# 进程与线程

# Outline

- ArkTS语言并发
- Stage模型
  - 应用间跳转
  - 进程模型和线程模型
- Basics Service Kit的应用事件（进程间通信、线程间通信）
- Background Tasks Kit
- IPC Kit

# ArkTS语言并发

# 异步并发概述 (Promise和async/await)

- Promise和async/await提供异步并发能力，是标准的JS异步语法。异步代码会被挂起并在之后继续执行，同一时间只有一段代码执行，适用于单次I/O任务的场景开发，例如一次网络请求、一次文件读写等操作。无需另外启动线程执行。

- 异步语法是一种编程语言的特性，允许程序在执行某些操作时不必等待其完成，而是可以继续执行其他操作。

# Promise

Promise是一种用于处理异步操作的对象，可以将异步操作转换为类似于同步操作的风格，以方便代码编写和维护。Promise提供了一个状态机制来管理异步操作的不同阶段，并提供了一些方法来注册回调函数以处理异步操作的成功或失败的结果。
Promise有三种状态：pending（进行中）、fulfilled（已完成）和rejected（已拒绝）。Promise对象创建后处于pending状态，并在异步操作完成后转换为fulfilled或rejected状态。
最基本的用法是通过构造函数实例化一个Promise对象，同时传入一个带有两个参数的函数，通常称为executor函数。executor函数接收两个参数：resolve和reject，分别表示异步操作成功和失败时的回调函数。例如，以下代码创建了一个Promise对象并模拟了一个异步操作：

```
const promise: Promise<number> = new Promise((resolve: Function, reject: Function) => {
setTimeout(() => {
  const randomNumber: number = Math.random();
  if (randomNumber > 0.5) {
    resolve(randomNumber);
  } else {
    reject(new Error('Random number is too small'));
  }
}, 1000);
})
```

上述代码中，setTimeout函数模拟了一个异步操作，并在1秒钟后随机生成一个数字。如果随机数大于0.5，则执行resolve回调函数并将随机数作为参数传递；否则执行reject回调函数并传递一个错误对象作为参数。

# Promise

Promise对象创建后，可以使用then方法和catch方法指定fulfilled状态和rejected状态的回调函数。then方法可接受两个参数，一个处理fulfilled状态的函数，另一个处理rejected状态的函数。只传一个参数则表示当Promise对象状态变为fulfilled时，then方法会自动调用这个回调函数，并将Promise对象的结果作为参数传递给它。使用catch方法注册一个回调函数，用于处理“失败”的结果，即捕获Promise的状态改变为rejected状态或操作失败抛出的异常。例如：

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

promise.then((result: number) => {
 console.info(`Random number is ${result}`);
}).catch((error: BusinessError) => {
 console.error(error.message);
});
```

上述代码中，then方法的回调函数接收Promise对象的成功结果作为参数，并将其输出到控制台上。如果Promise对象进入rejected状态，则catch方法的回调函数接收错误对象作为参数，并将其输出到控制台上。

# async/await

async/await是一种用于处理异步操作的Promise语法糖，使得编写异步代码变得更加简单和易读。通过使用async关键字声明一个函数为异步函数，并使用await关键字等待Promise的解析（完成或拒绝），以同步的方式编写异步操作的代码。
async函数是一个返回Promise对象的函数，用于表示一个异步操作。在async函数内部，可以使用await关键字等待一个Promise对象的解析，并返回其解析值。如果一个async函数抛出异常，那么该函数返回的Promise对象将被拒绝，并且异常信息会被传递给Promise对象的onRejected()方法。
下面是一个使用async/await的例子，其中模拟了一个以同步方式执行异步操作的场景，该操作会在3秒钟后返回一个字符串。

```
async function myAsyncFunction(): Promise<string> {
  const result: string = await new Promise((resolve: Function) => {
    setTimeout(() => {
      resolve('Hello, world!');
    }, 3000);
  });
  console.info(result); // 输出： Hello, world!
  return result
}

@Entry
@Component
struct Index {
  @State message: string = 'Hello World';
  build() {
    Row() {
      Column() {
        Text(this.message)
          .fontSize(50)
          .fontWeight(FontWeight.Bold)
          .onClick(async () => {
            let res = await myAsyncFunction();
            console.info("res is: " + res);
          })
      }
      .width('100%')
    }
    .height('100%')
  }
}
```

在上述示例代码中，使用了await关键字来等待Promise对象的解析，并将其解析值存储在result变量中。

# async/await

需要注意的是，由于要等待异步操作完成，因此需要将整个操作包在async函数中。除了在async函数中使用await外，还可以使用try/catch块来捕获异步操作中的异常。

```
async function myAsyncFunction(): Promise<void> {
  try {
    const result: string = await new Promise((resolve: Function) => {
      resolve('Hello, world!');
    });
  } catch (e) {
    console.error(`Get exception: ${e}`);
  }
}

myAsyncFunction();
```

# 单次I/O任务开发指导

- 实现单次I/O任务逻辑。

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

async function write(data: string, file: fileIo.File): Promise<void> {
  fileIo.write(file.fd, data).then((writeLen: number) => {
    console.info('write data length is: ' + writeLen)
  }).catch((err: BusinessError) => {
    console.error(`Failed to write data. Code is ${err.code}, message is ${err.message}`);
  })
}
```

# 单次I/O任务开发指导

- 采用异步能力调用单次I/O任务。示例中的filePath的获取方式请参见获取应用文件路径。

```
async function testFunc(): Promise<boolean>  {
  let context = getContext() as common.UIAbilityContext;
  let filePath: string = context.filesDir + "/test.txt"; // 应用文件路径
  let file: fileIo.File = await fileIo.open(filePath, fileIo.OpenMode.READ_WRITE | fileIo.OpenMode.CREATE);
  write('Hello World!', file).then(() => {
    console.info('Succeeded in writing data.');
    fileIo.close(file);
  }).catch((err: BusinessError) => {
    console.error(`Failed to write data. Code is ${err.code}, message is ${err.message}`);
    fileIo.close(file);
  })

  let result = await fileIo.access(filePath, fileIo.AccessModeType.EXIST);
  if (!result) {
    return false;
  }
  return true;
}

@Entry
@Component
struct Index {
  @State message: string = 'Hello World';
  build() {
    Row() {
      Column() {
        Text(this.message)
          .fontSize(50)
          .fontWeight(FontWeight.Bold)
          .onClick(async () => {
            let res = await testFunc();
            console.info("res is: " + res);
          })
      }
      .width('100%')
    }
    .height('100%')
  }
}
```

# 实践案例：24-NewsDataArkTS

# 步骤

- 1. Console控制台：
    - 进入HttpServerOfNews目录
    - npm install
    - Node app.js
- 2. 修改entry模块 CommonConstant.ets文件中Server IP
    - http://10.0.2.2:9588  虚拟机专用IP
- 3. 运行App

# 多线程并发模型

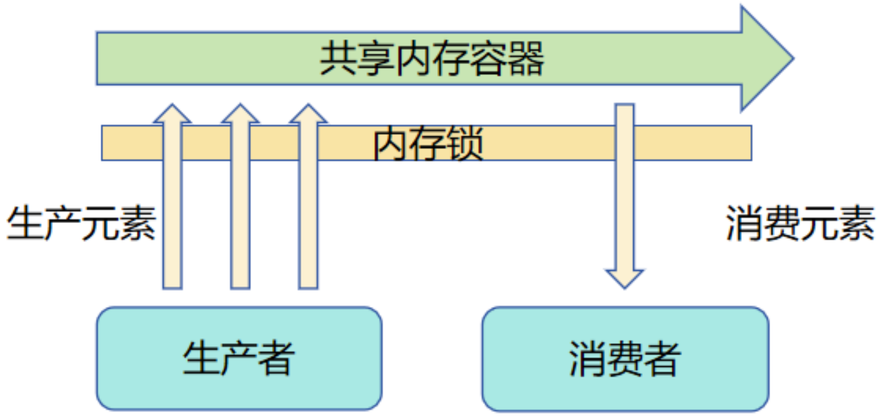
- 并发模型是用来实现不同应用场景中并发任务的编程模型，常见的并发模型分为基于内存共享的并发模型和基于消息通信的并发模型。

- Actor并发模型作为基于消息通信并发模型的典型代表，不需要开发者去面对线程锁带来的一系列复杂偶发的问题，同时并发度也相对较高，因此得到了广泛的支持和使用。

- 当前ArkTS提供了TaskPool和Worker两种并发能力，TaskPool和Worker都基于Actor并发模型实现。

# 内存共享型

为了避免不同生产者或消费者同时访问一块共享内存的容器时产生的脏读，脏写现象，同一时间只能有一个生产者或消费者访问该容器，也就是不同生产者和消费者争夺使用容器的锁。当一个角色获取锁之后其他角色需要等待该角色释放锁之后才能重新尝试获取锁以访问该容器。



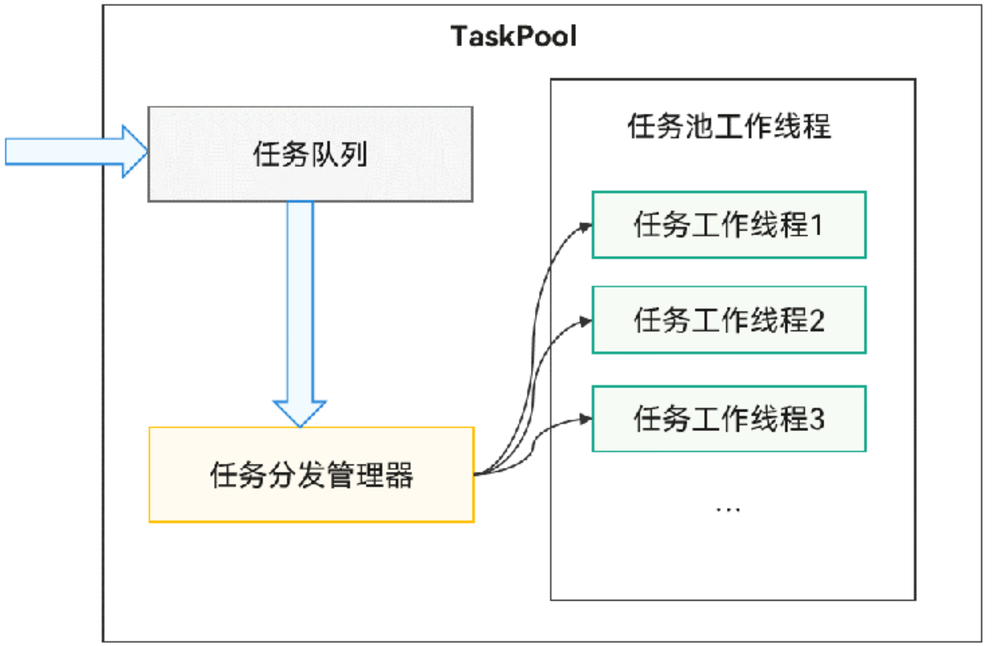
# Actor模型

Actor模型不同角色之间并不共享内存，生产者线程和UI线程都有自己独占的内存。生产者生产出结果后通过序列化通信将结果发送给UI线程，UI线程消费结果后再发送新的生产任务给生产者线程。

# TaskPool机制

TaskPool支持开发者在主线程封装任务抛给任务队列，系统选择合适的工作线程，进行任务的分发及执行，再将结果返回给主线程。接口直观易用，支持任务的执行、取消，以及指定优先级的能力，同时通过系统统一线程管理，结合动态调度及负载均衡算法，可以节约系统资源。

系统默认会启动一个任务工作线程，当任务较多时会扩容，工作线程数量上限跟当前设备的物理核数相关，具体数量内部管理，保证最优的调度及执行效率，长时间没有任务分发时会缩容，减少工作线程数量。



# 示例代码

```
// 正例
@Concurrent
async function asyncFunc(val1:number, val2:number): Promise<number> {
  let ret: number = await new Promise((resolve, reject) => {
    let value = val1 + val2;
    resolve(value);
  });
  return ret; // 支持。直接返回Promise的结果。
}

function taskpoolExecute() {
  taskpool.execute(asyncFunc, 10, 20).then((result: Object) => {
    console.info("taskPoolTest task result: " + result);
  }).catch((err: string) => {
    console.error("taskPoolTest test occur error: " + err);
  });
}
taskpoolExecute()
```

