

Paddle Fluid 开发者指南

1. 为什么需要 PaddlePaddle Fluid?

两个基础问题

1. 如何描述机器学习模型和优化过程?
 - 完备自洽，表达能力足以支持潜在出现的各种计算需求
2. 如何充分利用资源高效计算?
 - 支持异步设备、多卡、分布式计算
 - 降低计算/计算优化的开发成本
 -

如何描述模型和优化过程？

一组连续执行的layers				variable和operator构成的计算图	不再有模型的概念
2013	Caffe, Theano, Torch, PaddlePaddle				
2015		TensorFlow, MxNet, Caffe2, ONNX, n-graph			
2016				PyTorch, TensorFlow Eager Execution, PaddlePaddle Fluid	

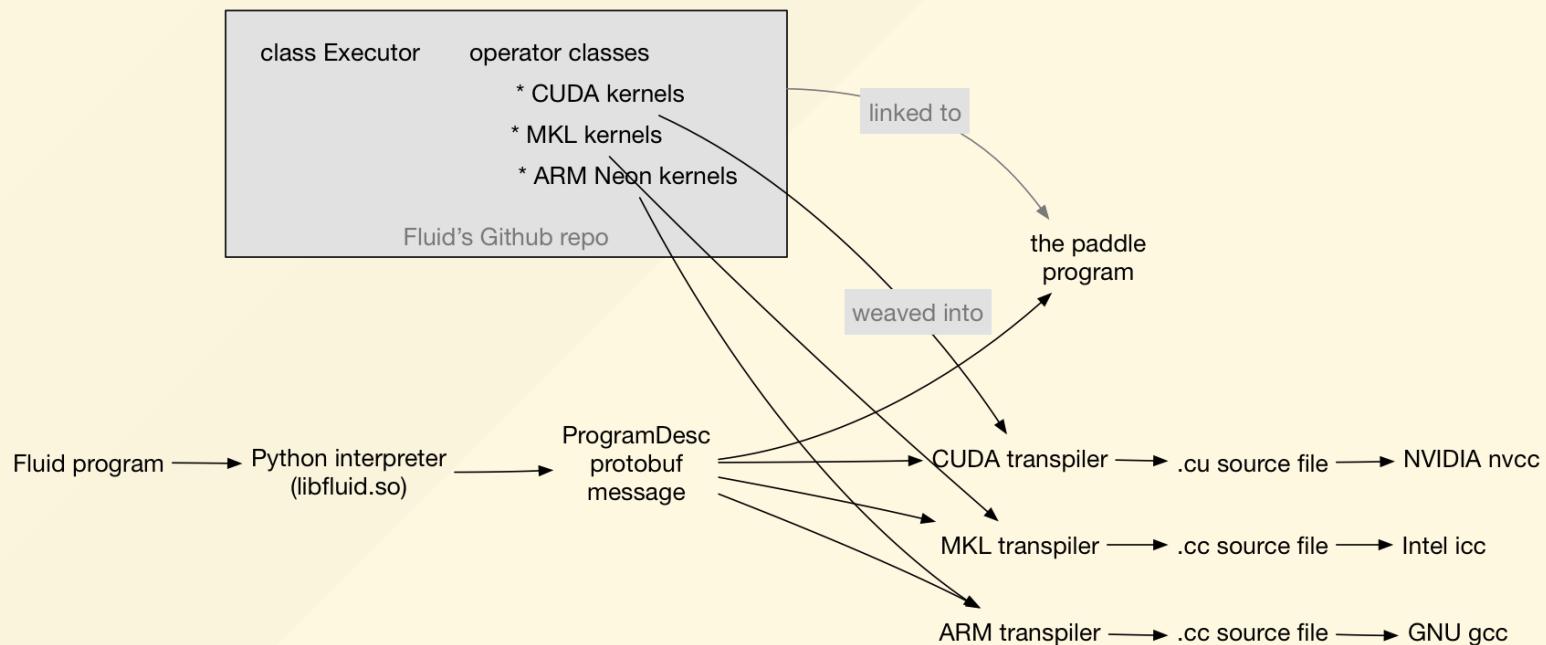
目标 😊

- 提高对各类机器学习任务的描述能力：能够描述潜在出现的任意机器学习模型。
- 代码结构逻辑清晰，各模块充分解耦：内外部贡献者能够专注于自己所需的功能模块，基于框架进行再次开发。
- 从设计上，留下技术优化的空间和潜力。
- 代码解耦后降低多设备支持、计算优化等的开发成本。
- 在统一的设计理念下，实现自动可伸缩，自动容错的分布式计算。

2. Design Overview

Fluid: 系统形态

- 编译器式的执行流程，区分编译时和运行时



让我们在Fluid程序实例中，区分编译时和运行时

Fluid 编译时

- 定义前向计算

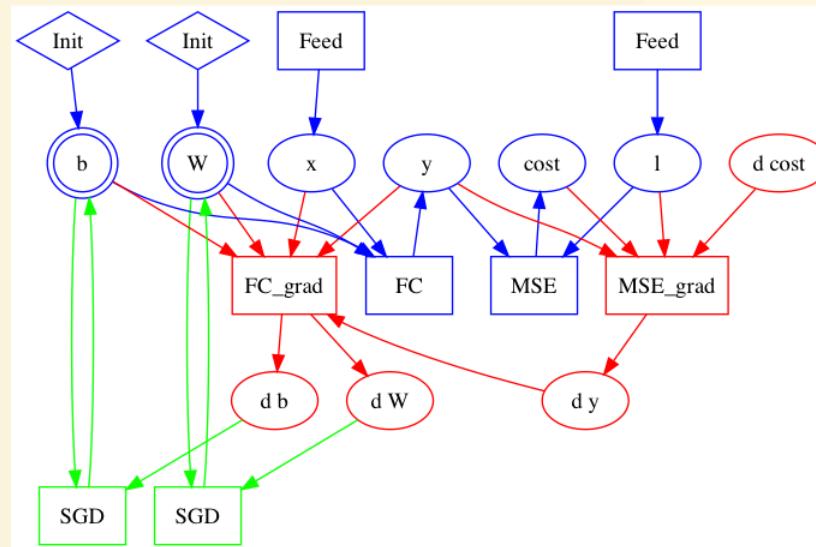
```
x = fluid.layers.data(name='x', shape=[13], dtype='float32')
y_predict = fluid.layers.fc(input=x, size=1, act=None)
y = fluid.layers.data(name='y', shape=[1], dtype='float32')
cost = fluid.layers.square_error_cost(input=y_predict, label=y)
avg_cost = fluid.layers.mean(x=cost)
```

- 添加反向、正则、优化

```
learning_rate = 0.01
sgd_optimizer = fluid.optimizer.SGD(learning_rate)
sgd_optimizer.minimize(avg_cost)
```

Program vs. 计算图

- 在科学计算领域，计算图是一种描述计算的经典方式。下图展示了从前向计算图（蓝色）开始，通过添加反向（红色）和优化算法相关（绿色）操作，构建出整个计算图的过程：



- Fluid 使用 Program 而不是计算图来描述模型和优化过程。Program 由 Block、Operator 和 Variable 构成，相关概念会在后文详细展开。
- 编译时 Fluid 接受前向计算（这里可以先简单的理解为是一段有序的计算流）Program，为这段前向计算按照：前向 → 反向 → 梯度 clip → 正则 → 优化的顺序，添加相关 Operator 和 Variable 到 Program 到完整的计算。

Fluid 运行时

- 读入数据

```
train_reader = paddle.batch(  
    paddle.reader.shuffle(paddle.dataset.uci_housing.train(), buf_size=500),  
    batch_size=20)  
feeder = fluid.DataFeeder(place=place, feed_list=[x, y])
```

- 定义执行程序的设备

```
place = fluid.CPUPlace()  
feeder = fluid.DataFeeder(place=place, feed_list=[x, y])
```

- 创建执行器（Executor），执行初始化 Program 和训练 Program

```
exe = fluid.Executor(place)  
exe.run(fluid.default_startup_program())  
PASS_NUM = 100  
for pass_id in range(PASS_NUM):  
    for data in train_reader():  
        avg_loss_value, = exe.run(fluid.default_main_program(),  
                                  feed=feeder.feed(data),  
                                  fetch_list=[avg_cost])  
        print(avg_loss_value)
```

总结：框架做什么？用户做什么？

构建训练	执行训练
用户：描述前向运算	框架：创建Operator（计算） + Variable（数据）
框架：添加反向运算	框架：创建Block
框架：添加优化运算	框架：内存管理/设备管理
框架：添加内存优化	框架：执行计算
框架：添加并行/多设备/分布式相关的计算单元	

总结：编译时

用户编写一段Python程序，描述模型的前向计算

1. 创建变量描述 `VarDesc`
2. 创建operators的描述 `OpDesc`
3. 创建operators的属性
4. 推断变量的类型和形状，进行静态检查：`inferShape`
5. 规划变量的内存复用
6. 创建反向计算
7. 添加优化相关的Operators
8. （可选）添加多卡/多机相关的Operator，生成在多卡/多机上运行的程序

总结：运行时

执行规划好的计算

1. 创建 Executor

2. 为将要执行的一段计算，在层级式的 Scope 空间中创建 Scope

3. 创建 Block，依次执行 Block

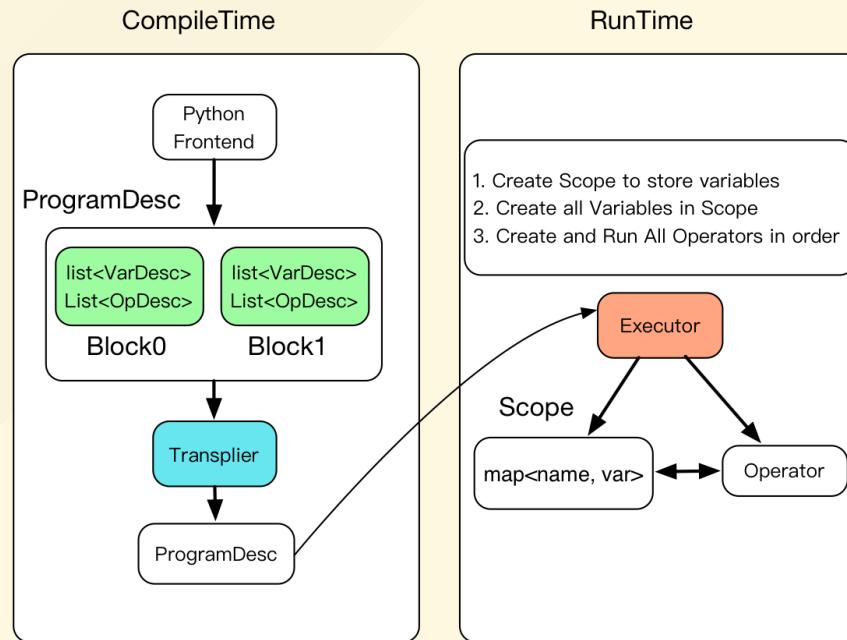


Figure. 编译时运行时概览

3. 用户如何描述计算？

Fluid: 像写程序一样定义计算

- 顺序执行

```
x = fluid.layers.data(name='x', shape=[13], dtype='float32')
y_predict = fluid.layers.fc(input=x, size=1, act=None)
y = fluid.layers.data(name='y', shape=[1], dtype='float32')
cost = fluid.layers.square_error_cost(input=y_predict, label=y)
```

