《移动应用开发》课程指导 v.0.1.0

To be continue... 2025-04-10

本课程指导会增量完成，最终作为整个课程的回顾指导

Part One: Stage模型

从一个import开始

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

UIAbility

UIAbility是包含UI界面的应用组件，继承自Ability。

- 提供组件生命周期回调
  - 创建
  - 销毁
  - 前后台切换等
- 具备组件协同的能力
  - Caller
    - 由startAbilityByCall接口返回，CallerAbility(调用者)可使用Caller与CalleeAbility(被调用者)进行通信。
  - Callee
    - UIAbility的内部对象，CalleeAbility(被调用者)可以通过Callee与Caller进行通信。

类属性

- *context*: **UIAbilityContext**, 上下文
- *launchWant*: **Want**, UIAbility启动时的参数
- *lastRequestWant*: **Want**, UIAbility最后请求时的参数
- *callee*: **Callee**, 调用Stub（桩）服务对象

类方法

- *onCreate(want: Want, launchParam: AbilityConstant.LaunchParam)*: **void**
  - UIAbility实例处于完全关闭状态下被创建完成后进入该生命周期回调，执行初始化业务逻辑操作。即UIAbility实例冷启动时进入该生命周期回调。***同步接口，不支持异步回调。***
- *onDestroy()*: **void | Promise<void>**
  - UIAbility生命周期回调，在销毁时回调，执行资源清理等操作。***使用同步回调或Promise异步回调。***
- *onWindowStageCreate(windowStage: window.WindowStage)*: **void**
  - 当WindowStage创建后调用。

- *onWindowStageDestroy()*: **void**
  - 当WindowStage销毁后调用。
- *onForeground()*: **void**
  - UIAbility生命周期回调，当应用从后台转到前台时触发。***同步接口，不支持异步回调。***
- *onBackground()*: **void**
  - UIAbility生命周期回调，当应用从前台转到后台时触发。***同步接口，不支持异步回调。***

其他重要辅助类

- 外部类
  - Caller
    - 通用组件Caller通信客户端调用接口, 用来向通用组件服务端发送约定数据。
  - Callee
    - 通用组件服务端注册和解除客户端caller通知送信的callback接口。

用法示例

系统默认的***EntryAbility.ets***。

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

const DOMAIN = 0x0000;

export default class EntryAbility extends UIAbility {
  onCreate(want: Want, launchParam: AbilityConstant.LaunchParam): void {

 this.context.getApplicationContext().setColorMode(ConfigurationConstant.ColorMod
e.COLOR_MODE_NOT_SET);
    hilog.info(DOMAIN, 'testTag', '%{public}s', 'Ability onCreate');
  }

  onDestroy(): void {
    hilog.info(DOMAIN, 'testTag', '%{public}s', 'Ability onDestroy');
  }

  onWindowStageCreate(windowStage: window.WindowStage): void {
    // Main window is created, set main page for this ability
    hilog.info(DOMAIN, 'testTag', '%{public}s', 'Ability onWindowStageCreate');

    windowStage.loadContent('pages/Index', (err) => {
      if (err.code) {
        hilog.error(DOMAIN, 'testTag', 'Failed to load the content. Cause: %
{public}s', JSON.stringify(err));
        return;
      }
      hilog.info(DOMAIN, 'testTag', 'Succeeded in loading the content.');
    });
  }
```

```
  onWindowStageDestroy(): void {
    // Main window is destroyed, release UI related resources
    hilog.info(DOMAIN, 'testTag', '%{public}s', 'Ability onWindowStageDestroy');
  }

  onForeground(): void {
    // Ability has brought to foreground
    hilog.info(DOMAIN, 'testTag', '%{public}s', 'Ability onForeground');
  }

  onBackground(): void {
    // Ability has back to background
    hilog.info(DOMAIN, 'testTag', '%{public}s', 'Ability onBackground');
  }
}
```

UIAbility作为一个组件

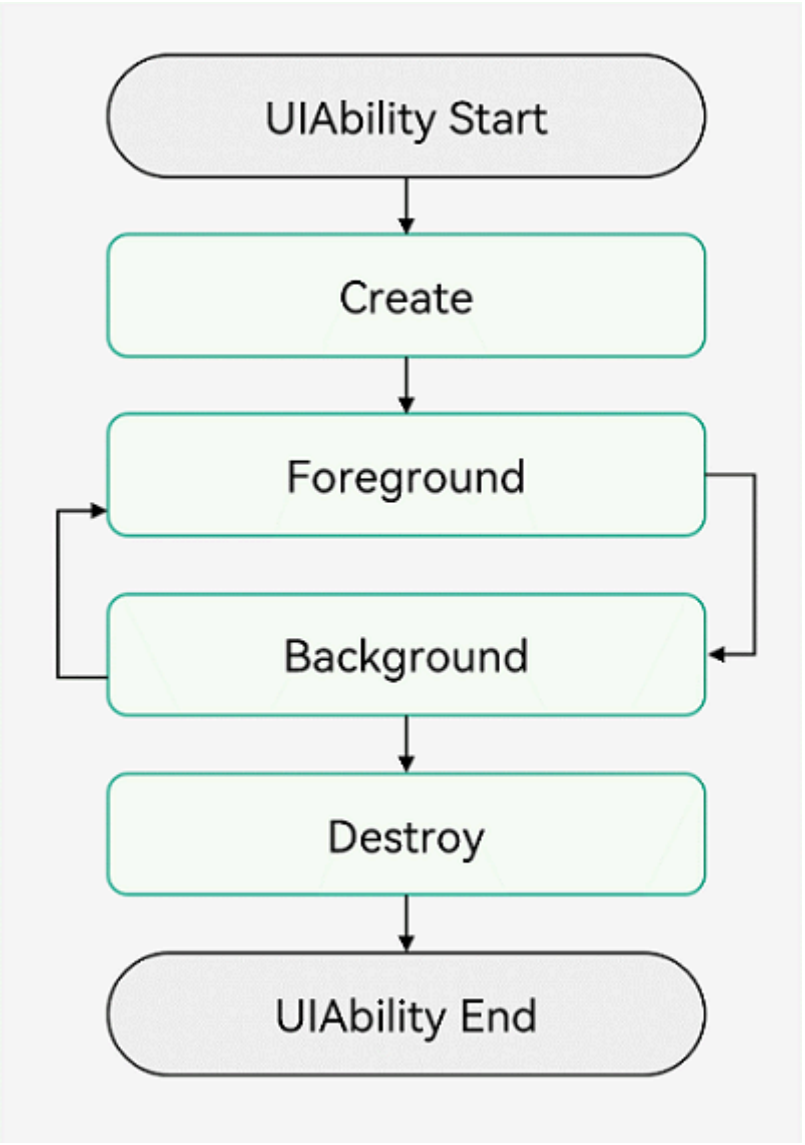
UIAbility组件是系统调度的基本单元，为应用提供绘制界面的窗口。一个应用可以包含一个或多个UIAbility组件。例如，在我们的应用中，可以***将登录/注册功能和主功能分别配置为独立的UIAbility***。

声明配置

在module.json5配置文件的abilities标签中声明UIAbility的名称、入口、标签等相关信息。

