第 3 章

Ability 框架开发——基于 Java

本章将基于 Java 对 Ability、公共事件与通知开发、后台任务调度和管控、线程管理开发、线程间通信、剪贴板开发进行介绍。受限于本书的篇幅，基于 JS 的 Ability 开发随书附赠。

3.1 开发概述

HarmonyOS 应用开发主要包括通用开发和原子化服务开发。其中，通用开发包括开发 Ability、开发 UI 和开发业务功能。

1. 开发 Ability

进行 HarmonyOS 应用开发要了解 Ability 如何使用。Ability 是 HarmonyOS 应用程序的重要组成部分，分为 FA 和 PA 两种类型。

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。

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。

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。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。

2. 开发 UI

FA 需要提供 UI 用于与用户进行交互，HarmonyOS 提供 Java UI 和 JS UI 两种 UI 框架：Java UI 提供细粒度的 UI 编程接口，使应用开发更加灵活；JS UI 提供相对高层的 UI 描述，使应用开发更加简单。针对轻量级智能穿戴(Lite Wearable)，现阶段只使用 JS 语言进行应用开发，示例工程参考地址为：https://developer.harmonyos.com/cn/docs/documentation/doc-guides/lite-wearable-experience-0000000000622606。

3. 开发业务功能

媒体：视频、音频、图像、相机等功能的开发。

安全：权限、生物特征识别等功能的开发。

AI：图像超分、语音识别等功能的开发。

网络连接：NFC、蓝牙、WLAN 等功能的开发。

设备管理：传感器、控制类小器件、位置等功能的开发。

数据管理：数据库、分布式数据/文件服务、数据搜索等功能的开发。

线程：线程管理、线程间通信等功能的开发。

IDL：声明系统服务和 Ability 对外提供的服务接口，并生成相关代码。

4. 原子化服务开发

HarmonyOS 除支持传统方式需要安装应用外，还支持提供特定功能的免安装应用（原子化服务），供用户在合适的场景和设备上便捷使用。

原子化服务相对于传统方式需要安装的应用更加轻量，同时提供丰富的入口、更精准的分发。

3.2 Ability 介绍

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（可以包含多个 Ability），HarmonyOS 支持应用以 Ability 为单位进行部署。

在配置文件（config.json）中注册 Ability 时，可以通过配置 Ability 元素中的 type 属性指定 Ability 模板类型，相关代码如下。

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

其中，type 的取值可以为 page、service 或 data，分别代表 Page、Service 和 Data 模板。为便于表述，后续将基于 Page 模板、Service 模板、Data 模板实现的 Ability 分别简称为 Page、Service 和 Data。

3.2.1 Page Ability

本部分包括 Page 与 AbilitySlice、AbilitySlice 路由配置、Page Ability 生命周期、AbilitySlice 间导航和跨设备迁移。

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应具有高度相关性。例如,新闻浏览功能可以通过一个 Page 实现,其中包含了两个 AbilitySlice:一个用于展示新闻列表,另一个用于展示新闻详情。Page 与 AbilitySlice 的关系如图 3-1 所示。

图 3-1 Page 与 AbilitySlice 的关系

相比桌面场景,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,当需要其他能力辅助时,会调用其他应用提供的能力。例如,外卖应用提供了联系商家的业务功能入口,当用户在使用该功能时,会跳转到通话应用的拨号页面。与此类似,HarmonyOS 支持不同 Page 之间的跳转,并可以指定跳转到目标 Page 中某个具体的 AbilitySlice。

2. AbilitySlice 路由配置

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

setMainRoute()方法与 addActionRoute()方法的代码如下。

```
public class MyAbility extends Ability {
    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
        //设置主路由
        setMainRoute(MainSlice.class.getName());
        //设置动作路由
        addActionRoute("action.pay", PaySlice.class.getName());
        addActionRoute("action.scan", ScanSlice.class.getName());
    }
}
```

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

```
{
    "module": {
        "abilities": [
            {
                "skills":[
                    {
                        "actions":[
                            "action.pay",
                            "action.scan"
                        ]
                    }
                ]
                ...
            }
        ]
        ...
    }
    ...
}
```

3. 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(释放资源),有助于提升应用的性能。

1) Page 生命周期回调

Page 生命周期的不同状态转换及其对应的回调如图 3-2 所示。

(1) onStart()。当系统首次创建 Page 实例时,触发该回调。对于一个 Page 实例,回调在其生命周期过程中仅触发一次,Page 在逻辑后进入 INACTIVE 状态。开发者必须重写此方法,并配置默认展示的 AbilitySlice。

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

(2) onActive()。Page 会在进入 INACTIVE 状态后到前台,然后系统调用此回调。Page 在此之后进入 ACTIVE 状态,它是应用与用户交互的状态。Page 将保持在此状态,除非某类事件发生导致 Page 失去焦点,例如用户单击返回按钮或导航到其他 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

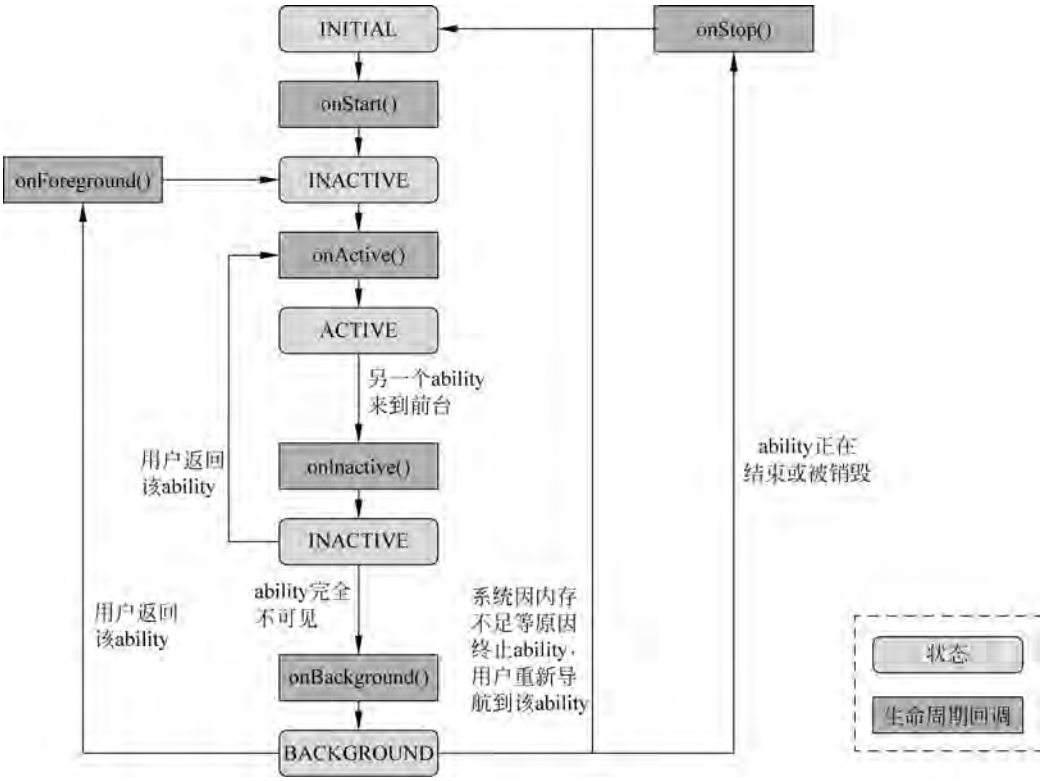

图 3-2　Page 生命周期

资源。

(3) 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。

(4) onBackground()。如果 Page 不再对用户可见，系统将调用此回调通知用户进行相应的资源释放，此后 Page 进入 BACKGROUND 状态。开发者应在此回调中释放 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。

(5) onForeground()。处于 BACKGROUND 状态的 Page 仍然驻留在内存中，当重新回到前台时(用户重新导航到此 Page)，系统先调用 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 onBackground()中释放的资源，最后 Page 的生命周期回到 ACTIVE 状态，系统通过 onActive()回调通知开发者用户。

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

发对处于 BACKGROUND 状态 Page 的销毁。

2）AbilitySlice 生命周期

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

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

```
@Override
protected void onStart(Intent intent) {
    super.onStart(intent);
    setUIContent(ResourceTable.Layout_main_layout);
}
```

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

3）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。一个 Page 拥有多个 AbilitySlice 时，例如 MyAbility 下有 FooAbilitySlice 和 BarAbilitySlice，当前 FooAbilitySlice 处于前台且获得焦点，并将导航到 BarAbilitySlice，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 将联动销毁，而不仅是处于前台的 AbilitySlice。

4. AbilitySlice 间导航

本部分介绍同一 Page 内导航和不同 Page 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，通过 present()方法实现导航。单击按钮导航到其他 AbilitySlice 的相关代码如下。

```
@Override
```

```
protected void onStart(Intent intent) {
    ...
    Button button = ...;
    button.setClickedListener(listener -> present(new TargetSlice(), new Intent()));
    ...
}
```

如果希望在用户从导航目标 AbilitySlice 返回时能够获得其返回结果，则应当使用 presentForResult()实现导航。用户从导航目标 AbilitySlice 返回时，系统将回调 onResult()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，相关代码如下。

```
int requestCode = positiveInteger; //任何正整数
@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) {
        //在此处理 resultIntent
    }
}
```

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，若指定的 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。前面的示例代码中，导航时指定的 AbilitySlice 实例均是新建的，即便重复执行此代码(此时作为导航目标的这些实例是同一个类)，也不会导致任何 AbilitySlice 出栈。

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，因此通过配置 Intent 的 Action 导航到目标 AbilitySlice。Page 间的导航使用 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。

5. 跨设备迁移

跨设备迁移(以下简称“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需求，开发步骤如下。

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

onRestoreData()：源侧设备上 Page 完成保存数据后，系统在目标侧设备上回调此方法，开发者在此回调中接收用于恢复 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，且系统回调此方法的时机在 onStart()之前。

onCompleteContinuation()：目标侧设备上恢复数据一旦完成，系统就会在源侧设备上回调 Page 的方法，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()：迁移过程中发生异常，系统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，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，例如可以提醒用户此时发生的异常信息。该接口从 API 版本 6 开始提供，目前为 Beta 版本。

onRemoteTerminated()：如果使用 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(两个实例)同时在两个设备上运行，迁移完成后，如果目标侧设备上 Page 因任何原因终止，则源侧 Page 通过此回调接收终止通知。

2）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。二者区别：通过后者发起的迁移，可以进行回迁。

```
try {
    continueAbility();
} catch (IllegalStateException e) {
    //可以继续进行其他处理
    ...
}
```

以 Page 从设备 A 迁移到设备 B 为例，具体流程如下。

(1) 设备 A 上的 Page 请求迁移。

(2) 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。

(3) 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的

数据。

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

(5) 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。

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

3) 请求回迁

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

```
try {
    reverseContinueAbility();
} catch (IllegalStateException e) {
    //可能另一个正在进行中
    ...
}
```

以 Page 从设备 A 迁移到设备 B 后并请求回迁为例,具体流程如下。

(1) 设备 A 上的 Page 请求回迁。

(2) 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。

(3) 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的数据。

(4) 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。

(5) 如果数据恢复成功,则系统终止设备 B 上 Page 的生命周期。

3.2.2 Service Ability

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

Service 是单实例的。在一个设备上,相同的 Service 只会存在一个实例。如果多个 Ability 共用这个实例,只有当与 Service 绑定的所有 Ability 都退出后,Service 才能退出。由于 Service 在主线程中执行,因此 Service 里面的操作时间过长,开发者必须在 Service 中创建新的线程进行处理,防止造成主线程阻塞,应用程序无响应。

1. 创建 Service

创建 Ability 的子类，实现 Service 相关的生命周期方法。Service 也是一种 Ability，Ability 为 Service 提供了以下生命周期方法，开发者可以重写这些方法，添加其他 Ability 请求与 Service Ability 交互时的处理方法。

(1) onStart()：该方法在创建 Service 时调用，用于 Service 的初始化。在 Service 的整个生命周期只会调用一次，调用时传入的 Intent 为空。

(2) onCommand()：在 Service 创建完成之后调用，该方法在客户端每次启动 Service 时都会调用，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()：在 Ability 和 Service 连接时调用，该方法返回 IRemoteObject 对象，开发者可以在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 Service 的 IPC 通信对象，只有第一个客户端连接 Service 时，系统才会调用 Service 的 onConnect 方法生成 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()：在 Ability 与绑定的 Service 断开连接时调用。

(5) onStop()：在 Service 销毁时调用。Service 应通过实现此方法清理任何资源，例如关闭线程、注册的侦听器等，相关代码如下。

```
public class ServiceAbility extends Ability {
    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
    }
    @Override
    public void onCommand(Intent intent, boolean restart, int startId) {
        super.onCommand(intent, restart, startId);
    }
    @Override
    public IRemoteObject onConnect(Intent intent) {
        return super.onConnect(intent);
    }
    @Override
    public void onDisconnect(Intent intent) {
        super.onDisconnect(intent);
    }
    @Override
    public void onStop() {
        super.onStop();
    }
}
```

2. 注册 Service

Service 需要在应用配置文件中进行注册，注册类型 Type 需要设置为 Service，相关代

码如下。

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

3. 启动 Service

通过 startAbility()启动 Service 及对应的停止方法。

1) 启动 Service

Ability 提供 startAbility()方法启动另外一个 Ability。Service 也是 Ability 的一种,同样可以将 Intent 传递给该方法启动 Service。startAbility()不仅支持启动本地 Service,还支持启动远程 Service。

可以通过构造包含 DeviceId、BundleName 与 AbilityName 的 Operation 对象设置目标 Service 信息,参数含义如下。

DeviceId:表示设备 ID。如果是本地设备,则可以直接留空;如果是远程设备,可以通过 ohos.distributedschedule.interwork.DeviceManager 提供的 getDeviceList 获取设备列表。

BundleName:表示包名称。

AbilityName:表示待启动的 Ability 名称。

启动本地设备 Service 的相关代码如下。

```
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 的相关代码如下。

```
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 通过 startAbility()方法启动 Service。如果 Service 尚未运行，则系统会先调用 onStart()方法初始化 Service，再回调 Service 的 onCommand()方法启动 Service。如果 Service 正在运行，则系统会直接回调 Service 的 onCommand()方法启动 Service。

2）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()停止本地 Service 或在其他 Ability 调用 stopAbility()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。

4. 连接 Service

如果 Service 需要与 Page Ability 或其他应用的 Service Ability 进行交互，则需要创建用于连接的 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 提供两种方法：onAbilityConnectDone()是处理连接 Service 成功的回调；onAbilityDisconnectDone()是处理 Service 异常死亡的回调，创建连接 Service 回调实例的相关代码如下。

```
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输
        //的 IRemoteObject 对象,并从中解析出服务器端传输的信息
    }
    //Service 异常死亡的回调
    @Override
    public void onAbilityDisconnectDone(ElementName elementName, int resultCode) {
    }
};
```

连接 Service 相关代码如下。

```
//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 创建自定义的实现类。Service 侧将自身的实例返回给调用侧的相关代码如下。

