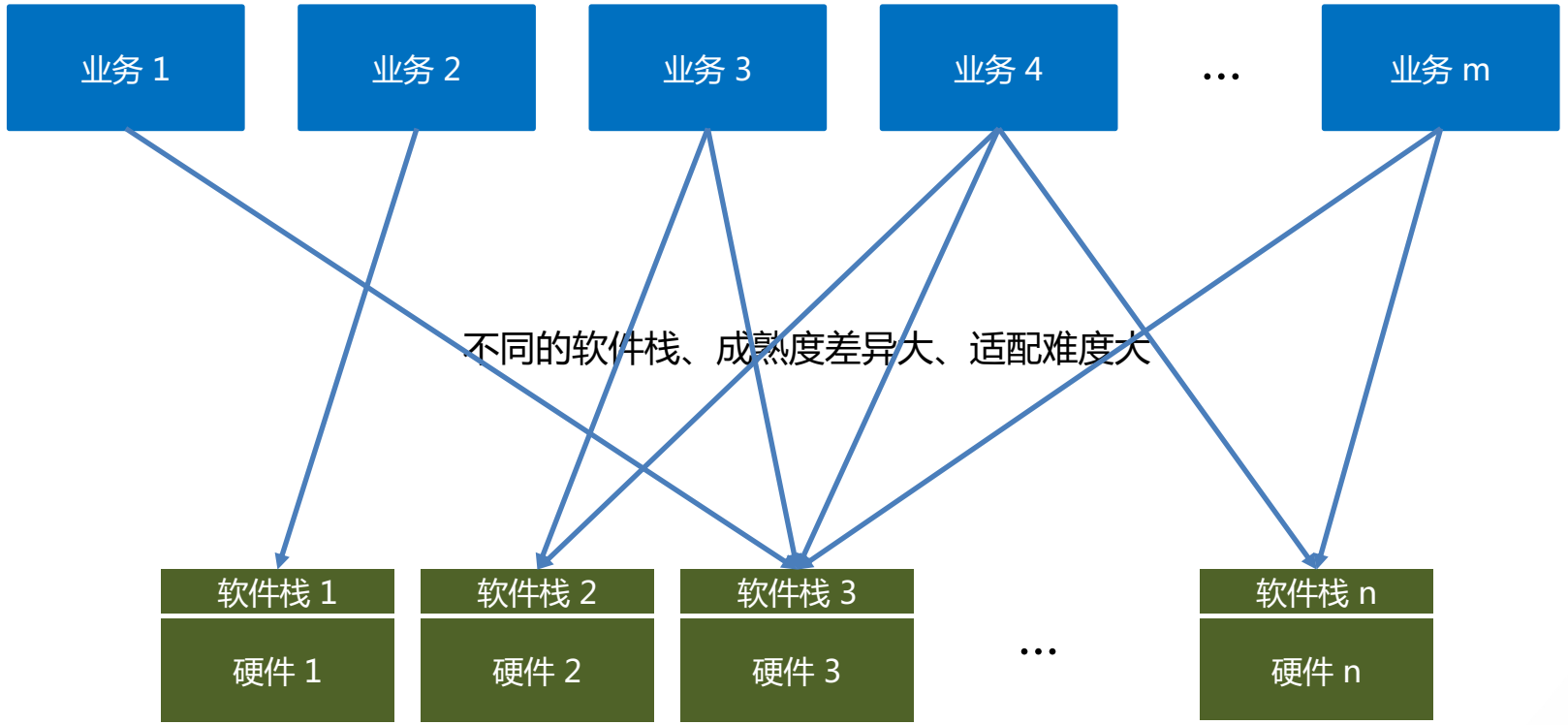
飞桨推理硬件适配方案

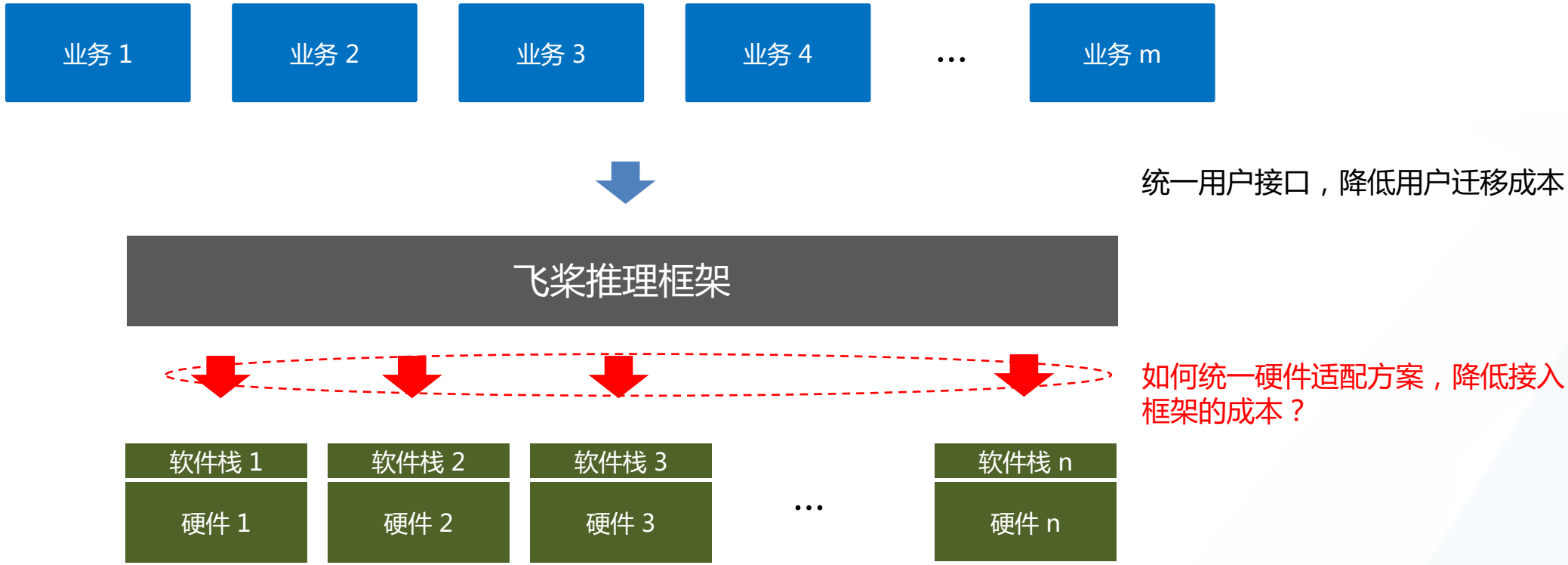
飞桨为什么需要统一推理硬件适配方案？

业务场景复杂（云、边缘、端），一个业务可能对多种硬件均有适配需求，迁移成本高



硬件种类多、功能差异大、限制不同

飞桨为什么需要统一推理硬件适配方案？



统一用户接口，降低用户迁移成本

如何统一硬件适配方案，降低接入框架的成本？

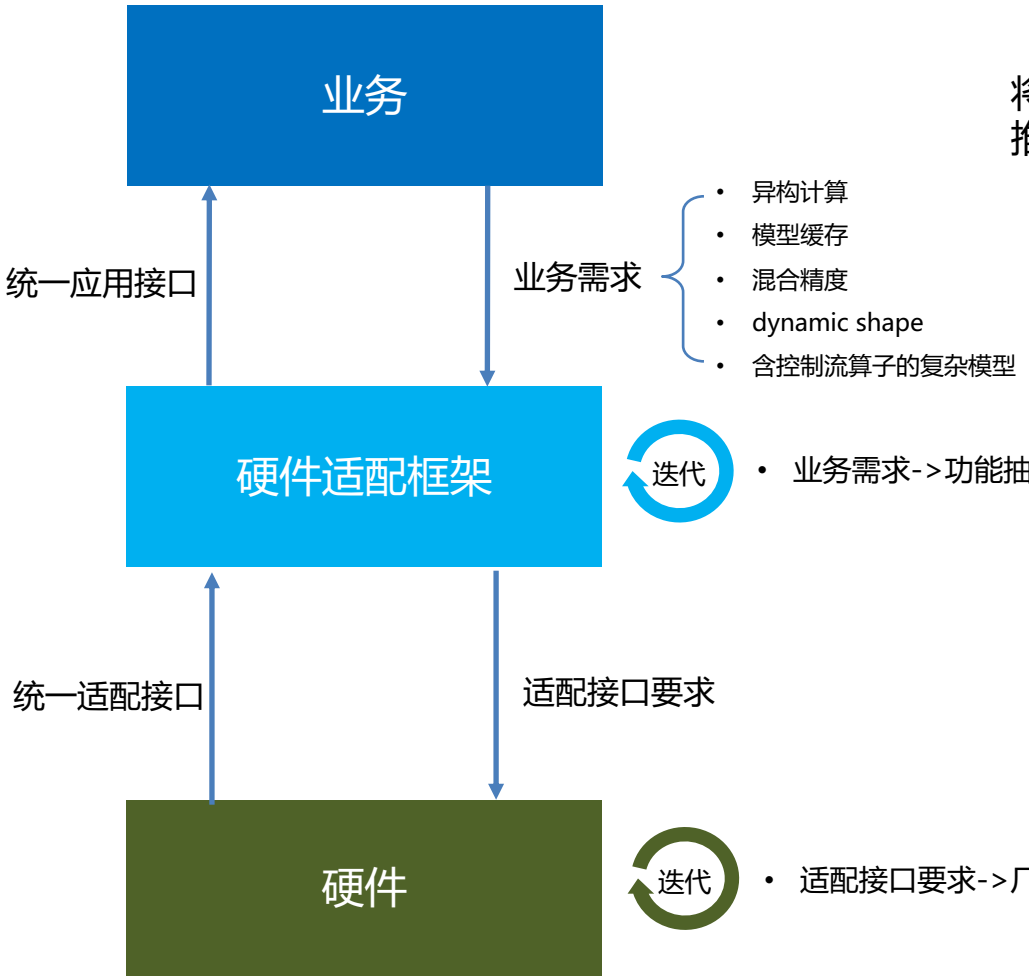
如何设计一个既满足业务要求、又具备普适性的硬件适配框架？

自底向上

框架能力和硬件能力之间取最大公约数，框架提供相关能力的接口抽象。

自顶向下

将业务需求转化为对适配方案的迭代，推动厂商对软、硬件能力的升级。



- 异构计算
- 模型缓存
- 混合精度
- dynamic shape
- 含控制流算子的复杂模型

迭代

- 业务需求->功能抽象->适配接口的统一

迭代

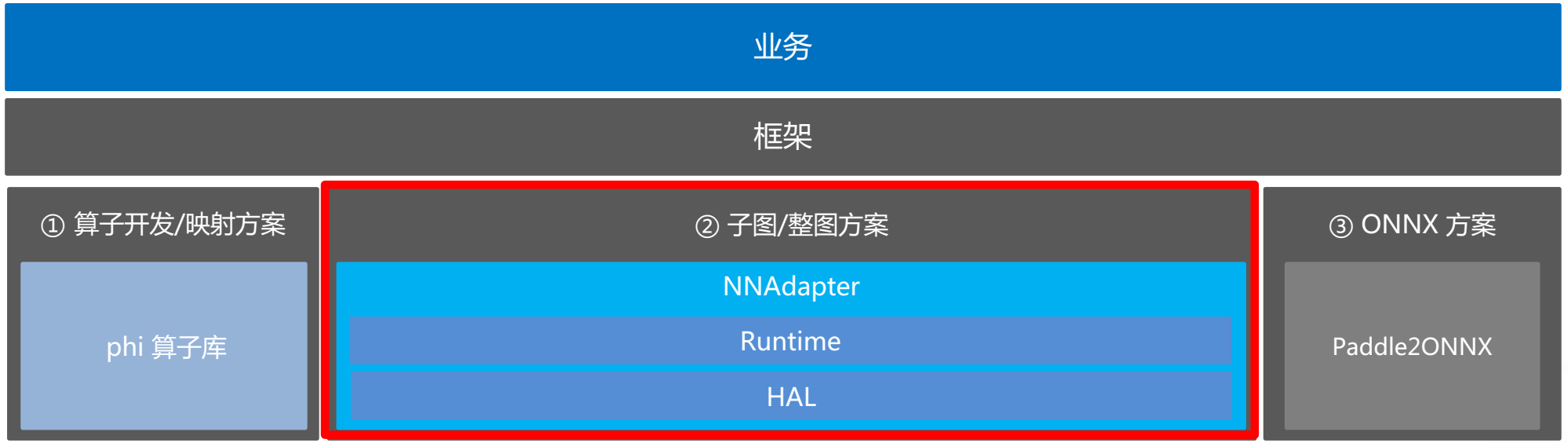
- 适配接口要求->厂商软、硬件能力的升级

- 广泛接入不同形态的硬件、寻找共性、完成基础能力的抽象并统一适配接口

飞桨将选择成熟度高的硬件作为标杆！

飞桨推理硬件适配方案

根据厂商提供的接口或工具，飞桨提供了三种适配方案：



不同接入方案由厂商软件栈成熟度决定

成熟 ← → 起步

训、推一体算子、图接口阶段 （开放可编程接口、高性能算子库、侧重训练、推理场景）	推理专用图引擎阶段 （提供类似 TensorRT 图引擎接口，原生适配开源框架）	离线工具阶段 （离线工具+Runtime API）