```
{
  "module": {
    // ...
    "abilities": [
      {
        "name": "EntryAbility", // UIAbility组件的名称
        "srcEntry": "./ets/entryability/EntryAbility.ets", // UIAbility组件的代码路径
        "description": "$string:EntryAbility_desc", // UIAbility组件的描述信息
        "icon": "$media:icon", // UIAbility组件的图标
        "label": "$string:EntryAbility_label", // UIAbility组件的标签
        "startWindowIcon": "$media:icon", // UIAbility组件启动页面图标资源文件的索引
        "startWindowBackground": "$color:start_window_background", // UIAbility组件启动页面背景颜色资源文件的索引
        // ...
      }
    ]
  }
}
```

生命周期状态说明

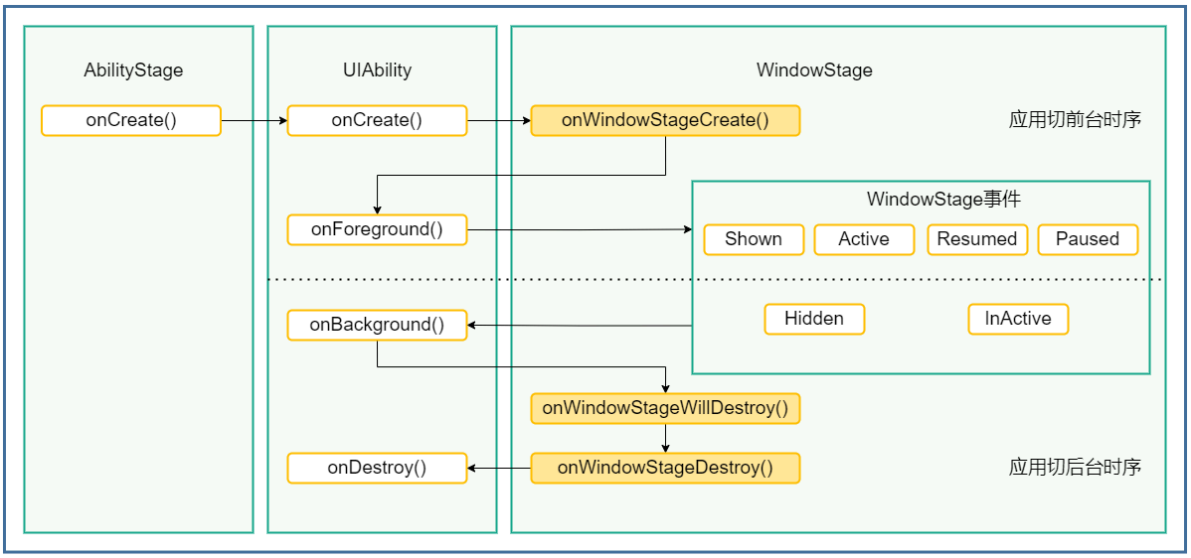


Create状态

Create状态为在应用加载过程中，UIAbility实例创建完成时触发，系统会调用onCreate()回调。可以在该回调中进行页面初始化操作，例如变量定义资源加载等，用于后续的UI展示。

WindowStageCreate和WindowStageDestroy状态

UIAbility实例创建完成之后，在进入Foreground之前，系统会创建一个WindowStage。WindowStage创建完成后会进入onWindowStageCreate()回调，可以在该回调中设置UI加载、设置WindowStage的事件订阅。



在onWindowStageCreate()回调中通过loadContent()方法设置应用要加载的页面，并根据需要调用on('windowStageEvent')方法订阅WindowStage的事件（获焦/失焦、切到前台/切到后台、前台可交互/前台不可交互）。

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

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

export default class EntryAbility extends UIAbility {
  // ...
  onWindowStageCreate(windowStage: window.WindowStage): void {
    // 设置WindowStage的事件订阅（获焦/失焦、切到前台/切到后台、前台可交互/前台不可交互）
    try {
      windowStage.on('windowStageEvent', (data) => {
        let stageEventType: window.WindowStageEventType = data;
        switch (stageEventType) {
          case window.WindowStageEventType.SHOWN: // 切到前台
            hilog.info(DOMAIN_NUMBER, TAG, `windowStage foreground.`);
            break;
          case window.WindowStageEventType.ACTIVE: // 获焦状态
            hilog.info(DOMAIN_NUMBER, TAG, `windowStage active.`);
            break;
          case window.WindowStageEventType.INACTIVE: // 失焦状态
            hilog.info(DOMAIN_NUMBER, TAG, `windowStage inactive.`);
            break;
          case window.WindowStageEventType.HIDDEN: // 切到后台
            hilog.info(DOMAIN_NUMBER, TAG, `windowStage background.`);
            break;
          case window.WindowStageEventType.RESUMED: // 前台可交互状态
            hilog.info(DOMAIN_NUMBER, TAG, `windowStage resumed.`);
            break;
          case window.WindowStageEventType.PAUSED: // 前台不可交互状态
            hilog.info(DOMAIN_NUMBER, TAG, `windowStage paused.`);
            break;
          default:
            break;
```

```
        }
      });
    } catch (exception) {
      hilog.error(DOMAIN_NUMBER, TAG,
        `Failed to enable the listener for window stage event changes. Cause:
${JSON.stringify(exception)}`);
    }
    hilog.info(DOMAIN_NUMBER, TAG, `%{public}s`, `Ability onWindowStageCreate`);
    // 设置UI加载
    windowStage.loadContent('pages/Index', (err, data) => {
      // ...
    });
  }
}
```

Foreground和Background状态

Foreground和Background状态分别在UIAbility实例切换至前台和切换至后台时触发，对应于onForeground()回调和onBackground()回调。

onForeground()回调，在UIAbility的UI可见之前，如UIAbility切换至前台时触发。可以在onForeground()回调中申请系统需要的资源，或者重新申请在onBackground()中释放的资源。

onBackground()回调，在UIAbility的UI完全不可见之后，如UIAbility切换至后台时候触发。可以在onBackground()回调中释放UI不可见时无用的资源，或者在此回调中执行较为耗时的操作，例如状态保存等。