```
//创建自定义 IRemoteObject 实现类
private class MyRemoteObject extends LocalRemoteObject {
    MyRemoteObject(){
    }
}
//将 IRemoteObject 返回给客户端
@Override
protected IRemoteObject onConnect(Intent intent) {
    return new MyRemoteObject();
}
```

5. Service Ability 生命周期

与 Page 类似，Service 也拥有生命周期，如图 3-3 所示。根据调用方法不同，其生命周期有以下两种路径。

启动 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()停止 Service，Service 停止后，系统会将其销毁。

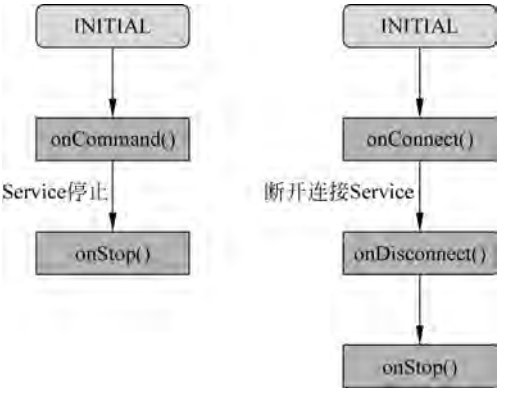

图 3-3　Service 生命周期

连接 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相同的 Service，而且当所有绑定全部取消后，系统会销毁 Service。

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

6. 前台 Service

一般情况下，Service 都是在后台运行的，后台 Service 的优先级比较低，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，它始终保持正在运行的图标在系统状态栏显示。

使用前台 Service 并不复杂，只需在 Service 创建的方法中，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()相关代码如下。

```
//创建通知，其中 1005 为 notificationId
NotificationRequest request = new NotificationRequest(1005);
NotificationRequest.NotificationNormalContent content = new NotificationRequest.Notification
NormalContent();
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，module > abilities 字段下对当前 Service 做如下配置。

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

针对 Service Ability 开发，启动、停止、连接、断开连接等支持对跨设备的 Service Ability 进行操作。示例工程参考地址为：https://gitee.com/openharmony/app_samples/tree/master/ability/ServiceAbility。

3.2.3 Data Ability

使用 Data 模板的 Ability(以下简称 Data)有助于应用管理其自身和其他应用存储数据的访问，并提供与其他应用共享数据的方法。Data 既可用于同设备不同应用的数据共享，

也支持跨设备不同应用的数据共享。数据的存放形式多样，可以是数据库，也可以是磁盘上的文件。Data 对外提供对数据的增、删、改、查，以及打开文件等接口，这些接口的具体实现由开发者提供。

Data 的提供方和使用方都通过 URI(Uniform Resource Identifier)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 3-4 所示。

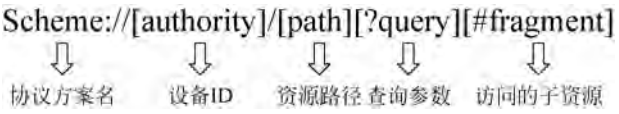

图 3-4　标识格式

Scheme：协议方案名，固定为 dataability，代表 Data Ability 所使用的协议类型。

authority：设备 ID，如果为跨设备场景，则为目标设备的 ID，如果为本地设备场景，则不需要填写。

path：资源的路径信息，代表特定资源的位置信息。

query：查询参数。

fragment：可以用于指示要访问的子资源。

URI 示例如下：

跨设备场景：dataability://device_id/com.domainname.dataability.persondata/person/10。

本地设备：dataability:///com.domainname.dataability.persondata/person/10。

本地设备的 device_id 字段为空，因此在 dataability: 后面有三个“/”。

1. 创建 Data

使用 Data 模板的 Ability 形式仍然是 Ability，需要为应用添加一个或多个 Ability 的子类，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不同 API 接口供用户使用，因此需要确定好使用何种类型的数据。本部分主要讲述创建 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，以及文件打开等功能，并对外提供这些接口。Data 支持以下两种数据形式：文件数据，例如文本、图片、音乐等，结构化数据，例如数据库等。

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,操作如下：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。

通过 MessageParcel 静态方法 dupFileDescriptor()复制到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 打开对应的文件代码如下。

```
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)是获取 uri 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

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,应该创建数据库连接,并获取连接对象,以便后续和数据库进行操作。为了避免影响应用启动速度,尽可能将非必要的耗时任务推迟到使用时执行,而不是在此方法中执行所有初始化,相关代码如下。

```
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);
}
```

Ability 定义了 6 个方法供用户处理对数据库的增、删、改、查。这 6 个方法在 Ability 中已默认实现，开发者可按需重写，如表 3-1 所示。

表 3-1 编写数据库方法及功能描述

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

在初始化数据库类 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，具体方法如下。

(1) query()：接收三个参数，分别是查询的目标路径、列名及查询条件，查询条件由类 DataAbilityPredicates 构建。根据传入的列名和查询条件查询用户表，相关代码如下。

```
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;
}
```

(2) insert()：接收两个参数，分别是插入的目标路径和数据值。其中，插入的数据由 ValuesBucket 封装，服务器端可以从该参数中解析出对应的属性，然后插入数据库中。此方法返回一个 int 类型的值用于标识结果。接收到传输过来的用户信息并保存到数据库中的相关代码如下。

```
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"));
        //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;
    }
```

(3) batchInsert()：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，可以直接调用。

(4) delete()：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，然后到数据库中执行。根据传入的条件删除用户表数据的相关代码如下。

```
    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;
    }
```

(5) update()：用来执行更新操作。用户可以在 ValuesBucket 参数中指定要更新的数据，在 DataAbilityPredicates 中构建更新的条件等，相关代码如下。

```
    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;
    }
```

(6) executeBatch()：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，用户可自行设置要执行的数据库操作。该方法系统已实现，可以直接调用。

2. 注册 UserDataAbility

与 Service 类似，在配置文件中必须注册 Data，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：类型设置为 Data。URI：对外提供的访问路径，全局唯一。permissions：访问该 Data Ability 时需要申请的访问权限。如果是非系统权限，在配置文件中进行自定义，相关代码如下。

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

3. 访问 Data

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

如果待访问的 Data 声明了访问需要权限，则访问此 Data 需要在配置文件中进行声明。

```
"reqPermissions": [
    {
        "name": "com.example.myapplication5.DataAbility.DATA"
    },
    //访问文件需要添加访问存储读写权限
```

```
    {
        "name": "ohos.permission.READ_USER_STORAGE"
    },
    {
        "name": "ohos.permission.WRITE_USER_STORAGE"
    }
]
```

创建 DataAbilityHelper：DataAbilityHelper 为开发者提供 creator()方法创建 DataAbilityHelper 实例。此方法为静态方法，有多个重载。最常见的方法是通过传入一个 context 对象创建 DataAbilityHelper 对象，获取 Helper 对象示例如下。

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

访问 Data Ability：DataAbilityHelper 为开发者提供一系列的接口，访问不同类型的数据(文件、数据库等)。

访问文件：DataAbilityHelper 为开发者提供 FileDescriptor openFile(Uri uri, String mode)方法操作文件。此方法需要传入两个参数，其中 URI 确定目标资源路径，Mode 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)。

此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，就可以对文件流进行自定义处理，访问文件示例如下。

```
FileDescriptor fd = helper.openFile(uri, "r");    //读取文件描述符
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 提供增、删、改、查及批量处理等方法操作数据库。

针对 Data Ability 开发，示例工程参考地址为：https://gitee.com/openharmony/app_samples/tree/master/ability/DataAbility。

3.2.4 Intent

Intent 是对象之间传递信息的载体。例如，当一个 Ability 需要启动另一个 Ability 时，或者一个 AbilitySlice 需要导航到另一个 AbilitySlice 时，可以通过 Intent 指定启动的目标同时携带相关数据。Intent 的构成元素包括 Operation 与 Parameters，如表 3-2 所示。

Intent 用于发起请求时，根据指定元素的不同，操作分为两种类型：如果同时指定了 BundleName 与 AbilityName，则根据 Ability 的全称(如 com.demoapp.FooAbility)直接启动应用；如果未同时指定 BundleName 和 AbilityName，则根据 Operation 中的其他属性启动应用。Intent 设置属性时，必须先使用 Operation 进行设置。如果需要新增或修改属性，必须在设置 Operation 后再执行操作。

表 3-2 Operation 与 Parameters 描述

属性	子属性	功能描述
Operation	Action	表示动作，通常使用系统预置 Action，应用也可以自定义 Action，例如 IntentConstants. ACTION_HOME 表示返回桌面动作
	Entity	表示类别，通常使用系统预置 Entity，应用也可以自定义 Entity，例如 Intent. ENTITY_HOME 表示在桌面显示图标
	URI	表示 URI 描述。如果在 Intent 中指定了 URI，则 Intent 将匹配指定的 URI 信息，包括 scheme、schemeSpecificPart、authority 和 path 信息
	Flags	表示处理 Intent 的方式，例如 Intent. FLAG_ABILITY_CONTINUATION 标记在本地的一个 Ability 是否可以迁移到远端设备继续运行
	BundleName	表示包描述。如果在 Intent 中同时指定 BundleName 和 AbilityName，则 Intent 可以直接匹配到指定的 Ability
	AbilityName	表示待启动的 Ability 名称。如果在 Intent 中同时指定 BundleName 和 AbilityName，则 Intent 可以直接匹配到指定的 Ability
	DeviceId	表示运行指定 Ability 的设备 ID
Parameters	—	Parameters 是一种支持自定义的数据结构，开发者可以通过 Parameters 传递某些请求所需的额外信息

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，相关代码如下。

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

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 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

有些场景下需要使用其他应用提供的某种能力，例如通过浏览器打开一个链接，而不关

心用户最终选择哪个浏览器应用，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，由用户选择使用哪个应用处理请求。以下示例展示使用 Intent 跨 Ability 查询天气信息。

1）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，相关代码如下。

```
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:
            //处理结果
            ...
            return;
        default:
            ...
    }
}
```

2）处理方

作为处理请求对象，需要在配置文件中声明对外提供的能力，以便系统可以找到并进行处理，相关代码如下。

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

```
                        ]
                    }
                ]
                ...
            }
        ]
        ...
    }
    ...
}
```

在 Ability 中配置路由以便支持以 action 导航到对应的 AbilitySlice，相关代码如下。

```
@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，相关代码如下。

```
@Override
protected void onActive() {
    ...
    Intent resultIntent = new Intent();
    setResult(0, resultIntent);    //0 为当前 Ability 销毁后返回的 resultCode
    ...
}
```

针对 Intent 开发，演示如何根据 Ability 全称启动应用和根据 Operation 的其他属性启动示例工程，参考地址为：https://gitee.com/openharmony/app_samples/tree/master/ability/Intent。

3.2.5 Ability 示例

在 HarmonyOS 中，Ability 是应用所具备能力的抽象，一个应用可以具备多种能力（包含多个 Ability），HarmonyOS 支持应用以 Ability 为单位进行部署。其中，Page Ability 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 指应用的单个页面及其控制逻辑的总和。示例工程参考地址为：https://developer.huawei.com/consumer/cn/codelabsPortal/carddetails/Ability_Intent。

在一个 Ability 中，可以使用 present()/presentForResult() 方法，从一个 AbilitySlice 导航到一个新的 AbilitySlice，即 Page 内不同 AbilitySlice 的跳转。同时，也可以在 AbilitySlice 中使用 startAbility()/startAbilityForResult() 方法启动一个新的 Ability。

在本示例中，尝试构建 2 个 Ability、3 个 AbilitySlice 完成两种类型的跳转。在 MainAbilitySlice 页面，写两个简单的按钮，其中一个实现 Page 内跳转，如图 3-5 所示，另一个实现 Page 间跳转，如图 3-6 所示。在 NewAbilitySlice 和 SecondAbilitySlice 页面，通过一个按钮让它回到第一个页面，单击第一个按钮跳转到第二个 AbilitySlice，再跳回；单击第二个按钮跳转到第二个 Ability，再跳回。

图 3-5　Page 内跳转

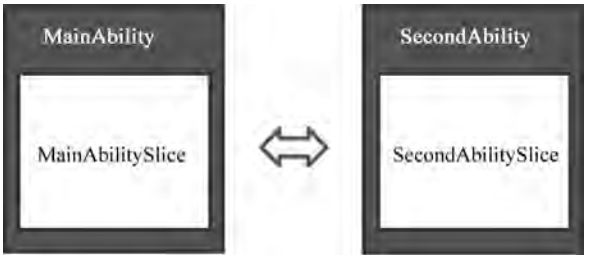

图 3-6　Page 间跳转

如图 3-7 所示，构造 Operation 对象，通过 startAbility()/startAbilityForResult()实现不同 Page 间导航。

图 3-7　Page 间导航

1. 新建项目

打开 DevEco Studio 开发环境，以使用 Java 语言开发、设备类型为 Phone 的 Application 为例，模板选择 Empty Feature Ability(Java)。工程创建完成后，使用 Phone 模拟器运行

工程。

2. 编写页面布局

在 Java UI 框架中，系统提供两种编写布局的方式：UI 布局和在代码中创建布局。以 XML 方式为例，在 XML 中声明 UI 布局和在代码中创建布局步骤如下。

在创建项目后，系统自动创建 MainAbility 和 MainAbilitySlice。在 Project 窗口单击 entry→src→main→resources→base→layout，打开 ability_main.xml 文件。

编写一个文本和两个按钮，使用 DependentLayout 布局，通过 Text 和 Button 组件实现，ability_main.xml 的相关代码如下。

```
<?xml version = "1.0" encoding = "utf - 8"?>
< DirectionalLayout
      xmlns:ohos = "http://schemas.huawei.com/res/ohos"
      ohos:height = "match_parent"
      ohos:width = "match_parent"
      ohos:alignment = "center"
      ohos:orientation = "vertical">
      < Text
          ohos:id = " $ + id:main_text"
          ohos:text = "嗨，我是 MainAbilitySlice，看标题就知道我在哪个 Page 啦"
          ohos:width = "320fp"
          ohos:height = "match_content"
          ohos:text_alignment = "center"
          ohos:multiple_lines = "true"
          ohos:text_size = "20fp"
          ohos:above = " $ + id:enter_second"
          ohos:bottom_margin = "50vp"/>
      < Button
          ohos:id = " $ + id:enter_newAbilitySlice"
          ohos:width = "260vp"
          ohos:height = "45vp"
          ohos:text = "导航到 NewAbilitySlice"
          ohos:multiple_lines = "true"
          ohos:text_size = "20fp"
          ohos:bottom_margin = "20vp"
          ohos:center_in_parent = "true"
          ohos:background_element = " $graphic:background_ability_main"/>
      < Button
          ohos:id = " $ + id:enter_second"
          ohos:width = "260vp"
          ohos:height = "45vp"
          ohos:text = "导航到 SecondAbility"
          ohos:multiple_lines = "true"
          ohos:text_size = "20fp"
          ohos:bottom_margin = "20vp"
          ohos:center_in_parent = "true"
```

