推理引擎 - 模型转换与优化

模型转换介绍

ZOMI

Talk Overview

1. **推理系统介绍**
   - 推理系统架构
   - 推理引擎叫故
2. **模型小型化**
   - CNN小型化结构
   - Transform小型化结构
3. **离线优化压缩**
   - 低比特量化
   - 模型剪枝
   - 知识蒸馏
4. **模型转换与优化**
   - 架构与流程
   - 模型转换技术
   - 模型离线优化
5. **Runtime与在线优化**
   - 动态batch
   - bin Packing
   - 多副本并行

Talk Overview

I. 模型转换技术

- Principle and architecture - 转换模块挑战与架构
- Model serialization - 模型序列化/反序列化
- protobuffer / flatbuffer 目标文件格式
- IR define - 自定义计算图 IR/Schema
- Technical details -转换流程和技术细节
- ONNX Introduction - ONNX 转换介绍

Talk Overview

I. 模型格式转换

- 转换模块挑战与架构
- 模型序列化/反序列化
- protobuffer / flatbuffer 格式
- 自定义计算图 IR
- 转换流程和技术细节
- ONNX 转换介绍

- **工程理论**
- **知识概念**

Talk Overview

I. 模型格式转换

- 转换模块挑战与架构
- 模型序列化/反序列化
- protobuffer / flatbuffer 格式
- 自定义计算图 IR
- 转换流程和技术细节
- ONNX 转换介绍

- **技术细节**
- **核心内容**

转换模块

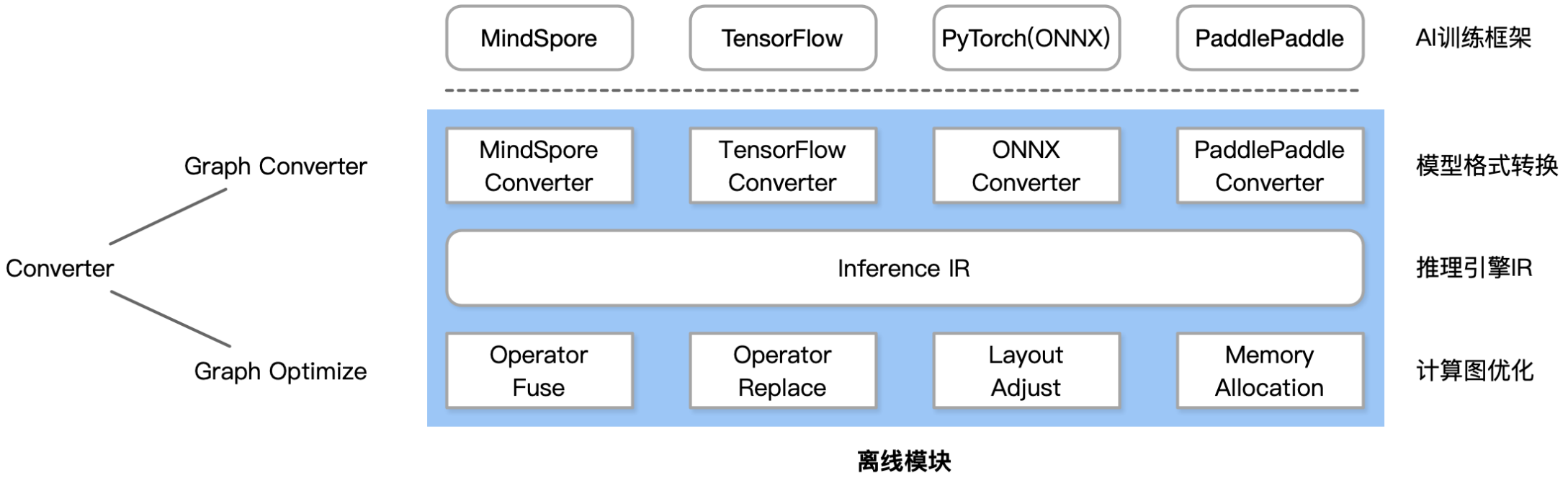
挑战与架构

Converter Challenge 转换模块挑战

1. AI 模型本身包含众多算子，推理引擎需要用有限算子实现不同框架 AI 模型所需要的算子。
2. 支持不同框架 Tensorflow、PyTorch、MindSpore、ONNX 等主流模型文件格式。
3. 支持 CNN / RNN / GAN / Transformer 等主流网络结构。
4. 支持多输入多输出，任意维度输入输出，支持动态输入，支持带控制流的模型。