优点	开放程度高、灵活、充分挖掘算力，支持的算子和模型数量多，支持训、推一体化。	借助开源框架生态，降低业务迁移成本，易于模型迭代和分发（一个模型在不同硬件种类、型号、软件版本上均能完成最优的适配），通过异构支持最新算子，扩大模型支持范围，性能优化完全交由厂商引擎内部实现，框架适配门槛较低。	厂商易于实现，可快速完成特定场景模型的支持。
缺点	需要做深度优化，对研发能力要求较高。	算子数量和性能都取决于厂商引擎内部实现，可定制化程度低，灵活性较差。	算子少、模型少、场景受限，不利于深度的软硬协同优化。

子图/整图方案（对比 ONNX 离线工具方案）的优势

- 算子模型适配的广度
  - ONNX 离线方案受 ONNX 算子表示能力的限制

    受限于 ONNX op set 更新的滞后性，一些较新的模型或自定义算子 ONNX 无法导出
  - 子图/整图方案支持异构计算

    Paddle Lite 具备丰富的 CPU 算子，支持 CPU + AI 加速器的异构计算

- 软硬协同优化的深度
  - 性能优化

    子图/整图方案能复用 Paddle Lite 在图上的通用优化和 ARM CPU 算子上的深度优化
  - 功能支持

    子图/整图方案支持 dynamic shape 、混合精度、控制流模型

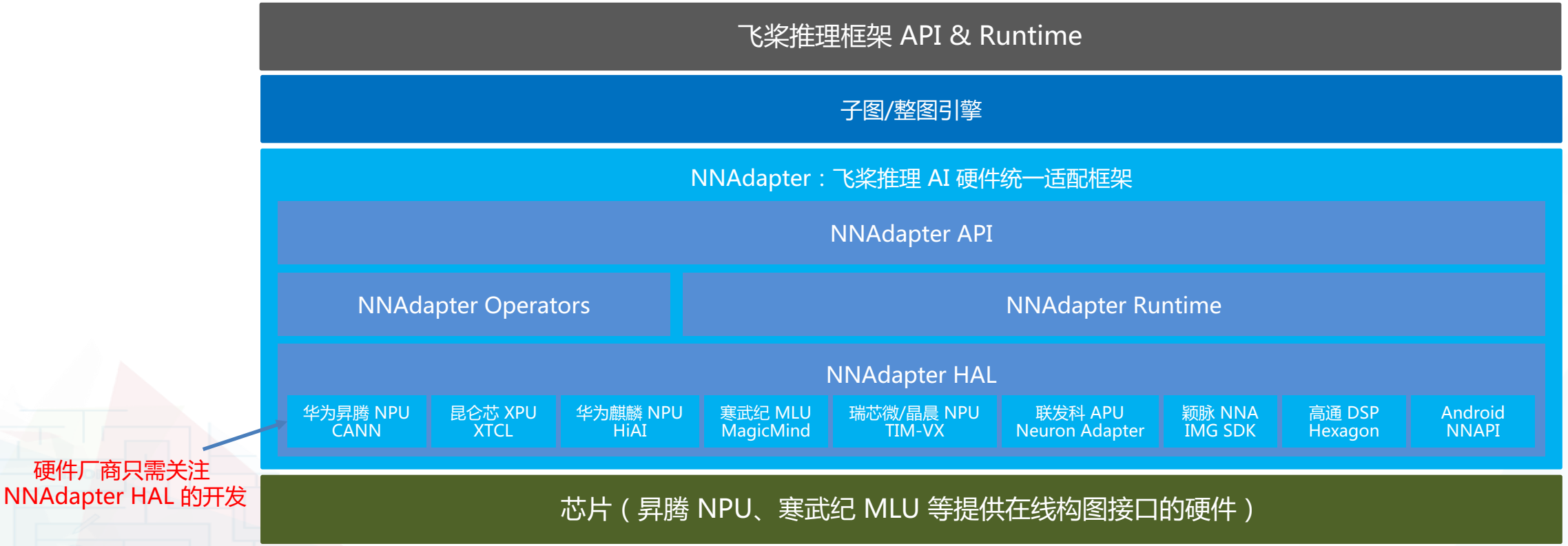
- 软、硬件和模型升级的易用性
  - 易于设备升级

    当设备芯片型号、软件版本发生变更时，离线方式需对存量模型重新完成离线生成并下发至设备，而在线方式允许设备在升级后自动完成模型生成
  - 易于模型迭代

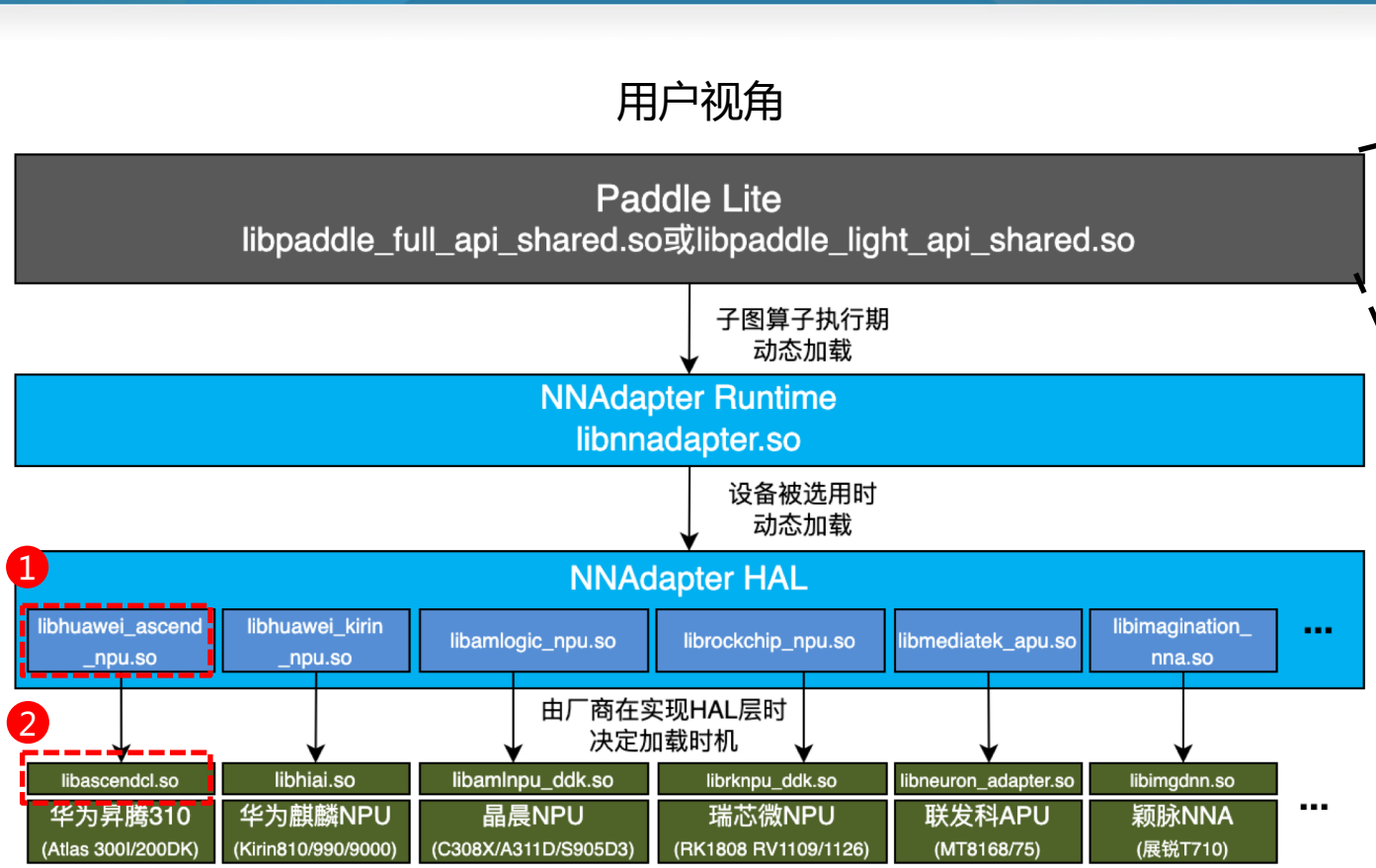
    离线方式需要生成适用于不同设备（芯片型号、软件版本）的模型并分别下发至相应的设备，而在线方式根据每个设备自身软、硬件环境自动生成性能最优的模型

