**PaddlePaddle学习笔记**

1. 背景

本文档记录了本人在入职学习PaddlePaddle代码过程中，学习到的一些知识点。这些知识点主要来自三方面：

1. 同事口头讲解；
2. 部门内现有文档；
3. 自己通过阅读代码、思考总结。

PaddlePaddle的代码是复杂的，入门周期较长。目前项目已经过多轮迭代，但在代码中无法Get到『历史渊源』。而从代码反映射『历史渊源』，进而看懂代码逻辑是很吃力的。因此，这个笔记，主要用文字表述一些『渊源』、『背景』、『目的』，总结一些『程序执行流程』。希望对大家有帮助。**里面有些逻辑也很难使用文字语言表达清楚，所以有些地方表述欠佳。很多时候本文档，通过文字总结程序的执行过程，类似于代码注释，但必须结合代码阅读才行，否则读不懂。**

**此外，目前（编写文档时）本人对Paddle的理解还有限。里面可能有一些猜测、结论是错误的，有可能误导阅读者。**

本文假设阅读的同学对深度学习有简单的了解。如果需要补充深度学习知识，可以看看百度AI Studio中的教程：

<https://aistudio.baidu.com/aistudio/course>

还有一个特别好的视频链接如下：

https://space.bilibili.com/88461692/channel/detail?cid=26587

1. 我对PaddlePaddle的理解

Paddle设计之初，希望使用Paddle编写深度学习模型的用户，就像正常编写程序一样『表述』模型。因此，有别于其它深度学习框架，创新的提出了Program的概念。而业界深度模型更多采用Graph的概念。Program最大特点是支持了类似与If、While等条件控制类语法，让Paddle对模型的表达能力更加强大。*（以上观点非官方，纯属个人见解）*

既然，支持了条件控制类语法，Paddle就引入了Block的概念，类似于C++中的{}，代表程序块、作用域等。每个Block中有大量的Operator、Variable，即算子和数据。这些算子和数据组成了一个复杂的深度学习网络模型。

范范的来说，Paddle框架的主要任务有：

1. 提供友好的API接口，让用户『表达』、『开发』模型更容易。Paddle将其转化为Program+Block+Operator+Variable的程序表达；
2. 对『Program+Block+Operator+Variable的程序表达』进行必要、且尽可能充分的优化，以使得模型训练高效；
3. 业界有大量成熟的网络模型（fc、conv2d等），Paddle应对这些模型进行封装，用户拿去用就可以了。
4. fc、conv2d等，可以理解为由无数个Operator组成的，一个Operator完成一个小的数学运算。因此Paddle需要开发大量的Operator，并有效的管理Operator。同时保证很好的Operator扩展性。
5. 用户定义好模型后，Paddle应负责模型的训练。Paddle应尽可能多的支持各种硬件，如CPU、GPU、ARM、FPGA等；
6. 在训练过程中，为了提升性能、加快训练，Paddle需要考虑程序优化、内存优化，也需要考虑并发、多卡、分布式；
7. 训练完得模型，应该可存储、可加载，可预测。

因此，路漫漫其修远兮。。。

1. 静态图+Executor

静态图+Executor是Paddle最早支持的方案。

从用户视角，其大致使用流程为：

1. 用户调用Python（Paddle API）代码定义网络模型。此时，Paddle将网络模型转义成（ProgramDesc、BlockDesc、OPDesc、VarDesc）的计算机的存储结构，这种存储结构的信息是充分持有用户所表达的信息的。
2. 用户提前准备好需要训练的数据。
3. 用户将定义好的Program和准备好的数据，交给Executor去Run。
   1. 基础数据结构、概念
      1. Variable和Operator

深度学习网络模型，本质上都是在进行数据运算，无论多复杂的数学公式，多大的模型，最终都可以拆解成一个一个的小的计算单元和数据存储单元。Operator就是最基础的计算单元，Variable就是最基础的数据存储单元。

因此Variable和Operator是一个网络模型的最小组成元素，任何模型均由无数个Variable和Operator组成。

Variable和Operator是Paddle底层的核心最核心的概念。几乎所有逻辑都是围绕这两个概念展开。

* + 1. Program和Block

Program和Block是Paddle提出的，对OP（Operator）和Var（Variable）的管理方法、概念。

Program可以理解为一段程序，他描述了一个模型的整体长相。

而程序中可能会有if、while等程序块，每个『块』，即一个Block。

因此一个Program中可能有1个或多个Block，其中首个Block存储的是Program中的OP、Var集合，剩余Block分别存储相应的子程序块中的OP、Var集合。

Program可以交付给Executor执行，Executor首先执行Block[0]中的OP，过程中根据情况穿插执行其余Block。

* + 1. Proto和Desc

在Paddle中Program、Block、Variable、Operator，每个都有多种表达。如proto::BlockDesc,BlockDesc、Block等。不都是Block吗，为啥还分出来好几种呢？

Protocol Buffer是，是Google开发的一种数据描述语言，在数据结构序列化、数据存储、数据通信方面功能强大。Paddle使用Proto定义Program、Block、Var、OP等数据结构，有利于数据的存储、共享等。Proto数据结构定义在.proto文件中，Google提供了protoc编译器，通过该编译器，可将.proto『编译』成C++、Python等代码。Paddle代码中『proto::xxxDesc』字样得代码即Protocol Buffer『编译』产生的。

为了更好的在框架中使用proto::xxxDesc，我们对proto::xxxDesc进行了一层包装，即我们的C++表示——xxxDesc，如VarDesc，OpDesc。

* + 1. Python中的Program、Block、Variable、Operator

Python端主要负责Program的构建，我们叫『编译时』。为了Python端构建Program时，方便使用ProgramDesc、BlockDesc、VarDesc、OpDesc。框架在Python端定义了与之对应的Program、Block、Variable、Operator四个Python类，每个类均有一个成员变量self.desc，self.desc持有C++的xxxDesc对象。这样，Python随时可以调用C++中的方法。

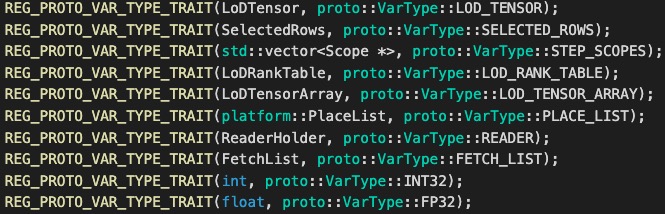
* + 1. Parameter

模型中的可学习的参数，我们称为Parameter。Parameter是一种特殊的Variable。我们训练的目的就是找到最合理的Parameter集合，能够让网络得出最好的预测结果。

* + 1. C++中的Variable、OperatorBase

以上所有的Program、Block、Variable、Operator，都属于对Program的『描述』，但都不是可执行的计算机表达。所有的『描述』都是为了能够快速构建、和优化Program。当Program交给Executor去执行时，Executor必须把Program中的『描述』信息，转化为真正可执行计算的Operator和可存储变量数据的Variable。这就是C++中的Variable和OperatorBase。

为了支持不同种类的数据，因此Variable是动态模板类。Variable代表可被算子计算使用的变量，即内存数据。Variable通过封装，内部可以存储多重类型的变量。如下图所示（来自var\_type\_traits.h）：



为了支持各类OP，OperatorBase只是个几类，定义了关键接口函数。OP数量很多，Paddle提供了较好的注册、使用机制。具体可学习胡晓光的《PaddlePaddle深入浅出》和陈威行的《Paddle Fluid框架执行逻辑梳理》。

* + 1. C++中的Scope

Scope是运行时概念，主要用于训练过程中Var的管理。在Executor运行过程中，将某些Var指针放到同一个Scope中，这些Var可能具有相同的生命周期、或者相同的属性，放到一起方便管理。

* + 1. Place

Paddle支持在多种计算机硬件进行模型训练。Place用于描述，训练所用的硬件。如CPU、GPU、ARM、FPGA等等。

Executor在实例化C++ OperatorBase、Variable时，会根据Place的不同，实例化对应的OPKernel负责计算。也会根据Place，将Variable用于存储数据的内存，分配到对应的Place中去。

