第 3 章 HarmonyOS 应用开发基础与原理

本章属于 HarmonyOS 应用开发的基础知识与原理学习，无论是传统的需要安装的 App 开发，还是原子化服务与服务卡片开发，这些基本概念、基本设置、配置、基本知识与基本技能，开发者需要了解和掌握。同时本章将对原子化服务与服务卡片开发的总体与基础知识部分进行详细的阐述。

3.1 HarmonyOS 应用开发综述

本节主要阐述 HarmonyOS 应用开发的基本概念，包括应用配置文件、资源文件、应用数据管理、应用安全管理、应用隐私保护与第三方应用调用管控机制等基本概念、各部分的详细构成与开发相关的基本知识。

3.1.1 综述与基本概念

1. 综述

关于 HarmonyOS 应用的明确定义与分类，我们在第 1 章中已进行了阐述。在 HarmonyOS 开发者门户 2021 年 8 月 13 日更新的文档材料中，把 HarmonyOS 应用分为传统方式的需要安装的应用和提供特定功能但免安装的应用，即原子化服务。

HarmonyOS 的用户应用程序包以 App Pack(Application Package)形式发布，它是由一个或多个 HAP(HarmonyOS Ability Package)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。

其中原子化服务的 HAP 包需要满足免安装的要求，而传统应用形式无须满足免安装的要求。

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 3-1 所示。

entry 是应用的主模块。在一个 App 中，对于同一设备类型必须有且只有一个 entry 类型的 HAP，可独立安装并运行。

feature 是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 才能独立运行。

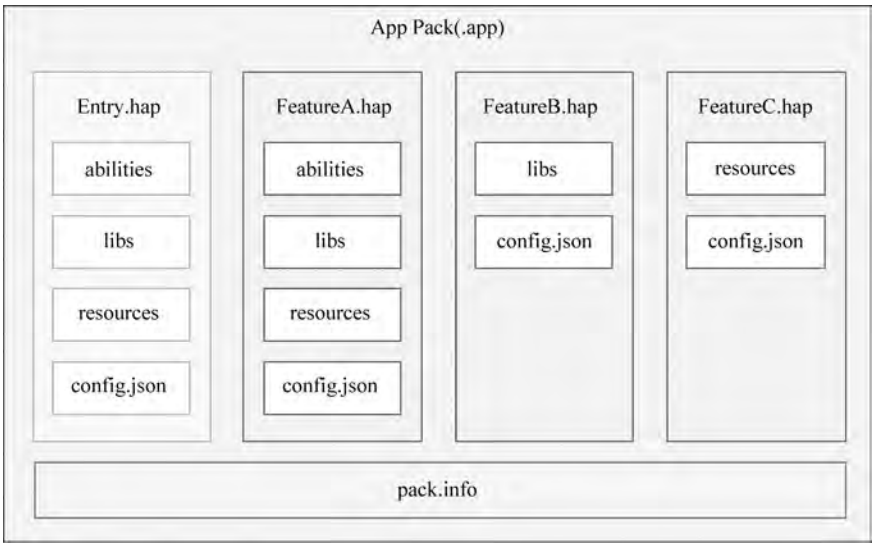

图 3-1　HarmonyOS App 逻辑视图

2. 基本概念

以下先对 HarmonyOS 应用开发涉及的一些基本概念进行汇总阐述，这些概念在后续的学习中会进一步详细阐述与实际使用，具体包括 Ability、库文件、资源文件、配置文件、pack. info、HAP、HAR 等。

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，一种是 FA(Feature Ability)，另一种是 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。

库文件是应用依赖的第三方代码，例如 so、jar、bin、har 等二进制文件，存放在 libs 目录。

应用的资源文件如字符串、图片、声频等存放于 resources 目录下，便于开发者使用和维护。

配置文件是应用的 Ability 信息，用于声明应用的 Ability，以及应用所需权限等信息，文件为 config. json。

pack. info 用于描述应用软件包中每个 HAP 的属性，由 DevEco Studio 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(HarmonyOS Ability Package)的 3 个属性：一是 delivery-with-install，表示该 HAP 是否支持随应用安装，true 表示支持随应用安装，false 表示不支持随应用安装。二是 Name，表示 HAP 文件名。三是 module-type，表示模块类型，entry 或 feature。四是 device-type，表示支持该 HAP 运行的设备类型。

HAR(HarmonyOS Ability Resources)可以提供构建应用所需的所有内容，包括源代

码、资源文件和 config.json 文件。HAR 不同于 HAP，HAR 不能独立安装并运行在设备上，只能作为应用模块的依赖项被引用。

3.1.2 应用配置文件

1. 简介

应用的每个 HAP 的根目录下都存放着一个 config.json 配置文件，该文件主要涵盖 3 方面的内容。一是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。二是应用在具体设备上的配置信息，包含应用的备份恢复、网络安全等能力。三是 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。

配置文件 config.json 采用 JSON 文件格式，其中包含了一系列配置项，每个配置项由属性和值两部分构成。属性的出现顺序不分先后，并且每个属性最多只允许出现一次。每个属性的值为 JSON 的基本数据类型，包括数值、字符串、布尔值、数组、对象或者 null 类型。属性值可以引用相应的资源文件。

2. 配置文件的元素

DevEco Studio 提供了两种编辑配置文件 config.json 的方式。在 config.json 的编辑窗口中，可在右上角切换代码编辑视图或可视化编辑视图，如图 3-2 所示。

图 3-2 config.json 文件的可视化编辑视图

config.json 配置文件的内部结构由 app、deviceConfig 和 module 共 3 部分组成，缺一不可。config.json 配置文件的每个组成部分及下级组成部分称为属性，后续的内容，我们将根据属性名称、属性含义、数据类型、是否可缺省、是否有子属性、子属性的详细情况进行阐述。

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。deviceConfig 表示应用在具体设备上的配置信息。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。这 3 部分的数据类型都为对象，都不可以缺省。下面分别阐述这 3 部分的详细内容。

1) app 对象的内部结构

app 对象包含应用的全局配置信息，相关属性说明如下。

(1) 属性 bundleName，无子属性。表示应用的包名，用于标识应用的唯一性。包名是由字母、数字、下画线(_)和点号(.)组成的字符串，必须以字母开头。支持的字符串长度为 7～127 字节。包名通常采用反域名形式表示；例如，cn.jltfcloud.daodejing。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。数量类型为字符串型。不能缺省。

(2) 属性 vendor，表示对应用开发厂商的描述。字符串长度不超过 255 字节。数据类型为字符串。可缺省，缺省值为空。

(3) 属性 version，无子属性，表示应用的版本信息。数据类型为对象型。不能缺省。其子属性包括 name、code、minCompatibleVersionCode。三者分别说明如下。

Name，表示应用的版本号，用于向应用的终用户端呈现。取值可以自定义，长度不超过 127 字节。Name 数据类型为字符串型，不可缺省。自定义规则如下：

API 5 及更早版本，推荐使用三段式数字版本号，也兼容两段式版本号，如 A.B.C 也兼容 A.B，其中 A、B、C 取值为 0～999 的整数。除此之外不支持其他格式。A 段，一般表示主版本号(Major)。B 段，一般表示次版本号(Minor)。C 段，一般表示修订版本号(Patch)。

从 API 6 版本起，推荐采用四段式数字版本号，如 A.B.C.D，其中 A、B、C 取值为 0～99 的整数，D 取值为 0～999 的整数。A 段，一般表示主版本号(Major)。B 段，一般表示次版本号(Minor)。C 段，一般表示特性版本号(Feature)。D 段，一般表示修订版本号(Patch)。

Code 表示应用的版本号，仅用于 HarmonyOS 管理该应用，不对应用的终用户端呈现。Code 的数据类型为数值型，不可缺省。取值规则如下。

API 5 及更早版本：二进制 32 位以内的非负整数，需要从 version.name 的值转换得到。转换规则为 code 值＝A×1,000,000＋B×1,000＋C。例如，version.name 字段取值为 2.2.1，则 code 的值为 2002001。

从 API 6 版本起：code 的取值不与 version.name 字段的取值关联，开发者可自定义 code 的取值，取值范围为小于 2^{31} 的非负整数，但是每次应用的版本更新，均需更新 code 字

段的值，新版本 code 取值必须大于旧版本 code 的值。

minCompatibleVersionCode 表示应用可兼容的最低版本号，用于在跨设备场景下，判断其他设备上该应用的版本是否兼容。格式与 version.code 字段的格式要求相同。数据类型为数值型，可缺省，缺省值为 code 标签值。

(4) 属性 multiFrameworkBundle，无子属性，表示应用是否为混合打包的 HarmonyOS 应用。混合打包场景应配置为 true，非混合打包场景应配置为 false。该标签值由 IDE 自动配置。数据类型为布尔型。可缺省，缺省值为 false。

(5) 属性 smartWindowsize，无子属性，该标签用于在悬浮窗场景下表示应用的模拟窗口的尺寸。配置格式为“正整数 * 正整数”，单位为 vp。正整数取值范围为[200，2000]。数据类型为字符串型。可缺省，缺省值为空。

(6) 属性 smartWindowDeviceType，无子属性，表示应用可以在哪些设备上使用模拟窗口打开。取值方式为智能手机(phone)；平板(tablet)；智慧屏(tv)。数据类型为字符串数组型。可缺省，缺省值为空。

(7) 属性 targetBundleList，无子属性，表示允许以免安装方式拉起的其他 HarmonyOS 应用，列表取值为每个 HarmonyOS 应用的 bundleName，多个 bundleName 之间用英文“，”区分，最多配置 10 个 bundleName。如果被拉起的应用不支持免安装方式，则拉起失败。数据类型为字符串型。可缺省，缺省值为空。

(8) app 对象内部结构示例，代码如下：

```
"//":"第 3 章/app 对象内部结构示例"
"app": {
   "bundleName": "cn.jltfcloud.daodejing.example",
   "vendor": "jltfcloud",
   "version": {
       "code": 2,
       "name": "2.0"
   },
   "apiVersion": {
       "compatible": 3,
       "target": 3,
       "releaseType": "Beta1"
   }
}
```

2) 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、phone、tablet、tv、car、wearable、liteWearable 和 smartVision 等属性。default 标签内的配置适用于所有设备，如果其他设备类型有特殊的需求，则需要在该设备类型的标签下进行配置。其内部结构说明如下。

属性 phone、tablet、tv、car、wearable、liteWearable、smartVision 分别表示手机、平板、智

慧屏、车机、智能穿戴、轻量级智能穿戴、智能摄像头特有的应用配置信息。以上属性的数据类型都为对象，除属性 default 不可缺省外，其他属性都可缺省，缺省值为空。

default、phone、tablet、tv、car、wearable、liteWearable 和 smartVision 等对象不同设备的内部结构说明如下。

(1) 属性 jointUserId 表示应用的共享 userid。通常情况下，不同的应用运行在不同的进程中，应用的资源无法共享。如果开发者的多个应用之间需要共享资源，则可以通过相同的 jointUserId 值实现，前提是这些应用的签名相同。

该标签仅对系统应用生效，并且仅适用于手机、平板、智慧屏、车机、智能穿戴。该字段在 API Version 3 及更高版本不再支持配置。数据类型为字符串型。可缺省，缺省值为空。

(2) 属性 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 运行在这个进程中。该标签仅适用于手机、平板、智慧屏、车机、智能穿戴。数据类型为字符串型。可缺省，缺省值为应用的软件包名。

(3) 属性 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。该标签仅适用于手机、平板、智慧屏、车机、智能穿戴。数据类型为布尔型。可缺省，缺省值为 false。

(4) 属性 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 包在安装时无须解压 libs，运行时会直接从 HAP 内加载 libs 库。该标签仅适用于手机、平板、智慧屏、车机、智能穿戴。数据类型为布尔型。可缺省，缺省值为 true。

(5) 属性 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网络安全，无须修改应用代码。数据类型为对象型。可缺省，缺省值为空。network 对象的内部结构进一步说明如下。

(6) 属性 cleartextTraffic 表示是否允许应用使用明文网络流量，例如，明文 HTTP。true 表示允许应用使用明文流量的请求。false 表示拒绝应用使用明文流量的请求。数据类型为布尔型。可缺省，缺省值为 false。

(7) 属性 securityConfig 表示应用的网络安全配置信息。数据类型为对象型。可缺省，缺省值为空。securityConfig 对象的内部结构进一步说明如下。

(8) 属性 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。数据类型为对象型。可缺省，缺省值为空。

子属性 cleartextPermitted 表示自定义的网域范围内是否允许明文流量传输。当 cleartextTraffic 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 表示允许明文流量传输。false 表示拒绝明文流量传输。数据类型为布尔型。不可缺省。

子属性 domains 表示域名配置信息，包含两个参数 subdomains 和 name。subdomains

(布尔型)：表示是否包含子域名。如果值为 true，则此网域规则将与相应网域及所有子网域(包括子网域的子网域)匹配。否则，该规则仅适用于精确匹配项。name(字符串型)表示域名名称。数据类型为对象数组型，不可缺省。

(9) deviceConfig 对象的内部结构示例，代码如下：

```
"//":"第 3 章/deviceConfig 对象内部结构示例"
 "deviceConfig": {
   "default": {
      "process": "cn.jltfcloud.daodejing.example",
       "supportBackup": false,
       "network": {
           "cleartextTraffic": true,
           "securityConfig": {
               "domainSettings": {
                  "cleartextPermitted": true,
                  "domains": [
                      {
                         "subdomains": true,
                         "name": "example.ohos.com"
                      }
                  ]
               }
           }
       }
   }
}
```

3) module 对象的内部结构

module 对象包含 HAP 包的配置信息，内部结构说明如下：

(1) 属性 mainAbility 表示 HAP 包的入口 ability 名称。该标签的值应配置为 module > abilities 中存在的 Page 类型 ability 的名称。该标签仅适用于手机、平板、智慧屏、车机、智能穿戴。数据类型为字符串型。如果存在 page 类型的 ability，则该字段不可缺省。

(2) 属性 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。该标签仅适用于手机、平板、智慧屏、车机、智能穿戴。数据类型为字符串型，不可缺省。

(3) 属性 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。该标签仅适用于手机、平板、智慧屏、车机、智能穿戴。数据类型为字符串型。不可缺省。

(4) 属性 description 表示 HAP 的描述信息。字符串长度不超过 255 字节。如果字符串超出最大长度或者需要支持多语言，则可以采用资源索引的方式添加描述内容。该标签

仅适用于手机、平板、智慧屏、车机、智能穿戴。数据类型为字符串型。可缺省，缺省值为空。

(5) 属性 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。数据类型为字符串数组型。可缺省，缺省值为空。

(6) 属性 deviceType 表示允许 Ability 运行的设备类型。系统预定义的设备类型包括 phone(手机)、tablet(平板)、tv(智慧屏)、car(车机)、wearable(智能穿戴)、liteWearable(轻量级智能穿戴)等。数据类型为字符串数组型。不可缺省。

(7) 属性 distro 表示 HAP 发布的具体描述。该标签仅适用于手机、平板、智慧屏、车机、智能穿戴。数据类型为对象型。不可缺省。

(8) 属性 metaData 表示 HAP 的元信息。数据类型为对象型。可缺省，缺省值为空。

(9) 属性 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。数据类型为对象数组型。可缺省，缺省值为空。

(10) 属性 js，表示基于 JS UI 框架开发的 JS 模块集合，其中每个元素代表一个 JS 模块信息。数据类型为对象数组型。可缺省，缺省值为空。

(11) 属性 shortcuts 表示应用的快捷方式信息。采用对象数组格式，其中每个元素表示一个快捷方式对象。数据类型对象数组。可缺省，缺省值为空。

(12) 属性 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。属性类型为对象数组。可缺省，缺省值为空。

(13) 属性 reqPermissions 表示应用运行时向系统申请的权限。数据类型为对象数组型。可缺省，缺省值为空。

(14) 属性 colorMode 表示应用自身的颜色模式。其中，dark 表示按照深色模式选取资源。light 表示按照浅色模式选取资源。auto 表示跟随系统的颜色模式值选取资源。该标签仅适用于手机、平板、智慧屏、车机、智能穿戴。数据类型为字符串型。可缺省，缺省值为 auto。

(15) 属性 resizeable 表示应用是否支持多窗口特性。该标签仅适用于手机、平板、智慧屏、车机、智能穿戴。数据类型为布尔类型。可缺省，缺省值为 true。

(16) module 对象内部结构示例，代码如下：

```
"//":"第 3 章/module 对象内部结构示例"
"module": {
    "mainAbility": "MainAbility",
    "package": "com.example.myapplication.entry",
    "name": ".MyOHOSAbilityPackage",
    "description": "$string:description_application",
    "supportedModes": [
        "drive"
    ],
    "deviceType": [
        "car"
    ],
```

```
    "distro": {
        "deliveryWithInstall": true,
        "moduleName": "ohos_entry",
        "moduleType": "entry"
    },
    "abilities": [
        ...
    ],
    "shortcuts": [
        ...
    ],
    "js": [
        ...
    ],
    "reqPermissions": [
        ...
    ],
    "defPermissions": [
        ...
    ],
    "colorMode": "light"
}
```

（17）distro 对象的内部结构说明如下。

① 属性 deliveryWithInstall 表示当前 HAP 是否支持随应用安装。true 表示支持随应用安装。false 表示不支持随应用安装。说明，建议将该属性设置为 true。设置为 false 可能导致最终应用上架应用市场时出现异常。数据类型为布尔型。不可缺省。

② 属性 moduleName 表示当前 HAP 的名称。数据类型为字符串型。不可缺省。

③ 属性 moduleType 表示当前 HAP 的类型，包括两种类型，即 entry 和 feature。数据类型为字符串型。不可缺省。

④ 属性 installationFree 表示当前该 FA 是否支持免安装特性。true 表示支持免安装特性，并且符合免安装约束。false 表示不支持免安装特性。数据类型为布尔型。

（18）distro 对象内部结构示例，代码如下：

```
"//":"第 3 章/distro 对象内部结构示例"
"distro": {
    "deliveryWithInstall": true,
    "moduleName": "ohos_entry",
    "moduleType": "entry",
    "installationFree": true
}
```

（19）metaData 对象的内部结构说明。

① 属性 parameters 表示调用 Ability 时所有调用参数的元信息。每个调用参数的元信息由 3 个标签组成，即 description、name、type。数据类型为对象型。可缺省，缺省值为空。

标签 description 表示对调用参数的描述，可以是表示描述内容的字符串，也可以是对描述内容的资源索引，以便支持多语言。数据类型为字符串型。可缺省，缺省值为空。

标签 name 表示调用参数的名称。数据类型为字符串型。可缺省，缺省值为空。

标签 type 表示调用参数的类型，如 Integer。数据类型为字符串。不可缺省。

② 属性 results 表示 Ability 返回值的元信息。每个返回值的元信息由 3 个标签组成，即 description、name、type。数据类型为对象型。可缺省，缺省值为空。

description 表示对返回值的描述，可以是表示描述内容的字符串，也可以是对描述内容的资源索引，以便支持多语言。数据类型为字符串型。可缺省，缺省值为空。

name 表示返回值的名字。数据类型为字符串型。可缺省，缺省值为空。

type 表示返回值的类型，如 Integer。数据类型为字符串型。不可缺省。

③ 属性 customizeData，表示父级组件的自定义元信息，parameters 和 results 在 module 中不可配。数据类型为对象型。可缺省，缺省值为空。

