**OpenHarmony竞赛训练营**

**基于边缘的异构多功能AI推理集群**

|  |  |
| --- | --- |
| 学校名称： | 南开大学 |
| 团队名称： | 拿奖公布蟹黄堡秘方队 |
|  |  |
| 队长： | 邹先予 |
| 队员1： | 岳志鑫 |
| 队员2： | 林雪松 |

OpenHarmony竞赛训练营组委会

2024年9月

# 摘 要

随着数据体量的急剧膨胀，各类应用对计算的需求已远远超出了传统CPU的能力所及，尤其面对复杂的模块调度和数据处理，单一的芯片较为乏力。而所谓异构计算（Heterogeneous computing，又称为异质运算），就是将CPU、GPU、FPGA、ASIC等不同制程架构、不同指令集、不同功能的计算单元，组合起来形成一个混合的计算系统，实现相互联通，充分调用硬件多个处理器，是面向性能功耗优化的系统级方案。与传统的通用计算芯片相比，异构架构具有高性能、低功耗等显著优点。即使新的算法出来，异构架构也可以依赖通用计算单元加以支持，不至于很快就被淘汰，延长了芯片的生命周期。所以在采用先进工艺所需要的投入越来越大的情况下，异构计算会成为业界主流的选择。

人工智能（AI）涵盖了图像识别、自然语言处理、语音识别等多种应用。这些技术在安防监控、智能家居、医疗健康和工业自动化等领域展现了巨大的潜力。随着物联网和边缘计算的发展，嵌入式系统成为AI应用的重要承载平台。

本文分析了异构计算架构的优势，并实现了不同AI模型应用在异构嵌入式开发板上的容器适配、异构数据传输和任务调度策略等，提高模型的运行效率和处理数据的多样性、准确性。此外，我们讨论了在资源受限条件下实现实时数据处理的挑战，并提出了相应的解决方案。通过图像推理识别的集群化应用展示了基于边缘的异构多功能AI推理集群的开发流程及其可行性验证，图像识别应用涵盖前端显示、视频解码、推理处理和集群调度等部分，利用嵌入式集群中的异构计算资源来处理不同的任务。例如使用 NPU 进行高效的神经网络推理，通过 CPU 进行图像绘制和逻辑处理。我们的项目可以在延迟不超过30ms的情况下解析视频流并使用硬件资源进行图像识别，灵活处理视频设备传入视频的不稳定情况和各部分传输间的延迟误差，并在前端页面多线程显示视频。通过资源的有效调度和利用，使得系统能够在保持高性能和高响应速度的同时，降低能耗和成本，满足更广泛的应用需求。具有实时处理、低延迟、部署灵活的特点。

本项目能够实时处理多个视频流并进行推理显示，通过异构处理和自动化调度，实现低延迟的视频处理和推理结果反馈，适用于安防监控、智能交通、物体识别等智能物联网场景。集群中的开发板功能分工明确，可以根据需求灵活配置和扩展，还提供了bert、resnet50、mobilenetv3、deeplabv3和yolov5等ai推理应用，满足多种数据的处理要求，提高了系统的整体适应性和可扩展性。

**关键词：异构计算;嵌入式计算机集群;AI应用;任务调度;**

目 录

[摘 要 2](#_Toc147761347)

[1 设计需求分析 4](#_Toc147761348)

[1.1 主要解决问题 4](#_Toc147761349)

[2 特色与创新 4](#_Toc147761351)

[3 设计描述 4](#_Toc147761352)

[3.1 总体设计 4](#_Toc147761353)

[3.2 实现思路 4](#_Toc147761354)

[3.3 系统结构 4](#_Toc147761355)

[3.3.1 模块划分 4](#_Toc147761356)

[3.3.2 系统架构说明 4](#_Toc147761357)

[3.3.3 文件结构 4](#_Toc147761358)

[3.4 模块功能描述 5](#_Toc147761359)

[3.4.1 agent模块 5](#_Toc147761360)

[3.4.2 scheduler服务部署调度模块 5](#_Toc147761361)

[3.5 业务/实现流程说明 5](#_Toc147761363)

[3.5.1 用例处理流程 5](#_Toc147761364)

[3.6 接口描述 5](#_Toc147761366)

[3.7 UI设计 6](#_Toc147761369)

[4 其他](#_Toc147761370)  6

[4.1 成员分工 6](#_Toc147761371)

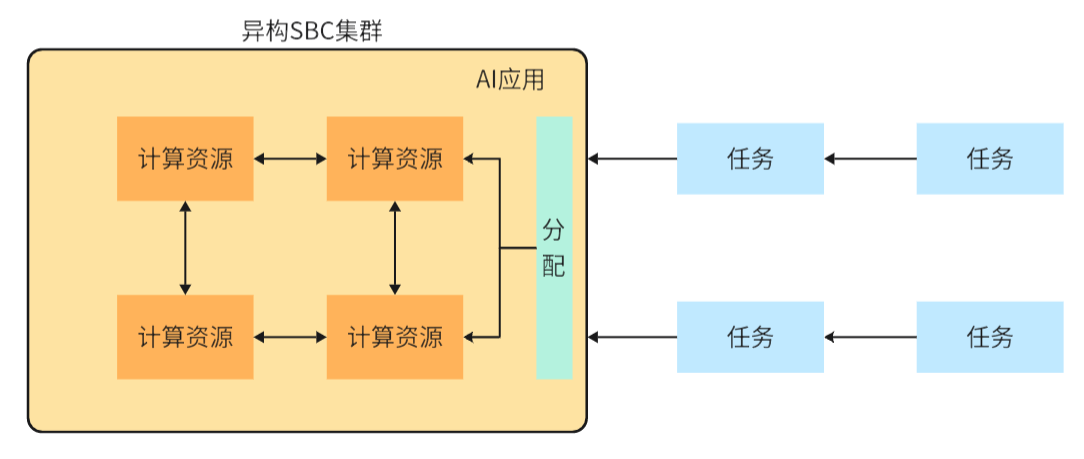
[4.2 困难与思考 6](#_Toc147761372)

[4.3 参考 6](#_Toc147761373)

# **1**设计需求分析

目前端侧设备对于AI推理的需求快速增加，AI推理任务复杂程度提升，对算力的需求超出了其承载极限。并且在嵌入式场景下，传统的数据中心AI推理方式由于网络的不可靠性无法使用。因此需要在端侧实现类似云计算的**端侧AI推理集群**，在不依靠互联网的情况下满足嵌入式设备的AI推理需求（如无人机阵列等）。

不同的端侧设备属性不同，为了同时兼顾功能和能耗，**异构计算**是一个可行的选择。通过组合不同的SoC可以取各家之长，如可以组合低功耗低AI算力与高功耗高AI算力的开发板，通过低功耗配置功能在满足算力要求的情况下降低端侧异构AI推理集群的功耗。



## 1.1主要解决问题

该作品构建了一个基于边缘的异构多功能AI推理集群，支持边缘计算常用的计算和AI推理任务，如视频编解码、目标检测、文本分类和图像分类等，并尝试实现AI算力智能调度算法。为了展示集群功能，该作品基于OpenHarmony设计了一个实时多路视频流目标检测APP，并使用集群算力进行视频编解码和AI推理，最终在APP中实时展示推理结果。

这里给出一个示例场景：AI智能算力箱。AI算力箱其中包含两个模块，分别是热插拔算力模块，也就是前面介绍的端侧异构AI推理集群。和UI交互模块。UI交互模块比较简单，就是和算力箱操作员交互的接口。重点是这个热插拔算力模块。其中由管理节点和异构计算节点组成，管理节点是集群的大脑，负责统筹规划集群的算力并负责调度。各个异构计算节点支持热插拔，可以按需配置。除此之外管理节点还需要对全局算力进行动态评估，动态感知可用算力，给出全局算力占用情况，以便操作员可以在算力过剩的情况下移除计算节点，在算力不足的时候加入计算节点。其他设备通过无线局域网连接到AI智能算力箱，使用它提供的AI推理服务。

文本

中度可信度描述已自动生成

# **2**特色与创新

本项目采用了异构集群与AI推理应用相结合的方式，以低功耗、边缘可用性为主提供AI算力支撑，并成功的实现了一个可行性验证测试项目，且针对资源利用率获取、任务分配等编写了一套监控程序，在高并发或任务种类较多时实现自动任务分配或任务迁移等策略，为端侧AI推理程序在嵌入式场景下利用多种资源进行计算提供了经验。