- 条件分支: switch、ifelse

```
a = fluid.Var(10)
b = fluid.Var(0)

switch = fluid.switch()
with switch.block():
    with switch.case(fluid.less_equal(a, 10)):
        fluid.print("Case 1")
    with switch.case(fluid.larger(a, 0)):
        fluid.print("Case 2")
    with switch.default():
        fluid.print("Case 3")
```

- “ A Lisp cond form may be compared to a continued if-then-else as found in many algebraic programming languages.”

Fluid: 像写程序一样定义计算

- 循环: [while](#)

```
d0 = layers.data("d0", shape=[10], dtype='float32')
data_array = layers.array_write(x=d0, i=i)
array_len = layers.fill_constant(shape=[1], dtype='int64', value=3)

cond = layers.less_than(x=i, y=array_len)
while_op = layers.While(cond=cond)
with while_op.block():
    d = layers.array_read(array=data_array, i=i)
    i = layers.increment(x=i, in_place=True)
    layers.array_write(result, i=i, array=d)
    layers.less_than(x=i, y=array_len, cond=cond)
```

- 完整实例请点查看 [→](#)
- beam search [→](#)

总结

1. 用户层提供的描述语法具有完备性、自治性，有能力支持对复杂计算过程描述
2. 使用方式和核心概念可以类比编程语言，认知能够直接迁移
3. 能够支持：定义问题，逐步求解

3. 核心概念

编译时概念：变量和计算的描述

- `VarDesc + TensorDesc + OpDesc → BlockDesc → ProgramDesc`
 - <https://github.com/PaddlePaddle/Paddle/blob/develop/paddle/framework/framework.proto>
- 什么是 Fluid Program
 - 在Fluid中，一个神经网络任务（训练/预测）被描述为一段 `Program`
 - `Program` 包含对 `Variable`（数据）和 `Operator`（对数据的操作）的描述
 - `Variable` 和 `Operator` 被组织为多个可以嵌套的 `Block`，构成一段完整的 `Fluid Program`
- “ 编译阶段最终，经过 Transpiler 的执行规划，变换处理，生成使用 `protobuf` 序列化后的 `ProgramDesc`。可以发送给多卡或者网络中的其它计算节点执行 ”

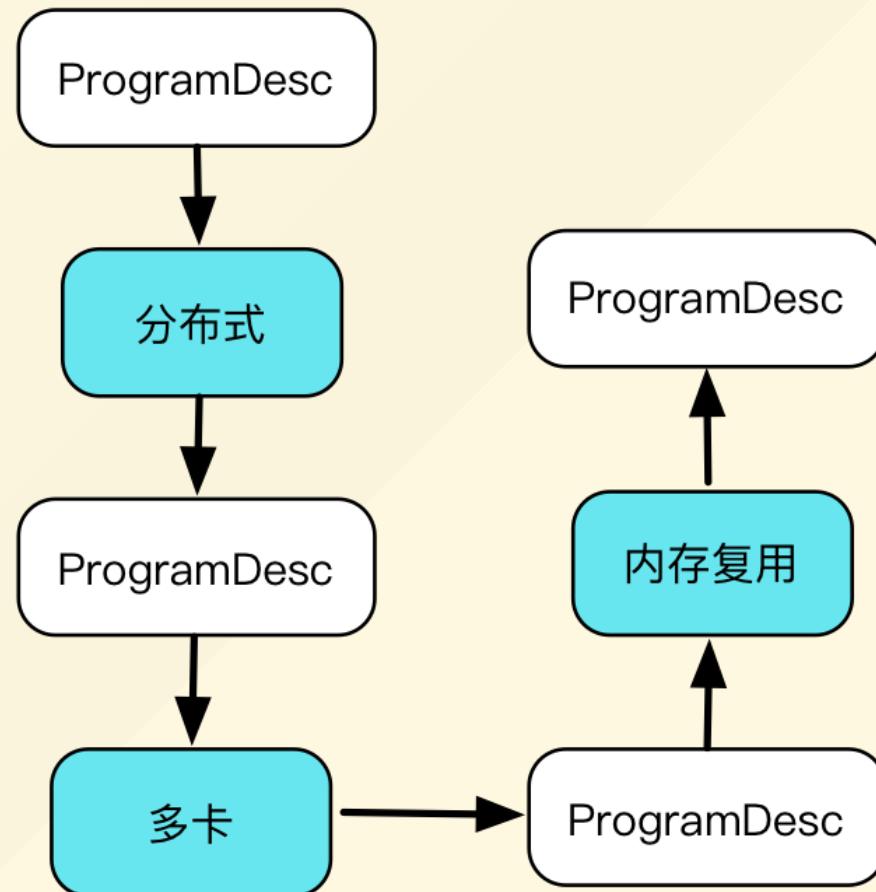
编译时概念：Transpiler

1. 接受一段 `ProgramDesc` 作为输入，生成一段新的 `ProgramDesc`

- *Memory optimization transpiler*: 向原始 `ProgramDesc` 中插入 `FreeMemoryOps`，在一次迭代优化结束前提前释放内存，使得能够维持较小的 memory footprint
- *Distributed training transpiler*: 将原始的 `ProgramDesc` 中转化为对应的分布式版本，生成两段新的 `ProgramDesc`：
 1. trainer 进程执行的 `ProgramDesc`
 2. parameter server 执行的 `ProgramDesc`

2. **WIP**: 接受一段 `ProgramDesc`，生成可直接被 `gcc`, `nvcc`, `icc` 等编译的代码，编译后得到可执行文件

Transplier



打印 ProgramDesc

```
x = fluid.layers.data(name='x', shape=[13], dtype='float32')
y_predict = fluid.layers.fc(input=x, size=1, act=None)
y = fluid.layers.data(name='y', shape=[1], dtype='float32')
cost = fluid.layers.square_error_cost(input=y_predict, label=y)
avg_cost = fluid.layers.mean(x=cost)
sgd_optimizer = fluid.optimizer.SGD(learning_rate=0.001)
sgd_optimizer.minimize(avg_cost)

exe = fluid.Executor(place)
exe.run(fluid.default_startup_program())
print(fluid.default_startup_program().to_string(True))
print(fluid.default_main_program().to_string(True))
```

- `default_startup_program`: 创建可学习参数，对参数进行初始化
- `default_main_program`: 由用户定义的模型，包括了前向、反向、优化及所有必要的计算
- 打印可读的 `Program`

```
from paddle.v2.fluid import debugger
print debugger pprint_program_codes(framework.default_main_program().desc)
```

输出效果

variable in block 0

```
blocks {
  idx: 0
  parent_idx: -1
  vars {
    name: "fc_0.w_0@GRAD"
    type: LOD_TENSOR
    lod_tensor {
      tensor {
        data_type: FP32
        dims: 13
        dims: 1
      }
    }
  }
  vars {
    name: "fc_0.tmp_1@GRAD"
    type: LOD_TENSOR
    lod_tensor {
      tensor {
        data_type: FP32
        dims: -1
        dims: 1
      }
    }
  }
}
....
```

variable in block 0

```
ops {
  inputs {
    parameter: "X"
    arguments: "x"
  }
  inputs {
    parameter: "Y"
    arguments: "fc_0.w_0"
  }
  outputs {
    parameter: "Out"
    arguments: "fc_0.tmp_0"
  }
  type: "mul"
  attrs {
    name: "y_num_col_dims"
    type: INT
    i: 1
  }
  attrs {
    name: "x_num_col_dims"
    type: INT
    i: 1
  }
}
```

运行时概念

- 数据相关

- `Tensor` / `LoDTensor` / `Variable`
- `Scope`

- 计算相关

- `Block`
- `Kernel`、`OpWithKernel`、`OpWithoutKernel`

protobuf messages C++ class objects		
Data	<u>VarDesc</u>	<u>Variable</u>
Operation	<u>OpDesc</u>	<u>Operator</u>
Block	BlockDesc	Block

