

13 - 高级特性

Outline

- 应用测试
- Notification Kit
- 第三方库
- NDK开发

应用测试

应用测试

单元测试

在HarmonyOS NEXT应用单元测试可以在测试框架下进行，测试框架由核心模块和扩展模块组成。其中核心模块是测试框架的最小集，包含执行必备核心接口和逻辑。扩展模块是在核心模块的基础上增加一些常用能力，例如用例超时控制、用例筛选、数据驱动、压力测试等。核心模块采用插件化机制，提供接入能力和运行时上下文，扩展模块通过插件的方式接入。

UI测试

通过简洁易用的API提供查找和操作界面控件能力，支持开发者编写基于界面操作的自动化测试脚本。

专项测试

专项测试包括兼容性、稳定性、安全、性能、功耗、UX等，开发者可以结合各维度的应用质量建议，通过提供的多种专项测试工具来保障应用质量。

自动化测试框架

概述

自动化测试框架arkxtest，作为工具集的重要组成部分，支持JS/TS语言的单元测试框架([JsUnit](#))及UI测试框架([UiTest](#))。

JsUnit提供单元测试用例执行能力，提供用例编写基础接口，生成对应报告，用于测试系统或应用接口。

UiTest通过简洁易用的API提供查找和操作界面控件能力，支持用户开发基于界面操作的自动化测试脚本。

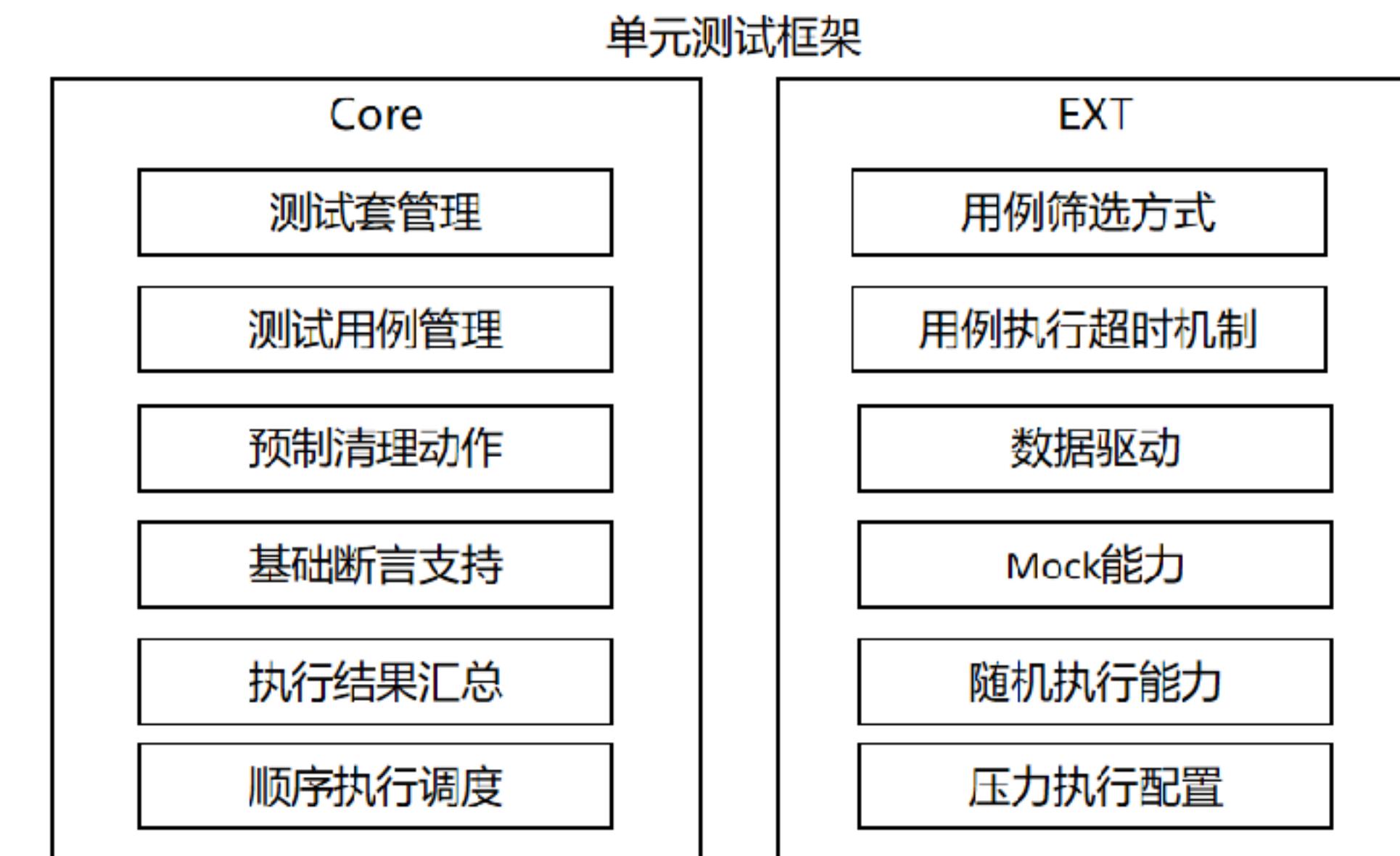
本指南介绍了测试框架的主要功能、实现原理、环境准备，以及测试脚本编写和执行方法。同时，以shell命令方式，对外提供了获取截屏、控件树、录制用户操作、便捷注入UI模拟操作等能力，助力开发者更灵活方便测试和验证。

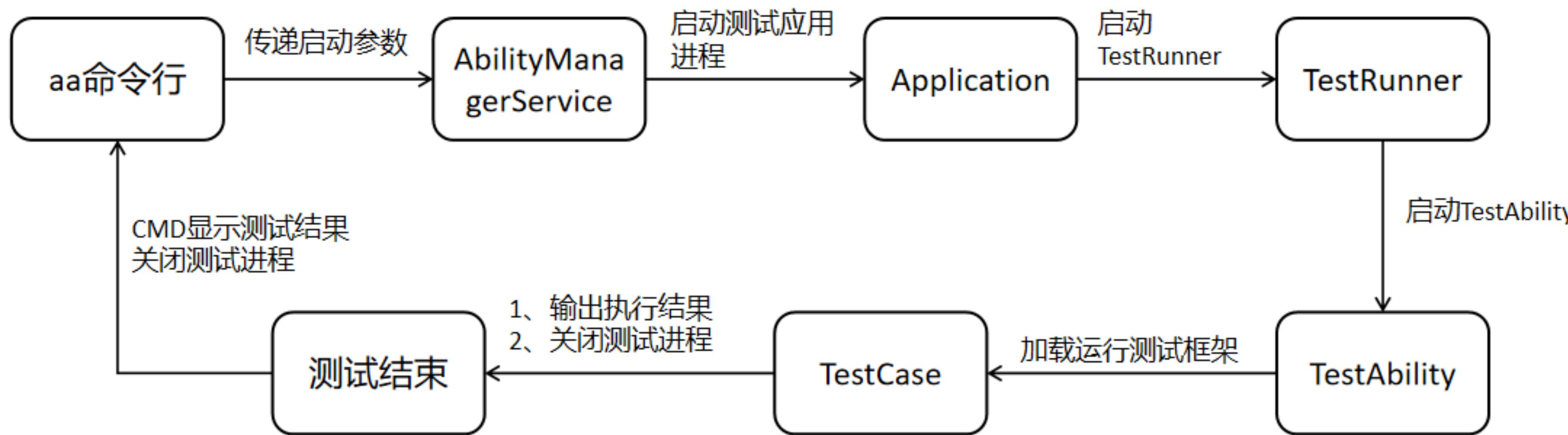
实现原理

测试框架分为单元测试框架和UI测试框架。

单元测试框架是测试框架的基础底座，提供了最基本的用例识别、调度、执行及结果汇总的能力。

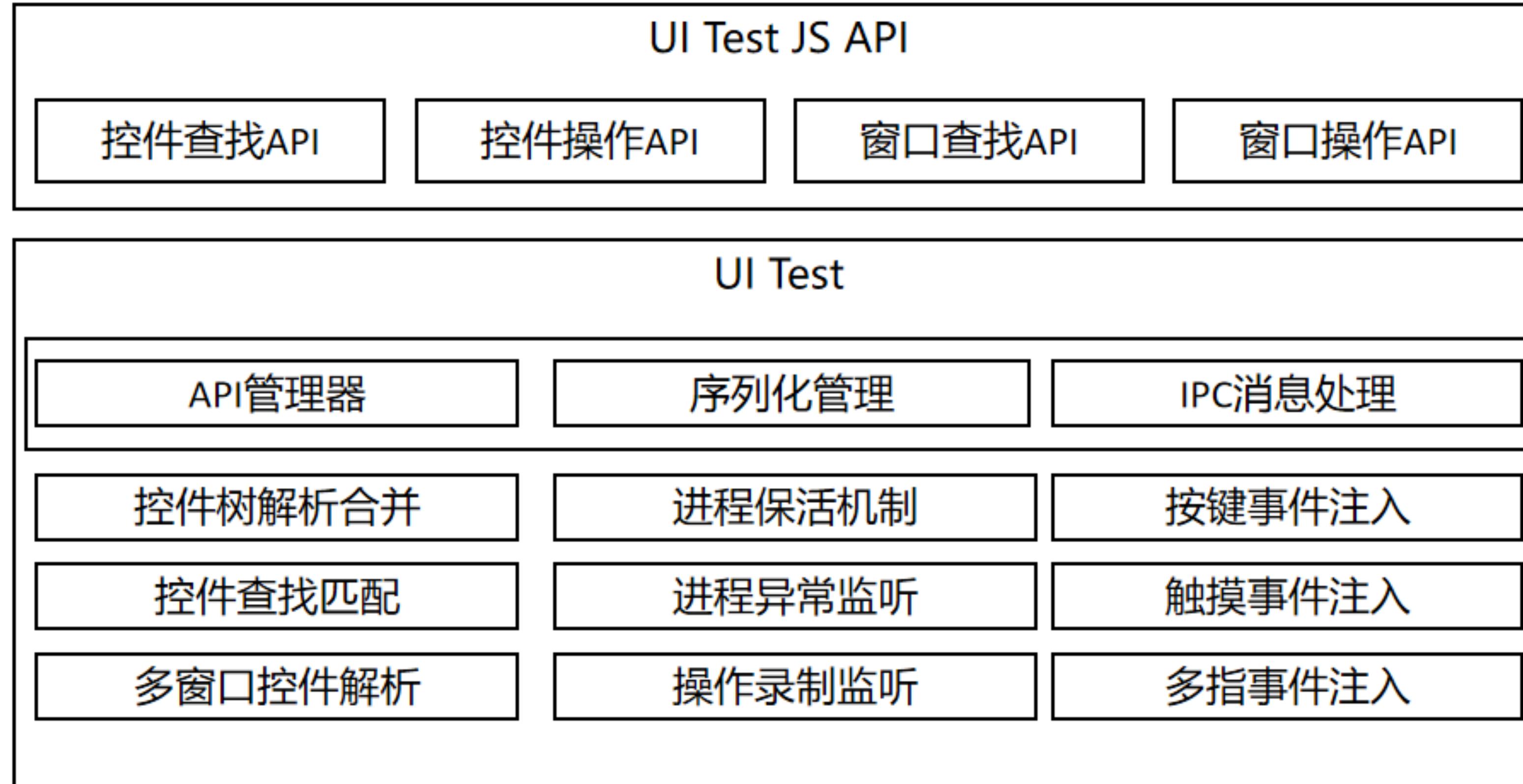
UI测试框架主要对外提供了UiTest API供开发人员在对应测试场景调用，而其脚本的运行基础仍是单元测试框架。





脚本基础流程运行图

UI测试框架



UI测试框架主要功能

编写单元测试代码

```
• import { describe, it, expect } from '@ohos/hypium';
• import { abilityDelegatorRegistry } from '@kit.TestKit';
• import { UIAbility, Want } from '@kit.AbilityKit';
•
• const delegator = abilityDelegatorRegistry.getAbilityDelegator()
• const bundleName = abilityDelegatorRegistry.getArguments().bundleName;
• function sleep(time: number) {
•   return new Promise<void>((resolve: Function) => setTimeout(resolve, time));
• }
• export default function abilityTest() {
•   describe('ActsAbilityTest', () =>{
•     it('testUiExample',0, async (done: Function) => {
•       console.info("uitest: TestUiExample begin");
•       //start tested ability
•       const want: Want = {
•         bundleName: bundleName,
•         abilityName: 'EntryAbility'
•       }
•       await delegator.startAbility(want);
•       await sleep(1000);
•       //check top display ability
•       const ability: UIAbility = await delegator.getCurrentTopAbility();
•       console.info("get top ability");
•       expect(ability.context.abilityInfo.name).toEqual('EntryAbility');
•       done();
•     })
•   })
• }
```

编写UI测试代码

- 编写Index.ets页面代码，作为被测示例demo。

```
• @Entry
• @Component
• struct Index {
•   @State message: string = 'Hello World'
•
•   build() {
•     Row() {
•       Column() {
•         Text(this.message)
•           .fontSize(50)
•           .fontWeight(FontWeight.Bold)
•         Text("Next")
•           .fontSize(50)
•           .margin({top:20})
•           .fontWeight(FontWeight.Bold)
•         Text("after click")
•           .fontSize(50)
•           .margin({top:20})
•           .fontWeight(FontWeight.Bold)
•       }
•       .width('100%')
•     }
•     .height('100%')
•   }
• }
```

- 在ohosTest > ets > test文件夹下.test.ets文件中编写具体测试代码。

```
• import { describe, it, expect } from '@ohos/hypium';
• // 导入测试依赖kit
• import { abilityDelegatorRegistry, Driver, ON } from '@kit.TestKit';
• import { UIAbility, Want } from '@kit.AbilityKit';
•
• const delegator: abilityDelegatorRegistry.AbilityDelegator =
abilityDelegatorRegistry.getAbilityDelegator()
• const bundleName = abilityDelegatorRegistry.getArguments().bundleName;
```

```
• function sleep(time: number) {
•   return new Promise<void>((resolve: Function) => setTimeout(resolve,
time));
• }
• export default function abilityTest() {
•   describe('ActsAbilityTest', () => {
•     it('testUiExample', 0, async (done: Function) => {
•       console.info("uitest: TestUiExample begin");
•       //start tested ability
•       const want: Want = {
•         bundleName: bundleName,
•         abilityName: 'EntryAbility'
•       }
•       await delegator.startAbility(want);
•       await sleep(1000);
•       //check top display ability
•       const ability: UIAbility = await delegator.getCurrentTopAbility();
•       console.info("get top ability");
•
•       expect(ability.context.abilityInfo.name).toEqual('EntryAbility');
•       //ui test code
•       //init driver
•       const driver = Driver.create();
•       await driver.delayMs(1000);
•       //find button on text 'Next'
•       const button = await driver.findComponent(ON.text('Next'));
•       //click button
•       await button.click();
•       await driver.delayMs(1000);
•       //check text
•       await driver.assertComponentExist(ON.text('after click'));
•       await driver.pressBack();
•       done();
•     })
•   })
• }
```

```
import ...

const delegator = abilityDelegatorRegistry.getAbilityDelegator()

export default function abilityTest() {
  describe('ActsAbilityTest', function () {
    it('testUiExample', 0, async function (done) {
      console.info("uitest: TestUiExample begin");
      //start tested ability
      await delegator.executeShellCommand('sa start -b com.ohos.uitest -a MainAbility').then(result =>{
        console.info('Uitest, start ability finished:' + result)
      }).catch(err => {
        console.info('Uitest, start ability failed: ' + err)
      })
      await sleep(1000);
      //check top display ability
      await delegator.getCurrentTopAbility().then((Ability)=>{
        console.info("get top ability");
        expect(Ability.context.abilityInfo.name).toEqual('MainAbility');
      })
      //ui test code
      var driver = await Driver.create();
      await driver.delayMs(1000);
      var button = await driver.findComponent(ON.text('Next'));
      await button.click();
      await driver.delayMs(1000);
      await driver.assertComponentExist(ON.text('after click'));
      await driver.pressBack();
      done();
    })

    function sleep(time) {
      return new Promise((resolve) => setTimeout(resolve, time));
    }
  })
}
```

执行测试用例

The screenshot shows a terminal window with the following output:

```
Run: testUiExample
```

Test Results

```
Tests passed: 1 of 1 test - 13sec 679ms
```

```
10/16 10:57:24: Launching com.ohos.uitest
$ hdc_std uninstall com.ohos.uitest
Testing started at 10:57 ...
$ hdc_std install C:\Users\Administrator\DevEcoStudioProjects\UiTest\entry\build\default\outputs\default\entry-default-signed.hap
$ hdc_std install C:\Users\Administrator\DevEcoStudioProjects\UiTest\entry\build\default\outputs\ohosTest\entry-ohosTest-signed.hap
Running tests

$ hdc_std shell aa test -b com.ohos.uitest -m entry_test -s unitest OpenHarmonyTestRunner -s class ActsAbilityTest#testUiExample -s timeout 15000

Started running tests

Tests run to completion.

test finished
```

Bottom navigation bar:

- Run
- TODO
- Problems
- Profiler
- Log
- Terminal
- Code Linter

查看结果

在cmd窗口执行test命令

示例代码1：执行所有测试用例。

- ```
hdc shell aa test -b xxx -m xxx -s unittest
OpenHarmonyTestRunner
```

示例代码2：执行指定的describe测试套用例，指定多个需用逗号隔开。

- ```
hdc shell aa test -b xxx -m xxx -s unittest  
OpenHarmonyTestRunner -s class s1,s2
```

示例代码3：执行指定测试套中指定的用例，指定多个需用逗号隔开。

- ```
hdc shell aa test -b xxx -m xxx -s unittest
OpenHarmonyTestRunner -s class
testStop#stop_1,testStop1#stop_0
```

示例代码4：执行指定除配置以外的所有的用例，设置不执行多个测试套需用逗号隔开。

- ```
hdc shell aa test -b xxx -m xxx -s unittest  
OpenHarmonyTestRunner -s notClass testStop
```

示例代码5：执行指定it名称的所有用例，指定多个需用逗号隔开。

- ```
hdc shell aa test -b xxx -m xxx -s unittest
OpenHarmonyTestRunner -s itName stop_0
```

示例代码6：用例执行超时时长配置。

- ```
hdc shell aa test -b xxx -m xxx -s unittest  
OpenHarmonyTestRunner -s timeout 15000
```

示例代码7：用例以breakOnError模式执行用例。

- ```
hdc shell aa test -b xxx -m xxx -s unittest
OpenHarmonyTestRunner -s breakOnError true
```

示例代码8：执行测试类型匹配的测试用例。

- ```
hdc shell aa test -b xxx -m xxx -s unittest  
OpenHarmonyTestRunner -s testType function
```

示例代码9：执行测试级别匹配的测试用例。

- ```
hdc shell aa test -b xxx -m xxx -s unittest
OpenHarmonyTestRunner -s level 0
```

示例代码10：执行测试规模匹配的测试用例。

- ```
hdc shell aa test -b xxx -m xxx -s unittest  
OpenHarmonyTestRunner -s size small
```

示例代码11：执行测试用例指定次数。

- ```
hdc shell aa test -b xxx -m xxx -s unittest
OpenHarmonyTestRunner -s stress 1000
```

# 基于shell命令的UI测试

## 截图使用示例

- # 存储路径: /data/local/tmp, 文件名: 时间戳 + .png。
- hdc shell uitest screenCap
- # 指定存储路径和文件名, 存放在/data/local/tmp/下。
- hdc shell uitest screenCap -p /data/local/tmp/1.png

## 获取控件树使用示例

- hdc shell uitest dumpLayout -p /data/local/tmp/1.json

## 用户录制操作

### 说明

录制过程中, 需等待当前操作的识别结果在命令行输出后, 再进行下一步操作。

- # 将当前界面操作记录到/data/local/tmp/record.csv, 结束录制操作使用Ctrl+C结束录制。
- hdc shell uitest uiRecord record
- # 读取并打印录制数据。
- hdc shell uitest uiRecord read

以下举例为: record数据中包含的字段及字段含义, 仅供参考

- {
  - "ABILITY": "com.ohos.launcher.MainAbility", // 前台应用界面
  - "BUNDLE": "com.ohos.launcher", // 操作应用
  - "CENTER\_X": "", // 预留字段, 暂未使用
  - "CENTER\_Y": "", // 预留字段, 暂未使用
  - "EVENT\_TYPE": "pointer", //
  - "LENGTH": "0", // 总体步长
  - "OP\_TYPE": "click", //事件类型, 当前支持点击、双击、长按、拖拽、滑动、抛滑动作录制

```
• "VELO": "0.000000", // 离手速度
• "direction.X": "0.000000", // 总体移动X方向
• "direction.Y": "0.000000", // 总体移动Y方向
• "duration": 33885000.0, // 手势操作持续时间
• "fingerList": [
• {
• "LENGTH": "0", // 总体步长
• "MAX_VEL": "40000", // 最大速度
• "VELO": "0.000000", // 离手速度
• "W1_BOUNDS": {"bottom":361,"left":37,"right":118,"top":280}, // 起点控件bounds
• "W1_HIER": "ROOT,3,0,0,0,0,0,0,0,0,5,0,0,0,0,0,0,0,0", // 起点控件hierarchy
• "W1_ID": "", // 起点控件id
• "W1_Text": "", // 起点控件text
• "W1_Type": "Image", // 起点控件类型
• "W2_BOUNDS": {"bottom":361,"left":37,"right":118,"top":280}, // 终点控件bounds
• "W2_HIER": "ROOT,3,0,0,0,0,0,0,0,0,5,0,0,0,0,0,0,0,0", // 终点控件hierarchy
• "W2_ID": "", // 终点控件id
• "W2_Text": "", // 终点控件text
• "W2_Type": "Image", // 终点控件类型
• "X2_POSI": "47", // 终点x
• "X_POSI": "47", // 起点x
• "Y2_POSI": "301", // 终点y
• "Y_POSI": "301", // 起点y
• "direction.X": "0.000000", // x方向移动量
• "direction.Y": "0.000000" // y方向移动量
• },
• "fingerNumber": "1" //手指数量
• }
```

# 注入UI模拟操作

- # 执行单击事件。
- hdc shell uitest uiInput click 100 100
- 
- # 执行双击事件。
- hdc shell uitest uiInput doubleClick 100 100
- 
- # 执行长按事件。
- hdc shell uitest uiInput longClick 100 100
- # 执行快滑操作, stepLength\_ 缺省。
- hdc shell uitest uiInput fling 10 10 200 200 500
- # 执行慢滑操作。
- hdc shell uitest uiInput swipe 10 10 200 200 500
- 
- # 执行拖拽操作。
- hdc shell uitest uiInput drag 10 10 100 100 500
- # 执行左滑操作
- hdc shell uitest uiInput dircFling 0 500
- # 执行向右滑动操作
- hdc shell uitest uiInput dircFling 1 600
- # 执行向上滑动操作。
- hdc shell uitest uiInput dircFling 2
- # 执行向下滑动操作。
- hdc shell uitest uiInput dircFling 3
- # 执行输入框输入操作。
- hdc shell uitest uiInput inputText 100 100 hello