其子属性包括 3 个。name 表示数据项的键名称，字符串类型（最大长度为 255 字节）。数据类型为字符串型，可缺省，缺省值为空。value 表示数据项的值，字符串类型（最大长度为 255 字节）。数据类型为字符串型。可缺省，缺省值为空。extra 表示用户自定义数据格式，标签值为标识该数据的资源的索引值。数据类型为字符串型。可缺省，缺省值为空。

（20）metaData 对象内部结构示例，代码如下：

```
"//":"第 3 章/metaData 对象内部结构示例"
"metaData": {
    "parameters" : [{
        "name" : "string",
        "type" : "Float",
        "description" : " $ string:parameters_description"
    }],
    "results" : [{
        "name" : "string",
        "type" : "Float",
        "description" : " $ string:results_description"
    }],
    "customizeData" : [{
        "name" : "string",
        "value" : "string",
        "extra" : " $ string:customizeData_description"
    }]
}
```

(21) abilities 对象的内部结构说明。

属性 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。该标签仅适用于手机、平板、智慧屏、车机、智能穿戴。在使用 DevEco Studio 新建项目时,默认生成首个 Ability 的配置,包括生成 MainAbility. java 文件,以及 config. json 文件中 MainAbility 的配置。如使用其他 IDE 工具,则可自定义名称。数据类型为字符串型。不可缺省。

属性 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,以便支持多语言。数据类型为字符串型。可缺省,缺省值为空。

属性 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。数据类型为字符串型。可缺省,缺省值为空。

属性 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,以便支持多语言。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。应用的 icon 和 label 是用户可感知的配置项,需要区别于当前所有已有的应用 icon 或 label(至少有一个不同)。数据类型为字符串型。可缺省,缺省值为空。

属性 uri 表示 Ability 的统一资源标识符。

格式为[scheme:][//authority][path][? query][# fragment]。数据类型为字符串型。可缺省,对于 data 类型的 Ability 不可缺省。

属性 launchType 表示 Ability 的启动模式,支持 standard、singleMission 和 singleton 3 种模式。standard 表示该 Ability 可以有多个实例。standard 模式适用于大多数应用场景。singleMission 表示此 Ability 在每个任务栈中只能有一个实例。singleton 表示该 Ability 在所有任务栈中仅可以有一个实例。例如,具有全局唯一性的呼叫来电界面可采用 singleton 模式。该标签仅适用于手机、平板、智慧屏、车机、智能穿戴。数据类型为字符串型。可缺省,缺省值为 standard。

属性 visible 表示 Ability 是否可以被其他应用调用。true 表示可以被其他应用调用。false 表示不能被其他应用调用。数据类型为布尔型。可缺省,缺省值为 false。

属性 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,则取值必须与 defPermissions 标签中定义的某个权限的 name 标签值一致。数据类型为字符串数组型。可缺省,缺省值为空。

属性 skills 表示 Ability 能够接收的 Intent 的特征。数据类型为对象数组型。可缺省,

缺省值为空。

属性 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。数据类型为字符串数组型。可缺省，缺省值为空。

属性 metaData 表示 Ability 的元信息。调用 Ability 时调用参数的元信息，例如参数个数和类型。Ability 执行完毕后返回值的元信息，例如：返回值的个数和类型。该标签仅适用于智慧屏、智能穿戴、车机。数据类型为对象型。可缺省，缺省值为空。

属性 type 表示 Ability 的类型。取值范围：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；CA 表示支持其他应用以窗口方式调起该 Ability。属性类型为字符串。不可缺省。

属性 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：unspecified 表示由系统自动判断显示方向。landscape 表示横屏模式。portrait 表示竖屏模式。followRecent 表示跟随栈中最近的应用。数据类型为字符串，可缺省，缺省值为 unspecified。

属性 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：dataTransfer 表示通过网络/对端设备进行数据下载、备份、分享、传输等业务；audioPlayback 表示声频输出业务；audioRecording 表示声频输入业务；pictureInPicture 表示画中画、小窗口播放视频业务；voip 表示音视频电话、VOIP 业务；location 表示定位、导航业务；bluetoothInteraction 表示蓝牙扫描、连接、传输业务；WiFiInteractionWLAN 表示扫描、连接、传输业务；screenFetch 表示录屏、截屏业务；multiDeviceConnection 表示多设备互联业务。数据类型为字符串数组型。可缺省，缺省值为空。

属性 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。该标签仅适用于手机、平板、智慧屏、车机、智能穿戴。数据类型为字符串型。可缺省，缺省值为空。

属性 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。该标签仅适用于手机、平板、智慧屏、车机、智能穿戴。数据类型为字符串型。可缺省，缺省值为空。

属性 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；数据类型为字符串数组型，可缺省，缺省值为空。

属性 mission 表示 Ability 指定的任务栈。该标签仅适用于 page 类型的 Ability。默认情况下应用中所有 Ability 同属一个任务栈。该标签仅适用于手机、平板、智慧屏、车机、智能穿戴。数据类型为字符串型。可缺省，缺省值为应用的包名。

属性 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（别名 Ability）的属性中仅 name、icon、label、visible、permissions、skills 生效，其他属性均沿用 targetAbility 中的属性值。目标 Ability 必须与别名 Ability 在同一应用中，并且在配置文件中目标 Ability 必须在别名之前进行声明。该标签仅适用于手机、平板、智慧屏、车机、智能穿戴。数据类型为字符串型。可缺省，缺省值为空。表示当前 Ability 不是一个别名 Ability。

属性 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。当配置为 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。该标签仅适用于手机、平板、智慧屏、车机、智能穿戴。数据类型为布尔类型。可缺省，缺省值为 false。

属性 supportPipMode 表示 Ability 是否支持用户进入 PIP 模式（用于在页面的最上层悬浮小窗口，俗称“画中画”，常见于视频播放等场景）。该标签仅适用于 page 类型的 Ability。该标签仅适用于手机、平板、智慧屏、车机、智能穿戴。数据类型为布尔类型。可缺省，缺省值为 false。

属性 formsEnabled 表示 Ability 是否支持卡片（forms）功能。该标签仅适用于 page 类型的 Ability。true 表示支持卡片能力。false 表示不支持卡片能力。数据类型为布尔类型。可缺省，缺省值为 false。

属性 forms 表示服务卡片的属性。该标签仅当 formsEnabled 为 true 时才能生效。数据类型为对象数组型。可缺省，缺省值为空。

属性 resizeable 表示 Ability 是否支持多窗口特性。该标签仅适用于手机、平板、智慧屏、车机、智能穿戴。数据类型为布尔型。可缺省，缺省值为 true。

（22）abilities 对象内部结构示例，代码如下：

```
"//":"第 3 章/abilities 对象内部结构示例"
"abilities": [
    {
       "name": ".MainAbility",
       "description": "himusic main ability",
       "icon": "$media:ic_launcher",
       "label": "HiMusic",
       "launchType": "standard",
       "orientation": "unspecified",
       "permissions": [
       ],
       "visible": true,
       "skills": [
           {
               "actions": [
                   "action.system.home"
```

```
                ],
                "entities": [
                    "entity.system.home"
                ]
            }
        ],
        "configChanges": [
            "locale",
            "layout",
            "fontSize",
            "orientation"
        ],
        "type": "page"
    },
    {
        "name": ".PlayService",
        "description": "himusic play ability",
        "icon": "$media:ic_launcher",
        "label": "HiMusic",
        "launchType": "standard",
        "orientation": "unspecified",
        "visible": false,
        "skills": [
            {
                "actions": [
                    "action.play.music",
                    "action.stop.music"
                ],
                "entities": [
                    "entity.audio"
                ]
            }
        ],
        "type": "service",
        "backgroundModes": [
            "audioPlayback"
        ]
    },
    {
        "name": ".UserADataAbility",
        "type": "data",
        "uri":
"dataability://com.huawei.hiworld.himusic.UserADataAbility",
        "visible": true
    }
]
```

(23) skills 对象的内部结构说明。

属性 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。数据类型为字符串数组型。可缺省,缺省值为空。

属性 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。数据类型为字符串数组型。可缺省,缺省值为空。

属性 uris 表示能够接收的 Intent 的 uri,可以包含一个或者多个 uri。数据类型为对象数组型。可缺省,缺省值为空。其子属性的数据类型都为字符串型。除了 scheme 不可缺省外,其他都可以缺省,缺省值为空。scheme 表示 uri 的 scheme 值。host 表示 uri 的 host 值。port 表示 uri 的 port 值。path 表示 uri 的 path 值。type 表示 uri 的 type 值。

(24) skills 对象内部结构示例,代码如下:

```
"//":"第 3 章/skills 对象内部结构示例"
"skills": [
    {
       "actions": [
          "action.system.home"
       ],
         "entities": [
          "entity.system.home"
       ],
       "uris": [
          {
               "scheme": "http",
               "host": "www.xxx.com",
               "port": "8080",
               "path": "query/student/name",
               "type": "text/*"
          }
       ]
    }
]
```

(25) JS 对象的内部结构说明。

属性 name 表示 JS Component 的名字。该标签不可缺省,默认值为 default。数据类型为字符串型。

属性 Pages 表示 JS Component 的页面用于列举 JS Component 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,数组的第 1 个元素代表 JS FA 首页。数据类型为数组型,不可缺省。

属性 window 用于定义与显示窗口相关的配置。该标签仅适用于手机、平板、智慧屏、车机、智能穿戴。数据类型为对象型。可缺省。

其子属性 designWidth 表示页面设计的基准宽度。以此为基准,根据实际设备宽度来

缩放元素大小。数据类型为数值型。可缺省，缺省值为750px。

其子属性autoDesignWidth表示页面设计的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。数据类型为布尔型。可缺省，缺省值为false。

属性type表示JS应用的类型。取值范围：normal用于将该JS Component标识为应用实例；form用于将该JS Component标识为卡片实例。数据类型为字符串型。可缺省，缺省值为normal。

(26) JS对象内部结构示例，代码如下：

```
"//":"第3章/JS对象内部结构示例"
"js": [
    {
       "name": "default",
       "pages": [
          "pages/index/index",
          "pages/detail/detail"
       ],
       "window": {
          "designWidth": 750,
          "autoDesignWidth": false
       },
       "type": "form"
    }
]
```

(27) shortcuts对象的内部结构说明。

属性shortcutId表示快捷方式的ID。字符串的最大长度为63字节。数据类型为字符串型。不可缺省。

属性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。字符串的最大长度为63字节。数据类型为字符串型。可缺省，缺省值为空。

属性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，即targetClass和targetBundle。数据类型为对象数组型。可缺省，缺省值为空。targetClass表示快捷方式目标类名。数据类型为字符串型。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。数据类型为字符串型。可缺省，缺省值为空。

(28) shortcuts对象内部结构示例，代码如下：

```
"//":"第3章/shortcuts对象内部结构示例"
"shortcuts": [
    {
```

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

(29) forms 对象的内部结构说明。

属性 name 表示卡片的类名。字符串的最大长度为 127 字节。数据类型为字符串型。不可缺省。

属性 description 表示卡片的描述。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。字符串的最大长度为 255 字节。数据类型为字符串型。可缺省,缺省值为空。

属性 isDefault 表示该卡片是否为默认卡片,每个 Ability 有且只有一个默认卡片。true 表示默认卡片。false 表示非默认卡片。数据类型为布尔型。不可缺省。

属性 type 表示卡片的类型。取值范围:Java 表示 Java 卡片;JS 表示 JS 卡片。数据类型为字符串型。不可缺省。

属性 colorMode 表示卡片的主题样式,取值范围:auto 表示自适应;dark 表示深色主题;light 表示浅色主题。数据类型为字符串型。可缺省,缺省值为 auto。

属性 supportDimensions 表示卡片支持的外观规格,取值范围:1 * 2:表示 1 行 2 列的二宫格;2 * 2:表示 2 行 2 列的四宫格;2 * 4:表示 2 行 4 列的八宫格;4 * 4:表示 4 行 4 列的十六宫格。数据类型为字符串数组型。不可缺省。

属性 defaultDimension 表示卡片的默认外观规格,取值必须在该卡片 supportDimensions 配置的列表中。数据类型为字符串型。不可缺省。

属性 landscapeLayouts 表示卡片外观规格对应的横向布局文件,与 supportDimensions 中的规格一一对应。仅当卡片类型为 Java 卡片时,需要配置该标签。数据类型为字符串数组型。不可缺省。

属性 portraitLayouts 表示卡片外观规格对应的竖向布局文件,与 supportDimensions 中的规格一一对应。仅当卡片类型为 Java 卡片时,需要配置该标签。数据类型为字符串数组型。不可缺省。

属性 updateEnabled 表示卡片是否支持周期性刷新,取值范围:true 表示支持周期性刷新,可以在定时刷新(updateDuration)和定点刷新(scheduledUpdateTime)两种方式中任选其一,优先选择定时刷新;false 表示不支持周期性刷新。数据类型为布尔型。不可缺省。

属性 scheduledUpdateTime 表示卡片的定点刷新的时刻，采用 24h 制，精确到分钟。数据类型为字符串型。可缺省，缺省值为“0:0”。

属性 updateDuration 表示卡片定时刷新的更新周期，单位为 30min，取值为自然数。

当取值为 0 时，表示该参数不生效。当取值为正整数 N 时，表示刷新周期为 $30\times N$ 分钟。数据类型为数值型。可缺省，缺省值为“0”。

属性 formConfigAbility 表示卡片的配置跳转链接，采用 URI 格式。数据类型为字符串型。可缺省，缺省值为空。

属性 jsComponentName 表示 JS 卡片的 Component 名称。字符串的最大长度为 127 字节。仅当卡片类型为 JS 卡片时，需要配置该标签。数据类型为字符串型。不可缺省。

属性 metaData 表示卡片的自定义信息，包含 customizeData 数组标签。数据类型为对象型。可缺省，缺省值为空。

属性 customizeData 表示自定义的卡片信息。数据类型为对象数组型。可缺省，缺省值为空。其子属性 name 表示数据项的键名称。字符串的最大长度为 255 字节。数据类型为字符串型。可缺省，缺省值为空。其子属性 value 表示数据项的值。字符串的最大长度为 255 字节。数据类型为字符串型。可缺省，缺省值为空。

(30) forms 对象内部结构示例，代码如下：

```
"//":"第 3 章/forms 对象内部结构示例"
"forms": [
    {
        "name": "Form_Js",
        "description": "It's Js Form",
        "type": "JS",
        "jsComponentName": "card",
        "colorMode": "auto",
        "isDefault": true,
        "updateEnabled": true,
        "scheduledUpdateTime": "11:00",
        "updateDuration": 1,
        "defaultDimension": "2 * 2",
        "supportDimensions": [
            "2 * 2",
            "2 * 4",
            "4 * 4"
        ]
    },
    {
        "name": "Form_Java",
        "description": "It's Java Form",
        "type": "Java",
```

```
            "colorMode": "auto",
            "isDefault": false,
            "updateEnabled": true,
            "scheduledUpdateTime": "21:05",
            "updateDuration": 1,
            "defaultDimension": "1 * 2",
            "supportDimensions": [
                "1 * 2"
            ],
            "landscapeLayouts": [
                " $ layout:ability_form"
            ],
            "portraitLayouts": [
               " $ layout:ability_form"
            ],
            "formConfigAbility": "ability://com.example.myapplication.fa/.MainAbility",
            "metaData": {
                "customizeData": [
                   {
                        "name": "originWidgetName",
                        "value": "com.huawei.weather.testWidget"
                    }
                ]
            }
        }
    ]
```

4）HAP 与 HAR 的配置文件的合并

（1）合并规则如下：

如果在应用模块中调用了 HAR，在编译构建 HAP 时，则需要将 HAP 的 config.json 文件与一个或多个 HAR 的 config.json 文件合并为一个 config.json 文件。在合并过程中，不同文件的同一个标签的取值可能发生冲突，此时需要通过配置 mergeRule 来解决冲突。

配置文件合并规则。HAP 与 HAR 的 config.json 文件在合并时，需要将 HAR 的配置信息全部合并到 HAP 的配置文件中。合并规则参见表 3-1。

表 3-1　合并规则

HAP	HAR	合并结果
无标签值	无标签值	无标签值
有标签值，取值为 A	无标签值	有标签值，取值为 A
无标签值	有标签值，取值为 B	有标签值，取值为 B
有标签值，取值为 A	有标签值，取值为 A	有标签值，取值为 A
有标签值，取值为 A	有标签值，取值为 B	冲突，需要添加 mergeRule，详见 mergeRule 对象的使用

HAP的优先级总是高于HAR。当HAP依赖于多个HAR时，先加载的HAR的优先级高于后加载的HAR，按照HAR的加载顺序依次合并到HAP文件中。两者合并规则见表3-1。

(2) mergeRule对象的使用。

mergeRule通常在HAP的config.json文件中使用，可以在abilities、defPermissions、reqPermissions、js等属性中添加。不同属性的合并策略将会详细说明。其中HAR配置文件中不能包含action.system.home和entity.system.home配置项，否则会导致编译报错。abilities对象中name字段的取值必须为完整的类名，否则会导致合并出错。

不同属性的合并策略，我们会根据属性名称，一级、二级、三级属性及分别的合并规则进行说明。

一级属性app合并时只保留HAP的config.json文件中的app对象。一级属性deviceConfig合并时只保留HAP的config.json文件中的deviceConfig对象。

一级属性module的二级属性package、name、description、supportedModes、deviceType、distro、shortcuts合并时只保留HAP的config.json文件中的取值。

一级属性module的二级属性defPermissions、reqPermissions、js、abilities合并时，当module中的name取值不同时，取值为HAP与HAR的config.json文件的并集。当module中的name取值相同时，需要在HAP的config.json文件中的相应属性下添加mergeRule字段，以解决合并冲突。

一级属性module的二级属性abilities的三级属性permissions、skills、backgroundModes、configChanges合并时取值为HAP与HAR的config.json文件中相应属性值的并集。

一级属性module的二级属性abilities的三级属性targetAbility合并时，如果targetAbility与abilities中的name冲突，则会导致编译报错。

一级属性module的二级属性abilities的三级属性其他，合并时abilities中的其他属性如果发生合并冲突，则需要添加mergeRule字段。

(3) mergeRule对象的内部结构说明。

属性remove表示HAP与HAR的config.json文件合并时需要移除的标签。数据类型为字符串数组型。可缺省。属性replace表示HAP与HAR的config.json文件合并冲突时需要替换的标签，始终保留高优先级的值。数据类型为字符串数组型。可缺省。

(4) mergeRule的使用示例。

在下面的示例中，HAP与HAR中的Ability的name取值相同，需要对两者config.json文件中的Ability进行合并。由于两个文件中的部分字段(例如launchType)存在冲突，所以需要在HAP的abilities标签下添加mergeRule。

第一步，合并前HAP的config.json文件，其中，remove表示合并后需要移除的子标签，replace表示合并后需要替换的子标签(HAP替换HAR)，代码如下：

```
"//":"第 3 章/mergeRule 使用示例第一步"
"abilities": [
    {
       "mergeRule": {
          "remove": ["orientation"],
          "replace": ["launchType"]
       }
       "name": "com.harmony.myapplication.entry.MainAbility",
       "type": "page",
       "launchType": "standard",
       "visible": false
    }
]
```

第二步，合并前 HAR 的 config.json 文件，代码如下：

```
"//":"第 3 章/mergeRule 使用示例第二步"
"abilities": [
    {
       "name": "com.harmony.myapplication.entry.MainAbility",
       "type": "page",
       "launchType": "singleton",
       "orientation": "portrait",
       "visible": false
    }
],
```

