HarmonyOS开发中级篇

Outline

- Ability Kit
- Network Kit
- ArkData
- ArkWeb

AbilityKit

AbilityKit

Ability Kit（程序框架服务）提供了应用程序开发和运行的应用模型，是系统为开发者提供的应用程序所需能力的抽象提炼，它提供了应用程序必备的组件和运行机制。有了应用模型，开发者可以基于一套统一的模型进行应用开发，使应用开发更简单、高效。

使用场景

- 应用的多Module开发：应用可通过不同类型的Module（HAP、HAR、HSP）来实现应用的功能开发。其中，HAP用于实现应用的功能和特性，HAR与HSP用于实现代码和资源的共享。
- 应用内的交互：应用内的不同组件之间可以相互跳转。比如，在支付应用中，通过入口UIAbility组件启动收付款UIAbility组件。
- 应用间的交互：当前应用可以启动其他应用，来完成某个任务或操作。比如，启动浏览器应用来打开网站、启动文件应用来浏览或编辑文件等。
- 应用的跨设备流转：通过应用的跨端迁移和多端协同，获得更好的使用体验。比如，在平板上播放的视频，迁移到智慧屏继续播放。

能力范围

- 提供应用进程创建和销毁、应用生命周期调度能力。
- 提供应用组件运行入口、应用组件生命周期调度、组件间交互等能力。
- 提供应用上下文环境、系统环境变化监听等能力。
- 提供应用流转能力。
- 提供多包机制、共享包、应用信息配置等能力，详见应用程序包概述。
- 提供程序访问控制能力，详见访问控制概述。
- 提供安全密码自动填充能力，详见密码自动填充服务概述。

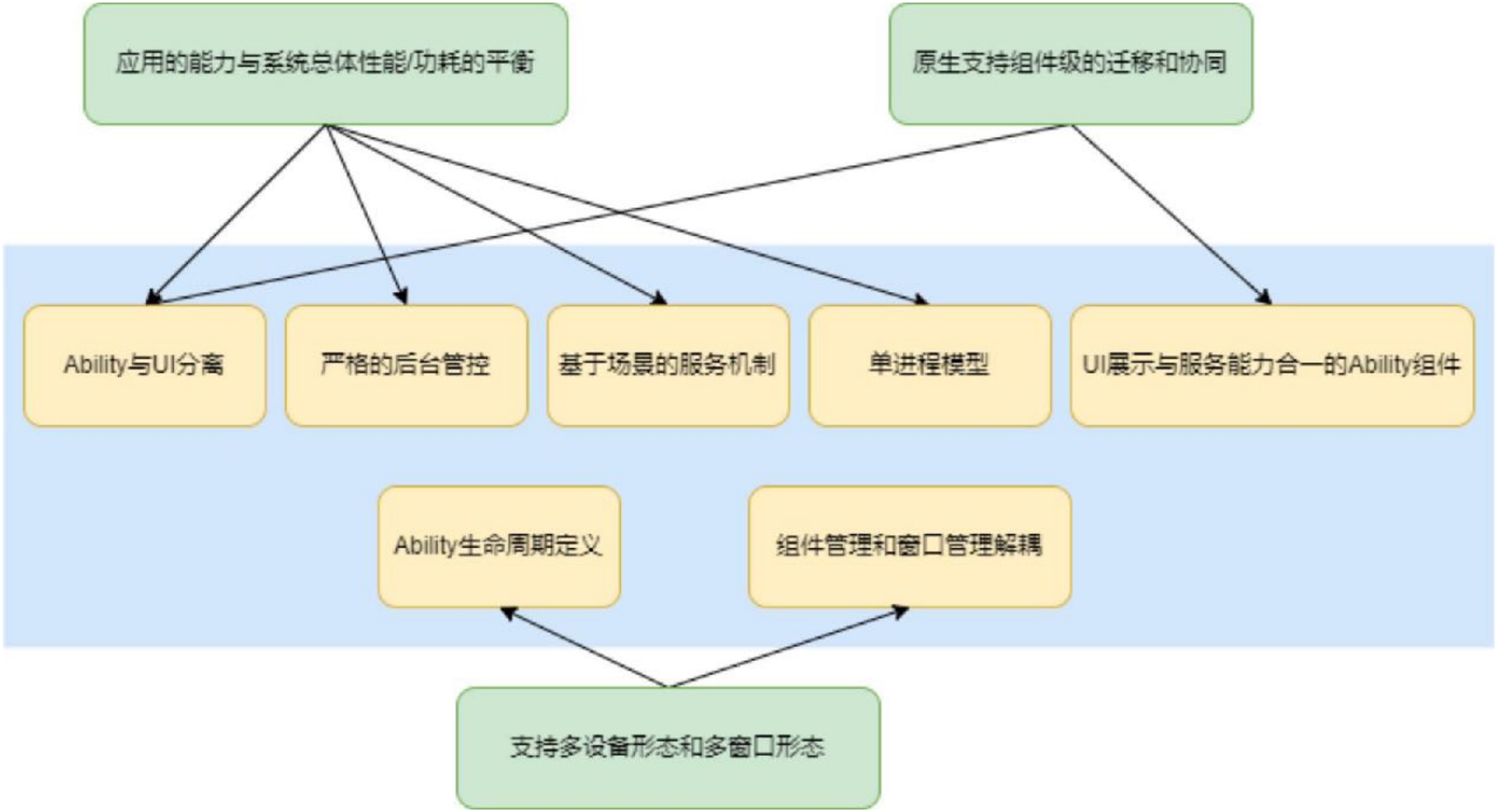
亮点/特征

- **为复杂应用而设计**
  - 多个应用组件共享同一个ArkTS引擎（运行ArkTS语言的虚拟机）实例，应用组件之间可以方便的共享对象和状态，同时减少复杂应用运行对内存的占用。
  - 采用面向对象的开发方式，使得复杂应用代码可读性高、易维护性好、可扩展性强。
  - 提供模块化能力开发的支持。
- **原生支持应用组件级的跨端迁移和多端协同**

  Stage模型实现了应用组件与UI解耦。
  - 在跨端迁移场景下，系统在多设备的应用组件之间迁移数据/状态后，UI便可利用ArkUI的声明式特点，通过应用组件中保存的数据/状态恢复用户界面，便捷实现跨端迁移。
  - 在多端协同场景下，应用组件具备组件间通信的RPC调用能力，天然支持跨设备应用组件的交互。
- **支持多设备和多窗口形态**

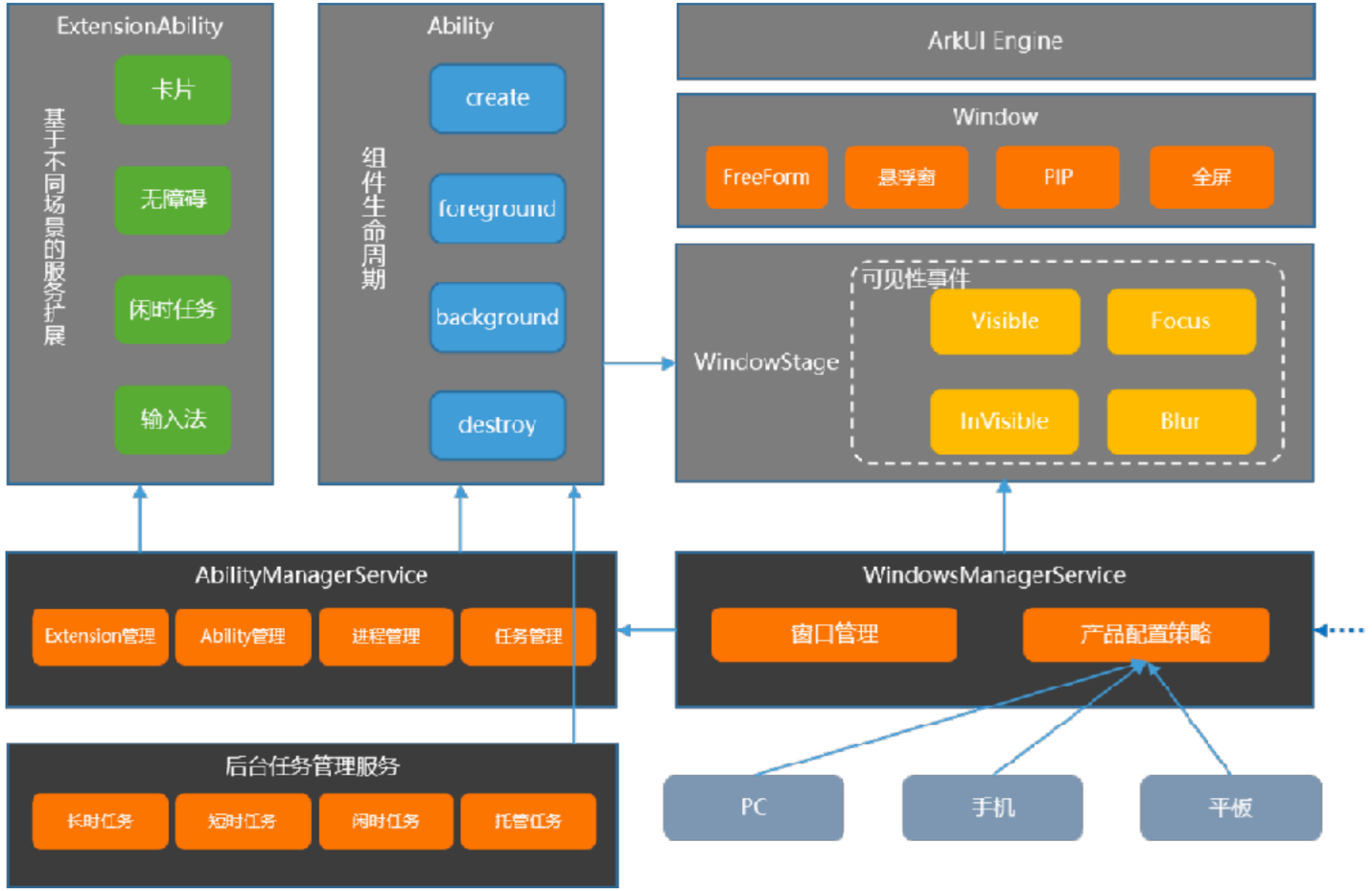
  应用组件管理和窗口管理在架构层面解耦。
  - 便于系统对应用组件进行裁剪（无屏设备可裁剪窗口）。
  - 便于系统扩展窗口形态。
  - 在多设备（如桌面设备和移动设备）上，应用组件可使用同一套生命周期。
- **平衡应用能力和系统管控成本**

  Stage模型重新定义应用能力的边界，平衡应用能力和系统管控成本。
  - 提供特定场景（如服务卡片、输入法）的应用组件，以便满足更多的使用场景。
  - 规范化后台进程管理：为保障用户体验，Stage模型对后台应用进程进行了有序治理，应用程序不能随意驻留在后台，同时应用后台行为受到严格管理，防止恶意应用行为。

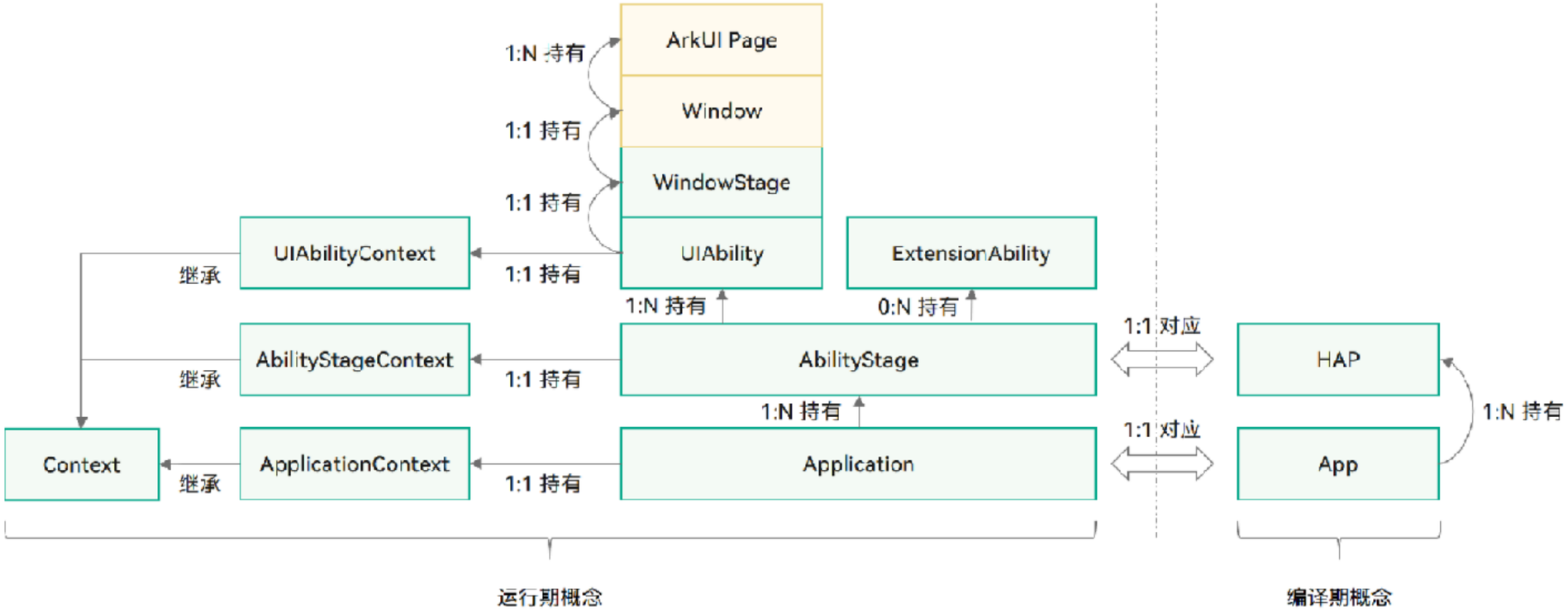


Stage模型

Stage 模型：是为了解决FA模型无法解决的开发场景问题，方便开发者更加方便地开发出分布式环境下的复杂应用。自 API 9 新增的模型。是日后长期演进的模型。
Stage模型将Ability分为PageAbility和ExtensionAbility两大类，其中ExtensionAbility又被扩展为ServiceExtensionAbility、FormExtensionAbility、DataShareExtensionAbility等一系列 ExtensionAbility，以便满足更多的使用场景。


ExtensionAbility
基于不同场景的服务扩展
卡片
无障碍
闲时任务
输入法
Ability
组件生命周期
create
foreground
background
destroy
ArkUI Engine
Window
FreeForm
悬浮窗
PIP
全屏
WindowStage
可见性事件
Visible
Focus
InVisible
Blur
AbilityManagerService
Extension管理
Ability管理
进程管理
任务管理
WindowsManagerService
窗口管理
产品配置策略
后台任务管理服务
长时任务
短时任务
闲时任务
托管任务
PC
手机
平板

特点



应用组件

UIAbility & ExtensionAbility

启动应用内的UIAbility组件

- UIAbility是系统调度的最小单元。在设备内的功能模块之间跳转时，会涉及到启动特定的UIAbility，包括应用内的其他UIAbility、或者其他应用的UIAbility（例如启动三方支付UIAbility）。

- 本文主要介绍启动应用内的UIAbility组件的方式。应用间的组件跳转详见应用间跳转。

  - 启动应用内的UIAbility

  - 启动应用内的UIAbility并获取返回结果

  - 启动UIAbility的指定页面

启动应用内的UIAbility

当一个应用内包含多个UIAbility时，存在应用内启动UIAbility的场景。例如在支付应用中从入口UIAbility启动收付款UIAbility。
假设应用中有两个UIAbility：EntryAbility和FuncAbility（可以在同一个Module中，也可以在不同的Module中），需要从EntryAbility的页面中启动FuncAbility。

- 在EntryAbility中，通过调用startAbility()方法启动UIAbility，want为UIAbility实例启动的入口参数，其中bundleName为待启动应用的Bundle名称，abilityName为待启动的Ability名称，moduleName在待启动的UIAbility属于不同的Module时添加，parameters为自定义信息参数。示例中的context的获取方式请参见获取UIAbility的上下文信息。

```
import { common, Want } from '@kit.AbilityKit';
import { hilog } from '@kit.PerformanceAnalysisKit';
import { BusinessError } from '@kit.BasicServicesKit';

const TAG: string = '[Page_UIAbilityComponentsInteractive]';
const DOMAIN_NUMBER: number = 0xFF00;

@Entry
@Component
struct Page_UIAbilityComponentsInteractive {
  private context = getContext(this) as common.UIAbilityContext;

  build() {
    Column() {
      //...
      List({ initialIndex: 0 }) {
        ListItem() {
          Row() {
            //...
          }
          .onClick(() => {
            // context为Ability对象的成员，在非Ability对象内部调用需要
            // 将Context对象传递过去
            let wantInfo: Want = {
              deviceId: '', // deviceId为空表示本设备
              bundleName: 'com.samples.stagemodelabilitydevelop',
              moduleName: 'entry', // moduleName非必选
              abilityName: 'FuncAbilityA',
              parameters: {
                // 自定义信息
                info: '来自EntryAbility Page_UIAbilityComponentsInteractive页面'
              },
            };
            // context为调用方UIAbility的UIAbilityContext
            this.context.startAbility(wantInfo).then(() => {
              hilog.info(DOMAIN_NUMBER, TAG, 'startAbility success.');
            }).catch((error: BusinessError) => {
              hilog.error(DOMAIN_NUMBER, TAG, 'startAbility failed.');
            });
          })
        }
        //...
      }
      //...
    }
    //...
  }
}
```

启动应用内的UIAbility

- 在FuncAbility的onCreate()或者onNewWant()生命周期回调文件中接收EntryAbility传递过来的参数。

```
import { AbilityConstant, UIAbility, Want } from '@kit.AbilityKit';

export default class FuncAbilityA extends UIAbility {
  onCreate(want: Want, launchParam: AbilityConstant.LaunchParam): void {
    // 接收调用方UIAbility传过来的参数
    let funcAbilityWant = want;
    let info = funcAbilityWant?.parameters?.info;
  }
  //...
}
```

说明
在被拉起的FuncAbility中，可以通过获取传递过来的want参数的parameters来获取拉起方UIAbility的PID、Bundle Name等信息。

- 在FuncAbility业务完成之后，如需要停止当前UIAbility实例，在FuncAbility中通过调用terminateSelf()方法实现。

```
import { common } from '@kit.AbilityKit';
import { hilog } from '@kit.PerformanceAnalysisKit';

const TAG: string = '[Page_FromStageModel]';
const DOMAIN_NUMBER: number = 0xFF00;

@Entry
@Component
struct Page_FromStageModel {
  build() {
    Column() {
      //...
      Button('FuncAbilityB')
        .onClick(() => {
          let context: common.UIAbilityContext = getContext(this) as common.UIAbilityContext; // UIAbilityContext
          // context为需要停止的UIAbility实例的AbilityContext
          context.terminateSelf((err) => {
            if (err.code) {
              hilog.error(DOMAIN_NUMBER, TAG, `Failed to terminate self. Code is ${err.code}, message is ${err.message}`);
              return;
            }
          });
        })
    }
    //...
  }
}
```

说明
调用terminateSelf()方法停止当前UIAbility实例时，默认会保留该实例的快照（Snapshot），即在最近任务列表中仍然能查看到该实例对应的任务。如不需要保留该实例的快照，可以在其对应UIAbility的module.json5配置文件中，将abilities标签的removeMissionAfterTerminate字段配置为true。

- 如需要关闭应用所有的UIAbility实例，可以调用ApplicationContext的killAllProcesses()方法实现关闭应用所有的进程。

启动应用内的UIAbility并获取返回结果

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

- 在EntryAbility中，调用startAbilityForResult()接口启动FuncAbility，异步回调中的data用于接收FuncAbility停止自身后返回给EntryAbility的信息。示例中的context的获取方式请参见获取UIAbility的上下文信息。

```
import { common, Want } from '@kit.AbilityKit';
import { hilog } from '@kit.PerformanceAnalysisKit';
import { promptAction } from '@kit.ArkUI';
import { BusinessError } from '@kit.BasicServicesKit';

const TAG: string = '[Page_UIAbilityComponentsInteractive]';
const DOMAIN_NUMBER: number = 0xFF00;

@Entry
@Component
struct Page_UIAbilityComponentsInteractive {
  build() {
    Column() {
      //...
      List({ initialIndex: 0 }) {
        ListItem() {
          Row() {
            //...
          }
          .onClick(() => {
            let context: common.UIAbilityContext = getContext(this) as common.UIAbilityContext; // UIAbilityContext
            const RESULT_CODE: number = 1001;
            let want: Want = {
              deviceId: '', // deviceId为空表示本设备
              bundleName: 'com.samples.stagemodelabilitydevelop',
              moduleName: 'entry', // moduleName非必选
              abilityName: 'FuncAbilityA',
              parameters: {
                // 自定义信息
                info: '来自EntryAbility UIAbilityComponentsInteractive页面'
              }
            };
            context.startAbilityForResult(want).then((data) => {
              if (data?.resultCode === RESULT_CODE) {
                // 解析被调用方UIAbility返回的信息
                let info = data.want?.parameters?.info;
                hilog.info(DOMAIN_NUMBER, TAG, JSON.stringify(info) ?? '');
                if (info !== null) {
                  promptAction.showToast({
                    message: JSON.stringify(info)
                  });
                }
              }
              hilog.info(DOMAIN_NUMBER, TAG, JSON.stringify(data.resultCode) ?? '');
            }).catch((err: BusinessError) => {
              hilog.error(DOMAIN_NUMBER, TAG, `Failed to start ability for result. Code is ${err.code}, message is ${err.message}`);
            });
          })
        }
        //...
      }
      //...
    }
    //...
  }
}
```