**案例代码： uitest**

# Notification Kit

# Notification Kit

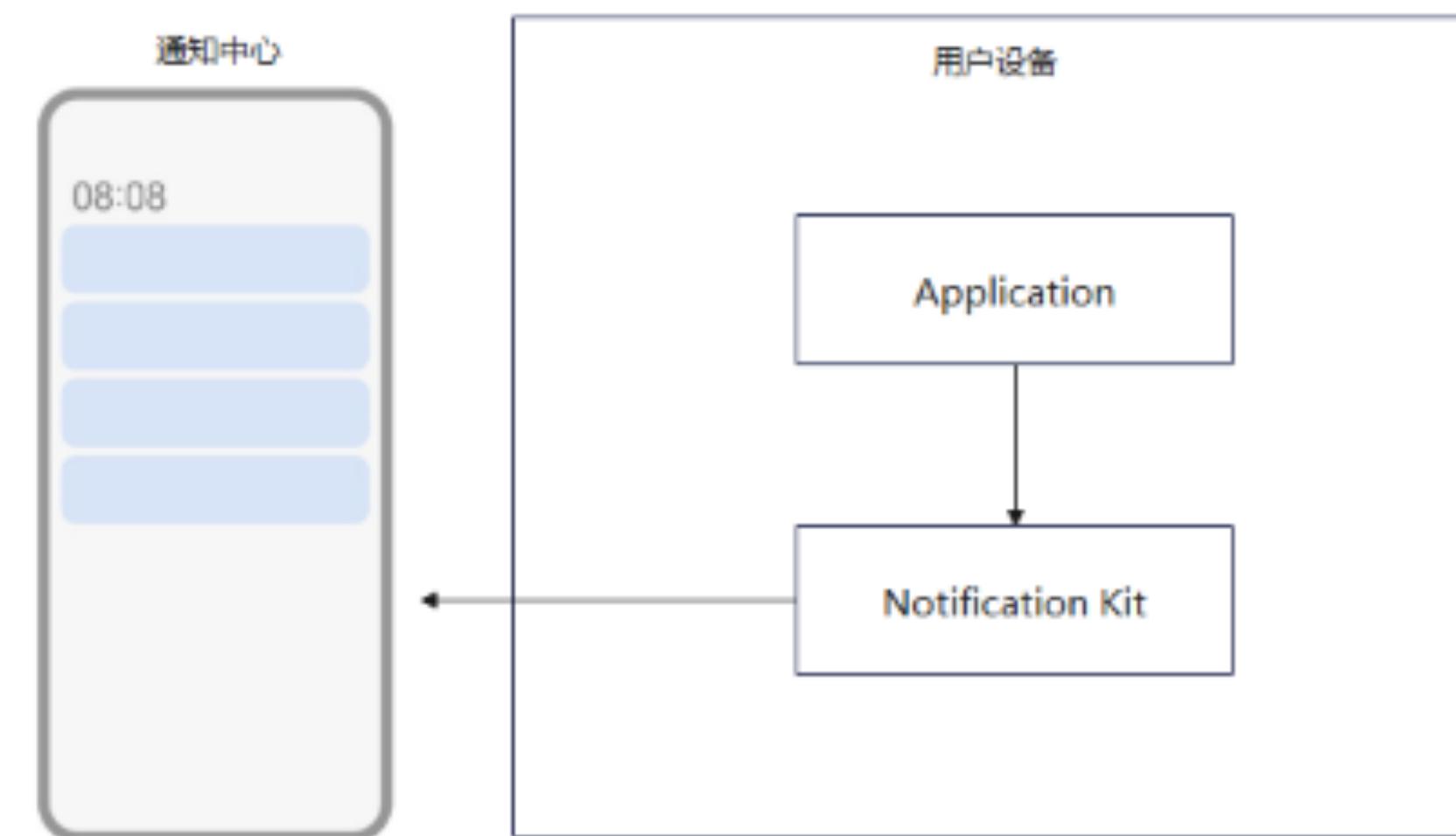
## 使用场景

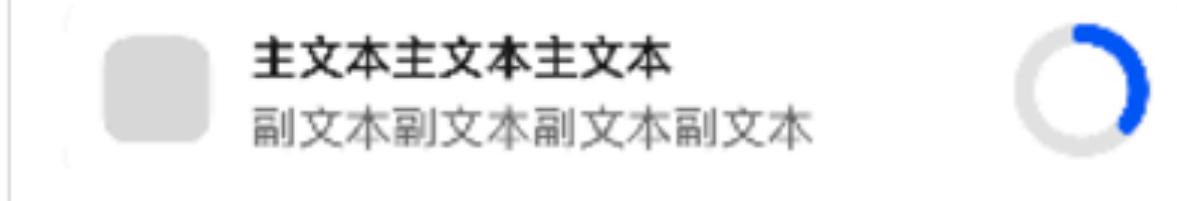
当应用处于前台运行时，开发者可以使用Notification Kit向用户发布通知。当应用转为后台时，本地通知发布通道关闭，开发者需要接入[Push Kit](#)进行云侧离线通知的发布。  
开发者可以在多种场景中运用本地通知能力。如同步用户的上传下载进度、发布即时的客服支付通知、更新运动步数等。

## 能力范围

Notification Kit支持的能力主要包括：

- 发布文本、进度条等类型通知。
- 携带或更新应用通知数字角标。
- 取消曾经发布的某条或全部通知。
- 查询已发布的通知列表。
- 查询应用自身通知开关状态。
- 应用通知用户的能力默认关闭，开发者可拉起授权框，请求用户授权发布通知。



| 类型   | 通知样式                                                                                                                                                 | 规格描述                       |
|------|------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------|
| 文本   |  <p>主文本 标题 10:08<br/>AAAAAAA...<br/>AAAAAAA...<br/>AAAAAAA...</p> | 通知文本内容最多显示三行，超长后以“...”截断。  |
| 多行文本 |  <p>主文本 标题 10:08<br/>1.AAAA...<br/>2.AAAA...<br/>3.AAAA...</p>   | 最多可显示三行内容，每行内容超长后以“...”截断。 |
| 通知角标 |  <p>1</p>                                                       | 以数字的形式展示在右上角。              |
| 进度条  |  <p>主文本 主文本 主文本<br/>副文本 副文本 副文本</p>                             | 进度类通知。                     |

# 通知样式

# 请求通知授权

## 接口说明

接口详情参见[API参考](#)。

### 表1 通知授权接口功能介绍

#### 接口名

#### 描述

isNotificationEnabled():Promise<boolean>

查询通知是否授权。

requestEnableNotification(context: UIAbilityContext): Promise<void>

请求发送通知的许可，第一次调用会弹窗让用户选择。

#### 开发步骤

- 导入NotificationManager模块。

```
• import { notificationManager } from '@kit.NotificationKit';
• import { BusinessError } from '@kit.BasicServicesKit';
• import { hilog } from '@kit.PerformanceAnalysisKit';
• import { common } from '@kit.AbilityKit';
•
• const TAG: string = '[PublishOperation]';
• const DOMAIN_NUMBER: number = 0xFF00;
```

- 请求通知授权。

可通过requestEnableNotification的错误码判断用户是否授权。若返回的错误码为1600004，即为拒绝授权。

```
• let context = getContext(this) as common.UIAbilityContext;
• notificationManager.isNotificationEnabled().then((data: boolean) => {
• hilog.info(DOMAIN_NUMBER, TAG, "isNotificationEnabled success, data: " + JSON.stringify(data));
• if(!data){
• notificationManager.requestEnableNotification(context).then(() => {
• hilog.info(DOMAIN_NUMBER, TAG, `[ANS] requestEnableNotification success`);
• }).catch((err: BusinessError) => {
• if(1600004 == err.code){
• hilog.error(DOMAIN_NUMBER, TAG, `[ANS] requestEnableNotification refused, code is ${err.code}, message is ${err.message}`);
• } else {
• hilog.error(DOMAIN_NUMBER, TAG, `[ANS] requestEnableNotification failed, code is ${err.code}, message is ${err.message}`);
• }
• });
• }
• }).catch((err: BusinessError) => {
• hilog.error(DOMAIN_NUMBER, TAG, `isNotificationEnabled fail, code is ${err.code}, message is ${err.message}`);
•});
```

# 管理通知角标

针对未读的通知，系统提供了角标设置接口，将未读通知个数显示在桌面图标的右上角角标上。

通知增加时，角标上显示的未读通知个数需要增加。

通知被查看后，角标上显示的未读通知个数需要减少，没有未读通知时，不显示角标。

## 接口说明

当角标设定个数取值0时，表示清除角标。取值大于99时，通知角标将显示99+。

- 增加角标数，支持如下两种方法：
  - 发布通知时，在[NotificationRequest](#)的badgeNumber字段里携带，桌面收到通知后，在原角标数上累加、呈现。
  - 调用接口[setBadgeNumber\(\)](#)设置，桌面按设置的角标数呈现。
- 减少角标数，目前仅支持通过[setBadgeNumber\(\)](#)设置。
- 接口名**
- 描述**
- `setBadgeNumber(badgeNumber: number, callback: AsyncCallback<void>): void`
- 设置角标个数。
- 

- `let setBadgeNumberCallback = (err: BusinessError): void => {`
- `if (err) {`
- `hilog.error(DOMAIN_NUMBER, TAG, `Failed to set badge number. Code is ${err.code}, message is ${err.message}`);`
- `return;`
- `}`
- `hilog.info(DOMAIN_NUMBER, TAG, `Succeeded in setting badge number.`);`
- `}`
- 
- `let badgeNumber = 9;`
- `notificationManager.setBadgeNumber(badgeNumber, setBadgeNumberCallback);`
- 
- 减少角标个数。

一条通知被查看后，应用需要调用接口设置剩下未读通知个数，桌面刷新角标。

## 开发步骤

- 导入NotificationManager模块。

```
• import { notificationManager } from '@kit.NotificationKit';
• import { hilog } from '@kit.PerformanceAnalysisKit';
• import { BusinessError } from '@kit.BasicServicesKit';
•
• const TAG: string = '[PublishOperation]';
• const DOMAIN_NUMBER: number = 0xFF00;
•
```

- 增加角标个数。

发布通知在[NotificationRequest](#)的badgeNumber字段里携带，可参考[通知发布](#)章节。

示例为调用setBadgeNumber接口增加角标，在发布完新的通知后，调用该接口。

```
• let setBadgeNumberCallback = (err: BusinessError): void => {
• if (err) {
• hilog.error(DOMAIN_NUMBER, TAG, `Failed to set badge number. Code is ${err.code}, message is ${err.message}`);
```

# setBadgeNumber为异步接口

由于setBadgeNumber为异步接口，使用setBadgeNumber连续设置角标时，为了确保执行顺序符合预期，需要确保上一次设置完成后才能进行下一次设置。

- 反例

每次接口调用是相互独立的、没有依赖关系的，实际执行时无法保证调用顺序。

示例如下：

```
● let badgeNumber: number = 10;
●
● notificationManager.setBadgeNumber(badgeNumber).then(() => {
● hilog.info(DOMAIN_NUMBER, TAG, `setBadgeNumber
10 success.`);
● });
● badgeNumber = 11;
●
● notificationManager.setBadgeNumber(badgeNumber).then(() => {
● hilog.info(DOMAIN_NUMBER, TAG, `setBadgeNumber
11 success.`);
● });

●
```

- 
- 正例

多次接口调用存在依赖关系，确保上一次设置完成后才能进行下一次设置。

示例如下：

```
● let badgeNumber: number = 10;
●
● notificationManager.setBadgeNumber(badgeNumber).then(() => {
● hilog.info(DOMAIN_NUMBER, TAG, `setBadgeNumber
10 success.`);
● badgeNumber = 11;
●
● notificationManager.setBadgeNumber(badgeNumber).then(() => {
● hilog.info(DOMAIN_NUMBER, TAG,
`setBadgeNumber 11 success.`);
● });
● });

●
```

文本类型通知主要应用于发送短信息、提示信息等，支持普通文本类型和多行文本类型。

表1 基础类型通知中的内容分类

| 类型                              | 描述      |
|---------------------------------|---------|
| NOTIFICATION_CONTENT_BASIC_TEXT | 普通文本类型。 |
| NOTIFICATION_CONTENT_MULTILINE  | 多行文本类型。 |

## 接口说明

通知发布接口说明详见下表，通知发布的详情可通过入参[NotificationRequest](#)来进行指定，可以包括通知内容、通知ID、通知的通道类型和通知发布时间等信息。

| 接口名                                                                        | 描述            |
|----------------------------------------------------------------------------|---------------|
| publish(request: NotificationRequest, callback: AsyncCallback<void>): void | 发布通知。         |
| cancel(id: number, label: string, callback: AsyncCallback<void>): void     | 取消指定的通知。      |
| cancelAll(callback: AsyncCallback<void>): void                             | 取消所有该应用发布的通知。 |

# 发布文本型通知

# 开发步骤

## 开发步骤

- 导入模块。

```
• import { notificationManager } from '@kit.NotificationKit';
• import { BusinessError } from '@kit.BasicServicesKit';
• import { hilog } from '@kit.PerformanceAnalysisKit';
•
• const TAG: string = '[PublishOperation]';
• const DOMAIN_NUMBER: number = 0xFF00;
•
• 构造NotificationRequest对象，并发布通知。
• 普通文本类型通知由标题、文本内容和附加信息三个字段组成，其中标题和文本内容是必填字段，大小均需要小于200字节，超出部分会被截断。
•
• let notificationRequest: notificationManager.NotificationRequest = {
• id: 1,
• content: {
• notificationContentType:
• notificationManager.ContentType.NOTIFICATION_CONTENT_BASIC_TEXT, // 普通文本类型通知
• normal: {
• title: 'test_title',
• text: 'test_text',
• additionalText: 'test_additionalText',
• }
• }
• };
• notificationManager.publish(notificationRequest, (err: BusinessError) => {
• if (err) {
• hilog.error(DOMAIN_NUMBER, TAG, `Failed to publish notification. Code is ${err.code}, message is ${err.message}`);
• return;
• }
• hilog.info(DOMAIN_NUMBER, TAG, 'Succeeded in publishing notification.');
• });
•
```

- 多行文本类型通知继承了普通文本类型的字段，同时新增了多行文本内容、内容概要和通知展开时的标题，其字段均小于200字节，超出部分会被截断。通知默认显示与普通文本相同，展开后，标题显示为展开后标题内容，多行文本内容多行显示。

```
• let notificationRequest: notificationManager.NotificationRequest = {
• id: 3,
• content: {
• notificationContentType:
• notificationManager.ContentType.NOTIFICATION_CONTENT_MULTILINE, // 多行文本类型通知
• multiLine: {
• title: 'test_title',
• text: 'test_text',
• briefText: 'test_briefText',
• longTitle: 'test_longTitle',
• lines: ['line_01', 'line_02', 'line_03', 'line_04'],
• }
• }
• };
• // 发布通知
• notificationManager.publish(notificationRequest, (err: BusinessError) => {
• if (err) {
• hilog.error(DOMAIN_NUMBER, TAG, `Failed to publish notification. Code is ${err.code}, message is ${err.message}`);
• return;
• }
• hilog.info(DOMAIN_NUMBER, TAG, 'Succeeded in publishing notification.');
• });
•
• 删除通知。
•
• notificationManager.cancel(1, (err: BusinessError) => {
• if (err) {
• hilog.error(DOMAIN_NUMBER, TAG, `Failed to cancel notification. Code is ${err.code}, message is ${err.message}`);
• return;
• }
• hilog.info(DOMAIN_NUMBER, TAG, 'Succeeded in cancel notification.');
• });
•
```

# 发布进度条通知

进度条通知也是常见的通知类型，主要应用于文件下载、事务处理进度显示。当前系统提供了进度条模板，发布通知应用设置好进度条模板的属性值，如模板名、模板数据，通过通知子系统发送到通知栏显示。

目前系统模板仅支持进度条模板，通知模板[NotificationTemplate](#)中的data参数为用户自定义数据，用于显示与模块相关的数据。

## 接口说明

[isSupportTemplate\(\)](#)是查询模板是否支持接口，目前仅支持进度条模板。

### 接口名

### 描述

`isSupportTemplate(templateName: string): Promise<boolean>`

查询模板是否存在。

# 开发步骤

- 导入模块。
- 构造进度条模板对象，并发布通知。

```
• import { notificationManager } from '@kit.NotificationKit';
• import { BusinessError } from '@kit.BasicServicesKit';
• import { hilog } from '@kit.PerformanceAnalysisKit';
•
• const TAG: string = '[PublishOperation]';
• const DOMAIN_NUMBER: number = 0xFF00;
•
• notificationManager.isSupportTemplate('downloadTemplate').then((data:boolean) => {
• hilog.info(DOMAIN_NUMBER, TAG, 'Succeeded in supporting download template notification.');
• let isSupportTpl: boolean = data; // isSupportTpl的值为true表示支持downloadTemplate模板类通知, false表示不支持
• }).catch((err: BusinessError) => {
• hilog.error(DOMAIN_NUMBER, TAG, `Failed to support download template notification. Code is ${err.code}, message is ${err.message}`);
• });
•
```

## 说明

查询系统支持进度条模板后，再进行后续的步骤操作。

```
• let notificationRequest: notificationManager.NotificationRequest =
{
• id: 5,
• content: {
• notificationContentType:
notificationManager.ContentType.NOTIFICATION_CONTENT_BASIC_TEXT,
• normal: {
• title: 'test_title',
• text: 'test_text',
• additionalText: 'test_additionalText'
• }
• },
• // 构造进度条模板, name字段当前需要固定配置为downloadTemplate
• template: {
• name: 'downloadTemplate',
• data: { title: 'File Title', fileName: 'music.mp4',
progressValue: 45 }
• }
• }
•
• // 发布通知
• notificationManager.publish(notificationRequest, (err: BusinessError) => {
• if (err) {
• hilog.error(DOMAIN_NUMBER, TAG, `Failed to publish notification. Code is ${err.code}, message is ${err.message}`);
• return;
• }
• hilog.info(DOMAIN_NUMBER, TAG, 'Succeeded in publishing notification.');
• });
• };
```

# 为通知添加行为意图

getWantAgent(info: WantAgentInfo, callback:  
AsyncCallback<WantAgent>): void  
创建WantAgent。



# 开发步骤

- 导入模块。

```
• import { notificationManager } from '@kit.NotificationKit';
• import { wantAgent, WantAgent } from '@kit.AbilityKit';
• import { BusinessError } from '@kit.BasicServicesKit';
• import { hilog } from '@kit.PerformanceAnalysisKit';
•
• const TAG: string = '[PublishOperation]';
• const DOMAIN_NUMBER: number = 0xFF00;
```

- 创建WantAgentInfo信息。

场景一：创建拉起UIAbility的WantAgent的WantAgentInfo信息。

```
• let wantAgentObj:WantAgent; // 用于保存创建成功的wantAgent对象，后续使用其完成触发的动作。
•
• // 通过WantAgentInfo的operationType设置动作类型
• let wantAgentInfo:wantAgent.WantAgentInfo = {
• wants: [
• {
• deviceId: '',
• bundleName: 'com.samples.notification',
• abilityName: 'SecondAbility',
• action: '',
• entities: [],
• uri: '',
• parameters: {}
• }
•],
• actionType: wantAgent.OperationType.START_ABILITY,
• requestCode: 0,
• wantAgentFlags:[wantAgent.WantAgentFlags.CONSTANT_FLAG]
• };
```

场景二：创建发布公共事件的WantAgent的WantAgentInfo信息。

```
• let wantAgentObj:WantAgent; // 用于保存创建成功的wantAgent对象，后续使用其完成触发的动作。
•
• // 通过WantAgentInfo的operationType设置动作类型
• let wantAgentInfo:wantAgent.WantAgentInfo = {
• wants: [
• {
• action: 'event_name', // 设置事件名
• parameters: {},
• }
•],
• actionType: wantAgent.OperationType.SEND_COMMON_EVENT,
```

- requestCode: 0,  
wantAgentFlags: [wantAgent.WantAgentFlags.CONSTANT\_FLAG],  
};

- 调用getWantAgent()方法进行创建WantAgent。

```
• // 创建WantAgent
• wantAgent.getWantAgent(wantAgentInfo, (err: BusinessError, data:WantAgent) => {
• if (err) {
• hilog.error(DOMAIN_NUMBER, TAG, `Failed to get want agent. Code is ${err.code}, message is ${err.message}`);
• return;
• }
• hilog.info(DOMAIN_NUMBER, TAG, 'Succeeded in getting want agent.');
• wantAgentObj = data;
• });
```

- 构造NotificationRequest对象，并发布WantAgent通知。

```
• // 构造NotificationRequest对象
• let notificationRequest: notificationManager.NotificationRequest = {
• content: {
• notificationContentType: notificationManager.ContentType.NOTIFICATION_CONTENT_BASIC_TEXT,
• normal: {
• title: 'Test_Title',
• text: 'Test_Text',
• additionalText: 'Test_AdditionalText',
• },
• },
• id: 6,
• label: 'TEST',
• // wantAgentObj使用前需要保证已被赋值（即步骤3执行完成）
• wantAgent: wantAgentObj,
• }
```

```
• notificationManager.publish(notificationRequest, (err: BusinessError) => {
• if (err) {
• hilog.error(DOMAIN_NUMBER, TAG, `Failed to publish notification. Code is ${err.code}, message is ${err.message}`);
• return;
• }
• hilog.info(DOMAIN_NUMBER, TAG, 'Succeeded in publishing notification.');
• });
```

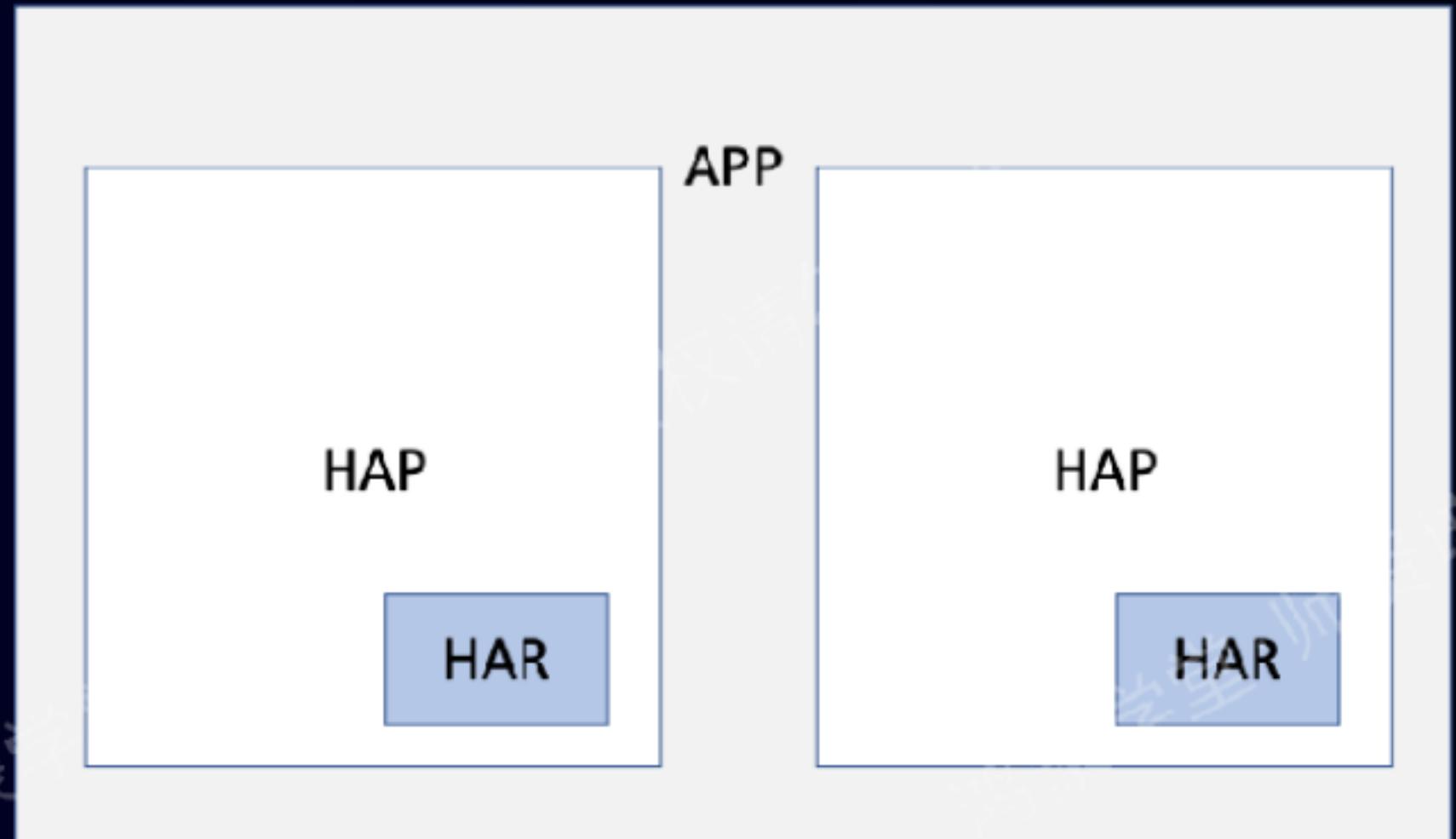
- 用户通过点击通知栏上的通知，系统会自动触发WantAgent的动作。

**实践案例：**

**DownloadNotification**

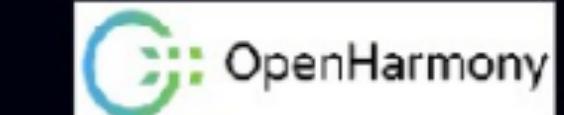
# 第三方库

- 三方库以HAR（Harmony Archive）的形态呈现，可以包含代码、C++库、资源和配置文件。
- 通过三方库HAR可以实现多个模块或多个工程共享ArkUI组件、资源等。
- HAR不能独立安装运行在设备上，HAR的代码和资源跟随使用方编译。



