# 鸿蒙编程语言白皮书

文档版本　　V1.0

发布日期　　2025-06-20

**注意**

华为会不定期对本文档的内容进行更新。

本文档仅作为使用指导，文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

## 华为终端有限公司

地址: 广东省东莞市松山湖园区新城路 2 号

网址: https://consumer.huawei.com

# CONTENT

# 03 鸿蒙编程语言演进策略

# 1 第一章：鸿蒙编程语言整体框架

鸿蒙是多语言生态，ArkTS、仓颉和 C/C++充分互补。ArkTS 是动态类型编程语言，主打易学易用、生态丰富、极简开发、持续创新四大特征；仓颉是静态类型编程语言，主打高性能、强安全、跨平台、智能化等特性。为满足不同业务场景诉求及不同开发者编程习惯，两者长期协同发展和长期演进，并保持生态兼容。ArkTS 和仓颉均通过垃圾回收机制自动管理内存，C/C++支持 Native 开发，需开发者手动管理内存，三种语言相互配合，共同支撑鸿蒙应用生态构建。


图 1-1：鸿蒙 APP 架构示意图

## 1.1 ArkTS 概述

ArkTS 是鸿蒙应用开发高级语言。

ArkTS 基于 TypeScript（简称 TS），保持了 TS 的基本语法和风格，同时通过引入静态类型校验模式和类型推断增强规则，强化开发期静态检查和分析能力，提升代码健壮性，并实现更好的程序执行稳定性和性能。ArkTS 同时也支持与 TS/JavaScript（简称 JS）高效互操作，可以完全复用 TS/JS 生态，已广泛应用于鸿蒙应用生态。

在标准 TS 的基础上，ArkTS 结合鸿蒙应用开发的诉求进行了创新和能力扩展，主要新增四大特性如下：

- **并发编程模型**：ArkTS 新增提供 TaskPool 和 Worker 两种并发编程 API 供开发者使用。同时，ArkTS 进一步提出了 Sendable 对象模型的机制来支持对象在并发任务间的引用传递，极大提升 ArkTS 对象在并发实例间的通信性能。
- **声明式语法**：ArkTS 结合 ArkUI 提供声明式 UI 描述、状态管理、渲染控制等强大的 UI 开发能力，拥有简洁且富有表达力的语法，通过简洁的语法和实时预览功能，大大提高了 UI 开发的效率，使得代码更易于编写和阅读。
- **强大的标准库**：ArkTS 拥有一个功能丰富的标准库，涵盖了从数据结构、算法到输入输出等方方面面，例如：高精度浮点运算、二进制 Buffer、XML 生成解析转换和多种容器库等丰富的操作方法，帮助开发者简化开发工作，提升开发效率。
- **模块化管理**：ArkTS 支持应用模块化开发、编译、打包和运行，例如：应用模块化按需加载能力，方便大型复杂应用的多模块业务场景，高性能启动运行，提高了代码的模块化管理和重用性。

方舟编译运行时（ArkCompiler）支持 ArkTS、TS、JS 的编译运行，目前它主要分为 ArkTS 编译工具链和 ArkTS 运行时两部分。其中 ArkTS 编译工具链负责在开发侧将高级语言编译为方舟字节码文件（*.abc），而 ArkTS 运行时则负责在设备侧运行字节码文件执行程序逻辑。

ArkTS 会结合鸿蒙应用开发的需求持续创新，平滑演进。进一步丰富并发编程、完善类型系统、现代化语法等显著改进和新特性，使开发者能够更快速地构建稳定且性能优越的应用。

## 1.2 仓颉概述

仓颉是鸿蒙应用开发高级语言。

仓颉作为一款面向鸿蒙应用开发的现代编程语言，是一款静态类型、静态编译的编程语言，通过现代语言特性的集成、全方位的编译优化和运行时实现、以及开箱即用的 DevEco Studio 工具链支持，为鸿蒙应用开发者打造友好开发体验和卓越程序性能。其具体特性表现为：

- **高性能**：仓颉基于静态类型和静态编译优化技术，具有“编译前端+编译后端+运行时”的全栈垂直优化能力，为鸿蒙应用提供卓越的性能支持。仓颉采用内存共享的并发模型，提供轻量用户态线程和易用的无锁并发数据结构让并发编程变得轻松高效；提供了低时延高效率的自动内存管理，支持鸿蒙应用以更高帧率、更少内存流畅运行，从而降低设备功耗，延长续航。仓颉运行时采用轻量化设计，使仓颉应用具有较低的基础开销。通过仓颉包按需动态加载技术，仓颉应用启动/运行占用资源更少。

- **强安全**：仓颉通过静态类型系统和自动内存管理，确保程序内存安全；同时，仓颉提供多种编译时和运行时检查，包括数组下标越界检查、类型转换检查、数值计算溢出检查、以及字符串编码合法性检查等，能够及时发现程序运行中的错误。
- **跨平台**：仓颉支持基于静态编译至机器码的跨 OS 平台执行能力，允许开发者实现“同构开发、异构运行”的跨 OS 平台代码共享，支持 OpenHarmony、Android、iOS、Windows、Linux、MacOS 等 OS 平台。
- **智能化**：仓颉通过元编程扩展出面向 LLM 智能体编程的 Agent DSL，该语言提供多种特性有效简化 Agent 编写复杂度，包括 Agent 声明式配置、提示词模式、多 Agent 协同、多轮对话等；同时，该语言兼容 MCP 生态，能够让开发者快速开发智能体应用。

## 1.3 C/C++概述

鸿蒙应用开发全面支持 C/C++语言开发能力，为开发者提供开发套件（NDK）及配套工具链。基于 C/C++实现的功能模块，可通过跨语言互操作封装为 ArkTS 和仓颉扩展模块，提供给 ArkTS 和仓颉高效使用。

### 1.3.1 C/C++在应用开发中的适用场景

- **高性能计算场景：**游戏引擎、物理仿真等计算密集型任务。
- **硬件加速场景：**需要深度优化 CPU 指令集的专用算法库。
- **生态复用场景：**复用存量代码。

### 1.3.2 C/C++支持组件

为构建符合鸿蒙技术标准的应用开发能力，提供了一套完整的 C/C++开发组件体系，具体包括：

- **NDK（Native Development Kit）：**作为鸿蒙应用开发的核心工具套件，NDK 对外发布提供完整的编译工具链、系统 C API 接口、C++运行时环境支持。
- **libc 标准运行时库：**基于开源 MUSL 库作优化，鸿蒙系统实现了标准 C 语言运行时支持、功能扩展、系统级性能的提升。
- **标准 C++运行时库：**采用 LLVM 项目的 libc++作为基础，提供完整的 C++标准库实现，支持 C++11、C++14、C++17 和 C++20 标准。
- **编译工具链：**选用毕昇编译器作为官方指定的 C/C++编译器，确保符合最新语言标准规范、支持特有的编译优化、提供稳定的工具链支持。这套组件体系共同构成了鸿蒙系统对开发能力的完整支持，为开发者提供了高效、稳定的 C/C++开发环境。

## 1.4 语言互操作介绍

### 1.4.1 ArkTS 与 C/C++互操作

为了更好复用 TS/JS 的生态，ArkTS 语言提供 Node-API 的兼容实现，提供 ArkTS 与 C/C++的跨语言操作；Node-API 是基于 Node.js 12.x 的 Node-API 规范扩展开发的机制，为开发者提供了 ArkTS 与 C/C++模块之间的交互能力。它提供了一组稳定的、跨平台的 API，可以在不同的操作系统上使用。

涉及到互操作的主要场景如下：

- 系统可以将框架层丰富的模块功能通过 ArkTS 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 接口使用，提高应用本身的执行效率。

#### 1.4.1.1 Node-API 的组成架构

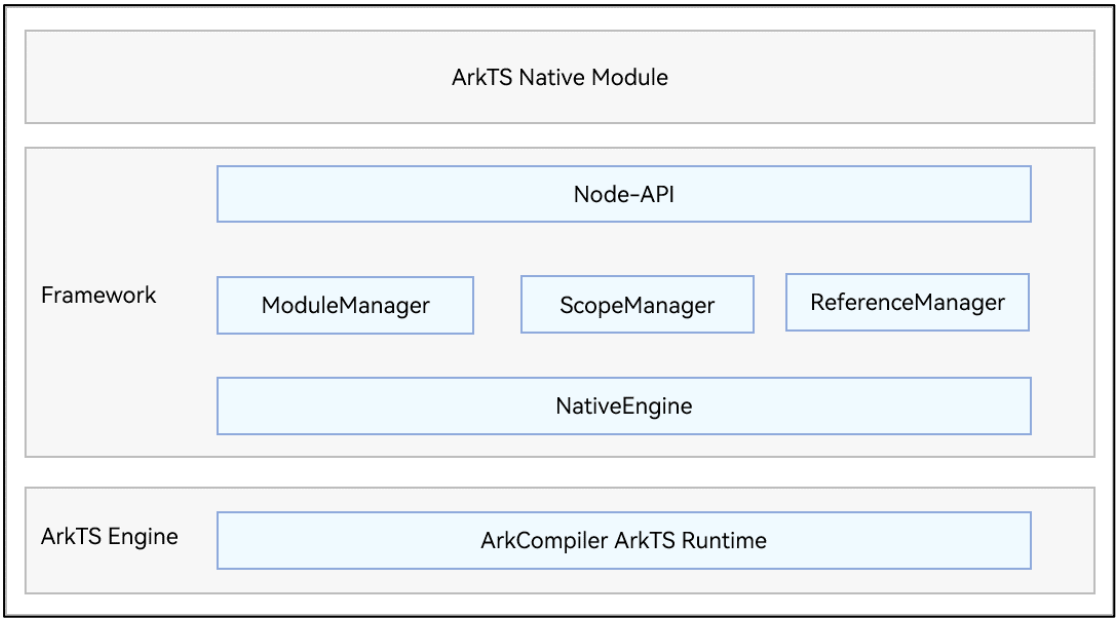


图 1-2：Node-API 的组成架构

- 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。

### 1.4.1.2 Node-API 的关键交互流程

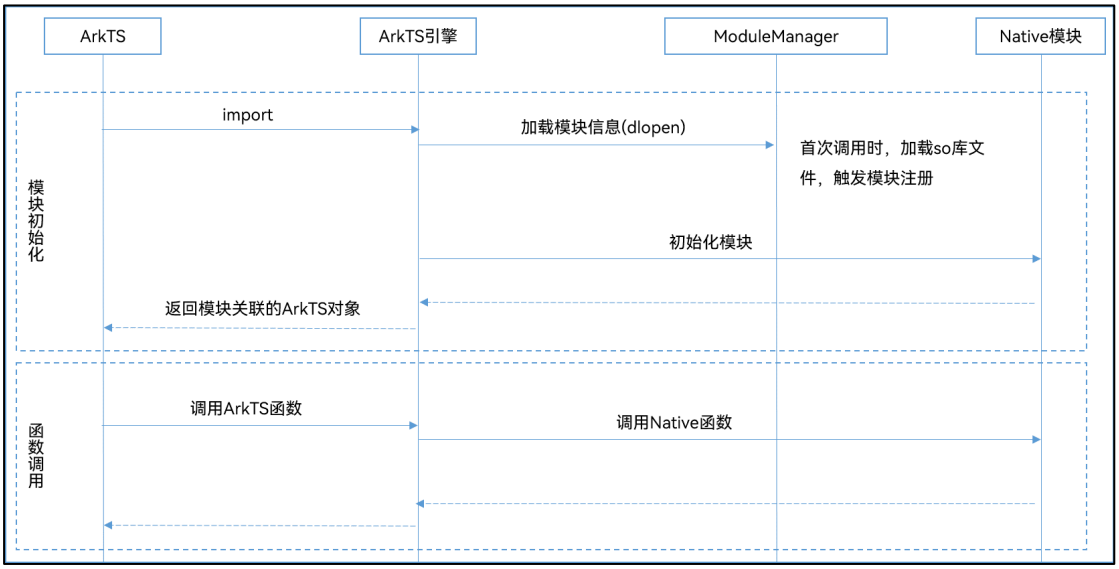


图 1-3：Node-API 的关键交互流程

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。

## 1.4.2 仓颉与 C/C++互操作

仓颉支持与 C 互操作，即支持仓颉函数与 C 语言函数的互相调用以及跨语言数据转换。对于 C++ 代码，开发者首先需要把 C++ 接口封装成 C 接口，再由仓颉通过互操作调用。

