华为认证 HarmonyOS 系列教程

HCIP-HarmonyOS Device Developer 学员用书

版本：1.0

华为技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。

本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

华为技术有限公司

地址： 深圳市龙岗区坂田华为总部办公楼 邮编：518129

网址： https://e.huawei.com

华为认证体系介绍

华为认证是华为公司基于“平台+生态”战略，围绕“云-管-端”协同的新ICT技术架构，打造的覆盖ICT（Information and Communications Technology 信息通信技术）全技术领域的认证体系，包含ICT技术架构与应用认证、云服务与平台认证两类认证。

根据ICT从业者的学习和进阶需求，华为认证分为工程师级别、高级工程师级别和专家级别三个认证等级。

华为认证覆盖ICT全领域，符合ICT融合的技术趋势，致力于提供领先的人才培养体系和认证标准，培养数字化时代新型ICT人才，构建良性ICT人才生态。

HCIP-HarmonyOS Device Developer V1.0定位于培养基于设备开发场景具备专业知识和技能水平的高级工程师。

通过HCIP-HarmonyOS Device Developer V1.0认证，您将掌握系统基本概念及原理、技术架构、设备开发流程、内核与驱动知识，具备设备子系统开发、移植的能力，能够胜任设备开发高级工程师岗位。

华为认证协助您打开行业之窗，开启改变之门，屹立在HarmonyOS行业的潮头浪尖！


华为认证
HCIE
Huawei Certified ICT Expert
HCIP
Huawei Certified ICT Professional
HCIA
Huawei Certified ICT Associate
Cloud Service & Platform
云服务与平台
ICT Infrastructure
ICT技术架构与应用
HarmonyOS
IoT
GaussDB
openGauss
MDC
Big Data
AI
openEuler
Kunpeng Application Developer
Cloud
Storage
Intelligent Computing
Enterprise Communication
Intelligent Vision
Data Center
Datacom
WLAN
Security
SDN
Transmission
Access
LTE
5G

前言

简介

本书为 HCIP-HarmonyOS Device Developer 认证培训学员用书，适用于准备参加 HCIP-HarmonyOS Device Developer 考试的学员或者希望了解设备开发知识的读者。

本书适用于 OpenHarmony 3.0 LTS 的设备开发。

内容描述

本实验指导书共包含 8 个章节，分别为系统及应用场景介绍、编译构建与启动恢复、LiteOS-A 内核、HDF 驱动开发、应用安装部署、工程调测、系统移植和子系统能力介绍。

读者知识背景

本课程为华为认证基础课程，为了更好地掌握本书内容，阅读本书的读者应首先具备以下基本条件：

- 具有基本的代码编程能力，熟悉 C 语言，了解基本的设备开发知识。

实验环境说明

设备介绍

为了满足 HCIP-HarmonyOS Device Developer 实验需要，建议每套实验环境采用以下配置：

设备名称、型号与版本的对应关系如下：

表1-1 实验环境配置表

设备名称	设备型号	软件版本
PC机	Windows系统	Windows 10 64位
BearPi-HM Micro开发板	BearPi-HM Micro	/

准备实验环境

检查设备

实验开始之前请每组学员检查自己的实验设备是否齐全，实验清单如下。

表1-2 实验清单表

设备名称	数量	备注
笔记本或台式机	每组1台	台式机要有无线网卡
BearPi-HM Micro开发套件	1	需要有USB线缆(Type-C)连接线

目录

1 系统及应用场景介绍

1.1 HarmonyOS 系统介绍

1.1.1 HarmonyOS 定义

HarmonyOS 是华为新一代的智能终端操作系统，为不同设备的智能化、互联与协同提供了统一的语言，为消费者带来简捷、流畅、连续、安全可靠的全场景全新交互体验。

1.1.2 HarmonyOS 特征

该操作系统有三大特征：“硬件互助，资源共享”、“一次开发，多端部署”、“统一 OS，弹性部署”。

1.1.2.1 统一 OS，弹性部署

HarmonyOS 通过组件化和组件弹性化等设计方法，做到硬件资源的可大可小，在多种终端设备间，按需弹性部署，全面覆盖了 ARM、RISC-V、x86 等各种 CPU，从百 KiB 到 GiB 级别的 RAM。

汇聚各形态终端设备能力，通过软总线连接构建设备之间的数据交互通路，打通 1+8+N 智慧生活的全场景。

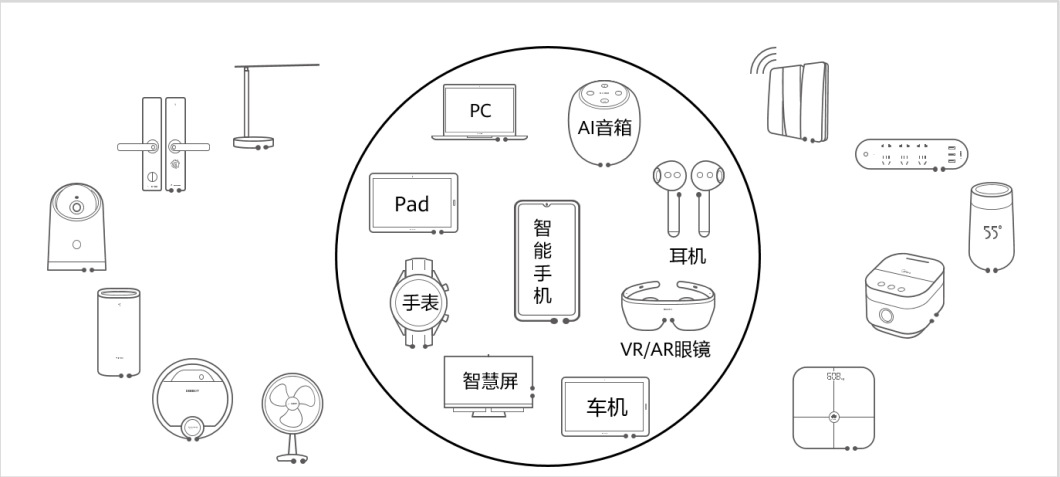

图1-1 1+8+N 智慧生活全场景生态图

HarmonyOS 分布式特性在多行业场景中得以体现，如智能家居、运动健康、智慧出行、智慧矿山等。针对多场景下的多形态智能硬件，基于 OpenHarmony 3.1、HarmonyOS Connect

服务包、HarmonyOS SDK API，可开发出切合场景使用的案例，例如手机与智慧屏、智能音箱之间的场景联动，可实现音频与视频的跨设备流传。

1.1.2.2 硬件互助，资源共享

分布式软总线是多设备终端的统一基座，为设备间的无缝互联提供了统一的分布式通信能力，能够快速发现并连接设备，高效地传输任务和数据。

分布式数据管理位于基于分布式软总线之上的能力，实现了应用程序数据和用户数据的分布式管理。用户数据不再与单一物理设备绑定，业务逻辑与数据存储分离，应用跨设备运行时数据无缝衔接，为打造一致、流畅的用户体验创造了基础条件。

分布式任务调度基于分布式软总线、分布式数据管理、分布式 Profile 等技术特性，构建统一的分布式服务管理（发现、同步、注册、调用）机制，支持对跨设备的应用进行远程启动、远程调用、绑定/解绑、以及迁移等操作，能够根据不同设备的能力、位置、业务运行状态、资源使用情况并结合用户的习惯和意图，选择最合适的设备运行分布式任务。

分布式设备虚拟化平台可以实现不同设备的资源融合、设备管理、数据处理，将周边设备作为手机能力的延伸，共同形成一个超级虚拟终端。

1.1.2.3 一次开发，多端部署

HarmonyOS 提供用户程序框架、Ability 框架以及 UI 框架，能够保证开发的应用在多终端运行时保证一致性。一次开发、多端部署。多终端软件平台 API 具备一致性，确保用户程序的运行兼容性。支持在开发过程中预览终端的能力适配情况（CPU/内存/外设/软件资源等）。支持根据用户程序与软件平台的兼容性来调度用户呈现。

1.2 HarmonyOS Connect 解决方案

1.2.1 HarmonyOS Connect 介绍

HarmonyOS Connect（中文名称：鸿蒙智联）是华为消费者业务面向生态智能硬件的全新技术品牌。HarmonyOS Connect（鸿蒙智联） 认证产品能够成为“超级终端”的一部分，给消费者带来全场景智慧生活新体验。

依托于华为 HarmonyOS、云服务、硬件、芯片等软硬件开放能力。HarmonyOS Connect 来自 Works With HUAWEI HiLink 品牌和 Powered by HarmonyOS 品牌的融合升级，旨在提供统一的生态产品开发、销售与运营服务。

在各项服务功能上，HarmonyOS Connect 承接了 Works With HUAWEI HiLink 的碰一碰、快速连接的功能体验，而在连接技术上，HarmonyOS Connect 使用了 Powered by HarmonyOS 的分布式软总线连接能力。

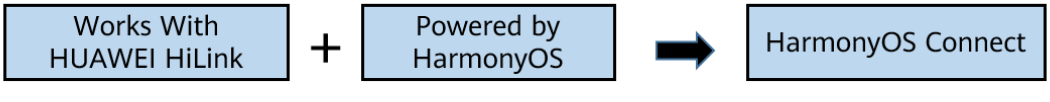

图1-2 HarmonyOS Connect 概念示意图

HiLink 主要针对的是应用开发者与第三方设备开发者，使其共同实现一个物联网的业务场景，其解决方式的核心都在连接 HiLink SDK 当中。HarmonyOS Connect 通过分布式软总线的方式连接所有设备，强能力设备可对弱能力设备进行设备虚拟化，将弱设备当做本机设备直接调用。

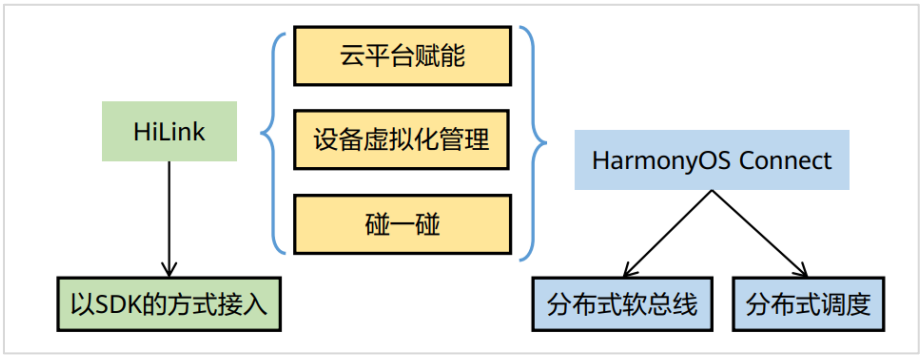

图1-3 HiLink 与 HarmonyOS Connect 异同点对比示意图

HarmonyOS Connect 是华为面向消费领域的智能硬件开放生态，为设备商、解决方案提供商、模组商等广大智能硬件合作伙伴提供全方位赋能，使合作伙伴设备快速融入华为 1+8+N 的全场景智慧生活，实现商业共赢。

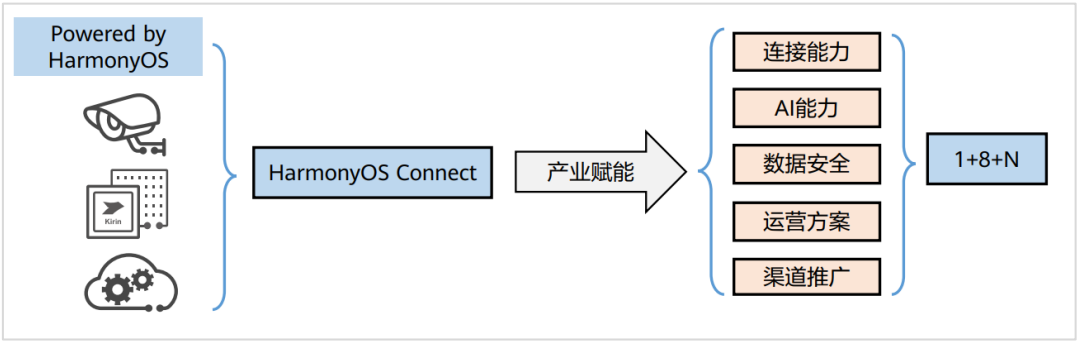

图1-4 HarmonyOS Connect 产业赋能示意图

HarmonyOS Connect 生态解决方案合作伙伴可通过华为智能硬件合作伙伴平台，提交模组认证申请，认证通过的模组可上架到平台的方案中心，提供给 HarmonyOS Connect 生态产品的合作伙伴使用。

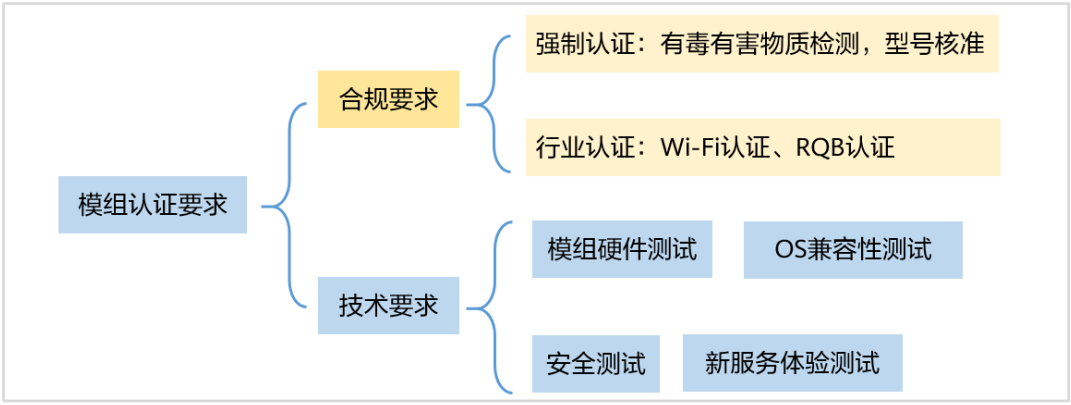

图1-5 HarmonyOS Connect 生态模组认证要求示意图

HarmonyOS Connect 智能硬件具备 4 项主要特点：

不同设备，同一语言。不同设备的智能化、互联与协同具备统一的语言。

硬件互助，形成超级终端。多种分布式技术助力多设备之间硬件互助、资源共享，实现跨终端无缝协同体验。

服务跨端流转，脱离单设备依赖。应用一次开发、多端部署，服务可以在手机、平板、智能穿戴等不同设备上跨端流转，无缝接续。

微内核架构，重塑终端设备可信安全。内核可形式化验证，实现“正确的人，通过正确的设备，正确地使用数据”。

1.2.2 HarmonyOS Connect 场景解决方案

1.2.2.1 智能家居

HarmonyOS Connect 的智能家居行业使能，涵盖智能家电、中控交互中心、安防产品和个护产品等端到端领域，具备碰一碰获取服务、多模态交互等分布式交互入口能力，集成智慧生活等云服务及后向运营模式，全方位提升家居体验。近十年以来，智能家居设备随着物联网技术的成熟，越来越多的出现在了消费者的家中。在这一过程中，边缘计算、云服务、云平台等各种技术也愈发成熟，智能家居设备也从单设备智能走向了全屋智能。

1.2.2.1.1 传统智能家居方案介绍与痛点

现如今，传统家电产品面临着产品智能化的难题。例如一套智能家居系统，如果无法和手机、平板电脑、电视和智能手表等交互终端设备的互联互通，设备联网率普遍偏低，难以为用户提供真正的智慧生活交互体验。产品之间互通性不足、协议标准不统一、用户交互体验差等等，这些挑战亟需一个能统一各类产品和服务的系统性解决方案。

智能家居设备大幅度增加，容易造成消费者无法有效管理全部设备，设备使用意愿低。

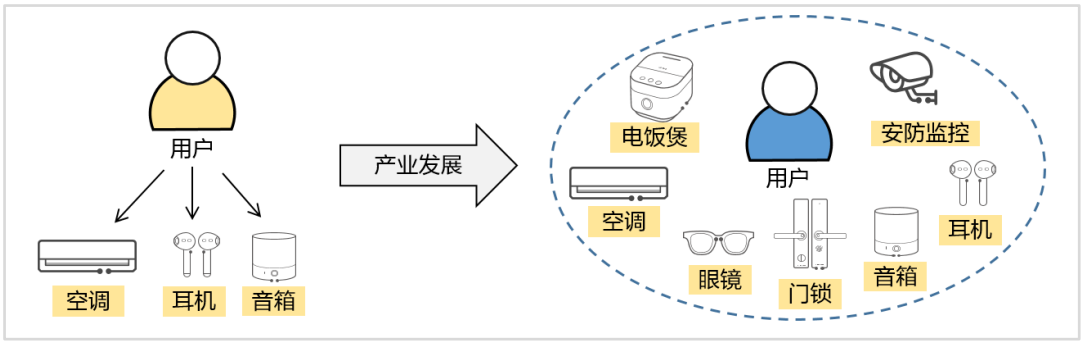

图1-6 常用智能家居变迁图

以一款蓝牙连接的智能牙刷为案例，来说明当前的智能家居设备的体验。

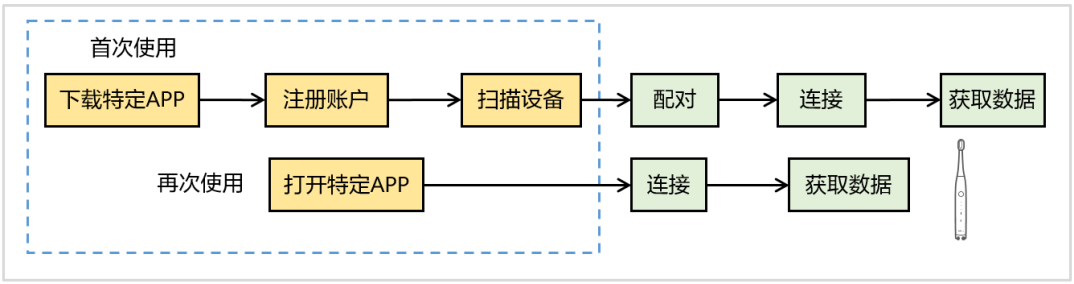

图1-7 当前智能家居体验示意图

一款传统的智能家居设备在到消费者手中之后，往往会伴随着一份说明书，说明书会指导消费者下载特定的 APP，再使用此 APP 连接设备，最终才能成功的使用此智能设备。

在这一系列的操作中，以下的几个动作往往是不可缺少的。

- 下载特定的 APP

扫描特定的二维码或者采用其他途径，下载特定的 APP，APP 的体积往往在 20MB~150MB 之间。虽然消费者只是想使用智能牙刷，但设备厂商往往会在 APP 中配置一些相关的功能，一些业务较多的设备厂商，甚至会让一个 APP 兼容品牌下的全部设备。APP 当中存在大量冗余的功能。

- 注册账户

如果消费者没有注册过账户，往往会要求消费者注册一个账户。

- 扫描并连接设备

智能牙刷开机，使得设备处于可被查找连接的状态下，接下来在手机 APP 中查找智能设备。查找连接设备的过程往往比较长，这是由于设备在连接之后，还会进行设备安全可信的校验，第一次查找设备并完成连接的耗时往往在 5s~15s。

如果设备采用的是 Wi-Fi 的方式，连接过程则稍有不同。设备会先开启 AP 模式，手机连接设备开启的热点，与设备连接成功之后对设备进行配网，之后再让设备接入家庭中的局域网，手机通过局域网连接设备，这一系列的操作往往需要 30s~1min。

如何减少扫描连接设备的用户等待时间一直是各物联网设备厂商非常关注的问题，但由于技术方案本身的限制，即使是每一步的连接耗时都做到了最短，等待时间仍旧比较长，复杂的操作步骤和较长的等待时间“劝退”了大量的消费者。目前市场上的传统方案智能家居设备，只有少

部分的设备配网连接成功了，大量的消费者购买了智能家居设备却没有使用设备的连接能力，这一现状让消费者与智能硬件厂商都非常的头疼。

除了智能设备第一次连接操作繁琐之外，传统的方案是基于 APP 实现的。当用户再次使用智能硬件的时候，需在手机中翻找到特定的 APP。现在手机中 APP 往往数量众多，翻找一个特定的 APP 是较为繁琐，用户翻找特定 APP 的意愿低，进一步导致设备的智能化能力难以得到发挥。

1.2.2.1.2 HarmonyOS Connect 智能家居应用场景

智能家居场景下的这些业务痛点，被 HarmonyOS Connect 很好解决了，HarmonyOS Connect 的到来对智能家居设备的使用是一次颠覆。再次审视传统的连接方案，这些看起来“必不可少”的操作步骤其实并不是一定需要的。

为什么一定要下载特定的 APP？为什么一定要注册新账户？有没有更快的设备查找方式？重复使用设备的时候，有没有更好的触发方式？这些问题的答案就在 HarmonyOS Connect 当中。

图1-8 HarmonyOS Connect 智能家居解决方案图

HarmonyOS Connect 智能家居具备碰一碰获取服务与多模态交互能力，集成智慧生活等服务，可全方位提升家居体验。通过使用应用原子化能力，使得用户可以快速与设备连接使用，非常便捷地发挥设备智能化能力。

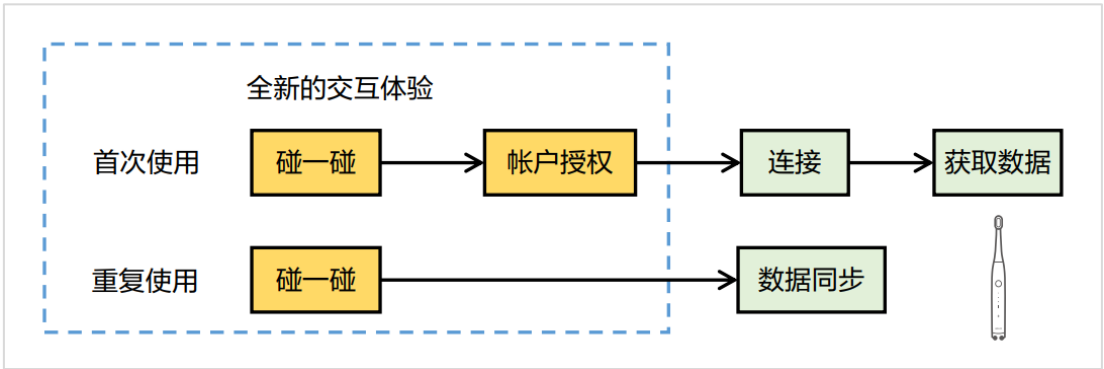

图1-9 “碰一碰”新体验示意图

“碰一碰”功能使用流程如上图所示。用户使用已经登陆了华为账号的 HarmonyOS 手机，将手机与智能设备的 NFC 标签一碰，可以快速实现账户授权，对智能设备的安全进行检验，之后快速实现手机与设备的连接并拉起原子化服务，用户此时即可使用智能硬件的相关服务。用户的操作只有一步，那就是手机与设备的一碰。同样的，在反复使用设备功能的时候，也可以直接使用手机与智能设备一碰，原子化服务被拉起，即可对硬件进行操作。

以智能牙刷为案例，搭载 HarmonyOS Connect 的智能家居具备以下三个特点：

碰一碰快速配网：通过搭载 HarmonyOS 的手机“碰一碰”产品机身上的 NFC 标签，即可拉起万能卡片对设备进行配网和操控；告别繁琐的家电设备配网操作，几秒之内即可完成授权和配网。

万能卡片快捷入口：用户无需从打开专用 APP，只需使用手机与牙刷 “碰一碰”后，启用万能卡片，借助菜单向导，便可快完成清洁、强度、时长。为用户每一次的刷牙行为进行质量分析，并给出牙齿健康的建议，主动提醒用户更换牙刷。

硬件互助突破壁垒：HarmonyOS Connect 为产品提供了统一的操作平台，使智能家电产品不再是“孤岛”，多个设备之间通过分布式能力联结成“超级终端”，为用户提供了智能化完整而流畅的全场景解决方案。

全新的交互方式带来更好的操控体验，和传统产品相比，无需下载专用 App ，摆脱下载专用 App 的苦恼，手机“碰一碰”，便能通过万能卡片快速操控设备，用户体验更佳。HarmonyOS Connect 智能硬件告别繁琐的配网步骤，只要手机与智能家电设备“碰一碰”即可通过万能卡片完成轻松配网，简单易操作，设备配网率成功率显著提升。在 HarmonyOS Connect 助力下，消费者获得的不仅是智能硬件设备，而是场景化的、完整的服务体验，实现智能家居产业的全新升级。

1.2.2.2 运动健康

HarmonyOS Connect 的运动健康行业使能，涵盖智能健身、健康监测、健康保健三大健康场景。利用分布式能力，实现超级运动终端、智能运动私教、个人健康顾问。开放个人画像能力，实现跨端全场景联动，助力伙伴跨代创新，打造运动健康产品最佳体验。

在智能健身场景下，搭载 HarmonyOS Connect 的运动健康生态产品提供完整的基础能力解决方案与丰富的增值服务能力，包括畅连通话、小艺助手、视频投屏、安全能力等，还提供专业运动私教等行业特性能力。客户可勾选、可定制，一站式快速构建室内健身全新产品、全新体验。

在健康监测场景下，搭载 HarmonyOS Connect 的运动健康生态产品提供完整的基础能力解决方案，碰一碰配网、碰一碰获取服务；适配行业主流芯片，一次开发即可接入智慧生活、小艺助手、场景联动；提供健康顾问等专业能力，打通跨设备体验。帮助开发者快速创新，快速将产品推向市场。

在健康保健场景下，搭载 HarmonyOS Connect 的运动健康生态产品提供碰一碰配网、碰一碰获取服务；适配行业主流芯片，一次开发即可接入智慧生活、小艺助手、安全能力；通过健康顾问、精品服务、关联产品等模块，打通健康保健场景。帮助开发者快速创新，打造极为专业、智能化的体验。

传统健身设备面临着产品智能化的难题。例如健腹滑板，如果无法和手机、平板电脑、电视和智能手表等交互终端设备的互联互通，设备联网率几乎为 0。健身次数、消耗卡路里没有记录，用户购买后不清楚如何使用，难以为用户提供真正的智慧生活交互体验。大多数用户购买设备后，使用率低，无法达到健身塑型的初心。

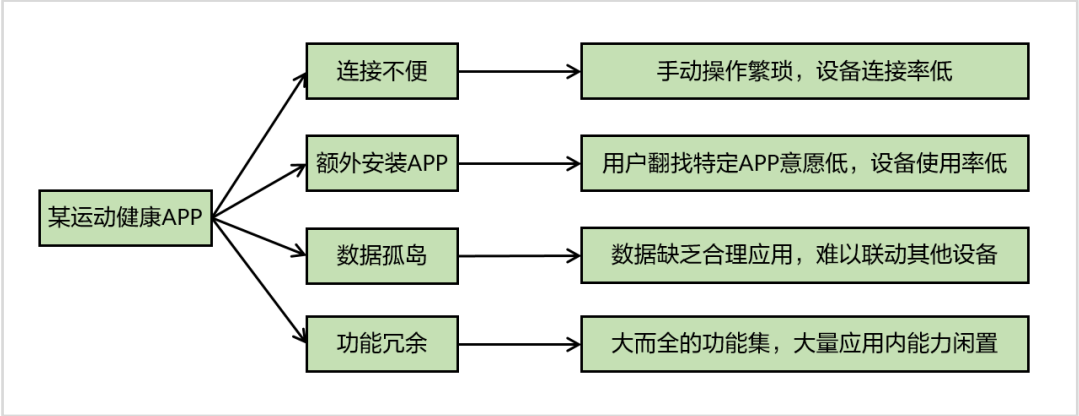

图1-10 某运动健康 APP 体验示意图

但是在 HarmonyOS Connect 的场景下，用户只需将 HarmonyOS 手机与健身设备上的“碰一碰”标签一碰，即可拉起万能卡片对设备进行操控，几秒之内即可完成授权和连接。除了“碰一碰”的能力，HarmonyOS 的手机还具有万能卡片的能力。“碰一碰”弹出的万能卡片可提供特色服务功能，实时记录用户锻炼数据，包括运动时间、动作次数、消耗卡路里等。运动中支持语音播报，用户可随时掌握运动进程，运动中可对用户进行语音激励，通过大量的视频指导课程，指导用户科学的健身塑形，让运动更科学。

HarmonyOS Connect 全新的交互方式，更便捷的设备控制和互动课程，操控体验好。和传统产品相比，无屏健身器材，也能有数据记录和数据分析，且无需下载 APP ，仅拿出手机“碰一碰”，便能通过万能卡片，快速操控健身器械。

设备连接率的提升让传统的健身器材焕发生机活力，进入万物互联的生态圈。“碰一碰”降低配网难度，提升设备联网率，万能卡片方便易用，可随时查看健身数据和指导课程，指导用户科学运动，让设备更有粘性。

在 HarmonyOS 与 HarmonyOS Connect 的加持下，让不同的设备以相同的“语言”组成万物互联的超级终端，打破了单一物理设备硬件能力的局限，实现更多设备和服务的互联互通，将为智能健身产品迎来新的发展机遇。

1.2.2.3 智慧出行

HarmonyOS Connect 的智慧出行行业使能，涵盖车载带屏类、车载 IoT 类、个人出行类三大类别。车载带屏幕设备包括车载智慧屏与流媒体后视镜，手机与设备硬件互助，手机应用和服务延展到设备，为设备带来更丰富的服务体验。车载 IoT 类包括行车记录仪、车载香氛、车载摄像头等设备，通过集成 HarmonyOS Connect 方案，分布式特性赋予车载 IoT 设备快捷连接、便捷控制能力，为驾车出行提供安全舒适体验。个人出行类包括智能滑板车、智能平衡车、智能两轮电动车、智能头盔等设备，通过集成 HarmonyOS Connect 方案，设备可实现快捷连接、便捷控制，通过分布式能力实现设备联动、服务流转，为个人出行提供良好体验。

1.2.2.3.1 智能滑板车

以滑板车为例，倘若用户使用无智能化的传统滑板车时，只需要使用钥匙就能将车子开走，而在常规的智慧出行场景下，则需完成一系列操作才能获取相关的智能服务。智能化的服务不是更加繁琐的操作，而应该更多的体现方便快捷的设计理念。

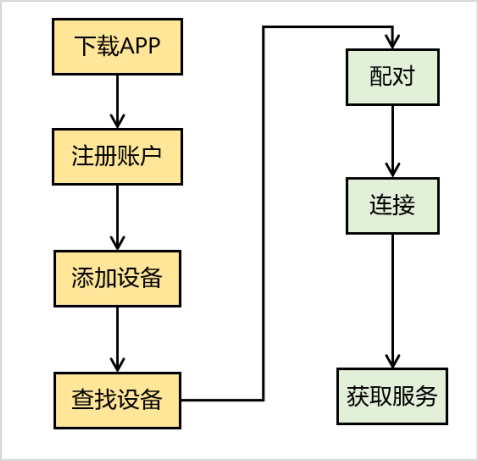

图1-11 某滑板车 APP 体验示意图

在 HarmonyOS Connect 的加持下，智能滑板车的使用将更加便捷。用户可通过智慧生活 APP 添加设备，下次开机自动连接，自动记录骑行数据和使用习惯，使用搭载了 HarmonyOS 的手机与滑板车靠近“碰一碰”，拉起万能卡片，一键开锁/关锁，查看电量和行驶里程等数据。搭载了 HarmonyOS Connect 能力的智能滑板车，既实现了设备的智能化体验，又进一步的简化了用户的操作，一触即发的功能设计理念得以体现。

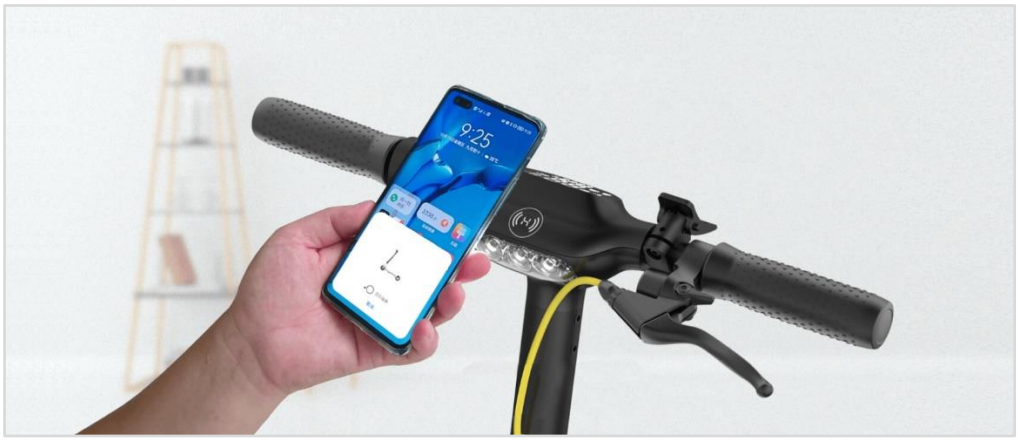

图1-12 “碰一碰”智能滑板车体验示意图

分布式特性也有助于在智能出行设备上开发出更多的业务场景与能力。用户可根据不同需求，在万能卡片中调整滑板车速度档位，定速巡航，骑行更安全。用户也可在万能卡片中自由选择及调节滑板车脚踩板氛围灯，拥有多种绚丽灯效，酷炫出行。在骑行前，用户也可通过智慧生活 APP 对车辆进行故障检测，查看车辆安全状况，检测可精准定位到具体的故障位置及部件，用户可根据检测结果及时排除故障，用车更安全。

1.2.2.3.2 智能两轮电动车

除了智能脚踏车，HarmonyOS Connect 在智能两轮电动车上也有应用。两轮电动车已有多年使用历史，消费者的使用习惯基本固化，用户已养成的习惯与复杂的联网操作导致智能电动车被购买后，往往只发挥了车辆本身的功能，智能化能力难以体现，产品联网率偏低，使用体验与传统两轮电动车没有显著创新改变，产品体验差异化不明显，APP 应用用户黏度低。

HarmonyOS Connect 的极简连接能力，使得用户可体会到“手机钥匙”的能力，只需将手机与电动车上的 NFC 标签一碰，便可以快速拉起万能卡片，完成电动车配网并获取服务。

两轮电动车整体趋势逐步向智能化、便捷化、人性化方向发展。在搭载了 HarmonyOS Connect 的两轮电动车上，用户体验提升显著。用户通过手机上的万能卡片功能即可便捷的操作电动车，极大的提升两轮电动车产品的科技感与消费者的认可度。“碰一碰”快速完成电动车配网操作，设备联网体验操作优化，搭载了 HarmonyOS Connect 的两轮电动车设备联网率提升明显。

1.2.2.3.3 智能座舱

在智能汽车领域，搭载 HarmonyOS 车机与 HarmonyOS Connect 硬件的智能座舱也已有落地。智能座舱带来的体验与传统的驾驶座舱相比，可大幅度减少复杂操作，服务能力实现直达，帮助用户获得更加舒适便捷的驾乘体验。

图1-13 HarmonyOS 车载桌面示意图

HarmonyOS 车载桌面，带来耳目一新的交互体验。卡片式 Smart Dock 自由收纳服务信息，一点即可触达高频功能，减少复杂操作，更加专注驾驶。强大的语音交互能力，屏幕上可见即可说，支持连续对话与随时插话。多音区智慧声源感知，可关照车内每个方位的需求。HarmonyOS 车机的语言助手可自动识别多种场景，在通勤时智能推荐歌单，在电量告急时推送附近充电站的导航信息，在回家途中提醒用户领取快递。智能座舱实现的是一趟出行，一站式的贴心关怀。

除了强大的车机能力与车内智能设备的能力之外，该智能座舱最具特色的是手机、手表与车机的联动使用。手表、手机靠近车身，可无感解锁车门，实现轻装出行；车机与手表联动，抬腕即可实现手表远程控车，完成开关车窗、开启空调、查看充电进度等操作。智能座舱还具备车家互联的能力，通过车机控制中心，获取家中的智能设备状态，在用户即将到家时，让家中的灯和空调等家居设备提前就位，智能座舱实现了“人、车、家”的全场景互联。

1.2.3 HarmonyOS Connect 产品解决方案

1.2.3.1 HarmonyOS Connect 服务包

HarmonyOS Connect 服务包为合作伙伴提供设备开发、原子化服务开发、设备全生命周期运营运维等一站式智能化解决方案，由基础服务、增强服务、HarmonyOS Connect 云等组成。

基础服务为设备赋予超级终端体验，增强服务提供设备运维分析、运营变现能力以及专属场景服务，HarmonyOS Connect 云连接设备与用户，共同助力开发者快速实现产品智能化升级，打造互联互通的 HarmonyOS Connect 生态。

图1-14 HarmonyOS Connect 服务包概念图

HarmonyOS Connect 服务包的基础服务主要有四方面能力。

极简连接。用户使用手机靠近或碰一碰设备，即可自动弹窗、一键连接，进度全程可视，两步极速配网。支持秒控配网 NAN、常规配网 SoftAP、蓝牙辅助配网等多种配网方式。同一华为帐号或家庭群组成员可实现多设备共享，并支持一键分享。跨设备互连更加方便快捷，轻松唤醒专属智能设备。

万能卡片作为设备与服务的载体，支持多设备运行，服务免安装，一步直达。开放服务中心、控制中心、智慧生活等多种入口。万能卡片不仅提供设备操控功能，多种服务也让单设备更智能。

极简交互实现全场景接入，支持碰一碰、靠近发现、扫一扫等多种用户场景化交互方式拉起服务卡片，以及通过服务中心、控制中心、智慧生活和负一屏中的设备卡片，完成设备状态查询、设备控制和更多服务。

硬件互动建立多设备间视频、音频、信息服务流转，组合不同设备的软硬件能力，融合成“超级终端”，为用户打造全场景智慧生活体验。

在以上四方面的基础服务之上，HarmonyOS Connect 还扩展了一系列的增强服务，具体可分为运维服务、运营服务与专属场景服务这三类。运维服务提供 OTA、补丁、运维分析等运维服务，为合作伙伴提供设备软件生命周期内可追溯、可快速修复、可升级的运维平台。运营服务提供设备统计、用户统计、客服等运营服务，为合作伙伴提供运营变现能力。专属场景服务提供多设备、场景化服务，覆盖用户主流场景，让设备更好用，帮助伙伴留住用户。

搭载了 HarmonyOS Connect 服务包的应用软件数量种类繁多，覆盖多使用场景，包括语音助手、音视频通话、服务分发、运动健康、智慧生活等场景。具体的应用案例可列举如下：小艺，语言助手让设备更智能；畅连，语音、视频、屏幕共享，多种方式让沟通无缝；服务中心可用于原子化服务的智能分发；运动健康可用作全天候运动健康管家；智慧生活实现多场景联动，设备管理更加便捷。

