# HarmonyOS 开发文档（二）

**V1.0**

鸿蒙学堂 hmxt.org 整理

2020 年 9 月 10 日

# 目 录

1 媒体 ........ 1
1.1 视频 ........ 1
1.1.1 概述 ........ 1
1.1.2 媒体编解码能力查询开发指导 ........ 2
1.1.3 视频编解码开发指导 ........ 3
1.1.4 视频播放开发指导 ........ 8
1.1.5 视频录制开发指导 ........ 10
1.1.6 视频提取开发指导 ........ 13
1.1.7 媒体描述信息开发指导 ........ 15
1.1.8 媒体元数据开发指导 ........ 17
1.2 图像 ........ 20
1.2.1 概述 ........ 20
1.2.2 图像解码开发指导 ........ 21
1.2.3 图像编码开发指导 ........ 24
1.2.4 位图操作开发指导 ........ 26
1.2.5 图像属性解码开发指导 ........ 29
1.3 相机 ........ 30
1.3.1 概述 ........ 30
1.3.2 设备开发指导 ........ 31
1.4 音频 ........ 46
1.4.1 概述 ........ 46
1.4.2 音频播放开发指导 ........ 48
1.4.3 音频采集开发指导 ........ 52
1.4.4 音量管理开发指导 ........ 56
1.4.5 短音播放开发指导 ........ 59
2 系统音的播放 ........ 63
2.1 媒体会话管理 ........ 64
2.1.1 概述 ........ 64
2.1.2 媒体会话开发指导 ........ 64
2.2 媒体数据管理 ........ 78
2.2.1 概述 ........ 78
2.2.2 媒体元数据获取开发指导 ........ 78
2.2.3 媒体存储数据操作开发指导 ........ 81
2.2.4 媒体扫描服务操作开发指导 ........ 84
2.2.5 视频与图像缩略图获取开发指导 ........ 86
3 安全 ........ 86

3.1 权限 ...... 87
3.1.1 概述 ...... 87
3.1.2 开发指导 ...... 89
3.1.3 应用权限列表 ...... 98
3.2 生物特征识别 ...... 103
3.2.1 概述 ...... 103
4 运作机制 ...... 103
4.1.2 开发指导 ...... 104
5 AI ...... 107
5.1 码生成 ...... 108
5.1.1 概述 ...... 108
5.1.2 开发指导 ...... 108
6 网络与连接 ...... 110
6.1 NFC ...... 110
6.1.1 概述 ...... 110
6.1.2 NFC 基础控制 ...... 110
6.1.3 Tag 读写 ...... 112
6.1.4 访问 SE 安全单元 ...... 119
6.1.5 卡模拟功能 ...... 122
6.1.6 NFC 消息通知 ...... 128
6.2 蓝牙 ...... 131
6.2.1 概述 ...... 131
6.2.2 传统蓝牙本机管理 ...... 132
6.2.3 传统蓝牙远端设备操作 ...... 134
6.2.4 BLE 扫描和广播 ...... 135
6.3 WLAN ...... 138
6.3.1 概述 ...... 138
6.3.2 WLAN 基础功能 ...... 138
6.3.3 P2P 功能 ...... 142
6.3.4 WLAN 消息通知 ...... 147
6.4 网络管理 ...... 150
6.4.1 概述 ...... 150
6.4.2 使用当前网络打开一个 URL 链接 ...... 150
6.4.3 使用当前网络进行 Socket 数据传输 ...... 152
6.4.4 使用指定网络进行数据访问 ...... 154
6.4.5 流量统计 ...... 156
6.4.6 管理 HTTP 缓存 ...... 157

6.5 电话服务 ......158
6.5.1 概述 ......158
6.5.2 发起一路呼叫......159
6.5.3 发送一条文本信息 ......161
6.5.4 获取当前蜂窝网络信号信息 ......163
6.5.5 开发步骤......163
6.5.6 观察蜂窝网络状态变化 ......164
7 设备管理 ......167
7.1 传感器......167
7.1.1 概述 ......167
7.1.2 开发指导......177
7.2 控制类小器件......182
7.2.1 概述 ......182
7.2.2 Light 开发指导......184
7.2.3 Vibrator 开发指导......186
7.3 位置 ......189
7.3.1 概述 ......189
7.3.2 获取设备的位置信息......191
7.3.3 （逆）地理编码转化......196
7.4 设置 ......197
7.4.1 概述 ......197
7.4.2 开发指导......198
8 数据管理 ......203
8.1 关系型数据库......203
8.1.1 概述 ......203
8.1.2 开发指导......205
8.2 对象映射关系型数据库 ......213
8.2.1 概述 ......213
8.2.2 开发指导......217
8.3 轻量级偏好数据库......225
8.3.1 概述 ......225
8.3.2 开发指导......226
8.4 分布式数据服务 ......231
8.4.1 概述 ......231
8.4.2 开发指导......236

8.5 分布式文件服务 ......241
8.5.1 概述 ......241
8.5.2 开发指导 ......243
8.6 融合搜索 ......244
8.6.1 概述 ......244
8.6.2 开发指导 ......246
8.7 数据存储管理 ......252
8.7.1 概述 ......252
8.7.2 开发指导 ......254

# 1 媒体

## 1.1 视频

### 1.1.1 概述

HarmonyOS 视频模块支持视频业务的开发和生态开放，开发者可以通过已开放的接口很容易地实现视频媒体的播放、操作和新功能开发。视频媒体的常见操作有视频编解码、视频合成、视频提取、视频播放以及视频录制等。

#### 基本概念

- **编码**

  编码是信息从一种形式或格式转换为另一种形式的过程。用预先规定的方法将文字、数字或其他对象编成数码，或将信息、数据转换成规定的电脉冲信号。在本模块中，编码是指编码器将原始的视频信息压缩为另一种格式的过程。

- **解码**

  解码是一种用特定方法，把数码还原成它所代表的内容或将电脉冲信号、光信号、无线电波等转换成它所代表的信息、数据等的过程。在本模块中，解码是指解码器将接收到的数据还原为视频信息的过程，与编码过程相对应。

- **帧率**

  帧率是以帧称为单位的位图图像连续出现在显示器上的频率（速率），以赫兹（Hz）为单位。该术语同样适用于胶片、摄像机、计算机图形和动作捕捉系统。

## 1.1.2 媒体编解码能力查询开发指导

### 1.1.2.1 场景介绍

媒体编解码能力查询主要指查询设备所支持的编解码器的 MIME（Multipurpose Internet Mail Extensions，媒体类型）列表，并判断设备是否支持指定 MIME 对应的编码器/解码器。

### 1.1.2.2 接口说明

| 接口名 | 功能描述 |
| --- | --- |
| getSupportedMimes() | 获取某设备所支持的编解码器的 MIME 列表。 |
| isDecodeSupportedByMime(String mime) | 判断某设备是否支持指定 MIME 对应的解码器。 |
| isEncodeSupportedByMime(String mime) | 判断某设备是否支持指定 MIME 对应的编码器。 |
| isDecoderSupportedByFormat(Format format) | 判断某设备是否支持指定媒体格式对应的解码器。 |
| isEncoderSupportedByFormat(Format format) | 判断某设备是否支持指定媒体格式对应的编码器。 |

**表 1** 媒体编解码能力查询类 CodecDescriptionList 的主要接口

### 1.1.2.3 开发步骤

1. 调用 CodecDescriptionList 类的静态 getSupportedMimes()方法，获取某设备所支持的编解码器的 MIME 列表。代码示例如下：

```
List<String> mimes = CodecDescriptionList.getSupportedMimes();
```

2. 调用 CodecDescriptionList 类的静态 isDecodeSupportedByMime 方法，判断某设备是否支持指定 MIME 对应的解码器，支持返回 true，否则返回 false。代码示例如下：

```
boolean result = CodecDescriptionList.isDecodeSupportedByMime(Format.VIDEO_VP9);
```

3. 调用 CodecDescriptionList 类的静态 isEncodeSupportedByMime 方法，判断某设备是否支持指定 MIME 对应的编码器，支持返回 true，否则返回 false。代码示例如下：

```
1. boolean result = CodecDescriptionList.isEncodeSupportedByMime(Format.AUDIO_FLAC);
```

4. 调用 CodecDescriptionList 类的静态 isDecoderSupportedByFormat/isEncoderSupportedByFormat 方法，判断某设备是否支持指定 Format 的编解码器，支持返回 true，否则返回 false。代码示例如下：

```
1. Format format = new Format();
2. format.putStringValue(Format.MIME, Format.VIDEO_AVC);
3. format.putIntValue(Format.WIDTH, 2560);
4. format.putIntValue(Format.HEIGHT, 1440);
5. format.putIntValue(Format.FRAME_RATE, 30);
6. format.putIntValue(Format.FRAME_INTERVAL, 1);
7. boolean result = CodecDescriptionList.isDecoderSupportedByFormat(format);
8. result = CodecDescriptionList.isEncoderSupportedByFormat(format);
```

### 1.1.3 视频编解码开发指导

#### 1.1.3.1 场景介绍

视频编解码的主要工作是将视频进行编码和解码。

#### 1.1.3.2 接口说明

| 接口名 | 功能描述 |
| --- | --- |
| createDecoder() | 创建解码器 Codec 实例。 |
| createEncoder() | 创建编码器 Codec 实例。 |
| registerCodecListener(ICodecListener listener) | 注册侦听器用来异步接收编码或解码后的数据。 |
| setSource(Source source, TrackInfo trackInfo) | 根据解码器的源轨道信息设置数据源，对于编码器 trackInfo 无效。 |
| setSourceFormat(Format format) | 编码器的管道模式下，设置编码器编码格式。 |

| 接口名 | 功能描述 |
| --- | --- |
| setCodecFormat(Format format) | 普通模式设置编/解码器参数。 |
| setVideoSurface(Surface surface) | 设置解码器的 Surface。 |
| getAvailableBuffer(long timeout) | 普通模式获取可用 ByteBuffer。 |
| writeBuffer(ByteBuffer buffer, BufferInfo info) | 推送源数据给 Codec。 |
| getBufferFormat(ByteBuffer buffer) | 获取输出 Buffer 数据格式。 |
| start() | 启动编/解码。 |
| stop() | 停止编/解码。 |
| release() | 释放所有资源。 |

**表 1** 视频编解码类 Codec 的主要接口

#### 1.1.3.3 普通模式开发步骤

在普通模式下进行编解码，应用必须持续地传输数据到 Codec 实例。

**编码的具体开发步骤如下**：

1. 创建编码 Codec 实例，可调用 createEncoder()创建。

```
1.  final Codec encoder = Codec.createEncoder();
```

2. 构造数据源格式，并设置给 Codec 实例，调用 setCodecFormat()，代码示例如下：

```
1.  Format fmt = new Format();
2.  fmt.putStringValue(Format.MIME, Format.VIDEO_AVC);
3.  fmt.putIntValue(Format.WIDTH, 1920);
4.  fmt.putIntValue(Format.HEIGHT, 1080);
5.  fmt.putIntValue(Format.BIT_RATE, 392000);
6.  fmt.putIntValue(Format.FRAME_RATE, 30);
7.  fmt.putIntValue(Format.FRAME_INTERVAL, -1);
8.  codec.setCodecFormat(fmt);
```

3. 如果需要编码过程中，检测是否读取到 Buffer 数据以及是否发生异常，可以构造 ICodecListener，ICodecListener 需要实现两个方法，实现读到 Buffer 数据时、编码发生异常时做相应的操作。举例中读到 buffer 时，获取 buffer 的 format 格式，异常时抛出运行时异常，代码示例如下：

```
Codec.ICodecListener listener = new Codec.ICodecListener() {
    @Override
    public void onReadBuffer(ByteBuffer byteBuffer, BufferInfo bufferInfo, int trackId) {
        Format fmt = codec.getBufferFormat(byteBuffer);
    }

    @Override
    public void onError(int errorCode, int act, int trackId) {
        throw new RuntimeException();
    }
};
```

4. 调用 start()方法开始编码。
5. 调用 getAvailableBuffer()取到一个可用的 ByteBuffer，把数据填入 ByteBuffer 里，然后再调用 writeBuffer()把 ByteBuffer 写入编码器实例。
6. 调用 stop()方法停止编码。
7. 编码任务结束后，调用 release()释放资源。

**解码的具体开发步骤如下：**

1. 创建解码 Codec 实例，可调用 createDecoder()创建。
2. 构造数据源格式，并设置给 Codec 实例，调用 setCodecFormat()，代码示例如下：

```
Format fmt = new Format();
fmt.putStringValue(Format.MIME, Format.VIDEO_AVC);
fmt.putIntValue(Format.WIDTH, 1920);
fmt.putIntValue(Format.HEIGHT, 1080);
fmt.putIntValue(Format.BIT_RATE, 392000);
fmt.putIntValue(Format.FRAME_RATE, 30);
fmt.putIntValue(Format.FRAME_INTERVAL, -1);
codec.setCodecFormat(fmt);
```

3. （可选）如果需要解码过程中，检测是否读取到 Buffer 数据以及是否发生异常，可以构造 ICodecListener，ICodecListener 需要实现两个方法，实现读到 Buffer 数据时、解码发生异常时做相应的操作。举例中读到 buffer 时，获取 buffer 的 format 格式，异常时抛出运行时异常，代码示例如下：

```
Codec.ICodecListener listener = new Codec.ICodecListener() {
    @Override
```

```
3.        public void onReadBuffer(ByteBuffer byteBuffer, BufferInfo bufferInfo, int trackId) {
4.            Format fmt = codec.getBufferFormat(byteBuffer);
5.        }
6.
7.        @Override
8.        public void onError(int errorCode, int act, int trackId) {
9.            throw new RuntimeException();
10.       }
11.   };
```

4. 调用 start()方法开始解码。
5. 调用 getAvailableBuffer 取到一个可用的 ByteBuffer，把数据填入 ByteBuffer 里，然后再调用 writeBuffer 把 ByteBuffer 写入解码器实例。
6. 调用 stop()方法停止解码。
7. 解码任务结束后，调用 release()释放资源。

#### 1.1.3.4 管道模式开发步骤

管道模式下应用只需要调用 Source 类的 setSource()方法，数据会自动解析并传输给 Codec 实例。管道模式编码支持视频流编码和音频流编码。

**编码的具体开发步骤如下：**

1. 调用 createEncoder()创建编码 Codec 实例。
2. 调用 setSource()设置数据源，支持设定文件路径或者文件 File Descriptor。
3. 构造数据源格式或者从 Extractor 中读取数据源格式，并设置给 Codec 实例，调用 setSourceFormat()，构造数据原格式代码示例如下：

```
1.  Format fmt = new Format();
2.  fmt.putStringValue(Format.MIME, Format.VIDEO_AVC);
3.  fmt.putIntValue(Format.WIDTH, 1920);
4.  fmt.putIntValue(Format.HEIGHT, 1080);
5.  fmt.putIntValue(Format.BIT_RATE, 392000);
6.  fmt.putIntValue(Format.FRAME_RATE, 30);
7.  fmt.putIntValue(Format.FRAME_INTERVAL, -1);
8.  codec.setSourceFormat(fmt);
```

4. （可选）如果需要编码过程中，检测是否读取到 Buffer 数据以及是否发生异常，可以构造 ICodecListener，ICodecListener 需要实现两个方法，实现读到 Buffer 数据时、编码发生异常时做相应的操作。举例中读到 buffer 时，获取 buffer 的 format 格式，异常时抛出运行时异常，代码示例如下：

```
1.  Codec.ICodecListener listener = new Codec.ICodecListener() {
2.      @Override
3.      public void onReadBuffer(ByteBuffer byteBuffer, BufferInfo bufferInfo, int trackId) {
4.          Format fmt = codec.getBufferFormat(byteBuffer);
5.      }
6.
7.      @Override
8.      public void onError(int errorCode, int act, int trackId) {
9.          throw new RuntimeException();
10.     }
11. };
```

5. 调用 start()方法开始编码。
6. 调用 stop()方法停止编码。
7. 编码任务结束后，调用 release()释放资源。

**解码的具体开发步骤如下：**

1. 调用 createDecoder()创建解码 Codec 实例。
2. 调用 setSource()设置数据源，支持设定文件路径或者文件 File Descriptor。
3. （可选）如果需要解码过程中，检测是否读取到 Buffer 数据以及是否发生异常，可以构造 ICodecListener，ICodecListener 需要实现两个方法，实现读到 Buffer 数据时、解码发生异常时做相应的操作。举例中读到 buffer 时，获取 buffer 的 format 格式，异常时抛出运行时异常，代码示例如下：

```
1.  Codec.ICodecListener listener = new Codec.ICodecListener() {
2.      @Override
3.      public void onReadBuffer(ByteBuffer byteBuffer, BufferInfo bufferInfo, int trackId) {
4.          Format fmt = codec.getBufferFormat(byteBuffer);
5.      }
6.
7.      @Override
8.      public void onError(int errorCode, int act, int trackId) {
9.          throw new RuntimeException();
10.     }
11. };
```

4. 调用 start()方法开始解码。

5. 调用 stop()方法停止解码。
6. 解码任务结束后，调用 release()释放资源。

### 1.1.4 视频播放开发指导

#### 1.1.4.1 场景介绍

视频播放包括播放控制、播放设置和播放查询，如播放的开始/停止、播放速度设置和是否循环播放等。

#### 1.1.4.2 接口说明

| 接口名 | 功能描述 |
| --- | --- |
| Player(Context context) | 创建 Player 实例。 |
| setSource(Source source) | 设置媒体源。 |
| prepare() | 准备播放。 |
| play() | 开始播放。 |
| pause() | 暂停播放。 |
| stop() | 停止播放。 |
| rewindTo(long microseconds) | 拖拽播放。 |
| setVolume(float volume) | 调节播放音量。 |
| setVideoSurface(Surface surface) | 设置视频播放的窗口。 |
| enableSingleLooping(boolean looping) | 设置为单曲循环。 |

| 接口名 | 功能描述 |
| --- | --- |
| isSingleLooping() | 检查是否单曲循环播放。 |
| isNowPlaying() | 检查是否播放。 |
| getCurrentTime() | 获取当前播放位置。 |
| getDuration() | 获取媒体文件总时长。 |
| getVideoWidth() | 获取视频宽度。 |
| getVideoHeight() | 获取视频高度。 |
| setPlaybackSpeed(float speed) | 设置播放速度。 |
| getPlaybackSpeed() | 获取播放速度。 |
| setAudioStreamType(int type) | 设置音频类型。 |
| getAudioStreamType() | 获取音频类型。 |
| setNextPlayer(Player next) | 设置当前播放结束后的下一个播放器。 |
| reset() | 重置播放器。 |
| release() | 释放播放资源。 |
| setPlayerCallback(IPlayerCallback callback) | 注册回调，接收播放器的事件通知或异常通知。 |

**表 1** 视频播放类 Player 的主要接口

#### 1.1.4.3 开发步骤

1. 创建 Player 实例，可调用 Player(Context context)，创建本地播放器，用于在本设备播放。
2. 构造数据源对象，并调用 Player 实例的 setSource(Source source)方法，设置媒体源，代码示例如下：

```
1.    Player impl = new Player(context);
```

```
2.  File file = new File("/path/test_audio.aac");
3.  in = new FileInputStream(file);
4.  FileDescriptor fd = in.getFD(); // 从输入流获取 FD 对象
5.  Source source = new Source(fd);
6.  impl.setSource(source);
```

3. 调用 prepare()，准备播放。

4. （可选）构造 IPlayerCallback，IPlayerCallback 需要实现 onPlayBackComplete 和 onError(int errorType, int errorCode)两个方法，实现播放完成和播放异常时做相应的操作。代码示例如下：

```
1.  @Override
2.  public void onPlayBackComplete() {
3.      HiLog.info("[PlayerCallback]", "onPlayBackComplete");
4.  
5.      if (impl != null) {
6.          impl.stop();
7.          impl = null;
8.      }
9.  }
10. 
11. @Override
12. public void onError(int errorType, int errorCode) {
13.     HiLog.error("[PlayerCallback]", "onError");
14. }
```

5. 调用 play()方法，开始播放。
6. （可选）调用 pause()方法和 resume()方法，可以实现暂停和恢复播放。
7. （可选）调用 rewindTo(long microseconds)方法实现播放中的拖拽功能。
8. （可选）调用 getDuration()方法和 getCurrentTime()方法，可以实现获取总播放时长以及当前播放位置功能。
9. 调用 stop()方法停止播放。
10. 播放结束后，调用 release()释放资源。

### 1.1.5 视频录制开发指导

#### 1.1.5.1 场景介绍

视频录制的主要工作是选择视频/音频来源后，录制并生成视频/音频文件。

### 1.1.5.2 接口说明

| 接口名 | 功能描述 |
| --- | --- |
| Recorder() | 创建 Recorder 实例。 |
| setSource(Source source) | 设置音视频源。 |
| setAudioProperty(AudioProperty property) | 设置音频属性。 |
| setVideoProperty(VideoProperty property) | 设置视频属性。 |
| setStorageProperty(StorageProperty property) | 设置音视频存储属性。 |
| prepare() | 准备录制资源。 |
| start() | 开始录制。 |
| stop() | 停止录制。 |
| pause() | 暂停录制。 |
| resume() | 恢复录制。 |
| reset() | 重置录制。 |
| setRecorderLocation(float latitude, float longitude) | 设置视频的经纬度。 |
| setOutputFormat(int outputFormat) | 设置输出文件格式。 |
| getVideoSurface() | 获取视频窗口。 |
| setRecorderProfile(RecorderProfile profile) | 设置媒体录制配置信息。 |
| registerRecorderListener(IRecorderListener listener) | 注册媒体录制回调。 |
| release() | 释放媒体录制资源。 |

| 接口名 | 功能描述 |
| --- | --- |

**表 1** 视频录制类 Recorder 的主要接口

### 1.1.5.3 开发步骤

1. 调用 Recorder()方法，创建 Recorder 实例。
2. 调用 setOutputFormat(int outputFormat)方法，设置录制文件存储格式。
3. 构造数据源对象，并调用 Recorder 实例的 setSource(Source source)方法，设置媒体源，代码示例如下：

```
Recorder recorder = new Recorder();
FileDescriptor fd = in.getFD();
Source source = new Source(fd);
source.setRecorderAudioSource(Recorder.AudioSource.DEFAULT);
recorder.setSource(source);
```

4. （可选）构造音频属性 AudioProperty 对象（不设置音频则是只录视频），并调用 Recorder 实例的 setAudioProperty(AudioProperty property)方法，设置录制的音频属性，代码示例如下：

```
final int AUDIO_NUM_CHANNELS_STEREO = 2;
final int AUDIO_SAMPLE_RATE_HZ = 8000;
AudioProperty audioProperty = new AudioProperty.Builder()
                        .setRecorderNumChannels(AUDIO_NUM_CHANNELS_STEREO)
                        .setRecorderSamplingRate(AUDIO_SAMPLE_RATE_HZ)
                        .setRecorderAudioEncoder(Recorder.AudioEncoder.DEFAULT)
                        .build();
recorder.setAudioProperty(audioProperty);
```

5. 构造存储属性 StorageProperty 对象，并调用 Recorder 实例的 setStorageProperty(StorageProperty property)方法，设置录制的存储属性，代码示例如下：

```
String path = "/path/audiotestRecord.mp4";
StorageProperty storageProperty = new StorageProperty.Builder()
                        .setRecorderPath(path)
                        .setRecorderMaxDurationMs(-1)
                        .setRecorderMaxFileSizeBytes(-1)
                        .build();
recorder.setStorageProperty(storageProperty);
```

6. （可选）构造视频属性 VideoProperty 对象，并调用 Recorder 实例的 setVideoProperty(VideoProperty property) 方法，设置录制的视频属性，代码示例如下：

```
1.  VideoProperty videoProperty = new VideoProperty.Builder()
2.                          .setRecorderVideoEncoder(Recorder.VideoEncoder.DEFAULT)
3.                          .setRecorderWidth(1080)
4.                          .setRecorderDegrees(0)
5.                          .setRecorderHeight(800)
6.                          .setRecorderBitRate(10000000)
7.                          .setRecorderRate(30)
8.                          .build();
9.  recorder.setVideoProperty(videoProperty);
```

7. 调用 prepare()，准备录制。

8. （可选）构造录制回调，首先构造对象 IRecorderListener，IRecorderListener 需要实现 onError(int what, int extra)，实现录制过程收到错误信息时做相应的操作。下面的代码例子中录制异常时，打印了相关的日志信息，代码示例如下：

```
1.  IRecorderListener listener = new RecorderErrorAndInfoListener() {
2.      @Override
3.      public void onError(int what, int extra) {
4.          HiLog.error("EncodeWriteFileListener onError what:%{public}d, extra:%{public}d", what, extra);
5.      }
6.  }
```

9. 调用 start()方法，开始录制。

10. （可选）调用 pause()方法和 resume()方法，可以实现暂停和恢复录制。
11. 调用 stop()方法停止录制。
12. 录制结束后，调用 release()释放资源。

### 1.1.6 视频提取开发指导

#### 1.1.6.1 场景介绍

视频提取主要工作是将多媒体文件中的音视频数据进行分离，提取出音频、视频数据源。

### 1.1.6.2 接口说明

| 接口名 | 功能描述 |
|---|---|
| Extractor() | 创建 Extractor 实例。 |
| setSource(Source source) | 设置媒体播放源。 |
| getStreamFormat(int id) | 获取对应索引的轨道数据的格式。 |
| getTotalStreams() | 获取媒体文件中总轨道数。 |
| selectStream(int id) | 根据轨道号选择媒体文件中对应的轨道。 |
| unselectStream(int id) | 取消轨道选择。 |
| rewindTo(long microseconds, int mode) | 根据时间和 mode 跳转到指定帧。 |
| next() | 跳转到下一帧。 |
| readBuffer(ByteBuffer buf, int offset) | 读取解复用后的数据。 |
| getStreamId() | 获取当前轨道号。 |
| getFrameTimestamp() | 获取当前媒体数据帧的时间戳。 |
| getFrameSize() | 获取当前媒体数据帧的数据大小。 |
| getFrameType() | 获取当前媒体数据帧的 flags。 |
| release() | 释放资源。 |

**表 1** 视频提取类 Extractor 的主要接口

### 1.1.6.3 开发步骤

1. 调用 Extractor()方法创建 Extractor 实例。

2. 构造数据源对象，并调用 Extractor 实例的 setSource(Source source)方法，设置媒体源，代码示例如下：

```
1. Extractor extractor = new Extractor();
2. FileDescriptor fd = in.getFD();
3. Source source = new Source(fd);
4. extractor.setSource(source);
```

3. 调用 getTotalStreams()方法获取媒体的轨道数量。
4. 调用 selectStream(int id)方法选择特定轨道的数据，进行提取。
5. （可选）调用 unselectStream(int id)方法取消选择轨道。
6. （可选）调用 rewindTo(long microseconds, int mode)方法实现提取过程中的跳转指定位置。
7. 调用 readBuffer(ByteBuffer buf, int offset)方法，可以实现获取提取出来的 Buffer 数据功能。
8. 调用 next()方法，实现提取下一帧的功能。
9. （可选）调用 getMediaStreamId()方法，可以实现获取当前选择的轨道编号的功能。
10. （可选）调用 getFrameTimestamp()方法，可以实现获取当前轨道内媒体数据帧时间戳的功能。
11. （可选）调用 getFrameSize()方法，可以实现获取当前轨道的媒体数据帧大小的功能。
12. （可选）调用 getFrameType()方法，可以实现获取当前轨道的媒体数据帧 flags 的功能。
13. 提取结束后，调用 release()释放资源。

### 1.1.7 媒体描述信息开发指导

#### 1.1.7.1 场景介绍

媒体描述信息主要工作是支持多媒体的相关描述信息的存取。

#### 1.1.7.2 接口说明

| 接口名 | 功能描述 |
|---|---|
| getMediaId() | 获取媒体标识。 |
| getTitle() | 获取媒体标题。 |
| getSubTitle() | 获取媒体副标题。 |
| getDescription() | 获取媒体描述信息。 |

| 接口名 | 功能描述 |
| --- | --- |
| getIcon() | 获取媒体图标。 |
| getIconUri() | 获取媒体图标的 Uri。 |
| getExtras() | 获取媒体添加的额外信息，例如应用和系统使用的内部信息。 |
| getMediaUri() | 获取媒体内容的 Uri。 |
| marshalling(Parcel parcel) | 将一个 AVDescription 对象写入到 Parcel 对象。 |
| unmarshalling(Parcel parcel) | 将一个 Parcel 对象写入到 AVDescription 对象。 |

**表 1** 媒体描述信息类 AVDescription 的主要接口

| 接口名 | 功能描述 |
| --- | --- |
| setMediaId(String mediaId) | 设置媒体标识。 |
| setTitle(CharSequence title) | 设置媒体标题。 |
| setSubTitle(CharSequence subTitle) | 设置媒体副标题。 |
| setDescription(String description) | 设置媒体描述信息。 |
| setIcon(PixelMap icon) | 设置媒体图标。 |
| setIconUri(Uri iconUri) | 设置媒体图标的 Uri。 |
| setExtras(PacMap extras) | 设置媒体的额外信息，例如应用和系统使用的内部信息。 |
| setIMediaUri(Uri mediaUri) | 设置媒体的 Uri。 |
| build() | 构造方法。 |

**表 2** 媒体描述信息内部静态类 AVDescription.Builder 的主要接口

#### 1.1.7.3 开发步骤

1. 调用 AVDescription.Builder 类的 build 方法创建 AVDescription 实例。代码示例如下：

```
1.  AVDescription avDescription = new AVDescription.Builder().setExtras(null)
2.                  .setMediaId("1")
3.                  .setDescription("Description")
4.                  .setIconUri(iconUri)
5.                  .setIMediaUri(mediaUri)
6.                  .setExtras(pacMap)
7.                  .setIcon(pixelMap)
8.                  .setTitle("title")
9.                  .setSubTitle("subTitle")
10.                 .build();
```

2. （可选）根据已有的 AVDescription 对象，可以获取媒体的描述信息，如获取媒体 Uri，代码示例如下：

```
1.  Uri uri = avDescription.getMediaUri();
```

3. （可选）根据已有的 AVDescription 对象，可以将媒体的描述信息写入 Parcel 对象，代码示例如下：

```
1.  Parcel parcel = Parcel.create();
2.  boolean result = avDescription.marshalling(parcel);
```

4. （可选）根据已有的 Parcel 对象，可以读取到 AVDescription 对象，实现媒体描述信息的写入，代码示例如下：

```
1.  boolean result = avDescription.unmarshalling(parcel);
```

### 1.1.8 媒体元数据开发指导

#### 1.1.8.1 场景介绍

媒体元数据主要用于媒体数据的存放和读取，包含诸如媒体资源的描述、创建日期、作者、封面图片等等。

### 1.1.8.2 接口说明

| 接口名 | 功能描述 |
|---|---|
| Builder() | 媒体元数据构造器的构造函数。 |
| Builder(AVMetadata source) | 媒体元数据构造器的带参构造函数。 |
| setText(String key, CharSequence value) | 用于存储媒体标题等信息。 |
| setString(String key, String value) | 用于存储媒体作者、艺术家、描述等。 |
| setLong(String key, long value) | 用于存储媒体 ID、媒体时长等信息。 |
| setPixelMap(String key, PixelMap value) | 用于存储媒体元数据相关的图片资源。 |
| build() | 媒体元数据生成函数。 |

**表 1** 媒体元数据存放类 AVMetadata.Builder 的主要接口

| 接口名 | 功能描述 |
|---|---|
| hasKey(String key) | 媒体元数据中是否包含某一个 key 的数据。 |
| getText(String key) | 获取 text 类型的 key 的数据，比如获取媒体标题等信息。 |
| getString(String key) | 获取 String 类型 key 的数据，比如获取媒体作者、艺术家、描述等。 |
| getLong(String key) | 获取 Long 类型 key 数据，比如获取媒体 ID、媒体时长等信息。 |
| getKeysSet() | 获取媒体元数据的集合。 |
| getPixelMap(String key) | 获取 PixelMap 类型 key 数据，获取媒体元数据相关的图片资源。 |
| marshalling(Parcel in) | 将一个 AVMetadata 对象写入到 Parcel 对象。 |
| getAVDescription() | 获取媒体的简要描述信息。 |

| 接口名 | 功能描述 |
| --- | --- |
| **表 2** 媒体元数据类 AVMetadata 的主要接口 | |

### 1.1.8.3 开发步骤

1. 调用 AVMetadata.Builder 类的 build 方法创建 AVMetadata 实例。代码示例如下：

```
1. AVMetadata avMetadata = new AVMetadata.Builder().setString(AVMetadata.AVTextKey.MEDIA_ID, "illuminate.mp3")
2.                 .setString(AVMetadata.AVTextKey.TITLE, "title")
3.                 .setString(AVMetadata.AVTextKey.ARTIST, "artist")
4.                 .setString(AVMetadata.AVTextKey.ALBUM, "album")
5.                 .setString(AVMetadata.AVTextKey.DISPLAY_SUBTITLE, "display_subtitle")
6.                 .setPixelMap(AVMetadata.AVPixelMapKey.DISPLAY_ICON_URI, pixelmap)
7.                 .build();
```

2. (可选)根据已有的 AVMetadata 对象，可以获取媒体元数据信息，如获取媒体标题等，代码示例如下：

```
1. String title = avMetadata.getString(AVMetadata.AVTextKey.TITLE);
```

3. 我们需要结合 AVSession 使用，将已有的媒体元数据 AVMetadata 对象下发给应用，具体参考 AVSession 使用，示例如下：

```
1. mediaSession.setAVMetadata(avMetadata);
```

4. 应用获取媒体元数据一般结合 AVControllerCallback 相关类使用，通过 onAVMetadataChanged 回调获取媒体元数据。

```
1. public class Callback extends AVControllerCallback {
2.     @Override
3.     public void onAVMetadataChanged(AVMetadata metadata) {
4.         // 歌曲信息回调
5.         AVDescription description = metadata.getAVDescription();
6.         // 获取标题
7.         String title = description.getTitle().toString();
8.         CharSequence sequence = metadata.getText(AVMetadata.AVTextKey.TITLE);
9.         if (sequence != null) {
10.            title = metadata.getText(AVMetadata.AVTextKey.TITLE).toString();
11.        }
12.        // 设置媒体 title
13.        musicTitle.setText(title);
```

```
14.             // 获取曲目专封面
15.             PixelMap iconPixelMap = description.getIcon();
16.             // 设置歌曲封面图
17.             musicCover.setPixelMap(iconPixelMap);
18.         }
19.     }
```

## 1.2 图像

### 1.2.1 概述

HarmonyOS 图像模块支持图像业务的开发，常见功能如图像解码、图像编码、基本的位图操作、图像编辑等。当然，也支持通过接口组合来实现更复杂的图像处理逻辑。

#### 基本概念

- **图像解码**

  图像解码就是不同的存档格式图片（如 JPEG、PNG 等）解码为无压缩的位图格式，以方便在应用或者系统中进行相应的处理。

- **PixelMap**

  PixelMap 是图像解码后无压缩的位图格式，用于图像显示或者进一步的处理。

- **渐进式解码**

  渐进式解码是在无法一次性提供完整图像文件数据的场景下，随着图像文件数据的逐步增加，通过多次增量解码逐步完成图像解码的模式。

- **预乘**

  预乘时，RGB 各通道的值被替换为原始值乘以 Alpha 通道不透明的比例（0~1）后的值，方便后期直接合成叠加；不预乘指 RGB 各通道的数值是图像的原始值，与 Alpha 通道的值无关。

- **图像编码**

图像编码就是将无压缩的位图格式，编码成不同格式的存档格式图片（JPEG、PNG 等），以方便在应用或者系统中进行相应的处理。

**约束与限制**

为及时释放本地资源，建议在图像解码的 ImageSource 对象、位图图像 PixelMap 对象或图像编码的 ImagePacker 对象使用完成后，主动调用 release()方法。

### 1.2.2 图像解码开发指导

#### 1.2.2.1 场景介绍

图像解码就是将所支持格式的存档图片解码成统一的 PixelMap 图像，用于后续图像显示或其他处理，比如旋转、缩放、裁剪等。当前支持格式包括 JPEG、PNG、GIF、HEIF、WebP、BMP。

#### 1.2.2.2 接口说明

ImageSource 主要用于图像解码。

| 接口名 | 描述 |
| --- | --- |
| create(String pathName, SourceOptions opts) | 从图像文件路径创建图像数据源。 |
| create(InputStream is, SourceOptions opts) | 从输入流创建图像数据源。 |
| create(byte[] data, SourceOptions opts) | 从字节数组创建图像源。 |
| create(byte[] data, int offset, int length, SourceOptions opts) | 从字节数组指定范围创建图像源。 |

| 接口名 | 描述 |
| --- | --- |
| create(File file, SourceOptions opts) | 从文件对象创建图像数据源。 |
| create(FileDescriptor fd, SourceOptions opts) | 从文件描述符创建图像数据源。 |
| createIncrementalSource(SourceOptions opts) | 创建渐进式图像数据源。 |
| createIncrementalSource(IncrementalSourceOptions opts) | 创建渐进式图像数据源，支持设置渐进式数据更新模式。 |
| createPixelmap(DecodingOptions opts) | 从图像数据源解码并创建 PixelMap 图像。 |
| createPixelmap(int index, DecodingOptions opts) | 从图像数据源解码并创建 PixelMap 图像，如果图像数据源支持多张图片的话，支持指定图像索引。 |
| updateData(byte[] data, boolean isFinal) | 更新渐进式图像源数据。 |
| updateData(byte[] data, int offset, int length, boolean isFinal) | 更新渐进式图像源数据，支持设置输入数据的有效数据范围。 |
| getImageInfo() | 获取图像基本信息。 |
| getImageInfo(int index) | 根据特定的索引获取图像基本信息。 |
| getSourceInfo() | 获取图像源信息。 |
| release() | 释放对象关联的本地资源。 |

**表 1** ImageSource 的主要接口

### 1.2.2.3 普通解码开发步骤

1. 创建图像数据源 ImageSource 对象，可以通过 SourceOptions 指定数据源的格式信息，此格式信息仅为给解码器的提示，正确提供能帮助提高解码效率，如果不设置或设置不正确，会自动检测正确的图像格式。不使用该选项时，可以将 create 接口传入的 SourceOptions 设置为 null。

```
1.  ImageSource.SourceOptions srcOpts = new ImageSource.SourceOptions();
```

```
2.    srcOpts.formatHint = "image/png";
3.    String pathName = "/path/to/image.png";
4.    ImageSource imageSource = ImageSource.create(pathName, srcOpts);
5.    ImageSource imageSourceNoOptions = ImageSource.create(pathName, null);
```

2. 设置解码参数，解码获取 PixelMap 图像对象，解码过程中同时支持图像处理操作。设置 desiredRegion 支持按矩形区域裁剪，如果设置为全 0，则不进行裁剪。设置 desiredSize 支持按尺寸缩放，如果设置为全 0，则不进行缩放。设置 rotateDegrees 支持旋转角度，以图像中心点顺时针旋转。如果只需要解码原始图像，不使用该选项时，可将给 createPixelMap 传入的 DecodingOptions 设置为 null。

```
1.    // 普通解码叠加旋转、缩放、裁剪
2.    ImageSource.DecodingOptions decodingOpts = new ImageSource.DecodingOptions();
3.    decodingOpts.desiredSize = new Size(100, 2000);
4.    decodingOpts.desiredRegion = new Rect(0, 0, 100, 100);
5.    decodingOpts.rotateDegrees = 90;
6.    PixelMap pixelMap = imageSource.createPixelmap(decodingOpts);
7.
8.    // 普通解码
9.    PixelMap pixelMapNoOptions = imageSource.createPixelmap(null);
```

3. 解码完成获取到 PixelMap 对象后，可以进行后续处理，比如渲染显示等。

#### 1.2.2.4 渐进式解码开发步骤

1. 创建渐进式图像数据源 ImageSource 对象，可以通过 SourceOptions 指定数据源的格式信息，此格式信息仅为提示，如果填写不正确，会自动检测正确的图像格式，使用 IncrementalSourceOptions 指定图像数据的更新方式为渐进式更新。

```
1.    ImageSource.SourceOptions srcOpts = new ImageSource.SourceOptions();
2.    srcOpts.formatHint = "image/jpeg";
3.    ImageSource.IncrementalSourceOptions incOpts = new ImageSource.IncrementalSourceOptions();
4.    incOpts.opts = srcOpts;
5.    incOpts.mode = ImageSource.UpdateMode.INCREMENTAL_DATA;
6.    imageSource = ImageSource.createIncrementalSource(incOpts);
```

2. 渐进式更新数据，在未获取到全部图像时，支持先更新部分数据来尝试解码，更新数据时设置 isFinal 为 false，当获取到全部数据后，最后一次更新数据时设置 isFinal 为 true，表示数据更新完毕。设置解码参数同普通解码。

```
1.    // 获取到一定的数据时尝试解码
2.    imageSource.updateData(data, 0, bytes, false);
3.    ImageSource.DecodingOptions decodingOpts = new ImageSource.DecodingOptions();
```

```
4.  PixelMap pixelMap = imageSource.createPixelmap(decodingOpts);
5.
6.  // 更新数据再次解码，重复调用直到数据全部更新完成
7.  imageSource.updateData(data, 0, bytes, false);
8.  PixelMap pixelMap = imageSource.createPixelmap(decodingOpts);
9.
10. // 数据全部更新完成时需要传入 isFinal 为 true
11. imageSource.updateData(data, 0, bytes, true);
12. PixelMap pixelMap = imageSource.createPixelmap(decodingOpts);
```

3. 解码完成获取到 PixelMap 对象后，可以进行后续处理，比如渲染显示等。

### 1.2.3 图像编码开发指导

#### 1.2.3.1 场景介绍

图像编码就是将 PixelMap 图像编码成不同存档格式图片，用于后续其他处理，比如保存、传输等。当前仅支持 JPEG 格式。

#### 1.2.3.2 接口说明

ImagePacker 主要用于图像编码。

| 接口名 | 描述 |
| --- | --- |
| create() | 创建图像打包器实例。 |
| initializePacking(byte[] data, PackingOptions opts) | 初始化打包任务，将字节数组设置为打包后输出目的。 |
| initializePacking(byte[] data, int offset, PackingOptions opts) | 初始化打包任务，将带偏移量的字节数组设置为打包后输出目的。 |
| initializePacking(OutputStream outputStream, PackingOptions opts) | 初始化打包任务，将输出流设置为打包后输出目的。 |

| 接口名 | 描述 |
| --- | --- |
| addImage(PixelMap pixelmap) | 将 PixelMap 对象添加到图像打包器中。 |
| addImage(ImageSource source) | 将图像数据源 ImageSource 中图像添加到图像打包器中。 |
| addImage(ImageSource source, int index) | 将图像数据源 ImageSource 中指定图像添加到图像打包器中。 |
| finalizePacking() | 完成图像打包任务。 |
| release() | 释放对象关联的本地资源。 |

**表 1** 图像编码类 ImagePacker 的主要接口

### 1.2.3.3 开发步骤

1. 创建图像编码 ImagePacker 对象。

```
ImagePacker imagePacker = ImagePacker.create();
```

2. 设置编码输出流和编码参数。设置 format 为编码的图像格式，当前支持 jpeg 格式。设置 quality 为图像质量，范围从 0-100，100 为最佳质量。

```
FileOutputStream outputStream = new FileOutputStream("/path/to/packed.file");
ImagePacker.PackingOptions packingOptions = new ImagePacker.PackingOptions();
packingOptions.format = "image/jpeg";
packingOptions.quality = 90;
boolean result = imagePacker.initializePacking(outputStream, packingOptions);
```

3. 添加需要编码的 PixelMap 对象，进行编码操作。

```
result = imagePacker.addImage(pixelMap);
long dataSize = imagePacker.finalizePacking();
```

4. 编码输出完成后，可以进行后续处理，比如保存、传输等。

## 1.2.4 位图操作开发指导

### 1.2.4.1 场景介绍

位图操作就是指对 PixelMap 图像进行相关的操作，比如创建、查询信息、读写像素数据等。

### 1.2.4.2 接口说明

| 接口名 | 描述 |
|---|---|
| create(InitializationOptions opts) | 根据图像大小、像素格式、alpha 类型等初始化选项创建 PixelMap。 |
| create(int[] colors, InitializationOptions opts) | 根据图像大小、像素格式、alpha 类型等初始化选项,以像素颜色数组为数据源创建 PixelMap。 |
| create(int[] colors, int offset, int stride, InitializationOptions opts) | 根据图像大小、像素格式、alpha 类型等初始化选项,以像素颜色数组、起始偏移量、行像素大小描述的数据源创建 PixelMap。 |
| create(PixelMap source, InitializationOptions opts) | 根据图像大小、像素格式、alpha 类型等初始化选项,以源 PixelMap 为数据源创建 PixelMap。 |
| create(PixelMap source, Rect srcRegion, InitializationOptions opts) | 根据图像大小、像素格式、alpha 类型等初始化选项,以源 PixelMap、源裁剪区域描述的数据源创建 PixelMap。 |
| getBytesNumberPerRow() | 获取每行像素数据占用的字节数。 |
| getPixelBytesCapacity() | 获取存储 Pixelmap 像素数据的内存容量。 |
| isEditable() | 判断 PixelMap 是否允许修改。 |
| isSameImage(PixelMap other) | 判断两个图像是否相同，包括 ImageInfo 属性信息和像素数据。 |
| readPixel(Position pos) | 读取指定位置像素的颜色值,返回的颜色格式为 PixelFormat.ARGB_8888。 |

| 接口名 | 描述 |
|---|---|
| readPixels(int[] pixels, int offset, int stride, Rect region) | 读取指定区域像素的颜色值,输出到以起始偏移量、行像素大小描述的像素数组，返回的颜色格式为 PixelFormat.ARGB_8888。 |
| readPixels(Buffer dst) | 读取像素的颜色值到缓冲区,返回的数据是 PixelMap 中像素数据的原样拷贝，即返回的颜色数据格式与 PixelMap 中像素格式一致。 |
| resetConfig(Size size, PixelFormat pixelFormat) | 重置 PixelMap 的大小和像素格式配置，但不会改变原有的像素数据也不会重新分配像素数据的内存，重置后图像数据的字节数不能超过 PixelMap 的内存容量。 |
| setAlphaType(AlphaType alphaType) | 设置 PixelMap 的 Alpha 类型。 |
| writePixel(Position pos, int color) | 向指定位置像素写入颜色值,写入颜色格式为 PixelFormat.ARGB_8888。 |
| writePixels(int[] pixels, int offset, int stride, Rect region) | 将像素颜色数组、起始偏移量、行像素的个数描述的源像素数据写入 PixelMap 的指定区域,写入颜色格式为 PixelFormat.ARGB_8888。 |
| writePixels(Buffer src) | 将缓冲区描述的源像素数据写入 PixelMap,写入的数据将原样覆盖 PixelMap 中的像素数据，即写入数据的颜色格式应与 PixelMap 的配置兼容。 |
| writePixels(int color) | 将所有像素都填充为指定的颜色值,写入颜色格式为 PixelFormat.ARGB_8888。 |
| getPixelBytesNumber() | 获取全部像素数据包含的字节数。 |
| setBaseDensity(int baseDensity) | 设置 PixelMap 的基础像素密度值。 |
| getBaseDensity() | 获取 PixelMap 的基础像素密度值。 |
| setUseMipmap(boolean useMipmap) | 设置 PixelMap 渲染是否使用 mipmap。 |
| useMipmap() | 获取 PixelMap 渲染是否使用 mipmap。 |
| getNinePatchChunk() | 获取图像的 NinePatchChunk 数据。 |
| getFitDensitySize(int targetDensity) | 获取适应目标像素密度的图像缩放的尺寸。 |
| getImageInfo() | 获取图像基本信息。 |

| 接口名 | 描述 |
| --- | --- |
| release() | 释放对象关联的本地资源。 |

**表 1** 位图操作类 PixelMap 的主要接口

### 1.2.4.3 开发步骤

1. 创建位图对象 PixelMap。

```
// 指定初始化选项创建
PixelMap pixelMap2 = PixelMap.create(initializationOptions);

// 从像素颜色数组创建
int[] defaultColors = new int[] {5, 5, 5, 5, 6, 6, 3, 3, 3, 0};
PixelMap.InitializationOptions initializationOptions = new PixelMap.InitializationOptions();
initializationOptions.size = new Size(3, 2);
initializationOptions.pixelFormat = PixelFormat.ARGB_8888;
PixelMap pixelMap1 = PixelMap.create(defaultColors, initializationOptions);

// 以另外一个 PixelMap 作为数据源创建
PixelMap pixelMap3 = PixelMap.create(pixelMap2, initializationOptions);
```

2. 从位图对象中获取信息。

```
long capacity = pixelMap.getPixelBytesCapacity();
long bytesNumber = pixelMap.getPixelBytesNumber();
int rowBytes = pixelMap.getBytesNumberPerRow();
byte[] ninePatchData = pixelMap.getNinePatchChunk();
```

3. 读写位图像素数据。

```
// 读取指定位置像素
int color = pixelMap.readPixel(new Position(1, 1));

// 读取指定区域像素
int[] pixelArray = new int[50];
Rect region = new Rect(0, 0, 10, 5);
pixelMap.readPixels(pixelArray, 0, 10, region);

```

```
9.  // 读取像素到 Buffer
10. IntBuffer pixelBuf = IntBuffer.allocate(50);
11. pixelMap.readPixels(pixelBuf);
12.
13. // 在指定位置写入像素
14. pixelMap.writePixel(new Position(1, 1), 0xFF112233);
15.
16. // 在指定区域写入像素
17. pixelMap.writePixels(pixelArray, 0, 10, region);
18.
19. // 写入 Buffer 中的像素
20. pixelMap.writePixels(intBuf);
```

### 1.2.5 图像属性解码开发指导

#### 1.2.5.1 场景介绍

图像属性解码就是获取图像中包含的属性信息，比如 EXIF 属性。

#### 1.2.5.2 接口说明

图像属性解码的功能主要由 ImageSource 和 ExifUtils 提供。

| 接口名 | 描述 |
| --- | --- |
| getThumbnailInfo() | 获取嵌入图像文件的缩略图的基本信息。 |
| getImageThumbnailBytes() | 获取嵌入图像文件缩略图的原始数据。 |
| getThumbnailFormat() | 获取嵌入图像文件缩略图的格式。 |

**表 1** ImageSource 的主要接口

| 接口名 | 描述 |
| --- | --- |
| getLatLong(ImageSource imageSource) | 获取嵌入图像文件的经纬度信息。 |

| 接口名 | 描述 |
| --- | --- |
| getAltitude(ImageSource imageSource, double defaultValue) | 获取嵌入图像文件的海拔信息。 |

表 2 ExifUtils 的主要接口

#### 1.2.5.3 开发步骤

1. 创建图像数据源 ImageSource 对象，可以通过 SourceOptions 指定数据源的格式信息，此格式信息仅为给解码器的提示，正确提供能帮助提高解码效率，如果不设置或设置不正确，会自动检测正确的图像格式。

```
ImageSource.SourceOptions srcOpts = new ImageSource.SourceOptions();
srcOpts.formatHint = "image/jpeg";
String pathName = "/path/to/image.jpg";
ImageSource imageSource = ImageSource.create(pathName, srcOpts);
```

2. 获取缩略图信息。

```
int format = imageSource.getThumbnailFormat();
byte[] thumbnailBytes = imageSource.getImageThumbnailBytes();

// 将缩略图解码为 PixelMap 对象
ImageSource.DecodingOptions decodingOpts = new ImageSource.DecodingOptions();
PixelMap thumbnailPixelmap = imageSource.createThumbnailPixelmap(decodingOpts, false);
```

## 1.3 相机

### 1.3.1 概述

HarmonyOS 相机模块支持相机业务的开发，开发者可以通过已开放的接口实现相机硬件的访问、操作和新功能开发，最常见的操作如：预览、拍照、连拍和录像等。

#### 1.3.1.1 基本概念

- **相机静态能力**

用于描述相机的固有能力的一系列参数，比如朝向、支持的分辨率等信息。

- **物理相机**

物理相机就是独立的实体摄像头设备。物理相机 ID 是用于标志每个物理摄像头的唯一字串。

- **逻辑相机**

逻辑相机是多个物理相机组合出来的抽象设备，逻辑相机通过同时控制多个物理相机设备来完成相机某些功能，如大光圈、变焦等功能。逻辑摄像机 ID 是一个唯一的字符串，标识多个物理摄像机的抽象能力。