仓颉与 C 互操作机制考虑兼顾互操作语法的简洁性，以及低互操作开销，主要提供以下能力：

- 声明式接口描述：支持 @C 注解和 foreign 关键字，允许在仓颉代码中声明被调用的 C 函数接口；
- 低互操作开销：在仓颉侧使用 @C 和 foreign 标注的结构体和函数，其实现与 C 在二进制层面保持兼容；提供 inout 关键字，可将仓颉栈上的变量引用传递到 C 侧，减少跨语言拷贝；提供 @FastNative 注解标注 foreign 函数，减少调用被标注函数的运行时开销；支持将仓颉 Array<T> 中数据的原始指针实例传递到 C 侧访问，避免大块内存拷贝；以上特性合理使用，可帮助开发者降低跨语言开销。

以下是一个示例，假设 C 侧有如下函数，会被仓颉代码调用：

```
#include <math.h>

typedef struct {
    double x;
    double y;
} Point;

Point rotate(Point input, double t) {
    Point output;
    output.x = input.x * cos(t) - input.y * sin(t);
    output.y = input.x * sin(t) + input.y * cos(t);
    return output;
}
```

对应的，仓颉侧可以通过编写如下代码来调用该 C 函数：

```
@C
struct Point { // 用仓颉结构体语法声明 C 结构体
    Point(var x: Float64, var y: Float64) {}
}

```

```
// 用仓颉函数语法声明 C 函数
@FastNative
foreign func rotate(point: Point, alpha: Float64): Point

func callInterop(x: Float64, y: Float64, alpha: Float64): Unit {
    // 引用 C 侧类型和函数时，就像在引用仓颉程序中的元素
    unsafe {
        let input = Point(x, y)
        let output = rotate(input, alpha)
        println("${output.x}, ${output.y}")
    }
}


Point rotate(Point input, double t) {
    Point output;
    output.x = input.x * cos(t) - input.y * sin(t);
    output.y = input.x * sin(t) + input.y * cos(t);
    return output;
}
```

其中：

- 通过 @C 注解和 foreign 关键字声明对应的 C 结构体和函数，然后在仓颉代码中调用相关函数；
- 由于与 C 语言的互操作过程中，会引入了 C 的许多不安全因素，因此在仓颉中使用 unsafe 关键字，用于对跨 C 调用的不安全行为进行标识；
- 同时例子中使用 @FastNative 注解来标注 rotate 函数，实际调用中运行时会基于这个标注优化互操作开销。

更详细描述请参考《仓颉编程语言开发指南》中 仓颉-C 互操作 [1] 章节。

### 1.4.3 仓颉与 ArkTS 互操作

在鸿蒙应用开发中，仓颉和 ArkTS 各具特点和优势，存在使用仓颉与 ArkTS 混合开发的诉求，例如以下场景：

- 场景一：在使用 ArkTS 开发时，通过跨语言互操作调用仓颉开发的代码模块，发挥仓颉高性能高并发优势，提升应用性能体验。
- 场景二：在使用仓颉开发时，通过跨语言互操作调用 ArkTS 库，复用 ArkTS 丰富的库生态。

针对互操作场景诉求，仓颉提供 ohos.ark_interop 互操作库来实现与 ArkTS 的互操作。该库主要提供以下关键数据结构：

- JSValue: 用于表示来自 ArkTS 中的对象（如数字、字符串、对象、函数），是仓颉与 ArkTS 数据转换的桥梁。
- JSContext: 用于表示与 ArkTS 互操作的上下文，提供模块加载、JSValue 创建等能力。
- JSCallInfo: 用于表示当发生来自于 ArkTS 互操作调用时，调用的参数集合。

[1] 仓颉-C 互操作参见 https://developer.huawei.com/consumer/cn/doc/cangjie-guides-V5/cangjie-c-V5

### 1.4.3.1 ArkTS 调用仓颉

针对场景一的诉求，可使用互操作库，在仓颉侧实现可被互操作调用的接口，示例如下：

```
1.  package ohos_app_cangjie_entry
2.
3.  // 导入互操作库
4.  import ohos.ark_interop.*
5.
6.  func addNumber(context: JSContext, callInfo: JSCallInfo): JSValue {
7.      // 从 JSCallInfo 获取参数列表
8.      let arg0: JSValue = callInfo[0]
9.      let arg1: JSValue = callInfo[1]
10.     // 把 JSValue 转换为仓颉类型
11.     let a: Float64 = arg0.toNumber()
12.     let b: Float64 = arg1.toNumber()
13.     // 实际仓颉函数行为
14.     let value = a + b
15.     // 把结果转换为 JSValue
16.     let result: JSValue = context.number(value).toJSValue()
17.     // 返回 JSValue
18.     return result
19. }
20.
21. // 必须注册该函数到 JSModule 中
22. let EXPORT_MODULE = JSModule.registerModule {
23.     runtime, exports =>
24.         exports["addNumber"] = runtime.function(addNumber).toJSValue()
25. }
26.
27.
28. Point rotate(Point input, double t) {
29.     Point output;
30.     output.x = input.x * cos(t) - input.y * sin(t);
31.     output.y = input.x * sin(t) + input.y * cos(t);
32.     return output;
```

```
33. }
```

然后在 ArkTS 侧就可以加载仓颉模块，并调用该接口

```
1.  // 导入仓颉动态库，该动态库名称为仓颉包名的名称，该名称需要和互操作接口所在的包名一致
2.  import { addNumber } from "libohos_app_cangjie_entry.so";
3.
4.  // 调用仓颉接口
5.  let result = addNumber(1, 2);
```

同时为了提升开发者易用性，仓颉提供声明式互操作宏机制，使开发者可以标注仓颉代码中需要被 ArkTS 跨语言使用的函数或类型，在编译阶段自动生成互操作“胶水层”代码及 ArkTS 接口声明，从而减少开发者手写互操作代码的复杂度。比如以上案例中仓颉侧代码可以简化成：

```
1.  import ohos.ark_interop.*
2.  import ohos.ark_interop_macro.*
3.
4.  @Interop[ArkTS]
5.  public func addNumber(a: Float64, b: Float64): Float64 {
6.      a + b
7.  }
```

### 1.4.3.2 仓颉调用 ArkTS

针对场景二的诉求，可使用互操作库，加载 ArkTS 模块，并调用接口，示例如下：

```
import ohos.ark_interop.*

func callInterop() {
    // 获取互操作上下文
    let context: JSContext = JSRuntime().mainContext
    // 加载目标 ArkTS 模块
    let module: JSObject = context.requireSystemNativeModule("usbManager").asObject(context)

    // 获取 ArkTS 模块导出函数
    let getDevices = module["getDevices"].asFunction(context)
    // 调用 ArkTS 函数
    let result = getDevices.call().asArray()
}
```

为了提升该场景开发者易用性，DevEco Studio 提供工具对 ArkTS 库进行互操作自动封装。当开发者在仓颉工程中导入 ArkTS 库时，该工具解析 ArkTS 库的接口声明文件（.d.ts 或者.d.ets 文件），生成仓颉调用 ArkTS 接口的互操作“胶水层”代码，开发者直接调用由工具生成的仓颉接口即可，从而避免开发者手写互操作代码的复杂度。

更详细描述请参考《仓颉编程语言开发指南》中 仓颉-ArkTS 互操作 [2] 章节。

[2] 仓颉-ArkTS 互操作更多参见 https://developer.huawei.com/consumer/cn/doc/cangjie-guides-V5/3_2_u4ed3_u9889-arkts-_u4e92_u64cd_u4f5c-V5

# 2 第二章：鸿蒙编程语言适用场景

ArkTS 主打易学易用、生态丰富、极简开发、持续创新四大特征；仓颉主打高性能、强安全、跨平台等特性；两者相辅相成，共同支撑开发者鸿蒙生态应用开发。

结合当前鸿蒙应用开发者遇到的主要问题及场景，我们锚定高效开发、高性能开发、安全、跨平台、技术积累可复用五大核心场景，以鸿蒙分布式操作系统的技术特性为基底，构建覆盖「开发-运行-安全-生态-沉淀」的全链路生产力闭环。这一选择既源于鸿蒙「万物互联」的战略定位，也直击传统开发模式中效率瓶颈、性能损耗、安全风险、多端割裂及历史资产投入浪费的核心痛点。

## 2.1 高效开发

### 2.1.1 ArkTS 高效开发能力概述

- **兼容 TS 高效语法**：极简语法，ArkTS 保留 TS 基本语法风格，兼容 TS 高效语法，显著提升代码的简洁性和健壮性，方便开发者根据场景灵活使用。
- **基于 TS 的高效语法进一步增强效率**

  ○ **扩展高频基础库**，ArkTS 基础类库提供了 XML 生成解析转换、二进制 Buffer、多种容器类库、URL 字符串解析和高精度浮点计算等能力，协助开发者简化开发工作，提升开发效率。

- **扩展并发能力**，针对 TS/JS 并发能力支持有限的问题，ArkTS 对并发编程 API 和能力进行了增强，提供了 TaskPool 和 Worker 两种并发 API 供开发者选择。另外，ArkTS 进一步提出了 Sendable 的概念来支持对象在并发实例间的引用传递，提升 ArkTS 对象在并发实例间的通信性能。

- **支持声明式 UI 开发**，ArkTS 提供了声明式 UI 范式、状态管理支持等相应的语言基础能力，让开发者可以以更简洁、更自然的方式开发应用。

- **继承 TS/JS 语言生态**，ArkTS 支持与标准 TS/JS 的高效互操作。开发者可以选择使用标准 JS/TS 进行代码复用或开发，更方便兼容现有生态。

#### 2.1.1.1 适用场景

**一、 兼容 TS 高效语法示例**

ArkTS 保留了 TS 大部分语法特性，兼容 TS 高效语法，提供了许多高效且简洁的语法特性，可以显著提升代码的可读性和开发效率。例如泛型、箭头函数、展开运算符等。

```
// 使用泛型编写可复用的代码
function identity<T>(arg: T): T {
  return arg
}
const output = identity<string>('hello') // 类型推断为 string

// 使用箭头函数，简化函数调用
const add = (a: number, b: number) => a + b

// 使用展开运算符，高效合并数组
const arr1 = [1, 2]
const arr2 = [...arr1, 3]  // [1, 2, 3]
```

全量 ArkTS 语法列表，参考 ArkTS 语言介绍 [3]。

## 二、 ArkTS 拓展高频基础库示例

以线性容器为例，ArkTS 提供以下常用的数据结构，充分考虑了数据访问的速度，实现便捷的操作。

```
1. import { ArrayList } from '@kit.ArkTS' // 导入 ArrayList 模块
2.
3. let arrayList: ArrayList<string> = new ArrayList()
4. arrayList.add('zero') // 增加一个值为'a'的元素
5. arrayList[0] = 'one' // 修改索引为 0 的元素
6. console.info(`result: ${arrayList[0]}`) // 输出：result: one
```