```
        ohos:background_element = " $graphic:background_ability_main"/>
</DirectionalLayout >
```

代码中的按钮使用 background_element＝" $graphic：background_ability_main " ，即 entry→src→main→resources→base→graphic 中的 background_ability_main. xml 文件，完成上述步骤后，实现第一个页面布局，相关代码如下。

```
< shape xmlns:ohos = "http://schemas.huawei.com/res/ohos"
        ohos:shape = "rectangle">
    < corners
            ohos:radius = "10"/>
    < solid
            ohos:color = " # ffc0c0c0"/>
    < stroke
            ohos:color = " # ff00ff7f"
            ohos:width = "0.4vp"/>
</shape >
```

创建第二个 Page，在 Project 窗口中，打开 entry→src→main→java→包名，右击 slice 文件夹，选择 New→Ability→Empty Page Ability(Java)，命名为 SecondAbility，按下 Enter 键，系统会创建 SecondAbility. java 和 slice/SecondAbilitySlice. java。

首先，在 Project 窗口中，单击 entry→src→main→resources→base→layout，打开 ability_second. xml 文件，增加信息显示区域和跳转按钮，完成第二个页面布局，相关代码如下。

```
<?xml version = "1.0" encoding = "utf - 8"?>
< DirectionalLayout
    xmlns:ohos = "http://schemas.huawei.com/res/ohos"
    ohos:height = "match_parent"
    ohos:width = "match_parent"
    ohos:alignment = "center"
    ohos:orientation = "vertical">
    < Text
        ohos:id = " $ + id:second_text"
        ohos:width = "320fp"
        ohos:height = "match_content"
        ohos:multiple_lines = "true"
        ohos:text = "嗨, 我是 SecondAbilitySlice"
        ohos:text_alignment = "center"
        ohos:text_size = "20fp"
        ohos:above = " $ + id:second_back_first"
        ohos:bottom_margin = "50vp"/>
    < Button
        ohos:id = " $ + id:second_back_first"
        ohos:width = "260vp"
        ohos:height = "45vp"
```

```
        ohos:text = "回到 MainAbility"
        ohos:multiple_lines = "true"
        ohos:text_size = "20fp"
        ohos:bottom_margin = "20vp"
        ohos:center_in_parent = "true"
        ohos:background_element = " $graphic:background_ability_main"/>
</DirectionalLayout >
```

其次，在 entry→src→main→java→包名→slice 文件夹上右击新建一个 NewAbilitySlice.java 文件，并在 entry→src→main→resources→base→layout 中新建 ability_main_new.xml 文件，用于实现 Page 内跳转调用。在 ability_main_new.xml 文件中，同样增加一个信息显示区域和跳转按钮，相关代码如下。

```
<?xml version = "1.0" encoding = "utf - 8"?>
< DirectionalLayout
    xmlns:ohos = "http://schemas.huawei.com/res/ohos"
    ohos:height = "match_parent"
    ohos:width = "match_parent"
    ohos:alignment = "center"
    ohos:orientation = "vertical">
    < Text
        ohos:id = " $ + id:new_text"
        ohos:width = "320fp"
        ohos:height = "match_content"
        ohos:multiple_lines = "true"
        ohos:text = "我是 NewAbilitySlice.麻烦看看标题，请告诉我现在在哪个 Page 呢?"
        ohos:text_alignment = "center"
        ohos:text_size = "20fp"
        ohos:above = " $ + id:second_back_first"
        ohos:bottom_margin = "50vp"/>
    < Button
        ohos:id = " $ + id:new_to_main"
        ohos:width = "260vp"
        ohos:height = "45vp"
        ohos:text = "回到 MainAbilitySlice"
        ohos:multiple_lines = "true"
        ohos:text_size = "20fp"
        ohos:bottom_margin = "20vp"
        ohos:center_in_parent = "true"
        ohos:background_element = " $graphic:background_ability_main"/>
</DirectionalLayout >
```

最后，对 Ability 页面的标题栏进行修改，便于分辨页面。在 Project 窗口中，单击 entry→src→main，打开 config.json 文件，修改 abilities 中两个 Label 字段分别为 Page MainAbility 和 Page SecondAbility。默认情况下，Label 字段生成为引用模式不可直接修改，按住 Ctrl 键单击该字段，跳转到被引用的 string.json 文件，修改对应字段即可，相关代码如下。

```
{
  "string": [
    {
      "name": "entry_MainAbility",
      "value": "Page MainAbility"
    },
    {
      "name": "entry_SecondAbility",
      "value": "Page SecondAbility"
    },
    ...
  ]
}
```

3. 编写 AbilitySlice 间的导航

打开 entry→src→main→java→包名→slice 中的 MainAbilitySlice.java 文件，添加两个按钮的响应逻辑，实现单击按钮跳转到下一页。

其中，Page 内 AbilitySlice 间的导航选择使用 presentForResult()实现，可以获得从导航目标 AbilitySlice 返回时的结果，相关代码如下。

```
private Text backValueText;
@Override
public void onStart(Intent intent) {
      super.onStart(intent);
      super.setUIContent(ResourceTable.Layout_ability_main);
      //开始进入 NewAbilitySlice
       Component enterNewAbilitySliceButton = findComponentById(ResourceTable.Id_enter_
newAbilitySlice);
       enterNewAbilitySliceButton.setClickedListener(listener -> presentForResult(new
NewAbilitySlice(), new Intent(), 0));
      //开始进入 SecondAbility page
      Component enterSecondAbilityButton = findComponentById(ResourceTable.Id_enter_second);
      enterSecondAbilityButton.setClickedListener(component -> startEnterSecondAbility());
      backValueText = (Text) findComponentById(ResourceTable.Id_main_text);
}
```

打开同文件夹下新建的 NewAbilitySlice.Java 文件，通过 setResult()完成返回结果的实现，相关代码如下。

```
package com.huawei.abilityintent.slice;
import com.huawei.abilityintent.ResourceTable;
import ohos.aafwk.ability.AbilitySlice;
import ohos.aafwk.content.Intent;
import ohos.agp.components.Component;
public class NewAbilitySlice extends AbilitySlice {
```

```
    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
        super.setUIContent(ResourceTable.Layout_ability_main_new);
        Component newToMainButton = findComponentById(ResourceTable.Id_new_to_main);
        newToMainButton.setClickedListener(component -> terminate());
    }
    @Override
    public void onActive() {
        super.onActive();
        Intent intent = new Intent();
        intent.setParam("key", "我从 NewAbilitySlice 跳回来咯");
        setResult(intent);
    }
    @Override
    public void onForeground(Intent intent) {
        super.onForeground(intent);
    }
}
```

继续在 MainAbilitySlice.java 中补充如下代码，通过 onResult 完成返回结果的接收，并显示在页面的 Text 控件中，相关代码如下。

```
/* presentForResult()结果 */
@Override
protected void onResult(int requestCode, Intent resultIntent) {
    if (requestCode != 0 || resultIntent == null) {
        return;
    }
    String result = resultIntent.getStringParam("key");
    backValueText.setText(result);
}
```

4. 编写 Page 之间的导航

在 entry→src→main→java→包名的 MainAbility.java 文件内，可以看到在 onStart() 中已经通过路由导航到 MainAbilitySlice 页面。

```
public class MainAbility extends Ability {
    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
        super.setMainRoute(MainAbilitySlice.class.getName());
    }
}
```

同理，SecondAbility.java 也已经导航到 SecondAbilitySlice.java 页面，接下来完成两

个 Page Ability 之间的导航。

打开 entry→src→main→java→包名→slice 中的 MainAbilitySlice.java 文件，进一步完成 Page 间导航按钮的响应逻辑，实现对应的方法 startEnterSecondAbility()。

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()。为了获取返回值，这里使用 startAbilityForResult()方法跳转到 SecondAbility，相关代码如下。

```
/* 显式启动 */
private void startEnterSecondAbility() {
    Intent intent = new Intent();
    Operation operation = new Intent.OperationBuilder().withDeviceId("")
            .withBundleName(getBundleName())
            .withAbilityName("com.huawei.abilityintent.SecondAbility")
            .build();
    intent.setOperation(operation);
    intent.setParam("key", "我从 MainAbility 进到了 SecondAbility");
    startAbilityForResult(intent, 1);
}
```

在同目录下的 SecondAbilitySlice.java 文件中，对从 MainAbility 中获取到的信息通过 Text 控件进行展示，并且通过按钮单击事件完成 terminate()操作，相关代码如下。

```
@Override
public void onStart(Intent intent) {
    super.onStart(intent);
    super.setUIContent(ResourceTable.Layout_ability_second);
     Component secondBackFirstButton = findComponentById(ResourceTable.Id_second_back_
first);
    secondBackFirstButton.setClickedListener(component -> terminate());
    Text showParametersText = (Text) findComponentById(ResourceTable.Id_second_text);
    showParametersText.setText(intent.getStringParam("key"));
}
```

同时，在 SecondAbility 页面中设置返回 MainAbility 需要的数据。在 SecondAbility.java 中，调用 setResult()设置返回结果，相关代码如下。

```
@Override
protected void onActive() {
    super.onActive();
    Intent intent = new Intent();
    intent.setParam("key", "我从 SecondAbility 跳回来啦");
    setResult(0, intent);
}
```

如果是在 AbilitySlice 中，也可以通过如下方式完成 setResult()设置返回结果，即 getAbility().setResult(0,intent)。

在 MainAbilitySlice.java 中增加如下代码，完成返回结果的回调函数 onAbilityResult()，将返回结果显示在页面上。

```
@Override
protected void onAbilityResult(int requestCode, int resultCode, Intent resultData) {
    if (resultCode != 0 || resultData == null) {
        return;
    }
    String result = resultData.getStringParam("key");
    backValueText.setText(result);
}
```

总之，Page 内可以使用 present()或 presentForResult()实现导航，使用 presentForResult()时可以获得从导航目标 AbilitySlice 返回时的返回结果。返回结果由导航目标 AbilitySlice 在其生命周期内通过 setResult()进行设置。当用户从导航目标 AbilitySlice 返回时，系统将回调 onResult()接收和处理返回结果，需要重写该方法。

Page 间的导航可以使用 startAbility()或 startAbilityForResult()方法，使用 startAbilityForResult()方法时可以获得导航目标 Ability 返回时的返回结果。在导航目标 Ability 中调用 setResult()设置返回结果，获得返回结果的回调为 onAbilityResult()。

另外一个实用示例是跨设备迁移支持将 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需求。通过两个页面、两台设备实现在不同设备上的迁移。

3.3 公共事件与通知开发

HarmonyOS 通过 CES(Common Event Service，公共事件服务)为应用程序提供订阅、发布、退订公共事件的能力，通过 ANS(Advanced Notification Service，通知增强服务)系统服务为应用程序提供发布通知的能力。

公共事件可分为系统公共事件和自定义公共事件。系统公共事件：将收集到的事件信息，根据系统策略发送给订阅该事件的用户程序。包括如下内容：终端设备用户可感知的亮灭屏事件，以及系统关键服务发布的系统事件(如 USB 插拔、网络连接、系统升级)等。自定义公共事件：应用自定义一些公共事件处理业务逻辑。

通知提供应用的即时/通信消息，用户可以直接删除或单击通知触发进一步的操作。IntentAgent 封装了一个指定行为的 Intent，可以通过 IntentAgent 启动 Ability 和发布公共事件。应用如果需要接收公共事件，需要订阅相应的事件。

目前，公共事件与通知开发具有一定的约束与限制，具体内容如下。

公共事件的约束与限制：目前公共事件仅支持动态订阅，不支持多用户，部分系统事件需要具有指定的权限。ThreadMode 表示线程模型，目前仅支持 HANDLER 模式，即在当前 UI 线程上执行回调函数。DeviceId 用来指定订阅本地公共事件还是远端公共事件。DeviceId 为 Null、空字符串或本地设备 DeviceId 时，表示订阅本地公共事件，否则表示订阅

远端公共事件。

通知的约束与限制：通知目前支持 6 种样式：普通文本、长文本、图片、社交、多行文本和媒体样式。创建通知时必须包含一种样式。注意：通知支持快捷回复。

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

3.3.1 公共事件开发

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

1. 公共事件开发接口关系

公共事件相关基础类包含 CommonEventData、CommonEventPublishInfo、CommonEventSubscribeInfo、CommonEventSubscriber 和 CommonEventManager，公共事件基础类关系如图 3-8 所示。

图 3-8 公共事件基础类关系

1）CommonEventData

CommonEventData 封装公共事件相关信息。用于在发布、分发和接收时处理数据。在构造 CommonEventData 对象时，相关参数需要注意以下事项：Code 为有序公共事件的结果码，Data 为有序公共事件的结果数据，仅用于有序公共事件场景。Intent 不允许为空，否则发布公共事件失败，主要接口及功能描述如表 3-3 所示。

表 3-3 CommonEventData 的主要接口及功能描述

主 要 接 口	功 能 描 述
CommonEventData()	创建公共事件数据
CommonEventData(Intent intent)	创建公共事件数据指定 Intent
CommonEventData(Intent intent,int code,String data)	创建公共事件数据，指定 Intent、Code 和 Data

续表

主 要 接 口	功 能 描 述
getIntent()	获取公共事件 Intent
setCode(int code)	设置有序公共事件的结果码
getCode()	获取有序公共事件的结果码
setData(String data)	设置有序公共事件的详细结果数据
getData()	获取有序公共事件的详细结果数据

2) CommonEventPublishInfo

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

有序公共事件：主要场景是多个订阅者有依赖关系或者对处理顺序有要求，例如高优先级订阅者可修改公共事件内容或处理结果，包括终止公共事件处理；或者低优先级订阅者依赖高优先级的处理结果等。订阅者可以通过 CommonEventSubscribeInfo.setPriority()方法指定优先级，默认为 0，优先级范围为[−1000，1000]，值越大优先级越高。

黏性公共事件：指公共事件的订阅动作在公共事件发布之后进行，订阅者也能收到公共事件类型。主要场景是由公共事件服务记录某些系统状态，例如蓝牙、WLAN、充电等事件和状态。不使用黏性公共事件机制时，应用可以通过直接访问系统服务获取该状态；在状态变化时，系统服务、硬件需要提供类似 observer 等方式通知应用。发布黏性公共事件可以通过 setSticky()方法设置，发布黏性公共事件需要申请如下权限。在 config.json 文件中的 reqPermissions 实现权限申请，字段说明如下。

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

CommonEventPublishInfo 主要接口及功能描述如表 3-4 所示。

表 3-4 CommonEventPublishInfo 主要接口及功能描述

主要接口	功能描述
CommonEventPublishInfo()	创建公共事件信息
CommonEventPublishInfo(CommonEventPublishInfo publishInfo)	复制公共事件信息
setSticky(boolean sticky)	设置公共事件的黏性属性
setOrdered(boolean ordered)	设置公共事件的有序属性
setSubscriberPermissions(String[] subscriberPermissions)	设置公共事件订阅者的权限,多参数仅第一个生效

3) CommonEventSubscribeInfo

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

线程模式(ThreadMode):设置订阅者的回调方法执行的线程模式。ThreadMode 有 HANDLER、POST、ASYNC 和 BACKGROUND 4 种模式,目前只支持 HANDLER 模式。

HANDLER:在 Ability 的主线程执行。

POST:在事件分发线程执行。

ASYNC:在一个新创建的异步线程执行。

BACKGROUND:在后台线程执行。

主要接口及功能描述如表 3-5 所示。

表 3-5 CommonEventSubscribeInfo 的主要接口及功能描述

主要接口	功能描述
CommonEventSubscribeInfo(MatchingSkills matchingSkills)	创建公共事件订阅器,指定 matchingSkills
CommonEventSubscribeInfo(CommonEventSubscribeInfo)	复制公共事件订阅器对象
setPriority(int priority)	设置优先级,用于有序公共事件
setThreadMode(ThreadMode threadMode)	指定订阅者回调函数运行在哪个线程上
setPermission(String permission)	设置发布者必须具备的权限
setDeviceId(String deviceId)	指定订阅哪台设备的公共事件

4) CommonEventSubscriber

CommonEventSubscriber. AsyncCommonEventResult 类处理有序公共事件异步执行。目前只能通过调用 CommonEventManager 的 subscribeCommonEvent()进行订阅。主要接口及功能描述如表 3-6 所示。

表 3-6 CommonEventSubscriber 的主要接口及功能描述

主要接口	功能描述
CommonEventSubscriber (CommonEventSubscribeInfo subscribeInfo)	构造公共事件订阅者实例
onReceiveEvent(CommonEventData data)	由开发者实现,在接收到公共事件时被调用

续表

主 要 接 口	功 能 描 述
AsyncCommonEventResult goAsyncCommonEvent()	设置有序公共事件异步执行
setCodeAndData(int code,String data)	设置有序公共事件异步结果
setData(String data)	设置有序公共事件异步结果数据
setCode(int code)	设置有序公共事件异步结果码
getData()	获取有序公共事件异步结果数据
getCode()	获取有序公共事件异步结果码
abortCommonEvent()	取消当前公共事件,仅对有序公共事件有效,取消后,公共事件不再向下一个订阅者传递
getAbortCommonEvent()	获取当前有序公共事件是否取消的状态
clearAbortCommonEvent()	清除当前有序公共事件 Abort 状态
isOrderedCommonEvent()	查询当前公共事件是否为有序公共事件
isStickyCommonEvent()	查询当前公共事件是否为黏性公共事件

5) CommonEventManager

CommonEventManager 是为应用提供订阅、退订和发布公共事件的静态接口类,方法及功能描述如表 3-7 所示。

表 3-7 CommonEventManager 的方法及功能描述

方 法	功 能 描 述
publishCommonEvent(CommonEventData eventData)	发布公共事件
publishCommonEvent (CommonEventData event, CommonEventPublishInfo publishInfo)	发布公共事件,指定发布信息
publishCommonEvent (CommonEventData event, CommonEventPublishInfo publishInfo,CommonEventSubscriber resultSubscriber)	发布有序公共事件,指定发布信息和最后一个接收者
subscribeCommonEvent(CommonEventSubscriber subscriber)	订阅公共事件
unsubscribeCommonEvent(CommonEventSubscriber subscriber)	退订公共事件

2. 发布公共事件

开发者可以发布 4 种公共事件:无序公共事件、携带权限公共事件、有序公共事件和黏性公共事件。

1) 发布无序公共事件

构造 CommonEventData 对象,设置 Intent,通过构造 operation 对象把需要发布的公共事件信息传入 Intent 对象。

调用 CommonEventManager.publishCommonEvent(CommonEventData) 接口发布公共事件,相关代码如下。