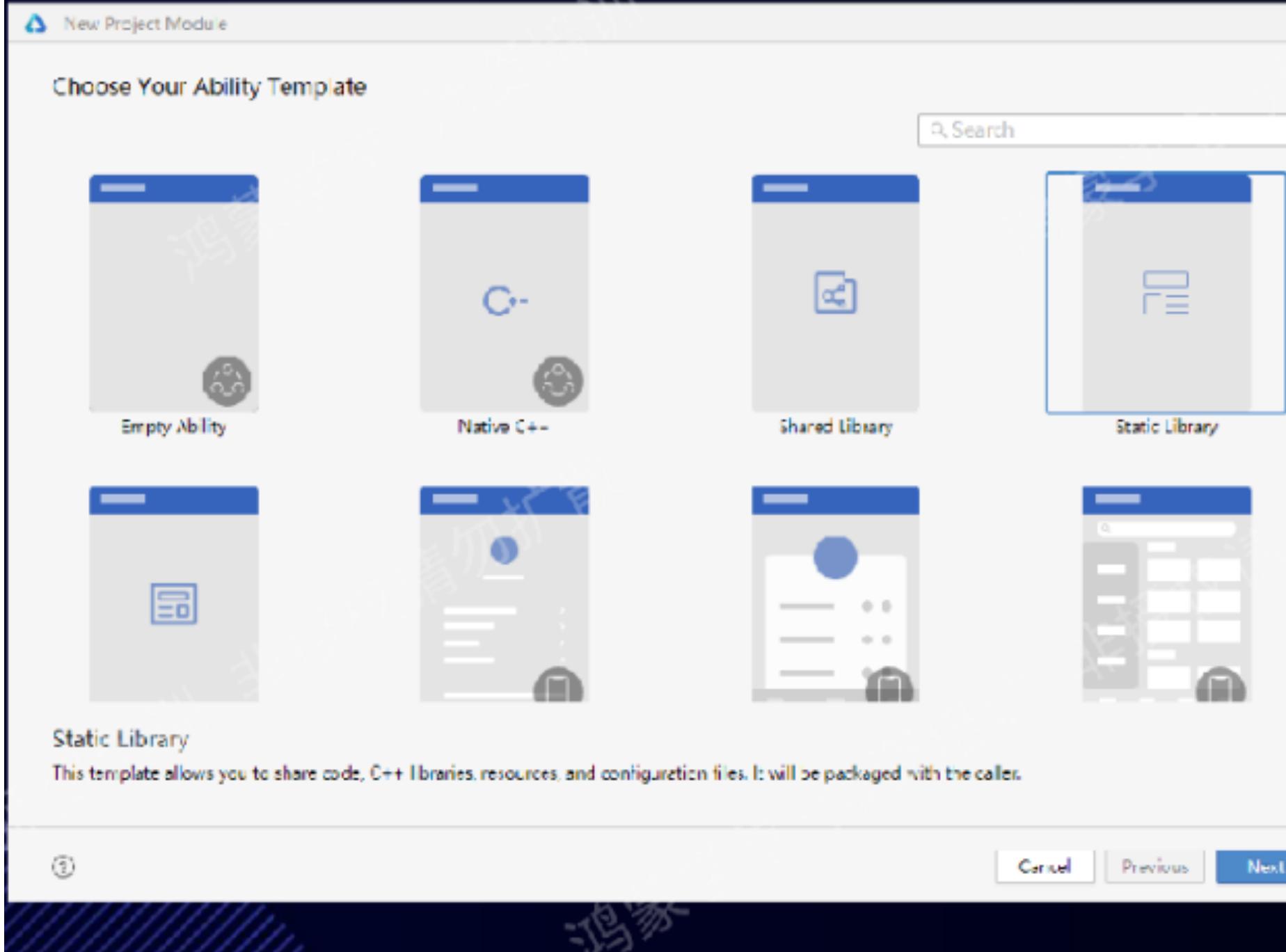
三方库在APP中的形态示意图

# 第三方库

|                   |  npm |  Flutter |  React Native |  OpenHarmony |
|-------------------|-----------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|
| ①源码               |      |          |               |              |
| ②包                | <code>package.json</code>                                                               | <code>pubspec.yaml</code>                                                                   | <code>package.json</code>                                                                        | <code>oh-package.json5</code>                                                                   |
| ③搜索<br>&使用包       | <a href="https://npmjs.com">npmjs.com</a>                                               | <a href="https://Pub.dev">Pub.dev</a>                                                       | <a href="https://reactnative.directory">reactnative.directory</a>                                | <a href="https://ohpm.openatom.cn">ohpm.openatom.cn</a>                                         |
| ④IDE环境集成<br>等基础设施 | NPM CLI                                                                                 | flutter                                                                                     | React Native CLI                                                                                 | DevEco Studio<br>(OHPM CLI)                                                                     |

# 三方库中心仓与包管理工具

在菜单栏选择File > New > Module, 选择Static Library:



三方库工程结构:

The screenshot shows a code editor with the file 'oh-package.json5' open. The file contains the following JSON configuration:

```
name: "library",
version: "1.0.0",
description: "Please describe the basic information.",
main: "index.ets",
author: "",
license: "Apache-2.0",
dependencies: {}
```

The project structure visible in the sidebar includes: MyApplication6 [MyApplication], Jvigor, idea, AppScope, entry, hvgor, library, src, main, ets, components, mainpage, MainPage.ets, resources, module.json5, .gitignore, build-profile.json5, hvgorfile.ts, index.ets, oh-package.json5, .gitignore, build-profile.json5, hvgorfile.ts, hvgor, hvgor.bat, local.properties, oh-package.json5.

# 三方库开发

## 开发三方库的通用流程

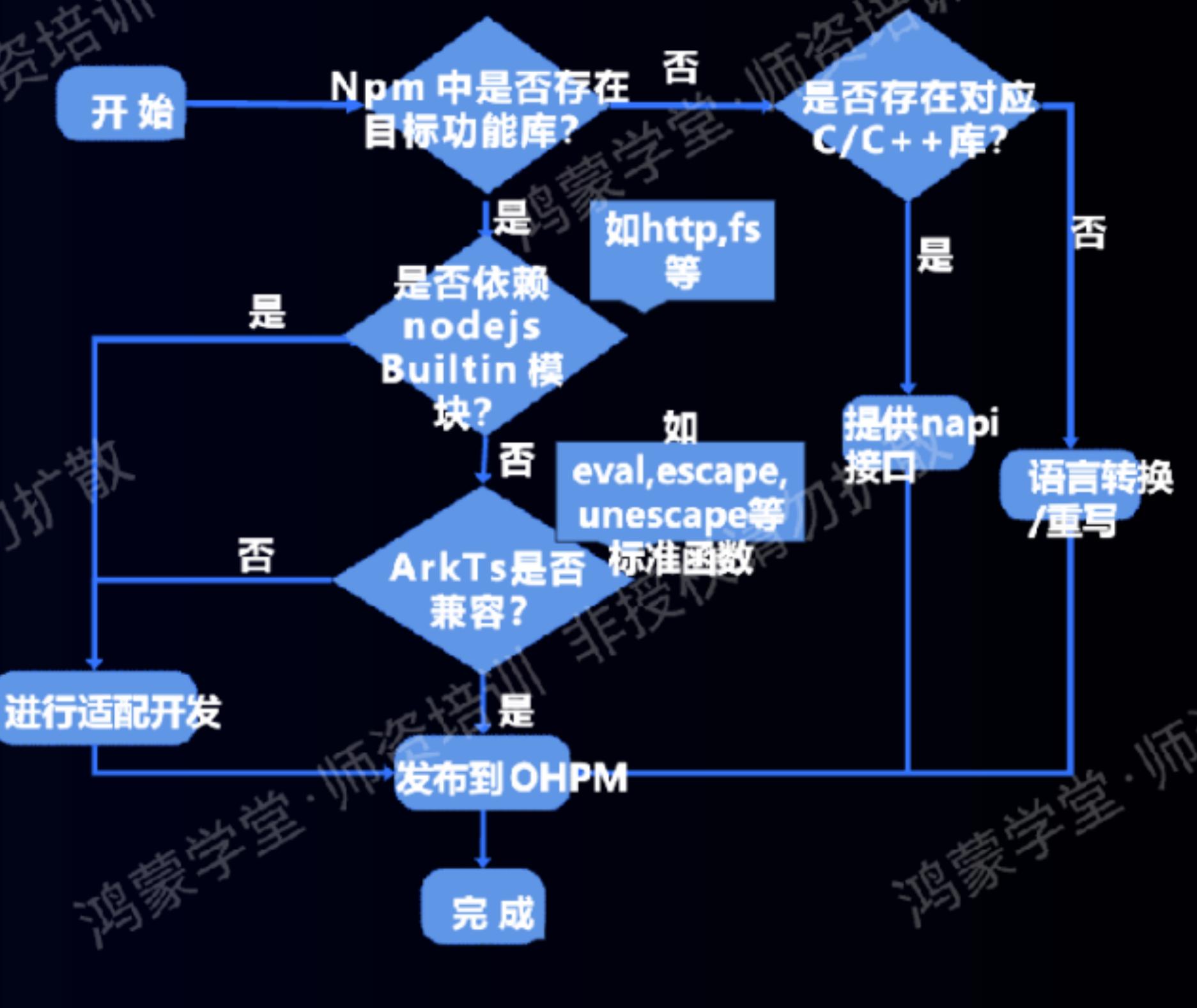
移植JS三方库

第一步：JS/TS库选型。

第二步：工具 ([js-e2e工具](#)) 扫描三方库，检查是否依赖node.js/web内置模块。

第三步：跑通XTS用例，验证所有对外暴露接口可用。

第四步：规范源码目录结构，开源代码到Gitee。



# 三方库开发移植

# 针对C\C++ 开源库的移植方式

其他开发者已验证编译通过的C/C++三方库：  
[https://gitee.com/openharmony-sig/tpc\\_c\\_cplusplus/blob/master/docs/thirdparty\\_list.md](https://gitee.com/openharmony-sig/tpc_c_cplusplus/blob/master/docs/thirdparty_list.md)

Step 1

## 运行时依赖分析

- 是否存在鸿蒙生态不支持的API
- 是否有鸿蒙生态当前不支持的框架?  
(如CEF、OpenGL、X11等)

风险识别工具：

[https://gitee.com/han\\_jin\\_fei/e2e/tree/master/thirdparty\\_compare](https://gitee.com/han_jin_fei/e2e/tree/master/thirdparty_compare)

详细指导：

<https://docs.qq.com/doc/DVmJVTFZpaGt0T0xL>

Step 2

## 编译构建

- 设置SDK
- 设置编译工具链
- 完成构建

交叉编译三方库的工具lycium:

[https://gitee.com/openharmony-sig/tpc\\_c\\_cplusplus/tree/master/lycium](https://gitee.com/openharmony-sig/tpc_c_cplusplus/tree/master/lycium)

Step 3

## 测试验证

- 执行开源库自带的测试用例

测试验证工具CItools:

[https://gitee.com/openharmony-sig/tpc\\_c\\_cplusplus/blob/master/lycium/CItools/README\\_zh.md](https://gitee.com/openharmony-sig/tpc_c_cplusplus/blob/master/lycium/CItools/README_zh.md)



# 三方库开发移植

## Case1:开源js库完全支持，不用修改，直接发布ohpm

典型三方库：

- 1.base64-js
- 2.bignumber.js
- 3.pako
- 4.dayjs

处理方式：

方式一.新建OpenHarmony开源三方库，发布到中心仓库

方式二.提供样例工程，补充xts用例，确认在OpenHarmony设备可完全验证后。

反馈三方库的名称+版本给中心仓管理员在后台直接配置。

联系方式： [web@openharmony.io](mailto:web@openharmony.io)。

# 三方库移植

## Case2:对现有node依赖模块进行适配，侵入式修改 (1)

典型三方库：

1.protobufjs

问题：

源库通过相对路径，读取proto模版文件，  
使用nodejs的fs模块

修改方式：

侵入式修改：通过引入resourceManager，  
从rawfile下读取proto模版文件。

由于读取rawfile的方法只有异步方式，  
只能在多处地方改成await +async模式，  
变异步为同步。

```
+
- ProtoBuf.LoadFromTofile = function(filename, callback, builder) {
+ ProtoBuf.LoadFromTofile = async function(filename, callback, builder, resourceManager) {
+ ProtoBuf.resourceManager = resourceManager;
- if (callback && typeof callback === 'object')
+ if (callback && typeof callback === 'function')
- builder = callback,
+ callback = null;
- else if (!callback || typeof callback !== 'function')
+ else if (!callback)
- callback = null;
+ return ProtoBufUtil.fetch(typeof filename === 'string' ? filename : filename['root']+"/"+filename['file'], Function(contents) {
+ return ProtoBufUtil.fetch(typeof filename === 'string' ? filename : filename['root']+"/"+filename['file'], async function(contents) {
- if (contents === null) {
+ if (contents === null) {
- callback(Error("Failed to fetch file"));
+ callback(null, ProtoBuf.loadProto(contents, builder, filename));
- }
+ try {
- callback(null, ProtoBuf.loadProto(contents, builder, filename));
+ callback(null, await ProtoBuf.loadProto(contents, builder, filename));
- } catch (e) {
+ } catch (e) {
- callback(e);
+ callback(e);
- }
+ }
- var contents = ProtoBufUtil.fetch(typeof filename === 'object' ? filename['root']+"/"+filename['file'] : filename);
+ var contents = await ProtoBufUtil.fetch(typeof filename === 'object' ? filename['root']+"/"+filename['file'] : filename);
- return contents === null ? null : ProtoBuf.loadProto(contents, builder, filename);
+ return contents === null ? null : await ProtoBuf.loadProto(contents, builder, filename);
- });
+ });
-
```

可参考该链接，修改protobufjs fetch为异步方法，支持通过resourcemanage读取rawfile下的文件：  
<https://gitee.com/openharmony-tpc/protobuf/commit/29952f82fcae008eb8b1fde89f4fd8ddfe21fdf7>

# 三方库移植

## Case2:对现有node依赖模块进行适配，侵入式修改（2）

典型三方库:

2. arangojs

问题:

原库使用了nodejs的buffer模块，实现base64编码

修改方式:

侵入式修改:使用ohos的buffer 处理base64编码

可参考该链接和代码:

<https://gitee.com/openharmony-sig/arangojs/blob/5f90db8fdfecfcff2259d19d86d7ab8cd8e6bfd2/library/src/main/ets/core/lib/btoa.ts>

```
1 /**
2 * @internal
3 */
4 // TODO 使用openharmony ohos.buffer 实现
5 import buffer from '@ohos.buffer';
6 export function base64Encode(str: string) {
7 return buffer.from(str).toString("base64");
8 }
```

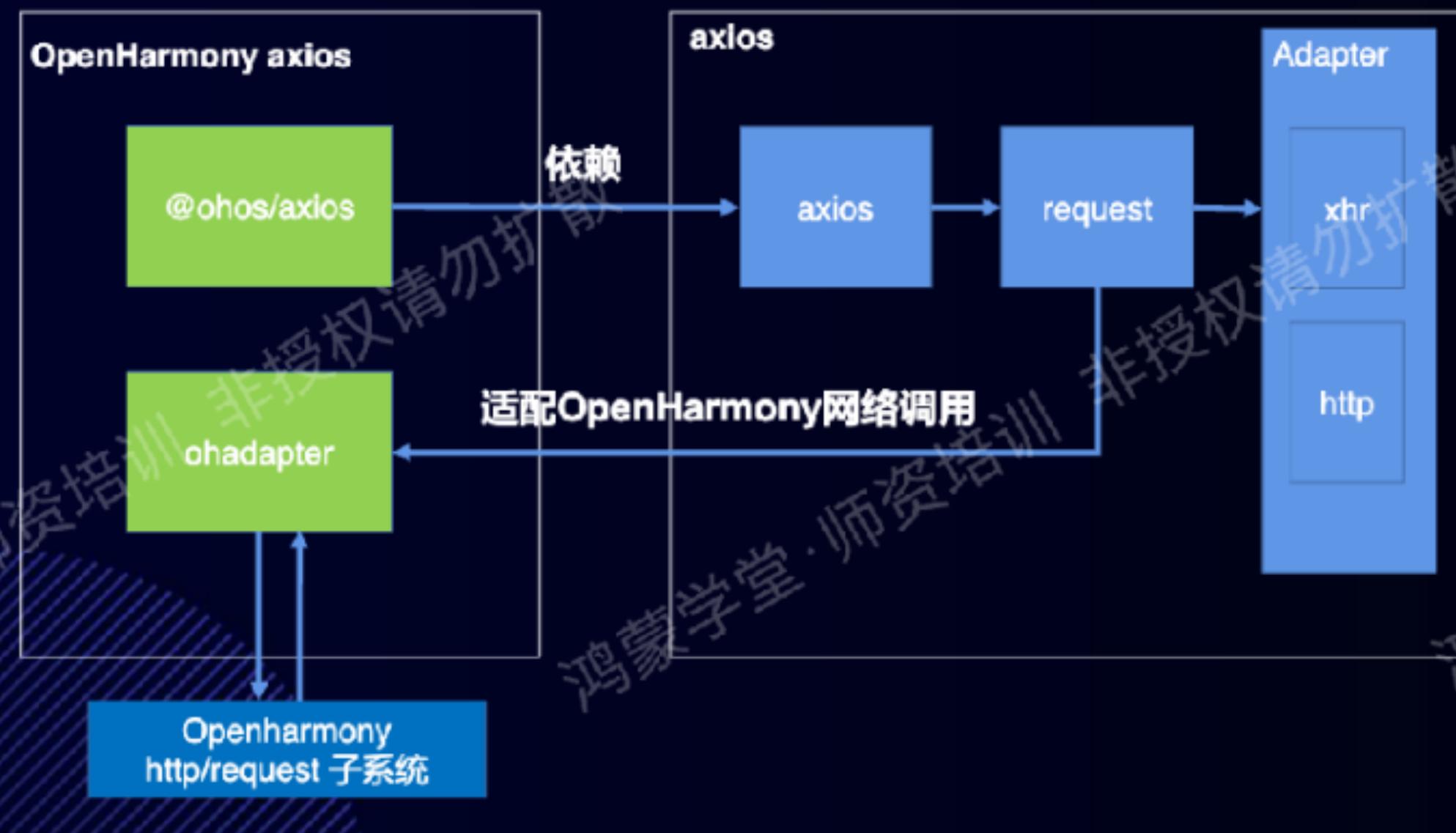


# 三方库移植

## Case3:通过架构设计适配，非侵入式修改

典型三方库：  
@ohos/axios

处理方式：  
基于架构适配鸿蒙



A screenshot of a code editor showing the 'ohos\_axios' directory structure and a file named 'index.js'.

The 'index.js' file contains the following code, with the 'knownAdapters' section highlighted in red:

```
1 import utils from './utils.js';
2 //Import httpAdapter from './http.js';
3 //Import xhrAdapter from './xhr.js';
4 import ohosAdapter from './ohos/index.js';
5 import AxiosAdapter from './ccme/adapterhorn.js';

7 const knownAdapters = {
8 // http: httpAdapter,
9 // xhr: xhrAdapter,
10 ohos: ohosAdapter
11};

12 util.forEach(knownAdapters, (fn, value) => {
13 if(fn) {
14 try {
15 Object.defineProperty(fn, 'name', {value});
16 } catch (e) {
17 // do nothing
18 }
19 }
20 });

21 module.exports = AxiosAdapter;
```



# 三方库开发移植

## Case4: 基于C库移植，通过NDK对js侧提供接口

典型三方库：

@ohos/mqtt, @ohos/mmkv, @ohos/coap

```
this.mqttAsyncClient = MqttAsync.createMqtt({
 url: "ip:port",
 clientId: "e5fatos4jh3l79lndb0bs",
 persistenceType: 1,
})

let options = {
 //set userName and password
 userName: "",
 password: "",
 connectTimeout: 30,
 version: 0,
};

try{
 let result = await this.mqttAsyncClient.connect(options)
 console.log("mqtt connect success "+ JSON.stringify(result));
}catch(err){
 console.log("mqtt connect fail "+ JSON.stringify(err));
}
```



# 三方库开发移植

## Case5:参考其他三方库的实现方式，完全重写

典型三方库：

1. 参考Android MPAndroid chart

<https://gitee.com/openharmony-sig/ohos-MPChart>

2. 参考Glide

<https://gitee.com/openharmony-tpc/ImageKnife>

3. 参考安卓greendao

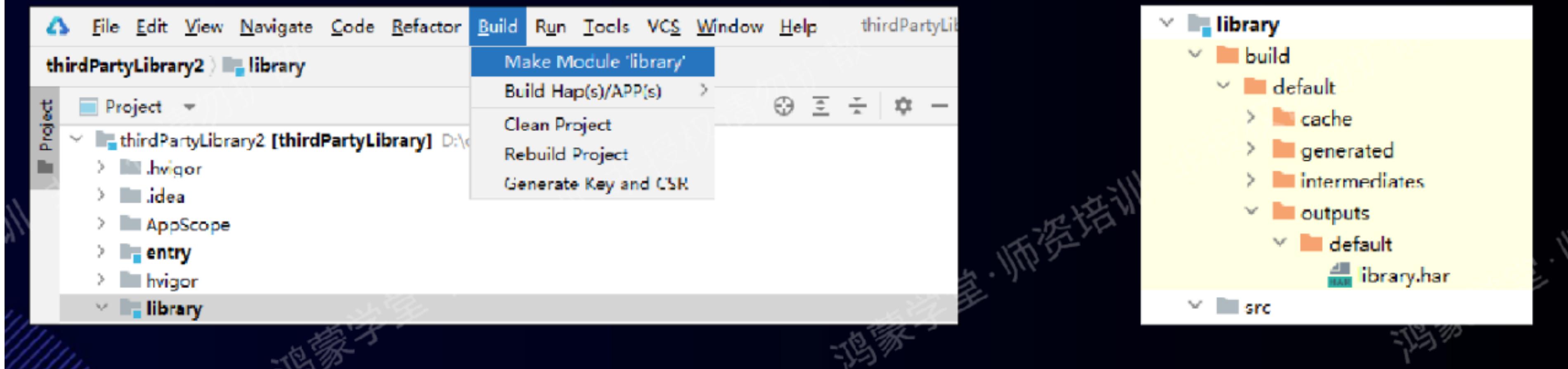
<https://gitee.com/openharmony-sig/dataORM>

4. 参考javamail

[https://gitee.com/openharmony-tpc/ohos\\_mail](https://gitee.com/openharmony-tpc/ohos_mail)

# 三方库开发移植

开发完Static Library模块后，选中模块名，然后通过DevEco Studio菜单栏的Build > Make Module \${libraryName}进行编译构建，生成三方库。三方库可用于本地工程其它模块的引用，或将三方库上传至ohpm仓库，供其他开发者下载使用。编译构建的三方库可在模块下的build目录下获取，包格式为har包。



# 编译

发布三方库到中心仓:  
<https://docs.qq.com/doc/DRHNIY3RpbWZMREVI>

**Step 1**  
利用工具 ssh-keygen 生成公、私钥，并配置公钥到中心仓“个人中心”

**Step 2**  
ohpm config 配置私钥路径和 publish\_id

**Step 3**  
ohpm publish my\_library.har

**更多参考:**  
<https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V4/har-publish-0000001597973129-V4>



The screenshot shows the details page for the 'axios' library in the HarmonyOS package manager. The library version is 1.0.4, released on 2022-09-28 10:18:14. It has 1 dependency and 5 versions. The description states: 'Axios 是一个基于 promise 的网络请求库，可以运行 nodejs 和浏览器中。本库基于 Axios 原库进行适配，使其可以在 HarmonyOS 上运行。' The page also lists the author (openourcecc), license (Apache License 2.0), and provides links for the main page and repository.