全量 ArkTS 基础库列表，参考 ArkTS 基础类库 [4]。

## 三、 ArkTS 拓展并发能力示例

详见 2.2.1.4 ArkTS 的并发能力 章节。

## 四、 支持声明式 UI 开发示例

ArkTS 提供了声明式 UI 范式、状态管理支持等相应的语言基础能力，帮助开发者提升鸿蒙应用界面开发效率。

```
1. // index.ets
2. @Entry
3. @Component
```

---

[3] 全量 ArkTS 语法列表，更多参见 https://developer.huawei.com/consumer/cn/doc/harmonyos-guides/introduction-to-arkts

[4] 全量 ArkTS 基础库列表，更多参见 https://developer.huawei.com/consumer/cn/doc/harmonyos-guides/arkts-utils

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

  build() {
    Row() {
      Column() {
        Text(this.message)
          .fontSize(50)
          .fontWeight(FontWeight.Bold)
      }
      .width('100%')
    }
    .height('100%')
  }
}
```

有关对 ArkUI 的支持，详情请参见 ArkUI-声明式 UI 开发框架 [5]。

## 五、 继承 TS/JS 语言生态示例

标准 TS/JS 三方库可以直接被 ArkTS 引用，供鸿蒙应用调用。当前 OpenHarmony 三方库中心仓已提供 1000+TS/JS 三方库，供开发者复用已有 TS/JS 资产。ArkTS 引用 TS/JS 示例如下：

```
// lib.ts
export class Test {
  greetings(msg: string) {
    console.log(msg)
  }
}

// app.ets
```

[5] ArkUI 声明式 UI 开发框架，更多参见 https://developer.huawei.com/consumer/cn/arkui/

```
import { Test } from 'lib'  // 和 TS/JS 语法一致，通过 import 关键字导入 TS/JS 中实体
let t: Test = new Test()  // 和 TS/JS 语法一致，通过 new 关键字创建类的实例
t.greetings('hello world')  // 和 TS/JS 语法一致，通过.访问类的实例方法
```

全量三方库列表，参考 OpenHarmony 三方库中心仓 [6]。

### 2.1.2 仓颉高效开发能力概述

- **静态类型系统**：仓颉是静态类型语言，程序中变量和表达式的类型均在编译期确定，并不会在运行期发生改变。静态类型系统能够在编译期发现程序中的错误，减少后期问题调试定位时间。
- **简明语法：**仓颉吸收了静态类型应用开发语言的优秀设计理念和语法特征，因此这些语言的开发者学习仓颉成本较低；同时，仓颉支持类型推断和丰富的语法糖，使编程简洁高效。此外，仓颉还提供元编程能力，允许开发者针对特定业务快速设计领域特定语言（DSL），进一步提升了仓颉的易用性。
- **多范式支持**：仓颉是一个典型的多范式编程语言，对过程式、面向对象和函数式编程都提供了良好的支持。开发者可以灵活选择或混合使用不同的编程范式，来满足不同开发场景的需求。
- **丰富的标准库：**仓颉提供了丰富、通用、性能卓越的标准库，包括集合类库、IO 库、数据库、网络库、安全库、OS 库、数学库、数据结构、排序算法、控制台读取写、正则表达式、time 库、命令行处理库、fs 库、sync 库等，助力开发者快速构建鸿蒙应用。
- **快速成长的生态：**仓颉已基于开源社区构建 183 个三方库，基本覆盖 Top5000 应用所需的高频三方库范围。同时仓颉针对鸿蒙应用开发的痛点场景，针对性优化了多个三方库，如 MarkDown、protobuf、

[6] OpenHarmony 三方库中心仓，更多参见 https://ohpm.openharmony.cn/#/cn/home

pako、bigint、lottie、formula-ffi、editor4cj 等，这些库在功能和性能上都有较好的表现。

#### 2.1.2.1 适用场景

仓颉可适用于的典型业务场景包括但不限于：

- 关注开发效率、性能和安全平衡的的复杂应用开发场景
- DSL 设计和实现，如 UI DSL、数据库访问 DSL、大数据处理 DSL 等
- 特定三方库需求场景，如 MarkDown 渲染、数据序列化等

以其中的 DSL 设计实现为例，仓颉向鸿蒙应用开发者提供了声明式 UI DSL 以提升 UI 场景的开发效率。与传统 DSL 需要特殊的定制实现不同，仓颉语言可通过一系列上文中提及的语言特性实现该声明式 UI DSL。本章节将以该 DSL 的使用和实现原理为例，展示仓颉具备的高效开发能力。

使用仓颉声明式 UI DSL，开发者可以通过类似如下代码完成 UI 页面的定义：

```
// 注：以下为模拟业务场景代码片段，部分函数未给出具体实现，仅供示意参考

@Component
class CustomView {
    @State var count: Int64 = 0
    ...
    func build() {
        Column {
            Text("${count} times")
                .align( Center )
                .margin(top: 50.vp, bottom: 50.vp)
            Button("Click")
                .align( Center )
                .onClick { evt =>
```