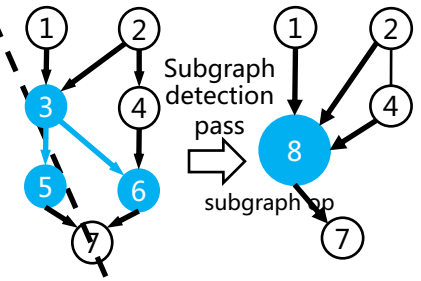
NNAdapter：飞桨推理 AI 硬件统一适配框架

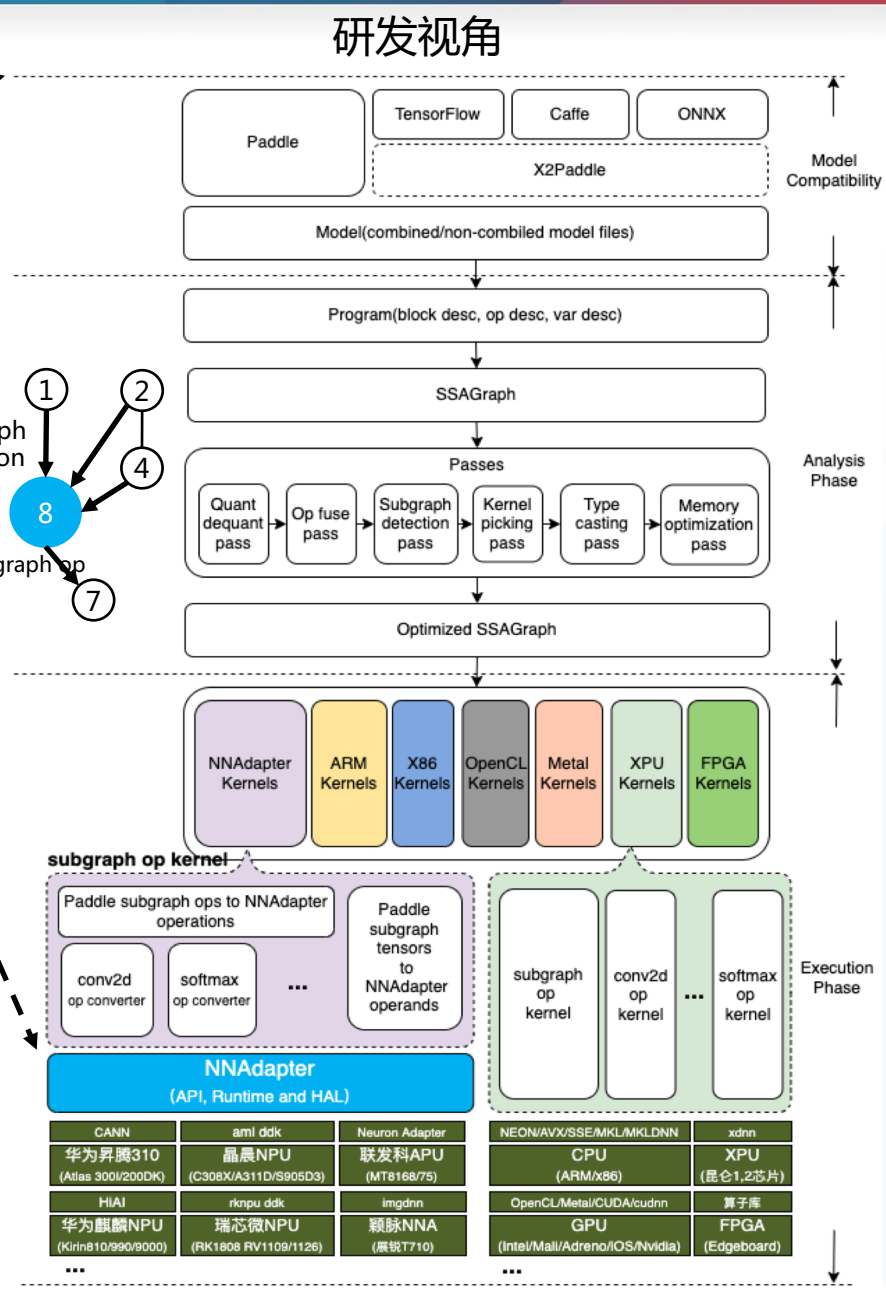
- 定义
  - 由一系列 C 接口组成的、支撑各种深度学习框架在各种硬件（特别是 AI ASIC 芯片）完成高效推理的通用接口，它是建立深度学习推理框架和硬件的桥梁。
  - 通过 API 和 HAL 分别标准化了**框架适配层接口**（为不同推理框架提供统一的适配接口）和**硬件抽象层接口**（为不同设备提供统一的访问接口），实现了**推理框架与硬件适配完全解耦**，进一步降低厂商适配门槛、缩短适配周期、减少沟通和维护成本。



飞桨推理框架 Paddle Lite 、NNAdapter 与厂商软件栈的关系





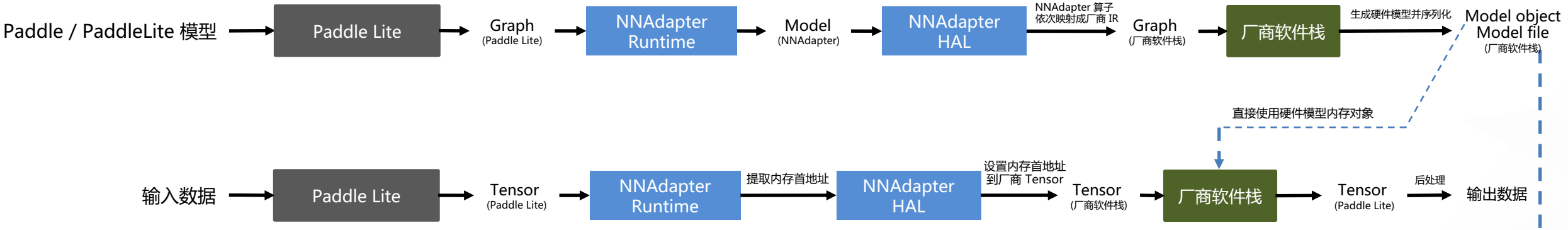


1 厂商硬件的 NNAdapter HAL：约 15~20 人天

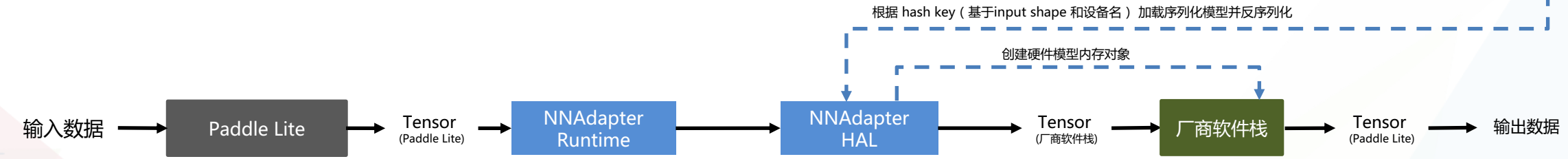
2（已具备该能力的厂商除外，如昇腾、寒武纪）厂商在线构图、模型生成和执行 DDK：约 45~60 人天

Paddle Lite + NNAdapter 执行流程

- 业务首次启动执行推理



- 业务第 2 次启动执行推理



NNAdapter API

- 定义
  - 由设备管理、多设备统一上下文管理、模型组网、编译和生成、执行等一系列 C 接口组成的 API，可实现硬件适配与推理框架完全解耦。
- 功能分类
  - 设备管理
    - 查询设备基本信息，包括设备名称、厂商名称、加速卡类型和 HAL 库版本，以及设备的获取和初始化等。
    - NNAdapterDevice_acquire, NNAdapterDevice_release, NNAdapterDevice_getName, NNAdapterDevice_getVendor, NNAdapterDevice_getType, NNAdapterDevice_getVersion
  - 多设备统一上下文管理
    - 创建多种设备统一的设备上下文，通过 Key-value 字串的方式为每种设备配置设备运行、模型编译和执行等参数。
    - NNAdapterContext_create, NNAdapterContext_destroy
  - 模型组网
    - 为实现与推理框架中的模型表达解耦，建立与设备无关的、统一的 NNAdapter 模型 Model 的中间表达，需要基于如下 API 将推理框架的模型中的算子、张量对象转化为 NNAdapter 的操作符 Operation 和操作数 Operand。
    - NNAdapterModel_create, NNAdapterModel_destroy, NNAdapterModel_finish, NNAdapterModel_addOperand, NNAdapterModel_setOperandValue, NNAdapterModel_getOperandType, NNAdapterModel_addOperation, NNAdapterModel_identifyInputsAndOutputs
  - 模型编译和生成
    - 基于创建的模型编译实例，通过在 HAL 层库中调用厂商 SDK 实现 NNAdapter 模型的中间表达向目标设备代码的转换。
    - NNAdapterCompilation_create, NNAdapterCompilation_destroy, NNAdapterCompilation_finish, NNAdapterCompilation_queryInputsAndOutputs
  - 模型执行
    - 创建执行计划实例，设置输入、输出，执行目标设备代码后将结果返回给推理框架。
    - NNAdapterExecution_create, NNAdapterExecution_destroy, NNAdapterExecution_setInput, NNAdapterExecution_setOutput, NNAdapterExecution_compute

