

# 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		Caffe, Theano, Torch, PaddlePaddles	
2016			PyTorch, TensorFlow Eager Execution, <b>PaddlePaddle Fluid</b>

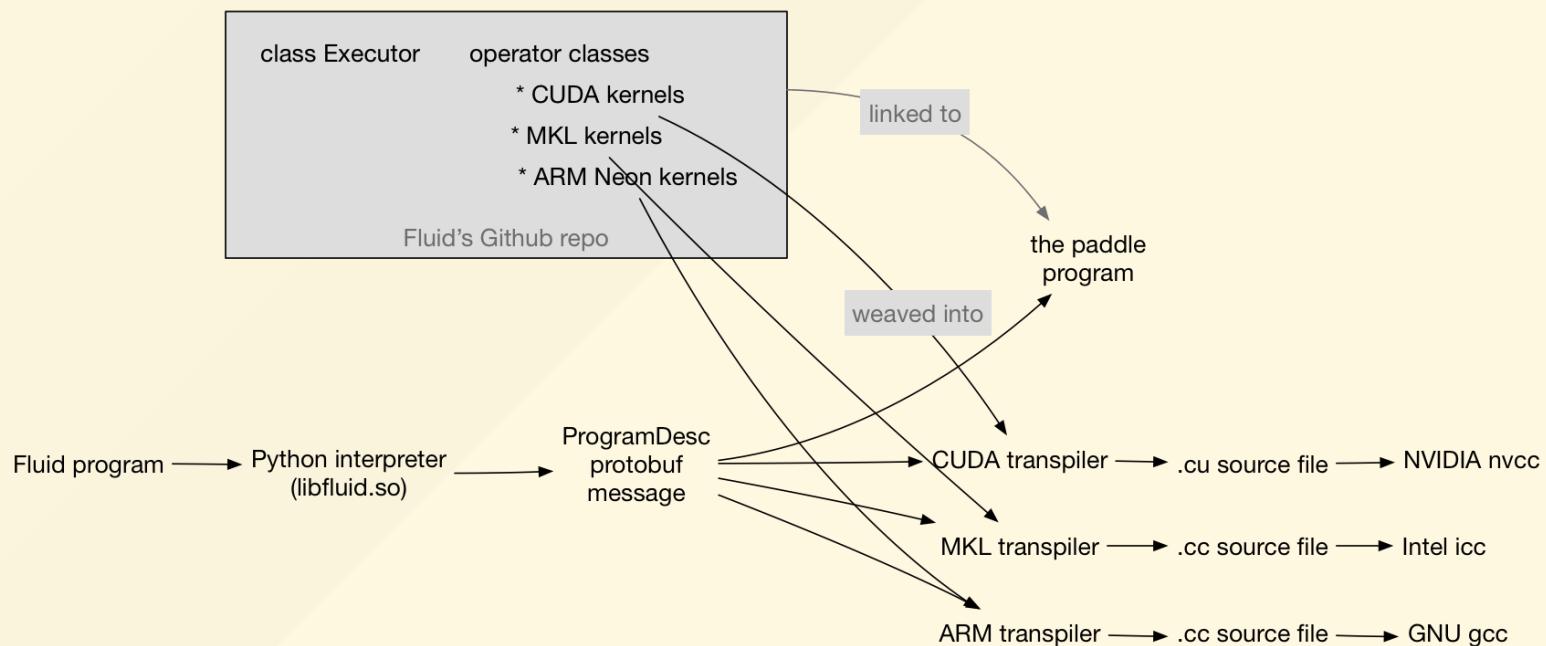
# 目标 😊

- 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

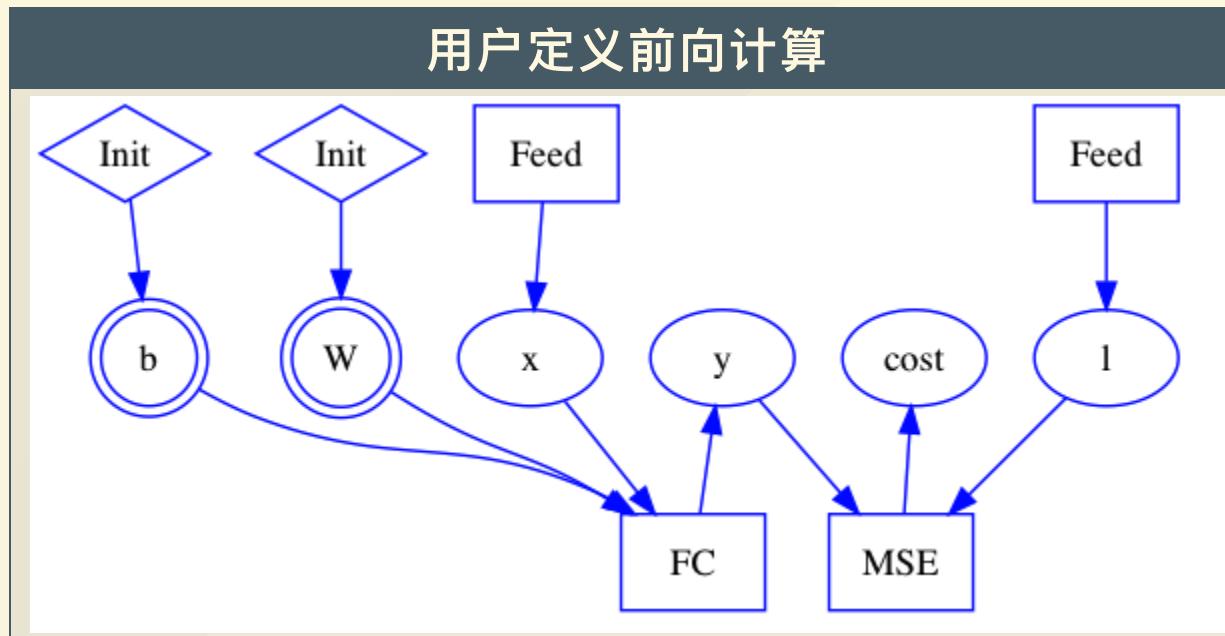


在Fluid程序实例中，区分编译时和运行时

# 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)
```

