_CONDITION 对应的 JSONArray 里可以添加搜索条件

// 对于索引库里的一条索引，JSONArray 下的每个 JSONObject 指定的条件都必须满足才会命中，JSONObject 里的条件组合满足其中一个，这个 JSONObject 指定的条件即可满足

JSONArray filterCondition = new JSONArray();

// 第一个条件，一个域上可能取多个值

JSONObject filter1 = new JSONObject();

filter1.put("bucket_id", new JSONArray(Arrays.asList(0, 1, 2))); // 一条索引在"bucket_id"的取值为 0 或 1 或 2 就能命中

filter1.put("id", new JSONArray(Arrays.asList(0, 1))); // 或者在"id"的取值为 0 或者 1 也可以命中

filterCondition.put(filter1);

// 第二个条件，一个值可能在多个域上命中

JSONObject filter2 = new JSONObject();

filter2.put("tag_type", new JSONArray(Arrays.asList("白云")));

filter2.put("ocr_text", new JSONArray(Arrays.asList("白云"))); // 一条索引只要在"tag_type"或者"ocr_text"上命中"白云"就能命中

filterCondition.put(filter2);

jsonObject.put(SearchParameter.FILTER_CONDITION, filterCondition); // 一条索引要同时满足第一和第二个条件才能命中


// SearchParameter.DEVICE_ID_LIST 对应设备 ID，匹配指定设备 ID 的索引才会命中

JSONObject deviceId = new JSONObject();

deviceId.put("device_id", new JSONArray(Arrays.asList("localDeviceId")));

jsonObject.put(SearchParameter.DEVICE_ID_LIST, deviceId);
```

```
// 可以在支持范围搜索的索引域上发起范围搜索，一条索引在指定域的值都落在对应的指定范围才会命中

JSONObject latitude = new JSONObject();

latitude.put(SearchParameter.LOWER, -40.0f); // inclusive

latitude.put(SearchParameter.UPPER, 40.0.f); // inclusive

jsonObject.put("latitude", latitude); // 纬度必须在[-40.0f, 40.0f]

JSONObject longitude new JSONObject();

longitude.put(SearchParameter.LOWER, -90.0); // inclusive

longitude.put(SearchParameter.UPPER, 90.0);  // inclusive

jsonObject.put("longitude", longitude); // 经度必须在[-90.0, 90.0]


// SearchParameter.ORDER_BY 对应搜索结果的排序，排序字段通过 SearchParameter.ASC 和 SearchParameter.DESC

// 指定搜索结果在这个字段上按照升序、降序排序，这里填充字段的顺序是重要的，比如这里两个索引之间会先在"id"

// 字段上升序排序，只有在"id"上相同时，才会继续在"datetaken"上降序排序，以此类推

JSONObject order = new JSONObject();

order.put("id", SearchParameter.ASC);

order.put("title", SearchParameter.ASC);

order.put("datetaken", SearchParameter.DESC);

jsonObject.put(SearchParameter.ORDER_BY, order);


// SearchParameter.GROUP_FIELD_LIST 对应的群组搜索的域，调用 gruopSearch 接口需要指定

jsonObject.put(SearchParameter.GROUP_FIELD_LIST, new JSONArray(Arrays.asList("tag_type", "ocr_text")));


// 得到查询字符串

String queryJsonStr = jsonObject.toString();
```

```
// 构建的 json 字符串如下：
/**
{
    "SearchParameter.QUERY": {
        "天空": [
            "title",
            "tag_type"
        ]
    },
    "SearchParameter.FILTER_CONDITION": [
        {
            "bucket_id": [
                0,
                1,
                2
            ],
            "id": [
                0,
                1
            ]
        },
        {
            "tag_type": [
                "白云"
            ],
            "ocr_text": [
                "白云"
            ]
```

```
        }
    ],
    "SearchParameter.DEVICE_ID_LIST": {
        "device_id": [
            "localDeviceId"
        ]
    },
    "latitude": {
        "SearchParameter.LOWER": -40.0,
        "SearchParameter.UPPER": 40.0
    },
    "longitude": {
        "SearchParameter.LOWER": -90.0,
        "SearchParameter.UPPER": 90.0
    },
    "SearchParameter.ORDER_BY": {
        "id": "ASC",
        "title": "ASC",
        "datetaken": "DESC"
    },
    "SearchParameter.GROUP_FIELD_LIST": [
        "tag_type",
        "ocr_text"
    ]
}
**/
```

开始搜索会话，发起搜索。

```
// 开始搜索会话

SearchSession searchSession =
searchAbility.beginSearch(SearchParameter.DEFAULT_GROUP, bundleName);

if (searchSession == null) {

    return;

}

try {

    int hit = searchSession.getSearchHitCount(queryJsonStr); // 获取总命中
数

    int batch = 50; // 每页最多返回 50 个结果

    for (int i = 0; i < hit; i += batch) {

        List<IndexData> result = searchSession.search(queryJsonStr, i,
batch);

        ...

        // 处理 IndexData

    }

    int groupLimit = 10; // 每个分组域上最多返回 10 个分组结果

    List<Recommendation> result = searchSession.groupSearch(queryJsonStr,
groupLimit);

    // 处理 Recommendation

    for (Recommendation recommendation : result) {

        HiLog.info(LABEL, "field: %{public}s, value: %{public}s,
count: %{public}d", recommedation.getField(), recommendation.getValue(),
recommendation.getCount());

    }

} finally {

    // 释放资源

    searchAbility.endSearch(SearchParameter.DEFAULT_GROUP, bundleName,
searchSession);

}
```

数据存储管理

概述

数据存储管理指导开发者基于 HarmonyOS 进行存储设备（包含本地存储、SD 卡、U 盘等）的数据存储管理能力的开发，包括获取存储设备列表，获取存储设备视图等。

基本概念

- **数据存储管理**

数据存储管理包括了获取存储设备列表，获取存储设备视图，同时也可以按照条件获取对应的存储设备视图信息。

- **设备存储视图**

存储设备的抽象表示，提供了接口访问存储设备的自身信息。

运作机制

用统一的视图结构可以表示各种存储设备，该视图结构的内部属性会因为设备的不同而不同。每个存储设备可以抽象成两部分，一部分是存储设备自身信息区域，一部分是用来真正存放数据的区域。

图 1　存储设备视图


存储设备列表
虚拟卡
视图信息
File1、File2...
SD卡
视图信息
File1、File2...
U盘
视图信息
File1、File2...
其他设备
视图信息
File1、File2...

开发指导

场景介绍

为了给用户展示存储设备信息，开发者可以使用数据存储管理接口获取存储设备视图信息，也可以根据用户提供的文件名获取对应存储设备的视图信息。

开放能力介绍

数据存储管理为开发者提供下面几种功能，具体的 API 参考。

表 1 数据存储管理接口功能介绍

功能分类	类名	接口名	描述
查询设备视图	ohos.data.usage.DataUsage	getVolumes()	获取当前用户可用的设备列表视图。
		getVolume(File file)	获取存储该文件的存储设备视图。
		getVolume(Context context, Uri uri)	获取该 URI 对应文件所在的存储设备视图。
		getDiskMountedStatus()	获取默认存储设备的挂载状态。
		getDiskMountedStatus(File path)	获取存储该文件设备的挂载状态。
		isDiskPluggable()	默认存储设备是否为

表 1 数据存储管理接口功能介绍

功能分类	类名	接口名	描述
			可插拔设备。
		isDiskPluggable(File path)	存储该文件的设备是否为可插拔设备。
		isDiskEmulated()	默认存储设备是否为虚拟设备。
		isDiskEmulated(File path)	存储该文件的设备是否为虚拟设备。
查询设备视图属性	ohos.data.usage.Volume	isEmulated()	该设备是否是虚拟存储设备。
		isPluggable()	该设备是否支持插拔。
		getDescription()	获取设备描述信息。
		getState()	获取设备挂载状态。
		getVolUuid()	获取设备唯一标识符。

开发步骤

查询设备视图

调用查询设备视图接口。

```
// 获取默认存储设备挂载状态
```

```
MountState status = DataUsage.getDiskMountedStatus();
// 获取存储设备列表
Optional<List<Volume>> list = DataUsage.getVolumes();
// 默认存储设备是否为可插拔设备
boolean pluggable = DataUsage.isDiskPluggable();
```

查询设备视图属性

1. 调用查询设备视图接口获取某个设备视图 Volume。
2. 调用 Volume 的接口即可查询视图属性。

```
// 获取 example.txt 文件所在的存储设备的视图属性
Optional<Volume> volume = DataUsage.getVolume(new
File("/sdcard/example.txt"));
volume.ifPresent(theVolume -> {
        System.out.println(theVolume.isEmulated());
        System.out.println(theVolume.isPluggable());
        System.out.println(theVolume.getDescription());
        System.out.println(theVolume.getVolUuid());
    }
);
```

设备

1.车机

1.1 概述

HarmonyOS 针对汽车场景提供了驾驶安全管控和车辆控制能力集，帮助开发者构建车载控制平台上可以使用的应用。开发者通过这些能力集，可以构建出更加适合于车载控制系统上运行的应用，提高驾驶员体验，也让乘客在旅途中享受优质的乘车服务。

基本概念

- **驾驶模式与非驾驶模式**

在汽车行业，不同地域、国家对于车载中控系统有限制，例如汽车行驶过程中不允许播放视频和消息弹框，以避免影响驾驶员安全。HarmonyOS 针对汽车定义了“驾驶模式”和“非驾驶模式”用来标识车辆状态：

驾驶模式：汽车行驶过程中，当车辆状态达到或者超过车厂定义的限制标准后，当前车辆的状态就定义为“驾驶模式”状态。

非驾驶模式：与“驾驶模式”状态相对，即车辆没有达到车厂规定的限制标准，则认为是处于“非驾驶模式”状态。

在驾驶模式状态下，HarmonyOS 系统会根据当前车辆限制标准，对系统能力做约束，例如不允许播放视频和弹框，而在非驾驶模式状态下，系统能力则不受影响。

- **驾驶模式支持应用**

HarmonyOS 在应用增加了“驾驶模式”状态支持。对于“驾驶模式”状态支持的应用，在车辆行驶过程中可以正常运行，而对于“驾驶模式”状态不支持的应用，则在车辆行驶过程中做限制，例如禁止播放视频，禁止文本弹框等，不同的厂商限制不同，具体详情请参考车厂说明。

HarmonyOS 应用市场在应用上架时会进行审核，对于“驾驶模式”状态支持的应用，HarmonyOS 规定开发者要遵守汽车行业应用开发规范要求，具体参考驾驶安全管控章节。

约束与限制

- HarmonyOS 车载应用要求支持“驾驶模式”和“非驾驶模式”状态切换。
- 驾驶模式下，默认不允许执行影响驾驶安全的所有操作，例如播放视频，弹框等。不同车厂、地域、国家对影响驾驶安全的操作限制不同，开发者需要基于具体限制开发应用，以确保驾驶员驾驶安全，共同营造安全的驾驶体验。

1.2 驾驶安全管控

1.2.1 开发驾驶模式支持应用

场景介绍

HarmonyOS 除了限制系统能力来保证驾驶员安全，同时提供了驾驶模式相关接口，允许开发者使用第三方能力库来开发驾驶模式下可用的安全应用，本章节主要简述如何开发驾驶模式下安全应用。

接口说明

HarmonyOS 提供了驾驶模式管理类 DrivingSafetyManager，开发者可以使用该类的开放能力，开发符合驾驶模式安全要求的应用。

表 1 DrivingSafetyManager 的主要接口

接口名	描述
getRestraint()	获取当前系统在“驾驶模式”状态下的约束条件。
isDrivingMode()	查询当前车辆是否处于“驾驶模式”状态。
isDrivingSafety()	判断当前的应用是否是安全的。

开发步骤

开发一个应用具备如下能力：

- 音乐播放能力。
- 通过弹框来显示通知信息。
- 视频播放能力（三方视频播放开发库）。
- 遵守地区法规，在车辆行驶过程中不能弹框和播放视频。

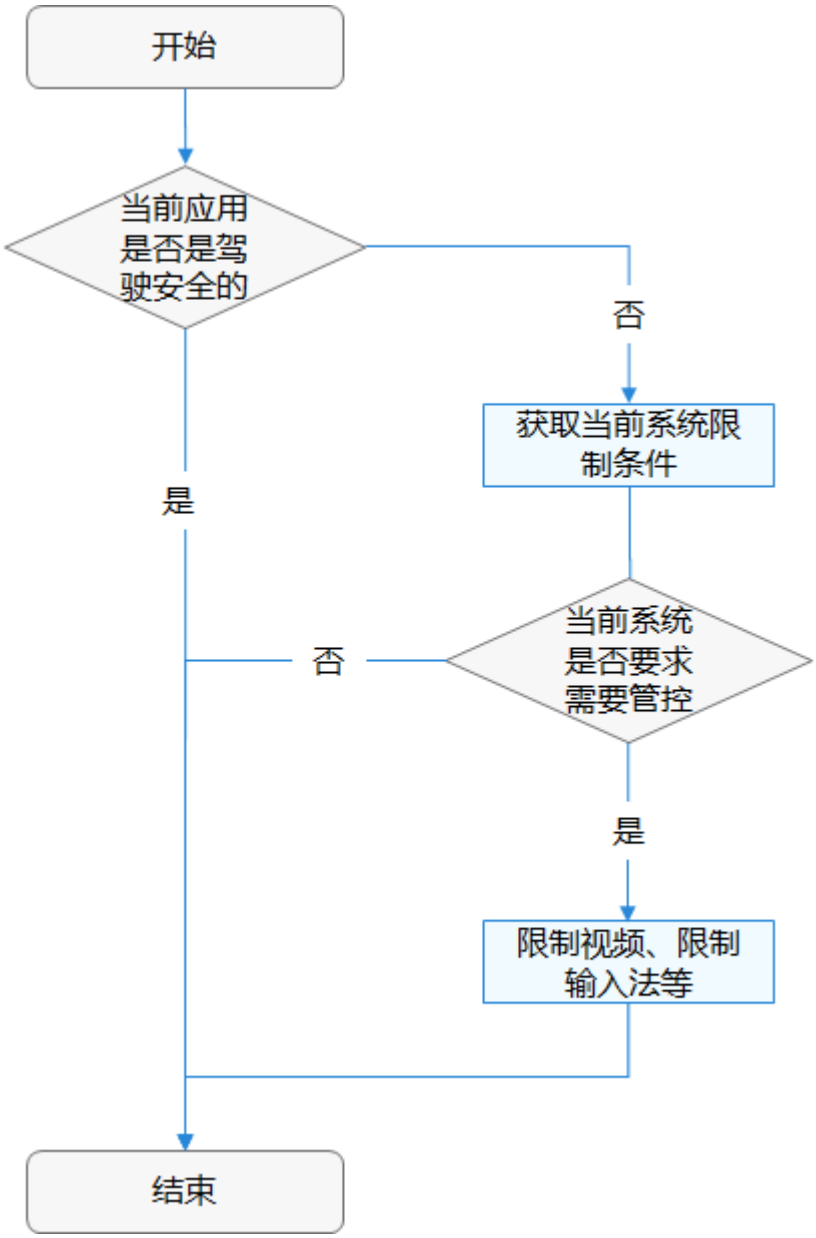
1. 在开始构建应用之前，请务必遵守 HarmonyOS 的约束和限制。
2. 为应用添加驾驶模式支持。

HarmonyOS 车载应用需要开发者指定当前应用是否支持“驾驶模式”状态。对于不支持驾驶模式状态的应用，在汽车进入“驾驶模式”状态后，不允许启动，对于已经启动的应用也会冻结操作并退出。因此，开发者需要在应用配置文件（config.json）中"supported-modes"配置项中增加"drive"模式，以表示该应用支持“驾驶模式”状态，保证车辆在行驶过程中，应用可以正常运行。如下所示：

```
."abilities": {
.    "name": ".carlink",
.    "icon": "$carlink:icon",
.    "label": "carlink",
.    "supported-modes": ["drive"],
.}
```

判断当前系统是否处于“驾驶模式”状态，应用后台通过调 isDrivingSafety()接口，判断当前应用是否是驾驶安全的：

- 如果是非驾驶安全的，则通过 getRestraint()接口获取当前系统的限制条件，根据系统限制条件，对当前的应用做处理，例如禁止视频播放，禁止输入法弹框；
- 如果是驾驶安全的，则无需处理。


开始
当前应用是否是驾驶安全的
否
获取当前系统限制条件
是
否
当前系统是否要求需要管控
是
限制视频、限制输入法等
结束

```
if (isDrivingSafety(context)) { // 判断应用当前状态是否是驾驶安全的
    int restraintCode = DrivingSafetyManager.getRestraint(); // 获取当前系统限制条件
    if (restraintCode < 0) {
        HiLog.error("GetRestraint error: %d", restraintCode);
        return;
    }
    if (restraintCode == 0) { // 当前系统不受限
         HiLog.error("No restraint");
         return;
    }
    // 限制视频播放
    if (0x2 & restraintCode != 0) {
        Player play = new Player(content); // 第三方视频播放器
        play.stop();
    }
    // 限制输入法弹窗
    if (0x4 & restraintCode != 0) {
        InputMethodController mIMController = InputMethodController.getInstance(); // 第三方输入法

mIMController.stopInput(InputMethodController.STOP_IM_NORMAL);
    }
    // 其他限制
    ...
}
```

1.2.2 定制化系统能力约束

场景介绍

HarmonyOS 提供了系统能力管控接口，允许车厂开发类似“系统设置”类应用，基于当前车型限制条件下，车厂可以提供一些系统能力，允许用户进行自定义管控策略。例如，某车型默认在“驾驶模式”状态下不允许播放视频，但可以允许消息弹出框正常弹出。用户可以根据习惯，为了驾驶安全，将消息弹出框也做限制，不允许在“驾驶模式”状态下弹出。本章节主要指导车厂如何使用定制化管控系统能力。

接口说明

HarmonyOS 提供的驾驶安全管控能力支持定制化管理，三方车厂可以通过 DrivingSafetyConfig 类的能力来开发管控类应用。

说明

1. 不同的车厂提供的能力不同，具体需要参考三方车厂能力限制说明；
2. 该开放能力只对 OEM 车厂开放，普通三方开发者不可调用。

表 1 DrivingSafetyConfig 的主要接口

接口名	描述
getSysDrivingSafetyConfigure()	查询指定的系统能力是否被管控。
setSysDrivingSafetyConfigure()	设定指定的系统能力是否被管控，具体需要参考三方车厂能力限制说明，不同车厂提供的限制能力不同。

目前，HarmonyOS 提供了两种系统能力管控的能力：

- SysDrivingSafetyControlItems.DM_IME：对系统输入法做管控
- SysDrivingSafetyControlItems.DM_Video：对系统视频播放器做管控
- SysDrivingSafetyControlItems.DM_AUTO_RUN：对自启动做管控
- SysDrivingSafetyControlItems.DM_REMOTE_CONTROL：对远程控制做管控
- SysDrivingSafetyControlItems.DM_UPGRADE：对升级做管控

开发步骤

1. 当开发者要查询当前的系统策略时，可以通过 getSysDrivingSafetyConfigure()接口获取。

2. 当开发者需要修改策略时，可以通过 setSysDrivingSafetyConfigure()接口修改当前系统能力管控策略。

```
// 构造查询结果对象
DrivingSafetyConfigResult result = new DrivingSafetyConfigResult();
// 调查询能力接口
try{
    int errorCode = DrivingSafetyConfig.getSysDrivingSafetyConfigure(SysDrivingSafetyControlItems.DM_IME, result);
    if (errorCode != 0) {
        HiLog.error("Get DrivingSafetyConfig Error: %d", errorCode));
        return;
    }
    Boolean isOpen = false;
    if (!result.isOpen()){ // 当前输入法策略为非管控状态
```

```
        isOpen = true; // 修改当前输入法策略为管控状态
    }
    // 调用修改管控能力接口，修改管控策略
    errorCode =
DrivingSafetyConfig.setSysDrivingSafetyConfigure(SysDrivingSafetyCont
rolItems.DM_IME, isOpen);
    if (errorCode != 0) {
        HiLog.error("Set DrivingSafetyConfigre Error: %d", errorCode);
        return;
    }
} catch (RemoteException | IllegalArgumentException e) {
    HiLog.error("System Error: %s", e.getMessage())
    return;
}
```

1.3 车辆控制

1.3.1 开发车辆控制应用

场景介绍

HarmonyOS 提供了车辆控制的能力接口，开发者可以基于其能力接口，开发相关的控制应用。例如，通过应用来控制车内空调温度、车窗开合程度、雨刷器、左右后视镜，查询发动机运行状况、转速等。

说明

车辆控制能力与车厂车型息息相关，HarmonyOS 提供统一的标准接口，具体能力请参考各个车辆说明。

接口说明

- 车机专有硬件服务连接类 Vehicle，支持车机专有硬件所有服务连接能力，同时携带自动重试机制。当车机专有硬件服务连接或者断开时，支持开发者实现自定义回调，具体开放能力如下：

表 1 Vehicle 的主要接口

接口名	描述
connect()	连接指定车机专有硬件服务。
disconnect()	断开指定车机专有硬件服务。
isConnected()	判断指定车机专有硬件服务是否已连接。

- 车辆座舱管理类 VehicleCabinManager，提供了车辆座舱信号访问控制方法，例如车门、空调。开发者可以通过定义的车辆信号标识来获取或者设置对应的信号值，完成对车辆座舱的控制，具体开放能力如下：

表 2 VehicleCabinManager 的主要接口

接口名	描述
getVehicleSignal()	获取座舱相关设备的信号值。
getVehicleSignalMultiAreas()	获取座舱指定信号的多区域值。
setVehicleActuator()	设置车辆座舱信号值。
subscribeVehicleSignal()	订阅指定的座舱信号。
unsubscribeVCabinSignal()	取消订阅指定的座舱信号。
unsubscribeVCabinSignalAll()	取消订阅全部座舱信号。

- 车辆车身管理类 VehicleBodyManager，提供了车辆车身设备控制相关的方法，例如雨刷器、挡风玻璃、清洁剂、车灯、引擎盖、行李箱等设备控制信息，具体开放能力如下：

表 3 VehicleBodyManager 的主要接口

接口名	描述
getVehicleSignal()	获取车身相关设备的信号值。
getVehicleSignalMultiAreas()	获取车身指定信号的多区域值。
setVehicleActuator()	设置车辆车身的信号值。
subscribeVehicleSignal()	订阅指定的车身信号。
unsubscribeVBodySignal()	取消订阅指定的车身信号。
unsubscribeVBodySignalAll()	取消订阅全部车身信号。

- 车辆底盘管理类 VehicleChassisManager，提供了车辆底盘设备控制相关的方法，例如获取车辆重量、轴距、方向盘转向角度等。具体开放能力如下：

表 4 VehicleChassisManager 的主要接口

接口名	描述
getVehicleSignal()	获取车辆底盘相关设备信号值。
getVehicleSignalMultiAreas()	获取车辆底盘指定信号的多区域值。
setVehicleActuator()	设置车辆底盘相关设备的状态值。
subscribeVehicleSignal()	订阅指定的车辆底盘信号。
unsubscribeVChassisSignal()	取消订阅指定的车辆底盘信号。
unsubscribeVChassisSignalAll()	取消订阅全部的车辆底盘信号。

- 车辆引擎管理类 VehicleDriveTrainManager，提供了车辆引擎相关控制方法，例如控制变速箱模式，获取发动机转速等，具体开放能力如下：

表 5 VehicleDriveTrainManager 的主要接口

接口名	描述
getVehicleSignal()	获取车辆引擎相关设备信号值。
getVehicleSignalMultiAreas()	获取车辆引擎指定信号的多区域值。
setVehicleActuator()	设置车辆引擎信号相关参数值。
subscribeVehicleSignal()	订阅指定的车辆引擎信号。
unsubscribeVDriveTrainSignal()	取消订阅指定的车辆引擎信号。
unsubscribeVDriveTrainSignalAll()	取消订阅全部车辆引擎信号。

- 通常在汽车使用过程中，驾驶员需要实时了解车辆的健康状态，从而判断车辆是那个部位出现故障，因此 HarmonyOS 提供了 OBD(on-board diagnostics)相关接口，供三方开发者开发车辆健康监测相关应用，更好服务于大众。

表 6 VehicleOBDManager 的主要接口

接口名	描述
getVehicleSignal()	获取 OBD 相关实时信号值。
getVehicleSignalMultiAreas()	获取 OBD 指定信号的多区域值。
setVehicleActuator()	设置 OBD 相关设备值。
subscribeVehicleSignal()	订阅指定的 OBD 设备信号。
unsubscribeVOBDSignal()	取消订阅指定的 OBD 设备信号。

表 6 VehicleOBDManager 的主要接口

接口名	描述
unsubscribeVOBDSignalAll()	取消订阅全部的 OBD 设备信号。

- 车辆配置属性管理类 VehicleConfigurationManager，提供了车辆静态属性信息查询接口，例如车辆燃油类型，车辆外观尺寸等基本属性信息，具体开放能力如下：

表 7 VehicleConfigurationManager 的主要接口

接口名	描述
getVehicleSize()	获取车辆尺寸，包括：长、宽、高等信息。
getVehicleFuelType()	获取车辆燃油类型。
getVehiclereFuelPosition(）	获取燃油口位置信息。
getVehiclereTransmissionConfiguration()	获取变速器类型。
getVehicleWheelDiameter()	获取轮胎尺寸。
getVehicleSteeringWheelConfiguration()	获取车辆方向盘配置信息。
getVehicleACRISS()	获取汽车租赁公司使用的 ACRISS 汽车分类代码。
getVehicleMcuVersion()	获取车辆 MCU 版本号。
getVehicleModel()	获取车辆制造型号。
getVehicleModelYear()	获取车辆生产时间。
getVehicleBrand()	获取车辆品牌信息。

表 7 VehicleConfigurationManager 的主要接口

接口名	描述
getVehicleVIN()	获取车辆识别号。
getVehicleWMI()	获取世界制造厂识别代码。
getDriverZone()	获取驾驶位信息。

开发步骤

连接指定车机专有硬件服务。

```
// 获取服务连接状态变化
ServiceConnectionListener listener = new ServiceConnectionListener(){
    @Override
    public void onServiceConnected(VehicleServiceName serviceName) {
    }
    @Override
    public void onServiceDisconnected(VehicleServiceName serviceName) {
    }
};
// 连接指定车机专有硬件服务
try {
    Vehicle.connect(VehicleServiceName.VEHICLECONTROL_SERVICE,
listener);
    Thread.sleep(2000);
    return true;
} catch (IllegalStateException | InterruptedException e) {
```

```
    Logger.info("Exception:" + e.toString());
    return false;
}
```

根据不同管理入口类，调用对应接口。

```
// VehicleCabinManager 类, 座舱天窗管理
String propId = VehicleCabinManager.ID_CABIN_SUNROOF_SWITCH;
int zoneId = VehicleZone.ZONE_NONE;
String value = "Inactive";
VehicleActuatorCallback callback = new VehicleActuatorCallback() {
    @Override
    public void onErrorActuator(String propId, int zoneId, int
outResult) {
    }
};
boolean result = false;
try {
    VehicleCabinManager.setVehicleActuator(propId, zoneId, callback,
value);
    result = true;
} catch (RemoteException | IllegalArgumentException e) {
    result = false;
}
```

```
if (!result) {
    System.out.println(String.format("Set sunroof error: %d", result));
}

// VehicleBodyManager 类，获取车身前挡风玻璃雨刷器状态
zoneId = VehicleZone.ZONE_FRONT;
String signal Value = VehicleBodyManager.getVehicleSignal(String.class,
VehicleBodyManager.ID_BODY_WINDSHIELD_WIPING_STATUS, zoneId);

// VehicleChassisManager 类，获取车辆轮胎宽度
zoneId = VehicleZone.ZONE_ROW1;
Short signalValue =
VehicleChassisManager.getVehicleSignal(Short.class,
VehicleChassisManager.ID_CHASSIS_AXLE_WHEELWIDTH, zoneId);

// VehicleDriveTrainManager 类，设置车辆变速箱模式
propId =
VehicleDriveTrainManager.ID_DRIVETRAIN_TRANSMISSION_PERFORMANCEMODE;
zoneId = VehicleZone.ZONE_NONE;
String transmissionValue = "sport";
VehicleActuatorCallback tmCallback = new VehicleActuatorCallback() {
    @Override
    public void onErrorActuator(String propId, int zoneId, int
outResult) {
    }
};
try {
    VehicleDriveTrainManager.setVehicleActuator(propId, zoneId,
tmCallback, transmissValue);
    result = true;
```

```
} catch (RemoteException | IllegalArgumentException e) {
    result = false;
}
if(!result) {
    System.out.println(String.format("Set transmiss performance mode error: %d", result));
}
// VehicleConfigurationManager 类，获取车辆识别码
String vin = VehicleConfigurationManager.getVehicleVIN();
```

1.3.2 OEM 扩展接口

场景介绍

为了支持不同 OEM 车型信号矩阵定制化需求，HarmonyOS 提供了 OEM 扩展接口，用于访问/设置/订阅/去订阅 OEM 自定义信号。

说明

该功能针对不同的 OEM 车厂/车型，提供了统一的 OEM 扩展接口。

接口说明

目前 OEM 扩展接口提供的功能有如下表所示：

表 1 VehicleVendorExtensionManager 的主要接口

接口名	描述
getVehicleSignal()	获取 OEM 自定义信号实时取值。
getVehicleSignalMultiAreas()	获取指定 OEM 自定义信号的多区域值。

表 1 VehicleVendorExtensionManager 的主要接口

接口名	描述
setVehicleActuator()	设置 OEM 自定义执行器参数值。
subscribeVehicleSignal()	订阅指定的 OEM 自定义信号。
unsubscribeVehicleSignal()	取消订阅指定的 OEM 自定义信号。
unsubscribeVehicleSignalAll()	取消订阅全部的 OEM 自定义信号。

开发步骤

根据不同管理入口类，调对应接口。

```
// 设置辅助输入信号值
String propId = "OEM_Status_DTCCountTest";
int zoneId = VehicleZone.ZONE_NONE;
Boolean value = true;
VehicleActuatorCallback callback = new VehicleActuatorCallback() {
    @Override
    public void onErrorActuator(String propId, int zoneId, int outResult) {

    }
};
bool result = true;
try {
    VehicleVendorExtensionManager.setVehicleActuator(propId, zoneId, callback, value);
} catch(RemoteException | IllegalArgumentException e) {
    result = false;
}
if(!result) {
    System.out.println(String.format("Set transmiss performance mode error: %d", result));
```

```
18. }
```

1.3.3 开发 TBOX 相关应用

场景介绍

如果某款车型上装载了车载 T-BOX（Telematics BOX）盒子，开发者可以通过 HarmonyOS 提供的 T-BOX 相关接口获取或设置相关信息，如访问 T-BOX 的 xCall、定时充电等信息。

说明

该功能与具体的车厂车型相关，部分低配车型可能不具备该项功能。

接口说明

目前 TBOX 提供的功能有如下表所示：

表 1 TBoxManager 的主要接口

接口名	描述
getProperty()	获取指定 TBOX 信号值。
setActuator()	设置指定 TBOX 执行器的信号值。

表 1 TBoxManager 的主要接口

接口名	描述
subscribeProperty()	订阅指定 TBOX 信号。
unsubscribeProperty()	取消订阅指定的 TBOX 信号。
unsubscribeAllProperty()	取消所有订阅的 TBOX 信号。
subscribeBatchProperties()	批量订阅 TBOX 信号。

开发步骤

根据不同管理入口类，调对应接口。

```
// 设置 TBOX 属性值
String incorrectPath = TBoxManager.ID_TBOX_BCALL_STATUS;
byte[] result = null;
TBoxPropertyManager manager = new TBoxPropertyManager();
boolean isTrue = false;
try {
    result = manager.getBuffer(tboxPropPath);
    isTrue = true;
} catch (RemoteException | IllegalArgumentException | UnsupportedOperationException e) {
    isTrue = false;
}
```

1.3.4 开发 CLUSTER 相关应用

场景介绍

通常在汽车使用过程中，驾驶员需要设置仪表屏亮度、时间单位等参数，将电台、音乐等娱乐数据或导航数据显示在仪表屏上，因此 HarmonyOS 提供了和仪表交互相关的接口，供三方开发者开发仪表设置、显示等相关应用。

说明

该功能与具体的车厂车型相关，部分低配车型可能不具备该项功能。

接口说明

目前 Cluster 提供的功能有如下表所示：

表 1 ClusterManager 的主要接口

接口名	描述
getClusterSignal()	获取指定 Cluster 信号值。
setClusterActuator()	设置指定 Cluster 执行器值。
sendClusterSignal()	发送指定字节数组类型的 Cluster 信号请求信息。
subscribeClusterSignal()	订阅指定 Cluster 信号。
subscribeBatchProperties()	批量订阅 Cluster 信号。
unsubscribeClusterSignal()	取消订阅指定的 Cluster 信号。
unsubscribeClusterSignalAll()	取消所有订阅的 Cluster 信号。

开发步骤

根据不同管理入口类，调对应接口。

```
// 设置 Cluster 属性值
String propId = ClusterManager.ID_CLUSTER_SETTINGS_BRIGHTNESS;
ClusterActuatorCallback callback = new ClusterActuatorCallback() {
    @Override
    public void onErrorActuator(String propId, int errorCode) {}
};
boolean result = false;
byte[] value = new byte[1];
try {
    ClusterManager.sendClusterSignal(propId, callback, value);
    result = true;
} catch (RemoteException | IllegalArgumentException | UnsupportedOperationException e) {
    result = false;
}
```

1.3.5 开发 ADAS 相关应用

场景介绍

通常在汽车使用过程中，驾驶员希望通过显示、声音、预警、故障告警等方式感知行车危险或规划行驶路线，因此 HarmonyOS 提供了 ADAS 辅助交互相关的接口，供三方开发者开发 ADAS 设置、自动泊车等相关应用。

说明

该功能与具体的车厂车型相关，部分低配车型可能不具备该项功能。

接口说明

目前 ADAS 提供的功能主要有以下三类：

- 驾驶辅助管理类 DrivingAssistManager，提供了驾驶辅助相关方法，例如设置前向/后向碰撞预警开关、设置盲点检测开关、设置导航目的地及导航路径等；
- 公共信息管理类 InfoAssistManager，提供了 ADAS 公共信息管理的相关方法，例如获取障碍物信息、行车记录仪信息、车道线信息、驾驶员状态信息等；
- 自主泊车管理类 ParkingAssistManager，提供了泊车控制的相关方法，例如启动泊车、暂停泊车、设置泊车车位、获取泊车状态等。

表 1 DrivingAssistManager 的主要接口

接口名	描述
byte[] getAdasSignal()	获取指定字节数组类型的驾驶辅助信号值。
<T> T getAdasSignal()	获取指定驾驶辅助信号值。
setAdasActuator()	设置指定驾驶辅助信号值。
sendAdasSignal()	发送指定字节数组类型的驾驶辅助信号请求信息。

表 1 DrivingAssistManager 的主要接口

接口名	描述
subscribeAdasSignal()	订阅指定驾驶辅助信号。
subscribeBatchProperties()	批量订阅指定驾驶辅助信号。
unsubscribeAdasSignal()	取消订阅指定的驾驶辅助信号。
unsubscribeAdasSignalAll()	取消所有订阅的驾驶辅助信号。

表 2 InfoAssistManager 的主要接口

接口名	描述
byte[] getAdasSignal()	获取指定字节数组类型的 Adas 信号值。
<T> T getAdasSignal	获取指定 Adas 信号值。
setAdasActuator()	设置指定 Adas 信号值。
sendAdasSignal()	发送指定字节数组类型的 Adas 信号请求信息。
subscribeAdasSignal()	订阅指定 Adas 信号。
subscribeBatchProperties()	批量订阅指定 Adas 信号。
unsubscribeAdasSignal()	取消订阅指定的 Adas 信号。
unsubscribeAdasSignalAll()	取消所有订阅的 Adas 信号。

表 3 ParkingAssistManager 的主要接口

接口名	描述
byte[] getAdasSignal()	获取指定字节数组类型泊车信号值。

表 3 ParkingAssistManager 的主要接口

接口名	描述
<T> T getAdasSignal()	获取指定泊车信号值。
setAdasActuator()	设置指定泊车信号值。
sendAdasSignal()	发送指定字节数组类型泊车信号请求值。
subscribeAdasSignal()	订阅指定泊车信号。
subscribeBatchProperties()	批量订阅指定的泊车信号。
unsubscribeAdasSignal()	取消订阅指定的泊车信号。
unsubscribeAdasSignalAll()	取消所有订阅的泊车信号。

开发步骤

根据不同管理入口类，调对应接口。

```
// DrivingAssistManager 类使用
boolean result = false;
try {
    Boolean signalValue = DrivingAssistManager.getAdasSignal(Boolean.class, DrivingAssistManager.ID_DRIVING_FCW_WARNING_SWITCH);
    result = true;
} catch (RemoteException | IllegalArgumentException | UnsupportedOperationException e) {
    result = false;
}

// ParkingAssistManager 类使用
String propId = ParkingAssistManager.ID_PARKING_APA_FUNCTION_SWITCH;
Boolean value = true;
AdasActuatorCallback callback = new AdasActuatorCallback() {
    @Override
    public void onErrorActuator(String propId, int outResult) {}
```

```
16. };
17. boolean result = false;
18. try {
19.     ParkingAssistManager.setAdasActuator(propId, callback, value);
20.     result = true;
21. } catch (RemoteException | IllegalArgumentException | UnsupportedOperationException e) {
22.     result = false;
23. }
24. // InfoAssistManager 类使用
25. boolean result = false;
26. byte[] request = {'q', 'w'};
27. try {
28.     byte[] response = InfoAssistManager.getAdasSignal(InfoAssistManager.ID_INFO_HDMINFO, request);
29.     result = true;
30. } catch (RemoteException | IllegalArgumentException | UnsupportedOperationException e) {
31.     result = false;
32. }
```

1.4 打造车载系统应用

1.4.1 创建车载应用项目

说明

开始前，请参考 DevEco Studio 快速开始完成环境搭建、创建并运行一个项目。

配置 config.json

1. 添加访问车机硬件信息权限申请。

```
"reqPermissions": [
  {
    "name": "ohos.permission.vehicle.READ_VEHICLE_HMI_INFO",
    "reason": "",
    "usedScene": {
      "ability": [
        ".MainAbility"
      ],
      "when": "inuse"
    }
  }
]
```

2.添加支持驾驶模式标签。