```
15.                 count++
16.             }
17.         }.width(100.percent)
18.         .height(100.percent)
19.     }
20.     ...
21. }
```

而上述代码中实际使用了一系列仓颉特性来实现高效的声明式开发：

- 基于元编程特性，使用 UI 组件声明和状态管理的宏 @Component、@State，其会由编译器自动展开生成相应的 UI 组件注册和内部状态处理的代码，简化了 UI 开发的复杂度。
- 基于属性特性，实现对实际状态数据的代理，对外保持读写 count 的形式，但在其内部实现中，通过 get 方法来捕获“读”操作，建立状态与组件的绑定关系；通过 set 方法捕获“写”操作，并通知其绑定的组件进行刷新。
- 基于尾随 Lambda 特性，实现以声明式的语法来描述组件间的分层关系，比如 Column 作为 Text 和 Button 的父组件，决定了子组件的排列方式；同时尾随 Lambda 也可以省略“()”，使语法更简洁。
- 基于命名参数和参数默认值的特性，使传参更清晰简洁，比如在设置 margin 时，只需要设置 top 和 bottom，未设置的参数选择默认值；同时命名参数使得开发者清晰的知道设置的是哪个参数，不用专门去记参数顺序，提高了代码可读性，不易犯错。
- 基于类型扩展能力，为整数类型扩展出带有长度单位的表达能力，比如 100.percent 等价于“100%”，而 50.vp 等价于“50 vp”，其相比只用整数，提供了类型校验的保障；而相比使用 Length 类，语法更简洁，可读性更高。
- 基于类型推断能力，仓颉支持类实例化时省略“new”关键字，通过类型推断实现省略枚举前缀（比如直接用 Center 而不是 Alignment.Center），进一步增强了表达的简洁性。

关于上述特性的详细介绍，请参见仓颉编程语言官方文档[7]。

## 2.2 高性能

### 2.2.1 ArkTS 高性能能力概述

常见的 TS/JS 运行时通常仅支持源码级的运行时解析，并采用解释器或 JIT 的执行模式。而 ArkTS 编译运行时则提供了更为多样的执行能力：它支持将 ArkTS、TS、JS 源代码预编译为字节码文件，运行时可直接加载字节码；同时支持 AOT、解释器和 JIT 的混合执行模式。其中 AOT 编译能显著提升应用的启动性能，使应用在启动时即基于优化后的本地机器码运行。

在模块加载机制上，TS/JS 默认采用预加载方式，即在应用启动时加载所有模块。同时，TS/JS 支持动态加载来延迟模块加载时机，但实现路径依赖异步语义，要求整个调用链保持异步风格，开发成本较高。为解决该问题，ArkTS 运行时引入了 **自动延迟加载**（lazy import）机制：只需在 `import` 语句中添加 `lazy` 关键字，即可使模块在首次使用时按需加载，支持同步与异步调用场景，无需改动业务逻辑结构。

此外，TS/JS 运行时通常为单线程架构，并仅提供基于消息传递的 Worker API。开发者在处理并发任务时需显式封装命令消息及其处理逻辑，使用繁琐。而 ArkTS 提供了 **TaskPool API**，允许开发者以函数方式直接提交并发任务，运

[7] 仓颉编程语言官方文档：https://cangjie-lang.cn/docs

行时自动调度至线程池执行。TaskPool 支持线程池的负载均衡与弹性扩缩容，极大简化了并发编程模型。

针对多线程通信性能问题，TS/JS 的单线程设计导致跨线程传输对象需复制，尤其在处理复杂对象时可能造成明显卡顿。而 ArkTS 引入了 **Sendable 类型**，支持线程间以引用方式共享对象。Sendable 对象在线程传递过程中无需拷贝，具备与 Java 等支持共享内存模型的语言类似的高效并发特性。

#### 2.2.1.1 混合执行模式

ArkTS 编译运行时的执行引擎包含解释器、JIT 编译器和 AOT 编译器，支持混合模式的执行，在应用性能与系统资源之间取得平衡。如下图所示：

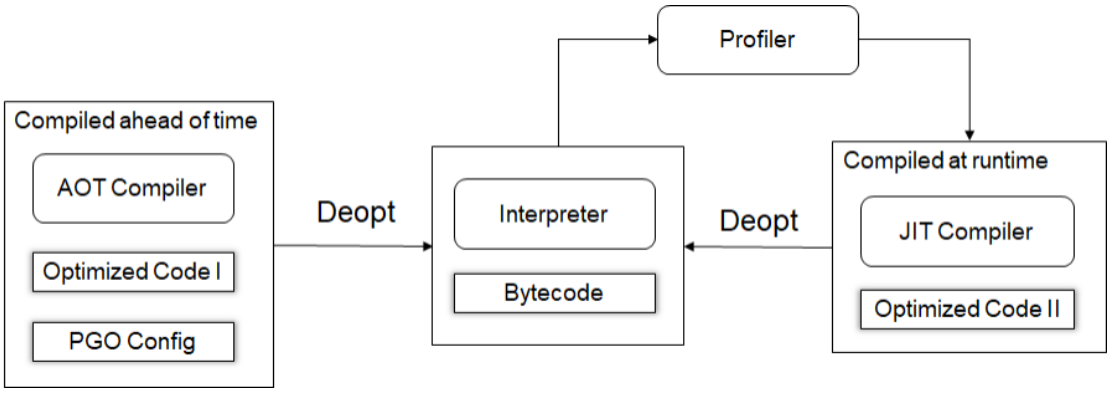


图 2-1：ArkTS 执行引擎

**执行模式原理**

- **解释器**：直接运行前端编译器生成的字节码。启动速度快、内存占用低，但执行性能相对较低。
- **JIT 编译器**：在运行过程中通过 Profiler（性能分析器）收集热点信息，对频繁执行的代码进行即时优化，生成高质量的本地机器码（如上图 Optimized Code II）。具备高运行性能，但启动阶段需要“预热”。

- **AOT 编译器**：在应用运行前，基于静态分析（如类型信息与历史热点信息）生成高质量的目标机器码（如上图 Optimized Code I）。其优势在于启动快、运行快，但会增加安装包体积与内存占用。

解释执行模式可以满足应用开发者一次编译，编出来的 App 包支持在鸿蒙多种设备运行。ArkTS 的优化编译（包括 AOT 闲时编译和 JIT 编译）能支持更高性能的运行，同时也会使用更多的内存空间。

#### 2.2.1.2 启动性能优化——动态加载与懒加载

为了解决大型、复杂应用开发过程中，部分代码编译时被多次拷贝导致包体积增大、文件依赖、代码与资源共享困难以及单例和全局变量污染等问题，同时为了方便开发者代码编写与功能维护，ArkTS 支持应用模块化编译打包运行。

**模块加载方式**

默认情况下，模块的加载是静态加载，即在应用启动时，所有模块都会被加载。这种方式在大型应用中会导致启动时间变长，影响用户体验。

应用开发的有些场景中，有些模块被使用的可能性很低，或者并不需要立即被使用。针对这些模块，ArkTS 提供了两种优化启动性能的方法：动态加载[8]和延迟加载[9]。

- **动态加载：**使用一个异步函数来动态导入模块，返回一个 Promise 对象，实现模块的异步加载，减少了启动时的加载量，提高了启动性能。
- **延迟加载：**import 关键字后面加上 lazy，模块会在第一次被使用时自动同步加载。

**动态加载与延迟加载的对比：**

| 加载方式 | 启动性能优化 | 代码侵入性 | 控制粒度 | 使用建议 |
| --- | --- | --- | --- | --- |
| 动态加载 | ★★★★ | ★★★★ | 精细控制 | 适用于加载时机明确、功能较独立的模块。 |
| 延迟加载 | ★★★★ | ★ | 自动触发 | 适用于快速实现懒加载、依赖较少的模块。 |

[8] 动态加载更多参见：https://developer.huawei.com/consumer/cn/doc/harmonyos-guides-V5/arkts-dynamic-import-V5

[9] 延迟加载更多参见：https://developer.huawei.com/consumer/cn/doc/harmonyos-guides-V5/arkts-lazy-import-V5#%E4%BD%BF%E7%94%A8%E6%96%B9%E5%BC%8F

### 2.2.1.3 SmartGC——感知场景的内存回收

ArkTS 运行时选择最为常见的基于对象追踪（即 Tracing GC）算法。GC 的性能通常以暂停时间、内存使用和吞吐率来衡量，而在 GC 的算法中又不能对这三项做到面面俱到，一般的 GC 算法只能针对其中一项或者两项做倾向性优化。ArkTS 运行时针对鸿蒙应用场景做了精细化的 GC 策略调整，在不同的场景采取不同的策略和算法达到场景最优化选择：

- 在用户敏感的场景：GC 以用户体验优先，即专注于暂停时间的最小化。
- 在后台运行的场景：GC 以吞吐率和内存使用优先，尽量回收内存以使得前台应用可以使用更充足的内存。

**一、 敏感场景**

在应用性能敏感场景，通过将 GC 触发水线临时调整到线程的堆较大值来避免触发 GC，避免 GC 可能导致的应用卡顿。

当前支持的敏感场景包括：应用冷启动、应用滑动、应用点击页面跳转、超长帧。

**二、 闲时 GC**

用算法检测应用空闲场景，根据检测到的空闲等级触发不同类型的压缩 GC，尽量回收空闲应用内存。闲时 GC 既可以减少系统整体内存负载，同时确保应用再次进入敏感场景时内存保持最佳，间接提升性能，减少 GC 导致的应用丢帧。

#### 2.2.1.4 ArkTS 的并发能力

ArkTS 提供的 TaskPool 和 Worker 均支持多线程并发能力。TaskPool 的工作线程会绑定系统的调度优先级，并支持负载均衡（自动扩缩容）。相比之下，Worker 需要开发者自行创建，不支持设置调度优先级。因此，性能方面 TaskPool 优于 Worker，推荐在大多数场景中使用 TaskPool。

在 ArkTS 并发模型中，普通对象所属的内存是隔离的，不会出现线程竞争同一内存资源的情况，开发者无需处理内存上锁相关的问题，提高开发效率。对普通对象的线程间通信，ArkTS 采用了标准的 Structure Clone 算法（序列化和反序列化），此时需要进行深拷贝，性能开销可能较大。对此，ArkTS 还提供了 Sendable 对象的共享能力来优化线程间通信开销。

**一、TaskPool[10]**

TaskPool 为应用程序提供多线程环境，降低资源消耗、提高系统性能，无需管理线程生命周期。其运作机制示意图如下：

[10] TaskPool 更多参见：https://developer.huawei.com/consumer/cn/doc/harmonyos-guides/taskpool-introduction

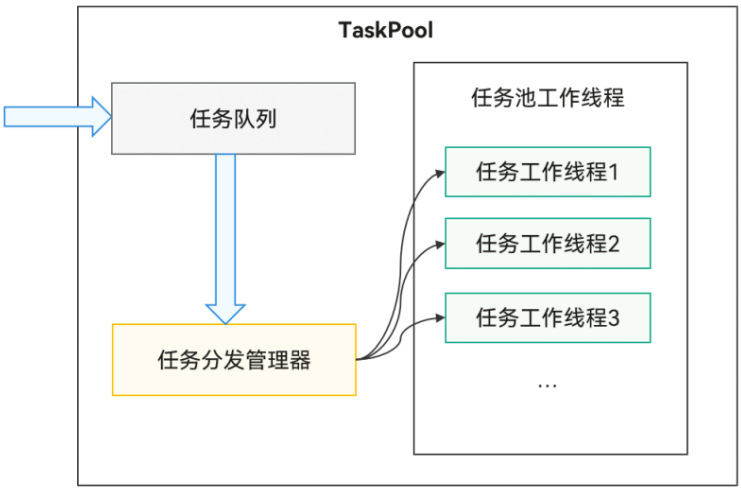


图 2-2：ArkTS TaskPool 工作机制

TaskPool 支持开发者在宿主线程提交任务到任务队列，系统选择合适的工作线程执行任务，再将结果返回给宿主线程。接口易用，支持任务执行、取消和指定优先级，同时通过系统统一线程管理，结合动态调度及负载均衡算法，可以节约系统资源。系统默认启动一个任务工作线程，任务多时会扩容。工作线程数量上限取决于设备的物理核数，内部管理具体数量，确保调度和执行效率最优。长时间无任务分发时会缩容，减少工作线程数量。

## 二、Worker[11]

[11] Worker 更多参见：https://developer.huawei.com/consumer/cn/doc/harmonyos-guides/worker-introduction