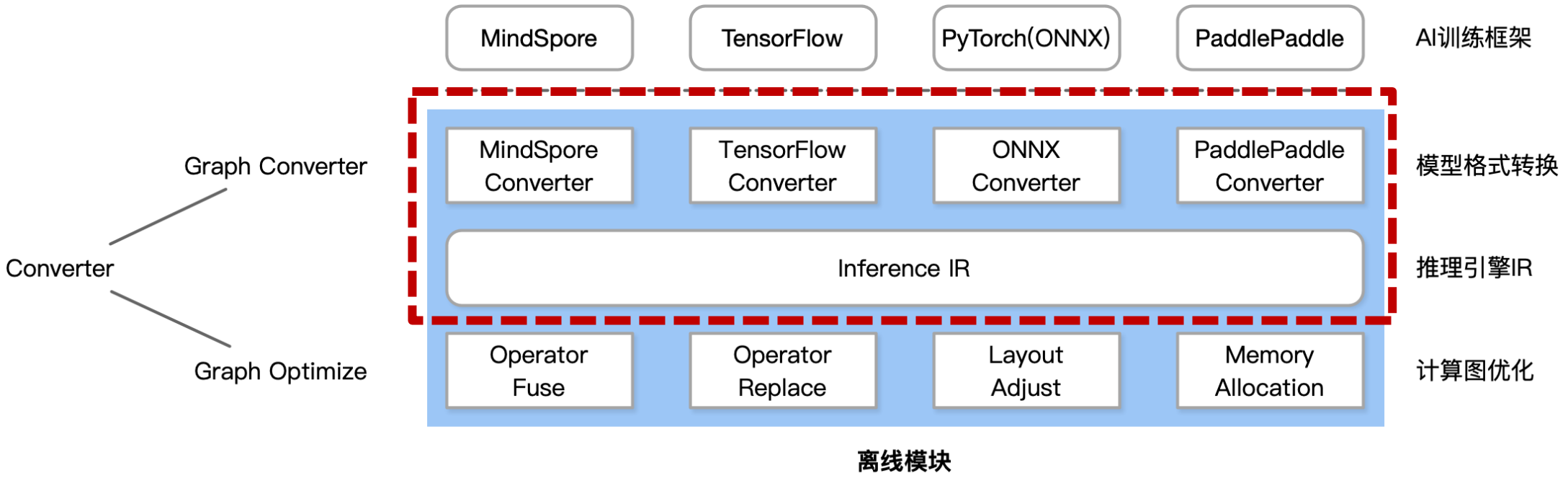
转换模块架构

- Converter由Frontends和Graph Optimize构成。前者负责支持不同的AI 训练框架；后者通过算子融合、算子替代、布局调整等方式优化计算图：