```
"abilities": {
  "name": ".carlink",
  "icon": "$carlink:icon",
  "label": "carlink",
  "supported-modes": ["drive"],
}
```

1.4.2 添加多媒体支持

本小节主要说明 HarmonyOS 车载多媒体的使用方法，以音乐 Demo 开发为例，开发步骤如下：

1. 在布局中添加音乐播放控件。

```
<?xml version="1.0" encoding="utf-8"?>
<DirectionalLayout xmlns:ohos="http://schemas.huawei.com/res/ohos"
                   ohos:id="$+id:play_music_root"
                   ohos:width="-1"
                   ohos:height="-1"
                   ohos:left_padding="24vp"
                   ohos:right_padding="24vp"
                   ohos:orientation="1">
    <AdaptiveBoxLayout ohos:id="$+id:title_bar"
                       ohos:width="-1"
                       ohos:height="-2"
                       ohos:top_margin="24vp">
        <Image ohos:id="$+id:arrow_down_btn"
```

```
            ohos:width="24vp"
            ohos:height="24vp"
            ohos:align_parent_left="$+id:title_bar"
            ohos:image_src="$media:default.png"/>
        <Image ohos:id="$+id:music_heart_btn"
            ohos:width="24vp"
            ohos:height="24vp"
            ohos:left_of="$+id:music_hiplay_btn"
            ohos:image_src="$media:default.png"/>
        <Image ohos:id="$+id:music_hiplay_btn"
            ohos:width="24vp"
            ohos:height="24vp"
            ohos:left_margin="16vp"
            ohos:align_parent_right="$+id:title_bar"
            ohos:image_src="$media:default.png"/>
    </AdaptiveBoxLayout>
    <DirectionalLayout ohos:id="$+id:cover_container"
                    ohos:width="-1"
                    ohos:height="-2"
                    ohos:weight="1"
                    ohos:orientation="1">
        <AdaptiveBoxLayout
            ohos:id="$+id:music_cover_adapt"
            ohos:width="-1"
            ohos:height="-1">
            <DirectionalLayout
                ohos:id="$+id:music_cover_wrap1"
                ohos:width="-2"
```

```
                ohos:height="-2"
                ohos:padding="20vp"
                ohos:orientation="1">
            <Image ohos:id="$+id:music_cover"
                   ohos:width="300vp"
                   ohos:height="300vp"
                   ohos:layout_alignment="17"
                   ohos:image_src="$media:default.png"/>
        </DirectionalLayout>
        <DirectionalLayout
                ohos:id="$+id:music_cover_wrap2"
                ohos:width="-1"
                ohos:height="-1"
                ohos:orientation="1">
            <DirectionalLayout
                    ohos:width="-1"
                    ohos:height="-2"
                    ohos:layout_alignment="17"
                    ohos:top_margin="20vp"
                    ohos:bottom_margin="20vp"
                    ohos:orientation="1">
                <Text ohos:id="$+id:music_title"
                      ohos:text_size="20vp"
                      ohos:shape="0"
                      ohos:text_color="#FF000000"
                      ohos:text_alignment="72"
                      ohos:width="-1"
                      ohos:height="-2"
```

```
                        ohos:multiple_lines="false"/>
                    <Text ohos:id="$+id:music_auth"
                        ohos:text_size="14vp"
                        ohos:shape="0"
                        ohos:top_margin="4vp"
                        ohos:text_color="#FF000000"
                        ohos:text_alignment="72"
                        ohos:width="-1"
                        ohos:height="-2"
                        ohos:multiple_lines="false"/>
                </DirectionalLayout>
                <Text ohos:id="$+id:music_lrc"
                    ohos:width="-1"
                    ohos:height="-2"
                    ohos:layout_alignment="17"
                    ohos:text="See the lights see the party the ball
grows"
                    ohos:text_size="13vp"
                    ohos:text_color="#FF000000"
                    ohos:text_alignment="72"/>
            </DirectionalLayout>
        </AdaptiveBoxLayout>
    </DirectionalLayout>

    <DirectionalLayout ohos:id="$+id:foot_wrap"
                    ohos:width="-1"
                    ohos:height="-2"
                    ohos:orientation="1">
```

```
        <DirectionalLayout ohos:id="$+id:progress_container"
                           ohos:width="-1"
                           ohos:height="-2"
                           ohos:top_margin="10vp"
                           ohos:orientation="0">
            <Text ohos:id="$+id:play_progress_time"
                  ohos:width="-2"
                  ohos:height="-2"
                  ohos:layout_alignment="16"
                  ohos:right_margin="6vp"
                  ohos:text_size="13vp"
                  ohos:text_color="#FF000000"
                  ohos:text_alignment="72"/>
            <SeekBar ohos:id="$+id:play_progress_bar"
                     ohos:width="-1"
                     ohos:height="14vp"
                     ohos:layout_alignment="16"
                     ohos:weight="1"/>
            <Text ohos:id="$+id:play_total_time"
                  ohos:width="-2"
                  ohos:height="-2"
                  ohos:layout_alignment="16"
                  ohos:left_margin="6vp"
                  ohos:text_size="13vp"
                  ohos:text_color="#FF000000"
                  ohos:text_alignment="72"/>
        </DirectionalLayout>
        <DirectionalLayout ohos:id="$+id:control_container"
```

```
                    ohos:width="-1"
                    ohos:height="96vp"
                    ohos:orientation="0">
        <DirectionalLayout ohos:id="$+id:control_box1"
                        ohos:width="-2"
                        ohos:height="-2"
                        ohos:weight="1"
                        ohos:layout_alignment="17"
                        ohos:orientation="1">
            <Image ohos:id="$+id:volume_down_btn"
                   ohos:width="24vp"
                   ohos:height="24vp"
                   ohos:layout_alignment="17"
                   ohos:image_src="$media:default.png"/>
        </DirectionalLayout>
        <DirectionalLayout ohos:id="$+id:control_box2"
                        ohos:width="-2"
                        ohos:height="-2"
                        ohos:weight="1"
                        ohos:layout_alignment="17"
                        ohos:orientation="1">
            <Image ohos:id="$+id:prev_btn"
                   ohos:width="40vp"
                   ohos:height="40vp"
                   ohos:layout_alignment="17"
                   ohos:image_src="$media:default.png"/>
        </DirectionalLayout>
        <DirectionalLayout ohos:id="$+id:control_box3"
```

```
                        ohos:width="-2"
                        ohos:height="-2"
                        ohos:weight="1"
                        ohos:layout_alignment="17"
                        ohos:orientation="1">
            <Image ohos:id="$+id:play_btn"
                   ohos:width="64vp"
                   ohos:height="64vp"
                   ohos:layout_alignment="17"
                   ohos:image_src="$media:default.png"/>
        </DirectionalLayout>
        <DirectionalLayout ohos:id="$+id:control_box4"
                        ohos:width="-2"
                        ohos:height="-2"
                        ohos:weight="1"
                        ohos:layout_alignment="17"
                        ohos:orientation="1">
            <Image ohos:id="$+id:next_btn"
                   ohos:width="40vp"
                   ohos:height="40vp"
                   ohos:layout_alignment="17"
                   ohos:image_src="$media:default.png"/>
        </DirectionalLayout>
        <DirectionalLayout ohos:id="$+id:control_box5"
                        ohos:width="-2"
                        ohos:height="-2"
                        ohos:weight="1"
                        ohos:layout_alignment="17"
```

```
                                ohos:orientation="1">
                    <Image ohos:id="$+id:volume_up_btn"
                           ohos:width="24vp"
                           ohos:height="24vp"
                           ohos:layout_alignment="17"
                           ohos:image_src="$media:default.png"/>
                </DirectionalLayout>
            </DirectionalLayout>
        </DirectionalLayout>
    </DirectionalLayout>
```

加载播放控件。

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

实现音乐播放管理类。

```
public class PlayManager {
    ...
    private Player player;
    public synchronized boolean play(String filePath, int startMilliSecond)
{
        ...
        FileDescriptor fd = IoUtil.getFileDescriptor(filePath);
        Source source = new Source(fd);
        player.setSource(source);
        boolean isSuccess = player.prepare();

        isSuccess = player.rewindTo(startMilliSecond * MICRO_MILLI_RATE,
REWIND_NEXT_SYNC);
```

```
        // 播放
        isSuccess = player.play();
        isPlaying.set(isSuccess);
        return isSuccess;
    }

    public synchronized void pause(int startMilliSecond) {
        ...
        player.pause();
    }

    public synchronized void stop() {
        if (player == null) {
            return;
        }
        player.stop();
        isPlaying.set(false);
        LogUtil.info(TAG, "stop success");
        player.release();
        player = null;
    }
}
```

调用音乐播放管理类的接口播放音乐。

```
// 指定歌曲播放
String path = "/data/music/files/data/wonderful_life.mp3";
PlayManager.getInstance().play(path,1);
```

在布局中增加视频播放控件。

```
// 视频布局实现方法
public class MySurfaceSlice extends AbilitySlice {
    ...
    public void makeSurfaceView() {
        ...
        mySurfaceProvider = new SurfaceProvider(this);

adaptiveBoxLayoutSurfaceView.AdaptiveBoxLayout.LayoutConfig().addCompon
ent(mySurfaceProvider);
    }
}
```

实现视频播放管理类。

```
public class VideoPlay {
    public synchronized void startPlay() {
        ...
        ret = playImpl.play();
    }

    public synchronized void preParePlay() {
        ...
        ret = playImpl.prepare();
    }

    public synchronized void pausePlay() {
        ...
        boolean pauseRet = playImpl.pause();;
```

```
    }

    public synchronized void setSourcePlay(String filePath) {
         ...
        FileDescriptor fd = IoUtil.getFileDescriptor(filePath);
        Source source = new Source(fd);
        playImpl.setSource(source);
    }

    @Override
    public synchronized void onStop() {
        ...
        super.onStop();
    }
}
```

调用视频播放管理类的接口播放视频。

```
// 调用视频播放类进行播放
String filePath = "/data/video/files/data/festival.mp4";
VideoPlay videoPlay = new VideoPlay()
videoPlay.setSourcePlay(filePath);
videoPlay.startPlay();
```

2.智能穿戴

2.1 概述

对于智能穿戴，应用可以通过 HarmonyOS 提供的接口实现音频、传感器、网络连接、UI 交互、消息提醒等常规业务的开发。开发者也可以根据智能穿戴的特点，打造针对智能穿戴的独特应用。

说明

本文档适用于智能穿戴应用开发，针对轻量级智能穿戴请参考轻量级智能穿戴开发。

基于 HarmonyOS，开发者既可以在智能穿戴上开发独立工作的应用，也可以开发跨设备协同工作的应用，为消费者带来更加灵活、智慧的分布式交互体验。

当开发者需要新建一个工程开发智能穿戴应用时，请参考打造智能穿戴应用。当开发者已有一个工程需要添加一个智能穿戴模块时，请参考添加智能穿戴模块。

表 1 智能穿戴应用开发的增强功能

功能	描述
应用向系统发送通	用于提醒用户有来自应用的信息。当应用向系统发出通知时，通知将按照应用维度在通知中心聚合显示，详情请参考创建智

表 1 智能穿戴应用开发的增强功能

功能	描述
知	能穿戴应用通知。
降低应用功耗	智能穿戴电池容量有限，为了让应用对用户更友好，开发者应当尽可能降低应用的功耗开销，详情请参考降低应用功耗。

基本概念

- **表盘**

智能穿戴配对完成后，开机首界面即为表盘，系统会预置一些表盘，让用户可以通过表盘快速地查看时间、计步、心率、天气等关键信息。用户也可以根据自己的喜好，长按表盘界面以选择切换自己喜爱的表盘。

2.2 打造智能穿戴应用

在开始进行智能穿戴应用开发前，请参考 DevEco Studio 快速开始完成环境搭建、创建并运行一个项目。设备类型选择“Wearable”。

智能穿戴应用支持 Java 和 JS 两种开发模式。但以下两种场景，暂时仅支持使用 Java 开发：

1. 如果开发的应用内嵌算法，需要通过 JNI （Java Native Interface，Java 本地接口）调用 so 库中的函数。
2. 应用需要较高的运算效率。

下面将介绍如何使用 JS 和 JAVA 开发一个睡眠检测应用界面。

适配圆形屏幕

在 HarmonyOS 智能穿戴应用的开发中，请使用通用的 UI 控件。针对圆形的智能穿戴，开发者需要将应用界面适配圆形屏幕，以带来更好的用户体验。应用在实际显示时仅会显示界面设计中的部分圆形界面，如示例图所示。

开发一个宽 400 高 1200 的竖长型界面，当上下滑动的时候，用户只能看到橘色圆圈内部的样式，其余部分不会展示。所以开发者在进行界面设计时，需要根据智能穿戴形状进行设计适配。

另外，穿戴设备的界面一般支持右滑退出，所以需要在 PageAblity 启动时进行设置，在 onStart 里调用 setSwipeToDismiss(true)，便于右滑退出。

图 1　圆形屏幕内容展示示例图

调试应用

在开启应用调试之前，需要在智能穿戴上开启开发者模式。

1. 进入“设置 > 关于手表”，查看智能穿戴设备信息。滑动到“版本号”的位置，点击 3 次，开启开发者模式。

2. 返回设置界面，进入新出现的“开发人员选项”界面，打开“开发人员选项”开关。

部分智能穿戴仅支持无线充电，开发者无法通过 USB 连接方式去开发和调试应用，可以通过 WLAN 进行调试。调试方法如下：

1. 使用路由器设置一个无密码的 WLAN 网络，将开发应用的 PC 接入该路由器。

2. 打开智能穿戴的“设置 > WLAN”，打开 WLAN 开关，将智能穿戴接入上述 WLAN 网络。

3. 进入开发人员选项，查看 IP 地址。

4. 在 PC 端打开 DevEco Studio，在上方导航栏选择“Tools > IP > Connect” 。

5. 在弹出的窗口中，输入智能穿戴的 IP 地址。完成连接，可以开始进行应用调试。

当不需要进行应用调试时，可以进入“设置 > 开发人员选项”，关闭“开发人员选项”开关，退出开发者模式。

2.3 添加智能穿戴模块

以下根据实际的开发样例来展示如何在已有的 HarmonyOS 工程中添加一个智能穿戴模块。

如图所示，这是一个睡眠检测应用，应用分为主界面和详情界面，可以选择使用PageSlider 实现界面间的切换。PageSlider 是一个布局管理器，用于实现左右滑动以及上下滑动的翻页效果。

图 1　开发样例效果图

1. 在 DevEco Studio 上方的导航栏中，依次点击“File > New > Module”，在“Device”中选择“Wearable”，添加一个新模块。

2. 在左侧的“Project”窗口，打开“entry > src > main > resources > base”，右键点击“base”文件夹，选择“New > Directory”，命名为“layout”。

3. 右键点击“layout”文件夹，选择“New > File”，新建两个 UI 布局文件，分别命名为“layout_sleep.xml”和“layout_detail.xml”。

主界面的 UI 布局文件是“layout_sleep.xml”，其完整示例代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<DirectionalLayout xmlns:ohos="http://schemas.huawei.com/res/ohos"
                ohos:width="match_parent"
                ohos:height="match_parent"
                ohos:background_element="#FF000000"
                ohos:orientation="vertical">

    <Image
            ohos:id="$+id:sleep_moon_img"
            ohos:width="46vp"
            ohos:height="46vp"
            ohos:top_margin="11vp"
            ohos:layout_alignment="horizontal_center"/>

    <Text
            ohos:width="match_parent"
            ohos:height="19.5vp"
            ohos:alpha="0.66"
            ohos:layout_alignment="horizontal_center"
            ohos:text_alignment="center"
            ohos:text="$string:sleep"
            ohos:text_color="$color:sleep_text_white"
            ohos:text_size="16vp"/>
```

```
    <DirectionalLayout
xmlns:ohos="http://schemas.huawei.com/res/ohos"
                  ohos:width="match_content"
                  ohos:height="65vp"
                  ohos:top_margin="8vp"
                  ohos:layout_alignment="horizontal_center"
                  ohos:orientation="horizontal">
        <Text
                ohos:id="$+id:sleep_hour_text"
                ohos:width="match_content"
                ohos:height="match_content"
                ohos:layout_alignment="center"
                ohos:text_alignment="center"
                ohos:text="$string:dash"
                ohos:text_color="$color:sleep_text_white"
                ohos:text_size="58vp"
        />
        <Text
                ohos:width="match_content"
                ohos:height="match_content"
                ohos:left_margin="2vp"
                ohos:alpha="0.66"
                ohos:layout_alignment="bottom"
                ohos:bottom_padding="9.5vp"
                ohos:text="$string:hour"
                ohos:text_color="$color:sleep_text_white"
                ohos:text_size="16vp"
        />
```

```
        <Text
                ohos:id="$+id:sleep_min_text"
                ohos:width="match_content"
                ohos:height="match_content"
                ohos:left_margin="2vp"
                ohos:layout_alignment="center"
                ohos:text_alignment="center"
                ohos:text="$string:double_dash"
                ohos:text_color="$color:sleep_text_white"
                ohos:text_size="58vp"
        />
        <Text
                ohos:width="match_content"
                ohos:height="match_content"
                ohos:left_margin="2vp"
                ohos:alpha="0.66"
                ohos:layout_alignment="bottom"
                ohos:bottom_padding="9.5vp"
                ohos:text="$string:minute"
                ohos:text_color="$color:sleep_text_white"
                ohos:text_size="16vp"
        />
    </DirectionalLayout>
    <DirectionalLayout
xmlns:ohos="http://schemas.huawei.com/res/ohos"
                ohos:width="match_content"
                ohos:height="25vp"
                ohos:top_margin="20.5vp"
```

```
                ohos:layout_alignment="horizontal_center"
                ohos:orientation="horizontal">
        <Text
                ohos:width="match_content"
                ohos:height="19.5vp"
                ohos:text="$string:goal"
                ohos:alpha="0.66"
                ohos:text_alignment="bottom"
                ohos:bottom_margin="1vp"
                ohos:text_color="$color:sleep_text_white"
                ohos:text_size="16vp"
        />
        <Text
                ohos:id="$+id:sleep_goal_text"
                ohos:width="match_content"
                ohos:height="match_parent"
                ohos:left_margin="2vp"
                ohos:text="$string:target_sleep_time"
                ohos:text_weight="600"
                ohos:text_color="$color:sleep_text_white"
                ohos:bottom_padding="2vp"
                ohos:text_size="21vp"
        />
        <Text
                ohos:width="match_content"
                ohos:height="19.5vp"
                ohos:left_margin="2vp"
                ohos:alpha="0.66"
```

```
                ohos:text="$string:hour"
                ohos:text_color="$color:sleep_text_white"
                ohos:text_size="16vp"
        />
    </DirectionalLayout>
</DirectionalLayout>
```

详情界面的 UI 布局文件是“layout_detail.xml”，其完整示例代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<DirectionalLayout xmlns:ohos="http://schemas.huawei.com/res/ohos"
            ohos:width="match_parent"
            ohos:height="match_parent"
            ohos:orientation="vertical"
            ohos:background_element="#FF000000">
    <Text
           ohos:id="$+id:detail_nodata_date"
           ohos:width="match_content"
           ohos:height="23vp"
           ohos:top_margin="20vp"
           ohos:layout_alignment="horizontal_center"
           ohos:text_alignment="bottom"
           ohos:text_color="$color:sleep_text_white"
           ohos:text_weight="600"
           ohos:text_size="19vp"/>

    <Image
           ohos:id="$+id:detail_nodata_img"
```

```
            ohos:width="46vp"
            ohos:height="46vp"
            ohos:top_margin="25vp"
            ohos:layout_alignment="horizontal_center"
            ohos:scale_type="scale_to_center"/>

    <Text
            ohos:width="match_content"
            ohos:height="match_content"
            ohos:alpha="0.66"
            ohos:top_margin="12vp"
            ohos:layout_alignment="horizontal_center"
            ohos:text_alignment="center"
            ohos:text="$string:no_data"
            ohos:text_color="$color:sleep_text_white"
            ohos:text_size="16vp"/>
    <Text
            ohos:width="match_content"
            ohos:height="match_content"
            ohos:alpha="0.66"
            ohos:layout_alignment="horizontal_center"
            ohos:text_alignment="center"
            ohos:text="$string:wearing_watch_tips"
            ohos:text_color="$color:sleep_text_white"
            ohos:text_size="16vp"/>
</DirectionalLayout>
```

在左侧项目文件栏中，选择“entry > src > main > java > 应用包名 > slice”，在对应的 AbilitySlice 文件的 onStart 里，使用代码创建 PageSlider，添加这两个相应的界面。

```
public class SleepPageSlice extends AbilitySlice {

    private static final String TAG = "SleepPageSlice";


    private static final HiLogLabel LABEL = new HiLogLabel(HiLog.LOG_APP,
0, TAG);


    private List<ComponentOwner> list = new ArrayList<>();


    private PageSliderProvider provider = new PageSliderProvider() {

        @Override

        public int getCount() {

            return list.size();

        }


        @Override

        public Object createPageInContainer(ComponentContainer
componentContainer, int index) {

            if (index >= list.size() || componentContainer == null) {

                HiLog.error(LABEL, "instantiateItem index error");

                return Optional.empty();

            }

            ComponentOwner container = list.get(index);

            componentContainer.addComponent(container.getComponent());

            container.instantiateComponent();
```

```
            return container.getComponent();
        }

        @Override
        public void destroyPageFromContainer(ComponentContainer
componentContainer, int index, Object object) {
            HiLog.info(LABEL, "destroyItem index:" + index);
            if (index >= list.size() || componentContainer == null) {
                return;
            }
            Component content = list.get(index).getComponent();
            componentContainer.removeComponent(content);
            return;
        }

        @Override
        public boolean isPageMatchToObject(Component component, Object
object) {
            return component == object;
        }

        @Override
        public void startUpdate(ComponentContainer container) {
            super.startUpdate(container);
            HiLog.info(LABEL, "startUpdate");
        }
    };

    @Override
```

```
    public void onStart(Intent intent) {
        super.onStart(intent);
        HiLog.info(LABEL, "onStart");

        // 添加子页面
        list.add(new SleepComponentOwner(this));
        list.add(new DetailComponentOwner(this));

        // 设置主界面
        DirectionalLayout layout = new DirectionalLayout(this);
        ComponentContainer.LayoutConfig config = new
ComponentContainer.LayoutConfig(
                ComponentContainer.LayoutConfig.MATCH_PARENT,
                ComponentContainer.LayoutConfig.MATCH_PARENT);
        layout.setLayoutConfig(config);

        // 使用 PageSlider 做滑动效果
        PageSlider slider = new PageSlider(this);
        ComponentContainer.LayoutConfig sliderConfig = new
ComponentContainer.LayoutConfig(
                ComponentContainer.LayoutConfig.MATCH_PARENT,
                ComponentContainer.LayoutConfig.MATCH_PARENT);
        slider.setLayoutConfig(sliderConfig);
        slider.setOrientation(DirectionalLayout.VERTICAL);
        slider.setProvider(provider);

        layout.addComponent(slider);

        setUIContent(layout);
```

```
    }
}
```

增加 ComponentOwner 容器接口和两个页面的实现方式，如下是容器的接口 ComponentOwner.java。

```
public interface ComponentOwner {
    // 获取存放的 component
    Component getComponent();

    // 当包含的 component 被添加到容器时回调
    void instantiateComponent();
}
```

增加第一个页面 SleepComponentOwner 和第二个页面 DetailComponentOwner，这两个页面从 xml 里加载。以下是首页 SleepComponentOwner 的实现。

```
public class SleepComponentOwner implements ComponentOwner {
    private static final String TAG = "SleepViewContainer";

    private static final HiLogLabel LABEL = new HiLogLabel(HiLog.LOG_APP,
0, TAG);

    // 目标睡眠时长默认值,单位：分钟
    private static final int DEFAULT_SLEEP_TARGET_TIME = 480;

    // 睡眠时长默认值,单位：分钟
    private static final int DEFAULT_SLEEP_TIME = 0;
```

```
    private CircleProgressDrawTask drawTask;

    private AbilityContext myContext;

    private Component root;

    public SleepComponentOwner(AbilityContext context) {
        init(context);
    }

    private void init(AbilityContext context) {
        myContext = context;
        LayoutScatter scatter = LayoutScatter.getInstance(context);
        root = scatter.parse(ResourceTable.Layout_layout_sleep, null,
false);
        drawTask = new CircleProgressDrawTask(root);
        drawTask.setMaxValue(DEFAULT_SLEEP_TARGET_TIME);

        Component imageView =
root.findComponentById(ResourceTable.Id_sleep_moon_img);
        imageView.setBackground(new VectorElement(context,
ResourceTable.Graphic_ic_icon_moon));
    }

    @Override
    public Component getComponent() {
        return root;
    }
```

```
    @Override
    public void instantiateComponent() {
        return;
    }
}
```

以下是第二个页面 DetailComponentOwner 的实现。

```
public class DetailComponentOwner implements ComponentOwner {
    private static final String TAG = "DetailViewContainer";

    private static final HiLogLabel LABEL = new HiLogLabel(HiLog.LOG_APP,
0, TAG);

    private AbilityContext myContext;

    private ComponentContainer root;

    public DetailComponentOwner(AbilityContext context) {
        init(context);
    }

    private void init(AbilityContext context) {
        root = new DirectionalLayout(context);
        ComponentContainer.LayoutConfig config = new
ComponentContainer.LayoutConfig(
                ComponentContainer.LayoutConfig.MATCH_PARENT,
                ComponentContainer.LayoutConfig.MATCH_PARENT);
        root.setLayoutConfig(config);
```

```
        myContext = context;
    }

    @Override
    public Component getComponent() {
        return root;
    }

    @Override
    public void instantiateComponent() {
        return;
    }
}
```

增加一个类型为 Page 的 Ability，并在 config.json 里进行注册。需要在 onStart 里调用 setSwipeToDismiss(true)，来设置右滑退出。示例代码如下：

```
public class PageAbility extends Ability {
    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
        super.setMainRoute(SleepPageSlice.class.getName());
        setSwipeToDismiss(true);
    }
}
```

如下是配置文件 config.json，注册 PageAbility 的时候，需要指明 action.system.home，用来保证该 Ability 能在 launcher 上显示对应的图标。

```
{
    "app": {
        "bundleName": "com.huawei.health.sleep",
        "vendor": "huawei",
        "version": {
            "code": 1,
            "name": "1.0.8.27"
        },
        "apiVersion: {
            "compatible": 3,
            "target": 3
        }
    },
    "deviceConfig": {
        "default": {
            }
        }
    },
    "module": {
        "package": "com.huawei.health.sleep",
        "name": ".SleepApplication",
        "distro": {
            "moduleType": "entry",
            "deliveryWithInstall": true,
            "moduleName": "entry"
        },
        "deviceType": [
            "wearable"
```

```
        ],
        "reqCapabilities": [
            "video_support"
        ],
        "abilities": [
            {
                "name": ".PageAbility",
                "description": "$string:ability_description",
                "icon": "$media:icon_app",
                "label": "$string:app_name",
                "launchType": "standard",
                "orientation": "portrait",
                "visible": true,
                "permissions": [],
                "skills": [
                    {
                        "actions": [
                            "action.system.home"
                        ],
                        "entities": [
                            "entity.system.home"
                        ],
                    }
                ],
                "type": "page",
                "formEnabled": false
            }
        ]
```

```
    }
}
```

在睡眠界面中，我们用到了圆环效果，这里我们看一下圆环效果是如何实现的，如何实现自定义 Component 的效果。调用方代码如下：

```
drawTask = new CircleProgressDrawTask(root);
```

Componet 类提供了 UI 的基本组件，包括方法 addDrawTask(Component.DrawTask task)。该方法可以给任意一个 Componet 添加一段自定义绘制的代码。自定义 Component 的实现方法如下：

1. 创建一个自定义 DrawTask，包含与该 Componet 相关的自定义属性和绘制的代码。
2. 在构造方法里传入宿主 Component，跟自定义的 DrawTask 绑定。

实现睡眠圆环效果的示例代码如下。

```
public class CircleProgressDrawTask implements Component.DrawTask {
    // 用于配置圆环的粗细，具体参数可以在 xml 文件中配置
    private static final String STROKE_WIDTH_KEY = "stroke_width";

    // 用于配置圆环的最大值，具体参数可以在 xml 文件中配置
    private static final String MAX_PROGRESS_KEY = "max_progress";

    // 用于配置圆环的当前值，具体参数可以在 xml 文件中配置
```

```
    private static final String CURRENT_PROGRESS_KEY =
"current_progress";

    // 用于配置起始位置的颜色，具体参数可以在 xml 文件中配置
    private static final String START_COLOR_KEY = "start_color";

    // 用于配置结束位置的颜色，具体参数可以在 xml 文件中配置
    private static final String END_COLOR_KEY = "end_color";

    // 用于配置背景色，具体参数可以在 xml 文件中配置
    private static final String BACKGROUND_COLOR_KEY =
"background_color";

    // 用于配置起始位置的角度，具体参数可以在 xml 文件中配置
    private static final String START_ANGLE = "start_angle";

    private static final float MAX_ARC = 360f;

    private static final int DEFAULT_STROKE_WIDTH = 10;

    private static final int DEFAULT_MAX_VALUE = 100;

    private static final int DEFAULT_START_COLOR = 0xFFB566FF;

    private static final int DEFAULT_END_COLOR = 0xFF8A2BE2;

    private static final int DEFAULT_BACKGROUND_COLOR = 0xA8FFFFFF;

    private static final int DEFAULT_START_ANGLE = -90;
```

```
    private static final float DEFAULT_LINER_MAX = 100f;

    private static final int HALF = 2;

    // 圆环的宽度，默认 10 个像素
    private int myStrokeWidth = DEFAULT_STROKE_WIDTH;

    // 最大的进度值，默认是 100
    private int myMaxValue = DEFAULT_MAX_VALUE;

    // 当前的进度值，默认是 0
    private int myCurrentValue = 0;

    // 起始位置的颜色，默认浅紫色
    private Color myStartColor = new Color(DEFAULT_START_COLOR);

    // 结束位置的颜色，默认深紫色
    private Color myEndColor = new Color(DEFAULT_END_COLOR);

    // 背景颜色，默认浅灰色
    private Color myBackgroundColor = new
Color(DEFAULT_BACKGROUND_COLOR);

    // 当前的进度值，默认从-90 度进行绘制
    private int myStartAngle = DEFAULT_START_ANGLE;

    private Component myComponent;
```

```
    // 传入要进行修改的 component
    public CircleProgressDrawTask(Component component) {
        myComponent = component;
        myComponent.addDrawTask(this);
    }

    // 设置当前进度并且刷新 component，value 值为当前进度
    public void setValue(int value) {
        myCurrentValue = value;
        myComponent.invalidate();
    }

    public void setMaxValue(int maxValue) {
        myMaxValue = maxValue;
        myComponent.invalidate();
    }

    @Override
    public void onDraw(Component component, Canvas canvas) {
        // 通过 canvas 实现绘制圆环的功能
        ......
    }
}
```

2.4 创建智能穿戴应用通知

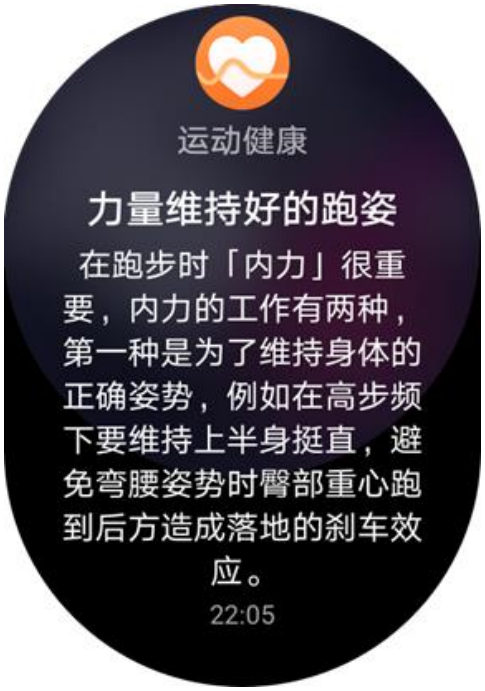
2.4.1 介绍

HarmonyOS 提供了通知功能，提醒用户有来自应用的信息。当应用向系统发出通知时，通知将以弹窗的形式显示，并同时出现在通知中心。用户可以在表盘界面上拉，通过通知中心查看通知的详细信息。常见的使用场景有：

- 显示接收到短消息、即时消息等。
- 显示应用的推送消息，如广告、版本更新等。
- 显示当前正在进行的事件，如勿扰模式等。

通知的样式

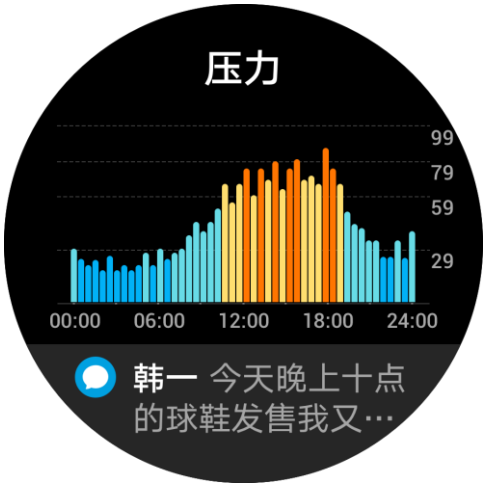
- 当在表盘界面收到消息通知，会全屏显示，示例如下：



- 在表盘界面上拉可以打开通知中心，消息通知会自动按应用维度进行聚合展示，示例如下:

- 聚合的消息会有小红点提示折叠的消息数量。点击消息可打开折叠的消息详情，示例如下：

- 当在非表盘界面收到消息通知，会在屏幕下方弹出浮窗显示，示例如下：



约束与限制

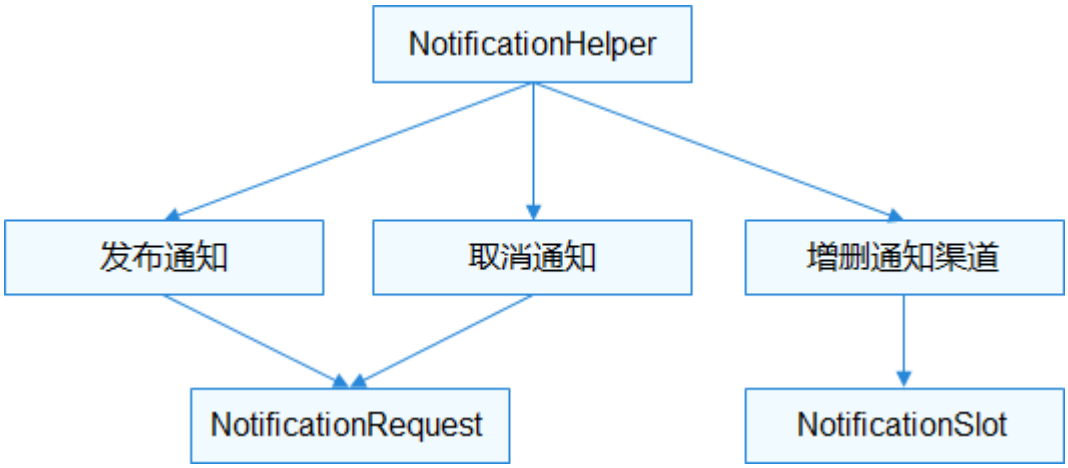
- 通知目前支持三种样式：普通文本、长文本、图片。
- 通知不支持快捷回复，不支持自定义布局。
- 目前通知订阅不支持多用户。
- 通知的订阅目前仅支持系统应用，不支持第三方应用。

2.4.2 开发指导

接口介绍

通知相关基础类包含 NotificationSlot、NotificationRequest 和 NotificationHelper。详细的接口信息请参考通知开发指导。基础类之间的关系如下所示：

图 1　通知基础类关系图



- **NotificationSlot**

NotificationSlot 可以对提示音、振动和重要级别等进行设置。一个应用可以创建一个或多个 NotificationSlot，在发送通知时，通过绑定不同的 NotificationSlot，实现不同用途。

说明

NotificationSlot 需要先通过 NotificationHelper 的 addNotificationSlot(NotificationSlot)方法发布后，通知才能绑定使用；所有绑定该 NotificationSlot 的通知在发布后都具备相应的特性，对象在创建后，将无法更改其设置属性，对于是否启动相应设置，用户有最终控制权。

不指定 NotificationSlot 时，当前通知会使用默认的 NotificationSlot，默认的 NotificationSlot 优先级为 LEVEL_DEFAULT，声音为系统默认提示音。

NotificationSlot 的级别目前支持如下几种：

- LEVEL_NONE： 表示通知不发布。
- LEVEL_MIN/LEVEL_LOW/LEVEL_DEFAULT/LEVEL_HIGH：表示通知发布后可在通知中心显示，自动弹出，触发提示音。

NotificationRequest

NotificationRequest 用于设置具体的通知对象，包括设置通知的属性，如：通知的小图标、自动删除等参数，以及设置具体的通知类型，如普通文本、长文本等。

通知的常用属性：

- 小图标

标识说明：1为通过 NotificationRequest.setLittleIcon(PixelMap)设置的小图标。

- 从通知启动 Ability：点击通知栏的通知，可以通过启动 Ability，触发新的事件。

通知设置 NotificationRequest 的 setIntentAgent(IntentAgent)后，点击通知栏上发布的通知，将触发通知中的 IntentAgent 承载的事件。IntentAgent 的设置请参考 IntentAgent 开发指导。

- 通知设置 ActionButton：通过点击通知按钮，可以触发按钮承载的事件。

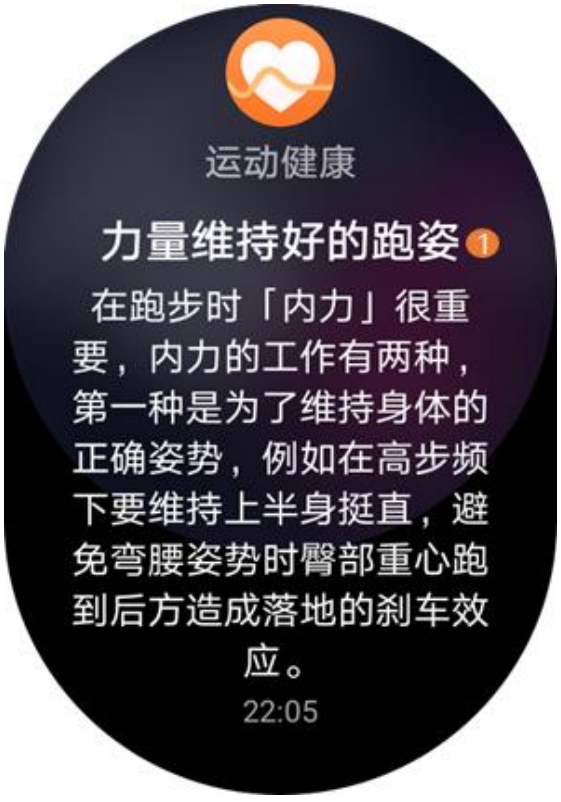
标识说明：1、2为两个通过NotificationRequest.addActionButton(NotificationActionButton)设置的通知附加按钮，点击按钮后可以触发相关的事件，具体事件内容如何设置需要参考NotificationActionButton。

具体的通知类型：目前支持三种类型，包括普通文本NotificationNormalContent、长文本NotificationLongTextContent、图片NotificationPitctureContent。

- 普通文本通知样式（NotificationNormalContent）

标识说明：1为通知的标题，通过 NotificationNormalContent.setTitle(String)设置。2为通知的内容，通过 NotificationNormalContent.setText(String)设置。通知标题和内容至少要设置一个。

- 长文本通知样式（NotificationLongTextContent）



标识说明：1为通知的长文本，通过NotificationLongTextContent.setLongText(String)设置，文本长度最大支持 1024 个字符。

- 图片通知样式（NotificationPictureContent）

标识说明：1为图片通知样式的图片，通过NotificationPictureContent.setBigPicture(PixelMap bigPicture)设置。

说明

通知发布后，通知的设置不可修改。如果下次发布通知使用相同的 ID，就会更新之前发布的通知。

- **NotificationHelper**

NotificationHelper 封装了发布、更新、订阅、删除通知等静态方法。订阅通知、退订通知和查询系统中所有处于活跃状态的通知，有权限要求需为系统应用或具有订阅者权限。

开发步骤

通知的开发指导分为创建 NotificationSlot、发布通知和取消通知等开发场景。

创建 NotificationSlot

NotificationSlot 可以设置公共通知的提示声等，并通过调用 NotificationHelper.addNotificationSlot()发布 NotificationSlot 对象。