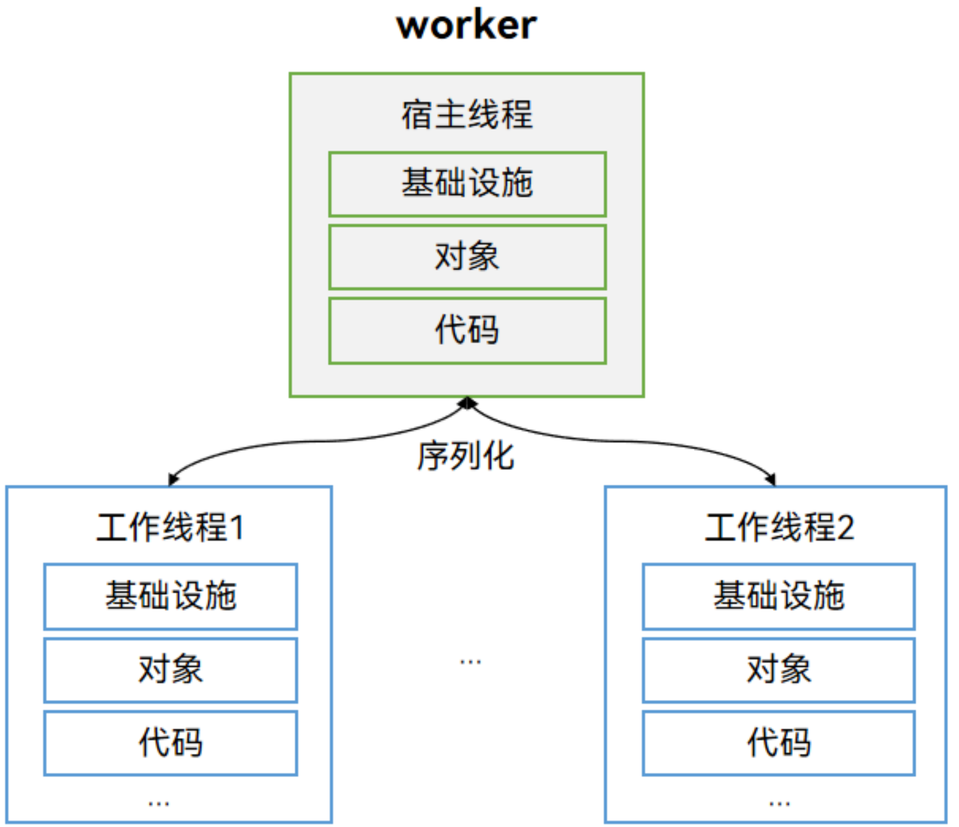
```
// 反例1:
@Concurrent
async function asyncFunc(val1:number, val2:number): Promise<number> {
  let ret: number = await new Promise((resolve, reject) => {
    let value = val1 + val2;
    resolve(value);
  });
  return Promise.resolve(ret); // 不支持。Promise.resolve仍是Promise，其状态是pending，无法作为返回值使用。
}
```

# 示例代码

```
// 反例2:
@Concurrent
async function asyncFunc(val1:number, val2:number): Promise<number> {
  // 不支持。其状态是pending，无法作为返回值使用。
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      let value = val1 + val2;
      resolve(value);
    }, 2000);
  });
}

function taskpoolExecute() {
  taskpool.execute(asyncFunc, 10, 20).then((result: Object) => {
    console.info("taskPoolTest task result: " + result);
  }).catch((err: string) => {
    console.error("taskPoolTest test occur error: " + err);
  });
}
taskpoolExecute()
```

# Worker机制

创建Worker的线程称为宿主线程（不一定是主线程，工作线程也支持创建Worker子线程），Worker自身的线程称为Worker子线程（或Actor线程、工作线程）。每个Worker子线程与宿主线程拥有独立的实例，包含基础设施、对象、代码段等，因此每个Worker启动存在一定的内存开销，需要限制Worker的子线程数量。Worker子线程和宿主线程之间的通信是基于消息传递的，Worker通过序列化机制与宿主线程之间相互通信，完成命令及数据交互。



# 创建Worker的注意事项

Worker线程文件需要放在"{moduleName}/src/main/ets/"目录层级之下，否则不会被打包到应用中。有手动和自动两种创建Worker线程目录及文件的方式。

- 手动创建：开发者手动创建相关目录及文件，此时需要配置build-profile.json5的相关字段信息，Worker线程文件才能确保被打包到应用中。
  Stage模型：

```
"buildOption": {
  "sourceOption": {
    "workers": [
      "./src/main/ets/workers/worker.ets"
    ]
  }
}
```

当使用Worker模块具体功能时，均需先构造Worker实例对象，其构造函数与API版本相关，且构造函数需要传入Worker线程文件的路径（scriptURL）。

```
// 导入模块
import { worker } from '@kit.ArkTS';

// API 9及之后版本使用：
const worker1: worker.ThreadWorker = new worker.ThreadWorker('entry/ets/workers/MyWorker.ets');
// API 8及之前版本使用：
const worker2: worker.Worker = new worker.Worker('entry/ets/workers/MyWorker.ets');
```

# 使用场景对比

- TaskPool和Worker均支持多线程并发能力。由于TaskPool的工作线程会绑定系统的调度优先级，并且支持负载均衡（自动扩缩容），而Worker需要开发者自行创建，存在创建耗时以及不支持设置调度优先级，故在性能方面使用TaskPool会优于Worker，因此大多数场景推荐使用TaskPool。

- TaskPool偏向独立任务维度，该任务在线程中执行，无需关注线程的生命周期，超长任务（大于3分钟且非长时任务）会被系统自动回收；而Worker偏向线程的维度，支持长时间占据线程执行，需要主动管理线程生命周期。

- 常见的一些开发场景及适用具体说明如下：

  - 运行时间超过3分钟（不包含Promise和async/await异步调用的耗时，例如网络下载、文件读写等I/O任务的耗时）的任务。例如后台进行1小时的预测算法训练等CPU密集型任务，需要使用Worker。

  - 有关联的一系列同步任务。例如在一些需要创建、使用句柄的场景中，句柄创建每次都是不同的，该句柄需永久保存，保证使用该句柄进行操作，需要使用Worker。

  - 需要设置优先级的任务。例如图库直方图绘制场景，后台计算的直方图数据会用于前台界面的显示，影响用户体验，需要高优先级处理，需要使用TaskPool。

  - 需要频繁取消的任务。例如图库大图浏览场景，为提升体验，会同时缓存当前图片左右侧各2张图片，往一侧滑动跳到下一张图片时，要取消另一侧的一个缓存任务，需要使用TaskPool。

- 大量或者调度点较分散的任务。例如大型应用的多个模块包含多个耗时任务，不方便使用Worker去做负载管理，推荐采用TaskPool。

# CPU密集型任务开发指导 (TaskPool)

CPU密集型任务是指需要占用系统资源处理大量计算能力的任务，需要长时间运行，这段时间会阻塞线程其它事件的处理，不适宜放在主线程进行。例如图像处理、视频编码、数据分析等。
基于多线程并发机制处理CPU密集型任务可以提高CPU利用率，提升应用程序响应速度。
当任务不需要长时间（3分钟）占据后台线程，而是一个个独立的任务时，推荐使用TaskPool，反之推荐使用Worker。
接下来将以图像直方图处理以及后台长时间的模型预测任务分别进行举例。

**使用TaskPool进行图像直方图处理**

- 实现图像处理的业务逻辑。
- 数据分段，通过任务组发起关联任务调度。
  创建TaskGroup并通过addTask()添加对应的任务，通过execute()执行任务组，并指定为高优先级，在当前任务组所有任务结束后，会将直方图处理结果同时返回。
- 结果数组汇总处理。

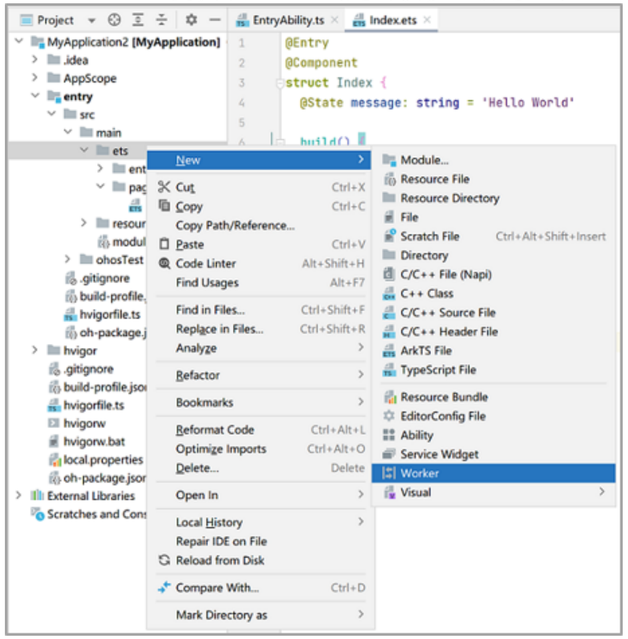
```
import { taskpool } from '@kit.ArkTS';

@Concurrent
function imageProcessing(dataSlice: ArrayBuffer): ArrayBuffer {
  // 步骤1：具体的图像处理操作及其他耗时操作
  return dataSlice;
}

function histogramStatistic(pixelBuffer: ArrayBuffer): void {
  // 步骤2：分成三段并发调度
  let number: number = pixelBuffer.byteLength / 3;
  let buffer1: ArrayBuffer = pixelBuffer.slice(0, number);
  let buffer2: ArrayBuffer = pixelBuffer.slice(number, number * 2);
  let buffer3: ArrayBuffer = pixelBuffer.slice(number * 2);

  let group: taskpool.TaskGroup = new taskpool.TaskGroup();
  group.addTask(imageProcessing, buffer1);
  group.addTask(imageProcessing, buffer2);
  group.addTask(imageProcessing, buffer3);

  taskpool.execute(group, taskpool.Priority.HIGH).then((ret: Object) => {
    // 步骤3：结果数组汇总处理
  })
}

@Entry
@Component
struct Index {
  @State message: string = 'Hello World'

  build() {
    Row() {
      Column() {
        Text(this.message)
          .fontSize(50)
          .fontWeight(FontWeight.Bold)
          .onClick(() => {
            let buffer: ArrayBuffer = new ArrayBuffer(24);
            histogramStatistic(buffer);
          })
      }
      .width('100%')
    }
    .height('100%')
  }
}
```

# CPU密集型任务开发指导 (Worker)

**使用Worker进行长时间数据分析**

本文通过某地区提供的房价数据训练一个简易的房价预测模型，该模型支持通过输入房屋面积和房间数量去预测该区域的房价，模型需要长时间运行，房价预测需要使用前面的模型运行结果，因此需要使用Worker。

- DevEco Studio提供了Worker创建的模板，新建一个Worker线程，例如命名为“MyWorker”。




- 在主线程中通过调用ThreadWorker的constructor()方法创建Worker对象，当前线程为宿主线程。

```
// Index.ets
import { worker } from '@kit.ArkTS';

const workerInstance: worker.ThreadWorker = new worker.ThreadWorker('entry/ets/workers/MyWorker.ts');
```

- 在宿主线程中通过调用onmessage()方法接收Worker线程发送过来的消息，并通过调用postMessage()方法向Worker线程发送消息。
  例如向Worker线程发送训练和预测的消息，同时接收Worker线程发送回来的消息。

```
// Index.ets
let done = false;

// 接收Worker子线程的结果
workerInstance.onmessage = (() => {
  console.info('MyWorker.ts onmessage');
  if (!done) {
    workerInstance.postMessage({ 'type': 1, 'value': 0 });
    done = true;
  }
})

workerInstance.onerror = (() => {
  // 接收Worker子线程的错误信息
})

// 向Worker子线程发送训练消息
workerInstance.postMessage({ 'type': 0 });
```

# CPU密集型任务开发指导 (Worker)

- 在MyWorker.ts文件中绑定Worker对象，当前线程为Worker线程。

```
// MyWorker.ts
import { worker, ThreadWorkerGlobalScope, MessageEvents, ErrorEvent } from '@kit.ArkTS';

let workerPort: ThreadWorkerGlobalScope = worker.workerPort;
```

- 在Worker线程中通过调用onmessage()方法接收宿主线程发送的消息内容，并通过调用postMessage()方法向宿主线程发送消息。
例如在Worker线程中定义预测模型及其训练过程，同时与主线程进行信息交互。

```
// MyWorker.ts
// 定义训练模型及结果
let result: Array<number>;
// 定义预测函数
function predict(x: number): number {
 return result[x];
}
// 定义优化器训练过程
function optimize(): void {
 result = [0];
}
// Worker线程的onmessage逻辑
workerPort.onmessage = (e: MessageEvents): void => {
 // 根据传输的数据的type选择进行操作
 switch (e.data.type as number) {
  case 0:
  // 进行训练
   optimize();
  // 训练之后发送主线程训练成功的消息
   workerPort.postMessage({ type: 'message', value: 'train success.' });
   break;
  case 1:
  // 执行预测
   const output: number = predict(e.data.value as number);
  // 发送主线程预测的结果
   workerPort.postMessage({ type: 'predict', value: output });
   break;
  default:
   workerPort.postMessage({ type: 'message', value: 'send message is invalid' });
   break;
 }
}
```

- 在Worker线程中完成任务之后，执行Worker线程销毁操作。销毁线程的方式主要有两种：根据需要可以在宿主线程中对Worker线程进行销毁；也可以在Worker线程中主动销毁Worker线程。
在宿主线程中通过调用onexit()方法定义Worker线程销毁后的处理逻辑。

```
// Worker线程销毁后，执行onexit回调方法
workerInstance.onexit = (): void => {
 console.info("main thread terminate");
}
```

方式一：在宿主线程中通过调用terminate()方法销毁Worker线程，并终止Worker接收消息。

```
// 销毁Worker线程
workerInstance.terminate();
```

方式二：在Worker线程中通过调用close()方法主动销毁Worker线程，并终止Worker接收消息。

```
// 销毁线程
workerPort.close();
```