- **帧捕获**

相机启动后对帧的捕获动作统称为帧捕获。主要包含单帧捕获、多帧捕获、循环帧捕获。

- **单帧捕获**

指的是相机启动后，在帧数据流中捕获一帧数据，常用于普通拍照。

- **多帧捕获**

指的是相机启动后，在帧数据流中连续捕获多帧数据，常用于连拍。

- **循环帧捕获**

指的是相机启动后，在帧数据流中一直捕获帧数据，常用于预览和录像。

#### 1.3.1.2 约束与限制

- 在同一时刻只能有一个相机应用在运行中。
- 相机模块内部有状态控制，开发者必须按照指导文档中的流程进行接口的顺序调用，否则可能会出现调用失败等问题。
- 为了开发的相机应用拥有更好的兼容性，在创建相机对象或者参数相关设置前请务必进行能力查询。

### 1.3.2 设备开发指导

#### 1.3.2.1 相机开发流程

相机模块主要工作是给相机应用开发者提供基本的相机 API 接口，用于使用相机系统的功能，进行相机硬件的访问、操作和新功能开发。相机的开发流程如图所示：

**图 1** 相机开发流程

### 1.3.2.2 接口说明

相机模块为相机应用开发者提供了 3 个包的内容，包括方法、枚举、以及常量/变量，方便开发者更容易地实现相机功能。详情请查阅对应开发场景。

| 包名 | 功能 |
| --- | --- |
| ohos.media.camera.CameraKit | 相机功能入口类。获取当前支持的相机列表及其静态能力信息，创建相机对象。 |
| ohos.media.camera.device | 相机设备操作类。提供相机能力查询、相机配置、相机帧捕获、相机状态回调等功能。 |
| ohos.media.camera.params | 相机参数类。提供相机属性、参数和操作结果的定义。 |

### 1.3.2.3 相机权限申请

在使用相机之前，需要申请相机的相关权限，保证应用拥有相机硬件及其他功能权限，应用权限的介绍请参考 权限章节，相机涉及权限如下表。

| 权限名称 | 权限属性值 | 是否必选 |
| --- | --- | --- |
| 相机权限 | ohos.permission.CAMERA | 必选 |
| 录音权限 | ohos.permission.MICROPHONE | 可选（需要录像时申请） |
| 存储权限 | ohos.permission.WRITE_USER_STORAGE | 可选（需要保存图像及视频到设备的外部存储时申请） |
| 位置权限 | ohos.permission.LOCATION | 可选（需要保存图像及视频位置信息时申请） |

**表 1** 相机权限列表

### 1.3.2.4 相机设备创建

CameraKit 类是相机的入口 API 类，用于获取相机设备特性、打开相机，其接口如下表。

| 接口名 | 描述 |
| --- | --- |
| createCamera(String cameraId, CameraStateCallback callback, EventHandler handler) | 创建相机对象。 |
| getCameraAbility(String cameraId) | 获取指定逻辑相机或物理相机的静态能力。 |
| getCameraIds() | 获取当前逻辑相机列表。 |
| getCameraInfo(String cameraId) | 获取指定逻辑相机的信息。 |
| getInstance(Context context) | 获取 CameraKit 实例。 |
| registerCameraDeviceCallback(CameraDeviceCallback callback, EventHandler handler) | 注册相机使用状态回调。 |
| unregisterCameraDeviceCallback(CameraDeviceCallback callback) | 注销相机使用状态回调。 |

**表 2** CameraKit 的主要接口

基于 HarmonyOS 实现一个相机应用，无论将来想应用到哪个或者哪些设备上，都必须先创建一个独立的相机设备，然后才能继续相机的其他操作。相机设备创建的建议步骤如下：

1. 通过 CameraKit.getInstance(Context context)方法获取唯一的 CameraKit 对象是创建新的相机应用的第一步操作。

```
1.  private void openCamera(){
2.      // 获取 CameraKit 对象
3.      CameraKit cameraKit = CameraKit.getInstance(context);
4.      if (cameraKit == null) {
5.          // 处理 cameraKit 获取失败的情况
6.      }
7.  }
```

如果此步骤操作失败，相机可能被占用或无法使用。如果被占用，必须等到相机释放后才能重新获取 CameraKit 对象。

2. 通过 getCameraIds()方法，获取当前使用的设备支持的逻辑相机列表。逻辑相机列表中存储了当前设备拥有的所有逻辑相机 ID，如果列表不为空，则列表中的每个 ID 都支持独立创建相机对象；否则，说明正在使用的设备无可用的相机，不能继续后续的操作。

```
1.  try {
2.      // 获取当前设备的逻辑相机列表
3.      String[] cameraIds = cameraKit.getCameraIds();
4.      if (cameraIds.length <= 0) {
5.          HiLog.error("cameraIds size is 0");
6.      }
7.  } catch (IllegalStateException e) {
8.      // 处理异常
9.  }
```

还可以继续查询指定相机 ID 的静态信息：

调用 getDeviceLinkType(String physicalId)方法获取物理相机连接方式；

调用 getCameraInfo(String cameraId)方法查询相机硬件朝向等信息；

调用 getCameraAbility(String cameraId)方法查询相机能力信息（比如支持的分辨率列表等）。

| 接口名 | 描述 |
| --- | --- |
| getDeviceLinkType(String physicalId) | 获取物理相机连接方式。 |
| getFacingType() | 获取相机朝向信息。 |
| getLogicalId() | 获取逻辑相机 ID。 |
| getPhysicalIdList() | 获取对应的物理相机 ID 列表。 |

**表 3** CameraInfo 的主要接口

| 接口名 | 描述 |
| --- | --- |
| getSupportedSizes(int format) | 根据格式查询输出图像的分辨率列表。 |

| 接口名 | 描述 |
| --- | --- |
| getSupportedSizes(Class<T> clazz) | 根据 Class 类型查询分辨率列表。 |
| getParameterRange(ParameterKey.Key<T> parameter) | 获取指定参数能够设置的值范围。 |
| getPropertyValue(PropertyKey.Key<T> property) | 获取指定属性对应的值。 |
| getSupportedAeMode() | 获取当前相机支持的自动曝光模式。 |
| getSupportedAfMode() | 获取当前相机支持的自动对焦模式。 |
| getSupportedFaceDetection() | 获取相机支持的人脸检测类型范围。 |
| getSupportedFlashMode() | 当前相机支持的闪光灯取值范围。 |
| getSupportedParameters() | 当前相机支持的参数设置。 |
| getSupportedProperties() | 获取当前相机的属性列表。 |
| getSupportedResults() | 获取当前相机支持的参数设置可返回的结果列表。 |
| getSupportedZoom() | 获取相机支持的变焦范围。 |

**表 4** CameraAbility 的主要接口

3. 通过 createCamera(String cameraId, CameraStateCallback callback, EventHandler handler)方法，创建相机对象，此步骤执行成功意味着相机系统的硬件已经完成了上电。

```
1.  // 创建相机设备
2.  cameraKit.createCamera(cameraIds[0], cameraStateCallback, eventHandler);
```

第一个参数 cameraId 可以是上一步获取的逻辑相机列表中的任何一个相机 ID。

第二和第三个参数负责相机创建和相机运行时的数据和状态检测，请务必保证在整个相机运行周期内有效。

```
1.  private final class CameraStateCallbackImpl extends CameraStateCallback {
2.      @Override
3.      public void onCreated(Camera camera) {
4.          // 创建相机设备
```

```
5.         }
6.
7.         @Override
8.         public void onConfigured(Camera camera) {
9.             // 配置相机设备
10.        }
11.
12.        @Override
13.        public void onPartialConfigured(Camera camera) {
14.            // 当使用了 addDeferredSurfaceSize 配置了相机，会接到此回调
15.        }
16.
17.        @Override
18.        public void onReleased(Camera camera) {
19.            // 释放相机设备
20.        }
21.    }
22.
23.    // 相机创建和相机运行时的回调
24.    CameraStateCallbackImpl cameraStateCallback = new CameraStateCallbackImpl();
25.    if(cameraStateCallback ==null) {
26.        HiLog.error("cameraStateCallback is null");
27.    }
28.    import ohos.eventhandler.EventHandler;
29.    import ohos.eventhandler.EventRunner;
30.
31.    // 执行回调的 EventHandler
32.    EventHandler eventHandler = new EventHandler(EventRunner.create("CameraCb"));
33.    if(eventHandler ==null) {
34.        HiLog.error("eventHandler is null");
35.    }
```

至此，相机设备的创建已经完成。相机设备创建成功会在 CameraStateCallback 中触发 onCreated(Camera camera)回调。在进入相机设备配置前，请确保相机设备已经创建成功。否则会触发相机设备创建失败的回调，并返回错误码，需要进行错误处理后，重新执行相机设备的创建。

#### 1.3.2.5 相机设备配置

创建相机设备成功后，在 CameraStateCallback 中会触发 onCreated(Camera camera)回调，并且带回 Camera 对象，用于执行相机设备的操作。

当一个新的相机设备成功创建后，首先需要对相机进行配置，调用 configure(CameraConfig)方法实现配置。相机配置主要是设置预览、拍照、录像用到的 Surface(详见 ohos.agp.graphics.Surface)，没有配置过 Surface，相应的功能不能使用。

为了进行相机帧捕获结果的数据和状态检测，还需要在相机配置时调用 setFrameStateCallback(FrameStateCallback, EventHandler)方法设置帧回调。

```
1.  private final class CameraStateCallbackImpl extends CameraStateCallback {
2.      @Override
3.      public void onCreated(Camera camera) {
4.          cameraConfigBuilder = camera.getCameraConfigBuilder();
5.          if (cameraConfigBuilder == null) {
6.              HiLog.error("onCreated cameraConfigBuilder is null");
7.              return;
8.          }
9.          // 配置预览的 Surface
10.         cameraConfigBuilder.addSurface(previewSurface);
11.         // 配置拍照的 Surface
12.         cameraConfigBuilder.addSurface(imageReceiver.getRecevingSurface());
13.         // 配置帧结果的回调
14.         cameraConfigBuilder.setFrameStateCallback(frameStateCallbackImpl, handler);
15.         try {
16.             // 相机设备配置
17.             camera.configure(cameraConfigBuilder.build());
18.         } catch (IllegalArgumentException e) {
19.             HiLog.error("Argument Exception");
20.         } catch (IllegalStateException e) {
21.             HiLog.error("State Exception");
```

```
22.             }
23.         }
24.     }
```

相机配置成功后，在 CameraStateCallback 中会触发 onConfigured(Camera camera)回调，然后才可以执行相机帧捕获相关的操作。

| 接口名 | 描述 |
| --- | --- |
| addSurface(Surface surface) | 相机配置中增加 Surface。 |
| build() | 相机配置的构建类。 |
| removeSurface(Surface surface) | 移除先前添加的 Surface。 |
| setFrameStateCallback(FrameStateCallback callback, EventHandler handler) | 设置用于相机帧结果返回的 FrameStateCallback 和 Handler。 |
| addDeferredSurfaceSize(Size surfaceSize, Class<T> clazz) | 添加延迟 Surface 的尺寸、类型。 |
| addDeferredSurface(Surface surface) | 设置延迟的 Surface，此 Surface 的尺寸和类型必须和使用 addDeferredSurfaceSize 配置的一致。 |

**表 5** CameraConfig.Builder 的主要接口

### 1.3.2.6 相机帧捕获

Camera 操作类，包括相机预览、录像、拍照等功能接口。

| 接口名 | 描述 |
| --- | --- |
| triggerSingleCapture(FrameConfig frameConfig) | 启动相机帧的单帧捕获。 |
| triggerMultiCapture(List<FrameConfig> frameConfigs) | 启动相机帧的多帧捕获。 |

| 接口名 | 描述 |
| --- | --- |
| configure(CameraConfig config) | 配置相机。 |
| flushCaptures() | 停止并清除相机帧的捕获，包括循环帧/单帧/多帧捕获。 |
| getCameraConfigBuilder() | 获取相机配置构造器对象。 |
| getCameraId() | 获取当前相机的 ID。 |
| getFrameConfigBuilder(int type) | 获取指定类型的相机帧配置构造器对象。 |
| release() | 释放相机对象及资源。 |
| triggerLoopingCapture(FrameConfig frameConfig) | 启动或者更新相机帧的循环捕获。 |
| stopLoopingCapture() | 停止当前相机帧的循环捕获。 |

**表 6** Camera 的主要接口

**启动预览（循环帧捕获）**

用户一般都是先看见预览画面才执行拍照或者其他功能，所以对于一个普通的相机应用，预览是必不可少的。启动预览的建议步骤如下：

1. 通过 getFrameConfigBuilder(FRAME_CONFIG_PREVIEW)方法获取预览配置模板，常用帧配置项见下表，更多的帧配置项以及详细使用方法请参考 API 接口说明的 FrameConfig.Builder 部分。

| 接口名 | 描述 | 是否必选 |
| --- | --- | --- |
| addSurface(Surface surface) | 配置预览 surface 和帧的绑定。 | 是 |
| setAfMode(int afMode, Rect rect) | 配置对焦模式。 | 否 |
| setAeMode(int aeMode, Rect rect) | 配置曝光模式。 | 否 |
| setZoom(float value) | 配置变焦值。 | 否 |

| 接口名 | 描述 | 是否必选 |
| --- | --- | --- |
| setFlashMode(int flashMode) | 配置闪光灯模式。 | 否 |
| setFaceDetection(int type, boolean isEnable) | 配置人脸检测或者笑脸检测。 | 否 |
| setParameter(Key<T> key, T value) | 配置其他属性（如自拍镜像等）。 | 否 |
| setMark(Object mark) | 配置一个标签，后续可以从 FrameConfig 中通过 Object getMark()拿到标签，判断两个是否相等，相等就说明是同一个配置。 | 否 |
| setCoordinateSurface(Surface surface) | 配置坐标系基准 Surface，后续计算 Ae/Af 等区域都会基于此 Surface 为基本的中心坐标系，不设置默认使用添加的第一个 Surface。 | 否 |

**表 7** 常用帧配置项

2. 通过 triggerLoopingCapture(FrameConfig)方法实现循环帧捕获(如预览/录像)。

```
1.  private final class CameraStateCallbackImpl extends CameraStateCallback {
2.      @Override
3.      public void onConfigured(Camera camera) {
4.          // 获取预览配置模板
5.          frameConfigBuilder = camera.getFrameConfigBuilder(FRAME_CONFIG_PREVIEW);
6.          // 配置预览 Surface
7.          frameConfigBuilder.addSurface(previewSurface);
8.          previewFrameConfig = frameConfigBuilder.build();
9.          try {
10.             // 启动循环帧捕获
11.             int triggerId = camera.triggerLoopingCapture(previewFrameConfig);
12.         } catch (IllegalArgumentException e) {
13.             HiLog.error("Argument Exception");
14.         } catch (IllegalStateException e) {
15.             HiLog.error("State Exception");
16.         }
17.     }
18. }
```

经过以上的操作，相机应用已经可以正常进行实时预览了。在预览状态下，开发者还可以执行其他操作，比如：

当预览帧配置更改时，可以通过 triggerLoopingCapture(FrameConfig)方法实现预览帧配置的更新；

```
1. // 预览帧变焦值变更
2. frameConfigBuilder.setZoom(1.2f);
3. // 调用 triggerLoopingCapture 方法实现预览帧配置更新
4. triggerLoopingCapture(frameConfigBuilder.build());
```

通过 stopLoopingCapture()方法停止循环帧捕获(停止预览)。

```
1. // 停止预览帧捕获
2. camera.stopLoopingCapture(frameConfigBuilder.build())
```

**实现拍照（单帧捕获）**

拍照功能属于相机应用的最重要功能之一，而且照片质量对用户至关重要。相机模块基于相机复杂的逻辑，从应用接口层到器件驱动层都已经默认的做好了最适合用户的配置，这些默认配置尽可能地保证用户拍出的每张照片的质量。发起拍照的建议步骤如下：

1. 通过 getFrameConfigBuilder(FRAME_CONFIG_PICTURE)方法获取拍照配置模板，并且设置拍照帧配置，如下表：

| 接口名 | 描述 | 是否必选 |
| --- | --- | --- |
| FrameConfig.Builder addSurface(Surface) | 实现拍照 Surface 和帧的绑定。 | 必选 |
| FrameConfig.Builder setImageRotation(int) | 设置图片旋转角度。 | 可选 |
| FrameConfig.Builder setLocation(Location) | 设置图片地理位置信息。 | 可选 |
| FrameConfig.Builder setParameter(Key<T>, T) | 配置其他属性（如自拍镜像等）。 | 可选 |

**表 8** 常用拍照帧配置

2. 拍照前准备图像帧数据的接收实现。

```
1. // 图像帧数据接收处理对象
2. private ImageReceiver imageReceiver;
```

```
3.  // 执行回调的 EventHandler
4.  private EventHandler eventHandler = new EventHandler(EventRunner.create("CameraCb"));
5.  // 拍照支持分辨率
6.  private Size pictureSize;
7.
8.  // 单帧捕获生成图像回调 Listener
9.  private final ImageReceiver.IImageArrivalListener imageArrivalListener = new ImageReceiver.IImageArrivalListener() {
10.     @Override
11.     public void onImageArrival(ImageReceiver imageReceiver) {
12.         StringBuffer fileName = new StringBuffer("picture_");
13.         fileName.append(UUID.randomUUID()).append(".jpg"); // 定义生成图片文件名
14.         File myFile = new File(dirFile, fileName.toString()); // 创建图片文件
15.         imageSaver = new ImageSaver(imageReceiver.readNextImage(), myFile); // 创建一个读写线程任务用于保存图片
16.         eventHandler.postTask(imageSaver); // 执行读写线程任务生成图片
17.     }
18. };
19.
20. // 保存图片, 图片数据读写，及图像生成见 run 方法
21. class ImageSaver implements Runnable {
22.     private final Image myImage;
23.     private final File myFile;
24.
25.     ImageSaver(Image image, File file) {
26.         myImage = image;
27.         myFile = file;
28.     }
29.
30.     @Override
31.     public void run() {
32.         Image.Component component = myImage.getComponent(ImageFormat.ComponentType.JPEG);
33.         byte[] bytes = new byte[component.remaining()];
34.         component.read(bytes);
35.         FileOutputStream output = null;
36.         try {
37.             output = new FileOutputStream(myFile);
38.             output.write(bytes); // 写图像数据
39.         } catch (IOException e) {
40.             HiLog.error("save picture occur exception!");
```

```
41.             } finally {
42.                 myImage.release();
43.                 if (output != null) {
44.                     try {
45.                         output.close(); // 关闭流
46.                     } catch (IOException e) {
47.                         HiLog.error("image release occur exception!");
48.                     }
49.                 }
50.             }
51.         }
52.     }
53.     private void takePictureInit() {
54.         List<Size> pictureSizes = cameraAbility.getSupportedSizes(ImageFormat.JPEG); // 获取拍照支持分辨率列表
55.         pictureSize = getpictureSize(pictureSizes) // 根据拍照要求选择合适的分辨率
56.         imageReceiver = ImageReceiver.create(Math.max(pictureSize.width, pictureSize.height),
57.             Math.min(pictureSize.width, pictureSize.height), ImageFormat.JPEG, 5); // 创建 ImageReceiver 对象，注意 creat 函数中宽度要大
    于高度；5 为最大支持的图像数，请根据实际设置。
58.         imageReceiver.setImageArrivalListener(imageArrivalListener);
59.     }
```

3. 通过 triggerSingleCapture(FrameConfig)方法实现单帧捕获(如拍照)。

```
1.  private void capture() {
2.      // 获取拍照配置模板
3.      framePictureConfigBuilder = cameraDevice.getFrameConfigBuilder(FRAME_CONFIG_PICTURE);
4.      // 配置拍照 Surface
5.      framePictureConfigBuilder.addSurface(imageReceiver.getRecevingSurface());
6.      // 配置拍照其他参数
7.      framePictureConfigBuilder.setImageRotation(90);
8.      try {
9.          // 启动单帧捕获(拍照)
10.         camera.triggerSingleCapture(framePictureConfigBuilder.build());
11.     } catch (IllegalArgumentException e) {
12.         HiLog.error("Argument Exception");
13.     } catch (IllegalStateException e) {
14.         HiLog.error("State Exception");
15.     }
16. }
```

为了捕获到质量更高和效果更好的图片，还可以在帧结果中实时监测自动对焦和自动曝光的状态，一般而言，在自动对焦完成，自动曝光收敛后的瞬间是发起单帧捕获的最佳时机。

**实现连拍（多帧捕获）**

连拍功能方便用户一次拍照获取多张照片，用于捕捉精彩瞬间。同普通拍照的实现流程一致，但连拍需要使用 triggerMultiCapture(List<FrameConfig> frameConfigs)方法。

**启动录像（循环帧捕获）**

启动录像和启动预览类似，但需要另外配置录像 Surface 才能使用。

1. 录像前需要进行音视频模块的配置。

```
1.  private Source source; // 音视频源
2.  private AudioProperty.Builder audioPropertyBuilder; // 音频属性
3.  private VideoProperty.Builder videoPropertyBuilder; // 视频属性
4.  private StorageProperty.Builder storagePropertyBuilder; // 音视频存储属性
5.  private Recorder mediaRecorder; // 录像操作对象
6.  private String recordName; // 音视频文件名
7.
8.  private void initMediaRecorder() {
9.      HiLog.info("initMediaRecorder begin");
10.     videoPropertyBuilder.setRecorderBitRate(10000000); // 设置录制比特率
11.     int rotation = DisplayManager.getInstance().getDefaultDisplay(this).get().getRotation();
12.     videoPropertyBuilder.setRecorderDegrees(getOrientation(rotation)); // 设置录像方向
13.     videoPropertyBuilder.setRecorderFps(30); // 设置录制采样率
14.     videoPropertyBuilder.setRecorderHeight(Math.min(recordSize.height, recordSize.width)); // 设置录像支持的分辨率，需保证
    width > height
15.     videoPropertyBuilder.setRecorderWidth(Math.max(recordSize.height, recordSize.width));
16.     videoPropertyBuilder.setRecorderVideoEncoder(Recorder.VideoEncoder.H264); // 设置视频编码方式
17.     videoPropertyBuilder.setRecorderRate(30); // 设置录制帧率
18.     source.setRecorderAudioSource(Recorder.AudioSource.MIC); // 设置录制音频源
19.     source.setRecorderVideoSource(Recorder.VideoSource.SURFACE); // 设置视频窗口
20.     mediaRecorder.setSource(source); // 设置音视频源
21.     mediaRecorder.setOutputFormat(Recorder.OutputFormat.MPEG_4); // 设置音视频输出格式
22.     StringBuffer fileName = new StringBuffer("record_"); // 生成随机文件名
```

```
23.         fileName.append(UUID.randomUUID()).append(".mp4");
24.         recordName = fileName.toString();
25.         File file = new File(dirFile, fileName.toString()); // 创建录像文件对象
26.         storagePropertyBuilder.setRecorderFile(file); // 设置存储音视频文件名
27.         mediaRecorder.setStorageProperty(storagePropertyBuilder.build());
28.         audioPropertyBuilder.setRecorderAudioEncoder(Recorder.AudioEncoder.AAC); // 设置音频编码格式
29.         mediaRecorder.setAudioProperty(audioPropertyBuilder.build()); // 设置音频属性
30.         mediaRecorder.setVideoProperty(videoPropertyBuilder.build()); // 设置视频属性
31.         mediaRecorder.prepare(); // 准备录制
32.         HiLog.info("initMediaRecorder end");
33.     }
```

2. 配置录像帧，启动录像。

```
1.  private final class CameraStateCallbackImpl extends CameraStateCallback {
2.      @Override
3.      public void onConfigured(Camera camera) {
4.          // 获取预览配置模板
5.          frameConfigBuilder = camera.getFrameConfigBuilder(FRAME_CONFIG_PREVIEW);
6.          // 配置预览 Surface
7.          frameConfigBuilder.addSurface(previewSurface);
8.          // 配置录像的 Surface
9.          mRecorderSurface = mediaRecorder.getVideoSurface();
10.         cameraConfigBuilder.addSurface(mRecorderSurface);
11.         previewFrameConfig = frameConfigBuilder.build();
12.         try {
13.             // 启动循环帧捕获
14.             int triggerId = camera.triggerLoopingCapture(previewFrameConfig);
15.         } catch (IllegalArgumentException e) {
16.             HiLog.error("Argument Exception");
17.         } catch (IllegalStateException e) {
18.             HiLog.error("State Exception");
19.         }
20.     }
21. }
```

3. 通过 camera.stopLoopingCapture()方法停止循环帧捕获（录像）。

#### 1.3.2.7 相机设备释放

使用完相机后，必须通过 release()来关闭相机和释放资源，否则可能导致其他相机应用无法启动。一旦相机被释放，它所提供的操作就不能再被调用，否则会导致不可预期的结果，或是会引发状态异常。

相机设备释放的示例代码如下：

```
1.  private void releaseCamera() {
2.      if (camera != null) {
3.          // 关闭相机和释放资源
4.          camera.release();
5.          camera = null;
6.      }
7.      // 拍照配置模板置空
8.      framePictureConfigBuilder = null;
9.      // 预览配置模板置空
10.     previewFrameConfig = null;
11. }
```

## 1.4 音频

### 1.4.1 概述

HarmonyOS 音频模块支持音频业务的开发，提供音频相关的功能，主要包括音频播放、音频采集、音量管理和短音播放等。

#### 1.4.1.1 基本概念

- **采样**

采样是指将连续时域上的模拟信号按照一定的时间间隔采样，获取到离散时域上离散信号的过程。

- **采样率**

  采样率为每秒从连续信号中提取并组成离散信号的采样次数，单位用赫兹（Hz）来表示。通常人耳能听到频率范围大约在 20Hz～20kHz 之间的声音。常用的音频采样频率有：8kHz、11.025kHz、22.05kHz、16kHz、37.8kHz、44.1kHz、48kHz、96kHz、192kHz 等。

- **声道**

  声道是指声音在录制或播放时在不同空间位置采集或回放的相互独立的音频信号，所以声道数也就是声音录制时的音源数量或回放时相应的扬声器数量。

- **音频帧**

  音频数据是流式的，本身没有明确的一帧帧的概念，在实际的应用中，为了音频算法处理/传输的方便，一般约定俗成取 2.5ms~60ms 为单位的数据量为一帧音频。这个时间被称之为“采样时间”，其长度没有特别的标准，它是根据编解码器和具体应用的需求来决定的。

- **PCM**

  PCM（Pulse Code Modulation），即脉冲编码调制，是一种将模拟信号数字化的方法，是将时间连续、取值连续的模拟信号转换成时间离散、抽样值离散的数字信号的过程。

- **短音**

  使用源于应用程序包内的资源或者是文件系统里的文件为样本，将其解码成一个 16bit 单声道或者立体声的 PCM 流并加载到内存中，这使得应用程序可以直接用压缩数据流同时摆脱 CPU 加载数据的压力和播放时重解压的延迟。

- **tone 音**

  根据特定频率生成的波形，比如拨号盘的声音。

- **系统音**

  系统预置的短音，比如按键音，删除音等。

#### 1.4.1.2 约束与限制

- 在使用完 AudioRenderer 音频播放类和 AudioCapturer 音频采集类后，需要调用 release()方法进行资源释放。
- 音频采集所使用的最终采样率与采样格式取决于输入设备，不同设备支持的格式及采样率范围不同，可以通过 AudioManager 类的 getDevices 接口查询。
- 在进行音频采集之前，需要申请麦克风权限 ohos.permission.MICROPHONE。

### 1.4.2 音频播放开发指导

#### 1.4.2.1 场景介绍

音频播放的主要工作是将音频数据转码为可听见的音频模拟信号并通过输出设备进行播放，同时对播放任务进行管理。

#### 1.4.2.2 接口说明

| 接口名 | 描述 |
| --- | --- |
| AudioRenderer(AudioRendererInfo audioRendererInfo, PlayMode pm) throws IllegalArgumentException | 构造函数，设置播放相关音频参数和播放模式，使用默认播放设备。 |
| AudioRenderer(AudioRendererInfo audioRendererInfo, PlayMode pm, AudioDeviceDescriptor outputDevice) throws IllegalArgumentException | 构造函数，设置播放相关音频参数、播放模式和播放设备。 |
| boolean start() | 播放音频流。 |
| boolean write(byte[] data, int offset, int size) | 将音频数据以 byte 流写入音频接收器以进行播放。 |
| boolean write(short[] data, int offset, int size) | 将音频数据以 short 流写入音频接收器以进行播放。 |

| 接口名 | 描述 |
| --- | --- |
| boolean write(float[] data, int offset, int size) | 将音频数据以 float 流写入音频接收器以进行播放。 |
| boolean write(java.nio.ByteBuffer data, int size) | 将音频数据以 ByteBuffer 流写入音频接收器以进行播放。 |
| boolean pause() | 暂停播放音频流。 |
| boolean stop() | 停止播放音频流。 |
| boolean release() | 释放播放资源。 |
| AudioDeviceDescriptor getCurrentDevice() | 获取当前工作的音频播放设备。 |
| boolean setPlaybackSpeed(float speed) | 设置播放速度。 |
| boolean setPlaybackSpeed(AudioRenderer.SpeedPara speedPara) | 设置播放速度与音调。 |
| boolean setVolume(ChannelVolume channelVolume) | 设置指定声道上的输出音量。 |
| boolean setVolume(float vol) | 设置所有声道上的输出音量。 |
| static int getMinBufferSize(int sampleRate, AudioStreamInfo.EncodingFormat format, AudioStreamInfo.ChannelMask channelMask) | 获取 Stream 播放模式所需的 buffer 大小。 |
| State getState() | 获取音频播放的状态。 |
| int getRendererSessionId() | 获取音频播放的 session ID。 |
| int getSampleRate() | 获取采样率。 |
| int getPosition() | 获取音频播放的帧数位置。 |
| boolean setPosition(int position) | 设置起始播放帧位置。 |
| AudioRendererInfo getRendererInfo() | 获取音频渲染信息。 |

| 接口名 | 描述 |
| --- | --- |
| boolean duckVolume() | 降低音量并将音频与另一个拥有音频焦点的应用程序混合。 |
| boolean unduckVolume() | 恢复音量。 |
| SpeedPara getPlaybackSpeed() | 获取播放速度、音调参数。 |
| boolean setSpeed(SpeedPara speedPara) | 设置播放速度、音调参数。 |
| Timestamp getAudioTime() | 获取播放时间戳信息。 |
| boolean flush() | 刷新当前的播放流数据队列。 |
| static float getMaxVolume() | 获取播放流可设置的最大音量。 |
| static float getMinVolume() | 获取播放流可设置的最小音量。 |
| StreamType getStreamType() | 获取播放流的音频流类型。 |

**表 1** 音频播放类 AudioRenderer 的主要接口

### 1.4.2.3 开发步骤

1. 构造音频流参数的数据结构 AudioStreamInfo，推荐使用 AudioStreamInfo.Builder 类来构造，模板如下，模板中设置的均为 AudioStreamInfo.Builder 类的默认值，根据音频流的具体规格来设置具体参数。

```
AudioStreamInfo audioStreamInfo = new AudioStreamInfo.Builder().sampleRate(
    AudioStreamInfo.SAMPLE_RATE_UNSPECIFIED)
    .audioStreamFlag(AudioStreamInfo.AudioStreamFlag.AUDIO_STREAM_FLAG_NONE)
    .encodingFormat(AudioStreamInfo.EncodingFormat.ENCODING_INVALID)
    .channelMask(AudioStreamInfo.ChannelMask.CHANNEL_INVALID)
    .streamUsage(AudioStreamInfo.StreamUsage.STREAM_USAGE_UNKNOWN)
    .build();
```

以真实的播放 pcm 流为例：

```
AudioStreamInfo audioStreamInfo = new AudioStreamInfo.Builder().sampleRate(44100) // 44.1kHz
    .audioStreamFlag(AudioStreamInfo.AudioStreamFlag.AUDIO_STREAM_FLAG_MAY_DUCK) // 混音
```

```
3.         .encodingFormat(AudioStreamInfo.EncodingFormat.ENCODING_PCM_16BIT) // 16-bit PCM
4.         .channelMask(AudioStreamInfo.ChannelMask.CHANNEL_OUT_STEREO) // 双声道
5.         .streamUsage(AudioStreamInfo.StreamUsage.STREAM_USAGE_MEDIA) // 媒体类音频
6.         .build();
```

2. 使用创建的音频流构建音频播放的参数结构 AudioRendererInfo，推荐使用 AudioRendererInfo.Builder 类来构造，模板如下，模板中设置的均为 AudioRendererInfo.Builder 类的默认值，根据音频播放的具体规格来设置具体参数。

```
1. AudioRendererInfo audioRendererInfo = new AudioRendererInfo.Builder().audioStreamInfo(audioStreamInfo)
2.         .audioStreamOutputFlag(AudioRendererInfo.AudioStreamOutputFlag.AUDIO_STREAM_OUTPUT_FLAG_NONE)
3.         .bufferSizeInBytes(0)
4.         .distributedDeviceId("")
5.         .isOffload(false)
6.         .sessionID(AudioRendererInfo.SESSION_ID_UNSPECIFIED)
7.         .build();
```

以真实的播放 pcm 流为例：

```
1. AudioRendererInfo audioRendererInfo = new AudioRendererInfo.Builder().audioStreamInfo(audioStreamInfo)
2.         .audioStreamOutputFlag(AudioRendererInfo.AudioStreamOutputFlag.AUDIO_STREAM_OUTPUT_FLAG_DIRECT_PCM) // pcm 格式
   的输出流
3.         .bufferSizeInBytes(100)
4.         .distributedDeviceId("E54***5E8") // 使用分布式设备 E54***5E8 播放
5.         .isOffload(false) // false 表示分段传输 buffer 并播放，true 表示整个音频流一次性传输到 HAL 层播放
6.         .build();
```

2. 根据要播放音频流指定 PlayMode，不同的 PlayMode 在写数据时存在差异，详情见步骤 7，其余播放流程是无区别的。并通过构造函数获取 AudioRenderer 类的实例化对象。
3. 使用构造函数获取 AudioRenderer 类的实例化对象，其中步骤 2，步骤 3 中的数据为构造函数的必选参数，指定播放设备为可选参数，根据使用场景选择不同的构造函数。
4. （可选）构造音频播放回调，首先构造对象 AudioInterrupt，其中 setInterruptListener 方法的入参需要实现接口类 InterruptListener，setStreamInfo 方法使用步骤 1 的 AudioStreamInfo 作为入参。然后调用 AudioManager 类的 activateAudioInterrupt(AudioInterrupt interrupt)方法进行音频播放回调注册。代码示例如下：

```
1. AudioInterrupt audioInterrupt = new AudioInterrupt();
2. AudioManager audioManager = new AudioManager();
3. audioInterrupt.setStreamInfo(streamInfo);
4. audioInterrupt.setInterruptListener(new AudioInterrupt.InterruptListener() {
5.     @Override
6.     public void onInterrupt(int type, int hint) {
7.         if (type == AudioInterrupt.INTERRUPT_TYPE_BEGIN
8.                 && hint == AudioInterrupt.INTERRUPT_HINT_PAUSE) {
9.             renderer.pause();
```

```
10.             } else if (type == AudioInterrupt.INTERRUPT_TYPE_BEGIN
11.                     && hint == AudioInterrupt.INTERRUPT_HINT_NONE) {
12.
13.             } else if (type == AudioInterrupt.INTERRUPT_TYPE_END && (
14.                     hint == AudioInterrupt.INTERRUPT_HINT_NONE
15.                             || hint == AudioInterrupt.INTERRUPT_HINT_RESUME)) {
16.                 renderer.play();
17.             } else {
18.             }
19.         }
20. });
21. audioManager.activateAudioInterrupt(audioInterrupt);
```

5. 调用 AudioRenderer 实例化对象的 start()方法启动播放任务。
6. 将要播放的音频数据读取为 byte 流或 short 流，对于选择 MODE_STREAM 模式的 PlayMode，需要循环调用 write 方法进行数据写入。对于选择 MODE_STATIC 模式的 PlayMode，只能通过调用一次 write 方法将要播放的音频数据全部写入，因此该模式限制在文件规格较小的音频数据播放场景下才能使用。
7. （可选）当需要对音频播放进行暂停或停止时，调用 AudioRenderer 实例化对象的 pause()或 stop()方法进行暂停或停止播放。
8. （可选）调用 AudioRenderer 实例化对象的 setSpeed 调节播放速度，setVolume 调节播放音量。
9. 播放任务结束后，调用 AudioRenderer 实例化对象的 release()释放资源。

### 1.4.3 音频采集开发指导

#### 1.4.3.1 场景介绍

音频采集的主要工作是通过输入设备将声音采集并转码为音频数据，同时对采集任务进行管理。

### 1.4.3.2 接口说明

| 接口名 | 描述 |
| --- | --- |
| AudioCapturer(AudioCapturerInfo audioCapturerInfo) throws IllegalArgumentException | 构造函数，设置录音相关音频参数，使用默认录音设备。 |
| AudioCapturer(AudioCapturerInfo audioCapturerInfo, AudioDeviceDescriptor devInfo) throws IllegalArgumentException | 构造函数，设置录音相关音频参数并指定录音设备。 |
| static int getMinBufferSize(int sampleRate, int channelCount, int audioFormat) | 获取指定参数条件下所需的最小缓冲区大小。 |
| boolean addSoundEffect(UUID type, String packageName) | 增加录音的音频音效。 |
| boolean start() | 开始录音。 |
| int read(byte[] data, int offset, int size) | 读取音频数据。 |
| int read(byte[] data, int offset, int size, boolean isBlocking) | 读取音频数据并写入传入的 byte 数组中。 |
| int read(float[] data, int offsetInFloats, int sizeInFloats) | 阻塞式读取音频数据并写入传入的 float 数组中。 |
| int read(float[] data, int offsetInFloats, int sizeInFloats, boolean isBlocking) | 读取音频数据并写入传入的 float 数组中。 |
| int read(short[] data, int offsetInShorts, int sizeInShorts) | 阻塞式读取音频数据并写入传入的 short 数组中。 |
| int read(short[] data, int offsetInShorts, int sizeInShorts, boolean isBlocking) | 读取音频数据并写入传入的 short 数组中。 |
| int read(java.nio.ByteBuffer buffer, int sizeInBytes) | 阻塞式读取音频数据并写入传入的 ByteBuffer 对象中。 |

| 接口名 | 描述 |
| --- | --- |
| int read(java.nio.ByteBuffer buffer, int sizeInBytes, boolean isBlocking) | 读取音频数据并写入传入的 ByteBuffer 对象中。 |
| boolean stop() | 停止录音。 |
| boolean release() | 释放录音资源。 |
| AudioDeviceDescriptor getSelectedDevice() | 获取输入设备信息。 |
| AudioDeviceDescriptor getCurrentDevice() | 获取当前正在录制音频的设备信息。 |
| int getCapturerSessionId() | 获取录音的 session ID。 |
| Set<SoundEffect> getSoundEffects() | 获取已经激活的音频音效列表。 |
| AudioCapturer.State getState() | 获取音频采集状态。 |
| int getSampleRate() | 获取采样率。 |
| int getAudioInputSource() | 获取录音的输入设备信息。 |
| int getBufferFrameCount() | 获取以帧为单位的缓冲区大小。 |
| int getChannelCount() | 获取音频采集通道数。 |
| AudioStreamInfo.EncodingFormat getEncodingFormat() | 获取音频采集的音频编码格式。 |
| boolean getAudioTime(Timestamp timestamp, Timestamp.Timebase timebase) | 获取一个即时的捕获时间戳。 |

**表 1** 音频采集类 AudioCapturer 的主要接口

### 1.4.3.3 开发步骤

1. 构造音频流参数的数据结构 AudioStreamInfo，推荐使用 AudioStreamInfo.Builder 类来构造，模板如下，模板中设置的均为 AudioStreamInfo.Builder 类的默认值，根据音频流的具体规格来设置具体参数。

```
AudioStreamInfo audioStreamInfo = new AudioStreamInfo.Builder().sampleRate(
    AudioStreamInfo.SAMPLE_RATE_UNSPECIFIED)
    .audioStreamFlag(AudioStreamInfo.AudioStreamFlag.AUDIO_STREAM_FLAG_NONE)
    .encodingFormat(AudioStreamInfo.EncodingFormat.ENCODING_INVALID)
    .channelMask(AudioStreamInfo.ChannelMask.CHANNEL_INVALID)
    .streamUsage(AudioStreamInfo.StreamUsage.STREAM_USAGE_UNKNOWN)
    .build();
```

2. （可选）通过采集的采样率、声道数和数据格式，调用 getMinBufferSize 方法获取采集任务所需的最小 buffer，参照该 buffer 值设置步骤 3 中 AudioCapturerInfo 的 bufferSizeInBytes。
3. 使用步骤 1 创建的音频流构建音频播放的参数结构 AudioCapturerInfo，推荐使用 AudioCapturerInfo.Builder 类来构造，根据音频采集的具体规格来设置具体参数。以真实的录制 pcm 流为例：

```
AudioStreamInfo audioStreamInfo = new AudioStreamInfo.Builder().encodingFormat(
    AudioStreamInfo.EncodingFormat.ENCODING_PCM_16BIT) // 16-bit PCM
    .channelMask(AudioStreamInfo.ChannelMask.CHANNEL_IN_STEREO) // 双声道
    .sampleRate(44100) // 44.1kHz
    .build();
AudioCapturerInfo audioCapturerInfo = new AudioCapturerInfo.Builder().audioStreamInfo(audioStreamInfo)
    .build();
```

4. （可选）设置采集设备，如麦克风、耳机等。通过 AudioManager.getDevices(AudioDeviceDescriptor.DeviceFlag.INPUT_DEVICES_FLAG)获取到设备支持的输入设备，然后依照 AudioDeviceDescriptor.DeviceType 选择要选用的输入设备类型。
5. 通过构造方法获取 AudioCapturer 类的实例化对象，其中步骤 3 的参数为必选参数，通过步骤 4 获取的指定录音设备为可选参数。
6. （可选）设置采集音效，如降噪、回声消除等。使用 addSoundEffect(UUID type, String packageName)进行音效设置，其中 UUID 参考类 SoundEffect 中提供的静态变量。
7. （可选）构造音频采集回调，首先继承抽象类 AudioCapturerCallback，并实现抽象方法 onCapturerConfigChanged(List<AudioCapturerConfig> configs)，然后调用 AudioManager 类的 registerAudioCapturerCallback(AudioCapturerCallback cb)方法进行音频采集回调注册。代码示例如下：

```
private AudioManager audioManager = new AudioManager();

public void main() {
    AudioCapturerCallback cb = new AudioCapturerCallback() {
        @Override
        public void onCapturerConfigChanged(List<AudioCapturerConfig> configs) {
            configs.forEach(config -> doSomething(config));
        }
    };
```

```
10.         audioManager.registerAudioCapturerCallback(cb);
11.     }
12.
13.     private void doSomething(AudioCapturerConfig config) {
14.         ...
15.     }
```

8. 调用 AudioCapturer 实例化对象的 start()方法启动采集任务。
9. 采集的音频数据读取为 byte 流，循环调用 read 方法进行数据读取。
10. 调用 AudioCapturer 实例化对象的 stop()方法停止采集。
11. 播放任务结束后，调用 AudioCapturer 实例化对象的 release()释放资源。

### 1.4.4 音量管理开发指导

#### 1.4.4.1 场景介绍

音量管理的主要工作是音量调节，输入/输出设备管理，注册音频中断、音频采集中断的回调等。

#### 1.4.4.2 接口说明

| 接口名 | 描述 |
| --- | --- |
| AudioManager() | 构造函数。 |
| AudioManager(Context context) | 构造函数，由使用者指定应用上下文 Context。 |
| AudioManager(String packageName) | 构造函数，由使用者指定包信息。 |
| activateAudioInterrupt(AudioInterrupt interrupt) | 激活音频中断状态检测。 |
| deactivateAudioInterrupt(AudioInterrupt interrupt) | 去激活音频中断状态检测。 |
| getAudioParameter(String key) | 获取音频硬件中指定参数 keys 所对应的参数值。 |

| 接口名 | 描述 |
| --- | --- |
| AudioDeviceDescriptor[] getDevices(DeviceFlag flag) | 获取设备信息。 |
| int getMaxVolume(AudioVolumeType volumeType) | 获取指定音频流音量最大档位。 |
| int getMinVolume(AudioVolumeType volumeType) | 获取指定音频流音量最小档位。 |
| int getRingerMode() | 获取铃声模式。 |
| int getVersion() | 获取音频套件版本。 |
| int getVolume(AudioVolumeType volumeType) | 获取指定音频流的音量档位。 |
| boolean isDeviceActive(int deviceType) | 判断设备的开关状态。 |
| boolean isMute(AudioVolumeType volumeType) | 特定的流是否处于静音状态。 |
| boolean mute(AudioVolumeType volumeType) | 将特定流设置为静音状态。 |
| boolean setAudioParameter(String key, String value) | 为音频硬件设置可变数量的参数值。 |
| boolean setDeviceActive(int deviceType, boolean state) | 设置设备的开关状态。 |
| boolean setRingerMode(AudioRingMode mode) | 设置铃声模式。 |
| boolean setVolume(AudioVolumeType volumeType, int volume) | 设置特定流的音量档位。 |
| boolean unmute(AudioVolumeType volumeType) | 将特定流设置为非静音状态。 |
| boolean setMasterMute(boolean isMute) | 将主音频输出设备设置为静音或取消静音状态。 |
| boolean setMicrophoneMute(boolean isMute) | 将麦克风设置为静音或取消静音状态。 |
| boolean isMicrophoneMute() | 判断麦克风是否处于静音状态。 |
| List<AudioCapturerConfig> getActiveCapturerConfigs() | 获取设备当前激活的音频采集任务的配置信息。 |

| 接口名 | 描述 |
| --- | --- |
| registerAudioCapturerCallback(AudioCapturerCallback cb) | 注册音频采集参数变更回调。 |
| void unregisterAudioCapturerCallback (AudioCapturerCallback cb) | 去注册音频采集参数变更回调。 |
| Uri getRingerUri(Context context, RingToneType type) | 获取指定铃声类型的 Uri。 |
| void setRingerUri(Context context, RingToneType type, Uri uri) | 设置指定铃声类型的 Uri。 |
| AudioManager.CommunicationState getCommunicationState() | 获取当前的通话模式。 |
| void setCommunicationState (AudioManager.CommunicationState communicationState) | 设置当前的通话模式。 |
| boolean changeVolumeBy(AudioVolumeType volumeType, int index) | 将当前音量增加或减少一定量。 |
| boolean connectBluetoothSco() | 连接到蓝牙 SCO 通道。 |
| boolean disconnectBluetoothSco() | 断开与蓝牙 SCO 通道的连接。 |
| java.util.List<AudioRendererInfo> getActiveRendererConfigs() | 获取有关活动音频流信息，包括使用类型、内容类型和标志。 |
| static int getMasterOutputFrameCount() | 获取主输出设备缓冲区中的帧数。 |
| static int getMasterOutputSampleRate() | 获取主输出设备的采样率。 |
| boolean isMasterMute() | 检查音频流是否全局静音。 |
| static boolean isStreamActive(AudioVolumeType volumeType) | 检查指定类型的音频流是否处于活动状态。 |

| 接口名 | 描述 |
| --- | --- |
| static int makeSessionId() | 创建一个会话 ID，AudioRendererInfo.Builder.sessionID(int)将使用该会话 ID 来设置音频播放参数，而 AudioCapturerInfo.Builder.sessionID(int)将使用该会话 ID 来设置记录参数。 |
| void registerAudioRendererCallback (AudioRendererCallback cb) | 注册音频播放参数变更回调。 |
| void unregisterAudioRendererCallback (AudioRendererCallback cb) | 去注册音频播放参数变更回调。 |

**表 1** 音量管理类 AudioManager 的主要接口

#### 1.4.4.3 开发步骤

音量管理提供的都是独立的功能，一般作为音频播放和音频采集的功能补充来使用。开发者根据具体使用场景选择方法即可。

音频中断状态检测和音频采集中断状态检测的使用样例，请参考音频播放和音频采集的开发步骤。

### 1.4.5 短音播放开发指导

#### 1.4.5.1 场景介绍

短音播放主要负责管理音频资源的加载与播放、tone 音的生成与播放以及系统音播放。

### 1.4.5.2 接口说明

短音播放开放能力分为音频资源、tone 音和系统音三部分，均定义在 SoundPlayer 类。

| 接口名 | 描述 |
|---|---|
| SoundPlayer(int taskType) | 构造函数，仅用于音频资源。 |
| int createSound(String path) | 从指定的路径加载音频数据生成短音资源。 |
| int createSound(Context context, int resourceId) | 根据应用程序上下文合音频资源 ID 加载音频数据生成短音资源。 |
| int createSound(AssetFD assetFD) | 从指定的 AssetFD 实例加载音频数据生成短音资源。 |
| int createSound(java.io.FileDescriptor fd, long offset, long length) | 根据文件描述符从文件加载音频数据生成音频资源。 |
| int createSound(java.lang.String path, AudioRendererInfo rendererInfo) | 根据从指定路径和播放信息加载音频数据生成短音资源。 |
| boolean setOnCreateCompleteListener (SoundPlayer.OnCreateCompleteListener listener) | 设置声音创建完成的回调。 |
| boolean setOnCreateCompleteListener (SoundPlayer.OnCreateCompleteListener listener, boolean isDiscarded) | 设置用于声音创建完成的回调，并根据指定的 isDiscarded 标志位确定是否丢弃队列中的原始回调通知消息。 |
| boolean deleteSound(int soundID) | 删除短音同时释放短音所占资源。 |
| boolean pause(int taskID) | 根据播放任务 ID 暂停对应的短音播放。 |
| int play(int soundID) | 使用默认参数播放短音。 |
| int play(int soundID, SoundPlayerParameters parameters) | 使用指定参数播放短音。 |
| boolean resume(int taskID) | 恢复短音播放任务。 |

| 接口名 | 描述 |
| --- | --- |
| boolean setLoop(int taskID, int loopNum) | 设置短音播放任务的循环次数。 |
| boolean setPlaySpeedRate(int taskID, float speedRate) | 设置短音播放任务的播放速度。 |
| boolean setPriority(int taskID, int priority) | 设置短音播放任务的优先级。 |
| boolean setVolume(int taskID, AudioVolumes audioVolumes) | 设置短音播放任务的播放音量。 |
| boolean setVolume(int taskID, float volume) | 设置短音播放任务的所有音频声道的播放音量。 |
| boolean stop(int taskID) | 停止短音播放任务。 |
| boolean pauseAll() | 暂停所有正在播放的任务。 |
| boolean resumeAll() | 恢复虽有已暂停的播放任务。 |

**表 1** 音频资源的加载与播放类 SoundPlayer 的主要接口

| 接口名 | 描述 |
| --- | --- |
| SoundPlayer() | 构造函数，仅用于 tone 音。 |
| boolean createSound(ToneDescriptor.ToneType type, int durationMs) | 创建具有音调频率描述和持续时间（毫秒）的 tone 音。 |
| boolean createSound(AudioStreamInfo.StreamType streamType, float volume) | 根据音量和音频流类型创建 tone 音。 |
| boolean play(ToneDescriptor.ToneType toneType, int durationMs) | 播放指定时长和 tone 音类型的 tone 音。 |
| boolean pause() | 暂停 tone 音播放。 |
| boolean play() | 播放创建好的 tone 音。 |
| boolean release() | 释放 tone 音资源。 |

| 接口名 | 描述 |
| --- | --- |

**表 2** tone 音的生成与播放 API 接口功能介绍

| 接口名 | 描述 |
| --- | --- |
| SoundPlayer(String packageName) | 构造函数，仅用于系统音。 |
| boolean playSound(SoundType type) | 播放系统音。 |
| boolean playSound(SoundType type, float volume) | 指定音量播放系统音。 |

**表 3** 系统音的播放 API 接口功能介绍

### 1.4.5.3 音频资源的加载与播放

1. 通过 SoundPlayer(int taskType)构造方法获取 SoundPlayer 类的实例化对象，其中入参 taskType 的取值范围和含义参考枚举类 AudioManager.AudioStreamType 的定义。
2. 调用 createSound(String path)方法从指定路径加载音频资源，并生成短音 ID，后续可使用通过短音 ID 进行短音资源的播放和删除等操作。
3. （可选）提供单独对音量，循环次数，播放速度和优先级进行的设置的方法，支持在短音播放过程中进行实时调整。
4. 短音播放提供两种方法，一种是包含播放参数设置的 play(int soundID, SoundPlayerParameters parameters)方法，用户可以在 SoundPlayerParameters 数据结构中定义音量，循环次数，播放速度和优先级，另一种是使用默认播放参数的 play(int soundID)方法。短音播放成功后返回任务 ID，供后续对任务的管理。
5. 通过任务 ID，可以对短音播放任务进行暂停，恢复和停止。
6. 短音资源使用完毕需要调用 deleteSound(int soundID)完成对资源的释放。