例如应用在使用过程中需要使用用户定位时，假设应用已获得用户的定位权限授权。在UI显示之前，可以在onForeground()回调中开启定位功能，从而获取到当前的位置信息。

当应用切换到后台状态，可以在onBackground()回调中停止定位功能，以节省系统的资源消耗。

Destroy状态

Destroy状态在UIAbility实例销毁时触发。可以在onDestroy()回调中进行系统资源的释放、数据的保存等操作。

UIAbility组件启动模式

UIAbility的启动模式是指UIAbility实例在启动时的不同呈现状态。针对不同的业务场景，系统提供了三种启动模式：

- singleton（单实例模式）
- multiton（多实例模式）
- specified（指定实例模式）（略，理解概念即可）

singleton启动模式

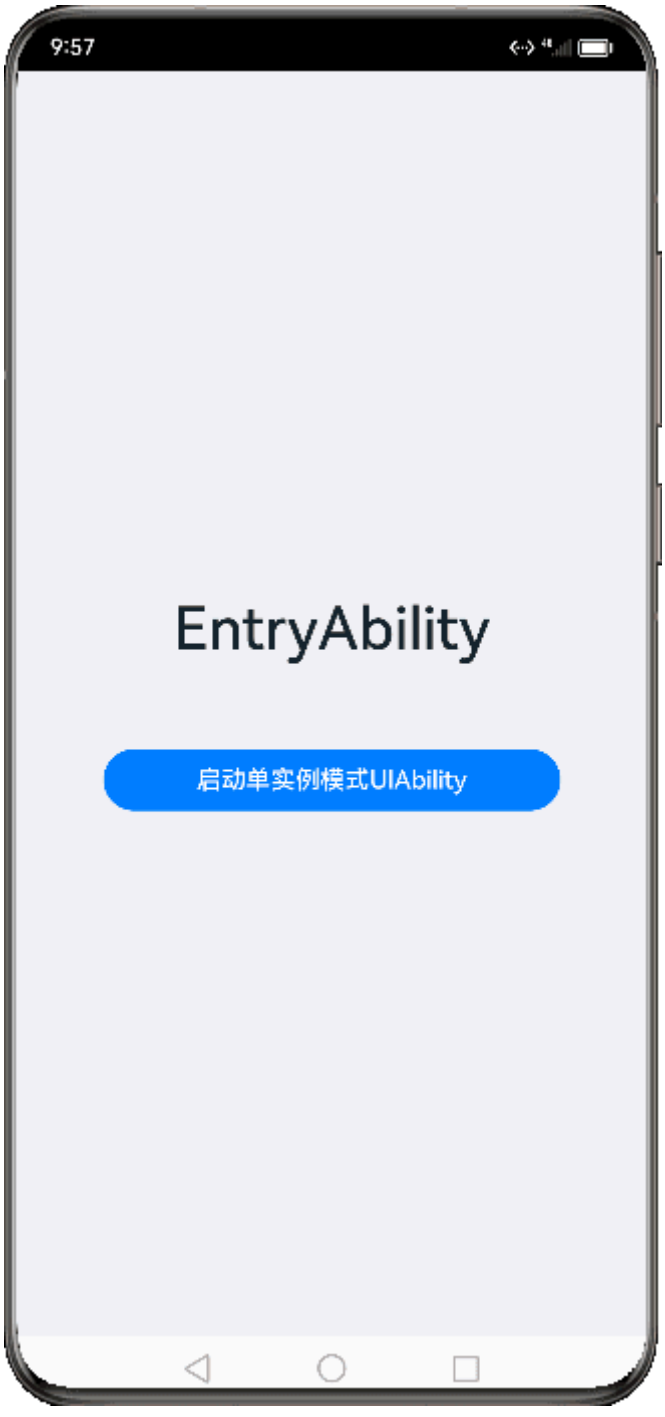
singleton启动模式为单实例模式，也是默认情况下的启动模式。

应用的UIAbility实例已创建，该UIAbility配置为单实例模式，再次调用startAbility()方法启动该UIAbility实例。由于启动的还是原来的UIAbility实例，并未重新创建一个新的UIAbility实例，此时只会进入该UIAbility的onNewWant()回调，不会进入其onCreate()和onWindowStageCreate()生命周期回调。

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

export default class EntryAbility extends UIAbility {
  // ...

  onNewWant(want: Want, launchParam: AbilityConstant.LaunchParam) {
    // 更新资源、数据
  }
}
```

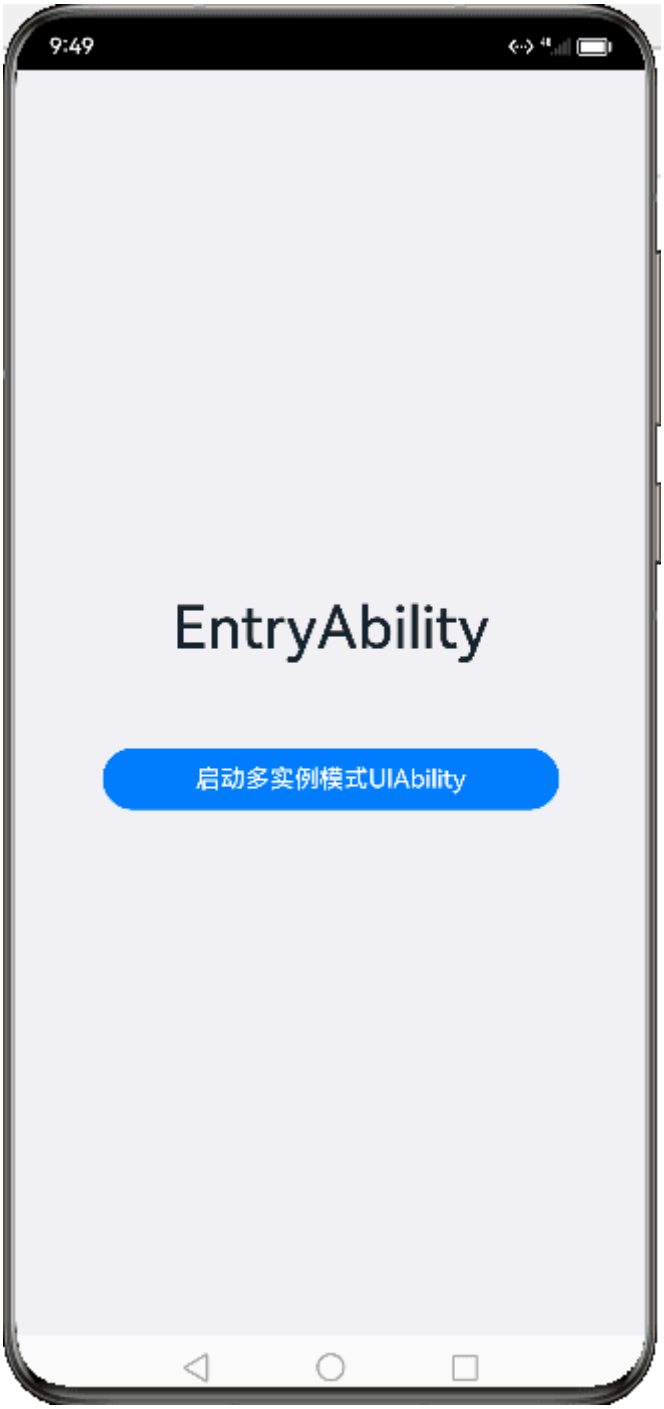


如果需要使用singleton启动模式，在module.json5配置文件中的launchType字段配置为singleton即可。