* + 1. Executor

概念上Executor是执行器，负责Program的执行。用户通过Python接口定义Executor实例，传入Program进行Program的执行。

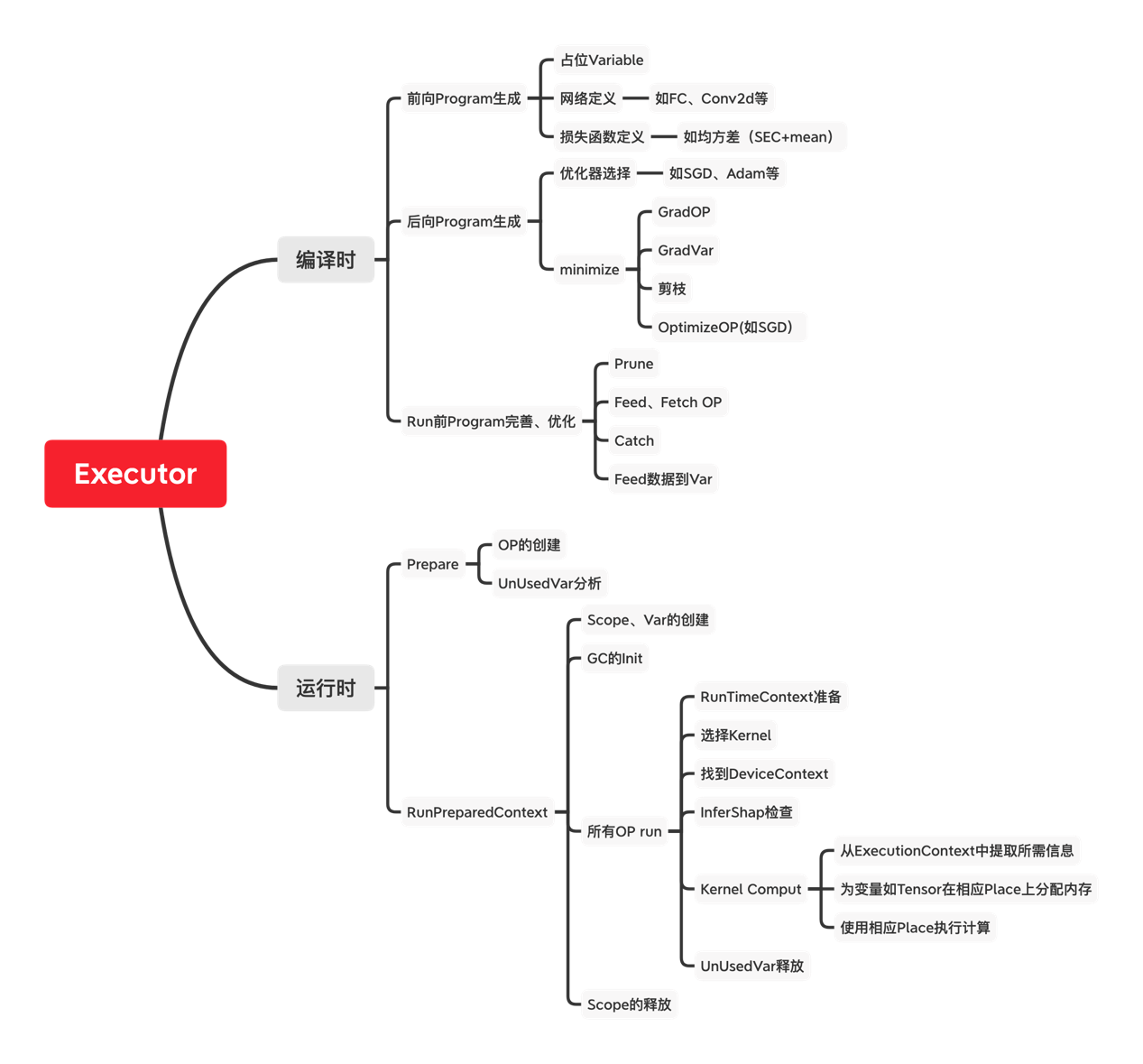
Python端的Executor是编译时和运行时的过度。他完成了Program最后的一些构造和优化，完成了C++ Executor的创建，启动了C++ Executor的Run。Python的Executor负责整个Program的启动执行，因此，他传入C++ Executor Run的BlockID为0。因为Block0为主Block。

C++的Executor可以说是真正的执行器。他拿到Program的Desc，将其转化为C++可执行的Variable和OperatorBase，并完成执行。

我们通常说的Executor指的是C++的Executor，或者指整个静态图+Executor。这需要我们结合语境判断。

* 1. 整体流程概览

Executor可分为编译时和运行时，两个阶段。编译时主要由Python完成，主要负责Desc（Desc指的是ProgramDesc+BlockDesc+OpDesc+VarDesc）的构造和优化。运行时获得编译时提供的Desc描述，将其转化为可执行的C++对象，并完成模型的执行。执行过程可拆解成下图所示（后文详细展开）：



这些阶段对应到（用户视角）代码中，可以参考下图代码中的注释：



* 1. 编译时

编译时的任务是生成Program。他的输入可以认为是用户调用的API，用户通过调用API定义网络模型。他的输出是Program，Program中拥有用户所描述的网络模型的全部信息。下页图片是一个Program的图片表达，3.2的Demo可以认为是编译时的输入，下页图片可以认为是编译时的输出。

图片见下页

Program由多个Block组成使用Vector存储。Block中有OP和Var。OP执行是有先后循序的，因此OP使用deque存储。OpDesc中定义了inputs\_、outputs\_，里面包含了Var的Name。

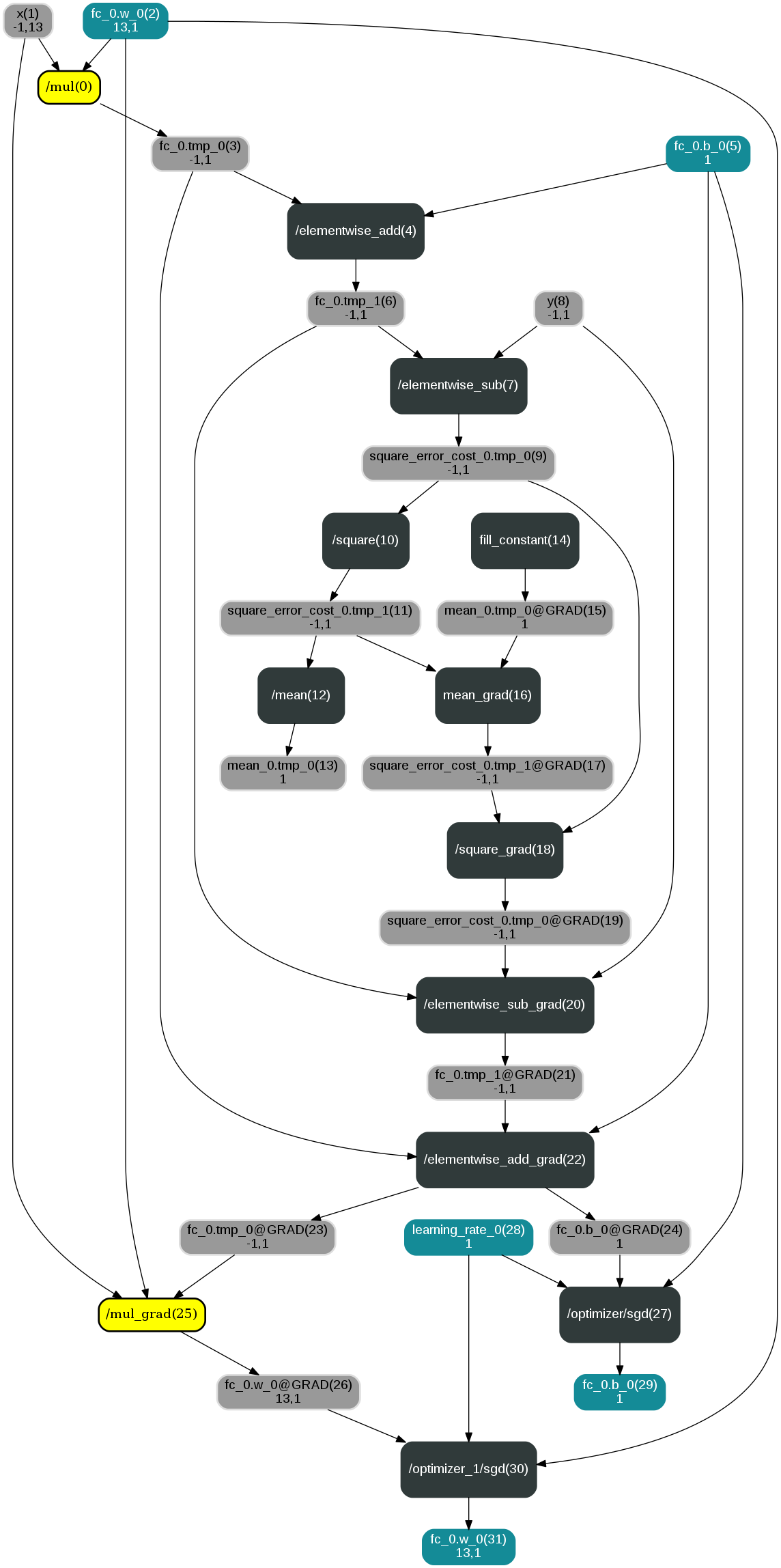
通过OpDesc中的inputs\_、outputs\_，我们隐晦的表达了一个图。这个图的节点是Var和Op，边是输入输出的依赖关系。这是个真正的图，因为它拥有图的所有必要信息，但是在数据结构存储中，表达的是不直接的，因此说它隐晦。

**结论，编译时，产出的是能够充分表达深度学习网络定义的Program。Program由Block构成，Block由OpDesc和VarDesc构成。Block的本质是以OpDesc、VarDesc为节点，以输入输出为方向的图。但在Block中存储的形式，让这个图的表达比较隐晦。**

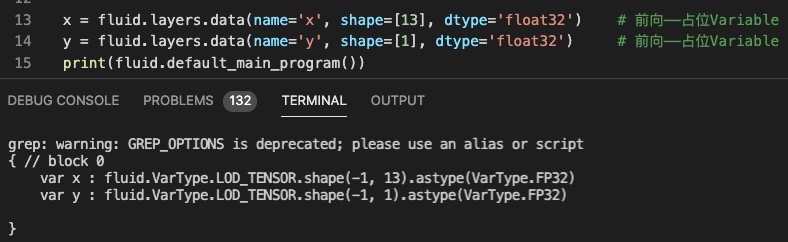
从下图中，我们看到了几类节点：

1. 我们自己定义的X、Y
2. FC网络所必要的OP，如mul、elementwise\_add、relu
3. FC网络计算所产生的一些临时中间Var，如fc\_0.tmp0
4. FC网络的参数（Parameter），如fc\_0.w\_0，fc\_0.b\_0
5. 计算均方差所产生必要OP和Var
6. 每个前向OP，所对应的，计算反向下降梯度的OP\_grad
7. 每个前向Var、Param的反向Var\_grad。注意：不是所有的Var都需要计算反向Var\_grad，比如我们输入的X、Y计算反向grad就没有意义
8. 最终优化参数的sgd\_op