下面的样例展示音频资源的加载与播放：

```
1.  public void demo() {
2.      // 步骤 1：实例化对象
3.      SoundPlayer soundPlayer = new SoundPlayer(AudioManager.AudioVolumeType.STREAM_MUSIC.getValue());
4.      // 步骤 2：指定音频资源加载并创建短音
5.      int soundId = soundPlayer.createSound("/system/xxx");
6.      // 步骤 3：指定音量，循环次数和播放速度
7.      SoundPlayerParameters parameters = new SoundPlayerParameters();
8.      parameers.setVolumes(new AudioVolumes());
9.      parameters.setLoop(10);
10.     parameters.setSpeed(1.0f);
```

```
11.     // 步骤 4：短音播放
12.     soundPlayer.play(soundId, parameters);
13.     // 步骤 5：停止播放
14.     soundPlayer.stop(soundId);
15.     // 步骤 6：释放短音资源
16.     soundPlayer.deleteSound(soundId);
17. }
```

#### 1.4.5.4 tone 音的生成与播放

1. 通过 SoundPlayer()构造方法获取 SoundPlayer 类的实例化对象。
2. 使用 SoundPlayer 的实例化对象，通过 createSound(ToneDescriptor.ToneType type, int durationMs)方法，指定 tone 音类型和 tone 音播放时长来创建 tone 音资源。
3. 使用 SoundPlayer 的实例化对象，通过 play、pause、release 方法完成 tone 音播放，tone 音暂停和 tone 音资源释放。

下面的样例展示 tone 音的生成与播放：

```
1. public void demo() {
2.     // 步骤 1：实例化对象
3.     SoundPlayer soundPlayer = new SoundPlayer();
4.     // 步骤 2：创建 DTMF_0（高频 1336Hz，低频 941Hz）持续时间 1000ms 的 tone 音
5.     soundPlayer.createSound(ToneDescriptor.ToneType.DTMF_0, 1000);
6.     // 步骤 3：tone 应播放，暂停和资源释放
7.     soundPlayer.play();
8.     soundPlayer.pause();
9.     soundPlayer.release();
10. }
```

# 2 系统音的播放

1. 通过 SoundPlayer(String packageName)构造方法获取 SoundPlayer 类的实例化对象。
2. 使用 SoundPlayer 的实例化对象，通过 playSound(SoundType type)或 playSound(SoundType type, float volume)方法指定系统音类型和音量，并进行系统音播放。

下面的样例展示系统音的播放：

```
1. public void demo() {
2.     // 步骤 1：实例化对象
3.     SoundPlayer soundPlayer = new SoundPlayer("packageName");
```

```
4.         // 步骤 2：播放键盘敲击音，音量为 1.0
5.         soundPlayer.playSound(SoundType.KEY_CLICK, 1.0f);
6.     }
```

# 2.1 媒体会话管理

## 2.1.1 概述

AVSession 是一套媒体播放控制框架，对媒体服务和界面进行解耦，并提供规范的通信接口，使应用可以自由、高效地在不同的媒体之间完成切换。

### 2.1.1.1 约束与限制

- 在使用完 AVSession 类后，需要及时进行资源释放。
- 调用 AVBrowser 的 subscribeByParentMediaId(String, AVSubscriptionCallback)之前，需要先执行 unsubscribeByParentMediaId(String)，防止重复订阅。
- 使用 AVBrowserService 的方法 onLoadAVElementList(String, AVBrowserResult)的 result 返回数据前，执行 detachForRetrieveAsync()。
- 播放器类需要使用 ohos.media.player.Player，否则无法正常接收按键事件。

## 2.1.2 媒体会话开发指导

### 2.1.2.1 场景介绍

**AVSession** 框架有四个主要的类，控制着整个框架的核心，下图简单的说明四个核心媒体框架控制类的关系。

- **AVBrowser**

  媒体浏览器，通常在客户端创建，成功连接媒体服务后，通过媒体控制器 AVBrowser 向服务端发送播放控制指令。

其主要流程为，调用 connect 方法向 AVBrowserService 发起连接请求，连接成功后在回调方法 AVConnectionCallback.onConnected 中发起订阅数据请求，并在回调方法 AVSubscriptionCallback.onAVElementListLoaded 中保存请求的媒体播放数据。

- **AVController**

  媒体控制器，在客户端 AVBrowser 连接服务成功后的回调方法 AVConnectionCallback.onConnected 中创建，用于向 Service 发送播放控制指令，并通过实现 AVControllerCallback 回调来响应服务端媒体状态变化，例如曲目信息变更、播放状态变更等，从而完成 UI 刷新。

- **AVBrowserService**

  媒体浏览器服务，通常在服务端，通过媒体会话 AVSession 与媒体浏览器建立连接，并通过实现 Player 进行媒体播放。其中有两个重要的方法：

1. onGetRoot，处理从媒体浏览器 AVBrowser 发来的连接请求，通过返回一个有效的 AVBrowserRoot 对象表示连接成功；
2. onLoadAVElementList，处理从媒体浏览器 AVBrowser 发来的数据订阅请求，通过 AVBrowserResult.sendAVElementList(List<AVElement>) 方法返回媒体播放数据。

- **AVSession**

  媒体会话，通常在 AVBrowserService 的 onStart 中创建，通过 setAVToken 方法设置到 AVBrowserService 中，并通过实现 AVSessionCallback 回调来接收和处理媒体控制器 AVController 发送的播放控制指令，如播放、暂停、跳转至上一曲、跳转至下一曲等。

  除了上述四个类，AVSession 框架还有 AVElement。

- **AVElement**

  媒体元素，用于将播放列表从 AVBrowserService 传递给 AVBrowser。

### 2.1.2.2 接口说明

| 接口名 | 描述 |
| --- | --- |
| AVBrowser(Context context, ElementName name, AVConnectionCallback callback, PacMap options) | 构造 AVBrowser 实例，用于浏览 AVBrowserService 提供的媒体数据。 |
| void connect() | 连接 AVBrowserService。 |
| void disconnect() | 与 AVBrowserService 断开连接。 |
| boolean isConnected() | 判断当前是否已经与 AVBrowserService 连接。 |
| ElementName getElementName() | 获取 AVBrowserService 的 ohos.bundle.ElementName 实例。 |
| String getRootMediaId() | 获取默认媒体 id。 |
| PacMap getOptions() | 获取 AVBrowserService 提供的附加数据。 |
| AVToken getAVToken() | 获取媒体会话的令牌。 |
| void getAVElement(String mediaId, AVElementCallback callback) | 输入媒体的 id，查询对应的 ohos.media.common.sessioncore.AVElement 信息，查询结果会通过 callback 返回。 |
| void subscribeByParentMediaId(String parentMediaId, AVSubscriptionCallback callback) | 查询指定媒体 id 包含的所有媒体元素信息，并订阅它的媒体信息更新通知。 |
| void subscribeByParentMediaId(String parentMediaId, PacMap options, AVSubscriptionCallback callback) | 基于特定于服务的参数来查询指定媒体 id 中的媒体元素的信息，并订阅它的媒体信息更新通知。 |
| void unsubscribeByParentMediaId(String parentMediaId) | 取消订阅对应媒体 id 的信息更新通知。 |

| 接口名 | 描述 |
| --- | --- |
| void unsubscribeByParentMediaId(String parentMediaId, AVSubscriptionCallback callback) | 取消订阅与指定 callback 相关的媒体 id 的信息更新通知。 |

表 1 AVBrowser 的主要接口

| 接口名 | 描述 |
| --- | --- |
| abstract AVBrowserRoot onGetRoot(String callerPackageName, int clientUid, PacMap options) | 回调方法，用于返回应用程序的媒体内容的根信息，在 AVBrowser.connect() 后进行回调。 |
| abstract void onLoadAVElementList(String parentMediaId, AVBrowserResult result) | 回调方法，用于返回应用程序的媒体内容的结果信息 AVBrowserResult，其中包含了子节点的 AVElement 列表，在 AVBrowser 的方法 subscribeByParentMediaId 或 notifyAVElementListUpdated 执行后进行回调。 |
| abstract void onLoadAVElement(String mediaId, AVBrowserResult result) | 回调方法，用于获取特定的媒体项目 AVElement 结果信息，在 AVBrowser.getAVElement 方法执行后进行回调。 |
| AVToken getAVToken() | 获取 AVBrowser 与 AVBrowserService 之间的会话令牌。 |
| void setAVToken(AVToken token) | 设置 AVBrowser 与 AVBrowserService 之间的会话令牌。 |
| final PacMap getBrowserOptions() | 获取 AVBrowser 在连接 AVBrowserService 时设置的服务参数选项。 |
| final AVRemoteUserInfo getCallerUserInfo() | 获取当前发送请求的调用者信息。 |
| void notifyAVElementListUpdated(String parentMediaId) | 通知所有已连接的 AVBrowser 当前父节点的子节点已经发生改变。 |
| void notifyAVElementListUpdated(String parentId, PacMap options) | 通知所有已连接的 AVBrowser 当前父节点的子节点已经发生改变，可设置服务参数。 |

表 2 AVBrowserService 的主要接口

| 接口名 | 描述 |
| --- | --- |
| AVController(Context context, AVToken avToken) | 构造 AVController 实例，用于应用程序与 AVSession 进行交互以控制媒体播放。 |
| static boolean setControllerForAbility(Ability ability, AVController controller) | 将媒体控制器注册到 ability 以接收按键事件。 |
| boolean setAVControllerCallback(AVControllerCallback callback) | 注册一个回调以接收来自 AVSession 的变更，例如元数据和播放状态变更。 |
| boolean releaseAVControllerCallback(AVControllerCallback callback) | 释放与 AVSession 之间的回调实例。 |
| List<AVQueueElement> getAVQueueElement() | 获取播放队列。 |
| CharSequence getAVQueueTitle() | 获取播放队列的标题。 |
| AVPlaybackState getAVPlaybackState() | 获取播放状态。 |
| boolean dispatchAVKeyEvent(KeyEvent keyEvent) | 应用分发媒体按键事件给会话以控制播放。 |
| void sendCustomCommand(String command, PacMap pacMap, GeneralReceiver receiverCb) | 应用向 AVSession 发送自定义命令，参考 ohos.media.common.sessioncore.AVSessionCallback.onCommand。 |
| IntentAgent getAVSessionAbility() | 获取启动用户界面的 IntentAgent。 |
| AVToken getAVToken() | 获取应用连接到会话的令牌。此令牌用于创建媒体播放控制器。 |
| void adjustAVPlaybackVolume(int direction, int flags) | 调节播放音量。 |
| void setAVPlaybackVolume(int value, int flags) | 设置播放音量，要求支持绝对音量控制。 |
| PacMap getOptions() | 获取与此控制器连接的 AVSession 的附加数据。 |

| 接口名 | 描述 |
| --- | --- |
| long getFlags() | 获取 AVSession 的附加标识，标记在 AVSession 中的定义。 |
| AVMetadata getAVMetadata() | 获取媒体资源的元数据 ohos.media.common.AVMetadata。 |
| AVPlaybackInfo getAVPlaybackInfo() | 获取播放信息。 |
| String getSessionOwnerPackageName() | 获得 AVSession 实例的应用程序的包名称。 |
| PacMap getAVSessionInfo() | 获取会话的附加数据。 |
| PlayControls getPlayControls() | 获取一个 PlayControls 实例，将用于控制播放，比如控制媒体播放、停止、下一首等。 |

**表 3** AVController 的主要接口

| 接口名 | 描述 |
| --- | --- |
| AVSession(Context context, String tag) | 构造 AVSession 实例，用于控制媒体播放。 |
| AVSession(Context context, String tag, PacMap sessionInfo) | 构造带有附加会话信息的 AVSession 实例，用于控制媒体播放。 |
| void setAVSessionCallback(AVSessionCallback callback) | 设置回调函数来控制播放器，控制逻辑由应用实现。如果 callback 为 null 则取消控制。 |
| boolean setAVSessionAbility(IntentAgent ia) | 给 AVSession 设置一个 IntentAgent，用来启动用户界面。 |
| boolean setAVButtonReceiver(IntentAgent ia) | 为媒体按键接收器设置一个 IntentAgent，以便应用结束后，可以通过媒体按键重新拉起应用。 |
| void enableAVSessionActive(boolean active) | 设置是否激活媒体会话。当会话准备接收命令时，将输入参数设置为 true。如果会话停止接收命令，则设置为 false。 |
| boolean isAVSessionActive() | 查询会话是否激活。 |

| 接口名 | 描述 |
| --- | --- |
| void sendAVSessionEvent(String event, PacMap options) | 向所有订阅此会话的控制器发送事件。 |
| void release() | 释放资源，应用播放完之后需调用。 |
| AVToken getAVToken() | 获取应用连接到会话的令牌。此令牌用于创建媒体播放控制器。 |
| AVController getAVController() | 获取会话构造时创建的控制器，方便应用使用。 |
| void setAVPlaybackState(AVPlaybackState state) | 设置当前播放状态。 |
| void setAVMetadata(AVMetadata avMetadata) | 设置媒体资源元数据 ohos.media.common.AVMetadata。 |
| void setAVQueue(List<AVQueueElement> queue) | 设置播放队列。 |
| void setAVQueueTitle(CharSequence queueTitle) | 设置播放队列的标题，UI 会显示此标题。 |
| void setOptions(PacMap options) | 设置此会话关联的附加数据。 |
| AVCallerUserInfo getCurrentControllerInfo() | 获取发送当前请求的媒体控制器信息。 |

**表 4** AVSession 的主要接口

| 接口名 | 描述 |
| --- | --- |
| AVElement(AVDescription description, int flags) | 构造 AVElement 实例。 |
| int getFlags() | 获取 flags 的值。 |
| boolean isScannable() | 判断媒体是否可扫描，如：媒体有子节点，则可继续扫描获取子节点内容。 |
| boolean isPlayable() | 检查媒体是否可播放。 |
| AVDescription getAVDescription() | 获取媒体的详细信息。 |
| String getMediaId() | 获取媒体的 id。 |

| 接口名 | 描述 |
| --- | --- |
| **表 5** AVElement 的主要接口 | |

### 2.1.2.3 开发步骤

使用 AVSession 媒体框架创建一个播放器示例，分为创建客户端和创建服务端。

**创建客户端**

在客户端 AVClientAbility 中声明 AVBrowser 和 AVController，并向服务端发送连接请求。

```
1.  public class AVClientAbility extends Ability {
2.      // 媒体浏览器
3.      private AVBrowser avBrowser;
4.      // 媒体控制器
5.      private AVController avController;
6.      @Override
7.      public void onStart(Intent intent) {
8.          super.onStart(intent);
9.          // 用于指向媒体浏览器服务的包路径和类名
10.         ElementName elementName = new ElementName("", "com.huawei.samples.audioplayer",
"com.huawei.samples.audioplayer.AVService");
11.         // connectionCallback 在调用 avBrowser.connect 方法后进行回调。
12.         avBrowser = new AVBrowser(context, elementName, connectionCallback, null);
13.         // avBrowser 发送对媒体浏览器服务的连接请求。
14.         avBrowser.connect();
15.         // 将媒体控制器注册到 ability 以接收按键事件。
16.         AVController.setControllerForAbility(this, avController);
17.     }
18. }
```

AVConnectionCallback 回调接口中的方法为可选实现，通常需要会在 onConnected 中订阅媒体数据和创建媒体控制器 AVController。

```
1.  // 发起连接（avBrowser.connect）后的回调方法实现
2.  private AVConnectionCallback connectionCallback = new AVConnectionCallback() {
3.      @Override
4.      public void onConnected() {
5.          // 成功连接媒体浏览器服务时回调该方法，否则回调 onConnectionFailed()。
```

```
6.            // 重复订阅会报错，所以先解除订阅。
7.            avBrowser.unsubscribeByParentMediaId(avBrowser.getRootMediaId());
8.            // 第二个参数 AVSubscriptionCallback，用于处理订阅信息的回调。
9.            avBrowser.subscribeByParentMediaId(AV_ROOTavBrowser.getRootMediaId(), avSubscriptionCallback);
10.           AVToken token = avBrowser.getAVToken();
11.           avController = new AVController(AVClient.this, token); // AVController 第一个参数为当前类的 context
12.           // 参数 AVControllerCallback，用于处理服务端播放状态及信息变化时回调。
13.           avController.setAVControllerCallback(avControllerCallback);
14.           // ...
15.       }
16.       // 其它回调方法（可选）
17.       // ...
18.   };
```

通常在订阅成功时，在 AVSubscriptionCallback 回调接口 onAVElementListLoaded 中保存服务端回传的媒体列表。

```
1.    // 发起订阅信息(avBrowser.subscribeByParentMediaId)后的回调方法实现
2.    private AVSubscriptionCallback avSubscriptionCallback = new AVSubscriptionCallback() {
3.        @Override
4.        public void onAVElementListLoaded(String parentId, List<AVElement> children) {
5.            // 订阅成功时回调该方法，parentID 为标识，children 为服务端回传的媒体列表
6.            super.onAVElementListLoaded(parentId, children);
7.            list.addAll(children);
8.            // ...
9.        }
10.   };
```

AVControllerCallback 回调接口中的方法均为可选方法，主要用于服务端播放状态及信息的变化后对客户端的回调，客户端可在这些方法中实现 UI 的刷新。

```
1.    // 服务对客户端的媒体数据或播放状态变更后的回调
2.    private AVControllerCallback avControllerCallback = new AVControllerCallback() {
3.        @Override
4.        public void onAVMetadataChanged(AVMetadata metadata) {
5.            // 当服务端调用 avSession.setAVMetadata(avMetadata)时，此方法会被回调。
6.            super.onAVMetadataChanged(metadata);
7.            AVDescription description = metadata.getAVDescription();
8.            String title = description.getTitle().toString();
9.            PixelMap pixelMap = description.getIcon();
10.           // ...
```

```
11.         }
12.         @Override
13.         public void onAVPlaybackStateChanged(AVPlaybackState playbackState) {
14.             // 当服务端调用 avSession.setAVPlaybackState(...)时，此方法会被回调。
15.             super.onAVPlaybackStateChanged(playbackState);
16.             long position = playbackState.getCurrentPosition();
17.             // ...
18.         }
19.         // 其它回调方法（可选）
20.         // ...
21.     };
```

完成以上实现后，则应用可以在 UI 事件中调用 avController 的方法向服务端发送播放控制指令。

```
1.  // 在 UI 播放与暂停按钮的点击事件中向服务端发送播放或暂停指令
2.  public void toPlayOrPause() {
3.      switch (avController.getAVPlaybackState().getAVPlaybackState()) {
4.          case AVPlaybackState.PLAYBACK_STATE_NONE: {
5.              avController.getPlayControls().prepareToPlay();
6.              avController.getPlayControls().play();
7.              break;
8.          }
9.          case AVPlaybackState.PLAYBACK_STATE_PLAYING: {
10.             avController.getPlayControls().pause();
11.             break;
12.         }
13.         case AVPlaybackState.PLAYBACK_STATE_PAUSED: {
14.             avController.getPlayControls().play();
15.             break;
16.         }
17.         default: {
18.             // ...
19.         }
20.     }
21. }
```

其它播放控制根据业务是否需要实现，比如：

```
1.  avController.getPlayControls().playNext();
2.  avController.getPlayControls().playPrevious();
3.  avController.getPlayControls().playFastForward();
4.  avController.getPlayControls().rewind();
5.  avController.getPlayControls().seekTo(1000);
6.  avController.getPlayControls().stop();
7.  // ...
```

也可以主动获取媒体信息、播放状态等数据：

```
1.  AVMetadata avMetadata = avController.getAVMetadata();
2.  AVPlaybackState avPlaybackState = avController.getAVPlaybackState();
3.  // ...
```

**创建服务端**

在服务端 AVService 中声明 AVSession 和 Player。

```
1.  public class AVService extends AVBrowserService {
2.      // 媒体会话
3.      private AVSession avSession;
4.      // 媒体播放器
5.      private Player player;
6.
7.      @Override
8.      public void onStart(Intent intent) {
9.          super.onStart(intent);
10.         avSession = new AVSession(this, "AVService");
11.         setAVToken(avSession.getAVToken());
12.         // 设置 sessioncallback，用于响应客户端的媒体控制器发起的播放控制指令。
13.         avSession.setAVSessionCallback(avSessionCallback);
14.         // 设置播放状态初始状态为 AVPlaybackState.PLAYBACK_STATE_NONE。
15.         AVPlaybackState playbackState = new
    AVPlaybackState.Builder().setAVPlaybackState(AVPlaybackState.PLAYBACK_STATE_NONE, 0, 1.0f).build();
16.         avSession.setAVPlaybackState(playbackState);
17.         // 完成播放器的初始化，如果使用多个 Player，也可以在执行播放时初始化。
18.         player = new Player(this);
```

```
19.         }
20.         @Override
21.         public AVBrowserRoot onGetRoot(String clientPackageName, int clientUid, PacMap rootHints) {
22.             // 响应客户端 avBrowser.connect()方法。若同意连接，则返回有效的 AVBrowserRoot 实例，否则返回 null
23.             return new AVBrowserRoot(AV_ROOT, null);
24.         }
25.         @Override
26.         public void onLoadAVElementList(String parentId, AVBrowserResult result) {
27.              LogUtil.info(TAG, "onLoadChildren");
28.              // 响应客户端 avBrowser.subscribeByParentMediaId(...)方法。
29.              // 先执行该方法 detachForRetrieveAsync()
30.              result.detachForRetrieveAsync();
31.              // externalAudioItems 缓存媒体文件，请开发者自行实现。
32.              result.sendAVElementList(externalAudioItems.getAudioItems());
33.         }
34.         @Override
35.         public void onLoadAVElementList(String s, AVBrowserResult avBrowserResult, PacMap pacMap) {
36.              // 响应客户端 avBrowser.subscribeByParentMediaId(String, PacMap, AVSubscriptionCallback)方法。
37.         }
38.         @Override
39.         public void onLoadAVElement(String s, AVBrowserResult avBrowserResult) {
40.             // 响应客户端 avBrowser.getAVElement(String, AVElementCallback)方法。
41.         }
42.     }
```

响应客户端的媒体控制器发起的播放控制指令的回调实现。

```
1.  private AVSessionCallback avSessionCallback = new AVSessionCallback() {
2.      @Override
3.      public void onPlay() {
4.          super.onPlay();
5.          // 当客户端调用 avController.getPlayControls().play()时，该方法会被回调。
6.          // 响应播放请求，开始播放。
7.          if (avSession.getAVController().getAVPlaybackState().getAVPlaybackState() == AVPlaybackState.PLAYBACK_STATE_PAUSED) {
8.              if (player.play()) {
9.                  AVPlaybackState playbackState = new AVPlaybackState.Builder().setAVPlaybackState(
10.                     AVPlaybackState.PLAYBACK_STATE_PLAYING, player.getCurrentTime(),
11.                     player.getPlaybackSpeed()).build();
12.                 avSession.setAVPlaybackState(playbackState);
13.             }
```

```
14.             }
15.         }
16.         @Override
17.         public void onPause() {
18.             super.onPause();
19.             // 当客户端调用 avController.getPlayControls().pause()时，该方法会被回调。
20.             // 响应暂停请求，暂停播放。
21.         }
22.         @Override
23.         public void onPlayNext() {
24.             super.onPlayNext();
25.             // 当客户端调用 avController.getPlayControls().playNext()时，该方法会被回调。
26.             // 响应播放下一曲请求，通过 avSession.setAVMetadata 设置下一曲曲目的信息。
27.             avSession.setAVMetadata(avNextMetadata);
28.         }
29.         // 重写以处理按键事件
30.         @Override
31.         public boolean onMediaButtonEvent(Intent mediaButtonIntent) {
32.             KeyEvent ke = mediaButtonIntent.getParcelableParam(AVSession.PARAM_KEY_EVENT);
33.             if (ke == null) {
34.                 LogUtil.error("onMediaButtonEvent", "getParcelableParam failed");
35.                 return false;
36.             }
37.             if (ke.isKeyDown()) {
38.                 // 只处理按键抬起事件
39.                 return true;
40.             }
41.
42.             switch (ke.getKeyCode()) {
43.                 case KeyEvent.KEY_MEDIA_PLAY_PAUSE: {
44.                     if (playbackState.getAVPlaybackState() == AVPlaybackState.PLAYBACK_STATE_PAUSED) {
45.                         onPlay();
46.                         break;
47.                     }
48.                     if (playbackState.getAVPlaybackState() == AVPlaybackState.PLAYBACK_STATE_PLAYING) {
49.                         onPause();
50.                         break;
51.                     }
```

```
52.                     break;
53.                 }
54.                 case KeyEvent.KEY_MEDIA_PLAY: {
55.                     onPlay();
56.                     break;
57.                 }
58.                 case KeyEvent.KEY_MEDIA_PAUSE: {
59.                     onPause();
60.                     break;
61.                 }
62.                 case KeyEvent.KEY_MEDIA_STOP: {
63.                     onStop();
64.                     break;
65.                 }
66.                 case KeyEvent.KEY_MEDIA_NEXT: {
67.                     onPlayNext();
68.                     break;
69.                 }
70.                 case KeyEvent.KEY_MEDIA_PREVIOUS: {
71.                     onPlayPrevious();
72.                     break;
73.                 }
74.                 default: {
75.                     break;
76.                 }
77.             }
78.             return true;
79.         }
80.         // 其它回调方法（可选）
81.         // ...
82.     }
```

## 2.2 媒体数据管理

### 2.2.1 概述

HarmonyOS 媒体数据管理模块支持多媒体数据管理相关的功能开发，常见操作如：获取媒体元数据、截取帧数据等。

在进行应用的开发前，开发者应了解以下基本概念：

- **PixelMap**

  PixelMap 是图像解码后无压缩的位图格式，用于图像显示或者进一步的处理。

- **媒体元数据**

  媒体元数据是用来描述多媒体数据的数据，例如媒体标题、媒体时长等数据信息。

#### 2.2.1.1 约束与限制

为及时释放 native 资源，建议在媒体数据管理 AVMetadataHelper 对象使用完成后，主动调用 release()方法。

### 2.2.2 媒体元数据获取开发指导

#### 2.2.2.1 场景介绍

媒体元数据是描述多媒体数据的数据，例如媒体标题、媒体时长、媒体的帧数据等。

### 2.2.2.2 接口说明

| 接口名 | 描述 |
| --- | --- |
| setSource(String path) | 读取指定路径的媒体文件，将其设置为媒体源。 |
| setSource(FileDescriptor fd) | 读取指定的媒体文件描述符，设置媒体源。 |
| setSource(FileDescriptor fd, long offset, long length) | 读取指定的媒体文件描述符，读取数据的起始位置的偏移量以及读取的数据长度，设置媒体源。 |
| setSource(String uri, Map<String, String> headers) | 读取指定的媒体文件 Uri，设置媒体源。 |
| setSource(Context context, Uri uri) | 读取指定的媒体的 Uri 和上下文，设置媒体源。 |
| resolveMetadata(int keyCode) | 获取媒体元数据中指定 keyCode 对应的值。 |
| fetchVideoScaledPixelMapByTime(long timeUs, int option, int dstWidth, int dstHeight) | 根据视频源中时间戳、获取选项以及图像帧缩放大小，获取帧数据。 |
| fetchVideoPixelMapByTime(long timeUs, int option) | 根据视频源中时间戳和获取选项，获取帧数据。 |
| fetchVideoPixelMapByTime(long timeUs) | 根据视频源中时间戳，获取最靠近时间戳的帧的数据。 |
| fetchVideoPixelMapByTime() | 随机获取数据源中某一帧的数据。 |
| resolveImage() | 获取音频源中包含的图像数据，比如专辑封面，如果有多个图像，返回任意一个图像的数据。 |
| fetchVideoPixelMapByIndex(int frameIndex, PixelMapConfigs configs) | 根据指定的图像像素格式选项，获取视频源中指定一帧的数据。 |
| fetchVideoPixelMapByIndex(int frameIndex) | 获取视频源中指定一帧的数据。 |
| fetchVideoPixelMapByIndex(int frameIndex, int numFrames, PixelMapConfigs configs) | 根据指定的图像像素格式选项，获取视频源中指定的连续多帧的数据。 |

| 接口名 | 描述 |
| --- | --- |
| fetchVideoPixelMapByIndex(int frameIndex, int numFrames) | 获取视频源中指定的连续多帧的数据。 |
| fetchImagePixelMapByIndex(int imageIndex, PixelMapConfigs configs) | 根据指定的图像像素格式选项，获取源图像中指定的图像。 |
| fetchImagePixelMapByIndex(int imageIndex) | 获取源图像中指定的图像。 |
| fetchImagePrimaryPixelMap(PixelMapConfigs configs) | 据指定的图像像素格式选项，获取源图像中默认图像。 |
| fetchImagePrimaryPixelMap() | 获取源图像中默认图像。 |
| release() | 释放读取的媒体资源。 |

**表 1** 媒体元数据获取相关类 AVMetadataHelper 的主要接口

### 2.2.2.3 获取帧数据的开发步骤

1. 创建媒体数据管理 AVMetadataHelper 对象，可以通过 setSource 设置要读取的媒体文件，如果不设置或设置不正确，则无法进行后续操作。

```
1. AVMetadataHelper avMetadataHelper = new AVMetadataHelper ();
2. avMetadataHelper.setSource("/path/short_video.mp4");
```

2. 指定获取帧数据的选项，以及获取帧的时间，获取媒体源的帧数据。

```
1. PixelMap pixelMap = avMetadataHelper.fetchVideoPixelMapByTime(1000L, 0x00);
```

3. 获取到 PixelMap 对象后，调用 release()函数释放读取的媒体资源。

```
1. avMetadataHelper.release();
```

### 2.2.2.4 获取媒体元数据的开发步骤

1. 创建媒体数据管理 AVMetadataHelper 对象，可以通过 setSource 设置要读取的媒体文件，如果不设置或设置不正确，则无法进行后续操作。

```
1. AVMetadataHelper avMetadataHelper= new AVMetadataHelper();
2. avMetadataHelper.setSource("/path/short_video.mp4");
```

2. 指定要获取的媒体元数据的 key，获取媒体元数据。如下代码获取媒体的时长信息：

```
1. String result = avMetadataHelper.resolveMetadata(AVMetadataHelper.AV_KEY_DURATION);
```

3. 获取到媒体元数据后，调用 release()函数释放读取的媒体资源。

```
1. avMetadataHelper.release();
```

#### 2.2.2.5 获取音频的图像数据的开发步骤

1. 创建媒体数据管理 AVMetadataHelper 对象，可以通过 setSource 设置要读取的音频媒体文件，如果不设置或设置不正确，则无法进行后续操作。

```
1. AVMetadataHelper avMetadataHelper= new AVMetadataHelper();
2. avMetadataHelper.setSource("/path/short_video.mp4");
```

2. 获取音频的图像数据。

```
1. byte[] data = avMetadataHelper.resolveImage();
```

3. 获取到图像数据后，调用 release()函数释放读取的媒体资源。

```
1. avMetadataHelper.release();
```

### 2.2.3 媒体存储数据操作开发指导

#### 2.2.3.1 场景介绍

媒体存储是提供了操作媒体图片、视频、音频等元数据的 Uri 链接信息。

#### 2.2.3.2 接口说明

| 接口名 | 描述 |
| --- | --- |
| appendPendingResource(Uri uri) | 更新给定的 Uri，用于处理包含待处理标记的媒体项。 |
| appendRequireOriginalResource(Uri uri) | 更新给定的 Uri， 用于调用者获取原始文件内容。 |

| 接口名 | 描述 |
| --- | --- |
| fetchVolumeName(Uri uri) | 获取给定 Uri 所属的卷名。 |
| fetchExternalVolumeNames(Context context) | 获取所有组成 external 的特定卷名的列表。 |
| fetchMediaResource(Context context, Uri documentUri) | 根据文档式的 Uri 获取对应的媒体式的 Uri。 |
| fetchDocumentResource(Context context, Uri mediaUri) | 根据媒体式的 Uri 获取对应的文档式的 Uri。 |
| fetchVersion(Context context) | 获取卷名为 external_primary 的不透明版本信息。 |
| fetchVersion(Context context, String volumeName) | 获取指定卷名的不透明版本信息。 |
| fetchLoggerResource() | 获取用于查询媒体扫描状态的 Uri。 |
| Audio.convertNameToKey(String name) | 将艺术家或者专辑名称转换为可用于分组，排序和搜索的“key”。 |
| Audio.Media.fetchResource(String volumeName) | 获取用于处理音频媒体信息的 Uri。 |
| Audio.Genres.fetchResource(String volumeName) | 获取用于处理音频流派信息的 Uri。 |
| Audio.Genres.fetchResourceForAudioId(String volumeName, int audioId) | 获取用户处理音频文件对应的流派信息的 Uri。 |
| Audio.Genres.Members.fetchResource(String volumeName, long genreId) | 获取用于处理音频流派子目录的成员信息的 Uri。 |
| Audio.Playlists.fetchResource(String volumeName) | 获取用于处理音频播放列表信息的 Uri。 |
| Audio.Playlists.Members.fetchResource(String volumeName, long playlistId) | 获取用于处理音频播放列表子目录的成员信息的 Uri。 |
| Audio.Playlists.Members.updatePlaylistItem(DataAbilityHelper dataAbilityHelper, long playlistId, int oldLocation, int newLocation) | 移动播放列表到新位置。 |
| Audio.Albums.fetchResource(String volumeName) | 获取用于处理音频专辑信息的 Uri。 |

| 接口名 | 描述 |
| --- | --- |
| Audio.Artists.fetchResource(String volumeName) | 获取用于处理音频艺术家信息的 Uri。 |
| Audio.Artists.Albums.fetchResource(String volumeName, long id) | 获取用于处理所有专辑出现艺术家的歌曲信息的 Uri。 |
| Audio.Downloads.fetchResource(String volumeName) | 获取用于处理下载条目信息的 Uri。 |
| Audio.Files.fetchResource(String volumeName) | 获取用于处理媒体文件及非媒体文件（文本，HTML，PDF 等）的 Uri。 |
| Audio.Images.Media.fetchResource(String volumeName) | 获取用于处理图像媒体信息的 Uri。 |
| Audio.Video.Media.fetchResource(String volumeName) | 获取用于处理视频媒体信息的 Uri。 |

**表 1** 媒体存储相关类 AVStorage 的主要接口

### 2.2.3.3 开发步骤

以播放视频文件为例：

1. 获取媒体外部存储提供的 Uri 链接。

```
1. AVStorage.Video.Media.EXTERNAL_DATA_ABILITY_URI
```

2. 根据媒体存储提供的 Uri 链接操作媒体元数据。

```
1. DataAbilityHelper helper = getDataAbilityHelper(context);
2. try {
3.     DataAbilityPredicates predicates = new DataAbilityPredicates();
4.     // 设置查询过滤条件
5.     predicates.equalTo(AVStorage.Video.Media.DATA, "xxxxx");
6.      // columns 为 null，查询记录所有字段，当前例子表示查询 id 字段
7.     ResultSet result = helper.query(AVStorage.Video.Media.EXTERNAL_DATA_ABILITY_URI, new String[]{AVStorage.Video.Media.ID}, 
predicates);
8.     if (result == null) {
9.         return;
10.    }
11.    while (result.goToNextRow()) {
```

```
12.             result.getInt(result.getColumnIndexForName(AVStorage.Video.Media.ID)); // 获取 id 字段的值
13.         }
14.         result.close();
15.     } catch (DataAbilityRemoteException e) {
16.         // ...
17.     }
```

3. 获取到媒体 ID 后，即可通过设置媒体源来进行业务操作，如：播放。

```
1.  Uri uri = Uri.appendEncodedPathToUri(AVStore.Video.Media.EXTERNAL_DATA_ABILITY_URI, String.valueOf(id)); // id 为步骤 2 获取到的
    id
2.  Player player = new Player(context);
3.  DataAbilityHelper helper = getDataAbilityHelper(context);
4.  player.setSource(new Source(helper.openFile(uri, "r")));
```

### 2.2.4 媒体扫描服务操作开发指导

#### 2.2.4.1 场景介绍

媒体扫描服务从新创建或下载的媒体文件中读取元数据，并将文件添加到媒体数据库中。

#### 2.2.4.2 接口说明

| 接口名 | 描述 |
| --- | --- |
| performLoggerFile(String path, String mimeType) | 请求通过文件的路径和类型扫描一个媒体文件。 |
| performLoggerFile(Context context, String[] paths, String[] mimeTypes, AVLogCompletedListener callback) | 一次扫描多个媒体文件。 |
| connect() | 连接到扫描服务。 |
| disconnect() | 从扫描服务断开连接。 |
| isConnected() | 检查扫描服务是否已连接。 |

| 接口名 | 描述 |
| --- | --- |
| **表 1** 媒体扫描服务相关类 AVLoggerConnection 的主要接口 | |

### 2.2.4.3 开发步骤

媒体扫描服务分为动态调用和静态调用，以扫描文件为例：

**动态调用**

1. 初始化 AVLoggerConnection，并注册回调函数。

```
1.  public class ZSacnnerTest implements AVLoggerScannerConnectionClient {
2.      private AVLoggerConnection zScanConn;
3.      public ZSacnnerInterTest(Context context) {
4.          // 实例化
5.          zScanConn = new AVLoggerConnection(context, this);
6.      }
7.      // ...
8.  }
```

2. 连接媒体扫描服务。

```
1.  zScanConn.onConnect(); // 连接扫描服务
```

3. 在 onLoggerConnected 回调函数中执行扫描。

```
1.  @Override
2.  public void onLoggerConnected() {
3.      zScanConn.performLoggerFile(filePaths[i], mimeTypes[i]); // 服务回调执行扫描
4.  }
```

4. 在 onLogCompleted 回调函数中通知扫描结果。

```
1.  @Override
2.  public void onLogCompleted(String path, String uri) {
3.       // 回调函数返回 URI 的值
4.       zScanConn.disConnect(); // 断开扫描服务
5.  }
```

**静态调用**

1. AVLoggerConnection 静态方法 performLoggerFile，扫描结果在 onLogCompleted 中通知。

```
1.  AVLoggerConnection.performLoggerFile(this, filePaths, null, new AVScanCompletedListener(){
2.      @Override
3.      public void onLogCompleted(String path, String uri) {
4.  
5.      }
6.  });
```

### 2.2.5 视频与图像缩略图获取开发指导

#### 2.2.5.1 场景介绍

用于应用获取视频文件或图像文件的缩略图。

#### 2.2.5.2 接口说明

| 接口名 | 描述 |
| --- | --- |
| createVideoThumbnail | 创建指定视频中代表性关键帧的缩略图。 |
| createImageThumbnail | 创建指定图像的缩略图。 |

**表 1** 视频与图像缩略图获取相关类 AVThumbnailUtils 的主要接口

#### 2.2.5.3 开发步骤

获取视频文件的缩略图。

```
1.  PixelMap resMap = AVThumbnailUtils.createVideoThumbnail(videoFile, size);
```

获取图片文件的缩略图。

```
1.  PixelMap resMap = AVThumbnailUtils.createImageThumbnail(imageFile, size);
```

# 3 安全

# 3.1 权限

## 3.1.1 概述

### 3.1.1.1 基本概念

- **应用沙盒**

  系统利用内核保护机制来识别和隔离应用资源，可将不同的应用隔离开，保护应用自身和系统免受恶意应用的攻击。默认情况下，应用间不能彼此交互，而且对系统的访问会受到限制。例如，如果应用 A（一个单独的应用）尝试在没有权限的情况下读取应用 B 的数据或者调用系统的能力拨打电话，操作系统会阻止此类行为，因为应用 A 没有被授予相应的权限。

- **应用权限**

  由于系统通过沙盒机制管理各个应用，在默认规则下，应用只能访问有限的系统资源。但应用为了扩展功能的需要，需要访问自身沙盒之外的系统或其他应用的数据（包括用户个人数据）或能力；系统或应用也必须以明确的方式对外提供接口来共享其数据或能力。为了保证这些数据或能力不会被不当或恶意使用，就需要有一种访问控制机制来保护，这就是应用权限。

  应用权限是程序访问操作某种对象的许可。权限在应用层面要求明确定义且经用户授权，以便系统化地规范各类应用程序的行为准则与权限许可。

- **权限保护的对象**

  权限保护的对象可以分为数据和能力。数据包含了个人数据（如照片、通讯录、日历、位置等）、设备数据（如设备标识、相机、麦克风等）、应用数据；能力包括了设备能力（如打电话、发短信、联网等）、应用能力（如弹出悬浮框、创建快捷方式等）等。

- **权限开放范围**

权限开放范围指一个权限能被哪些应用申请。按可信程度从高到低的顺序，不同权限开放范围对应的应用可分为：系统服务、系统应用、系统预置特权应用、同签名应用、系统预置普通应用、持有权限证书的后装应用、其他普通应用，开放范围依次扩大。

- **敏感权限**

  涉及访问个人数据（如：照片、通讯录、日历、本机号码、短信等）和操作敏感能力（如：相机、麦克风、拨打电话、发送短信等）的权限。

- **应用核心功能**

  一个应用可能提供了多种功能，其中应用为满足用户的关键需求而提供的功能，称为应用的核心功能。这是一个相对宽泛的概念，本规范用来辅助描述用户权限授权的预期。用户选择安装一个应用，通常是被应用的核心功能所吸引。比如导航类应用，定位导航就是这种应用的核心功能；比如媒体类应用，播放以及媒体资源管理就是核心功能，这些功能所需要的权限，用户在安装时内心已经倾向于授予（否则就不会去安装）。与核心功能相对应的是辅助功能，这些功能所需要的权限，需要向用户清晰说明目的、场景等信息，由用户授权。既不属于核心功能，也不是支撑核心功能的辅助功能，就是多余功能。不少应用存在并非为用户服务的功能，这些功能所需要的权限通常被用户禁止。

- **最小必要权限**

  保障应用某一服务类型正常运行所需要的应用权限的最小集，一旦缺少将导致该类型服务无法实现或无法正常运行的应用权限。

#### 3.1.1.2 运作机制

系统所有应用均在应用沙盒内运行。默认情况下，应用只能访问有限的系统资源。这些限制是通过 DAC（Discretionary Access Control）、MAC（Mandatory Access Control）以及本文描述的应用权限机制等多种不同的形式实现的。因应用需要实现其某些功能而必须访问系统

或其他应用的数据或操作某些器件，此时就需要系统或其他应用能提供接口，考虑到安全，就需要对这些接口采用一种限制措施，这就是称为“应用权限”的安全机制。

接口的提供涉及到其权限的命名和分组、对外开放的范围、被授予的应用、以及用户的参与和体验。应用权限管理模块的目的就是负责管理由接口提供方（访问客体）、接口使用方（访问主体）、系统（包括云侧和端侧）和用户等共同参与的整个流程，保证受限接口是在约定好的规则下被正常使用，避免接口被滥用而导致用户、应用和设备受损。

#### 3.1.1.3 约束与限制

- 同一应用自定义权限个数不能超过 1024 个。
- 同一应用申请权限个数不能超过 1024 个。
- 为避免与系统权限名冲突，应用自定义权限名不能以 ohos 开头，且权限名长度不能超过 256 字符。
- 自定义权限授予方式不能为 user_grant。
- 自定义权限开放范围不能为 restricted。

### 3.1.2 开发指导

#### 3.1.2.1 场景介绍

HarmonyOS 支持开发者自定义权限来保护能力或接口，同时开发者也可申请权限来访问受权限保护的对象。

#### 3.1.2.2 权限申请

开发者需要在 config.json 文件中的“reqPermissions”字段中声明所需要的权限。

```
1. {
2.     "reqPermissions": [{
3.         "name": "ohos.permission.CAMERA",
```

```
4.              "reason": "$string:permreason_camera",
5.              "usedScene": {
6.                  "ability": ["com.mycamera.Ability", "com.mycamera.AbilityBackground"],
7.                  "when": "always"
8.              }
9.          },{
10.         ...
11.         }]
12.     }
```

权限申请格式采用数组格式，可支持同时申请多个权限，权限个数最多不能超过 1024 个。

| 键 | 值说明 | 类型 | 取值范围 | 默认值 | 规则约束 |
| --- | --- | --- | --- | --- | --- |
| name | 必须，填写需要使用的权限名称。 | 字符串 | 自定义 | 无 | 未填写时，解析失败。 |
| reason | 可选，当申请的权限为 user_grant 权限时此字段必填。<br>描述申请权限的原因。 | 字符串 | 显示文字长度不能超过 256 个字节。 | 空 | user_grant 权限必填，否则不允许在应用市场上架。<br>需做多语种适配。 |
| usedScene | 可选，当申请的权限为 user_grant 权限时此字段必填。<br>描述权限使用的场景和时机。<br>场景类型有：ability、when（调用时机）。可配置多个 ability。 | ability：字符串数组<br>when：字符串 | ability：ability 的名称<br>when：inuse（使用时）、always（始终） | ability：空<br>when：inuse | user_grant 权限必填 ability，可选填 when。 |

**表 1** reqPermissions 权限申请字段说明

如果声明使用的权限的 grantMode 是 system_grant，则权限会在当应用安装的时候被自动授予。

如果声明使用的权限的 grantMode 是 user_grant，则必须经用户手动授权（用户在弹框中授权或进入权限设置界面授权）才可使用。用户会看到 reason 字段中填写的理由，来帮助用户决定是否给予授权。

**说明**

对于授权方式为 user_grant 的权限，每一次执行需要这一权限的操作时，都需要检查自身是否有该权限。当自身具有权限时，才可继续执行，否则应用需要请求用户授予权限。示例参见动态申请权限开发步骤。

### 3.1.2.3 自定义权限

开发者需要在 config.json 文件中的“defPermissions”字段中自定义所需的权限：

```
1. {
2.     "defPermissions": [{
3.         "name": "com.myability.permission.MYPERMISSION",
4.         "grantMode": "system_grant",
5.         "availableScope": ["signature"]
6.     }, {
7.     ...
8.     }]
9. }
```

权限定义格式采用数组格式，可支持同时定义多个权限，自定义的权限个数最多不能超过 1024 个。权限定义的字段描述详见表 2。

| 键 | 值说明 | 类型 | 取值范围 | 默认值 | 规则约束 |
| --- | --- | --- | --- | --- | --- |
| name | 必填，权限名称。为最大可能避免重名，采用反向域公司名+应用名+权限名组合。 | 字符串 | 自定义 | 无 | 第三方应用不允许填写系统存在的权限，否则安装失败。未填写解析失败。权限名长度不能超过 256 个字符。 |

| 键 | 值说明 | 类型 | 取值范围 | 默认值 | 规则约束 |
| --- | --- | --- | --- | --- | --- |
| grantMode | 必填，权限授予方式。 | 字符串 | user_grant（用户授权）<br>system_grant（系统授权）<br>取值含义参见：表 3。 | system_grant | 未填值或填写了取值范围以外的值时，自动赋予默认值；<br>不允许第三方应用填写 user_grant，填写后会自动赋予默认值。 |
| availableScope | 选填，权限限制范围。不填则表示此权限对所有应用开放。 | 字符串数组 | signature<br>privileged<br>restricted<br>可参见。取值含义请见：表 4。 | 空 | 填写取值范围以外的值时，权限限制范围不生效。<br>由于第三方应用并不在 restricted 的范围内，很少会出现权限定义者不能访问自身定义的权限的情况，所以不允许三方应用填写 restricted。 |
| label | 选填，权限的简短描述，若未填写，则使用到简短描述的地方由权限名取代。 | 字符串 | 自定义 | 空 | 需要多语种适配。 |
| description | 选填，权限的详细描述，若未填写，则使用到详细描述的地方由 label 取代。 | 字符串 | 自定义 | 空 | 需要多语种适配。 |

**表 2** defPermissions 权限定义字段说明

| 授予方式 (grantMode) | 说明 | 自定义权限是否可指定该级别 | 取值样例 |
| --- | --- | --- | --- |

| 键 | 值说明 | 类型 | 取值范围 | 默认值 | 规则约束 |
| --- | --- | --- | --- | --- | --- |
| system_grant | 在“config.json”里面声明，安装后系统自动授予。 | | | 是 | GET_NETWORK_INFO 、GET_WIFI_INFO |
| user_grant | 在“config.json”里面声明，并在使用时动态申请，用户授权后才可使用。 | | | 否，如自定义则强制修改为system_grant。 | CAMERA、MICROPHONE |

**表 3** 权限授予方式字段说明

| 权限范围(availableScope) | 说明 | 自定义权限是否可指定该级别 | 取值样例 |
| --- | --- | --- | --- |
| restricted | 需要开发者向华为申请后才能被使用的特殊权限。 | 否 | ANSWER_CALL、READ_CALL_LOG、RECEIVE_SMS |
| signature | 权限定义方和使用方的签名一致。需在“config.json”里面声明后，由权限管理模块负责签名校验一致后，可使用。 | 是 | 对应用（或 Ability）操作的系统接口上由系统定义权限以及应用自定义的权限。如：find 某 Ability，连接某 Ability。 |
| privileged | 预置在系统版本中的特权应用可申请的权限。 | 是 | SET_TIME、MANAGE_USER_STORAGE |

**表 4** 权限限制范围字段说明

### 3.1.2.4 访问权限控制

- **Ability 的访问权限控制**

在 config.json 中填写“abilities”到“permissions”字段，即只有拥有该权限的应用可访问此 Ability。下面的例子表明只有拥有“ohos.permission.CAMERA”权限的应用可以访问此 ability。

```
1. "abilities": [{
2.         "name": ".MainAbility",
3.         "description": "$string:description_main_ability",
4.         "icon": "$media:hiworld.png",
5.         "label": "HiCamera",
6.         "launchType": "standard",
7.         "orientation": "portrait",
8.         "visible": false,
9.         "permissions": [
10.             "ohos.permission.CAMERA"
11.         ],
12.     }]
```

| 键 | 值说明 | 类型 | 取值范围 | 默认值 | 规则约束 |
| --- | --- | --- | --- | --- | --- |
| permissions | 选填，权限名称。用以表示此 ability 受哪个权限保护，即只有拥有此权限的应用可访问此 ability。 | 字符串 | 自定义 | 无 | 目前仅支持填写一个权限名，若填写多个权限名，仅第一个权限名称有效。 |

**表 5** 权限保护字段说明

- **Ability 接口的访问权限控制**

在 Ability 实现中，如需要对特定接口对调用者做访问控制，可在服务侧的接口实现中，主动通过 verifyCallingPermission、verifyCallingOrSelfPermission 来检查访问者是否拥有所需要的权限。

```
1. if (verifyCallingPermission("ohos.permission.CAMERA") != IBundleManager.PERMISSION_GRANTED) {
2.     // 调用者无权限，做错误处理
3. }
4.     // 调用者权限校验通过，开始提供服务
```

### 3.1.2.5 API 接口说明

| 接口原型 | 接口详细描述 |
| --- | --- |
| public int verifyPermission(String permissionName, int pid, int uid) | 接口功能：查询指定 PID、UID 的应用是否已被授予某权限<br>输入参数：permissionName：权限名；pid：进程 id；uid：uid<br>输出参数：无<br>返回值: IBundleManager.PERMISSION_DENIED、IBundleManager.PERMISSION_GRANTED |
| public int verifyCallingPermission(String permissionName) | 接口功能：查询 IPC 跨进程调用方的进程是否已被授予某权限<br>输入参数：permissionName：权限名<br>输出参数：无<br>返回值: IBundleManager.PERMISSION_DENIED、IBundleManager.PERMISSION_GRANTED |
| public int verifySelfPermission(String permissionName) | 接口功能：查询自身进程是否已被授予某权限<br>输入参数：permissionName：权限名<br>输出参数：无<br>返回值: IBundleManager.PERMISSION_DENIED、IBundleManager.PERMISSION_GRANTED |
| public int verifyCallingOrSelfPermission(String permissionName) | 接口功能：当有远端调用检查远端是否有权限，否则检查自身是否拥有权限<br>输入参数：permissionName：权限名<br>输出参数：无<br>返回值: IBundleManager.PERMISSION_DENIED、IBundleManager.PERMISSION_GRANTED |
| public boolean canRequestPermission(String permissionName) | 接口功能：向系统权限管理模块查询某权限是否不再弹框授权了<br>输入参数：permissionName：权限名<br>输出参数：无<br>返回值：true 允许弹框，false 不允许弹框 |
| void requestPermissionsFromUser (String[] permissions, int requestCode) | 接口功能：向系统权限管理模块申请权限（接口可支持一次申请多个。若下一步操作涉及到多个敏感权限，可以这么用，其他情况建议不要这么用。因为弹框还是按权限组一个个去弹框，耗时比较长。用到哪个权限就去申请哪个） |

| 接口原型 | 接口详细描述 |
| --- | --- |
|  | 输入参数： permissionNames:权限名列表；requestCode: 请求应答会带回此编码以匹配本次申请的权限请求<br>输出参数：无<br>返回值：无 |
| void onRequestPermissionsFromUserResult (int requestCode, String[] permissions, int[] grantResults) | 接口功能：调用 requestPermissionsFromUser 后的应答接口<br>输入参数：requestCode：requestPermission 中传入的 requestCode；permissions：申请的权限名；grantResults：申请权限的结果<br>输出参数：无<br>返回值：无 |