```
try {
    Intent intent = new Intent();
    Operation operation = new Intent.OperationBuilder()
```

```
            .withAction("com.my.test")
            .build();
    intent.setOperation(operation);
    CommonEventData eventData = new CommonEventData(intent);
    CommonEventManager.publishCommonEvent(eventData);
    HiLog.info(LABEL_LOG, "Publish succeeded");
} catch (RemoteException e) {
    HiLog.error(LABEL_LOG, "Exception occurred during publishCommonEvent invocation.");
}
```

2）发布携带权限公共事件

构造 CommonEventPublishInfo 对象，设置订阅者的权限。订阅者在 config.json 中申请所需的权限，发布带权限公共事件相关代码如下。

```
Intent intent = new Intent();
Operation operation = new Intent.OperationBuilder()
        .withAction("com.my.test")
        .build();
intent.setOperation(operation);
CommonEventData eventData = new CommonEventData(intent);
CommonEventPublishInfo publishInfo = new CommonEventPublishInfo();
String[] permissions = {"com.example.MyApplication.permission"};
publishInfo.setSubscriberPermissions(permissions); //设置权限
try {
    CommonEventManager.publishCommonEvent(eventData, publishInfo);
    HiLog.info(LABEL_LOG, "Publish succeeded");
} catch (RemoteException e) {
    HiLog.error(LABEL_LOG, "Exception occurred during publishCommonEvent invocation.");
}
```

3）发布有序公共事件

构造 CommonEventPublishInfo 对象，通过 setOrdered(true)指定公共事件属性为有序公共事件，也可以指定一个最后的公共事件接收者，相关代码如下。

```
CommonEventSubscriber resultSubscriber = new MyCommonEventSubscriber();
CommonEventPublishInfo publishInfo = new CommonEventPublishInfo();
publishInfo.setOrdered(true); //设置属性为有序公共事件
try {
    CommonEventManager.publishCommonEvent(eventData, publishInfo, resultSubscriber);
                                    //指定 resultSubscriber 为有序公共事件最后一个接收者
} catch (RemoteException e) {
    HiLog.error(LABEL_LOG, "Exception occurred during publishCommonEvent invocation.");
}
```

4）发布黏性公共事件

构造 CommonEventPublishInfo 对象，通过 setSticky(true)指定公共事件属性为黏性

公共事件。发布者在 config.json 中申请发布黏性公共事件所需的权限，相关代码如下。

```
CommonEventPublishInfo publishInfo = new CommonEventPublishInfo();
publishInfo.setSticky(true); //设置属性为黏性公共事件
try {
    CommonEventManager.publishCommonEvent(eventData, publishInfo);
} catch (RemoteException e) {
    HiLog.error(LABEL, "Exception occurred during publishCommonEvent invocation.");
}
```

3. 订阅公共事件

订阅公共事件步骤如下。

（1）创建 CommonEventSubscriber 派生类，在 onReceiveEvent()回调函数中处理公共事件。此处不能执行耗时操作，否则会阻塞 UI 线程，产生用户单击没有反应等异常。

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

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

```
String event = "com.my.test";
MatchingSkills matchingSkills = new MatchingSkills();
matchingSkills.addEvent(event); //自定义事件
matchingSkills.addEvent(CommonEventSupport.COMMON_EVENT_SCREEN_ON);
//亮屏事件
CommonEventSubscribeInfo subscribeInfo = new CommonEventSubscribeInfo(matchingSkills);
MyCommonEventSubscriber subscriber = new MyCommonEventSubscriber(subscribeInfo);
try {
    CommonEventManager.subscribeCommonEvent(subscriber);
} catch (RemoteException e) {
    HiLog.error(LABEL, "Exception occurred during subscribeCommonEvent invocation.");
}
```

如果订阅拥有指定权限应用发布的公共事件，发布者需要在 config.json 中申请权限。如果订阅的公共事件是有序的，可以调用 setPriority()指定优先级，相关代码如下。

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

```
subscribeInfo.setPriority(100); //设置优先级取值范围[ - 1000,1000],值默认为 0
MyCommonEventSubscriber subscriber = new MyCommonEventSubscriber(subscribeInfo);
try {
     CommonEventManager.subscribeCommonEvent(subscriber);
} catch (RemoteException e) {
     HiLog.error(LABEL, "Exception occurred during subscribeCommonEvent invocation.");
}
```

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

```
EventRunner runner = EventRunner.create();                    //EventRunner 创建新线程,将耗时的操
                                                              //作放到新的线程上执行
MyEventHandler myHandler = new MyEventHandler(runner);        //MyEventHandler 为 EventHandler 的派
                                                              //生类,在不同线程间分发、处理事件和
                                                              //Runnable 任务
@Override
public void onReceiveEvent(CommonEventData commonEventData){
    final AsyncCommonEventResult result = goAsyncCommonEvent();
    Runnable task = new Runnable() {
        @Override
        public void run() {
            ........                                          //待执行的操作,由开发者定义
            result.finishCommonEvent();                       //调用 finish 结束异步操作
        }
    };
    myHandler.postTask(task);
}
```

4. 退订公共事件

在 Ability 的 onStop()中调用 CommonEventManager.unsubscribeCommonEvent()方法退订公共事件。调用后,之前订阅的所有公共事件均被退订,相关代码如下。

```
try {
    CommonEventManager.unsubscribeCommonEvent(subscriber);
} catch (RemoteException e) {
    HiLog.error(LABEL, "Exception occurred during unsubscribeCommonEvent invocation.");
}
```

针对公共事件开发,CommonEvent 演示了公共事件的订阅、发布和退订,示例工程参考地址为:https://gitee.com/openharmony/app_samples/tree/master/ability/CommonEvent。

3.3.2 通知开发

HarmonyOS 提供通知功能,即在一个应用的 UI 之外显示的消息,主要是提醒用户有

来自该应用中的信息。当应用向系统发出通知时，它将先以图标的形式显示在通知栏中，用户可以下拉通知栏查看详细信息。

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

1. 通知开发接口关系

通知相关基础类包含 NotificationSlot、NotificationRequest 和 NotificationHelper，如图 3-9 所示。

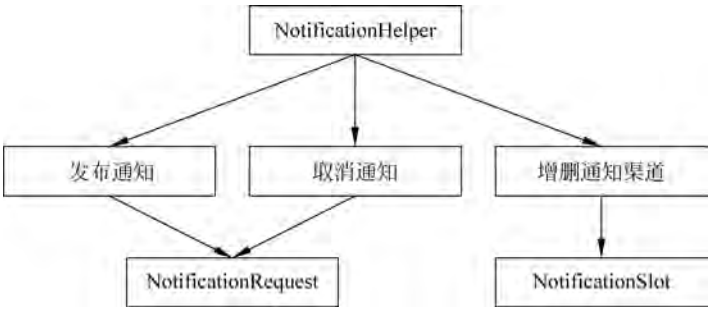

图 3-9　通知基础类关系图

1）NotificationSlot

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

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

不指定 NotificationSlot 时，当前通知会使用默认的 NotificationSlot，默认的 NotificationSlot 优先级为 LEVEL_DEFAULT，主要接口及功能描述如表 3-8 所示。

表 3-8　NotificationSlot 的主要接口及功能描述

主 要 接 口	功 能 描 述
NotificationSlot(String id,String name,int level)	构造 NotificationSlot
setLevel(int level)	设置 NotificationSlot 的级别
setName(String name)	设置 NotificationSlot 的命名
setDescription(String description)	设置 NotificationSlot 的描述信息
enableBypassDnd(boolean bypassDnd)	设置是否绕过系统的免打扰模式
setEnableVibration(boolean vibration)	设置收到通知时是否使能振动
setEnableLight(boolean isLightEnabled)	设置收到通知时是否开启呼吸灯，前提是当前硬件支持呼吸灯
setLedLightColor(int color)	设置收到通知时的呼吸灯颜色
setSlotGroup(String groupId)	绑定当前 NotificationSlot 到一个 NotificationSlot 组

NotificationSlot 的级别由低到高支持如下几种：LEVEL_NONE 表示通知不发布；LEVEL_MIN 表示通知可以发布，但不在状态栏显示，不自动弹出，无提示音，该级别不适用于前台服务的场景；LEVEL_LOW 表示通知发布后在状态栏显示，不自动弹出，无提示音；LEVEL_DEFAULT 表示通知发布后在状态栏显示，不自动弹出，触发提示音；LEVEL_HIGH 表示通知发布后在状态栏显示，自动弹出，触发提示音。

2）NotificationRequest

NotificationRequest 用于设置具体的通知对象，包括设置通知的属性，例如通知的分发时间、小图标、大图标、自动删除等参数，以及设置具体的通知类型，例如普通文本、长文本等，主要接口及功能描述如表 3-9 所示。

表 3-9 NotificationRequest 的主要接口及功能描述

主要接口	功能描述
NotificationRequest()	构建一个通知
NotificationRequest(int notificationId)	构建一个通知，指定通知的 ID。通知的 ID 在应用内具有唯一性，如果不指定，则默认为 0
setNotificationId(int notificationId)	设置当前通知 ID
setAutoDeletedTime(long time)	设置通知自动取消的时间戳
setContent(NotificationRequest. NotificationContent content)	设置通知的具体内容
setDeliveryTime(long deliveryTime)	设置通知分发的时间戳
setSlotId(String slotId)	设置通知的 NotificationSlot ID
setTapDismissed(boolean tapDismissed)	设置通知在用户单击后是否自动取消
setLittleIcon(PixelMap smallIcon)	设置通知的小图标，在通知左上角显示
setBigIcon(PixelMap bigIcon)	设置通知的大图标，在通知的右侧显示
setGroupValue(String groupValue)	设置分组通知，相同分组的通知在通知栏显示时，将会折叠在一组应用中显示
addActionButton(NotificationActionButton actionButton)	设置通知添加 ActionButton
setIntentAgent(IntentAgent agent)	设置通知承载指定的 IntentAgent，在通知中实现即将触发的事件

目前支持 6 种通知类型：普通文本 NotificationNormalContent、长文本 NotificationLongTextContent、图片 NotificationPictureContent、多行 NotificationMultiLineContent、社交 NotificationConversationalContent、媒体 NotificationMediaContent，主要接口及功能描述如表 3-10 所示。

表 3-10 通知类型的主要接口及功能描述

类名	主要接口	功能描述
NotificationNormalContent	setTitle(String title)	设置通知标题
	setText(String text)	设置通知内容
	setAdditionalText (String additionalText)	设置通知次要内容，是对通知内容的补充

续表

类　名	主要接口	功能描述
NotificationNormalContent	setBriefText(String briefText)	设置通知概要内容，是对通知内容的总结
	setExpandedTitle (String expandedTitle)	设置附加图片通知展开时的标题
	setBigPicture (PixelMap bigPicture)	设置通知的图片内容，附加在setText(String text)下方
NotificationLongTextContent	setLongText(String longText)	设置通知的长文本
NotificationConversationalContent	setConversationTitle (String conversationTitle)	设置社交通知的标题
	addConversationalMessage (ConversationalMessage message)	通知添加一条消息
NotificationMultiLineContent	addSingleLine(String line)	在当前通知中添加一行文本
NotificationMediaContent	setAVToken (AVToken avToken)	将媒体通知绑定指定的 AVToken
	setShownActions(int[] actions)	设置媒体通知待展示的按钮

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

3) NotificationHelper

NotificationHelper 封装了发布、更新、删除通知等静态方法，主要接口及功能描述如表 3-11 所示。

表 3-11　NotificationHelper 的主要接口及功能描述

主要接口	功能描述
publishNotification(NotificationRequest request)	发布一条通知
publishNotification(String tag,NotificationRequest request)	发布一条带 TAG 的通知
cancelNotification(int notificationId)	取消指定通知
cancelNotification(String tag,int notificationId)	取消指定带 TAG 的通知
cancelAllNotifications()	取消之前发布的所有通知
addNotificationSlot(NotificationSlot slot)	创建一个 NotificationSlot
getNotificationSlot(String slotId)	获取 NotificationSlot
removeNotificationSlot(String slotId)	删除一个 NotificationSlot
getActiveNotifications()	获取当前应用发的活跃通知
getActiveNotificationNums()	获取系统中当前应用发的活跃通知数量
setNotificationBadgeNum(int num)	设置通知的角标
setNotificationBadgeNum()	设置当前应用中活跃状态通知的数量在角标显示

2. 通知开发步骤

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

1）创建 NotificationSlot

NotificationSlot 可以设置公共通知的振动、重要级别等，并通过调用 NotificationHelper.addNotificationSlot()发布 NotificationSlot 对象，相关代码如下。