# 发布



## 发布三方库到私仓:

ohpm-repo提供了web界面和命令行两种方式，发布三方库到私仓。

- 在Web页面用管理员账号登录私仓，在个人中心 > 仓库管理中，点击管理三方包 > 上传三方包。



- 命令行工具发布，同中心仓发布流程，主要有以下步骤：

- 1、公私钥配置
- 2、发布码配置
- 3、发布仓库设置
- 4、编译打包并发布

## 更多参考:

<https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V4/ohpm-repo-0000001597733153-V4>

[https://ohpm.openharmony.cn/#/cn/help/ohpmrepository\\_introduction](https://ohpm.openharmony.cn/#/cn/help/ohpmrepository_introduction)



# 发布

三方库中心仓

三方库包管理工具 ohpm

开源三方库

业务代码

HarmonyOS应用

方法一、在IDE中运行命令行  
entry> ohpm install @ohos/axios

方法二、在IDE中编辑配置文件

"dependencies": {  
 "@ohos/axios": "^2.0.2",  
 "@ohos/aki": "1.0.5",  
 "@ohos/imageknife": "2.0.8"  
}

Fix type:  
image knife  
image-knife  
Save 'imageknife' to dictionary  
Sync Now  
Try Again  
Inject language or reference  
Sort properties alphabetically

entry  
oh\_modules  
ohpm  
@ohos  
axios  
lottie  
pullrefresh  
src  
main  
ohosTest  
.gitignore  
build-profile.json5  
hvigorfile.ts  
oh-package.json5  
oh-package-lock.json5

axios (网络库)

ohpm 指导：  
<https://developer.harmonyos.com/cn/docs/documentation/doc-guides-V4/ide-command-line-ohpm-0000001490235312-V4>

HUAWEI

# 发布

鸿蒙学堂·师资培训 非授权请勿使用

https://ohpm.openharmony.cn/#/cn/home

The screenshot shows the homepage of the OpenHarmony Component Hub. At the top, there is a search bar and a navigation bar with tabs: Home, Components, Examples, API, DevEco DevCloud, DevCloud, and Help.

**最受欢迎的三方库 >**

- @ohos/ax-cc**: A class for promise-based socket communication between OpenHarmony Agent and application, supporting direct interaction with OpenHarmony Native API.
- @ohos/zxing**: A library for barcode scanning, decoding, and encoding, supporting various formats like QR code, PDF417, Data Matrix, and more.
- @ohos/imageknife**: An image processing library based on OpenHarmony's built-in image compression engine, supporting image rotation, cropping, and more.
- @ohos/valdi**: A library for interacting with the RenderKit framework, providing support for ARKit-like features in OpenHarmony Native applications.

**最流行的三方库 >**

- @ohos/lottie**: A library for displaying Lottie animations, supporting both JSON and SVG formats.
- @ohos/disklrucache**: A memory management library for disk-based LRU cache.
- @ohos/svg**: A library for rendering SVG graphics, supporting both JSON and XML formats.
- @ohos/gpu-transform**: A library based on the OpenHarmony system, allowing users to visualize image blur, Mosaic, screen, and other transformation effects through GPU.

**最新发布的三方库 >**

- @ohos/mpchart**: A charting library for mobile applications, supporting various chart types like line, bar, and pie charts.
- @ohos/gif-drawable**: A library for displaying Gif images as drawables in OpenHarmony applications.
- @ohos/imageknife**: An image processing library based on OpenHarmony's built-in image compression engine, supporting image rotation, cropping, and more.
- flate**: A high-performance decompression library for Gzip.

**鸿蒙学堂·师资培训 非授权请勿使用**

**HUAWEI**

# 中心仓查找

The screenshot shows a Gitee repository page titled "三方组件资源汇总" (Three-party Component Resource Summary). The page content includes a brief introduction, a section for "JS/ArkTS Language", and a detailed "UI" section listing several open-source projects:

- JS/ArkTS语言**
- UI**
  - @ohos/pulltorefresh - 支持设置内嵌动画的各种属性，支持设置自定义动画的下拉刷新、上拉加载组件
  - @ohos/textlayoutbuilder - TextLayoutBuilder是一个可定制任意样式的文本构建工具，包括字体间距、大小、颜色、布局方式，富文本高亮显示等
  - @ohos/overscroll-decor - UI滚动组件
  - @ohos/mpchart - mpchart是一个包含各种类型图表的图库，主要用在业务数据统计，例如销售数据走势图，股价走势图等场景中使用，方便开发者快速实现图表UI，mpchart主要包含折线图、柱形图、饼状图、蜡烛图、气泡图、雷达图等自定义图表库

On the right side, there is a sidebar titled "近期动态" (Recent Activity) showing five recent commits or pull requests:

- 5天前更改了任务 #17117A API10的SDK新版本API变更，旧版4GDI无法编译过的状态为已完成
- 5天前提交了新的提交到 master 分支，/497545...edel55e
- 5天前合并了 PR #106 修改API10 SDK版本
- 6天前评论了 PR #106 修改API10 SDK版本
- 6天前评论了 PR #106 修改API10 SDK版本

At the bottom right of the page is a QR code.

[https://gitee.com/openharmony-tpc/tpc\\_resource](https://gitee.com/openharmony-tpc/tpc_resource)

# 社区查找

# 实践案例：ThirdPartyLibrary

# NDK开发

NDK (Native Development Kit) 是HarmonyOS SDK提供的Native API、相应编译脚本和编译工具链的集合，方便开发者使用C或C++语言实现应用的关键功能。NDK只覆盖了HarmonyOS一些基础的底层能力，如C运行时基础库libc、图形库、窗口系统、多媒体、压缩库、面向ArkTS/JS与C跨语言的Node-API等，并没有提供ArkTS/JS API的完整能力。

运行态，开发者可以使用NDK中的Node-API接口，访问、创建、操作JS对象；也允许JS对象使用Native动态库。

## NDK适用场景

**适合使用NDK的场景：应用涉及如下场景时，适合采用NDK开发**

- 性能敏感的场景，如游戏、物理模拟等计算密集型场景。
- 需要复用已有C或C++库的场景。
- 需要针对CPU特性进行专项定制库的场景，如Neon加速。

**不建议使用NDK的场景：应用涉及如下场景时，不建议采用NDK开发**

- 纯C或C++的应用。
- 希望在尽可能多的HarmonyOS设备上保持兼容的应用。

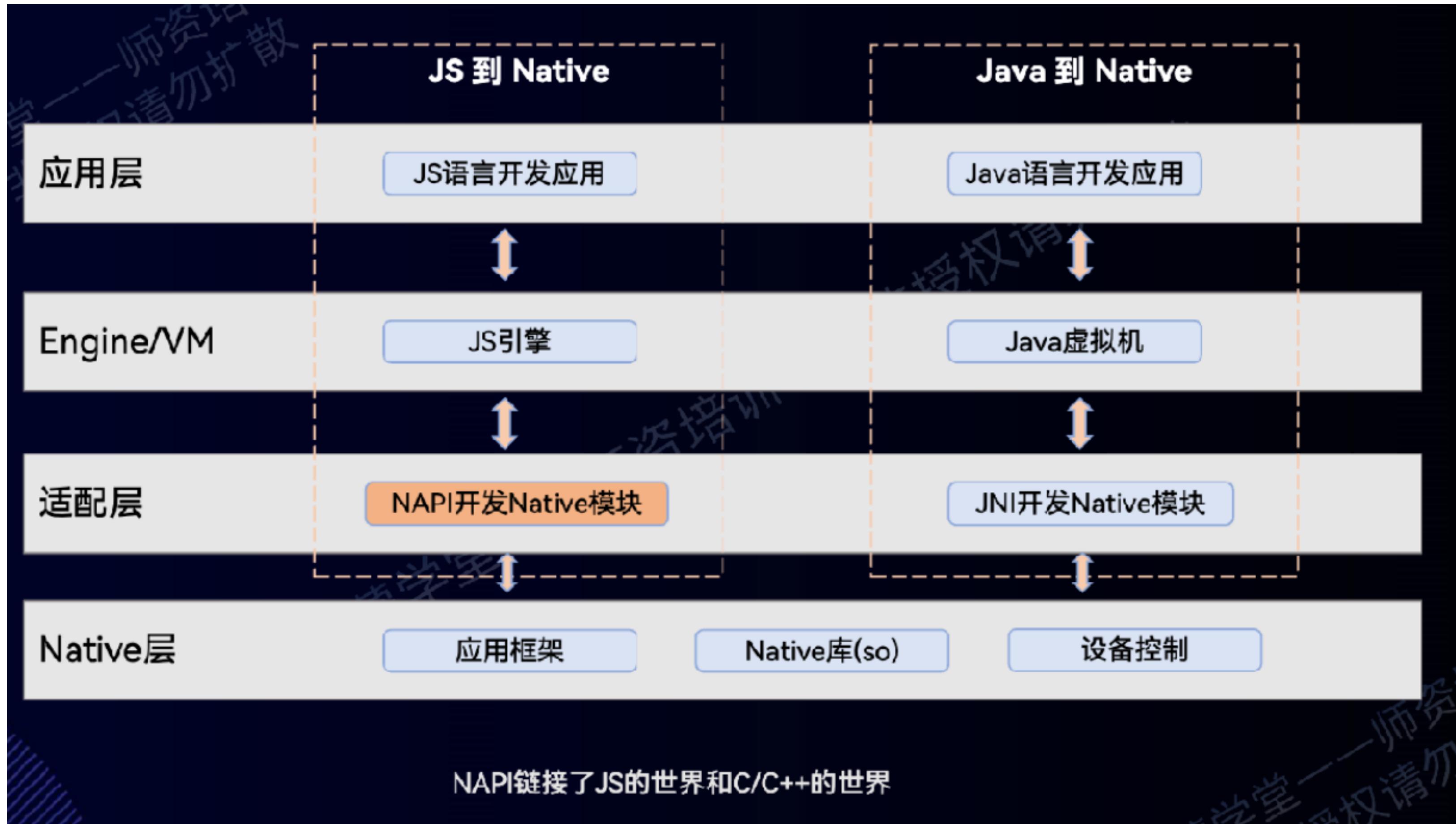
## NDK基本概念

- **Node-API**

曾用名NAPI，是HarmonyOS中提供ArkTS/JS与C/C++跨语言调用的接口，是NDK接口中的一部分。该接口是在Node.js提供的Node-API基础上扩展而来，但与Node.js中的Node-API不完全兼容。

- **C API**

HarmonyOS NDK的曾用名，不再使用。



# 对比JNI

# NDK目录简介

- build目录：放置预定义的toolchain脚本文件ohos.toolchain.cmake

```
build
└-- cmake
 |-- ohos.toolchain.cmake
 |-- sdk_native_platforms.cmake
```

CMake编译时需要读取该文件中的默认值，比如编译器架构、C++库链接方式等，因此在编译时会通过CMAKE\_TOOLCHAIN\_FILE指出该文件的路径，便于CMake在编译时定位到该文件。

- build-tools文件夹：放置NDK提供的编译工具

```
● # 键入下一行命令查看CMake的版本
● cmake -version
● # 结果
● cmake version 3.16.5
● CMake suite maintained and supported by Kitware (kitware.com/cmake).
```

- 

- llvm文件夹：放置NDK提供的编译器

```
clang clangd llc lld lldb lldb-verify lldb-modextract lldb-readobj scan-build
clang++ dsymutil llc lld lldb-argdumper lldb-config lldb-rm lldb-size scan-view
clang-10 git-clang-format llc lld lldb-mi lldb-cov lldb-objcopy lldb-strings
clang-check ld.1ld lldb-server lld lldb-cxxfilt lldb-cxxfilt lldb-objdump lldb-strip
clang-cl ld64.1ld lldb-dis lld lldb-addr2line lldb-dis lldb-profdata lldb-symbolizer
clang-format 1ld lldb-ar lld lldb-lib lldb-ranlib lldb-readelf sancov
clang-tidy 1ld-link lldb-as lld lldb-link lldb-link lldb-readelf sanstats
```

| 模块         | 模块简介                                                |
|------------|-----------------------------------------------------|
| 标准C库       | 以musl为基础提供的标准C库接口。                                  |
| 标准C++库     | C++运行时库libc++_shared。                               |
| 日志         | 打印日志到系统的HiLog接口。                                    |
| Node-API   | 当需要实现ArkTS/JS和C/C++之间的交互时，可以使用Node-API。             |
| libuv      | 三方异步IO库。                                            |
| zlib       | zlib库，提供基本的数据压缩、解压接口。                               |
| Rawfile    | 应用资源访问接口，可以读取应用中打包的各种资源。                            |
| XComponent | ArkUI XComponent组件提供surface与触屏事件等接口，方便开发者开发高性能图形应用。 |
| Drawing    | 系统提供的2D图形库，可以在surface进行绘制。                          |
| OpenGL     | 系统提供的OpenGL 3D图形接口。                                 |
| OpenSL ES  | 用于2D、3D音频加速的接口库。                                    |

# NDK常用模块

# NDK工程构建概述

HarmonyOS NDK默认使用CMake作为构建系统，随包提供了符合HarmonyOS工具链的基础配置文件 [ohos.toolchain.cmake](#)，用于预定义CMake变量来简化开发者配置。

常用的NDK工程构建方式有：

- 从源码构建

源码构建也有不同方式：

- 可以使用DevEco Studio提供的C++应用模板，用 [DevEco Studio](#) 来编译构建
- 也可以[使用命令行CMake来编译构建](#)

- [使用预构建库构建](#)

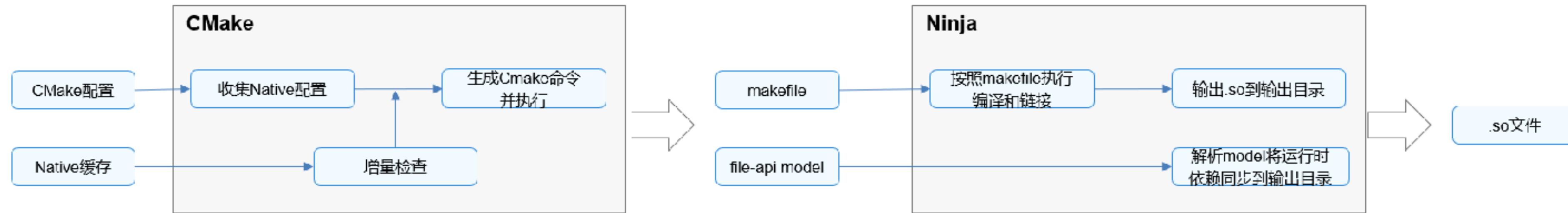
## ohos.toolchain.cmake简介

`ohos.toolchain.cmake`是HarmonyOS NDK提供给CMake的toolchain脚本，里面预定义了编译HarmonyOS应用需要设置的编译参数，如交叉编译设备的目标、C++运行时库的链接方式等；这些参数在调用CMake命令时，可以从命令行传入，来改变默认编译链接行为。此文件中的常用参数见下表。

| 参数            | 类型                           | 说明                                                                                                                                                         |
|---------------|------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| OHOS_STL      | c++_shared/c++_static        | libc++的链接方式。默认为c++_shared。<br>c++_shared表示采用动态链接<br>libc++_shared.so；c++_static表示采用静态链接libc++_static.a。<br>由于C++运行时中存在一些全局变量，因此同一应用中的全部Native库需要采用相同的链接方式。 |
| OHOS_ARCH     | armeabi-v7a/arm64-v8a/x86_64 | 设置当前Native交叉编译的目标架构，当前支持的架构为armeabi-v7a/arm64-v8a/x86_64。                                                                                                  |
| OHOS_PLATFORM | OHOS                         | 选择平台。当前只支持HarmonyOS平台。                                                                                                                                     |

上述参数最终会控制Clang的交叉编译命令，产生合适的命令参数。

- `--target={arch}-linux-ohos`参数，通知编译器生成相应架构下符合HarmonyOS ABI的二进制文件。
- `--sysroot={ndk_root}/sysroot`参数，告知编译器HarmonyOS系统头文件的所在位置。



# 编译过程

核心编译过程如下：

- 根据CMake配置脚本以及build-profile.json5中配置的externalNativeOptions构建参数，与缓存中的配置比对后，生成CMake命令并执行CMake。
- 执行Ninja，按照makefile执行编译和链接，将生成的.so以及运行时依赖的.so同步到输出目录，完成构建过程。

通过DevEco Studio提供的应用模板，可以快速生成CMake构建脚本模板，并在build-profile.json5中指定相关编译构建参数

# CMakeLists.txt脚本

通过DevEco Studio模板工程创建的NDK工程中，包含默认生成的CMakeLists.txt脚本，如下所示：

```
• # the minimum version of CMake.
• cmake_minimum_required(VERSION 3.4.1)
• project(MyApplication)
•
• # 定义一个变量，并赋值为当前模块cpp目录
• set(NATIVE RENDER_ROOT_PATH ${CMAKE_CURRENT_SOURCE_DIR})
•
• # 添加头文件.h目录，包括cpp, cpp/include, 告诉cmake去这里找到代码引入的头文件
• include_directories(${NATIVE_RENDER_ROOT_PATH}
• ${NATIVE_RENDER_ROOT_PATH}/include)
•
• # 声明一个产物libentry.so, SHARED表示产物为动态库, hello.cpp为产物的源代码
• add_library(entry SHARED hello.cpp)
•
• # 声明产物entry链接时需要的三方库libace_napi.z.so
• # 这里直接写三方库的名称是因为它是在ndk中，已在链接寻址路径中，无需额外声明
• target_link_libraries(entry PUBLIC libace_napi.z.so)
```

默认的CMakeLists.txt脚本中添加了编译所需的源代码、头文件以及三方库，开发者可根据实际工程添加自定义编译参数、函数声明、简单的逻辑控制等。

# externalNativeOptions

模块级build-profile.json5中externalNativeOptions参数是NDK工程C/C++文件编译配置的入口，可以通过path指定CMake脚本路径、arguments配置CMake参数、cppFlags配置C++编译器参数、abiFilters配置编译架构等。

externalNativeOptions具体参数说明如下表所示。

```
• "apiType": "stageMode",
• "buildOption": {
 • "arkOptions": {
 • },
 • },
• "externalNativeOptions": {
 • "path": "./src/main/cpp/CMakeLists.txt",
 • "arguments": "",
 • "cppFlags": "",
 • "abiFilters": [
 • "arm64-v8a",
 • "x86_64"
],
 },
}
```

| 配置项        | 类型     | 说明                                                                            |
|------------|--------|-------------------------------------------------------------------------------|
| path       | string | CMake构建脚本地址，即CMakeLists.txt文件地址。                                              |
| abiFilters | array  | 本机的ABI编译环境，包括：<br>- arm64-v8a<br>- x86_64<br>如不配置该参数，编译时默认编译出arm64-v8a架构相关so。 |
| arguments  | string | CMake编译参数。                                                                    |
| cppFlags   | string | C++编译器参数。                                                                     |

# 在NDK工程中使用预构建库

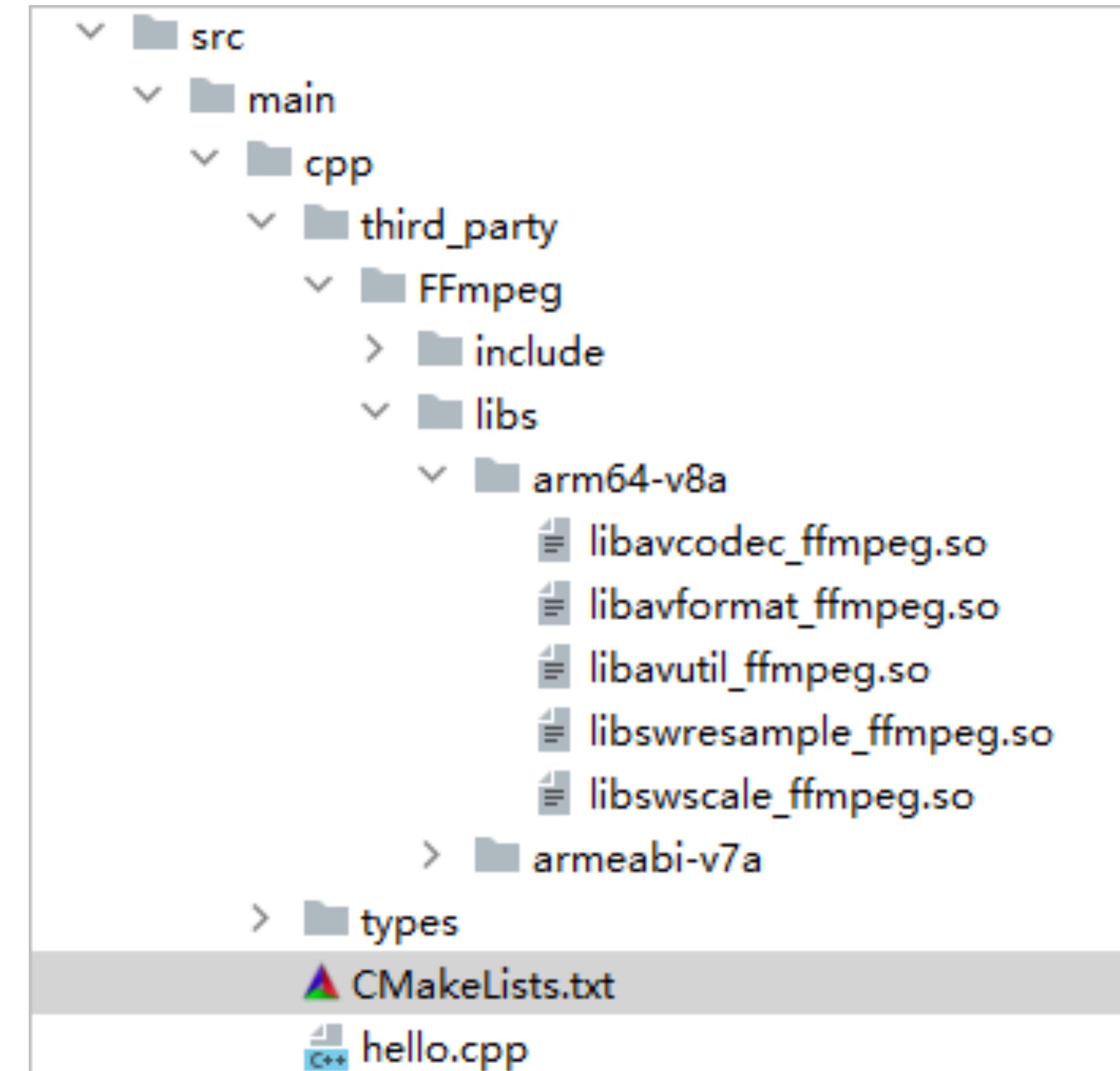
可以通过直接将预构建的库文件复制到项目文件中，来使用预构建库。例如在项目中需要使用预构建库libavcodec\_ffmpeg.so，其开发态存放路径如下图所示：