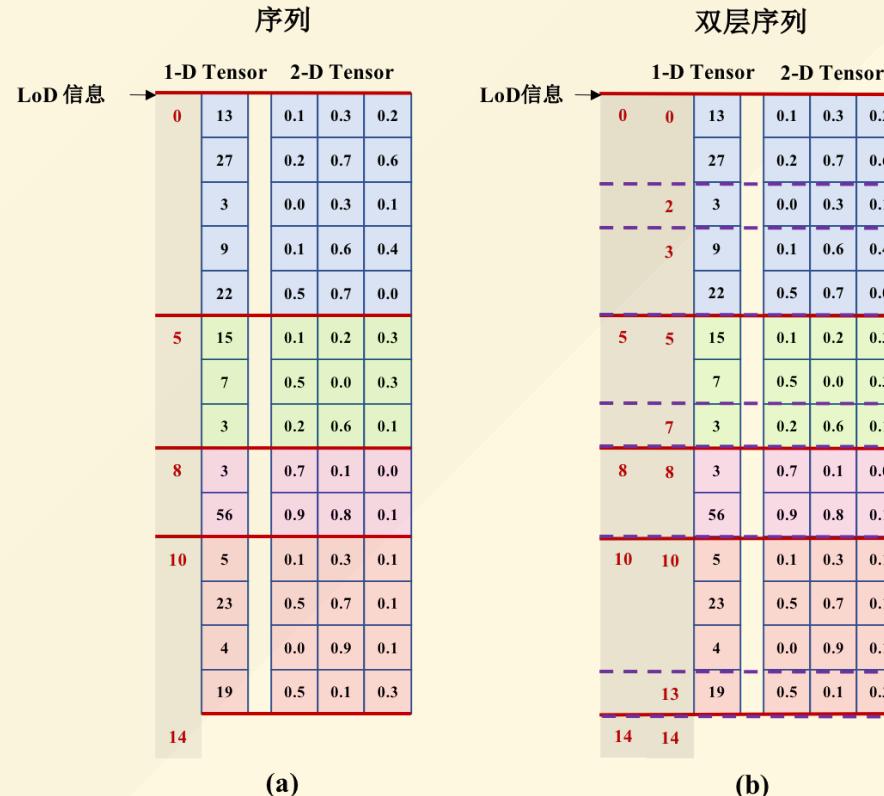
- 执行相关 : `Executor`

Tensor 和 LoD(Level-of-Detail) Tensor

- Tensor 是 n -dimensional array的推广，LoDTensor是在Tensor基础上附加了序列信息
- Fluid中输入、输出，网络中的可学习参数全部统一使用LoDTensor (n-dimension array) 表示
- 一个mini-batch输入数据是一个LoDTensor
 - 在Fluid中，RNN 处理变长序列无需padding，得益于 LoDTensor 表示
 - 可以简单将 LoD 理解为： `std::vector<std::vector<int>>`
 - 对非序列数据，LoD 信息为空

	TensorFlow	PaddlePaddle
RNN	Support	Support
recursive RNN	Support	Support
padding zeros	Must	No need
blob data type	Tensor	LoDTensor

LoD 信息实例



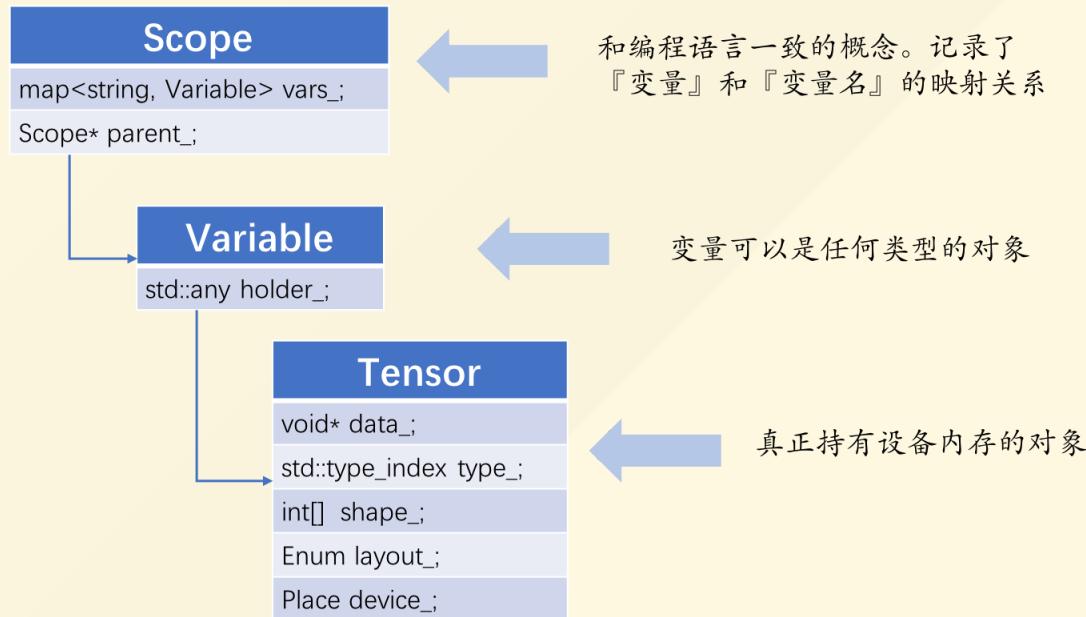
- 图(a)的LoD 信息

```
[0, 5, 8, 10, 14]
```

- 图(b)的 LoD 信息

```
[[0, 5, 8, 10, 14] /*level=1*/, [0, 2, 3, 5, 7, 8, 10, 13, 14] /*level=2*/]
```

Tensor, Variable, Scope 之间的关系

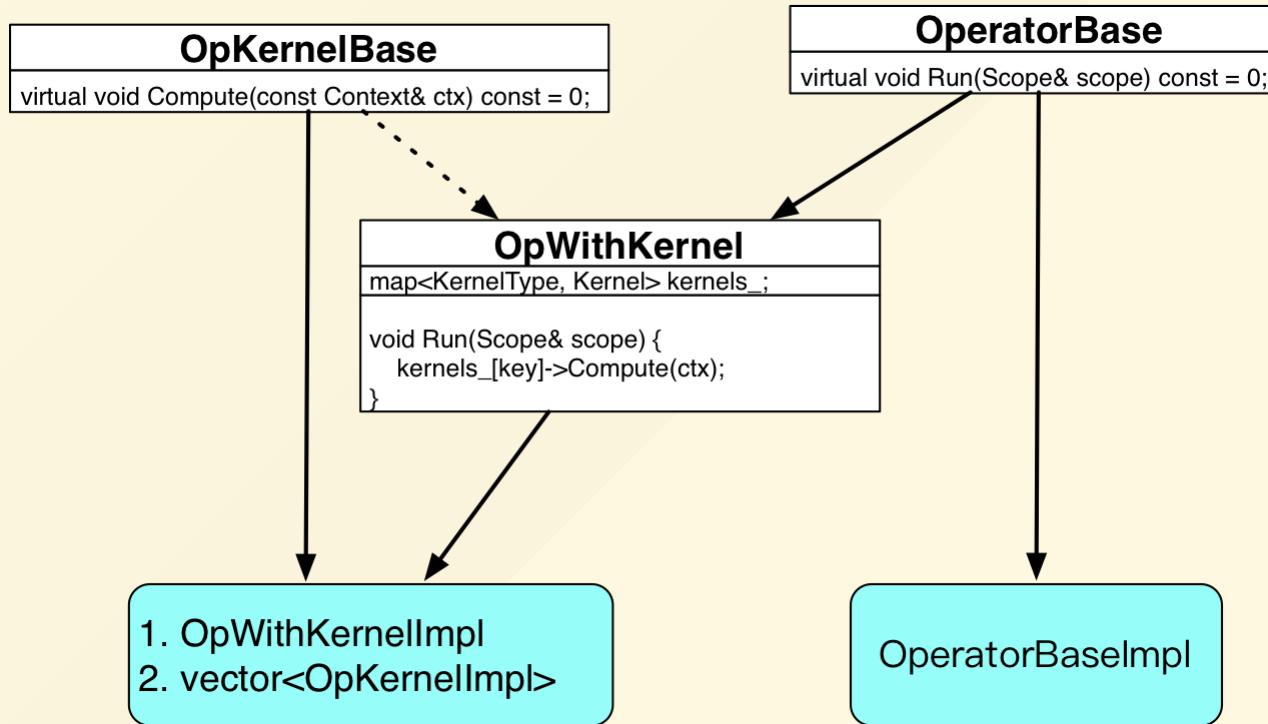


1. **Block** 是一个实现层的概念，不在应用层暴露给用户。目前用户无法自行创建并利用 **Block**，用户能够感知的只有 **Program** 这个概念。
2. 逻辑上，可以将 **Block** 类比为编程语言中的大括号：定义了一段作用域，其中运行一段代码
3. **Executor** 会为每一个 **Block** 创建一个 **Scope**，**Block** 是可嵌套的，因此 **Scope** 也是可嵌套的

Executor

接口	说明
<pre>class Executor { public: void Run(const ProgramDesc& pdesc, Scope* scope, int block_id) { auto& block = pdesc.Block(block_id); // create all variables for (auto& var : block.AllVars()) { scope->Var(var->Name()); } // create op and run it one by one for (auto& op_desc : block.AllOps()) { auto op = CreateOp(*op_desc); op->Run(*local_scope, place_); } } };</pre>	<p>输入</p> <ol style="list-style-type: none">1. ProgramDesc2. Scope3. block_id <p>解释执行步骤</p> <ol style="list-style-type: none">1. 创建所有 Variables2. 逐一创建 Operator 并运行

Operator/OpWithKernel/Kernel



- operator 无状态， Operator的核心是Run方法
- 一个operator可以注册多个kernel
- operator 可以无 kernel: while_op 、 ifelse op

Fluid Operator vs. PaddlePaddle layers

Layer	Operator
<pre>class Layer { public: virtual void forward() = 0; virtual void backward() = 0; private: std::vector<Parameter> paramters_; std::vector<Argument> inputs_; std::vector<Argument> outputs_; std::vector<Argument> inner_states; };</pre>	<pre>class Operator { public: virtual void Run(const Scope& scope) const = 0; private: std::vector<std::string> inputs_; std::vector<std::string> outputs_; std::vector<Attrs> attrs_; };</pre>
<ol style="list-style-type: none">1. 内部维护状态2. 包含forward和backward方法	<ol style="list-style-type: none">1. 内部无状态2. 只有Run方法

4. 内存管理

目标

- 为异构设备提供统一的内存分配、回收接口
- 最小化管理内存所需的时间，最小化管理开销
- 减少内存碎片
- 将内存管理与计算（Operators/Kernels）完全剥离
- 统一内存管理是内存优化的基础

Memory 接口

- 内存管理模块向上层应用逻辑提供三个基础接口：

```
template <typename Place>
void* Alloc(Place place, size_t size);

template <typename Place>
void Free(Place place, void* ptr);

template <typename Place>
size_t Used(Place place);

struct Usage : public boost::static_visitor<size_t> {
    size_t operator()(const platform::CPUPlace& cpu) const;
    size_t operator()(const platform::CUDAPlace& gpu) const;
};
```