其中1~5是前向产生的，根据用户的网络定义生成。6~8是Paddle根据1~5的信息，由Paddle编译时自动补全的。



* + 1. 前向
       1. 占位Variable



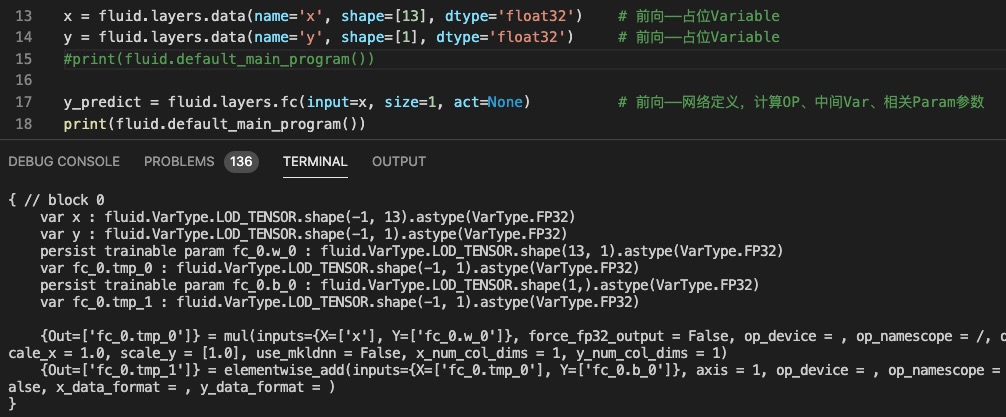
通过上图，我们可以看出，当执行完两个占位Variable API 后，Program的Block0中多了两个Var，分别为x、y，并记录了x、y的shape、类型等。

其创建过程大致如下：

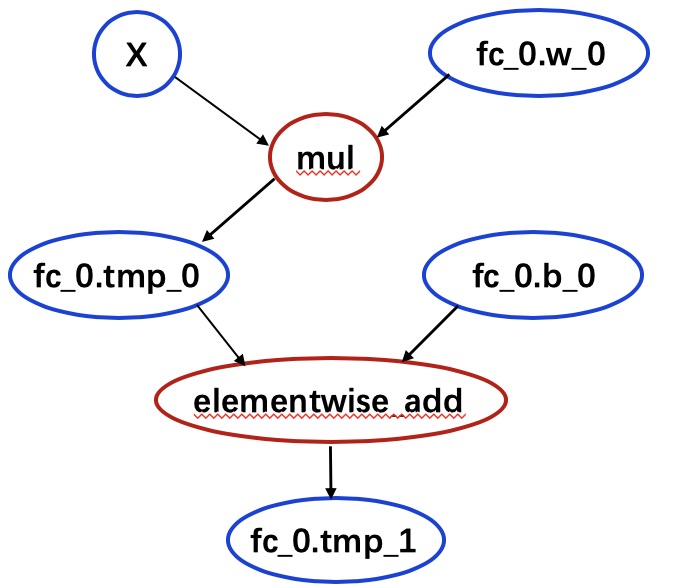
1. 因为这个Var是用户在全局声明的，因此，调用GlobalBlock（即Program的Block0）创建。
2. 创建Var即实例化了一个Python的Variable。Variable构造函数，通过调用C++ 的BlockDesc，创建一个C++ 的VarDesc，并在Python的Variable中持有一份(self.desc)。以下是Variable构造函数的部分代码截图：



* + - 1. FC网络定义



通过上图，我们看到fc函数，执行后，Program中多了fc\_0.tmp\_0, fc\_0.tmp\_1两个普通Var，多了fc\_0.w\_0，fc\_0.b\_0两个Parameter和mul、elementwise\_add两个OP，信息非常丰富（足够表达清楚网络模型需要的计算行为）。他们之间的关系如下：



总结：像fc这样的Python函数，他的任务就是构造OP、Var之间的关系，并把Desc存入Program中。

*题外话：这点映射出了Paddle中底层的大量工作：*

1. *我们需要开发大量类似于FC的网络模型，业界有的我们都要支持,如fc、conv2d；*
2. *每个网络模型都由N个OP组成，因此我们要开发更大量的OP；*
3. *OP要能够在不同的硬件（CPU、GPU等）运行，不同的硬件、不同的变量类型，OP的实现方式是不一样的。因此OP的开发是大量大量的！*
   * + - 1. 临时变量Var的构造

这里说的临时变量Var其实就是OP与OP之间传递数据的临时变量。FC创建Var过程与占位Variable基本一致，区别在于占位Variable的stop\_gradient为True，而FC中的Var的stop\_gradient为False。

为啥stop\_gradient不同呢？读完后文，这个疑问自会消失。

* + - * 1. Parameter Var的构造

Parameter的父类是Variable，但是Parameter的创建与Variable有些不同：

1. Parameter 的persistable为True，代表不能被优化掉
2. Parameter的Create，需要让StartupProgram和MainProgram都Create一份
3. Parameter的name通常为w、b字样
   * + - 1. FC中OP的创建

FC创建OP过程大致如下：

1. FC提供OP创建的基本信息，包括输入输出变量，OP类型，OP的Attr
2. 通过调用C++ BlockDesc的append\_op,获得一个创建好的C++OpDesc
3. 实例化Python端的Operator，在Operator的构造函数中，完善C++OpDesc中所需的信息

**总结：编译时，创建Var、OP的本质是构造C++的VarDesc、OpDesc，并通过与Python的对象关联，实现Python端对Desc的管理，进而实现Program的构造。**

***题外知识点：***

* ***StartupProgram***

*这里需要展开介绍一下StartupProgram。正常训练模型有两个Program，一个StartupProgram和一个MainProgram。StartupProgram负责Parameter的初始化，MainProgram负责训练模型的执行。*

*问题1：为啥不在MainProgram中加几个初始化的OP完成Parameter的初始化呢？因为MainProgram需要循环跑非常多轮，而Parameter只需要初始化一次。放在MainProgram中需要对初始化Parameter OP进行特殊处理，只允许第一次Run时执行。*

*问题2：StartupProgram怎么初始化的？ create\_parameter时，给StartupProgram传入了with\_initializer=True的参数，用于标记需要插入初始化OP。因此StartupProgram在创建Var的同时，还向Block中插入相应的初始化OP，如fill\_constant、uniform\_random。用户在开发时，在Run MainProgram前，只要先Run StartupProgram就可以完成Parameter的初始化。详见create\_parameter代码。*

*问题3：StartupProgram初始化Parameter，MainProgram就可以共享使用吗？在编译时，StartupProgram和MainProgram分别Create了一个VarDesc，也就是VarDesc由2份。但是在Executor Run两个Program时，传入的Scope都是GlobalScope。因此在Run StartupProgram时，在GlobalScope中已经实例化了C++的Variable，并分配了内存，初始化了数据。到Run MainProgram时，就可以从GlobalScope中find到了，不会二次实例化Variable。详见Python Executor Run代码。*

* ***stop\_gradient & persistable***

*stop\_gradient是Python端Variable的成员变量，在C++里没有。主要用于反向minimize使用，因此只有Python端才用的到。*

*stop\_gradient标记了一个Var是否需要计算反向的grad，即梯度下降值。占位Var是外部Feed进来的，计算grad是无意义的。而FC中的临时变量，可以向上传导参数的修改期望，因此需要计算grad。*

*因为Feed、Fetch Var是在minimize（反向OP、反向Var 添加）之后添加的，因此不关心stop\_gradient。*

*stop\_gradient主要用于标记前向Var，因此grad Var不需要标记。而且grad Var只创建了C++ VarDesc，没有创建Python Variable，因此不涉及stop\_gradient。*

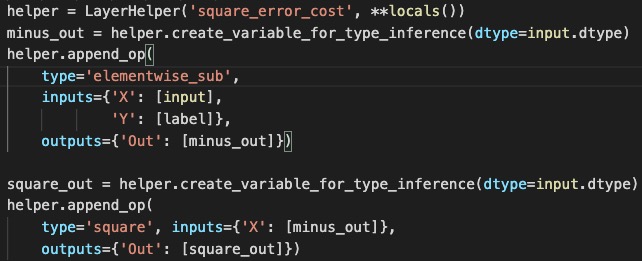
*persistable，是用于描述这个Var是否可以被优化掉，一般Feed Var、Fetch Var、Parameter是不允许被优化的。Persistable是在proto中就定义好的一个属性（Var有三个属性：name、type、persistable），因此与stop\_gradient不同，persistable属性需要同步到C++的VarDesc中。*

*以下做个总结（可能有些特殊情况下，这个总结是错的，但大多数情况下如此）*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | *Feed/Fetch Var* | *占位 Var* | *Parameter* | *tmp Var* | *grad Var* |
| *stop\_gradient* | *None* | *TRUE* | *TRUE* | *FALSE* | *无* |
| *persistable* | *TRUE* | *FALSE* | *TRUE* | *FALSE* | *FALSE* |