# **3**设计描述

系统功能首先按照算力提供者（内部系统）和算力使用者（外部系统）进行划分。算力提供者即异构AI推理集群。算力使用者通过调用集群API使用集群算力。

系统分解主要按照功能进行分解。

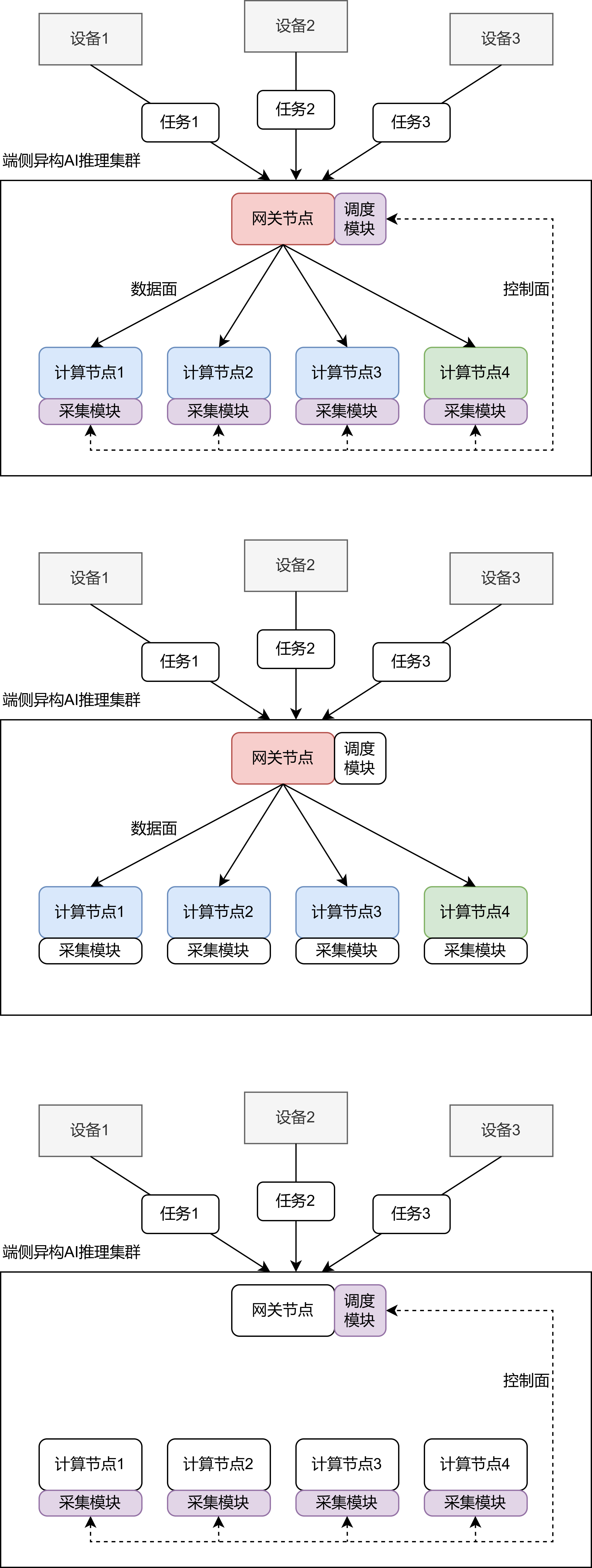
内部系统功能总体可以分为两大类，分别是控制面功能和数据面功能。控制面功能主要解决算力统计、任务调度和功耗控制问题。数据面功能主要为视频编解码、AI推理任务。

外部系统功能各异，由开发者自行分解。

## 3.1总体设计

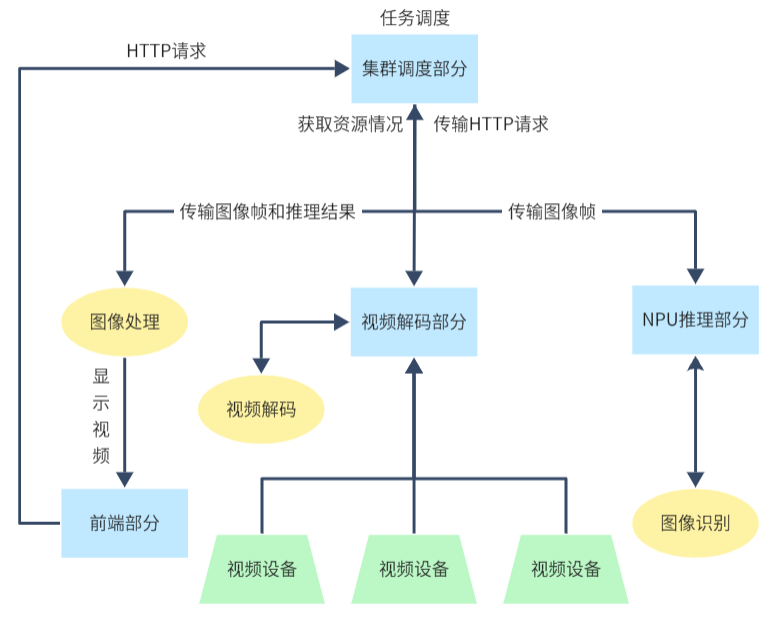
系统功能首先按照算力提供者（内部系统）和算力使用者（外部系统）进行划分。

内部系统功能总体可以分为两大类，分别是控制面功能和数据面功能。



控制面功能主要分为两个模块，分别为schedule模块和agent模块 ，schedule模块部署在网关节点，用于统筹规划集群的算力，是集群的“大脑”。agent模块部署在各个计算节点上，用于接收网关节点的控制命令并上报算力信息。数据面功能按计算任务类型进行分解，每个计算任务使用Docker容器封装，在调用相关任务时按需启动。如视频编解码模块、目标检测模块、图像分类模块，各个模块之间没有依赖，独立提供计算功能，对外暴露各自功能相关接口。