```
NotificationSlot slot = new NotificationSlot("slot_001", "slot_default", NotificationSlot.
LEVEL_MIN); //创建 notificationSlot 对象
slot.setDescription("NotificationSlotDescription");
slot.setEnableVibration(true);                    //设置振动提醒
slot.setEnableLight(true);                        //设置开启呼吸灯提醒
slot.setLedLightColor(Color.RED.getValue());      //设置呼吸灯的提醒颜色
try {
    NotificationHelper.addNotificationSlot(slot);
} catch (RemoteException ex) {
    HiLog.error(LABEL, "Exception occurred during addNotificationSlot invocation.");
}
```

2）发布通知

发布通知步骤如下。

（1）构建 NotificationRequest 对象，应用发布通知前，通过 NotificationRequest 的 setSlotId()方法与 NotificationSlot 绑定，使通知在发布后具备该对象的特征，相关代码如下。

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

（2）调用 setContent()设置通知的内容，相关代码如下。

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

（3）调用 publishNotification()发布通知，相关代码如下。

```
try {
    NotificationHelper.publishNotification(request);
} catch (RemoteException ex) {
    HiLog.error(LABEL, "Exception occurred during publishNotification invocation.");
}
```

3）取消通知

取消通知分为取消指定单条通知和取消所有通知，应用只能取消自己发布的通知。

(1) 调用 cancelNotification()取消指定的单条通知,相关代码如下。

```
int notificationId = 1;
try {
    NotificationHelper.cancelNotification(notificationId);
} catch (RemoteException ex) {
    HiLog.error(LABEL, "Exception occurred during cancelNotification invocation.");
}
```

(2) 调用 cancelAllNotifications()取消所有通知,相关代码如下。

```
try {
    NotificationHelper.cancelAllNotifications();
} catch (RemoteException ex) {
    HiLog.error(LABEL, "Exception occurred during cancelAllNotifications invocation.");
}
```

针对通知开发和通知功能,在一个应用的 UI 之外显示的消息,主要是提醒用户有来自该应用中的信息,演示如何发布通知和取消通知。示例工程参考地址为: https://gitee.com/openharmony/app_samples/tree/master/ability/Notification。

3.3.3 IntentAgent 开发

IntentAgent 封装了一个指定行为的 Intent,可以通过 triggerIntentAgent 接口主动触发,也可以与通知绑定被动触发。具体行为包括启动 Ability 和发布公共事件。例如,在单击通知后跳转到一个新的 Ability,不单击则不触发。

1. IntentAgent 接口关系

IntentAgent 相关基础类包括 IntentAgentHelper、IntentAgentInfo、IntentAgentConstant 和 TriggerInfo,如图 3-10 所示。

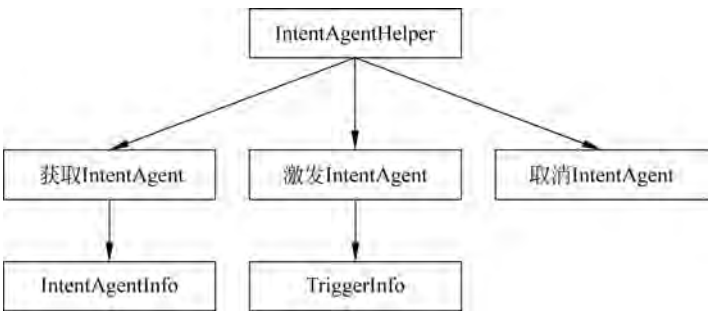

图 3-10 IntentAgent 基础类关系图

1) IntentAgentHelper

IntentAgentHelper 封装了获取、激发、取消 IntentAgent 等静态方法,主要接口及功能描述如表 3-12 所示。

表 3-12 IntentAgentHelper 的主要接口及功能描述

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

2) IntentAgentInfo

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

requestCode：使用者定义的一个私有值。

operationType：IntentAgentConstant.OperationType 枚举中的值。

flags：IntentAgentConstant.Flags 枚举中的值。

intents：将被执行的意图列表。operationType 的值为 START_ABILITY、START_SERVICE 和 SEND_COMMON_EVENT 时，intents 列表只允许包含一个 Intent；operationType 的值为 START_ABILITIES 时，intents 列表允许包含多个 Intent。

extraInfo：表明如何启动一个有页面的 ability，可以为 null，只在 operationType 的值为 START_ABILITY 和 START_ABILITIES 时有意义。

3) IntentAgentConstant

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

IntentAgentConstant.OperationType 类的枚举值如下。

(1) UNKNOWN_TYPE：不识别的类型。

(2) START_ABILITY：开启一个有页面的 Ability。

(3) START_ABILITIES：开启多个有页面的 Ability。

(4) START_SERVICE：开启一个无页面的 Ability。

(5) SEND_COMMON_EVENT：发送一个公共事件。

IntentAgentConstant.Flags 类的枚举值如下。

(1) ONE_TIME_FLAG：IntentAgent 仅能使用一次，只在 operationType 的值为 START_ABILITY、START_SERVICE 和 SEND_COMMON_EVENT 时有意义。

(2) NO_BUILD_FLAG：如果描述 IntentAgent 对象不存在，则不创建它，直接返回 null，只在 operationType 的值为 START_ABILITY、START_SERVICE 和 SEND_COMMON_EVENT

时有意义。

(3) CANCEL_PRESENT_FLAG：在生成一个新的 IntentAgent 对象前取消已存在的一个 IntentAgent 对象，只在 operationType 的值为 START_ABILITY、START_SERVICE 和 SEND_COMMON_EVENT 时有意义。

(4) UPDATE_PRESENT_FLAG：使用新的 IntentAgent 额外数据替换已存在 IntentAgent 中的额外数据，只在 operationType 的值为 START_ABILITY、START_SERVICE 和 SEND_COMMON_EVENT 时有意义。

(5) CONSTANT_FLAG：IntentAgent 是不可变的。

(6) REPLACE_ELEMENT：当前 Intent 中的 element 属性可被 IntentAgentHelper.triggerIntentAgent()中 Intent 的 element 属性取代。

(7) REPLACE_ACTION：当前 Intent 中的 Action 属性可被 IntentAgentHelper.triggerIntentAgent()中 Intent 的 Action 属性取代。

(8) REPLACE_URI：当前 Intent 中的 URI 属性可被 IntentAgentHelper. triggerIntentAgent()中 Intent 的 URI 属性取代。

(9) REPLACE_ENTITIES：当前 Intent 中的 entities 属性可被 IntentAgentHelper.triggerIntentAgent()中 Intent 的 entities 属性取代。

(10) REPLACE_BUNDLE：当前 Intent 中的 bundleName 属性可被 IntentAgentHelper.triggerIntentAgent()中 Intent 的 bundleName 属性取代。

4) TriggerInfo

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

(1) String permission：IntentAgent 的接收者权限名称，只在 operationType 的值为 SEND_COMMON_EVENT 时，参数才有意义。

(2) IntentParams extraInfo：激发 IntentAgent 时用户自定义的额外数据。

(3) Intent intent：额外的 Intent。如果 IntentAgentInfo 成员变量 flags 包含 CONSTANT_FLAG，则忽略该参数；如果 flags 包含 REPLACE_ELEMENT、REPLACE_ACTION、REPLACE_URI、REPLACE_ENTITIES 或 REPLACE_BUNDLE，则使用额外 Intent 的 element、action、uri，entities 或 bundleName 属性替换原始 Intent 中对应的属性。如果 Intent 为空，则不替换原始 Intent 的属性。

(4) int code：提供给 IntentAgent 目标的结果码。

2. IntentAgent 开发步骤

IntentAgent 开发步骤如下。

1) 获取 IntentAgent 代码

```
//指定要启动的 Ability 的 BundleName 和 AbilityName 字段
//将 Operation 对象设置到 Intent 中
```

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

2）通知中添加 IntentAgent 的代码

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

3）主动激发 IntentAgent 的代码

```
int code = 100;
IntentAgentHelper.triggerIntentAgent(this, agent, null, null, new TriggerInfo(null, null,
null, code));
```

针对 IntentAgent 开发指导，演示如何通过 IntentAgent 启动 Ability 和发布公共事件。示例工程参考地址为：https://gitee.com/openharmony/app_samples/tree/master/ability/IntentAgent。

3.3.4 后台代理定时提醒开发

在应用开发时，可以调用后台代理提醒类 ReminderRequest 创建定时提醒，包括倒计时、日历和闹钟 3 种类型。使用后台代理提醒功能后，应用可以被冻结或退出，计时和弹出

提醒的功能将被后台系统服务代理。

1. 后台代理定时提醒接口关系

ReminderRequest 涉及的基础类包括 ReminderHelper、ReminderRequestTimer、ReminderRequestCalendar 和 ReminderRequestAlarm，如图 3-11 所示。

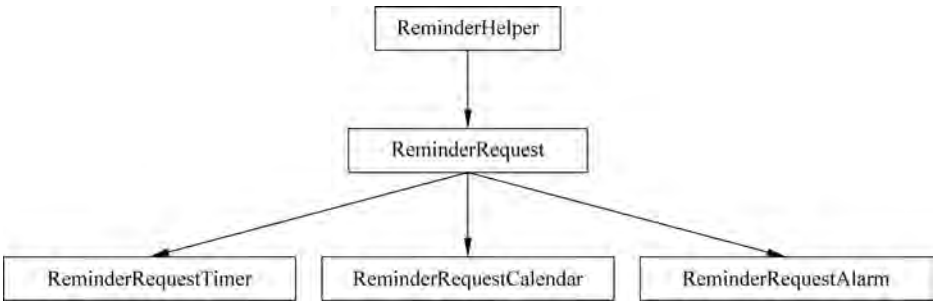

图 3-11 ReminderRequest 基础类关系图

1) ReminderHelper

ReminderHelper 是封装发布、取消提醒类通知的方法，其主要接口及功能描述如表 3-13 所示。

表 3-13 ReminderHelper 的主要接口及功能描述

主要接口	功能描述
public static int publishReminder(ReminderRequest reminderReq) throws RemoteException, ReminderManager.AppLimitExceedsException, ReminderManager.SysLimitExceedsException	发布一个定时提醒类通知 ReminderManager.AppLimitExceedsException 系统中保存的当前应用有效的提醒个数超出最大限制数量 30 个时抛出(不包括已经超时，即后续不会再提醒的实例) ReminderManager.SysLimitExceedsException 中保存的整个系统有效的提醒个数超出最大限制数量 2000 个时抛出(不包括已经超时，即后续不再提醒的实例)
public static void addNotificationSlot(NotificationSlot slot) throws RemoteException	注册一个提醒类需要使用的 NotificationSlot
public static void cancelReminder(int reminderId) throws RemoteException	取消一个指定的提醒类通知 (reminderId 从 publishReminder 的返回值获取)
public static void removeNotificationSlot(String slotId) throws RemoteException	删除一个 slot 实例
public static List<ReminderRequest> getValidReminders() throws RemoteException	获取当前应用设置的所有有效提醒
public static void cancelAllReminders() throws RemoteException	取消当前应用设置的所有提醒

2）ReminderRequest

ReminderRequest是后台代理提醒类基类，封装提醒相关的属性查询和设置的操作，主要接口及功能描述如表3-14所示。

表3-14 ReminderRequest的主要接口及功能描述

主要接口	功能描述
public long getRingDuration()	获取设置的提醒时长，单位为s，例如设置开始响铃后的时长
public int getSnoozeTimes()	获取设置的延迟提醒次数
public long getTimeInterval()	获取设置的延迟提醒间隔
public ReminderRequest setRingDuration (long ringDurationInSeconds)	设置提醒时长，单位为s，例如设置开始响铃后的时长
public ReminderRequest setSnoozeTimes (int snoozeTimes)	设置延迟提醒的次数（倒计时设置延迟提醒无效）
public ReminderRequest setTimeInterval (long timeIntervalInSeconds)	设置延迟提醒的时间间隔（倒计时设置延迟提醒无效）
public ReminderRequest setActionButton (String title,int type)	在提醒弹出的通知界面中添加NotificationActionButton
public ReminderRequest setIntentAgent (String pkgName,String abilityName)	设置单击通知信息后需要跳转目标包的信息
public ReminderRequest setMaxScreenIntentAgent (String pkgName,String abilityName)	设置提醒到达时跳转的目标包。如果设备正在使用中，则弹出一个通知框
public String getTitle()	获取提醒的标题
public ReminderRequest setTitle(String title)	设置提醒的标题
public String getContent()	获取提醒的内容
public ReminderRequest setContent(String content)	设置提醒的内容
public String getExpiredContent()	获取提醒“过期”时显示的扩展内容
public ReminderRequest setExpiredContent (String expiredContent)	设置提醒“过期”时显示的扩展内容
public String getSnoozeContent()	获取提醒“再响”时显示的扩展内容
public ReminderRequest setSnoozeContent (String snoozeContent)	设置提醒“再响”时显示的扩展内容
public int getNotificationId()	获取提醒使用notificationRequest的ID，参见NotificationRequest.setNotificationId(int id)
public ReminderRequest setNotificationId (int notificationId)	设置提醒使用notificationRequest的ID
public String getSlotId()	获取提醒使用的slot ID
public String SetSlotId(String slotId)	设置提醒使用的slot ID

3）ReminderRequestTimer

ReminderRequestTimer 为提醒类子类，用于倒计时提醒，主要接口及功能描述如表 3-15 所示。

表 3-15 ReminderRequestTimer 的主要接口及功能描述

主要接口	功能描述
public ReminderRequestTimer(long triggerTimeInSeconds)	创建一个倒计时提醒实例，经过指定时间后触发提醒

4）ReminderRequestCalendar

ReminderRequestCalendar 为提醒类子类，用于日历类提醒。可以指定提醒时间精确为某年某月某日某时某分，也可以指定哪些月份的哪些天的同一时间重复提醒，主要接口及功能描述如表 3-16 所示。

表 3-16 ReminderRequestCalendar 的主要接口及功能描述

主要接口	功能描述
public ReminderRequestCalendar(LocalDateTime dateTime,int[] repeatMonths,int[] repeatDays)	创建一个日历类提醒实例，在指定的时间触发提醒

5）ReminderRequestAlarm

ReminderRequestAlarm 为提醒类子类，用于闹钟类提醒。可以指定几点几分提醒，或者每周哪几天指定时间提醒。主要接口及功能描述如表 3-17 所示。

表 3-17 ReminderRequestAlarm 的主要接口及功能描述

主要接口	功能描述
public ReminderRequestAlarm(int hour, int minute,int[] daysOfWeek)	创建一个闹钟类提醒实例，在指定的时间触发提醒

2. 后台代理定时提醒开发步骤

开发步骤如下。

1）声明使用权限

使用后台代理提醒需要在配置文件中声明需要此权限。

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

2）创建提醒步骤

创建提醒共 8 个步骤，具体方法如下。

（1）设置渠道信息。