* + - 1. 均方差损失函数

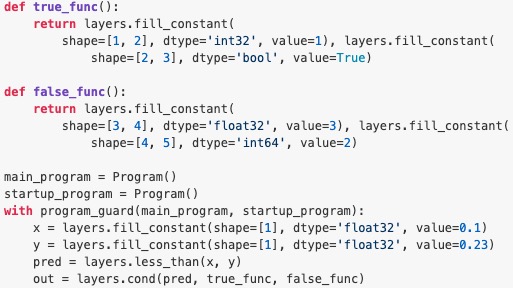
均方差，可以代表预测值与实际值的偏差。按计算先后顺序为差、方、均。这部分代码很简单，每个OP之后的输出均需要一个临时变量。以square\_error\_cost为例：



* + - 1. 控制流

3.2节中的Demo为顺序执行。如果我们希望在定义网络时通过IF、WHILE表述一些复杂的模型，则需要用到控制流。

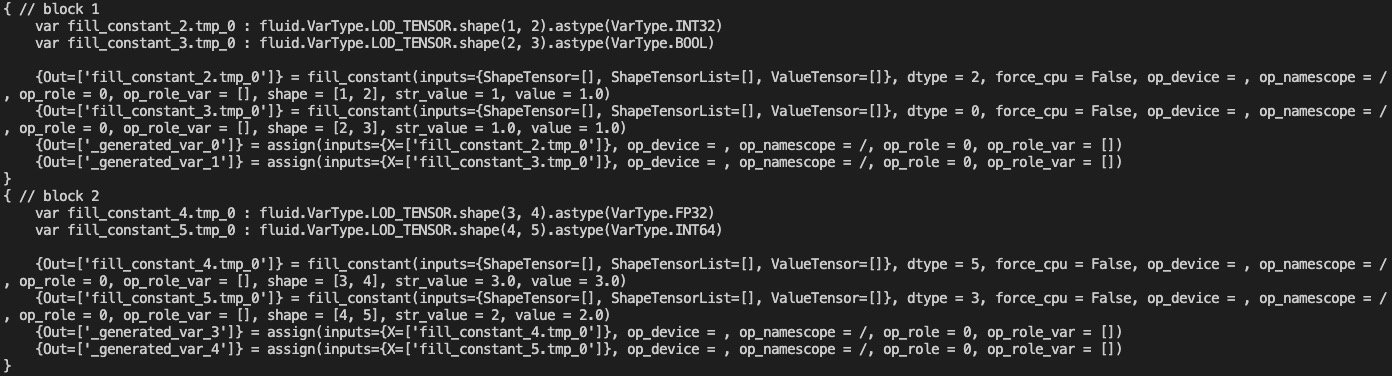
我们以Paddle官网cond的Demo为例，介绍控制流的编译时是如何构造Program的。代码如下：



首先我们看一下这段代码所生成的Program的Block0的Graph打印：



除了Block0外，编译时还会为true\_func和false\_func中的网络定义会分别创建一个新的子Block（Block1和Block2），分别描述两个分支，信息如下：



对于两个子Block的创建，请看下面代码中我添加的注释：



cond编译时构造Program的大致流程如下（详见python cond函数代码）：

1. 创建两个子Block，分别描述两个分支，完成两个子Block OP、Var的Append
2. 两个分支不可能都执行，到底执行哪一个？需要Block0中通过CondBlockOP决定。在Block0（父Block）中，创建两个CondBlockOP，每个OP分别用于决定其负责的Block是否被执行。其输入分别是pred和logical\_not(pred)，因此也引入了logical\_not的一个OP。那么哪个CondBlockOP负责哪个Block的执行呢？通过在CondBlockOP 的Attr中插入sub\_block，将整个BlockDesc都存进去。CondBlockOP Run的时候，从去获得blockid就知道其对应的Block是哪个了。
3. 将bool类型的pred转成int32类型的mask（就是castop）这部不知道处于啥原因。
4. 因为要返回两个Var，所以用两个select\_input\_op做选择，输出给外部。
   * 1. 后向

用户定义好前向网络后，编译时逻辑已将所有的OP、Var生成好，append到相应的Block中，而反向OP和相应的Var则由编译时逻辑自动生成。

* + - 1. 优化器选择

优化器，本质是深度学习的优化算法。Paddle支持常见的优化算法，如SGD、Adam、Adamax、Momentum、Adagrad等等。

这些优化算法分别由一个Python的类实现。如SGDOptimizer、AdamOptimizer。他们都集成自Optimizer类。而绝大部分编译时工作，其实是由Optimizer完成的，而子类主要是通过重写\_append\_optimize\_op函数，为Program添加optimize\_op（就是优化算法的实现）。optimize\_op是在minimize最后插入的。

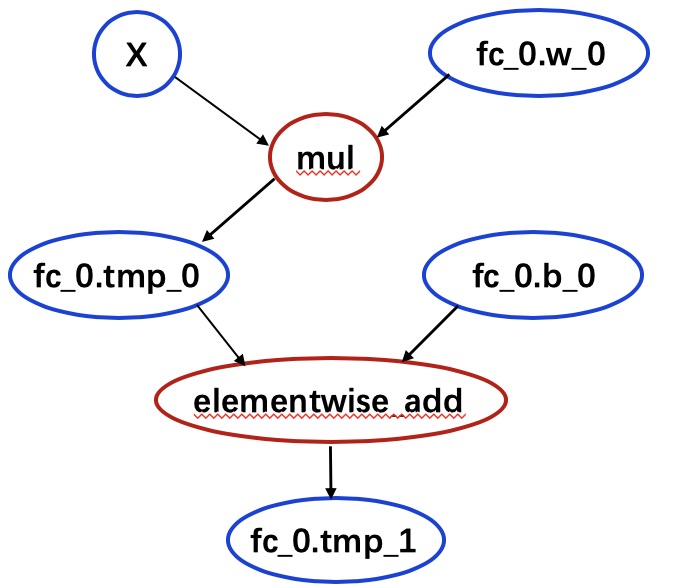
Optimizer的核心函数是minimize，该函数负责反向OP的生成。

* + - 1. minimize

minimize负责OP\_grad、Var@grad、Param@grad、Optimizer\_OP的生成。

minimize过程很是复杂，本人没能彻底看懂，大致流程如下：

1. 首先为每个Block构造no\_grad\_set，对于Block0，包括用户给的、not trainable的Parameter和stop\_grad为True的Variable。其它Block的no\_grad\_set为stop\_grad为True的Variable。
2. 为每个Block Append OP\_grad和Var@grad，其中Var@grad中也包括了Param@grad。Block0，直接Append到自己的Block中。ctrl\_flow Block，为其创建新的、对应的Block，往新的Block中Append；
   1. 遍历本Block中所有的OP，找到一个op\_path，op\_path是所有OP的一个子集，优化掉了部分不需要创建grad的op。
   2. 反向遍历op\_path中所有OP，调用C++的GradOpMaker，为该OP创建OP\_grad。
   3. 创建的OP\_grad，并不是都有用。结合input、output、no\_grad\_set，把不必要的路径做剪枝。打个比方：假如X是stop\_grad的，若OPx只生成了X一个变量，那么OPx没必要创建grad。同时，OPx之上的部分Var、OP也可以做剪枝（有点儿抽象不好理解）。以下图为例，假如fc\_0.tmp\_1为stop\_grad，那么elementwise\_add就不需要OP\_grad了，而之上的所有枝叶，均能被剪枝。



* 1. 把剩余的OP\_grad Append到对应的Block中去。

1. 2）的过程收集了大量的grad\_to\_var，需要rename保证名字不重复。
2. 将所有的Var@Grad Append到对应的Block中去。
3. 根据Parameter\_list和grad\_info\_map，生成一个清单。里面包含需要grad的Parameter和Param@Grad。
4. 调用每个优化器重载的\_append\_optimize\_op函数，根据Parameter和Param@Grad，生成对应的几个Optimizer\_OP。

minimize计算反向过程比较复杂，主要原因可能如下2点：

1. 所有的stop\_grad和not trainable Var，导致反向路径上并不是所有的OP均需要OP\_grad。甚至有些枝干，因为一个stop\_grad可以全部剪枝。因此minimize需要做优化，尽量减少不必要的OP的Append。
2. 由于control flow op以及对应的子Block的存在，导致反向过程多了很多特殊处理。

*这里结合PE会存在一个问题：minimize添加了反向到Program中，之后PE将Program转化成Graph，通过Pass对Graph进行fuse优化。假如mul+add需要fuse成fc\_op，那么add\_grad+mul\_grad也有必要fuse成fc\_op\_grad。这里恐怕设计的不太好。*

* + 1. Run前的补充优化

这部分，从API上，给用户的感觉是已经开始『Run』了，然而再真正启动C++ Executor Run之前，我们还会做一些准备工作。这些工作有些属于编译时，有些介于编译时和运行时之间。姑且全部算作编译时。

* + - 1. Prune(剪枝)

这部分没有细看。根据秋良和威行的介绍，大致背景如下：

在有些场景中，用户可能定义了一个复杂的网络。但用户最后Fetch的只是一个中间数据，或者提供的loss是从网络的一个中间节点开始反向的。那么类似这两种情况，用户所定义的复杂网络中，有很多网络分支是完全没必要Run的。我们可以提前剪枝掉，让用户尽早完成训练，获得期望的数据。

* + - 1. Feed、Fetch OP添加

用户在调用exe.run时，提供了main\_program、feed、fetch\_list，此时的main\_program拥有了整个网络的定义，包括后向定义。但是数据是如何送入网络的呢？用户又是如何观察网络迭代训练的过程数据的呢？