*更多 NNAdapter API 可参考 nnadapter.h

NNAdapter Operators

- 定义
  - 提供了一组类 ONNX 的算子定义，用于建立独立于推理框架的、与设备无关的、Runtime 层与 HAL 层统一的模型中间表达。
- 示例
  - 如下定义了『逐元素相加操作符 ADD』的基本功能、输入/输出操作数列表，其中输入/输出操作数列表中的每一个操作数需要严格按照定义的顺序给定。

```
typedef enum {
  /**
    * Performs element-wise binary addition(with Numpy-style broadcasting
    * https://numpy.org/doc/stable/user/basics.broadcasting.html).
    *
    * Inputs:
    * * 0: input0, A NNADAPTER_TENSOR_FLOAT32,
    * NNADAPTER_TENSOR_QUANT_INT8_SYMM_PER_LAYER tensor.
    * * 1: input1, A tensor with the same type as input0.
    * * 2: fuse_code, A NNADAPTER_INT32 scalar, specifies the activation to the
    * result, must be one of NNAdapterFuseCode values.
    *
    * Outputs:
    * * 0: output, The result with the same type as two inputs.
    *
    * Available since version 1.
    */
    NNADAPTER_ADD = 0,
   ......
} NNAdapterOperationCode;
```

*更多 NNAdapter 算子定义可参考 nnadapter.h

NNAdapter Runtime

- 定义
  - 它作为 API 和 HAL 层的桥梁，其作用不仅是将 API 的调用翻译成模型、操作数、操作符的中间表达以及设备 HAL 层接口的调用，还包括设备 HAL 层库的注册、模型缓存的分层序列化和反序列化。
- 功能
  - 设备管理
    - 用户进程的模型在某个设备上第一次推理时， Runtime 的 DeviceManager 发现该设备的 HAL 层库没有被加载，则会根据设备名找到并加载 HAL 层库，再依据约定的设备接口描述符号命名规则解析并获得该设备的设备接口描述实例的首地址，进而获得设备的基本信息和各功能函数地址，最后将它注册到 DeviceManager 统一管理。
  - 多种设备间的异构
    - 即同一个硬件的不同运算单元，例如某 SoC 芯片的 DSP 和 NPU，它将根据每一种设备支持的操作符列表进行子图划分， 按照拓扑顺序在不同的设备中执行模型片段。
  - 模型缓存的序列化和反序列化
    - Runtime 通过设备 HAL 层库调用厂商 SDK 将模型的中间表示转为设备代码的过程通常耗时较长，一般与模型规模成正比，与芯片 CPU 的处理能力成反比，因此，模型的在线编译和生成大大增加了用户进程启动后的第一次推理耗时，这在一些应用中是不可接受的，为了避免这个问题，Runtime 支持将已编译的设备代码缓存到文件系统中，而在下一次模型编译时直接加载该缓存文件，目前 NNAdapter 采用分层的序列化和反序列化策略。

* f3b1530890dd4efffb060e3c88fbb6f5 是用于唯一标识子图的 32B hash 值

NNAdapter HAL

- 定义
  - 为屏蔽硬件细节，向 Runtime 提供统一的设备访问接口，NNAdapter 在 Runtime 和 厂商 SDK 之间建立了 HAL 硬件抽象层，它是由基于 C 结构体实现的统一设备接口描述、模型、操作数和操作符的中间表达等数据结构组成。

```
typedef struct Hint {
    void* handler;
    void (*deleter)(void** handler);
} Hint;

typedef struct Operand {
    NNAdapterOperandType type;
    void* buffer;
    uint32_t length;
    Hint hints[NNADAPTER_MAX_SIZE_OF_HINTS];
} Operand;

typedef struct Argument {
    int index;
    void* memory;
    void* (*access)(void* memory, NNAdapterOperandType* type);
} Argument;

typedef struct Operation {
    NNAdapterOperationType type;
    std::vector<Operand*> input_operands;
    std::vector<Operand*> output_operands;
} Operation;

typedef struct Cache {
    const char* token;
    const char* dir;
    std::vector<NNAdapterOperandType> input_types;
    std::vector<NNAdapterOperandType> output_types;
    std::vector<uint8_t> buffer;
} Cache;
```

```
typedef struct Model {
    std::list<Operand> operands;
    std::list<Operation> operations;
    std::vector<Operand*> input_operands;
    std::vector<Operand*> output_operands;
} Model;

typedef struct Device {
    // Properties
    const char* name;
    const char* vendor;
    NNAdapterDeviceType type;
    int32_t version;
    // Interfaces
    int (*open_device)(void** device);
    void (*close_device)(void* device);
    int (*create_context)(void* device, const char* properties, void** context);
    void (*destroy_context)(void* context);
    int (*create_program)(void* context, Model* model, Cache* cache, void** program);
    void (*destroy_program)(void* program);
    int (*execute_program)(void* program, uint32_t input_count, Argument* input_arguments,
uint32_t output_count, Argument* output_arguments);
} Device;
```

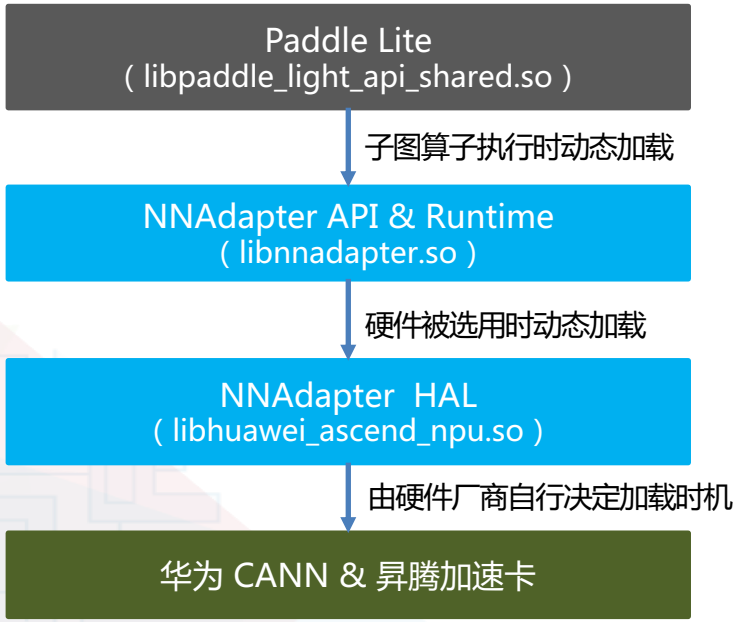
硬件 HAL 层的实现即是对 Device 类型实例化的过程

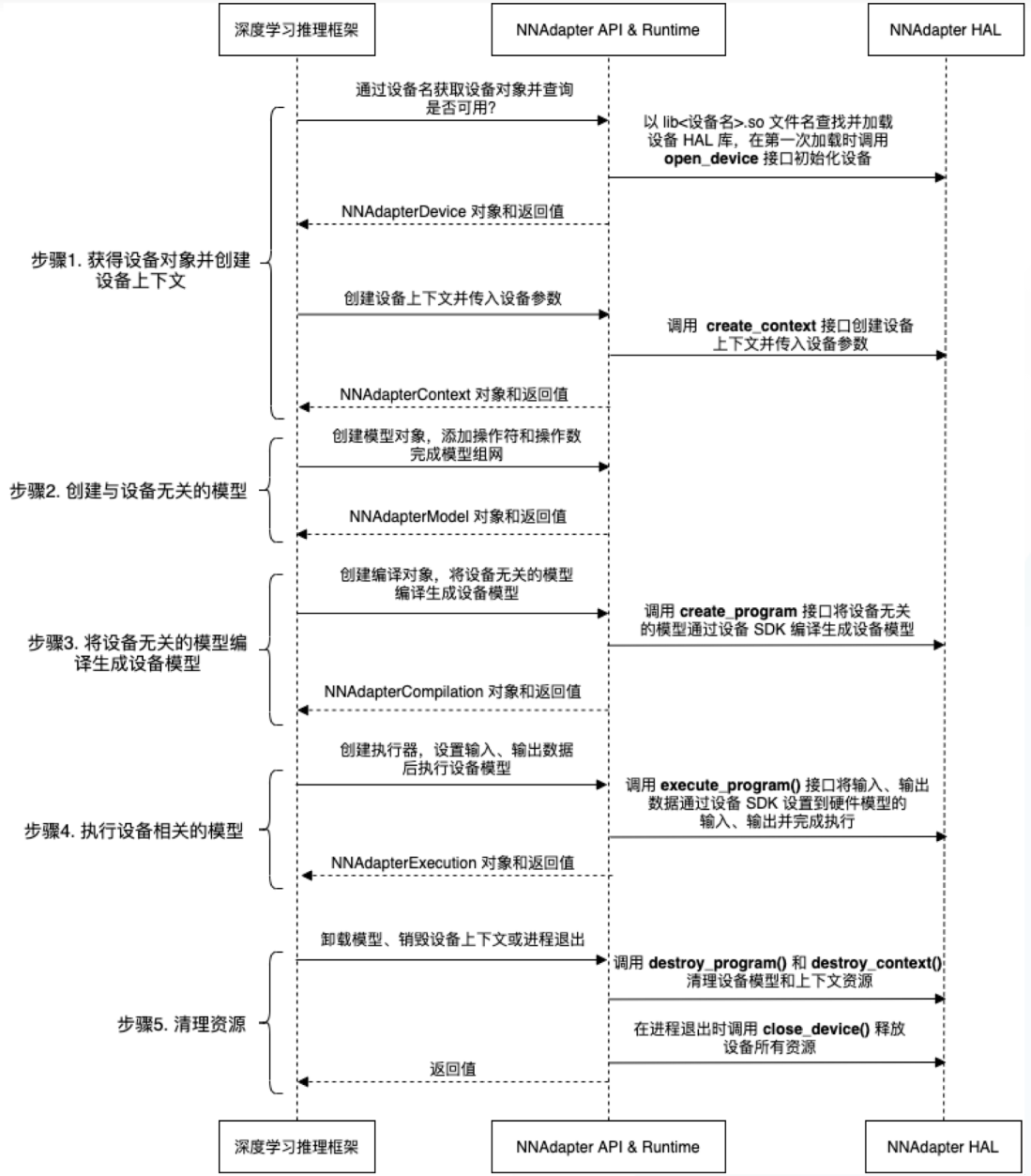
*更多 NNAdapter HAL 的定义可参考 types.h

NNAdapter HAL

- 示例（华为昇腾 HAL ）

```
__attribute__((visibility("default"))) nnadapter::driver::Device huawei_ascend_npu= {
    .name = "huawei_ascend_npu",
    .vendor = "Huawei",
    .type = NNADAPTER_ACCELERATOR,
    .version = 1,
    .open_device = nnadapter::huawei_ascend_npu::OpenDevice,
    .close_device = nnadapter::huawei_ascend_npu::CloseDevice,
    .create_context = nnadapter::huawei_ascend_npu::CreateContext,
    .destroy_context = nnadapter::huawei_ascend_npu::DestroyContext,
    .create_program = nnadapter::huawei_ascend_npu::CreateProgram,
    .destroy_program = nnadapter::huawei_ascend_npu::DestroyProgram,
    .execute_program = nnadapter::huawei_ascend_npu::ExecuteProgram,
};
```