在模块的CMakeLists.txt编译脚本中通过add\_library添加所需的预构建库，并声明预构建库路径等信息后，可以在target\_link\_libraries中声明链接该预构建库，脚本示例如下所示：

```
• add_library(library SHARED hello.cpp)
•
• add_library(avcodec_ffmpeg SHARED IMPORTED)
• set_target_properties(avcodec_ffmpeg
• PROPERTIES
• IMPORTED_LOCATION ${CMAKE_CURRENT_SOURCE_DIR}/
• third_party/FFmpeg/libs/${OHOS_ARCH}/libavcodec_ffmpeg.so)
•
• target_link_libraries(library PUBLIC libace_napi.z.so
• avcodec_ffmpeg)
```

在模块的CMakeLists.txt编译脚本中添加include\_directories：

```
• include_directories(
• ...
• ${CMAKE_CURRENT_SOURCE_DIR}/third_party/FFmpeg/include
•)
```



# 使用HAR中集成的预构建库

## 使用远程依赖HAR中集成的预构建库

当使用远程依赖HAR中集成的预构建库时，CMakeLists.txt文件中引用脚本如下所示：

```
• set(DEPENDENCY_PATH ${CMAKE_CURRENT_SOURCE_DIR}/../../../../oh_modules)
• add_library(library SHARED IMPORTED)
• set_target_properties(library
• PROPERTIES
• IMPORTED_LOCATION ${DEPENDENCY_PATH}/library/libs/${OHOS_ARCH}/liblibrary.so)
• add_library(entry SHARED hello.cpp)
• target_link_libraries(entry PUBLIC libace_napi.z library)
```

## 使用本地HAR中集成的预构建库

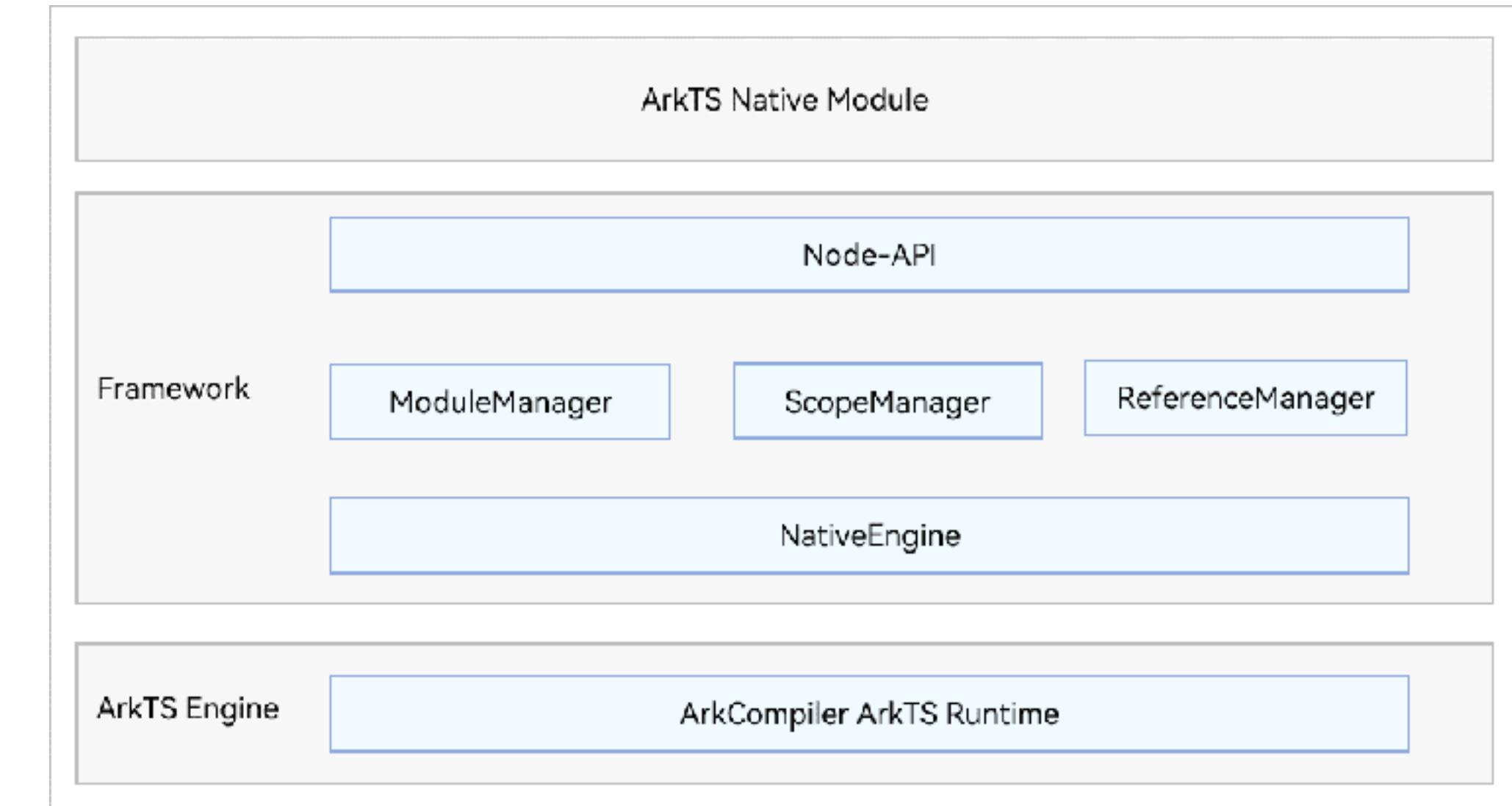
当使用本地HAR中集成的预构建库时，CMakeLists.txt文件中引用脚本如下所示：

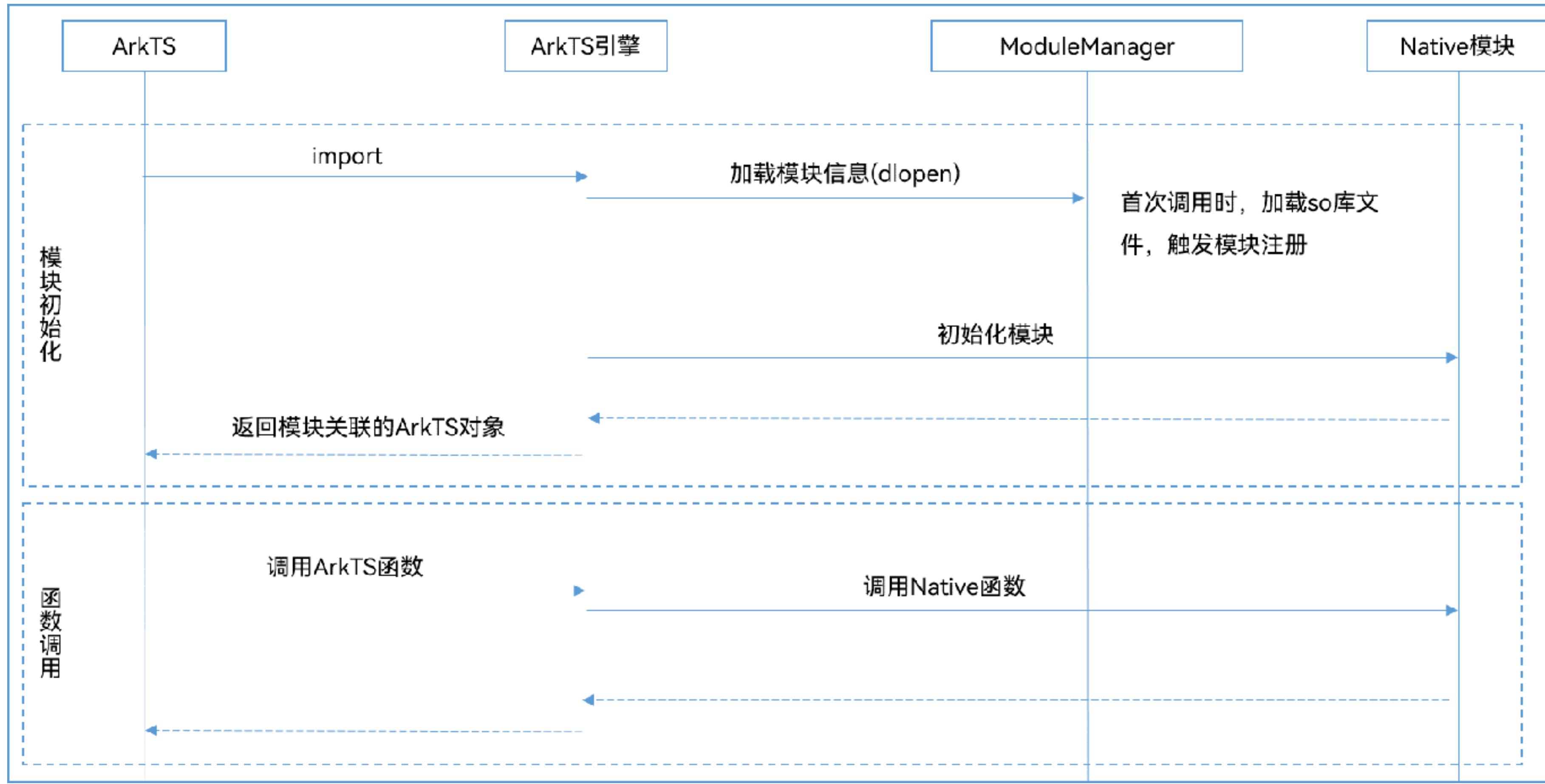
```
• set(LIBRARY_DIR "${NATIVERENDER_ROOT_PATH}/../../../../library/build/default/intermediates/libs/default/${OHOS_ARCH}/")
• add_library(library SHARED IMPORTED)
• set_target_properties(library
• PROPERTIES
• IMPORTED_LOCATION ${LIBRARY_DIR}/liblibrary.so)
• add_library(entry SHARED hello.cpp)
• target_link_libraries(entry PUBLIC libace_napi.z.so)
```

# Node-API

一般情况下HarmonyOS应用开发使用ArkTS/JS语言，但部分场景由于性能、效率等要求，比如游戏、物理模拟等，需要依赖使用现有的C/C++库。Node-API规范封装了I/O、CPU密集型、OS底层等能力并对外暴露ArkTS/JS接口，从而实现ArkTS/JS和C/C++的交互。主要场景如下：

- 系统可以将框架层丰富的模块功能通过ArkTS/JS接口开放给上层应用。
- 应用开发者也可以选择将一些对性能、底层系统调用有要求的核心功能用C/C++封装实现，再通过ArkTS/JS接口使用，提高应用本身的执行效率
- Native Module：开发者使用Node-API开发的模块，用于在ArkTS侧导入使用。
- Node-API：实现ArkTS与C/C++交互的逻辑。
- ModuleManager：Native模块管理，包括加载、查找等。
- ScopeManager：管理napi\_value的生命周期。
- ReferenceManager：管理napi\_ref的生命周期。
- NativeEngine：ArkTS引擎抽象层，统一ArkTS引擎在Node-API层的接口行为。
- ArkCompiler ArkTS Runtime：ArkTS运行时。





# Node-API的关键交互流程

ArkTS和C++之间的交互流程，主要分为以下两步：

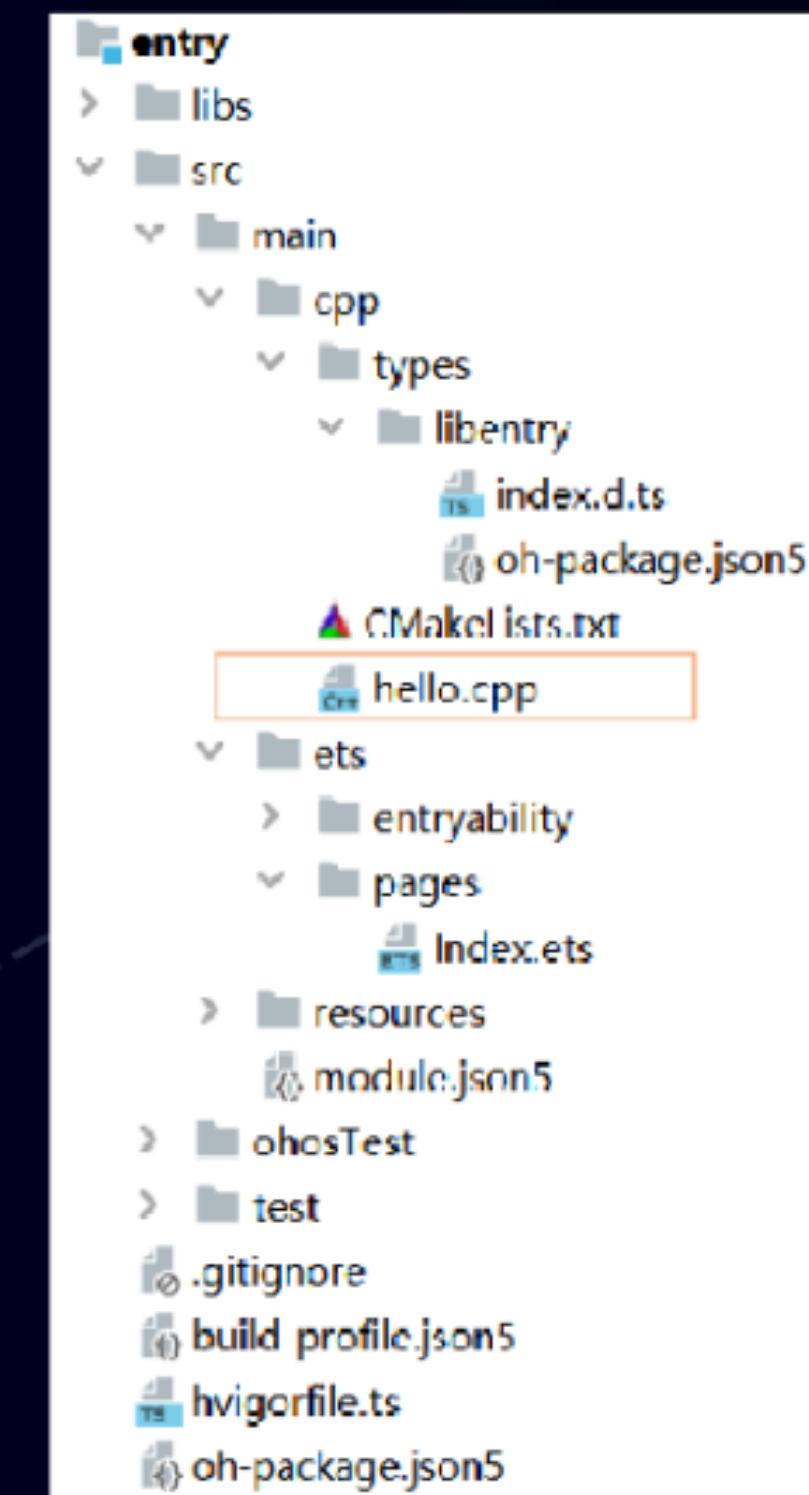
- **初始化阶段：**当ArkTS侧在import一个Native模块时，ArkTS引擎会调用ModuleManager加载模块对应的so及其依赖。首次加载时会触发模块的注册，将模块定义的方法属性挂载到exports对象上并返回该对象。
- **调用阶段：**当ArkTS侧通过上述import返回的对象调用方法时，ArkTS引擎会找到并调用对应的C/C++方法。

DecEco提供了Native C++模板，可以基于模板完成开发。



# 开发流程

- 鸿蒙学堂——师资培训  
未经许可请勿扩散
1. NAPI开发Native模块
    - ① 设置模块注册信息
    - ② 配置模块初始化信息
    - ③ 基于NAPI开发业务功能
  2. C++编译配置 (CMakeList)
  3. module构建配置中增加Native构建信息
  4. TS声明Native库支持的接口
  5. 修改模块配置，增加对TS接口依赖
  6. ETS文件中使用NAPI



```
EXTERN_C_START
static napi_value Init(napi_env env, napi_value exports)
{
 napi_property_descriptor desc[] = {
 {"add", nullptr, Add, nullptr, nullptr, napi_default, nullptr }
 };
 napi_define_properties(env, exports, sizeof(desc) / sizeof(desc[0]), desc);
 return exports;
}
EXTERN_C_END

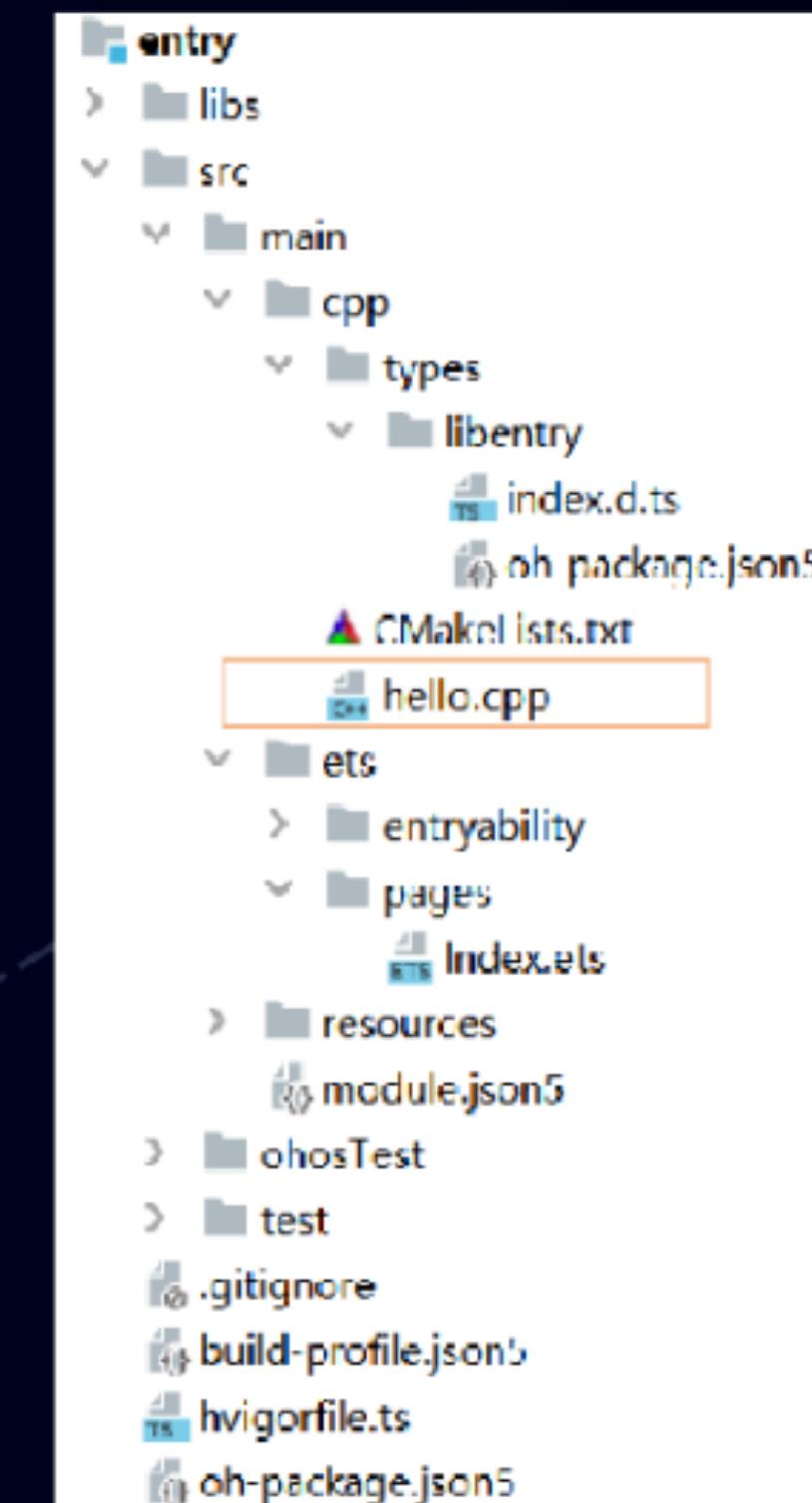
static napi_module demoModule = {
 .nm_version = 1,
 .nm_flags = 0,
 .nm_filename = nullptr,
 .nm_register_func = Init,
 .nm_modname = "libentry",
 .nm_priv = ((void*)0),
 .reserved = { 0 },
};

extern "C" __attribute__((constructor)) void
RegisterModule(void)
{
 napi_module_register(&demoModule);
}
```

注册模块信息

# 注册模块信息

- 鸿蒙学堂——师资培训——  
未经许可请勿扩散
1. NAPI开发Native模块
    - ① 设置模块注册信息
    - ② 配置模块初始化信息
    - ③ 基于NAPI开发业务功能
  2. C++编译配置 (CMakeList)
  3. module构建配置中增加Native构建信息
  4. TS声明Native库支持的接口
  5. 修改模块配置，增加对TS接口依赖
  6. ETS文件中使用NAPI



```
#include "napi/native_api.h"

static napi_value Add(napi_env env, napi_callback_info info)
{
 //
}

EXTERN_C_START
static napi_value Init(napi_env env, napi_value exports)
{
 napi_property_descriptor desc[] = {
 { "add", nullptr, Add, nullptr, nullptr, napi_default, nullptr } };
 napi_define_properties(env, exports, sizeof(desc) / sizeof(desc[0]), desc);
 return exports;
}
EXTERN_C_END

static napi_module demoModule = {
 .nm_version = 1,
 .nm_flags = 0,
 .nm_filename = nullptr,
 .nm_register_func = Init, —————> 初始化函数
 .nm_modname = "libentry",
 .nm_priv = ((void*)0),
 .reserved = { 0 },
};
```

Native函数

JS接口名称

初始化函数

# 配置模块初始化信息

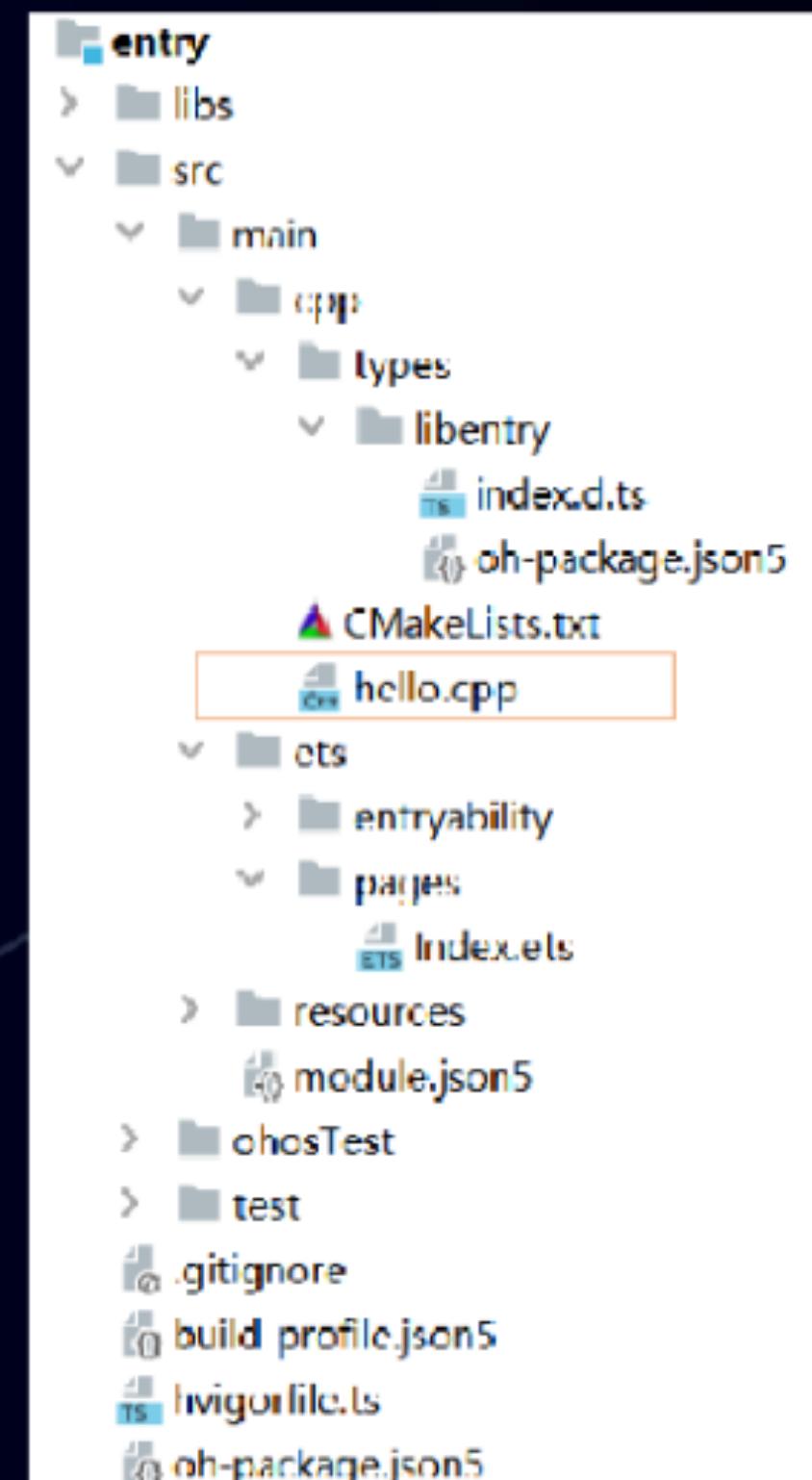
# 鸿蒙学堂——师资培训

## 1. NAPI开发Native模块

- ① 设置模块注册信息
- ② 配置模块初始化信息
- ③ 基于NAPI开发业务功能

## 2. C++编译配置 (CMakeList)

- 3. module构建配置中增加Native构建信息
- 4. TS声明Native库支持的接口
- 5. 修改模块配置，增加对TS接口依赖
- 6. ETS文件中使用NAPI



```
#include "napi/native_api.h"

static napi_value Add(napi_env env, napi_callback_info info)
{
 size_t requireArgc = 2;
 size_t argc = 2;
 napi_value args[2] = {nullptr};

 napi_get_cb_info(env, info, &argc, args, nullptr, nullptr);
 // TODO check args count

 napi_valuetype valuetype0;
 napi_typeof(env, args[0], &valuetype0);
 // TODO check arg type

 napi_valuetype valuetype1;
 napi_typeof(env, args[1], &valuetype1);
 // TODO check arg type

 double value0;
 napi_get_value_double(env, args[0], &value0);

 double value1;
 napi_get_value_double(env, args[1], &value1);

 napi_value sum;
 napi_create_double(env, value0 + value1, &sum);

 return sum;
}
```