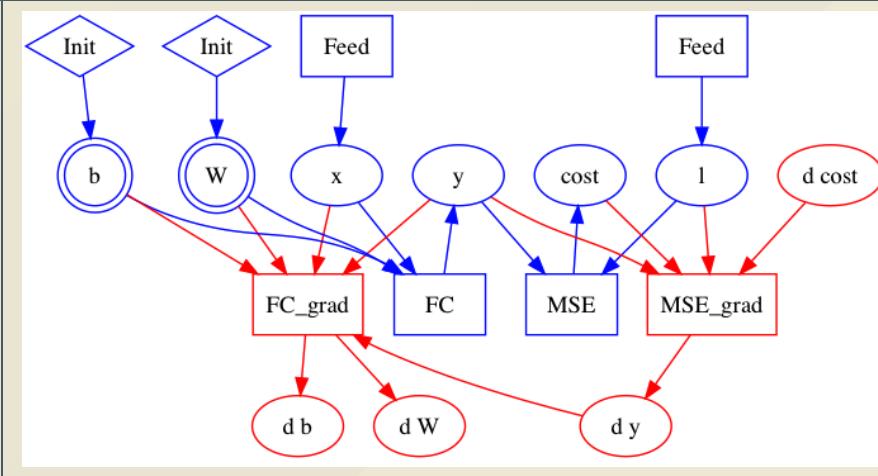


# Fluid 编译时

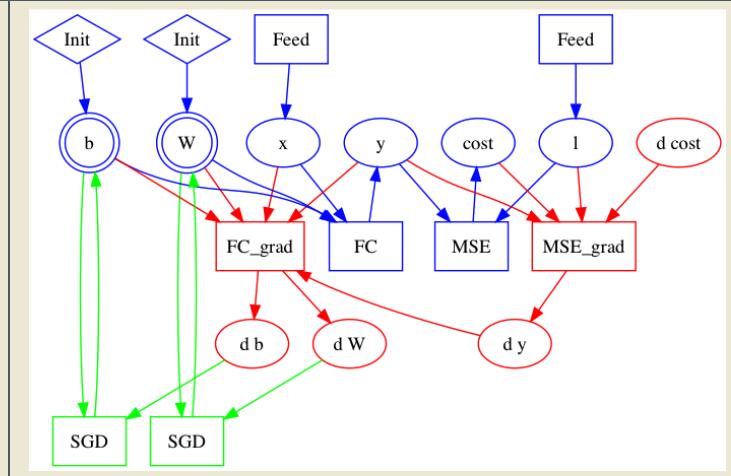
- 添加反向、正则、优化

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

添加反向operator



添加优化operator



# Fluid 运行时（一）

- 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 startup\_program

```
exe = fluid.Executor(place)  
exe.run(fluid.default_startup_program())
```

# Fluid 运行时 (二)

- 执行训练程序