外部系统目前只提供一个基于OpenHarmony的实时多路视频流目标检测APP，其作为一个独立模块进行开发。目标检测APP流程图如下所示。



## 3.2

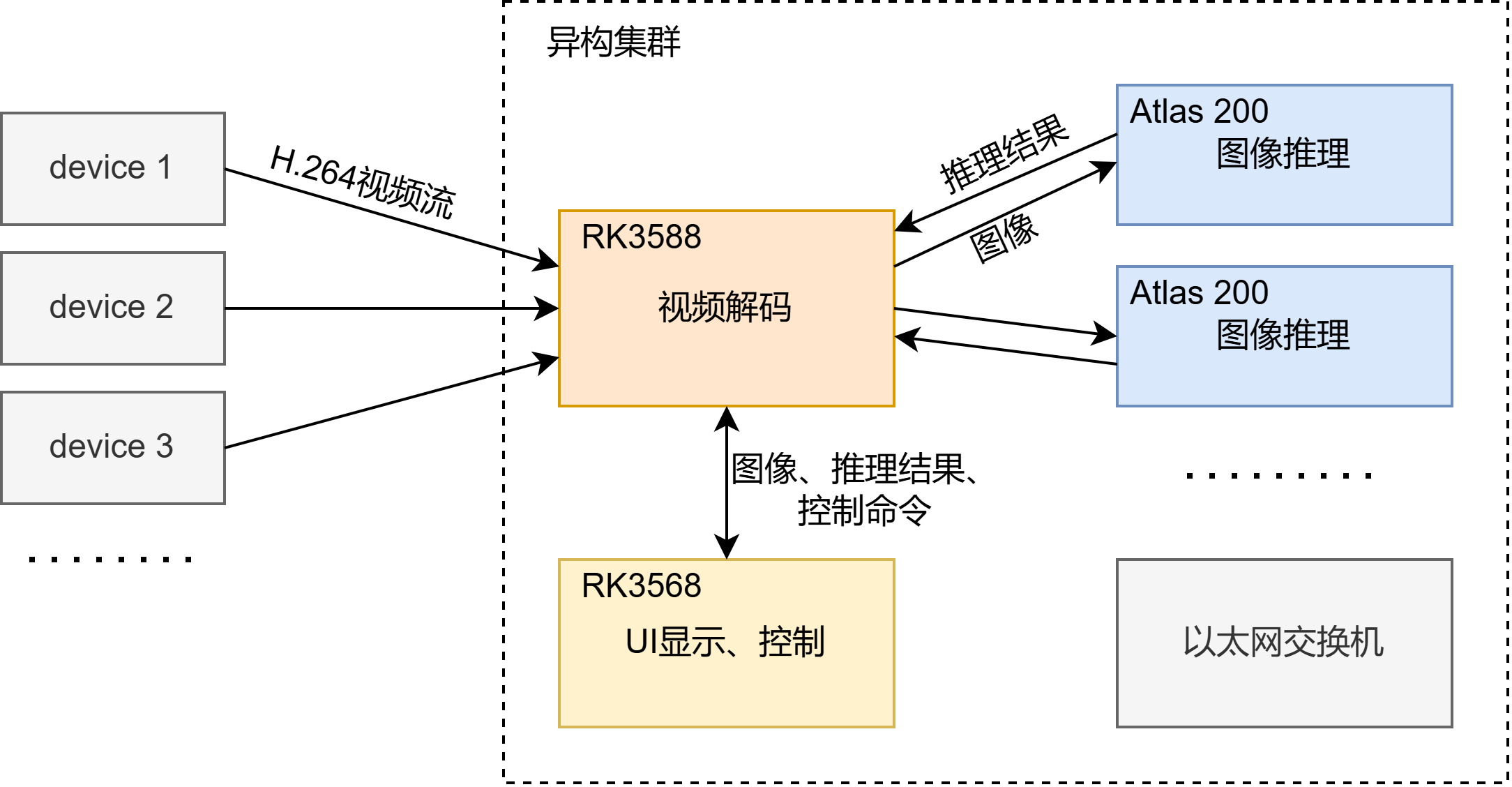
## 3.3实现思路

该工作在开发前首先对各个模块的功能和接口进行了定义，因此各个模块的开发工作无依赖关系，各模块由多人并行开发完成。

## 3.4系统结构

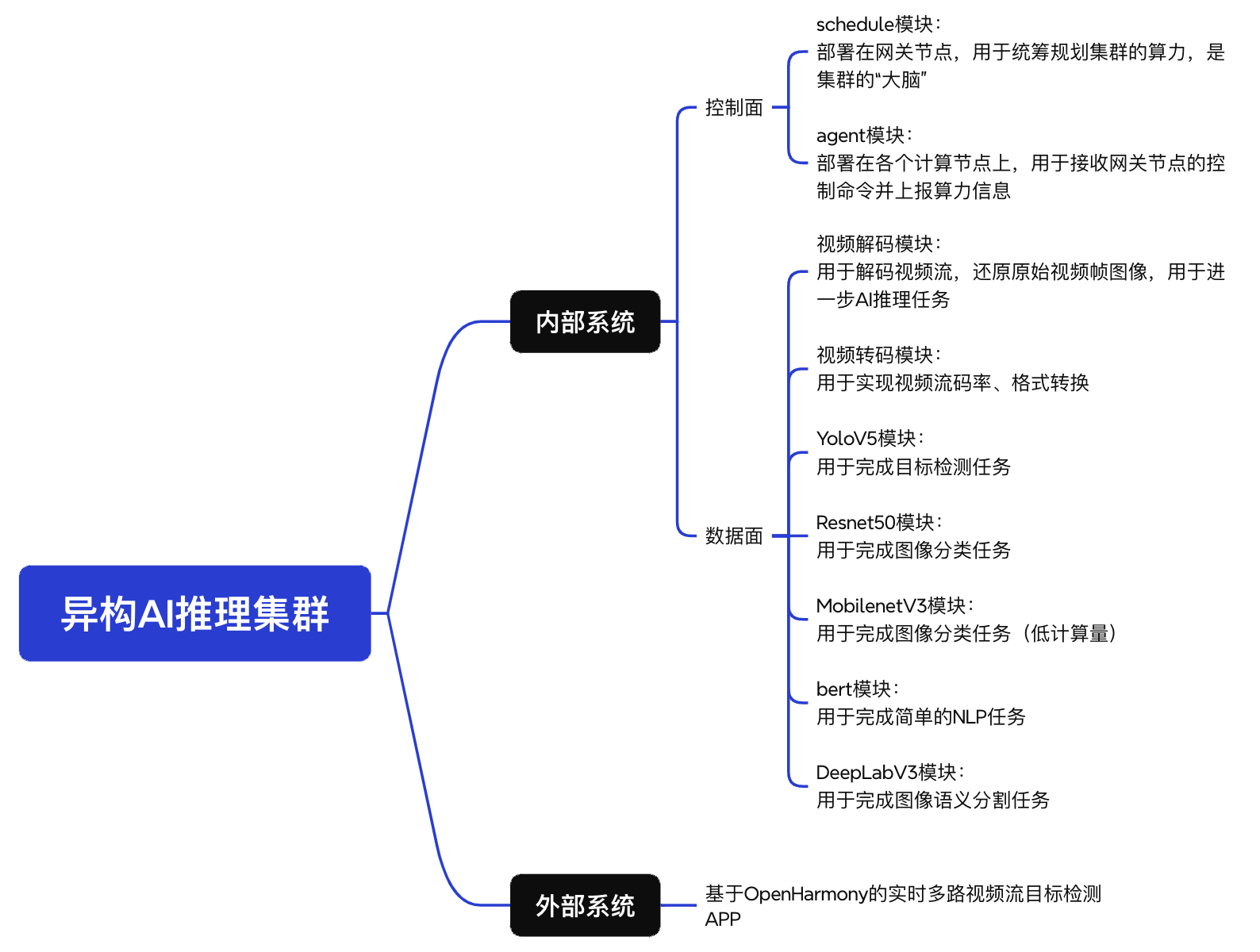
集群由四部分组成，分别是前端部分，NPU推理部分，视频解码部分和集群调度部分，各个部分具有各自的AI应用及功能，可以各自独立完成单独AI任务，也可以集群化共同完成复杂任务。

下图是我们的一个硬件组成结构，外设将视频流推流到异构AI推理集群中，首先通过RK3588开发板进行视频解码，然后将视频帧发送到Atlas 200开发板，使用Yolo-V5模型进行目标检测，然后返回推理结果。RK3568上面运行OpenHarmony系统，和RK3588进行交互，负责UI显示和控制逻辑，所有设备通过以太网交换机连接到一起，组成一个局域网，不需要访问互联网就可以实现AI推理。



### 3.3.1模块划分

主要分为内部系统与外部系统两个系统，内部系统包括schedule模块和agent模块，分别在控制层面进行控制，数据层面包括视频转码和视频解码媒体应用，Yolov5，Resnet50，Mobilenetv3，Bert和Deeplabv3等AI应用。外部系统主要是前端显示程序。