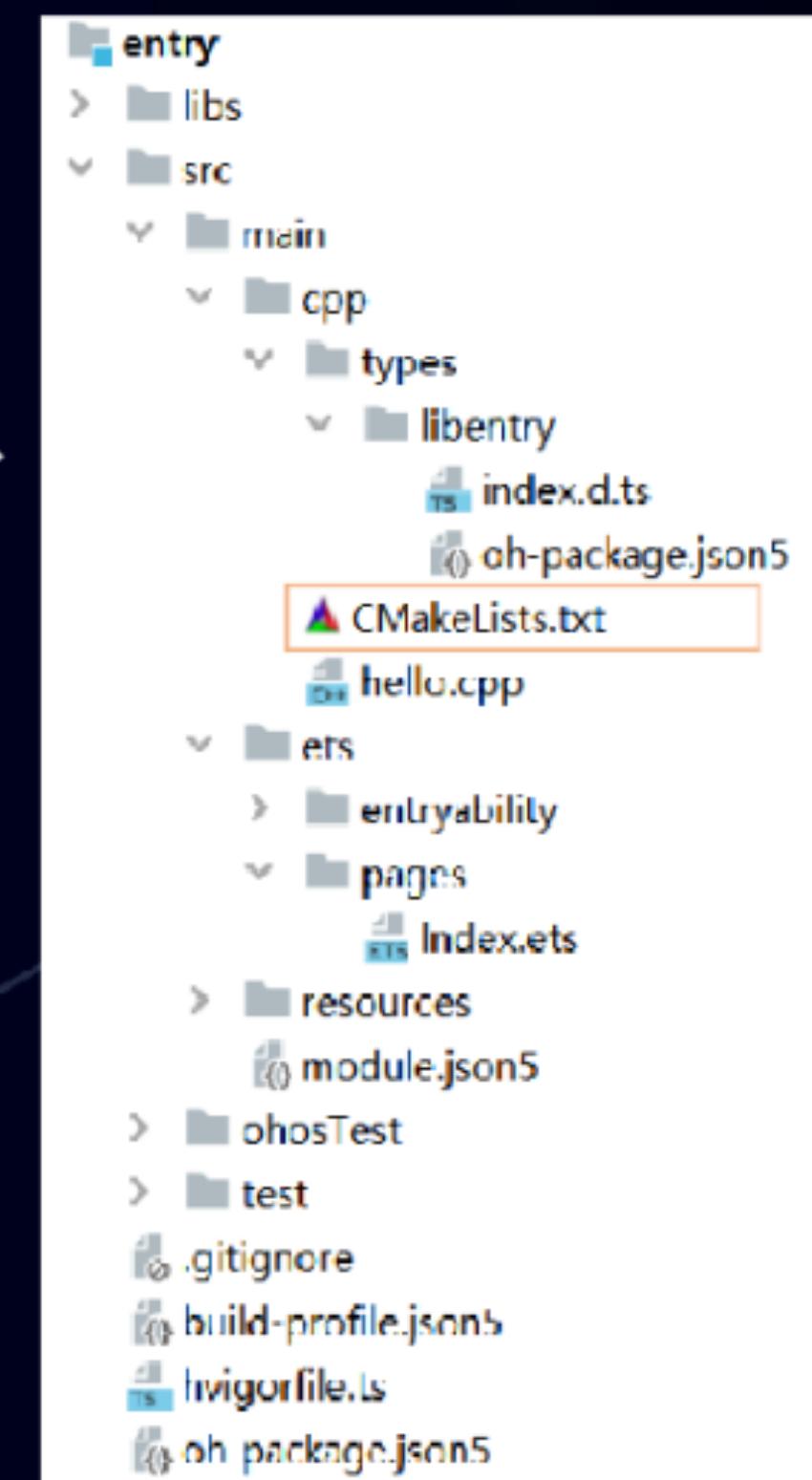
入参检查

参数提取

结果转换

# 业务功能

1. NAPI开发Native模块
2. C++编译配置 (CMakeList)
3. module构建配置中增加Native构建信息
4. TS声明Native库支持的接口
5. 修改模块配置，增加对TS接口依赖
6. ETS文件中使用NAPI



```
the minimum version of CMake.
cmake_minimum_required(VERSION 3.4.1)
project(entry)

set(NATIVERENDER_ROOT_PATH
${CMAKE_CURRENT_SOURCE_DIR})

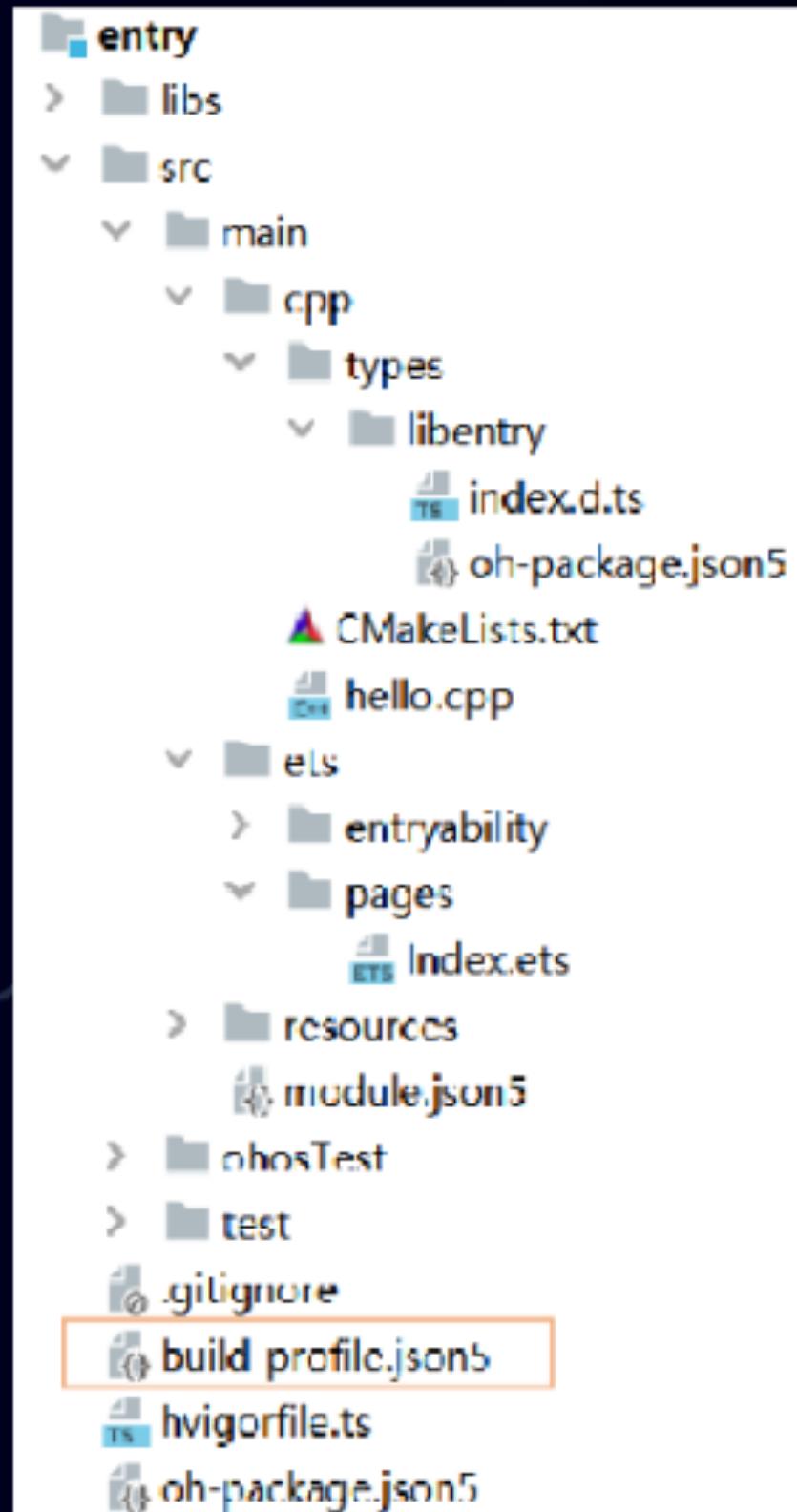
include_directories(${NATIVERENDER_ROOT_PATH}
${NATIVERENDER_ROOT_PATH}/include
)
add_library(entry SHARED hello.cpp)
target_link_libraries(entry PUBLIC libace_napi.z.so libc++.a)
```

CMakeList可以构建不同类型的目标文件：

```
add_executable(entry hello.cpp) # 生成可执行文件
add_library(entry STATIC hello.cpp) # 生成静态库
add_library(entry SHARED hello.cpp) # 生成动态库
```

# CMakeList

1. NAPI开发Native模块
2. C++编译配置 (CMakeList)
3. module构建配置中增加Native构建信息
4. TS声明Native库支持的接口
5. 修改模块配置，增加对TS接口依赖
6. ETS文件中使用NAPI

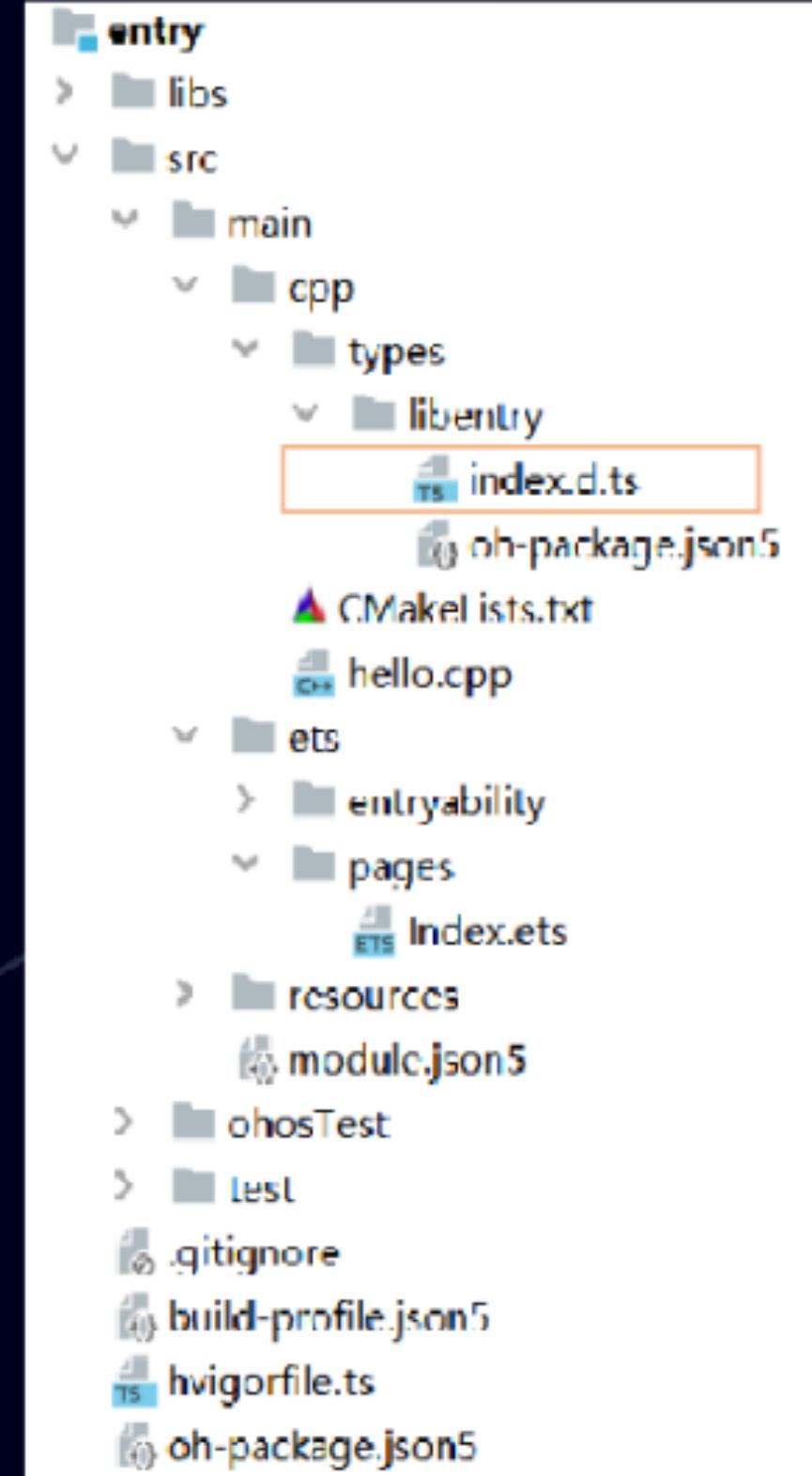


工程创建后，默认配置Native构建信息：

```
{
 "apiType": "stageMode",
 "buildOption": {
 "externalNativeOptions": {
 "path": "./src/main/cpp/CMakeLists.txt",
 "cppFlags": ""
 }
 },
 "targets": [
 {
 "name": "default"
 },
 {
 "name": "ohosTest"
 }
]
}
```

# module配置Native构建信息

1. NAPI开发Native模块
2. C++编译配置 (CMakeList)
3. module构建配置中增加Native构建信息
4. TS声明Native库支持的接口
5. 修改模块配置，增加对TS接口依赖
6. ETS文件中使用NAPI

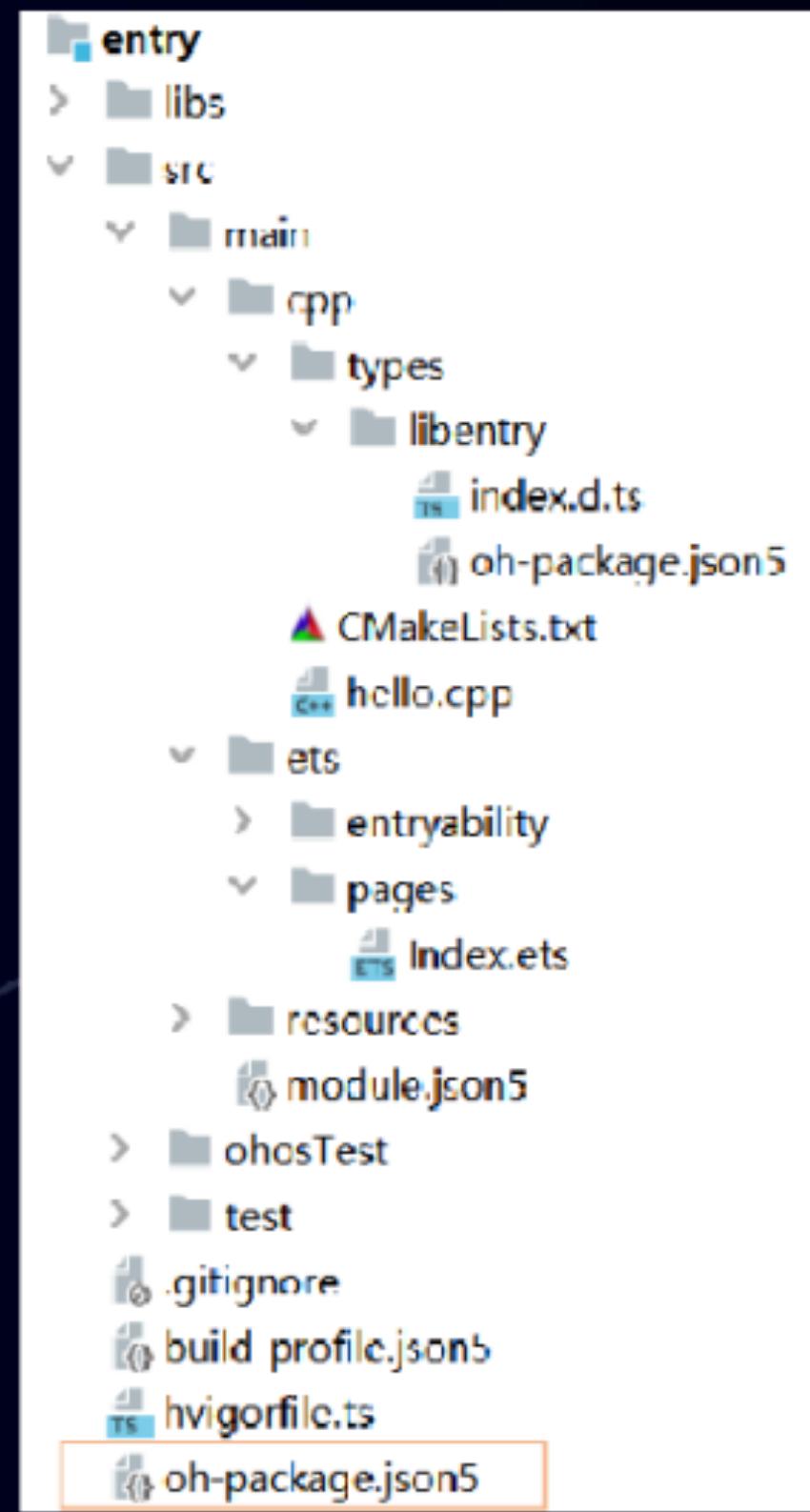


- 接口声明基于TS语法实现，TS是强类型语言。
- TS中的函数大部分和JS相同，区别在于TS会在函数参数的后面加上类型声明

```
// index.d.ts
export const add: (a: number, b: number) =>
 number;
```

# TS声明Native库支持的接口

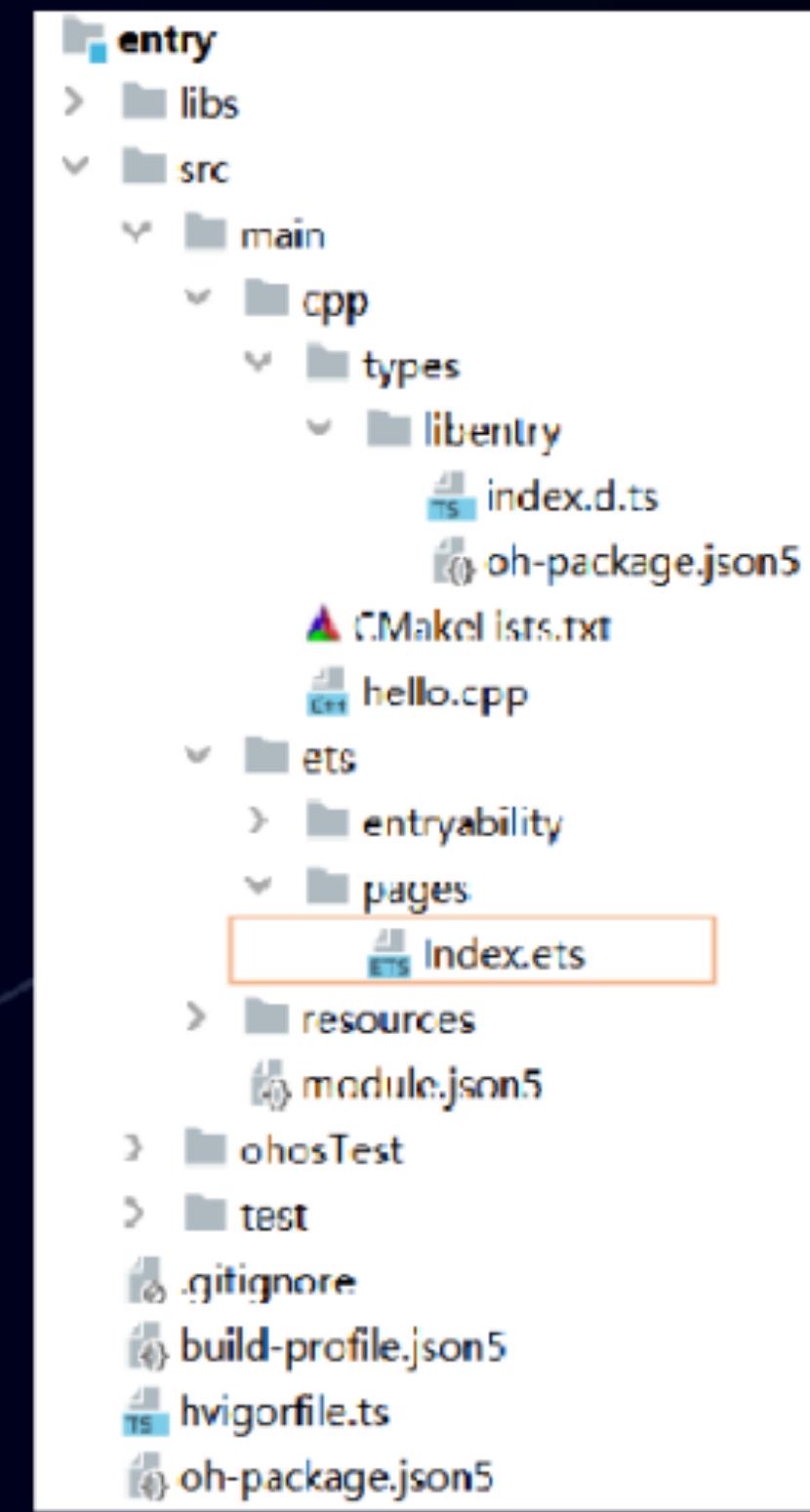
1. NAPI开发Native模块
2. C++编译配置 (CMakeList)
3. module构建配置中增加Native构建信息
4. TS声明Native库支持的接口
5. 修改模块配置，增加对TS接口依赖
6. ETS文件中使用NAPI



```
{
 "name": "entry",
 "version": "1.0.0",
 "description": "Please describe the basic information.",
 "main": "",
 "author": "",
 "license": "",
 "dependencies": {
 "libentry.so": "file:/src/main/cpp/types/libentry"
 }
}
```