- 模板参数 `Place` 指示内存分配发生的设备
- 实现时，需特化支持的 `Place`， 提供以上三个接口的实现

代码结构

内存管理模块可以理解为由以下两部分构成：

1. SystemAllocator：实际从物理设备上分配、释放的内存的接口
2. BuddyAllocator：内存管理算法

System Allocator

- SystemAllocator 是实现物理内存分配、回收的基类
 - 不同设备上的内存分配和回收终将转化为标准接口调用
 - 为不同设备实现MemoryAllocator，继承自SystemAllocator

```
class SystemAllocator {
public:
    virtual ~SystemAllocator() {}
    virtual void* Alloc(size_t& index, size_t size) = 0;
    virtual void Free(void* p, size_t size, size_t index) = 0;
    virtual bool UseGpu() const = 0;
};
```

CPU/GPU Allocator

```
class CPUAllocator : public SystemAllocator {
public:
    virtual void* Alloc(size_t& index, size_t size);
    virtual void Free(void* p, size_t size, size_t index);
    virtual bool UseGpu() const;
};

#ifndef PADDLE_WITH_CUDA
class GPUAllocator : public SystemAllocator {
public:
    virtual void* Alloc(size_t& index, size_t size);
    virtual void Free(void* p, size_t size, size_t index);
    virtual bool UseGpu() const;
private:
    size_t gpu_alloc_size_ = 0;
    size_t fallback_alloc_size_ = 0;
};
#endif
```

- CPUAllocator和GPUAllocator分别继承自SystemAllocator， 分别调用相应的标准库函数实现物理内存的分配和释放。
- 一旦大块、连续的物理内存分配之后， 将通过内存管理算法实现内存的按块分配、回收、重用等。

CPU Allocator

- CPU 内存的分配提供两种选项：
 1. non-pinned memory： 可分页内存
 2. pinned memory： 页锁定内存
 - 分配过大的页锁定内存有可能因为系统可使用的分页内存减少，影响系统性能， 默认CPU下分配的是可分页内存
- 通过gflags进行设置一次性分配内存的大小以及是否使用页锁定内存。

```
DEFINE_bool(use_pinned_memory, true, "If set, allocate cpu pinned memory.");
DEFINE_double(fraction_of_cpu_memory_to_use, 1,
             "Default use 100% of CPU memory for PaddlePaddle,"
             "reserve the rest for page tables, etc");
```

GPU Allocator

- 通过 cudaMalloc 分配GPU显存
- GPUAllocator::Alloc 首先会计算指定GPU device上的可用显存
 - 如果可用显存小于请求分配大小，调用cudaMalloc进行分配
 - 如果可用显存不足，目前会报错退出。
- 通过gflags控制GPU下一次性分配显存的大小：

```
DEFINE_double(fraction_of_gpu_memory_to_use, 0.92,
             "Default use 92% of GPU memory for PaddlePaddle,"
             "reserve the rest for page tables, etc");
```

内存管理算法: Buddy Memory Allocation

- Memory Arena: 一次性分配大块连续内存，之后会基于这块内存进行内存管理：动态分配、释放、重用内存块。
- 伙伴内存分配：
 - 将内存划分为 2 的幂次方个分区，使用 best-fit 方法来分配内存请求。
 - 当释放内存时，检查 buddy 块，查看相邻的内存块是否也被释放。如果是，将内存块合并，以最小化内存碎片。
 - 分配的内存在物理内存的自然边界对齐，提高内存访问效率。
 - 算法的时间效率高，单使用 best-fit 方法的缘故，会产生一定的内存浪费

Buddy Allocator

- BuddyAllocator 是一个单例，每个设备（如：GPU/CPU(0)/GPU(1)）拥有一个BuddyAllocator
- BuddyAllocator 内部拥有一个私有成员变量 SystemAllocator
- 当请求的内存超过BuddyAllocator管理的空余内存时，将会调用 SystemAllocator去指定的设备上分配物理内存

实例：CPU下内存管理接口的实现

- 对上层应用，统一通过BuddyAllocator来实现内存的分配、释放以及用量查询

```
template <>
void* Alloc<platform::CPUPlace>(platform::CPUPlace place, size_t size) {
    VLOG(10) << "Allocate " << size << " bytes on " << platform::Place(place);
    void* p = GetCPUBuddyAllocator()->Alloc(size);
    VLOG(10) << " pointer=" << p;
    return p;
}

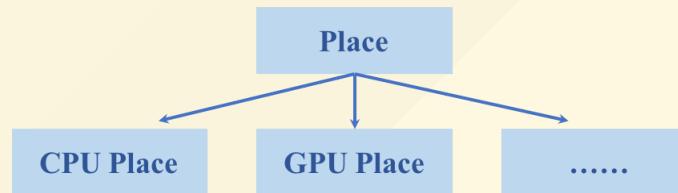
template <>
void Free<platform::CPUPlace>(platform::CPUPlace place, void* p) {
    VLOG(10) << "Free pointer=" << p << " on " << platform::Place(place);
    GetCPUBuddyAllocator()->Free(p);
}

template <>
size_t Used<platform::CPUPlace>(platform::CPUPlace place) {
    return GetCPUBuddyAllocator()->Used();
}
```

5. 多设备支持

多设备支持（一）

- step 1: 添加Place类型, 由用户实现添加到框架
 - 可以将Place类型理解为一个整数加上一个枚举型, 包括: 设备号 + 设备类型



- DeviceContext
 - 不同的Place会对应一个相应的DeviceContext, 用于组织管理与设备相关的信息
 - 例如, GpuDeviceContext中会管理Cuda stream
 - 目前实现中一些特殊的库也会对应有自己的DeviceContext: 例如:

```
class MKLDNNDeviceContext : public CPUDeviceContext {.....}
```

- 每种设备对应的DeviceContext需要管理的内容不尽相同, 视具体需求来实现

多设备支持（二）

- step 2: 增加KernelType，为相应的KernelType注册Kernel对象，由用户实现注册给框架 可以按照：

1. Place 执行设备
2. DataType 执行数据类型 FP32/FP64/INT32/INT64
3. Memory layout: 运行时 Tensor 在内存中的排布格式 NCHW、NHWC
4. 使用的库

来区分Kernel，为同一个operator注册多个 Kernel。

```
struct OpKernelType {
    proto::DataType data_type_;
    DataLayout data_layout_;
    platform::Place place_;
    LibraryType library_type_;
}
```

多设备支持（三）

step 3: 运行时的 KernelType 推断和Kernel切换，按需要修改Kernel推断和 Kernel切换规则

- Expected Kernel: 期待调用的Kernel: 由 (1) Place 和计算精度决定；或 (2) 用户在配置中显示指定使用的计算库，如 cudnn、mkldnn 等。
- Actual Kernel: 运行时从 Operator 的输入 (Variable) 可以推断出实际需要的 KernelType
- 当 Expected Kernel 和 Actual Kernel 不一致的时候，框架会插入 data_transformer 或者 data_layerout_transform 等，保证 Expected Kernel 可以执行，包括：
 - CPUPlace ➔ GPUPlace : 跨设备内存复制
 - NCHW ➔ nChw8c : Layout 转换
 - FP32 ➔ FP16 : 精度转换 尚未支持
 -
- 以上过程实现在 OperatorWithKernel 类的 Run 方法中 ➔

6. while_op

while_op

- 循环执行一段 Program，直到条件operator判断循环条件不满足时终止循环
- while_op 的特殊之处：
 1. while_op 没有 kernel
 2. while_op 拥有自己的 Block，会形成一段嵌套的 Block
 3. [while_op 内部创建了一个 Executor，来循环执行 Block](#)
- while_op 输入输出： LoDTensorArray

```
namespace paddle {
namespace framework {
using LoDTensorArray = std::vector<LoDTensor>;
}
```

- 每一次循环，从原始输入中“切出”一个片段
- LoDTensorArray 在Python端暴露，是Fluid支持的基础数据结构之一，用户可以直接创建并使用

while_op Run 方法概览

```
void Run(const framework::Scope &scope,
         const platform::Place &dev_place) const override {
    PADDLE_ENFORCE_NOT_NULL(scope.FindVar(Input(kCondition)));
    auto &cond = scope.FindVar(Input(kCondition))->Get<LoDTensor>();
    PADDLE_ENFORCE_EQ(cond.dims(), paddle::framework::make_ddim({1}));

    framework::Executor executor(dev_place);
    auto *block = Attr<framework::BlockDesc *>(kStepBlock);

    auto *program = block->Program();
    auto step_scopes =
        scope.FindVar(Output(kStepScopes))->GetMutable<StepScopeVar>();

    while (cond.data<bool>()[0]) {
        auto &current_scope = scope.NewScope();
        step_scopes->push_back(&current_scope);
        executor.Run(*program, &current_scope, block->ID(),
                     false /*create_local_scope*/);
    }
}
```

while_op 的重要应用：Dynamic RNN

什么是 dynamicRNN ?

1. 用户可以自定义在一个时间步之内的计算, 框架接受序列输入数据, 在其上循环调用用户定义的单步计算
2. 可学习参数在多个时间步之间共享
3. dynamicRNN 由 while_op 实现
4. 如果 dynamicRNN 中定义了 memory , 将会构成一个循环神经网络, 否则其行为就等于在输入序列上循环调用预定义的单步计算

dynamic RNN 用户接口

```
rnn = fluid.layers.DynamicRNN()
with rnn.block():
    current_word = rnn.step_input(trg_embedding)
    mem = rnn.memory(init=encoder_out)          memory is used to form the recurrent connection.
                                                It can be initialized by the output of an operator.
                                                It "points" to output of another operator.
    fc1 = fluid.layers.fc(input=[current_word, mem],
                          size=decoder_size,
                          act='tanh')
    out = fluid.layers.fc(input=fc1, size=target_dict_dim, act='softmax')

    rnn.update_memory(mem, fc1)                 memory is "forwarded" after the operator it pointing to
                                                is forwarded.

    rnn.output(out)

return rnn()      The step function defines computation in one timestep.
```

- dynamicRNN 中的重要元素

1. **step input**: dynamicRNN 每个时间步的输入
2. **step function**: 用户定义的单步计算
3. **memory**: 用于形成循环连接
4. **external/static memory**: 单步计算的每一步都可以全部读取到的外部输入

dynamicRNN 中的 Memory

dynamicRNN 中 `memory` 的行为非常类似于 C++ 中的引用变量

- `memory` “指向”一个operator的输出变量，记作： A
- `memory` 可以被 LoDTensor 初始化（当LoD信息为空时，为非序列，否则为序列）,默认 `memory` 被初始化为零
- `memory` 在 operator A 前向计算之后，进行前向计算
- 当 `memory` 的前向计算会 "指向" A 的输出 LoDTensor
- `memory` 的输出可以是另一个 operator 的输入，于是形成了“循环”连接

