
PaddlePaddle

V2.3.0

PaddlePaddle developers

2022 年 09 月 20 日

Contents

1 安装指南	1
2 使用指南	71
3 应用实践	411
4 API 文档	415
5 贡献指南	417
6 常见问题与解答	605
7 2.3.1 Release Note	637
8 2.3.0 Release Note	643

Chapter 1

安装指南

1.1 安装说明

本说明将指导您在 64 位操作系统编译和安装 PaddlePaddle

1. 操作系统要求：

- Windows 7 / 8 / 10，专业版 / 企业版
- Ubuntu 16.04 / 18.04 / 20.04
- CentOS 7
- MacOS 10.11 / 10.12 / 10.13 / 10.14
- 操作系统要求是 64 位版本

2. 处理器要求

- 处理器支持 MKL
- 处理器架构是 x86_64 (或称作 x64、Intel 64、AMD64) 架构，目前 PaddlePaddle 不支持 arm64 架构 (mac M1 除外，paddle 已支持 Mac M1 芯片)

3. Python 和 pip 版本要求：

- Python 的版本要求 3.6/3.7/3.8/3.9/3.10
- Python 具有 pip, 且 pip 的版本要求 20.2.2+
- Python 和 pip 要求是 64 位版本

4. PaddlePaddle 对 GPU 支持情况：

- 目前 PaddlePaddle 支持 NVIDIA 显卡的 CUDA 驱动和 AMD 显卡的 ROCm 架构
- 需要安装 cuDNN，版本要求 7.6(For CUDA10.1/10.2)
- 如果您需要 GPU 多卡模式，需要安装 NCCL 2

- 仅 Ubuntu/CentOS 支持 NCCL 2 技术
- 需要安装 CUDA，根据您系统不同，对 CUDA 版本要求不同：
 - Windows 安装 GPU 版本
 - * Windows 7/8/10 支持 CUDA 10.1/10.2/11.1/11.2/11.6 单卡模式
 - * 不支持 **nvidia-docker** 方式安装
 - Ubuntu 安装 GPU 版本
 - * Ubuntu 16.04/18.04/20.04 支持 CUDA 10.1/10.2/11.1/11.2/11.6
 - * 如果您是使用 **nvidia-docker** 安装，支持 CUDA 10.2/11.2
 - CentOS 安装 GPU 版本
 - * 如果您是使用本机 pip 安装：
 - CentOS 7 支持 CUDA 10.1/10.2/11.1/11.2/11.6
 - * 如果您是使用本机源码编译安装：
 - CentOS 7 支持 CUDA 10.1/10.2/11.1/11.2/11.6
 - CentOS 6 不推荐，不提供编译出现问题时的官方支持
 - * 如果您是使用 **nvidia-docker** 安装，在 CentOS 7 下支持 CUDA 10.2/11.2
 - MacOS 不支持：MacOS 平台不支持 GPU 安装

请确保您的环境满足以上条件。如您有其他需求，请参考 [多版本 whl 包安装列表](#)。

5. PaddlePaddle 对 NCCL 支持情况：

- Windows 支持情况
 - 不支持 NCCL
- Ubuntu 支持情况
 - Ubuntu 16.04/18.04/20.04:
 - * CUDA10.1 下支持 NCCL v2.4.2-v2.4.8
- CentOS 支持情况
 - CentOS 6: 不支持 NCCL
 - CentOS 7:
 - * CUDA10.1 下支持 NCCL v2.4.2-v2.4.8
- MacOS 支持情况
 - 不支持 NCCL

第一种安装方式：使用 pip 安装

您可以选择“使用 pip 安装”、“使用 conda 安装”、“使用 docker 安装”、“从源码编译安装”四种方式中的任意一种方式进行安装。

本节将介绍使用 pip 的安装方式。

1. 需要您确认您的操作系统满足上方列出的要求
2. 需要您确认您的处理器满足上方列出的要求
3. 确认您需要安装 PaddlePaddle 的 Python 是您预期的位置，因为您计算机可能有多个 Python

使用以下命令输出 Python 路径，根据您的环境您可能需要将说明中所有命令行中的 python 替换为具体的 Python 路径

在 Windows 环境下，输出 Python 路径的命令为：

```
where python
```

在 MacOS/Linux 环境下，输出 Python 路径的命令为：

```
which python
```

4. 检查 Python 的版本

使用以下命令确认是 3.6/3.7/3.8/3.9/3.10

```
python --version
```

5. 检查 pip 的版本，确认是 20.2.2+

```
python -m ensurepip  
python -m pip --version
```

6. 确认 Python 和 pip 是 64 bit，并且处理器架构是 x86_64（或称作 x64、Intel 64、AMD64）架构，目前 PaddlePaddle 不支持 arm64 架构（mac M1 除外，paddle 已支持 Mac M1 芯片）。下面的第一行输出的是“64bit”，第二行输出的是“x86_64”、“x64”或“AMD64”即可：

```
python -c "import platform;print(platform.architecture()[0]);  
print(platform.machine())"
```

7. 如果您希望使用 pip 进行安装 PaddlePaddle 可以直接使用以下命令：

(1). CPU 版本：如果您只是想安装 CPU 版本请参考如下命令安装

安装 CPU 版本的命令为：

```
python -m pip install paddlepaddle==2.3.2 -i https://mirror.baidu.  
↪com/pypi/simple
```

或

```
python -m pip install paddlepaddle==2.3.2 -i https://pypi.tuna.  
↪tsinghua.edu.cn/simple
```

(2). GPU 版本: 如果您想使用 GPU 版本请参考如下命令安装

注意:

- 需要您确认您的 GPU 满足上方列出的要求

请注意用以下指令安装的 PaddlePaddle 在 Windows、Ubuntu、CentOS 下只支持 CUDA10.2:

```
python -m pip install paddlepaddle-gpu==2.3.2 -i https://mirror.  
↪baidu.com/pypi/simple
```

或

```
python -m pip install paddlepaddle-gpu==2.3.2 -i https://pypi.tuna.  
↪tsinghua.edu.cn/simple
```

请确认需要安装 PaddlePaddle 的 Python 是您预期的位置，因为您计算机可能有多个 Python。
根据您的环境您可能需要将说明中所有命令行中的 python 替换为具体的 Python 路径。

8. 验证安装

使用 python 进入 python 解释器，输入 import paddle，再输入 paddle.utils.run_check()。

如果出现 PaddlePaddle is installed successfully!，说明您已成功安装。

9. 更多帮助信息请参考:

[Linux 下的 PIP 安装](#)

[MacOS 下的 PIP 安装](#)

[Windows 下的 PIP 安装](#)

第二种安装方式: 使用源代码编译安装

- 如果您只是使用 PaddlePaddle，建议使用 **pip** 安装即可。
- 如果您有开发 PaddlePaddle 的需求，请参考: [从源码编译](#)

1.1.1 Pip 安装

Linux 下的 PIP 安装

一、环境准备

1.1 目前飞桨支持的环境

- Linux 版本 (64 bit)
 - CentOS 7 (GPU 版本支持 CUDA 10.1/10.2/11.1/11.2/11.6)
 - Ubuntu 16.04 (GPU 版本支持 CUDA 10.1/10.2/11.1/11.2/11.6)
 - Ubuntu 18.04 (GPU 版本支持 CUDA 10.1/10.2/11.1/11.2/11.6)
 - Ubuntu 20.04 (GPU 版本支持 CUDA 10.1/10.2/11.1/11.2/11.6)
- Python 版本 3.6/3.7/3.8/3.9/3.10 (64 bit)
- pip 或 pip3 版本 20.2.2 或更高版本 (64 bit)

1.2 如何查看您的环境

- 可以使用以下命令查看本机的操作系统和位数信息：

```
uname -m && cat /etc/*release
```

- 确认需要安装 PaddlePaddle 的 Python 是您预期的位置，因为您计算机可能有多个 Python
 - 根据您的环境您可能需要将说明中所有命令行中的 python 替换为具体的 Python 路径

```
which python
```

- 需要确认 python 的版本是否满足要求
 - 使用以下命令确认是 3.6/3.7/3.8/3.9/3.10

```
python --version
```

- 需要确认 pip 的版本是否满足要求，要求 pip 版本为 20.2.2 或更高版本

```
python -m ensurepip
```

```
python -m pip --version
```

- 需要确认 Python 和 pip 是 64bit，并且处理器架构是 x86_64（或称作 x64、Intel 64、AMD64）架构。下面的第一行输出的是“64bit”，第二行输出的是“x86_64”、“x64”或“AMD64”即可：

```
python -c "import platform;print(platform.architecture()[0]);print(platform.  
↳machine())"
```

- 默认提供的安装包需要计算机支持 MKL
- 如果您对机器环境不了解, 请下载使用[快速安装脚本](#), 配套说明请参考[这里](#)。

二、开始安装

本文档为您介绍 pip 安装方式

首先请选择您的版本

- 如果您的计算机没有 NVIDIA® GPU, 请安装 *CPU* 版的 *PaddlePaddle*
- 如果您的计算机有 NVIDIA® GPU, 请确保满足以下条件并且安装 *GPU* 版 *PaddlePaddle*
 - CUDA 工具包 10.1/10.2 配合 cuDNN 7 (cuDNN 版本 >=7.6.5, 如需多卡支持, 需配合 NCCL2.7 及更高)
 - CUDA 工具包 11.1 配合 cuDNN v8.1.1(如需多卡支持, 需配合 NCCL2.7 及更高)
 - CUDA 工具包 11.2 配合 cuDNN v8.1.1(如需多卡支持, 需配合 NCCL2.7 及更高)
 - CUDA 工具包 11.6 配合 cuDNN v8.4.0(如需多卡支持, 需配合 NCCL2.7 及更高)
 - GPU 运算能力超过 3.5 的硬件设备

您可参考 NVIDIA 官方文档了解 CUDA 和 CUDNN 的安装流程和配置方法, 请见[CUDA](#), [cuDNN](#)

- 如果您需要使用多卡环境请确保您已经正确安装 nccl2, 或者按照以下指令安装 nccl2 (这里提供的是 CUDA10.2, cuDNN7 下 nccl2 的安装指令, 更多版本的安装信息请参考 [NVIDIA官方网站](#)) :
 - Centos 系统可以参考以下命令

```
wget http://developer.download.nvidia.com/compute/machine-learning/repos/  
↳rhel7/x86_64/nvidia-machine-learning-repo-rhel7-1.0.0-1.x86_64.rpm
```

```
rpm -i nvidia-machine-learning-repo-rhel7-1.0.0-1.x86_64.rpm
```

```
yum update -y
```

```
yum install -y libnnccl-2.7.8-1+cuda10.2 libnnccl-devel-2.7.8-1+cuda10.2  
↳libnnccl-static-2.7.8-1+cuda10.2
```

- Ubuntu 系统可以参考以下命令

```
wget https://developer.download.nvidia.com/compute/machine-learning/repos/
↪ubuntu1604/x86_64/nvidia-machine-learning-repo-ubuntu1604_1.0.0-1_amd64.deb
```

```
dpkg -i nvidia-machine-learning-repo-ubuntu1604_1.0.0-1_amd64.deb
```

```
sudo apt install -y libncccl2=2.7.8-1+cuda10.2 libncccl-dev=2.7.8-1+cuda10.2
```

2.1 CPU 版的 PaddlePaddle

```
python -m pip install paddlepaddle==2.3.2 -i https://pypi.tuna.tsinghua.edu.cn/simple
```

2.2 GPU 版的 PaddlePaddle

2.2.1 CUDA10.1 的 PaddlePaddle

```
python -m pip install paddlepaddle-gpu==2.3.2.post101 -f https://www.paddlepaddle.org.
↪cn/wheel/linux/mkl/avx/stable.html
```

2.2.2 CUDA10.2 的 PaddlePaddle

```
python -m pip install paddlepaddle-gpu==2.3.2 -i https://pypi.tuna.tsinghua.edu.cn/
↪simple
```

2.2.3 CUDA11.1 的 PaddlePaddle

```
python -m pip install paddlepaddle-gpu==2.3.2.post111 -f https://www.paddlepaddle.org.
↪cn/wheel/linux/mkl/avx/stable.html
```

2.2.4 CUDA11.2 的 PaddlePaddle

```
python -m pip install paddlepaddle-gpu==2.3.2.post112 -f https://www.paddlepaddle.org.
↪cn/wheel/linux/mkl/avx/stable.html
```

2.2.5 CUDA11.6 的 PaddlePaddle

```
python -m pip install paddlepaddle-gpu==2.3.2.post116 -f https://www.paddlepaddle.org.
↪cn/wheel/linux/mkl/avx/stable.html
```

注：

- 如果你使用的是安培架构的 GPU，推荐使用 CUDA11 以上。如果你使用的是非安培架构的 GPU，推荐使用 CUDA10.2，性能更优。

- 请确认需要安装 PaddlePaddle 的 Python 是您预期的位置，因为您计算机可能有多个 Python。根据您的环境您可能需要将说明中所有命令行中的 `python` 替换为 `python3` 或者替换为具体的 Python 路径。
- 如果您需要使用清华源，可以通过以下命令

```
python -m pip install paddlepaddle-gpu==[版本号] -i https://pypi.tuna.tsinghua.edu.cn/simple
```

- 上述命令默认安装 `avx` 的包。如果你的机器不支持 `avx`，需要安装 `noavx` 的 Paddle 包，可以通过以下命令安装，仅支持 `python3.8`:

首先使用如下命令将 wheel 包下载到本地，再使用 `python -m pip install [name].whl` 本地安装（`[name]` 为 wheel 包名称）：

- `cpu、mkl` 版本 `noavx` 机器安装：

```
python -m pip download paddlepaddle==2.3.2 -f https://www.paddlepaddle.org.cn/whl/linux/mkl/noavx/stable.html --no-index --no-deps
```

- `cpu、openblas` 版本 `noavx` 机器安装：

```
python -m pip download paddlepaddle==2.3.2 -f https://www.paddlepaddle.org.cn/whl/linux/openblas/noavx/stable.html --no-index --no-deps
```

- `gpu` 版本 `cuda10.1 noavx` 机器安装：

```
python -m pip download paddlepaddle-gpu==2.3.2.post101 -f https://www.paddlepaddle.org.cn/whl/linux/mkl/noavx/stable.html --no-index --no-deps
```

- `gpu` 版本 `cuda10.2 noavx` 机器安装：

```
python -m pip download paddlepaddle-gpu==2.3.2 -f https://www.paddlepaddle.org.cn/whl/linux/mkl/noavx/stable.html --no-index --no-deps
```

判断你的机器是否支持 `avx`，可以输入以下命令，如果输出中包含 `avx`，则表示机器支持 `avx`

```
cat /proc/cpuinfo | grep -i avx
```

- 如果你想安装 `avx、openblas` 的 Paddle 包，可以通过以下命令将 wheel 包下载到本地，再使用 `python -m pip install [name].whl` 本地安装（`[name]` 为 wheel 包名称）：

```
python -m pip download paddlepaddle==2.3.2 -f https://www.paddlepaddle.org.cn/whl/linux/openblas/avx/stable.html --no-index --no-deps
```

- 如果你想安装联编 `tensorrt` 的 Paddle 包，可以参考[下载安装 Linux 预测库](#)。

三、验证安装

安装完成后您可以使用 `python` 或 `python3` 进入 `python` 解释器, 输入 `import paddle`, 再输入 `paddle.utils.run_check()`

如果出现 `PaddlePaddle is installed successfully!`, 说明您已成功安装。

四、如何卸载

请使用以下命令卸载 PaddlePaddle:

- **CPU 版本的 PaddlePaddle:** `python -m pip uninstall paddlepaddle`
- **GPU 版本的 PaddlePaddle:** `python -m pip uninstall paddlepaddle-gpu`

MacOS 下的 PIP 安装

一、环境准备

1.1 目前飞桨支持的环境

- macOS 版本 **10.x/11.x (64 bit)** (不支持 GPU 版本)
- mac 机器上支持 mac M1 芯片、Intel 芯片
- Python 版本 **3.6/3.7/3.8/3.9/3.10 (64 bit)**
- pip 或 pip3 版本 **20.2.2 或更高版本 (64 bit)**

1.2 如何查看您的环境

- 可以使用以下命令查看本机的操作系统和位数信息:

```
uname -m && cat /etc/*release
```

- 确认需要安装 PaddlePaddle 的 Python 是您预期的位置, 因为您计算机可能有多个 Python
 - 使用以下命令输出 Python 路径, 根据的环境您可能需要将说明中所有命令行中的 `python` 替换为具体的 Python 路径

```
which python
```

- 需要确认 `python` 的版本是否满足要求
 - 使用以下命令确认是 3.6/3.7/3.8/3.9/3.10

```
python --version
```

- 需要确认 pip 的版本是否满足要求，要求 pip 版本为 20.2.2 或更高版本

```
python -m ensurepip
```

```
python -m pip --version
```

- 需要确认 Python 和 pip 是 64bit，并且处理器架构是 x86_64（或称作 x64、Intel 64、AMD64）架构或 arm64 架构（paddle 已原生支持 Mac M1 芯片）：

```
python -c "import platform;print(platform.architecture()[0]);print(platform.  
↳machine())"
```

- 默认提供的安装包需要计算机支持 MKL
- 如果您对机器环境不了解，请下载使用[快速安装脚本](#)，配套说明请参考[这里](#)。

二、开始安装

本文档为您介绍 pip 安装方式

首先请选择您的版本

- 目前在 MacOS 环境仅支持 CPU 版 PaddlePaddle

根据版本进行安装

确定您的环境满足条件后可以开始安装了，选择下面您要安装的 PaddlePaddle

```
python -m pip install paddlepaddle==2.3.2 -i https://pypi.tuna.tsinghua.edu.cn/simple
```

注：

- MacOS 上您需要安装 unrar 以支持 PaddlePaddle，可以使用命令 brew install unrar
- 请确认需要安装 PaddlePaddle 的 Python 是您预期的位置，因为您计算机可能有多个 Python。根据您的环境您可能需要将说明中所有命令行中的 python 替换为具体的 Python 路径。
- 默认下载最新稳定版的安装包，如需获取 develop 版本 nightly build 的安装包，请参考[这里](#)
- 使用 MacOS 中自带 Python 可能会导致安装失败。请使用[python 官网](#)提供的 python3.6.x、python3.7.x、python3.8.x、python3.9.x、python3.10.x。

三、验证安装

安装完成后您可以使用 `python` 进入 `python` 解释器，输入 `import paddle`，再输入 `paddle.utils.run_check()`

如果出现 `PaddlePaddle is installed successfully!`，说明您已成功安装。

四、如何卸载

请使用以下命令卸载 PaddlePaddle：

- `python -m pip uninstall paddlepaddle`

Windows 下的 PIP 安装

一、环境准备

1.1 目前飞桨支持的环境

- Windows 7/8/10 专业版/企业版 (64bit)
- GPU 版本支持 CUDA 10.1/10.2/11.1/11.2/11.6，且仅支持单卡
- Python 版本 3.6+/3.7+/3.8+/3.9+/3.10+ (64 bit)
- pip 版本 20.2.2 或更高版本 (64 bit)

1.2 如何查看您的环境

- 需要确认 python 的版本是否满足要求
 - 使用以下命令确认是 3.6/3.7/3.8/3.9/3.10

```
python --version
```

- 需要确认 pip 的版本是否满足要求，要求 pip 版本为 20.2.2 或更高版本

```
python -m ensurepip
```

```
python -m pip --version
```

- 需要确认 Python 和 pip 是 64bit，并且处理器架构是 x86_64（或称作 x64、Intel 64、AMD64）架构。下面的第一行输出的是“64bit”，第二行输出的是“x86_64”、“x64”或“AMD64”即可：

```
python -c "import platform;print(platform.architecture()[0]);print(platform.  
machine())"
```

- 默认提供的安装包需要计算机支持 MKL
- Windows 暂不支持 NCCL，分布式等相关功能

二、开始安装

本文档为您介绍 pip 安装方式

首先请您选择您的版本

- 如果您的计算机没有 NVIDIA® GPU，请安装 CPU 版的 PaddlePaddle
- 如果您的计算机有 NVIDIA® GPU，请确保满足以下条件并且安装 GPU 版 PaddlePaddle
 - CUDA 工具包 10.1/10.2 配合 cuDNN v7.6.5
 - CUDA 工具包 11.1 配合 cuDNN v8.1.1
 - CUDA 工具包 11.2 配合 cuDNN v8.2.1
 - CUDA 工具包 11.6 配合 cuDNN v8.4.0
 - GPU 运算能力超过 3.5 的硬件设备
- 注：目前官方发布的 windows 安装包仅包含 CUDA 10.1/10.2/11.1/11.2/11.6，如需使用其他 cuda 版本，请通过源码自行编译。您可参考 NVIDIA 官方文档了解 CUDA 和 CUDNN 的安装流程和配置方法，请见 [CUDA](#), [cuDNN](#)

根据版本进行安装

确定您的环境满足条件后可以开始安装了，选择下面您要安装的 PaddlePaddle

2.1 CPU 版的 PaddlePaddle

```
python -m pip install paddlepaddle==2.3.2 -i https://pypi.tuna.tsinghua.edu.cn/simple
```

2.2 GPU 版的 PaddlePaddle

2.2.1 CUDA10.1 的 PaddlePaddle

```
python -m pip install paddlepaddle-gpu==2.3.2.post101 -f https://www.paddlepaddle.org.
˓→cn/wheel/windows/mkl/avx/stable.html
```

2.2.2 CUDA10.2 的 PaddlePaddle

```
python -m pip install paddlepaddle-gpu==2.3.2 -i https://pypi.tuna.tsinghua.edu.cn/
˓→simple
```

2.2.3 CUDA11.1 的 PaddlePaddle

```
python -m pip install paddlepaddle-gpu==2.3.2.post111 -f https://www.paddlepaddle.org.
˓→cn/wheel/windows/mkl/avx/stable.html
```

2.2.4 CUDA11.2 的 PaddlePaddle

```
python -m pip install paddlepaddle-gpu==2.3.2.post112 -f https://www.paddlepaddle.org.
˓→cn/wheel/windows/mkl/avx/stable.html
```

2.2.5 CUDA11.6 的 PaddlePaddle

```
python -m pip install paddlepaddle-gpu==2.3.2.post116 -f https://www.paddlepaddle.org.
˓→cn/wheel/windows/mkl/avx/stable.html
```

注:

- 如果你使用的是安培架构的 GPU，推荐使用 CUDA11 以上。如果你使用的是非安培架构的 GPU，推荐使用 CUDA10.2，性能更优。
- 请确认需要安装 PaddlePaddle 的 Python 是您预期的位置，因为您计算机可能有多个 Python。根据您的环境您可能需要将说明中所有命令行中的 `python` 替换为具体的 Python 路径。
- 上述命令默认安装 `avx` 的包。如果你的机器不支持 `avx`，需要安装 `noavx` 的 Paddle 包，可以通过以下命令安装，仅支持 `python3.8`:

首先使用如下命令将 wheel 包下载到本地，再使用 `python -m pip install [name].whl` 本地安装 (`[name]` 为 wheel 包名称):

– cpu、mkl 版本 noavx 机器安装:

```
python -m pip download paddlepaddle==2.3.2 -f https://www.paddlepaddle.org.cn/wheel/
˓→windows/mkl/noavx/stable.html --no-index --no-deps
```

– cpu、openblas 版本 noavx 机器安装:

```
python -m pip download paddlepaddle==2.3.2 -f https://www.paddlepaddle.org.cn/whl/  
windows/openblas/noavx/stable.html --no-index --no-deps
```

- gpu 版本 cuda10.1 noavx 机器安装:

```
python -m pip download paddlepaddle-gpu==2.3.2.post101 -f https://www.  
paddlepaddle.org.cn/whl/windows/mkl/noavx/stable.html --no-index --no-deps
```

- gpu 版本 cuda10.2 noavx 机器安装:

```
python -m pip download paddlepaddle-gpu==2.3.2 -f https://www.paddlepaddle.org.cn/  
whl/windows/mkl/noavx/stable.html --no-index --no-deps
```

判断你的机器是否支持 avx，可以安装[CPU-Z](#)工具查看“处理器-指令集”。

- 如果你想安装 avx、openblas 的 Paddle 包，可以通过以下命令将 wheel 包下载到本地，再使用 `python -m pip install [name].whl` 本地安装（[name] 为 wheel 包名称）：

```
python -m pip download paddlepaddle==2.3.2 -f https://www.paddlepaddle.org.cn/whl/  
windows/openblas/avx/stable.html --no-index --no-deps
```

- 如果你想安装联编 tensorrt 的 Paddle 包，可以参考[下载安装 Windows 预测库](#)。

三、验证安装

安装完成后您可以使用 `python` 进入 `python` 解释器，输入 `import paddle`，再输入 `paddle.utils.run_check()`

如果出现 `PaddlePaddle is installed successfully!`，说明您已成功安装。

四、如何卸载

请使用以下命令卸载 PaddlePaddle：

- CPU 版本的 PaddlePaddle:** `python -m pip uninstall paddlepaddle`
- GPU 版本的 PaddlePaddle:** `python -m pip uninstall paddlepaddle-gpu`

1.1.2 Conda 安装

Linux 下的 Conda 安装

Anaconda 是一个免费开源的 Python 和 R 语言的发行版本，用于计算科学，Anaconda 致力于简化包管理和部署。Anaconda 的包使用软件包管理系统 Conda 进行管理。Conda 是一个开源包管理系统和环境管理系统，可在 Windows、macOS 和 Linux 上运行。

一、环境准备

在进行 PaddlePaddle 安装之前请确保您的 Anaconda 软件环境已经正确安装。软件下载和安装参见 Anaconda 官网 (<https://www.anaconda.com/>)。在您已经正确安装 Anaconda 的情况下请按照下列步骤安装 PaddlePaddle。

- CentOS 7 / Ubuntu16.04 / Ubuntu18.04 / Ubuntu20.04 (64bit)
- GPU 版本支持 CUDA 10.1/10.2/11.2/11.6
- conda 版本 4.8.3+ (64 bit)

1.1 创建虚拟环境

1.1.1 安装环境

首先根据具体的 Python 版本创建 Anaconda 虚拟环境，PaddlePaddle 的 Anaconda 安装支持以下四种 Python 安装环境。

如果您想使用的 python 版本为 3.6:

```
conda create -n paddle_env python=3.6
```

如果您想使用的 python 版本为 3.7:

```
conda create -n paddle_env python=3.7
```

如果您想使用的 python 版本为 3.8:

```
conda create -n paddle_env python=3.8
```

如果您想使用的 python 版本为 3.9:

```
conda create -n paddle_env python=3.9
```

1.1.2 进入 Anaconda 虚拟环境

```
conda activate paddle_env
```

1.2 其他环境检查

1.2.1 确认 Python 安装路径

确认您的 conda 虚拟环境和需要安装 PaddlePaddle 的 Python 是您预期的位置，因为您计算机可能有多个 Python。进入 Anaconda 的命令行终端，输入以下指令确认 Python 位置。

输出 Python 路径的命令为：

```
which python
```

根据您的环境，您可能需要将说明中所有命令行中的 python 替换为具体的 Python 路径

1.2.2 检查 Python 版本

使用以下命令确认版本 (Python 应对应 3.6/3.7/3.8/3.9)

```
python --version
```

1.2.3 检查系统环境

确认 Python 和 pip 是 64bit，并且处理器架构是 x86_64（或称作 x64、Intel 64、AMD64）架构。下面的第一行输出的是”64bit”，第二行输出的是”x86_64（或 x64、AMD64）”即可：

```
python -c "import platform;print(platform.architecture()[0]);print(platform.machine())"  
↔"
```

二、开始安装

本文档为您介绍 conda 安装方式

添加清华源（可选）

对于国内用户无法连接到 Anaconda 官方源的可以按照以下命令添加清华源：

```
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free/
```

```
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main/
```

```
conda config --set show_channel_urls yes
```

根据版本进行安装

选择下面您要安装的 PaddlePaddle

CPU 版的 PaddlePaddle

如果您的计算机没有 NVIDIA® GPU，请安装 CPU 版的 PaddlePaddle

```
conda install paddlepaddle==2.3.2 --channel https://mirrors.tuna.tsinghua.edu.cn/
˓→anaconda/cloud/Paddle/
```

GPU 版的 PaddlePaddle

- 对于 CUDA 10.1，需要搭配 cuDNN 7 (cuDNN>=7.6.5, 多卡环境下 NCCL>=2.7)，安装命令为：

```
conda install paddlepaddle-gpu==2.3.2 cudatoolkit=10.1 --channel https://mirrors.tuna.
˓→tsinghua.edu.cn/anaconda/cloud/Paddle/
```

- 对于 CUDA 10.2，需要搭配 cuDNN 7 (cuDNN>=7.6.5, 多卡环境下 NCCL>=2.7)，安装命令为：

```
conda install paddlepaddle-gpu==2.3.2 cudatoolkit=10.2 --channel https://mirrors.tuna.
˓→tsinghua.edu.cn/anaconda/cloud/Paddle/
```

- 对于 CUDA 11.2，需要搭配 cuDNN 8.1.1(多卡环境下 NCCL>=2.7)，安装命令为：

```
conda install paddlepaddle-gpu==2.3.2 cudatoolkit=11.2 -c https://mirrors.tuna.
˓→tsinghua.edu.cn/anaconda/cloud/Paddle/ -c conda-forge
```

- 对于 CUDA 11.6，需要搭配 cuDNN 8.4.0(多卡环境下 NCCL>=2.7)，安装命令为：

```
conda install paddlepaddle-gpu==2.3.2 cudatoolkit=11.6 -c https://mirrors.tuna.
˓→tsinghua.edu.cn/anaconda/cloud/Paddle/ -c conda-forge
```

您可参考 NVIDIA 官方文档了解 CUDA 和 CUDNN 的安装流程和配置方法, 请见CUDA, cuDNN

三、验证安装

安装完成后您可以使用 `python` 或 `python3` 进入 `python` 解释器, 输入 `import paddle`, 再输入 `paddle.utils.run_check()`

如果出现 `PaddlePaddle is installed successfully!`, 说明您已成功安装。

MacOS 下的 Conda 安装

Anaconda是一个免费开源的 Python 和 R 语言的发行版本, 用于计算科学, Anaconda 致力于简化包管理和部署。Anaconda 的包使用软件包管理系统 Conda 进行管理。Conda 是一个开源包管理系统和环境管理系统, 可在 Windows、macOS 和 Linux 上运行。

一、环境准备

在进行 PaddlePaddle 安装之前请确保您的 Anaconda 软件环境已经正确安装。软件下载和安装参见 Anaconda 官网 (<https://www.anaconda.com/>)。在您已经正确安装 Anaconda 的情况下请按照下列步骤安装 PaddlePaddle。

- MacOS 版本 10.x/11.x (64 bit) (不支持 GPU 版本)
- conda 版本 4.8.3+ (64 bit)

1.1 创建虚拟环境

1.1.1 安装环境

首先根据具体的 Python 版本创建 Anaconda 虚拟环境, PaddlePaddle 的 Anaconda 安装支持以下五种 Python 安装环境。

如果您想使用的 `python` 版本为 3.6:

```
conda create -n paddle_env python=3.6
```

如果您想使用的 `python` 版本为 3.7:

```
conda create -n paddle_env python=3.7
```

如果您想使用的 `python` 版本为 3.8:

```
conda create -n paddle_env python=3.8
```

如果您想使用的 `python` 版本为 3.9:

```
conda create -n paddle_env python=3.9
```

1.1.2 进入 Anaconda 虚拟环境

```
conda activate paddle_env
```

1.2 其他环境检查

1.2.1 确认 Python 安装路径

确认您的 conda 虚拟环境和需要安装 PaddlePaddle 的 Python 是您预期的位置，因为您计算机可能有多个 Python。进入 Anaconda 的命令行终端，输入以下指令确认 Python 位置。

输出 Python 路径的命令为：

```
which python
```

根据您的环境，您可能需要将说明中所有命令行中的 python 替换为具体的 Python 路径

1.2.2 检查 Python 版本

使用以下命令确认版本 (Python 应对应 3.6/3.7/3.8/3.9)

```
python --version
```

1.2.3 检查系统环境

确认 Python 和 pip 是 64bit，并且处理器架构是 x86_64（或称作 x64、Intel 64、AMD64）架构或 arm64 架构 (paddle 已原生支持 Mac M1 芯片)：

```
python -c "import platform;print(platform.architecture()[0]);print(platform.machine())\n" ↵"
```

二、开始安装

本文档为您介绍 conda 安装方式

添加清华源（可选）

- 对于国内用户无法连接到 Anaconda 官方源的可以按照以下命令添加清华源:

```
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/  
→free/
```

```
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/  
→main/
```

```
conda config --set show_channel_urls yes
```

安装 CPU 版 PaddlePaddle

- 目前在 MacOS 环境仅支持 CPU 版 PaddlePaddle, 请参考如下命令安装 Paddle:

```
conda install paddlepaddle==2.3.2 --channel https://mirrors.tuna.tsinghua.edu.cn/  
→anaconda/cloud/Paddle/
```

三、验证安装

安装完成后您可以使用 `python` 或 `python3` 进入 `python` 解释器, 输入 `import paddle`, 再输入 `paddle.utils.run_check()`

如果出现 `PaddlePaddle is installed successfully!`, 说明您已成功安装。

Windows 下的 Conda 安装

Anaconda 是一个免费开源的 Python 和 R 语言的发行版本, 用于计算科学, Anaconda 致力于简化包管理和部署。Anaconda 的包使用软件包管理系统 Conda 进行管理。Conda 是一个开源包管理系统和环境管理系统, 可在 Windows、macOS 和 Linux 上运行。

一、环境准备

在进行 PaddlePaddle 安装之前请确保您的 Anaconda 软件环境已经正确安装。软件下载和安装参见 Anaconda 官网 (<https://www.anaconda.com/>)。在您已经正确安装 Anaconda 的情况下请按照下列步骤安装 PaddlePaddle。

- Windows 7/8/10 专业版/企业版 (64bit)
- GPU 版本支持 CUDA 10.1/10.2/11.2/11.6，且仅支持单卡
- conda 版本 4.8.3+ (64 bit)

1.1 创建虚拟环境

1.1.1 安装环境

首先根据具体的 Python 版本创建 Anaconda 虚拟环境，PaddlePaddle 的 Anaconda 安装支持以下四种 Python 安装环境。

如果您想使用的 python 版本为 3.6:

```
conda create -n paddle_env python=3.6
```

如果您想使用的 python 版本为 3.7:

```
conda create -n paddle_env python=3.7
```

如果您想使用的 python 版本为 3.8:

```
conda create -n paddle_env python=3.8
```

如果您想使用的 python 版本为 3.9:

```
conda create -n paddle_env python=3.9
```

1.1.2 进入 Anaconda 虚拟环境

```
activate paddle_env
```

1.2 其他环境检查

1.2.1 确认 Python 安装路径

确认您的 conda 虚拟环境和需要安装 PaddlePaddle 的 Python 是您预期的位置，因为您计算机可能有多个 Python。进入 Anaconda 的命令行终端，输入以下指令确认 Python 位置。

输出 Python 路径的命令为：

```
where python
```

根据您的环境，您可能需要将说明中所有命令行中的 python 替换为具体的 Python 路径

1.2.2 检查 Python 版本

使用以下命令确认版本(应对应 3.6/3.7/3.8/3.9)

```
python --version
```

1.2.3 检查系统环境

确认 Python 和 pip 是 64bit，并且处理器架构是 x86_64（或称作 x64、Intel 64、AMD64）架构。下面的第一行输出的是”64bit”，第二行输出的是”x86_64（或 x64、AMD64）”即可：

```
python -c "import platform;print(platform.architecture()[0]);print(platform.machine())  
↔"
```

二、开始安装

本文档为您介绍 conda 安装方式

添加清华源（可选）

对于国内用户无法连接到 Anaconda 官方源的可以按照以下命令添加清华源：

```
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free/
```

```
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main/
```

```
conda config --set show_channel_urls yes
```

根据版本进行安装

选择下面您要安装的 PaddlePaddle

CPU 版的 PaddlePaddle

如果您的计算机没有 NVIDIA® GPU, 请安装 CPU 版的 PaddlePaddle

```
conda install paddlepaddle==2.3.2 --channel https://mirrors.tuna.tsinghua.edu.cn/
˓→anaconda/cloud/Paddle/
```

GPU 版的 PaddlePaddle

- 对于 CUDA 10.1, 需要搭配 cuDNN 7 (cuDNN>=7.6.5), 安装命令为:

```
conda install paddlepaddle-gpu==2.3.2 cudatoolkit=10.1 --channel https://mirrors.tuna.
˓→tsinghua.edu.cn/anaconda/cloud/Paddle/
```

- 对于 CUDA 10.2, 需要搭配 cuDNN 7 (cuDNN>=7.6.5), 安装命令为:

```
conda install paddlepaddle-gpu==2.3.2 cudatoolkit=10.2 --channel https://mirrors.tuna.
˓→tsinghua.edu.cn/anaconda/cloud/Paddle/
```

- 对于 CUDA 11.2, 需要搭配 cuDNN 8.1.1, 安装命令为:

```
conda install paddlepaddle-gpu==2.3.2 cudatoolkit=11.2 -c https://mirrors.tuna.
˓→tsinghua.edu.cn/anaconda/cloud/Paddle/ -c conda-forge
```

- 对于 CUDA 11.6, 需要搭配 cuDNN 8.4.0, 安装命令为:

```
conda install paddlepaddle-gpu==2.3.2 cudatoolkit=11.6 -c https://mirrors.tuna.
˓→tsinghua.edu.cn/anaconda/cloud/Paddle/ -c conda-forge
```

您可参考 NVIDIA 官方文档了解 CUDA 和 CUDNN 的安装流程和配置方法, 请见CUDA, cuDNN

三、验证安装

安装完成后您可以使用 python 或 python3 进入 python 解释器, 输入 import paddle, 再输入 paddle.utils.run_check()

如果出现 PaddlePaddle is installed successfully!, 说明您已成功安装。

1.1.3 Docker 安装

Linux 下的 Docker 安装

Docker是一个开源的应用容器引擎。使用 Docker，既可以将 PaddlePaddle 的安装 & 使用与系统环境隔离，也可以与主机共享 GPU、网络等资源。以下 Docker 安装与使用流程中，docker 里已经安装好了特定版本的 PaddlePaddle。

环境准备

- 目前支持的系统类型，请见[安装说明](#)，请注意目前暂不支持在 CentOS 6 使用 Docker
- 在本地主机上[安装 Docker](#)
- 如需在 Linux 开启 GPU 支持，请[安装 nvidia-docker](#)
- 镜像中 Python 版本为 3.7

安装步骤

1. 拉取 PaddlePaddle 镜像

对于国内用户，因为网络问题下载 docker 比较慢时，可使用百度提供的镜像：

- CPU 版的 PaddlePaddle：

```
docker pull registry.baidubce.com/paddlepaddle/paddle:2.3.2
```

- CPU 版的 PaddlePaddle，且镜像中预装好了 jupyter：

```
docker pull registry.baidubce.com/paddlepaddle/paddle:2.3.2-jupyter
```

- GPU 版的 PaddlePaddle：

```
nvidia-docker pull registry.baidubce.com/paddlepaddle/paddle:2.3.2-gpu-cuda10.2-  
˓→cudnn7
```

```
nvidia-docker pull registry.baidubce.com/paddlepaddle/paddle:2.3.2-gpu-cuda11.2-  
˓→cudnn8
```

如果您的机器不在中国大陆地区，可以直接从 DockerHub 拉取镜像：

- CPU 版的 PaddlePaddle：

```
docker pull paddlepaddle/paddle:2.3.2
```

- CPU 版的 PaddlePaddle，且镜像中预装好了 jupyter:

```
docker pull paddlepaddle/paddle:2.3.2-jupyter
```

- GPU 版的 PaddlePaddle:

```
nvidia-docker pull paddlepaddle/paddle:2.3.2-gpu-cuda10.2-cudnn7
```

```
nvidia-docker pull paddlepaddle/paddle:2.3.2-gpu-cuda11.2-cudnn8
```

您还可以访问[DockerHub](#)获取更多镜像。

2. 构建并进入 docker 容器

- 使用 CPU 版本的 PaddlePaddle:

```
docker run --name paddle_docker -it -v $PWD:/paddle registry.baidubce.com/
˓→paddlepaddle/paddle:2.3.2 /bin/bash
```

- --name paddle_docker: 设定 Docker 的名称, paddle_docker 是自己设置的名称;
- -it: 参数说明容器已和本机交互式运行;
- -v \$PWD:/paddle: 指定将当前路径 (PWD 变量会展开为当前路径的绝对路径) 挂载到容器内部的 /paddle 目录;
- registry.baidubce.com/paddlepaddle/paddle:2.3.2: 指定需要使用的 image 名称, 您可以通过 docker images 命令查看; /bin/bash 是在 Docker 中要执行的命令

- 使用 CPU 版本的 PaddlePaddle，且镜像中预装好了 jupyter:

```
mkdir ./jupyter_docker
```

```
chmod 777 ./jupyter_docker
```

```
cd ./jupyter_docker
```

```
docker run -p 80:80 --rm --env USER_PASSWD="password you set" -v $PWD:/home/
˓→paddle registry.baidubce.com/paddlepaddle/paddle:2.3.2-jupyter
```

- --rm: 关闭容器后删除容器;
- --env USER_PASSWD="password you set": 为 jupyter 设置登录密码, password you set 是自己设置的密码;
- -v \$PWD:/home/paddle: 指定将当前路径 (PWD 变量会展开为当前路径的绝对路径) 挂载到容器内部的 /home/paddle 目录;

- `registry.baidubce.com/paddlepaddle/paddle:2.3.2-jupyter`: 指定需要使用的 image 名称，您可以通过 `docker images` 命令查看
- 使用 GPU 版本的 PaddlePaddle:

```
nvidia-docker run --name paddle_docker -it -v $PWD:/paddle registry.baidubce.com/  
→paddlepaddle/paddle:2.3.2-gpu-cuda10.2-cudnn7 /bin/bash
```

- `--name paddle_docker`: 设定 Docker 的名称，`paddle_docker` 是自己设置的名称；
- `-it`: 参数说明容器已和本机交互式运行；
- `-v $PWD:/paddle`: 指定将当前路径（PWD 变量会展开为当前路径的绝对路径）挂载到容器内部的 `/paddle` 目录；
- `registry.baidubce.com/paddlepaddle/paddle:2.3.2-gpu-cuda10.2-cudnn7`: 指定需要使用的 image 名称，如果您希望使用 CUDA 11.2 的镜像，也可以将其替换成 `registry.baidubce.com/paddlepaddle/paddle:2.3.2-gpu-cuda11.2-cudnn8`。您可以通过 `docker images` 命令查看镜像。`/bin/bash` 是在 Docker 中要执行的命令

至此，您已经成功使用 Docker 安装 PaddlePaddle，更多 Docker 使用请参见[Docker 官方文档](#)

镜像简介

您可以在 [DockerHub](#) 中找到 PaddlePaddle 的各个发行的版本的 docker 镜像。

补充说明

- 当您需要第二次进入 Docker 容器中，使用如下命令：

启动之前创建的容器

```
docker start <Name of container>
```

进入启动的容器

```
docker attach <Name of container>
```

- 如您是 Docker 新手，您可以参考互联网上的资料学习，例如[Docker 教程](#)

如何卸载

请您进入 Docker 容器后，执行如下命令

- CPU 版本的 PaddlePaddle:

```
pip uninstall paddlepaddle
```

- GPU 版本的 PaddlePaddle:

```
pip uninstall paddlepaddle-gpu
```

或通过 `docker rm <Name of container>` 来直接删除 Docker 容器

MacOS 下的 Docker 安装

Docker 是一个开源的应用容器引擎。使用 Docker，既可以将 PaddlePaddle 的安装 & 使用与系统环境隔离，也可以与主机共享 GPU、网络等资源。以下 Docker 安装与使用流程中，docker 里已经安装好了特定版本的 PaddlePaddle。

环境准备

- MacOS 版本 10.x/11.x (64 bit) (不支持 GPU 版本)
- 在本地主机上[安装 Docker](#)
- 镜像中 Python 版本为 3.7

安装步骤

1. 拉取 PaddlePaddle 镜像

对于国内用户，因为网络问题下载 docker 比较慢时，可使用百度提供的镜像：

- CPU 版的 PaddlePaddle:

```
docker pull registry.baidubce.com/paddlepaddle/paddle:2.3.2
```

- CPU 版的 PaddlePaddle，且镜像中预装好了 jupyter:

```
docker pull registry.baidubce.com/paddlepaddle/paddle:2.3.2-jupyter
```

如果您的机器不在中国大陆地区，可以直接从 DockerHub 拉取镜像：

- CPU 版的 PaddlePaddle:

```
docker pull paddlepaddle/paddle:2.3.2
```

- CPU 版的 PaddlePaddle，且镜像中预装好了 jupyter:

```
docker pull paddlepaddle/paddle:2.3.2-jupyter
```

您还可以访问[DockerHub](#)获取更多镜像。

2. 构建并进入 docker 容器

- 使用 CPU 版本的 PaddlePaddle:

```
docker run --name paddle_docker -it -v $PWD:/paddle registry.baidubce.com/  
→paddlepaddle/paddle:2.3.2 /bin/bash
```

- --name paddle_docker: 设定 Docker 的名称, paddle_docker 是自己设置的名称;
- -it: 参数说明容器已和本机交互式运行;
- -v \$PWD:/paddle: 指定将当前路径 (PWD 变量会展开为当前路径的绝对路径) 挂载到容器内部的 /paddle 目录;
- registry.baidubce.com/paddlepaddle/paddle:2.3.2: 指定需要使用的 image 名称, 您可以通过 docker images 命令查看; /bin/bash 是在 Docker 中要执行的命令

- 使用 CPU 版本的 PaddlePaddle，且镜像中预装好了 jupyter:

```
mkdir ./jupyter_docker
```

```
chmod 777 ./jupyter_docker
```

```
cd ./jupyter_docker
```

```
docker run -p 80:80 --rm --env USER_PASSWD="password you set" -v $PWD:/home/  
→paddle registry.baidubce.com/paddlepaddle/paddle:2.3.2-jupyter
```

- --rm: 关闭容器后删除容器;
- --env USER_PASSWD="password you set": 为 jupyter 设置登录密码, password you set 是自己设置的密码;
- -v \$PWD:/home/paddle: 指定将当前路径 (PWD 变量会展开为当前路径的绝对路径) 挂载到容器内部的 /home/paddle 目录;
- registry.baidubce.com/paddlepaddle/paddle:2.3.2-jupyter: 指定需要使用的 image 名称, 您可以通过 docker images 命令查看

至此，您已经成功使用 Docker 安装 PaddlePaddle，更多 Docker 使用请参见[Docker 官方文档](#)

镜像简介

您可以在 [DockerHub](#) 中找到 PaddlePaddle 的各个发行的版本的 docker 镜像。

补充说明

- 当您需要第二次进入 Docker 容器中，使用如下命令：

启动之前创建的容器

```
docker start <Name of container>
```

进入启动的容器

```
docker attach <Name of container>
```

- 如您是 Docker 新手，您可以参考互联网上的资料学习，例如[Docker 教程](#)

如何卸载

请您进入 Docker 容器后，执行如下命令

- CPU 版本的 PaddlePaddle：

```
pip uninstall paddlepaddle
```

或通过 `docker rm <Name of container>` 来直接删除 Docker 容器

1.1.4 从源码编译

Linux 下从源码编译

环境准备

- Linux 版本 (64 bit)
 - CentOS 6 (不推荐，不提供编译出现问题时的官方支持)
 - CentOS 7 (GPU 版本支持 CUDA 10.1/10.2/11.1/11.2/11.6)
 - Ubuntu 14.04 (不推荐，不提供编译出现问题时的官方支持)

- Ubuntu 16.04 (GPU 版本支持 CUDA 10.1/10.2/11.1/11.2/11.6)
- Ubuntu 18.04 (GPU 版本支持 CUDA 10.1/10.2/11.1/11.2/11.6)
- Python 版本 3.6/3.7/3.8/3.9/3.10 (64 bit)

选择 CPU/GPU

- 如果您的计算机没有 NVIDIA® GPU, 请安装 CPU 版本的 PaddlePaddle
- 如果您的计算机有 NVIDIA® GPU, 请确保满足以下条件以编译 GPU 版 PaddlePaddle
 - CUDA 工具包 10.1/10.2 配合 cuDNN 7 (cuDNN 版本 $\geq 7.6.5$, 如需多卡支持, 需配合 NCCL2.7 及更高)
 - CUDA 工具包 11.1 配合 cuDNN v8.1.1(如需多卡支持, 需配合 NCCL2.7 及更高)
 - CUDA 工具包 11.2 配合 cuDNN v8.1.1(如需多卡支持, 需配合 NCCL2.7 及更高)
 - CUDA 工具包 11.6 配合 cuDNN v8.4.0(如需多卡支持, 需配合 NCCL2.7 及更高)
 - GPU 运算能力超过 3.5 的硬件设备

您可参考 NVIDIA 官方文档了解 CUDA 和 CUDNN 的安装流程和配置方法, 请见[CUDA, cuDNN](#)

安装步骤

在 Linux 的系统下有 2 种编译方式, 推荐使用 Docker 编译。Docker 环境中已预装好编译 Paddle 需要的各种依赖, 相较本机编译环境更简单。

- 使用 Docker 编译 (不提供在 CentOS 6 下编译中遇到问题的支持)
- 本机编译 (不提供在 CentOS 6 下编译中遇到问题的支持)

使用 Docker 编译

Docker 是一个开源的应用容器引擎。使用 Docker, 既可以将 PaddlePaddle 的安装 & 使用与系统环境隔离, 也可以与主机共享 GPU、网络等资源

使用 Docker 编译 PaddlePaddle, 您需要:

- 在本地主机上[安装 Docker](#)
- 如需在 Linux 开启 GPU 支持, 请[安装 nvidia-docker](#)

请您按照以下步骤安装:

1. 请首先选择您希望储存 PaddlePaddle 的路径，然后在该路径下使用以下命令将 PaddlePaddle 的源码从 github 克隆到本地当前目录下名为 Paddle 的文件夹中：

```
git clone https://github.com/PaddlePaddle/Paddle.git
```

2. 进入 Paddle 目录下：

```
cd Paddle
```

3. 拉取 PaddlePaddle 镜像

对于国内用户，因为网络问题下载 docker 比较慢时，可使用百度提供的镜像：

- CPU 版的 PaddlePaddle：

```
docker pull registry.baidubce.com/paddlepaddle/paddle:latest-dev
```

- GPU 版的 PaddlePaddle：

```
nvidia-docker pull registry.baidubce.com/paddlepaddle/paddle:latest-gpu-cuda10.2-  
→cudnn7-dev
```

如果您的机器不在中国大陆地区，可以直接从 DockerHub 拉取镜像：

- CPU 版的 PaddlePaddle：

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

- GPU 版的 PaddlePaddle：

```
nvidia-docker pull paddlepaddle/paddle:latest-gpu-cuda10.2-cudnn7-dev
```

上例中，latest-gpu-cuda10.2-cudnn7-dev 仅作示意用，表示安装 GPU 版的镜像。如果您还想安装其他 cuda/cudnn 版本的镜像，可以将其替换成 latest-dev-cuda11.2-cudnn8-gcc82、latest-gpu-cuda10.1-cudnn7-gcc82-dev、latest-gpu-cuda10.1-cudnn7-gcc54-dev 等。您可以访问[DockerHub](#)获取与您机器适配的镜像。

4. 创建并进入已配置好编译环境的 Docker 容器:

- 编译 CPU 版本的 PaddlePaddle:

```
docker run --name paddle-test -v $PWD:/paddle --network=host -it registry.  
→baidubce.com/paddlepaddle/paddle:latest-dev /bin/bash
```

- --name paddle-test: 为您创建的 Docker 容器命名为 paddle-test;
- -v \$PWD:/paddle: 将当前目录挂载到 Docker 容器中的/paddle 目录下 (Linux 中 PWD 变量会展开为当前路径的绝对路径);
- -it: 与宿主机保持交互状态;
- registry.baidubce.com/paddlepaddle/paddle:latest-dev: 使用名为 registry.baidubce.com/paddlepaddle/paddle:latest-dev 的镜像创建 Docker 容器, /bin/bash 进入容器后启动/bin/bash 命令。

- 编译 GPU 版本的 PaddlePaddle:

```
nvidia-docker run --name paddle-test -v $PWD:/paddle --network=host -it registry.  
→baidubce.com/paddlepaddle/paddle:latest-gpu-cuda10.2-cudnn7-dev /bin/bash
```

- --name paddle-test: 为您创建的 Docker 容器命名为 paddle-test;
- -v \$PWD:/paddle: 将当前目录挂载到 Docker 容器中的/paddle 目录下 (Linux 中 PWD 变量会展开为当前路径的绝对路径);
- -it: 与宿主机保持交互状态;
- registry.baidubce.com/paddlepaddle/paddle:latest-gpu-cuda10.2-cudnn7-dev: 使用名为 registry.baidubce.com/paddlepaddle/paddle:latest-gpu-cuda10.2-cudnn7-dev 的镜像创建 Docker 容器, /bin/bash 进入容器后启动/bin/bash 命令。

注意: 请确保至少为 docker 分配 4g 以上的内存, 否则编译过程可能因内存不足导致失败。

5. 进入 Docker 后进入 paddle 目录下:

```
cd /paddle
```

6. 切换到较稳定版本下进行编译:

```
git checkout [分支名]
```

例如：

```
git checkout release/2.3
```

注意：python3.6、python3.7 版本从 release/1.2 分支开始支持，python3.8 版本从 release/1.8 分支开始支持，python3.9 版本从 release/2.1 分支开始支持，python3.10 版本从 release/2.3 分支开始支持

7. 创建并进入/paddle/build 路径下：

```
mkdir -p /paddle/build && cd /paddle/build
```

8. 使用以下命令安装相关依赖：

- 安装 protobuf。

```
pip3.7 install protobuf
```

注意：以上用 Python3.7 命令来举例，如您的 Python 版本为 3.6/3.8/3.9/3.10，请将上述命令中的 pip3.7 改成 pip3.6/pip3.8/pip3.9/pip3.10

- 安装 patchelf，PatchELF 是一个小而实用的程序，用于修改 ELF 可执行文件的动态链接器和 RPATH。

```
apt install patchelf
```

9. 执行 cmake：

- 对于需要编译 CPU 版本 PaddlePaddle 的用户：

```
cmake .. -DPY_VERSION=3.7 -DWITH_GPU=OFF
```

- 对于需要编译 GPU 版本 PaddlePaddle 的用户：

```
cmake .. -DPY_VERSION=3.7 -DWITH_GPU=ON
```

- 具体编译选项含义请参见编译选项表
- 请注意修改参数-DPY_VERSION 为您希望编译使用的 python 版本，例如-DPY_VERSION=3.7 表示 python 版本为 3.7
- 我们目前不支持 CentOS 6 下使用 Docker 编译 GPU 版本的 PaddlePaddle

10. 执行编译:

使用多核编译

```
make -j$(nproc)
```

注意：编译过程中需要从 `github` 上下载依赖，请确保您的编译环境能正常从 `github` 下载代码。

11. 编译成功后进入`/paddle/build/python/dist` 目录下找到生成的`.whl` 包：

```
cd /paddle/build/python/dist
```

12. 在当前机器或目标机器安装编译好的`.whl` 包：

For Python3:

```
pip3.7 install -U [whl包的名字]
```

注意：以上用 Python3.7 命令来举例，如您的 Python 版本为 3.6/3.8/3.9/3.10，请将上述命令中的 pip3.7 改成 pip3.6/pip3.8/pip3.9/pip3.10。

恭喜，至此您已完成 PaddlePaddle 的编译安装。您只需要进入 Docker 容器后运行 PaddlePaddle，即可开始使用。更多 Docker 使用请参见 Docker 官方文档

本机编译

1. 检查您的计算机和操作系统是否符合我们支持的编译标准：

```
uname -m && cat /etc/*release
```

2. 更新系统源

- Centos 环境

更新 yum 的源：

```
yum update
```

并添加必要的 yum 源：

```
yum install -y epel-release
```

- Ubuntu 环境

更新 apt 的源：

```
apt update
```

3. 安装 NCCL (可选)

- 如果您需要使用 GPU 多卡，请确保您已经正确安装 nccl2，或者按照以下指令安装 nccl2（这里提供的 是 CUDA10.2, cuDNN7 下 nccl2 的安装指令，更多版本的安装信息请参考 NVIDIA[官方网站](#)）：

- Centos 系统可以参考以下命令

```
wget http://developer.download.nvidia.com/compute/machine-learning/repos/  
→rhel7/x86_64/nvidia-machine-learning-repo-rhel7-1.0.0-1.x86_64.rpm
```

```
rpm -i nvidia-machine-learning-repo-rhel7-1.0.0-1.x86_64.rpm
```

```
yum update -y
```

```
yum install -y libncc1-2.7.8-1+cuda10.2 libncc1-devel-2.7.8-1+cuda10.2  
→libncc1-static-2.7.8-1+cuda10.2
```

- Ubuntu 系统可以参考以下命令

```
wget https://developer.download.nvidia.com/compute/machine-learning/repos/  
→ubuntu1604/x86_64/nvidia-machine-learning-repo-ubuntu1604_1.0.0-1_amd64.deb
```

```
dpkg -i nvidia-machine-learning-repo-ubuntu1604_1.0.0-1_amd64.deb
```

```
sudo apt install -y libncc1=2.7.8-1+cuda10.2 libncc1-dev=2.7.8-1+cuda10.2
```

4. 安装必要的工具

- Centos 环境

bzip2 以及 make:

```
yum install -y bzip2
```

```
yum install -y make
```

cmake 需要 3.15 以上, 建议使用 3.16.0:

```
wget -q https://cmake.org/files/v3.16/cmake-3.16.0-Linux-x86_64.tar.gz
```

```
tar -zxvf cmake-3.16.0-Linux-x86_64.tar.gz
```

```
rm cmake-3.16.0-Linux-x86_64.tar.gz
```

```
PATH=/home/cmake-3.16.0-Linux-x86_64/bin:$PATH
```

gcc 需要 5.4 以上, 建议使用 8.2.0:

```
wget -q https://paddle-docker-tar.bj.bcebos.com/home/users/tianshuo/bce-python-  
↳ sdk-0.8.27/gcc-8.2.0.tar.xz && \  
tar -xvf gcc-8.2.0.tar.xz && \  
cd gcc-8.2.0 && \  
sed -i 's#ftp://gcc.gnu.org/pub/gcc/infrastructure/#https://paddle-ci.gz.bcebos.  
↳ com/#g' ./contrib/download_prerequisites && \  
unset LIBRARY_PATH CPATH C_INCLUDE_PATH PKG_CONFIG_PATH CPLUS_INCLUDE_PATH  
↳ INCLUDE && \  
../contrib/download_prerequisites && \  
cd .. && mkdir temp_gcc82 && cd temp_gcc82 && \  
..../gcc-8.2.0/configure --prefix=/usr/local/gcc-8.2 --enable-threads=posix --  
↳ disable-checking --disable-multilib && \  
make -j8 && make install
```

- Ubuntu 环境

bzip2 以及 make:

```
apt install -y bzip2
```

```
apt install -y make
```

cmake 需要 3.15 以上, 建议使用 3.16.0:

```
wget -q https://cmake.org/files/v3.16/cmake-3.16.0-Linux-x86_64.tar.gz
```

```
tar -zxvf cmake-3.16.0-Linux-x86_64.tar.gz
```

```
rm cmake-3.16.0-Linux-x86_64.tar.gz
```

```
PATH=/home/cmake-3.16.0-Linux-x86_64/bin:$PATH
```

gcc 需要 5.4 以上, 建议使用 8.2.0:

```
wget -q https://paddle-docker-tar.bj.bcebos.com/home/users/tianshuo/bce-python-
˓→sdk-0.8.27/gcc-8.2.0.tar.xz && \
tar -xvf gcc-8.2.0.tar.xz && \
cd gcc-8.2.0 && \
sed -i 's#ftp://gcc.gnu.org/pub/gcc/infrastructure/#https://paddle-ci.gz.bcebos.
˓→com/#g' ./contrib/download_prerequisites && \
unset LIBRARY_PATH CPATH C_INCLUDE_PATH PKG_CONFIG_PATH CPLUS_INCLUDE_PATH_
˓→INCLUDE && \
./contrib/download_prerequisites && \
cd .. && mkdir temp_gcc82 && cd temp_gcc82 && \
..../gcc-8.2.0/configure --prefix=/usr/local/gcc-8.2 --enable-threads=posix --
˓→disable-checking --disable-multilib && \
make -j8 && make install
```

5. 我们支持使用 virtualenv 进行编译安装, 首先请使用以下命令创建一个名为 paddle-venv 的虚环境:

- a. 安装 Python-dev:

(请参照 Python 官方流程安装)

- b. 安装 pip:

(请参照 Python 官方流程安装, 并保证拥有 20.2.2 及以上的 pip3 版本, 请注意, python3.6 及以上版本环境下, pip3 并不一定对应 python 版本, 如 python3.7 下默认只有 pip3.7)

- c. (Only For Python3) 设置 Python3 相关的环境变量, 这里以 python3.7 版本示例, 请替换成您使用的版本 (3.6、3.8、3.9、3.10):

1. 首先使用

```
find `dirname $(dirname $(which python3))` -name "libpython3.so"
```

找到 Python lib 的路径, 如果是 3.6、3.7、3.8、3.9、3.10, 请将 python3 改成 python3.6、python3.7、python3.8、python3.9、python3.10, 然后将下面 [python-lib-path] 替换为找到文件路径

2. 设置 PYTHON_LIBRARIES:

```
export PYTHON_LIBRARY=[python-lib-path]
```

3. 其次使用

```
find `dirname $(dirname $(which python3))`/include -name "python3.7m"
```

找到 Python Include 的路径, 请注意 python 版本, 然后将下面 [python-include-path] 替换为找到文件路径

4. 设置 PYTHON_INCLUDE_DIR:

```
export PYTHON_INCLUDE_DIRS=[python-include-path]
```

5. 设置系统环境变量路径:

```
export PATH=[python-lib-path]:$PATH
```

(这里将 [python-lib-path] 的最后两级目录替换为 /bin/)

- d. 安装虚环境 virtualenv 以及 virtualenvwrapper 并创建名为 paddle-venv 的虚环境: (请注意对应 python 版本的 pip3 的命令, 如 pip3.6、pip3.7、pip3.8、pip3.9、pip3.10)

1. 安装 virtualenv

```
pip install virtualenv
```

或

```
pip3 install virtualenv
```

2. 安装 virtualenvwrapper

```
pip install virtualenvwrapper
```

或

```
pip3 install virtualenvwrapper
```

3. 找到 virtualenvwrapper.sh:

```
find / -name virtualenvwrapper.sh
```

(请找到对应 Python 版本的 virtualenvwrapper.sh)

4. 查看 virtualenvwrapper.sh 中的安装方法:

```
cat virtualenvwrapper.sh
```

该 shell 文件中描述了步骤及命令

5. 按照 virtualenvwrapper.sh 中的描述, 安装 virtualwrapper

6. 设置 VIRTUALENVWRAPPER_PYTHON:

```
export VIRTUALENVWRAPPER_PYTHON=[python-lib-path]:$PATH
```

(这里将 [python-lib-path] 的最后两级目录替换为 /bin/)

7. 创建名为 paddle-venv 的虚环境:

```
mkvirtualenv paddle-venv
```

6. 进入虚环境:

```
workon paddle-venv
```

7. 执行编译前请您确认在虚环境中安装有编译依赖表中提到的相关依赖:

- 这里特别提供 patchELF 的安装方法, 其他的依赖可以使用 yum install 或者 pip install/pip3 install 后跟依赖名称和版本安装:

```
yum install patchelf
```

不能使用 yum 安装的用户请参见 patchElf [github](#)官方文档

8. 将 PaddlePaddle 的源码 clone 在当下目录下的 Paddle 的文件夹中, 并进入 Padde 目录下:

```
git clone https://github.com/PaddlePaddle/Paddle.git
```

```
cd Paddle
```

9. 切换到较稳定 release 分支下进行编译:

```
git checkout [分支名]
```

例如：

```
git checkout release/2.3
```

10. 并且请创建并进入一个叫 build 的目录下:

```
mkdir build && cd build
```

11. 执行 cmake:

具体编译选项含义请参见编译选项表

- 对于需要编译 CPU 版本 PaddlePaddle 的用户:

```
cmake .. -DPY_VERSION=3.7 -DPYTHON_INCLUDE_DIR=${PYTHON_INCLUDE_DIRS} \
-DPYTHON_LIBRARY=${PYTHON_LIBRARY} -DWITH_GPU=OFF
```

如果遇到 Could NOT find PROTOBUF (missing: PROTOBUF_LIBRARY PROTOBUF_INCLUDE_DIR) 可以重新执行一次 cmake 指令。请注意 PY_VERSION 参数更换为您需要的 python 版本

- 对于需要编译 GPU 版本 PaddlePaddle 的用户: (仅支持 CentOS7 (CUDA11.6/CUDA11.2/CUDA11.1/CUDA10.2/CUDA10.1))

1. 请确保您已经正确安装 nccl2，或者按照以下指令安装 nccl2 (这里提供的是 CUDA10.2, cuDNN7 下 nccl2 的安装指令，更多版本的安装信息请参考 NVIDIA[官方网站](#)) :

- Centos 系统可以参考以下命令

```
wget http://developer.download.nvidia.com/compute/machine-learning/repos/
→rhel7/x86_64/nvidia-machine-learning-repo-rhel7-1.0.0-1.x86_64.rpm
```

```
rpm -i nvidia-machine-learning-repo-rhel7-1.0.0-1.x86_64.rpm
```

```
yum install -y libncccl-2.7.8-1+cuda10.2 libncccl-devel-2.7.8-1+cuda10.2
→libncccl-static-2.7.8-1+cuda10.2
```

- Ubuntu 系统可以参考以下命令

```
wget https://developer.download.nvidia.com/compute/machine-learning/repos/
↪ubuntu1604/x86_64/nvidia-machine-learning-repo-ubuntu1604_1.0.0-1_amd64.deb
```

```
dpkg -i nvidia-machine-learning-repo-ubuntu1604_1.0.0-1_amd64.deb
```

```
sudo apt install -y libncc12=2.7.8-1+cuda10.2 libncc1-dev=2.7.8-1+cuda10.2
```

1. 如果您已经正确安装了 nccl2，就可以开始 cmake 了：(For Python3: 请给 PY_VERSION 参数配置正确的 python 版本)

```
cmake . -DPYTHON_EXECUTABLE:FILEPATH=[您可执行的Python3的路径] -DPYTHON_
↪INCLUDE_DIR:PATH=[之前的PYTHON_INCLUDE_DIRS] -DPYTHON_
↪LIBRARY:FILEPATH=[之前的PYTHON_LIBRARY] -DWITH_GPU=ON
```

注意：以上涉及 Python3 的命令，用 Python3.7 来举例，如您的 Python 版本为 3.6/3.8/3.9/3.10，请将上述命令中的 Python3.7 改成 Python3.6/Python3.8/Python3.9/Python3.10

12. 使用以下命令来编译：

```
make -j$(nproc)
```

使用多核编译

如果编译过程中显示“Too many open files”错误时，请使用指令 ulimit -n 8192 来增大当前进程允许打开的文件数，一般来说 8192 可以保证编译完成。

13. 编译成功后进入 /paddle/build/python/dist 目录下找到生成的.whl 包：

```
cd /paddle/build/python/dist
```

14. 在当前机器或目标机器安装编译好的.whl 包：

```
pip install -U (whl包的名字)
```

或

```
pip3 install -U (whl包的名字)
```

恭喜，至此您已完成 PaddlePaddle 的编译安装

验证安装

安装完成后您可以使用 python 或 python3 进入 python 解释器，输入

```
import paddle
```

再输入

```
paddle.utils.run_check()
```

如果出现 PaddlePaddle is installed successfully!，说明您已成功安装。

如何卸载

请使用以下命令卸载 PaddlePaddle：

- CPU 版本的 PaddlePaddle：

```
pip uninstall paddlepaddle
```

或

```
pip3 uninstall paddlepaddle
```

- GPU 版本的 PaddlePaddle：

```
pip uninstall paddlepaddle-gpu
```

或

```
pip3 uninstall paddlepaddle-gpu
```

使用 Docker 安装 PaddlePaddle 的用户，请进入包含 PaddlePaddle 的容器中使用上述命令，注意使用对应版本的 pip

MacOS 下从源码编译

环境准备

- MacOS 版本 10.x/11.x (64 bit) (不支持 GPU 版本)
- Python 版本 3.6/3.7/3.8/3.9/3.10 (64 bit)

选择 CPU/GPU

- 目前仅支持在 MacOS 环境下编译安装 CPU 版本的 PaddlePaddle

安装步骤

在 MacOS 系统下有 2 种编译方式，推荐使用 Docker 编译。Docker 环境中已预装好编译 Paddle 需要的各种依赖，相较本机编译环境更简单。

- *Docker* 源码编译
- 本机源码编译

使用 Docker 编译

Docker 是一个开源的应用容器引擎。使用 Docker，既可以将 PaddlePaddle 的安装 & 使用与系统环境隔离，也可以与主机共享 GPU、网络等资源

使用 Docker 编译 PaddlePaddle，您需要：

- 在本地主机上[安装 Docker](#)
- 使用 Docker ID 登陆 Docker，以避免出现 Authenticate Failed 错误

请您按照以下步骤安装：

1. 进入 Mac 的终端

2. 请选择您希望储存 PaddlePaddle 的路径，然后在该路径下使用以下命令将 PaddlePaddle 的源码从 [github](#) 克隆到本地当前目录下名为 **Paddle** 的文件夹中：

```
git clone https://github.com/PaddlePaddle/Paddle.git
```

3. 进入 **Paddle** 目录下：

```
cd Paddle
```

4. 拉取 PaddlePaddle 镜像

对于国内用户，因为网络问题下载 docker 比较慢时，可使用百度提供的镜像：

- CPU 版的 PaddlePaddle：

```
docker pull registry.baidubce.com/paddlepaddle/paddle:latest-dev
```

如果您的机器不在中国大陆地区，可以直接从 DockerHub 拉取镜像：

- CPU 版的 PaddlePaddle：

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

您可以访问DockerHub获取与您机器适配的镜像。

5. 创建并进入满足编译环境的 Docker 容器：

```
docker run --name paddle-test -v $PWD:/paddle --network=host -it registry.baidubce.  
com/paddlepaddle/paddle:latest-dev /bin/bash
```

- `--name paddle-test`: 为您创建的 Docker 容器命名为 `paddle-test`
- `-v: $PWD:/paddle`: 将当前目录挂载到 Docker 容器中的 `/paddle` 目录下 (Linux 中 `PWD` 变量会展开为当前路径的绝对路径)
- `-it`: 与宿主机保持交互状态
- `registry.baidubce.com/paddlepaddle/paddle:latest-dev`: 使用名为 `registry.baidubce.com/paddlepaddle/paddle:latest-dev` 的镜像创建 Docker 容器，`/bin/bash` 进入容器后启动 `/bin/bash` 命令

注意：请确保至少为 docker 分配 4g 以上的内存，否则编译过程可能因内存不足导致失败。您可以在 docker 用户界面的“Preferences-Resources”中设置容器的内存分配上限。

6. 进入 Docker 后进入 paddle 目录下：

```
cd /paddle
```

7. 切换到较稳定版本下进行编译:

```
git checkout [分支名]
```

例如：

```
git checkout release/2.3
```

注意：python3.6、python3.7 版本从 release/1.2 分支开始支持，python3.8 版本从 release/1.8 分支开始支持，python3.9 版本从 release/2.1 分支开始支持，python3.10 版本从 release/2.3 分支开始支持

8. 创建并进入/paddle/build 路径下：

```
mkdir -p /paddle/build && cd /paddle/build
```

9. 使用以下命令安装相关依赖：

- 安装 protobuf 3.1.0。

```
pip3.7 install protobuf==3.1.0
```

注意：以上用 Python3.7 命令来举例，如您的 Python 版本为 3.6/3.8/3.9/3.10，请将上述命令中的 pip3.7 改成 pip3.6/pip3.8/pip3.9/pip3.10

- 安装 patchelf，PatchELF 是一个小而实用的程序，用于修改 ELF 可执行文件的动态链接器和 RPATH。

```
apt install patchelf
```

10. 执行 cmake：

- 对于需要编译 CPU 版本 PaddlePaddle 的用户（我们目前不支持 MacOS 下 GPU 版本 PaddlePaddle 的编译）：

```
cmake .. -DPY_VERSION=3.7 -DWITH_GPU=OFF
```

- 具体编译选项含义请参见编译选项表
- 请注意修改参数-DPY_VERSION 为您希望编译使用的 python 版本，例如-DPY_VERSION=3.7 表示 python 版本为 3.7

11. 执行编译:

使用多核编译

```
make -j$(nproc)
```

注意：编译过程中需要从 `github` 上下载依赖，请确保您的编译环境能正常从 `github` 下载代码。

12. 编译成功后进入`/paddle/build/python/dist` 目录下找到生成的`.whl` 包：

```
cd /paddle/build/python/dist
```

13. 在当前机器或目标机器安装编译好的`.whl` 包：

```
pip3.7 install -U [whl包的名字]
```

注意：以上用 Python3.7 命令来举例，如您的 Python 版本为 3.6/3.8/3.9/3.10，请将上述命令中的 pip3.7 改成 pip3.6/pip3.8/pip3.9/pip3.10。

恭喜，至此您已完成 PaddlePaddle 的编译安装。您只需要进入 Docker 容器后运行 PaddlePaddle，即可开始使用。更多 Docker 使用请参见 Docker 官方文档

本机编译

请严格按照以下指令顺序执行

1. 检查您的计算机和操作系统是否符合我们支持的编译标准：

```
uname -m
```

并且在关于本机中查看系统版本。并提前安装OpenCV

2. 安装 Python 以及 pip:

请不要使用 **MacOS** 中自带 **Python**, 我们强烈建议您使用[Homebrew](#)安装 python(对于 **Python3** 请使用 [python官方下载](#)python3.6.x、python3.7.x、python3.8、python3.9、python3.10), pip 以及其他依赖, 这将会使您高效编译。

使用 Python 官网安装

请注意, 当您的 mac 上安装有多个 python 时请保证您正在使用的 python 是您希望使用的 python。

3. (Only For Python3) 设置 Python 相关的环境变量:

- a. 首先使用

```
find `dirname $(dirname $(which python3))` -name "libpython3.*.dylib"
```

找到 Pythonlib 的路径(弹出的第一个对应您需要使用的 python 的 dylib 路径), 然后(下面 [python-lib-path] 替换为找到文件路径)

- b. 设置 PYTHON_LIBRARIES:

```
export PYTHON_LIBRARY=[python-lib-path]
```

- c. 其次使用找到 PythonInclude 的路径 (通常是找到 [python-lib-path] 的上一级目录为同级目录的 include, 然后找到该目录下 python3.x 的路径), 然后 (下面 [python-include-path] 替换为找到路径)

- d. 设置 PYTHON_INCLUDE_DIR:

```
export PYTHON_INCLUDE_DIRS=[python-include-path]
```

- e. 设置系统环境变量路径:

```
export PATH=[python-bin-path]:$PATH
```

(这里 [python-bin-path] 为将 [python-lib-path] 的最后两级目录替换为/bin/后的目录)

- f. 设置动态库链接:

```
export LD_LIBRARY_PATH=[python-ld-path]
```

以及

```
export DYLD_LIBRARY_PATH=[python-ld-path]
```

(这里 [python-ld-path] 为 [python-bin-path] 的上一级目录)

- g. (可选) 如果您是在 MacOS 10.14 上编译 PaddlePaddle, 请保证您已经安装了[对应版本](#)的 Xcode。

4. 执行编译前请您确认您的环境中安装有编译依赖表中提到的相关依赖,否则我们强烈推荐使用 Homebrew 安装相关依赖。

MacOS 下如果您未自行修改或安装过“编译依赖表”中提到的依赖, 则仅需要使用 pip 安装 numpy, protobuf, wheel, 使用 homebrew 安装 wget, swig, unrar, 另外安装 cmake 即可

- a. 这里特别说明一下 **CMake** 的安装:

CMake 我们支持 3.15 以上版本, 推荐使用 CMake3.16, 请根据以下步骤安装:

1. 从 CMake[官方网站](#)下载 CMake 镜像并安装
2. 在控制台输入

```
sudo "/Applications/CMake.app/Contents/bin/cmake-gui" -install
```

- b. 如果您不想使用系统默认的 blas 而希望使用自己安装的 OPENBLAS 请参见[FAQ](#)

5. 将 PaddlePaddle 的源码 clone 在当下目录下的 Paddle 的文件夹中, 并进入 Padde 目录下:

```
git clone https://github.com/PaddlePaddle/Paddle.git
```

```
cd Paddle
```

6. 切换到较稳定 release 分支下进行编译:

```
git checkout [分支名]
```

例如:

```
git checkout release/2.3
```

注意: python3.6、python3.7 版本从 release/1.2 分支开始支持, python3.8 版本从 release/1.8 分支开始支持, python3.9 版本从 release/2.1 分支开始支持, python3.10 版本从 release/2.3 分支开始支持

7. 并且请创建并进入一个叫 build 的目录下:

```
mkdir build && cd build
```

8. 执行 cmake:

具体编译选项含义请参见编译选项表

- 若您的机器为 Mac M1 机器, 需要编译 **Arm 架构、CPU 版本 PaddlePaddle**:

```
cmake .. -DPY_VERSION=3.8 -DPYTHON_INCLUDE_DIR=${PYTHON_INCLUDE_DIRS} \
-DPYTHON_LIBRARY=${PYTHON_LIBRARY} -DWITH_GPU=OFF \
-DWITH_AVX=OFF -DWITH_ARM=ON
```

- 若您的机器不是 Mac M1 机器, 需要编译 **x86_64 架构、CPU 版本 PaddlePaddle**:

```
cmake .. -DPY_VERSION=3.8 -DPYTHON_INCLUDE_DIR=${PYTHON_INCLUDE_DIRS} \
-DPYTHON_LIBRARY=${PYTHON_LIBRARY} -DWITH_GPU=OFF
```

- DPY_VERSION=3.8 请修改为安装环境的 Python 版本
- 若编译 arm 架构的 paddlepaddle, 需要 cmake 版本为 3.19.2 以上

9. 使用以下命令来编译:

- 若您的机器为 Mac M1 机器, 需要编译 **Arm 架构、CPU 版本 PaddlePaddle**:

```
make TARGET=ARMV8 -j4
```

- 若您的机器不是 Mac M1 机器, 需要编译 **x86_64 架构、CPU 版本 PaddlePaddle**:

```
make -j4
```

10. 编译成功后进入 /paddle/build/python/dist 目录下找到生成的 .whl 包:

```
cd /paddle/build/python/dist
```

11. 在当前机器或目标机器安装编译好的.whl 包：

```
pip install -U (whl包的名字)
```

或

```
pip3 install -U (whl包的名字)
```

恭喜，至此您已完成 PaddlePaddle 的编译安装

验证安装

安装完成后您可以使用 python 或 python3 进入 python 解释器，输入

```
import paddle
```

再输入

```
paddle.utils.run_check()
```

如果出现 PaddlePaddle is installed successfully!，说明您已成功安装。

如何卸载

请使用以下命令卸载 PaddlePaddle

- CPU 版本的 PaddlePaddle:

```
pip uninstall paddlepaddle
```

或

```
pip3 uninstall paddlepaddle
```

使用 Docker 安装 PaddlePaddle 的用户，请进入包含 PaddlePaddle 的容器中使用上述命令，注意使用对应版本的 pip

Windows 下从源码编译

在 Windows 系统下提供 1 种编译方式：

- 本机编译

环境准备

- **Windows 7/8/10 专业版/企业版 (64bit)**
- **Python 版本 3.6/3.7/3.8/3.9/3.10 (64 bit)**
- **Visual Studio 2017/2019 社区版/专业版/企业版**

选择 CPU/GPU

- 如果你的计算机硬件没有 NVIDIA® GPU, 请编译 CPU 版本的 PaddlePaddle
- 如果你的计算机硬件有 NVIDIA® GPU, 推荐编译 GPU 版本的 PaddlePaddle, 建议安装 **CUDA 10.1/10.2/11.1/11.2/11.6**

本机编译过程

1. 安装必要的工具 cmake, git, python, Visual studio 2017/2019:

cmake: 建议安装 CMake3.17 版本, 官网下载[链接](#)。安装时注意勾选 Add CMake to the system PATH for all users, 将 CMake 添加到环境变量中。

git: 官网下载[链接](#), 使用默认选项安装。

python: 官网[链接](#), 可选择 3.6/3.7/3.8/3.9/3.10 中任一版本的 Windows installer(64-bit) 安装。安装时注意勾选 Add Python 3.x to PATH, 将 Python 添加到环境变量中。

Visual studio: 需根据 CUDA 版本选择对应的 Visual studio 版本, 当只编译 CPU 版本或者 CUDA 版本 < 11.2 时, 安装 VS2017; 当 CUDA 版本 >= 11.2 时, 安装 VS2019。官网[链接](#), 需要登录后下载, 建议下载 Community 社区版。在安装时需要在工作负载一栏中勾选 使用 C++ 的桌面开发和通用 Windows 平台开发, 并在语言包一栏中选择 英语。

2. 打开 Visual studio 终端: 在 Windows 桌面下方的搜索栏中搜索终端, 若安装的是 VS2017 版本, 则搜索 x64 Native Tools Command Prompt for VS 2017 或适用于 VS 2017 的 x64 本机工具命令提示符; 若安装的是 VS2019 版本, 则搜索 x64 Native Tools Command Prompt for VS 2019 或适用于 VS 2019 的 x64 本机工具命令提示符, 然后右键以管理员身份打开终端。后续的命令将在该终端执行。

3. 使用 pip 命令安装 Python 依赖:

- 通过 `python --version` 检查默认 python 版本是否是预期版本, 因为你的计算机可能安装有多个 python, 可通过修改系统环境变量的顺序来修改默认 Python 版本。

- 安装 numpy, protobuf, wheel, ninja

```
pip install numpy protobuf wheel ninja
```

4. 创建编译 Paddle 的文件夹（例如 D:\workspace），进入该目录并下载源码：

```
mkdir D:\workspace && cd /d D:\workspace  
  
git clone https://github.com/PaddlePaddle/Paddle.git  
  
cd Paddle
```

5. 切换到 2.2 分支下进行编译：

```
git checkout release/2.3
```

6. 创建名为 build 的目录并进入：

```
mkdir build  
  
cd build
```

7. 执行 cmake：

编译 CPU 版本的 Paddle：

```
cmake .. -GNinja -DWITH_GPU=OFF
```

编译 GPU 版本的 Paddle：

```
cmake .. -GNinja -DWITH_GPU=ON
```

其他编译选项含义请参见编译选项表。

注意：

1. 如果本机安装了多个 CUDA，将使用最新安装的 CUDA 版本。若需要指定 CUDA 版本，则需要设置环境变量和 cmake 选项，例如：

```
set CUDA_TOOLKIT_ROOT_DIR=C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v11.2  
set PATH=%CUDA_TOOLKIT_ROOT_DIR:/=%\bin;%CUDA_TOOLKIT_ROOT_DIR:/=%\libnvvp;%PATH  
cmake .. -GNinja -DWITH_GPU=ON -DCUDA_TOOLKIT_ROOT_DIR="%CUDA_TOOLKIT_ROOT_DIR%"
```

1. 如果本机安装了多个 Python，将使用最新安装的 Python 版本。若需要指定 Python 版本，则需要指定 Python 路径，例如：

```
cmake .. -GNinja -DWITH_GPU=ON -DPYTHON_EXECUTABLE=C:\Python38\python.exe -  
-DPYTHON_INCLUDE_DIR=C:\Python38\include -DPYTHON_LIBRARY=C:\Python38\libs\  
python38.lib
```

8. 执行编译:

```
ninja
```

9. 编译成功后进入 python\dist 目录下找到生成的 .whl 包:

```
cd python\dist
```

10. 安装编译好的 .whl 包:

```
pip install (whl包的名字) --force-reinstall
```

恭喜，至此你已完成 PaddlePaddle 的编译安装

验证安装

安装完成后你可以使用 python 进入 python 解释器，输入：

```
import paddle
```

```
paddle.utils.run_check()
```

如果出现 PaddlePaddle is installed successfully!，说明你已成功安装。

如何卸载

请使用以下命令卸载 PaddlePaddle：

- CPU 版本的 PaddlePaddle:

```
pip uninstall paddlepaddle
```

- GPU 版本的 PaddlePaddle:

```
pip uninstall paddlepaddle-gpu
```

飞腾/鲲鹏下从源码编译

环境准备

- 处理器: **FT2000+/Kunpeng 920 2426SK**
- 操作系统: **麒麟 v10/UOS**
- Python 版本 **2.7.15+/3.5.1+/3.6/3.7/3.8 (64 bit)**
- pip 或 pip3 版本 **9.0.1+ (64 bit)**

飞腾 FT2000+ 和鲲鹏 920 处理器均为 ARMV8 架构，在该架构上编译 Paddle 的方式一致，本文以 FT2000+ 为例，介绍 Paddle 的源码编译。

安装步骤

目前在 FT2000+ 处理器加国产化操作系统（麒麟 UOS）上安装 Paddle，只支持源码编译的方式，接下来详细介绍各个步骤。

源码编译

1. Paddle 依赖 cmake 进行编译构建，需要 cmake 版本 ≥ 3.15 ，如果操作系统提供的源包括了合适版本的 cmake，直接安装即可，否则需要源码安装

```
 wget https://github.com/Kitware/CMake/releases/download/v3.16.8/cmake-3.16.8.tar.  
 ↪gz
```

```
 tar -xzf cmake-3.16.8.tar.gz && cd cmake-3.16.8
```

```
 ./bootstrap && make && sudo make install
```

2. Paddle 内部使用 patchelf 来修改动态库的 rpath，如果操作系统提供的源包括了 patchelf，直接安装即可，否则需要源码安装，请参考[patchelf 官方文档](#)，后续会考虑在 ARM 上移出该依赖。

```
 ./bootstrap.sh
```

```
 ./configure
```

```
 make
```

```
make check
```

```
sudo make install
```

3. 根据requirements.txt安装 Python 依赖库，在飞腾加国产化操作系统环境中，pip 安装可能失败或不能正常工作，主要依赖通过源或源码安装的方式安装依赖库，建议使用系统提供源的方式安装依赖库。
4. 将 Paddle 的源代码克隆到当下目录下的 Paddle 文件夹中，并进入 Paddle 目录

```
git clone https://github.com/PaddlePaddle/Paddle.git
```

```
cd Paddle
```

5. 切换到 develop 分支下进行编译：

```
git checkout develop
```

6. 并且请创建并进入一个叫 build 的目录下：

```
mkdir build && cd build
```

7. 链接过程中打开文件数较多，可能超过系统默认限制导致编译出错，设置进程允许打开的最大文件数：

```
ulimit -n 4096
```

8. 执行 cmake：

具体编译选项含义请参见编译选项表

For Python2:

```
cmake .. -DPY_VERSION=2 -DPYTHON_EXECUTABLE=`which python2` -DWITH_ARM=ON -DWITH_
-TESTING=OFF -DCMAKE_BUILD_TYPE=Release -DON_INFER=ON -DWITH_XBYAK=OFF
```

For Python3:

```
cmake .. -DPY_VERSION=3 -DPYTHON_EXECUTABLE=`which python3` -DWITH_ARM=ON -DWITH_
-TESTING=OFF -DCMAKE_BUILD_TYPE=Release -DON_INFER=ON -DWITH_XBYAK=OFF
```

9. 使用以下命令来编译，注意，因为处理器为 ARM 架构，如果不加 TARGET=ARMV8 则会在编译的时候报错。

```
make TARGET=ARMV8 -j$(nproc)
```

10. 编译成功后进入 Paddle/build/python/dist 目录下找到生成的.whl 包。

11. 在当前机器或目标机器安装编译好的.whl 包：

```
pip install -U (whl包的名字) 或 pip3 install -U (whl包的名字)
```

恭喜，至此您已完成 PaddlePaddle 在 FT 环境下的编译安装。

验证安装

安装完成后您可以使用 `python` 或 `python3` 进入 `python` 解释器，输入 `import paddle`，再输入 `paddle.utils.run_check()`

如果出现 `PaddlePaddle is installed successfully!`，说明您已成功安装。

在 `mobilenetv1` 和 `resnet50` 模型上测试

```
wget -O profile.tar https://paddle-cetc15.bj.bcebos.com/profile.tar?authorization=bce-  
→auth-v1/4409a3f3dd76482ab77af112631f01e4/2020-10-09T10:11:53Z/-1/host/  
→786789f3445f498c6a1fd4d9cd3897ac7233700df0c6ae2fd78079eba89bf3fb
```

```
tar xf profile.tar && cd profile
```

```
python resnet.py --model_file ResNet50_inference/model --params_file ResNet50_  
→inference/params  
# 正确输出应为：[0.0002414 0.00022418 0.00053661 0.00028639 0.00072682 0.000213  
#           0.00638718 0.00128127 0.00013535 0.0007676 ]
```

```
python mobilenetv1.py --model_file mobilenetv1/model --params_file mobilenetv1/params  
# 正确输出应为：[0.00123949 0.00100392 0.00109539 0.00112206 0.00101901 0.00088412  
#           0.00121536 0.00107679 0.00106071 0.00099605]
```

```
python ernie.py --model_dir ernieL3H128_model/  
# 正确输出应为：[0.49879393 0.5012061 ]
```

如何卸载

请使用以下命令卸载 PaddlePaddle：

```
pip uninstall paddlepaddle
```

或

```
pip3 uninstall paddlepaddle
```

备注

已在 ARM 架构下测试过 resnet50, mobilenetv1, ernie, ELMo 等模型，基本保证了预测使用算子的正确性，如果您在使用过程中遇到计算结果错误，编译失败等问题，请到[issue](#)中留言，我们会及时解决。

预测文档见[doc](#)，使用示例见[Paddle-Inference-Demo](#)

申威下从源码编译

环境准备

- 处理器: **SW6A**
- 操作系统: 普华, **iSoft Linux 5**
- Python 版本 **2.7.15+/3.5.1+/3.6/3.7/3.8 (64 bit)**
- pip 或 pip3 版本 **9.0.1+ (64 bit)**

申威机器为 SW 架构，目前生态支持的软件比较有限，本文以比较 trick 的方式在申威机器上源码编译 Paddle，未来会随着申威软件的完善不断更新。

安装步骤

本文在申威处理器下安装 Paddle，接下来详细介绍各个步骤。

源码编译

1. 将 Paddle 的源代码克隆到当下目录下的 Paddle 文件夹中，并进入 Paddle 目录

```
git clone https://github.com/PaddlePaddle/Paddle.git
```

```
cd Paddle
```

2. 切换到 develop 分支下进行编译：

```
git checkout develop
```

3. Paddle 依赖 cmake 进行编译构建，需要 cmake 版本 ≥ 3.15 ，检查操作系统源提供 cmake 的版本，使用源的方式直接安装 cmake，`apt install cmake`，检查 cmake 版本，`cmake --version`，如果 `cmake >= 3.15` 则不需要额外的操作，否则请修改 Paddle 主目录的 `CMakeLists.txt`，`cmake_minimum_required(VERSION 3.15)` 修改为 `cmake_minimum_required(VERSION 3.0)`。

- 由于申威暂不支持 openblas，所以在此使用 blas + cblas 的方式，在此需要源码编译 blas 和 cblas。

```
pushd /opt
wget http://www.netlib.org/blas/blas-3.8.0.tgz
wget http://www.netlib.org/blas/blast-forum/cblas.tgz
tar xzf blas-3.8.0.tgz
tar xzf cblas.tgz
pushd BLAS-3.8.0
make
popd
pushd CBLAS
# 修改Makefile.in中BLLIB为BLAS-3.8.0的编译产物blas_LINUX.a
make
pushd lib
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PWD
ln -s cblas_LINUX.a libcblas.a
cp ../../BLAS-3.8.0/blas_LINUX.a .
ln -s blas_LINUX.a libblas.a
popd
popd
popd
```

- 根据requirements.txt安装 Python 依赖库，注意在申威系统中一般无法直接使用 pip 或源码编译安装 python 依赖包，建议使用源的方式安装，如果遇到部分依赖包无法安装的情况，请联系操作系统服务商提供支持。此外也可以通过 pip 安装的时候加--no-deps 的方式来避免依赖包的安装，但该种方式可能导致包由于缺少依赖不可用。
- 请创建并进入一个叫 build 的目录下：

```
mkdir build && cd build
```

- 链接过程中打开文件数较多，可能超过系统默认限制导致编译出错，设置进程允许打开的最大文件数：

```
ulimit -n 4096
```

- 执行 cmake：

具体编译选项含义请参见编译选项表

```
CBLAS_ROOT=/opt/CBLAS
```

For Python2:

```
cmake .. -DPY_VERSION=2 -DPYTHON_EXECUTABLE=`which python2` -DWITH_MKL=OFF -DWITH_
-TESTING=OFF -DCMAKE_BUILD_TYPE=Release -DON_INFER=ON -DWITH_PYTHON=ON -
-DREFERENCE_CBLAS_ROOT=${CBLAS_ROOT} -DWITH_CRYPTO=OFF -DWITH_XBYAK=OFF -DWITH_
-SW=ON -DCMAKE_CXX_FLAGS="-Wno-error -w"
```

(下页继续)

(续上页)

For Python3:

```
cmake .. -DPY_VERSION=3 -DPYTHON_EXECUTABLE=`which python3` -DWITH_MKL=OFF -DWITH_
→TESTING=OFF -DCMAKE_BUILD_TYPE=Release -DON_INFER=ON -DWITH_PYTHON=ON -
→DREFERENCE_CBLAS_ROOT=${CBLAS_ROOT} -DWITH_CRYPTO=OFF -DWITH_XBYAK=OFF -DWITH_
→SW=ON -DCMAKE_CXX_FLAGS="-Wno-error -w"
```

9. 编译。

```
make -j$(nproc)
```

10. 编译成功后进入 Paddle/build/python/dist 目录下找到生成的.whl 包。

11. 在当前机器或目标机器安装编译好的.whl 包:

python2 -m pip install -U (whl包的名字)、或 python3 -m pip install -U (whl包的名字)

恭喜，至此您已完成 PaddlePaddle 在 FT 环境下的编译安装。

验证安装

安装完成后您可以使用 `python` 或 `python3` 进入 `python` 解释器，输入 `import paddle`，再输入 `paddle.utils.run_check()`

如果出现 `PaddlePaddle is installed successfully!`, 说明您已成功安装。

在 mobilenetv1 和 resnet50 模型上测试

```
 wget -O profile.tar https://paddle-cetc15.bj.bcebos.com/profile.tar?authorization=bce-  
 ↲auth-v1/4409a3f3dd76482ab77af112631f01e4/2020-10-09T10:11:53Z/-1/host/  
 ↲786789f3445f498c6a1fd4d9cd3897ac7233700df0c6ae2fd78079eba89bf3fb
```

```
tar xf profile.tar && cd profile
```

```
python resnet.py --model_file ResNet50_inference/model --params_file ResNet50_
→inference/params
# 正确输出应为: [0.0002414  0.00022418 0.00053661 0.00028639 0.00072682 0.000213
#                  0.00638718 0.00128127 0.00013535 0.0007676 ]
```

```
python mobilenetv1.py --model_file mobilenetv1/model --params_file mobilenetv1/params  
# 正确输出应为: [0.00123949 0.00100392 0.00109539 0.00112206 0.00101901 0.00088412  
#           0.00121536 0.00107679 0.00106071 0.00099605]
```

```
python ernie.py --model_dir ernieL3H128_model/  
# 正确输出应为: [0.49879393 0.5012061 ]
```

如何卸载

请使用以下命令卸载 PaddlePaddle:

```
python3 -m pip uninstall paddlepaddle
```

或

```
python3 -m pip uninstall paddlepaddle
```

备注

已在申威下测试过 resnet50, mobilenetv1, ernie, ELMo 等模型，基本保证了预测使用算子的正确性，但可能会遇到浮点异常的问题，该问题我们后续会和申威一起解决，如果您在使用过程中遇到计算结果错误，编译失败等问题，请到[issue](#)中留言，我们会及时解决。

预测文档见[doc](#)，使用示例见[Paddle-Inference-Demo](#)

兆芯下从源码编译

环境准备

- 处理器: **ZHAOXIN KaiSheng KH-37800D**
- 操作系统: **centos7**
- Python 版本 **2.7.15+/3.5.1+/3.6/3.7/3.8 (64 bit)**
- pip 或 pip3 版本 **9.0.1+ (64 bit)**

兆芯为 x86 架构，编译方法与Linux 下从源码编译 cpu 版一致。

安装步骤

本文在 ZHAOXIN 处理器下安装 Paddle，接下来详细介绍各个步骤。

源码编译

1. Paddle 依赖 cmake 进行编译构建，需要 cmake 版本 ≥ 3.15 ，如果操作系统提供的源包括了合适版本的 cmake，直接安装即可，否则需要源码安装

```
wget https://github.com/Kitware/CMake/releases/download/v3.16.8/cmake-3.16.8.tar.  
--gz
```

```
tar -xzf cmake-3.16.8.tar.gz && cd cmake-3.16.8
```

```
./bootstrap && make && sudo make install
```

2. Paddle 内部使用 patchelf 来修改动态库的 rpath，如果操作系统提供的源包括了 patchelf，直接安装即可，否则需要源码安装，请参考[patchelf 官方文档](#)。

```
./bootstrap.sh
```

```
./configure
```

```
make
```

```
make check
```

```
sudo make install
```

3. 将 Paddle 的源代码克隆到当下目录下的 Paddle 文件夹中，并进入 Paddle 目录

```
git clone https://github.com/PaddlePaddle/Paddle.git
```

```
cd Paddle
```

4. 切换到 develop 分支下进行编译：

```
git checkout develop
```

5. 根据[requirements.txt](#)安装 Python 依赖库。

```
pip install -r python/requirements.txt
```

6. 请创建并进入一个叫 build 的目录下:

```
mkdir build && cd build
```

7. 链接过程中打开文件数较多, 可能超过系统默认限制导致编译出错, 设置进程允许打开的最大文件数:

```
ulimit -n 4096
```

8. 执行 cmake:

具体编译选项含义请参见[编译选项表](#)

For Python2:

```
cmake .. -DPY_VERSION=2 -DPYTHON_EXECUTABLE=`which python2` -DWITH_MKL=ON -DWITH_  
-TESTING=OFF -DCMAKE_BUILD_TYPE=Release -DON_INFER=ON -DWITH_PYTHON=ON
```

For Python3:

```
cmake .. -DPY_VERSION=3 -DPYTHON_EXECUTABLE=`which python3` -DWITH_MKL=ON -DWITH_  
-TESTING=OFF -DCMAKE_BUILD_TYPE=Release -DON_INFER=ON -DWITH_PYTHON=ON
```

9. 编译。

```
make -j$(nproc)
```

10. 编译成功后进入 Paddle/build/python/dist 目录下找到生成的.whl 包。

11. 在当前机器或目标机器安装编译好的.whl 包:

```
python2 -m pip install -U (whl包的名字) 或 python3 -m pip install -  
-U (whl包的名字)
```

恭喜, 至此您已完成 PaddlePaddle 在 FT 环境下的编译安装。

验证安装

安装完成后您可以使用 python 或 python3 进入 python 解释器, 输入

```
import paddle
```

再输入

```
paddle.utils.run_check()
```

如果出现 PaddlePaddle is installed successfully!，说明您已成功安装。

在 mobilenetv1 和 resnet50 模型上测试

```
wget -O profile.tar https://paddle-cetc15.bj.bcebos.com/profile.tar?authorization=bce-
→auth-v1/4409a3f3dd76482ab77af112631f01e4/2020-10-09T10:11:53Z/-1/host/
→786789f3445f498c6a1fd4d9cd3897ac7233700df0c6ae2fd78079eba89bf3fb
```

```
tar xf profile.tar && cd profile
```

```
python resnet.py --model_file ResNet50_inference/model --params_file ResNet50_
→inference/params
# 正确输出应为: [0.0002414 0.00022418 0.00053661 0.00028639 0.00072682 0.000213
#           0.00638718 0.00128127 0.00013535 0.0007676 ]
```

```
python mobilenetv1.py --model_file mobilenetv1/model --params_file mobilenetv1/params
# 正确输出应为: [0.00123949 0.00100392 0.00109539 0.00112206 0.00101901 0.00088412
#           0.00121536 0.00107679 0.00106071 0.00099605]
```

```
python ernie.py --model_dir ernieL3H128_model/
# 正确输出应为: [0.49879393 0.5012061 ]
```

如何卸载

请使用以下命令卸载 PaddlePaddle：

```
python3 -m pip uninstall paddlepaddle
```

或

```
python3 -m pip uninstall paddlepaddle
```

备注

已在 ZHAOXIN 下测试过 resnet50, mobilenetv1, ernie, ELMo 等模型，基本保证了预测使用算子的正确性，如果您在使用过程中遇到计算结果错误，编译失败等问题，请到[issue](#)中留言，我们会及时解决。

预测文档见[doc](#)，使用示例见[Paddle-Inference-Demo](#)

1.1.5 昆仑 XPU 芯片安装及运行飞桨

百度昆仑 AI 计算处理器（Baidu KUNLUN AI Computing Processor）是百度集十年 AI 产业技术实践于 2019 年推出的全功能 AI 芯片。基于自主研发的先进 XPU 架构，为云端和边缘端的人工智能业务而设计。百度昆仑与飞桨及其他国产软硬件强强组合，打造一个全面领先的国产化 AI 技术生态，部署和应用于诸多“人工智能+”的行业领域，包括智能云和高性能计算，智能制造、智慧城市和安防等。更多昆仑 XPU 芯片详情及技术指标请[点击这里](#)。参考以下内容可快速了解和体验昆仑 XPU 芯片上运行飞桨：

- 飞桨对昆仑 XPU 芯片的支持
- 飞桨框架昆仑 XPU 版安装说明
- 飞桨框架昆仑 XPU 版训练示例
- 飞桨预测库昆仑 XPU 版安装及使用示例

1.1.6 海光 DCU 芯片运行飞桨

DCU (Deep Computing Unit 深度计算器) 是海光 (HYGON) 推出的一款专门用户 AI 人工智能和深度学习的加速卡。Paddle ROCm 版当前可以支持在海光 CPU 与 DCU 上进行模型训练与推理。

参考以下内容可快速了解和体验在海光芯片上运行飞桨：

- 飞桨框架 ROCm 版支持模型
- 飞桨框架 ROCm 版安装说明
- 飞桨框架 ROCm 版训练示例
- 飞桨框架 ROCm 版预测示例

1.1.7 NGC 飞桨容器安装指南

整体介绍

NGC 飞桨容器针对 NVIDIA GPU 加速进行了优化，并包含一组经过验证的库，可启用和优化 NVIDIA GPU 性能。此容器还可能包含对 PaddlePaddle 源代码的修改，以最大限度地提高性能和兼容性。此容器还包含用于加速 ETL (DALI, RAPIDS),、训练 (cuDNN, NCCL) 和推理 (TensorRT) 工作负载的软件。

环境准备

使用 NGC 飞桨容器需要主机系统安装以下内容：

- Docker 引擎
- NVIDIA GPU 驱动程序
- NVIDIA 容器工具包

有关支持的版本，请参阅 NVIDIA 框架容器支持矩阵 和 NVIDIA 容器工具包文档。

不需要其他安装、编译或依赖管理。无需安装 NVIDIA CUDA Toolkit。

安装步骤

要运行容器，请按照 NVIDIA Containers For Deep Learning Frameworks User's Guide 中 Running A Container 一章中的说明发出适当的命令，并指定注册表、存储库和标签。有关使用 NGC 的更多信息，请参阅 NGC 容器用户指南。如果您有 Docker 19.03 或更高版本，启动容器的典型命令是：

```
docker run --gpus all --shm-size=1g --ulimit memlock=-1 -it --rm nvcr.io/
→nvidia/paddlepaddle:22.07-py3
```

如果您有 Docker 19.02 或更早版本，启动容器的典型命令是：

```
nvidia-docker run --shm-size=1g --ulimit memlock=-1 -it --rm nvcr.io/nvidia/
→paddlepaddle:22.07-py3
```

其中：* 22.07 是容器版本。PaddlePaddle 通过将其作为 Python 模块导入来运行：

```
$ python -c 'import paddle; paddle.utils.run_check()'
Running verify PaddlePaddle program ...
W0516 06:36:54.208734    442 device_context.cc:451] Please NOTE: device: 0,
→GPU Compute Capability: 8.0, Driver API Version: 11.7, Runtime API
→Version: 11.7
W0516 06:36:54.212574    442 device_context.cc:469] device: 0, cuDNN Version:
→8.4.
PaddlePaddle works well on 1 GPU.
W0516 06:37:12.706600    442 fuse_all_reduce_op_pass.cc:76] Find all_reduce_
→operators: 2. To make the speed faster, some all_reduce ops are fused_
→during training, after fusion, the number of all_reduce ops is 2.
PaddlePaddle works well on 8 GPUs.
PaddlePaddle is installed successfully! Let's start deep learning with_
→PaddlePaddle now.
```

有关入门和自定义 PaddlePaddle 映像的信息，请参阅容器内的 /workspace/README.md。

您可能希望从容器外部的位置提取数据和模型描述以供 PaddlePaddle 使用。为此，最简单的方法是将一个或多个主机目录挂载为 Docker 绑定挂载。例如：

```
docker run --gpus all -it --rm -v local_dir:container_dir nvcr.io/nvidia/  
↳ paddlepaddle:22.07-py3
```

注意：为了在队列之间共享数据，NCCL 可能需要共享系统内存用于 IPC 和固定（页面锁定）系统内存资源。操作系统对这些资源的限制可能需要相应增加。有关详细信息，请参阅系统文档。特别是，Docker 容器默认使用有限的共享和固定内存资源。在容器内使用 NCCL 时，建议您通过发出以下命令来增加这些资源：

```
--shm-size=1g --ulimit memlock=-1
```

在 docker run 命令中。

NGC 容器介绍

有关内容的完整列表，请参阅 [NGC 飞桨容器发行说明](#)。此容器映像包含 NVIDIA 版 PaddlePaddle 的完整源代码，位于 /opt/paddle/paddle。它是作为系统 Python 模块预构建和安装的。NVIDIA PaddlePaddle 容器针对与 NVIDIA GPU 一起使用进行了优化，并包含以下用于 GPU 加速的软件：

- [CUDA](#)
- [cuBLAS](#)
- [NVIDIA cuDNN](#)
- [NVIDIA NCCL \(optimized for NVLink\)](#)
- [NVIDIA Data Loading Library \(DALI\)](#)
- [TensorRT](#)
- [PaddlePaddle with TensorRT \(Paddle-TRT\)](#)

此容器中的软件堆栈已经过兼容性验证，不需要最终用户进行任何额外的安装或编译。此容器可以帮助您从端到端加速深度学习工作流程。

NGC 飞桨容器软件许可协议

当您下载或使用 NGC 飞桨容器时，即表示您已经同意并接受此 [最终用户许可协议](#) 的条款及其对应约束。

1.1.8 附录

飞桨支持的 Nvidia GPU 架构及安装方式

编译依赖表

编译选项表

BLAS

PaddlePaddle 支持 MKL 和 OpenBLAS 两种 BLAS 库。默认使用 MKL。如果使用 MKL 并且机器含有 AVX2 指令集，还会下载 MKL-DNN 数学库，详细参考[这里](#)。

如果关闭 MKL，则会使用 OpenBLAS 作为 BLAS 库。

CUDA/cuDNN

PaddlePaddle 在编译时/运行时会自动找到系统中安装的 CUDA 和 cuDNN 库进行编译和执行。使用参数 `-DCUDA_ARCH_NAME=Auto` 可以指定开启自动检测 SM 架构，加速编译。

PaddlePaddle 可以使用 cuDNN v5.1 之后的任何一个版本来编译运行，但尽量请保持编译和运行使用的 cuDNN 是同一个版本。我们推荐使用最新版本的 cuDNN。

编译选项的设置

PaddePaddle 通过编译时指定路径来实现引用各种 BLAS/CUDA/cuDNN 库。cmake 编译时，首先在系统路径 (`/usr/lib` 和 `/usr/local/lib`) 中搜索这几个库，同时也会读取相关路径变量来进行搜索。通过使用`-D` 命令可以设置，例如：

```
cmake .. -DWITH_GPU=ON -DWITH_TESTING=OFF -DCUDNN_ROOT=/opt/cudnnv5
```

注意：这几个编译选项的设置，只在第一次 cmake 的时候有效。如果之后想要重新设置，推荐清理整个编译目录 (`rm -rf`) 后，再指定。

安装包列表

您可以在 [Release History](#) 中找到 PaddlePaddle-gpu 的各个发行版本。

其中 postXX 对应的是 CUDA 和 cuDNN 的版本，postXX 之前的数字代表 Paddle 的版本

需要注意的是，命令中 paddlepaddle-gpu==2.3.2 在 windows 环境下，会默认安装支持 CUDA 10.2 和 cuDNN 7 的对应 [版本号] 的 PaddlePaddle 安装包

多版本 whl 包列表-Release

表格说明

- 纵轴

cpu-mkl: 支持 CPU 训练和预测，使用 Intel mkl 数学库

cuda10_cudnn7-mkl: 支持 GPU 训练和预测，使用 Intel mkl 数学库

- 横轴

一般是类似于“cp37-cp37m”的形式，其中：

37:python tag, 指 python3.7，类似的还有“36”、“38”、“39”等

mu: 指 unicode 版本 python，若为 m 则指非 unicode 版本 python

- 安装包命名规则

每个安装包都有一个专属的名字，它们是按照 Python 的官方规则来命名的，形式如下：

{distribution}-{version}{-{build tag}}?{-{python tag}}{-{abi tag}}{-{platform tag}}.whl

其中 build tag 可以缺少，其他部分不能缺少

distribution: wheel 名称 version: 版本，例如 0.14.0 (要求必须是数字格式)

python tag: 类似'py36', 'py37', 'py38', 'py39'，用于标明对应的 python 版本

abi tag: 类似'cp33m', 'abi3', 'none'

platform tag: 类似'linux_x86_64', 'any'

多版本 whl 包列表-develop

在 Docker 中执行 PaddlePaddle 训练程序

假设您已经在当前目录（比如在/home/work）编写了一个 PaddlePaddle 的程序: train.py （可以参考 [PaddlePaddleBook](#) 编写），就可以使用下面的命令开始执行训练：

```
cd /home/work
```

```
docker run -it -v $PWD:/work registry.baidubce.com/paddlepaddle/paddle /work/train.py
```

上述命令中，-it 参数说明容器已交互式运行；-v \$PWD:/work 指定将当前路径（Linux 中 PWD 变量会展开为当前路径的绝对路径）挂载到容器内部的:/work 目录：registry.baidubce.com/paddlepaddle/paddle 指定需要使用的容器；最后/work/train.py 为容器内执行的命令，即运行训练程序。

当然，您也可以进入到 Docker 容器中，以交互式的方式执行或调试您的代码：

```
docker run -it -v $PWD:/work registry.baidubce.com/paddlepaddle/paddle /bin/bash
```

```
cd /work
```

```
python train.py
```

注：**PaddlePaddle Docker 镜像为了减小体积，默认没有安装 vim，您可以在容器中执行 apt-get install -y vim 安装后，在容器中编辑代码。**

使用 Docker 启动 PaddlePaddle Book 教程

使用 Docker 可以快速在本地启动一个包含了 PaddlePaddle 官方 Book 教程的 Jupyter Notebook，可以通过网页浏览。PaddlePaddle Book 是为用户和开发者制作的一个交互式的 Jupyter Notebook。如果您想要更深入了解 deep learning，可以参考 [PaddlePaddle Book](#)。大家可以通过它阅读教程，或者制作和分享带有代码、公式、图表、文字的交互式文档。

我们提供可以直接运行 PaddlePaddle Book 的 Docker 镜像，直接运行：

```
docker run -p 8888:8888 registry.baidubce.com/paddlepaddle/book
```

国内用户可以使用下面的镜像源来加速访问：

```
docker run -p 8888:8888 registry.baidubce.com/paddlepaddle/book
```

然后在浏览器中输入以下网址：

```
http://localhost:8888/
```

使用 Docker 执行 GPU 训练

为了保证 GPU 驱动能够在镜像里面正常运行，我们推荐使用 nvidia-docker 来运行镜像。请不要忘记提前在物理机上安装 GPU 最新驱动。

```
nvidia-docker run -it -v $PWD:/work registry.baidubce.com/paddlepaddle/paddle:latest-gpu /bin/bash
```

注：如果没有安装 nvidia-docker，可以尝试以下的方法，将 CUDA 库和 Linux 设备挂载到 Docker 容器内：

```
export CUDA_SO="$ (\ls /usr/lib64/libcuda* | xargs -I{} echo '-v {}:{}'.') \ $(\ls /usr/lib64/libnvidia* | xargs -I{} echo '-v {}:{}'.')"
export DEVICES=$ (\ls /dev/nvidia* | xargs -I{} echo '--device {}:{}'.')
docker run ${CUDA_SO} \
${DEVICES} -it registry.baidubce.com/paddlepaddle/paddle:latest-gpu
```

Chapter 2

使用指南

飞桨开源框架 (PaddlePaddle) 是一个易用、高效、灵活、可扩展的深度学习框架。

你可参考飞桨框架的 [Github](#) 了解详情，也可阅读 [版本说明](#) 了解最新版本的特性。

使用教程分为如下的模块：

- 模型开发入门
- 模型开发更多用法
- 动态图转静态图
- 预测部署
- 分布式训练
- 性能调优
- 模型迁移
- 硬件支持
- 自定义算子
- 环境变量

2.1 模型开发入门

本部分将介绍飞桨框架 2.0 的开发流程。

为了快速上手飞桨框架 2.0，你可以参考 [10 分钟快速上手飞桨](#)；

当完成了快速上手的任务后，下面这些模块会阐述如何用飞桨框架 2.0，实现深度学习过程中的每一步。具体包括：

- [Tensor 介绍](#)：介绍飞桨基本数据类型 *Tensor* 的概念与常见用法。

- **数据集定义与加载**：飞桨框架数据加载的方式，主要为 `paddle.io.Dataset + paddle.io.DataLoader`，以及飞桨内置数据集的介绍。
- **数据预处理**：飞桨框架数据预处理的方法，主要是 `paddle.vision.transform.*`。
- **模型组网**：飞桨框架组网 API 的介绍，主要是 `paddle.nn.*`，然后是飞桨框架组网方式的介绍，即 `Sequential` 的组网与 `SubClass` 的组网。
- **训练与预测**：飞桨框架训练与预测的方法，有两种方式，一种是使用高层 API `paddle.Model` 封装模型，然后调用 `model.fit()`、`model.evaluate()`、`model.predict()` 完成模型的训练与预测；另一种是用基础 API 完成模型的训练与预测，也就是对高层 API 的拆解。
- **模型的加载与保存**：飞桨框架模型的加载与保存体系介绍。

2.1.1 Tensor 介绍

一、Tensor 的概念介绍

飞桨使用张量（Tensor）来表示神经网络中传递的数据，Tensor 可以理解为多维数组，类似于 Numpy 数组（ndarray）的概念。与 Numpy 数组相比，Tensor 除了支持运行在 CPU 上，还支持运行在 GPU 及各种 AI 芯片上，以实现计算加速；此外，飞桨基于 Tensor，实现了深度学习所必须的反向传播功能和多种多样的组网算子，从而可更快捷地实现深度学习组网与训练等功能。两者具体异同点可参见下文 *Tensor 与 Numpy 数组相互转换*。

在飞桨框架中，神经网络的输入、输出数据，以及网络中的参数均采用 Tensor 数据结构，示例如下：

```
def train(model):
    model.train()
    epochs = 2
    optim = paddle.optimizer.Adam(learning_rate=0.001, parameters=model.parameters())
    # 模型训练的两层循环
    for epoch in range(epochs):
        for batch_id, data in enumerate(train_loader()):
            x_data = data[0]
            y_data = data[1]
            print("x_data: ", x_data[0][0][0][0]) #←打印神经网络的输入：批数据中的第一个数据的第一个元素
            predicts = model(x_data)
            print("predicts: ", predicts[0]) #←打印神经网络的输出：批数据中的第一个数据的第一个元素
            print("weight: ", model.linear1.weight[0][0]) #←打印神经网络的权重：linear1层的weight中的第一个元素
            loss = F.cross_entropy(predicts, y_data)
            acc = paddle.metric.accuracy(predicts, y_data)
            loss.backward()
            optim.step()
```

(下页继续)

(续上页)

```
    optim.clear_grad()
    break
break
model = LeNet()
train(model)
```

```
x_data: Tensor(shape=[1], dtype=float32, place=Place(gpu:0), stop_gradient=True,
[-1.])
predicts: Tensor(shape=[1], dtype=float32, place=Place(gpu:0), stop_gradient=False,
[-0.72636688])
weight: Tensor(shape=[1], dtype=float32, place=Place(gpu:0), stop_gradient=False,
[0.02227839])
```

以上示例代码来源 [使用 LeNet 在 MNIST 数据集实现图像分类任务 5.1 小节](#)（篇幅原因仅截取部分），分别打印了神经网络的输入、输出数据和网络中的参数，可以看到均采用了 Tensor 数据结构。

二、Tensor 的创建

飞桨可基于给定数据手动创建 Tensor，并提供了多种方式，如：

2.1 指定数据创建

2.2 指定形状创建

2.3 指定区间创建

另外在常见深度学习任务中，数据样本可能是图片（image）、文本（text）、语音（audio）等多种类型，在送入神经网络训练或推理前均需要创建为 Tensor。飞桨提供了将这类数据手动创建为 Tensor 的方法，如：

2.4 指定图像、文本数据创建

由于这些操作在整个深度学习任务流程中比较常见且固定，飞桨在一些 API 中封装了 Tensor 自动创建的操作，从而无须手动转 Tensor。

2.5 自动创建 *Tensor* 的功能介绍

如果你熟悉 Numpy，已经使用 Numpy 数组创建好数据，飞桨可以很方便地将 Numpy 数组转为 Tensor，具体介绍如：

六、*Tensor* 与 Numpy 数组相互转换

2.1 指定数据创建

与 Numpy 创建数组方式类似，通过给定 Python 序列（如列表 list、元组 tuple），可使用 `paddle.to_tensor` 创建任意维度的 Tensor。示例如下：

(1) 创建类似向量 (vector) 的 1 维 Tensor：

```
import paddle # 后面的示例代码默认已导入 paddle 模块
ndim_1_Tensor = paddle.to_tensor([2.0, 3.0, 4.0])
print(ndim_1_Tensor)
```

```
Tensor(shape=[3], dtype=float32, place=Place(gpu:0), stop_gradient=True,
       [2., 3., 4.])
```

特殊地，如果仅输入单个标量 (scalar) 数据（例如 float/int/bool 类型的单个元素），则会创建形状为 [1] 的 Tensor，即 0 维 Tensor：

```
paddle.to_tensor(2)
paddle.to_tensor([2])
```

```
# 上述两种创建方式完全一致，形状均为 [1]，输出如下：
```

```
Tensor(shape=[1], dtype=int64, place=Place(gpu:0), stop_gradient=True,
       [2])
```

(2) 创建类似矩阵 (matrix) 的 2 维 Tensor：

```
ndim_2_Tensor = paddle.to_tensor([[1.0, 2.0, 3.0],
                                   [4.0, 5.0, 6.0]])
print(ndim_2_Tensor)
```

```
Tensor(shape=[2, 3], dtype=float32, place=Place(gpu:0), stop_gradient=True,
       [[1., 2., 3.],
        [4., 5., 6.]])
```

(3) 创建 3 维 Tensor：

```
ndim_3_Tensor = paddle.to_tensor([[[1, 2, 3, 4, 5],
                                   [6, 7, 8, 9, 10]],
                                   [[11, 12, 13, 14, 15],
                                    [16, 17, 18, 19, 20]]])
print(ndim_3_Tensor)
```

```
Tensor(shape=[2, 2, 5], dtype=int64, place=Place(gpu:0), stop_gradient=True,
       [[[1, 2, 3, 4, 5],
```

(下页继续)

(续上页)

```
[6 , 7 , 8 , 9 , 10]],

[[11, 12, 13, 14, 15],
 [16, 17, 18, 19, 20]])
```

上述不同维度的 Tensor 可视化的表示如下图所示：

需要注意的是，Tensor 必须形如矩形，即在任何一个维度上，元素的数量必须相等，否则会抛出异常，示例如下：

```
ndim_2_Tensor = paddle.to_tensor([[1.0, 2.0],
                                  [4.0, 5.0, 6.0]])
```

```
ValueError:
    Failed to convert input data to a regular ndarray :
    - Usually this means the input data contains nested lists with different
→lengths.
```

说明：

- 飞桨也支持将 Tensor 转换为 Python 序列数据，可通过 `paddle.tolist` 实现，飞桨实际的转换处理过程是 **Python 序列 <-> Numpy 数组 <-> Tensor**。
- 基于给定数据创建 Tensor 时，飞桨是通过拷贝方式创建，与原始数据不共享内存。

2.2 指定形状创建

如果要创建一个指定形状的 Tensor，可以使用 `paddle.zeros`、`paddle.ones`、`paddle.full` 实现。

```
paddle.zeros([m, n])          # 创建数据全为 0, 形状为 [m, n] 的 Tensor
paddle.ones([m, n])           # 创建数据全为 1, 形状为 [m, n] 的 Tensor
paddle.full([m, n], 10)        # 创建数据全为 10, 形状为 [m, n] 的 Tensor
```

例如，`paddle.ones([2, 3])` 输出如下：

```
Tensor(shape=[2, 3], dtype=float32, place=Place(gpu:0), stop_gradient=True,
      [[1., 1., 1.],
       [1., 1., 1.]])
```

2.3 指定区间创建

如果要在指定区间内创建 Tensor，可以使用 paddle.arange、paddle.linspace 实现。

```
paddle.arange(start, end, step) # 创建以步长step均匀分隔区间[start, end)的Tensor
paddle.linspace(start, end, num) # 创建以元素个数num均匀分隔区间[start, end)的Tensor
```

示例如下：

```
paddle.arange(start=1, end=5, step=1)
```

```
Tensor(shape=[4], dtype=int64, place=Place(gpu:0), stop_gradient=True,
       [1, 2, 3, 4])
```

说明：

除了以上指定数据、形状、区间创建 Tensor 的方法，飞桨还支持如下类似的创建方式，如：

- 创建一个空 Tensor，即根据 shape 和 dtype 创建尚未初始化元素值的 Tensor，可通过 paddle.empty 实现。
- 创建一个与其他 Tensor 具有相同 shape 与 dtype 的 Tensor，可通过 paddle.ones_like、paddle.zeros_like、paddle.full_like、paddle.empty_like 实现。
- 拷贝并创建一个与其他 Tensor 完全相同的 Tensor，可通过 paddle.clone 实现。
- 创建一个满足特定分布的 Tensor，如 paddle.rand, paddle.randn, paddle.randint 等。
- 通过设置随机种子创建 Tensor，可每次生成相同元素值的随机数 Tensor，可通过 paddle.seed 和 paddle.rand 组合实现。

2.4 指定图像、文本数据创建

在常见深度学习任务中，数据样本可能是图片（image）、文本（text）、语音（audio）等多种类型，在送入神经网络训练或推理前，这些数据和对应的标签均需要创建为 Tensor。以下是图像场景和 NLP 场景中手动转换 Tensor 方法的介绍。

- 对于图像场景，可使用 paddle.vision.transforms.ToTensor 直接将 PIL.Image 格式的数据转为 Tensor，使用 paddle.to_tensor 将图像的标签（Label，通常是 Python 或 Numpy 格式的数据）转为 Tensor。
- 对于文本场景，需将文本数据解码为数字后，再通过 paddle.to_tensor 转为 Tensor。不同文本任务标签形式不一样，有的任务标签也是文本，有的则是数字，均需最终通过 paddle.to_tensor 转为 Tensor。

下面以图像场景为例介绍，以下示例代码中将随机生成的图片转换为 Tensor。

```
import numpy as np
from PIL import Image
```

(下页继续)

(续上页)

```
import paddle.vision.transforms as T
import paddle.vision.transforms.functional as F

fake_img = Image.fromarray((np.random.rand(224, 224, 3) * 255.).astype(np.uint8)) # ← 创建随机图片
transform = T.ToTensor()
tensor = transform(fake_img) # 使用 ToTensor() 将图片转换为 Tensor
print(tensor)
```

```
Tensor(shape=[3, 224, 224], dtype=float32, place=Place(gpu:0), stop_gradient=True,
      [[[0.78039223, 0.72941178, 0.34117648, ..., 0.76470596, 0.57647061, 0.
      ← 94901967],
      ...,
      [0.49803925, 0.72941178, 0.80392164, ..., 0.08627451, 0.97647065, 0.
      ← 43137258]]])
```

说明：

实际编码时，由于飞桨数据加载的 `paddle.io.DataLoader` API 能够将原始 `paddle.io.Dataset` 定义的数据自动转换为 `Tensor`，所以可以不做手动转换。具体如下节介绍。

2.5 自动创建 `Tensor` 的功能介绍

除了手动创建 `Tensor` 外，实际在飞桨框架中有一些 API 封装了 `Tensor` 创建的操作，从而无需用户手动创建 `Tensor`。例如 `paddle.io.DataLoader` 能够基于原始 `Dataset`，返回读取 `Dataset` 数据的迭代器，迭代器返回的数据中的每个元素都是一个 `Tensor`。另外在一些高层 API，如 `paddle.Model.fit`、`paddle.Model.predict`，如果传入的数据不是 `Tensor`，会自动转为 `Tensor` 再进行模型训练或推理。

说明：

`paddle.Model.fit`、`paddle.Model.predict` 等高层 API 支持传入 `Dataset` 或 `DataLoader`，如果传入的是 `Dataset`，那么会用 `DataLoader` 封装转为 `Tensor` 数据；如果传入的是 `DataLoader`，则直接从 `DataLoader` 迭代读取 `Tensor` 数据送入模型训练或推理。因此即使没有写将数据转为 `Tensor` 的代码，也能正常执行，提升了编程效率和容错性。

以下示例代码中，分别打印了原始数据集的数据，和送入 `DataLoader` 后返回的数据，可以看到数据结构由 Python list 转为了 `Tensor`。

```
import paddle

from paddle.vision.transforms import Compose, Normalize

transform = Compose([Normalize(mean=[127.5],
                               std=[127.5],
```

(下页继续)

(续上页)

```

        data_format='CHW')))

test_dataset = paddle.vision.datasets.MNIST(mode='test', transform=transform)
print(test_dataset[0][1]) # 打印原始数据集的第一个数据的label
loader = paddle.io.DataLoader(test_dataset)
for data in enumerate(loader):
    x, label = data[1]
    print(label) # 打印由DataLoader返回的迭代器中的第一个数据的label
    break

```

```

[7] # 原始数据中label为Python list
Tensor(shape=[1, 1], dtype=int64, place=Place(gpu_pinned), stop_gradient=True,
      [[7]]) # 由DataLoader转换后，label为Tensor

```

三、Tensor 的属性

在前文中，可以看到打印 Tensor 时有 shape、dtype、place 等信息，这些都是 Tensor 的重要属性，想要了解如何操作 Tensor 需要对其属性有一定了解，接下来分别展开介绍 Tensor 的属性相关概念。

```

Tensor(shape=[3], dtype=float32, place=Place(gpu:0), stop_gradient=True,
      [2., 3., 4.])