```
// 创建 notificationSlot 对象
NotificationSlot slot = new NotificationSlot("slot_001","slot_default",
NotificationSlot.LEVEL_DEFAULT);
slot.setDescription("NotificationSlotDescription");
try {
    NotificationHelper.addNotificationSlot(slot);
} catch (RemoteException ex) {
    HiLog.warn(LABEL, "addNotificationSlot occur exception.");
}
```

发布通知

1. 构建 NotificationRequest 对象，应用发布通知前，通过 NotificationRequet 的 setSlot()方法与 NotificationSlot 绑定，使该通知在发布后都具备该对象的特征。

```
int notification_id = 1;
NotificationRequest request = new NotificationRequest(notification_id);
request.setSlotId(slot.getId());
```

调用 setContent()设置通知的内容。

```
String title = "title";
String text = "There is a normal notification content.";
NotificationRequest.NotificationNormalContent content = new
NotificationRequest.NotificationNormalContent();
content.setTitle(title)
       .setText(text);
NotificationRequest.NotificationContent notificationContent = new
NotificationRequest.NotificationContent(content);
// 设置通知的小图标
request.setLittleIcon(PixelMap);
// 设置通知的内容
request.setContent(notificationContent);
```

调用 setIntentAgent()设置通知可以触发的事件。

```
// 指定要启动的 ability 的 ElementName 字段
ElementName elementName = new ElementName("",
"com.example.testintentagent",
"com.example.testintentagent.IntentAgentAbility");
// 将 ElementName 字段添加到 Intent 中
Intent intent = new Intent();
intent.setElement(elementName);
List<Intent> intentList = new ArrayList<>();
intentList.add(intent);
```

```
// 指定启动一个有页面的 ability
IntentAgentInfo intenAgentinfo = new
IntentAgentInfo(request.getNotificationId(),
IntentAgentConstant.OperationType.START_ABILITY,
IntentAgentConstant.Flags.UPDATE_PRESENT_FLAG, intentList, null);
// 获取 IntentAgent 实例
IntentAgent intentAgent = IntentAgentHelper.getIntentAgent(mContext,
intenAgentinfo);
request.setIntentAgent(intentAgent);
request.setTapDismissed(true);
```

调用 publishNotification()发送通知。

```
try {
   NotificationHelper.publishNotification(request);
} catch (RemoteException ex) {
   HiLog.warn(LABEL, "publishNotification occur exception.");
}
```

取消通知

取消通知分为取消指定单条通知和取消所有通知，应用只能取消自己发布的通知。

- 调用 cancelNotification()取消指定的单条通知。

```
int notification_id = 1;
try {
    NotificationHelper.cancelNotification(notification_id);
} catch (RemoteException ex) {
    HiLog.warn(LABEL, "cancelNotification occur exception.");
```

```
}
```

2.5 降低应用功耗

当需要针对智能穿戴开发低功耗应用时，推荐开发者使用深色主题模式。

示例如下：

开发注意事项

智能穿戴电池容量有限，为了让应用对用户更友好，开发者应当尽可能降低应用的功耗开销。以下注意事项供开发者参考：

1. 避免长时间的屏幕常亮从而阻止系统休眠：除视频、游戏、导航等用户可感知的业务场景外，原则上应用不允许做屏幕常亮的设计。同时，禁止任何后台应用设置屏幕常亮。

2. 灭屏情况下，避免频繁唤醒系统：心跳类唤醒系统的频率建议每小时不超过 12 次，闹钟、日程提醒、邮件、IM 类应用按需唤醒系统，其他类的应用禁止唤醒系统。

3. 避免应用频繁自启：除被前台应用拉起的情况外，闹钟、日程提醒、邮件、IM 类应用按需自启，其他类的应用禁止自启。

4. 应用不应在后台长时间使用 GPS：除导航类、轨迹跟踪类、运动健康类应用，禁止非用户可感知业务进行后台定位。

5. 除用户可感知的业务外，禁止应用在后台造成 CPU 高负载耗电。

6. 除用户可感知的业务外，禁止应用在灭屏状态下长时间进行网络定位。

7. 除用户可感知的业务外，禁止应用 WLAN 在后台长时间处于扫描状态。

8. 除用户可感知的业务外，禁止应用在灭屏时后台频繁收发数据。

9. 除用户可感知的业务外，禁止应用运行不必要的后台服务，后台服务会被系统管控和约束。

3.智慧屏

3.1 概述

基于 HarmonyOS，开发者可以开发智慧屏应用，提供丰富的分布式体验。应用可以通过 HarmonyOS 的 API 实现摄像头拍摄、多模输入、分布式应用等能力，典型场景包括：

- 摄像头拍摄：示例参见相机。
- 多模输入：示例参见多模输入。
- 分布式应用：示例参见分布式任务调度。

约束与限制

- 智慧屏是依靠遥控器操作的设备，在智慧屏上应当始终存在一个焦点，告知用户当前可操作的位置。当焦点变化时，应当有明显的动效反馈。
- 智慧屏应用的交互设计必须按照表 1 中的要求来响应遥控器的操作。其中，“-”表示长按无特殊的功能设置，效果仅相当于连续多次点击该按键。

表 1 标准遥控器按键规范

按键	点击	长按
方向键/滑动触摸板	移动控制焦点。	-
确认键	进入当前获焦内容。	-
返回键	返回上一级。	-
首页键	返回首页。	调出多任务。
音量键	调节音量大小。	-
菜单键	调出对应界面的功能菜单。	调出控制中心。
开关机键	打开或关闭智慧屏。	-

工具

工具简介

HUAWEI DevEco Studio（以下简称 DevEco Studio）是基于 IntelliJ IDEA Community 开源版本打造，面向华为终端全场景多设备的一站式集成开发环境（IDE），为开发者提供工程模板创建、开发、编译、调试、发布等 E2E 的 HarmonyOS 应用开发服务。通过使用 DevEco Studio，开发者可以更高效的开发具备 HarmonyOS 分布式能力的应用，进而提升创新效率。

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：

- **多设备统一开发环境**：支持多种 HarmonyOS 设备的应用开发，包括智慧屏、智能穿戴，轻量级智能穿戴设备。
- **支持多语言的代码开发和调试**：包括 Java、XML（Extensible Markup Language）、C/C++ 、 JS（JavaScript）、CSS（Cascading Style Sheets）和 HML（HarmonyOS Markup Language）。
- **支持 FA（Feature Ability）和 PA（Particle Ability）快速开发**：通过工程向导快速创建 FA/PA 工程模板，一键式打包成 HAP（HarmonyOS Ability Package）。
- **支持多设备模拟器**：提供多设备的模拟器资源，包括智慧屏、智能穿戴等设备的模拟器，方便开发者高效调试。

快速开始

下载与安装软件

运行环境要求

当前 DevEco Studio 只支持 Windows 版本，为保证 DevEco Studio 正常运行，建议您的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

下载和安装 DevEco Studio

DevEco Studio 的编译构建依赖 JDK，DevEco Studio 预置了 Open JDK，版本为 1.8，安装过程中会自动安装 JDK。

登录 HarmonysOS 应用开发门户，点击右上角**注册**按钮，注册开发者帐号，注册指导参考注册华为帐号。如果已有华为开发者帐号，请直接点击**登录**按钮。

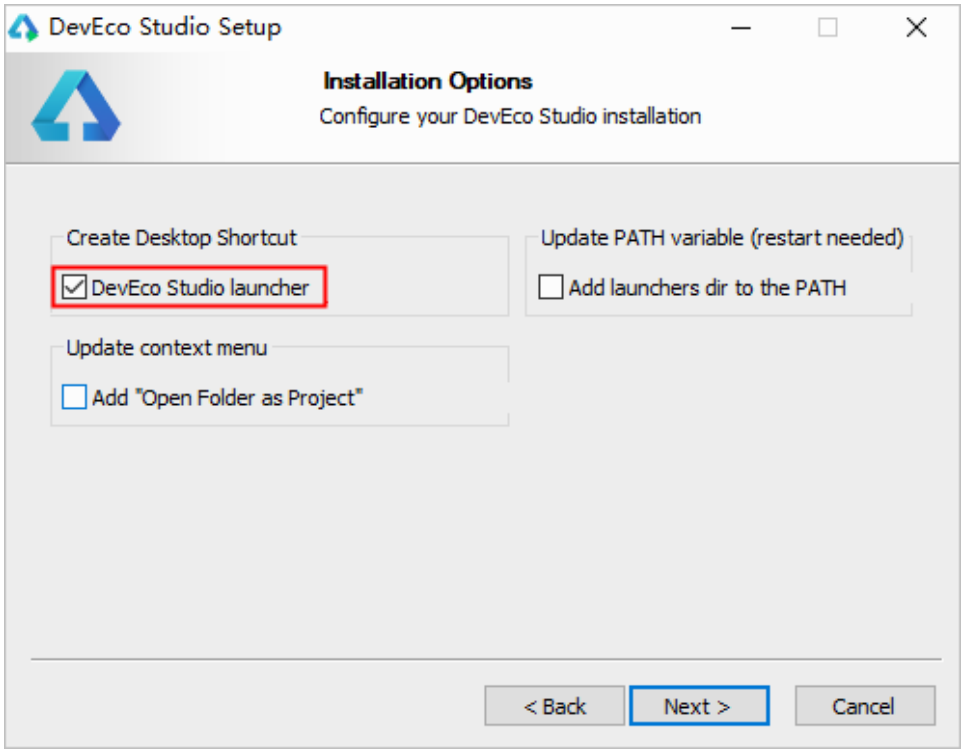
说明

使用 DevEco Studio 远程模拟器需要华为帐号进行实名认证，建议在注册华为帐号后，立即提交实名认证审核，认证方式包括“个人实名认证”和“企业实名认证”，详情请参考实名认证。

1.

2. 进入 HUAWEI DevEco Studio 产品页，下载 DevEco Studio 安装包。

3. 双击下载的“deveco-studio-xxxx.exe”，进入 DevEco Studio 安装向导，在如下安装选项界面勾选 **DevEco Studio launcher** 后，点击 **Next**，直至安装完成。



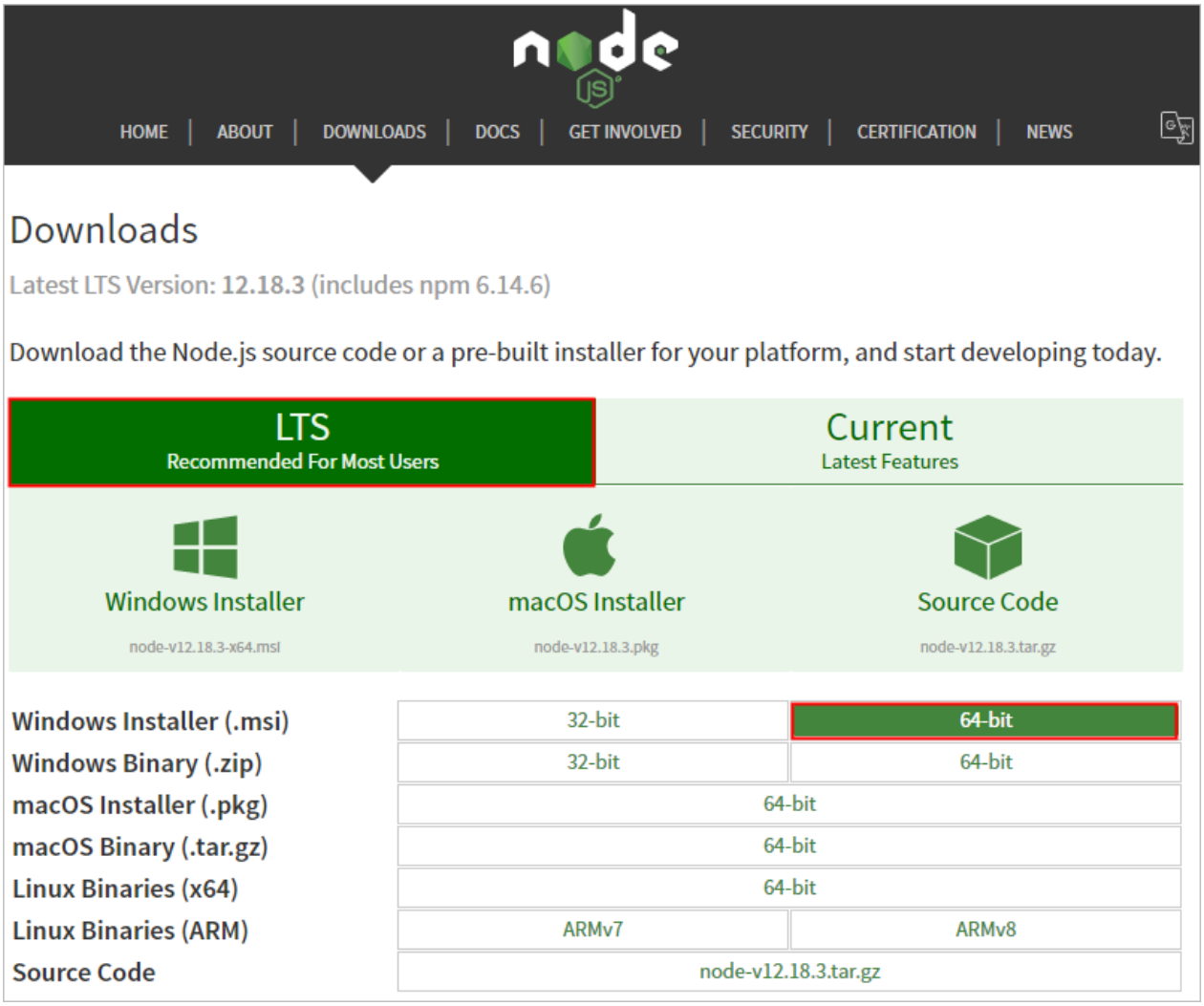
下载和安装 Node. js

Node.js 软件仅在使用到 JS 语言开发 HarmonyOS 应用时才需要安装。使用其它语言开发，不用安装 Node.js，请跳过此章节。

说明

如果已安装 Node.js，打开命令行工具，输入 **node -v** 命令，检查版本号信息，建议使用 V12.0.0 及以上版本。

1. 登录 Node.js 官方网站，下载 Node.js 软件包。请选择 LTS 版本，Windows 64 位对应的软件包。



2.点击下载后的软件包进行安装，全部按照默认设置点击 **Next**，直至 **Finish**。安装过程中，Node.js 会自动在系统的 path 环境变量中配置 node.exe 的目录路径。

软件安装完成后，接下来请配置开发环境。

配置开发环境

DevEco Studio 开发环境需要依赖于网络环境，需要连接上网络才能确保工具的正常使用，可以根据如下两种情况来配置开发环境：

- 如果可以直接访问 Internet，只需进行设置 npm 仓库和下载 HarmonyOS SDK 操作。
- 如果网络不能直接访问 Internet，需要通过代理服务器才可以访问，请根据本章节内容逐条设置开发环境。

npm 设置

设置 npm 代理

只有在同时满足以下两个条件时，需要配置 npm 代理，否则，请跳过本章节。

- 需要使用 JS 语言开发 HarmonyOS 应用。
- 网络不能直接访问 Internet，而是需要通过代理服务器才可以访问。这种情况下，配置 npm 代理，便于从 npm 服务器下载 JS 依赖。

打开命令行工具，按照如下方式进行 npm 代理设置和验证。

1. 执行如下命令设置 npm 代理。

- 如果使用的代理服务器需要认证，请按照如下方式进行设置（请将 **user**、**password**、**proxyserver** 和 **port** 按照实际代理服务器进行修改）。
-

```
.npm config set proxy http://user:password@proxyserver:port
.npm config set https-proxy http://user:password@proxyserver:port
```

- 如果使用的代理服务器不需要认证（不需要帐号和密码），请按照如下方式进行设置。
-

```
.npm config set proxy http:proxyserver:port
.npm config set https-proxy http:proxyserver:port
```

2. 代理设置完成后，执行如下命令进行验证。
3.

```
.npm info express
```

执行结果如下图所示，则说明代理设置成功。

设置 npm 仓库

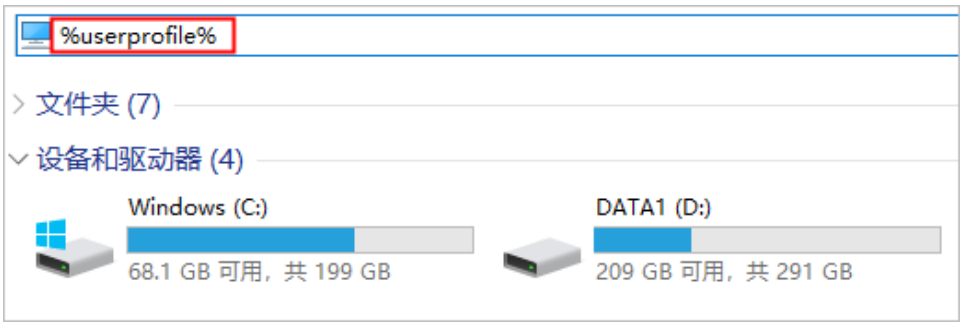
为了提升下载 JS SDK 时，使用 npm 安装 JS 依赖的速度，建议在命令行工具中执行如下命令，重新设置 npm 仓库地址。