```
NotificationSlot slot = new NotificationSlot("slot_id", "slot_name", NotificationSlot.LEVEL_
HIGH);
```

```
slot.setEnableLight(false);
slot.setEnableVibration(true);
```

(2) 向代理服务添加渠道对象。

```
try {
    ReminderHelper.addNotificationSlot(slot);
} catch (RemoteException e) {
    e.printStackTrace();
}
```

(3) 创建提醒类通知对象。

```
int[] repeatDay = {};
ReminderRequest reminder = new ReminderRequestAlarm(10, 30, repeatDay);
```

(4) 设置提醒内容。

```
reminder.setTitle("set title here").setContent("set content here");
```

(5) 设置提醒时长等属性。

```
reminder.setSnoozeTimes(1).setTimeInterval(5 * 60).setRingDuration(10);
```

(6) 设置 IntentAgent(假设包名为 com.ohos.aaa,Ability 类名为 FirstAbility)。

```
reminder.setIntentAgent("com.ohos.aaa", FirstAbility.class.getName());
```

(7) 设置提醒信息框中的“延迟提醒”和“关闭”按钮(可选)(ActionButton)。

```
 reminder.setActionButton("snooze", ReminderRequest.ACTION_BUTTON_TYPE_SNOOZE).
 setActionButton("close", ReminderRequest.ACTION_BUTTON_TYPE_CLOSE);
```

(8) 发布提醒类通知。

```
try {
    ReminderHelper.publishReminder(reminder);
} catch (ReminderManager.AppLimitExceedsException e) {
    e.printStackTrace();
} catch (ReminderManager.SysLimitExceedsException e) {
    e.printStackTrace();
} catch (RemoteException e) {
    e.printStackTrace();
}
```

3) 创建倒计时提醒示例

```
//经过 1 分钟后提醒
ReminderRequest reminderRequestTimer = new ReminderRequestTimer(60);
```

4）创建一次性日历提醒

```
//2021 年 3 月 2 日 14 点 30 分提醒
int[] repeatMonths = {};
int[] repeatDays = {};
ReminderRequestCalendar reminderRequestCalendar = new ReminderRequestCalendar( LocalDateTime. of
(2021, 3, 2, 14, 30), repeatMonths, repeatDays);
```

5）创建重复日历提醒

```
// 3 月份、5 月份的 9 号和 15 号 14 点 30 分提醒,延迟 10 分钟后再次提醒,默认延迟次数为 3 次
int[] repeatMonths = {3, 5};
int[] repeatDaysOfMonth = {9, 15};
ReminderRequestCalendar reminderRequestCalender = new ReminderRequestCalendar( LocalDateTime. of
(2021, 3, 2, 14, 30), repeatMonths, repeatDaysOfMonth);
reminderRequestCalender. setTimeInterval(10 * 60);
```

6）创建一次性闹钟提醒

```
// 13 点 59 分提醒,如果当前时间大于 13 点 59 分,则取后一天的 13 点 59 分
int[] repeatDay = {};
ReminderRequest reminderRequestAlarm = new ReminderRequestAlarm(13, 59, repeatDay);
```

7）创建重复的闹钟提醒

```
//每周 1、2、3、4 的 13 点 59 分提醒
int[] repeatDay = {1, 2, 3, 4};
ReminderRequest reminderRequestAlarm = new ReminderRequestAlarm(13, 59, repeatDay);
```

8）创建用于延迟提醒的 ActionButton 界面

```
reminderRequest. setActionButton("snooze", ReminderRequest. ACTION_BUTTON_TYPE_SNOOZE);
```

9）创建用于关闭提醒的 ActionButton 界面

```
reminderRequest. setActionButton("close", ReminderRequest. ACTION_BUTTON_TYPE_CLOSE);
```

notificationId 相同的不同通知请求，在通知栏展示的内容会被覆盖，对于提醒来说，可能不希望被覆盖，开发时可以注意设置不同的 notificationId。倒计时不支持持久化，系统重启后，所有倒计时失效。

3.4 后台任务调度和管控

对于有用户交互的 OS 来说，资源优先分配给与用户交互的业务进程，换句话说，在支撑 OS 运行的进程以外，用户能感知到的业务进程优先级最高，所以后台任务调度控制的范围是用户感知不到的业务进程。

HarmonyOS 将应用的资源使用生命周期划分为前台、后台和挂起 3 个阶段。前台运

行不受资源调度的约束，后台会根据应用业务的具体任务情况进行资源使用管理，在挂起状态时，会对应用的资源使用进行调度和控制约束，以保障其他体验类业务对资源的竞争使用。

后台任务调度和管控主要是对在后台状态下的资源使用进行控制，应用从前台退到后台，可能有各种业务需求，为了达到系统资源使用能效最优的目的，HarmonyOS 提供了后台任务能力。

后台任务特指应用或业务模块处于后台（无可见界面）时，需要继续执行或者后续执行的业务。为避免后台任务调度和管控对业务执行的影响，HarmonyOS 将后台任务分为 3 种类型。

无后台业务：退出后台后，无任务需要处理。

短时任务：退出后台后，如果有紧急不可推迟且短时间能完成的任务，应用退出后台要进行数据压缩，不可中断，则使用短时任务申请延迟进入挂起（Suspend）状态。

长驻任务：如果用户发起的可感知业务需要长时间后台运行，例如后台播放音乐、导航、上传下载、设备连接、VoIP 等，则使用长驻任务避免进入挂起状态。

3.4.1 短时任务

退到后台的应用有不可中断且短时间能完成的任务时，可以使用短时任务机制，该机制允许应用在后台短时间内完成任务，保障应用业务运行不受后台生命周期管理的影响。

短时任务仅针对应用的临时任务提供资源使用生命周期保障，限制单次最大使用时长为 3min，全天使用配额默认为 10min（具体时长系统根据应用场景和系统状态智能调整）。接口说明参考地址为：https://developer.harmonyos.com/cn/docs/documentation/doc-references/backgroundtaskmanager-0000001054440069。

短时任务的使用需要遵从如下约束和规则。

申请时机：允许应用在前台或退后台在被挂起之前（应用退到后台默认有 6～12s 的运行时长，具体时长由系统根据具体场景决定）申请延迟挂起，否则可能被挂起，导致申请失败。

超时：延迟挂起超时（Timeout），系统通过回调通知应用，应用需要取消对应的延迟挂起，或再次申请延迟挂起。超期不取消或不处理，该应用会被强制取消延迟挂起。

取消时机：任务完成后申请方应用主动取消延时申请，不要等到超时后被系统取消，否则会影响该应用的后台允许运行时长配额。

配额机制：为了防止应用滥用保活，或者申请后不取消，每个应用每天都会有一定配额（根据用户的使用习惯动态调整），配额消耗完不再允许申请短时任务，所以应用完成短时任务后立刻取消延时申请，避免消耗配额（此配额是指申请的时长，系统默认应用在后台运行的时间不计算在内）。

3.4.2 长驻任务

长驻任务类型给用户能直观感知到的且需要一直在后台运行的业务提供后台运行生命周期的保障。例如，业务需要在后台播放声音或者持续导航定位等，此类用户能够感知到后台业务行为，可以通过使用长驻任务对应的后台模式保障业务在后台运行，支撑应用完成在后台的业务。

1. 后台模式分类

HarmonyOS 提供 10 种后台模式，供需要在后台做长驻任务的业务使用，如表 3-18 所示。

表 3-18 长驻任务后台模式及功能描述

长驻任务后台模式	英 文 名	功 能 描 述
数据传输	data-transfer	通过网络/对端设备进行数据下载、备份、分享、传输等业务
播音	audio-playback	音频输出业务
录音	audio-recording	音频输入业务
画中画	picture-in-picture	画中画、小窗口播放视频业务
音视频通话	voip	音视频电话、VoIP 业务
导航/位置更新	location	定位、导航业务
蓝牙设备连接及传输	bluetooth-interaction	蓝牙扫描、连接、传输业务
WLAN 设备连接及传输	wifi-interaction	WLAN 扫描、连接、传输业务
屏幕抓取	screen-fetch	录屏、截屏业务
多设备互联	multiDeviceConnection	多设备互联、分布式调度和迁移等

2. 使用长驻任务

HarmonyOS 应用开发工具 DevEco Studio 在业务 ServiceAbility 创建时提供后台模式的选择，针对当前创建的 ServiceAbility 可以赋予对应的后台模式类型设置，如图 3-12 所示。

根据业务需要选择对应的后台模式后，会在应用的 config. json 文件中新创建的 ServiceAbility 组件下生成对应选择的后台模式配置，如图 3-13 所示，只有 ServiceAbility 才有对应的后台模式类型选择和配置。

在 Service 创建的方法中调用 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。完成对应的后台业务后，在销毁服务的方法中调用 cancelBackgroundRunning()方法，即可停止使用长驻任务。

3. 长驻任务使用约束

如果用户选择可感知业务，例如播音、导航、上传、下载等，触发对应后台模式，在任务启动或退入后台时，需要提醒用户。

图 3-12　后台模式类型设置

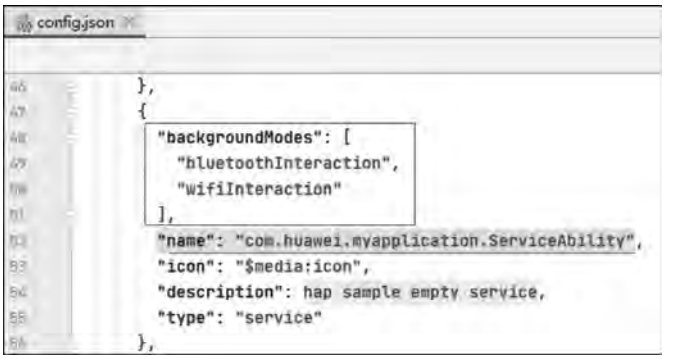

图 3-13　后台模式类型选择和配置

如果任务结束，应用会主动退出后台模式。若在后台运行期间，系统监测到应用并未使用对应后台模式的资源，则会被挂起。

避免不合理地申请后台长驻任务，长驻任务类型要与应用的类型匹配，如果执行的任务和申请的类型不匹配，也会被系统检测到并被挂起。

长驻任务是为了真正在后台长时间执行某任务，如果一个应用申请了长驻任务，但在实际运行过程中，并未真正运行或执行此类任务，也会被系统检测到并被挂起。

3.4.3　托管任务

托管任务是系统提供的一种后台代理机制。通过系统提供的代理 API 接口，用户可以把任务交由系统托管，例如后台下载、定时提醒、后台非持续定位。

1. 托管任务类型

托管任务-后台非持续定位(non-sustained Location)：如果应用未申请 location 长驻模

式，且在后台依然尝试获取位置信息，此时应用行为被视为使用非持续定位能力，后台非持续定位限制每 30min 提供一次位置信息。应用不需要高频次定位时，建议优先使用非持续定位。

托管任务-后台提醒代理(Reminder)：后台提醒代理主要提供了一种机制，使开发者在应用开发时，可以调用这些接口去创建定时提醒，包括倒计时、日历、闹钟三种提醒类型。使用后台代理提醒能力后，应用可以被冻结或退出，计时和弹出提醒的功能将被后台系统服务代理。

托管任务-后台下载代理：系统提供 DownloadSession 接口实现下载任务代理功能，应用提交下载任务后，应用被退出，下载任务仍然可以继续执行，且支持下载任务断点续传。

2. 托管任务使用约束

后台下载代理，系统会根据用户场景和设备状态，对不同的下载任务进行相应的管控，避免影响功耗和性能。

后台提醒需要申请 ohos. permission. PUBLISH_AGENT_REMINDER 权限，后台非持续定位需要申请 ohos. permission. LOCATION 和 ohos. permission. LOCATION_IN_BACKGROUND 权限。

资源滥用会影响系统性能和功耗，托管任务类型要与应用类型匹配，按照华为应用市场分类方法匹配规则，分为游戏类和应用类。游戏类没有后台长驻任务申请标准和后台托管任务申请标准，应用类的不同子类具有不同的限制约束。

原则上所有应用都会受后台任务调度和管控的约束，只是约束的时机和状态不同而已。系统提供了选择设置的方式供用户从使用需求维度干预后台任务调度和管控，以华为手机为例设置路径：手机管家→应用启动管理→选择手动管理→允许后台活动。

应用后台任务调度和管控以支撑用户使用体验为第一优先级，结合用户使用习惯、用户使用情景状态、设备模式状态、应用分类、应用资源占用统计等多个维度综合判断应用后台任务调度和管控的优先级。后台任务调度和管控机制仅对 HarmonyOS 应用进行调度和管控。

3.5 线程管理开发

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

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

如果应用的业务逻辑比较复杂，可能需要创建多个线程执行多个任务。这种情况下，代码复杂难以维护，任务与线程的交互也会更加繁杂。要解决此问题，可以使用 TaskDispatcher 分

发不同的任务。

3.5.1 线程管理开发接口关系

TaskDispatcher是一个任务分发器，它是Ability分发任务的基本接口，隐藏任务所在线程的实现细节。为保证应用有更好的响应性，需要设计任务的优先级。在UI线程上运行的任务默认以高优先级运行，如果某个任务无须等待结果，则可以用低优先级，如表3-19所示。

表3-19 线程优先级及功能描述

优 先 级	功 能 描 述
HIGH	最高任务优先级，比默认优先级、低优先级的任务有更高的概率得到执行
DEFAULT	默认任务优先级，比低优先级的任务有更高的概率得到执行
LOW	低任务优先级，比高优先级、默认优先级的任务有更低的概率得到执行

TaskDispatcher具有多种实现，每种实现对应不同的任务分发器。在分发任务时可以指定任务的优先级，由同一任务分发器分发出的任务具有相同的优先级。系统提供的任务分发器有GlobalTaskDispatcher、ParallelTaskDispatcher、SerialTaskDispatcher和SpecTaskDispatcher。

1. GlobalTaskDispatcher

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

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

2. ParallelTaskDispatcher

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

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

3. SerialTaskDispatcher

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

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

4. SpecTaskDispatcher

专有任务分发器是绑定到专有线程上的任务分发器。目前已有的专有线程为 UI 线程，通过 UITaskDispatcher 进行任务分发。

UITaskDispatcher：绑定到应用主线程的专有任务分发器，由 Ability 执行 getUITaskDispatcher()创建并返回。此分发器分发的所有任务都在主线程上按顺序执行，它在应用程序结束时被销毁，相关代码如下。

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

3.5.2 线程管理开发步骤

线程管理开发步骤如下。

1. syncDispatch

同步派发任务：派发任务并在当前线程等待任务执行完成。在返回前，当前线程会被阻塞。使用 GlobalTaskDispatcher 派发同步任务相关代码如下。