启动应用内的UIAbility并获取返回结果

- 在FuncAbility停止自身时，需要调用terminateSelfWithResult()方法，入参abilityResult为FuncAbility需要返回给EntryAbility的信息。

```
import { common } from '@kit.AbilityKit';
import { hilog } from '@kit.PerformanceAnalysisKit';

const TAG: string = '[Page_FuncAbilityA]';
const DOMAIN_NUMBER: number = 0xFF00;

@Entry
@Component
struct Page_FuncAbilityA {
  build() {
    Column() {
      //...
      List({ initialIndex: 0 }) {
        ListItem() {
          Row() {
            //...
          }
          .onClick(() => {
            let context: common.UIAbilityContext = getContext(this) as common.UIAbilityContext; // UIAbilityContext
            const RESULT_CODE: number = 1001;
            let abilityResult: common.AbilityResult = {
              resultCode: RESULT_CODE,
              want: {
                bundleName: 'com.samples.stagemodelabilitydevelop',
                moduleName: 'entry', // moduleName非必选
                abilityName: 'FuncAbilityB',
                parameters: {
                  info: '来自FuncAbility Index页面'
                },
              },
            };
            context.terminateSelfWithResult(abilityResult, (err) => {
              if (err.code) {
                hilog.error(DOMAIN_NUMBER, TAG, `Failed to terminate self with result. Code is ${err.code}, message is ${err.message}`);
                return;
              }
            });
          })
        }
        //...
      }
      //...
    }
    //...
  }
}
```

启动应用内的UIAbility并获取返回结果

- FuncAbility停止自身后，EntryAbility通过startAbilityForResult()方法回调接收被FuncAbility返回的信息，RESULT_CODE需要与前面的数值保持一致。

```
import { common, Want } from '@kit.AbilityKit';
import { hilog } from '@kit.PerformanceAnalysisKit';
import { promptAction } from '@kit.ArkUI';
import { BusinessError } from '@kit.BasicServicesKit';

const TAG: string = '[Page_UIAbilityComponentsInteractive]';
const DOMAIN_NUMBER: number = 0xFF00;

@Entry
@Component
struct Page_UIAbilityComponentsInteractive {
  build() {
    Column() {
      //...
      List({ initialIndex: 0 }) {
        ListItem() {
          Row() {
            //...
          }
          .onClick(() => {
            let context: common.UIAbilityContext = getContext(this) as common.UIAbilityContext; // UIAbilityContext
            const RESULT_CODE: number = 1001;

            let want: Want = {
              deviceId: '', // deviceId为空表示本设备
              bundleName: 'com.samples.stagemodelabilitydevelop',
              moduleName: 'entry', // moduleName非必选
              abilityName: 'FuncAbilityA',
              parameters: {
                // 自定义信息
                info: '来自EntryAbility UIAbilityComponentsInteractive页面'
              }
            };
            context.startAbilityForResult(want).then((data) => {
              if (data?.resultCode === RESULT_CODE) {
                // 解析被调用方UIAbility返回的信息
                let info = data.want?.parameters?.info;
                hilog.info(DOMAIN_NUMBER, TAG, JSON.stringify(info) ?? '');
                if (info !== null) {
                  promptAction.showToast({
                    message: JSON.stringify(info)
                  });
                }
              }
              hilog.info(DOMAIN_NUMBER, TAG, JSON.stringify(data.resultCode) ?? '');
            }).catch((err: BusinessError) => {
              hilog.error(DOMAIN_NUMBER, TAG, `Failed to start ability for result. Code is ${err.code}, message is ${err.message}`);
            });
          })
        }
        //...
      }
      //...
    }
    //...
  }
}
```

启动UIAbility的指定页面

- 一个UIAbility可以对应多个页面，在不同的场景下启动该UIAbility时需要展示不同的页面，例如从一个UIAbility的页面中跳转到另外一个UIAbility时，希望启动目标UIAbility的指定页面。

- UIAbility的启动分为两种情况：UIAbility冷启动和UIAbility热启动。
  - UIAbility冷启动：指的是UIAbility实例处于完全关闭状态下被启动，这需要完整地加载和初始化UIAbility实例的代码、资源等。
  - UIAbility热启动：指的是UIAbility实例已经启动并在前台运行过，由于某些原因切换到后台，再次启动该UIAbility实例，这种情况下可以快速恢复UIAbility实例的状态。

调用方UIAbility指定启动页面

调用方UIAbility启动另外一个UIAbility时，通常需要跳转到指定的页面。例如FuncAbility包含两个页面（Index对应首页，Second对应功能A页面），此时需要在传入的want参数中配置指定的页面路径信息，可以通过want中的parameters参数增加一个自定义参数传递页面跳转信息。示例中的context的获取方式请参见获取UIAbility的上下文信息。

```
import { common, Want } from '@kit.AbilityKit';
import { hilog } from '@kit.PerformanceAnalysisKit';
import { BusinessError } from '@kit.BasicServicesKit';

const TAG: string = '[Page_UIAbilityComponentsInteractive]';
const DOMAIN_NUMBER: number = 0xFF00;

@Entry
@Component
struct Page_UIAbilityComponentsInteractive {
  build() {
    Column() {
      //...
      List({ initialIndex: 0 }) {
        ListItem() {
          Row() {
            //...
          }
          .onClick(() => {
            let context: common.UIAbilityContext = getContext(this) as common.UIAbilityContext; // UIAbilityContext
            let want: Want = {
              deviceId: '', // deviceId为空表示本设备
              bundleName: 'com.samples.stagemodelabilityinteraction',
              moduleName: 'entry', // moduleName非必选
              abilityName: 'FuncAbility',
              parameters: { // 自定义参数传递页面信息
                router: 'funcA'
              }
            };
            // context为调用方UIAbility的UIAbilityContext
            context.startAbility(want).then(() => {
              hilog.info(DOMAIN_NUMBER, TAG, 'Succeeded in starting ability.');
            }).catch((err: BusinessError) => {
              hilog.error(DOMAIN_NUMBER, TAG, `Failed to start ability. Code is ${err.code}, message is ${err.message}`);
            });
          })
        }
        //...
      }
      //...
    }
    //...
  }
}
```

目标UIAbility冷启动

目标UIAbility冷启动时，在目标UIAbility的onCreate()生命周期回调中，接收调用方传过来的参数。然后在目标UIAbility的onWindowStageCreate()生命周期回调中，解析调用方传递过来的want参数，获取到需要加载的页面信息url，传入windowStage.loadContent()方法。

```
import { AbilityConstant, Want, UIAbility } from '@kit.AbilityKit';
import { hilog } from '@kit.PerformanceAnalysisKit';
import { window, UIContext } from '@kit.ArkUI';

const DOMAIN_NUMBER: number = 0xFF00;
const TAG: string = '[EntryAbility]';

export default class EntryAbility extends UIAbility {
  funcAbilityWant: Want | undefined = undefined;
  uiContext: UIContext | undefined = undefined;

  onCreate(want: Want, launchParam: AbilityConstant.LaunchParam): void {
    // 接收调用方UIAbility传过来的参数
    this.funcAbilityWant = want;
  }

  onWindowStageCreate(windowStage: window.WindowStage): void {
    // Main window is created, set main page for this ability
    hilog.info(DOMAIN_NUMBER, TAG, '%{public}s', 'Ability onWindowStageCreate');
    // Main window is created, set main page for this ability
    let url = 'pages/Index';
    if (this.funcAbilityWant?.parameters?.router && this.funcAbilityWant.parameters.router === 'funcA') {
      url = 'pages/Page_ColdStartUp';
    }
    windowStage.loadContent(url, (err, data) => {
      // ...
    });
  }
}
```

目标UIAbility热启动

- 在应用开发中，会遇到目标UIAbility实例之前已经启动过的场景，这时再次启动目标UIAbility时，不会重新走初始化逻辑，只会直接触发onNewWant()生命周期方法。为了实现跳转到指定页面，需要在onNewWant()中解析参数进行处理。

- 例如短信应用和联系人应用配合使用的场景。

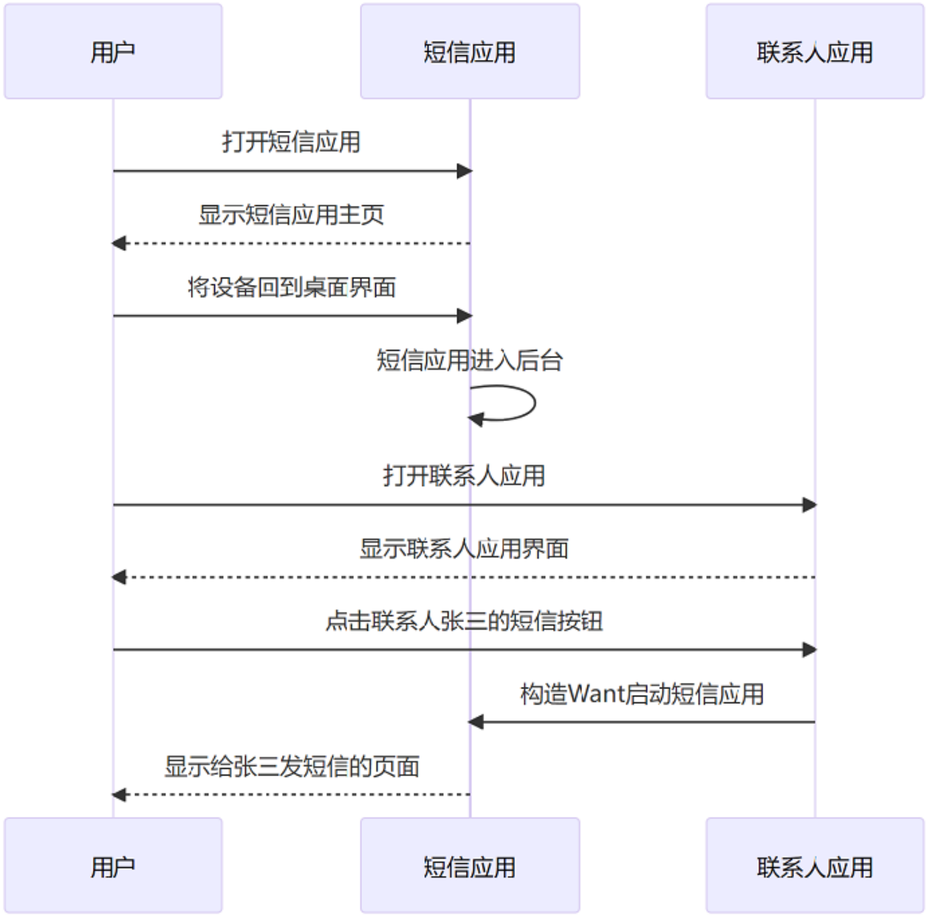
- 用户先打开短信应用，短信应用的UIAbility实例启动，显示短信应用的主页。

- 用户将设备回到桌面界面，短信应用进入后台运行状态。

- 用户打开联系人应用，找到联系人张三。

- 用户点击联系人张三的短信按钮，会重新启动短信应用的UIAbility实例。

- 由于短信应用的UIAbility实例已经启动过了，此时会触发该UIAbility的onNewWant()回调，而不会再走onCreate()和onWindowStageCreate()等初始化逻辑。



目标UIAbility热启动

开发代码

- 冷启动短信应用的UIAbility实例时，在onWindowStageCreate()生命周期回调中，通过调用getUIContext()接口获取UI上下文实例UIContext对象。

```
import { hilog } from '@kit.PerformanceAnalysisKit';
import { Want, UIAbility } from '@kit.AbilityKit';
import { window, UIContext } from '@kit.ArkUI';

const DOMAIN_NUMBER: number = 0xFF00;
const TAG: string = '[EntryAbility]';

export default class EntryAbility extends UIAbility {
  funcAbilityWant: Want | undefined = undefined;
  uiContext: UIContext | undefined = undefined;

  // ...

  onWindowStageCreate(windowStage: window.WindowStage): void {
    // Main window is created, set main page for this ability
    hilog.info(DOMAIN_NUMBER, TAG, '%{public}s', 'Ability onWindowStageCreate');
    let url = 'pages/Index';
    if (this.funcAbilityWant?.parameters?.router && this.funcAbilityWant.parameters.router === 'funcA') {
      url = 'pages/Page_ColdStartUp';
    }

    windowStage.loadContent(url, (err, data) => {
      if (err.code) {
        return;
      }

      let windowClass: window.Window;
      windowStage.getMainWindow((err, data) => {
        if (err.code) {
          hilog.error(DOMAIN_NUMBER, TAG, `Failed to obtain the main window. Code is ${err.code}, message is ${err.message}`);
          return;
        }
        windowClass = data;
        this.uiContext = windowClass.getUIContext();
      });
      hilog.info(DOMAIN_NUMBER, TAG, 'Succeeded in loading the content. Data: %{public}s', JSON.stringify(data) ?? '');
    });
  }
}
```

- 在短信应用UIAbility的onNewWant()回调中解析调用方传递过来的want参数，通过调用UIContext中的getRouter()方法获取Router对象，并进行指定页面的跳转。此时再次启动该短信应用的UIAbility实例时，即可跳转到该短信应用的UIAbility实例的指定页面。

```
import { AbilityConstant, Want, UIAbility } from '@kit.AbilityKit';
import { hilog } from '@kit.PerformanceAnalysisKit';
import type { Router, UIContext } from '@kit.ArkUI';
import type { BusinessError } from '@kit.BasicServicesKit';

const DOMAIN_NUMBER: number = 0xFF00;
const TAG: string = '[EntryAbility]';

export default class EntryAbility extends UIAbility {
  funcAbilityWant: Want | undefined = undefined;
  uiContext: UIContext | undefined = undefined;
  // ...
  onNewWant(want: Want, launchParam: AbilityConstant.LaunchParam): void {
    if (want?.parameters?.router && want.parameters.router === 'funcA') {
      let funcAUrl = 'pages/Page_HotStartUp';
      if (this.uiContext) {
        let router: Router = this.uiContext.getRouter();
        router.pushUrl({
          url: funcAUrl
        }).catch((err: BusinessError) => {
          hilog.error(DOMAIN_NUMBER, TAG, `Failed to push url. Code is ${err.code}, message is ${err.message}`);
        });
      }
    }
  }
}
```

说明

当被调用方UIAbility组件启动模式设置为multiton启动模式时，每次启动都会创建一个新的实例，那么onNewWant()回调就不会被用到。

实践案例： StageAbilityDemo

Network Kit

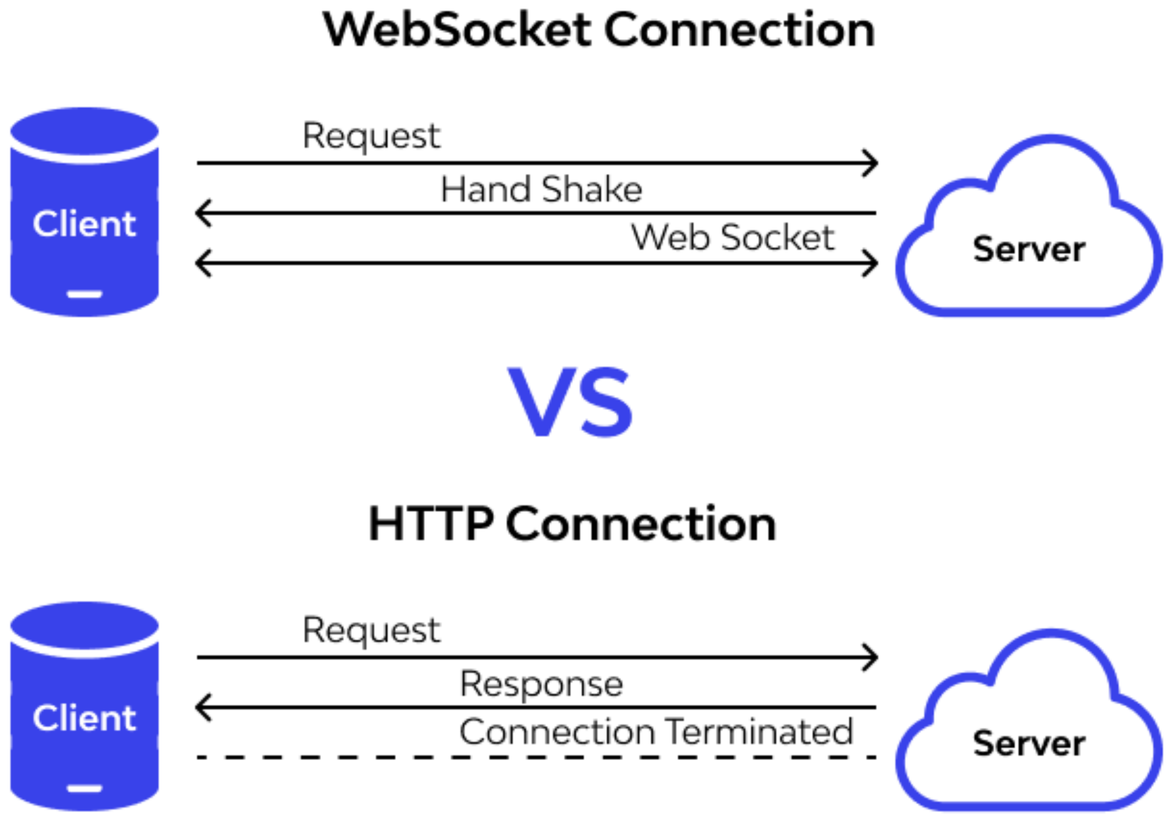
网络管理 - Network Kit

Network Kit（网络服务）主要提供以下功能：

- HTTP数据请求：通过HTTP发起一个数据请求。
- WebSocket连接：使用WebSocket建立服务器与客户端的双向连接。
- Socket连接：通过Socket进行数据传输。
- 网络连接管理：网络连接管理提供管理网络一些基础能力，包括WiFi/蜂窝/Ethernet等多网络连接优先级管理、网络质量评估、订阅默认/指定网络连接状态变化、查询网络连接信息、DNS解析等功能。
- MDNS管理：MDNS即多播DNS（Multicast DNS），提供局域网内的本地服务添加、移除、发现、解析等能力。

- 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
  - ohos.permission.GET_NETWORK_INFO
    - 获取网络连接信息。
  - ohos.permission.INTERNET
    - 允许程序打开网络套接字，进行网络连接。



HTTP vs WebSocket

接口名	描述
createHttp()	创建一个http请求。
request()	根据URL地址，发起HTTP网络请求。
requestInStream()[10+]	根据URL地址，发起HTTP网络请求并返回流式响应。
destroy()	中断请求任务。
on(type: 'headersReceive')	订阅HTTP Response Header 事件。
off(type: 'headersReceive')	取消订阅HTTP Response Header 事件。
once('headersReceive')[8+]	订阅HTTP Response Header 事件，但是只触发一次。
on('dataReceive')[10+]	订阅HTTP流式响应数据接收事件。
off('dataReceive')[10+]	取消订阅HTTP流式响应数据接收事件。
on('dataEnd')[10+]	订阅HTTP流式响应数据接收完毕事件。
off('dataEnd')[10+]	取消订阅HTTP流式响应数据接收完毕事件。
on('dataReceiveProgress')[10+]	订阅HTTP流式响应数据接收进度事件。
off('dataReceiveProgress')[10+]	取消订阅HTTP流式响应数据接收进度事件。
on('dataSendProgress')[11+]	订阅HTTP网络请求数据发送进度事件。
off('dataSendProgress')[11+]	取消订阅HTTP网络请求数据发送进度事件。