图1-15 HarmonyOS Connect 服务包增强服务应用案例

1.2.3.2 智慧场景联动

智慧场景联动是华为面向消费领域的智能软硬件开放生态平台，通过聚合设备能力和应用服务能力，赋能开发者，进行一站式场景开发和运营，提升消费者智慧全场景的生活体验。

智慧场景联动平台具备创新的引擎与丰富的编排能力吗，采用 ECA 联动模型引擎（Event、Condition 和 Action），实现更多的智能组合。ECA 联动模型引擎具备这几个特点：自然语言降低门槛，极简交互轻松上手，让每个能力可视可控可动；事件状态动作任意组合，真正实现按需设定千人千面；端侧、云侧引擎一体化联动，打破软件和硬件互动边界。

基于智慧场景联动平台，开发者开发并发布一款 APP 可遵从这几个操作步骤：APP 创建，在手机端即可完成场景编排；PC 端同步，可自动完成模板泛化并支持素材编辑；之后进行 APP 验证，下发手机端进行测试验证；最后进行 APP 的快速发布，一般情况下 5 个工作日内完成发布上限。

图1-16 智慧场景联动开发步骤

基于智慧场景联动，可实现多种智慧场景。例如手机与环闪壳之间的联动，可实现在进行手机自拍时，轻松连接调起环闪壳的 LED，实现对自拍场景下的补光，达到更符合用户需求的功能体验。

图1-17 手机环闪壳示意图

除了手机与手机环闪壳之间的联动之外，还有更多的场景也可以基于智慧场景联动的方式进行开发。例如在温馨回家的场景下，当用户回到家中，此时家中的多种智能设备可自动的做出反馈，例如室内灯光调节、音乐自动播放、天气预报自动播报、空调自适应调节温度等，以此来实现智能生活温馨回家的体验。

1.2.3.3 小艺智慧助手

小艺智慧助手是用户与设备的主要交互入口，支持基于用户、设备、服务建模的主动服务等智能化体验。支持多品类设备操控和丰富的语音服务，提供包括唤醒、前处理、语音识别、语义理解和服务闭环的端到端语音解决方案。可提供标准集成方案，帮助设备开发者更快速地集成小艺智慧助手。

小艺智慧助手平台的语音功能支持语音唤醒、多轮对话、协同唤醒，提供全面语音服务。它的典型特性是自然的贴合用户习惯，可持续聆听，实现家庭、出行、办公场景下的多端联动，实现跨设备服务接续，承载海量生态设备接入互动，支持应答语情感差异化，主动服务推荐。

小艺智慧助手主要有六大使用场景：出行、家居、健康、教育、办公、娱乐。小艺智慧助手平台提供了端到端的语音服务功能。此外，小艺智慧助手平台支持自定义技能开放，支持技能应用场景设计，技能与用户之间的对话设计，提供测试、发布及管理环境。

小艺智慧助手目前已有大量的应用，典型的应用案例包括家居智能中控、随身看&智慧盒子、学习台灯。家居智能中控基于典型家居场景，可覆盖用户 90%以上语音控制功能，支持家居控制、时间查询、闹钟设置、天气查询等语音控制功能。随身看&智慧盒子面向老人、儿童、中青年全年龄段用户，提供影音娱乐、会议会务、游戏社交等场景。学习台灯支持语音控制，可通过唤醒语音助手控制台灯开灯、关灯，支持语音问答、语音指读。

图1-18 小艺智慧助手场景案例图

1.3 OpenHarmony 生态组成

1.3.1 OpenHarmony 生态介绍

OpenHarmony 是由开放原子开源基金会孵化及运营的开源项目，也是 HarmonyOS 和 HarmonyOS Connect 共同的技术底座。囊括了 HarmonyOS 的核心能力：分布式软总线、分布式数据管理、分布式任务管理、分布式设备虚拟化等各项能力。

OpenHarmony 采用了组件化的设计方案，可以根据设备的资源能力和业务特征进行灵活裁剪，满足不同形态的终端设备对于操作系统的要求。OpenHarmony 将广泛应用在智能家居物联网终端、智能穿戴、智慧大屏、汽车智能座舱、音箱等智能终端，提供全场景跨设备的极致体验。需要注意的是 OpenHarmony 与华为商用的 HarmonyOS 并不完全相同，华为的商用 HarmonyOS 核心是 OpenHarmony，在其上有独特的 UI 方案。而开源开放的 OpenHarmony 拥有 HarmonyOS 核心的能力，包括分布式软总线及各项分布式能力，是一个完整的可运行操作系统。

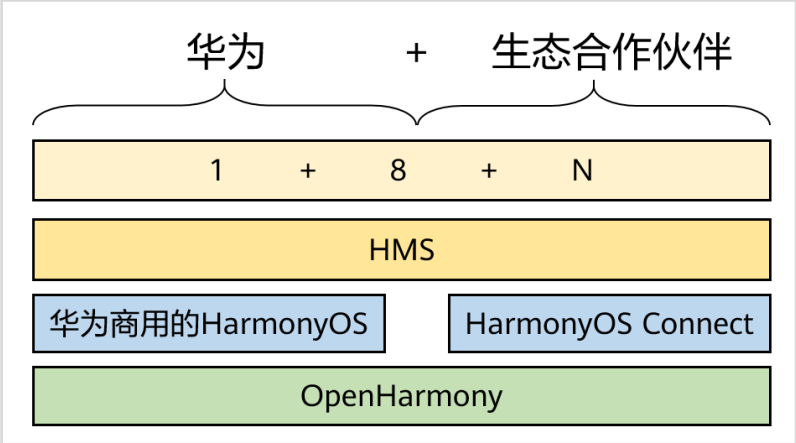

图1-19 OpenHarmony 生态示意图

在 OpenHarmony 的生态当中，OpenHarmony 是整个生态的基底，基于开源开放的 OpenHarmony，华为开发了商用的 HarmonyOS 操作系统用于手机、平板、手表、车机等硬件设备，此外华为还提供了 HarmonyOS Connect 解决方案用于第三方厂家的物联网硬件设备开发，多形态的物联网硬件在集成 HarmonyOS Connect 解决方案后将会具备一定的分布式能力，目前已有大量的硬件合作厂商基于 HarmonyOS Connect 解决方案推出了具备分布式特性的智能硬件，例如智能滑板车、智能打印机、智能血压计等多种可接入的智能硬件设备。

1.3.2 OpenHarmony 的典型开发板及芯片支持

OpenHarmony 支持如下几种系统类型：

轻量系统面向 MCU 类处理器的设备，此类设备硬件资源极其有限，支持最小内存 128KiB，OpenHarmony 可提供多种轻量级网络协议，轻量级的图形框架，以及丰富的 IOT 总线读写部件等。这一类设备的产品如智能家居领域的连接类模组、传感器设备、穿戴类设备等。典型

芯片包括 Hi3861 等，基于 Hi3861 芯片的开发套件可用作智慧路灯、智慧物流、人体红外等连接类设备应用场景。

小型系统面向应用处理器设备，支持最小内存 1MiB，OpenHarmony 可提供更高的安全能力、标准的图形框架、视频编解码的多媒体能力等。可支撑的产品如智能家居领域的 IP Camera、电子猫眼、路由器以及智慧出行域的行车记录仪等。典型芯片包括 Hi3518EV300 等，Hi3518EV300 开发套件可用作智慧视觉处理。

标准系统面向应用处理器设备，支持最小内存 128MiB，OpenHarmony 可以提供增强的交互能力、3D GPU 以及硬件合成能力、更多控件以及动效更丰富的图形能力、完整的应用框架。可支撑的产品如高端的冰箱显示屏。典型芯片包括 Hi3516DV300 等，Hi3516DV300 开发套件可用做带屏设备的开发场景，比如带屏冰箱、车机等。

2 编译构建与启动恢复

2.1 BearPi-HM_Micro 折叠开发板简介

小熊派折叠开发板图片可点击如下链接：

https://gitee.com/bearpi/bearpi-hm_micro_small#/bearpi/bearpi-hm_micro_small/

BearPi-HM_Micro 折叠开发板搭载了 OpenHarmony3.0 操作系统，支持 JS 应用开发部署。板载 STM32MP157 处理器，具备标准的 E53 接口及 S24 接口，通过接入丰富的 E53 案例扩展板，比如智慧路灯、人体红外、智慧物流、智慧井盖等，可轻松实现多种应用的开发和部署，且有防错口设计。标准的 S24 接口可外接高清摄像头扩展板，满足图像处理的案例需求。

BearPi-HM_Micro 折叠开发板的主控芯片 STM32MP157 使用意法半导体高性能的异构三核处理器（混合双 Cortex-A7 核和 Cortex-M4 核），具有计算和图形处理能力，兼备高能效实时控制和高功能集成度，有助于简化工业制造、消费电子、智能家居、医疗应用高性能解决方案的开发。

2.4G 的 SDIO WI-FI 实现稳定联网，拥有的 4 段式耳机、MIC 咪头、扬声器满足开发板音影交互需求。4.3 寸 800*480 分辨率 LCD 可触摸屏使得开发板和屏幕一体化。开发板支持一线开发（开发板供电、程序烧录、调试集合在一个 Type-C 口上）。

2.2 编译构建

受限于开发板的架构，硬件资源，开发板的系统是通过交叉编译生成的。交叉编译通俗地讲就是在一种平台上编译出能运行在体系结构不同的另一种平台上的程序。本次课程实验是在 Linux 环境中完成代码编译，获得编译产物，通过 Linux 和 Windows 资源共享，Windows 环境中获取编译产物后进行镜像烧录。

2.2.1 基础概念介绍

子系统：子系统是一个逻辑概念，它由一个或多个具体的部件组成。系统整体遵从分层设计，从下向上依次为：内核层、系统服务层、框架层和应用层。在多设备部署场景下，支持根据实际需求裁剪某些非必要的子系统或部件。

gn：Generate ninja 的缩写，用于产生 ninja 文件。

ninja：ninja 是一个专注于速度的小型构建系统。

hb：系统命令行工具，用来执行编译命令。

2.2.2 开发环境搭建

开发环境准备、源码获取、编译烧录等操作参考《HCIP-HarmonyOS Device Developer V1.0 实验环境搭建指南》。

2.3 添加部件

演示如何在 application 子系统中添加部件 my_sample。

2.3.1 编写源码

步骤 1 确定目录结构

开发者编写业务时，务必先在./applications/BearPi/BearPi-HM_Micro/samples 路径下新建一个目录（或一套目录结构），用于存放业务源码文件。

在 app 下新增业务 my_first_app，其中 hello_world.c 为业务代码，BUILD.gn 为编译脚本，具体规划目录结构如下：

```
.
└── applications
    └── BearPi
        └── BearPi-HM_Micro
            └── samples
                ├── my_first_app
                │   ├── hello_world.c
                │   └── BUILD.gn
```

步骤 2 编写业务代码

在 hello_world.c 中编写业务代码。

```
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("\n***************************************************\n");
    printf("\n\t\tHello BearPi!\n");
    printf("\n***************************************************\n\n");

    return 0;
}
```

编写将构建业务代码的 BUILD.gn 文件。BUILD.gn 文件由三部分内容（目标、源文件、头文件路径）构成。

```
import("//build/lite/config/component/lite_component.gni")

executable("hello_world_lib") {
```

```
    output_name = "hello_world"
    sources = [ "hello_world.c" ]
    include_dirs = []
    defines = []
    cflags_c = []
    ldflags = []
}

lite_component("my_app") {
    features = [
        ":hello_world_lib",
    ]
}
```

步骤 3 添加新部件

修改文件 build/lite/components/applications.json，添加部件 my_sample 的配置，如下所示为 applications.json 文件片段，"##start##"和"##end##"之间为新增配置（"##start##"和"##end##"仅用来标识位置，添加完配置后删除这两行）：

```
{
"components": [
    ##start##
    {
        "component": "my_sample",
        "description": "my samples",
        "optional": "true",
        "dirs": [
        "applications/BearPi/BearPi-HM_Micro/samples/my_first_app"
        ],
        "targets": [
        "//applications/BearPi/BearPi-HM_Micro/samples/my_first_app:my_app"
        ],
        "rom": "",
        "ram": "",
        "output": [],
        "adapted_kernel": [ "liteos_a" ],
        "features": [],
        "deps": {
        "components": [],
        "third_party": [ ]
        }
    },
    ##end##
    {
    "component": "bearpi_sample_app",
    "description": "bearpi_hm_micro samples.",
"optional": "true",
```

步骤 4 改单板配置文件

修改文件 vendor/bearpi/bearpi_hm_micro/config.json，新增 my_sample 部件的条目，如下所示代码片段为 applications 子系统配置，"##start##"和"##end##"之间为新增条目（"##start##"和"##end##"仅用来标识位置，添加完配置后删除这两行）：

```
{
"subsystem": "applications",
    "components": [
      { "component": "bearpi_sample_app", "features":[] },
    ##start##
      { "component": "my_sample", "features":[] },
    ##end##
      { "component": "bearpi_screensaver_app", "features":[] }
    ]
},
```

2.3.2 运行结果

示例代码按照 2.2 节编译构建的编译、烧录等步骤后，在命令行输入指令“./bin/hello_world”执行写入的 demo 程序，会显示如下结果：

```
OHOS # ./bin/hello_world
OHOS #
*****************************************************************************************

                                        Hello BearPi!

*****************************************************************************************
```

2.3.3 总结

在 config.json 中添加"my_sample"部件，"my_sample"部件在 applications.json 中被定义。

在 my_sample 的 targets 里面添加"my_app"的 lite_component 名称。

"my_app"字段链接到 my_first_app 文件下 BUILD.gn 里面的 lite_component。

lite_component 里指定 lib 库为"hello_world_lib"。

通过 hello_world_lib 里面 sources 来指定要编译的.c 文件，并通过 output_name 来指定生成的可执行程序名称。

2.4 启动恢复

2.4.1 init 启动引导部件

init 启动引导部件负责在系统启动阶段启动关键服务进程，若用户需要新增随开机自启动的系统服务，可将新增随开机自启动的系统服务加入配置文件 init.cfg 中。Init 启动引导部件对应

的进程为 init 进程，是内核完成初始化后启动的第一个用户态进程。init 进程启动之后，读取 init.cfg 配置文件，根据解析结果，执行相应命令并依次启动各关键系统服务进程，在启动系统服务进程的同时设置其对应权限。Init 启动引导部件的运行机制如下图所示。

图2-1 init 启动引导部件运行机制图

2.4.2 appspawn 启动引导部件

appspawn 启动引导部件负责接收用户程序框架的命令，孵化应用进程，设置新进程的权限，并调用应用程序框架的入口函数。当 Appspawn 被 Init 启动后，向 IPC（Inter-Process Communication 跨进程通信）框架注册服务名称，之后等待接收进程间消息，根据消息解析结果启动应用服务并赋予其对应权限。

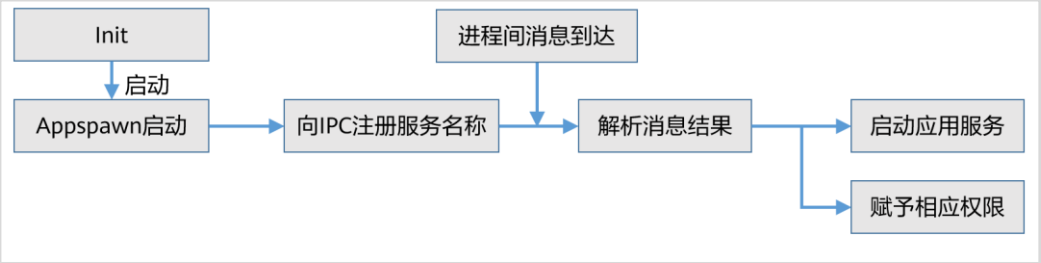

图2-2 appspawn 启动引导部件运行机制图

appspawn 注册的服务名称为“appspawn”，可通过包含“base\startup\appspawn_lite\services\include\appspawn_service.h”头文件，获取服务名称对应的宏 APPSPAWN_SERVICE_NAME 定义。在安全子系统限制规则下，目前仅 Ability Manager Service 有权限可以向 appspawn 发送的进程间消息。appspawn 接收的消息为 json 格式，如下所示：

```
"{\"bundleName\":\"testvalid1\",\"identityID\":\"1234\",\"uID\":1000,\"gID\":1000,\"capability\":[0]}"
```

bundleName 表示即将启动的应用服务进程名，长度≥7 字节，≤127 字节。

identityID 是 AMS 为新进程生成的标识符，由 appspawn 透传给新进程，长度≥1 字节，≤24 字节。

uID 即将启动的应用服务进程的 uID，必须为正值。

gID 即将启动的应用服务进程的 gID，必须为正值。

capability 即将启动的应用服务进程所需的 capability 权限，数量≤10 个。

2.4.3 bootstarp 服务启动部件

bootstarp 服务启动部件提供各服务和功能的启动入口标识。在 SAMGR（系统服务框架子系统）启动时，会调用 boostrap 标识的入口函数，并启动系统服务。它实现了服务的自动初始

化，即服务的初始化函数无需显式调用，而是将其使用宏定义的方式申明，就会在系统启动时自动被执行。Bootstarp 的原理是将服务启动的函数通过宏申明之后放在预定义好的 zInit 代码段中，系统启动的时候调用 OHOS_SystemInit 接口，遍历该代码段并调用其中的函数。

bootstrap 主要的服务自动初始化宏说明如下：

SYS_SERVICE_INIT(func)标识核心系统服务的初始化启动入口。

SYS_FEATURE_INIT(func)标识核心系统功能的初始化启动入口。

APP_SERVICE_INIT(func)标识应用层服务的初始化启动入口。

APP_FEATURE_INIT(func)标识应用层功能的初始化启动入口。

2.4.4 Syapara 系统属性部件

syspara 系统为各系统服务提供简单易用的键值对访问接口，使得各个系统服务可以通过各自的系统参数来进行业务功能的配置。

Syapara 系统属性部件的主要功能是负责提供获取与设置操作系统相关的系统属性。LiteOS-M 内核和 LiteOS-A 内核的平台，包括：Hi3861 平台，Hi3516DV300 平台，Hi3518EV300 平台。支持的系统属性包括：默认系统属性、OEM 厂商系统属性和自定义系统属性。OEM 厂商部分仅提供默认值，具体值需 OEM 产品方按需进行调整。

2.4.5 startup 启动部件

startup 启动部件的主要功能是负责提供大型系统获取与设置操作系统相关的系统属性。大型系统支持的系统属性包括：设备信息如设备类型、产品名称等，系统信息如系统版本、API 版本等默认系统属性。

3 LiteOS-A 内核

3.1 OpenHarmony 统一内核概述

内核是操作系统最为基本的部分，操作系统之所以能访问硬件设备，调用硬件设备，都依赖内核提供的对计算机硬件的访问能力。内核相关知识是做 OpenHarmony 硬件设备开发的基础，本章节主要分析 OpenHarmony 内核的主要运行机制，介绍内核如何管理系统的进程和线程，内存与网络，文件系统，以及设备的驱动程序。

OpenHarmony 具体使用到了 LiteOS 内核与 Linux 内核，而 LiteOS 内核具体又分为 LiteOS-A 内核与 LiteOS-M 内核，分别适用于 Cortex A 系列芯片与 Cortex M 系列芯片。在轻量系统、小型系统，可以选用 LiteOS；在标准系统上，可以选用 Linux。LiteOS 内核是面向 IoT 领域的实时操作系统内核，它同时具备 RTOS 轻快和 Linux 易用的特点，主要包括进程和线程调度、内存管理、IPC 机制、timer 管理等内核基本功能。在 OpenHarmony 当中，内核的源代码分为 kernel_liteos_a 和 kernel_liteos_m 这 2 个代码仓库，其中 kernel_liteos_a 主要针对小型系统（small system）和标准系统（standard system），而 kernel_liteos_m 则主要针对轻量系统（mini system）。

为了保证在不同硬件上集成的易用性，OpenHarmony 当前定义了三种基础系统类型，设备开发者通过选择基础系统类型完成必选组件集配置后，便可实现其最小系统的开发。这三种基础系统类型的参考定义如下：

轻量系统(mini system)面向 MCU 类处理器例如 ARM Cortex-M、RISC-V 32 位的设备，硬件资源极其有限，支持的设备最小内存为 128KiB，可以提供多种轻量级网络协议，轻量级的图形框架，以及丰富的 IOT 总线读写部件等。可支撑的产品如智能家居领域的连接类模组、传感器设备、穿戴类设备等。

小型系统(small system)面向应用处理器例如 ARM Cortex-A 的设备，支持的设备最小内存为 1MiB，可以提供更高的安全能力、标准的图形框架、视频编解码的多媒体能力。可支撑的产品如智能家居领域的 IP Camera、电子猫眼、路由器以及智慧出行域的行车记录仪等。

标准系统(standard system)面向应用处理器例如 ARM Cortex-A 的设备，支持的设备最小内存为 128MiB，可以提供增强的交互能力、3D GPU 以及硬件合成能力、更多控件以及动效更丰富的图形能力、完整的应用框架。可支撑的产品如高端的冰箱显示屏。

OpenHarmony 也提供了一系列可选的系统组件，方便设备开发者按需配置，以支撑其特色功能的扩展或定制开发。系统将这些可选的系统组件组合为一系列描述为特性或功能的系统能力，以方便设备开发者理解和选择。

表3-1 不同系统级别适配的 OpenHarmony 内核

系统级别	轻量系统	小型系统	标准系统
LiteOS-M	√	-	-
LiteOS-A	-	√	-
Linux	-	√	√

3.2 OpenHarmony 的 LiteOS-A 内核简介

OpenHarmony 轻量级内核是基于 IoT 领域轻量级物联网操作系统 Huawei LiteOS 内核演进发展的新一代内核，包含 LiteOS-M 和 LiteOS-A 两类内核。LiteOS-M 内核主要应用于轻量系统，面向的 MCU 一般是百 K 级内存，可支持 MPU 隔离，业界类似的内核有 FreeRTOS 或 ThreadX 等；LiteOS-A 内核主要应用于小型系统，面向设备一般是 M 级内存，可支持 MMU 隔离，业界类似的内核有 Zircon 或 Darwin 等。

为适应 IoT 产业的高速发展，OpenHarmony 轻量级内核不断优化和扩展，能够带给应用开发者友好的开发体验和统一开放的生态系统能力。轻量级内核 LiteOS-A 重要的新特性如下：

1、新增了丰富的内核机制：新增虚拟内存、系统调用、多核、轻量级 IPC（Inter-Process Communication，进程间通信）、DAC（Discretionary Access Control，自主访问控制）等机制，丰富了内核能力；为了更好地容软件和开发者体验，新增支持多进程，使得应用之间内存隔离、相互不影响，提升系统的健壮性。

2、引入统一驱动框架 HDF（Hardware Driver Foundation）：统一驱动标准，为设备厂商提供了更统一的接入方式，使驱动更加容易移植，力求做到一次开发，多系统部署。

3、支持 1200+标准 POSIX 接口，使得应用软件易于开发和移植，给应用开发者提供了更友好的开发体验。

4、轻量级内核与硬件高度解耦，新增单板，内核代码不用修改。

轻量级内核主要由基础内核、扩展组件、HDF 框架、POSIX 接口组成。轻量级内核的文件系统、网络协议等扩展功能（没有像微内核那样运行在用户态）运行在内核地址空间，主要考虑组件之间直接函数调用比进程间通信或远程过程调用要快得多。

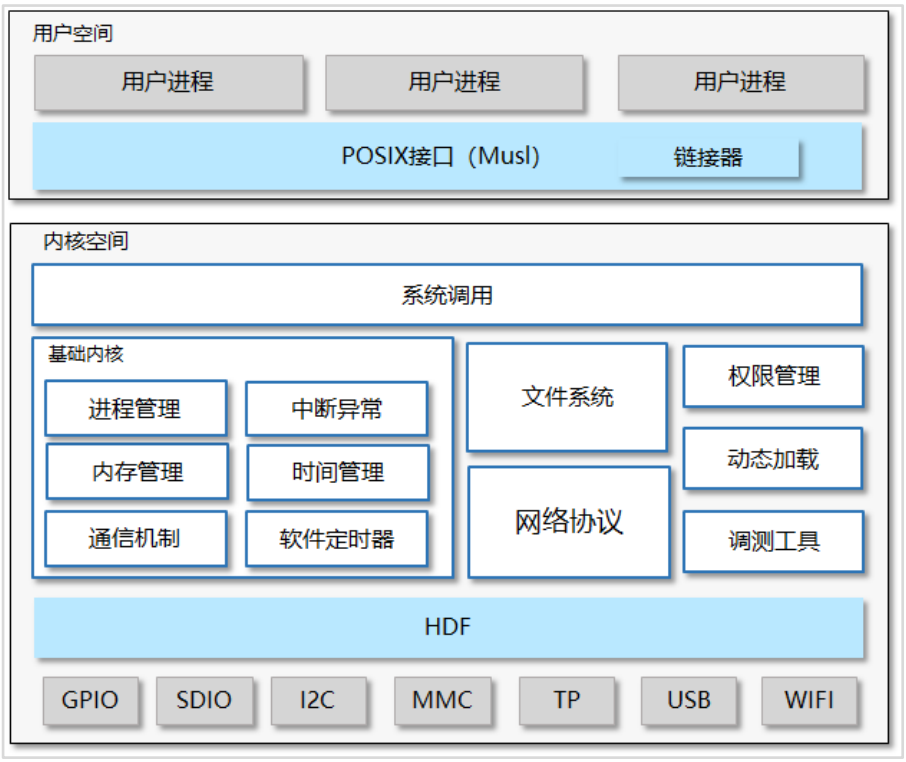

图3-1 LiteOS-A 内核架构图

基础内核主要包括内核的基础机制，如调度、内存管理、中断异常等，扩展组件主要包括文件系统、网络协议和安全等扩展功能，HDF 框架是外设驱动统一标准框架，POSIX 接口是为兼容 POSIX 标准的应用方便移植到 OpenHarmony。

基础内核：基础内核组件实现精简，主要包括内核的基础机制，如调度、内存管理、中断异常、内核通信等；

进程管理：支持进程和线程，基于 Task 实现进程，进程独立 4GiB 地址空间。

多核调度：支持任务和中断亲核性设置，支持绑核运行。

实时调度：支持高优先级抢占，同优先级时间片轮转。

虚拟内存：内核空间静态映射到 0-1GiB 地址，用户空间映射到 1-4GiB 地址。

内核通信：事件、信号量、互斥锁、队列。

时间管理：软件定时器、系统时钟。

文件系统：轻量级内核支持 FAT，JFFS2，NFS，ramfs，procfs 等众多文件系统，并对外提供完整的 POSIX 标准的操作接口；内部使用 VFS 层作为统一的适配层框架，方便移植新的文件系统，各个文件系统也能自动利用 VFS 层提供的丰富的功能。

主要特性有：完整的 POSIX 接口支持、文件级缓存(pagecache）、磁盘级缓存（bcache）、目录缓存(pathcache)、DAC 能力、支持嵌套挂载及文件系统堆叠等、支持特性的裁剪和资源占用的灵活配置。

网络协议：轻量级内核网络协议基于开源 LWIP 构建，对 LWIP 的 RAM 占用进行优化，同时提高 LWIP 的传输性能。

支持的协议有： IP、IPv6、 ICMP、 ND、MLD、 UDP、 TCP、IGMP、ARP、PPPoS、PPPoE。

API 采用 socket API。

扩展特性：多网络接口 IP 转发、TCP 拥塞控制、RTT 估计和快速恢复/快速重传。

应用程序：HTTP(S)服务、SNTP 客户端、SMTP(S)客户端、ping 工具、NetBIOS 名称服务、mDNS 响应程序、MQTT 客户端、TFTP 服务、DHCP 客户端、DNS 客户端、AutoIP/APIPA（零配置）、SNMP 代理。

HDF 框架：轻量级内核集成 HDF 框架，HDF 框架旨在为开发者提供更精准、更高效的开发环境，力求做到一次开发，多系统部署。

支持多内核平台、支持用户态驱动、可配置组件化驱动模型、基于消息的驱动接口模型、基于对象的驱动、设备管理、HDI（Hardware Driver Interface）统一硬件接口、支持电源管理、PnP。

扩展组件：对内核功能进行扩展，可选但很重要的机制。

动态链接：支持标准 ELF 链接执行、加载地址随机化。

进程通信：支持轻量级 LiteIPC，同时也支持标准的 Mqueue、Pipe、Fifo、Signal 等机制。

系统调用：支持 170+系统调用，同时有支持 VDSO 机制。

权限管理：支持进程粒度的特权划分和管控，UGO 三种权限配置。

3.3 内核启动

内核启动流程包含汇编启动阶段和 C 语言启动阶段两部分。汇编启动阶段完成 CPU 初始设置，关闭 dcache/icache，使能 FPU 及 neon，设置 MMU 建立虚实地址映射，设置系统栈，清理 bss 段，调用 C 语言 main 函数等。C 语言启动阶段包含 OsMain 函数及开始调度等，OsMain 函数用于内核基础初始化和架构、板级初始化等，其整体由内核启动框架主导初始化流程。

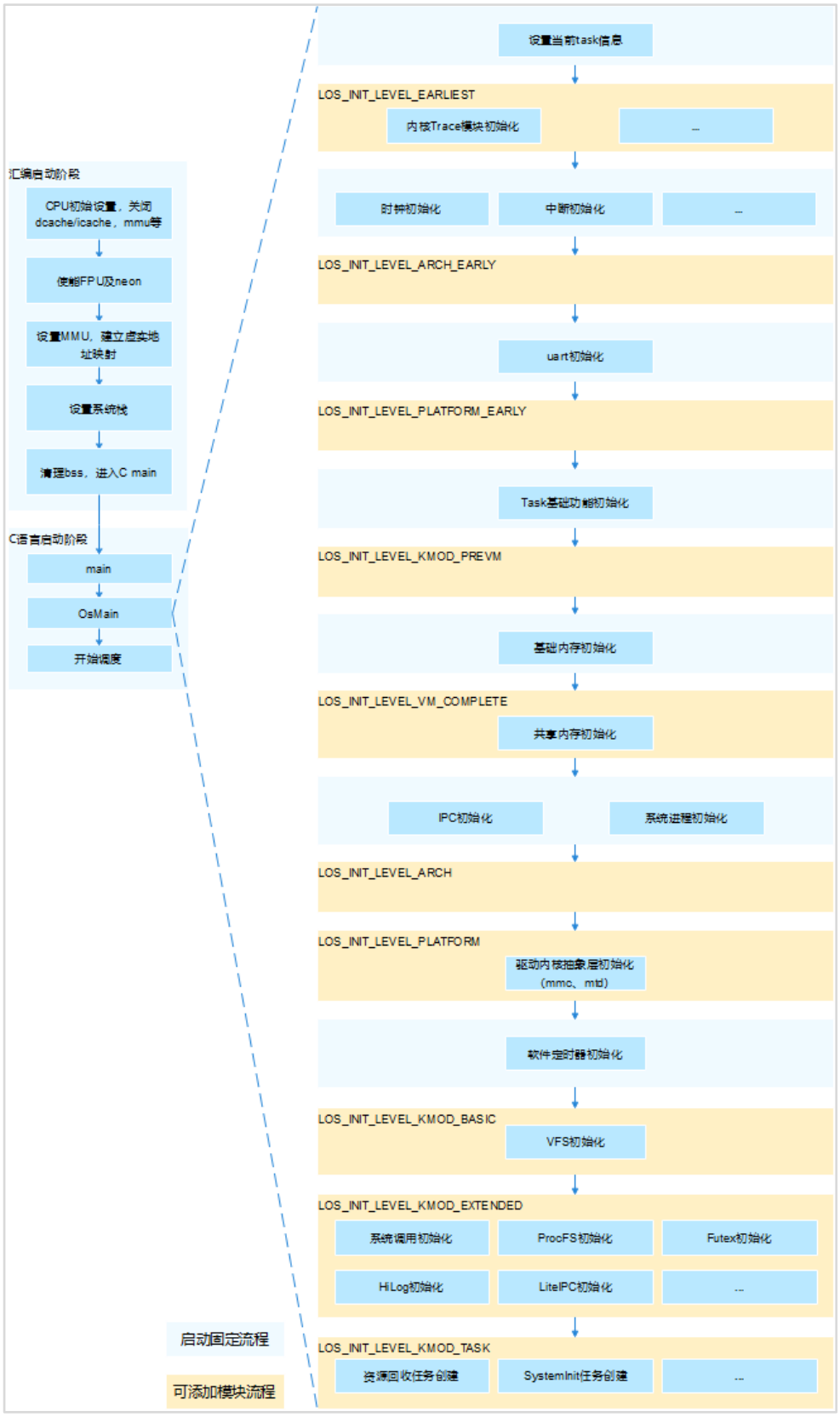

图3-2 内核启动原理图

根进程是系统第一个用户态进程，进程 ID 为 1，它是所有用户态进程的祖先。

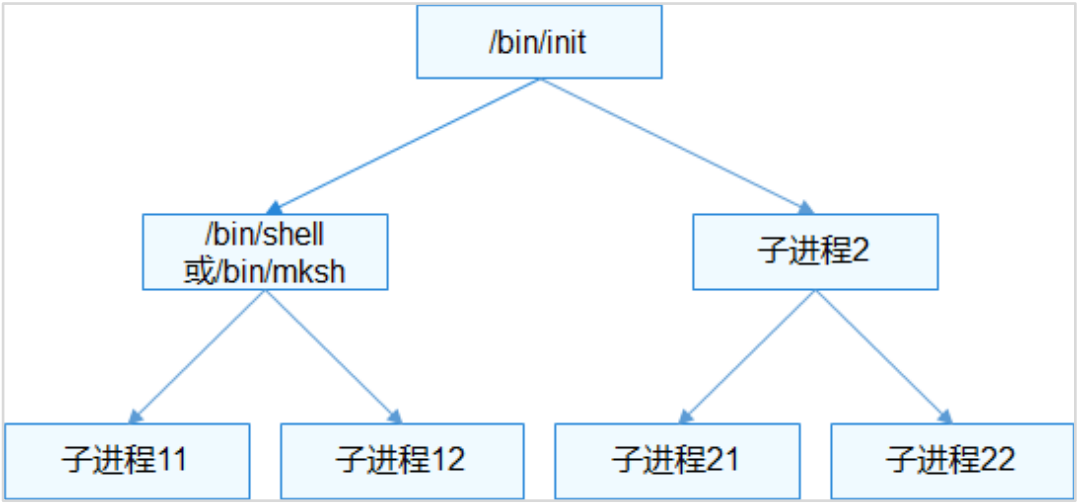

图3-3 根进程原理图

3.4 中断及异常处理

中断是指出现需要时，CPU 暂停执行当前程序，转而执行新程序的过程。即在程序运行过程中，出现了一个必须由 CPU 立即处理的事务。此时，CPU 暂时中止当前程序的执行转而处理这个事务，这个过程就叫做中断。通过中断机制，可以使 CPU 避免把大量时间耗费在等待、查询外设状态的操作上，大大提高系统实时性以及执行效率。

异常处理是操作系统对运行期间发生的异常情况（芯片硬件异常）进行处理的一系列动作，例如虚拟内存缺页异常、打印异常发生时函数的调用栈信息、CPU 现场信息、任务的堆栈情况等。

外设可以在没有 CPU 介入的情况下完成一定的工作，但某些情况下也需要 CPU 为其执行一定的工作。通过中断机制，在外设不需要 CPU 介入时，CPU 可以执行其它任务，而当外设需要 CPU 时，产生一个中断信号，该信号连接至中断控制器。中断控制器是一方面接收其它外设中断引脚的输入，另一方面它会发出中断信号给 CPU。可以通过对中断控制器编程来打开和关闭中断源、设置中断源的优先级和触发方式。常用的中断控制器有 VIC（Vector Interrupt Controller）和 GIC（General Interrupt Controller）。在 ARM Cortex-A7 中使用的中断控制器是 GIC。CPU 收到中断控制器发送的中断信号后，中断当前任务来响应中断请求。

异常处理就是可以打断 CPU 正常运行流程的一些事情，如未定义指令异常、试图修改只读的数据异常、不对齐的地址访问异常等。当异常发生时，CPU 暂停当前的程序，先处理异常事件，然后再继续执行被异常打断的程序。

以 ARMv7-a 架构为例，中断和异常处理的入口为中断向量表，中断向量表包含各个中断和异常处理的入口函数。

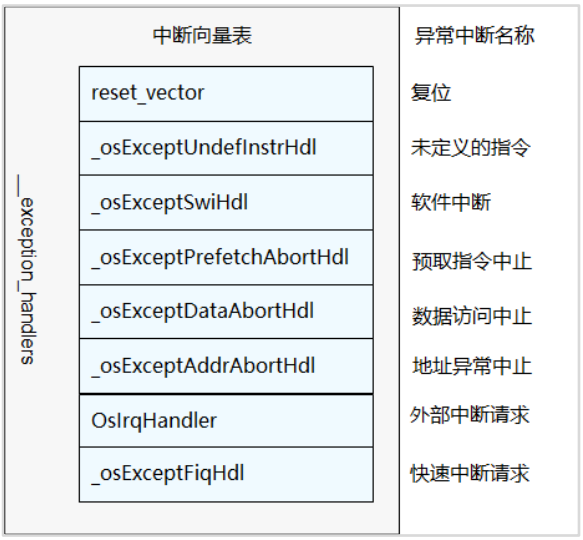

图3-4 中断向量表示意图

异常处理为内部机制，不对外提供接口，中断模块提供对外接口如下。

表3-2 中断部分接口说明

功能分类	接口名称	描述
创建和删除中断	LOS_HwiCreate	中断创建，注册中断号、中断触发模式、中断优先级、中断处理程序。中断被触发时，会调用该中断处理程序
创建和删除中断	LOS_HwiDelete	删除中断
打开和关闭所有中断	LOS_IntUnLock	打开当前处理器所有中断响应
打开和关闭所有中断	LOS_IntLock	关闭当前处理器所有中断响应
打开和关闭所有中断	LOS_IntRestore	恢复到使用LOS_IntLock关闭所有中断之前的状态
获取系统支持的最大中断数	LOS_GetSystemHwiMaximum	获取系统支持的最大中断数

实例开发流程共有两步，第一步调用中断创建接口 LOS_HwiCreate 创建中断，第二步调用 LOS_HwiDelete 接口删除指定中断，此接口根据实际情况使用，判断是否需要删除中断。本实例主要实现两个功能：创建中断与删除中断。