首先，用户是提前预处理好数据，每次通过feed，向exe.run中传入一个batch的真实数据，这个数据已经存储在内存中。那么这些数据需要送入相应的Var（占位Var）中，比如我们Demo中的占位Var X和Y。

实现方法是，Python端的Executor需要创建一个Feed Var，这个Var有多列（col），每列对应一个占位Var。之后为每个占位Var创建一个Feed OP，这个OP负责将Feed Var中相应列中的数据复制到占位Var中。这里的复制是指针拷贝，不是内存拷贝。

Fetch过程类似，也是需要创建一个Fetch Var和相应的多个Fetch OP。Fetch Var存储用户Fetch的所有数据。

Feed Var是如何拿到真实数据的，Fetch Var是如何将数据返回给用户的。我们再1.3.3.4中简单介绍。

* + - 1. Catch机制

我们知道，在训练过程中，1个batch需要exe.run一次，总体exe.run需要run非常多次。对于以上的Feed、Feath OP的添加，如果每次exe.run都执行一次，是不合理的、浪费效率的。

同样的问题还存在于C++ Executor 的Run中，如：

1. Context中将OPDesc转OperatorBase
2. Context中UnusedVar的计算
3. Scope中每个C++ Variable的Create

C++ Executor Run中涉及的三个业务我们后续展开，这里要表述的是，这些逻辑没必要多次执行，浪费效率。

为此，我们引入了Catch机制，在首次计算得到Program、Context、Scope后，我们在Python端Catch一份，以后每次用的时候，用Catch中的那份，而不是再重复构造。这就是Catch机制。

假如后续执行过程中，某些使用发生了改变，如fetch\_list。则需要重新计算Program、Context、Scope。因此我们引入了catch\_key，唯一标识一个Program、Feed、FetchList的组合。

* + - 1. Feed数据到Feed Var中，返回Fetch数据

之所以把这部分放到Catch机制后面表述，是因为虽然Feed OP、Fetch OP可以计算一次，Catch住，重复使用。但每次Feed的数据，Fetch的数据是不一样的，是不能Catch的。

在C++ Executor Run之前，Python端调用\_feed\_data函数，将用户提供的feed（真实数据），分别拷贝到Feed Var相应的col中（这里也是指针拷贝）。这个拷贝结合Feed Op完成，确保col于相应的占位Var正确对应。

类似的C++Executor Run结束后，将Fetch Var中的数据整理好返回给用户。

*补充一点题外话：我们每次向Executor中Feed的是一个Batch的数据，即N个X和N个Y。一个exe.run（即一个前向、反向过程），是执行1次还是N次呢？答案是1次。我们在每个OP执行的时候，都是拿着整个Batch的数据进行计算，并输出到下一个Var中的。*

*总结：在C++Run之前，有两种Var是已经准备好了的，包括数据显存和数据值。这两中Var分别是Parameter和Feed Var。*

* 1. 运行时

从1.3节，我们了解到，编译时已将用户定义的一个网络模型，转义成Program。这个Program是经过一定优化的，内部使用BlockDesc、OpDesc、VarDesc表述的一个『隐晦』的图。之所以隐晦，是因为OpDesc、VarDesc都是用Vector存储的，OpDesc中通过inputs、outputs隐晦的建立的关系（边）。需要补充说明的是：编译时Append OP是根据OP的执行先后循序Append的，即需要先运行的OP先Append，后运行的OP后Append。

因此，运行时的输入是：

1. Program；
2. Parameter Var、Feed Var；

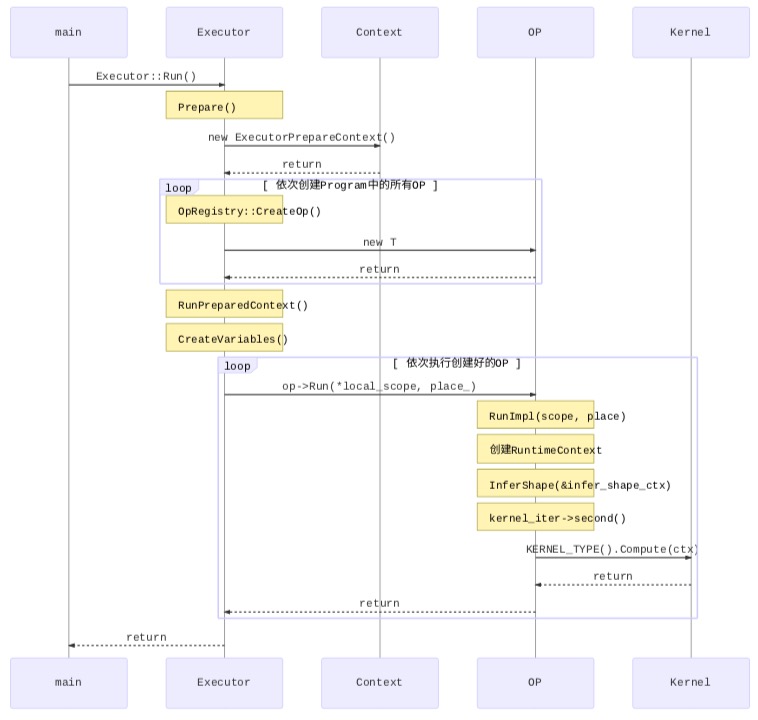
运行时的输出是：

1. 更新后的Parameter（这是我们训练的根本目的）。
2. 生成Fetch Var提供给用户。

运行时的主要任务有：

1. 转换Program，生成C++可执行的OperatorBase、Variable；
2. 顺序执行OP；
3. 完成内存清理，也要考虑内存优化。

以下是胡晓光给出的运行时时序图：



下文就每个环境，详细介绍。

* + 1. Prepare

Prepare做了两件事：

1. OpDesc到OperatorBase的转化
2. unused\_vars\_的构造

OpDesc到OperatorBase的转化非常简单，遍历所有OpDesc，OpDesc中包括了创建OperatorBase的所有信息。通过type在OpInfoMap中找到相应的OpInfo，调用creater即可创建。创建后的OperatorBase，以Vector存储在ExecutorPrepareContext中。

unused\_vars\_的类型是std::unordered\_map<const OperatorBase\*, std::vector<std::string>>。其含义是，当某个OperatorBase执行完成后，对应的几个Var将不在有其它OperatorBasse使用，可以提前释放。

什么样的var可以中途释放呢？

1. 不能是用户Fetch的，因为释放了用户就拿不到了；
2. 不能是persistable的，这个属性的意思就是不能被优化；
3. 只能是LOD\_TENSOR、SELECTED\_ROWS、LOD\_TENSOR\_ARRAY三中类型中的一种。这个为啥不清楚。
4. 在control flow的Block中，涉及到control op要使用的var不能被优化。代码中有三个函数在干这方面的事儿，看到的时候别疑惑：PrepareSafeEagerDeletionOnConditionalOpAndConditionalGradOp、PrepareSafeEagerDeletionOnWhileOpAndWhileGradOp、PrepareSafeEagerDeletionOnRecurrentOpAndRecurrentGradOp

构造unused\_vars\_的算法很容易看懂，遍历一遍ops，标记每个变量最后一次被那个ops使用（inputs、outputs）。之后反向生成unused\_vars\_即可。

**在Catch机制下Prepare过程是由Python端提前调用，将Context Catch住的。**

* + 1. RunPreparedContext

RunPrepareContext首先是在Scope中Create Variable，初始化Variable。（这个环节再Catch机制中，由Python主动调用CreateVariables，之后RunPrepareContext就不用Create了。）