### 3.3.2系统架构说明

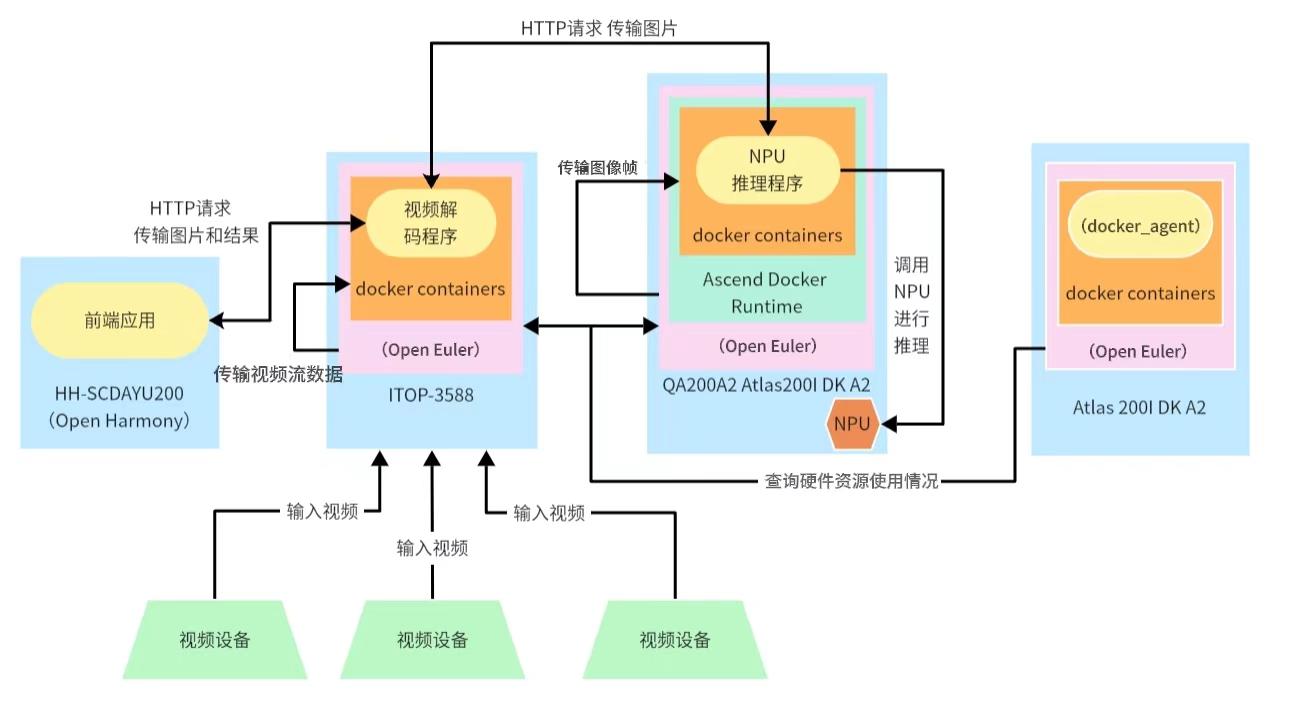
异构集群内包含多种运算资源，为了应用多种功能的ai推理应用，将每个ai推理容器都进行容器化适配，开发板都部署docker容器，便于系统迅速的进行任务分配和容器启动。使用一个图像识别应用，接收多个视频设备传入的视频，分别通过视频解码和图像推理识别的过程，最后用一个开发板作为前端进行展示。

前端部分：DAYU200开发板部署Open Harmony 4.0 Release系统，开发一个前端程序向视频解码部分获取后端数据并可视化显示在屏幕上。

集群调度部分：Atlas 200I DK A2开发板安装 docker ，容器内部署docker\_agent程序进行硬件资源的监控和获取。

推理部分：QA200A2 Atlas200I DK A2开发板部署 OpenEuler 22.03 LTS SP1 和docker，接收视频解码部分发送的HTTP 请求后拉起docker并运行推理程序进行推理，将推理结果返回给视频解码部分。

视频解码部分：iTOP-3588 开发板部署 OpenEuler 22.03 LTS SP1 和docker，在容器内部署多路解码程序，将收到的视频设备的视频解码为帧图像发送给NPU推理部分，将NPU推理部分返回的结果传输给前端部分。



### 3.3.3文件结构

仓库文件描述

├─ 📁atlas200dk

│ ├─ 📁bert

│ │ ├─ 📁acllite // 华为Ascend AI处理器的ACL(Ascend Computing Language)API

│ │ ├─ 📁scripts // 文字识别的数据集和分类标签

│ │ ├─ 📁model // bert的om模型

│ │ ├─ 📁src // bert分类程序代码

│ │ ├─ 📄Dockerfile // 制作包含所有运行环境和库文件的 docker image 语句

│ │ ├─ 📄README.md // 启动docker的挂载命令和程序接口说明等

│ │ └─ 📄repo.txt // 与 dockerfile 配套使用，用于更换 OpenEuler 的 yum 源

│ ├─ 📁deeplabv3

│ │ ├─ 📁acl // 华为Ascend AI处理器的ACL(Ascend Computing Language)API

│ │ ├─ 📁acllite // 华为Ascend AI处理器的ACL(Ascend Computing Language)API

│ │ ├─ 📁model // deeplabv3的om模型

│ │ ├─ 📁src // 调用NPU进行推理的程序代码

│ │ ├─ 📄Dockerfile // 制作包含所有运行环境和库文件的 docker image 语句

│ │ ├─ 📄README.md // 启动docker的挂载命令和程序接口说明等

│ │ └─ 📄repo.txt // 与 dockerfile 配套使用，用于更换 OpenEuler 的 yum 源

│ ├─ 📁mobilenetv3

│ │ ├─ 📁acllite // 华为Ascend AI处理器的ACL(Ascend Computing Language)API

│ │ ├─ 📁model // mobilenetv3的om模型

│ │ ├─ 📁src // 调用NPU进行推理的程序代码

│ │ ├─ 📄Dockerfile // 制作包含所有运行环境和库文件的 docker image 语句

│ │ ├─ 📄README.md // 启动docker的挂载命令和程序接口说明等

│ │ └─ 📄repo.txt // 与 dockerfile 配套使用，用于更换 OpenEuler 的 yum 源

│ ├─ 📁resnet50

│ │ ├─ 📁acl // 华为Ascend AI处理器的ACL(Ascend Computing Language)API

│ │ ├─ 📁model // resnet50的om模型

│ │ ├─ 📁src // 调用NPU进行推理的程序代码

│ │ ├─ 📄Dockerfile // 制作包含所有运行环境和库文件的 docker image 语句

│ │ ├─ 📄README.md // 启动docker的挂载命令和程序接口说明等

│ │ └─ 📄repo.txt // 与 dockerfile 配套使用，用于更换 OpenEuler 的 yum 源

│ ├─ 📁transcoding

│ │ ├─ 📁acl // 华为Ascend AI处理器的ACL(Ascend Computing Language)API

│ │ ├─ 📁data // 可存放输出的视频

│ │ ├─ 📁src // 视频转码的代码

│ │ ├─ 📄Dockerfile // 制作包含所有运行环境和库文件的 docker image 语句

│ │ ├─ 📄README.md // 启动docker的挂载命令和程序接口说明等

│ │ └─ 📄repo.txt // 与 dockerfile 配套使用，用于更换 OpenEuler 的 yum 源

│ ├─ 📁yolov5-infer-cpp

│ │ ├─ 📁acl // 华为Ascend AI处理器的ACL(Ascend Computing Language)API

│ │ ├─ 📁lib64 // 相关库文件

│ │ ├─ 📁model // yolov5的om模型

│ │ ├─ 📁src // 调用NPU进行推理的程序代码

│ │ ├─ 📄CMakeLists.txt

│ │ ├─ 📄Dockerfile // 制作包含所有运行环境和库文件的 docker image 语句

│ │ ├─ 📄README.md // 启动docker的挂载命令和程序接口说明等

│ │ └─ 📄repo.txt // 与 dockerfile 配套使用，用于更换 yum 源

├─ 📁DAYU200

│ └─ 📁serverless

│ ├─ 📁APPScore

│ ├─ 📁entry

│ │ ├─ 📁libs // 存放OpenCV库文件

│ │ ├─ 📁oh\_modules // c++与ArkTS代码的绑定库

│ │ └─ 📁src

│ │ └─ 📁main

│ │ ├─ 📁cpp

│ │ │ ├─ 📁common

│ │ │ │ └─ 📄common.cpp // OpenCV相关代码

│ │ │ ├─ 📁include // OpenCV相关库文件

│ │ │ ├─ 📁types

│ │ │ ├─ 📄CmakeLists.txt

│ │ │ └─ 📄hello.cpp // 实现HTTP请求和响应结果处理

│ │ ├─ 📁ets

│ │ │ ├─ 📁enttyability

│ │ │ ├─ 📁pages

│ │ │ │ └─ 📄index.ets // 前端主要页面代码

│ │ │ └─ 📁util // 工具类

│ │ └─ 📁resources

│ │ └─ 📄module.json // 更改权限代码

│ ├─ 📁hvigor

│ └─ 📁oh\_modules

├─ 📁rk3588

│ ├─ 📁resnet\_rk3588\_v1

│ │ ├─ 📁lib //图像分类程序的lib库

│ │ ├─ 📁model //图像分类程序的模型

│ │ ├─ 📁src //图像分类程序的源代码

│ │ ├─ 📄Dockerfile //制作包含所有运行环境和库文件的 docker image 语句

│ │ ├─ 📄CMakeLists

│ │ └─ 📄README.md //启动docker的挂载命令和程序接口说明等.