```
PASS_NUM = 100
for pass_id in range(PASS_NUM):
    fluid.io.save.persistables(exe, "./fit_a_line.model/")
    fluid.io.load.persistables(exe, "./fit_a_line.model/")
    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)
```

# 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

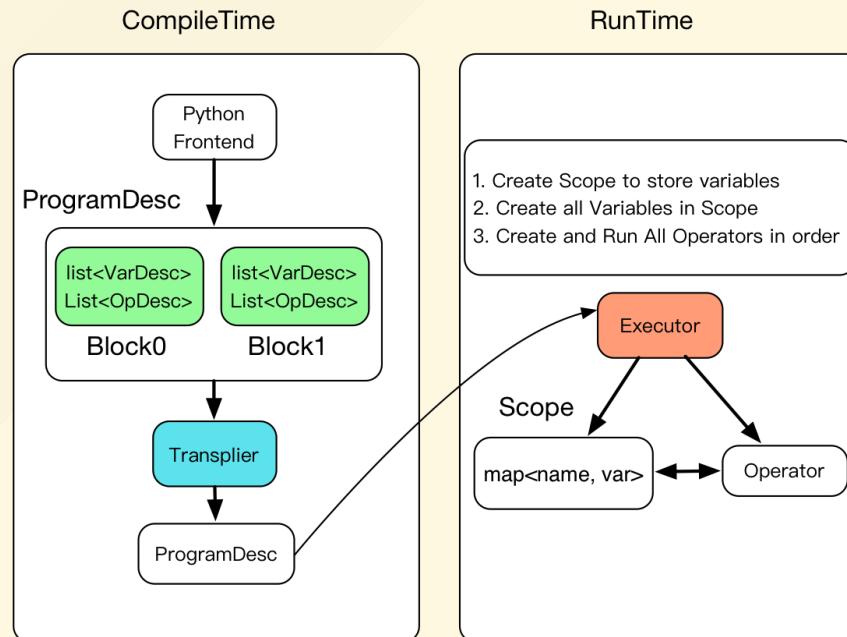


Figure. 编译时运行时概览

### 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

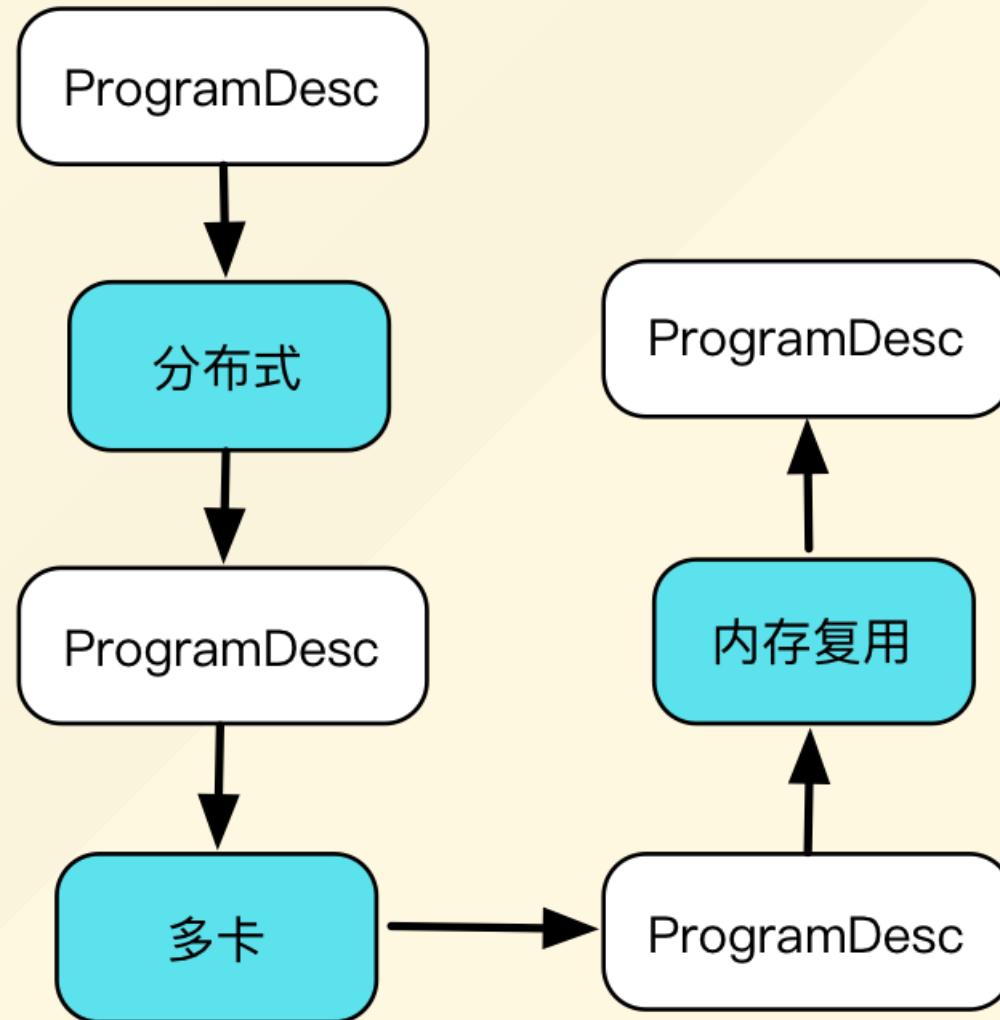
1. 用户层提供的描述语法具有完备性、自治性，有能力支持对复杂计算过程描述
2. 使用方式和核心概念可以类比编程语言，认知能够直接迁移
3. 能够支持：定义问题，逐步求解

### 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 ”

# Transplier



# 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

# 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`ProgramDesc`

```
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	<a href="#"><u>VarDesc</u></a>	<a href="#"><u>Variable</u></a>
Operation	<a href="#"><u>OpDesc</u></a>	<a href="#"><u>Operator</u></a>
Block	BlockDesc	Block