```

3.1 Tensor 的形状 (shape)

(1) 形状的介绍

形状是 Tensor 的一个重要的基础属性，可以通过 `Tensor.shape` 查看一个 Tensor 的形状，以下为相关概念：

- `shape`: 描述了 Tensor 每个维度上元素的数量。
- `ndim`: Tensor 的维度数量，例如向量的维度为 1，矩阵的维度为 2，Tensor 可以有任意数量的维度。
- `axis` 或者 `dimension`: Tensor 的轴，即某个特定的维度。
- `size`: Tensor 中全部元素的个数。

创建 1 个四维 Tensor，并通过图形来直观表达以上几个概念之间的关系：

```

ndim_4_Tensor = paddle.ones([2, 3, 4, 5])

```

```

print("Data Type of every element:", ndim_4_Tensor.dtype)
print("Number of dimensions:", ndim_4_Tensor.ndim)
print("Shape of Tensor:", ndim_4_Tensor.shape)

```

(下页继续)

(续上页)

```
print("Elements number along axis 0 of Tensor:", ndim_4_Tensor.shape[0])
print("Elements number along the last axis of Tensor:", ndim_4_Tensor.shape[-1])
```

```
Data Type of every element: paddle.float32
Number of dimensions: 4
Shape of Tensor: [2, 3, 4, 5]
Elements number along axis 0 of Tensor: 2
Elements number along the last axis of Tensor: 5
```

(2) 重置 Tensor 形状 (Reshape) 的方法

重新设置 Tensor 的 shape 在深度学习任务中比较常见，如一些计算类 API 会对输入数据有特定的形状要求，这时可通过 `paddle.reshape` 接口来改变 Tensor 的 shape，但并不改变 Tensor 的 size 和其中的元素数据。

以下示例代码中，创建 1 个 `shape=[3]` 的一维 Tensor，使用 `reshape` 功能将该 Tensor 重置为 `shape=[1, 3]` 的二维 Tensor。这种做法经常用在把一维的标签（label）数据扩展为二维，由于飞桨框架中神经网络通常需要传入一个 batch 的数据进行计算，因此可将数据增加一个 batch 维，方便后面的数据计算。

```
ndim_1_Tensor = paddle.to_tensor([1, 2, 3])
print("the shape of ndim_1_Tensor:", ndim_1_Tensor.shape)

reshape_Tensor = paddle.reshape(ndim_1_Tensor, [1, 3])
print("After reshape:", reshape_Tensor.shape)
```

```
the shape of ndim_1_Tensor: [3]
After reshape: [1, 3]
```

在指定新的 shape 时存在一些技巧：

- -1 表示这个维度的值是从 Tensor 的元素总数和剩余维度自动推断出来的。因此，有且只有一个维度可以被设置为 -1。
- 0 表示该维度的元素数量与原值相同，因此 shape 中 0 的索引值必须小于 Tensor 的维度（索引值从 0 开始计，如第 1 维的索引值是 0，第二维的索引值是 1）。

通过几个例子来详细了解：

```
origin:[3, 2, 5] reshape:[3, 10]           actual: [3, 10] # 直接指定目标 shape
origin:[3, 2, 5] reshape:[-1]               actual: [30] # ↴
↳ 转换为1维，维度根据元素总数推断出来是3*2*5=30
origin:[3, 2, 5] reshape:[-1, 5]            actual: [6, 5] # ↴
↳ 转换为2维，固定一个维度5，另一个维度根据元素总数推断出来是30÷5=6
origin:[3, 2, 5] reshape:[0, -1]            actual: [3, 6] # reshape:[0, -
↳ 1]中0的索引值为0，按照规则，转换后第0维的元素数量与原始Tensor第0维的元素数量相同，为3；第1维的元素
origin:[3, 2] reshape:[3, 1, 0]             error: # reshape:[3, 1, -
↳ 0]中0的索引值为2，但原Tensor只有2维，无法找到与第3维对应的元素数量，因此出错。 (下页继续)
```

(续上页)

从上面的例子可以看到，通过 `reshape[:-1]`，可以很方便地将 Tensor 按其在计算机上的内存分布展平为一维。

```
print("Tensor flattened to Vector:", paddle.reshape(ndim_3_Tensor, [-1]).numpy())
```

```
Tensor flattened to Vector: [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]
```

说明：

除了 `paddle.reshape` 可重置 Tensor 的形状，还可通过如下方法改变 shape：

- `paddle.squeeze`，可实现 Tensor 的降维操作，即把 Tensor 中尺寸为 1 的维度删除。
- `paddle.unsqueeze`，可实现 Tensor 的升维操作，即向 Tensor 中某个位置插入尺寸为 1 的维度。
- `paddle.flatten`，将 Tensor 的数据在指定的连续维度上展平。
- `transpose`，对 Tensor 的数据进行重排。

(3) 原位 (Inplace) 操作和非原位操作的区别

飞桨框架的 API 有原位 (Inplace) 操作和非原位操作之分，原位操作即在原 Tensor 上保存操作结果，输出 Tensor 将与输入 Tensor 共享数据，并且没有 Tensor 数据拷贝的过程。非原位操作则不会修改原 Tensor，而是返回一个新的 Tensor。通过 API 名称区分两者，如 `paddle.reshape` 是非原位操作，`paddle.reshape_` 是原位操作。

下面以 `reshape` 为例说明，通过对比 Tensor 的 name（每个 Tensor 创建时都会有一个独一无二的 name），判断是否为同一个 Tensor。

```
origin_tensor = paddle.to_tensor([1, 2, 3])
new_tensor = paddle.reshape(origin_tensor, [1, 3]) # 非原位操作
same_tensor = paddle.reshape_(origin_tensor, [1, 3]) # 原位操作
print("origin_tensor name: ", origin_tensor.name)
print("new_tensor name: ", new_tensor.name)
print("same_tensor name: ", same_tensor.name)
```

```
origin_tensor name: generated_tensor_0
new_tensor name: auto_0_ # 非原位操作后产生的Tensor与原始Tensor的名称不同
same_tensor name: generated_tensor_0 # 原位操作后产生的Tensor与原始Tensor的名称相同
```

3.2 Tensor 的数据类型 (dtype)

(1) 指定数据类型的介绍

Tensor 的数据类型 `dtype` 可以通过 `Tensor.dtype` 查看, 支持类型包括: `bool`、`float16`、`float32`、`float64`、`uint8`、`int8`、`int16`、`int32`、`int64`、`complex64`、`complex128`。

同一 Tensor 中所有元素的数据类型均相同, 通常通过如下方式指定:

- 通过给定 Python 序列创建的 Tensor, 可直接使用 `dtype` 参数指定。如果未指定:
 - 对于 Python 整型数据, 默认会创建 `int64` 型 Tensor;
 - 对于 Python 浮点型数据, 默认会创建 `float32` 型 Tensor, 并且可以通过 `paddle.set_default_dtype` 来调整浮点型数据的默认类型。

```
# 创建Tensor时指定dtype
ndim_1_tensor = paddle.to_tensor([2.0, 3.0, 4.0], dtype='float64')
print("Tensor dtype of ndim_1_tensor:", ndim_1_tensor.dtype)

# 创建Tensor时不指定dtype, 自动选择对应的默认类型
print("Tensor dtype from Python integers:", paddle.to_tensor(1).dtype)
print("Tensor dtype from Python floating point:", paddle.to_tensor(1.0).dtype)
```

```
Tensor dtype of ndim_1_tensor: paddle.float64
Tensor dtype from Python integers: paddle.int64
Tensor dtype from Python floating point: paddle.float32
```

- 通过 Numpy 数组或其他 Tensor 创建的 Tensor, 则与其原来的数据类型保持相同。
- Tensor 不仅支持 `float`、`int` 类型数据, 也支持 `complex` 复数类型数据。如果输入为复数, 则 Tensor 的 `dtype` 为 `complex64` 或 `complex128`, 其每个元素均为 1 个复数。如果未指定, 默认数据类型是 `complex64`:

```
ndim_2_Tensor = paddle.to_tensor([[1+1j), (2+2j)],
                                 [(3+3j), (4+4j)])
print(ndim_2_Tensor)
```

```
Tensor(shape=[2, 2], dtype=complex64, place=Place(gpu:0), stop_gradient=True,
       [[(1+1j), (2+2j)],
        [(3+3j), (4+4j)]])
```

(2) 修改数据类型的方法

飞桨框架提供了 `paddle.cast` 接口来改变 Tensor 的 `dtype`:

```
float32_Tensor = paddle.to_tensor(1.0)

float64_Tensor = paddle.cast(float32_Tensor, dtype='float64')
```

(下页继续)

(续上页)

```
print("Tensor after cast to float64:", float64_Tensor.dtype)

int64_Tensor = paddle.cast(float32_Tensor, dtype='int64')
print("Tensor after cast to int64:", int64_Tensor.dtype)
```

```
Tensor after cast to float64: paddle.float64
Tensor after cast to int64: paddle.int64
```

3.3 Tensor 的设备位置 (place)

初始化 Tensor 时可以通过 `Tensor.place` 来指定其分配的设备位置，可支持的设备位置有：CPU、GPU、固定内存、XPU（Baidu Kunlun）、NPU（Huawei）、MLU（寒武纪）、IPU（Graphcore）等。其中固定内存也称为不可分页内存或锁页内存，其与 GPU 之间具有更高的读写效率，并且支持异步传输，这对网络整体性能会有进一步提升，但其缺点是分配空间过多时可能会降低主机系统的性能，因为其减少了用于存储虚拟内存数据的可分页内存。

说明：

- 当未指定 place 时，Tensor 默认设备位置和安装的飞桨框架版本一致。如安装了 GPU 版本的飞桨，则设备位置默认为 GPU，即 Tensor 的 `place` 默认为 `paddle.CUDAPlace`。
- 使用 `paddle.device.set_device` 可设置全局默认的设备位置。`Tensor.place` 的指定值优先级高于全局默认值。

以下示例分别创建了 CPU、GPU 和固定内存上的 Tensor，并通过 `Tensor.place` 查看 Tensor 所在的设备位置：

- 创建 CPU 上的 Tensor

```
cpu_Tensor = paddle.to_tensor(1, place=paddle.CPUPlace())
print(cpu_Tensor.place)
```

```
Place(cpu)
```

- 创建 GPU 上的 Tensor

```
gpu_Tensor = paddle.to_tensor(1, place=paddle.CUDAPlace(0))
print(gpu_Tensor.place) # 显示 Tensor 位于 GPU 设备的第 0 张显卡上
```

```
Place(gpu:0)
```

- 创建固定内存上的 Tensor

```
pin_memory_Tensor = paddle.to_tensor(1, place=paddle.CUDAPinnedPlace())
print(pin_memory_Tensor.place)
```

```
Place(gpu_pinned)
```

3.4 Tensor 的名称 (name)

Tensor 的名称是其唯一的标识符，为 Python 字符串类型，查看一个 Tensor 的名称可以通过 `Tensor.name` 属性。默认地，在每个 Tensor 创建时，会自定义一个独一无二的名称。

```
print("Tensor name:", paddle.to_tensor(1).name)
```

```
Tensor name: generated_tensor_0
```

3.5 Tensor 的 stop_gradient 属性

`stop_gradient` 表示是否停止计算梯度，默认值为 `True`，表示停止计算梯度，梯度不再回传。在设计网络时，如不需要对某些参数进行训练更新，可以将参数的 `stop_gradient` 设置为 `True`。可参考以下代码直接设置 `stop_gradient` 的值。

```
eg = paddle.to_tensor(1)
print("Tensor stop_gradient:", eg.stop_gradient)
eg.stop_gradient = False
print("Tensor stop_gradient:", eg.stop_gradient)
```

```
Tensor stop_gradient: True
Tensor stop_gradient: False
```

四、Tensor 的操作

4.1 索引和切片

通过索引或切片方式可访问或修改 Tensor。飞桨框架使用标准的 Python 索引规则与 Numpy 索引规则，与 `Indexing a list or a string in Python` 类似。具有以下特点：

1. 基于 0-n 的下标进行索引，如果下标为负数，则从尾部开始计算。
2. 通过冒号：分隔切片参数，`start:stop:step` 来进行切片操作，其中 `start`、`stop`、`step` 均可缺省。

4.1.1 访问 Tensor

- 针对一维 Tensor，仅有单个维度上的索引或切片：

```
ndim_1_Tensor = paddle.to_tensor([0, 1, 2, 3, 4, 5, 6, 7, 8])
print("Origin Tensor:", ndim_1_Tensor.numpy()) # 原始1维Tensor
print("First element:", ndim_1_Tensor[0].numpy()) # 取Tensor第一个元素的值？
print("Last element:", ndim_1_Tensor[-1].numpy())
print("All element:", ndim_1_Tensor[:].numpy())
print("Before 3:", ndim_1_Tensor[:3].numpy())
print("From 6 to the end:", ndim_1_Tensor[6:].numpy())
print("From 3 to 6:", ndim_1_Tensor[3:6].numpy())
print("Interval of 3:", ndim_1_Tensor[::3].numpy())
print("Reverse:", ndim_1_Tensor[::-1].numpy())
```

```
Origin Tensor: [0 1 2 3 4 5 6 7 8]
First element: [0]
Last element: [8]
All element: [0 1 2 3 4 5 6 7 8]
Before 3: [0 1 2]
From 6 to the end: [6 7 8]
From 3 to 6: [3 4 5]
Interval of 3: [0 3 6]
Reverse: [8 7 6 5 4 3 2 1 0]
```

- 针对二维及以上的 Tensor，则会有多个维度上的索引或切片：

```
ndim_2_Tensor = paddle.to_tensor([[0, 1, 2, 3],
                                  [4, 5, 6, 7],
                                  [8, 9, 10, 11]])
print("Origin Tensor:", ndim_2_Tensor.numpy())
print("First row:", ndim_2_Tensor[0].numpy())
print("First row:", ndim_2_Tensor[0, :].numpy())
print("First column:", ndim_2_Tensor[:, 0].numpy())
print("Last column:", ndim_2_Tensor[:, -1].numpy())
print("All element:", ndim_2_Tensor[:].numpy())
print("First row and second column:", ndim_2_Tensor[0, 1].numpy())
```

```
Origin Tensor: [[ 0  1  2  3]
                [ 4  5  6  7]
                [ 8  9 10 11]]
First row: [0 1 2 3]
First row: [0 1 2 3]
First column: [0 4 8]
```

(下页继续)

(续上页)

```
Last column: [ 3  7 11]
All element: [[ 0  1  2  3]
               [ 4  5  6  7]
               [ 8  9 10 11]]
First row and second column: [1]
```

索引或切片的第一个值对应第 0 维，第二个值对应第 1 维，依次类推，如果某个维度上未指定索引，则默认为`:`。例如：

```
ndim_2_Tensor[1]
ndim_2_Tensor[1, :]
```

这两种操作的结果是完全相同的。

```
Tensor(shape=[4], dtype=int64, place=Place(gpu:0), stop_gradient=True,
       [4, 5, 6, 7])
```

4.1.2 修改 Tensor

与访问 Tensor 类似，修改 Tensor 可以在单个或多个维度上通过索引或切片操作。同时，支持将多种类型的数据赋值给该 Tensor，当前支持的数据类型有：`int`, `float`, `numpy.ndarray`, `omplex`, `Tensor`。

注意：

请慎重通过索引或切片修改 Tensor，该操作会原地修改该 Tensor 的数值，且原值不会被保存。如果被修改的 Tensor 参与梯度计算，仅会使用修改后的数值，这可能会给梯度计算引入风险。飞桨框架会自动检测不当的原位（`inplace`）使用并报错。

```
import numpy as np

x = paddle.to_tensor(np.ones((2, 3)).astype(np.float32)) # [[1., 1., 1.], [1., 1., 1.]]
# x : [[1., 1., 1.], [1., 1., 1.]]

x[0] = 0 # x : [[0., 0., 0.], [1., 1., 1.]]
# x : [[2.09999990, 2.09999990, 2.09999990], [1., 1., 1.]]
# x : [[3., 3., 3.], [3., 3., 3.]]

x[0:1] = 2.1 # x : [[1., 2., 3.], [3., 3., 3.]]
# x : [[2.09999990, 2.09999990, 2.09999990], [1., 1., 1.]]

x[...] = 3 # x : [[3., 3., 3.], [3., 3., 3.]]
```

```
x[0:1] = np.array([1,2,3]) # x : [[1., 2., 3.], [3., 3., 3.]]
```

```
x[1] = paddle.ones([3]) # x : [[1., 2., 3.], [1., 1., 1.]]
```

同时，飞桨还提供了丰富的 Tensor 操作的 API，包括数学运算、逻辑运算、线性代数等 100 余种 API，这些 API 调用有两种方法：

```
x = paddle.to_tensor([[1.1, 2.2], [3.3, 4.4]], dtype="float64")
y = paddle.to_tensor([[5.5, 6.6], [7.7, 8.8]], dtype="float64")

print(paddle.add(x, y), "\n") # 方法一
print(x.add(y), "\n") # 方法二
```

```
Tensor(shape=[2, 2], dtype=float64, place=Place(gpu:0), stop_gradient=True,
       [[6.60000000, 8.80000000],
        [11.00000000, 13.20000000]])

Tensor(shape=[2, 2], dtype=float64, place=Place(gpu:0), stop_gradient=True,
       [[6.60000000, 8.80000000],
        [11.00000000, 13.20000000]])
```

可以看出，使用 **Tensor** 类成员函数和 **Paddle API** 具有相同的效果，由于 **类成员函数**操作更为方便，以下均从 **Tensor** 类成员函数的角度，对常用 Tensor 操作进行介绍。

4.2 数学运算

x.abs()	#逐元素取绝对值
x.ceil()	#逐元素向上取整
x.floor()	#逐元素向下取整
x.round()	#逐元素四舍五入
x.exp()	#逐元素计算自然常数为底的指数
x.log()	#逐元素计算x的自然对数
x.reciprocal()	#逐元素求倒数
x.square()	#逐元素计算平方
x.sqrt()	#逐元素计算平方根
x.sin()	#逐元素计算正弦
x.cos()	#逐元素计算余弦
x.add(y)	#逐元素相加
x.subtract(y)	#逐元素相减
x.multiply(y)	#逐元素相乘
x.divide(y)	#逐元素相除
x.mod(y)	#逐元素相除并取余
x.pow(y)	#逐元素幂运算
x.max()	#指定维度上元素最大值，默认为全部维度
x.min()	#指定维度上元素最小值，默认为全部维度
x.prod()	#指定维度上元素累乘，默认为全部维度
x.sum()	#指定维度上元素的和，默认为全部维度

飞桨框架对 Python 数学运算相关的魔法函数进行了重写，例如：

<code>x + y</code>	<code>-> x.add(y)</code>	<code>#逐元素相加</code>
<code>x - y</code>	<code>-> x.subtract(y)</code>	<code>#逐元素相减</code>
<code>x * y</code>	<code>-> x.multiply(y)</code>	<code>#逐元素相乘</code>
<code>x / y</code>	<code>-> x.divide(y)</code>	<code>#逐元素相除</code>
<code>x % y</code>	<code>-> x.mod(y)</code>	<code>#逐元素相除并取余</code>
<code>x ** y</code>	<code>-> x.pow(y)</code>	<code>#逐元素幂运算</code>

4.3 逻辑运算

<code>x.isfinite()</code>	<code>#判断Tensor中元素是否是有限的数字，即不包括inf与nan</code>
<code>x.equal_all(y)</code>	<code>#判断两个Tensor的全部元素是否相等，并返回形状为[1]的布尔类Tensor</code>
<code>x.equal(y)</code>	<code>#判断两个Tensor的每个元素是否相等，并返回形状相同的布尔类Tensor</code>
<code>x.not_equal(y)</code>	<code>#判断两个Tensor的每个元素是否不相等</code>
<code>x.less_than(y)</code>	<code>#判断Tensor x的元素是否小于Tensor y的对应元素</code>
<code>x.less_equal(y)</code>	<code>#判断Tensor x的元素是否小于或等于Tensor y的对应元素</code>
<code>x.greater_than(y)</code>	<code>#判断Tensor x的元素是否大于Tensor y的对应元素</code>
<code>x.greater_equal(y)</code>	<code>#判断Tensor x的元素是否大于或等于Tensor y的对应元素</code>
<code>x.allclose(y)</code>	<code>#判断Tensor x的全部元素是否与Tensor y的全部元素接近，并返回形状为[1]的布尔类Tensor</code>

同样地，飞桨框架对 Python 逻辑比较相关的魔法函数进行了重写，以下操作与上述结果相同。

<code>x == y</code>	<code>-> x.equal(y)</code>	<code>#判断两个Tensor的每个元素是否相等</code>
<code>x != y</code>	<code>-> x.not_equal(y)</code>	<code>#判断两个Tensor的每个元素是否不相等</code>
<code>x < y</code>	<code>-> x.less_than(y)</code>	<code>#判断Tensor x的元素是否小于Tensor y的对应元素</code>
<code>x <= y</code>	<code>-> x.less_equal(y)</code>	<code>#判断Tensor x的元素是否小于或等于Tensor y的对应元素</code>
<code>x > y</code>	<code>-> x.greater_than(y)</code>	<code>#判断Tensor x的元素是否大于Tensor y的对应元素</code>
<code>x >= y</code>	<code>-> x.greater_equal(y)</code>	<code>#判断Tensor x的元素是否大于或等于Tensor y的对应元素</code>

以下操作仅针对 bool 型 Tensor：

<code>x.logical_and(y)</code>	<code>#对两个布尔类型Tensor逐元素进行逻辑与操作</code>
<code>x.logical_or(y)</code>	<code>#对两个布尔类型Tensor逐元素进行逻辑或操作</code>
<code>x.logical_xor(y)</code>	<code>#对两个布尔类型Tensor逐元素进行逻辑亦或操作</code>
<code>x.logical_not(y)</code>	<code>#对两个布尔类型Tensor逐元素进行逻辑非操作</code>

4.4 线性代数

```

x.t()                      #矩阵转置
x.transpose([1, 0])        #交换第 0 维与第 1 维的顺序
x.norm('fro')              #矩阵的弗罗贝尼乌斯范数
x.dist(y, p=2)             #矩阵 (x-y) 的 2 范数
x.matmul(y)                #矩阵乘法

```

注意

以上计算 API 也有原位 (inplace) 操作和非原位操作之分，如 `x.add(y)` 是非原位操作，`x.add_(y)` 为原位操作。

五、Tensor 的广播机制

在深度学习任务中，有时需要使用较小形状的 Tensor 与较大形状的 Tensor 执行计算，广播机制就是将较小形状的 Tensor 扩展到与较大形状的 Tensor 一样的形状，便于匹配计算，同时又没有对较小形状 Tensor 进行数据拷贝操作，从而提升算法实现的运算效率。飞桨框架提供的一些 API 支持广播（broadcasting）机制，允许在一些运算时使用不同形状的 Tensor。飞桨 Tensor 的广播机制主要遵循如下规则（参考 [Numpy 广播机制](#)）：

- 每个 Tensor 至少为一维 Tensor
- 从最后一个维度向前开始比较两个 Tensor 的形状，需要满足如下条件才能进行广播：两个 Tensor 的维度大小相等；或者其中一个 Tensor 的维度等于 1；或者其中一个 Tensor 的维度不存在。

举例如下：

```

# 可以广播的例子1
x = paddle.ones((2, 3, 4))
y = paddle.ones((2, 3, 4))
# 两个Tensor 形状一致，可以广播
z = x + y
print(z.shape)
# [2, 3, 4]

```

```

# 可以广播的例子2
x = paddle.ones((2, 3, 1, 5))
y = paddle.ones((3, 4, 1))
# 从最后一个维度向前依次比较：
# 第一次：y的维度大小是1
# 第二次：x的维度大小是1
# 第三次：x和y的维度大小相等
# 第四次：y的维度不存在
# 所以 x和y是可以广播的
z = x + y

```

(下页继续)

(续上页)

```
print(z.shape)
# [2, 3, 4, 5]
```

```
# 不可广播的例子
x = paddle.ones((2, 3, 4))
y = paddle.ones((2, 3, 6))
# 此时x和y是不可广播的，因为第一次比较：4不等于6
# z = x + y
# ValueError: (InvalidArgumentException) Broadcast dimension mismatch.
```

在了解两个 Tensor 在什么情况下可以广播的规则后，两个 Tensor 进行广播语义后的结果 Tensor 的形状计算规则如下：

- 如果两个 Tensor 的形状的长度不一致，会在较小长度的形状矩阵前部添加 1，直到两个 Tensor 的形状长度相等。
- 保证两个 Tensor 形状相等之后，每个维度上的结果维度就是当前维度上的较大值。

举例如下：

```
x = paddle.ones((2, 1, 4))
y = paddle.ones((3, 1)) #  
→y的形状长度为2，小于x的形状长度3，因此会在y的形状前部添加1，结果就是y的形状变为[1, ←  
→3, 1]
z = x + y
print(z.shape)
# z的形状：[2, 3,  
→4]，z的每一维度上的尺寸，将取x和y对应维度上尺寸的较大值，如第0维x的尺寸为2，y的尺寸为1，则z的第0维
```

六、Tensor 与 Numpy 数组相互转换

如果你已熟悉 Numpy，通过以下要点，可以方便地理解和迁移到 Tensor 的使用上：

- Tensor 的很多基础操作 API 和 Numpy 在功能、用法上基本保持一致。如前文中介绍的指定数据、形状、区间创建 Tensor，Tensor 的形状、数据类型属性，Tensor 的各种操作，以及 Tensor 的广播，可以很方便地在 Numpy 中找到相似操作。
- 但是，Tensor 也有一些独有的属性和操作，而 Numpy 中没有对应概念或功能，这是为了更好地支持深度学习任务。如前文中介绍的通过图像、文本等原始数据手动或自动创建 Tensor 的功能，能够更便捷地处理数据，Tensor 的设备位置属性，可以很方便地将 Tensor 迁移到 GPU 或各种 AI 加速硬件上，Tensor 的 stop_gradient 属性，也是 Tensor 独有的，以便更好地支持深度学习任务。

如果有 Numpy 数组，可使用 `paddle.to_tensor` 创建任意维度的 Tensor，创建的 Tensor 与原 Numpy 数组具有相同的形状与数据类型。

```
tensor_temp = paddle.to_tensor(np.array([1.0, 2.0]))
print(tensor_temp)
```

```
Tensor(shape=[2], dtype=float64, place=Place(gpu:0), stop_gradient=True,
       [1., 2.])
```

注意：

- 基于 Numpy 数组创建 Tensor 时，飞桨是通过拷贝方式创建，与原始数据不共享内存。

相对应地，飞桨也支持将 Tensor 转换为 Numpy 数组，可通过 `Tensor.numpy` 方法实现。

```
tensor_to_convert = paddle.to_tensor([1., 2.])
tensor_to_convert.numpy()
```

```
array([1., 2.], dtype=float32)
```

七、总结

Tensor 作为飞桨框架中重要的数据结构，具有丰富的 API 用以对 Tensor 进行创建、访问、修改、计算等一系列操作，从而满足深度学习任务的需要。更多 Tensor 相关的介绍，请参考 `paddle.Tensor API` 文档。

2.1.2 模型保存与载入

Title overline too short.

```
#####
模型保存与载入
#####
```

一、保存载入体系简介

Title underline too short.

```
一、保存载入体系简介
#####
```

1.1 基础 API 保存载入体系

Title underline too short.

1.1 基础 API 保存载入体系

飞桨框架 2.1 对模型与参数的保存与载入相关接口进行了梳理：对于训练调优场景，我们推荐使用 paddle.save/load 保存和载入模型；对于推理部署场景，我们推荐使用 paddle.jit.save/load（动态图）和 paddle.static.save/load_inference_model（静态图）保存载入模型。

飞桨保存载入相关接口包括：

paddle.save

paddle.load

paddle.jit.save

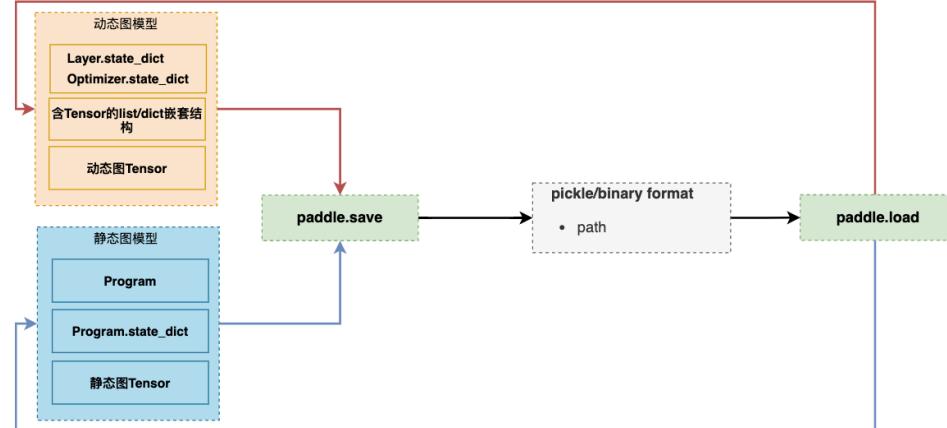
paddle.jit.load

paddle.static.save_inference_model

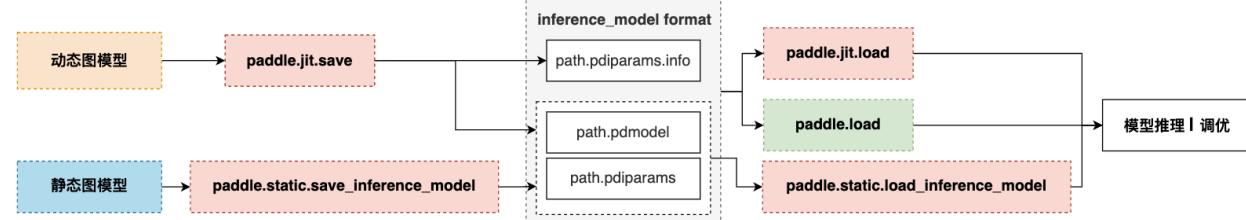
paddle.static.load_inference_model

各接口关系如下图所示：

训练调优



模型推理



1.2 高阶 API 保存载入体系

Title underline too short.

1.2 高阶 API 保存载入体系

- paddle.Model.fit (训练接口，同时带有参数保存的功能)
- paddle.Model.save
- paddle.Model.load

飞桨框架 2.0 高阶 API 仅有一套 Save/Load 接口，表意直观，体系清晰，若有需要，建议直接阅读相关 API 文档，此处不再赘述。

备注：本教程着重介绍飞桨框架 2.1 的各个保存载入接口的关系及各种使用场景，不对接口参数进行详细介绍，如果需要了解具体接口参数的含义，请直接阅读对应 API 文档。

模型保存常见问题

二、训练调优场景的模型 & 参数保存载入

Title underline too short.

二、训练调优场景的模型&参数保存载入

2.1 动态图参数保存载入

Title underline too short.

2.1 动态图参数保存载入

若仅需要保存/载入模型的参数，可以使用 paddle.save/load 结合 Layer 和 Optimizer 的 state_dict 达成目的，此处 state_dict 是对象的持久参数的载体，dict 的 key 为参数名，value 为参数真实的 numpy array 值。

结合以下简单示例，介绍参数保存和载入的方法，以下示例完成了一个简单网络的训练过程：

```
import numpy as np
import paddle
import paddle.nn as nn
import paddle.optimizer as opt
```

(下页继续)

(续上页)

```

BATCH_SIZE = 16
BATCH_NUM = 4
EPOCH_NUM = 4

IMAGE_SIZE = 784
CLASS_NUM = 10

# define a random dataset
class RandomDataset(paddle.io.Dataset):
    def __init__(self, num_samples):
        self.num_samples = num_samples

    def __getitem__(self, idx):
        image = np.random.random([IMAGE_SIZE]).astype('float32')
        label = np.random.randint(0, CLASS_NUM - 1, (1, )).astype('int64')
        return image, label

    def __len__(self):
        return self.num_samples

class LinearNet(nn.Layer):
    def __init__(self):
        super(LinearNet, self).__init__()
        self._linear = nn.Linear(IMAGE_SIZE, CLASS_NUM)

    def forward(self, x):
        return self._linear(x)

def train(layer, loader, loss_fn, opt):
    for epoch_id in range(EPOCH_NUM):
        for batch_id, (image, label) in enumerate(loader()):
            out = layer(image)
            loss = loss_fn(out, label)
            loss.backward()
            opt.step()
            opt.clear_grad()
            print("Epoch {} batch {}: loss = {}".format(
                epoch_id, batch_id, np.mean(loss.numpy())))

```

create network

```

layer = LinearNet()
loss_fn = nn.CrossEntropyLoss()

```

(下页继续)

(续上页)

```

adam = opt.Adam(learning_rate=0.001, parameters=layer.parameters())

# create data loader
dataset = RandomDataset(BATCH_NUM * BATCH_SIZE)
loader = paddle.io.DataLoader(dataset,
    batch_size=BATCH_SIZE,
    shuffle=True,
    drop_last=True,
    num_workers=2)

# train
train(layer, loader, loss_fn, adam)

```

2.1.1 参数保存

Title underline too short.

2.1.1 参数保存

参数保存时，先获取目标对象（Layer 或者 Optimzier）的 state_dict，然后将 state_dict 保存至磁盘，示例如下（接前述示例）：

```

# save
paddle.save(layer.state_dict(), "linear_net.pdparams")
paddle.save(adam.state_dict(), "adam.pdopt")

```

2.1.2 参数载入

Title underline too short.

2.1.2 参数载入

参数载入时，先从磁盘载入保存的 state_dict，然后通过 set_state_dict 方法配置到目标对象中，示例如下（接前述示例）：

```

# load
layer_state_dict = paddle.load("linear_net.pdparams")
opt_state_dict = paddle.load("adam.pdopt")

```

(下页继续)

(续上页)

```
layer.set_state_dict(layer_state_dict)
adam.set_state_dict(opt_state_dict)
```

2.2 静态图模型 & 参数保存载入

Title underline too short.

2.2 静态图模型 & 参数保存载入

若仅需要保存/载入模型的参数，可以使用 `paddle.save/load` 结合 Program 的 `state_dict` 达成目的，此处 `state_dict` 与动态图 `state_dict` 概念类似，`dict` 的 `key` 为参数名，`value` 为参数真实的值。若想保存整个模型，需要使用“`paddle.save`”将 Program 和 `state_dict` 都保存下来。

结合以下简单示例，介绍参数保存和载入的方法：

```
import paddle
import paddle.static as static

paddle.enable_static()

# create network
x = paddle.static.data(name="x", shape=[None, 224], dtype='float32')
z = paddle.static.nn.fc(x, 10)

place = paddle.CPUPlace()
exe = paddle.static.Executor(place)
exe.run(paddle.static.default_startup_program())
prog = paddle.static.default_main_program()
```

2.2.1 静态图模型 & 参数保存

Title underline too short.

2.2.1 静态图模型 & 参数保存

参数保存时，先获取 Program 的 `state_dict`，然后将 `state_dict` 保存至磁盘，示例如下（接前述示例）：

```
paddle.save(prog.state_dict(), "temp/model.pdparams")
```

如果想要保存整个静态图模型，除了 `state_dict` 还需要保存 Program

```
paddle.save(prog, "temp/model.pdmodel")
```

2.2.2 静态图模型 & 参数载入

Title underline too short.

2.2.2 静态图模型 & 参数载入

如果只保存了 state_dict，可以跳过此段代码，直接载入 state_dict。如果模型文件中包含 Program 和 state_dict，请先载入 Program，示例如下（接前述示例）：

```
prog = paddle.load("temp/model.pdmodel")
```

参数载入时，先从磁盘载入保存的 state_dict，然后通过 set_state_dict 方法配置到 Program 中，示例如下（接前述示例）：

```
state_dict = paddle.load("temp/model.pdparams")
prog.set_state_dict(state_dict)
```

三、训练部署场景的模型 & 参数保存载入

Title underline too short.

三、训练部署场景的模型 & 参数保存载入

```
#####
#####
```

3.1 动态图模型 & 参数保存载入（训练推理）

Title underline too short.

3.1 动态图模型 & 参数保存载入（训练推理）

若要同时保存/载入动态图模型结构和参数，可以使用 paddle.jit.save/load 实现。

3.1.1 动态图模型 & 参数保存

Title underline too short.

3.1.1 动态图模型 & 参数保存

模型 & 参数存储根据训练模式不同，有两种使用情况：

- (1) 动转静训练 + 模型 & 参数保存
- (2) 动态图训练 + 模型 & 参数保存

3.1.1.1 动转静训练 + 模型 & 参数保存

Title underline too short.

3.1.1.1.1 动转静训练 + 模型 & 参数保存

动转静训练相比直接使用动态图训练具有更好的执行性能，训练完成后，直接将目标 Layer 传入 paddle.jit.save 保存即可。：

一个简单的网络训练示例如下：

```
import numpy as np
import paddle
import paddle.nn as nn
import paddle.optimizer as opt

BATCH_SIZE = 16
BATCH_NUM = 4
EPOCH_NUM = 4

IMAGE_SIZE = 784
CLASS_NUM = 10

# define a random dataset
class RandomDataset(paddle.io.Dataset):
    def __init__(self, num_samples):
        self.num_samples = num_samples

    def __getitem__(self, idx):
        image = np.random.random([IMAGE_SIZE]).astype('float32')
        label = np.random.randint(0, CLASS_NUM - 1, (1, )).astype('int64')
```

(下页继续)

(续上页)

```
    return image, label

    def __len__(self):
        return self.num_samples

class LinearNet(nn.Layer):
    def __init__(self):
        super(LinearNet, self).__init__()
        self._linear = nn.Linear(IMAGE_SIZE, CLASS_NUM)

    @paddle.jit.to_static
    def forward(self, x):
        return self._linear(x)

def train(layer, loader, loss_fn, opt):
    for epoch_id in range(EPOCH_NUM):
        for batch_id, (image, label) in enumerate(loader()):
            out = layer(image)
            loss = loss_fn(out, label)
            loss.backward()
            opt.step()
            opt.clear_grad()
            print("Epoch {} batch {}: loss = {}".format(
                epoch_id, batch_id, np.mean(loss.numpy())))

# create network
layer = LinearNet()
loss_fn = nn.CrossEntropyLoss()
adam = opt.Adam(learning_rate=0.001, parameters=layer.parameters())

# create data loader
dataset = RandomDataset(BATCH_NUM * BATCH_SIZE)
loader = paddle.io.DataLoader(dataset,
    batch_size=BATCH_SIZE,
    shuffle=True,
    drop_last=True,
    num_workers=2)

# train
train(layer, loader, loss_fn, adam)
```

随后使用 `paddle.jit.save` 对模型和参数进行存储（接前述示例）：

```
# save
path = "example.model/linear"
paddle.jit.save(layer, path)
```

通过动静训练后保存模型 & 参数，有以下三项注意点：

- (1) Layer 对象的 forward 方法需要经由 paddle.jit.to_static 装饰

经过 paddle.jit.to_static 装饰 forward 方法后，相应 Layer 在执行时，会先生成描述模型的 Program，然后通过执行 Program 获取计算结果，示例如下：

```
import paddle
import paddle.nn as nn

IMAGE_SIZE = 784
CLASS_NUM = 10

class LinearNet(nn.Layer):
    def __init__(self):
        super(LinearNet, self).__init__()
        self._linear = nn.Linear(IMAGE_SIZE, CLASS_NUM)

    @paddle.jit.to_static
    def forward(self, x):
        return self._linear(x)
```

若最终需要生成的描述模型的 Program 支持动态输入，可以同时指明模型的 InputSpec，示例如下：

```
import paddle
import paddle.nn as nn
from paddle.static import InputSpec

IMAGE_SIZE = 784
CLASS_NUM = 10

class LinearNet(nn.Layer):
    def __init__(self):
        super(LinearNet, self).__init__()
        self._linear = nn.Linear(IMAGE_SIZE, CLASS_NUM)

    @paddle.jit.to_static(input_spec=[InputSpec(shape=[None, 784], dtype='float32')])
    def forward(self, x):
        return self._linear(x)
```

- (2) 请确保 Layer.forward 方法中仅实现预测功能，避免将训练所需的 loss 计算逻辑写入 forward 方法

Layer 更准确的语义是描述一个具有预测功能的模型对象，接收输入的样本数据，输出预测的结果，而 loss 计

算是仅属于模型训练中的概念。将 loss 计算的实现放到 Layer.forward 方法中，会使 Layer 在不同场景下概念有所差别，并且增大 Layer 使用的复杂性，这不是良好的编码行为，同时也会在最终保存预测模型时引入剪枝的复杂性，因此建议保持 Layer 实现的简洁性，下面通过两个示例对比说明：

错误示例如下：

```
import paddle
import paddle.nn as nn

IMAGE_SIZE = 784
CLASS_NUM = 10

class LinearNet(nn.Layer):
    def __init__(self):
        super(LinearNet, self).__init__()
        self._linear = nn.Linear(IMAGE_SIZE, CLASS_NUM)

    @paddle.jit.to_static
    def forward(self, x, label=None):
        out = self._linear(x)
        if label:
            loss = nn.functional.cross_entropy(out, label)
            avg_loss = nn.functional.mean(loss)
            return out, avg_loss
        else:
            return out
```

正确示例如下：

```
import paddle
import paddle.nn as nn

IMAGE_SIZE = 784
CLASS_NUM = 10

class LinearNet(nn.Layer):
    def __init__(self):
        super(LinearNet, self).__init__()
        self._linear = nn.Linear(IMAGE_SIZE, CLASS_NUM)

    @paddle.jit.to_static
    def forward(self, x):
        return self._linear(x)
```

- (3) 如果你需要保存多个方法，需要用 paddle.jit.to_static 装饰每一个需要被保存的方法。

备注：只有在 forward 之外还需要保存其他方法时才用这个特性，如果仅装饰非 forward 的方法，而 forward 没有被装饰，是不符合规范的。此时 paddle.jit.save 的 input_spec 参数必须为 None。

示例代码如下：

```

import paddle
import paddle.nn as nn
from paddle.static import InputSpec

IMAGE_SIZE = 784
CLASS_NUM = 10

class LinearNet(nn.Layer):
    def __init__(self):
        super(LinearNet, self).__init__()
        self._linear = nn.Linear(IMAGE_SIZE, CLASS_NUM)
        self._linear_2 = nn.Linear(IMAGE_SIZE, CLASS_NUM)

    @paddle.jit.to_static(input_spec=[InputSpec(shape=[None, IMAGE_SIZE], dtype=
→'float32')])
    def forward(self, x):
        return self._linear(x)

    @paddle.jit.to_static(input_spec=[InputSpec(shape=[None, IMAGE_SIZE], dtype=
→'float32')])
    def another_forward(self, x):
        return self._linear_2(x)

inps = paddle.randn([1, IMAGE_SIZE])
layer = LinearNet()
before_0 = layer.another_forward(inps)
before_1 = layer(inps)
# save and load
path = "example.model/linear"
paddle.jit.save(layer, path)

```

保存的模型命名规则：forward 的模型名字为：模型名 + 后缀，其他函数的模型名字为：模型名 + 函数名 + 后缀。每个函数有各自的 pdmodel 和 pdiparams 的文件，所有函数共用 pdiparams.info。上述代码将在 example.model 文件夹下产生 5 个文件：linear.another_forward.pdiparams、linear.pdiparams、linear.pdmodel、linear.another_forward.pdmodel、linear.pdiparams.info

- (4) 当使用 jit.save 保存函数时，jit.save 只保存这个函数对应的静态图 Program，不会保存和这个函数相关的参数。如果你必须保存参数，请使用 Layer 封装这个函数。

示例代码如下：

```
def fun(inputs):
    return paddle.tanh(inputs)

path = 'func/model'
inps = paddle.rand([3, 6])
origin = fun(inps)

paddle.jit.save(
    fun,
    path,
    input_spec=[
        InputSpec(
            shape=[None, 6], dtype='float32', name='x'),
    ])
load_func = paddle.jit.load(path)
load_result = load_func(inps)
```

3.1.1.2 动态图训练 + 模型 & 参数保存

Title underline too short.

3.1.1.2 动态图训练 + 模型&参数保存

动态图模式相比动转静模式更加便于调试，如果你仍需要使用动态图直接训练，也可以在动态图训练完成后调用 `paddle.jit.save` 直接保存模型和参数。

同样是一个简单的网络训练示例：

```
import numpy as np
import paddle
import paddle.nn as nn
import paddle.optimizer as opt
from paddle.static import InputSpec

BATCH_SIZE = 16
BATCH_NUM = 4
EPOCH_NUM = 4

IMAGE_SIZE = 784
CLASS_NUM = 10
```

(下页继续)

(续上页)

```

# define a random dataset
class RandomDataset(paddle.io.Dataset):
    def __init__(self, num_samples):
        self.num_samples = num_samples

    def __getitem__(self, idx):
        image = np.random.random([IMAGE_SIZE]).astype('float32')
        label = np.random.randint(0, CLASS_NUM - 1, (1, )).astype('int64')
        return image, label

    def __len__(self):
        return self.num_samples

class LinearNet(nn.Layer):
    def __init__(self):
        super(LinearNet, self).__init__()
        self._linear = nn.Linear(IMAGE_SIZE, CLASS_NUM)

    def forward(self, x):
        return self._linear(x)

def train(layer, loader, loss_fn, opt):
    for epoch_id in range(EPOCH_NUM):
        for batch_id, (image, label) in enumerate(loader()):
            out = layer(image)
            loss = loss_fn(out, label)
            loss.backward()
            opt.step()
            opt.clear_grad()
            print("Epoch {} batch {}: loss = {}".format(
                epoch_id, batch_id, np.mean(loss.numpy())))

```

```

# create network
layer = LinearNet()
loss_fn = nn.CrossEntropyLoss()
adam = opt.Adam(learning_rate=0.001, parameters=layer.parameters())

# create data loader
dataset = RandomDataset(BATCH_NUM * BATCH_SIZE)
loader = paddle.io.DataLoader(dataset,
    batch_size=BATCH_SIZE,
    shuffle=True,
    drop_last=True,

```

(下页继续)

(续上页)

```
num_workers=2)

# train
train(layer, loader, loss_fn, adam)
```

训练完成后使用 `paddle.jit.save` 对模型和参数进行存储：

```
# save
path = "example.dy_model/linear"
paddle.jit.save(
    layer=layer,
    path=path,
    input_spec=[InputSpec(shape=[None, 784], dtype='float32')])
```

动态图训练后使用 `paddle.jit.save` 保存模型和参数注意点如下：

- (1) 相比动转静训练，`Layer` 对象的 `forward` 方法不需要额外装饰，保持原实现即可
- (2) 与动转静训练相同，请确保 `Layer.forward` 方法中仅实现预测功能，避免将训练所需的 `loss` 计算逻辑写入 `forward` 方法
- (3) 在最后使用 `paddle.jit.save` 时，需要指定 `Layer` 的 `InputSpec`，`Layer` 对象 `forward` 方法的每一个参数均需要对应的 `InputSpec` 进行描述，不能省略。这里的 `input_spec` 参数支持两种类型的输入：
 - `InputSpec` 列表

使用 `InputSpec` 描述 `forward` 输入参数的 `shape`, `dtype` 和 `name`，如前述示例（此处示例中 `name` 省略，`name` 省略的情况下会使用 `forward` 的对应参数名作为 `name`，所以这里的 `name` 为 `x`）：

```
paddle.jit.save(
    layer=layer,
    path=path,
    input_spec=[InputSpec(shape=[None, 784], dtype='float32')])
```

- `Example Tensor` 列表

除使用 `InputSpec` 之外，也可以直接使用 `forward` 训练时的示例输入，此处可以使用前述示例中迭代 `DataLoader` 得到的 `image`，示例如下：

```
paddle.jit.save(
    layer=layer,
    path=path,
    input_spec=[image])
```

3.1.2 动态图模型 & 参数载入

Title underline too short.

3.1.2 动态图模型 & 参数载入

载入模型参数，使用 `paddle.jit.load` 载入即可，载入后得到的是一个 Layer 的派生类对象 `TranslatedLayer`，`TranslatedLayer` 具有 Layer 具有的通用特征，支持切换 `train` 或者 `eval` 模式，可以进行模型调优或者预测。

备注：为了规避变量名字冲突，载入之后会重命名变量。

载入模型及参数，示例如下：

```
import numpy as np
import paddle
import paddle.nn as nn
import paddle.optimizer as opt

BATCH_SIZE = 16
BATCH_NUM = 4
EPOCH_NUM = 4

IMAGE_SIZE = 784
CLASS_NUM = 10

# load
path = "example.model/linear"
loaded_layer = paddle.jit.load(path)
```

载入模型及参数后进行预测，示例如下（接前述示例）：

```
# inference
loaded_layer.eval()
x = paddle.randn([1, IMAGE_SIZE], 'float32')
pred = loaded_layer(x)
```

载入模型及参数后进行调优，示例如下（接前述示例）：

```
# define a random dataset
class RandomDataset(paddle.io.Dataset):
    def __init__(self, num_samples):
        self.num_samples = num_samples
```

(下页继续)

(续上页)

```

def __getitem__(self, idx):
    image = np.random.random([IMAGE_SIZE]).astype('float32')
    label = np.random.randint(0, CLASS_NUM - 1, (1, )).astype('int64')
    return image, label

def __len__(self):
    return self.num_samples

def train(layer, loader, loss_fn, opt):
    for epoch_id in range(EPOCH_NUM):
        for batch_id, (image, label) in enumerate(loader()):
            out = layer(image)
            loss = loss_fn(out, label)
            loss.backward()
            opt.step()
            opt.clear_grad()
            print("Epoch {} batch {}: loss = {}".format(
                epoch_id, batch_id, np.mean(loss.numpy())))
    # fine-tune
    loaded_layer.train()
    dataset = RandomDataset(BATCH_NUM * BATCH_SIZE)
    loader = paddle.io.DataLoader(dataset,
        batch_size=BATCH_SIZE,
        shuffle=True,
        drop_last=True,
        num_workers=2)
    loss_fn = nn.CrossEntropyLoss()
    adam = opt.Adam(learning_rate=0.001, parameters=loaded_layer.parameters())
    train(loaded_layer, loader, loss_fn, adam)
    # save after fine-tuning
    paddle.jit.save(loaded_layer, "fine-tune.model/linear", input_spec=[x])

```

此外，paddle.jit.save 同时保存了模型和参数，如果你只需要从存储结果中载入模型的参数，可以使用 paddle.load 接口载入，返回所存储模型的 state_dict，示例如下：

```

import paddle
import paddle.nn as nn

IMAGE_SIZE = 784
CLASS_NUM = 10

class LinearNet(nn.Layer):

```

(下页继续)

(续上页)

```

def __init__(self):
    super(LinearNet, self).__init__()
    self._linear = nn.Linear(IMAGE_SIZE, CLASS_NUM)

    @paddle.jit.to_static
    def forward(self, x):
        return self._linear(x)

# create network
layer = LinearNet()

# load
path = "example.model/linear"
state_dict = paddle.load(path)

# inference
layer.set_state_dict(state_dict, use_structured_name=False)
layer.eval()
x = paddle.randn([1, IMAGE_SIZE], 'float32')
pred = layer(x)

```

3.2 静态图模型 & 参数保存载入（推理部署）

Title underline too short.

3.2 静态图模型&参数保存载入（推理部署）

保存/载入静态图推理模型，可以通过 paddle.static.save/load_inference_model 实现。示例如下：

```

import paddle
import numpy as np

paddle.enable_static()

# Build the model
startup_prog = paddle.static.default_startup_program()
main_prog = paddle.static.default_main_program()
with paddle.static.program_guard(main_prog, startup_prog):
    image = paddle.static.data(name="img", shape=[64, 784])
    w = paddle.create_parameter(shape=[784, 200], dtype='float32')
    b = paddle.create_parameter(shape=[200], dtype='float32')
    hidden_w = paddle.matmul(x=image, y=w)

```

(下页继续)

(续上页)

```
hidden_b = paddle.add(hidden_w, b)
exe = paddle.static.Executor(paddle.CPUPlace())
exe.run(startup_prog)
```

3.2.1 静态图推理模型 & 参数保存

Title underline too short.

3.2.1 静态图推理模型&参数保存

静态图导出推理模型需要指定导出路径、输入、输出变量以及执行器。`save_inference_model` 会裁剪 Program 的冗余部分，并导出两个文件：`path_prefix.pdmodel`、`path_prefix.pdiparams`。示例如下（接前述示例）：

```
# Save the inference model
path_prefix = "./infer_model"
paddle.static.save_inference_model(path_prefix, [image], [hidden_b], exe)
```

3.2.2 静态图推理模型 & 参数载入

Title underline too short.

3.2.2 静态图推理模型&参数载入

载入静态图推理模型时，输入给 `load_inference_model` 的路径必须与 `save_inference_model` 的一致。示例如下（接前述示例）：

```
[inference_program, feed_target_names, fetch_targets] = (
    paddle.static.load_inference_model(path_prefix, exe))
tensor_img = np.array(np.random.random((64, 784)), dtype=np.float32)
results = exe.run(inference_program,
                  feed={feed_target_names[0]: tensor_img},
                  fetch_list=fetch_targets)
```

四、旧保存格式兼容载入

Title underline too short.

```
四、旧保存格式兼容载入
#####
```

如果你是从飞桨框架 1.x 切换到 2.1，曾经使用飞桨框架 1.x 的 fluid 相关接口保存模型或者参数，飞桨框架 2.1 也对这种情况进行了兼容性支持，请参考[兼容载入旧格式模型](#)

2.2 模型开发更多用法

- 模型可视化
- 自动微分
- 层与模型
- 自定义 Loss、Metric 及 Callback
- 梯度裁剪
- 模型导出 ONNX 协议

2.2.1 模型可视化

概述

VisualDL 是一个面向深度学习任务设计的可视化工具。VisualDL 利用了丰富的图表来展示数据，用户可以更直观、清晰地查看数据的特征与变化趋势，有助于分析数据、及时发现错误，进而改进神经网络模型的设计。

目前，VisualDL 支持 scalar, image, audio, graph, histogram, pr curve, high dimensional 七个组件，项目正处于高速迭代中，敬请期待新组件的加入。

Scalar -- 折线图组件

介绍

Scalar 组件的输入数据类型为标量，该组件的作用是将训练参数以折线图形式呈现。将损失函数值、准确率等标量数据作为参数传入 scalar 组件，即可画出折线图，便于观察变化趋势。

记录接口

Scalar 组件的记录接口如下：

```
add_scalar(tag, value, step, walltime=None)
```

接口参数说明如下：

Demo

- 基础使用

下面展示了使用 Scalar 组件记录数据的示例，代码文件请见Scalar 组件

```
from visualdl import LogWriter

if __name__ == '__main__':
    value = [i/1000.0 for i in range(1000)]
    # 初始化一个记录器
    with LogWriter(logdir='./log/scalar_test/train') as writer:
        for step in range(1000):
            # 向记录器添加一个tag为`acc`的数据
            writer.add_scalar(tag="acc", step=step, value=value[step])
            # 向记录器添加一个tag为`loss`的数据
            writer.add_scalar(tag="loss", step=step, value=1/(value[step] + 1))
```

运行上述程序后，在命令行执行

```
visualdl --logdir ./log --port 8080
```

接着在浏览器打开 <http://127.0.0.1:8080>，即可查看以下折线图。

- 多组实验对比

下面展示了使用 Scalar 组件实现多组实验对比

多组实验对比的实现分为两步：

- 创建子日志文件储存每组实验的参数数据
- 将数据写入 scalar 组件时，使用相同的 tag，即可实现对比不同实验的同一类型参数

```
from visualdl import LogWriter

if __name__ == '__main__':
    value = [i/1000.0 for i in range(1000)]
    # 步骤一： 创建父文件夹：log与子文件夹：scalar_test
```

(下页继续)

(续上页)

```

with LogWriter(logdir="./log/scalar_test") as writer:
    for step in range(1000):
        # 步骤二：向记录器添加一个tag为`train/acc`的数据
        writer.add_scalar(tag="train/acc", step=step, value=value[step])
        # 步骤二：向记录器添加一个tag为`train/loss`的数据
        writer.add_scalar(tag="train/loss", step=step, value=1/(value[step] + 1))
# 步骤一：创建第二个子文件夹scalar_test2
value = [i/500.0 for i in range(1000)]
with LogWriter(logdir="./log/scalar_test2") as writer:
    for step in range(1000):
        # 步骤二：在同样名为`train/acc`下添加scalar_test2的accuracy的数据
        writer.add_scalar(tag="train/acc", step=step, value=value[step])
        # 步骤二：在同样名为`train/loss`下添加scalar_test2的loss的数据
        writer.add_scalar(tag="train/loss", step=step, value=1/(value[step] + 1))

```

运行上述程序后，在命令行执行

```
visualdl --logdir ./log --port 8080
```

接着在浏览器打开 <http://127.0.0.1:8080>，即可查看以下折线图，对比「scalar_test」和「scalar_test2」的 Accuracy 和 Loss。

* 多组实验对比的应用案例可参考 AI Studio 项目：[VisualDL 2.0--眼疾识别训练可视化](#)

功能操作说明

- 支持数据卡片「最大化」、「还原」、「坐标系转化」(y 轴对数坐标)、「下载」折线图
- 数据点 Hover 展示详细信息
- 可搜索卡片标签，展示目标图像
- 可搜索打点数据标签，展示特定数据
- X 轴有三种衡量尺度
 1. Step: 迭代次数
 2. Walltime: 训练绝对时间
 3. Relative: 训练时长
- 可调整曲线平滑度，以便更好的展现参数整体的变化趋势

Image -- 图片可视化组件

介绍

Image 组件用于显示图片数据随训练的变化。在模型训练过程中，将图片数据传入 Image 组件，就可在 VisualDL 的前端网页查看相应图片。

记录接口

Image 组件的记录接口如下：

```
add_image(tag, img, step, walltime=None)
```

接口参数说明如下：

Demo

下面展示了使用 Image 组件记录数据的示例，代码文件请见[Image 组件](#)

```
import numpy as np
from PIL import Image
from visualdl import LogWriter


def random_crop(img):
    """获取图片的随机 100x100 分片
    """
    img = Image.open(img)
    w, h = img.size
    random_w = np.random.randint(0, w - 100)
    random_h = np.random.randint(0, h - 100)
    r = img.crop((random_w, random_h, random_w + 100, random_h + 100))
    return np.asarray(r)


if __name__ == '__main__':
    # 初始化一个记录器
    with LogWriter(logdir='./log/image_test/train') as writer:
        for step in range(6):
            # 添加一个图片数据
            writer.add_image(tag="eye",
                             img=random_crop("../docs/images/eye.jpg"),
                             step=step)
```

运行上述程序后，在命令行执行

```
visualdl --logdir ./log --port 8080
```

在浏览器输入 `http://127.0.0.1:8080`，即可查看图片数据。

功能操作说明

可搜索图片标签显示对应图片数据

支持滑动 Step/迭代次数查看不同迭代次数下的图片数据

Audio--音频播放组件

介绍

Audio 组件实时查看训练过程中的音频数据，监控语音识别与合成等任务的训练过程。

记录接口

Audio 组件的记录接口如下：

```
add_audio(tag, audio_array, step, sample_rate)
```

接口参数说明如下：

Demo

下面展示了使用 Audio 组件记录数据的示例，代码文件请见[Audio 组件](#)

```
from visualdl import LogWriter
import numpy as np
import wave

def read_audio_data(audio_path):
    """
    Get audio data.
    """
    CHUNK = 4096
    f = wave.open(audio_path, "rb")
    wavdata = []
    chunk = f.readframes(CHUNK)
```

(下页继续)

(续上页)

```
while chunk:  
    data = np.frombuffer(chunk, dtype='uint8')  
    wavdata.extend(data)  
    chunk = f.readframes(CHUNK)  
# 8k sample rate, 16bit frame, 1 channel  
shape = [8000, 2, 1]  
return shape, wavdata  
  
if __name__ == '__main__':  
    with LogWriter(logdir=".log") as writer:  
        audio_shape, audio_data = read_audio_data("./testing.wav")  
        audio_data = np.array(audio_data)  
        writer.add_audio(tag="audio_tag",  
                         audio_array=audio_data,  
                         step=0,  
                         sample_rate=8000)
```

运行上述程序后，在命令行执行

```
visualdl --logdir ./log --port 8080
```

在浏览器输入 <http://127.0.0.1:8080>，即可查看音频数据。

功能操作说明

- 可搜索音频标签显示对应音频数据
- 支持滑动 Step/迭代次数试听不同迭代次数下的音频数据
- 支持播放/暂停音频数据
- 支持音量调节
- 支持音频下载

Graph--网络结构组件

介绍

Graph 组件一键可视化模型的网络结构。用于查看模型属性、节点信息、节点输入输出等，并进行节点搜索，协助开发者们快速分析模型结构与了解数据流向。

Demo

共有两种启动方式：

- 前端模型文件拖拽上传：
 - 如只需使用 Graph 组件，则无需添加任何参数，在命令行执行 `visualdl` 后即可启动面板进行上传。
 - 如果同时需使用其他功能，在命令行指定日志文件路径（以 `./log` 为例）即可启动面板进行上传：

```
visualdl --logdir ./log --port 8080
```

- 后端启动 Graph：

- 在命令行加入参数`--model` 并指定模型文件路径（非文件夹路径），即可启动并查看网络结构可视化：

```
visualdl --model ./log/model --port 8080
```

功能操作说明

- 一键上传模型
 - 支持模型格式：PaddlePaddle、ONNX、Keras、Core ML、Caffe、Caffe2、Darknet、MXNet、ncnn、TensorFlow Lite
 - 实验性支持模型格式：TorchScript、PyTorch、Torch、ArmNN、BigDL、Chainer、CNTK、Deeplearning4j、MediaPipe、ML.NET、MNN、OpenVINO、Scikit-learn、Tengine、TensorFlow.js、TensorFlow
- 支持上下左右任意拖拽模型、放大和缩小模型
- 搜索定位到对应节点
- 点击查看模型属性
- 支持选择模型展示的信息
- 支持以 PNG、SVG 格式导出模型结构图
- 点击节点即可展示对应属性信息
- 支持一键更换模型

Histogram--直方图组件

介绍

Histogram 组件以直方图形式展示 Tensor (weight、bias、gradient 等) 数据在训练过程中的变化趋势。深入了解模型各层效果，帮助开发者精准调整模型结构。

记录接口

Histogram 组件的记录接口如下：

```
add_histogram(tag, values, step, walltime=None, buckets=10)
```

接口参数说明如下：

Demo

下面展示了使用 Histogram 组件记录数据的示例，代码文件请见Histogram 组件

```
from visualdl import LogWriter
import numpy as np

if __name__ == '__main__':
    values = np.arange(0, 1000)
    with LogWriter(logdir="./log/histogram_test/train") as writer:
        for index in range(1, 101):
            interval_start = 1 + 2 * index / 100.0
            interval_end = 6 - 2 * index / 100.0
            data = np.random.uniform(interval_start, interval_end, size=(10000))
            writer.add_histogram(tag='default tag',
                                values=data,
                                step=index,
                                buckets=10)
```

运行上述程序后，在命令行执行

```
visualdl --logdir ./log --port 8080
```

在浏览器输入 `http://127.0.0.1:8080`，即可查看训练参数直方图。

功能操作说明

- 支持数据卡片「最大化」、直方图「下载」
- 可选择 Offset 或 Overlay 模式
 - Offset 模式
 - Overlay 模式
- 数据点 Hover 展示参数值、训练步数、频次
 - 在第 240 次训练步数时，权重为-0.0031，且出现的频次是 2734 次
- 可搜索卡片标签，展示目标直方图
- 可搜索打点数据标签，展示特定数据流

PR Curve--PR 曲线组件

介绍

PR Curve 以折线图形式呈现精度与召回率的权衡分析，清晰直观了解模型训练效果，便于分析模型是否达到理想标准。

记录接口

PR Curve 组件的记录接口如下：

```
add_pr_curve(tag, labels, predictions, step=None, num_thresholds=10)
```

接口参数说明如下：

Demo

下面展示了使用 PR Curve 组件记录数据的示例，代码文件请见 *PR Curve* 组件

```
from visualdl import LogWriter
import numpy as np

with LogWriter("./log/pr_curve_test/train") as writer:
    for step in range(3):
        labels = np.random.randint(2, size=100)
        predictions = np.random.rand(100)
        writer.add_pr_curve(tag='pr_curve',
                            labels=labels,
```

(下页继续)

(续上页)

```
predictions=predictions,  
step=step,  
num_thresholds=5)
```

运行上述程序后，在命令行执行

```
visualdl --logdir ./log --port 8080
```

接着在浏览器打开 <http://127.0.0.1:8080>，即可查看 PR Curve

功能操作说明

- 支持数据卡片「最大化」、「还原」、「下载」PR 曲线
- 数据点 Hover 展示详细信息：阈值对应的 TP、TN、FP、FN
- 可搜索卡片标签，展示目标图表
- 可搜索打点数据标签，展示特定数据
- 支持查看不同训练步数下的 PR 曲线
- X 轴-时间显示类型有三种衡量尺度
 - Step：迭代次数
 - Walltime：训练绝对时间
 - Relative：训练时长

High Dimensional -- 数据降维组件

介绍

High Dimensional 组件将高维数据进行降维展示，用于深入分析高维数据间的关系。目前支持以下两种降维算法：

- PCA : Principle Component Analysis 主成分分析
- t-SNE : t-distributed stochastic neighbor embedding t-分布式随机领域嵌入

记录接口

High Dimensional 组件的记录接口如下：

```
add_embeddings(tag, labels, hot_vectors, walltime=None)
```

接口参数说明如下：

Demo

下面展示了使用 High Dimensional 组件记录数据的示例，代码文件请见[High Dimensional 组件](#)

```
from visualdl import LogWriter

if __name__ == '__main__':
    hot_vectors = [
        [1.3561076367500755, 1.3116267195134017, 1.6785401875616097],
        [1.1039614644440658, 1.8891609992484688, 1.32030488587171],
        [1.9924524852447711, 1.9358920727142739, 1.2124401279391606],
        [1.4129542689796446, 1.7372166387197474, 1.7317806077076527],
        [1.3913371800587777, 1.4684674577930312, 1.5214136352476377]]

    labels = ["label_1", "label_2", "label_3", "label_4", "label_5"]
    # 初始化一个记录器
    with LogWriter(logdir='./log/high_dimensional_test/train') as writer:
        # 将一组 labels 和对应的 hot_vectors 传入记录器进行记录
        writer.add_embeddings(tag='default',
                              labels=labels,
                              hot_vectors=hot_vectors)
```

运行上述程序后，在命令行执行

```
visualdl --logdir ./log --port 8080
```

接着在浏览器打开 <http://127.0.0.1:8080>，即可查看降维后的可视化数据。

2.2.2 自动微分机制介绍

PaddlePaddle 的神经网络核心是自动微分，本篇文章主要为你介绍如何使用飞桨的自动微分，以及飞桨的自动微分机制，帮助你更好的使用飞桨进行训练。

一、背景

神经网络是由节点和节点间的相互连接组成的。网络中每层的每个节点代表一种特定的函数，来对输入进行计算。每个函数都是由不同参数（权重 w 和偏置 b ）组成。神经网络训练的过程，就是不断让这些函数的参数进行学习、优化，以能够更好的处理后面输入的过程。

为了让神经网络的判断更加准确，首先需要有衡量效果的工具，于是损失函数应运而生。如果你想要神经网络的效果好，那么就要让损失函数尽可能的小，于是深度学习引入了能够有效计算函数最小值的算法-梯度下降等优化算法，以及参数优化更新过程-反向传播。

- 前向传播是输入通过每一层节点计算后得到每层输出，上层输出又作为下一层的输入，最终达到输出层。然后通过损失函数计算得到 loss 值。
- 反向传播是通过 loss 值来指导前向节点中的函数参数如何改变，并更新每层中每个节点的参数，来让整个神经网络达到更小的 loss 值。

自动微分机制就是让你只关注组网中的前向传播过程，然后飞桨框架来自动生成反向传播过程，从而来让你从繁琐的求导、求梯度的过程中解放出来。

二、如何使用飞桨的自动微分机制

本文通过一个比较简单的模型来还原飞桨的自动微分过程。本示例基于 Paddle2.0 编写。

```
# 加载飞桨和相关类库
import paddle
from paddle.vision.models import vgg11
import paddle.nn.functional as F
import numpy as np

print(paddle.__version__)
```

```
2.2.0
```

本案例首先定义网络。因为本示例着重展示如何使用飞桨进行自动微分，故组网部分不过多展开，直接使用高层 API 中封装好的模型 `vgg11`。

然后随机初始化一个输入 `x`，和对应标签 `label`。

```
model = vgg11()
```

(下页继续)

(续上页)

```
x = paddle.rand([1,3,224,224])
label = paddle.randint(0,1000)
```

然后将输入传入到模型中，进行前向传播过程。

```
# 前向传播
predicts = model(x)
```

前向传播结束后，你就得到模型的预测结果 `predicts`，这时可以使用飞桨中的对应损失函数 API 进行损失函数的计算。该例子中使用 `cross_entropy` 来计算损失函数，来衡量模型的预测情况。

```
# 计算损失
loss = F.cross_entropy(predicts, label)
```

随后进行反向传播，在飞桨中你只需要调用 `backward()` 即可自动化展开反向传播过程。各梯度保存在 `grad` 属性中。

```
# 开始进行反向传播
loss.backward()
```

然后来定义优化器，本例子中使用 Adam 优化器，设置 `learning_rate` 为 0.001，并把该模型的所有参数传入优化器中。

```
# 设置优化器
optim = paddle.optimizer.Adam(learning_rate=0.001, parameters=model.parameters())
```

最后通过 `step` 来开始执行优化器，并进行模型参数的更新

```
# 更新参数
optim.step()
```

通过以上步骤，你已经完成了一个神经网络前向传播、反向传播的所有过程。快自己动手试试吧！

三、飞桨中自动微分相关所有的使用方法说明

此章主要介绍飞桨中所有自动微分过程中会使用到的方法、属性等。属于第二部分的扩展阅读。

1、飞桨中的 `Tensor` 有 `stop_gradient` 属性，这个属性可以查看一个 `Tensor` 是否计算并传播梯度。

- 如果为 `True`，则该 `Tensor` 不会计算梯度，并会阻绝 Autograd 的梯度传播。
- 反之，则会计算梯度并传播梯度。用户自行创建的 `Tensor`，默认 `stop_gradient` 为 `True`，即默认不计算梯度；模型参数的 `stop_gradient` 默认都为 `False`，即默认计算梯度。

```
import paddle

a = paddle.to_tensor([1.0, 2.0, 3.0])
b = paddle.to_tensor([1.0, 2.0, 3.0], stop_gradient=False) #  
    ↪将b设置为需要计算梯度的属性
print(a.stop_gradient)
print(b.stop_gradient)
```

True

False

```
a.stop_gradient = False
print(a.stop_gradient)
```

False

2、接下来，本文用一个简单的计算图来了解如何调用 `backward()` 函数。开始从当前 `Tensor` 开始计算反向的神经网络，传导并计算计算图中 `Tensor` 的梯度。

```
import paddle

x = paddle.to_tensor([1.0, 2.0, 3.0], stop_gradient=False)
y = paddle.to_tensor([4.0, 5.0, 6.0], stop_gradient=False)
z = x ** 2 + 4 * y
```

假设上面创建的 `x` 和 `y` 分别是神经网络中的参数，`z` 为神经网络的损失值 `loss`。

$$\frac{\partial z}{\partial x} = 2x$$

$$\frac{\partial z}{\partial y} = 4$$

对 `z` 调用 `backward()`，飞桨即可以自动计算 `x` 和 `y` 的梯度，并且将他们存进 `grad` 属性中。

```
z.backward()
print("Tensor x's grad is: {}".format(x.grad))
print("Tensor y's grad is: {}".format(y.grad))
```

```
Tensor x's grad is: [2. 4. 6.]
Tensor y's grad is: [4. 4. 4.]
```

此外，飞桨默认会释放反向计算图。如果在 `backward()` 之后继续添加 OP，需要将 `backward()` 中的 `retain_graph` 参数设置为 `True`，此时之前的反向计算图会保留。

温馨小提示：将其设置为 `False` 会更加节省内存。因为他的默认值是 `False`，所以也可以直接不设置此参数。

```
import paddle

x = paddle.to_tensor([1.0, 2.0, 3.0], stop_gradient=False)
y = x + 3
y.backward(retain_graph=True) # 设置retain_graph为True，保留反向计算图
print("Tensor x's grad is: {}".format(x.grad))
```

```
Tensor x's grad is: [1. 1. 1.]
```

3、因为 `backward()` 会累积梯度，所以飞桨还提供了 `clear_grad()` 函数来清除当前 `Tensor` 的梯度。

```
import paddle
import numpy as np

x = np.ones([2, 2], np.float32)
inputs2 = []

for _ in range(10):
    tmp = paddle.to_tensor(x)
    tmp.stop_gradient = False
    inputs2.append(tmp)

ret2 = paddle.add_n(inputs2)
loss2 = paddle.sum(ret2)

loss2.backward()
print("Before clear {}".format(loss2.gradient()))

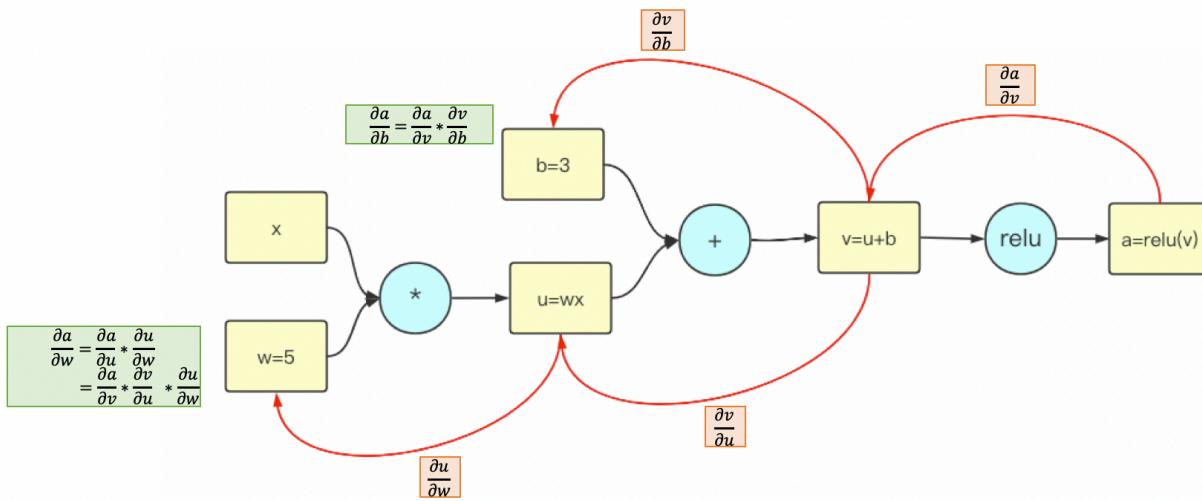
loss2.clear_grad()
print("After clear {}".format(loss2.gradient()))
```

```
Before clear [1.]
After clear [0.]
```

四、飞桨自动微分运行机制

本章主要介绍飞桨在实现反向传播进行自动微分计算时，内部是如何运行工作的。此部分为选读部分，更多是介绍飞桨内部实现机制，可以选择跳过，跳过不会影响你的正常使用。

飞桨的自动微分是通过 `trace` 的方式，记录前向 OP 的执行，并自动创建反向 var 和添加相应的反向 OP，然后来实现反向梯度计算的。



下面本文用一些的例子，来模拟这个过程。

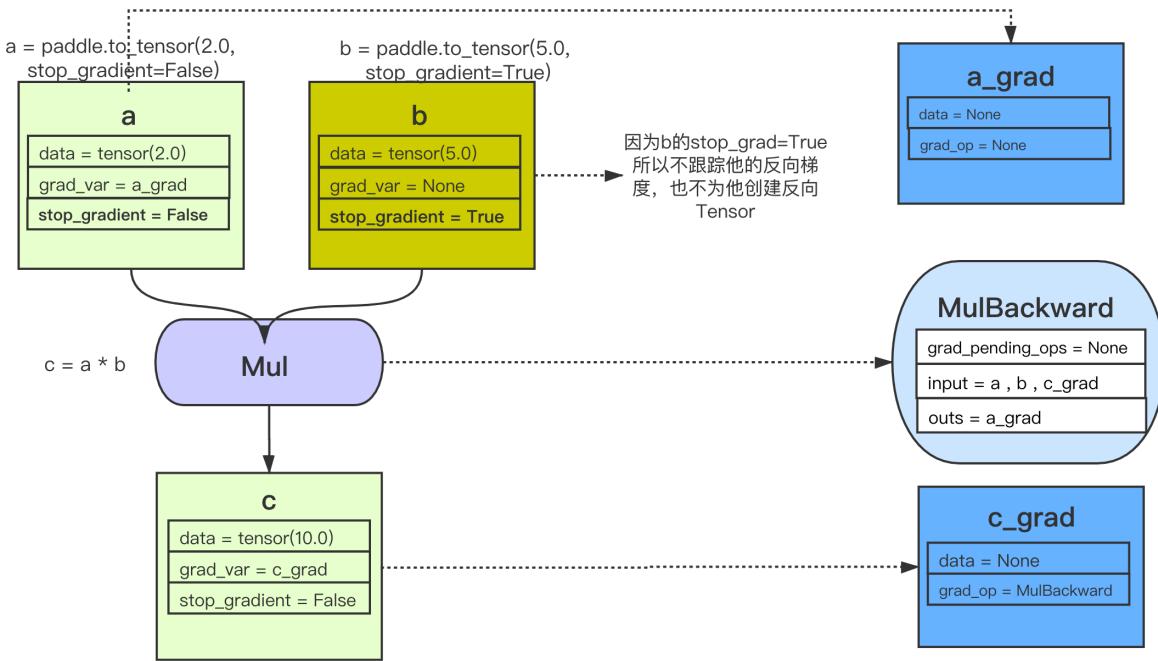
例子一：首先用一个比较简单的例子来让你了解整个过程。

```
import paddle

a = paddle.to_tensor(2.0, stop_gradient=False)
b = paddle.to_tensor(5.0, stop_gradient=True)
c = a * b
c.backward()
print("Tensor a's grad is: {}".format(a.grad))
print("Tensor b's grad is: {}".format(b.grad))
```

```
Tensor a's grad is: [5.]
Tensor b's grad is: None
```

在上面代码中 `c.backward()` 执行前，你可以理解整个计算图是这样的：



当创建 Tensor, Tensor 的 `stop_grad=False` 时, 会自动为此 Tensor 创建一个反向 Tensor。在此例子中, a 的反向 Tensor 就是 `a_grad`。在 `a_grad` 中, 会记录他的反向 OP, 因为 a 没有作为任何反向 op 的输入, 所以它的 `grad_op` 为 `None`。

当执行 OP 时, 会自动创建反向 OP, 不同的 OP 创建反向 OP 的方法不同, 传的内容也不同。本文以这个乘法 OP 为例:

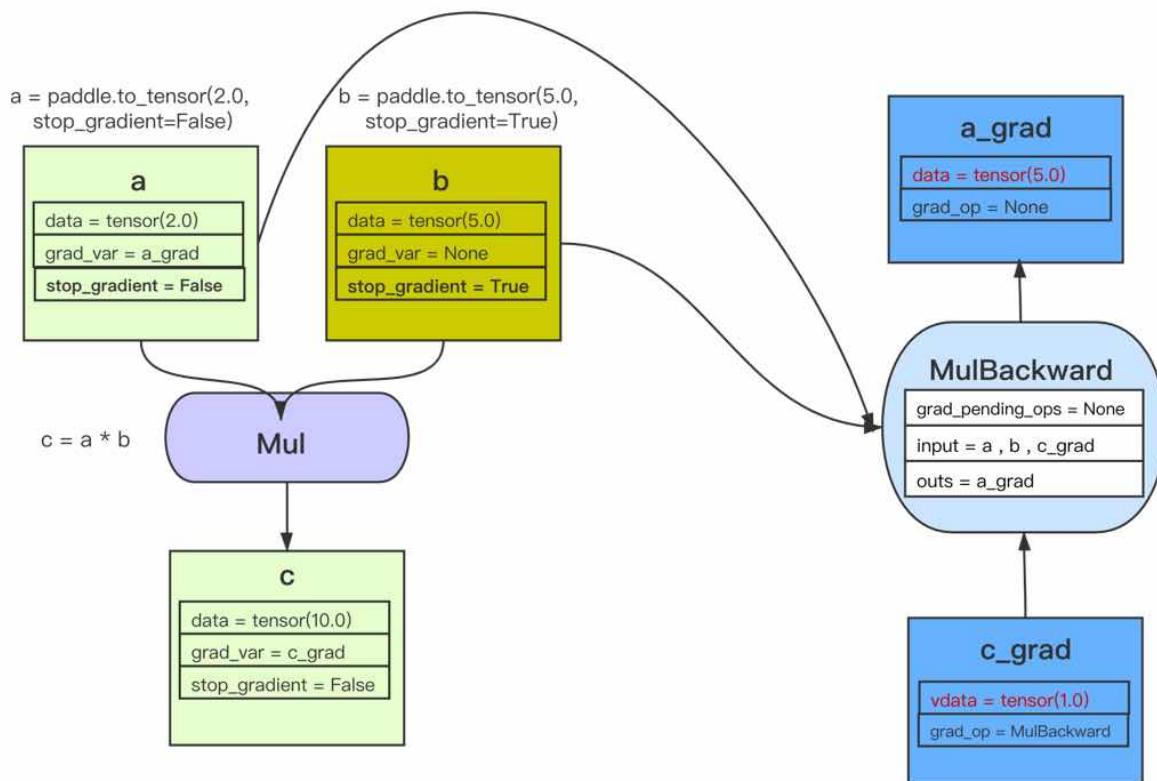
-乘法 OP 的反向 OP, 即 `MulBackward` 的输入是, 正向 OP 的两个输入, 以及正向 OP 的输出 Tensor 的反向 Tensor。在此例子中就是, `a`、`b`、`c_grad`

-乘法 OP 的反向 OP, 即 `MulBackward` 的输出是, 正向 OP 的两个输入的反向 Tensor (如果输入是 `stop_gradient=True`, 则即为 `None`)。在此例子中就是, `a_grad`、`None` (`b_grad`)

-乘法 OP 的反向 OP, 即 `MulBackward` 的 `grad_pending_ops` 是自动构建反向网络的时候, 让这个反向 op 知道它下一个可以执行的反向 op 是哪一个, 可以理解为反向网络中, 一个反向 op 指向下一个反向 op 的边。

当 `c` 通过乘法 OP 被创建后, `c` 会创建一个反向 Tensor: `c_grad`, 他的 `grad_op` 为该乘法 OP 的反向 OP, 即 `MulBackward`。

调用 `backward()` 后, 正式开始进行反向传播过程, 开始自动计算微分。



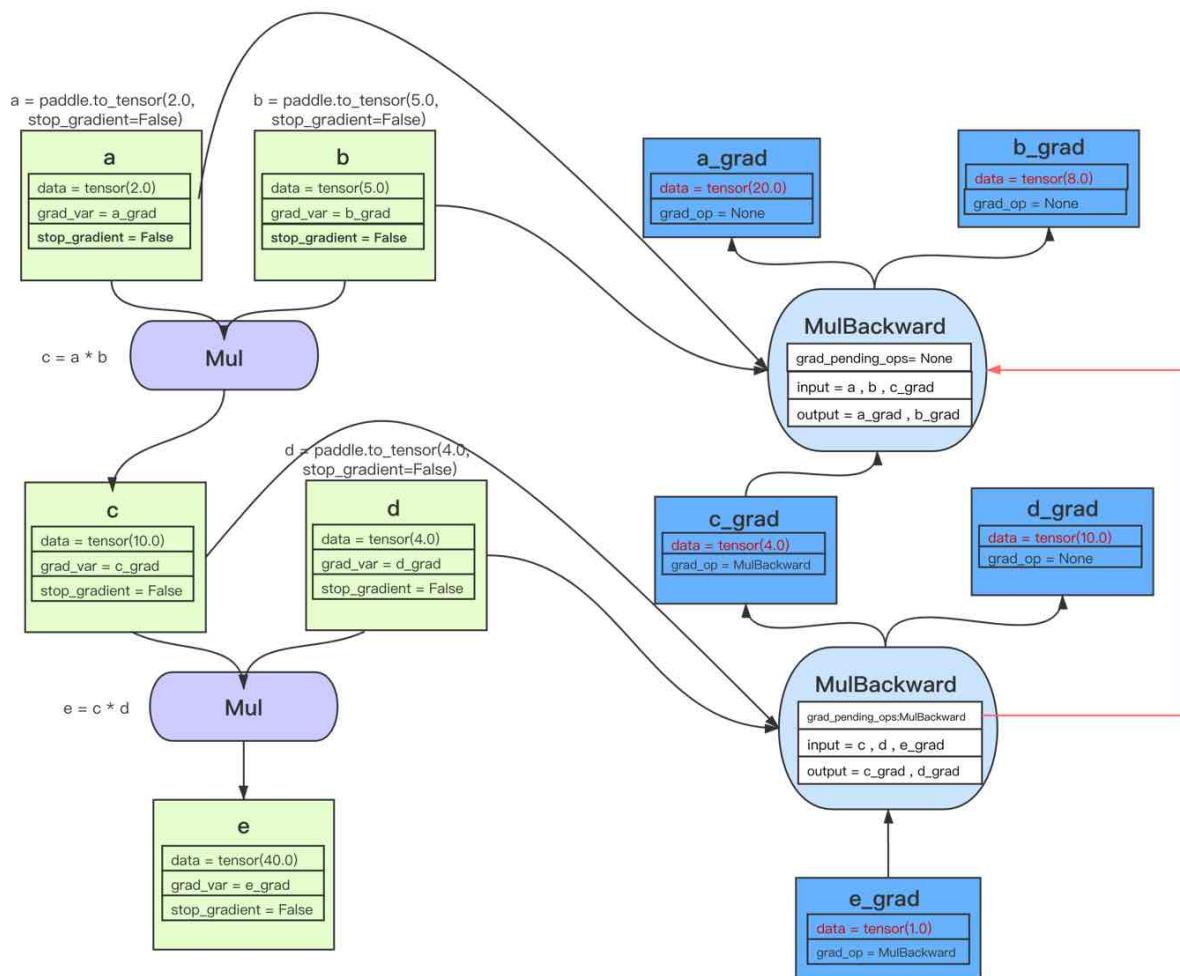
例子二：用一个稍微复杂一点的例子让你深入了解这个过程。

```
import paddle

a = paddle.to_tensor(2.0, stop_gradient=False)
b = paddle.to_tensor(5.0, stop_gradient=False)
c = a * b
d = paddle.to_tensor(4.0, stop_gradient=False)
e = c * d
e.backward()
print("Tensor a's grad is: {}".format(a.grad))
print("Tensor b's grad is: {}".format(b.grad))
print("Tensor c's grad is: {}".format(c.grad))
print("Tensor d's grad is: {}".format(d.grad))
```

```
Tensor a's grad is: [20.]
Tensor b's grad is: [8.]
Tensor c's grad is: [4.]
Tensor d's grad is: [10.]
```

该例子的正向和反向图构建过程即：



五、总结

本文章主要介绍了如何使用飞桨的自动微分，以及飞桨的自动微分机制。

2.2.3 Paddle 中的模型与层

模型是深度学习中的重要概念之一。模型的核心功能是将一组输入变量经过一系列计算，映射到另一组输出变量，该映射函数即代表一种深度学习算法。在 **Paddle** 框架中，模型包括以下两方面内容：

1. 一系列层的组合用于进行映射（前向执行）
2. 一些参数变量在训练过程中实时更新

本文档中，你将学习如何定义与使用 **Paddle** 模型，并了解模型与层的关系。

在 Paddle 中定义模型与层

在 **Paddle** 中，大多数模型由一系列层组成，层是模型的基础逻辑执行单元。层中持有两方面内容：一方面是计算所需的变量，以临时变量或参数的形式作为层的成员持有，另一方面则持有一个或多个具体的 **Operator** 来完成相应的计算。

从零开始构建变量、**Operator**，从而组建层、模型是一个很复杂的过程，并且当中难以避免的会出现很多冗余代码，因此 **Paddle** 提供了基础数据类型 `paddle.nn.Layer`，来方便你快速的实现自己的层和模型。模型和层都可以基于 `paddle.nn.Layer` 扩充实现，因此也可以说模型只是一种特殊的层。下面将演示如何利用 `paddle.nn.Layer` 建立自己的模型：

```
class Model(paddle.nn.Layer):

    def __init__(self):
        super(Model, self).__init__()
        self.flatten = paddle.nn.Flatten()

    def forward(self, inputs):
        y = self.flatten(inputs)
        return y
```

当前示例中，通过继承 `paddle.nn.Layer` 的方式构建了一个模型类型 `Model`，模型中仅包含一个 `paddle.nn.Flatten` 层。模型执行时，输入变量 `inputs` 会被 `paddle.nn.Flatten` 层展平。

子层接口

如果想要访问或修改一个模型中定义的层，则可以调用 **SubLayer** 相关的接口。

以上文创建的简单模型为例，如果想要查看模型中定义的所有子层：

```
model = Model()
print(model.sublayers())

print("-----")

for item in model.named_sublayers():
    print(item)
```

```
[Flatten()]
-----
('flatten', Flatten())
```

可以看到，通过调用 `model.sublayers()` 接口，打印出了前述模型中持有的全部子层（这时模型中只有一个 `paddle.nn.Flatten` 子层）。

而遍历 `model.named_sublayers()` 时，每一轮循环会拿到一组（子层名称（`'flatten'`），子层对象（`paddle.nn.Flatten`））的元组。

接下来如果想要进一步添加一个子层，则可以调用 `add_sublayer()` 接口：

```
fc = paddle.nn.Linear(10, 3)
model.add_sublayer("fc", fc)
print(model.sublayers())
```

```
[Flatten(), Linear(in_features=10, out_features=3, dtype=float32)]
```

可以看到 `model.add_sublayer()` 向模型中添加了一个 `paddle.nn.Linear` 子层，这样模型中总共有 `paddle.nn.Flatten` 和 `paddle.nn.Linear` 两个子层了。

通过上述方法可以往模型中添加成千上万个子层，当模型中子层数量较多时，如何高效地对所有子层进行统一修改呢？**Paddle** 提供了 `apply()` 接口。通过这个接口，可以自定义一个函数，然后将该函数批量作用在所有子层上：

```
def function(layer):
    print(layer)

model.apply(function)
```

```
Flatten()
Linear(in_features=10, out_features=3, dtype=float32)
```

(下页继续)

(续上页)

```
Model(
    (flatten): Flatten()
    (fc): Linear(in_features=10, out_features=3, dtype=float32)
)
```

当前例子中，定义了一个以 **layer** 作为参数的函数 **function**，用来打印传入的 **layer** 信息。通过调用 `model.apply()` 接口，将 **function** 作用在模型的所有子层中，也因此输出信息中打印了 **model** 中所有子层的信息。

另外一个批量访问子层的接口是 `children()` 或者 `named_children()`。这两个接口通过 **Iterator** 的方式访问每个子层：

```
sublayer_iter = model.children()
for sublayer in sublayer_iter:
    print(sublayer)
```

```
Flatten()
Linear(in_features=10, out_features=3, dtype=float32)
```

可以看到，遍历 `model.children()` 时，每一轮循环都可以按照子层注册顺序拿到对应 `paddle.nn.Layer` 的对象

层的变量成员

参数变量的添加与修改

有的时候希望向网络中添加一个参数作为输入。比如在使用图像风格转换模型时，会使用参数作为输入图像，在训练过程中不断更新该图像参数，最终拿到风格转换后的图像。

这时可以通过 `create_parameter()` 与 `add_parameter()` 组合，来创建并记录参数：

```
class Model(paddle.nn.Layer):

    def __init__(self):
        super(Model, self).__init__()
        img = self.create_parameter([1, 3, 256, 256])
        self.add_parameter("img", img)
        self.flatten = paddle.nn.Flatten()

    def forward(self):
        y = self.flatten(self.img)
        return y
```

上述例子创建并向模型中添加了一个名字为”img”的参数。随后可以直接通过调用 `model.img` 来访问该参数。对于已经添加的参数，可以通过 `parameters()` 或者 `named_parameters()` 来访问

```
model = Model()
model.parameters()
print("-----")
for item in model.named_parameters():
    print(item)
```

```
[Parameter containing:
Tensor(shape=[1, 3, 256, 256], dtype=float32, place=CPUPlace, stop_gradient=False,
...
-----
('img', Parameter containing:
Tensor(shape=[1, 3, 256, 256], dtype=float32, place=CPUPlace, stop_gradient=False,
...)
```

可以看到，`model.parameters()` 将模型中所有参数以数组的方式返回。

在实际的模型训练过程中，当调用反向图执行方法后，**Paddle** 会计算出模型中每个参数的梯度并将其保存在相应的参数对象中。如果已经对该参数进行了梯度更新，或者出于一些原因不希望该梯度累加到下一轮训练，则可以调用 `clear_gradients()` 来清除这些梯度值。

```
model = Model()
out = model()
out.backward()
model.clear_gradients()
```

非参数变量的添加

参数变量往往需要参与梯度更新，但很多情况下只是需要一个临时变量甚至一个常量。比如在模型执行过程中想将一个中间变量保存下来，这时需要调用 `create_tensor()` 接口：

```
class Model(paddle.nn.Layer):

    def __init__(self):
        super(Model, self).__init__()
        self.saved_tensor = self.create_tensor(name="saved_tensor0")
        self.flatten = paddle.nn.Flatten()
        self.fc = paddle.nn.Linear(10, 100)

    def forward(self, input):
```

(下页继续)

(续上页)

```

y = self.flatten(input)
# Save intermediate tensor
paddle.assign(y, self.saved_tensor)
y = self.fc(y)
return y

```

这里调用 `self.create_tensor()` 创造了一个临时变量并将其记录在模型的 `self.saved_tensor` 中。在模型执行时调用 `paddle.assign` 用该临时变量记录变量 `y` 的数值。

Buffer 变量的添加

Buffer 的概念仅仅影响动态图向静态图的转换过程。在上一节中创建了一个临时变量用来临时存储中间变量的值。但这个临时变量在动态图向静态图转换的过程中并不会被记录在静态的计算图当中。如果希望该变量成为静态图的一部分，就需要进一步调用 `register_buffers()` 接口：

```

class Model(paddle.nn.Layer):

    def __init__(self):
        super(Model, self).__init__()
        saved_tensor = self.create_tensor(name="saved_tensor0")
        self.register_buffer("saved_tensor", saved_tensor, persistable=True)
        self.flatten = paddle.nn.Flatten()
        self.fc = paddle.nn.Linear(10, 100)

    def forward(self, input):
        y = self.flatten(input)
        # Save intermediate tensor
        paddle.assign(y, self.saved_tensor)
        y = self.fc(y)
        return y

```

这样在动态图转静态图时 `saved_tensor` 就会被记录到静态图中。

对于模型中已经注册的 **Buffer**，可以通过 `buffers()` 或者 `named_buffers()` 进行访问：

```

model = Model()
print(model.buffers())
for item in model.named_buffers():
    print(item)

```

```

[Tensor(Not initialized)]
('saved_tensor', Tensor(Not initialized))

```

可以看到 `model.buffers()` 以数组形式返回了模型中注册的所有 **Buffer**

层的执行

经过一系列对模型的配置，假如已经准备好了个 **Paddle** 模型如下：

```
class Model(paddle.nn.Layer):

    def __init__(self):
        super(Model, self).__init__()
        self.flatten = paddle.nn.Flatten()

    def forward(self, inputs):
        y = self.flatten(inputs)
        return y
```

想要执行该模型，首先需要对执行模式进行设置

执行模式设置

模型的执行模式有两种，如果需要训练的话调用 `train()`，如果只进行前向执行则调用 `eval()`

```
x = paddle.randn([10, 1], 'float32')
model = Model()
model.eval() # set model to eval mode
out = model(x)
model.train() # set model to train mode
out = model(x)
```

```
Tensor(shape=[10, 1], dtype=float32, place=CPUPlace, stop_gradient=True,
      ...)
```

这里将模型的执行模式先后设置为 `eval` 和 `train`。两种执行模式是互斥的，新的执行模式设置会覆盖原有的设置。

执行函数

模式设置完成后可以直接调用执行函数。可以直接调用 `forward()` 方法进行前向执行，也可以调用 `__call__()`，从而执行在 `forward()` 当中定义的前向计算逻辑。

```
model = Model()
x = paddle.randn([10, 1], 'float32')
out = model(x)
print(out)
```

```
Tensor(shape=[10, 1], dtype=float32, place=CPUPlace, stop_gradient=True,
      ...)
```

这里直接调用 `__call__()` 方法调用模型的前向执行逻辑。

添加额外的执行逻辑

有时希望某些变量在进入层前首先进行一些预处理，这个功能可以通过注册 **hook** 来实现。**hook** 是一个作用于变量的自定义函数，在模型执行时调用。对于注册在层上的 **hook** 函数，可以分为 **pre_hook** 和 **post_hook** 两种。**pre_hook** 可以对层的输入变量进行处理，用函数的返回值作为新的变量参与层的计算。**post_hook** 则可以对层的输出变量进行处理，将层的输出进行进一步处理后，用函数的返回值作为层计算的输出。

通过 `register_forward_post_hook()` 接口，我们可以注册一个 **post_hook**：

```
def forward_post_hook(layer, input, output):
    return 2*output

x = paddle.ones([10, 1], 'float32')
model = Model()
forward_post_hook_handle = model.flatten.register_forward_post_hook(forward_post_hook)
out = model(x)
print(out)
```

```
Tensor(shape=[10, 1], dtype=float32, place=CPUPlace, stop_gradient=True,
      [[2.],
       [2.],
       ...])
```

同样的也可以使用 `register_forward_pre_hook()` 来注册 **pre_hook**：

```
def forward_pre_hook(layer, input):
    print(input)
    return input

x = paddle.ones([10, 1], 'float32')
model = Model()
forward_pre_hook_handle = model.flatten.register_forward_pre_hook(forward_pre_hook)
out = model(x)
```

```
(Tensor(shape=[10, 1], dtype=float32, place=CUDAPlace(0), stop_gradient=True,
      [[1.],
       [1.],
       [1.],
```

(下页继续)

(续上页)

```
[1.],  
[1.],  
[1.],  
[1.],  
[1.],  
[1.],  
[1.]]),)
```

模型数据保存

如果想要保存模型中参数而不存储模型本身，则可以首先调用 `state_dict()` 接口将模型中的参数以及永久变量存储到一个 **Python** 字典中，随后保存该字典。

```
model = Model()  
state_dict = model.state_dict()  
paddle.save(state_dict, "paddle_dy.pdparams")
```

可以随时恢复：

```
model = Model()  
state_dict = paddle.load("paddle_dy.pdparams")  
model.set_state_dict(state_dict)
```

如果想要连同模型一起保存，则可以参考[paddle.jit.save\(\)](#)

2.2.4 梯度裁剪方式介绍

神经网络是通过梯度下降来进行网络学习，随着网络层数的增加，“梯度爆炸”的问题可能会越来越明显。例如：在梯度反向传播中，如果每一层的输出相对输入的偏导 > 1 ，随着网络层数的增加，梯度会越来越大，则有可能发生“梯度爆炸”。

如果发生了“梯度爆炸”，在网络学习过程中会直接跳过最优解，所以有必要进行梯度裁剪，防止网络在学习过程中越过最优解。

Paddle 提供了三种梯度裁剪方式：

一、设定范围值裁剪

设定范围值裁剪：将参数的梯度限定在一个范围内，如果超出这个范围，则进行裁剪。

使用方式：需要创建一个 paddle.nn.ClipGradByValue 类的实例，然后传入到优化器中，优化器会在更新参数前，对梯度进行裁剪。

1. 全部参数裁剪（默认）

默认情况下，会裁剪优化器中全部参数的梯度：

```
import paddle

linear = paddle.nn.Linear(10, 10)
clip = paddle.nn.ClipGradByValue(min=-1, max=1)
sgd = paddle.optimizer.SGD(learning_rate=0.1, parameters=linear.parameters(), grad_
→clip=clip)
```

如果仅需裁剪部分参数，用法如下：

2. 部分参数裁剪

部分参数裁剪需要设置参数的 paddle.ParamAttr，其中的 need_clip 默认为 True，表示需要裁剪，如果设置为 False，则不会裁剪。

例如：仅裁剪 linear 中 weight 的梯度，则需要在创建 linear 层时设置 bias_attr 如下：

```
linear = paddle.nn.Linear(10, 10, bias_attr=paddle.ParamAttr(need_clip=False))
```

二、通过 L2 范数裁剪

通过 L2 范数裁剪：梯度作为一个多维 Tensor，计算其 L2 范数，如果超过最大值则按比例进行裁剪，否则不裁剪。

使用方式：需要创建一个 paddle.nn.ClipGradByNorm 类的实例，然后传入到优化器中，优化器会在更新参数前，对梯度进行裁剪。

裁剪公式如下：

$$Out = \begin{cases} X & if(norm(X) \leq clip_norm) \\ \frac{clip_norm * X}{norm(X)} & if(norm(X) > clip_norm) \end{cases}$$

其中 $norm(X)$ 代表 X 的 L2 范数

$$norm(X) = (\sum_{i=1}^n |x_i|^2)^{\frac{1}{2}}$$

1. 全部参数裁剪（默认）

默认情况下，会裁剪优化器中全部参数的梯度：

```
linear = paddle.nn.Linear(10, 10)
clip = paddle.nn.ClipGradByNorm(clip_norm=1.0)
sgd = paddle.optimizer.SGD(learning_rate=0.1, parameters=linear.parameters(), grad_
    ↪clip=clip)
```

如果仅需裁剪部分参数，用法如下：

2. 部分参数裁剪

部分参数裁剪的设置方式与上面一致，也是通过设置参数的 `paddle.ParamAttr`，其中的 `need_clip` 默认为 `True`，表示需要裁剪，如果设置为 `False`，则不会裁剪。

例如：仅裁剪 `linear` 中 `bias` 的梯度，则需要在创建 `linear` 层时设置 `weight_attr` 如下：

```
linear = paddle.nn.Linear(10, 10, weight_attr=paddle.ParamAttr(need_clip=False))
```

三、通过全局 L2 范数裁剪

Title underline too short.