```
npm config set registry https://mirrors.huaweicloud.com/repository/npm/
```

设置 Gradle 代理

如果网络不能直接访问 Internet，而是需要通过代理服务器才可以访问，这种情况下，需要设置 Gradle 代理，来访问和下载 Gradle 所需的依赖。否则，请跳过本章节。

打开“此电脑”，在文件夹地址栏中输入**%userprofile%**，进入个人数据界面。



1. 创建一个文件夹，命名为**.gradle.**。如果已有.gradle 文件夹，请跳过此操作。
2. 进入.gradle 文件夹，新建一个文本文档，命名为 **gradle**，并修改后缀为**.properties**。
3. 打开 **gradle.properties** 文件中，添加如下脚本，然后保存。

其中代理服务器、端口、用户名、密码和不使用代理的域名，请根据实际代理情况进行修改。其中不使用代理的 “nonProxyHosts”的配置间隔符是 “|”。

.

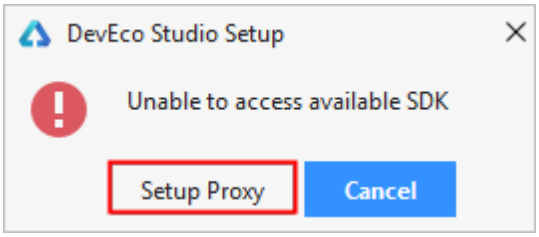
```
.systemProp.http.proxyHost=proxy.server.com
.systemProp.http.proxyPort=8080
.systemProp.http.nonProxyHosts=*.company.com|10.*|100.*
.systemProp.http.proxyUser=userId
.systemProp.http.proxyPassword=password
.systemProp.https.proxyHost=proxy.server.com
.systemProp.https.proxyPort=8080
.systemProp.https.nonProxyHosts=*.company.com|10.*|100.*
.systemProp.https.proxyUser=userId
.systemProp.https.proxyPassword=password
```

设置 DevEco Studio 代理

如果网络不能直接访问 Internet，而需要通过代理服务器才可以访问，这种情况下，需要设置 DevEco Studio 代理，来访问和下载外部资源。否则，请跳过本章节。

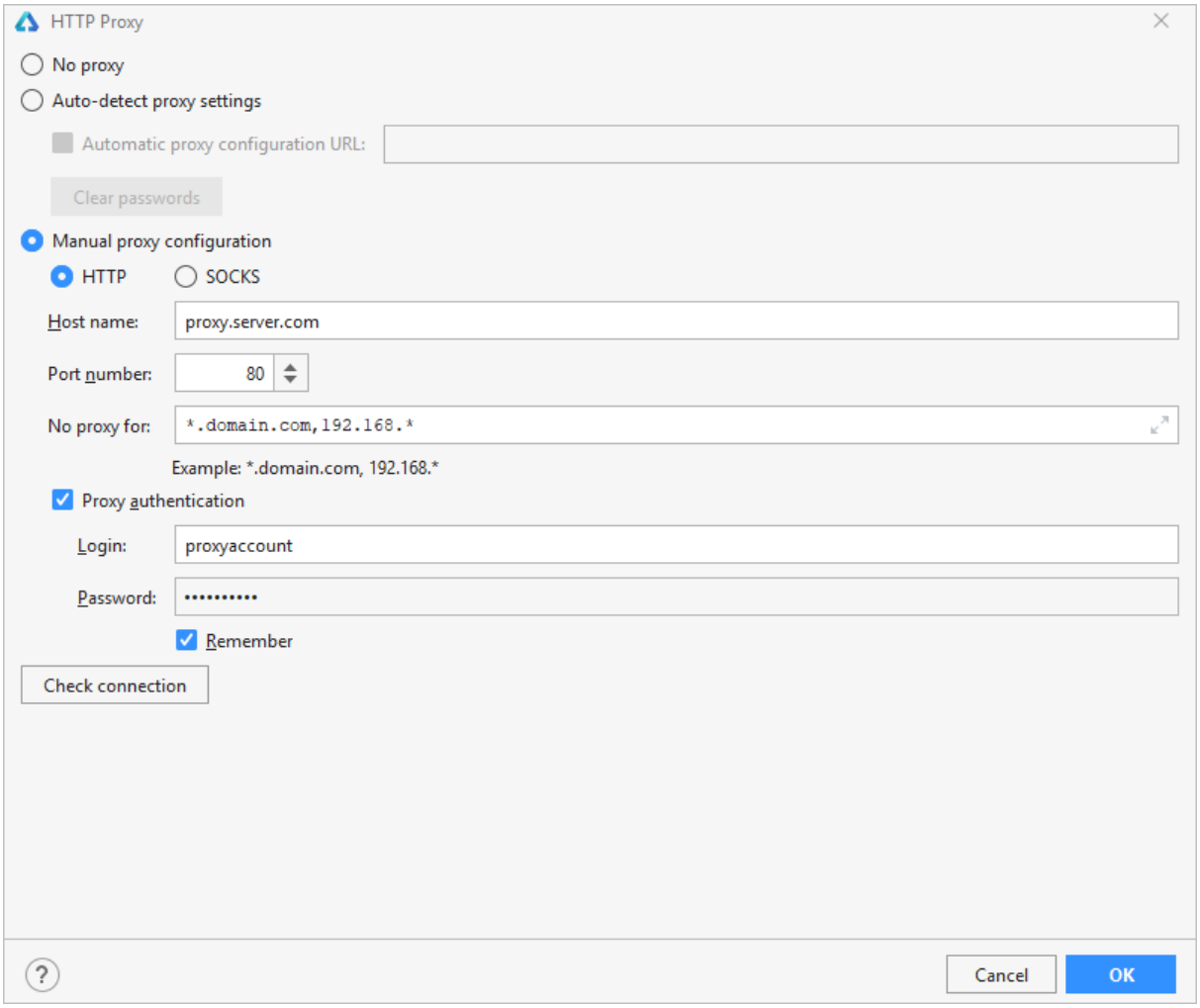
1. 运行已安装的 DevEco Studio，首次使用，请选择 **Do not import settings**，点击 **OK**。

2. 根据 DevEco Studio 欢迎界面的提示，点击 **Setup Proxy**，或者在欢迎页点击 **Configure > Settings > Appearance&Behavior > System Settings > HTTP Proxy** 进入 HTTP Proxy 设置界面。



设置 DevEco Studio 的 HTTP Proxy 信息。

- **HTTP** 配置项，设置代理服务器信息。
- **Host name**：代理服务器主机名或 IP 地址。
- **Port number**：代理服务器对应的端口号。
- **No proxy for**：不需要通过代理服务器访问的 URL 或者 IP 地址（地址之间用英文逗号分隔）。
- **Proxy authentication** 配置项，如果代理服务器需要通过认证鉴权才能访问，则需要设置。否则，请跳过该配置项。
- **Login**：访问代理服务器的用户名。
- **Password**：访问代理服务器的密码。
- **Remember**：勾选，记住密码。

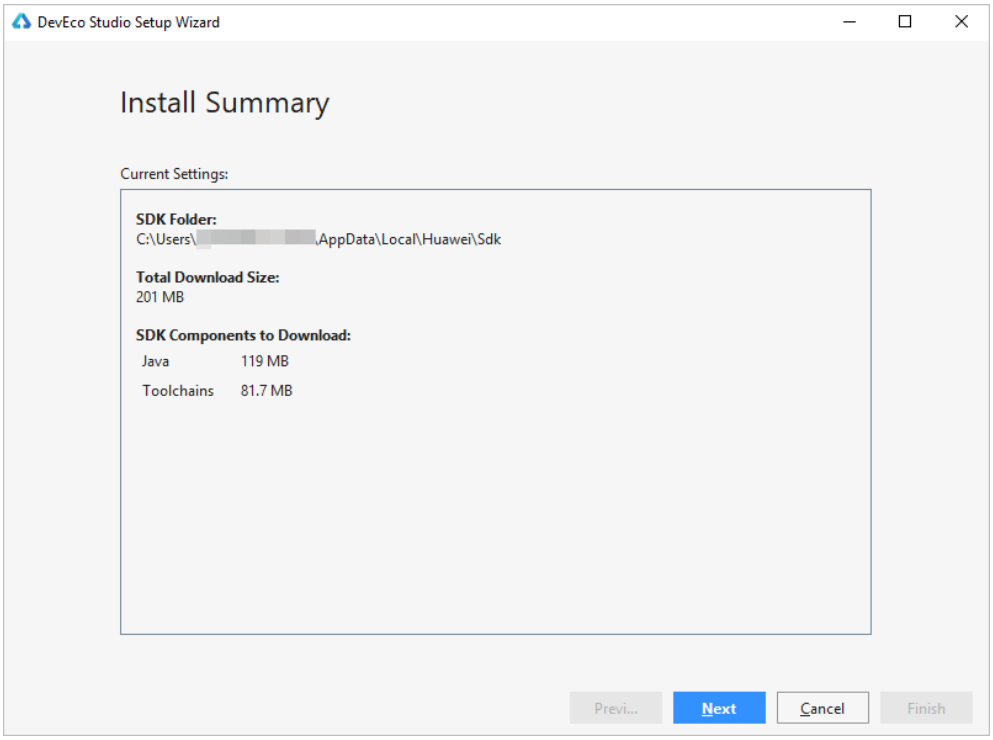


1. 配置完成后，点击 **Check connection**，输入网络地址（如：https://developer.harmonyos.com），检查网络连通性。提示“Connection successful”表示代理设置成功。

2. 点击 **OK** 按钮完成 DevEco Studio 代理配置。

3. DevEco Studio 代理设置完成后，会提示安装 HarmonyOS SDK，可以点击 **Next** 下载到默认目录中；如果想更改 SDK 的存储目录，请点击 **Cancel**，并根据下载 HarmonyOS SDK 进行操作。

4.



下载 HarmonyOS SDK

Devco Studio 提供 SDK Manager 统一管理 SDK 及工具链，下载各种编程语言的 SDK 包时，SDK Manager 会自动下载该 SDK 包依赖的工具链。

SDK Manager 提供多种编程语言的 SDK 包，各 SDK 包的说明请参考：

- **Native**：C/C++语言 SDK 包，默认不自动下载，需手动勾选下载。对应的接口文档请参考《Native API 参考》。
- **JS**：JS 语言 SDK 包，默认不自动下载，需手动勾选下载。对应的接口文档请参考《JS API 参考》。

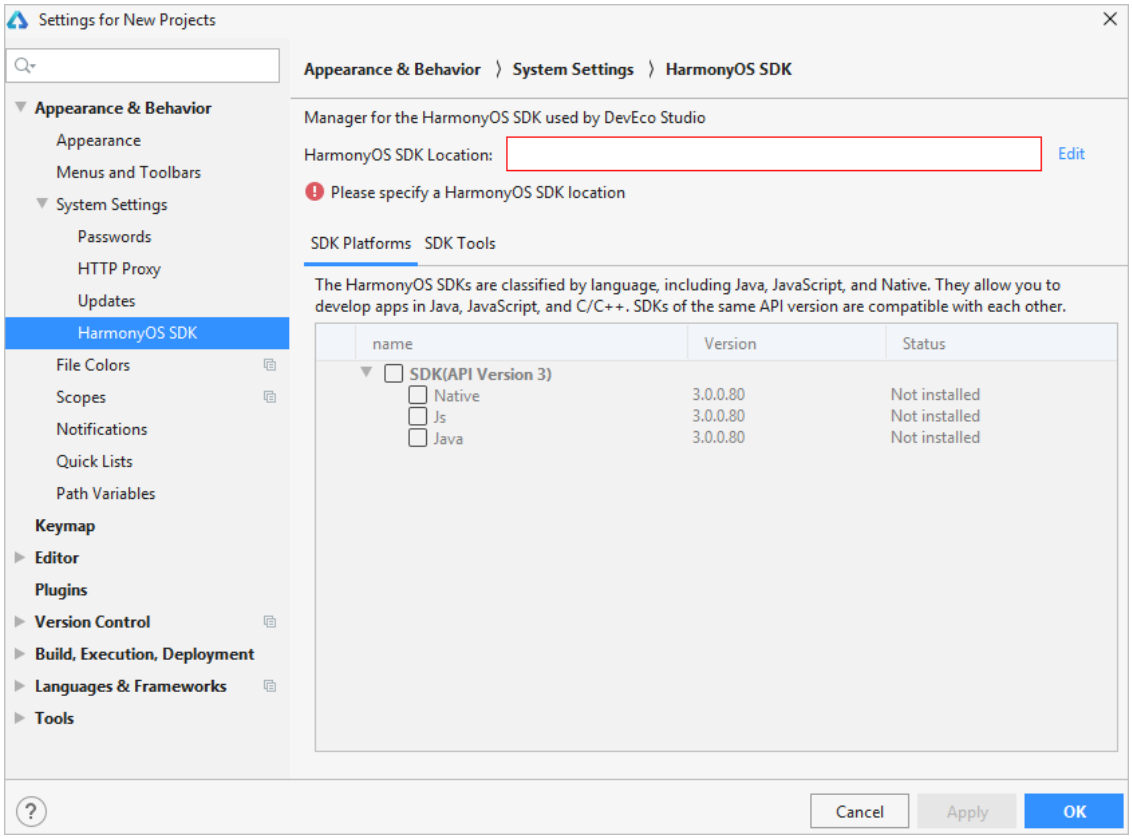
- **Java**：Java 语言 SDK 包，首次下载 SDK 时默认下载。对应的接口文档请参考《Java API 参考》。

同时还提供 SDK 对应的工具链（SDK Tools）：

- **Toolchains**：SDK 工具链，HarmonyOS 应用开发必备工具集，包括编译、打包、签名、数据库管理等工具的集合，首次下载 SDK 时默认下载。
- **Previewer**：Lite Wearable 预览器，在开发过程中可以动态预览 Lite Wearable 应用的界面呈现效果，默认不自动下载，需手动勾选下载。

首次下载 HarmonyOS SDK 时，只会**默认下载 Java SDK 和 Toolchains**。因此，如果还需要使用 JS 或 C/C++语言开发应用时，需手动下载对应的 SDK 包。

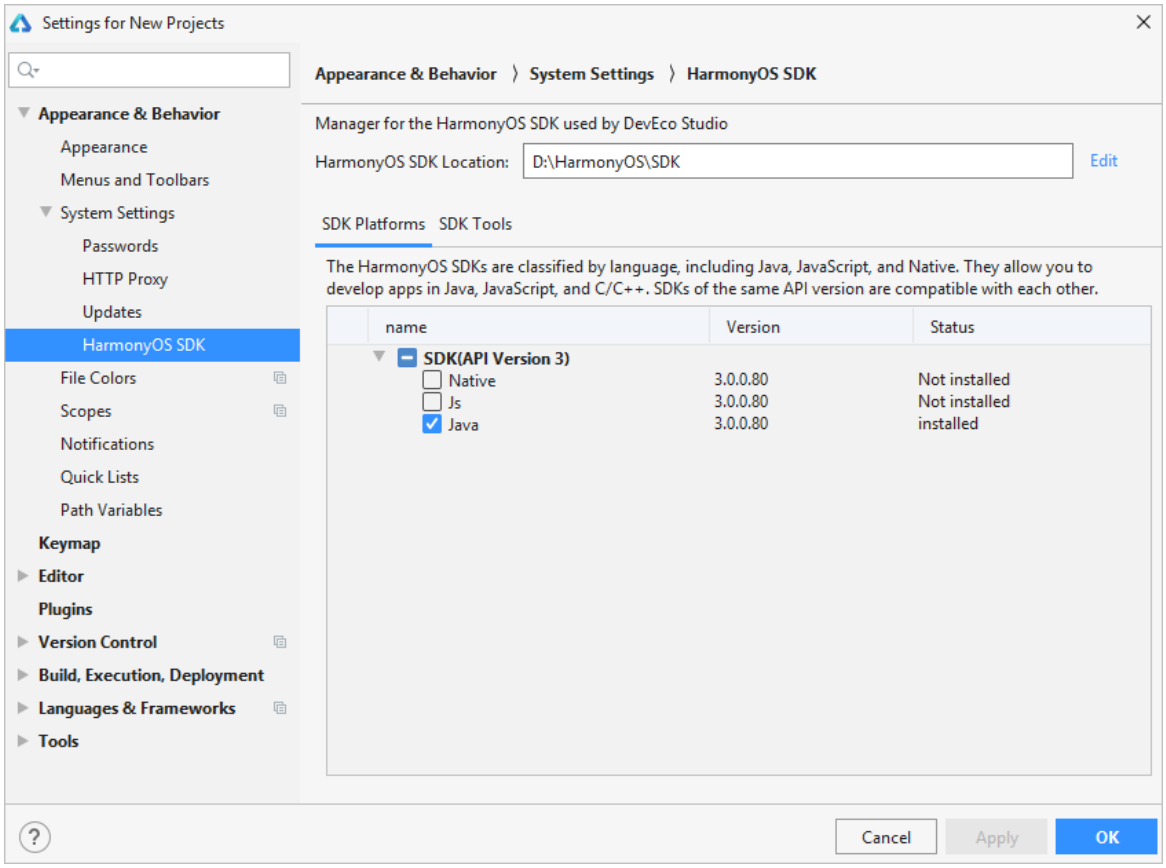
1. 在菜单栏点击 **Configure > Settings** 或者默认快捷键 **Ctrl+Alt+S**，打开 Settings 配置界面。
2. 进入 **Appearance&Behavior > System Settings > HarmonyOS SDK** 菜单界面，点击 **Edit** 按钮，设置 HarmonyOS SDK 存储路径。



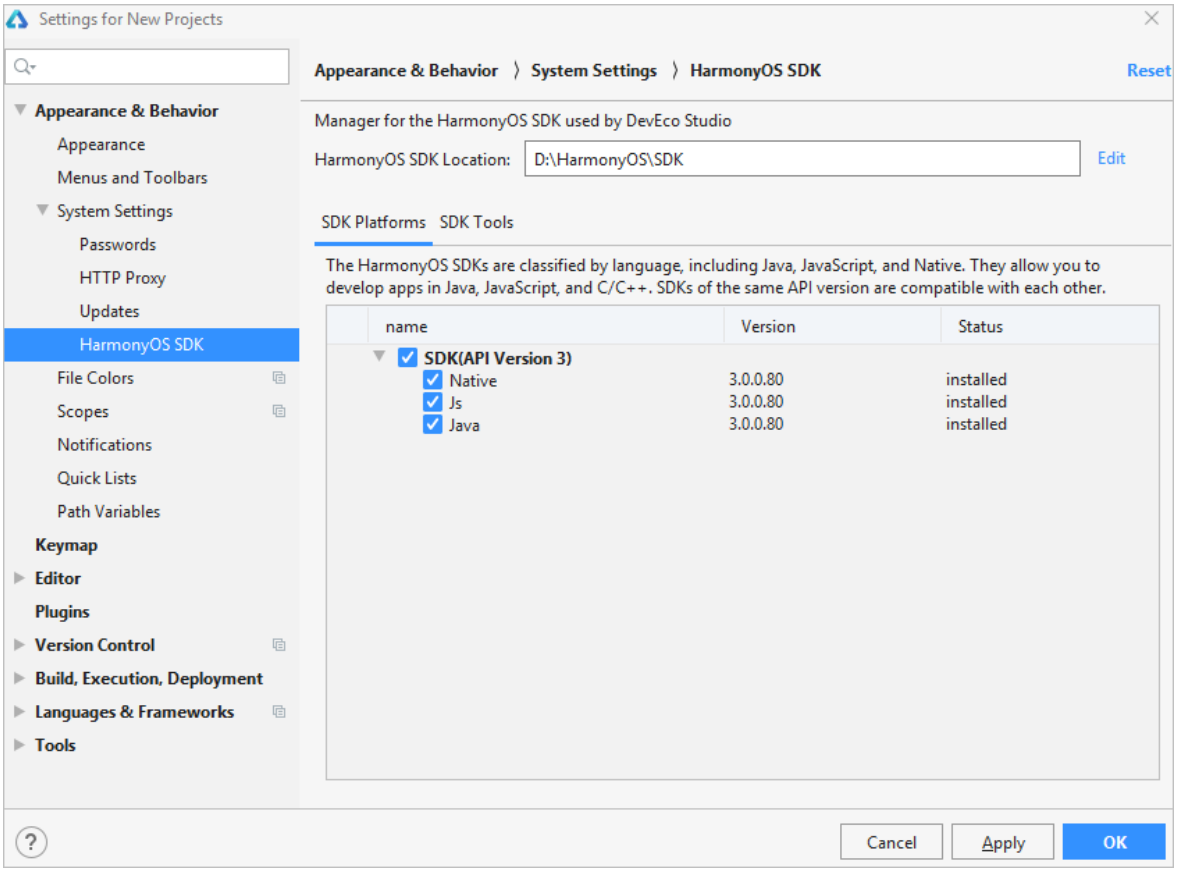
选择 HarmonyOS SDK 存储路径(不能包含中文),然后点击 **Next**。在弹出的 **License Agreement** 窗口，点击 **Accept** 开始下载 SDK。

> 说明
>
> 如果本地已有 SDK 包，请选择本地已有 SDK 包的存储路径，DevEco Studio 会增量更新 SDK 及工具链。

1. 等待 HarmonyOS SDK 及工具下载完成，点击 **Finish**，可以看到默认的 SDK Platforms > **Java** SDK 及 SDK Tools > **Toolchains** 已完成下载。



1. 如果工程还会用到 **JS** 或者 **C/C++**语言，请在 SDK Platform 中，勾选对应的 SDK 包，点击 **Apply**，SDK Manager 会自动将 SDK 包和工具链，下载到 3 中设置的 SDK 存储路径中。（JS SDK 下载失败或者缓慢？）



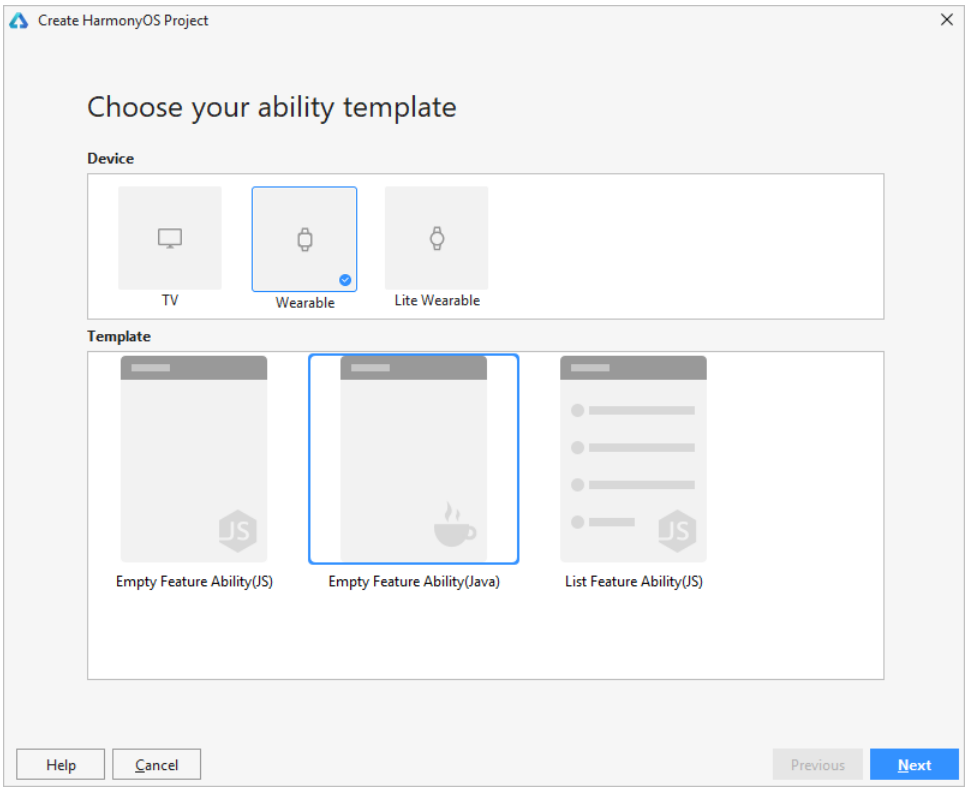
开发环境配置完成后，可以通过运行 HelloWorld 工程来验证环境设置是否正确。

运行 Hello World

DevEco Studio 开发环境配置完成后，可以通过运行 HelloWorld 工程来验证环境设置是否正确。以 Wearable 工程为例，在 Wearable 远程模拟器中运行该工程。

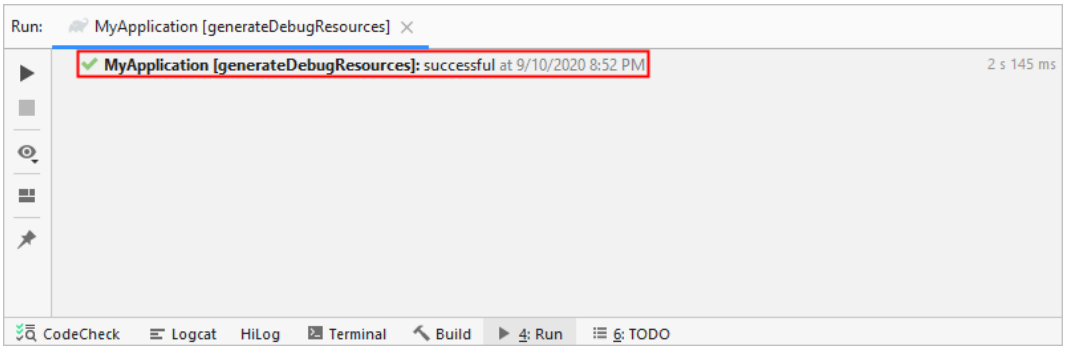
打开 DevEco Studio，在欢迎页点击 **Create HarmonyOS Project**，创建一个新工程。

选择设备类型和模板，以 Wearable 为例，选择 Empty Feature Ability(Java)，点击 **Next**。

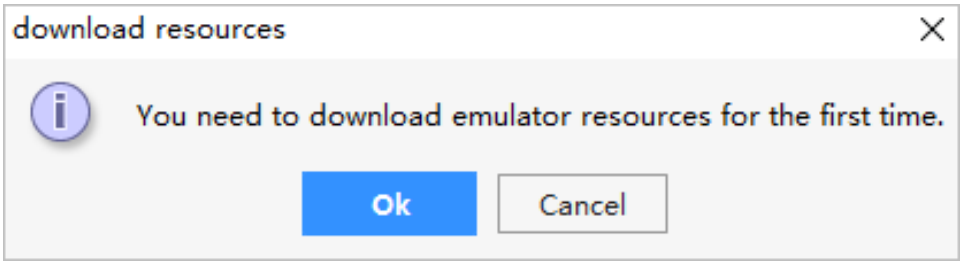


填写项目相关信息，保持默认值即可，点击 **Finish**。

工程创建完成后，DevEco Studio 会自动进行工程的同步，同步成功如下图所示。首次创建工程时，会自动下载 Gradle 工具（Gradle 下载失败如何解决?），时间较长，请耐心等待。



在 DevEco Studio 菜单栏，点击 **Tools > HVD Manager**。首次使用模拟器，需下载模拟器相关资源，请点击 **OK**，等待资源下载完成后，点击模拟器界面左下角的 **Refresh** 按钮。（查看使用远程模拟器的常见问题）



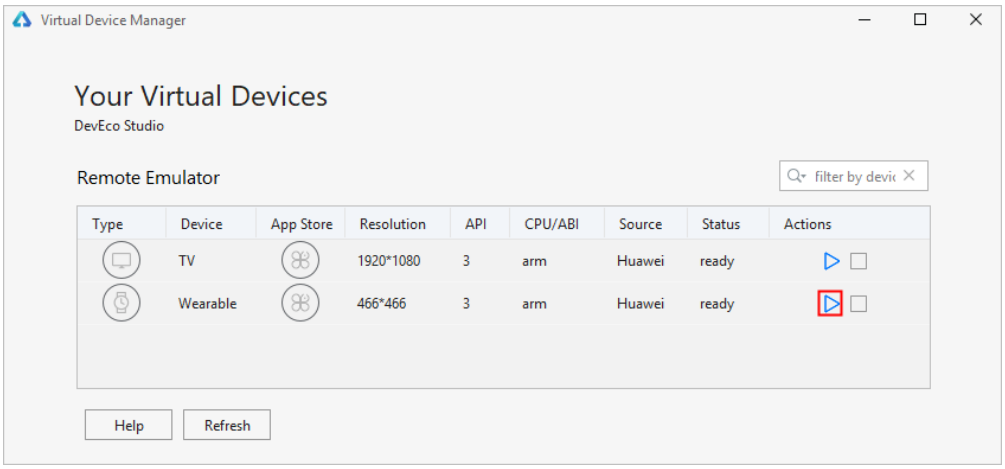
在浏览器中弹出华为帐号登录界面，请输入已实名认证的华为帐号的用户名和密码进行登录。

说明

推荐使用 Chrome 浏览器，如果使用 Safari、360 等其他浏览器，要取消**阻止跨站跟踪**和**阻止所有 Cookie** 功能。

登录后，请点击界面的**允许**按钮进行授权。

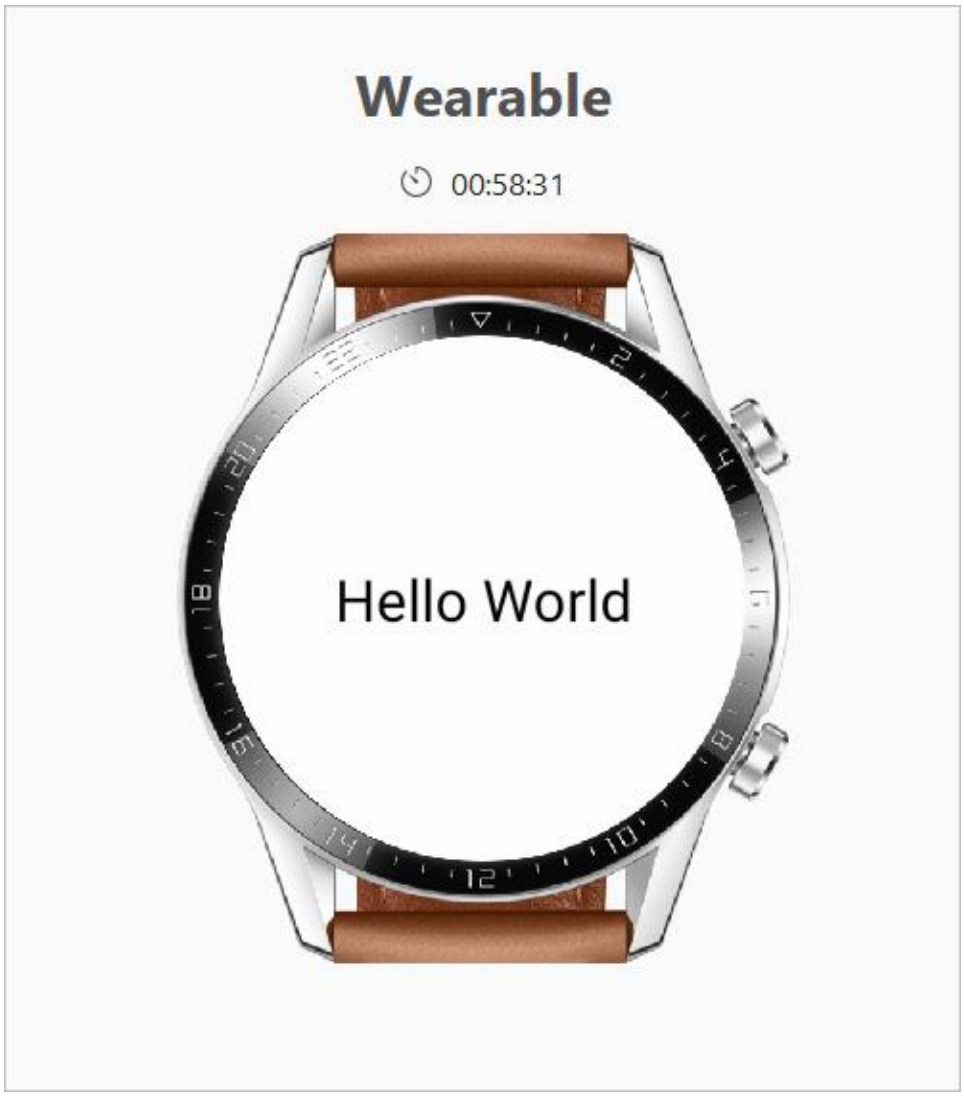
在设备列表中，选择 Wearable 设备，并点击 ▷ 按钮，运行模拟器。



点击 DevEco Studio 工具栏中的 ▶ 按钮运行工程，或使用默认快捷键 **Shift+F10** 运行工程。

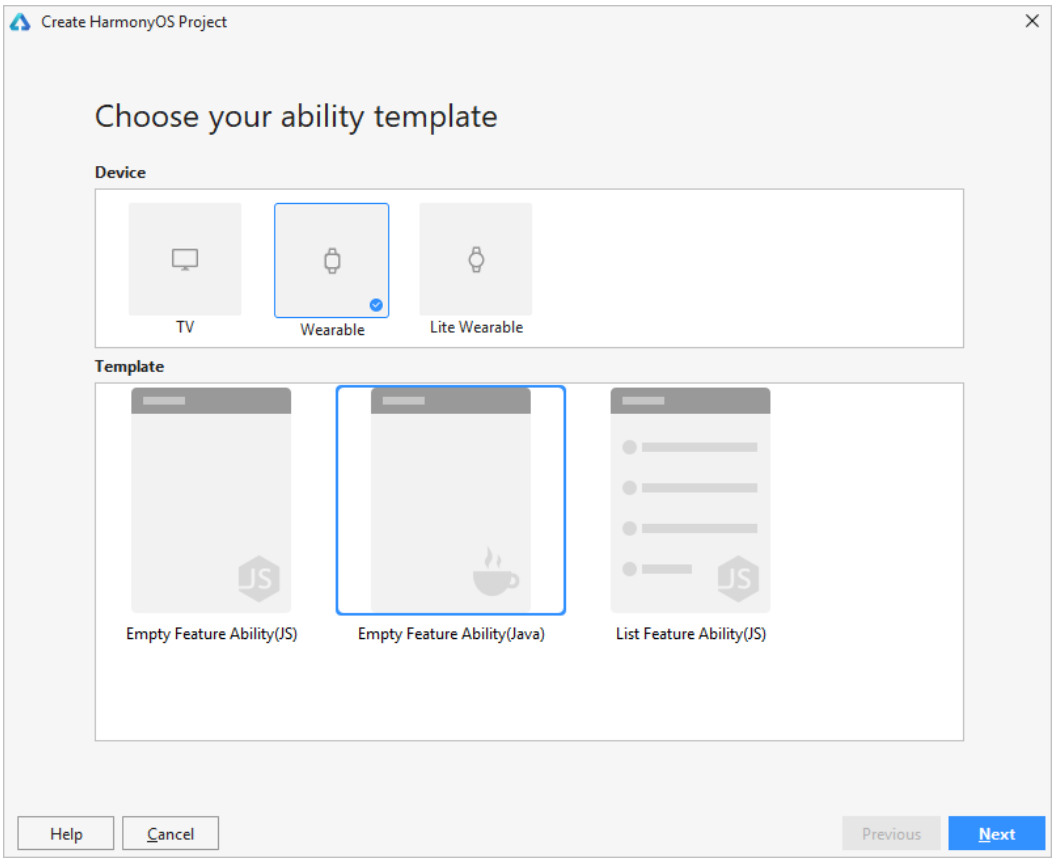
在弹出的 Select Deployment Target 界面选择 **Connected Devices**，点击 **OK** 按钮。

DevEco Studio 会启动应用的编译构建，完成后应用即可运行在 **Remote Device** 上。



支持的设备模板和编程语言

DevEco Studio 支持包括智慧屏、智能穿戴和轻量级智能穿戴的 HarmonyOS 应用开发，可以根据工程向导轻松创建适应于各类设备的工程，并自动生成对应的代码和资源模板。同时，DevEco Studio 还提供了多种编程语言供开发者进行 HarmonyOS 应用开发，包括 Java、JS 和 C/C++三种编程语言，并支持多种语言的混合开发场景。因此，在创建对应设备的工程时，工具会预置多种 Ability 的模板，并推荐您使用适合的开发语言。



支持的各设备类型工程模板和对应开发语言的对应关系，如下表所示。

表 1 各设备开发模板

Device	工程模板
TV	Empty Feature Ability（JS）

表 1 各设备开发模板

Device	工程模板
	Empty Feature Ability（Java）
	List Container Ability（Java）
	List Feature Ability（JS）
	Split Panel Ability（Java）
	Tab Feature Ability（JS）
Wearable	Empty Feature Ability（JS）
	Empty Feature Ability（Java）
	List Feature Ability（JS）
Lite Wearable	Empty Feature Ability
	List Feature Ability

创建一个新的工程

当开始开发一个 HarmonyOS 应用时，首先需要根据工程创建向导，创建一个新的工程，工具会自动生成对应的代码和资源模板。

如果创建的工程包含 JS 语言，请确保已经下载了 JS SDK 包，具体可参考下载 HarmonyOS SDK。

说明

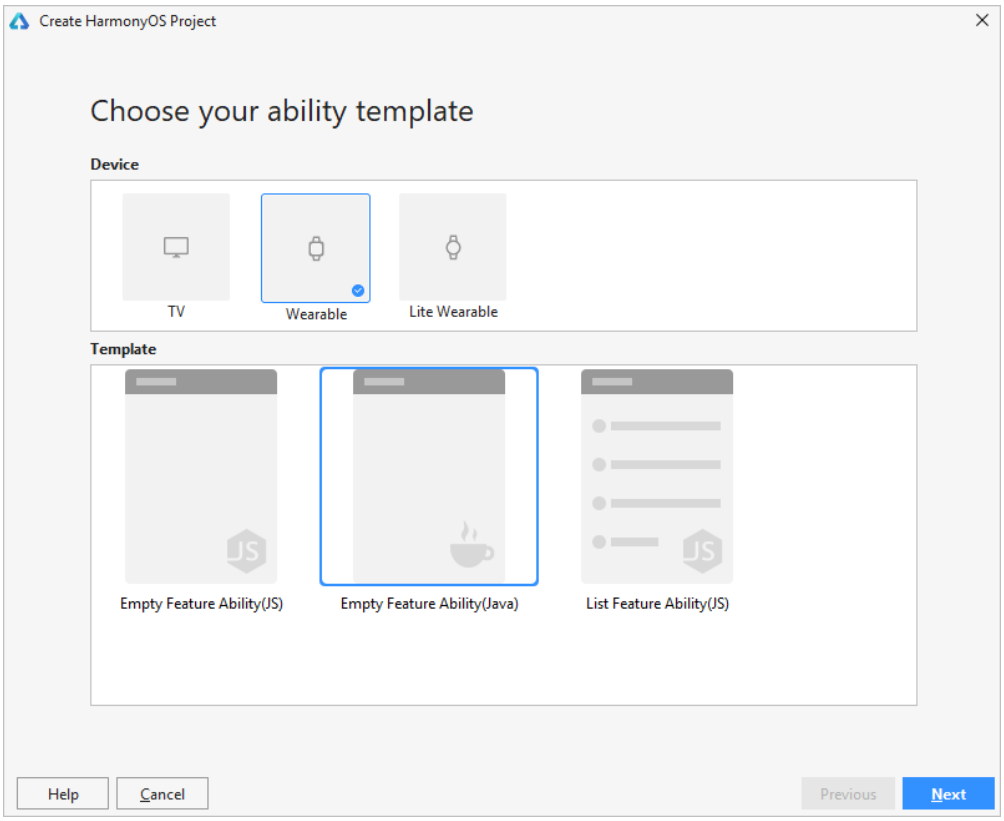
在运行 DevEco Studio 工程时，建议每一个运行窗口有 2GB 以上的可用内存空间。

创建和配置新工程

1. 通过如下两种方式，打开工程创建向导界面。

- 如果当前未打开任何工程，可以在 DevEco Studio 的欢迎页，选择 **Create HarmonyOS Project** 开始创建一个新工程。
- 如果已经打开了工程，可以在菜单栏选择 **File > New > New Project** 来创建一个新工程。

2. 根据工程创建向导，选择需要进行开发的设备类型，然后选择对应的 Ability 模板。



1. 点击 **Next**，进入到工程配置阶段，需要根据向导配置工程的基本信息。

- **Project name**：工程的名称，可以自定义。
- **Package name**：软件包名称，默认情况下，应用 ID 也会使用该名称，应用发布时，应用 ID 需要唯一。
- **Save location**：工程文件本地存储路径。
- **Compatible SDK**：兼容的 SDK 版本。

2. 点击 **Finish**，工具会自动生成示例代码和相关资源，等待工程创建完成。

导入现有工程

导入现有工程分为以下两种情况：

- 导入 DevEco Studio 创建的 HarmonyOS 应用工程：
- 如果当前未打开任何工程，可以在 DevEco Studio 的欢迎页，选择 **Open Project** 打开现有工程。

- 如果已经打开了工程，可以在菜单栏选择 **File > Open** 来打开现有工程。
- 导入其它 IDE 创建的工程，比如导入 Visual Studio Code 创建的轻量级智能穿戴 HarmonyOS 应用工程：
- 如果当前未打开任何工程，可以在 DevEco Studio 的欢迎页，选择 **Import Project** 导入现有工程。
- 如果已经打开了工程，可以在菜单栏选择 **File > Import Project** 导入现有工程。

导入现有工程时，DevEco Studio 会提醒您可以选择在新的窗口打开工程，或者选择在当前窗口打开工程。

在工程中添加 Module

Module 是 HarmonyOS 应用的基本功能单元，包含了源代码、资源文件、第三方库及应用清单文件，每一个 Module 都可以独立进行编译和运行。一个 HarmonyOS 应用通常会包含一个或多个 Module，因此，可以在工程中，创建多个 Module，每个 Module 分为 Ability 和 Library（HarmonyOS Library 和 Java Library）两种类型。

如 HarmonyOS 工程介绍，在一个 APP 中，对于同一类型设备有且只有一个 Entry Module，其余 Module 的类型均为 Feature。因此，在创建一个类型为 Ability 的 Module 时，遵循如下原则：

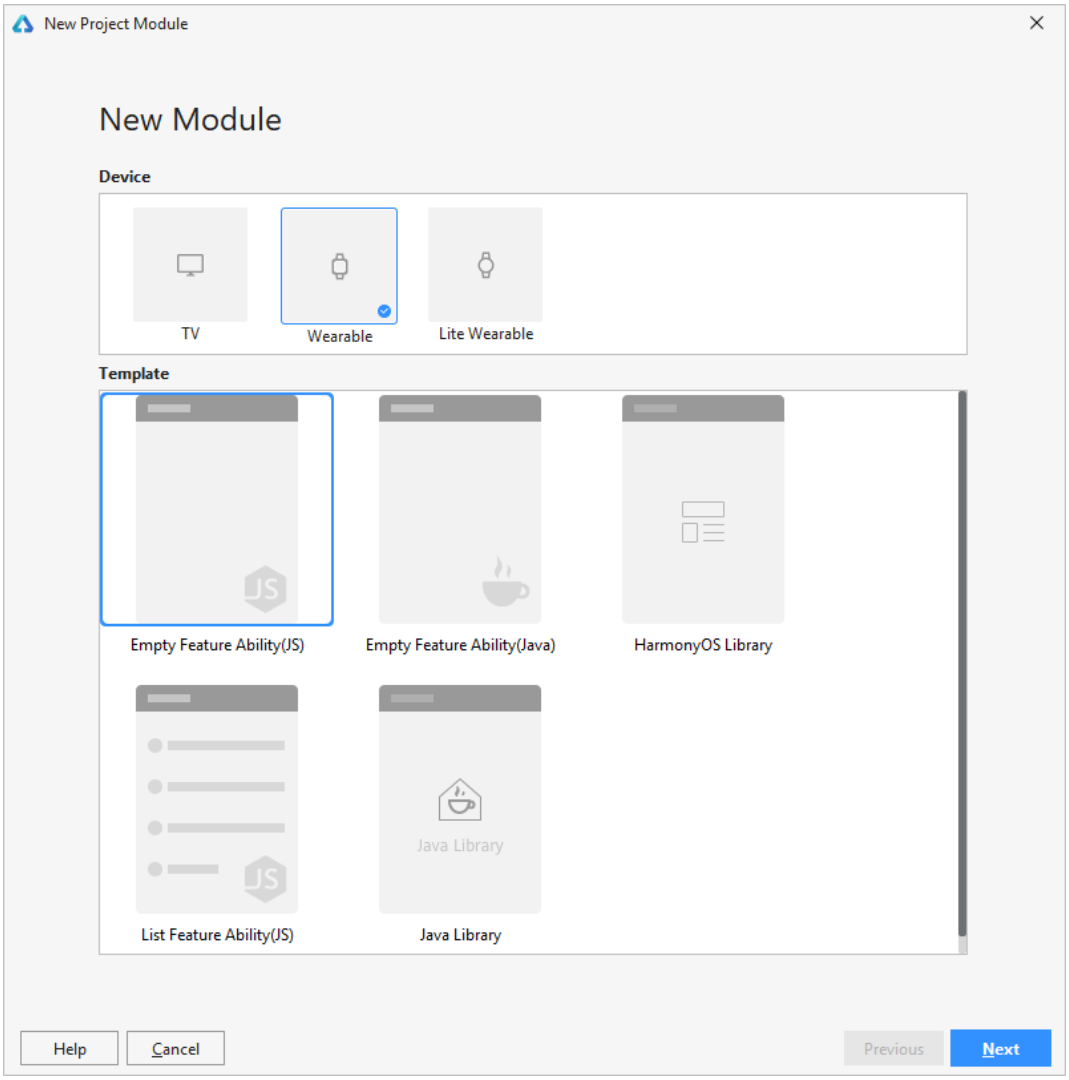
- 若新增 Module 的设备类型为已有设备时，则 Module 的类型将自动设置为“Feature”。
- 若新增 Module 的设备类型为当前还没有创建 Module，则 Module 的类型将自动设置为“Entry”。

新增 Module

1. 通过如下两种方法，在工程中添加新的 Module。

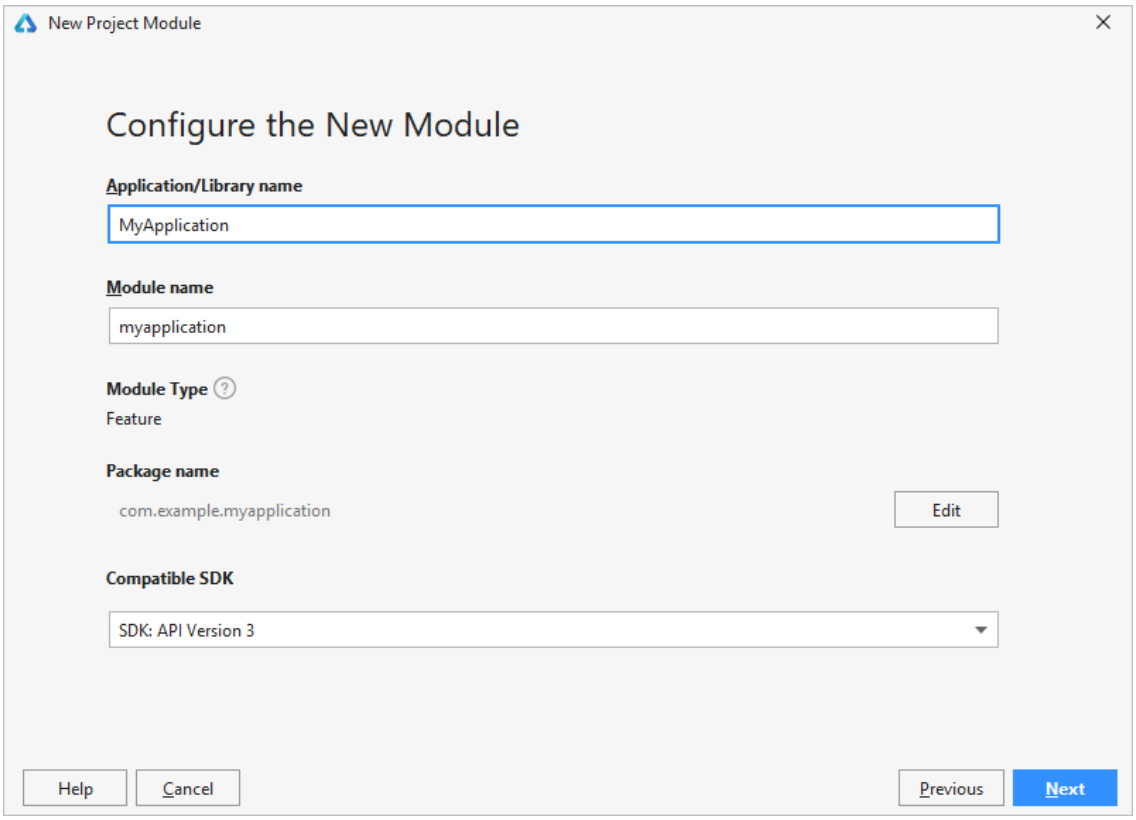
- 方法 1：鼠标移到工程目录顶部，点击鼠标右键，选择 **New>Module**，开始创建新的 Module。
- 方法 2：在菜单栏选择 **File > New > Module**，开始创建新的 Module。

2. 在 New Project Module 界面中，选择 Module 对应的设备类型和模板。

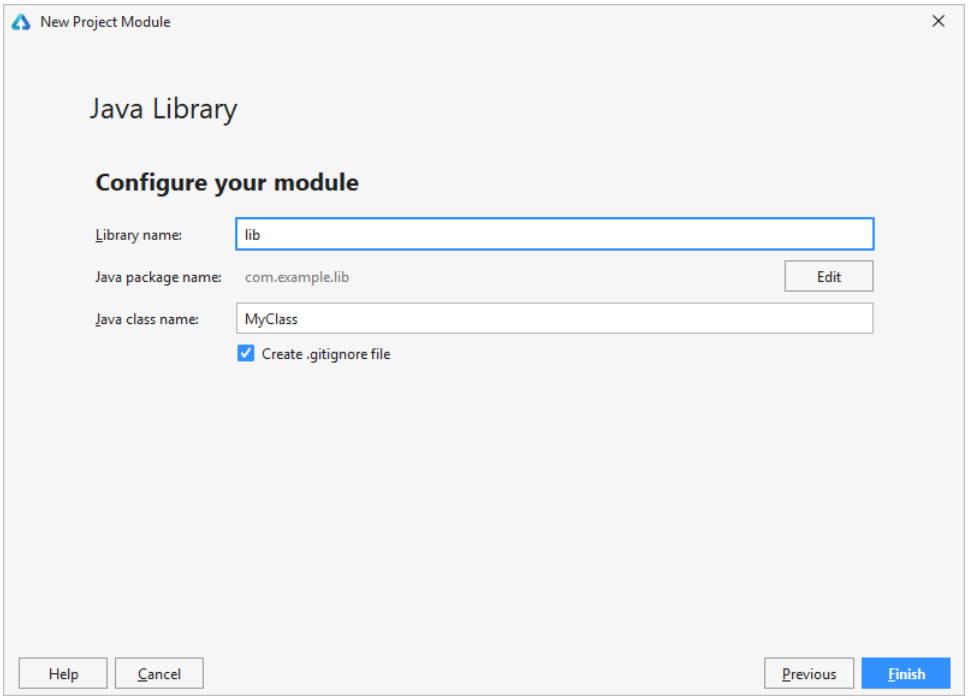


1. 点击 **Next**，在 Module 配置页面，设置新增 Module 的基本信息。

- Module 类型为 Ability 或者 HarmonyOS Library 时，请根据如下内容进行设置，然后点击 **Next**。
- **Application/Library name**：新增 Module 所属的类名称。
- **Module name**：新增模块的名称。
- **Module Type**：仅 Module 类型为 Ability 时存在，工具自动根据设备类型下的模块进行设置。
- **Package name**：软件包名称，可以点击 **Edit** 修改默认包名称，需全局唯一。
- **Compatible SDK**：兼容的 SDK 版本。



- Module 类型为 Java Library 时，请根据如下内容进行设置，然后点击 **Finish** 完成创建。
- **Library Name**：Java Library 类名称。
- **Java package name**：软件包名称，可以点击 **Edit** 修改默认包名称，需全局唯一。
- **Java class name**：class 文件名称。
- **Create.gitignore file**：是否自动创建.gitignore 文件，勾选表示创建。



设置新增 Ability 或 HarmonyOS Library 的 Page Name。
若该 Module 类型为 Ability，需要设置 Visible 参数，表示该 Ability 是否可以被其它应用所调用。

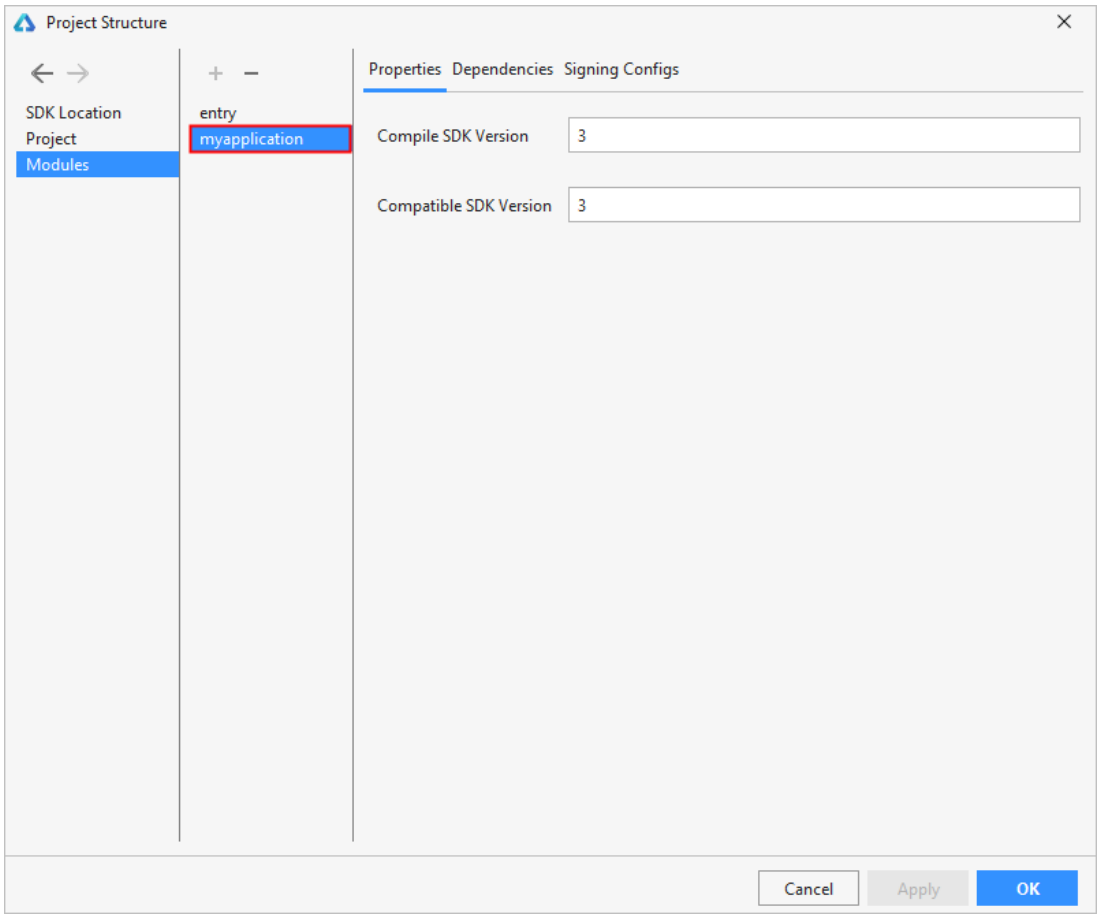
- 勾选（true）：可以被其它应用调用。
- 不勾选（false）：不能被其它应用调用。

点击 **Finish**，等待创建完成后，可以在工程目录中查看和编辑新增的 Module。

删除 Module

为防止开发者在删除 Module 的过程中，误将其它的模块删除，DevEco Studio 提供统一的模块管理功能，需要先在模块管理中，移除对应的模块后，才允许删除。

1. 在菜单栏中选择 **File > Project Structure > Modules**，选择需要删除的 Module，点击 **—** 按钮，并在弹出的对话框中点击 **Yes**。



在工程目录中选中该模块，点击鼠标右键，选中 **Delete**，并在弹出的对话框中点击 **Delete**。

代码编辑

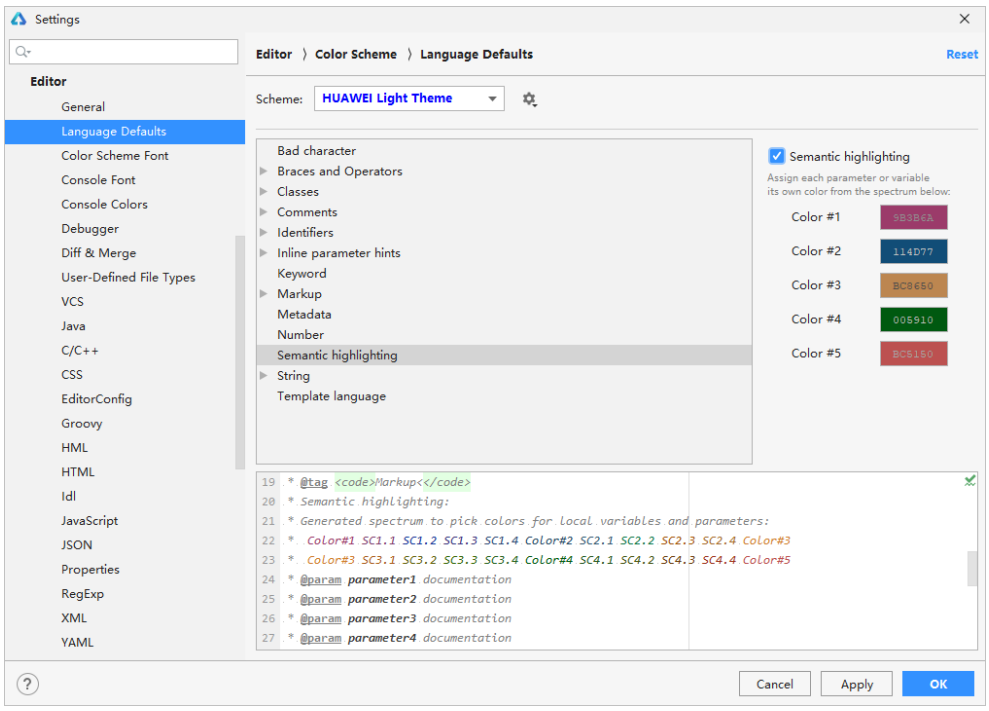
编辑器使用技巧

DevEco Studio 支持多种语言进行 HarmonyOS 应用的开发，包括 Java、JS 和 C/C++。在编写应用阶段，您可以通过掌握各种代码编写的各种常用技巧，来提升编码效率。

代码高亮

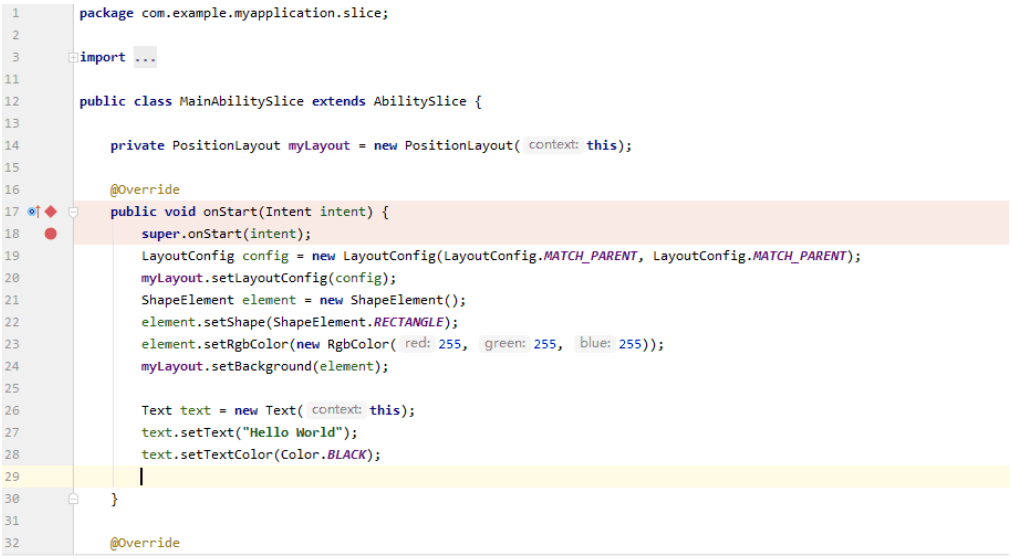
支持对代码关键字、运算符、字符串、类名称、接口名、枚举值等进行高亮颜色显示，可以在菜单栏打开 **File > Settings**（或快捷键 **Ctrl+Alt+S**）面板，在 **Editor > Color Scheme** 自定义各语言高亮显示颜色。

同时还可以动态的对**变量名**和**参数名**进行语义高亮，默认情况下为关闭状态，可以在菜单栏打开 **File > Settings**（或快捷键 **Ctrl+Alt+S**）面板，在 **Editor > Color Scheme > Language Defaults > Semantic highlighting** 中，打开语义高亮开关。



代码智能补齐

编辑器工具会分析上下文并理解项目内容，并根据输入的内容，提示可补齐的类，方法，字段和关键字的名称等。