第三步，将上述两个 config.json 文件按照 mergeRule 进行合并，处理完成后 mergeRule 字段也会被移除。合并后的结果文件，代码如下：

```
"//":"第 3 章/mergeRule 使用示例第三步"
"abilities": [
    {
       "name": "com.harmony.myapplication.entry.MainAbility",
       "type": "page",
       "launchType": "standard",
       "visible": false
    }
],
```

(5) bundleName 占位符的使用。

HAR 的 config.json 文件中多处需要使用包名，例如自定义权限、自定义 action 等场景，但是包名只有当 HAR 编译到 HAP 时才能确定下来。在编译之前，HAR 中的包名可以采用占位符来表示，采用{bundleName}形式。支持 bundleName 占位符的标签有

actions、entities、permissions、readPermission、writePermission、defPermissions. name、uri。

使用示例如下：

第一步，在 HAR 中自定义 action 时，使用{bundleName}来代替包名，代码如下：

```
"//":"第 3 章/bundleName 使用示例第一步"
"skills": [
    {
        "actions": [
            "{bundleName}.ACTION_PLAY"
       ],
        "entities": [
            "{bundleName}.ENTITY_PLAY"
        ],
    }
],
```

第二步，将 HAR 编译到 bundleName 为 com. huawei. hiworld 的 HAP 包后，原来的{bundleName}将被替换为 HAP 的实际包名。替换后的结果如下：

```
"//":"第 3 章/bundleName 使用示例第二步"
"app": {
  "bundleName": "com.huawei.hiworld",
    …
},
"module": {
    "abilities": [
        {
            "skills": [
                {
                    "actions": [
                        "com.huawei.hiworld.ACTION_PLAY"
                    ],
                    "entities": [
                        "com.huawei.hiworld.ENTITY_PLAY"
                    ],
                }
            ],
```

3. 配置文件示例

该示例的应用被声明为 3 个 Ability，代码如下：

```
"//":"第 3 章/配置文件示例"
{
    "app": {
```

```
        "bundleName": "com.huawei.hiworld.himusic",
        "vendor": "huawei",
        "version": {
            "code": 2,
            "name": "2.0"
        },
        "apiVersion": {
            "compatible": 3,
            "target": 3,
            "releaseType": "Beta1"
        }
    },
    "deviceConfig": {
        "default": {
        }
    },
    "module": {
        "mainAbility": "MainAbility",
        "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": "$media:ic_launcher",
                "label": "$string:HiMusic",
                "launchType": "standard",
                "orientation": "unspecified",
                "visible": true,
                "skills": [
                    {
                        "actions": [
                            "action.system.home"
                        ],
                        "entities": [
```

```
                        "entity.system.home"
                    ]
                }
            ],
            "type": "page",
            "formsEnabled": false
        },
        {
            "name": ".PlayService",
            "description": "himusic play ability",
            "icon": "$media:ic_launcher",
            "label": "$string:HiMusic",
            "launchType": "standard",
            "orientation": "unspecified",
            "visible": false,
            "skills": [
                {
                    "actions": [
                        "action.play.music",
                        "action.stop.music"
                    ],
                    "entities": [
                        "entity.audio"
                    ]
                }
            ],
            "type": "service",
            "backgroundModes": [
                "audioPlayback"
            ]
        },
        {
            "name": ".UserADataAbility",
            "type": "data",
            "uri": "dataability://com.huawei.hiworld.himusic.UserADataAbility",
            "visible": true
        }
    ],
    "reqPermissions": [
        {
            "name": "ohos.permission.DISTRIBUTED_DATASYNC",
            "reason": "",
            "usedScene": {
                "ability": [
                    "com.huawei.hiworld.himusic.entry.MainAbility",
```

```
                        "com.huawei.hiworld.himusic.entry.PlayService"
                    ],
                    "when": "inuse"
                }
            }
        ]
    }
}
```

3.1.3 资源文件

1. 资源文件的分类

resources 目录，应用的资源文件（如字符串、图片、声频等）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。资源目录示例，代码如下：

```
//第 3 章/资源文件的分类 - resources 目录
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
```

1）base 目录与限定词目录分类

按照两级目录的形式来组织，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。官方提供了限定词目录表。

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，具体要求参见官方提供的资源组目录。

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。通过指定资源类型（type）和资源名称（name）来引用。

2）rawfile 目录分类

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。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。通过指定文件路径和文件名来引用。

3）限定词目录详细说明

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，包括移动国家码和移动网络码、语言、文字、国家或地区、横竖屏、设备类型、颜色模式和屏幕密度等维度，限定词之间通过下画线(_)或者中画线(-)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，以及限定词目录与设备状态的匹配规则。

(1) 限定词目录的命名要求。

限定词的组合顺序：移动国家码_移动网络码-语言_文字_国家或地区-横竖屏-设备类型-颜色模式-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画线(_)连接，移动国家码和移动网络码之间也采用下画线(_)连接，除此之外的其他限定词之间均采用中画线(-)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取值的具体要求，否则，将无法匹配目录中的资源文件。

(2) 限定词取值的具体要求如下，包括限定词类型、含义与取值说明。

移动国家码和移动网络码，移动国家码(MCC)和移动网络码(MNC)的值取自设备注册的网络。MCC 后面可以跟随 MNC，使用下画线(_)连接，也可以单独使用。例如：mcc460 表示中国，mcc460_mnc00 表示中国_中国移动。详细取值范围需要查阅 ITU-T E.212(国际电联相关标准)。

语言表示设备使用的语言类型，由 2～3 个小写字母组成。例如：zh 表示中文，en 表示英语，mai 表示迈蒂利语。详细取值范围需要查阅 ISO 639(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。详细取值范围需要查阅 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。详细取值范围需要查阅 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，取值范围：vertical 为竖屏，horizontal 为横屏。

设备类型，表示设备的类型，取值范围：phone 为手机，tablet 为平板，car 为车机，tv 为智慧屏，wearable 为智能穿戴。

颜色模式，表示设备的颜色模式，取值范围：dark 为深色模式，light 为浅色模式。

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),适用于 dpi 取值为(0, 120]的设备。mdpi 表示中规模的屏幕密度(Medium-scale Dots Per Inch),适用于 dpi 取值为(120, 160]的设备。ldpi 表示大规模的屏幕密度(Large-scale Dots Per Inch),适用于 dpi 取值为(160, 240]的设备。xldpi 表示特大规模的屏幕密度(Extra Large-scale Dots Per Inch),适用于 dpi 取值为(240, 320]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 dpi 取值为(320, 480]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 dpi 取值为(480, 640]的设备。

(3) 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移动国家码和移动网络码→区域(可选组合:语言、语言_文字、语言_国家或地区、语言_文字_国家或地区)→横竖屏→设备类型→颜色模式→屏幕密度。

如果限定词目录中包含移动国家码和移动网络码、语言、文字、横竖屏、设备类型、颜色模式限定词,则对应限定词的取值必须与当前的设备状态完全一致,只有这样该目录才能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。详细的资源组目录说明如下:

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 为布尔型;color 为颜色;float 为浮点型;intarray 为整型数组;integer 为整型;pattern 为样式;plural 为复数形式;strarray 为字符串型数组;string 为字符串。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;这些文件名称分别是 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、声频、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。

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。

4) 创建资源文件

在 resources 目录下,可按照限定词目录和资源组目录的说明创建子目录和目录内的文件。同时,DevEco Studio 也提供了创建资源目录和资源文件的界面。

创建资源目录及资源文件。在 resources 目录右击菜单选择 New→Harmony Resource File,此时可同时创建目录和文件。文件默认创建在 base 目录的对应资源组下。如果选择了限定词,则会按照命名规范自动生成限定词+资源组目录,并将文件创建在目录中。目录

名自动生成，格式固定为“限定词.资源组”，例如创建一个限定词为横竖屏类别下的竖屏，资源组为绘制资源的目录，自动生成的目录名称为 vertical.graphic。

创建资源目录，在 resources 目录右击菜单选择 New→Harmony Resource Directory，此时可创建资源目录，具体如图 3-3 所示。

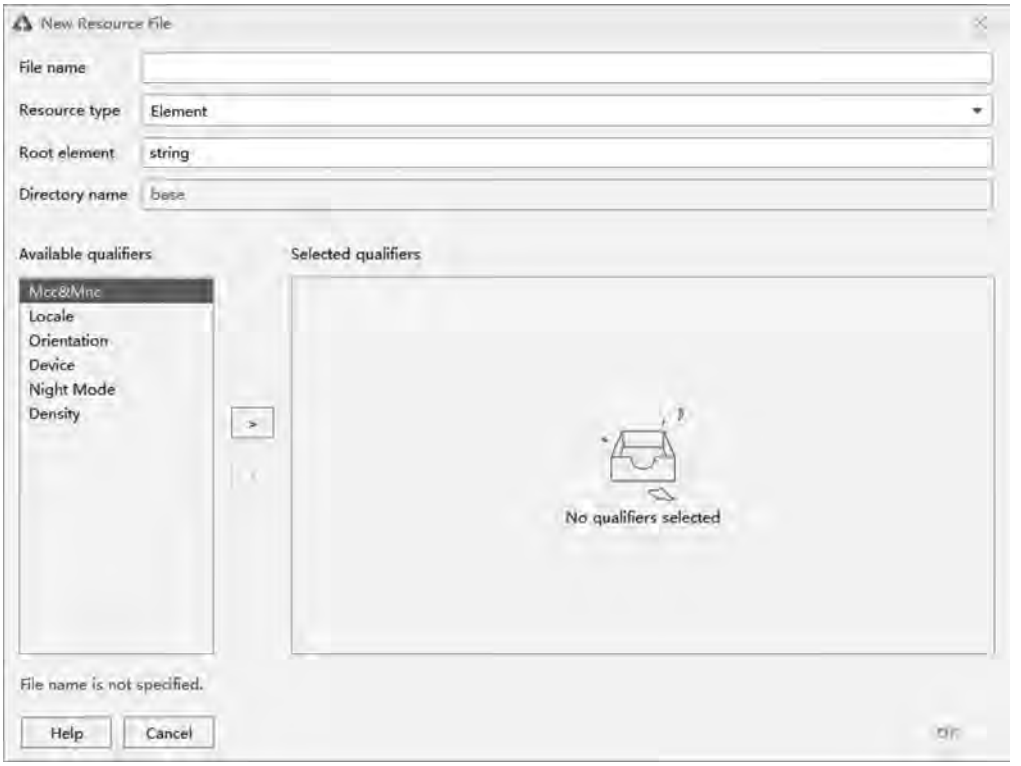

图 3-3　创建资源文件

选择资源组类型，设置限定词，创建后可自动生成目录名称。目录名称格式固定为“限定词.资源组”，例如创建一个限定词为横竖屏类别下的竖屏，资源组为绘制资源的目录，自动生成的目录名称为 vertical.graphic，如图 3-4 所示。

创建资源文件，在资源目录右击菜单并选择 New→XXX Resource File，即可创建对应资源组目录的资源文件。例如，在 element 目录下可新建 Element Resource File，如图 3-5 所示。

2. 资源文件的使用

1）资源文件的引用方法

base 目录与限定词目录中的资源文件可通过指定资源类型（type）和资源名称（name）来引用。

Java 文件引用资源文件的格式为 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。

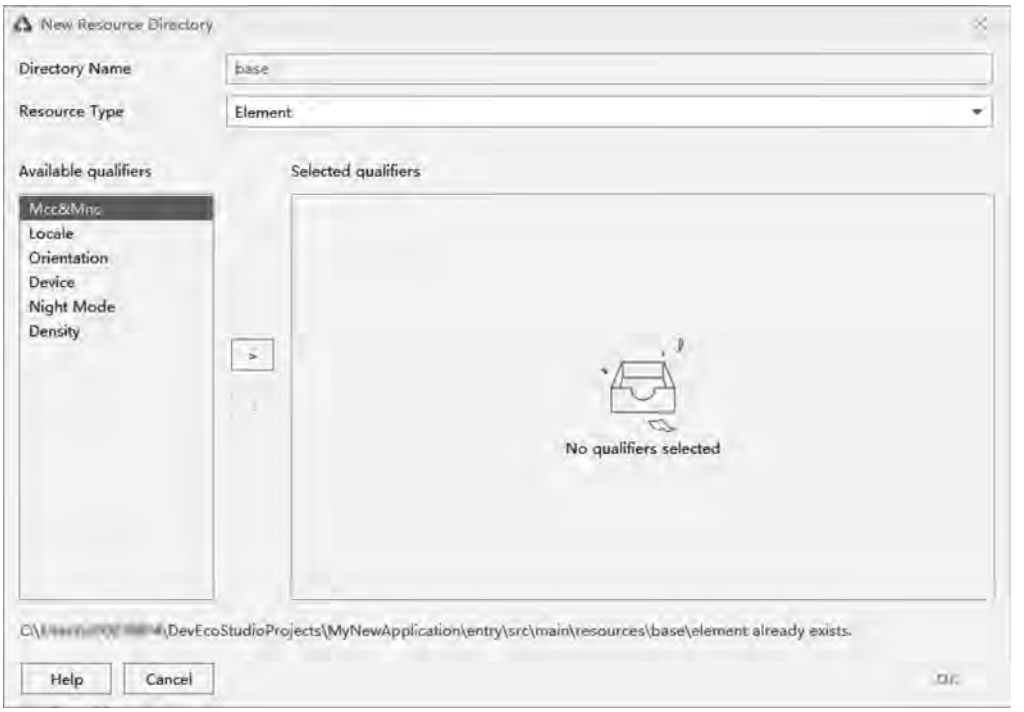

图 3-4　创建资源文件

图 3-5　创建资源文件

示例一：在 Java 文件中，引用 string.json 文件中类型为 String、名称为 app_name 的资源，代码如下：

```
//第 3 章/引用 app_name 资源文件
ohos.global.resource.ResourceManager resManager = this.getResourceManager();
String result = resManager.getElement(ResourceTable.String_app_name).getString();
```

示例二：在 Java 文件中，引用 color.json 文件中类型为 Color、名称为 red 的资源，代码

如下：

```
ohos.global.resource.ResourceManager resManager = this.getResourceManager();
int color = resManager.getElement(ResourceTable.Color_red).getColor();
```

XML 文件引用资源文件的格式为 $type:name。特别地，如果引用的是系统资源，则采用 $ohos:type:name。

在 XML 文件中，引用 string.json 文件中类型为 String，名称为 app_name 的资源，代码如下：

```
//第 3 章/XML 文件引用资源文件
<?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">
    < Text ohos:text = " $ string:app_name"/>     //引用资源文件
</DirectionalLayout >
```

rawfile 目录中的资源文件可通过指定文件路径和文件名称来引用。在 Java 文件中，引用一个路径为 resources/rawfile/、名称为 example.js 的资源文件，代码如下：

```
//第 3 章/引用 example.js 资源文件
ohos.global.resource.ResourceManager resManager = this.getResourceManager();
ohos.global.resource.RawFileEntry rawFileEntry = resManager.getRawFileEntry("resources/
rawfile/example.js");
```

2）系统资源文件

目前支持的部分系统资源文件如下。具体描述方式包括系统资源名称、含义与类型。

ic_app 表示 HarmonyOS 应用的默认图标，文件类型为媒体。

request_location_reminder_title 表示“请求使用设备定位功能”的提示标题，文件类型为字符串型。

request_location_reminder_content 表示“请求使用设备定位功能”的提示内容，即在下拉快捷栏打开“位置信息”开关，文件类型为字符串型。

3）颜色模式的定义

应用可以在 config.json 文件的 module 字段下定义 colorMode 字段，colorMode 字段用来定义应用自身的颜色模式，值可以是 dark、light、auto(默认值)，示例代码如下：

```
"colorMode": "light"
```

当应用的颜色模式值是 dark 时，无论系统当前的颜色模式是什么，应用始终会按照深

色模式选取资源；同理，当应用的颜色模式值是 light 时，无论系统当前的颜色模式是什么，应用始终会按照浅色模式选取资源；当应用的颜色模式值是 auto 时，应用会跟随系统的颜色模式值选取资源。应用可以在代码中获取应用当前的颜色模式，具体获取方式如下：

```
int colorMode = Configuration.colorMode;
```

4）为 Element 资源文件添加注释或特殊标识

Element 目录下的不同种类元素的资源均采用 JSON 文件表示，资源的名称 name 和取值 value 是每一条资源的必备字段。如果需要为某条资源备注信息，以便于资源的理解和使用，则可以通过 comment 字段添加注释。如果 value 字段中的部分文本不需要被翻译人员处理，也不会被显示在应用界面上，则可以通过特殊结构来标识无须翻译的内容。

5）通过 comment 字段添加注释

通过 comment 字段，可以为 JSON 文件的资源添加注释，示例代码如下：

```
"//":"第 3 章/通过 comment 字段添加注释"
{
    "string":[
        {
            "name":"message_arrive",
            "value":"We will arrive at %s",
            "comment":"Transfer Arrival Time. %s is time,like 5:00 am"
        }
    ]
}
```

6）通过特殊结构来标识无须翻译的内容

在 string、strarray、plural 这三类资源中，可以通过特殊标识来处理无须被翻译的内容。例如，一个字符串资源的 Value 取值为 We will arrive at %s，其中的变量%s 在翻译过程中希望保持不变。有以下两种处理方式。

方式一：在 value 字段中添加{}，示例代码如下：

```
"//":"第 3 章/通过特殊结构来标识无须翻译的内容 - 方法一"
{
    "string":[
        {
            "name":"message_arrive",
            "value":["We will arrive at",{
                "id":"time",
                "example":"5:00 am",
                "value":"%s"
            }
```

```
            ]
        }
    ]
}
```

方式二：添加< xliff:g ></xliff:g >标记对，示例代码如下：

```
"//":"第 3 章/通过特殊结构来标识无须翻译的内容 - 方法二"
{
    "string":[
        {
            "name":"message_arrive",
            "value":"We will arrive at <xliff:g id = 'time' example = '5:00 am'> % s </xliff:g >"
        }
    ]
}
```

7）系统资源文件各项示例

（1）boolean.json 示例，代码如下：

```
"//":"第 3 章/系统资源文件各项示例 - boolean.json 示例"
{
   "boolean":[
        {
            "name":"boolean_1",
            "value":true
        },
        {
            "name":"boolean_ref",
            "value":" $ boolean:boolean_1"
        }
   ]
}
```

（2）color.json 示例，代码如下：

```
"//":"第 3 章/系统资源文件各项示例 - color.json 示例"
{
   "color":[
        {
            "name":"red",
            "value":"#ff0000"
        },
        {
            "name":"red_ref",
```

```
            "value":"$color:red"
        }
    ]
}
```

(3) float.json 示例,代码如下:

```
"//":"第3章/系统资源文件各项示例 - float.json 示例"
{
   "float":[
       {
           "name":"float_1",
           "value":"30.6"
       },
       {
           "name":"float_ref",
           "value":"$float:float_1"
       },
       {
           "name":"float_px",
           "value":"100px"
       }
   ]
}
```

(4) intarray.json 示例,代码如下:

```
"//":"第3章/系统资源文件各项示例 - intarray.json 示例"
{
   "intarray":[
       {
           "name":"intarray_1",
           "value":[
               100,
               200,
               "$integer:integer_1"
           ]
       }
   ]
}
```

(5) integer.json 示例,代码如下:

```
"//":"第3章/系统资源文件各项示例 - integer.json 示例"
{
```

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

（6）pattern.json 示例，代码如下：

```
"//":"第 3 章/系统资源文件各项示例 - 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":"noTitle",
                    "value":"Yes"
                }
            ]
        }
    ]
}
```