Paddle Lite 为 NNAdapter 新增的接口

- 查询设备
  - bool check_nnadapter_device_name(const std::string& nnadapter_device_name)
- 设置设备
  - void set_nnadapter_device_names(const std::vector<std::string>& nnadapter_device_names)
- 基于 key-value 方式设置设备参数、模型编译等参数
  - void set_nnadapter_context_properties(const std::string& nnadapter_context_properties)
- 设置模型缓存信息
  - void set_nnadapter_model_cache_dir(const std::string& nnadapter_model_cache_dir)
  - void set_nnadapter_model_cache_buffers(const std::string& key, const std::vector<char>& buffer)
- 设置 dynamic shape 信息
  - void set_nnadapter_dynamic_shape_info(const std::map<std::string, std::vector<std::vector<int64_t>>>& nnadapter_dynamic_shape_info)
- 设置自定义子图划分配置信息
  - void set_nnadapter_subgraph_partition_config_path(const std::string& subgraph_partition_config_path)
  - void set_nnadapter_subgraph_partition_config_buffer(const std::string& subgraph_partition_config_buffer)
- 设置自定义混合精度配置信息
  - void set_nnadapter_mixed_precision_quantization_config_path(const std::string& mixed_precision_quantization_config_path)
  - void set_nnadapter_mixed_precision_quantization_config_buffer(const std::string&mixed_precision_quantization_config_buffer)

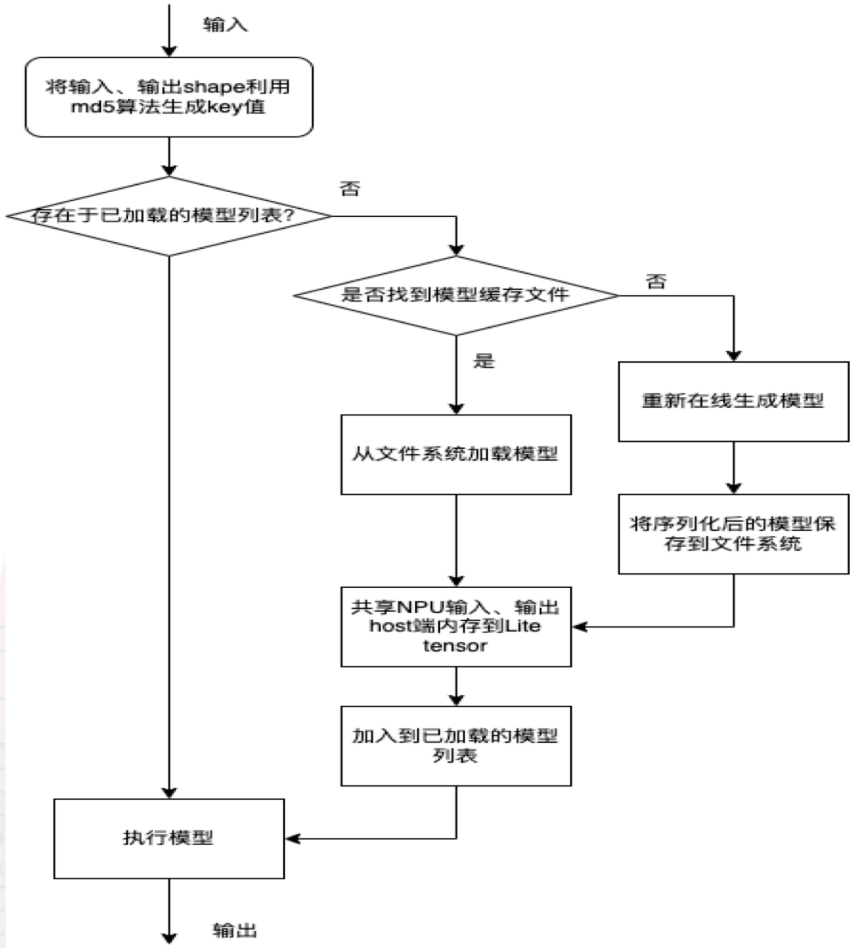
NNAdapter 具备哪些特性？

- 支持模型缓存：避免模型在线构建和编译，降低应用首次推理耗时
- 模型 output shape 的推导：支持动转静模型和动态 shape 模型
- 混合精度模型中张量数据类型的转换：自动插入 Quantize 和 Dequantize 算子
- 数据布局的转换：自动 NCHW 到 NHWC 转换，解决部分硬件唯一支持 NHWC 算子的问题
- 对称、非对称量化类型的转换：解决部分硬件唯一支持非对称量化类型
- 解决多输入/输出的量化 op 的量化参数一致性约束问题

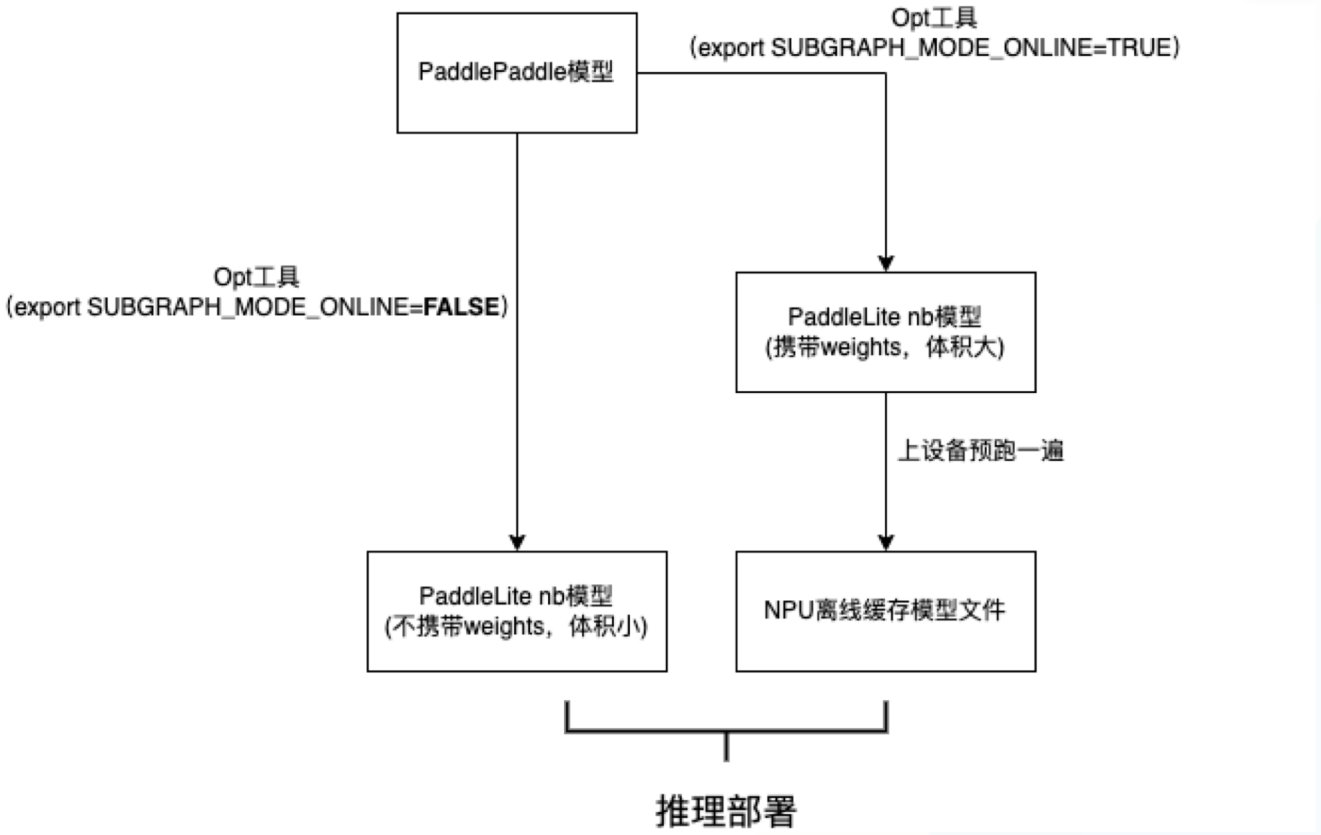
特性（1）：模型缓存

- 问题
  - 在每次启动后的第一次推理时，都需要编译、生成硬件模型而导致耗时过长
  - 待部署的模型中携带有原始模型的所有参数，往往比缓存后的硬件模型的体积大