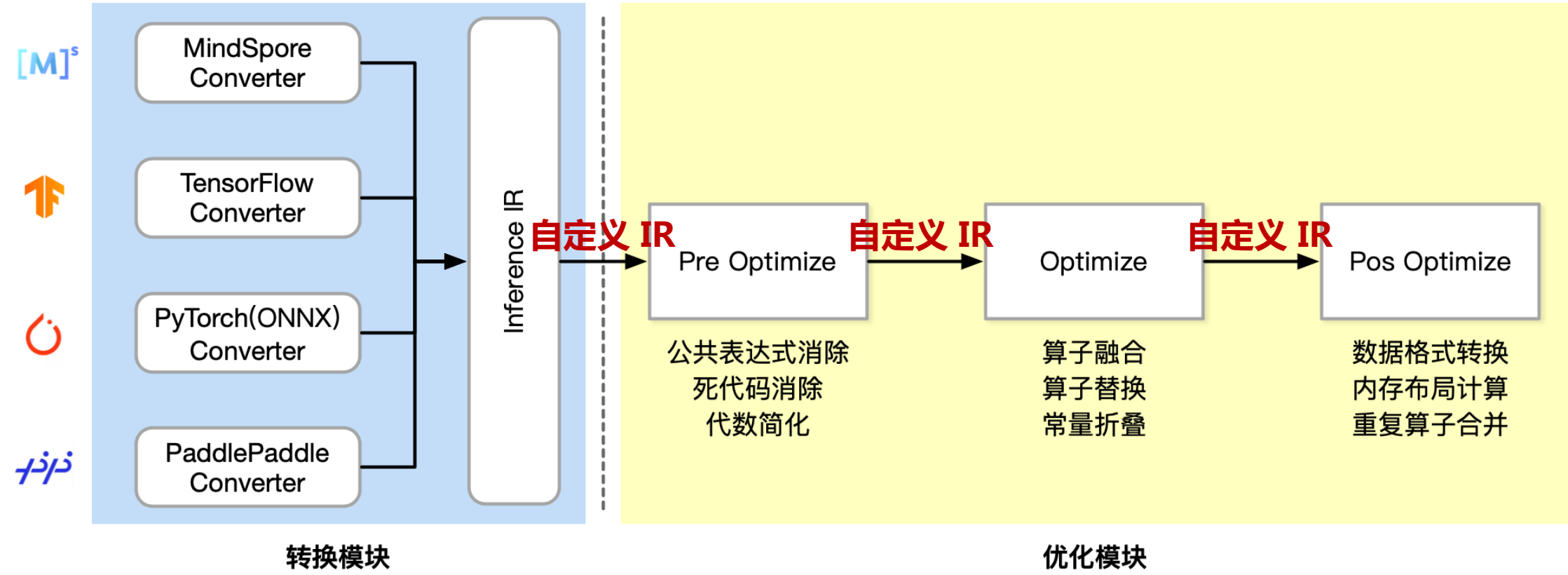


转换模块架构

- Converter由Frontends和Graph Optimize构成。前者负责支持不同的AI 训练框架；后者通过算子融合、算子替代、布局调整等方式优化计算图：



转换模块的工作流程



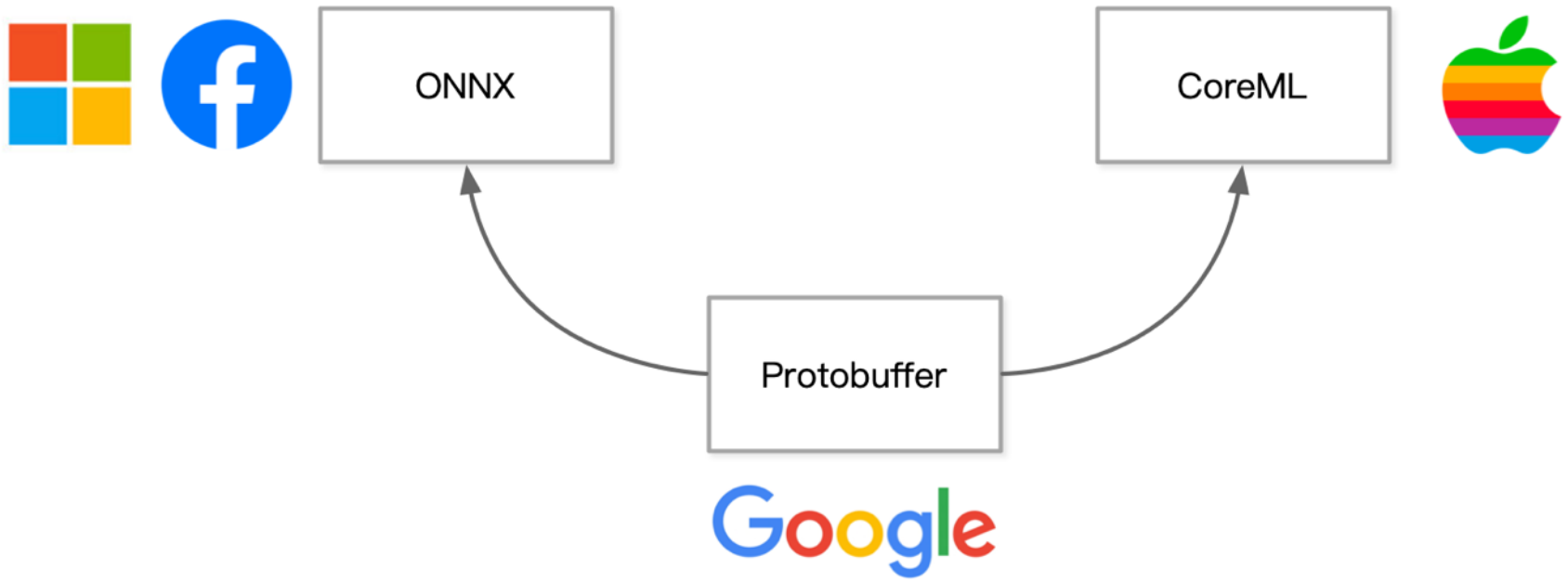
模型序列化

模型序列化

- **模型序列化：**模型序列化是模型部署的第一步，如何把训练好的模型存储起来，以供后续的模型预测使用，是模型部署的首先要考虑的问题。
- **模型反序列化：**将硬盘当中的二进制数据反序列化的存储到内存中，得到网络模型对应的内存对象。无论是序列化与反序列的目的是将数据、模型长久的保存。