（7）plural.json 示例，代码如下：

```
"//":"第 3 章/系统资源文件各项示例 - plural.json 示例"
{
   "plural":[
       {
           "name":"eat_apple",
           "value":[
               {
                   "quantity":"one",
                   "value":" % d apple"
               },
               {
                   "quantity":"other",
                    "value":" % d apples"
               }
           ]
       }
   ]
}
```

（8）strarray.json 示例，代码如下：

```
"//":"第 3 章/系统资源文件各项示例 - strarray.json 示例"
{
   "strarray":[
      {
          "name":"size",
          "value":[
              {
                  "value":"small"
              },
              {
                  "value":" $ string:hello"
              },
              {
                  "value":"large"
              },
              {
                  "value":"extra large"
              }
           ]
      }
   ]
}
```

（9）string.json 示例，代码如下：

```
"//":"第 3 章/系统资源文件各项示例 - 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"
       }
   ]
}
```

3. 国际化能力的支持

时间日期国际化，不同的区域具有不同的时间日期显示习惯。例如，英语（美国）区域 short 时间格式为“9:31 AM”；简体中文（中国）区域 short 时间格式为“上午 9:31”；芬兰语（芬兰）区域 short 时间格式为“9.31”。

因此为开发者提供了获取不同区域的时间日期规格的能力。界面时间日期字串和时间类控件显示，应当遵循当地习惯的规则。当需要展示时间或日期时，建议获取当前地区的时间日期规格，并对显示的字串根据获取的规格进行格式化后再使用。

获取不同区域的时间日期规划能力，示例 1，代码如下：

```
Locale locale = new Locale("de", "CH");
String skeleton = "MMMMd";
String bestPattern = DateFormatUtil.getBestPattern(skeleton, locale); //返回值为"d. MMMM"
```

获取不同区域的时间日期规划能力，示例 2，代码如下：

```
String languageTag = "zh";
String out = DateFormatUtil.format("EEEEdMMMMy", languageTag, "Asia/Shanghai", 0, 3600 *
1000);   //返回值为"1970 年 1 月 1 日星期四"
```

电话号码国际化。不同区域的电话号码有不同的格式化效果，当需要展示本地电话号码时，应遵循当地电话号码的格式化原则，因此为开发者提供了对不同地区电话号码格式化的能力，以便于在显示电话号码时正确地格式化，并提供了获取电话号码归属地的能力，开发者可以使用相关接口获取电话号码的归属地信息。

使用相关接口获取电话号码的归属地信息，示例 1，代码如下：

```
//第 3 章/获取电话号码的归属地信息示，示例 1
InputFormatter formatter = InputFormatter.getInstance("CN");
formatter.inputNumberAndRememberPosition('1');          //返回值为"1"
formatter.inputNumber('5');                             //返回值为"15"
formatter.inputNumber('6');                             //返回值为"156"
formatter.inputNumberAndRememberPosition('1');          //返回值为"156 1"
```

使用相关接口获取电话号码的归属地信息，示例 2，代码如下：

```
//第 3 章/获取电话号码的归属地信息，示例 2
Locale.Builder builder = new Locale.Builder();
builder.setLanguage("zh");
builder.setRegion("CN");
builder.setScript("Hant");
Locale locale = builder.build();
String displayName = PhoneNumberAttribution.getAttribute(" + 8615611xxxxxx", "CN", locale);
                                                        //x 为任意数字，返回值为"北京市"
```

文本识别。提供了对地址、时间日期与电话号码的文本识别能力，可以调用相关接口识别一段文本中包含的地址、时间日期与电话号码，示例代码如下：

```
//当 Locale.getDefault().getLanguage()为"en"时
String source = "it is 123 main St";
int[] re = TextRecognitionUtils.getAddress(source);
if (re[0] == 1) {
    result = source.substring(re[1], re[2] + 1);  //返回值为"123 main St"
}
```

度量衡格式化。提供了对度量衡国际化能力的支持，可支持度量衡体系和维度之间的转换，与不同国家度量衡体系的自动转换。在开发包含度量衡的功能时，可以调用此能力满足多语言和不同国家用户的需求，示例 1，代码如下：

```
//第 3 章/度量衡格式化，示例 1
Locale zhCN = Locale.CHINA;
MeasureFormatter mes = MeasureFormatter.getInstance(zhCN);
mes.format(MeasureOptions.Unit.AREA_UK_ACRE,
           10000,
           MeasureOptions.Usage.AREA_LAND_AGRICULT,
```

```
            MeasureOptions.FormatStyle.WIDE,
            MeasureOptions.Style.AUTO_STYLE_ON);  //返回值为"4,046.856 公顷"
```

示例 2 代码如下：

```
//第 3 章/度量衡格式化,示例 2

Locale enUS = Locale.US;
MeasureFormatter mes = MeasureFormatter.getInstance(enUS);
mes.format(MeasureOptions.Unit.VOLUME_US_CUP,
            1000,
            MeasureOptions.Unit.VOLUME_SI_LITER,
            MeasureOptions.FormatStyle.WIDE);  //返回值为"236.588 liters"
```

敏感禁忌。提供对政治敏感地区、城市及语言的获取能力，以及对地区名称更正的能力，示例代码如下：

```
Locale locale = Locale.getDefault();
ArrayList<String> result = LocaleHelperUtils.getBlockedRegions(context, locale);
       //返回值包含"EH"与"XK"(西撒哈拉与科索沃),这两个地区为有政治争议的地区需谨慎使用
```

3.1.4　应用数据管理

HarmonyOS 应用数据管理支持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。

1. 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)、对象关系映射数据库(Object Relational Mapping Database)、轻量级偏好数据库(Light Weight Preference Database)等，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。

2. 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。通过结合账号、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。

3. 分布式文件服务

在多个终端设备间为单个设备上由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的路径，可以实现同一应用文件的跨设备访问。

4. 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。

5. 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。

3.1.5 应用安全管理

1. 应用开发准备阶段

依据国家《移动互联网应用程序信息服务管理规定》，同时为了促进生态健康有序发展，保护应用开发者和用户的合法权益，需要每位 HarmonyOS 开发者注册账号，并建议同步进行实名认证。实名认证包括个人开发者实名认证和企业开发者实名认证，没有完成实名认证的开发者，无法进行应用上架发布。

在发布 HarmonyOS 应用前，可以在本地对应用进行调试。HarmonyOS 通过数字证书和 Profile 文件对应用进行管控，只有经过签名的 HAP 才允许安装到设备上运行。

2. 应用开发调试阶段

1）编码安全

避免不对外交互的 Ability 被其他应用直接访问。避免带有敏感功能的公共事件被其他应用直接访问。避免通过隐式方式对组件进行调用，防止组件劫持。避免通过隐式方式发送公共事件，防止公共事件携带的数据被劫持。

应用作为数据使用方需校验数据提供方的身份，防止被仿冒后进行攻击。对跨信任边界传入的 Intent 须进行合法性判断，防止应用异常崩溃。避免在配置文件中开启应用备份和恢复开关。避免将敏感数据存放到剪贴板中。避免将敏感数据写入公共数据库、存储区中。避免直接使用不可信数据来拼接 SQL 语句。避免向可执行函数传递不可信数据。避免使用 Socket 方式进行本地通信，如需使用，localhost 端口号应随机生成，并对端口连接对象进行身份认证和鉴权。

建议使用 Https 代替 Http 进行通信，并对 Https 证书进行严格校验。建议使用校验机制保证 WebView 在加载网站服务时 URL 网址的合法性。对于涉及支付及高保密数据的应用，建议进行手机 root 环境监测。建议开启安全编译选项，增加应用分析逆向难度。禁止应用执行热更新操作，应用更新可以通过应用市场上架来完成。建议应用在开发阶段进行自测试，具体参考应用安全测试。

2）权限使用

应用申请的权限，都必须有明确、合理的使用场景和功能说明，确保用户能够清晰明了地知道申请权限的目的、场景、用途；禁止诱导、误导用户授权；应用使用权限必须与申请所述一致。

应用权限申请遵循最小化原则，只申请业务功能所必要的权限，禁止申请不必要的权限。

应用在首次启动时，避免频繁弹窗申请多个敏感权限；敏感权限须在用户使用对应业

务功能时动态申请。

当用户拒绝授予某个权限时，与此权限无关的其他业务功能应能正常使用，不能影响应用的正常注册或登录。

业务功能所需要的权限被用户拒绝且禁止后不再提示，当用户主动触发使用此业务功能或为实现业务功能所必须时，应用程序可通过界面内文字引导，让用户主动到“系统设置”中授权。

非系统应用自定义权限名，禁止使用系统权限名前缀，如以 ohos 开头为系统权限，建议以应用包名或公司反域名为前缀，防止与系统或其他应用定义的权限重名。

有关于应用动态申请敏感权限的详细信息，在动态申请权限中会阐述。

3. 应用发布分发阶段

应用发布，应用调试完毕后，可以进行打包 HarmonyOS 应用，在 AGC 提交上架申请。为了确保 HarmonyOS 应用的完整性，确保提交应用的开发者身份合法，HarmonyOS 通过数字证书和 Profile 文件对应用进行管控。上架到华为应用市场的 App 必须通过签名才允许上架，因此，为了保证应用能够顺利发布，需要提前申请相应的发布证书与发布 Profile。

提交发布申请后，应用市场将对应用进行安全审核，包括权限、隐私、安全等，如果审核不通过，则不能上架；应用发布成功后，华为应用市场会对上架应用进行重签名，原有的应用签名将被替换为新签名。

3.1.6 应用隐私保护

应用开发者在产品设计阶段就需要考虑用户隐私的保护，从而提高应用的安全性。HarmonyOS 应用开发需要遵从隐私保护规则，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。

1. 数据收集及使用应公开透明

当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。

当应用申请操作系统敏感权限时，需要明确告知用户权限申请的目的和用途，并获得用户的同意。

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。如在收集个人数据前，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。

应用向第三方披露任何个人信息须在隐私政策中说明披露内容、目的和披露对象。

对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。

应用需要提供用户查看隐私声明的入口。

应用的隐私声明应覆盖本应用所有收集的个人数据。在后台持续读取位置信息场景时，应申请 ohos.permission.LOCATION_IN_BACKGROUND 权限；当应用存在调用第

三方的原子化服务场景时，需要在应用的隐私声明中明确第三方责任，如涉及个人数据收集，则需要告知用户第三方的名称及收集的个人数据类型、目的和方式，以及申请的敏感权限、申请目的等。

2. 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，并且是适当、必要的。开发者应尽可能对个人数据进行匿名化或假名化处理，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处理。

在对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。如应用不需要相机权限就能够实现其功能时，则不应该向用户申请相机权限。

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。如通信社交类应用，不应收集用户的网页浏览记录等。

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。如应用不得将“生物特征”“健康数据”等敏感个人数据用于服务改进、投放广告或营销等非业务核心功能。

系统禁止应用在后台访问相机和话筒的数据。

应用在使用第三方支付交易过程中，如非适用法律要求或为提供第三方支付服务所必需的，不得记录用户交易类鉴权信息，或向第三方披露与用户特定交易无关的用户个人信息。

应用不得仅出于广告投放或数据分析的目的而请求位置权限。

禁止在日志中打印敏感个人数据，如需要打印个人数据时，应对个人数据进行匿名化或假名化处理；避免使用 IMEI 和序列号等永久性的标识符，尽量使用可以重置的标识符，如系统提供了 NetworkID 和 DVID 作为分布式场景下的设备标识符，广告业务场景下则建议使用 OAID，基于应用的分析则建议使用 ODID 和 AAID，其他需要唯一标识符的场景可以使用 UUID 接口生成；不再使用的数据需要及时清除，降低数据泄露的风险。如分布式业务场景下设备断开分布式网络，临时缓存的数据需要及时删除。

3. 数据处理的选择和控制

对个人数据处理必须征得用户的同意或遵守适用的法律法规，用户对其个人数据要有充分的控制权。

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。

应用申请使用敏感权限要提供应用弹窗提醒，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，通过用户单击“允许”或“仅使用期间允许”或“允许本次使用”的方式完成用户授权，让用户对应用权限的授予和个人数据的使用做到透明、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。如通信社交类应用，用户可以拒绝授

予相机权限，不应该影响与相机无关的功能操作，如语音通话。

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。如通信社交类应用，在没有启用位置相关的功能时，不建议在启动应用时就申请位置权限。

应用若使用个人数据用于个性化广告和精准营销，则需提供独立的关闭选项。

需要向用户提供对个人数据的控制能力，如在云服务上存储了个人数据，需要提供删除数据的方法。

当应用同时支持单设备和跨设备场景时，用户能够单独关闭跨设备应用场景。

4. 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，应默认开启或采取安全保护措施。

数据存储，应用产生的密钥及用户的敏感个人数据需要存储在应用的私有目录下。应用可以调用系统提供的本地数据库 RdbStore 的加密接口对敏感个人数据进行加密存储。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，对于敏感个人数据需要采用分布式数据库提供的加密接口进行加密。

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。

本地传输是应用通过 intent 跨应用传输数据时避免包含敏感个人数据，防止隐式调用导致 intent 劫持，从而导致个人数据泄露。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 端口号应随机生成，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。

远程传输时应使用 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，防止对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。

5. 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时要满足最小化的原则，不能默认选择上传云服务。

6. 未成年人数据保护要求

如果应用是为未成年人设计的，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。专为未成年人设计的应用不建议请求获取位置权限。

3.1.7 第三方应用调用管控机制

1. 为什么要进行调用管控

后台进程启动过多，会消耗系统的内存、CPU 等资源，造成用户设备耗电快、卡顿等现象，因此，为了保证用户体验，系统会对第三方用户应用程序之间的 PA 调用进行管控，减少不必要的关联拉起。其中需要说明的是第三方应用的概念，第三方应用是相对于系统应用（不可卸载或者 appId＜10000 的应用）而言的，由第三方开发的用户应用程序。

2. 相关概念

前台，如果用户应用程序有可见的 FA 正在显示，则认为用户应用程序在前台。用户应用程序内调用指同一用户应用程序内的 FA、PA 之间的访问。

调用管控的总体思路。第一、用户应用程序内调用不管控。第二、第三方用户应用程序间调用严格管控，禁止第三方用户应用程序在后台调用其他第三方应用的 PA；严格管控第三方用户应用程序在前台调用其他用户应用程序的 PA。

3. 管控规则

用户应用程序内调用不管控。第三方用户应用程序间调用需要管控。第三方应用程序 A 调用第三方应用程序 B 的 PA，具体限制为禁止 A 在后台调用 B 的 PA。当 B 有进程存活时，允许 A 在前台调用 B 的 PA；当 B 无进程存活时，禁止 A 的调用。

3.2 原子化服务总体开发要求

本节主要阐释原子化服务总体开发的各项基本要求与原理，具体包括便捷服务基础信息开发要求、原子化服务与服务卡片的相关运行原理、服务卡片多种语言开发的对比、JS 服务卡片开发与语法阐述。

3.2.1 综述

原子化服务相对于传统方式的需要安装的应用更加轻量，同时提供更丰富的入口、更精准的分发，需要满足一些开发规则及要求。

第一条规则是原子化服务内所有 HAP 包，包括 Entry HAP 和 Feature HAP 均需满足免安装要求。原子化服务由一个或多个 HAP 包组成，1 个 HAP 包对应 1 个 FA 或 1 个 PA。

免安装的 HAP 包不能超过 10MB，以提供秒开体验。超过此大小的 HAP 包不符合免安装要求，也无法在服务中心提供服务。

通过 DevEco Studio 工程向导创建原子化服务，Project Type 字段选择 Service。

对于原子化服务升级场景，版本更新时要保持免安装属性。如果新版本不支持免安装，则将不允许新版本上架发布。

支持免安装 HAP 包的设备类型如下。手机、平板、智能穿戴、智慧屏支持免安装 HAP

包，支持的版本为 HarmonyOS 2.0 及以上。轻智能穿戴、车机、音箱、计算机、耳机、眼镜在本书创作期间，还在规划中。

第二条规则是如果某便捷服务的入口需要在服务中心提供服务，则该服务对应的 HAP 包必须包含 FA，并且 FA 中必须指定一个唯一的 mainAbility，定位为用户操作入口，mainAbility 必须为 Page Ability。同时，mainAbility 中至少配置 2×2(小尺寸)规格的默认服务卡片，也可以同时提供其他规格的卡片及该便捷服务对应的基础信息，包括图标、名称、描述、快照。

通过 DevEco Studio 工程向导创建工程时，Project Type 字段选择 Service，同时勾选 Show in Service Center。这样，工程中将自动指定 mainAbility，并添加默认服务卡片信息，开发者根据实际业务继续开发即可。

3.2.2 便捷服务基础信息开发指导

1. 基本概念

原子化服务中的每个便捷服务应有独立的图标、名称、描述、快照，这些称为便捷服务基础信息。基础信息应能够准确反映便捷服务提供方的特征及便捷服务的核心体验。便捷服务基础信息将展示在服务中心、搜索等界面。基础信息具有详细设计规范，我们在设计相关章节中会详细阐述。笔者创作本书时支持配置基础信息的设备类型包括手机、平板、智能穿戴和智慧屏。

2. 开发步骤

配置便捷服务的图标、名称、描述信息。在作为该便捷服务入口的 HAP 包的 config.json 配置文件中，为 mainAbility 配置图标(icon)、名称(label)、描述(description)。其中，mainAbility 的 label 标签是便捷服务对用户显示的名称，必须配置，并且应以资源索引的方式配置，以支持多语言。不同 HAP 包的 mainAbility 的 label 要唯一，以免造成用户看到多个同名服务而无法区分。此外，label 的命名应与服务内容强关联，能够通过显而易见的语义看出服务的关键内容。便捷信息在以下示例代码中关于图标、名称、描述的说明如下：

(1) label 在 entry\src\main\resources\base\element\string.json 文件中，用于定义便捷服务对用户显示的名称，然后在 config.json 文件中以索引方式引用"label"。

(2) icon 表示开发者将便捷服务的图标 png 文件放至 entry\src\main\resources\base\media 目录下，然后在 config.json 文件中以索引方式引用"icon"。

(3) description 在 entry\src\main\resources\base\element\string.json 文件中，用于定义便捷服务的简要描述，然后在 config.json 文件中以索引方式引用"description"。

便捷信息，示例代码如下：

```
"//": "第 3 章/配置便捷服务信息"
{
   ...
   "abilities": [
```

```
      {
        "skills": [
          {
            "entities": [
              "entity.system.home"
            ],
            "actions": [
              "action.system.home"
            ]
          }
        ],
        "name": "com.example.xxx.MainAbility",
        "icon": "$media:icon",
        "description": "$string:mainability_description",
       "label": "$string:mainability_label",
       "type": "page",
        "launchType": "standard"
      }
    ],
    ...
}
```

配置便捷服务的快照。如前文所述，mainAbility 中至少配置 2×2（小尺寸）规格的默认服务卡片，该卡片对应的快照图，需要配置为便捷服务的快照入口，用于在服务中心显示。

配置方式为通过 DevEco Studio 工程向导创建 Project Type 为 Service 的新工程或在已有 Project Type 为 Service 的工程中添加新模块时，勾选 Show in Service Center，则会同步创建一个 2×2 的默认服务卡片模板，同时还会创建该卡片对应的快照图，如图 3-6 所示。