三、通过全局 L2 范数裁剪

将优化器中全部参数的梯度组成向量，对该向量求解 L2 范数，如果超过最大值则按比例进行裁剪，否则不裁剪。

使用方式：需要创建一个 `paddle.nn.ClipGradByGlobalNorm` 类的实例，然后传入到优化器中，优化器会在更新参数前，对梯度进行裁剪。

裁剪公式如下：

$$Out[i] = \begin{cases} X[i] & if(global_norm \leq clip_norm) \\ \frac{clip_norm * X[i]}{global_norm} & if(global_norm > clip_norm) \end{cases}$$

其中：

$$global_norm = \sqrt{\sum_{i=0}^{n-1} (norm(X[i]))^2}$$

其中 `norm(X)` 代表 `X` 的 L2 范数

1. 全部参数裁剪（默认）

默认情况下，会裁剪优化器中全部参数的梯度：

```
linear = paddle.nn.Linear(10, 10)
clip = paddle.nn.ClipGradByGlobalNorm(clip_norm=1.0)
sgd = paddle.optimizer.SGD(learning_rate=0.1, parameters=linear.parameters(), grad_
↪clip=clip)
```

如果仅需裁剪部分参数，用法如下：

2. 部分参数裁剪

部分参数裁剪的设置方式与上面一致，也是通过设置参数的 `paddle.ParamAttr`，其中的 `need_clip` 默认为 `True`，表示需要裁剪，如果设置为 `False`，则不会裁剪。可参考上面的示例代码进行设置。

2.2.5 模型导出 ONNX 协议

一、简介

ONNX (Open Neural Network Exchange) 是针对机器学习所设计的开源文件格式，用于存储训练好的模型。它使得不同的人工智能框架可以采用相同格式存储模型并交互。通过 ONNX 格式，Paddle 模型可以使用 OpenVINO、ONNX Runtime 等框架进行推理。

Paddle 转 ONNX 协议由 `paddle2onnx` 实现，下面介绍如何将 Paddle 模型转换为 ONNX 模型并验证正确性。

本教程涉及的示例代码，可点击 IPython 获取，除 Paddle 以外，还需安装以下依赖：

```
pip install paddle2onnx onnx onnxruntime // -i https://mirror.baidu.com/pypi/simple
↪如果网速不好，可以使用其他源下载
```

二、模型导出为 ONNX 协议

Title underline too short.

二、模型导出为 ONNX 协议

```
#####
#
```

2.1 动态图导出 ONNX 协议

Title underline too short.

2.1 动态图导出 ONNX 协议

```
-----
```

Paddle 动态图模型转换为 ONNX 协议，首先会将 Paddle 的动态图 `paddle.nn.Layer` 转换为静态图，详细原理可以参考 [动态图转静态图](#)。然后依照 ONNX 的算子协议，将 Paddle 的算子一一映射为 ONNX 的算子。

动态图转换 ONNX 调用 paddle.onnx.export() 接口即可实现，该接口通过 input_spec 参数为模型指定输入的形状和数据类型，支持 Tensor 或 InputSpec，其中 InputSpec 支持动态的 shape。

关于 paddle.onnx.export 接口更详细的使用方法，请参考 [API](#)。

```
import paddle
from paddle import nn
from paddle.static import InputSpec

class LinearNet(nn.Layer):
    def __init__(self):
        super(LinearNet, self).__init__()
        self._linear = nn.Linear(784, 10)

    def forward(self, x):
        return self._linear(x)

# export to ONNX
layer = LinearNet()
save_path = 'onnx.save/linear_net'
x_spec = InputSpec([None, 784], 'float32', 'x')
paddle.onnx.export(layer, save_path, input_spec=[x_spec])
```

2.2 静态图导出 ONNX 协议

Title underline too short.

2.2 静态图 导出ONNX协议

Paddle 2.0 以后将主推动态图组网方式，如果您的模型来自于旧版本的 Paddle，使用静态图组网，请参考 paddle2onnx 的 [使用文档](#) 和 [示例](#)。

三、ONNX 模型的验证

ONNX 官方工具包提供了 API 可验证模型的正确性，主要包括两个方面，一是算子是否符合对应版本的协议，二是网络结构是否完整。

```
# check by ONNX
import onnx

onnx_file = save_path + '.onnx'
onnx_model = onnx.load(onnx_file)
```

(下页继续)

(续上页)

```
onnx.checker.check_model(onnx_model)
print('The model is checked!')
```

Duplicate explicit target name: "paddle2onnx".

如果模型检查失败, 请到 [Paddle](#) 或 [paddle2onnx](#) 提出 Issue, 我们会跟进相应的问题。

四、ONNXRuntime 推理

Title underline too short.

```
四、ONNXRuntime 推理
#####
#
```

本节介绍使用 ONNXRuntime 对已转换的 Paddle 模型进行推理, 并与使用 Paddle 进行推理的结果进行对比。

```
import numpy as np
import onnxruntime

x = np.random.random((2, 784)).astype('float32')

# predict by ONNX Runtime
ort_sess = onnxruntime.InferenceSession(onnx_file)
ort_inputs = {ort_sess.get_inputs()[0].name: x}
ort_outs = ort_sess.run(None, ort_inputs)

print("Exported model has been predicted by ONNXRuntime!")

# predict by Paddle
layer.eval()
tensor_x = paddle.to_tensor(x)
paddle_outs = layer(tensor_x)

# compare ONNX Runtime and Paddle results
np.testing.assert_allclose(ort_outs[0], paddle_outs.numpy(), rtol=1.0, atol=1e-05)

print("The difference of results between ONNXRuntime and Paddle looks good!")
```

五、相关链接

- 算子转换支持列表
- 模型转换支持列表

2.3 动态图转静态图

2.3.1 什么是动态图和静态图？

在深度学习模型构建上，飞桨框架支持动态图编程和静态图编程两种方式，其代码编写和执行方式均存在差异。

- **动态图编程：**采用 Python 的编程风格，解析式地执行每一行网络代码，并同时返回计算结果。在 [模型开发](#) 章节中，介绍的都是动态图编程方式。
- **静态图编程：**采用先编译后执行的方式。需先在代码中预定义完整的神经网络结构，飞桨框架会将神经网络描述为 *Program* 的数据结构，并对 *Program* 进行编译优化，再调用执行器获得计算结果。

动态图编程体验更佳、更易调试，但是因为采用 Python 实时执行的方式，开销较大，在性能方面与 C++ 有一定差距；静态图调试难度大，但是将前端 Python 编写的神经网络预定义为 *Program* 描述，转到 C++ 端重新解析执行，脱离了 Python 依赖，往往执行性能更佳，并且预先拥有完整网络结构也更利于全局优化。

2.3.2 什么场景下需要动态图转静态图？

Title overline too short.

```
=====
什么场景下需要动态图转静态图？
=====
```

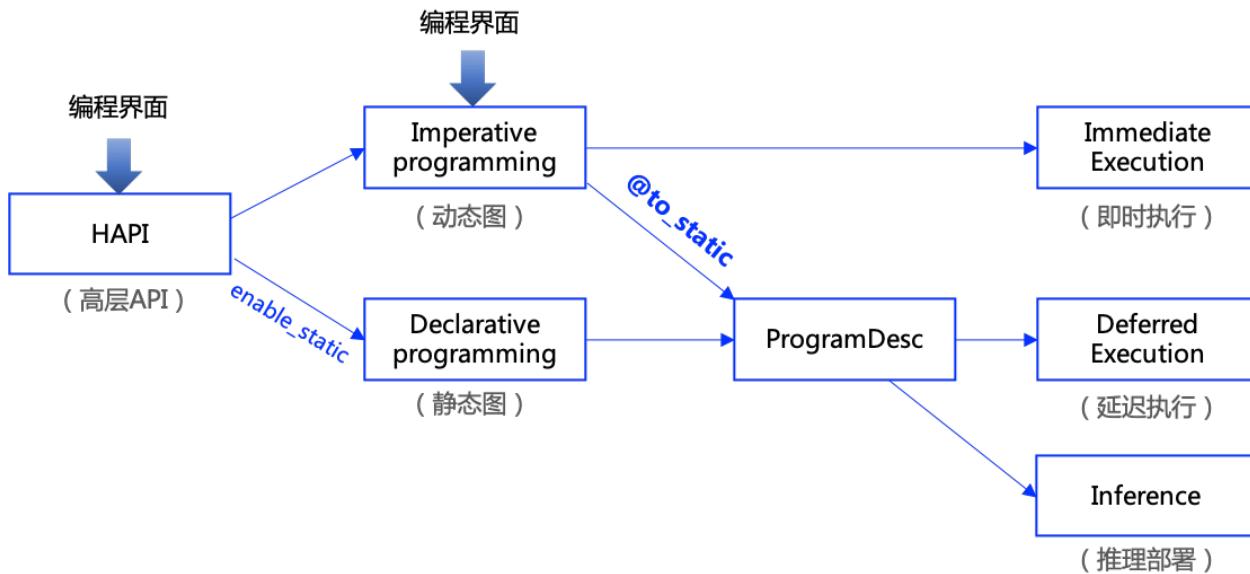
飞桨框架在设计时，考虑同时兼顾动态图的高易用性和静态图的高性能优势，采用『动静统一』的方案：

- 在模型开发时，推荐采用动态图编程。可获得更好的编程体验、更易用的接口、更友好的调试交互机制。
- 在模型训练或者推理部署时，只需添加一行装饰器 `@to_static`，即可将动态图代码转写为静态图代码，并在底层自动使用静态图执行器运行。可获得更好的模型运行性能。

方案如下图所示：

图 1 飞桨框架动静统一方案示意图

备注：飞桨框架 2.0 及以上版本默认的编程模式是动态图模式，包括使用高层 API 编程和基础的 API 编程。如果想切换到静态图模式编程，可以在程序的开始执行 `enable_static()` 函数。如果程序已经使用动态图的模式



编写了，想转成静态图模式训练或者保存模型用于部署，可以使用装饰器 `@to_static`。

想了解动态图和静态图的详细对比介绍，可参见 [动态图和静态图的差异](#)。

以下将详细介绍动静转换的各个模块内容：

- [使用样例](#)：介绍了动静转换 `@to_static` 的基本用法
- [转换原理](#)：介绍了动静转换的内部原理
- [支持语法](#)：介绍了动静转换功能已支持的语法概况
- [案例解析](#)：介绍使用 `@to_static` 时常见的问题和案例解析
- [报错调试](#)：介绍了动静转换 `@to_static` 的调试方法和经验

使用样例

一、使用 `@to_static` 进行动静转换

动静转换 (`@to_static`) 通过解析 Python 代码（抽象语法树，下简称：AST）实现一行代码即可将动态图转为静态图的功能，只需在待转化的函数前添加一个装饰器 `@paddle.jit.to_static`。

使用 `@to_static` 即支持 可训练可部署，也支持只部署（详见模型导出），常见使用方式如下：

- 方式一：使用 `@to_static` 装饰器装饰 SimpleNet (继承了 `nn.Layer`) 的 `forward` 函数：

```
import paddle
from paddle.jit import to_static
```

(下页继续)

(续上页)

```

class SimpleNet(paddle.nn.Layer):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.linear = paddle.nn.Linear(10, 3)

    @to_static # 动静转换
    def forward(self, x, y):
        out = self.linear(x)
        out = out + y
        return out

net = SimpleNet()
net.eval()
x = paddle.rand([2, 10])
y = paddle.rand([2, 3])
out = net(x, y)           # 动转静训练
paddle.jit.save(net, './net') # 导出预测模型

```

- 方式二：调用 paddle.jit.to_static() 函数，仅做预测模型导出时推荐此种用法。

```

import paddle
from paddle.jit import to_static

class SimpleNet(paddle.nn.Layer):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.linear = paddle.nn.Linear(10, 3)

    def forward(self, x, y):
        out = self.linear(x)
        out = out + y
        return out

net = SimpleNet()
net.eval()
net = paddle.jit.to_static(net) # 动静转换
x = paddle.rand([2, 10])
y = paddle.rand([2, 3])
out = net(x, y)           # 动转静训练
paddle.jit.save(net, './net') # 导出预测模型

```

方式一和方式二的主要区别是，前者直接在 forward() 函数定义处装饰，后者显式调用了 jit.to_static() 方法，默认会对 net.forward 进行动静转换。

两种方式均支持动转静训练，如下是 @to_static 的基本执行流程：

二、动静模型导出

动静模块是架在动态图与静态图的一个桥梁，旨在打破动态图模型训练与静态部署的鸿沟，消除部署时对模型代码的依赖，打通与预测端的交互逻辑。下图展示了动态图模型训练——> 动转静模型导出——> 静态预测部署的流程。

在处理逻辑上，动静主要包含两个主要模块：

- **代码层面**: 将模型中所有的 `layers` 接口在静态图模式下执行以转为 `Op`，从而生成完整的静态 `Program`
- **Tensor 层面**: 将所有的 `Parameters` 和 `Buffers` 转为可导出的 `Variable` 参数 (`persistable=True`)

2.1 通过 `forward` 导出预测模型

通过 `forward` 导出预测模型导出一般包括三个步骤：

- 切换 `eval()` 模式：类似 `Dropout`、`LayerNorm` 等接口在 `train()` 和 `eval()` 的行为存在较大的差异，在模型导出前，请务必确认模型已切换到正确的模式，否则导出的模型在预测阶段可能出现输出结果不符合预期的情况。
- 构造 `InputSpec` 信息：`InputSpec` 用于表示输入的 `shape`、`dtype`、`name` 信息，且支持用 `None` 表示动态 `shape`（如输入的 `batch_size` 维度），是辅助动静转换的必要描述信息。
- 调用 `save` 接口：调用 `paddle.jit.save` 接口，若传入的参数是类实例，则默认对 `forward` 函数进行 `@to_static` 装饰，并导出其对应的模型文件和参数文件。

如下是一个简单的示例：

```
import paddle
from paddle.jit import to_static
from paddle.static import InputSpec

class SimpleNet(paddle.nn.Layer):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.linear = paddle.nn.Linear(10, 3)

    def forward(self, x, y):
        out = self.linear(x)
        out = out + y
        return out

    def another_func(self, x):
```

(下页继续)

(续上页)

```

        out = self.linear(x)
        out = out * 2
    return out

net = SimpleNet()
# train(net) 模型训练 (略)

# step 1: 切换到 eval() 模式
net.eval()

# step 2: 定义 InputSpec 信息
x_spec = InputSpec(shape=[None, 3], dtype='float32', name='x')
y_spec = InputSpec(shape=[3], dtype='float32', name='y')

# step 3: 调用 jit.save 接口
net = paddle.jit.save(net, path='simple_net', input_spec=[x_spec, y_spec]) # 动静转换

```

执行上述代码样例后，在当前目录下会生成三个文件，即代表成功导出预测模型：

simple_net.pdiparams	// 存放模型中所有的权重数据
simple_net.pdmodel	// 存放模型的网络结构
simple_net.pdiparams.info	// 存放额外的其他信息

2.2 使用 InputSpec 指定模型输入 Tensor 信息

动静转换在生成静态图 Program 时，依赖输入 Tensor 的 shape、dtype 和 name 信息。因此，Paddle 提供了 InputSpec 接口，用于指定输入 Tensor 的描述信息，并支持动态 shape 特性。

2.2.1 构造 InputSpec

方式一：直接构造

InputSpec 接口在 paddle.static 目录下，只有 shape 是必须参数，dtype 和 name 可以缺省，默认取值分别为 float32 和 None。使用样例如下：

```

from paddle.static import InputSpec

x = InputSpec([None, 784], 'float32', 'x')
label = InputSpec([None, 1], 'int64', 'label')

print(x)      # InputSpec(shape=(-1, 784), dtype=VarType.FP32, name=x)
print(label)  # InputSpec(shape=(-1, 1), dtype=VarType.INT64, name=label)

```

方式二：由 Tensor 构造

可以借助 `InputSpec.from_tensor` 方法，从一个 `Tensor` 直接创建 `InputSpec` 对象，其拥有与源 `Tensor` 相同的 `shape` 和 `dtype`。使用样例如下：

```
import numpy as np
import paddle
from paddle.static import InputSpec

x = paddle.to_tensor(np.ones([2, 2], np.float32))
x_spec = InputSpec.from_tensor(x, name='x')
print(x_spec) # InputSpec(shape=(2, 2), dtype=VarType.FP32, name=x)
```

注：若未在 `from_tensor` 中指定新的 `name`，则默认使用与源 `Tensor` 相同的 `name`。

方式三：由 numpy.ndarray

也可以借助 `InputSpec.from_numpy` 方法，从一个 `Numpy.ndarray` 直接创建 `InputSpec` 对象，其拥有与源 `ndarray` 相同的 `shape` 和 `dtype`。使用样例如下：

```
import numpy as np
from paddle.static import InputSpec

x = np.ones([2, 2], np.float32)
x_spec = InputSpec.from_numpy(x, name='x')
print(x_spec) # InputSpec(shape=(2, 2), dtype=VarType.FP32, name=x)
```

注：若未在 `from_numpy` 中指定新的 `name`，则默认使用 `None`。

2.2.2 基本用法

方式一：在 `@to_static` 装饰器中调用

如下是一个简单的使用样例：

```
import paddle
from paddle.nn import Layer
from paddle.jit import to_static
from paddle.static import InputSpec

class SimpleNet(Layer):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.linear = paddle.nn.Linear(10, 3)

    @to_static(input_spec=[InputSpec(shape=[None, 10], name='x'), InputSpec(shape=[3], name='y')])
```

(下页继续)

(续上页)

```

def forward(self, x, y):
    out = self.linear(x)
    out = out + y
    return out

net = SimpleNet()

# save static model for inference directly
paddle.jit.save(net, './simple_net')

```

在上述的样例中，`@to_static` 装饰器中的 `input_spec` 为一个 `InputSpec` 对象组成的列表，用于依次指定参数 `x` 和 `y` 对应的 `Tensor` 签名信息。在实例化 `SimpleNet` 后，可以直接调用 `paddle.jit.save` 保存静态图模型，不需要执行任何其他的代码。

注：

1. `input_spec` 参数中不仅支持 `InputSpec` 对象，也支持 `int`、`float` 等常见 Python 原生类型。
2. 若指定 `input_spec` 参数，则需为被装饰函数的所有必选参数都添加对应的 `InputSpec` 对象，如上述样例中，不支持仅指定 `x` 的签名信息。
3. 若被装饰函数中包括非 `Tensor` 参数，推荐函数的非 `Tensor` 参数设置默认值，如
`forward(self, x, use_bn=False)`

方式二：在 `to_static` 函数中调用

若期望在动态图下训练模型，在训练完成后保存预测模型，并指定预测时需要的签名信息，则可以选择在保存模型时，直接调用 `to_static` 函数。使用样例如下：

```

class SimpleNet(Layer):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.linear = paddle.nn.Linear(10, 3)

    def forward(self, x, y):
        out = self.linear(x)
        out = out + y
        return out

net = SimpleNet()

# train process (Pseudo code)
for epoch_id in range(10):
    train_step(net, train_reader)

net = to_static(net, input_spec=[InputSpec(shape=[None, 10], name='x'),  

                                InputSpec(shape=[3], name='y')])

```

(下页继续)

(续上页)

```
# save static model for inference directly
paddle.jit.save(net, './simple_net')
```

如上述样例代码中，在完成训练后，可以借助 `to_static(net, input_spec=...)` 形式对模型实例进行处理。Paddle 会根据 `input_spec` 信息对 `forward` 函数进行递归的动转静，得到完整的静态图，且包括当前训练好的参数数据。

方式三：通过 list 和 dict 推导

上述两个样例中，被装饰的 `forward` 函数的参数均为 `Tensor`。这种情况下，参数个数必须与 `InputSpec` 个数相同。但当被装饰的函数参数为 `list` 或 `dict` 类型时，`input_spec` 需要与函数参数保持相同的嵌套结构。

当函数的参数为 `list` 类型时，`input_spec` 列表中对应元素的位置，也必须是包含相同元素的 `InputSpec` 列表。使用样例如下：

```
class SimpleNet(Layer):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.linear = paddle.nn.Linear(10, 3)

    @to_static(input_spec=[[InputSpec(shape=[None, 10], name='x'), ↵
                           InputSpec(shape=[3], name='y')]])
    def forward(self, inputs):
        x, y = inputs[0], inputs[1]
        out = self.linear(x)
        out = out + y
        return out
```

其中 `input_spec` 参数是长度为 1 的 `list`，对应 `forward` 函数的 `inputs` 参数。`input_spec[0]` 包含了两个 `InputSpec` 对象，对应于参数 `inputs` 的两个 `Tensor` 签名信息。

当函数的参数为 `dict` 时，`input_spec` 列表中对应元素的位置，也必须是包含相同键（key）的 `InputSpec` 列表。使用样例如下：

```
class SimpleNet(Layer):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.linear = paddle.nn.Linear(10, 3)

    @to_static(input_spec=[InputSpec(shape=[None, 10], name='x'), { 'x': ↵
                           InputSpec(shape=[3], name='bias')}])
    def forward(self, x, bias_info):
        x_bias = bias_info['x']
        out = self.linear(x)
```

(下页继续)

(续上页)

```
out = out + x_bias
return out
```

其中 `input_spec` 参数是长度为 2 的 list，对应 `forward` 函数的 `x` 和 `bias_info` 两个参数。`input_spec` 的最后一个元素是包含键名为 `x` 的 `InputSpec` 对象的 dict，对应参数 `bias_info` 的 `Tensor` 签名信息。

方式四：指定非 Tensor 参数类型

目前，`to_static` 装饰器中的 `input_spec` 参数仅接收 `InputSpec` 类型对象。若被装饰函数的参数列表除了 `Tensor` 类型，还包含其他如 `Int`、`String` 等非 `Tensor` 类型时，推荐在函数中使用 `kwargs` 形式定义非 `Tensor` 参数，如下述样例中的 `use_act` 参数。

```
class SimpleNet(Layer):
    def __init__(self, ):
        super(SimpleNet, self).__init__()
        self.linear = paddle.nn.Linear(10, 3)
        self.relu = paddle.nn.ReLU()

    def forward(self, x, use_act=False):
        out = self.linear(x)
        if use_act:
            out = self.relu(out)
        return out

net = SimpleNet()
# 方式一: save inference model with use_act=False
net = to_static(net, input_spec=[InputSpec(shape=[None, 10], name='x')])
paddle.jit.save(net, path='./simple_net')

# 方式二: save inference model with use_act=True
net = to_static(net, input_spec=[InputSpec(shape=[None, 10], name='x'), True])
paddle.jit.save(net, path='./simple_net')
```

在上述样例中，假设 `step` 为奇数时，`use_act` 取值为 `False`；`step` 为偶数时，`use_act` 取值为 `True`。动转静支持非 `Tensor` 参数在训练时取不同的值，且保证了取值不同的训练过程都可以更新模型的网络参数，行为与动态图一致。

在借助 `paddle.jit.save` 保存预测模型时，动转静会根据 `input_spec` 和 `kwargs` 的默认值保存推理模型和网络参数。建议将 `kwargs` 参数默认值设置为预测时的取值。

更多关于动转静 `to_static` 搭配 `paddle.jit.save/load` 的使用方式，可以参考 [【模型的存储与载入】](#)。

三、动、静态图部署区别

当训练完一个模型后，下一阶段就是保存导出，实现模型和参数的分发，进行多端部署。如下两小节，将介绍动态图和静态图的概念和差异性，以帮助理解动转静如何起到桥梁作用的。

3.1 动态图预测部署

动态图下，**模型**指的是 Python 前端代码；**参数**指的是 `model.state_dict()` 中存放的权重数据。

```
net = SimpleNet()

# .... 训练过程(略)

layer_state_dict = net.state_dict()
paddle.save(layer_state_dict, "net.pdiparams") # 导出模型
```

上图展示了动态图下模型训练——> 参数导出——> 预测部署的流程。如图中所示，动态图预测部署时，除了已经序列化的参数文件，还须提供**最初的模型组网代码**。

如下是一个简单的 Model 示例：

```
import paddle

class SimpleNet(paddle.nn.Layer):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.linear = paddle.nn.Linear(10, 3)

    def forward(self, x, y):
        out = self.linear(x)
        out = out + y
        return out

net = SimpleNet()
```

动态图下，当实例化一个 `SimpleNet()` 对象时，隐式地执行了如下几个步骤：

- 创建一个 `Linear` 对象，记录到 `self._sub_layer` 中（dict 类型）
 - 创建一个 `ParamBase` 类型的 `weight`，记录到 `self._parameters` 中（dict 类型）
 - 创建一个 `ParamBase` 类型的 `bias`，记录到 `self._parameters` 中

一个复杂模型可能包含很多子类，框架层就是通过 `self._sub_layer` 和 `self._parameters` 两个核心数据结构关联起来的，这也是后续动转静原理上操作的两个核心属性。

```
sgd = paddle.optimizer.SGD(learning_rate=0.1, parameters=net.parameters())
```

所有待更新参数

3.2 静态图预测部署

静态图部署时，模型指的是 Program；参数指的是所有的 Persistable=True 的 Variable。二者都可以序列化导出为磁盘文件，与前端代码完全解耦。

```
main_program = paddle.static.default_main_program()

# ..... 训练过程（略）

prog_path='main_program.pdmodel'
paddle.save(main_program, prog_path) # 导出为 .pdmodel

para_path='main_program.pdiparams'
paddle.save(main_program.state_dict(), para_path) # 导出为 .pdiparams
```

上图展示了静态图下模型训练——> 模型导出——> 预测部署的流程。如图所示，静态图模型导出时将 Program 和模型参数都导出为磁盘文件，Program 中包含了模型所有的计算描述 (OpDesc)，不存在计算逻辑有遗漏的地方。

静态图编程，总体上包含两个部分：

- 编译期：组合各个 Layer 接口，搭建网络结构，执行每个 Op 的 InferShape 逻辑，最终生成 Program
- 执行期：构建执行器，输入数据，依次执行每个 OpKernel，进行训练和评估

在静态图编译期，变量 Variable 只是一个符号化表示，并不像动态图 Tensor 那样持有实际数据。

如下是 SimpleNet 的静态图模式下的组网代码：

```
import paddle
# 开启静态图模式
paddle.enable_static()

# placeholder 信息
x = paddle.static.data(shape=[None, 10], dtype='float32', name='x')
y = paddle.static.data(shape=[None, 3], dtype='float32', name='y')

out = paddle.static.nn.fc(x, 3)
out = paddle.add(out, y)
# 打印查看 Program 信息
```

(下页继续)

(续上页)

```

print(paddle.static.default_main_program())

# { // block 0
#     var x : LOD_TENSOR.shape(-1, 10).dtype(float32).stop_gradient(True)
#     var y : LOD_TENSOR.shape(-1, 3).dtype(float32).stop_gradient(True)
#     persist trainable param fc_0.w_0 : LOD_TENSOR.shape(10, 3).dtype(float32).stop_
↪gradient(False)
#     var fc_0.tmp_0 : LOD_TENSOR.shape(-1, 3).dtype(float32).stop_gradient(False)
#     persist trainable param fc_0.b_0 : LOD_TENSOR.shape(3,).dtype(float32).stop_
↪gradient(False)
#     var fc_0.tmp_1 : LOD_TENSOR.shape(-1, 3).dtype(float32).stop_gradient(False)
#     var elementwise_add_0 : LOD_TENSOR.shape(-1, 3).dtype(float32).stop_
↪gradient(False)

#     {Out=['fc_0.tmp_0']} = mul(inputs={X=['x'], Y=['fc_0.w_0']}, force_fp32_output =_
↪False, op_device = , op_namescope = /, op_role = 0, op_role_var = [], scale_out = 1.
↪0, scale_x = 1.0, scale_y = [1.0], use_mkldnn = False, x_num_col_dims = 1, y_num_
↪col_dims = 1)
#     {Out=['fc_0.tmp_1']} = elementwise_add(inputs={X=['fc_0.tmp_0'], Y=['fc_0.b_0']},_
↪Scale_out = 1.0, Scale_x = 1.0, Scale_y = 1.0, axis = 1, mkldnn_data_type =_
↪float32, op_device = , op_namescope = /, op_role = 0, op_role_var = [], use_mkldnn_
↪= False, use_quantizer = False, x_data_format = , y_data_format = )
#     {Out=['elementwise_add_0']} = elementwise_add(inputs={X=['fc_0.tmp_1'], Y=['y']},_
↪Scale_out = 1.0, Scale_x = 1.0, Scale_y = 1.0, axis = -1, mkldnn_data_type =_
↪float32, op_device = , op_namescope = /, op_role = 0, op_role_var = [], use_mkldnn_
↪= False, use_quantizer = False, x_data_format = , y_data_format = )
}

```

静态图中的一些概念：

- **Program:** 与 Model 对应，描述网络的整体结构，内含一个或多个 Block
- **Block**
 - **global_block:** 全局 Block，包含所有 Parameters、全部 Ops 和 Variables
 - **sub_block:** 控制流，包含控制流分支内的所有 Ops 和必要的 Variables
- **OpDesc:** 对应每个前端 API 的计算逻辑描述
- **Variable:** 对应所有的数据变量，如 Parameter，临时中间变量等，全局唯一 name。

注：更多细节，请参考 [【官方文档】模型的存储与载入。](#)

转换原理

在框架内部, 动转静模块在转换上主要包括对 InputSpec 的处理, 对函数调用的递归转写, 对 IfElse、For、While 控制语句的转写, 以及 Layer 的 Parameters 和 Buffers 变量的转换。下面将从这四个方面介绍动转静模块的转换原理。

一、设置 Placeholder 信息

静态图下, 模型起始的 Placeholder 信息是通过 paddle.static.data 来指定的, 并以此作为编译期的 InferShape 推导起点, 即用于推导输出 Tensor 的 shape。

```
import paddle
# 开启静态图模式
paddle.enable_static()

# placeholder 信息
x = paddle.static.data(shape=[None, 10], dtype='float32', name='x')
y = paddle.static.data(shape=[None, 3], dtype='float32', name='y')

out = paddle.static.nn.fc(x, 3)
out = paddle.add(out, y)
```

动转静代码示例, 通过 InputSpec 设置 Placeholder 信息:

```
import paddle
from paddle.jit import to_static

class SimpleNet(paddle.nn.Layer):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.linear = paddle.nn.Linear(10, 3)

    @to_static
    def forward(self, x, y):
        out = self.linear(x)
        out = out + y
        return out

net = SimpleNet()

# 通过 InputSpec 设置 Placeholder 信息
x_spec = InputSpec(shape=[None, 10], name='x')
y_spec = InputSpec(shape=[3], name='y')
```

(下页继续)

(续上页)

```
net = paddle.jit.to_static(net, input_spec=[x_spec, y_spec]) # 动静转换
```

在导出模型时，需要显式地指定输入 Tensor 的签名信息，优势是：

- 可以指定某些维度为 None，如 batch_size, seq_len 维度
- 可以指定 Placeholder 的 name，方便预测时根据 name 输入数据

注：InputSpec 接口的高阶用法，请参看 [【使用 InputSpec 指定模型输入 Tensor 信息】](#)

二、函数转写

在 NLP、CV 领域中，一个模型常包含层层复杂的子函数调用，动静转中是如何实现只需装饰最外层的 **forward** 函数，就能递归处理所有的函数。

如下是一个模型样例：

```
import paddle
from paddle.jit import to_static

class SimpleNet(paddle.nn.Layer):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.linear = paddle.nn.Linear(10, 3)

    @to_static
    def forward(self, x, y):
        out = self.my_fc(x)           # <---- self.other_func
        out = add_two(out, y)         # <---- other plain func
        return out

    def my_fc(self, x):
        out = self.linear(x)
        return out

# 此函数可以在任意文件
def add_two(x, y):
    out = x + y
    return out

net = SimpleNet()
# 查看转写的代码内容
paddle.jit.set_code_level(100)

x = paddle.zeros([2, 10], 'float32')
```

(下页继续)

(续上页)

```
y = paddle.zeros([3], 'float32')

out = net(x, y)
```

可以通过 `paddle.jit.set_code_level(100)` 在执行时打印代码转写的结果到终端，转写代码如下：

```
def forward(self, x, y):
    out = paddle.jit.dy2static.convert_call(self.my_fc)(x)
    out = paddle.jit.dy2static.convert_call(add_two)(out, y)
    return out

def my_fc(self, x):
    out = paddle.jit.dy2static.convert_call(self.linear)(x)
    return out

def add_two(x, y):
    out = x + y
    return out
```

如上所示，所有的函数调用都会被转写如下形式：

out = paddle.jit.dy2static.convert_call(self.my_fc) (x)	
^	^	^	^
返回列表	convert_call	原始函数	参数列表

即使函数定义分布在不同的文件中，`convert_call` 函数也会递归地处理和转写所有嵌套的子函数。

三、控制流转写

控制流 `if/for/while` 的转写和处理是动转静中比较重要的模块，也是动态图模型和静态图模型实现上差别最大的一部分。如下图所示，对于控制流的转写分为两个阶段：转写期和执行期。在转写期，动转静模块将控制流语句转写为统一的形式；在执行期，根据控制流是否依赖 `Tensor` 来决定是否将控制流转写为相应的 `cond_op/while_op`。

转写上有两个基本原则：

- 并非所有动态图中的 `if/for/while` 都会转写为 `cond_op/while_op`
- 只有控制流的判断条件 依赖了 `Tensor` (如 `shape` 或 `value`)，才会转写为对应 Op

3.1 IfElse

无论是否会转写为 cond_op，动静都会首先对代码进行处理，转写为 **cond** 接口可以接受的写法

示例一：不依赖 Tensor 的控制流

如下代码样例中的 if label is not None, 此判断只依赖于 label 是否为 None (存在性)，并不依赖 label 的 Tensor 值 (数值性)，因此属于**不依赖 Tensor 的控制流**。

```
def not_depend_tensor_if(x, label=None):
    out = x + 1
    if label is not None:           # <---- python bool 类型
        out = paddle.nn.functional.cross_entropy(out, label)
    return out

print(to_static(not_depend_tensor_if).code)
# 转写后的代码：
"""
def not_depend_tensor_if(x, label=None):
    out = x + 1

    def true_fn_1(label, out):  # true 分支
        out = paddle.nn.functional.cross_entropy(out, label)
        return out

    def false_fn_1(out):        # false 分支
        return out

    out = paddle.jit.dy2static.convert_ifelse(label is not None, true_fn_1,
                                              false_fn_1, (label, out), (out,), (out,))
    return out
"""


```

示例二：依赖 Tensor 的控制流

如下代码样例中的 if paddle.mean(x) > 5, 此判断直接依赖 paddle.mean(x) 返回的 Tensor 值 (数值性)，因此属于**依赖 Tensor 的控制流**。

```
def depend_tensor_if(x):
    if paddle.mean(x) > 5.:           # <---- Bool Tensor 类型
        out = x - 1
    else:
        out = x + 1
    return out
```

(下页继续)

(续上页)

```

print(to_static(depend_tensor_if).code)
# 转写后的代码：

"""
def depend_tensor_if(x):
    out = paddle.jit.dy2static.data_layer_not_check(name='out', shape=[-1],
        dtype='float32')

    def true_fn_0(x):      # true 分支
        out = x - 1
        return out

    def false_fn_0(x):     # false 分支
        out = x + 1
        return out

    out = paddle.jit.dy2static.convert_ifelse(paddle.mean(x) > 5.0,
        true_fn_0, false_fn_0, (x,), (x,), (out,))

    return out
"""

```

规范化代码之后，所有的 IfElse 均转为了如下形式：

out = convert_ifelse(paddle.mean(x) > 5.0, true_fn_0, false_fn_0, (x,), (x,), (out,))	^ ^ ^ ^ ^ ^ ^ ^
输出 convert_ifelse	判断条件
分支输入 ↗	true 分支
输出 ↗	false 分支
	分支输入 ↗

convert_ifelse 是框架底层的函数，在逐行执行用户代码生成 Program 时，执行到此处时，会根据判断条件的类型 (bool 还是 Bool Tensor)，自适应决定是否转为 cond_op。

```

def convert_ifelse(pred, true_fn, false_fn, true_args, false_args, return_vars):

    if isinstance(pred, Variable): # 触发 cond_op 的转换
        return _run_paddle_cond(pred, true_fn, false_fn, true_args, false_args,
                               return_vars)
    else:                      # 正常的 python if
        return _run_py_ifelse(pred, true_fn, false_fn, true_args, false_args)

```

3.2 For/While

For/While 也会先进行代码层面的规范化，在逐行执行用户代码时，才会决定是否转为 while_op。

示例一：不依赖 Tensor 的控制流如下代码样例中的 while $a < 10$, 此循环条件中的 a 是一个 int 类型，并不是 Tensor 类型，因此属于不依赖 Tensor 的控制流。

```
def not_depend_tensor_while(x):
    a = 1

    while a < 10:           # ----- a is python scalar
        x = x + 1
        a += 1

    return x

print(to_static(not_depend_tensor_while).code)
"""

def not_depend_tensor_while(x):
    a = 1

    def while_condition_0(a, x):
        return a < 10

    def while_body_0(a, x):
        x = x + 1
        a += 1
        return a, x

    [a, x] = paddle.jit.dy2static.convert_while_loop(while_condition_0,
                                                    while_body_0, [a, x])

    return x
"""


```

示例二：依赖 Tensor 的控制流

如下代码样例中的 for i in range(bs), 此循环条件中的 bs 是一个 paddle.shape 返回的 Tensor 类型，且将其 Tensor 值作为了循环的终止条件，因此属于依赖 Tensor 的控制流。

```
def depend_tensor_while(x):
    bs = paddle.shape(x)[0]

    for i in range(bs):       # ----- bas is a Tensor
        x = x + 1
```

(下页继续)

(续上页)

```

    return x

print(to_static(depend_tensor_while).code)
"""
def depend_tensor_while(x):
    bs = paddle.shape(x)[0]
    i = 0

    def for_loop_condition_0(x, i, bs):
        return i < bs

    def for_loop_body_0(x, i, bs):
        x = x + 1
        i += 1
        return x, i, bs

    [x, i, bs] = paddle.jit.dy2static.convert_while_loop(for_loop_condition_0,
        for_loop_body_0, [x, i, bs])
    return x
"""

```

convert_while_loop 的底层的逻辑同样会根据 判断条件是否为 **Tensor** 来决定是否转为 while_op

四、Parameters 与 Buffers

4.1 动态图 layer 生成 Program

文档开始的样例中 forward 函数包含两行组网代码: Linear 和 add 操作。以 Linear 为例，在 Paddle 的框架底层，每个 Paddle 的组网 API 的实现包括两个分支:

```

class Linear(...):
    def __init__(self, ...):
        # ... (略)

    def forward(self, input):

        if in_dygraph_mode():  # 动态图分支
            core.ops.matmul(input, self.weight, pre_bias, ...)
            return out
        else:                  # 静态图分支
            self._helper.append_op(type="matmul", inputs=inputs, ...)      # <-----』

```

(下页继续)

→ 生成一个 Op

(续上页)

```

if self.bias is not None:
    self._helper.append_op(type='elementwise_add', ...)
# <-----_
→ 生成一个 Op

return out

```

动态图 layer 生成 Program，其实是开启 paddle.enable_static() 时，在静态图下逐行执行用户定义的组网代码，依次添加(对应 append_op 接口)到默认的主 Program (即 main_program) 中。

4.2 动态图 Tensor 转为静态图 Variable

上面提到，所有的组网代码都会在静态图模式下执行，以生成完整的 Program。但静态图 **append_op** 有一个前置条件必须满足：

前置条件: append_op() 时，所有的 inputs, outputs 必须都是静态图的 Variable 类型，不能是动态图的 Tensor 类型。

原因: 静态图下，操作的都是**描述类单元**：计算相关的 OpDesc，数据相关的 varDesc。可以分别简单地理解为 Program 中的 Op 和 Variable。

因此，在动转静时，我们在需要在某个统一的入口处，将动态图 Layers 中 Tensor 类型（包含具体数据）的 Weight、Bias 等变量转换为同名的静态图 **Variable**。

- ParamBase → Parameters
- VarBase → Variable

技术实现上，我们选取了框架层面两个地方作为类型转换的入口：

- Paddle.nn.Layer 基类的 __call__ 函数

```

def __call__(self, *inputs, **kwargs):
    # param_guard 会对将 Tensor 类型的 Param 和 buffer 转为静态图 Variable
    with param_guard(self._parameters), param_guard(self._buffers):
        # ... forward_pre_hook 逻辑

        outputs = self.forward(*inputs, **kwargs) # 此处为 forward 函数

        # ... forward_post_hook 逻辑

    return outputs

```

- Block.append_op 函数中，生成 Op 之前

```

def append_op(self, *args, **kwargs):
    if in_dygraph_mode():

```

(下页继续)

(续上页)

```
# ... (动态图分支)

else:
    inputs=kargs.get("inputs", None)
    outputs=kargs.get("outputs", None)
    # param_guard 会确保将 Tensor 类型的 inputs 和 outputs 转为静态图 Variable
    with param_guard(inputs), param_guard(outputs):
        op = Operator(
            block=self,
            desc=op_desc,
            type=kargs.get("type", None),
            inputs=inputs,
            outputs=outputs,
            attrs=kargs.get("attrs", None))
```

以上，是动态图转为静态图的两个核心逻辑，总结如下：

- 动态图 layer 调用在动转静时会走底层 append_op 的分支，以生成 Program
- 动态图 Tensor 转为静态图 Variable，并确保编译期的 InferShape 正确执行

4.3 Buffer 变量

什么是 **Buffers** 变量？

- **Parameters**: persistable 为 True，且每个 batch 都被 Optimizer 更新的变量
- **Buffers**: persistable 为 True，is_trainable = False，不参与更新，但与预测相关；如 BatchNorm 层中的均值和方差

在动态图模型代码中，若一个 paddle.to_tensor 接口生成的 Tensor 参与了最终预测结果的计算，则此 Tensor 需要在转换为静态图预测模型时，也需要作为一个 persistable 的变量保存到 .pdiparam 文件中。

举一个例子（错误写法）：

```
import paddle
from paddle.jit import to_static

class SimpleNet(paddle.nn.Layer):
    def __init__(self, mask):
        super(SimpleNet, self).__init__()
        self.linear = paddle.nn.Linear(10, 3)

        # mask value, 此处不会保存到预测模型文件中
        self.mask = mask # 假设为 [0, 1, 1]
```

(下页继续)

(续上页)

```
def forward(self, x, y):
    out = self.linear(x)
    out = out + y
    mask = paddle.to_tensor(self.mask)      # <---- 每次执行都转为一个 Tensor
    out = out * mask
    return out
```

推荐的写法是：

```
class SimpleNet(paddle.nn.Layer):
    def __init__(self, mask):
        super(SimpleNet, self).__init__()
        self.linear = paddle.nn.Linear(10, 3)

        # 此处的 mask 会当做一个 buffer Tensor, 保存到 .pdiparam 文件
        self.mask = paddle.to_tensor(mask) # 假设为 [0, 1, 1]

    def forward(self, x, y):
        out = self.linear(x)
        out = out + y
        out = out * self.mask           # <---- 直接使用 self.mask
        return out
```

总结一下 buffers 的用法：

- 若某个非 Tensor 数据需要当做 Persistable 的变量序列化到磁盘，则最好在 `__init__` 中调用 `self.XX = paddle.to_tensor(xx)` 接口转为 buffer 变量

支持语法

一、主要针对场景

本文档概览性介绍了飞桨动转静功能的语法支持情况，旨在提供一个便捷的语法速查表，主要适用于如下场景：

1. 不确定当前动态图模型是否可以正确转化为静态图
2. 转化过程中出现了问题但不知道如何排查
3. 当出现不支持的语法时，如何修改源码适配动转静语法

若你初次接触动转静功能，或对此功能尚不熟悉，推荐阅读：使用样例；

若你动静转换遇到了问题，或想学习调试的技巧，推荐阅读：报错调试。

二、语法支持速查列表

三、详细说明

3.1 if-else

主要逻辑: 在动态图中，模型代码是一行一行解释执行的，因此控制流的条件变量是在运行期确定的，意味着 False 的逻辑分支不会被执行。

在静态图中，控制流通过 cond 接口实现。每个分支分别通过 true_fn 和 false_fn 来表示。

当 if 中的条件是 Tensor 时，动转静会自动把该 if-elif-else 语句转化为静态图的 cond API 语句。

当 if 中的条件不是 Tensor 时，会按普通 Python if-else 的逻辑运行。

注：当条件为 Tensor 时，只接受 numel() == 1 的 bool Tensor，否则会报错。

错误修改指南:

当模型代码中的 if-else 转换或执行报错时，可以参考如下方式排查：

- 使用 if 语句时，请确定条件变量是否是 Paddle.Tensor 类型。若不是 Tensor 类型，则会按照常规的 python 逻辑执行，而不会转化为静态图。
- 若 if 中条件变量为 Tensor 类型，需确保其为 boolean 类型，且 tensor.numel() 为 1。

3.2 while 循环

主要逻辑:

当 while 循环中的条件是 Tensor 时，动转静会把该 while 语句转化为静态图中的 while_loop API 语句，否则会按普通 Python while 运行。

注：while 循环条件中的 Tensor 须是 numel 为 1 的 bool Tensor，否则会报错。

错误修改指南:

类似 if-elif-else，注意事项也相似。

3.3 for 循环

主要逻辑:

for 循环按照使用方法的不同，语义有所不同。正常而言，for 循环的使用分为如下种类：

- for _ in range(len) 循环：动转静会先将其转化为等价的 Python while 循环，然后按 while 循环的逻辑进行动静转换。
- for _ in x 循环：当 x 是 Python 容器或迭代器，则会用普通 Python 逻辑运行。当 x 是 Tensor 时，会转化为依次获取 x[0], x[1], ...。
- for idx, val in enumerate(x) 循环：当 x 是 Python 容器或迭代器，则会用普通 Python 逻辑运行。当 x 是 Tensor 时，idx 会转化为依次 0, 1, ... 的 1-D Tensor。val 会转化为循环中每次对应拿出 x[0], x[1], ...。

从实现而言，for 循环最终会转化为对应的 while 语句，然后使用 WhileOp 来进行组网。

使用样例：

此处使用上述 For 的第二个用法举例。如果 x 是一个多维 Tensor，则也是返回 x[0], x[1], ...

```
def ForTensor(x):
    """Fetch element in x and print the square of each x element"""
    for i in x:
        print(i * i)

# 调用方法, ForTensor(paddle.to_tensor(x))
```

3.4 流程控制语句说明 (return / break / continue)

主要逻辑:

目前的动转静支持 for、while 等循环中添加 break, continue 语句改变控制流，也支持在循环内部任意位置添加 return 语句，支持 return 不同长度 tuple 和不同类型的 Tensor。

使用样例：

```
# break 的使用样例
def break_usage(x):
    tensor_idx = -1
    for idx, val in enumerate(x):
        if val == 2.0:
            tensor_idx = idx
```

(下页继续)

(续上页)

```
break # <----- jump out of while loop when break ;
return tensor_idx
```

当时输入 `x = Tensor([1.0, 2.0, 3.0])` 时，输出的 `tensor_idx` 是 `Tensor([1])`。

注：这里虽然 `idx` 是 -1，但是返回值还是 `Tensor`。因为 `tensor_idx` 在 `while loop` 中转化为了 `Tensor`。

3.5 与、或、非

主要逻辑：

动静模块支持将与、或、非三种运算符进行转换并动态判断，按照两个运算符 `x` 和 `y` 的不同，会有不同的语义：

- 如果运算符两个都是 `Tensor`，会组网静态图。
- 如果运算符都不是 `Tensor`，那么使用原始 `python` 语义。
- 如果一个是 `Tensor`，那么会走默认的 `python` 语义（最后还是 `tensor` 的运算符重载结果）

注：若按照 `paddle` 的语义执行，与、或、非不再支持 `lazy` 模式，意味着两个表达式都会被 `eval`，而不是按照 `x` 的值来判断是否对 `y` 进行 `eval`。

使用样例：

```
def and(x, y):
    z = y and x
    return z
```

3.6 类型转换运算符

主要逻辑：

动态图中可以直接用 `Python` 的类型转化语法来转化 `Tensor` 类型。如若 `x` 是 `Tensor` 时，`float(x)` 可以将 `x` 的类型转化为 `float`。

动静在运行时判断 `x` 是否是 `Tensor`，若是，则在动静时使用静态图 `cast` 接口转化相应的 `Tensor` 类型。

使用样例：

```
def float_convert(x):
    z = float(x)
```

(下页继续)

(续上页)

```
    return z
# 如果输入是 x = Tensor([True]) , 则 z = Tensor([1.0])
```

3.7 对一些 python 函数调用的转换

主要逻辑：

动转静支持大部分的 python 函数调用。函数调用都会被统一包装成为 `convert_xxx()` 的形式，在函数运行期判别类型。若是 Paddle 类型，则转化为静态图的组网；反之则按照原来的 python 语义执行。常见函数如下：

- `print` 函数若参数是 `Tensor`，在动态图模式中 `print(x)` 可以打印 `x` 的值。动转静时会转化为静态图的 `Print` 接口实现；若参数不是 `Tensor`，则按照 Python 的 `print` 语句执行。
- `len` 函数若 `x` 是 `Tensor`，在动态图模式中 `len(x)` 可以获得 `x` 第 0 维度的长度。动转静时会转化为静态图 `shape` 接口，并返回 `shape` 的第 0 维。若 `x` 是个 `TensorArray`，那么 `len(x)` 将会使用静态图接口 `control_flow.array_length` 返回 `TensorArray` 的长度；对于其他情况，会按照普通 Python `len` 函数运行。
- `lambda` 表达式动转静允许写带有 Python `lambda` 表达式的语句，并且我们会适当改写使得返回对应结果。
- 函数内再调用函数（非递归调用）对于函数内调用其他函数的情况，动转静会对内部的函数递归地进行识别和转写，以实现在最外层函数只需加一次装饰器即可的效果。

使用样例：

这里以 `lambda` 函数为例，展示使用方法

```
def lambda_call(x):
    t = lambda x : x * x
    z = t(x)
    return z
# 如果输入是 x = Tensor([2.0]) , 则 z = Tensor([4.0])
```

不支持用法：

- 函数的递归调用

动转静暂不支持一个函数递归调用本身。原因是递归常常会用 `if-else` 构造停止递归的条件。此停止条件在静态图下只是一个 `cond` 组网，并不能在编译阶段得到递归条件的具体值，会导致函数运行时一直组网递归直至栈溢出。

```
def recur_call(x):
    if x > 10:
        return x
    return recur_call(x * x) # < ----- 如果输入是 x = Tensor([2.0]) , 动态图输出为
    ↪Tensor([16]), 静态图会出现调用栈溢出
```

3.8 List 和 Dict 容器

主要逻辑：

- List：若一个 list 的元素都是 Tensor，动转静将其转化为 TensorArray。静态图 TensorArray 仅支持 append, pop, 修改操作，其他 list 操作（如 sort）暂不支持。若并非所有元素是 Tensor，动转静会将其作为普通 Python list 运行。
- Dict：动转静支持原生的 Python dict 语法。

注：List 不支持多重嵌套和其他的操作。具体错误案例见下面不支持用法。

使用样例：

```
def list_example(x, y):
    a = [x] # < ----- 支持直接创建
    a.append(x) # < ----- 支持调用append、pop操作
    a[1] = y # < ----- 支持下标修改append
    return a[0] # < ----- 支持下标获取
```

不支持用法：

- List 的多重嵌套

如 `l = [[tensor1, tensor2], [tensor3, tensor4]]`，因为现在动转静将元素全是 Tensor 的 list 转化为 TensorArray，但 TensorArray 还不支持多维数组，因此这种情况下，动转静无法正确运行。遇到这类情况我们建议尽量用一维 list，或者自己使用 PaddlePaddle 的 `create_array`, `array_read`, `array_write` 接口编写为 TensorArray。

- List 的其他的操作，例如 sort 之类

```
# 不支持的list sort 操作
def sort_list(x, y):
    a = [x, y]
    sort(a) # < ----- ↪
    ↪不支持，因为转化为TensorArray之后不支持sort操作。但是支持简单的append,
    ↪pop和按下标修改
    return a
```

3.9 paddle shape 函数

主要逻辑:

动转静部分支持 shape 函数:

- 【支持】当直接简单的使用 shape 时，可以正确获取 tensor 的 shape。
- 【不支持】当直接使用支持改变变量的 shape 后(例如 reshape 操作)调用其 shape 作为 PaddlePaddle API 参数。

如 `x = reshape(x, shape=shape_tensor)`，再使用 `x.shape[0]` 的值进行其他操作。这种情况会由于动态图和静态图的本质不同而使得动态图能够运行，但静态图运行失败。其原因是动态图情况下，API 是直接返回运行结果，因此 `x.shape` 在经过 `reshape` 运算后是确定的。但是在转化为静态图后，因为静态图 API 只是组网，`shape_tensor` 的值在组网时是不知道的，所以 `reshape` 接口组网完，静态图并不知道 `x.shape` 的值。PaddlePaddle 静态图用-1 表示未知的 shape 值，此时 `x` 的 shape 每个维度会被设为-1，而不是期望的值。同理，类似 `expand` 等更改 shape 的 API，其输出 Tensor 再调用 shape 也难以进行动转静。

使用样例:

```
def get_shape(x):
    return x.shape[0]
```

不支持用法举例:

```
def error_shape(x, y):
    y = y.cast('int32')
    t = x.reshape(y)
    return t.shape[0] # ----- 输入在x = Tensor([2.0, 1.0]), y =
    ↪Tensor([2])时，动态图输出为2，而静态图输出为 -1。不支持
```

案例解析

在【使用样例】章节介绍了动转静的用法和机制，下面会结合一些具体的模型代码，解答动转静中比较常见的问题。

一、@to_static 放在哪里？

`@to_static` 装饰器开启启动转静功能的唯一接口，支持两种使用方式:

- 方式一（推荐用法）：显式地通过 `model = to_static(model)` 调用

```
from paddle.jit import to_static
```

(下页继续)

(续上页)

```
model = SimpleNet()
model = to_static(model, input_spec=[x_spec, y_spec])
```

- 方式二：在组网代码的 forward 函数处装饰

```
class SimpleNet(paddle.nn.Layer):
    def __init__(self, ...):
        # .....

    @to_static
    def forward(self, x, y):
        # .....
        return out
```

如果只是进行预测模型导出，推荐使用方式一，优势在于：

- 使用方式更简洁，不用特意去找模型的 forward 函数在哪
- 与其他模块解耦，预测模型的导出逻辑可以单独一个模块

需要注意的地方：

- 默认是将 model 的 forward 函数作为入口函数
- 建议模型搭建时，尽量考虑将预测主逻辑放到 forward 函数中
 - 将训练独有的逻辑放到子函数中，通过 if self.training 来控制
 - 最大程度抽离训练和预测的逻辑为 公共子函数

二、何时指定 InputSpec？

在动转静的原理介绍中，静态图 Program 的生成需要依赖 Placeholder 信息，此信息可通过两种方式获得：

- 方式一（推荐）：在 @to_static 接口中指定 input_spec 参数，显式地提供每个输入 Variable 的 Placeholder 信息

```
model = SimpleNet()

x_spec = InputSpec(shape=[None, 10], name='x')  # 动态 shape
y_spec = InputSpec(shape=[3], name='y')

net = paddle.jit.to_static(net, input_spec=[x_spec, y_spec])
```

- 方式二：输入具体的 Tensor(s) 数据，显式地执行一次前向，以此 Tensor(s) 的 shape 和 dtype 作为 Placeholder 信息

```
# 假设：模型 forward 定义处已经被 @to_static 装饰了
model = SimpleNet()

x = paddle.randn([4, 10], 'float32')
y = paddle.randn([3], 'float32')

out = model(x, y)      # 执行一次前向，触发 Program 的转换
paddle.jit.save(model, './simple_net')
```

- 优点：直接用输入数据，简单方便
- 缺点：无法指定动态 shape，如 batch_size, seq_len 等。

注：InputSpec 接口的高阶用法，请参看 [【使用 InputSpec 指定模型输入 Tensor 信息】](#)

三、内嵌 Numpy 操作？

动态图模型代码中的 numpy 相关的操作可以转为静态图么？

答：不能。所有与组网相关的 numpy 操作都必须用 paddle 的 API 重新实现。即不支持 Layer → numpy operations → Layer 的组网方式。

原因：

- 静态图 Program 的计算逻辑描述单元是 Op
- numpy 操作无法识别为框架的 Op，须用 Paddle API 重新实现才可以

举个例子：

```
def forward(self, x):
    out = self.linear(x)  # [bs, 3]

    # 以下将 tensor 转为了 numpy 进行一系列操作
    x_data = x.numpy().astype('float32')  # [bs, 10]
    weight = np.random.randn([10, 3])
    mask = paddle.to_tensor(x_data * weight)  # 此处又转回了 Tensor

    out = out * mask
    return out
```

上述样例需要将 forward 中的所有的 numpy 操作都转为 Paddle API：

```
def forward(self, x):
    out = self.linear(x)  # [bs, 3]

    weight = paddle.randn([10, 3], 'float32')
```

(下页继续)

(续上页)

```

mask = x * weight

out = out * mask
return out

```

在之前排查的模型中，存在较多中间转为 numpy 的操作，会无法生成完整的 Program，并导致：

- 模型动转静时报各种奇怪的错误
- 可以转换成功，但加载模型预测时，可能会报 **Segment Fault** 等错误

若遇到类似报错，建议排查下模型代码是否存在此类写法。

四、`to_tensor()` 的使用

`paddle.to_tensor()` 接口是动态图模型代码中使用比较频繁的一个接口。`to_tensor` 功能强大，将可以将一个 scalar, list, tuple, numpy.ndarray 转为 `paddle.Tensor` 类型。

此接口是动态图独有的接口，在动转静时，会转换为 `assign` 接口：

```

import paddle
import numpy as np

# 动态图
x = paddle.to_tensor(np.array([2, 3, 4]))

# 动转静后代码
x = paddle.assign(np.array([2, 3, 4]))

```

举个比较常见的例子（错误写法）：

```

class SimpleNet(paddle.nn.Layer):
    def __init__(self, mask):
        super(SimpleNet, self).__init__()
        self.linear = paddle.nn.Linear(10, 3)
        self.mask = np.array(mask) # 假设为 [0, 1, 1]

    def forward(self, x, y):
        out = self.linear(x)
        out = out + y

        mask = paddle.to_tensor(self.mask) # ----- 每次都会调用 assign_op
        out = out * mask

    return out

```

推荐的写法：

```
class SimpleNet(paddle.nn.Layer):
    def __init__(self, mask):
        super(SimpleNet, self).__init__()
        self.linear = paddle.nn.Linear(10, 3)
        self.mask = paddle.to_tensor(mask) # <---- 转为 buffers

    def forward(self, x, y):
        out = self.linear(x)
        out = out + y

        out = out * self.mask # <---- 省去重复的assign_op, 性能更佳

    return out
```

对于 to_tensor 的使用，建议是：

- 推荐尽量放到 __init__ 函数中一次性进行初始化

五、建议都继承 nn.Layer

动态图模型常常包含很多嵌套的子网络，建议各个自定义的子网络 sublayer 无论是否包含了参数，都继承 nn.Layer。

从 Parameters 和 Buffers 章节可知，有些 paddle.to_tensor 接口转来的 Tensor 也可能参与预测逻辑分支的计算，即模型导出时，也需要作为参数序列化保存到 .pdiparams 文件中。

原因：若某个 sublayer 包含了 buffer Variables，但却没有继承 nn.Layer，则可能导致保存的 .pdiparams 文件缺失部分重要参数。

举个例子：

```
class SimpleNet(object): # <---- 继承 Object
    def __init__(self, mask):
        super(SimpleNet, self).__init__()
        self.linear = paddle.nn.Linear(10, 3) # <---- Linear 参数永远都不会被更新
        self.mask = paddle.to_tensor(mask) # <---- mask 可能未保存到 .pdiparams
        # 文件中

    def forward(self, x, y):
        out = self.linear(x)
        out = out + y
        out = out * self.mask
    return out
```

同时，所有继承 nn.Layer 的 sublayer 都建议：

- 重写 forward 函数，尽量避免重写“call”函数

`__call__` 函数通常会包含框架层面的一些通用的处理逻辑，比如 `pre_hook` 和 `post_hook`。重写此函数可能会覆盖框架层面的逻辑。

- 尽量将 forward 函数作为 sublayers 的调用入口

推荐这样写，但动静转也支持对 sublayers 的其他函数转写处理

六、forward 函数推荐写法

6.1 默认参数

模型的 forward 函数的入参可能除了 Tensor 类型之外，还有很多其他复杂的类型，如 str、float、bool 等非 Tensor 类型。

```
class SimpleNet(paddle.nn.Layer):
    def __init__(self, mask):
        super(SimpleNet, self).__init__()
        self.linear = paddle.nn.Linear(10, 3)
        self.mask = paddle.to_tensor(mask)

    def forward(self, x, y, cmd='bn', rate=0.1, flag=False):    # <-- 默认参数
        out = self.linear(x)
        out = out + y
        # .... (略)
        out = out * self.mask
        return out
```

关于所有子函数中的非 Tensor 类型参数：

- 建议都提供一个默认的取值
- 建议默认值最好取预测时的值

原因：`jit.save` 导出预测模型时，提供了 `input_spec` 参数用于指定 Placeholder 信息。目前仅支持指定 Tensor 类型信息，非 Tensor 类型信息均使用函数定义的默认值。

6.2 train 和 infer 分支

模型的 forward 等子函数常同时包含 训练和 预测两个分支的代码逻辑。

```
def forward(self, x):
    if self.training:
        out = paddle.mean(x)
    else:
```

(下页继续)

(续上页)

```

    out = paddle.sum(x)

    return out

model = SimpleNet()
model.eval()          # <---- 一键切换分支，则只会导出 eval 相关的预测分支

jit.save(mode, model_path)

```

推荐使用 `self.training` 或其他非 `Tensor` 类型的 `bool` 值进行区分。

此 `flag` 继承自 `nn.Layer`，因此可通过 `model.train()` 和 `model.eval()` 来全局切换所有 `sublayers` 的分支状态。

七、非 forward 函数导出

`@to_static` 与 `jit.save` 接口搭配也支持导出非 `forward` 的其他函数，具体使用方式如下：

```

class SimpleNet(paddle.nn.Layer):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.linear = paddle.nn.Linear(10, 3)

    def forward(self, x, y):
        out = self.linear(x)
        out = out + y
        return out

    def another_func(self, x):
        out = self.linear(x)
        out = out * 2
        return out

net = SimpleNet()
# train(net) # 模型训练

# step 1: 切换到 eval() 模式 (同上)
net.eval()

# step 2: 定义 InputSpec 信息 (同上)
x_spec = InputSpec(shape=[None, 3], dtype='float32', name='x')

# step 3: @to_static 装饰

```

(下页继续)

(续上页)

```
static_func = to_static(net.another_func, input_spec=[x_spec])

# step 4: 调用 jit.save 接口
net = paddle.jit.save(static_func, path='another_func')
```

使用上的区别主要在于：

- **@to_static** 装饰：导出其他函数时需要显式地用 `@to_static` 装饰，以告知动静转换模块将其识别、并转为静态图 Program；
- **save** 接口参数：调用 `jit.save` 接口时，需将上述被 `@to_static` 装饰后的函数作为参数；

执行上述代码样例后，在当前目录下会生成三个文件：

```
another_func.pdiparams      // 存放模型中所有的权重数据
another_func.pdimodel       // 存放模型的网络结构
another_func.pdiparams.info // 存放额外的其他信息
```

关于动转静 `@to_static` 的用法，以及搭配 `paddle.jit.save` 接口导出预测模型的用法案例，可以参考 [使用样例](#)。

八、再谈控制流

前面 [【控制流转写】([./basic_usage_cn.html#sikongzhiliuzhuanxie](#))] 提到，不论控制流 `if/for/while` 语句是否需要转为静态图中的 `cond_op/while_op`，都会先进行代码规范化，如 `IfElse` 语句会规范为如下范式：

```
def true_fn_0(out):
    # ...
    return out

def false_fn_0(out):
    # ...
    return out

out = convert_ifelse(paddle.mean(x) > 5.0, true_fn_0, false_fn_0, (x,), (x,), (out,))
^           ^           ^           ^           ^           ^
|           |           |           |           |           |
输出     convert_ifelse      判断条件      true分支   false分支 分支输入 分支输入_
→ 输出
```

8.1 list 与 LoDTensorArray

当控制流中，出现了 `list.append` 类似语法时，情况会有一点点特殊。

Paddle 框架中的 `cond_op` 和 `while_loop` 对输入和返回类型有一个要求：

输入或者返回类型必须是：LoDTensor 或者 LoDTensorArray 即：不支持其他非 LoDTensor 类型

因此控制流中类似：

```
def forward(self, x):
    bs = paddle.shape(x)[0]
    outs = []                      # <----- list 类型
    for i in range(bs):            # <----- 依赖 Tensor 的 for
        outs.append(x)             # <----- list.append

    return outs
```

转写之后的代码：

```
def forward(x):
    bs = paddle.shape(x)[0]      # <---- bs 是个静态图 Variable, shape = (1, )
    outs = paddle.tensor.create_array(dtype='float32')      # <-- list 转为 LoDTensorArray
    i = 0

    def for_loop_condition_0(outs, bs, i, x):
        return i < bs

    def for_loop_body_0(outs, bs, i, x):
        paddle.tensor.array_write(x=x, i=paddle.tensor.array_length(outs),
                                  array=outs)                         # <---- list.append() 转为 array_write
        i += 1
        return outs, bs, i, x

    [outs, bs, i, x] = paddle.jit.dy2static.convert_while_loop(
        for_loop_condition_0, for_loop_body_0, [outs, bs, i, x])

return outs
```

关于控制流中包含 `list` 相关操作的几点说明：

- 并非所有的 `list` 都会转为 `LoDTensorArray`

只有在此控制流语句是依赖 `Tensor` 时，才会触发 `list → LoDTensorArray` 的转换

- 暂不支持依赖 Tensor 的控制流中，使用多层嵌套的 `list.append` 操作

```
def forward(x):
    bs = paddle.shape(x)[0]
    outs = []                      # <---- 多层嵌套 list

    for i in range(bs):
        outs[0].append(x)

    return outs
```

因为框架底层的 `LoDTensorArray = std::vector< LoDTensor >`，不支持两层以上 `vector` 嵌套

8.2 x.shape 与 paddle.shape(x)

模型中比较常见的控制流转写大多数与 `batch_size` 或者 `x.shape` 相关。

`x.shape[i]` 的返回值可能是固定的值，也可能是 `None`，表示动态 `shape`（如 `batch_size`）。

如果比较明确 `x.shape[i]` 对应的是 动态 `shape`，推荐使用 `paddle.shape(x)[i]`

如上面的例子：

```
def forward(self, x):
    bs = paddle.shape(x)[0]          # <---- x.shape[0] 表示 batch_size, 动态 shape
    outs = []
    for i in range(bs):
        outs.append(x)

    return outs
```

动态 `shape` 推荐使用 `paddle.shape(x)[i]`，动转静也对 `x.shape[i]` 做了很多兼容处理。前者写法出错率可能更低些。

九、jit.save 与默认参数

最后一步是预测模型的导出，Paddle 提供了 `paddle.jit.save` 接口，搭配 `@to_static` 可以导出预测模型。

使用样例如下：

```
import paddle
import paddle.nn as nn
from paddle.static import InputSpec
```

(下页继续)

(续上页)

```

IMAGE_SIZE = 784
CLASS_NUM = 10

class LinearNet(nn.Layer):
    def __init__(self):
        super(LinearNet, self).__init__()
        self._linear = nn.Linear(IMAGE_SIZE, CLASS_NUM)

    def forward(self, x):
        return self._linear(x)

layer = LinearNet()
layer = paddle.jit.to_static(layer, input_spec=[InputSpec(shape=[None, IMAGE_SIZE],  
dtype='float32')])

path = "example.model/linear"
paddle.jit.save(layer, path) # <---- Lazy mode, 此处才会触发 Program 的转换

```

更多用法可以参考： [【官网文档】jit.save](#)

报错调试

一、动转静报错日志

1.1 错误日志怎么看

如下是一个动转静报错实例代码：

```

import paddle
import numpy as np

@paddle.jit.to_static
def func(x):
    two = paddle.full(shape=[1], fill_value=2, dtype="int32")
    x = paddle.reshape(x, shape=[1, two])
    return x

def train():
    x = paddle.to_tensor(np.ones([3]).astype("int32"))
    func(x)

```

(下页继续)

(续上页)

```
if __name__ == '__main__':
    train()
```

执行后，报错日志如下图：

报错日志从上到下一共可以分为 4 个部分：

- **原生的 Python 报错栈**: 如 1 中的前两行所示，表示 /workspace/Paddle/run_dy2stat_error.py 文件第 145 行调用的函数 `train()` 导致的后续一系列报错。
- **动转静报错栈起始标志位**: `In transformed code`, 表示动转静报错信息栈，指运行转换后的代码时的报错信息。实际场景中，可以直接搜索 `In transformed code` 关键字，从这一行以下开始看报错日志即可。
- **用户代码报错栈**: 隐藏了框架层面的无用的报错信息，突用户代码报错栈。我们在出错代码下添加了波浪线和 `HERE` 指示词来提示具体的出错位置，并扩展了出错行代码上下文，帮助你快速定位出错位置。如上图 3 中所示，可以看出最后出错的用户代码为 `x = paddle.reshape(x, shape=[1, two])`。
- **框架层面报错信息**: 提供了静态图组网报错信息。一般可以直接根据最后三行的信息，定位具体是在生成哪个 OpDesc 时报的错误，一般是与执行此 Op 的 `inference` 逻辑报的错误。如上报错信息表明是 `reshape` Op 出错，出错原因是 tensor `x` 的 `shape` 为 [3]，将其 `reshape` 为 [1, 2] 是不被允许的。

NOTE: 在某些场景下，会识别报错类型并给出修改建议，如下图所示。`Revise suggestion` 下面是出错的排查建议，你可以根据建议对代码进行排查修改。

1.2 报错信息定制化展示

1.2.1 未经动转静报错模块处理的原生报错信息

若你想查看 Paddle 原生报错信息栈，即未被动转静模块处理过的报错信息栈，可以设置环境变量 `TRANSLATOR_DISABLE_NEW_ERROR=1` 关闭动转静报错模块。该环境变量默认值为 0，表示默认开启动转静报错模块。在 1.1 小节的代码中添加下面的代码即可以查看原生的报错信息：

```
import os
os.environ["TRANSLATOR_DISABLE_NEW_ERROR"] = '1'
```

可以得到如下的报错信息：

1.2.2 C++ 报错栈

默认会隐藏 C++ 报错栈，你可设置 C++ 端的环境变量 `FLAGS_call_stack_level=2` 来显示 C++ 报错栈信息。如可以在终端输入 `export FLAGS_call_stack_level=2` 来进行设置，之后可以看到 C++ 端的报错栈：

二、调试方法

在调试前请确保转换前的动态图代码能够成功运行，下面介绍动静中推荐的几种调试方法。

2.1 pdb 调试

`pdb` 是 Python 中的一个模块，该模块定义了一个交互式 Python 源代码调试器。它支持在源码行间设置断点和单步执行，列出源代码和变量，运行 Python 代码等。

2.1.1 调试步骤

- step1：在想要进行调试的代码前插入 `import pdb; pdb.set_trace()` 开启 `pdb` 调试。

```
import paddle
import numpy as np

@paddle.jit.to_static
def func(x):
    x = paddle.to_tensor(x)
    import pdb; pdb.set_trace()          # <----- 开启 pdb 调试
    two = paddle.full(shape=[1], fill_value=2, dtype="int32")
    x = paddle.reshape(x, shape=[1, two])
    return x

func(np.ones([3]).astype("int32"))
```

- step2：正常运行.py 文件，在终端会出现下面类似结果，在 (Pdb) 位置后输入相应的 `pdb` 命令进行调试。

```
> /tmp/tmpm0iw5b5d.py(9) func()
-> two = paddle.full(shape=[1], fill_value=2, dtype='int32')
(Pdb)
```

- step3：在 `pdb` 交互模式下输入 `l`、`p` 等命令可以查看动静后静态图相应的代码、变量，进而排查相关的问题。

```

> /tmp/tmpm0iw5b5d.py(9) func()
-> two = paddle.full(shape=[1], fill_value=2, dtype='int32')
(Pdb) l
 4     import numpy as np
 5     def func(x):
 6         x = paddle.assign(x)
 7         import pdb
 8         pdb.set_trace()
 9 ->     two = paddle.full(shape=[1], fill_value=2, dtype='int32')
10     x = paddle.reshape(x, shape=[1, two])
11     return x
[EOF]
(Pdb) p x
var assign_0.tmp_0 : LOD_TENSOR.shape(3,).dtype(int32).stop_gradient(False)
(Pdb)

```

2.1.2 常用命令

更多 pdb 使用方法可以查看 [pdb 的官方文档](#)

2.2 打印转换后的静态图代码

你可以打印转换后的静态图代码，有 2 种方法：

2.2.1 set_code_level() 或 TRANSLATOR_CODE_LEVEL

通过调用 `set_code_level()` 或设置环境变量 `TRANSLATOR_CODE_LEVEL`，可以在日志中查看转换后的代码：

```

import paddle
import numpy as np

@paddle.jit.to_static
def func(x):
    x = paddle.to_tensor(x)
    if x > 3:
        x = x - 1
    return x

paddle.jit.set_code_level() # 也可设置 os.environ["TRANSLATOR_CODE_LEVEL"] = '100
                           # 效果相同
func(np.ones([1]))

```

此外，如果你想将转化后的代码也输出到 `sys.stdout`，可以设置参数 `also_to_stdout` 为 `True`，否则将仅输出到 `sys.stderr`。`set_code_level` 函数可以设置查看不同的 AST Transformer 转化后的代码，详情请见 `set_code_level`。

2.2.2 被装饰后的函数的 code 属性

如下代码中，装饰器 `@to_static` 会将函数 `func` 转化为一个类对象 `StaticFunction`，可以使用 `StaticFunction` 的 `code` 属性来获得转化后的代码。

```
import paddle
import numpy as np

@paddle.jit.to_static
def func(x):
    x = paddle.to_tensor(x)
    if x > 3:
        x = x - 1
    return x

func(np.ones([1]))
print(func.code)
```

运行后可以看到动转静后的静态图代码：

```
def func(x):
    x = paddle.assign(x)

    def true_fn_0(x):
        x = x - 1
        return x

    def false_fn_0(x):
        return x
    x = paddle.jit.dy2static.convert_ifelse(x > 3, true_fn_0, false_fn_0, (
        x,), (x,), (x,))
    return x
```

2.3 使用 print 查看变量

`print` 函数可以用来查看变量，该函数在动转静中会被转化。当仅打印 Paddle Tensor 时，实际运行时会被转换为 Paddle 算子 Print，否则仍然运行 `print`。

```
import paddle
import numpy as np

@paddle.jit.to_static
def func(x):
    x = paddle.to_tensor(x)

    # 打印 x, x是Paddle Tensor, 实际运行时会运行Paddle Print(x)
    print(x)
    # 打印注释, 非Paddle Tensor, 实际运行时仍运行print
    print("Here call print function.")

    if len(x) > 3:
        x = x - 1
    else:
        x = paddle.ones(shape=[1])
    return x

func(np.ones([1]))
```

运行后可以看到 x 的值：

```
Variable: assign_0.tmp_0
- lod: {}
- place: CUDAPlace(0)
- shape: [1]
- layout: NCHW
- dtype: double
- data: [1]
```

2.4 日志打印

动转静在日志中记录了额外的调试信息，以帮助你了解动转静过程中函数是否被成功转换。你可以调用 `paddle.jit.set_verbosity(level=0, also_to_stdout=False)` 或设置环境变量 `TRANSLATOR_VERTBOSITY=level` 来设置日志详细等级，并查看不同等级的日志信息。目前，`level` 可以取值 0-3：

- 0: 无日志
- 1: 包括了动转静转化流程的信息，如转换前的源码、转换的可调用对象

- 2: 包括以上信息，还包括更详细函数转化日志
- 3: 包括以上信息，以及更详细的动转静日志

注意： 日志中包括了源代码等信息，请在共享日志前确保它不包含敏感信息。打印日志的示例代码：

```
import paddle
import numpy as np
import os

@paddle.jit.to_static
def func(x):
    x = paddle.to_tensor(x)
    if len(x) > 3:
        x = x - 1
    else:
        x = paddle.ones(shape=[1])
    return x

paddle.jit.set_verbosity(3)
# 或者设置os.environ["TRANSLATOR_VERBOSITY"] = '3'
func(np.ones([1]))
```

运行结果：

```
Sun Sep 26 08:50:20 Dynamic-to-Static INFO: (Level 1) Source code:
@paddle.jit.to_static
def func(x):
    x = paddle.to_tensor(x)
    if len(x) > 3:
        x = x - 1
    else:
        x = paddle.ones(shape=[1])
    return x

Sun Sep 26 08:50:20 Dynamic-to-Static INFO: (Level 1) Convert callable object:_
↪convert <built-in function len>.
```

此外，如果你想将日志也输出到 sys.stdout，可以设置参数 also_to_stdout 为 True，否则将仅输出到 sys.stderr，详情请见 set_verbosity。

三、快速确定问题原因

经过对报错信息的种类进行汇总整理，可以将动转静的问题大致分为如下几个类别：

3.1 (NotFound) Input("X")

报错信息大致如下：

```
RuntimeError: (NotFound) Input("Filter") of ConvOp should not be null.
[Hint: Expected ctx->HasInputs("Filter") == true, but received ctx->HasInputs(
→"Filter"):0 != true:1.]
[operator < conv2d > error]
```

此类问题的原因一般是：

执行到报错所在行的 Paddle API 时，某些输入或者 weight 的类型还是动态图的 Tensor，而非静态图的 Variable 或者 Parameter.

排查建议：

- 首先确认代码所在的 sublayer 是否继承了 nn.Layer
- 此行代码所在函数是否绕开了 forward 函数，单独调用的（2.1 版本之前）
- 如何查看是 Tensor 还是 Variable 类型，可以通过 pdb 交互式调试

3.2 Expected input_dims[i] == input_dims[0]

报错信息大致如下：

```
[Hint: Expected input_dims[i] == input_dims[0], but received input_dims[i]:-1, -1 !=_
→input_dims[0]:16, -1.]
[operator < xxx_op > error]
```

此类问题的原因一般是：

逐个 append_op 生成静态图 Program 时，在执行到某个 Paddle API 时，编译期 infershape 不符合要求。

排查建议：

- 代码层面，判断是否是上游使用了 reshape 导致 -1 的污染性传播
动态图由于执行时 shape 都是已知的，所以 reshape(x, [-1, 0, 128]) 是没有问题的。但静态图组网时都是编译期的 shape（可能为-1），因此使用 reshape 接口时，尽量减少 -1 的使用。
- 可以结合调试技巧，判断是否是某个 API 的输出 shape 在动静态图下有 diff 行为

比如某些 Paddle API 动态图下返回的是 1-D Tensor，但静态图却是始终和输入保持一致，如 ctx->SetOutputDim("Out", ctx->GetInputDim("X"));

3.3 desc->CheckGuards() == true

报错信息大致如下：

```
[Hint: Expected desc->CheckGuards() == true, but received desc->CheckGuards(): 0 !=  
true: 1.]
```

此类问题的原因一般是：

执行到报错所在行的 Paddle API 时，某些输入或者 weight 的类型还是动态图的 Tensor，而非静态图的 Variable 或者 Parameter.

如下是当前动、静态图对 slice 语法功能的汇总情况：

排查建议：

- 模型代码是否存在上述复杂的 Tensor slice 切片操作
- 推荐使用 paddle.slice 接口替换复杂的 Tensor slice 操作

3.4 Segment Fault

当动转静出现段错误时，报错栈信息也会很少，但导致此类问题的原因一般也比较明确。此类问题的一般原因是：

某个 sublayer 未继承 nn.Layer，同时在 __init__.py 函数中存在 paddle.to_tensor 接口的调用。导致在生成 Program 或者保存模型参数时，在静态图模式下访问了动态图的 Tensor 数据。

排查建议：

- 每个 sublayer 是否继承了 nn.Layer

3.5 Container 的使用建议

动态图下，提供了如下几种 container 的容器类：

- ParameterList

```
class MyLayer(paddle.nn.Layer):  
    def __init__(self, num_stacked_param):  
        super(MyLayer, self).__init__()  
  
        w1 = paddle.create_parameter(shape=[2, 2], dtype='float32')  
        w2 = paddle.create_parameter(shape=[2], dtype='float32')
```

(下页继续)

(续上页)

```
# 此用法下, MyLayer.parameters() 返回为空
self.params = [w1, w2]                                     # <----- 错误用法

self.params = paddle.nn.ParameterList([w1, w2])      # <----- 正确用法
```

- LayerList

```
class MyLayer(paddle.nn.Layer):
    def __init__(self):
        super(MyLayer, self).__init__()

        layer1 = paddle.nn.Linear(10, 10)
        layer2 = paddle.nn.Linear(10, 16)

        # 此用法下, MyLayer.parameters() 返回为空
        self.linears = [layer1, layer2]                         # <----- 错误用法

        self.linears = paddle.nn.LayerList([layer1, layer2])    # <----- 正确用法
```

2.4 推理部署

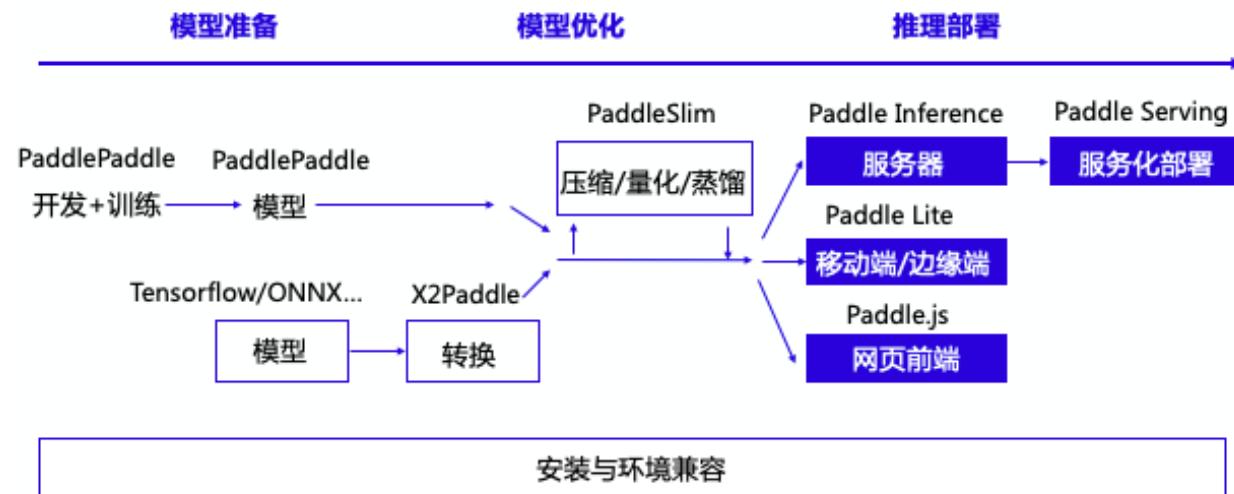
2.4.1 飞桨推理产品简介

作为飞桨生态重要的一部分，飞桨提供了多个推理产品，完整承接深度学习模型应用的最后一公里。

整体上分，推理产品主要包括如下子产品

名称	英文表示	适用场景
飞桨原生推理库	Paddle Inference	高性能服务器端、云端推理
飞桨服务化推理框架	Paddle Serving	自动服务、模型管理等高阶功能
飞桨轻量化推理引擎	Paddle Lite	移动端、物联网等
飞桨前端推理引擎	Paddle.js	浏览器中推理、小程序等

各产品在推理生态中的关系如下



用户使用飞桨推理产品的工作流如下

1. 获取一个飞桨的推理模型，其中有两种方法
 1. 利用飞桨训练得到一个推理模型
 2. 用 X2Paddle 工具从第三方框架（比如 TensorFlow 或者 Caffe 等）产出的模型转化
2. （可选）对模型进行进一步优化，PaddleSlim 工具可以对模型进行压缩，量化，裁剪等工作，显著提升模型执行的速度性能，降低资源消耗
3. 将模型部署到具体的推理产品上

服务器部署—Paddle Inference

Paddle Inference 是飞桨的原生推理库，作用于服务器端和云端，提供高性能的推理能力。

由于能力直接基于飞桨的训练算子，因此 Paddle Inference 可以通用支持飞桨训练出的所有模型。

Paddle Inference 功能特性丰富，性能优异，针对不同平台不同的应用场景进行了深度的适配优化，做到高吞吐、低时延，保证了飞桨模型在服务器端即训即用，快速部署。

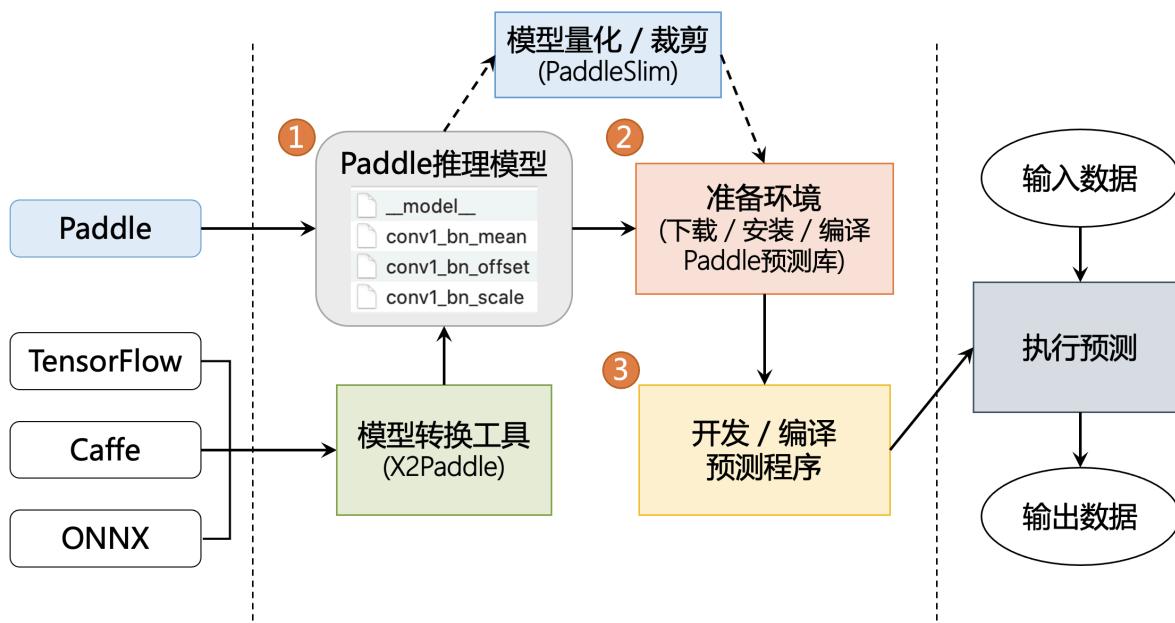
一些常见的文档链接如下：

- 完整使用文档位于：[Paddle Inference 文档](#)
- 代码示例位于[inference demo](#)
- 点此 [安装与编译 Linux 预测库](#)
- 点此 [安装与编译 Windows 预测库](#)

与主框架 model.predict 区别

飞桨推理产品 paddle inference 和主框架的 Model.predict 均可实现推理预测，Paddle Inference 是飞桨的原生推理库，作用于服务器端和云端，提供高性能的推理能力，主框架的 Model 对象是一个具备训练、测试、推理的神经网络。相比于 Model.predict, inference 可使用 MKLDNN、CUDNN、TensorRT 进行预测加速，同时支持用 X2Paddle 工具从第三方框架（TensorFlow、Pytorch、Caffe 等）产出的模型，可联动 PaddleSlim，支持加载量化、裁剪和蒸馏后的模型部署。Model.predict 适用于训练好的模型直接进行预测，paddle inference 适用于对推理性能、通用性有要求的用户，针对不同平台不同的应用场景进行了深度的适配优化，保证模型在服务器端即训即用，快速部署。

预测流程图



高性能实现

内存/显存复用提升服务吞吐量

在推理初始化阶段，对模型中的 OP 输出 Tensor 进行依赖分析，将两两互不依赖的 Tensor 在内存/显存空间上进行复用，进而增大计算并行量，提升服务吞吐量。

细粒度 OP 横向纵向融合减少计算量

在推理初始化阶段，按照已有的融合模式将模型中的多个 OP 融合成一个 OP，减少了模型的计算量的同时，也减少了 Kernel Launch 的次数，从而能提升推理性能。目前 Paddle Inference 支持的融合模式多达几十个。

内置高性能的 CPU/GPU Kernel

内置同 Intel、Nvidia 共同打造的高性能 kernel，保证了模型推理高性能的执行。

多功能集成

集成 TensorRT 加快 GPU 推理速度

Paddle Inference 采用子图的形式集成 TensorRT，针对 GPU 推理场景，TensorRT 可对一些子图进行优化，包括 OP 的横向和纵向融合，过滤冗余的 OP，并为 OP 自动选择最优的 kernel，加快推理速度。

集成 oneDNN CPU 推理加速引擎

一行代码开始 oneDNN 加速，快捷高效。

支持 PaddleSlim 量化压缩模型的部署

PaddleSlim 是飞桨深度学习模型压缩工具，Paddle Inference 可联动 PaddleSlim，支持加载量化、裁剪和蒸馏后的模型并部署，由此减小模型存储空间、减少计算占用内存、加快模型推理速度。其中在模型量化方面，Paddle Inference 在 X86 CPU 上做了深度优化，常见分类模型的单线程性能可提升近 3 倍，ERNIE 模型的单线程性能可提升 2.68 倍。

支持 X2Paddle 转换得到的模型

除支持飞桨训练的模型外，也支持用 X2Paddle 工具从第三方框架（比如 TensorFlow、Pytorch 或者 Caffe 等）产出的模型。

多场景适配

主流软硬件环境兼容适配

支持服务器端 X86 CPU、NVIDIA GPU 芯片，兼容 Linux/Mac/Windows 系统，同时对飞腾、鲲鹏、曙光、昆仑等国产 CPU/NPU 进行适配。。支持所有飞桨训练产出的模型，完全做到即训即用。

主流、国产操作系统全适配

适配主流操作系统 Linux、Windows、MacOS，同时适配麒麟 OS、统信 OS、普华 OS、中科方德等国产操作系统

多语言接口支持

支持 C++、Python、C、Go、Java 和 R 语言 API，对于其他语言，提供了 ABI 稳定的 C API，提供配套的教程、API 文档及示例。

交流与反馈

- 欢迎您通过 Github Issues 来提交问题、报告与建议
- 微信公众号：飞桨 PaddlePaddle
- 微信群：部署交流群

移动端/嵌入式部署—Paddle Lite

Paddle-Lite 为 Paddle-Mobile 的升级版，定位支持包括手机移动端在内更多场景的轻量化高效预测，支持更广泛的硬件和平台，是一个高性能、轻量级的深度学习预测引擎。在保持和 PaddlePaddle 无缝对接外，也兼容支持其他训练框架产出的模型。

完整使用文档位于 [Paddle-Lite 文档](#)。

特性

轻量级

执行阶段和计算优化阶段实现良好解耦拆分，移动端可以直接部署执行阶段，无任何第三方依赖。包含完整的 80 个 Op+85 个 Kernel 的动态库，对于 ARMV7 只有 800K，ARMV8 下为 1.3M，并可以裁剪到更低。在应用部署时，载入模型即可直接预测，无需额外分析优化。

高性能

极致的 ARM CPU 性能优化，针对不同微架构特点实现 kernel 的定制，最大发挥计算性能，在主流模型上展现出领先的速度优势。支持量化模型，结合 [PaddleSlim 模型压缩工具](#) 中量化功能，可以提供高精度高性能的预测能力。在 Huawei NPU，FPGA 上也具有非常好的性能表现。

最新性能数据位于 [Benchmark 文档](#)。

通用性

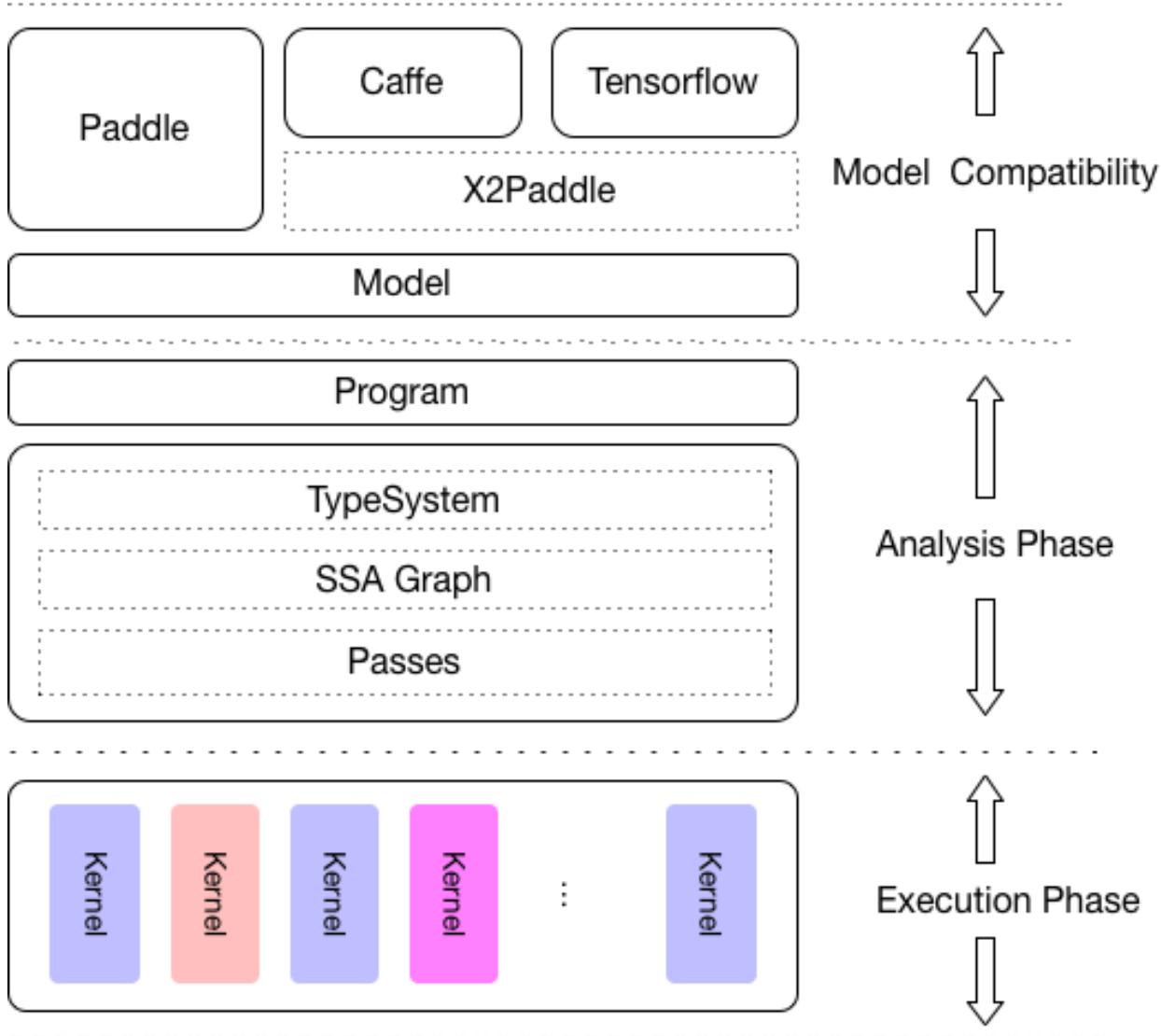
硬件方面，Paddle-Lite 的架构设计为多硬件兼容支持做了良好设计。除了支持 ARM CPU、Mali GPU、Adreno GPU，还特别支持了华为 NPU，以及 FPGA 等边缘设备广泛使用的硬件。即将支持包括寒武纪、比特大陆等 AI 芯片，未来会增加对更多硬件的支持。

模型支持方面，Paddle-Lite 和 PaddlePaddle 训练框架的 Op 对齐，提供更广泛的模型支持能力。目前已严格验证 18 个模型 85 个 OP 的精度和性能，对视觉类模型做到了较为充分的支持，覆盖分类、检测和定位，包含了特色的 OCR 模型的支持。未来会持续增加更多模型的支持验证。

框架兼容方面：除了 PaddlePaddle 外，对其他训练框架也提供兼容支持。当前，支持 Caffe 和 TensorFlow 训练出来的模型，通过 [X2Paddle] (<https://github.com/PaddlePaddle/X2Paddle>) 转换工具实现。接下来将会对 ONNX 等格式模型提供兼容支持。

架构

Paddle-Lite 的架构设计着重考虑了对多硬件和平台的支持，并且强化了多个硬件在一个模型中混合执行的能力，多个层面的性能优化处理，以及对端侧应用的轻量化设计。



其中，Analysis Phase 包括了 MIR(Machine IR) 相关模块，能够对原有的模型的计算图针对具体的硬件列表进行算子融合、计算裁剪在内的多种优化。Execution Phase 只涉及到 Kernel 的执行，且可以单独部署，以支持极致的轻量级部署。

Paddle-Mobile 升级为 Paddle-Lite 的说明

原 Paddle-Mobile 作为一个致力于嵌入式平台的 PaddlePaddle 预测引擎，已支持多种硬件平台，包括 ARM CPU、Mali GPU、Adreno GPU，以及支持苹果设备的 GPU Metal 实现、ZU5、ZU9 等 FPGA 开发板、树莓派等 arm-linux 开发板。在百度内已经过广泛业务场景应用验证。对应设计文档可参考: [mobile/README](#)

Paddle-Mobile 整体升级重构并更名为 Paddle-Lite 后，原 paddle-mobile 的底层能力大部分已集成到新架构下。作为过渡，暂时保留原 Paddle-mobile 代码。主体代码位于 mobile/ 目录中，后续一段时间会继续维护，并完成全部迁移。新功能会统一到新架构下开发。

metal, web 的模块相对独立，会继续在 ./metal 和 ./web 目录下开发和维护。对苹果设备的 GPU Metal 实

现的需求及 web 前端预测需求，可以直接进入这两个目录。

致谢

Paddle-Lite 借鉴了以下开源项目：

- ARM compute library
- Anakin , Anakin 对应底层的一些优化实现已被集成到 Paddle-Lite。Anakin 作为 PaddlePaddle 组织下的一个高性能预测项目，极具前瞻性，对 Paddle-Lite 有重要贡献。Anakin 已和本项目实现整合。之后，Anakin 不再升级。

交流与反馈

- 欢迎您通过 Github Issues 来提交问题、报告与建议
- 微信公众号：飞桨 PaddlePaddle
- QQ 群: 696965088
- 论坛: 欢迎大家在PaddlePaddle 论坛分享在使用 PaddlePaddle 中遇到的问题和经验，营造良好的论坛氛围

模型压缩—PaddleSlim

PaddleSlim 是一个模型压缩工具库，包含模型剪裁、定点量化、知识蒸馏、超参搜索和模型结构搜索等一系列模型压缩策略。

对于业务用户，PaddleSlim 提供完整的模型压缩解决方案，可用于图像分类、检测、分割等各种类型的视觉场景。同时也在持续探索 NLP 领域模型的压缩方案。另外，PaddleSlim 提供且在不断完善各种压缩策略在经典开源任务的 benchmark，以便业务用户参考。

对于模型压缩算法研究者或开发者，PaddleSlim 提供各种压缩策略的底层辅助接口，方便用户复现、调研和使用最新论文方法。PaddleSlim 会从底层能力、技术咨询合作和业务场景等角度支持开发者进行模型压缩策略相关的创新工作。

功能

- 模型剪裁
 - 卷积通道均匀剪裁
 - 基于敏感度的卷积通道剪裁
 - 基于进化算法的自动剪裁
- 定点量化
 - 在线量化训练 (training aware)

- 离线量化 (post training)
- 知识蒸馏
 - 支持单进程知识蒸馏
 - 支持多进程分布式知识蒸馏
- 神经网络结构自动搜索 (NAS)
 - 支持基于进化算法的轻量神经网络结构自动搜索
 - 支持 One-Shot 网络结构自动搜索
 - 支持 FLOPS / 硬件延时约束
 - 支持多平台模型延时评估
 - 支持用户自定义搜索算法和搜索空间

安装

依赖:

Paddle >= 1.7.0

```
pip install paddleslim -i https://pypi.org/simple
```

使用

- 快速开始: 通过简单示例介绍如何快速使用 PaddleSlim。
- 进阶教程: PaddleSlim 高阶教程。
- 模型库: 各个压缩策略在图像分类、目标检测和图像语义分割模型上的实验结论, 包括模型精度、预测速度和可供下载的预训练模型。
- API 文档
- Paddle 检测库: 介绍如何在检测库中使用 PaddleSlim。
- Paddle 分割库: 介绍如何在分割库中使用 PaddleSlim。
- PaddleLite: 介绍如何使用预测库 PaddleLite 部署 PaddleSlim 产出的模型。

部分压缩策略效果

分类模型

数据: ImageNet2012; 模型: MobileNetV1;

图像检测模型

数据: Pascal VOC; 模型: MobileNet-V1-YOLOv3

数据: COCO; 模型: MobileNet-V1-YOLOv3

搜索

数据: ImageNet2012; 模型: MobileNetV2

2.5 分布式训练

您可以通过以下内容, 了解飞桨分布式训练的特性和使用指南:

- 分布式训练快速开始: 使用飞桨框架快速开始分布式训练。
- 使用 FleetAPI 进行分布式训练: 使用飞桨框架 FleetAPI 完成分布式训练。

2.5.1 分布式训练快速开始

FleetX 是飞桨分布式训练扩展包, 为了可以让用户更快速了解和使用飞桨分布式训练特性, 提供了大量分布式训练例子, 可以查阅 <https://github.com/PaddlePaddle/FleetX/tree/develop/examples>, 以下章节的例子都可以在这找到, 用户也可以直接将仓库下载到本地直接。

一、Collective 训练快速开始

Title underline too short.