- 技术方案 ① 模型缓存机制，避免在线编译、生成硬件模型

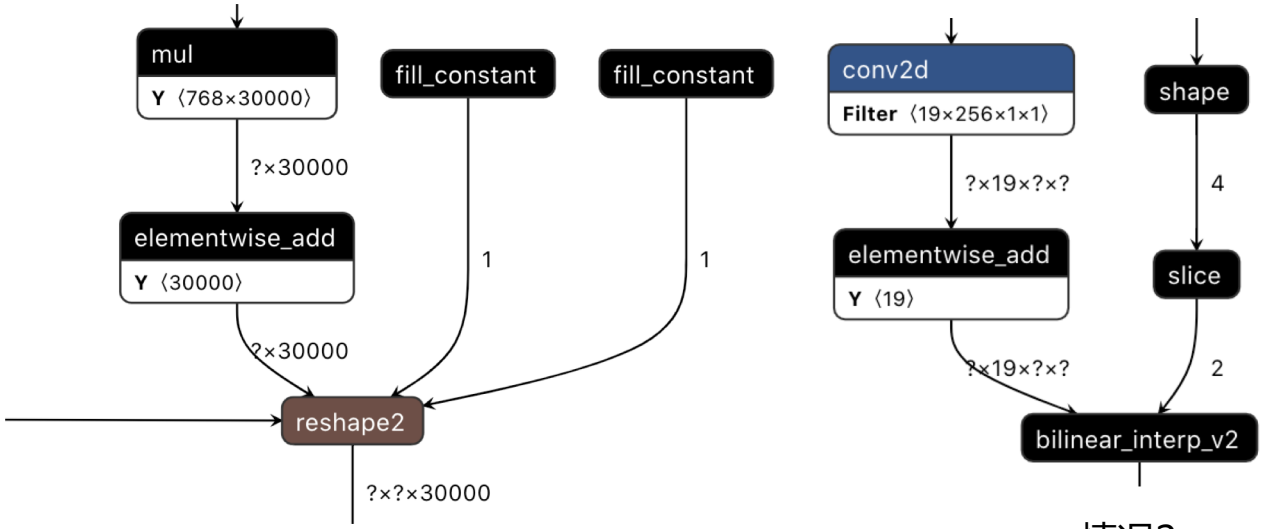


② 两次离线处理，生成模型缓存后删除原始参数，减少部署模型的体积，降低内存



特性（2）：模型 output shape 的推导

- 问题
  - 解决动转静模型在某些硬件上的支持问题

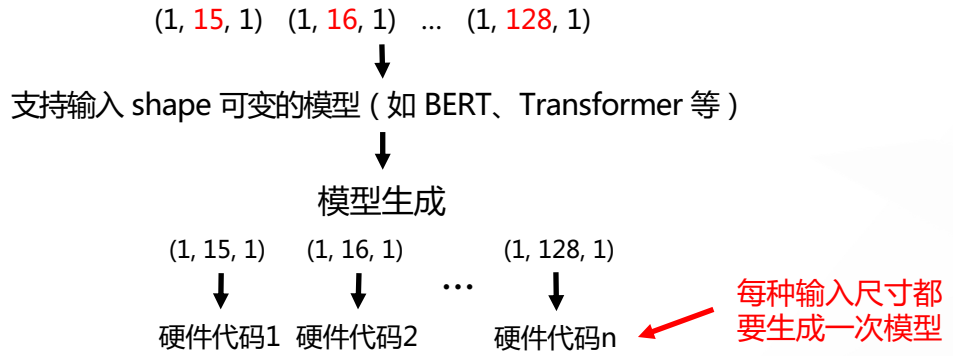


情况1　　情况2

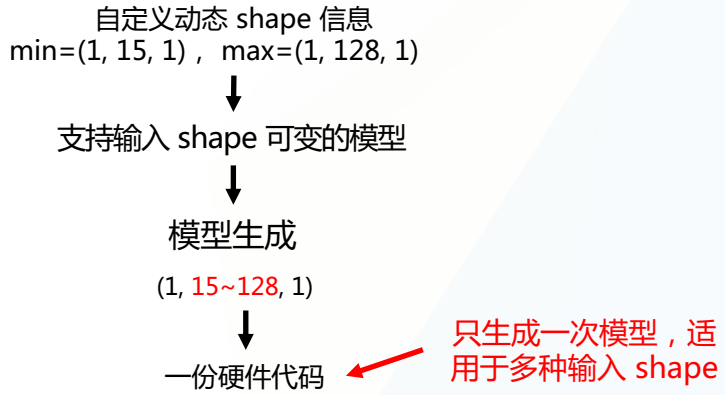
以上两种情况均无法在图分析阶段进行输出 shape 的推导，导致这类模型在某些要求设置输出 shape的硬件上无法得到支持

- 技术方案
  - 调用 NNAadpter API 进行构图时（非计算过程中），NNAdapter Runtime 会自动对每个算子的 output shape 进行推导。

- 问题
  - 解决动态 shape 模型的多次编译问题

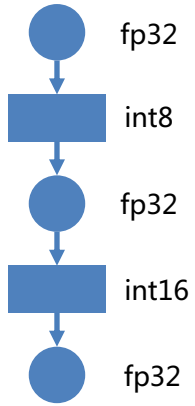


- 技术方案
  - 允许用户传入输入 shape 变化范围；
  - 调用 NNAdapter API 进行构图时，自动对shape的可能取值进行 output shape 推导，帮助硬件估算内存分配的最大值。