- 执行相关 : `Executor`

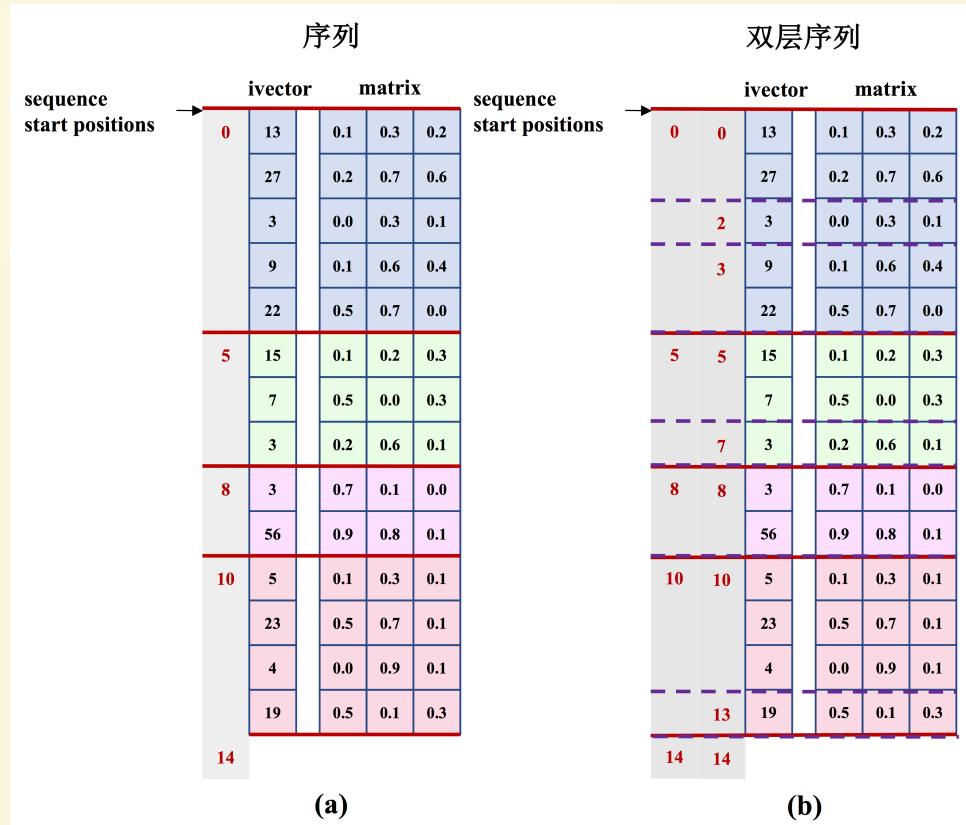
# Tensor 和 LoD(Level-of-Detail) Tensor

- Fluid中输入输出数据统一使用Tensor (n-dimension array) 表示
- 一个mini-batch数据是一个Tensor
- Fluid中RNN 处理变长序列时无需padding，得益于 LoDTensor 表示

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

- LoDTensor用于表示序列，是在Tensor的基础上附加了序列在一个batch 中的起始偏移
- 可以简单地将 LoD 存储结构理解为一个： `std::vector<std::vector<int>>`

# LoD 信息



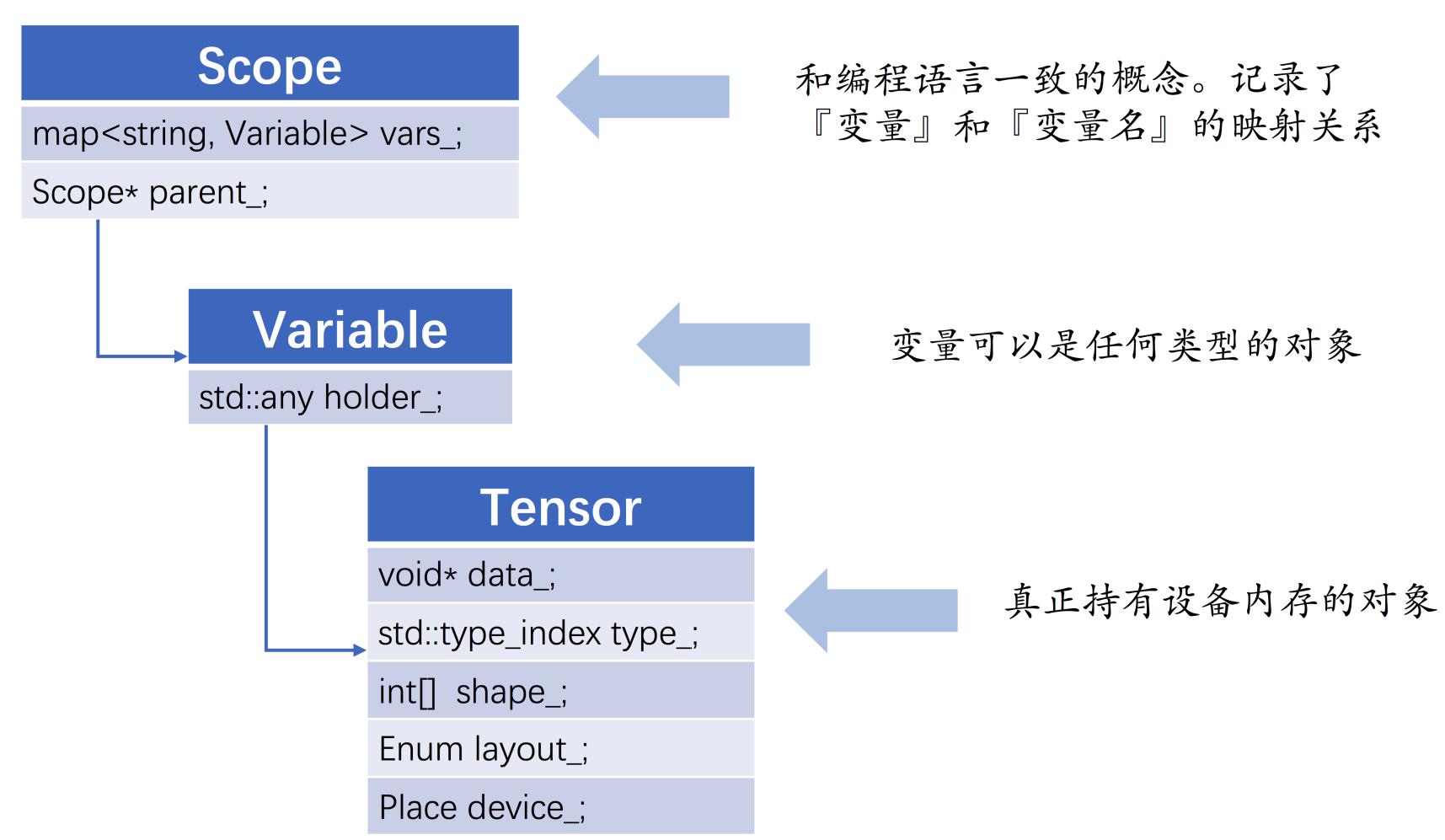
- 图(a)的LoD 信息

```
[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*/]
```