HTTP数据请求

request接口开发步骤

- 从@kit.NetworkKit中导入http命名空间。
- 调用createHttp()方法，创建一个HttpRequest对象。
- 调用该对象的on()方法，订阅http响应头事件，此接口会比request请求先返回。可以根据业务需要订阅此消息。
- 调用该对象的request()方法，传入http请求的url地址和可选参数，发起网络请求。
- 按照实际业务需要，解析返回结果。
- 调用该对象的off()方法，取消订阅http响应头事件。
- 当该请求使用完毕时，调用destroy()方法主动销毁。

request接口开发步骤

```
// 引入包名
import { http } from '@kit.NetworkKit';
import { BusinessError } from '@kit.BasicServicesKit';

// 每一个httpRequest对应一个HTTP请求任务，不可复用
let httpRequest = http.createHttp();
// 用于订阅HTTP响应头，此接口会比request请求先返回。可以根据业务需要订阅此消息
// 从API 8开始，使用on('headersReceive', Callback)替代on('headerReceive', AsyncCallback)。 8+
httpRequest.on('headersReceive', (header) => {
  console.info('header: ' + JSON.stringify(header));
});
httpRequest.request(
  // 填写HTTP请求的URL地址，可以带参数也可以不带参数。URL地址需要开发者自定义。请求的参数可以在extraData中指定
  "EXAMPLE_URL",
  {
    method: http.RequestMethod.POST, // 可选，默认为http.RequestMethod.GET
    // 开发者根据自身业务需要添加header字段
    header: {
      'Content-Type': 'application/json'
    },
    // 当使用POST请求时此字段用于传递请求体内容，具体格式与服务端协商确定
    extraData: "data to send",
    expectDataType: http.HttpDataType.STRING, // 可选，指定返回数据的类型
    usingCache: true, // 可选，默认为true
    priority: 1, // 可选，默认为1
    connectTimeout: 60000, // 可选，默认为60000ms
    readTimeout: 60000, // 可选，默认为60000ms
    usingProtocol: http.HttpProtocol.HTTP1_1, // 可选，协议类型默认值由系统自动指定
    usingProxy: false, // 可选，默认不使用网络代理，自API 10开始支持该属性
    caPath:'/path/to/cacert.pem', // 可选，默认使用系统预制证书，自API 10开始支持该属性
    clientCert: { // 可选，默认不使用客户端证书，自API 11开始支持该属性
      certPath: '/path/to/client.pem', // 默认不使用客户端证书，自API 11开始支持该属性
      keyPath: '/path/to/client.key', // 若证书包含Key信息，传入空字符串，自API 11开始支持该属性
      certType: http.CertType.PEM, // 可选，默认使用PEM，自API 11开始支持该属性
      keyPassword: "passwordToKey" // 可选，输入key文件的密码，自API 11开始支持该属性
    },
    multiFormDataList: [ // 可选，仅当Header中，'content-Type'为'multipart/form-data'时生效，自API 11开始支持该属性
      {
        name: "Part1", // 数据名，自API 11开始支持该属性
        contentType: 'text/plain', // 数据类型，自API 11开始支持该属性
        data: 'Example data', // 可选，数据内容，自API 11开始支持该属性
        remoteFileName: 'example.txt' // 可选，自API 11开始支持该属性
      }, {
        name: "Part2", // 数据名，自API 11开始支持该属性
        contentType: 'text/plain', // 数据类型，自API 11开始支持该属性
        // data/app/el2/100/base/com.example.myapplication/haps/entry/files/fileName.txt
        filePath: `${getContext(this).filesDir}/fileName.txt`, // 可选，传入文件路径，自API 11开始支持该属性
        remoteFileName: 'fileName.txt' // 可选，自API 11开始支持该属性
      }
    ]
  }, (err: BusinessError, data: http.HttpResponse) => {
    if (!err) {
      // data.result为HTTP响应内容，可根据业务需要进行解析
      console.info('Result:' + JSON.stringify(data.result));
      console.info('code:' + JSON.stringify(data.responseCode));
      // data.header为HTTP响应头，可根据业务需要进行解析
      console.info('header:' + JSON.stringify(data.header));
      console.info('cookies:' + JSON.stringify(data.cookies)); // 8+
      // 当该请求使用完毕时，调用destroy方法主动销毁
      httpRequest.destroy();
    } else {
      console.error('error:' + JSON.stringify(err));
      // 取消订阅HTTP响应头事件
      httpRequest.off('headersReceive');
      // 当该请求使用完毕时，调用destroy方法主动销毁
      httpRequest.destroy();
    }
  }
);
```

requestInStream接口开发步骤

- 从@kit.NetworkKit中导入http命名空间。
- 调用createHttp()方法，创建一个HttpRequest对象。
- 调用该对象的on()方法，可以根据业务需要订阅HTTP响应头事件、HTTP流式响应数据接收事件、HTTP流式响应数据接收进度事件和HTTP流式响应数据接收完毕事件。
- 调用该对象的requestInStream()方法，传入http请求的url地址和可选参数，发起网络请求。
- 按照实际业务需要，可以解析返回的响应码。
- 调用该对象的off()方法，取消订阅响应事件。
- 当该请求使用完毕时，调用destroy()方法主动销毁。

requestInStream接口开发步骤

```
// 引入包名
import { http } from '@kit.NetworkKit';
import { BusinessError } from '@kit.BasicServicesKit';

// 每一个httpRequest对应一个HTTP请求任务，不可复用
let httpRequest = http.createHttp();
// 用于订阅HTTP响应头事件
httpRequest.on('headersReceive', (header: Object) => {
  console.info('header: ' + JSON.stringify(header));
});
// 用于订阅HTTP流式响应数据接收事件
let res = new ArrayBuffer(0);
httpRequest.on('dataReceive', (data: ArrayBuffer) => {
  const newRes = new ArrayBuffer(res.byteLength + data.byteLength);
  const resView = new Uint8Array(newRes);
  resView.set(new Uint8Array(res));
  resView.set(new Uint8Array(data), res.byteLength);
  res = newRes;
  console.info('res length: ' + res.byteLength);
});
// 用于订阅HTTP流式响应数据接收完毕事件
httpRequest.on('dataEnd', () => {
  console.info('No more data in response, data receive end');
});
// 用于订阅HTTP流式响应数据接收进度事件
class Data {
  receiveSize: number = 0;
  totalSize: number = 0;
}
httpRequest.on('dataReceiveProgress', (data: Data) => {
  console.log("dataReceiveProgress receiveSize:" + data.receiveSize + ", totalSize:" + data.totalSize);
});

let streamInfo: http.HttpRequestOptions = {
  method: http.RequestMethod.POST,  // 可选，默认为http.RequestMethod.GET
  // 开发者根据自身业务需要添加header字段
  header: {
    'Content-Type': 'application/json'
  },
  // 当使用POST请求时此字段用于传递请求体内容，具体格式与服务端协商确定
  extraData: "data to send",
  expectDataType:  http.HttpDataType.STRING,// 可选，指定返回数据的类型
  usingCache: true, // 可选，默认为true
  priority: 1, // 可选，默认为1
  connectTimeout: 60000, // 可选，默认为60000ms
  readTimeout: 60000, // 可选，默认为60000ms。若传输的数据较大，需要较长的时间，建议增大该参数以保证数据传输正常终止
  usingProtocol: http.HttpProtocol.HTTP1_1 // 可选，协议类型默认值由系统自动指定
}

// 填写HTTP请求的URL地址，可以带参数也可以不带参数。URL地址需要开发者自定义。请求的参数可以在extraData中指定
httpRequest.requestInStream("EXAMPLE_URL", streamInfo).then((data: number) => {
  console.info("requestInStream OK!");
  console.info('ResponseCode :' + JSON.stringify(data));
  // 取消订阅HTTP响应头事件
  httpRequest.off('headersReceive');
  // 取消订阅HTTP流式响应数据接收事件
  httpRequest.off('dataReceive');
  // 取消订阅HTTP流式响应数据接收进度事件
  httpRequest.off('dataReceiveProgress');
  // 取消订阅HTTP流式响应数据接收完毕事件
  httpRequest.off('dataEnd');
  // 当该请求使用完毕时，调用destroy方法主动销毁
  httpRequest.destroy();
}).catch((err: Error) => {
  console.info("requestInStream ERROR : err = " + JSON.stringify(err));
});
```

WebSocket连接

使用WebSocket建立服务器与客户端的双向连接，需要先通过createWebSocket()方法创建WebSocket对象，然后通过connect()方法连接到服务器。当连接成功后，客户端会收到open事件的回调，之后客户端就可以通过send()方法与服务器进行通信。当服务器发信息给客户端时，客户端会收到message事件的回调。当客户端不要此连接时，可以通过调用close()方法主动断开连接，之后客户端会收到close事件的回调。

接口说明

WebSocket连接功能主要由webSocket模块提供。使用该功能需要申请ohos.permission.INTERNET权限。具体接口说明如下表。

接口名	描述
createWebSocket()	创建一个WebSocket连接。
connect()	根据URL地址，建立一个WebSocket连接。
send()	通过WebSocket连接发送数据。
close()	关闭WebSocket连接。
on(type: 'open')	订阅WebSocket的打开事件。
off(type: 'open')	取消订阅WebSocket的打开事件。
on(type: 'message')	订阅WebSocket的接收到服务器消息事件。
off(type: 'message')	取消订阅WebSocket的接收到服务器消息事件。
on(type: 'close')	订阅WebSocket的关闭事件。
off(type: 'close')	取消订阅WebSocket的关闭事件
on(type: 'error')	订阅WebSocket的Error事件。
off(type: 'error')	取消订阅WebSocket的Error事件。

开发步骤

开发步骤

- 导入需要的webSocket模块。
- 创建一个WebSocket连接，返回一个WebSocket对象。
- （可选）订阅WebSocket的打开、消息接收、关闭、Error事件。
- 根据URL地址，发起WebSocket连接。
- 使用完WebSocket连接之后，主动断开连接。

```
import { webSocket } from '@kit.NetworkKit';
import { BusinessError } from '@kit.BasicServicesKit';

let defaultIpAddress = "ws://";
let ws = webSocket.createWebSocket();
ws.on('open', (err: BusinessError, value: Object) => {
  console.log("on open, status:" + JSON.stringify(value));
  // 当收到on('open')事件时，可以通过send()方法与服务器进行通信
  ws.send("Hello, server!", (err: BusinessError, value: boolean) => {
    if (!err) {
      console.log("Message sent successfully");
    } else {
      console.log("Failed to send the message. Err:" + JSON.stringify(err));
    }
  });
});
ws.on('message', (err: BusinessError, value: string | ArrayBuffer) => {
  console.log("on message, message:" + value);
  // 当收到服务器的`bye`消息时（此消息字段仅为示意，具体字段需要与服务器协商），主动断开连接
  if (value === 'bye') {
    ws.close((err: BusinessError, value: boolean) => {
      if (!err) {
        console.log("Connection closed successfully");
      } else {
        console.log("Failed to close the connection. Err: " + JSON.stringify(err));
      }
    });
  }
});
ws.on('close', (err: BusinessError, value: webSocket.CloseResult) => {
  console.log("on close, code is " + value.code + ", reason is " + value.reason);
});
ws.on('error', (err: BusinessError) => {
  console.log("on error, error:" + JSON.stringify(err));
});
ws.connect(defaultIpAddress, (err: BusinessError, value: boolean) => {
  if (!err) {
    console.log("Connected successfully");
  } else {
    console.log("Connection failed. Err:" + JSON.stringify(err));
  }
});
```

Socket连接

简介

Socket 连接主要是通过 Socket 进行数据传输，支持 TCP/UDP/Multicast/TLS 协议。

基本概念

- Socket：套接字，就是对网络中不同主机上的应用进程之间进行双向通信的端点的抽象。
- TCP：传输控制协议(Transmission Control Protocol)。是一种面向连接的、可靠的、基于字节流的传输层通信协议。
- UDP：用户数据报协议(User Datagram Protocol)。是一个简单的面向消息的传输层，不需要连接。
- Multicast：多播，基于UDP的一种通信模式，用于实现组内所有设备之间广播形式的通信。
- LocalSocket：本地套接字，IPC(Inter-Process Communication)进程间通信的一种，实现设备内进程之间相互通信，无需网络。
- TLS：安全传输层协议(Transport Layer Security)。用于在两个通信应用程序之间提供保密性和数据完整性。

场景介绍

应用通过 Socket 进行数据传输，支持 TCP/UDP/Multicast/TLS 协议。主要场景有：

- 应用通过 TCP/UDP Socket进行数据传输
- 应用通过 TCP Socket Server 进行数据传输
- 应用通过 Multicast Socket 进行数据传输
- 应用通过 Local Socket进行数据传输
- 应用通过 Local Socket Server 进行数据传输
- 应用通过 TLS Socket 进行加密数据传输
- 应用通过 TLS Socket Server 进行加密数据传输

接口名	描述
constructUDPSocketInstance()	创建一个 UDPSocket 对象。
constructTCPSocketInstance()	创建一个 TCPSocket 对象。
constructTCPSocketServerInstance()	创建一个 TCPSocketServer 对象。
constructMulticastSocketInstance()	创建一个 MulticastSocket 对象。
constructLocalSocketInstance()	创建一个 LocalSocket 对象。
constructLocalSocketServerInstance()	创建一个 LocalSocketServer 对象。
listen()	绑定、监听并启动服务，接收客户端的连接请求。（仅 TCP/LocalSocket 支持）。
bind()	绑定 IP 地址和端口，或是绑定本地套接字路径。
send()	发送数据。
close()	关闭连接。
getState()	获取 Socket 状态。
connect()	连接到指定的 IP 地址和端口，或是连接到本地套接字（仅 TCP/LocalSocket 支持）。
getRemoteAddress()	获取对端 Socket 地址（仅 TCP 支持，需要先调用 connect 方法）。
setExtraOptions()	设置 Socket 连接的其他属性。
getExtraOptions()	获取 Socket 连接的其他属性（仅 LocalSocket 支持）。
addMembership()	加入到指定的多播组 IP 中 (仅 Multicast 支持)。

接口说明

应用TCP/UDP协议进行通信

- import 需要的 socket 模块。
- 创建一个 TCPSocket 连接，返回一个 TCPSocket 对象。
- （可选）订阅 TCPSocket 相关的订阅事件。
- 绑定 IP 地址和端口，端口可以指定或由系统随机分配。
- 连接到指定的 IP 地址和端口。
- 发送数据。
- Socket 连接使用完毕后，主动关闭。

```
import { socket } from '@kit.NetworkKit';
import { BusinessError } from '@kit.BasicServicesKit';

class SocketInfo {
  message: ArrayBuffer = new ArrayBuffer(1);
  remoteInfo: socket.SocketRemoteInfo = {} as socket.SocketRemoteInfo;
}
// 创建一个TCPSocket连接，返回一个TCPSocket对象。
let tcp: socket.TCPSocket = socket.constructTCPSocketInstance();
tcp.on('message', (value: SocketInfo) => {
  console.log("on message");
  let buffer = value.message;
  let dataView = new DataView(buffer);
  let str = "";
  for (let i = 0; i < dataView.byteLength; ++i) {
    str += String.fromCharCode(dataView.getUint8(i));
  }
  console.log("on connect received:" + str);
});
tcp.on('connect', () => {
  console.log("on connect");
});
tcp.on('close', () => {
  console.log("on close");
});

// 绑定本地IP地址和端口。
let ipAddress : socket.NetAddress = {} as socket.NetAddress;
ipAddress.address = "192.168.xxx.xxx";
ipAddress.port = 1234;
tcp.bind(ipAddress, (err: BusinessError) => {
  if (err) {
    console.log('bind fail');
    return;
  }
  console.log('bind success');

  // 连接到指定的IP地址和端口。
  ipAddress.address = "192.168.xxx.xxx";
  ipAddress.port = 5678;

  let tcpConnect : socket.TCPConnectOptions = {} as socket.TCPConnectOptions;
  tcpConnect.address = ipAddress;
  tcpConnect.timeout = 6000;

  tcp.connect(tcpConnect).then(() => {
    console.log('connect success');
    let tcpSendOptions: socket.TCPSendOptions = {
      data: 'Hello, server!'
    }
    tcp.send(tcpSendOptions).then(() => {
      console.log('send success');
    }).catch((err: BusinessError) => {
      console.log('send fail');
    });
  }).catch((err: BusinessError) => {
    console.log('connect fail');
  });
});

// 连接使用完毕后，主动关闭。取消相关事件的订阅。
setTimeout(() => {
  tcp.close().then(() => {
    console.log('close success');
  }).catch((err: BusinessError) => {
    console.log('close fail');
  });
  tcp.off('message');
  tcp.off('connect');
  tcp.off('close');
}, 30 * 1000);
```

应用通过 TCP Socket Server 进行数据传输

应用通过 TCP Socket Server 进行数据传输

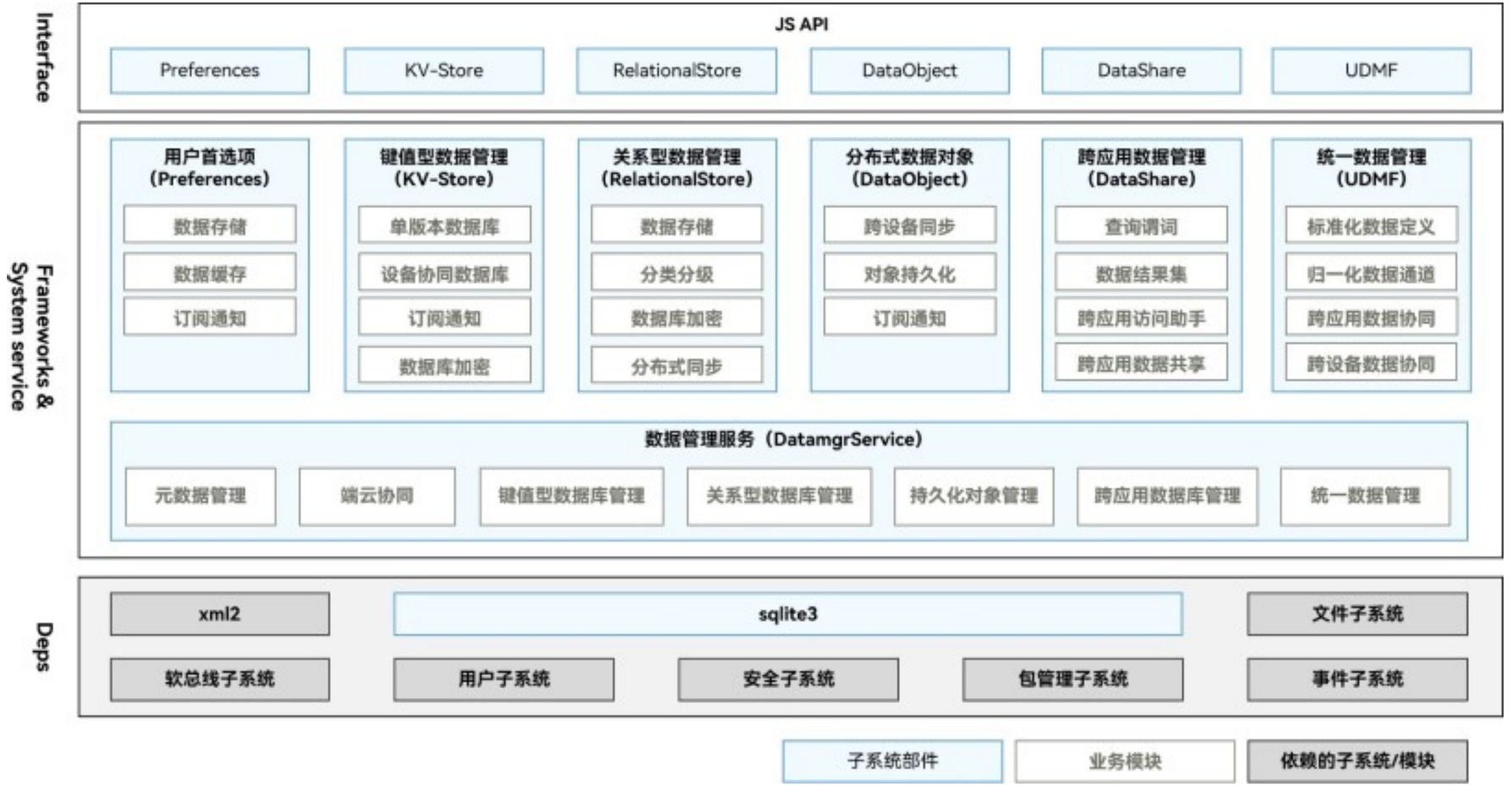
服务端 TCP Socket 流程：

- import 需要的 socket 模块。
- 创建一个 TCPSocketServer 连接，返回一个 TCPSocketServer 对象。
- 绑定本地 IP 地址和端口，监听并接受与此套接字建立的客户端 TCPSocket 连接。
- 订阅 TCPSocketServer 的 connect 事件，用于监听客户端的连接状态。
- 客户端与服务端建立连接后，返回一个 TCPSocketConnection 对象，用于与客户端通信。
- 订阅 TCPSocketConnection 相关的事件，通过 TCPSocketConnection 向客户端发送数据。
- 主动关闭与客户端的连接。
- 取消 TCPSocketConnection 和 TCPSocketServer 相关事件的订阅。

```
import { socket } from '@kit.NetworkKit';
import { BusinessError } from '@kit.BasicServicesKit';

// 创建一个TCPSocketServer连接，返回一个TCPSocketServer对象。
let tcpServer: socket.TCPSocketServer = socket.constructTCPSocketServerInstance();
// 绑定本地IP地址和端口，进行监听

let ipAddress : socket.NetAddress = {} as socket.NetAddress;
ipAddress.address = "192.168.xxx.xxx";
ipAddress.port = 4651;
tcpServer.listen(ipAddress).then(() => {
  console.log('listen success');
}).catch((err: BusinessError) => {
  console.log('listen fail');
});

class SocketInfo {
  message: ArrayBuffer = new ArrayBuffer(1);
  remoteInfo: socket.SocketRemoteInfo = {} as socket.SocketRemoteInfo;
}
// 订阅TCPSocketServer的connect事件
tcpServer.on("connect", (client: socket.TCPSocketConnection) => {
  // 订阅TCPSocketConnection相关的事件
  client.on("close", () => {
    console.log("on close success");
  });
  client.on("message", (value: SocketInfo) => {
    let buffer = value.message;
    let dataView = new DataView(buffer);
    let str = "";
    for (let i = 0; i < dataView.byteLength; ++i) {
      str += String.fromCharCode(dataView.getUint8(i));
    }
    console.log("received message--:" + str);
    console.log("received address--:" + value.remoteInfo.address);
    console.log("received family--:" + value.remoteInfo.family);
    console.log("received port--:" + value.remoteInfo.port);
    console.log("received size--:" + value.remoteInfo.size);
  });

  // 向客户端发送数据
  let tcpSendOptions : socket.TCPSendOptions = {} as socket.TCPSendOptions;
  tcpSendOptions.data = 'Hello, client!';
  client.send(tcpSendOptions).then(() => {
    console.log('send success');
  }).catch((err: Object) => {
    console.error('send fail: ' + JSON.stringify(err));
  });

  // 关闭与客户端的连接
  client.close().then(() => {
    console.log('close success');
  }).catch((err: BusinessError) => {
    console.log('close fail');
  });

  // 取消TCPSocketConnection相关的事件订阅
  setTimeout(() => {
    client.off("message");
    client.off("close");
  }, 10 * 1000);
});

// 取消TCPSocketServer相关的事件订阅
setTimeout(() => {
  tcpServer.off("connect");
}, 30 * 1000);
```

实践案例：　HttpRequest

ArkData


Interface
JS API
Preferences
KV-Store
RelationalStore
DataObject
DataShare
UDMF
Frameworks & System service
用户首选项 (Preferences)
数据存储
数据缓存
订阅通知
键值型数据管理 (KV-Store)
单版本数据库
设备协同数据库
订阅通知
数据库加密
关系型数据管理 (RelationalStore)
数据存储
分类分级
数据库加密
分布式同步
分布式数据对象 (DataObject)
跨设备同步
对象持久化
订阅通知
跨应用数据管理 (DataShare)
查询谓词
数据结果集
跨应用访问助手
跨应用数据共享
统一数据管理 (UDMF)
标准化数据定义
归一化数据通道
跨应用数据协同
跨设备数据协同
数据管理服务（DatamgrService）
元数据管理
端云协同
键值型数据库管理
关系型数据库管理
持久化对象管理
跨应用数据库管理
统一数据管理
Deps
xml2
sqlite3
文件子系统
软总线子系统
用户子系统
安全子系统
包管理子系统
事件子系统
子系统部件
业务模块
依赖的子系统/模块