之后循环、顺序执行Context中的OP，每个OP执行完后根据UnusedVar，释放已经不再使用的Var交给GC。

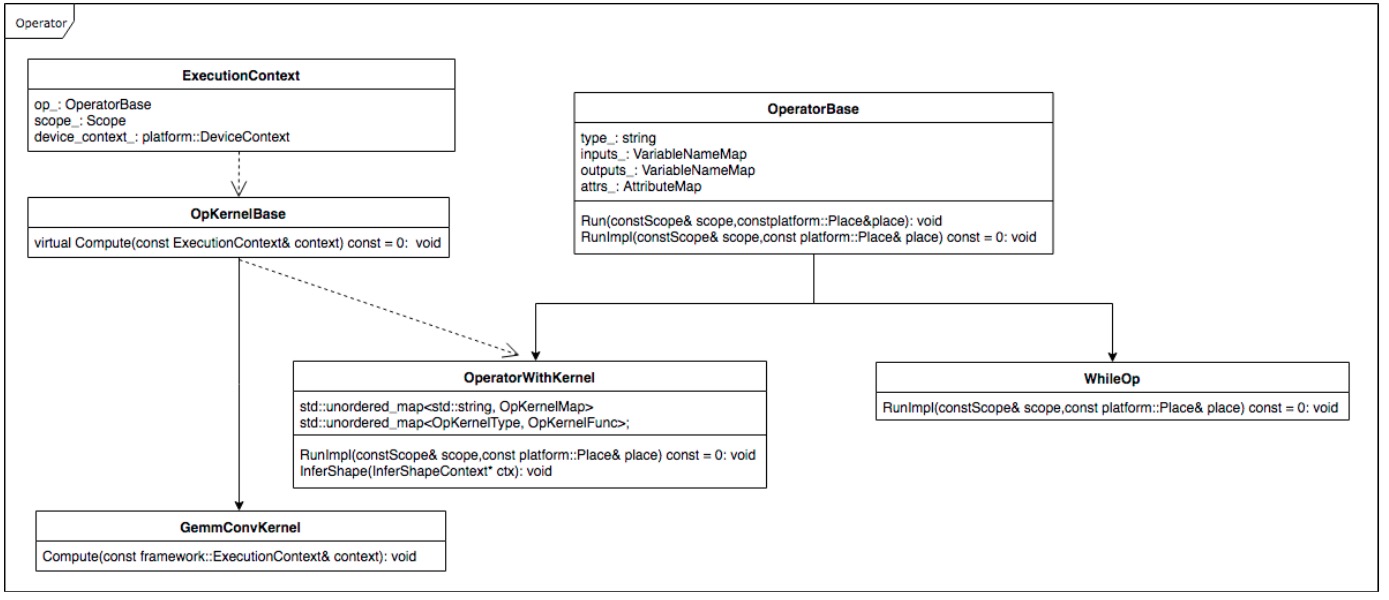
最后视情况释放Scope。

* + 1. OperatorBase

Paddle中所有OP都是继承自OperatorBase，OperatorBase的核心方法是Run，Run函数调用RunImpl实际执行计算。

Paddle的设计理念是可以在多种设备（Place）上运行，加上变量类型的不同。往往一个OP在不同情况（Place、变量类型）下的代码实现完全不同。因此引入了OpKernel的概念，即OperatorWithKernel，他继承自OperatorBase，并重写了RunImpl方法。

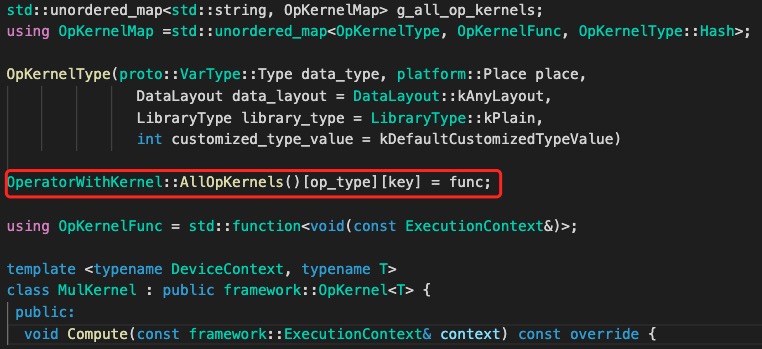
此外，还有一种控制流OP，如WhileOp，是直接继承自OperatorBase重写RunImpl的，我们在下节介绍。以下是胡晓光给出的类图：



* + 1. OperatorWithKernel OP Run
       1. Kernel的介绍

Kernel的注册于OP的注册类似。这里不再介绍。

Kernel在Paddle中的管理代码片段截取如下：



通过**[op\_tpye][key]**，可以看出，Kernel通过op\_type(std::string)+key(OpKernelType)两级索引管理OpKernelFunc。而key又由很多信息组成，不同信息组成的key不同，就需要不同的OpKernelFunc支持。

而func是指向一个Kernel Compute函数的函数指针。Compute是Kernel执行计算的主体，传入的是ExecutionContext参数。

* + - 1. RuntimeContext

RuntimeContext中存储了OP运算时需要用到的Variable指针，包括inputs和outputs，使用VariableValueMap = std::map<std::string, std::vector<Variable\*>>方式存储。

* + - 1. OP 的 Run

**Step1:RuntimeContext的构造**

OperatorWithKernel中存储了其计算的输入、输出的变量名，在进行计算前，他需要找到『变量值』，即Scope中的Variable。OperatorWithKernel通过构造RuntimeContext将『变量值』存到RuntimeContext中。

为了提升性能，OperatorWithKernel中也有一个小的Catch功能，可以通过设置Arrt kEnableCacheRuntimeContext将该功能打开。此功能若打开，OperatorWithKernel对象中将持有RuntimeContext，进而不用每次都构造RuntimeContext。

**Step2：DeviceContext的准备**

DeviceContext是提前初始化好的，不同的硬件所需初始化的工作不同，初始化后需要保留的变量不同。因此使用不同的DeviceContext负责初始化的执行和变量的存储。OP在Run时根据Place到DeviceContextPool中获得对应的DeviceContext。

*题外话：上面已经提及好多个XXXContext，都有哪些Context呢？都是干嘛用的呢？胡晓光做了总结：*



**Step3：Kernel的选择**

3.4.3.1中我们介绍了Kernel的管理。Kernel的选择，其实就是构造op\_type和key，然后到AllOpKernels()里去查找的过程。在ChooseKernel函数中实现。最终将Kernel的函数指针放到了kernel\_func\_成员变量中。

**Step4：数据转换**

为啥要数据转换？对于Kernel的一些输入变量，如果其变量格式和Kernel可处理的变量格式不一样，会导致Kernel无法处理。这里所说的变量主要指LoDTensor和SelectedRows两种类型。格式不同的原因包括（NeedTransform函数中实现）：

1. 其所在的Place的类型不同
2. 基础数据DataType（如BOOL、INT16、FLOAT）
3. DataLayout（如NHWC、NCHW等）不同

因此需要进行数据格式转换。

这里有个问题，转换好的、新的Variable（trans\_var），并没有更新到RuntimeContext中。而后续计算使用Variable时，都是使用的RuntimeContext中的Variable。那数据不是白转换了吗？——这个是在PrepareData函数中，每次数据转换完，他替换了ctx得input\_vars[i]。因此，这只是更改了RuntimeContext中的Variable的指针。而原来的Scope中的Variable还是指向数据转换之前的个Tensor的。TransferInplaceVarsBack函数可以把Scope中的数据格式也给换成新的。

**Step5：构造ExecutionContext，并执行Kernel的Compute函数**

ExecutionContext可以认为是一系列参数的载体，这些参数，Compute将来可能会用到。

执行Kernel就是调用已经选好的Kernel的函数指针，去执行

*其余还有很多开关涉及的逻辑，在这里不再介绍。*

* + - 1. Kernel 的 Compute

Kernel的Compute负责更底层的负责OP的实际计算逻辑，主要步骤大概有：

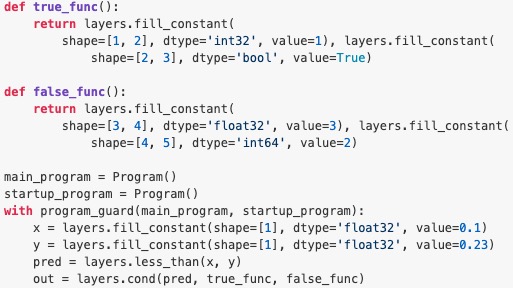
1. 从Context中取得inputs的Tensor指针；
2. 为output Variable的Tensor分配真实的内存（可能是内存也可能是显存）；
3. 调用更底层的算子执行。

Kernel Compute还不是最底层的实际执行逻辑，还有更底层的。。。但于框架调度关系已经不大。

* + 1. Control Flow OP Run 的细节

Paddle中的控制流也是OP，他不负责实际数据的计算，因此不需要Kernel。这种OP直接继承自OperatorBase，并重写RunImpl函数。并在RunImpl函数中实现控制流的主体逻辑。

我们仍以3.3.1.4中的cond举例介绍，他的C++ OP是ConditionalOp。



首先，C++ Executor 的Run的功能是对一个Block进行Run。因此Python调用时，因为要Run Program主体，因此传入的Block ID为0。那么什么时候Run其它的Block呢，由谁发起的呢？

在Block 0中，Control Flow OP的定义。比如Cond的Demo。Paddle为true\_func和false\_func分别创建了一个Block，也同时向Block 0中加入了两个ConditionalBlockOp，每个OP指向其中一个Block。但C++ Executor运行Block 0时，根据pred Var的值，分别运行这两个ConditionalBlockOp，其中一个会执行need\_run。

在need\_run中，为该Block重新准备了Scope、Executor，然后让这个Executor去Run这个子Block。代码中的skip\_vars，就是3.4.1节中讲到的，不能被优化的Var。

值得注意的是，Control Flow OP Run的过程中，需要重新构造OperatorBase、UnusedVar、Variable（Scope），性能上是有些欠理想的地方的！

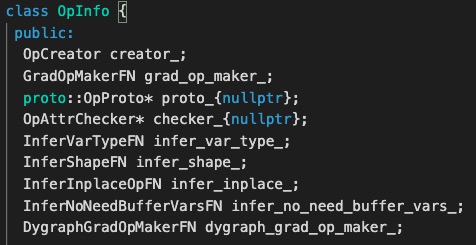
* 1. 其它
     1. OP的管理

Paddle需要不断新增OP，支持各种计算。因此Paddle必须定义统一的调用接口，和OP管理方案。统一调用接口即OperatorBase。OP的管理则包括了OP的开发、OP的注册和创建使用。

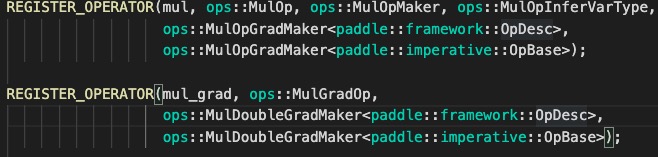
OP的开发可参考官方文档：<https://www.paddlepaddle.org.cn/documentation/docs/zh/advanced_guide/addon_development/new_op/new_op.html>