Worker 的主要作用是为应用程序提供一个多线程的运行环境，满足应用程序在执行过程中与宿主线程分离，在后台线程中运行脚本进行耗时操作，避免计算密集型或高延迟的任务阻塞宿主线程。

每个 Worker 子线程和宿主线程拥有独立的实例，包含基础设施、对象、代码段等。因此，启动每个 Worker 存在一定的内存开销，需要限制 Worker 子线程的数量。Worker 子线程和宿主线程通过消息传递机制通信，利用序列化反序列化机制完成参数对象的重建（注：Sendable 对象直接引用共享）。

## 三、 Sendable[12]

ArkTS 提供了 Sendable 对象类型，在并发通信时支持通过引用传递来解决并发通信开销大的问题。Sendable 对象为可共享的，其跨线程前后指向同一个 JS 对象。如果底层是 Native 实现，还需要考虑线程安全性。通信过程如下图所示：

[12] Sendable 更多参见：https://developer.huawei.com/consumer/cn/doc/harmonyos-guides/arkts-sendable

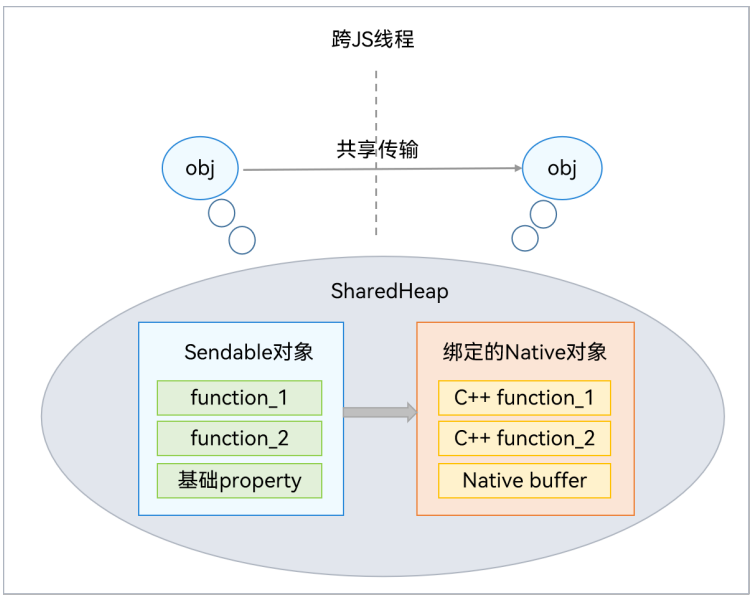


图 2-3：ArkTS 对象共享机制

当多个并发实例尝试同时更新 Sendable 数据时，会发生数据竞争，例如 ArkTS 共享容器的多线程操作。因此，ArkTS 提供异步锁机制来避免不同并发实例间的数据竞争，并提供了异步等待机制来控制多线程处理数据的时序。同时，还可以通过对象冻结接口将对象冻结为只读，从而避免数据竞争问题。Sendable 对象提供了并发实例间高效的通信能力，即引用传递，适用于开发者自定义大对象需要线程间通信的场景，例如子线程读取数据库数据并返回给宿主线程。

**总结与选型建议**

| 并发能力 | 适用场景 | 性能优化建议 | 生命周期管理 | 推荐级别 |
| --- | --- | --- | --- | --- |
| TaskPool | 任务型并发 | ★★★★ | 自动 | ☑ 推荐 |
| Worker | 交互式后台服务，定时器等 | ★★★ | 手动 | ⚠ 特殊场景使用 |
| Sendable | 大对象跨线程传输 | ★★★★ | 自动（GC 回收） | ☑ 大对象场景推荐 |

### 2.2.2 仓颉高性能能力概述

仓颉具有静态类型系统，且其源码被静态编译为直接执行的机器指令。这种静态编译执行方式具有更高的启动性能和安全性。借助仓颉编译器强大的静态分析能力，低效或有风险的代码在应用构建过程中识别，开发工具自动帮助或提示开发者优化代码，达成最佳应用体验。

### 2.2.2.1 静态类型和静态编译优化

仓颉编译器具有多层级优化能力，通过前端-后端-运行时融合的垂直分析优化技术（图 2-4），仓颉在计算机语言基准测试 Benchmarks Game 上展现了优越的基础性能 [13]。

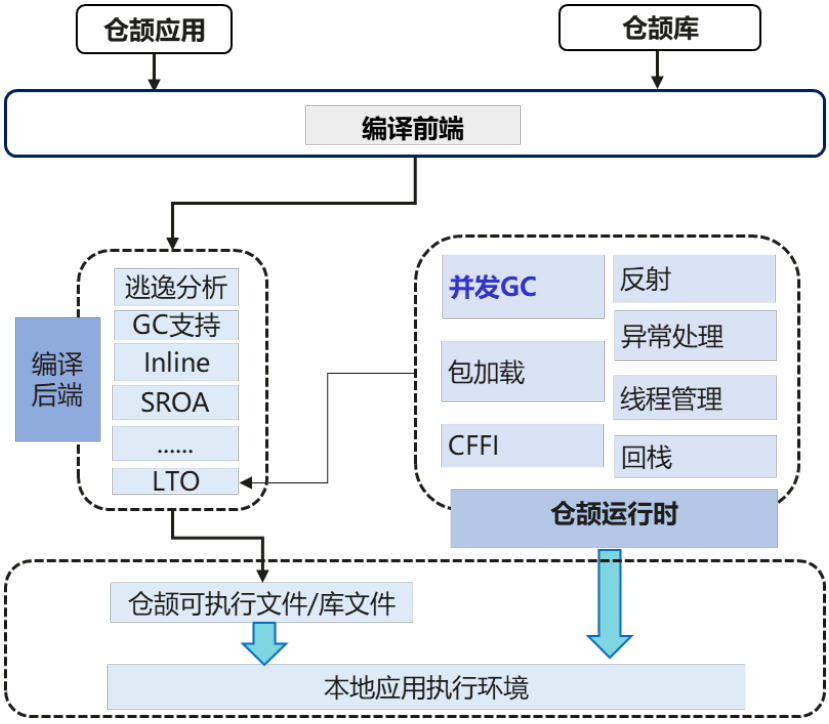


图 2-4：仓颉应用的编译执行架构

仓颉的静态编译先进技术包括：

- 语言相关的 high-level 编译前端：前端检查仓颉源码的规范性和安全性，对其开展 high-level 语言相关优化（面向对象优化、去虚化、内联等），然后编译为低层级中间表示文件输出。

[13] 仓颉基础性能参见：https://developer.huawei.com/consumer/cn/doc/cangjie-guides-V5/cj-wp-performance-V5

- 多重优化 Pass 的编译后端：基于前端输出的低层级中间表示文件开展深入的静态分析和优化（逃逸分析、常量传播、内联、循环展开、循环不变量外提、向量化等），生成高效的可执行机器指令。
- 仓颉对象在编译时具有确定类型，其对象布局在编译时确定，所以成员数据的访问语句在编译时能生成简洁高效的内存访问指令。
- 仓颉的虚函数调用、接口函数调用通过仓颉编译器的去虚化技术可以转化为直接调用，显著减少调用开销；对于具有运行时多态的接口函数调用，仓颉编译器和运行时通过 inline cache 大幅提升其调用性能。
- 仓颉值类型提供了更紧凑的数据排布和访问方式，值类型变量减少了间接寻址和 cache miss，具有更好的空间局部性. SROA(Scalar Replacement of Aggregates)优化对值类型更友好。通过 SROA，值类型变量的数据可被直接映射到物理寄存器，避免了内存访问操作。

仓颉语言用包组织源码，支持应用功能以包为粒度按需加载。

#### 2.2.2.2 低时延高效率的自动内存管理

**一、极低时延**

时延敏感性是鸿蒙移动应用的关键特征之一。造成应用时延高的因素有很多，从编程语言角度看，GC 暂停耗时是关键因素之一。与现有的 STW GC 或近似并发 GC（图 2-5）不同，仓颉的并发 GC 采用轻量同步机制，具有更短的 GC 暂停耗时，仓颉应用线程完成 GC 同步的平均耗时小于 2 毫秒，典型情况下完成一次 GC 同步的耗时在百微秒量级。仓颉并发 GC 可以帮助应用大幅减少 GC 暂停导致的丢帧，是时延敏感场景的优选编程语言，如：

- 长列表、复杂页面快速滑动
- 点击响应要求高的页面

- 长列表、长图快速滑动
- 视频直播

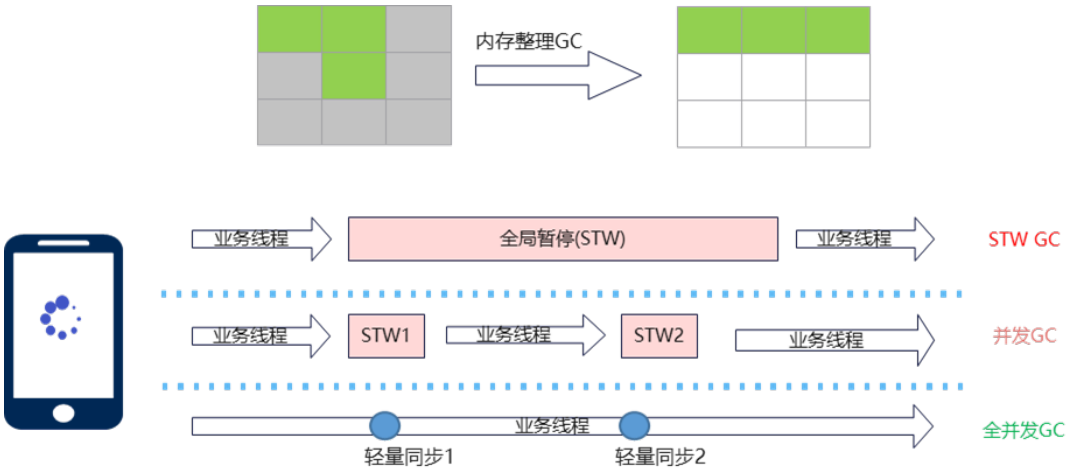


图 2-5：仓颉 GC 轻量同步机制

## 二、精简对象布局

仓颉对象采用了精简的内存布局。对象内存中仅保留一个 8 字节的头记录其类型信息，其余数据都是有效内容，用仓颉语言开发的应用在运行时占用更少的内存。

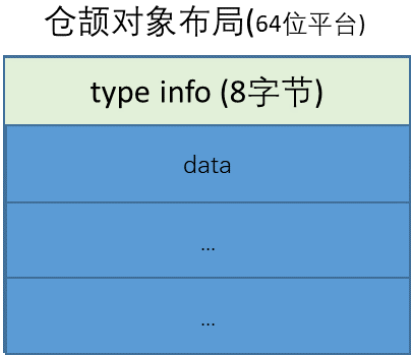


图 2-6：仓颉对象基础布局

## 三、 优化内存峰值

仓颉 GC 采用了内存整理(compact)技术，通过把存活对象搬移到指定的一块连续内存中，全部回收堆内存中的死亡对象及其中的内存碎片，减少了堆内存用量。仓颉 GC 采用的内存整理技术可实现内存块一边整理一边释放，降低 GC 过程中的内存峰值（图 2-7）。

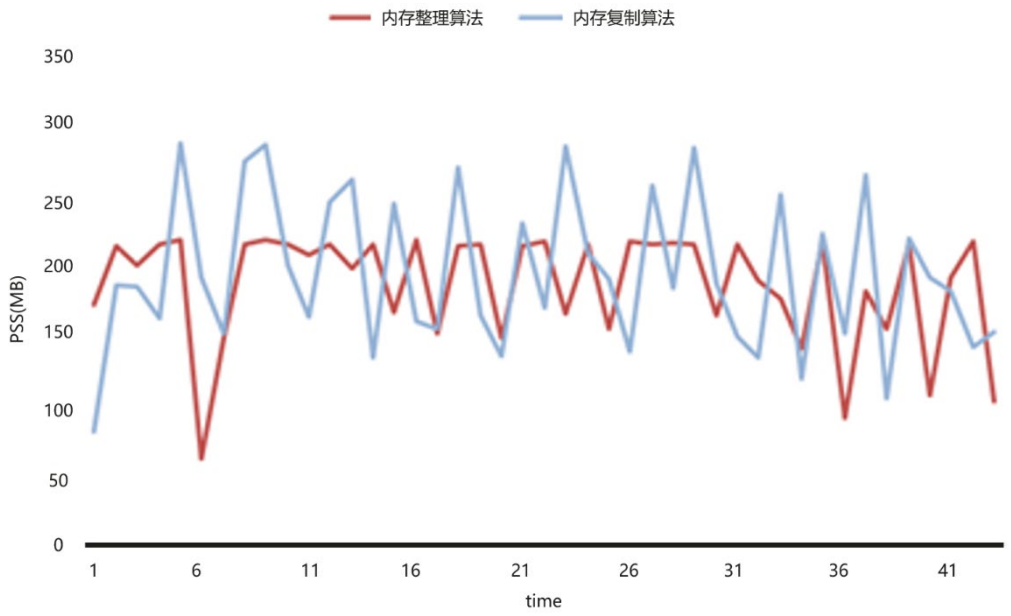


图 2-7：仓颉内存整理技术降低内存峰值

## 四、 紧凑的值类型

仓颉的值类型也可以用于优化堆内存。值类型数据作为局部变量时占用的内存直接分配在函数栈空间里，在函数退出时随之释放。相比堆对象，值类型变量具有更高的内存分配和回收效率，内存回收更及时。通过在应用代码中合理地使用值类型，可以减少堆内存用量，降低 GC 负载。

仓颉编译器也能通过逃逸分析识别出函数内的局部对象，像对待值类型变量一样把它分配在栈空间，开发者不需改动源码即可与手动改为值类型具有等同的内存优化效果。

综上，仓颉自动内存管理技术具有时间空间多维度优势，是时延敏感场景和内存敏感场景（如内存受限的中低端设备）的优选编程语言。

#### 2.2.2.3 简洁轻量的并发编程

仓颉采用数据共享的多线程模型，提供了轻量用户态线程和高效易用的无锁并发数据结构让并发编程变得轻松，将高效并发处理的能力直接置于开发者的手中。

在仓颉提供的轻量化线程模型中，开发者可以通过简单的 spawn 语法创建仓颉线程。与业界一些其他语言的使用 async/await 语法的并发模型不同，使用仓颉的轻量化线程模型，开发者无需在编程过程中手动标记（如用 async 标记异步函数并用 await 标记其调用点），也彻底杜绝了这些标记的“传染性”（包含 await 的函数必须标记为 async）导致的“函数染色”问题，大大降低了并发编程的复杂性。

在运行层面，仓颉线程是用户态线程，与传统的操作系统线程相比，其实现完全在用户空间进行，不依赖操作系统的线程管理，这从根本上减少了线程创建和销毁的开销，在性能上具有明显优势。在线程调度上，仓颉使用了 M:N 线程模型，即 M 个仓颉线程在 N 个 native 线程（通常即为操作系统线程）上调度执行，其中 M 和 N 不一定相等。每个仓颉线程都受到底层 native 线程的

调度执行，并且多个仓颉线程可以由一个 native 线程执行。每个 native 线程会不断地选择一个就绪的仓颉线程完成执行。仓颉线程的调度支持抢占，如果仓颉线程在执行过程中发生阻塞（例如等待互斥锁的释放），那么 native 线程会将当前的仓颉线程挂起，并继续选择下一个就绪的仓颉线程。发生阻塞的仓颉线程在重新就绪后会继续被 native 线程调度执行。

对于仓颉线程间的数据共享和同步，仓颉也提供了一系列简洁易用的机制来确保线程安全，主要包括：

- 原子操作：不可分割的操作序列，其执行过程要么完全完成，要么完全不执行，不会因线程调度或中断导致中间状态
- 互斥锁和 synchronized 机制：对临界区加以保护，使得任意时刻最多只有一个线程能够执行临界区的代码
- 条件变量：协调仓颉线程间的同步，允许仓颉线程在条件不满足时主动挂起，并在条件满足后被唤醒
- 线程局部变量：每一个仓颉线程都有它独立的一个存储空间来保存这些线程局部变量，每个仓颉线程可以安全地访问他们各自的线程局部变量，而不受其他仓颉线程的影响

更进一步，仓颉提供了基于细粒度并发算法实现的无锁并发对象，而用户通过调用并发对象的接口来操作多线程共享内存，从而实现：

- 无锁编程体验：用户通过接口调用实现高效的共享内存并发访问。
- 并发安全保障：仓颉并发对象的接口可保证无数据竞争，核心接口具有并发原子性。
- 提升性能：仓颉并发对象的设计使用细粒度并发算法。

- 保证并发原子性：仓颉并发对象的核心方法具有并发“原子性”，即从用户视角来看，该方法调用执行不会被其它线程打断。

关于并发模型相关特性的更详细介绍，请参见仓颉编程语言官方文档[14]。

#### 2.2.2.4 适用场景

仓颉高性能静态编译和包按需加载让仓颉应用启动/执行耗时更短、开销更低，是计算密集场景的优选编程语言，如：

- 应用首页/首屏：对启动速度/响应速度要求高的页面、冷启动性能要求高的应用页面
- 高性能中间件：智能设备的底层基础服务、数据实时采集计算
- 高性能三方库替换：替换部分 C/C++模块提供内存安全的对等功能库
- 基于仓颉并发编程模型，仓颉可适用于的典型业务场景包括但不限于：
- 长列表页面：如联系人列表中的头像和信息获取、商品列表中的视频、图片和文字描述获取等
- 重载计算任务：如图片/视频编解码、压缩/解压缩、加密/解密、各类算法模型的并行运行等
- UI 主线程上的任务卸载：如端口监听任务、中台和后台任务、业务模块加载任务等

以其中的联系人列表中的头像和信息获取为例，在仓颉实现代码如下所示：