**表 6** 应用权限接口说明

### 3.1.2.6 动态申请权限开发步骤

1. 在 config.json 文件中声明所需要的权限。

```
{
    " reqPermissions": [{
        "name": "ohos.permission.CAMERA",
        "reason": "$string:permreason_camera",
        "usedScene": {
            "ability": ["com.mycamera.Ability", "com.mycamera.AbilityBackground"],
            "when": "always"}
        }, {
        ...
        }]
    }]
}
```

2. 使用 ohos.app.Context.verifySelfPermission 接口查询应用是否已被授予该权限。

- 如果已被授予权限，可以结束权限申请流程。
- 如果未被授予权限，继续执行下一步。

3. 使用 canRequestPermission 查询是否可动态申请。

- 如果不可动态申请，说明已被用户或系统永久禁止授权，可以结束权限申请流程。

- 如果可动态申请，继续执行下一步。

4. 使用 requestPermissionFromUser 动态申请权限，通过回调函数接受授予结果。

**样例代码如下：**

```
0.  if (verifySelfPermission("ohos.permission.CAMERA") != IBundleManager.PERMISSION_GRANTED) {
1.      // 应用未被授予权限
2.      if (canRequestPermission("ohos.permission.CAMERA")) {
3.          // 是否可以申请弹框授权(首次申请或者用户未选择禁止且不再提示)
4.          requestPermissionsFromUser(
5.                  new String[] { "ohos.permission.CAMERA" } , MY_PERMISSIONS_REQUEST_CAMERA);
6.      } else {
7.          // 显示应用需要权限的理由，提示用户进入设置授权
8.      }
9.  } else {
10.     // 权限已被授予
11. }
12.
13. @override
14. public void onRequestPermissionsFromUserResult (int requestCode, String[] permissions, int[] grantResults){
15.     switch (requestCode) {
16.         case MY_PERMISSIONS_REQUEST_CAMERA: {
17.             // 匹配 requestPermissions 的 requestCode
18.             if (grantResults.length > 0
19.                     && grantResults[0] == IBundleManager.PERMISSION_GRANTED) {
20.                 // 权限被授予
21.                 // 注意：因时间差导致接口权限检查时有无权限，所以对那些因无权限而抛异常的接口进行异常捕获处理
22.             } else {
23.                 // 权限被拒绝
24.             }
25.             return;
26.         }
27.     }
28. }
```

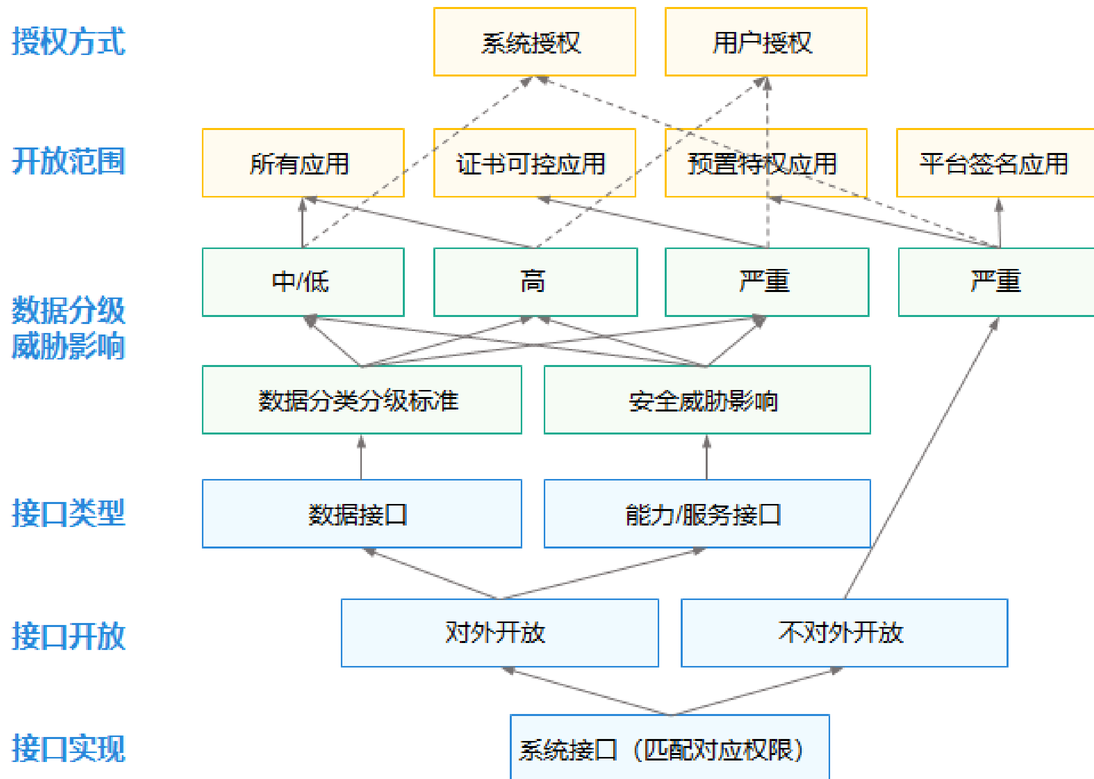
### 3.1.3 应用权限列表

#### 3.1.3.1 权限分类分级模型

HarmonyOS 的应用权限严格按照权限分类分级模型进行定义，如图 1 所示，具体过程可分为三步：

1. 根据不同应用所需实现的功能，明确接口是否需要对外开放。
2. 根据接口所涉数据的敏感程度或所涉能力的安全威胁影响，对所有的开放接口进行分级（包括中、低、高、严重）。不对外开放的接口均为严重级别。
3. 根据不同的分级，确定权限的开放范围与授权方式。

**图 1** 权限分类分级模型




HarmonyOS 已定义的权限列表详见《API 参考》中的“ohos.security.SystemPermission”。下面重点介绍对所有应用开放的 HarmonyOS 的应用权限。

### 3.1.3.2 敏感权限

敏感权限的申请需要按照动态申请流程向用户申请授权。

| 权限分类名称 | 权限名 | 说明 |
|---|---|---|
| 位置 | ohos.permission.LOCATION | 允许应用在前台运行时获取位置信息。如果应用在后台运行时也要获取位置信息，则需要同时申请 ohos.permission.LOCATION_IN_BACKGROUND 权限。 |
| | ohos.permission.LOCATION_IN_BACKGROUND | 允许应用在后台运行时获取位置信息，需要同时申请 ohos.permission.LOCATION 权限。 |
| 相机 | ohos.permission.CAMERA | 允许应用使用相机拍摄照片和录制视频。 |
| 麦克风 | ohos.permission.MICROPHONE | 允许应用使用麦克风进行录音。 |
| 日历 | ohos.permission.READ_CALENDAR | 允许应用读取日历信息。 |
| | ohos.permission.WRITE_CALENDAR | 允许应用在设备上添加、移除或修改日历活动。 |
| 健身运动 | ohos.permission.ACTIVITY_MOTION | 允许应用读取用户当前的运动状态。 |
| 健康 | ohos.permission.READ_HEALTH_DATA | 允许应用读取用户的健康数据。 |
| 媒体 | ohos.permission.MEDIA_LOCATION | 允许应用访问用户媒体文件中的地理位置信息。 |
| | ohos.permission.READ_MEDIA | 允许应用读取用户外部存储中的媒体文件信息。 |
| | ohos.permission.WRITE_MEDIA | 允许应用读写用户外部存储中的媒体文件信息。 |
| 帐号 | ohos.permission.GET_APP_ACCOUNTS | 允许应用访问系统帐号的分布式信息权限。 |

**表 1** 敏感权限说明

### 3.1.3.3 非敏感权限

非敏感权限不涉及用户的敏感数据或危险操作，仅需在 config.json 中声明，应用安装后即被授权。

| 权限名 | 说明 |
| --- | --- |
| ohos.permission.GET_NETWORK_INFO | 允许应用获取数据网络信息。 |
| ohos.permission.GET_WIFI_INFO | 允许获取 WLAN 信息。 |
| ohos.permission.USE_BLUETOOTH | 允许应用查看蓝牙的配置。 |
| ohos.permission.DISCOVER_BLUETOOTH | 允许应用配置本地蓝牙，并允许其查找远端设备且与之配对连接。 |
| ohos.permission.SET_NETWORK_INFO | 允许应用控制数据网络。 |
| ohos.permission.SET_WIFI_INFO | 允许配置 WLAN 设备。 |
| ohos.permission.SPREAD_STATUS_BAR | 允许应用以缩略图方式呈现在状态栏。 |
| ohos.permission.INTERNET | 允许使用网络 socket。 |
| ohos.permission.MODIFY_AUDIO_SETTINGS | 允许应用程序修改音频设置。 |
| ohos.permission.RECEIVER_STARTUP_COMPLETED | 允许应用接收设备启动完成广播。 |
| ohos.permission.RUNNING_LOCK | 允许申请休眠运行锁，并执行相关操作。 |
| ohos.permission.ACCESS_BIOMETRIC | 允许应用使用生物识别能力进行身份认证。 |
| ohos.permission.RCV_NFC_TRANSACTION_EVENT | 允许应用接收卡模拟交易事件。 |
| ohos.permission.COMMONEVENT_STICKY | 允许发布粘性公共事件的权限。 |
| ohos.permission.SYSTEM_FLOAT_WINDOW | 提供显示悬浮窗的能力。 |

| 权限名 | 说明 |
| --- | --- |
| ohos.permission.VIBRATE | 允许应用程序使用马达。 |
| ohos.permission.USE_TRUSTCIRCLE_MANAGER | 允许调用设备间认证能力。 |
| ohos.permission.USE_WHOLE_SCREEN | 允许通知携带一个全屏 IntentAgent。 |
| ohos.permission.SET_WALLPAPER | 允许设置静态壁纸。 |
| ohos.permission.SET_WALLPAPER_DIMENSION | 允许设置壁纸尺寸。 |
| ohos.permission.REARRANGE__MISSIONS | 允许调整任务栈。 |
| ohos.permission.CLEAN_BACKGROUND_PROCESSES | 允许根据包名清理相关后台进程。 |
| ohos.permission.KEEP_BACKGROUND_RUNNING | 允许 Service Ability 在后台继续运行。 |
| ohos.permission.GET_BUNDLE_INFO | 查询其他应用的信息。 |
| ohos.permission.ACCELEROMETER | 允许应用程序读取加速度传感器的数据。 |
| ohos.permission.GYROSCOPE | 允许应用程序读取陀螺仪传感器的数据。 |
| ohos.permission.MULTIMODAL_INTERACTIVE | 允许应用订阅语音或手势事件。 |
| ohos.permission.radio.ACCESS_FM_AM | 允许用户获取收音机相关服务。 |
| ohos.permission.NFC_TAG | 允许应用读写 Tag 卡片。 |
| ohos.permission.NFC_CARD_EMULATION | 允许应用实现卡模拟功能。 |

**表 2** 非敏感权限说明

### 3.1.3.4 受限开放的权限

受限开放的权限通常是不允许三方应用申请的。如果有特殊场景需要使用，请提供相关申请材料到应用市场申请相应权限证书。如果应用未申请相应的权限证书，却试图在 config.json 文件中声明此类权限，将会导致应用安装失败。另外，由于此类权限涉及到用户敏感数据或危险操作，当应用申请到权限证书后，还需按照动态申请权限的流程向用户申请授权。

| 权限分类名称 | 权限名 | 说明 |
| --- | --- | --- |
| 信息 | ohos.permission.READ_MESSAGES | 允许应用读取短信息。 |
| | ohos.permission.RECEIVE_MMS | 允许应用接收彩信。 |
| | ohos.permission.RECEIVE_SMS | 允许应用接收短信息。 |
| | ohos.permission.RECEIVE_WAP_MESSAGES | 允许应用接收 WAP 消息。 |
| | ohos.permission.SEND_MESSAGES | 允许应用发送短信。 |
| | ohos.permission.READ_CELL_MESSAGES | 允许应用读取小区广播消息。 |
| 通话记录 | ohos.permission.READ_CALL_LOG | 允许应用读取通话记录。 |
| | ohos.permission.WRITE_CALL_LOG | 允许应用在设备上添加、修改和删除通话记录。 |
| 通讯录 | ohos.permission.READ_CONTACTS | 允许应用读取联系人数据。 |
| | ohos.permission.WRITE_CONTACTS | 允许应用添加、移除和更改联系人数据。 |
| 电话 | ohos.permission.ANSWER_CALL | 允许应用接听来电。 |

**表 3** 受限开放权限说明

## 3.2 生物特征识别

### 3.2.1 概述

提供生物特征识别认证能力，即基于人体固有的生理特征和行为特征来识别用户身份，供第三方应用调用，可应用于设备解锁、支付、应用登录等身份认证场景。

当前生物特征识别能力提供 2D 人脸识别、3D 人脸识别两种人脸识别能力，设备具备哪种识别能力，取决于设备的硬件能力和技术实现。3D 人脸识别技术识别率、防伪能力都优于 2D 人脸识别技术，但具有 3D 人脸能力（比如 3D 结构光、3D TOF 等）的设备才可以使用 3D 人脸识别技术。

#### 3.2.1.1 基本概念

生物特征识别（又叫生物认证）：通过计算机与光学、声学、生物传感器和生物统计学原理等高科技手段密切结合，利用人体固有的生理特性（如指纹、面容、虹膜等）和行为特征（如笔迹、声音、步态等）来进行个人身份的鉴定。

人脸识别：基于人的脸部特征信息进行身份识别的一种生物特征识别技术，用摄像机或摄像头采集含有人脸的图像或视频流，并自动在图像中检测和跟踪人脸，进而对检测到的人脸进行脸部识别，通常也叫做人像识别、面部识别、人脸认证。

# 4 运作机制

人脸识别会在摄像头和 TEE（Trusted Execution Environment）之间建立安全通道，人脸图像信息通过安全通道传递到 TEE 中，由于人脸图像信息从 REE（Rich Execution Environment）侧无法获取，从而避免了恶意软件从 REE 侧进行攻击。对人脸图像采集、特征提取、活体检测、特征比对等处理完全在 TEE 中，基于 TrustZone 进行安全隔离，外部的人脸框架只负责人脸的认证发起和处理认证结果等数据，不涉及人脸数据本身。

人脸特征数据通过 TEE 的安全存储区进行存储，采用高强度的密码算法对人脸特征数据进行加密和完整性保护，外部无法获取到加密人脸特征数据的密钥，保证用户的人脸特征数据不会泄露。本能力采集和存储的人脸特征数据不会在用户未授权的情况下被传出 TEE，这意味着，用户未授权时，无论是系统应用还是三方应用都无法获得人脸特征数据，也无法将人脸特征数据传送或备份到任何外部存储介质。

#### 4.1.1.1 约束与限制

- 当前版本提供的生物特征识别能力只包含人脸识别，且只支持本地认证，不提供认证界面。
- 要求设备上具备摄像器件，且人脸图像像素大于 100*100。
- 要求设备上具有 TEE 安全环境，人脸特征信息高强度加密保存在 TEE 中。
- 对于面部特征相似的人（比如双胞胎、兄弟姐妹等）、面部特征不断发育的儿童，人脸特征匹配率有所不同。如果对此担忧，可考虑其他认证方式。

### 4.1.2 开发指导

#### 4.1.2.1 场景介绍

当前生物特征识别支持 2D 人脸识别、3D 人脸识别，可应用于设备解锁、应用登录、支付等身份认证场景。

#### 4.1.2.2 接口说明

BiometricAuthentication 类提供了生物认证的相关方法，包括检测认证能力、认证和取消认证等，用户可以通过人脸等生物特征信息进行认证操作。在执行认证前，需要检查设备是否支持该认证能力，具体指认证类型、安全级别和是否本地认证。如果不支持，需要考虑使用其他认证能力。

| 接口名 | 功能描述 |
| --- | --- |
| getInstance(Ability ability) | 获取 BiometricAuthentication 的单例对象。 |
| checkAuthenticationAvailability(AuthType type, SecureLevel level, boolean isLocalAuth) | 检测设备是否具有生物认证能力。 |
| execAuthenticationAction(AuthType type, SecureLevel level, boolean isLocalAuth,boolean isAppAuthDialog, SystemAuthDialogInfo information) | 调用者使用该方法进行生物认证。可以使用自定义的认证界面，也可以使用系统提供的认证界面。当使用系统认证界面时，调用者可以自定义提示语。该方法直到认证结束才返回认证结果。 |
| getAuthenticationTips() | 获取生物认证过程中的提示信息。 |
| cancelAuthenticationAction() | 取消生物认证操作。 |
| setSecureObjectSignature(Signature sign) | 设置需要关联认证结果的 Signature 对象，在进行认证操作后，如果认证成功则 Signature 对象被授权可以使用。设置前 Signature 对象需要正确初始化，且配置为认证成功才能使用。 |
| getSecureObjectSignature() | 在认证成功后，可通过该方法获取已授权的 Signature 对象。如果未设置过 Signature 对象，则返回 null。 |
| setSecureObjectCipher(Cipher cipher) | 设置需要关联认证结果的 Cipher 对象，在进行认证操作后，如果认证成功则 Cipher 对象被授权可以使用。设置前 Cipher 对象需要正确初始化，且配置为认证成功才能使用。 |

| 接口名 | 功能描述 |
| --- | --- |
| getSecureObjectCipher() | 在认证成功后，可通过该方法获取已授权的 Cipher 对象。如果未设置过 Cipher 对象，则返回 null。 |
| setSecureObjectMac(Mac mac) | 设置需要关联认证结果的 Mac 对象，在进行认证操作后，如果认证成功则 Mac 对象被授权可以使用。设置前 Mac 对象需要正确初始化，且配置为认证成功才能使用。 |
| getSecureObjectMac() | 在认证成功后，可通过该方法获取已授权的 Mac 对象。如果未设置过 Mac 对象，则返回 null。 |

**表 1** 生物特征识别开放能力列表：

### 4.1.2.3 开发步骤

开发前请完成以下准备工作：

1. 在应用配置权限文件中，增加 ohos.permission.ACCESS_BIOMETRIC 的权限声明。
2. 在使用生物特征识别认证能力的代码文件中增加 import ohos.biometrics.authentication.BiometricAuthentication。

开发过程：

1. 获取 BiometricAuthentication 的单例对象，代码示例如下：

```
BiometricAuthentication  mBiometricAuthentication = BiometricAuthentication.getInstance(MainAbility.mAbility);
```

2. 检测设备是否具有生物认证能力：

   2D 人脸识别建议使用 SECURE_LEVEL_S2，3D 人脸识别建议使用 SECURE_LEVEL_S3。代码示例如下：

```
int retChkAuthAvb = mBiometricAuthentication.checkAuthenticationAvailability(
BiometricAuthentication.AuthType.AUTH_TYPE_BIOMETRIC_FACE_ONLY, BiometricAuthentication.SecureLevel.SECURE_LEVEL_S2, true);
```

3. （可选）设置需要关联认证结果的 Signature 对象或 Cipher 对象或 Mac 对象，代码示例如下：

```
// 定义一个 Signature 对象 sign;
mBiometricAuthentication.setSecureObjectSignature(sign);
```

```
3.
4. // 定义一个 Cipher 对象 cipher;
5. mBiometricAuthentication.setSecureObjectCipher(cipher);
6.
7. // 定义一个 Mac 对象 mac;
8. mBiometricAuthentication.setSecureObjectMac(mac);
```

4. 在新线程里面执行认证操作，避免阻塞其他操作，代码示例如下：

```
1. new Thread(new Runnable() {
2.     @Override
3.     public void run() {
4.         int retExcAuth;
5.         retExcAuth =
   mBiometricAuthentication.execAuthenticationAction(    BiometricAuthentication.AuthType.AUTH_TYPE_BIOMETRIC_FACE_ONLY
   ,    BiometricAuthentication.SecureLevel.SECURE_LEVEL_S2, true, false, null);
6.     }
7. }).start();
```

5. 获得认证过程中的提示信息，代码示例如下：

```
1. AuthenticationTips  mTips = mBiometricAuthentication.getAuthenticationTips();
```

6. （可选）认证成功后获取已设置的 Signature 对象或 Cipher 对象或 Mac 对象，代码示例如下：

```
1. Signature  sign = mBiometricAuthentication.getSecureObjectSignature();
2.
3. Cipher cipher = mBiometricAuthentication.getSecureObjectCipher();
4.
5. Mac mac = mBiometricAuthentication.getSecureObjectMac();
```

7. 认证过程中取消认证，代码示例如下：

```
1. int  ret = mBiometricAuthentication.cancelAuthenticationAction();
```

# 5 AI

# 5.1 码生成

## 5.1.1 概述

码生成能够根据开发者给定的字符串信息和二维码图片尺寸，返回相应的二维码图片字节流。

调用方可以通过二维码字节流生成二维码图片。

### 5.1.1.1 约束与限制

- 当前仅支持生成 QR 二维码（Quick Response Code）。由于 QR 二维码算法的限制，字符串信息的长度不能超过 2953 个字符。
- 生成的二维码图片的宽度不能超过 1920 像素，高度不能超过 1680 像素。由于 QR 二维码是通过正方形阵列承载信息的，建议二维码图片采用正方形，当二维码图片采用长方形时，会在 QR 二维码信息的周边区域留白。

## 5.1.2 开发指导

### 5.1.2.1 场景介绍

码生成能够根据给定的字符串信息，生成相应的二维码图片。常见应用场景举例：

- 社交或通讯类应用：根据输入的联系人信息，生成联系人二维码。
- 购物或支付类应用：根据输入的支付链接，生成收款或付款二维码。

### 5.1.2.2 接口说明

码生成提供了的 IBarcodeDetector()接口，常用方法的功能描述如下：

| 接口名 | 方法 | 功能描述 |
| --- | --- | --- |
| IBarcodeDetector | int detect(String barcodeInput, byte[] bitmapOutput, int width, int height); | 根据给定的信息和二维码图片尺寸，生成二维码图片字节流。 |

| 接口名 | 方法 | 功能描述 |
|---|---|---|
| IBarcodeDetector | int release(); | 停止 QR 码生成服务，释放资源。 |

### 5.1.2.3 开发步骤

1. 在使用码生成 SDK 时，需要先将相关的类添加至工程。

```
1.  import ohos.cvinterface.common.ConnectionCallback;
2.  import ohos.cvinterface.common.VisionManager;
3.  import ohos.cvinterface.qrcode.IBarcodeDetector;
```

2. 定义 ConnectionCallback 回调，实现连接能力引擎成功与否后的操作。

```
1.  ConnectionCallback connectionCallback = new ConnectionCallback() {
2.      @Override
3.      public void onServiceConnect() {
4.          // Do something when service connects successfully
5.      }
6.
7.      @Override
8.      public void onServiceDisconnect() {
9.          // Do something when service connects unsuccessfully
10.     }
11. };
```

3. 调用 VisionManager.init()方法，将此工程的 context 和 connectionCallback 作为入参，建立与能力引擎的连接，context 应为 ohos.aafwk.ability.Ability 或 ohos.aafwk.ability.AbilitySlice 的实例或子类实例。

```
1.  int result = VisionManager.init(context, connectionCallback);
```

4. 实例化 IBarcodeDetector 接口，将此工程的 context 作为入参。

```
1.  IBarcodeDetector barcodeDetector = VisionManager.getBarcodeDetector(context);
```

5. 定义码生成图像的尺寸，并根据图像大小分配字节流数组空间。

```
1.  final int SAMPLE_LENGTH = 152;
2.  byte[] byteArray = new byte[SAMPLE_LENGTH * SAMPLE_LENGTH * 4];
```

6. 调用 IBarcodeDetector 的 detect()方法，根据输入的字符串信息生成相应的二维码图片字节流。

```
1.  int result = barcodeDetector.detect("This is a TestCase of IBarcodeDetector", byteArray, SAMPLE_LENGTH, SAMPLE_LENGTH);
```

如果返回值为 0，表明调用成功。

7. 当码生成能力使用完毕后，调用 IBarcodeDetector 的 release()方法，释放资源。

```
1.   result = barcodeDetector.release();
```

8. 调用 VisionManager.destroy()方法，断开与能力引擎的连接。

```
1.   VisionManager.destroy();
```

# 6 网络与连接

## 6.1 NFC

### 6.1.1 概述

NFC（Near Field Communication，近距离无线通信技术）是一种非接触式识别和互联技术，让移动设备、消费类电子产品、PC 和智能设备之间可以进行近距离无线通信。

HarmonyOS 提供了 NFC 基础控制、Tag 读写、安全单元访问、卡模拟以及 NFC 消息通知的功能。

### 6.1.2 NFC 基础控制

#### 6.1.2.1 场景介绍

应用或者其他模块可以通过接口完成以下功能：

1. 查询本机是否支持 NFC 能力。
2. 开启或者关闭本机 NFC。

### 6.1.2.2 接口说明

| 类名 | 接口名 | 功能描述 |
| --- | --- | --- |
| NfcController | getInstance(Context context) | 获得一个 NFC 控制类的单例。 |
| | openNfc() | 打开本机 NFC。 |
| | closeNfc() | 关闭本机 NFC。 |
| | isNfcOpen() | 查询本机 NFC 是否已打开。 |
| | getNfcState() | 获取本机 NFC 的开关状态。 |
| | isNfcAvailable() | 查询本机是否支持 NFC 功能。 |
| NfcPermissionException | NfcPermissionException(String errorMessage) | 构造一个 NFC 权限异常的实例。 |

**表 1** NFC 开关控制功能的主要接口

### 6.1.2.3 开发步骤

1. 调用 NfcController 类的 getInstance()接口，获取 NfcController 实例，管理本机 NFC 操作。
2. 调用 isNfcOpen()接口，查询 NFC 是否打开。
3. 调用 openNfc()接口打开 NFC；或者调用 closeNfc()接口关闭 NFC。

```
1.  // 查询本机是否支持 NFC
2.  NfcController nfcController = NfcController.getInstance(context);
3.  boolean isAvailable = nfcController.isNfcAvailable();
4.  if (isAvailable) {
5.      // 调用查询 NFC 是否打开接口，返回值为 NFC 是否是打开的状态
6.      boolean isOpen = nfcController.isNfcOpen();
7.  
8.      if (!isOpen) {
9.          // 调用打开 NFC 接口,返回值为函数是否正常执行
10.         boolean isEnableSuccess = nfcController.openNfc();
11.     } else {
```

```
12.             // 调用关闭 NFC 接口,返回值为函数是否正常执行
13.             boolean isDisableSuccess = nfcController.closeNfc();
14.         }
15.     }
```

### 6.1.3 Tag 读写

#### 6.1.3.1 场景介绍

应用或其他模块可以通过接口访问多种协议或技术的 Tag 卡片。

#### 6.1.3.2 接口说明

| 类名 | 接口名 | 功能描述 |
| --- | --- | --- |
| TagInfo | getTagId() | 获取当前 Tag 的 ID。 |
| | getTagSupportedProfiles() | 获取当前 Tag 支持的协议或技术。 |
| | isProfileSupported(int profile) | 判断 Tag 是否支持指定的协议或技术。 |
| TagManager | getTagInfo() | 获取标签信息。 |
| | connectTag() | 建立与 Tag 设备的连接。 |
| | reset() | 重置与 Tag 设备的连接，同时会把写 Tag 的超时时间恢复为默认值。 |
| | isTagConnected() | 判断与 Tag 设备是否保持连接。 |
| | setSendDataTimeout(int timeout) | 设置发送数据到 Tag 的超时时间，单位是 ms。 |
| | getSendDataTimeout() | 查询发送数据到 Tag 设备的超时时间，单位是 ms。 |
| | sendData(byte[] data) | 写数据到 Tag 设备中。 |

| 类名 | 接口名 | 功能描述 |
| --- | --- | --- |
| | checkConnected() | 检查 Tag 设备是否已连接。 |
| | getMaxSendLength() | 获取发送数据的最大长度。在发送数据到 Tag 设备时，用于查询最大可发送数据的长度。 |
| IsoDepTag | IsoDepTag(TagInfo tagInfo) | 根据分发 Tag 信息获取 IsoDep 类型 Tag 标签对象。 |
| | getHiLayerResponse() | 获取基于 NfcB 技术的 IsoDep 类型 Tag 的高层响应内容。 |
| | getHistoricalBytes() | 获取基于 NfcA 技术的 IsoDep 类型 Tag 的历史字节内容。 |
| NfcATag | getInstance(TagInfo tagInfo) | 根据分发 Tag 信息获取 NfcA 标签对象。 |
| | getSak() | 获取 NfcA 类型 Tag 的 SAK。 |
| | getAtqa() | 获取 NfcA 类型 Tag 的 ATQA。 |
| NfcBTag | getInstance(TagInfo tagInfo) | 根据分发 Tag 信息获取 NfcB 标签对象。 |
| | getRespAppData() | 获取 NfcB 标签对象的应用数据。 |
| | getRespProtocol() | 获取 NfcB 标签对象的协议信息。 |
| NdefTag | getNdefMessage() | 获取当前连接的 NDEF Tag 设备的 NDEF 信息。 |
| | getNdefMaxSize() | 获取最大的 NDEF 信息尺寸。 |
| | getTagType() | 获取 NDEF Tag 设备类型。 |
| | readNdefMessage() | 从当前连接的 NDEF Tag 设备读取 NDEF 信息。 |
| | writeNdefMessage(NdefMessage msg) | 写 NDEF 信息到当前连接的 NDEF Tag 设备。 |

| 类名 | 接口名 | 功能描述 |
|---|---|---|
| | canSetReadOnly() | 检查 NDEF Tag 设备是否可被设置为只读。 |
| | setReadOnly() | 设置 NDEF Tag 设备为只读。 |
| | isNdefWritable() | 判断 NDEF Tag 设备是否可写。 |
| MifareClassicTag | getInstance(TagInfo tagInfo) | 根据分发 Tag 信息获取 MifareClassic 标签对象。 |
| | getMifareType() | 获取 MifareClassic Tag 设备的 Mifare 类型。 |
| | getTagSize() | 获取 MifareClassic Tag 设备的尺寸。 |
| | getSectorsNum() | 获取 MifareClassic Tag 设备内所有扇区数。 |
| | getBlocksNum() | 获取 MifareClassic Tag 设备内所有块数。 |
| | getBlocksNumForSector(int sectorId) | 获取 MifareClassic Tag 设备内一个扇区的块数。 |
| | getSectorId(int blockId) | 用块 ID 获取 MifareClassic Tag 设备内扇区号。 |
| | getFirstBlockId(int sectorId) | 获取 MifareClassic Tag 设备内特定扇区的第一个块 ID。 |
| | authenSectorUseKey(int sectorId, byte[] key, byte keyType) | 用密钥鉴权 MifareClassic Tag 设备内特定扇区。 |
| | readBlock(int blockId) | 读取 MifareClassic Tag 设备内特定块内容。 |
| | writeBlock(int blockId, byte[] data) | 在 MifareClassic Tag 设备内特定块写内容。 |
| | incBlock(int blockId, int value) | 在 MifareClassic Tag 设备内特定块的内容加上一个值。 |
| | decBlock(int blockId, int value) | 从 MifareClassic Tag 设备内特定块的内容减去一个值。 |

| 类名 | 接口名 | 功能描述 |
| --- | --- | --- |
| | restoreBlock(int blockId) | 把 MifareClassic Tag 设备内特定块的内容移动到一个内部的缓存区。 |
| MifareUltralightTag | getInstance(TagInfo tagInfo) | 根据分发 Tag 信息获取 MifareUltralight 标签对象。 |
| | getMifareType() | 获取 MifareUltralight Tag 设备类型。 |
| | readFourPages(int pageOffset) | 从 MifareUltralight Tag 设备的特定页数开始读四页。 |
| | writeOnePage(int pageOffset, byte[] data) | 在 MifareUltralight Tag 设备的特定页数写数据。 |

**表 1** Tag 读写功能的主要接口

### 6.1.3.3 读取卡片类型

1. 从 Intent 中获取 TagInfo，初始化 TagInfo 实例。
2. TagInfo 实例调用 getTagSupportedProfiles()接口查询当前 Tag 支持的技术或协议类型。
3. 调用 isProfileSupported(int profile)接口查询是否支持 NfcA、IsoDep、MifareClassic 等类型。若支持，可使用 TagInfo 实例构造 NfcATag、IsoDep、MifareClassic 等实例。
4. 根据不同的 Tag 技术类型的实例，调用不同的 API 完成 Tag 的访问。

```
1.  // 从 Intent 中获取 TagInfo，初始化 TagInfo 实例
2.  TagInfo tagInfo = getIntent().getParcelableExtra(NfcController.EXTRA_TAG_INFO);
3.
4.  // 查询 Tag 设备支持的技术或协议，返回值为支持的技术或协议列表
5.  int[] profiles = tagInfo.getTagSupportedProfiles();
6.
7.  // 查询是否支持 NfcA，若支持，构造一个 NfcATag
8.  boolean isSupportedNfcA = tagInfo.isProfileSupported(TagManager.NFC_A);
9.  if (isSupportedNfcA) {
10.     NfcATag tagNfcA = NfcATag.getInstance(tagInfo);
11. }
12.
13. // 查询是否支持 NfcB，若支持，构造一个 NfcBTag
```

```
14. boolean isSupportedNfcB = tagInfo.isProfileSupported(TagManager.NFC_B);
15. if (isSupportedNfcB) {
16.     NfcBTag tagNfcB = NfcBTag.getInstance(tagInfo);
17. }
18.
19. // 查询是否支持 IsoDep，若支持，构造一个 IsoDepTag
20. boolean isSupportedIsoDep = tagInfo.isProfileSupported(TagManager.ISO_DEP);
21. if (isSupportedIsoDep) {
22.     IsoDepTag tagIsoDep = new IsoDepTag(tagInfo);
23. }
24.
25. // 查询是否支持 NDEF，若支持，构造一个 NdefTag
26. boolean isSupportedNdefDep = tagInfo.isProfileSupported(TagManager.NDEF);
27. if (isSupportedNdefDep) {
28.     NdefTag tagNdef = new NdefTag(tagInfo);
29. }
30.
31. // 查询是否支持 MifareClassic，若支持，构造一个 MifareClassicTag
32. boolean isSupportedMifareClassic = tagInfo.isProfileSupported(TagManager.MIFARE_CLASSIC);
33. if (isSupportedMifareClassic) {
34.     MifareClassicTag mifareClassicTag = MifareClassicTag.getInstance(tagInfo);
35. }
36.
37. // 查询是否支持 MifareUltralight，若支持，构造一个 MifareUltralightTag
38. boolean isSupportedMifareUltralight = tagInfo.isProfileSupported(TagManager.MIFARE_ULTRALIGHT);
39. if (isSupportedMifareUltralight) {
40.     MifareUltralightTag mifareUltralightTag = MifareUltralightTag.getInstance(tagInfo);
41. }
```

**访问 NfcA 卡片**

1. 调用 connectTag()接口连接 Tag 设备。
2. 调用 isTagConnected()接口查询 Tag 设备连接状态。
3. 调用 sendData(byte[] data)接口发送数据到 Tag。

```
1. // 连接 Tag 设备，返回值为是否连接成功
2. boolean connSuccess = tagNfcA.connectTag();
3.
4. // 查询 Tag 连接状态
5. boolean isConnected= tagNfcA.isTagConnected();
6.
```

```
7. // 发送数据到 Tag，返回值为 Tag 的响应数据
8. byte[] data = {0x13, 0x59, 0x22};
9. byte[] response = tagNfcA.sendData(data);
```

**访问 NfcB 卡片**

1. 调用 connectTag()接口连接 Tag 设备。
2. 调用 isTagConnected()接口查询 Tag 设备连接状态。
3. 调用 sendData(byte[] data)接口发送数据到 Tag。

```
1. // 连接 Tag 设备，返回值为是否连接成功
2. boolean connSuccess = tagNfcB.connectTag();
3.
4. // 查询 Tag 连接状态
5. boolean isConnected= tagNfcB.isTagConnected();
6.
7. // 发送数据到 Tag，返回值为 Tag 的响应数据
8. byte[] data = {0x13, 0x59, 0x22};
9. byte[] response = tagNfcB.sendData(data);
```

**访问 IsoDep 卡片**

1. 调用 getTagInfo()接口获取 TagInfo 对象。
2. 调用 connectTag()接口连接 Tag 设备。
3. 调用 isTagConnected()接口查询 Tag 设备连接状态。
4. 调用 sendData(byte[] data)接口发送数据到 Tag。

```
1. // 连接 Tag 设备，返回值为是否连接成功
2. boolean connSuccess = tagIsoDep.connectTag();
3.
4. // 查询 Tag 连接状态
5. boolean isConnected= tagIsoDep.isTagConnected();
6.
7. // 发送数据到 Tag，返回值为 Tag 的响应数据
8. byte[] data = {0x13, 0x59, 0x22};
9. byte[] response = tagIsoDep.sendData(data);
```

**访问 Ndef 卡片**

1. 调用 getTagInfo()接口获取 TagInfo 对象。
2. 调用 connectTag()接口连接 Tag 设备。
3. 调用 isTagConnected()接口查询 Tag 设备连接状态。
4. 调用 sendData(byte[] data)接口发送数据到 Tag。

```
1. // 连接 Tag 设备，返回值为是否连接成功
```

```
2.  boolean connSuccess = tagNdef.connectTag();
3.
4.  // 查询 Tag 连接状态
5.  boolean isConnected = tagNdef.isTagConnected();
6.
7.  // 发送数据到 Tag，返回值为 Tag 的响应数据
8.  byte[] data = {0x13, 0x59, 0x22};
9.  byte[] response = tagNdef.sendData(data);
```

**访问 MifareClassic 卡片**

1. 调用 getTagInfo()接口获取 TagInfo 对象。
2. 调用 connectTag()接口连接 Tag 设备。
3. 调用 isTagConnected()接口查询 Tag 设备连接状态。
4. 调用 sendData(byte[] data)接口发送数据到 Tag。

```
1.  // 连接 Tag 设备，返回值为是否连接成功
2.  boolean connSuccess = mifareClassicTag.connectTag();
3.
4.  // 查询 Tag 连接状态
5.  boolean isConnected = mifareClassicTag.isTagConnected();
6.
7.  // 发送数据到 Tag，返回值为 Tag 的响应数据
8.  byte[] data = {0x13, 0x59, 0x22};
9.  byte[] response = mifareClassicTag.sendData(data);
```

**访问 MifareUltralight 卡片**

1. 调用 getTagInfo()接口获取 TagInfo 对象。
2. 调用 connectTag()接口连接 Tag 设备。
3. 调用 isTagConnected()接口查询 Tag 设备连接状态。
4. 调用 sendData(byte[] data)接口发送数据到 Tag。

```
1.  // 连接 Tag 设备，返回值为是否连接成功
2.  boolean connSuccess = mifareUltralightTag.connectTag();
3.
4.  // 查询 Tag 连接状态
5.  boolean isConnected = mifareUltralightTag.isTagConnected();
6.
7.  // 发送数据到 Tag，返回值为 Tag 的响应数据
8.  byte[] data = {0x13, 0x59, 0x22};
9.  byte[] response = mifareUltralightTag.sendData(data);
```

## 6.1.4 访问 SE 安全单元

### 6.1.4.1 场景介绍

应用或者其他模块可以通过接口完成以下功能：

1. 获取安全单元的个数和名称。
2. 判断安全单元是否在位。
3. 在指定安全单元上打开基础通道。
4. 在指定安全单元上打开逻辑通道。
5. 发送 APDU（Application Protocol Data Unit）数据到安全单元上。

### 6.1.4.2 接口说明

| 类名 | 接口名 | 功能描述 |
| --- | --- | --- |
| SEService | SEService() | 创建一个安全单元服务的实例。 |
| | isConnected() | 查询安全单元服务是否已连接。 |
| | shutdown() | 关闭安全单元服务。 |
| | getReaders() | 获取全部安全单元。 |
| | getVersion() | 获得安全单元服务的版本。 |
| | OnCallback | 用于回调的内部类，用于定义回调接口。在服务连接成功后，回调该接口通知应用。 |
| Reader | getName() | 获取安全单元的名称。 |
| | isSecureElementPresent() | 检查安全单元是否在位。 |
| | openSession() | 打开当前安全单元上的 session。 |

| 类名 | 接口名 | 功能描述 |
| --- | --- | --- |
| | closeSession() | 关闭当前安全单元上的所有 session。 |
| Session | openBasicChannel(Aid aid) | 打开基础通道。 |
| | openLogicalChannel(Aid aid) | 创建逻辑通道。 |
| | getATR() | 获得重设安全单元指令的响应。 |
| | closeSessionChannels() | 关闭当前 session 的所有通道。 |
| Channel | isClosed() | 判断通道是否关闭。 |
| | isBasicChannel() | 判断是否是基础通道。 |
| | transmit(byte[] command) | 发送指令到安全单元。 |
| | getSelectResponse() | 获得应用程序选择指令的响应。 |
| | closeChannel() | 关闭通道。 |
| Aid | Aid(byte[] aid, int offset, int length) | 构造一个 AID 类的实例。 |
| | isAidValid() | 查询 AID 是否有效。 |
| | getAidBytes() | 获取 AID 的字节数组形式的值。 |

**表 1** NFC 访问安全单元功能的的主要接口

### 6.1.4.3 开发步骤

1. 调用 SEService 类的构造函数，创建一个安全单元服务的实例，用于访问安全单元。
2. 调用 isConnected()接口，查询安全单元服务的连接状态。
3. 调用 getReaders()接口，获取本机的全部安全单元。
4. 调用 Reader 类的 openSession()接口打开 Session，返回一个打开的 Session 实例。

5. 调用 Session 类的 openBasicChannel(Aid aid)接口打开基础通道，或者调用 openLogicalChannel(Aid aid)接口打开逻辑通道，返回一个打开通道 Channel 实例。
6. 调用 Channel 类的 transmit(byte[] command)，发送 APDU 到安全单元。
7. 调用 Channel 类的 closeChannel()接口关闭通道。
8. 调用 Session 类的 closeSessionChannels()接口关闭 Session 的所有通道。
9. 调用 Reader 类的 closeSessions()接口关闭安全单元的所有 Session。
10. 调用 SEService 类的 shutdown()接口关闭安全单元服务。

```
1.  private class AppServiceConnectedCallback implements SEService.OnCallback {
2.      @Override
3.      public void serviceConnected() {
4.          // 应用自实现
5.      }
6.  }
7.  // 创建安全单元服务实例
8.  SEService sEService = new SEService(context, new AppServiceConnectedCallback());
9.  // 查询安全单元服务的连接状态
10. boolean isConnected = sEService.isConnected();
11.
12. // 获取本机的全部安全单元，并获取指定的安全单元 eSE
13. Reader[] elements = sEService.getReaders();
14. Reader eSe = null;
15. for (int i = 0; i < elements.length; i++) {
16.     if ("eSE".equals(elements[i].getName())) {
17.         eSe = elements[i];
18.         break;
19.     }
20. }
21.
22. // 查询安全单元是否在位
23. boolean isPresent = eSe.isSecureElementPresent();
24.
25. // 打开 Session
26. Optional<Session> optionalSession = eSe.openSession();
27. Session session = optionalSession.orElse(null);
28.
29. // 打开通道
30. if (eSe != null) {
31.     byte[] aidValue = new byte[]{(byte)0x01, (byte)0x02, (byte)0x03, (byte)0x04, (byte)0x05};
```

```
32.         // 创建 Aid 实例
33.         Aid aid = new Aid(aidValue, 0, aidValue.length);
34.         // 打开基础通道
35.         Optional<Channel> optionalChannel = session.openBasicChannel(aid);
36.         Channel basicChannel = optionalChannel.orElse(null);
37.         // 打开逻辑通道
38.         optionalChannel = session.openLogicalChannel(aid);
39.         Channel logicalChannel = optionalChannel.orElse(null);
40.
41.         // 发送指令给安全单元，返回值为安全单元对指令的响应
42.         byte[] resp = logicalChannel.transmit(new byte[]{(byte)0x00, (byte)0xa4, (byte)0x00, (byte)0x00, (byte)0x02, (byte)0x00,
    (byte)0x00});
43.
44.         // 关闭通道资源
45.         basicChannel.closeChannel()
46.         logicalChannel.closeChannel();
47.     }
48.
49.     // 关闭 Session 资源
50.     session.close();
51.
52.     // 关闭安全单元资源
53.     eSe.closeSessions();
54.
55.     // 关闭安全单元服务资源 sEService.shutdown();
```

## 6.1.5 卡模拟功能

### 6.1.5.1 场景介绍

应用或者其他模块可以通过接口完成以下功能：

1. 查询是否支持指定安全单元的卡模拟功能，安全单元包括 HCE（Host Card Emulation）、ESE（Embedded Secure Element）和 SIM（Subscriber Identity Module）卡。
2. 开关卡模拟以及查询卡模拟状态，可以打开或关闭指定技术类型的卡模拟。
3. 获取 NFC 信息，信息包括当前激活的安全单元、Hisee 上电状态、是否支持 RSSI 查询等信息。

4. 根据 NFC 服务的类型获取刷卡时选择服务的方式，应用或者其它模块可以查询支付（Payment）类型和非支付（Other）类型业务选择服务的方式。
5. 动态设置和注销前台优先应用。
6. NFC 应用的 AID 相关操作，包括注册和删除应用的 AID、查询应用是否是指定 AID 的默认应用、获取应用的 AID 等。
7. 定义 Host 和 OffHost 服务的抽象类，三方应用通过继承抽象类来实现 NFC 卡模拟功能。

### 6.1.5.2 接口说明

| 类名 | 接口名 | 功能描述 |
|---|---|---|
| CardEmulation | getInstance(NfcController controller) | 创建一个卡模拟类的实例。 |
| | isSupported(int feature) | 查询是否支持卡模拟功能。 |
| | setListenMode(int mode) | 设置卡模拟模式。 |
| | isListenModeEnabled() | 查询卡模拟功能是否打开。 |
| | getNfcInfo(String key) | 获取 NFC 的信息。 |
| | getSelectionType(String category) | 根据 NFC 服务的类型获取刷卡时选择服务的方式。 |
| | registerForegroundPreferred(Ability appAbility, ElementName appName) | 动态设置前台优先应用。 |
| | unregisterForegroundPreferred(Ability appAbility) | 取消设置前台优先应用。 |
| | isDefaultForAid(ElementName appName, String aid) | 判断应用是否是指定 AID 的默认处理应用。 |
| | registerAids(ElementName appName, String type, List<String> aids) | 给应用注册指定类型的 AID。 |
| | removeAids(ElementName appName, String type) | 删除应用的指定类型的 AID。 |

| 类名 | 接口名 | 功能描述 |
|---|---|---|
| | getAids(ElementName appName, String type) | 获取应用中指定类型的 AID 列表。 |
| HostService | sendResponse(byte[] response) | 发送响应的数据到对端设备。 |
| | handleRemoteCommand(byte[] cmd, IntentParams params) | 处理对端设备发送的命令。 |
| | disabledCallback(int errCode) | 连接异常的回调。 |
| OffHostService | onConnect(Intent intent) | 连接服务并获取远程服务对象。 |

**表 1** NFC 卡模拟功能的主要接口

## 查询是否支持卡模拟功能

1. 调用 NfcController 类的 getInstance(Context context)接口，获取 NfcController 实例。
2. 调用 CardEmulation 类的 getInstance(NfcController controller)接口，获取 CardEmulation 实例，去管理本机卡模拟模块操作。
3. 调用 isSupported(int feature)接口去查询是否 HCE、UICC、ESE 卡模拟。

```
1. // 获取 NFC 控制对象
2. NfcController nfcController = NfcController.getInstance(context);
3. // 获取卡模拟控制对象
4. CardEmulation cardEmulation = CardEmulation.getInstance(nfcController);
5. // 查询是否支持 HCE、UICC、ESE 卡模拟，返回值表示是否支持对应安全单元的卡模拟
6. boolean isSupportedHce = cardEmulation.isSupported(CardEmulation.FEATURE_HCE);
7. boolean isSupportedUicc = cardEmulation.isSupported(CardEmulation.FEATURE_UICC);
8. boolean isSupportedEse = cardEmulation.isSupported(CardEmulation.FEATURE_ESE);
```

## 开关卡模拟及查询卡模拟状态

1. 调用 NfcController 类的 getInstance(Context context)接口，获取 NfcController 实例。
2. 调用 CardEmulation 类的 getInstance(NfcController controller)接口，获取 CardEmulation 实例，去管理本机卡模拟模块操作。
3. 调用 setListenMode(int mode)接口去打开或者关闭卡模拟。
4. 调用 isListenModeEnabled()接口去查询卡模拟是否打开。

```
1. // 获取 NFC 控制对象
2. NfcController nfcController = NfcController.getInstance(context);
3. // 获取卡模拟控制对象
```

```
4. CardEmulation cardEmulation = CardEmulation.getInstance(nfcController);
5. // 打开卡模拟
6. cardEmulation.setListenMode(CardEmulation.ENABLE_MODE_ALL);
7. // 调用查询卡模拟开关状态的接口，返回值为卡模拟是否是打开的状态
8. boolean isEnabled = cardEmulation.isListenModeEnabled(); // true
9. // 关闭卡模拟
10. cardEmulation.setListenMode(CardEmulation.DISABLE_MODE_A_B);
11. // 调用查询卡模拟开关状态的接口，返回值为卡模拟是否是打开的状态
12. isEnabled = cardEmulation.isListenModeEnabled(); // false
```

### 获取 NFC 信息

1. 调用 NfcController 类的 getInstance(Context context)接口，获取 NfcController 实例。
2. 调用 CardEmulation 类的 getInstance(NfcController controller)接口，获取 CardEmulation 实例，去管理本机卡模拟模块操作。
3. 调用 getNfcInfo(String key)接口去获取 NFC 信息。

```
1. // 获取 NFC 控制对象
2. NfcController nfcController = NfcController.getInstance(context);
3. // 获取卡模拟控制对象
4. CardEmulation cardEmulation = CardEmulation.getInstance(nfcController);
5. // 查询本机当前使能的安全单元类型
6. String seType = cardEmulation.getNfcInfo(CardEmulation.KEY_ENABLED_SE_TYPE); // ENABLED_SE_TYPE_ESE
7. // 查询 Hisee 上电状态
8. String hiseeState = cardEmulation.getNfcInfo(CardEmulation.KEY_HISEE_READY);
9. // 查询是否支持 rssi 的查询
10. String rssiAbility = cardEmulation.getNfcInfo(CardEmulation.KEY_RSSI_SUPPORTED);
```

### 根据 NFC 服务的类型获取刷卡时选择服务的方式

1. 调用 NfcController 类的 getInstance(Context context)接口，获取 NfcController 实例。
2. 调用 CardEmulation 类的 getInstance(NfcController controller)接口，获取 CardEmulation 实例，去管理本机卡模拟模块操作。
3. 调用 getSelectionType(Sring category)接口去获取选择服务的方式。

```
1. // 获取 NFC 控制对象
2. NfcController nfcController = NfcController.getInstance(context);
3. // 获取卡模拟控制对象
4. CardEmulation cardEmulation = CardEmulation.getInstance(nfcController);
5. // 获取选择服务的方式
6. int result = cardEmulation.getSelectionType(CardEmulation.CATEGORY_PAYMENT); // SELECTION_TYPE_PREFER_DEFAULT
```

```
7.  result = cardEmulation.getSelectionType(CardEmulation.CATEGORY_OTHER); // SELECTION_TYPE_ASK_IF_CONFLICT
```