序列化分类

- **序列化分类：**跨平台跨语言通用序列化方法，主要优四种格式：XML，JSON，Protobuffer 和 flatbuffer。而使用最广泛为 Protobuffer，Protobuffer为一种是二进制格式。



PyTorch 模型序列化 I

- PyTorch 内部格式只存储已训练模型的状态，主要是对网络模型的权重等信息加载。
- PyTorch 内部格式类似于 Python 语言级通用序列化方法 pickle。
- （包括 weights、biases、Optimizer）

```
# Saving & Loading Model for Inference
torch.save(model.state_dict(), PATH)

model = TheModelClass(*args, **kwargs)
model.load_state_dict(torch.load(PATH))
model.eval()
```

PyTorch 模型序列化 II

- ONNX：内部支持 torch.onnx.export：

```
import torch
import torchvision

dummy_input = torch.randn(10, 3, 224, 224, device="cuda")
model = torchvision.models.alexnet(pretrained=True).cuda()

input_names = [ "actual_input_1" ] + [ "learned_%d" % i for i in range(16) ]
output_names = [ "output1" ]

torch.onnx.export(model, dummy_input, "alexnet.onnx",
                  verbose=True, input_names=input_names,
                  output_names=output_names)
```

目标文件格式

protobuffer / flatbuffer

Protocol Buffers (a.k.a., protobuf)

- protocol buffers 是一种语言无关、平台无关、可扩展的序列化结构数据的方法，它可用于数据通信协议、数据存储等。**特点为**：语言无关、平台无关； 比 XML 更小更快更为简单；扩展性、兼容性好。
- protocol buffers 中可以定义数据的结构，然后使用特殊生成的源代码轻松的在各种数据流中使用各种语言进行编写和读取结构数据。甚至可以更新数据结构，而不破坏由旧数据结构编译的已部署程序。

Protobuffer 文件语法详解

- **基本语法**：字段规则 数据类型 名称 = 域值 [选项 = 选项值]

```
1
2   // 字段规则 数据类型 名称 = 域值 [选项 = 选项值];
3   message Net {
4     optional string name = 'conv_1x1_0_3';
5     repeated Layer layer = 2;
6   }
```

Protobuffer Define the MNIST Network

Writing the Data Layer

```
layer {
  name: "mnist"
  type: "Data"
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "mnist_train_lmdb"
    backend: LMDB
    batch_size: 64
  }
  top: "data"
  top: "label"
}
```

Writing the Convolution Layer

```
layer {
  name: "conv1"
  type: "Convolution"
  convolution_param {
    num_output: 20
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
  }
  bottom: "data"
  top: "conv1"
}
```

Protobuffer Define the MNIST Network

Caffe

使用protobuffer格式写模型定义

```
layer {
  name: "mnist"
  type: "Data"
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "mnist_train_lmdb"
    backend: LMDB
    batch_size: 64
  }
```

TensorFlow

Python 调用TF封装好的API

```

tf.io.encode_proto(
    sizes,
    values,
    field_names,
    message_type,
    descriptor_source='local://',
    name=None
)

```

Protobuffer 编码模式

- 计算机里一般常用的是二进制编码，如int类型由32位组成，每位代表数值2的n次方，n的范围是0-31。Protobuffer 采用 TLV 编码模式，即把一个信息按照 tag-length-value 的模式进行编码。tag 和 value 部分类似于字典的 key 和 value，length 表示 value 的长度，此外 Protobuffer 用 message 来作为描述对象的结构体。

```
message xxx {
    // 字段规则: required -> 字段只能也必须出现 1 次
    // 字段规则: optional -> 字段可出现 0 次或1次
    // 字段规则: repeated -> 字段可出现任意多次（包括 0）
    // 类型: int32、int64、sint32、sint64、string、32-bit ....
    // 字段编号: 0 ~ 536870911（除去 19000 到 19999 之间的数字）
    字段规则 类型 名称 = 字段编号;
}
```

Protobuffer 遍解码过程

1. 根 message 由多个 TLV 形式的 field 组成，解析 message 的时候逐个去解析 field。
2. 由于 field 是 TLV 形式，因此可以知道每个 field 的长度，然后通过偏移上一个 field 长度找到下一个 field 的起始地址。其中 field 的 value 也可以是一个嵌套 message。
3. 对于 field 先解析 tag 得到 field_num 和 type。field_num 是属性 ID，type 帮助确定后面的 value 一种编码算法对数据进行解码。

FlatBuffers

- FlatBuffers 主要针对部署和对性能有要求的应用。相对于 Protocol Buffers，FlatBuffers 不需要解析，只通过序列化后的二进制buffer即可完成数据访问。FlatBuffers 的主要特点有：