# I/O密集型任务开发指导 (TaskPool)

使用异步并发可以解决单次I/O任务阻塞的问题，但是如果遇到I/O密集型任务，同样会阻塞线程中其它任务的执行，这时需要使用多线程并发能力来进行解决。I/O密集型任务的性能重点通常不在于CPU的处理能力，而在于I/O操作的速度和效率。这种任务通常需要频繁地进行磁盘读写、网络通信等操作。此处以频繁读写系统文件来模拟I/O密集型并发任务的处理。

- 定义并发函数，内部密集调用I/O能力。

```
// write.ets
import { fileIo } from '@kit.CoreFileKit'

// 定义并发函数，内部密集调用I/O能力
// 写入文件的实现
export async function write(data: string, filePath: string): Promise<void> {
  let file: fileIo.File = await fileIo.open(filePath, fileIo.OpenMode.READ_WRITE | fileIo.OpenMode.CREATE);
  await fileIo.write(file.fd, data);
  fileIo.close(file);
}
```

```
// Index.ets
import { write } from './write'
import { BusinessError } from '@kit.BasicServicesKit';
import { taskpool } from '@kit.ArkTS';
import { common } from '@kit.AbilityKit';

@Concurrent
async function concurrentTest(context: common.UIAbilityContext): Promise<boolean> {
  let filePath1: string = context.filesDir + "/path1.txt"; // 应用文件路径
  let filePath2: string = context.filesDir + "/path2.txt";
  // 循环写文件操作
  let fileList: Array<string> = [];
  fileList.push(filePath1);
  fileList.push(filePath2)
  for (let i: number = 0; i < fileList.length; i++) {
    write('Hello World!', fileList[i]).then(() => {
      console.info(`Succeeded in writing the file. FileList: ${fileList[i]}`);
    }).catch((err: BusinessError) => {
      console.error(`Failed to write the file. Code is ${err.code}, message is ${err.message}`)
      return false;
    })
  }
  return true;
}
```

# I/O密集型任务开发指导 (TaskPool)

- 使用TaskPool执行包含密集I/O的并发函数：通过调用execute()方法执行任务，并在回调中进行调度结果处理。示例中获取filePath1和filePath2的方式请参见获取应用文件路径，在TaskPool中使用context需先在并发函数外部准备好，通过入参传递给并发函数才可使用。

```
// Index.ets
@Entry
@Component
struct Index {
  @State message: string = 'Hello World';
  build() {
    Row() {
      Column() {
        Text(this.message)
          .fontSize(50)
          .fontWeight(FontWeight.Bold)
          .onClick(() => {
            let context = getContext() as common.UIAbilityContext;

            // 使用TaskPool执行包含密集I/O的并发函数
            // 数组较大时，I/O密集型任务任务分发也会抢占主线程，需要使用多线程能力
            taskpool.execute(concurrentTest, context).then(() => {
              // 调度结果处理
              console.info("taskpool: execute success")
            })
          })
      }
      .width('100%')
    }
    .height('100%')
  }
}
```

# 实践案例：operaterdbintaskpool

# Stage模型的应用间跳转

# 应用间跳转

- 应用跳转是指从一个应用跳转至另外一个应用，传递相应的数据、执行特定的功能。通过应用跳转可以满足用户更为真实丰富的场景诉求、提升交互体验的便捷性和流畅性。

- 应用跳转的两种类型

  - 拉起指定应用：拉起方应用明确指定跳转的目标应用，来实现应用跳转。指向性跳转可以分为指定应用链接、指定Ability两种方式。

    - 指定应用链接（推荐）：通过openLink或startAbility接口来指定应用链接，拉起目标应用页面。

    - 指定Ability（不推荐）：通过startAbility接口指定具体的Ability，显式拉起目标应用页面。
      说明
      API 11及以前版本，可以使用显式want拉起其他应用。从API 12开始，已不再支持三方应用使用该方式拉起其他应用。

  - 拉起指定类型的应用：拉起方应用通过指定应用类型，拉起垂类应用面板。该面板将展示目标方接入的垂域应用，由用户选择打开指定应用。

# 应用链接

- 应用链接是指可以将用户引导至应用内特定位置或相关网页的URL，详见URL的基本格式。

- 应用链接跳转的运作机制
  - 目标应用在配置文件中注册自己的URL，并对外提供URL。
  - 拉起方应用在跳转接口中传入目标应用的URL等信息。
  - 系统接收到URL等相关信息，会寻找对应匹配项，并跳转至目标应用。

- 典型场景：拉起系统应用
  - 拉起系统应用是应用间跳转的一种典型场景。系统提供了一些能力和接口，在确保访问安全的前提下，可以让开发者快捷地实现系统应用跳转，详见拉起系统应用。

# Deep Linking与App Linking的对比

**Deep Linking**：是一种通过链接跳转至应用特定页面的技术，其特点是支持开发者定义任意形式的scheme。
**App Linking**：是一种特定类型的Deep Linking，其限定了scheme必须为https，同时通过增加域名校验机制，可以从已匹配到的应用中筛选过滤出目标应用，消除应用查询和定位中产生的歧义，直达受信的目标应用。

相比Deep Linking，App Linking具有更高的安全性和可靠性，用户体验更佳，推荐作开发者的首选App Linking方案。

| 类型 | Deep Linking | App Linking |
| --- | --- | --- |
| 实现效果 | - 如果已安装目标应用，将直接拉起目标应用。<br>- 如果未安装目标应用，将提示16000019错误码。 | - 如果已安装目标应用，将直接拉起目标应用。<br>- 如果未安装目标应用，会跳转到对应的浏览器来打开网址。 |
| 约束限制 | - URL的Scheme可以自定义。<br>- 无需校验域名。 | - URL的Scheme必须为https。<br>- 需要云侧服务器进行域名校验。 |
| 安全性 | 缺乏域名校验机制，容易被其他应用所仿冒。 | 通过域名校验机制，可以匹配到受信的目标应用。 |

# 使用App Linking实现应用间跳转

**简介**

使用App Linking进行跳转时，系统会根据接口传入的uri信息（HTTPS链接）将用户引导至目标应用中的特定内容，无论应用是否已安装，用户都可以访问到链接对应的内容，整个跳转体验相比Deep Linking方式更加顺畅。
例如：当开发者使用App Linking接入“扫码直达”服务后，用户可通过控制中心扫一扫等系统级扫码入口，扫描应用的二维码、条形码并跳转到开发者应用对应服务页，实现一步直达的体验。

**说明**

该能力目前仅适用于API 12及以上版本的HarmonyOS应用，如果您开发的是元服务，请参考使用App Linking实现元服务跳转。

**适用场景**

- 适用于应用的扫码直达、社交分享、沉默唤醒、广告引流等场景。
- 适用于对安全性要求较高的场景，避免出现被其它应用仿冒的问题。
- 适用于对体验要求较高的应用，不管目标应用是否安装，用户点击该链接都可以正常访问。

**实现原理**

- App Linking在Deep Linking基础上增加了域名校验环节，通过域名校验，可帮助用户消除歧义，识别合法归属于域名的应用，使链接更加安全可靠。
- App Linking要求对于同一HTTPS网址，有应用和网页两种内容的呈现方式。当应用安装时则优先打开应用去呈现内容；当应用未安装时，则打开浏览器呈现Web版的内容。

# 显式Want跳转切换应用链接跳转适配指导

**启动其他应用的UIAbility**

- 将待跳转的应用安装到设备，在其对应UIAbility的module.json5配置文件中配置skills标签的entities字段、actions字段和uri字段：
  - "actions"列表中包含"ohos.want.action.viewData"。
  - "entities"列表中包含"entity.system.browsable"。
  - "uris"列表中包含"scheme"为"https"且"domainVerify"为true的元素。uri的匹配规则参考uri匹配, domainVerify为true代表开启域名检查，通过applinking匹配该应用时需经过配置的域名校验后才能匹配到。applinking域名配置具体可参考AppLinking。

```
{
  "module": {
    // ...
    "abilities": [
      {
        // ...
        "skills": [
          {
            "entities": [
              "entity.system.browsable"
            ],
            "actions": [
              "ohos.want.action.viewData"
            ],
            "uris": [
              {
                "scheme": "https",
                "host": "www.example.com",
              }
            ],
            "domainVerify": true
          }
        ]
      }
    ]
  }
}
```

# 显式Want跳转切换应用链接跳转适配指导

**启动其他应用的UIAbility**

- 
- 调用方通过openLink接口执行跳转，在接口入参需要传入转换后的link和配置options，不再传入bundleName、moduleName和abilityName。系统会根据传入的link匹配到符合skill配置的应用。
  - 当options中的appLinkingOnly为true时，匹配到的应用会经过应用市场域名检查（需联网）返回域名校验检查的唯一匹配项或未匹配结果。
  - 当options中的appLinkingOnly为false时，会优先尝试以AppLinking的方式拉起，如果没有匹配的应用则改为使用DeepLinking的方式拉起目标应用。

```
import { common } from '@kit.AbilityKit';
import OpenLinkOptions from '@ohos.app.ability.OpenLinkOptions';
import { BusinessError } from '@ohos.base';
import hilog from '@ohos.hilog';

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

@Entry
@Component
struct Index {
  build() {
    Button('start link', { type: ButtonType.Capsule, stateEffect: true })
      .width('87%')
      .height('5%')
      .margin({ bottom: '12vp' })
      .onClick(() => {
        let context: common.UIAbilityContext = getContext(this) as common.UIAbilityContext;
        let link: string = "https://www.example.com";
        let openLinkOptions: OpenLinkOptions = {
          // 匹配的abilities选项是否需要通过AppLinking域名校验，匹配到唯一配置过的应用ability
          appLinkingOnly: true,
          // 同want中的parameter，用于传递的参数
          parameters: {demo_key: "demo_value"}
        };

        try {
          context.openLink(link, openLinkOptions)
            .then(() => {
              hilog.info(DOMAIN_NUMBER, TAG, 'open link success.');
            }).catch((err: BusinessError) => {
              hilog.error(DOMAIN_NUMBER, TAG, `open link failed. Code is ${err.code}, message is ${err.message}`);
            })
        } catch (paramError) {
          hilog.error(DOMAIN_NUMBER, TAG, `Failed to start link. Code is ${paramError.code}, message is ${paramError.message}`);
        }
      })
  }
}
```

# 显式Want跳转切换应用链接跳转适配指导

**启动其他应用的UIAbility并获取返回结果**

- 将待跳转的应用安装到设备，在其对应UIAbility的module.json5配置文件中配置skills标签的entities字段、actions字段和uri字段：
  - "actions"列表中包含"ohos.want.action.viewData"。
  - "entities"列表中包含"entity.system.browsable"。
  - "uris"列表中包含"scheme"为"https"且"domainVerify"为true的元素。uri的匹配规则参考uri匹配, domainVerify为true代表开启域名检查，通过applinking匹配该应用时需经过配置的域名校验后才能匹配到。applinking域名配置具体可参考App Linking。

```
{
  "module": {
    // ...
    "abilities": [
      {
        // ...
        "skills": [
          {
            "entities": [
              "entity.system.browsable"
            ],
            "actions": [
              "ohos.want.action.viewData"
            ],
            "uris": [
              {
                "scheme": "https",
                "host": "www.example.com",
              }
            ],
            "domainVerify": true
          }
        ]
      }
    ]
  }
}
```

# 显式Want跳转切换应用链接跳转适配指导

**启动其他应用的UIAbility并获取返回结果**

- 调用方通过openLink接口执行跳转，在接口入参需要传入转换后的link和配置options，不再传入bundleName、moduleName和abilityName。系统会根据传入的link匹配到符合skills配置的应用。AbilityResult回调结果返回通过入参传入回调函数，在启动ability停止自身后返回给调用方的信息。启动成功和失败结果仍通过Promise返回。
  - 当options中的appLinkingOnly为true时，匹配到的应用会经过应用市场域名检查（需联网）返回域名校验检查的唯一匹配项或未匹配结果。
  - 当options中的appLinkingOnly为false时，会优先尝试以AppLinking的方式拉起，如果没有匹配的应用则改为使用DeepLinking的方式拉起目标应用。