```
package com.example.myapplication.slice;

import ...

public class MainAbilitySlice extends AbilitySlice {

    private PositionLayout myLayout = new PositionLayout( context: this);

    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
        LayoutConfig config = new LayoutConfig(LayoutConfig.MATCH_PARENT, LayoutConfig.MATCH_PARENT);
        myLayout.setLayoutConfig(config);
        ShapeElement element = new ShapeElement();
        element.setShape(ShapeElement.RECTANGLE);
        element.setRgbColor(new RgbColor( red: 255,  green: 255,  blue: 255));
        myLayout.setBackground(element);

        Text text = new Text( context: this);
        text.setText("Hello World");
        text.setTextColor(Color.BLACK);

    }

    @Override
```

代码错误检查

如果输入的语法不符合编码规范，或者出现拼写错误，编辑器会实时的进行代码分析，并在代码中突出显示错误或警告，并给出对应的修改建议。

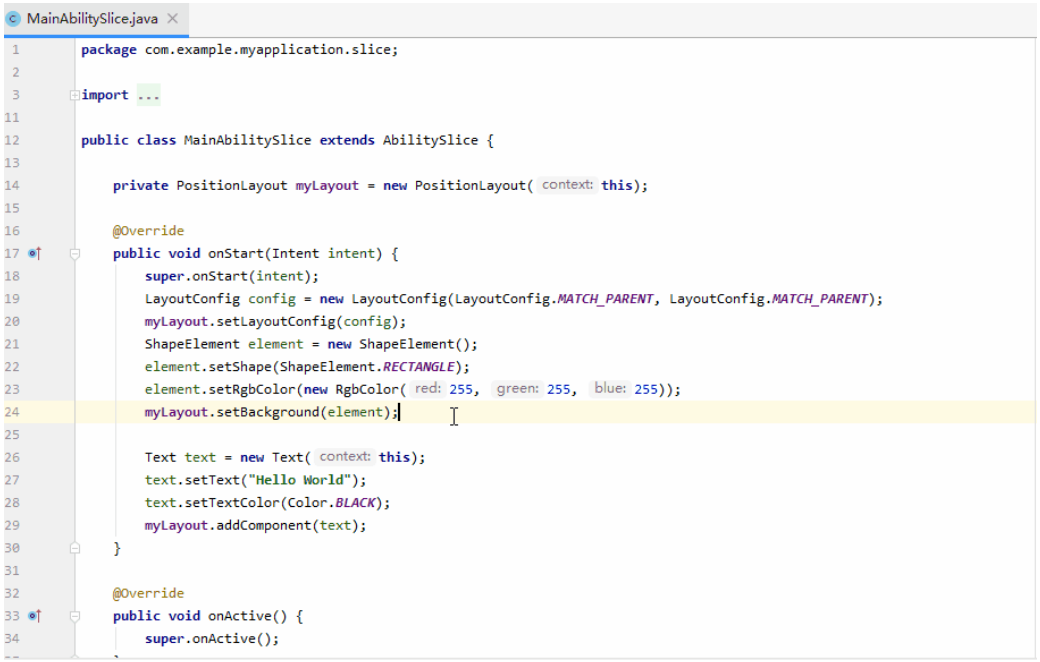
```
package com.example.myfirstapp.slice;

import ...

public class MainAbilitySlice extends AbilitySlice {

    private DirectionalLayout myLayout = new DirectionalLayout( context: this);

    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
        tyyxx
        Config(LayoutConfig.MATCH_PARENT, LayoutConfig.MATCH_PARENT);
        Element();
        element.setRgbColor(new RgbColor( red: 255,  green: 255,  blue: 255));
        myLayout.setBackground(element);
```

Cannot resolve symbol 'tyyxx'

Create local variable 'tyyxx' Alt+Shift+Enter More actions... Alt+Enter

代码自动跳转

在编辑器中，可以按住 **Ctrl** 键，鼠标点击代码中的类、方法、参数、变量等名称，可以自动跳转到定义处。



代码格式化

支持对选定范围的代码或者当前整个文件进行代码格式化操作，可以很好的提升代码的美观度和可读性。

- 使用快捷键 **Ctrl + Alt + L** 可以快速对选定范围的代码进行格式化。
- 使用快捷件 **Ctrl + Alt + Shift + L** 可以快速对当前整个文件进行格式化。

如果在进行格式化时，对于部分代码片段不需要进行自动的格式化处理，可以通过如下方式进行设置：

1. **首先**，在 **File>Settings>Editor>Code Style**，点击“Formatter Control”，勾选“Enable formatter markers in comments”。

2. **其次**，在 Java 或 C/C++代码中，对不需要进行格式化操作的代码块前增加“//@formatter:off”，对不格式化代码块的最后增加“//@formatter:on”，即表示对该范围的代码块不需要进行格式化操作。

```
17    public void onStart(Intent intent) {
18        //@formatter:off
19          super.onStart(intent);
20        LayoutConfig config = new LayoutConfig(LayoutConfig.MATCH_PARENT, LayoutConfig.MATCH_PARENT);
21          myLayout.setLayoutConfig(config);
22        ShapeElement element = new ShapeElement();
23          element.setShape(ShapeElement.RECTANGLE);
24        element.setRgbColor(new RgbColor( red: 255,  green: 255,  blue: 255));
25          myLayout.setBackground(element);
26        //@formatter:on
27        Text text = new Text( context: this);
28        text.setText("Hello World");
29        text.setTextColor(Color.BLACK);
30        myLayout.addComponent(text);
31        super.setUIContent(myLayout);
32    }
```

代码折叠

支持对代码块的快速折叠和展开，可以使用快捷键 **Ctrl + NumPad+** 快速展开已折叠的代码；使用快捷键 **Ctrl + NumPad-**折叠已展开的代码块。

```
14    private PositionLayout myLayout = new PositionLayout( context: this);
15
16    @Override
17    public void onStart(Intent intent) {
18        super.onStart(intent);
19        LayoutConfig config = new LayoutConfig(LayoutConfig.MATCH_PARENT, LayoutConfig.MATCH_PARENT);
20        myLayout.setLayoutConfig(config);
21        ShapeElement element = new ShapeElement();
22        element.setShape(ShapeElement.RECTANGLE);
23        element.setRgbColor(new RgbColor( red: 255,  green: 255,  blue: 255));
24        myLayout.setBackground(element);
25
26        Text text = new Text( context: this);
27        text.setText("Hello World");
28        text.setTextColor(Color.BLACK);
29        myLayout.addComponent(text);
30        super.setUIContent(myLayout);
31    }
32
33    @Override
34    public void onActive() {
35        super.onActive();
36    }
37
38    @Override
39    public void onForeground(Intent intent) {
40        super.onForeground(intent);
```

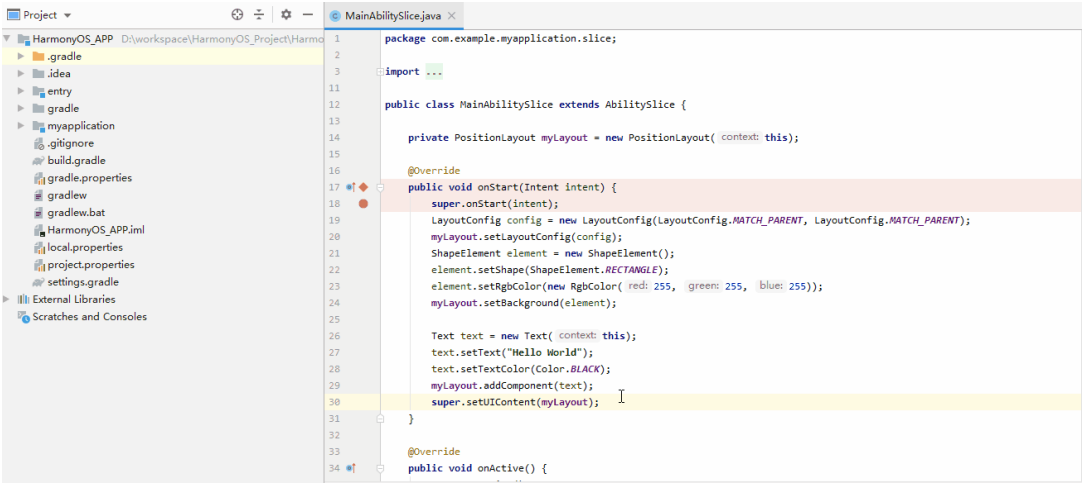
代码快速注释

支持对选择的代码块进行快速注释，使用快捷键 **Ctrl+/**快速进行注释。对于已注释的代码快，再次使用快捷键 **Ctrl+/**取消注释。