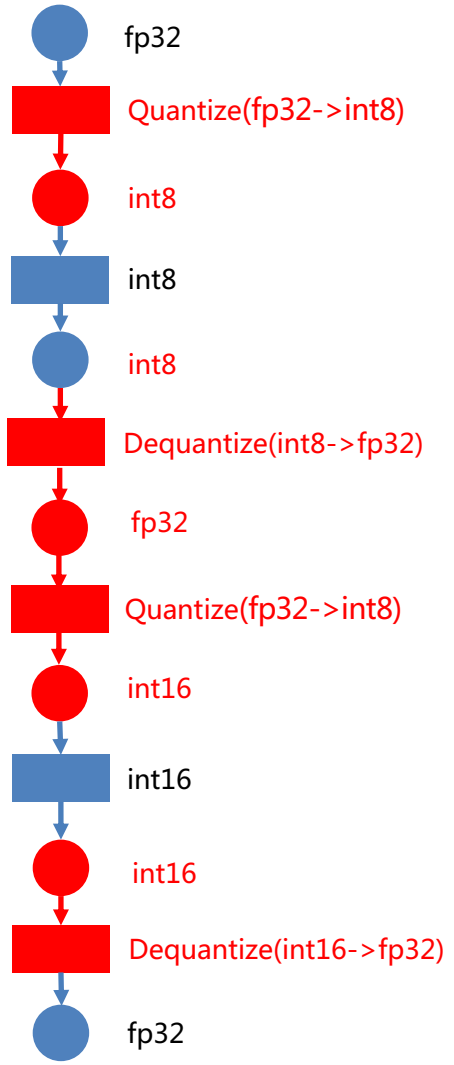


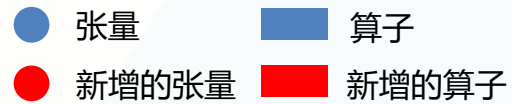
特性（3）：混合精度模型中张量数据类型的转换

- 问题
  - 混合精度模型无法进行不同算子输入、输出张量数据类型间的转换
- 技术方案



自动插入 Quantize 和 Dequantize 算子，进行张量数据类型的转换

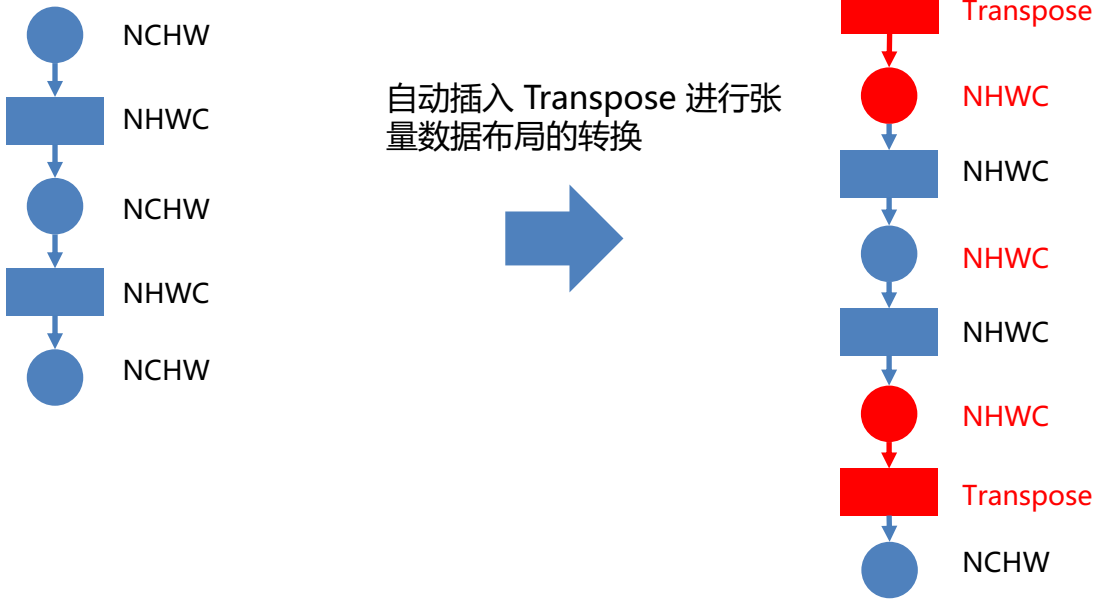




特性（4）：数据布局的转换

- 问题
  - 部分硬件的算子的输入、输出张量只支持 NHWC 的数据布局

- 技术方案



* 假设某硬件的算子只支持 NHWC 数据布局

张量　算子

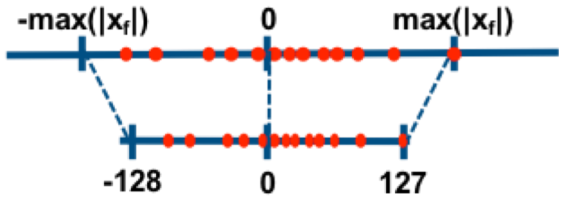
新增张量　新增算子

特性（5）：对称、非对称量化类型的转换

- 问题
  - 部分硬件只支持 TFLite 的非对称量化方式， 不支持 Paddle 所采用的对称量化方式

- 技术方案

对称量化
（仅含 scale）

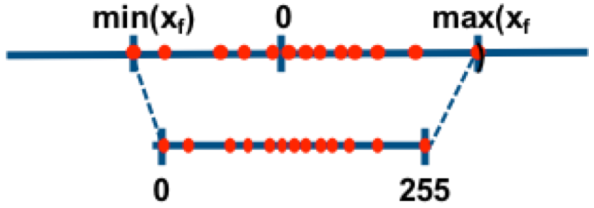


$$scale=\frac{\max(|x|)}{2^{n-1}-1}$$

$$x\_q=\left\lfloor\frac{x}{scale}\right\rfloor$$

$$x' = x\_q * scale$$

非对称量化
（scale 和 zeroPoint）

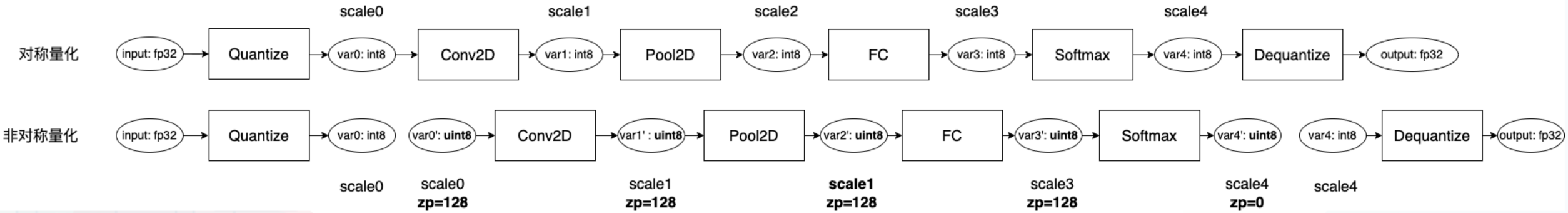


$$scale=\frac{\max(x)-\min(x)}{2^{n}-1}$$

$$zp=\left\lfloor\frac{0-\min(x)}{scale}\right\rfloor$$

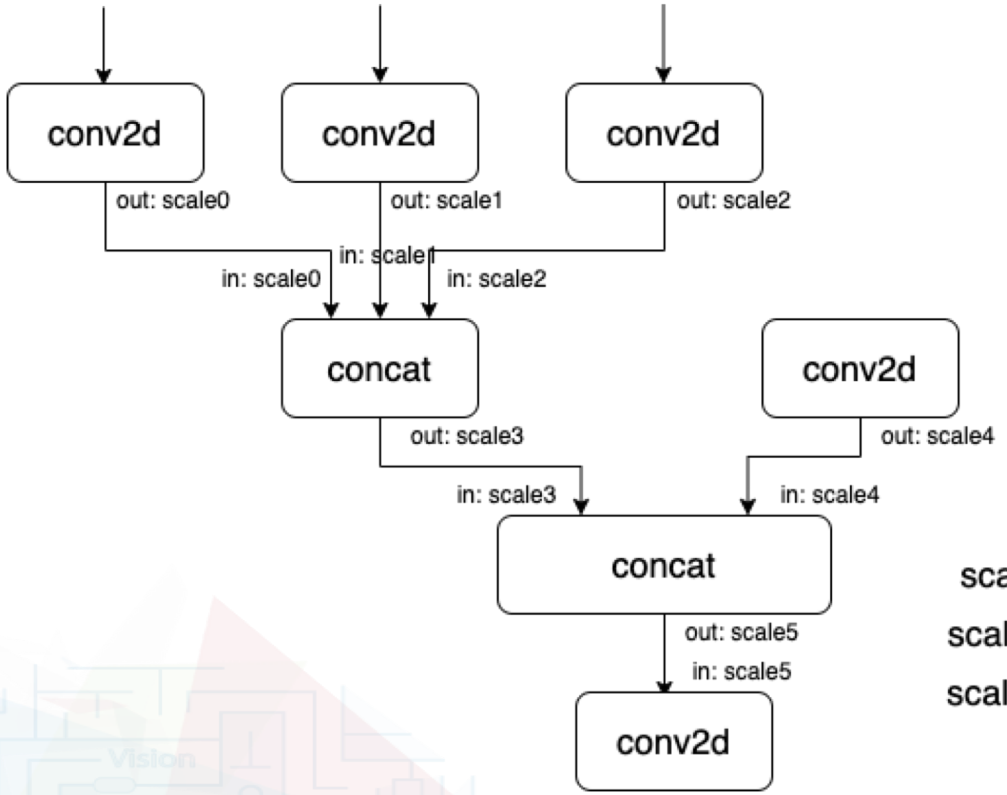
$$x\_q=\left\lfloor\frac{x-\min(x)}{scale}\right\rfloor+zp$$

$$x' = (x\_q - zp) * scale$$

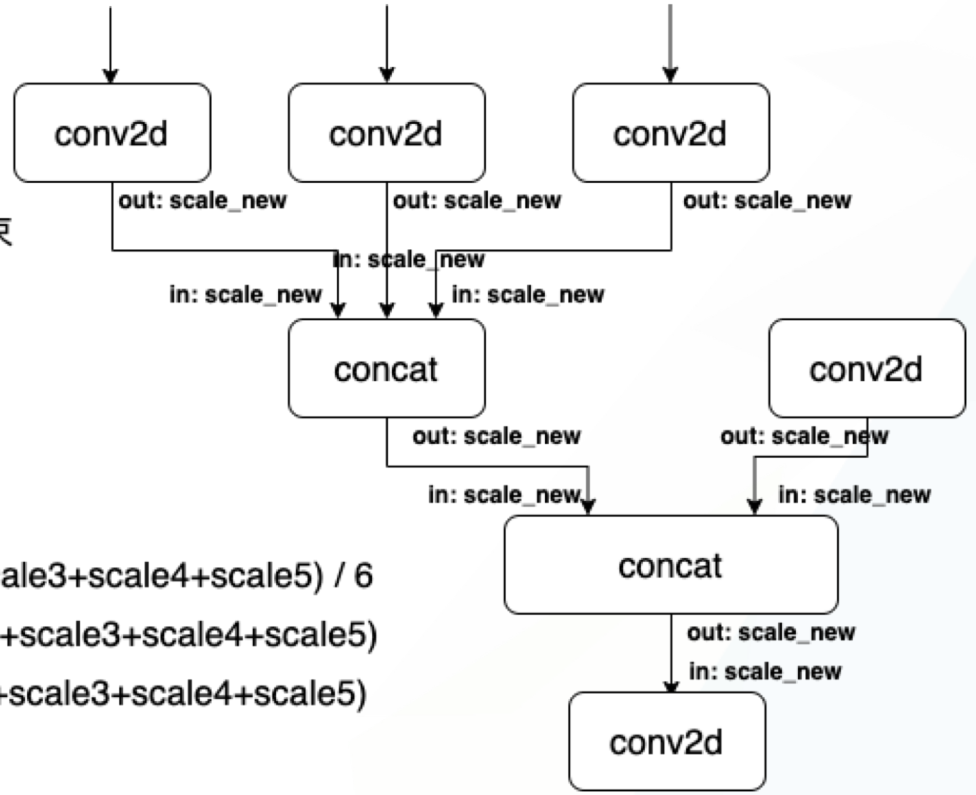


特性（6）：解决多输入/输出的量化 op 的量化参数一致性约束问题

- 问题
  - 针多输入、多输出的量化op（例如concat或者split），部分硬件要求所以输入/输出张量的量化信息保持一致
- 技术方案



输入、输出 scale 一致性约束

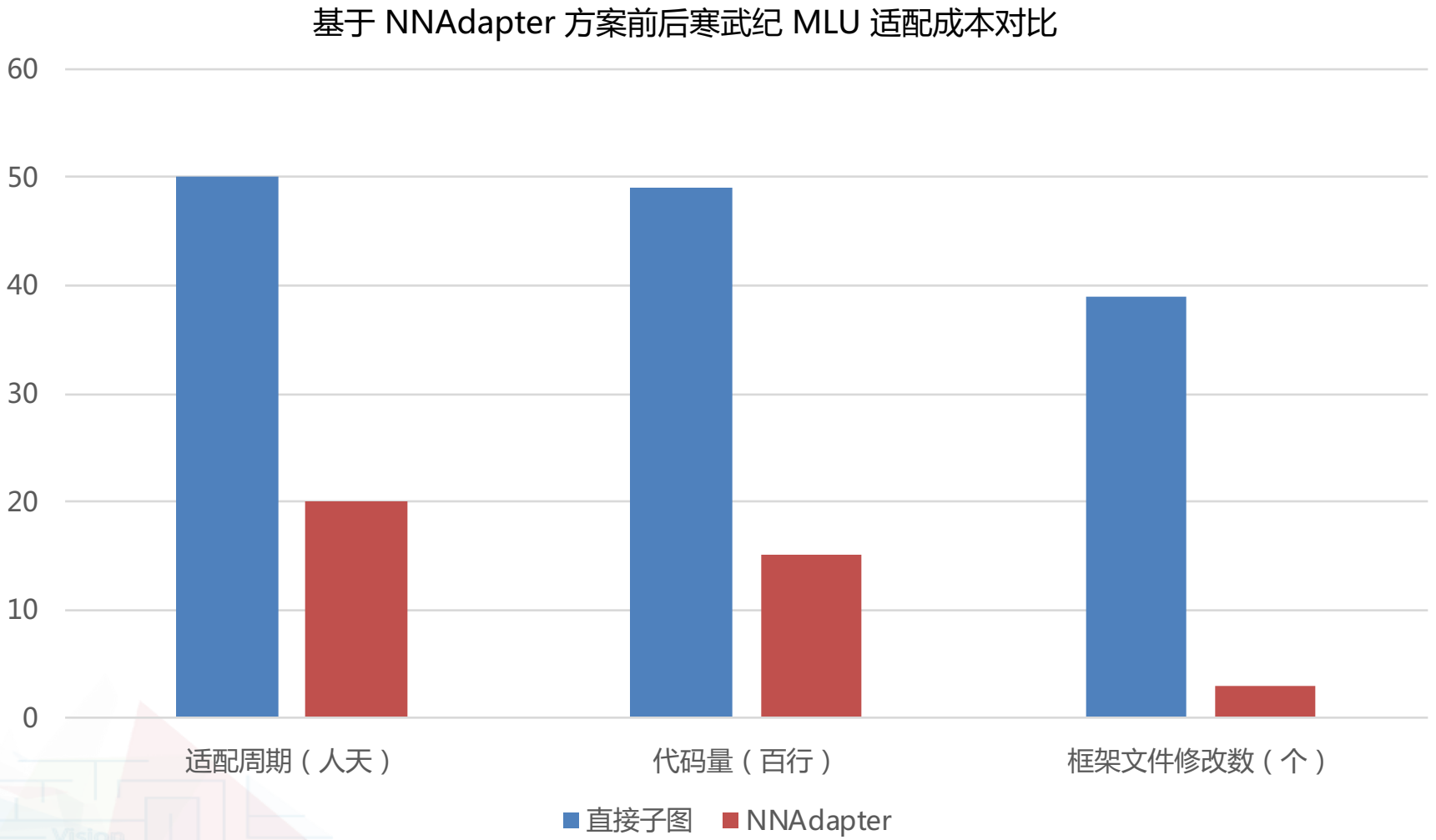


scale_new=(scale0+scale1+scale2+scale3+scale4+scale5) / 6

scale_new=max(scale0+scale1+scale2+scale3+scale4+scale5)

scale_new=min(scale0+scale1+scale2+scale3+scale4+scale5)