# 修改模块配置，增加TS接口依赖

1. NAPI开发Native模块
2. C++编译配置 (CMakeList)
3. module构建配置中增加Native构建信息
4. TS声明Native库支持的接口
5. 修改模块配置，增加对TS接口依赖
6. ETS文件中使用NAPI



```
import testNapi from "libentry.so"

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

 build() {
 Row() {
 Column() {
 Text(this.message)
 .fontSize(50)
 .fontWeight(FontWeight.Bold)
 .onClick(() => {
 console.log("Test NAPI 2 + 3 = " + testNapi.add(2, 3));
 })
 }
 .width('100%')
 }
 .height('100%')
 }
 }
}
```

# 使用NAPI

实践案例：  
**NativeTemplateDemo**

同步调用支持带Callback，具体方式由应用开发者决定，通过是否传递Callback函数进行区分



# 开发同步函数



鸿蒙学堂——师资培训专用  
非授权请勿扩散



# 开发同步函数

同步调用不带callback: native底层业务处理完，直接将计算结果回传给应用

```
// 示例代码 index.d.ts, 提供给应用编译调用
export const add: (a: number, b: number) => number;

// 示例代码 hello.cpp, 适配层代码, 同步调用不带callback
napi_value add(napi_env env, napi_callback_info info) {
 ...
 napi_get_cb_info(env, info, &argc, args,
 nullptr, nullptr);

 double value0;
 napi_get_value_double(env, args[0], &value0);

 double value1;
 napi_get_value_double(env, args[1], &value1);

 napi_value sum;
 napi_create_double(env, add(value0, value1),
 &sum);
 return sum;
}
```

napi\_get\_cb\_info 接口获取ArkTS传入的参数

napi\_get\_value\_double 将参数从napi\_value转化为double

衔接业务代码

将参数处理后通过napi\_create\_double创建napi\_value对象返回给应用



# 开发同步函数

同步调用带callback: native底层业务处理完，回调到应用层，执行应用层定义的回调代码片段

```
// 示例代码 index.d.ts, 提供给应用编译调用
export const addSync: (a: number, b: number,
callback:(result:number) => void) => void;
// 示例代码 hello.cpp, 适配层代码, 同步调用带callback
napi_value addSync(napi_env env, napi_callback_info info) {
 ...
 napi_get_cb_info(env, info, &argc, args, &context,
nullptr);

 double value0;
 napi_get_value_double(env, args[0], &value0);

 double value1;
 napi_get_value_double(env, args[1], &value1);

 napi_value callbackArg[1] = {nullptr};

 napi_create_double(env, add(value0 , value1),
&callbackArg[0]);
 napi_value res = nullptr;
 napi_call_function(env, context, args[2], 1, callbackArg,
&res);

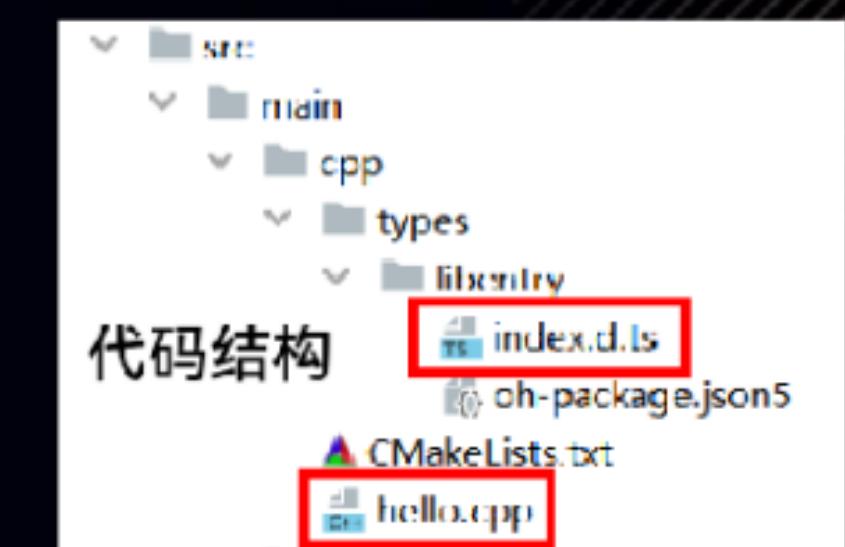
 return res;
}
```

napi\_get\_cb\_info 接口获取ArkTS传入的参数

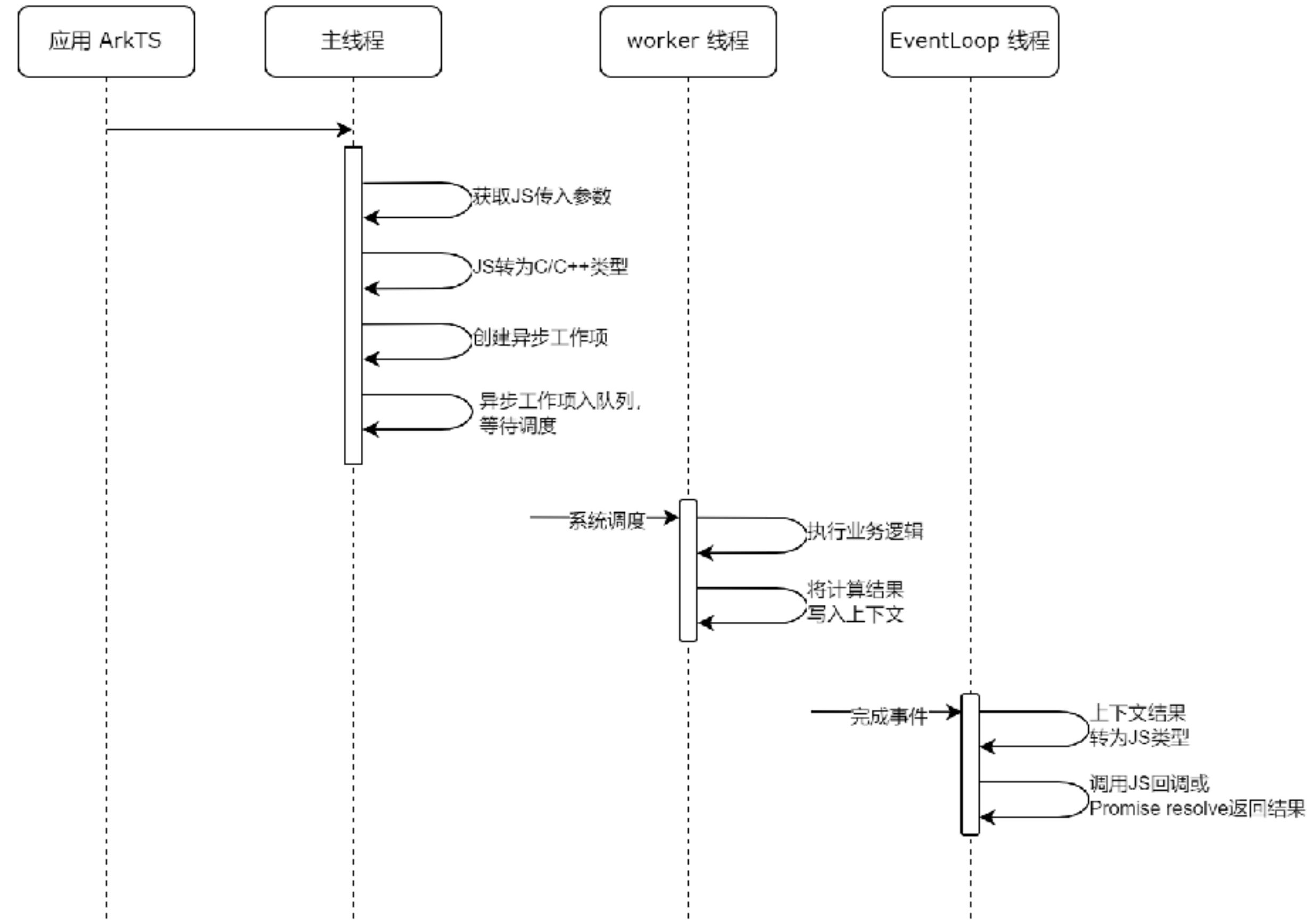
napi\_get\_value\_double 将参数从napi\_value转化为double

衔接业务代码

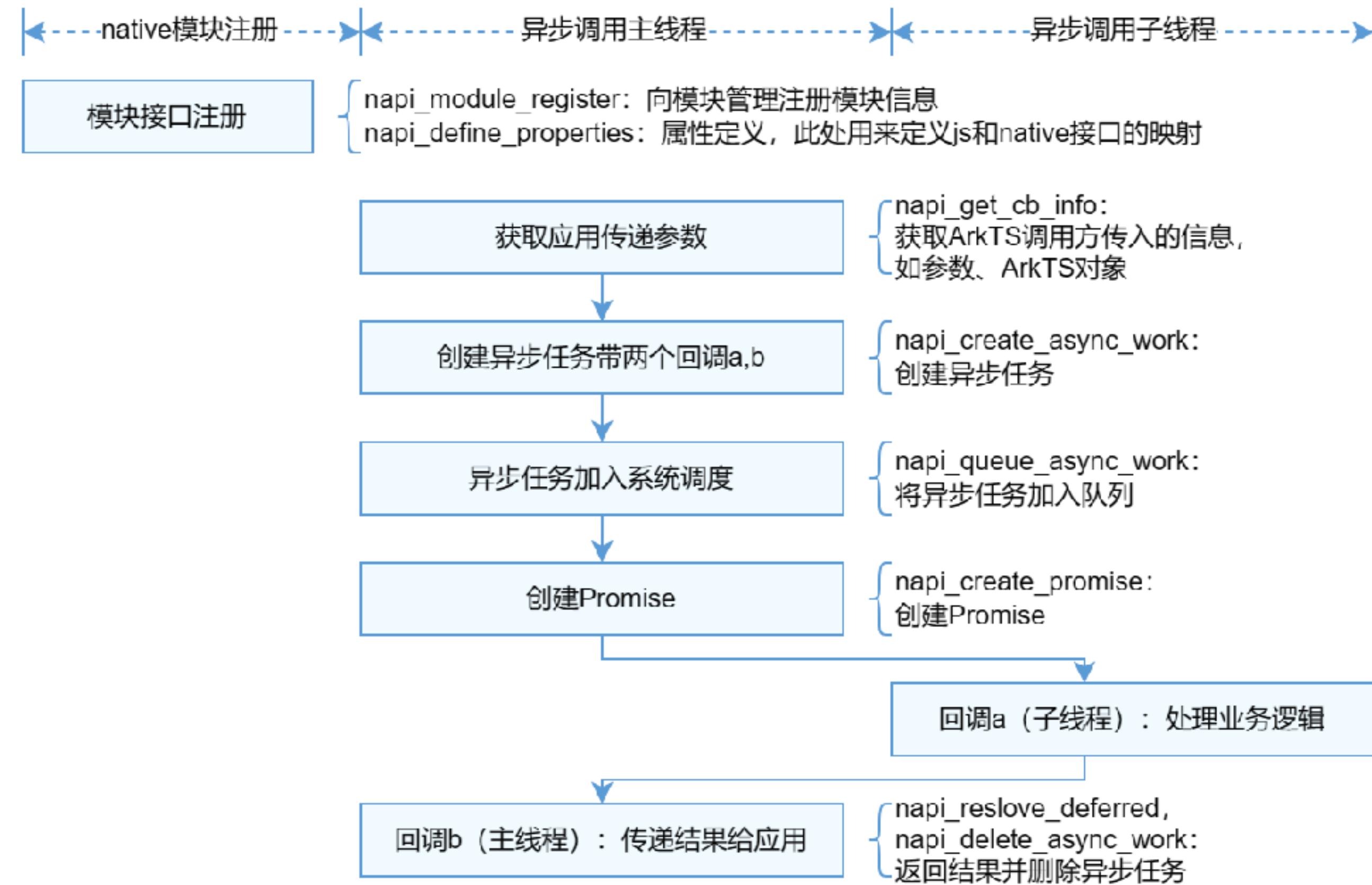
将参数处理后通过回调返回给应用



# 开发同步函数



# 开发异步函数



# 使用Promise方式示例

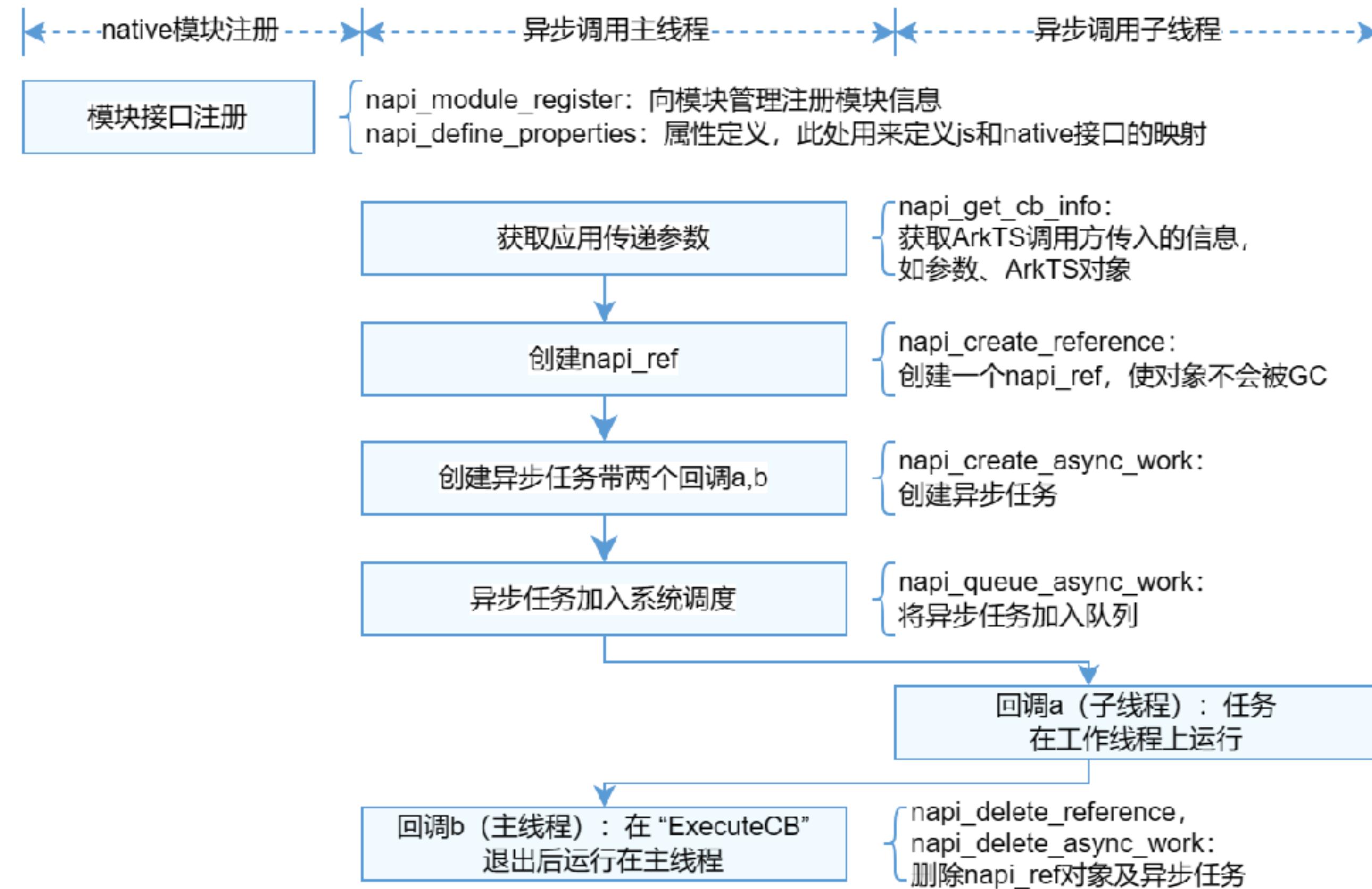
# 实例代码

- 使用napi\_create\_async\_work创建异步任务，并使用napi\_queue\_async\_work将异步任务加入队列，等待执行。

```
• struct CallbackData {
• napi_async_work asyncWork = nullptr;
• napi_deferred deferred = nullptr;
• napi_ref callback = nullptr;
• double args = 0;
• double result = 0;
• };
•
• static napi_value AsyncWork(napi_env env,
• napi_callback_info info)
• {
• size_t argc = 1;
• napi_value args[1];
• napi_get_cb_info(env, info, &argc, args,
• nullptr, nullptr);
•
• napi_value promise = nullptr;
• napi_deferred deferred = nullptr;
• napi_create_promise(env, &deferred,
• &promise);
•
• auto callbackData = new CallbackData();
• callbackData->deferred = deferred;
• napi_get_value_double(env, args[0],
• &callbackData->args);
•
• napi_value resourceName = nullptr;
• napi_create_string_utf8(env,
• "AsyncCallback", NAPI_AUTO_LENGTH,
• &resourceName);
• // 创建异步任务
• napi_create_async_work(env, nullptr,
• resourceName, ExecuteCB, CompleteCB,
• callbackData, &callbackData->asyncWork);
• // 将异步任务加入队列
```

- napi\_queue\_async\_work(env, callbackData->asyncWork);
- return promise;
- 
- 定义异步任务的第一个回调函数，该函数在工作线程中执行，处理具体的业务逻辑。
- static void ExecuteCB(napi\_env env, void \*data){  
 CallbackData \*callbackData = reinterpret\_cast<CallbackData \*>(data);  
 callbackData->result = callbackData->args;  
}
- 定义异步任务的第二个回调函数，该函数在主线程执行，将结果传递给ArkTS侧。
- static void CompleteCB(napi\_env env, napi\_status status, void \*data){  
 CallbackData \*callbackData = reinterpret\_cast<CallbackData \*>(data);  
 napi\_value result = nullptr;  
 napi\_create\_double(env, callbackData->result, &result);  
 if (callbackData->result > 0) {  
 napi\_resolve\_deferred(env, callbackData->deferred, result);  
 } else {  
 napi\_reject\_deferred(env, callbackData->deferred, result);  
 }

- napi\_delete\_async\_work(env, callbackData->asyncWork);  
delete callbackData;
- 
- 模块初始化以及ArkTS侧调用接口。  
  
// 模块初始化  
static napi\_value Init(napi\_env env, napi\_value exports){  
 napi\_property\_descriptor desc[] = {  
 { "asyncWork", nullptr, AsyncWork, nullptr, nullptr, napi\_default, nullptr }  
 };  
 napi\_define\_properties(env, exports, sizeof(desc) / sizeof(desc[0]), desc);  
 return exports;  
}  
  
// 接口对应的.d.ts描述  
export const asyncWork(data: number): Promise<number>;  
  
// ArkTS侧调用接口  
nativeModule.asyncWork(1024).then((result) =>  
{  
 hilog.info(0x0000, 'XXX', 'result is %{public}d', result);  
});



# 使用callback方式示例

# 实例代码

- 使用napi\_create\_async\_work创建异步任务，并使用napi\_queue\_async\_work将异步任务加入队列，等待执行。

```
• struct CallbackData {
• napi_async_work asyncWork = nullptr;
• napi_ref callbackRef = nullptr;
• double args[2] = {0};
• double result = 0;
• };
•
• napi_value AsyncWork(napi_env env, napi_callback_info info)
• {
• size_t argc = 3;
• napi_value args[3];
• napi_get_cb_info(env, info, &argc, args, nullptr, nullptr);
• auto asyncContext = new CallbackData();
• // 将接收到的参数保存到callbackData
• napi_get_value_double(env, args[0], &asyncContext->args[0]);
• napi_get_value_double(env, args[1], &asyncContext->args[1]);
• // 将传入的callback转换为napi_ref延长其生命周期，防止被GC掉
• napi_create_reference(env, args[2], 1, &asyncContext->callbackRef);
• napi_value resourceName = nullptr;
• napi_create_string_utf8(env, "asyncWorkCallback", NAPI_AUTO_LENGTH, &resourceName);
• // 创建异步任务
• napi_create_async_work(env, nullptr, resourceName, ExecuteCB, CompleteCB,
• asyncContext, &asyncContext->asyncWork);
• // 将异步任务加入队列
• napi_queue_async_work(env, asyncContext->asyncWork);
• return nullptr;
• }
```

- 定义异步任务的第一个回调函数，该函数在工作线程中执行，处理具体的业务逻辑。

```
• static void ExecuteCB(napi_env env, void *data)
• {
• CallbackData *callbackData = reinterpret_cast<CallbackData *>(data);
• callbackData->result = callbackData->args[0] + callbackData->args[1];
• }
```

- 定义异步任务的第二个回调函数，该函数在主线程执行，将结果传递给ArkTS侧。

```
• static void CompleteCB(napi_env env, napi_status status, void *data)
• {
• CallbackData *callbackData = reinterpret_cast<CallbackData *>(data);
• napi_value callbackArg[1] = {nullptr};
• napi_create_double(env, callbackData->result, &callbackArg[0]);
• napi_value callback = nullptr;
• napi_get_reference_value(env, callbackData->callbackRef, &callback);
• // 执行回调函数
• napi_value result;
• napi_value undefined;
• napi_get_undefined(env, &undefined);
• napi_call_function(env, undefined, callback, 1, callbackArg, &result);
• // 删除napi_ref对象以及异步任务
• napi_delete_reference(env, callbackData->callbackRef);
• napi_delete_async_work(env, callbackData->asyncWork);
• delete callbackData;
• }
```

- 模块初始化以及ArkTS侧调用接口。

```
• // 模块初始化
• static napi_value Init(napi_env env, napi_value exports)
• {
• napi_property_descriptor desc[] = {
• {"asyncWork", nullptr, AsyncWork, nullptr, nullptr, napi_default,
• nullptr}
• };
• napi_define_properties(env, exports, sizeof(desc) / sizeof(desc[0]), desc);
• return exports;
• }
```

```
• // 接口对应的.d.ts描述
• export const asyncWork(arg1: number, arg2: number,
• callback: (result: number) => void): void;
•
• // ArkTS侧调用接口
• let num1: number = 123;
• let num2: number = 456;
• nativeModule.asyncWork(num1, num2, (result) => {
• hilog.info(0x0000, 'XXX', 'result is %{public}d', result);
• });
```

The screenshot shows two code snippets side-by-side. On the left, in a file named index.ts, there is a synchronous call to demoTest.add(3, 2) and an asynchronous call to demoTest.addSync(10, 2). On the right, in a file named libentry.ts, there is an asynchronous call to demoTest.addAsyncByCallback(3, 5) and an asynchronous call to demoTest.addAsyncByPromise(3, 2). A watermark across the bottom right reads '鸿蒙学堂' and '非授权请勿转载'.

```
// 示例代码index.ts
import demoTest from 'libentry.so'

...
// 同步调用
let result = demoTest.add(3, 2)

// 同步调用, 返回callback
demoTest.addSync(10, 2, (result)=>{
 this.currentContent = `target value is
${result}`
});

...

// 异步调用, 返回callback
demoTest.addAsyncByCallback(3, 5,
async(data)=>{
 this.currentContent = 'callback, value:
'+data
})

// 异步调用, 返回Promise
let promiseValue =
demoTest.addAsyncByPromise(3, 2);
promiseValue.then((val:number) => {
 this.currentContent = `type:
+typeof(promiseValue)+',
value: '+val
`})
```

同步调用

异步调用



# 同步vs异步

# Native侧调TS回调

- 设置模块注册信息

ArkTS侧import native模块时，会加载其对应的so。加载so时，首先会调用napi\_module\_register方法，将模块注册到系统中，并调用模块初始化函数。napi\_module有两个关键属性：一个是.nm\_register\_func，定义模块初始化函数；另一个是.nm\_modname，定义模块的名称，也就是ArkTS侧引入的so库的名称，模块系统会根据此名称来区分不同的so。

```
• // entry/src/main/cpp/hello.cpp
•
• // 准备模块加载相关信息，将上述Init函数与本模块名等信息记录下来。
• static napi_module demoModule = {
• .nm_version = 1,
• .nm_flags = 0,
• .nm_filename = nullptr,
• .nm_register_func = Init,
• .nm_modname = "entry",
• .nm_priv = nullptr,
• .reserved = {0},
• };
•
• // 加载so时，该函数会自动被调用，将上述demoModule模块注册到系统中。
• extern "C" __attribute__((constructor)) void
RegisterDemoModule() {
 napi_module_register(&demoModule);
}
```