### 动态设置和注销前台优先应用

1. 调用 NfcController 类的 getInstance(Context context)接口，获取 NfcController 实例。
2. 调用 CardEmulation 类的 getInstance(NfcController controller)接口，获取 CardEmulation 实例，去管理本机卡模拟模块操作。
3. 调用 registerForegroundPreferred(Ability appAbility, ElementName appName)接口去动态设置前台优先应用。
4. 调用 unregisterForegroundPreferred(Ability appAbility)接口去取消设置前台优先应用。

```
1.  // 获取 NFC 控制对象
2.  NfcController nfcController = NfcController.getInstance(context);
3.  // 获取卡模拟控制对象
4.  CardEmulation cardEmulation = CardEmulation.getInstance(nfcController);
5.  // 动态设置前台优先应用
6.  cardEmulation.registerForegroundPreferred(new Ability(), new ElementName());
7.  // 注销前台优先应用
8.  cardEmulation.unregisterForegroundPreferred(new Ability());
```

### NFC 应用的 AID 相关操作

1. 调用 NfcController 类的 getInstance(Context context)接口，获取 NfcController 实例。
2. 调用 CardEmulation 类的 getInstance(NfcController controller)接口，获取 CardEmulation 实例，去管理本机卡模拟模块操作。
3. 调用 registerAids(ElementName appName, String type, List<String> aids)接口去给应用注册指定类型的 AID。
4. 调用 removeAids(ElementName appName, String type)接口去删除应用的指定类型的 AID。
5. 调用 isDefaultForAid(ElementName appName, String aid)接口去判断应用是否是指定 AID 的默认处理应用。
6. 调用 getAids(ElementName appName, String type)接口去获取应用中指定类型的 AID 列表。

```
1.  // 获取 NFC 控制对象
2.  NfcController nfcController = NfcController.getInstance(context);
3.  // 获取卡模拟控制对象
4.  CardEmulation cardEmulation = CardEmulation.getInstance(nfcController);
5.  // 给应用注册指定类型的 AID
6.  List<String> aids = new ArrayList<String>();
7.  aids.add(0, "A0028321901280");
8.  aids.add(1, "A0028321901281");
9.  try {     cardEmulation.registerAids(new ElementName(), CardEmulation.CATEGORY_PAYMENT, aids);
10. } catch (IllegalArgumentException e) {
11.     HiLog.error(LABEL, "IllegalArgumentException when registerAids");
12. }
```

```
13. // 删除应用的指定类型的 AID
14. cardEmulation.removeAids(new ElementName(), CardEmulation.CATEGORY_PAYMENT);
15. cardEmulation.removeAids(new ElementName(), CardEmulation.CATEGORY_OTHER);
16. // 判断应用是否是指定 AID 的默认处理应用
17. String aid = "A0028321901280";
18. cardEmulation.isDefaultForAid(new ElementName(), aid);
19. // 获取应用中指定类型的 AID 列表
20. try {
21.     cardEmulation.getAids(new ElementName(), CardEmulation.CATEGORY_PAYMENT);
22. } catch (NullPointerException e) {
23.     HiLog.error(LABEL, "NullPointerException when getAids");
24. } catch (IllegalArgumentException e) {
25.     HiLog.error(LABEL, "IllegalArgumentException when getAids");
26. }
```

### Host 服务的抽象类

1. 三方应用的服务继承 HostService，实现 HCE 卡模拟功能。
2. 三方应用自定义实现抽象方法 handleRemoteCommand(byte[] cmd, IntentParams params)和 disabledCallback()。
3. 三方应用自定义功能。

```
1. // 三方 HCE 应用的服务继承 HostService，实现 HCE 卡模拟功能
2. public class AppService extends HostService {
3.     @Override
4.     public byte[] handleRemoteCommand(byte[] cmd, IntentParams params) {
5.     // 三方应用自定义接口实现。
6.     }
7.
8.     @Override
9.     public void disabledCallback(int errCode) {
10.    // 三方应用自定义接口实现。
11.    }
12.
13.    // 三方应用自定义功能
14. }
```

### 6.1.6 NFC 消息通知

#### 6.1.6.1 场景介绍

NFC 消息通知（Notification）是 HarmonyOS 内部或者与应用之间跨进程通讯的机制，注册者在注册消息通知后，一旦符合条件的消息被发出，注册者即可接收到该消息。

#### 6.1.6.2 接口说明

| 描述 | 通知名 | 附加参数 |
| --- | --- | --- |
| NFC 状态 | usual.event.nfc.action.ADAPTER_STATE_CHANGED | extra_nfc_state |
| 进场消息 | usual.event.nfc.action.RF_FIELD_ON_DETECTED | extra_nfc_transaction |
| 离场消息 | usual.event.nfc.action.RF_FIELD_OFF_DETECTED | - |

**表 1** NFC 消息通知的相关广播介绍

**注册并获取 NFC 状态改变消息**

1. 构建消息通知接收者 NfcStateEventSubscriber。
2. 注册 NFC 状态改变消息。
3. NfcStateEventSubscriber 接收并处理 NFC 状态改变消息。

```
1. // 构建消息接收者/注册者
2. class NfcStateEventSubscriber extends CommonEventSubscriber {
3.     NfcStateEventSubscriber (CommonEventSubscribeInfo info) {
4.         super(info);
5.     }
6.
7.     @Override
8.     public void onReceiveEvent(CommonEventData commonEventData) {
9.         if (commonEventData == null || commonEventData.getIntent() == null) {
10.            return;
11.        }
```

```
12.             if (NfcController.STATE_CHANGED.equals(commonEventData.getIntent().getAction())) {
13.                 IntentParams params = commonEventData.getIntent().getParams();
14.                 if (params != null) {
15.                     int currState = commonEventData.getIntent().getIntParam(NfcController.EXTRA_NFC_STATE,
NfcController.STATE_OFF);
16.                 }
17.             }
18.         }
19. }
20.
21. // 注册消息
22. MatchingSkills matchingSkills = new MatchingSkills();
23. // 增加获取 NFC 状态改变消息
24. matchingSkills.addEvent(NfcController.STATE_CHANGED);
25. matchingSkills.addEvent(CommonEventSupport.COMMON_EVENT_NFC_ACTION_ADAPTER_STATE_CHANGED);
26. CommonEventSubscribeInfo subscribeInfo = new CommonEventSubscribeInfo(matchingSkills);
27. NfcStateEventSubscriber subscriber = new NfcStateEventSubscriber(subscribeInfo);
28. try {
29.     CommonEventManager.subscribeCommonEvent(subscriber);
30. } catch (RemoteException e) {
31.     HiLog.e(TAG, "doSubscribe occur exception:" + e.toString());
32. }
```

### 注册并获取 NFC 场强消息

1. 构建消息通知接收者 NfcFieldOnAndOffEventSubscriber。
2. 注册 NFC 场强消息。
3. NfcFieldOnAndOffEventSubscriber 接收并处理 NFC 场强消息。

```
1. // 构建消息接收者/注册者
2. class NfcFieldOnAndOffEventSubscriber extends CommonEventSubscriber {
3.     NfcFieldOnAndOffEventSubscriber (CommonEventSubscribeInfo info) {
4.         super(info);
5.     }
6.
7.     @Override
8.     public void onReceiveEvent(CommonEventData commonEventData) {
9.         if (commonEventData == null || commonEventData.getIntent() == null) {
10.             return;
11.         }
```

```
12.             if (NfcController.FIELD_ON_DETECTED.equals(commonEventData.getIntent().getAction())) {
13.                 IntentParams params = commonEventData.getIntent().getParams();
14.                 if (params == null) {
15.                     HiLog.i(TAG, "Pure FIELD_ON_DETECTED");
16.                 } else {
17.                     HiLog.i(TAG, "Transaction FIELD_ON_DETECTED");
18.                     Intent transactionIntent = (Intent) params.getParam("transactionIntent");
19.                 }
20.             } else if (NfcController.FIELD_OFF_DETECTED.equals(commonEventData.getIntent().getAction())) {
21.                 HiLog.i(TAG, "FIELD_OFF_DETECTED");
22.             }
23.             HiLog.i(TAG, "MyFieldOnAndOffEventSubscriber onReceiveEvent .....:" + commonEventData.getIntent().getAction());
24.         }
25.     }
26.
27.     // 注册消息
28.     MatchingSkills matchingSkills = new MatchingSkills();
29.     // 增加获取 NFC 状态改变消息
30.     matchingSkills.addEvent(NfcController.FIELD_ON_DETECTED);
31.     matchingSkills.addEvent(NfcController.FIELD_OFF_DETECTED);
32.     CommonEventSubscribeInfo subscribeInfo = new CommonEventSubscribeInfo(DomainMode.BOTH, matchingSkills);
33.     HiLog.i(TAG, "subscribeInfo permission: " + subscribeInfo.getPermission());
34.     MyFieldOnAndOffEventSubscriber subscriber = new MyFieldOnAndOffEventSubscriber(subscribeInfo);
35.     try {
36.         CommonEventManager.subscribeCommonEvent(subscriber);
37.     } catch (RemoteException e) {
38.         HiLog.e(TAG, "doSubscribe occur exception:" + e.toString());
39.     }
```

## 6.2 蓝牙

### 6.2.1 概述

蓝牙是短距离无线通信的一种方式，支持蓝牙的两个设备必须配对后才能通信。HarmonyOS 蓝牙主要分为传统蓝牙和低功耗蓝牙（通常称为 BLE，Bluetooth Low Energy）。传统蓝牙指的是蓝牙版本 3.0 以下的蓝牙，低功耗蓝牙指的是蓝牙版本 4.0 以上的蓝牙。

当前蓝牙的配对方式有两种：蓝牙协议 2.0 以下支持 PIN 码（Personal Identification Number，个人识别码）配对，蓝牙协议 2.1 以上支持简单配对。

**传统蓝牙**

HarmonyOS 传统蓝牙提供的功能有：

- 传统蓝牙本机管理：打开和关闭蓝牙、设置和获取本机蓝牙名称、扫描和取消扫描周边蓝牙设备、获取本机蓝牙 profile 对其他设备的连接状态、获取本机蓝牙已配对的蓝牙设备列表。
- 传统蓝牙远端设备操作：查询远端蓝牙设备名称和 MAC 地址、设备类型和配对状态，以及向远端蓝牙设备发起配对。

**BLE**

BLE 设备交互时会分为不同的角色：

- 中心设备和外围设备：中心设备负责扫描外围设备、发现广播。外围设备负责发送广播。
- GATT（Generic Attribute Profile，通用属性配置文件）服务端与 GATT 客户端：两台设备建立连接后，其中一台作为 GATT 服务端，另一台作为 GATT 客户端。

HarmonyOS 低功耗蓝牙提供的功能有：

- BLE 扫描和广播：根据指定状态获取外围设备、启动或停止 BLE 扫描、广播。

#### 6.2.1.1 约束与限制

调用蓝牙的打开接口需要 ohos.permission.USE_BLUETOOTH 权限，调用蓝牙扫描接口需要 ohos.permission.LOCATION 权限和 ohos.permission.DISCOVER_BLUETOOTH 权限。

### 6.2.2 传统蓝牙本机管理

#### 6.2.2.1 场景介绍

传统蓝牙本机管理主要是针对蓝牙本机的基本操作，包括打开和关闭蓝牙、设置和获取本机蓝牙名称、扫描和取消扫描周边蓝牙设备、获取本机蓝牙 profile 对其他设备的连接状态、获取本机蓝牙已配对的蓝牙设备列表。

#### 6.2.2.2 接口说明

| 接口名 | 功能描述 |
| --- | --- |
| getDefaultHost(Context context) | 获取 BluetoothHost 实例，去管理本机蓝牙操作。 |
| enableBt() | 打开本机蓝牙。 |
| disableBt() | 关闭本机蓝牙。 |
| setLocalName(String name) | 设置本机蓝牙名称。 |
| getLocalName() | 获取本机蓝牙名称。 |
| getBtState() | 获取本机蓝牙状态。 |
| startBtDiscovery() | 发起蓝牙设备扫描。 |
| cancelBtDiscovery() | 取消蓝牙设备扫描。 |

| 接口名 | 功能描述 |
| --- | --- |
| isBtDiscovering() | 检查蓝牙是否在扫描设备中。 |
| getProfileConnState(int profile) | 获取本机蓝牙 profile 对其他设备的连接状态。 |
| getPairedDevices() | 获取本机蓝牙已配对的蓝牙设备列。 |

**表 1** 蓝牙本机管理类 BluetoothHost 的主要接口

### 打开蓝牙

1. 调用 BluetoothHost 的 getDefaultHost(Context context)接口，获取 BluetoothHost 实例，管理本机蓝牙操作。
2. 调用 enableBt()接口，打开蓝牙。
3. 调用 getBtState()，查询蓝牙是否打开。

```
1. // 获取蓝牙本机管理对象
2. BluetoothHost bluetoothHost = BluetoothHost.getDefaultHost(context);
3. // 调用打开接口
4. bluetoothHost.enableBt();
5. // 调用获取蓝牙开关状态接口
6. int state = bluetoothHost.getBtState();
```

### 蓝牙扫描

1. 开始蓝牙扫描前要先注册广播 BluetoothRemoteDevice.EVENT_DEVICE_DISCOVERED。
2. 调用 startBtDiscovery()接口开始进行扫描外围设备。
3. 如果想要获取扫描到的设备，必须在注册广播时继承实现 CommonEventSubscriber 类的 onReceiveEvent(CommonEventData data)方法，并接收 EVENT_DEVICE_DISCOVERED 广播。

```
1. //开始扫描
2. mBluetoothHost.startBtDiscovery();
3. //接收系统广播
4. public class MyCommonEventSubscriber extends CommonEventSubscriber {
5.     @Override
6.     public void onReceiveEvent(CommonEventData var){
7.         Intent info = var.getIntent();
8.         if(info == null) return;
9.         //获取系统广播的 action
10.        String action = info.getAction();
11.        //判断是否为扫描到设备的广播
```

```
12.            if(action == BluetoothRemoteDevice.EVENT_DEVICE_DISCOVERED){
13.                IntentParams myParam = info.getParams();
14.                BluetoothRemoteDevice device =
(BluetoothRemoteDevice)myParam.getParam(BluetoothRemoteDevice.REMOTE_DEVICE_PARAM_DEVICE);
15.            }
16.        }
17.    }
```

### 6.2.3 传统蓝牙远端设备操作

#### 6.2.3.1 场景介绍

传统蓝牙远端管理操作主要是针对远端蓝牙设备的基本操作，包括获取远端蓝牙设备地址、类型、名称和配对状态，以及向远端设备发起配对。

#### 6.2.3.2 接口说明

| 接口名 | 功能描述 |
| --- | --- |
| getDeviceAddr() | 获取远端蓝牙设备地址。 |
| getDeviceClass() | 获取远端蓝牙设备类型。 |
| getDeviceName() | 获取远端蓝牙设备名称。 |
| getPairState() | 获取远端设备配对状态。 |
| startPair() | 向远端设备发起配对。 |

**表 1** 蓝牙远端设备管理类 BluetoothRemoteDevice 的主要接口

#### 6.2.3.3 开发步骤

1. 调用 BluetoothHost 的 getDefaultHost(Context context)接口，获取 BluetoothHost 实例，管理本机蓝牙操作。

2. 调用 enableBt()接口，打开蓝牙。
3. 调用 startBtDiscovery()，扫描设备。
4. 调用 startPair()，发起配对。

```
1. // 获取蓝牙本机管理对象
2. BluetoothHost bluetoothHost = BluetoothHost.getDefaultHost(context);
3. // 调用打开接口
4. bluetoothHost.enableBt();
5. // 调用扫描接口
6. bluetoothHost.startBtDiscovery();
7. //设置界面会显示出扫描结果列表，点击蓝牙设备去配对
8. BluetoothRemoteDevice device = bluetoothHost.getRemoteDev(TEST_ADDRESS);
9. device.startPair();
```

### 6.2.4 BLE 扫描和广播

#### 6.2.4.1 场景介绍

通过 BLE 扫描和广播提供的开放能力，可以根据指定状态获取外围设备、启动或停止 BLE 扫描、广播。

#### 6.2.4.2 接口说明

| 接口名 | 功能描述 |
| --- | --- |
| startScan(List<BleScanFilter> filters) | 进行 BLE 蓝牙扫描，并使用 filters 对结果进行过滤。 |
| stopScan() | 停止 BLE 蓝牙扫描。 |
| getDevicesByStates(int[] states) | 根据状态获取连接的外围设备。 |
| BleCentralManager(BleCentralManagerCallback callback) | 获取中心设备管理对象。 |

**表 1** BLE 中心设备管理类 BleCentralManager 的主要接口

| 接口名 | 功能描述 |
| --- | --- |

| 接口名 | 功能描述 |
| --- | --- |
| onScanCallback(BleScanResult result) | 扫描到 BLE 设备的结果回调。 |
| onStartScanFailed(int resultCode) | 启动扫描失败的回调。 |

**表 2** 中心设备管理回调类 BleCentralManagerCallback 的主要接口

| 接口名 | 功能描述 |
| --- | --- |
| BleAdvertiser(Context context, BleAdvertiseCallback callback) | 用于获取广播操作对象。 |
| startAdvertising(BleAdvertiseSettings settings, BleAdvertiseData advData, BleAdvertiseData scanResponse) | 进行 BLE 广播，第一个参数为广播参数，第二个为广播数据，第三个参数是扫描和广播数据参数的响应。 |
| stopAdvertising() | 停止 BLE 广播。 |
| startResultEvent(int result) | 广播回调结果。 |

**表 3** BLE 广播相关的 BleAdvertiser 类和 BleAdvertiseCallback 类的主要接口

## 中心设备进行 BLE 扫描

1. 进行 BLE 扫描之前先要继承 BleCentralManagerCallback 类实现 onScanCallback 和 onStartScanFailed 回调函数，用于接收扫描结果。
2. 调用 BleCentralManager(BleCentralManagerCallback callback)接口获取中设备管理对象。
3. 获取扫描过滤器，过滤器为空时为不使用过滤器扫描，然后调用 startScan()开始扫描 BLE 设备，在回调中获取扫描到的 BLE 设备。

```
1.  // 实现扫描回调
2.  public class ScanCallback implements BleCentralManagerCallback{
3.      List<BleScanResult>results = new ArrayList<BleScanResult>();
4.      @Override
5.      public void onScanCallback(BleScanResult var1) {
6.          // 对扫描结果进行处理
7.          results.add(var1);
8.      }
9.      @Override
10.     public void onStartScanFailed(int var1) {
```

```
11.             HiLog.info(TAG,"Start Scan failed,Code:" + var1);
12.         }
13.     }
14. // 获取中心设备管理对象
15. private ScanCallback centralManagerCallback = new ScanCallback();
16. private BleCentralManager centralManager = new BleCentralManager(centralManagerCallback);
17. // 创建扫描过滤器然后开始扫描
18. List<BleScanFilter> filters = new ArrayList<BleScanFilter>();
19. centralManager.startScan(filters);
```

### 外围设备进行 BLE 广播

1. 进行 BLE 广播前需要先继承 advertiseCallback 类实现 startResultEvent 回调，用于获取广播结果。
2. 调用接口 BleAdvertiser(Context context, BleAdvertiseCallback callback)获取广播对象，构造广播参数和广播数据。
3. 调用 startAdvertising(BleAdvertiseSettings settings, BleAdvertiseData advData, BleAdvertiseData scanResponse)接口开始 BLE 广播。

```
1. // 实现 BLE 广播回调
2. private BleAdvertiseCallback advertiseCallback = new BleAdvertiseCallback() {
3.     @Override
4.     public void startResultEvent(int result) {
5.         if(result == BleAdvertiseCallback.RESULT_SUCC){
6.             // 开始 BLE 广播成功
7.         }
8.         else {
9.             // 开始 BLE 广播失败
10.         }
11.     }
12. };
13. // 获取 BLE 广播对象
14. private BleAdvertiser advertiser = new BleAdvertiser(this,advertiseCallback);
15. // 创建 BLE 广播参数和数据
16. private BleAdvertiseData data = new BleAdvertiseData.Builder()
17.                         .addServiceUuid(SequenceUuid.uuidFromString(Server_UUID))      // 添加服务的 UUID
18.                         .addServiceData(SequenceUuid.uuidFromString(Server_UUID),new byte[]{0x11})     // 添加广播数据内容
19.                         .build();
20. private BleAdvertiseSettings advertiseSettings = new BleAdvertiseSettings.Builder()
21.                        .setConnectable(true)                  // 设置是否可连接广播
22.                        .setInterval(BleAdvertiseSettings.INTERVAL_SLOT_DEFAULT)     // 设置广播间隔
```

```
23.                         .setTxPower(BleAdvertiseSettings.TX_POWER_DEFAULT)        // 设置广播功率
24.                         .build();
25. // 开始广播
26. advertiser.startAdvertising(advertiseSettings,data,null);
```

## 6.3 WLAN

### 6.3.1 概述

无线局域网（Wireless Local Area Networks，WLAN），是通过无线电、红外光信号或者其他技术发送和接收数据的局域网，用户可以通过 WLAN 实现结点之间无物理连接的网络通讯。常用于用户携带可移动终端的办公、公众环境中。

HarmonyOS WLAN 服务系统为用户提供 WLAN 基础功能、P2P（peer-to-peer）功能和 WLAN 消息通知的相应服务，让应用可以通过 WLAN 和其他设备互联互通。

#### 6.3.1.1 约束与限制

本开发指南提供多个开发场景的指导，涉及多个 API 接口的调用。在调用 API 前，应用需要先申请对应的访问权限，具体请参照对应场景的开放能力介绍。

### 6.3.2 WLAN 基础功能

#### 6.3.2.1 场景介绍

应用或者其他模块可以通过接口完成以下功能：

1. 获取 WLAN 状态，查询 WLAN 是否打开。
2. 发起扫描并获取扫描结果。

3. 获取连接态详细信息，包括连接信息、IP 信息等。
4. 获取设备国家码。
5. 获取设备是否支持指定的能力。

### 6.3.2.2 接口说明

WifiDevice 提供 WLAN 的基本功能，其接口说明如下。

| 接口名 | 描述 | 所需权限 |
| --- | --- | --- |
| getInstance(Context context) | 获取 WLAN 功能管理对象实例，通过该实例调用 WLAN 基本功能 API。 | NA |
| isWifiActive() | 获取当前 WLAN 打开状态。 | ohos.permission.GET_WIFI_INFO |
| scan() | 发起 WLAN 扫描。 | ohos.permission.SET_WIFI_INFO<br>ohos.permission.LOCATION |
| getScanInfoList() | 获取上次扫描结果。 | ohos.permission.GET_WIFI_INFO<br>ohos.permission.LOCATION |
| isConnected() | 获取当前 WLAN 连接状态。 | ohos.permission.GET_WIFI_INFO |
| getLinkedInfo() | 获取当前的 WLAN 连接信息。 | ohos.permission.GET_WIFI_INFO |
| getIpInfo() | 获取当前连接的 WLAN IP 信息。 | ohos.permission.GET_WIFI_INFO |
| getSignalLevel(int rssi, int band) | 通过 RSSI 与频段计算信号格数。 | NA |
| getCountryCode() | 获取设备的国家码。 | ohos.permission.LOCATION<br>ohos.permission.GET_WIFI_INFO |
| isFeatureSupported(long featureId) | 获取设备是否支持指定的特性。 | ohos.permission.GET_WIFI_INFO |

**表 1** WifiDevice 的主要接口

### 6.3.2.3 获取 WLAN 状态

1. 调用 WifiDevice 的 getInstance(Context context)接口，获取 WifiDevice 实例，用于管理本机 WLAN 操作。
2. 调用 isWifiActive()接口查询 WLAN 是否打开。

```
1. // 获取 WLAN 管理对象
2. WifiDevice wifiDevice = WifiDevice.getInstance(context);
3. // 调用获取 WLAN 开关状态接口
4. boolean isWifiActive  = wifiDevice.isWifiActive(); // 若 WLAN 打开，则返回 true，否则返回 false
```

### 6.3.2.4 发起扫描并获取结果

1. 调用 WifiDevice 的 getInstance(Context context)接口，获取 WifiDevice 实例，用于管理本机 WLAN 操作。
2. 调用 scan()接口发起扫描。
3. 调用 getScanInfoList()接口获取扫描结果。

```
1. // 获取 WLAN 管理对象
2. WifiDevice wifiDevice = WifiDevice.getInstance(context);
3. // 调用 WLAN 扫描接口
4. boolean isScanSuccess = wifiDevice.scan(); // true
5. // 调用获取扫描结果
6.  List<WifiScanInfo> scanInfos = wifiDevice.getScanInfoList();
```

### 6.3.2.5 获取连接态详细信息

1. 调用 WifiDevice 的 getInstance(Context context)接口，获取 WifiDevice 实例，用于管理本机 WLAN 操作。
2. 调用 isConnected()接口获取当前连接状态。
3. 调用 getLinkedInfo()接口获取连接信息。
4. 调用 getIpInfo()接口获取 IP 信息。

```
1. // 获取 WLAN 管理对象
2. WifiDevice wifiDevice = WifiDevice.getInstance(context);
3. // 调用 WLAN 连接状态接口,确定当前设备是否连接 WLAN
4. boolean isConnected = wifiDevice.isConnected();
5. if (isConnected) {
6.     // 获取 WLAN 连接信息
7.     Optional<WifiLinkedInfo> linkedInfo = wifiDevice.getLinkedInfo();
8.     // 获取连接信息中的 SSID
```

```
9.          String ssid = linkedInfo.get().getSsid();
10.         // 获取 WLAN 的 IP 信息
11.         Optional<IpInfo> ipInfo = wifiDevice.getIpInfo();
12.         // 获取 IP 信息中的 IP 地址与网关
13.         int ipAddress = ipInfo.get().getIpAddress();
14.         int gateway = ipInfo.get().getGateway();
15.     }
```

### 6.3.2.6 获取设备国家码

1. 调用 WifiDevice 的 getInstance(Context context)接口，获取 WifiDevice 实例，用于管理本机 WLAN 操作。
2. 调用 getCountryCode()接口获取设备的国家码。

```
1.  // 获取 WLAN 管理对象
2.  WifiDevice wifiDevice = WifiDevice.getInstance(context);
3.  // 获取当前设备的国家码
4.  String countryCode = wifiDevice.getCountryCode();
```

### 6.3.2.7 判断设备是否支持指定的能力

1. 调用 WifiDevice 的 getInstance(Context context)接口，获取 WifiDevice 实例，用于管理本机 WLAN 操作。
2. 调用 isFeatureSupported(long featureId)接口判断设备是否支持指定的能力。

```
1.  // 获取 WLAN 管理对象
2.  WifiDevice wifiDevice = WifiDevice.getInstance(context);
3.  // 获取当前设备是否支持指定的能力
4.  boolean isSupport = wifiDevice.isFeatureSupported(WifiUtils.WIFI_FEATURE_INFRA);
5.  isSupport = wifiDevice.isFeatureSupported(WifiUtils.WIFI_FEATURE_INFRA_5G);
6.  isSupport = wifiDevice.isFeatureSupported(WifiUtils.WIFI_FEATURE_PASSPOINT);
7.  isSupport = wifiDevice.isFeatureSupported(WifiUtils.WIFI_FEATURE_P2P);
8.  isSupport = wifiDevice.isFeatureSupported(WifiUtils.WIFI_FEATURE_MOBILE_HOTSPOT);
9.  isSupport = wifiDevice.isFeatureSupported(WifiUtils.WIFI_FEATURE_AWARE);
10. isSupport = wifiDevice.isFeatureSupported(WifiUtils.WIFI_FEATURE_AP_STA);
11. isSupport = wifiDevice.isFeatureSupported(WifiUtils.WIFI_FEATURE_WPA3_SAE);
12. isSupport = wifiDevice.isFeatureSupported(WifiUtils.WIFI_FEATURE_WPA3_SUITE_B);
13. isSupport = wifiDevice.isFeatureSupported(WifiUtils.WIFI_FEATURE_OWE);
```

## 6.3.3 P2P 功能

### 6.3.3.1 场景介绍

WLAN P2P 功能用于设备与设备之间的点对点数据传输，应用可以通过接口完成以下功能：

1. 发现对端设备。
2. 建立与移除群组。
3. 向对端设备发起连接。
4. 获取 P2P 相关信息。

### 6.3.3.2 接口说明

WifiP2pController 提供 WLAN P2P 功能，接口说明如下。

| 接口名 | 描述 | 所需权限 |
| --- | --- | --- |
| init(EventRunner eventRunner, WifiP2pCallback callback) | 初始化 P2P 的信使，当且仅当信使被成功初始化，P2P 的其他功能才可以正常使用。 | ohos.permission.GET_WIFI_INFO ohos.permission.SET_WIFI_INFO |
| discoverDevices(WifiP2pCallback callback) | 搜索附近可用的 P2P 设备。 | ohos.permission.GET_WIFI_INFO |
| stopDeviceDiscovery(WifiP2pCallback callback) | 停止搜索附近的 P2P 设备。 | ohos.permission.GET_WIFI_INFO |
| createGroup(WifiP2pConfig wifiP2pConfig, WifiP2pCallback callback) | 建立 P2P 群组。 | ohos.permission.GET_WIFI_INFO |
| removeGroup(WifiP2pCallback callback) | 移除 P2P 群组。 | ohos.permission.GET_WIFI_INFO |
| requestP2pInfo(int requestType, WifiP2pCallback callback) | 请求 P2P 相关信息，如群组信息、连接信息、设备信息等。 | ohos.permission.GET_WIFI_INFO |
| connect(WifiP2pConfig wifiP2pConfig, WifiP2pCallback callback) | 向指定设备发起连接。 | ohos.permission.GET_WIFI_INFO |

| 接口名 | 描述 | 所需权限 |
| --- | --- | --- |
| cancelConnect(WifiP2pCallback callback) | 取消向指定设备发起的连接。 | ohos.permission.GET_WIFI_INFO |

**表 1** WifiP2pController 的主要接口

#### 6.3.3.3 启动与停止 P2P 搜索的开发步骤

1. 调用 WifiP2pController 的 getInstance(Context context)接口，获取 P2P 控制器实例，用于管理 P2P 操作。
2. 调用 init(EventRunner eventRunner, WifiP2pCallback callback)初始化 P2P 控制器实例。
3. 发起 P2P 搜索。
4. 获取 P2P 搜索回调信息。
5. 停止 P2P 搜索。

```
1.  try {
2.      // 获取 P2P 管理对象
3.      WifiP2pController wifiP2pController = WifiP2pController.getInstance(this);
4.      // 初始化 P2P 管理对象，用于建立 P2P 信使等行为
5.      wifiP2pController.init(EventRunner.create(true), null);
6.      // 创建 P2P 回调对象
7.      P2pDiscoverCallBack p2pDiscoverCallBack = new P2pDiscoverCallBack();
8.      // 发起 P2P 搜索
9.      wifiP2pController.discoverDevices(p2pDiscoverCallBack);
10.     // 停止 P2P 搜索
11.     wifiP2pController.stopDeviceDiscovery(p2pDiscoverCallBack);
12. } catch (RemoteException re) {
13.     HiLog.warn(LABEL, "exception happened.");
14. }
15.
16. // 获取 P2P 启动与停止搜索的回调信息（失败或者成功）
17. private class P2pDiscoverCallBack extends WifiP2pCallback {
18.     @Override
19.     public void eventExecFail(int reason) {
20.         HiLog.info(LABEL, "discoverDevices eventExecFail reason : %{public}d", reason);
21.     }
22.
23.     @Override
```

```
24.     public void eventExecOk() {
25.         HiLog.info(LABEL, "discoverDevices eventExecOk");
26.     }
27. }
```

### 6.3.3.4 创建与移除群组的开发步骤

1. 调用 WifiP2pController 的 getInstance(Context context)接口，获取 P2P 控制器实例，用于管理 P2P 操作。
2. 调用 init(EventRunner eventRunner, WifiP2pCallback callback)初始化 P2P 控制器实例。
3. 创建 P2P 群组。
4. 移除 P2P 群组。

```
1. try {
2.     // 获取 P2P 管理对象
3.     WifiP2pController wifiP2pController = WifiP2pController.getInstance(this);
4.     // 初始化 P2P 管理对象，用于建立 P2P 信使等行为
5.     wifiP2pController.init(EventRunner.create(true), null);
6.     // 创建用于 P2P 建组需要的配置
7.     WifiP2pConfig wifiP2pConfig = new WifiP2pConfig("DEFAULT_GROUP_NAME", "DEFAULT_PASSPHRASE");
8.     wifiP2pConfig.setDeviceAddress("02:02:02:02:03:04");
9.     wifiP2pConfig.setGroupOwnerBand(0);
10.    // 创建 P2P 回调对象
11.    P2pCreateGroupCallBack p2pCreateGroupCallBack = new P2pCreateGroupCallBack();
12.    // 创建 P2P 群组
13.    wifiP2pController.createGroup(wifiP2pConfig, p2pCreateGroupCallBack);
14.    // 移除 P2P 群组
15.    wifiP2pController.removeGroup(p2pCreateGroupCallBack);
16. } catch (RemoteException re) {
17.     HiLog.warn(LABEL, "exception happened.");
18. }
19.
20. private class P2pCreateGroupCallBack extends WifiP2pCallback {
21.     @Override
22.     public void eventExecFail(int reason) {
23.         HiLog.info(LABEL, "CreateGroup eventExecFail reason : %{public}d", reason);
24.     }
25.
26.     @Override
```

```
27.     public void eventExecOk() {
28.         HiLog.info(LABEL, "CreateGroup eventExecOk");
29.     }
30. }
```

#### 6.3.3.5 发起 P2P 连接的开发步骤

1. 调用 WifiP2pController 的 getInstance(Context context)接口，获取 P2P 控制器实例，用于管理 P2P 操作。
2. 调用 init(EventRunner eventRunner, WifiP2pCallback callback)初始化 P2P 控制器实例。
3. 调用 requestP2pInfo()查询 P2P 可用设备信息。
4. 根据场景不同，从可用设备信息中选择目标设备。
5. 调用 connect 接口发起连接。

```
1. try {
2.     // 获取 P2P 管理对象
3.     WifiP2pController wifiP2pController = WifiP2pController.getInstance(this);
4.     // 初始化 P2P 管理对象，用于建立 P2P 信使等行为
5.     wifiP2pController.init(EventRunner.create(true), null);
6.     // 查询可用 P2P 设备信息，通过回调获取 P2P 设备信息
7.     P2pRequestPeersCallBack p2pRequestPeersCallBack = new P2pRequestPeersCallBack();
8.     wifiP2pController.requestP2pInfo(WifiP2pController.DEVICE_LIST_REQUEST, p2pRequestPeersCallBack);
9. } catch (RemoteException re) {
10.     HiLog.warn(LABEL, "exception happened.");
11. }
12.
13. private class P2pRequestPeersCallBack extends WifiP2pCallback {
14.     @Override
15.     public void eventP2pDevicesList(List<WifiP2pDevice> devices) {
16.         HiLog.info(LABEL, "eventP2pDevicesList when start connect group");
17.         // 根据场景不同，选择不同的设备进行连接，这里，通过 MAC 地址搜索到指定设备
18.         WifiP2pConfig wifiP2pConfig = getSameP2pConfigFromDevices(devices);
19.         try {
20.             if (wifiP2pConfig != null) {
21.                 // 向指定的设备发起连接
22.                 wifiP2pController.connect(wifiP2pConfig, null);
23.                 }
24.         } catch (RemoteException re) {
25.             HiLog.warn(LABEL, "exception happened in connect.");
```

```
26.             }
27.         }
28.     }
29.
30.     private WifiP2pConfig getSameP2pConfigFromDevices(List<WifiP2pDevice> devices) {
31.         if (devices == null) {
32.             return null;
33.         }
34.         for (int i = 0; i < devices.size(); i++) {
35.             WifiP2pDevice p2pDevice = devices.get(i);
36.             HiLog.info(LABEL, "p2pDevice.getDeviceAddress() : %{private}s", p2pDevice.getDeviceAddress());
37.             if (p2pDevice.getDeviceAddress() != null
38.                     && p2pDevice.getDeviceAddress().equals(TARGET_P2P_MAC_ADDRESS)) {
39.                 HiLog.info(LABEL, "received same mac address");
40.                 WifiP2pConfig wifiP2pConfig = new WifiP2pConfig("DEFAULT_GROUP_NAME", "DEFAULT_PASSPHRASE");
41.                 wifiP2pConfig.setDeviceAddress(p2pDevice.getDeviceAddress());
42.                 return wifiP2pConfig;
43.             }
44.         }
45.         return null;
46.     }
```

### 6.3.3.6 请求 P2P 相关信息的开发步骤

1. 调用 WifiP2pController 的 getInstance()接口，获取 P2P 控制器实例，用于管理 P2P 操作。
2. 调用 init()初始化 P2P 控制器实例。
3. 调用 requestP2pInfo()查询 P2P 群组信息。
4. 调用 requestP2pInfo()查询 P2P 设备信息。
5. 根据场景不同，可以调用 requestP2pInfo 获取需要的信息。

```
1. try {
2.     // 获取 P2P 管理对象
3.     WifiP2pController wifiP2pController = WifiP2pController.getInstance(this);
4.     // 初始化 P2P 管理对象，用于建立 P2P 信使等行为
5.     wifiP2pController.init(EventRunner.create(true), null);
6.     // 查询可用 P2P 群组信息，通过回调获取 P2P 群组信息
7.     P2pRequestGroupInfoCallBack p2pRequestGroupInfoCallBack = new P2pRequestGroupInfoCallBack();
8.     wifiP2pController.requestP2pInfo(WifiP2pController.GROUP_INFO_REQUEST, p2pRequestGroupInfoCallBack);
```

```
9.         // 查询可用 P2P 设备信息，通过回调获取 P2P 设备信息
10.        P2pRequestDeviceInfoCallBack p2pRequestDeviceInfoCallBack = new P2pRequestDeviceInfoCallBack();
11.        wifiP2pController.requestP2pInfo(WifiP2pController.DEVICE_INFO_REQUEST, p2pRequestDeviceInfoCallBack);
12.        // 通过调用 requestP2pInfo 接口，可以查询以下关键信息
13.        wifiP2pController.requestP2pInfo(WifiP2pController.NETWORK_INFO_REQUEST, callback); // 网络信息
14.        wifiP2pController.requestP2pInfo(WifiP2pController.DEVICE_LIST_REQUEST, callback); // 设备列表信息
15.    } catch (RemoteException re) {
16.        HiLog.warn(LABEL, "exception happened.");
17.    }
18.
19.    // 群组信息回调
20.    private class P2pRequestGroupInfoCallBack extends WifiP2pCallback {
21.        @Override
22.        public void eventP2pGroup(WifiP2pGroup group) {
23.            HiLog.info(LABEL, "P2pRequestGroupInfoCallBack eventP2pGroup");
24.            doSthFor(group);
25.        }
26.    }
27.    // 设备信息回调
28.    private class P2pRequestDeviceInfoCallBack extends WifiP2pCallback {
29.        @Override
30.        public void eventP2pGroup(WifiP2pDevice p2pDevice) {
31.            HiLog.info(LABEL, "eventP2pGroup");
32.                doSthFor(p2pDevice);
33.        }
34.    }
```

### 6.3.4 WLAN 消息通知

#### 6.3.4.1 场景介绍

WLAN 消息通知（Notification）是 HarmonyOS 内部或者与应用之间跨进程通讯的机制，注册者在注册消息通知后，一旦符合条件的消息被发出，注册者即可接收到该消息并获取消息中附带的信息。

### 6.3.4.2 接口说明

| 描述 | 通知名 | 附加参数 |
| --- | --- | --- |
| WLAN 状态 | usual.event.wifi.POWER_STATE | active_state |
| WLAN 扫描 | usual.event.wifi.SCAN_FINISHED | scan_state |
| WLAN RSSI 变化 | usual.event.wifi.RSSI_VALUE | rssi_value |
| WLAN 连接状态 | usual.event.wifi.CONN_STATE | conn_state |
| Hotspot 状态 | usual.event.wifi.HOTSPOT_STATE | hotspot_active_state |
| Hotspot 连接状态 | usual.event.wifi.WIFI_HS_STA_JOIN usual.event.wifi.WIFI_HS_STA_LEAVE | - |
| P2P 状态 | usual.event.wifi.p2p.STATE_CHANGE | p2p_state |
| P2P 连接状态 | usual.event.wifi.p2p.CONN_STATE_CHANGE | linked_info<br>net_info<br>group_info |
| P2P 设备列表变化 | usual.event.wifi.p2p.PEERS_STATE_CHANGE | peers_list |
| P2P 搜索状态变化 | usual.event.wifi.p2p.PEER_DISCOVERY_STATE_CHANGE | peers_discovery |
| P2P 当前设备变化 | usual.event.wifi.p2p.CURRENT_DEVICE_CHANGE | p2p_device |
| P2P 群组信息变化 | usual.event.wifi.p2p.GROUP_STATE_CHANGED | - |

**表 1** WLAN 消息通知的相关广播介绍

### 6.3.4.3 开发步骤

1. 构建消息通知接收者 WifiEventSubscriber。
2. 注册 WLAN 变化消息。

3. WifiEventSubscriber 接收并处理 WLAN 广播消息。

```
1.  // 构建消息接收者/注册者
2.  class WifiEventSubscriber extends CommonEventSubscriber {
3.      WifiEventSubscriber (CommonEventSubscribeInfo info) {
4.          super(info);
5.      }
6.
7.      @Override
8.      public void onReceiveEvent(CommonEventData commonEventData) {
9.          if (WifiEvents.EVENT_ACTIVE_STATE.equals(commonEventData.getIntent().getAction())) {
10.             // 获取附带参数
11.             IntentParams params = commonEventData.getIntent().getParams();
12.             if (params == null) {
13.                 return;
14.             }
15.             int wifiState= (int) params.getParam(WifiEvents.PARAM_ACTIVE_STATE);
16.
17.             if (wifiState== WifiEvents.STATE_ACTIVE) { // 处理 WLAN 被打开消息
18.                 HiLog.info(LABEL, false, "Receive WifiEvents.STATE_ACTIVE %{public}d", wifiState);
19.             } else if (wifiState == WifiEvents.STATE_INACTIVE) { // 处理 WLAN 被关闭消息
20.                 HiLog.info(LABEL, false, "Receive WifiEvents.STATE_INACTIVE %{public}d", wifiState);
21.             } else { // 处理 WLAN 异常状态
22.                 HiLog.info(LABEL, false, "Unknown wifi state");
23.             }
24.         }
25.     }
26. }
27.
28. // 注册消息
29. MatchingSkills match = new MatchingSkills();
30. // 增加获取 WLAN 状态变化消息
31. filter.addEvent(WifiEvents.EVENT_ACTIVE_STATE);
32. CommonEventSubscribeInfo subscribeInfo = new CommonEventSubscribeInfo(match);
33. subscribeInfo.setPriority(100);
34. WifiEventSubscriber subscriber = new WifiEventSubscriber(subscribeInfo);
35.
36. try {
37.     CommonEventManager.subscribeCommonEvent(subscriber);
```

```
38.  } catch (RemoteException e) {
39.      HiLog.warn(LABEL, false, "subscribe in wifi events failed!");
40.  }
```

# 6.4 网络管理

## 6.4.1 概述

HarmonyOS 网络管理模块主要提供以下功能：

- 数据连接管理：网卡绑定，打开 URL，数据链路参数查询。
- 数据网络管理：指定数据网络传输，获取数据网络状态变更，数据网络状态查询。
- 流量统计：获取蜂窝网络、所有网卡、指定应用或指定网卡的数据流量统计值。
- HTTP 缓存：有效管理 HTTP 缓存，减少数据流量。

### 6.4.1.1 约束与限制

使用网络管理模块的相关功能时，需要请求相应的权限。

| 权限名 | 权限描述 |
| --- | --- |
| ohos.permission.GET_NETWORK_INFO | 获取网络连接信息。 |
| ohos.permission.SET_NETWORK_INFO | 修改网络连接状态。 |
| ohos.permission.INTERNET | 允许程序打开网络套接字，进行网络连接。 |

## 6.4.2 使用当前网络打开一个 URL 链接

### 6.4.2.1 场景介绍

应用使用当前的数据网络打开一个 URL 链接。

#### 6.4.2.2 接口说明

应用使用当前网络打开一个 URL 链接，所使用的接口说明如下。

| 类名 | 接口名 | 功能描述 |
| --- | --- | --- |
| NetManager | getInstance(Context context) | 获取网络管理的实例对象。 |
| | hasDefaultNet() | 查询当前是否有默认可用的数据网络。 |
| | getDefaultNet() | 获取当前默认的数据网络句柄。 |
| | addDefaultNetStatusCallback(NetStatusCallback callback) | 获取当前默认的数据网络状态变化。 |
| | setAppNet(NetHandle netHandle) | 应用绑定该数据网络。 |
| NetHandle | openConnection(URL url, Proxy proxy) throws IOException | 使用该网络打开一个 URL 链接。 |

**表 1** 网络管理功能的主要接口

#### 6.4.2.3 开发步骤

1. 调用 NetManager.getInstance(Context)获取网络管理的实例对象。
2. 调用 NetManager.getDefaultNet()获取默认的数据网络。
3. 调用 NetHandle.openConnection()打开一个 URL。
4. 通过 URL 链接实例访问网站。

```
1.  NetManager netManager = NetManager.getInstance(null);
2.
3.  if (!netManager.hasDefaultNet()) {
4.      return;
5.  }
6.  NetHandle netHandle = netManager.getDefaultNet();
7.
8.  // 可以获取网络状态的变化
9.  NetStatusCallback callback = new NetStatusCallback() {
10.     // 重写需要获取的网络状态变化的 override 函数
```

```
11. }
12. netManager.addDefaultNetStatusCallback(callback);
13.
14. // 通过 openConnection 来获取 URLConnection
15. try {
16.     HttpURLConnection connection = null;
17.     String urlString = "https://www.huawei.com/";
18.     URL url = new URL(urlString);
19.
20.     URLConnection urlConnection = netHandle.openConnection(url,
21.                 java.net.Proxy.NO_PROXY);
22.     if (urlConnection instanceof HttpURLConnection) {
23.         connection = (HttpURLConnection) urlConnection;
24.     }
25.     connection.setRequestMethod("GET");
26.     connection.connect();
27.     // 之后可进行 url 的其他操作
28. } finally {
29.     connection.disconnect();
30. }
```

### 6.4.3 使用当前网络进行 Socket 数据传输

#### 6.4.3.1 场景介绍

应用使用当前的数据网络进行 Socket 数据传输。

#### 6.4.3.2 接口说明

应用使用当前网络进行 Socket 数据传输，所使用的接口说明如下。

| 类名 | 接口名 | 功能描述 |
| --- | --- | --- |
| NetManager | getByName(String host) | 解析主机名，获取其 IP 地址。 |

| 类名 | 接口名 | 功能描述 |
| --- | --- | --- |
| NetHandle | bindSocket(Socket socket) | 绑定 Socket 到该数据网络。 |
| | bindSocket(DatagramSocket socket) | 绑定 DatagramSocket 到该数据网络。 |

**表 1** 网络管理功能的主要接口

### 6.4.3.3 开发步骤

1. 调用 NetManager.getInstance(Context)获取网络管理的实例对象。
2. 调用 NetManager.getDefaultNet()获取默认的数据网络。
3. 调用 NetHandle.bindSocket()绑定网络。
4. 使用 socket 发送数据。

```
1.  NetManager netManager = NetManager.getInstance(null);
2.
3.  if (!netManager.hasDefaultNet()) {
4.      return;
5.  }
6.  NetHandle netHandle = netManager.getDefaultNet();
7.
8.  // 通过 Socket 绑定来进行数据传输
9.  try {
10.     InetAddress address = netHandle.getByName("www.huawei.com");
11.     DatagramSocket socket = new DatagramSocket();
12.     netHandle.bindSocket(socket);
13.     byte[] buffer = new byte[1024];
14.     DatagramPacket request = new DatagramPacket(buffer, buffer.length, address, port);
15.     // buffer 赋值
16.
17.     // 发送数据
18.     socket.send(request);
19. } catch(IOException e) {
20.     e.printStackTrace();
21. }
```

### 6.4.4 使用指定网络进行数据访问

#### 6.4.4.1 场景介绍

应用可以调用 API 接口来使用指定网络进行数据传输。在进行数据传输前，需要先建立自定义的网络类型。

#### 6.4.4.2 接口说明

应用使用指定网络进行数据访问，所使用的接口说明如下。

| 类名 | 接口名 | 功能描述 |
|---|---|---|
| NetSpecifier | Builder() | 创建一个指定网络实例。 |
| NetManager | setupSpecificNet(NetSpecifier netSpecifier, NetStatusCallback callback) | 建立指定的数据网络。 |
| | removeNetStatusCallback(NetStatusCallback callback) | 停止获取数据网络状态。 |

**表 1** 网络管理功能的主要接口

#### 6.4.4.3 开发步骤

1. 调用 NetSpecifier.Builder()构建指定数据网络的实例。
2. 调用 NetManager.setupSpecificNet()建立数据网络，通过 callback 获取网络状态变化。
3. 进行数据发送。

```
1.  NetManager netManager = NetManager.getInstance(null);
2.
3.  private class MmsCallback extends NetStatusCallback {
4.      @Override
5.      public void onAvailable(NetHandle netHandle) {
6.          // 通过 setAppNet 把后续应用所有的请求都通过该网络进行发送
7.          netManager.setAppNet(netHandle);
8.
```

```
9.          try {
10.             HttpURLConnection connection = null;
11.             String urlString = "https://www.huawei.com/";
12.             URL url = new URL(urlString);
13.             URLConnection urlConnection = netHandle.openConnection(url, java.net.Proxy.NO_PROXY);
14.             if (urlConnection instanceof HttpURLConnection) {
15.                 connection = (HttpURLConnection) urlConnection;
16.             }
17.             connection.setRequestMethod("GET");
18.             connection.connect();
19.             // 之后可进行 url 的其他操作
20.         } finally {
21.             connection.disconnect();
22.         }
23.
24.         // 如果业务执行完毕，可以停止获取
25.         netManager.removeNetStatusCallback(this);
26.     }
27. }
28.
29. MmsCallback callback = new MmsCallback();
30.
31. // 配置一个彩信类型的蜂窝网络
32. NetSpecifier req = new NetSpecifier.Builder()
33.     .addCapability(NetCapabilities.NET_CAPABILITY_MMS)
34.     .addBearer(NetCapabilities.BEARER_CELLULAR)
35.     .build();
36.
37. // 建立数据网络，通过 callback 获取网络变更状态
38. netManager.setupSpecificNet(req, callback);
```

### 6.4.5 流量统计

#### 6.4.5.1 场景介绍

应用通过调用 API 接口，可以获取蜂窝网络、所有网卡、指定应用或指定网卡的数据流量统计值。

#### 6.4.5.2 接口说明

应用进行流量统计，所使用的接口主要由 DataFlowStatistics 提供。

| 接口名 | 功能描述 |
| --- | --- |
| getCellularRxBytes() | 获取蜂窝数据网络的下行流量。 |
| getCellularTxBytes() | 获取蜂窝数据网络的上行流量。 |
| getAllRxBytes() | 获取所有网卡的下行流量。 |
| getAllTxBytes() | 获取所有网卡的上行流量。 |
| getUidRxBytes(int uid) | 获取指定 UID 的下行流量。 |
| getUidTxBytes(int uid) | 获取指定 UID 的上行流量。 |
| getIfaceRxBytes(String nic) | 获取指定网卡的下行流量。 |
| getIfaceTxBytes(String nic) | 获取指定网卡的上行流量。 |

**表 1** DataFlowStatistics 的主要接口

#### 6.4.5.3 开发步骤

调用 DataFlowStatistics 的接口可进行流量统计，以统计指定应用进程的流量为例。

```
1.  long rx = DataFlowStatistics.getUidRxBytes(uid);
2.  long tx = DataFlowStatistics.getUidTxBytes(uid);
3.
4.  // 进行数据收发
5.
6.  // 统计流量
7.  rx = DataFlowStatistics.getUidRxBytes(uid) - rx;
8.  tx = DataFlowStatistics.getUidTxBytes(uid) - tx;
```