DynamicRNN 实现细节

- `while_op` 无法独立构成dynamicRNN，必须和一组相关的 operator 及数据结构配合
 - 依赖的 operators (这里仅列出最重要的，并非全部):
 - `lod_rank_table` operator
 - `lod_tensor_to_array` operator
 - `array_to_lod_tensor` operator
 - `shrink_memory` operator
 - 依赖的数据结构
 - `TensorArray`
 - `LoDRankTable`
- 在Fluid中，RNN接受变长序列输入，无需填充，以上数据结构和相关的operator配合工作，实现了对变长输入以batch计算

dynamicRNN 如何实现 batch 计算？

- 问题：
 - RNN 可以看作是一个展开的前向网络，前向网络的深度是最长序列的长度
 - 如果不对变长序列进行填充，将它们填充到一样长度，每个mini-batch输入将会不等长，每个样本展开长度不一致，导致前向和反向计算实现困难

实例：RNN encoder-decoder with attention

- 以机器翻译的RNN encoder-decoder 模型（涉及了dynamicRNN的所有设计要素）为例，下图是 RNN encoder-decoder 的原始输入：

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18		
19	20	21	22	23					
24	25	26	27	28	29	30			

source word sequences

1	2	3	4	5
6	7	8	9	10
11	12			
13	14	15	16	
17	18	19	20	21
22				

target word sequences

Figure. RNN encoder-decoder 原始batch 输入数据

- source word sequences 是encoder RNN的输出，是一个LoDTensor
- target word sequences 是look_upable的输入，是一个LoDTensor
- 上图中一个矩形方块是CPU/GPU内存中一片连续的内存空间，表示一个dense vector

dynamicRNN 如何实现 batch 计算？

1. 对一个mini batch中不等长样本进行排序，最长样本变成batch中的第一个，最短样本是batch中最后一个
 - LoDTensor → LoDRankTable + lod_rank_table operaator
 - 可以将LoDRankTable理解为对LoDTensor中的多个序列按照长度排序LoDRankTable存储了排序之后的index
2. 构建每个时间步的batch输入：随着时间步增加，每个时间步的batch输入可能会逐渐缩小
 - TensorArray + lod_tensor_to_array → LoDTensor (without LoD)
3. 每个时间步输出写入一个输出 LoDTensorArray
4. dynamicRNN循环结束后，按照LoDRankTable中记录的信息对输出LoDTensorArray重排序，还原会原始输入顺序
 - TensorArray + array_to_lod_tensor → LoDTensor

运行实例

Input 1 for dynamicRNN:

source word sequences, **this is the external memory.**

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18		
19	20	21	22	23					
24	25	26	27	28	29	30			

11	12	13	14	15	16	17	18		
24	25	26	27	28	29	30			
1	2	3	4	5	6	7	8	9	10
19	20	21	22	23					

External memory is sorted according to LoDRankTable created from the **step input**

Input2 for dynamicRNN:

target word sequences, **this is the step input.**

1	2	3	4	5			
6	7	8	9	10	11	12	
13	14	15	16				
17	18	19	20	21	22		



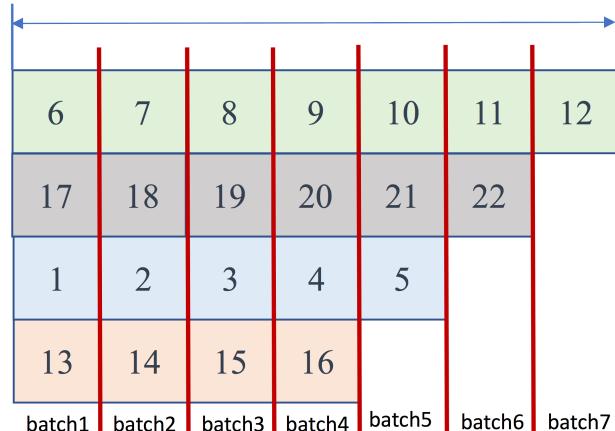
6	7	8	9	10	11	12	
17	18	19	20	21	22		
1	2	3	4	5			
13	14	15	16				

Sorted the **step input by its length.**

运行实例

11	12	13	14	15	16	17	18
24	25	26	27	28	29	30	
1	2	3	4	5	6	7	8
19	20	21	22	23			

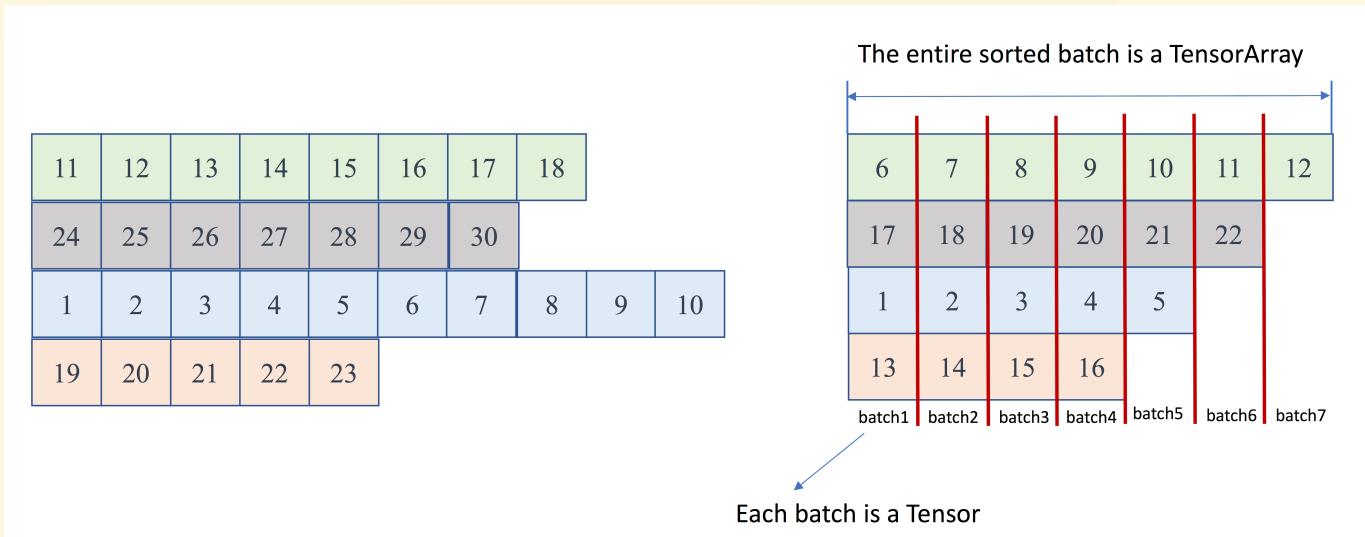
The entire sorted batch is a TensorArray



Each batch is a Tensor

- 执行到第5~7个batch时，batch size将会缩小

运行实例



- 第5 ~ 7个batch时RNN的**memory**会发生什么?
 - **memory** 指向某个operator的输出Tensor，在该operator前向计算之后，“取回”其计算结果
 - 5 ~ 7时，遇到了序列的结束，下一个时间步计算不再需要在已经结束的序列上展开
 - 在**dynamicRNN**中**shrink_memory** operator 用来缩小**memory**的batch输入