```
1.  // 注：以下为模拟业务场景代码片段，部分函数未给出具体实现，仅供示意参考
2.
3.  func fetchData(url: String) {
```

[14] 仓颉编程语言官方文档：https://cangjie-lang.cn/docs

```
    let response = http_get(url)
    process(response)
}

func fetchListData() {
    let urls = ["https://example.com/data1", "https://example.com/data2", ...]
    let futures = ArrayList<Future<Unit>>()
    for (url in urls) {
        let fut = spawn {fetchData(url)} // 创建仓颉线程进行网络请求
        futures.add(fut)
    }
    for (fut in futures) { // 等待所有仓颉线程完成
        fut.get()
    }
}
```

在这个例子中，spawn 创建的各个仓颉线程将独立地执行 fetchData 函数。仓颉的运行时环境会自动调度这些仓颉线程，而开发者只需关注业务逻辑的实现。最后通过 get 函数接口等待仓颉线程完成并获取结果，确保主线程能够同步地获取所有结果。

进一步地，对于需要在不同的仓颉线程中安全访问共享数据的场景，如下示例代码展示了如何使用 synchronized 关键字来保护共享数据。

```
import std.sync.*
import std.time.*
import std.collection.*

var count: Int64 = 0
let mtx = ReentrantMutex()

func concurrentCounter(): Int64 {
    let list = ArrayList<Future<Unit>>()

```

```
    // creat 1000 threads.
    for (i in 0..1000) {
        let fut = spawn {
            sleep(Duration.millisecond) // sleep for 1ms.
            // 通过 synchronized 即可完成对共享数据的保护，减少复杂易错的基于锁的编程处理
            synchronized(mtx) {
                count++
            }
        }
        list.add(fut)
    }

    // Wait for all threads finished.
    for (f in list) {
        f.get()
    }

    println("count = ${count}")
    return 0
}
```

## 2.3 安全

### 2.3.1 ArkTS 安全能力概述

ArkTS 不仅在语言层面引入了类型系统、空值安全等特性，在编译工具链和运行时的设计上也提供了额外的安全机制。其目标是在保障开发者效率的同时，降低运行时攻击面，提升系统整体可信度。

#### 2.3.1.1 应用字节码文件的合法性校验

为了防止恶意篡改应用字节码并上传至应用市场，鸿蒙应用市场在上架审核阶段引入**字节码合法性校验机制**，具体包括：

- 检查所有字节码文件是否符合 ArkTS 字节码规范。
- 规避字节码级别的运行时崩溃或被利用行为。

该机制削减了从分发渠道源头控制恶意应用流入用户终端的风险。

#### 2.3.1.2 代码签名与运行时验签

ArkTS 对所有上架应用进行代码签名，并在运行时加载字节码前执行签名校验：

- **签名机制**：在打包阶段对应用内容进行数字签名，包含字节码文件。
- **运行时校验**：ArkTS 运行时在执行字节码前验证签名完整性。
- **篡改防护**：一旦检测到签名异常或字节码被篡改，应用将无法执行。

这一机制防止了上线后应用注入代码或篡改行为。

### 2.3.2 仓颉安全能力概述

仓颉是静态类型语言，程序中所有变量和表达式的类型都是在编译期确定的，并且禁止隐式类型转换，即在程序运行过程中不会发生改变，能够在编译期尽量早的发现程序中的错误，提升程序安全性。

仓颉在编译和运行时支持多种安全检测，如数组越界、除 0、整型溢出、移位检查等。以数组越界检查为例，支持编译时和运行时安全检查。

仓颉采用 tracing GC 技术，通过在运行时跟踪对象之间的引用关系，来识别活动对象和垃圾对象。垃圾收集（GC）是一种自动内存管理机制，它能够自动识别和回收不再需要使用的对象，将开发者从手工释放内存中解放出来，不仅可以提高开发效率，还能有效避免各种常见内存错误，提升程序的安全性。

仓颉支持运行时地址随机化、堆栈不可执行、控制流完整性等漏洞利用缓解机制，能有效避免运行时漏洞利用，让攻击者难以利用漏洞控制应用执行流并窃取、篡改应用客户数据。

#### 2.3.2.1 适用场景

鸿蒙应用中部分场景对安全有较高诉求，例如金融类应用需要在后台使用多个线程处理客户金融数据，并在不同线程之间进行数据传递，对数据安全性有较高诉求。使用仓颉完成金融类业务逻辑功能实现，在不增加额外安全检查的基础上，可保障默认安全。

仓颉编译器在编译时和运行时进行安全检测，提升应用安全性，示例如下。

```
// 错误示例：整型与浮点型隐式转换
let a: Int32 = 10
let b: Float64 = a      // 编译错误：需要显式转换
let c: Float64 = 3.14 + 5 // 编译错误：Int 与 Float 不能直接运算

// 正确写法
let validB = Float64(a)  // 显式转换
```

```
let validC = 3.14 + 5.0  // 统一类型后运算
```

```
//类、接口类型转换
open class Animal {}
class Cat <: Animal {}

// 错误示例：向下转型未显式声明
let animal: Animal = Cat()
let cat1: Cat = animal   // 编译错误：需要显式转换

// 正确写法
let cat2 = animal as Cat  // 返回 Option<Cat>
if (let Some(value) <- cat2) {
    // 安全访问转换后的对象 value，value 的类型为 Cat
}
```

```
import std.reflect.*
//类的成员字段的访问时的类型检查
public class Account {
    private var _balance: Float64 = 0.0

    public mut prop balance: Float64 {
        get() { _balance}
        set(v) { _balance = v}
    }
}

main() {
    let acc = Account()
    acc.balance = 100.5     // 合法操作
    //acc.balance = "500"     // 编译错误：String 无法转为 Float64

    // 反射操作属性
```

```
    let propInfo = TypeInfo.of(acc).getInstanceProperty("balance")
    propInfo.setValue(acc, 200.0)  // 合法
    propInfo.setValue(acc, true)   // 运行时错误
}
```

```
//泛型字段检查
public class Box<T> {
    public var content: T

    init(c: T) {
        content = c
    }
}

main() {
    let intBox = Box<Int64>(100)
    intBox.content = 200    // 合法
    intBox.content = "text"  // 编译错误：String 无法转为 Int64