```
11
12  public class MainAbilitySlice extends AbilitySlice {
13
14      private PositionLayout myLayout = new PositionLayout( context: this);
15
16      @Override
17      public void onStart(Intent intent) {
18          super.onStart(intent);
19          LayoutConfig config = new LayoutConfig(LayoutConfig.MATCH_PARENT, LayoutConfig.MATCH_PARENT);
20          myLayout.setLayoutConfig(config);
21          ShapeElement element = new ShapeElement();
22          element.setShape(ShapeElement.RECTANGLE);
23          element.setRgbColor(new RgbColor( red: 255,  green: 255,  blue: 255));
24          myLayout.setBackground(element);
25
26          Text text = new Text( context: this);
27          text.setText("Hello World");
28          text.setTextColor(Color.BLACK);
29          myLayout.addComponent(text);
30          super.setUIContent(myLayout);
31      }
32
33      @Override
34      public void onActive() { super.onActive(); }
37
38      @Override
39      public void onForeground(Intent intent) { super.onForeground(intent); }
```

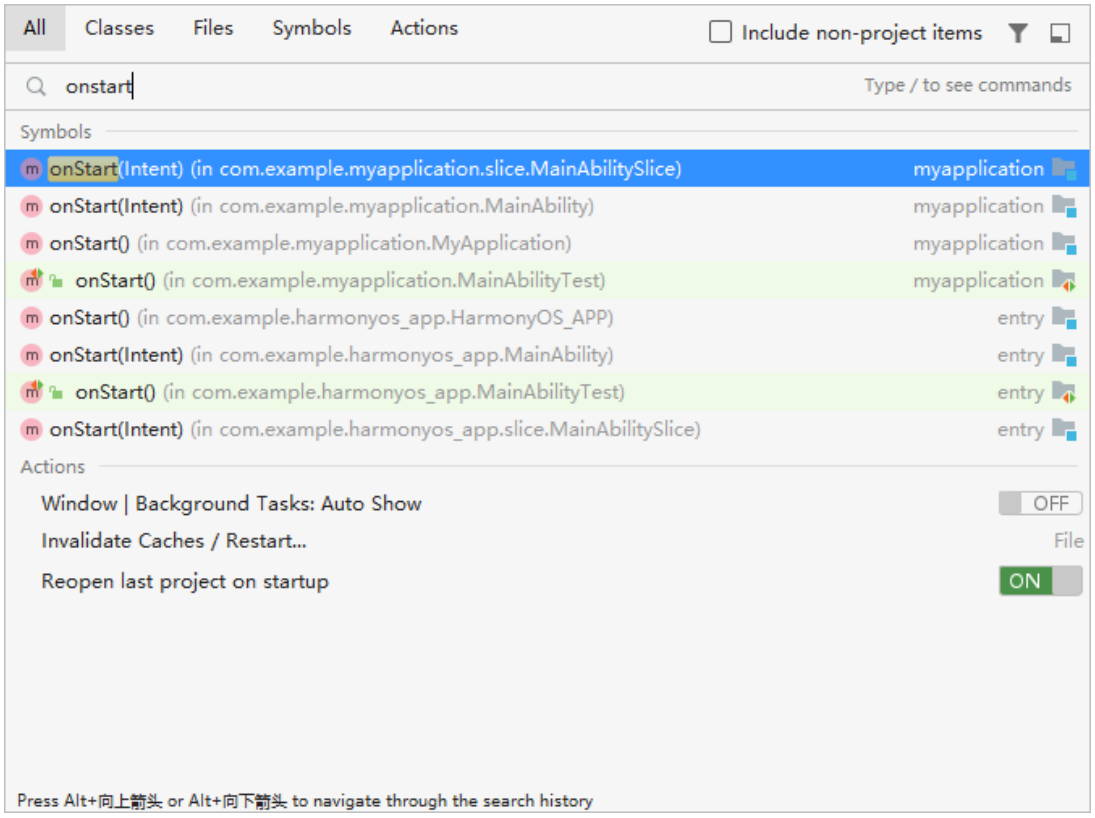
代码结构树

支持快速查看代码文档的结构树，包括全局变量和函数，类成员变量和方法等，并可以跳转到对应代码行。可使用快捷键 **Alt + 7 / Ctrl + F12** 快速打开代码结构树。



代码查找

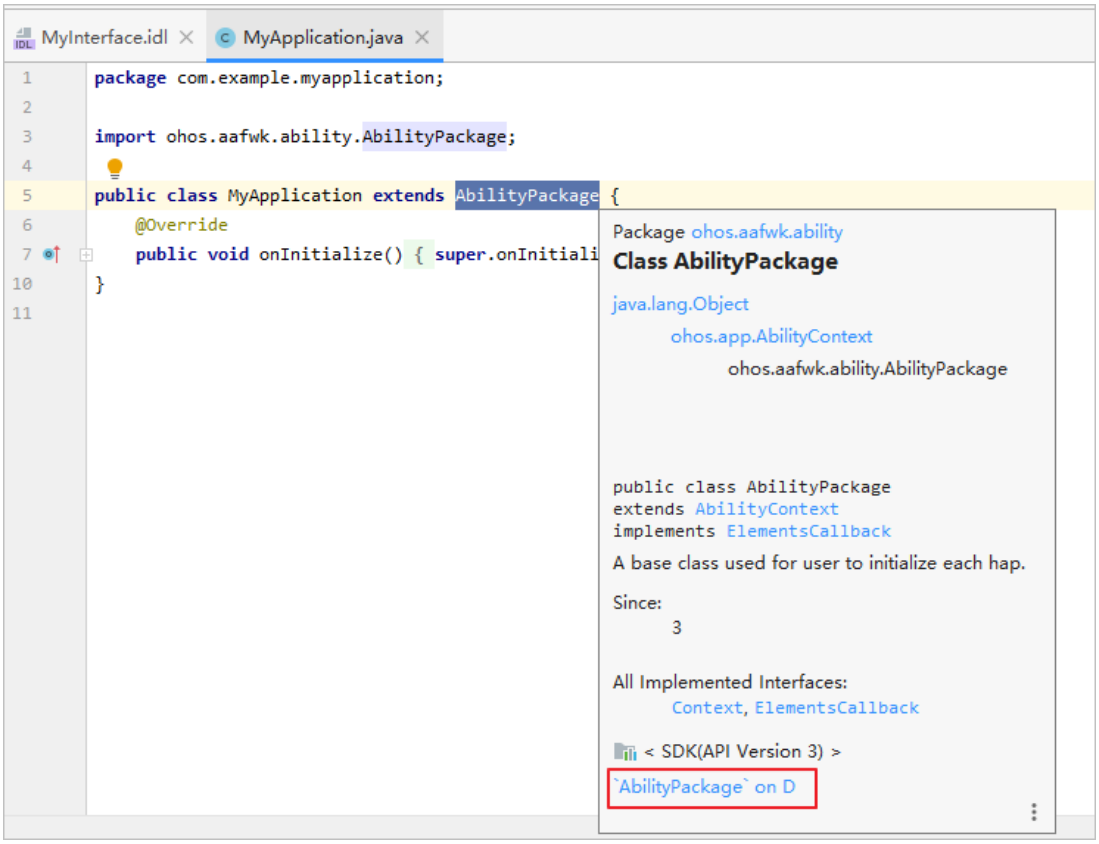
通过对符号、类或文件的即时导航来查找代码。检查调用或类型层次结构，轻松地搜索工程里的所有内容。通过使用连续按压**两次 Shift** 快捷键，打开代码查找界面。



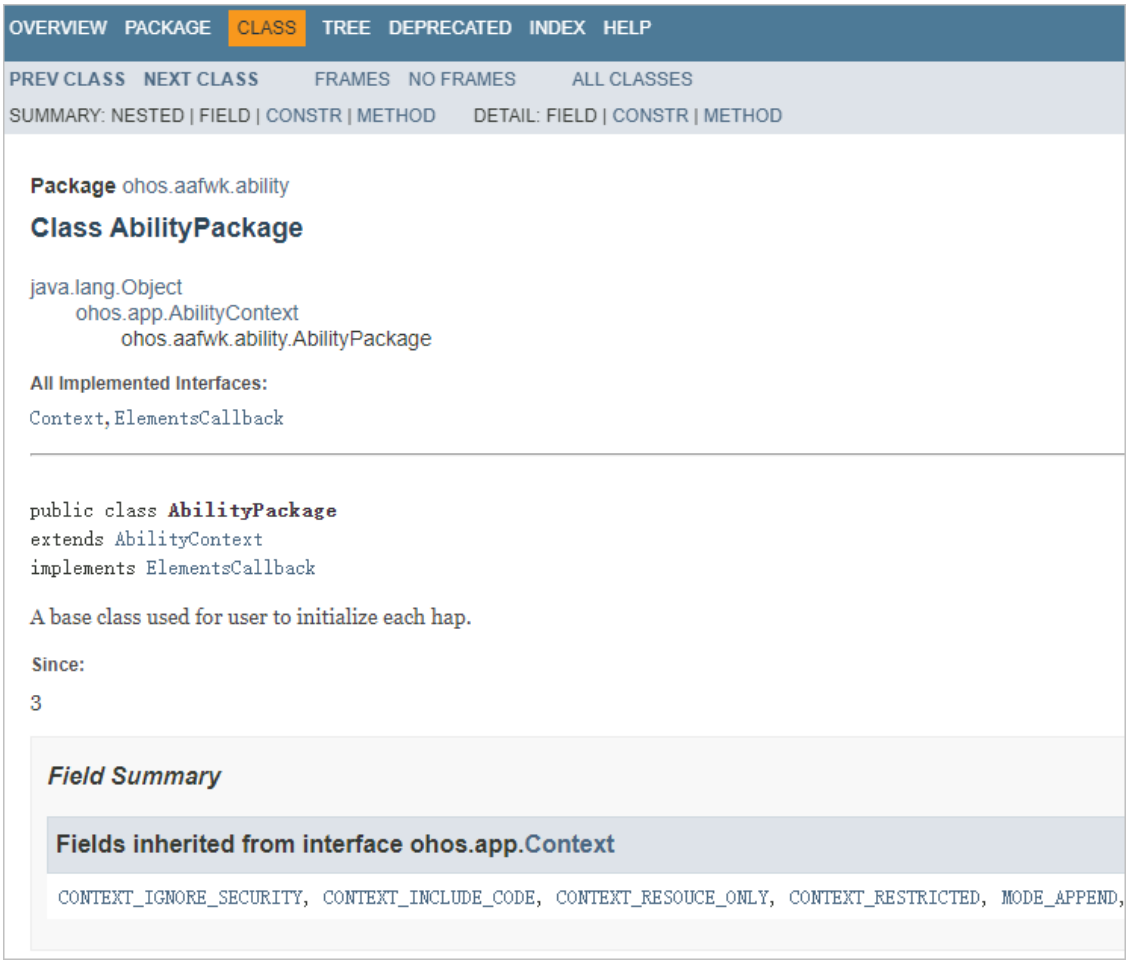
查看 Java 接口文档

在 Java 代码选中 HarmonyOS API 或选中 Java 类时，使用快捷键 **Ctrl+Q**，在弹出的“Documentation”最下方，会显示相应文档的链接。

例如：图示红框中的“‘AbilityPackage’on D”



点击文档的链接，比如：“‘AbilityPackage’on D”，将打开详细说明文档。



OVERVIEW PACKAGE CLASS TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Package ohos.aafwk.ability

Class AbilityPackage

java.lang.Object
ohos.app.AbilityContext
ohos.aafwk.ability.AbilityPackage

All Implemented Interfaces:

Context, ElementsCallback

public class **AbilityPackage**
extends AbilityContext
implements ElementsCallback

A base class used for user to initialize each hap.

Since:

3

Field Summary

Fields inherited from interface ohos.app.Context

CONTEXT_IGNORE_SECURITY, CONTEXT_INCLUDE_CODE, CONTEXT_RESOUCE_ONLY, CONTEXT_RESTRICTED, MODE_APPEND,

在模块中添加 Ability

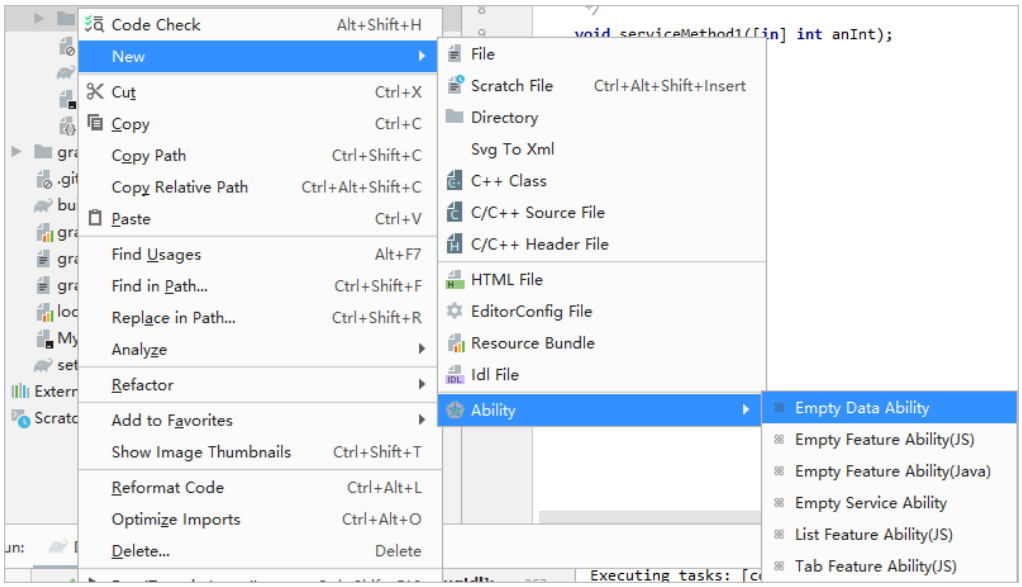
Ability 是应用所具备的能力的抽象，一个 Module 可以包含一个或多个 Ability。Ability 分为两种类型：FA（Feature Ability）和 PA（Particle Ability），DevEco Studio 支持创建的 Ability 模板和应用场景如下表所示。

Ability 类型	Ability 模板	使用场景

Ability 类型	Ability 模板	使用场景
Particle Ability	Empty Data Ability	Data Ability 有助于应用管理其自身和其他应用所存储数据的访问，并提供与其他应用共享数据的方法。Data 既可用于同设备不同应用的数据共享，也支持跨设备之间不同应用的数据共享。
	Empty Service Ability	Service Ability 可在后台长时间运行而不提供用户交互界面。Service 可由其他应用或 Ability 启动，即使用户切换到其他应用，Service 仍将在后台继续运行。
Feature Ability	Empty Feature Ability(JS)	用 JS 和 Java 编写带 UI 界面的空模板。
	Empty Feature Ability(Java)	用 Java 和 xml 编写带 UI 界面的空模板。
	List Feature Ability(JS)	用 JS 和 Java 编写带 UI 界面的目录列表模板。
	Tab Feature Ability(JS)	用 JS 和 Java 编写带 UI 界面的表单模板。

创建 Particle Ability

1. 选中对应的模块，点击鼠标右键，选择 **New > Ability**，然后选择 Empty Data Ability 或者 Empty Service Ability。



1. 根据选择的 Ability 模板，设置 Ability 的基本信息。

- **Empty Data Ability** 基本信息设置：
- **Data Name**：Data Ability 类名称。
- **Visible**：表示该 Ability 是否可以被其它应用所调用，勾选上则表示允许被调用。
- **Package name**：新增 Ability 对应的包名称。
- **Empty Service Ability** 基本信息设置：
- **Service Name**：Service Ability 类名称。
- **Visible**：表示该 Ability 是否可以被其它应用所调用，勾选上则表示允许被调用。
- **Package name**：新增 Ability 对应的包名称。
- **Enable background mode**：指定用于满足特定类型的后台服务，可以将多个后台服务类型分配给特定服务。各服务与 config.json 文件的映射关系如下表所示。

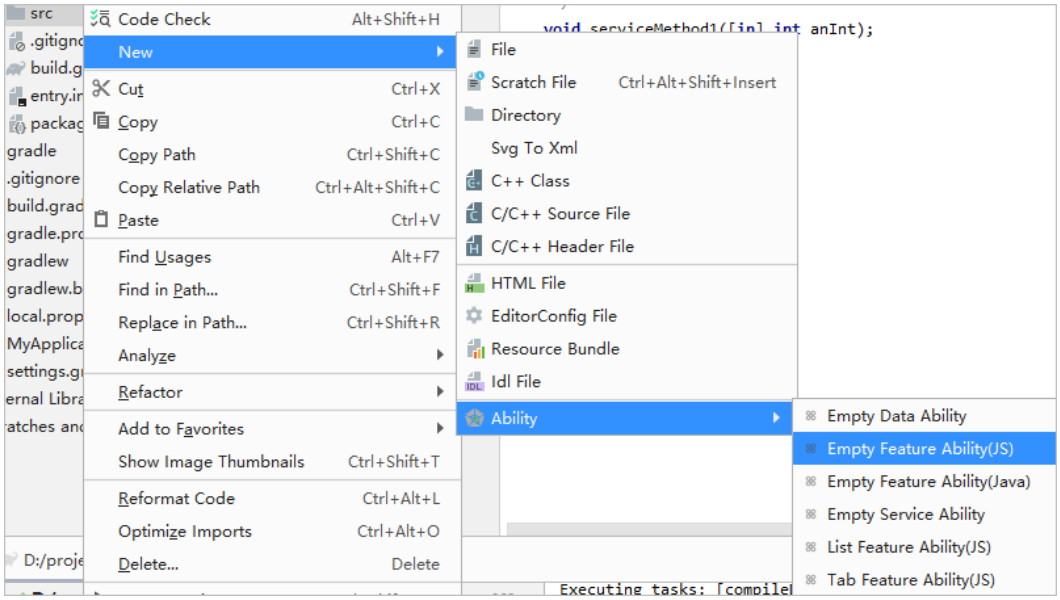
Background modes	对应 config.json 字段名称	描述
Data upload/download, backup/restore	data-transfer	通过网络/对端设备进行数据下载，备份分享，

Background modes	对应 config.json 字段名称	描述
		传输等业务
Audio playback	audio-playback	音频输出业务
Audio recording	audio-recording	音频输入业务
Picture-in-picture	picture-in-picture	画中画，小窗口播放视频业务
Voice/video call over IP	voip	音视频电话、VOIP 业务
Location update	location	定位，导航业务
Bluetooth communication	bluetooth-interaction	蓝牙扫描、连接、传输业务（穿戴）
Wifi communication	wifi-interaction	WLAN 扫描、连接、传输业务（多屏，克隆）
Screen recording, screenshot	screen-fetch	录屏，截屏业务

点击 Finish 完成 Ability 的创建，可以在工程目录对应的模块中查看和编辑 Ability。

创建 Feature Ability

1. 选中对应的模块，点击鼠标右键，选择 **New > Ability** ，然后选择对应的 Feature Ability 模板。



根据选择的 Ability 模板，设置 Feature Ability 的基本信息。

- **Page Name**：Feature Ability 类名称。
- **Launcher Ability**：表示该 Ability 在终端桌面上是否有启动图标，一个 HAP 可以有多个启动图标，来启动不同的 FA。
- **Visible**：表示该 Ability 是否可以被其它应用所调用，勾选上则表示允许被调用。
- **JS Component Name**：JS 组件名称，只有涉及 JS 开发语言时才需要设置。
- **Package name**：新增 Ability 对应的包名称。

1. 点击 Finish 完成 Ability 的创建，可以在工程目录对应的模块中查看和编辑 Ability。

添加 JS Component 和 JS Page

在支持 JS 语言的工程中，支持添加新的 JS Component 和 JS Page，在此之前，需要了解它们的基本概念。

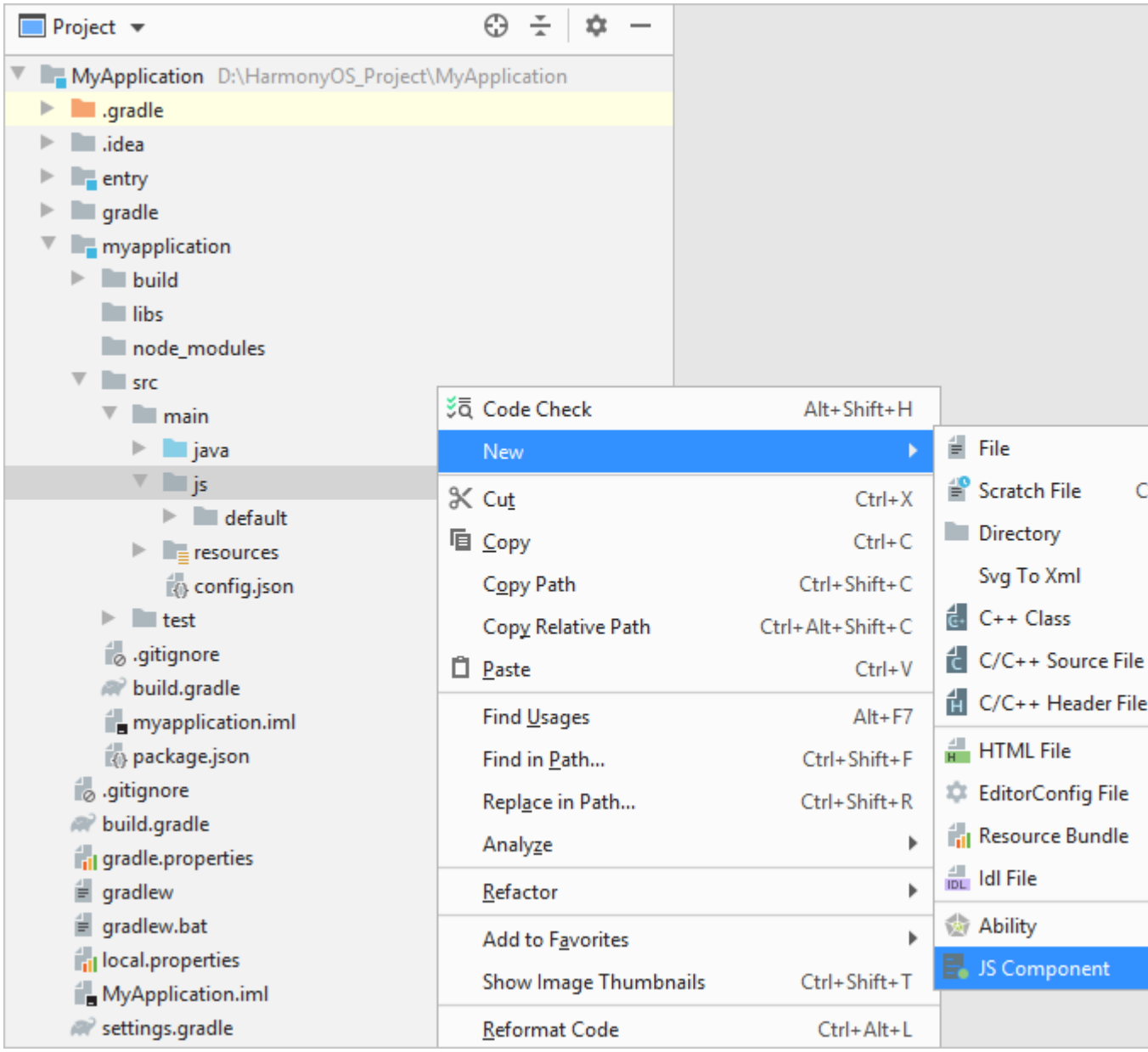
- JS Component：在 JS 工程中，可以存在多个 JS Component（例如 js 目录下的 **default** 文件夹就是一个 JS Component），一个 JS FA 对应一个 JS Component，可以独立编译、运行和调试。

> 说明
>
> 轻量级智能穿戴对应的 JS 工程，只存在一个 JS FA，因此，轻量级智能穿戴的 JS 工程不允许创建新的 JS Component。

- JS Page：Page 是表示 JS FA 的一个前台页面，由 JS、HML 和 CSS 文件组成，是 Component 的最基本单元，构成了 JS FA 的每一个界面。

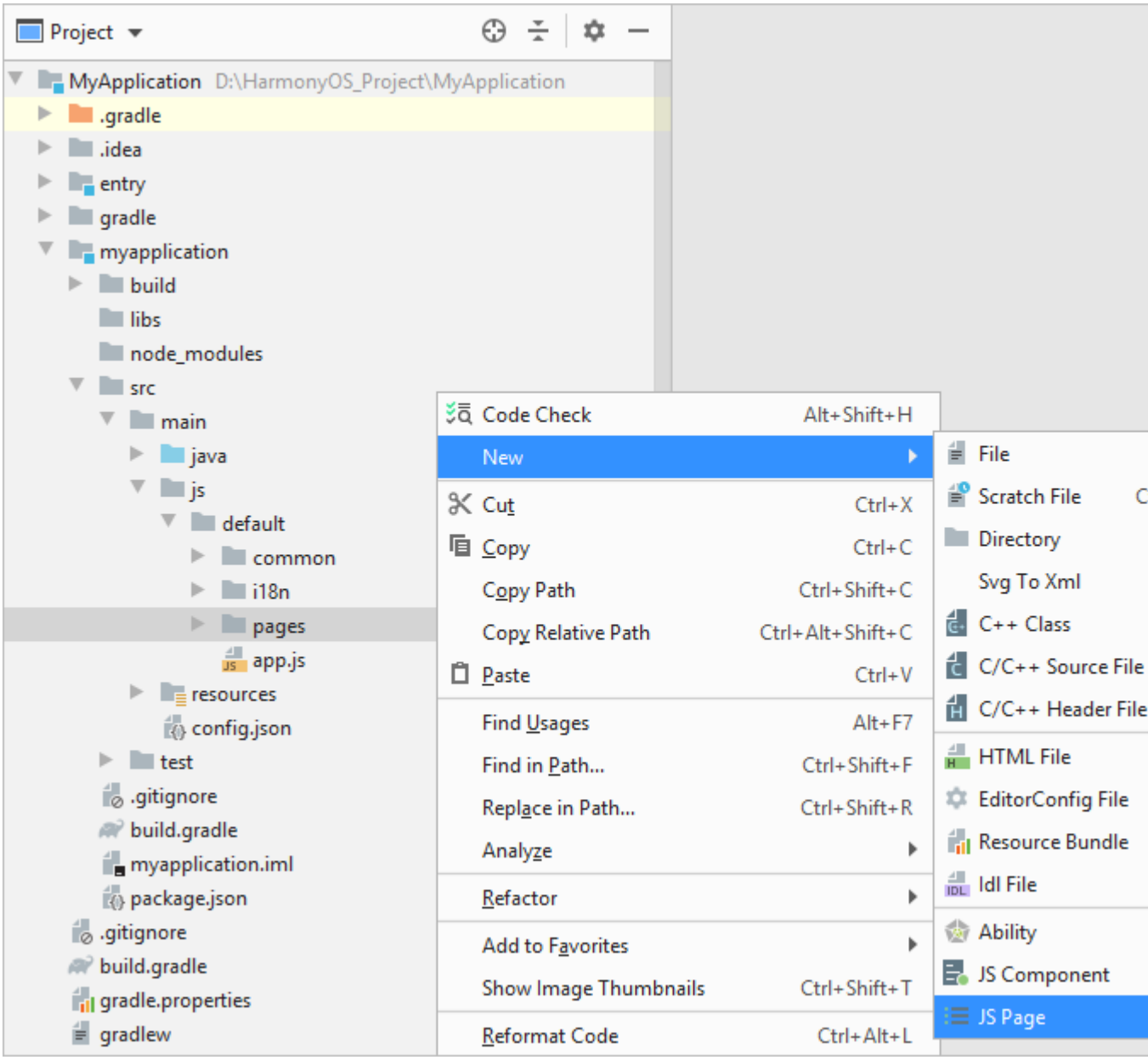
添加 JS Component

在 JS 工程目录中，选中 js 文件夹，然后点击鼠标右键，选择 **New > JS Component**，输入 JS Component Name， 点击 **Finish** 完成添加。



添加 JS Page

在 JS 工程目录中，选择需要添加 Page 的 Component 下的 pages 文件夹，然后点击鼠标右键，选择 **New > JS Page**，输入 JS Page Name，点击 **Finish** 完成添加。



定义 HarmonyOS IDL 接口

HarmonyOS IDL 简介

HarmonyOS Interface Definition Language（简称 HarmonyOS IDL）是 HarmonyOS 的接口描述语言。HarmonyOS IDL 与其他接口语言类似，通过 HarmonyOS IDL 定义客户端与服务端均认可的编程接口，可以实现在二者间的跨进程通信（IPC，Inter-Process Communication）。跨进程通信意味着我们可以在一个进程访问另一个进程的数据，或调用另一个进程的方法。

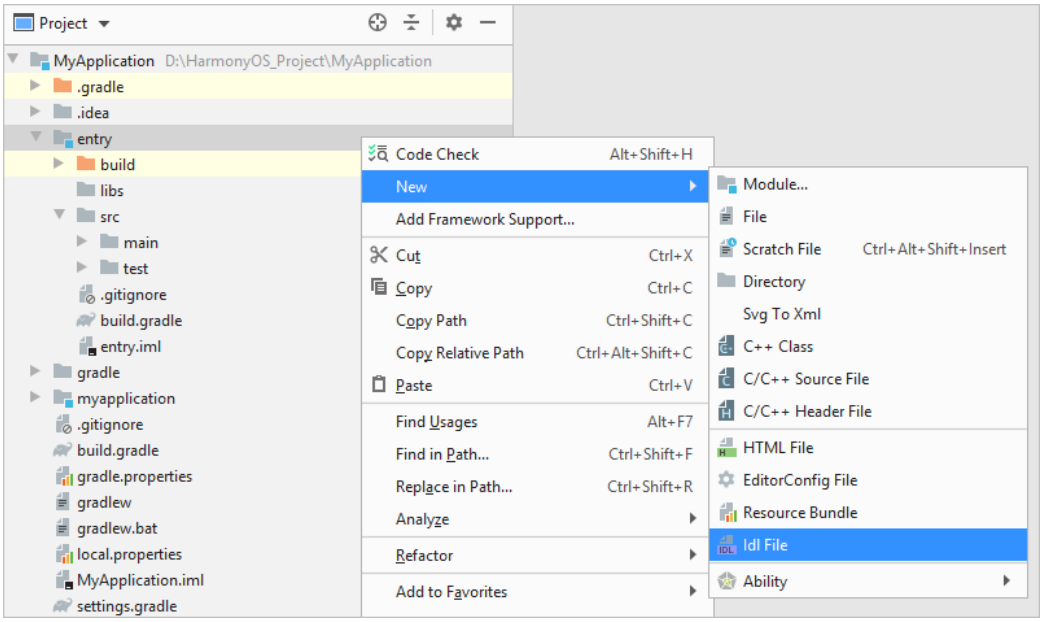
通常我们把应用接口提供方（供调用）称为服务端，调用方称为客户端。客户端通过绑定服务端的 Ability 来与之进行交互，类似于绑定服务。关于 DevEco Studio 接口语言的详细描述请参考 HarmonyOS IDL 接口使用规范。

说明

只能使用 Java 或 C++语言构建.idl 文件，因此仅 Java、Java+JS、C/C++工程支持 IDL。

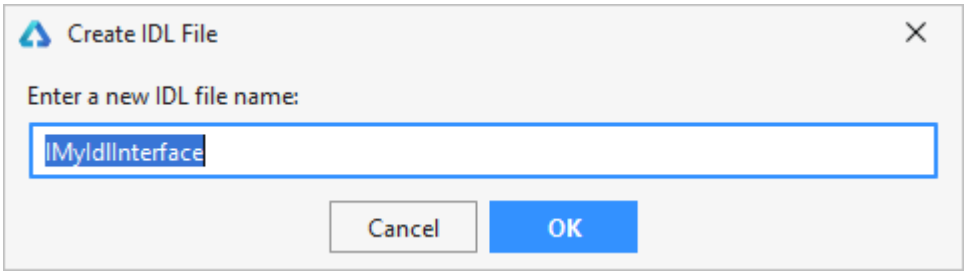
创建.idl 文件

1. 在已经创建/打开的 HarmonyOS 工程中，选择 module 目录或其子目录，点击鼠标右键，选择 **New>Idl File**。

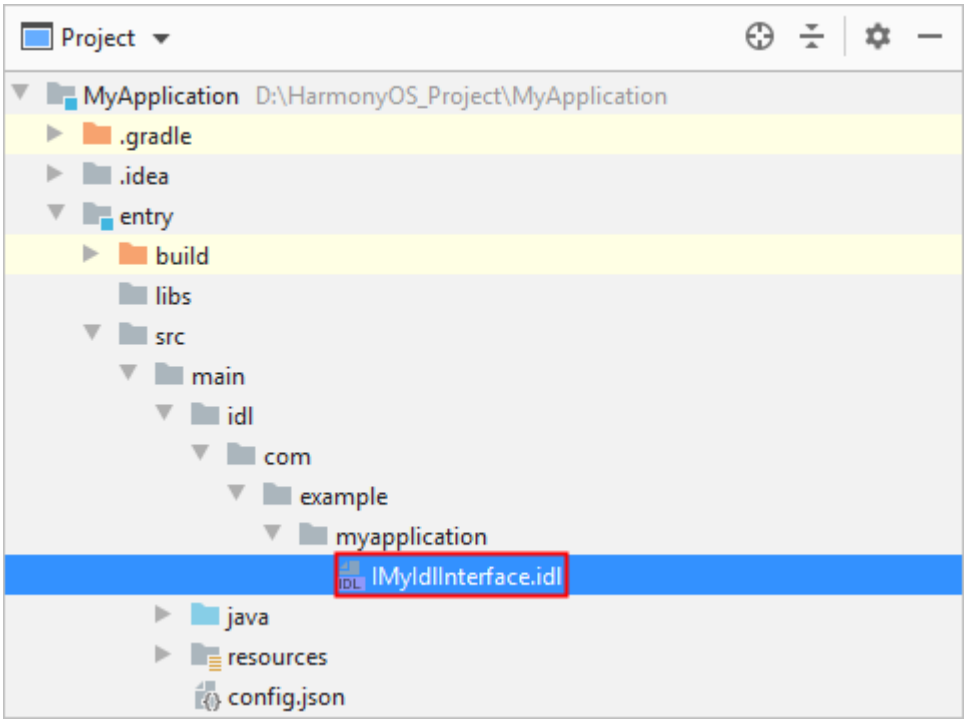


创建 **IDL File**。可以直接输入 IDL 接口名称，也可以通过包名格式定义 idl 接口名称。两种方式的差异仅在于.idl 文件的文件目录结构。

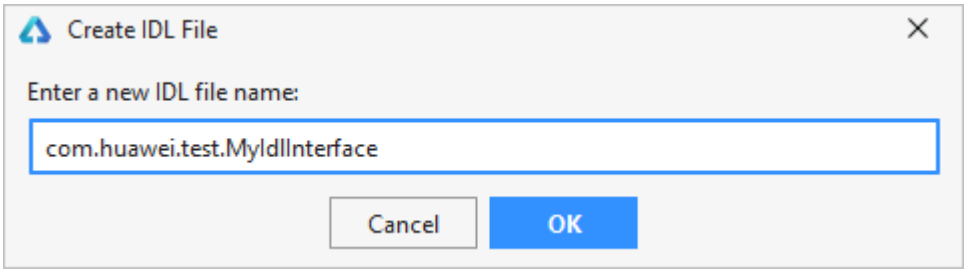
- 按名称创建，创建 **IDL File** 时，输入接口名称，直接点击 **OK**。



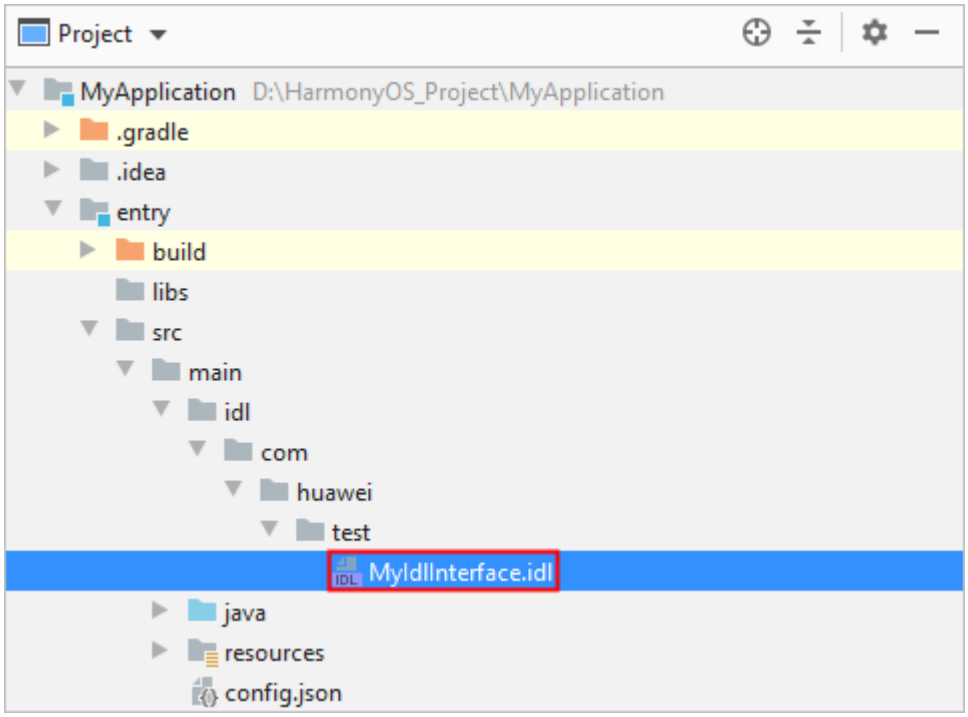
DevEco Studio 在相应“module”的 **src>main** 路径下生成 **idl** 文件夹，并按照对应模块的包名生成同样的目录结构及 **IDL** 文件。



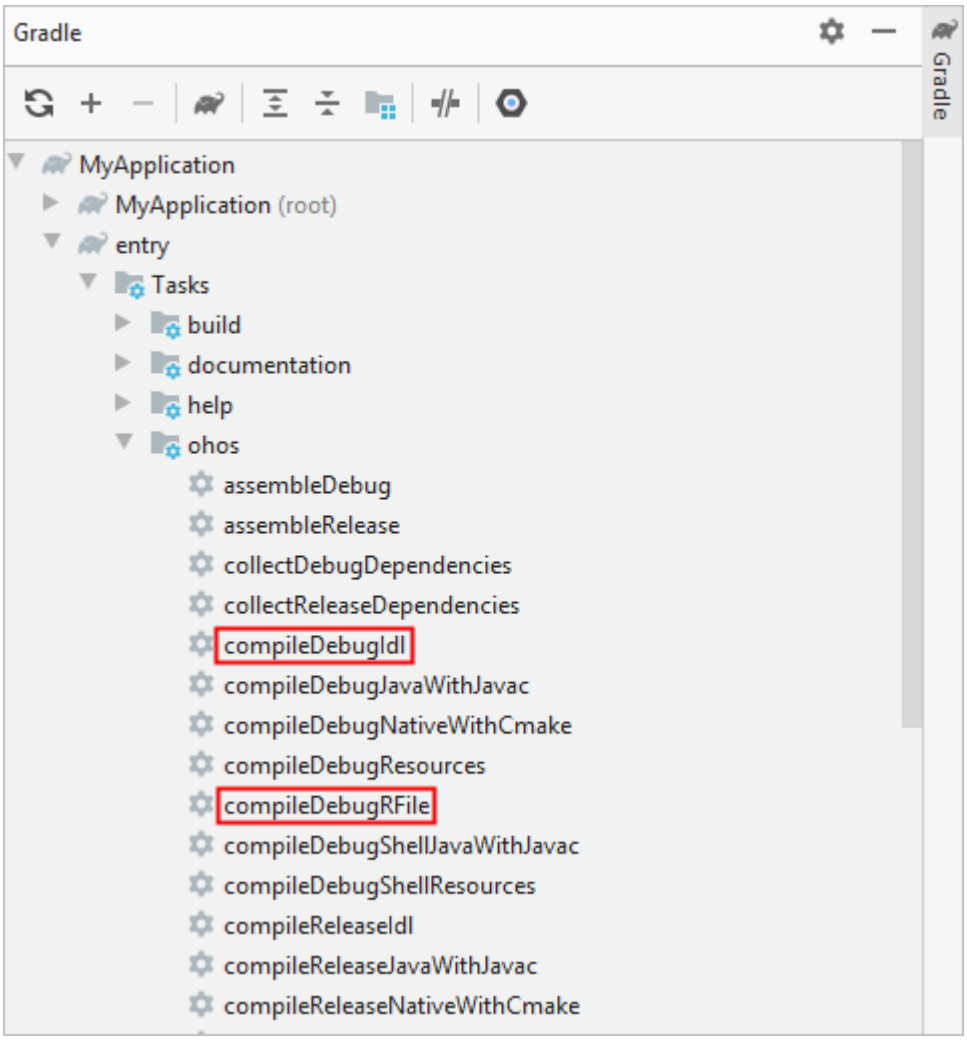
- 按包名创建，自定义.idl 文件存储路径和接口名称。创建“IDL File”时，按照包名创建 **IDL** 文件。包名利用“.”作为分隔符，如输入“com.huawei.test.MyIdlInterface”。



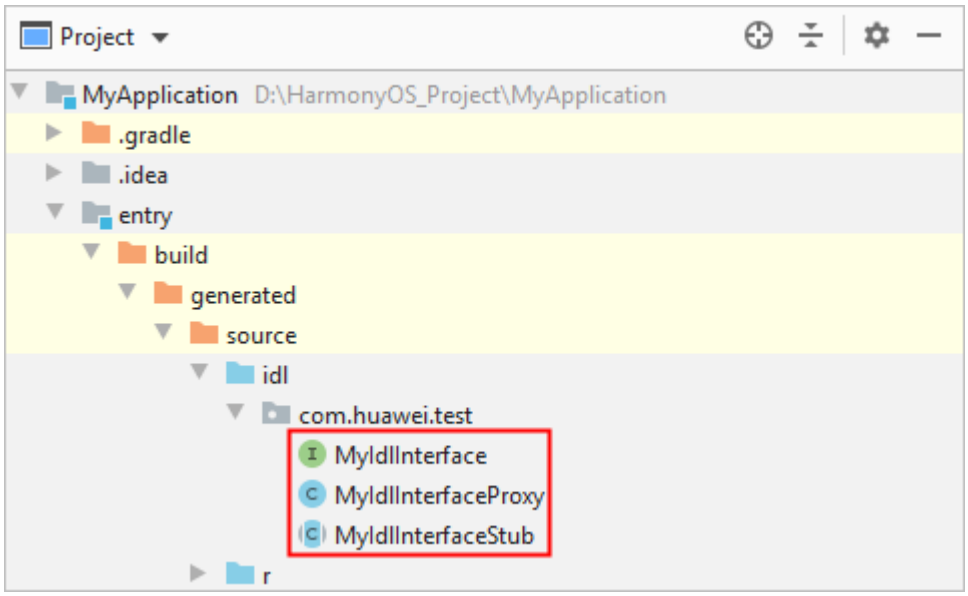
DevEco Studio 在相应“module”的 **src>main** 路径下生成 **idl** 文件夹，并按照输入的包名生成相应目录结构及 **IDL** 文件。可以在此路径继续新增 **IDL** 文件。



点击工程右边栏的 **Gradle**，在 **Tasks > ohos** 中选择 **compileDebugIdl** 或 **compileReleaseIdl**，对模块下的 **IDL** 文件进行编译。



编译完成后，在 **build > generated > source > Idl> {Package Name}**目录下，生成对应的接口类、桩类和代理类，如下图所示。



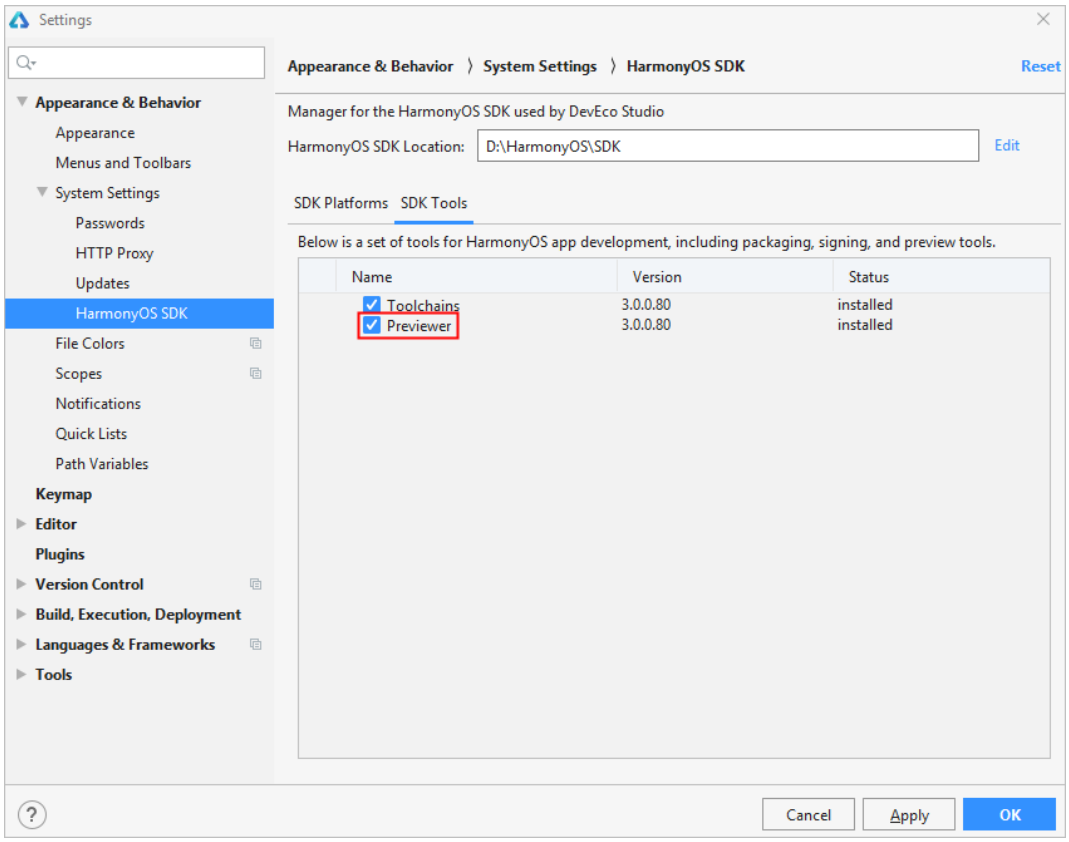
实现 HarmonyOS IDL 接口

开发者可以使用 Java 或 C++编程语言构建.idl 文件，关于 HarmonyOS IDL 接口的实现请参考 IDL 开发指南。

使用预览器查看应用效果

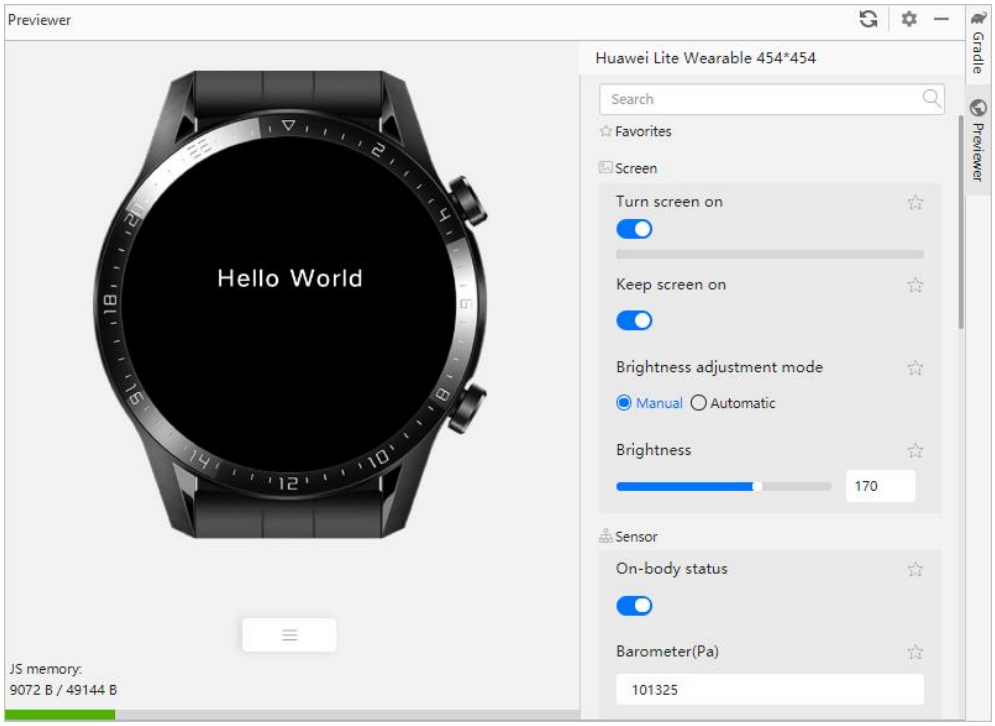
DevEco Studio 仅针对 Lite Wearable 提供预览器的功能。预览器支持代码热加载，在开发应用的同时，只要将开发的代码保存到源码中，即可在预览器中实时查看应用效果，方便开发者随时调整代码。

在使用预览器查看应用界面的 UI 效果前，需要确保 **HarmonyOS SDK > SDK Tools** 中，已下载 Previewer 资源，详情请参考下载 HarmonyOS SDK。



打开预览器有两种方式，显示效果如下图所示。

- 通过菜单栏，点击 **View>Tool Windows>Previewer**，打开预览器。
- 在编辑窗口右上角的侧边工具栏，点击 **Previewer**，打开预览器。

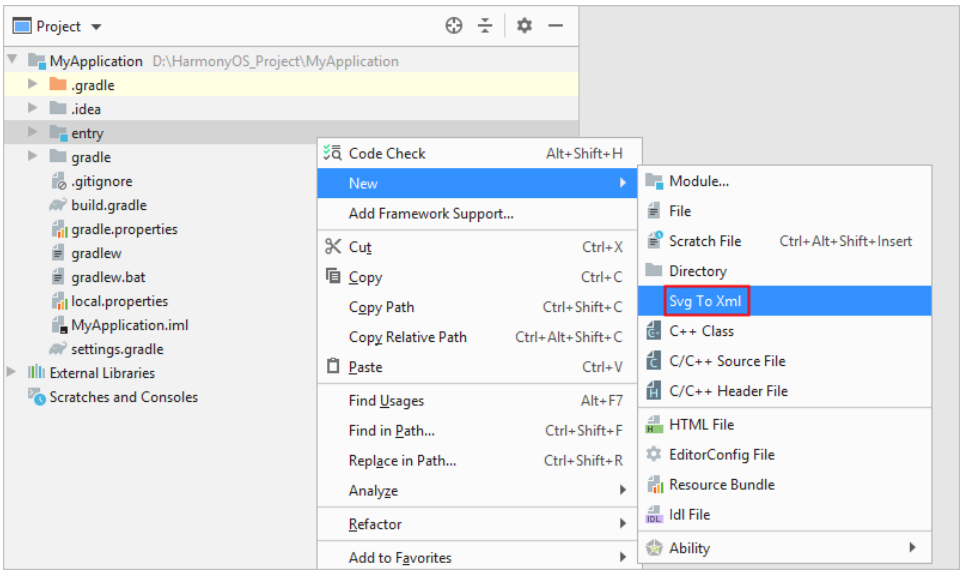


预览器还提供 Lite Wearable 的场景化数据注入功能，如点亮/关闭屏幕，调节屏幕亮度，向应用注入步数、心率、经纬度等信息，可以在开发阶段模拟真实的使用场景，便于开发者验证应用使用体验。

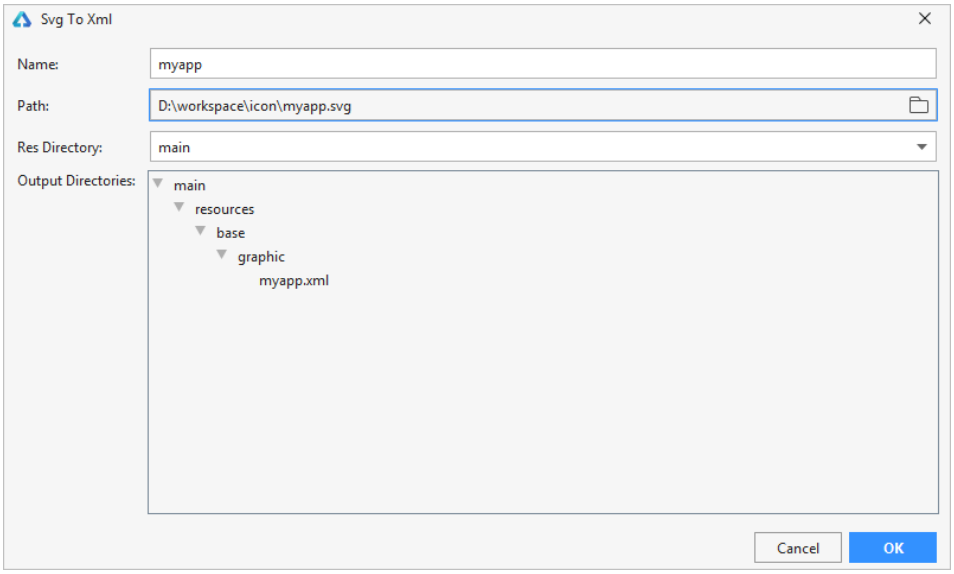
将 SVG 文件转换为 XML 文件

SVG（Scalable Vector Graphics）可缩放矢量图形，是一种图像文件格式。目前由于 HarmonyOS 图形渲染引擎不支持 SVG 格式图片的渲染，开发者需要将 SVG 格式的图片文件转为为 XML 格式的文件，然后在布局文件中引用转换后的 XML 文件。这样，就可以在模拟器/预览器或者设备上运行应用时，正常的渲染该图像文件。转换方法如下：

1. 选中应用模块，点击鼠标右键，选择 **New>Svg To Xml**。



选择需要转换的 svg 文件，并命名，点击 **OK** 按钮开始转换。



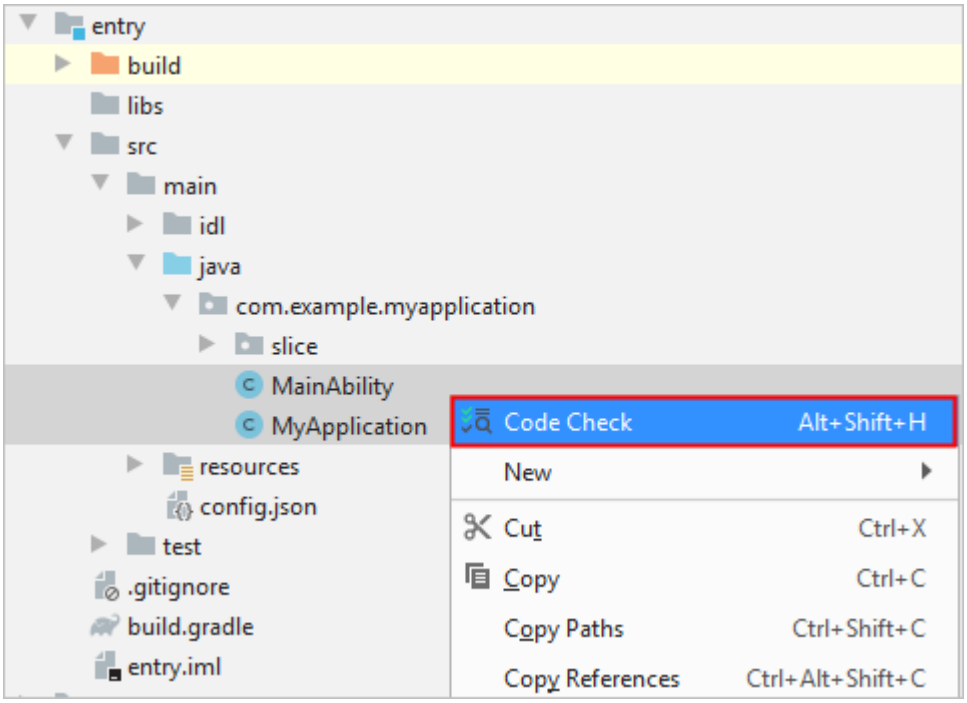
转换成功后，可以在 **resources > base > graphic** 文件下找到转换后的 xml 文件，并在布局文件中，引用该 xml 文件名即可完成对图标文件的引用。

代码安全检查

DevEco Studio 针对 Java 语言代码进行安全检查，扫描代码安全问题，并根据扫描结果提示进行修改，有助于开发提高代码的健壮性。常见的代码安全问题包括如下几类：

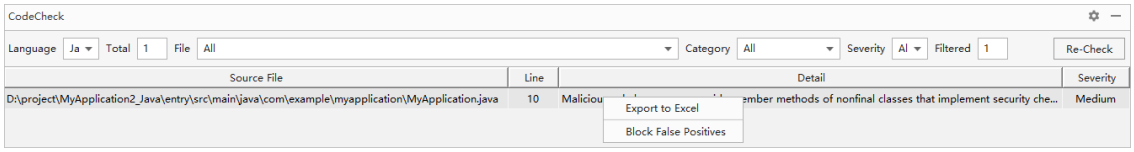
- 凭据管理
- 认证问题和会话管理
- 权限控制
- 加密问题
- 信息泄露
- 完整性保护
- 隐私保护
- 不正确输入校验
- 安全编译

检查方法：鼠标选中已打开的代码编辑文件、或者鼠标点击选中文件或文件夹，或者按 Ctrl+鼠标点击选中多个文件，然后点击鼠标右键，选中 **Code Check**。



扫描完成后：

- 双击某个扫描结果可以跳转到对应代码，可以根据 **Detail** 的建议进行修改。
- 如果某个扫描结果不需要修改，可以对该扫描结果进行屏蔽，屏蔽后再执行 **Code Check** 将不再显示该扫描结果。
- 在扫描结果处，点击鼠标右键，选择 **Block False Positives** 可以屏蔽该行的安全检查。
- 如需恢复安全检查屏蔽的错误信息，可以在**.idea>shield_config.xml** 中删除某条屏蔽信息，或者直接删除**.idea>shield_config.xml** 文件来删除全部屏蔽信息。



编译构建

编译构建概述

编译构建是将 HarmonyOS 应用的源代码、资源、第三方库等打包生成 HAP 或者 APP 的过程。其中，HAP 可以直接运行在真机设备或者模拟器中；APP 则是用于应用上架到华为应用市场。HAP 和 APP 的关系说明请参考 HarmonyOS 工程介绍。

为了确保 HarmonyOS 应用的完整性，HarmonyOS 通过数字证书和授权文件来对应用进行管控，只有签名过的 HAP 才允许安装到设备上进行运行（如果不带签名信息，仅可以运行在模拟器中）；同时，上架到华为应用市场的 APP 也必须通过签名才允许上架。因此，为了保证应用能够发布和安装到设备上， 需要提前申请相应的证书与 Profile 文件，详情请参考申请证书和 Profile。

申请证书和 Profile 文件时，用于调试和上架的证书与授权文件不能交叉使用：

- 应用调试证书与应用调试 Profile 文件、应用发布证书与应用发布 Profile 文件具有匹配关系，必须成对使用，不可交叉使用。
- 应用调试证书与应用调试 Profile 文件必须应用于调试场景，用于发布场景将导致应用发布审核不通过；应用发布证书与应用发布 Profile 文件必须应用于发布场景，用于调试场景将导致应用无法安装。

编译构建前配置

在进行 HarmonyOS 应用的编译构建前，需要对工程和编译构建的 Module 进行设置，请根据实际情况进行修改。

- build.gradle：HarmonyOS 应用依赖 gradle 进行构建，需要通过 build.gradle 来对工程编译构建参数进行设置。build.gradle 分为工程级和模块级两种类型，其中工程根目录下的工程级 build.gradle 用于工程的全局设置，各模块下的 build.gradle 只对本模块生效。
- config.json：应用清单文件，用于描述应用的全局配置信息、在具体设备上的配置信息和 HAP 的配置信息。

工程级 build.gradle

- apply plugin：在工程级 Gradle 中引入打包 app 的插件，不需要修改。
-

```
apply plugin: 'com.huawei.ohos.hap'
```

- ohos 闭包：工程配置，包括如下配置项：
- compileSdkVersion：依赖的 SDK 版本。

```
.compileSdkVersion 3     //应用编译构建的目标 SDK 版本
.     defaultConfig {
.         compatibleSdkVersion 3     //应用兼容的最低 SDK 版本
. }
```

- signingConfigs：发布 APP 时的签名信息，在编译构建生成 APP 中进行设置后自动生成。

- buildscript 闭包：Gradle 脚本执行依赖，包括 Maven 仓地址和插件。
-

```
.buildscript {
.    repositories {
.        maven {
.            url 'https://mirrors.huaweicloud.com/repository/maven/'
.        }
.        maven {
.            url 'https://developer.huawei.com/repo/'
.        }
.        jcenter()
.    }
.       dependencies {
.         classpath 'com.huawei.ohos:hap:2.0.0.6'
.    }
.}
```

- allprojects 闭包：工程自身所需要的依赖，比如引用第三方库的 Maven 仓库和依赖包。

```
.allprojects {
.    repositories {
.       maven {
.            url 'https://mirrors.huaweicloud.com/repository/maven/'
.       }
.       maven {
.            url 'https://developer.huawei.com/repo/'
.       }
```

```
.        jcenter()
.    }
.}
```

模块级 build.gradle

- apply plugin：在模块级 Gradle 中引入打包 hap 和 library 的插件，无需修改。

```
.apply plugin: 'com.huawei.ohos.hap'    //打包 hap 包的插件
.apply plugin: 'com.huawei.ohos.library'    //将 HarmonyOS Library 打包为 har 的插件
.apply plugin: 'com.huawei.ohos.java-library'    //将 Java Library 打包为 jar 的插件
```

- ohos 闭包：模块配置，包括如下配置项：
- compileSdkVersion：依赖的 SDK 版本。

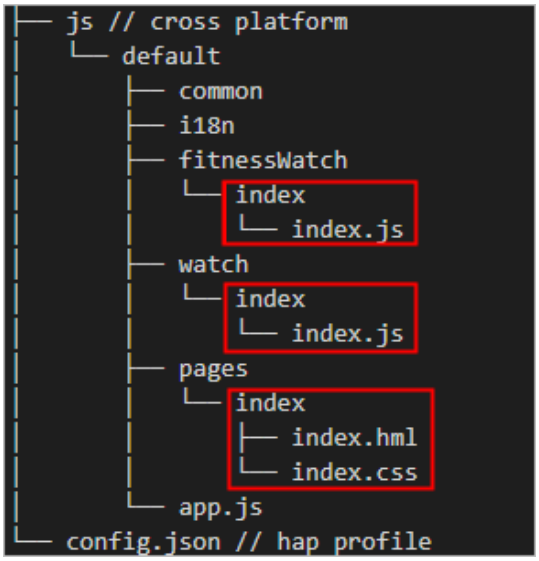
```
.compileSdkVersion 3    //应用编译构建的目标 SDK 版本
.    defaultConfig {
.        compatibleSdkVersion 3    //应用兼容的最低 SDK 版本
. }
```

- signingConfigs：在编译构建生成 HAP 中进行设置后自动生成。
- externalNativeBuild：C/C++编译构建代码设置项。

```
.externalNativeBuild {
.    path "src/main/cpp/CMakeLists.txt"    //CMake 配置入口，提供 CMake 构建脚本的相对路径
```

```
.     arguments "-v"     //传递给 CMake 的可选编译参数
.     abiFilters "arm64-v8a"      //用于设置本机的 ABI 编译环境
.     cppFlags ""    //设置 C++编译器的可选参数
  entryModules：该 Feature 模块关联的 Entry 模块。


.entryModules "entry"
```

- mergeJsSrc：跨设备的应用编译构建，是否需要合并 JS 代码。Wearable 和 Lite Wearable 共用一个工程，如下图所示。当进行编译构建时，将 Wearable/Lite Wearable 目录下的 JS 文件与 pages 目录（Wearable 和 Lite Wearable 共用的源码）下的 JS 文件进行合并打包。




```
.mergejssrc true     //合并 JS 代码打包时，请在 ohos 闭包下手动添加，true 表示需要合并 JS 代码，false 表示不需要合并 JS 代码。
  annotationEnabled：支持数据库注释。
```

```
.compileOptions{
.          annotationEnabled true   //true 表示支持，false 表示不支持
  dependencies 闭包：该模块所需的依赖项。