数据管理

运作机制

- 持久化
  - 用户首选项（Preferences）：提供了轻量级配置数据的持久化能力，并支持订阅数据变化的通知能力。不支持分布式同步，常用于保存应用配置信息、用户偏好设置等。
  - 键值型数据管理（KV-Store）：提供了键值型数据库的读写、加密、手动备份以及订阅通知能力。应用需要使用键值型数据库的分布式能力时，KV-Store会将同步请求发送给DatamgrService由其完成跨设备数据同步。
  - 关系型数据管理（RelationalStore）：提供了关系型数据库的增删改查、加密、手动备份以及订阅通知能力。应用需要使用关系型数据库的分布式能力时，RelationalStore部件会将同步请求发送给DatamgrService由其完成跨设备数据同步。
- 分布式交互
  - 分布式数据对象（DataObject）：独立提供对象型结构数据的分布式能力。如果应用需要重启后仍获取之前的对象数据（包含跨设备应用和本设备应用），则使用数据管理服务（DatamgrService）的对象持久化能力，做暂时保存。
  - 跨应用数据管理（DataShare）：提供了数据提供者provider、数据消费者consumer以及同设备跨应用数据交互的增、删、改、查以及订阅通知等能力。DataShare不与任何数据库绑定，可以对接关系型数据库、键值型数据库。如果开发C/C++应用甚至可以自行封装数据库。在提供标准的provider-consumer模式基础上，同时提供了静默数据访问能力，即不再拉起provider而是直接通过DatamgrService代理访问provider的数据（目前仅关系型数据库支持静默数据访问方式）。
  - 统一数据管理框架（UDMF）：提供了数据跨应用、跨设备交互标准，定义了跨应用、跨设备数据交互过程中的数据语言，提升数据交互效率。提供安全、标准化数据流通通路，支持不同级别的数据访问权限与生命周期管理策略，实现高效的数据跨应用、跨设备共享。
- 数据管理服务（DatamgrService）：提供其它部件的同步及跨应用共享能力，包括RelationalStore和KV-Store跨设备同步，DataShare静默访问provider数据，暂存DataObject同步对象数据等。

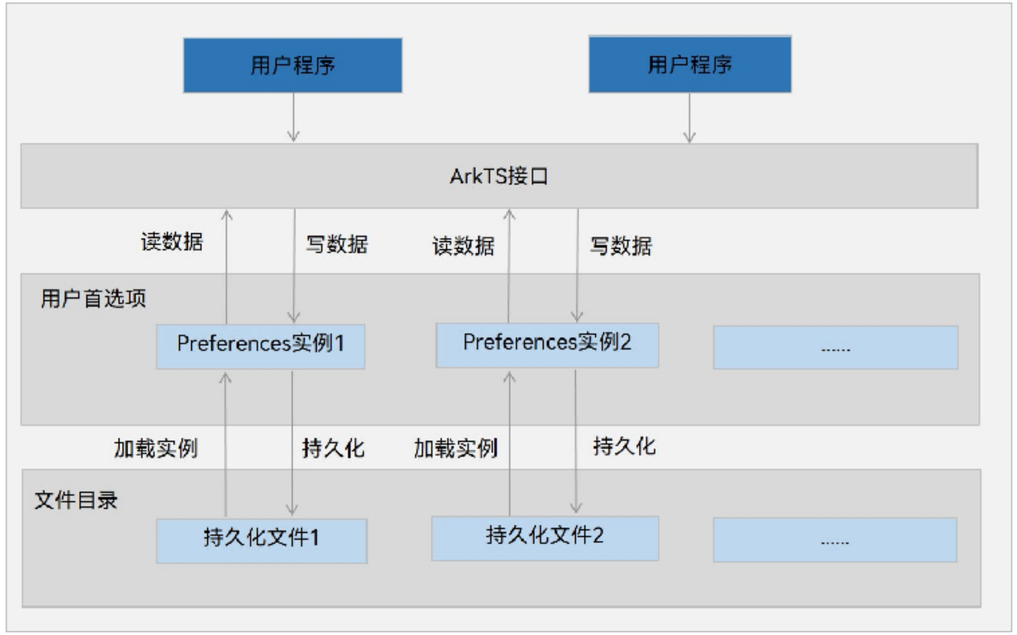
用户首选项

使用场景：

一般用于记住用户的一些选择，例如终端设置（如位置开关、NFC开关），用户的使用习惯（如“仅此一次”、“始终”）。

原理：

1. 用户程序通过JS接口调用用户首选项读写对应的数据文件。开发者可以将用户首选项持久化文件的内容加载到Preferences实例，每一个文件唯一对应到一个Preferences实例，系统会通过静态容器将该实例存储在内存中，直到主动从内存中移除该实例或者删除该文件。 2. 应用首选项的持久化文件保存在应用沙箱内部，可以通过context 获取其路径。



接口名称	描述
getPreferencesSync(context: Context, options: Options): Preferences	获取Preferences实例。该接口存在异步接口。
putSync(key: string, value: ValueType): void	将数据写入Preferences实例，可通过flush将Preferences实例持久化。该接口存在异步接口。
hasSync(key: string): boolean	检查Preferences实例是否包含名为给定Key的存储键值对。给定的Key值不能为空。该接口存在异步接口。
getSync(key: string, defValue: ValueType): ValueType	获取键对应的值，如果值为null或者非默认值类型，返回默认数据defValue。该接口存在异步接口。
deleteSync(key: string): void	从Preferences实例中删除名为给定Key的存储键值对。该接口存在异步接口。
flush(callback: AsyncCallback<void>): void	将当前Preferences实例的数据异步存储到用户首选项持久化文件中。
on(type: 'change', callback: Callback<string>): void	订阅数据变更，订阅的数据发生变更后，在执行flush方法后，触发callback回调。
off(type: 'change', callback?: Callback<string>): void	取消订阅数据变更。
deletePreferences(context: Context, options: Options, callback: AsyncCallback<void>): void	从内存中移除指定的Preferences实例。若Preferences实例有对应的持久化文件，则同时删除其持久化文件。

接口说明

开发步骤

- 导入@kit.ArkData模块。

```
import { preferences } from '@kit.ArkData';
```

- 获取Preferences实例。

```
import { UIAbility } from '@kit.AbilityKit';
import { BusinessError } from '@kit.BasicServicesKit';
import { window } from '@kit.ArkUI';

let dataPreferences: preferences.Preferences | null = null;

class EntryAbility extends UIAbility {
  onWindowStageCreate(windowStage: window.WindowStage) {
    let options: preferences.Options = { name: 'myStore' };
    dataPreferences = preferences.getPreferencesSync(this.context, options);
  }
}
```

- 写入数据。
  使用putSync()方法保存数据到缓存的Preferences实例中。在写入数据后，如有需要，可使用flush()方法将Preferences实例的数据存储到持久化文件。
  说明
  当对应的键已经存在时，putSync()方法会覆盖其值。可以使用hasSync()方法检查是否存在对应键值对。

  示例代码如下所示：

```
import { util } from '@kit.ArkTS';
if (dataPreferences.hasSync('startup')) {
  console.info("The key 'startup' is contained.");
} else {
  console.info("The key 'startup' does not contain.");
  // 此处以此键值对不存在时写入数据为例
  dataPreferences.putSync('startup', 'auto');
  // 当字符串有特殊字符时，需要将字符串转为Uint8Array类型再存储
  let uInt8Array1 = new util.TextEncoder().encodeInto("~！ @#￥%……&* （） ——+？ ");
  dataPreferences.putSync('uInt8', uInt8Array1);
}
```

- 读取数据。
  使用getSync()方法获取数据，即指定键对应的值。如果值为null或者非默认值类型，则返回默认数据。
  示例代码如下所示：

```
let val = dataPreferences.getSync('startup', 'default');
console.info("The 'startup' value is " + val);
// 当获取的值为带有特殊字符的字符串时，需要将获取到的Uint8Array转换为字符串
let uInt8Array2 : preferences.ValueType = dataPreferences.getSync('uInt8', new Uint8Array(0));
let textDecoder = util.TextDecoder.create('utf-8');
val = textDecoder.decodeToString(uInt8Array2 as Uint8Array);
console.info("The 'uInt8' value is " + val);
```

开发步骤

- 删除数据。
  使用deleteSync()方法删除指定键值对，示例代码如下所示：

```
dataPreferences.deleteSync('startup');
```

- 数据持久化。
  应用存入数据到Preferences实例后，可以使用flush()方法实现数据持久化。示例代码如下所示：

```
dataPreferences.flush((err: BusinessError) => {
  if (err) {
    console.error(`Failed to flush. Code:${err.code}, message:${err.message}`);
    return;
  }
  console.info('Succeeded in flushing.');
})
```

- 订阅数据变更。
  应用订阅数据变更需要指定observer作为回调方法。订阅的Key值发生变更后，当执行flush()方法时，observer被触发回调。示例代码如下所示：

```
let observer = (key: string) => {
  console.info('The key' + key + 'changed.');
}
dataPreferences.on('change', observer);
// 数据产生变更，由'auto'变为'manual'
dataPreferences.put('startup', 'manual', (err: BusinessError) => {
  if (err) {
    console.error(`Failed to put the value of 'startup'. Code:${err.code},message:${err.message}`);
    return;
  }
  console.info("Succeeded in putting the value of 'startup'.");
  if (dataPreferences !== null) {
    dataPreferences.flush((err: BusinessError) => {
      if (err) {
        console.error(`Failed to flush. Code:${err.code}, message:${err.message}`);
        return;
      }
      console.info('Succeeded in flushing.');
    })
  }
})
```

- 删除指定文件。
  使用deletePreferences()方法从内存中移除指定文件对应的Preferences实例，包括内存中的数据。若该Preference存在对应的持久化文件，则同时删除该持久化文件，包括指定文件及其备份文件、损坏文件。
  说明
  - 调用该接口后，应用不允许再使用该Preferences实例进行数据操作，否则会出现数据一致性问题。
  - 成功删除后，数据及文件将不可恢复。

  示例代码如下所示：

```
preferences.deletePreferences(this.context, options, (err: BusinessError) => {
  if (err) {
    console.error(`Failed to delete preferences. Code:${err.code}, message:${err.message}`);
    return;
  }
  console.info('Succeeded in deleting preferences.');
})
```

案例： Preference

键值型数据库

场景介绍

键值型数据库存储键值对形式的数据，当需要存储的数据没有复杂的关系模型，比如存储商品名称及对应价格、员工工号及今日是否已出勤等，由于数据复杂度低，更容易兼容不同数据库版本和设备类型，因此推荐使用键值型数据库持久化此类数据。

约束限制

- 设备协同数据库，针对每条记录， Key的长度=896 Byte，Value的长度<4 MB。
- 单版本数据库，针对每条记录， Key的长度=1 KB，Value的长度<4 MB。
- 每个应用程序最多支持同时打开16个键值型分布式数据库。
- 键值型数据库事件回调方法中不允许进行阻塞操作，例如修改UI组件。

以下是键值型数据库持久化功能的相关接口，大部分为异步接口。异步接口均有callback和Promise两种返回形式，下表均以callback形式为例，更多接口及使用方式请见分布式键值数据库。

接口名称	描述
createKVManager(config: KVManagerConfig): KVManager	创建一个KVManager对象实例，用于管理数据库对象。
getKVStore<T>(storeId: string, options: Options, callback: AsyncCallback<T>): void	指定options和storeId，创建并得到指定类型的KVStore数据库。
put(key: string, value: Uint8Array | string | number | boolean, callback: AsyncCallback<void>): void	添加指定类型的键值对到数据库。
get(key: string, callback: AsyncCallback<boolean | string | number | Uint8Array>): void	获取指定键的值。
delete(key: string, callback: AsyncCallback<void>): void	从数据库中删除指定键值的数据。

接口说明

开发步骤 - 1

若要使用键值型数据库，首先要获取一个KVManager实例，用于管理数据库对象。示例代码如下所示：
Stage模型示例：

```
// 导入模块
import { distributedKVStore } from '@kit.ArkData';

// Stage模型
import { window } from '@kit.ArkUI';
import { UIAbility } from '@kit.AbilityKit';
import { BusinessError } from '@kit.BasicServicesKit';

let kvManager: distributedKVStore.KVManager | undefined = undefined;

export default class EntryAbility extends UIAbility {
  onCreate() {
    let context = this.context;
    const kvManagerConfig: distributedKVStore.KVManagerConfig = {
      context: context,
      bundleName: 'com.example.datamanagertest'
    };
    try {
      // 创建KVManager实例
      kvManager = distributedKVStore.createKVManager(kvManagerConfig);
      console.info('Succeeded in creating KVManager.');
      // 继续创建获取数据库
    } catch (e) {
      let error = e as BusinessError;
      console.error(`Failed to create KVManager. Code:${error.code},message:${error.message}`);
    }
  }
}
if (kvManager !== undefined) {
  kvManager = kvManager as distributedKVStore.KVManager;
  //进行后续操作
  //...
}
```

开发步骤 - 2

- 创建并获取键值数据库。示例代码如下所示：

```
let kvStore: distributedKVStore.SingleKVStore | undefined = undefined;
try {
  const options: distributedKVStore.Options = {
    createIfMissing: true,
    encrypt: false,
    backup: false,
    autoSync: false,
    // kvStoreType不填时，默认创建多设备协同数据库
    kvStoreType: distributedKVStore.KVStoreType.SINGLE_VERSION,
    // 多设备协同数据库：kvStoreType: distributedKVStore.KVStoreType.DEVICE_COLLABORATION,
    securityLevel: distributedKVStore.SecurityLevel.S1
  };
  kvManager.getKVStore<distributedKVStore.SingleKVStore>('storeId', options, (err, store: distributedKVStore.SingleKVStore) => {
    if (err) {
      console.error(`Failed to get KVStore: Code:${err.code},message:${err.message}`);
      return;
    }
    console.info('Succeeded in getting KVStore.');
    kvStore = store;
    // 请确保获取到键值数据库实例后，再进行相关数据操作
  });
} catch (e) {
  let error = e as BusinessError;
  console.error(`An unexpected error occurred. Code:${error.code},message:${error.message}`);
}
if (kvStore !== undefined) {
  kvStore = kvStore as distributedKVStore.SingleKVStore;
  //进行后续操作
  //...
}
```

- 调用put()方法向键值数据库中插入数据。示例代码如下所示：

```
const KEY_TEST_STRING_ELEMENT = 'key_test_string';
const VALUE_TEST_STRING_ELEMENT = 'value_test_string';
try {
  kvStore.put(KEY_TEST_STRING_ELEMENT, VALUE_TEST_STRING_ELEMENT, (err) => {
    if (err !== undefined) {
      console.error(`Failed to put data. Code:${err.code},message:${err.message}`);
      return;
    }
    console.info('Succeeded in putting data.');
  });
} catch (e) {
  let error = e as BusinessError;
  console.error(`An unexpected error occurred. Code:${error.code},message:${error.message}`);
}
```

开发步骤 - 3

- 调用get()方法获取指定键的值。示例代码如下所示：

```
try {
  kvStore.put(KEY_TEST_STRING_ELEMENT, VALUE_TEST_STRING_ELEMENT, (err) => {
    if (err !== undefined) {
      console.error(`Failed to put data. Code:${err.code},message:${err.message}`);
      return;
    }
    console.info('Succeeded in putting data.');
    kvStore = kvStore as distributedKVStore.SingleKVStore;
    kvStore.get(KEY_TEST_STRING_ELEMENT, (err, data) => {
      if (err != undefined) {
        console.error(`Failed to get data. Code:${err.code},message:${err.message}`);
        return;
      }
      console.info(`Succeeded in getting data. Data:${data}`);
    });
  });
} catch (e) {
  let error = e as BusinessError;
  console.error(`Failed to get data. Code:${error.code},message:${error.message}`);
}
```

- 调用delete()方法删除指定键值的数据。示例代码如下所示：

```
try {
  kvStore.put(KEY_TEST_STRING_ELEMENT, VALUE_TEST_STRING_ELEMENT, (err) => {
    if (err !== undefined) {
      console.error(`Failed to put data. Code:${err.code},message:${err.message}`);
      return;
    }
    console.info('Succeeded in putting data.');
    kvStore = kvStore as distributedKVStore.SingleKVStore;
    kvStore.delete(KEY_TEST_STRING_ELEMENT, (err) => {
      if (err !== undefined) {
        console.error(`Failed to delete data. Code:${err.code},message:${err.message}`);
        return;
      }
      console.info('Succeeded in deleting data.');
    });
  });
} catch (e) {
  let error = e as BusinessError;
  console.error(`An unexpected error occurred. Code:${error.code},message:${error.message}`);
}
```

关系型数据库

场景介绍

关系型数据库基于SQLite组件，适用于存储包含复杂关系数据的场景，比如一 个班级的学生信息，需要包括姓名、学号、各科成绩等，又或者公司的雇员信 息，需要包括姓名、工号、职位等，由于数据之间有较强的对应关系，复杂程 度比键值型数据更高，此时需要使用关系型数据库来持久化保存数据。

基本概念

. **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，可以更方便地拿到用户想要的数据。

运作机制

关系型数据库对应用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。

约束限制

- 系统默认日志方式是WAL（Write Ahead Log）模式，系统默认落盘方式是 FULL模式。
- 数据库中连接池的最大数量是4个，用以管理用户的读操作。
- 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。
- 当应用被卸载完成后，设备上的相关数据库文件及临时文件会被自动清除。

以下是关系型数据库持久化功能的相关接口，大部分为异步接口。异步接口均有callback和Promise两种返回形式，下表均以callback形式为例，更多接口及使用方式请见关系型数据库。

接口名称	描述
getRdbStore(context: Context, config: StoreConfig, callback: AsyncCallback<RdbStore>): void	获得一个RdbStore，操作关系型数据库，用户可以根据自己的需求配置RdbStore的参数，然后通过RdbStore调用相关接口可以执行相关的数据操作。
executeSql(sql: string, bindArgs: Array<ValueType>, callback: AsyncCallback<void>):void	执行包含指定参数但不返回值的SQL语句。
insert(table: string, values: ValuesBucket, callback: AsyncCallback<number>):void	向目标表中插入一行数据。
update(values: ValuesBucket, predicates: RdbPredicates, callback: AsyncCallback<number>):void	根据predicates的指定实例对象更新数据库中的数据。
delete(predicates: RdbPredicates, callback: AsyncCallback<number>):void	根据predicates的指定实例对象从数据库中删除数据。
query(predicates: RdbPredicates, columns: Array<string>, callback: AsyncCallback<ResultSet>):void	根据指定条件查询数据库中的数据。
deleteRdbStore(context: Context, name: string, callback: AsyncCallback<void>): void	删除数据库。

接口说明

开发步骤

使用关系型数据库实现数据持久化，需要获取一个RdbStore，其中包括建库、建表、升降级等操作。示例代码如下所示：
Stage模型示例：

```
import { relationalStore } from '@kit.ArkData'; // 导入模块
import { UIAbility } from '@kit.AbilityKit';
import { BusinessError } from '@kit.BasicServicesKit';
import { window } from '@kit.ArkUI';

// 此处示例在Ability中实现，使用者也可以在其他合理场景中使用
class EntryAbility extends UIAbility {
  onWindowStageCreate(windowStage: window.WindowStage) {
    const STORE_CONFIG :relationalStore.StoreConfig= {
      name: 'RdbTest.db', // 数据库文件名
      securityLevel: relationalStore.SecurityLevel.S1, // 数据库安全级别
      encrypt: false, // 可选参数，指定数据库是否加密，默认不加密
     customDir: 'customDir/subCustomDir', // 可选参数，数据库自定义路径。数据库将在如下的目录结构中被创建：context.databaseDir + '/rdb/' + customDir，其中context.databaseDir是应用沙箱对应的路径，'/rdb/'表示创建的是关系型数据库，customDir表示自定义的路径。当此参数不填时，默认在本应用沙箱目录下创建RdbStore实例。
      isReadOnly: false // 可选参数，指定数据库是否以只读方式打开。该参数默认为false，表示数据库可读可写。该参数为true时，只允许从数据库读取数据，不允许对数据库进行写操作，否则会返回错误码801。
    };

    // 判断数据库版本，如果不匹配则需进行升降级操作
    // 假设当前数据库版本为3，表结构：EMPLOYEE (NAME, AGE, SALARY, CODES, IDENTITY)
    const SQL_CREATE_TABLE = 'CREATE TABLE IF NOT EXISTS EMPLOYEE (ID INTEGER PRIMARY KEY AUTOINCREMENT, NAME TEXT NOT NULL, AGE INTEGER, SALARY REAL, CODES BLOB, IDENTITY UNLIMITED INT)'; // 建表Sql语句, IDENTITY为bigint类型，sql中指定类型为UNLIMITED INT

    relationalStore.getRdbStore(this.context, STORE_CONFIG, (err, store) => {
      if (err) {
        console.error(`Failed to get RdbStore. Code:${err.code}, message:${err.message}`);
        return;
      }
      console.info('Succeeded in getting RdbStore.');

      // 当数据库创建时，数据库默认版本为0
      if (store.version === 0) {
        store.executeSql(SQL_CREATE_TABLE); // 创建数据表
        // 设置数据库的版本，入参为大于0的整数
        store.version = 3;
      }

      // 如果数据库版本不为0且和当前数据库版本不匹配，需要进行升降级操作
      // 当数据库存在并假定版本为1时，例应用从某一版本升级到当前版本，数据库需要从1版本升级到2版本
      if (store.version === 1) {
        // version = 1：表结构：EMPLOYEE (NAME, SALARY, CODES, ADDRESS) => version = 2：表结构：EMPLOYEE (NAME, AGE, SALARY, CODES, ADDRESS)
        if (store !== undefined) {
          (store as relationalStore.RdbStore).executeSql('ALTER TABLE EMPLOYEE ADD COLUMN AGE INTEGER');
          store.version = 2;
        }
      }

      // 当数据库存在并假定版本为2时，例应用从某一版本升级到当前版本，数据库需要从2版本升级到3版本
      if (store.version === 2) {
        // version = 2：表结构：EMPLOYEE (NAME, AGE, SALARY, CODES, ADDRESS) => version = 3：表结构：EMPLOYEE (NAME, AGE, SALARY, CODES)
        if (store !== undefined) {
          (store as relationalStore.RdbStore).executeSql('ALTER TABLE EMPLOYEE DROP COLUMN ADDRESS TEXT');
          store.version = 3;
        }
      }
    });

    // 请确保获取到RdbStore实例后，再进行数据库的增、删、改、查等操作
  }
}