│ ├─ 📁video\_decoder\_docker\_multiway\_v3

│ │ ├─ 📁src //视频解码程序的源代码

│ │ ├─ 📄Dockerfile //制作包含所有运行环境和库文件的 docker image 语句

│ │ └─ 📄README.md //启动docker的挂载命令和程序接口说明等.

│ └─ 📁video\_transcoder\_docker\_multiway\_v1

│ ├─ 📁src //视频转码程序的源代码

│ ├─ 📄Dockerfile //制作包含所有运行环境和库文件的 docker image 语句

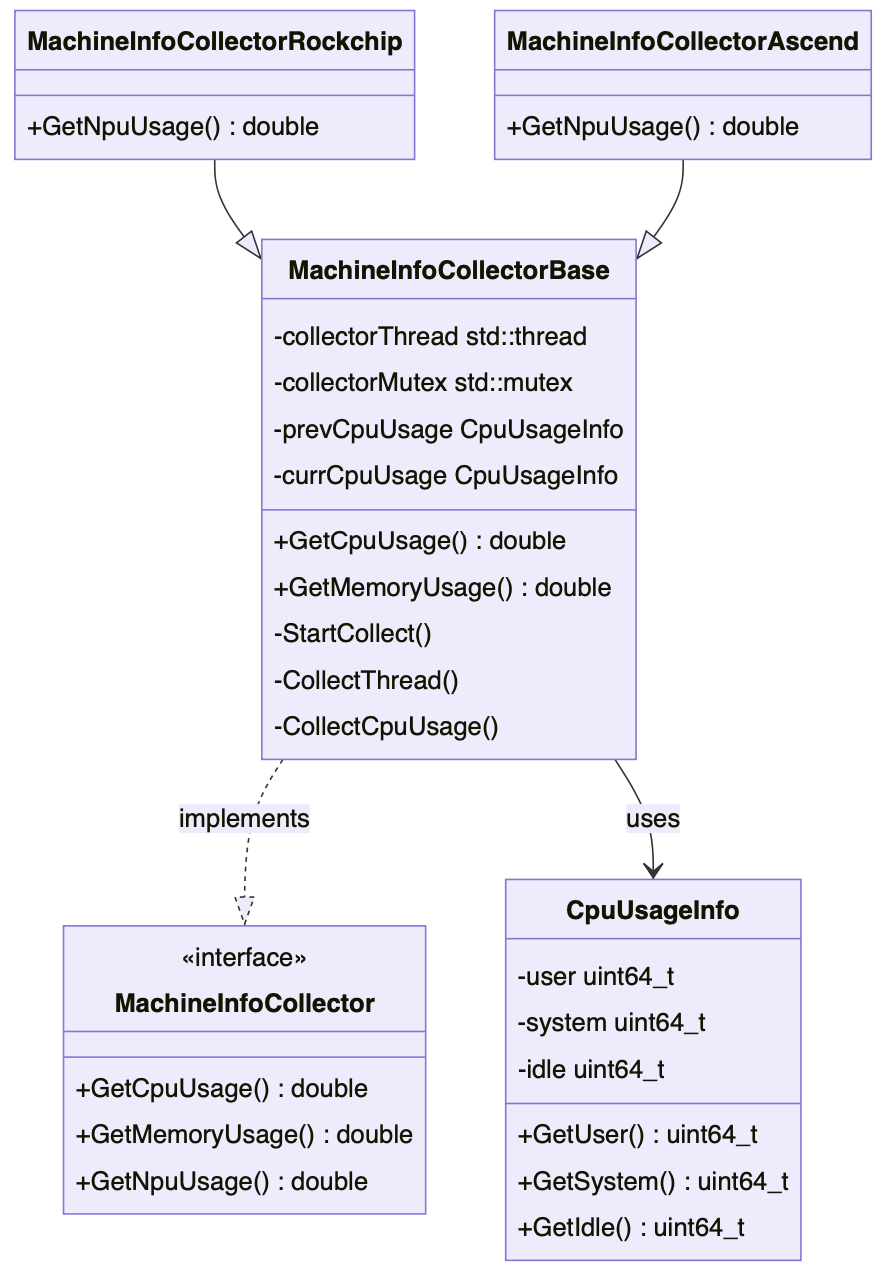
│ └─ 📄README.md //启动docker的挂载命令和程序接口说明等.

└─ 📄README.md //简要说明文档

## 3.53模块功能描述

### 3.4.1 agent模块

考虑到集群的异构特性，不同节点所拥有的硬件资源数量、运算速度、功耗具有较大差异。我们采用agent主动上到给scheduler当前节点的硬件资源信息如mem、gpu、cpu，提供节点资源信息为scheduler提供部署决策依据。



其中MachineInfoCollector作为接口，其中包含cpu、mem、npu资源利用率获取方法。

MachineInfoCollectorBase继承自MachineInfoCollector，实现了cpu、mem资源利用率的获取方法。为了准确获取当前时刻系统cpu利用率，在程序中启动了后台线程实时监控user、system、idle时长，通过微分的方式计算得到准确的利用率。系统mem利用率通过sysinfo系统调用获得。

npu利用率在不同架构的机器上有所不同，因此分为了MachineInfoCollectorRockchip类和MachineInfoCollectorAscend类，其通过调用各npu厂商暴露的资源利用率获取接口，获得npu利用率。

### 3.4.2 scheduler服务部署调度模块

服务部署调度模块区别于任务分配将请求均衡分发到服务节点上，调度算法本质上是将有限的计算内存资源分配给若干服务的部署过程，以达到资源利用率最高，服务分配到最合适的机器上进行运算。

为此服务部署调度算法分为节点信息的收集、服务容器的部署、调整和回收三个部分

i）集群节点资源信息的收集

作为调度器的核心，需要存储待部署任务信息、已部署运行服务信息、节点自身负载和其服务间部署映射关系、当前集群可用节点及支持的可部署任务，基于这些信息才可以合理分配资源实现调度。详细信息如下：

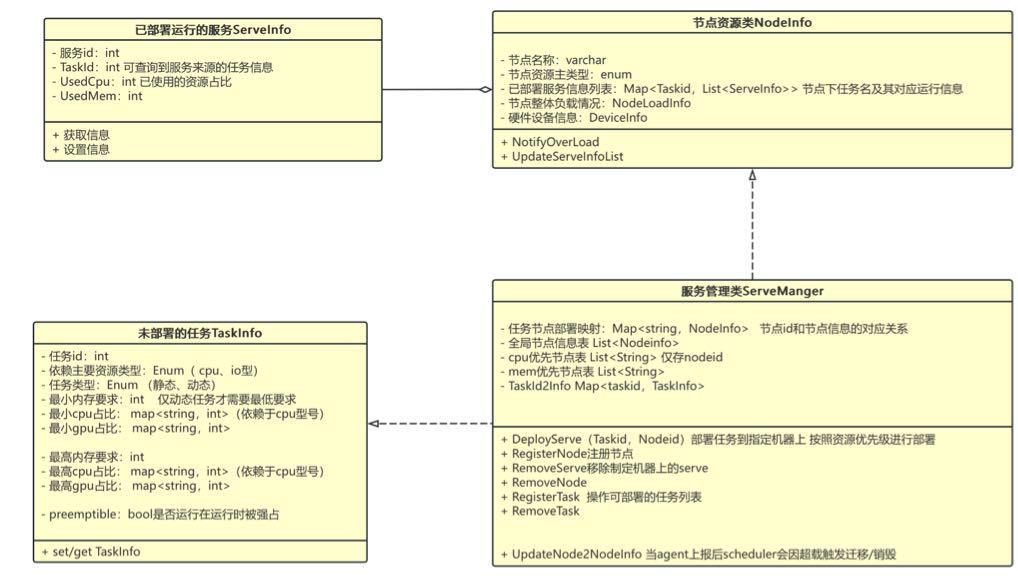
|  |  |
| --- | --- |
| 待部署任务数据 | 详细描述 |
| 任务id | 任务标识 |
| 任务主要依赖资源 | gpu或mem或cpu资源依赖优先 |
| 最小资源要求 | cpu、gpu占比 |
| 最大资源要求 | cpu、gpu占比，存在无上限的任务 |
| 是否允许抢占 | 超载情况下紧急请求可触发抢占资源分配 |