```
{
  "module": {
    // ...
    "abilities": [
      {
        "launchType": "singleton",
        // ...
      }
    ]
  }
}
```

multiton启动模式

multiton启动模式为多实例模式，每次调用startAbility()方法时，都会在应用进程中创建一个新的该类型UIAbility实例。即在最近任务列表中可以看到有多个该类型的UIAbility实例。这种情况下可以将UIAbility配置为multiton（多实例模式）。



multiton启动模式的开发使用，在module.json5配置文件中的launchType字段配置为multiton即可。

```
{
  "module": {
    // ...
    "abilities": [
      {
        "launchType": "multiton",
        // ...
      }
    ]
  }
}
```

UIAbility组件基本用法

UIAbility组件的基本用法包括：

- 指定UIAbility的启动页面
- 获取UIAbility的上下文UIAbilityContext

指定UIAbility的启动页面

应用中的UIAbility在启动过程中，需要指定启动页面，否则应用启动后会因为没有默认加载页面而导致白屏。可以在UIAbility的onWindowStageCreate()生命周期回调中，通过WindowStage对象的loadContent()方法设置启动页面。

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

export default class EntryAbility extends UIAbility {
  onWindowStageCreate(windowStage: window.WindowStage): void {
    // Main window is created, set main page for this ability
    windowStage.loadContent('pages/Index', (err, data) => {
      // ...
    });
  }
  // ...
}
```

获取UIAbility的上下文信息

UIAbility类拥有自身的上下文信息，该信息为UIAbilityContext类的实例，UIAbilityContext类拥有abilityInfo、currentHapModuleInfo等属性。通过UIAbilityContext可以获取UIAbility的相关配置信息，如包代码路径、Bundle名称、Ability名称和应用程序需要的环境状态等属性信息，以及可以获取操作UIAbility实例的方法（如startAbility()、connectServiceExtensionAbility()、terminateSelf()等）。

如果需要在页面中获得当前Ability的Context，可调用getContext接口获取当前页面关联的UIAbilityContext或ExtensionContext。

- 在UIAbility中可以通过this.context获取UIAbility实例的上下文信息

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

export default class EntryAbility extends UIAbility {
  onCreate(want: Want, launchParam: AbilityConstant.LaunchParam): void {
    // 获取UIAbility实例的上下文
    let context = this.context;
    // ...
  }
}
```

- 在页面中获取UIAbility实例的上下文信息，包括导入依赖资源context模块和在组件中定义一个context变量两个部分

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

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

  startAbilityTest(): void {
    let want: Want = {
      // Want参数信息
    };
    this.context.startAbility(want);
  }

  // 页面展示
  build() {
    // ...
  }
}
```

- 当业务完成后，如果想要终止当前UIAbility实例，可以通过调用terminateSelf()方法实现

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

@Entry
@Component
struct Page_UIAbilityComponentsBasicUsage {
  // 页面展示
  build() {
    Column() {
      //...
      Button('FuncAbilityB')
        .onClick(() => {
          let context = getContext(this) as common.UIAbilityContext;
          try {
            context.terminateSelf((err: BusinessError) => {
              if (err.code) {
                // 处理业务逻辑错误
                console.error(`terminateSelf failed, code is ${err.code}, message
is ${err.message}`);
```

```
                return;
              }
              // 执行正常业务
              console.info('terminateSelf succeed');
            });
          } catch (err) {
            // 捕获同步的参数错误
            let code = (err as BusinessError).code;
            let message = (err as BusinessError).message;
            console.error(`terminateSelf failed, code is ${code}, message is
${message}`);
          }
        })
    }
  }
}
```

Want

Want是对象间信息传递的载体，可以用于**应用组件间的信息传递**。

Want的使用场景之一是**作为startAbility的参数**，其包含了指定的启动目标，以及启动时需携带的相关数据，例如：

- bundleName和abilityName字段分别指明目标Ability所在应用的包名以及对应包内的Ability名称。当UIAbility A需要启动UIAbility B并传入一些数据时，可使用Want作为载体将这些数据传递给UIAbility B。

类属性

- *deviceId*: **string**, 表示运行指定Ability的设备ID。如果未设置该字段，则表明指定本设备。
- *bundleName*: **string**, 表示待启动Ability所在的应用Bundle名称。
- *moduleName*: **string**, 表示待启动的Ability所属的模块名称。
- *abilityName*: **string**, 表示待启动Ability名称。如果在Want中该字段同时指定了BundleName和AbilityName，则Want可以直接匹配到指定的Ability。AbilityName需要在一个应用的范围内保证唯一。
- *action*: **string**, 表示要执行的通用操作（如：查看、分享、应用详情）。在隐式Want中，您可以定义该字段，配合uri或parameters来表示对数据要执行的操作。
- *entities*: **Array<string>**, 表示目标Ability额外的类别信息（如：浏览器、视频播放器）。在隐式Want中是对action字段的补充。
- *uri*: **string**, 表示携带的数据，一般配合type使用，指明待处理的数据类型。
- *type*: **string**, 表示MIME type类型描述，打开文件的类型，主要用于文管打开文件。比如：'text/xml' 、 'image/*'等。
- *parameters*: **Record<string, Object>**, 表示WantParams描述。
- *flags*: **number**, 表示处理Want的方式。默认传数字。

用法示例

在***EntryAbility.ets***中，我们已看到在UIAbility中可以通过this.context获取UIAbility实例的上下文信息。下面演示如何在页面中获取UIAbility实例的上下文信息。

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

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

  startAbilityTest(): void {
    let want: Want = {
        deviceId: '', // deviceId为空表示本设备
        bundleName: 'com.example.myapplication',
        abilityName: 'UIAbilityB',
        moduleName: 'entry' // moduleName非必选
        parameters: {
            'keyForString': 'str',
            'keyForInt': 100,
            'keyForDouble': 99.99,
            'keyForBool': true,
            'keyForObject': {
                'keyForObjectString': 'str',
                'keyForObjectInt': -200,
                'keyForObjectDouble': 35.5,
                'keyForObjectBool': false,
            },
            'keyForArrayString': ['str1', 'str2', 'str3'],
            'keyForArrayInt': [100, 200, 300, 400],
            'keyForArrayDouble': [0.1, 0.2],
            'keyForArrayObject': [{ obj1: 'aaa' }, { obj2: 100 }],
            'keyFd': { 'type': 'FD', 'value': fd } // {'type':'FD', 'value':fd}是
固定用法，用于表示该数据是FD
        },
    };

    context.startAbility(want, (err: BusinessError) => {
        if (err.code) {
            // 显式拉起Ability，通过bundleName、abilityName和moduleName可以唯一确定一个
Ability
            console.error(`Failed to startAbility. Code: ${err.code}, message:
${err.message}`);
        }
    });
  }

  // 页面展示
  build() {
    // ...
  }
}
```