.dependencies {
.    entryImplementation project(':entry')    //该 Feature 模块依赖的
Entry 模块
.    implementation fileTree(dir: 'libs', include: ['*.jar','*.har'])
//该模块依赖的本地库，支持 jar 和 har 包
.    testCompile'junit:junit:4.12'    //测试用例框架，无需修改
.}
```

config.json 清单文件

HarmonyOS 应用的每个模块下包含一个 config.json 清单文件，在编译构建前，需要对照检查和修改 **config.json** 文件，详情请参考 **config.json 清单文件介绍**。

准备签名文件

生成密钥和证书请求文件

HarmonyOS 应用通过数字证书和授权文件来保证应用的完整性，在申请数字证书和 Profile 文件前，需要通过 DevEco Studio 来生成私钥（存放在.p12 文件中）和证书请求文件（.csr 文件）。同时，也可以使用命令行工具的方式来生成密钥和证书请求文件，用于构筑工程流水线。

使用 DevEco Studio 生成证书请求文件

使用 DevEco Studio 生成证书请求文件的方式有以下两种情况：

- 如果还未生成密钥文件，则可以一键生成密钥和证书请求文件。
- 如果已有密钥文件，则可以使用已有密钥生成证书请求文件。

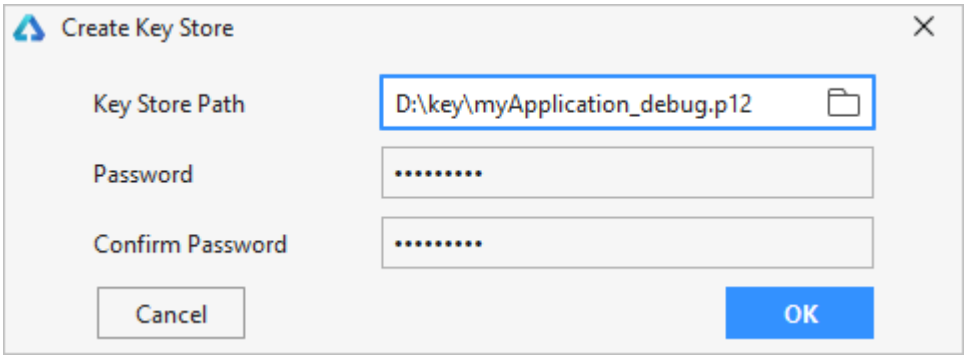
一键生成密钥和证书请求文件

1. 在主菜单栏点击 **Build > Generate Key**。

2. 在 **Key Store Path** 中，可以点击 **Choose Existing** 选择已有的密钥库文件；如果没有密钥库文件，点击 **New** 进行创建。下面以新创建密钥库文件为例进行说明。

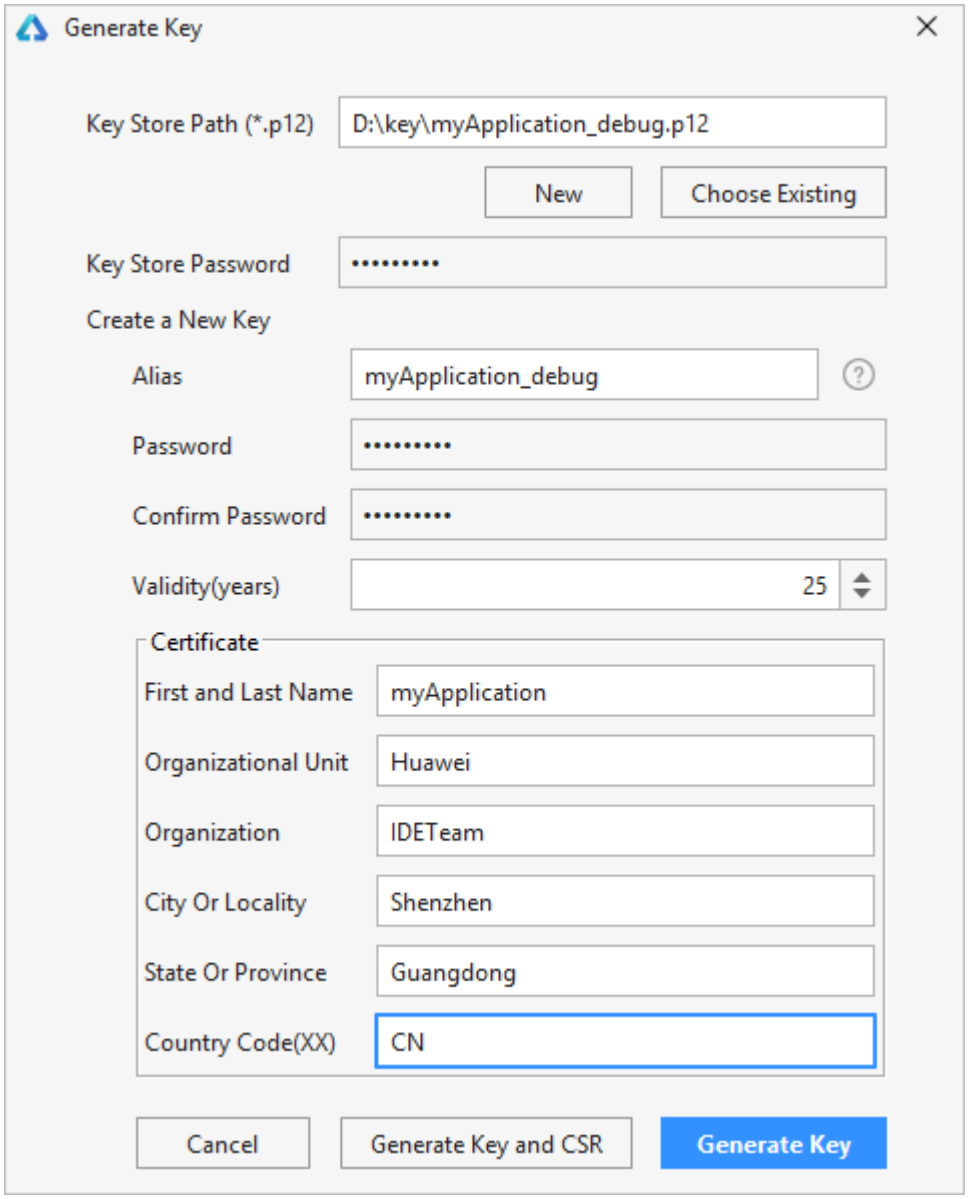
3. 在 **Create Key Store** 窗口中，填写密钥库信息后，点击 **OK**。

- **Key Store Path**：选择密钥库文件存储路径。
- **Password**：设置密钥库密码，必须由大写字母、小写字母、数字和特殊符号中的两种以上字符的组合，长度至少为 8 位。请记住该密码，后续签名配置需要使用。
- **Confirm Password**：再次输入密钥库密码。

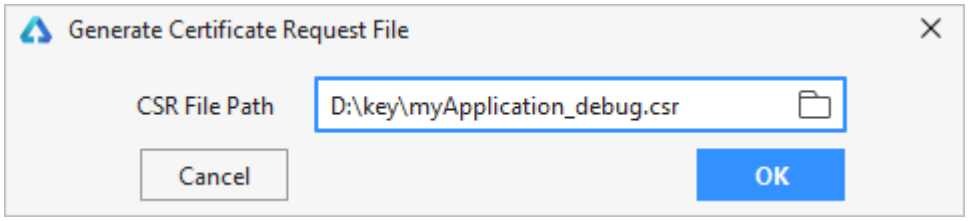


1. 在 **Generate Key** 界面中，继续填写密钥信息后，点击 **Generate Key and CSR**。

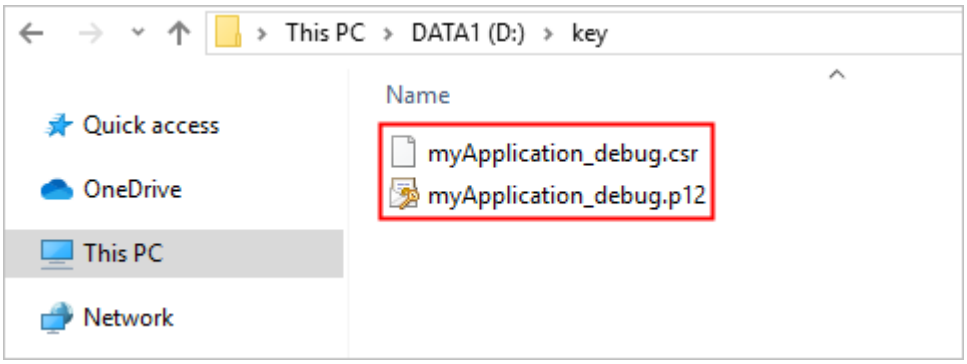
- **Alias**：密钥的别名信息，用于标识密钥名称。请记住该别名，后续签名配置需要使用。
- **Password**：输入密钥对应的密码，密钥密码需要与密钥库密码保持一致。请记住该密码，后续签名配置需要使用。
- **Confirm Password**：再次输入密钥密码。
- **Validity**：证书有效期，建议设置为 25 年及以上，覆盖应用的完整生命周期。
- **Certificate**：输入证书基本信息，如组织、城市或地区、国家码等。



在弹出的窗口中，点击 **CSR File Path** 对应的 图标，选择 CSR 文件存储路径。



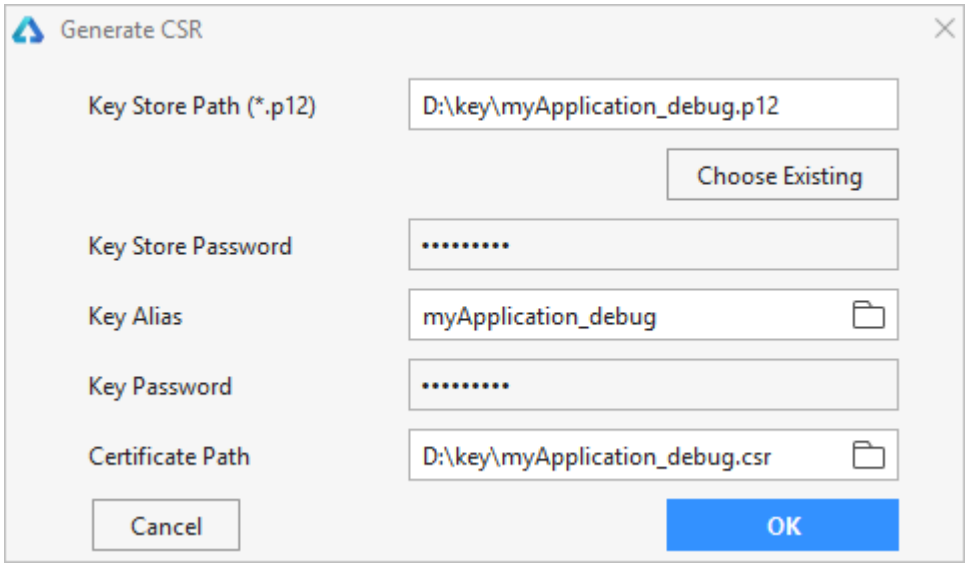
点击 **OK** 按钮，创建 CSR 文件成功，工具会同时生成密钥文件（.p12）和证书请求文件（.csr）。



使用已有密钥生成证书请求文件

1. 在主菜单栏点击 **Build > Generate Certificate Request File**。
2. 在 Generate CSR 界面，填写证书请求文件生成参数，点击 **OK**。

- **key Store Path**：点击 **Choose Existing** 选择已有的密钥库文件，后缀格式为.p12。
- **Key Store Password**：输入创建密钥时填写的密钥库密码。
- **Key Alias**：输入创建密钥时填写的别名信息。
- **Key Password**：输入创建密钥时填写的密钥密码。
- **Certificate Path**：点击 按钮，选择证书请求文件存储路径和名称。

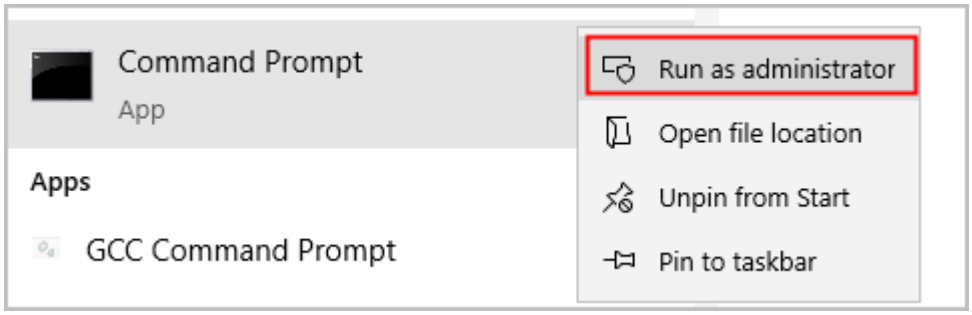


打开证书请求文件存储目录，获取证书请求文件（.csr 文件）。

使用命令行工具生成证书请求文件

使用 Open JDK 携带的 Keytool 工具生成证书请求文件。

1. 使用管理员身份运行命令行工具。



切换到 keytool 工具所在路径，实际路径请根据安装目录进行修改。

执行如下命令，生成密钥文件。例如，生成的密钥名称为 ide_demo_app.p12，存储到 D 盘根目录下。

```
keytool -genkeypair -alias "ide_demo_app" -keyalg EC -sigalg
SHA256withECDSA -dname "C=CN,O=HUAWEI,OU=HUAWEI IDE,CN=ide_demo_app"
-keystore d:\\idedemokey.p12 -storetype pkcs12 -validity 9125 -storepass
123456 -keypass 123456
```

生成密钥文件的参数说明如下：

说明

请记录下 **alias**、**storepass** 和 **keypass** 的值，后续编译构建生成 HAP 和编译构建生成 APP 会使用到。

- **alias**：密钥的别名信息，用于标识密钥名称。
- **sigalg**：签名算法，固定为 **SHA256withECDSA**。
- **dname**：按照操作界面提示进行输入。
- C：国家/地区代码，如 CN。
- O：组织名称，如 HUAWEI。
- OU：组织单位名称，如 HUAWEI IDE。
- CN：名字与姓氏，建议与别名一致。
- **validity**：证书有效期，建议设置为 9125（25 年）。
- **storepass**：设置密钥库密码。
- **keypass**：设置密钥的密码，请与 **storepass** 保持一致。

1. 执行如下命令，执行后需要输入 **storepass** 密码，生成证书请求文件，后缀格式为.csr。

```
keytool -certreq -alias "ide_demo_app" -keystore d:\\idedemokey.p12
-storetype pkcs12 -file d:\\idedemokey.csr
```

.

生成证书请求文件的参数说明如下：

- **alias**：与 3 中输入的 alias 保持一致。
- **file**：生成的证书请求文件名称，后缀为.csr。

申请证书和 Profile

目前华为应用市场只支持 Lite Wearable（轻量级智能穿戴）的 HarmonyOS 应用的上架，因此只支持轻量级智能穿戴设备的证书和 Profile 文件申请。智慧屏、智能穿戴等设备的证书、Profile 文件的申请，以及对应应用上架功能，敬请期待。

对于 Lite Wearable 的调试、发布证书和 Profile 的申请，请参考：

- 申请调试证书和 Profile
- 申请发布证书和 Profile

说明

在申请证书和 Profile 前，请根据生成密钥和证书请求文件准备好证书请求文件。

编译构建生成 HAP

HAP 可以直接在模拟器或者真机设备上运行，用于 HarmonyOS 应用开发阶段的调试和查看运行效果。HAP 按构建类型和是否签名可以分为以下四种形态：

- **构建类型为 Debug 的 HAP**（带调试签名信息）：携带调试签名信息，具备单步调试等调试手段的 HAP，用于开发者在真机或者模拟器中进行应用调试。
- **构建类型为 Debug 的 HAP**（不带签名）：不带调试签名信息，具备单步调试等调试手段的 HAP，仅能运行在模拟器中。
- **构建类型为 Release 的 HAP**（带调试签名信息）：携带调试签名信息，不具备调试能力的 HAP，用于开发者在真机或者模拟器中查看和验证应用运行效果。相对于 Debug 类型的 HAP 包，体积更小，运行效果与用户实际体验一致。
- **构建类型为 Release 的 HAP**（不带签名）：不带调试签名信息，不具备调试能力的 HAP，仅能运行在模拟器中查看和验证应用运行效果。相对于 Debug 类型的 HAP 包，体积更小，运行效果与用户实际体验一致。

根据 HarmonyOS 工程介绍，一个 HarmonyOS 工程下可以存在多个 Module，在编译构建时，可以选择对单个 Module 进行编译构建；也可以对整个工程进行编译构建，同时生成多个 HAP。

前提条件

- 已完成 build.gradle 和 config.json 的设置，详情请参考编译构建前配置。
- 已完成调试证书和 Profile 文件的申请，详情请参考申请证书和 Profile。

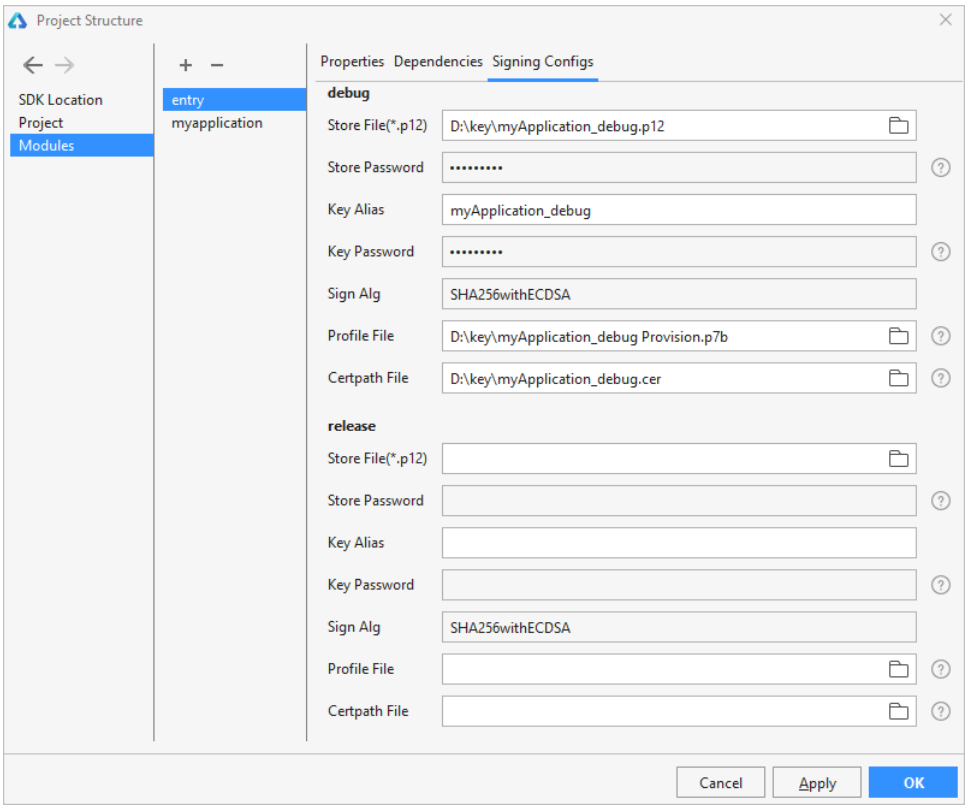
构建类型为 Debug 的 HAP（带调试签名信息）

如果一个工程目录下存在多个 Module，当对单个 Module 进行构建时，只需要对指定的 Module 进行签名；如果对整个工程进行构建，则需要对所有的 Module 进行签名。

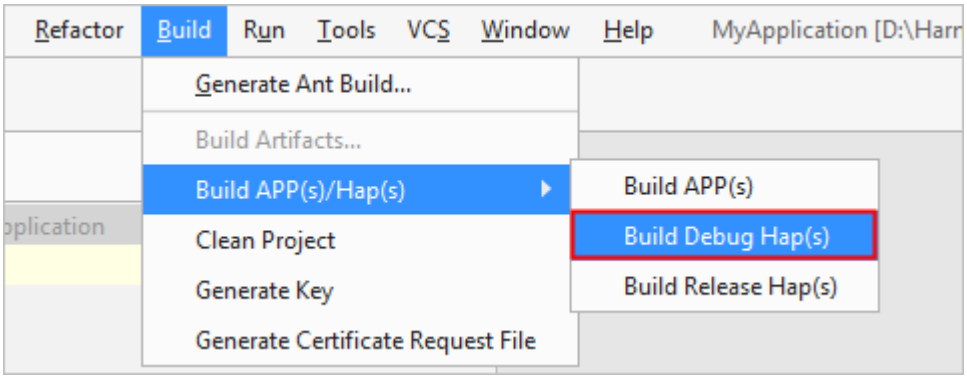
1. 打开 **File>Project Structure**，在 **Modules>entry（模块名称）>Signing Configs > debug** 窗口中，配置指定模块的调试签名信息。

- **Store File**：选择密钥库文件，文件后缀为.p12。

- **Store Password**：输入密钥库密码。
- **Key Alias**：输入密钥的别名信息。
- **Key Password**：输入密钥的密码。
- **SignAlg**：签名算法，固定为 SHA256withECDSA。
- **Profile File**：选择申请的调试 Profile 文件，文件后缀为.p7b。
- **Certpath File**：选择申请的调试数字证书文件，文件后缀为.cer。



1. 在主菜单栏，点击 **Build > Build APP(s)/Hap(s) > Build Debug Hap(s)**，生成已签名的 Debug HAP。



构建类型为 Debug 的 HAP（不带签名）

对于构建类型为 Debug 的 HAP，如果没有配置签名参数，则默认不对 HAP 进行签名，该方式生成的 HAP 仅能运行在模拟器上。

在主菜单栏，点击 **Build > Build APP(s)/Hap(s) > Build Debug Hap(s)**，生成不带签名的调试 Debug HAP。

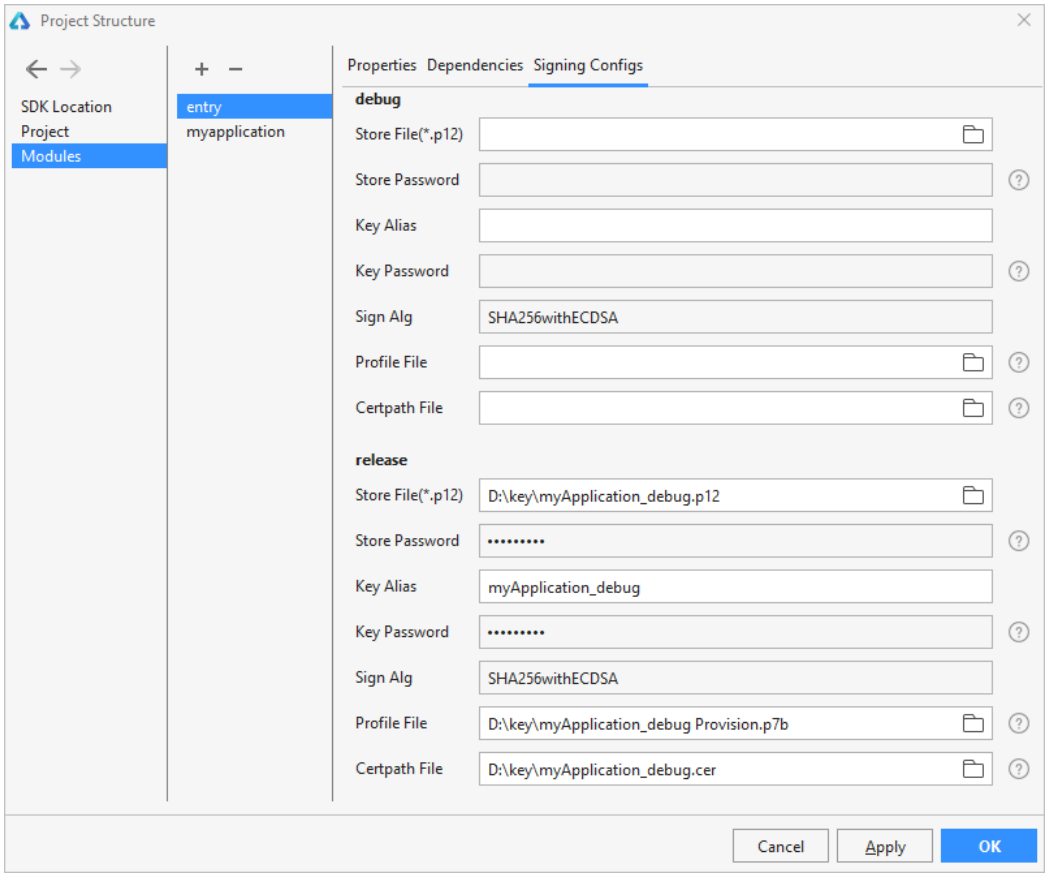
构建类型为 Release 的 HAP（带调试签名信息）

如果一个工程目录下存在多个 Module，当对单个 Module 进行构建时，只需要对指定的 Module 进行签名；如果对整个工程进行构建，则需要对所有的 Module 进行签名。

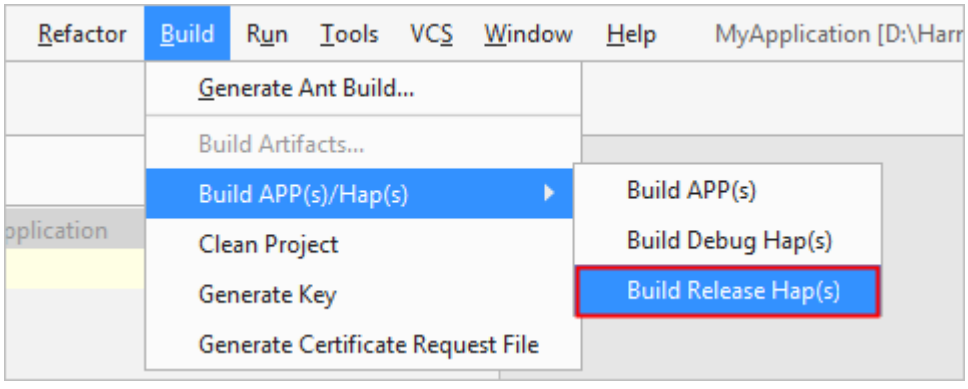
1. 打开 **File>Project Structure**，在 **Modules>entry（模块名称）>Signing Configs > release** 窗口中 ，配置指定模块的调试签名信息。

- **Store File**：选择密钥库文件，文件后缀为.p12。
- **Store Password**：输入密钥库密码。
- **Key Alias**：输入密钥的别名信息。
- **Key Password**：输入密钥的密码。
- **SignAlg**：签名算法，固定为 SHA256withECDSA。
- **Profile File**：选择申请的调试 Profile 文件，文件后缀为.p7b。

- **Certpath File**：选择申请的调试数字证书文件，文件后缀为.cer。



1. 在主菜单栏，点击 **Build > Build APP(s)/Hap(s) > Build Release Hap(s)**，生成已签名的 Release HAP。



构建类型为 Release 的 HAP（不带签名）

对于构建类型为 Release 的 HAP，如果没有配置签名参数，则默认不对 HAP 进行签名，该方式生成的 HAP 仅能运行在模拟器上。

在主菜单栏，点击 **Build > Build APP(s)/Hap(s) > Build Release Hap(s)**，生成不带签名的调试 Release HAP。

应用运行

使用模拟器运行应用

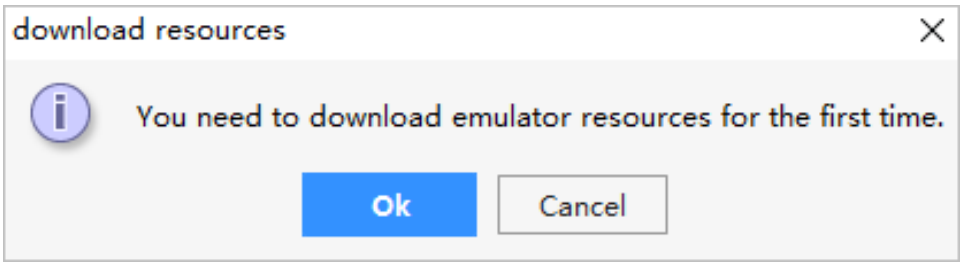
DevEco Studio 提供远程模拟器（**Remote Emulator**）功能，可以将开发的 TV 和 Wearable 应用运行在模拟器上。在模拟器上运行应用不需要签名。

模拟器每次使用时长为 1 小时，到期后模拟器会自动释放资源，请及时完成 HarmonyOS 应用的调试。如果模拟器到期释放后，需重新申请模拟器资源。（查看使用远程模拟器的常见问题）

说明

Lite Wearable 暂不支持在模拟器中运行，可以选择预览器运行和调试应用，具体请参见使用预览器查看应用效果。

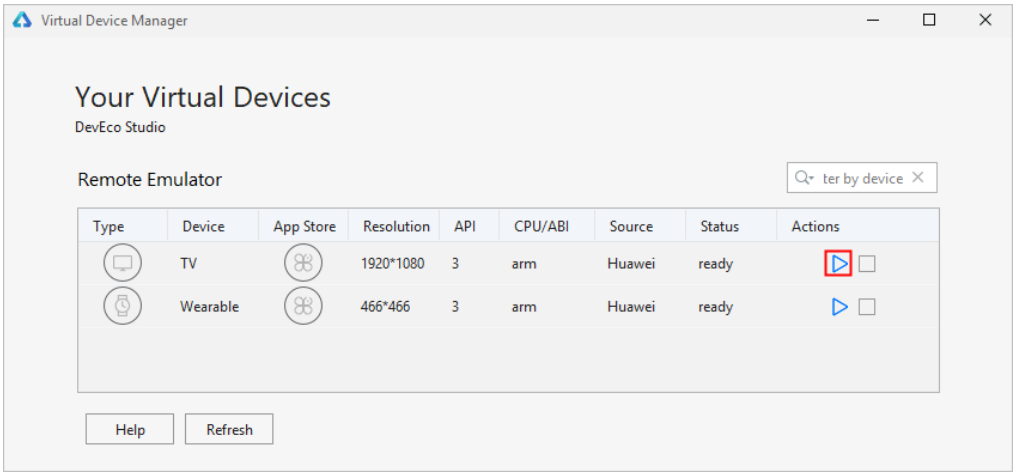
在 DevEco Studio 菜单栏，点击 **Tools > HVD Manager**。首次使用模拟器，请点击 **OK** 按钮下载模拟器相关资源。



在浏览器中弹出华为帐号登录界面，请输入已实名认证的华为帐号的用户名和密码进行登录。

登录后，请点击界面的**允许**按钮进行授权。

点击已经连接的远程模拟设备运行按钮 ▷，启动远程模拟设备（同一时间只能启动一个设备）。



点击 DevEco Studio 的 **Run > Run'模块名称'**或 ▶，或使用默认快捷键 **Shift+F10**。

在弹出的 Select Deployment Target 界面选择 **Connected Devices**，点击 **OK** 按钮。

DevEco Studio 会启动应用的编译构建，完成后应用即可运行在 **Remote Device** 上。

模拟器侧边栏按钮作用：

：释放当前正在使用的模拟器，每台模拟器单次使用时长为 1 小时。

：设置模拟器分辨率。

：返回模拟器主界面。

：后退按钮。

使用真机设备运行应用

在 TV 中运行应用

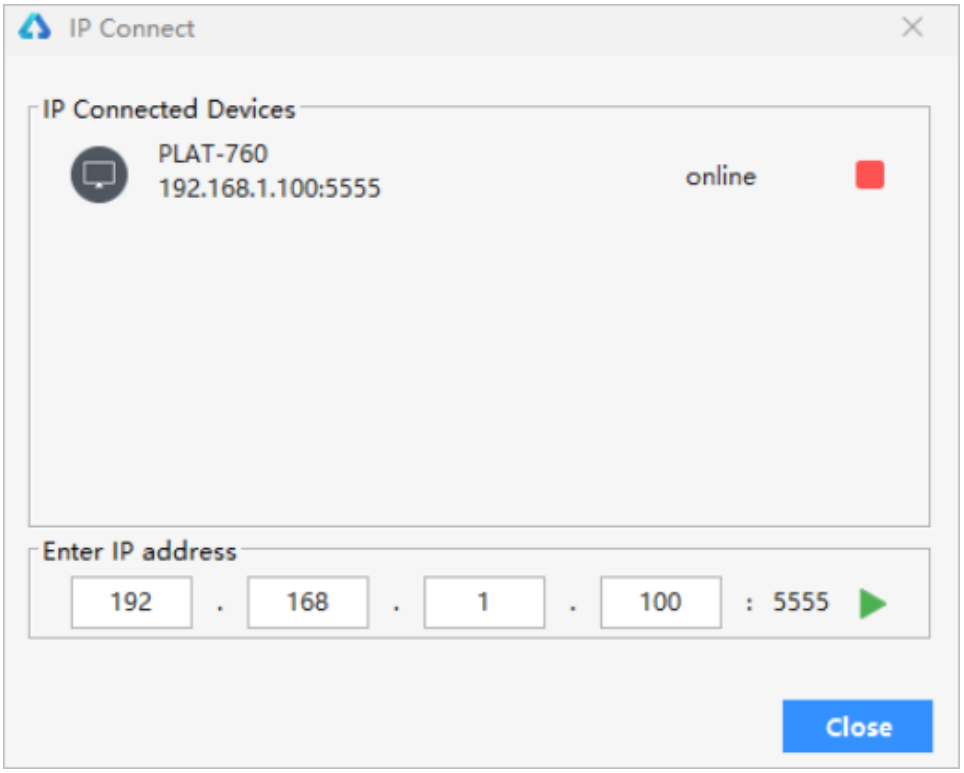
在 TV（智慧屏）中安装和运行 HarmonyOS 应用，采用 **IP Connect** 的连接方式。该连接方式要求 TV 和 PC 端在同一个网段，建议将 TV 和 PC 连接到同一个 WLAN 下；如果采用网线连接，需要手动设置 PC 和 TV 的本地 IP 地址。

前提条件

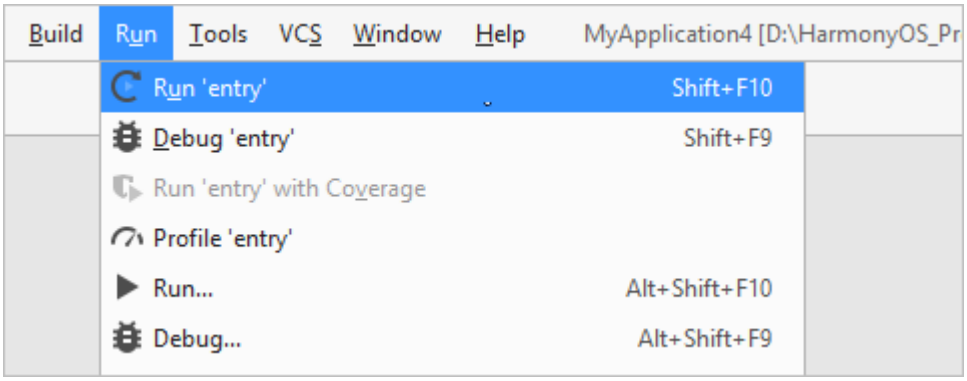
- 已将 TV 和 PC 连接到同一网络或设置为同一个网段。
- 已获取 TV 端的 IP 地址。
- 在 TV 中运行应用，需要提前根据编译构建生成 HAP 完成 HAP 的签名配置。

操作步骤

1. 在 DevEco Studio 菜单栏中，点击 **Tools>IP Connect**，输入连接设备的 IP 地址，点击  ，连接正常后，设备状态为 **online**。



在菜单栏中，点击 **Run>Run'模块名称'**或 ▶，或使用默认快捷键 **Shift+F10** 运行应用。



在弹出的界面，选择已连接的 TV 设备，点击 **OK** 按钮。

DevEco Studio 启动 HAP 的编译构建和安装。安装成功后，点击 TV 桌面上的应用图标，运行 HarmonyOS 应用。

在 Wearable 中运行应用

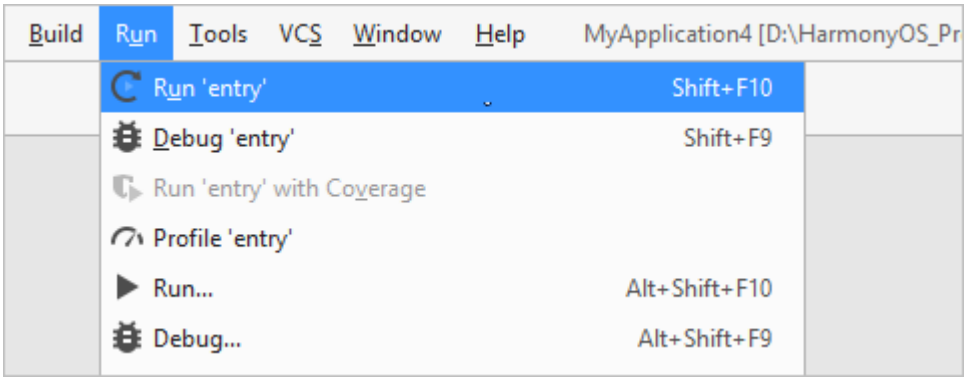
在 Wearable 中安装和运行 HarmonyOS 应用，采用 **USB** 连接或采用 **WLAN** 的连接方式。

前提条件

在 Wearable 中运行应用，需要提前根据编译构建生成 HAP 章节，完成 HAP 的签名配置。

采用 USB 连接安装应用

1. 使用 USB 方式，连接 Wearable 和 PC 端。
2. 在菜单栏中，点击 **Run>Run'模块名称'**或 ▶，或使用默认快捷键 **Shift+F10** 运行应用。



1. 在弹出的界面，选择已连接的 Wearable 设备，点击 **OK** 按钮。
2. DevEco Studio 启动 HAP 的编译构建和安装。安装成功后，点击 Wearable 中的应用图标，运行 HarmonyOS 应用。

采用 WLAN 连接安装应用

方法与在 TV 中运行应用类似，只是连接的设备为 Wearable。

在 Lite Wearable 中运行应用

Lite Wearable 的 HarmonyOS 应用安装，依赖华为手机上的**运动健康**和**应用调测助手** APP 辅助进行。

前提条件

- 已将**运动健康 APP** 升级至最新版本。
- 从华为应用市场安装**应用调测助手 APP**。
- 在 Lite Wearable 中运行应用，需要提前根据编译构建生成 HAP 完成 HAP 的签名配置。

操作步骤

1. 使用 USB 连接线将手机和电脑进行连接，确保连接状态是正常的。
2. 手机与电脑使用 USB 连接时，在手机上选择**传输文件**连接方式。
3. 在工程目录中的 **Build > outputs >hap** 中选择生成的 HAP，通过手工拷贝的方式将 HAP 拷贝至手机中的“/sdcard/haps/”目录。

说明

1.

如果在手机存储根目录下没有“haps”文件夹，请手工创建后再拷贝 HAP 到该文件夹下。

将 Lite Wearable 通过蓝牙与华为手机进行连接。

- 进入**运动健康** APP，在设备页签中，点击**添加设备**按钮。

- 进入**手表**列表中，选择对应的 Lite Wearable 型号。
- 点击**开始配对**，按照界面指引完成 Lite Wearable 与华为手机之间的连接。

1. 打开**应用调测助手** APP，界面会显示已经与华为手机连接的 Lite Wearable。

> 说明
>
> 如果 Lite Wearable 与华为手机未连接，请点击应用调测助手 APP 界面的连接设备按钮，手机会自动打开运动健康 APP 添加 Lite Wearable。

1. 点击**应用调测助手** APP 界面中的**安装手表应用**按钮，选择需要安装的 HarmonyOS 安装包进行安装。
2. 安装完成后，点击 Lite Wearable 中的应用图标，运行 HarmonyOS 应用。

应用调试

基本调试操作

DevEco Studio 提供了基于各种编写代码及不同设备的调试功能，如果使用了多种代码编写应用，请参考选择调试代码类型进行配置后启动调试，调试过程中基于不同的代码进行断点管理。

选择调试代码类型

点击 **Run > Edit Configurations > Debugger**，在 **HarmonyOS App** 中，选择相应模块， 可以进行 Java/JS/C++调试配置。

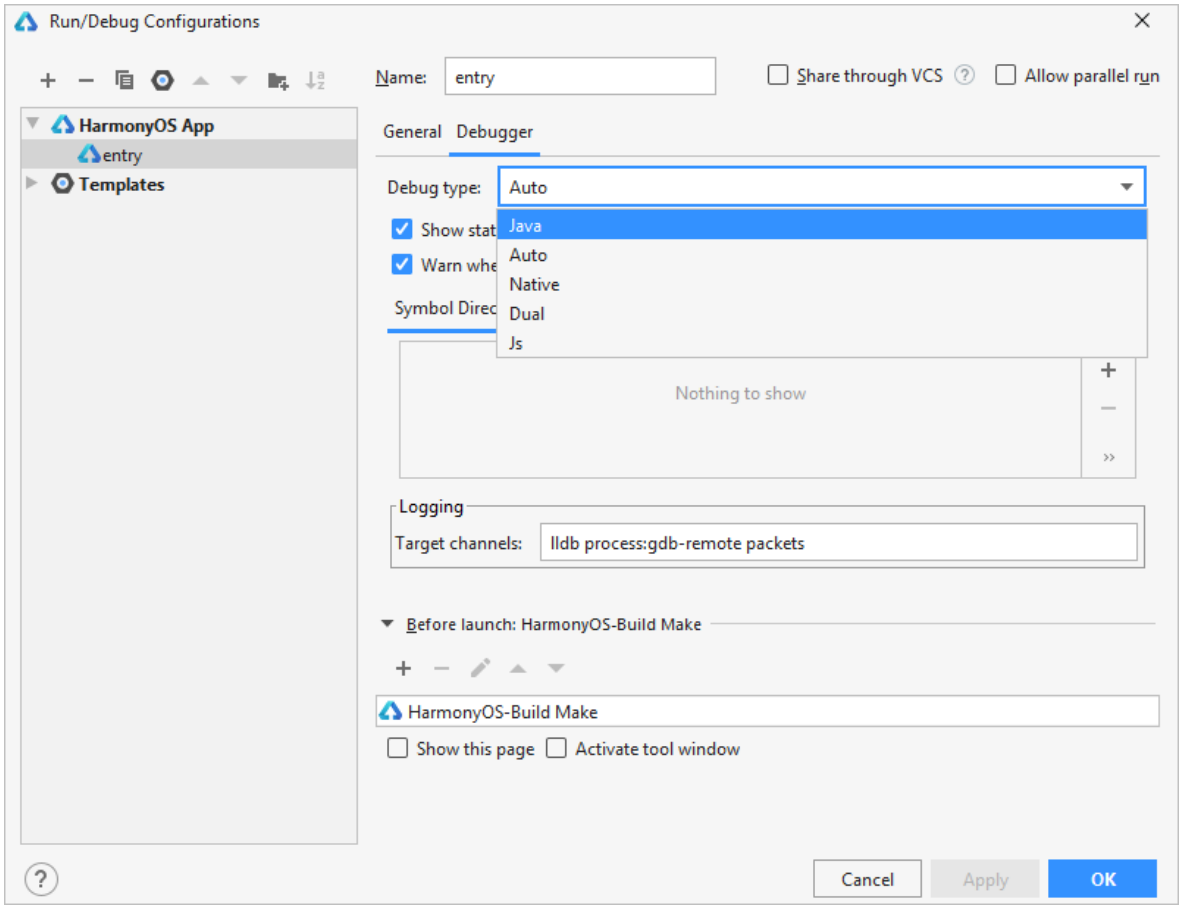


表 1 Java/Auto/Native/Dual/Js 调试类型配置项

调试类型	调试代码
Java	Java
Auto	Java C/C++ 根据代码自动匹配调试类型
Native	C/C++
Dual	C/C++ Java 同时调试 C/C++ 代码与 Java 代码
Js	JavaScript

- 对于 TV 和 Wearable 设备，请根据应用编写的代码来配置调试类型，然后进行调试。
- 对于 Lite Wearable 设备，与调试类型配置无关，可直接进行调试。

启动调试

1. 在工具栏中，点击 **Debug**  。

2. 在弹出的界面，选择需要调试的设备。

- 真实设备：一般为可以用 USB 或 IP 方式连接的实体设备。
- Remote Device ：远程设备模拟器，支持 TV 和 Wearable，请参考使用模拟器运行应用启动连接设备后，方可选择进行调试。

3. 如果需要设置断点调试，则需要选定要设置断点的有效代码行，在行号（比如：24 行）的区域后，单击鼠标左键设置断点（如图示的红点）。