### 6.4.6 管理 HTTP 缓存

#### 6.4.6.1 场景介绍

应用重复打开一个相同网页时，可以优先从缓存文件里读取内容，从而减少数据流量，降低设备功耗，提升应用性能。

#### 6.4.6.2 接口说明

管理 HTTP 缓存的功能主要由 HttpResponseCache 类提供。

| 接口名 | 功能描述 |
| --- | --- |
| install(File directory, long size) | 使能 HTTP 缓存，设置缓存保存目录及大小。 |
| getInstalled() | 获取缓存实例。 |
| flush() | 立即保存缓存信息到文件系统中。 |
| close() | 关闭缓存功能。 |
| delete() | 关闭并清除缓存内容。 |

**表 1** HttpResponseCache 的主要接口

#### 6.4.6.3 开发步骤

1. 配置缓存目录及最大缓存空间。
2. 保存缓存。
3. 关闭缓存。

```
// 初始化时设置缓存目录 dir 及最大缓存空间
HttpResponseCache.install(dir, 10 * 1024 * 1024);

// 访问 URL

// 为确保缓存保存到文件系统可以执行 flush 操作
HttpResponseCache.getInstalled().flush();

// 结束时关闭缓存
HttpResponseCache.getInstalled().close();
```

## 6.5 电话服务

### 6.5.1 概述

电话服务系统，除了为用户提供拨打语音/视频呼叫以及发送标准短信的功能以外，还提供了一系列的 API 用于获取无线蜂窝网络和 SIM 卡相关的一些信息。

其中拨打电话相关功能由 DistributedCallManager 提供，短信服务能力由 ShortMessageManager 提供。

应用还可以通过调用 RadioInfoManager 中的 API，来获取当前注册网络名称、网络服务状态以及信号强度等信息；以及调用 SimInfoManager 中的 API，来获取 SIM 卡的相关信息。

#### 6.5.1.1 约束与限制

1. 部分 API 接口需要一定访问权限才能调用，因此三方应用在调用有权限控制的 API 时，需要先申请对应权限，权限申请详见权限章节。
2. 语音呼叫和短信功能暂不支持传入卡槽编号(SlotId)，双卡场景下的使用规则详见发起一路呼叫和发送一条文本信息的场景介绍。
3. 注册获取 SIM 卡状态接口仅针对有 SIM 卡在位场景生效，若用户拔出 SIM 卡，则接收不到回调事件。应用可通过调用 hasSimCard 接口来确定当前卡槽是否有卡在位。

### 6.5.2 发起一路呼叫

#### 6.5.2.1 场景介绍

当应用需要发起一路呼叫给一个指定的号码时，使用本业务。呼叫可以是音频呼叫，也可以是视频呼叫。

如果设备支持同时插入两张 SIM 卡，且拨打电话时两张 SIM 卡均在位，呼叫时会弹出弹框让用户选择从卡 1 还是卡 2 呼出。

#### 6.5.2.2 接口说明

DistributedCallManager 为开发者提供呼叫管理功能，具体功能分类如下表。

| 功能分类 | 接口名 | 描述 | 所需权限 |
| --- | --- | --- | --- |
| 能力获取 | hasVoiceCapability() | 检查当前设备是否支持语音呼叫。 | 无 |
| 获取管理对象 | getInstance(Context context) | 获取呼叫管理对象。 | 无 |

| 功能分类 | 接口名 | 描述 | 所需权限 |
| --- | --- | --- | --- |
| 发起呼叫 | dial(String number, boolean isVideoCall) | 发起音频或视频呼叫。 | ohos.permission.PLACE_CALL |
| 观察通话业务状态变化 | addObserver(CallStateObserver observer, int mask) | 观察通话业务状态变化。 | ohos.permission.READ_CALL_LOG（获取通话号码需要该权限） |

**表 1** DistributedCallManager 的主要接口

### 6.5.2.3 开发步骤

1. 调用 DistributedCallManager 的 getInstance 接口，创建/获取呼叫管理对象。
2. 调用 hasVoiceCapability()接口获取当前设备呼叫能力，如果支持继续下一步；如果不支持则无法发起呼叫。
3. 发起一路呼叫。
4. 注册观察呼叫状态变化。

```
1.  // 创建呼叫管理对象
2.  DistributedCallManager dcManager = DistributedCallManager.getInstance(context);
3.
4.  // 调用查询能力接口
5.  if (!dcManager.hasVoiceCapability()) {
6.      return;
7.  }
8.
9.  // 如果设备支持呼叫能力，则继续发起呼叫
10. dcManager.dial(destinationNum, isVideoCall);
11.
12. // 创建继承 CallStateObserver 的类 MyCallStateObserver
13. class MyCallStateObserver extends CallStateObserver {
14.     // 构造方法，在当前线程的 runner 中执行回调，slotId 需要传入要观察的卡槽 ID（0 或 1）
15.     MyCallStateObserver(int slotId) {
16.         super(slotId);
17.     }
18.
19.     // 构造方法，在执行 runner 中执行回调，slotId 需要传入要观察的卡槽 ID（0 或 1）
20.     MyCallStateObserver(int slotId, EventRunner runner) {
```

```
21.             super(slotId, runner);
22.         }
23.
24.         // 通话状态变化的回调方法
25.         @Override
26.         public void onCallStateUpdated(int state, String number) {
27.             ...
28.         }
29.     }
30.
31.     // 执行回调的 runner
32.     EventRunner runner = EventRunner.create();
33.
34.     // 创建 MyCallStateObserver 的对象
35.     MyCallStateObserver observer = new MyCallStateObserver(slotId, runner);
36.
37.     // 观察 OBSERVE_CALL_STATE 的变化
38.     dcManager.addObserver(observer, CallStateObserver.OBSERVE_CALL_STATE);
```

### 6.5.3 发送一条文本信息

#### 6.5.3.1 场景介绍

应用需要发送一条短信给一个指定的号码时，使用本业务。发送信息需要经过短信中心，短信中心号码可以是运营商默认的，也可以由应用自己指定。

如果设备支持同时插入 2 张 SIM 卡，且 2 张 SIM 卡均在位时，短信会从默认 SIM 卡发出。应用可通过调用 getDefaultSmsSlotId 来获取当前发短信的默认 SIM 卡位置。目前 API 暂不支持短信发送结果通知和送达报告。

#### 6.5.3.2 接口说明

ShortMessageManager 为开发者提供短信管理功能，具体功能分类如下表。

| 功能分类 | 接口名 | 描述 | 所需权限 |
|---|---|---|---|
| 能力获取 | hasSmsCapability() | 检查当前设备是否支持短信收发。 | 无 |
| 获取管理对象 | getInstance(Context context) | 获取短信管理对象。 | 无 |
| 获取默认短信卡 | getDefaultSmsSlotId() | 获取默认短信卡对应卡槽 ID。 | 无 |
| 长短信转化 | splitMessage(String content) | 将超过 140 个字节的长短信（如中文 70 个字符，英文 160 个字符）拆分成多条短信。 | ohos.permission.SEND_MESSAGES |
| 发送短信 | sendMessage(String destinationHost, String serviceCenter, String content) | 发送单条短信。 | ohos.permission.SEND_MESSAGES |
| | sendMultipartMessage(String destinationHost, String serviceCenter, ArrayList<String> parts) | 发送拆分后的多条短信。 | ohos.permission.SEND_MESSAGES |

**表 1** ShortMessageManager 的主要接口

### 6.5.3.3 开发步骤

1. 调用 ShortMessageManager 的 getInstance 接口，创建/获取短信收发管理对象。
2. 调用 hasSmsCapability()接口获取当前设备短信收发能力，如果支持继续下一步；如果不支持则无法收发短信。
3. 发送短信。

```
// 创建短信收发管理对象
ShortMessageManager smManager = ShortMessageManager.getInstance(context);

// 检查短信能力
if (!smManager.hasSmsCapability()) {
    return;
}
```

```
8.
9.  // 如果设备支持收发短信，则继续发送短信
10. // 发送短信前可先调用 splitMessage()接口判断拆分后的短信条数，然后决定调用长短信或普通短信发送接口
11. ArrayList<String> msgs = smManager.splitMessage(messageContent);
12. if (msgs.size() > 1) { // 长短信拆分发送
13.     smManager.sendMultipartMessage(destinationNumber, serviceCenter, msgs);
14. } else { // 一般文本短信发送
15.     smManager.sendMessage(destinationNumber, serviceCenter, messageContent);
16. }
```

### 6.5.4 获取当前蜂窝网络信号信息

#### 6.5.4.1 场景介绍

应用通常需要获取用户所在蜂窝网络下信号信息，以便获取当前驻网质量。开发者可以通过本业务，获取到用户指定 SIM 卡当前所在网络下的信号信息。

#### 6.5.4.2 接口说明

RadioInfoManager 类中提供了获取当前网络信号信息列表的方法。

| 功能分类 | 接口名 | 描述 | 所需权限 |
| --- | --- | --- | --- |
| 获取管理对象 | getInstance(Context context) | 获取网络管理对象。 | 无 |
| 信号强度信息 | getSignalInfoList(int slotId) | 获取当前注册蜂窝网络信号强度信息。 | 无 |

**表 1** RadioInfoManager 的主要接口

### 6.5.5 开发步骤

1. 调用 RadioInfoManager 的 getInstance 接口，获取到 RadioInfoManager 实例。
2. 调用 getSignalInfoList(slotId)方法，返回所有 SignalInformation 列表。
3. 遍历 SignalInformation 列表，并分别根据 signalNetworkType 转换为对应制式的 SignalInformation 子类对象。

4. 调用子类中的方法，获取信号强度信息。

```
1.  // 获取 RadioInfoManager 对象。
2.  RadioInfoManager radioInfoManager = RadioInfoManager.getInstance(context);
3.
4.  // 获取信号信息。
5.  List<SignalInformation> signalList = radioInfoManager.getSignalInfoList(slotId);
6.
7.  // 检查信号信息列表大小。
8.  if (signalList.size() == 0) {
9.      return;
10. }
11. // 依次遍历 list 获取当前驻网 networkType 对应的信号信息。
12. LteSignalInformation lteSignal;
13. for (SignalInformation signal : signalList) {
14.     int signalNetworkType = signal.getSignalNetworkType();
15.     if (signalNetworkType == TelephonyConstants.NETWORK_TYPE_LTE) {
16.         lteSignal = (LteSignalInformation) signal;
17.     }
18. }
19. // 调用子类中相应方法，获取对应制式的信号强度信息。
20. int signalLevel = lteSignal.getSignalLevel();
```

### 6.5.6 观察蜂窝网络状态变化

#### 6.5.6.1 场景介绍

应用可以通过观察蜂窝网络状态变化，来接收最新蜂窝网络服务状态信息、信号信息等。

#### 6.5.6.2 接口说明

RadioStateObserver 类中提供了观察蜂窝网络状态变化的方法，为了能够实时观察蜂窝网络状态变化，应用必须包含以下权限。

| 观察状态名称 | 权限名称 |
| --- | --- |
| 网络状态信息(NETWORK_STATE) | ohos.permission.GET_NETWORK_INFO |
| 信号信息(SIGNAL_INFO) | NA |

**表 1** 观察蜂窝网络状态变化需要的权限

需要使用 RadioInfoManager 的如下接口将继承 RadioStateObserver 类的对象注册到系统服务：

| 接口名 | 观察事件的掩码 |
| --- | --- |
| addObserver | OBSERVE_MASK_NETWORK_STATE |
| | OBSERVE_MASK_SIGNAL_INFO |
| removeObserver | N/A |

**表 2** 添加观察和停止观察接口 API 介绍

### 6.5.6.3 开发步骤

#### 添加观察事件

1. 调用 RadioInfoManager 的 getInstance 接口，获取到 RadioInfoManager 实例。
2. 创建继承 RadioStateObserver 的类 MyRadioStateObserver，并覆写状态变化回调方法。
3. 创建 MyRadioStateObserver 的对象 observer。
4. 调用 RadioInfoManager 的 addObserver 方法，传入已创建的 MyRadioStateObserver 对象 observer 和需要观察的 mask。

```
1. // 获取 RadioInfoManager 对象。
2. RadioInfoManager radioInfoManager = RadioInfoManager.getInstance(context);
3. // 创建继承 RadioStateObserver 的类 MyRadioStateObserver
```

```
4.  class MyRadioStateObserver extends RadioStateObserver {
5.      // 构造方法，在当前线程的 runner 中执行回调，slotId 需要传入要观察的卡槽 ID（0 或 1）。
6.      MyRadioStateObserver(int slotId) {
7.          super(slotId);
8.      }
9.
10.     // 构造方法，在执行 runner 中执行回调。
11.     MyRadioStateObserver(int slotId, EventRunner runner) {
12.         super(slotId, runner);
13.     }
14.
15.     // 网络注册状态变化的回调方法。
16.     @Override
17.     public void onNetworkStateUpdated(NetworkState state) {
18.         ...
19.     }
20.
21.     // 信号信息变化的回调方法。
22.     @Override
23.     public void onSignalInfoUpdated(List<SignalInformation> signalInfos) {
24.         ...
25.     }
26. }
27.
28. // 执行回调的 runner。
29. EventRunner runner = EventRunner.create();
30.
31. // 创建 MyRadioStateObserver 的对象。
32. MyRadioStateObserver observer = new MyRadioStateObserver(slotId, runner);
33.
34. // 添加回调，以 NETWORK_STATE 和 SIGNAL_INFO 为例。
35. radioInfoManager.addObserver(observer, RadioStateObserver.OBSERVE_MASK_NETWORK_STATE |
    RadioStateObserver.OBSERVE_MASK_SIGNAL_INFO);
```

### 停止观察

1. 调用 RadioInfoManager 的 getInstance 接口，获取到 RadioInfoManager 实例。
2. 调用 RadioInfoManager 的 removeObserver 方法，传入添加观察事件时创建的 MyRadioStateObserver 对象 observer。

```
1.  // 获取 RadioInfoManager 对象。
```

```
2. RadioInfoManager radioInfoManager = RadioInfoManager.getInstance(context);
3. // 停止观察
4. radioInfoManager.removeObserver(observer);
```

# 7 设备管理

## 7.1 传感器

### 7.1.1 概述

#### 7.1.1.1 基本概念

HarmonyOS 传感器是应用访问底层硬件传感器的一种设备抽象概念。开发者根据传感器提供的 Sensor API，可以查询设备上的传感器，订阅传感器的数据，并根据传感器数据定制相应的算法，开发各类应用，比如指南针、运动健康、游戏等。

根据传感器的用途，可以将传感器分为六大类：运动类传感器、环境类传感器、方向类传感器、光线类传感器、健康类传感器、其他类传感器（如霍尔传感器），每一大类传感器包含不同类型的传感器，某种类型的传感器可能是单一的物理传感器，也可能是由多个物理传感器复合而成。传感器列表如图 1 所示。

| 分类 | API 类名 | 传感器类型 | 中文描述 | 说明 | 主要用途 |
| --- | --- | --- | --- | --- | --- |
| 运动类 | ohos.sensor.agent.CategoryMotionAgent | SENSOR_TYPE_ACCELEROMETER | 加速度传感器 | 测量三个物理轴（x、y 和 z）上，施加在设备上的加速度（包括重力加速度），单位 : $m/s^2$ | 检测运动状态 |

| 分类 | API 类名 | 传感器类型 | 中文描述 | 说明 | 主要用途 |
| --- | --- | --- | --- | --- | --- |
| | | SENSOR_TYPE_ACCELEROMETER_UNCALIBRATED | 未校准加速度传感器 | 测量三个物理轴（x、y 和 z）上，施加在设备上的未校准的加速度（包括重力加速度），单位：m/s² | 检测加速度偏差估值 |
| | | SENSOR_TYPE_LINEAR_ACCELERATION | 线性加速度传感器 | 测量三个物理轴（x、y 和 z）上，施加在设备上的线性加速度（不包括重力加速度），单位：m/s² | 检测每个单轴方向上的线性加速度 |
| | | SENSOR_TYPE_GRAVITY | 重力传感器 | 测量三个物理轴（x、y 和 z）上，施加在设备上的重力加速度，单位：m/s² | 测量重力大小 |
| | | SENSOR_TYPE_GYROSCOPE | 陀螺仪传感器 | 测量三个物理轴（x、y 和 z）上，设备的旋转角速度，单位：rad/s | 测量旋转的角速度 |
| | | SENSOR_TYPE_GYROSCOPE_UNCALIBRATED | 未校准陀螺仪传感器 | 测量三个物理轴（x、y 和 z）上，设备的未校准旋转角速度，单位：rad/s | 测量旋转的角速度及偏差估值 |

| 分类 | API 类名 | 传感器类型 | 中文描述 | 说明 | 主要用途 |
|---|---|---|---|---|---|
| | | SENSOR_TYPE_SIGNIFICANT_MOTION | 大幅度动作传感器 | 测量三个物理轴（x、y 和 z）上，设备是否存在大幅度运动；如果取值为 1 则代表存在大幅度运动，取值为 0 则代表没有大幅度运动 | 用于检测设备是否存在大幅度运动 |
| | | SENSOR_TYPE_DROP_DETECTION | 跌落检测传感器 | 检测设备的跌落状态；如果取值为 1 则代表发生跌落，取值为 0 则代表没有发生跌落 | 用于检测设备是否发生了跌落 |
| | | SENSOR_TYPE_PEDOMETER_DETECTION | 计步器检测传感器 | 检测用户的计步动作；如果取值为 1 则代表用户产生了计步行走的动作；取值为 0 则代表用户没有发生运动 | 用于检测用户是否有计步的动作 |
| | | SENSOR_TYPE_PEDOMETER | 计步器传感器 | 统计用户的行走步数 | 用于提供用户行走的步数数据 |

| 分类 | API 类名 | 传感器类型 | 中文描述 | 说明 | 主要用途 |
| --- | --- | --- | --- | --- | --- |
| 环境类 | ohos.sensor.agent.CategoryEnvironmentAgent | SENSOR_TYPE_AMBIENT_TEMPERATURE | 环境温度传感器 | 测量环境温度，单位：摄氏度（°C） | 测量环境温度 |
| | | SENSOR_TYPE_MAGNETIC_FIELD | 磁场传感器 | 测量三个物理轴向（x、y、z）上，环境地磁场，单位：μT | 创建指南针 |
| | | SENSOR_TYPE_MAGNETIC_FIELD_UNCALIBRATED | 未校准磁场传感器 | 测量三个物理轴向（x、y、z）上，未校准环境地磁场，单位：μT | 测量地磁偏差估值 |
| | | SENSOR_TYPE_HUMIDITY | 湿度传感器 | 测量环境的相对湿度，以百分比（%）表示 | 监测露点、绝对湿度和相对湿度 |
| | | SENSOR_TYPE_BAROMETER | 气压计传感器 | 测量环境气压，单位：hPa 或 mbar | 测量环境气压 |
| | | SENSOR_TYPE_SAR | 比吸收率传感器 | 测量比吸收率，单位：W/kg | 测量设备的电磁波能量吸收比值。 |

| 分类 | API 类名 | 传感器类型 | 中文描述 | 说明 | 主要用途 |
| --- | --- | --- | --- | --- | --- |
| 方向类 | ohos.sensor.agent.CategoryOrientationAgent | SENSOR_TYPE_6DOF | 6 自由度传感器 | 测量上下、前后、左右方向上的位移，单位：m 或 mm；测量俯仰、偏摆、翻滚的角度，单位：rad | 检测设备的三个平移自由度以及旋转自由度，用于目标定位追踪，如：VR |
| | | SENSOR_TYPE_SCREEN_ROTATION | 屏幕旋转传感器 | 检测设备屏幕的旋转状态 | 用于检测设备屏幕是否发生了旋转 |
| | | SENSOR_TYPE_DEVICE_ORIENTATION | 设备方向传感器 | 测量设备的旋转方向，单位：rad | 用于检测设备旋转方向的角度值 |

| 分类 | API 类名 | 传感器类型 | 中文描述 | 说明 | 主要用途 |
|---|---|---|---|---|---|
| | | SENSOR_TYPE_ORIENTATION | 方向传感器 | 测量设备围绕所有三个物理轴（x、y、z）旋转的角度值，单位：rad | 用于提供屏幕旋转的 3 个角度值 |
| | | SENSOR_TYPE_ROTATION_VECTOR | 旋转矢量传感器 | 测量设备旋转矢量，复合传感器：由加速度传感器、磁场传感器、陀螺仪传感器合成 | 检测设备相对于东北天坐标系的方向 |
| | | SENSOR_TYPE_GAME_ROTATION_VECTOR | 游戏旋转矢量传感器 | 测量设备游戏旋转矢量，复合传感器：由加速度传感器、陀螺仪传感器合成 | 应用于游戏场景 |
| | | SENSOR_TYPE_GEOMAGNETIC_ROTATION_VECTOR | 地磁旋转矢量传感器 | 测量设备地磁旋转矢量，复合传感器：由加速度传感器、磁场传感器合成 | 用于测量地磁旋转矢量 |
| 光线类 | ohos.sensor.agent.CategoryLightAgent | SENSOR_TYPE_PROXIMITY | 接近光传感器 | 测量可见物体相对于设备显示屏的接近或远离状态 | 通话中设备相对人的位置 |

| 分类 | API 类名 | 传感器类型 | 中文描述 | 说明 | 主要用途 |
|---|---|---|---|---|---|
| | | SENSOR_TYPE_TOF | ToF 传感器 | 测量光在介质中行进一段距离所需的时间 | 人脸识别 |
| | | SENSOR_TYPE_AMBIENT_LIGHT | 环境光传感器 | 测量设备周围光线强度，单位：lux | 自动调节屏幕亮度，检测屏幕上方是否有遮挡 |
| | | SENSOR_TYPE_COLOR_TEMPERATURE | 色温传感器 | 测量环境中的色温 | 应用于设备的影像处理 |
| | | SENSOR_TYPE_COLOR_RGB | RGB 颜色传感器 | 测量环境中的 RGB 颜色值 | 通过三原色的反射比率实现颜色检测 |
| | | SENSOR_TYPE_COLOR_XYZ | XYZ 颜色传感器 | 测量环境中的 XYZ 颜色值 | 用于辨识真色色 |

| 分类 | API 类名 | 传感器类型 | 中文描述 | 说明 | 主要用途 |
|---|---|---|---|---|---|
| | | | | | 点，还原色彩更真实 |
| 健康类 | ohos.sensor.agent.CategoryBodyAgent | SENSOR_TYPE_HEART_RATE | 心率传感器 | 测量用户的心率数值 | 用于提供用户的心率健康数据 |
| | | SENSOR_TYPE_WEAR_DETECTION | 佩戴检测传感器 | 检测用户是否佩戴 | 用于检测用户是否佩戴智能穿戴 |
| 其他类 | ohos.sensor.agent.CategoryOtherAgent | SENSOR_TYPE_HALL | 霍尔传感器 | 测量设备周围是否存在磁力吸引 | 设备的皮套模式 |
| | | SENSOR_TYPE_GRIP_DETECTOR | 手握检测传感器 | 检测设备是否有抓力施加 | 用于检查设备侧边是否被手握住 |

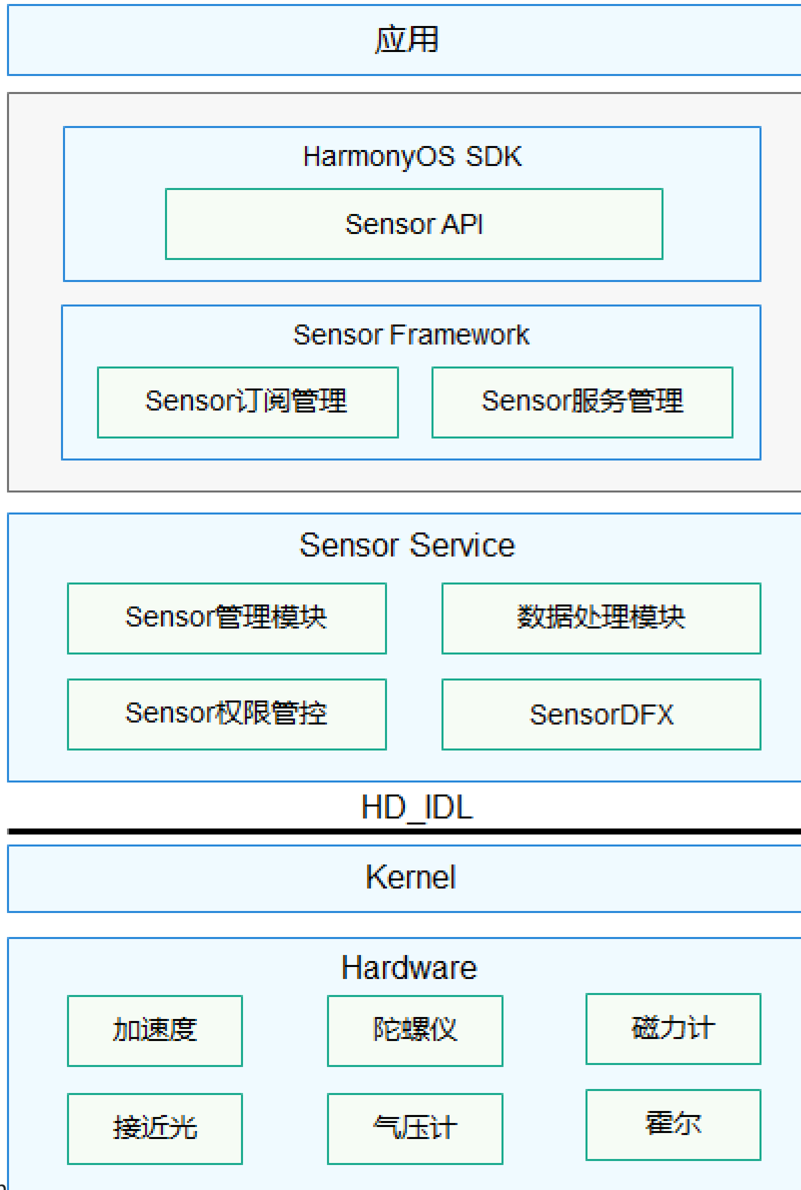
| 分类 | API 类名 | 传感器类型 | 中文描述 | 说明 | 主要用途 |
| --- | --- | --- | --- | --- | --- |
| | | SENSOR_TYPE_MAGNET_BRACKET | 磁铁支架传感器 | 检测设备是否被磁吸 | 检测设备是否位于车内或者室内 |
| | | SENSOR_TYPE_PRESSURE_DETECTOR | 按压检测传感器 | 检测设备是否有压力施加 | 用于检测设备的正上方是否存在按压 |

**表 1** 传感器列表

#### 7.1.1.2 运作机制

HarmonyOS 传感器包含如下四个模块：Sensor API、Sensor Framework、Sensor Service、HD_IDL 层。

**图 1** HarmonyOS 传感器




- Sen[illegible]sor API：提供传感器的基础 API，主要包含查询传感器的列表、订阅/取消传感器的数据、执行控制命令等，简化应用开发。
- Sensor Framework：主要实现传感器的订阅管理，数据通道的创建、销毁、订阅与取消订阅，实现与 SensorService 的通信。
- Sensor Service：主要实现 HD_IDL 层数据接收、解析、分发，前后台的策略管控，对该设备 Sensor 的管理；Sensor 权限管控等。
- HD_IDL 层：对不同的 FIFO、频率进行策略选择；以及对不同设备（车机、智能穿戴、智慧屏等）的适配。

#### 7.1.1.3 约束与限制

1. 针对某些传感器，开发者需要请求相应的权限，才能获取到相应传感器的数据。

| 传感器 | HarmonyOS 权限名 | 敏感级别 | 权限描述 |
|---|---|---|---|
| 加速度传感器、加速度未校准传感器、线性加速度传感器 | ohos.permission.ACCELEROMETER | system_grant | 允许订阅 Motion 组对应的加速度传感器的数据 |
| 陀螺仪传感器、陀螺仪未校准传感器 | ohos.permission.GYROSCOPE | system_grant | 允许订阅 Motion 组对应的陀螺仪传感器的数据 |
| 计步器 | ohos.permission.ACTIVITY_MOTION | user_grant | 允许订阅运动状态 |
| 心率 | ohos.permission.READ_HEALTH_DATA | user_grant | 允许读取健康数据 |

**表 2** HarmonyOS 传感器权限列表

2. 传感器数据订阅和取消订阅接口成对调用，当不再需要订阅传感器数据时，开发者需要调用取消订阅接口进行资源释放。

### 7.1.2 开发指导

#### 7.1.2.1 场景介绍

- 通过方向传感器数据，可以感知用户设备当前的朝向，从而达到为用户指明方位的目的。
- 通过重力和陀螺仪传感器数据，能感知设备倾斜和旋转量，提高用户在游戏场景中的体验。
- 通过接近光传感器数据，感知距离遮挡物的距离，使设备能够自动亮灭屏，达到防误触目的。
- 通过气压计传感器数据，可以准确的判断设备当前所处的海拔。
- 通过环境光传感器数据，设备能够实现背光自动调节。

#### 7.1.2.2 接口说明

HarmonyOS 传感器提供的功能有：查询传感器的列表、订阅/取消订阅传感器数据、查询传感器的最小采样时间间隔、执行控制命令。

以订阅方向类别的传感器数据为例，本节示例涉及的接口如下：

| 接口名 | 描述 |
| --- | --- |
| getAllSensors() | 获取属于方向类别的传感器列表。 |
| getAllSensors(int) | 获取属于方向类别中特定类型的传感器列表。 |
| getSingleSensor(int) | 查询方向类别中特定类型的默认 sensor（如果存在多个则返回第一个）。 |
| setSensorDataCallback(ICategoryOrientationDataCallback, CategoryOrientation, long) | 以设定的采样间隔订阅给定传感器的数据。 |
| setSensorDataCallback(ICategoryOrientationDataCallback, CategoryOrientation, long, long) | 以设定的采样间隔和时延订阅给定传感器的数据。 |
| releaseSensorDataCallback(ICategoryOrientationDataCallback, CategoryOrientation) | 取消订阅指定传感器的数据。 |
| releaseSensorDataCallback(ICategoryOrientationDataCallback) | 取消订阅的所有传感器数据。 |

表 1 CategoryOrientationAgent 的主要接口

| 接口名 | 描述 |
| --- | --- |
| getSensorMinSampleInterval(int) | 查询给定传感器的最小采样间隔。 |
| runCommand(int, int, int) | 针对某个传感器执行命令，刷新传感器的数据。 |

表 2 SensorAgent 的主要接口

### 7.1.2.3 开发步骤

**权限配置**

如果设备上使用了表 2 中的传感器，需要请求相应的权限，开发者才能获取到传感器数据。

| 敏感级别 | 传感器 | HarmonyOS 权限名 | 权限描述 |
| --- | --- | --- | --- |
| system_grant | 加速度传感器、加速度未校准传感器、线性加速度传感器 | ohos.permission.ACCELEROMETER | 允许订阅 Motion 组对应的加速度传感器的数据。 |
| user_grant | 计步器 | ohos.permission.ACTIVITY_MOTION | 允许订阅运动状态。 |

**表 3** 不同敏感级别的 HarmonyOS 传感器举例

开发者需要在 config.json 里面配置权限：

- 开发者如果需要获取加速度的数据，需要进行如下权限配置。

```
1.  "reqPermissions": [
2.      {
3.          "name": "ohos.permission.ACCELEROMETER",
4.          "reason": "",
5.          "usedScene": {
6.              "ability": [
7.                  ".MainAbility"
8.              ],
9.              "when": "inuse"
10.         }
11.     }
12. ]
```

- 对于需要用户授权的权限，如计步器传感器，需要进行如下权限配置。

```
1.  "reqPermissions": [
2.      {
3.          "name": "ohos.permission.ACTIVITY_MOTION",
4.          "reason": "",
5.          "usedScene": {
6.              "ability": [
7.                  ".MainAbility"
8.              ],
9.          "when": "inuse"
10.         }
11.     }
```

```
12. ]
```

由于敏感权限需要用户授权，因此，开发者在应用启动时或者调用订阅数据接口前，需要调用权限检查和请求权限接口。

```
1.  @Override
2.  public void onStart(Intent intent) {
3.      super.onStart(intent);
4.      if (verifySelfPermission("ohos.permission.ACTIVITY_MOTION") != 0) {
5.          if (canRequestPermission("ohos.permission.ACTIVITY_MOTION")) {
6.              requestPermissionsFromUser(new String[] {"ohos.permission.ACTIVITY_MOTION"}, 1);
7.          }
8.      }
9.      // ...
10. }
11.
12. @Override
13. public void onRequestPermissionsFromUserResult(int requestCode, String[] permissions,
14.         int[] grantResults) {
15.     switch (requestCode) {
16.         case 1: {
17.             // 匹配 requestPermissionsFromUser 的 requestCode
18.             if (grantResults.length > 0 && grantResults[0] == 0) {
19.                 // 权限被授予
20.             } else {
21.                 // 权限被拒绝
22.             }
23.             return;
24.         }
25.     }
26. }
```

**使用传感器**

以使用方向类别的传感器为例，运动类、环境类、健康类等类别的传感器使用方法类似。

1. 获取待订阅数据的传感器。
2. 创建传感器回调。
3. 订阅传感器数据。

4. 接收并处理传感器数据。
5. 取消订阅传感器数据。

```
1.  private Button btnSubscribe;
2.  
3.  private Button btnUnsubscribe;
4.  
5.  private CategoryOrientationAgent categoryOrientationAgent = new CategoryOrientationAgent();
6.  
7.  private ICategoryOrientationDataCallback orientationDataCallback;
8.  
9.  private CategoryOrientation orientationSensor;
10. 
11. private long interval = 100000000;
12. 
13. @Override
14. public void onStart(Intent intent) {
15.     super.onStart(intent);
16.     super.setUIContent(ResourceTable.Layout_sensor_layout);
17.     findComponent(rootComponent);
18. 
19.     // 创建传感器回调对象。
20.     orientationDataCallback = new ICategoryOrientationDataCallback() {
21.         @Override
22.         public void onSensorDataModified(CategoryOrientationData categoryOrientationData) {
23.             // 对接收的 categoryOrientationData 传感器数据对象解析和使用
24.             int dim = categoryOrientationData.getSensorDataDim(); //获取传感器的维度信息
25.             float degree = categoryOrientationData.getValues()[0]; // 获取方向类传感器的第一维数据
26.         }
27. 
28.         @Override
29.         public void onAccuracyDataModified(CategoryOrientation categoryOrientation, int i) {
30.             // 使用变化的精度
31.         }
32. 
33.         @Override
34.         public void onCommandCompleted(CategoryOrientation categoryOrientation) {
35.             // 传感器执行命令回调
36.         }
```

```
37.         };
38.
39.         btnSubscribe.setClickedListener(v -> {
40.             // 获取传感器对象，并订阅传感器数据
41.             orientationSensor = categoryOrientationAgent.getSingleSensor(
42.                     CategoryOrientation.SENSOR_TYPE_ORIENTATION);
43.             if (orientationSensor != null) {
44.                 categoryOrientationAgent.setSensorDataCallback(
45.                         orientationDataCallback, orientationSensor, interval);
46.             }
47.         });
48.         // 取消订阅传感器数据
49.         btnUnsubscribe.setClickedListener(v -> {
50.             if (orientationSensor != null) {
51.                 categoryOrientationAgent.releaseSensorDataCallback(
52.                         orientationDataCallback, orientationSensor);
53.             }
54.         });
55.     }
56.
57.     private void findComponent(Component component) {
58.         btnSubscribe = (Button) component.findComponentById(Resource.Id.btnSubscribe);
59.         btnUnsubscribe = (Button) component.findComponentById(Resource.Id.btnUnsubscribe);
60.     }
```

## 7.2 控制类小器件

### 7.2.1 概述

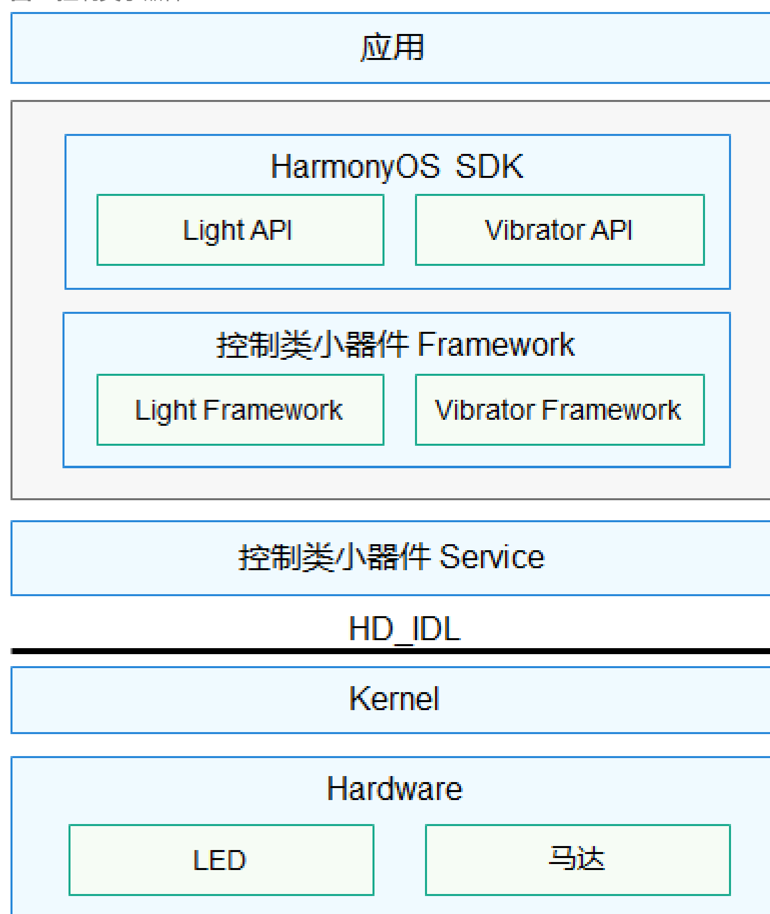
#### 7.2.1.1 基本概念

控制类小器件指的是设备上的 LED 灯和振动器。其中，LED 灯主要用作指示（如充电状态）、闪烁功能（如三色灯）等；振动器主要用于闹钟、开关机振动、来电振动等场景。

#### 7.2.1.2 运作机制

控制类小器件主要包含以下四个模块：控制类小器件 API、控制类小器件 Framework、控制类小器件 Service、HD_IDL 层。

图 1 控制类小器件




- 控制类小器件 API：提供灯和振动器基础的 API，主要包含灯的列表查询、打开灯、关闭灯等接口，振动器的列表查询、振动器的振动器效果查询、触发/关闭振动器等接口。
- 控制类小器件 Framework：主要实现灯和振动器的框架层管理，实现与控制类小器件 Service 的通信。
- 控制类小器件 Service：实现灯和振动器的服务管理。
- HD_IDL 层：对不同设备（车机、智能穿戴、智慧屏等）的适配。

#### 7.2.1.3 约束与限制

- 在调用 Light API 时，请先通过 getLightIdList 接口查询设备所支持的灯的 ID 列表，以免调用打开接口异常。

- 在调用 Vibrator API 时，请先通过 getVibratorIdList 接口查询设备所支持的振动器的 ID 列表，以免调用振动接口异常。
- 在使用振动器时，开发者需要配置请求振动器的权限 ohos.permission.VIBRATE，才能控制振动器振动。

### 7.2.2 Light 开发指导

#### 7.2.2.1 场景介绍

当设备需要设置不同的闪烁效果时，可以调用 Light 模块，例如，LED 灯能够设置灯颜色、灯亮和灯灭时长的闪烁效果。

**说明**

使用该功能依赖于硬件设备是否具有 LED 灯。

#### 7.2.2.2 接口说明

灯模块主要提供的功能有：查询设备上灯的列表，查询某个灯设备支持的效果，打开和关闭灯设备。LightAgent 类开放能力如下，具体请查阅 API 参考文档。

| 接口名 | 描述 |
| --- | --- |
| getLightIdList() | 获取硬件设备上的灯列表。 |
| isSupport(int) | 根据指定灯 Id 查询硬件设备是否有该灯。 |
| isEffectSupport(int, String) | 查询指定的灯是否支持指定的闪烁效果。 |
| turnOn(int, String) | 对指定的灯创建指定效果的一次性闪烁。 |
| turnOn(int, LightEffect) | 对指定的灯创建自定义效果的一次性闪烁。 |
| turnOn(String) | 对指定的灯创建指定效果的一次性闪烁。 |
| turnOn(LightEffect) | 对指定的灯创建自定义效果的一次性闪烁。 |

| 接口名 | 描述 |
| --- | --- |
| turnOff(int) | 关闭指定的灯。 |
| turnOff() | 关闭指定的灯。 |

表 1 LightAgent 的主要接口

### 7.2.2.3 开发步骤

1. 查询硬件设备上灯的列表。
2. 查询指定的灯是否支持指定的闪烁效果。
3. 创建不同的闪烁效果。
4. 关闭指定的灯。

```
1.  private LightAgent lightAgent = new LightAgent();
2.  
3.  @Override
4.  public void onStart(Intent intent) {
5.      super.onStart(intent);
6.      super.setUIContent(ResourceTable.Layout_light_layout);
7.  
8.      // ...
9.  
10.     // 查询硬件设备上的灯列表
11.     List<Integer> myLightList = lightAgent.getLightIdList();
12.     if (myLightList.isEmpty()) {
13.         return;
14.     }
15.     int lightId = myLightList.get(0);
16. 
17.     // 查询指定的灯是否支持指定的闪烁效果
18.     boolean isSupport = lightAgent.isEffectSupport(lightId, LightEffect.LIGHT_ID_KEYBOARD);
19. 
20.     // 创建指定效果的一次性闪烁
21.     boolean turnOnResult = lightAgent.turnOn(lightId, LightEffect.LIGHT_ID_KEYBOARD);
22. 
```

```
23.     // 创建自定义效果的一次性闪烁
24.     LightBrightness lightBrightness = new LightBrightness(255, 255, 255);
25.     LightEffect lightEffect = new LightEffect(lightBrightness, 1000, 1000);
26.     boolean turnOnEffectResult = lightAgent.turnOn(lightId, lightEffect);
27.
28.     // 关闭指定的灯
29.     boolean turnOffResult = lightAgent.turnOff(lightId);
30. }
```

## 7.2.3 Vibrator 开发指导

### 7.2.3.1 场景介绍

当设备需要设置不同的振动效果时，可以调用 Vibrator 模块，例如，设备的按键可以设置不同强度和时长的振动，闹钟和来电可以设置不同强度和时长的单次或周期性振动。

### 7.2.3.2 接口说明

振动器模块主要提供的功能有：查询设备上振动器的列表，查询某个振动器是否支持某种振动效果，触发和关闭振动器。VibratorAgent 类开放能力如下，具体请查阅 API 参考文档。

| 接口名 | 描述 |
| --- | --- |
| getVibratorIdList() | 获取硬件设备上的振动器列表。 |
| isSupport(int) | 根据指定的振动器 Id 查询硬件设备是否存在该振动器。 |
| isEffectSupport(int, String) | 查询指定的振动器是否支持指定的震动效果。 |
| startOnce(int, String) | 对指定的振动器创建指定效果的一次性振动。 |
| startOnce(String) | 对指定的振动器创建指定效果的一次性振动。 |
| startOnce(int, int) | 对指定的振动器创建指定振动时长的一次性振动。 |

| 接口名 | 描述 |
| --- | --- |
| startOnce(int) | 对指定的振动器创建指定振动时长的一次性振动。 |
| start(int, VibrationPattern) | 对指定的振动器创建自定义效果的波形或一次性振动。 |
| start(VibrationPattern) | 对指定的振动器创建自定义效果的波形或一次性振动。 |
| stop(int, String) | 关闭指定的振动器指定模式的振动。 |
| stop(String) | 关闭指定的振动器指定模式的振动。 |

**表 1** VibratorAgent 的主要接口

### 7.2.3.3 开发步骤

1. 控制设备上的振动器，需要在“config.json”里面进行配置请求权限。具体如下：

```
1.  "reqPermissions": [
2.      {
3.          "name": "ohos.permission.VIBRATE",
4.          "reason": "",
5.          "usedScene": {
6.              "ability": [
7.                  ".MainAbility"
8.              ],
9.              "when": "inuse"
10.         }
11.     }
12. ]
```

2. 查询硬件设备上的振动器列表。
3. 查询指定的振动器是否支持指定的震动效果。
4. 创建不同效果的振动。
5. 关闭指定的振动器指定模式的振动。

```
1.  private VibratorAgent vibratorAgent = new VibratorAgent();
2.  
3.  private int[] timing = {1000, 1000, 2000, 5000};
```

```
4.
5.  private int[] intensity = {50, 100, 200, 255};
6.
7.  @Override
8.  public void onStart(Intent intent) {
9.      super.onStart(intent);
10.     super.setUIContent(ResourceTable.Layout_vibrator_layout);
11.
12.     // ...
13.
14.     // 查询硬件设备上的振动器列表
15.     List<Integer> vibratorList = vibratorAgent.getVibratorIdList();
16.     if (vibratorList.isEmpty()) {
17.         return;
18.     }
19.     int vibratorId = vibratorList.get(0);
20.
21.     // 查询指定的振动器是否支持指定的振动效果
22.     boolean isSupport = vibratorAgent.isEffectSupport(vibratorId,
23.             VibrationPattern.VIBRATOR_TPYE_CAMERA_CLICK);
24.
25.     // 创建指定效果的一次性振动
26.     boolean vibrateEffectResult = vibratorAgent.vibrate(vibratorId,
27.             VibrationPattern.VIBRATOR_TPYE_CAMERA_CLICK);
28.
29.     // 创建指定振动时长的一次性振动
30.     int vibratorTiming = 1000;
31.     boolean vibrateResult = vibratorAgent.vibrate(vibratorId, vibratorTiming);
32.
33.     // 创建自定义效果的周期性波形振动
34.     int count = 5;
35.     VibrationPattern vibrationPeriodEffect = VibrationPattern.createPeriod(timing, intensity, count);
36.     boolean vibratePeriodResult = vibratorAgent.vibrate(vibratorId, vibrationPeriodEffect);
37.
38.     // 创建自定义效果的一次性振动
39.     VibrationPattern vibrationOnceEffect = VibrationPattern.createSingle(3000, 50);
40.     boolean vibrateSingleResult = vibratorAgent.vibrate(vibratorId, vibrationOnceEffect);
41.
```

```
42.     // 关闭指定的振动器自定义模式的振动
43.     boolean stopResult = vibratorAgent.stop(vibratorId,
44.             VibratorAgent.VIBRATOR_STOP_MODE_CUSTOMIZED);
45. }
```

## 7.3 位置

### 7.3.1 概述

移动终端设备已经深入人们日常生活的方方面面，如查看所在城市的天气、新闻轶事、出行打车、旅行导航、运动记录。这些习以为常的活动，都离不开定位用户终端设备的位置。

当用户处于这些丰富的使用场景中时，系统的位置能力可以提供实时准确的位置数据。对于开发者，设计基于位置体验的服务，也可以使应用的使用体验更贴近每个用户。

当应用在实现基于设备位置的功能时，如：驾车导航，记录运动轨迹等，可以调用该模块的 API 接口，完成位置信息的获取。

#### 7.3.1.1 基本概念

位置能力用于确定用户设备在哪里，系统使用位置坐标标示设备的位置，并用多种定位技术提供服务，如 GNSS 定位、基站定位、WLAN/蓝牙定位（基站定位、WLAN/蓝牙定位后续统称“网络定位技术”）。通过这些定位技术，无论用户设备在室内或是户外，都可以准确地确定设备位置。

- **坐标**

  系统以 1984 年世界大地坐标系统为参考，使用经度、纬度数据描述地球上的一个位置。

- **GNSS 定位**

基于全球导航卫星系统，包含：GPS、GLONASS、北斗、Galileo 等，通过导航卫星，设备芯片提供的定位算法，来确定设备准确位置。定位过程具体使用哪些定位系统，取决于用户设备的硬件能力。

- **基站定位**

  根据设备当前驻网基站和相邻基站的位置，估算设备当前位置。此定位方式的定位结果精度相对较低，并且需要设备可以访问蜂窝网络。

- **WLAN、蓝牙定位**

  根据设备可搜索到的周围 WLAN、蓝牙设备位置，估算设备当前位置。此定位方式的定位结果精度依赖设备周围可见的固定 WLAN、蓝牙设备的分布，密度较高时，精度也相较与基站定位方式更高，同时也需要设备可以访问网络。

#### 7.3.1.2 运作机制

位置能力作为系统为应用提供的一种基础服务，需要应用在所使用的业务场景，向系统主动发起请求，并在业务场景结束时，主动结束此请求，在此过程中系统会将实时的定位结果上报给应用。

#### 7.3.1.3 约束与限制

使用设备的位置能力，需要用户进行确认并主动开启位置开关。如果位置开关没有开启，系统不会向任何应用提供位置服务。

设备位置信息属于用户敏感数据，所以即使用户已经开启位置开关，应用在获取设备位置前仍需向用户申请位置访问权限。在用户确认允许后，系统才会向应用提供位置服务。

## 7.3.2 获取设备的位置信息

### 7.3.2.1 场景介绍

开发者可以调用 HarmonyOS 位置相关接口，获取设备实时位置，或者最近的历史位置。

对于位置敏感的应用业务，建议获取设备实时位置信息。如果不需要设备实时位置信息，并且希望尽可能的节省耗电，开发者可以考虑获取最近的历史位置。

### 7.3.2.2 接口说明

获取设备的位置信息，所使用的接口说明如下。

| 接口名 | 功能描述 |
| --- | --- |
| Locator(Context context) | 创建 Locator 实例对象。 |
| RequestParam(int scenario) | 根据定位场景类型创建定位请求的 RequestParam 对象。 |
| onLocationReport(Location location) | 获取定位结果。 |
| startLocating(RequestParam request, LocatorCallback callback) | 向系统发起定位请求。 |
| requestOnce(RequestParam request, LocatorCallback callback) | 向系统发起单次定位请求。 |
| stopLocating(LocatorCallback callback) | 结束定位。 |
| getCachedLocation() | 获取系统缓存的位置信息。 |

**表 1** 获取位置信息 API 功能介绍

#### 7.3.2.3 开发步骤

1. 应用在使用系统能力前，需要检查是否已经获取用户授权访问设备位置信息。如未获得授权，可以向用户申请需要的位置权限。

   系统提供的定位权限有：

- ohos.permission.LOCATION
- ohos.permission.LOCATION_IN_BACKGROUND

访问设备的位置信息，必须申请 ohos.permission.LOCATION 权限，并且获得用户授权。

如果应用在后台运行时也需要访问设备位置，除需要将应用声明为允许后台运行外，还必须申请 ohos.permission.LOCATION_IN_BACKGROUND 权限，这样应用在切入后台之后，系统依然可以继续上报位置信息。

开发者可以在应用 config.json 文件中声明所需要的权限，示例代码如下：

```
1.  {
2.      " reqPermissions": [{
3.          "name": "ohos.permission.LOCATION",
4.          "reason": "$string:reason_description",
5.          "usedScene": {
6.              "ability": ["com.myapplication.LocationAbility"],
7.              "when": "inuse"
8.          }, {
9.          ...
10.         }]
11.     }]
12. }
```

**说明**

配置字段详细说明见权限开发指导。在使用系统位置能力时，向用户动态申请位置权限，申请方式请参考动态申请权限开发步骤。

2. 实例化 Locator 对象，所有与基础定位能力相关的功能 API，都是通过 Locator 提供的。

```
1.  Locator locator = new Locator(context);
```