代码实现如下，演示如何创建中断和删除中断，当指定的中断号 HWI_NUM_TEST 产生中断时，会调用中断处理函数：

```
#include "los_hwi.h"
/*中断处理函数*/
STATIC VOID HwiUsrIrq(VOID)
```

```
{
    printf("in the func HwiUsrIrq \n");
}

static UINT32 Example_Interrupt(VOID)
{
    UINT32 ret;
    HWI_HANDLE_T hwiNum = 7;
    HWI_PRIOR_T hwiPrio = 3;
    HWI_MODE_T mode = 0;
    HWI_ARG_T arg = 0;

    /*创建中断*/
    ret = LOS_HwiCreate(hwiNum, hwiPrio, mode, (HWI_PROC_FUNC)HwiUsrIrq, (HwiIrqParam *)arg);
    if(ret == LOS_OK){
        printf("Hwi create success!\n");
    } else {
        printf("Hwi create failed!\n");
        return LOS_NOK;
    }

    /* 延时 50 个 Ticks， 当有硬件中断发生时，会调用函数 HwiUsrIrq*/
    LOS_TaskDelay(50);

    /*删除中断*/
    ret = LOS_HwiDelete(hwiNum, (HwiIrqParam *)arg);
    if(ret == LOS_OK){
        printf("Hwi delete success!\n");
    } else {
        printf("Hwi delete failed!\n");
        return LOS_NOK;
    }
    return LOS_OK;
}
```

编译运行得到的结果如下：

```
Hwi create success!
Hwi delete success!
```

3.5 进程管理

3.5.1 基本概念

进程是系统资源管理的最小单元。OpenHarmony 的 LiteOS-A 内核提供的进程模块主要用于实现用户态进程的隔离，内核态被视为一个进程空间，不存在其它进程(KIdle 除外，KIdle 进程是系统提供的空闲进程，和 KProcess 共享一个进程空间）。

LiteOS-A 内核的进程模块主要为用户提供多个进程，实现了进程之间的切换和通信，帮助用户管理业务程序流程。LiteOS-A 内核的进程有以下几个特点：

- LiteOS-A 内核的进程采用抢占式调度机制，采用高优先级优先+同优先级时间片轮转的调度算法。
- LiteOS-A 内核的进程一共有 32 个优先级(0-31)，用户进程可配置的优先级有 22 个(10-31)，最高优先级为 10，最低优先级为 31。高优先级的进程可抢占低优先级进程，低优先级进程必须在高优先级进程阻塞或结束后才能得到调度。
- 每一个用户态进程均拥有自己独立的进程空间，相互之间不可见，实现进程间隔离。用户态根进程 init 由内核态创建，其它用户态子进程均由 init 进程 fork 而来。

进程状态说明：初始化 Init 表示进程正在被创建；就绪 Ready 表示进程在就绪列表中，等待 CPU 调度；运行 Running 表示进程正在运行；阻塞 Pending 表示进程被阻塞挂起；本进程内所有的线程均被阻塞时，进程被阻塞挂起；僵尸态 Zombies 表示进程运行结束，等待父进程回收其控制块资源。

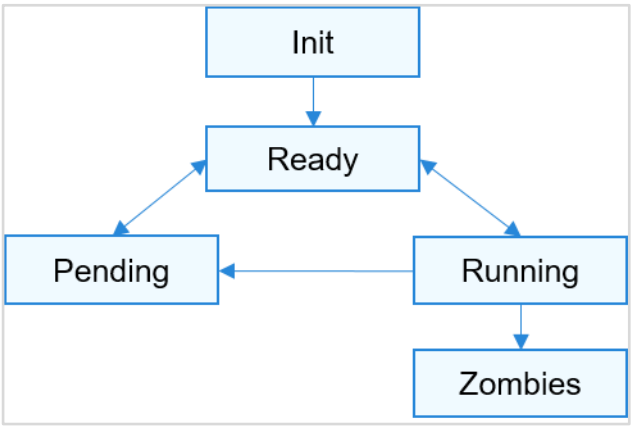

图3-5 进程状态迁移示意图

进程状态迁移说明：

- Init→Ready：

进程创建或 fork 时，拿到该进程控制块后进入 Init 状态，处于进程初始化阶段，当进程初始化完成将进程插入调度队列，此时进程进入就绪状态。

- Ready→Running：

进程创建后进入就绪态，发生进程切换时，就绪列表中最高优先级的进程被执行，从而进入运行态。若此时该进程中已无其它线程处于就绪态，则进程从就绪列表删除，只处于运行态；若

此时该进程中还有其它线程处于就绪态，则该进程依旧在就绪队列，此时进程的就绪态和运行态共存，但对外呈现的进程状态为运行态。

- Running→Pending：

进程在最后一个线程转为阻塞态时， 进程内所有的线程均处于阻塞态，此时进程同步进入阻塞态，然后发生进程切换。

- Pending→Ready：

阻塞进程内的任意线程恢复就绪态时，进程被加入到就绪队列，同步转为就绪态。

- Ready→Pending：

进程内的最后一个就绪态线程转为阻塞态时，进程从就绪列表中删除，进程由就绪态转为阻塞态。

- Running→Ready：

进程由运行态转为就绪态的情况有以下两种：

1、有更高优先级的进程创建或者恢复后，会发生进程调度，此刻就绪列表中最高优先级进程变为运行态，那么原先运行的进程由运行态变为就绪态。

2、若进程的调度策略为 LOS_SCHED_RR，且存在同一优先级的另一个进程处于就绪态，则该进程的时间片消耗光之后，该进程由运行态转为就绪态，另一个同优先级的进程由就绪态转为运行态。

- Running→Zombies：当进程的主线程或所有线程运行结束后，进程由运行态转为僵尸态，等待父进程回收资源。

3.5.2 运行机制

LiteOS-A 内核提供的进程模块主要用于实现用户态进程的隔离，支持用户态进程的创建、退出、资源回收、设置/获取调度参数、获取进程 ID、设置/获取进程组 ID 等功能。用户态进程通过 fork 父进程而来，fork 进程时会将父进程的进程虚拟内存空间 clone 到子进程，子进程实际运行时通过写时复制机制将父进程的内容按需复制到子进程的虚拟内存空间。进程只是资源管理单元，实际运行是由进程内的各个线程完成的，不同进程内的线程相互切换时会进行进程空间的切换。

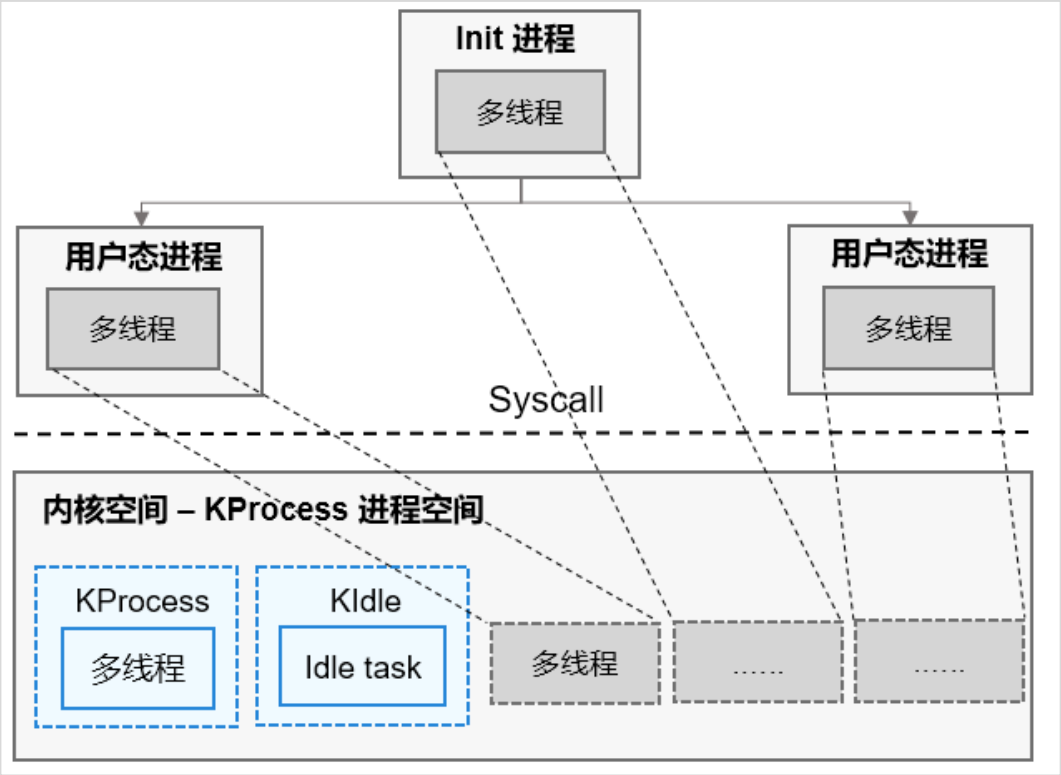

图3-6 进程管理的示意图

3.6 线程管理

从系统的角度看，线程是竞争系统资源的最小运行单元。线程可以使用或等待 CPU、使用内存空间等系统资源，并独立于其它线程运行。LiteOS-A 内核中同优先级进程内的线程统一调度、运行。LiteOS-A 内核中的线程采用抢占式调度机制，同时支持时间片轮转调度和 FIFO 调度方式。LiteOS-A 内核的线程一共有 32 个优先级(0-31)，最高优先级为 0，最低优先级为 31。当前进程内，高优先级的线程可抢占低优先级线程，低优先级线程必须在高优先级线程阻塞或结束后才能得到调度。

线程的状态说明：初始化 Init 表示该线程正在被创建；就绪 Ready 表示该线程在就绪列表中，等待 CPU 调度；运行 Running 表示该线程正在运行；阻塞表示 Blocked 表示该线程被阻塞挂起，Blocked 状态包括 pending(因为锁、事件、信号量等阻塞)、suspended（主动 pend）、delay(延时阻塞)、pendtime(因为锁、事件、信号量时间等超时等待)；退出 Exit 表示该线程运行结束，等待父线程回收其控制块资源。

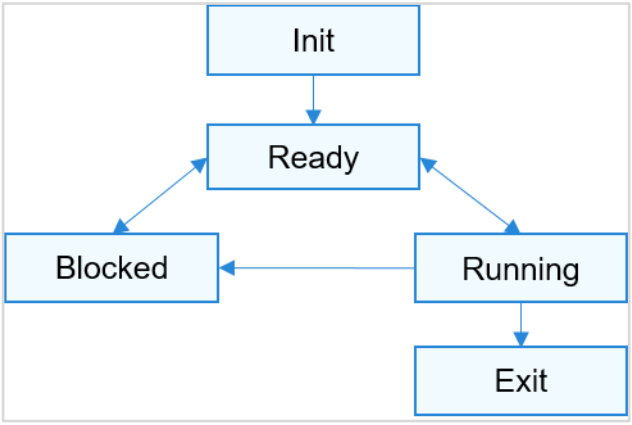

图3-7 线程状态迁移示意图

线程状态迁移详情：

- Init→Ready：

线程创建拿到控制块后为初始化阶段(Init 状态)，当线程初始化完成将线程插入调度队列，此时线程进入就绪状态。

- Ready→Running：

线程创建后进入就绪态，发生线程切换时，就绪列表中最高优先级的线程被执行，从而进入运行态，此刻该线程从就绪列表中删除。

- Running→Blocked：

正在运行的线程发生阻塞（挂起、延时、读信号量等）时，线程状态由运行态变成阻塞态，然后发生线程切换，运行就绪列表中剩余最高优先级线程。

- Blocked→Ready ：

阻塞的线程被恢复后（线程恢复、延时时间超时、读信号量超时或读到信号量等），此时被恢复的线程会被加入就绪列表，从而由阻塞态变成就绪态。

- Ready→Blocked：

线程也有可能在就绪态时被阻塞（挂起），此时线程状态会由就绪态转变为阻塞态，该线程从就绪列表中删除，不会参与线程调度，直到该线程被恢复。

- Running→Ready：

有更高优先级线程创建或者恢复后，会发生线程调度，此刻就绪列表中最高优先级线程变为运行态，那么原先运行的线程由运行态变为就绪态，并加入就绪列表中。

- Running→Exit：

运行中的线程运行结束，线程状态由运行态变为退出态。若为设置了分离属性（LOS_TASK_STATUS_DETACHED）的线程，运行结束后将直接销毁。

LiteOS-A 内核线程管理模块提供线程创建、线程延时、线程挂起和线程恢复、锁线程调度和解锁线程调度、根据 ID 查询线程控制块信息功能。用户创建线程时，系统会将线程栈进行初始化，预置上下文。此外，系统还会将“线程入口函数”地址放在相应位置。这样在线程第一次启动进入运行态时，将会执行线程入口函数。

3.7 调度器

LiteOS-A 内核提供了高优先级优先+同优先级时间片轮转的抢占式调度机制，系统从启动开始基于 real time 的时间轴向前运行，使得该调度算法具有很好的实时性。调度算法将 tickless 机制天然嵌入到调度算法中，一方面使得系统具有更低的功耗，另一方面也使得 tick 中断按需响应，减少无用的 tick 中断响应，进一步提高系统的实时性。进程调度策略支持 SCHED_RR，线程调度策略支持 SCHED_RR 和 SCHED_FIFO。调度的最小单元为线程。

采用进程优先级队列+线程优先级队列的方式，进程优先级范围为 0-31，共有 32 个进程优先级桶队列，每个桶队列对应一个线程优先级桶队列；线程优先级范围也为 0-31，一个线程优先级桶队列也有 32 个优先级队列。

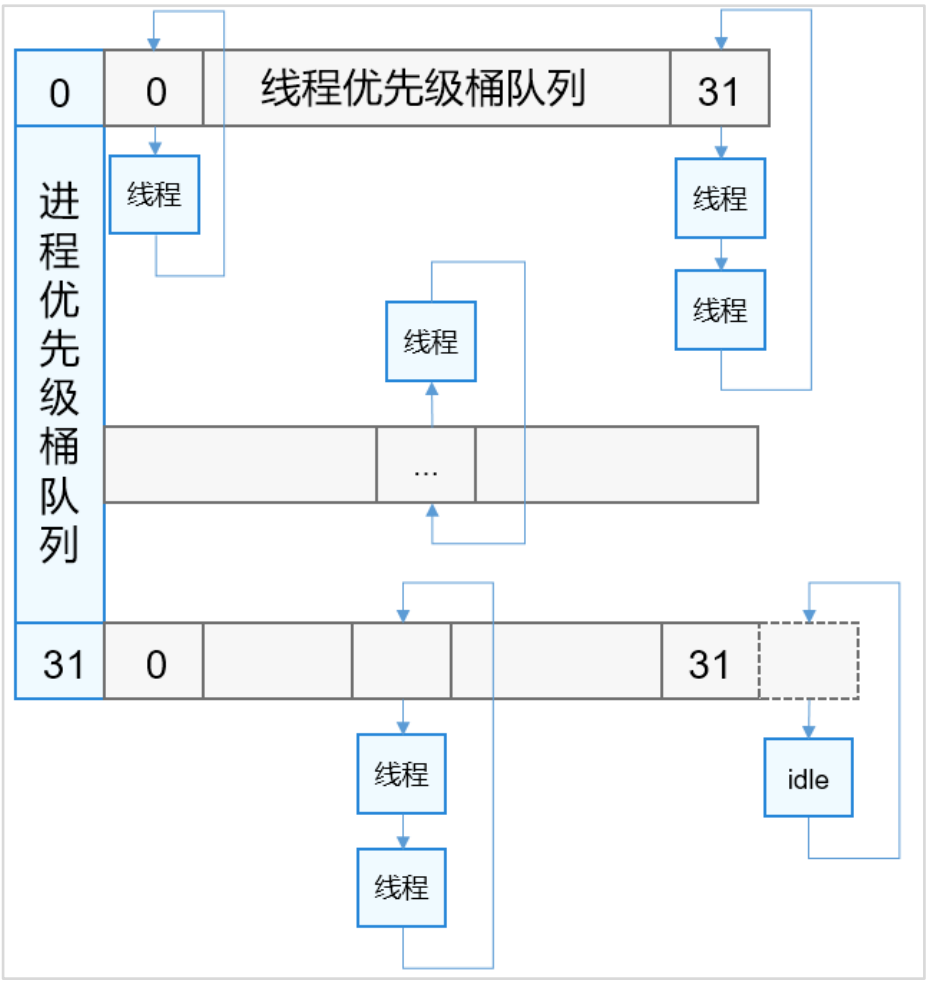

图3-8 调度优先级桶队列示意图

在系统启动内核初始化之后开始调度，运行过程中创建的进程或线程会被加入到调度队列，系统根据进程和线程的优先级及线程的时间片消耗情况选择最优的线程进行调度运行，线程一旦调度到就会从调度队列上删除，线程在运行过程中发生阻塞，会被加入到对应的阻塞队列中并触发一次调度，调度其它线程运行。如果调度队列上没有可以调度的线程，则系统就会选择 KIdle 进程的线程进行调度运行。

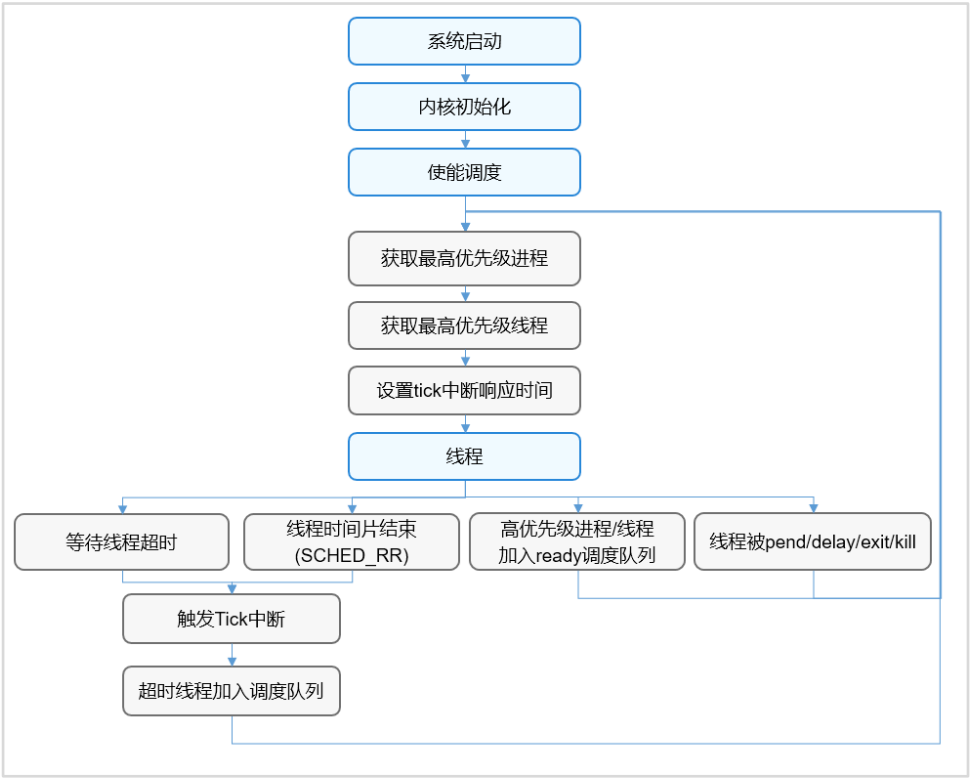

图3-9 调度流程示意图

3.8 内存管理

3.8.1 堆内存管理

内存管理模块管理系统的内存资源，它是操作系统的核心模块之一，主要包括内存的初始化、分配以及释放。OpenHarmony 的 LiteOS-A 内核的堆内存管理提供内存初始化、分配、释放等功能。在系统运行过程中，堆内存管理模块通过对内存的申请/释放来管理用户和 OS 对内存的使用，使内存的利用率和使用效率达到最优，同时最大限度地解决系统的内存碎片问题。

堆内存管理，即在内存资源充足的情况下，根据用户需求，从系统配置的一块比较大的连续内存（内存池，也是堆内存）中分配任意大小的内存块。当用户不需要该内存块时，又可以释放回系统供下一次使用。与静态内存相比，动态内存管理的优点是按需分配，缺点是内存池中容易出现碎片。OpenHarmony 的 LiteOS-A 堆内存在 TLSF 算法的基础上，对区间的划分进行了优化，获得更优的性能，降低了碎片率。

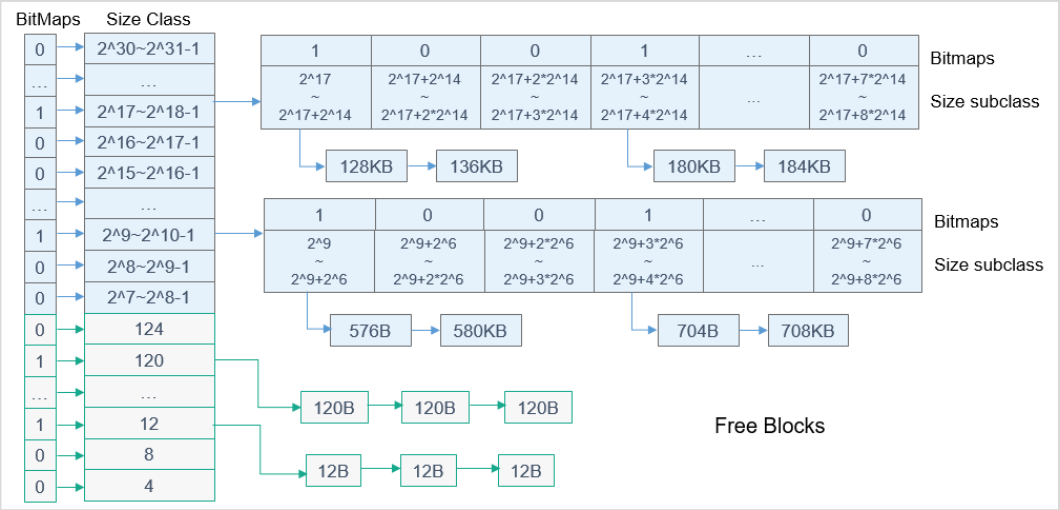

图3-10 动态内存核心算法框图

根据空闲内存块的大小，使用多个空闲链表来管理。根据内存空闲块大小分为两个部分：[4, 127]和[2^7, 2^31]，如上图 size class 所示：

对[4,127]区间的内存进行等分，如上图绿色部分所示，分为 31 个小区间，每个小区间对应内存块大小为 4 字节的倍数。每个小区间对应一个空闲内存链表和用于标记对应空闲内存链表是否为空的一个比特位，值为 1 时，空闲链表非空。[4,127]区间的 31 个小区间内存对应 31 个比特位进行标记链表是否为空。

大于 127 字节的空闲内存块，按照 2 的次幂区间大小进行空闲链表管理。总共分为 24 个小区间，每个小区间又等分为 8 个二级小区间，见上图蓝色的 Size Class 和 Size SubClass 部分。每个二级小区间对应一个空闲链表和用于标记对应空闲内存链表是否为空的一个比特位。总共 24*8=192 个二级小区间，对应 192 个空闲链表和 192 个比特位进行标记链表是否为空。

例如，当有 40 字节的空闲内存需要插入空闲链表时，对应小区间[40,43]，第 10 个空闲链表，位图标记的第 10 比特位。把 40 字节的空闲内存挂载第 10 个空闲链表上，并判断是否需要更新位图标记。当需要申请 40 字节的内存时，根据位图标记获取存在满足申请大小的内存块的空闲链表，从空闲链表上获取空闲内存节点。如果分配的节点大于需要申请的内存大小，进行分割节点操作，剩余的节点重新挂载到相应的空闲链表上。当有 580 字节的空闲内存需要插入空闲链表时，对应二级小区间[2^9,2^9+2^6]，第 31+2*8=47 个空闲链表，并使用位图的第 47 个比特位来标记链表是否为空。把 580 字节的空闲内存挂载第 47 个空闲链表上，并判断是否需要更新位图标记。当需要申请 580 字节的内存时，根据位图标记获取存在满足申请大小的内存块的空闲链表，从空闲链表上获取空闲内存节点。如果分配的节点大于需要申请的内存大小，进行分割节点操作，剩余的节点重新挂载到相应的空闲链表上。如果对应的空闲链表为空，则向更大的内存区间去查询是否有满足条件的空闲链表，实际计算时，会一次性查找到满足申请大小的空闲链表。

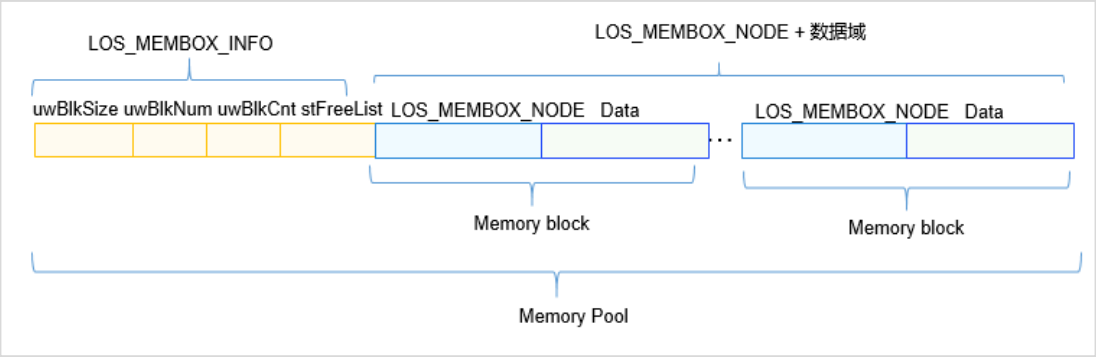

图3-11 内存管理结构示意图

内存池池头部分包含内存池信息、位图标记数组和空闲链表数组。内存池信息包含内存池起始地址及堆区域总大小，内存池属性。位图标记数组有 7 个 32 位无符号整数组成，每个比特位标记对应的空闲链表是否挂载空闲内存块节点。空闲内存链表包含 223 个空闲内存头节点信息，每个空闲内存头节点信息维护内存节点头和空闲链表中的前驱、后继空闲内存节点。

内存池节点部分包含 3 种类型节点：未使用空闲内存节点，已使用内存节点和尾节点。每个内存节点维护一个前序指针，指向内存池中上一个内存节点，还维护内存节点的大小和使用标记。空闲内存节点和已使用内存节点后面的内存区域是数据域，尾节点没有数据域。

3.8.2 物理内存管理

物理内存是计算机上最重要的资源之一，指的是实际的内存设备提供的、可以通过 CPU 总线直接进行寻址的内存空间，其主要作用是为操作系统及程序提供临时存储空间。LiteOS-A 内核管理物理内存是通过分页实现的，除了内核堆占用的一部分内存外，其余可用内存均以 4k 为单位划分成页帧，内存分配和内存回收便是以页帧为单位进行操作。内核采用伙伴算法管理空闲页面，可以降低一定的内存碎片率，提高内存分配和释放的效率，但是一个很小的块往往也会阻塞一个大块的合并，导致不能分配较大的内存块。

运行机制如下图所示，LiteOS-A 内核的物理内存使用分布视图，主要由内核镜像、内核堆及物理页组成。

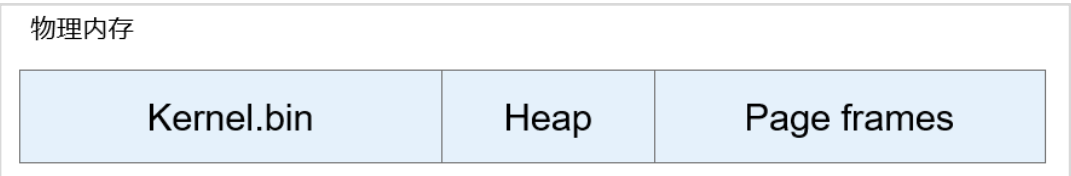

图3-12 物理内存示意图

伙伴算法把所有空闲页帧分成 9 个内存块组，每组中内存块包含 2 的幂次方个页帧，例如：第 0 组的内存块包含 2 的 0 次方个页帧，即 1 个页帧；第 8 组的内存块包含 2 的 8 次方个页帧，即 256 个页帧。相同大小的内存块挂在同一个链表上进行管理。

申请内存系统申请 12k 内存，即 3 个页帧时，9 个内存块组中索引为 3 的链表挂着一块大小为 8 个页帧的内存块满足要求，分配出 12k 内存后还剩余 20k 内存，即 5 个页帧，将 5 个页帧分成 2 的幂次方之和，即 4 跟 1，尝试查找伙伴进行合并。4 个页帧的内存块没有伙伴则直接插到索引为 2 的链表上，继续查找 1 个页帧的内存块是否有伙伴，索引为 0 的链表上此时

有 1 个，如果两个内存块地址连续则进行合并，并将内存块挂到索引为 1 的链表上，否则不做处理。

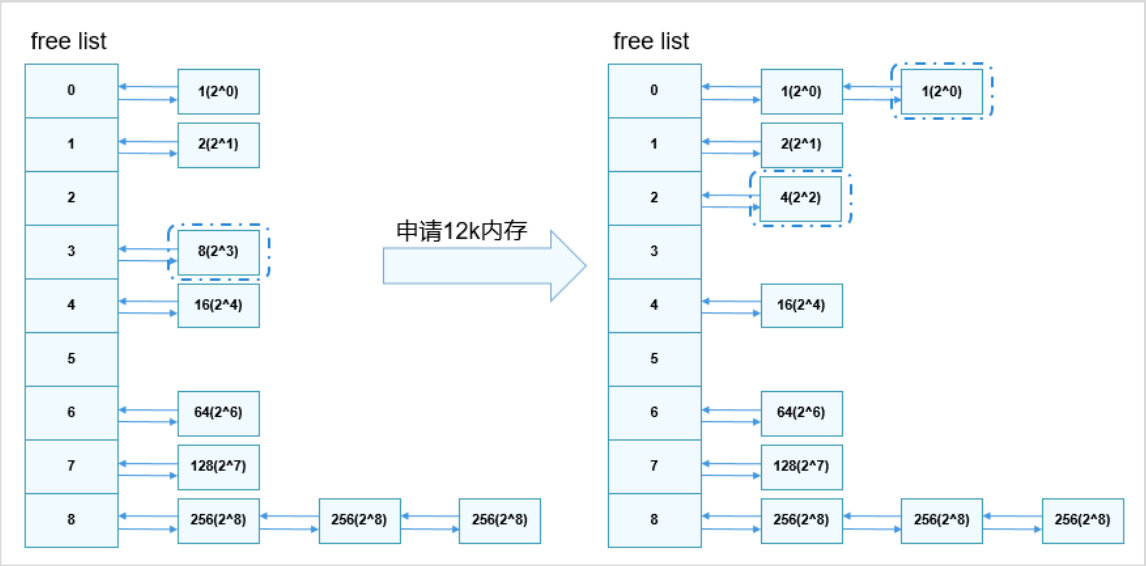

图3-13 申请内存示意图

释放内存：系统释放 12k 内存，即 3 个页帧，将 3 个页帧分成 2 的幂次方之和，即 2 跟 1，尝试查找伙伴进行合并，索引为 1 的链表上有 1 个内存块，若地址连续则合并，并将合并后的内存块挂到索引为 2 的链表上，索引为 0 的链表上此时也有 1 个，如果地址连续则进行合并，并将合并后的内存块挂到索引为 1 的链表上，此时继续判断是否有伙伴，重复上述操作。

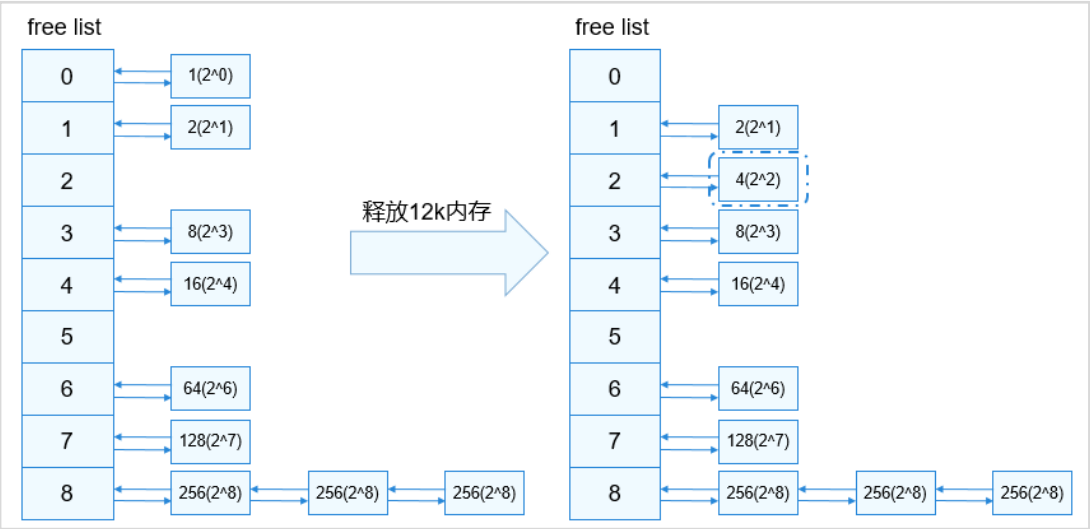

图3-14 内存释放示意图

3.8.3 虚拟内存管理

虚拟内存管理是计算机系统管理内存的一种技术。每个进程都有连续的虚拟地址空间，虚拟地址空间的大小由 CPU 的位数决定，32 位的硬件平台可以提供的最大的寻址空间为 0-4G。整

个 4G 空间分成两部分，LiteOS-A 内核占据 3G 的高地址空间，1G 的低地址空间留给进程使用。各个进程空间的虚拟地址空间是独立的，代码、数据互不影响。

系统将虚拟内存分割为称为虚拟页的内存块，大小一般为 4k 或 64k，LiteOS-A 内核默认的页的大小是 4k，根据需要可以对 MMU（Memory Management Units）进行配置。虚拟内存管理操作的最小单位就是一个页，LiteOS-A 内核中一个虚拟地址区间 region 包含地址连续的多个虚拟页，也可只有一个页。同样，物理内存也会按照页大小进行分割，分割后的每个内存块称为页帧。虚拟地址空间划分：内核态占高地址 3G(0x40000000 ~ 0xFFFFFFFF)，用户态占低地址 1G(0x01000000 ~ 0x3F000000)，具体见下表，详细可以查看或配置 los_vm_zone.h。

表3-3 内核态地址规划

Zone名称	起始地址	结束地址	用途	属性
DMA zone	0x40000000	0x43FFFFFF	USB、网络等dma内存访问	Uncache
Normal zone	0x80000000	0x83FFFFFF	内核代码、数据段和堆内存和栈	Cache
high mem zone	0x84000000	0x8BFFFFFF	连续虚拟内存分配，物理内存不连续	Cache

表3-4 用户态虚地址规划

Zone名称	起始地址	结束地址	用途	属性
代码段	0x0200000	0x09FFFFFF	用户态代码段地址空间	Cache
堆	0x0FC00000(起始地址随机)	0x17BFFFFF	用户态堆地址空间	Cache
栈	0x37000000	0x3EFFFFFF(起始地址随机)	用户态栈空间地址	Cache
共享库	0x1F800000(起始地址随机)	0x277FFFFF	用户态共享库加载地址空间，包括mmap	Cache

虚拟内存管理中，虚拟地址空间是连续的，但是其映射的物理内存并不一定是连续的，如下图所示。可执行程序加载运行，CPU 访问虚拟地址空间的代码或数据时存在两种情况：

CPU 访问的虚拟地址所在的页，如 V0，已经与具体的物理页 P0 做映射，CPU 通过找到进程对应的页表条目（详见虚实映射一节），根据页表条目中的物理地址信息访问物理内存中的内容并返回。

CPU 访问的虚拟地址所在的页，如 V2，没有与具体的物理页做映射，系统会触发缺页异常，系统申请一个物理页，并把相应的信息拷贝到物理页中，并且把物理页的起始地址更新到页表条目中。此时 CPU 重新执行访问虚拟内存的指令便能够访问到具体的代码或数据。

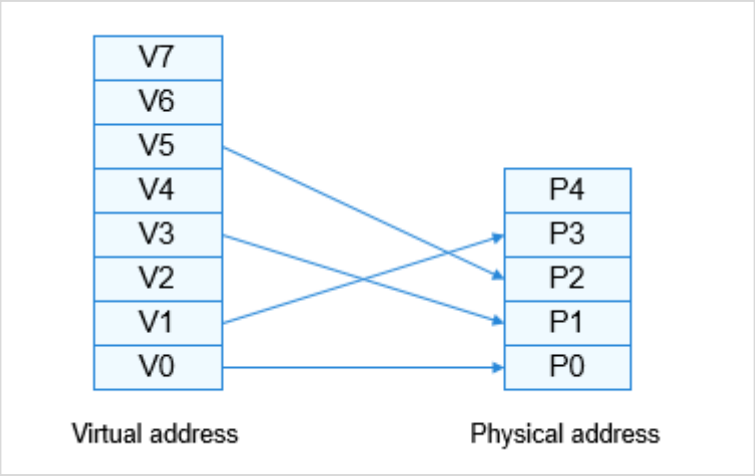

图3-15 内存映射示意图

3.8.4 虚实映射

虚实映射是指系统通过内存管理单元（MMU 全称 Memory Management Unit）将进程空间的虚拟地址与实际的物理地址做映射，并指定相应的访问权限、缓存属性等。程序执行时，CPU 访问的是虚拟内存，通过 MMU 页表条目找到对应的物理内存，并做相应的代码执行或数据读写操作。MMU 的映射由页表（全称 Page Table）来描述，其中保存虚拟地址和物理地址的映射关系以及访问权限等。每个进程在创建的时候都会创建一个页表，页表由一个个页表条目（Page Table Entry， 简称 PTE）构成，每个页表条目描述虚拟地址区间与物理地址区间的映射关系。MMU 中有一块页表缓存，称为快表（TLB 全称 Translation Lookaside Buffers），做地址转换时，MMU 首先在 TLB 中查找，如果找到对应的页表条目可直接进行转换，提高了查询效率。

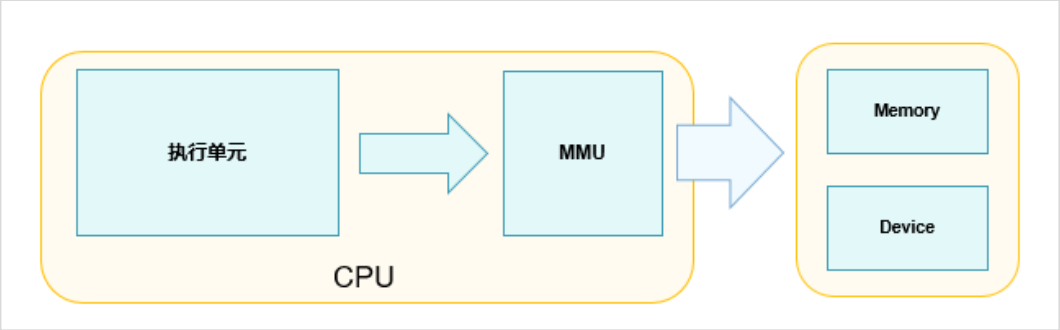

图3-16 CPU 访问内存或外设的示意图

虚实映射其实就是一个建立页表的过程。MMU 有多级页表，LiteOS-A 内核采用二级页表描述进程空间。每个一级页表条目描述符占用 4 个字节，可表示 1MiB 的内存空间的映射关系，即 1GiB 用户空间（LiteOS-A 内核中用户空间占用 1GiB）的虚拟内存空间需要 1024 个。系统创建用户进程时，在内存中申请一块 4KiB 大小的内存块作为一级页表的存储区域，二级页表根据当前进程的需要做动态的内存申请。