工程创建完成后，会在工程目录下生成快照（EntryCard）目录，如图 3-7 所示。

在该目录下，每个拥有快照（EntryCard）的模块，都会生成一个和模块名相同的文件夹，同时还会默认生成一张 2×2（小尺寸）的快照，即一张 png 格式的图片。开发者可以将其替换为事先设计好的 2×2 快照，样式上应与对应的服务卡片保持一致，将新的快照复制到图 3-7 所示目录下，删除默认图片，新图片命名遵循格式“服务卡片名-2×2.png”。其中“服务卡片名”可以查看 config.json 文件的 forms 数组中的 name 字段。

3.2.3 服务卡片概述

1. 综述

（1）服务卡片（以下简称“卡片”）是 FA 的一种界面展示形式，将 FA 的重要信息或操作前置到卡片，以达到服务直达，减少体验层级的目的。

（2）卡片常用于嵌入其他应用（笔者创作本书期间只支持系统应用）中，作为其界面的一部分显示，并支持拉起页面、发送消息等基础交互功能。卡片使用方负责显示卡片。

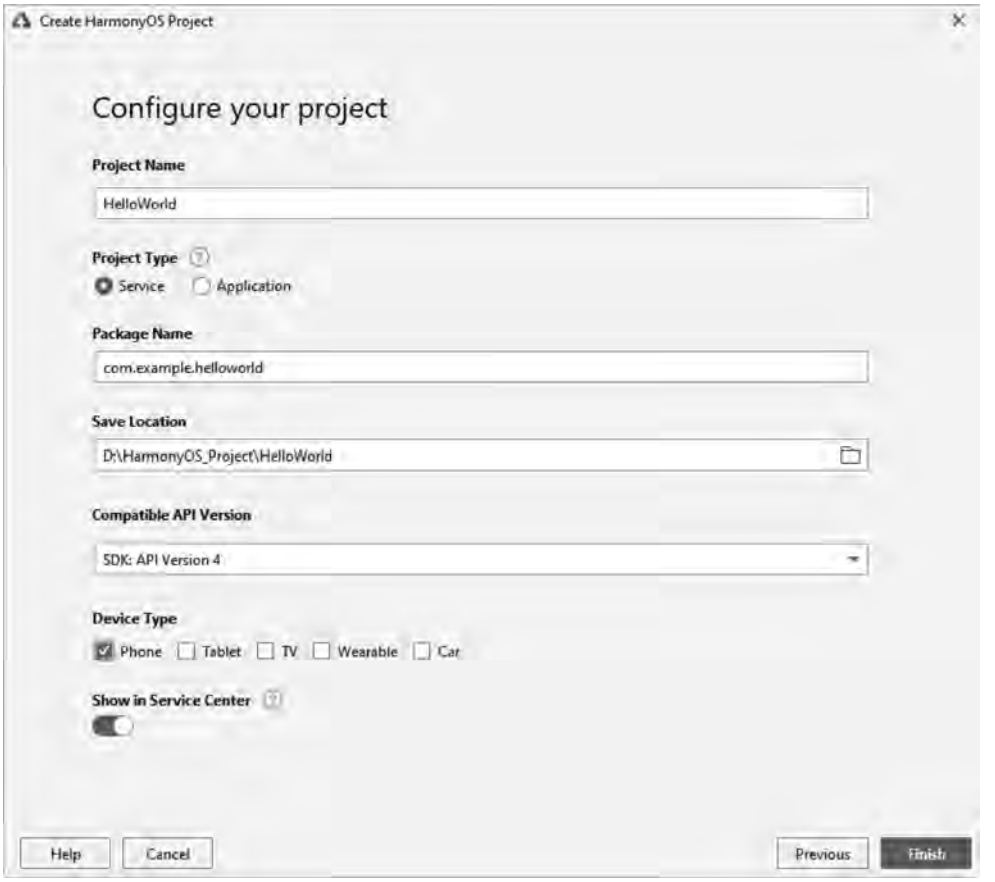

图 3-6　创建工程

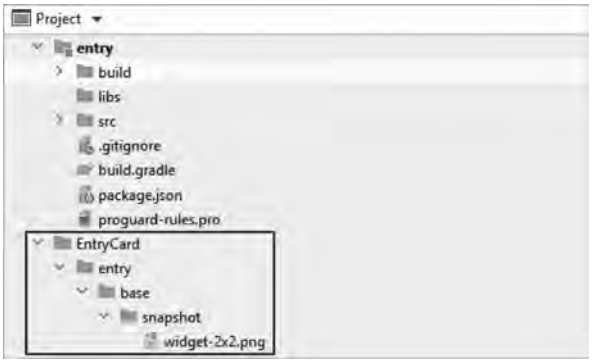

图 3-7　目录结构

2. 基本概念

1）卡片角色

卡片提供方：提供卡片显示内容的 HarmonyOS 应用或原子化服务，控制卡片的显示内容、控件布局及控件单击事件。

卡片使用方：显示卡片内容的宿主应用，控制卡片在宿主中展示的位置。

卡片管理服务：用于管理系统中所添加卡片的常驻代理服务，包括卡片对象的管理与使用，以及卡片周期性刷新等。

需要说明的是卡片使用方和提供方不要求常驻运行，在需要添加/删除/请求更新卡片时，卡片管理服务会拉起卡片提供方获取卡片信息。

2）运作机制

卡片运作机制主要包括卡片管理服务、卡片提供方、卡片使用方 3 部分的运转协同，具体如图 3-8 所示。

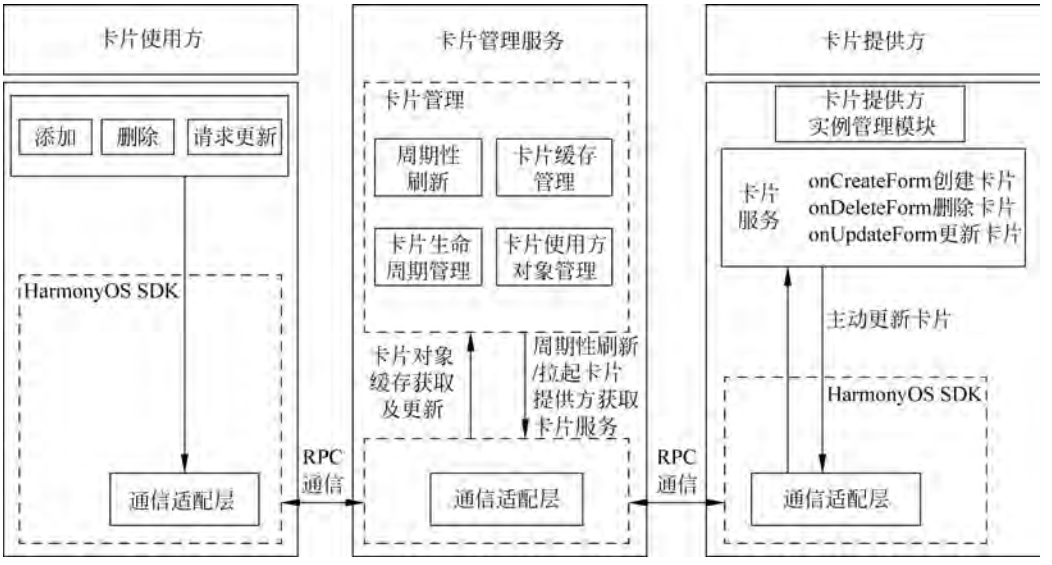

图 3-8　卡片运作机制

卡片管理服务包含以下模块：

周期性刷新，在卡片添加后，根据卡片的刷新策略启动定时任务周期性触发卡片的刷新。卡片缓存管理，在卡片添加到卡片管理服务后，对卡片的视图信息进行缓存，以便下次获取卡片时可以直接返回缓存数据，降低时延。卡片生命周期管理，对于卡片切换到后台或者被遮挡时，暂停卡片的刷新，及卡片的升级/卸载场景下对卡片数据的更新和清理。卡片使用方对象管理，对卡片使用方的 RPC 对象进行管理，用于使用方请求进行校验及对卡片更新后的回调处理。通信适配层，负责与卡片使用方和提供方进行 RPC 通信。

卡片提供方包含以下模块：

卡片服务，由卡片提供方开发者实现，开发者实现 onCreateForm（创建卡片）、onUpdateForm（更新卡片）和 onDeleteForm（删除卡片）等请求，提供相应的卡片服务。卡片提供方实例管理模块，由卡片提供方开发者实现，负责对卡片管理服务分配的卡片实例进行持久化管理。通信适配层，由 HarmonyOS SDK 提供，负责与卡片管理服务通信，用于将卡片的更新数据主动推送到卡片管理服务。

卡片使用方主要包括请求更新、删除、添加及通过 HarmonyOS SDK 与卡片管理服务进行的各项通信适配功能。

3. 服务卡片开发简介

具体的开发场景：开发者仅需作为卡片提供方进行服务卡片内容的开发，卡片使用方和卡片代理服务由系统自动处理。卡片提供方控制卡片实际显示的内容、控件布局及控件单击事件。开发者可以通过集成以下接口来提供卡片服务。

接口说明，HarmonyOS 中的服务卡片为卡片提供方开发者提供以下接口功能，见表 3-2。

表 3-2 卡片提供的接口功能

类名	接 口 名	描 述
Ability	ProviderFormInfo onCreateForm(Intent intent)	卡片提供方接收创建卡片通知接口
	void onUpdateForm(long formId)	卡片提供方接收更新卡片通知接口
	void onDeleteForm(long formId)	卡片提供方接收删除卡片通知接口
	void onTriggerFormEvent(long formId, String message)	卡片提供方处理卡片事件接口（JS 卡片使用）
	Boolean updateForm(long formId, ComponentProvider component)	卡片提供方主动更新卡片（Java 卡片使用）
	boolean updateForm(long formId, FormBindingData formBindingData)	卡片提供方主动更新卡片（JS 卡片使用），仅更新 formBindingData 中携带的信息，卡片中其余信息保持不变
	void onCastTempForm(long formId)	卡片提供方接收临时卡片转常态卡片通知
	void onEventNotify(Map < Long, Integer > formEvents)	卡片提供方收到事件通知，其中 Ability.FORM_VISIBLE 表示卡片可见通知，Ability.FORM_INVISIBLE 表示卡片不可见通知
	FormState onAcquireFormState(Intent intent)	卡片提供方接收查询卡片状态通知接口。默认返回卡片初始状态
ProviderFormInfo	ProviderFormInfo(int resId,Context context)	Java 卡片返回对象构造函数
	ProviderFormInfo()	JS 卡片返回对象构造函数
	void mergeActions(ComponentProvider componentProviderActions)	在提供方侧调用该接口，将开发者在 ComponentProvider 中设置的 actions 配置数据合并到当前对象中
	void setJsBindingData(FormBindingData data)	设置 JS 卡片的内容信息（JS 卡片使用）

其中，onEventNotify 仅系统应用才会回调，其他接口的回调机制如图 3-9 所示。

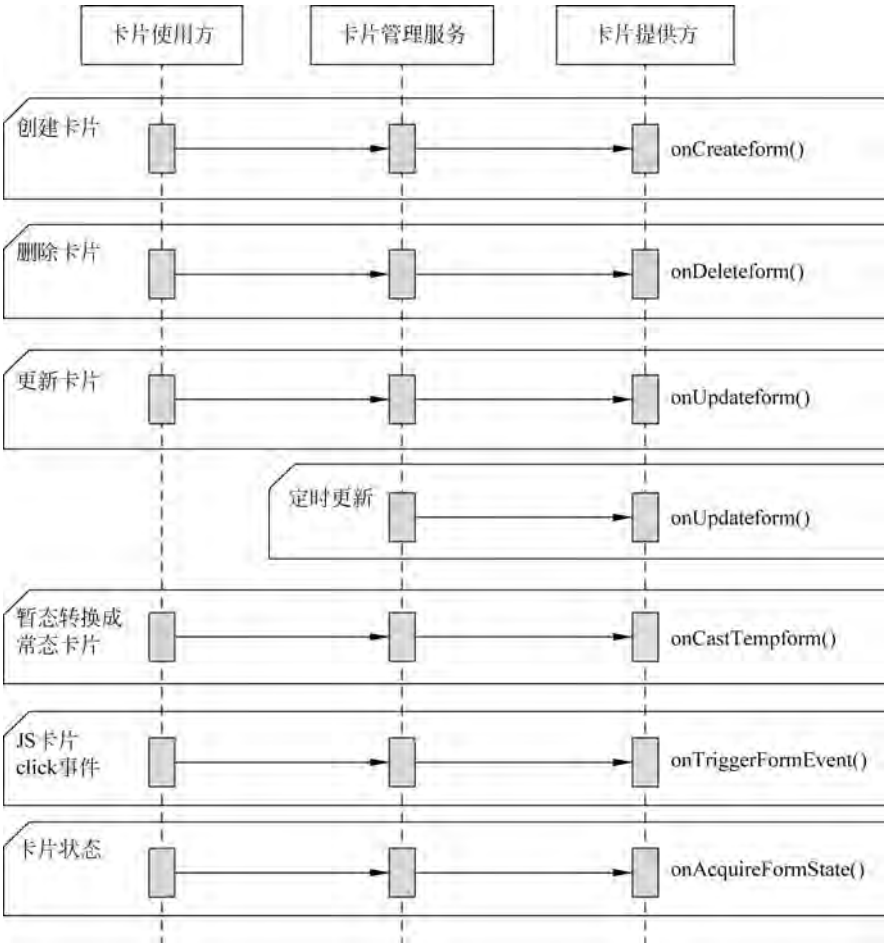

图 3-9　回调机制

需要说明的是卡片管理服务不负责保持卡片的活跃状态，设置了定时更新的除外，当使用方做出相应的请求时，管理服务会拉起提供方并回调相应接口。

4. Java 卡片与 JS 卡片的选型指导

Java 卡片与 JS 卡片的场景差异见表 3-3。支持版本 HarmonyOS 2.0 及以上。

表 3-3　Java 卡片与 JS 卡片的场景差异对比

场　　景	Java 卡片	JS 卡片
实时刷新(类似时钟)	Java 使用 ComponentProvider 做实时刷新代价比较大	JS 可以做到端侧刷新，但是需要定制化组件
开发方式	Java UI 在卡片提供方需要同时对数据和组件进行处理，生成 ComponentProvider 远端渲染	JS 卡片在使用方加载渲染，提供方只要处理数据、组件和逻辑分离

续表

场　　景	Java 卡片	JS 卡片
组件支持	Text、Image、DirectionalLayout、PositionLayout、DependentLayout	div、list、list-item、swiper、stack、image、text、span、progress、button（定制：chart、clock、calendar）
卡片内动效	不支持	暂不开放
阴影模糊　动态适应布局 自定义卡片跳转页面	不支持	不支持

综上所述，JS 卡片比 Java 卡片支持的控件和能力都更丰富。Java 卡片适合作为一个直达入口，没有复杂的页面和事件。JS 卡片适合有复杂界面的卡片。对于同一个 Page ability，在 config. json 文件中最多支持配置 16 张卡片。

3.2.4　JS 服务卡片开发与语法

1. 框架说明

1）文件组织

（1）目录结构：JS 服务卡片（entry/src/main/js/Component）的典型开发目录结构如图 3-10 所示。

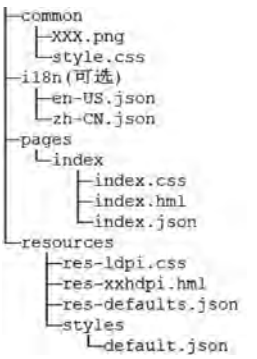

图 3-10　目录结构

目录结构中文件分类包括 3 个类别。. hml 结尾的 HML 模板文件，这个文件用来描述卡片页面的模板布局结构。. css 结尾的 CSS 样式文件，这个文件用于描述页面样式。. json 结尾的 JSON 文件，这个文件用于配置卡片中使用的变量 action 事件。

各个文件夹的作用：pages 目录用于存放卡片模板页面。common 目录用于存放公共资源文件，例如图片资源。resources 目录用于存放资源配置文件，例如多分辨率加载配置文件。i18n 目录用于配置不同语言场景资源内容，例如应用文本词条、图片路径等资源。需要说明的是 i18n 和 resources 是开发时保留的文件夹，不可重命名，resources/styles/default. json 可以配置主题样式。JS 服务卡片不同于 JS 应用，JS 应用使用 JS 文件处理数据逻辑，而卡片则通过卡片提供方应用处理数据并传递给卡片进行显示，卡片和卡片提供方应用间通过 JSON 配置文件约定相应的数据和事件交互接口，所以不包含 JS 应用的 JS 文件。

（2）文件访问规则：应用资源可通过绝对路径或相对路径的方式进行访问，本开发框架中绝对路径以"/"开头，相对路径以". /"或"../"开头。引用代码文件时需使用相对路径，例如：../common/style. css。引用资源文件时推荐使用绝对路径，例如：/common/xxx. png。公共代码文件和资源文件推荐放在 common 下，通过前述规则进行访问。CSS 样式文件中通过 url()函数创建< url >数据类型，如：url(/common/xxx. png)。需要说明的是当代码文件 A 需要引用代码文件 B 时，如果代码文件 A 和文件 B 位于同一目录，则代码文

件 B 引用资源文件时可使用相对路径，也可使用绝对路径；如果代码文件 A 和文件 B 位于不同目录，则代码文件 B 引用资源文件时必须使用绝对路径，因为 Webpack 打包时，代码文件 B 的目录会发生变化。在 JSON 文件中定义的数据为资源文件路径时，需使用绝对路径。

2）配置文件

JS 标签中包含的信息见表 3-4。

表 3-4　JS 标签

标签	类型	默认值	必填	描　　述
name	String	default	是	标识 JS 实例的名字
pages	Array	—	是	路由信息，详见后续说明
window	Object	—	否	窗口信息，详见后续说明
type	String	normal	否	form 卡片，normal 应用

需要说明的是 name、pages、window、type 等标签配置需要在配置文件中的 JS 标签中完成设置。

pages 用于定义卡片页面信息，由卡片页面路径和卡片页面名组成，卡片仅包含一个页面，示例代码如下：

```
"//":"第 3 章/定义卡片页面信息":
{
  …
  "pages": [
    "pages/index/index"      //卡片仅包含一个页面
  ]
  …
}
```

pages 列表中仅包含一个页面。页面文件名不能使用组件名称，例如：text. hml、button. hml 等。

window 用于定义与显示窗口相关的配置。对于卡片尺寸适配问题，有两种配置方法，建议用 autoDesignWidth。

指定卡片 designWidth 150px(2×2)，所有与大小相关的样式(例如 width、font-size)均以 designWidth 和实际卡片宽度的比例进行缩放，例如当 designWidth 为 150 时，如果将 width 设置为 100px，则在卡片实际宽度为 300 物理像素时，width 实际渲染像素为 200 物理像素。将 autoDesignWidth 设置为 true，此时 designWidth 字段将会被忽略，渲染组件和布局时按屏幕密度进行缩放。屏幕逻辑宽度由设备宽度和屏幕密度自动计算得出，在不同设备上可能不同，应使用相对布局来适配多种设备。例如：在 466×466 分辨率，320dpi 的设备上，屏幕密度为 2(以 160dpi 为基准)，1px 等于渲染出 2 物理像素。

组件样式中<length>类型的默认值，按屏幕密度进行计算和绘制，如在屏幕密度为2（以160dpi为基准）的设备上，默认<length>为1px时，设备上实际渲染出2物理像素。autoDesignWidth、designWidth的设置不影响默认值的计算方式和绘制结果，见表3-5。

表3-5　autoDesignWidth、designWidth的说明

属　性	类型	必填	默认值	描　述
designWidth	number	否	150px	页面显示设计时的参考值，实际显示效果基于设备宽度与参考值之间的比例进行缩放
autoDesignWidth	boolean	否	false	页面设计基准宽度是否自动计算，当设为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