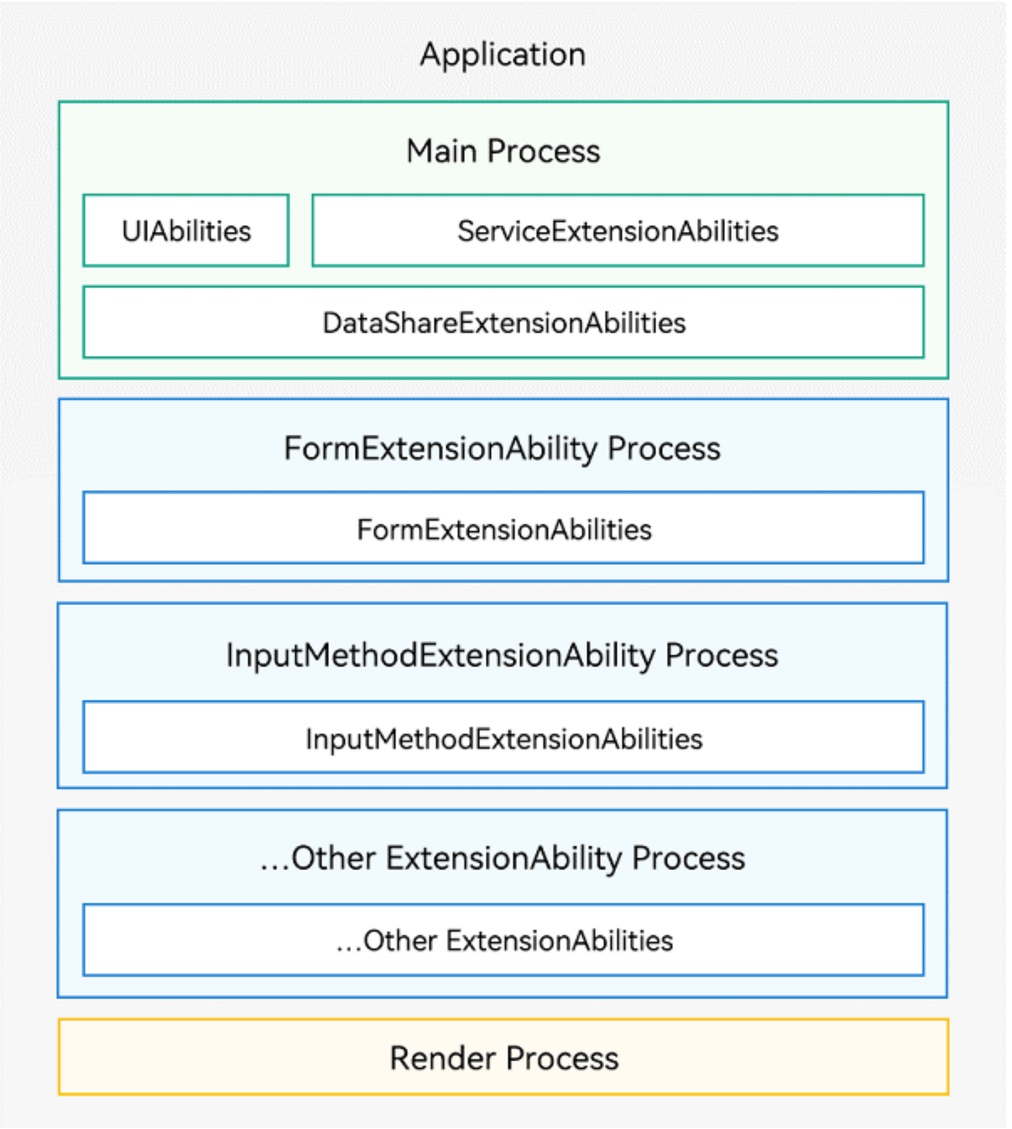
```
import { common } from '@kit.AbilityKit';
import OpenLinkOptions from '@ohos.app.ability.OpenLinkOptions';
import { BusinessError } from '@ohos.base';
import hilog from '@ohos.hilog';

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

@Entry
@Component
struct Index {
  build() {
    Button('start link', { type: ButtonType.Capsule, stateEffect: true })
      .width('87%')
      .height('5%')
      .margin({ bottom: '12vp' })
      .onClick(() => {
        let context: common.UIAbilityContext = getContext(this) as common.UIAbilityContext;
        let link: string = "https://www.example.com";
        let openLinkOptions: OpenLinkOptions = {
          // 匹配的abilities选项是否需要通过AppLinking域名校验，匹配到唯一配置过的应用ability
          appLinkingOnly: true,
          // 同want中的parameter，用于传递的参数
          parameters: {demo_key: "demo_value"}
        };
        try {
          context.openLink(link, openLinkOptions, (err, data) => {
            // AbilityResult callback回调，仅在被拉起ability死亡时触发
            hilog.info(DOMAIN_NUMBER, TAG, 'open link success. Callback result:' + JSON.stringify(data));
          }).then(() => {
            hilog.info(DOMAIN_NUMBER, TAG, 'open link success.');
          }).catch((err: BusinessError) => {
            hilog.error(DOMAIN_NUMBER, TAG, `open link failed. Code is ${err.code}, message is ${err.message}`);
          })
        } catch (paramError) {
          hilog.error(DOMAIN_NUMBER, TAG, `Failed to start link. Code is ${paramError.code}, message is ${paramError.message}`);
        }
      })
  }
}
```

# Stage模型的进程和线程

# 进程模型

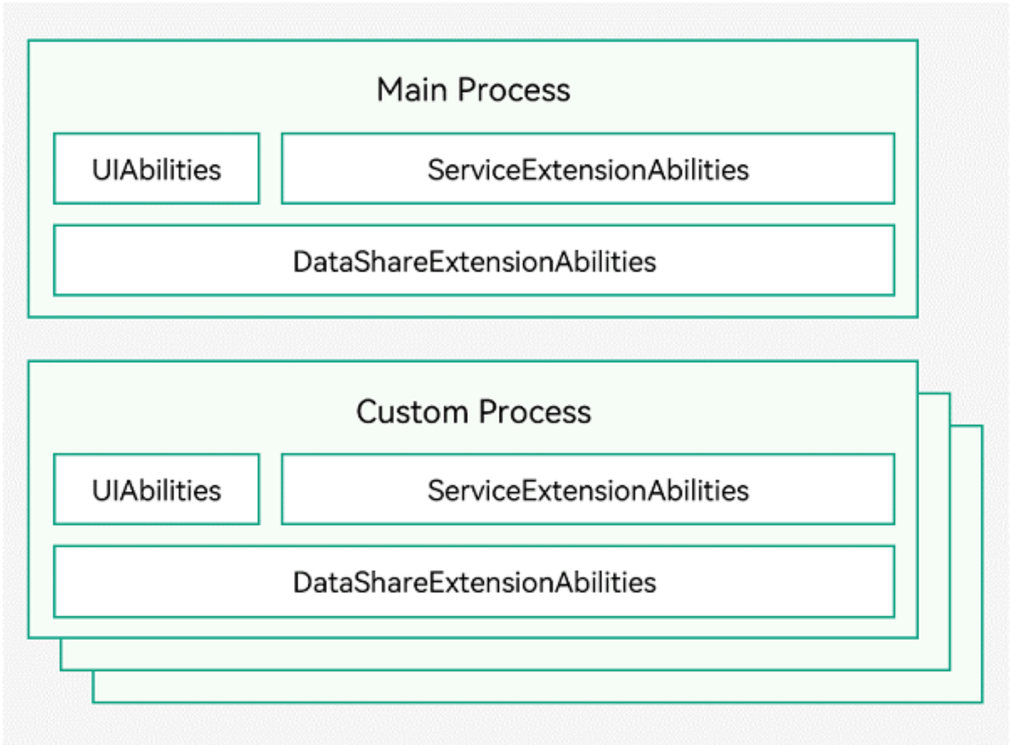
系统的进程模型如下图所示。

- 应用中（同一Bundle名称）的所有UIAbility、ServiceExtensionAbility和DataShareExtensionAbility均是运行在同一个独立进程（主进程）中，如下图中绿色部分的“Main Process”。
- 应用中（同一Bundle名称）的所有同一类型ExtensionAbility（除ServiceExtensionAbility和DataShareExtensionAbility外）均是运行在一个独立进程中，如下图中蓝色部分的“FormExtensionAbility Process”、“InputMethodExtensionAbility Process”、其他ExtensionAbility Process。
- WebView拥有独立的渲染进程，如下图中黄色部分的“Render Process”。



# 系统应用的多进程机制

对于系统应用可以通过申请多进程权限，为指定HAP配置一个自定义进程名，该HAP中的UIAbility、DataShareExtensionAbility、ServiceExtensionAbility就会运行在自定义进程中。不同的HAP可以通过配置不同的进程名运行在不同进程中。



# 线程模型

Stage模型下的线程主要有如下三类：

- 主线程
  - 执行UI绘制。
  - 管理主线程的ArkTS引擎实例，使多个UIAbility组件能够运行在其之上。
  - 管理其他线程的ArkTS引擎实例，例如使用TaskPool（任务池）创建任务或取消任务、启动和终止Worker线程。
  - 分发交互事件。
  - 处理应用代码的回调，包括事件处理和生命周期管理。
  - 接收TaskPool以及Worker线程发送的消息。
- TaskPool Worker线程
  - 用于执行耗时操作，支持设置调度优先级、负载均衡等功能，推荐使用。
- Worker线程
  - 用于执行耗时操作，支持线程间通信。

# Basics Service Kit的应用事件（进程间通信、线程间通信）

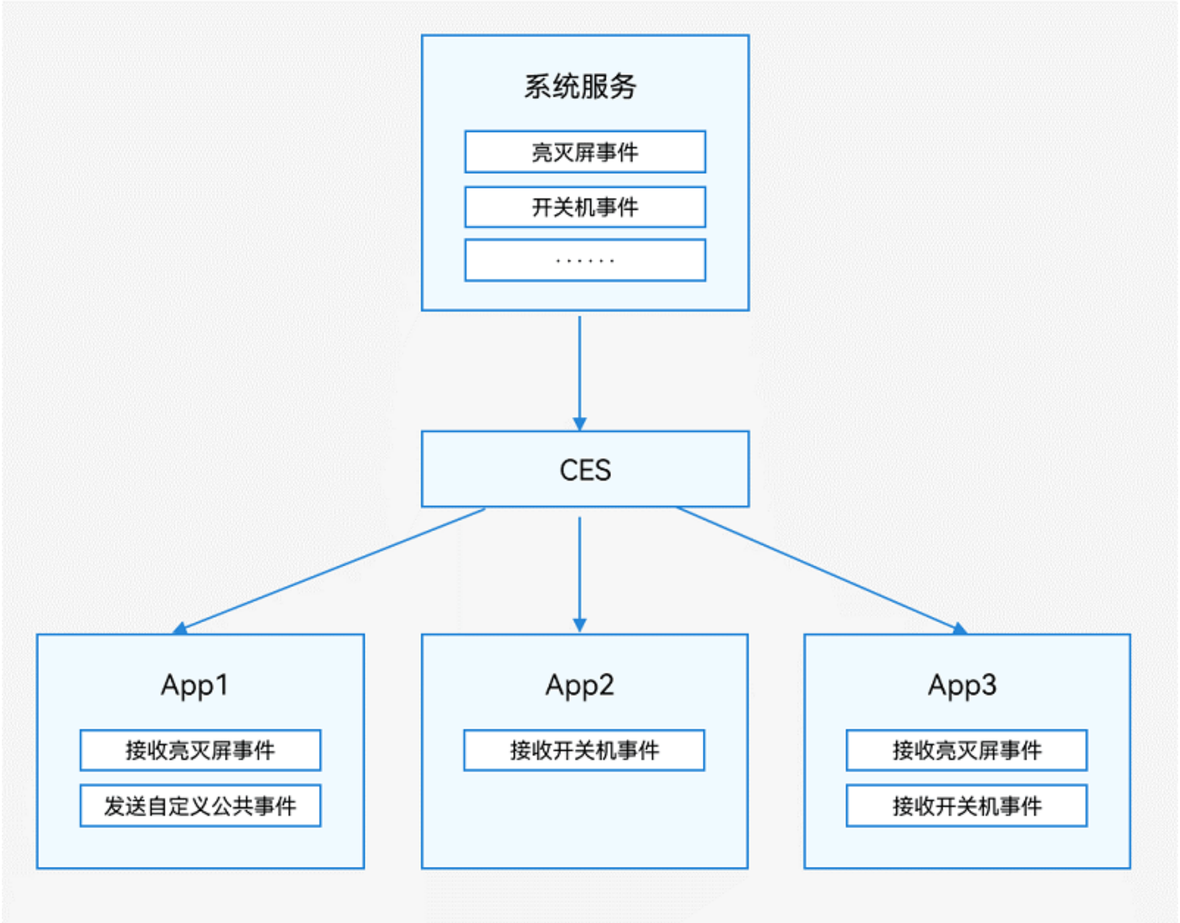
# 公共事件服务（CES）

公共事件从系统角度可分为：系统公共事件和自定义公共事件。

- 系统公共事件：CES内部定义的公共事件，当前仅支持系统应用和系统服务发布，例如HAP安装、更新、卸载等公共事件。目前支持的系统公共事件请参见系统公共事件列表。
- 自定义公共事件：应用定义的公共事件，可用于实现跨进程的事件通信能力。

公共事件按发送方式可分为：无序公共事件、有序公共事件和粘性公共事件。

- 无序公共事件：CES在转发公共事件时，不考虑订阅者是否接收到该事件，也不保证订阅者接收到该事件的顺序与其订阅顺序一致。
- 有序公共事件：CES在转发公共事件时，根据订阅者设置的优先级等级，优先将公共事件发送给优先级较高的订阅者，等待其成功接收该公共事件之后再将事件发送给优先级较低的订阅者。如果有多个订阅者具有相同的优先级，则他们将随机接收到公共事件。
- 粘性公共事件：能够让订阅者收到在订阅前已经发送的公共事件就是粘性公共事件。普通的公共事件只能在订阅后发送才能收到，而粘性公共事件的特殊性就是可以先发送后订阅，同时也支持先订阅后发送。发送粘性事件必须是系统应用或系统服务，粘性事件发送后会一直存在系统中，且发送者需要申请ohos.permission.COMMONEVENT_STICKY权限，配置方式请参见声明权限。


系统服务
亮灭屏事件
开关机事件
.....
CES
App1
接收亮灭屏事件
发送自定义公共事件
App2
接收开关机事件
App3
接收亮灭屏事件
接收开关机事件