- 模块初始化

实现ArkTS接口与C++接口的绑定和映射。

```
• // entry/src/main/cpp/hello.cpp
• EXTERN_C_START
• // 模块初始化
• static napi_value Init(napi_env env, napi_value exports) {
• // ArkTS接口与C++接口的绑定和映射
• napi_property_descriptor desc[] = {
• {"callNative", nullptr, CallNative, nullptr,
• nullptr, nullptr, napi_default, nullptr},
• {"nativeCallArkTS", nullptr, NativeCallArkTS,
• nullptr, nullptr, napi_default, nullptr},
• };
• // 在exports对象上挂载callNative/NativeCallArkTS两个
• // Native方法
• napi_define_properties(env, exports, sizeof(desc) /
• sizeof(desc[0]), desc);
• return exports;
}
```

- }
- EXTERN\_C\_END

- // 模块基本信息
- static napi\_module demoModule = {
- .nm\_version = 1,
- .nm\_flags = 0,
- .nm\_filename = nullptr,
- .nm\_register\_func = Init,
- .nm\_modname = "entry",
- .nm\_priv = nullptr,
- .reserved = {0},

- 在index.d.ts文件中，提供JS侧的接口方法。

```
• // entry/src/main/cpp/types/libentry/index.d.ts
• export const callNative: (a: number, b: number) => number;
• export const nativeCallArkTS: (cb: (a: number) => number)
=> number;
```

- 在oh-package.json文件中将index.d.ts与cpp文件关联起来。

```
• {
• "name": "libentry.so",
• "types": "./index.d.ts",
• "version": "",
• "description": "Please describe the basic information."
• }
```

- 在CMakeLists.txt文件中配置CMake打包参数。

```
• # entry/src/main/cpp/CMakeLists.txt
• cmake_minimum_required(VERSION 3.4.1)
• project(MyApplication2)
•
• set(NATIVE_RENDER_ROOT_PATH ${CMAKE_CURRENT_SOURCE_DIR})
•
• include_directories(${NATIVE_RENDER_ROOT_PATH}
• ${NATIVE_RENDER_ROOT_PATH}/include)
•
• # 添加名为entry的库
• add_library(entry SHARED hello.cpp)
• # 构建此可执行文件需要链接的库
• target_link_libraries(entry PUBLIC libace_napi.z.so)
```

- 实现Native侧的CallNative以及NativeCallArkTS接口。具体代码如下：

```
• // entry/src/main/cpp/hello.cpp
• static napi_value CallNative(napi_env env,
napi_callback_info info)
{
 size_t argc = 2;
 // 声明参数数组
 napi_value args[2] = {nullptr};
 // 获取传入的参数并依次放入参数数组中
 napi_get_cb_info(env, info, &argc, args, nullptr,
nullptr);
 // 依次获取参数
 double value0;
 napi_get_value_double(env, args[0], &value0);
 double value1;
 napi_get_value_double(env, args[1], &value1);
 // 返回两数相加的结果
 napi_value sum;
 napi_create_double(env, value0 + value1, &sum);
 return sum;
}
static napi_value NativeCallArkTS(napi_env env,
napi_callback_info info)
{
 size_t argc = 1;
 // 声明参数数组
 napi_value args[1] = {nullptr};
 // 获取传入的参数并依次放入参数数组中
 napi_get_cb_info(env, info, &argc, args, nullptr,
nullptr);
 // 创建一个int，作为ArkTS的入参
 napi_value argv = nullptr;
 napi_create_int32(env, 2, &argv);
 // 调用传入的callback，并将其结果返回
 napi_value result = nullptr;
 napi_call_function(env, nullptr, args[0], 1, &argv,
&result);
 return result;
}
```

# Native侧调TS回调

ArkTS侧通过import引入Native侧包含处理逻辑的so来使用C/C++的方法。

```
• // entry/src/main/ets/pages/Index.ets
• // 通过import的方式，引入Native能力。
• import nativeModule from 'libentry.so'
•
• @Entry
• @Component
• struct Index {
• @State message: string = 'Test Node-API callNative result: ';
• @State message2: string = 'Test Node-API nativeCallArkTS result: ';
• build() {
• Row() {
• Column() {
• // 第一个按钮，调用add方法，对应到Native侧的CallNative方法，进行两数相加。
• Text(this.message)
• .fontSize(50)
• .fontWeight(FontWeight.Bold)
• .onClick(() => {
• this.message += nativeModule.callNative(2, 3);
• })
• // 第二个按钮，调用nativeCallArkTS方法，对应到Native的NativeCallArkTS，在Native调用ArkTS function。
• Text(this.message2)
• .fontSize(50)
• .fontWeight(FontWeight.Bold)
• .onClick(() => {
• this.message2 += nativeModule.nativeCallArkTS((a: number)=> {
• return a * 2;
• });
• })
• }
• }
• }
• .width('100%')
• }
• .height('100%')
}
```

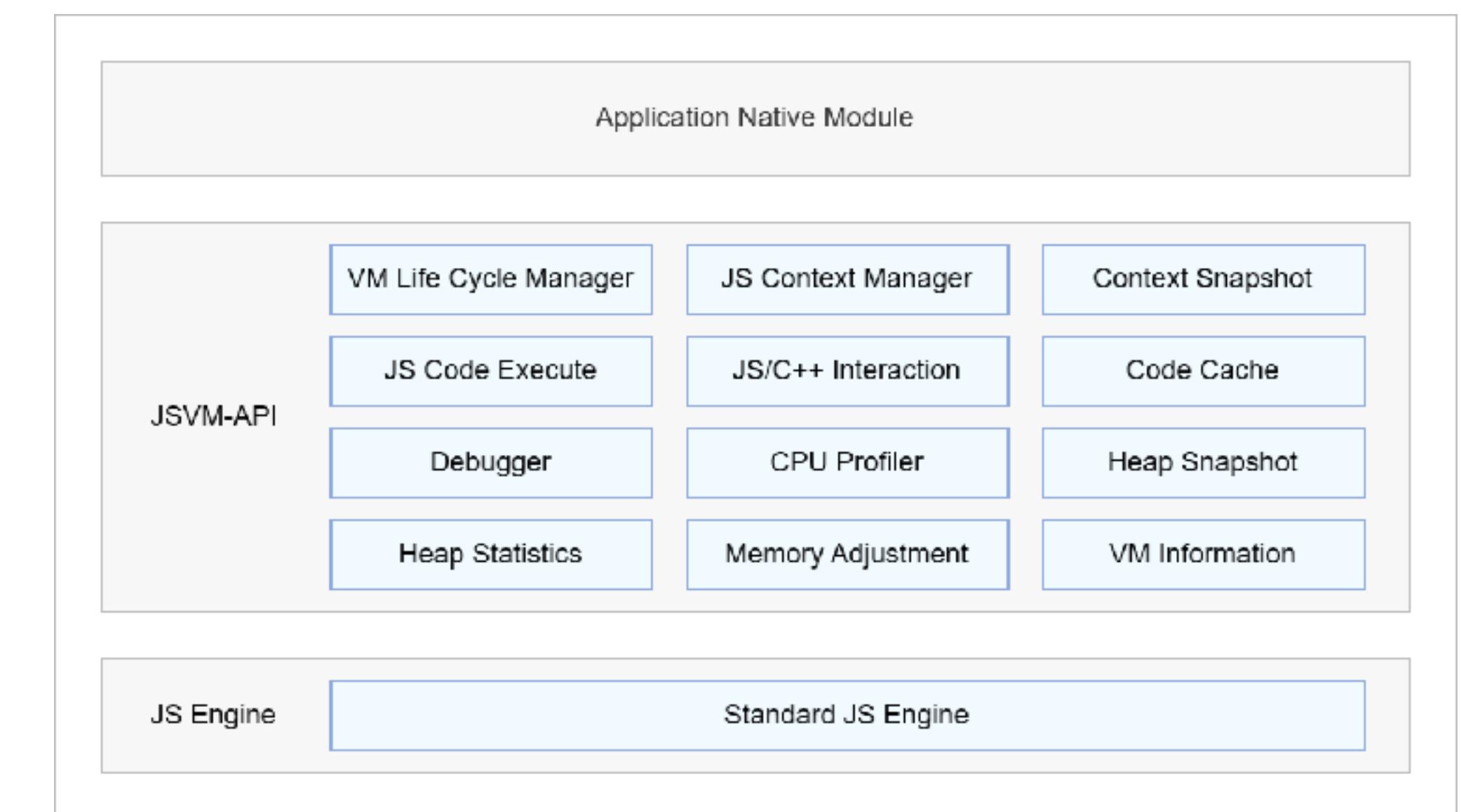
# JSVM-API简介

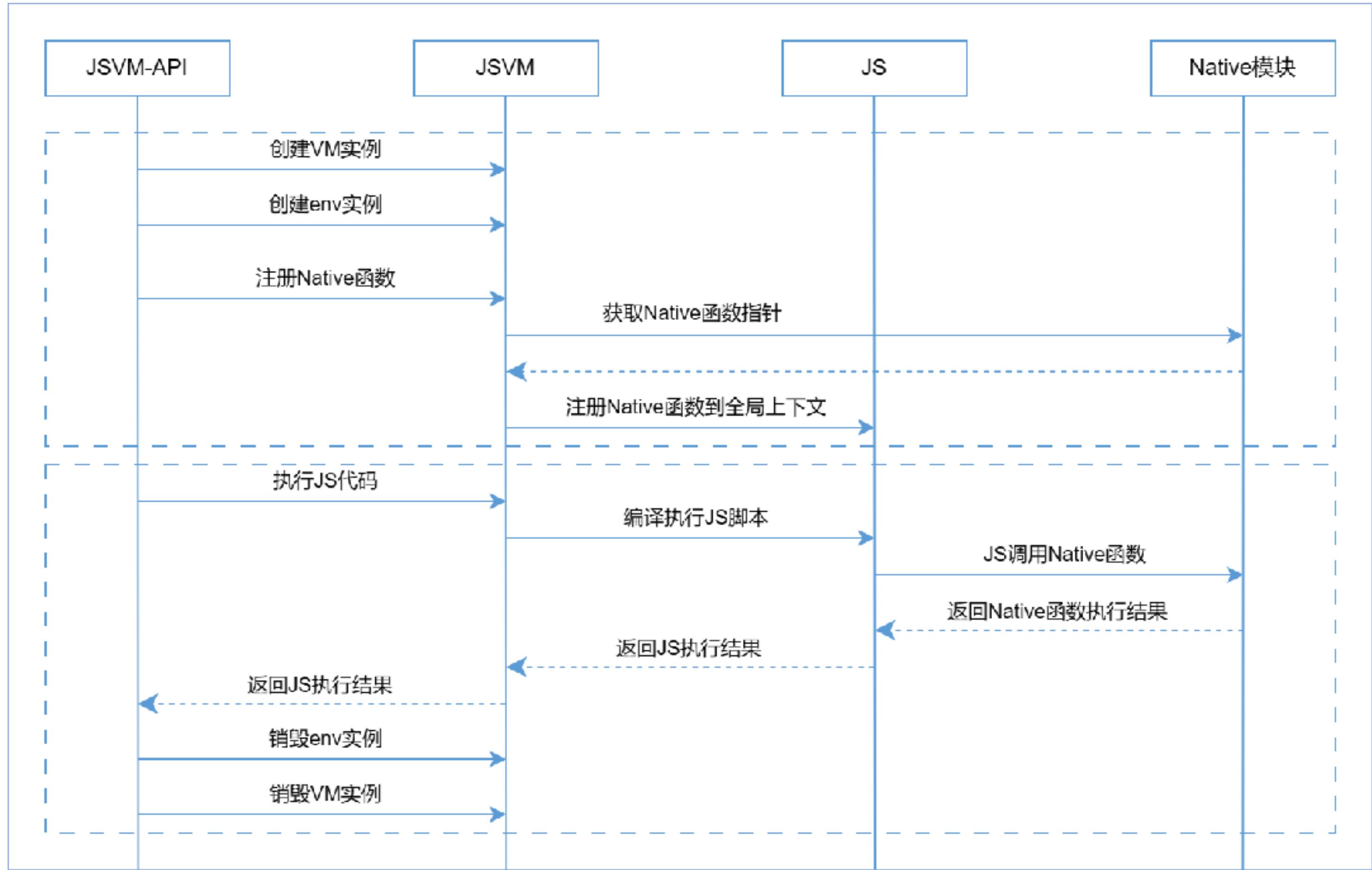
HarmonyOS JSVM-API是基于标准JS引擎提供的一套稳定的API，为开发者提供了较为完整的JS引擎能力，包括创建和销毁引擎，执行JS代码，JS/C++交互等关键能力。

HarmonyOS JSVM-API是C语言接口，遵循C99标准。

通过JSVM-API，开发者可以在应用运行期间直接执行一段动态加载的JS代码。也可以选择将一些对性能、底层系统调用有较高要求的核心功能用C/C++实现并将C++方法注册到JS侧，在JS代码中直接调用，提高应用的执行效率。

- Native Module：开发者使用JSVM-API开发的模块，用于在Native侧使用。
- VM Life Cycle Manager：管理JSVM\_Vm的生命周期。
- JS Context Manager：管理JSVM\_Env的生命周期。
- Context Snapshot：上下文快照，可用以缩短JS Context的创建时间。
- JS Code Execute：执行JS代码。
- JS/C++ Interaction：连接JS层与C++层，用于支撑JS与C++之间的交互。
- Code Cache：编译后的JS代码的缓存，能提升JS代码执行的启动速度。
- Debugger：调试器，用于调试JS代码。
- CPU Profiler：该工具能记录JS代码执行所用的时间，使用此工具能帮助开发者分析JS代码的性能瓶颈，为代码优化提供数据支撑。
- Heap Snapshot：JS堆内存分析/调优工具，可以进行内存优化和发现内存泄漏问题。
- Heap Statistics：JS堆统计信息，包括内存大小及上下文数量。
- Memory Adjustment：调整外部内存大小、虚拟机内存压力，以加快触发GC。
- VM Information：JSVM\_Vm的信息。
- Standard JS Engine：标准JS引擎。





# 交互流程

JSVM-API和Native模块之间的交互流程，主要分为以下两步：

- **初始化阶段：**在Native模块上初始化JSVM和JS上下文，并完成Native函数的注册。Native方法将会被挂载到JS执行环境的全局上下文即GlobalThis。
- **调用阶段：**当JS侧调用通过JSVM-API注册到JS全局上下文的方法时，JS引擎会找到并调用对应的C/C++方法。

# 实例代码

## Native侧方法的实现

- 设置模块注册信息  
具体见[设置模块注册信息](#)
- 模块初始化  
实现ArkTS接口与C++接口的绑定和映射。

```
• // entry/src/main/cpp/hello.cpp
• EXTERN_C_START
• // 模块初始化
• static napi_value Init(napi_env env, napi_value exports)
• {
• // ArkTS接口与C++接口的绑定和映射
• napi_property_descriptor desc[] = {
• {"runTest", nullptr, RunTest, nullptr, nullptr,
• nullptr, napi_default, nullptr},
• };
• // 在exports对象上挂载RunJsvm的Native方法
• napi_define_properties(env, exports, sizeof(desc) /
• sizeof(desc[0]), desc);
• return exports;
• }
• EXTERN_C_END
```

- 在index.d.ts文件中，提供JS侧的接口方法。

```
• // entry/src/main/cpp/types/libentry/index.d.ts
• export const runTest: () => void;
```

- 在oh-package.json文件中将index.d.ts与cpp文件关联起来。

```
• {
• "name": "libentry.so",
• "types": "./index.d.ts",
• "version": "",
• "description": "Please describe the basic information."
• }
```

- 在CMakeLists.txt文件中配置CMake打包参数。

```
• # entry/src/main/cpp/CMakeLists.txt
• cmake_minimum_required(VERSION 3.4.1)
• project(JSVDemo)
```

- set(NATIVE\_RENDER\_ROOT\_PATH \${CMAKE\_CURRENT\_SOURCE\_DIR})
• # 日志打印配置
• add\_definitions( "-DLOG\_DOMAIN=0xd0d0" )
• add\_definitions( "-DLOG\_TAG=\"testTag\"" )
• include\_directories(\${NATIVE\_RENDER\_ROOT\_PATH}
• \${NATIVE\_RENDER\_ROOT\_PATH}/include)

• # 添加名为entry的库
• add\_library(entry SHARED hello.cpp)
• # 构建此可执行文件需要链接的库
• target\_link\_libraries(entry PUBLIC libace\_napi.z.so
• libjsvm.so libhilog\_ndk.z.so)

- 实现Native侧的runTest接口。具体代码如下：

```
• #include "napi/native_api.h"
• #include "hilog/log.h"
• #include "ark_runtime/jsvm.h"

• #define LOG_DOMAIN 0x3200
• #define LOG_TAG "APP"

• static int g_aa = 0;

• #define CHECK_RET(theCall)
• do {
• JSVM_Status cond = theCall;
• if ((cond) != JSVM_OK) {
• const JSVM_ExtendedErrorInfo *info;
• OH_JSVMSGetLastErrorInfo(env, &info);
• OH_LOG_ERROR(LOG_APP, "jsvm fail file: %
• {public}s line: %{public}d ret = %{public}d message = %
• {public}s", \
• __FILE__, __LINE__, cond, info !=
• nullptr ? info->errorMessage : "");
• return -1;
• }
• } while (0)
```

```
• #define CHECK(theCall)
• do {
• JSVM_Status cond = theCall;
• if ((cond) != JSVM_OK) {
• OH_LOG_ERROR(LOG_APP, "jsvm fail file: %
• {public}s line: %{public}d ret = %{public}d", __FILE__,
• __LINE__, cond);
• return -1;
• }
• } while (0)

• // 用于调用theCall并检查其返回值是否为JSVM_OK。
• // 如果不是，则调用OH_JSVMSGetLastErrorInfo处理错误并返回retVal。
• #define JSVM_CALL_BASE(env, theCall, retVal)
• do {
• JSVM_Status cond = theCall;
• if (cond != JSVM_OK) {
• const JSVM_ExtendedErrorInfo *info;
• OH_JSVMSGetLastErrorInfo(env, &info);
• OH_LOG_ERROR(LOG_APP, "jsvm fail file: %
• {public}s line: %{public}d ret = %{public}d message = %
• {public}s", \
• __FILE__, __LINE__, cond, info !=
• nullptr ? info->errorMessage : "");
• return retVal;
• }
• } while (0)

• // JSVM_CALL_BASE的简化版本，返回nullptr
• #define JSVM_CALL(theCall) JSVM_CALL_BASE(env, theCall,
• nullptr)
```

# 实例代码

```
• #define JSVM_CALL(theCall) JSVM_CALL_BASE(env, theCall, nullptr)
•
• // OH_JSVM_StrictEquals的样例方法
• static JSVM_Value IsStrictEquals(JSVM_Env env,
JSVM_CallbackInfo info) {
• // 接受两个入参
• size_t argc = 2;
• JSVM_Value args[2] = {nullptr};
• JSVM_CALL(OH_JSVM_GetCbInfo(env, info, &argc, args,
nullptr, nullptr));
• // 调用OH_JSVM_StrictEquals接口判断给定的两个JavaScript
value是否严格相等
• bool result = false;
• JSVM_Status status = OH_JSVM_StrictEquals(env,
args[0], args[1], &result);
• if (status != JSVM_OK) {
• OH_LOG_ERROR(LOG_APP, "JSVM OH_JSVM_StrictEquals:
failed");
• } else {
• OH_LOG_INFO(LOG_APP, "JSVM OH_JSVM_StrictEquals:
success: %{public}d", result);
• }
• JSVM_Value isStrictEqual;
• JSVM_CALL(OH_JSVM_GetBoolean(env, result,
&isStrictEqual));
• return isStrictEqual;
• }
• // IsStrictEquals注册回调
• static JSVM_CallbackStruct param[] = {
• {.data = nullptr, .callback = IsStrictEquals},
• };
• static JSVM_CallbackStruct *method = param;
• // IsStrictEquals方法别名, 供JS调用
• static JSVM_PropertyDescriptor descriptor[] = {
• {"isStrictEquals", nullptr, method++, nullptr,
nullptr, nullptr, JSVM_DEFAULT},
• };
• // 样例测试js
• const char *srcCallNative = R"JS(let data = '123';
let value = 123;
isStrictEquals(data,value);)JS";
•
• static int32_t TestJSVM() {
• JSVM_InitOptions initOptions = {0};
• JSVM_VM vm;
• JSVM_Env env = nullptr;
• JSVM_VMScope vmScope;
• JSVM_EnvScope envScope;
```

```
• JSVM_HandleScope handleScope;
• JSVM_Value result;
• // 初始化JavaScript引擎实例
• if (g_aa == 0) {
• g_aa++;
• CHECK(OH_JSVM_Init(&initOptions));
• }
• // 创建JSVM环境
• CHECK(OH_JSVM_CreateVM(nullptr, &vm));
• CHECK(OH_JSVM_CreateEnv(vm, sizeof(descriptor) /
sizeof(descriptor[0]), descriptor, &env));
• CHECK(OH_JSVM_OpenVMScope(vm, &vmScope));
• CHECK_RET(OH_JSVM_OpenEnvScope(env, &envScope));
• CHECK_RET(OH_JSVM_OpenHandleScope(env, &handleScope));
•
• // 通过script调用测试函数
• JSVM_Script script;
• JSVM_Value jsSrc;
• CHECK_RET(OH_JSVM_CreateStringUtf8(env, srcCallNative,
JSVM_AUTO_LENGTH, &jsSrc));
• CHECK_RET(OH_JSVM_CompileScript(env, jsSrc, nullptr,
0, true, nullptr, &script));
• CHECK_RET(OH_JSVM_RunScript(env, script, &result));
•
• // 销毁JSVM环境
• CHECK_RET(OH_JSVM_CloseHandleScope(env, handleScope));
• CHECK_RET(OH_JSVM_CloseEnvScope(env, envScope));
• CHECK(OH_JSVM_CloseVMScope(vm, vmScope));
• CHECK(OH_JSVM_DestroyEnv(env));
• CHECK(OH_JSVM_DestroyVM(vm));
• return 0;
• }
•
• static napi_value RunTest(napi_env env, napi_callback_info
info) {
• TestJSVM();
• return nullptr;
• }
•
• // 模块注册信息, 供arkts侧调用
• EXTERN_C_START
• static napi_value Init(napi_env env, napi_value exports) {
• napi_property_descriptor desc[] = {"runTest", nullptr,
RunTest, nullptr, nullptr, nullptr, napi_default,
nullptr};
• napi_define_properties(env, exports, sizeof(desc) /
sizeof(desc[0]), desc);
• return exports;
• }
•
• EXTERN_C_END
• static napi_module demoModule = {
• .nm_version = 1,
• .nm_flags = 0,
• .nm_filename = nullptr,
• .nm_register_func = Init,
• .nm_modname = "entry",
• .nm_priv = ((void *)0),
• .reserved = {0},
• };
•
• extern "C" __attribute__((constructor)) void
RegisterEntryModule(void)
{ napi_module_register(&demoModule); }
```

## ArkTS侧调用C/C++方法实现

```
import hilog from '@ohos.hilog'
// 通过import的方式, 引入Native能力。
import napitest from 'libentry.so'

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

 build() {
 Row() {
 Column() {
 Text(this.message)
 .fontSize(50)
 .fontWeight(FontWeight.Bold)
 .onClick(() => {
 // runtest
 napitest.runTest();
 })
 .width('100%')
 }
 .height('100%')
 }
 }
}
```

预期输出结果

• JSVM OH\_JSVM\_StrictEquals: success: 1