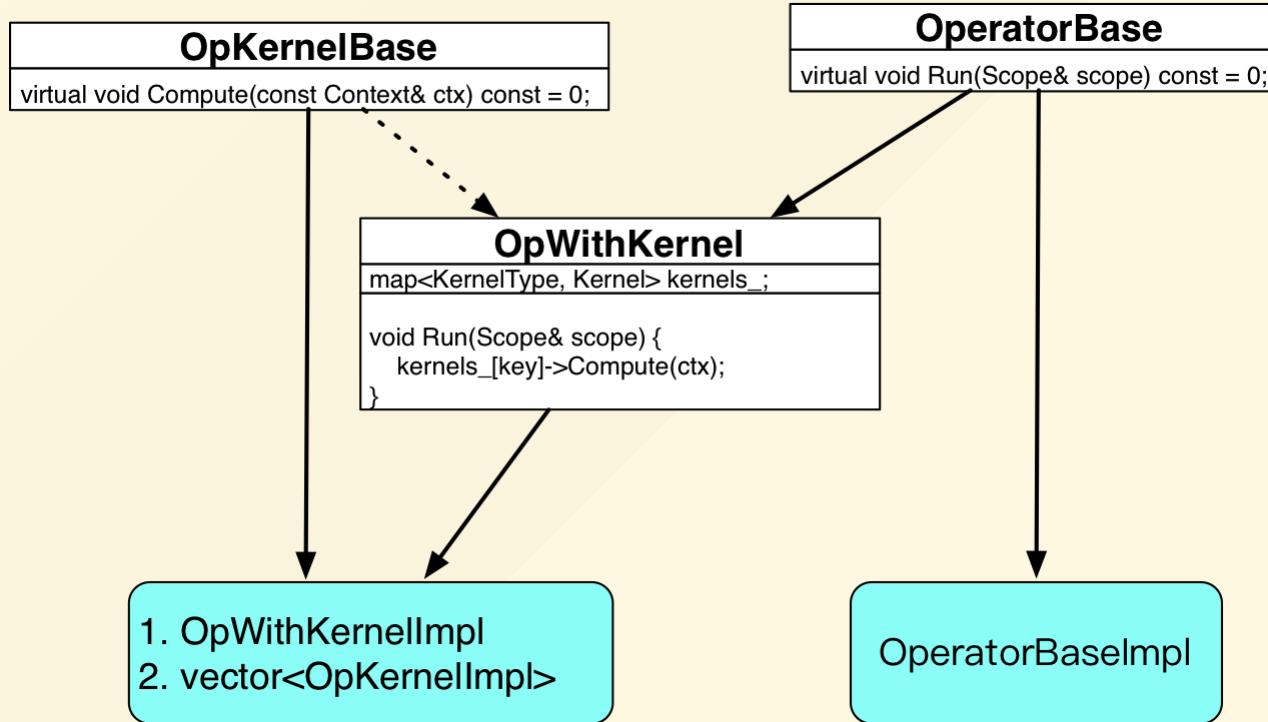
# Tensor vs. Variable vs. Scope



# Executor

接口	说明
<pre>class Executor { public:     void Run(const ProgramDesc&amp; pdesc,              Scope* scope,              int block_id) {         auto&amp; block = pdesc.Block(block_id);          // create all variables         for (auto&amp; var : block.AllVars()) {             scope-&gt;Var(var-&gt;Name());         }          // create op and run it one by one         for (auto&amp; op_desc : block.AllOps()) {             auto op = CreateOp(*op_desc);             op-&gt;Run(*local_scope, place_);         }     } };</pre>	<p>输入</p> <ol style="list-style-type: none"><li>1. ProgramDesc</li><li>2. Scope</li><li>3. block_id</li></ol> <p>解释执行步骤</p> <ol style="list-style-type: none"><li>1. 创建所有 Variables</li><li>2. 逐一创建 Operator 并运行</li></ol>

# Operator/OpWithKernel/Kernel



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

## 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，需要为每一种支持的Place，提供以上三个接口的特化实现

# 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  $2^k$  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;
}

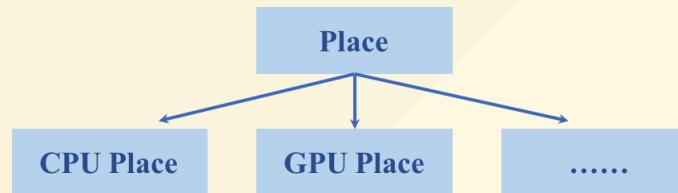
template <>
void Free<platform::CPUPlace>(platform::CPUPlace place, void* p) {
    VLOG(10) << "Free pointer=" << p << " on " << platform::Place(place);
    GetCPUBuddyAllocator()->Free(p);
}

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, 用与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
- step 3: 运行时的 KernelType 推断和Kernel切换，按需要添加Kernel推断和Kernel切换规则
  - CPUPlace ➡ GPUPlace : 跨设备内存复制
  - FP32 ➡ FP16 : 精度转换
  - NCHW ➡ nChw8c : Layout转换

# 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

# 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		



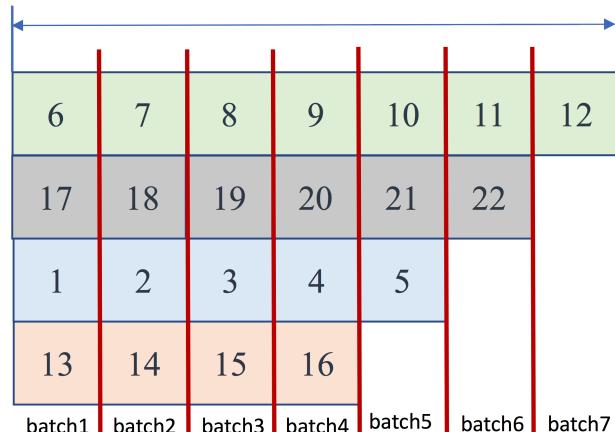
6	7	8	9	10	11	12	
17	18	19	20	21	22		
1	2	3	4	5			
13	14	15	16				