    let strBox = Box<String>("init")
    strBox.content = 100    // 编译错误：Int64 无法转为 String
}
```

## 2.4 跨平台

### 2.4.1 仓颉支撑跨 OS 平台能力概述

面对业界不同的移动 OS 平台，应用开发对语言的跨移动 OS 平台运行能力也有强烈的诉求。而部分 OS 平台因安全和商业政策上的考量，不允许三方应

用通过端侧编译（如：JIT）生成可执行代码，导致基于解释执行的语言跨平台方案会存在性能问题。

仓颉支持针对不同 OS 平台静态编译至二进制，无需依赖 JIT 等机制，即可允许开发者实现跨 OS 平台代码共享。仓颉支持的 OS 平台范围：鸿蒙，Android，iOS，MacOS，Windows 和 Linux。

在此基础能力上，仓颉正在持续构建和完善跨移动 OS 平台应用开发的相关工程能力，具体的路线图和版本发布计划请参见 **3.3.2.4 章节**。使用仓颉语言开发的应用，未来可借助仓颉跨平台能力和工程方法轻松运行于多个OS平台，从而减轻多个平台代码开发和维护的工作量。

## 2.5 技术资产保护

### 2.5.1 ArkTS 技术资产保护能力概述

ArkTS 为开发者提供了源码混淆工具 ArkGuard，其主要针对 ArkTS、TS 和 JS 语言提供基础混淆功能，将代码中的变量名、函数名、类名、文件名等替换为简短无意义的名称，增加通过逆向产物猜测其用途的难度。

#### 2.5.1.1 ArkGuard 混淆能力

ArkGuard 支持基础的名称混淆，不支持控制混淆、数据混淆等高级混淆功能，已有名称混淆选项汇总如下表：

| 功能 | 选项 |
| --- | --- |
| 属性名称混淆 | -enable-property-obfuscation |

| | |
|---|---|
| 字符串属性名称混淆 | -enable-string-property-obfuscation |
| 顶层作用域名称混淆 | -enable-toplevel-obfuscation |
| 导入导出名称混淆 | -enable-export-obfuscation |
| 文件名混淆 | -enable-filename-obfuscation |

### 2.5.2 仓颉技术资产保护能力概述

仓颉提供了外形混淆、数据混淆、控制流混淆等多种混淆技术用于保护开发者的软件资产，提升攻击者逆向攻击仓颉软件的难度。攻击者可采用逆向工程技术对程序进行攻击，并获取程序的符号名、路径信息和行号信息、特征字符串和特征常数，以及控制流信息。仓颉混淆技术可以对这些信息进行混淆和隐藏，让攻击者难以借用这些信息辅助理解程序的运行逻辑。

- **外形混淆**
  - 符号混淆：通过使用随机名称生成算法，将符号名替换为无关的随机字符串，隐藏符号信息。
  - 路径信息混淆：将函数路径信息统一替换为字符串 SOURCE，隐藏函数路径信息。
  - 行号混淆：统一将行号替换为 0，隐藏行号信息。
  - 函数重排：随机重新排列函数顺序，隐藏函数二进制编译特征。
- **数据混淆**
  - 字符串加密：编译时将字符串进行加密，隐藏字符串信息。
  - 常量混淆：将普通常量运算替换为等价的、更难理解的算数运算，以此隐藏常量特征。

- **控制流混淆**

  ○ 虚假分支：在程序中插入虚假分支，生成不透明谓词作为分支跳转条件，对抗静态符号执行求解器，隐藏分支信息。

  ○ 控制流平坦化：将函数基本块组织为 switch-case 形式，隐藏基本块之间的跳转关系。

  仓颉混淆详细介绍请见《仓颉编程语言白皮书》[15]。

#### 2.5.2.1 适用场景

代码资产在本文中指防止攻击者对开发者的编译结果实施逆向攻击，从而获取开发者源代码。对代码资产安全有诉求或金融类应用业务逻辑是重点保护对象，同时根据《 JRT 0092-2019 移动金融客户端应用软件安全管理规范》“5.3.3 抗攻击能力”章节要求“客户端代码应使用代码混淆”，因此需要进行代码资产保护。

金融类应用对运行时安全和代码资产安全尤为关注，希望通过运行时安全能力保护伙伴的应用用户数据在应用使用过程中不被泄露、篡改、窃取等，通过代码资产安全能力保护伙伴应用代码逻辑不被逆向。

- 外形混淆举例：使用随机名称生成算法，将符号名替换为无关的随机字符串，隐藏符号信息。

  ○ 混淆前：逆向工具可以直接看到原始符号名

[15] 仓颉编程语言白皮书：https://developer.huawei.com/consumer/cn/doc/cangjie-guides-V5/cj-wp-obfuscate-V5

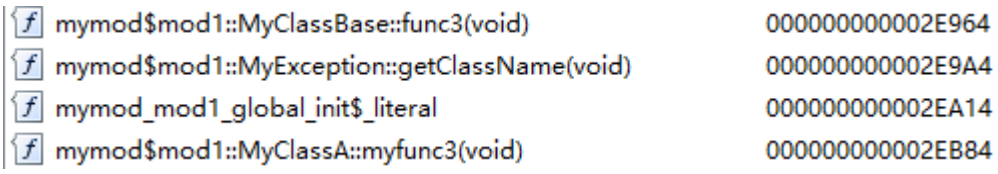


图 2-8：混淆前符号信息示例

○ 混淆后：逆向工具无法看到原始符号名

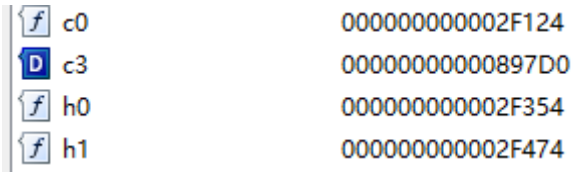


图 2-9：混淆后符号信息示例

- 数据混淆举例：编译时将字符串进行加密，程序初始化时*动态解密*，隐藏字符串信息。

○ 混淆前：逆向工具可以看到敏感字符串内容

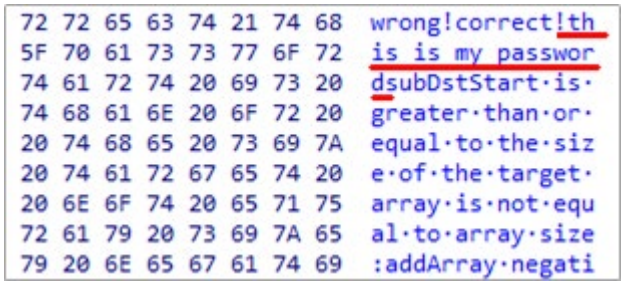


图 2-10：混淆前敏感字符串示例

○ 混淆后：逆向工具无法看到敏感字符串内容

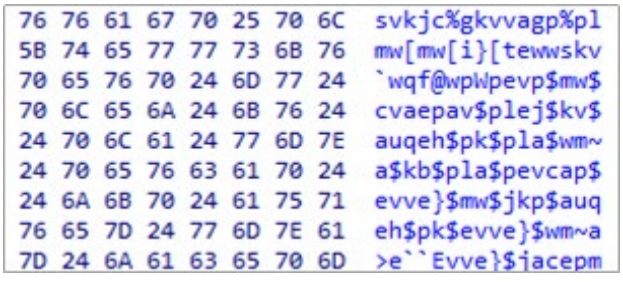


图 2-11：混淆后敏感字符串示例

- 控制流混淆举例：将函数基本块组织为 switch-case 形式，隐藏基本块之间的跳转关系。

○ 混淆前：逆向工具可以轻易分析基本块之间的跳转关系，即执行的控制流

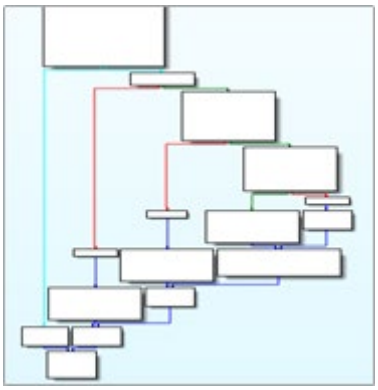

图 2-12：混淆前程序控制流示例

○ 混淆后：整个函数变成一个大的 switch-case 结构，难以分析基本块之间的跳转关系

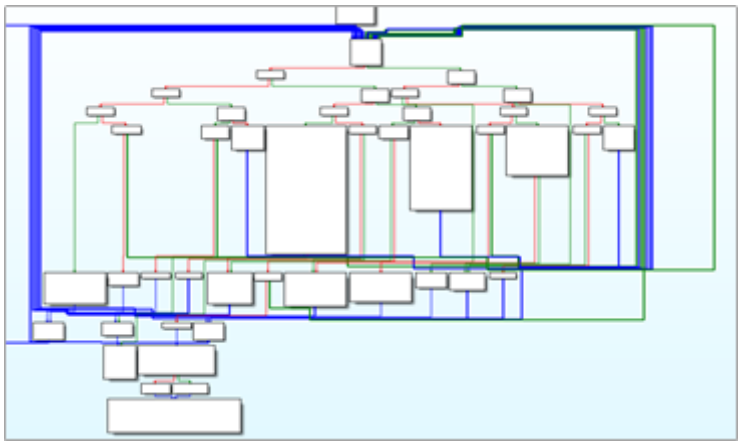

图 2-13：混淆后程序控制流示例

### 2.5.3 技术资产保护建议

与其他代码混淆工具一样，ArkTS 及仓颉的混淆工具只能在一定程度上增加逆向工程的难度，并不能彻底阻止逆向工程，对于源码安全有高要求的开发

者，建议考虑使用应用市场提供的应用加密[16]功能或者三方安全加固等安全措施来保护代码。

[16] 应用加密更多参见：https://developer.huawei.com/consumer/cn/doc/harmonyos-guides-V5/code-protect-V5

# 3 第三章：鸿蒙编程语言演进策略

## 3.1 语言演进整体策略

### 3.1.1 ArkTS 语言演进策略

当前 ArkTS 语言的整体设计和实现思路是基于 TS 的编译类型检查规则和 JS 的运行时语义，前端通过语法糖提供声明式范式扩展，运行时在动态类型的基础架构上扩展局部的共享内存并发能力。它提供了无缝兼容和复用 TS/JS 生态的能力，同时也通过有限扩展提供了鸿蒙应用开发场景所必需的声明式和并发编程能力，有效支持了鸿蒙初期应用生态的快速构建和生态丰富。同时，在鸿蒙应用生态发展过程中，我们不断获取到对 ArkTS 语言更多的新特性诉求。因此，结合鸿蒙应用生态繁荣对编程语言在开发效率、运行性能、安全等方面的优化需求以及语言的长期可持续发展演进需求，ArkTS 也将在已有基础上持续演进。

ArkTS 未来演进中，将会进一步定义和完善语言规范（含语法演进和类型系统增强等），对声明式 UI 范式和并发编程提供语言的支持。

ArkTS 演进也将提供基于语言规范的编译器实现，去除 TS/JS 转换流程和对 TS 工具链的依赖，通过完善的编译器静态分析和检查能力保证 ArkTS 代码类型信息的准确和完备，并带来更快的编译速度和更小的编译内存占用。同时提供

通用的编译器插件和扩展机制，方便基于特定的业务需求定制编译流程和提供更强的编译期 AOP 能力。

同时，运行时引入类型信息和对应的语义实现，基于确定的类型信息进行进一步增强的编译、并发和模块加载等能力优化，从而提升整体运行性能。

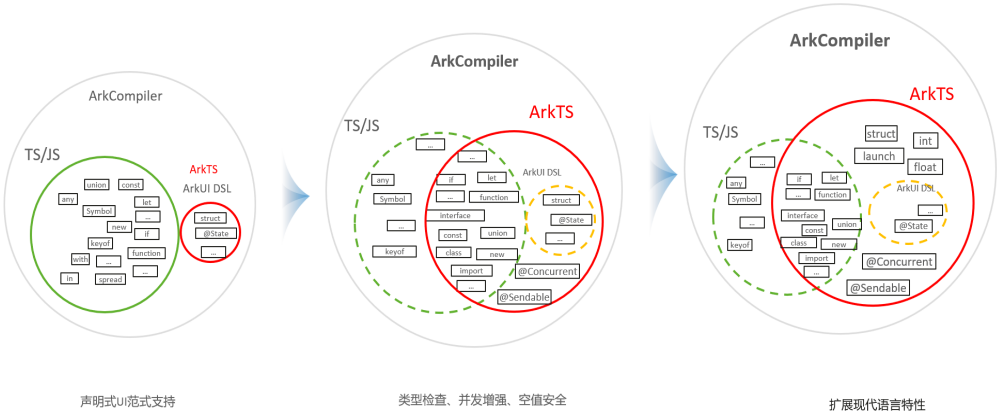


图 3-1：ArkTS 演进路线

### 3.1.2 仓颉语言演进策略

仓颉始未来将持续深耕高效开发、高性能、强安全等领域，持续提升高效开发体验，提供默认高性能和强安全能力；在跨平台和智能化领域持续完善和探索，为鸿蒙应用开发者提供一码三端、AI Agent 编程能力，并将仓颉打造为鸿蒙生态应用开发的首选静态编程语言。仓颉未来探索方向如下：

- 高性能：通过 Actor、并发优先级算法、结构化并发提性能；通过内存所有权机制降内存；通过硬件深度协同的优化技术降功耗。
- 强安全：通过安全宏、FFI 安全检查增强编译阶段安全能力；通过前向控制流完整性技术增强运行阶段安全能力；通过数据流分析技术增强应用级别的安全能力构建。

- 跨平台：完善跨 OS 平台的线下和线上调优工具链，优化跨 OS 平台的调试体验，优化跨平台框架持续提升鸿蒙应用跨 OS 平台代码占比，持续探索仓颉与 Java/OC/Swift/Kotlin 互通。

- 智能化：未来将从 AI 辅助编程和智能应用极简开发两个方面提升仓颉智能化能力。在 AI 辅助编程方面，将从代码续写、UI 生成、卡片生成等实际业务场景进行文件级/模块级 AI 辅助编程能力建设，从应用生成等场景进行应用级 AI 辅助编程能力建设，并探索从人主导到 AI 主导的开发形态转变；在智能应用的极简开发方面，将通过构建 Agent DSL 支持声明式智能应用开发范式，详细参考 3.2 章节内容。

## 3.2 智能化演进策略

### 3.2.1 仓颉智能应用开发能力规划

仓颉通过元编程能力和 DSL 能力构建 Agent DSL 能力，包括：单 Agent 编程开发、多 Agent 协同、支持 MCP 协议等。未来，将持续深化仓颉 Agent DSL 与 AI 技术的融合，推动其在教育、医疗、金融、制造等领域的广泛应用，真正实现“语言即智能”的愿景。

#### 3.2.1.1 Agent DSL 的设计理念

AI Agent 所代表的是一种动态、不确定、具有意图驱动行为的计算范式。因此，仓颉 Agent DSL 设计理念：

- **声明式编程**：允许开发者以声明方式定义 Agent 的行为模式、交互机制与执行逻辑。

- **细粒度控制与抽象统一**：既提供对 Agent 执行流程的精确控制能力，又保持高层次语义的一致表达。

- **可扩展性与开放性**：支持第三方模型、工具（如 MCP 协议）的无缝集成，构建可持续演进的 Agent 生态。

#### 3.2.1.2 Agent 核心要素的语言级支持

现代 AI Agent 的开发通常围绕模型、提示词、规划和工具四大核心要素展开。

- **模型：**通过提供统一的模型调用方式，屏蔽不同模型接入时的差异，降低开发者接入成本。