```
TaskDispatcher globalTaskDispatcher = getGlobalTaskDispatcher(TaskPriority.DEFAULT);
globalTaskDispatcher.syncDispatch(new Runnable() {
    @Override
    public void run() {
        HiLog.info(LABEL_LOG, "sync task1 run");
    }
});
HiLog.info(LABEL_LOG, "after sync task1");
globalTaskDispatcher.syncDispatch(new Runnable() {
    @Override
    public void run() {
        HiLog.info(LABEL_LOG, "sync task2 run");
    }
});
HiLog.info(LABEL_LOG, "after sync task2");
globalTaskDispatcher.syncDispatch(new Runnable() {
    @Override
    public void run() {
        HiLog.info(LABEL_LOG, "sync task3 run");
    }
});
HiLog.info(LABEL_LOG, "after sync task3");
```

执行结果如下。

```
sync task1 run
after sync task1
sync task2 run
after sync task2
sync task3 run
after sync task3
```

如果对 syncDispatch 使用不当，将会导致死锁，可能导致死锁发生的情形如下。

专有线程上，利用该分发器进行 syncDispatch。在被某个串行任务分发器(dispatcher_a)派发的任务中，再次利用同一个串行任务分发器(dispatcher_a)对象派发任务。在被某个串行任务分发器(dispatcher_a)派发的任务中，经过数次派发任务，最终又利用(dispatcher_a)串行任务分发器派发任务。例如，dispatcher_a 派发的任务使用 dispatcher_b 进行任务的派发，在 dispatcher_b 派发的任务中又利用 dispatcher_a 进行派发任务。串行任务分发器(dispatcher_a)派发的任务中利用串行任务分发器(dispatcher_b)进行同步派发任务，同时 dispatcher_b 派发的任务中利用串行任务分发器(dispatcher_a)进行同步派发任务。在特定的线程执行顺序下将导致死锁。

2. asyncDispatch

异步派发任务：派发任务并立即返回，返回值是一个可用于取消任务的接口。GlobalTaskDispatcher 派发异步任务相关代码如下。

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

可能的执行结果如下。

```
after async task1
async task1 run
```

3. delayDispatch

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

```
final long callTime = System.currentTimeMillis();
final long delayTime = 50L;
TaskDispatcher globalTaskDispatcher = getGlobalTaskDispatcher(TaskPriority.DEFAULT);
Revocable revocable = globalTaskDispatcher.delayDispatch(new Runnable() {
```

```
    @Override
    public void run() {
        HiLog.info(LABEL_LOG, "delayDispatch task1 run");
        final long actualDelay = System.currentTimeMillis() - callTime;
        HiLog.info(LABEL_LOG, "actualDelayTime >= delayTime: %{public}b", (actualDelay >=
delayTime));
    }
}, delayTime);
HiLog.info(LABEL_LOG, "after delayDispatch task1");
```

可能的执行结果如下。

```
after delayDispatch task1
delayDispatch task1 run
actualDelayTime >= delayTime : true
```

4. Group

任务组：表示一组任务，且该组任务之间有一定的联系，由 TaskDispatcher 执行 createDispatchGroup 创建并返回。将任务加入任务组，返回一个用于取消任务的接口。将一系列相关联的下载任务放入一个任务组，执行完下载任务后关闭应用，任务组的使用方式代码如下。

```
String dispatcherName = "parallelTaskDispatcher";
TaskDispatcher dispatcher = createParallelTaskDispatcher(dispatcherName, TaskPriority.
DEFAULT);
//创建任务组
Group group = dispatcher.createDispatchGroup();
//将任务 1 加入任务组,返回一个用于取消任务的接口
dispatcher.asyncGroupDispatch(group, new Runnable() {
    @Override
    public void run() {
        HiLog.info(LABEL_LOG, "download task1 is running");
    }
});
//将与任务 1 相关联的任务 2 加入任务组
dispatcher.asyncGroupDispatch(group, new Runnable() {
    @Override
    public void run() {
        HiLog.info(LABEL_LOG, "download task2 is running");
    }
});
//在任务组中的所有任务执行完成后执行指定任务
dispatcher.groupDispatchNotify(group, new Runnable() {
    @Override
    public void run() {
        HiLog.info(LABEL_LOG, "the close task is running after all tasks in the group are
completed");
```

```
    }
});
```

一种可能的执行结果如下。

```
download task1 is running
download task2 is running
the close task is running after all tasks in the group are completed
```

另一种可能的执行结果如下。

```
download task2 is running
download task1 is running
the close task is running after all tasks in the group are completed
```

5. Revocable

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

```
TaskDispatcher dispatcher = getUITaskDispatcher();
Revocable revocable = dispatcher.delayDispatch(new Runnable() {
    @Override
    public void run() {
        HiLog.info(LABEL_LOG, "delay dispatch");
    }
}, 10);
boolean revoked = revocable.revoke();
HiLog.info(LABEL_LOG, "%{public}b", revoked);
```

6. syncDispatchBarrier

同步设置屏障任务：在任务组上设立任务执行屏障，同步等待任务组中的所有任务执行完成，再执行指定任务。在全局并发任务分发器（GlobalTaskDispatcher）上同步设置任务屏障，将不会起到屏障作用，同步设置屏障代码如下。

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

```
dispatcher.asyncGroupDispatch(group, new Runnable() {
    @Override
    public void run() {
        HiLog.info(LABEL_LOG, "task2 is running");  //2
    }
});
dispatcher.syncDispatchBarrier(new Runnable() {
    @Override
    public void run() {
        HiLog.info(LABEL_LOG, "barrier");  //3
    }
});
HiLog.info(LABEL_LOG, "after syncDispatchBarrier");  //4
```

1 和 2 的执行顺序不定；3 和 4 总是在 1 和 2 之后按顺序执行，一种可能的执行结果如下。

```
task1 is running
task2 is running
barrier
after syncDispatchBarrier
```

另外一种执行结果如下。

```
task2 is running
task1 is running
barrier
after syncDispatchBarrier
```

7. asyncDispatchBarrier

异步设置屏障任务：在任务组上设立任务执行屏障后直接返回，指定任务将在任务组中的所有任务执行完成后再执行。在全局并发任务分发器(GlobalTaskDispatcher)上异步设置任务屏障，将不会起到屏障作用。可以使用并发任务分发器(ParallelTaskDispatcher)分离不同的任务组，实现微观并行、宏观串行，异步设置屏障代码如下。

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

```
    @Override
    public void run() {
        HiLog.info(LABEL_LOG, "task2 is running");  //2
    }
});
  dispatcher.asyncDispatchBarrier(new Runnable() {
    @Override
    public void run() {
        HiLog.info(LABEL_LOG, "barrier");  //3
    }
});
HiLog.info(LABEL_LOG, "after asyncDispatchBarrier");  //4
```

1 和 2 的执行顺序不定，但总在 3 之前执行；4 不需要等待 1、2、3 执行完成，可能的执行结果如下。

```
task1 is running
task2 is running
after asyncDispatchBarrier
barrier
```

8. applyDispatch

对指定任务执行多次的相关代码如下。

```
final int total = 10;
final CountDownLatch latch = new CountDownLatch(total);
final List<Long> indexList = new ArrayList<>(total);
TaskDispatcher dispatcher = getGlobalTaskDispatcher(TaskPriority.DEFAULT);
//执行任务 total 次
dispatcher.applyDispatch((index) -> {
    indexList.add(index);
    latch.countDown();
}, total);
//设置任务超时
try {
    latch.await();
} catch (InterruptedException exception) {
   HiLog.error(LABEL_LOG, "latch exception");
}
HiLog.info(LABEL_LOG, "list size matches, %{public}b", (total == indexList.size()));
```

执行结果如下。

```
list size matches, true
```

针对线程管理，演示如何使用 TaskDispatcher 分发不同任务，例如同步派发、异步派发、异步延迟派发等。示例工程参考地址为：https://gitee.com/openharmony/app_

samples/tree/master/thread/TaskDispatcher。

3.6 线程间通信

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

3.6.1 概述

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

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

EventHandler 有两个主要作用：在不同线程间分发和处理 InnerEvent 事件或 Runnable 任务；延迟处理 InnerEvent 事件或 Runnable 任务。

EventHandler 的运作机制如图 3-14 所示。使用 EventHandler 实现线程间通信的主要流程如下。

(1) EventHandler 投递具体的 InnerEvent 事件或者 Runnable 任务到 EventRunner 所创建线程的事件队列。

(2) EventRunner 循环从事件队列中获取 InnerEvent 事件或者 Runnable 任务。

(3) 处理事件或任务。

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

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

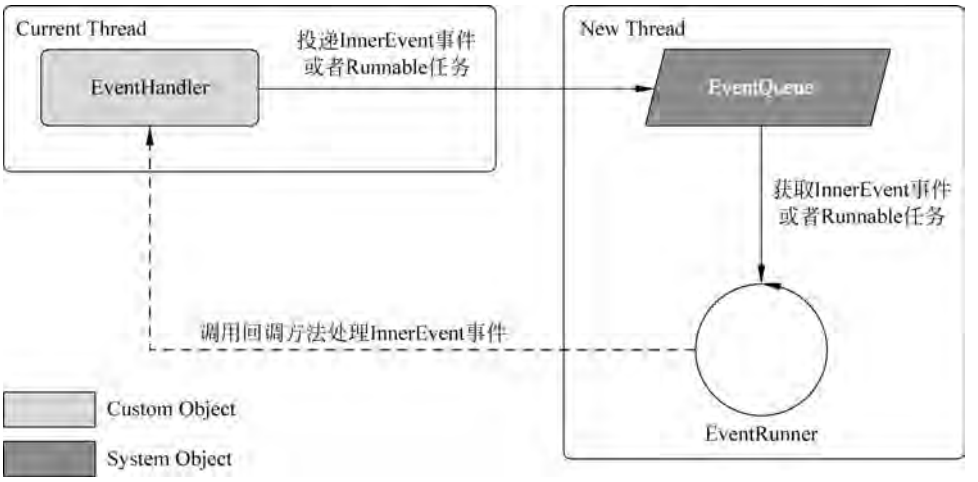

图 3-14 EventHandler 的运作机制

3.6.2 线程间接口关系

本节介绍 EventHandler、EventRunner 和 InnerEvent 3 种应用开发接口。

1. EventHandler

对于 EventHandler 开发场景，主要功能是将 InnerEvent 事件或者 Runnable 任务投递到其他的线程进行处理，使用场景如下。

InnerEvent 事件：需要将 InnerEvent 事件投递到新的线程，按照优先级和延时进行处理。投递时，EventHandler 的优先级可在 IMMEDIATE、HIGH、LOW 和 IDLE 中选择，并设置合适的 delayTime。Runnable 任务：需要将它投递到新的线程，并按照优先级和延时进行处理。投递时，EventHandler 的优先级可在 IMMEDIATE、HIGH、LOW 和 IDLE 中选择，并设置合适的 delayTime。还需要在新创建的线程中投递事件到原线程进行处理。

EventRunner 将根据优先级的高低从事件队列中获取事件或者 Runnable 任务进行处理。EventHandler 的属性 Priority(优先级)如表 3-20 所示。

表 3-20 EventHandler 的属性及功能描述

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

EventHandler 的主要接口及功能描述如表 3-21 所示。

表 3-21 EventHandler 的主要接口及功能描述

主要接口	功能描述
EventHandler(EventRunner runner)	利用已有的 EventRunner 创建 EventHandler
current()	在 processEvent 回调中，获取当前的 EventHandler
processEvent(InnerEvent event)	回调处理事件，由开发者实现
sendEvent(InnerEvent event)	发送一个事件到事件队列，延时为 0ms，优先级为 LOW
sendEvent(InnerEvent event,long delayTime)	发送一个延时事件到事件队列，优先级为 LOW
sendEvent(InnerEvent event, long delayTime, EventHandler.Priority priority)	发送一个指定优先级的延时事件到事件队列
sendEvent(InnerEvent event,EventHandler.Priority priority)	发送一个指定优先级的事件到事件队列，延时为 0ms
sendSyncEvent(InnerEvent event)	发送一个同步事件到事件队列，延时为 0ms，优先级为 LOW
sendSyncEvent(InnerEvent event,EventHandler.Priority priority)	发送一个指定优先级的同步事件到事件队列，延时为 0ms，优先级不能是 IDLE
postSyncTask(Runnable task)	发送一个 Runnable 同步任务到事件队列，延时为 0ms，优先级为 LOW
postSyncTask(Runnable task, EventHandler.Priority priority)	发送一个指定优先级的 Runnable 同步任务到事件队列，延时为 0ms
postTask(Runnable task)	发送一个 Runnable 任务到事件队列，延时为 0ms，优先级为 LOW
postTask(Runnable task,long delayTime)	发送一个 Runnable 延时任务到事件队列，优先级为 LOW
postTask(Runnable task, long delayTime, EventHandler.Priority priority)	发送一个指定优先级的 Runnable 延时任务到事件队列
postTask(Runnable task,EventHandler.Priority priority)	发送一个指定优先级的 Runnable 任务到事件队列，延时为 0ms
sendTimingEvent(InnerEvent event,long taskTime)	发送一个定时事件到队列，在 taskTime 时间执行，如果 taskTime 小于当前时间，立即执行，优先级为 LOW
sendTimingEvent(InnerEvent event,long taskTime, EventHandler.Priority priority)	发送一个带优先级的事件到队列，在 taskTime 时间执行，如果 taskTime 小于当前时间，立即执行
postTimingTask(Runnable task,long taskTime)	发送一个 Runnable 任务到队列，在 taskTime 时间执行，如果 taskTime 小于当前时间，立即执行，优先级为 LOW
postTimingTask(Runnable task,long taskTime, EventHandler.Priority priority)	发送一个带优先级的 Runnable 任务到队列，在 taskTime 时间执行，如果 taskTime 小于当前时间，立即执行
removeEvent(int eventId)	删除指定 ID 的事件
removeEvent(int eventId,long param)	删除指定 ID 和 param 的事件

续表

主要接口	功能描述
removeEvent(int eventId, long param, Object object)	删除指定 ID、param 和 object 的事件
removeAllEvent()	删除 EventHandler 的所有事件
getEventName(InnerEvent event)	获取事件的名称
getEventRunner()	获取 EventHandler 绑定的 EventRunner
isIdle()	判断队列是否为空
hasInnerEvent(Runnable runnable)	根据指定的 runnable 参数,检查是否有还未被处理的任务。可以根据不同的入参进行检查,详见 EventHandler

2. EventRunner

EventRunner 工作模式可以分为托管模式和手动模式。两种模式是在调用 EventRunner 的 create()方法时,通过选择不同的参数实现。默认为托管模式。托管模式:不需要调用 run()和 stop()方法启动、停止 EventRunner。当 EventRunner 实例化时,系统调用 run()启动 EventRunner;当 EventRunner 不被引用时,系统调用 stop()停止 EventRunner。手动模式:需要自行调用 EventRunner 的 run()方法和 stop()方法确保线程的启动、停止。EventRunner 的主要接口及功能描述如表 3-22 所示。

表 3-22 EventRunner 的主要接口及功能描述

主要接口	功能描述
create()	创建一个拥有新线程的 EventRunner
create (boolean inNewThread)	创建一个拥有新线程的 EventRunner,inNewThread 为 true 时,EventRunner 为托管模式,系统将自动管理 EventRunner;inNewThread 为 false 时,EventRunner 为手动模式
create (String newThreadName)	创建一个拥有新线程的 EventRunner,新线程的名称是 newThreadName
current()	获取当前线程的 EventRunner
run()	EventRunner 为手动模式时,调用该方法启动新线程
stop()	EventRunner 为手动模式时,调用该方法停止新线程

3. InnerEvent

InnerEvent 的属性及功能描述如表 3-23 所示。

表 3-23 InnerEvent 的属性及功能描述

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

InnerEvent 的主要接口及功能描述如表 3-24 所示。

表 3-24 InnerEvent 的主要接口及功能描述

主 要 接 口	功 能 描 述
drop()	释放一个事件实例
get()	获得一个事件实例
get(int eventId)	获得一个指定的 eventId 的事件实例
get(int eventId,long param)	获得一个指定的 eventId 和 param 的事件实例
get(int eventId,long param,Object object)	获得一个指定的 eventId、param 和 object 的事件实例
get(int eventId,Object object)	获得一个指定的 eventId 和 object 的事件实例
PacMap getPacMap()	获取 PacMap,如果没有,则新建一个
Runnable getTask()	获取 Runnable 任务
PacMap peekPacMap()	获取 PacMap
void setPacMap(PacMap pacMap)	设置 PacMap

3.6.3 线程间通信开发步骤

本节介绍 EventHandler 投递 InnerEvent 事件、EventHandler 投递 Runnable 任务和在新创建线程中投递事件到原线程。

1. EventHandler 投递 InnerEvent 事件

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

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