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			

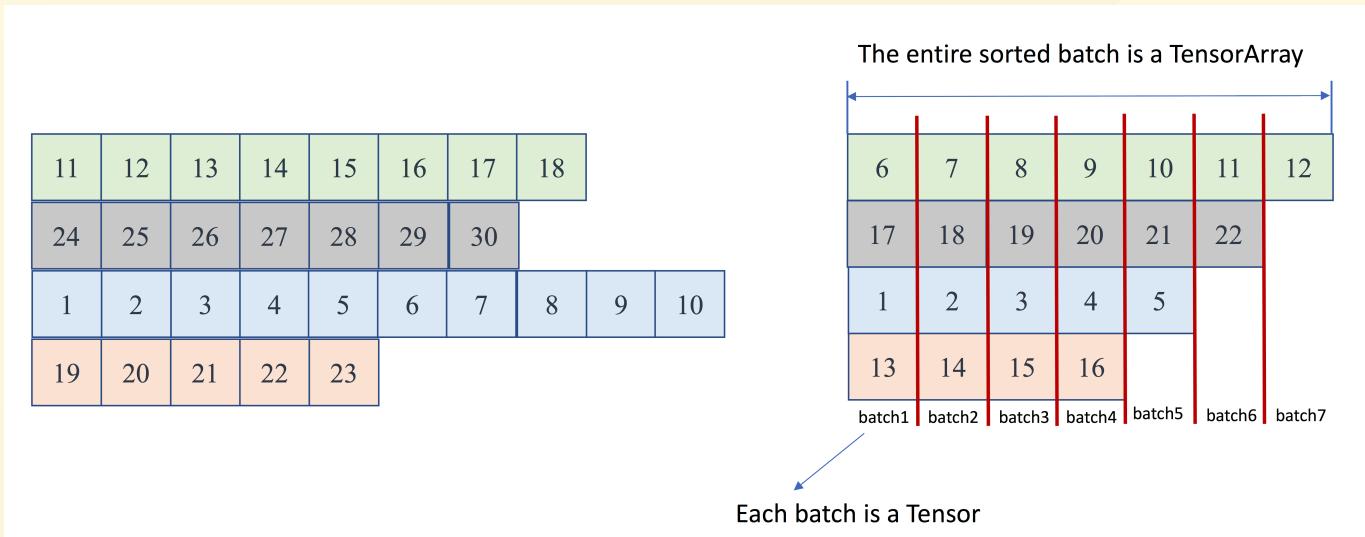
The entire sorted batch is a TensorArray