```

开发步骤

获取到RdbStore后，调用insert()接口插入数据。示例代码如下所示：

```
let store: relationalStore.RdbStore | undefined = undefined;

let value1 = 'Lisa';
let value2 = 18;
let value3 = 100.5;
let value4 = new Uint8Array([1, 2, 3, 4, 5]);
let value5 = BigInt('15822401018187971961171');
// 以下三种方式可用
const valueBucket1: relationalStore.ValuesBucket = {
  'NAME': value1,
  'AGE': value2,
  'SALARY': value3,
  'CODES': value4,
  'IDENTITY': value5,
};
const valueBucket2: relationalStore.ValuesBucket = {
  NAME: value1,
  AGE: value2,
  SALARY: value3,
  CODES: value4,
  IDENTITY: value5,
};
const valueBucket3: relationalStore.ValuesBucket = {
  "NAME": value1,
  "AGE": value2,
  "SALARY": value3,
  "CODES": value4,
  "IDENTITY": value5,
};

if (store !== undefined) {
  (store as relationalStore.RdbStore).insert('EMPLOYEE', valueBucket1,
(err: BusinessError, rowId: number) => {
    if (err) {
      console.error(`Failed to insert data. Code:${err.code}, message:$
{err.message}`);
      return;
    }
    console.info(`Succeeded in inserting data. rowId:${rowId}`);
  })
}

```

开发步骤

- 根据谓词指定的实例对象，对数据进行修改或删除。
  调用update()方法修改数据，调用delete()方法删除数据。示例代码如下所示：

```
let value6 = 'Rose';
let value7 = 22;
let value8 = 200.5;
let value9 = new Uint8Array([1, 2, 3, 4, 5]);
let value10 = BigInt('15822401018187971967863');
// 以下三种方式可用
const valueBucket4: relationalStore.ValuesBucket = {
  'NAME': value6,
  'AGE': value7,
  'SALARY': value8,
  'CODES': value9,
  'IDENTITY': value10,
};
const valueBucket5: relationalStore.ValuesBucket = {
  NAME: value6,
  AGE: value7,
  SALARY: value8,
  CODES: value9,
  IDENTITY: value10,
};
const valueBucket6: relationalStore.ValuesBucket = {
  "NAME": value6,
  "AGE": value7,
  "SALARY": value8,
  "CODES": value9,
  "IDENTITY": value10,
};

// 修改数据
let predicates1 = new relationalStore.RdbPredicates('EMPLOYEE'); // 创建表'EMPLOYEE'的predicates
predicates1.equalTo('NAME', 'Lisa'); // 匹配表'EMPLOYEE'中'NAME'为'Lisa'的字段
if (store !== undefined) {
  (store as relationalStore.RdbStore).update(valueBucket4, predicates1, (err: BusinessError, rows: number) => {
    if (err) {
      console.error(`Failed to update data. Code:${err.code}, message:${err.message}`);
      return;
    }
    console.info(`Succeeded in updating data. row count: ${rows}`);
  })
}

// 删除数据
predicates1 = new relationalStore.RdbPredicates('EMPLOYEE');
predicates1.equalTo('NAME', 'Lisa');
if (store !== undefined) {
  (store as relationalStore.RdbStore).delete(predicates1, (err: BusinessError, rows: number) => {
    if (err) {
      console.error(`Failed to delete data. Code:${err.code}, message:${err.message}`);
      return;
    }
    console.info(`Delete rows: ${rows}`);
  })
}
```

开发步骤

- 根据谓词指定的查询条件查找数据。
  调用query()方法查找数据，返回一个ResultSet结果集。示例代码如下所示：

```
let predicates2 = new relationalStore.RdbPredicates('EMPLOYEE');
predicates2.equalTo('NAME', 'Rose');
if (store !== undefined) {
  (store as relationalStore.RdbStore).query(predicates2, ['ID', 'NAME', 'AGE', 'SALARY', 'IDENTITY'], (err: BusinessError, resultSet) => {
    if (err) {
      console.error(`Failed to query data. Code:${err.code}, message:${err.message}`);
      return;
    }
    console.info(`ResultSet column names: ${resultSet.columnNames}, column count: ${resultSet.columnCount}`);
    // resultSet是一个数据集合的游标，默认指向第-1个记录，有效的数据从0开始。
    while (resultSet.goToNextRow()) {
      const id = resultSet.getLong(resultSet.getColumnIndex('ID'));
      const name = resultSet.getString(resultSet.getColumnIndex('NAME'));
      const age = resultSet.getLong(resultSet.getColumnIndex('AGE'));
      const salary = resultSet.getDouble(resultSet.getColumnIndex('SALARY'));
      const identity = resultSet.getValue(resultSet.getColumnIndex('IDENTITY'));
      console.info(`id=${id}, name=${name}, age=${age}, salary=${salary}, identity=${identity}`);
    }
    // 释放数据集的内存
    resultSet.close();
  })
}
```

- **说明**
  当应用完成查询数据操作，不再使用结果集（ResultSet）时，请及时调用close方法关闭结果集，释放系统为其分配的内存。

- 在同路径下备份数据库。示例代码如下所示：

```
if (store !== undefined) {
  // "Backup.db"为备份数据库文件名，默认在RdbStore同路径下备份。也可指定路径：customDir + "backup.db"
  (store as relationalStore.RdbStore).backup("Backup.db", (err: BusinessError) => {
    if (err) {
      console.error(`Failed to backup RdbStore. Code:${err.code}, message:${err.message}`);
      return;
    }
    console.info(`Succeeded in backing up RdbStore.`);
  })
}
```

- 从备份数据库中恢复数据。示例代码如下所示：

```
if (store !== undefined) {
  (store as relationalStore.RdbStore).restore("Backup.db", (err: BusinessError) => {
    if (err) {
      console.error(`Failed to restore RdbStore. Code:${err.code}, message:${err.message}`);
      return;
    }
    console.info(`Succeeded in restoring RdbStore.`);
  })
}
```

开发步骤

- 若数据库文件损坏，需要重建数据库。
  进行开库及增删改查等操作时抛出错误码14800011表示数据库文件损坏。重建数据库的示例代码如下所示：

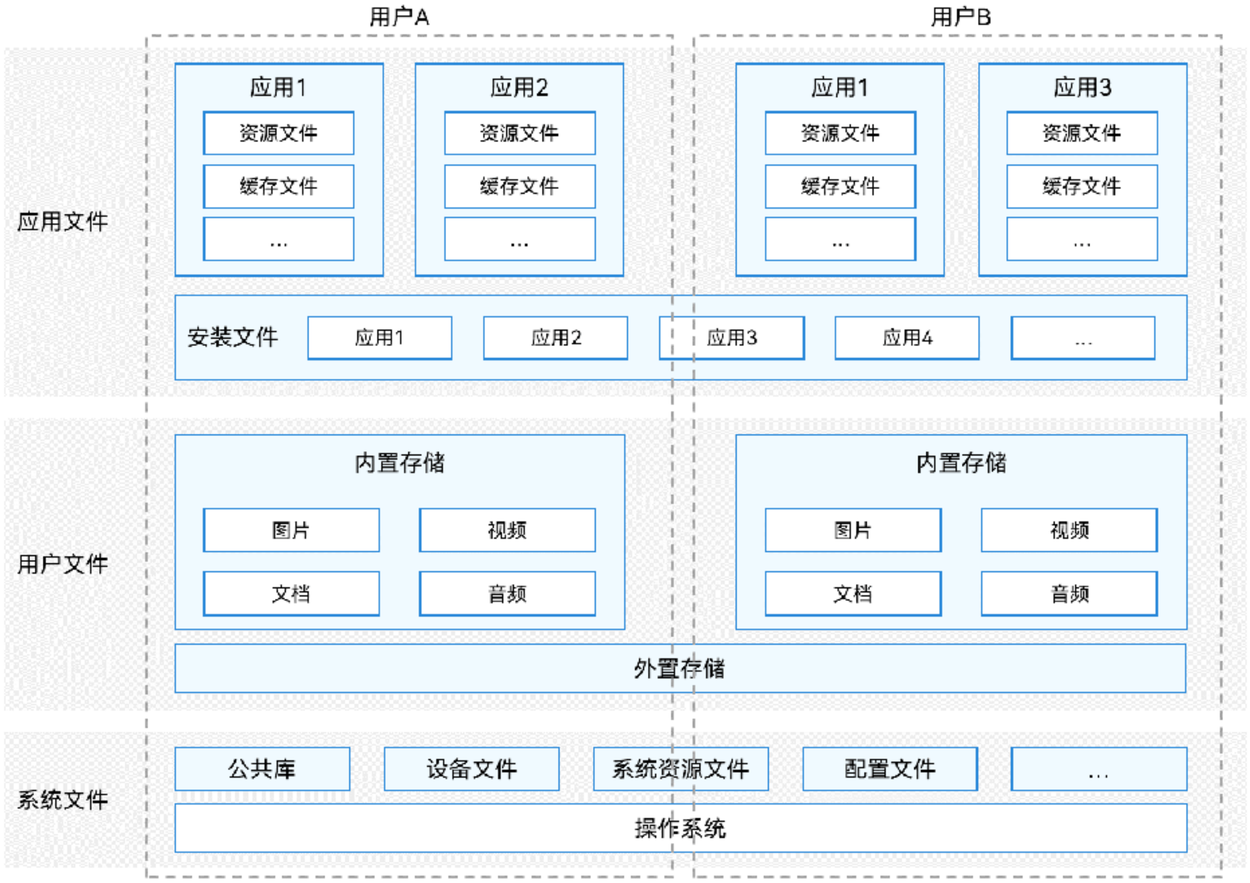
```
if (store !== undefined) {
  // 数据库文件损坏后需要关闭所有数据库连接和结果集，使用store.close()方法或把对象置为null
  (store as relationalStore.RdbStore).close();
  store = undefined;
  // 将config.allowRebuild配置为true，重新调用getRdbStore开库
  const STORE_CONFIG: relationalStore.StoreConfig = {
    name: 'RdbTest.db',
    securityLevel: relationalStore.SecurityLevel.S1,
    allowRebuild: true
  };

  relationalStore.getRdbStore(this.context, STORE_CONFIG).then(async (rdbStore: relationalStore.RdbStore) => {
    store = rdbStore;
    console.info('Get RdbStore successfully.')
  }).catch((err: BusinessError) => {
    console.error(`Get RdbStore failed, code is ${err.code},message is ${err.message}`);
  })

  if (store !== undefined) {
    // 查看重建结果
    if ((store as relationalStore.RdbStore).rebuilt === relationalStore.RebuildType.REBUILT) {
      console.info('Succeeded in rebuilding RdbStore.');
      // 将损坏前备份的数据恢复到新数据库中
      (store as relationalStore.RdbStore).restore("Backup.db", (err: BusinessError) => {
        if (err) {
          console.error(`Failed to restore RdbStore. Code:${err.code}, message:${err.message}`);
          return;
        }
        console.info(`Succeeded in restoring RdbStore.`);
      })
    }
  }
}
```

- 删除数据库。
  调用deleteRdbStore()方法，删除数据库及数据库相关文件。示例代码如下：
  Stage模型示例：

```
relationalStore.deleteRdbStore(this.context, 'RdbTest.db', (err: BusinessError) => {
  if (err) {
    console.error(`Failed to delete RdbStore. Code:${err.code}, message:${err.message}`);
    return;
  }
  console.info('Succeeded in deleting RdbStore.');
});
```

案例： Rdb

Core File Kit 文件管理



在Core File Kit套件中，按文件所有者的不同，有如下文件分类模型，其示意图如下面文件分类模型示意图：

- 应用文件：文件所有者为应用，包括应用安装文件、应用资源文件、应用缓存文件等。
- 用户文件：文件所有者为登录到该终端设备的用户，包括用户私有的图片、视频、音频、文档等。
- 系统文件：与应用和用户无关的其它文件，包括公共库、设备文件、系统资源文件等。这类文件不需要开发者进行文件管理，本文不展开介绍。

按文件系统管理的文件存储位置（数据源位置）的不同，有如下文件系统分类模型：

- 本地文件系统：提供本地设备或外置存储设备（如U盘、移动硬盘）的文件访问能力。本地文件系统是最基本的文件系统，本文不展开介绍。
- 分布式文件系统：提供跨设备的文件访问能力。所谓跨设备，指文件不一定存储在本地设备或外置存储设备，而是通过计算机网络与其它分布式设备相连。

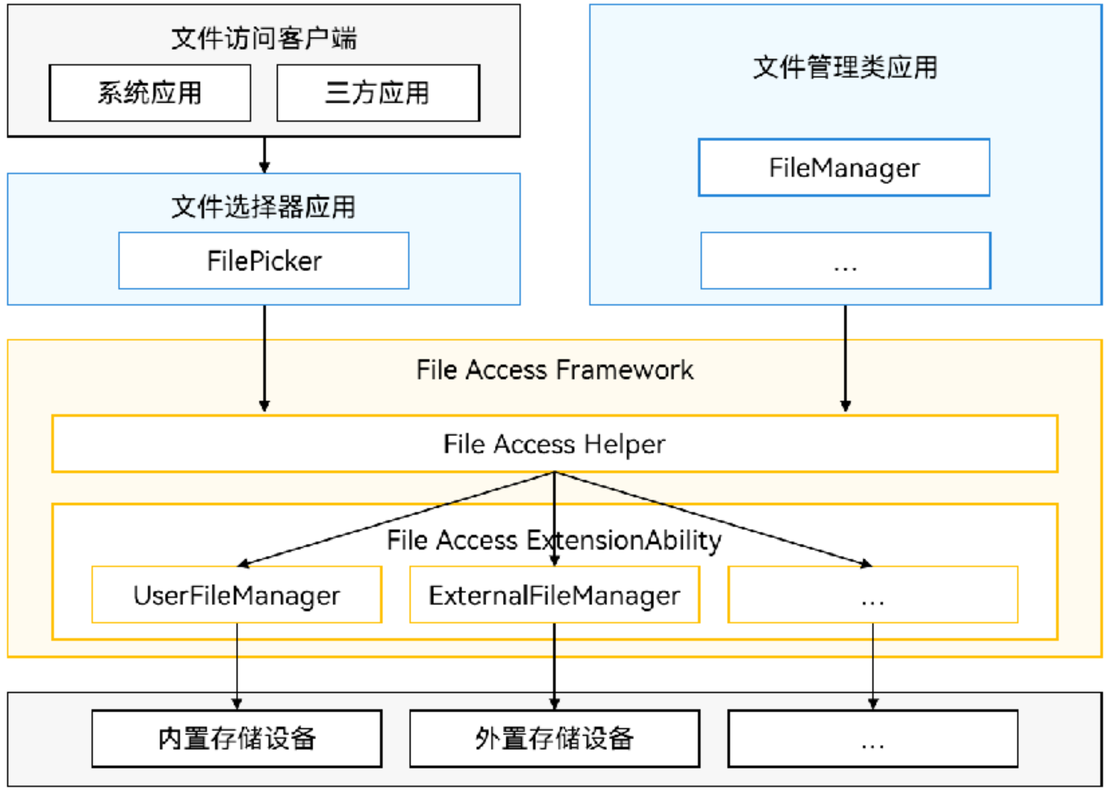
文件类型

特点

- 沙箱隔离：
  访问和管理应用文件，对于每个应用，系统会在内部存储空间映射出一个专属的“应用沙箱目录”，它是“应用文件目录”与一部分系统文件（应用运行必需的少量系统文件）所在的目录组成的集合。有以下优点：
  - 隔离性：应用沙箱提供了一个完全隔离的环境，使用户可以安全地访问应用文件。
  - 安全性：应用沙箱限制了应用可见的数据的最小范围，保护了应用文件的安全。
-
- 应用分享：
  应用之间可以通过分享URI（Uniform Resource Identifier）或文件描述符FD（File Descriptor）的方式，进行文件共享。有以下优点：
  - 便携性：应用之间进行文件分享，省去了用户在多个应用间切换的麻烦，简化了操作步骤，提高了效率。
  - 高效性：应用间的文件分享能够更快地完成文件的传输，减少了因多次跳转和等待而浪费的时间。
  - 数据一致性：应用间的文件分享能够确保数据的完整性和一致性，避免数据在传输过程中出现损坏或丢失的情况。
  - 安全性：应用间的文件分享可以确保文件的安全性，避免文件被非法获取或篡改。同时，通过文件授权访问的方式，可以进一步增强文件的安全性。

文件访问框架

- 各类系统应用或三方应用（即图中的文件访问客户端）若需访问用户文件，如选择一张照片或保存多个文档等，可以通过拉起“文件选择器应用”来实现。
- FilePicker：系统预置应用，提供文件访问客户端选择和保存文件的能力，且不需要配置任何权限。FilePicker的使用指导请参见选择用户文件。
- FileManager：对于设备开发者，还可以按需开发自己的文件选择器或文件管理器应用。该功能不向三方应用开放。
- File Access Framework（用户文件访问框架）的主要功能模块如下：
  - File Access Helper：提供给文件管理器和文件选择器访问用户文件的API接口。
  - File Access ExtensionAbility：提供文件访问框架能力，由内卡文件管理服务UserFileManager和外卡文件管理服务ExternalFileManager组成，实现对应的文件访问功能。
    - UserFileManager：内卡文件管理服务，基于File Access ExtensionAbility框架实现，用于管理内置存储设备上的文件。
    - ExternalFileManager：外卡文件管理服务，基于File Access ExtensionAbility框架实现，用于管理外置存储设备上的文件。



应用文件

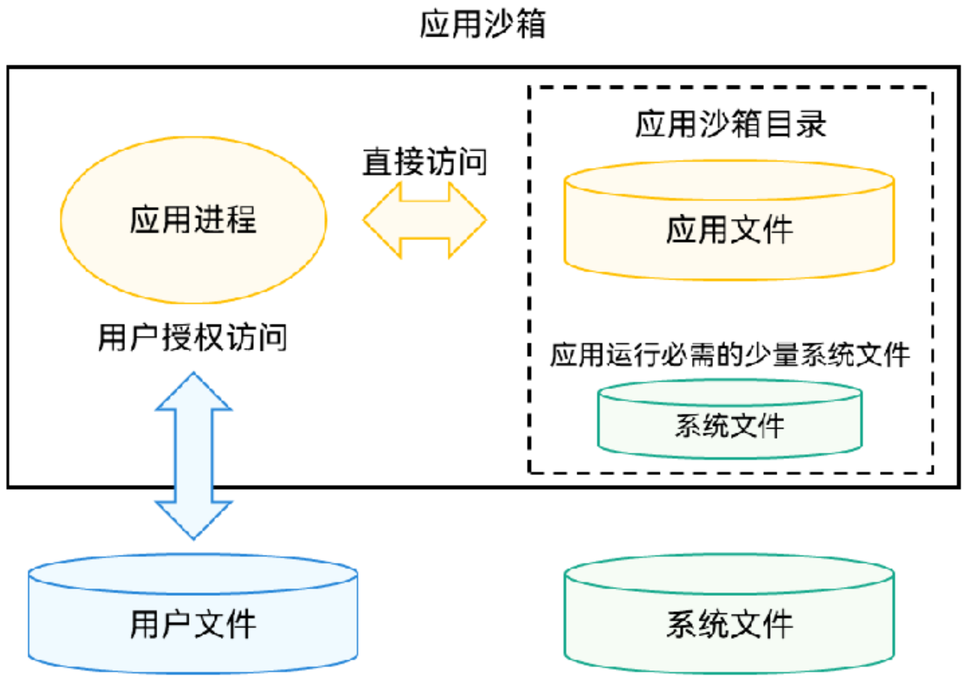
应用文件：文件所有者为应用，包括应用安装文件、应用资源文件、应用缓存文件等。

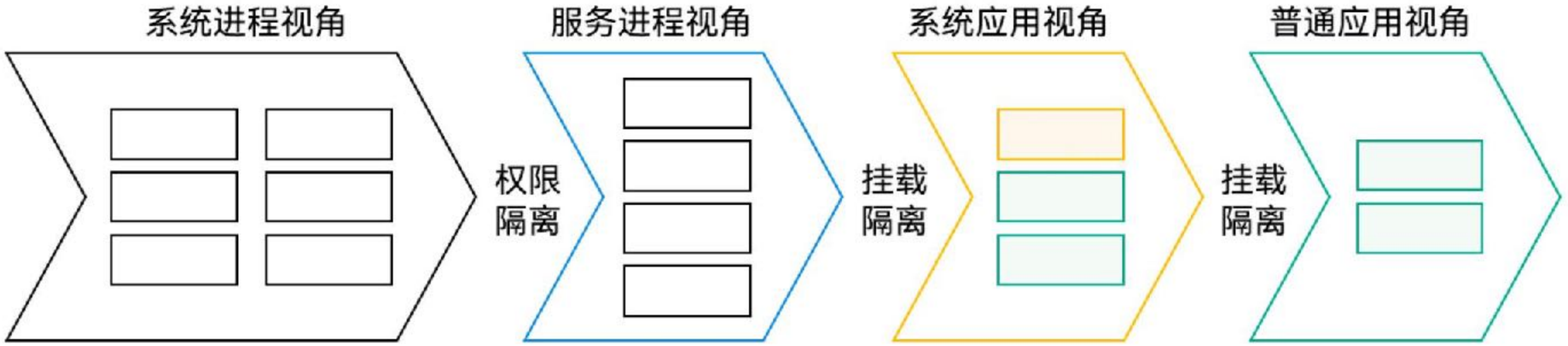
- 设备上应用所使用及存储的数据，以文件、键值对、数据库等形式保存在一个应用专属的目录内。该专属目录我们称为“应用文件目录”，该目录下所有数据以不同的文件格式存放，这些文件即应用文件。
- “应用文件目录”与一部分系统文件（应用运行必须使用的系统文件）所在的目录组成了一个集合，该集合称为“应用沙箱目录”，代表应用可见的所有目录范围。因此“应用文件目录”是在“应用沙箱目录”内的。
- 系统文件及其目录对于应用是只读的；应用仅能保存文件到“应用文件目录”下，根据目录的使用规范和注意事项来选择将数据保存到不同的子目录中。