# 运作机制

# 公共事件订阅

### 场景介绍

动态订阅是指当应用在运行状态时对某个公共事件进行订阅，在运行期间如果有订阅的事件发布那么订阅了这个事件的应用将会收到该事件及其传递的参数。
例如，某应用希望在其运行期间收到电量过低的事件，并根据该事件降低其运行功耗，那么该应用便可动态订阅电量过低事件，收到该事件后关闭一些非必要的任务来降低功耗。
订阅部分系统公共事件需要先申请权限，订阅这些事件所需要的权限请见公共事件权限列表。

### 接口说明

详细接口见接口文档。
**接口名 接口描述**
createSubscriber(subscribeInfo: CommonEventSubscribeInfo, callback: AsyncCallback<CommonEventSubscriber>): void
创建订阅者对象（callback）。
createSubscriber(subscribeInfo: CommonEventSubscribeInfo): Promise<CommonEventSubscriber>
创建订阅者对象（promise）。
subscribe(subscriber: CommonEventSubscriber, callback: AsyncCallback): void
订阅公共事件。

# 开发步骤

**开发步骤**

- 导入模块。

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

const TAG: string = 'ProcessModel';
const DOMAIN_NUMBER: number = 0xFF00;
```

- 创建订阅者信息，详细的订阅者信息数据类型及包含的参数请见CommonEventSubscribeInfo文档介绍。

```
// 用于保存创建成功的订阅者对象，后续使用其完成订阅及退订的动作
let subscriber: commonEventManager.CommonEventSubscriber | null = null;
// 订阅者信息，其中的event字段需要替换为实际的事件名称。
let subscribeInfo: commonEventManager.CommonEventSubscribeInfo = {
    events: ['event'], // 订阅灭屏公共事件
};
```

- 创建订阅者，保存返回的订阅者对象subscriber，用于执行后续的订阅、退订等操作。

```
// 创建订阅者回调
commonEventManager.createSubscriber(subscribeInfo, (err: BusinessError, data: commonEventManager.CommonEventSubscriber) => {
  if (err) {
    hilog.error(DOMAIN_NUMBER, TAG, `Failed to create subscriber. Code is ${err.code}, message is ${err.message}`);
    return;
  }
  hilog.info(DOMAIN_NUMBER, TAG, 'Succeeded in creating subscriber.');
  subscriber = data;
})
```

- 创建订阅回调函数，订阅回调函数会在接收到事件时触发。订阅回调函数返回的data内包含了公共事件的名称、发布者携带的数据等信息，公共事件数据的详细参数和数据类型请见CommonEventData文档介绍。

```
// 订阅公共事件回调
if (subscriber !== null) {
  commonEventManager.subscribe(subscriber, (err: BusinessError, data: commonEventManager.CommonEventData) => {
    if (err) {
      hilog.error(DOMAIN_NUMBER, TAG, `Failed to subscribe common event. Code is ${err.code}, message is ${err.message}`);
      return;
    }
    // ...
  })
} else {
  hilog.error(DOMAIN_NUMBER, TAG, `Need create subscriber`);
}
```

# 取消订阅

**场景介绍**

动态订阅者完成业务需要时，需要主动取消订阅，订阅者通过调用unsubscribe()方法取消订阅事件。

**接口说明**

**接口名**

**接口描述**

unsubscribe(subscriber: CommonEventSubscriber, callback?: AsyncCallback)

取消订阅公共事件。

**开发步骤**

- 导入模块。

```
import { BusinessError, commonEventManager } from '@kit.BasicServicesKit';
import { hilog } from '@kit.PerformanceAnalysisKit';

const TAG: string = 'ProcessModel';
const DOMAIN_NUMBER: number = 0xFF00;
```

- 根据动态订阅公共事件章节的步骤来订阅某个事件。
- 调用CommonEvent中的unsubscribe()方法取消订阅某事件。

```
// subscriber为订阅事件时创建的订阅者对象
if (this.subscriber !== null) {
commonEventManager.unsubscribe(this.subscriber, (err: BusinessError) => {
    if (err) {
      hilog.error(DOMAIN_NUMBER, TAG, `UnsubscribeCallBack err = ${JSON.stringify(err)}`);
    } else {
      hilog.info(DOMAIN_NUMBER, TAG, `Unsubscribe success`);
      this.subscriber = null;
    }
  })
}
```

# 公共事件发布 - 1

**场景介绍**

当需要发布某个自定义公共事件时，可以通过publish()方法发布事件。发布的公共事件可以携带数据，供订阅者解析并进行下一步处理。

**注意**

已发出的粘性公共事件后来订阅者也可以接收到，其他公共事件都需要先订阅再接收，订阅参考公共事件订阅章节。

**接口说明**

详细接口见接口文档。

**接口名**

**接口描述**

publish(event: string, callback: AsyncCallback)
发布公共事件。
publish(event: string, options: CommonEventPublishData, callback: AsyncCallback)
指定发布信息并发布公共事件。

**发布不携带信息的公共事件**

不携带信息的公共事件，只能发布无序公共事件。

- 导入模块。

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

const TAG: string = 'ProcessModel';
const DOMAIN_NUMBER: number = 0xFF00;

```

- 传入需要发布的事件名称和回调函数，发布事件。

```
// 发布公共事件，其中的event字段需要替换为实际的事件名称。
commonEventManager.publish('event', (err: BusinessError) => {
  if (err) {
    hilog.info(DOMAIN_NUMBER, TAG, `PublishCallBack err = ${JSON.stringify(err)}`);
  } else {
    //...
    hilog.info(DOMAIN_NUMBER, TAG, `Publish success`);
  }
});
```

# 公共事件发布 - 2

**发布携带信息的公共事件**

携带信息的公共事件，可以发布为无序公共事件、有序公共事件和粘性事件，可以通过参数CommonEventPublishData的isOrdered、isSticky的字段进行设置。

- 导入模块。

```
import { BusinessError, commonEventManager } from '@kit.BasicServicesKit';
import { hilog } from '@kit.PerformanceAnalysisKit';

const TAG: string = 'ProcessModel';
const DOMAIN_NUMBER: number = 0xFF00;
```

- 构建需要发布的公共事件信息。

```
// 公共事件相关信息
let options: commonEventManager.CommonEventPublishData = {
    code: 1, // 公共事件的初始代码
    data: 'initial data', // 公共事件的初始数据
};
```

- 传入需要发布的事件名称、需要发布的指定信息和回调函数，发布事件。

```
// 发布公共事件，其中的event字段需要替换为实际的事件名称。
commonEventManager.publish('event', options, (err: BusinessError) => {
    if (err) {
      hilog.error(DOMAIN_NUMBER, TAG, 'PublishCallBack err = ' + JSON.stringify(err));
    } else {
      //...
      hilog.info(DOMAIN_NUMBER, TAG, 'Publish success');
    }
});
```

# 使用Emitter进行线程间通信

Emitter主要提供线程间发送和处理事件的能力，包括对持续订阅事件或单次订阅事件的处理、取消订阅事件、发送事件到事件队列等。
Emitter的开发步骤如下：

- 订阅事件

```
import { emitter } from '@kit.BasicServicesKit';
import { promptAction } from '@kit.ArkUI';
import { hilog } from '@kit.PerformanceAnalysisKit';

const TAG: string = 'ThreadModel';
const DOMAIN_NUMBER: number = 0xFF00;

// 定义一个eventId为1的事件
let event: emitter.InnerEvent = {
  eventId: 1
};

// 收到eventId为1的事件后执行该回调
let callback = (eventData: emitter.EventData): void => {
  promptAction.showToast({
    message: JSON.stringify(eventData)
  });
  hilog.info(DOMAIN_NUMBER, TAG, 'event callback:' + JSON.stringify(eventData));
};

// 订阅eventId为1的事件
emitter.on(event, callback);
promptAction.showToast({
  message: JSON.stringify('emitter subscribe success')
});
```

- 发送事件

```
import { emitter } from '@kit.BasicServicesKit';

// 定义一个eventId为1的事件，事件优先级为Low
let event: emitter.InnerEvent = {
  eventId: 1,
  priority: emitter.EventPriority.LOW
};

let eventData: emitter.EventData = {
  data: {
    content: 'c',
    id: 1,
    isEmpty: false
  }
};

// 发送eventId为1的事件，事件内容为eventData
emitter.emit(event, eventData);
```

# 实践案例：SystemCommonEvent

# Background Tasks Kit

# Background Tasks Kit简介

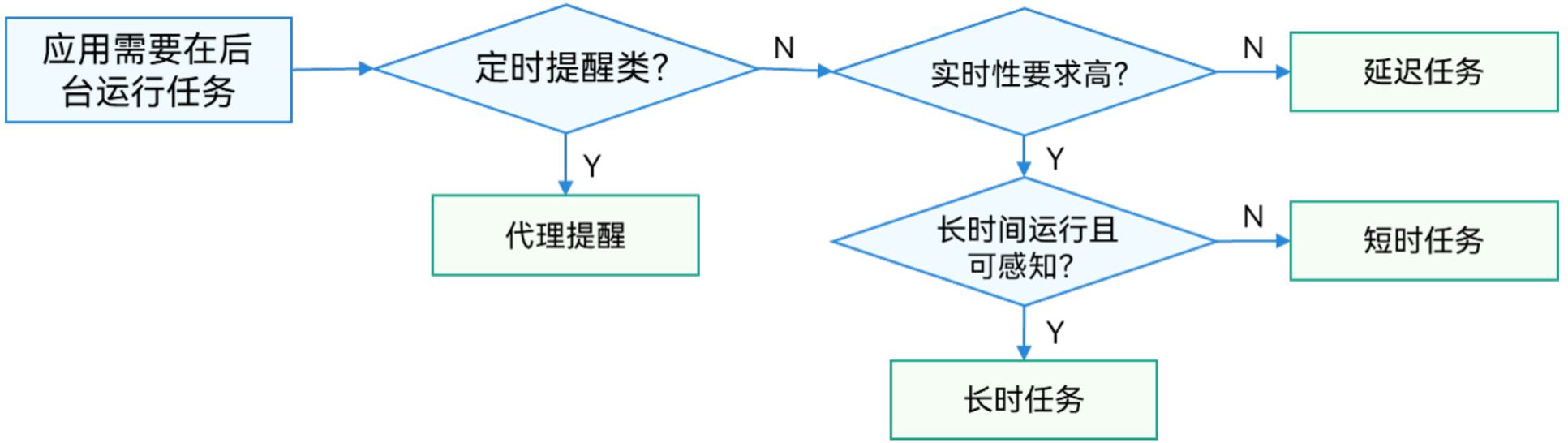
- 设备返回主界面、锁屏、应用切换等操作会使应用退至后台。应用退至后台后，如果继续活动，可能会造成设备耗电快、用户界面卡顿等现象。为了降低设备耗电速度、保障用户使用流畅度，系统会对退至后台的应用进行管控，包括进程挂起和进程终止。挂起后，应用进程无法使用软件资源（如公共事件、定时器等）和硬件资源（CPU、网络、GPS、蓝牙等）。如何合理使用请参考合理使用后台硬件资源。

- 应用退至后台一小段时间（由系统定义），应用进程会被挂起。

- 应用退至后台，在后台被访问一小段时间（由系统定义）后，应用进程会被挂起。

- 资源不足时，系统会终止部分应用进程（即回收该进程的所有资源）。

- 同时，为了保障后台音乐播放、日历提醒等功能的正常使用，系统提供了规范内受约束的后台任务，扩展应用在后台运行时间




开发者可以根据如下的功能介绍，选择合适的后台任务，以满足应用退至后台后继续运行的需求。

- 短时任务：适用于实时性要求高、耗时不长的任务，例如状态保存。
- 长时任务：适用于长时间运行在后台、用户可感知的任务，例如后台播放音乐、导航、设备连接等，使用长时任务避免应用进程被挂起。
- 延迟任务：对于实时性要求不高、可延迟执行的任务，系统提供了延迟任务，即满足条件的应用退至后台后被放入执行队列，系统会根据内存、功耗等统一调度。
- 代理提醒：代理提醒是指应用退后台或进程终止后，系统会代理应用做相应的提醒。适用于定时提醒类业务，当前支持的提醒类型包括倒计时、日历和闹钟三类。