用户程序加载启动时，会将代码段、数据段映射进虚拟内存空间（详细可参考动态加载与链接一节），此时并没有物理页做实际的映射；程序执行时，如下图粗箭头所示，CPU 访问虚拟地址，通过 MMU 查找是否有对应的物理内存，若该虚拟地址无对应的物理地址则触发缺页

异常，内核申请物理内存并将虚实映射关系及对应的属性配置信息写进页表，并把页表条目缓存至 TLB，接着 CPU 可直接通过转换关系访问实际的物理内存；若 CPU 访问已缓存至 TLB 的页表条目，无需再访问保存在内存中的页表，可加快查找速度。

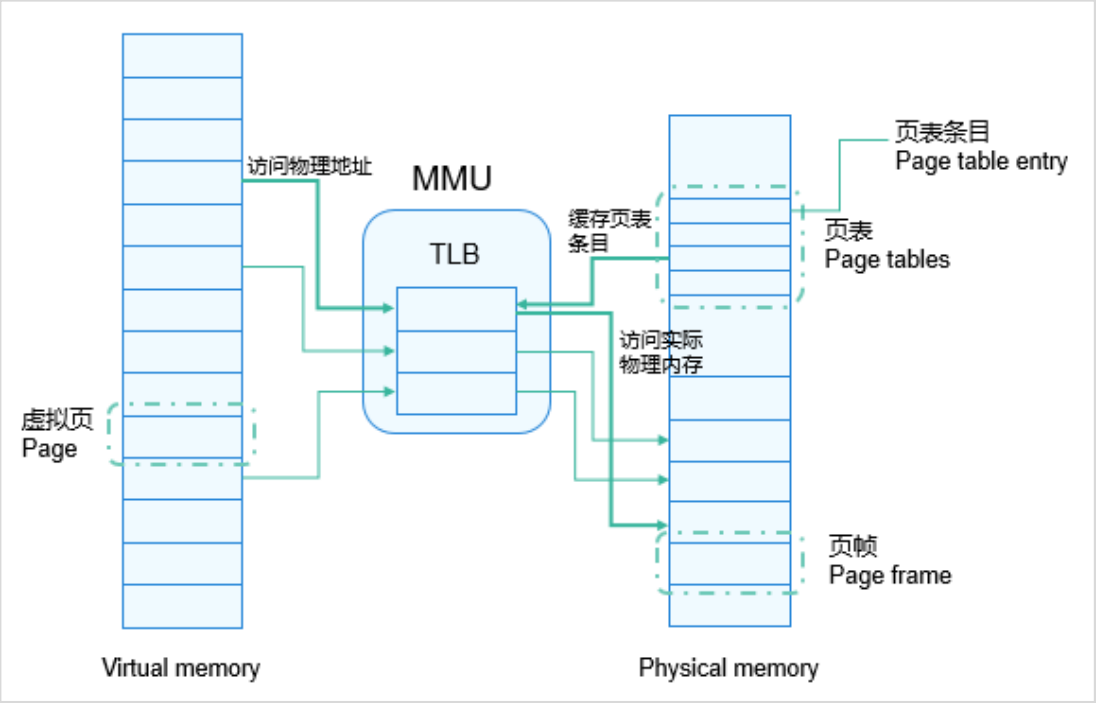

图3-17 CPU 访问内存示意图

3.9 内核通信机制

3.9.1 事件

事件（Event）是一种任务间通信的机制，可用于任务间的同步。多任务环境下，任务之间往往需要同步操作，一个等待即是一个同步。事件可以提供一对多、多对多的同步操作。一对多同步模型：一个任务等待多个事件的触发。可以是任意一个事件发生时唤醒任务处理事件，也可以是几个事件都发生后才唤醒任务处理事件。多对多同步模型：多个任务等待多个事件的触发。

任务通过创建事件控制块来触发事件或等待事件。事件间相互独立，内部实现为一个 32 位无符号整型，每一位标识一种事件类型。第 25 位不可用，因此最多可支持 31 种事件类型。事件仅用于任务间的同步，不提供数据传输功能。多次向事件控制块写入同一事件类型，在被清零前等效于只写入一次。多个任务可以对同一事件进行读写操作。支持事件读写超时机制。

事件运作原理：

事件初始化：会创建一个事件控制块，该控制块维护一个已处理的事件集合，以及等待特定事件的任务链表。

写事件：会向事件控制块写入指定的事件，事件控制块更新事件集合，并遍历任务链表，根据任务等待具体条件满足情况决定是否唤醒相关任务。

读事件：如果读取的事件已存在时，会直接同步返回。其他情况会根据超时时间以及事件触发情况，来决定返回时机：等待的事件条件在超时时间耗尽之前到达，阻塞任务会被直接唤醒，否则超时时间耗尽该任务才会被唤醒。

读事件条件满足与否取决于入参 eventMask 和 mode，eventMask 即需要关注的事件。mode 是具体处理方式，分以下三种情况：

LOS_WAITMODE_AND：表示 eventMask 中所有事件都发生时，才返回。

LOS_WAITMODE_OR：表示 eventMask 中任何事件发生时，就返回。

LOS_WAITMODE_CLR：事件读取成功后，对应读取到的事件会被清零。需要配合 LOS_WAITMODE_AND 或者 LOS_WAITMODE_OR 来使用。

事件清零：根据指定掩码，去对事件控制块的事件集合进行清零操作。当掩码为 0 时，表示将事件集合全部清零。当掩码为 0xffff 时，表示不清除任何事件，保持事件集合原状。

事件销毁：销毁指定的事件控制块。

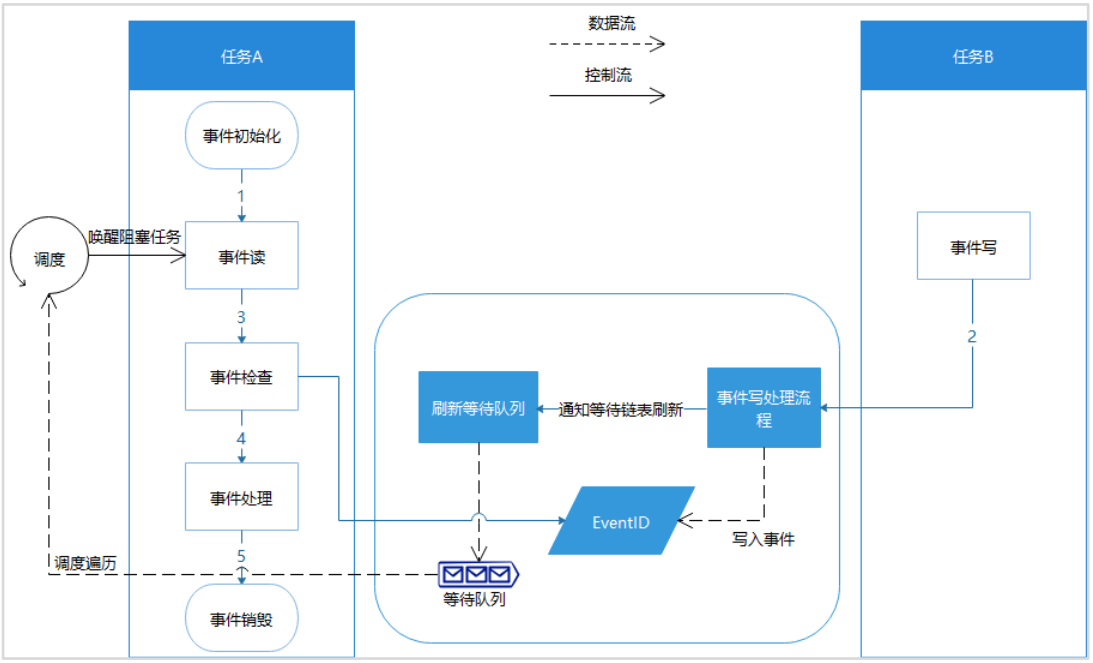

图3-18 事件运作原理图

3.9.2 信号量

信号量（Semaphore）是一种实现任务间通信的机制，可以实现任务间同步或共享资源的互斥访问。

一个信号量的数据结构中，通常有一个计数值，用于对有效资源数的计数，表示剩下的可被使用的共享资源数，其值的含义分两种情况：

0，表示该信号量当前不可获取，因此可能存在正在等待该信号量的任务。

正值，表示该信号量当前可被获取。

以同步为目的的信号量和以互斥为目的的信号量在使用上有如下不同：

用作互斥时，初始信号量计数值不为 0，表示可用的共享资源个数。在需要使用共享资源前，先获取信号量，然后使用一个共享资源，使用完毕后释放信号量。这样在共享资源被取完，即信号量计数减至 0 时，其他需要获取信号量的任务将被阻塞，从而保证了共享资源的互斥访问。另外，当共享资源数为 1 时，建议使用二值信号量，一种类似于互斥锁的机制。

用作同步时，初始信号量计数值为 0。任务 1 获取信号量而阻塞，直到任务 2 或者某中断释放信号量，任务 1 才得以进入 Ready 或 Running 态，从而达到了任务间的同步。

信号量允许多个任务在同一时刻访问共享资源，但会限制同一时刻访问此资源的最大任务数目。当访问资源的任务数达到该资源允许的最大数量时，会阻塞其他试图获取该资源的任务，直到有任务释放该信号量。

信号量初始化：初始化时为配置的 N 个信号量申请内存（N 值可以由用户自行配置，通过 LOSCFG_BASE_IPC_SEM_LIMIT 宏实现），并把所有信号量初始化成未使用，加入到未使用链表中供系统使用

信号量创建：从未使用的信号量链表中获取一个信号量，并设定初值。

信号量申请：若其计数器值大于 0，则直接减 1 返回成功。否则任务阻塞，等待其它任务释放该信号量，等待的超时时间可设定。当任务被一个信号量阻塞时，将该任务挂到信号量等待任务队列的队尾。

信号量释放：若没有任务等待该信号量，则直接将计数器加 1 返回。否则唤醒该信号量等待任务队列上的第一个任务。

信号量删除：将正在使用的信号量置为未使用信号量，并挂回到未使用链表。

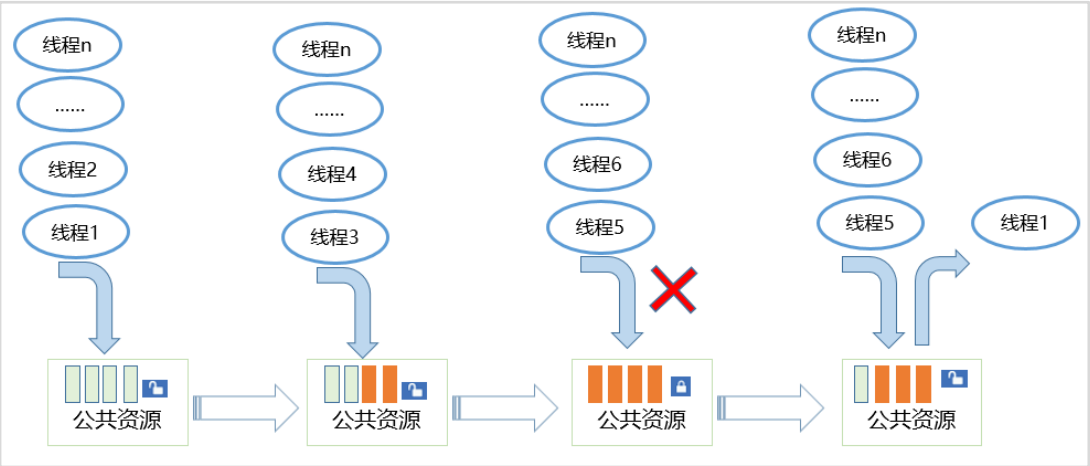

图3-19 信号量运作示意图

3.9.3 互斥锁

互斥锁又称互斥型信号量，用于实现对共享资源的独占式处理。当有任务持有时，这个任务获得该互斥锁的所有权。当该任务释放它时，任务失去该互斥锁的所有权。当一个任务持有互斥锁时，其他任务将不能再持有该互斥锁。多任务环境下往往存在多个任务竞争同一共享资源的应用场景，互斥锁可被用于对共享资源的保护从而实现独占式访问。

互斥量属性包含 3 个属性：协议属性、优先级上限属性和类型属性。协议属性用于处理不同优先级的任务申请互斥锁，协议属性包含如下三种：

- LOS_MUX_PRIO_NONE

不对申请互斥锁的任务的优先级进行继承或保护操作。

- LOS_MUX_PRIO_INHERIT

优先级继承属性，默认设置为该属性，对申请互斥锁的任务的优先级进行继承。在互斥锁设置为本协议属性情况下，申请互斥锁时，如果高优先级任务阻塞于互斥锁，则把持有互斥锁任务的优先级备份到任务控制块的优先级位图中，然后把任务优先级设置为和高优先级任务相同的优先级；持有互斥锁的任务释放互斥锁时，从任务控制块的优先级位图恢复任务优先级。

- LOS_MUX_PRIO_PROTECT

优先级保护属性，对申请互斥锁的任务的优先级进行保护。在互斥锁设置为本协议属性情况下，申请互斥锁时，如果任务优先级小于互斥锁优先级上限，则把任务优先级备份到任务控制块的优先级位图中，然后把任务优先级设置为互斥锁优先级上限属性值；释放互斥锁时，从任务控制块的优先级位图恢复任务优先级。

互斥锁的类型属性用于标记是否检测死锁，是否支持递归持有，类型属性包含如下三种：

- LOS_MUX_NORMAL

普通互斥锁，不会检测死锁。如果任务试图对一个互斥锁重复持有，将会引起这个线程的死锁。如果试图释放一个由别的任务持有的互斥锁，或者如果一个任务试图重复释放互斥锁都会引发不可预料的结果。

- LOS_MUX_RECURSIVE

递归互斥锁，默认设置为该属性。在互斥锁设置为本类型属性情况下，允许同一个任务对互斥锁进行多次持有锁，持有锁次数和释放锁次数相同，其他任务才能持有该互斥锁。如果试图持有已经被其他任务持有的互斥锁，或者如果试图释放已经被释放的互斥锁，会返回错误码。

- LOS_MUX_ERRORCHECK

错误检测互斥锁，会自动检测死锁。在互斥锁设置为本类型属性情况下，如果任务试图对一个互斥锁重复持有，或者试图释放一个由别的任务持有的互斥锁，或者如果一个任务试图释放已经被释放的互斥锁，都会返回错误码。

多任务环境下会存在多个任务访问同一公共资源的场景，而有些公共资源是非共享的，需要任务进行独占式处理。互斥锁怎样来避免这种冲突呢?

用互斥锁处理非共享资源的同步访问时，如果有任务访问该资源，则互斥锁为加锁状态。此时其他任务如果想访问这个公共资源则会被阻塞，直到互斥锁被持有该锁的任务释放后，其他任务才能重新访问该公共资源，此时互斥锁再次上锁，如此确保同一时刻只有一个任务正在访问这个公共资源，保证了公共资源操作的完整性。

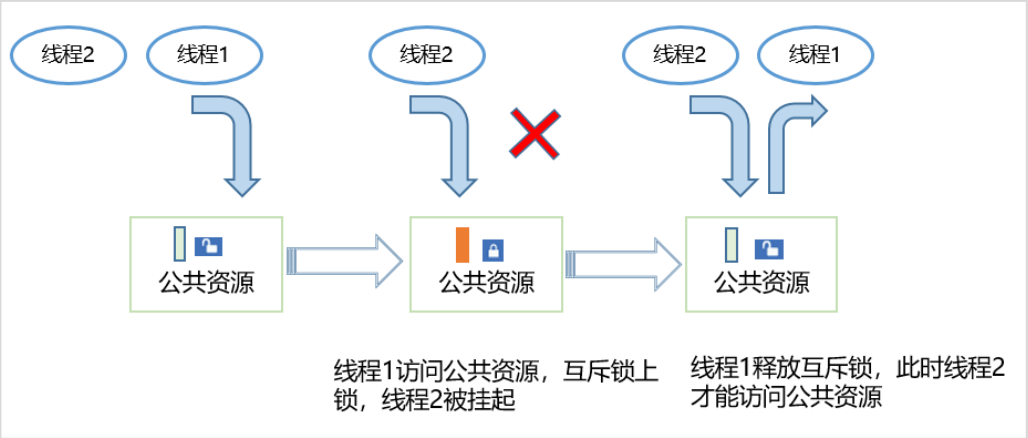

图3-20 互斥锁运作示意图

3.9.4 消息队列

队列又称消息队列，是一种常用于任务间通信的数据结构。队列接收来自任务或中断的不固定长度消息，并根据不同的接口确定传递的消息是否存放在队列空间中。

任务能够从队列里面读取消息，当队列中的消息为空时，挂起读取任务；当队列中有新消息时，挂起的读取任务被唤醒并处理新消息。任务也能够往队列里写入消息，当队列已经写满消息时，挂起写入任务；当队列中有空闲消息节点时，挂起的写入任务被唤醒并写入消息。

可以通过调整读队列和写队列的超时时间来调整读写接口的阻塞模式，如果将读队列和写队列的超时时间设置为 0，就不会挂起任务，接口会直接返回，这就是非阻塞模式。反之，如果将都队列和写队列的超时时间设置为大于 0 的时间，就会以阻塞模式运行。

消息队列提供了异步处理机制，允许将一个消息放入队列，但不立即处理。同时队列还有缓冲消息的作用，可以使用队列实现任务异步通信，队列具有如下特性：

消息以先进先出的方式排队，支持异步读写。

读队列和写队列都支持超时机制。

每读取一条消息，就会将该消息节点设置为空闲。

发送消息类型由通信双方约定，可以允许不同长度（不超过队列的消息节点大小）的消息。

一个任务能够从任意一个消息队列接收和发送消息。

多个任务能够从同一个消息队列接收和发送消息。

创建队列时所需的队列空间，接口内系统自行动态申请内存。

队列运作原理：

创建队列时，创建队列成功会返回队列 ID。

在队列控制块中维护着一个消息头节点位置 Head 和一个消息尾节点位置 Tail，用于表示当前队列中消息的存储情况。Head 表示队列中被占用的消息节点的起始位置。Tail 表示被占用的消息节点的结束位置，也是空闲消息节点的起始位置。队列刚创建时，Head 和 Tail 均指向队列起始位置。

写队列时，根据 readWriteableCnt[1]判断队列是否可以写入，不能对已满（readWriteableCnt[1]为 0）队列进行写操作。写队列支持两种写入方式：向队列尾节点写入，也可以向队列头节点写入。尾节点写入时，根据 Tail 找到起始空闲消息节点作为数据写入对象，如果 Tail 已经指向队列尾部则采用回卷方式。头节点写入时，将 Head 的前一个节点作为数据写入对象，如果 Head 指向队列起始位置则采用回卷方式。

读队列时，根据 readWriteableCnt[0]判断队列是否有消息需要读取，对全部空闲（readWriteableCnt[0]为 0）队列进行读操作会引起任务挂起。如果队列可以读取消息，则根据 Head 找到最先写入队列的消息节点进行读取。如果 Head 已经指向队列尾部则采用回卷方式。

删除队列时，根据队列 ID 找到对应队列，把队列状态置为未使用，把队列控制块置为初始状态，并释放队列所占内存。

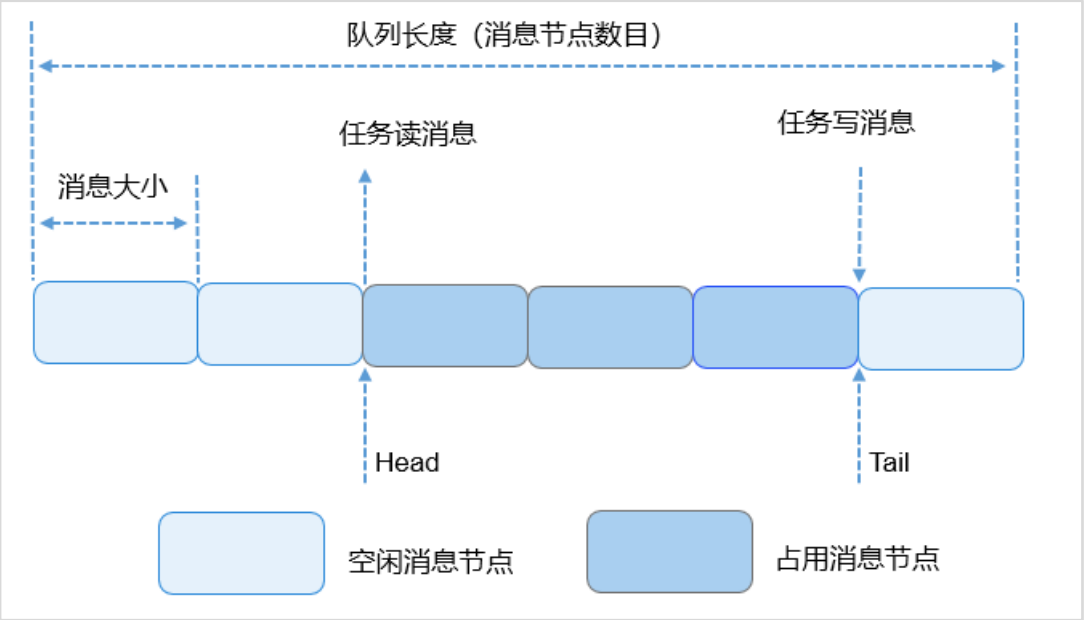

图3-21 队列读写数据操作示意图

上图对读写队列做了示意，图中只画了尾节点写入方式，没有画头节点写入，但是两者是类似的。

3.9.5 读写锁

读写锁与互斥锁类似，可用来同步同一进程中的各个任务，但与互斥锁不同的是，其允许多个读操作并发重入，而写操作互斥。

相对于互斥锁的开锁或闭锁状态，读写锁有三种状态：读模式下的锁，写模式下的锁，无锁。

读写锁的使用规则：

保护区无写模式下的锁，任何任务均可以为其增加读模式下的锁。

保护区处于无锁状态下，才可增加写模式下的锁。

多任务环境下往往存在多个任务访问同一共享资源的应用场景，读模式下的锁以共享状态对保护区访问，而写模式下的锁可被用于对共享资源的保护从而实现独占式访问。

这种共享-独占的方式非常适合多任务中读数据频率远大于写数据频率的应用中，提高应用多任务并发度。

相较于互斥锁，读写锁如何实现读模式下的锁及写模式下的锁来控制多任务的读写访问呢?

若 A 任务首次获取了写模式下的锁，有其他任务来获取或尝试获取读模式下的锁，均无法再上锁。

若 A 任务获取了读模式下的锁，当有任务来获取或尝试获取读模式下的锁时，读写锁计数均加一。

3.9.6 用户态快速互斥锁

Futex(Fast userspace mutex，用户态快速互斥锁)是内核提供的一种系统调用能力，通常作为基础组件与用户态的相关锁逻辑结合组成用户态锁，是一种用户态与内核态共同作用的锁，例如用户态 mutex 锁、barrier 与 cond 同步锁、读写锁。其用户态部分负责锁逻辑，内核态部分负责锁调度。

当用户态线程请求锁时，先在用户态进行锁状态的判断维护，若此时不产生锁的竞争，则直接在用户态进行上锁返回；反之，则需要进行线程的挂起操作，通过 Futex 系统调用请求内核介入来挂起线程，并维护阻塞队列。

当用户态线程释放锁时，先在用户态进行锁状态的判断维护，若此时没有其他线程被该锁阻塞，则直接在用户态进行解锁返回；反之，则需要进行阻塞线程的唤醒操作，通过 Futex 系统调用请求内核介入来唤醒阻塞队列中的线程。

当用户态产生锁的竞争或释放需要进行相关线程的调度操作时，会触发 Futex 系统调用进入内核，此时会将用户态锁的地址传入内核，并在内核的 Futex 中以锁地址来区分用户态的每一把锁，因为用户态可用虚拟地址空间为 1GiB，为了便于查找、管理，内核 Futex 采用哈希桶来存放用户态传入的锁。当前哈希桶共有 80 个，0~63 号桶用于存放私有锁（以虚拟地址进行哈希），64~79 号桶用于存放共享锁（以物理地址进行哈希），私有/共享属性通过用户态锁的初始化以及 Futex 系统调用入参确定。

3.9.7 信号

信号(signal)是一种常用的进程间异步通信机制，用软件的方式模拟中断信号，当一个进程需要传递信息给另一个进程时，则会发送一个信号给内核，再由内核将信号传递至指定进程，而指定进程不必进行等待信号的动作。

3.10 时间管理

时间管理以系统时钟为基础。时间管理提供给应用程序所有和时间有关的服务。系统时钟是由定时/计数器产生的输出脉冲触发中断而产生的，一般定义为整数或长整数。输出脉冲的周期叫做一个“时钟滴答”。系统时钟也称为时标或者 Tick。一个 Tick 的时长可以静态配置。用户是以秒、毫秒为单位计时，而操作系统时钟计时是以 Tick 为单位的，当用户需要对系统操作时，例如任务挂起、延时等，输入秒为单位的数值，此时需要时间管理模块对二者进行转换。

Tick 与秒之间的对应关系可以配置。

Cycle：系统最小的计时单位。Cycle 的时长由系统主频决定，系统主频就是每秒钟的 Cycle 数。

Tick：Tick 是操作系统的基本时间单位，对应的时长由系统主频及每秒 Tick 数决定，由用户配置。系统的时间管理模块提供时间转换、统计、延迟功能以满足用户对时间相关需求的实现。

3.11 软件定时器

软件定时器，是基于系统 Tick 时钟中断且由软件来模拟的定时器，当经过设定的 Tick 时钟计数值后会触发用户定义的回调函数。定时精度与系统 Tick 时钟的周期有关。硬件定时器受硬件的限制，数量上不足以满足用户的实际需求，因此为了满足用户需求，提供更多的定时器，Huawei LiteOS 操作系统提供软件定时器功能。软件定时器扩展了定时器的数量，允许创建更多的定时业务。

软件定时器功能上支持：

静态裁剪：能通过宏关闭软件定时器功能。

软件定时器创建。

软件定时器启动。

软件定时器停止。

软件定时器删除。

软件定时器剩余 Tick 数获取。

运行机制软件定时器是系统资源，在模块初始化的时候已经分配了一块连续的内存，系统支持的最大定时器个数由 los_config.h 中的 LOSCFG_BASE_CORE_SWTMR_LIMIT 宏配置。软件定时器使用了系统的一个队列和一个任务资源，软件定时器的触发遵循队列规则，先进先出。同一时刻设置的定时时间短的定时器总是比定时时间长的靠近队列头，满足优先被触发的准则。软件定时器以 Tick 为基本计时单位，当用户创建并启动一个软件定时器时，系统会根据当前系统 Tick 时间及用户设置的定时间隔确定该定时器的到期 Tick 时间，并将该定时器控制结构挂入计时全局链表。

当 Tick 中断到来时，在 Tick 中断处理函数中扫描软件定时器的计时全局链表，看是否有定时器超时，若有则将超时的定时器记录下来。

Tick 中断处理函数结束后，软件定时器任务（优先级为最高）被唤醒，在该任务中调用之前记录下来的定时器的超时回调函数。

定时器状态：

OS_SWTMR_STATUS_UNUSED（未使用）

系统在定时器模块初始化的时候将系统中所有定时器资源初始化成该状态。

OS_SWTMR_STATUS_CREATED（创建未启动/停止）

在未使用状态下调用 LOS_SwtmrCreate 接口或者启动后调用 LOS_SwtmrStop 接口后，定时器将变成该状态。

OS_SWTMR_STATUS_TICKING（计数）

在定时器创建后调用 LOS_SwtmrStart 接口，定时器将变成该状态，表示定时器运行时的状态。

定时器模式：

系统的软件定时器提供三类定时器机制：

第一类是单次触发定时器，这类定时器在启动后只会触发一次定时器事件，然后定时器自动删除。

第二类是周期触发定时器，这类定时器会周期性的触发定时器事件，直到用户手动停止定时器，否则将永远持续执行下去。

第三类也是单次触发定时器，但与第一类不同之处在于这类定时器超时后不会自动删除，需要调用定时器删除接口删除定时器。

3.12 原子操作

在支持多任务的操作系统中，修改一块内存区域的数据需要“读取-修改-写入”三个步骤。然而同一内存区域的数据可能同时被多个任务访问，如果在修改数据的过程中被其他任务打断，就会造成该操作的执行结果无法预知。

使用开关中断的方法固然可以保证多任务执行结果符合预期，但是显然这种方法会影响系统性能。

ARMv6 架构引入了 LDREX 和 STREX 指令，以支持对共享存储器更缜密的非阻塞同步。由此实现的原子操作能确保对同一数据的“读取-修改-写入”操作在它的执行期间不会被打断，即操作的原子性。

3.13 扩展组件

3.13.1 系统调用

OpenHarmony 的 LiteOS-A 实现了用户态与内核态的区分隔离，用户态程序不能直接访问内核资源，而系统调用则为用户态程序提供了一种访问内核资源、与内核进行交互的通道。

用户程序通过调用 System API（系统 API，通常是系统提供的 POSIX 接口）进行内核资源访问与交互请求，POSIX 接口内部会触发 SVC/SWI 异常，完成系统从用户态到内核态的切换，然后对接到内核的 Syscall Handler（系统调用统一处理接口）进行参数解析，最终分发至具体的内核处理函数。

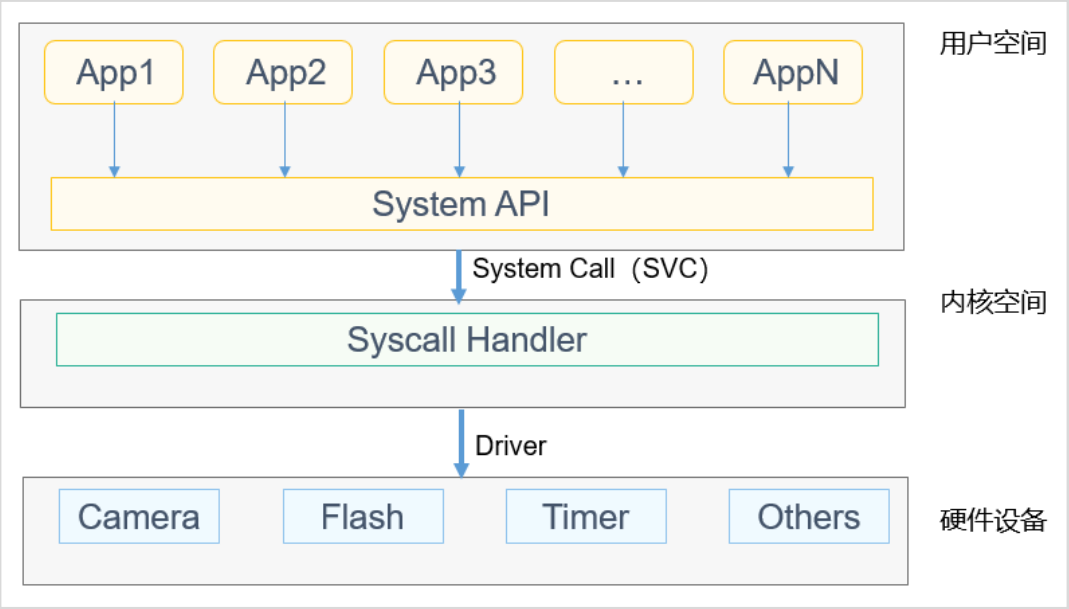

图3-22 系统调用原理图

Syscall Handler 的具体实现在 kernel/liteos_a/syscall/los_syscall.c 中 OsArmA32SyscallHandle 函数，在进入系统软中断异常时会调用此函数，并且按照 kernel/liteos_a/syscall/syscall_lookup.h 中的清单进行系统调用的入参解析，执行各系统调用最终对应的内核处理函数。

3.13.2 动态加载与链接

LiteOS-A 内核的动态加载与链接机制主要是由内核加载器以及动态链接器构成，内核加载器用于加载应用程序以及动态链接器，动态链接器用于加载应用程序所依赖的共享库，并对应用程序和共享库进行符号重定位。与静态链接相比，动态链接是将应用程序与动态库推迟到运行时再进行链接的一种机制。

动态链接的优势：

第一点优势，多个应用程序可以共享一份代码，最小加载单元为页，相对静态链接可以节约磁盘和内存空间；

第二点优势，共享库升级时，理论上将旧版本的共享库覆盖即可（共享库中的接口向下兼容），无需重新链接；

第三点优势，加载地址可以进行随机化处理，防止攻击，保证安全性。

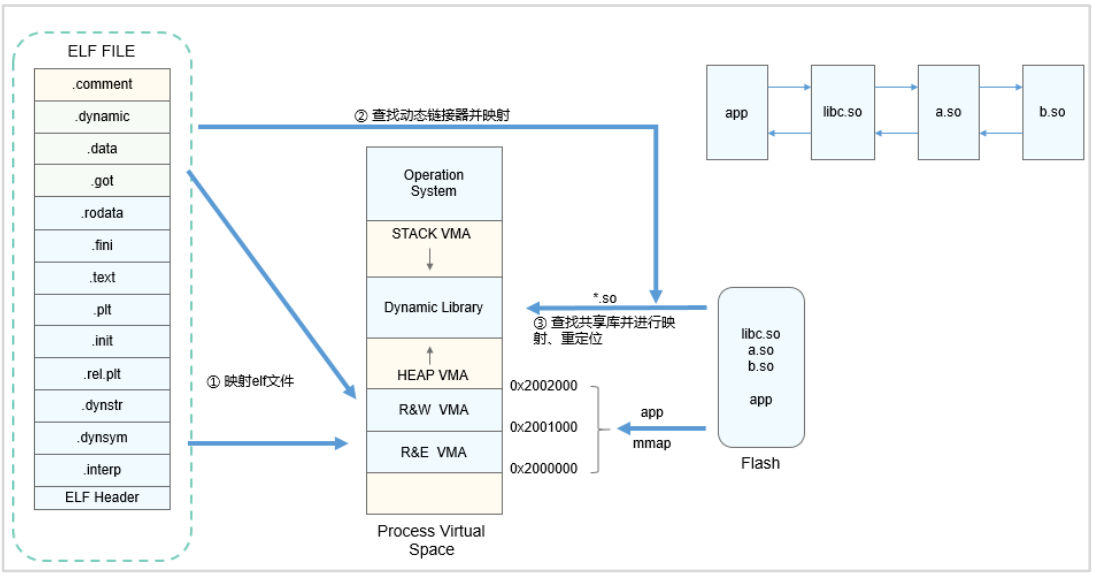

图3-23 动态链接原理图

内核将应用程序 ELF 文件的 PT_LOAD 段信息映射至进程空间。对于 ET_EXEC 类型的文件，根据 PT_LOAD 段中 p_vaddr 进行固定地址映射；对于 ET_DYN 类型（位置无关的可执行程序，通过编译选项“-fPIE”得到）的文件，内核通过 mmap 接口选择 base 基址进行映射（load_addr = base + p_vaddr）。

若应用程序是静态链接的（静态链接不支持编译选项“-fPIE”），设置堆栈信息后跳转至应用程序 ELF 文件中 e_entry 指定的地址并运行；若程序是动态链接的，应用程序 ELF 文件中会有 PT_INTERP 段，保存动态链接器的路径信息（ET_DYN 类型）。musl 的动态链接器是 libc-musl.so 的一部分，libc-musl.so 的入口即动态链接器的入口。内核通过 mmap 接口选择 base 基址进行映射，设置堆栈信息后跳转至 base + e_entry（该 e_entry 为动态链接器的入口）地址并运行动态链接器。

动态链接器自举并查找应用程序依赖的所有共享库并对导入符号进行重定位，最后跳转至应用程序的 e_entry（或 base + e_entry），开始运行应用程序。

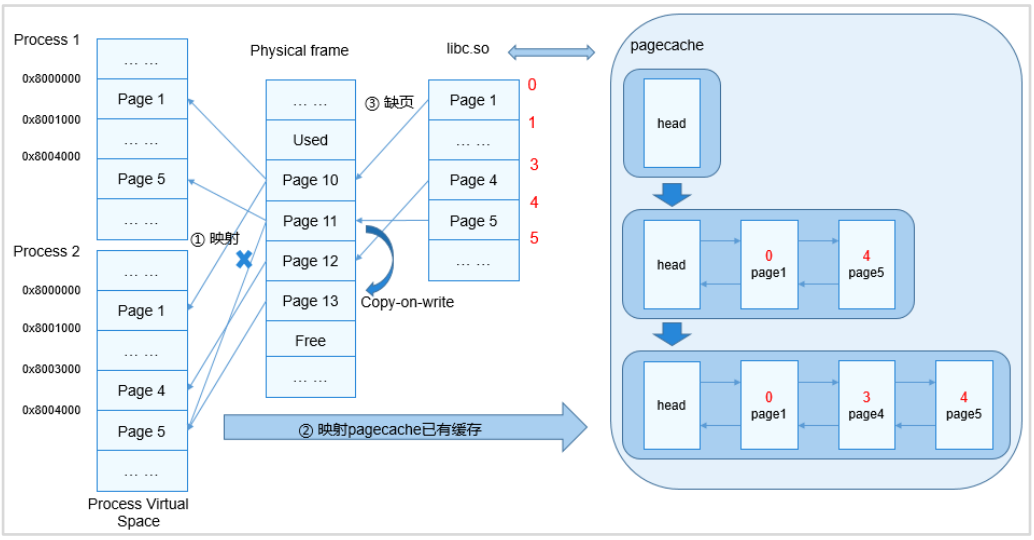

图3-24 动态链接器原理图

程序的执行过程如下：

1.加载器与链接器调用 mmap 映射 PT_LOAD 段；

2.内核调用 map_pages 接口查找并映射 pagecache 已有的缓存；

3.程序执行时，虚拟内存区间若无具体的物理内存做映射，系统将触发缺页中断，将 elf 文件内容读入物理内存，并将该内存块加入 pagecache；

4.将已读入文件内容的物理内存与虚拟地址区间做映射；

5.程序继续执行；

3.13.3 虚拟动态共享库

VDSO（Virtual Dynamic Shared Object，虚拟动态共享库）相对于普通的动态共享库，区别在于其 so 文件不保存在文件系统中，存在于系统镜像中，由内核在运行时确定并提供给应用程序，故称为虚拟动态共享库。

LiteOS-A 内核通过 VDSO 机制实现上层用户态程序可以快速读取内核相关数据的一种通道方法，可用于实现部分系统调用的加速，也可用于实现非系统敏感数据（硬件配置、软件配置）的快速读取。