应用沙箱

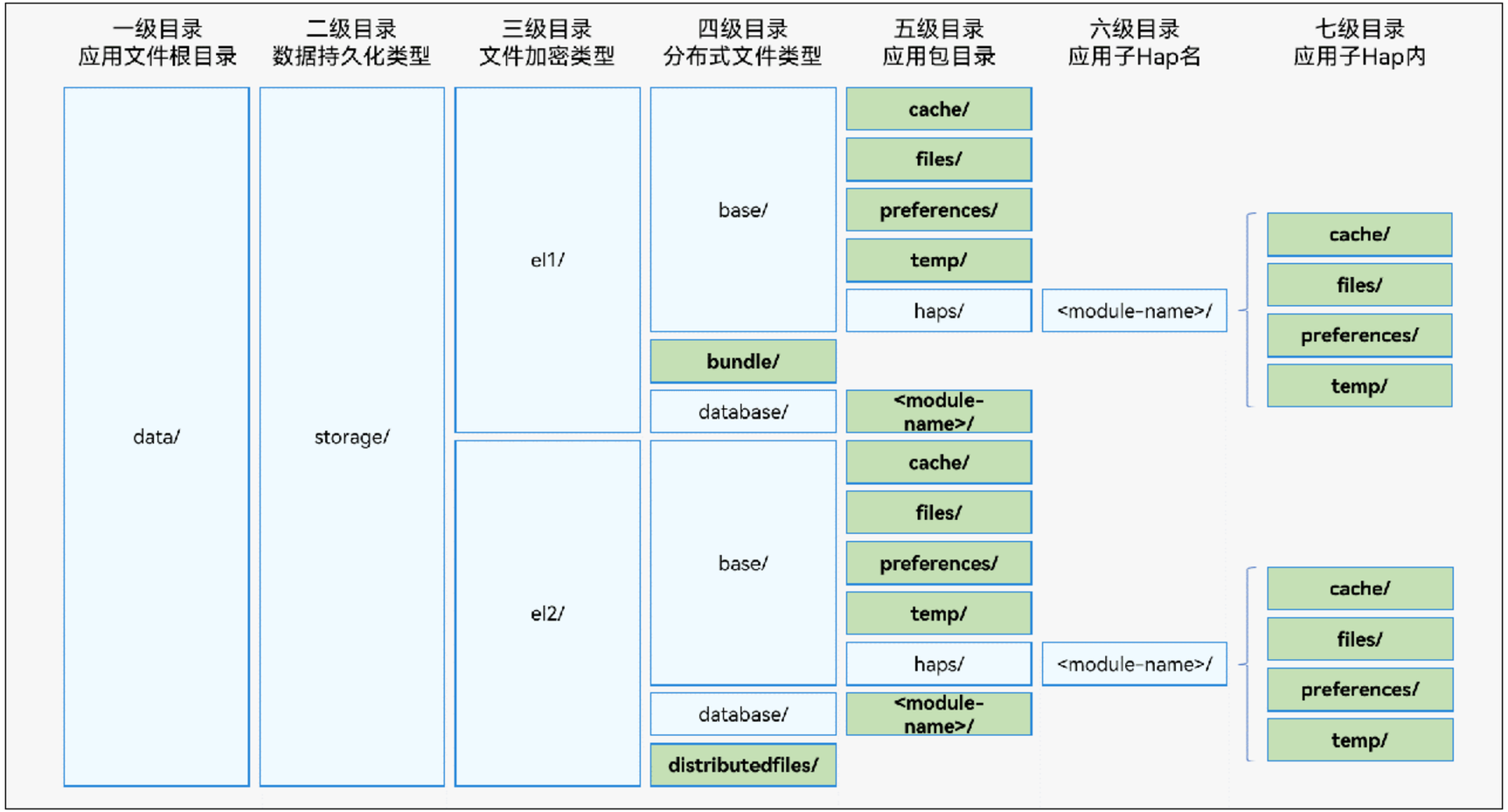
应用沙箱是一种以安全防护为目的的隔离机制，避免数据受到恶意路径穿越访问。在这种沙箱的保护机制下，应用可见的目录范围即为“应用沙箱目录”。

- 对于每个应用，系统会在内部存储空间映射出一个专属的“应用沙箱目录”，它是“应用文件目录”与一部分系统文件（应用运行必需的少量系统文件）所在的目录组成的集合。
- 应用沙箱限制了应用可见的数据范围。在“应用沙箱目录”中，应用仅能看到自己的应用文件以及少量的系统文件（应用运行必需的少量系统文件）。因此，本应用的文件也不为其他应用可见，从而保护了应用文件的安全。
- 应用可以在“应用文件目录”下保存和处理自己的应用文件；系统文件及其目录对于应用是只读的；而应用若需访问用户文件，则需要通过特定API同时经过用户的相应授权才能进行。





应用沙箱路径



- 一级目录data/：代表应用文件目录。
- 二级目录storage/：代表本应用持久化文件目录。
- 三级目录el1/、el2/：代表不同文件加密类型。
  - el1，设备级加密区：设备开机后即可访问的数据区。
  - el2，用户级加密区：设备开机后，若处于无密码状态，可直接访问；若处于有密码状态，则需要至少一次解锁对应用户的锁屏界面后（密码、指纹、人脸等方式解锁皆可），才能够访问的加密数据区。
    应用如无特殊需要，应将数据存放在el2加密目录下，以尽可能保证数据安全。但是对于某些场景，一些应用文件需要在用户解锁前就可被访问，例如时钟、闹铃、壁纸等，此时应用需要将这些文件存放到设备级加密区（el1）。切换应用文件加密类型目录的方法请参见获取和修改加密分区。
- 四级、五级目录：
  通过ApplicationContext可以获取distributedfiles目录或base下的files、cache、preferences、temp等目录的应用文件路径，应用全局信息可以存放在这些目录下。
  通过UIAbilityContext、AbilityStageContext、ExtensionContext可以获取HAP级别应用文件路径。HAP信息可以存放在这些目录下，存放在此目录的文件会跟随HAP的卸载而删除，不会影响App级别目录下的文件。在开发态，一个应用包含一个或者多个HAP，详见Stage模型应用程序包结构。

应用文件目录与应用文件路径

应用沙箱路径和真实物理路径的对应关系

沙箱路径下读写文件，经过映射转换，实际读写的是真实物理路径中的应用文件，应用沙箱路径与真实物理路径对应关示。

JSERID>当前固定为100，<EXTENSIONPATH>为moduleName-extensionName。应用是否以Extension独立沙箱运行nsionAbility组件。

沙箱路径	物理路径
/storage/el1/bundle	应用安装包目录： /data/app/el1/bundle/public/<PACKAGENAME>
/storage/el1/base	应用el1级别加密数据目录： - 非独立沙箱运行的应用：/data/app/el1/<USERID>/base/<PACKAGENAME> - 以独立沙箱运行的Extension应用： /data/app/el1/<USERID>/base/+extension-<EXTENSIONPATH>+<PACKAGENAME>
/storage/el2/base	应用el2级别加密数据目录： - 非独立沙箱运行的应用：/data/app/el2/<USERID>/base/<PACKAGENAME> - 以独立沙箱运行的Extension应用： /data/app/el2/<USERID>/base/+extension-<EXTENSIONPATH>+<PACKAGENAME>
/storage/el1/database	应用el1级别加密数据库目录： - 非独立沙箱运行的应用：/data/app/el1/<USERID>/database/<PACKAGENAM - 以独立沙箱运行的Extension应用： /data/app/el1/<USERID>/database/+extension-<EXTENSIONPATH>+<PACKAGENAME>
/storage/el2/database	应用el2级别加密数据库目录： - 非独立沙箱运行的应用：/data/app/el2/<USERID>/database/<PACKAGENAM - 以独立沙箱运行的Extension应用： /data/app/el2/<USERID>/database/+extension-<EXTENSIONPATH>+<PACKAGENAME>
/storage/el2/distributedfiles	/mnt/hmdfs/<USERID>/account/merge_view/data/<PACGENAME>

开发步骤

新建并读写一个文件

以下示例代码演示了如何新建一个文件并对其读写。

```
// pages/xxx.ets
import { fileIo as fs, ReadOptions } from '@kit.CoreFileKit';
import { common } from '@kit.AbilityKit';
import { buffer } from '@kit.ArkTS';

// 获取应用文件路径
let context = getContext(this) as common.UIAbilityContext;
let filesDir = context.filesDir;

function createFile(): void {
  // 新建并打开文件
  let file = fs.openSync(filesDir + '/test.txt', fs.OpenMode.READ_WRITE | fs.OpenMode.CREATE);
  // 写入一段内容至文件
  let writeLen = fs.writeSync(file.fd, "Try to write str.");
  console.info("The length of str is: " + writeLen);
  // 从文件读取一段内容
  let arrayBuffer = new ArrayBuffer(1024);
  let readOptions: ReadOptions = {
    offset: 0,
    length: arrayBuffer.byteLength
  };
  let readLen = fs.readSync(file.fd, arrayBuffer, readOptions);
  let buf = buffer.from(arrayBuffer, 0, readLen);
  console.info("the content of file: " + buf.toString());
  // 关闭文件
  fs.closeSync(file);
}
```

读取文件内容并写入到另一个文件

以下示例代码演示了如何从一个文件读写内容到另一个文件。

```
// pages/xxx.ets
import { fileIo as fs, ReadOptions, WriteOptions } from '@kit.CoreFileKit';
import { common } from '@kit.AbilityKit';

// 获取应用文件路径
let context = getContext(this) as common.UIAbilityContext;
let filesDir = context.filesDir;

function readWriteFile(): void {
  // 打开文件
  let srcFile = fs.openSync(filesDir + '/test.txt', fs.OpenMode.READ_WRITE | fs.OpenMode.CREATE);
  let destFile = fs.openSync(filesDir + '/destFile.txt', fs.OpenMode.READ_WRITE | fs.OpenMode.CREATE);
  // 读取源文件内容并写入至目的文件
  let bufSize = 4096;
  let readSize = 0;
  let buf = new ArrayBuffer(bufSize);
  let readOptions: ReadOptions = {
    offset: readSize,
    length: bufSize
  };
  let readLen = fs.readSync(srcFile.fd, buf, readOptions);
  while (readLen > 0) {
    readSize += readLen;
    let writeOptions: WriteOptions = {
      length: readLen
    };
    fs.writeSync(destFile.fd, buf, writeOptions);
    readOptions.offset = readSize;
    readLen = fs.readSync(srcFile.fd, buf, readOptions);
  }
  // 关闭文件
  fs.closeSync(srcFile);
  fs.closeSync(destFile);
}
```

说明

使用读写接口时，需注意可选项参数offset的设置。对于已存在且读写过的文件，文件偏移指针默认在上次读写操作的终止位置。

开发步骤

以流的形式读写文件

以下示例代码演示了如何使用流接口进行文件读写。

```
// pages/xxx.ets
import { fileIo as fs, ReadOptions } from '@kit.CoreFileKit';
import { common } from '@kit.AbilityKit';

// 获取应用文件路径
let context = getContext(this) as common.UIAbilityContext;
let filesDir = context.filesDir;

async function readWriteFileWithStream(): Promise<void> {
  // 打开文件流
  let inputStream = fs.createStreamSync(filesDir + '/test.txt', 'r+');
  let outputStream = fs.createStreamSync(filesDir + '/destFile.txt', "w+");
  // 以流的形式读取源文件内容并写入目的文件
  let bufSize = 4096;
  let readSize = 0;
  let buf = new ArrayBuffer(bufSize);
  let readOptions: ReadOptions = {
    offset: readSize,
    length: bufSize
  };
  let readLen = await inputStream.read(buf, readOptions);
  readSize += readLen;
  while (readLen > 0) {
    const writeBuf = readLen < bufSize ? buf.slice(0, readLen) : buf;
    await outputStream.write(writeBuf);
    readOptions.offset = readSize;
    readLen = await inputStream.read(buf, readOptions);
    readSize += readLen;
  }
  // 关闭文件流
  inputStream.closeSync();
  outputStream.closeSync();
}
```

说明

使用流接口时，需注意流的及时关闭。同时流的异步接口应严格遵循异步接口使用规范，避免同步、异步接口混用。流接口不支持并发读写。

查看文件列表

以下示例代码演示了如何查看文件列表。

```
import { fileIo as fs, Filter, ListFileOptions } from '@kit.CoreFileKit';
import { common } from '@kit.AbilityKit';

// 获取应用文件路径
let context = getContext(this) as common.UIAbilityContext;
let filesDir = context.filesDir;

// 查看文件列表
function getListFile(): void {
  let listFileOption: ListFileOptions = {
    recursion: false,
    listNum: 0,
    filter: {
      suffix: [".png", ".jpg", ".txt"],
      displayName: ["test*"],
      fileSizeOver: 0,
      lastModifiedAfter: new Date(0).getTime()
    }
  };
  let files = fs.listFileSync(filesDir, listFileOption);
  for (let i = 0; i < files.length; i++) {
    console.info(`The name of file: ${files[i]}`);
  }
}
```

开发步骤

使用文件流

以下示例代码演示了如何使用文件可读流，文件可写流。

```
// pages/xxx.ets
import { fileIo as fs } from '@kit.CoreFileKit';
import { common } from '@kit.AbilityKit';

// 获取应用文件路径
let context = getContext(this) as common.UIAbilityContext;
let filesDir = context.filesDir;

function copyFileWithReadable(): void {
  // 创建文件可读流
  const rs = fs.createReadStream(`${filesDir}/read.txt`);
  // 创建文件可写流
  const ws = fs.createWriteStream(`${filesDir}/write.txt`);
  // 暂停模式拷贝文件
  rs.on('readable', () => {
    const data = rs.read();
    if (!data) {
      return;
    }
    ws.write(data);
  });
}

function copyFileWithData(): void {
  // 创建文件可读流
  const rs = fs.createReadStream(`${filesDir}/read.txt`);
  // 创建文件可写流
  const ws = fs.createWriteStream(`${filesDir}/write.txt`);
  // 流动模式拷贝文件
  rs.on('data', (emitData) => {
    const data = emitData?.data;
    if (!data) {
      return;
    }
    ws.write(data as Uint8Array);
  });
}
```

以下代码演示了如何使用文件哈希流。

```
// pages/xxx.ets
import { fileIo as fs } from '@kit.CoreFileKit';
import { hash } from '@kit.CoreFileKit';
import { common } from '@kit.AbilityKit';

// 获取应用文件路径
let context = getContext(this) as common.UIAbilityContext;
let filesDir = context.filesDir;

function hashFileWithStream() {
  const filePath = `${filesDir}/test.txt`;
  // 创建文件可读流
  const rs = fs.createReadStream(filePath);
  // 创建哈希流
  const hs = hash.createHash('sha256');
  rs.on('data', (emitData) => {
    const data = emitData?.data;
    hs.update(new Uint8Array(data?.split('').map((x: string) => x.charCodeAt(0))).buffer);
  });
  rs.on('close', async () => {
    const hashResult = hs.digest();
    const fileHash = await hash.hash(filePath, 'sha256');
    console.info(`hashResult: ${hashResult}, fileHash: ${fileHash}`);
  });
}
```

应用文件分享

应用文件分享是应用之间通过分享URI（Uniform Resource Identifier）或文件描述符FD（File Descriptor）的方式，进行文件共享的过程。

- 基于URI分享方式，应用可分享单个文件，通过ohos.app.ability.wantConstant的wantConstant.Flags接口以只读或读写权限授权给其他应用。应用可通过ohos.file.fs的open接口打开URI，并进行读写操作。当前仅支持临时授权，分享给其他应用的文件在被分享应用退出时权限被收回。
- 基于FD分享方式，应用可分享单个文件，通过ohos.file.fs的open接口以指定权限授权给其他应用。应用从Want中解析拿到FD后可通过ohos.file.fs的读写接口对文件进行读写。

文件URI规范

文件URI的格式为：
格式为file://<bundleName>/<path>

- file：文件URI的标志。
- bundleName：该文件资源的属主。
- path：文件资源在应用沙箱中的路径。

开发步骤

- 获取文件在应用沙箱中的路径，并转换为文件URI。

```
import { UIAbility } from '@kit.AbilityKit';
import { fileUri } from '@kit.CoreFileKit';
import { window } from '@kit.ArkUI';

export default class EntryAbility extends UIAbility {
  onWindowStageCreate(windowStage: window.WindowStage) {
    // 获取文件的沙箱路径
    let pathInSandbox = this.context.filesDir + "/test1.txt";
    // 将沙箱路径转换为uri
    let uri = fileUri.getUriFromPath(pathInSandbox);
    // 获取的uri为"file://com.example.demo/data/storage/el2/base/files/test1.txt"
  }
}
```

- 设置获取文件的权限以及选择要分享的应用。
  分享文件给其他应用需要使用startAbility接口，将获取到的URI填充在want的参数uri中，标注URI的文件类型，type字段可参考Want属性，并通过设置want的flag来设置对应的读写权限，action字段配置为"ohos.want.action.sendData"表示进行应用文件分享，开发示例如下。
  说明
  写权限分享时，同时授予读权限。

```
import { fileUri } from '@kit.CoreFileKit';
import { window } from '@kit.ArkUI';
import { wantConstant } from '@kit.AbilityKit';
import { UIAbility } from '@kit.AbilityKit';
import { Want } from '@kit.AbilityKit';
import { BusinessError } from '@kit.BasicServicesKit';

export default class EntryAbility extends UIAbility {
  onWindowStageCreate(windowStage: window.WindowStage) {
    // 获取文件沙箱路径
    let filePath = this.context.filesDir + '/test1.txt';
    // 将沙箱路径转换为uri
    let uri = fileUri.getUriFromPath(filePath);
    let want: Want  = {
      // 配置被分享文件的读写权限，例如对被分享应用进行读写授权
      flags: wantConstant.Flags.FLAG_AUTH_WRITE_URI_PERMISSION | wantConstant.Flags.FLAG_AUTH_READ_URI_PERMISSION,
      // 配置分享应用的隐式拉起规则
      action: 'ohos.want.action.sendData',
      uri: uri,
      type: 'text/plain'
    }
    this.context.startAbility(want)
      .then(() => {
        console.info('Invoke getCurrentBundleStats succeeded.');
      })
      .catch((err: BusinessError) => {
        console.error(`Invoke startAbility failed, code is ${err.code}, message is ${err.message}`);
      });
  }
  // ...
}
```

开发步骤

使用其他应用分享的文件

被分享应用需要在module.json5配置文件的actions标签的值配置为"ohos.want.action.sendData"，表示接收应用分享文件，配置uris字段，表示接收URI的类型，即只接收其他应用分享该类型的URI，如下表示本应用只接收scheme为file，类型为txt的文件，示例如下。

```
{
  "module": {
    ...
    "abilities": [
      {
        ...
        "skills": [
          {
            ...
            "actions": [
              "ohos.want.action.sendData"
            ],
            "uris": [
              {
                "scheme": "file",
                "type": "text/plain"
              }
            ]
          }
        ]
      }
    ]
  }
}
```

被分享方的UIAbility被启动后，可以在其onCreate()或者onNewWant回调中获取传入的Want参数信息。

通过接口want的参数获取分享文件的URI，获取文件URI后通过fs.open接口打开文件，获取对应的file对象后，可对文件进行读写操作。

```
// xxx.ets
import { fileIo as fs } from '@kit.CoreFileKit';
import { Want } from '@kit.AbilityKit';
import { BusinessError } from '@kit.BasicServicesKit';

function getShareFile() {
  try {
    let want: Want = {}; // 此处实际使用时应该修改为获取到的分享方传递过来的want信息

    // 从want信息中获取uri字段
    let uri = want.uri;
    if (uri == null || uri == undefined) {
      console.info('uri is invalid');
      return;
    }
    try {
      // 根据需要对被分享文件的URI进行相应操作。例如读写的方式打开URI获取file对象
      let file = fs.openSync(uri, fs.OpenMode.READ_WRITE);
      console.info('open file successfully!');
    } catch (err) {
      let error: BusinessError = err as BusinessError;
      console.error(`Invoke openSync failed, code is ${error.code}, message is ${error.message}`);
    }
  } catch (error) {
    let err: BusinessError = error as BusinessError;
    console.error(`Invoke openSync failed, code is ${err.code}, message is ${err.message}`);
  }
}
```

用户文件

- 用户文件：文件所有者为登录到该终端设备的用户，包括用户私有的图片、视频、音频、文档等。
  - 用户文件存放在用户目录下，归属于该设备上登录的用户。
  - 用户文件存储位置主要分为内置存储、外置存储。
  - 应用对用户文件的创建、访问、删除等行为，需要提前获取用户授权，或由用户操作完成。