# 后台任务类型

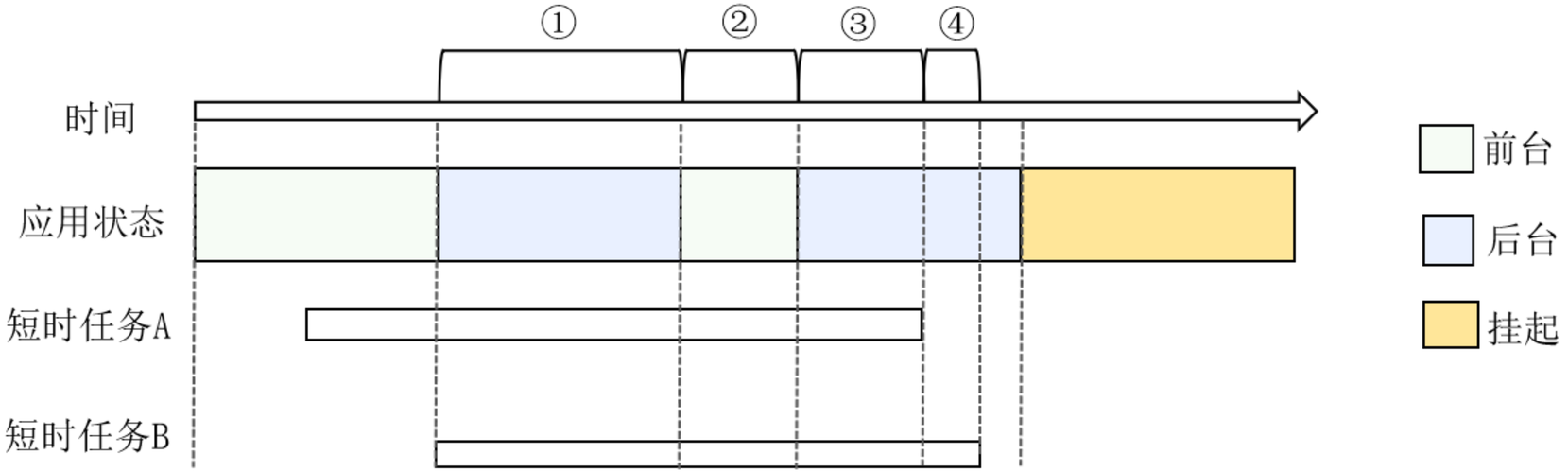
# 短时任务

**概述**

应用退至后台一小段时间后，应用进程会被挂起，无法执行对应的任务。如果应用在后台仍需要执行耗时不长的任务，如状态保存等，可以通过本文申请短时任务，扩展应用在后台的运行时间。

**约束与限制**

- **申请时机**：应用需要在前台或onBackground回调内，申请短时任务，否则会申请失败。
- **数量限制**：一个应用同一时刻最多申请3个短时任务。以图1为例，在①②③时间段内的任意时刻，应用申请了2个短时任务；在④时间段内的任意时刻，应用申请了1个短时任务。
- **配额机制**：一个应用会有一定的短时任务配额（根据系统状态和用户习惯调整），单日（24小时内）配额默认为10分钟，单次配额最大为3分钟，低电量时单次配额默认为1分钟，配额消耗完后不允许再申请短时任务。同时，系统提供获取对应短时任务剩余时间的查询接口，用以查询本次短时任务剩余时间，以确认是否继续运行其他业务。
- **配额计算**：仅当应用在后台时，对应用下的短时任务计时；同一个应用下的同一个时间段的短时任务，不重复计时。以下图为例：应用有两个短时任务A和B，在前台时申请短时任务A，应用退至后台后开始计时为①，应用进入前台②后不计时，再次进入后台③后开始计时，短时任务A结束后，由于阶段④仍然有短时任务B，所以该阶段继续计时。因此，在这个过程中，该应用短时任务总耗时为①+③+④。



| 参数名 | 描述 | 配置项 | 场景举例 |
| --- | --- | --- | --- |
| DATA_TRANSFER | 数据传输 | dataTransfer | 后台下载大文件，如浏览器后台下载等。 |
| AUDIO_PLAYBACK | 音视频播放 | audioPlayback | 音乐类应用在后台播放音乐。<br>支持在元服务中使用。 |
| AUDIO_RECORDING | 录制 | audioRecording | 录音机在后台录音。 |
| LOCATION | 定位导航 | location | 导航类应用后台导航。 |
| BLUETOOTH_INTERACTION | 蓝牙相关 | bluetoothInteraction | 通过蓝牙传输分享的文件。 |
| MULTI_DEVICE_CONNECTION | 多设备互联 | multiDeviceConnection | 分布式业务连接。<br>支持在元服务中使用。 |
| TASK_KEEPING | 计算任务（仅对2in1开放） | taskKeeping | 杀毒软件。 |

# 长时任务类型

应用退至后台后，在后台需要长时间运行用户可感知的任务，如播放音乐、导航等。为防止应用进程被挂起，导致对应功能异常，可以申请长时任务，使应用在后台长时间运行。
申请长时任务后，系统会做相应的校验，确保应用在执行相应的长时任务。同时，系统有与长时任务相关联的通知栏消息，用户删除通知栏消息时，系统会自动停止长时任务。

# 开发步骤

接口说明

表2 主要接口

以下是长时任务开发使用的相关接口，下表均以Promise形式为例，更多接口及使用方式请见后台任务管理。

| 接口名 | 描述 |
| --- | --- |
| startBackgroundRunning(context: Context, bgMode: BackgroundMode, wantAgent: WantAgent): Promise<void> | 申请长时任务 |
| stopBackgroundRunning(context: Context): Promise<void> | 取消长时任务 |

开发步骤

本文以申请录制长时任务为例，示例中包含“申请长时任务”和“取消长时任务”两个按钮，显示效果为：

- 点击“申请长时任务”按钮，应用申请录制长时任务成功，通知栏显示“正在运行录制任务”通知。
- 点击“取消长时任务”按钮，取消长时任务，通知栏撤销相关通知。

**Stage模型**

- 需要申请ohos.permission.KEEP_BACKGROUND_RUNNING权限，配置方式请参见声明权限。
- 声明后台模式类型，以及添加uris等配置。
  - 声明后台模式类型（必填项）：在module.json5配置文件中为需要使用长时任务的UIAbility声明相应的长时任务类型（配置文件中填写长时任务类型的配置项）。
  - 添加uris等配置（可选项）：若使用deeplink、applink等跳转功能，具体请参考如下示例可选项。其中，必填项的配置不可更改，可选项的具体配置请参考应用间跳转。

```
"module": {
    "abilities": [
        {
            "backgroundModes": [
             // 长时任务类型的配置项
            "audioRecording"
            ],
            "skills": [
                // 必填项：申请长时任务时entities和actions值
                {
                    "entities": [
                        "entity.system.home"
                    ],
                    "actions": [
                        "action.system.home"
                    ]
                },
                // 可选项：添加deeplink、applink等跳转功能
                {
                    "entities": [
                        "test"
                    ],
                    "actions": [
                        "test"
                    ],
                    "uris": [
                        {
                            "scheme": "test"
                        }
                    ]
                }
            ]
        }
    ],
    ...
}
```

- 导入模块。

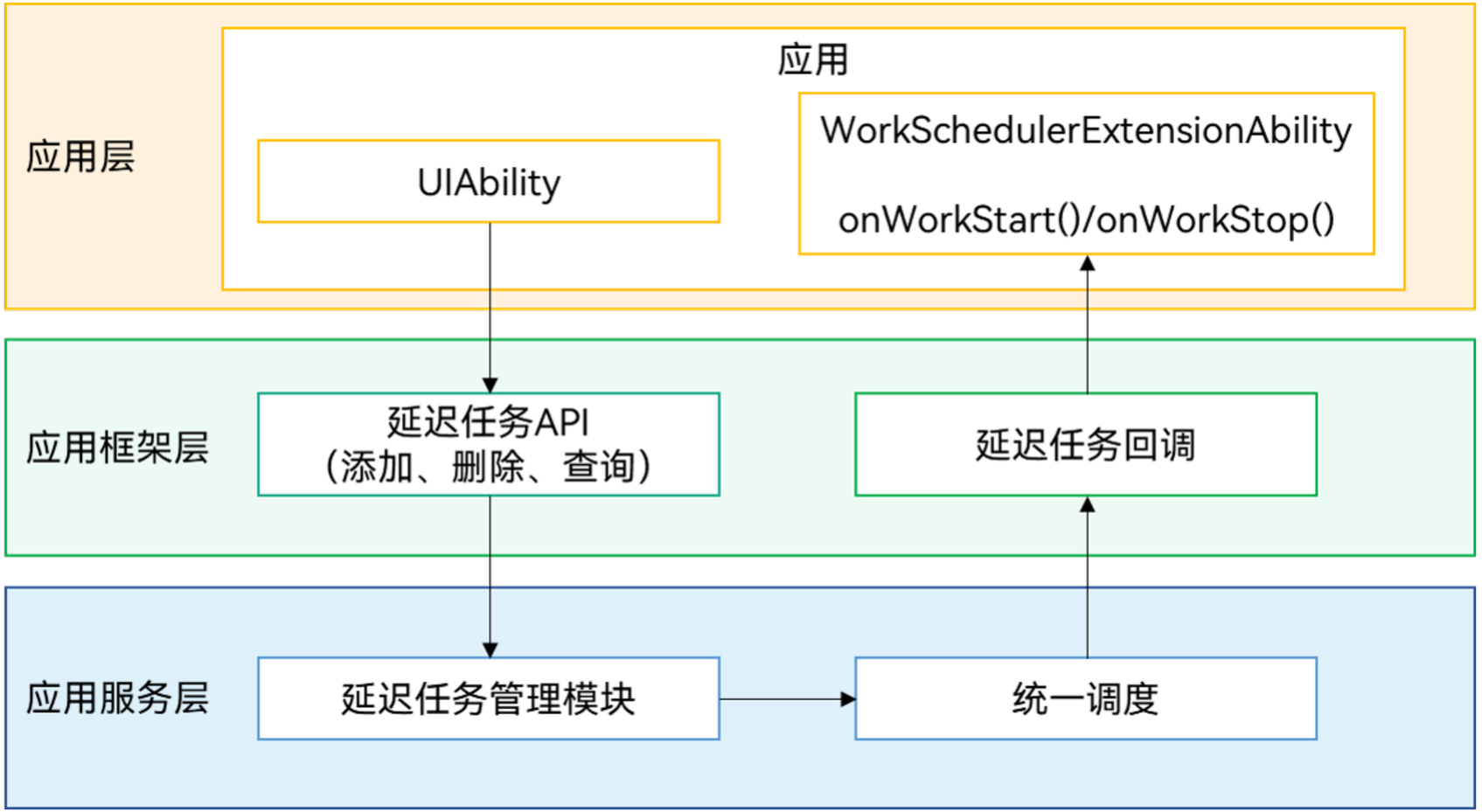
  长时任务相关的模块为@ohos.resourceschedule.backgroundTaskManager和@ohos.app.ability.wantAgent，其余模块按实际需要导入。

```
import { backgroundTaskManager } from '@kit.BackgroundTasksKit';
import { AbilityConstant, UIAbility, Want } from '@kit.AbilityKit';
import { window } from '@kit.ArkUI';
import { rpc } from '@kit.IPCKit'
import { BusinessError } from '@kit.BasicServicesKit';
import { wantAgent, WantAgent } from '@kit.AbilityKit';
```

# 开发步骤

- 申请和取消长时任务。
  - 在Stage模型中，长时任务支持设备本应用申请，也支持跨设备或跨应用申请，跨设备或跨应用仅对系统应用开放。
- 设备本应用申请长时任务示例代码如下：

```
@Entry
@Component
struct Index {
  @State message: string = 'ContinuousTask';
 // 通过getContext方法，来获取page所在的UIAbility上下文。
  private context: Context = getContext(this);

  startContinuousTask() {
    let wantAgentInfo: wantAgent.WantAgentInfo = {
      // 点击通知后，将要执行的动作列表
      // 添加需要被拉起应用的bundleName和abilityName
      wants: [
        {
          bundleName: "com.example.myapplication",
          abilityName: "MainAbility"
        }
      ],
      // 指定点击通知栏消息后的动作是拉起ability
      actionType: wantAgent.OperationType.START_ABILITY,
      // 使用者自定义的一个私有值
      requestCode: 0,
      // 点击通知后，动作执行属性
      actionFlags: [wantAgent.WantAgentFlags.UPDATE_PRESENT_FLAG]
    };

    // 通过wantAgent模块下getWantAgent方法获取WantAgent对象
    wantAgent.getWantAgent(wantAgentInfo).then((wantAgentObj: WantAgent) => {
      backgroundTaskManager.startBackgroundRunning(this.context,
        backgroundTaskManager.BackgroundMode.AUDIO_RECORDING, wantAgentObj).then(() => {
        // 此处执行具体的长时任务逻辑，如放音等。
        console.info(`Succeeded in operationing startBackgroundRunning.`);
      }).catch((err: BusinessError) => {
        console.error(`Failed to operation startBackgroundRunning. Code is ${err.code}, message is ${err.message}`);
      });
    });
  }

  stopContinuousTask() {
    backgroundTaskManager.stopBackgroundRunning(this.context).then(() => {
      console.info(`Succeeded in operationing stopBackgroundRunning.`);
    }).catch((err: BusinessError) => {
      console.error(`Failed to operation stopBackgroundRunning. Code is ${err.code}, message is ${err.message}`);
    });
  }

  build() {
    Row() {
      Column() {
        Text("Index")
          .fontSize(50)
          .fontWeight(FontWeight.Bold)

        Button() {
          Text('申请长时任务').fontSize(25).fontWeight(FontWeight.Bold)
        }
        .type(ButtonType.Capsule)
        .margin({ top: 10 })
        .backgroundColor('#0D9FFB')
        .width(250)
        .height(40)
        .onClick(() => {
          // 通过按钮申请长时任务
          this.startContinuousTask();
        })

        Button() {
          Text('取消长时任务').fontSize(25).fontWeight(FontWeight.Bold)
        }
        .type(ButtonType.Capsule)
        .margin({ top: 10 })
        .backgroundColor('#0D9FFB')
        .width(250)
        .height(40)
        .onClick(() => {
          // 此处结束具体的长时任务的执行

          // 通过按钮取消长时任务
          this.stopContinuousTask();
        })
      }
      .width('100%')
    }
    .height('100%')
  }
}
```