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

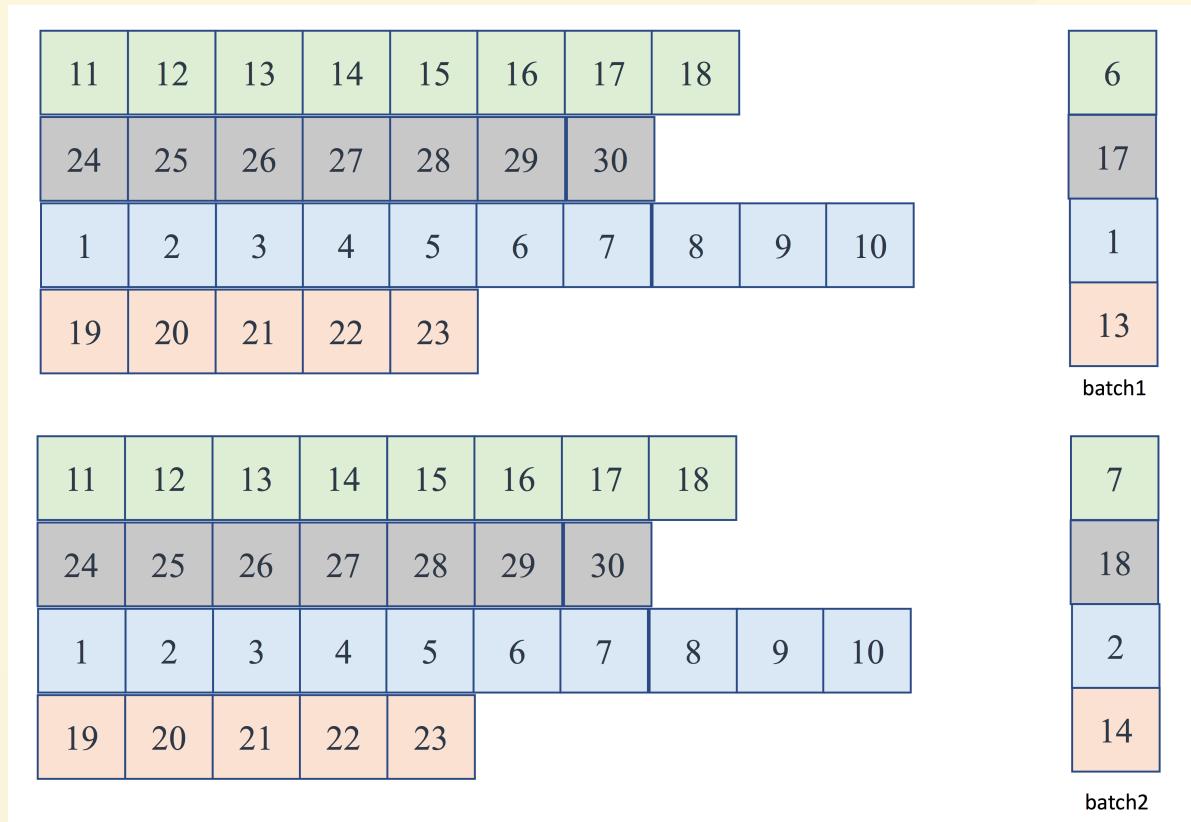


Figure. 第1、2个batch输入dynamicRNN的batch输入

# 运行实例：batch 3 ~ 4

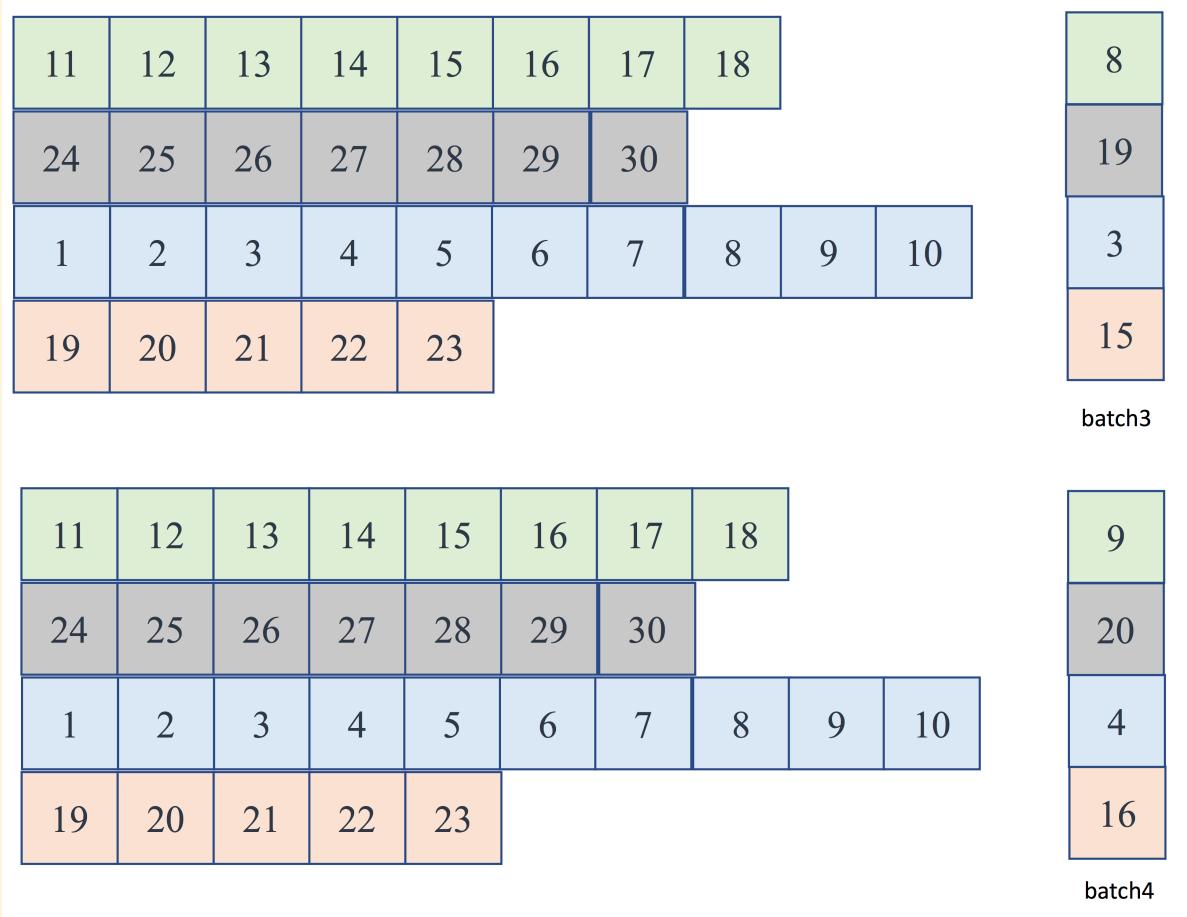


Figure. 第3、4个batch输入dynamicRNN的batch输入