VDSO 其核心思想就是内核看护一段内存，并将这段内存映射（只读）进用户态应用程序的地址空间，应用程序通过链接 vdso.so 后，将某些系统调用替换为直接读取这段已映射的内存从而避免系统调用达到加速的效果。

VDSO 总体可分为数据页与代码页两部分：数据页提供内核映射给用户进程的内核时数据；代码页提供屏蔽系统调用的主要逻辑。

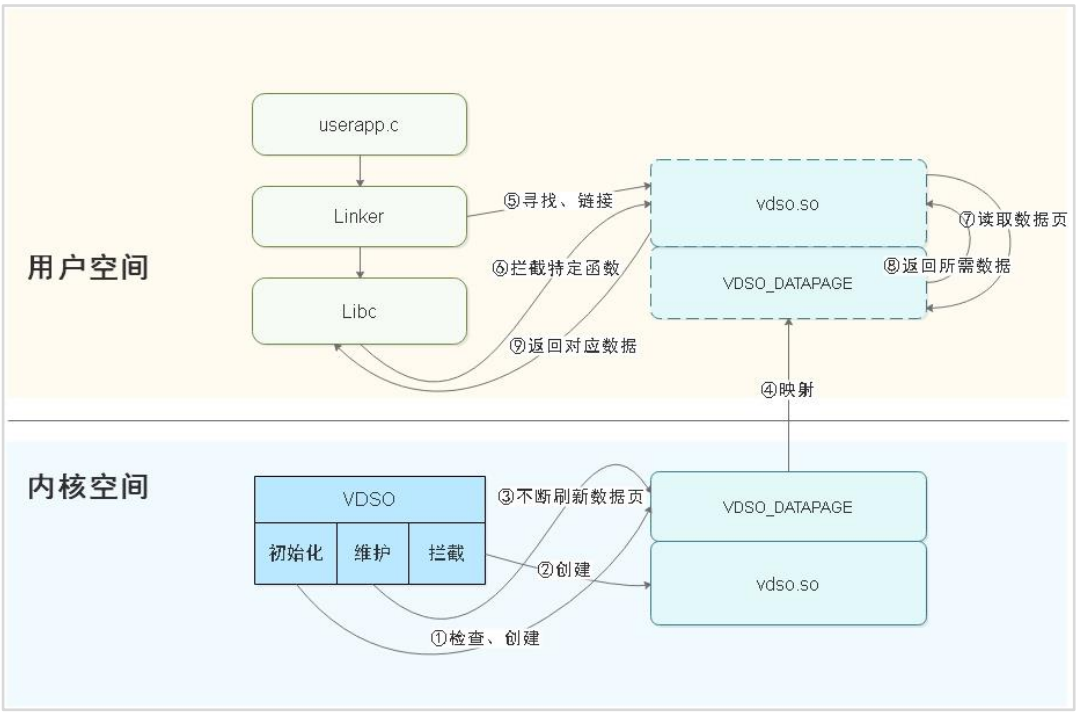

图3-25 虚拟动态共享库示意图

当前 VDSO 机制有以下几个主要步骤：

1.内核初始化时进行 VDSO 数据页的创建；

2.内核初始化时进行 VDSO 代码页的创建；

3.根据系统时钟中断不断将内核一些数据刷新进 VDSO 的数据页；

4.用户进程创建时将代码页映射进用户空间；

5.用户程序在动态链接时对 VDSO 的符号进行绑定；

6.当用户程序进行特定系统调用时（例如 clock_gettime(CLOCK_REALTIME_COARSE，&ts)），VDSO 代码页会将其拦截；

7.VDSO 代码页将正常系统调用转为直接读取映射好的 VDSO 数据页；

8.从 VDSO 数据页中将数据传回 VDSO 代码页；

9.将从 VDSO 数据页获取到的数据作为结果返回给用户程序。

当前 VDSO 机制支持 LibC 库 clock_gettime 接口的 CLOCK_REALTIME_COARSE 与 CLOCK_MONOTONIC_COARSE 功能，clock_gettime 接口的使用方法详见 POSIX 标准。用户调用 C 库接口 clock_gettime(CLOCK_REALTIME_COARSE, &ts)或者 clock_gettime(CLOCK_MONOTONIC_COARSE, &ts)即可使用 VDSO 机制。

使用 VDSO 机制得到的时间精度会与系统 tick 中断的精度保持一致，适用于对时间没有高精度要求且短时间内会高频触发 clock_gettime 或 gettimeofday 系统调用的场景，若有高精度要求，不建议采用 VDSO 机制。

3.13.4 轻量级进程间通信

LiteIPC 是 OpenHarmonyLiteOS-A 内核提供的一种新型 IPC（Inter-Process Communication，即进程间通信）机制，不同于传统的 System V IPC 机制，LiteIPC 主要是为 RPC（Remote Procedure Call，即远程过程调用）而设计的，而且是通过设备文件的方式对上层提供接口的，而非传统的 API 函数方式。

LiteIPC 中有两个主要概念，一个是 ServiceManager，另一个是 Service。整个系统只能有一个 ServiceManager，而 Service 可以有多个。ServiceManager 有两个主要功能：一是负责 Service 的注册和注销，二是负责管理 Service 的访问权限（只有有权限的任务（Task）可以向对应的 Service 发送 IPC 消息）。

首先将需要接收 IPC 消息的任务通过 ServiceManager 注册成为一个 Service，然后通过 ServiceManager 为该 Service 任务配置访问权限，即指定哪些任务可以向该 Service 任务发送 IPC 消息。LiteIPC 的核心思想就是在内核态为每个 Service 任务维护一个 IPC 消息队列，该消息队列通过 LiteIPC 设备文件向上层用户态程序分别提供代表收取 IPC 消息的读操作和代表发送 IPC 消息的写操作。

3.14 文件系统

文件系统（File System，或者简称 FS），是操作系统中输入输出的一种主要形式，主要负责和内外部的存储设备交互。文件系统对上通过 C 库提供的 POSIX 标准的操作接口，具体可以参考 C 库的 API 文档说明。对下，通过内核态的 VFS 虚拟层，屏蔽了各个具体文件系统的差异。基本架构如下：

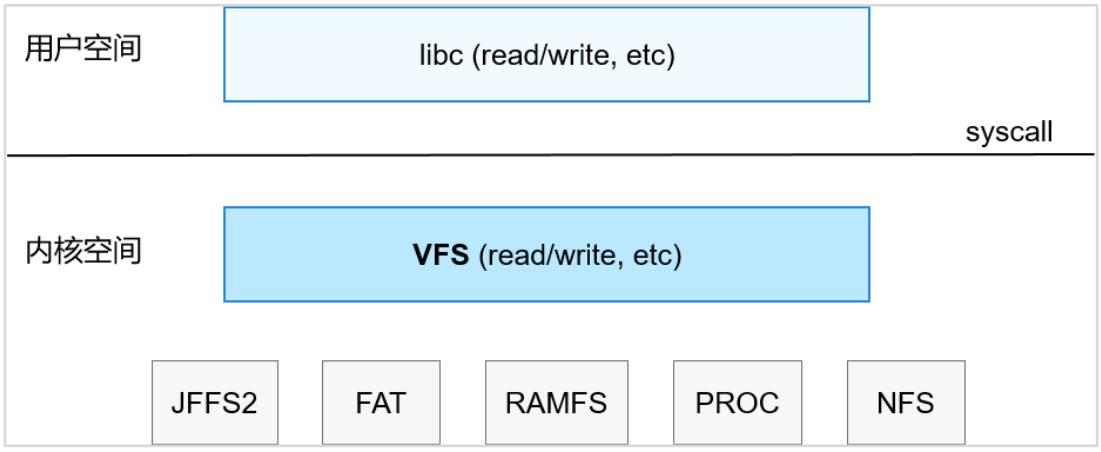

图3-26 文件系统示意图

3.14.2 虚拟文件系统

VFS（Virtual File System）是文件系统的虚拟层，它不是一个实际的文件系统，而是一个异构文件系统之上的软件粘合层，为用户提供统一的类 Unix 文件操作接口。由于不同类型的文件系统接口不统一，若系统中有多个文件系统类型，访问不同的文件系统就需要使用不同的非标准接口。而通过在系统中添加 VFS 层，提供统一的抽象接口，屏蔽了底层异构类型的文件

系统的差异，使得访问文件系统的系统调用不用关心底层的存储介质和文件系统类型，提高开发效率。

VFS 框架是通过在内存中的树结构来实现的，树的每个结点都是一个 Vnode 结构体，父子结点的关系以 PathCache 结构体保存。VFS 最主要的两个功能是：查找节点，统一调用（标准）。

当前，VFS 层主要通过函数指针，实现对不同文件系统类型调用不同接口实现标准接口功能；通过 Vnode 与 PathCache 机制，提升路径搜索以及文件访问的性能；通过挂载点管理进行分区管理；通过 FD 管理进行进程间 FD 隔离等。下面将对这些机制进行简要说明。

1、文件系统操作函数指针：VFS 层通过函数指针的形式，将统一调用按照不同的文件系统类型，分发到不同文件系统中进行底层操作。各文件系统的各自实现一套 Vnode 操作、挂载点操作以及文件操作接口，并以函数指针结构体的形式存储于对应 Vnode、挂载点、File 结构体中，实现 VFS 层对下访问。

2、Vnode：Vnode 是具体文件或目录在 VFS 层的抽象封装，它屏蔽了不同文件系统的差异，实现资源的统一管理。Vnode 节点主要有以下几种类型：

挂载点：挂载具体文件系统，如/、/storage

设备节点：/dev 目录下的节点，对应于一个设备，如/dev/mmcblk0

文件/目录节点：对应于具体文件系统中的文件/目录，如/bin/init

Vnode 通过哈希以及 LRU 机制进行管理。当系统启动后，对文件或目录的访问会优先从哈希链表中查找 Vnode 缓存，若缓存没有命中，则并从对应文件系统中搜索目标文件或目录，创建并缓存对应的 Vnode。当 Vnode 缓存数量达到上限时，将淘汰长时间未访问的 Vnode，其中挂载点 Vnode 与设备节点 Vnode 不参与淘汰。当前系统中 Vnode 的规格默认为 512，该规格可以通过 LOSCFG_MAX_VNODE_SIZE 进行配置。Vnode 数量过大，会造成较大的内存占用；Vnode 数量过少，则会造成搜索性能下降。

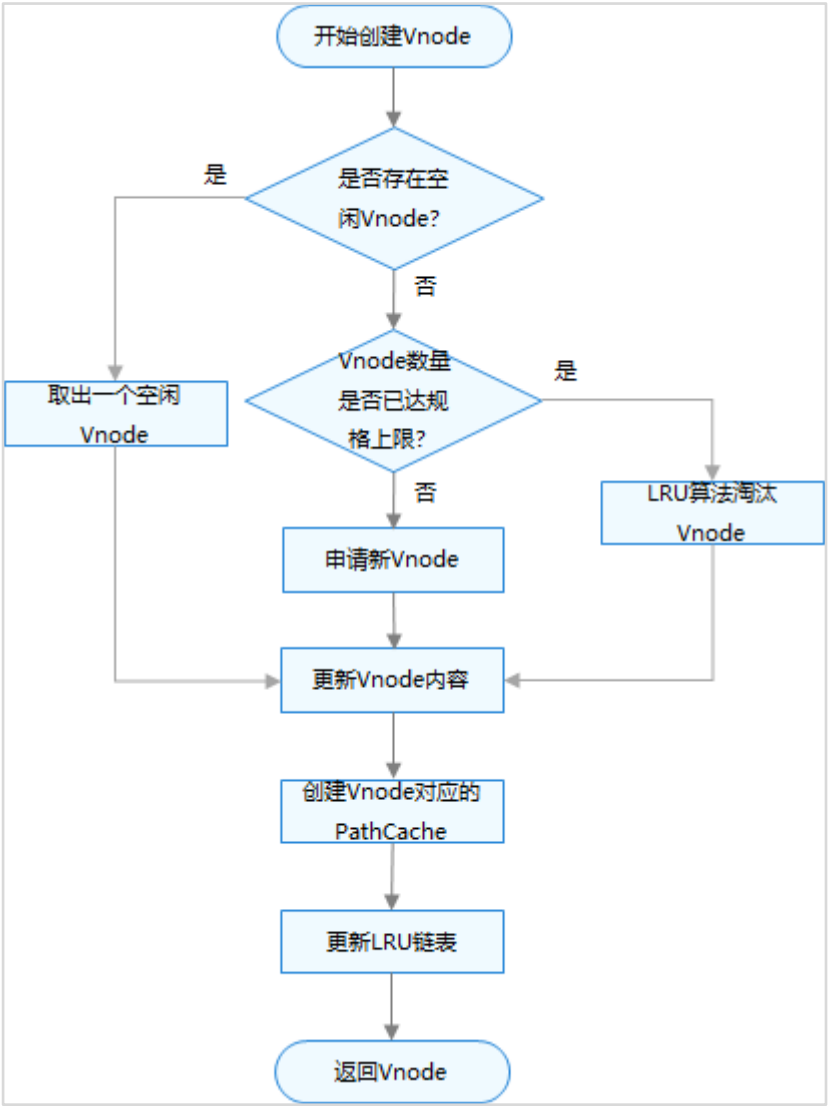

图3-27 Vnode 的创建流程示意图

3、PathCache 是路径缓存，与 Vnode 对应。PathCache 同样通过哈希链表存储，通过父 Vnode 中缓存的 PathCache 可以快速获取子 Vnode，加速路径查找。

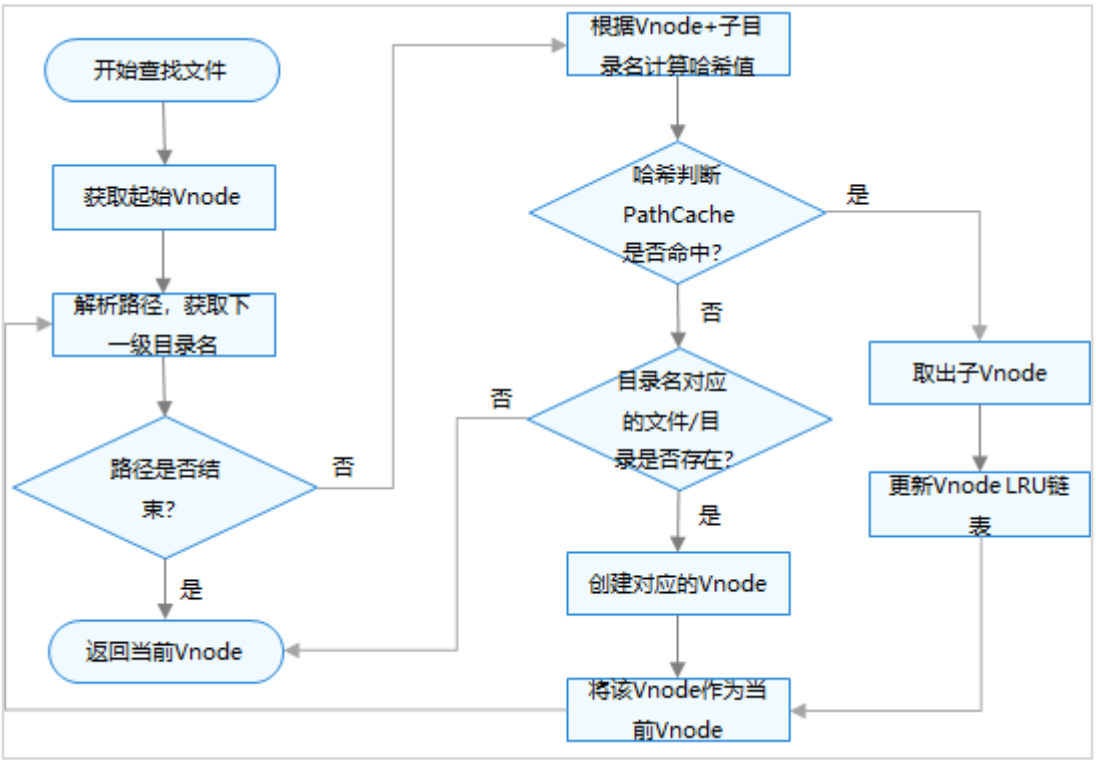

图3-28 文件/目录的查找流程示意图

4、PageCache 是文件级别的内核缓存。当前 PageCache 仅支持对二进制文件操作，在初次访问该文件时通过 mmap 映射到内存中，减少内核内存的占用，也大大提升了对同一个文件的读写操作速度。另外基于 PageCache 可实现以文件为基底的进程间通信。

5、Fd 管理：Fd（File Description）是描述一个打开的文件/目录的描述符。当前内核中，fd 总规格为 896，分为三种类型：

普通文件描述符，系统总规格为 512。

Socket 描述符，系统总规格为 128。

消息队列描述符，系统总规格为 256。

当前 LiteOS-A 内核中，对不同进程中的 fd 进行隔离，即进程只能访问本进程的 fd，所有进程的 fd 映射到全局 fd 表中进行统一分配管理。进程的文件描述符最多有 256 个。

6、挂载点管理：当前内核中，对系统中所有挂载点通过链表进行统一管理。挂载点结构体中，记录了该挂载分区内的所有 Vnode。当分区卸载时，会释放分区内的所有 Vnode。

3.14.3 支持的文件系统

3.14.3.1 FAT

FAT 文件系统是 File Allocation Table（文件配置表）的简称，主要包括 DBR 区、FAT 区、DATA 区三个区域。其中，FAT 区各个表项记录存储设备中对应簇的信息，包括簇是否被使用、文件下一个簇的编号、是否文件结尾等。FAT 文件系统有 FAT12、FAT16、FAT32 等多种格式，其中，12、16、32 表示对应格式中 FAT 表项的字节数，它们同时也限制了文件系统中的最大文件大小。FAT 文件系统支持多种介质，特别在可移动存储介质（U 盘、SD 卡、移

动硬盘等）上广泛使用，使嵌入式设备和 Windows、Linux 等桌面系统保持很好的兼容性，方便用户管理操作文件。

内核支持 FAT12、FAT16 与 FAT32 三种格式的 FAT 文件系统，具有代码量小、资源占用小、可裁切、支持多种物理介质等特性，并且与 Windows、Linux 等系统保持兼容，支持多设备、多分区识别等功能。支持硬盘多分区，可以在主分区以及逻辑分区上创建 FAT 文件系统。

LiteOS-A 内核通过 Bcache 提升 FAT 文件系统性能，Bcache 是 block cache 的简称。当发生读写时，Bcache 会缓存读写扇区附近的扇区，以减少 I/O 次数，提高性能。Bcache 的基本缓存单位为 block，每个 block 大小一致（默认有 28 个 block，每个 block 缓存 64 个扇区的数据）。当 Bcache 脏块率（脏扇区数/总扇区数）达到阈值时，会触发写回；如果脏块率未达到阈值，则不会将缓存数据写回磁盘。如果需要保证数据写回，开发者应当调用 sync 和 fsync 触发写回。FAT 文件系统的部分接口也会触发写回操作（如 close、umount 等），但开发者不应当基于这些接口触发写回。

3.14.3.2 JFFS2

JFFS2 是 Journalling Flash File System Version 2（日志文件系统）的缩写，是针对 MTD 设备的日志型文件系统。

内核的 JFFS2 主要应用于 NOR FLASH 闪存，其特点是：可读写、支持数据压缩、提供了崩溃/掉电安全保护、提供“写平衡”支持等。闪存与磁盘介质有许多差异，直接将磁盘文件系统运行在闪存设备上，会导致性能和安全问题。为解决这一问题，需要实现一个特别针对闪存的文件系统，JFFS2 就是这样一种文件系统。

这里仅列举几个对开发者和使用者会有一定影响的 JFFS2 的重要机制/特征：

1、Mount 机制及速度问题：按照 JFFS2 的设计，所有的文件会按照一定的规则，切分成大小不等的节点，依次存储到 flash 设备上。在 mount 流程中，需要获取到所有的这些节点信息并缓存到内存里。因此，mount 速度和 flash 设备的大小和文件数量的多少成线性比例关系。这是 JFFS2 的原生设计问题，对于 mount 速度非常介意的用户，可以在内核编译时开启“Enable JFFS2 SUMMARY”选项，可以极大提升 mount 的速度。这个选项的原理是将 mount 需要的信息提前存储到 flash 上，在 mount 时读取并解析这块内容，使得 mount 的速度变得相对恒定。这个实际是空间换时间的做法，会消耗 8%左右的额外空间。

2、写平衡的支持：由于 flash 设备的物理属性，读写都只能基于某个特定大小的“块”进行，为了防止某些特定的块磨损过于严重，在 JFFS2 中需要对写入的块进行“平衡”的管理，保证所有的块的写入次数都是相对平均的，进而保证 flash 设备的整体寿命。

3、GC(garbage collection)机制：在 JFFS2 里发生删除动作，实际的物理空间并不会立即释放，而是由独立的 GC 线程来做空间整理和搬移等 GC 动作，和所有的 GC 机制一样，在 JFFS2 里的 GC 会对瞬时的读写性能有一定影响。另外，为了有空间能被用来做空间整理，JFFS2 会对每个分区预留 3 块左右的空间，这个空间是用户不可见的。

4、压缩机制：当前使用的 JFFS2，底层会自动的在每次读/写时进行解压/压缩动作，实际 IO 的大小和用户请求读写的大小并不会一样。特别在写入时，不能通过写入大小来和 flash 剩余空间的大小来预估写入一定会成功或者失败。

5、硬链接机制：JFFS2 支持硬链接，底层实际占用的物理空间是一份，对于同一个文件的多个硬连接，并不会增加空间的占用；反之，只有当删除了所有的硬链接时，实际物理空间才会被释放。

3.14.3.3 NFS

NFS 是 Network File System（网络文件系统）的缩写。它最大的功能是可以通过网络，让不同的机器、不同的操作系统彼此分享其他用户的文件。因此，用户可以简单地将它看做是一个文件系统服务，在一定程度上相当于 Windows 环境下的共享文件夹。

LiteOS-A 内核的 NFS 文件系统指的是 NFS 的客户端，NFS 客户端能够将远程的 NFS 服务端分享的目录挂载到本地的机器中，运行程序和共享文件，但不占用当前系统的存储空间，在本地端的机器看起来，远程服务端的目录就好像是自己的一个磁盘一样。

3.14.3.4 RAMFS

RAMFS 是一个可动态调整大小的基于 RAM 的文件系统。RAMFS 没有后备存储源。向 RAMFS 中进行的文件写操作也会分配目录项和页缓存，但是数据并不写回到任何其他存储介质上，掉电后数据丢失。

RAMFS 文件系统把所有的文件都放在 RAM 中，所以读/写操作发生在 RAM 中，可以用 RAMFS 来存储一些临时性或经常要修改的数据，例如/tmp 和/var 目录，这样既避免了对存储器的读写损耗，也提高了数据读写速度。

3.14.3.5 Procfs

procfs 是进程文件系统的简称，是一种虚拟文件系统，他用文件的形式，展示进程或其他系统信息。相比调用接口的方式获取信息，以文件操作的方式获取系统信息更为方便。

LiteOS-A 内核中，procfs 在开机时会自动挂载到/proc 目录下，仅支持内核模块创建文件节点来提供查询服务。

procfs 文件的创建无法使用一般的文件系统接口，需要使用 ProcMkdir 接口创建目录，使用 CreateProcEntry 接口创建文件。文件节点功能的开发就是实现 read 和 write 函数的钩子挂到 CreateProcEntry 创建的文件中。当用户使用读写 procfs 的文件时，就会调用到钩子函数来实现自定义的功能。

4 HDF 驱动开发

4.1 驱动概述

硬件驱动框架 HDF（Hardware Driver Foundation）为驱动开发者提供驱动框架能力，包括驱动加载、驱动服务管理和驱动消息机制管理。旨在构建统一的驱动架构平台，为驱动开发者提供更精准、更高效的开发环境，力求做到一次开发，多系统部署。

HDF 驱动加载包括按需加载和按序加载。按需加载是 HDF 框架支持驱动在系统启动过程中默认加载，或者在系统启动之后动态加载；按序加载是 HDF 框架支持驱动在系统启动的过程中按照驱动的优先级进行加载。

驱动服务是 HDF 驱动设备对外提供能力的对象，由 HDF 框架统一管理。当驱动以接口的形式对外提供能力时，可以使用 HDF 框架的驱动服务管理能力。驱动服务管理主要包含驱动服务的发布和获取。HDF 框架定义了驱动对外发布服务的策略，是由配置文件中的 policy 字段来控制，policy 字段的取值范围以及含义如下：

```
typedef enum {
    /* 驱动不提供服务 */
    SERVICE_POLICY_NONE = 0,
    /* 驱动对内核态发布服务 */
    SERVICE_POLICY_PUBLIC = 1,
    /* 驱动对内核态和用户态都发布服务 */
    SERVICE_POLICY_CAPACITY = 2,
    /* 驱动服务不对外发布服务，但可以被订阅 */
    SERVICE_POLICY_FRIENDLY = 3,
    /* 驱动私有服务不对外发布服务，也不能被订阅 */
    SERVICE_POLICY_PRIVATE = 4,
    /* 错误的服务策略 */
    SERVICE_POLICY_INVALID
} ServicePolicy;
```

HDF 框架提供统一的驱动消息机制，支持用户态应用向内核态驱动发送消息，也支持内核态驱动向用户态应用发送消息。

4.2 驱动框架介绍

HDF 驱动框架有以下特点。弹性化的框架，组件化的驱动模型，规范化的驱动平台，归一化的平台底座，归一化的配置界面，驱动的动态安装。

HDF 驱动框架主要由驱动基础框架、驱动程序、驱动配置文件和驱动接口这四个部分组成。

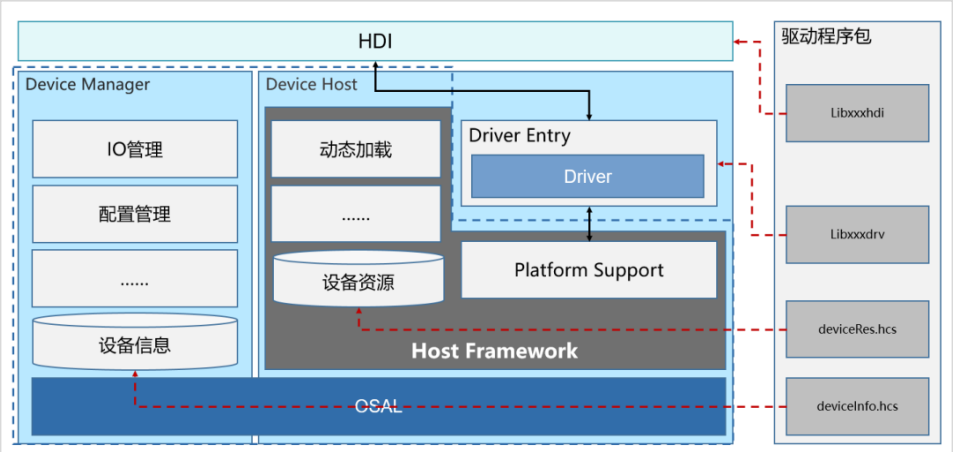

图4-1 HDF 驱动框架示意图

HDF 驱动基础框架提供统一的硬件资源管理，驱动加载管理以及设备节点管理等功能，由 Device Manager 和 Device Host 组成。

- Device Manager，负责根据设备信息（对应文件 deviceInfo.hcs）加载 Device Host，并控制 Device Host 完成 Driver 的加载。
- Device Host，对应一类硬件设备，负责加载运行 Driver。

Driver 驱动程序，实现驱动的具体功能。每个驱动程序都对应一个 Driver Entry。Driver Entry 主要完成驱动的初始化、驱动接口绑定和驱动卸载功能。

HCS（HDF Configuration Source）驱动配置文件，主要由设备信息（Device Information）和设备资源（Device Resource）组成。设备信息（对应文件 deviceInfo.hcs）完成设备信息的配置。如配置接口发布策略，驱动加载的方式等。设备资源（对应文件 deviceRes.hcs）完成设备资源的配置。如 GPIO 管脚、寄存器等资源信息的配置。

HDI（Hardware Device Interfaces），提供上层调用硬件功能的接口定义。外设代码需完成 HDI 接口的实现，具体的驱动实现不需要再重复定义 HDI 接口，只需要按照 HDI 接口实现即可接入系统功能，从而保证了上层调用者无需改动即可访问操作新适配的设备。

4.3 驱动模型介绍

HDF 框架以组件化的驱动模型作为核心设计思路，为开发者提供更精细化的驱动管理，让驱动开发和部署更加规范。HDF 框架将一类设备驱动放在同一个 Host 里面，开发者也可以将驱动功能分层独立开发和部署，支持一个驱动多个 Node。

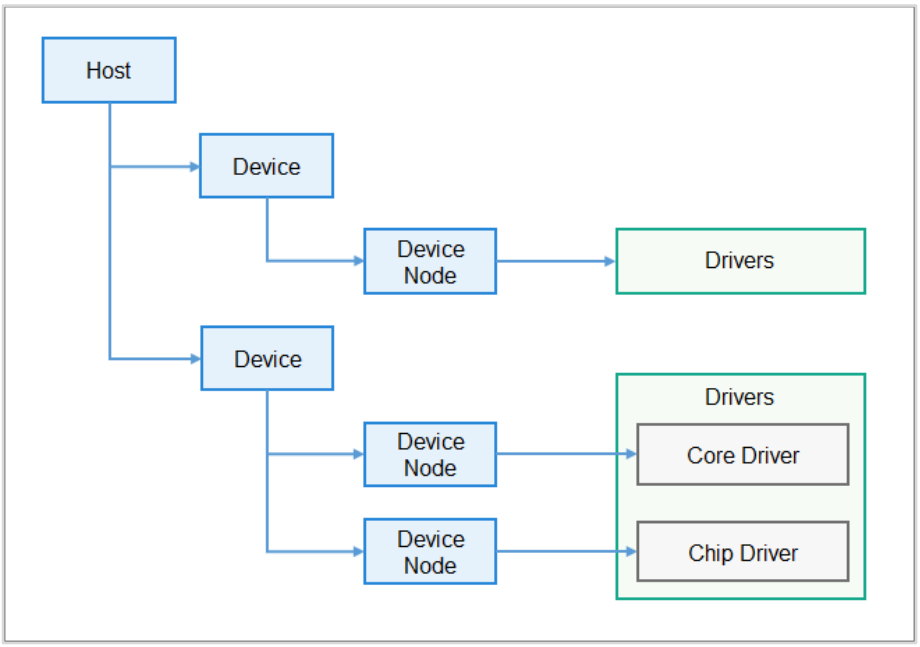

图4-2 驱动模型示意图

Host 属于一类设备聚合，如 Camera，Audio，Display 等，Device 是具体的设备，Device Node 就是具体设备的设备节点（类似于 linux 下/dev 下的设备节点）。Driver 是具体的驱动程序，分为 Core Driver（负责收集设备的信号或者状态，上报上层）和 Chip Driver（不同厂商的 IC 电路的驱动，类似于同一款摄像机可能使用不用厂商的 Sensor，就需要适配不同厂商的 Sensor）。

4.4 驱动实现步骤

以 LED 灯为例，掌握如何开发一个设备的 HDF 驱动，以及如何在应用层调用驱动。任务如下图所示。

图4-3 点亮 LED 灯任务示意图

4.4.2 驱动代码开发

步骤 1 确定目录结构。

在 device\st\drivers 路径下新建一个名为 led 的目录，并创建驱动文件 led.c 和编译构建文件 BUILD.gn。

```
.
└── device
    └── st
        └── drivers
            └── led
                ├── led.c
                └── BUILD.gn
```

步骤 2 实现 LED 驱动。

驱动实现包含驱动业务代码编写和驱动入口注册，在 led.c 文件中添加以下代码：

```
#include "hdf_device_desc.h"
#include "hdf_log.h"
#include "device_resource_if.h"
#include "osal_io.h"
#include "osal.h"
#include "osal_mem.h"
#include "gpio_if.h"

#define HDF_LOG_TAG led_driver // 打印日志所包含的标签，如果不定义则用默认定义的 HDF_TAG 标签
#define LED_WRITE_READ 1          // 读写操作码 1

enum LedOps {
    LED_OFF,
    LED_ON,
    LED_TOGGLE,
};

struct Stm32Mp1ILed {
    uint32_t gpioNum;
};
static struct Stm32Mp1ILed g_Stm32Mp1ILed;
uint8_t status = 0;
// Dispatch 是用来处理用户态发下来的消息
int32_t LedDriverDispatch(struct HdfDeviceIoClient *client, int cmdCode, struct HdfSBuf *data, struct HdfSBuf *reply)
{
    uint8_t contrl;
    HDF_LOGE("Led driver dispatch");
    if (client == NULL || client->device == NULL)
    {
        HDF_LOGE("Led driver device is NULL");
        return HDF_ERR_INVALID_OBJECT;
    }
```

```
    switch (cmdCode)
    {
    /* 接收到用户态发来的 LED_WRITE_READ 命令 */
    case LED_WRITE_READ:
        /* 读取 data 里的数据，赋值给 contrl */
        HdfSbufReadUint8(data,&contrl);
        switch (contrl)
        {
        /* 开灯 */
        case LED_ON:
            GpioWrite(g_Stm32Mp1ILed.gpioNum, GPIO_VAL_LOW);
            status = 1;
            break;
        /* 关灯 */
        case LED_OFF:
            GpioWrite(g_Stm32Mp1ILed.gpioNum, GPIO_VAL_HIGH);
            status = 0;
            break;
        /* 状态翻转 */
        case LED_TOGGLE:
            if(status == 0)
            {
                GpioWrite(g_Stm32Mp1ILed.gpioNum, GPIO_VAL_LOW);
                status = 1;
            }
            else
            {
                GpioWrite(g_Stm32Mp1ILed.gpioNum, GPIO_VAL_HIGH);
                status = 0;
            }
            break;
        default:
            break;
        }
        /* 把 LED 的状态值写入 reply，可被带至用户程序 */
        if (!HdfSbufWriteInt32(reply, status))
        {
            HDF_LOGE("replay is fail");
            return HDF_FAILURE;
        }
        break;
    default:
        break;
    }
    return HDF_SUCCESS;
}
```

```
// 读取驱动私有配置
static int32_t Stm32LedReadDrs(struct Stm32Mp1ILed *led, const struct DeviceResourceNode *node)
{
    int32_t ret;
    struct DeviceResourceIface *drsOps = NULL;

    drsOps = DeviceResourceGetIfaceInstance(HDF_CONFIG_SOURCE);
    if (drsOps == NULL || drsOps->GetUint32 == NULL) {
        HDF_LOGE("%s: invalid drs ops!", __func__);
        return HDF_FAILURE;
    }
    /* 读取 led.hcs 里面 led_gpio_num 的值 */
    ret = drsOps->GetUint32(node, "led_gpio_num", &led->gpioNum, 0);
    if (ret != HDF_SUCCESS) {
        HDF_LOGE("%s: read led gpio num fail!", __func__);
        return ret;
    }
    return HDF_SUCCESS;
}

//驱动对外提供的服务能力，将相关的服务接口绑定到 HDF 框架
int32_t HdfLedDriverBind(struct HdfDeviceObject *deviceObject)
{
    if (deviceObject == NULL)
    {
        HDF_LOGE("Led driver bind failed!");
        return HDF_ERR_INVALID_OBJECT;
    }
    static struct IDeviceIoService ledDriver = {
        .Dispatch = LedDriverDispatch,
    };
    deviceObject->service = (struct IDeviceIoService *)(&ledDriver);
    HDF_LOGD("Led driver bind success");
    return HDF_SUCCESS;
}

// 驱动自身业务初始的接口
int32_t HdfLedDriverInit(struct HdfDeviceObject *device)
{
    struct Stm32Mp1ILed *led = &g_Stm32Mp1ILed;
    int32_t ret;

    if (device == NULL || device->property == NULL) {
        HDF_LOGE("%s: device or property NULL!", __func__);
        return HDF_ERR_INVALID_OBJECT;
    }
    /* 读取 hcs 私有属性值 */
```

```
    ret = Stm32LedReadDrs(led, device->property);
    if (ret != HDF_SUCCESS) {
        HDF_LOGE("%s: get led device resource fail:%d", __func__, ret);
        return ret;
    }
    /* 将 GPIO 管脚配置为输出 */
    ret = GpioSetDir(led->gpioNum, GPIO_DIR_OUT);
    if (ret != 0)
    {
        HDF_LOGE("GpioSerDir: failed, ret %d\n", ret);
        return ret;
    }
    HDF_LOGD("Led driver Init success");
    return HDF_SUCCESS;
}

// 驱动资源释放的接口
void HdfLedDriverRelease(struct HdfDeviceObject *deviceObject)
{
    if (deviceObject == NULL)
    {
        HDF_LOGE("Led driver release failed!");
        return;
    }
    HDF_LOGD("Led driver release success");
    return;
}

// 定义驱动入口的对象，必须为 HdfDriverEntry（在 hdf_device_desc.h 中定义）类型的全局变量
struct HdfDriverEntry g_ledDriverEntry = {
    .moduleVersion = 1,
    .moduleName = "HDF_LED",
    .Bind = HdfLedDriverBind,
    .Init = HdfLedDriverInit,
    .Release = HdfLedDriverRelease,
};

// 调用 HDF_INIT 将驱动入口注册到 HDF 框架中
HDF_INIT(g_ledDriverEntry);
```

在 led/BUILD.gn 文件中添加以下代码。

```
import("//drivers/adapter/khdf/liteos/hdf.gni")

hdf_driver("hdf_led") {
    sources = [
    "led.c",
    ]
```

```
}
```

步骤 3 将 hdf_led 添加到可将业务构建成静态库的 BUILD.gn 文件中。

在/device/st/drivers/BUILD.gn 文件中添加以下代码，将 hdf_led 编译进内核。（"##start##"和"##end##"之间为新增配置"，##start##"和"##end##"仅用来标识位置，添加完配置后删除这两行）。

```
import("//drivers/adapter/khdf/liteos/hdf.gni")

group("drivers") {
deps = [
    "uart",
    "iwdg",
    "i2c",
    "gpio",
    ##start##
    "led",
    ##end##
    "stm32mp1xx_hal",
    "wifi/driver/hi3881",
    "wifi/driver:hdf_vendor_wifi",
]
}
```