# 延迟任务

应用退至后台后，需要执行实时性要求不高的任务，例如有网络时不定期主动获取邮件等，可以使用延迟任务。当应用满足设定条件（包括网络类型、充电类型、存储状态、电池状态、定时状态等）时，将任务添加到执行队列，系统会根据内存、功耗、设备温度、用户使用习惯等统一调度拉起应用。

# 约束与限制

- **数量限制**：一个应用同一时刻最多申请10个延迟任务。
- **执行频率限制**：系统会根据应用的活跃分组，对延迟任务做分级管控，限制延迟任务调度的执行频率。
  **表1** 应用活跃程度分组
- **应用活跃分组**
- **延迟任务执行频率**
- 活跃分组
- 最小间隔2小时
- 经常使用分组
- 最小间隔4小时
- 常用使用
- 最小间隔24小时
- 极少使用分组
- 最小间隔48小时
- 受限使用分组
- 禁止
- 从未使用分组
- 禁止
-

- **超时**：WorkSchedulerExtensionAbility单次回调最长运行2分钟。如果超时不取消，系统会终止对应的Extension进程。
- **调度延迟**：系统会根据内存、功耗、设备温度、用户使用习惯等统一调度，如当系统内存资源不足或温度达到一定挡位时，系统将延迟调度该任务。
- **WorkSchedulerExtensionAbility接口调用限制**：为实现对WorkSchedulerExtensionAbility能力的管控，在WorkSchedulerExtensionAbility中限制以下接口的调用：
  @ohos.resourceschedule.backgroundTaskManager (后台任务管理)
  @ohos.backgroundTaskManager (后台任务管理)
  @ohos.multimedia.camera (相机管理)
  @ohos.multimedia.audio (音频管理)
  @ohos.multimedia.media (媒体服务)

| 接口名 | 接口描述 |
| --- | --- |
| startWork(work: WorkInfo): void; | 申请延迟任务 |
| stopWork(work: WorkInfo, needCancel?: boolean): void; | 取消延迟任务 |
| getWorkStatus(workId: number, callback: AsyncCallback<WorkInfo>): void; | 获取延迟任务状态（Callback形式） |
| getWorkStatus(workId: number): Promise<WorkInfo>; | 获取延迟任务状态（Promise形式） |
| obtainAllWorks(callback: AsyncCallback<Array<WorkInfo>>): void; | 获取所有延迟任务（Callback形式） |
| obtainAllWorks(): Promise<Array<WorkInfo>>; | 获取所有延迟任务（Promise形式） |
| stopAndClearWorks(): void; | 停止并清除任务 |
| isLastWorkTimeOut(workId: number, callback: AsyncCallback<boolean>): void; | 获取上次任务是否超时（针对RepeatWork，Callback形式） |
| isLastWorkTimeOut(workId: number): Promise<boolean>; | 获取上次任务是否超时（针对RepeatWork，Promise形式） |

# 接口说明

# 开发步骤

- 以下是延迟任务回调开发使用的相关接口，更多接口及使用方式请见延迟任务调度回调文档。
- 接口名　　接口描述
- onWorkStart(work: workScheduler.WorkInfo): void　　延迟调度任务开始的回调
- onWorkStop(work: workScheduler.WorkInfo): void　　延迟调度任务结束的回调
- **开发步骤**
- 延迟任务调度开发步骤分为两步：实现延迟任务调度扩展能力、实现延迟任务调度。
- 　延迟任务调度扩展能力：实现WorkSchedulerExtensionAbility开始和结束的回调接口。
- 　延迟任务调度：调用延迟任务接口，实现延迟任务申请、取消等功能。

# 开发步骤

**实现延迟任务回调拓展能力**

- 新建工程目录。
  在工程entry Module对应的ets目录(./entry/src/main/ets)下，新建目录及ArkTS文件，例如新建一个目录并命名为WorkSchedulerExtension。在WorkSchedulerExtension目录下，新建一个ArkTS文件并命名为WorkSchedulerExtension.ets，用以实现延迟任务回调接口。
- 导入模块。

```
import { WorkSchedulerExtensionAbility, workScheduler } from '@kit.BackgroundTasksKit';
```

- 实现WorkSchedulerExtension生命周期接口。

```
export default class MyWorkSchedulerExtensionAbility extends WorkSchedulerExtensionAbility {
  // 延迟任务开始回调
  onWorkStart(workInfo: workScheduler.WorkInfo) {
    console.info(`onWorkStart, workInfo = ${JSON.stringify(workInfo)}`);
    // 打印 parameters中的参数，如：参数key1
    // console.info(`work info parameters: ${JSON.parse(workInfo.parameters?.toString()).key1}`)
  }

  // 延迟任务结束回调
  onWorkStop(workInfo: workScheduler.WorkInfo) {
    console.info(`onWorkStop, workInfo is ${JSON.stringify(workInfo)}`);
  }
}
```

- 在module.json5配置文件中注册WorkSchedulerExtensionAbility，并设置如下标签：
  - type标签设置为“workScheduler”。
  - srcEntry标签设置为当前ExtensionAbility组件所对应的代码路径。

```
{
  "module": {
      "extensionAbilities": [
        {
          "name": "MyWorkSchedulerExtensionAbility",
          "srcEntry": "./ets/WorkSchedulerExtension/WorkSchedulerExtension.ets",
          "label": "$string:WorkSchedulerExtensionAbility_label",
          "description": "$string:WorkSchedulerExtensionAbility_desc",
          "type": "workScheduler"
        }
      ]
  }
}
```

# 开发步骤

**实现延迟任务调度**

- 导入模块。

```
import { workScheduler } from '@kit.BackgroundTasksKit';
import { BusinessError } from '@kit.BasicServicesKit';
```

- 申请延迟任务。

```
// 创建workinfo
const workInfo: workScheduler.WorkInfo = {
  workId: 1,
  networkType: workScheduler.NetworkType.NETWORK_TYPE_WIFI,
  bundleName: 'com.example.application',
  abilityName: 'MyWorkSchedulerExtensionAbility'
}

try {
  workScheduler.startWork(workInfo);
  console.info(`startWork success`);
} catch (error) {
  console.error(`startWork failed. code is ${(error as BusinessError).code} message is ${(error as BusinessError).message}`);
}
```

- 取消延迟任务。

```
// 创建workinfo
const workInfo: workScheduler.WorkInfo = {
  workId: 1,
  networkType: workScheduler.NetworkType.NETWORK_TYPE_WIFI,
  bundleName: 'com.example.application',
  abilityName: 'MyWorkSchedulerExtensionAbility'
}

try {
  workScheduler.stopWork(workInfo);
  console.info(`stopWork success`);
} catch (error) {
  console.error(`stopWork failed. code is ${(error as BusinessError).code} message is ${(error as BusinessError).message}`);
}
```

# 代理提醒

## 功能介绍

应用退到后台或进程终止后，仍然有一些提醒用户的定时类任务，例如时钟提醒等，为满足此类功能场景，系统提供了代理提醒（reminderAgentManager）的能力。当应用退至后台或进程终止后，系统会代理应用做相应的提醒。当前支持的提醒类型包括：倒计时、日历和闹钟。为了防止代理提醒被用于滥用于广告、营销类提醒，影响用户体验，代理增加了管控机制，管控后的使用方法请参考管控限制。

- 倒计时类：基于倒计时的提醒功能。
- 日历类：基于日历的提醒功能。
- 闹钟类：基于时钟的提醒功能。

## 约束与限制

- **个数限制**：一个三方应用支持最多30个有效提醒（有效即发布成功），一个系统应用支持最多10000个有效提醒，整个系统最多支持12000个有效提醒。
- **跳转限制**：点击提醒通知后跳转的应用必须是申请代理提醒的本应用。
- **管控限制**：管控后可通过日历Calendar Kit 替代代理提醒，实现相应的提醒功能，具体请参考开发指南；或者参考如下邮件格式向华为侧申请代理提醒权限，当前仅对纯工具类应用开放申请。

# 开发步骤

**接口说明**

以下是代理提醒的相关接口，下表均以Promise形式为例，更多接口及使用方式请见后台代理提醒文档。

**接口名**

**描述**

publishReminder(reminderReq: ReminderRequest): Promise<number>

发布一个定时提醒类通知

cancelReminder(reminderId: number): Promise<void>

取消一个指定的提醒类通知

getValidReminders(): Promise<Array<ReminderRequest>>

获取当前应用设置的所有有效的提醒

cancelAllReminders(): Promise<void>

取消当前应用设置的所有提醒

addNotificationSlot(slot: NotificationSlot): Promise<void>

注册一个提醒类需要使用的通知通道（NotificationSlot）

removeNotificationSlot(slotType: notification.SlotType): Promise<void>

删除指定的通知通道（NotificationSlot）

**开发步骤**

- 申请ohos.permission.PUBLISH_AGENT_REMINDER权限，配置方式请参阅声明权限。
- 请求通知授权。获得用户授权后，才能使用代理提醒功能。
- 导入模块。

```
import { reminderAgentManager } from '@kit.BackgroundTasksKit';
import { notificationManager } from '@kit.NotificationKit';
import { BusinessError } from '@kit.BasicServicesKit';
```

# 开发步骤

- 定义目标提醒代理。开发者根据实际需要，选择定义如下类型的提醒。
  - 定义倒计时实例。

```
let targetReminderAgent: reminderAgentManager.ReminderRequestTimer = {
  reminderType: reminderAgentManager.ReminderType.REMINDER_TYPE_TIMER,   // 提醒类型为倒计时类型
  triggerTimeInSeconds: 10,
  actionButton: [ // 设置弹出的提醒通知信息上显示的按钮类型和标题
    {
      title: 'close',
      type: reminderAgentManager.ActionButtonType.ACTION_BUTTON_TYPE_CLOSE
    }
  ],
  wantAgent: {     // 点击提醒通知后跳转的目标UIAbility信息
    pkgName: 'com.example.myapplication',
    abilityName: 'EntryAbility'
  },
  maxScreenWantAgent: { // 全屏显示提醒到达时自动拉起的目标UIAbility信息
    pkgName: 'com.example.myapplication',
    abilityName: 'EntryAbility'
  },
  title: 'this is title', // 指明提醒标题
  content: 'this is content', // 指明提醒内容
  expiredContent: 'this reminder has expired', // 指明提醒过期后需要显示的内容
  notificationId: 100, // 指明提醒使用的通知的ID号，相同ID号的提醒会覆盖
  slotType: notificationManager.SlotType.SOCIAL_COMMUNICATION // 指明提醒的Slot类型
}
```

  - 定义日历实例。

```
let targetReminderAgent: reminderAgentManager.ReminderRequestCalendar = {
  reminderType: reminderAgentManager.ReminderType.REMINDER_TYPE_CALENDAR, // 提醒类型为日历类型
  dateTime: {   // 指明提醒的目标时间
    year: 2023,
    month: 1,
    day: 1,
    hour: 11,
    minute: 14,
    second: 30
  },
  repeatMonths: [1], // 指明重复提醒的月份
  repeatDays: [1], // 指明重复提醒的日期
  actionButton: [ // 设置弹出的提醒通知信息上显示的按钮类型和标题
    {
      title: 'close',
      type: reminderAgentManager.ActionButtonType.ACTION_BUTTON_TYPE_CLOSE
    },
    {
      title: 'snooze',
      type: reminderAgentManager.ActionButtonType.ACTION_BUTTON_TYPE_SNOOZE
    },
  ],
  wantAgent: { // 点击提醒通知后跳转的目标UIAbility信息
    pkgName: 'com.example.myapplication',
    abilityName: 'EntryAbility'
  },
  maxScreenWantAgent: { // 全屏显示提醒到达时自动拉起的目标UIAbility信息
    pkgName: 'com.example.myapplication',
    abilityName: 'EntryAbility'
  },
  ringDuration: 5, // 指明响铃时长（单位：秒）
  snoozeTimes: 2, // 指明延迟提醒次数
  timeInterval: 5*60, // 执行延迟提醒间隔（单位：秒）
  title: 'this is title', // 指明提醒标题
  content: 'this is content', // 指明提醒内容
  expiredContent: 'this reminder has expired', // 指明提醒过期后需要显示的内容
  snoozeContent: 'remind later', // 指明延迟提醒时需要显示的内容
  notificationId: 100, // 指明提醒使用的通知的ID号，相同ID号的提醒会覆盖
  slotType: notificationManager.SlotType.SOCIAL_COMMUNICATION // 指明提醒的Slot类型
}
```