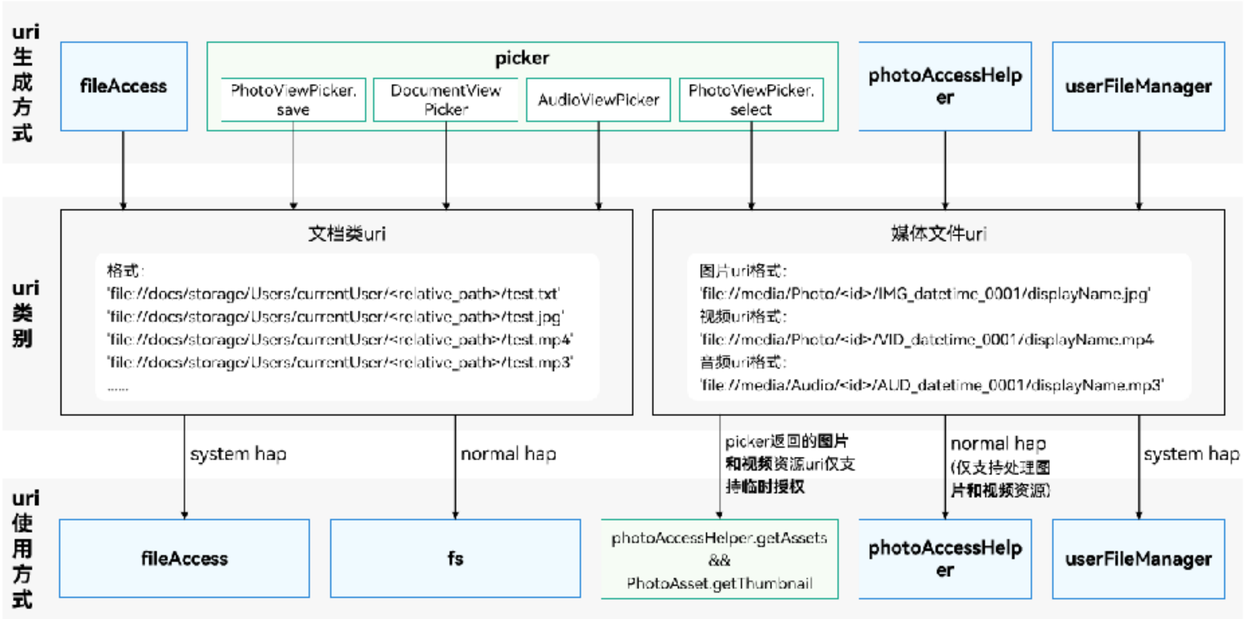
- **用户文件存储位置**

- **内置存储**

- 内置存储，是指用户文件存储在终端设备的内部存储设备（空间）上。内置存储设备无法被移除。内置存储的用户文件主要有：
  - 用户特有的文件：这部分文件归属于登录该设备的用户，不同用户登录后，仅可看到该用户自己的文件。
    按照这些文件的特征/属性，以及用户/应用的使用习惯，可分为：
    - 图片/视频类媒体文件
      所具有的特征包括拍摄时间、地点、旋转角度、文件宽高等信息，以媒体文件的形式存储在系统中，通常是以所有文件、相册的形式对外呈现，不会展示其在系统中存储的具体位置。
    - 音频类媒体文件
      所具有的特征包括所属专辑、音频创作者、持续时间等信息，以媒体文件的形式存储在系统中，通常会以所有文件、专辑、作家等形式对外部呈现，不会展示其在系统中存储的具体位置。
    - 其他文件（统称为文档类文件）
      以普通文件的形式存储在系统中，该类文件既包括普通的文本文件、压缩文件等，又包括以普通文件形式存储的图片/视频、音频文件，该类文件通常是以目录树的形式对外展示。
  - 多用户共享的文件：用户可以通过将文件放在共享文件区，实现多个用户之间文件的共享访问。
    共享文件区的文件，也是以普通文件的形式存储在系统中，以目录树的形式对外展示。
- **外置存储**

- 外置存储，是指用户文件存储在外置可插拔设备上（如SD卡、U盘等）。外置存储设备上的文件，和内置存储设备共享区文件一样，可以被所有登录到系统中的用户看到。
- 外置存储设备具备可插拔属性，因此系统提供了设备插拔事件的监听及挂载功能，用于管理外置存储设备，，该部分功能仅对系统应用开放。
- 外置存储设备上的文件，全部以普通文件的形式呈现，和内置存储设备上的文档类文件一样，采用目录树的形式对外展示。

用户文件URI


uri生成方式
fileAccess
picker
PhotoViewPicker.save
DocumentViewPicker
AudioViewPicker
PhotoViewPicker.select
photoAccessHelper
userFileManager
uri类别
文档类uri
格式：
'file://docs/storage/Users/currentUser/<relative_path>/test.txt'
'file://docs/storage/Users/currentUser/<relative_path>/test.jpg'
'file://docs/storage/Users/currentUser/<relative_path>/test.mp4'
'file://docs/storage/Users/currentUser/<relative_path>/test.mp3'
.....
媒体文件uri
图片uri格式：
'file://media/Photo/<id>/IMG_datetime_0001/displayName.jpg'
视频uri格式：
'file://media/Photo/<id>/VID_datetime_0001/displayName.mp4
音频uri格式：
'file://media/Audio/<id>/AUD_datetime_0001/displayName.mp3'
system hap
normal hap
picker返回的图片和视频资源uri仅支持临时授权
normal hap(仅支持处理图片和视频资源)
system hap
uri使用方式
fileAccess
fs
photoAccessHelper.getAssets && PhotoAsset.getThumbnail
photoAccessHelper
userFileManager

文档类uri介绍

- 文档类uri的格式类型为：
- ‘file://docs/storage/Users/currentUser/<relative_path>/test.txt'

- 其中各个字段表示的含义为：
- Uri字段　说明
- ‘file://docs/storage/Users/currentUser/' 文件管理器的根目录。
- ‘<relative_path>/' 文件在根目录下的相对路径。例如：'Download/'和'Documents/'。
- ‘test.txt' 用户文件系统中存储的文件名，支持的文件类型为文件管理器支持的所有类型，以文件管理器为准。例如txt、jpg、mp4和mp3等格式的文件。

- 文档类uri获取方式
- 通过DocumentViewPicker接口选择或保存文件，返回选择或保存的文件uri。
- 通过AudioViewPicker接口选择或保存文件，返回选择或保存的文件uri。

媒体文件uri

- 媒体文件uri的格式类型为：
  - 图片uri格式：
  - 'file://media/Photo/<id>/IMG_datetime_0001/displayName.jpg'
  - 视频uri格式：
  - 'file://media/Photo/<id>/VID_datetime_0001/displayName.mp4'
  - 音频uri格式：
  - ‘file://media/Audio/<id>/AUD_datetime_0001/displayName.mp3'

- 其中各个字段表示的含义为：
- Uri字段　说明
- ‘file://media' 表示这个uri是媒体文件。
- ‘Photo' Photo表示这个uri是媒体文件中的图片或者视频类文件。
- ‘Audio' 表示这个uri是媒体文件中的音频类文件。
- ‘<id>' 表示在数据库中多个表中处理后的值，并不是指表中的file_id列，注意请不要使用此id去数据库中查询具体文件。
- ‘IMG_datetime_0001' 表示图片文件在用户文件系统中存储的文件名去掉后缀剩下的部分。
- ‘VID_datetime_0001' 表示视频文件在用户文件系统中存储的文件名去掉后缀剩下的部分。
- ‘AUD_datetime_0001' 表示音频文件在用户文件系统中存储的文件名去掉后缀剩下的部分。

- 媒体文件uri获取方式
- 通过PhotoAccessHelper的PhotoViewPicker选择媒体文件，返回选择的媒体文件文件的uri。
- 通过photoAccessHelper模块中的getAssets或createAsset接口获取媒体文件对应文件的uri。

开发步骤

下面为通过临时授权方式使用媒体文件uri进行获取缩略图和读取文件部分信息的示例代码：

```
import { photoAccessHelper } from '@kit.MediaLibraryKit';
import { BusinessError } from '@kit.BasicServicesKit';
import { dataSharePredicates } from '@kit.ArkData';

// 定义一个uri数组，用于接收PhotoViewPicker选择图片返回的uri
let uris: Array<string> = [];
const context = getContext(this);

// 调用PhotoViewPicker.select选择图片
async function photoPickerGetUri() {
  try {
    let PhotoSelectOptions = new photoAccessHelper.PhotoSelectOptions();
    PhotoSelectOptions.MIMEType = photoAccessHelper.PhotoViewMIMETypes.IMAGE_TYPE;
    PhotoSelectOptions.maxSelectNumber = 1;
    let photoPicker = new photoAccessHelper.PhotoViewPicker();
    photoPicker.select(PhotoSelectOptions).then((PhotoSelectResult: photoAccessHelper.PhotoSelectResult) => {
      console.info('PhotoViewPicker.select successfully, PhotoSelectResult uri: ' + JSON.stringify(PhotoSelectResult));
      uris = PhotoSelectResult.photoUris;
    }).catch((err: BusinessError) => {
      console.error('PhotoViewPicker.select failed with err: ' + JSON.stringify(err));
    });
  } catch (error) {
    let err: BusinessError = error as BusinessError;
    console.error('PhotoViewPicker failed with err: ' + JSON.stringify(err));
  }
}

async function uriGetAssets() {
try {
    let phAccessHelper = photoAccessHelper.getPhotoAccessHelper(context);
    let predicates: dataSharePredicates.DataSharePredicates = new dataSharePredicates.DataSharePredicates();
    // 配置查询条件，使用PhotoViewPicker选择图片返回的uri进行查询
    predicates.equalTo('uri', uris[0]);
    let fetchOption: photoAccessHelper.FetchOptions = {
      fetchColumns: [photoAccessHelper.PhotoKeys.WIDTH, photoAccessHelper.PhotoKeys.HEIGHT, photoAccessHelper.PhotoKeys.TITLE, photoAccessHelper.PhotoKeys.DURATION],
      predicates: predicates
    };
    let fetchResult: photoAccessHelper.FetchResult<photoAccessHelper.PhotoAsset> = await phAccessHelper.getAssets(fetchOption);
    // 得到uri对应的PhotoAsset对象，读取文件的部分信息
    const asset: photoAccessHelper.PhotoAsset = await fetchResult.getFirstObject();
    console.info('asset displayName: ', asset.displayName);
    console.info('asset uri: ', asset.uri);
    console.info('asset photoType: ', asset.photoType);
    console.info('asset width: ', asset.get(photoAccessHelper.PhotoKeys.WIDTH));
    console.info('asset height: ', asset.get(photoAccessHelper.PhotoKeys.HEIGHT));
    console.info('asset title: ' + asset.get(photoAccessHelper.PhotoKeys.TITLE));
    // 获取缩略图
    asset.getThumbnail((err, pixelMap) => {
      if (err == undefined) {
        console.info('getThumbnail successful ' + JSON.stringify(pixelMap));
      } else {
        console.error('getThumbnail fail', err);
      }
    });
  } catch (error){
    console.error('uriGetAssets failed with err: ' + JSON.stringify(error));
  }
}
```

选择用户文件

- 用户需要分享文件、保存图片、视频等用户文件时，开发者可以通过系统预置的文件选择器（FilePicker），实现该能力。通过Picker访问相关文件，将拉起对应的应用，引导用户完成界面操作，接口本身无需申请权限。picker获取的uri只具有临时权限，获取持久化权限需要通过FilePicker设置永久授权方式获取。

- 根据用户文件的常见类型，选择器（FilePicker）分别提供以下选项：

  - PhotoViewPicker：适用于图片或视频类型文件的选择与保存（该接口在后续版本不再演进）。请使用PhotoAccessHelper的PhotoViewPicker来选择图片文件。请使用安全控件创建媒体资源。

  - DocumentViewPicker：适用于文件类型文件的选择与保存。DocumentViewPicker对接的选择资源来自于FilePicker, 负责文件类型的资源管理，文件类型不区分后缀，比如浏览器下载的图片、文档等，都属于文件类型。

  - AudioViewPicker：适用于音频类型文件的选择与保存。AudioViewPicker目前对接的选择资源来自于FilePicker。

开发步骤

- 选择文档类文件
- 导入选择器模块和基础文件API模块。

```
import  { picker } from '@kit.CoreFileKit';
import { fileIo as fs } from '@kit.CoreFileKit';
import { common } from '@kit.AbilityKit';
import { BusinessError } from '@kit.BasicServicesKit';
```

- 创建文件类型、文件选择选项实例。

```
const documentSelectOptions = new picker.DocumentSelectOptions();
// 选择文档的最大数目（可选）
documentSelectOptions.maxSelectNumber = 5;
// 指定选择的文件或者目录路径（可选）
documentSelectOptions.defaultFilePathUri = "file://docs/storage/Users/currentUser/test";
// 选择文件的后缀类型['后缀类型描述|后缀类型']（可选） 若选择项存在多个后缀名，则每一个后缀名之间用英文逗号进行分隔（可选），后缀类型名不能超过100,选择所有文件：'所有文件(*.*)|.*';
 documentSelectOptions.fileSuffixFilters = ['图片(.png, .jpg)|.png,.jpg', '文档|.txt', '视频|.mp4', '.pdf'];
//选择是否对指定文件或目录授权，true为授权，当为true时，defaultFilePathUri为必选参数，拉起文管授权界面；false为非授权，拉起常规文管界面（可选）
documentSelectOptions.authMode = true;
```

- 创建文件选择器DocumentViewPicker实例。调用select()接口拉起FilePicker应用界面进行文件选择。

```
let uris: Array<string> = [];
let context = getContext(this) as common.Context; // 请确保 getContext(this) 返回结果为 UIAbilityContext
// 创建文件选择器实例
const documentViewPicker = new picker.DocumentViewPicker(context);

documentViewPicker.select(documentSelectOptions).then((documentSelectResult: Array<string>) => {
  //文件选择成功后，返回被选中文档的uri结果集。
  uris = documentSelectResult;
  console.info('documentViewPicker.select to file succeed and uris are:' + uris);
}).catch((err: BusinessError) => {
  console.error(`Invoke documentViewPicker.select failed, code is ${err.code}, message is ${err.message}`);
})
```

开发步骤

注意

1、使用picker获取的select()返回的uri权限是临时只读权限,待退出应用后台后，获取的临时权限就会失效。
2、如果想要获取持久化权限(仅在2in1设备上生效)，请参考文件持久化授权访问。
3、开发者可以根据结果集中uri做进一步的处理。建议定义一个全局变量保存uri。
4、如有获取元数据需求，可以通过基础文件API和文件URI根据uri获取部分文件属性信息，比如文件大小、访问时间、修改时间、文件名、文件路径等。

- 待界面从FilePicker返回后，使用基础文件API的fs.openSync接口通过uri打开这个文件得到文件描述符(fd)。

```
let uri: string = '';
//这里需要注意接口权限参数是fs.OpenMode.READ_ONLY。
let file = fs.openSync(uri, fs.OpenMode.READ_ONLY);
console.info('file fd: ' + file.fd);
```

- 通过fd使用fs.readSync接口读取这个文件内的数据。

```
let buffer = new ArrayBuffer(4096);
let readLen = fs.readSync(file.fd, buffer);
console.info('readSync data to file succeed and buffer size is:' + readLen);
//读取完成后关闭fd。
fs.closeSync(file);
```

保护用户文件

- 在从网络下载文件到本地、或将已有用户文件另存为新的文件路径等场景下，需要使用FilePicker提供的保存用户文件的能力。picker获取的uri只具有临时权限，获取持久化权限需要通过FilePicker设置永久授权方式获取。

- 对音频、图片、视频、文档类文件的保存操作类似，均通过调用对应picker的save()接口并传入对应的saveOptions来实现。通过Picker访问相关文件，无需申请权限。

- 当前所有picker的save接口都是用户可感知的，具体行为是拉起FilePicker, 将文件保存在系统文件管理器管理的特定目录，与图库管理的资源隔离，无法在图库中看到。

- 如需要在图库中看到所保存的图片、视频资源，请使用用户无感的安全控件创建媒体资源。

开发步骤

保存文档类文件

- 导入选择器模块和基础文件API模块。

```
import { picker } from '@kit.CoreFileKit';
import { fileIo as fs } from '@kit.CoreFileKit';
import { BusinessError } from '@kit.BasicServicesKit';
import { common } from '@kit.AbilityKit';
```

- 创建文档保存选项实例。

```
// 创建文件管理器选项实例
const documentSaveOptions = new picker.DocumentSaveOptions();
// 保存文件名（可选）
documentSaveOptions.newFileNames = ["DocumentViewPicker01.txt"];
// 保存文件类型['后缀类型描述|后缀类型'],选择所有文件：'所有文件(*.*)|.*'（可选），如果选择项存在多个后缀，默认选择第一个。
documentSaveOptions.fileSuffixChoices = ['文档|.txt', '.pdf'];
```

- 创建文件选择器DocumentViewPicker实例。调用save()接口拉起FilePicker界面进行文件保存。

```
let uris: Array<string> = [];
// 请确保 getContext(this) 返回结果为 UIAbilityContext
let context = getContext(this) as common.Context;
// 创建文件选择器实例。
const documentViewPicker = new picker.DocumentViewPicker(context);
//用户选择目标文件夹，用户选择与文件类型相对应的文件夹，即可完成文件保存操作。保存成功后，返回保存文档的uri。
documentViewPicker.save(documentSaveOptions).then((documentSaveResult: Array<string>) => {
  uris = documentSaveResult;
  console.info('documentViewPicker.save to file succeed and uris are:' + uris);
}).catch((err: BusinessError) => {
  console.error(`Invoke documentViewPicker.save failed, code is ${err.code}, message is ${err.message}`);
})
```

注意

1、不能在picker的回调里直接使用此uri进行打开文件操作，需要定义一个全局变量保存URI。
2、使用picker的save()接口获取到URI的权限是临时读写权限,待退出应用后台后，获取的临时权限就会失效。
3、如果想要获取持久化权限(仅在2in1设备上生效)，请参考文件持久化授权访问。
4、可以通过便捷方式，直接将文件保存到Download目录下。

- 待界面从FilePicker返回后，使用基础文件API的fs.openSync接口，通过uri打开这个文件得到文件描述符(fd)。

```
const uri = '';
//这里需要注意接口权限参数是fs.OpenMode.READ_WRITE。
let file = fs.openSync(uri, fs.OpenMode.READ_WRITE);
console.info('file fd: ' + file.fd);
```

- 通过fd使用基础文件API的fs.writeSync接口对这个文件进行编辑修改，编辑修改完成后关闭fd。

```
let writeLen: number = fs.writeSync(file.fd, 'hello, world');
console.info('write data to file succeed and size is:' + writeLen);
fs.closeSync(file);
```

获取并使用公共目录

目录环境能力接口（ohos.file.environment）提供获取公共目录路径的能力，支持三方应用在公共文件用户目录下进行文件访问操作。

约束限制

- 使用此方式，需确认设备具有以下系统能力：SystemCapability.FileManagement.File.Environment.FolderObtain，当前仅支持2in1设备。

```
if (!canIUse('SystemCapability.FileManagement.File.Environment.FolderObtain')) {
  console.error('this api is not supported on this device');
  return;
}
```

- 公共目录获取接口仅用于获取公共目录路径，不对公共目录访问权限进行校验。若需访问公共目录需申请对应的公共目录访问权限。三方应用需要访问公共目录时，需通过弹窗授权向用户申请授予 Download 目录权限、Documents 目录权限或 Desktop 目录权限，具体参考访问控制-向用户申请授权。

```
"requestPermissions" : [
  "ohos.permission.READ_WRITE_DOWNLOAD_DIRECTORY",
  "ohos.permission.READ_WRITE_DOCUMENTS_DIRECTORY",
]
```

示例

- 获取公共目录路径。

```
import { BusinessError } from '@kit.BasicServicesKit';
import { Environment } from '@kit.CoreFileKit';

function getUserDirExample() {
  try {
    const downloadPath = Environment.getUserDownloadDir();
    console.info(`success to getUserDownloadDir: ${downloadPath}`);
    const documentsPath = Environment.getUserDocumentDir();
    console.info(`success to getUserDocumentDir: ${documentsPath}`);
  } catch (error) {
    const err: BusinessError = error as BusinessError;
    console.error(`failed to get user dir, because: ${JSON.stringify(err)}`);
  }
}
```

- 以 Download 目录为例，访问 Download 目录下的文件。

```
import { BusinessError } from '@kit.BasicServicesKit';
import { Environment } from '@kit.CoreFileKit';
import { fileIo as fs } from '@kit.CoreFileKit';
import { common } from '@kit.AbilityKit';

function readUserDownloadDirExample() {
  // 检查是否具有 READ_WRITE_DOWNLOAD_DIRECTORY 权限，无权限则需要向用户申请授予权限。
  try {
    // 获取 Download 目录
    const downloadPath = Environment.getUserDownloadDir();
    console.info(`success to getUserDownloadDir: ${downloadPath}`);
    const context = getContext() as common.UIAbilityContext;
    const dirPath = context.filesDir;
    console.info(`success to get filesDir: ${dirPath}`);
    // 查看 Download 目录下的文件并拷贝到沙箱目录中
    let fileList: string[] = fs.listFileSync(downloadPath);
    fileList.forEach((file, index) => {
      console.info(`${downloadPath} ${index}: ${file}`);
      fs.copyFileSync(`${downloadPath}/${file}`, `${dirPath}/${file}`);
    });
    // 查看沙箱目录下对应的文件
    fileList = fs.listFileSync(dirPath);
    fileList.forEach((file, index) => {
      console.info(`${dirPath} ${index}: ${file}`);
    });
  } catch (error) {
    const err: BusinessError = error as BusinessError;
    console.error(`Error code: ${err.code}, message: ${err.message}`);
  }
}
```

- 以 Download 目录为例，保存文件到 Download 目录。

```
import { BusinessError } from '@kit.BasicServicesKit';
import { Environment } from '@kit.CoreFileKit';
import { fileIo as fs } from '@kit.CoreFileKit';

function writeUserDownloadDirExample() {
// 检查是否具有 READ_WRITE_DOWNLOAD_DIRECTORY 权限，无权限则需要向用户申请授予权限。
  try {
    // 获取 Download 目录
    const downloadPath = Environment.getUserDownloadDir();
    console.info(`success to getUserDownloadDir: ${downloadPath}`);
    // 保存 temp.txt 到 Download 目录下
    const file = fs.openSync(`${downloadPath}/temp.txt`, fs.OpenMode.CREATE | fs.OpenMode.READ_WRITE);
    fs.writeSync(file.fd, 'write a message');
    fs.closeSync(file);
  } catch (error) {
    const err: BusinessError = error as BusinessError;
    console.error(`Error code: ${err.code}, message: ${err.message}`);
  }
}
```