在UIAbilityB中接受从UIAbilityA传来的Want:

```
// 以UIAbilityB实例首次启动为例，会进入到UIAbilityB的onCreate生命周期
import { UIAbility, Want, AbilityConstant } from '@kit.AbilityKit';

class UIAbilityB extends UIAbility {
  onCreate(want: Want, launchParam: AbilityConstant.LaunchParam) {
    console.log(`onCreate, want parameters:
${want.parameters?.developerParameters}`);
  }
}
```

- *context*: **UIAbilityContext**, 上下文
- *launchWant*: **Want**, UIAbility启动时的参数
- *lastRequestWant*: **Want**, UIAbility最后请求时的参数
- *callee*: **Callee**, 调用Stub（桩）服务对象

Part 2: 关系型数据库（Relational Database, RDB)

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

- 关系型数据库基于SQLite组件提供了一套完整的对本地数据库进行管理的机制
- 对外提供了一系列的增、删、改、查等接口
- 也可以直接运行用户输入的SQL语句来满足复杂的场景需要

2. SQLite的语法：

- https://sqlite.org/lang.html
- https://www.runoob.com/sqlite/sqlite-syntax.html

注意：

> 为保证插入并读取数据成功，建议一条数据***不要超过2M***。超出该大小，插入成功，读取失败。

3. 关系型数据库相关的常用功能（类）：

- **RdbPredicates**： 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。
- **RdbStore**：提供管理关系数据库(RDB)方法的接口。
- **ResultSet**：提供用户调用关系型数据库查询接口之后返回的结果集合。
- **Transaction**：提供管理事务对象的接口。

4. 模块导入

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

2.1 *relationalStore*

类方法

getRdbStore

> getRdbStore(context: Context, config: StoreConfig, callback: AsyncCallback): void

获得一个相关的RdbStore，操作关系型数据库，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，使用callback异步回调。

加密参数encrypt只在首次创建数据库时生效，因此在创建数据库时，选择正确的加密参数非常重要，并且在之后无法更改加密参数。

参数说明

- *context*: **Context**，上下文环境。（注意和FA模型的Context做区分）
- *config*: **StoreConfig**，数据库配置。
- *callback*: **AsyncCallback<RdbStore>**，异步回调。

> getRdbStore(context: Context, config: StoreConfig): Promise

获得一个相关的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，使用Promise异步回调。

加密参数encrypt只在首次创建数据库时生效，因此在创建数据库时，选择正确的加密参数非常重要，并且在之后无法更改加密参数。

参数说明

- *返回值*：**Promise<RdbStore>**，返回一个Promise对象，异步返回RdbStore。

deleteRdbStore

> deleteRdbStore(context: Context, name: string, callback: AsyncCallback): void

删除数据库文件，使用callback异步回调。

参数说明

- *name*: **string**，数据库名称。

> deleteRdbStore(context: Context, name: string): Promise

使用指定的数据库文件配置删除数据库，使用Promise异步回调。

> deleteRdbStore(context: Context, config: StoreConfig, callback: AsyncCallback): void

使用指定的数据库文件配置删除数据库，使用callback异步回调。

> deleteRdbStore(context: Context, config: StoreConfig): Promise

使用指定的数据库文件配置删除数据库，使用Promise异步回调。

重要辅助类

StoreConfig

管理关系数据库配置。

类属性

- *name*: **string**，数据库名称。
- *securityLevel*: **SecurityLevel**，数据库安全级别。
- *encrypt*: **boolean**，是否加密，默认不加密。
- *customDir*: **string**，自定义数据库路径。
- *autoCleanDirtyData*: **boolean**，指定是否自动清理云端删除后同步到本地的数据，默认自动清理。
- *allowRebuild*: **boolean**，指定数据库是否支持异常时自动删除，并重建一个空库空表，默认不删除。
- *isReadOnly*: **boolean**，指定数据库是否只读，默认可读写。
- *cryptoParam*: **CryptoParam**，数据库加密参数。

SecurityLevel

数据库的安全级别枚举。请使用枚举名称而非枚举值。数据库的安全等级仅支持由低向高设置，不支持由高向低设置。

枚举名称

- *S1*: 表示数据库的安全级别为低级别，当数据泄露时会产生较低影响。例如，包含壁纸等系统数据的数据库。
- *S2*: 表示数据库的安全级别为中级别，当数据泄露时会产生较大影响。例如，包含录音、视频等用户生成数据或通话记录等信息的数据库。
- *S3*: 表示数据库的安全级别为高级别，当数据泄露时会产生重大影响。例如，包含用户运动、健康、位置等信息的数据库。
- *S4*: 表示数据库的安全级别为关键级别，当数据泄露时会产生严重影响。例如，包含认证凭据、财务数据等信息的数据库。