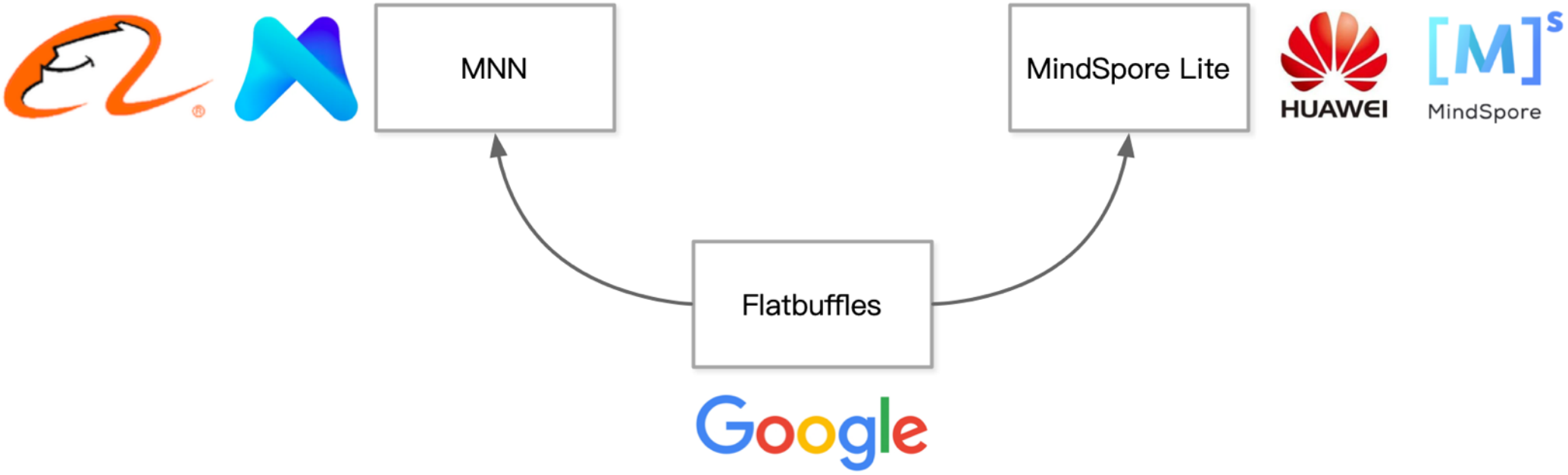
1. 数据访问不需要解析
2. 内存高效、速度快
3. 生成的代码量小
4. 可扩展性强
5. 强类型检测
6. 易于使用

Protobuf VS FlatBuffers

	Proto Bufers	Flatbuffers
支持语言	C/C++, C#, Go, Java, Python, Ruby, Objective-C, Dart	C/C++, C#, Go, Java, JavaScript, TypeScript, Lua, PHP, Python, Rust, Lobster
版本	2.x/3.x，不相互兼容	1.x
协议文件	.proto，需指定协议文件版本	.fbs
代码生成工具	有（生成代码量较多）	有（生成代码量较少）
协议字段类型	bool, bytes, int32, int64, uint32, uint64, sint32, sint64, fixed32, fixed64, sfixed32, sfixed64, float, double, string	bool, int8, uint8, int16, uint16, int32, uint32, int64, uint64, float, double, string, vector

FlatBuffers

- 使用Flatbuffers和Protobuf很相似, 都会用到先定义一个schema文件, 用于定义我们要序列化的数据结构的组织关系。



参考文献

1. Pradana, Muhammad Adna, Andrian Rakhmatsyah, and Aulia Arif Wardana. "Flatbuffers implementation on mqtt publish/subscribe communication as data delivery format." 2019 6th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI). IEEE, 2019.
2. Feng, Jianhua, and Jinhong Li. "Google protocol buffers research and application in online game." IEEE conference anthology. IEEE, 2013.
3. Popić, Srđan, et al. "Performance evaluation of using Protocol Buffers in the Internet of Things communication." 2016 International Conference on Smart Systems and Technologies (SST). IEEE, 2016.
4. Jiang, Xiaotang, et al. "Mnn: A universal and efficient inference engine." Proceedings of Machine Learning and Systems 2 (2020): 1-13.
5. https://google.github.io/flatbuffers/
6. https://github.com/google/flatbuffers
7. https://engineering.fb.com/2015/07/31/android/improving-facebook-s-performance-on-android-with-flatbuffers/
8. https://developers.google.com/protocol-buffers/docs/tutorials?hl=zh-cn
9. https://github.com/protocolbuffers/protobuf
10. https://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html