运行实例：batch 1 ~ 2

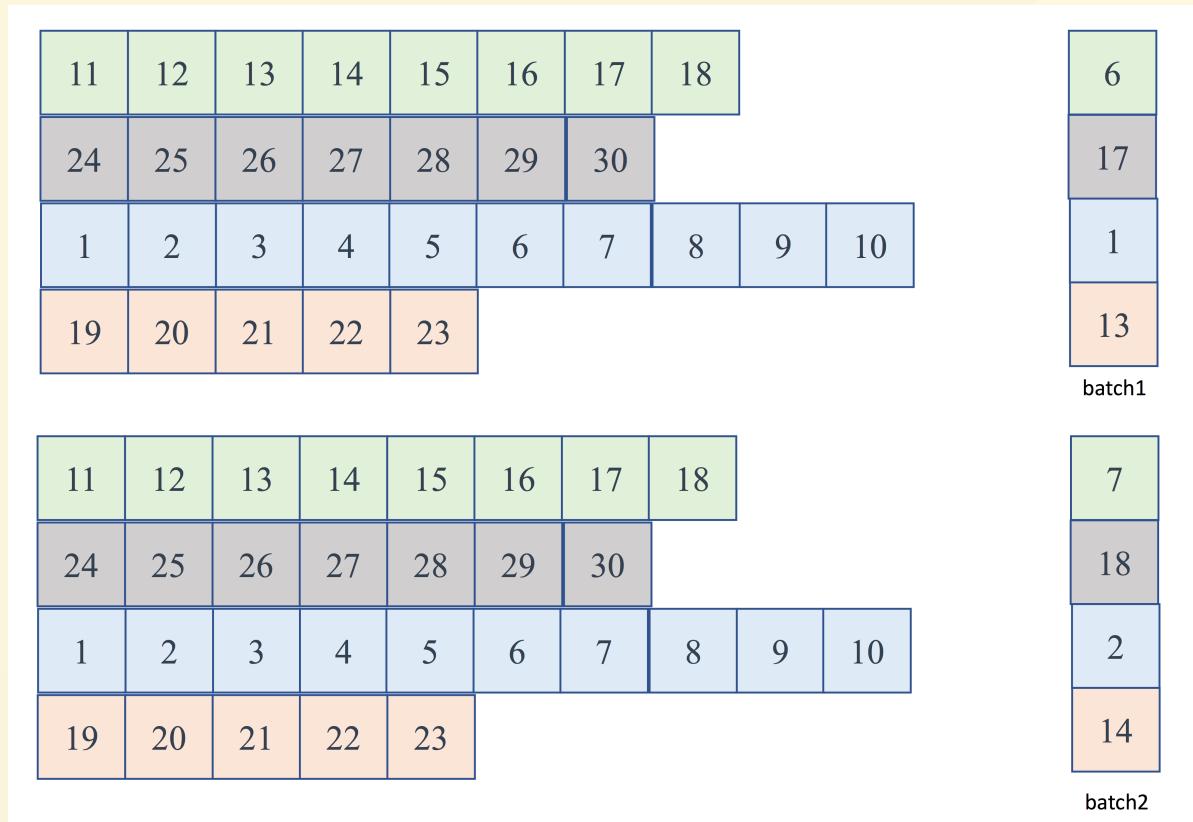


Figure. 第1、2个batch输入dynamicRNN的batch输入

运行实例：batch 3 ~ 4

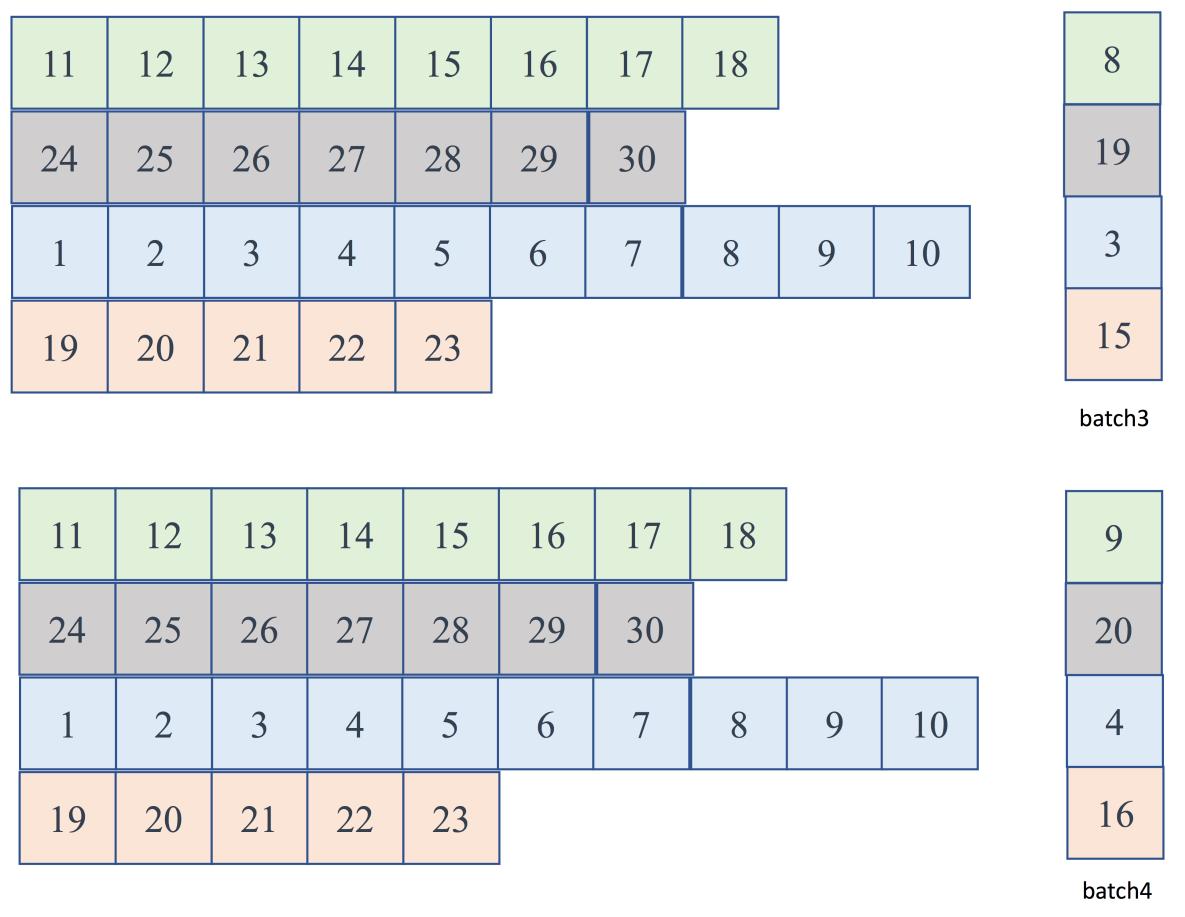


Figure. 第3、4个batch输入dynamicRNN的batch输入

运行实例：batch 5 ~ 7

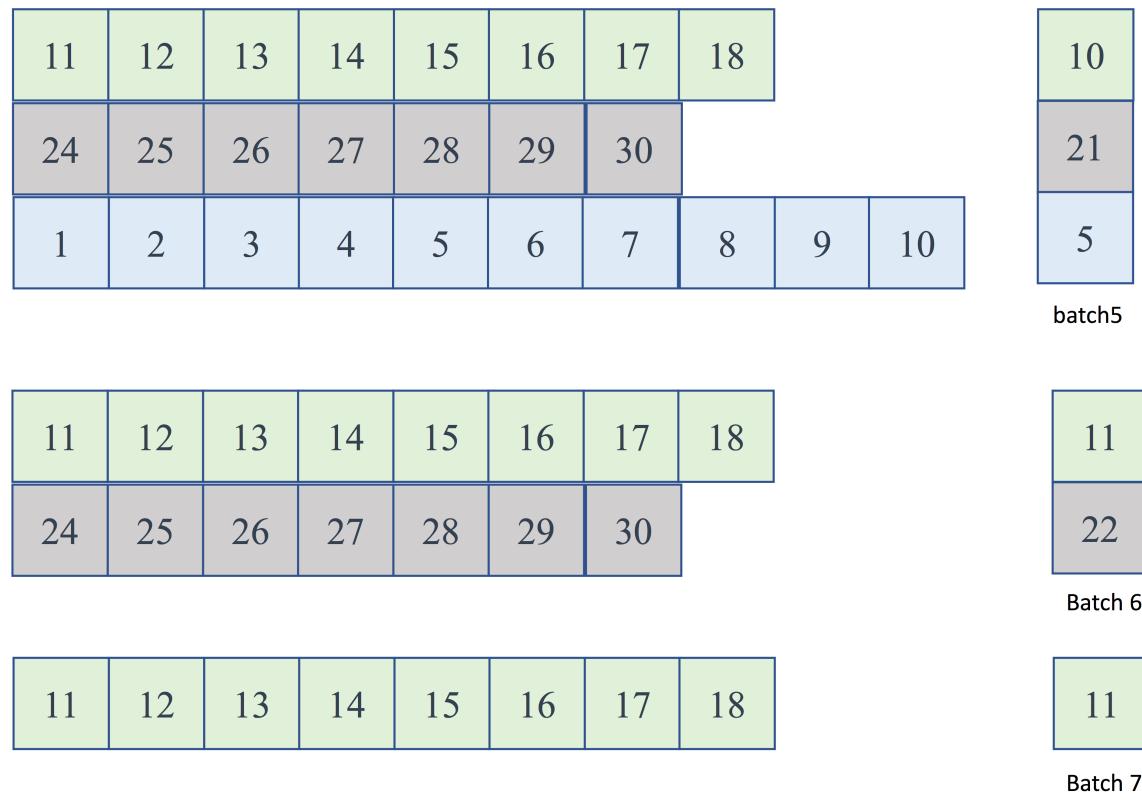
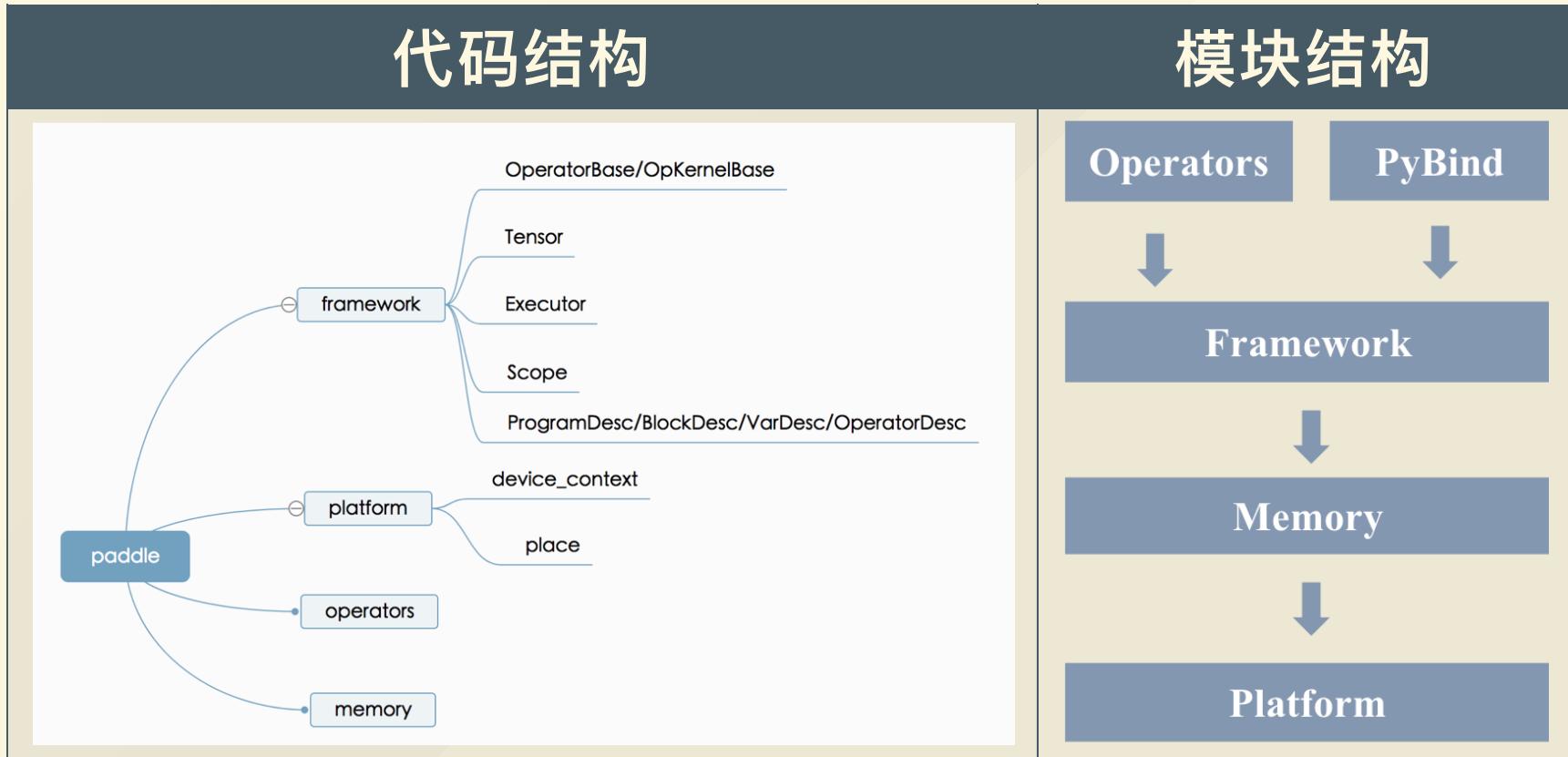