JS服务卡片配置文件示例一，代码如下：

```
"//": "第3章/JS服务卡片配置文件实例一"
  …
  "window": {
    "autoDesignWidth": true
  }
  …
}
```

JS服务卡片配置文件实例二，代码如下：

```
"//": "第3章/JS服务卡片配置文件实例二"
{
  "app": {
    "bundleName": "com.demo.player",
    "version": {
        "code": 1,
        "name": "1.0"
    },
    "vendor": "example"
  }
  "module": {
    …
    "js": [
      {
        "name": "myJsForm",
        "pages": [
          "pages/index/index"
        ],
        "window": {
          "autoDesignWidth": true
```

```
        },
        "type": "form"   //可选[normal(默认缺省), form],使能 form 类型
      }
    ],
    "abilities": [
      {
        ...
        "forms": [
          {
            "name": " $ string: form_name",        //卡片名称,用于标识区分卡片
            "isDefault": true,                     //是否为默认卡片,每个 ability 有且只能有
                                                   //一个默认卡片
            "description": " $ string: form_description",     //卡片功能简介,不超过 256 个字符
            "colorMode": "auto",                   //String 类型,取值为 auto、dark、light,标识
                                                   //支持的色调主题
            "supportDimensions":["1 * 2","2 * 2","2 * 4","4 * 4"],
                                                   //卡片外观规格,一个卡片可以有多个规格
            "defaultDimension": "2 * 2",           //缺省展现外观,不可缺省,取值必须在
                                                   //supportDimensions 配置的列表中
            "updateEnabled": true,                 //是否允许定时刷新
            "scheduledUpdateTime": "10:30",        //定点更新,采用 24h 计数,精确到分钟
            "updateDuration": 1,                   //更新频率; 单位为 30min 的倍数
            "type": "JS",                          //Form 类型可选 [Java, JS]
            "JsComponentName": "myJsForm"          //仅选 JS 卡片时需要指定,需要和声明的 JS
                                                   //Component 名字对应
          }
        ],
        "formsEnabled": true
      }
    ]
  }
}
```

2. 语法

1) HML 语法参考

HML(HarmonyOS Markup Language)是一套类 HTML 的标记语言,通过组件和事件构建出页面的内容。页面具备数据绑定、事件绑定、条件渲染和逻辑控制等高级能力。相关页面功能的实现举例说明如下。

(1) HML 语法页面结构,代码如下:

```
<!-- 第 3 章/HML 语法参考 - 页面结构 xxx.hml 代码 -->
<div class = "item - container">
  <text class = "item - title"> Image Show </text >
  <div class = "item - content">
```

```
    <image src="/common/xxx.png" class="image"></image>
  </div>
</div>
```

（2）HML 语法数据绑定及相关说明，代码如下：

```
<!-- 第 3 章/HML 语法参考 - 数据绑定 xxx.hml 代码 -->
<div class="item-container">
  <text>{{content}} </text>              <!-- 输出：Hello World!-->
  <text>{{key1}} {{key2}}</text>         <!-- 输出：Hello World6 +-->
  <text>key1 {{key1}}</text>             <!-- 输出：key1 Hello 6 +-->
  <text>{{flag1 && flag2}}</text>        <!-- 输出：false 6 +-->
  <text>{{flag1 || flag2}}</text>        <!-- 输出：true 6 +-->
  <text>{{!flag1}}</text>                <!-- 输出：false 6 +-->
</div>
```

HML 语法中 xxx.json 配置文件说明，代码如下：

```
"//": "第 3 章/HML 语法参考 - 配置文件说明 xxx.json 代码"
{
  "data": {
    "content": "Hello World!",
    "key1": "Hello",
    "key2": "World",
    "flag1": true,
    "flag2": false
  }
}
```

需要说明是 key 值支持对象操作符和数组操作符，如{{key.value}}、{{key[0]}}。从 API Version 6 开始支持字符串拼接、逻辑运算和三元表达式。

HML 语法字符串拼接说明，伪代码如下：

```
//第 3 章/HML 字符串拼接
支持变量跟变量：{{key1}}{{key2}}等
支持常量跟变量："my name is {{name}}, i am from {{city}}." "key1 {{key1}}"
逻辑运算：
与：{{flag1 && flag2}}(仅支持两个 boolean 变量间的与运算)
或：{{flag1 || flag2}} (仅支持两个 boolean 变量间的或运算)
非：{{!flag1}} (仅支持 boolean 变量的非运算)
三元表达式
{{flag ?key1 : key2}}(flag 为 boolean 变量，key1 和 key2 可以是变量，也可以是常量)
```

注意，当非 boolean 类型值进行 bool 运算时默认值为 false，以上所有变量解析与运算

解析均不支持嵌套。

(3) 事件绑定：卡片仅支持 click 通用事件，事件的定义只能采用直接命令式，事件定义必须包含 action 字段，用以说明事件类型。卡片支持两种事件类型，即跳转事件(router)和消息事件(message)。跳转事件可以跳转到卡片提供方的 Z 侧应用；消息事件可以将开发者自定义信息传递给卡片提供方。事件参数支持变量，变量以"{{}}"修饰。

跳转事件中若定义了 params 字段，则在被拉起应用的 onStart 的 intent 中，可用 params 作为 key 将跳转事件定义的 params 字段的值取为(deprecated)7+，见表 3-6 和表 3-7。

表 3-6 跳转事件格式

选 择 器	样例	默认值	样 例 描 述
action	string	router	表示事件类型
abilityName(deprecated)7+	string	—	跳转 ability 名
params(deprecated)7+	Object	—	跳转应用携带的额外参数
want7+	Object	—	跳转目标应用的信息，参考[want 格式表]

表 3-7 want 格式 7+

选 择 器	样例	默认值	样 例 描 述
bundleName7+	string	—	跳转 bundle 名
abilityName7+	string	—	跳转 Ability 名
action7+	string	—	动作，用于指定 Intent 的操作行为
uri7+	string	—	用于指定动作要操纵的数据路径
type7+	string	—	数据类型，用于指定 Data 类型的定义
flag7+	number	—	标志位，用于指定 Intent 的运行模式(启动标志)
entities7+	Array	—	类别，用于指定 Intent 的操作类别
parameters7+	Object	—	自定义参数，其中内容将会以键-值对的形式传递到目标应用的 onStart 的 intent 中

HML 语法事件绑定，xxx.json 配置文件说明，代码如下：

```
"//": "第 3 章/HML 语法参考 - 事件绑定 xxx.json 配置文件说明代码"
{
  "data": {
    "mainAbility": "xxx.xxx.xxx"
  },
  "actions": {
    "routerEventName1": {
      "action": "router",
      "want": {
       "bundleName": "com.example.myapplication",
```

```
        "abilityName": "com.example.myapplication.MainAbility"
      }
    },
    "routerEventName2": {
      "action": "router",
      "want": {
        "action": "xxx.intent.action.DIAL",
        "uri": "tel:12345678"
      }
    }
  }
}
```

(4) 消息事件格式,见表 3-8。

表 3-8　消息事件格式

选　择　器	样　　例	默　认　值	样 例 描 述
action	string	message	表示事件类型
params	Object	-	跳转应用携带的额外参数

HML 语法消息事件格式,xxx.json 配置文件说明,代码如下:

```
"//": "第 3 章/HML 语法消息事件格式":
{
 "actions": {
   "activeEvent": {
     "action": "message",
     "params": {}
   }
 }
}
```

HML 语法绑定路由事件和消息事件,代码如下:

```
<!-- 第 3 章/HML 语法参考 - 绑定路由事件和消息事件 xxx.hml 代码 -->
<div>
  <!-- 正常格式 -->
  <div onclick = "activeEvent"></div>
  <!-- 缩写 -->
  <div @click = "activeEvent"></div>
</div>
```

(5) HML 语法列表渲染说明,代码如下:

```
<!-- 第 3 章/HML 语法参考 - 列表渲染说明 xxx.hml 代码 -->
<div class = "array - container">
```

```
    <!-- div列表渲染 -->
    <!-- 默认$item代表数组中的元素，$idx代表数组中的元素索引 -->
    <div for="{{array}}" tid="id">
      <text>{{$item.name}}</text>
    </div>
    <!-- 自定义元素变量名称 -->
    <div for="{{value in array}}" tid="id">
      <text>{{value.name}}</text>
    </div>
    <!-- 自定义元素变量、索引名称 -->
    <div for="{{(index, value) in array}}" tid="id">
      <text>{{value.name}}</text>
    </div>
  </div>
  xxx.json:
  {
    "data": {
      "array": [
        {"id": 1, "name": "jack", "age": 18},
        {"id": 2, "name": "tony", "age": 18}
      ]
    }
  }
```

tid属性主要用来加速for循环的重渲染，旨在列表中的数据有变更时，提高重新渲染的效率。tid属性还用来指定数组中每个元素的唯一标识，如果未指定，则数组中每个元素的索引为该元素的唯一id。例如上述tid=id表示数组中的每个元素的id属性为该元素的唯一标识。for循环支持的写法，伪代码如下：

```
for="array": 其中array为数组对象,array的元素变量默认为$item
for="v in array": 其中v为自定义的元素变量,元素索引默认为$idx
for="(i, v) in array": 其中元素索引为i,元素变量为v,遍历数组对象array
```

需说明的是数组中的每个元素必须存在tid指定的数据属性，否则运行时可能会导致异常。数组中被tid指定的属性要保证唯一性，如果不是，则会造成性能损耗。例如，示例中只有id和name可以作为tid字段，因为它们属于唯一字段。tid不支持表达式。不支持for嵌套使用。

for对应的变量数组，当前要求数组中的object是相同类型，不支持多种object类型混合写在一个数组中。

(6) 条件渲染：条件渲染分为两种，即if/elif/else和show。

当使用if/elif/else写法时，节点必须是兄弟节点，否则编译无法通过。示例代码如下：

```
<!-- 第 3 章/HML 语法参考 - 条件渲染(if/elif/else)xxx.hml 代码 -->
<div>
  <text if = "{{show}}"> Hello - TV </text>
  <text elif = "{{display}}"> Hello - Wearable </text>
  <text else> Hello - World </text>
</div>
xxx.json:
{
  "data": {
    "show": false,
    "display": true
  }
}
```

当 show 为真时，节点正常渲染；当 show 为假时，节点不渲染，效果等同 display 样式为 none，示例代码如下：

```
<!-- 第 3 章/HML 语法参考 - 条件渲染(show)xxx.hml 代码 -->
<text show = "{{visible}}"> Hello World </text>
xxx.json:
{
  "data": {
    "visible": false
  }
}
```

(7) 逻辑控制块：<block>控制块使循环渲染和条件渲染变得更加灵活。block 在构建时不会被当作真实的节点编译。

需说明的是 block 标签只支持 if 属性，示例代码如下：

```
<!-- 第 3 章/HML 语法参考 - 逻辑控制块 xxx.hml 代码 -->
<div>
  <block if = "{{show}}">
    <text> Hello </text>
    <text> World </text>
  </block>
</div>
xxx.json:
{
  "data": {
    "show": true
  }
}
```

2）CSS 语法参考

CSS 是描述 HML 页面结构的样式语言。所有组件均有系统默认样式，也可在页面 CSS 样式文件中对组件、页面自定义不同的样式。

（1）尺寸单位：逻辑像素 px，书中以< length >表示。默认卡片具有的逻辑宽度为 150px，实际显示时会将页面布局缩放至屏幕的实际宽度，如 100px 在宽度为 300 的卡片上，实际渲染为 200 物理像素（从 150px 向 300 物理像素渲染，所有尺寸放大 2 倍）。

额外将 autoDesignWidth 配置为 true 时，逻辑像素 px 将按照屏幕密度进行缩放，如 100px 在屏幕密度为 3 的设备上，实际渲染为 300 物理像素。当应用需要适配多种设备时，建议采用此方法。

百分比，书中以< percentage >表示，表示该组件占父组件尺寸的百分比，如将组件的 width 设置为 50%，代表其宽度为父组件的 50%。

（2）样式导入：为了模块化管理和代码复用，CSS 样式文件支持用@import 语句导入 CSS 文件。

（3）声明样式：每个页面目录下存在一个与布局 HML 文件同名的 CSS 文件，用来描述该 HML 页面中组件的样式，决定组件应该如何显示。

第一，内部样式，支持使用 style、class 属性来控制组件的样式，示例代码如下：

```
<!-- 第 3 章/CSS 语法参考 - 内部样式 xxx.HML 代码 -->
<div class = "container">
  <text style = "color: red"> Hello World </text>
</div>
/* index.css */
.container {
  justify - content: center;
}
```

第二，文件导入，合并外部样式文件。例如，在 common 目录中定义样式文件 style.css，并在 index.css 文件中进行导入，代码如下：

```
/* 第 3 章/CSS 语法参考 - 外部定义样式 style.css 文件代码 */
.title {
  font - size: 50px;
}
/* 第 3 章/CSS 语法参考 - 导入外部定义样式 index.css 文件代码 */
@import '../../common/style.css';  //导入外部样式
.container {
  justify - content: center;
}
```

（4）选择器：CSS选择器用于选择需要添加样式的元素，支持的选择器见表3-9。

表3-9　CSS选择器的样式元素

选　择　器	样　　例	样 例 描 述
.class	.container	用于选择class=container的组件
#id	#titleId	用于选择id=titleId的组件

示例代码如下：

```
/* 第3章/CSS语法参考 - 选择器 */
<!-- 页面布局 xxx.hml -->
<div id = "containerId" class = "container">
   <text id = "titleId" class = "title">标题</text>
   <div class = "content">
      <text id = "contentId">内容</text>
   </div>
</div>
/* 页面样式 xxx.css */
/* 对 class = "title"的组件设置样式 */
.title {
  font - size: 30px;
}
/* 对 id = "contentId"的组件设置样式 */
#contentId {
  font - size: 20px;
}
```

（5）选择器的优先级：选择器的优先级的计算规则与w3c规则保持一致（只支持内联样式、id、class），其中内联样式为在元素style属性中声明的样式。当多条选择器声明匹配到同一元素时，各类选择器的优先级由高到低的顺序为内联样式 > id > class。

（6）样式预编译：预编译提供了利用特有语法生成CSS的程序，可以提供变量、运算等功能，令开发者更便捷地定义组件样式，目前支持less、sass和scss的预编译。当使用样式预编译时，需要将原CSS文件的后缀改为less、sass或scss，如将index.css改为index.less、index.sass或index.scss。

当前文件使用样式预编译，例如将原index.css改为index.less，代码如下：

```
/* 第3章/CSS语法参考 - 样式预编译 index.less */
/* 定义变量 */
@colorBackground: #000000;
.container {
  background - color: @colorBackground;      /* 使用当前less文件中定义的变量 */
}
```

引用预编译文件，例如 common 中存在 style.scss 文件，将原 index.css 改为 index.scss，并引入 style.scss，代码如下：

```
/*第3章/CSS语法参考-引入预编译文件style.scss*/
/*定义变量*/
$colorBackground: #000000;
在index.scss中引用:
/*index.scss*/
/*引入外部SCSS文件*/
@import '../../common/style.scss';
.container {
 background-color: $colorBackground;      /*使用style.scss中定义的变量*/
}
```

需说明的是引用的预编译文件建议放在 common 目录进行管理。

3）配置数据和事件

卡片使用 JSON 文件配置所用的变量和事件，变量的声明在 data 字段下，事件的声明在 actions 字段下，示例代码如下：

```
"//":"第3章/卡片配置数据和事件"
{
   "data": {
       "temperature": "35°C",
       "city": "hangzhou"
   },
   "actions": {
       "routerEventName": {
           "action": "router",
           "abilityName": "com.example.myapplication.FormAbility",
           "params": {
               "message": "weather",
               "temperature": "{{temperature}}"
           }
       },
       "messageEventName": {
           "action": "message",
           "params": {
               "message": "weather update"
           }
       }
   }
}
```

3. 资源访问

1）访问 JS 模块资源

卡片工程可以访问的资源包括 JS 模块的 resources 资源、应用 resources 资源 6+（所有

JS模块共享)和系统预置资源6+。

(1) 资源限定词：资源限定词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，包括深色模式、屏幕密度等维度，限定词之间通过中画线(-)连接。开发者在resources目录下创建限定词文件时，需要掌握限定词文件的命名要求以及限定词文件与设备状态的匹配规则。

(2) 资源限定词的命名要求：限定词的组合顺序为深色模式和屏幕密度。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，顺序不可颠倒。限定词之间均采用中画线(-)连接，例如res-dark-ldpi.json。每类限定词的取值必须符合表3-10的条件，否则，将无法匹配目录中的资源文件，限定词对大小写敏感。resources资源文件的资源限定词有前缀res，例如res-ldpi.json。resources资源文件的默认资源限定文件为res-defaults.json。资源限定文件中不支持使用枚举格式的颜色设置资源，具体见表3-10。

表3-10 资源限定词

类 型	含义与取值说明
深色模式	表示设备的深色模式，取值如dark
屏幕密度	表示设备的屏幕密度(单位为dpi)，取值：ldpi表示低密度屏幕(～120dpi)(0.75基准密度)；mdpi表示中密度屏幕(～160dpi)(基准密度)；hdpi表示高密度屏幕(～240dpi)(1.5基准密度)；xhdpi表示超高密度屏幕(～320dpi)(2.0基准密度)；xxhdpi表示超超高密度屏幕(～480dpi)(3.0基准密度)；xxxhdpi表示超超超高密度屏幕(～640dpi)(4.0基准密度)

需要说明的是如果当前设备的DPI不完全匹配定义的DPI，则将选取更接近当前设备DPI的资源文件。例如当前设备为2.7×基准密度，会选择res-xxhdpi.json文件中定义的资源；开发者还可以定义一个res-defaults.json资源文件，用于当对应密度资源文件中没有对应的资源词条时，应用将尝试在res-defaults.json文件中匹配对应的资源词条，如果仍未查找到对应词条，则图片加载失败。

(3) 限定词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深色模式 > 屏幕密度。在资源限定词目录均未匹配的情况下，匹配默认资源限定文件。

如果限定词目录中包含资源限定词，则对应限定词的取值必须与当前的设备状态完全一致，这样该目录才能参与设备的资源匹配。例如资源限定文件res-hdpi.json与当前设备密度xhdpi无法匹配。

(4) 引用JS模块内的resources资源：在应用开发的HML和JS文件中使用$r的语法，可以对JS模块内的resources目录下的json资源进行格式化，获取相应的资源内容。具体见表3-11。

表 3-11 引用 JS 模块内的 resources 资源

属性	类 型	描 述

$r	(key: string) => string	获取资源限定下具体的资源内容。例如：this. $r('strings. hello')
		参数说明：key 定义在资源限定文件中的键值，如 strings. hello
		res-defaults. json 示例：
		{
		strings: {
		hello: 'hello world'
		}
		}

引用 JS 模块内 resources 资源，示例代码如下：

```
//第 3 章/引用 JS 模块内 resources 资源
resources/res - dark. json:
{
    "image": {
        "clockFace": "common/dark_face. png"
    },
    "colors": {
    "background": "#000000"
    }
}
resources/res - defaults. json:
{
    "image": {
        "clockFace": "common/face. png"
    },
    "colors": {
    "background": "#ffffff"
    }
}
<!-- xxx. hml -->
<div style = "background - color: {{ $r('colors. background') }}">
    <image src = "{{ $r('image. clockFace') }}"></image>
</div>
```