# 运行实例：batch 5 ~ 7

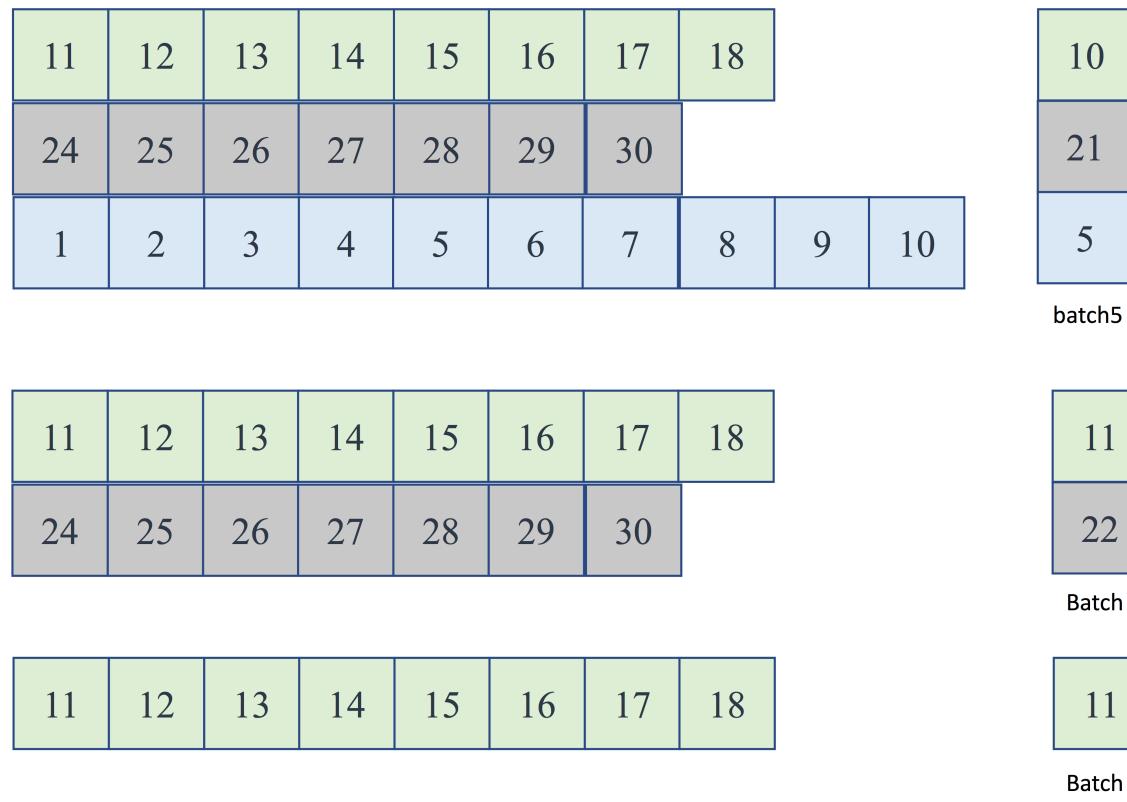
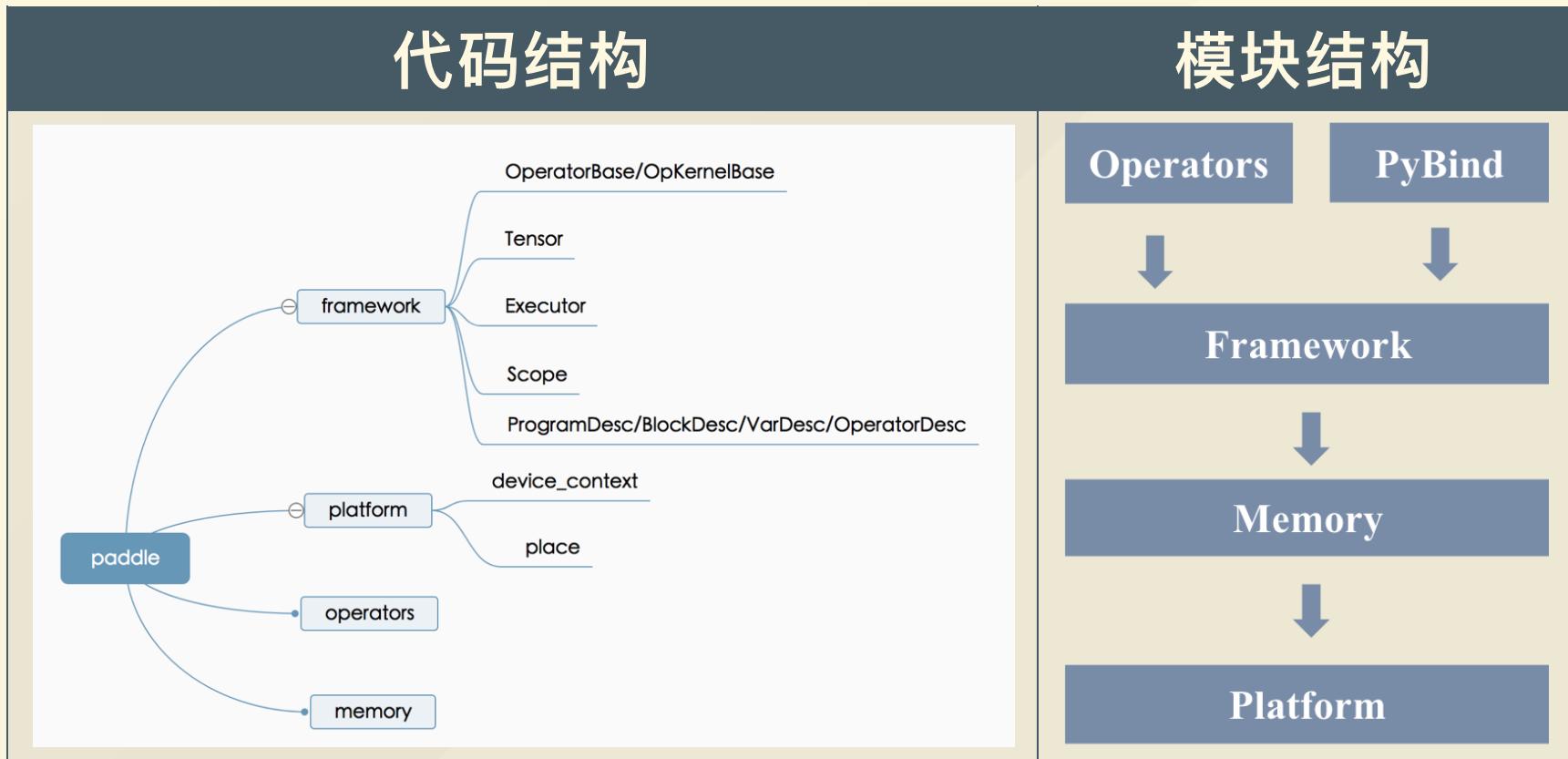


Figure. 第5、6、7个batch输入dynamicRNN的batch输入

## 7. Fluid 代码结构

# Fluid 代码结构



## 8. 文档

- 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 [→](#)

谢谢 ❤