Figure. 第5、6、7个batch输入dynamicRNN的batch输入

7. Fluid 代码结构

Fluid 代码结构



8. 文档总结

- 设计概览
 - 重构概览 [→](#)
 - fluid [→](#)
 - fluid_compiler [→](#)
- 核心概念
 - variable 描述 [→](#)
 - Tensor [→](#)
 - LoDTensor [→](#)
 - TensorArray [→](#)
 - Program [→](#)
 - Block [→](#)
 - Scope [→](#)

- 重要功能模块

- backward [→](#)
- 内存优化 [→](#)
- evaluator [→](#)
- python API [→](#)
- regularization [→](#)

- 开发指南

- 支持新设硬件设备库 [→](#)
- 添加新的Operator [→](#)
- 添加新的Kernel [→](#)

9. 开发指南

建议开发环境：使用 Docker 编译和测试

Docker编译PaddlePaddle源码: [→](#)

PaddlePaddle 在 Dockerhub 地址: [→](#)

1. 获取PaddlePaddle的Docker镜像

```
docker pull paddlepaddle/paddle:latest-dev
```

2. 启动 docker container

```
docker run -it -v $PWD/Paddle:/paddle paddlepaddle/paddle:latest-dev /bin/bash
```

3. 进入docker container后，从源码编译，请参考文档 [→](#)

一些说明

1. PaddlePaddle的Docker镜像为了减小体积， 默认没有安装vim， 可以在容器中执行`apt-get install -y vim`来安装vim。
2. 开发推荐使用tag为`latest-dev`的镜像， 其中打包了所有编译依赖。`latest`及`lastest-gpu`是production镜像， 主要用于运行PaddlePaddle程序。
3. 在Docker中运行GPU程序， 推荐使用nvidia-docker， [否则需要将CUDA库和设备挂载到Docker容器内。](#)

```
nvidia-docker run -it -v $PWD/Paddle:/paddle paddlepaddle/paddle:latest-dev /bin/bash
```

如何贡献

- 提交PullRequest前请务必阅读： 
- 代码要求
 - 1. 代码注释遵守 Doxygen 的样式
 - 2. 确保编译器选项 WITH_STYLE_CHECK 已打开，并且编译能通过代码样式检查
 - 3. 所有代码必须具有单元测试，且能够通过所有单元测试
- 使用 `pre-commit` 钩子提交Pull Request
 - 1. 帮助格式化源代码 (C++, Python)
 - 2. 在提交前自动检查一些基本事宜：如每个文件只有一个 EOL, Git 中不要添加大文件等
 - 3. 安装pre-commit，并在PaddlePaddle根目录运行：

```
→ pip install pre-commit  
→ pre-commit install
```

如何贡献

1. 开始开发之前请先建立issue。

- 让其它同学知道某项工作已经有人在进行，以避免多人开发同一功能的情况。

2. 提交PR必须关联相关的issue。做法请参考： ➔

- 目的：为了在提交的版本中留有记录描述这个PR是为了开发什么样的功能，为了解决什么样的问题。
- 当PR被merge后，关联的issue会被自动关闭。

3. PR review 中，reviewer的每条comment都必须回复。

- 如修改完可直接回复：Done。
- 目的：review comment 中可能会有（1）询问类型的问题；（2）可以在下一个PR修改的问题；（3）comment意见不合理等。需要明确回复，以便reviewer和其他人有历史可查，便于区分是否已经进行修改，或者准备下一个PR修改，或者意见不合理可以不用进行修改。

10. 添加新的 Operator

概念简介

添加一个新的operator，会涉及实现以下C++类的派生类：

1. `framework::OperatorBase`: Operator(简写，Op)基类。
2. `framework::OpKernel`: Op计算函数的基类，称作Kernel。
3. `framework::OperatorWithKernel`: 继承自OperatorBase，Op有计算函数，称作有Kernel。
4. `class OpProtoAndCheckerMaker`：描述该Op的输入、输出、属性、注释,主要用于Python API接口生成

依据是否包含kernel，可以将Op分为两种：

1. 包含Kernel的Op：继承自OperatorWithKernel，绝大多数operator都属于这一类
2. 不包含kernel的Op，继承自OperatorBase，只有少量Op属于这一类，例如`while_op`, `ifelse_op`

这里主要介绍带Kernel的Op如何编写。

添加新的Operator需要修改/添加哪些文件？

内容	定义位置
OpProtoMake 定义	.cc 文件， Backward Op不需要OpProtoMaker
Op定义	.cc 文件
Kernel实现	CPU、CUDA共享Kernel实现在.h文件中，否则，CPU 实现在.cc文件中，CUDA 实现在.cu文件中。
注册Op	Op注册实现在.cc文件；Kernel注册CPU实现在.cc文件中，CUDA实现在.cu文件中

- 添加 Operator 之前请阅读：[Operator 命名规范](#)及[Operator Markdown
注释规范](#)。
- 实现新的op都添加至目录[paddle/operators](#)下，文件命名以`*_op.h`（如有）`、 *_op.cc`、 `*_op.cu`（如有）结尾。
- 根据文件名自动构建op和Python端绑定，请务必遵守以上命名，否则需要进一步修改PyBind相关文件及CMakeLists.txt。

实现带Kernel的Operator step1: 定义ProtoMaker类

下面均以clip op为例进行介绍

- clip_op计算公式: $Out = \min(\max(X, min), max)$
- 首先定义ProtoMaker来描述该Op的输入、输出，并添加注释（下面代码段中的注释进行了简化，实现时需按照规范添加注释）：

```
template <typename AttrType>
class ClipOpMaker : public framework::OpProtoAndCheckerMaker {
public:
    ClipOpMaker(OpProto* proto, OpAttrChecker* op_checker)
        : OpProtoAndCheckerMaker(proto, op_checker) {
        AddInput("X", "(Tensor)The input of clip op.");
        AddOutput("Out", "(Tensor),The output of clip op.");
        AddAttr<AttrType>(
            "min", "(float),Minimum value.");
        AddAttr<AttrType>(
            "max", "(float),Maximum value.");
        AddComment(R"DOC(
        ....
)DOC");
    }
};
```

实现带Kernel的Operator step2: 定义Operator类

下面的代码段实现了 `clip_op` 的定义：

```
class ClipOp : public framework::OperatorWithKernel {
public:
    using framework::OperatorWithKernel::OperatorWithKernel;

    void InferShape(framework::InferShapeContext* ctx) const override {
        PADDLE_ENFORCE(ctx->HasInput("X"),
                       "Input(X) of ClipOp should not be null.");
        PADDLE_ENFORCE(ctx->HasOutput("Out"),
                       "Output(Out) of ClipOp should not be null.");
        auto x_dims = ctx->GetInputDim("X");
        auto max = ctx->Attrs().Get<float>("max");
        auto min = ctx->Attrs().Get<float>("min");
        PADDLE_ENFORCE_LT(min, max, "max should be greater than min.");
        ctx->SetOutputDim("Out", x_dims);
        ctx->ShareLoD("X", /*->*/ "Out");
    }
};
```

Operator 类中需要完成的工作

1. clip_op 继承自 OperatorWithKernel ,

```
using framework::OperatorWithKernel::OperatorWithKernel;
```

表示使用基类 OperatorWithKernel 的构造函数。

2. 重写 InferShape 接口。

- InferShape 为 const 函数，不能修改 Op 的成员变量
- InferShape 的参数为 const framework::InferShapeContext &ctx ，从中可获取到输入输出以及属性
- InferShape 会被调用两次，一次是编译时（创建 op ） , 一次是运行时（调用 op 的 Run 方法时） , 需要完成以下功能：
 1. 做检查，尽早报错：检查输入数据维度、类型等是否合法
 2. 设置输出 Tensor 的形状

通常 OpProtoMaker 和 Op 类的定义写在 .cc 文件中。

补充说明

1. `InferShape` 目前支持两种实现方式，二者最后都会生成一个functor注册给 `OplInfo` 结构体。

1. 继承 `framework::InferShapeBase`, 实现为一个functor (参考 [mul_op](#))

2. override `InferShape` 函数 (参考 [clip_op](#))

2. 什么是 `functor` ?

◦ 类或结构体仅重载了 `()`，一般是可被多个kernel复用的计算函数。

```
template <typename T>
class CrossEntropyFunctor<platform::CPUDeviceContext, T> {
public:
    void operator()(const platform::CPUDeviceContext& ctx,
                     framework::Tensor* out,
                     const framework::Tensor* prob,
                     const framework::Tensor* labels, const bool softLabel) {
        ....
    }
};
```

◦ 在 `clip_op` 内也会看到将一段计算函数抽象为functor的使用法： ➔。

实现带Kernel的Operator step3: 定义OpKernel类

- ClipKernel 继承自 framework::OpKernel， 带有下面两个模板参数：
 1. typename DeviceContext: 表示设备类型，不同设备共享同一个Kernel时，需添加该模板参数。不共享时，需要提供针对不同设备的特化实现。
 2. typename T: 表示支持的数据类型，如 float, double 等
- 在 ClipKernel 类中重写 Compute 方法
 1. Compute 接受输入参数： const framework::ExecutionContext& context
 - ExecutionContext 是从 Scope 中将运行时 Op 的输入、输出 Variable 组织在一起，使得 Op 在调用 Compute 方法时，能够简单地通过名字拿到需要的输入输出 Variable
 - 与 InferShapeContext 相比， ExecutionContext 中增加了设备类型
 2. 在 Compute 函数里实现 OpKernel 的具体计算逻辑