CryptoParam

数据库加密参数配置。此配置只有在StoreConfig的encrypt选项设置为真时才有效。

类属性

- *encryptionKey*: **Uint8Array**，指定数据库加/解密使用的密钥。
- *iterationCount*: **number**，整数类型，指定数据库PBKDF2算法的迭代次数，默认值为10000。
- *encryptionAlgo*： **EncryptionAlgo**，指定数据库加解密使用的加密算法。如不指定，默认值为AES_256_GCM。
- *hmacAlgo*： **HmacAlgo**， 指定数据库加解密使用的HMAC算法。如不指定，默认值为SHA256。
- *kdfAlgo*: **KdfAlgo**，指定数据库加解密使用的PBKDF2算法。如不指定，默认使用和HMAC算法相等的算法。
- *cryptoPageSize*: number，整数类型，指定数据库加解密使用的页大小。如不指定，默认值为1024字节。

ValueType

> type ValueType = null | number | string | boolean | Uint8Array | Asset | Assets | Float32Array | bigint

用于表示允许的数据字段类型，接口参数具体类型根据其功能而定。

ValuesBucket

> type ValuesBucket = Record<string, ValueType>

用于存储键值对的类型。

辅助类型

> Record<string, ValueType>

表示键值对类型。键的类型为string，值的类型为ValueType。

PRIKeyType

> type PRIKeyType = number | string

用于表示数据库表某一行主键的数据类型。

SyncMode

指数据库同步模式。请使用枚举名称而非枚举值。

枚举名称

- *SYNC_MODE_PUSH*: 表示数据从本地设备推送到远程设备。
- *SYNC_MODE_PULL*: 表示数据从远程设备拉至本地设备。
- *SYNC_MODE_TIME_FIRST*: 表示数据从修改时间较近的一端同步到修改时间较远的一端。
- *SYNC_MODE_NATIVE_FIRST*: 表示数据从本地设备同步到云端。
- *SYNC_MODE_CLOUD_FIRST*: 表示数据从云端同步到本地设备。

相关类型

SubscribeType

描述订阅类型。请使用枚举名称而非枚举值。

DistributedType

描述表的分布式类型的枚举。请使用枚举名称而非枚举值。

Origin

表示数据来源。请使用枚举名称而非枚举值。

DistributedConfig

记录表的分布式配置信息。

SqlExecutionInfo

描述数据库执行的SQL语句的统计信息。

类属性

- *sql*: **string**，执行的SQL语句。
- *sql*: **Array<string>**，表示执行的SQL语句的数组。当batchInsert的参数太大时，可能有多个SQL。
- *totalTime*: **number**，表示执行SQL语句的总时间，单位为μs。
- *waitTime: number**，表示获取句柄的时间，单位为μs。
- *prepareTime*: **number**，表示准备SQL和绑定参数的时间，单位为μs。
- *executeTime*: **number**，表示执行SQL语句的时间，单位为μs。

2.2 *RdbPredicates*

表示关系型数据库（RDB）的谓词。该类确定RDB中条件表达式的值是true还是false。谓词间支持多语句拼接，拼接时默认使用and()连接。

类方法

equalTo

> equalTo(field: string, value: ValueType): RdbPredicates

配置谓词以匹配数据表的field列中值为value的字段。

参数说明

- *field*: **string**，数据表的字段（列）名。
- *value*: **ValueType**，指示要与谓词匹配的值。
- *返回值*：**RdbPredicates**，返回与指定字段匹配的谓词。

notEqualTo

> notEqualTo(field: string, value: ValueType): RdbPredicates

配置谓词以匹配数据表的field列中值不为value的字段。

contains

> contains(field: string, value: string): RdbPredicates

配置谓词以匹配数据表的field列中包含value的字段。

beginsWith

> beginsWith(field: string, value: string): RdbPredicates

配置谓词以匹配数据表的field列中以value开头的字段。

endsWith

> endsWith(field: string, value: string): RdbPredicates

配置谓词以匹配数据表的field列中以value结尾的字段。

isNull

> isNull(field: string): RdbPredicates

配置谓词以匹配数据表的field列中值为null的字段。

isNotNull

> isNotNull(field: string): RdbPredicates

配置谓词以匹配数据表的field列中值不为null的字段。

like

> like(field: string, value: string): RdbPredicates

配置谓词以匹配数据表的field列中值类似于value的字段。

glob

> glob(field: string, value: string): RdbPredicates

配置谓词匹配数据字段为string的指定字段。

between

> between(field: string, low: ValueType, high: ValueType): RdbPredicates

配置谓词以匹配数据表的field列中值在给定范围内的字段（包含范围边界）。

notBetween

> notBetween(field: string, low: ValueType, high: ValueType): RdbPredicates

配置谓词以匹配数据表的field列中值超出给定范围的字段（不包含范围边界）。

greaterThan

> greaterThan(field: string, value: ValueType): RdbPredicates

配置谓词以匹配数据表的field列中值大于value的字段。

lessThan

> lessThan(field: string, value: ValueType): RdbPredicates

配置谓词以匹配数据表的field列中值小于value的字段。

orderByAsc

> orderByAsc(field: string): RdbPredicates

配置谓词以匹配数据表的field列中值按升序排序的列。

orderByDesc

> orderByDesc(field: string): RdbPredicates

配置谓词以匹配数据表的field列中值按降序排序的列。

distinct

> distinct(): RdbPredicates

配置谓词以过滤重复记录并仅保留其中一个。

limitAs

> limitAs(value: number): RdbPredicates

设置最大数据记录数的谓词。

offsetAs

> offsetAs(rowOffset: number): RdbPredicates

配置谓词以指定返回结果的起始位置。

groupBy

> groupBy(fields: Array): RdbPredicates

配置谓词按指定列分组查询结果。

indexedBy

> indexedBy(field: string): RdbPredicates

配置谓词以指定索引列。

in

> in(field: string, value: Array): RdbPredicates

配置谓词以匹配数据表的field列中值在给定范围内的字段。

beginWrap

> beginWrap(): RdbPredicates

向谓词添加左括号。

endWrap

> endWrap(): RdbPredicates

向谓词添加右括号。

or

> or(): RdbPredicates

将或条件添加到谓词中。

and

> and(): RdbPredicates

向谓词添加和条件。

2.3 *RdbStore*

提供管理关系数据库(RDB)方法的接口。

在使用以下相关接口前，请使用executeSql接口初始化数据库表结构和相关数据。

类方法

executeSql

> executeSql(sql: string, callback: AsyncCallback):void

执行包含指定参数但不返回值的SQL语句，语句中的各种表达式和操作符之间的关系操作符号不超过1000个，使用callback异步回调。

此接口不支持执行查询、附加数据库和事务操作，可以使用querySql、query、attach、beginTransaction、commit等接口代替。

不支持分号分隔的多条语句。

参数说明

- *sql*: **string**，指定要执行的SQL语句。
- *callback*: **AsyncCallback<void>**，指定callback回调函数。