步骤 4 驱动配置

驱动配置包含两部分，HDF 框架定义的驱动设备描述和驱动的私有配置信息。

- 在配置文件中添加 LED 设备描述

HDF 框架加载驱动所需要的信息来源于 HDF 框架定义的驱动设备描述，因此基于 HDF 框架开发的驱动必须要在 HDF 框架定义的 device_info.hcs 配置文件中添加对应的设备描述。本案例需要在 device\st\bearpi_hm_micro\liteos_a\hdf_config\device_info\device_info.hcs 中添加 LED 设备描述。（"##start##"和"##end##"之间为新增配置，"##start##"和"##end##"仅用来标识位置，添加完配置后删除这两行）。

```
platform :: host {
hostName = "platform_host";
priority = 50;
##start##
device_led :: device {                    // led 设备节点
    device0 :: deviceNode {               // led 驱动的 DeviceNode 节点
        policy = 2;                       // policy 字段是驱动服务发布的策略，在驱动服务管理章节有详细介绍
        priority = 10;                    // 驱动启动优先级（0-200），值越大优先级越低，建议默认配 100，优先级相同则不保证 device 的加载顺序
        preload = 1;                      // 驱动按需加载字段
        permission = 0777;                // 驱动创建设备节点权限
        moduleName = "HDF_LED"; // 驱动名称，该字段的值必须和驱动入口结构的 moduleName 值一致
```

```
            serviceName = "hdf_led";      // 驱动对外发布服务的名称，必须唯一
            deviceMatchAttr = "st_stm32mp157_led"; // 驱动私有数据匹配的关键字，必须和驱动私有数据配置表中的 match_attr 值相等
        }
    }
    ##end##
    device_gpio :: device {
        device0 :: deviceNode {
            policy = 0;
            priority = 10;
            permission = 0644;
            moduleName = "HDF_PLATFORM_GPIO";
            serviceName = "HDF_PLATFORM_GPIO";
            deviceMatchAttr = "st_stm32mp157_gpio";
        }
    }
```

- 配置驱动私有信息

如果驱动有私有配置，则可以添加一个驱动的配置文件，用来填写一些驱动的私有配置信息，HDF 框架在加载驱动的时候，会将对应的配置信息获取并保存在 HdfDeviceObject 中的 property 里面，通过 Bind 和 Init 传递给驱动。

在 device\st\bearpi_hm_micro\liteos_a\hdf_config 路径下新建 led 文件夹，并创建驱动配置文件 led_config.hcs。

```
.
└── device
    └── st
        └── bearpi_hm_micro
            └── liteos_a
                └── hdf_config
                    └── led
                        └── led_config.hcs
```

在 device\st\bearpi_hm_micro\liteos_a\hdf_config\led\led_config.hcs 中添加 LED 私有配置描述。

```
root {
    LedDriverConfig {
        led_gpio_num = 13;
        match_attr = "st_stm32mp157_led";    //该字段的值必须和 device_info.hcs 中的 deviceMatchAttr 值一致
    }
}
```

配置信息定义之后，需要将该配置文件添加到板级配置入口文件 device\st\bearpi_hm_micro\liteos_a\hdf_config\hdf.hcs。

```
#include "device_info/device_info.hcs"
#include "led/led_config.hcs"
```

4.4.3 业务代码开发

步骤 1 确定目录结构。

在./applications/BearPi/BearPi-HM_Micro/samples 路径下新建一个目录（或一套目录结构），用于存放业务源码文件。

例如：在 sample 下新增业务 my_led，其中 my_led.c 为业务代码，BUILD.gn 为编译脚本，具体规划目录结构如下：

```
.
└── applications
    └── BearPi
        └── BearPi-HM_Micro
            └── samples
                ├── my_led
                │   ├── my_led.c
                │   └── BUILD.gn
```

步骤 2 编写业务代码。

在 my_led.c 中添加以下业务代码：

```
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <stdio.h>
#include "hdf_sbuf.h"
#include "hdf_io_service_if.h"

#define LED_WRITE_READ 1
#define LED_SERVICE "hdf_led"

static int SendEvent(struct HdfIoService *serv, uint8_t eventData)
{
    int ret = 0;
    struct HdfSBuf *data = HdfSBufObtainDefaultSize();
    if (data == NULL)
    {
        printf("fail to obtain sbuf data!\r\n");
        return 1;
    }

    struct HdfSBuf *reply = HdfSBufObtainDefaultSize();
    if (reply == NULL)
    {
        printf("fail to obtain sbuf reply!\r\n");
        ret = HDF_DEV_ERR_NO_MEMORY;
        goto out;
```

```
    }
    /* 写入数据 */
    if (!HdfSbufWriteUint8(data, eventData))
    {
        printf("fail to write sbuf!\r\n");
        ret = HDF_FAILURE;
        goto out;
    }
    /* 通过 Dispatch 发送到驱动 */
    ret = serv->dispatcher->Dispatch(&serv->object, LED_WRITE_READ, data, reply);
    if (ret != HDF_SUCCESS)
    {
        printf("fail to send service call!\r\n");
        goto out;
    }

    int replyData = 0;
    /* 读取驱动的回复数据 */
    if (!HdfSbufReadInt32(reply, &replyData))
    {
        printf("fail to get service call reply!\r\n");
        ret = HDF_ERR_INVALID_OBJECT;
        goto out;
    }
    printf("\r\nGet reply is: %d\r\n", replyData);
out:
    HdfSBufRecycle(data);
    HdfSBufRecycle(reply);
    return ret;
}

int main(int argc, char **argv)
{
    int i;

    /* 获取服务 */
    struct HdfIoService *serv = HdfIoServiceBind(LED_SERVICE);
    if (serv == NULL)
    {
        printf("fail to get service %s!\r\n", LED_SERVICE);
        return HDF_FAILURE;
    }

    for (i=0; i < argc; i++)
    {
        printf("\r\nArgument %d is %s.\r\n", i, argv[i]);
    }
```

```
    SendEvent(serv, atoi(argv[1]));

    HdfIoServiceRecycle(serv);
    printf("exit");

    return HDF_SUCCESS;
}
```

步骤 3 编写构建业务的 BUILD.gn 文件。

BUILD.gn 文件由三部分内容（目标、源文件、头文件路径）构成，配置如下：

```
import("//build/lite/config/component/lite_component.gni")

HDF_FRAMEWORKS = "//drivers/framework"

executable("led_lib") {
    output_name = "my_led"
    sources = [
        "my_led.c",
    ]

    include_dirs = [
    "$HDF_FRAMEWORKS/ability/sbuf/include",
    "$HDF_FRAMEWORKS/core/shared/include",
    "$HDF_FRAMEWORKS/core/host/include",
    "$HDF_FRAMEWORKS/core/master/include",
    "$HDF_FRAMEWORKS/include/core",
    "$HDF_FRAMEWORKS/include/utils",
    "$HDF_FRAMEWORKS/utils/include",
    "$HDF_FRAMEWORKS/include/osal",
    "//drivers/adapter/uhdf/posix/include",
    "//third_party/bounds_checking_function/include",
    "//base/hiviewdfx/hilog_lite/interfaces/native/innerkits",
    ]

    deps = [
        "//base/hiviewdfx/hilog_lite/frameworks/featured:hilog_shared",
        "//drivers/adapter/uhdf/manager:hdf_core",
        "//drivers/adapter/uhdf/posix:hdf_posix_osal",
    ]
}

lite_component("my_led") {
    features = [
        ":led_lib",
    ]
}
```

首先导入 gni 组件，将源码 my_led_.c 编译成 led_lib 库文件，输出的可执行文件名称由 output_name 定义为 my_led，include_dirs 里面加入 my_led.c 里面需要用到的.h 的头文件路径，deps 里面加入所依赖的库。

然后将 led_lib 打包成 lite_component，命名为 my_led 组件。

步骤 4 添加新组件

修改文件 build/lite/components/applications.json，添加组件 my_sample 的配置，如下所示为 applications.json 文件片段，"##start##"和"##end##"之间为新增配置（"##start##"和"##end##"仅用来标识位置，添加完配置后删除这两行），可基于在 hello_world 基础上在 dirs 里添加 my_led 路径。在 targets 里面添加 my_led 目标项。

```
{
"components": [
    ##start##
    {
        "component": "my_sample",
        "description": "my samples",
        "optional": "true",
        "dirs": [
            "applications/BearPi/BearPi-HM_Micro/samples/my_first_app",
            "applications/BearPi/BearPi-HM_Micro/samples/my_led"
        ],
        "targets": [
            "//applications/BearPi/BearPi-HM_Micro/samples/my_first_app:my_app",
            "//applications/BearPi/BearPi-HM_Micro/samples/my_led:my_led"
        ],
        "rom": "",
        "ram": "",
        "output": [],
        "adapted_kernel": [ "liteos_a" ],
        "features": [],
        "deps": {
        "components": [],
        "third_party": [ ]
        }
    },
    ##end##
    {
    "component": "bearpi_sample_app",
    "description": "bearpi_hm_micro samples.",
    "optional": "true",
    "dirs": [
        "applications/BearPi/BearPi-HM_Micro/samples/launcher",
        "applications/BearPi/BearPi-HM_Micro/samples/setting"
    ],
    "targets": [
        "//applications/BearPi/BearPi-HM_Micro/samples/launcher:launcher_hap",
```

```
            "//applications/BearPi/BearPi-HM_Micro/samples/setting:setting_hap"
        ],
        "rom": "",
        "ram": "",
        "output": [
            "launcher.so",
            "setting.so"
        ],
        "adapted_kernel": [ "liteos_a","linux" ],
        "features": [],
        "deps": {
            "third_party": [
            "bounds_checking_function",
            "wpa_supplicant"
            ],
            "kernel_special": {},
            "board_special": {},
            "components": [
            "aafwk_lite",
            "appexecfwk_lite",
            "surface",
            "ui",
            "graphic_utils",
            "kv_store",
            "syspara_lite",
            "permission",
            "ipc_lite",
            "samgr_lite",
            "utils_base"
            ]
        }
},
```

步骤 5 修改单板配置文件

修改文件 vendor/bearpi/bearpi_hm_micro/config.json，新增 my_sample 组件的条目，如下所示代码片段为 applications 子系统配置，"##start##"和"##end##"之间为新增条目（"##start##"和"##end##"仅用来标识位置，添加完配置后删除这两行）：

```
{
"subsystem": "applications",
        "components": [
          { "component": "bearpi_sample_app", "features":[] },
        ##start##
          { "component": "my_sample", "features":[] },
        ##end##
          { "component": "bearpi_screensaver_app", "features":[] }
        ]
```

```
    },
```

4.4.4 运行结果

参考《HCIP-HarmonyOS Device Developer V1.0 实验环境搭建指南》进行示例代码编译、烧录后，在命令行输入以下指令可控制开发板的 LED 灯。

```
OHOS #
OHOS # ./bin/my_led 0
OHOS #
Argument 0 is bin/my_led.

Argument 1 is 0.

Get reply is: 0
exit01-01 00:01:06.784 19 43 E 02500/led_driver: Led driver dispatch

OHOS #
OHOS # ./bin/my_led 1
OHOS #
Argument 0 is bin/my_led.

Argument 1 is 1.

Get reply is: 1
exit01-01 00:01:08.833 20 43 E 02500/led_driver: Led driver dispatch

OHOS #
OHOS # ./bin/my_led 2
OHOS #
Argument 0 is bin/my_led.

Argument 1 is 2.

Get reply is: 0
exit01-01 00:01:11.391 21 43 E 02500/led_driver: Led driver dispatch
```

4.4.5 总结

device_info.hcs 文件中的 moduleName 必须要和驱动文件中的 moduleName 字段匹配，这样驱动才会加载起来。

device_info.hcs 文件中的 deviceMatchAttr 的字段必须和私有配置文件中 led_config.hcs 的 match_attr 的字段匹配，这样私有配置才能生效。

5 应用安装部署

5.1 用户应用程序

用户应用程序泛指运行在设备的操作系统之上，为用户提供特定服务的程序，简称“应用”。

在系统上运行的应用，有两种形态：

1. 传统方式的需要安装的应用。
2. 提供特定功能，免安装的应用（即原子化服务）。

在文档中，如无特殊说明，“应用”所指代的对象包括上述两种形态。

5.2 用户应用程序包结构

Ability 是系统的关键组件和编程抽象。一个应用可以包含一个或者多个 Ability。Ability 有多个具体子类型，每种子类型提供不同的能力。

用户应用程序包以 APP Pack（Application Package）形式发布，它是由一个或多个 HAP 以及描述每个 HAP 属性的 pack.info 组成。HAP 是 Ability 的部署包，应用代码围绕 Ability 组件展开。

一个 HAP 是由代码、资源、第三方库及应用配置文件组成的模块包，可分为 entry 和 feature 两种模块类型，如下图所示：

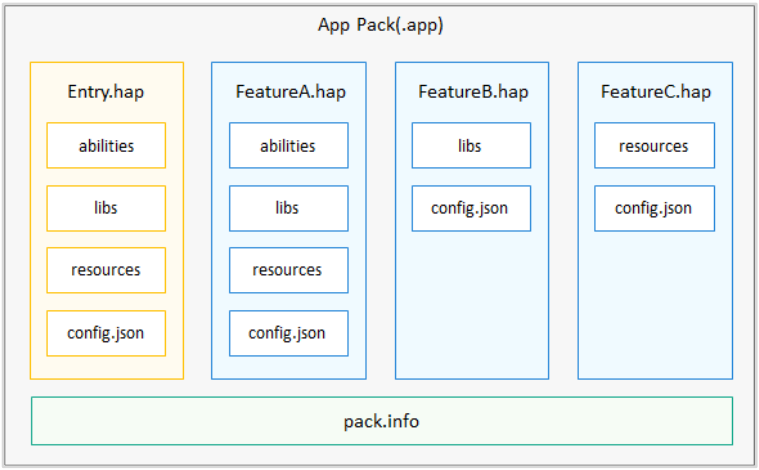

图5-1 HAP 组成结构示意图

- entry：应用的主模块。一个 APP 中，对于同一设备类型，可以有一个或多个 entry 类型的 HAP，来支持该设备类型中不同规格（如 API 版本、屏幕规格等）的具体设备。如果同一设备类型存在多个 entry 模块，则必须配置 distroFilter 分发规则，使得应用市场在做应用的云端分发时，对该设备类型下不同规格的设备进行精确分发。
- feature：应用的动态特性模块。一个 APP 可以包含一个或多个 feature 类型的 HAP，也可以不含。只有包含 Ability 的 HAP 才能够独立运行。

5.3 准备工作

- 源码获取（gitee 路径：https://gitee.com/bearpi/bearpi-hm_micro_small）
- HAP 包
- 开发板
- 准备一张 SD 卡（需要格式化为 FAT32）
- 一个读卡器

受限于目前开发板系统提供的挂载方式，SD 卡要格式化为 FAT32，而不是其他格式（比如可以存放更大容量的 NTFS 格式）。

5.4 安装 HAP 包

步骤 1 将源码路径下 applications/BearPi/BearPi-HM_Micro/tools/hap_tools/hap_example 中的 bm、LED_1.0.0.hap 拷贝到 SD 卡中。

步骤 2 将 SD 卡插入到开发板中，并按开发板的 RESET 按键重启开发板。

步骤 3 输入以下命令，挂载 SD 卡。

```
mount /dev/mmcblk0p0 /sdcard vfat
```

步骤 4 输入以下命令，打开调试模式。

```
./bm set -s disable
./bm set -d enable
```

步骤 5 安装应用。

```
./bm install -p LED_1.0.0.hap
```

命令行返回 SUCCESS 字样，则表示应用安装成功。

5.5 运行结果

屏幕出现名为 led 的应用程序。

6 工程调测

6.1 工程调测概述

软件项目的正式发布不是轻而易举的，需要经过一系列对软件的正确性、完整性、安全性的考验，遇到软件故障，我们需要定位故障并解决，故而解决问题的多种调测工具或调测手段就显得极为重要。

6.2 内核调测

6.2.1 TRACE 调测

Trace 调测旨在帮助开发者获取内核的运行流程，各个模块、任务的执行顺序，从而可以辅助开发者定位一些时序问题或者了解内核的代码运行过程。

内核提供一套 Hook 框架，将 Hook 点预埋在各个模块的主要流程中，开发者通过注册的形式在自己所需的 Hook 点上注册回调函数，当内核运行至对应流程中会由内核主动调用 Hook 函数，将当前流程的关键数据传递给开发者。

当系统触发到一个 Hook 点时，Trace 模块会对输入信息进行封装，添加 Trace 帧头信息，包含事件类型、运行的 cpuid、运行的任务 id、运行的相对时间戳等信息。

Trace 提供 2 种工作模式，离线模式和在线模式。

离线模式会将 trace frame 记录到预先申请好的循环 buffer 中。如果循环 buffer 记录的 frame 过多则可能出现翻转，会覆盖之前的记录，故保持记录的信息始终是最新的信息。Trace 循环 buffer 的数据可以通过 shell 命令导出进行详细分析，导出信息已按照时间戳信息完成排序。

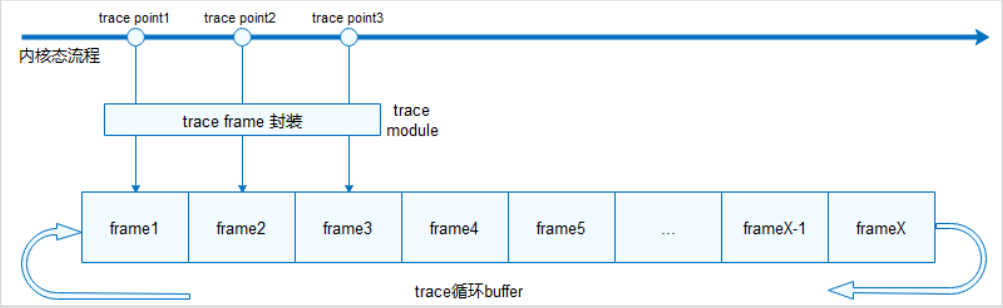

图6-1 Trace 离线模式运行原理图

在线模式需要配合 IDE 使用，实时将 trace frame 记录发送给 IDE，IDE 端进行解析并可视化展示。

内核态下使用 TRACE 模块，有下面几种功能。

功能分类	接口名	描述
启停控制	LOS_TraceStart	启动Trace
	LOS_TraceStop	停止Trace
操作Trace记录的数据	LOS_TraceRecordDump	输出Trace缓冲区数据
	LOS_TraceRecordGet	获取Trace缓冲区的首地址
	LOS_TraceReset	清除Trace缓冲区中的事件
过滤Trace记录的数据	LOS_TraceEventMaskSet	设置事件掩码，仅记录某些模块的事件
屏蔽某些中断号事件	LOS_TraceHwiFilterHookReg	注册过滤特定中断号事件的钩子函数
插桩函数	LOS_TRACE_EASY	简易插桩
	LOS_TRACE	标准插桩

用户态使用 Trace 模块是新增 trace 字符设备，位于"/dev/trace",通过对设备节点的 read、write、ioctl，实现用户态 trace 的读写和控制。

read：用户态读取 Trace 记录数据；

write：用户态事件写入

ioctl：用户态 Trace 控制操作，包括

```
#define TRACE_IOC_MAGIC     'T'
#define TRACE_START         _IO(TRACE_IOC_MAGIC, 1)
#define TRACE_STOP          _IO(TRACE_IOC_MAGIC, 2)
#define TRACE_RESET         _IO(TRACE_IOC_MAGIC, 3)
#define TRACE_DUMP              _IO(TRACE_IOC_MAGIC, 4)
#define TRACE_SET_MASK      _IO(TRACE_IOC_MAGIC, 5)
```

分别对应 Trace 启动(LOS_TraceStart)、停止(LOS_TraceStop)、清除记录(LOS_TraceReset)、dump 记录(LOS_TraceRecordDump)、设置事件过滤掩码(LOS_TraceEventMaskSet)。

6.2.2 内存信息统计

内存信息包括内存池大小、内存使用量、剩余内存大小、最大空闲内存、内存水线、内存节点数统计、碎片率等。

- 内存水线：即内存池的最大使用量，每次申请和释放时，都会更新水线值，实际业务可根据该值，优化内存池大小；
- 碎片率：衡量内存池的碎片化程度，碎片率高表现为内存池剩余内存很多，但是最大空闲内存块很小，可以用公式（fragment=100-100*最大空闲内存块大小/剩余内存大小）来度量；
- 其他参数：通过调用接口，扫描内存池的节点信息，统计出相关信息。

功能配置比如获取内存水线，需要打开 LOSCFG_MEM_WATERLINE 配置。

6.2.3 内存泄漏检测

内存泄漏（Memory Leak）是指程序中已动态分配的堆内存由于某种原因程序未释放或无法释放，造成系统内存的浪费，导致程序运行速度减慢甚至系统崩溃等严重后果。

内存泄漏缺陷具有隐蔽性、积累性的特征，比其他内存非法访问错误更难检测。因为内存泄漏的产生原因是内存块未被释放，属于遗漏型缺陷而不是过错型缺陷。此外，内存泄漏通常不会直接产生可观察的错误症状，而是逐渐积累，降低系统整体性能，极端的情况下可能使系统崩溃。

随着计算机应用需求的日益增加，应用程序的设计与开发也相应的日趋复杂，开发人员在程序实现的过程中处理的变量也大量增加，如何有效进行内存分配和释放，防止内存泄漏的问题变得越来越突出。例如服务器应用软件，需要长时间的运行，不断的处理由客户端发来的请求，如果没有有效的内存管理，每处理一次请求信息就有一定的内存泄漏。这样不仅影响到服务器的性能，还可能造成整个系统的崩溃。

开发人员进行程序开发的过程使用动态存储变量时，不可避免地面对内存管理的问题。程序中动态分配的存储空间，在程序执行完毕后需要进行释放。没有释放动态分配的存储空间而造成内存泄漏，是使用动态存储变量的主要问题。一般情况下，开发人员使用系统提供的内存管理基本函数，如 malloc、realloc、calloc、free 等，完成动态存储变量存储空间的分配和释放。

内存泄漏检测机制作为内核的可选功能，用于辅助定位动态内存泄漏问题。开启该功能，动态内存机制会自动记录申请内存时的函数调用关系。如果出现泄漏，就可以利用这些记录的信息，找到内存申请的地方，方便进一步确认。

6.2.4 踩内存检测

踩内存是访问了不应该访问的内存地址，比如过说数组越界，指针未初始化。

踩内存检测机制作为内核的可选功能，用于检测动态内存池的完整性。通过该机制，可以及时发现内存池是否发生了踩内存问题，并给出错误信息，便于及时发现系统问题，提高问题解决效率，降低问题定位成本。

若需要打开该功能，需要打开 LOSCFG_BASE_MEM_NODE_INTEGRITY_CHECK 配置。开启这个功能，每次申请内存，会实时检测内存池的完整性。如果不开启该功能，也可以调用 LOS_MemIntegrityCheck 接口检测，但是每次申请内存时，不会实时检测内存完整性，而且由于节点头没有魔鬼数字（开启时才有，省内存），检测的准确性也会相应降低，但对于系统的性能没有影响，故根据实际情况开关该功能。通过调用 LOS_MemIntegrityCheck 接口检测

内存池是否发生了踩内存，如果没有踩内存问题，那么接口返回 0 且没有 log 输出；如果存在踩内存问题，那么会输出相关 log。

6.3 性能分析

6.3.1 环境准备

搭建 Windows+Linux 混合开发环境。

参考 1.4.3 节《设备模拟器运行实验》中搭建 Windows+Ubuntu 混合开发环境。

6.3.2 栈分析

分析工具是基于静态二进制分析手段，提供任务栈开销估算值和函数调用关系图示，为栈内存使用、分析、优化、问题定位等开发场景提供较为准确的静态内存分析数据参考。

步骤 1 源码编译

在 DevEco Device Tool 的 Quick Access >主界面 > 工程中，点击导入工程。

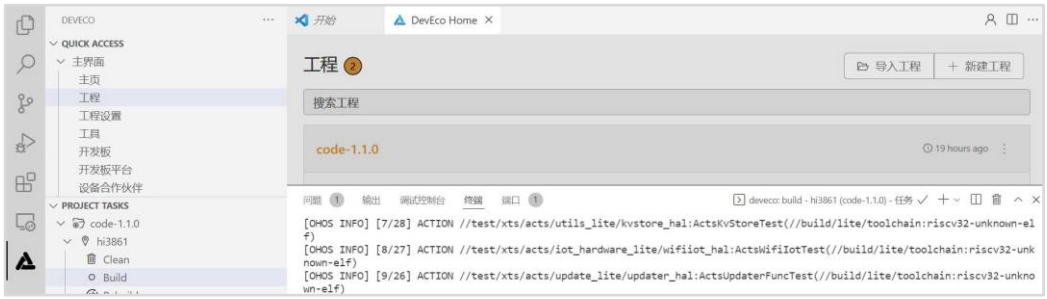

图6-2 源码编译示意图

打开 DevEco Device Tool 界面，在“PROJECT TASKS”中，点击对应开发板下的 build 按钮，执行编译。

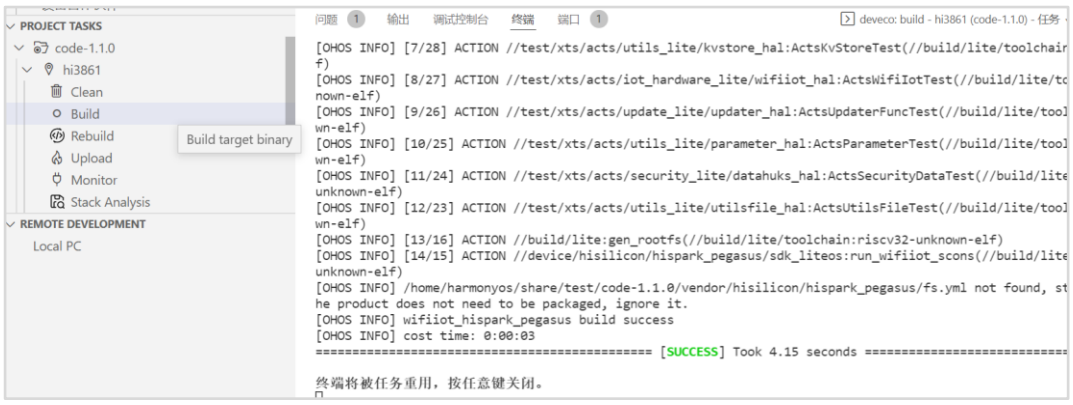

图6-3 执行编译示意图

步骤 2 点击 Stack Analysis，启动栈分析工具

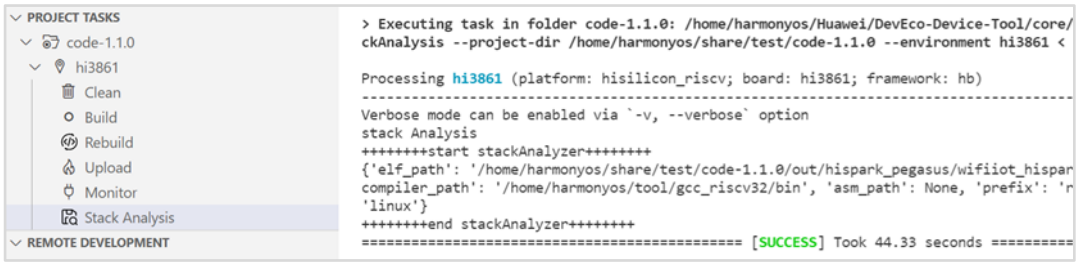

图6-4 栈分析工具示意图

步骤 3 分析结果展示

功能列表显示每个函数的函数名称和函数内部栈开销；调用图显示每个函数的调用关系，包括函数名称、调用深度、函数最大栈开销。

图6-5 分析结果展示示意图

6.3.3 镜像分析

镜像分析工具对工程构建出的 elf 文件进行内存占用分析。

步骤 1 源码编译

步骤 2 点击 Image Analysis，启动镜像分析工具

图6-6 镜像分析工具示意图

步骤 3 分析结果展示

镜像分析结果按照内存区域、详细信息、文件大小和模块大小 4 个界面进行展示。

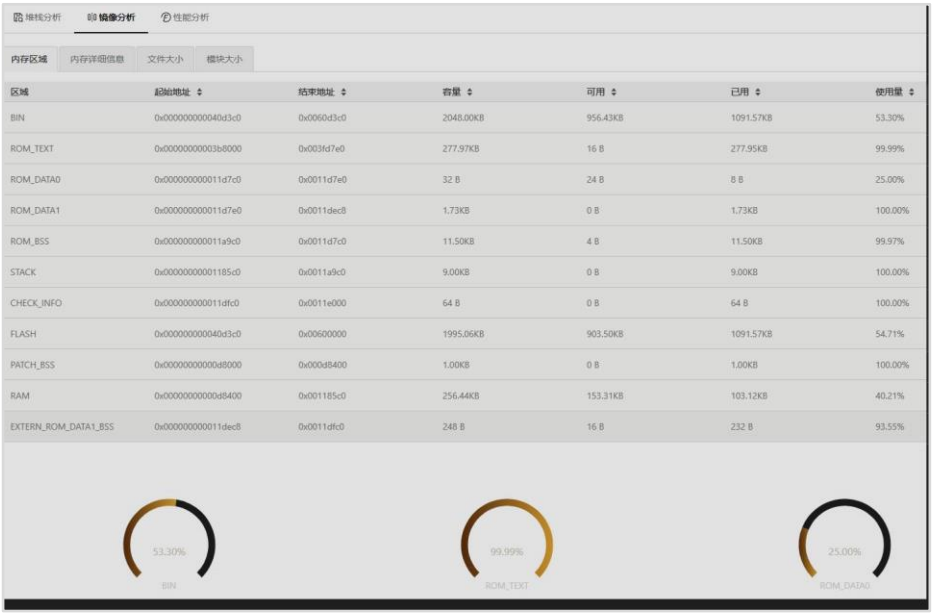

图6-7 镜像分析结果案例图

6.3.4 Profiling 可视化分析

步骤 1 编译分析文件并烧录

找到下方的文件文件“app_demo_sysinfo.c”和“app_demo_sysinfo.h”。

```
~/Huawei/DevEco-Device-Tool/platforms/hisilicon_riscv/profiling/src $ ls
app_demo_sysinfo.c  app_demo_sysinfo.h
```

将“app_demo_sysinfo.c”和“app_demo_sysinfo.h”文件拷贝至应用程序目录。

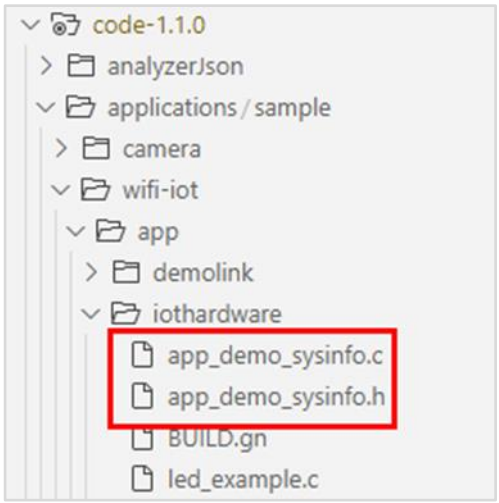

图6-8 应用程序位置示意图

在应用的编译脚本 BUILD.gn 文件中添加信息。

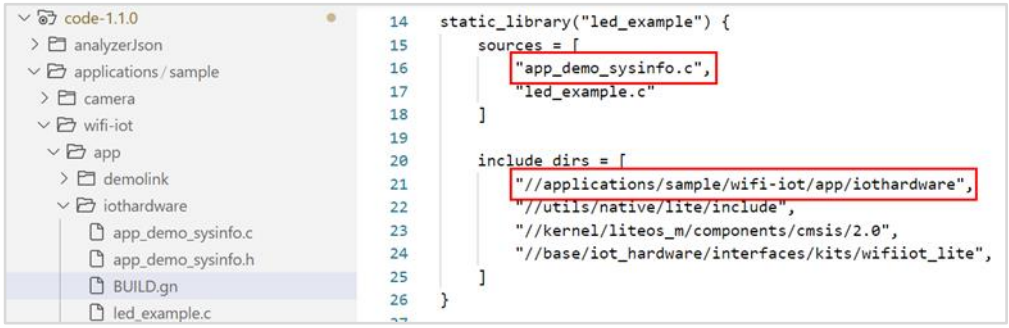

图6-9 BUILD.gn 文件示意图

应用入口源文件 led_example.c 中添加信息头文件，入口函数添加调用函数 app_demo_heap_task()。

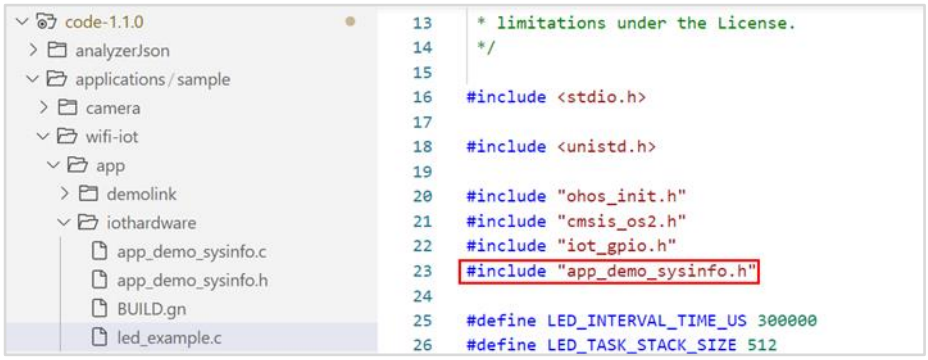

图6-10 添加信息示意图

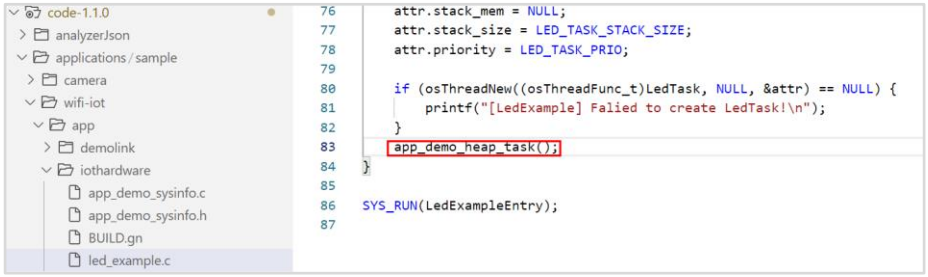

图6-11 添加调用函数示意图

修改 app/BUILD.gn，使 iothardware 参与编译。

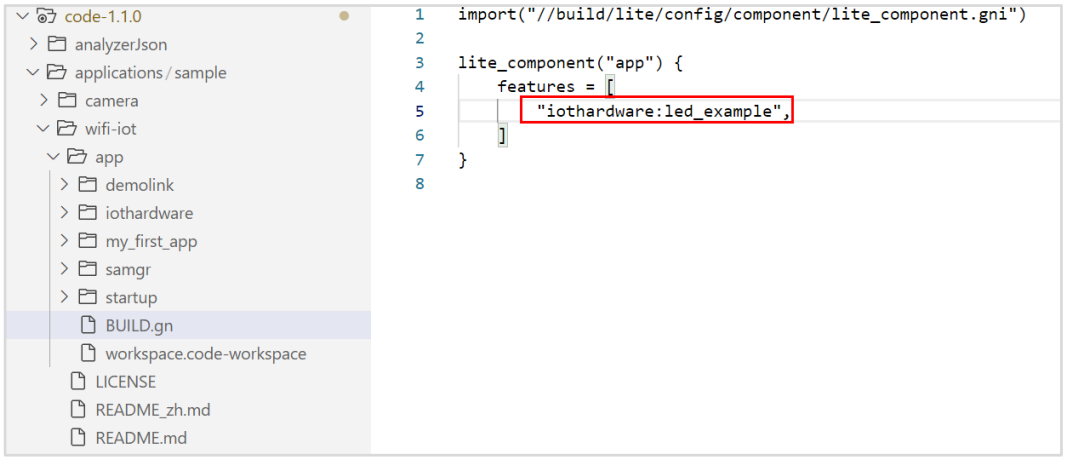

图6-12 修改编译配置示意图

步骤 2 结果展示

重新启动编译并烧录到开发板中，待烧录完成后，复位开发板。

在 DevEco Device Tools 中，点击 Profiling > 实时捕获，选择串口和设置波特率，点击 Capture，即可显示系统资源和内存资源的占用情况了。

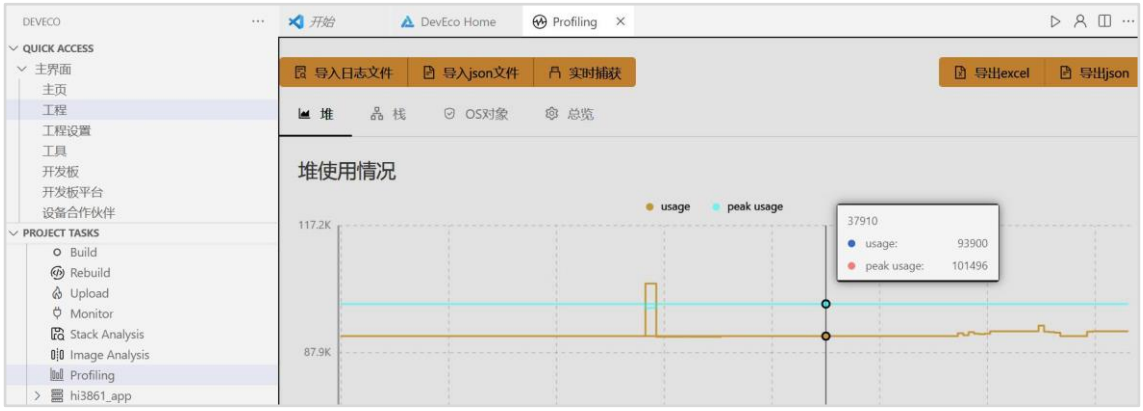

图6-13 内存资源与系统资源占用展示案例图

可视化数据曲线包括 Heap（堆内存）、Stack（栈内存）、OS Objects（系统资源）和 All in one（实时的内存占用情况）的使用情况。

7 系统移植

7.1 系统移植概述

OpenHarmony 系统的移植适配包含的步骤如下图所示。

图7-1 移植步骤示意图

1. 移植准备：移植前请先详细了解开发板的信息，比如 CPU 架构、RAM 和 ROM 大小、FLASH 类型，选择合适的 OpenHarmony 系统类型。移植前还需要详细了解 OpenHarmony 的系统架构和特性，编译框架，目录结构信息。然后创建芯片解决方案，配置芯片相关的编译链接选项。
2. 内核移植：根据选择的系统类型移植对应内核，轻量系统使用 LiteOS-M，小型系统使用 LiteOS-A 或 Linux。内核的移植主要包含内核功能配置、启动和初始化配置、内存和中断的配置等等。
3. 驱动移植：按 HDF 驱动框架和开发板硬件情况完成驱动适配。常见的驱动有 GPIO、I2C、LCD、TP、WLAN。
4. 部件移植：除了部分与硬件和产品解决方案相关的部件，大多数部件在内核和驱动移植成功后都可按需拼装到产品解决方案中，无需额外的移植和适配。
5. 调试验证：上板调试验证移植结果，主要包括部件拼装、编译制作产品镜像、烧录和调试。