其中入参需要提供当前应用程序的 AbilityInfo 信息，便于系统管理应用的定位请求。

3. 实例化 RequestParam 对象，用于告知系统该向应用提供何种类型的位置服务，以及位置结果上报的频率。

**方式一：**

为了面向开发者提供贴近其使用场景的 API 使用方式，系统定义了几种常见的位置能力使用场景，并针对使用场景做了适当的优化处理，应用可以直接匹配使用，简化开发复杂度。系统当前支持场景如下表所示。

| 场景名称 | 常量定义 | 说明 |
| --- | --- | --- |
| 导航场景 | SCENE_NAVIGATION | 适用于在户外定位设备实时位置的场景，如车载、步行导航。在此场景下，为保证系统提供位置结果精度最优，主要使用 GNSS 定位技术提供定位服务，结合场景特点，在导航启动之初，用户很可能在室内、车库等遮蔽环境，GNSS 技术很难提供位置服务。为解决此问题，我们会在 GNSS 提供稳定位置结果之前，使用系统网络定位技术，向应用提供位置服务，以在导航初始阶段提升用户体验。<br>此场景默认以最小 1 秒间隔上报定位结果，使用此场景的应用必须申请 ohos.permission.LOCATION 权限，同时获得用户授权。 |
| 轨迹跟踪场景 | SCENE_TRAJECTORY_TRACKING | 适用于记录用户位置轨迹的场景，如运动类应用记录轨迹功能。主要使用 GNSS 定位技术提供定位服务。<br>此场景默认以最小 1 秒间隔上报定位结果，并且应用必须申请 ohos.permission.LOCATION 权限，同时获得用户授权。 |
| 出行约车场景 | SCENE_CAR_HAILING | 适用于用户出行打车时定位当前位置的场景，如网约车类应用。<br>此场景默认以最小 1 秒间隔上报定位结果，并且应用必须申请 ohos.permission.LOCATION 权限，同时获得用户授权。 |
| 生活服务场景 | SCENE_DAILY_LIFE_SERVICE | 生活服务场景，适用于不需要定位用户精确位置的使用场景，如新闻资讯、网购、点餐类应用，做推荐、推送时定位用户大致位置即可。<br>此场景默认以最小 1 秒间隔上报定位结果，并且应用至少申请 ohos.permission.LOCATION 权限，同时获得用户授权。 |

| 场景名称 | 常量定义 | 说明 |
|---|---|---|
| 无功耗场景 | SCENE_NO_POWER | 无功耗场景，适用于不需要主动启动定位业务。系统在响应其他应用启动定位业务并上报位置结果时，会同时向请求此场景的应用程序上报定位结果，当前的应用程序不产生定位功耗。<br>此场景默认以最小 1 秒间隔上报定位结果，并且应用需要申请 ohos.permission.LOCATION 权限，同时获得用户授权。 |

**表 2** 定位场景类型说明

以导航场景为例，实例化方式如下：

```
1.  RequestParam requestParam = new RequestParam(RequestParam.SCENE_NAVIGATION);
```

## 方式二：

如果定义的现有场景类型不能满足所需的开发场景，系统提供了基本的定位优先级策略类型。

| 策略类型 | 常量定义 | 说明 |
|---|---|---|
| 定位精度优先策略 | PRIORITY_ACCURACY | 定位精度优先策略主要以 GNSS 定位技术为主，在开阔场景下可以提供米级的定位精度，具体性能指标依赖用户设备的定位硬件能力，但在室内等强遮蔽定位场景下，无法提供准确的位置服务。<br>应用必须申请 ohos.permission.LOCATION 权限，同时获得用户授权。 |
| 快速定位优先策略 | PRIORITY_FAST_FIRST_FIX | 快速定位优先策略会同时使用 GNSS 定位、基站定位和 WLAN、蓝牙定位技术，以便室内和户外场景下，通过此策略都可以获得位置结果，当各种定位技术都有提供位置结果时，系统会选择其中精度较好的结果返回给应用。因为对各种定位技术同时使用，对设备的硬件资源消耗较大，功耗也较大。<br>应用必须申请 ohos.permission.LOCATION 权限，同时获得用户授权。 |
| 低功耗定位优先策略 | PRIORITY_LOW_POWER | 低功耗定位优先策略主要使用基站定位和 WLAN、蓝牙定位技术，也可以同时提供室内和户外场景下的位置服务，因为其依赖周边基站、可见 WLAN、蓝牙设备的分布情况，定位结果的精度波动范围较大，如果对定位结果精度要求不高，或者使用场景多在有基站、可见 WLAN、蓝牙设备高密度分布的情况下，推荐使用，可以有效节省设备功耗。<br>应用至少申请 ohos.permission.LOCATION 权限，同时获得用户授权。 |

| 策略类型 | 常量定义 | 说明 |
| --- | --- | --- |

表 3 定位优先级策略类型说明：

以定位精度优先策略为例，实例化方式如下：

```
2.  RequestParam requestParam = new RequestParam(RequestParam.PRIORITY_ACCURACY,0,0);
```

后两个入参用于限定系统向应用上报定位结果的频率，分别为位置上报的最小时间间隔，和位置上报的最小距离间隔，开发者可以参考 API 具体说明进行开发。

4. 实例化 LocatorCallback 对象，用于向系统提供位置上报的途径。

   应用需要自行实现系统定义好的回调接口，并将其实例化。系统在定位成功确定设备的实时位置结果时，会通过 onLocationReport 接口上报给应用。应用程序可以在 onLocationReport 接口的实现中完成自己的业务逻辑。

```
1.  MyLocatorCallback locatorCallback = new MyLocatorCallback();
2.
3.  public class MyLocatorCallback implements LocatorCallback {
4.      @Override
5.      public void onLocationReport(Location location) {
6.      }
7.
8.      @Override
9.      public void onStatusChanged(int type) {
10.     }
11.
12.     @Override
13.     public void onErrorReport(int type) {
14.     }
15. }
```

5. 启动定位。

```
1.  locator.startLocating(requestParam, locatorCallback);
```

如果应用不需要持续获取位置结果，可以使用如下方式启动定位，系统会上报一次实时定位结果后，自动结束应用的定位请求。应用不需要执行结束定位。

```
2.  locator.requestOnce(requestParam, locatorCallback);
```

6. （可选）结束定位。

```
1.  locator.stopLocating(locatorCallback);
```

如果应用使用场景不需要实时的设备位置，可以获取系统缓存的最近一次历史定位结果。

```
2. locator.getCachedLocation();
```

此接口的使用需要应用向用户申请 LOCATION 位置权限。

### 7.3.3 （逆）地理编码转化

#### 7.3.3.1 场景介绍

使用坐标描述一个位置，非常准确，但是并不直观，面向用户表达并不友好。

系统向开发者提供了地理编码转化能力（将坐标转化为地理编码信息），以及逆地理编码转化能力（将地理描述转化为具体坐标）。其中地理编码包含多个属性来描述位置，包括国家、行政区划、街道、门牌号、地址描述等等，这样的信息更便于用户理解。

#### 7.3.3.2 接口说明

进行坐标和地理编码信息的相互转化，所使用的接口说明如下。

| 接口名 | 功能描述 |
|---|---|
| GeoConvert() | 创建 GeoConvert 实例对象。 |
| getAddressFromLocation(double latitude, double longitude, int maxItems) | 根据指定的经纬度坐标获取地理位置信息。 |
| getAddressFromLocationName(String description, int maxItems) | 根据地理位置信息获取相匹配的包含坐标数据的地址列表。 |
| getAddressFromLocationName(String description, double minLatitude, double minLongitude, double maxLatitude, double maxLongitude, int maxItems) | 根据指定的位置信息和地理区域获取相匹配的包含坐标数据的地址列表。 |

**表 1** 地理编码转化能力和逆地理编码转化能力的 API 功能介绍

#### 7.3.3.3 开发步骤

1. 实例化 GeoConvert 对象，所有与(逆)地理编码转化能力相关的功能 API，都是通过 GeoConvert 提供的。

```
1.  GeoConvert geoConvert = new GeoConvert();
```

2. 获取转化结果。

- 坐标转化地理位置信息。

```
1.  geoConvert.getAddressFromLocation(纬度值, 经度值, 1);
```

参考接口 API 说明，应用可以获得与此坐标匹配的 GeoAddress 列表，应用可以根据实际使用需求，读取相应的参数数据。

- 位置描述转化坐标。

```
1.  geoConvert.getAddressFromLocationName("北京大兴国际机场", 1);
```

参考接口 API 说明，应用可以获得与位置描述相匹配的 GeoAddress 列表，其中包含对应的坐标数据，请参考 API 使用。

如果需要查询的位置描述可能出现多地重名的请求，可以同过设置一个经纬度范围，以便高效获取期望的准确结果。

```
2.  geoConvert.getAddressFromLocationName("北京大兴国际机场", 纬度下限, 经度下限, 纬度上限, 经度上限, 1);
```

## 7.4 设置

### 7.4.1 概述

应用程序可以对系统各类设置项进行增、删、改、查等操作。例如，三方应用提前注册飞行模式设置项的回调，当用户通过系统设置修改终端的飞行模式状态时，三方应用会检测到此设置项发生变化并进行适配。如检测到飞行模式开启，将进入离线状态；检测到飞行模式关闭，其将重新获取在线数据。

#### 7.4.1.1 基本概念

系统设置数据项分为 TTS（Text To Speech）、Wireless、Network、Input、Sound、Display、Date、Call、General 九类，应用程序可以根据自身拥有的权限对其进行操作。

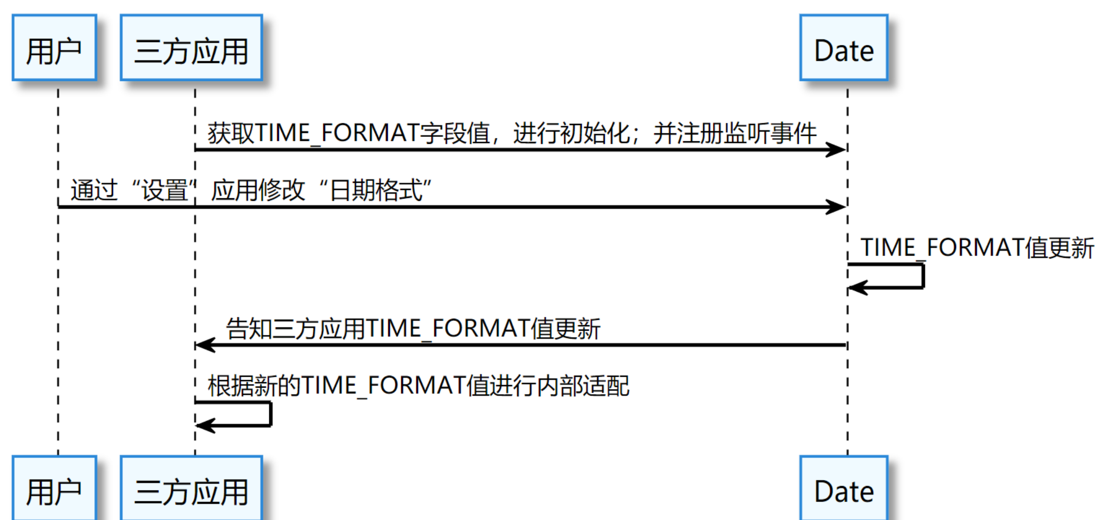
### 7.4.2 开发指导

#### 7.4.2.1 场景介绍

TTS、Wireless、Network、Input、Sound、Display、Date、Call、General 九类定义了表征终端设备状态的相关字段，如屏幕亮度、日期格式、字体显示大小等，应用程序可以根据自身所拥有的权限对其进行增、删、改、查等操作，并进行相应的场景适配。

例如：TIME_FORMAT——表示日期格式，应用程序可进行读写。

**图 1** 数据表更新过程



### 7.4.2.2 接口说明

SystemSettings 提供系统设置的相关接口，包括 TTS、Wireless、Network、Input、Sound、Display、Date、Call、General 九类字段的存储和检索接口。应用程序通过 AppSettings 类提供的方法对其自身的能力进行查询。

| 接口名 | 描述 |
| --- | --- |
| canShowOverlays(Context context) | 检查指定应用程序是否可以显示在其他应用之上。 |
| checkSetPermission(Context context) | 通过应用上下文检查指定的应用是否具有修改系统设置的权限。 |

表 1 AppSettings 的主要接口

| 接口名 | 描述 |
| --- | --- |
| getUri(String name) | 为特定的字段构造 URI，用于 DataAbility 的数据监视。 |
| getValue(DataAbilityHelper dataAbilityHelper, String name) | 获取指定字段的值。 |
| setValue(DataAbilityHelper dataAbilityHelper, String name, String value) | 设置指定字段的值。 |

表 2 SystemSettings 的主要接口

| 字段名 | 字段描述 |
| --- | --- |
| DEFAULT_TTS_PITCH | 文本转语音引擎的默认音调。 |
| DEFAULT_TTS_RATE | 文本转语音引擎的默认语速。 |

表 3 SystemSettings.TTS 提供的典型字段

| 字段名 | 字段描述 |
| --- | --- |

| 字段名 | 字段描述 |
| --- | --- |
| BLUETOOTH_STATUS | 蓝牙开启状态。 |
| WIFI_STATUS | WLAN 是否启用。 |
| WIFI_TO_MOBILE_DATA_AWAKE_TIMEOUT | 从 WLAN 断开连接后等待建立移动数据连接时保持唤醒锁的最长时间。 |

**表 4** SystemSettings.Wireless 提供的典型字段

| 字段名 | 字段描述 |
| --- | --- |
| DATA_ROAMING_STATUS | 数据漫游开启状态。 |
| NETWORK_PREFERENCE_USAGE | 设置用户经常使用的网络。 |

**表 5** SystemSettings.Network 提供的典型字段

| 字段名 | 字段描述 |
| --- | --- |
| DEFAULT_INPUT_METHOD | 设置默认的输入法，并记录此输入法的 ID。 |
| ACTIVATED_INPUT_METHODS | 已激活的输入法列表。 |
| AUTO_CAPS_TEXT_INPUT | 设置文本编辑器是否启用自动大写。 |

**表 6** SystemSettings.Input 提供的典型字段

| 字段名 | 字段描述 |
| --- | --- |
| HAPTIC_FEEDBACK_STATUS | 设置是否开启触摸反馈。 |
| VIBRATE_WHILE_RINGING | 设置来电响铃时是否震动。 |
| DEFAULT_NOTIFICATION_SOUND | 系统默认通知铃声的存储区。 |

**表 7** SystemSettings.Sound 提供的字段

| 字段名 | 字段描述 |
| --- | --- |
| **字段名** | **字段描述** |
| FONT_SCALE | 设置字体大小因子。 |
| SCREEN_BRIGHTNESS_STATUS | 设置屏幕亮度。 |
| AUTO_SCREEN_BRIGHTNESS | 设置是否打开屏幕亮度自动调节模式。 |
| SCREEN_OFF_TIMEOUT | 设置设备屏幕自动休眠时间。 |

**表 8** SystemSettings.Display 提供的典型字段

| 字段名 | 字段描述 |
| --- | --- |
| DATE_FORMAT | 设置日期格式。 |
| TIME_FORMAT | 设置以 12 或 24 小时制显示时间。 |
| AUTO_GAIN_TIME | 是否从网络（NITZ）自动获取日期，时间和时区的值。 |
| AUTO_GAIN_TIME_ZONE | 是否从网络（NITZ）自动获取时区的值。 |

**表 9** SystemSettings.Date 提供的典型字段

| 字段名 | 字段描述 |
| --- | --- |
| SETUP_WIZARD_FINISHED | 识别开机向导是否已经运行过。 |
| AIRPLANE_MODE_STATUS | 飞行模式是否开启。 |
| DEVICE_NAME | 设备名称。 |
| ACCESSIBILITY_STATUS | 设置辅助功能是否可用。 |

**表 10** SystemSettings.General 提供的典型字段

| 字段名 | 字段描述 |
| --- | --- |
| RTT_CALLING_STATUS | 设置来去电是否启动 RTT 模式进行应答。 |

表 11 SystemSettings.Call 提供的典型字段

### 7.4.2.3 开发步骤

1. 应用程序打开某个 Slice 时，在 OnStart()时，注册相关设置项的回调，并读取一次该设置项的值，进行初始化适配。

```
@Override
public void onStart(Intent intent) {
    // ...
    dataAbilityHelper = DataAbilityHelper.creator(this);
    IDataAbilityObserver dataAbilityObserver = new IDataAbilityObserver() {
        @Override
        public void onChange() {
            String timeFormat = SystemSettings.getValue(dataAbilityHelper, SystemSettings.Date.TIME_FORMAT);
            setTimeFormat(timeFormat);
        }
    };
    dataAbilityHelper.registerObserver(SystemSettings.getUri(SystemSettings.Date.TIME_FORMAT), dataAbilityObserver);
}

void setTimeFormat(String timeFormat) {
    if ("12".equals(timeFormat)) {
        // Display in 12-hour format
    } else {
        // Display in 24-hour format
    }
}
```

2. 在 onStop()时，解除回调注册。

```
dataAbilityHelper.unregisterObserver(SystemSettings.getUri(SystemSettings.Date.TIME_FORMAT), dataAbilityObserver);
```

# 8 数据管理

## 8.1 关系型数据库

### 8.1.1 概述

关系型数据库（Relational Database，RDB）是一种基于关系模型来管理数据的数据库。

HarmonyOS 关系型数据库基于 SQLite 组件提供了一套完整的对本地数据库进行管理的机制，对外提供了一系列的增、删、改、查接口，也可以直接运行用户输入的 SQL 语句来满足复杂的场景需要。HarmonyOS 提供的关系型数据库功能更加完善，查询效率更高。

#### 8.1.1.1 基本概念

- **关系型数据库**

  创建在关系模型基础上的数据库，以行和列的形式存储数据。

- **谓词**

  数据库中用来代表数据实体的性质、特征或者数据实体之间关系的词项，主要用来定义数据库的操作条件。

- **结果集**

  指用户查询之后的结果集合，可以对数据进行访问。结果集提供了灵活的数据访问方式，可以更方便的拿到用户想要的数据。

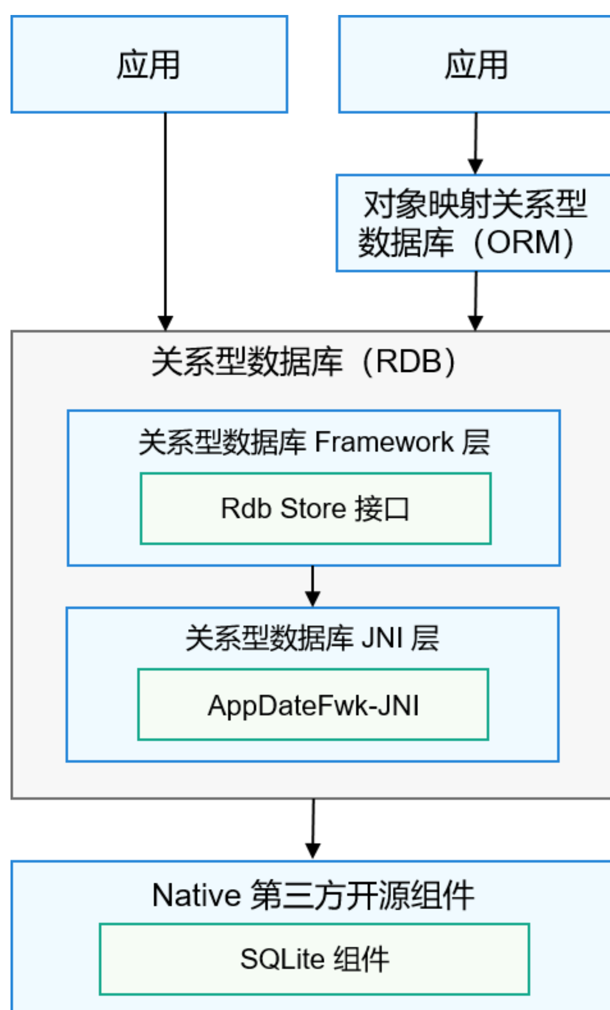
- **SQLite 数据库**

  一款轻型的数据库，是遵守 ACID 的关系型数据库管理系统。它是一个开源的项目。

#### 8.1.1.2 运作机制

HarmonyOS 关系型数据库对外提供通用的操作接口，底层使用 SQLite 作为持久化存储引擎，支持 SQLite 具有的所有数据库特性，包括但不限于事务、索引、视图、触发器、外键、参数化查询和预编译 SQL 语句。

**图 1** 关系型数据库运作机制




#### 8.1.1.3 默认配置

- 如果不指定数据库的日志模式，那么系统默认日志方式是 WAL（Write Ahead Log）模式。
- 如果不指定数据库的落盘模式，那么系统默认落盘方式是 FULL 模式。
- HarmonyOS 数据库使用的共享内存默认大小是 2MB。

#### 8.1.1.4 约束与限制

- 数据库中连接池的最大数量是 4 个，用以管理用户的读写操作。
- 为保证数据的准确性，数据库同一时间只能支持一个写操作。

### 8.1.2 开发指导

#### 8.1.2.1 场景介绍

关系型数据库是在 SQLite 基础上实现的本地数据操作机制，提供给用户无需编写原生 SQL 语句就能进行数据增删改查的方法，同时也支持原生 SQL 操作。

#### 8.1.2.2 接口说明

**数据库的创建和删除**

关系型数据库提供了数据库创建方式，以及对应的删除接口，涉及的 API 如下所示。

| 类名 | 接口名 | 描述 |
| --- | --- | --- |
| StoreConfig.Builder | public builder() | 对数据库进行配置，包括设置数据库名、存储模式、日志模式、同步模式，是否为只读，及对数据库加密。 |
| RdbOpenCallback | public abstract void onCreate(RdbStore store) | 数据库创建时被回调，开发者可以在该方法中初始化表结构，并添加一些应用使用到的初始化数据。 |
| RdbOpenCallback | public abstract void onUpgrade(RdbStore store, int currentVersion, int targetVersion) | 数据库升级时被回调。 |

| 类名 | 接口名 | 描述 |
| --- | --- | --- |
| DatabaseHelper | public RdbStore getRdbStore(StoreConfig config, int version, RdbOpenCallback openCallback, ResultSetHook resultSetHook) | 根据配置创建或打开数据库。 |
| DatabaseHelper | public boolean deleteRdbStore(String name) | 删除指定的数据库。 |

**表 1** 数据库创建和删除 API

### 数据库的加密

关系型数据库提供数据库加密的能力，创建数据库时传入指定密钥、创建加密数据库，后续打开加密数据库时，需要传入正确密钥。

| 类名 | 接口名 | 描述 |
| --- | --- | --- |
| StoreConfig.Builder | Builder setEncryptKey(byte[] encryptKey) | 为数据库配置类设置数据库加密密钥，创建或打开数据库时传入包含数据库加密密钥的配置类，即可创建或打开加密数据库。 |

**表 2** 数据库传入密钥接口

### 数据库的增删改查

关系型数据库提供本地数据增删改查操作的能力，相关 API 如下所示。

- **新增**

关系型数据库提供了插入数据的接口，通过 ValuesBucket 输入要存储的数据，通过返回值判断是否插入成功，插入成功时返回最新插入数据所在的行号，失败则返回-1。

| 类名 | 接口名 | 描述 |
| --- | --- | --- |
| RdbStore | long insert(String table, ValuesBucket initialValues) | 向数据库插入数据。<br>table：待添加数据的表名。 |

| 类名 | 接口名 | 描述 |
| --- | --- | --- |
| | | initialValues：以 ValuesBucket 存储的待插入的数据。它提供一系列 put 方法，如 putString(String columnName, String values)，putDouble(String columnName, double value)，用于向 ValuesBucket 中添加数据。 |

表 3 数据库插入 API

- **更新**

调用更新接口，传入要更新的数据，并通过 AbsRdbPredicates 指定更新条件。该接口的返回值表示更新操作影响的行数。如果更新失败，则返回 0。

| 类名 | 接口名 | 描述 |
| --- | --- | --- |
| RdbStore | int update(ValuesBucket values, AbsRdbPredicates predicates) | 更新数据库表中符合谓词指定条件的数据。<br>values：以 ValuesBucket 存储的要更新的数据。<br>predicates：指定了更新操作的表名和条件。AbsRdbPredicates 的实现类有两个：RdbPredicates 和 RawRdbPredidates。<br>RdbPredicates：支持调用谓词提供的 equalTo 等接口，设置更新条件。<br>RawRdbPredidates：仅支持设置表名、where 条件子句、whereArgs 三个参数，不支持 equalTo 等接口调用。 |

表 4 数据库更新 API

- **删除**

调用删除接口，通过 AbsRdbPredicates 指定删除条件。该接口的返回值表示删除的数据行数，可根据此值判断是否删除成功。如果删除失败，则返回 0。

| 类名 | 接口名 | 描述 |
| --- | --- | --- |
| RdbStore | int delete(AbsRdbPredicates predicates) | 删除数据。<br>predicates：Rdb 谓词，指定了删除操作的表名和条件。AbsRdbPredicates 的实现类有两个：RdbPredicates 和 RawRdbPredidates。<br>RdbPredicates：支持调用谓词提供的 equalTo 等接口，设置更新条件。 |

| 类名 | 接口名 | 描述 |
| --- | --- | --- |
| | | RawRdbPredidates：仅支持设置表名、where 条件子句、whereArgs 三个参数，不支持 equalTo 等接口调用。 |

表 5 数据库删除 API

- **查询**

  关系型数据库提供了两种查询数据的方式：

  - 直接调用查询接口。使用该接口，会将包含查询条件的谓词自动拼接成完整的 SQL 语句进行查询操作，无需用户传入原生的 SQL。
  - 执行原生的用于查询的 SQL 语句。

| 类名 | 接口名 | 描述 |
| --- | --- | --- |
| RdbStore | ResultSet query(AbsRdbPredicates predicates, String[] columns) | 查询数据。<br>predicates：谓词，可以设置查询条件。AbsRdbPredicates 的实现类有两个：RdbPredicates 和 RawRdbPredidates。<br>RdbPredicates：支持调用谓词提供的 equalTo 等接口，设置查询条件。<br>RawRdbPredidates：仅支持设置表名、where 条件子句、whereArgs 三个参数，不支持 equalTo 等接口调用。<br>columns：规定查询返回的列。 |
| RdbStore | ResultSet querySql(String sql, String[] sqlArgs) | 执行原生的用于查询操作的 SQL 语句。<br>sql：原生用于查询的 sql 语句。<br>sqlArgs：sql 语句中占位符参数的值，若 select 语句中没有使用占位符，该参数可以设置为 null。 |

表 6 数据库查询 API

**数据库谓词的使用**

关系型数据库提供了用于设置数据库操作条件的谓词 AbsRdbPredicates，其中包括两个实现子类 RdbPredicates 和 RawRdbPredicates：

- RdbPredicates：开发者无需编写复杂的 SQL 语句，仅通过调用该类中条件相关的方法，如 equalTo、notEqualTo、groupBy、orderByAsc、beginsWith 等，就可自动完成 SQL 语句拼接，方便用户聚焦业务操作。
- RawRdbPredicates：可满足复杂 SQL 语句的场景，支持开发者自己设置 where 条件子句和 whereArgs 参数。不支持 equalTo 等条件接口的使用。

| 类名 | 接口名 | 描述 |
| --- | --- | --- |
| RdbPredicates | RdbPredicates equalTo(String field, String value) | 设置谓词条件，满足 filed 字段与 value 值相等。 |
| RdbPredicates | RdbPredicates notEqualTo(String field, String value) | 设置谓词条件，满足 filed 字段与 value 值不相等。 |
| RdbPredicates | RdbPredicates beginsWith(String field, String value) | 设置谓词条件，满足 field 字段以 value 值开头。 |
| RdbPredicates | RdbPredicates between(String field, int low, int high) | 设置谓词条件，满足 field 字段在最小值 low 和最大值 high 之间。 |
| RdbPredicates | RdbPredicates orderByAsc(String field) | 设置谓词条件，根据 field 字段升序排列。 |
| RawRdbPredicates | void setWhereClause(String whereClause) | 设置 where 条件子句。 |
| RawRdbPredicates | void setWhereArgs(List<String> whereArgs) | 设置 whereArgs 参数，该值表示 where 子句中占位符的值。 |

**表 7** 数据库谓词 API

## 查询结果集的使用

关系型数据库提供了查询返回的结果集 ResultSet，他指向查询结果中的一行数据，供用户对查询结果进行遍历和访问。ReusltSet 的对外 API 如下表格。

| 类名 | 接口名 | 描述 |
| --- | --- | --- |
| ResultSet | boolean goTo(int offset) | 从结果集当前位置移动指定偏移量。 |

| 类名 | 接口名 | 描述 |
| --- | --- | --- |
| ResultSet | boolean goToRow(int position) | 将结果集移动到指定位置。 |
| ResultSet | boolean goToNextRow() | 将结果集向后移动一行。 |
| ResultSet | boolean goToPreviousRow() | 将结果集向前移动一行。 |
| ResultSet | boolean isStarted() | 判断结果集是否被移动过。 |
| ResultSet | boolean isEnded() | 判断结果集当前位置是否在最后一行之后。 |
| ResultSet | boolean isAtFirstRow() | 判断结果集当前位置是否在第一行。 |
| ResultSet | boolean isAtLastRow() | 判断结果集当前位置是否在最后一行。 |
| ResultSet | int getRowCount() | 获取当前结果集中的记录条数。 |
| ResultSet | int getColumnCount() | 获取结果集中的列数。 |
| ResultSet | String getString(int columnIndex) | 获取当前行指定索引列的值，以 String 类型返回。 |
| ResultSet | byte[] getBlob(int columnIndex) | 获取当前行指定列的值，以字节数组形式返回。 |
| ResultSet | double getDouble(int columnIndex) | 获取当前行指定列的值，以 double 型返回。 |

**表 8** 结果集 API

### 事务

关系型数据库提供事务机制，来保证用户操作的原子性。对单条数据进行数据库操作时，无需开启事务；插入大量数据时，开启事务可以保证数据的准确性。如果中途操作出现失败，会执行回滚操作。

| 类名 | 接口名 | 描述 |
| --- | --- | --- |
| RdbStore | beginTransaction() | 开启事务。 |
| RdbStore | markAsCommit() | 设置事务的标记为成功。 |
| RdbStore | endTransaction() | 结束事务。 |

表 9 事务 API

#### 事务和结果集观察者

关系型数据库提供了事务和结果集观察者能力，当对应的事件被触发时，观察者会收到通知。

| 类名 | 接口名 | 描述 |
| --- | --- | --- |
| RdbStore | beginTransactionWithObserver(TransactionObserver transactionObserver) | 开启事务，并观察事务的启动、提交和回滚。 |
| ResultSet | void registerObserver(DataObserver observer) | 注册结果集的观察者。 |
| ResultSet | void unregisterObserver(DataObserver observer) | 注销结果集的观察者。 |

#### 数据库的备份和恢复

用户可以将当前数据库的数据进行保存进行备份，还可以在需要的时候进行数据恢复。

| 类名 | 接口名称 | 描述 |
| --- | --- | --- |
| RdbStore | boolean restore(String srcName) | 数据库恢复接口，从指定的非加密数据库文件中恢复数据。 |
| RdbStore | boolean restore(String srcName, byte[] srcEncryptKey, byte[] destEncryptKey) | 数据库恢复接口，从指定的数据库文件（加密和非加密均可）中恢复数据。 |
| RdbStore | boolean backup(String destName) | 数据库备份接口，备份出的数据库文件是非加密的。 |

| 类名 | 接口名称 | 描述 |
| --- | --- | --- |
| RdbStore | boolean backup(String destName, byte[] destEncryptKey) | 数据库备份接口，此方法经常用在备份出加密数据库场景。 |

表 10 数据库备份和恢复

### 8.1.2.3 开发步骤

1. 创建数据库。

a. 配置数据库相关信息，包括数据库的名称、存储模式、是否为只读模式等。

b. 初始化数据库表结构和相关数据。

c. 创建数据库。

示例代码如下：

```
1. StoreConfig config = StoreConfig.newDefaultConfig("RdbStoreTest.db");
2. private static final RdbOpenCallback callback = new RdbOpenCallback() {
3.     @Override
4.     public void onCreate(RdbStore store) {
5.         store.executeSql("CREATE TABLE IF NOT EXISTS test (id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT NOT NULL,
age INTEGER, salary REAL, blobType BLOB)");
6.     }
7.     @Override
8.     public void onUpgrade(RdbStore store, int oldVersion, int newVersion) {
9.     }
10. };
11. DatabaseHelper helper = new DatabaseHelper(context);
12. RdbStore store = helper.getRdbStore(config, 1, callback, null);
```

2. 插入数据。

d. 构造要插入的数据，以 ValuesBucket 形式存储。

e. 调用关系型数据库提供的插入接口。

示例代码如下：

```
1. ValuesBucket values = new ValuesBucket();
2. values.putInteger("id", 1);
```

```
3.	values.putString("name", "zhangsan");
4.	values.putInteger("age", 18);
5.	values.putDouble("salary", 100.5);
6.	values.putByteArray("blobType", new byte[] {1, 2, 3});
7.	long id = store.insert("test", values);
```

2. 查询数据。

f. 构造用于查询的谓词对象，设置查询条件。

g. 指定查询返回的数据列。

h. 调用查询接口查询数据。

i. 调用结果集接口，遍历返回结果。

示例代码如下：

```
1.	String[] columns = new String[] {"id", "name", "age", "salary"};
2.	RdbPredicates rdbPredicates = new RdbPredicates("test").equalTo("age", 25).orderByAsc("salary");
3.	ResultSet resultSet = store.query(rdbPredicates, columns);
4.	resultSet.goToNextRow();
```

## 8.2 对象映射关系型数据库

### 8.2.1 概述

HarmonyOS 对象映射关系型（Object Relational Mapping，ORM）数据库是一款基于 SQLite 的数据库框架，屏蔽了底层 SQLite 数据库的 SQL 操作，针对实体和关系提供了增删改查等一系列的面向对象接口。应用开发者不必再去编写复杂的 SQL 语句， 以操作对象的形式来操作数据库，提升效率的同时也能聚焦于业务开发。

#### 8.2.1.1 基本概念

- **对象映射关系型数据库的三个主要组件**：
  - 数据库：被开发者用@Database 注解，且继承了 OrmDatabase 的类，对应关系型数据库。
  - 实体对象：被开发者用@Entity 注解，且继承了 OrmObject 的类，对应关系型数据库中的表。

- 对象数据操作接口：包括数据库操作的入口 OrmContext 类和谓词接口（OrmPredicate）等。

- **谓词**

  数据库中是用来代表数据实体的性质、特征或者数据实体之间关系的词项，主要用来定义数据库的操作条件。对象映射关系型数据库将 SQLite 数据库中的谓词封装成了接口方法供开发者调用。开发者通过对象数据操作接口，可以访问到应用持久化的关系型数据。

- **对象映射关系型数据库**

  通过将实例对象映射到关系上，实现使用操作实例对象的语法，来操作关系型数据库。它是在 SQLite 数据库的基础上提供的一个抽象层。

- **SQLite 数据库**

  一款轻型的数据库，是遵守 ACID 的关系型数据库管理系统。

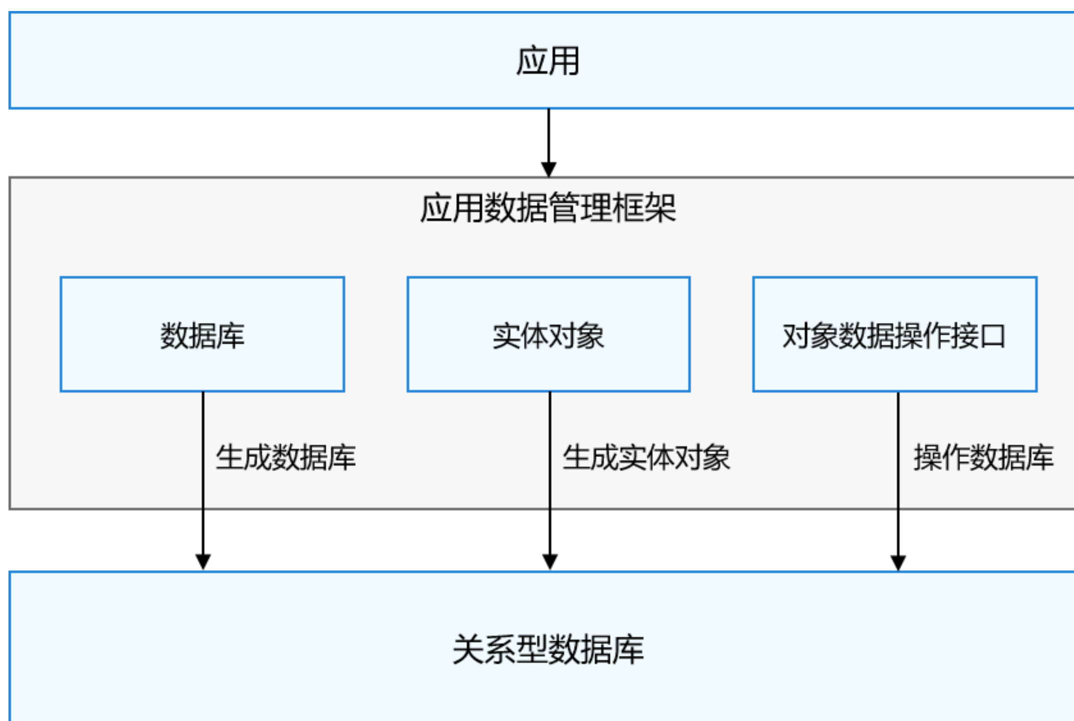
### 8.2.1.2 运作机制

对象映射关系型数据库操作是基于关系型数据库操作接口完成的，实际是在关系型数据库操作的基础上又实现了对象关系映射等特性。因此对象映射关系型数据库跟关系型数据库一样，都使用 SQLite 作为持久化引擎，底层使用的是同一套数据库连接池和数据库连接机制。

使用对象映射关系型数据库的开发者需要先配置实体模型与关系映射文件。应用数据管理框架提供的类生成工具会解析这些文件，生成数据库帮助类，这样应用数据管理框架就能在运行时，根据开发者的配置创建好数据库，并在存储过程中自动完成对象关系映射。开发者再通过对象数据操作接口，如 OrmContext 接口和谓词接口等操作持久化数据库。

对象数据操作接口提供一组基于对象映射的数据操作接口，实现了基于 SQL 的关系模型数据到对象的映射，让用户不需要再和复杂的 SQL 语句打交道，只需简单地操作实体对象的属性和方法。对象数据操作接口支持对象的增删改查操作，同时支持事务操作等。

**图 1** 对象映射关系型数据库运作机制




### 8.2.1.3 默认配置

- 如果不指定数据库的日志模式，那么系统默认日志方式是 WAL（Write Ahead Log）模式。
- 如果不指定数据库的落盘模式，那么系统默认落盘方式是 FULL 模式。
- HarmonyOS 数据库使用的共享内存默认大小是 2MB。

### 8.2.1.4 约束与限制

HarmonyOS 对象映射关系数据库是建立在 HarmonyOS 关系型数据库的基础之上的，所以关系型数据库的一些约束与限制请参考约束与限制。

此外当开发者建立实体对象类时，对象属性的类型可以在下表的类型中选择。不支持使用自定义类型。

| 类型名称 | 描述 | 初始值 |
| --- | --- | --- |
| Integer | 封装整型 | null |

| 类型名称 | 描述 | 初始值 |
| --- | --- | --- |
| int | 整型 | 0 |
| Long | 封装长整型 | null |
| long | 长整型 | 0L |
| Double | 封装双精度浮点型 | null |
| double | 双精度浮点型 | 0 |
| Float | 封装单精度浮点型 | null |
| float | 单精度浮点型 | 0 |
| Short | 封装短整型 | null |
| short | 短整型 | 0 |
| String | 字符串型 | null |
| Boolean | 封装布尔型 | null |
| boolean | 布尔型 | 0 |
| Byte | 封装字节型 | null |
| byte | 字节型 | 0 |
| Character | 封装字符型 | null |
| char | 字符型 | ' ' |
| Date | 日期类 | null |
| Time | 时间类 | null |

| 类型名称 | 描述 | 初始值 |
| --- | --- | --- |
| Timestamp | 时间戳类 | null |
| Calendar | 日历类 | null |
| Blob | 二进制大对象 | null |
| Clob | 字符大对象 | null |

**表 1** 实体对象属性支持的类型

### 8.2.2 开发指导

#### 8.2.2.1 场景介绍

对象映射关系型数据库适用于开发者使用的数据可以分解为一个或多个对象，且需要对数据进行增删改查等操作，但是不希望编写过于复杂的 SQL 语句的场景。

该对象映射关系型数据库的实现是基于关系型数据库，除了数据库版本升降级等场景外，操作对象映射关系型数据库一般不需要编写 SQL 语句，但是仍然要求使用者对于关系型数据库的基本概念有一定的了解。

#### 8.2.2.2 开发能力介绍

对象映射关系型数据库目前可以支持数据库和表的创建，对象数据的增删改查、对象数据变化回调、数据库升降级和备份等功能。

**数据库和表的创建**

1. 创建数据库。开发者需要定义一个表示数据库的类，继承 OrmDatabase，再通过@Database 注解内的 entities 属性指定哪些数据模型类属于这个数据库。

属性：

- version：数据库版本号。
- entities：数据库内包含的表。

2. 创建数据表。开发者可通过创建一个继承了 OrmObject 并用@Entity 注解的类，获取数据库实体对象，也就是表的对象。

属性：

- tableName：表名。
- primaryKeys：主键名，一个表里只能有一个主键，一个主键可以由多个字段组成。
- foreignKeys：外键列表。
- indices：索引列表。

| 接口名称 | 描述 |
| --- | --- |
| @Database | 被@Database 注解且继承了 OrmDatabase 的类对应数据库类。 |
| @Entity | 被@Entity 注解且继承了 OrmObject 的类对应数据表类。 |
| @Column | 被@Column 注解的变量对应数据表的字段。 |
| @PrimaryKey | 被@PrimaryKey 注解的变量对应数据表的主键。 |
| @ForeignKey | 被@ForeignKey 注解的变量对应数据表的外键。 |
| @Index | 被@Index 注解的内容对应数据表索引的属性。 |

**表 1** 注解对照表

### 数据库的加密

对象映射关系型数据库提供数据库加密的能力，创建数据库时传入指定密钥、创建加密数据库，后续打开加密数据库时，需要传入正确密钥。

| 类名 | 接口名 | 描述 |
|---|---|---|
| OrmConfig.Builder | Builder setEncryptKey(byte[] encryptKey) | 为数据库配置类设置数据库加密密钥，创建或打开数据库时传入包含数据库加密密钥的配置类，即可创建或打开加密数据库。 |

**表 2** 数据库传入密钥接口

### 对象数据的增删改查

通过对象数据操作接口，开发者可以对对象数据进行增删改查操作。

| 类名 | 接口名称 | 描述 |
|---|---|---|
| OrmContext | <T extends OrmObject> boolean insert(T object) | 添加方法。 |
| OrmContext | <T extends OrmObject> boolean update(T object) | 更新方法。 |
| OrmContext | <T extends OrmObject> List<T> query(OrmPredicates predicates) | 查询方法。 |
| OrmContext | <T extends OrmObject> boolean delete(T object) | 删除方法。 |
| OrmContext | <T extends OrmObject> OrmPredicates where(Class<T> clz) | 设置谓词方法。 |

**表 3** 对象数据操作接口

### 对象数据的变化观察者设置

通过使用对象数据操作接口，开发者可以在某些数据上设置观察者，接收数据变化的通知。

| 类名 | 接口名称 | 描述 |
|---|---|---|
| OrmContext | void registerStoreObserver(String alias, OrmObjectObserver observer) | 注册数据库变化回调。 |
| OrmContext | void registerContextObserver(OrmContext watchedContext, OrmObjectObserver observer) | 注册上下文变化回调。 |

| 类名 | 接口名称 | 描述 |
| --- | --- | --- |
| OrmContext | void registerEntityObserver(String entityName, OrmObjectObserver observer) | 注册数据库实体变化回调。 |
| OrmContext | void registerObjectObserver(OrmObject ormObject, OrmObjectObserver observer) | 注册对象变化回调。 |

**表 4** 数据变化观察者接口

### 数据库的升降级

通过调用数据库升降级接口，开发者可以将数据库切换到不同的版本。

| 类名 | 接口名称 | 描述 |
| --- | --- | --- |
| OrmMigration | public void onMigrate(int beginVersion, int endVersion) | 数据库版本升降级接口。 |

**表 5** 数据库升降级接口

### 数据库的备份恢复

开发者可以将当前数据库的数据进行备份，在必要的时候进行数据恢复。

| 类名 | 接口名称 | 描述 |
| --- | --- | --- |
| OrmContext | boolean backup(String destPath) | 数据库备份接口。 |
| OrmContext | boolean restore(String srcPath); | 数据库恢复备份接口。 |

**表 6** 数据库备份与恢复接口

## 8.2.2.3 开发步骤

1. 配置“build.gradle”文件。

- 如果使用注解处理器的模块为“com.huawei.ohos.hap”模块，则需要在模块的“build.gradle”文件的“ohos”节点中添加以下配置：

```
1. compileOptions{
2.     annotationEnabled true
3. }
```

- 如果使用注解处理器的模块为“com.huawei.ohos.library”模块，则需要在模块的“build.gradle”文件的“dependencies”节点中配置注解处理器。查看“orm_annotations_java.jar”、“orm_annotations_processor_java.jar”、“javapoet_java.jar”这 3 个 jar 包在 HUAWEI SDK 中的对应目录，并将目录的这三个 jar 包导进来。

```
1. dependencies {
2.     compile files("orm_annotations_java.jar 的路径","orm_annotations_processor_java.jar 的路径","javapoet_java.jar 的路径")
3.     annotationProcessor files("orm_annotations_java.jar 的路径","orm_annotations_processor_java.jar 的路径","javapoet_java.jar 的路径")
4. }
```

- 如果使用注解处理器的模块为“java-library”模块，则需要在模块的“build.gradle”文件的“dependencies”节点中配置注解处理器，并导入“ohos.jar”。

```
1. dependencies {
2.     compile files("ohos.jar 的路径","orm_annotations_java.jar 的路径","orm_annotations_processor_java.jar 的路径","javapoet_java.jar 的路径")
3.     annotationProcessor files("orm_annotations_java.jar 的路径","orm_annotations_processor_java.jar 的路径","javapoet_java.jar 的路径")
4. }
```

2. 构造数据库，即创建数据库类并配置对应的属性。

例如，定义了一个数据库类 BookStore.java，数据库包含了“User”，"Book"，"AllDataType"三个表，版本号为“1”。数据库类的 getVersion 方法和 getHelper 方法不需要实现，直接将数据库类设为虚类即可。

```
0. @Database(entities = {User.class, Book.class, AllDataType.class}, version = 1)
1. public abstract class BookStore extends OrmDatabase {
2. }
```

3. 构造数据表，即创建数据库实体类并配置对应的属性（如对应表的主键，外键等）。数据表必须与其所在的数据库在同一个模块中。

例如，定义了一个实体类 User.java，对应数据库内的表名为“user”；indices 为“firstName”和“lastName”两个字段建立了复合索引“name_index”，并且索引值是唯一的；“ignoreColumns”表示该字段不需要添加到“user”表的属性中。

```
0. @Entity(tableName = "user", ignoredColumns = {"ignoreColumn1", "ignoreColumn2"},
1.     indices = {@Index(value = {"firstName", "lastName"}, name = "name_index", unique = true)})
```

```
2.  public class User extends OrmObject {
3.      // 此处将 userId 设为了自增的主键。注意只有在数据类型为包装类型时，自增主键才能生效。
4.      @PrimaryKey(autoGenerate = true)
5.      private Integer userId;
6.      private String firstName;
7.      private String lastName;
8.      private int age;
9.      private double balance;
10.     private int ignoreColumn1;
11.     private int ignoreColumn2;
12.
13.     // 开发者自行添加字段的 getter 和 setter 方法。
14. }
```

**说明**

示例中的 getter & setter 的方法名为小驼峰格式，除了手写方法，IDE 中包含自动生成 getter 和 setter 方法的 Generate 插件。

- 当变量名的格式类似“firstName”时，getter 和 setter 方法名应为“getFirstName”和“setFirstName”。
- 当变量名的格式类似“mAge”，即第一个字母小写，第二个字母大写的格式时，getter 和 setter 方法名应为“getmAge”和“setmAge”。
- 当变量名格式类似“x”，即只有一个字母时，getter 和 setter 方法名应为“getX”和“setX”。

变量为 boolean 类型时，上述规则仍然成立，即“isFirstName”，“ismAge”，“isX”。

4. 使用对象数据操作接口 OrmContext 创建数据库。

例如，通过对象数据操作接口 OrmContext，创建一个别名为“BookStore”，数据库文件名为“BookStore.db”的数据库。如果数据库已经存在，执行以下代码不会重复创建。通过 context.getDatabaseDir()可以获取创建的数据库文件所在的目录。

```
0. DatabaseHelper helper = new DatabaseHelper(context); // context 入参类型为 ohos.app.Context，注意不要使用 slice.getContext()来获取 context，请直接传入 slice，否则会出现找不到类的报错。
1. OrmContext context = helper.getOrmContext("BookStore", "BookStore.db", BookStore.class);
```

5. （可选）数据库升降级。如果开发者有多个版本的数据库，通过设置数据库版本迁移类可以实现数据库版本升降级。

数据库版本升降级的调用示例如下。其中 BookStoreUpgrade 类也是一个继承了 OrmDatabase 的数据库类，与 BookStore 类的区别在于配置的版本号不同。

```
0. OrmContext context = helper.getOrmContext("BookStore", "BookStore.db", BookStoreUpgrade.class, new TestOrmMigration32(), new TestOrmMigration23(), new TestOrmMigration12(), new TestOrmMigration21());
```

TestOrmMigration12 的实现示例如下：

```
private static class TestOrmMigration12 extends OrmMigration {
    // 此处用于配置数据库版本迁移的开始版本和结束版本，super(startVersion, endVersion)即数据库版本号从 1 升到 2。
    public TestOrmMigration12() {super(1, 2); }
    @Override
    public void onMigrate(RdbStore store) {
        store.executeSql("ALTER TABLE `Book` ADD COLUMN `addColumn12` INTEGER");
    }
}
```

**说明**

数据库版本迁移类的起始版本和结束版本必须是连续的。

- 如果 BookStoreUpgrade 类的版本号配置为“2”，而当前 BookStore.db 的实际版本号为“1”时，TestOrmMigration12 类的 onMigrate 方法会自动被调用。开发者可在 onMigrate 方法中填写升级需要执行的 sql 语句。
- 如果 BookStoreUpgrade 的类版本号配置为“3”，而当前 BookStore.db 的实际版本号为“1”时，TestOrmMigration12，TestOrmMigration23 的 onMigrate 方法会自动被调用完成数据库升级。数据库版本降级同理。

6. 使用对象数据操作接口 OrmContext 对数据库进行增删改查、注册观察者、备份数据库等。

- 增加数据。例如，在数据库的名为“user”的表中，新建一个 User 对象并设置对象的属性。直接传入 OrmObject 对象的增加接口，只有在 flush()接口被调用后才会持久化到数据库中。

```
User user = new User();
user.setFirstName("Zhang");
user.setLastName("San");
user.setAge(29);
user.setBalance(100.51);
boolean isSuccessed = context.insert(user);
isSuccessed = context.flush();
```

- 更新或删除数据，分为两种情况：
  - 通过直接传入 OrmObject 对象的接口来更新数据，需要先从表中查到需要更新的 User 对象列表，然后修改对象的值，再调用更新接口持久化到数据库中。删除数据与更新数据的方法类似，只是不需要更新对象的值。

例如，更新“user”表中 age 为“29”的行，需要先查找“user”表中对应数据，得到一个 User 的列表。然后选择列表中需要更新的 User 对象（如第 0 个对象），设置需要更新的值，并调用 update 接口传入被更新的 User 对象。最后调用 flush 接口持久化到数据库中。

```
// 更新数据
```

```
2.  OrmPredicates predicates = context.where(User.class);
3.  predicates.equalTo("age",29);
4.  List<User> users = context.query(predicates);
5.  User user = users.get(0);
6.  user.setFirstName("Li");
7.  context.update(user);
8.  context.flush();
9.
10. // 删除数据
11. OrmPredicates predicates = context.where(User.class);
12. predicates.equalTo("age",29);
13. List<User> users = context.query(predicates);
14. User user = users.get(0);
15. context.delete(user);
16. context.flush();
```

- 通过传入谓词的接口来更新和删除数据，方法与 OrmObject 对象的接口类似，只是无需 flush 就可以持久化到数据库中。

```
1. ValuesBucket valuesBucket = new ValuesBucket();
2. valuesBucket.putInteger("age", 31);
3. valuesBucket.putString("firstName", "ZhangU");
4. valuesBucket.putString("lastName", "SanU");
5. valuesBucket.putDouble("balance", 300.51);
6. OrmPredicates update = context.where(User.class).equalTo("userId", 1);
7. context.update(update, valuesBucket);
```

- 查询数据。在数据库的“user”表中查询 lastName 为“San”的 User 对象列表，示例如下：

```
0. OrmPredicates query = context.where(User.class).equalTo("lastName", "San");
1. List<User> users = context.query(query);
```