> executeSql(sql: string, bindArgs: Array, callback: AsyncCallback):void

执行包含指定参数但不返回值的SQL语句，语句中的各种表达式和操作符之间的关系操作符号不超过1000个，使用callback异步回调。

此接口不支持执行查询、附加数据库和事务操作，可以使用querySql、query、attach、beginTransaction、commit等接口代替。

不支持分号分隔的多条语句。

参数说明

- *bindArgs*: **Array<ValueType>**，指定SQL语句中参数的值。该值与sql参数语句中的占位符相对应。当sql参数语句完整时，该参数需为空数组。

> executeSql(sql: string, bindArgs?: Array):Promise

执行包含指定参数但不返回值的SQL语句，语句中的各种表达式和操作符之间的关系操作符号不超过1000个，使用Promise异步回调。

此接口不支持执行查询、附加数据库和事务操作，可以使用querySql、query、attach、beginTransaction、commit等接口代替。

不支持分号分隔的多条语句。

参数说明

- *返回值*: **Promise<void>**，无返回结果的Promise对象。

execute

> execute(sql: string, args?: Array):Promise

执行包含指定参数的SQL语句，语句中的各种表达式和操作符之间的关系操作符号不超过1000个，返回值类型为ValueType，使用Promise异步回调。

该接口支持执行增删改操作，支持执行PRAGMA语法的sql，支持对表的操作（建表、删表、修改表）,返回结果类型由执行具体sql的结果决定。

此接口不支持执行查询、附加数据库和事务操作，可以使用querySql、query、attach、beginTransaction、commit等接口代替。

不支持分号分隔的多条语句。

参数说明

- args　　Array　　否　SQL语句中参数的值。该值与sql参数语句中的占位符相对应。当sql参数语句完整时，该参数不填。

Part 3: 代码调试（Debug）指南

牢记：Programming is debugging.

3.1 通过输出日志调试

主要使用*console*模块，其提供了一个简单的调试控制台，类似于浏览器提供的JavaScript控制台机制。

在调试中的主要作用有：

- 定位问题代码所在位置
- 输出变量值
- 监视变量变化

主要方法

> log(message: string, ...arguments: any[]): void

以格式化输出方式打印日志信息。

> info(message: string, ...arguments: any[]): void

以格式化输出方式打印日志信息。(console.log()的别名）。

> debug(message: string, ...arguments: any[]): void

以格式化输出方式打印调试信息。

> warn(message: string, ...arguments: any[]): void

以格式化输出方式打印警告信息。

> error(message: string, ...arguments: any[]): void

以格式化输出方式打印错误信息。

> sassert(value?: Object, ...arguments: Object[]): void

断言打印。语句结果值。若value为假(false)或者省略，则输出以"Assertion failed"开头。如果 value 为真值(true)，则无打印。

示例：

```
console.assert(true, 'does nothing');  // 表达式结果值为true，无打印。
console.assert(2 % 1 == 0, 'does nothing');  // 表达式结果值为true，无打印。

console.assert(false, 'console %s work', 'didn\'t');
// Assertion failed: console didn't work

console.assert();
// Assertion failed
```

> dir(dir?: Object): void

打印对象内容。

示例：

```
class bar {
  baz: boolean = true;
}
let b: bar = {baz: true}
class foo{
  bar: bar = b;
}
let c: foo = {bar: b}
class  c1{
  foo: foo = c;
}
let a: c1 = {foo: c}
console.dir(a);
// Object: {"foo":{"bar":{"baz":true}}}

console.dir(); // 无打印
```

> table(tableData?: Object): void

以表格形式打印数据。

示例：

```
console.table([1, 2, 3]);
// ┌─────────┬────────┐
// | (index) | Values |
// ├─────────┼────────┤
// |    0    |   1    |
// |    1    |   2    |
// |    2    |   3    |
// └─────────┴────────┘
```

```
console.table({ a: [1, 2, 3, 4, 5], b: 5, c: { e: 5 } });

// ┌─────────┬───┬───┬───┬───┬───┬───┬────────┐
// │ (index) │ 0 │ 1 │ 2 │ 3 │ 4 │ e │ Values │
// ├─────────┼───┼───┼───┼───┼───┼───┼────────┤
// │    a    │ 1 │ 2 │ 3 │ 4 │ 5 │   │        │
// │    b    │   │   │   │   │   │   │   5    │
// │    c    │   │   │   │   │   │ 5 │        │
// └─────────┴───┴───┴───┴───┴───┴───┴────────┘
```

> time(label?: string): void

启动可用于计算操作持续时间的计时器。可使用console.timeEnd()关闭计时器并打印经过的时间(单位：ms)。

> timeLog(label?: string, ...arguments: Object[]): void

对于先前通过调用 console.time() 启动的计时器，打印经过时间和其他data参数。

> timeEnd(label?: string): void

停止之前通过调用 console.time() 启动的计时器并将打印经过的时间(单位：ms)。

3.2 通过断点调试

在疑似有问题的代码行处打断点逐行跟踪代码执行顺序，查看变量值、调用堆栈、调用参数、返回值、异常信息等信息，分析代码执行流程，定位问题。

主要思想

1. 断点：在代码中设置断点，可以设置在代码的任意一行，当程序运行到该行时，会暂停运行，并进入调试环境。
2. 单步调试：可以按单步调试的方式逐行执行代码，查看变量值、调用栈、调用堆栈、调用参数、返回值、异常信息等信息。
3. 断点条件：可以设置断点条件，只有满足断点条件时，才会暂停运行。
4. 监视变量：可以监视变量的值，当变量值发生变化时，会自动暂停运行。

Part 4: 重要概念

4.1 异步并发

在ArkTS中，异步是用来描述并发的一种执行方式。

并发是指在同一时间（段）内，存在多个任务同时执行，但多个并发任务不会在同一时刻并行执行。（注意区分并发和并行）

ArkTS提供了两种处理并发的策略：

- 异步并发
  - 代码执行到一定程度后会被暂停，去执行其他任务，待其他任务执行完毕后再继续执行，实现了在同一时间（段）内，存在多个任务同时执行
- 多线程并发
  - 线程级并发，通常涉及数据交换和共享等复杂操作