OP的注册在陈威行的《Paddle Fluid框架执行逻辑梳理》讲的比较细。本文做点文字表述：

1. 所有的OP有OpInfoMap统一管理，他是单例实现，核心成员是std::unordered\_map<std::string, OpInfo> map\_。注册时向其插入OpInfo。需要创建时，查找OpInfo，完成相应的创建、Check等操作。
2. OpInfo包括了OP构造环节所需的所有类、函数指针：

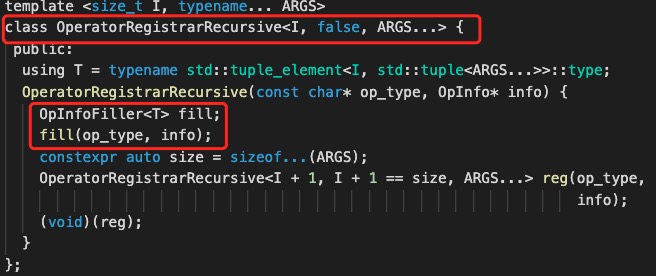


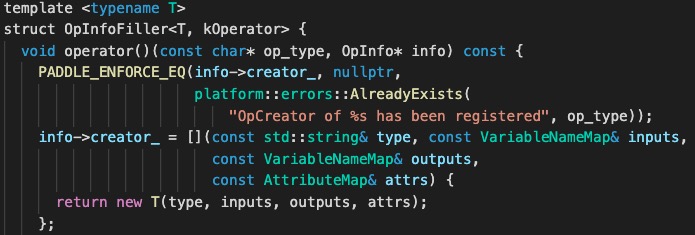
1. OP通过REGISTER\_OPERATOR进行注册，一次性将所有『构造执行实体』传入。OP、OP\_grad需要注册两次。



1. REGISTER\_OPERATOR实例化了一个OperatorRegistrar，OperatorRegistrar用到了C++中的模板元编程，模板可以被递归调用，在递归过程中，完成OPInfo中一个成员变量的初始化。

模板元编程，使得OperatorRegistrarRecursive被循环构造，每次构造根据模板参数的不同，OpInfoFiller<T> fill会选择不同的Filler，每个Filler负责了OpInfo一个成员变量的初始化。



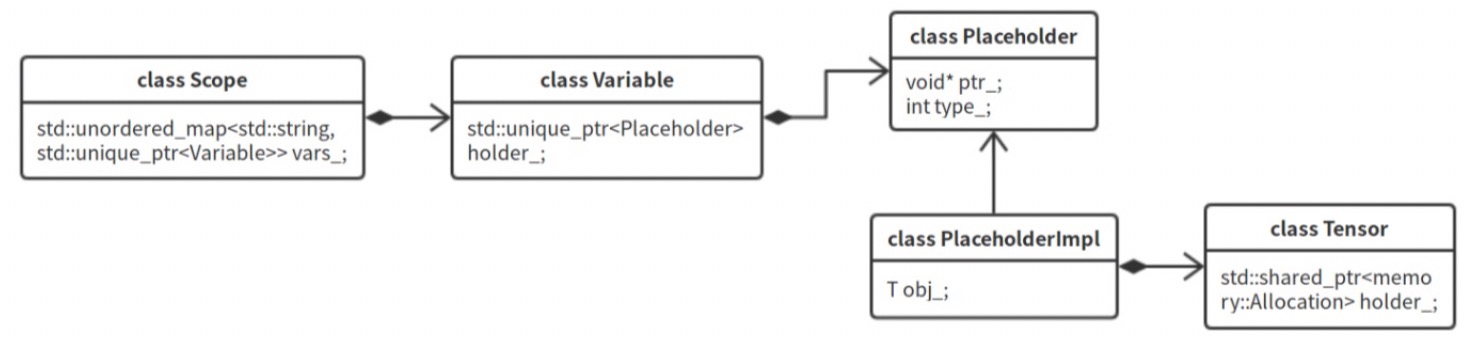


OP的创建，则是在C++ Executor Run时，根据Desc中的信息，通过OpInfoMap找到相应的OpInfo创建OP。

OpInfo中出了创建OP的creater，还有很多和构造相关的类，他们都是干嘛用的，尚未深入学习。

* + 1. Scope、Variable、Tensor、mem

以下是相关类的类图，对应相应的代码一目了然。Placeholder、PlaceholderImpl也很简单，不再展开介绍。



Scope是运行时的一个概念，可以认为他是Variable的『容器』,他通过unique\_ptr方式持有一个Variable的指针。若Scope析构，相应的Variable也会被释放。注意，在RuntimeContext中持有的是Variable\*，而这个指针是从Scope中unique\_ptr中Get的，因此不需要释放。但，如果Scope析构了，RuntimeContext中的这个指针就是野指针了。

因此，Scope是用于管理Variable用的，如果有一些临时Variable在某些特定环节结束后即可释放。我们可以Scope.new\_scope()。这样，在父Scope下创建一个子Scope。然后将临时Variable放到子Scope中。当子Scope释放时，这些临时Variable也就跟着释放了。因此，Scope用于管理Variable释放，很好用。

**内存分配**。Scope中的Variable是在Executor Run之前，统一Create的，这时候就创建了PlaceholderImpl和Tensor。但是！这个时候的Tensor是空的（holder\_为NULL）。Tensor的实际内存得到OpKernel Compute时才会分配。

**注意，以上说法有个特例，那就是FeedVar。这个Variable，在Executor Run之前就放到Scope中了，而且将Tensor的holder\_也赋值了。**

还有值得分析的是，Tensor中的holder\_是share\_ptr类型的。在Paddle中经常存在多个Tensor贡献一块内存数据的情况。因为Tensor holder\_所持有的数据是大量的，内存申请、内存复制都非常耗时，也浪费内存空间。因此代码中有很多地方调用ShareDataWith函数共享内存。其实，Scope、Variable、Tensor都是『管理』内存的，而不真实持有内存，可以说都是索引。只有到holder\_才真实持有内存。

**内存释放**：Scope一般都是根据业务需要，其管理的Variable不再使用了，就会统一释放Scope。Scope释放的时候，一定释放了Variable和Tensor。但是，只有Tensor的holder\_是shared\_prt,因此需要引用计数归零时，才会释放真实内存（holder\_）。

* + 1. gc

gc(Garbage Collection)，用于内存的回收、清理。在Executor中的内存回收比较简单。3.4.3的Prepare函数中，构造了一个UnusedVar，描述了这个Variable在某个OP之后就无人再用了，可以回收了。那么在某个OP执行之后，我们就可以提前释放其对应的UnusedVar。这里所谓的释放Var并不是释放Variable，而是释放Variable对应的Tensor中的holder\_。因为holder\_中的内存才是占用较大空间的内存。

**gc的意义**。为什么我们要在Executor运行过程中清理内存的，清理内存也是要消耗CPU的计算资源的，那不是拖满了Executor的计算速度吗？——Executor每次只训练一个Batch，如果每次训练的Batch大，那么需要训练的Batch的总个数就会少，循环执行exe.run的次数就会少。这样可以大大加快训练时间。但是，如果Batch太大了，内存是装不下的。如果我们能够优化内存的使用，可以让每次Run的Batch增大，就可以提升训练速度。这是内存回收的意义所在。

Executor中gc的执行过程：

1. 声明一个std::deque<std::shared\_ptr<memory::Allocation>> garbages，将需要释放的holder\_(share\_ptr<memory::Allocation>),move到garbages中。这样，Tensor的holder\_就位nullptr了。
2. 通过gc->Add(std::move(garbages));将garbages再move到gc的Container中，这个Container其实是garbages的引用。
3. 再通过auto \*container = new Container(std::move(objs))，new 一个指针类型的Container。
4. 通过ClearCallback([container] { delete container; })，将指针类型的container交个gc去delete。当container delete时，里面的shared\_ptr将跟随释放，其引用计数将减1。
   * 1. pybind11

pybind11是众多Python调用C++的方法之一。pybind11使用比较简单，对C++11的支持也很好。Paddle使用pybind11完成Python调用C++内核。

Paddle的C++内核会被编译成core.so，Python端可通过core.XXX接口进行调用。

* + 1. Var的唯一取名

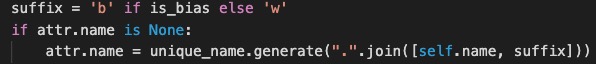
unique\_name.py定义了一个字典，在UniqueNameGenerator中。记录了每个name被命名的次数idx，首次使用id index 为0。之后每次name发生重复，则一次为1、2、3…。其核心函数为generate。

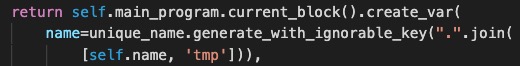


占位Var的取名以用户命名为准，内部不再做加工。若用户定义了两个变量，name重复。则框架内部只有一个Var。而用户态的两个Var均指向这一个框架内的1一个Var。若用户没有提供name，则框架为其创建唯的一个name。

layers下的函数均有可能重复使用，如fc、mean等。这类函数均有一个唯一函数名（name）。如fc\_0,fc\_1,,mean\_0等，由LayerHelper+unique\_name.py共同完成。

fc内部声明的临时变量、Parameter的name均以fc的唯一函数名+”.”为前缀（如fc\_0.），以其自身名字为后缀(如tmp、w、b)，并通过unique\_name.py为其后缀添加唯一ids。如：





反向Var的取名，根据其前向VarName，加上”@GRAD”即可。[如fc\_0.tmp\_0对应的反向Var名字为fc\_0.tmp\_0@GRAD](mailto:如fc_0.tmp_0对应的反向Var名字为fc_0.tmp_0@GRAD)。其实现在\_append\_grad\_suffix\_函数中。