- 注册观察者。

```
0. // 定义一个观察者类。
1. private class MyOrmObjectObserver implements OrmObjectObserver {
2.     @Override
3.     public void onChange(OrmContext changeContext,AllChangeToTarget subAllChange {
4.         // 用户可以在此处定义观察者行为
5.     }
6. }
7.
8. // 调用 registerEntityObserver 方法注册一个观察者 observer。
```

```
9.  MyOrmObjectObserver observer = new MyOrmObjectObserver();
10. context.registerEntityObserver("user", observer);
11.
12. // 当以下方法被调用，并 flush 成功时，观察者 observer 的 onChange 方法会被触发。其中，方法的入参必须为 User 类的对象。
13. public <T extends OrmObject> boolean insert(T object)
14. public <T extends OrmObject> boolean update(T object)
15. public <T extends OrmObject> boolean delete(T object)
```

- 备份数据库。其中原数据库名为“OrmBackUp.db”，备份数据库名为“OrmBackup001.db”。

```
0. OrmContext context = helper.getObjectContext("OrmBackup", "OrmBackup.db", BookStore.class);
1. context.backup("OrmBackup001.db");
2. context.close();
```

7. 删除数据库，例如删除 OrmBackup.db。

```
0. helper.deleteRdbStore("OrmBackup.db");
```

## 8.3 轻量级偏好数据库

### 8.3.1 概述

轻量级偏好数据库主要提供轻量级 Key-Value 操作，支持本地应用存储少量数据，数据存储在本地文件中，同时也加载在内存中的，所以访问速度更快，效率更高。轻量级偏好数据库属于非关系型数据库，不宜存储大量数据，经常用于操作键值对形式数据的场景。

#### 8.3.1.1 基本概念

- **Key-Value 数据库**

  一种以键值对存储数据的一种数据库，类似 Java 中的 map。Key 是关键字，Value 是值。

- **非关系型数据库**

  区别于关系数据库，不保证遵循 ACID（Atomic、Consistency、Isolation 及 Durability）特性，不采用关系模型来组织数据，数据之间无关系，扩展性好。

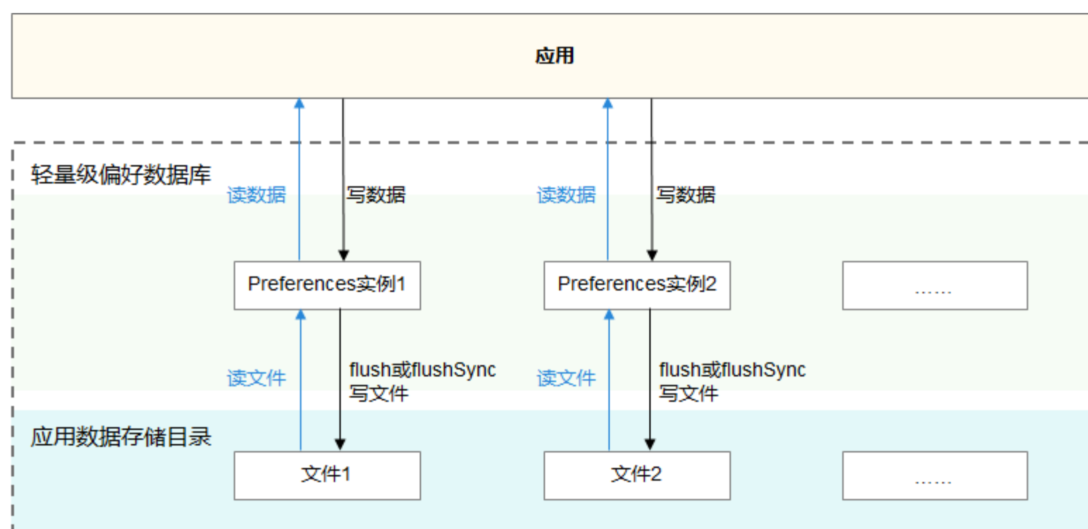
- **偏好数据**

  用户经常访问和使用的数据。

#### 8.3.1.2 运作机制

1. 本模块提供偏好型数据库的操作类，应用通过这些操作类完成数据库操作。
2. 借助 DatabaseHelper API，应用可以将指定文件的内容加载到 Preferences 实例，每个文件最多有一个 Preferences 实例，系统会通过静态容器将该实例存储在内存中，直到应用主动从内存中移除该实例或者删除该文件。
3. 获取到文件对应的 Preferences 实例后，应用可以借助 Preferences API，从 Preferences 实例中读取数据或者将数据写入 Preferences 实例，通过 flush 或者 flushSync 将 Preferences 实例持久化。

**图 1** 轻量级偏好数据库运作机制




#### 8.3.1.3 约束与限制

- Key 键为 String 类型，要求非空且大小不超过 80 个字符。
- 如果 Value 值为 String 类型，可以为空但是长度不超过 8192 个字符。
- 存储的数据量应该是轻量级的，建议存储的数据不超过一万条，否则会在内存方面产生较大的开销。

### 8.3.2 开发指导

#### 8.3.2.1 场景介绍

轻量级偏好数据库是轻量级存储，主要用于保存应用的一些常用配置，并不适合存储大量数据和频繁改变数据的场景。用户的数据保存在文件中，可以持久化的存储在设备上。需要注意的

是用户访问的实例包含文件所有数据，并一直加载在设备的内存中，并通过轻量级偏好数据库的 API 完成数据操作。

### 8.3.2.2 接口说明

轻量级偏好数据库向本地应用提供了操作偏好型数据库的 API，支持本地应用读写少量数据及观察数据变化。数据存储形式为键值对，键的类型为字符串型，值的存储数据类型包括整型、字符串型、布尔型、浮点型、长整型、字符串型 Set 集合。

**创建数据库**

通过数据库操作的辅助类可以获取到要操作的 Preferences 实例，用于进行数据库的操作。

| 类名 | 接口名 | 描述 |
| --- | --- | --- |
| DatabaseHelper | Preferences getPreferences(String name) | 获取文件对应的 Preferences 单实例，用于数据操作。 |

表 1 轻量级偏好数据库创建接口

**查询数据**

通过调用 Get 系列的方法，可以查询不同类型的数据。

| 类名 | 接口名 | 描述 |
| --- | --- | --- |
| Preferences | int getInt(String key, int defValue) | 获取键对应的 int 类型的值。 |
| Preferences | float getFloat(String key, float defValue) | 获取键对应的 float 类型的值。 |

表 2 轻量级偏好数据库查询接口

**插入数据**

通过 Put 系列的方法可以修改 Preferences 实例中的数据，通过 flush 或者 flushSync 将 Preferences 实例持久化。

| 类名 | 接口名 | 描述 |
| --- | --- | --- |
| Preferences | Preferences putInt(String key, int value) | 设置 Preferences 实例中键对应的 int 类型的值。 |
| Preferences | Preferences putString(String key, String value) | 设置 Preferences 实例中键对应的 String 类型的值。 |
| Preferences | void flush() | 将 Preferences 实例异步写入文件。 |
| Preferences | boolean flushSync() | 将 Preferences 实例同步写入文件。 |

**表 3** 轻量级偏好数据库插入接口

### 观察数据变化

轻量级偏好数据库还提供了一系列的接口变化回调，用于观察数据的变化。开发者可以通过重写 onChange 方法来定义观察者的行为。

| 类名 | 接口名 | 描述 |
| --- | --- | --- |
| Preferences | void registerObserver(PreferencesObserver preferencesObserver) | 注册观察者，用于观察数据变化。 |
| Preferences | void unRegisterObserver(PreferencesObserver preferencesObserver) | 注销观察者。 |
| Preferences.PreferencesObserver | void onChange(Preferences preferences, String key) | 观察者的回调方法，任意数据变化都会回调该方法。 |

**表 4** 轻量级偏好数据库接口变化回调

### 删除数据文件

通过调用以下两种接口，可以删除数据文件。

| 类名 | 接口名 | 描述 |
|---|---|---|
| DatabaseHelper | boolean deletePreferences(String name) | 删除文件和文件对应的 Preferences 单实例。 |
| DatabaseHelper | void removePreferencesFromCache(String name) | 删除文件对应的 Preferences 单实例。 |

**表 5** 轻量级偏好数据库删除接口

**移动数据库文件**

| 类名 | 接口名 | 描述 |
|---|---|---|
| DatabaseHelper | boolean movePreferences(Context sourceContext, String sourceName, String targetName) | 移动数据库文件。 |

**表 6** 轻量级偏好数据库移动接口

### 8.3.2.3 开发步骤

1. 准备工作，导入对轻量级偏好数据库 SDK 到开发环境。
2. 获取 Preferences 实例。

读取指定文件，将数据加载到 Preferences 实例，用于数据操作。

```
DatabaseHelper databaseHelper = new DatabaseHelper(context);// context 入参类型为 ohos.app.Context
String fileName = "name";
Preferences preferences = databaseHelper.getPreferences(fileName);
```

3. 从指定文件读取数据。

首先获取指定文件对应的 Preferences 实例，然后借助 Preferences API 读取数据。

java 接口 读取整型数据

```
int value = preferences.getInt("intKey", 0);
```

4. 将数据写入指定文件。

首先获取指定文件对应的 Preferences 实例，然后借助 Preferences API 将数据写入

Preferences 实例，通过 flush 或者 flushSync 将 Preferences 实例持久化。

异步：

```
1.  preferences.putInt("intKey", 3);
2.  preferences.putString("StringKey", "String value");
3.  preferences.flush();
```

同步：

```
1.  preferences.putInt("intKey", 3);
2.  preferences.putString("StringKey", "String value");
3.  preferences.flushSync();
```

5. 注册观察者。

开发者可以向 Preferences 实例注册观察者，观察者对象需实现 Preferences.PreferencesObserver 接口。flushSync()或 flush()执行后，该 Preferences 实例注册的所有观察者的 onChange()方法都会被回调。不再需要观察者时请注销。

```
1.  private class PreferencesChangeCounter implements Preferences.PreferencesObserver {
2.      final AtomicInteger notifyTimes = new AtomicInteger(0);
3.      @Override
4.      public void onChange(Preferences preferences, String key) {
5.          if ("intKey".equals(key)) {
6.              notifyTimes.incrementAndGet();
7.          }
8.      }
9.  }
10. // 向 preferences 实例注册观察者
11. PreferencesChangeCounter counter = new PreferencesChangeCounter();
12. preferences.registerObserver(counter);
13. // 修改数据 preferences.putInt("intKey", 3);
14. boolean result = preferences.flushSync();
15. // 修改数据后，onChange 方法会被回调，notifyTimes == 1
16. int notifyTimes = counter.notifyTimes.intValue();
17. // 向 preferences 实例注销观察者
18. preferences.unRegisterObserver(counter);
```

6. 移除 Preferences 实例。

从内存中移除指定文件对应的 Preferences 单实例。移除 Preferences 单实例时，应用不允许再使用该实例进行数据操作，否则会出现数据一致性问题。

```
1.  Context context = AppContext.getInstance().getCurrentAbility();
2.  DatabaseHelper databaseHelper = new DatabaseHelper(context);
```

```
3.  // 指定文件名称
4.  String fileName = "name";
5.  databaseHelper.removePreferencesFromCache(fileName);
```

7. 删除指定文件。

从内存中移除指定文件对应的 Preferences 单实例，并删除指定文件及其备份文件、损坏文件。删除指定文件时，应用不允许再使用该实例进行数据操作，否则会出现数据一致性问题

```
1.  Context context = AppContext.getInstance().getCurrentAbility();
2.  DatabaseHelper databaseHelper = new DatabaseHelper(context);
3.  // 指定文件名称
4.  String fileName = "name";
5.  boolean result = databaseHelper.deletePreferences(fileName);
```

8. 移动指定文件。

从源路径移动文件到目标路径。移动文件时，应用不允许再操作该文件数据，否则会出现数据一致性问题。

```
1.  Context targetContext = XXX;
2.  DatabaseHelper databaseHelper = new DatabaseHelper(targetContext);
3.  // 指定文件名称
4.  String srcFile = "srcFile";
5.  String targetFile = "targetFile";
6.  Context srcContext = XXX;
7.  boolean result = databaseHelper.movePreferences(srcContext,srcFile,targetFile);
```

## 8.4 分布式数据服务

### 8.4.1 概述

分布式数据服务（Distributed Data Service，DDS） 为应用程序提供不同设备间数据库数据分布式的能力。通过调用分布式数据接口，应用程序将数据保存到分布式数据库中。通过结合帐号、应用和数据库三元组，分布式数据服务对属于不同的应用的数据进行隔离，保证不同应

用之间的数据不能通过分布式数据服务互相访问。在通过可信认证的设备间，分布式数据服务支持应用数据相互同步，为用户提供在多种终端设备上一致的数据访问体验。

#### 8.4.1.1 基本概念

- **KV 数据模型**

  “KV 数据模型”是“Key-Value 数据模型”的简称，“Key-Value”即“键-值”。它是一种 NoSQL 类型数据库，其数据以键值对的形式进行组织、索引和存储。

  KV 数据模型适合不涉及过多数据关系和业务关系的业务数据存储，比 SQL 数据库存储拥有更好的读写性能，同时因在分布式场景中降低了数据库版本兼容和数据同步过程中冲突解决的复杂度而被广泛使用。分布式数据库也是基于 KV 数据模型，对外提供 KV 类型的访问接口。

- **分布式数据库事务性**

  分布式数据库事务支持本地事务（和传统数据库的事务概念一致）和同步事务，同步事务是指在设备之间同步数据时，是以本地事务为单位进行同步，一次本地事务的修改要么都同步成功，要么都同步失败。

- **分布式数据库一致性**

  在分布式场景中一般会涉及多个设备，组网内设备之间看到的数据是否一致称为分布式数据库的一致性。分布式数据库一致性可以分为**强一致性**、**弱一致性**和**最终一致性**。

  - **强一致性**：是指某一设备成功增、删、改数据后，组网内设备对该数据的读取操作都将得到更新后的值。
  - **弱一致性**：是指某一设备成功增、删、改数据后，组网内设备可能能读取到本次更新数据，也可能读取不到，不能保证在多长时间后每个设备的数据一定是一致的。
  - **最终一致性**：是指某一设备成功增、删、改数据后，组网内设备可能读取不到本次更新数据，但在某个时间窗口之后组网内设备的数据能够达到一致状态。

  强一致性对分布式数据的管理要求非常高，在服务器的分布式场景可能会遇到。因为移动终端设备的不常在线、以及无中心的特性，分布式数据服务不支持强一致，只支持最终一致性。

- **分布式数据库同步**

底层通信组件完成设备发现和认证，会通知上层应用程序（包括分布式数据服务）设备上线。收到设备上线的消息后分布式数据服务可以在两个设备之间建立加密的数据传输通道，利用该通道在两个设备之间进行数据同步。

分布式数据服务提供了两种同步模式：**手动同步**和**自动同步模式。手动同步模式**完全由应用程序调用接口来触发，并且支持指定同步的设备列表和同步模式（**PULL、PUSH 和 PULL_PUSH 三种同步模式**）。**自动同步模式**由分布式数据库来完成数据同步（同步时机包括设备上线、应用程序修改数据等），业务不感知同步操作。

- **单版本分布式数据库**

  单版本是指数据在本地保存是以单个 KV 条目为单位的方式保存，对每个 Key 最多只保存一个条目项，当数据在本地被用户修改时，不管它是否已经被同步出去，均直接在这个条目上进行修改。同步也以此为基础，按照它在本地被写入或更改的顺序将当前最新一次修改逐条同步至远端设备。

- **设备协同分布式数据库**

  设备协同分布式数据库建立在单版本分布式数据库之上，对应用程序存入的 KV 数据中的 Key 前面拼接了本设备的 DeviceID 标识符，这样能保证每个设备产生的数据严格隔离，底层按照设备的维度管理这些数据，设备协同分布式数据库支持以设备的维度查询分布式数据，但是不支持修改远端设备同步过来的数据。

- **分布式数据库冲突解决策略**

  分布式数据库多设备提交冲突场景，在给提交冲突做合并的过程中，如果多个设备同时修改了同一数据，则称这种场景为数据冲突。数据冲突采用默认冲突解决策略，基于提交时间戳，取时间戳较大的提交数据，当前不支持定制冲突解决策略。

- **数据库 Schema 化管理与谓词查询**

单版本数据库支持在创建和打开数据库时指定 Schema，数据库根据 Schema 定义感知 KV 记录的 Value 格式，以实现对 Value 值结构的检查，并基于 Value 中的字段实现索引建立和支持谓词查询。

- **分布式数据库备份能力**

  提供分布式数据库备份能力，业务通过设置 backup 属性为 true，可以触发分布式数据服务每日备份。当分布式数据库发生损坏，分布式数据服务会删除损坏数据库，并且从备份数据库中恢复上次备份的数据。如果不存在备份数据库，则创建一个新的数据库。同时支持加密数据库的备份能力。

#### 8.4.1.2 运作机制

分布式数据服务支撑 HarmonyOS 系统上应用程序数据库数据分布式管理，支持数据在相同帐号的多端设备之间相互同步，为用户在多端设备上提供一致的用户体验，分布式数据服务包含五部分：

- **服务接口**

  分布式数据服务提供专门的数据库创建、数据访问、数据订阅等接口给应用程序调用，接口支持 KV 数据模型，支持常用的数据类型，同时确保接口的兼容性、易用性和可发布性。

- **服务组件**

  服务组件负责服务内元数据管理、权限管理、加密管理、备份和恢复管理以及多用户管理等、同时负责初始化底层分布式 DB 的存储组件、同步组件和通信适配层。

- **存储组件**

  存储组件负责数据的访问、数据的缩减、事务、快照、数据库加密，以及数据合并和冲突解决等特性。

- **同步组件**

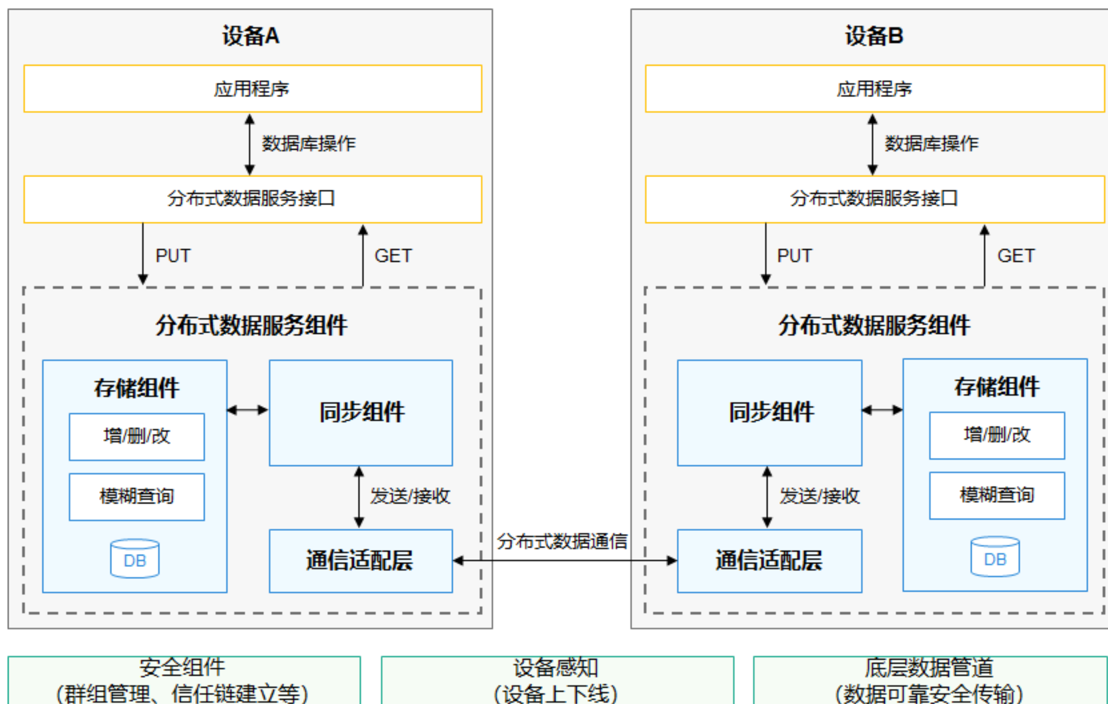
同步组件连结了存储组件与通信组件，其目标是保持在线设备间的数据库数据一致性，包括将本地产生的未同步数据同步给其他设备，接收来自其他设备发送过来的数据，并合并到本地设备中。

- **通信适配层**

通信适配层负责调用底层公共通信层的接口完成通信管道的创建、连接，接收设备上下线消息，维护已连接和断开设备列表的元数据，同时将设备上下线信息发送给上层同步组件，同步组件维护连接的设备列表，同步数据时根据该列表，调用通信适配层的接口将数据封装并发送给连接的设备。

应用程序通过调用分布式数据服务接口实现分布式数据库创建、访问、订阅功能，服务接口通过操作服务组件提供的能力，将数据存储至存储组件，存储组件调用同步组件实现将数据同步，同步组件使用通信适配层将数据同步至远端设备，远端设备通过同步组件接收数据，并更新至本端存储组件，通过服务接口提供给应用程序使用。

**图 1** 数据分布式运作示意图



#### 8.4.1.3 约束与限制

- 应用程序如需使用分布式数据服务完整功能，需要申请 ohos.permission.DISTRIBUTED_DATASYNC 权限。
- 分布式数据服务的数据模型仅支持 KV 数据模型，不支持外键、触发器等关系型数据库中的技术点。
- 分布式数据服务支持的 KV 数据模型规格：
  - 设备协同数据库，Key 最大支持 896Byte，Value 最大支持 4MB - 1Byte。
  - 单版本数据库，Key 最大支持 1KB，Value 最大支持 4MB - 1Byte。
  - 每个应用程序最多支持同时打开 16 个 KvStore。
- 由于支持的存储类型不完全相同等原因，分布式数据服务无法完全代替业务沙箱内数据库数据的存储功能，开发人员需要确定要做分布式同步的数据，把这些数据保存到分布式数据服务中。
- 分布式数据服务当前不支持应用程序自定义冲突解决策略。

### 8.4.2 开发指导

#### 8.4.2.1 场景介绍

分布式数据服务主要实现对用户设备中应用程序的数据内容的分布式同步。当设备 1 上的应用 A 在分布式数据库中增、删、改数据后，设备 2 上的应用 A 也可以获取到该数据库变化。可在分布式图库、信息、通讯录、文件管理器等场景中使用。

#### 8.4.2.2 接口说明

HarmonyOS 系统中的分布式数据服务模块为开发者提供下面几种功能：

| 功能分类 | 接口名称 | 描述 |
| --- | --- | --- |
| 分布式数据库创建、打开、关闭和删除。 | isCreateIfMissing() | 数据库不存在时是否创建。 |
| | setCreateIfMissing(boolean isCreateIfMissing) | 数据库不存在时是否创建。 |
| | isEncrypt() | 获取数据库是否加密。 |

| 功能分类 | 接口名称 | 描述 |
| --- | --- | --- |
| | setEncrypt(boolean isEncrypt) | 设置数据库是否加密。 |
| | getStoreType() | 获取分布式数据库的类型。 |
| | setStoreType(KvStoreType storeType) | 设置分布式数据库的类型。 |
| | KvStoreType.DEVICE_COLLABORATION | 设备协同分布式数据库类型。 |
| | KvStoreType.SINGLE_VERSION | 单版本分布式数据库类型。 |
| | getKvStore(Options options, String storeId) | 根据 Options 配置创建和打开标识符为 storeId 的分布式数据库。 |
| | closeKvStore(KvStore kvStore) | 关闭分布式数据库。 |
| | deleteKvStore(String storeId) | 删除分布式数据库。 |
| 分布式数据增、删、改、查。 | getStoreId() | 根据配置构造帐号键值数据库管理类实例。 |
| | putBoolean(String key, boolean value)<br>putInt(String key, int value)<br>putFloat(String key, float value)<br>putDouble(String key, double value)<br>putString(String key, String value)<br>putByteArray(String key, byte[] value)<br>putBatch(List<Entry> entries) | 插入和更新数据。 |
| | delete(String key)<br>deleteBatch(List<String> keys) | 删除数据。 |
| | getInt(String key)<br>getFloat(String key)<br>getDouble(String key)<br>getString(String key)<br>getByteArray(String key) | 查询数据。 |

| 功能分类 | 接口名称 | 描述 |
| --- | --- | --- |
| | getEntries(String keyPrefix) | |
| 分布式数据谓词查询。 | select()<br>reset()<br>equalTo(String field, int value)<br>equalTo(String field, long value)<br>equalTo(String field, double value)<br>equalTo(String field, String value)<br>equalTo(String field, boolean value)<br>notEqualTo(String field, int value)<br>notEqualTog(String field, long value)<br>notEqualTo(String field, boolean value)<br>notEqualTo(String field, String value)<br>notEqualTo(String field, double value)<br>greaterThan(String field, int value)<br>greaterThan(String field, long value)<br>greaterThan(String field, double value)<br>greaterThan(String field, String value)<br>lessThan(String field, int value)<br>lessThan(String field, long value)<br>lessThan(String field, double value)<br>lessThan(String field, String value)<br>greaterThanOrEqualTo(String field, int value)<br>greaterThanOrEqualTo(String field, long value)<br>greaterThanOrEqualTo(String field, double value)<br>greaterThanOrEqualTo(String field, String value)<br>lessThanOrEqualTo(String field, int value)<br>lessThanOrEqualTo(String field, long value)<br>lessThanOrEqualTo(String field, double value)<br>lessThanOrEqualTo(String field, String value)<br>isNull(String field)<br>orderByDesc(String field)<br>orderByAsc(String field)<br>limit(int number, int offset)<br>like(String field, String value) | 对于 Schema 数据库谓词查询数据。 |

| 功能分类 | 接口名称 | 描述 |
| --- | --- | --- |
| | unlike(String field, String value)<br>inInt(String field, List<Integer> valueList)<br>inLong(String field, List<Long> valueList)<br>inDouble(String field, List<Double> valueList)<br>inString(String field, List<String> valueList)<br>notInInt(String field, List<Integer> valueList)<br>notInLong(String field, List<Long> valueList)<br>notInDouble(String field, List<Double> valueList)<br>notInString(String field, List<String> valueList)<br>and()<br>or() | |
| 订阅分布式数据变化。 | subscribe(SubscribeType subscribeType,<br>KvStoreObserver observer) | 订阅数据库中数据的变化。 |
| 分布式数据同步。 | sync(List<String> deviceIdList, SyncMode mode) | 在手动模式下，触发数据库同步。 |

**表 1** 分布式数据服务关键 API 功能介绍

### 8.4.2.3 开发步骤

以单版本分布式数据库为例，说明开发步骤。

1. 根据配置构造分布式数据库管理类实例。

a. 根据应用上下文创建 KvManagerConfig 对象。

b. 创建分布式数据库管理器实例。

以下为创建分布式数据库管理器的代码示例：

```
1. Context context;
2. ...
3. KvManagerConfig config = new KvManagerConfig(context);
4. KvManager kvManager = KvManagerFactory.getInstance().createKvManager(config);
```

2. 获取/创建单版本分布式数据库。

c. 声明需要创建的单版本分布式数据库 ID 描述。

d. 创建单版本分布式数据库。

以下为创建单版本分布式数据库的代码示例：

```
1. Options CREATE = new Options();
2.
3. CREATE.setCreateIfMissing(true).setEncrypt(false).setKvStoreType(KvStoreType.SINGLE_VERSION);
4. String storeID = "testApp";
5. SingleKvStore singleKvStore = kvManager.getKvStore(CREATE, storeID);
```

2. 订阅分布式数据变化。

1. 客户端需要实现 KvStoreObserver 接口。

2. 构造并注册 KvStoreObserver 实例。

以下为订阅单版本分布式数据库所有（本地及远端）数据变化通知的代码示例：

```
1. class KvStoreObserverClient implements KvStoreObserver() {
2.     @Override
3.     public void onChange(ChangeNotification notification) {
4.         List<Entry> insertEntries = notification.getInsertEntries();
5.         List<Entry> updateEntries = notification.getUpdateEntries();
6.         List<Entry> deleteEntries = notification.getDeleteEntries();
7.     }
8. }
9.
10. KvStoreObserver kvStoreObserverClient = new KvStoreObserverClient();
11. singleKvStore.subscribe(SubscribeType.SUBSCRIBE_TYPE_ALL, kvStoreObserverClient);
```

3. 将数据写入单版本分布式数据库。

e. 构造需要写入单版本分布式数据库的 Key(键)和 Value(值)。

f. 将键值数据写入单版本分布式数据库。

以下为将字符串类型键值数据写入单版本分布式数据库的代码示例：

```
1. String key = "todayWeather";
2. String value = "Sunny";
3. singleKvStore.putString(key, value);
```

2. 查询单版本分布式数据库数据。

g. 构造需要从单版本分布式数据库快照中查询的 Key(键)。

h. 从单版本分布式数据库快照中获取数据。

以下为从单版本分布式数据库中查询字符串类型数据的代码示例：

```
1.  String key = "todayWeather";
2.  String value = singleKvStore.getString(key);
```

2. 同步数据到其他设备。

   1. 获取已连接的设备列表。

   2. 选择同步方式进行数据同步。

   以下为单版本分布式数据库进行数据同步的代码示例，其中同步方式为 PUSH_ONLY：

```
1.  List<DeviceInfo> deviceInfoList = kvManager.getConnectedDevicesInfo(DeviceFilterStrategy.NO_FILTER);
2.  List<String> deviceIdList = new ArrayList<>();
3.  for (DeviceInfo deviceInfo : deviceInfoList) {
4.      deviceIdList.add(deviceInfo.getId());
5.  }
6.  singleKvStore.sync(deviceIdList, SyncMode.PUSH_ONLY);
```

3. 关闭单版本分布式数据库。以下为关闭单版本分布式数据库的代码示例：

```
1.  kvManager.closeKvStore(singleKvStore);
```

4. 删除单版本分布式数据库。以下为删除单版本分布式数据库的代码示例：

```
1.  kvManager.deleteKvStore(storeID);
```

## 8.5 分布式文件服务

### 8.5.1 概述

分布式文件服务能够为用户设备中的应用程序提供多设备之间的文件共享能力，支持相同帐号下同一应用文件的跨设备访问，应用程序可以不感知文件所在的存储设备，能够在多个设备之间无缝获取文件。

#### 8.5.1.1 基本概念

- **分布式文件**

分布式文件是指依赖于分布式文件系统，分散存储在多个用户设备上的文件，应用间的分布式文件目录互相隔离，不同应用的文件不能互相访问。

- **文件元数据**

  文件元数据是用于描述文件特征的数据，包含文件名，文件大小，创建、访问、修改时间等信息。

### 8.5.1.2 运作机制

分布式文件服务采用无中心节点的设计，每个设备都存储一份全量的文件元数据和本设备上产生的分布式文件，元数据在多台设备间互相同步，当应用需要访问分布式文件时，分布式文件服务首先查询本设备上的文件元数据，获取文件所在的存储设备，然后对存储设备上的分布式文件服务发起文件访问请求，将文件内容读取到本地。

**图 1** 分布式文件服务运作示意图

### 8.5.1.3 约束与限制

- 应用程序如需使用分布式文件服务完整功能，需要申请 ohos.permission.DISTRIBUTED_DATASYNC 权限。
- 多个设备需要打开蓝牙，连接同一 WLAN 局域网，登录相同华为帐号才能实现文件的分布式共享。
- 存在多设备并发写的场景下，为了保证文件独享，开发者需要对文件进行加锁保护。
- 应用访问分布式文件时，如果文件所在设备离线，文件不能访问。
- 非持锁情况下，并发写冲突时，后一次会覆盖前一次。
- 网络情况差时，访问存储在远端的分布式文件时，可能会长时间不返回或返回失败，应用需要考虑这种场景的处理。
- 当两台设备有同名文件时，同步元数据时会产生冲突，分布式文件服务根据时间戳将文件按创建的先后顺序重命名，为避免此场景，建议应用在文件名上做设备区分，例如，deviceID+时间戳。

## 8.5.2 开发指导

### 8.5.2.1 场景介绍

应用可以通过分布式文件服务实现多个设备间的文件共享，设备 1 上的应用 A 创建了分布式文件 a，设备 2 上的应用 A 能够通过分布式文件服务读写设备 1 上的文件 a。

### 8.5.2.2 接口说明

分布式文件兼容 POSIX 文件操作接口，应用使用 Context.getDistributedDir()接口获取目录后，可以直接使用 libc 或 JDK 访问分布式文件。

| 接口名 | 描述 |
| --- | --- |
| Context.getDistributedDir() | 获取文件的分布式目录 |

**表 1** 分布式文件服务 API 接口功能介绍

### 8.5.2.3 开发步骤

应用可以通过 Context.getDistributedDir()接口获取属于自己的分布式目录，然后通过 libc 或 JDK 接口，在该目录下创建、删除、读写文件或目录。

1. 设备 1 上的应用 A 创建文件 hello.txt，并写入内容"Hello World"。

```
Context context;
... // context 初始化
File distDir = context.getDistributedDir();
String filePath = distDir + File.separator + "hello.txt";
FileWriter fileWriter = new FileWriter(filePath,true);
fileWriter.write("Hello World");
fileWriter.close();
```

2. 设备 2 上的应用 A 通过 Context.getDistributedDir()接口获取分布式目录。
3. 设备 2 上的应用 A 读取文件 hello.txt。

```
FileReader fileReader = new FileReader(filePath);
```

```
2.  char[] buffer = new char[1024];
3.  fileReader.read(buffer);
4.  fileReader.close();
5.  System.out.println(buffer);
```

## 8.6 融合搜索

### 8.6.1 概述

HarmonyOS 融合搜索为开发者提供搜索引擎级的全文搜索能力，可支持应用内搜索和系统全局搜索，为用户提供更加准确、高效的搜索体验。

#### 8.6.1.1 基本概念

- **全文索引**

  记录字或词的位置和次数等属性，建立的倒排索引。

- **全文搜索**

  通过全文索引进行匹配查找结果的一种搜索引擎技术。

- **全局搜索**

  可以在系统全局统一的入口进行的搜索行为。

- **全局搜索应用**

  HarmonyOS 上提供全局搜索入口的应用，一般为桌面下拉框或悬浮搜索框。

- **索引源应用**

  通过融合搜索索引接口对其数据建立索引的应用。

- **可搜索配置**

  每个索引源应用应该提供一个包括应用包名、是否支持全局搜索等信息的可搜索实体，以便全局搜索应用发起搜索。

- **群组**

  经过认证的可信设备圈，可从账号模块获取群组 ID。

- **索引库**

  一种搜索引擎的倒排索引库，包含多个索引文件的整个目录构成一个索引库。

- **索引域**

  索引数据的字段名，比如一张图片有文件名、存储路径、大小、拍摄时间等，文件名就是其中的一个索引域。

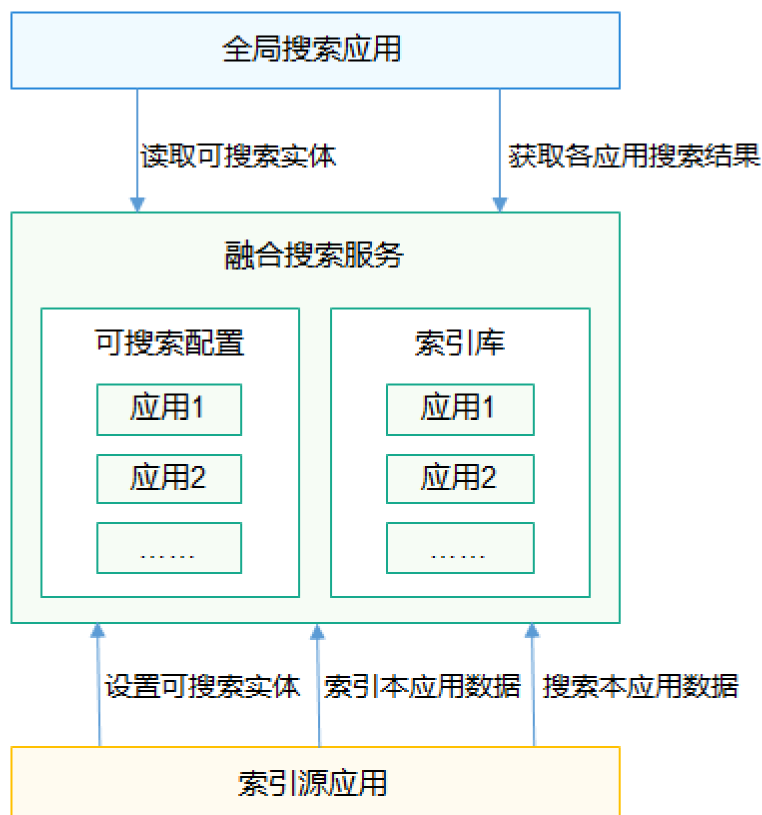
- **索引属性**

  描述索引域的信息，包括索引类型、是否为主键、是否存储、是否支持分词等。

### 8.6.1.2 运作机制

索引源应用通过融合搜索接口设置可搜索实体，并为其数据内容构建全文索引。全局搜索应用接收用户发起的搜索请求，遍历支持全局搜索的可搜索实体，解析用户输入并构造查询条件，最后通过融合搜索接口获取各应用搜索结果。

**图 1** 融合搜索运作示意图



#### 8.6.1.3 约束与限制

- 构建索引或者发起搜索前，索引源应用必须先设置索引属性，并且必须有且仅有一个索引域设置为主键，且主键索引域不能分词，索引和搜索都会使用到索引属性。
- 索引源应用的数据发生变动时，开发者应同步通过融合搜索索引接口更新索引，以保证索引和应用原始数据的一致性。
- 批量创建、更新、删除索引时，应控制单次待索引内容大小，建议分批创建索引，防止内存溢出。
- 分页搜索和分组搜索应控制每页返回结果数量，防止内存溢出。
- 构建和搜索本机索引时，应该使用提供的 SearchParameter.DEFAULT_GROUP 作为群组 ID，分布式索引使用通过账号模块获取的群组 ID。
- 搜索时需先创建搜索会话，并务必在搜索结束时关闭搜索会话，释放内存资源。
- 使用融合搜索服务接口需要在“config.json”配置文件中添加“ohos.permission.ACCESS_SEARCH_SERVICE”权限。
- 搜索时的 SearchParamter.DEVICE_ID_LIST 必须与创建索引时的 deviceId 一致。

### 8.6.2 开发指导

#### 8.6.2.1 场景介绍

索引源应用，一般为有持久化数据的应用，可以通过融合搜索接口为其应用数据建立索引，并配置全局搜索可搜索实体，帮助用户通过全局搜索应用查找本应用内的数据。应用本身也提供搜索框时，也可直接在应用内部通过融合搜索接口实现全文搜索功能。

#### 8.6.2.2 接口说明

HarmonyOS 中的融合搜索为开发者提供以下几种能力，详见 API 参考。

| 类名 | 接口名 | 描述 |
|---|---|---|
| SearchAbility | public List<IndexData> insert(String groupId, String bundleName, List<IndexData> indexDataList) | 索引插入 |

| 类名 | 接口名 | 描述 |
| --- | --- | --- |
| | public List<IndexData> update(String groupId, String bundleName, List<IndexData> indexDataList) | 索引更新 |
| | public List<IndexData> delete(String groupId, String bundleName, List<IndexData> indexDataList) | 索引删除 |
| SearchSession | public int getSearchHitCount(String queryJsonStr) | 搜索命中结果数量 |
| | public List<IndexData> search(String queryJsonStr, int start, int limit) | 分页搜索 |
| | public List<Recommendation> groupSearch(String queryJsonStr, int groupLimit) | 分组搜索 |

**表 1** 融合搜索接口功能介绍

### 8.6.2.3 开发步骤

1. 实例化 SearchAbility，连接融合搜索服务。

```
1.  SearchAbility searchAbility = new SearchAbility(context);
2.  CountDownLatch lock = new CountDownLatch(1);
3.  // 连接服务
4.  searchAbility.connect(new ServiceConnectCallback() {
5.      @Override
6.      public void onConnect() {
7.          lock.countDown();
8.      }
9.
10.     @Override
11.     public void onDisconnect() {
12.     }
13. });
14. // 等待回调，最长等待时间可自定义
15. lock.await(3000, TimeUnit.MILLISECONDS);
16. // 连接失败可重试
```

2. 设置索引属性。

```
1. // 构造索引属性
2. List<IndexForm> indexFormList = new ArrayList<>();
3. IndexForm primaryKey = new IndexForm("id", IndexType.NO_ANALYZED, true, true, false); // 主键，不分词
4. indexFormList.add(primaryKey);
5. IndexForm title = new IndexForm("title", IndexType.ANALYZED, false, true, true); // 分词
6. indexFormList.add(title);
7. IndexForm tagType = new IndexForm("tag_type", IndexType.SORTED, false, true, false); // 分词，同时支持排序、分组
8. indexFormList.add(tagType);
9. IndexForm ocrText = new IndexForm("ocr_text", IndexType.SORTED_NO_ANALYZED, false, true, false); // 支持排序、分组，不分词，所
   以也支持范围搜索
10. indexFormList.add(ocrText);
11. IndexForm dateTaken = new IndexForm("datetaken", IndexType.LONG, false, true, false); // 支持排序和范围查询
12. indexFormList.add(dateTaken);
13. IndexForm bucketId = new IndexForm("bucket_id", IndexType.INTEGER, false, true, false); // 支持排序和范围查询
14. indexFormList.add(bucketId);
15. IndexForm latitude = new IndexForm("latitude", IndexType.FLOAT, false, true, false); // 支持范围搜索
16. indexFormList.add(latitude);
17. IndexForm longitude = new IndexForm("longitude", IndexType.DOUBLE, false, true, false); // 支持范围搜索
18. indexFormList.add(longitude);
19.
20. // 设置索引属性
21. int result = searchAbility.setIndexForm(bundleName, 1, indexFormList);
22. // 设置失败可重试
```

3. 插入索引。

```
1. // 构建索引数据
2. List<IndexData> indexDataList = new ArrayList<>();
3. for (int i = 0; i < 5; i++) {
4.     IndexData indexData = new IndexData();
5.     indexData.put("id", "id" + i);
6.     indexData.put("title", "title" + i);
7.     indexData.put("tag_type", "tag_type" + i);
8.     indexData.put("ocr_text", "ocr_text" + i);
9.     indexData.put("datetaken", System.currentTimeMillis());
10.    indexData.put("bucket_id", i);
11.    indexData.put("latitude", i / 5.0 * 180);
12.    indexData.put("longitude", i / 5.0 * 360);
```

```
13.         indexDataList.add(indexData);
14.     }
15.     // 插入索引
16.     List<IndexData> failedList = searchAbility.insert(SearchParameter.DEFAULT_GROUP, bundleName, indexDataList);
17.     // 失败的记录可以持久化，稍后重试
```

4. 构建查询。

```
1.  // 构建查询
2.  JSONObject jsonObject = new JSONObject();
3.
4.  // SearchParameter.QUERY 对应用户输入，搜索域应该都是分词的
5.  // 这里假设用户输入是“天空”，要在"title", "tag_type"这两个域上发起搜索
6.  JSONObject query = new JSONObject();
7.  query.put("天空", new JSONArray(Arrays.asList("title", "tag_type")));
8.  jsonObject.put(SearchParameter.QUERY, query);
9.
10. // SearchParameter.FILTER_CONDITION 对应的 JSONArray 里可以添加搜索条件
11. // 对于索引库里的一条索引，JSONArray 下的每个 JSONObject 指定的条件都必须满足才会命中，JSONObject 里的条件组合满足其中一个，
    这个 JSONObject 指定的条件即可满足
12. JSONArray filterCondition = new JSONArray();
13. // 第一个条件，一个域上可能取多个值
14. JSONObject filter1 = new JSONObject();
15. filter1.put("bucket_id", new JSONArray(Arrays.asList(0, 1, 2))); // 一条索引在"bucket_id"的取值为 0 或 1 或 2 就能命中
16. filter1.put("id", new JSONArray(Arrays.asList(0, 1))); // 或者在"id"的取值为 0 或者 1 也可以命中
17. filterCondition.put(filter1);
18. // 第二个条件，一个值可能在多个域上命中
19. JSONObject filter2 = new JSONObject();
20. filter2.put("tag_type", new JSONArray(Arrays.asList("白云")));
21. filter2.put("ocr_text", new JSONArray(Arrays.asList("白云"))); // 一条索引只要在"tag_type"或者"ocr_text"上命中"白云"就能命中
22. filterCondition.put(filter2);
23. jsonObject.put(SearchParameter.FILTER_CONDITION, filterCondition); // 一条索引要同时满足第一和第二个条件才能命中
24.
25. // SearchParameter.DEVICE_ID_LIST 对应设备 ID，匹配指定设备 ID 的索引才会命中
26. JSONObject deviceId = new JSONObject();
27. deviceId.put("device_id", new JSONArray(Arrays.asList("localDeviceId")));
28. jsonObject.put(SearchParameter.DEVICE_ID_LIST, deviceId);
29.
30. // 可以在支持范围搜索的索引域上发起范围搜索，一条索引在指定域的值都落在对应的指定范围才会命中
31. JSONObject latitude = new JSONObject();
```

```
32.  latitude.put(SearchParameter.LOWER, -40.0f); // inclusive
33.  latitude.put(SearchParameter.UPPER, 40.0.f); // inclusive
34.  jsonObject.put("latitude", latitude); // 纬度必须在[-40.0f, 40.0f]
35.  JSONObject longitude new JSONObject();
36.  longitude.put(SearchParameter.LOWER, -90.0); // inclusive
37.  longitude.put(SearchParameter.UPPER, 90.0);   // inclusive
38.  jsonObject.put("longitude", longitude); // 经度必须在[-90.0, 90.0]
39.
40.  // SearchParameter.ORDER_BY 对应搜索结果的排序，排序字段通过 SearchParameter.ASC 和 SearchParameter.DESC
41.  // 指定搜索结果在这个字段上按照升序、降序排序，这里填充字段的顺序是重要的，比如这里两个索引之间会先在"id"
42.  // 字段上升序排序，只有在"id"上相同时，才会继续在"datetaken"上降序排序，以此类推
43.  JSONObject order = new JSONObject();
44.  order.put("id", SearchParameter.ASC);
45.  order.put("title", SearchParameter.ASC);
46.  order.put("datetaken", SearchParameter.DESC);
47.  jsonObject.put(SearchParameter.ORDER_BY, order);
48.
49.  // SearchParameter.GROUP_FIELD_LIST 对应的群组搜索的域，调用 gruopSearch 接口需要指定
50.  jsonObject.put(SearchParameter.GROUP_FIELD_LIST, new JSONArray(Arrays.asList("tag_type", "ocr_text")));
51.
52.  // 得到查询字符串
53.  String queryJsonStr = jsonObject.toString();
54.  // 构建的 json 字符串如下:
55.  /**
56.  {
57.      "SearchParameter.QUERY": {
58.          "天空": [
59.              "title",
60.              "tag_type"
61.          ]
62.      },
63.      "SearchParameter.FILTER_CONDITION": [
64.          {
65.              "bucket_id": [
66.                  0,
67.                  1,
68.                  2
69.              ],
```

```
70.                 "id": [
71.                     0,
72.                     1
73.                 ]
74.             },
75.             {
76.                 "tag_type": [
77.                     "白云"
78.                 ],
79.                 "ocr_text": [
80.                     "白云"
81.                 ]
82.             }
83.         ],
84.         "SearchParameter.DEVICE_ID_LIST": {
85.             "device_id": [
86.                 "localDeviceId"
87.             ]
88.         },
89.         "latitude": {
90.             "SearchParameter.LOWER": -40.0,
91.             "SearchParameter.UPPER": 40.0
92.         },
93.         "longitude": {
94.             "SearchParameter.LOWER": -90.0,
95.             "SearchParameter.UPPER": 90.0
96.         },
97.         "SearchParameter.ORDER_BY": {
98.             "id": "ASC",
99.             "title": "ASC",
100.            "datetaken": "DESC"
101.        },
102.        "SearchParameter.GROUP_FIELD_LIST": [
103.            "tag_type",
104.            "ocr_text"
105.        ]
106.    }
107.    **/
```

5. 开始搜索会话，发起搜索。

```
1.  // 开始搜索会话
2.  SearchSession searchSession = searchAbility.beginSearch(SearchParameter.DEFAULT_GROUP, bundleName);
3.  if (searchSession == null) {
4.      return;
5.  }
6.  try {
7.      int hit = searchSession.getSearchHitCount(queryJsonStr); // 获取总命中数
8.      int batch = 50; // 每页最多返回 50 个结果
9.      for (int i = 0; i < hit; i += batch) {
10.         List<IndexData> result = searchSession.search(queryJsonStr, i, batch);
11.         ...
12.         // 处理 IndexData
13.     }
14.     int groupLimit = 10; // 每个分组域上最多返回 10 个分组结果
15.     List<Recommendation> result = searchSession.groupSearch(queryJsonStr, groupLimit);
16.     // 处理 Recommendation
17.     for (Recommendation recommendation : result) {
18.         HiLog.info(LABEL, "field: %{public}s, value: %{public}s, count: %{public}d", recommedation.getField(),
    recommendation.getValue(), recommendation.getCount());
19.     }
20. } finally {
21.     // 释放资源
22.     searchAbility.endSearch(SearchParameter.DEFAULT_GROUP, bundleName, searchSession);
23. }
```

## 8.7 数据存储管理

### 8.7.1 概述

数据存储管理指导开发者基于 HarmonyOS 进行存储设备（包含本地存储、SD 卡、U 盘等）的数据存储管理能力的开发，包括获取存储设备列表，获取存储设备视图等。

#### 8.7.1.1 基本概念

- **数据存储管理**

  数据存储管理包括了获取存储设备列表，获取存储设备视图，同时也可以按照条件获取对应的存储设备视图信息。

- **设备存储视图**

  存储设备的抽象表示，提供了接口访问存储设备的自身信息。

#### 8.7.1.2 运作机制

用统一的视图结构可以表示各种存储设备，该视图结构的内部属性会因为设备的不同而不同。

每个存储设备可以抽象成两部分，一部分是存储设备自身信息区域，一部分是用来真正存放数据的区域。

**图 1** 存储设备视图

## 8.7.2 开发指导

### 8.7.2.1 场景介绍

为了给用户展示存储设备信息，开发者可以使用数据存储管理接口获取存储设备视图信息，也可以根据用户提供的文件名获取对应存储设备的视图信息。

### 8.7.2.2 开放能力介绍

数据存储管理为开发者提供下面几种功能，具体的 API 参考。

| 功能分类 | 类名 | 接口名 | 描述 |
| --- | --- | --- | --- |
| 查询设备视图 | ohos.data.usage.DataUsage | getVolumes() | 获取当前用户可用的设备列表视图。 |
| | | getVolume(File file) | 获取存储该文件的存储设备视图。 |
| | | getVolume(Context context, Uri uri) | 获取该 URI 对应文件所在的存储设备视图。 |
| | | getDiskMountedStatus() | 获取默认存储设备的挂载状态。 |
| | | getDiskMountedStatus(File path) | 获取存储该文件设备的挂载状态。 |
| | | isDiskPluggable() | 默认存储设备是否为可插拔设备。 |
| | | isDiskPluggable(File path) | 存储该文件的设备是否为可插拔设备。 |
| | | isDiskEmulated() | 默认存储设备是否为虚拟设备。 |
| | | isDiskEmulated(File path) | 存储该文件的设备是否为虚拟设备。 |
| 查询设备视图属性 | ohos.data.usage.Volume | isEmulated() | 该设备是否是虚拟存储设备。 |
| | | isPluggable() | 该设备是否支持插拔。 |
| | | getDescription() | 获取设备描述信息。 |
| | | getState() | 获取设备挂载状态。 |
| | | getVolUuid() | 获取设备唯一标识符。 |

**表 1** 数据存储管理接口功能介绍

### 8.7.2.3 开发步骤

#### 查询设备视图

调用查询设备视图接口。

```
1.  // 获取默认存储设备挂载状态
2.  MountState status = DataUsage.getDiskMountedStatus();
3.  // 获取存储设备列表
4.  Optional<List<Volume>> list = DataUsage.getVolumes();
5.  // 默认存储设备是否为可插拔设备
6.  boolean pluggable = DataUsage.isDiskPluggable();
```

#### 查询设备视图属性

1. 调用查询设备视图接口获取某个设备视图 Volume。
2. 调用 Volume 的接口即可查询视图属性。

```
1.  // 获取 example.txt 文件所在的存储设备的视图属性
2.  Optional<Volume> volume = DataUsage.getVolume(new File("/sdcard/example.txt"));
3.  volume.ifPresent(theVolume -> {
4.              System.out.println(theVolume.isEmulated());
5.              System.out.println(theVolume.isPluggable());
6.              System.out.println(theVolume.getDescription());
7.              System.out.println(theVolume.getVolUuid());
8.      }
9.  );
```