一、Collective 训练快速开始

本节将采用 CV 领域非常经典的模型 ResNet50 为例, 介绍如何使用 Fleet API (`paddle.distributed.fleet`) 完成 Collective 训练任务。数据方面我们采用 Paddle 内置的 flowers 数据集, 优化器使用 Momentum 方法。循环迭代多个 epoch, 每轮打印当前网络具体的损失值和 acc 值。具体代码保存在 FleetX/examples/resnet 下面, 其中包含动态图和静态图两种执行方式。`resnet_dygraph.py` 为动态图模型相关代码, `train_fleet_dygraph.py` 为动态图训练脚本。`resnet_static.py` 为静态图模型相关代码, 而 `train_fleet_static.py` 为静态图训练脚本。

1.1 版本要求

在编写分布式训练程序之前，用户需要确保已经安装 paddlepaddle-2.0.0-rc-cpu 或 paddlepaddle-2.0.0-rc-gpu 及以上版本的飞桨开源框架。

1.2 操作方法

与单机单卡的普通模型训练相比，无论静态图还是动态图，Collective 训练的代码都只需要补充三个部分代码：

1. 导入分布式训练需要的依赖包。
2. 初始化 Fleet 环境。
3. 设置分布式训练需要的优化器。

下面将逐一进行讲解。

1.2.1 导入依赖

Title underline too short.

```
1.2.1 导入依赖  
*****
```

导入必要的依赖，例如分布式训练专用的 Fleet API(paddle.distributed.fleet)。

```
from paddle.distributed import fleet
```

1.2.2 初始化 fleet 环境

Title underline too short.

```
1.2.2 初始化fleet环境  
*****
```

包括定义缺省的分布式策略，然后通过将参数 is_collective 设置为 True，使训练架构设定为 Collective 架构。

```
strategy = fleet.DistributedStrategy()  
fleet.init(is_collective=True, strategy=strategy)
```

1.2.3 设置分布式训练使用的优化器

Title underline too short.

```
1.2.3 设置分布式训练使用的优化器  
*****
```

使用 `distributed_optimizer` 设置分布式训练优化器。

```
optimizer = fleet.distributed_optimizer(optimizer)
```

1.3 动态图完整代码

`train_fleet_dygraph.py` 的完整训练代码如下所示。

```
# -*- coding: UTF-8 -*-
import numpy as np
import argparse
import ast
import paddle
# 导入必要分布式训练的依赖包
from paddle.distributed import fleet
# 导入模型文件
from resnet_dygraph import ResNet

base_lr = 0.1    # 学习率
momentum_rate = 0.9 # 冲量
l2_decay = 1e-4 # 权重衰减

epoch = 10 #训练迭代次数
batch_size = 32 #训练批次大小
class_dim = 102

# 设置数据读取器
def reader_decorator(reader):
    def __reader__():
        for item in reader():
            img = np.array(item[0]).astype('float32').reshape(3, 224, 224)
            label = np.array(item[1]).astype('int64').reshape(1)
            yield img, label

    return __reader__

# 设置优化器
```

(下页继续)

(续上页)

```
def optimizer_setting(parameter_list=None):
    optimizer = paddle.optimizer.Momentum(
        learning_rate=base_lr,
        momentum=momentum_rate,
        weight_decay=paddle.regularizer.L2Decay(l2_decay),
        parameters=parameter_list)
    return optimizer

# 设置训练函数
def train_resnet():
    # 初始化Fleet环境
    fleet.init(is_collective=True)

    resnet = ResNet(class_dim=class_dim, layers=50)

    optimizer = optimizer_setting(parameter_list=resnet.parameters())
    optimizer = fleet.distributed_optimizer(optimizer)
    # 通过Fleet API获取分布式model，用于支持分布式训练
    resnet = fleet.distributed_model(resnet)

    train_reader = paddle.batch(
        reader_decorator(paddle.dataset.flowers.train(use_xmap=True)),
        batch_size=batch_size,
        drop_last=True)

    train_loader = paddle.io.DataLoader.from_generator(
        capacity=32,
        use_double_buffer=True,
        iterable=True,
        return_list=True,
        use_multiprocess=True)
    train_loader.set_sample_list_generator(train_reader)

    for eop in range(epoch):
        resnet.train()

        for batch_id, data in enumerate(train_loader()):
            img, label = data
            label.stop_gradient = True

            out = resnet(img)
            loss = paddle.nn.functional.cross_entropy(input=out, label=label)
            avg_loss = paddle.mean(x=loss)
```

(下页继续)

(续上页)

```

acc_top1 = paddle.metric.accuracy(input=out, label=label, k=1)
acc_top5 = paddle.metric.accuracy(input=out, label=label, k=5)

dy_out = avg_loss.numpy()

avg_loss.backward()

optimizer.minimize(avg_loss)
resnet.clear_gradients()
if batch_id % 5 == 0:
    print("[Epoch %d, batch %d] loss: %.5f, acc1: %.5f, acc5: %.5f" %
        (eop, batch_id, dy_out, acc_top1, acc_top5))
# 启动训练
if __name__ == '__main__':
    train_resnet()

```

1.4 静态图完整代码

`train_fleet_static.py` 的完整训练代码如下所示。

```

# -*- coding: UTF-8 -*-
import numpy as np
import argparse
import ast
import paddle
# 导入必要分布式训练的依赖包
import paddle.distributed.fleet as fleet
# 导入模型文件
import resnet_static as resnet
import os

base_lr = 0.1    # 学习率
momentum_rate = 0.9 # 冲量
l2_decay = 1e-4 # 权重衰减

epoch = 10    # 训练迭代次数
batch_size = 32 # 训练批次大小
class_dim = 10

# 设置优化器
def optimizer_setting(parameter_list=None):
    optimizer = paddle.optimizer.Momentum(

```

(下页继续)

(续上页)

```

        learning_rate=base_lr,
        momentum=momentum_rate,
        weight_decay=paddle.regularizer.L2Decay(12_decay),
        parameters=parameter_list)

    return optimizer
# 设置数据读取器
def get_train_loader(feed_list, place):
    def reader_decorator(reader):
        def __reader__():
            for item in reader():
                img = np.array(item[0]).astype('float32').reshape(3, 224, 224)
                label = np.array(item[1]).astype('int64').reshape(1)
                yield img, label

        return __reader__
    train_reader = paddle.batch(
        reader_decorator(paddle.dataset.flowers.train(use_xmap=True)),
        batch_size=batch_size,
        drop_last=True)
    train_loader = paddle.io.DataLoader.from_generator(
        capacity=32,
        use_double_buffer=True,
        feed_list=feed_list,
        iterable=True)
    train_loader.set_sample_list_generator(train_reader, place)
    return train_loader
# 设置训练函数
def train_resnet():
    print("Start collective training example:")
    paddle.enable_static() # 使能静态图功能
    paddle.vision.set_image_backend('cv2')

    image = paddle.static.data(name="x", shape=[None, 3, 224, 224], dtype='float32')
    label= paddle.static.data(name="y", shape=[None, 1], dtype='int64')
    # 调用ResNet50模型
    model = resnet.ResNet(layers=50)
    out = model.net(input=image, class_dim=class_dim)
    avg_cost = paddle.nn.functional.cross_entropy(input=out, label=label)
    acc_top1 = paddle.metric.accuracy(input=out, label=label, k=1)
    acc_top5 = paddle.metric.accuracy(input=out, label=label, k=5)
    # 设置训练资源，本例使用GPU资源
    place = paddle.CUDAPlace(int(os.environ.get('FLAGS_selected_gpus', 0)))
    print("Run on {}".format(place))

```

(下页继续)

(续上页)

```

train_loader = get_train_loader([image, label], place)
# 初始化Fleet环境
strategy = fleet.DistributedStrategy()
fleet.init(is_collective=True, strategy=strategy)
optimizer = optimizer_setting()

# 通过Fleet API获取分布式优化器，将参数传入飞桨的基础优化器
optimizer = fleet.distributed_optimizer(optimizer)
optimizer.minimize(avg_cost)

exe = paddle.static.Executor(place)
print("Execute startup program.")
exe.run(paddle.static.default_startup_program())

epoch = 10
step = 0
for eop in range(epoch):
    for batch_id, data in enumerate(train_loader()):
        loss, acc1, acc5 = exe.run(paddle.static.default_main_program(), 
        feed=data, fetch_list=[avg_cost.name, acc_top1.name, acc_top5.name])
        if batch_id % 5 == 0:
            print("[Epoch %d, batch %d] loss: %.5f, acc1: %.5f, acc5: %.5f" % 
            (eop, batch_id, loss, acc1, acc5))
# 启动训练
if __name__ == '__main__':
    train_resnet()

```

1.5 运行示例

假设要运行 2 卡的任务，那么只需在命令行中执行：

动态图：

```
python3 -m paddle.distributed.launch --gpus=0,1 train_fleet_dygraph.py
```

您将看到显示如下日志信息：

```
----- Configuration Arguments -----
gpus: 0,1
heter_worker_num: None
heter_workers:
http_port: None
```

(下页继续)

(续上页)

```

ips: 127.0.0.1
log_dir: log
nproc_per_node: None
server_num: None
servers:
training_script: train_fleet_dygraph.py
training_script_args: []
worker_num: None
workers:
-----
WARNING 2021-05-06 11:32:50,804 launch.py:316] Not found distinct arguments and_
↪compiled with cuda. Default use collective mode
launch train in GPU mode
INFO 2021-05-06 11:32:50,806 launch_utils.py:472] Local start 2 processes. First_
↪process distributed environment info (Only For Debug):
↪
↪+=====+
|           Distributed Envs          Value
↪|           |
↪+-----+
↪|           PADDLE_TRAINER_ENDPOINTS    127.0.0.1:20923,127.0.0.1:10037
↪|           |
↪|           FLAGS_selected_gpus       0
↪|           |
↪|           PADDLE_TRAINER_ID         0
↪|           |
↪|           PADDLE_TRAINERS_NUM       2
↪|           |
↪|           PADDLE_CURRENT_ENDPOINT   127.0.0.1:20923
↪|
↪+=====+
INFO 2021-05-06 11:32:50,806 launch_utils.py:475] details abouts PADDLE_TRAINER_
↪ENDPOINTS can be found in log/endpoints.log, and detail running logs maybe found in_
↪log/workerlog.0
grep: warning: GREP_OPTIONS is deprecated; please use an alias or script
I0506 11:32:51.828132 6427 nccl_context.cc:189] init nccl context nranks: 2 local_
↪rank: 0 gpu id: 0 ring id: 0
W0506 11:32:52.365190 6427 device_context.cc:362] Please NOTE: device: 0, GPU_
↪Compute Capability: 7.0, Driver API Version: 11.0, Runtime API Version: 11.0
W0506 11:32:52.368203 6427 device_context.cc:372] device: 0, cuDNN Version: 8.0.

```

(下页继续)

(续上页)

```
[Epoch 0, batch 0] loss: 4.98047, acc1: 0.00000, acc5: 0.00000
[Epoch 0, batch 5] loss: 39.06348, acc1: 0.03125, acc5: 0.09375
...
...
```

静态图：

```
python3 -m paddle.distributed.launch --gpus=0,1 train_fleet_static.py
```

您将看到显示如下日志信息：

```
----- Configuration Arguments -----
gpus: 0,1
heter_worker_num: None
heter_workers:
http_port: None
ips: 127.0.0.1
log_dir: log
nproc_per_node: None
server_num: None
servers:
training_script: train_fleet_static.py
training_script_args: []
worker_num: None
workers:
-----
WARNING 2021-05-06 11:36:30,019 launch.py:316] Not found distinct arguments and
↳ compiled with cuda. Default use collective mode
launch train in GPU mode
INFO 2021-05-06 11:36:30,021 launch_utils.py:472] Local start 2 processes. First
↳ process distributed environment info (Only For Debug):
-
↳ +-----+-----+-----+
| | Distributed Envs | Value |
| |-----+-----+-----|
| | PADDLE_TRAINER_ID | 0 |
| |-----+-----+-----|
| | PADDLE_CURRENT_ENDPOINT | 127.0.0.1:10039 |
| |-----+-----+-----|
| | PADDLE_TRAINER_ENDPOINTS | 127.0.0.1:10039,127.0.0.1:31719 |
| |-----+-----+-----|
| | PADDLE_TRAINERS_NUM | 2 |
| |-----+-----+-----|

```

(下页继续)

(续上页)

```

|          FLAGS_selected_gpus          0
|_
+-+-----+
INFO 2021-05-06 11:36:30,021 launch_utils.py:475] details abouts PADDLE_TRAINER_
ENDPOINTS can be found in log/endpoints.log, and detail running logs maybe found in_
log/workerlog.0
grep: warning: GREP_OPTIONS is deprecated; please use an alias or script
Start collective training example:
Run on CUDAPlace(0).
server not ready, wait 3 sec to retry...
not ready endpoints:['127.0.0.1:31719']
Execute startup program.
W0506 11:36:35.667778 6697 device_context.cc:362] Please NOTE: device: 0, GPU_
Compute Capability: 7.0, Driver API Version: 11.0, Runtime API Version: 11.0
W0506 11:36:35.671609 6697 device_context.cc:372] device: 0, cuDNN Version: 8.0.
Start training:
W0506 11:36:39.900507 6697 fuse_all_reduce_op_pass.cc:79] Find all_reduce operators:_  

161. To make the speed faster, some all_reduce ops are fused during training, after_
fusion, the number of all_reduce ops is 5.
[Epoch 0, batch 0] loss: 4.67622, acc1: 0.00000, acc5: 0.09375
[Epoch 0, batch 5] loss: 30.24010, acc1: 0.00000, acc5: 0.06250
...

```

从单机多卡到多机多卡训练，在代码上不需要做任何改动，只需再额外指定 ips 参数即可。其内容为多机的 ip 列表，命令如下所示：

```

# 动态图
python3 -m paddle.distributed.launch --ips="xx.xx.xx.xx,yy.yy.yy.yy" --gpus 0,1,2,3,4,
5,6,7 train_fleet_dygraph.py

# 静态图
python3 -m paddle.distributed.launch --ips="xx.xx.xx.xx,yy.yy.yy.yy" --gpus 0,1,2,3,4,
5,6,7 train_fleet_static.py

```

二、ParameterServer 训练快速开始

Title underline too short.

二、ParameterServer 训练快速开始

本节将采用推荐领域非常经典的模型 wide_and_deep 为例，介绍如何使用 Fleet API(paddle.distributed.fleet)完成参数服务器训练任务，本次快速开始的完整示例代码位于 https://github.com/PaddlePaddle/FleetX/tree/develop/examples/wide_and_deep。

2.1 版本要求

在编写分布式训练程序之前，用户需要确保已经安装 paddlepaddle-2.0.0-rc-cpu 或 paddlepaddle-2.0.0-rc-gpu 及以上版本的飞桨开源框架。

2.2 操作方法

参数服务器训练的基本代码主要包括以下几个部分：

1. 导入分布式训练需要的依赖包。
2. 定义分布式模式并初始化分布式训练环境。
3. 加载模型及数据。
4. 定义参数更新策略及优化器。
5. 开始训练。

下面将逐一进行讲解。

2.2.1 导入依赖

Title underline too short.

2.2.1 导入依赖

导入必要的依赖，例如分布式训练专用的 Fleet API(paddle.distributed.fleet)。

```
import paddle
import paddle.distributed.fleet as fleet
```

2.2.2 定义分布式模式并初始化分布式训练环境

Title underline too short.

```
2.2.2 定义分布式模式并初始化分布式训练环境
#####
#
```

通过 fleet.init() 接口，用户可以定义训练相关的环境，注意此环境是用户预先在环境变量中配置好的，包括：训练节点个数，服务节点个数，当前节点的序号，服务节点完整的 IP:PORT 列表等。

```
# 当前参数服务器模式只支持静态图模式，因此训练前必须指定 ``paddle.enable_static()``  
paddle.enable_static()  
fleet.init(is_collective=False)
```

2.2.3 加载模型及数据

Title underline too short.

```
2.2.3 加载模型及数据
#####
#
```

```
# 模型定义参考 examples/wide_and_deep 中 model.py  
from model import WideDeepModel  
from reader import WideDeepDataset  
  
model = WideDeepModel()  
model.net(is_train=True)  
  
def distributed_training(exe, train_model, train_data_path=". ./data", batch_size=10,  
epoch_num=1):  
    train_data = WideDeepDataset(data_path=train_data_path)  
    reader = train_model.loader.set_sample_generator(  
        train_data, batch_size=batch_size, drop_last=True, places=paddle.CPUPlace())  
  
    for epoch_id in range(epoch_num):  
        reader.start()  
        try:  
            while True:  
                loss_val = exe.run(program=paddle.static.default_main_program(),  
                                   fetch_list=[train_model.cost.name])  
                loss_val = np.mean(loss_val)  
                print("TRAIN ---> pass: {} loss: {}".format(epoch_id, loss_val))  
        except paddle.common_ops_import.core.EOFException:  
            reader.reset()
```

2.2.4 定义同步训练 Strategy 及 Optimizer

Title underline too short.

2.2.4 定义同步训练 Strategy 及 Optimizer

#####

在 Fleet API 中，用户可以使用 `fleet.DistributedStrategy()` 接口定义自己想要使用的分布式策略。

其中 `a_sync` 选项用于定义参数服务器相关的策略，当其被设定为 `False` 时，分布式训练将在同步的模式下进行。反之，当其被设定成 `True` 时，分布式训练将在异步的模式下进行。

```
# 定义异步训练
dist_strategy = fleet.DistributedStrategy()
dist_strategy.a_sync = True

# 定义同步训练
dist_strategy = fleet.DistributedStrategy()
dist_strategy.a_sync = False

# 定义Geo异步训练, Geo异步目前只支持SGD优化算法
dist_strategy = fleet.DistributedStrategy()
dist_strategy.a_sync = True
dist_strategy.a_sync_configs = {"k_steps": 100}

optimizer = paddle.optimizer.SGD(learning_rate=0.0001)
optimizer = fleet.distributed_optimizer(optimizer, dist_strategy)
optimizer.minimize(model.loss)
```

2.2.5 开始训练

Title underline too short.

2.2.5 开始训练

#####

完成模型及训练策略以后，我们就可以开始训练模型了。因为在参数服务器模式下会有不同的角色，所以根据不同节点分配不同的任务。

对于服务器节点，首先用 `init_server()` 接口对其进行初始化，然后启动服务并开始监听由训练节点传来的梯度。

同样对于训练节点，用 `init_worker()` 接口进行初始化后，开始执行训练任务。运行 `exe.run()` 接口开始训练，并得到训练中每一步的损失值。

```

if fleet.is_server():
    fleet.init_server()
    fleet.run_server()
else:
    exe = paddle.static.Executor(paddle.CPUPlace())
    exe.run(paddle.static.default_startup_program())

fleet.init_worker()

distributed_training(exe, model)

fleet.stop_worker()

```

2.3 运行训练脚本

定义完训练脚本后，我们就可以用 `python3 -m paddle.distributed.launch` 指令运行分布式任务了。其中 `server_num`, `worker_num` 分别为服务节点和训练节点的数量。在本例中，服务节点有 1 个，训练节点有 2 个。

```
python3 -m paddle.distributed.launch --server_num=1 --worker_num=2 --gpus=0,1 train.py
```

您将看到显示如下日志信息：

```

----- Configuration Arguments -----
gpus: 0,1
heter_worker_num: None
heter_workers:
http_port: None
ips: 127.0.0.1
log_dir: log
nproc_per_node: None
server_num: 1
servers:
training_script: train.py
training_script_args: []
worker_num: 2
workers:
-----
INFO 2021-05-06 12:14:26,890 launch.py:298] Run parameter-server mode. pserver_
arguments: ['--worker_num', '--server_num'], cuda count:8
INFO 2021-05-06 12:14:26,892 launch_utils.py:973] Local server start 1 processes.
First process distributed environment info (Only For Debug):

```

(下页继续)

(续上页)

	Distributed Envs	Value	
	PADDLE_TRAINERS_NUM	2	
	TRAINING_ROLE	PSERVER	
	POD_IP	127.0.0.1	
	PADDLE_GLOO_RENDEZVOUS	3	
	PADDLE_PServers_IP_PORT_LIST	127.0.0.1:34008	
	PADDLE_PORT	34008	
	PADDLE_WITH_GLOO	0	
	PADDLE_HETER_TRAINER_IP_PORT_LIST		
	PADDLE_TRAINER_ENDPOINTS	127.0.0.1:18913,127.0.0.1:10025	
	PADDLE_GLOO_HTTP_ENDPOINT	127.0.0.1:23053	
	PADDLE_GLOO_FS_PATH	/tmp/tmp8vqb8arq	

INFO 2021-05-06 12:14:26,902 launch_utils.py:1041] Local worker start 2 processes.

First process distributed environment info (Only For Debug):

	Distributed Envs	Value	
	PADDLE_GLOO_HTTP_ENDPOINT	127.0.0.1:23053	
	PADDLE_GLOO_RENDEZVOUS	3	
	PADDLE_PServers_IP_PORT_LIST	127.0.0.1:34008	

(下页继续)

(续上页)

PADDLE_WITH_GLOO	0	
PADDLE_TRAINER_ENDPOINTS	127.0.0.1:18913,127.0.0.1:10025	
FLAGS_selected_gpus	0	
PADDLE_GLOO_FS_PATH	/tmp/tmp8vqb8arq	
PADDLE_TRAINERS_NUM	2	
TRAINING_ROLE	TRAINER	
XPU_VISIBLE_DEVICES	0	
PADDLE_HETER_TRAINER_IP_PORT_LIST		
PADDLE_TRAINER_ID	0	
CUDA_VISIBLE_DEVICES	0	
FLAGS_selected_xpus	0	
=====		
INFO 2021-05-06 12:14:26,921 launch_utils.py:903] Please check servers, workers and heter_worker logs in log/workerlog.* , log/serverlog.* and log/heterlog.*		
INFO 2021-05-06 12:14:33,446 launch_utils.py:914] all workers exit, going to finish parameter server and heter_worker.		
INFO 2021-05-06 12:14:33,446 launch_utils.py:926] all parameter server are killed		

2.5.2 使用 FleetAPI 进行分布式训练

FleetAPI 设计说明

Fleet 是 PaddlePaddle 分布式训练的高级 API。Fleet 的命名出自于 PaddlePaddle，象征一个舰队中的多只双桨船协同工作。Fleet 的设计在易用性和算法可扩展性方面做出了权衡。用户可以很容易从单机版的训练程序，通过添加几行代码切换到分布式训练程序。此外，分布式训练的算法也可以通过 Fleet API 接口灵活定义。

Fleet API 快速上手示例

下面会针对 Fleet API 最常见的两种使用场景，用一个模型做示例，目的是让用户有快速上手体验的模板。

- 假设我们定义 MLP 网络如下：

```
import paddle

def mlp(input_x, input_y, hid_dim=128, label_dim=2):
    fc_1 = paddle.static.nn.fc(input=input_x, size=hid_dim, act='tanh')
    fc_2 = paddle.static.nn.fc(input=fc_1, size=hid_dim, act='tanh')
    prediction = paddle.static.nn.fc(input=[fc_2], size=label_dim, act='softmax')
    cost = paddle.static.nn.cross_entropy(input=prediction, label=input_y)
    avg_cost = paddle.static.nn.mean(x=cost)
    return avg_cost
```

- 定义一个在内存生成数据的 Reader 如下：

```
import numpy as np

def gen_data():
    return {"x": np.random.random(size=(128, 32)).astype('float32'),
            "y": np.random.randint(2, size=(128, 1)).astype('int64')}
```

- 单机 Trainer 定义

```
import paddle
from nets import mlp
from utils import gen_data

input_x = paddle.static.data(name="x", shape=[None, 32], dtype='float32')
input_y = paddle.static.data(name="y", shape=[None, 1], dtype='int64')

cost = mlp(input_x, input_y)
optimizer = paddle.optimizer.SGD(learning_rate=0.01)
optimizer.minimize(cost)
place = paddle.CUDAPlace(0)

exe = paddle.static.Executor(place)
exe.run(paddle.static.default_startup_program())
step = 1001
for i in range(step):
    cost_val = exe.run(feed=gen_data(), fetch_list=[cost.name])
    print("step%d cost=%f" % (i, cost_val[0]))
```

- Parameter Server 训练方法

参数服务器方法对于大规模数据，简单模型的并行训练非常适用，我们基于单机模型的定义给出使用 Parameter Server 进行训练的示例如下：

```
import paddle
paddle.enable_static()

import paddle.distributed.fleet.base.role_maker as role_maker
import paddle.distributed.fleet as fleet

from nets import mlp
from utils import gen_data

input_x = paddle.static.data(name="x", shape=[None, 32], dtype='float32')
input_y = paddle.static.data(name="y", shape=[None, 1], dtype='int64')

cost = mlp(input_x, input_y)
optimizer = paddle.optimizer.SGD(learning_rate=0.01)

role = role_maker.PaddleCloudRoleMaker()
fleet.init(role)

strategy = paddle.distributed.fleet.DistributedStrategy()
strategy.a_sync = True

optimizer = fleet.distributed_optimizer(optimizer, strategy)
optimizer.minimize(cost)

if fleet.is_server():
    fleet.init_server()
    fleet.run_server()

elif fleet.is_worker():
    place = paddle.CPUPlace()
    exe = paddle.static.Executor(place)
    exe.run(paddle.static.default_startup_program())

    step = 1001
    for i in range(step):
        cost_val = exe.run(
            program=paddle.static.default_main_program(),
            feed=gen_data(),
            fetch_list=[cost.name])
        print("worker_index: %d, step%d cost = %f" %
              (fleet.worker_index(), i, cost_val[0]))
```

- Collective 训练方法

Collective Training 通常在 GPU 多机多卡训练中使用，一般在复杂模型的训练中比较常见，我们基于上面的单机模型定义给出使用 Collective 方法进行分布式训练的示例如下：

```
import paddle
paddle.enable_static()

import paddle.distributed.fleet.base.role_maker as role_maker
import paddle.distributed.fleet as fleet

from nets import mlp
from utils import gen_data

input_x = paddle.static.data(name="x", shape=[None, 32], dtype='float32')
input_y = paddle.static.data(name="y", shape=[None, 1], dtype='int64')

cost = mlp(input_x, input_y)
optimizer = paddle.optimizer.SGD(learning_rate=0.01)
role = role_maker.PaddleCloudRoleMaker(is_collective=True)
fleet.init(role)

optimizer = fleet.distributed_optimizer(optimizer)
optimizer.minimize(cost)
place = paddle.CUDAPlace(0)

exe = paddle.static.Executor(place)
exe.run(paddle.static.default_startup_program())

step = 1001
for i in range(step):
    cost_val = exe.run(
        program=paddle.static.default_main_program(),
        feed=gen_data(),
        fetch_list=[cost.name])
    print("worker_index: %d, step%d cost = %f" %
          (fleet.worker_index(), i, cost_val[0]))
```

Fleet API 相关的接口说明

Fleet API 接口

- `init(role_maker=None)`
 - fleet 初始化，需要在使用 fleet 其他接口前先调用，用于定义多机的环境配置
- `is_worker()`
 - Parameter Server 训练中使用，判断当前节点是否是 Worker 节点，是则返回 True，否则返回 False
- `is_server(model_dir=None)`
 - Parameter Server 训练中使用，判断当前节点是否是 Server 节点，是则返回 True，否则返回 False
- `init_server()`
 - Parameter Server 训练中，fleet 加载 model_dir 中保存的模型相关参数进行 parameter server 的初始化
- `run_server()`
 - Parameter Server 训练中使用，用来启动 server 端服务
- `init_worker()`
 - Parameter Server 训练中使用，用来启动 worker 端服务
- `stop_worker()`
 - 训练结束后，停止 worker
- `distributed_optimizer(optimizer, strategy=None)`
 - 分布式优化算法装饰器，用户可带入单机 optimizer，并配置分布式训练策略，返回一个分布式的 optimizer

RoleMaker

- `PaddleCloudRoleMaker`
 - 描述：PaddleCloudRoleMaker 是一个高级封装，支持使用 `paddle.distributed.launch` 或者 `paddle.distributed.launch_ps` 启动脚本
 - Parameter Server 训练示例：

```
import paddle
paddle.enable_static()

import paddle.distributed.fleet.base.role_maker as role_maker
import paddle.distributed.fleet as fleet
```

(下页继续)

(续上页)

```
role = role_maker.PaddleCloudRoleMaker()  
fleet.init(role)
```

- 启动方法:

```
python -m paddle.distributed.launch_ps --worker_num 2 --server_num 2 trainer.  
py
```

- Collective 训练示例:

```
import paddle  
paddle.enable_static()  
  
import paddle.distributed.fleet.base.role_maker as role_maker  
import paddle.distributed.fleet as fleet  
  
role = role_maker.PaddleCloudRoleMaker(is_collective=True)  
fleet.init(role)
```

- 启动方法:

```
python -m paddle.distributed.launch trainer.py
```

- UserDefinedRoleMaker

- 描述: 用户自定义节点的角色信息, IP 和端口信息
- 示例:

```
import paddle  
paddle.enable_static()  
  
import paddle.distributed.fleet.base.role_maker as role_maker  
import paddle.distributed.fleet as fleet  
  
role = role_maker.UserDefinedRoleMaker(  
    current_id=0,  
    role=role_maker.Role.SERVER,  
    worker_num=2,  
    server_endpoints=["127.0.0.1:36011", "127.0.0.1:36012"])  
  
fleet.init(role)
```

2.6 性能调优

你可以通过以下内容，了解飞桨框架性能调优相关的内容：

- 自动混合精度训练：使用飞桨框架进行自动混合精度训练。
- 模型量化：使用飞桨框架进行模型量化。
- 模型性能分析：使用飞桨性能分析器对模型进行性能调试。

2.6.1 自动混合精度训练 (AMP)

一般情况下，训练深度学习模型时默认使用的数据类型（`dtype`）是 `float32`，每个数据占用 32 位的存储空间。为了节约显存消耗，业界提出了 16 位的数据类型（如 GPU 支持的 `float16`、`bfloat16`），每个数据仅需要 16 位的存储空间，比 `float32` 节省一半的存储空间，并且一些芯片可以在 16 位的数据上获得更快的计算速度，比如按照 NVIDIA 的数据显示，V100 GPU 上矩阵乘和卷积计算在 `float16` 的计算速度最大可达 `float32` 的 8 倍。

考虑到一些算子（OP）对数据精度的要求较高（如 `softmax`、`cross_entropy`），仍然需要采用 `float32` 进行计算；还有一些算子（如 `conv2d`、`matmul`）对数据精度不敏感，可以采用 `float16` / `bfloat16` 提升计算速度并降低存储空间，飞桨框架提供了自动混合精度（Automatic Mixed Precision，以下简称为 AMP）训练的方法，可在模型训练时，自动为算子选择合适的数据计算精度（`float32` 或 `float16` / `bfloat16`），在保持训练精度（accuracy）不损失的条件下，能够加速训练，可参考 2018 年百度与 NVIDIA 联合发表的论文：[MIXED PRECISION TRAINING](#)。本文将介绍如何使用飞桨框架实现自动混合精度训练。

一、概述

1.1 浮点数据类型

`Float16` 和 `bfloat16` (brain floating point) 都是一种半精度浮点数据类型，在计算机中使用 2 字节（16 位）存储。与计算中常用的单精度浮点数（`float32`）和双精度浮点数（`float64`）类型相比，`float16` 及 `bfloat16` 更适于在精度要求不高的场景中使用。

对比 `float32` 与 `float16` / `bfloat16` 的浮点格式，如图 1 所示：

上述数据类型存在如下数值特点：

- `float32` 的指数位占 8 位，尾数位占 23 位，可表示的数据动态范围是 $[2^{-126}, 2^{127}]$ ，是深度学习模型时默认使用的数据类型。
- `float16` 的指数位占 5 位，尾数位占 10 位，相比 `float32`，可表示的数据动态范围更低，最小可表示的正数数值为 2^{-14} ，最大可表示的数据为 65504，容易出现数值上溢出问题。
- `bfloat16` 的指数位 8 位，尾数为 7 位，其特点是牺牲精度从而获取更大的数据范围，可表示的数据范围与 `float32` 一致，但是与 `float16` 相比 `bfloat16` 可表示的数据精度更低，相比 `float16` 更易出现数值下溢出的问题。

1.2 AMP 计算过程

1.2.1 auto_cast 策略

飞桨框架采用了 **auto_cast** 策略实现模型训练过程中计算精度的自动转换及使用。通常情况下，模型参数使用单精度浮点格式存储（float32），在训练过程中，将模型参数从单精度浮点数（float32）转换为半精度浮点数（float16 或 bfloat16）参与前向计算，并得到半精度浮点数表示中间状态，然后使用半精度浮点数计算参数梯度，最后将参数梯度转换为单精度浮点数格式后，更新模型参数。计算过程如下图 2 所示：

上图中蓝色虚线框内的逻辑即是 AMP 策略下参数精度转换（cast）逻辑，通常 cast 操作所带来的开销是有限的，当使用 float16 / bfloat16 在前向计算（forward compute）及反向传播（backward propagation）过程中取得的计算性能收益大于 cast 所带来的开销时，开启 AMP 训练将得到更优的训练性能。

当模型参数在训练前即使用半精度浮点格式存数时（float16 / bfloat16），训练过程中将省去图 2 中的 cast 操作，可进一步提升模型训练性能，但是需要注意模型参数采用低精度数据类型进行存储，可能对模型最终的训练精度带来影响。计算过程如下图 3 所示：

1.2.2 grad_scaler 策略

如 1.1 所述，半精度浮点数的表示范围远小于单精度浮点数的表示范围，在深度学习领域，参数、中间状态和梯度的值通常很小，因此以半精度浮点数参与计算时容易出现数值下溢（underflow）的情况，即接近零的值下溢为零值。为了避免这个问题，飞桨采用 **grad_scaler** 策略。主要内容是：对训练 loss 乘以一个称为 loss_scaling 的缩放值，根据链式法则，在反向传播过程中，参数梯度也等价于相应地乘以了 loss_scaling 的值，在参数更新时再将梯度值相应地除以 loss_scaling 的值。

然而，在模型训练过程中，选择合适的 loss_scaling 值是个较大的挑战，因此，飞桨提供了 动态 loss_scaling 的机制：

1. 训练开始前，为 loss_scaling 设置一个较大的初始值 init_loss_scaling，默认为 2^{15} ，并设置 4 个用于动态调整 loss_scaling 大小的参数：incr_ratio=2.0、decr_ratio=0.5、incr_every_n_steps=1000、decr_every_n_nan_or_inf=2；
2. 启动训练后，在每次计算完成梯度后，对所有的梯度之进行检查，判断是否存在 nan/inf 并记录连续出现 nan/inf 的次数或连续未出现 nan/inf 的次数；
3. 当连续 incr_every_n_step 次迭代未出现 nan/inf 时，将 loss_scaling 乘 incr_ratio；
4. 当连续 decr_every_n_nan_or_inf 次迭代出现 nan/inf 时，将 loss_scaling 乘 decr_ratio；

1.3 支持硬件说明

飞桨框架支持如下硬件的混合精度训练，不同硬件已经支持的数据精度如下表所示：

以 Nvidia GPU 为例，介绍硬件加速机制：

在使用相同的超参数下，混合精度训练使用半精度浮点（float16 / bfloat16）和单精度（float32）浮点可达到与使用纯单精度（float32）训练相同的准确率，并可加速模型的训练速度，这主要得益于 Nvidia 从 Volta 架构开始推出的 Tensor Core 技术。

在使用 float16 计算时具有如下特点：

- float16 可降低一半的内存带宽和存储需求，这使得在相同的硬件条件下，可使用更大更复杂的模型以及更大的 batch size 大小。
- float16 可以充分利用 Nvidia Volta、Turing、Ampere 架构 GPU 提供的 Tensor Cores 技术。在相同的 GPU 硬件上，Tensor Core 的 float16 计算吞吐量是 float32 的 8 倍。

从 NVIDIA Ampere 架构开始，GPU 支持 bfloat16，其计算性能与 float16 持平。

说明：通过 nvidia-smi 指令可帮助查看 NVIDIA 显卡架构信息，混合精度训练适用的 NVIDIA GPU 计算能力至少为 7.0 的版本。此外如果已开启自动混合精度训练，飞桨框架会自动检测硬件环境是否符合要求，如不符合则将提供类似如下的警告信息：UserWarning: AMP only support NVIDIA GPU with Compute Capability 7.0 or higher, current GPU is: Tesla K40m, with Compute Capability: 3.5.。

1.4 适用场景说明

混合精度训练想要取得较高的收益通常都是在显存利用率较高的场景下，具体地讲就是模型中存在计算负载较大的 matmul、conv 等算子。如果模型本身上述算子的占比较小，那么混合精度取得的收益十分有限，同时为了引入混合精度训练还会带来精度转换（cast）的开销。

二、动态图训练开启 AMP 示例

使用飞桨框架提供的 API，能够在原始训练代码基础上快速开启自动混合精度训练，并根据设定的策略，在训练时相关算子（OP）的计算中，自动选择 float16 或 float32 进行计算。

依据 float16 数据类型在模型中的使用程度划分，飞桨框架的混合精度策略分为两个等级：

- **Level = ‘O1’**：采用黑白名单策略进行混合精度训练，黑名单中的 OP 将采用 float32 计算，白名单中的 OP 将采用 float16 计算，auto_cast 策略会自动将白名单 OP 的输入参数数据类型从 float32 转为 float16。飞桨框架默认设置了 [黑白名单 OP 列表](#)，对于不在黑白名单中的 OP，会依据该 OP 的全部输入数据类型进行推断，当全部输入均为 float16 时，OP 将直接采用 float16 计算，否则采用 float32 计算。计算逻辑可参考图 2。
- **Level = ‘O2’**：采用了比 O1 更为激进的策略，除了框架不支持 float16 计算的 OP 以及 O2 模式下自定义黑名单中的 OP，其他全部采用 float16 计算，此外，飞桨框架提供了将网络参数从 float32 转换为 float16

的接口，相比 O1 将进一步减少 auto_cast 逻辑中的 cast 操作，训练速度会有更明显的提升，但可能影响训练精度。计算逻辑可参考图 3。

飞桨框架推荐使用动态图模式训练模型，下面以动态图模式下单卡（GPU）训练场景为例，分别介绍使用基础 API 和高层 API 开启 AMP 训练的不同使用方式。

2.1 使用基础 API

飞桨框架在动态图模式下实现 AMP 训练，通常需要组合 paddle.amp.auto_cast 和 paddle.amp.GradScaler API 使用。

- paddle.amp.auto_cast：创建上下文环境，开启自动混合精度策略。
- paddle.amp.GradScaler：用于控制 loss 缩放比例，规避浮点数下溢问题。

另外在自动混合精度策略设置为 Level = ‘O2’ 模式时，除了使用以上两个 API，同时使用 paddle.amp.decorate API 将网络参数从 float32 转换为 float16，减少 auto_cast 逻辑中的 cast 操作。

2.1.1 动态图 float32 训练

本例作为参照组，先执行一个普通的 float32 训练，用于对比 AMP 训练的加速效果。

1) 构建一个神经网络

为了更明显地对比出 AMP 训练所能带来的性能提升，构建一个由多达九层 Linear 组成的神经网络，每层 Linear 网络由 matmul 算子及 add 算子组成，其中 matmul 算子是 float16 数据下加速效果比较好的算子。

```
import time
import paddle
import paddle.nn as nn
import numpy

paddle.seed(100)
numpy.random.seed(100)
place = paddle.CUDAPlace(0)
# 定义神经网络 SimpleNet，该网络由九层 Linear 组成
class SimpleNet(paddle.nn.Layer):
    def __init__(self, input_size, output_size):
        super(SimpleNet, self).__init__()
        # 九层 Linear，每层 Linear 网络由 matmul 算子及 add 算子组成
        self.linears = paddle.nn.LayerList(
            [paddle.nn.Linear(input_size, output_size) for i in range(9)])
    def forward(self, x):
        for i, l in enumerate(self.linears):

```

(下页继续)

(续上页)

```
x = self.linears[i](x)
return x
```

2) 设置训练的相关参数及训练数据

将 `input_size`、`output_size` 和 `batch_size` 的值设为较大的值，尽可能占满显存，可更明显地对比 AMP 训练加速效果；根据 Tensor Core 的使用建议，当矩阵维数是 8 的倍数时，`float16` 精度下加速效果更优，因此将 `batch_size` 设置为 8 的倍数。

```
epochs = 2
input_size = 8192    # 设为较大的值，可更明显地对比AMP训练加速效果
output_size = 8192   # 设为较大的值，可更明显地对比AMP训练加速效果
batch_size = 2048    # batch_size为8的倍数加速效果更优
nums_batch = 10

# 定义Dataloader
from paddle.io import Dataset
class RandomDataset(Dataset):
    def __init__(self, num_samples):
        self.num_samples = num_samples

    def __getitem__(self, idx):
        data = numpy.random.random([input_size]).astype('float32')
        label = numpy.random.random([output_size]).astype('float32')
        return data, label

    def __len__(self):
        return self.num_samples

dataset = RandomDataset(nums_batch * batch_size)
loader = paddle.io.DataLoader(dataset, batch_size=batch_size, shuffle=False, drop_
    ↴last=True, num_workers=0)
```

3) 执行训练并记录训练时长

```
mse = paddle.nn.MSELoss() # 定义损失计算函数
model = SimpleNet(input_size, output_size) # 定义SimpleNet模型
optimizer = paddle.optimizer.SGD(learning_rate=0.0001, parameters=model.parameters())
    ↴ # 定义SGD优化器

train_time = 0 # 记录总训练时长
for epoch in range(epochs):
    for i, (data, label) in enumerate(loader):
        start_time = time.time() # 记录开始训练时刻
```

(下页继续)

(续上页)

```

label._to(place) # 将label数据拷贝到gpu
# 前向计算 (9层Linear网络, 每层由matmul、add算子组成)
output = model(data)
# loss计算
loss = mse(output, label)
# 反向传播
loss.backward()
# 更新参数
optimizer.step()
optimizer.clear_grad(set_to_zero=False)
# 记录训练loss及训练时长
train_loss = loss.numpy()
train_time += time.time() - start_time

print("loss:", train_loss)
print("使用float32模式训练耗时:{:.3f} sec".format(train_time/(epochs*nums_batch)))
# loss: [0.6486028]
# 使用float32模式训练耗时:0.529 sec

```

注：如果该示例代码在你的机器上显示显存不足相关的错误，请尝试将 `input_size`、`output_size`、`batch_size` 调小。

2.1.2 动态图 AMP-O1 训练

使用 AMP-O1 训练，需要在 float32 训练代码的基础上添加两处逻辑：

- 逻辑 1：使用 `paddle.amp.auto_cast` 创建 AMP 上下文环境，开启自动混合精度策略 `Level = 'O1'`。在该上下文环境影响范围内，框架会根据预设的黑白名单，自动确定每个 OP 的输入数据类型（`float32` 或 `float16` / `bfloat16`）。也可以在该 API 中添加自定义黑白名单 OP 列表。
- 逻辑 2：使用 `paddle.amp.GradScaler` 控制 loss 缩放比例，规避浮点数下溢问题。在模型训练过程中，框架默认开启动态 **loss scaling** 机制 (`use_dynamic_loss_scaling=True`)，具体介绍见 1.2.2 `grad_scaler` 策略。

```

mse = paddle.nn.MSELoss() # 定义损失计算函数
model = SimpleNet(input_size, output_size) # 定义SimpleNet模型
optimizer = paddle.optimizer.SGD(learning_rate=0.0001, parameters=model.parameters())
# 定义SGD优化器

# 逻辑2: 可选, 定义GradScaler, 用于缩放loss比例, 避免浮点数溢出, 默认开启动态更新loss_scaling机制
scaler = paddle.amp.GradScaler(init_loss_scaling=1024)

```

(下页继续)

(续上页)

```

train_time = 0 # 记录总训练时长
for epoch in range(epochs):
    for i, (data, label) in enumerate(loader):
        start_time = time.time() # 记录开始训练时刻
        label._to(place) # 将label数据拷贝到gpu
        # 逻辑1: 创建 AMP-O1 auto_cast 环境, 开启自动混合精度训练, 将 add_
        ↪算子添加到自定义白名单中 (custom_white_list) ,
        # 因此前向计算过程中该算子将采用 float16 数据类型计算
        with paddle.amp.auto_cast(custom_white_list={'elementwise_add'}, level='O1'):
            output = model(data) #
        ↪前向计算 (9层 Linear 网络, 每层由 matmul、add 算子组成)
        loss = mse(output, label) # loss计算
        # 逻辑2: 使用 GradScaler 完成 loss 的缩放, 用缩放后的 loss 进行反向传播
        scaled = scaler.scale(loss) # loss缩放, 乘以系数loss_scaling
        scaled.backward() # 反向传播
        scaler.step(optimizer) # 更新参数 (参数梯度先除系数loss_
        ↪scaling再更新参数)
        scaler.update() # 基于动态loss_scaling策略更新loss_scaling系数
        optimizer.clear_grad(set_to_zero=False)
        # 记录训练loss及训练时长
        train_loss = loss.numpy()
        train_time += time.time() - start_time

print("loss:", train_loss)
print("使用AMP-O1模式耗时:{:.3f} sec".format(train_time/(epochs*nums_batch)))
# loss: [0.6486219]
# 使用AMP-O1模式耗时:0.118 sec

```

2.1.3 动态图 AMP-O2 训练

使用 AMP-O2 训练, 需要在 float32 训练代码的基础上添加三处逻辑:

O2 模式采用了比 O1 更为激进的策略, 除了框架不支持 FP16 计算的 OP, 其他全部采用 FP16 计算, 需要在训练前将网络参数从 FP32 转为 FP16, 在 FP32 代码的基础上添加三处逻辑:

- 逻辑 1: 在训练前使用 paddle.amp.decorate 将网络参数从 float32 转换为 float16。
- 逻辑 2: 使用 paddle.amp.auto_cast 创建 AMP 上下文环境, 开启自动混合精度策略 Level = 'O2' 在该上下文环境影响范围内, 框架会将所有支持 float16 的 OP 均采用 float16 进行计算 (自定义的黑名单除外), 其他 OP 采用 float32 进行计算。
- 逻辑 3: 使用 paddle.amp.GradScaler 控制 loss 缩放比例, 规避浮点数下溢问题。用法与 AMP-O1 中相同。

```

mse = paddle.nn.MSELoss() # 定义损失计算函数
model = SimpleNet(input_size, output_size) # 定义SimpleNet模型
optimizer = paddle.optimizer.SGD(learning_rate=0.0001, parameters=model.parameters()) ↵
# 定义SGD优化器

# 逻辑1: 在level='O2'模式下, 将网络参数从FP32转换为FP16
model = paddle.amp.decorate(models=model, level='O2')

# 逻辑3: 可选, 定义
→GradScaler, 用于缩放loss比例, 避免浮点数溢出, 默认开启动态更新loss_scaling机制
scaler = paddle.amp.GradScaler(init_loss_scaling=1024)

train_time = 0 # 记录总训练时长
for epoch in range(epochs):
    for i, (data, label) in enumerate(loader):
        start_time = time.time() # 记录开始训练时刻
        label._to(place) # 将label数据拷贝到gpu
        # 逻辑2: 创建AMP-O2 auto_cast
        →环境, 开启自动混合精度训练, 前向计算过程中该算子将采用float16数据类型计算
        with paddle.amp.auto_cast(level='O2'):
            output = model(data) #
            loss = mse(output, label) # loss计算
            # 逻辑3: 使用GradScaler完成loss的缩放, 用缩放后的loss进行反向传播
            scaled = scaler.scale(loss) # loss缩放, 乘以系数loss_scaling
            scaled.backward() # 反向传播
            scaler.step(optimizer) # 更新参数 (参数梯度先除系数loss_
            →scaling再更新参数)
            scaler.update() # 基于动态loss_scaling策略更新loss_scaling系数
            optimizer.clear_grad(set_to_zero=False)
            # 记录训练loss及训练时长
            train_loss = loss.numpy()
            train_time += time.time() - start_time

print("loss=", train_loss)
print("使用AMP-O2模式耗时:{:.3f} sec".format(train_time/(epochs*nums_batch)))
# loss= [0.6743]
# 使用AMP-O2模式耗时:0.102 sec

```

2.1.4 对比不同模式下训练速度

动态图 FP32 及 AMP 训练的精度速度对比如下表所示：

从上表统计结果可以看出，相比普通的 float32 训练模式，**AMP-O1** 模式训练速度提升约为 **4.5** 倍，**AMP-O2** 模式训练速度提升约为 **5.2** 倍。

注：上述实验构建了一个理想化的实验模型，其 matmul 算子占比较高，所以加速比较明显，实际模型的加速效果与模型特点有关，理论上数值计算如 matmul、conv 占比较高的模型加速效果更明显。此外，受机器环境影响，上述示例代码的训练耗时统计可能存在差异，该影响主要包括：GPU 利用率、CPU 利用率等，本示例的测试机器配置如下：

2.2 使用高层 API

飞桨框架 2.0 开始全新推出高层 API，是对飞桨基础 API 的进一步封装与升级，提供了更加简洁易用的 API，提升了飞桨框架的易学易用性，并增强飞桨的功能。高层 API 下 AMP 使用示例如下，主要通过 paddle.Model.prepare 的 **amp_configs** 参数传入 AMP 相关配置。

```
import paddle
import paddle.nn as nn
import paddle.vision.transforms as T

def run_example_code():
    device = paddle.set_device('gpu')
    # 利用高层API定义神经网络
    net = nn.Sequential(nn.Flatten(1), nn.Linear(784, 200), nn.Tanh(), nn.Linear(200, ↴10))
    model = paddle.Model(net)
    # 定义优化器
    optim = paddle.optimizer.SGD(learning_rate=1e-3, parameters=model.parameters())
    # 初始化神经网络
    amp_configs = {
        "level": "O1", # level对应AMP模式：O1、O2
        "custom_white_list": {'conv2d'}, # 自定义白名单，同时还支持custom_black_list
        "use_dynamic_loss_scaling": True # 动态loss_scaling策略
    }
    model.prepare(optim,
                 paddle.nn.CrossEntropyLoss(),
                 paddle.metric.Accuracy(),
                 amp_configs=amp_configs)
    # 数据准备
    transform = T.Compose([TTranspose(), T.Normalize([127.5], [127.5])])
    data = paddle.vision.datasets.MNIST(mode='train', transform=transform)
    # 使用amp进行模型训练
```

(下页继续)

(续上页)

```
model.fit(data, epochs=2, batch_size=32, verbose=1)

if paddle.is_compiled_with_cuda():
    run_example_code()
```

三、其他使用场景

前文介绍了动态图模式下单卡（GPU）训练的方法，与之类似，分布式训练和动转静训练时可以采用同样的方法开启AMP。接下来主要介绍不同的静态图模式下开启AMP训练的方法，以及AMP训练的进阶用法，如梯度累加。

3.1 动态图下使用梯度累加

梯度累加是指在模型训练过程中，训练一个batch的数据得到梯度后，不立即用该梯度更新模型参数，而是继续下一个batch数据的训练，得到梯度后继续循环，多次循环后梯度不断累加，直至达到一定次数后，用累加的梯度更新参数，这样可以起到变相扩大batch_size的作用。受限于显存大小，可能无法开到更大的batch_size，使用梯度累加可以实现增大batch_size的作用。

动态图模式天然支持梯度累加，即只要不调用梯度清零clear_grad方法，动态图的梯度会一直累积，在自动混合精度训练中，梯度累加的使用方式如下：

```
mse = paddle.nn.MSELoss() # 定义损失计算函数
model = SimpleNet(input_size, output_size) # 定义SimpleNet模型
optimizer = paddle.optimizer.SGD(learning_rate=0.0001, parameters=model.parameters())
# 定义SGD优化器

accumulate_batches_num = 10 # 梯度累加中batch的数量

# 定义GradScaler，用于缩放loss比例，避免浮点数溢出，默认开启动态更新loss_scaling机制
scaler = paddle.amp.GradScaler(init_loss_scaling=1024)

train_time = 0 # 记录总训练时长
for epoch in range(epochs):
    for i, (data, label) in enumerate(loader):
        start_time = time.time() # 记录开始训练时刻
        label._to(place) # 将label数据拷贝到gpu
        # 创建AMP-O2 auto_cast_
    # 环境，开启自动混合精度训练，前向计算过程中该算子将采用float16数据类型计算
    with paddle.amp.auto_cast(level='O1'):
        output = model(data)
        loss = mse(output, label)
    # 使用GradScaler完成loss的缩放，用缩放后的loss进行反向传播
```

(下页继续)

(续上页)

```

scaled = scaler.scale(loss)    # loss缩放，乘以系数loss_scaling
scaled.backward()              # 反向传播
# 当累计的batch为accumulate_batchs_num时，更新模型参数
if (i + 1) % accumulate_batchs_num == 0:
    scaler.step(optimizer)    # 更新参数（参数梯度先除系数loss_
    ↪scaling再更新参数）
    scaler.update()           # 基于动态loss_scaling策略更新loss_scaling系数
    optimizer.clear_grad(set_to_zero=False)
# 记录训练loss及训练时长
train_loss = loss.numpy()
train_time += time.time() - start_time

print("loss:", train_loss)
print("使用AMP-O1模式耗时:{:.3f} sec".format(train_time/(epochs*nums_batch)))
# loss: [0.6602017]
# 使用AMP-O1模式耗时:0.113 sec

```

上面的例子中，每经过`accumulate_batchs_num`个batch的训练步骤，进行1次参数更新。

3.2 静态图训练开启 AMP

飞桨框架在静态图模式下实现AMP训练，功能逻辑与动态图类似，只是调用的接口有区别，使用如下API：`paddle.static.amp.decorate`、`paddle.static.amp.fp16_guard`。

- `paddle.static.amp.decorate`: 对传入的优化器进行装饰，增添AMP逻辑，同时可通过该接口配置`grad_scaler`策略的相关参数。
- `paddle.static.amp.fp16_guard`: 在AMP-O2模式下，控制float16的作用域，只有在上下文管理器`fp16_guard`内部才会使用float16计算。

3.2.1 静态图 float32 训练

采用与2.1.1节动态图训练相同的网络结构，静态图网络初始化如下：

```

paddle.enable_static() # 开启静态图模式
place = paddle.CUDAPlace(0)
# 定义静态图的program
main_program = paddle.static.default_main_program()
startup_program = paddle.static.default_startup_program()
# 定义由9层Linear组成的神经网络
model = SimpleNet(input_size, output_size)
# 定义损失函数
mse_loss = paddle.nn.MSELoss()

```

静态图训练代码如下：

```
# 定义训练数据及标签
data = paddle.static.data(name='data', shape=[batch_size, input_size], dtype='float32'
˓→')
label = paddle.static.data(name='label', shape=[batch_size, input_size], dtype=
˓→'float32')
# 前向计算
predict = model(data)
# 损失计算
loss = mse_loss(predict, label)
# 定义优化器
optimizer = paddle.optimizer.SGD(learning_rate=0.0001, parameters=model.parameters())
optimizer.minimize(loss)
# 定义静态图执行器
exe = paddle.static.Executor(place)
exe.run(startup_program)

train_time = 0 # 记录总训练时长
for epoch in range(epochs):
    for i, (train_data, train_label) in enumerate(loader()):
        start_time = time.time() # 记录开始训练时刻
        # 执行训练
        train_loss = exe.run(main_program, feed={data.name: train_data, label.name:_
˓→train_label }, fetch_list=[loss.name], use_program_cache=True)
        # 记录训练时长
        train_time += time.time() - start_time

    print("loss:", train_loss)
print("使用FP32模式耗时:{:.3f} sec".format(train_time/(epochs*nums_batch)))
# loss: [array([0.6486028], dtype=float32)]
# 使用FP32模式耗时:0.531 sec
```

3.2.2 静态图 AMP-O1 训练

静态图通过 `paddle.static.amp.decorate` 对优化器进行封装、通过 `paddle.static.amp.CustomOpLists` 定义黑白名单，即可开启混合精度训练，示例代码如下：

```
# 定义训练数据及标签
data = paddle.static.data(name='data', shape=[batch_size, input_size], dtype='float32'
˓→')
label = paddle.static.data(name='label', shape=[batch_size, input_size], dtype=
˓→'float32')
# 前向计算
```

(下页继续)

(续上页)

```

predict = model(data)
# 损失计算
loss = mse_loss(predict, label)
# 定义静态图执行器
optimizer = paddle.optimizer.SGD(learning_rate=0.0001, parameters=model.parameters())

# 1) 通过 `CustomOpLists` 自定义黑白名单
amp_list = paddle.static.amp.CustomOpLists(custom_white_list=['elementwise_add'])

# 2) 通过 `decorate` 对优化器进行封装：
optimizer = paddle.static.amp.decorate(
    optimizer=optimizer,
    amp_lists=amp_list,
    init_loss_scaling=128.0,
    use_dynamic_loss_scaling=True)

optimizer.minimize(loss)

# 定义静态图执行器
exe = paddle.static.Executor(place)
exe.run(startup_program)

train_time = 0 # 记录总训练时长
for epoch in range(epochs):
    for i, (train_data, train_label) in enumerate(loader()):
        start_time = time.time() # 记录开始训练时刻
        # 执行训练
        train_loss = exe.run(main_program, feed={data.name: train_data, label.name: train_label }, fetch_list=[loss.name], use_program_cache=True)
        # 记录训练时长
        train_time += time.time() - start_time

    print("loss:", train_loss)
print("使用AMP-O1模式耗时:{:.3f} sec".format(train_time/(epochs*nums_batch)))
# loss: [array([0.6486222], dtype=float32)]
# 使用AMP-O1模式耗时:0.117 sec

```

paddle.static.amp.CustomOpLists 用于自定义黑白名单，将 add 算子加入了白名单中，Linear 网络将全部执行在 float16 下。

3.2.3 静态图 AMP-O2 训练

静态图开启 AMP-O2 有两种方式：

- 使用 `paddle.static.amp.fp16_guard` 控制 float16 应用的范围，在该语境下的所有 OP 将执行 float16 计算，其他 OP 执行 float32 计算；
- 不使用 `paddle.static.amp.fp16_guard` 控制 float16 应用的范围，网络的全部 OP 执行 float16 计算（除去自定义黑名单的 OP、不支持 float16 计算的 OP）

1) 设置 `paddle.static.amp.decorate` 的参数 `use_pure_fp16` 为 `True`，同时设置参数 `use_fp16_guard` 为 `False`。

```
data = paddle.static.data(name='data', shape=[batch_size, input_size], dtype='float32'
                         ↪)
label = paddle.static.data(name='label', shape=[batch_size, input_size], dtype=
                           ↪'float32')

predict = model(data)
loss = mse_loss(predict, label)

optimizer = paddle.optimizer.SGD(learning_rate=0.0001, parameters=model.parameters())

# 1) 通过 `decorate` 对优化器进行封装，use_fp16_
    ↪guard 设置为 False，网络的全部 op 执行 FP16 计算
optimizer = paddle.static.amp.decorate(
    optimizer=optimizer,
    init_loss_scaling=128.0,
    use_dynamic_loss_scaling=True,
    use_pure_fp16=True,
    use_fp16_guard=False)

optimizer.minimize(loss)

exe = paddle.static.Executor(place)
exe.run(startup_program)

# 2) 利用 `amp_init` 将网络的 FP32 参数转换 FP16 参数。
optimizer.amp_init(place, scope=paddle.static.global_scope())

train_time = 0 # 记录总训练时长
for epoch in range(epochs):
    for i, (train_data, train_label) in enumerate(loader()):
        start_time = time.time() # 记录开始训练时刻
        # 执行训练
        train_loss = exe.run(main_program, feed={data.name: train_data, label.name:
                                                ↪train_label }, fetch_list=[loss.name], use_program_cache=True)
```

(下页继续)

(续上页)

```

# 记录训练时长
train_time += time.time() - start_time

print("loss:", train_loss)
print("使用AMP-O2模式耗时:{:.3f} sec".format(train_time/(epochs*nums_batch)))
# loss: [array([0.6743], dtype=float16)]
# 使用AMP-O2模式耗时:0.098 sec

```

注：在 AMP-O2 模式下，网络参数将从 float32 转为 float16，输入数据需要相应输入 float16 类型数据，因此需要将 class RandomDataset 中初始化的数据类型设置为 float16。

- 2) 设置 paddle.static.amp.decorate 的参数 use_pure_fp16 为 True，同时设置参数 use_fp16_guard 为 True，通过 paddle.static.amp.fp16_guard 控制使用 float16 的计算范围。

在模型定义的代码中加入 fp16_guard 控制部分网络执行在 float16 下：

```

class SimpleNet(paddle.nn.Layer):
    def __init__(self, input_size, output_size):
        super(SimpleNet, self).__init__()
        self.linears = paddle.nn.LayerList(
            [paddle.nn.Linear(input_size, output_size) for i in range(9)])

    def forward(self, x):
        for i, l in enumerate(self.linears):
            if i > 0:
                # 在模型定义中通过fp16_guard控制使用float16的计算范围
                with paddle.static.amp.fp16_guard():
                    x = self.linears[i](x)
            else:
                x = self.linears[i](x)
        return x

```

该模式下的训练代码如下：

```

data = paddle.static.data(name='data', shape=[batch_size, input_size], dtype='float32')
label = paddle.static.data(name='label', shape=[batch_size, input_size], dtype='float32')

predict = model(data)
loss = mse_loss(predict, label)

optimizer = paddle.optimizer.SGD(learning_rate=0.0001, parameters=model.parameters())

```

(下页继续)

(续上页)

```
# 1) 通过 `decorate` 对优化器进行封装:  
optimizer = paddle.static.amp.decorate(  
    optimizer=optimizer,  
    init_loss_scaling=128.0,  
    use_dynamic_loss_scaling=True,  
    use_pure_fp16=True,  
    use_fp16_guard=True)  
  
optimizer.minimize(loss)  
  
exe = paddle.static.Executor(place)  
exe.run(startup_program)  
  
# 2) 利用 `amp_init` 将网络的 FP32 参数转换 FP16 参数。  
optimizer.amp_init(place, scope=paddle.static.global_scope())  
  
train_time = 0 # 记录总训练时长  
for epoch in range(epochs):  
    for i, (train_data, train_label) in enumerate(loader()):  
        start_time = time.time() # 记录开始训练时刻  
        # 执行训练  
        train_loss = exe.run(main_program, feed={data.name: train_data, label.name:  
        ↪train_label }, fetch_list=[loss.name], use_program_cache=True)  
        # 记录训练时长  
        train_time += time.time() - start_time  
  
    print("loss:", train_loss)  
print("使用AMP-O2模式耗时:{:.3f} sec".format(train_time/(epochs*nums_batch)))  
# loss: [array([0.6691731], dtype=float32)]  
# 使用AMP-O2模式耗时:0.140 sec
```

3.2.4 对比不同模式下训练速度

静态图 FP32 及 AMP 训练的精度速度对比如下表所示：

从上表统计结果可以看出，使用自动混合精度训练 O1 模式训练速度提升约为 4.5 倍，O2 模式训练速度提升约为 5.4 倍。

四、其他注意事项

飞桨 AMP 提升模型训练性能的根本原因是：利用 Tensor Core 来加速 float16 下的 matmul 和 conv 等运算，为了获得最佳的加速效果，Tensor Core 对矩阵乘和卷积运算有一定的使用约束，约束如下：

1. 通用矩阵乘 (GEMM) 定义为： $C = A * B + C$ ，其中：

- A 维度为： $M \times K$
- B 维度为： $K \times N$
- C 维度为： $M \times N$

矩阵乘使用建议如下：根据 Tensor Core 使用建议，当矩阵维数 M、N、K 是 8 (A100 架构 GPU 为 16) 的倍数时 (FP16 数据下)，性能最优。

2. 卷积计算定义为： $NKPQ = NCHW * KCRS$ ，其中：

- N 代表：batch size
- K 代表：输出数据的通道数
- P 代表：输出数据的高度
- Q 代表：输出数据的宽度
- C 代表：输入数据的通道数
- H 代表：输入数据的高度
- W 代表：输入数据的宽度
- R 代表：滤波器的高度
- S 代表：滤波器的宽度

卷积计算使用建议如下：

- 输入/输出数据的通道数 (C/K) 可以被 8 整除 (FP16)，(cudnn7.6.3 及以上的版本，如果不是 8 的倍数将会被自动填充)
- 对于网络第一层，通道数设置为 4 可以获得最佳的运算性能 (NVIDIA 为网络的第一层卷积提供了特殊实现，使用 4 通道性能更优)
- 设置内存中的张量布局为 NHWC 格式 (如果输入 NCHW 格式，Tensor Core 会自动转换为 NHWC，当输入输出数值较大的时候，这种转置的开销往往更大)

五、AMP 常见问题及处理方法

飞桨 AMP 常见问题及处理方法如下：

1. 开启 AMP 训练后无加速效果或速度下降

可能原因 1：所用显卡并不支持 AMP 加速，可在训练日志中查看如下 warning 信息：`UserWarning: AMP only support NVIDIA GPU with Compute Capability 7.0 or higher, current GPU is: Tesla K40m, with Compute Capability: 3.5.;`

可能原因 2：模型是轻计算、重调度的类型，计算负载较大的 `matmul`、`conv` 等操作占比较低，可通过 `nvidia-smi` 实时查看显卡显存利用率（`Memory Usage` 及 `GPU_Util` 参数）。

针对上述原因，建议关闭混合精度训练。

2. AMP-O2 与分布式训练同时使用时抛出 `RuntimeError`: For distributed AMP training, you should first use `paddle.amp.decorate()` to decotate origin model, and then call `paddle.DataParallel` get distributed model.

原因：AMP-O2 的分布式训练，要求 `paddle.amp.decorate` 需要声明在 `paddle.DataParallel` 初始化分布式训练的网络前。

正确用法如下：

```
import paddle
model = SimpleNet(input_size, output_size) # 定义SimpleNet模型
model = paddle.amp.decorate(models=model, level='O2') # paddle.amp.
# paddle.amp.decorate 需要在 paddle.DataParallel 前
dp_model = paddle.DataParallel(model)
```

2.6.2 飞桨模型量化

深度学习技术飞速发展，在很多任务和领域都超越传统方法。但是，深度学习模型通常需要较大的存储空间和计算量，给部署应用带来了不小挑战。

模型量化作为一种常见的模型压缩方法，使用整数替代浮点数进行存储和计算，可以减少模型存储空间、加快推理速度、降低计算内存，助力深度学习应用的落地。

飞桨提供了模型量化全流程解决方案，首先使用 PaddleSlim 产出量化模型，然后使用 Paddle Inference 和 Paddle Lite 部署量化模型。

产出量化模型

飞桨模型量化全流程解决方案中，PaddleSlim 负责产出量化模型。

PaddleSlim 支持三种模型量化方法：动态离线量化方法、静态离线量化方法和量化训练方法。这三种量化方法的特点如下图。

动态离线量化方法不需要使用样本数据，也不会对模型进行训练。在模型产出阶段，动态离线量化方法将模型权重从浮点数量化成整数。在模型部署阶段，将权重从整数反量化成浮点数，使用浮点数运算进行预测推理。这种方式主要减少模型存储空间，对权重读取费时的模型有一定加速作用，对模型精度影响较小。

静态离线量化方法要求有少量无标签样本数据，需要执行模型的前向计算，不会对模型进行训练。在模型产出阶段，静态离线量化方法使用样本数据执行模型的前向计算，同时对量化 OP 的输入输出进行采样，然后计算量化信息。在模型部署阶段，使用计算好的量化信息对输入进行量化，基于整数运算进行预测推理。静态离线量化方法可以减少模型存储空间、加快模型推理速度、降低计算内存，同时量化模型只存在较小的精度损失。

量化训练方法要求有大量有标签样本数据，需要对模型进行较长时间的训练。在模型产出阶段，量化训练方法使用模拟量化的思想，在模型训练过程中会更新权重，实现拟合、减少量化误差的目的。在模型部署阶段，量化训练方法和静态离线量化方法一致，采用相同的预测推理方式，在存储空间、推理速度、计算内存三方面实现相同的收益。更重要的是，量化训练方法对模型精度只有极小的影响。

根据使用条件和压缩目的，大家可以参考下图选用不同的模型量化方法产出量化模型。

产出量化模型的使用方法、Demo 和 API，请参考[PaddleSlim 文档](#)。

部署量化模型

飞桨模型量化全流程解决方案中，Paddle Inference 负责在服务器端（X86 CPU 和 Nvidia GPU）部署量化模型，Paddle Lite 负责在移动端（ARM CPU）上部署量化模型。

X86 CPU 和 Nvidia GPU 上支持部署 PaddleSlim 静态离线量化方法和量化训练方法产出的量化模型。ARM CPU 上支持部署 PaddleSlim 动态离线量化方法、静态离线量化方法和量化训练方法产出的量化模型。

因为动态离线量化方法产出的量化模型主要是为了压缩模型体积，主要应用于移动端部署，所以在 X86 CPU 和 Nvidia GPU 上暂不支持这类量化模型。

NV GPU 上部署量化模型

使用 PaddleSlim 静态离线量化方法和量化训练方法产出量化模型后，可以使用 Paddle Inference 在 Nvidia GPU 上部署该量化模型。

Nvidia GPU 上部署常规模型的流程是：准备 TensorRT 环境、配置 Config、创建 Predictor、执行。Nvidia GPU 上部署量化模型和常规模型大体相似，需要改动的是：指定 TensorRT 配置时将 precision_mode 设置为 paddle_infer.PrecisionType.Int8，将 use_calib_mode 设为 False。

```
config.enable_tensorrt_engine(  
    workspace_size=1<<30,  
    max_batch_size=1,  
    min_subgraph_size=5,  
    precision_mode=paddle_infer.PrecisionType.Int8,  
    use_static=False,  
    use_calib_mode=False)
```

Paddle Inference 的详细说明, 请参考[文档](#)。

Nvidia GPU 上部署量化模型的详细说明, 请参考[文档](#)。

X86 CPU 上部署量化模型

使用 PaddleSlim 静态离线量化方法和量化训练方法产出量化模型后, 可以使用 Paddle Inference 在 X86 CPU 上部署该量化模型。

X86 CPU 上部署量化模型, 首先检查 X86 CPU 支持指令集, 然后转换量化模型, 最后在 X86 CPU 上执行预测。

Paddle Inference 的详细说明, 请参考[文档](#)。

X86 CPU 上部署量化模型的详细说明, 请参考[文档](#)。

1) 检查 X86 CPU 支持指令集

大家可以在命令行中输入 lscpu 查看本机支持指令。

在支持 avx512、不支持 avx512_vnni 的 X86 CPU 上 (如: SkyLake, Model name: Intel(R) Xeon(R) Gold X1XX), 量化模型性能为原始模型性能的 1.5 倍左右。

在支持 avx512 和 avx512_vnni 的 X86 CPU 上 (如: Cascade Lake, Model name: Intel(R) Xeon(R) Gold X2XX), 量化模型的精度和性能最高, 量化模型性能为原始模型性能的 3~3.7 倍。

2) 转换量化模型

下载[转换脚本](#)到本地。

```
wget https://github.com/PaddlePaddle/Paddle/blob/develop/python/paddle/fluid/contrib/  
→slim/tests/save_quant_model.py
```

使用脚本转换量化模型, 比如:

```
python save_quant_model.py \  
--quant_model_path=/PATH/TO/PADDLESIM/GENERATE/MODEL \  
--int8_model_save_path=/PATH/TO/SAVE/CONVERTED/MODEL
```

3) 执行预测

准备预测库，加载转换后的量化模型，创建 Predictor，进行预测。

注意，在 X86 CPU 预测端部署量化模型，必须开启 MKLDNN，不要开启 IrOptim（模型已经转换好）。

4) 数据展示

[图像分类 INT8 模型在 Intel\(R\) Xeon\(R\) Gold 6271 上精度](#)

[图像分类 INT8 模型在 Intel\(R\) Xeon\(R\) Gold 6271 单核上性能](#)

[Ernie INT8 模型在 Intel\(R\) Xeon\(R\) Gold 6271 的精度结果](#)

[Ernie INT8 模型在 Intel\(R\) Xeon\(R\) Gold 6271 上单样本耗时](#)

ARM CPU 上部署量化模型

Paddle Lite 可以在 ARM CPU 上部署 PaddleSlim 动态离线量化方法、静态离线量化方法和量化训练方法产出的量化模型。

Paddle Lite 部署量化模型的方法和常规非量化模型完全相同，主要包括使用 opt 工具进行模型优化、执行预测。

Paddle Lite 的详细说明，请参考[文档](#)。

Paddle Lite 部署动态离线量化方法产出的量化模型，请参考[文档](#)。

Paddle Lite 部署静态离线量化方法产出的量化模型，请参考[文档](#)。

Paddle Lite 部署量化训练方法产出的量化模型，请参考[文档](#)。

模型量化前后性能对比

2.6.3 模型性能分析

Paddle Profiler 是飞桨框架自带的低开销性能分析器，可以对模型运行过程的性能数据进行收集、统计和展示。性能分析器提供的数据可以帮助定位模型的瓶颈，识别造成程序运行时间过长或者 GPU 利用率低的原因，从而寻求优化方案来获得性能的提升。

在这篇文档中，主要介绍如何使用 Profiler 工具来调试程序性能，以及阐述当前提供的所有功能特性。主要内容如下：

- 使用 [Profiler](#) 工具调试程序性能
- 功能特性
- 更多细节

一、使用 Profiler 工具调试程序性能

在模型性能分析中，通常采用如下四个步骤：

- 获取模型正常运行时的 ips(iterations per second, 每秒的迭代次数)，给出 baseline 数据。
- 开启性能分析器，定位性能瓶颈点。
- 优化程序，检查优化效果。
- 获取优化后模型正常运行时的 ips，和 baseline 比较，计算真实的提升幅度。

下面是 Paddle 的应用实践教学中关于使用神经网络对 cifar10 进行分类的示例代码，里面加上了启动性能分析的代码。通过这个比较简单的示例，来看性能分析工具是如何通过上述四个步骤在调试程序性能中发挥作用。

```
def train(model):
    print('start training ... ')
    # turn into training mode
    model.train()

    opt = paddle.optimizer.Adam(learning_rate=learning_rate,
                               parameters=model.parameters())

    train_loader = paddle.io.DataLoader(cifar10_train,
                                         shuffle=True,
                                         batch_size=batch_size)

    valid_loader = paddle.io.DataLoader(cifar10_test, batch_size=batch_size)

    # 创建性能分析器相关的代码
    def my_on_trace_ready(prof): # 定义回调函数，性能分析器结束采集数据时会被调用
        callback = profiler.export_chrome_tracing('./profiler_demo') #_
    ↪创建导出性能数据到profiler_demo文件夹的回调函数
        callback(prof) # 执行该导出函数
        prof.summary(sorted_by=profiler.SortedKeys.GPUTotal) #_
    ↪打印表单，按GPUTotal排序表单项

    p = profiler.Profiler(scheduler = [3,14], on_trace_ready=my_on_trace_ready, timer_
    ↪only=True) # 初始化Profiler对象

    p.start() # 性能分析器进入第0个step

    for epoch in range(epoch_num):
        for batch_id, data in enumerate(train_loader()):
            x_data = data[0]
            y_data = paddle.to_tensor(data[1])
```

(下页继续)

(续上页)

```
y_data = paddle.unsqueeze(y_data, 1)

logits = model(x_data)
loss = F.cross_entropy(logits, y_data)

if batch_id % 1000 == 0:
    print("epoch: {}, batch_id: {}, loss is: {}".format(epoch, batch_id, loss.numpy()))
    loss.backward()
    opt.step()
    opt.clear_grad()

    p.step() # 指示性能分析器进入下一个step
    if batch_id == 19:
        p.stop() # 关闭性能分析器
        exit() # 做性能分析时，可以将程序提前退出

# evaluate model after one epoch
model.eval()
accuracies = []
losses = []
for batch_id, data in enumerate(valid_loader()):
    x_data = data[0]
    y_data = paddle.to_tensor(data[1])
    y_data = paddle.unsqueeze(y_data, 1)

    logits = model(x_data)
    loss = F.cross_entropy(logits, y_data)
    acc = paddle.metric.accuracy(logits, y_data)
    accuracies.append(acc.numpy())
    losses.append(loss.numpy())

    avg_acc, avg_loss = np.mean(accuracies), np.mean(losses)
    print("[validation] accuracy/loss: {} / {}".format(avg_acc, avg_loss))
    val_acc_history.append(avg_acc)
    val_loss_history.append(avg_loss)
model.train()
```

1. 获取性能调试前模型正常运行的 ips

上述程序在创建 Profiler 时候，timer_only 设置的值为 True，此时将只开启 benchmark 功能，不开启性能分析器，程序输出模型正常运行时的 benchmark 信息如下

=====Perf=====					
↳ Summary=====					
Reader Ratio: 53.514%					
Time Unit: s, IPS Unit: steps/s					
		avg		max	
reader_cost	0.01367		0.01407		0.01310
batch_cost	0.02555		0.02381		0.02220
ips	39.13907		45.03588		41.99930

其中 Reader Ratio 表示数据读取部分占训练 batch 迭代过程的时间占比，reader_cost 代表数据读取时间，batch_cost 代表 batch 迭代的时间，ips 表示每秒能迭代多少次，即跑多少个 batch。可以看到，此时的 ips 为 39.1，可将这个值作为优化对比的 baseline。

2. 开启性能分析器，定位性能瓶颈点

修改程序，将 Profiler 的 timer_only 参数设置为 False，此时代表不只开启 benchmark 功能，还将开启性能分析器，进行详细的性能分析。

```
p = profiler.Profiler(scheduler = [3, 14], on_trace_ready=my_on_trace_ready, timer_
↳only=False)
```

性能分析器会收集程序在第 3 到 14 次（不包括 14）训练迭代过程中的性能数据，并在 profiler_demo 文件夹中输出一个 json 格式的文件，用于展示程序执行过程的 timeline，可通过 chrome 浏览器的chrome://tracing插件打开这个文件进行查看。

性能分析器还会直接在终端打印统计表单（建议重定向到文件中查看），查看程序输出的 Model Summary 表单

-----Model Summary-----					

Time unit: ms					
Name	Calls	CPU Total / Avg / Max / Min / Ratio(%)	GPU Total / Avg /	Memory Total / Avg /	Network Total / Avg /
ProfileStep	11	294.53 / 26.78 / 35.28 / 24.56 / 100.00	13.22 / 1.20 / 1.	20 / 1.20 / 100.00	20 / 1.20 / 100.00

(下页继续)

(续上页)

Dataloader	11	141.49 / 12.86 / 17.53 / 10.34 / 48.04 → / 0.00 / 0.00	0.00 / 0.00 / 0.00
Forward	11	51.41 / 4.67 / 6.18 / 3.93 / 17.45 → / 0.35 / 29.50	3.92 / 0.36 / 0.36
Backward	11	21.23 / 1.93 / 2.61 / 1.70 / 7.21 → / 0.74 / 61.51	8.14 / 0.74 / 0.74
Optimization	11	34.74 / 3.16 / 3.65 / 2.41 / 11.79 → / 0.06 / 5.03	0.67 / 0.06 / 0.06
Others	-	45.66 / - / - / - / 15.50 → 3.96	0.53 / - / - / - /
<hr/>			
<hr/>			

其中 `ProfileStep` 表示训练 batch 的迭代 step 过程，对应代码中每两次调用 `p.step()` 的间隔时间；`Dataloader` 表示数据读取的时间，即 `for batch_id, data in enumerate(train_loader())` 的执行时间；`Forward` 表示模型前向的时间，即 `logits = model(x_data)` 的执行时间，`Backward` 表示反向传播的时间，即 `loss.backward()` 的执行时间；`Optimization` 表示优化器的时间，即 `opt.step()` 的执行时间。通过 `timeline` 可以看到，`Dataloader` 占了执行过程的很大比重，Model Summary 显示其甚至接近了 50%。分析程序发现，这是由于模型本身比较简单，需要的计算量小，再加上 `Dataloader` 准备数据时只用了单进程来读取，使得程序读取数据时和执行计算时没有并行操作，导致 `Dataloader` 占比过大。

3. 优化程序，检查优化效果

识别到了问题产生的原因，对程序继续做如下修改，将 `Dataloader` 的 `num_workers` 设置为 4，使得能有多个进程并行读取数据。

```
train_loader = paddle.io.DataLoader(cifar10_train,
                                    shuffle=True,
                                    batch_size=batch_size,
                                    num_workers=4)
```

重新对程序进行性能分析，新的 `timeline` 和 Model Summary 如下所示

Model Summary					
<hr/>					
<hr/>					
Name	Calls	CPU Total / Avg / Max / Min / Ratio(%)	GPU Total / Avg /		
→Max / Min / Ratio(%)					
<hr/>					
<hr/>					

(下页继续)

(续上页)

ProfileStep	11	90.94 / 8.27 / 11.82 / 7.85 / 100.00 → 22 / 1.19 / 100.00	13.27 / 1.21 / 1.
Dataloader	11	1.82 / 0.17 / 0.67 / 0.11 / 2.00 → / 0.00 / 0.00	0.00 / 0.00 / 0.00
Forward	11	29.58 / 2.69 / 3.53 / 2.52 / 32.52 → / 0.34 / 30.67	3.82 / 0.35 / 0.35
Backward	11	15.21 / 1.38 / 1.95 / 1.31 / 16.72 → / 0.74 / 60.71	8.30 / 0.75 / 0.77
Optimization	11	17.55 / 1.60 / 1.92 / 1.55 / 19.30 → / 0.06 / 4.82	0.66 / 0.06 / 0.06
Others	-	26.79 / - / - / - / 29.46 → 3.80	0.52 / - / - / - /
<hr/>			
<hr/>			

可以看到，从 Dataloader 中取数据的时间大大减少，变成了平均只占一个 step 的 2%，并且平均一个 step 所需要的时间也相应减少了。

4. 获取优化后模型正常运行的 ips，确定真实提升幅度

重新将 timer_only 设置的值为 True，获取优化后模型正常运行时的 benchmark 信息

=====Perf=====																									
→ Summary=====																									
Reader Ratio: 1.653%																									
Time Unit: s, IPS Unit: steps/s																									
<table border="1"> <thead> <tr> <th></th><th>avg</th><th>max</th><th>min</th><th></th></tr> </thead> <tbody> <tr> <td>reader_cost</td><td>0.00011</td><td>0.00024</td><td>0.00009</td><td></td></tr> <tr> <td>batch_cost</td><td>0.00660</td><td>0.00629</td><td>0.00587</td><td></td></tr> <tr> <td>ips</td><td>151.45498</td><td>170.28927</td><td>159.06308</td><td></td></tr> </tbody> </table>							avg	max	min		reader_cost	0.00011	0.00024	0.00009		batch_cost	0.00660	0.00629	0.00587		ips	151.45498	170.28927	159.06308	
	avg	max	min																						
reader_cost	0.00011	0.00024	0.00009																						
batch_cost	0.00660	0.00629	0.00587																						
ips	151.45498	170.28927	159.06308																						

此时 ips 的值变成了 151.5，相比优化前的 baseline 39.1，模型真实性能提升了 287%。

Note 由于 Profiler 开启的时候，收集性能数据本身也会造成程序性能的开销，因此正常跑程序时请不要开启性能分析器，性能分析器只作为调试程序性能时使用。如果想获得程序正常运行时候的 benchmark 信息（如 ips），可以像示例一样将 Profiler 的 timer_only 参数设置为 True，此时不会进行详尽的性能数据收集，几乎不影响程序正常运行的性能，所获得的 benchmark 信息也趋于真实。此外，benchmark 信息计算的数据范围是从调用 Profiler 的 start 方法开始，到调用 stop 方法结束这个过程的数据。而 Timeline 和性能数据的统计表单的数据范围是所指定的采集区间，如这个例子中的第 3 到 14 次迭代，这会导致开启性能分析器时统计表单和 benchmark 信息输出的值不同（如统计到的 Dataloader 的时间占比）。此外，当 benchmark 统计的范围和性能分析器统计的范围不同时，由于 benchmark 统计的是平均时间，如果 benchmark 统计的范围覆盖了性能分析器开启的范围，也覆盖了关闭性能调试时的正常执行的范围，此时 benchmark 的值没有意义，因此开启性能分析器时请以性能分析器输出的统计表单为参考，这也是为何上面示例里在开启性能分析器时没贴 benchmark

信息的原因。

二、功能特性

当前 Profiler 提供 Timeline、统计表单、benchmark 信息共三个方面的展示功能。

1. Timeline 展示

对于采集的性能数据，导出为 chrome tracing timeline 格式的文件后，可以进行可视化分析。当前，所采用的可视化工具为 chrome 浏览器里的，可以按照如下方式进行查看

1. 查看 CPU 和 GPU 在不同线程或 stream 下的事件发生的时间线。将同一线程下所记录的数据分为 Python 层和 C++ 层，以便根据需要进行折叠和展开。对于有名字的线程，标注线程名字。
2. 所展示的事件名字上标注事件所持续的时间，点击具体的事件，可在下方的说明栏中看到更详细的事
件信息。通过按键'w'、's' 可以进行放大和缩小，通过'a'、'd' 可以进行左移和右移。
3. 对于 GPU 上的事件，可以通过点击下方的'launch' 链接查看所发起它的 CPU 上的事件。

2. 统计表单展示

统计表单负责对采集到的数据(Event)从多个不同的角度进行解读，也可以理解为对 timeline 进行一些量化的指标计算。目前提供 Device Summary、Overview Summary、Model Summary、Distributed Summary、Operator Summary、Kernel Summary、Memory Manipulation Summary 和 UserDefined Summary 的统计表单，每个表单从不同的角度进行统计计算。每个表单的统计内容简要叙述如下：

- Device Summary

-----Device Summary-----	
Device	Utilization (%)
CPU (Process)	77.13
CPU (System)	25.99
GPU2	55.50

Note:

```

CPU(Process) Utilization = Current process CPU time over all cpu cores / elapsed_time, so max utilization can be reached 100% * number of cpu cores.
CPU(System) Utilization = All processes CPU time over all cpu cores(busy time) / (busy time + idle time).
GPU Utilization = Current process GPU time / elapsed time.

```

DeviceSummary 提供 CPU 和 GPU 的平均利用率信息。其中

- CPU(Process): 指的是进程的 cpu 平均利用率，算的是从 Profiler 开始记录数据到结束这一段过程，进程所利用到的 **cpu core** 的总时间与该段时间的占比。因此如果是多核的情况，对于进程来说 cpu 平均利用率是有可能超过 100% 的，因为同时用到的多个 core 的时间进行了累加。
- CPU(System): 指的是整个系统的 cpu 平均利用率，算的是从 Profiler 开始记录数据到结束这一段过程，整个系统所有进程利用到的 **cpu core** 总时间与该段时间乘以 **cpu core** 的数量的占比。可以当成是从 cpu 的视角来算的利用率。
- GPU: 指的是进程的 gpu 平均利用率，算的是从 Profiler 开始记录数据到结束这一段过程，进程在 gpu 上所调用的 **kernel** 的执行时间与 该段时间的占比。

- Overview Summary

-----Overview Summary-----			
Time unit: ms			
Event Type	Calls	CPU Time	
Ratio (%)			
ProfileStep	8	4945.15	100.00
CudaRuntime	28336	2435.63	49.25
UserDefined	486	2280.54	46.12
Dataloader	8	1819.15	36.79
Forward	8	1282.64	25.94
Operator	8056	1244.41	25.16
OperatorInner	21880	374.18	7.57
Backward	8	160.43	3.24
Optimization	8	102.34	2.07

Calls			
Ratio (%)			

(下页继续)

(续上页)

Kernel	13688	2744.61	
↳ 55.50			
Memcpy	496	29.82	
↳ 0.60			
Memset	104	0.12	
↳ 0.00			
Communication	784	257.23	
↳ 5.20			
<hr/>			
<hr/>			
Note:			
In this table, We sum up all collected events in terms of event type.			
The time of events collected on host are presented as CPU Time, and as GPU Time if on device.			
Events with different types may overlap or inclusion, e.g. Operator includes OperatorInner, so the sum of ratios is not 100%.			
The time of events in the same type with overlap will not calculate twice, and all time is summed after merged.			
Example:			
Thread 1:			
Operator: _____ _____			
Thread 2:			
Operator: _____ __			
After merged:			
Result: _____ _____			
<hr/>			
<hr/>			

Overview Summary 用于展示每种类型的 Event 一共分别消耗了多少时间，对于多线程或多 stream 下，如果同一类型的 Event 有重叠的时间段，采取取并集操作，不对重叠的时间进行重复计算。

- Model Summary

-----Model Summary-----					
<hr/>					
Time unit: ms					
Name	Calls	CPU Total / Avg / Max / Min / Ratio(%)	GPU Total / Avg / Max / Min / Ratio(%)		
ProfileStep	8	4945.15 / 618.14 / 839.15 / 386.34 / 100.00	2790.80 / 348.85 / 372.39 / 344.60 / 100.00		

(下页继续)

(续上页)

Dataloader	8	1819.15 / 227.39 / 451.69 / 0.32 / 36.79	0.00 / 0.00
↳ / 0.00 / 0.00 / 0.00			
Forward	8	1282.64 / 160.33 / 161.49 / 159.19 / 25.94	1007.64 /
↳ 125.96 / 126.13 / 125.58 / 35.90			
Backward	8	160.43 / 20.05 / 21.00 / 19.21 / 3.24	1762.11 /
↳ 220.26 / 243.83 / 216.05 / 62.49			
Optimization	8	102.34 / 12.79 / 13.42 / 12.47 / 2.07	17.03 / 2.
↳ 13 / 2.13 / 2.13 / 0.60			
Others	-	1580.59 / - / - / - / 31.96	28.22 / - /
↳ - / - / 1.00			
<hr/>			
<hr/>			

Model Summary 用于展示模型训练或者推理过程中，dataloader、forward、backward、optimization 所消耗的时间。其中 GPU Time 对应着在该段过程内所发起的 GPU 侧活动的时间。

- Distributed Summary

-----Distribution Summary-----		
Time unit: ms		
Name	Total Time	Ratio (%)
ProfileStep	4945.15	100.00
Communication	257.23	5.20
Computation	2526.52	51.09
Overlap	39.13	0.79

Distribution Summary 用于展示分布式训练中通信 (Communication)、计算 (Computation) 以及这两者 Overlap 的时间。

Communication: 所有和通信有关活动的时间，包括和分布式相关的算子 (op) 以及 gpu 上的 kernel 的时间等。

Computation: 即是所有 kernel 在 GPU 上的执行时间，但是去除了和通信相关的 kernel 的时间。

Overlap: Communication 和 Computation 的重叠时间

- Operator Summary

(由于原始表单较长，这里截取一部分进行展示)
-----Operator Summary-----
↳
Time unit: ms

(下页继续)

(续上页)

Name	Calls	CPU Total / Avg
↳Max / Min / Ratio(%) GPU Total / Avg / Max / Min / Ratio(%)		
<hr/>		
<hr/>		
----- Thread: All threads -----		
-----merged-----		
conv2d_grad_grad_node	296	53.70 / 0.18 / 0.40
↳ 0.14 / 4.34 679.11 / 2.29 / 5.75 / 0.24 / 24.11		
conv2d_grad::infer_shape	296	0.44 / 0.00 / 0.00
↳ 0.00 / 0.81 0.00 / 0.00 / 0.00 / 0.00 / 0.00		
conv2d_grad::compute	296	44.09 / 0.15 / 0.31
↳ 0.10 / 82.10 644.39 / 2.18 / 5.75 / 0.24 / 94.89		
cudnn::maxwell::gemm::computeWgradOffsetsKern...	224	- / - / - / - / -
↳ 0.50 / 0.00 / 0.00 / 0.00 / 0.08		
void scalePackedTensor_kernel<float, float>(c...	224	- / - / - / - / -
↳ 0.79 / 0.00 / 0.01 / 0.00 / 0.12		
cudnn::maxwell::gemm::computeBOffsetsKernel(c...	464	- / - / - / - / -
↳ 0.95 / 0.00 / 0.01 / 0.00 / 0.15		
maxwell_scudnn_128x32_stridedB_splitK_large_nn	8	- / - / - / - / -
↳ 15.70 / 1.96 / 1.97 / 1.96 / 2.44		
cudnn::maxwell::gemm::computeOffsetsKernel(cu...	240	- / - / - / - / -
↳ 0.54 / 0.00 / 0.00 / 0.00 / 0.08		
maxwell_scudnn_128x32_stridedB_interior_nn	8	- / - / - / - / -
↳ 9.53 / 1.19 / 1.19 / 1.19 / 1.48		
maxwell_scudnn_128x64_stridedB_splitK_interio...	8	- / - / - / - / -
↳ 28.67 / 3.58 / 3.59 / 3.58 / 4.45		
maxwell_scudnn_128x64_stridedB_interior_nn	8	- / - / - / - / -
↳ 5.53 / 0.69 / 0.70 / 0.69 / 0.86		
maxwell_scudnn_128x128_stridedB_splitK_interi...	184	- / - / - / - / -
↳ 167.03 / 0.91 / 2.28 / 0.19 / 25.92		
maxwell_scudnn_128x128_stridedB_interior_nn	200	- / - / - / - / -
↳ 105.10 / 0.53 / 0.97 / 0.09 / 16.31		
MEMSET	104	- / - / - / - / -
↳ 0.12 / 0.00 / 0.00 / 0.00 / 0.02		
maxwell_scudnn_128x128_stridedB_small_nn	24	- / - / - / - / -
↳ 87.58 / 3.65 / 4.00 / 3.53 / 13.59		
void cudnn::winograd_nonfused::winogradWgradD...	72	- / - / - / - / -
↳ 15.66 / 0.22 / 0.36 / 0.09 / 2.43		
void cudnn::winograd_nonfused::winogradWgradD...	72	- / - / - / - / -
↳ 31.64 / 0.44 / 0.75 / 0.19 / 4.91		
maxwell_sgemm_128x64_nt	72	- / - / - / - / -
↳ 62.03 / 0.86 / 1.09 / 0.75 / 9.63		
void cudnn::winograd_nonfused::winogradWgradO...	72	- / - / - / - / -
↳ 14.45 / 0.20 / 0.49 / 0.04 / 2.24		

(下页继续)

(续上页)

void cudnn:::winograd:::generateWinogradTilesKe...	48	- / - / - / - / -
↳ 1.78 / 0.04 / 0.06 / 0.02 / 0.28		
maxwell_scudnn_winograd_128x128_ldg1_ldg4_til...	24	- / - / - / - / -
↳ 45.94 / 1.91 / 1.93 / 1.90 / 7.13		
maxwell_scudnn_winograd_128x128_ldg1_ldg4_til...	24	- / - / - / - / -
↳ 40.93 / 1.71 / 1.72 / 1.69 / 6.35		
maxwell_scudnn_128x32_stridedB_splitK_interio...	24	- / - / - / - / -
↳ 9.91 / 0.41 / 0.77 / 0.15 / 1.54		
GpuMemcpyAsync:CPU->GPU	64	0.68 / 0.01 / 0.02 /
↳ 0.01 / 1.27 0.09 / 0.00 / 0.00 / 0.00 / 0.01		
MEMCPY_HtoD	64	- / - / - / - / -
↳ 0.09 / 0.00 / 0.00 / 0.00 / 100.00		
void phi::funcs::ConcatKernel_<float>(float con...	16	- / - / - / - / -
↳ 2.84 / 0.18 / 0.36 / 0.06 / 0.42		
void phi::funcs::ForRangeElemwiseOp<paddle::imp...	16	- / - / - / - / -
↳ 1.33 / 0.08 / 0.16 / 0.01 / 0.20		
ncclAllReduceRingLLKernel_sum_f32(ncclColl)	16	- / - / - / - / -
↳ 26.35 / 1.65 / 3.14 / 0.20 / 3.88		
void phi::funcs::SplitKernel_<float>(float cons...	16	- / - / - / - / -
↳ 2.49 / 0.16 / 0.37 / 0.06 / 0.37		
void axpy_kernel_val<float, float>(cublasAxpyPa...	16	- / - / - / - / -
↳ 1.63 / 0.10 / 0.14 / 0.07 / 0.24		
sync_batch_norm_grad grad_node	376	37.90 / 0.10 / 0.31
↳ 0.08 / 3.07 670.62 / 1.78 / 39.29 / 0.13 / 23.81		
sync_batch_norm_grad::infer_shape	376	1.60 / 0.00 / 0.01 /
↳ 0.00 / 4.22 0.00 / 0.00 / 0.00 / 0.00 / 0.00		
sync_batch_norm_grad::compute	376	23.26 / 0.06 / 0.10
↳ 0.06 / 61.37 555.96 / 1.48 / 39.29 / 0.13 / 82.90		
void paddle::operators::KeBackwardLocalStats<...	376	- / - / - / - / -
↳ 129.62 / 0.34 / 1.83 / 0.04 / 23.32		
ncclAllReduceRingLLKernel_sum_f32(ncclColl)	376	- / - / - / - / -
↳ 128.00 / 0.34 / 37.70 / 0.01 / 23.02		
void paddle::operators::KeBNBackwardScaleBias...	376	- / - / - / - / -
↳ 126.37 / 0.34 / 1.84 / 0.03 / 22.73		
void paddle::operators::KeBNBackwardData<floa...	376	- / - / - / - / -
↳ 171.97 / 0.46 / 2.58 / 0.04 / 30.93		
GpuMemcpyAsync:CPU->GPU	64	0.71 / 0.01 / 0.02 /
↳ 0.01 / 1.88 0.08 / 0.00 / 0.00 / 0.00 / 0.01		
MEMCPY_HtoD	64	- / - / - / - / -
↳ 0.08 / 0.00 / 0.00 / 0.00 / 100.00		
void phi::funcs::ConcatKernel_<float>(float con...	16	- / - / - / - / -
↳ 6.40 / 0.40 / 0.53 / 0.34 / 0.95		
void phi::funcs::ForRangeElemwiseOp<paddle::imp...	16	- / - / - / - / -
↳ 6.23 / 0.39 / 0.56 / 0.27 / 0.93		

(下页继续)

(续上页)

ncclAllReduceRingLLKernel_sum_f32(ncclColl)	16	- / - / - / - / -
↳	95.02 / 5.94 / 7.56 / 4.75 / 14.17	
void phi::funcs::SplitKernel_<float>(float cons...	16	- / - / - / - / -
↳	6.93 / 0.43 / 0.76 / 0.34 / 1.03	

Operator Summary 用于展示框架中算子 (op) 的执行信息。对于每一个 Op，可以通过打印表单时候的 `op_detail` 选项控制是否打印出 Op 执行过程里面的子过程。同时展示每个子过程中的 GPU 上的活动，且子过程的活动算时间占比时以上层的时间为总时间。

- Kernel Summary

```
(由于原始表单较长，这里截取一部分进行展示)
-----Kernel Summary-----
Time unit: ms
-----
Name
↳ Calls GPU Total / Avg / Max / Min / Ratio(%)
-----
void paddle::operators::KeNormAffine<float, (paddle::experimental::DataLayout)2>
↳ 376 362.11 / 0.96 / 5.43 / 0.09 / 12.97
ncclAllReduceRingLLKernel_sum_f32(ncclColl)
↳ 784 257.23 / 0.33 / 37.70 / 0.01 / 9.22
maxwell_scudnn_winograd_128x128_ldg1_ldg4_tile418n_nt
↳ 72 176.84 / 2.46 / 3.35 / 1.90 / 6.34
void paddle::operators::KeBNBackwardData<float,
(paddle::experimental::DataLayout)2> 376 171.97 / 0.46 / 2.58 / 0.04
/ 6.16
maxwell_scudnn_128x128_stridedB_splitK_interior_nn
↳ 184 167.03 / 0.91 / 2.28 / 0.19 / 5.99
void paddle::operators::KeBackwardLocalStats<float, 256,
(paddle::experimental::DataLay... 376 129.62 / 0.34 / 1.83 / 0.04 / 4.64
void paddle::operators::KeBNBackwardScaleBias<float, 256,
(paddle::experimental::DataLa... 376 126.37 / 0.34 / 1.84 / 0.03 / 4.53
void phi::funcs::VectorizedElementwiseKernel<float,
phi::funcs::CudaReluGradFunctor<flo... 216 115.61 / 0.54 / 2.31 / 0.07 / 4.
14
void paddle::operators::math::KernelDepthwiseConvFilterGradSp<float, 1, 1, 3,
(paddle::... 72 113.87 / 1.58 / 2.04 / 1.36 / 4.08
maxwell_scudnn_128x128_stridedB_interior_nn
↳ 200 105.10 / 0.53 / 0.97 / 0.09 / 3.77
maxwell_scudnn_128x128_relu_interior_nn
↳ 184 103.17 / 0.56 / 0.98 / 0.12 / 3.70
```

(下页继续)

(续上页)

```
maxwell_scudnn_winograd_128x128_ldg1_ldg4_tile228n_nt
↳      48      90.87 / 1.89 / 2.09 / 1.69 / 3.26
maxwell_scudnn_128x128_stridedB_small_nn
↳      24      87.58 / 3.65 / 4.00 / 3.53 / 3.14
```

Kernel Summary 用于展示在 GPU 执行的 kernel 的信息。

- Memory Manipulation Summary

```
-----Memory Manipulation Summary-----
-----
Time unit: ms
-----
Name          Calls   CPU Total / Avg / Max / Min / Ratio(%)
↳ GPU Total / Avg / Max / Min / Ratio(%)
-----
GpuMemcpySync:GPU->CPU           48      1519.87 / 31.66 / 213.82 / 0.02 / 30.
↳ 73    0.07 / 0.00 / 0.00 / 0.00 / 0.00
GpuMemcpyAsync:CPU->GPU          216      2.85 / 0.01 / 0.04 / 0.01 / 0.06
↳ 0.29 / 0.00 / 0.00 / 0.00 / 0.01
GpuMemcpyAsync(same_gpu):GPU->GPU 168      3.61 / 0.02 / 0.05 / 0.01 / 0.07
↳ 0.33 / 0.00 / 0.01 / 0.00 / 0.01
GpuMemcpySync:CUDAPinned->GPU    40       713.89 / 17.85 / 85.79 / 0.04 / 14.44
↳ 29.11 / 0.73 / 3.02 / 0.00 / 1.03
BufferedReader:MemoryCopy         6        40.17 / 6.69 / 7.62 / 5.87 / 0.81
↳ 0.00 / 0.00 / 0.00 / 0.00 / 0.00
-----
```

Memory Manipulation Summary 用于展示框架中调用内存操作所花费的时间。

- UserDefined Summary

```
-----UserDefined Summary-----
-----
Time unit: ms
-----
Name          Calls   CPU Total / Avg / Max / Min / Ratio(%)
↳ GPU Total / Avg / Max / Min / Ratio(%)
-----
-----Thread: All threads merged-----
↳
```

(下页继续)

(续上页)

MyRecord	8	0.15 / 0.02 / 0.02 / 0.02 / 0.00	0.00 / 0.00 / 0.00
↳	/ 0.00	/ 0.00	↳
-----	-----	-----	-----

UserDefined Summary 用于展示用户自定义记录的 Event 所花费的时间。

3. Benchmark 信息

benckmark 信息用于展示模型的吞吐量以及时间开销。

=====Perf=====					
↳Summary=====					
Reader Ratio: 0.989%					
Time Unit: s, IPS Unit: steps/s					
		avg		max	
reader_cost		0.00010		0.00011	
batch_cost		0.00986		0.00798	
ips		101.41524		127.25977	
					125.29320

其中 ReaderRatio 表示数据读取部分占 batch 迭代过程的时间占比, reader_cost 代表数据读取时间, batch_cost 代表 batch 迭代的时间, ips 表示每秒能迭代多少次, 即跑多少个 batch。

三、更多细节

关于 paddle.profiler 模块更详细的使用说明, 可以参考[API 文档](#)。目前 Paddle 的性能分析工具主要还只提供时间方面的分析, 之后会提供更多信息的收集来辅助做更全面的分析, 如提供显存分析来监控显存泄漏问题。此外, Paddle 的可视化工具 VisualDL 正在对 Profiler 的数据展示进行开发, 敬请期待。

2.7 模型迁移

您可以通过下面的内容, 了解如何迁移模型到飞桨 2.X:

- 升级指南: 介绍飞桨框架 2.0 的主要变化和如何升级到最新版飞桨。
- 版本迁移工具: 介绍飞桨框架版本转换工具的使用。
- 兼容载入旧格式模型: 介绍飞桨框架如何在 2.x 版本加载 1.x 版本保存的模型。
- Paddle API 映射表 : 说明 Paddle 1.8 版本与 Paddle 2.0 API 对应关系。
- PyTorch API 映射表 : 说明 PyTorch 1.8 版本与 Paddle 2.0 API 对应关系。

2.7.1 升级指南

升级概要

飞桨 2.0 版本，相对 1.8 版本有重大升级，涉及开发方面的重要变化如下：

- 动态图功能完善，动态图模式下数据表示概念为 `Tensor`，推荐使用动态图模式；
- API 目录体系调整，API 的命名和别名进行了统一规范化，虽然兼容老版 API，但请使用新 API 体系开发；
- 数据处理、组网方式、模型训练、多卡启动、模型保存和推理等开发流程都有了对应优化，请对应查看说明；

以上变化请仔细阅读本指南。对于已有模型的升级，飞桨还提供了 2.0 转换工具（见附录）提供更自动化的辅助。其他一些功能增加方面诸如动态图对量化训练、混合精度的支持、动静转换等方面不在本指南列出，具体可查看[Release Note](#)或对应文档。

一、动态图

推荐优先使用动态图模式

飞桨 2.0 版本将会把动态图作为默认模式（如果还想使用静态图，可通过调用 `paddle.enable_static` 切换）。

```
import paddle
```

使用 `Tensor` 概念表示数据

静态图模式下，由于组网时使用的数据不能实时访问，Paddle 用 `Variable` 来表示数据。动态图下，从直观性等角度考虑，将数据表示概念统一为 `Tensor`。动态图下 `Tensor` 的创建主要有两种方法：

1. 通过调用 `paddle.to_tensor` 函数，将 python scalar/list，或者 `numpy.ndarray` 数据转换为 Paddle 的 `Tensor`。具体使用方法，请查看官网的 API 文档。

```
import paddle
import numpy as np

paddle.to_tensor(1)
paddle.to_tensor((1.1, 2.2))
paddle.to_tensor(np.random.randn(3, 4))
```

1. 通过调用 `paddle.zeros`, `paddle.ones`, `paddle.full`, `paddle.arange`, `paddle.rand`, `paddle.randn`, `paddle.randint`, `paddle.normal`, `paddle.uniform` 等函数，创建并返回 `Tensor`。

二、API

API 目录结构

为了 API 组织更加简洁和清晰，将原来 paddle.fluid.xxx 的目录体系全新升级为 paddle.xxx，并对子目录的组织进行了系统的条理化优化。同时还增加了高层 API，可以高低搭配使用。paddle.fluid 目录下暂时保留了 1.8 版本 API，主要是兼容性考虑，未来会被删除。基于 2.0 的开发任务，请使用 paddle 目录下的 API，不要再使用 paddle.fluid 目录下的 API。如果发现 Paddle 目录下有 API 缺失的情况，推荐使用基础 API 进行组合实现；你也可以通过在 [github](#) 上提 issue 的方式反馈。

2.0 版本的 API 整体目录结构如下：

API 别名规则

- 为了方便使用，API 会在不同的路径下建立别名：
 - 所有 device, framework, tensor 目录下的 API，均在 paddle 根目录建立别名；除少数特殊 API 外，其他 API 在 paddle 根目录下均没有别名。
 - paddle.nn 目录下除 functional 目录以外的所有 API，在 paddle.nn 目录下均有别名；functional 目录中的 API，在 paddle.nn 目录下均没有别名。
- 推荐优先使用较短的路径的别名，比如 paddle.add -> paddle.tensor.add，推荐优先使用 paddle.add
- 以下为一些特殊的别名关系，推荐使用左边的 API 名称：
 - paddle.tanh -> paddle.tensor.tanh -> paddle.nn.functional.tanh
 - paddle.remainder -> paddle.mod -> paddle.floor_mod
 - paddle.rand -> paddle.uniform
 - paddle.randn -> paddle.standard_normal
 - Layer.set_state_dict -> Layer.set_dict

常用 API 名称变化

- 加、减、乘、除使用全称，不使用简称
- 对于当前逐元素操作，不加 elementwise 前缀
- 对于按照某一轴操作，不加 reduce 前缀
- Conv, Pool, Dropout, BatchNorm, Pad 组网类 API 根据输入数据类型增加 1D, 2D, 3D 后缀

三、开发流程

数据处理

数据处理推荐使用 paddle.io 目录下的 **Dataset**, **Sampler**, **BatchSampler**, **DataLoader** 接口, 不推荐 reader 类接口。一些常用的数据集已经在 paddle.vision.datasets 和 paddle.text.datasets 目录实现, 具体参考 API 文档。

```
from paddle.io import Dataset

class MyDataset(Dataset):
    """
    步骤一：继承paddle.io.Dataset类
    """

    def __init__(self, mode='train'):
        """
        步骤二：实现构造函数，定义数据读取方式，划分训练和测试数据集
        """
        super(MyDataset, self).__init__()

        if mode == 'train':
            self.data = [
                ['traindata1', 'label1'],
                ['traindata2', 'label2'],
                ['traindata3', 'label3'],
                ['traindata4', 'label4'],
            ]
        else:
            self.data = [
                ['testdata1', 'label1'],
                ['testdata2', 'label2'],
                ['testdata3', 'label3'],
                ['testdata4', 'label4'],
            ]

    def __getitem__(self, index):
        """
        步骤三：实现__getitem__
        ↪方法，定义指定index时如何获取数据，并返回单条数据（训练数据，对应的标签）
        """
        data = self.data[index][0]
        label = self.data[index][1]

        return data, label
```

(下页继续)

(续上页)

```
def __len__(self):
    """
    步骤四：实现__len__方法，返回数据集总数目
    """
    return len(self.data)

# 测试定义的数据集
train_dataset = MyDataset(mode='train')
val_dataset = MyDataset(mode='test')

print('=====train dataset=====')  
for data, label in train_dataset:  
    print(data, label)

print('=====evaluation dataset=====')  
for data, label in val_dataset:  
    print(data, label)
```

组网方式

Sequential 组网

针对顺序的线性网络结构可以直接使用 Sequential 来快速完成组网，可以减少类的定义等代码编写。

```
import paddle

# Sequential形式组网
mnist = paddle.nn.Sequential(
    paddle.nn.Flatten(),
    paddle.nn.Linear(784, 512),
    paddle.nn.ReLU(),
    paddle.nn.Dropout(0.2),
    paddle.nn.Linear(512, 10)
)
```

SubClass 组网

针对一些比较复杂的网络结构，就可以使用 Layer 子类定义的方式来进行模型代码编写，在 `__init__` 构造函数中进行组网 Layer 的声明，在 `forward` 中使用声明的 Layer 变量进行前向计算。子类组网方式也可以实现 sublayer 的复用，针对相同的 layer 可以在构造函数中一次性定义，在 ‘forward’ 中多次调用。

```
import paddle

# Layer类继承方式组网
class Mnist(paddle.nn.Layer):
    def __init__(self):
        super(Mnist, self).__init__()

        self.flatten = paddle.nn.Flatten()
        self.linear_1 = paddle.nn.Linear(784, 512)
        self.linear_2 = paddle.nn.Linear(512, 10)
        self.relu = paddle.nn.ReLU()
        self.dropout = paddle.nn.Dropout(0.2)

    def forward(self, inputs):
        y = self.flatten(inputs)
        y = self.linear_1(y)
        y = self.relu(y)
        y = self.dropout(y)
        y = self.linear_2(y)

    return y

mnist = Mnist()
```

模型训练

使用高层 API

增加了 `paddle.Model` 高层 API，大部分任务可以使用此 API 用于简化训练、评估、预测类代码开发。注意区别 Model 和 Net 概念，Net 是指继承 `paddle.nn.Layer` 的网络结构；而 Model 是指持有一个 Net 对象，同时指定损失函数、优化算法、评估指标的可训练、评估、预测的实例。具体参考高层 API 的代码示例。

```
import paddle
from paddle.vision.transforms import ToTensor

train_dataset = paddle.vision.datasets.MNIST(mode='train', transform=ToTensor())
test_dataset = paddle.vision.datasets.MNIST(mode='test', transform=ToTensor())
```

(下页继续)

(续上页)

```

lenet = paddle.vision.models.LeNet()

# Mnist 继承paddle.nn.Layer属于Net, model包含了训练功能
model = paddle.Model(lenet)

# 设置训练模型所需的optimizer, loss, metric
model.prepare(
    paddle.optimizer.Adam(learning_rate=0.001, parameters=model.parameters()),
    paddle.nn.CrossEntropyLoss(),
    paddle.metric.Accuracy()
)

# 启动训练
model.fit(train_dataset, epochs=2, batch_size=64, log_freq=200)

# 启动评估
model.evaluate(test_dataset, log_freq=20, batch_size=64)

```

使用基础 API

```

import paddle
from paddle.vision.transforms import ToTensor

train_dataset = paddle.vision.datasets.MNIST(mode='train', transform=ToTensor())
test_dataset = paddle.vision.datasets.MNIST(mode='test', transform=ToTensor())
lenet = paddle.vision.models.LeNet()
loss_fn = paddle.nn.CrossEntropyLoss()

# 加载训练集 batch_size 设为 64
train_loader = paddle.io.DataLoader(train_dataset, batch_size=64, shuffle=True)

def train():
    epochs = 2
    adam = paddle.optimizer.Adam(learning_rate=0.001, parameters=lenet.parameters())
    # 用Adam作为优化函数
    for epoch in range(epochs):
        for batch_id, data in enumerate(train_loader()):
            x_data = data[0]
            y_data = data[1]
            predicts = lenet(x_data)
            acc = paddle.metric.accuracy(predicts, y_data)
            loss = loss_fn(predicts, y_data)

```

(下页继续)

(续上页)

```
loss.backward()
if batch_id % 100 == 0:
    print("epoch: {}, batch_id: {}, loss is: {}, acc is: {}".format(epoch,
→ batch_id, loss.numpy(), acc.numpy()))
    adam.step()
    adam.clear_grad()

# 启动训练
train()
```

单机多卡启动

2.0 增加 paddle.distributed.spawn 函数来启动单机多卡训练，同时原有的 paddle.distributed.launch 的方式依然保留。

方式 1、launch 启动

高层 API 场景

当调用 paddle.Model 高层来实现训练时，想要启动单机多卡训练非常简单，代码不需要做任何修改，只需要在启动时增加一下参数-m paddle.distributed.launch。

```
# 单机单卡启动， 默认使用第0号卡
$ python train.py

# 单机多卡启动， 默认使用当前可见的所有卡
$ python -m paddle.distributed.launch train.py

# 单机多卡启动， 设置当前使用的第0号和第1号卡
$ python -m paddle.distributed.launch --selected_gpus='0,1' train.py

# 单机多卡启动， 设置当前使用第0号和第1号卡
$ export CUDA_VISIBLE_DEVICES=0,1
$ python -m paddle.distributed.launch train.py
```

基础 API 场景

如果使用基础 API 实现训练，想要启动单机多卡训练，需要对单机单卡的代码进行 3 处修改，具体如下：

```

import paddle
from paddle.vision.transforms import ToTensor

# 第1处改动，导入分布式训练所需要的包
import paddle.distributed as dist

train_dataset = paddle.vision.datasets.MNIST(mode='train', transform=ToTensor())
test_dataset = paddle.vision.datasets.MNIST(mode='test', transform=ToTensor())
lenet = paddle.vision.models.LeNet()
loss_fn = paddle.nn.CrossEntropyLoss()

# 加载训练集 batch_size 设为 64
train_loader = paddle.io.DataLoader(train_dataset, batch_size=64, shuffle=True)

def train(model):
    # 第2处改动，初始化并行环境
    dist.init_parallel_env()

    # 第3处改动，增加paddle.DataParallel封装
    lenet = paddle.DataParallel(model)
    epochs = 2
    adam = paddle.optimizer.Adam(learning_rate=0.001, parameters=lenet.parameters())
    # 用Adam作为优化函数
    for epoch in range(epochs):
        for batch_id, data in enumerate(train_loader()):
            x_data = data[0]
            y_data = data[1]
            predicts = lenet(x_data)
            acc = paddle.metric.accuracy(predicts, y_data)
            loss = loss_fn(predicts, y_data)
            loss.backward()
            if batch_id % 100 == 0:
                print("epoch: {}, batch_id: {}, loss is: {}, acc is: {}".format(epoch,
→ batch_id, loss.numpy(), acc.numpy()))
            adam.step()
            adam.clear_grad()

    # 启动训练
    train(lenet)

```

修改完后保存文件，然后使用跟高层 API 相同的启动方式即可

注意：单卡训练不支持调用 `init_parallel_env`，请使用以下几种方式进行分布式训练。

```
# 单机多卡启动， 默认使用当前可见的所有卡
$ python -m paddle.distributed.launch train.py

# 单机多卡启动， 设置当前使用的第0号和第1号卡
$ python -m paddle.distributed.launch --selected_gpus '0,1' train.py

# 单机多卡启动， 设置当前使用第0号和第1号卡
$ export CUDA_VISIBLE_DEVICES=0,1
$ python -m paddle.distributed.launch train.py
```

方式 2、spawn 启动

`launch` 方式启动训练，以文件为单位启动多进程，需要在启动时调用 `paddle.distributed.launch`，对于进程的管理要求较高。飞桨框架 2.0 版本增加了 `spawn` 启动方式，可以更好地控制进程，在日志打印、训练退出时更友好。使用示例如下：

```
from __future__ import print_function

import paddle
import paddle.nn as nn
import paddle.optimizer as opt
import paddle.distributed as dist

class LinearNet(nn.Layer):
    def __init__(self):
        super(LinearNet, self).__init__()
        self._linear1 = nn.Linear(10, 10)
        self._linear2 = nn.Linear(10, 1)

    def forward(self, x):
        return self._linear2(self._linear1(x))

def train(print_result=False):

    # 1. 初始化并行训练环境
    dist.init_parallel_env()

    # 2. 创建并行训练 Layer 和 Optimizer
    layer = LinearNet()
    dp_layer = paddle.DataParallel(layer)

    loss_fn = nn.MSELoss()
```

(下页继续)

(续上页)

```

adam = opt.Adam(
    learning_rate=0.001, parameters=dp_layer.parameters())

# 3. 运行网络
inputs = paddle.randn([10, 10], 'float32')
outputs = dp_layer(inputs)
labels = paddle.randn([10, 1], 'float32')
loss = loss_fn(outputs, labels)

if print_result is True:
    print("loss:", loss.numpy())

loss.backward()

adam.step()
adam.clear_grad()

# 使用方式1：仅传入训练函数
# 适用场景：训练函数不需要任何参数，并且需要使用所有当前可见的GPU设备并行训练
if __name__ == '__main__':
    dist.spawn(train)

# 使用方式2：传入训练函数和参数
# 适用场景：训练函数需要一些参数，并且需要使用所有当前可见的GPU设备并行训练
if __name__ == '__main__':
    dist.spawn(train, args=(True,))

# 使用方式3：传入训练函数、参数并指定并行进程数
# 适用场景：训练函数需要一些参数，并且仅需要使用部分可见的GPU设备并行训练，例如：
# 当前机器有8张GPU卡 {0,1,2,3,4,5,6,7}，此时会使用前两张卡 {0,1}；
# 或者当前机器通过配置环境变量 CUDA_VISIBLE_DEVICES=4,5,6,7，仅使4张
# GPU卡可见，此时会使用可见的前两张卡 {4,5}
if __name__ == '__main__':
    dist.spawn(train, args=(True,), nprocs=2)

# 使用方式4：传入训练函数、参数、指定进程数并指定当前使用的卡号
# 适用场景：训练函数需要一些参数，并且仅需要使用部分可见的GPU设备并行训练，但是
# 可能由于权限问题，无权配置当前机器的环境变量，例如：当前机器有8张GPU卡
# {0,1,2,3,4,5,6,7}，但你无权配置CUDA_VISIBLE_DEVICES，此时可以通过
# 指定参数 selected_gpus 选择希望使用的卡，例如 selected_gpus='4,5'，
# 可以指定使用第4号卡和第5号卡
if __name__ == '__main__':
    dist.spawn(train, nprocs=2, selected_gpus='4,5')

```

(下页继续)

(续上页)

```
# 使用方式5：指定多卡通信的起始端口
# 使用场景：端口建立通信时提示需要重试或者通信建立失败
# Paddle默认会通过在当前机器上寻找空闲的端口用于多卡通信，但当机器使用环境
# 较为复杂时，程序找到的端口可能不够稳定，此时可以自行指定稳定的空闲起始
# 端口以获得更稳定的训练体验
if __name__ == '__main__':
    dist.spawn(train, nprocs=2, started_port=12345)
```

模型保存

Paddle 保存的模型有两种格式，一种是训练格式，保存模型参数和优化器相关的信息，可用于恢复训练；一种是预测格式，保存预测的静态图网络结构以及参数，用于预测部署。

高层 API 场景

高层 API 下用于预测部署的模型保存方法为：

```
model = paddle.Model(Mnist())
# 预测格式，保存的模型可用于预测部署
model.save('mnist', training=False)
# 保存后可以得到预测部署所需要的模型
```

基础 API 场景

动态图训练的模型，可以通过动静转换功能，转换为可部署的静态图模型，具体做法如下：

```
import paddle
from paddle.jit import to_static
from paddle.static import InputSpec

class SimpleNet(paddle.nn.Layer):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.linear = paddle.nn.Linear(10, 3)

    # 第1处改动
    # 通过InputSpec指定输入数据的形状，None表示可变长
    # 通过to_static装饰器将动态图转换为静态图Program
    @to_static(input_spec=[InputSpec(shape=[None, 10], name='x'), InputSpec(shape=[3],
    ↵ name='y')])
```

(下页继续)

(续上页)

```

def forward(self, x, y):
    out = self.linear(x)
    out = out + y
    return out

net = SimpleNet()

# 第2处改动
# 保存静态图模型，可用于预测部署
paddle.jit.save(net, './simple_net')

```

推理

推理库 Paddle Inference 的 API 做了升级，简化了写法，以及去掉了历史上冗余的概念。API 的变化为纯增，原有 API 保持不变，但推荐新的 API 体系，旧 API 在后续版本会逐步删除。

C++ API

重要变化：

- 命名空间从 paddle 变更为 paddle_infer
- PaddleTensor, PaddleBuf 等被废弃，ZeroCopyTensor 变为默认 Tensor 类型，并更名为 Tensor
- 新增 PredictorPool 工具类简化多线程 predictor 的创建，后续也会增加更多周边工具
- CreatePredictor (原 CreatePaddlePredictor) 的返回值由 unique_ptr 变为 shared_ptr 以避免 Clone 后析构顺序出错的问题

API 变更

使用新 C++ API 的流程与之前完全一致，只有命名变化

```

#include "paddle_inference_api.h"
using namespace paddle_infer;

Config config;
config.SetModel("xxx_model_dir");

auto predictor = CreatePredictor(config);

// Get the handles for the inputs and outputs of the model
auto input0 = predictor->GetInputHandle("X");

```

(下页继续)

(续上页)

```
auto output0 = predictor->GetOutputHandle("Out");  
  
for (...) {  
    // Assign data to input0  
    MyServiceSetData(input0);  
  
    predictor->Run();  
  
    // get data from the output0 handle  
    MyServiceGetData(output0);  
}
```

Python API

Python API 的变更与 C++ 基本对应，会在 2.0 版发布。

附录

2.0 转换工具

为了降级代码升级的成本，飞桨提供了转换工具，可以帮助将 Paddle 1.8 版本开发的代码，升级为 2.0 的 API。由于相比于 Paddle 1.8 版本，2.0 版本的 API 进行了大量的升级，包括 API 名称，参数名称，行为等。转换工具当前还不能覆盖所有的 API 升级；对于无法转换的 API，转换工具会报错，提示手动升级。

https://github.com/PaddlePaddle/paddle_upgrade_tool

对于转换工具没有覆盖的 API，请查看官网的 API 文档，手动升级代码的 API。

2.0 文档教程

以下提供了 2.0 版本的一些示例教程：

你可以在官网[应用实践](#)栏目内进行在线浏览，也可以下载在这里提供的源代码：
<https://github.com/PaddlePaddle/docs/tree/develop/docs/practices>

2.7.2 版本迁移工具

在飞桨框架 2.0 中，Paddle API 的位置、命名、参数、行为，进行了系统性的调整和规范，将 API 体系从 1.X 版本的 paddle.fluid.* 迁移到了 paddle.* 下。paddle.fluid 目录下暂时保留了 1.8 版本 API，主要是兼容性考虑，未来会被删除。

使用版本迁移工具自动迁移 Paddle 1.X 的代码到 Paddle 2.0

Title underline too short.