- **提示词：**将传统的文本提示词提升为语言的一等公民，开发者可以结构化地定义 Agent 的行为模式、目标与预期输出。让开发者只需关注“做什么”，而语言运行时负责“怎么做”。
- **规划：**开发者可以声明式地定义 Agent 的执行流程。
- **工具：**提供统一接口，支持外部工具和服务的无缝接入。无论是仓颉自身编写的模块，还是已有的 Agent 工具生态（如各类 MCP 服务），均可被自然地嵌入 Agent 的执行流程中，从而增强其功能边界。

#### 3.2.1.3 多 Agent 协同

多个 Agent 的协同工作逐渐成为主流趋势，因此，仓颉 Agent DSL 引入了一套简洁直观的流式语法，用于描述 Agent 之间的交互关系与协作模式。如下：

- **线性协同：**适用于阶段化的处理流程，例如数据依次经过三个 Agent 的流水线处理。
- **主从式协同：**适用于层次化组织结构，例如一个主 Agent 负责任务分解与结果整合，子 Agent 则专注于特定子任务。
- **自由协同：**适用于基于共享上下文的松耦合交互，例如多个 Agent 可以自由发言、响应事件，并在统一上下文中交换信息。

#### 3.2.1.4 全场景智能

仓颉 Agent DSL 具备全场景部署能力，无论是在资源受限的嵌入式设备，还是高性能的云端服务器，Agent DSL 均能提供一致的开发体验与运行表现。包括：

- **多平台部署能力**：Agent DSL 可被轻松部署于不同硬件架构和操作系统。
- **大模型统一抽象层**：通过构建统一的模型抽象接口，支撑将大语言模型适配到本地小模型和云端大模型的不同计算环境中。
- **异构工具集成能力**：Agent DSL 支持接入多种工具，包括鸿蒙系统 API、云侧 MCP 服务以及应用自身的功能模块，从而动态感知设备能力、网络状态与用户偏好，实现“一次编写，处处智能”。

未来，新开发 AI 应用场景下，在仓颉的智能化和跨平台能力支撑下，不仅能轻松实现全场景智能化应用，还能实现一码多端，大幅减少代码开发和维护的工作量。

## 3.3 未来一年语言演进策略

### 3.3.1 ArkTS 语言

ArkTS 作为鸿蒙应用开发的官方语言，会持续保持演进迭代。进一步丰富并发编程、完善类型系统、现代化语法等新特性，使开发者能够更快速地构建稳定且性能优越的应用。

**鸿蒙开放 API 会默认提供 ArkTS 的版本。**

#### 3.3.1.1 更强的并发能力

ArkTS 将提供全共享上下文，对象均默认分配在运行时的进程共享堆中，具备进程可见性，从而实现对象在并发场景下的高效访问。

此外，在现有 TaskPool 和 Worker API 的基础上，ArkTS 也计划引入更轻量的协程 API，该 API 同样支持结构化并发，进一步简化并发任务的开发，提升并发编程的表达能力与易用性。

ArkTS 还将支持绑定全共享上下文至非托管线程，开发者可以通过 attach 将 ArkTS 运行时环境挂载到自行创建的 Native 线程中，从而在该线程内安全地调用 ArkTS 系统 API。

#### 3.3.1.2 更高的开发效率

针对生态应用代码量快速增长引起的大型工程编译耗时长、内存占用高的挑战，ArkTS 编译工具链在当前基础上，将持续优化编译性能和构建峰值内存占用，完善调试调优能力，进一步提升开发效率。包括但不限于如下措施：

- TSC 编译缓存优化、架构模块化重构
- TreeShaking 能力
- ArkTS Linter 增量检查能力
- ArkTS 字节码生成器优化
- ArkTS 编译插件性能内存优化
- 字节码插桩能力

此外，从整体的 ArkTS 前端编译架构上，ArkTS 将引入前端编译器实现，去除 TS/JS 转换流程和对 TSC 工具的依赖，从而进一步提升 ArkTS 的编译和调试体验。

#### 3.3.1.3 更丰富的 SDK

ArkTS 的 SDK 将会带来更强大的功能和更灵活的集成体验。SDK 会提供更多的 Kit 包含：

- 数据增强套件
- 桌面扩展服务
- 图形加速服务
- 屏幕时间守护服务
- 企业数字空间服务

Kit 数量将达到 110 个，API 数量将超过 6W，满足各垂类应用开发者的业务开发诉求。

同时，新增的开放能力也允许开发者按需调用 AI 能力，深度自定义接口满足开发者个性化场景需求。API 的调用性能也会持续优化，且提供更详细的错误日志与调试工具，帮助开发者提升效率。

### 3.3.2 仓颉语言

仓颉作为鸿蒙应用开发的官方语言，将以提升开发者体验为目标，从语言特性构建、兼容现有生态、完善工具链易用性等方面持续建设语言能力。

鸿蒙 5.0 为首个支持仓颉应用开发的版本，鸿蒙 5.1 为首个内置仓颉运行时及 API 的版本，鸿蒙 6.0 为首个开发环境内置仓颉的版本。仓颉商用后支持鸿蒙 5.0 以上版本鸿蒙应用开发。因此使用鸿蒙 SDK 的仓颉应用开发者，应用的最低目标版本可设置为鸿蒙 5.0。

#### 3.3.2.1 API 发展策略

仓颉 API 是为了支撑鸿蒙应用使用仓颉开发而提供的一系列的 API，包括 API 接口定义，资料文档，样例代码。目前仓颉 API 是鸿蒙 SDK 开放能力的子集，并会随着仓颉商用逐步补齐。

仓颉 API 已在鸿蒙 5.0 首次以预览版方式面向应用定向开放，并持续在后续版本中持续丰富 API 能力，目标是为鸿蒙应用提供仓颉的全量鸿蒙系统能力。仓颉 API 将优先提供涉及 5 大类别，45+ Kit 的关键系统能力。

- 应用框架（APP Framework）：提供方舟 UI 框架、程序框架服务、方舟数据管理、方舟 Web、文件基础服务等 Kit 相关的开放能力。
- 系统（System）：提供安全（加解密算法框架服务、密钥管理服务等）、网络（短距通信服务、网络服务等）、调测调优（性能分析服务、应用测试服务）等 Kit 相关的开放能力。
- 媒体（Media）：提供相机、图片处理、音频服务、音视频播控等 Kit 相关的开放能力。
- 图形（Graphics）：提供方舟 2D 图形等相关的开放能力。
- 应用服务（APP Services）：提供广告、联系人、位置、用户通知等 Kit 相关的开放能力。

当前仓颉暂不支持的 API，用户可通过仓颉->ArkTS 和仓颉->C 跨语言互操作的方式调用 ArkTS API 或者 NDK API。

#### 3.3.2.2 开发工具

DevEco Studio 支持使用仓颉进行鸿蒙应用/元服务的开发(纯仓颉应用、仓颉+ArkTS 混合应用)，支持开发纯仓颉的静态库和动态库，在下载安装 DevEco Studio 后，开箱即用。在 DevEco Studio 上，仓颉的特性全貌将主要有以下方面：

- 代码编辑：代码高亮、代码补全、语法诊断、悬浮提示、定义跳转、引用查找、格式化等编码辅助能力，包括元编程相关的编码辅助能力。
- 编译构建：支持编译仓颉的 HAP/APP、支持编译仓颉的 HAR/HSP、支持推送仓颉 HAP 包至手机运行能力。
- 代码调试：支持仓颉 HAP 在手机的调试能力，包括断点能力、单步调试、调试信息（线程、堆栈、变量等）可视化查看能力。
- 场景化调优：支持仓颉场景化的性能调优，解决快速定界、效率提升、内存分析、内核分析和卡顿分析相关问题，帮助应用开发者定位到问题代码。
- 模拟器：支持手机、平板和鸿蒙 PC。
- 测试框架：支持仓颉鸿蒙工程推包至鸿蒙设备进行测试。

#### 3.3.2.3 资料文档

仓颉为开发者提供了 鸿蒙应用开发指南、API 参考、Samples 等各类资源，旨在帮助开发者更快速地掌握使用仓颉开发 鸿蒙应用的方法。

- 应用开发指南：入门部分提供快速入门、开发基础知识、仓颉语法介绍等知识，旨在帮助开发者掌握仓颉知识以及快速了解 鸿蒙应用开发基础；开发部分提供“应用框架（APP Framework）、系统（System）、媒体（Media）、图形（Graphics）、应用服务（APP Services）”五大类别，45+ Kit 的详细开发步骤。工具部分提供如何在 DevEco Studio 中进行开发、调试、发布、上架应用的操作指导。

- API 参考：面向开发者提供鸿蒙系统开放接口的全集，供开发者了解具体接口使用方法。API 参考详细地描述了每个接口的功能、使用限制、参数名、参数类型、参数含义、取值范围、权限、注意事项、错误码及返回值等。仓颉提供“应用框架（APP Framework）、系统（System）、媒体（Media）、图形（Graphics）、应用服务（APP Services）”五大类别，45+ Kit 对应 API ，以及仓颉标准库、仓颉与 ArkTS 混合开发相关 API 。

- Codelabs：以教学为目的的代码样例及详细的开发指导，帮助开发者一步步地完成指定场景的应用开发并掌握相关知识。Codelabs 支持互动式操作，通过文字、代码和效果联动为开发者带来更佳的学习体验。仓颉提供仓颉基础语法、仓颉在高并发、强安全等场景下的 Codelabs 内容，让开发者更快速地掌握使用仓颉开发高质量应用的方法。

- Samples：面向不同类型的开发者提供的使用仓颉开发鸿蒙生态应用的优秀实践，每个 Samples 都是一个可运行的工程，为开发者提供实例化的代码参考。

#### 3.3.2.4 跨平台

仓颉将跨移动 OS 平台作为语言关键能力，演进路标如下：

- 2025 年完成仓颉跨平台开发能力构建，主要包括：

  a) 仓颉代码在鸿蒙、iOS、Android 三个 OS 平台的编译和运行能力；

  b) 仓颉与 Java、Objective-C 语言互操作能力；

  c) 鸿蒙、iOS、Android 三平台的代码调试能力；

d)鸿蒙和 Android 平台的代码调优能力。

- 2025 年 Q4 提供首个 Beta 版本，并在部分应用场景联创落地。
- 2026 年 Q2 发布首个 Release 版本，并在后续版本中持续迭代完善和优化跨平台开发体验，包括：

a)仓颉与 Kotlin、Swift 语言互操作能力；

b)跨语言调试能力；

c)iOS 平台的调优能力；

d)跨平台相关的系统库和三方库。

# 致谢

本白皮书的诞生，离不开众多伙伴严谨细致的审阅与极具价值的建议。在此，我们向所有为白皮书撰写工作贡献宝贵力量的伙伴们，致以最崇高的敬意与最衷心的感谢！

特别鸣谢美团、美团众包、京东、京东金融、抖音、腾讯会议、高德地图、国土调查局、大麦、菜鸟、便单等合作伙伴（排名不分先后）。在白皮书的审阅过程中，你们以专业审慎的态度，逐字逐句推敲技术表述；以行业前沿的视角，精准指出内容逻辑的优化方向；以丰富的实践经验，提出切实可行的修改建议。每一处批注、每一条反馈，都饱含着对技术的严谨态度与对生态共建的热忱，让白皮书的内容更具权威性、专业性与实用性，也让鸿蒙编程语言的技术内涵与应用前景得以更清晰、准确地展现。

“独行者快，众行者远”，正是你们无私的付出与真诚的帮助，让这份白皮书不断完善、日臻成熟。未来，我们期待与更多伙伴继续携手，以鸿蒙编程语言为桥梁，共绘技术生态的宏伟蓝图，探索更多创新可能！