ClipKernel 代码概览

```
template <typename DeviceContext, typename T>
class ClipKernel : public framework::OpKernel<T> {
public:
    void Compute(const framework::ExecutionContext& context) const override {
        auto max = context.Attr<T>("max");
        auto min = context.Attr<T>("min");
        auto* x = context.Input<Tensor>("X");
        auto* out = context.Output<Tensor>("Out");
        T* out_data = out->mutable_data<T>(context.GetPlace());
        const T* x_data = x->data<T>();
        int64_t numel = x->numel();
        Transform<DeviceContext> trans;
        trans(context.template device_context<DeviceContext>(),
              x_data,
              x_data + numel, out_data, ClipFunctor<T>(min, max));
    }
};
```

- 为了使 `OpKernel` 的计算过程书写更加简单，并且 CPU、CUDA 的代码可以复用，Fluid 使用 Eigen 作为基础的矩阵运算库
- Fluid 对 Eigen unsupported Tensor 提供了一些基本的封装，可以在 `Compute` 接口中直接调用
 - 关于在 PaddlePaddle 中如何使用 Eigen 库，请参考 [使用文档](#)。

实现带Kernel的Operator step4: 实现反向Op

- 反向Op没有ProtoMaker，除此之外定义与实现方式前向Op完全一致，不再赘述
- 这里仅对反向Op的输入输出进行说明：
 1. 反向Op的输入
 - 前向Op的输出
 - 反向传播过程中传递给当前Op的梯度
 - 需要注意，Fluid中，不区分Cost Op和中间层Op，所有Op都必须正确处理接收到的梯度
 2. 反向Op的输出
 - 对可学习参数的求导结果
 - 对所有输入的求导结果

实现带Kernel的Operator step5: 注册Op及Kernel

至此Op和Op kernel都已经实现完毕，接下来，需要在 `.cc` 和 `.cu` 文件中注册op和kernel

1. 在 `.cc` 文件中注册前向、反向Op类，注册CPU Kernel。

```
namespace ops = paddle::operators;
REGISTER_OP(clip, ops::ClipOp, ops::ClipOpMaker<float>, clip_grad,
            ops::ClipOpGrad);
REGISTER_OP_CPU_KERNEL(
    clip, ops::ClipKernel<paddle::platform::CPUDeviceContext, float>);
REGISTER_OP_CPU_KERNEL(
    clip_grad, ops::ClipGradKernel<paddle::platform::CPUDeviceContext, float>);
```

- 在上面的代码片段中：

1. `REGISTER_OP` : 注册 `ops::ClipOp` 类，类型名为 `clip`，该类的 `ProtoMaker` 为 `ops::ClipOpMaker`，注册 `ops::ClipOpGrad`，类型名为 `clip_grad`
2. `REGISTER_OP_WITHOUT_GRADIENT` : 用于注册没有反向的Op，例如：优化算法相关的Op
3. `REGISTER_OP_CPU_KERNEL` : 注册 `ops::ClipKernel` 类，并特化模板参数为 `paddle::platform::CPUPlace` 和 `float` 类型，同理，注册 `ops::ClipGradKernel` 类

2. 按照同样方法，在 `.cu` 文件中注册GPU Kernel

- 如果CUDA Kernel的实现基于Eigen，需在 `.cu` 的开始加上宏定义

```
#define EIGEN_USE_GPU
```

编译和Python端绑定

- 运行下面命令可以仅编译新添加的Op:

```
make mul_op
```

- 需注意，运行单元测试需要编译整个工程
- 如果遵循前文的文件命名规则，构建过程中，会自动为新增的op添加Python端绑定，并链接到生成的lib库中

实现带Kernel的Operator step6: 添加前向单测及梯度检测

- 新增Op的单元测试统一添加至：[python/paddle/v2/fluid/tests](#) 目录
- 前向Operator单测
 1. Op单元测试继承自 `OpTest`，各项具体的单元测试在 `TestClipOp` 里完成，所有单测case都以 `TestXX` 命名
 2. 单元测试Operator，需要：
 1. 在 `setUp` 函数定义输入、输出，以及相关的属性参数
 2. 生成随机的输入数据
 3. 在Python脚本中实现与前向operator相同的计算逻辑，得到输出值，与operator前向计算的输出进行对比
 4. 反向梯度检测流程测试框架已经实现，直接调用相应接口 `check_grad` 即可
- `clip_op` 单测代码请参考 ➡，这里不再展开

编译执行单测

- `python/paddle/v2/framework/tests` 目录下新增的 `test_*.py` 单元测试会被自动加入工程进行编译
 - 运行单元测试时需要编译整个工程，并且编译时需要打开 `WITH_TESTING`, 即 `cmake paddle_dir -DWITH_TESTING=ON`
- 编译成功后，执行下面的命令来运行单元测试：

```
make test ARGS="-R test_mul_op -V"
```

或者：

```
ctest -R test_mul_op
```

添加Op的一些注意事项

- 为每个Op创建单独的 `*_op.h` (如有) 、 `*_op.cc` 和 `*_op.cu` (如有) 。不允许一个文件中包含多个Op，将会导致编译出错。
- 注册Op时的类型名，需要和该Op的名字一样。不允许在 `A_op.cc` 里面，注册 `REGISTER_OP(B, ...)`，会导致单元测试出错。
- 如果Op没有实现CUDA Kernel，不要创建空的 `*_op.cu`，会导致单元测试出错。
- 如果多个Op依赖一些共用的函数，可以创建非 `*_op.*` 格式的文件来存放，如 `gather.h` 文件。

10. 使用相关问题

定义前向计算

- 当在python端执行时：

```
import paddle.v2.fluid as fluid
```

`framework.py` 定义了两个全局 `Program`：

```
# program is a global instance.  
_main_program_ = Program()  
_startup_program_ = Program()
```

- 前向定义的过程就是不断往 `mian_program` 中添加Op和Variable
- 如果需要执行一个新的 `mian_program` 时，可以调用调用：

```
def switch_main_program(program):  
    """  
    Switch the main program to a new program.  
    This function returns the previous main program.  
    """  
  
    ....
```

自定义参数的初始化

- 调用 `fluid.ParamAttr(.....)` 接口，自定义参数的初始化

```
w_param_attrs = ParamAttr(name=None,  
    initializer=UniformInitializer(low=-1.0, high=1.0, seed=0),  
    learning_rate=1.0,  
    regularizer=L1Decay(1.0),  
    trainable=True,  
    clip=GradientClipByValue(-1.0, 1.0),  
)  
y_predict = fluid.layers.fc(input=x, size=1, param_attr=w_param_attrs)
```

- 补充问题：如何创建 `Variable`

```
cur_program = Program()  
cur_block = cur_program.current_block()  
new_var = cur_block.create_var(name="X", shape=[-1, 16, 16], dtype="float32")
```

添加反向Op

- 调用 `fluid.backward.append_backward(X)` (`X`是一个Variable) , 来为一段前向 `ProgramDesc` 添加反Op

```
data = fluid.layers.data(name="data", shape=(2,3,4))
out = fluid.layers.fc(input=data, size=128, act=None)
loss = fluid.layers.reduce_sum(out)
fluid.backward.append_backward(loss=loss)
```

- 添加优化相关的Op

```
sgd_optimizer = fluid.optimizer.SGD(learning_rate=0.001)
sgd_optimizer.minimize(loss)
```

- 可以随时调用 `print(fluid.default_main_program())` 来输出当前的 `main_program`
- 当构建完成整个 `Program` 后, 调用下面的接口执行内存优化:

```
fluid.memory_optimize(fluid.default_main_program())
```

- 注: 内存优化目前仍在持续开发中, 有可能不够稳定。

总结：编译时执行流程

- 用户定义前向计算
- 添加反向Op到 `default_main_program`
- 添加 gradient clipping Op 到
- 添加 regularization Op 到 `default_main_program`
- 为指定的优化算法，添加相关的状态 variable of optimizer 到 `default_startup_program`
 - 状态相关 variable 是指如学习率, 历史 momentum, 二阶momentum等
- 添加初始化 variable 的Op 到 `default_startup_program`
- 为整个网络最后一个op，添加设置其接受到的梯度的Op到 `default_main_program`
- 进行内存优化规划

Feed 数据(一): 通过 feed 字典

- 执行executor的run方法时，指定feed字典，feed op 会将指定的数据放到x和y两个Variable中

```
y_data = np.random.randint(0, 8, [1]).astype("int32")
y_tensor = core.Tensor()
y_tensor.set(y_data, place)

x_data = np.random.uniform(0.1, 1, [11, 8]).astype("float32")
x_tensor = core.Tensor()
x_tensor.set(x_data, place)
.....
cost = exe.run(
    fluid.default_main_program(),
    feed={'x': x_tensor,
          'y': y_tensor},
    fetchlist=[avg_cost])
```

- 这种方法较为底层，一般用于单测中

Feed 数据(二): 使用 DataFeeder接口

- 编写一个data_reader函数， data_reader是一个Python generator

```
def demo_reader():
    def random_generator():
        yield np.random.uniform(0.1, 1, [4]), np.random.randint(0, 1, [1])
    return random_generator
```

- 在训练任务中使用 DataFeeder 接口

```
cost = exe.run(
    fluid.default_main_program(),
    feed={'x': x_tensor,
          'y': y_tensor},
    fetchlist=[avg_cost])

train_reader = paddle.batch(
    paddle.reader.shuffle(demo_reader(), buf_size=500), batch_size=4)
feeder = fluid.DataFeeder(place=place, feed_list=[x, y])
for data in train_reader():
    cost = exe.run(
        fluid.default_main_program(),
        feed=feeder.feed(data),
        fetch_list=[cost])
```

常见问题

- 如何使用 evaluator ? 

```
accuracy = fluid.evaluator.Accuracy(input=predict, label=label)
for pass_id in range(PASS_NUM):
    accuracy.reset()
    for data in train_reader():
        loss, acc = exe.run(fluid.default_main_program(),
                            feed=feeder.feed(data),
                            fetch_list=[avg_cost] + accuracy.metrics)
    pass_acc = accuracy.eval(exe)
    # acc 当前一个batch 的 accuracy
    # pass_acc 当前batch 的 accuracy
pass_total_acc = accuracy.eval(exe) # 整个pass的accuracy
```

- 如何在训练中测试? 
- 如何保存训练好的模型? 
- 如何加载训练好的模型进行预测? 
- 如何在同一个训练任务中定义多个Program, 并交替运行? 
- 如何profile? Fluid 实现了profile 工具, 可以直接调用。请参考示例 

谢谢 ❤