使用版本迁移工具自动迁移 Paddle 1.X 的代码到 Paddle 2.0

飞桨提供了版本迁移工具，该工具按 Paddle 2.0 对于 Paddle 1.X 的变化，能够自动实现以下功能：

- 按照[API 映射表](#)，将转换工具能否转换这列为 True 的 API 由 Paddle 1.X 转为 Paddle 2.0，为 False 的 API 打印 WARNING，提示手动升级。
- 因为 Paddle 2.0.0 默认开启动态图，所以删除用于开启动态图上下文的 `with paddle.fluid.dygraph.guard(place)`，并修改该上下文的代码缩进；
- 删除组网 API 中的 `act` 参数，并自动添加相关的激活函数；

目前，版本迁移工具能够处理的 API 数量为 X 个，如果你有代码迁移的需求，使用转换工具能够节省你部分时间，帮助你快速完成代码迁移。

警告： 版本迁移工具并不能处理所有的情况，对于 API 的处理只能按照[API 映射表](#)中的关系完成 API 的变化。如代码中包含有转换工具能否转换这列为 False 的 API 或不在此表中的 API，在使用本工具后，仍然需要手工来进行检查并做相应的调整。

安装

版本迁移工具可以通过 pip 的方式安装，方式如下：

```
$ pip install paddle_upgrade_tool
```

基本用法

paddle_upgrade_tool 可以使用下面的方式，快速使用：

```
$ paddle_upgrade_tool --inpath /path/to/model.py
```

这将在命令行中，以 diff 的形式，展示 model.py 从 Paddle 1.x 转换为 Paddle 2.0 的变化。如果你确认上述变化没有问题，只需要再执行：

```
$ paddle_upgrade_tool --inpath /path/to/model.py --write
```

就会原地改写 model.py，将上述变化改写到你的源文件中。注意：版本转换工具会默认备份源文件，到 `~/.paddle_upgrade_tool/` 下。

参数说明如下：

- `--inpath` 输入文件路径，可以为单个文件或文件夹。
- `--write` 是否原地修改输入的文件，默认值 `False`，表示不修改。如果为 `True`，表示对文件进行原地修改。添加此参数也表示对文件进行原地修改。
- `--backup` 可选，是否备份源文件，默认值为 `~/.paddle_upgrade_tool/`，在此路径下备份源文件。
- `--no-log-file` 可选，是否需要输出日志文件，默认值为 `False`，即输出日志文件。
- `--logfilepath` 可选，输出日志的路径，默认值为 `report.log`，输出日志文件的路径。
- `--no-confirm` 可选，输入文件夹时，是否逐文件确认原地写入，只在`--write` 为 `True` 时有效，默认值为 `False`，表示需要逐文件确认。
- `--parallel` 可选，控制转换文件的并发数，当 `no-confirm` 为 `True` 时不生效，默认值:None。
- `--log-level` 可选，log 级别，可为 `['DEBUG', 'INFO', 'WARNING', 'ERROR']` 默认值: `INFO`。
- `--refactor` 可选，debug 时使用。
- `--print-match` 可选，debug 时使用。

使用教程

开始

在使用 `paddle_upgrade_tool` 前，需要确保已经安装了 Paddle 2.0.0+ 版本。

```
import paddle
print (paddle.__version__)
```

```
2.0.0
```

克隆paddlePaddle/models来作为工具的测试。

```
$ git clone https://github.com/PaddlePaddle/models
```

```
Cloning into 'models'...
remote: Enumerating objects: 8, done.
remote: Counting objects: 100% (8/8), done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 35011 (delta 1), reused 0 (delta 0), pack-reused 35003
Receiving objects: 100% (35011/35011), 356.97 MiB | 1.53 MiB/s, done.
Resolving deltas: 100% (23291/23291), done.
```

查看帮助文档

你可以直接通过下面的方式，查看帮助文档。

```
$ paddle_upgrade_tool -h
```

```
usage: paddle_upgrade_tool [-h] [--log-level {DEBUG,INFO,WARNING,ERROR}]
                           [--no-log-file] [--logfilepath LOG_FILEPATH] -i
                           INPATH [-b [BACKUP]] [-w] [--no-confirm]
                           [-p PARALLEL]
                           [-r {refactor_import,norm_api_alias,args_to_kwargs,
                           ↪refactor_kwargs,api_rename,refactor_with,post_refactor}]
                           [--print-match]

optional arguments:
  -h, --help            show this help message and exit
  --log-level {DEBUG,INFO,WARNING,ERROR}
                        set log level, default is INFO
  --no-log-file         don't log to file
  --logfilepath LOG_FILEPATH
                        set log file path, default is "report.log"
  -i INPATH, --inpath INPATH
                        the file or directory path you want to upgrade.
  -b [BACKUP], --backup [BACKUP]
                        backup directory, default is the
                        "~/.paddle_upgrade_tool/".
  -w, --write           modify files in-place.
  --no-confirm          write files in-place without confirm, ignored without
                        --write.
  -p PARALLEL, --parallel PARALLEL
                        specify the maximum number of concurrent processes to
```

(下页继续)

(续上页)

```

use when refactoring, ignored with --no-confirm.

-r {refactor_import,norm_api_alias,args_to_kwargs,refactor_kwarg,api_rename,
→refactor_with,post_refactor}, --refactor {refactor_import,norm_api_alias,args_to_
→kwargs,refactor_kwarg,api_rename,refactor_with,post_refactor}

this is a debug option. Specify refactor you want to
run. If none, all refactors will be run.

--print-match      this is a debug option. Print matched code and node
                   for each file.

```

Paddle 1.x 的例子

Title underline too short.

Paddle 1.x 的例子

^^^^^^^^^^^^^^^^^

这里是一个基于 Paddle 1.x 实现的一个 mnist 分类，部分内容如下：

```
$ head -n 198 models/dygraph/mnist/train.py | tail -n 20
```

```

with fluid.dygraph.guard(place):
    if args.ce:
        print("ce mode")
        seed = 33
        np.random.seed(seed)
        fluid.default_startup_program().random_seed = seed
        fluid.default_main_program().random_seed = seed

    if args.use_data_parallel:
        strategy = fluid.dygraph.parallel.prepare_context()
        mnist = MNIST()
        adam = AdamOptimizer(learning_rate=0.001, parameter_list=mnist.parameters())
        if args.use_data_parallel:
            mnist = fluid.dygraph.parallel.DataParallel(mnist, strategy)

        train_reader = paddle.batch(
            paddle.dataset.mnist.train(), batch_size=BATCH_SIZE, drop_last=True)
        if args.use_data_parallel:
            train_reader = fluid.contrib.reader.distributed_batch_reader(
                train_reader)

```

使用 paddle_upgrade_tool 进行转化

paddle_upgrade_tool 支持单文件的转化，你可以通过下方的命令直接转化单独的文件。

```
$ paddle_upgrade_tool --inpath models/dygraph/mnist/train.py
```

注意，对于参数的删除及一些特殊情况，迁移工具都会打印 **WARNING** 信息，需要你仔细核对相关内容。如果你觉得上述信息没有问题，可以直接对文件进行原地修改，方式如下：

```
$ paddle_upgrade_tool --inpath models/dygraph/mnist/train.py --write
```

此时，命令行会弹出下方的提示：

```
"models/dygraph/mnist/train.py" will be modified in-place, and it has been backed up  
to "~/.paddle_upgrade_tool/train.py_backup_2020_09_09_20_35_15_037821". Do you want  
to continue? [Y/n]:
```

输入 `y` 后即开始执行代码迁移。为了高效完成迁移，工具这里采用了原地写入的方式。此外，为了防止特殊情况，工具会备份转换前的代码到 `~/.paddle_upgrade_tool` 目录下，如果需要，你可以在备份目录下找到转换前的代码。

代码迁移完成后，会生成一个 `report.log` 文件，记录了迁移的详情。内容如下：

```
$ cat report.log
```

注意事项

- 本迁移工具不能完成所有 API 的迁移，有少量的 API 需要你手动完成迁移，具体信息可见 **WARNING**。

使用 Paddle 2.0

完成迁移后，代码就从 Paddle 1.x 迁移到了 Paddle 2.0，你就可以在 Paddle 2.0 下进行相关的开发。

2.7.3 兼容载入旧格式模型

如果你是从飞桨框架 1.x 切换到 2.1，曾经使用飞桨框架 1.x 的 `fluid` 相关接口保存模型或者参数，飞桨框架 2.1 也对这种情况进行了兼容性支持，包括以下几种情况。

飞桨 1.x 模型准备及训练示例，该示例为后续所有示例的前序逻辑：

```
import numpy as np
import paddle
import paddle.fluid as fluid
```

(下页继续)

(续上页)

```
import paddle.nn as nn
import paddle.optimizer as opt

BATCH_SIZE = 16
BATCH_NUM = 4
EPOCH_NUM = 4

IMAGE_SIZE = 784
CLASS_NUM = 10

# enable static mode
paddle.enable_static()

# define a random dataset
class RandomDataset(paddle.io.Dataset):
    def __init__(self, num_samples):
        self.num_samples = num_samples

    def __getitem__(self, idx):
        image = np.random.random([IMAGE_SIZE]).astype('float32')
        label = np.random.randint(0, CLASS_NUM - 1, (1, )).astype('int64')
        return image, label

    def __len__(self):
        return self.num_samples

image = fluid.data(name='image', shape=[None, 784], dtype='float32')
label = fluid.data(name='label', shape=[None, 1], dtype='int64')
pred = fluid.layers.fc(input=image, size=10, act='softmax')
loss = fluid.layers.cross_entropy(input=pred, label=label)
avg_loss = fluid.layers.mean(loss)

optimizer = fluid.optimizer.SGD(learning_rate=0.001)
optimizer.minimize(avg_loss)

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

# create data loader
dataset = RandomDataset(BATCH_NUM * BATCH_SIZE)
loader = paddle.io.DataLoader(dataset,
    feed_list=[image, label],
```

(下页继续)

(续上页)

```

places=place,
batch_size=BATCH_SIZE,
shuffle=True,
drop_last=True,
num_workers=2)

# train model
for data in loader():
    exe.run(
        fluid.default_main_program(),
        feed=data,
        fetch_list=[avg_loss])

```

1 从 paddle.fluid.io.save_inference_model 保存结果中载入模型 & 参数

1.1 同时载入模型和参数

使用 paddle.jit.load 配合 **configs 载入模型和参数。

如果你是按照 paddle.fluid.io.save_inference_model 的默认格式存储的，可以按照如下方式载入（接前述示例）：

```

# save default
model_path = "fc.example.model"
fluid.io.save_inference_model(
    model_path, ["image"], [pred], exe)

# enable dynamic mode
paddle.disable_static(place)

# load
fc = paddle.jit.load(model_path)

# inference
fc.eval()
x = paddle.randn([1, IMAGE_SIZE], 'float32')
pred = fc(x)

```

如果你指定了存储的模型文件名，可以按照以下方式载入（接前述示例）：

```

# save with model_filename
model_path = "fc.example.model.with_model_filename"
fluid.io.save_inference_model(

```

(下页继续)

(续上页)

```
model_path, ["image"], [pred], exe, model_filename="__simplenet__")

# enable dynamic mode
paddle.disable_static(place)

# load
fc = paddle.jit.load(model_path, model_filename="__simplenet__")

# inference
fc.eval()
x = paddle.randn([1, IMAGE_SIZE], 'float32')
pred = fc(x)
```

如果你指定了存储的参数文件名，可以按照以下方式载入（接前述示例）：

```
# save with params_filename
model_path = "fc.example.model.with_params_filename"
fluid.io.save_inference_model(
    model_path, ["image"], [pred], exe, params_filename="__params__")

# enable dynamic mode
paddle.disable_static(place)

# load
fc = paddle.jit.load(model_path, params_filename="__params__")

# inference
fc.eval()
x = paddle.randn([1, IMAGE_SIZE], 'float32')
pred = fc(x)
```

1.2 仅载入参数

如果你仅需要从 `paddle.fluid.io.save_inference_model` 的存储结果中载入参数，以 `state_dict` 的形式配置到已有代码的模型中，可以使用 `paddle.load` 配合 `**configs` 载入。

如果你是按照 `paddle.fluid.io.save_inference_model` 的默认格式存储的，可以按照如下方式载入（接前述示例）：

```
model_path = "fc.example.model"

load_param_dict = paddle.load(model_path)
```

如果你指定了存储的模型文件名，可以按照以下方式载入（接前述示例）：

```
model_path = "fc.example.model.with_model_filename"

load_param_dict = paddle.load(model_path, model_filename="__simplenet__")
```

如果你指定了存储的参数文件名，可以按照以下方式载入（接前述示例）：

```
model_path = "fc.example.model.with_params_filename"

load_param_dict = paddle.load(model_path, params_filename="__params__")
```

备注：一般预测模型不会存储优化器 Optimizer 的参数，因此此处载入的仅包括模型本身的参数。

备注：由于 structured_name 是动态图下独有的变量命名方式，因此从静态图存储结果载入的 state_dict 在配置到动态图的 Layer 中时，需要配置 Layer.set_state_dict(use_structured_name=False)。

2 从 paddle.fluid.save 存储结果中载入参数

paddle.fluid.save 的存储格式与 2.x 动态图接口 paddle.save 存储格式是类似的，同样存储了 dict 格式的参数，因此可以直接使用 paddle.load 载入 state_dict，但需要注意不能仅传入保存的路径，而要传入保存参数的文件名，示例如下（接前述示例）：

```
# save by fluid.save
model_path = "fc.example.model.save"
program = fluid.default_main_program()
fluid.save(program, model_path)

# enable dynamic mode
paddle.disable_static(place)

load_param_dict = paddle.load("fc.example.model.save.pdparams")
```

备注：由于 paddle.fluid.save 接口原先在静态图模式下的定位是存储训练时参数，或者说存储 Checkpoint，故尽管其同时存储了模型结构，目前也暂不支持从 paddle.fluid.save 的存储结果中同时载入模型和参数，后续如有需求再考虑支持。

3 从 paddle.fluid.io.save_params/save.persistables 保存结果中载入参数

这两个接口在飞桨 1.x 版本时，已经不再推荐作为存储模型参数的接口使用，故并未继承至飞桨 2.x，之后也不会再推荐使用这两个接口存储参数。

对于使用这两个接口存储参数兼容载入的支持，分为两种情况，下面以 paddle.fluid.io.save_params 接口为例介绍相关使用方法：

3.1 使用默认方式存储，各参数分散存储为单独的文件，文件名为参数名

这种存储方式仍然可以使用 paddle.load 接口兼容载入，使用示例如下（接前述示例）：

```
# save by fluid.io.save_params
model_path = "fc.example.model.save_params"
fluid.io.save_params(exe, model_path)

# load
state_dict = paddle.load(model_path)
print(state_dict)
```

3.2 指定了参数存储的文件，将所有参数存储至单个文件中

将所有参数存储至单个文件中会导致存储结果中丢失 Tensor 名和 Tensor 数据之间的映射关系，因此这部分丢失的信息需要用户传入进行补足。为了确保正确性，这里不仅要传入 Tensor 的 name 列表，同时要传入 Tensor 的 shape 和 dtype 等描述信息，通过检查和存储数据的匹配性确保严格的正确性，这导致载入数据的恢复过程变得比较复杂，仍然需要一些飞桨 1.x 的概念支持。后续如果此项需求较为普遍，飞桨将会考虑将该项功能兼容支持到 paddle.load 中，但由于信息丢失而导致的使用复杂性仍然是存在的，因此建议你避免仅使用这两个接口存储参数。

目前暂时推荐你使用 paddle.static.load_program_state 接口解决此处的载入问题，需要获取原 Program 中的参数列表传入该方法，使用示例如下（接前述示例）：

```
# save by fluid.io.save_params
model_path = "fc.example.model.save_params_with_filename"
fluid.io.save_params(exe, model_path, filename="__params__")

# load
import os
params_file_path = os.path.join(model_path, "__params__")
var_list = fluid.default_main_program().all_parameters()
state_dict = paddle.io.load_program_state(params_file_path, var_list)
```

4 从 paddle.static.save 保存结果中载入参数

paddle.static.save 接口生成三个文件: *.pdparams、*.pdopt、*.pdmodel，分别保存了组网的参数、优化器的参数、静态图的 Program。推荐您使用 paddle.load 分别加载这三个文件，然后使用 set_state_dict 接口将参数设置到 Program 中。如果您已经在代码中定义了 Program，您可以不加载 *.pdmodel 文件；如果您不需要恢复优化器中的参数，您可以不加载 *.pdopt 文件。使用示例如下：

```
import os
import paddle

paddle.enable_static()
x = paddle.static.data(
    name="static_x", shape=[None, 224], dtype='float32')
z = paddle.static.nn.fc(x, 10)
z = paddle.static.nn.fc(z, 10, bias_attr=False)

place = paddle.CPUPlace()
exe = paddle.static.Executor(place)
exe.run(paddle.static.default_startup_program())
prog = paddle.static.default_main_program()

path = os.path.join("test_static_save_load", "model")
paddle.static.save(prog, path)

# load program
program=paddle.load(path + '.pdmodel')

state_dict_param = paddle.load(path + '.pdparams')
program.set_state_dict(state_dict_param)

state_dict_opt = paddle.load(path + '.pdopt')
program.set_state_dict(state_dict_opt)
```

2.7.4 Paddle 1.8 与 Paddle 2.0 API 映射表

Title underline too short.

Paddle 1.8 与 Paddle 2.0 API 映射表
=====

本文档基于 Paddle 1.8 梳理了常用 API 与 Paddle 2.0 对应关系。你可以根据对应关系，快速熟悉 Paddle 2.0 的接口使用。

备注:

- 2.0 版本将会是一个长期维护的版本，我们将会发布新增二位版本号版本进行功能增强、以及性能优化，通过发布新增三位版本号版本进行 bugfix。
- 我们还会继续维护 1.8 版本，但仅限于严重的 bugfix。

备注：其中，迁移工具能否转换，是指使用迁移工具能否直接对 PaddlePaddle 1.8 的 API 进行迁移，了解更多关于迁移工具的内容，请参考[版本迁移工具](#)

序号	PaddlePaddle 1.8 API	PaddlePaddle 2.0 API	迁移工具能否转换
0	paddle.fluid.BuildStrategy	paddle.static.BuildStrategy	True
1	paddle.fluid.CompiledProgram	paddle.static.CompiledProgram	True
2	paddle.fluid.cpu_places	paddle.static.cpu_places	True
3	paddle.fluid.CPUPlace	paddle.CPUPlace	True
4	paddle.fluid.cuda_places	paddle.static.cuda_places	True
5	paddle.fluid.CUDAPinnedPlace	paddle.CUDAPinnedPlace	True
6	paddle.fluid.CUDAPlace	paddle.CUDAPlace	True
7	pad-dle.fluid.default_main_program	pad-dle.static.default_main_program	True
8	pad-dle.fluid.default_startup_program	pad-dle.static.default_startup_program	True
9	paddle.fluid.disable_dygraph	paddle.enable_static	True
10	paddle.fluid.embedding	pad-dle.nn.functional.embedding(动态图), pad-dle.static.nn.embedding(静态图)	False
11	paddle.fluid.enable_dygraph	paddle.disable_static	True
12	paddle.fluid.enable_imperative	paddle.disable_static	True
13	paddle.fluid.ExecutionStrategy	paddle.static.ExecutionStrategy	True
14	paddle.fluid.Executor	paddle.static.Executor	True
15	paddle.fluid.global_scope	paddle.static.global_scope	True
16	paddle.fluid.gradients	paddle.static.gradients	True
17	paddle.fluid.in_dygraph_mode	paddle.in_dynamic_mode	True
18	pad-dle.fluid.is_compiled_with_cuda	paddle.is_compiled_with_cuda	True
19	paddle.fluid.load	paddle.static.load	True
21	paddle.fluid.name_scope	paddle.static.name_scope	True

下页继续

表 2.1 - 续上页

序号	PaddlePaddle 1.8 API	PaddlePaddle 2.0 API	迁移工具能否转换
22	paddle.fluid.one_hot	paddle.nn.functional.one_hot	False
23	paddle.fluid.ParallelExecutor	paddle.static.ParallelExecutor	True
24	paddle.fluid.ParamAttr	paddle.ParamAttr	True
25	paddle.fluid.Program	paddle.static.Program	True
26	paddle.fluid.program_guard	paddle.static.program_guard	True
27	paddle.fluid.require_version	paddle.utils.require_version	True
28	paddle.fluid.save	paddle.save	False
29	paddle.fluid.scope_guard	paddle.static.scope_guard	True
30	paddle.fluid.Variable	paddle.static.Variable	True
31	pad-dle.fluid.WeightNormParamAttr	pad-dle.static.WeightNormParamAttr	True
32	pad-dle.fluid.backward.append_backward	paddle.static.append_backward	True
33	paddle.fluid.backward.gradients	paddle.static.gradients	True
34	pad-dle.fluid.clip.GradientClipByGlobalNorm	pad-dle.nn.ClipGradByGlobalNorm	False
35	pad-dle.fluid.clip.GradientClipByNorm	paddle.nn.ClipGradByNorm	False
36	pad-dle.fluid.clip.GradientClipByValue	paddle.nn.ClipGradByValue	False
37	pad-dle.fluid.dataset.InMemoryDataset	pad-dle.distributed.InMemoryDataset	True
38	paddle.fluid.dataset.QueueDataset	paddle.distributed.QueueDataset	True
39	paddle.fluid.dygraph.BatchNorm	paddle.nn.BatchNorm1D, pad-dle.nn.BatchNorm2D, pad-dle.nn.BatchNorm3D	True
40	paddle.fluid.dygraph.BCELoss	paddle.nn.BCELoss	True
41	pad-dle.fluid.dygraph.BilinearTensorProduct	paddle.nn.Bilinear	True
42	paddle.fluid.dygraph.Conv2D	paddle.nn.Conv2D	True
43	pad-dle.fluid.dygraph.Conv2DTranspose	paddle.nn.Conv2DTranspose	False
44	paddle.fluid.dygraph.Conv3D	paddle.nn.Conv3D	True
45	pad-dle.fluid.dygraph.Conv3DTranspose	paddle.nn.Conv3DTranspose	True
46	paddle.fluid.dygraph.CosineDecay	pad-dle.optimizer.lr.CosineAnnealingDecay	False

下页继续

表 2.1 - 续上页

序号	PaddlePaddle 1.8 API	PaddlePaddle 2.0 API	迁移工具能否转换
47	paddle.fluid.dygraph.DataParallel	paddle.DataParallel	True
48	pad-dle.fluid.dygraph.disable_dygraph	paddle.enable_static	True
49	paddle.fluid.dygraph.Dropout	paddle.nn.Dropout, pad-dle.nn.Dropout2D, pad-dle.nn.Dropout3D	False
50	paddle.fluid.dygraph.Embedding	paddle.nn.Embedding	False
51	pad-dle.fluid.dygraph.enable_dygraph	paddle.disable_static	True
52	pad-dle.fluid.dygraph.enable_imperative	paddle.disable_static	True
53	paddle.fluid.dygraph.grad	paddle.grad	True
54	paddle.fluid.dygraph.GroupNorm	paddle.nn.GroupNorm	True
55	pad-dle.fluid.dygraph.InstanceNorm	paddle.nn.InstanceNorm1D, paddle.nn.InstanceNorm2D, paddle.nn.InstanceNorm3D	False
56	paddle.fluid.dygraph.L1Loss	paddle.nn.L1Loss	True
57	paddle.fluid.dygraph.Layer	paddle.nn.Layer	True
58	paddle.fluid.dygraph.LayerList	paddle.nn.LayerList	True
59	paddle.fluid.dygraph.LayerNorm	paddle.nn.LayerNorm	False
60	paddle.fluid.dygraph.Linear	paddle.nn.Linear	True
61	paddle.fluid.dygraph.load_dygraph	paddle.load	True
62	paddle.fluid.dygraph.MSELoss	paddle.nn.MSELoss	True
63	pad-dle.fluid.dygraph.NaturalExpDecay	pad-dle.optimizer.lr.NaturalExpDecay	False
64	paddle.fluid.dygraph.NLLLoss	paddle.nn.NLLLoss	True
65	paddle.fluid.dygraph.NoamDecay	paddle.optimizer.lr.NoamDecay	False
66	pad-dle.fluid.dygraph.ParameterList	paddle.nn.ParameterList	True
67	paddle.fluid.dygraph.no_grad	paddle.no_grad	True
68	pad-dle.fluid.dygraph.PolynomialDecay	pad-dle.optimizer.lr.PolynomialDecay	False
69	paddle.fluid.dygraph.Pool2D	paddle.nn.MaxPool2D, pad-dle.nn.AvgPool2D	False
70	paddle.fluid.dygraph.PRelu	paddle.nn.PReLU	False
71	pad-dle.fluid.dygraph.ProgramTranslator	paddle.jit.ProgramTranslator	True

下页继续

表 2.1 - 续上页

序号	PaddlePaddle 1.8 API	PaddlePaddle 2.0 API	迁移工具能否转换
72	paddle.fluid.dygraph.Sequential	paddle.nn.Sequential	True
73	pad-dle.fluid.dygraph.SpectralNorm	paddle.nn.SpectralNorm	True
74	paddle.fluid.dygraph.to_variable	paddle.to_tensor	True
75	paddle.fluid.dygraph.TracedLayer	paddle.jit.TracedLayer	True
76	paddle.fluid.executor.Executor	paddle.static.Executor	True
77	paddle.fluid.executor.global_scope	paddle.static.global_scope	True
78	paddle.fluid.executor.scope_guard	paddle.static.scope_guard	True
79	paddle.fluid.initializer.Bilinear	paddle.nn.initializer.Bilinear	True
80	pad-dle.fluid.initializer.BilinearInitializer	paddle.nn.initializer.Bilinear	True
81	paddle.fluid.initializer.Constant	paddle.nn.initializer.Constant	True
82	pad-dle.fluid.initializer.ConstantInitializer	paddle.nn.initializer.Constant	True
83	paddle.fluid.initializer.MSRA	pad-dle.nn.initializer.KaimingNormal, pad-dle.nn.initializer.KaimingUniform	False
84	pad-dle.fluid.initializer.MSRAInitializer	pad-dle.nn.initializer.KaimingNormal, pad-dle.nn.initializer.KaimingUniform	False
85	paddle.fluid.initializer.Normal	paddle.nn.initializer.Normal	True
86	pad-dle.fluid.initializer.NormalInitializer	paddle.nn.initializer.Normal	True
87	pad-dle.fluid.initializer.NumpyArrayInitializer	paddle.nn.initializer.Assign	True
88	pad-dle.fluid.initializer.TruncatedNormal	pad-dle.nn.initializer.TruncatedNormal	False
89	pad-dle.fluid.initializer.TruncatedNormal	pad-dle.nn.initializer.TruncatedNormal	False
90	paddle.fluid.initializer.Uniform	paddle.nn.initializer.Uniform	True
91	pad-dle.fluid.initializer.UniformInitializer	paddle.nn.initializer.Uniform	True

下页继续

表 2.1 - 续上页

序号	PaddlePaddle 1.8 API	PaddlePaddle 2.0 API	迁移工具能否转换
92	paddle.fluid.initializer.Xavier	pad-dle.nn.initializer.XavierNormal, pad-dle.nn.initializer.XavierUniform	False
93	pad-dle.fluid.initializer.XavierInitializer	pad-dle.nn.initializer.XavierNormal, pad-dle.nn.initializer.XavierUniform	False
94	paddle.fluid.io.DataLoader	paddle.io.DataLoader	True
95	paddle.fluid.io.load	paddle.static.load	True
96	pad-dle.fluid.io.load_inference_model	pad-dle.static.load_inference_model	True
97	paddle.fluid.io.load_program_state	paddle.static.load_program_state	True
98	paddle.fluid.io.save	paddle.save, paddle.static.save	False
99	pad-dle.fluid.io.save_inference_model	pad-dle.static.save_inference_model	True
100	paddle.fluid.io.set_program_state	paddle.static.set_program_state	True
101	paddle.fluid.layers.abs	paddle.abs	True
102	paddle.fluid.layers.accuracy	paddle.metric.accuracy	True
103	paddle.fluid.layers.acos	paddle.acos	True
104	pad-dle.fluid.layers.adaptive_pool2d	pad-dle.nn.functional.adaptive_avg_pool2d, pad-dle.nn.functional.adaptive_max_pool2d	False
105	pad-dle.fluid.layers.adaptive_pool3d	pad-dle.nn.functional.adaptive_max_pool3d, pad-dle.nn.functional.adaptive_avg_pool3d	False
106	paddle.fluid.layers.embedding	pad-dle.nn.functional.embedding(动态图), pad-dle.static.nn.embedding(静态图)	False
107	paddle.fluid.layers.addmm	paddle.addmm	True
108	paddle.fluid.layers.affine_grid	paddle.nn.functional.affine_grid	True
109	paddle.fluid.layers.allclose	paddle.allclose	True
110	paddle.fluid.layers.arange	paddle.arange	True

下页继续

表 2.1 - 续上页

序号	PaddlePaddle 1.8 API	PaddlePaddle 2.0 API	迁移工具能否转换
111	paddle.fluid.layers.argmax	paddle.argmax	True
112	paddle.fluid.layers.argmin	paddle.argmin	True
113	paddle.fluid.layers.argsort	paddle.argsort	True
114	paddle.fluid.layers.asin	paddle.asin	True
115	paddle.fluid.layers.atan	paddle.atan	True
116	paddle.fluid.layers.auc	paddle.metric.Auc	True
117	paddle.fluid.layers.batch_norm	paddle.static.nn.batch_norm	False
118	pad-dle.fluid.layers.bilinear_tensor_product	paddle.nn.functional.bilinear	True
119	paddle.fluid.layers.bmm	paddle.bmm	True
120	paddle.fluid.layers.case	paddle.static.nn.case	True
121	paddle.fluid.layers.cast	paddle.cast	True
122	paddle.fluid.layers.Categorical	paddle.distribution.Categorical	True
123	paddle.fluid.layers.ceil	paddle.ceil	True
124	paddle.fluid.layers.chunk_eval	paddle.metric.chunk_eval	True
125	paddle.fluid.layers.clamp	paddle.clip	False
126	paddle.fluid.layers.clip_by_norm	paddle.nn.clip_by_norm	False
127	paddle.fluid.layers.concat	paddle.concat	True
128	paddle.fluid.layers.cond	paddle.static.nn.cond	True
129	paddle.fluid.layers.conv2d	paddle.nn.functional.conv2d(动态图), paddle.static.nn.conv2d(静态图),	False
130	pad-dle.fluid.layers.conv2d_transpose	pad-dle.nn.functional.conv2d_transpose(动态图), pad-dle.static.nn.conv2d_transpose(静态图)	False
131	paddle.fluid.layers.conv3d	paddle.nn.functional.conv3d(动态图), paddle.static.nn.conv3d(静态图)	False
132	pad-dle.fluid.layers.conv3d_transpose	pad-dle.nn.functional.conv3d_transpose(动态图), pad-dle.static.nn.conv3d_transpose(静态图)	False
133	paddle.fluid.layers.cos	paddle.cos	True

下页继续

表 2.1 - 续上页

序号	PaddlePaddle 1.8 API	PaddlePaddle 2.0 API	迁移工具能否转换
134	paddle.fluid.layers.cos_sim	pad-dle.nn.functional.cosine_similarity	True
135	pad-dle.fluid.layers.create_parameter	paddle.create_parameter	True
136	paddle.fluid.layers.crf_decoding	paddle.static.nn.crf_decoding	True
137	paddle.fluid.layers.crop	paddle.crop	True
138	paddle.fluid.layers.cross	paddle.cross	True
139	paddle.fluid.layers.cumsum	paddle.cumsum	False
140	paddle.fluid.layers.data	paddle.static.data	True
141	paddle.fluid.layers.data_norm	paddle.static.nn.data_norm	True
142	pad-dle.fluid.layers.deformable_conv	paddle.static.nn.deform_conv2d	False
143	paddle.fluid.layers.diag	paddle.diag	False
144	paddle.fluid.layers.diag_embed	paddle.nn.functional.diag_embed	True
145	paddle.fluid.layers.dice_loss	paddle.nn.functional.dice_loss	True
146	paddle.fluid.layers.dist	paddle.dist	True
147	paddle.fluid.layers.dot	paddle.dot	True
148	paddle.fluid.layers.dropout	paddle.nn.functional.dropout, pad-dle.nn.functional.dropout2d, pad-dle.nn.functional.dropout3d	False
149	paddle.fluid.layers.dynamic_gru	paddle.nn.GRU	False
150	pad-dle.fluid.layers.dynamic_decode	paddle.nn.dynamic_decode	True
151	pad-dle.fluid.layers.elementwise_add	paddle.add	True
152	pad-dle.fluid.layers.elementwise_div	paddle.divide	True
153	pad-dle.fluid.layers.elementwise_equal	paddle.equal	True
154	pad-dle.fluid.layers.elementwise_floordiv	paddle.floor_divide	True
155	pad-dle.fluid.layers.elementwise_max	paddle.maximum	True
156	pad-dle.fluid.layers.elementwise_min	paddle.minimum	True
157	pad-dle.fluid.layers.elementwise_mod	paddle.mod	True

下页继续

表 2.1 - 续上页

序号	PaddlePaddle 1.8 API	PaddlePaddle 2.0 API	迁移工具能否转换
158	paddle.fluid.layers.elementwise_mul	paddle.multiply	True
159	paddle.fluid.layers.elu	paddle.nn.functional.elu	True
160	paddle.fluid.layers.embedding	pad-dle.nn.functional.embedding(动态图), paddle.static.nn.embedding(静态图)	True
161	paddle.fluid.layers.erf	paddle.erf	True
162	paddle.fluid.layers.exp	paddle.exp	True
163	paddle.fluid.layers.expand	paddle.expand	False
164	paddle.fluid.layers.expand_as	paddle.expand_as	True
165	pad-dle.fluid.layers.exponential_decay	pad-dle.optimizer.lr.ExponentialDecay	False
166	paddle.fluid.layers.eye	paddle.eye	True
167	paddle.fluid.layers.fc	paddle.nn.functional.linear(动态图), paddle.static.nn.fc(静态图)	True
168	paddle.fluid.layers.flatten	paddle.flatten	False
169	paddle.fluid.layers.flip	paddle.flip	True
170	paddle.fluid.layers.floor	paddle.floor	True
171	paddle.fluid.layers.full_like	paddle.full_like	True
172	paddle.fluid.layers.gather	paddle.gather	True
173	paddle.fluid.layers.gather_nd	paddle.gather_nd	True
174	paddle.fluid.layers.gelu	paddle.nn.functional.gelu	True
175	paddle.fluid.layers.greater_equal	paddle.greater_equal	True
176	paddle.fluid.layers.greater_than	paddle.greater_than	True
177	paddle.fluid.layers.group_norm	paddle.static.nn.group_norm	True
178	paddle.fluid.layers.GRUCell	paddle.nn.GRUCell	False
179	paddle.fluid.layers.hard_shrink	paddle.nn.functional.hardshrink	True
180	paddle.fluid.layers.hard_sigmoid	paddle.nn.functional.hardsigmoid	True
181	paddle.fluid.layers.hard_swish	paddle.nn.functional.hardswish	True
182	paddle.fluid.layers.has_inf	paddle.isinf	False
183	paddle.fluid.layers.has_nan	paddle.isnan	False
184	paddle.fluid.layers.hsigmoid	pad-dle.nn.functional.hsigmoid_loss	False
185	paddle.fluid.layers.increment	paddle.increment	True

下页继续

表 2.1 - 续上页

序号	PaddlePaddle 1.8 API	PaddlePaddle 2.0 API	迁移工具能否转换
186	pad-dle.fluid.layers.inverse_time_decay	pad-dle.optimizer.lr.InverseTimeDecay	False
187	paddle.fluid.layers.index_select	paddle.index_select	True
188	paddle.fluid.layers.instance_norm	paddle.static.nn.instance_norm	False
189	paddle.fluid.layers.interpolate	paddle.nn.functional.interpolate	False
190	paddle.fluid.layers.is_empty	paddle.is_empty	True
191	paddle.fluid.layers.isfinite	paddle.isfinite	True
192	paddle.fluid.layers.kldiv_loss	paddle.nn.functional.kl_div	True
193	paddle.fluid.layers.kron	paddle.kron	True
194	paddle.fluid.layers.label_smooth	paddle.nn.functional.label_smooth	True
195	paddle.fluid.layers.layer_norm	paddle.static.nn.layer_norm	True
196	paddle.fluid.layers.leaky_relu	paddle.nn.functional.leaky_relu	True
197	paddle.fluid.layers.less_equal	paddle.less_equal	True
198	paddle.fluid.layers.less_than	paddle.less_than	True
199	paddle.fluid.layers.linspace	paddle.linspace	True
200	paddle.fluid.layers.log	paddle.log	True
201	paddle.fluid.layers.log1p	paddle.log1p	True
202	paddle.fluid.layers.log_loss	paddle.nn.functional.log_loss	True
203	paddle.fluid.layers.log_softmax	paddle.nn.functional.log_softmax	True
204	paddle.fluid.layers.logical_and	paddle.logical_and	True
205	paddle.fluid.layers.logical_not	paddle.logical_not	True
206	paddle.fluid.layers.logical_or	paddle.logical_or	True
207	paddle.fluid.layers.logical_xor	paddle.logical_xor	True
208	paddle.fluid.layers.logsigmoid	paddle.nn.functional.log_sigmoid	True
209	paddle.fluid.layers.logsumexp	paddle.logsumexp	True
210	paddle.fluid.layers.lrn	pad-dle.nn.functional.local_response_norm	True
211	paddle.fluid.layers.lstm	paddle.nn.LSTM	False
212	pad-dle.fluid.layers.margin_rank_loss	pad-dle.nn.functional.margin_ranking_loss	False
213	paddle.fluid.layers.maxout	paddle.nn.functional.maxout	True
214	paddle.fluid.layers.mean_iou	paddle.metric.mean_iou	True
215	paddle.fluid.layers.meshgrid	paddle.meshgrid	True
216	paddle.fluid.layers.mse_loss	paddle.nn.functional.mse_loss	True
217	paddle.fluid.layers.mul	paddle.matmul	False
218	paddle.fluid.layers.multi_box_head	paddle.static.nn.multi_box_head	True
219	paddle.fluid.layers.multiplex	paddle.multiplex	True

下页继续

表 2.1 - 续上页

序号	PaddlePaddle 1.8 API	PaddlePaddle 2.0 API	迁移工具能否转换
220	paddle.fluid.layers.nce	paddle.static.nn.nce	True
221	paddle.fluid.layers.nonzero	paddle.nonzero	True
222	paddle.fluid.layers.Normal	paddle.distribution.Normal	True
223	paddle.fluid.layers.not_equal	paddle.not_equal	True
224	paddle.fluid.layers.npair_loss	paddle.nn.functional.npair_loss	True
225	paddle.fluid.layers.one_hot	paddle.nn.functional.one_hot	False
226	paddle.fluid.layers.ones	paddle.ones	True
227	paddle.fluid.layers.ones_like	paddle.ones_like	True
228	paddle.fluid.layers.pad2d	paddle.nn.functional.pad	False
229	pad- dle.fluid.layers.piecewise_decay	pad- dle.optimizer.lr.PiecewiseDecay	False
230	paddle.fluid.layers.pixel_shuffle	paddle.nn.functional.pixel_shuffle	True
231	paddle.fluid.layers.pool2d	paddle.nn.functional.avg_pool2d, paddle.nn.functional.max_pool2d	False
232	paddle.fluid.layers.pool3d	paddle.nn.functional.avg_pool3d, paddle.nn.functional.max_pool3d	False
233	paddle.fluid.layers.pow	paddle.pow	True
234	paddle.fluid.layers.prelu	paddle.nn.functional.prelu(动态 图), paddle.static.nn.prelu(静态 图)	True
235	paddle.fluid.layers.Print	paddle.static.Print	True
236	paddle.fluid.layers.py_func	paddle.static.py_func	True
237	paddle.fluid.layers.randint	paddle.randint	True
238	paddle.fluid.layers.randn	paddle.randn	True
239	paddle.fluid.layers.random_crop	paddle.vision.RandomCrop	False
240	paddle.fluid.layers.randperm	paddle.randperm	True
241	paddle.fluid.layers.rank	paddle.rank	True
242	paddle.fluid.layers.reciprocal	paddle.reciprocal	True
243	paddle.fluid.layers.reduce_all	paddle.all	True
244	paddle.fluid.layers.reduce_any	paddle.any	True
245	paddle.fluid.layers.reduce_max	paddle.max	True
246	paddle.fluid.layers.reduce_mean	paddle.mean	True
247	paddle.fluid.layers.reduce_min	paddle.min	True
248	paddle.fluid.layers.reduce_prod	paddle.prod	True
249	paddle.fluid.layers.reduce_sum	paddle.sum	True
250	paddle.fluid.layers.relu	paddle.nn.functional.relu	True
251	paddle.fluid.layers.relu6	paddle.nn.functional.relu6	False

下页继续

表 2.1 - 续上页

序号	PaddlePaddle 1.8 API	PaddlePaddle 2.0 API	迁移工具能否转换
252	paddle.fluid.layers.reshape	paddle.reshape	True
253	paddle.fluid.layers.rnn	paddle.nn.RNN	False
254	paddle.fluid.layers.roll	paddle.roll	True
255	paddle.fluid.layers.round	paddle.round	True
256	paddle.fluid.layers.rsqrt	paddle.rsqrt	True
257	paddle.fluid.layers.RNNCell	paddle.nn.RNNCellBase	False
258	paddle.fluid.layers.scale	paddle.scale	True
259	paddle.fluid.layers.scatter	paddle.scatter	True
260	paddle.fluid.layers.scatter_nd_add	paddle.scatter_nd_add	True
261	paddle.fluid.layers.scatter_nd	paddle.scatter_nd	True
262	paddle.fluid.layers.selu	paddle.nn.functional.selu	True
263	paddle.fluid.layers.shape	paddle.shape	True
264	paddle.fluid.layers.shard_index	paddle.shard_index	True
265	paddle.fluid.layers.sigmoid	paddle.nn.functional.sigmoid	True
266	pad-dle.fluid.layers.sigmoid_cross_entropy	pad-dle.nn.functional.binary_crossentropy	False
267	pad-dle.fluid.layers.sigmoid_focal_loss	pad-dle.nn.functional.sigmoid_focal_loss	True
268	paddle.fluid.layers.sign	paddle.sign	True
269	paddle.fluid.layers.sin	paddle.sin	True
270	paddle.fluid.layers.size	paddle.numel	True
271	paddle.fluid.layers.slice	paddle.slice	True
272	paddle.fluid.layers.smooth_l1	pad-dle.nn.functional.smooth_l1_loss	False
273	paddle.fluid.layers.softmax	paddle.nn.functional.softmax	True
274	pad-dle.fluid.layers.softmax_with_cross_entropy	paddle.nn.functional.cross_entropy	True
275	paddle.fluid.layers.softplus	paddle.nn.functional.softplus	True
276	paddle.fluid.layers.softshrink	paddle.nn.functional.softshrink	True
277	paddle.fluid.layers.softsign	paddle.nn.functional.softsign	True
278	paddle.fluid.layers.spectral_norm	paddle.static.nn.spectral_norm	True
279	paddle.fluid.layers.split	paddle.split	True
280	paddle.fluid.layers.sqrt	paddle.sqrt	True
281	paddle.fluid.layers.square	paddle.square	True
282	pad-dle.fluid.layers.square_error_cost	pad-dle.nn.functional.square_error_cost	True
283	paddle.fluid.layers.squeeze	paddle.squeeze	True

下页继续

表 2.1 - 续上页

序号	PaddlePaddle 1.8 API	PaddlePaddle 2.0 API	迁移工具能否转换
284	paddle.fluid.layers.stack	paddle.stack	True
285	paddle.fluid.layers.stanh	paddle.stanh	True
286	paddle.fluid.layers.strided_slice	paddle.strided_slice	True
287	paddle.fluid.layers.sums	paddle.add_n	True
288	paddle.fluid.layers.swish	paddle.nn.functional.swish	True
289	paddle.fluid.layers.switch_case	paddle.static.nn.switch_case	True
290	paddle.fluid.layers.t	paddle.t	True
291	paddle.fluid.layers.tanh	paddle.tanh	True
292	paddle.fluid.layers.tanh_shrink	paddle.nn.functional.tanhshrink	True
293	pad-dle.fluid.layers.thresholded_relu	pad-dle.nn.functional.thresholded_relu	True
294	paddle.fluid.layers.topk	paddle.topk	True
295	paddle.fluid.layers.trace	paddle.trace	True
296	paddle.fluid.layers.transpose	paddle.transpose	True
297	paddle.fluid.layers.tril	paddle.tril	True
298	paddle.fluid.layers.triu	paddle.triu	True
299	paddle.fluid.layers.unfold	paddle.nn.functional.unfold	True
300	paddle.fluid.layers.Uniform	paddle.distribution.Uniform	True
301	paddle.fluid.layers.unique	paddle.unique	True
302	paddle.fluid.layers.unsqueeze	paddle.unsqueeze	True
303	paddle.fluid.layers.unstack	paddle.unstack	True
304	paddle.fluid.layers.warpctc	paddle.nn.functional.ctc_loss	False
305	paddle.fluid.layers.where	paddle.where	False
306	paddle.fluid.layers.while_loop	paddle.static.nn.while_loop	True
307	paddle.fluid.layers.zeros	paddle.zeros	True
308	paddle.fluid.layers.zeros_like	paddle.zeros_like	True
309	paddle.fluid.metrics.Accuracy	Duplicate explicit target name: "paddle.metric.accuracy". paddle.metric.Accuracy	True
310	paddle.fluid.metrics.Precision	paddle.metric.Precision	True
311	paddle.fluid.metrics.Recall	paddle.metric.Recall	True
312	paddle.fluid.optimizer.Adadelta	paddle.optimizer.Adadelta	True
313	pad-dle.fluid.optimizer.AdadeltaOptimizer	paddle.optimizer.Adadelta	True
314	paddle.fluid.optimizer.Adagrad	paddle.optimizer.Adagrad	True
315	pad-dle.fluid.optimizer.AdagradOptimizer	paddle.optimizer.Adagrad	True

下页继续

表 2.1 - 续上页

序号	PaddlePaddle 1.8 API	PaddlePaddle 2.0 API	迁移工具能否转换
316	paddle.fluid.optimizer.Adam	paddle.optimizer.Adam	True
317	paddle.fluid.optimizer.Adamax	paddle.optimizer.Adamax	True
318	pad-dle.fluid.optimizer.AdamaxOptimizer	paddle.optimizer.Adamax	True
319	pad-dle.fluid.optimizer.AdamOptimizer	paddle.optimizer.Adam	True
320	paddle.fluid.optimizer.Momentum	paddle.optimizer.Momentum	True
321	pad-dle.fluid.optimizer.MomentumOptimizer	paddle.optimizer.Momentum	True
322	pad-dle.fluid.optimizer.RMSPropOptimizer	paddle.optimizer.RMSProp	True
323	paddle.fluid.optimizer.SGD	paddle.optimizer.SGD	True
324	pad-dle.fluid.optimizer.SGDOptimizer	paddle.optimizer.SGD	True
325	paddle.fluid.regularizer.L1Decay	paddle.regularizer.L1Decay	True
326	pad-dle.fluid.regularizer.L1DecayRegularizer	paddle.regularizer.L1Decay	True
327	paddle.fluid.regularizer.L2Decay	paddle.regularizer.L2Decay	True
328	pad-dle.fluid.regularizer.L2DecayRegularizer	paddle.regularizer.L2Decay	True
329	paddle.fluid.unique_name.generate	paddle.utils.unique_name.generate	True
330	paddle.fluid.unique_name.guard	paddle.utils.unique_name.guard	True
331	paddle.fluid.layers.matmul	Duplicate explicit target name: "paddle.matmul". paddle.matmul	True
332	paddle.fluid.layers.clip	Duplicate explicit target name: "paddle.clip". paddle.clip	True
333	paddle.fluid.data	Duplicate explicit target name: "paddle.static.data". paddle.static.data	True
334	paddle.fluid.layers.load	Duplicate explicit target name: "paddle.load". paddle.load	False

下页继续

表 2.1 - 续上页

序号	PaddlePaddle 1.8 API	PaddlePaddle 2.0 API	迁移工具能否转换
335	paddle.fluid.dygraph.InverseTimeDecay	Duplicate explicit target name: "paddle.optimizer.lr.inversetimedecay". paddle.optimizer.lr.InverseTimeDecay	False
336	paddle.fluid.metrics.Auc	Duplicate explicit target name: "paddle.metric.auc". paddle.metric.Auc	False
337	paddle.fluid.dygraph.ExponentialDecay	Duplicate explicit target name: "paddle.optimizer.lr.exponentialdecay". paddle.optimizer.lr.ExponentialDecay	False
338	paddle.fluid.layers.full	paddle.full	True
339	paddle.fluid.layers.elementwise_sub	paddle.subtract	False
340	paddle.fluid.dygraph.PiecewiseDecay	Duplicate explicit target name: "paddle.optimizer.lr.piecewisedecay". paddle.optimizer.lr.PiecewiseDecay	False
341	paddle.fluid.unique_name.switch	paddle.utils.unique_name.switch	True
342	paddle.fluid.layers.fill_constant	paddle.full	True
343	paddle.fluid.layers.equal	Duplicate explicit target name: "paddle.equal". paddle.equal	True
344	paddle.fluid.layers.mean	Duplicate explicit target name: "paddle.mean". paddle.mean	True
345	paddle.fluid.dygraph.save_dygraph	Duplicate explicit target name: "paddle.save". paddle.save	True
346	paddle.fluid.layers.yolo_box	paddle.vision.ops.yolo_box	True
347	paddle.fluid.layers.yolov3_loss	paddle.vision.ops.yolo_loss	True
348	paddle.fluid.dygraph.ParallelEnv	paddle.distributed.ParallelEnv	True

下页继续

表 2.1 - 续上页

序号	PaddlePaddle 1.8 API	PaddlePaddle 2.0 API	迁移工具能否转换
349	paddle.fluid.dygraph.GRUUnit	Duplicate explicit target name: "paddle.nn.grucell". paddle.nn.GRUCell	True
350	paddle.fluid.layers.row_conv	paddle.static.nn.row_conv	True
351	paddle.fluid.layers.pad	Duplicate explicit target name: "paddle.nn.functional.pad". paddle.nn.functional.pad	False
352	paddle.fluid.layers.cross_entropy	Duplicate explicit target name: "paddle.nn.functional.cross_entropy". paddle.nn.functional.cross_entropy	True
353	paddle.fluid.layers.range	Duplicate explicit target name: "paddle.arange". paddle.arange	True
354	paddle.fluid.layers.DynamicRNN	Duplicate explicit target name: "paddle.nn.rnn". paddle.nn.RNN	False
355	pad- dle.fluid.layers.BeamSearchDecoder	paddle.nn.BeamSearchDecoder	True
356	pad- dle.fluid.layers.elementwise_pow	Duplicate explicit target name: "paddle.pow". paddle.pow	True
357	paddle.fluid.layers.assign	paddle.assign	True
358	paddle.fluid.layers.dynamic_lstm	Duplicate explicit target name: "paddle.nn.lstm". paddle.nn.LSTM	False
359	paddle.fluid.layers.lstm_unit	paddle.nn.LSTMCell	False
360	paddle.fluid.layers.LSTMCell	paddle.nn.LSTMCell	False
361	paddle.fluid.optimizer.RMSProp	Duplicate explicit target name: "paddle.optimizer.rmsprop". paddle.optimizer.RMSProp	True
362	paddle.fluid.layers.gru_unit	Duplicate explicit target name: "paddle.nn.grucell". paddle.nn.GRUCell	False

2.7.5 PyTorch 1.8 与 Paddle 2.0 API 映射表

本文档基于X2Paddle研发过程梳理了 PyTorch (v1.8.1) 常用 API 与 PaddlePaddle 2.0.0 API 对应关系与差异分析。通过本文档，帮助开发者快速迁移 PyTorch 使用经验，完成模型的开发与调优。

X2Paddle 介绍

X2Paddle 致力于帮助其它主流深度学习框架开发者快速迁移至飞桨框架，目前提供三大功能

- 预测模型转换
 - 支持 Caffe/TensorFlow/ONNX/PyTorch 的模型一键转为飞桨的预测模型，并使用 PaddleInference/PaddleLite 进行 CPU/GPU/Arm 等设备的部署
- PyTorch 训练项目转换
 - 支持 PyTorch 项目 Python 代码（包括训练、预测）一键转为基于飞桨框架的项目代码，帮助开发者快速迁移项目，并可享受 AIStudio 平台对于飞桨框架提供的海量免费计算资源
- API 映射文档
 - 详细的 API 文档对比分析，帮助开发者快速从 PyTorch 框架的使用迁移至飞桨框架的使用，大大降低学习成本

详细的项目信息与使用方法参考 X2Paddle 在 Github 上的开源项目: <https://github.com/PaddlePaddle/X2Paddle>

API 映射表目录

基础操作类 API 映射列表

梳理了基础操作的 PyTorch-PaddlePaddle API 映射列表，主要包括了构造 Tensor、数学计算、逻辑计算相关的 API。

持续更新...

组网类 API 映射列表

梳理了与构造网络相关的 PyTorch-PaddlePaddle API 映射列表。

持续更新...

Loss 类 API 映射列表

梳理了计算 loss 相关的 PyTorch-PaddlePaddle API 映射列表。

持续更新...

工具类 API 映射列表

梳理了与数据处理、分布式处理等相关的 PyTorch-PaddlePaddle API 映射列表。

持续更新...

视觉类 API 映射列表

梳理了与视觉处理相关的 PyTorch-PaddlePaddle API 映射列表。

持续更新...

2.8 硬件支持

你可以通过以下内容，了解飞桨框架硬件支持相关的内容：

- 飞桨硬件支持：说明飞桨产品支持的硬件。
- 昆仑 XPU 芯片运行飞桨：介绍如何在昆仑 XPU 芯片环境下安装和使用飞桨。
- 海光 DCU 芯片运行飞桨：介绍如何在海光 DCU 芯片环境下安装和使用飞桨。
- 昇腾 NPU 芯片运行飞桨：介绍如何在昇腾环境下安装和使用飞桨。
- Graphcore IPU 芯片运行飞桨：介绍如何在 IPU 环境上安装和使用飞桨。

2.8.1 飞桨产品硬件支持表

飞桨各个产品支持的硬件信息如下：

PaddlePaddle

Paddle Inference

Paddle Lite

注意：如果你想了解更多芯片支持的信息，请联系我们，邮箱为 Paddle-better@baidu.com。

2.8.2 昆仑芯片运行飞桨

百度昆仑 AI 计算处理器（Baidu KUNLUN AI Computing Processor）是百度集十年 AI 产业技术实践于 2019 年推出的全功能 AI 芯片。基于自主研发的先进 XPU 架构，为云端和边缘端的人工智能业务而设计。百度昆仑与飞桨及其他国产软硬件强强组合，打造一个全面领先的国产化 AI 技术生态，部署和应用于诸多“人工智能+”的行业领域，包括智能云和高性能计算，智能制造、智慧城市和安防等。更多昆仑 XPU 芯片详情及技术指标请[点击这里](#)。参考以下内容可快速了解和体验昆仑 XPU 芯片上运行飞桨：

昆仑芯 2 代芯片：

- 飞桨对昆仑 2 代芯片的支持：飞桨支持昆仑 2 代芯片（R200、R300）运行
- 飞桨框架昆仑 2 代芯片安装说明：飞桨框架昆仑 2 代芯片（R200、R300）安装说明
- 飞桨框架昆仑 2 代芯片训练示例：飞桨框架昆仑 2 代芯片（R200、R300）训练示例

昆仑芯 1 代芯片：

- 飞桨对昆仑 1 代芯片的支持：飞桨支持昆仑 1 代芯片（K100、K200）运行
- 飞桨框架昆仑 1 代芯片安装说明：飞桨框架昆仑 1 代芯片（K100、K200）安装说明
- 飞桨框架昆仑 1 代芯片训练示例：飞桨框架昆仑 1 代芯片（K100、K200）训练示例
- 飞桨预测库昆仑 1 代芯片安装及使用：飞桨预测库昆仑 1 代芯片（K100、K200）版安装及使用示例

飞桨对昆仑 2 代芯片的支持

飞桨自 2.3rc 版本起支持在昆仑 2 代芯片上（R200, R300）运行，经验证的模型训练的支持情况如下：

训练支持

可进行单机单卡/单机多卡训练的模型，如下所示：

模型放置在飞桨模型套件中，作为 `github.com/PaddlePaddle` 下的独立 repo 存在，`git clone` 下载即可获取所需的模型文件：

- 注：支持基于 Kernel Primitive 算子的昆仑 2 代芯片支持，[点击这里](#)。

飞桨框架昆仑 2 代芯片安装说明

在昆仑 2 代芯片上，飞桨框架支持基于 `python` 的训练和原生预测，当前最新版本为 2.3rc，提供两种安装方式：

1. 预编译的支持昆仑 2 代芯片的 wheel 包

目前此 wheel 包只支持一种环境：

英特尔 CPU+ 昆仑 2 代芯片 +Linux 操作系统

2. 源码编译安装

其他环境请选择源码编译安装。

安装方式一：通过 Wheel 包安装

下载安装包

环境 1：英特尔 CPU+ 昆仑 2 代芯片 +Linux 操作系统

Linux 发行版建议选择 CentOS 7 系统

Python3.7

```
wget https://paddle-inference-lib.bj.bcebos.com/2.3.0-rc0/python/Linux/XPU2/x86-64_
→gcc8.2_py36_avx_mkl/paddlepaddle_xpu-2.3.0rc0-cp37-cp37m-linux_x86_64.whl
```

```
python3.7 -m pip install -U paddlepaddle_xpu-2.3.0rc0-cp37-cp37m-linux_x86_64.whl
```

验证安装

安装完成后您可以使用 python 或 python3 进入 python 解释器，输入

```
import paddle
```

再输入

```
paddle.utils.run_check()
```

如果出现 PaddlePaddle is installed successfully!，说明您已成功安装。

- 注：支持基于 Kernel Primitive 算子的昆仑 2 代芯片编译 whl 包，[点击这里查看](#)。

安装方式二：从源码编译支持昆仑 XPU 的包

环境准备

英特尔 CPU+ 昆仑 2 代芯片 +CentOS 系统

- 处理器：Intel(R) Xeon(R) Gold 6148 CPU @2.40GHz
- 操作系统：CentOS 7.8.2003（建议使用 CentOS 7）
- Python 版本：3.7 (64 bit)
- pip 或 pip3 版本：9.0.1+ (64 bit)

- **cmake 版本: 3.15+**
- **gcc/g++ 版本: 8.2+**

源码编译安装步骤：

(1) Paddle 依赖 cmake 进行编译构建，需要 cmake 版本 ≥ 3.15 ，如果操作系统提供的源包括了合适版本的 cmake，直接安装即可，否则需要

```
wget https://github.com/Kitware/CMake/releases/download/v3.16.8/cmake-3.16.8.tar.gz
tar -xzf cmake-3.16.8.tar.gz && cd cmake-3.16.8
./bootstrap && make && sudo make install
```

(2) Paddle 内部使用 patchelf 来修改动态库的 rpath，如果操作系统提供的源包括了 patchelf，直接安装即可，否则需要源码安装，请参考

```
./bootstrap.sh
./configure
make
make check
sudo make install
```

(3) 根据 requirements.txt 安装 Python 依赖库

(4) 将 Paddle 的源代码克隆到当下目录下的 Paddle 文件夹中，并进入 Paddle 目录

```
git clone https://github.com/PaddlePaddle/Paddle.git
cd Paddle
```

使用较稳定的版本编译，建议切换到 release2.3 分支下：

```
git checkout release/2.3
```

(5) 进行 Wheel 包的编译，请创建并进入一个叫 build 的目录下

```
mkdir build && cd build
```

具体编译选项含义可参见 [编译选项表](#)

英特尔 CPU+ 昆仑 2 代芯 +CentOS 系统

链接过程中打开文件数较多，可能超过系统默认限制导致编译出错，设置进程允许打开的最大文件数：

```
ulimit -n 4096
```

执行 cmake，完成编译

Python3.7

```
cmake .. -DPY_VERSION=3.7 \
-DCMAKE_BUILD_TYPE=Release \
-DWITH_GPU=OFF \
-DWITH_XPU=ON \
-DON_INFER=ON \
-DWITH_PYTHON=ON \
-DWITH_AVX=ON \
-DWITH_MKL=ON \
-DWITH_MKLDNN=ON \
-DWITH_XPU_BKCL=ON \
-DWITH_DISTRIBUTE=ON \
-DWITH_NCCL=OFF

make -j$(nproc)
```

(6) 编译成功后进入 Paddle/build/python/dist 目录下找到生成的.whl 包。

(7) 将生成的.whl 包 copy 至带有昆仑 XPU 的目标机器上，并在目标机器上根据requirements.txt 安装 Python 依赖库。（如果编译机器同时为带有昆仑 XPU 的目标机器，略过此步）

(8) 在带有昆仑 XPU 的目标机器安装编译好的.whl 包：pip install -U (whl 包的名字) 或 pip3 install -U (whl 包的名字)。恭喜，至此您已完成昆仑 XPU 机器上 PaddlePaddle 的编译安装。

验证安装

安装完成后您可以使用 python 或 python3 进入 python 解释器，输入

```
import paddle
```

再输入

```
paddle.utils.run_check()
```

如果出现 PaddlePaddle is installed successfully!，说明您已成功安装。

如何卸载

使用以下命令卸载 PaddlePaddle：

```
pip uninstall paddlepaddle
```

或

```
pip3 uninstall paddlepaddle
```

- 注：支持基于 Kernel Primitive 算子的昆仑 2 代芯片源码编译，[点击这里查看](#)。

飞桨框架昆仑 2 代芯片训练示例

使用 XPU 训练与 CPU/GPU 相同，只需要简单配置 XPU，就可以执行在昆仑设备上。

ResNet50 下载并运行示例：

1、安装依赖：

```
git clone https://github.com/PaddlePaddle/PaddleClas.git -b develop
cd PaddleClas
python -m pip install -r requirements.txt
```

2、下载数据集：基于 CIFAR100 数据集的 ResNet50 训练任务

```
cd dataset
rm -rf ILSVRC2012
wget -nc https://paddle-imagenet-models-name.bj.bcebos.com/data/whole_chain/whole_
↪chain_CIFAR100.tar
tar xf whole_chain_CIFAR100.tar
ln -s whole_chain_CIFAR100 ILSVRC2012
cd ILSVRC2012
mv train.txt train_list.txt
mv test.txt val_list.txt
```

3、配置 XPU 进行训练的命令非常简单：

```
cd ../..
#FLAGS 指定单卡或多卡训练，此示例运行1个卡
export FLAGS_selected_xpus=2
export XPUSIM_DEVICE_MODEL=KUNLUN2
#启动训练
python tools/train.py \
-c ppcls/configs/ImageNet/ResNet/ResNet50.yaml \
-o Global.device=xpu
```

YOLOv3-DarkNet53 下载并运行示例：

1、安装依赖：

```
git clone https://github.com/PaddlePaddle/PaddleDetection.git -b develop
cd PaddleDetection/
pip install -U pip Cython
pip install -r requirements.txt
```

2、下载数据集

```
cd dataset/voc/  
python download_voc.py  
python create_list.py  
cd ../../
```

3、配置 XPU 进行训练的命令非常简单：

```
#FLAGS指定单卡或多卡训练，此示例运行1个卡  
export FLAGS_selected_xpus=2  
export XPUSIM_DEVICE_MODEL=KUNLUN2  
#启动训练  
python -u tools/train.py \  
-c configs/yolov3/yolov3_darknet53_270e_voc.yml \  
-o use_gpu=false use_xpu=true LearningRate.base_lr=0.000125 \  
--eval
```

飞桨对昆仑 XPU 芯片的支持

飞桨自 2.0 版本起支持在昆仑 XPU 上运行，经验证的模型训练和预测的支持情况如下：

训练支持

可进行单机单卡/单机多卡训练的模型，如下所示：

模型放置在飞桨模型套件中，作为 github.com/PaddlePaddle 下的独立 repo 存在，git clone 下载即可获取所需的模型文件：

预测支持

飞桨框架集成了 python 原生预测功能，安装飞桨框架即可使用。在框架之外，飞桨提供多个高性能预测库，包括 Paddle Inference、Paddle Serving、Paddle Lite 等，支持云边端不同环境下的部署场景，适合相对应的多种硬件平台、操作系统、编程语言，同时提供服务化部署能力。当前预测库验证支持的模型包括：

随着 ARM 架构的高性能、低功耗、低成本的优势日益突显，ARM CPU 更多地进入 PC 和服务器领域，众多新锐国产 CPU 也采用 ARM 架构。在这一趋势下，我们开始尝试在 ARM CPU + 昆仑 XPU 的硬件环境上运行飞桨，当前已验证 ResNet50、YOLOv3 的训练和预测效果。后续版本将持续增加昆仑 XPU 在更多模型任务上的验证。

飞桨框架昆仑 XPU 版安装说明

飞桨框架支持基于 python 的训练和原生预测，当前最新版本为 2.1，提供两种安装方式：

1. 预编译的支持昆仑 XPU 的 wheel 包

目前此 wheel 包只支持两种环境：

英特尔 CPU+ 昆仑 XPU+CentOS 系统

飞腾 CPU+ 昆仑 XPU+ 麒麟 V10 系统

2. 源码编译安装

其他环境请选择源码编译安装。

安装方式一：通过 Wheel 包安装

下载安装包

环境 1：英特尔 CPU+ 昆仑 XPU+CentOS 系统

Linux 发行版建议选择 CentOS 7 系统

Python3.7

```
wget https://paddle-wheel.bj.bcebos.com/kunlun/paddlepaddle-2.1.0-cp37-cp37m-linux_x86_64.whl
```

```
python3.7 -m pip install -U paddlepaddle-2.1.0-cp37-cp37m-linux_x86_64.whl
```

Python3.6

```
wget https://paddle-wheel.bj.bcebos.com/kunlun/paddlepaddle-2.1.0-cp36-cp36m-linux_x86_64.whl
```

```
python3.6 -m pip install -U ``paddlepaddle-2.1.0-cp36-cp36m-linux_x86_64.whl
```

环境 2：飞腾 CPU+ 昆仑 XPU+ 麒麟 V10 系统

如果您想使用预编译的支持昆仑 XPU 的 wheel 包，请联系飞桨官方邮件组：Paddle-better@baidu.com

验证安装

安装完成后您可以使用 python 或 python3 进入 python 解释器，输入

```
import paddle
```

再输入

```
paddle.utils.run_check()
```

如果出现 PaddlePaddle is installed successfully!，说明您已成功安装。

安装方式二：从源码编译支持昆仑 XPU 的包

环境准备

英特尔 CPU+ 昆仑 XPU+CentOS 系统

- 处理器: Intel(R) Xeon(R) Gold 6148 CPU @2.40GHz
- 操作系统: CentOS 7.8.2003 (建议使用 CentOS 7)
- Python 版本: 3.6/3.7 (64 bit)
- pip 或 pip3 版本: 9.0.1+ (64 bit)
- cmake 版本: 3.15+
- gcc/g++ 版本: 8.2+

飞腾 CPU+ 昆仑 XPU+ 麒麟 V10 系统

- 处理器: Phytium,FT-2000+/64
- 操作系统: Kylin release V10 (SP1)/(Tercel)-aarch64-Build04/20200711
- Python 版本: 3.6/3.7 (64 bit)
- pip 或 pip3 版本: 9.0.1+ (64 bit)
- cmake 版本: 3.15+
- gcc/g++ 版本: 8.2+

源码编译安装步骤：

(1) Paddle 依赖 cmake 进行编译构建，需要 cmake 版本 ≥ 3.15 ，如果操作系统提供的源包括了合适版本的 cmake，直接安装即可，否则需要

```
wget https://github.com/Kitware/CMake/releases/download/v3.16.8/cmake-3.16.8.tar.gz
tar -xzf cmake-3.16.8.tar.gz && cd cmake-3.16.8
./bootstrap && make && sudo make install
```

(2) Paddle 内部使用 patchelf 来修改动态库的 rpath，如果操作系统提供的源包括了 patchelf，直接安装即可，否则需要源码安装，请参考

```
./bootstrap.sh
./configure
make
make check
sudo make install
```

(3) 根据 requirements.txt 安装 Python 依赖库

(4) 将 Paddle 的源代码克隆到当下目录下的 Paddle 文件夹中，并进入 Paddle 目录

```
git clone https://github.com/PaddlePaddle/Paddle.git
cd Paddle
```

使用较稳定的版本编译，建议切换到 release2.1 分支下：

```
git checkout release/2.1
```

(5) 进行 Wheel 包的编译，请创建并进入一个叫 build 的目录下

```
mkdir build && cd build
```

具体编译选项含义可参见 [编译选项表](#)

英特尔 CPU+ 昆仑 XPU+CentOS 系统

链接过程中打开文件数较多，可能超过系统默认限制导致编译出错，设置进程允许打开的最大文件数：

```
ulimit -n 2048
```

执行 cmake，完成编译

Python3

```
cmake .. -DPY_VERSION=3.6 \
-DCMAKE_BUILD_TYPE=Release \
```

(下页继续)

(续上页)

```
-DWITH_GPU=OFF \
-DWITH_XPU=ON \
-DON_INFER=ON \
-DWITH_PYTHON=ON \
-DWITH_AVX=ON \
-DWITH_MKL=ON \
-DWITH_MKLDNN=ON \
-DWITH_XPU_BKCL=ON \
-DWITH_DISTRIBUTE=ON \
-DWITH_NCCL=OFF
```

make -j20

Python2

```
cmake .. -DPY_VERSION=2.7 \
-DCMAKE_BUILD_TYPE=Release \
-DWITH_GPU=OFF \
-DWITH_XPU=ON \
-DON_INFER=ON \
-DWITH_PYTHON=ON \
-DWITH_AVX=ON \
-DWITH_MKL=ON \
-DWITH_MKLDNN=ON \
-DWITH_XPU_BKCL=ON \
-DWITH_DISTRIBUTE=ON \
-DWITH_NCCL=OFF
```

make -j20

飞腾 CPU+ 昆仑 XPU+ 麒麟 V10 系统

在该环境下，编译前需要手动拉取 XPU SDK，可使用以下命令：

```
wget https://paddle-wheel.bj.bcebos.com/kunlun/xpu_sdk_v2.0.0.61.tar.gz
tar xvf xpu_sdk_v2.0.0.61.tar.gz
mv output xpu_sdk_v2.0.0.61 xpu_sdk
```

执行 cmake，完成编译

```
ulimit -n 4096
python_exe="/usr/bin/python3.7"
export XPU_SDK_ROOT=$PWD/xpu_sdk
```

(下页继续)

(续上页)

```

cmake .. -DPY_VERSION=3.7 \
          -DPYTHON_EXECUTABLE=$python_exe \
          -DWITH_ARM=ON \
          -DWITH_AARCH64=ON \
          -DWITH_TESTING=OFF \
          -DCMAKE_BUILD_TYPE=Release \
          -DON_INFERENCE=ON \
          -DWITH_XBYAK=OFF \
          -DWITH_XPU=ON \
          -DWITH_GPU=OFF \
          -DWITH_LITE=ON \
          -DLITE_GIT_TAG=release/v2.9 \
          -DXPU_SDK_ROOT=${XPU_SDK_ROOT}

make VERBOSE=1 TARGET=ARMV8 -j32

```

(6) 编译成功后进入 Paddle/build/python/dist 目录下找到生成的.whl 包。

(7) 将生成的.whl 包 copy 至带有昆仑 XPU 的目标机器上，并在目标机器上根据requirements.txt 安装 Python 依赖库。(如果编译机器同时为带有昆仑 XPU 的目标机器，略过此步)

(8) 在带有昆仑 XPU 的目标机器安装编译好的.whl 包： pip install -U (whl 包的名字) 或 pip3 install -U (whl 包的名字)。恭喜，至此您已完成昆仑 XPU 机器上 PaddlePaddle 的编译安装。

验证安装

安装完成后您可以使用 python 或 python3 进入 python 解释器，输入

```
import paddle
```

再输入

```
paddle.utils.run_check()
```

如果出现 PaddlePaddle is installed successfully!，说明您已成功安装。

如何卸载

使用以下命令卸载 PaddlePaddle：

```
pip uninstall paddlepaddle
```

或

```
pip3 uninstall paddlepaddle
```

飞桨框架昆仑 XPU 版训练示例

使用 XPU 训练与 cpu/gpu 相同，只需要加上-o use_xpu=True，表示执行在昆仑设备上。

ResNet50 下载并运行示例：

模型文件下载命令：

```
cd path_to_clone_PaddleClas  
git clone -b release/static https://github.com/PaddlePaddle/PaddleClas.git
```

也可以访问 PaddleClas 的github repo直接下载源码。

配置 XPU 进行训练的命令非常简单：

```
#FLAGS指定单卡或多卡训练，此示例运行2个卡  
export FLAGS_selected_xpus=0,1  
#启动训练  
python3.7 tools/static/train.py -c configs/quick_start/ResNet50_vd_finetune_kunlun.  
→yaml -o use_gpu=False -o use_xpu=True -o is_distributed=False
```

如果需要指定更多的卡（比如 8 卡），需要配置合适的训练参数，可使用如下命令自动修改：

```
export FLAGS_selected_xpus=0,1,2,3,4,5,6,7  
python3.7 -m paddle.distributed.launch \  
    --ips=${ips} \  
    --xpus=${FLAGS_selected_xpus} \  
    --log_dir log \  
    tools/static/train.py \  
    -c ${config_yaml} \  
    -o is_distributed=False \  
    -o epochs=${epochs} \  
    -o TRAIN.batch_size=${total_batch_size} \  
    -o LEARNING_RATE.params.lr=${lr} \  
    -o use_gpu=False \  
    -o use_xpu=True
```

其他模型的训练示例可在飞桨对昆仑 XPU 芯片的支持中支持模型列表下的模型链接中找到。

飞桨预测库昆仑 XPU 版安装及使用示例

在昆仑 XPU 硬件上常用的高性能预测库主要包括以下 3 个，分别适用不同的云边端场景：

Paddle Inference 2.2 版本的安装及使用方式，请[点击查看](#)。

Paddle Serving 0.8.3 版本的安装及使用方式，请[点击查看](#)。

Paddle Lite 2.10 版本的安装及使用方式，请[点击查看](#)。

2.8.3 海光 DCU 芯片运行飞桨

DCU (Deep Computing Unit 深度计算器) 是海光 (HYGON) 推出的一款专门用于 AI 人工智能和深度学习的加速卡。Paddle ROCm 版当前可以支持在海光 CPU 与 DCU 上进行模型训练与预测。

参考以下内容可快速了解和体验在海光芯片上运行飞桨：

- 飞桨框架 ROCm 版支持模型：飞桨框架 ROCm 版支持模型
- 飞桨框架 ROCm 版安装说明：飞桨框架 ROCm 版安装说明
- 飞桨框架 ROCm 版训练示例：飞桨框架 ROCm 版训练示例
- 飞桨框架 ROCm 版预测示例：飞桨框架 ROCm 版预测示例

飞桨框架 ROCm 版支持模型

目前 Paddle ROCm 版基于海光 CPU(X86) 和 DCU 支持以下模型的单机单卡/单机多卡的训练与推理。

[图像分类](#)

[目标检测](#)

[图像分割](#)

[自然语言处理](#)

[字符识别](#)

[推荐系统](#)

[视频分类](#)

[语音合成](#)

生成对抗网络

模型套件

模型放置在飞桨模型套件中，各领域套件是 github.com/PaddlePaddle 下的独立 repo，git clone 下载即可获取所需的模型文件：

飞桨框架 ROCm 版安装说明

飞桨框架 ROCm 版支持基于海光 CPU 和 DCU 的 Python 的训练和原生预测，当前支持的 ROCm 版本为 4.0.1，提供两种安装方式：

- 通过预编译的 wheel 包安装
- 通过源代码编译安装

安装方式一：通过 wheel 包安装

注意：当前仅提供基于 CentOS 7.8 & ROCm 4.0.1 的 docker 镜像，与 Python 3.7 的 wheel 安装包。

第一步：准备 ROCm 4.0.1 运行环境 (推荐使用 Paddle 镜像)

可以直接从 Paddle 的官方镜像库拉取预先装有 ROCm 4.0.1 的 docker 镜像，或者根据 [ROCM 安装文档](#) 来准备相应的运行环境。

```
# 拉取镜像
docker pull paddlepaddle/paddle:latest-dev-rocm4.0-miopen2.11

# 启动容器，注意这里的参数，例如shm-size, device等都需要配置
docker run -it --name paddle-rocm-dev --shm-size=128G \
    --device=/dev/kfd --device=/dev/dri --group-add video \
    --cap-add=SYS_PTRACE --security-opt seccomp=unconfined \
    paddlepaddle/paddle:latest-dev-rocm4.0-miopen2.11 /bin/bash

# 检查容器是否可以正确识别海光DCU设备
rocm-smi

# 预期得到以下结果：
=====
===== ROCm System Management Interface =====
===== Concise Info =====
GPU Temp AvgPwr SCLK MCLK Fan Perf PwrCap VRAM% GPU%
0 50.0c 23.0W 1319Mhz 800Mhz 0.0% auto 300.0W 0% 0%
1 48.0c 25.0W 1319Mhz 800Mhz 0.0% auto 300.0W 0% 0%
2 48.0c 24.0W 1319Mhz 800Mhz 0.0% auto 300.0W 0% 0%
3 49.0c 27.0W 1319Mhz 800Mhz 0.0% auto 300.0W 0% 0%
```

(下页继续)

(续上页)

```
=====
===== End of ROCm SMI Log =====
```

第二步：下载 Python3.7 wheel 安装包

```
pip install --pre paddlepaddle-rocm -f https://www.paddlepaddle.org.cn/whl/rocm/
→develop.html
```

第三步：验证安装包

安装完成之后，运行如下命令。如果出现 PaddlePaddle is installed successfully!，说明已经安装成功。

```
python -c "import paddle; paddle.utils.run_check()"
```

安装方式二：通过源码编译安装

注意：当前 Paddle 只支持 CentOS 7.8 & ROCm 4.0.1 编译环境，且根据 ROCm 4.0.1 的需求，支持的编译器为 devtoolset-7。

第一步：准备 ROCm 4.0.1 编译环境 (推荐使用 Paddle 镜像)

可以直接从 Paddle 的官方镜像库拉取预先装有 ROCm 4.0.1 的 docker 镜像，或者根据 [ROCM 安装文档](#) 来准备相应的运行环境。

```
# 拉取镜像
docker pull paddlepaddle/paddle:latest-dev-rocm4.0-miopen2.11

# 启动容器，注意这里的参数，例如shm-size, device等都需要配置
docker run -it --name paddle-rocm-dev --shm-size=128G \
--device=/dev/kfd --device=/dev/dri --group-add video \
--cap-add=SYS_PTRACE --security-opt seccomp=unconfined \
paddlepaddle/paddle:latest-dev-rocm4.0-miopen2.11 /bin/bash

# 检查容器是否可以正确识别海光DCU设备
rocm-smi

# 预期得到以下结果：
=====
===== ROCm System Management Interface =====
===== Concise Info =====
GPU Temp AvgPwr SCLK MCLK Fan Perf PwrCap VRAM% GPU%
0 50.0c 23.0W 1319Mhz 800Mhz 0.0% auto 300.0W 0% 0%
1 48.0c 25.0W 1319Mhz 800Mhz 0.0% auto 300.0W 0% 0%
2 48.0c 24.0W 1319Mhz 800Mhz 0.0% auto 300.0W 0% 0%
3 49.0c 27.0W 1319Mhz 800Mhz 0.0% auto 300.0W 0% 0%
```

(下页继续)

(续上页)

```
=====
===== End of ROCm SMI Log =====
```

请在编译之前，检查如下的环境变量是否正确，如果没有则需要安装相应的依赖库，并导出相应的环境变量。以 Paddle 官方的镜像举例，环境变量如下：

```
# PATH 与 LD_LIBRARY_PATH 中存在 devtoolset-7，如果没有运行以下命令
source /opt/rh/devtoolset-7/enable

# PATH 中存在 cmake 3.16.0
export PATH=/opt/cmake-3.16/bin:${PATH}

# PATH 与 LD_LIBRARY_PATH 中存在 rocm 4.0.1
export PATH=/opt/rocm/opencl/bin:/opt/rocm/bin:${PATH}
export LD_LIBRARY_PATH=/opt/rocm/lib:${LD_LIBRARY_PATH}

# PATH 中存在 Python 3.7
# 注意：镜像中的 python 3.7 通过 miniconda 安装，请通过 conda activate base
# 命令加载Python 3.7环境
export PATH=/opt/conda/bin:${PATH}
```

第二步：下载 Paddle 源码并编译，CMAKE 编译选项含义请参见编译选项表

```
# 下载源码，默认 develop 分支
git clone https://github.com/PaddlePaddle/Paddle.git
cd Paddle

# 创建编译目录
mkdir build && cd build

# 执行 cmake
cmake .. -DPY_VERSION=3.7 -DWITH_ROCM=ON -DWITH_TESTING=ON -DWITH_DISTRIBUTE=ON \
-DWITH_MKL=ON -DCMAKE_BUILD_TYPE=Release

# 使用以下命令来编译
make -j$(nproc)
```

第三步：安装与验证编译生成的 wheel 包

编译完成之后进入 Paddle/build/python/dist 目录即可找到编译生成的.whl 安装包，安装与验证命令如下：

```
# 安装命令
python -m pip install -U paddlepaddle_rocm-0.0.0-cp37-cp37m-linux_x86_64.whl
```

(下页继续)

(续上页)

```
# 验证命令  
python -c "import paddle; paddle.utils.run_check()"
```

如何卸载

请使用以下命令卸载 Paddle:

```
pip uninstall paddlepaddle-rocm
```

飞桨框架 ROCm 版训练示例

使用海光 CPU/DCU 进行训练与使用 Intel CPU/Nvidia GPU 训练相同，当前 Paddle ROCm 版本完全兼容 Paddle CUDA 版本的 API，直接使用原有的 GPU 训练命令和参数即可。

ResNet50 训练示例

第一步：下载 ResNet50 代码，并准备 ImageNet1k 数据集

```
cd path_to_clone_PaddleClas  
git clone https://github.com/PaddlePaddle/PaddleClas.git
```

也可以访问 PaddleClas 的 [Github Repo](#) 直接下载源码。请根据数据说明文档准备 ImageNet1k 数据集。

第二步：运行训练

```
export HIP_VISIBLE_DEVICES=0,1,2,3  
  
cd PaddleClas/  
python -m paddle.distributed.launch --gpus="0,1,2,3" tools/train.py -c ./ppcls/  
→configs/ImageNet/ResNet/ResNet50.yaml
```

第三步：获取 4 卡训练得到的 Best Top1 Accuracy 结果如下

```
# CUDA 结果为 CUDA 10.1 + 4卡V100 训练  
2021-03-24 01:16:08,548 - INFO - The best top1 acc 0.76332, in epoch: 118  
  
# ROCm 结果为 ROCm 4.0.1 + 4卡DCU 训练  
2021-04-07 10:26:31,651 - INFO - The best top1 acc 0.76308, in epoch: 109
```

YoloV3 训练示例

第一步：下载 YoloV3 代码

```
cd path_to_clone_PaddleDetection  
git clone https://github.com/PaddlePaddle/PaddleDetection.git
```

也可以访问 PaddleDetection 的 Github Repo 直接下载源码。

第二步：准备 VOC 数据集

```
cd PaddleDetection/dataset/voc  
python download_voc.py  
python create_list.py
```

第三步：修改 config 文件的参数

模型 Config 文件 configs/yolov3/yolov3_darknet53_270e_voc.yml 中的默认参数为 8 卡设计，使用 DCU 单机 4 卡训练需要修改参数如下：

```
# 修改 configs/yolov3/_base_/optimizer_270e.yml  
base_lr: 0.0005  
  
# 修改 configs/yolov3/_base_/yolov3_reader.yml  
worker_num: 1
```

第四步：运行训练

```
export HIP_VISIBLE_DEVICES=0,1,2,3  
  
cd PaddleDetection/  
python -m paddle.distributed.launch --gpus 0,1,2,3 tools/train.py -c configs/yolov3/  
→yolov3_darknet53_270e_voc.yml --eval
```

第五步：获取 4 卡训练得到的 mAP 结果如下

```
# CUDA 结果为 CUDA 10.1 + 4卡V100 训练  
[03/23 05:26:17] ppdet.metrics.metrics INFO: mAP (0.50, 11point) = 82.59%  
  
# ROCm 结果为 ROCm 4.0.1 + 4卡DCU 训练  
[03/28 16:02:52] ppdet.metrics.metrics INFO: mAP (0.50, 11point) = 83.02%
```

飞桨框架 ROCm 版预测示例

使用海光 CPU/DCU 进行预测与使用 Intel CPU/Nvidia GPU 预测相同，支持飞桨原生推理库 (Paddle Inference)，适用于高性能服务器端、云端推理。当前 Paddle ROCm 版本完全兼容 Paddle CUDA 版本的 C++/Python API，直接使用原有的 GPU 预测命令和参数即可。

C++ 预测部署

注意：更多 C++ 预测 API 使用说明请参考 [Paddle Inference - C++ API](#)

第一步：源码编译 C++ 预测库

当前 Paddle ROCm 版只支持通过源码编译的方式提供 C++ 预测库。编译环境准备请参考 [飞桨框架 ROCm 版安装说明](#)：通过源码编译安装。

```
# 下载源码，切换到 release/2.1 分支
git clone -b release/2.1 https://github.com/PaddlePaddle/Paddle.git
cd Paddle

# 创建编译目录
mkdir build && cd build

# 执行 cmake，注意这里需打开预测优化选项 ON_INFER
cmake .. -DPY_VERSION=3.7 -DWITH_ROCM=ON -DWITH_TESTING=OFF -DON_INFER=ON \
-DWITH_MKL=ON -DCMAKE_BUILD_TYPE=Release -DCMAKE_EXPORT_COMPILE_COMMANDS=ON

# 使用以下命令来编译
make -j$(nproc)
```

编译完成之后，build 目录下的 paddle_inference_install_dir 即为 C++ 预测库，目录结构如下：

```
build/paddle_inference_install_dir
├── CMakeCache.txt
└── paddle
    ├── include
    │   ├── crypto
    │   ├── experimental
    │   ├── internal
    │   ├── paddle_analysis_config.h
    │   ├── paddle_api.h
    │   ├── paddle_infer_declare.h
    │   ├── paddle_inference_api.h
    │   ├── paddle_mkldnn_quantizer_config.h
    │   ├── paddle_pass_builder.h
    │   └── paddle_tensor.h
```

C++ 预测库头文件目录

C++ 预测库头文件

(下页继续)

(续上页)

```

|   └── lib
|       ├── libpaddle_inference.a          C++ 静态预测库文件
|       └── libpaddle_inference.so        C++ 动态态预测库文件
└── third_party
    ├── install                         第三方链接库和头文件
    |   ├── cryptopp
    |   ├── gflags
    |   ├── glog
    |   ├── mkldnn
    |   ├── mklml
    |   ├── protobuf
    |   └── xxhash
    └── threadpool
        └── ThreadPool.h
└── version.txt

```

其中 `version.txt` 文件中记录了该预测库的版本信息，包括 Git Commit ID、使用 OpenBlas 或 MKL 数学库、ROCM/MIOPEN 版本号，如：

```

GIT COMMIT ID: e75412099f97a49701324788b468d80391293ea9
WITH_MKL: ON
WITH_MKLDNN: ON
WITH_GPU: OFF
WITH_ROCM: ON
HIP version: 4.0.20496-4f163c68
MIOpen version: v2.11
CXX compiler version: 7.3.1

```

第二步：准备预测部署模型

下载 ResNet50 模型后解压，得到 Paddle 预测格式的模型，位于文件夹 ResNet50 下。如需查看模型结构，可将 `inference.pdmodel` 文件通过模型可视化工具 Netron 打开。

```

wget https://paddle-inference-dist.bj.bcebos.com/Paddle-Inference-Demo/resnet50.tgz
tar zxf resnet50.tgz

# 获得模型目录即文件如下
resnet50/
├── inference.pdmodel
├── inference.pdiparams.info
└── inference.pdiparams

```

第三步：获取预测示例代码并编译运行

预先要求：

本章节 C++ 预测示例代码位于 `Paddle-Inference-Demo/c++/resnet50`。

请先将示例代码下载到本地，再将第一步中编译得到的 `paddle_inference_install_dir` 重命名为 `paddle_inference` 文件夹，移动到示例代码的 `Paddle-Inference-Demo/c++/lib` 目录下。使用到的文件如下所示：

<code>-rw-r--r-- 1 root root 3479 Jun 2 03:14 README.md</code>	README 说明
<code>-rw-r--r-- 1 root root 3051 Jun 2 03:14 resnet50_test.cc</code>	预测 C++ 源码程序
<code>drwxr-xr-x 2 root root 4096 Mar 5 07:43 resnet50</code>	→ 第二步中下载并解压的预测部署模型文件夹
<code>-rw-r--r-- 1 root root 387 Jun 2 03:14 run.sh</code>	运行脚本
<code>-rwxr-xr-x 1 root root 1077 Jun 2 03:14 compile.sh</code>	编译脚本
<code>-rw-r--r-- 1 root root 9032 Jun 2 07:26 ../lib/CMakeLists.txt</code>	CMAKE 文件
<code>drwxr-xr-x 1 root root 9032 Jun 2 07:26 ../lib/paddle_inference</code>	第一步编译的到的 → Paddle Infernece C++ 预测库文件夹

编译运行预测样例之前，需要根据运行环境配置编译脚本 `compile.sh`。

```
# 根据预编译库中的 version.txt 信息判断是否将以下标记打开
WITH_MKL=ON
WITH_GPU=OFF # 注意这里需要关掉 WITH_GPU
USE_TENSORRT=OFF

WITH_ROCM=ON # 注意这里需要打开 WITH_ROCM
ROCM_LIB=/opt/rocm/lib
```

运行 `run.sh` 脚本进行编译和运行，即可获取最后的预测结果：

```
bash run.sh

# 成功执行之后，得到的预测输出结果如下：
...
I0602 04:12:03.708333 52627 analysis_predictor.cc:595] ===== optimize end =====
I0602 04:12:03.709321 52627 naive_executor.cc:98] --- skip [feed], feed -> inputs
I0602 04:12:03.710139 52627 naive_executor.cc:98] --- skip [save_infer_model/scale_0.
→tmp_1], fetch -> fetch
I0602 04:12:03.711813 52627 device_context.cc:624] oneDNN v2.2.1
WARNING: Logging before InitGoogleLogging() is written to STDERR
I0602 04:12:04.106405 52627 resnet50_test.cc:73] run avg time is 394.801 ms
I0602 04:12:04.106503 52627 resnet50_test.cc:88] 0 : 0
I0602 04:12:04.106525 52627 resnet50_test.cc:88] 100 : 2.04163e-37
I0602 04:12:04.106552 52627 resnet50_test.cc:88] 200 : 2.1238e-33
I0602 04:12:04.106573 52627 resnet50_test.cc:88] 300 : 0
I0602 04:12:04.106591 52627 resnet50_test.cc:88] 400 : 1.6849e-35
I0602 04:12:04.106603 52627 resnet50_test.cc:88] 500 : 0
```

(下页继续)

(续上页)

```
I0602 04:12:04.106618 52627 resnet50_test.cc:88] 600 : 1.05767e-19
I0602 04:12:04.106643 52627 resnet50_test.cc:88] 700 : 2.04094e-23
I0602 04:12:04.106670 52627 resnet50_test.cc:88] 800 : 3.85254e-25
I0602 04:12:04.106683 52627 resnet50_test.cc:88] 900 : 1.52391e-30
```

Python 预测部署示例

注意：更多 Python 预测 API 使用说明请参考 Paddle Inference - Python API

第一步：安装 Python 预测库

Paddle ROCm 版的 Python 预测库请参考 飞桨框架 ROCm 版安装说明 进行安装或编译。

第二步：准备预测部署模型

下载 ResNet50 模型后解压，得到 Paddle 预测格式的模型，位于文件夹 ResNet50 下。如需查看模型结构，可将 inference.pdmodel 文件通过模型可视化工具 Netron 打开。

```
wget https://paddle-inference-dist.bj.bcebos.com/Paddle-Inference-Demo/resnet50.tgz
tar zxf resnet50.tgz

# 获得模型目录即文件如下
resnet50/
├── inference.pdmodel
├── inference.pdiparams.info
└── inference.pdiparams
```

第三步：准备预测部署程序

将以下代码保存为 python_demo.py 文件：

```
import argparse
import numpy as np

# 引用 paddle inference 预测库
import paddle.inference as paddle_infer

def main():
    args = parse_args()

    # 创建 config
    config = paddle_infer.Config(args.model_file, args.params_file)

    # 根据 config 创建 predictor
    predictor = paddle_infer.create_predictor(config)
```

(下页继续)

(续上页)

```

# 获取输入的名称
input_names = predictor.get_input_names()
input_handle = predictor.get_input_handle(input_names[0])

# 设置输入
fake_input = np.random.randn(args.batch_size, 3, 318, 318).astype("float32")
input_handle.reshape([args.batch_size, 3, 318, 318])
input_handle.copy_from_cpu(fake_input)

# 运行predictor
predictor.run()

# 获取输出
output_names = predictor.get_output_names()
output_handle = predictor.get_output_handle(output_names[0])
output_data = output_handle.copy_to_cpu() # numpy.ndarray类型
print("Output data size is {}".format(output_data.size))
print("Output data shape is {}".format(output_data.shape))

def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument("--model_file", type=str, help="model filename")
    parser.add_argument("--params_file", type=str, help="parameter filename")
    parser.add_argument("--batch_size", type=int, default=1, help="batch size")
    return parser.parse_args()

if __name__ == "__main__":
    main()

```

第四步：执行预测程序

```

# 参数输入为本章节第2步中下载的 ResNet50 模型
python python_demo.py --model_file ./resnet50/inference.pdmodel \
                      --params_file ./resnet50/inference.pdiparams \
                      --batch_size 2

# 成功执行之后，得到的预测输出结果如下：
...
I0602 04:14:13.455812 52741 analysis_predictor.cc:595] ===== optimize end =====
I0602 04:14:13.456934 52741 naive_executor.cc:98] --- skip [feed], feed -> inputs
I0602 04:14:13.458562 52741 naive_executor.cc:98] --- skip [save_infer_model/scale_0.
˓→tmp_1], fetch -> fetch
Output data size is 2000

```

(下页继续)

(续上页)

```
Output data shape is (2, 1000)
```

2.8.4 昇腾 NPU 芯片运行飞桨

华为昇腾 910 (Ascend 910) 是一款具有超高算力的 AI 处理器。Paddle NPU 版当前可以支持在华为鲲鹏 CPU 与昇腾 NPU 上进行模型训练与推理。

参考以下内容可快速了解和体验在昇腾芯片上运行飞桨：

- 飞桨框架昇腾 NPU 版安装说明：飞桨框架昇腾 NPU 版安装说明
- 飞桨框架昇腾 NPU 版训练示例：飞桨框架昇腾 NPU 版训练示例

飞桨框架昇腾 NPU 版安装说明

飞桨框架 NPU 版支持基于华为鲲鹏 CPU 与昇腾 NPU 的 Python 的训练和原生推理。

环境准备

当前 Paddle 昇腾 910 NPU 版支持的华为 CANN 社区版 5.0.2.alpha005，请先根据华为昇腾 910 NPU 的要求，进行相关 NPU 运行环境的部署和配置，参考华为官方文档 [CANN 社区版安装指南](#)。

Paddle 昇腾 910 NPU 版目前仅支持源码编译安装，其中编译与运行相关的环境要求如下：

- **CPU 处理器:** 鲲鹏 920
- **操作系统:** Ubuntu 18.04 / CentOS 7.6 / KylinV10SP1 / EulerOS 2.8
- **CANN 社区版:** 5.0.2.alpha005
- **Python 版本:** 3.7
- **Cmake 版本:** 3.15+
- **GCC/G++ 版本:** 8.2+

安装方式：通过源码编译安装

第一步：准备 CANN 社区版 5.0.2.alpha005 运行环境 (推荐使用 Paddle 镜像)

可以直接从 Paddle 的官方镜像库拉取预先装有 CANN 社区版 5.0.2.alpha005 的 docker 镜像，或者根据 [CANN 社区版安装指南](#) 来准备相应的开发与运行环境。

```
# 拉取镜像
docker pull paddlepaddle/paddle:latest-dev-cann5.0.2.alpha005-gcc82-aarch64

# 启动容器，注意这里的参数 --
→device, 容器仅映射设备ID为4到7的4张NPU卡，如需映射其他卡相应增改设备ID号即可
docker run -it --name paddle-npu-dev -v /home/<user_name>:/workspace \
    --pids-limit 409600 --network=host --shm-size=128G \
    --cap-add=SYS_PTRACE --security-opt seccomp=unconfined \
    --device=/dev/davinci4 --device=/dev/davinci5 \
    --device=/dev/davinci6 --device=/dev/davinci7 \
    --device=/dev/davinci_manager \
    --device=/dev/devmm_svm \
    --device=/dev/hisi_hdc \
    -v /usr/local/Ascend/driver:/usr/local/Ascend/driver \
    -v /usr/local/bin/npu-smi:/usr/local/bin/npu-smi \
    -v /usr/local/dcmi:/usr/local/dcmi \
    paddlepaddle/paddle:latest-dev-cann5.0.2.alpha005-gcc82-aarch64 /bin/bash

# 检查容器中是否可以正确识别映射的昇腾DCU设备
npu-smi info

# 预期得到类似如下的结果
+-----+
| npu-smi 1.9.3           Version: 21.0.rc1 |
+-----+-----+-----+
| NPU   Name      | Health     | Power (W)  Temp (C) |
| Chip          | Bus-Id     | AICore (%) | Memory-Usage (MB) | HBM-Usage (MB) |
+=====+=====+=====+=====+=====+=====+
| 4    910A       | OK         | 67.2       30          |
| 0              | 0000:C2:00.0 | 0          303 / 15171    0 / 32768   |
+=====+=====+=====+=====+=====+
| 5    910A       | OK         | 63.8       25          |
| 0              | 0000:82:00.0 | 0          2123 / 15171   0 / 32768   |
+=====+=====+=====+=====+=====+
| 6    910A       | OK         | 67.1       27          |
| 0              | 0000:42:00.0 | 0          1061 / 15171   0 / 32768   |
+=====+=====+=====+=====+=====+
| 7    910A       | OK         | 65.5       30          |
| 0              | 0000:02:00.0 | 0          2563 / 15078   0 / 32768   |
+=====+=====+=====+=====+=====+
```

第二步：下载 Paddle 源码并编译，CMAKE 编译选项含义请参见[编译选项表](#)

```
# 下载源码
```

(下页继续)

(续上页)

```
git clone https://github.com/PaddlePaddle/Paddle.git
cd Paddle

# 创建编译目录
mkdir build && cd build

# 执行cmake
cmake .. -DPY_VERSION=3.7 -DWITH_ASCEND=OFF -DWITH_ARM=ON -DWITH_ASCEND_CL=ON \
-DWITH_ASCEND_INT64=ON -DWITH_DISTRIBUTE=ON -DWITH_TESTING=ON -DON_INFERENCE=ON \
-DCMAKE_BUILD_TYPE=Release -DCMAKE_EXPORT_COMPILE_COMMANDS=ON

# 使用以下命令来编译
make TARGET=ARMV8 -j$(nproc)
```

第三步：安装与验证编译生成的 wheel 包

编译完成之后进入 Paddle/build/python/dist 目录即可找到编译生成的.whl 安装包，安装与验证命令如下：

```
# 安装命令
python -m pip install -U paddlepaddle_npu-0.0.0-cp37-cp37m-linux_aarch64.whl

# 验证命令
python -c "import paddle; paddle.utils.run_check()"

# 预期得到类似以下结果：
Running verify PaddlePaddle program ...
PaddlePaddle works well on 1 NPU.
PaddlePaddle works well on 4 NPUs.
PaddlePaddle is installed successfully! Let's start deep learning with PaddlePaddle ↵
now.
```

如何卸载

请使用以下命令卸载 Paddle:

```
pip uninstall paddlepaddle-npu
```

飞桨框架昇腾 NPU 版训练示例

YOLOv3 训练示例

第一步：下载并安装 PaddleDetection 套件

```
# 下载套件代码
cd path_to_clone_PaddleDetection
git clone -b develop https://github.com/PaddlePaddle/PaddleDetection.git

# 编译安装
cd PaddleDetection
python setup.py install

# 安装其他依赖
pip install -r requirements.txt
```

也可以访问 PaddleDetection 的 [Github Repo](#) 下载 develop 分支的源码。

第二步：准备 VOC 训练数据集

```
cd PaddleDetection/static/dataset/roadsign_voc
python download_roadsign_voc.py

# 下载完成之后，当前目录结构如下
PaddleDetection/static/dataset/roadsign_voc/
├── annotations
├── download_roadsign_voc.py
├── images
├── label_list.txt
├── train.txt
└── valid.txt
```

第三步：运行单卡训练

```
export FLAGS_selected_npus=0

# 单卡训练
python -u tools/train.py -c configs/yolov3_darknet_roadsign.yml -o use_npu=True

# 单卡评估
python -u tools/eval.py -c configs/yolov3_darknet_roadsign.yml -o use_npu=True

# 精度结果
INFO:ppdet.utils.voc_eval:mAP(0.50, integral) = 76.78%
```

第四步：运行多卡训练

注意：多卡训练请参考本页下一章节进行“NPU 多卡训练配置”的准备。

```
# NPU 多卡训练配置
export FLAGS_selected_npus=0,1,2,3
export RANK_TABLE_FILE=/root/hccl_4p_0123_127.0.0.1.json

# 设置 HCCL 相关环境变量
export HCCL_CONNECT_TIMEOUT=7200
export HCCL_WHITELIST_DISABLE=1
export HCCL_SECURITY_MODE=1

# 多卡训练
python -m paddle.distributed.fleet.launch --run_mode=collective \
    tools/train.py -c configs/yolov3_darknet_roadsign.yml -o use_npu=True

# 多卡训练结果评估
python -u tools/eval.py -c configs/yolov3_darknet_roadsign.yml -o use_npu=True

# 精度结果
INFO:ppdet.utils.voc_eval:mAP(0.50, integral) = 83.00%
```

NPU 多卡训练配置

预先要求：请先根据华为昇腾 910 NPU 的文档 [配置 device 的网卡 IP](#) 进行相关 NPU 运行环境的部署和配置，配置完成后检查机器下存在 /etc/hccn.conf 文件。

如果是物理机环境，请根据华为官网的 [hccl_tools 说明文档](#) 进行操作。如果是根据 Paddle 官方镜像启动的容器环境，请根据以下步骤进行操作：

第一步：根据容器启动时映射的设备 ID，创建容器内的 /etc/hccn.conf 文件

例如物理机上的 8 卡的原始 /etc/hccn.conf 文件内容如下：

```
address_0=192.168.10.21
netmask_0=255.255.255.0
address_1=192.168.20.21
netmask_1=255.255.255.0
address_2=192.168.30.21
netmask_2=255.255.255.0
address_3=192.168.40.21
netmask_3=255.255.255.0
address_4=192.168.10.22
netmask_4=255.255.255.0
```