相较于旧方案，获得了哪些收益？



适配周期
（人力成本指标）
⬇ 60%

代码量
（复杂度指标）
⬇ 69%

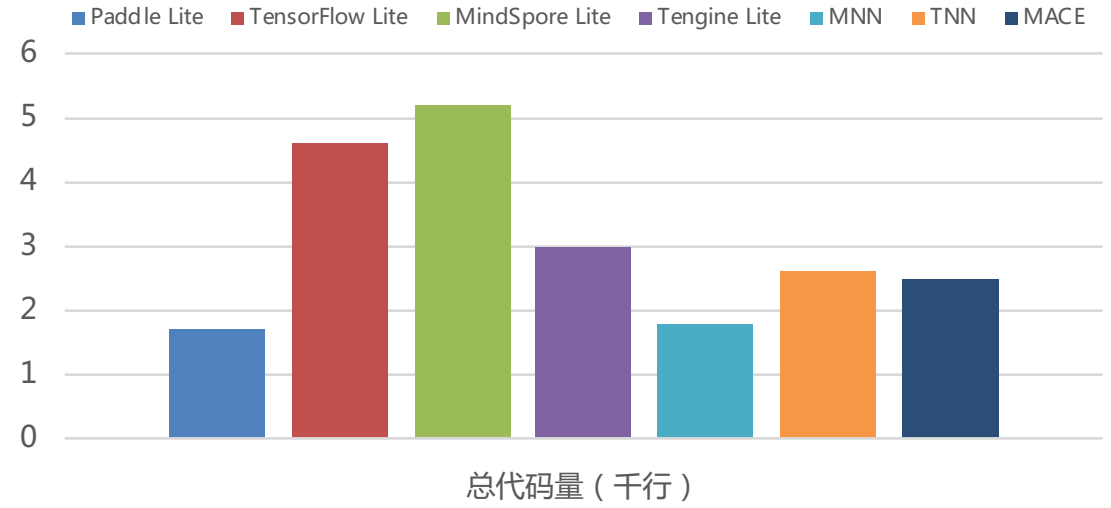
框架文件修改数
（框架耦合度指标）
⬇ 85%

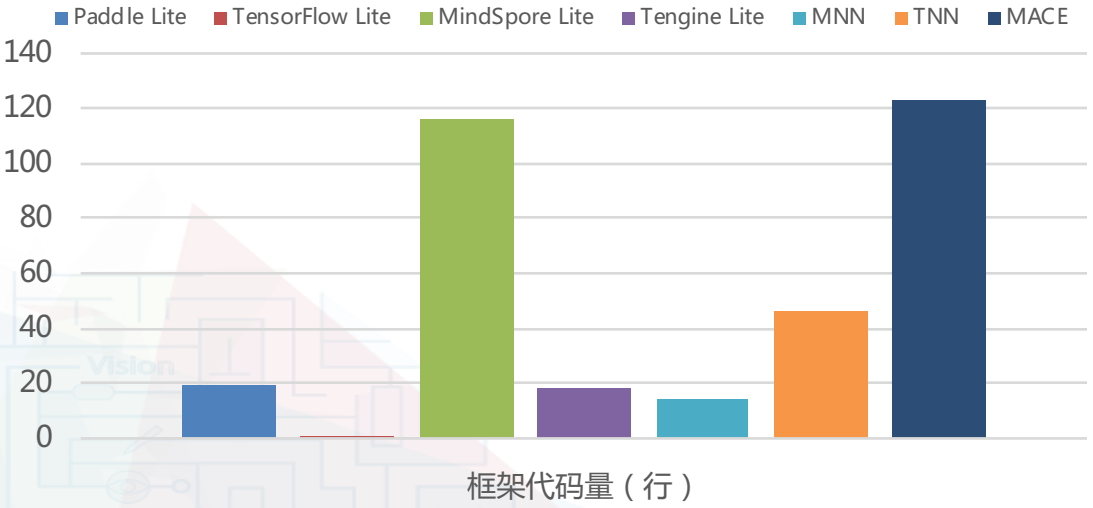
直接子图方案 代码量统计
NNAdapter 方案 代码量统计

* 适配成本是指成功在目标硬件适配 ResNet50 模型所花费的人力成本

附录1：与竞品（在适配、维护成本上）的对比

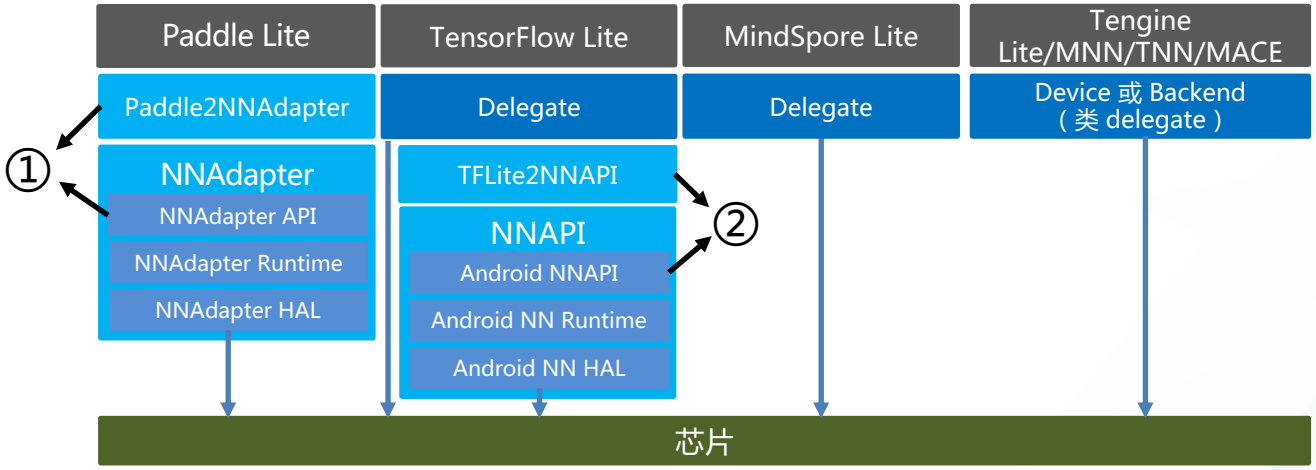
适配成本





详细数据：https://github.com/PaddlePaddle/Paddle-Lite/issues/8342

维护成本



① Paddle2NNAdapter 模块基于 NNAdapter API 提供的标准与硬件无关的设备、组网、模型执行接口，建立 Paddle Lite 的模型、算子、张量到 NNAdapter 的转换，消化了 Paddle Lite 框架变动和算子升级带来的影响，减少了硬件 HAL 层额外的适配工作，尽可能降低 HAL 层的二次维护成本。

② Android NNAPI 提供了类似 NNAdapter API 的标准接口，同样的，TFLite2NNAPI 模块也起到与 Paddle2NNAdapter 类似的作用，使得 Android NN HAL 层能够兼容不同版本的 TFLite 。

附录2：与竞品（在更多指标上）的对比

	Paddle Lite	TensorFlow Lite	MindSpore Lite	Tengine Lite	MNN	TNN	MACE
适配成本 （总代码量、框架代码量、框架文件修改数）	一般 （1.7K、19、1）	一般 （4.6K、1、1）	一般 （5.2K、116、5）	一般 （3.0K、18、2）	一般 （1.8K、14、1）	一般 （2.6K、46、3）	一般 （2.5K、123、17）
维护成本	低 （与框架完全解耦，框架升级被Paddle2NNAdapter层消化）	低 （NNAPI 与框架完全解耦，框架升级被TFLite2NNAPI层消化）	可接受	可接受	可接受	可接受	可接受
是否提供 CI 保障？ （稳定性）	提供算子和模型 CI，CI 较稳定	无 （未展示给外部开发者）	无 （未展示给外部开发者）	算子和部分模型有 CI，CI 不稳定	无 （未展示给外部开发者）	无	无 （未展示给外部开发者）
是否提供硬件接入文档？	较为详细，提供 demo 和参考示例 Pull request	较为详细，提供demo和工具	简单的说明文档和示例代码	简单的说明文档和示例代码	简单的说明文档	无	无
是否提供用户文档？	较为详细、提供预编译库和一键运行demo	说明文档和简单的示例代码，未提供 demo	简单的说明文档，未提供 demo	较为详细的编译文档，未直接提供预编译库和demo	太简单的说明文档	无	无
多硬件支持： 硬件覆盖数量（算子支持个数，开源模型数量）	昇腾（84，76）、晶晨（34）、瑞芯微（34）、联发科（20）、麒麟（61）、颖脉（15）、寒武纪（51）、恩智浦（34）、昆仑（XTCL方式，26）、 NNAPI（20）	除行业芯片，安卓生态内大部分 AI 芯片 NNAPI（90+）	昇腾（140+）、麒麟（65+）	晶晨（39）、瑞芯微（39）、恩智浦（39）	麒麟（45+）	麒麟（59）、瑞芯微（23）	联发科（21）