```
1  package com.example.myapplication.slice;
2
3  import ...
11
12 public class MainAbilitySlice extends AbilitySlice {
13
14     private PositionLayout myLayout = new PositionLayout( context: this);
15
16     @Override
17     public void onStart(Intent intent) {
18         super.onStart(intent);
19         LayoutConfig config = new LayoutConfig(LayoutConfig.MATCH_PARENT, LayoutConfig.MATCH_PARENT);
20         myLayout.setLayoutConfig(config);
21         ShapeElement element = new ShapeElement();
22         element.setShape(ShapeElement.RECTANGLE);
23         element.setRgbColor(new RgbColor( red: 255,  green: 255,  blue: 255));
24         myLayout.setBackground(element);
25
26         Text text = new Text( context: this);
27         text.setText("Hello World");
28         text.setTextColor(Color.BLACK);
29         myLayout.addComponent(text);
30         super.setUIContent(myLayout);
31     }
```

设置断点后，调试能够在正确的断点处中断，并高亮显示该行。

断点管理

在设置的程序断点红点处，点击鼠标右键，然后点击 **More**（或按快捷键 **Ctrl+Shift+F8**），可以管理断点。

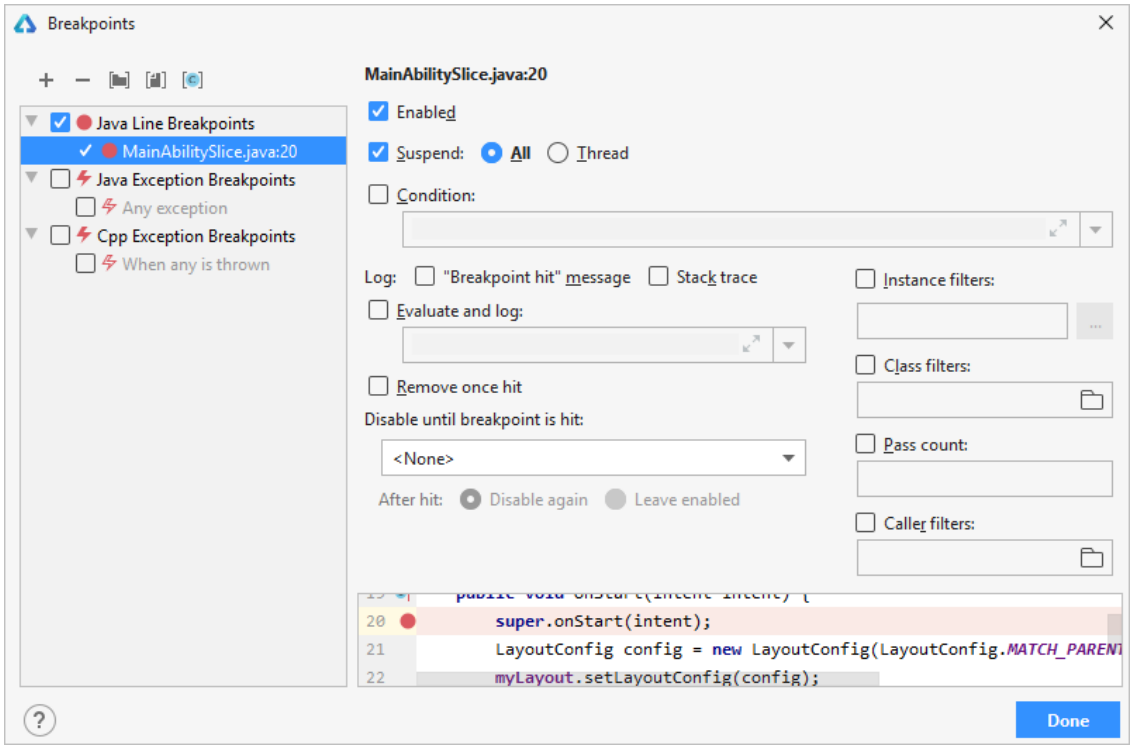


表 2 不同代码类型的断点管理功能

代码类型	断点管理
JS（JavaScript）	普通行断点
Java	普通行断点 Exception（异常）断点
C/C++	普通行断点 Exception（异常）断点 Symbolic（符号）断点 设置 Watchpoint（仅支持 x86、x86_64 架构）

各语言调试功能

JS 调试功能

对 JS 进行调试的界面如下：

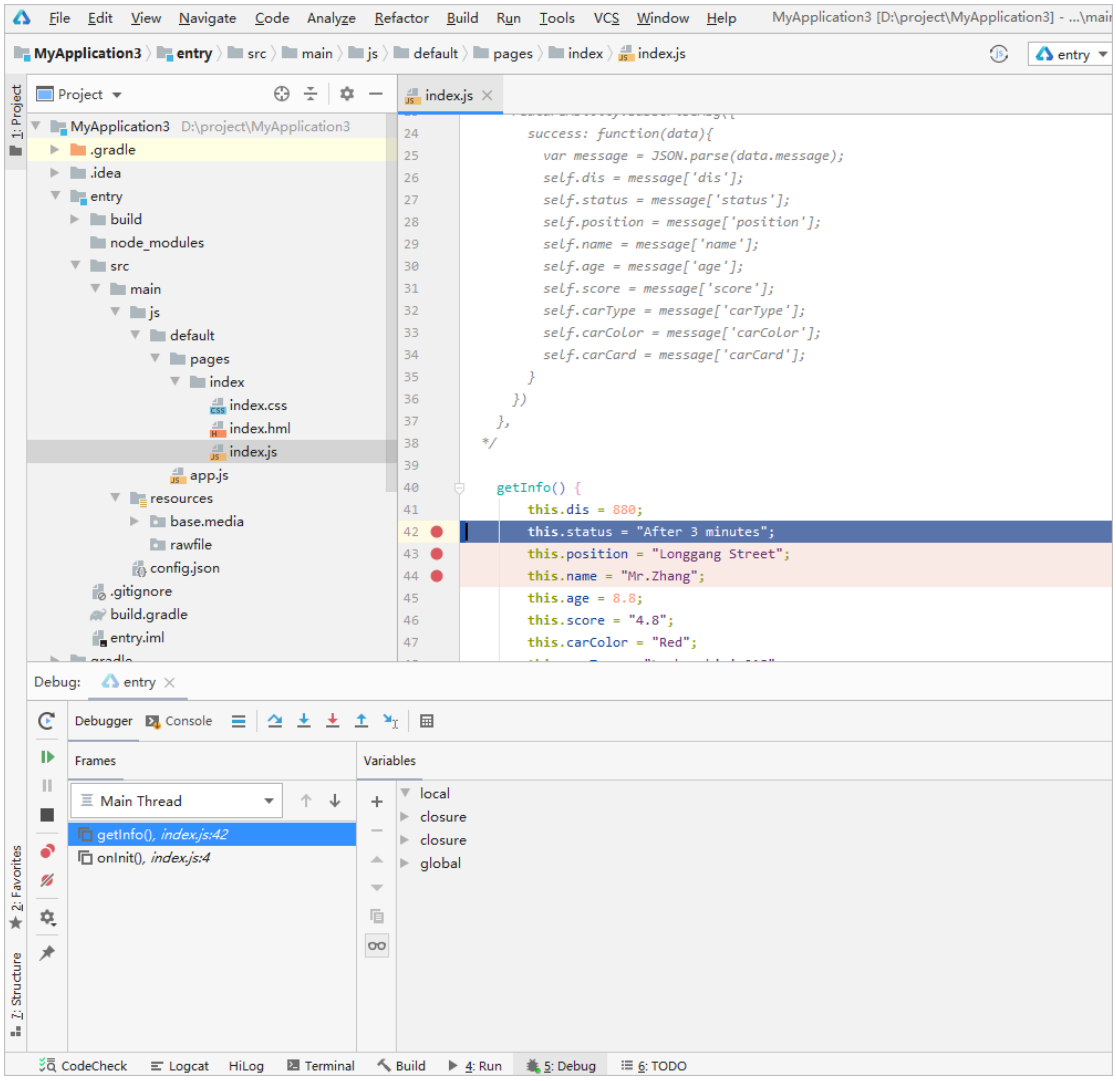


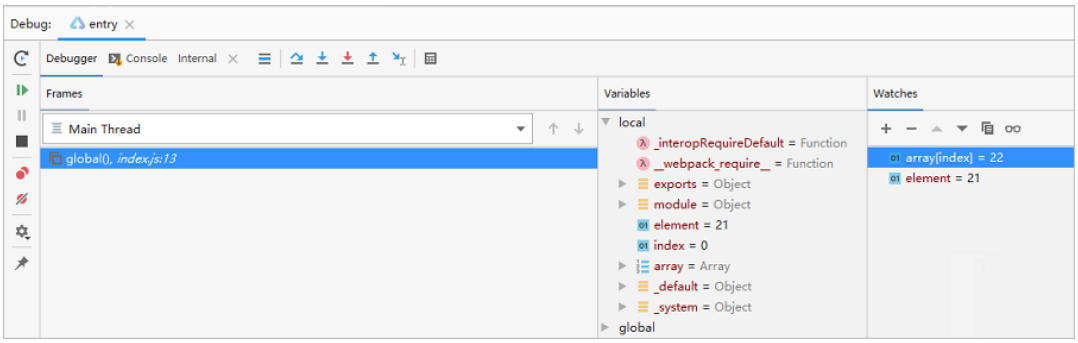
表 1 调试器按钮

按钮	名称	快捷键	功能

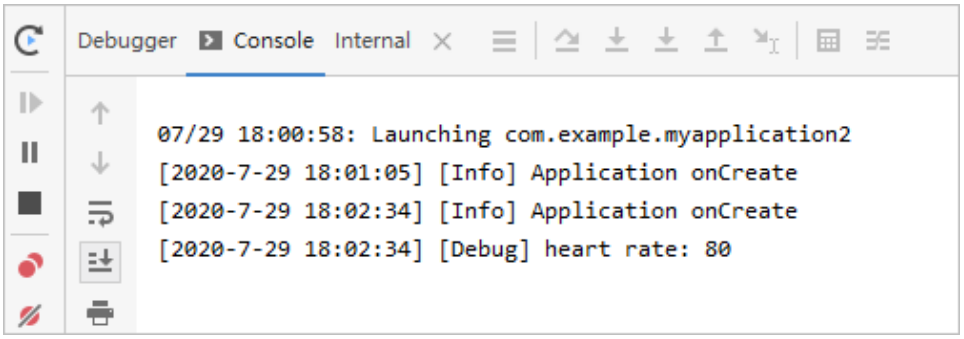
表 1 调试器按钮

按钮	名称	快捷键	功能
	Resume Program	**F9**	当程序执行到断点时停止执行，点击此按钮程序继续执行。
	Step Over	**F8**	在单步调试时，直接前进到下一行（如果在函数中存在子函数时，不会进入子函数内单步执行，而是将整个子函数当作一步执行）。
	Step Into	**F7**	在单步调试时，遇到子函数后，进入子函数并继续单步执行。
	Force Step Into	**Alt+Shift+F7**	在单步调试时，强制下一步。
	Step Out	**Shift+F8**	在单步调试执行到子函数内时，点击 Step Out 会执行完子函数剩余部分，并跳出返回到上一层函数。
	Rerun	**Ctrl+F5**	重新启动调试。
	Stop	**Ctrl+F2**	停止调试任务。
	Run To Cursor	-	断点执行到鼠标停留处，仅 TV、Wearable 支持。

- 常用的调试功能：
- **变量值查看：**在调试过程中，可以通过调试侧边栏中的 **Variables** 查看已执行程序中包含的变量的当前取值。
- **变量监控：**也可以在 **Watches** 中添加关注的变量，对添加的变量进行监控。
- **调用栈信息查看：**可以在 **Frames** 中查看函数的调用栈信息。



- **调试日志打印：**调试控制台 **Console** 可以打印调试的日志信息。



Java 调试功能

- 通过 **Attach Debugger to Process** 选择进程进行调试，能根据调试类型，在已运行应用的设备上，自动进入相应的调试模式。

- 具备 Step Into, Step Out, Step Over, Force Step Into, Rerun、Run To Cursor 等基本调试能力，详细描述请参考表 调试器按钮。
- 支持 Inline Values，即编辑器显示变量值。
- 调试中断后，能够恢复执行。

C/C++调试功能

- 通过 **Attach Debugger to Process** 选择进程进行调试，能根据调试类型，在已运行应用的设备上，自动进入相应的调试模式。

- Native 类型调试器，能启动 Debug Session 和 LLDB Server 运行调试。
- 具备 Step Into, Step Out, Step Over, Force Step Into, Rerun、Run To Cursor 等基本调试能力，详细描述请参考表 1。
- 支持 Force Step Over。
- 支持 Inline Values，即编辑器显示变量值。
- 调试中断后，能够恢复执行。
- LLDB 命令控制台：
- 支持使用 LLDB 命令自助调试
- 支持 UI 调试按钮/快捷键多指令输入

应用发布

编译构建生成 APP

开发者完成 HarmonyOS 应用开发后，需要将应用打包成 APP，用于发布到华为应用市场。打包 APP 时，DevEco Studio 会将工程目录下的所有 HAP 模块打包到 APP 中，因此，如果工程目录中存在不需要打包到 APP 的 HAP 模块，请手动删除后再进行编译构建生成 APP。

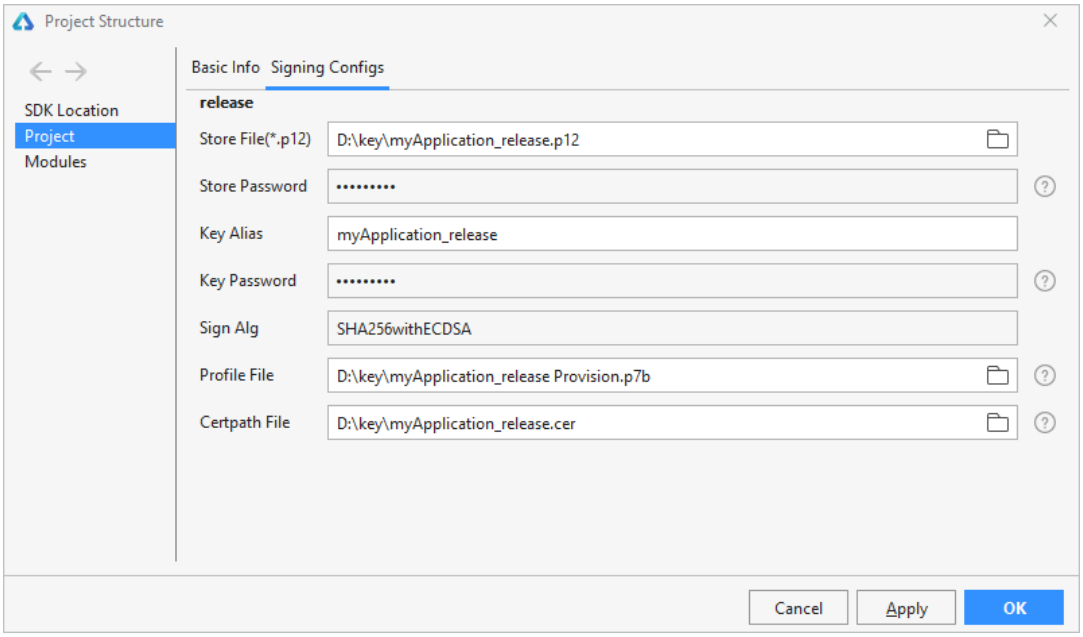
前提条件

- 已完成发布证书和 Profile 文件的申请，详情请参考申请证书和 Profile。
- 已完成 build.gradle 和 config.json 的设置，详情请参考编译构建前配置。

操作步骤

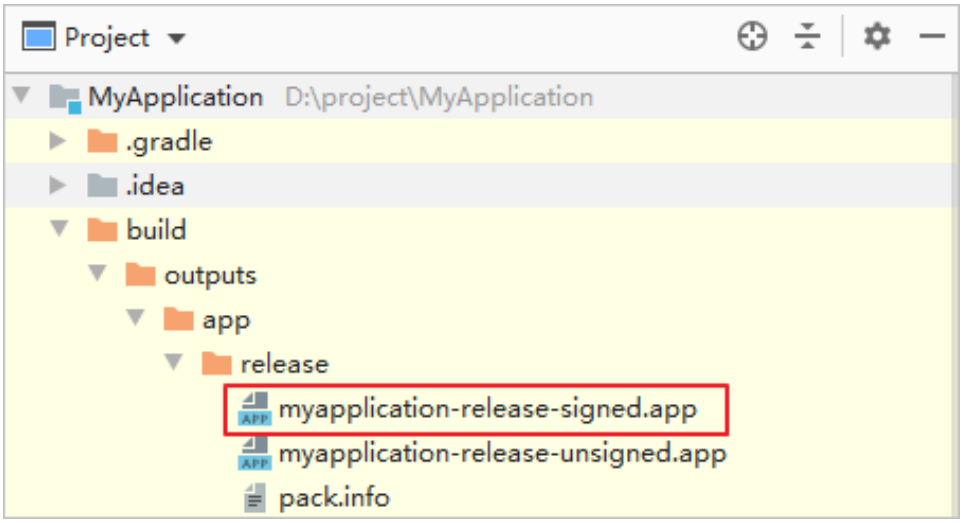
1. 在 **Project Structure > Project > Signing Configs** 窗口中，配置工程的签名信息，设置完成后，点击 **OK** 按钮。

- **Store File**：选择密钥库文件，文件后缀为.p12。
- **Store Password**：输入密钥库密码。
- **Key Alias**：输入密钥的别名信息。
- **Key Password**：输入密钥的密码。
- **SignAlg**：签名算法，固定为 SHA256withECDSA。
- **Profile File**：选择申请的发布 Profile 文件，文件后缀为.p7b。
- **Certpath File**：选择申请的发布数字证书文件，文件后缀为.cer。



点击 **Build > Build APP(s)/Hap(s) > Build APP(s)**，等待编译构建完成已签名的 APP。

编译构建完成后，可以在 **build > outputs > app > release** 目录下，获取带签名的 APP。



上架华为应用市场

将 HarmonyOS 应用打包成 APP 后，通过 AppGallery Connect 将 HarmonyOS 应用分发到不同的设备上。您可以根据发布 HarmonyOS 应用指导将 APP 上架到华为应用市场。

术语

A

- **Ability**

应用的重要组成部分，是应用所具备能力的抽象。Ability 分为两种类型，Feature Ability 和 Particle Ability。

- **AbilityForm**

表单，是 Feature Ability 的一种界面展示形式，用于嵌入到其他应用中并作为其界面的一部分显示，并支持基础的交互功能。

- **AbilitySlice**

切片，是单个可视化界面及其交互逻辑的总和，是 Feature Ability 的组成单元。一个 Feature Ability 可以包含一组业务关系密切的可视化界面，每一个可视化界面对应一个 AbilitySlice。

- **ANS**

Ability Notification Service，是 HarmonyOS 中负责处理通知的订阅、发布和更新等操作的系统服务。

C

- **CES**

Common Event Service，是 HarmonyOS 中负责处理公共事件的订阅、发布和退订的系统服务。

D

- **DV**

Device Virtualization，设备虚拟化，通过虚拟化技术可以实现不同设备的能力和资源融合。

F

- **FA**

Feature Ability，元程序，代表有界面的 Ability，用于与用户进行交互。

H

- **HAP**

HarmonyOS Ability Package，一个 HAP 文件包含应用的所有内容，由代码、资源、三方库及应用配置文件组成，其文件后缀名为.hap。

- **HDF**

HarmonyOS Driver Foundation，HarmonyOS 驱动框架，提供统一外设访问能力和驱动开发、管理框架。

I

- **IDN**

Intelligent Distributed Networking，是 HarmonyOS 特有的分布式组网能力单元。开发者可以通过 IDN 获取分布式网络内的设备列表以及注册分布式网络内设备在网状态变化信息。

M

- **MSDP**

Mobile Sensing Development Platform，移动感知平台。MSDP 子系统提供两类核心能力：分布式融合感知和分布式设备虚拟化两大部分。

- 分布式融合感知：借助 HarmonyOS 分布式能力，将各设备感知源进行汇总融合，对用户的空间状态、移动状态、手势、健康状态等进行精准感知，构建全场景泛在基础感知能力，支撑智慧生活新体验。

- 分布式器件虚拟化：借助 HarmonyOS 分布式能力，构筑器件虚拟化平台，将外部设备的各类器件（如 Camera、显示器、SPK/MIC 等）虚拟化为本地设备的器件延伸使用。同时具备将自身器件共享给其他设备使用的能力。

P

- **PA**

Particle Ability，元服务，代表无界面的 Ability，主要为 Feature Ability 提供支持，例如作为后台服务提供计算能力，或作为数据仓库提供数据访问能力。

S

- **SA**

System Ability，即系统能力，是由 OS 提供的基础软件服务和硬件服务。

- **Super virtual device，超级虚拟终端**

亦称超级终端，通过分布式技术将多个终端的能力进行整合，存放在一个虚拟的硬件资源池里，根据业务需要统一管理和调度终端能力，来对外提供服务。

常见问题

环境安装

无法自动下载 SDK 和相关工具，如何解决？
检查是否成功连接 Internet 网络，如果所在网络被管控，不能直接访问外网，请参考配置 DevEco Studio 代理后进行重试。

Java SDK 下载正常，但是 JS SDK 下载失败，如何解决？
JS SDK 下载失败，可能存在以下原因：

- 未安装 Node.js，请根据下载和安装 Node.js 进行处理。
- 您的网络受限，需要通过配置代理才能访问，请根据 npm 代理设置进行处理。

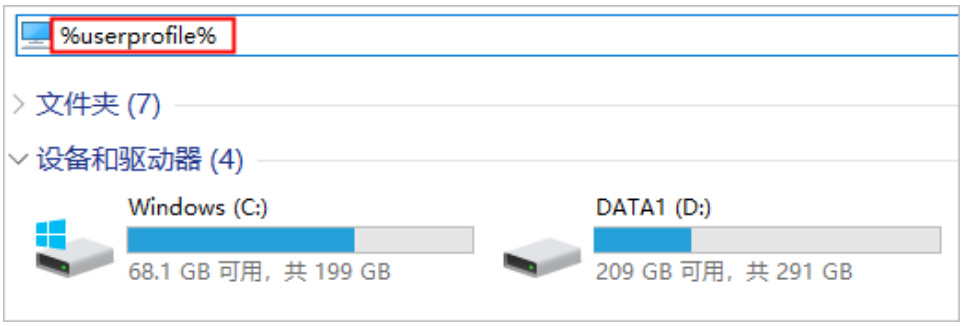
下载 JS SDK 时，JS 依赖下载缓慢，如何解决？
对于国内用户，可以将 npm 仓库设置为华为公有云仓库。在命令行工具中执行如下命令，重新设置 npm 仓库地址后，再执行 JS SDK 的下载。

```
npm config set registry https://mirrors.huaweicloud.com/repository/npm/
```

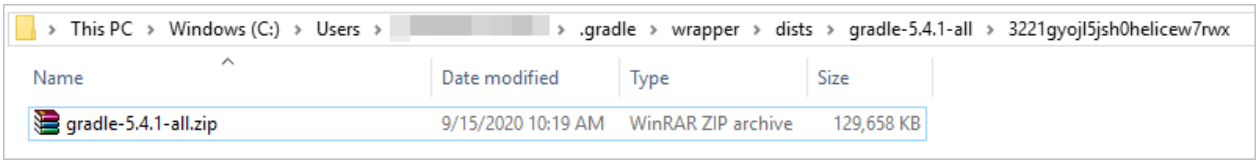
1. Gradle 插件下载失败，如何解决？

Gradle 下载失败，可能存在如下两种原因：

- 网络受限，请检查网络设置或者 DevEco Studio 代理设置。
- 网络正常，但是通过 DevEco Studio 下载缓慢或失败，可以通过如下方式解决。
- 点击链接下载 Gradle 插件，建议使用下载工具进行下载。
- 打开“此电脑”，在文件夹地址栏中输入**%userprofile%**，进入个人数据界面。



- 进入 **.gradle > wrapper > dists > gradle-5.4.1-all** 目录，将下载的“gradle-5.4.1-all.zip”拷贝到该目录下临时文件夹中。如果存在多个临时文件夹，建议每个文件夹都拷贝一份。



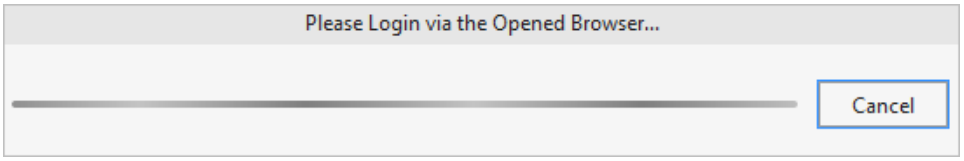
重启 DevEco Studio，等待工程同步完成。

模拟器运行

1. 点击 Tools > HVD Manager 登录过程中一直卡住不动，如何解决？

问题现象

使用模拟器前需跳转至浏览器登录页面，使用个人实名帐号进行登录，但模拟器的登录一直长时间处于如下图所示状态。



解决措施

可能存在如下几种原因：

- 未点击**允许**按钮：通过浏览器登录个人实名帐号后，需要点击**允许**按钮进行授权。

- 首次实名认证完成，间隔时间较短：进行实名认证后，请等待约 10 分钟后再重新登录。登录成功后，点击左下角的 **Refresh** 按钮即可获取远程模拟器设备。
- 可能 cookie 跨域被浏览器禁止：请点击浏览器地址栏中的 按钮，检查 op.hicloud.com 是否设置为允许，如果被禁止，请设置为允许。

- 当前用户登录超时：在 DevEco Studio 中，点击 **Tools > DevEco Login >Personal Center**，先退出登录；然后再点击 **Tools > HVD Manager** 重新登录。

HVD Manager 跳转到华为帐号验证界面没有允许按钮，如何解决？

可能存在以下两种情况：

建议将 Chrome 浏览器设置为默认浏览器，并使用 Chrome 浏览器重新登录。

若刚完成实名认证，请等待约 10 分钟后再重新使用。

模拟器屏幕显示“The device may be in standby or screen-off state, please touch or drag the screen to wake up your device”，如何解决？

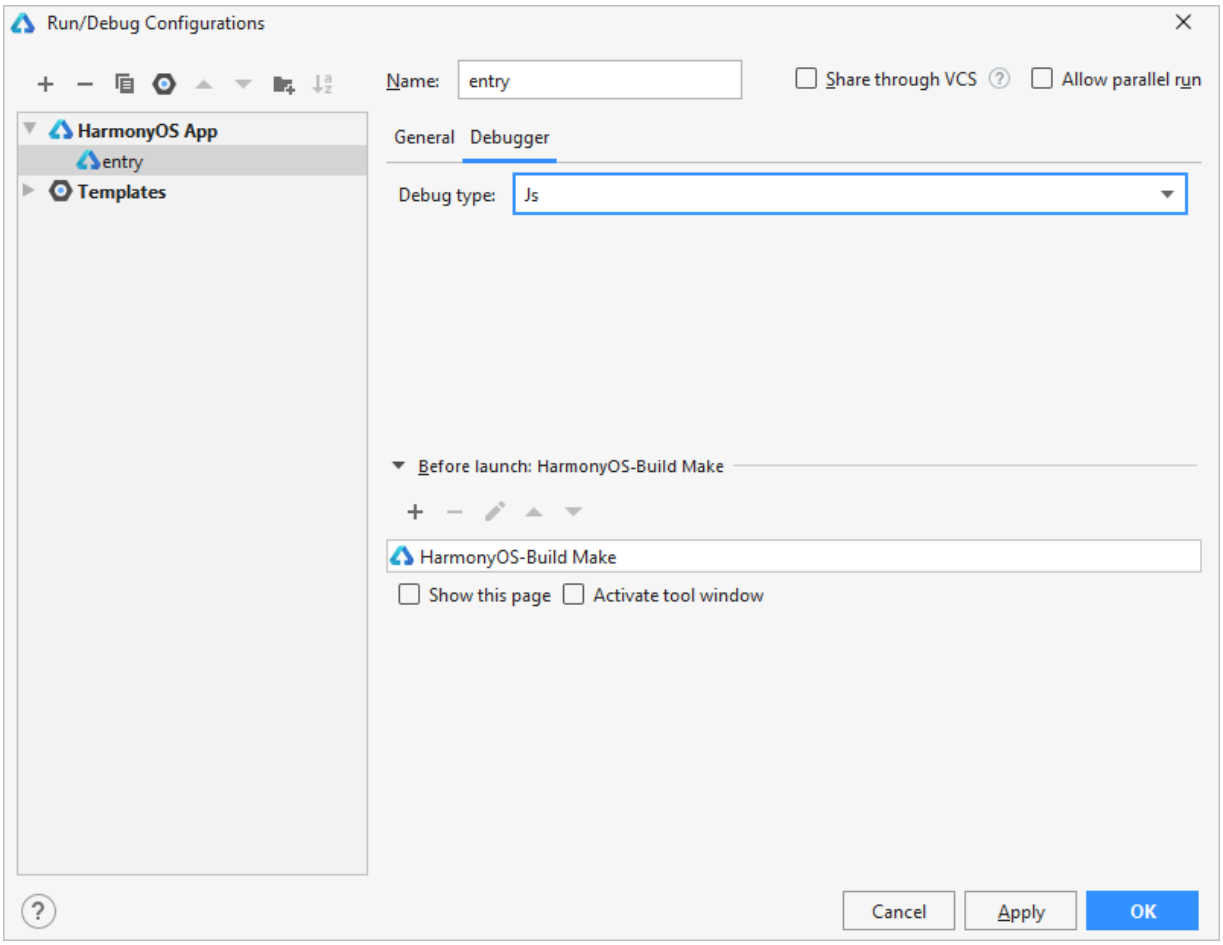
可能原因为网络问题导致数据获取缓慢，不能显示真机图像，可以尝试滑动屏幕或者更换设备后重试。

模拟器屏幕显示“Network exception, please release and try again”，如何解决？

网络不稳定导致数据流中断，请释放模拟器资源后重新申请模拟器继续使用。

使用模拟器可以正常进入 Java 断点，但是不能进入 JS 断点，如何解决？

点击 **Run > Edit Configurations > Debugger**，将 **Debug type** 设置为 Js 即可。



功能开发

如何查看“config.json”文件的所有字段说明？
“config.json”文件的各字段说明请查阅配置文件的元素。

怎么实现 Ability 可以被其他应用调用？
开发者需要在“config.json”文件中将“abilities”字段的“visible”标签设置为 true。

权限声明在哪里进行添加？
开发者需要在“config.json”文件中的“reqPermissions”字段中声明所需要的权限，具体配置方法请参考申请权限。

使用数据库注解相关功能前有什么注意事项？

使用注解功能需在模块的“build.gradle”文件的“ohos”节点中增加如下配置项（不使用注解功能无须配置）：

```
compileOptions{
    annotationEnabled true
}
```

使用<image>标签引入本地图片，但图片无法加载？
图片无法加载的可能情况有三种：

没有给图片设置宽度和高度，需要在对应的 page 目录下的 css 样式文件中设置图片的宽高。使用<image>标签的图片不会自动缩放，图片宽高超过组件的宽高会自动截取。

图片引入路径错误。图片引入的路径必须是项目编译后的静态文件的路径。

在导入图片或添加/删除页面后没有重新编译。需要重新编译刷新 target 文件中的代码。

如何在后一个页面获取前一个页面传递过来的参数？
有三种方式可以获取前一个页面的参数。以如下场景为例：有两个页面“index”和“detail”，第二个页面“detail”需要获取从第一个页面“index”传递过来的参数。

如果参数需要在页面中引用，可以直接在“detail.hml”中使用`{{参数名}}`的形式进行引用。

如果需要对参数进行操作，在“detail.js”中，直接用 this.参数名的形式使用。

可以在“detail.js”的 data 域中定义一个同名参数进行接收，注意以这种方式接受的参数将覆盖已有的参数。

如何查询设备支持的硬件/软件功能？如何查询设备是否支持某个硬件/软件特性？

应用通过调用 IBundleManager 接口类中的 getSystemAvailableCapabilities 方法，可以查询设备支持的硬件/软件功能列表。具体的功能定义可以通过 ohos.utils.CapabilityConstants 类查询。

应用通过调用 IBundleManager 接口类中的 hasSystemCapability 方法，可以查询设备是否支持某个硬件/软件功能。具体的功能定义可以通过 ohos.utils.CapabilityConstants 类查询。

图片为什么显示不全？
父类容器大小不能小于子组件容器大小。

调测验证

编译工程提示“This device type does not match project profile.”或安装时出现“DEVICE_NOT_SUPPORT_ERROR”，如何解决？

出现这种情况是由于“config.json”中配置的设备类型与调试设备类型不匹配，需要在“module”标签下配置对“deviceType”的定义。具体请参考表 7 的“deviceType”。

安装 HAP 失败，并提示“INCONSISTENT_BUNDLE_VERSION”，如何解决？
系统中有重复应用，卸载系统中已有的包名相同的应用。

提示“signingConfig 'debug' can not be null or empty”，如何解决？
检查“entry”下的 build.gradle 是否配置了签名。如果配了依然报错，检查是否误配到了工程级的 build.gradle 当中。

安装 HAP 失败，并提示“STRING_LENGTH_ERROR”，如何解决？

可能原因有：

- 包信息超过最大长度。包信息中包含的各属性字符串长度需要同时满足以下条件，否则会报错。
- bundleName 的长度为 7~127 个字节。
- vendor 的长度为 0~255 个字节。
- version.name 的长度为 0~127 个字节。

- 同时安装两个不同 module 生成的 hap 时，包信息不一致。需要比对两个 module 的“config.json”文件中“app”标签配置内容是否一致。

51CTO 鸿蒙技术社区 西米晗 整理
https://harmonyos.51cto.com/

欢迎关注 HarmonyOS 技术社区公众号