(下页继续)

(续上页)

```
address_5=192.168.20.22
netmask_5=255.255.255.0
address_6=192.168.30.22
netmask_6=255.255.255.0
address_7=192.168.40.22
netmask_7=255.255.255.0
```

容器启动命令中映射的设备 ID 为 4 到 7 的 4 张 NPU 卡，则创建容器内的 /etc/hccn.conf 文件内容如下：

注意：这里的 address_4 和 netmask_4 需要相应的修改为 address_0 和 netmask_0，以此类推

```
address_0=192.168.10.22
netmask_0=255.255.255.0
address_1=192.168.20.22
netmask_1=255.255.255.0
address_2=192.168.30.22
netmask_2=255.255.255.0
address_3=192.168.40.22
netmask_3=255.255.255.0
```

第二步：根据华为官网的 hccl_tools 说明文档，生成单机四卡的配置文件

```
# 下载 hccl_tools.py 文件到本地
wget https://raw.githubusercontent.com/mindspore-ai/mindspore/v1.4.0/model_zoo/utils/
↳hccl_tools/hccl_tools.py

# 生成单机两卡的配置文件，单机可以设置 IP 为 127.0.0.1
python hccl_tools.py --device_num "[0,4]" --server_ip 127.0.0.1
```

运行成功之后在当前目录下获得名为 hccl_4p_0123_127.0.0.1.json 的文件，内容如下：

```
{
  "version": "1.0",
  "server_count": "1",
  "server_list": [
    {
      "server_id": "127.0.0.1",
      "device": [
        {
          "device_id": "0",
          "device_ip": "192.168.10.22",
          "rank_id": "0"
        },
        {
          "device_id": "1",
          "device_ip": "192.168.10.22",
          "rank_id": "1"
        }
      ]
    }
  ]
}
```

(下页继续)

(续上页)

```
{
    {
        "device_id": "1",
        "device_ip": "192.168.20.22",
        "rank_id": "1"
    },
    {
        "device_id": "2",
        "device_ip": "192.168.30.22",
        "rank_id": "2"
    },
    {
        "device_id": "3",
        "device_ip": "192.168.40.22",
        "rank_id": "3"
    }
],
"host_nic_ip": "reserve"
}
],
"status": "completed"
}
```

第三步：运行 Paddle 多卡训练之前，需要先配置名为 RANK_TABLE_FILE 的环境变量，指向上一步生成的 json 文件的绝对路径

```
# 1) 设置 ranktable 文件的环境变量
export RANK_TABLE_FILE=$(readlink -f hccl_4p_0123_127.0.0.1.json)
# 或者直接修改为 json 文件的绝对路径
export RANK_TABLE_FILE=/root/hccl_4p_0123_127.0.0.1.json

# 2) 设置 HCCL 相关环境变量
export HCCL_CONNECT_TIMEOUT=7200
export HCCL_WHITELIST_DISABLE=1
export HCCL_SECURITY_MODE=1

# 3) 启动分布式任务，注意这里的 run_mode 当前仅支持 collective 模式
python -m paddle.distributed.fleet.launch --run_mode=collective train.py ...
```

2.8.5 Graphcore IPU 芯片运行飞桨

Title overline too short.

```
#####
Graphcore IPU芯片运行飞桨
#####
```

IPU 是 Graphcore 推出的用于 AI 计算的专用芯片，Paddle IPU 版可以支持在 IPU 上进行模型训练与预测。

参考以下内容了解和体验在 IPU 芯片上运行飞桨：

- 飞桨框架 IPU 版安装说明：飞桨框架 IPU 版安装说明
- 飞桨框架 IPU 版训练示例：飞桨框架 IPU 版训练示例
- 飞桨框架 IPU 版预测示例：飞桨框架 IPU 版预测示例

飞桨框架 IPU 版安装说明

飞桨框架 IPU 版支持基于 IPU 的 Python 的训练和原生推理，目前仅支持通过源代码编译安装。

通过源代码编译安装

建议在 Docker 环境内编译和使用飞桨框架 IPU 版，下面的说明将使用基于 Ubuntu18.04 的容器进行编译，使用的 Python 版本为 Python3.7。

第一步构建 Docker 镜像

```
# 下载源码
git clone https://github.com/PaddlePaddle/Paddle.git
cd Paddle

# 构建 Docker 镜像
docker build -t paddlepaddle/paddle:latest-dev-ipu \
-f tools/dockerfile/Dockerfile.ipu .
```

第二步下载 Paddle 源码并编译

```
# 创建并运行 Docker 容器
# 需要将主机端的 ipuof 配置文件映射到容器中，可通过设置 IPUOF_CONFIG_PATH_
˓→ 环境变量指向 ipuof 配置文件传入
# 更多关于 ipuof 配置的信息可访问 https://docs.graphcore.ai/projects/vipu-admin/en/
˓→ latest/cli_reference.html?highlight=ipuof#ipuof-configuration-file
docker run --ulimit memlock=-1:-1 --net=host --cap-add=IPC_LOCK \
--device=/dev/infiniband/ --ipc=host \
```

(下页继续)

(续上页)

```
--name paddle-dev-ipu -w /home \
-v ${IPUOF_CONFIG_PATH}:/ipuof.conf \
-e IPUOF_CONFIG_PATH=/ipuof.conf \
-it paddlepaddle/paddle:latest-dev-ipu bash

# 容器内下载源码
git clone https://github.com/PaddlePaddle/Paddle.git
cd Paddle

# 创建编译目录
mkdir build && cd build

# 执行 CMake
cmake .. -DWITH_IPU=ON -DWITH_PYTHON=ON -DPY_VERSION=3.7 -DWITH_MKL=ON \
-DPOPLAR_DIR=/opt/poplar -DPOPART_DIR=/opt/popart -DCMAKE_BUILD_TYPE=Release

# 开始编译
make -j$(nproc)

# 安装编译生成的 wheel 包
pip install -U python/dist/paddlepaddle-0.0.0-cp37-cp37m-linux_x86_64.whl
```

第三步验证安装

```
python -c "import paddle; paddle.utils.run_check()"
```

飞桨框架 IPU 版训练示例

BERT-Base 训练示例

示例将默认用户已安装飞桨框架 IPU 版，并且已经配置运行时需要的环境（建议在 Docker 环境中使用飞桨框架 IPU 版）。

示例代码位于 [Paddle-BERT with Graphcore IPUs](#)

第一步：下载源码并安装依赖

```
# 下载源码
git clone https://github.com/PaddlePaddle/PaddleNLP.git
cd model_zoo/bert/static_ipu/

# 安装依赖
pip install -r requirements.txt
```

第二步：准备数据集

按照 README.md 的描述准备用于预训练的数据集。

第三步：执行模型训练

按照 README.md 的描述开始 BERT-Base 模型的预训练和在 SQuAD v1.1 数据集上的模型微调。

飞桨框架 IPU 版预测示例

飞桨框架 IPU 版支持飞桨原生推理库 (Paddle Inference)，适用于云端推理。

C++ 预测示例

第一步：源码编译 C++ 预测库

当前 Paddle IPU 版只支持通过源码编译的方式提供 C++ 预测库，编译环境准备请参考 [飞桨框架 IPU 版安装说明](#)。

```
# 下载源码
git clone https://github.com/PaddlePaddle/Paddle.git
cd Paddle

# 创建编译目录
mkdir build && cd build

# 执行 CMake，注意这里需打开预测优化选项 ON_INFER
cmake .. -DWITH_IPU=ON -DWITH_PYTHON=ON -DPY_VERSION=3.7 -DWITH_MKL=ON -DON_INFER=ON \
-DPOPLAR_DIR=/opt/poplar -DPOPART_DIR=/opt/popart -DCMAKE_BUILD_TYPE=Release

# 开始编译
make -j$(nproc)
```

成功编译后，C++ 预测库将存放于 build/paddle_inference_install_dir 目录下。

第二步：获取预测示例代码并编译运行

```
# 获取示例代码
git clone https://github.com/PaddlePaddle/Paddle-Inference-Demo
```

将获得的 C++ 预测库拷贝并重命名一份到 Paddle-Inference-Demo/c++/lib/paddle_inference。

```
cd Paddle-Inference-Demo/c++/paddle-ipu

# 编译
bash ./compile.sh
```

(下页继续)

(续上页)

```
# 运行  
bash ./run.sh
```

2.9 自定义算子

介绍如何使用飞桨的自定义算子（Operator，简称 Op）机制，包括以下两类：

1. C++ 算子：编写方法较为简洁，不涉及框架内部概念，无需重新编译飞桨框架，以外接模块的方式使用的算子
2. Python 算子：使用 Python 编写实现前向（forward）和反向（backward）方法，在模型组网中使用的自定义 API
 - 自定义 C++ 算子
 - 自定义 Python 算子

2.9.1 自定义 C++ 算子

概述

算子（Operator，简称 Op）是构建神经网络的基础组件，飞桨框架提供了丰富的算子库，能够满足绝大多数场景的使用需求。但是出于以下几点原因，您可能希望定制化算子的 C++ 实现，从而满足特定需求：

1. 已有的算子无法组合出您需要的运算逻辑；
2. 使用已有算子组合得到的运算逻辑无法满足您的性能需求。

为此，我们提供了自定义外部算子的机制，以此机制实现的自定义算子，能够以即插即用的方式用于模型训练与推理，不需要重新编译安装飞桨框架。

使用自定义算子机制，仅需要以下两个步骤：

1. 实现算子的 C++ 运算逻辑，完成算子构建
2. 调用 python 接口完成算子编译与注册

随后即可在模型中使用，下面通过实现一个 relu 运算，介绍具体的实现、编译与应用流程。

注意事项：

- 在使用本机制实现自定义算子之前，请确保已经正确安装了 PaddlePaddle 2.3 及以上版本
- 该机制已支持 Linux、Mac 和 Windows 平台。

- 本自定义外部算子机制仅保证源码级别的兼容，不保证二进制级别的兼容，例如，基于飞桨 2.3 版本编写的自定义算子源码实现，在飞桨 2.3 或者后续版本中编译链接使用没有问题，但基于飞桨 2.3 之前的版本编译得到的自定义算子动态库文件（*.so, *.dylib, *.dll），在 2.3 或者后续发布的版本中可能会加载失败。

自定义算子 C++ 实现

使用自定义算子机制，需要编写以下组件的 C++ 实现，包括：

- 算子的运算函数**：算子核心的计算逻辑实现，主要是对输入 Tensor 进行处理，得到输出 Tensor 的过程
- 算子的维度与类型推导函数**：用于在组网编译和运行时，正确推导出输出 Tensor 的 shape 和 data type
- 算子构建**：描述算子的输入输出信息、并关联前述运算、维度推导与类型推导函数

下面结合示例进行介绍。

运算函数与基础 API

基本写法要求

在编写运算函数之前，需要引入 PaddlePaddle 扩展头文件，示例如下：

```
#include "paddle/extension.h"
```

算子运算函数有特定的函数写法要求，在编码过程中需要遵守，基本形式如下：

```
std::vector<paddle::Tensor> OpFunction(const paddle::Tensor& x, ..., int attr, ...){  
    ...  
}
```

- 函数输入参数可以是 paddle::Tensor , std::vector<paddle::Tensor> 或者一些基础类型的 Attribute ，具体地：
 - paddle::Tensor 需要以 const paddle::Tensor& 的形式作为输入，可以有一个或多个
 - std::vector<paddle::Tensor> 需要以 const std::vector<paddle::Tensor>& 的形式作为输入，可以有一个或多个
 - Attribute 目前仅支持如下数据类型，建议按如下形式作为输入，可以有一个或多个：
 - * bool
 - * int
 - * float

```
* int64_t
* const std::string&
* const std::vector<int>&
* const std::vector<float>&
* const std::vector<int64_t>&
* const std::vector<std::string>&
```

- 函数返回值只能是 `std::vector<paddle::Tensor>`

注：其他类型的数值作为函数输入参数或者返回值将无法编译通过

设备类型

设备类型使用 `Place` 表示，`Place` 含有内存类型 `AllocationType` 与设备 ID 信息，是 `Tensor` 的基础描述信息之一。

其中设备类型是枚举类型：

```
enum class AllocationType : int8_t {
    UNDEFINED = 0,
    CPU = 1,
    GPU = 2,
    GPUPINNED = 3,
    ...
};
```

设备 ID 是一个 `int8_t` 的数值，用于表示当前使用的设备卡号。

一些 `Place` 使用示例如下：

```
auto cpu_place = paddle::CPUPlace();
auto gpu_place = paddle::GPUPlace(); //_
// 默认设备ID为0，一般在自定义算子内使用默认的构造方式即可
auto gpu_place = paddle::GPUPlace(1); // GPU 1号卡
```

此外，`Place` 还有两个常用的方法：

- `GetType()`: 获取 `Place` 的内存类型 `AllocationType`
- `GetDeviceId()`: 获取 `Place` 的设备 ID

使用示例如下：

```

auto gpu_place = paddle::GPUPlace();
auto alloc_type = gpu_place.GetType(); // paddle::AllocationType::GPU
auto dev_id = gpu_place.GetDeviceId(); // 0

```

详细的 Place 定义请参考 [paddle/phi/common/place.h](#)。

注：目前自定义算子仅在 CPU 与 GPU 上进行了验证，其他类型会视需求在后续版本支持

数据类型

数据类型使用 DataType 表示，同样是 Tensor 的基础描述信息之一，目前主要支持的类型如下：

```

enum class DataType {
    UNDEFINED = 0,
    BOOL,
    INT8,
    UINT8,
    INT16,
    INT32,
    UINT32,
    INT64,
    UINT64,
    BFLOAT16,
    FLOAT16,
    UINT16,
    FLOAT32,
    FLOAT64,
    COMPLEX64,
    COMPLEX128,
    ...
}

```

详细的 DataType 定义请参考 [paddle/phi/common/data_type.h](#)。

Tensor API

(1) Tensor 构造

对于 paddle::Tensor 的构造，我们推荐使用相应的初始化 paddle API，包括：

```

PADDLE_API Tensor empty(const IntArray& shape, DataType dtype=DataType::FLOAT32,_
    ↵const Place& place=CPUPlace());
PADDLE_API Tensor full(const IntArray& shape, const Scalar& value, DataType_
    ↵dtype=DataType::FLOAT32, const Place& place=CPUPlace());

```

(下页继续)

(续上页)

```
PADDLE_API Tensor empty_like(const Tensor& x, DataType dtype=DataType::UNDEFINED,  
                           const Place& place={});  
PADDLE_API Tensor full_like(const Tensor& x, const Scalar& value, DataType  
                           dtype=DataType::UNDEFINED, const Place& place={});
```

使用示例如下：

```
auto tensor = paddle::empty({3, 4}); // default: float32, cpu  
auto tensor = paddle::full({3, 4}, 1.0); // default: float32, cpu  
auto gpu_tensor = paddle::empty({3, 4}, paddle::DataType::FLOAT64,  
                                paddle::GPUPlace());  
auto gpu_tensor = paddle::full({3, 4}, 1.0, paddle::DataType::FLOAT64,  
                                paddle::GPUPlace());
```

(2) Tensor 成员方法

此外 paddle::Tensor 自身目前提供了一些基础的功能 API，在定义算子最后那个常用的包括：

- 设备、数据类型获取 API：
 - const Place& place() const: 获取 Tensor 所在的设备
 - DataType dtype() const: 获取 Tensor 的数据类型
- 长度与维度获取 API：
 - int64_t numel() const: 获取 Tensor 的数据长度
 - std::vector<int64_t> shape() const: 获取 Tensor 的维度信息
- 数据访问 API：
 - template <typename T> const T* data() const: 模板类方法，获取数据内存的起始地址（只读）
 - template <typename T> T* data(): 模板类方法，获取数据内存的起始地址（读写）
- 状态或属性判断 API：
 - bool defined() const: 确认 Tensor 是否有效
 - bool initialized() const: 确认 Tensor 是否已被初始化
 - bool is_cpu() const: 确认 Tensor 是否在 CPU 上
 - bool is_gpu() const: 确认 Tensor 是否在 GPU 上
- 工具类 API：
 - Tensor copy_to(const Place& place, bool blocking) const:
 - * 模板类方法，输入参数 place，将当前 Tensor 拷贝到指定设备上并返回

- Tensor cast(DataType target_type) const:
 - * 输入参数 target_type，将当前 Tensor 转换为指定数据类型的 Tensor 并返回
- Tensor slice(const int64_t begin_idx, const int64_t end_idx) const:
 - * 输入参数起始行 begin_idx 和终止行 end_idx，返回当前 Tensor 从起始行（含）到终止行（不含）的一个视图
 - * 目前仅支持对当前 Tensor 的第一个维度（即 axis = 0）进行切分
- cudaStream_t stream() const:
 - * 用于获取当前 Tensor 所处的 CUDA Stream（仅在 GPU 编译版本中生效）
 - * 仅能够获取函数输入 Tensor 的 stream

后续我们会继续扩展其他 Tensor API，详细的 Tensor 定义请参考 [paddle/phi/api/include/tensor.h](#)。

Exception API

- PD_CHECK(COND, ...): 输入 bool 条件表达式进行检查，如果值为 false，则抛出异常，支持变长参数输入，伪代码示例如下：

```
// case 1: No error message specified
PD_CHECK(a > b)
// The key error message like:
// Expected a > b, but it is not satisfied.
// [/User/custom_op/custom_relu_op.cc:82]

// case 2: Error message specified
PD_CHECK(a > b, "PD_CHECK returns ", false, ", expected a > b.")
// The key error message like:
// PD_CHECK returns false, expected a > b.
// [/User/custom_op/custom_relu_op.cc:82]
```

- PD_THROW: 用于直接抛出异常，支持变长参数输入

```
// case 1: No error message specified
PD_THROW()
// The key error message like:
// An error occurred.
// [/User/custom_op/custom_relu_op.cc:82]

// case 2: Error message specified
PD_THROW("PD_THROW returns ", false)
// The key error message like:
```

(下页继续)

(续上页)

```
// PD_THROW returns false
// [/User/custom_op/custom_relu_op.cc:82]
```

类 Python 的 C++ 运算 API

自 paddle 2.3 版本开始，我们提供定义与用法与相应 Python API 类似的 C++ API，其 API 命名、参数顺序及类型均和相应的 paddle Python API 对齐，可以通过查找相应 Python API 的官方文档了解其用法，并在自定义算子开发时使用。通过调用这些接口，可以省去封装基础运算的时间，从而提高开发效率。

在 2.3 版本支持的 C++ API 列表如下，可以通过 paddle::xxx 进行调用：

```
PADDLE_API Tensor abs(const Tensor& x);
PADDLE_API Tensor acos(const Tensor& x);
PADDLE_API Tensor acosh(const Tensor& x);
PADDLE_API Tensor add(const Tensor& x, const Tensor& y);
PADDLE_API Tensor allclose(const Tensor& x, const Tensor& y, const Scalar& rtol,
→const Scalar& atol, bool equal_nan);
PADDLE_API std::tuple<Tensor, Tensor> argsort(const Tensor& x, int axis, bool
→descending);
PADDLE_API Tensor asin(const Tensor& x);
PADDLE_API Tensor asinh(const Tensor& x);
PADDLE_API Tensor atan(const Tensor& x);
PADDLE_API Tensor atan2(const Tensor& x, const Tensor& y);
PADDLE_API Tensor atanh(const Tensor& x);
PADDLE_API Tensor bernoulli(const Tensor& x);
PADDLE_API Tensor ceil(const Tensor& x);
PADDLE_API Tensor cholesky(const Tensor& x, bool upper);
PADDLE_API Tensor cholesky_solve(const Tensor& x, const Tensor& y, bool upper);
PADDLE_API Tensor clip(const Tensor& x, const Scalar& min, const Scalar& max);
PADDLE_API Tensor concat(const std::vector<Tensor>& x, const Scalar& axis);
PADDLE_API Tensor conj(const Tensor& x);
PADDLE_API Tensor cos(const Tensor& x);
PADDLE_API Tensor cosh(const Tensor& x);
PADDLE_API Tensor cross(const Tensor& x, const Tensor& y, int axis=9);
PADDLE_API Tensor det(const Tensor& x);
PADDLE_API Tensor diag(const Tensor& x, int offset, float padding_value);
PADDLE_API Tensor diagonal(const Tensor& x, int offset, int axis1, int axis2);
PADDLE_API Tensor digamma(const Tensor& x);
PADDLE_API Tensor dist(const Tensor& x, const Tensor& y, float p);
PADDLE_API Tensor divide(const Tensor& x, const Tensor& y);
PADDLE_API Tensor dot(const Tensor& x, const Tensor& y);
PADDLE_API Tensor elu(const Tensor& x, float alpha);
PADDLE_API Tensor empty(const IntArray& shape, DataType dtype=DataType::FLOAT32,
→const Place& place=CPUPlace());
```

(下页继续)

(续上页)

```

PADDLE_API Tensor empty_like(const Tensor& x, DataType dtype=DataType::UNDEFINED,  

const Place& place={});  

PADDLE_API Tensor equal_all(const Tensor& x, const Tensor& y);  

PADDLE_API Tensor erf(const Tensor& x);  

PADDLE_API Tensor erfinv(const Tensor& x);  

PADDLE_API Tensor exp(const Tensor& x);  

PADDLE_API Tensor expand(const Tensor& x, const IntArray& shape);  

PADDLE_API Tensor expm1(const Tensor& x);  

PADDLE_API std::tuple<Tensor, Tensor> flatten(const Tensor& x, int start_axis, int  

stop_axis);  

PADDLE_API Tensor flip(const Tensor& x, const std::vector<int>& axis);  

PADDLE_API Tensor floor(const Tensor& x);  

PADDLE_API Tensor floor_divide(const Tensor& x, const Tensor& y);  

PADDLE_API Tensor full(const IntArray& shape, const Scalar& value, DataType  

dtype=DataType::FLOAT32, const Place& place=CPUPlace());  

PADDLE_API Tensor gather(const Tensor& x, const Tensor& index, const Scalar& axis=0);  

PADDLE_API Tensor gather_nd(const Tensor& x, const Tensor& index);  

PADDLE_API Tensor gelu(const Tensor& x, bool approximate);  

PADDLE_API Tensor gumbel_softmax(const Tensor& x, float temperature, bool hard, int  

axis);  

PADDLE_API Tensor imag(const Tensor& x);  

PADDLE_API Tensor increment(const Tensor& x, float value);  

PADDLE_API Tensor index_sample(const Tensor& x, const Tensor& index);  

PADDLE_API Tensor is_empty(const Tensor& x);  

PADDLE_API Tensor isclose(const Tensor& x, const Tensor& y, const Scalar& rtol, const  

Scalar& atol, bool equal_nan);  

PADDLE_API Tensor isfinite(const Tensor& x);  

PADDLE_API Tensor isinf(const Tensor& x);  

PADDLE_API Tensor isnan(const Tensor& x);  

PADDLE_API Tensor kron(const Tensor& x, const Tensor& y);  

PADDLE_API std::tuple<Tensor, Tensor> kthvalue(const Tensor& x, int k, int axis, bool  

keepdim);  

PADDLE_API Tensor label_smooth(const Tensor& label, paddle::optional<const Tensor&>  

prior_dist, float epsilon);  

PADDLE_API Tensor lerp(const Tensor& x, const Tensor& y, const Tensor& weight);  

PADDLE_API Tensor lgamma(const Tensor& x);  

PADDLE_API Tensor log(const Tensor& x);  

PADDLE_API Tensor log10(const Tensor& x);  

PADDLE_API Tensor log1p(const Tensor& x);  

PADDLE_API Tensor log2(const Tensor& x);  

PADDLE_API Tensor logit(const Tensor& x, float eps=1e-6f);  

PADDLE_API Tensor masked_select(const Tensor& x, const Tensor& mask);  

PADDLE_API Tensor matmul(const Tensor& x, const Tensor& y, bool transpose_x=false,  

bool transpose_y=false);

```

(下页继续)

(续上页)

```

PADDLE_API Tensor matrix_power(const Tensor& x, int n);
PADDLE_API Tensor maximum(const Tensor& x, const Tensor& y);
PADDLE_API Tensor maxout(const Tensor& x, int groups, int axis);
PADDLE_API Tensor minimum(const Tensor& x, const Tensor& y);
PADDLE_API std::tuple<Tensor, Tensor> mode(const Tensor& x, int axis, bool keepdim);
PADDLE_API Tensor multi_dot(const std::vector<Tensor>& x);
PADDLE_API Tensor multinomial(const Tensor& x, int num_samples, bool replacement);
PADDLE_API Tensor multiply(const Tensor& x, const Tensor& y);
PADDLE_API Tensor mv(const Tensor& x, const Tensor& vec);
PADDLE_API std::tuple<Tensor, Tensor> nll_loss(const Tensor& input, const Tensor&
→ label, paddle::optional<const Tensor> weight, int64_t ignore_index, const
→ std::string& reduction);
PADDLE_API Tensor one_hot(const Tensor& x, const Scalar& num_classes);
PADDLE_API Tensor pixel_shuffle(const Tensor& x, int upscale_factor, const
→ std::string& data_format);
PADDLE_API Tensor poisson(const Tensor& x);
PADDLE_API std::tuple<Tensor, Tensor> qr(const Tensor& x, const std::string& mode);
PADDLE_API Tensor real(const Tensor& x);
PADDLE_API Tensor reciprocal(const Tensor& x);
PADDLE_API Tensor relu(const Tensor& x);
PADDLE_API Tensor reshape(const Tensor& x, const IntArray& shape);
PADDLE_API Tensor roll(const Tensor& x, const IntArray& shifts, const std::vector
→ <int64_t>& axis);
PADDLE_API Tensor round(const Tensor& x);
PADDLE_API Tensor rsqrt(const Tensor& x);
PADDLE_API Tensor scatter(const Tensor& x, const Tensor& index, const Tensor& updates,
→ bool overwrite);
PADDLE_API Tensor scatter_nd_add(const Tensor& x, const Tensor& index, const Tensor&
→ updates);
PADDLE_API Tensor selu(const Tensor& x, float scale, float alpha);
PADDLE_API Tensor sign(const Tensor& x);
PADDLE_API Tensor silu(const Tensor& x);
PADDLE_API Tensor sin(const Tensor& x);
PADDLE_API Tensor sinh(const Tensor& x);
PADDLE_API std::vector<Tensor> split(const Tensor& x, const IntArray& num_or_sections,
→ const Scalar& axis);
PADDLE_API Tensor sqrt(const Tensor& x);
PADDLE_API Tensor square(const Tensor& x);
PADDLE_API Tensor stack(const std::vector<Tensor>& x, int axis);
PADDLE_API Tensor strided_slice(const Tensor& x, const std::vector<int>& axes, const
→ IntArray& starts, const IntArray& ends, const IntArray& strides);
PADDLE_API Tensor subtract(const Tensor& x, const Tensor& y);
PADDLE_API Tensor tanh(const Tensor& x);

```

(下页继续)

(续上页)

```
PADDLE_API Tensor thresholded_relu(const Tensor& x, float threshold);
PADDLE_API Tensor tile(const Tensor& x, const IntArray& repeat_times);
PADDLE_API Tensor trace(const Tensor& x, int offset, int axis1, int axis2);
PADDLE_API Tensor triangular_solve(const Tensor& x, const Tensor& y, bool upper, bool
→ transpose, bool unitriangular);
PADDLE_API std::vector<Tensor> unbind(const Tensor& input, int axis);
PADDLE_API std::tuple<Tensor, Tensor, Tensor> unique(const Tensor& x, bool
→ return_index, bool return_inverse, bool return_counts, const std::vector<int>& axis,
→ DataType dtype=DataType::INT64);
PADDLE_API std::tuple<Tensor, Tensor> unsqueeze(const Tensor& x, const IntArray& axis);
PADDLE_API Tensor where(const Tensor& condition, const Tensor& x, const Tensor& y);
```

注：后续我们会提供更方便的查阅 C++ API 文档的入口。

在 2.3 版本，我们共支持了大约 250 个类似的 C++ API，能够覆盖大部分的基础运算，但是除前述的 109 个 C++ API 之外，剩余的 C++ API 由于一些历史原因，其参数列表尚未和相应的 Python API 对齐，因此目前剩余这些 API 只能作为 experimental 的 API 使用，需要通过 paddle::experimental::xxx 进行调用，且这些 experimental API 在下个版本可能会有不兼容的升级，如果不介意随下一版本升级的话，可以使用，追求稳定的话则不建议使用。

如有需要，目前支持的全量 API 列表（包含 experimental API）请参考 paddle 安装路径下的 api.h 头文件，以 Python3.7 为例，其路径是 python3.7/site-packages/paddle/include/paddle/phi/api/include/api.h。

运算函数实现

对函数写法以及基础 API 的定义有了初步认识后，下面结合具体的示例进行介绍。

CPU 实现

以 relu 算子为例，一个支持 float32 类型的 CPU relu 算子运算函数可以实现如下：

- relu_cpu_fp32.cc

```
#include "paddle/extension.h"

#include <vector>

#define CHECK_INPUT(x) PD_CHECK(x.is_cpu(), #x " must be a CPU Tensor.")

std::vector<paddle::Tensor> ReluCPUForward(const paddle::Tensor& x) {
    CHECK_INPUT(x);
```

(下页继续)

(续上页)

```

auto out = paddle::empty_like(x);

auto x_numel = x.numel();
auto* x_data = x.data<float>();
auto* out_data = out.data<float>();

for (int64_t i = 0; i < x_numel; ++i) {
    out_data[i] = std::max(static_cast<float>(0.), x_data[i]);
}

return {out};
}

std::vector ReluCPUBackward(const paddle::Tensor& x,
                                            const paddle::Tensor& out,
                                            const paddle::Tensor& grad_out) {
    CHECK_INPUT(x);
    CHECK_INPUT(out);
    CHECK_INPUT(grad_out);

    auto grad_x = paddle::empty_like(x);

    auto out_numel = out.numel();
    auto* out_data = out.data<float>();
    auto* grad_out_data = grad_out.data<float>();
    auto* grad_x_data = grad_x.data<float>();

    for (int64_t i = 0; i < out_numel; ++i) {
        grad_x_data[i] =
            grad_out_data[i] * (out_data[i] > static_cast<float>(0) ? 1. : 0.);
    }

    return {grad_x};
}

```

主要逻辑包括：

1. 创建输出 Tensor
2. 获取输入和输出 Tensor 的数据区起始地址
3. 计算得到输出 Tensor 的数值，返回结果

前述 relu 示例实现仅支持 float32 类型的计算，如果仅有一种数据类型的支持需求，用以上写法即可。

如果需要同时支持多种数据类型，例如同时支持 float32 与 float64 的计算，可以使用相应的 DIAPATCH

宏进行声明，示例如下：

- relu_cpu.cc

```
#include "paddle/extension.h"

#include <vector>

#define CHECK_INPUT(x) PD_CHECK(x.is_cpu(), "#x must be a CPU Tensor.")

template <typename data_t>
void relu_cpu_forward_kernel(const data_t* x_data,
                             data_t* out_data,
                             int64_t x_numel) {
    for (int64_t i = 0; i < x_numel; ++i) {
        out_data[i] = std::max(static_cast<data_t>(0.), x_data[i]);
    }
}

template <typename data_t>
void relu_cpu_backward_kernel(const data_t* grad_out_data,
                             const data_t* out_data,
                             data_t* grad_x_data,
                             int64_t out_numel) {
    for (int64_t i = 0; i < out_numel; ++i) {
        grad_x_data[i] =
            grad_out_data[i] * (out_data[i] > static_cast<data_t>(0) ? 1. : 0.);
    }
}

std::vector<paddle::Tensor> ReluCPUForward(const paddle::Tensor& x) {
    CHECK_INPUT(x);

    auto out = paddle::empty_like(x);

    PD_DISPATCH_FLOATING_TYPES(
        x.type(), "relu_cpu_forward_kernel", ([&] {
            relu_cpu_forward_kernel<data_t>(
                x.data<data_t>(), out.data<data_t>(), x.numel());
        }));
}

return {out};
}

std::vector<paddle::Tensor> ReluCPUBackward(const paddle::Tensor& x,
```

(下页继续)

(续上页)

```

const paddle::Tensor& out,
const paddle::Tensor& grad_out) {

CHECK_INPUT(x);
CHECK_INPUT(out);
CHECK_INPUT(grad_out);

auto grad_x = paddle::empty_like(x);

PD_DISPATCH_FLOATING_TYPES(out.type(), "relu_cpu_backward_kernel", ([&] {
    relu_cpu_backward_kernel<data_t>(
        grad_out.data<data_t>(),
        out.data<data_t>(),
        grad_x.data<data_t>(),
        out.numel());
}));

return {grad_x};
}

```

注：编写模板计算函数时，模板参数名 `data_t` 用于适配不同的数据类型，不可更改为其他命名，否则会编译失败

示例中的 `PD_DISPATCH_FLOATING_TYPES` 会展开得到 `float32` 与 `float64` 的 `switch-case` 实现，从而在运行时根据输入的数据类型，选择实际需要执行的分支。

例如，`ReluCPUForward` 中的 `PD_DISPATCH_FLOATING_TYPES` 实际代码展开如下：

```

switch(x.type()) {
    case paddle::DataType::FLOAT32:
        relu_cpu_forward_kernel<float>(
            x.data<float>(), out.data<float>(), x.numel());
        break;
    case paddle::DataType::FLOAT64:
        relu_cpu_forward_kernel<double>(
            x.data<double>(), out.data<float>(), x.numel());
        break;
    default:
        PD_THROW(
            "function relu_cpu_forward_kernel is not implemented for data type `",
            paddle::ToString(x.type()), "`");
}

```

目前定义的 `dispatch` 宏包括：

- `PD_DISPATCH_FLOATING_TYPES`：`dispatch` 生成 `float` 和 `double` 对应的实现

- PD_DISPATCH_FLOATING_AND_HALF_TYPES : dispatch 生成 float , double 和 paddle::float16 对应的实现
- PD_DISPATCH_INTEGRAL_TYPES : dispatch 生成 int8_t, uint8_t, int16_t, int 的 int64_t 对应的实现
- PD_DISPATCH_COMPLEX_TYPES: dispatch 生成 paddle::complex64 和 paddle::complex128 对应的实现
- PD_DISPATCH_FLOATING_AND_INTEGRAL_TYPES : dispatch 生成 前述 PD_DISPATCH_FLOATING_TYPES 和 PD_DISPATCH_INTEGRAL_TYPES 两个宏全部数据类型对应的实现
- PD_DISPATCH_FLOATING_AND_COMPLEX_TYPES: dispatch 生成 前述 PD_DISPATCH_FLOATING_TYPES 和 PD_DISPATCH_COMPLEX_TYPES 两个宏全部数据类型对应的实现
- PD_DISPATCH_FLOATING_AND_INTEGRAL_AND_COMPLEX_TYPES: dispatch 生成 前述 PD_DISPATCH_FLOATING_TYPES , PD_DISPATCH_INTEGRAL_TYPES 和 PD_DISPATCH_COMPLEX_TYPES 三个宏全部数据类型对应的实现

当然, 如果这几个宏无法满足您实际使用的需求, 您可以直接通过 switch-case 语句实现, 将来视需求我们也会添加更多的宏。

CPU&CUDA 混合实现

通常只有 CPU 的算子实现是不够的, 实际生产环境中一般需要使用 GPU 算子。此处将前述 relu_cpu.cc 中算子的 CPU 实现改为 GPU 示例如下:

- relu_cuda.cu

```
#include "paddle/extension.h"

template <typename data_t>
__global__ void relu_cuda_forward_kernel(const data_t* x,
                                         data_t* y,
                                         int64_t num) {
    int64_t gid = blockIdx.x * blockDim.x + threadIdx.x;
    for (int64_t i = gid; i < num; i += blockDim.x * gridDim.x) {
        y[i] = max(x[i], static_cast<data_t>(0.));
    }
}

template <typename data_t>
__global__ void relu_cuda_backward_kernel(const data_t* dy,
                                         const data_t* y,
```

(下页继续)

(续上页)

```

        data_t* dx,
        int64_t num) {
int64_t gid = blockIdx.x * blockDim.x + threadIdx.x;
for (int64_t i = gid; i < num; i += blockDim.x * gridDim.x) {
    dx[i] = dy[i] * (y[i] > 0 ? 1. : 0.);
}
}

std::vector relu_cuda_forward(const paddle::Tensor& x) {
    auto out = paddle::empty_like(x);

    int64_t numel = x.numel();
    int64_t block = 512;
    int64_t grid = (numel + block - 1) / block;
    PD_DISPATCH_FLOATING_TYPES(
        x.type(), "relu_cuda_forward_kernel", ([&] {
            relu_cuda_forward_kernel<data_t><<<grid, block, 0, x.stream()>>>(
                x.data<data_t>(),
                out.data<data_t>(),
                numel);
        }));
    }

    return {out};
}

std::vector relu_cuda_backward(const paddle::Tensor& x,
                                              const paddle::Tensor& out,
                                              const paddle::Tensor& grad_out) {
    auto grad_x = paddle::empty_like(x);

    int64_t numel = out.numel();
    int64_t block = 512;
    int64_t grid = (numel + block - 1) / block;
    PD_DISPATCH_FLOATING_TYPES(
        out.type(), "relu_cuda_backward_kernel", ([&] {
            relu_cuda_backward_kernel<data_t><<<grid, block, 0, x.stream()>>>(
                grad_out.data<data_t>(),
                out.data<data_t>(),
                grad_x.data<data_t>(),
                numel);
        }));
    }

    return {grad_x};
}

```

- relu_cuda.cc

```
#include "paddle/extension.h"

#include <vector>

#define CHECK_INPUT(x) PD_CHECK(x.is_gpu(), #x " must be a GPU Tensor.")

std::vector<paddle::Tensor> relu_cuda_forward(const paddle::Tensor& x);
std::vector<paddle::Tensor> relu_cuda_backward(const paddle::Tensor& x,
                                                const paddle::Tensor& out,
                                                const paddle::Tensor& grad_out);

std::vector<paddle::Tensor> ReluCUDAForward(const paddle::Tensor& x) {
    CHECK_INPUT(x);

    return relu_cuda_forward(x);
}

std::vector<paddle::Tensor> ReluCUDABackward(const paddle::Tensor& x,
                                              const paddle::Tensor& out,
                                              const paddle::Tensor& grad_out) {
    CHECK_INPUT(x);
    CHECK_INPUT(out);
    CHECK_INPUT(grad_out);

    return relu_cuda_backward(x, out, grad_out);
}
```

在 .cu 文件中实现对应的 CUDA kernel 和计算函数，在 .cc 文件中声明调用即可。

注意这里的 CHECK_INPUT 也改为检查输入 Tensor 是否在 GPU 上，如果后续仍然在 CPU 上执行，将会报错如下，可以看到报错提示与 CHECK_INPUT 缩写提示一致。至于错误类型，PaddlePaddle 将外部扩展自定义算子视为第三方模块，错误类型统一为 OSSError: (External)，与其他第三方库报错类型一致。报错示例如下：

```
Traceback (most recent call last):
  File "relu_test_jit_dy.py", line 70, in <module>
    out = net(image)
  File "/usr/local/lib/python3.7/site-packages/paddle/fluid/dygraph/layers.py", line 902, in __call__
    outputs = self.forward(*inputs, **kwargs)
  File "relu_test_jit_dy.py", line 45, in forward
    tmp_out = custom_ops.custom_relu(tmp1)
  File "/root/.cache/paddle_extensions/custom_jit_ops/custom_jit_ops.py", line 16, in custom_relu
```

(下页继续)

(续上页)

```

    helper.append_op(type="custom_relu", inputs=ins, outputs=outs, attrs=attrs)
File "/usr/local/lib/python3.7/site-packages/paddle/fluid/layer_helper.py", line 43,
→ in append_op
    return self.main_program.current_block().append_op(*args, **kwargs)
File "/usr/local/lib/python3.7/site-packages/paddle/fluid/framework.py", line 3079,_
→ in append_op
    kwargs.get("stop_gradient", False))
File "/usr/local/lib/python3.7/site-packages/paddle/fluid/dygraph/tracer.py", line_
→ 45, in trace_op
    not stop_gradient)
OSError: (External) x must be a GPU Tensor.
[/work/scripts/custom_op/guide/relu_cuda.cc:13] (at /work/paddle/paddle/fluid/
→ framework/custom_operator.cc:168)
[operator < custom_relu > error]

```

实际使用时，一般您只需要根据您实际使用的设备，编写对应设备的算子实现即可，例如您使用 GPU 训练，仅需要实现算子的 CUDA 版本即可使用，如果您需要您的自定义算子同时支持多种设备，例如同时支持 CPU 与 GPU，只需要将 CPU 和 GPU 的实现整合到一起，并在前反向函数中实现对应的分支即可，示例如下：

- relu.cc

```

#include "paddle/extension.h"

#include <vector>

#define CHECK_CPU_INPUT(x) PD_CHECK(x.is_cpu(), "#x " must be a CPU Tensor.")

template <typename data_t>
void relu_cpu_forward_kernel(const data_t* x_data,
                            data_t* out_data,
                            int64_t x_numel) {
    for (int64_t i = 0; i < x_numel; ++i) {
        out_data[i] = std::max(static_cast<data_t>(0.), x_data[i]);
    }
}

template <typename data_t>
void relu_cpu_backward_kernel(const data_t* grad_out_data,
                            const data_t* out_data,
                            data_t* grad_x_data,
                            int64_t out_numel) {
    for (int64_t i = 0; i < out_numel; ++i) {
        grad_x_data[i] =
            grad_out_data[i] * (out_data[i] > static_cast<data_t>(0) ? 1. : 0.);
    }
}

```

(下页继续)

(续上页)

```

}

std::vector relu_cpu_forward(const paddle::Tensor& x) {
    CHECK_CPU_INPUT(x);

    auto out = paddle::empty_like(x);

    PD_DISPATCH_FLOATING_TYPES(
        x.type(), "relu_cpu_forward_kernel", ([&] {
            relu_cpu_forward_kernel<data_t>(
                x.data<data_t>(), out.data<data_t>(), x.numel());
        }));
}

return {out};
}

std::vector relu_cpu_backward(const paddle::Tensor& x,
                                             const paddle::Tensor& out,
                                             const paddle::Tensor& grad_out) {
    CHECK_CPU_INPUT(x);
    CHECK_CPU_INPUT(out);
    CHECK_CPU_INPUT(grad_out);

    auto grad_x = paddle::empty_like(x);

    PD_DISPATCH_FLOATING_TYPES(out.type(), "relu_cpu_backward_kernel", ([&] {
        relu_cpu_backward_kernel<data_t>(
            grad_out.data<data_t>(),
            out.data<data_t>(),
            grad_x.data<data_t>(),
            out.numel());
    }));
}

return {grad_x};
}

// NOTE: If your custom operator may be compiled in an environment with CUDA,
// or it may be compiled in an environment without CUDA, in order to adapt the
// compilation environment, you can use the PADDLE_WITH_CUDA macro control
// the CUDA related code.
#ifdef PADDLE_WITH_CUDA
std::vector relu_cuda_forward(const paddle::Tensor& x);

```

(下页继续)

(续上页)

```

std::vector relu_cuda_backward(const paddle::Tensor& x,
                                             const paddle::Tensor& out,
                                             const paddle::Tensor& grad_out);
#endif

std::vector ReluForward(const paddle::Tensor& x) {
    if (x.is_cpu()) {
        return relu_cpu_forward(x);
    #ifdef PADDLE_WITH_CUDA
    } else if (x.is_gpu()) {
        return relu_cuda_forward(x);
    #endif
    } else {
        PD_THROW("Unsupported device type for forward function of custom relu operator.");
    }
}

std::vector ReluBackward(const paddle::Tensor& x,
                                       const paddle::Tensor& out,
                                       const paddle::Tensor& grad_out) {
    if (x.is_cpu()) {
        return relu_cpu_backward(x, out, grad_out);
    #ifdef PADDLE_WITH_CUDA
    } else if (x.is_gpu()) {
        return relu_cuda_backward(x, out, grad_out);
    #endif
    } else {
        PD_THROW("Unsupported device type for backward function of custom relu operator.
        ");
    }
}

```

- relu.cu

```

#include "paddle/extension.h"

#define CHECK_CUDA_INPUT(x) PD_CHECK(x.is_gpu(), "#x " must be a GPU Tensor.")

template <typename data_t>
__global__ void relu_cuda_forward_kernel(const data_t* x,
                                         data_t* y,
                                         int64_t num) {
    int64_t gid = blockIdx.x * blockDim.x + threadIdx.x;
    for (int64_t i = gid; i < num; i += blockDim.x * gridDim.x) {

```

(下页继续)

(续上页)

```

y[i] = max(x[i], static_cast<data_t>(0.));
}

}

template <typename data_t>
__global__ void relu_cuda_backward_kernel(const data_t* dy,
                                         const data_t* y,
                                         data_t* dx,
                                         int64_t num) {
    int64_t gid = blockIdx.x * blockDim.x + threadIdx.x;
    for (int64_t i = gid; i < num; i += blockDim.x * gridDim.x) {
        dx[i] = dy[i] * (y[i] > 0 ? 1. : 0.);
    }
}

std::vector relu_cuda_forward(const paddle::Tensor& x) {
    CHECK_CUDA_INPUT(x);

    auto out = paddle::empty_like(x);

    int64_t numel = x.numel();
    int64_t block = 512;
    int64_t grid = (numel + block - 1) / block;
    PD_DISPATCH_FLOATING_TYPES(
        x.type(), "relu_cuda_forward_kernel", ([&] {
            relu_cuda_forward_kernel<data_t><<<grid, block, 0, x.stream()>>>(
                x.data<data_t>(), out.data<data_t>(), numel);
        }));
}

return {out};
}

std::vector relu_cuda_backward(const paddle::Tensor& x,
                                              const paddle::Tensor& out,
                                              const paddle::Tensor& grad_out) {
    CHECK_CUDA_INPUT(x);
    CHECK_CUDA_INPUT(out);
    CHECK_CUDA_INPUT(grad_out);

    auto grad_x = paddle::empty_like(x);

    int64_t numel = out.numel();
    int64_t block = 512;
}

```

(下页继续)

(续上页)

```

int64_t grid = (numel + block - 1) / block;
PD_DISPATCH_FLOATING_TYPES(
    out.type(), "relu_cuda_backward_kernel", ([&] {
        relu_cuda_backward_kernel<data_t><<<grid, block, 0, x.stream()>>>(
            grad_out.data<data_t>(),
            out.data<data_t>(),
            grad_x.data<data_t>(),
            numel);
    }));
}

return {grad_x};
}

```

维度与类型推导函数实现

PaddlePaddle 框架同时支持动态图与静态图的执行模式，在静态图模式下，组网阶段需要完成 Tensor shape 和 dtype 的推导，从而生成正确的模型描述，用于后续 Graph 优化与执行。因此，除了算子的运算函数之外，还需要实现前向运算的维度和类型的推导函数。

维度推导（InferShape）和类型推导（InferDtype）的函数写法也是有要求的，形式如下：

```

std::vector<std::vector<int64_t>> OpInferShape(std::vector<int64_t> x_shape, ...) {
    return {x_shape, ...};
}

std::vector<paddle::DataType> OpInferDtype(paddle::DataType x_dtype, ...) {
    return {x_dtype, ...};
}

```

函数的输入参数与返回值类型固定，具体类型如上述代码片段所示，其他要求如下：

- 函数输入参数与前述运算函数的输入 Tensor 按顺序一一对应，依次为输入参数的 shape 和 dtype，这里的对应规则为：
 - paddle::Tensor -> std::vector<int64_t>
 - std::vector<paddle::Tensor> -> std::vector<std::vector<int64_t>>
- 函数返回值 vector 中的 shape 或 dtype 信息也需要与返回 Tensor 按顺序一一对应
- 维度推导函数支持 Attribute 的输入，在实现维度推导函数时，可以不使用 Attribute 的输入参数，也可以使用，但如果要使用的话，需要和 Forward 函数的 Attribute 参数保持一致
- 类型推导函数不支持 Attribute 的输入

以 relu 为例，其维度与类型推导函数如下：

- relu_cpu_fp32.cc / relu_cpu.cc / relu_cuda.cc / relu.cc (需将以下代码追加到前述文件中)

```
// 维度推导
std::vector<std::vector<int64_t>> ReluInferShape(std::vector<int64_t> x_shape) {
    return {x_shape};
}

// 类型推导
std::vector<paddle::DataType> ReluInferDtype(paddle::DataType x_dtype) {
    return {x_dtype};
}
```

注：如果是 CUDA 算子，ReluInferShape 和 ReluInferDtype 仅需要在.cc 文件中实现，不需要在.cu 中重复实现

对于仅有一个输入 Tensor 和一个输出 Tensor 的自定义算子，如果输出 Tensor 和输入 Tensor 的 shape 和 dtype 一致，可以省略 InferShape 和 InferDtype 函数的实现，其他场景下均需要实现这两个函数。因此，对于这里的 relu 算子来说，这两个函数可以不写。

此外，以 concat 为例，如果其将 axis 参数作为前向函数的 Attribute 输入，其维度与类型推导函数如下：

```
// 前向函数
std::vector<paddle::Tensor> ConcatForwardStaticAxis(
    const std::vector<paddle::Tensor>& inputs, int64_t axis) { ... }

// 维度推导
std::vector<std::vector<int64_t>> ConcatInferShapeStaticAxis(
    const std::vector<std::vector<int64_t>>& input_shapes,
    int64_t axis) { ... }

// 类型推导
std::vector<paddle::DataType> ConcatInferDtypeStaticAxis(
    const std::vector<paddle::DataType>& input_dtotypes) { ... }
```

构建算子

最后，需要调用 PD_BUILD_OP 系列宏，构建算子的描述信息，并关联前述算子运算函数和维度、类型推导函数。

我们提供了 3 个构建算子的宏：

- PD_BUILD_OP：用于构建前向算子
- PD_BUILD_GRAD_OP：用于构建前向算子对应的反向算子
- PD_BUILD_DOUBLE_GRAD_OP：用于构建前反向算子对应的二阶反向算子

注：二阶以上的反向算子构建暂不支持。

对于 relu CPU 示例来说，构建算子描述如下：

- `relu_cpu_fp32.cc / relu_cpu.cc`（需将以下代码追加到前述文件中）

```
PD_BUILD_OP(custom_relu)
    .Inputs({ "X" })
    .Outputs({ "Out" })
    .SetKernelFn(PD_KERNEL(ReluCPUForward))
    .SetInferShapeFn(PD_INFER_SHAPE(ReluInferShape))
    .SetInferDtypeFn(PD_INFER_DTYPE(ReluInferDtype));

PD_BUILD_GRAD_OP(custom_relu)
    .Inputs({ "X", "Out", paddle::Grad("Out") })
    .Outputs({paddle::Grad("X")})
    .SetKernelFn(PD_KERNEL(ReluCPUBackward));
```

这里写法上需要注意以下几点：

- `PD_BUILD_OP` 系列宏后面的括号内为算子名，也是后面在 `python` 端使用的接口名，注意前后不需要引号，注意该算子名不能与 PaddlePaddle 内已有算子名重名，比如 `relu` 为 PaddlePaddle 内已有算子，如果直接使用 `relu` 作为算子名将无法注册成功，所以此处增加了前缀 `custom_`
- `PD_BUILD_OP`、`PD_BUILD_GRAD_OP` 和 `PD_BUILD_DOUBLE_GRAD_OP` 构建同一个算子的前向、反向、二阶反向实现，宏后面使用的算子名需要保持一致，比如该示例中均使用 `custom_relu`
- `PD_BUILD_OP`、`PD_BUILD_GRAD_OP` 和 `PD_BUILD_DOUBLE_GRAD_OP` 必须顺次调用，不允许在未调用 `PD_BUILD_OP` 构建前向算子的情况下，直接调用 `PD_BUILD_GRAD_OP` 构建反向算子
- `Inputs` 与 `Outputs` 的输入参数为 `std::vector<std::string>`，依次是前面算子运算函数的输入输出 `Tensor` 的 `name`，需要按顺序一一对应，此处的 `name` 与函数输入参数的变量名没有强关联，比如函数输入参数是 `const paddle::Tensor& x`，`Inputs` 中的 `name` 可以是 `Input`，`x`，`X`，`In` 等等
- `PD_BUILD_OP` 与 `PD_BUILD_GRAD_OP` 中的 `Inputs` 与 `Outputs` 的 `name` 有强关联，对于前向算子的某个输入，如果反向算子仍然要复用，那么其 `name` 一定要保持一致，因为内部执行时，会以 `name` 作为 `key` 去查找对应的变量，比如这里前向算子的 `X`，`Out` 与反向算子的 `X`，`Out` 指代同一个 `Tensor`
- 在声明反向算子的 `Inputs` 与 `Outputs` 时，前向 `Tensor` 对应的梯度 `Tensor` 名需要由 `paddle::Grad` 处理前向 `Tensor` 名得到，不能够随意声明，例如这里 `"X"` 对应的梯度 `Tensor` 名为 `paddle::Grad("X")`
- 如果算子的 `Inputs` 与 `Outputs` 中包含变长的 `Tensor` 输入和输出，其 `Tensor` 名需要由 `paddle::Vec` 方法处理得到，例如对于前述 `concat` 算子的前向输入 `const std::vector<paddle::Tensor>& inputs`，其 `Tensor` 名可以为 `paddle::Vec("X")`，对应的梯度 `Tensor` 名为 `paddle::Grad(paddle::Vec("X"))`，此处 `paddle::Grad` 需要在 `paddle::Vec` 的外面

- 此处 SetKernelFn 、 SetInferShapeFn 与 SetInferDtypeFn 中的 PD_KERNEL 、 PD_INFER_SHAPE 、 PD_INFER_DTYPE 宏用于自动转换并统一函数的签名，不可以省略
- 反向算子构建暂时不支持调用 SetInferShapeFn 和 SetInferDtypeFn 自定义维度与类型推导函数，框架会根据前向 Tensor 的 shape 和 dtype ，设定其对应梯度 Tensor 的 shape 和 dtype

如前述介绍，此处 relu 也可以省略 InferShape 和 InferDtype 函数的实现，因此也可以写为：

```
PD_BUILD_OP(custom_relu)
    .Inputs({ "X" })
    .Outputs({ "Out" })
    .SetKernelFn(PD_KERNEL(ReluCPUForward));

PD_BUILD_GRAD_OP(custom_relu)
    .Inputs({ "X", "Out", paddle::Grad("Out") })
    .Outputs({paddle::Grad("X") })
    .SetKernelFn(PD_KERNEL(ReluCPUBackward));
```

类似地， GPU 示例构建算子描述如下，替换 KernelFn 即可：

- relu_cuda.cc （需将以下代码追加到前述文件中）

```
PD_BUILD_OP(custom_relu)
    .Inputs({ "X" })
    .Outputs({ "Out" })
    .SetKernelFn(PD_KERNEL(ReluCUDAForward));

PD_BUILD_GRAD_OP(custom_relu)
    .Inputs({ "X", "Out", paddle::Grad("Out") })
    .Outputs({paddle::Grad("X") })
    .SetKernelFn(PD_KERNEL(ReluCUDABackward));
```

对于 concat 算子，其包含变长的输入输出，因此 PD_BUILD_OP 声明时需要用到 paddle::Vec 方法，示例如下：

```
PD_BUILD_OP(custom_concat_with_attr)
    .Inputs({paddle::Vec("X") })
    .Outputs({ "Out" })
    .Attrs({ "axis: int64_t" })
    .SetKernelFn(PD_KERNEL(ConcatForwardStaticAxis))
    .SetInferShapeFn(PD_INFER_SHAPE(ConcatInferShapeStaticAxis))
    .SetInferDtypeFn(PD_INFER_DTYPE(ConcatInferDtypeStaticAxis));

PD_BUILD_GRAD_OP(custom_concat_with_attr)
    .Inputs({paddle::Vec("X"), paddle::Grad("Out") })
    .Outputs({paddle::Grad(paddle::Vec("X")) })
```

(下页继续)

(续上页)

```
.Attrs({ "axis: int64_t" })
.SetKernelFn(PD_KERNEL(ConcatBackwardStaticAxis));
```

Attribute 声明

对于 Attribute 的声明，和 Inputs、Outputs 的声明有所不同，需要按照如下格式声明字符串：

```
<name>: <attr-type-expr>
```

其中，name 为 Attribute 变量的 name，<attr-type-expr> 为 Attribute 变量的类型，类型字符串需要与 C++ 类型严格一致。通过如下示例说明：

假如有前向运算函数形式如下：

```
std::vector<paddle::Tensor> AttrTestForward(
    const paddle::Tensor& x,
    bool bool_attr,
    int int_attr,
    float float_attr,
    int64_t int64_attr,
    const std::string& str_attr,
    const std::vector<int>& int_vec_attr,
    const std::vector<float>& float_vec_attr,
    const std::vector<int64_t>& int64_vec_attr,
    const std::vector<std::string>& str_vec_attr) {...}
```

对应的 BUILD_OP 写法为：

```
PD_BUILD_OP(attr_test)
    .Inputs({"X"})
    .Outputs({"Out"})
    .Attrs("bool_attr: bool",
           "int_attr: int",
           "float_attr: float",
           "int64_attr: int64_t",
           "str_attr: std::string",
           "int_vec_attr: std::vector<int>",
           "float_vec_attr: std::vector<float>",
           "int64_vec_attr: std::vector<int64_t>",
           "str_vec_attr: std::vector<std::string>")
    .SetKernelFn(PD_KERNEL(AttrTestForward));
```

如果该算子需要反向实现，反向算子的 Attribute 输入参数需要是前向算子 Attribute 输入参数的子集，不能新增前向算子没有的 Attribute，示例如下：

```
std::vector<paddle::Tensor> AttrTestBackward(
    const paddle::Tensor& grad_out,
    int int_attr,
    const std::vector<float>& float_vec_attr,
    const std::vector<std::string>& str_vec_attr) { ... }

PD_BUILD_GRAD_OP(attr_test)
    .Inputs({paddle::Grad("Out")})
    .Outputs({paddle::Grad("X")})
    .Attrs({"int_attr: int",
            "float_vec_attr: std::vector<float>",
            "str_vec_attr: std::vector<std::string>"})
    .SetKernelFn(PD_KERNEL(AttrTestBackward));
```

这里的 `int_attr`、`float_vec_attr`、`str_vec_attr` 均是前向算子声明中出现过的参数，这里仅限定 `Attrs` 方法中字符串的命名，函数的输入参数命名没有限制，只需要确保数据类型一致即可，例如这里 `AttrTestBackward` 也可以改为如下写法：

```
std::vector<paddle::Tensor> AttrTestBackward(
    const paddle::Tensor& grad_out,
    int a,
    const std::vector<float>& b,
    const std::vector<std::string>& c) { ... }
```

自定义算子编译与使用

本机制提供了两种编译自定义算子的方式，分别为 使用 `setuptools` 编译与 即时编译，下面依次通过示例介绍。

注：在进行编译之前，需要根据实际需求，将前述 运算函数实现，维度与类型推导函数实现，构建算子三节中的代码示例组合到一起，具体地，需要将 维度与类型推导函数实现，构建算子两节中的代码片段追加到 运算函数实现 小节中对应的 *.cc 文件中

使用 `setuptools` 编译

该方式是对 python 内建库中的 `setuptools.setup` 接口的进一步封装，能够自动地生成 Python API 并以 Module 的形式安装到 site-packages 目录。编译完成后，支持通过 `import` 语句导入使用。

您需要编写 `setup.py` 文件，配置自定义算子的编译规则。

例如，前述 `relu` 示例的 `setup` 文件可以实现如下：

- `setup_cpu.py` (for `relu_cpu.cc`)

```
from paddle.utils.cpp_extension import CppExtension, setup

setup(
    name='custom_setup_ops',
    ext_modules=CppExtension(
        sources=['relu_cpu.cc']
    )
)
```

- setup_cuda.py (for relu_cuda.cc & relu_cuda.cu)

```
from paddle.utils.cpp_extension import CUDAExtension, setup

setup(
    name='custom_setup_ops',
    ext_modules=CUDAExtension(
        sources=['relu_cuda.cc', 'relu_cuda.cu']
    )
)
```

其中 `paddle.utils.cpp_extension.setup` 能够自动搜索和检查本地的 `cc(Linux)`、`cl.exe(Windows)` 和 `nvcc` 编译命令和版本环境，根据用户指定的 Extension 类型，完成 CPU 或 GPU 设备的算子编译安装。

执行 `python setup_cpu.py install` 或者 `python setup_cuda.py install` 即可一键完成自定义算子的编译和安装。

以 `python setup_cuda.py install` 为例，执行日志如下：

```
running install
running bdist_egg
running egg_info
writing custom_setup_ops.egg-info/PKG-INFO
writing dependency_links to custom_setup_ops.egg-info/dependency_links.txt
writing top-level names to custom_setup_ops.egg-info/top_level.txt
reading manifest file 'custom_setup_ops.egg-info/SOURCES.txt'
writing manifest file 'custom_setup_ops.egg-info/SOURCES.txt'
installing library code to build/custom_setup_ops/bdist.linux-x86_64/egg
running install_lib
running build_ext
/usr/local/lib/python3.7/site-packages/paddle/fluid/layers/utils.py:77:_
  ↪DeprecationWarning: Using or importing the ABCs from 'collections' instead of from
  ↪'collections.abc' is deprecated, and in 3.8 it will stop working
    return (isinstance(seq, collections.Sequence) and
Compiling user custom op, it will cost a few seconds.....
```

(下页继续)

(续上页)

```

creating build/custom_setup_ops/bdist.linux-x86_64/egg
copying build/custom_setup_ops/lib.linux-x86_64-3.7/version.txt -> build/custom_setup_
↪ops/bdist.linux-x86_64/egg
copying build/custom_setup_ops/lib.linux-x86_64-3.7/relu_cpu.o -> build/custom_setup_
↪ops/bdist.linux-x86_64/egg
copying build/custom_setup_ops/lib.linux-x86_64-3.7/relu_cuda.o -> build/custom_setup_
↪ops/bdist.linux-x86_64/egg
copying build/custom_setup_ops/lib.linux-x86_64-3.7/relu_cuda.cu.o -> build/custom_
↪setup_ops/bdist.linux-x86_64/egg
copying build/custom_setup_ops/lib.linux-x86_64-3.7/custom_setup_ops.so -> build/
↪custom_setup_ops/bdist.linux-x86_64/egg
creating stub loader for custom_setup_ops.so
byte-compiling build/custom_setup_ops/bdist.linux-x86_64/egg/custom_setup_ops.py to_
↪custom_setup_ops.cpython-37.pyc
creating build/custom_setup_ops/bdist.linux-x86_64/egg/EGG-INFO
copying custom_setup_ops.egg-info/PKG-INFO -> build/custom_setup_ops/bdist.linux-x86_
↪_64/egg/EGG-INFO
copying custom_setup_ops.egg-info/SOURCES.txt -> build/custom_setup_ops/bdist.linux-
↪x86_64/egg/EGG-INFO
copying custom_setup_ops.egg-info/dependency_links.txt -> build/custom_setup_ops/
↪bdist.linux-x86_64/egg/EGG-INFO
copying custom_setup_ops.egg-info/not-zip-safe -> build/custom_setup_ops/bdist.linux-
↪x86_64/egg/EGG-INFO
copying custom_setup_ops.egg-info/top_level.txt -> build/custom_setup_ops/bdist.linux-
↪x86_64/egg/EGG-INFO
writing build/custom_setup_ops/bdist.linux-x86_64/egg/EGG-INFO/native_libs.txt
creating 'dist/custom_setup_ops-0.0.0-py3.7-linux-x86_64.egg' and adding 'build/
↪custom_setup_ops/bdist.linux-x86_64/egg' to it
removing 'build/custom_setup_ops/bdist.linux-x86_64/egg' (and everything under it)
Processing custom_setup_ops-0.0.0-py3.7-linux-x86_64.egg
creating /usr/local/lib/python3.7/site-packages/custom_setup_ops-0.0.0-py3.7-linux-
↪x86_64.egg
Extracting custom_setup_ops-0.0.0-py3.7-linux-x86_64.egg to /usr/local/lib/python3.7/
↪site-packages
Adding custom-setup-ops 0.0.0 to easy-install.pth file

Installed /usr/local/lib/python3.7/site-packages/custom_setup_ops-0.0.0-py3.7-linux-
↪x86_64.egg
Processing dependencies for custom-setup-ops==0.0.0
Finished processing dependencies for custom-setup-ops==0.0.0

```

执行成功后，如日志所示，自定义算子模块 custom_setup_ops 被安装至如下目录：

```
/usr/local/lib/python3.7/site-packages/custom_setup_ops-0.0.0-py3.
7-linux-x86_64.egg
```

custom_setup_ops-0.0.0-py3.7-linux-x86_64.egg 目录中内容如下:

```
custom_setup_ops_pd_.so  EGG-INFO/      relu_cpu.o      relu_cuda.o
custom_setup_ops.py      __pycache__/  relu_cuda.cu.o  version.txt
```

其中 custom_setup_ops_pd_.so 为自定义算子编译生成的动态库, custom_setup_ops.py 为根据 PaddlePaddle 接口的定义规则, 自动生成的自定义算子 python 模块源码, 其示例内容为 (自动生成的代码后续可能会更新):

```
import os
import sys
import types
import paddle

def inject_ext_module(module_name, api_names):
    if module_name in sys.modules:
        return sys.modules[module_name]

    new_module = types.ModuleType(module_name)
    for api_name in api_names:
        setattr(new_module, api_name, eval(api_name))

    return new_module

def __bootstrap__():
    cur_dir = os.path.dirname(os.path.abspath(__file__))
    so_path = os.path.join(cur_dir, "custom_relu_module_setup_pd_.so")

    assert os.path.exists(so_path)

    # load custom op shared library with abs path
    new_custom_ops = paddle.utils.cpp_extension.load_op_meta_info_and_register_op(so_
    ↩path)
    m = inject_ext_module(__name__, new_custom_ops)

__bootstrap__()

from paddle.fluid.core import VarBase
from paddle.fluid.framework import in_dygraph_mode, _dygraph_tracer
from paddle.fluid.layer_helper import LayerHelper

def custom_relu(x):
    # prepare inputs and outputs
    ins = {'X' : x}
    attrs = {}
```

(下页继续)

(续上页)

```

outs = {}
out_names = ['Out']

# The output variable's dtype use default value 'float32',
# and the actual dtype of output variable will be inferred in runtime.
if in_dygraph_mode():
    for out_name in out_names:
        outs[out_name] = VarBase()
    _dygraph_tracer().trace_op(type="custom_relu", inputs=ins, outputs=outs,
                                attrs=attrs)
else:
    helper = LayerHelper("custom_relu", **locals())
    for out_name in out_names:
        outs[out_name] = helper.create_variable(dtype='float32')

    helper.append_op(type="custom_relu", inputs=ins, outputs=outs, attrs=attrs)

res = [outs[out_name] for out_name in out_names]

return res[0] if len(res)==1 else res

```

随后，可以直接在构建模型过程中导入使用，简单示例如下：

```

import paddle
from custom_setup_ops import custom_relu

x = paddle.randn([4, 10], dtype='float32')
relu_out = custom_relu(x)

```

注：`setupools` 的封装是为了简化自定义算子编译和使用流程，即使不依赖于 `setupools`，也可以自行编译生成动态库，并封装相应的 Python API，然后在基于 PaddlePaddle 实现的模型中使用。

如果需要详细了解相关接口，或需要配置其他编译选项，请参考以下 API 文档：

- `paddle.utils.cpp_extension.setup`
- `paddle.utils.cpp_extension.setupCppExtension`
- `paddle.utils.cpp_extension.CUDAExtension`

即时编译 (JIT Compile)

即时编译将 `setuptools.setup` 编译方式做了进一步的封装，通过将自定义算子对应的 `.cc` 和 `.cu` 文件传入 API `paddle.utils.cpp_extension.load`，在后台生成 `setup.py` 文件，并通过子进程的方式，隐式地执行源码文件编译、符号链接、动态库生成、组网 API 接口生成等一系列过程。不需要本地预装 CMake 或者 Ninja 等工具命令，仅需必要的编译器命令环境。Linux 下需安装版本不低于 5.4 的 GCC，并软链到 `/usr/bin/cc`，Windows 下需安装版本不低于 2017 的 Visual Studio；若编译支持 GPU 设备的算子，则需要提前安装 CUDA，其中自带 nvcc 编译环境。

对于前述 `relu` 示例，使用方式如下：

- for `relu_cuda.cc` & `relu_cuda.cu`

```
import paddle
from paddle.utils.cpp_extension import load

custom_ops = load(
    name="custom_jit_ops",
    sources=["relu_cuda.cc", "relu_cuda.cu"])

x = paddle.randn([4, 10], dtype='float32')
out = custom_ops.custom_relu(x)
```

`load` 返回一个包含自定义算子 API 的 `Module` 对象，可以直接使用自定义算子 `name` 调用 API。

以 Linux 平台为例，`load` 接口调用过程中，如果不指定 `build_directory` 参数，Linux 会默认在 `~/.cache/paddle_extensions` 目录下生成一个 `{name}_setup.py` (Windows 默认目录为 `C:\Users\xxx\.cache\paddle_extensions`)，然后通过 `subprocess` 执行 `python {name}_setup.py build`，然后载入动态库，生成 Python API 之后返回。

对于本示例，默认生成路径内容如下：

```
λ ls ~/.cache/paddle_extensions/
custom_jit_ops/  custom_jit_ops_setup.py
```

其中，`custom_jit_ops_setup.py` 是生成的 `setup` 编译文件，`custom_jit_ops` 目录是编译生成的内容。

如果需要详细了解 `load` 接口，或需要配置其他编译选项，请参考 API 文档 `paddle.utils.cpp_extension.load`。

同时编译多个算子

以上两种方式均支持同时编译多个自定义算子，只需要将多个算子对应的源文件均传入对应的参数，编译生成的动态库中会包含多个算子的实现，导入 Module 之后，同样以算子名作为 API 名进行调用，示例如下：

- setuptools 编译

```
from paddle.utils.cpp_extension import CUDAExtension, setup

setup(
    name='custom_setup_ops',
    ext_modules=CUDAExtension(
        sources=['relu_op.cc', 'relu_op.cu', 'tanh_op.cc', 'tanh_op.cu']
    )
)
```

注：此处需要是多个不同算子的实现，而不能是同一个算子的不同版本实现，例如这里不能将前述的 `relu_cpu.cc` 和 `relu_cuda.cc/cu` 一起编译，因为他们的算子名是相同的，都是 `custom_relu`，如果需要同一个算子在不同设备上的实现，建议将不同设备上的实现整合到一起，例如前述的 `relu.cc/cu`

调用方式：

```
import paddle
# Suppose the op names are `custom_relu` and `custom_tanh`
from custom_ops import custom_relu, custom_tanh

x = paddle.randn([4, 10], dtype='float32')
relu_out = custom_relu(x)
tanh_out = custom_tanh(x)
```

- JIT compile

```
from paddle.utils.cpp_extension import load

custom_ops = load(
    name='custom_jit_ops',
    sources=['relu_op.cc', 'relu_op.cu', 'tanh_op.cc', 'tanh_op.cu'])

x = paddle.randn([4, 10], dtype='float32')
# Suppose the op names are `custom_relu` and `custom_tanh`
relu_out = custom_ops.custom_relu(x)
tanh_out = custom_ops.custom_tanh(x)
```

ABI 兼容性检查

以上两种方式，编译前均会执行 ABI 兼容性检查。对于 Linux，会检查 cc 命令对应的 GCC 版本是否与所安装的 PaddlePaddle 的 GCC 版本一致。例如对于 CUDA 10.1 以上的 PaddlePaddle 默认使用 GCC 8.2 编译，则本地 cc 对应的编译器版本也需为 8.2。对于 Windows，则会检查本地的 Visual Studio 版本是否与所安装的 PaddlePaddle 的 Visual Studio 版本一致 (≥ 2017)。如果上述版本不一致，则会打印出相应 warning，且可能由于引发自定义 OP 编译执行报错。

在模型中使用自定义算子

经过前述过程，自定义算子的编写、编译安装及 API 生成均已完成，现在您可以在网络模型中使用您自定义生成的算子了，本方案生成的自定义算子在动态图和静态图模式下均能够使用。

以下验证用例均基于前述源文件 `relu_cuda.cc` 和 `relu_cuda.cu` 测试 `custom_relu` 在 GPU 环境中的使用，均采用 JIT Compile 的方式编译自定义算子。

通过定义一个简单的网络模型，完成训练迭代和存储推理模型的基本过程。

动态图模式

动态图模式的使用示例如下：

```
import numpy as np

import paddle
import paddle.nn as nn
from paddle.vision.transforms import Compose, Normalize
from paddle.utils.cpp_extension import load

EPOCH_NUM = 4
BATCH_SIZE = 64

# jit compile custom op
custom_ops = load(
    name="custom_jit_ops",
    sources=["relu_cuda.cc", "relu_cuda.cu"])

class LeNet(nn.Layer):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2D(in_channels=1, out_channels=6, kernel_size=5, stride=1,
                           padding=2)
        self.max_pool1 = nn.MaxPool2D(kernel_size=2, stride=2)
```

(下页继续)

(续上页)

```

    self.conv2 = nn.Conv2D(in_channels=6, out_channels=16, kernel_size=5, ↵
    ↵stride=1)
    self.max_pool2 = nn.MaxPool2D(kernel_size=2, stride=2)
    self.linear1 = nn.Linear(in_features=16*5*5, out_features=120)
    self.linear2 = nn.Linear(in_features=120, out_features=84)
    self.linear3 = nn.Linear(in_features=84, out_features=10)

    def forward(self, x):
        x = self.conv1(x)
        x = custom_ops.custom_relu(x)
        x = self.max_pool1(x)
        x = custom_ops.custom_relu(x)
        x = self.conv2(x)
        x = self.max_pool2(x)
        x = paddle.flatten(x, start_axis=1, stop_axis=-1)
        x = self.linear1(x)
        x = custom_ops.custom_relu(x)
        x = self.linear2(x)
        x = custom_ops.custom_relu(x)
        x = self.linear3(x)
        return x

# set device
paddle.set_device("gpu")

# model
net = LeNet()
loss_fn = nn.CrossEntropyLoss()
opt = paddle.optimizer.Adam(learning_rate=0.001, parameters=net.parameters())

# data loader
transform = Compose([Normalize(mean=[127.5],
                               std=[127.5],
                               data_format='CHW')])
train_dataset = paddle.vision.datasets.MNIST(mode='train', transform=transform)
train_loader = paddle.io.DataLoader(train_dataset,
                                    batch_size=BATCH_SIZE,
                                    shuffle=True,
                                    drop_last=True,
                                    num_workers=2)

# train

```

(下页继续)

(续上页)

```

for epoch_id in range(EPOCH_NUM):
    for batch_id, (image, label) in enumerate(train_loader()):
        out = net(image)
        loss = loss_fn(out, label)
        loss.backward()

        if batch_id % 300 == 0:
            print("Epoch {} batch {}: loss = {}".format(
                epoch_id, batch_id, paddle.mean(loss).numpy()))

        opt.step()
        opt.clear_grad()

# save inference model
path = "custom_relu_test_dynamic/net"
paddle.jit.save(net, path,
    input_spec=[paddle.static.InputSpec(shape=[None, 1, 28, 28], dtype='float32')])

```

静态图模式

静态图模式的使用示例如下：

```

import numpy as np

import paddle
import paddle.nn as nn
import paddle.static as static
from paddle.vision.transforms import Compose, Normalize
from paddle.utils.cpp_extension import load

EPOCH_NUM = 4
BATCH_SIZE = 64

# jit compile custom op
custom_ops = load(
    name="custom_jit_ops",
    sources=["relu_cuda.cc", "relu_cuda.cu"])

class LeNet(nn.Layer):
    def __init__(self):
        super(LeNet, self).__init__()

```

(下页继续)

(续上页)

```

    self.conv1 = nn.Conv2D(in_channels=1, out_channels=6, kernel_size=5, stride=1,
↳ padding=2)
    self.max_pool1 = nn.MaxPool2D(kernel_size=2, stride=2)
    self.conv2 = nn.Conv2D(in_channels=6, out_channels=16, kernel_size=5, ↳
↳ stride=1)
    self.max_pool2 = nn.MaxPool2D(kernel_size=2, stride=2)
    self.linear1 = nn.Linear(in_features=16*5*5, out_features=120)
    self.linear2 = nn.Linear(in_features=120, out_features=84)
    self.linear3 = nn.Linear(in_features=84, out_features=10)

def forward(self, x):
    x = self.conv1(x)
    x = custom_ops.custom_relu(x)
    x = self.max_pool1(x)
    x = custom_ops.custom_relu(x)
    x = self.conv2(x)
    x = self.max_pool2(x)
    x = paddle.flatten(x, start_axis=1, stop_axis=-1)
    x = self.linear1(x)
    x = custom_ops.custom_relu(x)
    x = self.linear2(x)
    x = custom_ops.custom_relu(x)
    x = self.linear3(x)
    return x

# set device
paddle.enable_static()
paddle.set_device("gpu")

# model
image = static.data(shape=[None, 1, 28, 28], name='image', dtype='float32')
label = static.data(shape=[None, 1], name='label', dtype='int64')

net = LeNet()
out = net(image)
loss = nn.functional.cross_entropy(out, label)

opt = paddle.optimizer.Adam(learning_rate=0.001)
opt.minimize(loss)

# data loader
transform = Compose([Normalize(mean=[127.5],

```

(下页继续)

(续上页)

```
        std=[127.5],
        data_format='CHW')))

train_dataset = paddle.vision.datasets.MNIST(mode='train', transform=transform)
train_loader = paddle.io.DataLoader(train_dataset,
    feed_list=[image, label],
    batch_size=BATCH_SIZE,
    shuffle=True,
    drop_last=True,
    num_workers=2)

# prepare
exe = static.Executor()
exe.run(static.default_startup_program())

places = paddle.static.cuda_places()
compiled_program = static.CompiledProgram(
    static.default_main_program()).with_data_parallel(
        loss_name=loss.name, places=places)

# train
for epoch_id in range(EPOCH_NUM):
    for batch_id, (image_data, label_data) in enumerate(train_loader()):
        loss_data = exe.run(compiled_program,
            feed={'image': image_data,
                  'label': label_data},
            fetch_list=[loss])
        if batch_id % 300 == 0:
            print("Epoch {} batch {}: loss = {}".format(
                epoch_id, batch_id, np.mean(loss_data)))

# save inference model
path = "custom_relu_test_static/net"
static.save_inference_model(path, [image], [out], exe)
```

算子在推理场景中的使用

基于本机制编写的自定义算子，也能够在 PaddlePaddle 推理场景中使用，仍然基于前述示例介绍使用流程，这里基于 `relu_cuda.cc` 和 `relu_cuda.cu` 介绍。

源码改动

由于训练和推理接口管理上存在一些差别，自定义算子 `relu_cuda.cc` 源码中的引入的头文件需要替换一下：

```
#include "paddle/extension.h"
```

改为

```
#include "paddle/include/experimental/ext_all.h"
```

其他地方不需要做改动。

算子与推理库联合编译

编写推理的测试程序，其中需要使用前述验证过程中存储的 `inference model`，目录为 `custom_relu_dynamic/net` 或者 `custom_relu_static/net`，下面通过示例介绍使用流程，该示例需要准备的文件包括：

```
- cmake
  - external
    - boost.cmake
- CMakeLists.txt
- custom_op_test.cc
- relu_cuda.cc
- relu_cuda.cu
- run.sh
```

下面依次对各新增文件进行介绍。

编写推理程序

下面是一个简单的推理 Demo，导入前述 `custom_relu_dynamic/net` 中存储的模型和参数，进行预测：

```
#include <numeric>
#include <gflags/gflags.h>
#include <glog/logging.h>

#include "paddle/include/paddle_inference_api.h"
```

(下页继续)

(续上页)

```
using paddle_infer::Config;
using paddle_infer::Predictor;
using paddle_infer::CreatePredictor;

void run(Predictor *predictor, const std::vector<float> &input,
         const std::vector<int> &input_shape, std::vector<float> *out_data) {
    auto input_names = predictor->GetInputNames();
    auto input_t = predictor->GetInputHandle(input_names[0]);
    input_t->Reshape(input_shape);
    input_t->CopyFromCpu(input.data());

    CHECK(predictor->Run());

    auto output_names = predictor->GetOutputNames();
    auto output_t = predictor->GetOutputHandle(output_names[0]);
    std::vector<int> output_shape = output_t->shape();
    int out_num = std::accumulate(output_shape.begin(), output_shape.end(), 1,
                                  std::multiplies<int>());
    out_data->resize(out_num);
    output_t->CopyToCpu(out_data->data());
}

int main() {
    paddle::AnalysisConfig config;
    config.EnableUseGpu(100, 0);
    config.SetModel("custom_relu_dynamic/net.pdmodel",
                   "custom_relu_dynamic/net.pdiparams");
    auto predictor{paddle_infer::CreatePredictor(config)};
    std::vector<int> input_shape = {1, 1, 28, 28};
    std::vector<float> input_data(1 * 1 * 28 * 28, 1);
    std::vector<float> out_data;
    run(predictor.get(), input_data, input_shape, &out_data);
    for (auto e : out_data) {
        LOG(INFO) << e << '\n';
    }
    return 0;
}
```

编写 CMake 文件

编写 CMakeList 编译构建文件，示例如下：

在当前目录创建文件 CMakeLists.txt，其内容为：

- CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(cpp_inference_demo CXX C)

option(WITH_MKL      "Compile demo with MKL/OpenBlas support, default use MKL."
      ↵ ON)
option(WITH_GPU       "Compile demo with GPU/CPU, default use CPU."
      ↵ ON)
option(USE_TENSORRT "Compile demo with TensorRT."    ON)
option(CUSTOM_OPERATOR_FILES "List of file names for custom operators" "")

set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} "${CMAKE_CURRENT_SOURCE_DIR}/cmake")

if(WITH_GPU)
    find_package(CUDA REQUIRED)
    add_definitions("-DPADDLE_WITH_CUDA")
endif()

if(NOT WITH_STATIC_LIB)
    add_definitions("-DPADDLE_WITH_SHARED_LIB")
else()
    # PD_INFER_DECL is mainly used to set the dllimport/dllexport attribute in dynamic_
    ↵library mode.
    # Set it to empty in static library mode to avoid compilation issues.
    add_definitions("/DPD_INFER_DECL=")
endif()

macro(safe_set_static_flag)
    foreach(flag_var
        CMAKE_CXX_FLAGS CMAKE_CXX_FLAGS_DEBUG CMAKE_CXX_FLAGS_RELEASE
        CMAKE_CXX_FLAGS_MINSIZEREL CMAKE_CXX_FLAGS_RELWITHDEBINFO)
        if(${flag_var} MATCHES "/MD")
            string(REPLACE "/MD" "/MT" ${flag_var} "${${flag_var}}")
        endif(${flag_var} MATCHES "/MD")
    endforeach(flag_var)
endmacro()

if(NOT DEFINED PADDLE_LIB)
    message(FATAL_ERROR "please set PADDLE_LIB with -DPADDLE_LIB=/path/paddle/lib")
```

(下页继续)

(续上页)

```

endif()
if(NOT DEFINED DEMO_NAME)
    message(FATAL_ERROR "please set DEMO_NAME with -DDEMO_NAME=demo_name")
endif()

include_directories("${PADDLE_LIB}/")
set(PADDLE_LIB_THIRD_PARTY_PATH "${PADDLE_LIB}/third_party/install/")
include_directories("${PADDLE_LIB_THIRD_PARTY_PATH}protobuf/include")
include_directories("${PADDLE_LIB_THIRD_PARTY_PATH}glog/include")
include_directories("${PADDLE_LIB_THIRD_PARTY_PATH}gflags/include")
include_directories("${PADDLE_LIB_THIRD_PARTY_PATH}xxhash/include")

link_directories("${PADDLE_LIB_THIRD_PARTY_PATH}protobuf/lib")
link_directories("${PADDLE_LIB_THIRD_PARTY_PATH}glog/lib")
link_directories("${PADDLE_LIB_THIRD_PARTY_PATH}gflags/lib")
link_directories("${PADDLE_LIB_THIRD_PARTY_PATH}xxhash/lib")
link_directories("${PADDLE_LIB}/paddle/lib")

if (WIN32)
    add_definitions(/DGOOGLE_GLOG_DLL_DECL="")
    option(MSVC_STATIC_CRT "use static C Runtime library by default" ON)
    if (MSVC_STATIC_CRT)
        if (WITH_MKL)
            set(FLAG_OPENMP "/openmp")
        endif()
        set(CMAKE_C_FLAGS_DEBUG    "${CMAKE_C_FLAGS_DEBUG} /bigobj /MTd ${FLAG_OPENMP}")
        set(CMAKE_C_FLAGS_RELEASE   "${CMAKE_C_FLAGS_RELEASE} /bigobj /MT ${FLAG_OPENMP}")
        set(CMAKE_CXX_FLAGS_DEBUG   "${CMAKE_CXX_FLAGS_DEBUG} /bigobj /MTd ${FLAG_OPENMP}")
        set(CMAKE_CXX_FLAGS_RELEASE  "${CMAKE_CXX_FLAGS_RELEASE} /bigobj /MT ${FLAG_
OPENMP}")
        safe_set_static_flag()
        if (WITH_STATIC_LIB)
            add_definitions(-DSTATIC_LIB)
        endif()
    endif()
else()
    if(WITH_MKL)
        set(FLAG_OPENMP "-fopenmp")
    endif()
    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++14 ${FLAG_OPENMP}")
endif()

if(WITH_GPU)

```

(下页继续)

(续上页)

```

if(NOT WIN32)
    set(CUDA_LIB "/usr/local/cuda/lib64/" CACHE STRING "CUDA Library")
else()
    if(CUDA_LIB STREQUAL "")
        set(CUDA_LIB "C:\\Program\\ Files\\NVIDIA GPU Computing Toolkit\\CUDA\\v8.0\\lib\\
→\\x64")
    endif()
endif(NOT WIN32)
endif()

if (USE_TENSORRT AND WITH_GPU)
    set(TENSORRT_ROOT "" CACHE STRING "The root directory of TensorRT library")
    if("${TENSORRT_ROOT}" STREQUAL "")
        message(FATAL_ERROR "The TENSORRT_ROOT is empty, you must assign it a value_
→with CMake command. Such as: -DTENSORRT_ROOT=TENSORRT_ROOT_PATH ")
    endif()
    set(TENSORRT_INCLUDE_DIR ${TENSORRT_ROOT}/include)
    set(TENSORRT_LIB_DIR ${TENSORRT_ROOT}/lib)
endif()

if (NOT WIN32)
    if (USE_TENSORRT AND WITH_GPU)
        include_directories("${TENSORRT_INCLUDE_DIR}")
        link_directories("${TENSORRT_LIB_DIR}")
    endif()
endif(NOT WIN32)

if(WITH_MKL)
    set(MATH_LIB_PATH "${PADDLE_LIB_THIRD_PARTY_PATH}mkml")
    include_directories("${MATH_LIB_PATH}/include")
    if(WIN32)
        set(MATH_LIB ${MATH_LIB_PATH}/lib/mkml${CMAKE_STATIC_LIBRARY_SUFFIX}
            ${MATH_LIB_PATH}/lib/libiomp5md${CMAKE_STATIC_LIBRARY_SUFFIX})
    else()
        set(MATH_LIB ${MATH_LIB_PATH}/lib/libmklml_intel${CMAKE_SHARED_LIBRARY_SUFFIX}
            ${MATH_LIB_PATH}/lib/libiomp5${CMAKE_SHARED_LIBRARY_SUFFIX})
    endif()
    set(MKLDNN_PATH "${PADDLE_LIB_THIRD_PARTY_PATH}mkldnn")
    if(EXISTS ${MKLDNN_PATH})
        include_directories("${MKLDNN_PATH}/include")
        if(WIN32)
            set(MKLDNN_LIB ${MKLDNN_PATH}/lib/mkldnn.lib)
        else(WIN32)

```

(下页继续)

(续上页)

```

    set(MKLDNN_LIB ${MKLDNN_PATH}/lib/libmkldnn.so.0)
endif(WIN32)
endif()
else()
    set(OPENBLAS_LIB_PATH "${PADDLE_LIB_THIRD_PARTY_PATH}openblas")
    include_directories("${OPENBLAS_LIB_PATH}/include/openblas")
    if(WIN32)
        set(MATH_LIB ${OPENBLAS_LIB_PATH}/lib/openblas${CMAKE_STATIC_LIBRARY_SUFFIX})
    else()
        set(MATH_LIB ${OPENBLAS_LIB_PATH}/lib/libopenblas${CMAKE_STATIC_LIBRARY_SUFFIX})
    endif()
endif()

if(WITH_STATIC_LIB)
    set(DEPS ${PADDLE_LIB}/paddle/lib/libpaddle_inference${CMAKE_STATIC_LIBRARY_SUFFIX})
else()
    if(WIN32)
        set(DEPS ${PADDLE_LIB}/paddle/lib/libpaddle_inference${CMAKE_STATIC_LIBRARY_SUFFIX})
    else()
        set(DEPS ${PADDLE_LIB}/paddle/lib/libpaddle_inference${CMAKE_SHARED_LIBRARY_SUFFIX})
    endif()
endif()

if(NOT WIN32)
    set(EXTERNAL_LIB "-lrt -ldl -lpthread")
    set(DEPS ${DEPS}
        ${MATH_LIB} ${MKLDNN_LIB}
        glog gflags protobuf xxhash
        ${EXTERNAL_LIB})
else()
    set(DEPS ${DEPS}
        ${MATH_LIB} ${MKLDNN_LIB}
        glog gflags_static libprotobuf xxhash ${EXTERNAL_LIB})
    set(DEPS ${DEPS} shlwapi.lib)
endif(NOT WIN32)

if(WITH_GPU)
    if(NOT WIN32)
        if(USE_TENSORRT)
            set(DEPS ${DEPS} ${TENSORRT_LIB_DIR}/libnvinfer${CMAKE_SHARED_LIBRARY_SUFFIX})
            set(DEPS ${DEPS} ${TENSORRT_LIB_DIR}/libnvinfer_plugin${CMAKE_SHARED_LIBRARY_SUFFIX})
        endif()
    endif()
endif()

```

(下页继续)

(续上页)

```

endif()
set(DEPS ${DEPS} ${CUDA_LIB}/libcudart${CMAKE_SHARED_LIBRARY_SUFFIX})
else()
if(USE_TENSORRT)
    set(DEPS ${DEPS} ${TENSORRT_LIB_DIR}/nvinfer${CMAKE_STATIC_LIBRARY_SUFFIX})
    set(DEPS ${DEPS} ${TENSORRT_LIB_DIR}/nvinfer_plugin${CMAKE_STATIC_LIBRARY_
→SUFFIX})
endif()
set(DEPS ${DEPS} ${CUDA_LIB}/cudart${CMAKE_STATIC_LIBRARY_SUFFIX} )
set(DEPS ${DEPS} ${CUDA_LIB}/cUBLAS${CMAKE_STATIC_LIBRARY_SUFFIX} )
set(DEPS ${DEPS} ${CUDA_LIB}/cudnn${CMAKE_STATIC_LIBRARY_SUFFIX} )
endif()
endif()

cuda_add_library(pd_infer_custom_op ${CUSTOM_OPERATOR_FILES} SHARED)
add_executable(${DEMO_NAME} ${DEMO_NAME}.cc)
set(DEPS ${DEPS} boost pd_infer_custom_op)

if(WIN32)
    if(USE_TENSORRT)
        add_custom_command(TARGET ${DEMO_NAME} POST_BUILD
            COMMAND ${CMAKE_COMMAND} -E copy ${TENSORRT_LIB_DIR}/nvinfer${CMAKE_-
→SHARED_LIBRARY_SUFFIX}
            ${CMAKE_BINARY_DIR}/${CMAKE_BUILD_TYPE}
            COMMAND ${CMAKE_COMMAND} -E copy ${TENSORRT_LIB_DIR}/nvinfer_plugin$_
→${CMAKE_SHARED_LIBRARY_SUFFIX}
            ${CMAKE_BINARY_DIR}/${CMAKE_BUILD_TYPE}
        )
    endif()
    if(WITH_MKL)
        add_custom_command(TARGET ${DEMO_NAME} POST_BUILD
            COMMAND ${CMAKE_COMMAND} -E copy ${MATH_LIB_PATH}/lib/mklml.dll ${CMAKE_-
→BINARY_DIR}/Release
            COMMAND ${CMAKE_COMMAND} -E copy ${MATH_LIB_PATH}/lib/libiomp5md.dll $_
→${CMAKE_BINARY_DIR}/Release
            COMMAND ${CMAKE_COMMAND} -E copy ${MKLDNN_PATH}/lib/mkldnn.dll ${CMAKE_-
→BINARY_DIR}/Release
        )
    else()
        add_custom_command(TARGET ${DEMO_NAME} POST_BUILD
            COMMAND ${CMAKE_COMMAND} -E copy ${OPENBLAS_LIB_PATH}/lib/openblas.dll $_
→${CMAKE_BINARY_DIR}/Release
        )
    
```

(下页继续)

(续上页)

```

endif()

if(NOT WITH_STATIC_LIB)
    add_custom_command(TARGET ${DEMO_NAME} POST_BUILD
        COMMAND ${CMAKE_COMMAND} -E copy "${PADDLE_LIB}/paddle/lib/paddle_fluid.dll" ${CMAKE_BINARY_DIR}/${CMAKE_BUILD_TYPE}
    )
endif()
endif()

target_link_libraries(${DEMO_NAME} ${DEPS})

```

编写编译执行脚本

编写编译执行脚本 run.sh，脚本内容如下：

- run.sh

```

mkdir -p build
cd build
rm -rf *

DEMO_NAME=custom_op_test

WITH_MKL=ON
WITH_GPU=ON
USE_TENSORRT=OFF

LIB_DIR=${YOUR_LIB_DIR}/paddle_inference_install_dir
CUDNN_LIB=/usr/local/cudnn/lib64
CUDA_LIB=/usr/local/cuda/lib64
TENSORRT_ROOT=/root/work/nvidia/TensorRT-6.0.1.5.cuda-10.1.cudnn7.6-OSS7.2.1
CUSTOM_OPERATOR_FILES="relu_cuda.cc;relu_cuda.cu"

cmake .. -DPADDLE_LIB=${LIB_DIR} \
-DWITH_MKL=${WITH_MKL} \
-DDEMO_NAME=${DEMO_NAME} \
-DWITH_GPU=${WITH_GPU} \
-DWITH_STATIC_LIB=OFF \
-DUSE_TENSORRT=${USE_TENSORRT} \
-DCUDNN_LIB=${CUDNN_LIB} \
-DCUDA_LIB=${CUDA_LIB} \
-DTENSORRT_ROOT=${TENSORRT_ROOT} \

```

(下页继续)

(续上页)

```
-DCUSTOM_OPERATOR_FILES=${CUSTOM_OPERATOR_FILES}  
  
make -j
```

此处要根据实际情况对执行脚本中的几处配置进行调整：

```
# 根据预编译库中的version.txt信息判断是否将以下三个标记打开  
WITH_MKL=ON  
WITH_GPU=ON  
USE_TENSORRT=OFF  
  
# 配置预测库的根目录  
LIB_DIR=${YOUR_LIB_DIR}/paddle_inference_install_dir  
  
# 如果上述的WITH_GPU或USE_TENSORRT设为ON，请设置对应的CUDA, CUDNN, ↵  
→TENSORRT的路径。  
CUDNN_LIB=/paddle/nvidia-downloads/cudnn_v7.5_cuda10.1/lib64  
CUDA_LIB=/paddle/nvidia-downloads/cuda-10.1/lib64  
# TENSORRT_ROOT=/paddle/nvidia-downloads/TensorRT-6.0.1.5
```

然后，运行 sh run.sh，完成编译，会在目录下产生 build 目录。

运行推理程序

```
# 进入build目录  
cd build  
# 运行样例  
../custom_op_test
```

运行结束后，程序会将模型结果打印到屏幕，说明运行成功。

更多推理使用文档

- Paddle Inference 快速开始
- Paddle Inference API 文档

2.9.2 自定义 Python 算子

动态图自定义 Python 算子

Paddle 通过 PyLayer 接口和 PyLayerContext 接口支持动态图的 Python 端自定义 OP。

相关接口概述

PyLayer 接口描述如下：

```
class PyLayer:
    @staticmethod
    def forward(ctx, *args, **kwargs):
        pass

    @staticmethod
    def backward(ctx, *args, **kwargs):
        pass

    @classmethod
    def apply(cls, *args, **kwargs):
        pass
```

其中，

- forward 是自定义 Op 的前向函数，必须被子类重写，它的第一个参数是 PyLayerContext 对象，其他输入参数的类型和数量任意。
- backward 是自定义 Op 的反向函数，必须被子类重写，其第一个参数为 PyLayerContext 对象，其他输入参数为 forward 输出 Tensor 的梯度。它的输出 Tensor 为 forward 输入 Tensor 的梯度。
- apply 是自定义 Op 的执行方法，构建完自定义 Op 后，通过 apply 运行 Op。

PyLayerContext 接口描述如下：

```
class PyLayerContext:
    def save_for_backward(self, *tensors):
        pass

    def saved_tensor(self):
        pass
```

其中，

- save_for_backward 用于暂存 backward 需要的 Tensor，这个 API 只能被调用一次，且只能在 forward 中调用。

- `saved_tensor` 获取被 `save_for_backward` 暂存的 `Tensor`。

如何编写动态图 Python Op

以下以 `tanh` 为例，介绍如何利用 `PyLayer` 编写 Python Op。

- 第一步：创建 `PyLayer` 子类并定义前向函数和反向函数

前向函数和反向函数均由 Python 编写，可以方便地使用 Paddle 相关 API 来实现一个自定义的 OP。需要遵守以下规则：

1. `forward` 和 `backward` 都是静态函数，它们的第一个参数是 `PyLayerContext` 对象。
2. `backward` 除了第一个参数以外，其他参数都是 `forward` 函数的输出 `Tensor` 的梯度，因此，`backward` 输入的 `Tensor` 的数量必须等于 `forward` 输出 `Tensor` 的数量。如果您需在 `backward` 中使用 `forward` 中的 `Tensor`，您可以利用 `save_for_backward` 和 `saved_tensor` 这两个方法传递 `Tensor`。
3. `backward` 的输出可以是 `Tensor` 或者 `list/tuple(Tensor)`，这些 `Tensor` 是 `forward` 输入 `Tensor` 的梯度。因此，`backward` 的输出 `Tensor` 的个数等于 `forward` 输入 `Tensor` 的个数。如果 `backward` 的某个返回值（梯度）在 `forward` 中对应的 `Tensor` 的 `stop_gradient` 属性为 `False`，这个返回值必须是 `Tensor` 类型。

```
import paddle
from paddle.autograd import PyLayer

# 通过创建`PyLayer`子类的方式实现动态图Python Op
class cus_tanh(PyLayer):
    @staticmethod
    def forward(ctx, x):
        y = paddle.tanh(x)
        # ctx 为 PyLayerContext 对象，可以把 y 从 forward 传递到 backward。
        ctx.save_for_backward(y)
        return y

    @staticmethod
    # 因为 forward 只有一个输出，因此除了 ctx 外，backward 只有一个输入。
    def backward(ctx, dy):
        # ctx 为 PyLayerContext 对象， saved_tensor 获取在 forward 时暂存的 y。
        y, = ctx.saved_tensor()
        # 调用 Paddle API 自定义反向计算
        grad = dy * (1 - paddle.square(y))
        # forward 只有一个 Tensor 输入，因此，backward 只有一个输出。
        return grad
```

- 第二步：通过 `apply` 方法组建网络。`apply` 的输入为 `forward` 中除了第一个参数 (`ctx`) 以外的输入，`apply` 的输出即为 `forward` 的输出。

```

data = paddle.randn([2, 3], dtype="float32")
data.stop_gradient = False
# 通过 apply 运行这个 Python 算子
z = cus_tanh.apply(data)
z.mean().backward()

print(data.grad)

```

动态图自定义 Python 算子的注意事项

- 为了从 forward 到 backward 传递信息，您可以在 forward 中给 PyLayerContext 添加临时属性，在 backward 中读取这个属性。如果传递 Tensor 推荐使用 save_for_backward 和 saved_tensor，如果传递非 Tensor 推荐使用添加临时属性的方式。

```

import paddle
from paddle.autograd import PyLayer
import numpy as np

class tanh(PyLayer):
    @staticmethod
    def forward(ctx, x1, func1, func2=paddle.square):
        # 添加临时属性的方式传递 func2
        ctx.func = func2
        y1 = func1(x1)
        # 使用 save_for_backward 传递 y1
        ctx.save_for_backward(y1)
        return y1

    @staticmethod
    def backward(ctx, dy1):
        y1, = ctx.saved_tensor()
        # 获取 func2
        re1 = dy1 * (1 - ctx.func(y1))
        return re1

input1 = paddle.randn([2, 3]).astype("float64")
input2 = input1.detach().clone()
input1.stop_gradient = False
input2.stop_gradient = False
z = tanh.apply(x1=input1, func1=paddle.tanh)

```

- forward 的输入和输出的类型任意，但是至少有一个输入和输出为 Tensor 类型。

```
# 错误示例
class cus_tanh(PyLayer):
    @staticmethod
    def forward(ctx, x1, x2):
        y = x1+x2
        # y.shape: 列表类型，非Tensor, 输出至少包含一个Tensor
        return y.shape

    @staticmethod
    def backward(ctx, dy):
        return dy, dy

data = paddle.randn([2, 3], dtype="float32")
data.stop_gradient = False
# 由于forward输出没有Tensor引发报错
z, y_shape = cus_tanh.apply(data, data)

# 正确示例
class cus_tanh(PyLayer):
    @staticmethod
    def forward(ctx, x1, x2):
        y = x1+x2
        # y.shape: 列表类型，非Tensor
        return y, y.shape

    @staticmethod
    def backward(ctx, dy):
        # forward两个Tensor输入，因此，backward有两个输出。
        return dy, dy

data = paddle.randn([2, 3], dtype="float32")
data.stop_gradient = False
z, y_shape = cus_tanh.apply(data, data)
z.mean().backward()

print(data.grad)
```

- 如果 forward 的某个输入为 Tensor 且 stop_gredient = True，则在 backward 中与其对应的返回值应为 None。

```
class cus_tanh(PyLayer):
    @staticmethod
    def forward(ctx, x1, x2):
```

(下页继续)

(续上页)

```

y = x1+x2
return y

@staticmethod
def backward(ctx, dy):
    # x2.stop_gradient=True, 其对应梯度需要返回None
    return dy, None

data1 = paddle.randn([2, 3], dtype="float32")
data1.stop_gradient = False
data2 = paddle.randn([2, 3], dtype="float32")
z = cus_tanh.apply(data1, data2)
fake_loss = z.mean()
fake_loss.backward()
print(data1.grad)

```

- 如果 forward 的所有输入 Tensor 都是 stop_gradient = True 的，则 backward 不会被执行。

```

class cus_tanh(PyLayer):
    @staticmethod
    def forward(ctx, x1, x2):
        y = x1+x2
        return y

    @staticmethod
    def backward(ctx, dy):
        return dy, None

data1 = paddle.randn([2, 3], dtype="float32")
data2 = paddle.randn([2, 3], dtype="float32")
z = cus_tanh.apply(data1, data2)
fake_loss = z.mean()
fake_loss.backward()
# 因为 data1.stop_gradient = True、data2.stop_gradient = True，所以 backward 不会被执行。
print(data1.grad is None)

```

静态图自定义 Python 算子

Paddle 通过 `py_func` 接口支持静态图的 Python 端自定义 OP。`py_func` 的设计原理在于 Paddle 中的 Tensor 可以与 numpy 数组可以方便的互相转换，从而可以使用 Python 中的 numpy API 来自定义一个 Python OP。

`py_func` 接口概述

`py_func` 具体接口为：

```
def py_func(func, x, out, backward_func=None, skip_vars_in_backward_input=None):
    pass
```

其中，

- `x` 是 Python Op 的输入变量，可以是单个 `Tensor | tuple[Tensor] | list[Tensor]`。多个 `Tensor` 以 `tuple[Tensor]` 或 `list[Tensor]` 的形式传入。
- `out` 是 Python Op 的输出变量，可以是单个 `Tensor | tuple[Tensor] | list[Tensor]`，也可以是 Numpy Array。
- `func` 是 Python Op 的前向函数。在运行网络前向时，框架会调用 `out = func(*x)`，根据前向输入 `x` 和前向函数 `func` 计算前向输出 `out`。在 `func` 建议先主动将 `Tensor` 转换为 numpy 数组，方便灵活的使用 numpy 相关的操作，如果未转换成 numpy，则可能某些操作无法兼容。
- `backward_func` 是 Python Op 的反向函数。若 `backward_func` 为 `None`，则该 Python Op 没有反向计算逻辑；若 `backward_func` 不为 `None`，则框架会在运行网路反向时调用 `backward_func` 计算前向输入 `x` 的梯度。
- `skip_vars_in_backward_input` 为反向函数 `backward_func` 中不需要的输入，可以是单个 `Tensor | tuple[Tensor] | list[Tensor]`。

如何使用 `py_func` 编写 Python Op

以下以 `tanh` 为例，介绍如何利用 `py_func` 编写 Python Op。

- 第一步：定义前向函数和反向函数

前向函数和反向函数均由 Python 编写，可以方便地使用 Python 与 numpy 中的相关 API 来实现一个自定义的 OP。

若前向函数的输入为 `x_1, x_2, ..., x_n`，输出为 `y_1, y_2, ..., y_m`，则前向函数的定义格式为：

```
def forward_func(x_1, x_2, ..., x_n):
    ...
    return y_1, y_2, ..., y_m
```

默认情况下，反向函数的输入参数顺序为：所有前向输入变量 + 所有前向输出变量 + 所有前向输出变量的梯度，因此对应的反向函数的定义格式为：

```
def backward_func(x_1, x_2, ..., x_n, y_1, y_2, ..., y_m, dy_1, dy_2, ..., dy_m):
    ...
    return dx_1, dx_2, ..., dx_n
```

若反向函数不需要某些前向输入变量或前向输出变量，可设置 `skip_vars_in_backward_input` 进行排除（步骤三中会叙述具体的排除方法）。

注：`x_1, ..., x_n` 为输入的多个 Tensor，请以 `tuple(Tensor)` 或 `list[Tensor]` 的形式在 `py_func` 中传入。建议先主动将 Tensor 通过 `numpy.array` 转换为数组，否则 Python 与 numpy 中的某些操作可能无法兼容使用在 Tensor 上。

此处我们利用 numpy 的相关 API 完成 `tanh` 的前向函数和反向函数编写。下面给出多个前向与反向函数定义的示例：

```
import numpy as np

# 前向函数1：模拟tanh激活函数
def tanh(x):
    # 可以直接将Tensor作为np.tanh的输入参数
    return np.tanh(x)

# 前向函数2：将两个2-D Tensor相加，输入多个Tensor以list[Tensor]或tuple(Tensor)形式
def element_wise_add(x, y):
    # 必须先手动将Tensor转换为numpy数组，否则无法支持numpy的shape操作
    x = np.array(x)
    y = np.array(y)

    if x.shape != y.shape:
        raise AssertionError("the shape of inputs must be the same!")

    result = np.zeros(x.shape, dtype='int32')
    for i in range(len(x)):
        for j in range(len(x[0])):
            result[i][j] = x[i][j] + y[i][j]

    return result

# 前向函数3：可用于调试正在运行的网络（打印值）
def debug_func(x):
    # 可以直接将Tensor作为print的输入参数
    print(x)

# 前向函数1对应的反向函数，默认的输入顺序为：x、out、out的梯度
def tanh_grad(x, y, dy):
```

(下页继续)

(续上页)

```
# 必须先手动将Tensor转换为numpy数组，否则"+/-"等操作无法使用
return np.array(dy) * (1 - np.square(np.array(y)))
```

注意，前向函数和反向函数的输入均是 Tensor 类型，输出可以是 Numpy Array 或 Tensor。由于 Tensor 实现了 Python 的 buffer protocol 协议，因此即可通过 numpy.array 直接将 Tensor 转换为 numpy Array 来进行操作，也可直接将 Tensor 作为 numpy 函数的输入参数。但建议先主动转换为 numpy Array，则可以任意的使用 python 与 numpy 中的所有操作（例如“numpy array 的 +/-shape”）。

tanh 的反向函数不需要前向输入 x，因此我们可定义一个不需要前向输入 x 的反向函数，并在后续通过 skip_vars_in_backward_input 进行排除：

```
def tanh_grad_without_x(y, dy):
    return np.array(dy) * (1 - np.square(np.array(y)))
```

- 第二步：创建前向输出变量

我们需调用 Program.current_block().create_var 创建前向输出变量。在创建前向输出变量时，必须指明变量的名称 name、数据类型 dtype 和维度 shape。

```
import paddle

paddle.enable_static()

def create_tmp_var(program, name, dtype, shape):
    return program.current_block().create_var(name=name, dtype=dtype, shape=shape)

in_var = paddle.static.data(name='input', dtype='float32', shape=[-1, 28, 28])

# 手动创建前向输出变量
out_var = create_tmp_var(paddle.static.default_main_program(), name='output', dtype=
                           'float32', shape=[-1, 28, 28])
```

- 第三步：调用 py_func 组建网络

py_func 的调用方式为：

```
paddle.static.nn.py_func(func=tanh, x=in_var, out=out_var, backward_func=tanh_grad)
```

若我们不希望在反向函数输入参数中出现前向输入，则可使用 skip_vars_in_backward_input 进行排查，简化反向函数的参数列表。

```
paddle.static.nn.py_func(func=tanh, x=in_var, out=out_var, backward_func=tanh_grad_
                           ↪without_x,
                           skip_vars_in_backward_input=in_var)
```

至此，使用 py_func 编写 Python Op 的步骤结束。我们可以与使用其他 Op 一样进行网路训练/预测。

静态图自定义 Python 算子注意事项

- `py_func` 的前向函数和反向函数内部不应调用 `paddle.xx` 组网接口，因为前向函数和反向函数是在网络运行时调用的，而 `paddle.xx` 是在组建网络的阶段调用。
- `skip_vars_in_backward_input` 只能跳过前向输入变量和前向输出变量，不能跳过前向输出的梯度。
- 若某个前向输出变量没有梯度，则 `backward_func` 将接收到 `None` 的输入。若某个前向输入变量没有梯度，则我们应在 `backward_func` 中主动返回 `None`。

2.10 环境变量 FLAGS

2.10.1 调用说明

PaddlePaddle 中的环境变量 FLAGS 支持两种设置方式。

- 通过 `export` 来设置环境变量，如 `export FLAGS_eager_delete_tensor_gb = 1.0`。
- 通过 API: `get_flag` 和 `set_flags` 来打印和设置环境变量 FLAGS。API 使用详情请参考 `cn_api_paddle_get_flags` 与 `cn_api_paddle_set_flags`。

2.10.2 环境变量 FLAGS 功能分类

cudnn

FLAGS_conv_workspace_size_limit

(始于 0.13.0)

用于选择 cuDNN 卷积算法的工作区限制大小（单位为 MB）。cuDNN 的内部函数在这个内存限制范围内获得速度最快的匹配算法。通常，在较大的工作区内可以选择更快的算法，但同时也会显著增加内存空间。用户需要在内存和速度之间进行权衡。

取值范围

Uint64 型，缺省值为 512。即 512MB 显存工作区。

示例

FLAGS_conv_workspace_size_limit=1024 - 将用于选择 cuDNN 卷积算法的工作区限制大小设置为 1024MB。

FLAGS_cudnn_batchnorm_spatial_persistent

(始于 1.4.0)

表示是否在 batchnorm 中使用新的批量标准化模式 CUDNN_BATCHNORM_SPATIAL_PERSISTENT 函数。

取值范围

Bool 型，缺省值为 False。

示例

FLAGS_cudnn_batchnorm_spatial_persistent=True - 开启 CUDNN_BATCHNORM_SPATIAL_PERSISTENT 模式。

注意

此模式在某些任务中可以更快，因为将为 CUDNN_DATA_FLOAT 和 CUDNN_DATA_HALF 数据类型选择优化路径。我们默认将其设置为 False 的原因是此模式可能使用原子整数缩减 (scaled atomic integer reduction) 而导致某些输入数据范围的数字溢出。

FLAGS_cudnn_deterministic

(始于 0.13.0)

cuDNN 对于同一操作有几种算法，一些算法结果是非确定性的，如卷积算法。该 flag 用于调试。它表示是否选择 cuDNN 中的确定性函数。

取值范围

Bool 型，缺省值为 False。

示例

FLAGS_cudnn_deterministic=True - 选择 cuDNN 中的确定性函数。

注意

现在，在 cuDNN 卷积和池化 Operator 中启用此 flag。确定性算法速度可能较慢，因此该 flag 通常用于调试。

FLAGS_cudnn_exhaustive_search

(始于 1.2.0)

表示是否使用穷举搜索方法来选择卷积算法。在 cuDNN 中有两种搜索方法，启发式搜索和穷举搜索。穷举搜索尝试所有 cuDNN 算法以选择其中最快的算法。此方法非常耗时，所选择的算法将针对给定的层规格进行缓存。一旦更改了图层规格（如 batch 大小，feature map 大小），它将再次搜索。

取值范围

Bool 型，缺省值为 False。

示例

FLAGS_cudnn_exhaustive_search=True - 使用穷举搜索方法来选择卷积算法。

数值计算

FLAGS_enable_cublas_tensor_op_math

(始于 1.2.0)

该 flag 表示是否使用 Tensor Core，但可能会因此降低部分精确度。

取值范围

Bool 型，缺省值为 False。

示例

FLAGS_enable_cublas_tensor_op_math=True - 使用 Tensor Core。

FLAGS_use_mkldnn

(始于 0.13.0)

在预测或训练过程中，可以通过该选项选择使用 Intel MKL-DNN (<https://github.com/intel/mkl-dnn>) 库运行。“用于深度神经网络的英特尔 (R) 数学核心库 (Intel(R) MKL-DNN)”是一个用于深度学习应用程序的开源性能库。该库加速了英特尔 (R) 架构上的深度学习应用程序和框架。Intel MKL-DNN 包含矢量化和线程化构建块，您可以使用它们来实现具有 C 和 C ++ 接口的深度神经网络 (DNN)。

取值范围

Bool 型，缺省值为 False。

示例

FLAGS_use_mkldnn=True - 开启使用 MKL-DNN 运行。

注意

FLAGS_use_mkldnn 仅用于 python 训练和预测脚本。要在 CAPI 中启用 MKL-DNN，请设置选项 -DWITH_MKLDNN=ON。英特尔 MKL-DNN 支持英特尔 64 架构和兼容架构。该库对基于以下设备的系统进行了优化：英特尔 SSE4.1 支持的英特尔凌动 (R) 处理器；第 4 代，第 5 代，第 6 代，第 7 代和第 8 代英特尔 (R) Core (TM) 处理器；英特尔 (R) Xeon (R) 处理器 E3, E5 和 E7 系列（原 Sandy Bridge, Ivy Bridge, Haswell 和 Broadwell）；英特尔 (R) Xeon (R) 可扩展处理器（原 Skylake 和 Cascade Lake）；英特尔 (R) Xeon Phi (TM) 处理器（原 Knights Landing and Knights Mill）；兼容处理器。

调试

FLAGS_check_nan_inf

(始于 0.13.0)

用于调试。它用于检查 Operator 的结果是否含有 Nan 或 Inf。

取值范围

Bool 型，缺省值为 False。

示例

FLAGS_check_nan_inf=True - 检查 Operator 的结果是否含有 Nan 或 Inf。

FLAGS_cpu_deterministic

(始于 0.15.0)

该 flag 用于调试。它表示是否在 CPU 侧确定计算结果。在某些情况下，不同求和次序的结果可能不同，例如， $a+b+c+d$ 的结果可能与 $c+a+b+d$ 的结果不同。

取值范围

Bool 型，缺省值为 False。

示例

FLAGS_cpu_deterministic=True - 在 CPU 侧确定计算结果。

FLAGS_enable_rpc_profiler

(始于 1.0.0)

是否启用 RPC 分析器。

取值范围

Bool 型，缺省值为 False。

示例

FLAGS_enable_rpc_profiler=True - 启用 RPC 分析器并在分析器文件中记录时间线。

FLAGS_multiple_of_cupti_buffer_size

(始于 1.4.0)

该 flag 用于分析。它表示 CUPTI 设备缓冲区大小的倍数。如果在 profiler 过程中程序挂掉或者在 chrome://tracing 中加载 timeline 文件时出现异常，请尝试增大此值。

取值范围

Int32 型，缺省值为 1。

示例

FLAGS_multiple_of_cupti_buffer_size=1 - 将 CUPTI 设备缓冲区大小的倍数设为 1。

FLAGS_reader_queue_speed_test_mode

(始于 1.1.0)

将 pyreader 数据队列设置为测试模式。在测试模式下，pyreader 将缓存一些数据，然后执行器将读取缓存的数据，因此阅读器不会成为瓶颈。

取值范围

Bool 型，缺省值为 False。

示例

FLAGS_reader_queue_speed_test_mode=True - 启用 pyreader 测试模式。

注意

仅当使用 py_reader 时该 flag 才有效。

check nan inf 工具

check nan inf 工具用于检查 Operator 的结果是否含有 nan(not a number, 非有效数) 或 inf(infinite, 无穷大数)。支持 float32、double、float16 三类浮点型，整型由于不存在 nan、inf 不作检查。

使用

1. 使用方法

设置环境变量为 FLAGS_check_nan_inf 为 True 或者 1 即可。

```
export FLAGS_check_nan_inf=1    # 或者=True
```

2. 进阶使用

添加上述环境变量后，可以通过设置环境变量跳过 op、op 类型及 op 变量的检查。设置的格式如下：

```
PADDLE_INF_NAN_SKIP_OP="op0,op1,op2"
PADDLE_INF_NAN_SKIP_ROLE="role1,role2,role3"
PADDLE_INF_NAN_SKIP_VAR="op0:var0,op0:var1,op1:var0"
```

其中上面三个环境变量分别表示跳过 op、op 类型和 op 里变量的检查。

2.1 跳过 op 检查

如下设置中前一个只跳过 mul op 的 nan inf 检查，后一个设置则跳过 mul、softmax_with_cross_entropy 这两个 op 的检查。注意：op 跳过只接受精准匹配，要跳过 softmax_with_cross_entropy 的检查，不能设置环境变量为 softmax_with 或者 with_cross 进行模糊匹配，必须设置 softmax_with_cross_entropy 全名。

```
export PADDLE_INF_NAN_SKIP_OP="mul"
export PADDLE_INF_NAN_SKIP_OP="mul,softmax_with_cross_entropy"
```

2.2 跳过 op 类型检查

目前接受的类型有：forward、backward、optimize、rpc、dist、lrsched、loss、default。正常 fp32 训练中，不需要跳过 op 类型进行 nan inf 检查。但在 fp16 中，在反向过程出现 inf 会对其进行修正，所以一般需要跳过 backward 的检查，这也是添加该功能的缘由。如下设置中前一个只跳过 backward 的检查，后一个设置跳过 backward、optimize 两种类型的检查。同上，op 类型跳过也只支持精准匹配。

```
export PADDLE_INF_NAN_SKIP_ROLE="backward"
export PADDLE_INF_NAN_SKIP_ROLE="backward,optimize"
```

2.3 跳过指定 op 中变量的检查

如下设置中前一个跳过 mul op 中 fc_0.tmp_0 变量，后一个设置则跳过 mul op 中 fc_0.tmp_0 和 fc_0.tmp_1 变量及 dropout op 的 new_relative 变量。

```
export PADDLE_INF_NAN_SKIP_VAR="mul:fc_0.tmp_0"
export PADDLE_INF_NAN_SKIP_VAR="mul:fc_0.tmp_0,mul:fc_0.tmp_1,dropout:new_relative"
```

注意：指定 op 变量检查中，对于 op 只接受精准匹配，对于变量则为模糊匹配，如上述的 mlu op 中的 fc_0.tmp_0 和 fc_0.tmp_1 变量可用 c_0.tmp 进行匹配。

试用

可以使用单测中的[check_nan_inf_base.py](#)文件进行试用。该脚本已设置 `FLAGS_check_nan_inf=1` 打开 check nan inf 功能。直接 `python check_nan_inf_base.py` 执行即可。

1. GPU 日志信息

其中 GPU 的 `check nan` 信息由于在 GPU 中打印，所以 `nan inf` 信息会出现在出错信息栈前面。工具中会打印出现 `inf`、`nan` 的 `op` 及 `tensor` 名称，每个 `block` 会打印 `nan`、`inf`、`num` 中的 3 个值，并打印各自 `block` 中 `nan`、`inf`、`num` 的数量。

```
numel:2000 idx:1928 value:inf
numel:2000 idx:1932 value:inf
In block 0, there has 400,0,624 nan,inf,num
In block 1, there has 281,106,576 nan,inf,num
Error: /paddle/paddle/fluid/framework/details/nan_inf_utils_detail.cu:88 Assertion `false' failed. ===ERROR: in [op=mul] [tensor=fc_0.tmp_0] find nan or inf===
Error: /paddle/paddle/fluid/framework/details/nan_inf_utils_detail.cu:88 Assertion `false' failed. ===ERROR: in [op=mul] [tensor=fc_0.tmp_0] find nan or inf===
/paddle/build/build_ubuntu_wxdev_debug_gpu_n_grpc/python/paddle/fluid/executor.py:773: UserWarning: The following exception is not an EOF exception.
  "The following exception is not an EOF exception."
-----
C++ Call Stacks (More useful to developers):
-----
```

2. CPU 日志信息

CPU 中打印的 nan、inf、num 会在出错信息栈前面显示，同样打印了 nan、inf、num 中的三个值，并打印 nan、inf、num 的数量。check nan 信息中 op 及 tensor 的名称会在最后显示。

```
numel:2000 index:401 value:0.013274
numel:2000 index:402 value:-0.213966
numel:2000 index:1602 value:-inf
numel:2000 index:1605 value:inf
numel:2000 index:1614 value:inf
In cpu, there has 705,95,1200 nan,inf,num
```

```
-----  
C++ Call Stacks (More useful to developers):  
-----
```

```
-----  
Error Message Summary:  
-----
```

```
PreconditionNotMetError: ===ERROR: in [op=mul] [tensor=fc_3.tmp_0] find nan or inf===
[Hint: Expected has_nan_inf == false, but received has_nan_inf:1 != false:0.] at (/paddle
[operator < mul > error]
```

速度

测试环境：v100 32G 单卡测试，Resnet50 模型，imagenet 数据集。不同环境模型数据集下速度可能不同，以下速度仅供参考

不检查 nan inf 速度，每张卡 307.7 images/s。检查 nan inf 速度，每张卡 250.2 images/s。

原理

1. 工具原理

对于浮点类型操作，正常数值 num，无穷大 inf，非数值 nan 有如下运行关系。更详细可查看INF, NAN, and NULL

```
nan - nan = nan, inf - inf = nan, num - num = 0,
nan + nan = nan, inf + inf = inf, nan + 0 = nan,
inf + 0 = inf, nan + inf = nan, 0 + 0 = 0
```

基于此使用如下操作仅需最后检查 sum 是否为 nan 或者 inf 就行了。

```
for (value:values) : sum += (value-value)
```

注意：本文档的进阶使用、速度、原理目前仅在 *develop* 版本的 *paddle* 生效，并将随 *1.7* 版本的 *paddle* 发布。此前版本的 *check nan inf* 工具在 *GPU* 上不推荐使用，旧工具速度为 *0.25 images/s*，测试会拖慢 *1000* 多倍的训练速度。

设备管理

FLAGS_paddle_num_threads

(始于 0.15.0)

控制每个 paddle 实例的线程数。

取值范围

Int32 型，缺省值为 1。

示例

FLAGS_paddle_num_threads=2 - 将每个实例的最大线程数设为 2。

FLAGS_selected_gpus

(始于 1.3)

设置用于训练或预测的 GPU 设备。

取值范围

以逗号分隔的设备 ID 列表，其中每个设备 ID 是一个非负整数，且应小于您的机器拥有的 GPU 设备总数。

示例

FLAGS_selected_gpus=0,1,2,3,4,5,6,7 - 令 0-7 号 GPU 设备用于训练和预测。

注意

使用该 flag 的原因是希望在 GPU 设备之间使用聚合通信，但通过 CUDA_VISIBLE_DEVICES 只能使用共享内存。

分布式

FLAGS_communicator_fake_rpc

Title underline too short.