异步并发的优点是可以提高程序的运行效率，减少等待时间，提高程序的响应能力。

异步编程常见场景

（此部分参考于：https://blog.csdn.net/hqy1989/article/details/142602296 ）

- 网络请求：如HttpRequests等。
- 文件读写：如读写本地文件。
- 数据库操作：如增删改查数据。
- 定时器：如setTimeout、setInterval等。
- 事件处理：如用户点击屏幕等。

共同本质特征是调用后耗时长。

示例：

```
console.log('1 -----> ', 1)
setTimeout(() => {
  console.log('2 -----> ', 2)
}, 100)
console.log('3 -----> ', 3)
setTimeout(() => {
  console.log('4 -----> ', 4)
}, 50)
console.log('5 -----> ', 5)
```

> 输出：猜一下是什么？并进行验证。

异步编程的实现

异步编程的实现方式有多种，常见的有：

- 回调函数
- Promise
    - async/await（基于Promise的语法糖，允许我们以同步的方式写异步代码）

回调函数

回调函数是异步编程的一种实现方式，它是将回调函数作为参数传递给另一个函数，在另一个函数执行完毕后，调用回调函数。

回调函数的优点是简单易用，缺点是不利于代码的阅读和维护。

Promise

Promise是异步编程的第三种实现方式，它是一种对象，用于表示一个异步操作的最终结果（完成或失败及其结果值）。并提供统一的接口，用来处理异步操作的结果。

1. Promise是一个承诺，它代表着某个未来才会发生的事件（通常是一个异步操作），例如：

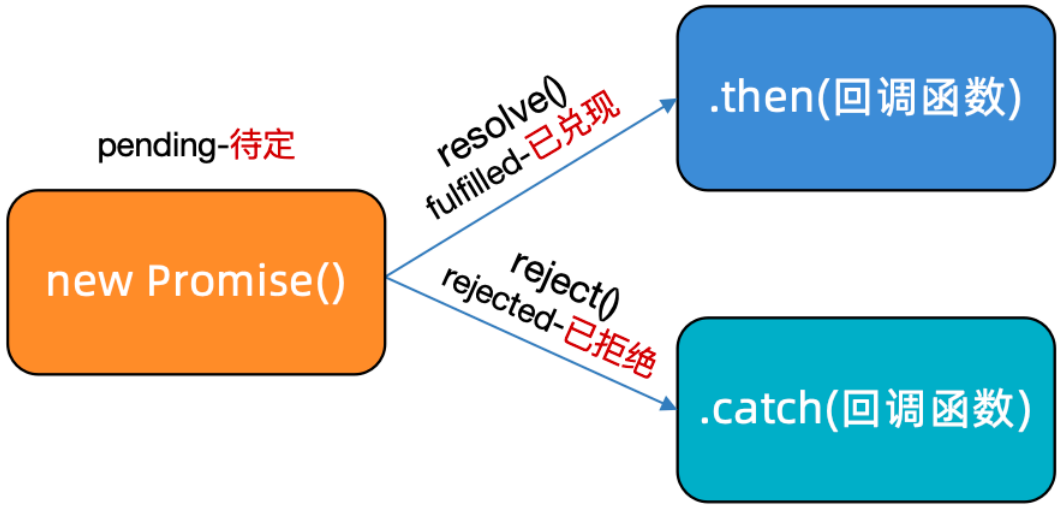
- 你会尽你所能，做出一个让老师眼前一亮的课程设计。

2. Promise对象有三种状态：

- pending（进行中），例如：
  - 现在的承诺状态就是进行中，因为课程设计还没有开始。
- fulfilled（已成功），例如：
  - 假设现在为期一周的课程设计已经结束，老师给了你满意的评价，那这个承诺就是成功的。
- rejected（已失败），例如：
  - 反之，这个承诺就是失败的。

3. Promise可以处理异步操作的结果和捕获异常

- 如果是fulfied状态，则用resolve()处理结果值，并调用then方法指定的回调函数。
- 如果是rejected状态，则用reject()处理错误信息，并调用catch方法指定的回调函数。



示例：

```
function asyncOperation(): Promise<string> {
  return new Promise<string>((resolve, reject) => {
    // 模拟异步操作
    setTimeout(() => {
      if (Math.random() > 0.5) {
        resolve('操作成功');
      } else {
        reject('操作失败');
      }
    }, 1000);
  });
}

asyncOperation()
  .then(result => console.log(result)) //随机数大于0.5返回操作成功
  .catch((error:string) => console.error(error)); //随机数小于0.5返回操作失败
```

4. Promise的链式调用

Promise 的链式写法允许在一个 Promise 的 then 方法中返回另一个 Promise，从而形成链式的调用。

示例：

```
function createRandomPromise(delay: number, successMessage: string,
failureMessage: string): Promise<string> {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (Math.random() > 0.5) {
        resolve(successMessage);
      } else {
        reject(failureMessage);
      }
    }, delay);
  });
}


createRandomPromise(1000, '操作成功', '操作失败')
  .then(result => {
    console.log(result);
    return createRandomPromise(1500, '第二次操作成功', '第二次操作失败');
  })
  .then(result => {
    console.log(result);
    return createRandomPromise(2000, '第三次操作成功', '第三次操作失败');
  })
  .then(result => {
    console.log(result);
  })
  .catch((error: string) => {
    console.error(error);
  });
```

async/await

async/await是基于Promise的语法糖，允许我们以同步的方式写异步代码。使得异步代码的阅读和编写更像是传统的同步代码，提高了代码的可读性和可维护性。

async关键字用于声明一个异步函数，而await关键字则用于等待一个Promise的解决（fulfill）或拒绝（reject）。

示例：

```
async function fetchUserData(userId: number) {
  const response = await http
    .createHttp()
    .request(`https://example.com/api/user/${userId}`) ;
  const data = response.result as User;
  return data;
}
```

await 关键字：
await 只能在 async 函数内部使用，它用于等待一个Promise解决为其值。
await 将暂停函数的执行，直到等待的Promise完成（无论成功或失败）并返回结果。
如果 Promise 被拒绝（即抛出错误），那么 await 表达式会抛出异常，需要在 async 函数中使用 try/catch 块来处理错误。

示例：

```
async function getData() {
  try {
    const response = await http.createHttp().request('https://example.com/data');
    const data =  response.result;
    return data;
  } catch (error) {
    console.error('获取数据时出错:', error);
    return null;
  }
}
```