需要说明的是资源限定文件中不支持颜色枚举格式。

2）访问应用资源

从 API Version 6 开始，在 HML/CSS/JSON 文件中，可以引用应用资源，包括颜色、圆角和图片类型的资源。

应用资源由开发者在 resources 目录中定义，目前仅支持使用在 color. json 文件中自定义的颜色资源、在 float. json 文件中自定义的圆角资源及在 media 目录中存放的图片资源。

resources 目录的基础结构如下所示，同一个资源，可以在 base 子目录和 dark 子目录各定义一个值。浅色模式下用 base 目录中定义的值，深色模式下用 dark 目录下定义的值。若某资源仅在 base 目录中有定义，则其在深浅色模式下的表现相同。具体示例代码如下：

```
//第 3 章/resources 目录的基础结构
├── java
├── js
└── resources  -> 与 java、js 目录同级的 resources 目录
    ├── base  -> 定义浅色模式下的颜色、圆角或图片
    |    ├── element
    |    |    ├── color.json
    |    |    └── float.json
    |    └── media
    |         └── my_background_image.png
    └── dark  -> 定义深色模式下的颜色、圆角或图片(如果未定义,则深色模式下继续使用 base 目录中的相关定义)
         ├── element
         |    ├── color.json
         |    └── float.json
         └── media
              └── my_background_image.png
```

引用 JS 模块内的 resources 资源，访问应用资源，color. json 文件的格式说明，代码如下：

```
"//": "第 3 章/访问应用资源 - color.json 文件的格式说明"
{
    "color": [
        {
            "name": "my_background_color",
            "value": "#ffff0000"
        },
        {
           "name": "my_foreground_color",
            "value": "#ff0000ff"
        }
    ]
}
```

引用 JS 模块内的 resources 资源，访问应用资源，float. json 文件的格式说明，代码如下：

```
"//": "第 3 章/访问应用资源 - float.json 文件的格式说明"
{
```

```
    "float":[
        {
            "name":"my_radius",
            "value":"28.0vp"
        },
        {
            "name":"my_radius_xs",
            "value":"4.0vp"
        }
    ]
}
```

在卡片工程的 CSS 文件中，通过@app.type.resource_id 的形式引用应用资源。根据引用资源类型的不同，type 可以取 color(颜色)、float(圆角)和 media(图片)。resource_id 代表应用资源 id，即 color.json 或 float.json 文件中的 name 字段，或者 media 目录中的图片文件的名称(不包含图片类型后缀)，具体示例代码如下：

```
//第 3 章/通过@app.type.resource_id 的形式引用应用资源
.divA {
    background-color: "@app.color.my_background_color";
    border-radius: "@app.float.my_radius";
}
.divB {
    background-image: "@app.media.my_background_image";
}
```

在 HML 文件中，通过{{ $r('app.type.resource_id')}}的形式引用应用资源，各个字段的含义与 CSS 文件相同，具体示例代码如下：

```
<div style="background-color:{{ $r('app.color.my_background_color')}};"></div>
<div style="border-radius:{{ $r('app.float.my_radius')}};"></div>
<div style="background-image:{{ $r('app.media.my_background_image')}};"></div>
```

在 JSON 文件中，通过 this.$r('app.type.resource_id')的形式引用应用资源，各个字段的含义与 CSS 文件相同，具体示例代码如下：

```
//第 3 章/通过 this.$r('app.type.resource_id')的形式引用应用资源
{
    "data":{
        "myColor": "this.$r('app.color.my_background_color')",
        "myRadius": "this.$r('app.float.my_radius')",
        "myImage":"this.$r('app.media.my_background_image')"
    }
}
```

3）访问系统资源

在 HML/CSS/JSON 文件中，可以引用系统预置资源，包括颜色、圆角和图片类型的资源。说明，从 API Version 6 开始支持。在卡片工程的 CSS 文件中，通过@sys.type.resource_id 的形式引用系统资源。根据引用资源类型的不同，type 可以取 color（颜色）、float（圆角）和 media（图片）。resource_id 代表系统资源 id，系统资源预置在系统中，具体示例代码如下：

```
//第 3 章/访问系统资源 - 系统资源预置
.divA {
    background-color: "@sys.color.fa_background";
    border-radius: "@sys.float.fa_corner_radius_card";
}
.divB {
    background-image: "@sys.media.fa_card_background";
}
```

在 HML 文件中，通过{{ $r('sys.type.resource_id')}}的形式引用系统资源，各个字段的含义与 CSS 文件相同，具体示例代码如下：

```
<div style="background-color:{{ $r('sys.color.fa_background')}};"></div>
<div style="border-radius:{{ $r('sys.float.fa_corner_radius_card')}};"></div>
<div style="background-image:{{ $r('sys.media.fa_card_background')}};"></div>
```

在 JSON 文件中，通过 this.$r('sys.type.resource_id')的形式引用系统资源，各个字段的含义与 CSS 文件相同，具体示例代码如下：

```
//第 3 章/通过 this.$r('sys.type.resource_id')的形式引用系统资源
{
    "data":{
        "sysColor": "this.$r('sys.color.fa_background')",
        "sysRadius": "this.$r('sys.float.fa_corner_radius_card')",
        "sysImage":"this.$r('sys.media.fa_card_background')"
    }
}
```

4. 多语言支持

基于开发框架的应用会覆盖多个国家和地区，开发框架支持多语言能力后，应用开发者无须开发多个不同语言的版本就可以同时支持多种语言的切换，为项目维护带来便利。开发者仅需要通过定义资源文件和引用资源两个步骤，就可以使用开发框架的多语言能力。

1）定义资源文件

资源文件用于存放应用在多种语言场景下的资源内容，开发框架使用 JSON 文件保存资源。

在文件组织中指定的 i18n 文件夹内放置每个语言地区下的资源定义文件即可，资源文件命名为“语言-地区.json”格式，例如英文（美国）的资源文件命名为 en-US.json。当开发框架无法在应用中找到系统语言的资源文件时，默认使用 en-US.json 文件中的资源内容。

由于不同语言针对单复数有不同的匹配规则，在资源文件中可使用 zero、one、two、few、many、other 定义不同单复数场景下的词条内容。例如，中文不区分单复数，仅存在 other 场景；英文存在 one、other 场景；阿拉伯语存在上述 6 种场景。

以 en-US.json 和 ar-AE.json 为例，演示资源文件的格式，代码如下：

```
"//":"第 3 章/定义资源文件示例"
{
    "strings": {
        "hello": "Hello world!",
        "symbol": "@ # $ % ^ & * ()_ +-= {}[]\\|:;\"'<>,./?",
        "plurals": {
            "one": "one person",
            "other": "other people"
        }
    },

    "files": {
        "image": "image/en_picture.PNG"
    }
}
{
    "strings": {
        "plurals": {
            "zero": "لا أحد",
            "one": "واحده",
            "two": "اثنان",
            "few": "اشخاص ستة",
            "many": "شخص خمسون",
            "other": "شخص مائة"
        }
    }
}
```

2）引用资源

在应用开发的页面中使用多语言的语法，包含简单格式化和单复数格式化两种，这两种方法都可以在 HML 或 JS 中使用。

（1）简单格式化方法。

在应用中使用 $t 方法引用资源，$t 既可以在 HML 中使用，也可以在 JS 中使用。系统将根据当前语言环境和指定的资源路径（通过 $t 的 path 参数设置），显示对应语言的资源文件中的内容，具体见表 3-12 和表 3-13。

表 3-12 简单格式化

属性	类型	参数	必填	描 述
$t	Function	参见表 3-13	是	根据系统语言完成简单的替换：this. $t('strings. hello')

表 3-13 $t 参数说明

参数	类型	必填	描 述
path	string	是	资源路径

多语言支持-引用资源-简单格式化，示例代码如下：

```
<!-- xxx.hml -->
<div>
  <text>{{ $t('strings.hello') }}</text>
  <image src="{{ $t('files.image') }}" class="image"></image>
</div>
```

(2) 单复数格式化方法，具体见表 3-14 和表 3-15。

表 3-14 单复数格式化

属性	类型	参数	必填	描 述
$t	Function	参见表 3-15	是	根据系统语言完成单复数替换：this. $tc('strings. plurals') 说明：定义资源的内容，通过 JSON 格式的 key 区分，key 为 zero、one、two、few、many、other

表 3-15 $tc 参数说明

参数	类型	必填	描 述
path	string	是	资源路径
count	number	是	要表达的值

多语言支持-引用资源-单复数格式化，示例代码如下：

```
<!-- 第 3 章/引用资源 - 单复数格式化方法 xxx.hml 代码 -->
<div>
  <!-- 传递数值为 0 时："0 people" 阿拉伯语中此处匹配 key 为 zero 的词条 -->
  <text>{{ $tc('strings.plurals', 0) }}</text>
  <!-- 传递数值为 1 时："one person" 阿拉伯语中此处匹配 key 为 one 的词条 -->
  <text>{{ $tc('strings.plurals', 1) }}</text>
  <!-- 传递数值为 2 时："2 people" 阿拉伯语中此处匹配 key 为 two 的词条 -->
  <text>{{ $tc('strings.plurals', 2) }}</text>
```

```
    <!-- 传递数值为 6 时："6 people" 阿拉伯语中此处匹配 key 为 few 的词条 -->
    <text>{{ $tc('strings.plurals', 6) }}</text>
    <!-- 传递数值为 50 时："50 people" 阿拉伯语中此处匹配 key 为 many 的词条 -->
    <text>{{ $tc('strings.plurals', 50) }}</text>
    <!-- 传递数值为 100 时："100 people" 阿拉伯语中此处匹配 key 为 other 的词条 -->
    <text>{{ $tc('strings.plurals', 100) }}</text>
</div>
```

5. 低版本兼容

卡片特性不断增加，使用了新特性的卡片，在不支持这些新特性的旧系统上可能显示异常。可以在卡片工程中指定最小 SDK 版本，防止使用新特性的卡片被推送后安装在旧系统上。也可以参考本章节的内容，在卡片开发阶段做前向兼容适配。

说明：低版本兼容能力从 API Version 6 开始支持。

开发者可以通过 JSON 配置文件配置前向兼容能力。该文件提供了 apiVersion 属性，用于兼容版本，该字段和卡片配置文件的数据字段 data、事件字段 actions 同级。在 apiVersion 标签下定义的内容会基于当前运行版本信息，覆盖原始的 data 标签内容。

示例说明，假设 JS 服务卡片框架从 API Version 6 开始支持引用系统内置资源颜色，从 API Version 7 开始支持 slider 组件（仅用于举例，不代表实际情况），则可以按照以下方式，做前向兼容，代码如下：

```
<!-- 第 3 章/引用资源 - 低版本兼容 xxx.hml 代码 -->
<div style="background-color: {{myBackgroundColor}}">
    <text>hello world</text>
    <slider if="{{canUseSlider}}" min="0" max="100"></slider>
</div>
xxx.json 配置文件：
{
  "data": {
    "myBackgroundColor": "#87ceeb",
    "canUseSlider": "false"
  },
  "apiVersion": {
    "6": {
      "myBackgroundColor": "@sys.color.fa_background"
    },
    "7": {
      "canUseSlider": "true"
    }
  }
}
```

JS服务卡片开发框架会根据应用中的配置及当前系统运行版本，选取最合适的数据。假设系统运行版本在5及以下，则实际解析的myBackgroundColor值为#87ceeb，canUseSlider值为false；假设系统运行版本为6，则实际解析的myBackgroundColor值为@sys.color.fa_background，canUseSlider值为false；假设系统运行版本为7及以上，则实际解析的myBackgroundColor值为@sys.color.fa_background，canUseSlider值为true。

6. 设置主题样式

卡片支持修改自定义主题字段。用app_background设置卡片背景颜色。

如将相关卡片背景色设置为透明，代码如下：

```
{
  "styles": {
    "app_background": "#00000000"
  }
}
```

3.3 Ability框架

本节主要阐述HarmonyOS应用、原子化服务与服务卡片开发所具备的抽象能力Ability，包括三大分类：运行原理、基本开发逻辑及它们之间的信息传递方式。

3.3.1 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。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，后文中将基于Page模板、Service模板、Data模板实现的Ability分别简称为Page、Service、Data。

Ability配置示例代码如下：

```
"//": "第 3 章/Ability 类型设置"
{
    "module": {
        ...
        "abilities": [
            {
                ...
                "type": "page"
                ...
            }
        ]
        ...
    }
    ...
}
```

3.3.2 Page Ability 基本概念

1. 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具有高度相关性。例如,邮件功能可以通过一个 Page 实现,其中包含两个 AbilitySlice。一个 AbilitySlice 用于展示邮件列表,另一个 AbilitySlice 用于展示邮件详情。Page 和 AbilitySlice 的关系如图 3-11 所示。

图 3-11 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,打车应用提供了联系司机的业务功能入口,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 路由可配置。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,则可以通过 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,同 Page 间导航方法不一样。

setMainRoute()方法与 addActionRoute()方法的使用,示例代码如下:

```
//第 3 章/Page 和 Ability - 路由配置 setMainRoute()方法与 addActionRoute()方法的使用
public class MyAbility extends Ability {
    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
        //设置主路由
        setMainRoute(MainSlice.class.getName());

        //设置操作路由
        addActionRoute("action.pay", PaySlice.class.getName());
        addActionRoute("action.scan", ScanSlice.class.getName());
    }
}
```

为在 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,代码如下:

```
"//": "第 3 章/注册 addActionRoute()方法中使用的动作命名"
{
    "module": {
       "abilities": [
           {
               "skills":[
                   {
                       "actions":[
                           "action.pay",
                           "action.scan"
                       ]
                   }
               ]
               ...
           }
       ]
       ...
   }
    ...
}
```

2. Page Ability 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,

例如释放资源，这有助于提升应用的性能和稳健性。

Page 生命周期的不同状态转换及其对应的回调如图 3-12 所示。

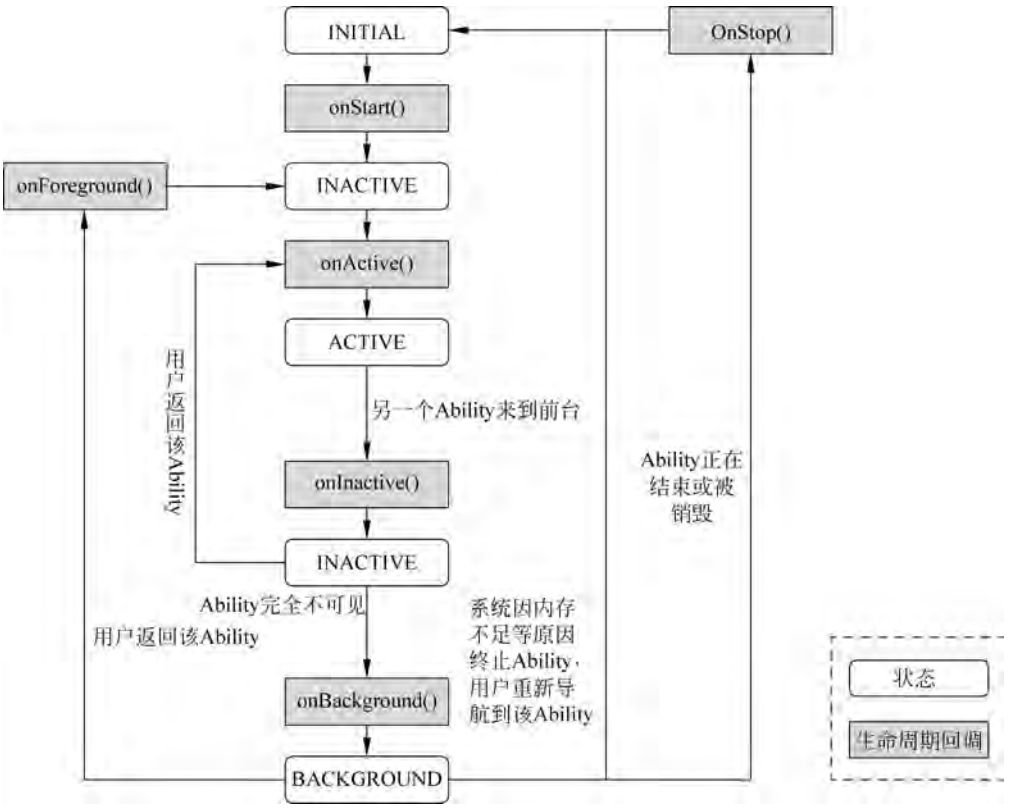

图 3-12　Page 生命周期

具体说明如下：

（1）INACTIVE 状态是一种短暂存在的状态，可理解为“激活中”。

（2）onStart()：当系统首次创建 Page 实例时，触发该回调。对于一个 Page 实例，该回调在其整个生命周期中仅触发一次，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，代码如下：

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

（3）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，除非某类事件发生而导致 Page 失去焦点，例如用户单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。

(4) 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。

(5) 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。

(6) onForeground()：处于 BACKGROUND 状态的 Page 仍然驻留在内存中，当重新回到前台时(例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。

(7) 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方面。一是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。二是用户行为触发 Page 的 terminateAbility()方法调用，例如使用应用的退出功能。三是配置变更导致系统暂时销毁 Page 并重建。四是系统出于资源管理的目的，自动触发对处于 BACKGROUND 状态 Page 的销毁。

3. AbilitySlice 生命周期

AbilitySlice 作为 Page 的组成单元，其生命周期依托于其所属 Page 的生命周期。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 中的不同 AbilitySlice 之间进行导航时，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，示例代码如下：

```
    //第 3 章/重写 AbilitySlice 的 onStart()回调,setUIContent()方法的使用
    @Override
    protected void onStart(Intent intent) {
        super.onStart(intent);

         setUIContent(ResourceTable.Layout_main_layout);
    }
```

AbilitySlice 实例的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，当通过导航启动某个 AbilitySlice 时，由系统负责实例化，但是在同一个 Page 中的不同 AbilitySlice 间进行导航时则由应用负责实例化。

4. 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，并且即将导航到 BarAbilitySlice 时，在此期间的生命周期状态变化顺序如下：

FooAbilitySlice 从 ACTIVE 状态变为 INACTIVE 状态。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）。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，代码如下：

```
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 将被联动销毁，而不仅是处于前台的 AbilitySlice。

5. AbilitySlice 间导航

1）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，可以通过 present()方法实现导航。如下代码片段展示如何通过单击按钮导航到其他 AbilitySlice 的方法，代码如下：

```
//第 3 章/AbilitySlice 间导航案例
@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，具体的代码如下：

```
//第 3 章/presentForResult()方法的使用
int requestCode = positiveInteger; //Any positive integer.

@Override
protected void onStart(Intent intent) {

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

}

@Override
protected void onResult(int requestCode, Intent resultIntent) {
    if (requestCode == positiveInteger) {
        //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，如果指定的 AbilitySlice 实例已经在栈中存在，则栈中位于此实例之上的 AbilitySlice 均会出栈并终止其生命周期。在前面的示例代码中，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属于同一个类，所以不会导致任何 AbilitySlice 出栈。

2）不同 Page 间导航

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。

6. 设备迁移

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：

设备 A 上的 Page 请求迁移。HarmonyOS 处理迁移任务，并回调设备 A 上 Page 的保存数据方法，用于保存迁移必需的数据。HarmonyOS 在设备 B 上启动同一个 Page，并回调其恢复数据方法。

7. 具有迁移功能的 Page 开发

第一步，实现 IAbilityContinuation 接口。

一个应用可能包含多个 Page，仅需要在支持迁移的 Page 中通过以下方法实现 IAbilityContinuation 接口。同时，此 Page 所包含的所有 AbilitySlice 也需要实现此接口。

onStartContinuation()的使用。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，例如，弹框让用户确认是否开始迁移。

onSaveData()的使用。如果 onStartContinuation()返回值为 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()的使用。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收用于恢复 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，系统回调此方法的时机都在 onStart()之前。

onCompleteContinuation()的使用。在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。

onFailedContinuation()的使用。如果在迁移过程中发生异常，系统则会在发起端设备上回调 FA 的此方法，以便通知应用迁移流程发生的异常。并不是所有异常都会回调 FA 的此方法，仅局限于该接口枚举的异常。开发者可以在此检查异常信息，并在此处理迁移异常发生后的动作，例如，应用可以提醒用户此时发生的异常信息。该接口从 API Version 6 开始提供，目前为 Beta 版本。

onRemoteTerminated()的使用。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，如果目标侧设备上的 Page 因某种原因而终止，则源侧 Page 可通过此回调接收终止通知。

第二步，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，实现代码如下：