```
FLAGS_communicator_fake_rpc
*****
```

(始于 1.5.0)

当设为 True 时，通信器不会实际进行 rpc 调用，因此速度不会受到网络通信的影响。该 flag 用于调试。

取值范围

Bool 型，缺省值为 False。

示例

FLAGS_communicator_fake_rpc=True - 启用通信器 fake 模式。

注意

该 flag 仅用于 paddlepaddle 的开发者，普通用户不应用其设置。

FLAGS_communicator_independent_recv_thread

Title underline too short.

```
FLAGS_communicator_independent_recv_thread
*****
```

(始于 1.5.0)

使用独立线程以从参数服务器接收参数。

取值范围

Bool 型，缺省值为 True。

示例

FLAGS_communicator_independent_recv_thread=True - 使用独立线程以从参数服务器接收参数。

注意

开发者使用该 flag 进行框架的调试与优化，普通用户不应对设置。

FLAGS_communicator_max_merge_var_num

(始于 1.5.0)

要通过通信器合并为一个梯度并发送的最大梯度数。训练器将所有梯度放入队列，然后通信器将从队列中取出梯度并在合并后发送。

取值范围

Int32 型，缺省值为 20。

示例

FLAGS_communicator_max_merge_var_num=16 - 将要通过通信器合并为一个梯度并发送的最大梯度数设为 16。

注意

该 flag 和训练器线程数有着密切关联，缺省值应和线程数一致。

FLAGS_communicator_merge_sparse_grad

(始于 1.5.0)

在发送之前，合并稀疏梯度。

取值范围

Bool 型，缺省值 true。

示例

FLAGS_communicator_merge_sparse_grad=true - 设置合并稀疏梯度。

注意

合并稀疏梯度会耗费时间。如果重复 ID 较多，内存占用会变少，通信会变快；如果重复 ID 较少，则并不会节约内存。

FLAGS_communicator_min_send_grad_num_before_recv

Title underline too short.

```
FLAGS_communicator_min_send_grad_num_before_recv  
*****
```

(始于 1.5.0)

在通信器中，有一个发送线程向参数服务器发送梯度，一个接收线程从参数服务器接收参数，且它们之间彼此独立。该 flag 用于控制接收线程的频率。仅当发送线程至少发送 FLAGS_communicator_min_send_grad_num_before_recv 数量的梯度时，接收线程才会从参数服务器接收参数。

取值范围

Int32 型，缺省值为 20。

示例

FLAGS_communicator_min_send_grad_num_before_recv=10 - 在接收线程从参数服务器接收参数之前，发送线程发送的梯度数为 10。

注意

由于该 flag 和训练器的训练线程数强相关，而每个训练线程都会发送其梯度，所以缺省值应和线程数一致。

FLAGS_communicator_send_queue_size

(始于 1.5.0)

每个梯度的队列大小。训练器将梯度放入队列，然后通信器将其从队列中取出并发送出去。当通信器很慢时，队列可能会满，训练器在队列有空间之前被持续阻塞。它用于避免训练比通信快得多，以致太多的梯度没有及时发出的情况。

取值范围

Int32 型，缺省值为 20。

示例

FLAGS_communicator_send_queue_size=10 - 设置每个梯度的队列大小为 10。

注意

该 flag 会影响训练速度，若队列大小过大，速度会变快但结果可能会变差。

FLAGS_communicator_send_wait_times

(始于 1.5.0)

合并数没有达到 max_merge_var_num 的情况下发送线程等待的次数。

取值范围

Int32 型，缺省值为 5。

示例

FLAGS_communicator_send_wait_times=5 - 将合并数没有达到 max_merge_var_num 的情况下发送线程等待的次数设为 5。

FLAGS_communicator_thread_pool_size

(始于 1.5.0)

设置用于发送梯度和接收参数的线程池大小。

取值范围

Int32 型，缺省值为 5。

示例

FLAGS_communicator_thread_pool_size=10 - 设置线程池大小为 10。

注意

大部分情况下，用户不需要设置该 flag。

FLAGS_dist_threadpool_size

(始于 1.0.0)

控制用于分布式模块的线程数。如果未设置，则将其设置为硬线程。

取值范围

Int32 型，缺省值为 0。

示例

FLAGS_dist_threadpool_size=10 - 将用于分布式模块的最大线程数设为 10。

FLAGS_rpc_deadline

(始于 1.0.0)

它控制 rpc 通信的 deadline 超时。

取值范围

Int32 型，缺省值为 180000，单位为 ms。

示例

FLAGS_rpc_deadline=180000 - 将 deadline 超时设为 3 分钟。

FLAGS_rpc_disable_reuse_port

(始于 1.2.0)

FLAGS_rpc_disable_reuse_port 为 True 时，grpc 的 GRPC_ARG_ALLOW_REUSEPORT 会被设置为 False 以禁用 SO_REUSEPORT。

取值范围

Bool 型，缺省值为 False。

示例

FLAGS_rpc_disable_reuse_port=True - 禁用 SO_REUSEPORT。

FLAGS_rpc_get_thread_num

(始于 1.0.0)

它控制用于从参数服务器获取参数的线程数。

取值范围

Int32 型，缺省值为 12。

示例

FLAGS_rpc_get_thread_num=6 - 将从参数服务器获取参数的线程数设为 6。

FLAGS_rpc_send_thread_num

(始于 1.0.0)

它控制用于发送 rpc 的线程数。

取值范围

Int32 型，缺省值为 12。

示例

FLAGS_rpc_send_thread_num=6 - 将用于发送的线程数设为 6。

FLAGS_rpc_server_profile_path

since(v0.15.0)

设置分析器输出日志文件路径前缀。完整路径为 FLAGS_rpc_server_profile_path_listener_id，其中 listener_id 为随机数。

取值范围

String 型，缺省值为”./profile_ps”。

示例

FLAGS_rpc_server_profile_path=”/tmp/pserver_profile_log” - 在”/tmp/pserver_profile_log_listener_id” 中生成配置日志文件。

FLAGS_apply_pass_to_program

since(v2.2.0)

它控制当使用 Fleet API 时，是否在 Program 上使用 IR Pass 优化。

取值范围

Bool 型，缺省值为 false。

示例

FLAGS_apply_pass_to_program=true - 当使用 Fleet API 时，在 Program 上使用 IR Pass 优化。

FLAGS_allreduce_record_one_event

since(v2.2.0)

使 allreduce 操作只等待一个事件而不是多个事件。目前只适用于 fuse allreduce 的场景，否则精度会有误。

取值范围

Bool 型，缺省值为 false。

示例

FLAGS_allreduce_record_one_event=true - 使 allreduce 操作只等待一个事件而不是多个事件。

执行器

FLAGS_enable_parallel_graph

(始于 1.2.0)

该 flag 用于 ParallelExecutor 以禁用并行图执行模式。

取值范围

Bool 型，缺省值为 False。

示例

FLAGS_enable_parallel_graph=False - 通过 ParallelExecutor 强制禁用并行图执行模式。

FLAGS_pe_profile_fname

(始于 1.3.0)

该 flag 用于 ParallelExecutor 的调试。ParallelExecutor 会通过 gpertools 生成配置文件结果，并将结果存储在 FLAGS_pe_profile_fname 指定的文件中。仅在编译选项选择 *WITH_PROFILER=ON* 时有效。如果禁用则设为 empty。

取值范围

String 型，缺省值为 empty ("")。

示例

FLAGS_pe_profile_fname="./parallel_executor.perf" - 将配置文件结果存储在 parallel_executor.perf 中。

FLAGS_print_sub_graph_dir

(始于 1.2.0)

该 flag 用于调试。如果程序中转换图的某些子图失去连接，则结果可能会出错。我们可以将这些断开连接的子图打印到该 flag 指定的文件中。如果禁用则设为 empty。

取值范围

String 型，缺省值为 empty ("")。

示例

FLAGS_print_sub_graph_dir=./sub_graphs.txt - 将断开连接的子图打印到”./sub_graphs.txt”。

FLAGS_use_ngraph

(始于 1.4.0)

在预测或训练过程中，可以通过该选项选择使用英特尔 nGraph (<https://github.com/NervanaSystems/ngraph>) 引擎。它将在英特尔 Xeon CPU 上获得很大的性能提升。

取值范围

Bool 型，缺省值为 False。

示例

FLAGS_use_ngraph=True - 开启使用 nGraph 运行。

注意

英特尔 nGraph 目前仅在少数模型中支持。我们只验证了 [ResNet-50] (https://github.com/PaddlePaddle/models/blob/develop/PaddleCV/image_classification/README_ngraph.md) 的训练和预测。

存储管理

FLAGS_allocator_strategy

Title underline too short.

```
FLAGS_allocator_strategy
*****
```

(始于 1.2)

用于选择 PaddlePaddle 的分配器策略。

取值范围

String 型, ['naive_best_fit', 'auto_growth'] 中的一个。缺省值如果编译 Paddle CMake 时使用-DON_INFER=ON 为'naive_best_fit'。其他默认情况为'auto_growth'。PaddlePaddle pip 安装包的默认策略也是'auto_growth'

示例

FLAGS_allocator_strategy=naive_best_fit - 使用预分配 best fit 分配器, PaddlePaddle 会先占用大多比例的可用内存/显存, 在 Paddle 具体数据使用时分配, 这种方式预占空间较大, 但内存/显存碎片较少(比如能够支持模型的最大 batch size 会变大)。

FLAGS_allocator_strategy=auto_growth - 使用 auto growth 分配器。PaddlePaddle 会随着真实数据需要再占用内存/显存, 但内存/显存可能会产生碎片(比如能够支持模型的最大 batch size 会变小)。

FLAGS_eager_delete_scope

(始于 0.12.0)

同步局域删除。设置后, 它将降低 GPU 内存使用量, 但同时也会减慢销毁变量的速度(性能损害约 1%)。

取值范围

Bool 型, 缺省值为 True。

示例

FLAGS_eager_delete_scope=True - 同步局域删除。

FLAGS_eager_delete_tensor_gb

(始于 1.0.0)

表示是否使用垃圾回收策略来优化网络的内存使用。如果 FLAGS_eager_delete_tensor_gb < 0, 则禁用垃圾回收策略。如果 FLAGS_eager_delete_tensor_gb >= 0, 则启用垃圾回收策略, 并在运行网络时回收内存垃圾, 这有利于节省内存使用量。它仅在您使用 Executor 运行程序、编译程序或使用并行数据编译程序时才有用。垃圾回收器直到垃圾的内存大小达到 FLAGS_eager_delete_tensor_gb GB 时才会释放内存垃圾。

取值范围

Double 型，单位为 GB，缺省值为 0.0。

示例

FLAGS_eager_delete_tensor_gb=0.0 - 垃圾占用大小达到 0.0GB 时释放内存垃圾，即一旦出现垃圾则马上释放。

FLAGS_eager_delete_tensor_gb=1.0 - 垃圾占用内存大小达到 1.0GB 时释放内存垃圾。

FLAGS_eager_delete_tensor_gb=-1.0 - 禁用垃圾回收策略。

注意

建议用户在训练大型网络时设置 FLAGS_eager_delete_tensor_gb=0.0 以启用垃圾回收策略。

FLAGS_fast_eager_deletion_mode

(始于 1.3)

是否使用快速垃圾回收策略。如果未设置，则在 CUDA 内核结束时释放 gpu 内存。否则 gpu 内存将在 CUDA 内核尚未结束的情况下被释放，从而使垃圾回收策略更快。仅在启用垃圾回收策略时有效。

取值范围

Bool 型，缺省值为 True。

示例

FLAGS_fast_eager_deletion_mode=True - 启用快速垃圾回收策略。

FLAGS_fast_eager_deletion_mode=False - 禁用快速垃圾回收策略。

FLAGS_fraction_of_cpu_memory_to_use

(始于 1.2.0)

表示分配的内存块占 CPU 总内存大小的比例。将来的内存使用将从该内存块分配。如果内存块没有足够的 cpu 内存，将从 cpu 请求分配与内存块相同大小的新的内存块，直到 cpu 没有足够的内存为止。

取值范围

Double 型，范围 [0, 1]，表示初始分配的内存块占 CPU 内存的比例。缺省值为 1.0。

示例

FLAGS_fraction_of_cpu_memory_to_use=0.1 - 分配总 CPU 内存大小的 10% 作为初始 CPU 内存块。

FLAGS_fraction_of_cuda_pinned_memory_to_use

(始于 1.2.0)

表示分配的 CUDA Pinned 内存块占 CPU 总内存大小的比例。将来的 CUDA Pinned 内存使用将从该内存块分配。如果内存块没有足够的 cpu 内存，将从 cpu 请求分配与内存块相同大小的新的内存块，直到 cpu 没有足够的内存为止。

取值范围

Double 型，范围 [0, 1]，表示初始分配的内存块占 CPU 内存的比例。缺省值为 0.5。

示例

FLAGS_fraction_of_cuda_pinned_memory_to_use=0.1 - 分配总 CPU 内存大小的 10% 作为初始 CUDA Pinned 内存块。

FLAGS_fraction_of_gpu_memory_to_use

(始于 1.2.0)

表示分配的显存块占 GPU 总可用显存大小的比例。将来的显存使用将从该显存块分配。如果显存块没有足够的 gpu 显存，将从 gpu 请求分配与显存块同样大小的新的显存块，直到 gpu 没有足够的显存为止。

取值范围

Double 型，范围 [0, 1]，表示初始分配的显存块占 GPU 可用显存的比例。

示例

FLAGS_fraction_of_gpu_memory_to_use=0.1 - 分配 GPU 总可用显存大小的 10% 作为初始 GPU 显存块。

注意

Windows 系列平台会将 FLAGS_fraction_of_gpu_memory_to_use 默认设为 0.5, Linux 则会默认设为 0.92。

FLAGS_fuse_parameter_groups_size

(始于 1.4.0)

FLAGS_fuse_parameter_groups_size 表示每一组中参数的个数。缺省值是一个经验性的结果。如果 fuse_parameter_groups_size 为 1，则表示组的大小和参数梯度的数目一致。如果 fuse_parameter_groups_size 为-1，则表示只有一个组。缺省值为 3，这只是一个经验值。

取值范围

Int32 型，缺省值为 3。

示例

FLAGS_fuse_parameter_groups_size=3 - 将单组参数的梯度大小设为 3。

FLAGS_fuse_parameter_memory_size

(始于 1.5.0)

FLAGS_fuse_parameter_memory_size 表示作为通信调用输入（例如 NCCLAllReduce）的单组参数梯度的上限内存大小。默认值为-1.0，表示不根据 memory_size 设置组。单位是 MB。

取值范围

Double 型，缺省值为-1.0。

示例

FLAGS_fuse_parameter_memory_size=16 - 将单组参数梯度的上限大小设为 16MB。

FLAGS_init_allocated_mem

(始于 0.15.0)

是否对分配的内存进行非零值初始化。该 flag 用于调试，以防止某些 Ops 假定已分配的内存都是初始化为零的。

取值范围

Bool 型，缺省值为 False。

示例

FLAGS_init_allocated_mem=True - 对分配的内存进行非零初始化。

FLAGS_init_allocated_mem=False - 不会对分配的内存进行非零初始化。

FLAGS_initial_cpu_memory_in_mb

(始于 0.14.0)

初始 PaddlePaddle 分配器的 CPU 内存块大小，单位为 MB。分配器将 FLAGS_initial_cpu_memory_in_mb 和 FLAGS_fraction_of_cpu_memory_to_use*（总物理内存）的最小值作为内存块大小。

取值范围

Uint64 型，缺省值为 500，单位为 MB。

示例

FLAGS_initial_cpu_memory_in_mb=100 - 在 FLAGS_fraction_of_cpu_memory_to_use*（总物理内存）大于 100MB 的情况下，首次提出分配请求时，分配器预先分配 100MB 内存，并在预分配的内存耗尽时再次分配 100MB。

FLAGS_initial_gpu_memory_in_mb

(始于 1.4.0)

预分配一块指定大小的 GPU 显存块。之后的显存使用将从该显存块分配。如果显存块没有足够的显存，将从 GPU 请求大小为 FLAGS_reallocate_gpu_memory_in_mb 的显存块，直到 GPU 没有剩余显存为止。

取值范围

Uint64 型，大于 0，为初始 GPU 显存大小，单位为 MB。

示例

FLAGS_initial_gpu_memory_in_mb=4096 - 分配 4GB 作为初始 GPU 显存块大小。

注意

如果设置该 flag，则 FLAGS_fraction_of_gpu_memory_to_use 设置的显存大小将被该 flag 覆盖。PaddlePaddle 将用该 flag 指定的值分配初始 GPU 显存。如果未设置该 flag，即 flag 默认值为 0 时，会关闭此显存策略。PaddlePaddle 会使用 FLAGS_fraction_of_gpu_memory_to_use 的策略来分配初始显存块。

FLAGS_memory_fraction_of_eager_deletion

(始于 1.4)

垃圾回收策略释放变量的内存大小百分比。如果 FLAGS_memory_fraction_of_eager_deletion = 1.0，则将释放网络中的所有临时变量。如果 FLAGS_memory_fraction_of_eager_deletion = 0.0，则不会释放网络中的任何临时变量。如果 $0.0 < \text{FLAGS_memory_fraction_of_eager_deletion} < 1.0$ ，则所有临时变量将根据其内存大小降序排序，并且仅释放具有最大内存大小的 FLAGS_memory_fraction_of_eager_deletion 比例的变量。该 flag 仅在运行并行数据编译程序时有效。

取值范围

Double 型，范围为 [0.0, 1.0]，缺省值为 1.0。

示例

FLAGS_memory_fraction_of_eager_deletion=0 - 保留所有临时变量，也就是禁用垃圾回收策略。

FLAGS_memory_fraction_of_eager_deletion=1 - 释放所有临时变量。

FLAGS_memory_fraction_of_eager_deletion=0.5 - 仅释放 50% 比例的占用内存最多的变量。

FLAGS_reallocate_gpu_memory_in_mb

(始于 1.4.0)

如果耗尽了分配的 GPU 显存块，则重新分配额外的 GPU 显存块。

取值范围

Int64 型，大于 0，为重新分配的显存大小，单位为 MB。

示例

FLAGS_reallocate_gpu_memory_in_mb=1024 - 如果耗尽了分配的 GPU 显存块，重新分配 1GB。

注意

如果设置了该 flag，则 FLAGS_fraction_of_gpu_memory_to_use 设置的显存大小将被该 flag 覆盖，PaddlePaddle 将用该 flag 指定的值重分配额外 GPU 显存。如果未设置该 flag，即 flag 默认值为 0 时，会关闭此显存策略。PaddlePaddle 会使用 FLAGS_fraction_of_gpu_memory_to_use 的策略来重新分配额外显存。

FLAGS_use_pinned_memory

(始于 0.12.0)

是否使用 pinned memory。设为 True 后，CPU 分配器将调用 mlock 来锁定内存页。

取值范围

Bool 型，缺省值为 True。

示例

FLAGS_use_pinned_memory=True - 锁定分配的 CPU 内存页面。

昇腾 NPU

FLAGS_npu_precision_mode

(develop)

FLAGS_npu_precision_mode 用于配置昇腾 NPU 芯片算子精度模式。仅在编译选项选择 ‘WITH_ASCEND_CL = ON’时有效。

取值范围

String 型，取值范围:[‘force_fp32’, ‘force_fp16’, ‘allow_fp32_to_fp16’, ‘must_keep_origin_dtype’, ‘allow_mix_precision’]。缺省值为”，此时运行精度模式为‘allow_fp32_to_fp16’。具体含义查看请 [点击这里](#)。

示例

FLAGS_npu_precision_mode=”allow_mix_precision” - 表示使用混合精度模式。

其他

FLAGS_benchmark

(始于 0.12.0)

用于基准测试。设置后，它将使局部删除同步，添加一些内存使用日志，并在内核启动后同步所有 cuda 内核。

取值范围

Bool 型，缺省值为 False。

示例

FLAGS_benchmark=True - 同步以测试基准。

FLAGS_inner_op_parallelism

(始于 1.3.0)

大多数 Operators 都在单线程模式下工作，但对于某些 Operators，使用多线程更合适。例如，优化稀疏梯度的优化 Op 使用多线程工作会更快。该 flag 用于设置 Op 内的线程数。

取值范围

Int32 型，缺省值为 0，这意味着 operator 将不会在多线程模式下运行。

示例

FLAGS_inner_op_parallelism=5 - 将 operator 内的线程数设为 5。

注意

目前只有稀疏的 adam op 支持 inner_op_parallelism。

FLAGS_max_body_size

(始于 1.0.0)

控制 BRPC 中的最大消息大小。

取值范围

Int32 型，缺省值为 2147483647。

示例

FLAGS_max_body_size=2147483647 - 将 BRPC 消息大小设为 2147483647。

FLAGS_sync_nccl_allreduce

(始于 1.3)

如果 `FLAGS_sync_nccl_allreduce` 为 `True`, 则会在 `allreduce_op_handle` 中调用 `cudaStreamSynchronize(nccl_stream)`, 这种模式在某些情况下可以获得更好的性能。

取值范围

Bool 型, 缺省值为 `True`。

示例

`FLAGS_sync_nccl_allreduce=True` - 在 `allreduce_op_handle` 中调用 `cudaStreamSynchronize(nccl_stream)`。

FLAGS_tracer_profile_fname

(始于 1.4.0)

`FLAGS_tracer_profile_fname` 表示由 `gperftools` 生成的命令式跟踪器的分析器文件名。仅在编译选项选择 ‘`WITH_PROFILER = ON`’时有效。如果禁用则设为 `empty`。

取值范围

String 型, 缺省值为 (“`gperf`”）。

示例

`FLAGS_tracer_profile_fname="gperf_profile_file"` - 将命令式跟踪器的分析器文件名设为”`gperf_profile_file`”。

Chapter 3

应用实践

如果你已经初步了解了 PaddlePaddle，期望可以针对实际问题建模、搭建自己网络，本模块提供了一些 PaddlePaddle 的具体典型案例：

快速上手：

- [hello paddle](#)：简单介绍 PaddlePaddle，完成你的第一个 PaddlePaddle 项目。
- [动态图](#)：介绍使用 PaddlePaddle 动态图。
- [高层 API 详细介绍](#)：详细介绍 PaddlePaddle 高层 API。
- [模型加载与保存](#)：介绍 PaddlePaddle 模型的加载与保存。
- [线性回归](#)：介绍使用 PaddlePaddle 实现线性回归任务。

计算机视觉：

- [图像分类](#)：介绍使用 PaddlePaddle 在 MNIST 数据集上完成图像分类。
 - [Duplicate explicit target name: ”图像分类”](#)。
- [图像分类](#)：介绍使用 PaddlePaddle 在 Cifar10 数据集上完成图像分类。
- [以图搜图](#)：介绍使用 PaddlePaddle 实现以图搜图。
 - [图像分割](#)：介绍使用 PaddlePaddle 实现 U-Net 模型完成图像分割。
 - [OCR](#)：介绍使用 PaddlePaddle 实现 OCR。
 - [图像超分](#)：介绍使用 PaddlePaddle 完成图像超分。
 - [人脸关键点检测](#)：介绍使用 PaddlePaddle 完成人脸关键点检测。
 - [点云分类](#)：介绍使用 PaddlePaddle 完成点云分类。

自然语言处理：

- [N-Gram](#)：介绍使用 PaddlePaddle 实现 N-Gram 模型。

- 文本分类：介绍使用 PaddlePaddle 在 IMDB 数据集上完成文本分类。
- 情感分类：介绍使用预训练词向量完成情感分类。
- 文本翻译：介绍使用 PaddlePaddle 实现文本翻译。
- 数字加法：介绍使用 PaddlePaddle 实现数字加法。

推荐：

- 电影推荐：介绍使用 PaddlePaddle 实现协同过滤完成电影推荐。

强化学习：

- 演员-评论家算法：介绍使用 PaddlePaddle 实现演员-评论家算法。
- 优势-演员-评论家算法 (A2C)：介绍使用 PaddlePaddle 实现 A2C 算法。
- 深度确定梯度策略 (DDPG)：介绍使用 PaddlePaddle 实现 DDPG 算法。

时间序列：

- 异常数据检测：介绍使用 PaddlePaddle 完成时序数据异常点检测。

动转静：

- 使用动转静完成以图搜图：介绍使用 PaddlePaddle 通过动转静完成以图搜图。

3.1 快速上手

这里提供了一些简单的案例，可以帮助你快速上手 PaddlePaddle：

- hello paddle：简单介绍 PaddlePaddle，完成你的第一个 PaddlePaddle 项目。
- 动态图：介绍使用 PaddlePaddle 动态图。
- 高层 API 详细介绍：详细介绍 PaddlePaddle 高层 API。
- 模型加载与保存：介绍 PaddlePaddle 模型的加载与保存。
- 线性回归：介绍使用 PaddlePaddle 实现线性回归任务。

3.2 计算机视觉

这里提供了一些计算机视觉的案例：

- 图像分类：介绍使用 PaddlePaddle 在 MNIST 数据集上完成图像分类。
- Duplicate explicit target name: ”图像分类”.

图像分类：介绍使用 PaddlePaddle 在 Cifar10 数据集上完成图像分类。

- 以图搜图：介绍使用 PaddlePaddle 实现以图搜图。

- 图像分割 : 介绍使用 PaddlePaddle 实现 U-Net 模型完成图像分割。
- OCR : 介绍使用 PaddlePaddle 实现 OCR。
- 图像超分 : 介绍使用 PaddlePaddle 完成图像超分。
- 人脸关键点检测 : 介绍使用 PaddlePaddle 完成人脸关键点检测。
- 点云分类 : 介绍使用 PaddlePaddle 完成点云分类。

3.3 自然语言处理

这里提供了一些自然语言处理的示例:

- N-Gram : 介绍使用 PaddlePaddle 实现 N-Gram 模型。
- 文本分类 : 介绍使用 PaddlePaddle 在 IMDB 数据集上完成文本分类。
- 情感分类 : 介绍使用预训练词向量完成情感分类。
- 文本翻译 : 介绍使用 PaddlePaddle 实现文本翻译。
- 数字加法 : 介绍使用 PaddlePaddle 实现数字加法。

3.4 推荐

这里提供了一篇推荐的示例:

- 电影推荐 : 介绍使用 PaddlePaddle 实现协同过滤完成电影推荐。

3.5 强化学习

这里提供了一些强化学习的示例:

- 演员-评论家算法 : 介绍使用 PaddlePaddle 实现演员-评论家算法。
- 深度确定梯度策略 (DDPG) : 介绍使用 PaddlePaddle 实现 DDPG 算法。

3.6 时序数据

这里提供了一篇时序数据的示例:

- 异常数据检测 : 介绍使用 PaddlePaddle 完成时序数据异常点检测。

3.7 动转静

这里提供了一篇动转静的示例：

- 使用动转静完成以图搜图：介绍使用 PaddlePaddle 通过动转静完成以图搜图。

Chapter 4

API 文档

欢迎使用飞桨框架 (PaddlePaddle), PaddlePaddle 是一个易用、高效、灵活、可扩展的深度学习框架，致力于让深度学习技术的创新与应用更简单。

在本版本中，飞桨框架对 API 做了许多优化，您可以参考下表来了解飞桨框架最新版的 API 目录结构与说明。更详细的说明，请参见 [版本说明](#)。此外，您可参考 PaddlePaddle 的 [GitHub](#) 了解详情。

注: **paddle.fluid.***, **paddle.dataset.*** 会在未来的版本中废弃，请您尽量不要使用这两个目录下的 API。

目录	功能和包含的 API
paddle.*	paddle 根目录下保留了常用 API 的别名，包括：paddle.tensor, paddle.framework, paddle.device 目录下的所有 API
paddle.tensor	Tensor 操作相关的 API，包括创建 zeros, 矩阵运算 matmul, 变换 concat, 计算 add, 查找 argmax 等
pad-dle.framework	框架通用 API 和动态图模式的 API，包括 no_grad、save、load 等。
paddle.device	设备管理相关 API，包括 set_device, get_device 等。
paddle.linalg	线性代数相关 API，包括 det, svd 等。
paddle.fft	快速傅里叶变换的相关 API，包括 fft, fft2 等。
paddle.amp	自动混合精度策略，包括 auto_cast、GradScaler 等。
paddle.autograd	自动求导相关 API，包括 backward、PyLayer 等。
paddle.callbacks	日志回调类，包括 ModelCheckpoint、ProgBarLogger 等。
pad-dle.distributed	分布式相关基础 API。
pad-dle.distributed.fleet	分布式相关高层 API。
paddle.hub	模型拓展相关的 API，包括 list、load、help 等。
paddle.io	数据输入输出相关 API，包括 Dataset, DataLoader 等。
paddle.jit	动态图转静态图相关 API，包括 to_static、ProgramTranslator、TracedLayer 等。
paddle.metric	评估指标计算相关的 API，包括 Accuracy, Auc 等。
paddle.nn	组网相关的 API，包括 Linear、卷积 Conv2D、循环神经网络 RNN、损失函数 CrossEntropyLoss、激活函数 ReLU 等。
paddle.onnx	paddle 转换为 onnx 协议相关 API，包括 export 等。
paddle.optimizer	优化算法相关 API，包括 SGD, Adagrad, Adam 等。
pad-dle.optimizer.lr	学习率衰减相关 API，包括 NoamDecay、StepDecay、PiecewiseDecay 等。
pad-dle.regularizer	正则化相关 API，包括 L1Decay、L2Decay 等。
paddle.static	静态图下基础框架相关 API，包括 Variable, Program, Executor 等
paddle.static.nn	静态图下组网专用 API，包括全连接层 fc、控制流 while_loop/cond 。
paddle.text	NLP 领域 API，包括 NLP 领域相关的数据集，如 Imdb、Movielens 。
paddle.utils	工具类相关 API，包括 CppExtension、CUDAExtension 等。
paddle.vision	视觉领域 API，包括数据集 Cifar10、数据处理 ColorJitter、常用基础网络结构 ResNet 等。
paddle.sparse	稀疏领域的 API。

Chapter 5

贡献指南

我们非常欢迎你参与到飞桨的建设中，以下内容致力于说明所有加入飞桨的方式，并尽量帮助你顺利地贡献飞桨。

同样的，如果你觉得本篇文档有缺失，或者是有描述不清楚的地方，我们也非常欢迎你一同贡献本系列文档。

- [概述](#)：贡献指南概述。
- [代码规范](#)：代码规范说明。
- [Git 操作指南](#)：Git 操作相关说明与 Paddle CI 手册。
- [编译安装](#)：如何从源码编译安装 Paddle。
- [API 开发指南](#)：API 开发相关说明。
- [Kernel Primitives API](#)：介绍 PaddlePaddle 为加快算子开发提供的 Block 级 CUDA 函数。
- [曙光开发指南](#)：曙光开发相关说明。
- [自定义新硬件接入指南](#)：介绍如何通过自定义硬件功能为飞桨接入新硬件后端。
- [文档贡献指南](#)：飞桨文档贡献指南。

5.1 概述

飞桨社区非常欢迎你加入到飞桨。你可以通过以下方式参与贡献：

- 新建一个 ISSUE 来反馈 bug
- 新建一个 ISSUE 来提出新功能需求
- 提 PR 来修复一个 bug
- 提 PR 来实现一个新功能
- 优化我们的文档

感谢你对飞桨开源项目的贡献！

5.2 代码规范

请参考以下规范，进行代码开发：

- C++: [Google C++ Style Guide](#)
- Python: [Google Python Style Guide](#)

5.3 Git 操作指南

以下将简单介绍 Git 操作指南以及 Paddle CI 手册，以帮助你快速参与到飞桨的开发中。

- [本地开发指南](#)：如何在本地进行 Paddle 开发。
- [提交 PR 注意事项](#)：提交 PR 相关的注意事项。
- [Code Review 约定](#)：飞桨框架对于 Code Review 的一些约定介绍。
- [Paddle CI 手册](#)：Paddle CI 介绍说明。

5.3.1 本地开发指南

本文将指导你如何在本地进行代码开发

代码要求

- 代码注释请遵守 Doxygen 的样式。
- 所有代码必须具有单元测试。
- 通过所有单元测试。
- 请遵守提交代码的一些约定。

以下教程将指导你提交代码。

Fork

跳转到PaddlePaddle GitHub 首页，然后单击 Fork 按钮，生成自己目录下的仓库，比如 <https://github.com/USERNAME/Paddle>。

克隆 (Clone)

将远程仓库 clone 到本地：

```
git clone https://github.com/USERNAME/Paddle  
cd Paddle
```

创建本地分支

Paddle 目前使用Git 流分支模型进行开发，测试，发行和维护，具体请参考 [Paddle 分支规范](#)。

所有的 feature 和 bug fix 的开发工作都应该在一个新的分支上完成，一般从 develop 分支上创建新分支。

使用 git checkout -b 创建并切换到新分支。

```
git checkout -b my-cool-stuff
```

值得注意的是，在 checkout 之前，需要保持当前分支目录 clean，否则会把 untracked 的文件也带到新分支上，这可以通过 git status 查看。

使用 pre-commit 钩子

Paddle 开发人员使用 [pre-commit](#) 工具来管理 Git 预提交钩子。它可以帮助我们格式化源代码 (C++, Python)，在提交 (commit) 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)。

pre-commit 测试是 CI 中单元测试的一部分，不满足钩子的 PR 不能被提交到 Paddle，Paddle 使用的 pre-commit 是 1.10.4 版本。首先安装并在当前目录运行它：

```
pip install pre-commit==1.10.4  
pre-commit install
```

Paddle 使用 clang-format 来调整 C/C++ 源代码格式，请确保 clang-format 版本是 3.8。

注：通过 pip install pre-commit 和 conda install -c conda-forge pre-commit 安装的 yapf 稍有不同的，Paddle 开发人员使用的是 pip install pre-commit。

开始开发

在本例中，我删除了 README.md 中的一行，并创建了一个新文件。

通过 git status 查看当前状态，这会提示当前目录的一些变化，同时也可以通过 git diff 查看文件具体被修改的内容。

```
git status
On branch test
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    test

no changes added to commit (use "git add" and/or "git commit -a")
```

编译

关于编译 PaddlePaddle 的源码，请参见[从源码编译](#)选择对应的操作系统。

单测

python/paddle/fluid/tests/unittests/ 目录下新增的 test_*.py 单元测试会被自动加入工程进行编译。

注意事项：

- 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。
- 执行单测一定要用 **ctest** 命令，不可直接 python test_*.py。

参考上述编译过程，编译成功后，在 build 目录下执行下面的命令来运行单元测试：

执行：

```
ctest -R test_mul_op -V
```

提交 (commit)

接下来我们取消对 README.md 文件的改变，然后提交新添加的 test 文件。

```
git checkout -- README.md
git status
On branch test
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    test

nothing added to commit but untracked files present (use "git add" to track)
git add test
```

Git 每次提交代码，都需要写提交说明，这可以让其他人知道这次提交做了哪些改变，这可以通过 git commit 完成。

```
git commit
CRLF end-lines remover..... (no files to check) Skipped
yapf..... (no files to check) Skipped
Check for added large files..... Passed
Check for merge conflicts..... Passed
Check for broken symlinks..... Passed
Detect Private Key..... (no files to check) Skipped
Fix End of Files..... (no files to check) Skipped
clang-formater..... (no files to check) Skipped
[my-cool-stuff c703c041] add test file
  1 file changed, 0 insertions(+), 0 deletions(-)
  create mode 100644 233
```

保持本地仓库最新

在准备发起 Pull Request 之前，需要同步原仓库 (<https://github.com/PaddlePaddle/Paddle>) 最新的代码。

首先通过 git remote 查看当前远程仓库的名字。

```
git remote
origin
git remote -v
origin      https://github.com/USERNAME/Paddle (fetch)
origin      https://github.com/USERNAME/Paddle (push)
```

这里 origin 是我们 clone 的远程仓库的名字，也就是自己用户名下的 Paddle，接下来我们创建一个原始 Paddle 仓库的远程主机，命名为 upstream。

```
② git remote add upstream https://github.com/PaddlePaddle/Paddle
② git remote
origin
upstream
```

获取 upstream 的最新代码并更新当前分支。

```
② git fetch upstream
② git pull upstream develop
```

Push 到远程仓库

将本地的修改推送到 GitHub 上，也就是 <https://github.com/USERNAME/Paddle>。

```
# 推送到远程仓库 origin 的 my-cool-stuff 分支上
② git push origin my-cool-stuff
```

5.3.2 提交 PR 注意事项

提交 Pull Request

- 请注意 commit 的数量：

原因：如果仅仅修改一个文件但提交了十几个 commit，每个 commit 只做了少量的修改，这会给评审人带来很大困扰。评审人需要逐一查看每个 commit 才能知道做了哪些修改，且不排除 commit 之间的修改存在相互覆盖的情况。

建议：每次提交时，保持尽量少的 commit。可以通过 `git rebase -i HEAD~3` 将最新的 3 个 commit 合并成一个（你可以根据实际情况修改该数值），再 Push 到远程仓库，可以参考 [rebase 用法](#)。

- 请注意每个 commit 的名称：应能反映当前 commit 的内容，不能太随意。
- 请不要频繁 Merge develop 分支（在过 CI 时，会自动 Merge develop），这样会使 CI 重跑，更加延长 CI 通过时间。
- 评审人 review 过后，不允许使用 `git push -f` 强行提交代码，这样评审人无法看到修改前后的 diff，使评审变得困难。

完成 Pull Request PR 创建

切换到所建分支，然后点击 Compare & pull request。



选择目标分支：



如果解决了某个 Issue 的问题，请在该 Pull Request 的第一个评论框中加上：fix #issue_number，这样当该 Pull Request 被合并后，会自动关闭对应的 Issue。关键词包括：close, closes, closed, fix, fixes, fixed, resolve, resolves, resolved，请选择合适的词汇。详细可参考[Closing issues via commit messages](#)

接下来等待 review，如果有需要修改的地方，参照上述步骤更新 origin 中的对应分支即可。

签署 CLA 协议和通过单元测试

签署 CLA

在首次向 PaddlePaddle 提交 Pull Request 时，你需要签署一次 CLA(Contributor License Agreement) 协议，以保证你的代码可以被合入，具体签署方式如下：

- 请你查看 PR 中的 Check 部分，找到 license/cla，并点击右侧 detail，进入 CLA 网站
- 请你点击 CLA 网站中的“Sign in with GitHub to agree”，点击完成后将会跳转回你的 Pull Request 页面

通过单元测试

你在 Pull Request 中每提交一次新的 commit 后，会触发 CI 单元测试，请确认你的 commit message 中已加入必要的说明，请见[提交 \(commit\)](#)

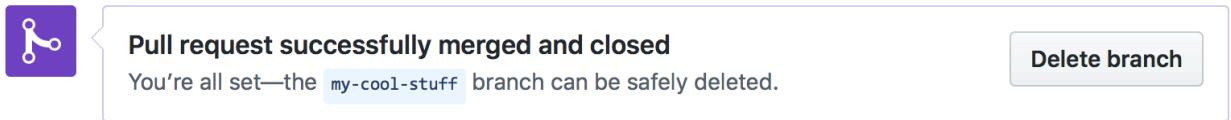
请你关注你 Pull Request 中的 CI 单元测试进程，它将会在几个小时内完成

当所需的测试后都出现了绿色的对勾，表示你本次 commit 通过了各项单元测试，你只需要关注显示 Required 任务，不显示的可能是我们正在测试的任务

如果所需的测试后出现了红色叉号，代表你本次的 commit 未通过某项单元测试，在这种情况下，请你点击 detail 查看报错详情，优先自行解决报错问题，无法解决的情况，以评论的方式添加到评论区中，我们的工作人员将和你一起查看

删除远程分支

在 PR 被 merge 进主仓库后，我们可以在 PR 的页面删除远程仓库的分支。



也可以使用 `git push origin : 分支名` 删除远程分支，如：

```
git push origin :my-cool-stuff
```

删除本地分支

最后，删除本地分支。

```
# 切换到 develop 分支
git checkout develop

# 删除 my-cool-stuff 分支
git branch -D my-cool-stuff
```

至此，我们就完成了一次代码贡献的过程。

5.3.3 Code Review 约定

为了使评审人在评审代码时更好地专注于代码本身，请你每次提交代码时，遵守以下约定：

- 1) 请保证 CI 中测试任务能顺利通过。如果没过，说明提交的代码存在问题，评审人一般不做评审。
- 2) 如果解决了某个 Issue 的问题，请在该 Pull Request 的第一个评论框中加上：`fix #issue_number`，这样当该 Pull Request 被合并后，会自动关闭对应的 Issue。关键词包括：`close, closes, closed, fix, fixes, fixed, resolve, resolves, resolved`，请选择合适的词汇。详细可参考[Closing issues via commit messages](#)。

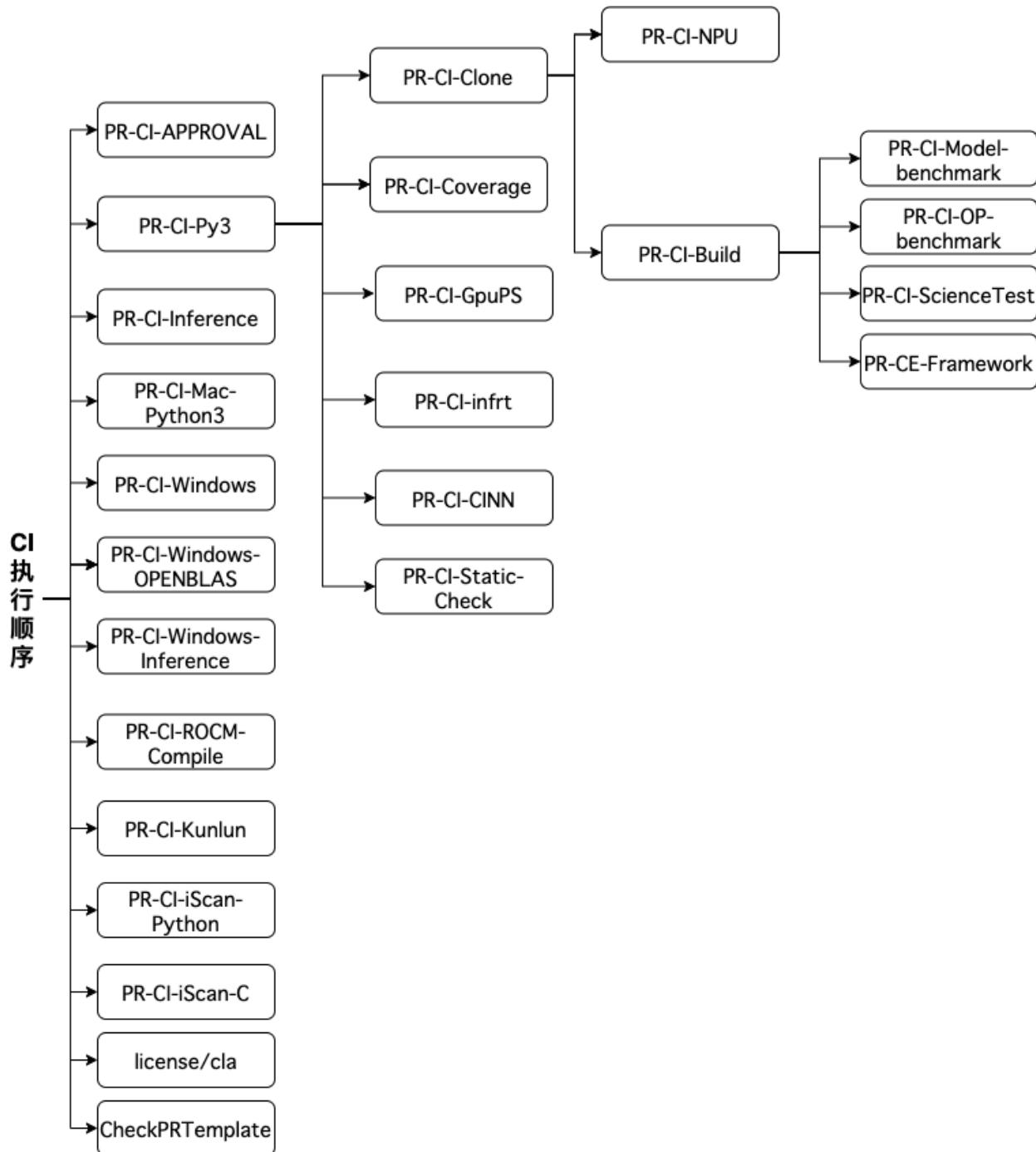
此外，在回复评审人意见时，请你遵守以下约定：

- 1) 评审人的每个意见都必须回复（这是开源社区的基本礼貌，别人帮了忙，应该说谢谢）：
 - 对评审意见同意且按其修改完的，给个简单的 `Done` 即可；
 - 对评审意见不同意的，请给出你自己的反驳理由。
- 2) 如果评审意见比较多：
 - 请给出总体的修改情况。
 - 请采用[start a review](#)进行回复，而非直接回复的方式。原因是每个回复都会发送一封邮件，会造成邮件灾难

5.3.4 Paddle CI 手册

整体介绍

当你提交一个 PR (Pull_Request)，你的 PR 需要经过一些 CI (Continuous Integration)，以触发 develop 分支的为例为你展示 CI 执行的顺序：



如上图所示，提交一个 PR，你需要：

- 签署 CLA 协议
- PR 描述需要符合规范
- 通过不同平台（Linux/Mac/Windows/XPU/NPU 等）的编译与单测
- 通过静态代码扫描工具的检测

需要注意的是：如果你的 PR 只修改文档部分，你可以在 commit 中添加说明（commit message）以只触发文档相关的 CI，写法如下：

```
# PR仅修改文档等内容，只触发PR-CI-Static-Check
git commit -m 'test=document_fix'
```

各流水线介绍

下面以触发 develop 分支为例，分平台对每条 CI 进行简单介绍。

CLA

贡献者许可证协议Contributor License Agreements是指当你要给 Paddle 贡献代码的时候，需要签署的一个协议。如果不签署那么你贡献给 Paddle 项目的修改，即 PR 会被 Github 标志为不可被接受，签署了之后，这个 PR 就是可以在 review 之后被接受了。

CheckPRTemplate

检查 PR 描述信息是否按照模板填写。

- 通常 10 秒内检查完成，如遇长时间未更新状态，请 re-edit 一下 PR 描述重新触发该 CI。

```
### PR types
<!-- One of [ New features | Bug fixes | Function optimization | Performance
-->
(必填) 从上述选项中，选择并填写PR类型

### PR changes
<!-- One of [ OPs | APIs | Docs | Others ] -->
(必填) 从上述选项中，选择并填写PR所修改的内容

### Describe
<!-- Describe what this PR does -->
(必填) 请填写PR的具体修改内容
```

Linux 平台

PR-CI-Clone

该 CI 主要是将当前 PR 的代码从 GitHub clone 到 CI 机器，方便后续的 CI 直接使用。

PR-CI-APPROVAL

该 CI 主要的功能是检测 PR 中的修改是否通过了审批。在其他 CI 通过之前，你可以无需过多关注该 CI，其他 CI 通过后会有相关人员进行 review 你的 PR。

- 执行脚本: paddle/scripts/paddle_build.sh assert_file_approvals

PR-CI-Build

该 CI 主要是编译出当前 PR 的编译产物，并且将编译产物上传到 BOS（百度智能云对象存储）中，方便后续的 CI 可以直接复用该编译产物。

- 执行脚本: paddle/scripts/paddle_build.sh build_pr_dev

PR-CI-Py3

该 CI 主要的功能是为了检测当前 PR 在 CPU、Python3 版本的编译与单测是否通过。

- 执行脚本: paddle/scripts/paddle_build.sh cicheck_py37

PR-CI-Coverage

该 CI 主要的功能是检测当前 PR 在 GPU、Python3 版本的编译与单测是否通过，同时增量代码需满足行覆盖率大于 90% 的要求。

- 编译脚本: paddle/scripts/paddle_build.sh cpu_cicheck_coverage
- 测试脚本: paddle/scripts/paddle_build.sh gpu_cicheck_coverage

PR-CE-Framework

该 CI 主要是为了测试 P0 级框架 API 与预测 API 的功能是否通过。此 CI 使用 PR-CI-Build 的编译产物，无需单独编译。

- 框架 API 测试脚本 (PaddlePaddle/PaddleTest): PaddleTest/framework/api/run_paddle_ci.sh
- 预测 API 测试脚本 (PaddlePaddle/PaddleTest) : PaddleTest/inference/python_api_test/parallel_run.sh

PR-CI-Science Test

该 CI 主要是为了科学计算相关的单测是否通过。此 CI 使用 PR-CI-Build 的编译产物，无需单独编译。

- 测试脚本 (PaddlePaddle/PaddleScience): PaddleScience/tests/test_examples/run.sh

PR-CI-OP-benchmark

该 CI 主要的功能是 PR 中的修改是否会造成 OP 性能下降或者精度错误。此 CI 使用 PR-CI-Build 的编译产物，无需单独编译。

- 执行脚本: tools/ci_op_benchmark.sh run_op_benchmark

关于 CI 失败解决方案等详细信息可查阅PR-CI-OP-benchmark Manual

PR-CI-Model-benchmark

该 CI 主要的功能是检测 PR 中的修改是否会导致模型性能下降或者运行报错。此 CI 使用 PR-CI-Build 的编译产物，无需单独编译。

- 执行脚本: tools/ci_model_benchmark.sh run_all

关于 CI 失败解决方案等详细信息可查阅PR-CI-Model-benchmark Manual

PR-CI-Static-Check

该 CI 主要的功能是检查代码风格是否符合规范，检测 develop 分支与当前 PR 分支的增量的 API 英文文档是否符合规范，以及当变更 API 或 OP 时需要 TPM approval。

- 编译脚本: paddle/scripts/paddle_build.sh build_and_check_cpu
- 示例文档检测脚本: paddle/scripts/paddle_build.sh build_and_check_gpu

PR-CI-infrt

该 CI 主要是为了检测 infrt 是否编译与单测通过

- 编译脚本: paddle/scripts/infrt_build.sh build_only
- 测试脚本: paddle/scripts/infrt_build.sh test_only

PR-CI-CINN

该 CI 主要是为了编译含 CINN 的 Paddle，并运行 Paddle-CINN 对接的单测，保证训练框架进行 CINN 相关开发的正确性。

- 编译脚本: paddle/scripts/paddle_build.sh build_only
- 测试脚本: paddle/scripts/paddle_build.sh test

PR-CI-Inference

该 CI 主要的功能是为了检测当前 PR 对 C++ 预测库与训练库的编译和单测是否通过。

- 编译脚本: paddle/scripts/paddle_build.sh build_inference
- 测试脚本: paddle/scripts/paddle_build.sh gpu_inference

PR-CI-GpuPS

该 CI 主要是为了保证 GPUBOX 相关代码合入后编译可以通过。

- 编译脚本: paddle/scripts/paddle_build.sh build_gpubox

MAC

PR-CI-Mac-Python3

该 CI 是为了检测当前 PR 在 MAC 系统下 python35 版本的编译与单测是否通过，以及做 develop 与当前 PR 的单测增量检测，如有不同，提示需要 approval。

- 执行脚本: paddle/scripts/paddle_build.sh maccheck_py35

Windows

PR-CI-Windows

该 CI 是为了检测当前 PR 在 Windows 系统下 MKL 版本的 GPU 编译与单测是否通过，以及做 develop 与当前 PR 的单测增量检测，如有不同，提示需要 approval。

- 执行脚本: paddle/scripts/paddle_build.bat wincheck_mkl

PR-CI-Windows-OPENBLAS

该 CI 是为了检测当前 PR 在 Windows 系统下 OPENBLAS 版本的 CPU 编译与单测是否通过。

- 执行脚本: `paddle/scripts/paddle_build.bat wincheck_openblas`

PR-CI-Windows-Inference

该 CI 是为了检测当前 PR 在 Windows 系统下预测模块的编译与单测是否通过。

- 执行脚本: `paddle/scripts/paddle_build.bat wincheck_inference`

XPU 机器

PR-CI-Kunlun

该 CI 主要的功能是检测 PR 中的修改能否在昆仑芯片上编译与单测通过。

- 执行脚本: `paddle/scripts/paddle_build.sh check_xpu_coverage`

NPU 机器

PR-CI-NPU

该 CI 主要是为了检测当前 PR 对 NPU 代码编译跟测试是否通过。

- 编译脚本: `paddle/scripts/paddle_build.sh build_only`
- 测试脚本: `paddle/scripts/paddle_build.sh gpu_cicheck_py35`

Sugon-DCU 机器

PR-CI-ROCM-Compile

该 CI 主要的功能是检测 PR 中的修改能否在曙光芯片上编译通过。

- 执行脚本: `paddle/scripts/musl_build/build_paddle.sh build_only`

静态代码扫描

PR-CI-iScan-C

该 CI 是为了检测当前 PR 的 C++ 代码是否可以通过静态代码扫描。

PR-CI-iScan- Python

该 CI 是为了检测当前 PR 的 Python 代码是否可以通过静态代码扫描。

CI 失败如何处理

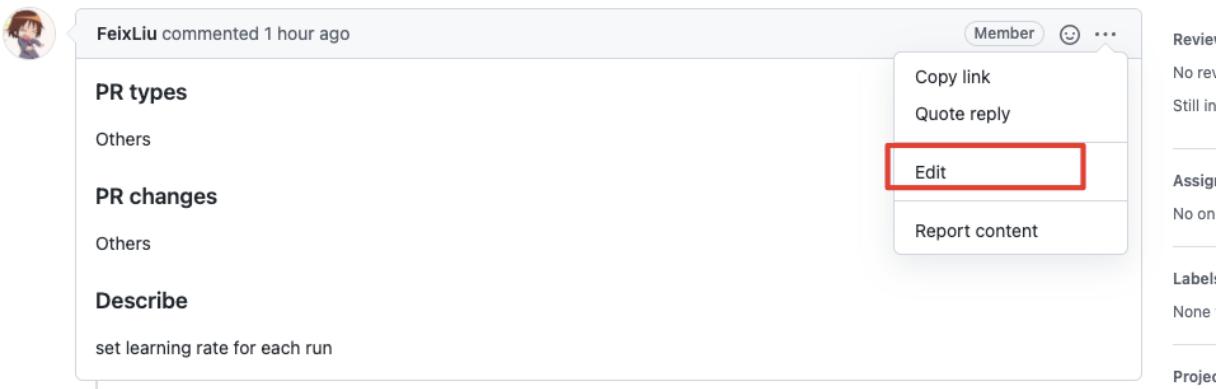
CLA 失败

- 如果你的 cla 一直是 pending 状态, 那么需要等其他 CI 都通过后, 点击 Close pull request , 再点击 Reopen pull request , 并等待几分钟 (建立在你已经签署 cla 协议的前提下); 如果上述操作重复 2 次仍未生效, 请重新提一个 PR 或评论区留言。
- 如果你的 cla 是失败状态, 可能原因是你提交 PR 的账号并非你签署 cla 协议的账号, 如下图所示:
- 建议你在提交 PR 前设置:

```
git config -l local user.email 你的邮箱
git config -l local user.name 你的名字
```

CheckPRTemplate 失败

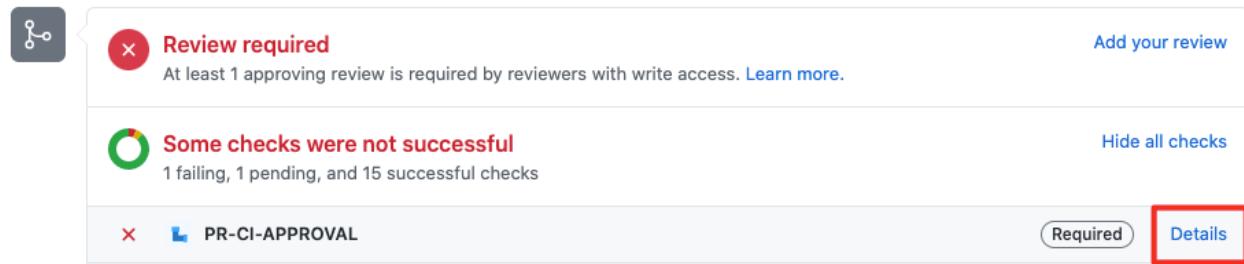
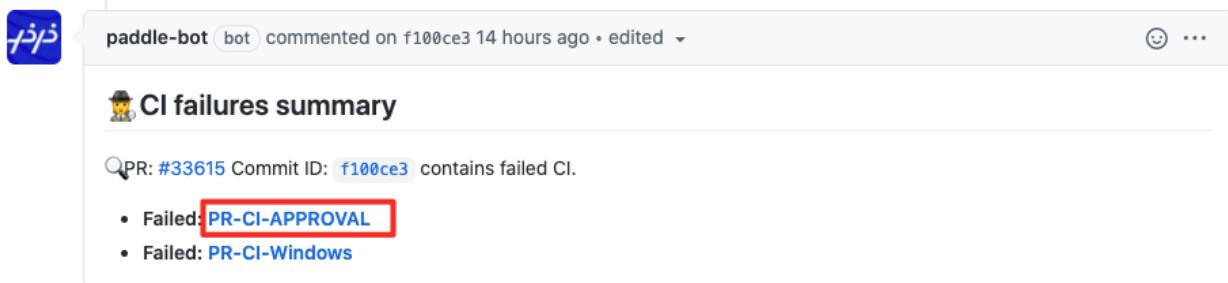
如果你的 CheckPRTemplate 状态一直未变化, 这是由于通信原因状态未返回到 GitHub。你只需要重新编辑一下 PR 描述保存后就可以重新触发该条 CI, 步骤如下:





其他 CI 失败

当你的 PR 的 CI 失败时, paddle-bot 会在你的 PR 页面发出一条评论, 同时此评论 GitHub 会同步到你的邮箱, 让你第一时间感知到 PR 的状态变化 (注意: 只有第一条 CI 失败的时候会发邮件, 之后失败的 CI 只会更新 PR 页面的评论。)



你可以通过点击 paddle-bot 评论中的 CI 名字, 也可通过点击 CI 列表中的 Details 来查看 CI 的运行日志, 如上图。通常运行日志的末尾会告诉你 CI 失败的原因。

由于网络代理、机器不稳定等原因, 有时候 CI 的失败也并不是你的 PR 自身的原因, 这时候你只需要 rerun 此 CI 即可 (你需要将你的 GitHub 授权于效率云 CI 平台)。



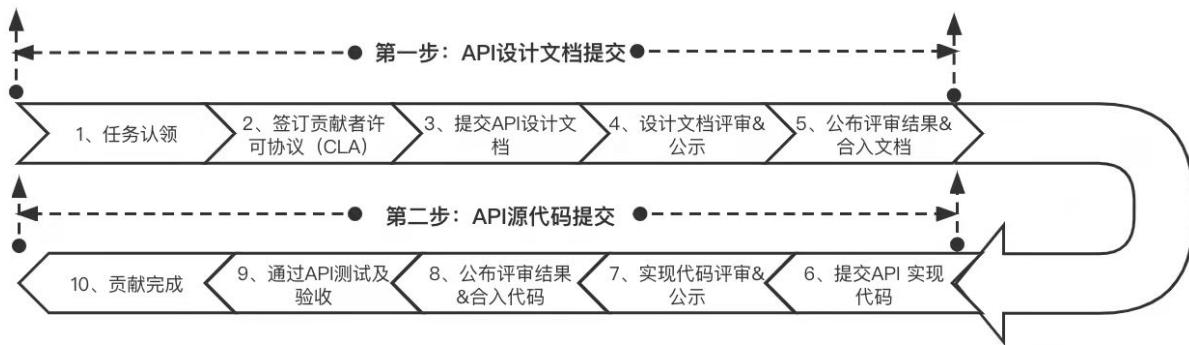
如果 CI 失败你无法判断原因, 请联系 @lelelelelez。

若遇到其他问题, 请联系 @lelelelelez。

5.4 新增 API 开发 & 提交流程

飞桨作为一个开源项目, 我们鼓励生态开发者为 paddlepaddle 贡献 API, 当你想要为飞桨开发新 API 功能时, 请遵守此 API 贡献流程在 Github 上完成文档设计和代码设计并提交至相应的 github 仓库。

5.4.1 API 贡献流程如下



5.4.2 流程介绍

1、任务认领

如果你想参与飞桨 API 开源贡献, 可以在 Github paddle 项目上的 issue 区域进行任务认领, 飞桨官网会发布一些新增 API 的任务, 用户可以认领飞桨发布的任务, 也可以产出自己的新增 API 想法, 并按照此贡献流程提交设计文档。

2、签订贡献者许可协议 (CLA)

对于你贡献的源代码, 你将拥有合法的知识产权, 为了保护你的权益, 你需要先签署一份 贡献者许可协议 。

注意: 当你签署完 CLA 后, 我们才会继续对您提交的设计方案和实现代码进行评审及合入

3、提交 API 设计文档

API 设计文档的目的是为了社区开发者更容易的参与开源项目共建, 开发者通过与飞桨专家和社区其他用户进行广泛的交流, 完善设计方案和 pr 请求, 在后续提交实现代码之前确保 API 设计方案与飞桨设计理念一致, 也让后续代码评审及代码合入变得更加容易。

当你想要发起一个新增 API 的贡献时, 你需要先对 API 进行开发设计, 并提交一份 API 设计文档。设计时请遵守飞桨 API 设计及命名规范。同时, 飞桨为大家提供了 API 设计文档撰写模版和 API 设计文档示例。完成后, 你需要将设计文档提交至 Github 开发者社区仓库, 并根据本地开发指南提交 PR。

此过程请参考相应的开发规范，并提交以下内容：

提交内容	参考文档	提交位置
1、API 设计文档	<ul style="list-style-type: none"> • API 设计及命名规范 • API 设计文档模版 • API 设计文档示例 	Github 开发者社区仓库

同时，飞桨为大家提供了 [API 设计文档模版](#) 和 [API 设计文档 demo](#)，你可以使用这份模版撰写 API 设计文档。完成后，你需要将设计文档提交至 [Github 开发者社区仓库](#)，并根据 [本地开发指南](#) 提交 PR。

4、设计文档评审 & 公示

飞桨专家对你提交的 API 设计文档进行审核，同时此文档也将接受来自开发者社区的评估，大家可以在 pr 评论区进行广泛的交流。开发者根据飞桨专家和其他开发者的反馈意见进行讨论并做出修改，最终评审通过后会在开源社区中同步。

如果你的 API 功能比较复杂，我们可能会在社区中针对 API 设计文档发起评审会议，会提前在 pr 评论区公布会议时间、会议地址、参会人、议题等内容，请及时关注 pr 中最新动态，你也可以在评论区自行申请评审会。会议结束后，我们会在 pr 中发出会议结论。

5、公布评审结果 & 合入文档

当设计文档评审 & 公示通过后，你的 API 设计文档将会合入至 [飞桨开发者社区仓库](#)，并在开源社区中同步。

6、提交 API 实现代码

当 API 设计文档合入后，开发者根据评审通过的 API 设计内容进行代码开发。此过程请参考相应的开发规范，并提交以下内容：

提交内容	参考文档	提交位置
1、API 实现代码	<ul style="list-style-type: none"> • API 设计及命名规范 • Python API 开发指南（请期待） • C++ API 开发指南 	Github 飞桨训练框架仓库
2、API 英文文档	<ul style="list-style-type: none"> • API 文档书写规范 	Github 飞桨训练框架仓库
3、API 中文文档	<ul style="list-style-type: none"> • API 文档书写规范 	Github 飞桨文档仓库
4、API 单测代码	<ul style="list-style-type: none"> • API 验收标准 	Github 飞桨训练框架仓库

当开发者完成以上代码设计后，需要将代码提交至 [Github 飞桨训练框架仓库](#)，并根据 [本地开发指南](#) 提交

PR、准备接受社区的评审。

7、实现代码评审 & 公示

飞桨官方会及时安排专家进行 API 代码审核，代码也将接受来自开发者社区的评审，开发者可以在 pr 评论区进行广泛的交流，开发者对飞桨专家和其他开发者的反馈意见进行讨论并做出修改，最终评审通过后会在开源社区中同步。

如果你的 API 功能比较复杂，官方可能会在社区中针对 API 实现代码发起评审会议，会提前在 pr 评论区公布会议时间、会议地址、参会人、议题等内容，请及时关注 pr 中最新动态，你也可以在评论区自行申请评审会。会议结束后，我们会在 pr 中发出会议结论。

8、公布评审结果 & 合入代码

当设计文档评审 & 公示通过后，官方会在开源社区中同步，你的 API 实现代码将会合入至 [Github 飞桨训练框架仓库](#)。

9、通过 API 测试及验收

当你的代码合入 [Github 飞桨训练框架仓库](#) 后，官方会对你的代码进行集成测试，并通知你测试结果。如果测试通过，恭喜你贡献流程已经全部完成；如果测试不通过，我们会联系你进行代码修复，请及时关注 [github](#) 上的最新动态；

注意：代码合入 develop 分之后的第二天你可以从官网下载 develop 编译的安装包体验此功能。飞桨后续也会将此功能纳入正式版的发版计划～

10、贡献完成

感谢您的贡献！

飞桨 API 的设计和命名规范

API 设计规范

总体原则

1. 单一职责，每个 API 应只完成单一的任务
2. 接口设计应考虑通用性，避免只适用于某些单一场景
3. 符合行业标准，综合参考开源深度学习框架的接口设计，借鉴各框架的优点；除非飞桨 API 设计有明确优势的情况下可保留自有特色，否则需要符合行业标准
4. 功能类似的接口，参数名和行为需要保持一致，比如，lstm 和 gru
5. 优先保证清晰，然后考虑简洁，避免使用不容易理解的缩写
6. 历史一致，如无必要原因，应避免接口修改
7. 动静统一，如无特殊原因，动态图和静态图下的 API 输入、输出要求一致。开发者使用相同的代码，均可以在动态图和静态图模式下执行

动态图与静态图模式

关于飞桨框架支持的开发模式，为了便于用户理解，代码和 API 均采用“动态图”和“静态图”的说法；文档优先使用“动态图”和“静态图”的说法，不推荐使用“命令式编程”和“声明式编程”的说法。

API 目录结构规范

- 公开 API 代码应该放置到以下列出的对应位置目录/文件中，并添加到目录下 `__init__.py` 的 `all` 列表中。非公开 API 不能添加到 `all` 列表中
- 常用的 API 可以在更高层级建立别名，当前规则如下：
 - `paddle.tensor` 目录下的 API，均在 `paddle` 根目录建立别名，其他所有 API 在 `paddle` 根目录下均没有别名。
 - `paddle.nn` 目录下除了 `functional` 目录以外的所有 API，在 `paddle.nn` 目录下均有别名。
 - 一些特殊情况比如特别常用的 API 会直接在 `paddle` 下建立别名

```
paddle.nn.functional.mse_loss # functional下的函数不建立别名，使用完整名称
paddle.nn.Conv2D # 为 paddle.nn.layer.conv.Conv2D建立的别名
```

3. 一些特殊情况比如特别常用的 API 会直接在 `paddle` 下建立别名

```
paddle.tanh # 为常用函数paddle.tensor.math.tanh建立的别名
paddle.linspace# 为常用函数paddle.fluid.layers.linspace建立的别名
```

API 行为定义规范

- 动静统一要求。除了 `paddle.static` 目录中的 API 外，其他目录的所有 API 原则上均需要支持动态图和静态图模式下的执行，且输入、输出要求一致。开发者使用相同的代码，可以在动态图和静态图模式下执行。

```
#静态图专用
paddle.fluid.gradients(targets, inputs, target_gradients=None, no_grad_set=None)
#动态图专用
paddle.fluid.dygraph.grad(outputs, inputs, grad_outputs=None, retain_graph=None,
    ↪create_graph=False, only_inputs=True, allow_unused=False, no_grad_vars=None,
    ↪backward_strategy=None)
#动、静态图通用
paddle.nn.functional.conv2d(x, weight, bias=None, padding=0, stride=1, dilation=1,
    ↪ groups=1, use_cudnn=True, act=None, data_format="NCHW", name=None)
paddle.nn.Conv2D(num_channels, num_filters, filter_size, padding=0, stride=1,
    ↪ dilation=1, groups=1, param_attr=None, bias_attr=None, use_cudnn=True, act=None,
    ↪ data_format="NCHW", dtype='float32')
```

- API 不需要用户指定执行硬件，框架可以自动根据当前配置，选择执行的库。

- 设置缺省参数类型组网类 API 去除 dtype 参数，比如 Linear, Conv2d 等，通过使用 paddle.set_default_dtype 和 paddle.get_default_dtype 设置全局的数据类型。
- 数据类型转换规则

1. 不支持 Tensor 和 Tensor 之间的隐式数据类型转换，隐藏类型转换虽然方便，但风险很高，很容易出现转换错误。如果发现类型不匹配，进行隐式类型转换，一旦转换造成精度损失，会导致模型的精度降低，由于没有任何提示，问题非常难以追查；而如果直接向用户报错或者警告，用户确认后，修改起来会很容易。避免了出错的风险。

```
import paddle
a = paddle.randn([3, 1, 2, 2], dtype='float32')
b = paddle.randint(0, 10, [3, 1, 2, 2], dtype='int32')
c = a + b
# 执行后会出现以下类型不匹配警告：
# ..... \paddle\fluid\dygraph\math_op_patch.py:239: UserWarning: The dtype of
# ↵left and right variables are not the same, left dtype is paddle.float32, ↵
# ↵but right dtype is paddle.int32, the right dtype will convert to paddle.
# ↵float32 format(lhs_dtype, rhs_dtype, lhs_dtype))
```

2. 支持 Tensor 和 python Scalar 之间的隐式类型转换，当 Tensor 的数据类型和 python Scalar 是同一类的数据类型时（都是整型，或者都是浮点型），或者 Tensor 是浮点型而 python Scalar 是整型的，默认会将 python Scalar 转换成 Tensor 的数据类型。而如果 Tensor 的数据类型是整型而 python Scalar 是浮点型时，计算结果会是 float32 类型的。

```
import paddle
a = paddle.to_tensor([1.0], dtype='float32')
b = a + 1 # 由于python scalar默认采用int64，转换后b的类型为'float32'
c = a + 1.0 # 虽然python scalar是float64，但计算结果c的类型为'float32'
a = paddle.to_tensor([1], dtype='int32')
b = a + 1.0 # 虽然python scalar是float64，但计算结果b的类型为'float32'
c = a + 1 # 虽然python scalar是int64，但计算结果c的类型为'int32'
```

数据类型规范

- 参数数据类型

对于 loss 类 API，比如 cross_entropy, bce_loss 等，输入的 label 需要支持 [int32, int64, float32, float64] 数据类型。

- 返回值数据类型

实现需要返回下标 indices 的 API，比如 argmax、argmin、argsort、topk、unique 等接口时，需要提供 dtype 参数，用于控制返回值类型是 int32 或者 int64，默认使用 dtype='int64'（与 numpy, tf, pytorch 保持一致），主要目的是当用户在明确输入数据不超过 int32 表示范围时，可以手动设置 dtype='int32' 来减少显存的

占用；对于 `dtype='int32'` 设置，需要对输入数据的下标进行检查，如果超过 `int32` 表示范围，通过报错提示用户使用 `int64` 数据类型。

API 命名规范

API 的命名应使用准确的深度学习相关英文术语，具体参考附录的中英术语表。

类名与方法名的规范

- 类名的命名应采用驼峰命名法，通常采用名词形式

```
paddle.nn.Conv2D  
paddle.nn.BatchNorm  
paddle.nn.Embedding  
paddle.nn.LogSoftmax  
paddle.nn.SmoothL1Loss  
paddle.nn.LeakyReLU  
paddle.nn.Linear  
paddle.optimizer.lr.LambdaDecay
```

- 由多个单词组成的类名，最后一个单词应表示类型

```
# SimpleRNNCell 继承自 RNNCellBase  
paddle.nn.SimpleRNNCell  
  
# BrightnessTransform 继承 BaseTransform  
paddle.vision.BrightnessTransform
```

- 函数的名称应采用全小写，单词之间采用下划线分割

```
# 使用小写  
paddle.nn.functional.conv2d  
paddle.nn.functional.embedding  
# 如果由多个单词构成应使用下划线连接  
paddle.nn.functional.mse_loss  
paddle.nn.functional.batch_norm  
paddle.nn.functional.log_softmax
```

- 但一些约定俗成的例子可以保持不加下划线

```
paddle.isfinite  
paddle.isnan  
paddle.isinf  
paddle.argsort  
paddle.cumsum
```

- API 命名时，缩写的使用不应引起歧义或误解；在容易引起歧义或误解的情况下，需要使用全称，比如

```
# pytorch 使用 ge, lt 之类的缩写，可读性较差，应保留全称，与 numpy 和 paddle 保持一致
paddle.tensor.greater_equal
paddle.tensor.less_than
# optimizer 不使用缩写
paddle.optimizer.SGD
# parameter 不使用缩写
paddle.nn.create_parameter
```

- 在用于 API 命名时，常见的缩写列表如下：

```
conv、max、min、prod、norm、gru、lstm、add、func、op、num、cond
```

- 在用于 API 命名时，以下建议使用全称，不推荐使用缩写

| 不规范命名 | 规范命名 | :----- | :----- | :----- | :-----
floor_divide	lr	learning_rate	act	activation	eps	epsilon	val	value	var	variable	param	parameter	
prog	program	idx	index	exe	executor	buf	buffer	trans	transpose	img	image	loc	location
len	length												

- API 命名不应包含版本号

```
# 不使用版本号
paddle.nn.multiclass_nms2
```

- 常见的数学计算 API 中的逐元素操作不需要加上 elementwise 前缀，按照某一轴操作不需要加上 reduce 前缀，一些例子如下

| paddle2.0 之前 | pytorch | numpy | tensorflow | paddle2.0 之后 | :----- | :----- | :----- | :-----
| :----- | :----- | :----- | :----- | :----- | :----- | :----- | :-----
| elementwise_add | add | add | add | add | elementwise_sub | sub | subtract | subtract | elementwise_mul |
| mul | multiply | multiply | multiply | elementwise_div | div | divide | divide | elementwise_min | min |
| minimum | minimum | minimum | minimum | elementwise_max | max | maximum | maximum | maximum | reduce_sum | sum |
| sum | reduce_sum | sum | reduce_prod | prod | prod | reduce_prod | prod | reduce_min | min | min | reduce_min |
| min | reduce_max | max | max | reduce_max | max | reduce_all | all | all | reduce_all | all | reduce_any | any |
| any | reduce_any | any | reduce_mean | mean | mean | reduce_mean | mean |

- 整数取模和取余

目前整除和取余取模运算符重载在不同的语言和库中对应关系比较复杂混乱（取余运算中余数和被除数同号，取模运算中模和除数同号。取余整除是对商向 0 取整，取模整除是对商向负取整）

- 常用组网 API 命名规范

```
# 卷积：
paddle.nn.Conv2D #采用 2D 后缀，2D 表示维度时通常大写
paddle.nn.Conv2DTranspose
```

(下页继续)

(续上页)

```
paddle.nn.functional.conv2d
paddle.nn.functional.conv2d_transpose
# 池化：
paddle.nn.MaxPool2D
paddle.nn.AvgPool2D
paddle.nn.MaxUnpool2D
paddle.nn.functional.max_pool2d
paddle.nn.functional.avg_pool2d
paddle.nn.functional.max_unpool2d
# 归一化：
paddle.nn.BatchNorm2D
paddle.nn.functional.batch_norm
```

参数命名规范

- 参数名称全部使用小写

```
paddle.nn.functional.mse_loss: def mse_loss(input, label, reduction='mean', ↵
    name=None):
```

- 参数名可以区分单复数形态，单数表示输入参数是一个或多个变量，复数表示输入明确是含有多个变量的列表

```
paddle.nn.Softmax(axis=-1) # axis明确为一个int数
paddle.squeeze(x, axis=None, dtype=None, keepdim=False, name=None): # ↵
    axis可以为一数也可以为多个
paddle.strided_slice(x, axes, starts, ends, strides, name=None)
    #axis明确是多个数，则参数用复数形式axes
```

- 函数操作只有一个待操作的张量参数时，用 x 命名；如果有 2 个待操作的张量参数时，且含义明确时，用 x, y 命名

```
paddle.sum(x, axis=None, dtype=None, keepdim=False, name=None)
paddle.divide(x, y, name=None)
```

- 原则上所有的输入都用 x 表示，包括 functional 下的 linear, conv2d, lstm, batch_norm 等
- 原则上都要具备 name 参数，用于标记 layer，方便调试和可视化
- loss 类的函数，使用 input 表示输入，使用 label 表示真实预测值/类别，部分情况下，为了更好的便于用户理解，可以选用其他更恰当的参数名称。如 softmax_with_logits 时，输入参数名可用 logits

```
paddle.nn.functional.mse_loss(input, label, reduction='mean', name=None)
paddle.nn.functional.cross_entropy(input,
                                    label,
                                    weight=None,
                                    ignore_index=-100,
                                    reduction='mean',
                                    soft_label=False,
                                    axis=-1,
                                    name=None):
```

- Tensor 名称和操作

附-中英术语表

以下深度学习相关术语主要来自 deep learning 这本书中的术语表

飞桨 API Python 端开发指南

本文将介绍为 Paddle 开发新的 API 时需要在 Python 端完成的内容以及注意事项。

开发 Python API 代码

这分为两种情况，Paddle 的 API 包含需要开发 c++ operator 的和不需要开发 operator 而仅使用现有 Python API 组合得到的两种，但两种情况下均有 Python 端的开发工作。

1. 包含 c++ operator 的开发的情况，需要在 Python 端添加相应 API 以调用对应的 operator;
2. 不需要开发 c++ operator 的情况，需要在 Python 端添加相应 API 以调用其他 API 组合实现功能;

文件位置与 API 名称

Python API 的文件位置遵循功能相似的放在一起的原则。大的功能分类可以参考 API 目录结构规范。

大部分常用的数组运算 API (在 numpy 中有功能相似的 numpy.*** API) 放在 paddle/tensor 目录下。具体的功能细分如下：

与 paddle/tensor 功能类似，paddle.nn.functional 中也包含许多用于操作 tensor 的函数，但是这里主要是放一些更常用于神经网络中的函数，比如 batch_norm, conv2d，这些往往可能在 numpy 中没有直接对应的函数。

写新的 API 的时候可以参考该 API 的功能和哪一类更为相似，如果有不确定的情况，请 新建 ISSUE 说明。

将 API 绑定为 Tensor 的方法

在 paddle 中的许多计算函数，既能够作为独立函数使用，也能作为 Tensor 的方法使用。作为 Tensor 方法使用则可以更方便地链式调用。例子如下：

```
x = paddle.randn([2, 3])

paddle.abs(x) # 与 x.abs() 等价
paddle.sin(paddle.abs(x)) # 与 x.abs().sin() 等价

paddle.sum(x, axis=0) # 与 x.sum(axis=0) 等价
```

这两种使用方式的对应规则是，当作为 Tensor 方法调用时，相当于自动把该 Tensor 作为独立函数的第一个参数传入，其余参数的传入则和独立函数的使用一致。目前 paddle/tensor 子目录下的许多 API 都支持这样的调用方式。

如需让新增的函数支持作为 Tensor 方法调用则需要将函数名添加到 Python/paddle/tensor/__init__.py 中的 tensor_method_func 列表中。具体的做法是在 Python/paddle/tensor/__init__.py 中 import 所需的函数，然后将其名字加入 tensor_method_func 列表。

API 的正式名称与公开 API 列表

在 Python 中，如果模块 a 中导入了模块 b 提供的函数或者类 f，那么开发者想要使用函数 f，既可以模块 a 中导入，也可以从模块 b 中导入。

```
# b.py
def f():
    pass

# a.py
from b import f

# client.py
from b import f # it's ok
from a import f # it's ok, too
```

但是 Paddle 对于 API 有一个推荐的名称，比如函数 logsumexp 定义在 Python/paddle/tensor/math.py，但是又在 Python/paddle/tensor/__init__.py 中被 import，并且也在 Python/paddle/__init__.py 中被 import。

```
# Python/paddle/tensor/math.py
def logsumexp(...):
    ...
```

(下页继续)

(续上页)

```
# Python/paddle/tensor/__init__.py
from .math import logsumexp

# Python/paddle/__init__.py
from .tensor.math import logsumexp
```

实际上 `import paddle` 之后，可以通过 `paddle.logsumexp`, `paddle.tensor.logsumexp` 和 `paddle.tensor.math.logsumexp` 来调用这个函数，但是推荐使用 `paddle.logsumexp` 这个名称。这在 Paddle 中的做法是仅在 `Python/paddle/__init__.py` 文件的 `__all__` 列表中加入 "logsumexp"，而不在 `Python/paddle/tensor/__init__.py` 和 `Python/paddle/tensor/math.py` 的 `__all__` 列表中加入 "logsumexp"。

遵循这样的规范，按照 API 的正式命名，到对应的文件中，总可以找到 `__all__` 列表，其中包含想要查找的函数。而且这样一个 API 只出现在一个 `__all__` 列表中。

这个列表也作为某个模块或包的公开 API 列表，不加入列表的不视为公开 API。

比如已知 API 正式名称 `paddle.logsumexp`；

1. 如果 `Python/paddle.py` 存在 (`paddle` 是一个模块)，则可以在 `Python/paddle.py` 中查找 `__all__` 列表；
2. 如果 `Python/paddle` 是一个文件夹 (`paddle` 是一个包)，则可以在 `Python/paddle/__init__.py` 中查找 `__all__` 列表。

将 API 名字加入 `__all__` 列表的时候需要遵循上述的规则，仅将 API 名字加入正式名称对应的包或者模块的 `__all__` 列表中。

Tip: 当出现类似把一个元素放入一个集中管理的列表的操作时，可以考虑按照字母表顺序插入列表中的合适位置。因为如果有多人同时新增 API 时，这样的方式比直接加在末尾更不容易出现冲突。

Python API 的一般写法

下面介绍 `paddle` Python API 开发的一些惯例，主要函数类的接口。

这类的接口需要兼容动态图和静态图。在动态图下，函数会被多次执行；而在静态图下，函数仅在组网时被调用，真正被多次执行的是组网得到的结果。但 API 在动态图和静态图下的行为是保持一致的。

关于 API 的命名，参数命名等的一般规范，可以参考 [飞桨 API](#) 的设计和命名规范。

Python API 一般包含如下的部分：

输入参数检查

这一步包括必要的类型检查、值检查、输入 Tensor 的形状、`dtype` 等检查，确保组网能正常运行等。其中检测 Tensor 的数据类型可以用 `check_variable_and_dtype` 和 `check_type` 函数进行检测。

如果输入参数检查的代码逻辑比较复杂但仅用于某个函数，可以在该函数内定义一个检查输入参数的内函数，比如 `paddle.mm` 中定义的内函数 `__check_input`。

```
def __check_input(x, y):
    var_names = {'x': x, 'y': y}
    for name, val in var_names.items():
        check_variable_and_dtype(val, name,
                                ['float16', 'float32', 'float64'], 'mm')

    x_shape = list(x.shape)
    y_shape = list(y.shape)
    if len(x_shape) == 1:
        x_shape = [1] + x_shape
    if len(y_shape) == 1:
        y_shape = y_shape + [1]

    # check the inner 2 dimensions
    if x_shape[-1] != y_shape[-2]:
        if not ((x_shape[-1] == -1) or (y_shape[-2] == -1)):
            raise ValueError(
                "After performing an optional transpose, Input X's width should"
                "be "
                "equal to Y's width for multiplication"
                "prerequisites. But received X's shape: %s, Y's shape: %s\n"
                "% (x_shape, y_shape))"

    if len(y_shape) > 2 and len(x_shape) > 2:
        for i, dim_x in enumerate(x_shape[:-2]):
            # don't check neg shape
            if dim_x < 0 or y_shape[i] < 0:
                continue
            if dim_x != y_shape[i]:
                raise ValueError(
                    "When the matrix is larger than 2 dimensions, the higher"
                    "dimensional values of the two matrices need to be equal."
                    "But received x_shape[%d] != y_shape[%d]. X's shape: %s,"
                    "Y's shape: %s.\n" % (i, i, x_shape, y_shape))
```

注意：输入数据类型的检查一般仅在静态图分支中使用。主要原因是静态图下该函数仅被执行一次，发生在组网时，而动态图下该函数会被多次执行，Python 端过多的输入检查会影响执行效率。并且由于动态图即时执行的优势，如果发生错误也可以通过分析 c++ 端的报错信息定位问题。

例子：

```
def mm(input, mat2, name=None):
    # 为了突出重点，省略部分代码

    # 动态图，直接调用 op 对应的 CPython 函数
    if paddle.in_dynamic_mode():
        return _C_ops.matmul_v2(input, mat2)

    # 静态分支
    ## 检测输入
    __check_input(input, mat2)

    ## 构造输出，添加 op，返回输出
    helper = LayerHelper('mm', **locals())
    out = helper.create_variable_for_type_inference(dtype=input.dtype)
    helper.append_op(
        type='matmul_v2', inputs={'X': input,
                                  'Y': mat2}, outputs={'Out': out})
    return out
```

执行计算

如果是调用现有的 Python API 组合来实现功能，那么因为 paddle 的大多数 Python API 兼容动态图和静态图，所以一般不需要像上述代码那样区分动态图和静态图，直接调用所需的 API 即可。例：

```
def ones(shape, dtype=None, name=None):
    if dtype is None:
        dtype = 'float32'
    return fill_constant(value=1.0, shape=shape, dtype=dtype, name=name)
```

因为 fill_constant 里已经处理了动态图和静态图的情况，所以直接调用即可。

而如果 API 的实现中需要调用一个 op 时，则需要根据动态图和静态图使用不同的写法，用 paddle.in_dynamic_mode() 获取当前状态走不同的分支。

动静态图分支

参考前面 paddle.nn.functional.kl_div 的代码，动态图分支的写法一般是调用 API 对应的 CPython 函数。

```
_C_ops.matmul_v2(input, mat2)
```

_C_ops 是 Python/paddle/_C_ops.py，其中从 paddle 编译得到的二进制文件中 import 了 c++ operator 对应的 Python C 函数，函数名和 operator 名一致。如希望调用名为 matmul_v2 的 operator，则使用 _C_ops.matmul_v2，然后传入参数。

其中参数分为两个部分，Tensor 对于 Tensor 类型的输入，直接按照定义 opmaker 时添加输入的次序，以按位置传参的方式传入。关于 opmaker 可以参考 定义 OpProtoMaker 类（本文中用 opmaker 简称 operator）。

而对于非 Tensor 类型的输入（对应 opmaker 中的 Attribute），则以 attribute 名，attribute 值交替的方式传入，这类似 Python 中的按关键字传参的方式。然后返回调用函数得到的结果。

而对于静态图，则一般分为创建输出 Tensor，添加 operator 两步。

```
loss = _C_ops.kldiv_loss(input, label, 'reduction', 'none')

# layerhelper 创建准备工作
helper = LayerHelper('kl_div', **locals())

# 创建输出 Tensor
loss = helper.create_variable_for_type_inference(dtype=input.dtype)

# 将输入 Tensor, 输出 Tensor, 非 Tensor 的 attributes 以三个字典的形式
# 作为参数添加 operator
helper.append_op(
    type='kldiv_loss',
    inputs={'X': input,
            'Target': label},
    outputs={'Loss': loss},
    attrs={'reduction': 'none'})
```

上述的代码中，动态图分支的 input, label 对应静态图分支中的 inputs 字典，其次序和 opmaker 中定义的有关。而静态图中的 attrs 字典在动态图分支中则以 key, value 交替的形式排列，如果有多个 attribute，则依次排列。

开发单元测试代码

添加 Operator 单元测试

如果开发了 c++ operator，那么需要添加 operator 的单元测试，需要继承 OpTest 写作测试用例。文件位置在 Python/paddle/fluid/tests/unittests/，一般以 test_\${op_name}_op.py 的形式命名。

单元测试相关的开发规范可以参考

C++ OP 开发（新增原生算子），Op 开发手册 (Operator Development Manual).

在此不作展开，主要讲述 Python API 的单元测试。

添加 Python API 单元测试

无论是否开发了 c++ operator, 对于 Python API 都需要添加单元测试, 文件路径在 Python/paddle/fluid/tests/unittests/, 一般以 test_\${api_name}.py 的形式命名。

如果为这个 API 也开发了对应的 c++ operator, 那么也可以把对 API 的单元测试和 operator 的单元测试写在同一个文件中, 文件位置在 Python/paddle/fluid/tests/unittests/, 一般以 test_\${op_name}_op.py 的形式命名。

对 Python API 的单元测试直接继承 `UnitTest.TestCase`, 一般来说需要用 `numpy/scipy` 中的对应功能作为参考, 如果 `numpy/scipy` 中没有现成的对应函数, 可以用 `numpy/scipy` 实现一个作为参考, 并以这个为基准对新增的 paddle Python API 进行测试, 如 `test_softmax_op`。

如果新增的 API 没有使用新增的 c++ operator, 可以不必测试反向功能 (因为 operator 的新增本身要求 operator 单元测试, 这本身就会测试反向功能)。常见的流程是构建相同的输入, 调用参考的实现和新增的 Python API, 对比结果是否一致。一般用 `self.assertTrue(numpy.allclose(actual, desired))` 或者 `numpy.testing.assert_allclose(actual, desired)` 来进行数值对比。

其中, `numpy.testing.assert_allclose` 相对误差和绝对误差是 `rtol=1e-07, atol=0`; `numpy.allclose` 的相对误差和绝对误差是 `rtol=1e-05, atol=1e-08`, 前者比后者更严格。一般进行单元测试的时候, 都使用默认的误差阈值, 如需设置自定义的阈值, 需要说明原因。

注意: 对于 Python API 的单元测试, 必须添加动态图和静态图的测试 case, 以确保所添加的 API 以及它在不区分动态图静态图分支的情况下调用的其他 API 在两种情况下工作都正常, 而且结果符合预期。

因为 `unittest` 各个 `case` 的运行次序是不确定的, 为了保证不同的测试 `case` 运行在正确的运行模式 (动态图/静态图) 上, 常见的做法有:

1. 在每个测试 `case` 的起始部分, 显式切换 paddle 的运行模式, 用 `paddle.enable_static` 和 `paddle.disable_static` 分别激活和取消静态图模式。

比如 `Python/paddle/fluid/tests/unittests/test_activation_op.py` 中的 `TestHardtanhAPI` 中在 `test_static_api` 和 `test_dygraph_api` 的开头分别切换了状态。

```
class TestHardtanhAPI(unittest.TestCase):
    # test paddle.nn.Hardtanh, paddle.nn.functional.hardtanh
    def setUp(self):
        np.random.seed(1024)
        self.x_np = np.random.uniform(-3, 3, [10, 12]).astype('float32')
        self.place=paddle.CUDAPlace(0) if paddle.is_compiled_with_cuda() \
            else paddle.CPUPlace()

    def test_static_api(self):
        paddle.enable_static()
        with paddle.static.program_guard(paddle.static.Program()):
            x = paddle.fluid.data('X', [10, 12])
```

(下页继续)

(续上页)

```

        out1 = F.hardtanh(x)
        m = paddle.nn.Hardtanh()
        out2 = m(x)
        exe = paddle.static.Executor(self.place)
        res = exe.run(feed={'X': self.x_np}, fetch_list=[out1, out2])
        out_ref = ref_hardtanh(self.x_np)
        for r in res:
            self.assertEqual(np.allclose(out_ref, r), True)

    def test_dygraph_api(self):
        paddle.disable_static(self.place)
        x = paddle.to_tensor(self.x_np)
        out1 = F.hardtanh(x)
        m = paddle.nn.Hardtanh()
        out2 = m(x)
        out_ref = ref_hardtanh(self.x_np)
        for r in [out1, out2]:
            self.assertEqual(np.allclose(out_ref, r.numpy()), True)

        out1 = F.hardtanh(x, -2.0, 2.0)
        m = paddle.nn.Hardtanh(-2.0, 2.0)
        out2 = m(x)
        out_ref = ref_hardtanh(self.x_np, -2.0, 2.0)
        for r in [out1, out2]:
            self.assertEqual(np.allclose(out_ref, r.numpy()), True)
        paddle.enable_static()

```

- 将静态图和动态图测试定义为不以 `test` 开头的函数（因此它们默认不算 `test case`），然后定义一个 `test` 开头的函数，切换不同的状态去运行它。例如 `Python/paddle/fluid/tests/unittests/test_l1_loss.py:73`

```

def test_cpu(self):
    paddle.disable_static(place=paddle.fluid.CPUPlace())
    self.run_imperative()
    paddle.enable_static()

    with fluid.program_guard(fluid.Program()):
        self.run_static()

```

- 将动态图和静态图的测试 `case` 分在不同的 Python 文件中，`import paddle` 后在模块级别设置 `paddle` 的运行模式。

比如 `Python/paddle/fluid/tests/unittests/rnn/test_rnn_cells.py` 和 `Python/paddle/fluid/tests/unittests/rnn/test_rnn_cells_static.py` 的做法。

4. 在测试模块级别设定 paddle 的运行模式为静态图（一般是在一个模块的开始，而不是写在 `if __name__ == "__main__":` 里）。然后在需要使用动态图的 case 里，将动态图部分的代码至于 `dygraph.guard` 上下文管理器内。

这是老式的写法，目前不再推荐这么写，但已有的代码库中也存在这样的模式。

注意单元测试要求新增代码单元测试行覆盖率达到 90%.

运行单元测试

运行单元测试需要在 build 目录下，以 `ctest ${test_name}` 的命令运行。其中 `test_name` 指的是所需运行测试 `target` 的名字，和上述添加的单元测试文件名字相同，但不带 `.py` 后缀。

比如运行 `Python/paddle/fluid/tests/unittests/test_logsumexp.py` 就可以用 `ctest test_logsumexp` 运行。

这需要在 `cmake` 生成构建方案的时候加入选项 `-DWITH_TESTING=ON`，这样单元测试就会生成对应的测试 `target`。

对于需要开发 `c++ operator` 的 API，可以把 `operator` 的单元测试与 Python API 的单元测试写在一个文件中，也可以分开两个文件，分别测试 `operator` 和 Python API.

`ctest` 还可以批量运行名字匹配某个正则表达式的测试 `target`，通过 `-R` 参数传入正则表达式。比如通过 `ctest -R test_logsumexp` 就可以运行所有以 `test_logsumexp` 开头的单测 `target`.

此外，需要单元测试输出更详细的信息以便 `debug` 时，可以在运行 `ctest` 时传入 `-v` 或者 `-VV` 选项以查看更详细的输出，如 `ctest -V -R test_logsumexp`。

API 文档写作

文档写作可以参考 [文档贡献指南](#). 里面有详细的关于文档格式要求，文件所放的位置，以及提交代码的方式的详细说明。

提前 PR 后，github 上的 Bot 会给出根据所提交的中文文档所生成的官网文档的链接，可以点进去查看新增的文档所渲染出的页面效果，看是否符合预期。尤其需要注意检查是否有错别字，数学公式，示例代码渲染是否正确等问题。例如

<https://github.com/PaddlePaddle/docs/pull/4418>

CI 系统相关说明

当添加新的 API 时需要通过所有的 Required 流水线才能 merge 代码。

注意：其中 PR-CI-APPROVAL 和 PR-CI-Static-Check 这两个 CI 流水线会有需要特定开发者 approve 才能通过的机制，在除了这两个之外的流水线通过后，可以联系开发者提醒他们 review 代码。

[TODO 遇到不明原因的 CI 失败怎么办？]

其他 Tips

调试 Python 代码时减少重编译

如果你的修改不涉及 c++ 代码，那么一般不需要重新编译就可以重新运行测试，以验证刚发生的修改是否解决了问题。

paddle 编译过程中，对于 Python 代码的处理方式是，先把它们 copy 到 build 目录，对于 Python API 和 Python 单元测试所在的文件也是如此处理。

比如 Python/paddle/fluid/tests/unittests/test_bmm_op.py copy 到 build 目录后位置是 build/Python/paddle/fluid/tests/unittests/test_bmm_op.py。并且通过 ctest 运行单元测试时，会把 build/Python 这个目录加入 PYTHONPATH，因此它所调用的单元测试文件和 Python API 代码文件也是 build 目录里的那一份。

如果你的修改没有涉及任何 c++ 文件，那么你也可以直接在 build 目录下修改对应的文件，直到确保问题解决，然后，把文件 copy 回去覆盖 Paddle 目录的对应文件。（注意：不要忘记这一步，因为重新 build 的时候，会再次从 Paddle 目录 copy Python 文件，如果最后忘了 copy 到 Paddle 目录，那么你的修改会因为再次的编译而被覆盖。）

参考资料

1. Op 开发手册 (Operator Development Manual)
2. 飞桨 API 的设计和命名规范
3. 新增 API 测试及验收规范
4. 文档贡献指南
5. 飞桨 API 文档书写规范

C++ OP 开发

注：飞桨原生算子的开发范式正在进行重构与升级，升级后算子开发方式会大幅简化，我们会及时更新本文档内容，升级后的算子开发范式预计会在 2.3 版本正式上线。

1. 概念简介

本教程对新增原生算子的方法进行介绍，首先新增一个算子大概需要以下几个步骤：

1. 新增算子描述及定义：描述前反向算子的输入、输出、属性，实现 InferMeta 函数
2. 新增算子 Kernel：实现算子在各种设备上的计算逻辑
3. 封装 Python API：封装 Python 端调用算子的接口
4. 添加单元测试：验证新增算子的正确性

以上 4 个步骤添加的文件，在 Paddle 中的位置如下（假设算子名为 xxx）：

关于 Python API 所处位置，可以参考 [飞桨官方 API 文档](#)，了解各个目录存放 API 的性质，从而决定具体的放置目录。

接下来，我们以 Trace 操作，计算输入 Tensor 在指定平面上的对角线元素之和，并输出相应的计算结果，即 TraceOp 为例来介绍如何新增算子。

2. 新增算子描述及定义

算子描述及定义是定义运算的基本属性，本身是设备无关的。

首先简单介绍新增算子（以下简称 Op）描述需要用到的基类。

- framework::OpProtoAndCheckerMaker：描述该 Op 的输入、输出、属性、注释。
- framework::OperatorBase：Operator（简写，Op）基类。
- framework::OperatorWithKernel：继承自 OperatorBase，Op 有计算函数，称作有 Kernel。

根据是否包含 Kernel，可以将 Op 分为两种：包含 Kernel 的 Op 和不包含 kernel 的 Op：

- 包含 Kernel 的 Op 继承自 OperatorWithKernel：这类 Op 的功能实现与输入的数据类型、数据布局、数据所在的设备以及 Op 实现所调用第三方库等有关。比如 ConvOp，如果使用 CPU 计算，一般通过调用 mkl 库中的矩阵乘操作实现，如果使用 GPU 计算，一般通过调用 cublas 库中的矩阵乘操作实现，或者直接调用 cudnn 库中的卷积操作。
- 不包含 Kernel 的 Op 继承自 OperatorBase：因为这类 Op 的功能实现与设备以及输入的数据不相关。比如 WhileOp、IfElseOp 等。

注：本教程仅介绍如何实现带有计算 Kernel 的算子，不带 Kernel 的算子主要用于特殊场景，一般没有需求。

2.1 定义 OpProtoMaker 类

Trace 运算由一个输入，三个属性与一个输出组成。

首先定义 ProtoMaker 来描述该 Op 的输入、输出、属性并添加注释：

```
class TraceOpMaker : public framework::OpProtoAndCheckerMaker {
public:
    void Make() override {
        AddInput("Input",
            "(Tensor) The input tensor, from which the diagonals are taken.");
        AddOutput("Out", "(Tensor) the sum along diagonals of the input tensor");
        AddAttr<int>(
            "offset",
            R"DOC((int, default 0), offset of the diagonal from the main diagonal. Can be_
↪both positive and negative. Defaults to 0.
        )DOC")
            .SetDefault(0);
        AddAttr<int>(
            "axis1",
            R"DOC((int, default 0), the first axis of the 2-D planes from which the_
↪diagonals should be taken.
        Can be either positive or negative. Default: 0.
        )DOC")
            .SetDefault(0);
        AddAttr<int>(
            "axis2",
            R"DOC((int, default 1), the second axis of the 2-D planes from which the_
↪diagonals should be taken.
        Can be either positive or negative. Default: 1.
        )DOC")
            .SetDefault(1);
        AddComment(R"DOC(
Trace Operator.