# 开发步骤

- 定义闹钟实例。

```
let targetReminderAgent: reminderAgentManager.ReminderRequestAlarm = {
  reminderType: reminderAgentManager.ReminderType.REMINDER_TYPE_ALARM, // 提醒类型为闹钟类型
  hour: 23, // 指明提醒的目标时刻
  minute: 9, // 指明提醒的目标分钟
  daysOfWeek: [2], // 指明每周哪几天需要重复提醒
  actionButton: [ // 设置弹出的提醒通知信息上显示的按钮类型和标题
    {
      title: 'close',
      type: reminderAgentManager.ActionButtonType.ACTION_BUTTON_TYPE_CLOSE
    },
    {
      title: 'snooze',
      type: reminderAgentManager.ActionButtonType.ACTION_BUTTON_TYPE_SNOOZE
    },
  ],
  wantAgent: { // 点击提醒通知后跳转的目标UIAbility信息
    pkgName: 'com.example.myapplication',
    abilityName: 'EntryAbility'
  },
  maxScreenWantAgent: { // 全屏显示提醒到达时自动拉起的目标UIAbility信息
    pkgName: 'com.example.myapplication',
    abilityName: 'EntryAbility'
  },
  ringDuration: 5, // 指明响铃时长（单位：秒）
  snoozeTimes: 2, // 指明延迟提醒次数
  timeInterval: 5*60, // 执行延迟提醒间隔（单位：秒）
  title: 'this is title', // 指明提醒标题
  content: 'this is content', // 指明提醒内容
  expiredContent: 'this reminder has expired', // 指明提醒过期后需要显示的内容
  snoozeContent: 'remind later', // 指明延迟提醒时需要显示的内容
  notificationId: 99, // 指明提醒使用的通知的ID号，相同ID号的提醒会覆盖
  slotType: notificationManager.SlotType.SOCIAL_COMMUNICATION // 指明提醒的Slot类型
}
```

- 发布相应的提醒代理。代理发布后，应用即可使用后台代理提醒功能。

```
reminderAgentManager.publishReminder(targetReminderAgent).then((res: number) => {
  console.info('Succeeded in publishing reminder. ');
  let reminderId: number = res; // 发布的提醒ID
}).catch((err: BusinessError) => {
  console.error(`Failed to publish reminder. Code: ${err.code}, message: ${err.message}`);
})
```

- 根据需要删除提醒任务。

```
let reminderId: number = 1;
// reminderId的值从发布提醒代理成功之后的回调中获得
reminderAgentManager.cancelReminder(reminderId).then(() => {
  console.log('Succeeded in canceling reminder.');
}).catch((err: BusinessError) => {
  console.error(`Failed to cancel reminder. Code: ${err.code}, message: ${err.message}`);
});
```

# 实践案例：PedometerApp

# IPC Kit

# IPC

**基本概念**

IPC（Inter-Process Communication）与RPC（Remote Procedure Call）用于实现跨进程通信，不同的是前者使用Binder驱动，用于设备内的跨进程通信，后者使用软总线驱动，用于跨设备跨进程通信。需要跨进程通信的原因是因为每个进程都有自己独立的资源和内存空间，其他进程不能随意访问不同进程的内存和资源，IPC/RPC便是为了突破这一点。

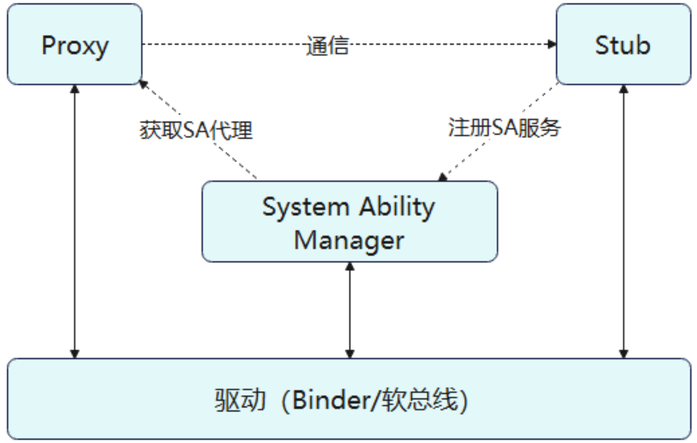
**说明**

Stage模型不能直接使用本文介绍的IPC和RPC，需要通过以下能力实现相关业务场景：

- IPC典型使用场景在后台服务，应用的后台服务通过IPC机制提供跨进程的服务调用能力。
- RPC典型使用场景在多端协同，多端协同通过RPC机制提供远端接口调用与数据传递能力。

**实现原理**

IPC和RPC通常采用客户端-服务器（Client-Server）模型，在使用时，请求服务的（Client）一端进程可获取提供服务（Server）一端所在进程的代理（Proxy），并通过此代理读写数据来实现进程间的数据通信，更具体的讲，首先请求服务的（Client）一端会建立一个服务提供端（Server）的代理对象，这个代理对象具备和服务提供端（Server）一样的功能，若想访问服务提供端（Server）中的某一个方法，只需访问代理对象中对应的方法即可，代理对象会将请求发送给服务提供端（Server）；然后服务提供端（Server）处理接受到的请求，处理完之后通过驱动返回处理结果给代理对象；最后代理对象将请求结果进一步返回给请求服务端（Client）。通常，Server会先注册系统能力（System Ability）到系统能力管理者（System Ability Manager，缩写SAMgr）中，SAMgr负责管理这些SA并向Client提供相关的接口。Client要和某个具体的SA通信，必须先从SAMgr中获取该SA的代理，然后使用代理和SA通信。下文直接使用Proxy表示服务请求方，Stub表示服务提供方。




**约束与限制**

- 单个设备上跨进程通信时，传输的数据量最大约为1MB，过大的数据量请使用匿名共享内存。
- 不支持在RPC中订阅匿名Stub对象（没有向SAMgr注册Stub对象）的死亡通知。
- 不支持把跨设备的Proxy对象传递回该Proxy对象所指向的Stub对象所在的设备，即指向远端设备Stub的Proxy对象不能在本设备内进行二次跨进程传递。

# IPC与RPC通信开发指导

### 场景介绍

IPC/RPC的主要工作是让运行在不同进程的Proxy和Stub互相通信，包括Proxy和Stub运行在不同设备的情况。

### 开发步骤

### ArkTS侧开发步骤

**说明**

- 此文档中的示例代码描述的是系统应用跨进程通信。
- 当前不支持三方应用实现ServiceExtensionAbility，三方应用的UIAbility组件可以通过Context连接系统提供的ServiceExtensionAbility。
- 当前使用场景： 仅限客户端是三方应用，服务端是系统应用。
- 添加依赖

```
// FA模型需要从@kit.AbilityKit导入featureAbility
// import { featureAbility } from '@kit.AbilityKit';
import { rpc } from '@kit.IPCKit';
```

# IPC与RPC通信开发指导

- 绑定Ability

  首先，构造变量want，指定要绑定的Ability所在应用的包名、组件名，如果是跨设备的场景，还需要绑定目标设备NetworkId（组网场景下对应设备的标识符，可以使用distributedDeviceManager获取目标设备的NetworkId）；然后，构造变量connect，指定绑定成功、绑定失败、断开连接时的回调函数；最后，FA模型使用featureAbility提供的接口绑定Ability，Stage模型通过context获取服务后用提供的接口绑定Ability。

```
// FA模型需要从@kit.AbilityKit导入featureAbility
// import { featureAbility } from "@kit.AbilityKit";
import { Want, common } from '@kit.AbilityKit';
import { rpc } from '@kit.IPCKit';
import { hilog } from '@kit.PerformanceAnalysisKit';
import { distributedDeviceManager } from '@kit.DistributedServiceKit';
import { BusinessError } from '@kit.BasicServicesKit';

let dmInstance: distributedDeviceManager.DeviceManager | undefined;
let proxy: rpc.IRemoteObject | undefined;
let connectId: number;

// 单个设备绑定Ability
let want: Want = {
  // 包名和组件名写实际的值
  bundleName: "ohos.rpc.test.server",
  abilityName: "ohos.rpc.test.server.ServiceAbility",
};
let connect: common.ConnectOptions = {
  onConnect: (elementName, remoteProxy) => {
    hilog.info(0x0000, 'testTag', 'RpcClient: js onConnect called');
    proxy = remoteProxy;
  },
  onDisconnect: (elementName) => {
    hilog.info(0x0000, 'testTag', 'RpcClient: onDisconnect');
  },
  onFailed: () => {
    hilog.info(0x0000, 'testTag', 'RpcClient: onFailed');
  }
};

let context: common.UIAbilityContext = getContext(this) as common.UIAbilityContext; // UIAbilityContext
// 建立连接后返回的Id需要保存下来，在解绑服务时需要作为参数传入
connectId = context.connectServiceExtensionAbility(want,connect);

// 跨设备绑定
try{
  dmInstance = distributedDeviceManager.createDeviceManager("ohos.rpc.test");
} catch(error) {
  let err: BusinessError = error as BusinessError;
  hilog.error(0x0000, 'testTag', 'createDeviceManager errCode:' + err.code + ', errMessage:' + err.message);
}

// 使用distributedDeviceManager获取目标设备NetworkId
if (dmInstance != undefined) {
  let deviceList = dmInstance.getAvailableDeviceListSync();
  let networkId = deviceList[0].networkId;
  let want: Want = {
    bundleName: "ohos.rpc.test.server",
    abilityName: "ohos.rpc.test.service.ServiceAbility",
    deviceId: networkId,
    flags: 256
  };
  // 建立连接后返回的Id需要保存下来，在断开连接时需要作为参数传入

  // 第一个参数是本应用的包名，第二个参数是接收distributedDeviceManager的回调函数
  connectId = context.connectServiceExtensionAbility(want,connect);
}
```

# IPC与RPC通信开发指导

- 服务端处理客户端请求
  服务端被绑定的Ability在onConnect方法里返回继承自rpc.RemoteObject的对象，该对象需要实现onRemoteMessageRequest方法，处理客户端的请求。

```
import { rpc } from '@kit.IPCKit';
import { Want } from '@kit.AbilityKit';
class Stub extends rpc.RemoteObject {
  constructor(descriptor: string) {
    super(descriptor);
  }
  onRemoteMessageRequest(code: number, data: rpc.MessageSequence, reply: rpc.MessageSequence, option: rpc.MessageOption): boolean | Promise<boolean> {
    // 根据code处理客户端的请求
    return true;
  }

  onConnect(want: Want) {
    const robj: rpc.RemoteObject = new Stub("rpcTestAbility");
    return robj;
  }
}
```

# IPC与RPC通信开发指导

- 客户端处理服务端响应
  客户端在onConnect回调里接收到代理对象，调用sendMessageRequest方法发起请求，在期约（用于表示一个异步操作的最终完成或失败及其结果值）或者回调函数里接收结果。

```
import { rpc } from '@kit.IPCKit';
import { hilog } from '@kit.PerformanceAnalysisKit';

// 使用期约
let option = new rpc.MessageOption();
let data = rpc.MessageSequence.create();
let reply = rpc.MessageSequence.create();
// 往data里写入参数
let proxy: rpc.IRemoteObject | undefined;
if (proxy != undefined) {
  proxy.sendMessageRequest(1, data, reply, option)
    .then((result: rpc.RequestResult) => {
      if (result.errCode != 0) {
        hilog.error(0x0000, 'testTag', 'sendMessageRequest failed, errCode: ' + result.errCode);
        return;
      }
      // 从result.reply里读取结果
    })
    .catch((e: Error) => {
      hilog.error(0x0000, 'testTag', 'sendMessageRequest got exception: ' + e);
    })
    .finally(() => {
      data.reclaim();
      reply.reclaim();
    })
}

// 使用回调函数
function sendRequestCallback(err: Error, result: rpc.RequestResult) {
  try {
    if (result.errCode != 0) {
      hilog.error(0x0000, 'testTag', 'sendMessageRequest failed, errCode: ' + result.errCode);
      return;
    }
    // 从result.reply里读取结果
  } finally {
    result.data.reclaim();
    result.reply.reclaim();
  }
}
let options = new rpc.MessageOption();
let datas = rpc.MessageSequence.create();
let replys = rpc.MessageSequence.create();
// 往data里写入参数
if (proxy != undefined) {
  proxy.sendMessageRequest(1, datas, replys, options, sendRequestCallback);
}
```

# IPC与RPC通信开发指导

- 断开连接
  IPC通信结束后，FA模型使用featureAbility的接口断开连接，Stage模型在获取context后用提供的接口断开连接。

```
// FA模型需要从@kit.AbilityKit导入featureAbility
// import { featureAbility } from "@kit.AbilityKit";
import { Want, common } from '@kit.AbilityKit';
import { rpc } from '@kit.IPCKit';
import { hilog } from '@kit.PerformanceAnalysisKit';

function disconnectCallback() {
  hilog.info(0x0000, 'testTag', 'disconnect ability done');
}
// FA模型使用此方法断开连接
// featureAbility.disconnectAbility(connectId, disconnectCallback);

let proxy: rpc.IRemoteObject | undefined;
let connectId: number;

// 单个设备绑定Ability
let want: Want = {
  // 包名和组件名写实际的值
  bundleName: "ohos.rpc.test.server",
  abilityName: "ohos.rpc.test.server.ServiceAbility",
};
let connect: common.ConnectOptions = {
  onConnect: (elementName, remote) => {
    proxy = remote;
  },
  onDisconnect: (elementName) => {
  },
  onFailed: () => {
    proxy;
  }
};
// FA模型使用此方法连接服务
// connectId = featureAbility.connectAbility(want, connect);

connectId = this.context.connectServiceExtensionAbility(want,connect);


this.context.disconnectServiceExtensionAbility(connectId);
```