```
private static final int EVENT_MESSAGE_NORMAL = 1;
private static final int EVENT_MESSAGE_DELAY = 2;
private class MyEventHandler extends EventHandler {
    private MyEventHandler(EventRunner runner) {
        super(runner);
    }
    //重写实现 processEvent()方法
    @Override
    public void processEvent(InnerEvent event) {
        super.processEvent(event);
        if (event == null) {
            return;
        }
        int eventId = event.eventId;
        switch (eventId) {
            case EVENT_MESSAGE_NORMAL:
                //待执行的操作,由开发者定义
                break;
            case EVENT_MESSAGE_DELAY:
                //待执行的操作,由开发者定义
```

```
                break;
            default:
                break;
        }
    }
}
```

(2) 以手动模式为例,创建 EventRunner。

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

(3) 创建 EventHandler 子类的实例。

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

(4) 获取 InnerEvent 事件。

```
//获取事件实例,其属性 eventId、param 和 object 由开发者确定,代码中只是示例
long param = 0L;
Object object = null;
InnerEvent normalInnerEvent = InnerEvent.get(EVENT_MESSAGE_NORMAL, param, object);
InnerEvent delayInnerEvent = InnerEvent.get(EVENT_MESSAGE_DELAY, param, object);
```

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

```
//优先级 IMMEDIATE,投递之后立即处理,延时为 0ms,该语句等价于同步投递 sendSyncEvent(event1,
EventHandler.Priority.IMMEDIATE);
myHandler.sendEvent(normalInnerEvent, 0, EventHandler.Priority.IMMEDIATE);
myHandler.sendEvent(delayInnerEvent, 2, EventHandler.Priority.IMMEDIATE);
//延时 2ms 后立即处理
```

(6) 启动和停止 EventRunner,如果为托管模式,则不需要此步骤。

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

2. EventHandler 投递 Runnable 任务

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

(1) 创建 EventHandler 的子类,先创建 EventRunner,然后创建 EventHandler 子类的实例,步骤与 EventHandler 投递 InnerEvent 事件的步骤(1)~(3)相同。

(2) 创建 Runnable 任务。

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

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

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

```
//优先级为 immediate,延时 0ms,该语句等价于同步投递 myHandler.postSyncTask(task1,
EventHandler.Priority.IMMEDIATE);
myHandler.postTask(normalTask, 0, EventHandler.Priority.IMMEDIATE);
myHandler.postTask(delayTask, 2, EventHandler.Priority.IMMEDIATE);   //延时 2ms 后立即执行
```

(4) 启动和停止 EventRunner,如果是托管模式,则不需要此步骤。

```
runner.run();
//待执行操作
runner.stop();                          //停止 EventRunner
```

3. 在新创建线程中投递事件到原线程

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

(1) 创建 EventHandler 的子类,在子类中重写实现方法 processEvent()处理事件,相关代码如下。

```
private static final int EVENT_MESSAGE_CROSS_THREAD = 1;
private class MyEventHandler extends EventHandler {
    private MyEventHandler(EventRunner runner) {
        super(runner);
    }
    //重写实现 processEvent()方法
    @Override
    public void processEvent(InnerEvent event) {
        super.processEvent(event);
        if (event == null) {
            return;
        }
        int eventId = event.eventId;
        switch (eventId) {
            case EVENT_MESSAGE_CROSS_THREAD:
                Object object = event.object;
                if (object instanceof EventRunner) {
                    //将原先线程的 EventRunner 实例投递给新创建的线程
                    EventRunner runner2 = (EventRunner) object;
                    //将原先线程的 EventRunner 实例与新创建线程的 EventHandler 绑定
                    EventHandler myHandler2 = new EventHandler(runner2) {
```

```
                    @Override
                    public void processEvent(InnerEvent event) {
                        //需要在原先线程执行的操作
                    }
                };
                int eventId2 = 1;
                long param2 = 0L;
                Object object2 = null;
                InnerEvent event2 = InnerEvent.get(eventId2, param2, object2);
                myHandler2.sendEvent(event2); //投递事件到原先的线程
            }
            break;
        default:
            break;
    }
  }
}
```

(2) 以手动模式为例,创建 EventRunner。

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

(3) 创建 EventHandler 子类的实例。

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

(4) 获取 InnerEvent 事件。

```
//获取事件实例,其属性 eventId、param 和 object 由开发者确定,代码中只是示例
long param = 0L;
InnerEvent event = InnerEvent.get(EVENT_MESSAGE_CROSS_THREAD, param, EventRunner.current());
```

(5) 投递事件,在新线程上直接处理。

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

(6) 启动和停止 EventRunner,如果是托管模式,则不需要此步骤。

```
runner.run();                    //待执行操作
runner.stop();                   //停止 EventRunner
```

针对线程间通信,演示将 InnerEvent 事件投递到新的线程、将 Runnable 任务投递到新的线程、在新创建的线程投递事件到原线程进行处理。示例工程参考地址如下: https://gitee.com/openharmony/app_samples/tree/master/thread/EventHandler。

3.7 剪贴板开发

用户通过系统剪贴板服务,可实现应用之间的简单数据传递。例如,在应用 A 中复制

的数据，可以在应用 B 中粘贴，反之亦可。

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

3.7.1 剪贴板开发接口关系

同一设备的应用程序 A、B 之间可以借助系统剪贴板服务完成简单数据的传递，即应用程序 A 向剪贴板服务写入数据后，应用程序 B 可以从中读取数据，如图 3-15 所示。

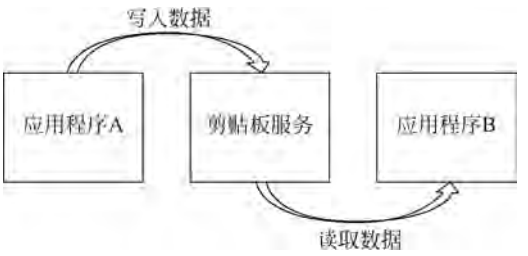

图 3-15 剪贴板服务示意图

在使用剪贴板服务时，需要注意以下几点：只有在前台获取到焦点的应用才有读取系统剪贴板的权限（系统默认输入法应用除外）。写入剪贴板服务中的数据不会随应用程序结束而销毁。对同一用户而言，写入剪贴板服务的数据会被下次写入的剪贴板数据覆盖。在同一设备内，剪贴板单次传递内容不应超过 500KB。

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

1. SystemPasteboard

SystemPasteboard 的主要接口及功能描述如表 3-25 所示。

表 3-25 SystemPasteboard 的主要接口及功能描述

主 要 接 口	功 能 描 述
getSystemPasteboard(Context context)	获取系统剪贴板服务的对象实例
getPasteData()	读取当前系统剪贴板中的数据
hasPasteData()	判断当前系统剪贴板中是否有内容
setPasteData(PasteData data)	将剪贴板数据写入系统剪贴板
clear()	清空系统剪贴板数据

续表

主 要 接 口	功 能 描 述
addPasteDataChangedListener (IPasteDataChangedListener listener)	用户程序添加系统剪贴板数据变化的回调，当系统剪贴板数据发生变化时，会触发用户程序的回调实现
removePasteDataChangedListener (IPasteDataChangedListener listener)	用户程序移除系统剪贴板数据变化的回调

2. PasteData

PasteData 是剪贴板服务操作的数据对象，其中内容节点定义为 PasteData. Record，属性集合定义为 PasteData. DataProperty，如表 3-26 所示。

表 3-26 PasteData 的主要接口及功能描述

主 要 接 口	功 能 描 述
PasteData()	构造器，创建一个空内容数据对象
creatPlainTextData(CharSequence text)	构建一个包含纯文本内容节点的数据对象
creatHtmlData(String htmlText)	构建一个包含 HTML 内容节点的数据对象
creatUriData(Uri uri)	构建一个包含 URI 内容节点的数据对象
creatIntentData(Intent intent)	构建一个包含 Intent 内容节点的数据对象
getPrimaryMimeType()	获取数据对象中首个内容节点的 MIME 类型，如果没有查询到内容，将返回一个空字符串
getPrimaryText()	获取数据对象中首个内容节点的纯文本内容，如果没有查询到内容，将返回一个空对象
addTextRecord(CharSequence text)	向数据对象中添加一个纯文本内容节点，该方法会自动更新数据属性中的 MIME 类型集合，最多只能添加 128 个内容节点
addRecord(Record record)	向数据对象中添加一个内容节点，该方法会自动更新数据属性中的 MIME 类型集合，最多只能添加 128 个内容节点
getRecordCount()	获取数据对象中内容节点的数量
getRecordAt(int index)	获取数据对象在指定下标处的内容节点，如果操作失败会返回空对象
removeRecordAt(int index)	移除数据对象在指定下标处的内容节点，如果操作成功会返回 true，操作失败会返回 false
getMimeTypes()	获取数据对象中所有内容节点的 MIME 类型列表，当内容节点为空时，返回列表为空对象
getProperty()	获取该数据对象的属性集合成员

PasteData 中定义的常量名及功能描述如表 3-27 所示。

表 3-27 PasteData 中定义的常量名及功能描述

常 量 名	功 能 描 述
MIMETYPE_TEXT_PLAIN= "text/plain"	纯文本的 MIME 类型定义
MIMETYPE_TEXT_HTML= "text/html"	HTML 的 MIME 类型定义
MIMETYPE_TEXT_URI= "text/uri"	URI 的 MIME 类型定义
MIMETYPE_TEXT_INTENT= "text/ohos.intent"	Intent 的 MIME 类型定义
MAX_RECORD_NUM=128	单个 PasteData 中所能包含的 Record 的数量上限

3. PasteData.Record

一个 PasteData 中包含若干个特定 MIME 类型的 PasteData.Record，每个 Record 是存放剪贴板数据内容信息的最小单位，如表 3-28 所示。

表 3-28 PasteData.Record 的主要接口及功能描述

主 要 接 口	功 能 描 述
createPlainTextRecord(CharSequence text)	构造一个 MIME 类型为纯文本的内容节点
createHtmlTextRecord(String htmlText)	构造一个 MIME 类型为 HTML 的内容节点
createUriRecord(Uri uri)	构造一个 MIME 类型为 URI 的内容节点
createIntentRecord(Intent intent)	构造一个 MIME 类型为 Intent 的内容节点
getPlainText()	获取该内容节点中的文本内容，如果没有内容返回空对象
getHtmlText()	获取该内容节点中的 HTML 内容，没有内容将返回空对象
getUri()	获取该内容节点中的 URI 内容，如果没有内容返回空对象
getIntent()	获取该内容节点中的 Intent 内容，如果没有内容返回空对象
getMimeType()	获取该内容节点的 MIME 类型
convertToText(Context context)	将该内容节点的内容转为文本形式

4. PasteData.DataProperty

每个 PasteData 中都有一个 PasteData.DataProperty 成员，存放该数据对象的属性集合，例如自定义标签、MIME 类型集合列表等，主要接口及功能描述如表 3-29 所示。

表 3-29 PasteData.DataProperty 的主要接口及功能描述

主 要 接 口	功 能 描 述
getMimeTypes()	获取所属数据对象的 MIME 类型集合列表，当内容节点为空时，返回列表为空对象
hasMimeType(String mimeType)	判断所属数据对象中是否包含特定 MIME 类型的内容
getTimestamp()	获取所属数据对象被写入系统剪贴板时的时间戳，如果该数据对象尚未被写入，则返回 0
setTag(CharSequence tag)	设置自定义标签
getTag()	获取自定义标签

续表

主 要 接 口	功 能 描 述
setAdditions(PacMap extraProps)	设置一些附加键值对信息
getAdditions()	获取附加键值对信息

5. IPasteDataChangedListener

IPasteDataChangedListener 是定义剪贴板数据变化回调的接口类，开发者需要实现此接口编码触发回调时的处理逻辑，主要接口及功能描述如表 3-30 所示。

表 3-30 IPasteDataChangedListener 的主要接口及功能描述

主 要 接 口	功 能 描 述
onChanged()	系统剪贴板数据发生变化时的回调接口

3.7.2 剪贴板开发步骤

剪贴板的主要开发步骤如下。

(1) 应用 A 获取系统剪贴板服务，相关代码如下。

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

(2) 应用 A 向系统剪贴板中写入一条纯文本数据，相关代码如下。

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

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

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

```
            } else if (mimeType.equals(PasteData.MIMETYPE_TEXT_PLAIN)) {
                text.setText(record.getPlainText().toString());
                break;
            } else {
                //跳过其他 Mime 类型的记录
            }
        }
    }
```

(4) 应用 C 注册添加系统剪贴板数据变化回调,当系统剪贴板数据发生变化时触发处理逻辑,相关代码如下。

```
IPasteDataChangedListener listener = new IPasteDataChangedListener() {
    @Override
    public void onChanged() {
        PasteData pasteData = pasteboard.getPasteData();
        if (pasteData == null) {
            return;
        }
        //处理系统剪贴板上数据更改的操作
    }
};
pasteboard.addPasteDataChangedListener(listener);
```

针对剪贴板开发指导,演示应用之间的数据剪贴。示例工程参考地址为:https://gitee.com/openharmony/app_samples/tree/master/ability/Pasteboard。