实践案例： FilesManger

ArkWeb

ArkWeb简介

使用场景

ArkWeb（方舟Web）提供了Web组件，用于在应用程序中显示Web页面内容。常见使用场景包括：

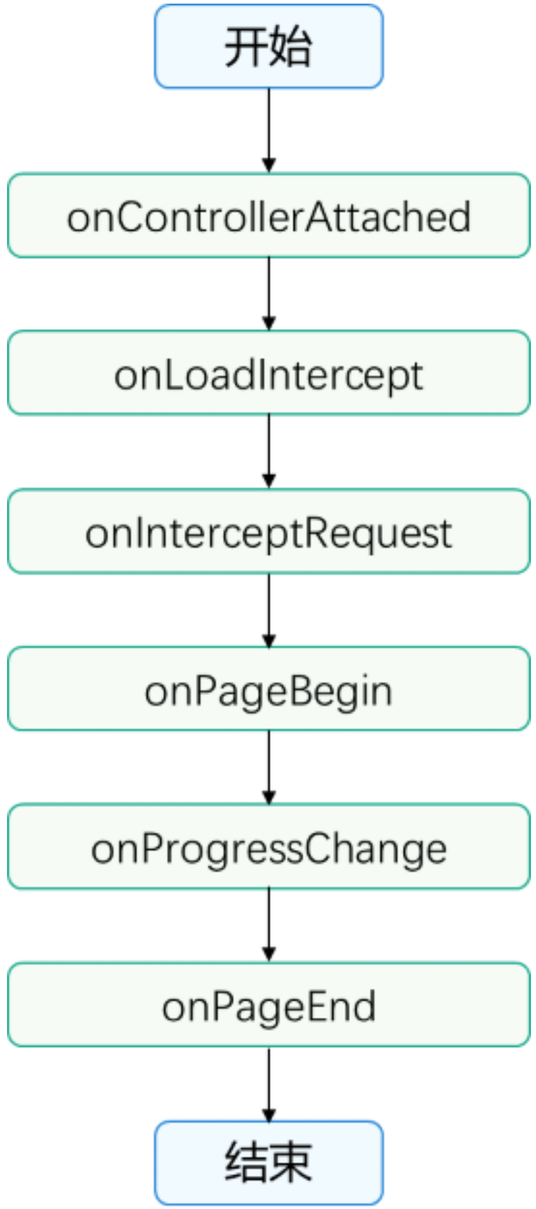
- 应用集成Web页面：应用可以在页面中使用Web组件，嵌入Web页面内容，以降低开发成本，提升开发、运营效率。
- 浏览器网页浏览场景：浏览器类应用可以使用Web组件，打开三方网页，使用无痕模式浏览Web页面，设置广告拦截等。
- 小程序：小程序类宿主应用可以使用Web组件，渲染小程序的页面。

能力范围

Web组件为开发者提供了丰富的控制Web页面能力。包括：

- Web页面加载：声明式加载Web页面和离屏加载Web页面等。
- 生命周期管理：组件生命周期状态变化，通知Web页面的加载状态变化等。
- 常用属性与事件：UserAgent管理、Cookie与存储管理、字体与深色模式管理、权限管理等。
- 与应用界面交互：自定义文本选择菜单、上下文菜单、文件上传界面等与应用界面交互能力。
- App通过JavaScriptProxy，与Web页面进行JavaScript交互。
- 安全与隐私：无痕浏览模式、广告拦截、高级安全模式等。
- 维测能力：Devtools工具调试能力，使用crashpad收集Web组件崩溃信息。
- 其他高阶能力：与原生组件同层渲染、Web组件的网络托管、Web组件的媒体播放托管、Web组件输入框拉起自定义输入法、等。

Web组件的生命周期


开始
onControllerAttached
onLoadIntercept
onInterceptRequest
onPageBegin
onProgressChange
onPageEnd
结束

函数和事件

- aboutToAppear函数：在创建自定义组件的新实例后，在执行其build函数前执行。一般建议在此设置WebDebug调试模式setWebDebuggingAccess、设置Web内核自定义协议URL的跨域请求与fetch请求的权限customizeSchemes、设置Cookie(configCookie)等。
- onControllerAttached事件：当Controller成功绑定到Web组件时触发该回调，推荐在此事件中注入JS对象registerJavaScriptProxy、设置自定义用户代理setCustomUserAgent，可以在回调中使用loadUrl，getWebId等操作网页不相关的接口。但因该回调调用时网页还未加载，因此无法在回调中使用有关操作网页的接口，例如zoomIn、zoomOut等。
- onLoadIntercept事件：当Web组件加载url之前触发该回调，用于判断是否阻止此次访问。默认允许加载。
- onOverrideUrlLoading事件：当URL将要加载到当前Web中时，让宿主应用程序有机会获得控制权，回调函数返回true将导致当前Web中止加载URL，而返回false则会导致Web继续照常加载URL。onLoadIntercept接口和onOverrideUrlLoading接口行为不一致，触发时机也不同，所以在应用场景上存在一定区别。主要是在LoadUrl和iframe（HTML标签，表示HTML内联框架元素，用于将另一个页面嵌入到当前页面中）加载时，onLoadIntercept事件会正常回调到，但onOverrideUrlLoading事件在LoadUrl加载时不会触发，在iframe加载HTTP(s)协议或about:blank时也不会触发。详细介绍请见onLoadIntercept和onOverrideUrlLoading的说明。
- onInterceptRequest事件：当Web组件加载url之前触发该回调，用于拦截url并返回响应数据。
- onPageBegin事件：网页开始加载时触发该回调，且只在主frame（表示一个HTML元素，用于展示HTML页面的HTML元素）触发。如果是iframe或者frameset（用于包含frame的HTML标签）的内容加载时则不会触发此回调。多frame页面有可能同时开始加载，即使主frame已经加载结束，子frame也有可能才开始或者继续加载中。同一页面导航（片段、历史状态等）或者在提交前失败、被取消的导航等也不会触发该回调。
- onProgressChange事件：告知开发者当前页面加载的进度。多frame页面或者子frame有可能还在继续加载而主frame可能已经加载结束，所以在onPageEnd事件后依然有可能收到该事件。
- onPageEnd事件：网页加载完成时触发该回调，且只在主frame触发。多frame页面有可能同时开始加载，即使主frame已经加载结束，子frame也有可能才开始或者继续加载中。同一页面导航（片段、历史状态等）或者在提交前失败、被取消的导航等也不会触发该回调。推荐在此回调中执行JavaScript脚本loadUrl等。需要注意的是收到该回调并不能保证Web绘制的下一帧将反映此时DOM的状态。
- onRenderExited事件：应用渲染进程异常退出时触发该回调，可以在此回调中进行系统资源的释放、数据的保存等操作。如果应用希望异常恢复，需要调用loadUrl接口重新加载页面。
- onDisAppear事件：组件卸载消失时触发此回调。该事件为通用事件，指组件从组件树上卸载时触发的事件。

使用Web组件加载页面

页面加载是Web组件的基本功能。根据页面加载数据来源可以分为三种常用场景，包括加载网络页面、加载本地页面、加载HTML格式的富文本数据。
页面加载过程中，若涉及网络资源获取，需要配置ohos.permission.INTERNET网络访问权限。

加载网络页面

开发者可以在Web组件创建时，指定默认加载的网络页面 。在默认页面加载完成后，如果开发者需要变更此Web组件显示的网络页面，可以通过调用loadUrl()接口加载指定的网页。Web组件的第一个参数变量src不能通过状态变量（例如：@State）动态更改地址，如需更改，请通过loadUrl()重新加载。
在下面的示例中，在Web组件加载完“www.example.com”页面后，开发者可通过loadUrl接口将此Web组件显示页面变更为“www.example1.com”。

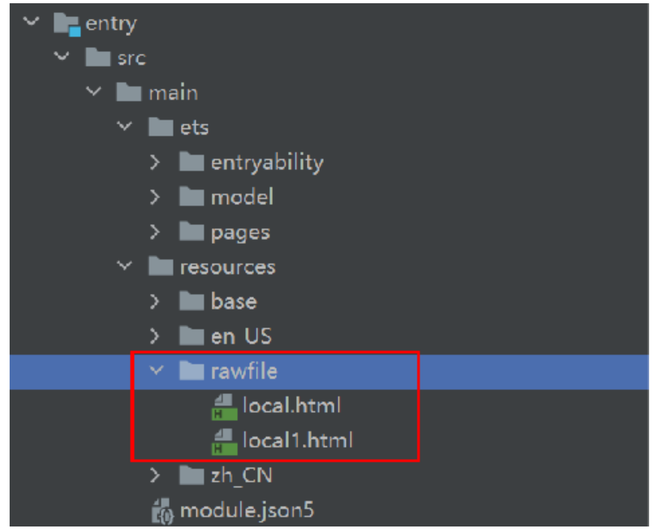
```
// xxx.ets
import { webview } from '@kit.ArkWeb';
import { BusinessError } from '@kit.BasicServicesKit';

@Entry
@Component
struct WebComponent {
  controller: webview.WebviewController = new webview.WebviewController();

  build() {
    Column() {
      Button('loadUrl')
        .onClick(() => {
          try {
            // 点击按钮时，通过loadUrl，跳转到www.example1.com
            this.controller.loadUrl('www.example1.com');
          } catch (error) {
            console.error(`ErrorCode: ${(error as BusinessError).code},  Message: ${(error as BusinessError).message}`);
          }
        })
      // 组件创建时，加载www.example.com
      Web({ src: 'www.example.com', controller: this.controller })
    }
  }
}
```

加载本地页面

加载本地页面

在下面的示例中展示加载本地页面文件的方法：
将本地页面文件放在应用的rawfile目录下，开发者可以在Web组件创建的时候指定默认加载的本地页面，并且加载完成后可通过调用loadUrl()接口变更当前Web组件的页面。

- 将资源文件放置在应用的resources/rawfile目录下。
  图1 资源文件路径



- 应用侧代码。

```
// xxx.ets
import { webview } from '@kit.ArkWeb';
import { BusinessError } from '@kit.BasicServicesKit';

@Entry
@Component
struct WebComponent {
  controller: webview.WebviewController = new webview.WebviewController();

  build() {
    Column() {
      Button('loadUrl')
        .onClick(() => {
          try {
            // 点击按钮时，通过loadUrl，跳转到local1.html
            this.controller.loadUrl($rawfile("local1.html"));
          } catch (error) {
            console.error(`ErrorCode: ${(error as BusinessError).code}, Message: ${(error as BusinessError).message}`);
        }
      })
      // 组件创建时，通过$rawfile加载本地文件local.html
      Web({ src: $rawfile("local.html"), controller: this.controller })
    }
  }
}
```

- local.html页面代码。

```
<!-- local.html -->
<!DOCTYPE html>
<html>
  <body>
    <p>Hello World</p>
  </body>
</html>
```

- local1.html页面代码。

```
<!-- local1.html -->
<!DOCTYPE html>
<html>
  <body>
    <p>This is local1 page</p>
  </body>
</html>
```

加载沙箱路径下的本地页面文件

- 通过构造的单例对象GlobalContext获取沙箱路径。

```
// GlobalContext.ets
export class GlobalContext {
  private constructor() {}
  private static instance: GlobalContext;
  private _objects = new Map<string, Object>();

  public static getContext(): GlobalContext {
    if (!GlobalContext.instance) {
      GlobalContext.instance = new GlobalContext();
    }
    return GlobalContext.instance;
  }

  getObject(value: string): Object | undefined {
    return this._objects.get(value);
  }

  setObject(key: string, objectClass: Object): void {
    this._objects.set(key, objectClass);
  }
}
```

```
// xxx.ets
import { webview } from '@kit.ArkWeb';
import { GlobalContext } from '../GlobalContext';

let url = 'file://' + GlobalContext.getContext().getObject("filesDir") + '/index.html';

@Entry
@Component
struct WebComponent {
  controller: webview.WebviewController = new webview.WebviewController();

  build() {
    Column() {
      // 加载沙箱路径文件。
      Web({ src: url, controller: this.controller })
    }
  }
}
```

- 修改EntryAbility.ets。
  以filesDir为例，获取沙箱路径。若想获取其他路径，请参考应用文件路径。

```
// xxx.ets
import { AbilityConstant, UIAbility, Want } from '@kit.AbilityKit';
import { webview } from '@kit.ArkWeb';
import { GlobalContext } from '../GlobalContext';

export default class EntryAbility extends UIAbility {
  onCreate(want: Want, launchParam: AbilityConstant.LaunchParam) {
    // 通过在GlobalContext对象上绑定filesDir，可以实现UIAbility组件与UI之间的数据同步。
    GlobalContext.getContext().setObject("filesDir", this.context.filesDir);
    console.log("Sandbox path is " + GlobalContext.getContext().getObject("filesDir"));
  }
}
```

- 加载的html文件。

```
<!-- index.html -->
<!DOCTYPE html>
<html>
    <body>
        <p>Hello World</p>
    </body>
</html>
```

加载HTML格式的文本数据

Web组件可以通过loadData()接口实现加载HTML格式的文本数据。当开发者不需要加载整个页面，只需要显示一些页面片段时，可通过此功能来快速加载页面。

```
// xxx.ets
import { webview } from '@kit.ArkWeb';
import { BusinessError } from '@kit.BasicServicesKit';

@Entry
@Component
struct WebComponent {
  controller: webview.WebviewController = new webview.WebviewController();

  build() {
    Column() {
      Button('loadData')
        .onClick(() => {
          try {
            // 点击按钮时，通过loadData，加载HTML格式的文本数据
            this.controller.loadData(
              "<html><body bgcolor=\"white\">Source:<pre>source</pre></body></html>",
              "text/html",
              "UTF-8"
            );
          } catch (error) {
            console.error(`ErrorCode: ${(error as BusinessError).code},  Message: ${(error as BusinessError).message}`);
          }
        })
      // 组件创建时，加载www.example.com
      Web({ src: 'www.example.com', controller: this.controller })
    }
  }
}
```

动态创建Web组件

支持命令式创建Web组件，这种方式创建的组件不会立即挂载到组件树，即不会对用户呈现（组件状态为Hidden和InActive），开发者可以在后续使用中按需动态挂载。后台启动的Web实例不建议超过200个。

```
// 载体Ability
// EntryAbility.ets
import { createNWeb } from "../pages/common"
onWindowStageCreate(windowStage: window.WindowStage): void {
  windowStage.loadContent('pages/Index', (err, data) => {
    // 创建Web动态组件（需传入UIContext），loadContent之后的任意时机均可创建
    createNWeb("https://www.example.com",
windowStage.getMainWindowSync().getUIContext());
    if (err.code) {
      return;
    }
  });
}
```

```
// 创建NodeController
// common.ets
import { UIContext, NodeController, BuilderNode, Size, FrameNode } from
'@kit.ArkUI';
import { webview } from '@kit.ArkWeb';

// @Builder中为动态组件的具体组件内容
// Data为入参封装类
class Data{
  url: string = "https://www.example.com";
  controller: WebviewController = new webview.WebviewController();
}

@Builder
function WebBuilder(data:Data) {
  Column() {
    Web({ src: data.url, controller: data.controller })
      .width("100%")
      .height("100%")
  }
}

let wrap = wrapBuilder<Data[]>(WebBuilder);

// 用于控制和反馈对应的NodeContainer上的节点的行为，需要与NodeContainer一起使用
export class myNodeController extends NodeController {
  private rootnode: BuilderNode<Data[]> | null = null;
  // 必须要重写的方法，用于构建节点数、返回节点挂载在对应NodeContainer中
  // 在对应NodeContainer创建的时候调用、或者通过rebuild方法调用刷新
  makeNode(uiContext: UIContext): FrameNode | null {
    console.log(" uicontext is undefined : "+ (uiContext === undefined));
    if (this.rootnode != null) {
      // 返回FrameNode节点
      return this.rootnode.getFrameNode();
    }
    // 返回null控制动态组件脱离绑定节点
    return null;
  }
  // 当布局大小发生变化时进行回调
  aboutToResize(size: Size) {
    console.log("aboutToResize width : " + size.width  +  " height : " + size.height );
  }

  // 当controller对应的NodeContainer在Appear的时候进行回调
  aboutToAppear() {
    console.log("aboutToAppear");
  }

  // 当controller对应的NodeContainer在Disappear的时候进行回调
  aboutToDisappear() {
    console.log("aboutToDisappear");
  }

  // 此函数为自定义函数，可作为初始化函数使用
  // 通过UIContext初始化BuilderNode，再通过BuilderNode中的build接口初始化@Builder中的内容
  initWeb(url:string, uiContext:UIContext, control:WebviewController) {
    if(this.rootnode != null)
    {
      return;
    }
    // 创建节点，需要uiContext
    this.rootnode = new BuilderNode(uiContext);
    // 创建动态Web组件
    this.rootnode.build(wrap, { url:url, controller:control });
  }
}
// 创建Map保存所需要的NodeController
let NodeMap:Map<string, myNodeController | undefined> = new Map();
// 创建Map保存所需要的WebViewController
let controllerMap:Map<string, WebviewController | undefined> = new Map();

// 初始化需要UIContext 需在Ability获取
export const createNWeb = (url: string, uiContext: UIContext) => {
  // 创建NodeController
  let baseNode = new myNodeController();
  let controller = new webview.WebviewController() ;
  // 初始化自定义Web组件
  baseNode.initWeb(url, uiContext, controller);
  controllerMap.set(url, controller)
  NodeMap.set(url, baseNode);
}
// 自定义获取NodeController接口
export const getNWeb = (url : string) : myNodeController | undefined => {
  return NodeMap.get(url);
}
```

```
// 使用NodeController的Page页
// Index.ets
import { getNWeb } from "./common"
@Entry
@Component
struct Index {
  build() {
    Row() {
      Column() {
        // NodeContainer用于与NodeController节点绑定，rebuild会触发makeNode
        // Page页通过NodeContainer接口绑定NodeController，实现动态组件页面显示
        NodeContainer(getNWeb("https://www.example.com"))
          .height("90%")
          .width("100%")
      }
      .width('100%')
    }
    .height('100%')
  }
}
```

历史记录导航

在前端页面点击网页中的链接时，Web组件默认会自动打开并加载目标网址。当前端页面替换为新的加载链接时，会自动记录已经访问的网页地址。可以通过forward()和backward()接口向前/向后浏览上一个/下一个历史记录。
页面加载过程中，若涉及网络资源获取，需要配置ohos.permission.INTERNET网络访问权限。
在下面的示例中，点击应用的按钮来触发前端页面的后退操作。

```
// xxx.ets
import { webview } from '@kit.ArkWeb';

@Entry
@Component
struct WebComponent {
  webviewController: webview.WebviewController = new webview.WebviewController();

  build() {
    Column() {
      Button('loadData')
        .onClick(() => {
          if (this.webviewController.accessBackward()) {
            this.webviewController.backward();
          }
        })
      Web({ src: 'https://www.example.com/cn/', controller: this.webviewController })
    }
  }
}
```

如果存在历史记录，accessBackward()接口会返回true。同样，您可以使用accessForward()接口检查是否存在前进的历史记录。如果您不执行检查，那么当用户浏览到历史记录的末尾时，调用forward()和backward()接口时将不执行任何操作。

页面跳转

当点击网页中的链接需要跳转到应用内其他页面时，可以通过使用Web组件的onLoadIntercept()接口来实现。在下面的示例中，应用首页Index.ets加载前端页面route.html，在前端route.html页面点击超链接，可跳转到应用的ProfilePage.ets页面。

- 应用首页Index.ets页面代码。

```
// index.ets
import { webview } from '@kit.ArkWeb';
import { router } from '@kit.ArkUI';

@Entry
@Component
struct WebComponent {
  webviewController: webview.WebviewController = new webview.WebviewController();

  build() {
    Column() {
      // 资源文件route.html存放路径src/main/resources/rawfile
      Web({ src: $rawfile('route.html'), controller: this.webviewController })
        .onLoadIntercept((event) => {
          if (event) {
            let url: string = event.data.getRequestUrl();
            if (url.indexOf('native://') === 0) {
              // 跳转其他界面
              router.pushUrl({ url: url.substring(9) });
              return true;
            }
          }
          return false;
        })
    }
  }
}
```

-
- route.html前端页面代码。

```
<!-- route.html -->
<!DOCTYPE html>
<html>
<body>
  <div>
      <a href="native://pages/ProfilePage">个人中心</a>
  </div>
</body>
</html>
```

-
- 跳转页面ProfilePage.ets代码。

```
@Entry
@Component
struct ProfilePage {
  @State message: string = 'Hello World';

  build() {
    Column() {
      Text(this.message)
        .fontSize(20)
    }
  }
}
```

跨应用跳转

Web组件可以实现点击前端页面超链接跳转到其他应用。
在下面的示例中，点击call.html前端页面中的超链接，跳转到电话应用的拨号界面。

- 应用侧代码。

```
// xxx.ets
import { webview } from '@kit.ArkWeb';
import { call } from '@kit.TelephonyKit';

@Entry
@Component
struct WebComponent {
  webviewController: webview.WebviewController = new webview.WebviewController();

  build() {
    Column() {
      Web({ src: $rawfile('call.html'), controller: this.webviewController })
        .onLoadIntercept((event) => {
          if (event) {
            let url: string = event.data.getRequestUrl();
            // 判断链接是否为拨号链接
            if (url.indexOf('tel://') === 0) {
              // 跳转拨号界面
              call.makeCall(url.substring(6), (err) => {
                if (!err) {
                  console.info('make call succeeded.');
                } else {
                  console.info('make call fail, err is:' + JSON.stringify(err));
                }
              });
              return true;
            }
          }
          return false;
        })
    }
  }
}
```

- 前端页面call.html代码。

```
<!-- call.html -->
<!DOCTYPE html>
<html>
<body>
  <div>
    <a href="tel://xxx xxxx xxx">拨打电话</a>
  </div>
</body>
</html>
```

实践案例：
ArkWebPageAdaptation