|  |  |
| --- | --- |
| 已部署运行服务数据 | 详细描述 |
| 服务id | 服务标识 |
| 节点名称 | 服务所在的id |
| 任务id | 该服务来源 |
| 已用cpu | cpu占比 |
| 已用mem | 已占用内存 |
| 已用gpu | 已占用gpu |

|  |  |
| --- | --- |
| 集群节点负载信息数据 | 详细描述 |
| 节点名称 | 节点标识 |
| 节点资源主类型 | 节点倾向于cpu或gpu等运算类型服务 |
| 已部署服务信息列表 | TaskId到serveid的映射 |
| 启动服务的信息 | 服务当前运行负载情况（详见iii部分类图设计） |
| 硬件设备信息 | 设备具体型号信息 |

|  |  |
| --- | --- |
| 调度器集群管理数据 | 详细描述 |
| 全局节点信息 | 存储注册所有节点信息 |
| cpu节点优先队列 | 支持cpu计算的节点优先级队列 |
| gpu节点优先队列 | 支持gpu计算的节点优先级队列 |
| 可处理的任务及其已部署的服务映射 | taskid映射到已部署的服务id列表 |
| 支持的任务类型信息 | taskinfo列表 |

ii类图设计）



ii ）服务的部署过程

首先不同类型的服务作为调度的基本单位，对于服务而言其占用的内存是可以确定的，但是对于处理器资源其占用会随请求量而增长。如果全使用静态调度必然会导致机器资源使用超出负载，所以需要动态调度处理动态服务。

我们把服务 按照使用资源是否存在上限分成两种类型

第一种 静态服务: 内存、cpu、gpu等占比按照其压测环境下可负载的的最大额度计算。 此类服务适用于资源估算准确且需要长时间运行。

第二种 动态服务: 占用资源随请求量变化浮动很大，一般一个机器上部署一个服务（服务内部支持多线程 可以达到节点拥有的的资源上限） ，但是我们又不应该使其达到节点负载上限，因为这会严重影响同机器上其他不可抢占服务。

具体调度流程如下：

①初始化

节点向调度器注册节点资源信息；

开发者向调度器添加任务，用于部署；

当请求达到时如果支持，则进入下面的服务部署阶段。

②当节点中服务部署不会超出界限

静态服务 按照其所需资源上限、所依赖资源节点服务质量降序 部署到第一个可装入节点

动态服务 按照其所需资源下限、所依赖资源节点服务质量降序 部署到第一个可装入节点

③当节点中已部署服务超出节点资源负载上限

情况一：新来的服务部署反馈错误，并提示资源请求向集群加入新的计算节点，提升集群的计算上限

情况二：由于动态的服务导致的超出上限，此时触发服务扩容策略或者停止增长（取决于部署的策略）

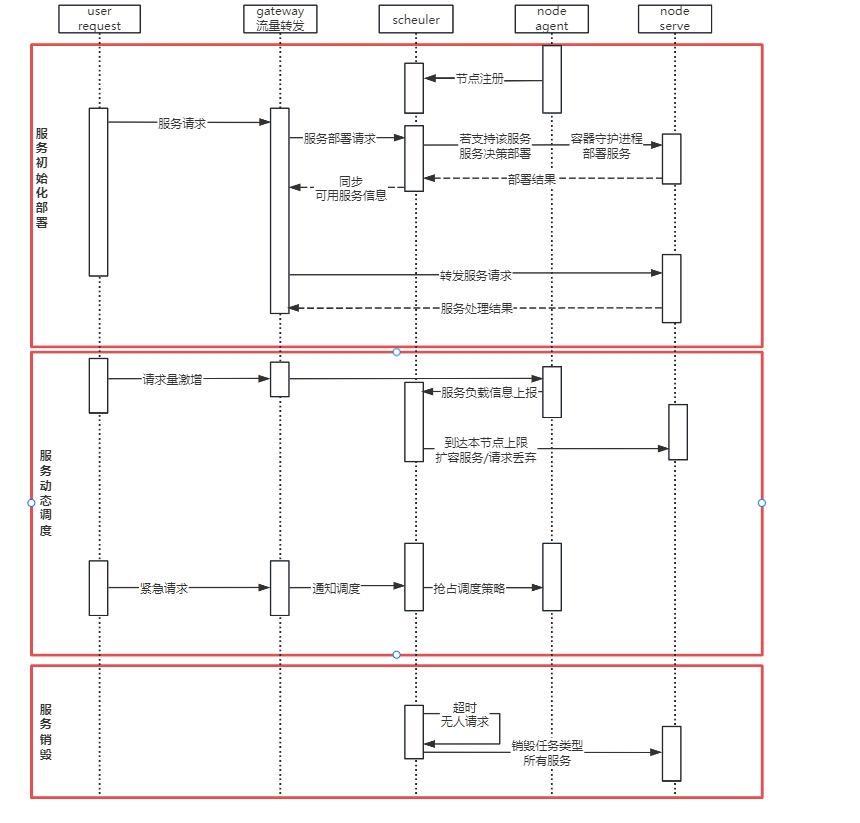
a）服务扩容策略。 在其他节点部署相同服务，在请求负载处注册，请求流量转发，增大可服务请求数量

b）服务停止增长。 服务不在横向扩展，此时超载的服务出现处理超时失败

④当节点已部署服务长期没有访问时

调度器释放对应任务类型下所有节点部署的服务，释放资源

iv）调度过程时序图

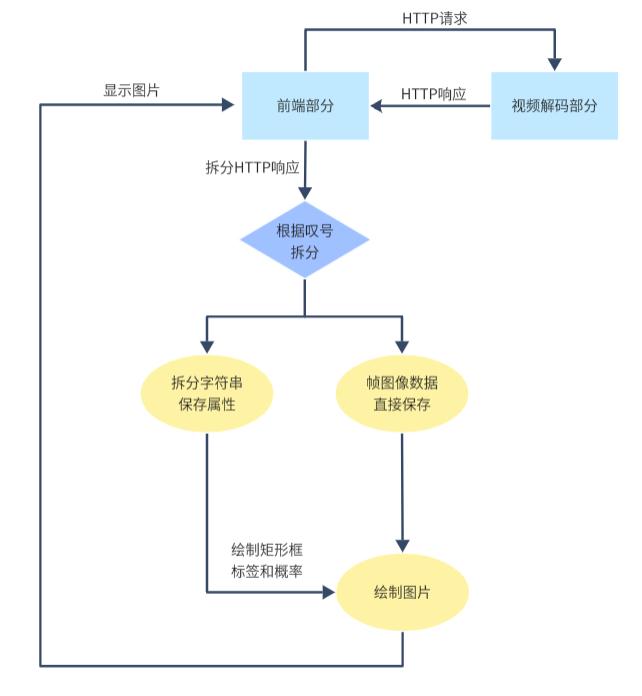


## 业务/实现流程说明

### 3.5.1用例处理流程

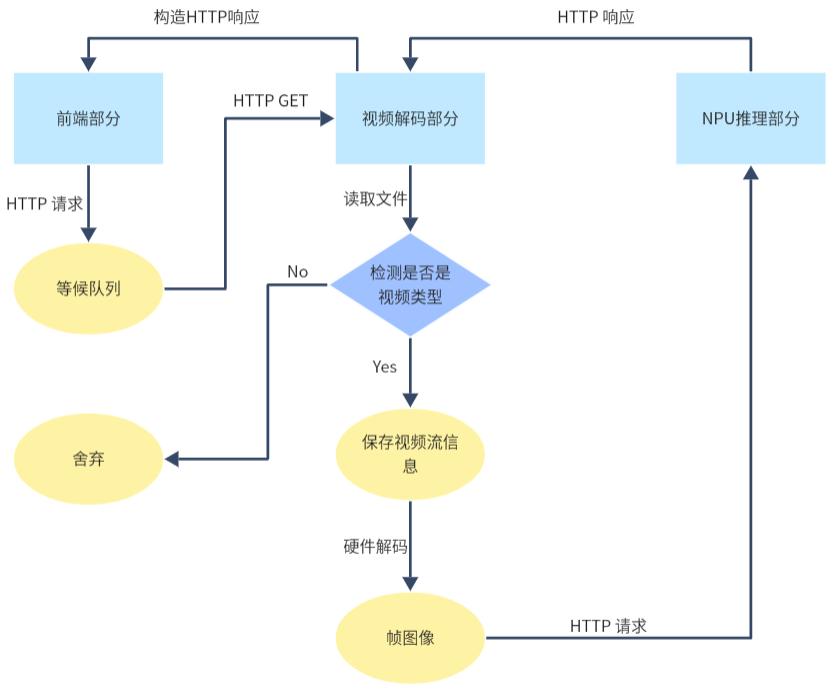
使用搭载了瑞芯微 RK3568 芯片的润和 HH-SCDAYU200 开发板作为前端开发板。在此开发板上部署鸿蒙系统，开发一个前端显示程序。