7.2 移植准备

系统整体工程较为复杂，大部分目录及实现为系统部件，如果不涉及复杂的特性增强，不需要关注每一层实现，移植过程中重点关注如下目录即可：

表7-1 移植相关目录表

目录名称	描述
/build	命令处理器基础编译构建框架
/kernel	基础内核，其中芯片架构相关实现在arch目录下

/device	板级相关实现，各个三方厂商按照系统规范适配实现
/vendor	产品级相关实现，主要由华为或者产品厂商贡献

以 OpenHarmony3.0 为例，device 目录规则：device/{芯片解决方案厂商}/{开发板}。以 hisilicon 的 hispark_taurus 为例（小型系统）：

```
device
└── hisilicon                    # 芯片解决方案厂商名
    └── hispark_taurus           # 开发板名称
        ├── BUILD.gn             # 开发板编译入口
        └── sdk_liteos           # liteos 版本
            ├── ...
            └── config.gni       # liteos 版本编译工具链和编译选项配置
```

以 OpenHarmony3.0 为例，vendor 目录规则：vendor/{产品解决方案厂商}/{产品名称}。以 hisilicon 的 hispark_taurus 为例（小型系统）：

```
vendor                           # 产品解决方案厂商
└── hisilicon                    # 产品解决方案厂商名称
    └── hispark_taurus           # 产品名称
        ├── hals                 # 产品解决方案厂商 OS 适配
        ├── BUILD.gn             # 产品编译脚本
        └── config.json          # 产品配置文件
```

在进行移植之前，需要进行编译适配。编译适配主要使用 hb set 命令，设置整个项目的根目录、单板目录、产品目录、单板公司名等环境变量，为编译做准备。以 hisilicon 的 hispark_taurus 为例。

1. 在 vendor/hisilicon/hispark_taurus 下新增 config.json 文件用于描述这个产品样例所使用的单板、内核等信息，描述信息可参考如下内容：

```
{
  "product_name": "ipcamera_hispark_taurus",     --- 用于 hb set 进行选择时，显示的产品名称
  "ohos_version": "OpenHarmony 3.0",             --- 系统版本
  "device_company": "hisilicon",                 --- 单板厂商名
  "board": "hispark_taurus ",                    --- 单板名
  "kernel_type": "liteos_a",                     --- 内核类型
  "kernel_version": "",                          --- 内核版本，一块单板可能适配了多个 linux 内核版本，所以需要指定某个具体的内核版本进行编译
  "subsystems": [ ]                              --- 选择所需要编译构建的子系统
}
```

2. 在 device/hisilicon/hispark_taurus/sdk_liteos 目录下新增 config.gni 文件，用于描述这个产品样例所使用的单板、内核等信息，描述信息可参考如下内容：

```
# Kernel type, e.g. "linux", "liteos_a", "liteos_m".
kernel_type = "liteos_a"

# Kernel version.
```

```
kernel_version = ""
......
# Board CPU type, e.g. "cortex-a7", "riscv32".
board_cpu = "cortex-a7"
# Compiler type, "gcc" or "clang".
board_toolchain_type = "clang"
```

7.3 内核移植

7.3.1 LiteOS 内核

步骤 1 基础适配

以 LiteOS-A 为例，LiteOS-A 提供系统运行所需的系统初始化流程和定制化配置选项。移植过程中，需要关注初始化流程中跟硬件配置相关的函数。

LiteOS-A 的初始化流程主要包含以下七步：

1. 新增 target_config.h 文件，并且编写单板内存相关的配置宏 DDR_MEM_ADDR 和 DDR_MEM_SIZE，分别表示内存起始地址和内存的长度，预链接脚本 board.ld.S 会根据这两个宏进行展开生成链接脚本 board.ld。
2. 新增定义 MMU 映射全局数组(g_archMmuInitMapping)，指定各个内存段属性及虚实映射关系，内核启动阶段根据该表建立内存映射关系。
3. 如果是多核，需要新增定义从核操作函数句柄(struct SmpOps)，其中 SmpOps->SmpCpuOn 函数需要实现唤醒从核的功能；接着定义 SmpRegFunc 函数，调用 LOS_SmpOpsSet 接口进行句柄注册；最后通过启动框架完成注册过程，即 LOS_MODULE_INIT(SmpRegFunc, LOS_INIT_LEVEL_EARLIEST)。
4. 链接阶段根据链接脚本 board.ld 生成内核镜像。
5. 单核 CPU 镜像运行入口为汇编文件 reset_vector_up.S，多核 CPU 的入口为 reset_vector_mp.S，在汇编文件中进行中断向量表初始化、MMU 页表初始化等操作。
6. reset_vector.S 汇编代码最终会跳转到 C 语言的 main 函数，进行硬件时钟、软件定时器、内存和任务等初始化，这个过程会依赖 target_config.h 的特性宏配置，最后会创建 SystemInit 任务，并且开启任务调度 OsSchedStart()。
7. SystemInit 任务在单板代码中实现，其中调用 DeviceManagerStart 函数进行 HDF 驱动初始化，这个过程会调用单板代码中的驱动配置文件 hdf.hcs 以及 drivers 源码实现。

内核基础适配需要单板进行适配的代码包含三部分：

1. 新增 target_config.h 文件，其中新增单板硬件配置参数和特性开关的配置参数。

表7-2 配置参数表

配置项	说明
OS_SYS_CLOCK	系统cycle的频率
DDR_MEM_ADDR	系统内存的起始地址
DDR_MEM_SIZE	板级相关实现，各个三方厂商按照系统规范适配实现
PERIPH_PMM_BASE	外设寄存器的起始地址
PERIPH_PMM_SIZE	外设寄存器的长度大小
OS_HWI_MIN	系统中断最小值
OS_HWI_MAX	系统中断最大值
NUM_HAL_INTERRUPT_UART0	UART0中断号
UART0_REG_BASE	UART0寄存器基址
GIC_BASE_ADDR	GIC中断寄存器基址
GICD_OFFSET	GICD相对GIC基址的偏移地址
GICC_OFFSET	GICC相对GIC基址的偏移地址

2. SystemInit 函数用于单板用户态业务初始化，典型的初始化场景如下图：

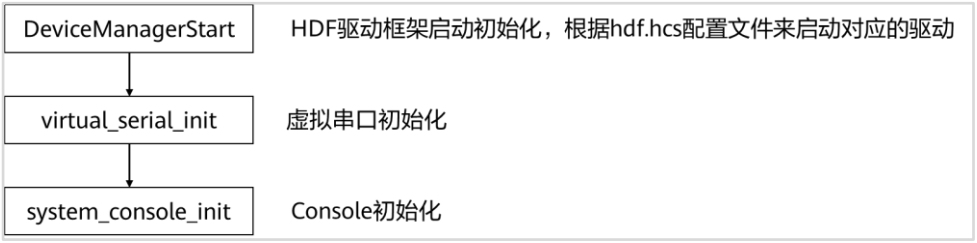

图7-2 典型初始化场景示意图

3. main 函数用于内核基础初始化和单板内核态业务初始化。

步骤 2 编程样例

在单板 SDK 文件中修改代码。

```
/* 内核启动框架头文件 */
#include "los_init.h"
……

/* 新增模块的初始化函数 */
unsigned int OsSampleModInit(void)
{
    PRINTK("OsSampleModInit SUCCESS!\n");
    ……
}
……
```

```
/* 在启动框架的目标层级中注册新增模块 */
LOS_MODULE_INIT(OsSampleModInit, LOS_INIT_LEVEL_KMOD_EXTENDED);
```

步骤 3 验证

```
main core booting up...
OsSampleModInit SUCCESS!
releasing 1 secondary cores
cpu 1 entering scheduler
cpu 0 entering scheduler
```

根据上述系统启动阶段的打印可知，内核在启动时进行了该注册模块的初始化函数调用，完成该模块的初始化操作。

系统启动完毕后进入内核态 shell，能够运行 task 命令能够正常显示即可。

```
OHOS # help
******************shell commands:*************************
arp             cat             cd              chgrp           chmod           chown           cp
cpup
date            dhclient        dmesg           dns             format          free            help
hwi
ifconfig        ipdebug         kill            log             ls              lsfd            memcheck
mkdir
mount           netstat         oom             partinfo        partition       ping            ping6
pmm
pwd             reset           rm              rmdir           sem             shm             stack
statfs
su              swtmr           sync            systeminfo      task            telnet          touch
umount
uname           v2p             virstatfs       vmm             watch           writeproc
```

7.3.2 Linux 内核

适配编译和烧录启动。

1. 准备内核 config（特别是芯片相关的 config）。

config 文件所在源码目录：kernel/linux/config/

以 hi3516dv300 芯片为例，可在对应的 linux-4.19/arch/arm/configs/目录下新建 <YOUR_CHIP>_small_defconfig，如 hi3516dv300_small_defconfig 表示针对 hi3516dv300 小型系统的 defconfig。该 config 文件可以由基础 defconfig 文件 small_common_defconfig 与该芯片相关的 config 组合生成。

2. 准备芯片补丁。

补丁文件所在源码目录：kernel/linux/patches/linux-4.19

以 hi3516dv300 芯片为例，参考已有的 patch 目录 hi3516dv300_small_patch 目录，新建 <YOUR_CHIP>_patch 目录，放置相关芯片补丁，注意 hdf.patch 等驱动补丁。

3. 编译。

具体内核编译入口脚本位于工程目录 kernel/linux/patches/下面，版本级整编命令会通过 BUILD.gn 进入 kernel_module_build.sh 和 kernel.mk，需要在这 2 个文件中针对性进行 patch 及 defconfig 文件路径、编译器、芯片架构、内核 Image 格式等的适配。

通过编译错误日志调整补丁，典型错误场景：

（1）补丁合入失败，出现冲突，需要进行上下文适配修改。

（2）编译失败，内核版本差异（函数实现调整等）需要针对性进行内核适配。

4. 烧录启动。

由于不同芯片的开发板的烧录方式不一样，此处不表述具体的烧录方式。需要注意烧录的各镜像的大小及启动参数的配置，参考 hi3516dv300 采用 uboot 启动参数：

```
setenv bootargs 'mem=128M console=ttyAMA0,115200 root=/dev/mmcblk0p3 ro rootfstype=ext4
rootwait blkdevparts=mmcblk0:1M(boot),9M(kernel),50M(rootfs),50M(userfs)'
```

7.4 驱动移植

7.4.1 HDF 驱动框架

驱动主要包含两部分，平台驱动和器件驱动。平台驱动主要包括通常在 SOC 内的 GPIO、I2C、SPI 等；器件驱动则主要包含通常在 SOC 外的器件，如 LCD、TP、WLAN 等。

系统 HDF 驱动框架采用 C 语言面向对象编程模型构建，通过平台解耦、内核解耦，来达到兼容不同内核，统一平台底座的目的，从而帮助开发者实现驱动一次开发，多系统部署的效果。

为了达成这个目标，系统 HDF 驱动框架提供了：

1. 操作系统适配层（OSAL，operatingsystem abstraction layer）：对内核操作相关接口进行统一封装，屏蔽不同系统操作接口。
2. 平台驱动接口：提供 Board 部分驱动（例如，I2C/SPI/UART 总线等平台资源）支持，同时对 Board 硬件操作进行统一的适配接口抽象，方便开发者只需开发新硬件抽象接口，即可获得新增 Board 部分驱动支持。
3. 驱动模型屏蔽：驱动与不同系统组件间的交互，使得驱动更具备通用型。

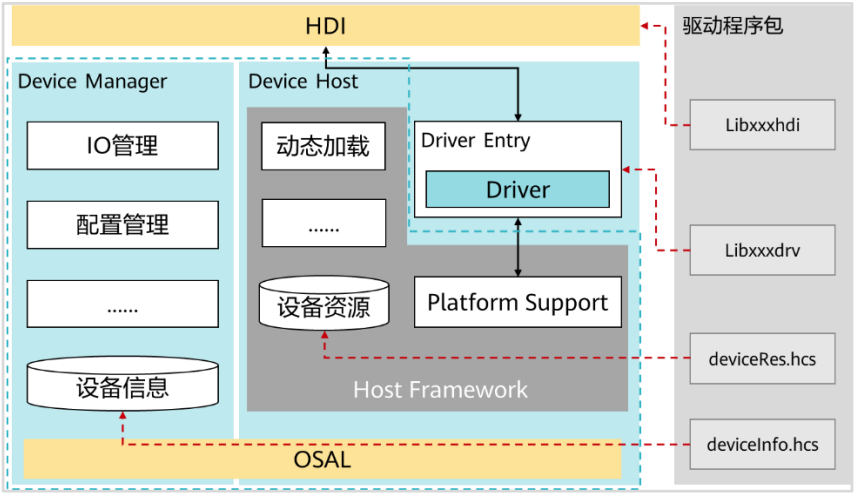

图7-3 HDF 驱动框架示意图

HDF 驱动框架主要由驱动基础框架、驱动程序、驱动配置文件和驱动接口这四个部分组成。

1. HDF 驱动基础框架提供统一的硬件资源管理，驱动加载管理以及设备节点管理等功能。驱动框架采用的是主从模式设计，由 Device Manager 和 Device Host 组成。
2. 驱动程序实现驱动具体的功能，每个驱动由一个或者多个驱动程序组成，每个驱动程序都对应着一个 Driver Entry。Driver Entry 主要完成驱动的初始化和驱动接口绑定功能。
3. 驱动配置文件 hcs 主要由设备信息（Device Information）和设备资源（Device Resource）组成。Device Information 完成设备信息的配置。如配置接口发布策略，驱动加载的方式等。Device Resource 完成设备资源的配置。如 GPIO 管脚、寄存器等资源信息的配置。
4. 驱动接口 HDI(Hardware Driver interface)提供标准化的接口定义和实现，驱动框架提供 IO Service 和 IO Dispatcher 机制，使得不同部署形态下驱动接口趋于形式一致。

7.4.2 平台驱动移植

在源码目录//device/vendor_name/soc_name/drivers 目录下创建平台驱动。

建议的目录结构：

```
device
├── vendor_name
│   ├── drivers
│   │   │   ├── common
│   │   │   ├── Kconfig   # 厂商驱动内核菜单入口
│   │   │   └── lite.mk    # 构建的入口
│   ├── soc_name
│   │   ├── drivers
│   │   │   ├── dmac
│   │   │   ├── gpio
│   │   │   ├── i2c
│   │   │   ├── LICENSE
│   │   │   ├── mipi_dsi
│   │   │   ├── mmc
│   │   │   ├── pwm
│   │   │   ├── README.md   # docs 如果需要的话
│   │   │   ├── README_zh.md
│   │   │   ├── rtc
│   │   │   ├── spi
│   │   │   ├── uart
│   │   │   └── watchdog
│   ├── board_name
```

以 GPIO 为例，讲解如何移植平台驱动，移植过程包含以下步骤：

步骤 1 创建 GPIO 驱动

在源码目录//device/vendor_name/soc_name/drivers/gpio 中创建文件 soc_name_gpio.c 内容模板如下：

```
#include "gpio_core.h"

// 定义 GPIO 结构体，如果需要的话
struct SocNameGpioCntlr {
    struct GpioCntlr cntlr;  // 这是 HDF GPIO 驱动框架需要的结构体
    int myData; // 以下是当前驱动自身需要的
};

// Bind 方法在 HDF 驱动中主要用户对外发布服务，这里我们不需要，直接返回成功即可
static int32_t GpioBind(struct HdfDeviceObject *device)
{
    (void)device;
    return HDF_SUCCESS;
}

// Init 方法时驱动初始化的入口，我们需要在 Init 方法中完成模型实例的注册
static int32_t GpioInit(struct HdfDeviceObject *device)
{
    SocNameGpioCntlr *impl = CreateGpio(); // 你的创建代码
    ret = GpioCntlrAdd(&impl->cntlr);  // 注册 GPIO 模型实例
    if (ret != HDF_SUCCESS) {
        HDF_LOGE("%s: err add controller:%d", __func__, ret);
        return ret;
    }
    return HDF_SUCCESS;
}

// Release 方法会在驱动卸载时被调用，这里主要完成资源回收
static void GpioRelease(struct HdfDeviceObject *device)
{
    // GpioCntlrFromDevice 方法能从抽象的设备对象中获得 init 方法注册进去的模型实例。
    struct GpioCntlr *cntlr = GpioCntlrFromDevice(device);
    //资源释放...
}

struct HdfDriverEntry g_gpioDriverEntry = {
    .moduleVersion = 1,
    .Bind = GpioBind,
    .Init = GpioInit,
    .Release = GpioRelease,
    .moduleName = "SOC_NAME_gpio_driver", // 这个名字我们稍后会在配置文件中用到，用来加载驱动。
```

```
};
HDF_INIT(g_gpioDriverEntry); // 注册一个 GPIO 的驱动入口
```

步骤 2 创建厂商驱动构建入口

device/vendor_name/drivers/lite.mk 是厂商驱动的构建的入口。我们需要从这个入口开始，进行构建。

```
#文件 device/vendor_name/drivers/lite.mk

SOC_VENDOR_NAME := $(subst $/",,$(LOSCFG_DEVICE_COMPANY))
SOC_NAME := $(subst $/",,$(LOSCFG_PLATFORM))
BOARD_NAME := $(subst $/",,$(LOSCFG_PRODUCT_NAME))

# 指定 SOC 进行构建
LIB_SUBDIRS += $(LITEOSTOPDIR)/../../device/$(SOC_VENDOR_NAME)/$(SOC_NAME)/drivers/
```

步骤 3 创建 SOC 驱动构建入口

```
#文件 device/vendor_name/soc_name/drivers/lite.mk

SOC_DRIVER_ROOT := $(LITEOSTOPDIR)/../../device/$(SOC_VENDOR_NAME)/$(SOC_NAME)/drivers/

# 判断如果打开了 GPIO 的内核编译开关
ifeq ($(LOSCFG_DRIVERS_HDF_PLATFORM_GPIO), y)
    # 构建完成要链接一个叫 hdf_gpio 的对象
    LITEOS_BASELIB += -lhdf_gpio
    # 增加构建目录 gpio
    LIB_SUBDIRS      += $(SOC_DRIVER_ROOT)/gpio
endif

# 后续其他驱动在此基础上追加
```

步骤 4 创建 GPIO 构建入口

```
include $(LITEOSTOPDIR)/config.mk
include $(LITEOSTOPDIR)/../../drivers/adapter/khdf/liteos/lite.mk

# 指定输出对象的名称，注意要与 SOC 驱动构建入口里的 LITEOS_BASELIB 保持一致
MODULE_NAME := hdf_gpio

# 增加 HDF 框架的 INCLUDE
LOCAL_CFLAGS += $(HDF_INCLUDE)

# 要编译的文件
LOCAL_SRCS += soc_name_gpio.c

# 编译参数
```

```
LOCAL_CFLAGS += -fstack-protector-strong -Wextra -Wall -Werror -fsigned-char -fno-strict-aliasing -fno-common

include $(HDF_DRIVER)
```

步骤 5 配置产品加载驱动

产品的所有设备信息被定义在源码文件//vendor/vendor_name/product_name/config/device_info/device_info.hcs 中。

平台驱动请添加到 platform 的 host 中，moduleName 字段的定义与步骤 1 中 moduleName 的定义要保持一致。

```
root {
    ...
    platform :: host {
        device_gpio :: device {
            device0 :: deviceNode {
                policy = 0;
                priority = 10;
                permission = 0644;
                moduleName = "SOC_NAME_gpio_driver";
            }
        }
    }
}
```

7.4.3 器件驱动移植

以 WLAN 驱动移植为例。WLAN 驱动分为两部分，一部分负责管理 WLAN 设备，另一个部分负责处理 WLAN 流量。

步骤 1 创建 HDF WLAN 芯片驱动

在目录/device/vendor_name/peripheral/wifi/chip_name/ 创建文件hdf_wlan_chip_name.c。内容模板如下：

```
static int32_t HdfWlanHisiChipDriverInit(struct HdfDeviceObject *device) {
    static struct HdfChipDriverFactory factory = CreateChipDriverFactory(); // 需要移植者实现的方法
    struct HdfChipDriverManager *driverMgr = HdfWlanGetChipDriverMgr();
    if (driverMgr->RegChipDriver(&factory) != HDF_SUCCESS) { // 注册驱动工厂
        HDF_LOGE("%s fail: driverMgr is NULL!", __func__);
        return HDF_FAILURE;
    }
    return HDF_SUCCESS;
}

struct HdfDriverEntry g_hdfXXXChipEntry = {
    .moduleVersion = 1,
    .Init = HdfWlanXXXChipDriverInit,
```

```
    .Release = HdfWlanXXXChipRelease,
    .moduleName = "HDF_WIFI_CHIP_XXX" // 注意：这个名字要与配置一致
};

HDF_INIT(g_hdfXXXChipEntry);
```

步骤 2 编写配置文件描述驱动支持的芯片

在产品配置目录下创建芯片的配置文件，保存至源码路径 //vendor/vendor_name/product_name/config/wifi/wlan_chip_chip_name.hcs。该文件模板如下：

```
root {
    wlan_config {
        chip_name :& chipList {
            chip_name :: chipInst {
                match_attr = "hdf_wlan_chips_chip_name"; /* 这是配置匹配属性，用于提供驱动的配置根 */
                driverName = "driverName"; /* 需要与 HdfChipDriverFactory 中的 driverName 相同*/
                sdio {
                    vendorId = 0xXXXX; /* your vendor id */
                    deviceId = [0xXXXX]; /*your supported devices */
                }
            }
        }
    }
}
```

步骤 3 编写配置文件，加载驱动

产品的所有设备信息被定义在源码文件 //vendor/vendor_name/product_name/config/device_info/device_info.hcs 中。修改该文件，在名为 network 的 host 中，名为 device_wlan_chips 的 device 中增加配置。模板如下：

```
                deviceN :: deviceNode {
                    policy = 0;
                    preload = 2;
                    moduleName = "HDF_WLAN_CHIPS";
                    deviceMatchAttr = "hdf_wlan_chips_chip_name";
                    serviceName = "driverName";
                }
```

步骤 4 修改 Kconfig 文件，让移植的 WLAN 模组出现再内核配置中

在 device/vendor_name/drivers/Kconfig 中增加配置菜单，模板如下：

```
config DRIVERS_HDF_WIFI_chip_name
    bool "Enable chip_name Host driver"
    default n
    depends on DRIVERS_HDF_WLAN    help
```

```
        Answer Y to enable chip_name Host driver.
```

步骤 5 修改构建脚本，让驱动参与内核构建

在源码文件//device/vendor_name/drivers/lite.mk 末尾追加如下内容：

```
ifeq ($(LOSCFG_DRIVERS_HDF_WIFI_chip_name), y)
    # 构建完成要链接一个叫 hdf_wlan_chipdriver_chip_name 的对象，建议按这个命名，防止冲突
    LITEOS_BASELIB += -lhdf_wlan_chipdriver_chip_name
    # 增加构建目录 gpio
    LIB_SUBDIRS      += ../peripheral/wifi/chip_name
endif
```

7.5 部件移植

在产品解决方案的框架搭建完成后，便可以开始添加更多的系统部件，为应用的开发做准备。有些部件是系统必选的，以小型系统为例，必选部件如下：

表7-3 必选部件表

子系统	部件
分布式软总线	softbus
分布式调度	samgr、safwk、dmsfwk
启动恢复	init、appspawn、bootstrap、syspara
元能力	aafwk
升级与安全	Ota、permission
电源管理	powermgr
公共基础库	kv_store

在产品解决方案的框架搭建完成后，便可以开始添加更多的系统部件，为应用的开发做准备。添加系统基础的部件如 safwk_lite、samgr_lite、syspara_lite、bootstrap_lite、init_lite 和 kv_store 到 config.json 中。

```
{
    "product_name": "qemu_small_system_demo",
    "ohos_version": "OpenHarmony 3.0",
    "device_company": "qemu",
    "board": "arm_virt",
    "kernel_type": "liteos_a",
    "kernel_version": "3.0.0",
    "subsystems": [
      {
        "subsystem": "distributed_schedule",
```

```
        "components": [
          { "component": "safwk_lite", "features":[] },
          { "component": "samgr_lite", "features":[] }
        ]
      },
      {
        "subsystem": "startup",
        "components": [
          { "component": "syspara_lite", "features":[] },
          { "component": "bootstrap_lite", "features":[] },
          { "component": "init_lite", "features":[] }
        ]
      },
      {
        "subsystem": "kernel",
        "components": [
          { "component": "liteos_a", "features":[] }
        ]
      },
      {
        "subsystem": "utils",
        "components": [
          { "component": "kv_store", "features":[] },
          { "component": "os_dump", "features":[] }
        ]
      }
    ],
    "vendor_adapter_dir": "//device/qemu/hardware",
    "third_party_dir": "//third_party",
    "product_adapter_dir": "//vendor/my_company/my_product/hals",
}
```

8 子系统能力介绍

8.1 子系统概述

OpenHarmony 的系统架构共分为四层，分别是应用层、应用框架层、系统服务层与内核层，在这些分层结构的内部，各个业务能力分做了不同的子系统，例如 AI 子系统、内核子系统、驱动子系统、图形图像子系统等，不同的子系统负责各自领域的业务。

系统功能按照“系统 > 子系统 > 功能/模块”逐级展开，在多设备部署场景下，支持根据实际需求裁剪某些非必要的子系统或功能/模块。

在 OpenHarmony 当中，常见的子系统包括 AI 子系统、OTA 升级子系统、XTS 子系统、图形图像子系统、Sensor 服务子系统、媒体子系统、IoT 专有业务子系统、安全子系统等，围绕课程内容，在本章当中主要介绍 AI 子系统、OTA 升级子系统与 XTS 子系统。

8.2 AI 子系统

8.2.1 AI 的基本原理与能力

AI 也就是 Artificial Intelligence 人工智能的缩写，意指利用计算机来对人的意识、思维信息过程、智能行为进行模拟（如学习、推理、思考、规划等）和延伸，使计算机能实现更高层次的应用。AI 的开发一般会涉及到这些流程步骤：数据集使用，模型训练，导出离线模型，模型轻量化，模型集成，业务开发。

图8-1 AI 开发常规流程示意图

AI 的使用领域及其广泛，设备的强智能化能力往往离不开 AI 能力的支持。

机器视觉领域，被拍摄目标的形态信息、像素分布、颜色和亮度信息，将被转变成数字化图像信号，传送给图像处理系统。

在生物识别领域，AI 能力可利用生理特性和行为特征进行个人身份的鉴定。

在编码遗传领域，利用遗传算法和图灵完备语言，开发的程序理论上能完成任何类型的任务。

AI 能力在构建专家系统时也会发挥极大的作用，专家系统是一个模拟人类专家解决领域问题的系统，数据库中包含有大量的某个领域专家水平的知识和经验，跟进用户的咨询，进行推理和判断，AI 能力可以模拟人类专家的决策过程。

在机器学领域当中，应用机器的视觉触觉听觉等技术、机器人语言和智能控制软件等，来承担危险的任务。

当前的 AI 能力已经用于生活中的方方面面，例如最为常见的物体识别、植物种类识别等功能。

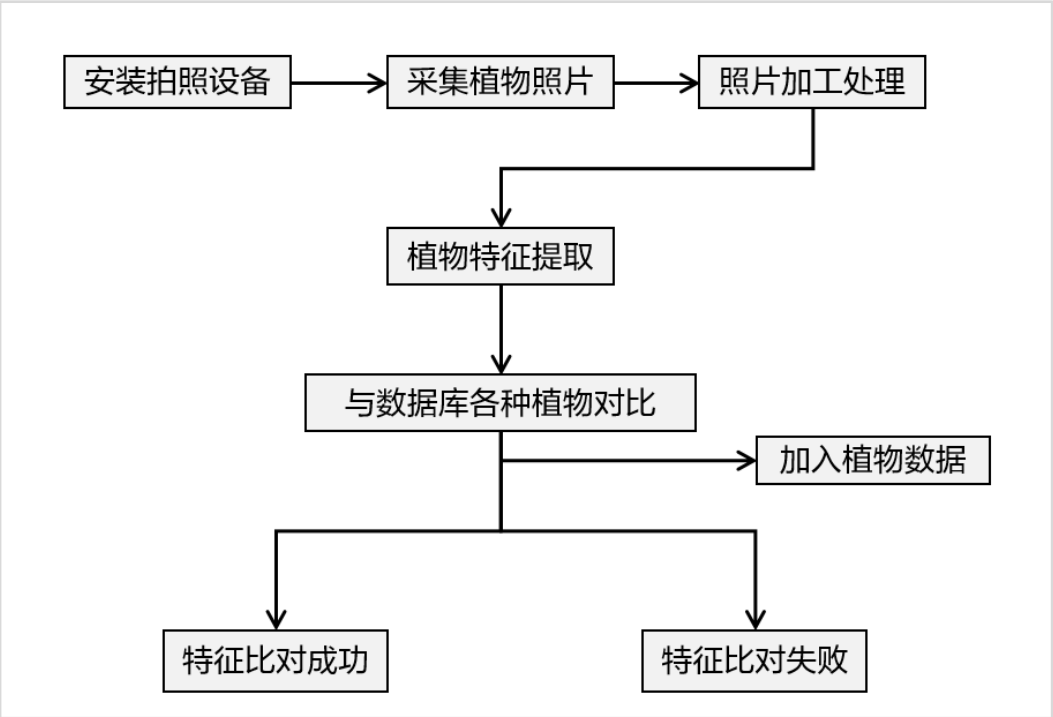

图8-2 图像识别业务流程示意图

8.2.2 OpenHarmony 的 AI 子系统

AI 业务子系统是 OpenHarmony 提供原生的分布式 AI 能力的子系统。AI 业务子系统提供了统一的 AI 引擎框架，实现算法能力快速插件化集成。

框架中主要包含插件管理、模块管理和通信管理等模块，完成对 AI 算法能力的生命周期管理和按需部署。

插件管理主要实现插件的生命周期管理及插件的按需部署,快速集成 AI 能力插件；

模块管理主要实现任务的调度及管理客户端的实例；

通信管理主要实现客户端和服务端之间的跨进程通信及引擎与插件之间的数据传输。后续，会逐步定义统一的 AI 能力接口，便于 AI 能力的分布式调用。同时，提供适配不同推理框架层级的统一推理接口。

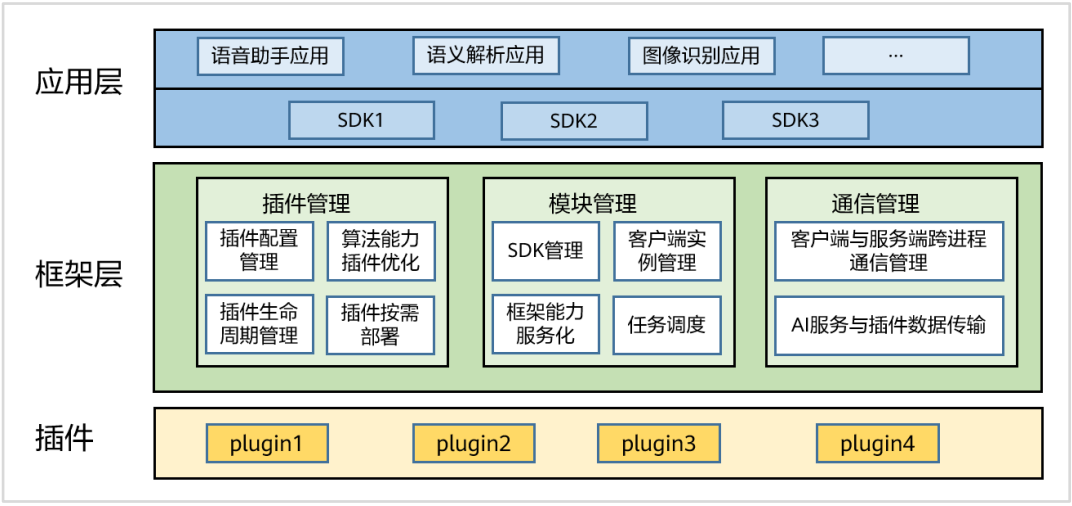

图8-3 AI 子系统结构示意图

AI 引擎框架包含 client、server 和 common 三个主要模块，其中 client 提供 server 端连接管理功能，北向 SDK 在算法对外接口中需封装调用 client 提供的公共接口；server 提供插件加载以及任务管理等功能，各 Plugin 实现由 server 提供的插件接口，完成插件接入；common 提供与平台相关的操作方法、引擎协议以及相关工具类，供其他各模块调用。

在 AI 引擎框架的整体规划中，北向 SDK 属于 client 端的一部分，插件由 server 端调用，属于 server 端的一部分，在 OpenHarmony 源码中，AI 引擎框架为接入的插件与北向 SDK 规划的路径：

SDK 代码路径：//foundation/ai/engine/services/client/algorithm_sdk

插件代码路径：//foundation/ai/engine/services/server/plugin

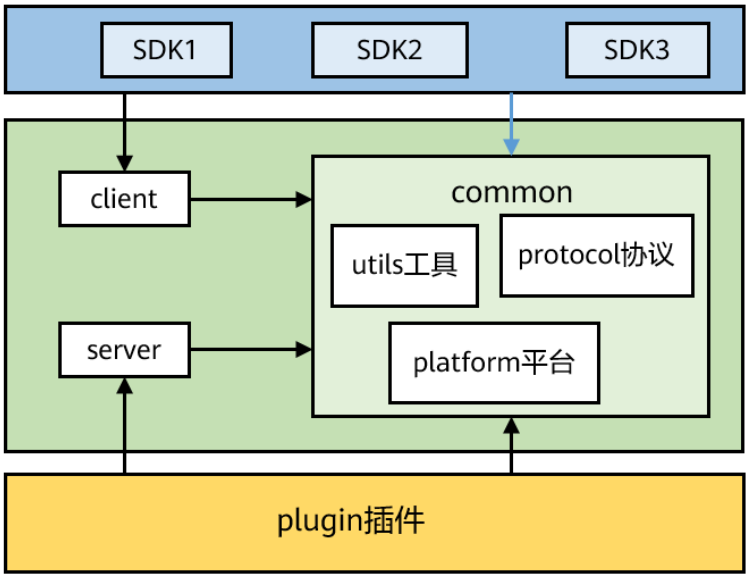

图8-4 AI 引擎框架示意图

SDK 需按算法调用顺序，封装 client 对外提供接口；对于异步插件对应的 SDK，需要实现 client 提供的回调接口 IClientCb。

AI 引擎将的 client 端采用单例实现，对接多个 SDK，因此各 SDK 需要保存与 client 交互的通用数据，用于连接 server 端进行任务推理、结果返回等；需保存数据包含 clientInfo、algorithmInfo、configInfo 三种数据类型，定义在 SDK 的成员变量里即可。

插件的推理数据，会由三方调用者通过 client、server 传递到插件中；不同的算法所需要的数据类型是不一致的，比如 cv 的需要图片数据、asr 的需要语音数据。

为了适配数据类型的差异，AI 引擎对外提供了对基本数据类型的编解码能力，将不同数据类型转换为 AI 引擎可以使用的通用数据类型。

AI 引擎框架规定了一套算法插件接入规范，各插件需实现规定接口以实现获取插件版本信息、算法推理类型、同步执行算法、异步执行算法、加载算法插件、卸载算法插件、设置算法配置信息、获取指定算法配置信息等功能。

8.2.3 OpenHarmony 的 AI 子系统开发与使用

在 OpenHarmony 当中进行 AI 开发，可在源码的/foundation/ai/engine 目录当中进行，/foundation/ai/engine 的主体结构如下。

```
/foundation/ai/engine                    # AI 子系统主目录
├── interfaces
│   └── kits                             # AI 子系统对外接口
└── services
    ├── client                           # AI 子系统 Client 模块
    │   ├── client_executor              # Client 模块执行主体
    │   └── communication_adapter        # Client 模块通信适配层，支持拓展
    ├── common                           # AI 子系统公共工具、协议模块
    │   ├── platform
    │   ├── protocol
    │   └── utils
    └── server                           # AI 子系统服务端模块
        ├── communication_adapter        # Server 模块通信适配层，支持拓展
        ├── plugin
        │   ├── asr
        │   │   └── keyword_spotting     # ASR 算法插件参考：唤醒词识别
        │   └── cv
        │       └── image_classification # CV 算法插件参考：图片分类
        ├── plugin_manager
        └── server_executor              # Server 模块执行主体
```

轻量级 AI 引擎框架模块，代码所在路径：//foundation/ai/engine/services

编译指令如下：

设置编译路径

```
hb set -root dir(项目代码根目录)
```

设置编译产品（执行后用方向键和回车进行选择）：

```
hb set -p
```

执行编译：

```
hb build -f（编译全仓）
```

或者只编译 ai_engine 组件。

```
hb build ai_engine（只编译 ai_engine 组件）
```

插件开发（以唤醒词识别为例）

位置：//foundation/ai/engine/services/server/plugin/asr/keyword_spotting

注意：插件需要实现 server 提供的 IPlugin 接口和 IPluginCallback 接口

```
#include "plugin/i_plugin.h
class KWSPlugin : public IPlugin {          # Keywords Spotting Plugin（KWSPlugin）继承 IPlugin 算法插件基类 public:
    KWSPlugin();
    ~KWSPlugin();
    const long long GetVersion() const override;
    const char* GetName() const override;
    const char* GetInferMode() const override;
    int32_t Prepare(long long transactionId, const DataInfo &amp;amp;inputInfo, DataInfo &amp;amp;outputInfo) override;
    int32_t SetOption(int optionType, const DataInfo &amp;amp;inputInfo) override;
    int32_t GetOption(int optionType, const DataInfo &amp;amp;inputInfo, DataInfo &amp;amp;outputInfo) override;
    int32_t SyncProcess(IRequest *request, IResponse *&amp;amp;response) override;
    int32_t AsyncProcess(IRequest *request, IPluginCallback*callback) override;
    int32_t Release(bool isFullUnload, long long transactionId, const DataInfo &amp;amp;inputInfo) override;
    ...
};
```

注意：SyncProcess 和 AsyncProcess 接口只需要根据算法实际情况实现一个接口即可，另一个用空方法占位（这里 KWS 插件为同步算法，故异步接口为空实现）。

```
#include "aie_log.h"
#include "aie_retcode_inner.h"
...
const long long KWSPlugin::GetVersion() const
{
    return ALGOTYPE_VERSION_KWS;
}

const char *KWSPlugin::GetName() const
{
    return ALGORITHM_NAME_KWS.c_str();
}

const char *KWSPlugin::GetInferMode() const
{
    return DEFAULT_INFER_MODE.c_str();
```

```
}
...
int32_t KWSPlugin::AsyncProcess(IRequest *request, IPluginCallback *callback)
{
    return RETCODE_SUCCESS;
}
```