而OP是无状态的，因此OP不需要为其取名。其输入+输出+type+attr已经能唯一描述一个计算。

* + 1. LodTensor&SelectedRows

LodTensor和SelectedRows都是压缩后的Tensor。

对于一些每行长短不一致，而且长度差异很大的Tensor。如果我们将所有短的行全部不填成最长行的长度。一方面占用大量的显存。另一方面也增大了计算量。这时候，我们使用LodTensor。LoDTensor将长度不一致的维度拼接为一个大的维度，并引入了一个索引数据结构（LoD）来将张量分割成序列。具体可参考：<https://www.paddlepaddle.org.cn/documentation/docs/zh/beginners_guide/basic_concept/lod_tensor.html>

对于Tensor中有大量行全部为0的情况，我们可以记录这个Tensor一共有多少行，然后把有非0数据的行存下来（包括行号），全部为0的行不再存储。这样可以大大压缩内存使用量。

* + 1. Program Graph的打印

Paddle生成的Program中Var、OP数量很大，是一个非常复杂的Graph。如果靠短点调试或者日志打印。以我们大脑的能力很难想象Program中Graph的长相。Paddle中有很好的工具可以将其转换为Graph图片输出。

Program的图片打印可以参考：

<https://github.com/wzzju/wzzju.github.io/blob/master/assets/tools/paddle_model_visual.py>

PE中Pass后的Graph可参考以下链接中的debug\_graphviz\_path的使用：

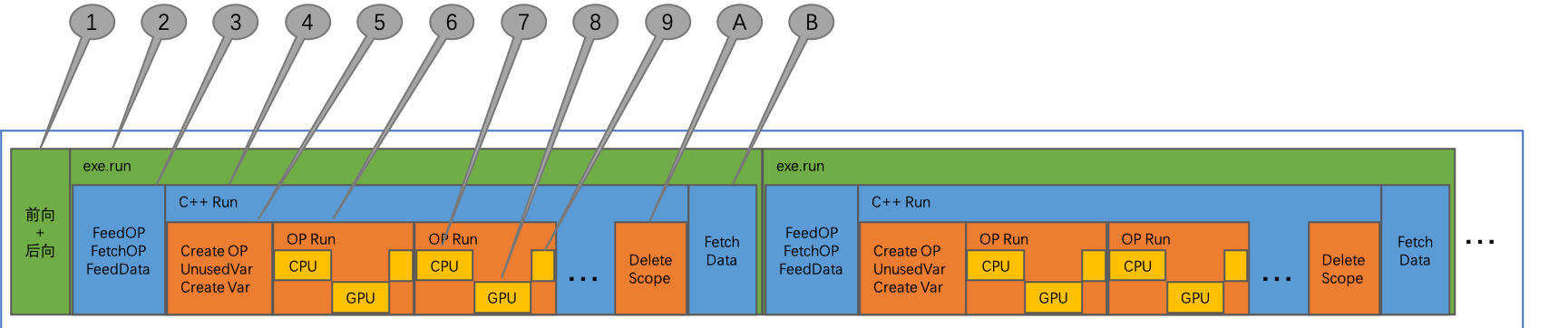
<https://www.paddlepaddle.org.cn/documentation/docs/zh/api_cn/fluid_cn/BuildStrategy_cn.html#buildstrategy>

他们生成的Graph的格式为Graphviz，可以网上查询Graphviz如何使用。转换成png的命令可参考：

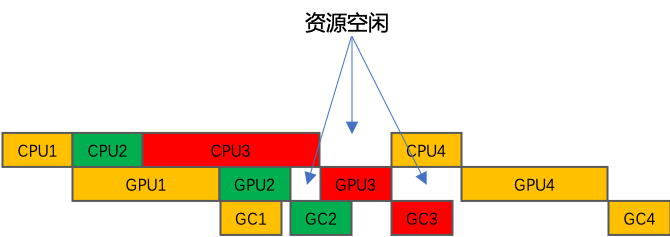
*dot graph\_original\_graph -T png -o original.png*

* 1. 从Executor看PE的性能优化

Executor的执行可以概括为下图：



注意，图中对给人的感觉OP的CPU、GPU是串行交替执行的。事实并非如此，CPU、GPU之间是一定程度上并行的：



以下先对CPU、GPU的并行进行一个解释：

上图假设有4个OP（OP1，OP2，OP3，OP4）那么CPU、GPU、GC的执行有如下规律：

1. GC也由CPU执行，但是单独线程。因此环境准备与GC之间是可以并发的；
2. GC1必须等到GPU1执行完才能执行，GPU1必须在CPU1环境准备好后，由CPU1触发；
3. GC1的销毁任务，是CPU1提前提供的，GC1阻塞等到GPU1结束后，开始销毁；
4. CPU1结束后，大多数情况可以直接启动CPU2的环境准备工作。这需要有个前提，就是OP内没有特定的cudaStreamSynchronize wait动作。如果有必要的cudaStreamSynchronize则会造成CPU3~CPU4间的CPU资源浪费，进而势必存在GPU3~GPU4间的资源浪费。
5. 如果CPU3的环境准备时间太长，导致GPU所有任务都完成了，那么也会造成GPU的无任务状态，造成GPU2~GPU3间的资源浪费。

以下对Executor的执行过程，以及存在的可能的优化点进行分析：

1. Program Desc的生成，这部分主要是『原样』描述前向网络，自动生成后向网络。

**优化点1：**而后向网络中有些路径是没有必要的，如stop\_grad，因此可以做剪枝。

1. 循环执行exe.run，每次执行一个Batch。假如1个数据集由N个Batch，一共训练M轮，则exe.run需执行N\*M次。
2. 补全Feed、Fetch OP和Var，并将训练数据集数据放入Var中。

**优化点2：**如果用户Fetch的变量，不需要将整个网络跑完即可提供给用户。那么网络中可能存在『不需要』的分支。我们可以进行Prune剪枝。

**优化点3：**除了stop\_grad、Prune，在C++ Executor Run之前还有很多能做的优化。因为Program Desc是一个接近于用户表述的网络的『原始』描述。在PE中，Paddle将Program转换成Graph，通过**BuildStrategy**（一系列的**Pass）**，对Graph进行优化。这样，可以在C++ 实际Run之前，进行Desc级的大幅优化。

**优化点4：**如果每次都进行补全Feed、Fetch OP和Var的操作，属于重复计算。我们可以将Program **『Catch』**住，减少这部分重复计算。

1. C++ 的Executor Run，主要包括了环境的准备、所有OP的Run、Scope的释放。
2. OP Run之前的环境准备。包括OP的Create、UnusedVar的构造、Scope中Variable的Create。

**优化点4：**OP的Create和UnusedVar的构造，组成了ExecutorPrepareContext，如果每次exe.run都重复的执行这两个构造，也是不需要的。可以将Context **『Catch』**住。

**优化点4：**Scope中的Variable的构造也不需要重复构造，可以将Scope**『Catch』**住。

1. 一次C++ Run将有N个OP循序执行。每个OP执行均包括了计算前的准备、计算、内存的回收。计算前的准备和内存的回收，由CPU完成。计算往往由GPU完成。
2. CPU完成计算前的准备工作。3.4节中的内容几乎全部属于计算前的准备。
3. GPU完成计算。在实际训练中，绝大多数使用GPU完成实际计算任务。因涉及GPU编程，是Kernel更底层的逻辑，本文中没有介绍。
4. 内存的回收，指的是GC由CPU完成。

**优化点5：**训练过程是期望CPU、GPU都100%负荷工作的，即尽量充分利用计算资源。因此Executor、PE都应考虑，如何减少CPU、GPU的空闲间隙，尤其是GPU的空闲间隙。

**优化点6：**CPU是多核的，GPU跟有有大量的计算单元。一个OP可能无法完全让CPU、GPU资源利用率达到100%。同时，很多时候，存在多个OP均已具备Run的条件（Run的条件，即inputs的Var均已计算得出）。这时候，可以**让多个OP并行Run**，进而充分利用CPU和GPU。PE的**GraphExecutor**中实现了类似功能。

**优化点7：**一台服务器，可以有8个GPU。可以将1个Batch分成8份，让8个GPU同时计算，然后将8个GPU的Parameter@Grad全部收集、再分发到各卡去sgd。这就是**PE的一机多卡**。

**优化点8：**CPU 7）、9）两步，最频繁的操作就是显存的申请和释放。如果能够加速显存的申请和释放，那么CPU的计算效率将大幅提升。同时，如果能够尽早回收显存（GC），可以让Batch更大。这就是**显存优化**。

**优化点9：**部分训练由于计算量、数据量过大，单机多卡无法满足需求。我们需要调配多台计算同时计算。这就需要**分布式**控制。

1. 将Scope中的Variable释放。
2. 计算玩后，将用户Fetch的结果返回。

性能疑问：

1. Pass的优化有一个问题，他是对整个前向+反向的Graph进行优化。那么比如前向有个fuse，一定对应个反向的fuse。前向生成完就进行pass，再生成后向，将更合理一些。
2. 控制流是没有catch的，需要在控制流OP中进行子Block的OP Create、UnusedVar构造、Scope Var Create。
3. 1个OP就将GPU占满，其它排队。那么再并行计算就没意义了。CUDA的Gird、Block、Thread是否需要控制？
4. 8个GPU并行，reduce all必须8个都计算完，会不会出现7个等1个的情况？