具体步骤如：发送 HTTP 请求获取结果，将获取的结果拆分为推理结果和图像数据。利用鸿蒙开发的 ArkTS 和Native C++ 将图像数据转为图片显示在 Canvas 画板上，将推理结果内的坐标、标签、概率等属性绘制成矩形框和文字显示在图像上，通过不断发送 HTTP 请求并刷新画板，最终实现视频的效果。开发多线程运行方式，保证多窗口场景下的高效性。具体前端部分流程如



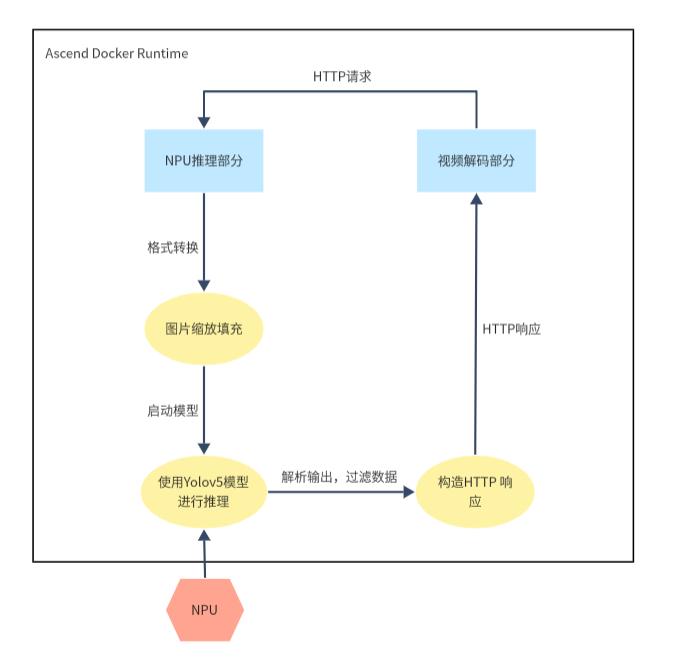
使用搭载了瑞芯微 RK3588 芯片的迅为iTOP-3588开发板作为视频解码开发板，部署OpenEuler和k3s 等系统，在此开发板上开发一个docker内的多线程解析视频的视频流处理程序

具体步骤如：接收到前端部分的HTTP请求后，首先对输入文件进行读取，使用 httplib 库创建 HTTP 服务器，使用 FFmpeg 库来初始化网络，然后打开视频流、查找视频流信息，最后打开解码器，创建缩放上下文，将转换出的帧图像使用 HTTP 传输给NPU推理部分，将NPU推理部分的结果返回给前端部。设置多线程保证任务量集中时仍能保持高效的视频解码处理。具体视频解码部分流程如图所示。

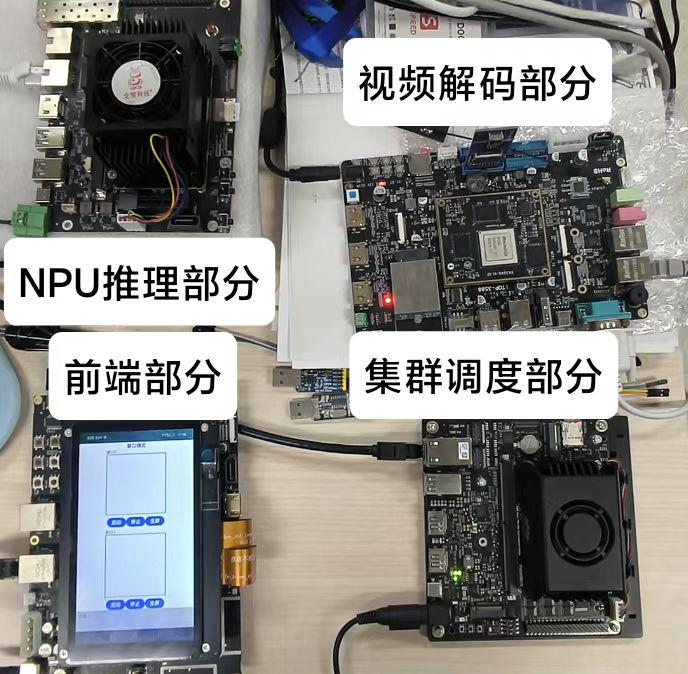


使用 QA200A2 Atlas200I DK A2开发者套件作为NPU推理开发板，部署OpenEuler和k3s 等系统，启动 docker 并利用容器进行 NPU 推理。

具体步骤如：使用C++编写了一个Yolov5模型推理程序，借助Ascend Docker Runtime为所有的训练或推理作业提供昇腾AI处理器（Ascend NPU）容器化支持。首先从HTTP请求中读取图像数据，使用 OpenCV 创建图像矩阵 img，调用华为 Ascend AI 处理器的 ACL（Ascend Computing Language）API交互，用于模型的初始化、加载和执行，然后调用 Yolov5 的模型进行推理，并输出最后的检测信息，开发多线程保证有效的传输和推理。具体NPU推理部分流程如图所示。



前端 APP 选择窗口点击启动按钮发送Http/start请求，视频解码开发板和NPU推理开发板启动并创建容器，启动视频解码程序和NPU推理程序。前端会自动循环发送Http Get请求获取帧图像的数据，并转换为图片的格式，使用Canvas不断把接收到的HTTP 响应的帧图像绘制在对应的窗口区域实现视频效果，点击放大按钮可以观看单个窗口的大屏显示情况。



## 3.7接口描述

使用不同的HTTP地址表示各个功能的接口。

atlas开发板的应用：

**Yolov5**:post访问http://192.168.137.100:8080/predict，请求体为form-data，key为image，value为文件。

**bert**:post请求访问地址为http://ip:6000/textclassify,请求体为form-data,key为text，value为文本。

**deeplabv3**:post请求访问地址为http://ip:6001/segmentation,请求体为form-data,key为image，value为图片

**mobilenetv3:**post请求访问地址为http://ip:5001/mbclassify,请求体为form-data,key为image，value为图片

**resnet50**:post请求访问地址为http://ip:5000/classify,请求体为form-data,key为image，value为图片

rk3588开发板的应用：

**resnet\_rk3588\_v1**:http://ip:2080

**video\_decoder\_docker\_multiway\_v3**:

开始处理视频请求:http://ip:8081/start?url=rtsp://ip2:8554/camera\_test

停止处理视频请求:http://ip/stop?url=rtsp://ip2:8554/camera\_test

获取视频帧请求:

http://ip:8081/video?url=rtsp://ip2:8554/camera\_test

**video\_transcoder\_docker\_multiway\_v1**:

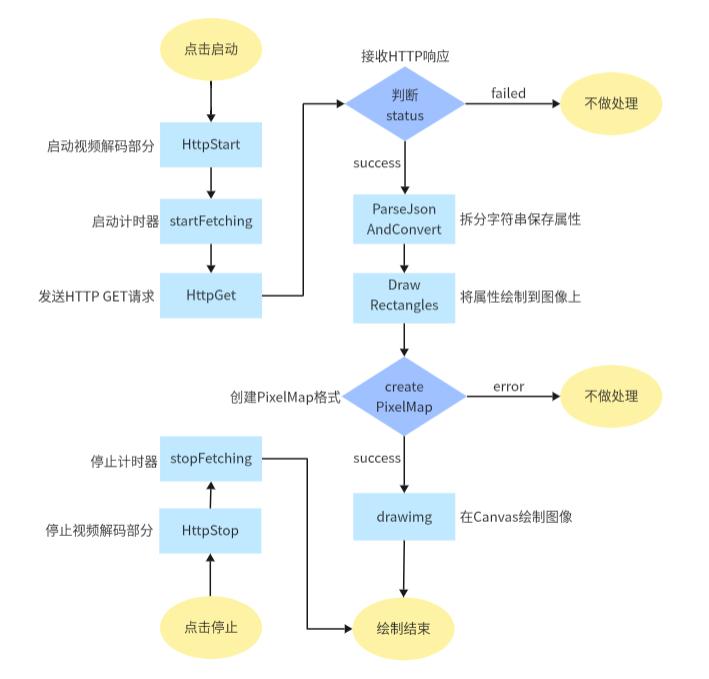
开始处理视频请求:http://192.168.137.2:8081/start?url=xxx?bit\_rate=400000