插件 SDK 开发（以唤醒词识别 kws_sdk 为例）

位置：//foundation/ai/engine/services/client/algorithm_sdk/asr/keyword_spotting

唤醒词识别 SDK：

```
class KWSSdk {
public:
    KWSSdk();
    virtual ~KWSSdk();

    /**
     * @brief Create a new session with KWS Plugin
     *
     * @return Returns KWS_RETCODE_SUCCESS(0) if the operation is successful,
     *             returns a non-zero value otherwise.
     */
    int32_t Create();

    /**
     * @brief Synchronously execute keyword spotting once
     *
     * @param audioInput pcm data.
     * @return Returns KWS_RETCODE_SUCCESS(0) if the operation is successful,
     *             returns a non-zero value otherwise.
     */
    int32_t SyncExecute(const Array<int16_t> &audioInput);

    /**
     * @brief Asynchronously execute keyword spotting once
     *
     * @param audioInput pcm data.
     * @return Returns KWS_RETCODE_SUCCESS(0) if the operation is successful,
     *             returns a non-zero value otherwise.
     */
    int32_t AsyncExecute(const Array<int16_t> &audioInput);

    /**
     * @brief Set callback
     *
     * @param callback Callback function that will be called during the process.
     * @return Returns KWS_RETCODE_SUCCESS(0) if the operation is successful,
     *             returns a non-zero value otherwise.
```

```
     */
    int32_t SetCallback(const std::shared_ptr<KWSCallback> &callback);

    /**
     * @brief Destroy the created session with KWS Plugin
     *
     * @return Returns KWS_RETCODE_SUCCESS(0) if the operation is successful,
     *             returns a non-zero value otherwise.
     */
    int32_t Destroy();
```

注意：SDK 调用 AI 引擎客户端接口顺序应遵循 AieClientInit->AieClientPrepare->AieClientSyncProcess/AieClientAsyncProcess->AieClientRelease->AieClientDestroy，否则调用接口会返回错误码；同时应保证各个接口都有调用到，要不然会引起内存泄漏。

```
int32_t KWSSdk::KWSSdkImpl::Create()
{
    if (kwsHandle_ != INVALID_KWS_HANDLE) {
        HILOGE("[KWSSdkImpl]The SDK has been created");
        return KWS_RETCODE_FAILURE;
    }
    if (InitComponents() != RETCODE_SUCCESS) {
        HILOGE("[KWSSdkImpl]Fail to init sdk components");
        return KWS_RETCODE_FAILURE;
    }
    int32_t retCode = AieClientInit(configInfo_, clientInfo_, algorithmInfo_, nullptr);
    if (retCode != RETCODE_SUCCESS) {
        HILOGE("[KWSSdkImpl]AieClientInit failed. Error code[%d]", retCode);
        return KWS_RETCODE_FAILURE;
    }
    if (clientInfo_.clientId == INVALID_CLIENT_ID) {
        HILOGE("[KWSSdkImpl]Fail to allocate client id");
        return KWS_RETCODE_FAILURE;
    }
    DataInfo inputInfo = {
        .data = nullptr,
        .length = 0,
    };
    DataInfo outputInfo = {
        .data = nullptr,
        .length = 0,
    };
    retCode = AieClientPrepare(clientInfo_, algorithmInfo_, inputInfo, outputInfo, nullptr);
    if (retCode != RETCODE_SUCCESS) {
        HILOGE("[KWSSdkImpl]AieclientPrepare failed. Error code[%d]", retCode);
        return KWS_RETCODE_FAILURE;
    }
    if (outputInfo.data == nullptr || outputInfo.length <= 0) {
```

```
        HILOGE("[KWSSdkImpl]The data or length of output info is invalid");
        return KWS_RETCODE_FAILURE;
    }
    MallocPointerGuard<unsigned char> pointerGuard(outputInfo.data);
    retCode = PluginHelper::UnSerializeHandle(outputInfo, kwsHandle_);
    if (retCode != RETCODE_SUCCESS) {
        HILOGE("[KWSSdkImpl]Get handle from inputInfo failed");
        return KWS_RETCODE_FAILURE;
    }
    return KWS_RETCODE_SUCCESS;
}

int32_t KWSSdk::KWSSdkImpl::SyncExecute(const Array<uint16_t> &audioInput)
{
    intptr_t newHandle = 0;
    Array<int32_t> kwsResult = {
        .data = nullptr,
        .size = 0
    };
    DataInfo inputInfo = {
        .data = nullptr,
        .length = 0
    };
    DataInfo outputInfo = {
        .data = nullptr,
        .length = 0
    };
    int32_t retCode = PluginHelper::SerializeInputData(kwsHandle_, audioInput, inputInfo);
    if (retCode != RETCODE_SUCCESS) {
        HILOGE("[KWSSdkImpl]Fail to serialize input data");
        callback_->OnError(KWS_RETCODE_SERIALIZATION_ERROR);
        return RETCODE_FAILURE;
    }
    retCode = AieClientSyncProcess(clientInfo_, algorithmInfo_, inputInfo, outputInfo);
    if (retCode != RETCODE_SUCCESS) {
        HILOGE("[KWSSdkImpl]AieClientSyncProcess failed. Error code[%d]", retCode);
        callback_->OnError(KWS_RETCODE_PLUGIN_EXECUTION_ERROR);
        return RETCODE_FAILURE;
    }
    if (outputInfo.data == nullptr || outputInfo.length <= 0) {
        HILOGE("[KWSSdkImpl] The data or length of outputInfo is invalid. Error code[%d]", retCode);
        callback_->OnError(KWS_RETCODE_NULL_PARAM);
        return RETCODE_FAILURE;
    }
    MallocPointerGuard<unsigned char> pointerGuard(outputInfo.data);
    retCode = PluginHelper::UnSerializeOutputData(outputInfo, newHandle, kwsResult);
    if (retCode != RETCODE_SUCCESS) {
        HILOGE("[KWSSdkImpl]UnSerializeOutputData failed. Error code[%d]", retCode);
```

```
            callback_->OnError(KWS_RETCODE_UNSERIALIZATION_ERROR);
            return retCode;
        }
        if (kwsHandle_ != newHandle) {
            HILOGE("[KWSSdkImpl]The handle[%lld] of output data is not equal to the current
handle[%lld]",
                (long long)newHandle, (long long)kwsHandle_);
            callback_->OnError(KWS_RETCODE_PLUGIN_SESSION_ERROR);
            return RETCODE_FAILURE;
        }
        callback_->OnResult(kwsResult);
        return RETCODE_SUCCESS;
}

int32_t KWSSdk::KWSSdkImpl::Destroy()
{
        if (kwsHandle_ == INVALID_KWS_HANDLE) {
            return KWS_RETCODE_SUCCESS;
        }
        DataInfo inputInfo = {
            .data = nullptr,
            .length = 0
        };
        int32_t retCode = PluginHelper::SerializeHandle(kwsHandle_, inputInfo);
        if (retCode != RETCODE_SUCCESS) {
            HILOGE("[KWSSdkImpl]SerializeHandle failed. Error code[%d]", retCode);
            return KWS_RETCODE_FAILURE;
        }
        retCode = AieClientRelease(clientInfo_, algorithmInfo_, inputInfo);
        if (retCode != RETCODE_SUCCESS) {
            HILOGE("[KWSSdkImpl]AieClientRelease failed. Error code[%d]", retCode);
            return KWS_RETCODE_FAILURE;
        }
        retCode = AieClientDestroy(clientInfo_);
        if (retCode != RETCODE_SUCCESS) {
            HILOGE("[KWSSdkImpl]AieClientDestroy failed. Error code[%d]", retCode);
            return KWS_RETCODE_FAILURE;
        }
        mfccProcessor_ = nullptr;
        pcmIterator_ = nullptr;
        callback_ = nullptr;
        kwsHandle_ = INVALID_KWS_HANDLE;
        return KWS_RETCODE_SUCCESS;
}
```

sample 开发 （参考唤醒词识别 demo）(opens new window)，位置：//applications/sample/camera/ai/asr/keyword_spotting，

```
//调用 Create
```

```
bool KwsManager::PreparedInference()
{
    if (capturer_ == nullptr) {
        printf("[KwsManager] only load plugin after AudioCapturer ready\n");
        return false;
    }
    if (plugin_ != nullptr) {
        printf("[KwsManager] stop created InferencePlugin at first\n");
        StopInference();
    }
    plugin_ = std::make_shared<KWSSdk>();
    if (plugin_ == nullptr) {
        printf("[KwsManager] fail to create inferencePlugin\n");
        return false;
    }
    if (plugin_->Create() != SUCCESS) {
        printf("[KwsManager] KWSSdk fail to create.\n");
        return false;
    }
    std::shared_ptr<KWSCallback> callback = std::make_shared<MyKwsCallback>();
    if (callback == nullptr) {
        printf("[KwsManager] new Callback failed.\n");
        return false;
    }
    plugin_->SetCallback(callback);
    return true;
}
//调用 SyncExecute
void KwsManager::ConsumeSamples()
{
    uintptr_t sampleAddr = 0;
    size_t sampleSize = 0;
    int32_t retCode = SUCCESS;
    while (status_ == RUNNING) {
        {
            std::lock_guard<std::mutex> lock(mutex_);
            if (cache_ == nullptr) {
                printf("[KwsManager] cache_ is nullptr.\n");
                break;
            }
            sampleSize = cache_->GetCapturedBuffer(sampleAddr);
        }
        if (sampleSize == 0 || sampleAddr == 0) {
            continue;
        }
        Array<int16_t> input = {
            .data = (int16_t *)(sampleAddr),
            .size = sampleSize >> 1
```

```
        };
        {
            std::lock_guard<std::mutex> lock(mutex_);
            if (plugin_ == nullptr) {
                printf("[KwsManager] cache_ is nullptr.\n");
                break;
            }
            if ((retCode = plugin_->SyncExecute(input)) != SUCCESS) {
                printf("[KwsManager] SyncExecute KWS failed with retCode = [%d]\n", retCode);
                continue;
            }
        }
    }
}
//调用 Destroy
void KwsManager::StopInference()
{
    printf("[KwsManager] StopInference\n");
    if (plugin_ != nullptr) {
        int ret = plugin_->Destroy();
        if (ret != SUCCESS) {
            printf("[KwsManager] plugin_ destroy failed.\n");
        }
        plugin_ = nullptr;
    }
}
```

8.3 OTA 升级子系统

8.3.1 OTA 的基本原理

OTA（Over the Air）也叫空中下载升级技术，它提供对设备远程升级的能力，例如对路口的摄像头进行软件升级，而无需拆卸硬件设备。OTA 升级可通过集群应用、网络技术和分布式服务端，在同一时间可满足大量用户的终端升级需求。

OTA 升级总体可以分为三个部分：升级包制作、终端与云端的交互、设备升级。

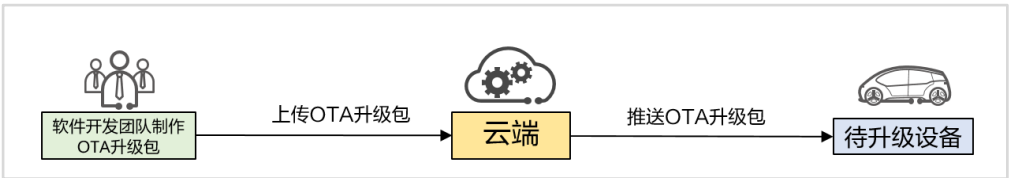

图8-5 OTA 升级原理示意图

OTA 常见的升级方式分为全量包升级与差分包升级。

全量包升级也称为整包升级，是将软件的更新部分和软件未更新的部分作为一个整体，在待升级设备下载 OTA 升级包之后，进行覆盖安装。目前 OpenHarmony 仅支持全量包升级。

差分包升级也称为增量升级，OTA 升级包当中主要是软件的更新部分，在待升级的设备下载 OTA 升级包之后，进行软件的增量更新。

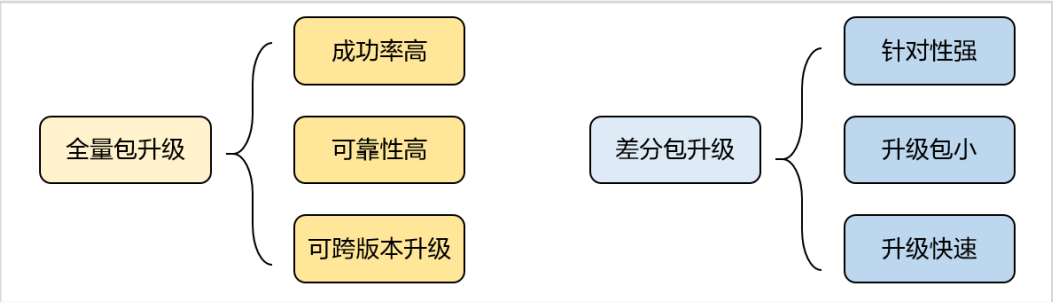

图8-6 全量包升级与差分包升级对比图

8.3.1 OTA 的技术架构

OTA 升级技术在智能硬件终端领域中已有非常广泛的应用，下图为一种常见的 OTA 升级架构。

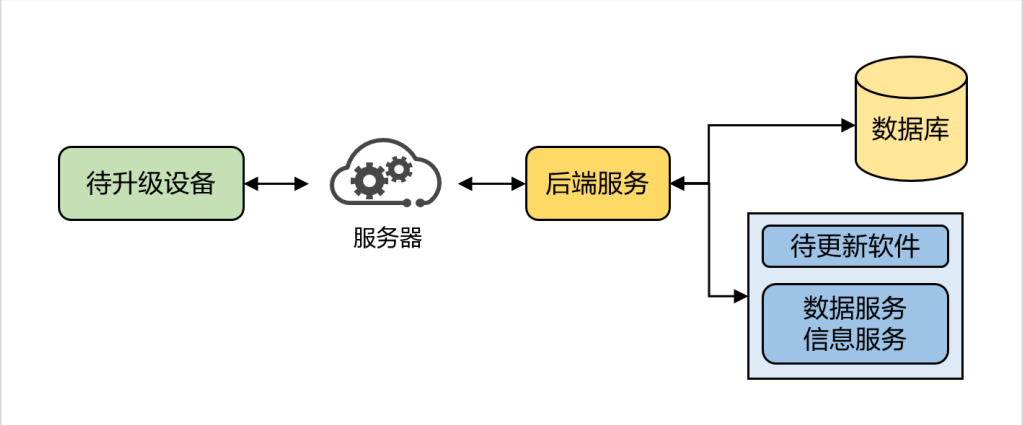

图8-7 OTA 升级架构图

在 OTA 升级当中，终端设备一般指的是待升级的终端设备，终端设备形态差异多样，根据设备的形态特征，获取 OTA 升级包的方式也有多种。

以搭载了 OpenHarmony 的智能硬件为例，它的 OTA 升级过程如下图所示。

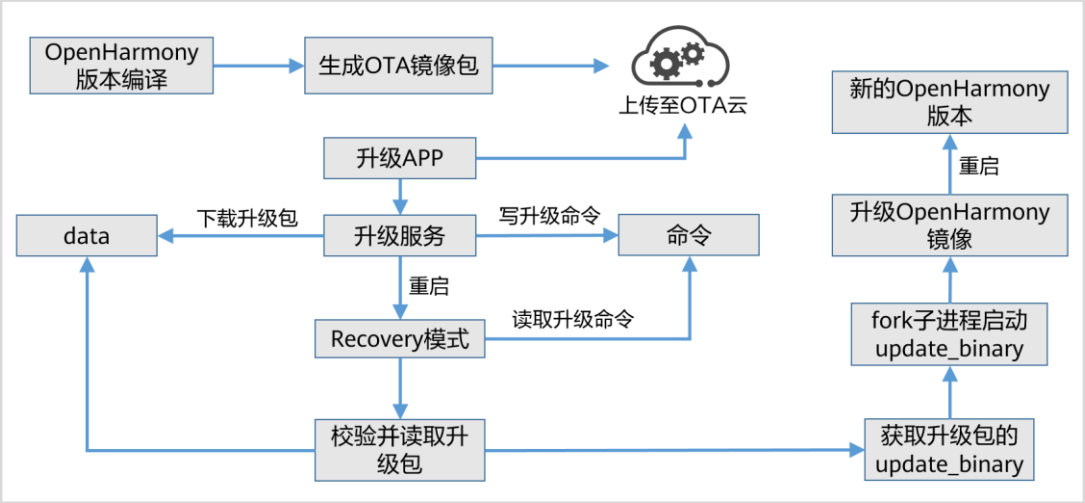

图8-8 OTA 升级开发流程示意图

浏览器方式，用户登陆 OTA 服务器，从中查找并下载需要的升级包。PUSH 方式，厂商推送更新通知，用户确认更新，服务器推送更新包至用户设备，交互简单，传输效率高，针对性强。

在 OTA 升级的过程中，云端有这几个作用：分发 OTA 升级包，对接后台的 Web 界面，提供各终端的状态。

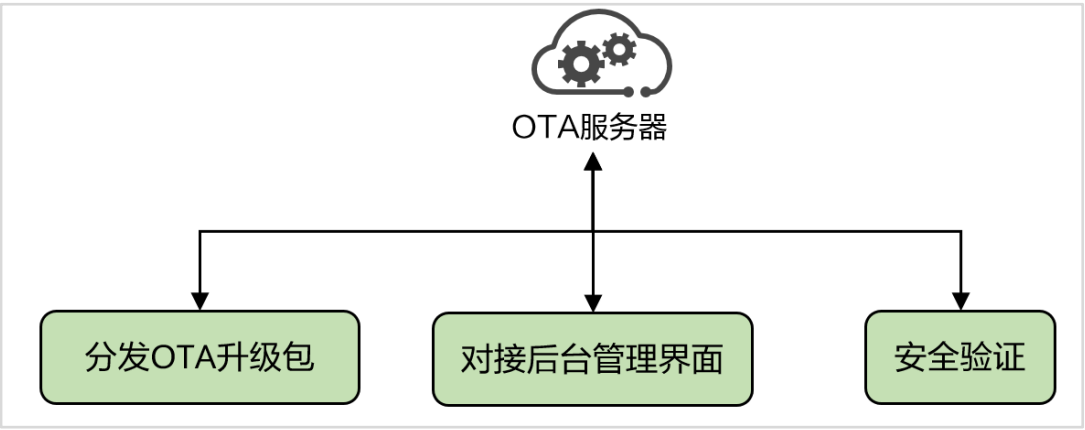

图8-9 云端服务器的作用示意图

在 OTA 的升级过程中，身份认证是极其重要的，多种身份认证的技术手段有助于保证软硬件的安全可信，在 OpenHarmony 的 OTA 升级当中，常用哈希算法进行身份安全验证。

8.3.2 OpenHarmony 的 OTA 升级

在 OpenHarmony 小型系统上进行 OTA 升级实验一般遵循以下操作步骤。

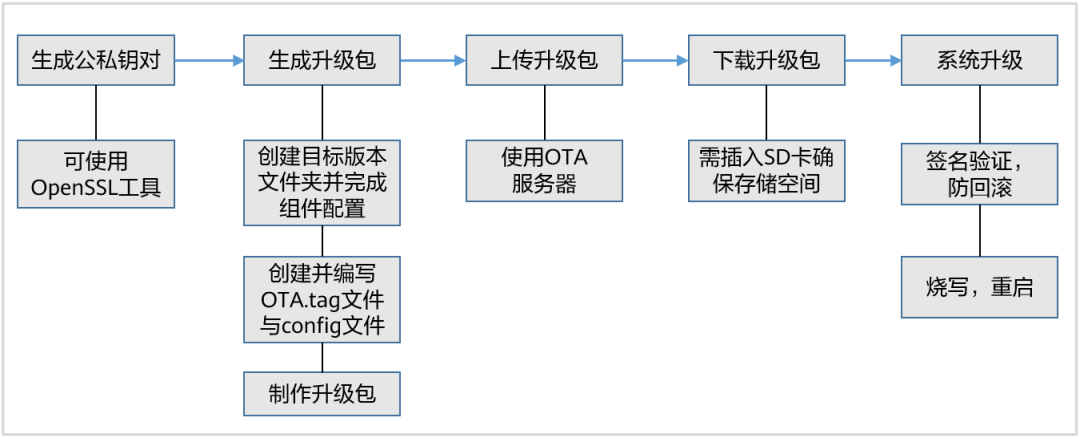

图8-10 小型系统上 OTA 升级操作步骤示意图

标准系统的升级包与小型系统的升级包制作流程有一定的差异，以下是标准系统上的升级包制作过程。

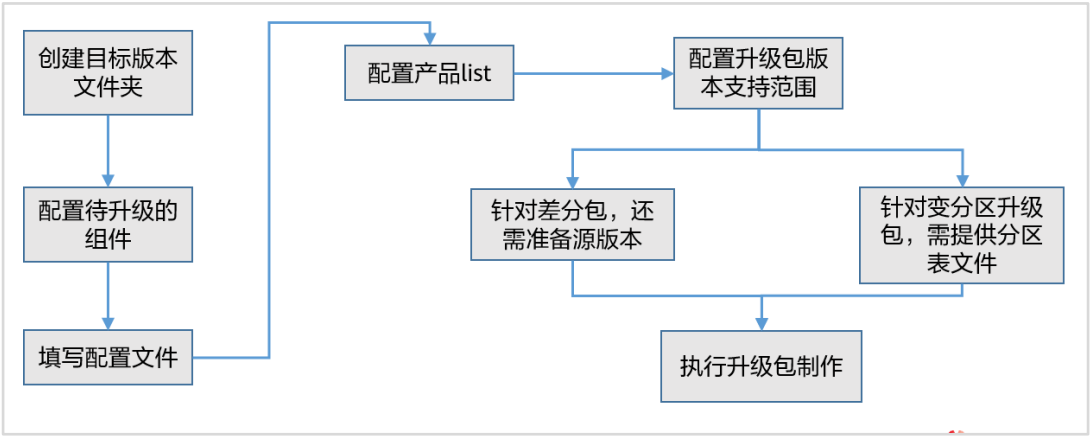

图8-11 标准系统上 OTA 升级操作步骤示意图

具体来说，操作流程可按照以下步骤进行。

第一步，生成升级包需要在 Linux 系统下面进行，开发套件支持基于 Hi3861/Hi3518EV300/Hi3516DV300 芯片的开源套件，在 Linux 开发环境中创建目标版本（target_package）文件夹，文件格式如下：

```
target_package
├── OTA.tag
├── config
├── {component_1}
├── {component_2}
├── ......
├── {component_N}
└── updater_config
    └── updater_specified_config.xml
```

第二步，将待升级的组件，包括镜像文件（例如：rootfs.img 等）等放入目标版本文件夹的根目录下，代替上结构中的{component_N}部分。

第三步，填写“update_config”文件夹中的“updater_specified_config.xml”组件配置文件。组件配置文件“updater_specified_config.xml”的格式如下：

```
<?xml version="1.0"?>
<package>
    <head name="Component header information">
        <info fileVersion="01" prdID="hisi" softVersion="OpenHarmony x.x" date="202x.xx.xx" time="xx:xx:xx">head info</info>
    </head>
    <group name="Component information">
    <component compAddr="ota_tag" compId="27" resType="5" compType="0" compVer="1.0">./OTA.tag</component>
    <component compAddr="config" compId="23" resType="5" compType="0" compVer="1.0">./config</component>
    <component compAddr="bootloader" compId="24" resType="5" compType="0" compVer="1.0">./u-boot-xxxx.bin</component>
    </group>
</package>
```

第四步，创建“OTA.tag 文件”，内容为 OTA 升级包的魔数，固定如下：

```
package_type:ota1234567890qwertw
```

第五步，创建“config 文件”，内容为设置 bootargs 以及 bootcmd 的信息。

例如配置如下：

```
setenv bootargs 'mem=128M console=ttyAMA0,115200 root=/dev/mmcblk0p3 rw rootfstype=ext4 rootwait blkdevparts=mmcblk0:1M
(u-boot.bin),9M(kernel.bin),50M(rootfs_ext4.img),50M(userfs.img)' setenv bootcmd 'mmc read 0x0 0x82000000 0x800 0x4800;bootm 0x82000000'
```

第六步，执行升级包制作命令。

```
python build_update.py ./target_package/ ./output_package/ -pk ./rsa_private_key3072.pem -nz -nl2x
```

至此，一个轻量和小型系统上的 OTA 升级包就制作完成了，后续可用此升级包进行 OTA 升级。

8.4 XTS 子系统

8.4.1 XTS 简介

XTS(X Test Suite)子系统是 OpenHarmony 生态认证测试套件的集合，当前包括 acts(application compatibility test suite)应用兼容性测试套件，后续会拓展 dcts(device compatibility test suite)设备兼容性测试套件等。

XTS 子系统当前包括 acts 与 tools 软件包：

acts，存放 acts 相关测试用例源码与配置文件，其目的是帮助终端设备厂商尽早发现软件与 OpenHarmony 的不兼容性，确保软件在整个开发过程中满足 OpenHarmony 的兼容性要求。

tools，用于存放 acts 相关测试用例开发框架。

8.4.2 XTS 目录

```
/test/xts
├── acts                    # 测试代码存放目录
│   └── subsystem           # 标准系统子系统测试用例源码存放目录
│   └── subsystem_lite      # 轻量系统、小型系统子系统测试用例源码存放目录
│   └── BUILD.gn            # 标准系统测试用例编译配置
│   └── build_lite          # 轻量系统、小型系统测试用例编译配置存放目录
│       └── BUILD.gn        # 轻量系统、小型系统测试用例编译配置
└── tools                   # 测试工具代码存放目录
```

8.4.3 XTS 认证开发示例（轻量系统）

当前使用的测试框架是 hctest，hctest 测试框架支持使用 C 语言编写测试用例，是在开源测试框架 unity 的基础上进行增强和适配。用例目录规范：测试用例存储到 test/xts/acts 仓中，src 目录是用例编写样例。测试框架的引用可以通过 hctest.h 头文件来进行。

```
#include "hctest.h"
```

之后使用宏定义 LITE_TEST_SUIT 定义子系统、模块、测试套件名称，定义 Setup 与 TearDown，命名方式需要注意，测试套件名称+Setup，测试套件名称+TearDown，Setup 与 TearDown 必须存在，可以为空函数。

最后使用宏定义 LITE_TEST_CASE 写测试用例，包括三个参数：测试套件名称，测试用例名称，用例属性（测试类型、用例粒度、用例级别）。

```
LITE_TEST_CASE(IntTestSuite, TestCase001, Function | MediumTest | Level1)
{
  //do something
};
```

使用宏定义 RUN_TEST_SUITE 注册测试套件

```
RUN_TEST_SUITE(IntTestSuite);
```

测试模块的配置文件(BUILD.gn)样例：每个测试模块目录下新建 BUILD.gn 编译文件，用于指定编译后静态库的名称、依赖的头文件、依赖的库等，acts 下 BUILD.gn 增加编译选项。需要将测试模块加入到 acts 目录下的编译脚本中，编译脚本路径：test/xts/acts/build_lite/BUILD.gn。测试套件编译命令。随版本编译，debug 版本编译时会同步编译 acts 测试套件，acts 测试套件编译中间件为静态库，最终链接到版本镜像中 。

8.4.4 C 语言用例执行指导（适用于轻量系统产品用例开发）

将版本镜像烧录进开发板。

测试步骤：

1. 使用串口工具登录开发板，并保存串口打印信息。
2. 重启设备，查看串口日志。

测试结果分析指导：

基于串口打印日志进行分析；

每个测试套件执行以 Start to run test suite 开始，以 xx Tests xx Failures xx Ignored 结束。

8.4.5 C++语言用例开发编译指导（适用于小型系统、标准系统用例开发）

当前使用的测试框架是 hcpptest，hcpptest 测试框架是在开源的 googletest 测试框架的基础上进行的增强和适配。

1. 规范用例目录：测试用例存储到 test/xts/acts 仓中。

```
├── acts
│ └──subsystem_lite
│ │ └── module_posix
│ │ │ └── BUILD.gn
│ │ │ └── src
│ └──build_lite
│ │ └── BUILD.gn
```

2. 测试模块 src 下用例编写样例：

步骤 1　引用测试框架：

需要引用 gtest.h 如：#include "gtest/gtest.h"

```
#include "gtest/gtest.h"
```

步骤 2　定义 Setup 与 TearDown

```
using namespace std;
using namespace testing::ext;
class TestSuite: public testing::Test {
protected:
// Preset action of the test suite, which is executed before the first test case
static void SetUpTestCase(void){
}
// Test suite cleanup action, which is executed after the last test case
static void TearDownTestCase(void){
}
// Preset action of the test case
virtual void SetUp()
{
}
// Cleanup action of the test case
virtual void TearDown()
{
}
};
```

步骤 3 使用宏定义 HWTEST 或 HWTEST_F 写测试用例

普通测试用例的定义：HWTEST（测试套名称， 测试用例名称， 用例标注）。

包含 SetUp 和 TearDown 的测试用例的定义 ： HWTEST_F（测试套名称， 测试用例名称，用例标注）。

宏定义包括三个参数：测试套件名称，测试用例名称，用例属性（测试类型、用例粒度、用例级别）。

```
HWTEST_F(TestSuite, TestCase_0001, Function | MediumTest | Level1) {
// do something
}
```

3. 测试模块下用例配置文件（BUILD.gn）样例：

每个测试模块目录下新建 BUILD.gn 编译文件，用于指定编译后可执行文件的名称、依赖的头文件、依赖的库等；具体写法如下。每个测试模块将独立编译成.bin 可执行文件， 该文件可直接 push 到单板上进行测试。

举例：

```
import("//test/xts/tools/lite/build/suite_lite.gni")
hcpptest_suite("ActsDemoTest") {
    suite_name = "acts"
    sources = [
        "src/TestDemo.cpp"
    ]


    include_dirs = [
        "src",
        ...
    ]
    deps = [
        ...
    ]
    cflags = [ "-Wno-error" ]
}
```

4. acts 目录下增加编译选项（BUILD.gn）样例：

将测试模块加入到 acts 目录下的编译脚本中，编译脚本为：test/xts/acts/build_lite/BUILD.gn。

```
 lite_component("acts") {
...
else if(board_name == "liteos_a") {
        features += [
            ...
```

```
            "//xts/acts/subsystem_lite/module_posix:ActsDemoTest"
        ]
    }
}
```

5. 测试套件编译命令。

随版本编译，debug 版本编译时会同步编译 acts 测试套件。

8.4.6 C++语言用例执行指导（适用于小型系统、标准系统用例开发）

目前的用例执行采用 nfs 共享的方式，mount 到单板去执行。

环境搭建:

1. 使用有限网线或无线将开发板与 PC 进行连接。
2. 开发板配置 IP、子网掩码、网关，确保开发板与 PC 处于同一个网段。
3. PC 安装 nfs 服务器并完成注册，启动 nfs 服务。
4. 开发板配置 mount 命令，确保开发板可以访问 PC 端的 nfs 共享文件。

格式：mount [nfs 服务器 IP]:[/nfs 共享目录] [/开发板目录] nfs

举例：

```
mount 192.168.1.10:/nfs /nfs nfs
```

用例执行

测试套件执行 ActsDemoTest.bin 触发用例执行，基于串口打印日志进行分析。

8.4.7 JS 语言用例开发指导（适用于标准系统）

当前使用的测试框架是 HJSUnit，用于支撑 application 测试（特指基于 JS 应用框架使用 Javascript 语言开发的 APP）进行自动化测试。

用例编写基础语法

测试用例为 js 语言，必须满足 JavaScript 语言编程规范:

表8-1 JavaScript 语言编程规范

用例语法	描述	要求
beforeAll	测试套级别的预置条件，在所有测试用例开始前执行且仅执行一次，支持一个参数：预置动作函数	可选
afterAll	测试套级别的清理条件，在所有测试用例结束后执行且仅执行一次，支持一个参数：清理动作函数	可选
beforeEach	测试用例级别的预置条件，在每条测试用例开始前执行，执行次数与 it 定义的测试用例数一致，支持一个参数：预置	可选

	动作函数	
afterEach	测试用例级别的清理条件，在每条测试用例结束后执行，执行次数与 it 定义的测试用例数一致，支持一个参数：清理动作函数	可选
describe	定义一个测试套，支持两个参数：测试套名称和测试套函数； describe 支持嵌套，每个 describe 内均可以定义 beforeAll 、beforeEach 、afterEach 和 afterAll	必选
it	定义一条测试用例，支持三个参数：用例名称，过滤参数和用例函数 备注： 过滤参数：过滤参数为一个 32 位的 Int 类型参数，0 位置1表示不筛选、默认执行；0-10 位 置1表示测试用例类型；16-18 位 置1表示测试用例规模；24-28 位 置1表示测试层级。 测试用例类型。置位0-10分别表示：FUNCTION 方法类测试、PERFORMANCE 性能类测试、POWER 功耗类测试、RELIABILITY 可靠性测试、SECURITY 安全合规测试、GLOBAL 整体性测试、COMPATIBILITY 兼容性测试、USER 用户测试、STANDARD 标准测试、SAFETY 安全特性测试，RESILIENCE 压力测试。 测试用例规模。置位16-18分别表示：SMALL 小型测试、MEDIUM 中型测试、LARGE 大型测试。 测试层级。置位24-28分别表示：LEVEL0-0 级测试、LEVEL1-1 级测试、LEVEL2-2 级测试、LEVEL3-3 级测试、LEVEL4-4 级测试。	必选

1. 规范用例目录：测试用例存储到 entry/src/main/js/test 目录。

```
├── BUILD.gn
│ └──entry
│ │ └──src
│ │ │ └──main
│ │ │ │ └──js
│ │ │ │ │ └──default
│ │ │ │ │ │ └──pages
│ │ │ │ │ │ │ └──index
│ │ │ │ │ │ │ │ └──index.js          # 入口文件
│ │ │ │ │ │ └──test                   # 测试代码存放目录
│ │ │ └── resources                   # hap 资源存放目录
│ │ │ └── config.json                 # hap 配置文件
```

2. index.js 示例

```
// 拉起 js 测试框架，加载测试用例
import {Core, ExpectExtend} from 'deccjsunit/index'


export default {
    data: {
        title: ""
    },
    onInit() {
        this.title = this.$t('strings.world');
    },
    onShow() {
        console.info('onShow finish')
        const core = Core.getInstance()
        const expectExtend = new ExpectExtend({
            'id': 'extend'
        })
        core.addService('expect', expectExtend)
        core.init()
        const configService = core.getDefaultService('config')
        configService.setConfig(this)
        require('../../../test/List.test')
        core.execute()
    },
    onReady() {
    },
}
```

3. 单元测试用例示例

```
// Example1: 使用 HJSUnit 进行单元测试
describe('appInfoTest', function () {
    it('app_info_test_001', 0, function () {
        var info = app.getInfo()
        expect(info.versionName).assertEqual('1.0')
        expect(info.versionCode).assertEqual('3')
    })
})
```

9 附录：术语及缩略语

表9-1 术语及缩略语表

缩略语或术语	全称	描述
HPM	HarmonyOS Package Manager	HarmonyOS包管理工具
CMSIS	Cortex Microcontroller Software Interface Standard	微控制器软件接口标准
POSIX	Portable Operating System Interface	可移植操作系统接口
WLAN	Wireless Local Area Network	无线局域网
GPIO	General-purpose input/output	通用型之输入输出
I2C	Inter – Integrated Circuit	集成电路间总线
SPI	Serial Peripheral Interface	串行外设接口
AD	Analog to Digital Convert	模拟-数字信号转换
DA	Digital to Analog Convert	数字-模拟信号转换
IoT	Internet of Things	物联网
DFX	Design for X	面向产品生命周期各环节的设计
ROTS	Real Time Operating System	实时操作系统
API	Application Programming Interface	应用程序编程接口
MCU	Microcontroller Unit	微控制单元
MMU	Memory Management Unit	内存管理单元
UCOS	u control operation system	微型嵌入式实时系统
FAT	File Allocation Table	文件配置表

YAFFS2	Yet Another Flash File System	嵌入式文件系统
RAMFS	Ram File System	基于内存的文件系统
JFFS2	Journalling Flash File System Version2	闪存日志型文件系统第2版
NFS	Network Files System	网络文件系统
PROC	process	操作系统的/proc目录,即proc文件系统
IPC	Inter-Process Communication	进程间通信
VFS	virtual File System	虚拟文件系统
CPU	central processing unit	中央处理器
SCHED_RR	Schedule Round-Robin	时间片轮询调度
SCHED_FIFO	Schedule First Input First Output	先进先出调度
UDP	User Datagram Protocol	用户数据报协议
TCP	Transmission Control Protocol	传输控制协议
DIR	Directory	根目录区
DBR	Dos Boot Record	操作系统引导记录区
MBR	Master Boot Record	主引导分区
HDF	Hardware Driver Foundation	硬件驱动框架
LED	Light Emitting Diode	发光二极管
SD	Secure Digital	安全数字卡
HCS	HDF Configuration Source	HDF驱动框架的配置描述源码
HC-GEN	HDF Configuration Generator	HCS配置转换工具
UART	Universal Asynchronous Receiver/Transmitter	通用异步收发传输器
SDA	SerialData	串行数据线
SCL	SerialClock	串行时钟线

ACK	Acknowledge character	确认字符
SCLK	System clock	系统时钟信号
MISO	SPI Bus Master Input/Slave Output	SPI 总线主输入/从输出
MOSI	SPI Bus Master Output/Slave Input	SPI 总线主输出/从输入
SDIO	Secure Digital Input and Output	安全数字输入输出接口
GPS	Global Positioning System	全球定位系统
RTC	real-time clock	实时时钟
ADC	Analog to Digital Converter	模数转换器
PWM	Pulse Width Modulation	脉冲宽度调制
OTA	Over the Air	远程升级
Gn	Generate ninja	ninja生成器
FA	Feature Ability	代表有界面的Ability，用于与用户进行交互
IP	Internet Protocol	网际互连协议
SAMGR	/samgr	HarmonyOS系统服务框架子系统
OEM	Original Equipment Manufacturer	原始设备制造商
DMA	Direct Memory Access	直接存储器访问
AR	Augmented Reality	增强现实
VR	Virtual Reality	虚拟现实技术
FPS	Frames Per Second	帧速率
AI	Artificial Intelligence	人工智能
UDID	Unique Device Identifier	设备的唯一设备识别符
DFR	Design for Reliability	可靠性
DFT	Design for Testability	可测试性
XTS	/xts	HarmonyOS生态认证测试

		套件的集合
acts	application compatibility test suite	应用兼容性测试套件
AP	Access Point	无线接入点
BIOS	Basic Input Output System	基本输入输出系统
JTAG	Joint Test Action Group	联合测试工作组
GCC	GNU Compiler Collection	GNU编译器套件
GNU	GNU's Not Unix!	GNU是一个操作系统，其内容软件完全以通用公共许可证的方式发布