```
//第 3 章/设备迁移/continueAbility()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：

设备 A 上的 Page 请求迁移。

系统回调设备 A 上的 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。

如果可以立即迁移，则系统回调设备 A 上的 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所需的数据。

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。

系统回调设备 A 上的 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。

如果在迁移过程中发生异常，系统则回调设备 A 上的 Page 及其 AbilitySlice 栈中所有 AbilitySlice 实例的 IAbilityContinuation.onFailedContinuation()方法，通知在迁移过程中发生异常，并不是所有异常都会回调 FA 的此方法，仅局限于该接口枚举的异常。

第三步，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，代码如下：

```
//第 3 章/使用 continueAbilityReversibly()请求回迁
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：

设备 A 上的 Page 请求回迁。

系统回调设备 B 上的 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。

如果可以立即迁移，则系统回调设备 B 上的 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所需的数据。

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

如果数据恢复成功，则系统终止设备 B 上的 Page 的生命周期。

3.3.3　Service Ability 基本概念

1. 综述

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 退出后，Service 才能退出。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，开发者则必须在 Service 里创建新的线程来处理，这里涉及线程之间的通信，防止造成主线程阻塞，从而导致应用程序无响应。

2. 创建 Service

创建 Ability 的子类，实现与 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，开发者可以重写这些方法，来添加其他 Ability 请求与 Service Ability 交互时的处理方法。

(1) onStart()的使用：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被调用一次，调用时传入的 Intent 应为空。

(2) onCommand()的使用：在 Service 创建完成之后调用，该方法在客户端每次启动该 Service 时都会被调用，开发者可以在该方法中做一些调用统计、初始化类的操作。

(3) onConnect()的使用：在 Ability 和 Service 连接时调用，该方法返回 IRemoteObject 对象，开发者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，只有当第 1 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，而无须再次调用 onConnect()方法。

(4) onDisconnect()的使用：在 Ability 与绑定的 Service 断开连接时调用。

(5) onStop()的使用：在 Service 被销毁时调用。Service 应通过实现此方法来清理所有资源，如关闭线程、注册的侦听器等。

创建 Service 的示例代码如下：

```
//第 3 章/Service Ability 的使用案例
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) {
```

```
            return super.onConnect(intent);
        }

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

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

注册 Service，Service 也需要在应用配置文件中进行注册，需要将注册类型 type 设置为 service。注册 Service 的示例代码如下：

```
"//": "第 3 章注册 Service"
{
    "module": {
       "abilities": [
           {
               "name": ".ServiceAbility",
               "type": "service",
               "visible": true
               ...
           }
       ]
       ...
    }
    ...
}
```

3. 启动 Service

接下来介绍通过 startAbility()启动 Service 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 对象设置目标 Service 信息。这 3 个参数的含义包括以下内容。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，则可以通过 ohos.distributedschedule.interwork.DeviceManager 提供的 getDeviceList 获取设备列表。BundleName 表示包名称。AbilityName 表示待启动的 Ability 名称。

启动本地设备 Service 的示例代码如下：

```
//第 3 章/启动本地 Service Ability
Intent intent = new Intent();
Operation operation = new Intent.OperationBuilder()
        .withDeviceId("")
        .withBundleName("com.domainname.hiworld.himusic")
        .withAbilityName("com.domainname.hiworld.himusic.ServiceAbility")
        .build();
intent.setOperation(operation);
startAbility(intent);
```

启动远程设备 Service 的示例代码如下：

```
//第 3 章/启动远程 Service Ability
Intent intent = new Intent();
Operation operation = new Intent.OperationBuilder()
        .withDeviceId("deviceId")
        .withBundleName("com.domainname.hiworld.himusic")
        .withAbilityName("com.domainname.hiworld.himusic.ServiceAbility")
        .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.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。

4. 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。

5. 连接 Service

如果 Service 需要与 Page Ability 或其他应用的 Service Ability 进行交互，则需创建用于连接的 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 提供了两种方法供开发者实现；onAbilityConnectDone()是用来处理连接 Service 成功的回调，onAbilityDisconnectDone()是用来处理 Service 异常死亡的回调。

创建连接 Service 回调实例的示例代码如下：

```
//第 3 章/创建连接 Service 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) {
        //Client 侧需要定义与 Service 侧相同的 IRemoteObject 实现类.开发者获取服务器端传
        //过来 IRemoteObject 对象,并从中解析出服务器端传过来的信息
    }

    //Service 异常死亡的回调
    @Override
    public void onAbilityDisconnectDone(ElementName elementName, int resultCode) {
    }
};
```

连接 Service 的示例代码如下：

```
//第 3 章/ 连接 Service
Intent intent = new Intent();
Operation operation = new Intent.OperationBuilder()
        .withDeviceId("deviceId")
        .withBundleName("com.domainname.hiworld.himusic")
        .withAbilityName("com.domainname.hiworld.himusic.ServiceAbility")
        .build();
intent.setOperation(operation);
connectAbility(intent, connection);
```

同时，Service 侧也需要在调用 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 LocalRemoteObject 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示例代码如下：

```
//第 3 章/创建自定义 IRemoteObject 实现类
private class MyRemoteObject extends LocalRemoteObject {
    MyRemoteObject(){
    }
}
```

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

6. Service Ability 的生命周期

与 Page 类似，Service 也拥有生命周期，如图 3-13 所示。根据调用方法的不同，其生命周期的启动与销毁有以下两种路径。

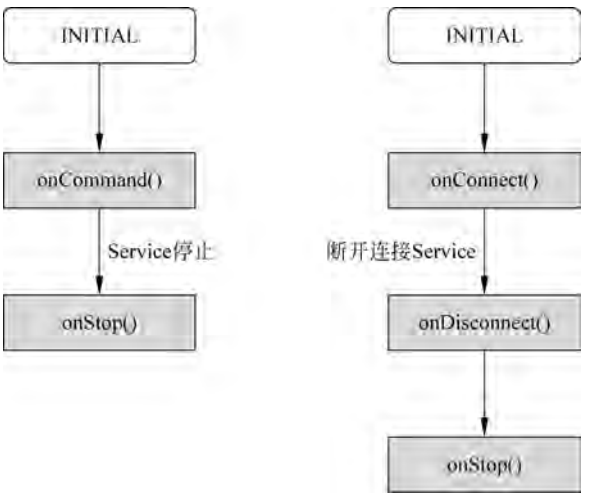

图 3-13　Service 的生命周期

一是启动 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。

另一种是连接 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的 Service，而且当所有绑定全部取消后，系统即会销毁该 Service。

connectAbility()也可以连接通过 startAbility()创建的 Service。

7. 前台 Service

一般情况下，Service 是在后台运行的，后台 Service 的优先级都比较低，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显示在系统状态栏上。

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，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()示例代码如下：

```
//第 3 章/前台 Service 的 onStart()示例代码
//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,需要在 module→abilities 字段下对当前 Service 做如下配置,代码如下：

```
"//": "第 3 章/前台 service 的配置",
{
    "name": ".ServiceAbility",
    "type": "service",
    "visible": true,
    "backgroundModes": ["dataTransfer", "location"]
}
```

3.3.4 Data Ability

1. 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、查操作,以及打开文件等接口,这些接口的具体实现由开发者提供。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,格式如图 3-14 所示。

图 3-14 URI 格式

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

我们用两种场景的URI进行示例说明。

跨设备场景，代码如下：

```
dataability://device_id/com.domainname.dataability.persondata/person/10
```

本地设备，代码如下：

```
dataability:///com.domainname.dataability.persondata/person/10
```

本地设备的device_id字段为空，因此在dataability:后面有3个“/”。

2. 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供用户使用，因此，开发者首先需要确定使用何种类型的数据。本节主要讲述创建Data的基本步骤和需要使用的接口。Data提供方可以自定义数据的增、删、改、查操作，以及文件打开等功能，并对外提供这些接口。

确定数据的存储方式，Data支持两种数据形式：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用于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。实现UserDataAbility，需要在Project窗口当前工程的主目录(entry→src→main→java > com. xxx. xxx)选择File→New→Ability→Empty Data Ability，设置Data Name后完成UserDataAbility的创建。

Data提供了文件存储和数据库存储两组接口供用户使用。

1) 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。

开发者可通过MessageParcel静态方法dupFileDescriptor()复制待操作文件流的文件描述符，并将其返回，供远端应用访问文件。

根据传入的uri打开对应的文件示例，代码如下：

```
//第3章/文件存储 - 根据传入的uri打开对应的文件示例
private static final HiLogLabel LABEL_LOG = new HiLogLabel(HiLog.LOG_APP, 0xD00201, "Data_Log");
```

```
@Override
public FileDescriptor openFile(Uri uri, String mode) throws FileNotFoundException {
    File file = new File(uri.getDecodedPathList().get(0));
                                                    //get(0)用于获取 URI 完整字段中查询参数字段
    if (mode == null || !"rw".equals(mode)) {
        file.setReadOnly();
    }
    FileInputStream fileIs = new FileInputStream(file);
    FileDescriptor fd = null;
    try {
        fd = fileIs.getFD();
    } catch (IOException e) {
        HiLog.info(LABEL_LOG, "failed to getFD");
    }

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

2）数据库存储

第一步，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，以便后续对数据库进行操作。为了避免影响应用启动速度，开发者应当尽可能地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。

初始化的时候连接数据库，示例代码如下：

```
//第 3 章/数据库存储 - 连接数据库示例
private static final String DATABASE_NAME = "UserDataAbility.db";
private static final String DATABASE_NAME_ALIAS = "UserDataAbility";
private static final HiLogLabel LABEL_LOG = new HiLogLabel(HiLog.LOG_APP, 0xD00201, "Data_Log");
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);
}
```

第二步，编写数据库操作方法。

Ability 定义了 6 种方法供用户对数据库表数据进行增、删、改、查处理。这 6 种方法在 Ability 中已默认实现，开发者可按需重写。具体方法如下：

方法 ResultSet query(Uri uri, String[] columns, DataAbilityPredicates predicates)用

来查询数据库。方法 int insert(Uri uri, ValuesBucket value)用来向数据库插入单条数据。方法 int batchInsert(Uri uri, ValuesBucket[] values)用来向数据库插入多条数据。方法 int delete(Uri uri, DataAbilityPredicates predicates)用来删除一条或多条数据。方法 int update(Uri uri, ValuesBucket value, DataAbilityPredicates predicates)用来更新数据库。方法 DataAbilityResult[] executeBatch(ArrayList<DataAbilityOperation> operations)用来批量操作数据库。

这些方法的具体使用说明如下：

query()方法接收 3 个参数，分别是查询的目标路径、查询的列名和查询条件。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示例代码如下：

```
//第 3 章/编写数据库操作方法 - 数据查询
public ResultSet query(Uri uri, String[] columns, DataAbilityPredicates predicates) {
    if (ormContext == null) {
        HiLog.error(LABEL_LOG, "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(LABEL_LOG, "resultSet is null");
    }

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

insert()方法接收两个参数，分别是插入的目标路径和插入的数值。其中，插入的数值由 ValuesBucket 封装，服务器端可以从该参数中解析出对应的属性，然后插入数据库。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的示例代码如下：

```
//第 3 章/编写数据库操作方法 - 添加数据
public int insert(Uri uri, ValuesBucket value) {
    //参数校验
    if (ormContext == null) {
        HiLog.error(LABEL_LOG, "failed to insert, ormContext is null");
        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"));

        //编写数据库操作方法 - 插入数据库
        boolean isSuccessful = ormContext.insert(user);
        if (!isSuccessful) {
            HiLog.error(LABEL_LOG, "failed to insert");
            return -1;
        }
        isSuccessful = ormContext.flush();
        if (!isSuccessful) {
            HiLog.error(LABEL_LOG, "failed to insert flush");
            return -1;
        }
        DataAbilityHelper.creator(this, uri).notifyChange(uri);
        int id = Math.toIntExact(user.getRowId());
        return id;
    }
```

batchInsert()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()方法用来执行删除操作。删除条件由类 DataAbilityPredicates 构建,服务器端在接收到该参数之后可以从中解析出要删除的数据,然后在数据库中执行。根据传入的条件删除用户表数据的示例代码如下:

```
    //第 3 章/编写数据库操作方法 - 删除数据
    public int delete(Uri uri, DataAbilityPredicates predicates) {
        if (ormContext == null) {
            HiLog.error(LABEL_LOG, "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()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,以及在 DataAbilityPredicates 中构建更新的条件等。更新用户表的数据的示例代码如下:

```
//第 3 章/更新用户表的数据
public int update(Uri uri, ValuesBucket value, DataAbilityPredicates predicates) {
    if (ormContext == null) {
        HiLog.error(LABEL_LOG, "failed to update, ormContext is null");
        return -1;
    }

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

executeBatch()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。

需要说明的是在上述代码示例中,初始化了数据库类 BookStore.class,并通过实体类 User.class 对该数据库的表 User 进行增、删、改、查操作。

3) 注册 UserDataAbility

和 Service 类似,开发者必须在配置文件中注册 Data。配置文件中该字段在创建 Data Ability 时会自动创建,name 与创建的 Data Ability 一致。

需要关注以下属性:type 为类型,设置为 data;uri 为对外提供的访问路径,全局唯一;permissions 为访问该 data ability 时需要申请的访问权限。

如果权限非系统权限,则需要在配置文件中进行自定义,示例代码如下:

```
"//":"第 3 章/注册 UserDataAbility - 自定义 DataAbility 访问权限"
{
    "name": ".UserDataAbility",
    "type": "data",
    "visible": true,
    "uri": "dataability://com.example.myapplication5.DataAbilityTest",
    "permissions": [
        "com.example.myapplication5.DataAbility.DATA"
    ]
}
```

3. 访问 Data

开发者可以通过 DataAbilityHelper 类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。

1）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 时应在配置文件中声明需要此权限。声明时需要参考权限申请字段说明，示例代码如下：

```
"//":"第 3 章/声明使用权限 - 声明数据访问权限"
"reqPermissions": [
    {
        "name": "com.example.myapplication5.DataAbility.DATA"
    },
    //访问文件还需要添加访问存储读写权限
    {
        "name": "ohos.permission.READ_USER_STORAGE"
    },
    {
        "name": "ohos.permission.WRITE_USER_STORAGE"
    }
]
```

2）创建 DataAbilityHelper

DataAbilityHelper 为开发者提供了 creator()方法，以此创建 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，示例代码如下：

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

3）访问 Data Ability

DataAbilityHelper 为开发者提供了一系列接口，用于访问不同类型的数据（文件、数据库等）。

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，示例代码如下：

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

```
FileInputStream fis = new FileInputStream(fd);

//使用文件描述符封装成的文件流,进行文件操作
```

访问数据库。DataAbilityHelper 为开发者提供了增、删、改、查及批量处理等方法,以此来操作数据库。对数据库的操作主要包括以下方法。

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

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 为 101～103 的用户,并把结果打印出来,示例代码如下:

```
//第 3 章/DataAbility 数据查询
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示例代码如下:

```
//第 3 章/DataAbility 数据添加
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示例代码如下:

```
//第 3 章/DataAbility 数据批量添加

DataAbilityHelper helper = DataAbilityHelper.creator(this);

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

delete()删除方法,其中删除条件可以通过 DataAbilityPredicates 来构建。删除用户表中 id 为 101～103 的用户,示例代码如下:

```
//第 3 章/DataAbility 数据删除
DataAbilityHelper helper = DataAbilityHelper.creator(this);

//构造删除条件
DataAbilityPredicates predicates = new DataAbilityPredicates();
predicates.between("userId", 101, 103);
helper.delete(uri, predicates);
```

update()更新方法,更新数据由 ValuesBucket 传入,更新条件由 DataAbilityPredicates 构建。更新 id 为 102 的用户,示例代码如下:

```
//第 3 章/DataAbility 数据删除
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()
```

DataAbilityOperation 中提供了设置操作类型、数据和操作条件的方法，开发者可自行设置要执行的数据库操作。插入多条数据的示例代码如下：

```
//第 3 章/DataAbilityOperation 的使用
DataAbilityHelper helper = DataAbilityHelper.creator(abilityObj, insertUri);

//构造批量操作
ValuesBucket value1 = initSingleValue();
DataAbilityOperation opt1 = DataAbilityOperation.newInsertBuilder(insertUri).withValuesBucket
(value1).build();
ValuesBucket value2 = initSingleValue2();
DataAbilityOperation opt2 = DataAbilityOperation.newInsertBuilder(insertUri).withValuesBucket
(value2).build();
ArrayList<DataAbilityOperation> operations = new ArrayList<DataAbilityOperation>();
operations.add(opt1);
operations.add(opt2);
DataAbilityResult[] result = helper.executeBatch(insertUri, operations);
```

3.3.5 Intent

1. 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，或者当一个 AbilitySlice 需要导航到另一个 AbilitySlice 时，可以通过 Intent 指定启动的目标，与此同时指定携带相关数据。

Intent 的构成元素包括 Operation 与 Parameters，具体描述如下。

1）属性 Operation

子属性 Action 表示动作，通常使用系统预置的 Action，应用也可以自定义 Action。例如 IntentConstants.ACTION_HOME 表示返回桌面动作。子属性 Entity 表示类别，通常使用系统预置的 Entity，应用也可以自定义 Entity。例如 Intent.ENTITY_HOME 表示在桌面显示图标。子属性 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。子属性 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。子属性 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。子属性 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。子属性 DeviceId 表示运行指定 Ability 的设备 ID。

2）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。

2. 两种类型

当 Intent 用于发起请求时，根据指定元素的不同，可分为以下两种类型：

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。

Intent 设置属性时，必须先使用 Operation 设置属性。如果需要新增或修改属性，则必须在设置 Operation 后再执行操作。关于 Intent 最简单的使用方法，可参见快速入门的示例代码。其中“实现页面跳转”重点描述了使用 Intent 实现两个页面跳转关系的操作。

1）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：

```
//第 3 章/DataAbilityOperation 的使用
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。

2）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应用具体是哪一个应用。例如如果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。以下示例展示如何使用 Intent 跨 Ability 查询天气信息。

请求方在 Ability 中构造 Intent 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，示例代码如下：

```
//第 3 章/重写 onAbilityResult()回调方法
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，以便系统据此找到此能力并作为候选的请求处理者，示例代码如下：

```
"//":"第 3 章/声明对外提供的能力"
{
    "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，示例代码如下：

```
//第 3 章/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，示例代码如下：

```
//第 3 章/请求结果处理
@Override
protected void onActive() {
    …
    Intent resultIntent = new Intent();
    setResult(0, resultIntent);     //0 为当前 Ability 销毁后返回的 resultCode
    …
}
```