停止处理视频请求: http://192.168.137.2:8081/stop?url=xxx

## 3.8UI设计

使用 DevEco Studio 进行前端开发，采用基于 api9 的 ArkTS 和 Native C++ Stage 模型。

帧图像数据和推理结果由 HTTP 响应的 body 传输，定义格式为：推理结果 +!+ 帧图像数据。通过!对响应进行分割，叹号前的推理结果转成字符串保存，将字符串内的属性进行拆分得到图像框的坐标、概率和标签的数据，叹号后的帧图像数据经过格式转换后直接生成图片，由 Canvas 进行绘制，连续的 HTTP 请求带来的连续图像则显示成视频。通过 @ohos.request 和 @ohos.net.http 模块中的接口实现 HTTP 传输，通过 @ohos.multimedia.image 实现图片的生成和绘制。具体函数调用关系如图所示。



通过HTTP响应内容的status字段判断HTTP请求是否正确响应，若为failed则丢弃，若为success则使用HTTP结果创建Arraybuffer。由于我们在视频解码开发板的程序代码设置的结构为：推理结果+!+帧图像数据，所以我们也对结果根据叹号进行查找，拆分成字符串和数据两部分，将叹号前的Arraybuffer转为字符串，并叹号后的数据保存。将字符串传入函数ParseJsonAndConvert进行属性存储。

函数ParseJsonAndConvert使用json::parse把字符串转为json对象，再按关键词进行拆分，提取并储存属性的函数ParseJsonAndConvert：其中env表示上下文环境，str表示HTTP响应的字符串部分，rectangles用于存储矩形框坐标，labels储存识别结果的标签，probabilities储存识别概率。

将字符串后的帧图像数据转换成OpenCV Mat对象，并根据字符串中储存的属性将对应的矩形框、标签和概率绘制到图片上。由于鸿蒙ArkTS端的显示函数接受的是BGRA\_888格式，所以还需要将图片转换成BGRA格式，再将Arraybuffer返回给ArkTS端的UI进行绘制。

createPixelMap是鸿蒙@ohos.multimedia.image接口的一个函数，通过属性创建PixelMap，默认采用BGRA\_8888格式处理数据，通过Promise返回结果，我们可以用来接收C++端返回的Arraybuffer并转换成图像。

得到PixelMap格式的图片后，Canvas可以提供将图片绘制到指定区域的画板，contexts包含Canvas的上下文，可以设置其为全局变量进行函数内调用绘制。

C++端通过HTTP请求不同的url地址实现不同的线程，ArkTS端通过不同的计时器循环发送HTTP请求，通过传入的index参数决定是哪个窗口的线程。支持多线程，设计了两个窗口。通过不同的index判断各自窗口的属性和函数，设置三个按钮，分别是启动、停止和全屏。

启动按钮:点击后会向视频解码开发板发送启动请求，将视频解码程序启动，然后会不断发送GET请求，将获得的结果绘制成图片显示到对应按钮的窗口上。

停止按钮：点击后会发送停止视频解码程序的请求，同时停止GET请求的发送，绘制结束。

全屏按钮：点击后该窗口会放大到全屏位置，此时屏幕只会显示一个窗口的视频信息，在放大的界 面可以点击返回主页按钮，恢复成窗口模式。实现效果如图所示。

具体测试情况如图所示

# **4**其他

## 4.1成员分工

邹先予：负责rk3588相关ai应用和docker\_agent模块的程序编写。

岳志鑫：负责atlas的相关ai应用和鸿蒙前端的程序编写。

林雪松：负责schedule模块的程序编写以及docker容器调度等工作。

## 4.2困难与思考

**1**.由于前端开发板发送的HTTP请求速度较快，堆积的HTTP请求就会阻塞视频解码程序，造成崩溃。

解决方法：如果存在等待的HTTP请求无法得到结果导致堆积的情况，在 500 ms 后就返回一个failed作为HTTP响应，防止其一直等待。同时增加了故障恢复设置，如果再次出现意外崩溃，会自动重启并继续视频解码工作。增加了多线程的功能，便于实现多线程响应请求，更快速处理高并发的情况。

**2**.最初使用python编写的yolov5-infer程序并不支持多线程，而且存在内存泄漏的问题。

解决方法：使用c++重写该程序，在c++代码中更加完善的实现了NPU推理程序，保证了程序的安全性，增加了多线程推理的功能。

**3**.在容器内进行 NPU 推理时总会出现设备检测异常，环境异常、模型初始化异常等问题。

解决方法：经查找发现是容器内无法访问到开发板的硬件设备和对应的AscendCL推理环境，将相关硬件设备和所需文件利用 Ascend Docker Runtime 全部挂载到容器内，便可以在容器内调用开发板上的设备和相关程序进行 NPU 推理。

**4**.虽然前端设置了储存HTTP请求的响应的队列，但是由于HTTP传输所消耗的时间差异较大，有可能出现后请求的结果先返回的情况，从而导致画面顺序不一致的情况出现。同时如果队列请求的速度跟不上画面刷新的速度，画面就会出现卡顿掉帧的情况。

解决方法：使用效率更高的native c++方式进行开发，将ArkTS与napi的c++代码进行绑定，由于c++代码的传输延时短，且传输速度稳定，所以我们无需再使用队列来储存数据，而是得到返回的数据后可以直接进行处理，通过opencv库函数等方法解析结果并绘制矩形框，使得最终的ArkTS端只需要一个渲染视频的步骤，大大节省了UI渲染的时间和延迟，显示的视频更加稳定，避免了画面卡顿和顺序不一致的情况。

**5**.由于前端开发板使用鸿蒙开发的 ArkTs 语言，猜测存在一些底层逻辑的原因导致 HTTP 请求耗时较长，一次HTTP请求大约需要300 ms，而后端传输和推理大概需要30 ms左右，可实现30帧视频的效果，所以前后端开发板实时传输存在问题，前端视频显示也十分卡顿。

解决方法：设置一个队列用于储存 HTTP 请求的响应，不选择等到HTTP请求得到响应后再发送HTTP请求，而是选择每隔 30 ms 就发送一个HTTP请求，将每个请求的响应储存在队列中，需要使用时就在队列中取出队头元素进行处理，一定程度上解决了传输延迟较大的问题。ArkTS端只需要一个渲染视频的步骤，大大节省了UI渲染的时间和延迟，显示的视频更加稳定，避免了画面卡顿和顺序不一致的情况。

## 4.3参考

## **[1]** Miao X, Shi C, Duan J, et al. Spotserve: Serving generative large language models on preemptible instances[C]//Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. 2024: 1112-1127.

## **[2]** Cao D, Yoo J, Xu Z, et al. Microedge: a multi-tenant edge cluster system architecture for scalable camera processing[C]//Proceedings of the 23rd ACM/IFIP International Middleware Conference. 2022: 322-334.

### **[3]** Zhang L, Fu Z, Shi B, et al. More is Different: Prototyping and Analyzing a New Form of Edge Server with Massive Mobile {SoCs}[C]//2024 USENIX Annual Technical Conference (USENIX ATC 24). 2024: 285-302.

#### **[4]** Nanos A, Kretsis A, Mainas C, et al. Hardware-Accelerated FaaS for the Edge-Cloud Continuum[C]//2023 IEEE 31st International Conference on Network Protocols (ICNP). IEEE, 2023: 1-6.