Return the sum along diagonals of the input tensor.

The behavior of this operator is similar to how `numpy.trace` works.

If Input is 2-D, returns the sum of diagonal.
If Input has larger dimensions, then returns an tensor of diagonals sum, diagonals be_
↪taken from
the 2-D planes specified by dim1 and dim2.

)DOC");
    }
};
```

`TraceOpMaker`继承自 `framework::OpProtoAndCheckerMaker`。

开发者通过覆盖 `framework::OpProtoAndCheckerMaker` 中的 `Make` 函数来定义 `Op` 所对应的 `Proto`, 通过 `AddInput` 添加输入参数, 通过 `AddOutput` 添加输出参数, 通过 `AddAttr` 添加属性参数, 通过 `AddComment` 添加 `Op` 的注释。这些函数会将对应内容添加到 `OpProto` 中。

上面的代码在 `TraceOp` 中添加两个输入 `X` 和 `Y`, 添加了一个输出 `Out`, 并简要解释了各自含义, 命名请遵守命名规范。

注意: `OpProtoMaker` 中不允许定义未使用的输入、输出或属性。

2.2 定义 `GradOpMaker` 类

通常情况下, 大部分 `Op` 只有一个对应的反向 `Op`, 每个 `Op` 都会有一个对应的 `GradOpMaker`。为方便代码编写, `paddle` 为只有一个反向的 `Op` 提供了一个模板类`SingleGradOpMaker`。`TraceOp` 的 `GradOpMaker` 需要继承这个模板类, 并在 `Apply()` 方法中设置反向 `Op` 的输入、输出和属性。此外, `paddle` 还提供了一个默认的 `GradOpMaker`, `DefaultGradOpMaker`, 该模板类会使用前向 `Op` 的全部输入 (`Input`) 输出 (`Output`) 以及输出变量所对应的梯度 (`Output@Grad`) 作为反向 `Op` 的输入, 将前向 `Op` 的输入变量所对应的梯度 (`Input@Grad`) 作为输出。

注意:

不要将反向 `Op` 不会用到的变量放到反向 `Op` 的输入列表中, 这样会导致这些不会被反向 `Op` 用到的变量的空间不能够及时回收, 进而有可能导致用到该 `Op` 的模型可以设置的 `batch_size` 较低。比如 `relu` 操作的前向操作为: `out.device(d) = x.cwiseMax(static_cast<T>(0))`; 反向操作为: `dout = dout * (out > static_cast<T>(0)).template cast<T>()`。显然, 反向操作中只是用到了 `out`、`dout`、`dx`, 没有用到 `x`。因此, 通常不建议使用默认的 `DefaultGradOpMaker`。

下面示例定义了 `TraceOp` 的 `GradOpMaker`。

```
template <typename T>
class TraceGradOpMaker : public framework::SingleGradOpMaker<T> {
public:
    using framework::SingleGradOpMaker<T>::SingleGradOpMaker;

protected:
    void Apply(GradOpPtr<T> grad_op) const override {
        grad_op->SetType("trace_grad");
        grad_op->SetInput("Input", this->Input("Input"));
        grad_op->SetInput(framework::GradVarName("Out"), this->OutputGrad("Out"));
        grad_op->SetOutput(framework::GradVarName("Input"),
                           this->InputGrad("Input"));
        grad_op->SetAttrMap(this->Attrs());
    }
};
```

注意:

- 有些 Op 的前向逻辑和反向逻辑是一样的，比如`ScaleOp`. 这种情况下，前向 Op 和反向 Op 的 Kernel 可以为同一个。
- 有些前向 Op 所对应的反向 Op 可能有多个，比如`SumOp`, 这种情况下，GradMaker 需要继承`framework::GradOpDescMakerBase`。
- 有些 Op 的反向对应另一个 Op 的前向，比如`SplitOp`, 这种情况下，`SplitGradMaker`中定义的 SplitOp 反向 Op 的 Type 就是 `concat`,
- 为高效地同时支持命令式编程模式(动态图)和声明式编程模式(静态图)，`SingleGradOpMaker` 是一个模板类，在注册 Operator 时需要同时注册 `TraceOpGradMaker<OpDesc>`(静态图使用) 和 `TraceOpGradMaker<OpBase>`(动态图使用)。

2.3 定义 Op 类

下面实现了 TraceOp 的定义：

```
class TraceOp : public framework::OperatorWithKernel {
public:
    using framework::OperatorWithKernel::OperatorWithKernel;
};
```

TraceOp 继承自 `OperatorWithKernel`。public 成员：

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

这句表示使用基类 `OperatorWithKernel` 的构造函数，也可写成：

```
TraceOp(const std::string &type, const framework::VariableNameMap &inputs,
        const framework::VariableNameMap &outputs,
        const framework::AttributeMap &attrs)
: OperatorWithKernel(type, inputs, outputs, attrs) {}
```

此外，Operator 类需要在有必要时重写 `GetExpectedKernelType` 接口。

`GetExpectedKernelType` 接口 `OperatorWithKernel` 类中用于获取指定设备（例如 CPU, GPU）上指定数据类型（例如 `double`, `float`）的 `OpKernel` 的方法。该方法的重写可见请参考 [原生算子开发注意事项 第 4 点](#) `GetExpectedKernelType` 方法重写。

通常 `OpProtoMaker` 和 `Op` 类的定义写在 `.cc` 文件中，和下面将要介绍的注册函数一起放在 `.cc` 中

2.4 实现 InferMeta 函数

InferMeta 函数是根据输入参数，推断算子输出 Tensor 基本信息的函数，推断的信息包括输出 Tensor 的 shape、data type 及 data layout，同时它也承担了检查输入数据维度、类型等是否合法的功能。

TraceOp 的 InferMeta 函数 实现如下：

```
void TraceInferMeta(  
    const MetaTensor& x, int offset, int axis1, int axis2, MetaTensor* out) {  
    int dim1 = axis1;  
    int dim2 = axis2;  
  
    auto x_dims = x.dims();  
  
    int dim1_ = dim1 < 0 ? x_dims.size() + dim1 : dim1;  
    int dim2_ = dim2 < 0 ? x_dims.size() + dim2 : dim2;  
  
    PADDLE_ENFORCE_GE(  
        x_dims.size(),  
        2,  
        phi::errors::OutOfRange(  
            "Input's dim is out of range (expected at least 2, but got %ld).",  
            x_dims.size()));  
    PADDLE_ENFORCE_LT(  
        dim1_,  
        x_dims.size(),  
        phi::errors::OutOfRange(  
            "Attr(dim1) is out of range (expected to be in range of [%ld, "  
            "%ld], but got %ld).",  
            -(x_dims.size()),  
            (x_dims.size() - 1),  
            dim1));  
    PADDLE_ENFORCE_LT(  
        dim2_,  
        x_dims.size(),  
        phi::errors::OutOfRange(  
            "Attr(dim2) is out of range (expected to be in range of [%ld, "  
            "%ld], but got %ld).",  
            -(x_dims.size()),  
            (x_dims.size() - 1),  
            dim2));  
    PADDLE_ENFORCE_NE(  
        dim1_,  
        dim2_,  
        phi::errors::InvalidArgumentException("The dimensions should not be identical "));
```

(下页继续)

(续上页)

```

        "%ld vs %ld.",
        dim1,
        dim2));
}

auto sizes = vectorize(x_dims);
if (x_dims.size() == 2) {
    sizes.clear();
    sizes.push_back(1);
} else {
    sizes.erase(sizes.begin() + std::max(dim1_, dim2_));
    sizes.erase(sizes.begin() + std::min(dim1_, dim2_));
}
out->set_dims(phi::make_ddim(sizes));
out->set_dtype(x.dtype());
}

```

其中，`MetaTensor` 是对底层异构 `Tensor` 的抽象封装，仅支持对底层 `Tensor` 的维度、数据类型、布局等属性进行读取和设置，具体方法请参考 `meta_tensor.h`。

InferMeta 的实现位置

`InferMeta` 的文件放置规则（以 `Tensor` 输入个数为判定标准）：

- `nullary.h`: 没有输入 `Tensor` 参数的函数
- `unary.h`: 仅有一个输入 `Tensor` 参数的函数
- `binary.h`: 有两个输入 `Tensor` 参数的函数
- `ternary.h`: 有三个输入 `Tensor` 参数的函数
- `multiary.h`: 有三个以上输入 `Tensor` 或者输入为 `vector<Tensor>` 的函数
- `backward.h`: 反向 op 的 `InferMeta` 函数一律在此文件中，不受前序规则限制

InferMeta 的编译时与运行时

在我们的静态图网络中，`InferMeta` 操作在编译时 (compile time) 和运行时 (run time) 都会被调用，在 compile time 时，由于真实的维度未知，框架内部用-1 来表示，在 run time 时，用实际的维度表示，因此维度的值在 compile time 和 run time 时可能不一致，如果存在维度的判断和运算操作，`InferMeta` 就需要区分 compile time 和 run time。

对于此类 `InferMeta` 函数，需要在函数声明的参数列表末尾增加 `MetaConfig` 参数，例如：

```

void ConcatInferMeta(const std::vector<MetaTensor*>& x,
                     const Scalar& axis_scalar,
                     MetaTensor* out,
                     MetaConfig config = MetaConfig());

```

然后在函数体中，使用 config.is_runtime 判断出于编译时还是运行时。

具体地，以下两种情况需要区分 compile time 和 run time。

1. 检查

如以下代码：

```
int i = xxxx;
PADDLE_ENFORCE_GT(x.dims()[i] , 10)
```

在 compile time 的时候，x.dims()[i] 可能等于-1，导致这个 PADDLE_ENFORCE_GT 报错退出。

如果用了以下 paddle 中定义的宏进行判断：

```
PADDLE_ENFORCE_EQ (x.dims()[i] , 10)
PADDLE_ENFORCE_NE (x.dims()[i] , 10)
PADDLE_ENFORCE_GT (x.dims()[i] , 10)
PADDLE_ENFORCE_GE (x.dims()[i] , 10)
PADDLE_ENFORCE_LT (x.dims()[i] , 10)
PADDLE_ENFORCE_LE (x.dims()[i] , 10)
```

都需要注意区分 compile time 和 run time

2. 运算

如以下代码：

```
auto x_dim = x.dims();
int i = xxxx;
y_dim[0] = x_dim[i] + 10
```

在 compile time 的时候，x_dim[i] 可能等于-1，得到的 y_dim[0] 等于 9，是不符合逻辑的

如果用到了类似以下的运算操作

```
y_dim[i] = x_dim[i] + 10
y_dim[i] = x_dim[i] - 10
y_dim[i] = x_dim[i] * 10
y_dim[i] = x_dim[i] / 10
y_dim[i] = x_dim[i] + z_dim[i]
```

都需要区分 compile time 和 run time

3. 处理的标准

- 检查：compile time 的时候不判断维度等于-1 的情况，但在 runtime 的时候检查
- 运算：-1 和其他数做任何运算都要等于-1

4. 参考代码

(1) 判断的实现方法可以参考 `SigmoidCrossEntropyWithLogitsInferMeta`, `SigmoidCrossEntropyWithLogits` 要求 `X` 和 `labels` 的两个输入，除了最后一维以外，其他的维度完全一致

```
bool check = true;
if ((!config.is_runtime) &&
    (phi::product(x_dims) <= 0 || phi::product(labels_dims) <= 0)) {
    check = false;
}

if (check) {
    PADDLE_ENFORCE_EQ(
        phi::slice_ddim(x_dims, 0, rank),
        phi::slice_ddim(labels_dims, 0, rank),
        phi::errors::InvalidArgumentException(
            "Input(X) and Input(Label) shall have the same shape "
            "except the last dimension. But received: the shape of "
            "Input(X) is [%s], the shape of Input(Label) is [%s].",
            x_dims,
            labels_dims));
}
```

(2) 运算的实现可以参考 `ConcatInferMeta`, `concat` 在 `InferShape` 判断时，调用 `ComputeAndCheckShape`，除了进行 `concat` 轴之外，其他的维度完全一致；在生成 `output` 的维度时，把 `concat` 轴的维度求和，其他的维度和输入保持一致。

```
const size_t n = inputs_dims.size();
auto out_dims = inputs_dims[0];
size_t in_zero_dims_size = out_dims.size();
for (size_t i = 1; i < n; i++) {
    PADDLE_ENFORCE_EQ(
        inputs_dims[i].size(),
        out_dims.size(),
        phi::errors::InvalidArgumentException("The shape of input[0] and input[%d] "
            "is expected to be equal."
            "But received input[0]'s shape = "
            "[%s], input[%d]'s shape = [%s].",
            i,
            inputs_dims[0],
            i,
            inputs_dims[i]));
}
for (size_t j = 0; j < in_zero_dims_size; j++) {
    if (j == axis) {
        if (is_runtime) {
            out_dims[axis] += inputs_dims[i][j];
        } else {
```

(下页继续)

(续上页)

```

    if (inputs_dims[i][j] == -1 || out_dims[j] == -1) {
        out_dims[axis] = -1;
    } else {
        out_dims[axis] += inputs_dims[i][j];
    }
}

} else {
    bool check_shape =
        is_runtime || (inputs_dims[0][j] > 0 && inputs_dims[i][j] > 0);
    if (check_shape) {
        // check all shape in run time
        PADDLE_ENFORCE_EQ(inputs_dims[0][j],
                           inputs_dims[i][j],
                           phi::errors::InvalidArgument(
                               "The %d-th dimension of input[0] and input[%d] "
                               "is expected to be equal."
                               "But received input[0]'s shape = "
                               "[%s], input[%d]'s shape = [%s].",
                               j,
                               i,
                               inputs_dims[0],
                               i,
                               inputs_dims[i]));
    }
    if (!is_runtime && out_dims[j] == -1 && inputs_dims[i][j] > 0) {
        out_dims[j] = inputs_dims[i][j];
    }
}
}

```

2.5 注册 Op

在 `xxx_op.cc` 文件中声明 `InferShapeFunctor`，并注册前向、反向 Op。

```
namespace ops = paddle::operators;
DECLARE_INFER_SHAPE_FUNCTOR(trace, TraceInferShapeFunctor,
                           PD_INFER_META(phi::TraceInferMeta));
REGISTER_OPERATOR(trace, ops::TraceOp, ops::TraceOpMaker,
                  ops::TraceGradOpMaker<paddle::framework::OpDesc>,
                  ops::TraceGradOpMaker<paddle::imperative::OpBase>,
                  TraceInferShapeFunctor);
```

(下页继续)

(续上页)

```
REGISTER_OPERATOR(trace_grad, ops::TraceOpGrad,
                  ops::TraceGradNoNeedBufferVarsInferer);
```

在上面的代码中，首先使用 `DECLARE_INFER_SHAPE_FUNCTOR` 声明 `InferShapeFunctor`，然后使用 `REGISTER_OPERATOR` 注册了 `ops::TraceOp` 类，算子名为 `trace`，该类的 `ProtoMaker` 为 `ops::TraceOpMaker`，其 `GradOpMaker` 分别是 `ops::TraceOpGradMaker<paddle::framework::OpDesc>`（静态图模式使用）和 `ops::TraceOpGradMaker<paddle::imperative::OpBase>`（动态图模式使用），同时将前面声明的 `TraceInferShapeFunctor` 一并放入注册列表。前向算子注册完成后，再使用 `REGISTER_OPERATOR` 注册 `ops::TraceGradOp`，类型名为 `trace_grad`。

3. 新增算子 Kernel

新增算子 Kernel 在 `paddle/phi/kernels` 目录中完成

3.1 kernels 目录结构

`paddle/phi/kernels` 基本目录结构如下

```
paddle/phi/kernels
./ (根目录放置设备无关的kernel声明和实现)
./cpu (仅放置cpu后端的kernel实现)
./gpu (仅放置gpu后端的kernel实现)
./xpu (仅放置百度kunlun后端的kernel实现)
./gpudnn
./funcs (放置一些支持多设备的、在多个kernel中使用的公共functor和functions)
...
```

一般情况下，新增算子仅需要关注 `kernels` 根目录及 `kernel` 所支持设备的子目录即可：

- `kernels` 根目录，放置设备无关的 `kernel.h` 和 `kernel.cc`
 - 例如，一个 `kernel` 除了一些简单的设备无关的 C++ 逻辑，关键计算逻辑均是复用已有的 `phi kernel` 函数实现的，那么这个 `kernel` 实现是天然能够适配所有设备及后端的，所以它的声明和实现均直接放置到 `kernels` 目录下即可
- `kernels` 下一级子目录，原则上按照 `backend` 分类按需新建，放置特定后端的 `kernel` 实现代码

下面给出两种典型 `kernel` 新增时文件放置位置的说明：

1. 新增与设备无关的 Kernel

该类 Kernel 实现与所有硬件设备无关，只需要一份代码实现，可参考 `reshape kernel`。其新增文件及目录包括：

- paddle/phi/kernels/xxx_kernel.h
- paddle/phi/kernels/xxx_kernel.cc

如果是反向 kernel，则使用 grad_kernel 后缀即可：

- paddle/phi/kernels/xxx_grad_kernel.h
- paddle/phi/kernels/xxx_grad_kernel.cc

2. 新增与设备相关、且 CPU&GPU 分别实现的 Kernel

还有部分 Kernel 的实现，CPU 和 GPU 上逻辑不同，此时没有共同实现的代码，需要区分 CPU 和 GPU 硬件。CPU 的实现位于 paddle/phi/kernels/cpu 目录下；GPU 的实现位于 paddle/phi/kernels/gpu 下，可参考 dot kernel, cast kernel 等。其新增文件及目录包括：

- paddle/phi/kernels/xxx_kernel.h
- paddle/phi/kernels/cpu/xxx_kernel.cc
- paddle/phi/kernels/gpu/xxx_kernel.cu

相应地，反向 kernel 新增文件为：

- paddle/phi/kernels/xxx_grad_kernel.h
- paddle/phi/kernels/cpu/xxx_grad_kernel.cc
- paddle/phi/kernels/gpu/xxx_grad_kernel.cu

3.2 Kernel 写法

3.2.1 声明 Kernel 函数

- 以 trace op 为例，首先在 paddle/phi/kernels 目录下新建 trace_kernel.h 文件，用于放置前向 Kernel 函数声明。

注：Kernel 函数声明的参数列表原则上与 Python API 参数列表一致

```
template <typename T, typename Context>
void TraceKernel(const Context& dev_ctx,
                 const DenseTensor& x,
                 int offset,
                 int axis1,
                 int axis2,
                 DenseTensor* out);
```

注：所有的 kernel 声明，统一放在 namespace phi 中，缩短函数的调用前缀使调用写法更加简洁

说明如下：

1. 模板为固定写法,第一个模板参数为数据类型 `T`,第二个模板参数为设备上下文 `Context`,`template <typename T, typename Context>`
2. 函数命名: `Kernel` 的命名统一加 `Kernel` 后缀。即: `Kernel` 名称 + `Kernel` 后缀, 驼峰式命名, 例如: `AddKernel`
3. 参数顺序: `Context, InputTensor …, Attribute …, OutTensor*`。即: 第一位参数为 `Context`, 后边为输入的 `Tensor`, 接着是输入的属性参数, 最后是输出的 `Tensor` 的指针参数。如果 `Kernel` 没有输入 `Tensor` 或者没有属性参数, 略过即可
4. 第 1 个函数参数, 类型为 `const Context&` 的 `dev_ctx`
5. 第 2 个函数参数, 输入 `Tensor`, 类型一般为 `const DenseTensor&`
6. 第 3-5 个函数参数, 均为 `attribute` (根据具体的含义, 选择特定的 `int`, `float`, `vector` 等类型), 多个 `attribute` 可以参考 `python` 端 API 定义的顺序, 变量命名对齐 `python api`
7. 第 6 个函数参数, 输出 `Tensor`, 类型一般为 `DenseTensor*`, 多个 `output` 可以参考 `python` 端 API 定义的顺序, 变量命名对齐 `python api`

特殊情况说明:

1. 特殊模板参数: 对于某些 `Kernel` (如 `reshape`, `copy`), 这些 `kernel` 不关注数据类型 `T`, 可以省去第一个模板参数, 即为: `template <typename Context>`
2. 特殊输入类型: 对于某些特殊 `Kernel` (如 `concat` 和 `split kernel`) 的部分输入或输出是数组类型的 `DenseTensor` (`OpMaker` 中有 `AsDuplicable` 标记), 此时输入类型为: `const std::vector<const DenseTensor*>&;` 输出类型为: `std::vector<DenseTensor*>`

3.2.2 实现 Kernel 函数

此处 trace op 的 `kernel` 属于前述第 2 中情况, 即 CPU 与 GPU Kernel 需要分别实现。

- cpu kernel 实现位于: `paddle/phi/kernels/cpu/trace_kernel.cc`
- gpu kernel 实现位于: `paddle/phi/kernels/gpu/trace_kernel.cu`

下面为 `TraceKernel` 的 cpu 实现:

```
template <typename T, typename Context>
void TraceKernel(const Context& dev_ctx,
                 const DenseTensor& x,
                 int offset,
                 int axis1,
                 int axis2,
                 DenseTensor* out) {
    auto* out_data = dev_ctx.template Alloc<T>(out);

    const DenseTensor diag =

```

(下页继续)

(续上页)

```

    funcs::Diagonal<T, Context>(dev_ctx, &x, offset, axis1, axis2);

if (diag.numel() > 0) {
    auto x = phi::EigenMatrix<T>::Reshape(diag, diag.dims().size() - 1);
    auto output = phi::EigenVector<T>::Flatten(*out);
    auto reduce_dim = Eigen::array<int, 1>({1});
    output.device(*dev_ctx.eigen_device()) = x.sum(reduce_dim);
    out->Resize(out->dims());
} else {
    std::fill(out_data, out_data + out->numel(), static_cast<T>(0));
}
}

```

Kernel 复用:

此处 TraceKernel 的实现并未复用其他 Kernel，但如果有需要也是可以复用的，Kernel 复用时，直接 include 相应 Kernel 头文件，在函数中调用即可，例如 `triangular_solve_kernel` 复用 `empty` 和 `expand kernel`。

首先在 triangular_solve_kernel.cc 头部 include 相应头文件：

```
#include "paddle/phi/kernels/empty_kernel.h"  
#include "paddle/phi/kernels/expand_kernel.h"
```

然后在 Kernel 实现中即可直接调用以上两个头文件中的 Kernel，代码片段如下：

```

// Tensor broadcast to 'out' and temp 'x_bst'
ScalarArray x_bst_dims(x_bst_dims_vec);
DenseTensor x_bst = phi::Empty<T, Context>(dev_ctx, x_bst_dims);
const T* x_bst_data = x_bst.data<T>();
ExpandKernel<T, Context>(dev_ctx, x, x_bst_dims, &x_bst);

```

反向 Kernel 的实现与前向是类似的，此处不再赘述，可以直接参考前述对应链接中的代码实现。

公共函数管理:

如果有一些函数会被多个 Kernel 调用，可以创建非 kernel 的文件管理代码，规则如下：

- 仅有当前 kernel 使用的辅助函数（具体到设备，比如 trace 的 cpu kernel），一律和 kernel 实现放到同一个设备文件夹中
 - 如果辅助函数相关代码较少，就直接和 kernel 实现放到同一个.cc/cu 中
 - 如果辅助函数相关代码较多，就在 kernel 所在的设备目录创建.h 管理代码
 - 有同设备多个 kernel 使用的辅助函数，在 kernel 所在的设备目录创建.h 放置代码
 - 有跨设备多个 kernel 使用的辅助函数，在 kernels/funcs 目录下创建.h/cc/cu 管理代码
 - 如果当前依赖的辅助函数可以直接归类到 kernels/funcs 目录下已有的文件中，则直接放过去，不用创建新的文件

反向 Kernel 参数映射函数添加

现阶段，反向 Kernel 除了实现外，还需要添加一个参数映射函数。

仍然以 trace op 为例，首先在 paddle/phi/ops/compat 目录下新建 trace_sig.cc 文件，用于放置这里的映射函数。

- 由于函数式 kernel 的一个最重要的特别就是参数顺序和类型（顺序和类型是关键，变量名称不影响），我们需要定义一个函数来做一个从 OpMaker 中如何获取信息，并且按照顺序传递给新的 kernel 函数；这个模块就是 OpArgumentMapping，trace 反向 op 的 OpArgumentMapping 定义如下，KernelSignature 共包含 4 个内容
 1. kernel 名称，这个是我们给 kernel 注册的时候的名称
 2. input list：这个要和 OpMaker（或者 GradOpMaker）中定义的 Key 要完全一致
 3. attribute list：这个要和 OpMaker（或者 GradOpMaker）中定义的 Key 要完全一致
 4. output list：这个要和 OpMaker（或者 GradOpMaker）中定义的 Key 要完全一致

```
#include "paddle/phi/core/compat/op_utils.h"

namespace phi {

KernelSignature TraceGradOpArgumentMapping(const ArgumentMappingContext& ctx) {
    return KernelSignature("trace_grad",
        {GradVarName("Out"), "Input"}, 
        {"offset", "axis1", "axis2"}, 
        {GradVarName("Input")});
}

} // namespace phi

PD_REGISTER_ARG_MAPPING_FN(trace_grad, phi::TraceGradOpArgumentMapping);
```

注：没有 input list 或 attribute list 的，相应花括号内留空，不能省略花括号

3.2.3 注册 Kernel 函数

注册 kernel 的方式比较简单，直接使用注册宏注册即可，示例如下：

```
PD_REGISTER_KERNEL(trace,
    CPU,
    ALL_LAYOUT,
    phi::TraceKernel,
    float,
    double,
```

(下页继续)

(续上页)

```

int,
int64_t,
phi::dtype::float16,
phi::dtype::complex<float>,
phi::dtype::complex<double>) {}

```

字段说明：

1. trace: kernel 名称，和 Op 的名称一致
2. CPU: backend 名称，一般主要就是 CPU 和 GPU
3. ALL_LAYOUT: kernel 支持的 Tensor 布局，一般为 ALL_LAYOUT，及支持所有布局类型
4. phi::TraceKernel: kernel 的函数名称，记得带上 namespace phi
5. 剩余的均为 Kernel 支持的数据类型

注意：

1. 如果忘记添加注册相关的头文件，会曝出一个 xx 的错误，如果遇到，请检查 include 的头文件
2. phi 下的注册宏后边是带函数体 {}，不是直接加分号，此处与旧的注册宏有小区别
3. 注册 kernel 的宏声明需要在 global namespace

3.3 编译测试

实现完 Op 和 Kernel 之后，建议先编译测试一下，编译成功之后，再继续后面的步骤。

详细的编译环境准备和执行流程可参考[从源码编译](#)，下面简单介绍几个主要步骤。在 Paddle 代码目录下创建并切换到 build 目录：

```
mkdir build && cd build
```

执行 cmake 命令，具体选项可参考[从源码编译](#)中的介绍，下面的命令为编译 Python3.7, GPU 版本，带测试，Release 版本的 Paddle。

```
cmake .. -DPY_VERSION=3.7 -DWITH_GPU=ON -DWITH_TESTING=ON -DCMAKE_BUILD_TYPE=Release
```

在 build 目录下，运行下面命令可以进行编译整个 paddle：

```
make -j$(nproc)
```

注意：新增 op 后请重新执行 cmake 命令，然后再执行 make 命令编译 paddle。

4. 封装 Python API

系统会对新增的 Op 即 Kernel 自动绑定 Python，并链接到生成的 lib 库中，然后在 Python 端定义相应的 API，在 API 内调用新增算子，并添加相应的中英文文档描述即可。

`paddle.trace` 的 Python API 实现位于 `python/paddle/tensor/math.py` 中，具体实现如下：

```
def trace(x, offset=0, axis1=0, axis2=1, name=None):
    """
    **trace**

    This OP computes the sum along diagonals of the input tensor x.

    If ``x`` is 2D, returns the sum of diagonal.

    If ``x`` has larger dimensions, then returns an tensor of diagonals sum,
    ↪diagonals be taken from
        the 2D planes specified by axis1 and axis2. By default, the 2D planes formed by
    ↪the first and second axes
        of the input tensor x.

    The argument ``offset`` determines where diagonals are taken from input tensor x:

    - If offset = 0, it is the main diagonal.
    - If offset > 0, it is above the main diagonal.
    - If offset < 0, it is below the main diagonal.
    - Note that if offset is out of input's shape indicated by axis1 and axis2, 0
    ↪will be returned.

    Args:
        x(Tensor): The input tensor x. Must be at least 2-dimensional. The input data
    ↪type should be float32, float64, int32, int64.
        offset(int, optional): Which diagonals in input tensor x will be taken.
    ↪Default: 0 (main diagonals).
        axis1(int, optional): The first axis with respect to take diagonal. Default:
    ↪0.
        axis2(int, optional): The second axis with respect to take diagonal. Default:
    ↪1.
        name (str, optional): Normally there is no need for user to set this property.
    ↪For more information, please refer to :ref:`api_guide_Name`. Default: None.

    Returns:
        Tensor: the output data type is the same as input data type.

    Examples:
```

(下页继续)

(续上页)

```

.. code-block:: python

    import paddle

    case1 = paddle.randn([2, 3])
    case2 = paddle.randn([3, 10, 10])
    case3 = paddle.randn([3, 10, 5, 10])
    data1 = paddle.trace(case1) # data1.shape = [1]
    data2 = paddle.trace(case2, offset=1, axis1=1, axis2=2) # data2.shape =
    ↪ [3]
    data3 = paddle.trace(case3, offset=-3, axis1=1, axis2=-1) # data2.shape =
    ↪ [3, 5]
    """
    def __check_input(input, offset, dim1, dim2):
        check_dtype(x.dtype, 'Input',
                    ['int32', 'int64', 'float16', 'float32', 'float64'],
                    'trace')

        input_shape = list(x.shape)
        assert len(input_shape) >= 2,
                                         \
            "The x must be at least 2-dimensional, " \
            "But received Input x's dimensional: %s.\n" % \
            len(input_shape)

        axis1_ = axis1 if axis1 >= 0 else len(input_shape) + axis1
        axis2_ = axis2 if axis2 >= 0 else len(input_shape) + axis2

        assert ((0 <= axis1_) and (axis1_ < len(input_shape))), \
            "The argument axis1 is out of range (expected to be in range of [%d, %d], \
            ↪but got %d).\n" \
            "% (-(len(input_shape)), len(input_shape) - 1, axis1)

        assert ((0 <= axis2_) and (axis2_ < len(input_shape))), \
            "The argument axis2 is out of range (expected to be in range of [%d, %d], \
            ↪but got %d).\n" \
            "% (-(len(input_shape)), len(input_shape) - 1, axis2)

        assert axis1_ != axis2_, \
            "axis1 and axis2 cannot be the same axis." \
            "But received axis1 = %d, axis2 = %d\n"(axis1, axis2)

    __check_input(input, offset, axis1, axis2)

```

(下页继续)

(续上页)

```

if paddle.in_dynamic_mode():
    return _C_ops.trace(x, 'offset', offset, 'axis1', axis1, 'axis2', axis2)

inputs = {'Input': [x]}
attrs = {'offset': offset, 'axis1': axis1, 'axis2': axis2}
helper = LayerHelper('trace', **locals())

out = helper.create_variable_for_type_inference(dtype=x.dtype)

helper.append_op(
    type='trace',
    inputs={'Input': [x]},
    attrs={'offset': offset,
           'axis1': axis1,
           'axis2': axis2},
    outputs={'Out': [out]})

return out

```

概念解释：LayerHelper 是一个用于创建 op 输出变量、向 program 中添加 op 的辅助工具类

- Python API 实现要点
 - 对输入参数进行合法性检查，即 `__check_input(input, offset, axis1, axis2)`
 - 添加动态图分支调用，即 `if paddle.in_dynamic_mode()` 分支
 - 添加静态图分支调用，即 dygraph mode 分支后剩余的代码
- Python API 放置位置
 - 根据 API 自身属性，结合现有目录分类情况，放置导致对应子目录中的相应文件中
 - 可以参考 [飞桨官方 API 文档](#) 中对各个子目录 功能和包含的 API 的介绍
- Python API 文档
 - 参考示例格式进行添加，内容尽可能准确、翔实，详细规范请参考 [PaddlePaddle 文档](#)

5. 添加单元测试

单测包括对比前向 Op 不同设备 (CPU、CUDA) 的实现、对比反向 OP 不同设备 (CPU、CUDA) 的实现、反向 Op 的梯度测试。下面介绍介绍[TraceOp](#)的单元测试。

注意：

单测中的测试用例需要尽可能的覆盖 Kernel 中的所有分支。

5.1 前向 Operator 单测

Op 单元测试继承自 OpTest。各项具体的单元测试在 TestTraceOp 里完成。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. 反向计算已经自动集成进测试框架，直接调用相应接口即可。

```
import unittest
import numpy as np
from op_test import OpTest


class TestTraceOp(OpTest):
    def setUp(self):
        self.op_type = "trace"
        self.init_config()
        self.outputs = {'Out': self.target}

    def test_check_output(self):
        self.check_output()

    def test_check_grad(self):
        self.check_grad(['Input'], 'Out')

    def init_config(self):
        self.case = np.random.randn(20, 6).astype('float64')
        self.inputs = {'Input': self.case}
        self.attrs = {'offset': 0, 'axis1': 0, 'axis2': 1}
        self.target = np.trace(self.inputs['Input'])
```

上面的代码首先导入依赖的包，下面是对 setUp 函数中操作的重要变量的详细解释：

- self.op_type = "trace"：定义类型，与 operator 注册时注册的类型一致。
- self.inputs：定义输入，类型为 numpy.array，并初始化。
- self.outputs：定义输出，并在 Python 脚本中完成与 operator 同样的计算逻辑，返回 Python 端的计算结果。

5.2 反向 operator 单测

而反向测试中：

- `test_check_grad` 中调用 `check_grad` 使用数值法检测梯度正确性和稳定性。
 - 第一个参数 `['Input']`：指定对输入变量 `Input` 做梯度检测。
 - 第二个参数 `'Out'`：指定前向网络最终的输出目标变量 `Out`。
- 对于存在多个输入的反向 Op 测试，需要指定只计算部分输入梯度的 case
 - 例如，`test_elementwise_sub_op.py` 中的 `test_check_grad_ignore_x` 和 `test_check_grad_ignore_y` 分支用来测试只需要计算一个输入梯度的情况
 - 此处第三个参数 `max_relative_error`：指定检测梯度时能容忍的最大错误值。

```
def test_check_grad_ignore_x(self):
    self.check_grad(
        ['Y'], 'Out', max_relative_error=0.005, no_grad_set=set("X"))

def test_check_grad_ignore_y(self):
    self.check_grad(
        ['X'], 'Out', max_relative_error=0.005, no_grad_set=set('Y'))
```

其他有关单元测试添加的注意事项请参考《Op 开发手册》及《Paddle 单元测试规范》。

5.3 编译和执行

`python/paddle/fluid/tests/unittests/` 目录下新增的 `test_*.py` 单元测试会被自动加入工程进行编译。

请注意，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`。

参考上述【3.3 编译测试】过程，编译成功后，在 `build` 目录下执行下面的命令来运行单元测试：

```
make test ARGS="-R test_trace_op -v"
```

或者执行：

```
ctest -R test_trace_op -v
```

注意事项：

- 注册 Op 时的类型名，需要和该 Op 的名字一样。即不允许在 `A_op.cc` 里面，注册 `REGISTER_OPERATOR(B, ...)` 等，这将会导致单元测试出错。

6. 其他编码要点

6.1 报错检查

实现 Op 时检查数据的合法性需要使用 PADDLE_ENFORCE 以及 PADDLE_ENFORCE_EQ 等宏定义，基本格式如下：

```
PADDLE_ENFORCE(表达式, 错误提示信息)
PADDLE_ENFORCE_EQ(比较对象A, 比较对象B, 错误提示信息)
```

如果表达式为真，或者比较对象 A=B，则检查通过，否则会终止程序运行，向用户反馈相应的错误提示信息。为了确保提示友好易懂，开发者需要注意其使用方法。

总体原则：任何使用了 PADDLE_ENFORCE 与 PADDLE_ENFORCE_XX 检查的地方，必须有详略得当的备注解释！错误提示信息不能为空！

报错提示信息书写建议：

1. [required] 哪里错了？为什么错了？
 - 例如：ValueError: Mismatched label shape
2. [optional] 期望的输入是什么样的？实际的输入是怎样的？
 - 例如：Expected labels dimension=1. Received 4.
3. [optional] 能否给出修改意见？
 - 例如：Suggested Fix: If your classifier expects one-hot encoding label, check your n_classes argument to the estimator and/or the shape of your label. Otherwise, check the shape of your label.

更详细的报错检查规范介绍请参考《[Paddle 报错信息文案书写规范](#)》。

C++ OP 开发注意事项

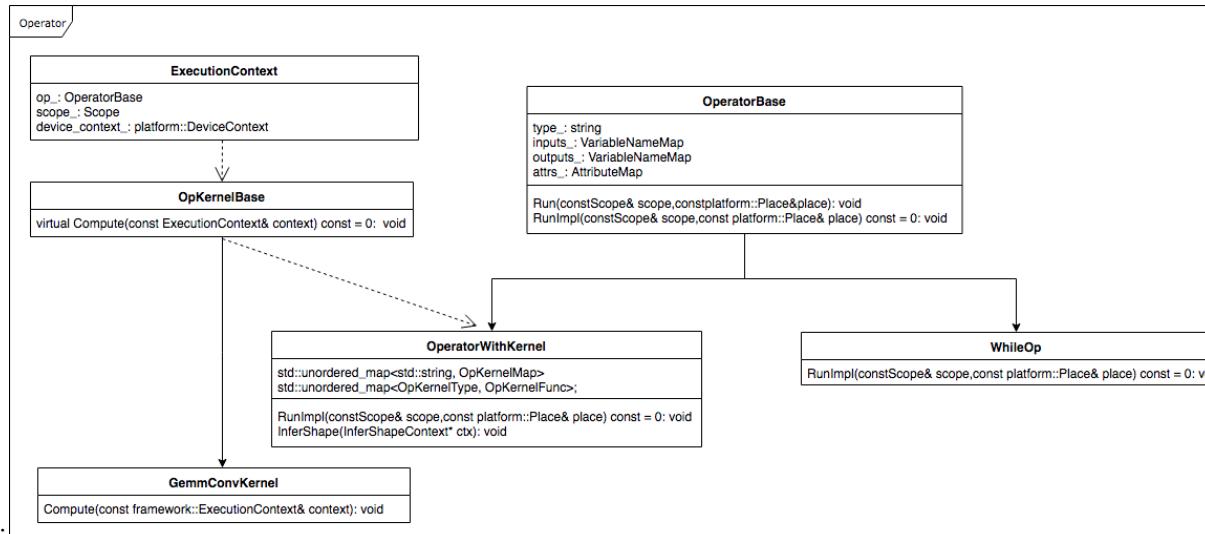
Paddle 中 Op 的构建逻辑

1. Paddle 中 Op 的构建逻辑

Paddle 中所有的 Op 都继承自 OperatorBase，且所有的 Op 都是无状态的，每个 Op 包含的成员变量只有四个：type、inputs、outputs、attribute。

Op 的核心方法是 Run，Run 方法需要两方面的资源：数据资源和计算资源，这两个资源分别通过 Scope 和 Place 获取。框架内部有一个全局的 DeviceContextPool，用来记录 Place 和 DeviceContext 之间的对应的关系，即每个 Place 有且仅有一个 DeviceContext 与之对应，DeviceContext 中存放了当前设备的计算资源。比如对于 GPU，这些资源包括 cudnn_handle、cublas_handle、stream 等，Op 内部所有的计算（数据拷贝和 CUDA Kernel 等）都必须在 DeviceContext 中进行。

Paddle 框架的设计理念是可以在多种设备及第三方库上运行，有些 Op 的实现可能会因为设备或者第三方库的不同而不同。为此，Paddle 引入了 OpKernel 的方式，即一个 Op 可以有多个 OpKernel，这类 Op 继承自 OperatorWithKernel，这类 Op 的代表是 conv_op，conv_op 的 OpKernel 有：GemmConvKernel、CUDNNConvOpKernel、ConvMKLDNNOpKernel，且每个 OpKernel 都有 double 和 float 两种数据类型。不需要 OpKernel 的代表有 WhileOp 等。



Operator 继承关系图：

进一步了解可参考: [multi_devices](#), [scope](#), [Developer's_Guide_to_Paddle_Fluid](#)

2.Op 的注册逻辑

```

每个 Operator 的注册项包括: C++ OpCreator creator_; GradOpMakerFN grad_op_maker_;
proto::OpProto* proto_{nullptr}; OpAttrChecker* checker_{nullptr};
InferVarTypeFN infer_var_type_; InferShapeFN infer_shape_;
  
```

通常 Op 注释时需要调用 REGISTER_OPERATOR, 即: REGISTER_OPERATOR(op_type, OperatorBase op_maker_and_checker_maker, op_grad_opmaker, op_infer_var_shape, op_infer_var_type)

注意:

- 对于所有 Op, 前三个参数是必须的, `op_type` 指明 op 的名字, `OperatorBase` 是该 Op 的对象, `op_maker_and_checker_maker` 是 op 的 maker 以及 Op 中 attr 的 checker。
- 如果该 Op 有反向, 则必须要有 `op_grad_opmaker`, 因为在 `backward` 会根据正向的 Op 中获取反向 Op 的 Maker。
- 框架提供了一个默认的 `op_grad_opmaker: DefaultGradOpDescMaker`, 这个 Maker 会将前向 Op 的输入和输出都作为反向 Op 的输入, 将前向 Op 的输入的梯度作为反向 Op 的输出, 并将前向 Op 的属性拷贝过来。注意: `DefaultGradOpDescMaker` 会将前向 Op 的所有输入输出都做反向 Op 的输入, 即使这个输入是没有必要的, 这将会导致无法对没有用到的变量做内存优化。
- 框架没有提供默认的 `op_infer_var_shape` 方法。如果该 Op 是无 OpKernel 的, 通常需要用户添加对应

的 `op_infer_var_shape` 方法；如果该 Op 是有 `OpKernel` 的，需要实现 `OperatorWithKernel` 中的 `InferShape` 方法，此时不需要提供 `op_infer_var_shape` 方法。具体实现可参考[while_op.cc](#), [conv_op.cc](#)。

5. 框架没有提供默认的 `op_infer_var_type` 方法，用户需要根据实际情况添加 `op_infer_var_type`。严格来说每个 Op 都应该注册一个 `InferVarType`，`op_infer_var_type` 根据输入的 Var 的 type 和 dtype 推断输出 Var 的 type 和 dtype。注意：在 Python 端的 `LayerHelper` 中 `create_variable_for_type_inference` 操作返回的 `Variable` 里面是 `LoDTensor`，C++ 端的 `InferVarType` 可以修改 `Variable` 的 type 和 dtype。

更多内容请参考：[如何写新的 Op](#)

写 Op 注意事项

1. Op 可以支持输入输出类型

Paddle 的 Op 的输入输出都是 `Variable`，从设计上讲，`Variable` 中可以存放任意类型，Op 的输入输出 `Variable` 可能是任意类型，通常情况下 `Variable` 中存放的是 `LoDTensor`、`SelectedRows`。

注意：

- 代码中经常出现 `context.Input<Tensor>("Input")`，并不表示“Input”的 `Variable` 是 `Tensor`，而是从“Input”的 `Variable` 的 `LoDTensor` 中获取 `Tensor`。如果“Input”的 `Variable` 是 `SelectedRows`，则会报错。
- 如 果“Input” 是 `SelectedRows`, `context->GetInputDim("Input")` 返回的是 `var->Get<SelectedRows>().GetCompleteDims()`，而不是 `SelectedRows` 中 `Tensor` 的 `Dim`。

2. 在 Op 内部不能对输入的数据做任何的改写

在 Op 内部绝不允许对输入数据做任何改写，因为可能存在其他 Op 需要读这个数据。

3. OpKernel 需要注册的数据类型

目前要求所有 `OpKernel` 都要注册 `double` 和 `float` 数据类型。

4. GetExpectedKernelType 方法重写

`GetExpectedKernelType` 方法是 `OperatorWithKernel` 类中用于获取指定设备（例如 CPU, GPU）上指定数据类型（例如 `double`, `float`）的 `OpKernel` 的方法。该方法通过获取输入变量内部的 `Tensor` 数据类型得知需要的 `Kernel` 数据类型，但是由于 `Tensor` 在此处可能尚未被初始化，所以在该方法内使用输入变量时需要进行必要的初始化检查。在新增含 `Kernel` 的 Op 的时候，关于该方法的重写需要注意以下两点。

4.1 仅在必要时重写此方法

基类 OperatorWithKernel 中的 GetExpectedKernelType 方法对于派生类 Op 的所有输入变量进行了完备的初始化检查，建议在新增的 Op 中直接使用基类的此方法，例如：

- [MeanOp](#): 该 Op 的所有输入变量在 Run 之前应该全部被初始化，初始化检查是必要且合理的

但是在一些情况下，直接使用基类的 GetExpectedKernelType 方法无法满足需求，则需要对该方法进行重写，具体情况及示例如下：

1. OP 的输入有多个，且数据类型不同，例如 [AccuracyOp](#)，需要重写 GetExpectedKernelType 方法，指定用某一输入变量获取 kernel 类型
2. Op 包含 Disposable 的输入变量，该类输入变量是可选的，当用户未输入时，该类变量未被初始化属于合理情况，例如 [ConvOp](#)，存在 Bias 等可选的输入变量，需要重写 GetExpectedKernelType 方法，指定用必须提供的输入变量获取 kernel 类型
3. Op 的部分输入变量即使未被初始化也属于合理情况，例如 [ConcatOp](#)，输入变量 X 中有个 Tensor 需要连接，其中可能包含未被初始化的 Tensor，需要重写 GetExpectedKernelType 方法，使用输入变量 X 获得 kernel 的过程中，合理忽略掉部分 Tensor 为空的情况
4. OP 的 Kernel 类型与输入变量无关（可能由其他参数指定），例如 [FillOp](#)，该 Op 没有输入，Kernel 类型通过 Op 的 dtype 参数指定，因此需要重写 GetExpectedKernelType 方法，用参数指定的数据类型获取 kernel 类型
5. Op Kernel 的部分参数在使用某些库时，需要指定为相应的值，因此需要重写 GetExpectedKernelType 方法，覆盖默认参数
 - 使用 CUDNN 库：需要指定 OpKernel 的 LibraryType 为 kCUDNN，例如 [AffineGridOp](#)
 - 使用 MKLDNN 库：需要指定 OpKernel 的 LibraryType 和 DataLayout 为 kMKLDNN [MulOp](#)

4.2 重写此方法时需要对输入变量进行初始化检查

在需要重写 GetExpectedKernelType 方法时，一般会根据某一输入变量获取 Kernel 的数据类型，此时请使用 OperatorWithKernel::IndicateVarDataType 接口获取变量的 dtype，该方法对指定的输入变量进行了必要的初始化检查，详见[Paddle PR #20044](#)，实现示例如下，：

```
framework::OpKernelType GetExpectedKernelType(
    const framework::ExecutionContext& ctx) const override {
  return framework::OpKernelType(
    OperatorWithKernel::IndicateVarDataType(ctx, "X"), ctx.GetPlace());
}
```

如果未使用带有初始化检查的方法，直接使用了 `Tensor->type()`，可能会导致报出 `holder_ should not be null. Tensor not initialized yet when Tensor::type()` 的错误，例如[Paddle issue #19522](#)，用户仅凭该错误信息将无法得知具体出错的 Op，不利于调试。

5.Op 兼容性问题

对 Op 的修改需要考虑兼容性问题，要保证 Op 修改之后，之前的模型都能够正常加载及运行，即新版本的 Paddle 预测库能成功加载运行旧版本训练的模型。所以，需要保证 Op 的 **Input**、**Output** 和 **Attribute** 不能被修改（文档除外）或删除，可以新增 **Input**、**Output** 和 **Attribute**，但是新增的 **Input**、**Output** 必须设置 **AsDisposable**，新增的 **Attribute** 必须设置默认值。更多详细内容请参考OP 修改规范: [Input/Output/Attribute 只能做兼容修改](#)。

6.ShareDataWith 的调用

ShareDataWith 的功能是使两个 Tensor 共享底层 buffer，在调用这个操作的时候需要特别注意，在 Op 内部不能将 ShareDataWith 作用在 Op 的输出上，即 Op 输出的 Tensor 必须是 Malloc 出来的。

7. 稀疏梯度参数更新方法

目前稀疏梯度在做更新的时候会先对梯度做 merge，即对相同参数的梯度做累加，然后做参数以及附加参数（如 velocity）的更新。

8. 显存优化

8.1 为可原位计算的 Op 注册 Inplace

有些 Op 的计算逻辑中，输出可以复用输入的显存空间，也可称为原位计算。例如 `reshape_op` 中，输出 `Out` 可以复用输入 `X` 的显存空间，因为该 Op 的计算逻辑不会改变 `X` 的实际数据，只是修改它的 `shape`，输出和输入复用同一块显存空间不影响结果。对于这类 OP，可以注册 `Inlace`，从而让框架在运行时自动地进行显存优化。

Paddle 提供了 `DECLARE_INPLACE_OP_INFERENCE` 宏用于注册 `Inplace`，该宏第一个参数是一个类名，如 `ReshapeOpInplaceInToOut`；第二个参数是一对复用的输入输出，以 `{"X", "Out"}` 的形式给出。在 `REGISTER_OPERATOR` 时，可以将类名传入，从而为该 Op 注册 `Inplace`。

```
DECLARE_INPLACE_OP_INFERENCE(ReshapeOpInplaceInToOut, {"X", "Out"});  
  
REGISTER_OPERATOR(  
    reshape, ops::ReshapeOp, ops::ReshapeOpMaker,  
    paddle::framework::DefaultGradOpMaker<paddle::framework::OpDesc, true>,  
    paddle::framework::DefaultGradOpMaker<paddle::imperative::OpBase, true>,  
    ops::ReshapeOpInplaceInToOut);
```

8.2 减少 OP 中的无关变量

通常反向 Op 会依赖于前向 Op 的某些输入 (Input)、输出 (Output)，以供反向 Op 计算使用。但有些情况下，反向 Op 不需要前向 Op 的所有输入和输出；有些情况下，反向 Op 只需要前向 Op 的部分输入和输出；有些情况下，反向 Op 只需要使用前向 Op 中输入和输出变量的 Shape 和 LoD 信息。若 Op 开发者在注册反向 Op 时，将不必要的前向 Op 输入和输出作为反向 Op 的输入，会导致这部分显存无法被框架现有的显存优化策略优化，从而导致模型显存占用过高。

所以在写注册反向 Op 时需要注意以下几点：

- Paddle 提供的 DefaultGradOpMaker，默认会将前向 op 的所有输入 (Input)、输出 (Output) 以及输出变量所对应的梯度 (Output@Grad) 作为反向 Op 的输入，将前向 Op 输入所对应的梯度 (Input@Grad) 作为反向 Op 的输出。所以在使用 DefaultGradOpMaker 时需要考虑是否有些变量在计算中不被用到。
- 如果 DefaultGradOpMaker 不能够满足需求，需要用户自己手动构建 GradOpMaker，具体实现请参考[相关文档](#)；
- 如果有些反向 Op 需要依赖前向 Op 的输入或输出变量的的 Shape 或 LoD，但不依赖于变量中 Tensor 的 Buffer，且不能根据其他变量推断出该 Shape 和 LoD，则可以通过 DECLARE_NO_NEED_BUFFER_VARS_INFERER 接口对该变量（以下称该变量为 x）在反向 Op 中进行注册 NoNeedBufferVars。一旦注册了 NoNeedBufferVars，反向 op 中就不能读写该变量对应的 Tensor 中的 buffer，只能调用 Tensor 的 dims() 和 lod() 方法，同时，反向 Op 中的 GetExpectedKernelType() 必须要重写，并且 GetExpectedKernelType() 中不能访问 x 变量中 Tensor 的 type() 方法。比如在 SliceOpGrad 中只会用到 Input 中变量的 Shape 信息，所以需要为对 Input 在 SliceOpGrad 上进行注册：

```
namespace paddle {
namespace operators {
// ...
class SliceOpGrad : public framework::OperatorWithKernel {
public:
    using framework::OperatorWithKernel::OperatorWithKernel;

    void InferShape(framework::InferShapeContext* ctx) const override {
        // ...
    }

    framework::OpKernelType GetExpectedKernelType(
        const framework::ExecutionContext& ctx) const override {
        // Note: don't get data type from ctx.Input<framework::Tensor>("Input");
        auto dtype = ctx.Input<framework::Tensor>(framework::GradVarName("Out"))->type();
        return framework::OpKernelType( dtype, ctx.GetPlace());
    }
};

}
```

(下页继续)

(续上页)

```

template <typename T>
class SliceOpGradMaker : public framework::SingleGradOpMaker<T> {
public:
    using framework::SingleGradOpMaker<T>::SingleGradOpMaker;

protected:
    void Apply(GradOpPtr<T> bind) const override {
        bind->SetInput("Input", this->Input("Input"));
        if (this->HasInput("StartsTensor")) {
            bind->SetInput("StartsTensor", this->Input("StartsTensor"));
        }
        if (this->HasInput("EndsTensor")) {
            bind->SetInput("EndsTensor", this->Input("EndsTensor"));
        }
        if (this->HasInput("StartsTensorList")) {
            bind->SetInput("StartsTensorList", this->Input("StartsTensorList"));
        }
        if (this->HasInput("EndsTensorList")) {
            bind->SetInput("EndsTensorList", this->Input("EndsTensorList"));
        }
        bind->SetInput(framework::GradVarName("Out"), this->OutputGrad("Out"));
        bind->SetOutput(framework::GradVarName("Input"), this->InputGrad("Input"));
        bind->SetAttrMap(this->Attrs());
        bind->SetType("slice_grad");
    }
};

DECLARE_NO_NEED_BUFFER_VARS_INFERENCE(SliceOpGradNoNeedBufferVarsInference,
                                      "Input");
} // namespace operators
} // namespace paddle
namespace ops = paddle::operators;
REGISTER_OPERATOR(slice, ops::SliceOp, ops::SliceOpMaker,
                 ops::SliceOpGradMaker<paddle::framework::OpDesc>,
                 ops::SliceOpGradMaker<paddle::imperative::OpBase>);
REGISTER_OPERATOR(slice_grad, ops::SliceOpGrad,
                 ops::SliceDoubleOpGradMaker<paddle::framework::OpDesc>,
                 ops::SliceDoubleOpGradMaker<paddle::imperative::OpBase>,
                 ops::SliceOpGradNoNeedBufferVarsInference);

```

9. 混合设备调用

由于 GPU 是异步执行的，当 CPU 调用返回之后，GPU 端可能还没有真正的执行，所以如果在 Op 中创建了 GPU 运行时需要用到的临时变量，当 GPU 开始运行的时候，该临时变量可能在 CPU 端已经被释放，这样可能会导致 GPU 计算出错。

关于 GPU 中的一些同步和异步操作：

```
The following device operations are asynchronous with respect to the host:
    Kernel launches;
    Memory copies within a single device's memory;
    Memory copies from host to device of a memory block of 64 KB or less;
    Memory copies performed by functions that are suffixed with Async;
    Memory set function calls.
```

关于 cudaMemcpy 和 cudaMemcpyAsync 注意事项：

- 如果数据传输是从 GPU 端到非页锁定的 CPU 端，数据传输将是同步，即使调用的是异步拷贝操作。
- 如果数据传输是从 CPU 端到 CPU 端，数据传输将是同步的，即使调用的是异步拷贝操作。

更多内容可参考：[Asynchronous Concurrent Execution, API synchronization behavior](#)

10. LoD 在 Op 内部的传导规范

LoD 是 Paddle 框架用来表示变长序列数据的属性，除了仅支持输入是 padding data 的 Op 外，所有 Op 的实现都要考虑 LoD 的传导问题。

根据 OP 的计算过程中是否用到 LoD，我们可以将涉及到 LoD 传导问题的 OP 分为两类：LoD-Transparent 与 LoD-Based。

这两类 OP 的 LoD 传导需要考虑前向和反向两个过程。

前向传导

在前向传导过程，与输入的 LoD 相比较，Op 输出的 LoD 可能出现不变、改变和消失这三种情况：

- 不变：适用于所有的 LoD-Transparent OP 与部分的 LoD-Based OP。可以在 InferShape 中调用 ShareLoD() 直接将输入 Var 的 LoD 共享给输出 Var，可参考 [lstm_op](#)；如果有多个输入且都可能存在 LoD 的情况，通常默认共享第一个输入，例如 [elementwise_ops forward](#)；
- 改变：适用于部分 LoD-Based OP。在实现 OpKernel 时需考虑输出 LoD 的正确计算，真实的 LoD 在前向计算结束后才能确定，此时仍需要在 InferShape 中调用 ShareLoD()，以确保 CompileTime 时对 LoD Level 做了正确的传导，可参考 [sequence_expand_op](#)；
- 消失：适用于输出不再是序列数据的 LoD-Based OP。此时不用再考虑前向的 LoD 传导问题，可参考 [sequence_pool_op](#)；

其它重要的注意事项：

- 实现 LoD-Based OP 时，需要处理好 LoD 传导的边界情况，例如对长度为零的输入的支持，并完善相应的单测，单测 case 覆盖空序列出现在 batch 开头、中间和末尾等位置的情况，可参考 `test_lstm_op.py`
- 对 LoD Level 有明确要求的 OP，推荐的做法是在 `InferShape` 中即完成 LoD Level 的检查，例如 `sequence_pad_op`。

反向传导

通常来讲，OP 的某个输入 Var 所对应的梯度 GradVar 的 LoD 应该与 Var 自身相同，所以应直接将 Var 的 LoD 共享给 GradVar，可以参考 `elementwise ops` 的 `backward`

Op 性能优化

1. 第三方库的选择

在写 Op 过程中优先使用高性能（如 `cudnn`、`mkldnn`、`mklml`、`eigen` 等）中提供的操作，但是一定要做 benchmark，有些库中的操作在深度学习任务中可能会比较慢。因为高性能库（如 `eigen` 等）中提供的操作为了更为通用，在性能方面可能并不是很好，通常深度学习模型中数据量较小，所以有些情况下可能高性能库中提供的某些操作速度较慢。比如 Elementwise 系列的所有 Op（前向和反向），Elementwise 操作在模型中调用的次数比较多，尤其是 `Elementwise_add`，在很多操作之后都需要添加偏置项。在之前的实现中 `Elementwise_op` 直接调用 `Eigen` 库，由于 Elementwise 操作在很多情况下需要对数据做 Broadcast，而实验发现 `Eigen` 库做 Broadcast 的速度比较慢，慢的原因在这个 PR#6229 中有描述。

2. Op 性能优化

Op 的计算速度与输入的数据量有关，对于某些 Op 可以根据输入数据的 Shape 和 Op 的属性参数来选择不同的计算方式。比如 `concat_op`，当 `axis>=1` 时，在对多个 tensor 做拼接过程中需要对每个 tensor 做很多次拷贝，如果是在 GPU 上，需要调用 `cudaMemcpy`。相对 CPU 而言，GPU 属于外部设备，所以每次调用 GPU 的操作都会有一定的额外开销，并且当需要拷贝的次数较多时，这种开销就更为凸现。目前 `concat_op` 的实现会根据输入数据的 Shape 以及 axis 值来选择不同的调用方式，如果输入的 tensor 较多，且 axis 不等于 0，则将多次拷贝操作转换成一个 CUDA Kernel 来完成；如果输入 tensor 较少，且 axis 等于 0，使用直接进行拷贝。相关实验过程在该 PR (#8669) 中有介绍。

由于 CUDA Kernel 的调用有一定的额外开销，所以如果 Op 中出现多次调用 CUDA Kernel，可能会影响 Op 的执行速度。比如之前的 `sequence_expand_op` 中包含很多 CUDA Kernel，通常这些 CUDA Kernel 处理的数据量较小，所以频繁调用这样的 Kernel 会影响 Op 的计算速度，这种情况下最好将这些小的 CUDA Kernel 合并成一个。在优化 `sequence_expand_op` 过程（相关 PR#9289）中就是采用这种思路，优化后的 `sequence_expand_op` 比之前的实现平均快出约 1 倍左右，相关实验细节在该 PR (#9289) 中有介绍。

减少 CPU 与 GPU 之间的拷贝和同步操作的次数。比如 `fetch` 操作，在每个迭代之后都会对模型参数进行更新并得到一个 loss，并且数据从 GPU 端到没有页锁定的 CPU 端的拷贝是同步的，所以频繁的 `fetch` 多个参数

会导致模型训练速度变慢。

Op 数值稳定性问题

1. 有些 Op 存在数值稳定性问题

出现数值稳定性的主要原因是程序在多次运行时，对浮点型数据施加操作的顺序可能不同，进而导致最终计算结果不同。而 GPU 是通过多线程并行计算的方式来加速计算的，所以很容易出现对浮点数施加操作的顺序不固定现象。

目前发现 cudnn 中的卷积操作、cudnn 中的 MaxPooling、CUDA 中 CudaAtomicXX、ParallelExecutor 的 Reduce 模式下参数梯度的聚合等操作运行结果是非确定的。

为此 Paddle 中添加了一些 FLAGS，比如使用 FLAGS_cudnn_deterministic 来强制 cudnn 使用确定性算法、FLAGS_cpu_deterministic 强制 CPU 端的计算使用确定性方法。

其他

1. 报错信息

Enforce 提示信息不能为空，并且需要写明，因为报错信息可以更快更方便地分析出错误的原因。

2.Op 的数学公式

如果 Op 有数学公式，一定要在代码中将数学公式写明，并在 Python API 的 Doc 中显示，因为用户在对比不同框架的计算结果时可能需要了解 Paddle 对 Op 是怎么实现的。

** 注意： ** 在 merge 到 develop 分支之前一定进行公式预览。可参考[dynamic_lstm](#)。

3.Op 变量名的命名要规范

在定义 Op 时，Op 的输入输出以及属性的命名需要符合规范，具体命名规则请参考：[name_convention](#)。

4.Python 端 Op 接口中参数的顺序

Python API 中参数的顺序一般按照重要性来排，以 fc 为例：

```
def fc(input,
       size,
       num_flatten_dims=1,
       param_attr=None,
       bias_attr=None,
```

(下页继续)

(续上页)

```
act=None,
is_test=False,
name=None)
```

飞桨 API 文档书写规范

1. 至关重要: API 文档对该 API 的描述, 一定要与 API 的行为保持一致。中英文文档的内容要严格一致。
2. API 文档的字段: API 名称、API 功能描述、API 参数、API 返回、API 代码示例、API 属性 (class)、API 方法 (methods) 等。是否写 API 抛出异常的情况, 不做强制要求。
3. API 功能描述: 请注意, 看文档的用户没有和开发同学一样的知识背景。因此, 请提示用户在什么场景下使用该 API。请使用深度学习领域通用的词汇和说法。(深度学习常用术语表)。
4. API 参数: 写清楚对输入参数的要求, 写清楚在不同情况下的行为区别 (例默认值时的行为)。同类型参数 (如: 输入 tensor x, 每个 API 中的 name 参数), 可以直接从这里 copy 内容: 常用文档写法。
5. API 代码示例: 中英文文档当中的代码示例完全一致 (means identical, comments 可不用翻译)。代码示例使用 2.0 版本中的 API, 可运行。尽量不用 random 输入, 注释形式给出输出值。
6. 其他: 2.0 中的 API, 对于 Variable、LodTensor、Tensor, 统一使用 Tensor.to_variable 也统一改为 to_tensor。
7. 构造输入数据时, 尽量使用 paddle 提供的 API, 如:paddle.zeros, paddle.ones, paddle.full, paddle.arange, paddle.rand, paddle.randn, paddle.randint, paddle.normal, paddle.uniform。
8. 对于 Linear, Conv2D, L1Loss 这些 class 形式的 API, 需要写清楚当这个 callable 被调用时的输入输出的形状。(i.e.forward 函数的参数)。位置放在现在的 Parameters/参数这个 block 后面, 具体为:

中文时:

形状:

- input: 形状为 (批大小, 通道数, 高度, 宽度), 即, HCHW 格式的 4-D Tensor。
- output: 形状为 (批大小, 卷积核个数, 输出图像的高度, 输出图像的高度) 的 4-D Tensor。

英文时:

Shape:

- input: 4-D tensor with shape: (batch, num_channels, height, width), i.e.: \hookrightarrow HCHW.
- output: 4-D tensor with shape: (batch, num_filters, new_height, new_width).

典型案例

```
paddle.concat paddle.split paddle.squeeze paddle.full_like paddle.ones paddle.
ones_like
```

英文模板

```
def add(x, y, name=None):
    """
    Add two tensors element-wise. The equation is:

    .. math::
        \text{out} = \text{x} + \text{y}

    **Note**:
    ``paddle.add`` supports broadcasting. If you want know more about broadcasting,
    →please refer to :ref:`user_guide_broadcasting` .

    Args:
        x (Tensor): the input tensor, it's data type should be float32, float64,
        ↪int32, int64.
        y (Tensor): the input tensor, it's data type should be float32, float64,
        ↪int32, int64.
        name (str, optional): Name for the operation (optional, default is None). For
        →more information, please refer to :ref:`api_guide_Name` .

    Returns:
        N-D Tensor. A location into which the result is stored. It's dimension
        →equals with $x$.

    Examples:
        .. code-block:: python

            import paddle
            import numpy as np

            paddle.enable_imperative()
            np_x = np.array([2, 3, 4]).astype('float64')
            np_y = np.array([1, 5, 2]).astype('float64')
            x = paddle.imperative.to_variable(np_x)
            y = paddle.imperative.to_variable(np_y)
```

(下页继续)

(续上页)

```

z = paddle.add(x, y)
np_z = z.numpy()
# [3., 8., 6.]

z = paddle.add(x, y, alpha=10)
np_z = z.numpy()
# [12., 53., 24.]
"""

```

中文模板

```
.. _cn_api_tensor_add:
```

```
add
```

```
.. py:function:: paddle.add(x, y, name=None)
```

该OP是逐元素相加算子，输入 ``x`` 与输入 ``y`` -|

→逐元素相加，并将各个位置的输出元素保存到返回结果中。计算公式为：

```
.. math::
```

```
out = x + y
```

```
.. note::
```

``paddle.add`` 遵守broadcasting，如您想了解更多，请参见 :ref:`cn_user_guide_broadcasting`。

```
..
```

```
-|
```

→说明：以上为API描述部分，只需要尽可能简单的描述出API的功能作用即可，要让用户能快速看懂。这个case可+|计算公式 + 注解部分；

参数

```
:::::::
```

- x (Tensor) - 输入的Tensor，数据类型为：float32、float64、int32、int64。
- y (Tensor) - 输入的Tensor，数据类型为：float32、float64、int32、int64。
- name (str, 可选) - 操作的名称(可选，默认值为None)。更多信息请参见 :ref:`api_guide_Name`。

```
..
```

```
-|
```

→说明：API参数可优先copy常用文档写法中的参数，参数的描述要准确，还要重点描述参数的功能作用及使用场景

返回

(下页继续)

(续上页)

```
::::::::::
``Tensor``，维度和数据类型都与 ``x`` 相同，存储运算后的结果。
..
    返回为 返回类型 + 描述的格式即可。
```

代码示例

```
::::::::::
```

```
.. code-block:: python

    import paddle
    import numpy as np

    paddle.enable_imperative()
    np_x = np.array([2, 3, 4]).astype('float64')
    np_y = np.array([1, 5, 2]).astype('float64')
    x = paddle.imperative.to_variable(np_x)
    y = paddle.imperative.to_variable(np_y)

    z = paddle.add(x, y)
    np_z = z.numpy()
    # [3., 8., 6.]

    z = paddle.add(x, y, alpha=10)
    np_z = z.numpy()
    # [12., 53., 24.]

..
尽量不使用random的输入；
优先使用动态图，在一个代码示例中给出多个使用场景；
```

API 文档各模块写作说明

API 名称

API 名称直接写 API 的名字即可，不需要将全路径写全；

** 如 paddle.add **

```
add
-----
```

API 声明

API 的声明部分，要给出 API 的声明信息；

function: 如 paddle.add

```
.. py:function:: paddle.add(x, y, name=None)
```

class: 如 paddle.nn.Conv2d

```
.. py:class:: paddle.nn.Conv2d(num_channels, num_filters, filter_size, padding=0,_
← stride=1, dilation=1, groups=None, param_attr=None, bias_attr=None, use_cudnn=True,_
← act=None, name=None, data_format="NCHW", dtype="float32")
```

API 功能描述

API 功能描述部分只需要尽可能简单的描述出 API 的功能作用即可，要让用户能快速看懂。如 paddle.add[https://www.paddlepaddle.org.cn/documentation/docs/zh/api/paddle/tensor/math/add_cn.html#add]：

可以拆解为 3 个部分，功能作用 + 计算公式 + 注解部分，其中：

- 功能作用：描述该 API 文档的功能作用；由于用户没有对应的背景，所以需要补充必要的细节，比如是不是 element_wise 的，如 paddle.add：

该OP是逐元素相加算子，输入 ``x`` 与输入 ``y``
→逐元素相加，并将各个位置的输出元素保存到返回结果中。

- 计算公式：给出该 API 的计算公式，由于公式中每个变量都对应 API 的参数，所以不需要做额外的说明，如 paddle.add：

计算公式为：

```
.. math::  
    \text{out} = \text{x} + \text{y}
```

- 注解部分：加入 API 有需要特殊说明的部分，可以在注解部分给出，比如：该 API 与其他 API 功能相似，需要给出该 API 与另一个 API 的使用上的区别。

注意事项：

1. 写作 API 文档中，请使用深度学习领域通用的词汇和说法。（[深度学习常用术语表](#)）。
2. 文档中的前后说明要一致，比如维度的说明，统一使用 4-D Tensor 的格式，不确定的写“多维”，示例如下：
3. 文档相互引用的方式：[如何让文档相互引用](#)

4. 功能描述中涉及到的专有数据结构如 Tensor、LoDTensor 或 Variable，中英文都统一使用 Tensor 无需翻译。

5. 如果涉及到一些通用的知识，如 broadcasting，可以以 Note 的方式写出来，示例如下：

中文：

```
.. note::
    ``paddle.add`` 遵守broadcasting, 如您想了解更多, 请参见 :ref:`cn_user_guide_broadcasting` .
```

英文：

```
**Note**:
``paddle.add`` supports broadcasting. If you want know more about broadcasting,
please refer to :ref:`user_guide_broadcasting` .
```

总结：paddle.add 的描述如下

该OP是逐元素相加算子，输入 ``x`` 与输入 ``y``
 →逐元素相加，并将各个位置的输出元素保存到返回结果中。计算公式为：

```
.. math::
    \text{out} = \text{x} + \text{y}
.. note::
    ``paddle.add`` 遵守broadcasting, 如您想了解更多, 请参见 :ref:`cn_user_guide_broadcasting` .
```

API 参数（重要）

注意：一些通用的参数说明，直接 copy 常用文档写法中的描述即可。

API 参数部分，要解释清楚每个参数的意义和使用场景。需要注意以下两点：

1、对于有默认值的参数，至少要讲清楚在默认值下的逻辑，而不仅仅是介绍这个参数是什么以及默认值是什么；

如 stop_gradient 的对比，要添加默认值为 True 的行为，即表示停止计算梯度。

```
stop_gradient (bool, 可选) - 提示是否应该停止计算梯度，默认值为False。
# wrong

stop_gradient (bool, 可选) -
    →提示是否应该停止计算梯度，默认值为True，表示停止计算梯度。
# right
```

或者如 `return_numpy` 的对比：

```
return_numpy (bool) - 该变量表示是否将 fetched tensor 转换为 numpy。默认为：True。
# wrong

return_numpy (bool) - 该参数表示是否将返回的计算结果（fetch_
↳ list 中指定的变量）转化为 numpy；如果为 False，则每个变量返回的类型为 Tensor，否则返回变量的类型为 numpy_
↳ ndarray。默认为：True。
# right
```

可以看出，第二行的 `return_numpy` 的描述更为清晰，分别描述了 True 和 False 的两种情况。而第一行的说明过于简单。

2、在讲清楚每个 API 参数是什么的同时，还需要描述清楚每个参数的具体作用是什么。如再如 `feeded_var_names`：

```
feeded_var_names (list[str]) - 字符串列表，包含着 Inference_
↳ Program 预测时所需提供数据的所有变量名称（即所有输入变量的名称）。
target_vars (list[Variable]) - Variable（详见 基础概念_）
↳ 类型列表，包含着模型的所有输出变量。通过这些输出变量即可得到模型的预测结果。
```

用户看了描述以后，完全不知道这两个参数可以用来做网络裁剪，所以需要将一些参数的功能作用描述出来。

总结：`paddle.add`：

参数
::::::::::
<ul style="list-style-type: none"> - x (Tensor) - 输入的 Tensor，数据类型为：float32、float64、int32、int64。 - y (Tensor) - 输入的 Tensor，数据类型为：float32、float64、int32、int64。 - name (str, 可选) - 操作的名称(可选，默认值为None)。更多信息请参见 :ref:`api_ ↳ guide_Name`。

API 返回

先描述 API 返回值的类型，然后描述 API 的返回值及其含义。

如 `paddle.add`

返回
::::::::::
``Tensor``，维度和数据类型都与 ``x`` 相同，存储运算后的结果。

API 抛出异常

API 抛出异常部分，不做强制要求，可以在一些特殊的场景下给出抛出异常的信息，如：

场景 1： API 使用有先后依赖关系如：paddle.fluid.layers.DynamicRNN.update_memory()框中的部分：

场景 2： 多个参数相互影响如paddle.layers.pool2d框出的部分：

场景 3： 该 API 的使用场景有特殊限制如paddle.fluid.layers.DynamicRNN.step_input()框出的部分：

API 代码示例（重要）

代码示例是 API 文档的核心部分之一，毕竟 talk is cheap, show me the code。所以，在 API 代码示例中，应该对前文描述的 API 使用中的各种场景，尽可能的在一个示例中给出，并用注释给出对应的结果。如

注意事项

- 示例代码需要与当前版本及推荐用法保持一致：**develop 分支下 fluid namespace 以外的 API，不能再有 fluid 关键字，只需要提供动态图的示例代码。**
- 尽量不用 random 输入，需要以注释形式给出输出值。
- 中英文示例代码，不做任何翻译，保持相同（means identical）。
- 原则上，所有提供的 API 都需要提供示例代码，对于 class member methods, abstract API, callback, 等情况，可以在提交 PR 时说明相应的使用方法的文档的位置或文档计划后，通过白名单审核机制通过 CI 检查。
- 对于仅为 GPU 环境提供的 API，当该示例代码在 CPU 上运行时，运行后给出含有"Not compiled with CUDA"的错误提示，也可认为该 API 行为正确。

英文 API 代码示例格式规范如下：

```
def api():
    """
    Examples:
        .. code-block:: python

            示例代码位置

        .. code-block:: python

            示例代码位置 (存在多个示例代码)
    """

```

英文格式如 paddle.multiply:

```
def multiply(x, y, axis=-1, name=None):
    """
    Examples:
        .. code-block:: python

            import paddle

            x = paddle.to_tensor([[1, 2], [3, 4]])
            y = paddle.to_tensor([[5, 6], [7, 8]])
            res = paddle.multiply(x, y)
            print(res) # [[5, 12], [21, 32]]

            x = paddle.to_tensor([[1, 2, 3], [1, 2, 3]])
            y = paddle.to_tensor([2])
            res = paddle.multiply(x, y)
            print(res) # [[2, 4, 6], [2, 4, 6]]
```

中文格式如 paddle.add:

代码示例

```
.. code-block:: python

    import paddle
    x = paddle.to_tensor([2, 3, 4], 'float64')
    y = paddle.to_tensor([1, 5, 2], 'float64')
    z = paddle.add(x, y)
    print(z) # [3., 8., 6.]
```

API 属性

API 的属性用来描述 API 所包含的属性。如果 API 有属性，每个属性需要分为四个部分描述：

- 名称：属性名称直接写属性的名字即可，不需要将全路径写全；
- 注意：列举出使用该属性时应注意的一些问题，如果没有可以不填；如不同的版本、是否是只读属性、使用的一些 tricks 等等，如 Program 的 rand_seed：

```
.. note::
    必须在相关OP被添加之前设置。
```

- 描述：API 功能描述部分要求一致；
- 返回：API 返回部分要求一致；

- 代码示例：与 API 代码示例部分要求一致；

总结：**paddle.Program.random_seed**

```
random_seed
.....
..note:
必须在相关OP被添加之前设置。

程序中随机运算符的默认随机种子。0意味着随机生成随机种子。

**返回**
int64，返回该Program中当前正在使用的random seed。。

**代码示例**
.. code-block:: python

    import paddle
    prog = paddle.default_main_program()
    random_seed = prog.random_seed
    x_var = paddle.data(name="X", shape=[3,3], dtype="float32")
    print(random_seed)
    # 0
    # default random seed is 0
    # Here we need to set random seed before we use dropout
    prog.random_seed = 1
    z_var = paddle.nn.functional.dropout(x_var, 0.7)
    print(prog.random_seed)
    # 1
    # the random seed is change to 1
```

API 方法

API 的方法用来描述 API 所包含的方法，一些类的 API 会有这个内容，没有方法的 API 可以不写此模块。如果有，每个方法需要分为六个部分描述：

- 名称：方法名称直接写方法的名字即可，不需要将全路径写全；
- 声明：与 API 声明的要求一致。
- 参数：与 API 参数的要求一致。
- 描述：与 API 功能描述的要求一致。
- 返回：与 API 返回的要求一致。。

- 代码示例：与 API 代码示例的要求一致。

总结：**paddle.Program.parse_from_string**

```
parse_from_string
''

.. py:function:: paddle.Program.parse_from_string(binary_str_type)
    通过对 protobuf 的反序列化，转换成 ``Program``

    **参数**
    binary_str_type (**str**) - protobuf 二进制字符串

    **返回**
    ``Program``，反序列化后的 ``Program``

    **代码示例**
    .. code-block:: python

        import paddle

        startup_prog = paddle.Program()
        main_prog = paddle.Program()
        with paddle.program_guard(startup_prog, main_prog):
            x = paddle.data(name='X', shape=[1000, 784], dtype='float32')
            y = paddle.data(name='Y', shape=[784, 100], dtype='float32')
            z = paddle.mul(x=x, y=y)

        binary_str = paddle.default_main_program().desc.serialize_to_string()
        prog_restored = paddle.default_main_program().parse_from_string(
            binary_
            ↪ str)

        print(paddle.default_main_program())
        print(prog_restored)

        # The two Programs printed here should be same
```

注解

注解部分描述一些用户使用该 API 时需要额外注意的一些注意事项，可以出现在任意需要提示用户的地方；可以是当前版本存在的一些问题，如例 1；也可以是该 API 使用上的一些注意事项，如例 2；

例 1 paddle.fluid.cuda.cuda_places: 中文：

```
.. note::  
多卡任务请先使用 `FLAGS_selected_gpus` 环境变量设置可见的GPU设备，下个版本将会修正 `CUDA_`  
`VISIBLE_DEVICES` 环境变量无效的问题。
```

英文：

```
**Note**:  
For multi-card tasks, please use `FLAGS_selected_gpus` environment variable to set  
the visible GPU device.  
The next version will fix the problem with `CUDA_VISIBLE_DEVICES` environment  
variable.
```

例 2 paddle.sqrt 中文：

```
.. note::  
请确保输入中的数值是非负数。
```

英文：

```
**Note**:  
input value must be greater than or equal to zero.
```

警告

警告部分需要慎重使用，一般是不推荐用户使用的 API 方法（例 1），或者是已经有计划要废弃的 API（例 2），需要用警告来说明。

例 1 paddle.fluid.layers.DynamicRNN 中文：

```
.. warning::  
目前不支持在DynamicRNN的 block 中任何层上配置 is_sparse = True。
```

英文：

```
Warning:  
Currently it is not supported to set :code:`is_sparse = True` of any  
layers defined within DynamicRNN's :code:`block` function.
```

例 2 paddle.fluid.clip.set_gradient_clip 中文：

```
.. warning::
```

此 API 对位置使用的要求较高，其必须位于组建网络之后，``minimize``
 ↪之前，因此在未来版本中可能被删除，故不推荐使用。推荐在 ``optimizer``
 ↪初始化时设置梯度裁剪。有三种裁剪策略：``GradientClipByGlobalNorm``、
 ↪``GradientClipByNorm``、``GradientClipByValue``。如果在 ``optimizer``
 ↪中设置过梯度裁剪，又使用了 ``set_gradient_clip``，``set_gradient_clip``
 ↪将不会生效。

英文：

Warning:

```
This API must be used after building network, and before ``minimize``,  

and it may be removed in future releases, so it is not recommended.  

It is recommended to set ``grad_clip`` when initializing the ``optimizer``,  

this is a better method to clip gradient. There are three clipping strategies:  

:ref:`api_fluid_clip_GradientClipByGlobalNorm` , :ref:`api_fluid_clip_`  

↪GradientClipByNorm` ,  

:ref:`api_fluid_clip_GradientClipByValue` .
```

新增 API 测试及验收规范

CI 通过性

进入 PaddlePaddle 主库的代码，涉及到的相关检测 CI 必须全部 Pass。用来验证对之前功能点的兼容和影响，用来保障新合入代码对历史代码不产生影响。

新增代码必须要有相应的单测保障测试覆盖率达到准入要求（测试覆盖率（行覆盖率）90%）。

PR 内容描述要求

单元测试内容需要和开发代码放在同一个 PR 提交，后续修改也需要基于此 PR。

PR 内容描述测试部分需要明确指出测试相关的文件列表，并写明测试 Case 的运行方法和自测结果。

API 测试内容及单元测试要求

API 命名，参数命名，暴露方式，代码目录层级需按照设计文档要求保持一致或参考 API 通用设计文档的要求。

新增 API 测试代码必须覆盖动态图和静态图的测试场景。（原则上需要支持动态图和静态图两种方式调用，如果仅支持其中一种，需要在设计文档 RFC 和 API 文档中体现）

新增 API 测试代码必须覆盖 CPU 和 GPU 的测试场景。（原则上需要支持 CPU 和 GPU 两种硬件平台调用，如果仅支持其中一种，需要在设计文档 RFC 和 API 文档中体现）

新增 API 涉及 tensor 的 dtype 需要有相关 Case 进行测试覆盖。

新增 API 的入参，需要对全部入参进行参数有效性和边界值测试。确定每个入参都可以正确生效。

新增 API 的前向计算，需要对数值的绝对正确性进行验证，需要有通过 numpy 或其他数学方法实现的函数的对比结果。

新增 API 的反向计算，需要复用现有单测框架反向计算验证方式保障反向正确性。

- 使用 Python 组合方式新增的 API，由于反向计算已经在各组合 API 单测中分别验证了，因此，该 API 的反向计算不要求验证。
- 如现有单测框架无法满足要求，需要通过 numpy 推导或函数直接实现反向等方式验证反向计算结果正确性。

异常测试，对于参数异常值输入，应该有友好的报错信息及异常反馈，需要有相关测试 Case 验证。

文档测试

中文文档、英文文档齐全，内容一一对应。

文档清晰可读，易于用户使用

给出易于理解的 api 介绍，文字描述，公式描述。

参数命名通俗易懂无歧义，明确给出传参类型，对参数含义以及使用方法进行详细说明，对返回值进行详细说明。

异常类型和含义进行详细说明。

示例代码需要做到复制粘贴即可运行，并且需要明确给出预期运行结果（如果可以）。

阅读无障碍：无错别字、上下文连贯、内容清晰易懂、链接可正常跳转、图片公式显示正常。

交流与改进

提测代码的单测部分必须 paddlepaddle 测试人员 review，确保完整覆盖了待测功能点后，会给予 approved。如果 review 过程中发现测试缺失和遗漏的测试点，会通过 github 代码行 comment 的和 request changes 的方式交流改进，待 PR 修改完毕后给予 approved。

后续维护

代码成功 merge 后，如果发现对框架造成了严重影响，例如阻塞了某些方向的功能开发，或者和部分功能存在严重冲突导致 Bug，会对代码进行 Revert 并通知贡献者。待对 PR 修复后重新合入。

5.5 Kernel Primitive API

Title overline too short.

```
#####
Kernel Primitive API
#####
```

Kernel Primitive API 对算子 Kernel 实现中的底层代码进行了抽象与功能封装，提供高性能的 Block 级 IO 运算和 Compute 运算。使用 Kernel Primitive API 进行 Kernel 开发可以更加专注计算逻辑的实现，在保证性能的同时大幅减少代码量，同时实现了算子计算与硬件解耦。

本部分为 PaddlePaddle 高级开发人员提供了用于 Kernel 开发的 CUDA Kernel Primitive API，该类 API 能够帮助开发者在提升开发效率的同时收获较好的性能。Kernel Primitive API 主要包括 IO API、Compute API 以及 OpFunc，IO API 能够高效的完成全局内存与寄存器间的数据读写操作。Compute API 为通用计算函数，如 ElementwiseBinary、ElementwiseUnary 等；OpFunc 用于定义 Compute API 中的计算规则，例如实现 Add 操作则需要定义 AddFunctor 用于 ElementwiseBinary 调用，开发者可以直接使用默认的 OpFunc 也可以根据需要进行自定义，具体的实现规则将在 OpFunc 小节中进行详细介绍。当前 API 均是 Block 级别的多线程 API，开发者可以直接传入当前 Block 的数据指针以及操作类型完成相应的计算，目前仅支持全局数据指针和寄存器指针。

5.5.1 API 列表

API 名称	功能简介
ReadData	IO，将数据从全局内存读取到寄存器中。
ReadDataBc	IO，Broadcast 形式的数据读取，根据当前 Block 的数据偏移和原始输入指针计算输入坐标，并将输入数据读取到寄存器中。
ReadDataReduce	IO，Reduce 形式的数据读取，将需要进行规约的数据从全局内存读取到寄存器中。
WriteData	IO，将数据从寄存器写入全局内存中。
ElementwiseUnary	Compute API，输入与输出 Shape 相同的一元计算 API，根据 OpFunc 计算规则完成一元逐元素运算。
ElementwiseBinary	Compute API，输入与输出 Shape 相同的二元计算 API，根据 OpFunc 计算规则完成二元逐元素运算。
ElementwiseTernary	Compute API，输入与输出 Shape 相同的三元计算 API，根据 OpFunc 计算规则完成三元逐元素运算。
ElementwiseAny	Compute API，输入与输出 Shape 相同的多元计算 API，根据 OpFunc 计算规则完成多元逐元素运算。
CycleBinary	Compute API，input1 和 input2 具有不同 Shape，input2 和 output 具有相同 Shape，根据 OpFunc 计算规则完成二元循环操作。
Reduce	Compute API，根据 Reduce 模式完成数据规约。

5.5.2 OpFunc 列表

Functor 名称	功能简介
ExpFunctor	一元 Functor，对输入数据进行 Exp 操作。
IdentityFunctor	一元 Functor，对输入数据进行类型转换操作。
DivideFunctor	一元 Functor，对输入数据除以固定值。
SquareFunctor	一元 Functor，对数据进行 Square 操作。
MinFunctor	二元 Functor，返回输入中的最小值。
MaxFunctor	二元 Functor，返回输入中的最大值。
AddFunctor	二元 Functor，返回输入之和。
MulFunctor	二元 Functor，返回输入的乘积。
LogicalOrFunctor	二元 Functor，返回输入的逻辑或的操作结果。
LogicalAndFunctor	二元 Functor，返回输入的逻辑与的操作结果。
DivFunctor	二元 Functor，返回两数相除的结果。
FloorDivFunctor	二元 Functor，返回两数相除的结果。

5.5.3 API 介绍

- [IO API](#) : 介绍 IO 类 API 的定义和功能。
- [Compute API](#) : 介绍 Compute 类 API 的定义和功能。
- [OpFunc](#) : 介绍 Kernel Primitive API 提供的 Functor。

5.5.4 示例

- [ElementwiseAdd](#) : 加法操作，输入和输出具有相同 Shape。
- [Reduce](#) : 针对最高维进行规约操作。

API 介绍

- [IO API](#) : 介绍 IO 类 API 的定义和功能。
- [Compute API](#) : 介绍 Compute 类 API 的定义和功能。
- [OpFunc](#) : 介绍 Kernel Primitive API 提供的 Functor。

API 介绍 - IO

介绍目前 Kernel Primitive API 提供的用于全局内存和寄存器进行数据交换的 API。当前实现的 IO 类 API 均是 Block 级别的多线程 API，函数内部以 `blockDim.x` 或 `blockDim.y` 进行线程索引。

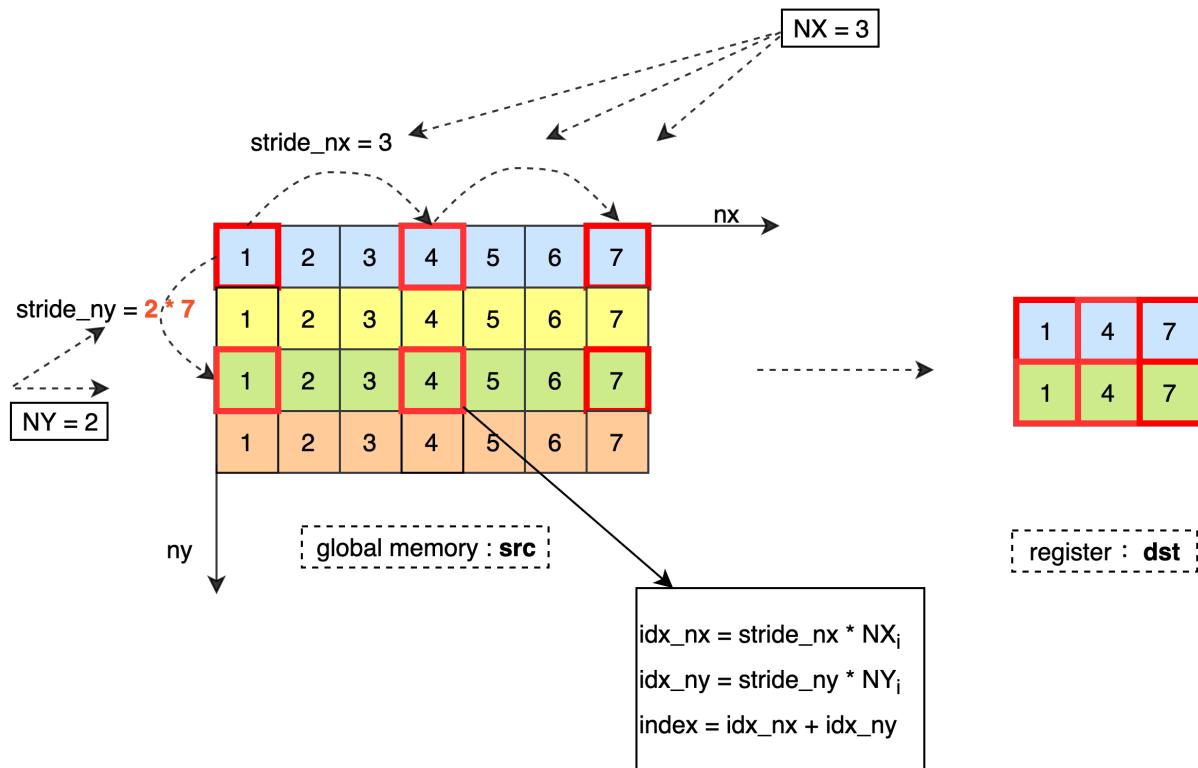
ReadData

函数定义

```
template <typename Tx, typename Ty, int NX, int NY, int BlockSize, bool IsBoundary = false>
__device__ void ReadData(Ty* dst, const Tx* src, int size_nx, int size_ny, int stride_nx, int stride_ny);
```

函数说明

将 `Tx` 类型的 2D 数据从全局内存中读取到寄存器，并按照 `Ty` 类型存储到寄存器 `dst` 中。最低维每读取 1 个元素需要偏移 `stride_nx` 个元素，最高维每读取 1 个元素需要偏移 `stride_ny` 个元素，直到加载 $NX * NY$ 个数据到寄存器 `dst` 中。当 `IsBoundary = true` 需要保证当前最高维偏移个数不超过 `size_ny`，列偏移个数不超过 `size_nx`。数据处理过程如下：



模板参数

Tx：数据存储在全局内存中的数据类型。**Ty**：数据存储到寄存器上的类型。**NX**：每个线程读取 **NX** 列数据。**NY**：每个线程读取 **NY** 行数据。**BlockSize**：设备属性，标识当前设备线程索引方式。对于 GPU，`threadIdx.x` 用作线程索引，当前该参数暂不支持。**IsBoundary**：标识是否进行访存边界判断。当 Block 处理的数据总数小于 **NX * NY * blockDim.x** 时，需要进行边界判断以避免访存越界。

函数参数

dst：输出寄存器指针，数据类型为 **Ty**，大小为 **NX * NY**。**src**：当前 Block 的输入数据指针，数据类型为 **Tx**。**size_nx**：当前 Block 在最低维最多偏移 **size_nx** 个元素，参数仅在 **IsBoundary = true** 时参与计算。**size_ny**：当前 Block 在最低维最多偏移 **size_ny** 个元素，参数仅在 **IsBoundary = true** 时参与计算。**stride_nx**：最低维每读取 1 个元素需要跳转 **stride_nx** 个元素。**stride_ny**：最高维每读取 1 个元素需要跳转 **stride_ny** 个元素。

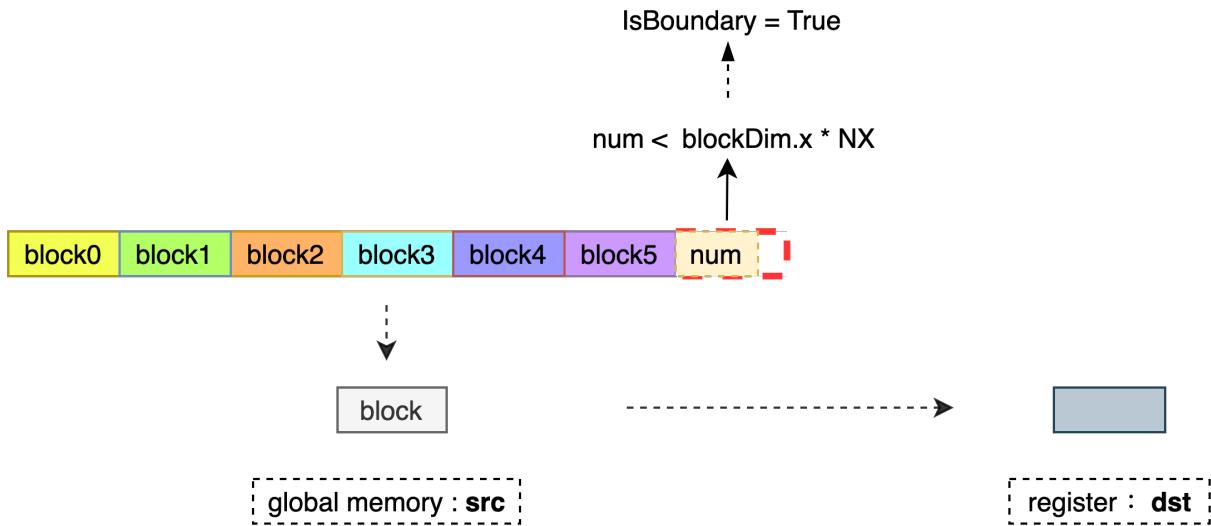
ReadData

函数定义

```
template <typename T, int NX, int NY, int BlockSize, bool IsBoundary = false>
__device__ void ReadData(T* dst, const T* src, int num);
```

函数说明

将 **T** 类型的 1D 数据从全局内存 **src** 中读取到寄存器 **dst** 中。每次连续读取 **NX** 个数据，当前仅支持 **NY = 1**，直到加载 **NX** 个数据到寄存器 **dst** 中。当 **IsBoundary = true** 需要保证 Block 读取的总数据个数不超过 **num**，以避免访存越界。当 (**NX % 4 = 0** 或 **NX % 2 = 0**) 且 **IsBoundary = false** 时，会有更高的访存效率。数据处理过程如下：



模板参数

T : 元素类型。NX : 每个线程连续读取 NX 列数据。NY : 每个线程读取 NY 行数据, 当前仅支持为 NY = 1。BlockSize : 设备属性, 标识当前设备线程索引方式。对于 GPU, threadIdx.x 用作线程索引, 当前该参数暂不支持。IsBoundary : 标识是否进行访存边界判断。当 Block 处理的数据总数小于 NX * NY * blockDim.x 时, 需要进行边界判断以避免访存越界。

函数参数

dst ; 输出寄存器指针, 大小为 NX * NY。src ; 当前 Block 的输入数据指针。num ; 当前 Block 最多读取 num 个元素, 参数仅在 IsBoundary = true 时使用。

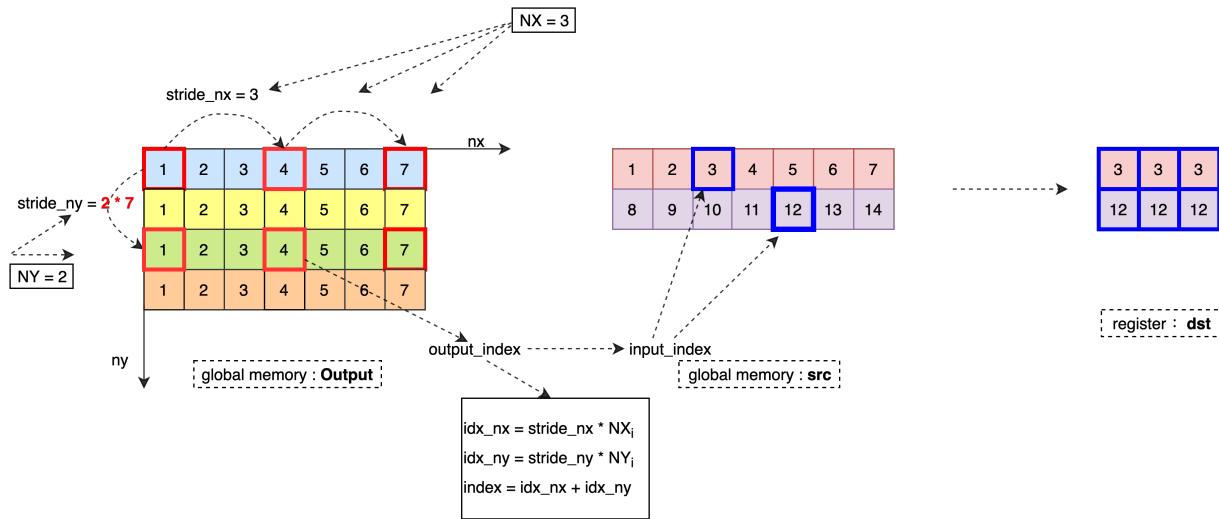
ReadDataBc

函数定义

```
template <typename T, int NX, int NY, int BlockSize, int Rank, bool IsBoundary =  
false>  
__device__ void ReadDataBc(T* dst, const T* src,  
                           uint32_t block_offset,  
                           details::BroadcastConfig<Rank> config,  
                           int total_num_output,  
                           int stride_nx,  
                           int stride_ny);
```

函数说明

将需要进行 broadcast 的 2D 数据按照 T 类型从全局内存 src 中读取到寄存器 dst 中，其中 src 为原始输入数据指针，根据 config 计算当前输出数据对应的输入数据坐标，并将坐标对应的数据读取到寄存器中。数据处理过程如下：



模板参数

T：元素类型。NX：每个线程读取 NX 列数据。NY：每个线程读取 NY 行数据。BlockSize：设备属性，标识当前设备线程索引方式。对于 GPU，threadIdx.x 用作线程索引，当前该参数暂不支持。Rank：原始输出数据的维度。IsBoundary：标识是否进行访存边界判断。当 Block 处理的数据总数小于 $NX * NY * blockDim.x$ 时，需要进行边界判断以避免访存越界。

函数参数

dst：输出寄存器指针，大小为 $NX * NY$ 。src：原始输入数据指针。block_offset：当前 Block 的数据偏移。config：输入输出坐标映射函数，可通过 `BroadcastConfig(const std::vector<int64_t>& out_dims, const std::vector<int64_t>& in_dims, int dim_size)` 进行定义。total_num_output：原始输出的总数据个数，避免访存越界，参数仅在 `IsBoundary = true` 时参与计算。stride_nx：最低维每读取 1 个元素需要跳转 `stride_nx` 个元素。`stride_ny`：最高维每读取 1 个元素需要跳转 `stride_ny` 个元素。

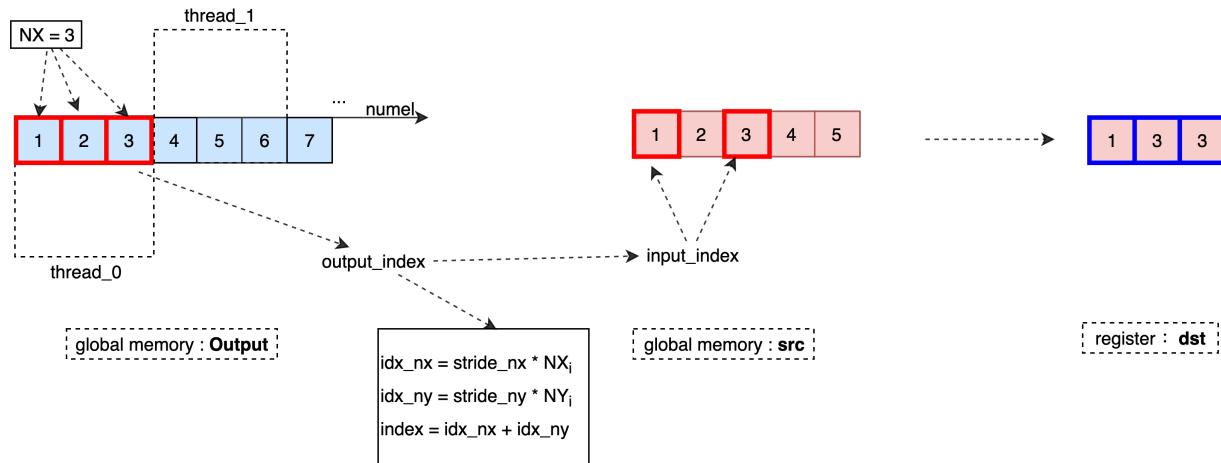
ReadDataBc

函数定义

```
template <typename T, int NX, int NY, int BlockSize, int Rank, bool IsBoundary =  
false>  
__device__ void ReadDataBc(T* dst, const T* src,  
                           uint32_t block_offset,  
                           details::BroadcastConfig<Rank> config,  
                           int total_num_output);
```

函数说明

将需要进行 `broadcast` 的 1D 数据按照 `T` 类型从全局内存 `src` 中读取到寄存器 `dst` 中，其中 `src` 为原始输入数据指针，根据 `config` 计算当前输出数据对应的输入数据坐标，并将坐标对应的数据读取到寄存器中。数据处理过程如下：



模板参数

T：元素类型。**NX**：每个线程连续读取 NX 列数据。**NY**：每个线程读取 NY 行数据。当前仅支持 **NY = 1**。**BlockSize**：设备属性，标识当前设备线程索引方式。对于 GPU，**threadIdx.x** 用作线程索引，当前该参数暂不支持。**Rank**：原始输出数据的维度。**IsBoundary**：标识是否进行访存边界判断。当 Block 处理的数据总数小于 **NX * NY * blockDim.x** 时，需要进行边界判断以避免访存越界。

函数参数

dst：输出寄存器指针，大小为 **NX * NY**。**src**：原始输入数据指针。**block_offset**：当前 Block 的数据偏移。**config**：输入输出坐标映射函数，可通过 **BroadcastConfig(const std::vector<int64_t>& out_dims, const std::vector<int64_t>& in_dims, int dim_size)** 进行定义。**total_num_output**：原始输出的总数据个数，避免访存越界，参数仅在 **IsBoundary = true** 时使用。

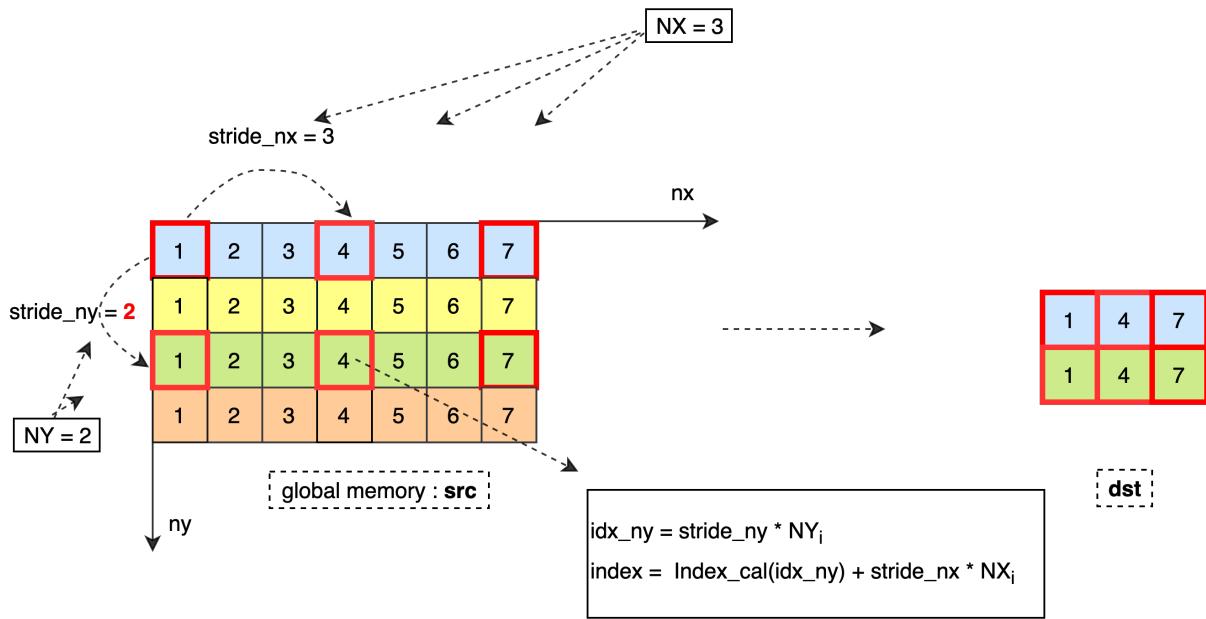
ReadDataReduce

函数定义

```
template <typename Tx, typename Ty, int NX, int NY, int BlockSize, int Rank, typename
→IndexCal, typename Functor, bool IsBoundary = false>
__device__ void ReadDataReduce(Tx* dst,
                               const Ty* src,
                               int block_offset,
                               const IndexCal& index_cal,
                               int size_nx,
                               int size_ny,
                               int stride_nx,
                               int stride_ny,
                               Functor func,
                               bool reduce_last_dim);
```

函数说明

根据 **index_cal** 计算当前输出数据对应的输入数据坐标，将坐标对应的数据从全局内存 **src** 中读取到寄存器 **dst** 中。根据是否需要进行规约操作将数据映射成 2D 数据，总是将最后一维所在的维度映射到线程变化最快的维度，保证数据访存效率最高。数据处理过程如下：



模板参数

Ty : 数据存储在全局内存中的数据类型。Tx : 数据存储到寄存器上的类型。NX : 每个线程读取 NX 列数据。NY : 每个线程读取 NY 行数据。BlockSize : 设备属性, 标识当前设备线程索引方式。对于 GPU, threadIdx.x 用作线程索引, 当前该参数暂不支持。Rank : 原始输出数据的维度。IndexCal : 输入输出坐标映射规则。定义方式如下:

```
struct IndexCal {
    __device__ inline int operator()(int index) const {
        return ...
    }
};
```

Functor : 输入元素在存储到寄存器前做的数据变换, 如: dst[i] = SquareFunctor(src[i])。IsBoundary ; 标识是否进行访存边界判断。当 Block 处理的数据总数小于 NX * NY * blockDim.x 时, 需要进行边界判断以避免访存越界。

函数参数

dst : 输出寄存器指针, 大小为 NX * NY。src : 原始输入数据指针。block_offset ; 当前 Block 的数据偏移。config ; 输入输出坐标映射函数, 可以定义为 IndexCal()。size_nx : 当前 Block 最多读取 size_nx 个不需要进行规约的数据, 参数仅在 IsBoundary = true 时参与计算。size_ny : 当前 Block 最多读取 size_ny 个需要进行规约的数据, 参数仅在 IsBoundary = true 时参与计算。stride_nx : 最低维每读取 1 个元素需要跳转 stride_nx 列。stride_ny : 最高维每读取 1 个元素需要跳转 stride_ny 行。func : 输入数据存储到寄存器前做的数据变换, 如: dst[i] = SquareFunctor(src[i])。reduce_last_dim:

原始输入数据的最低维是否进行 reduce, 当 `reduce_last_dim = true` 按照 `threadIdx.x` 进行索引, 否则使用 `threadIdx.y`。

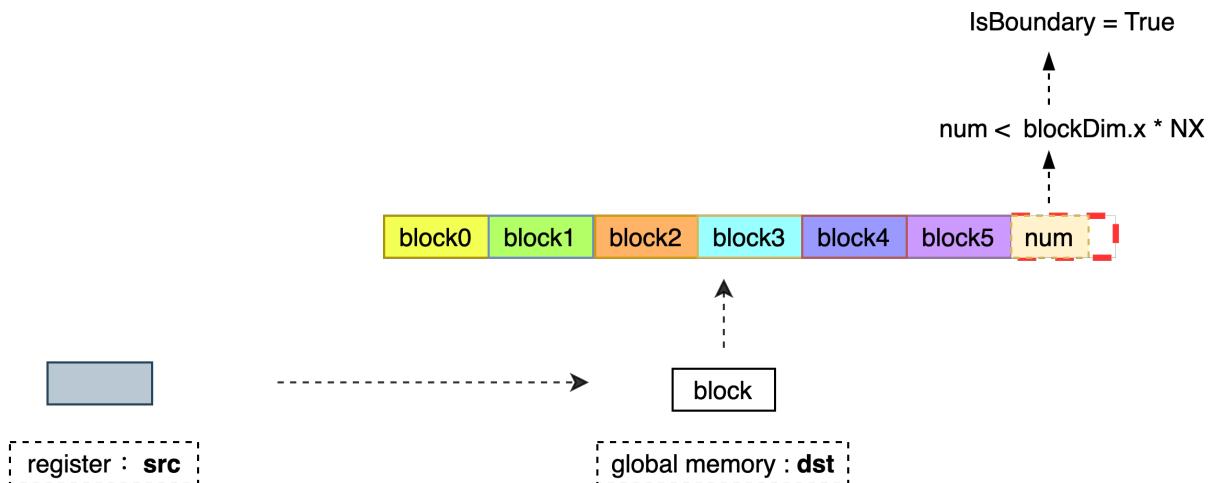
WriteData

函数定义

```
template <typename T, int NX, int NY, int BlockSize, bool IsBoundary = false>
__device__ void WriteData(T* dst, T* src, int num);
```

函数说明

将 `T` 类型的 1D 数据从寄存器 `src` 写到全局内存 `dst` 中。每次连续读取 `NX` 个数据, 当前仅支持 `NY = 1`, 直到写 `NX` 个数据到全局内存 `dst` 中。当 `IsBoundary = true` 需要保证当前 Block 向全局内从中写的总数据个数不超过 `num`, 以避免访存越界。当 $(NX \% 4 = 0$ 或 $NX \% 2 = 0$) 且 `IsBoundary = false` 时, 会有更高的访存效率。数据处理过程如下:



模板参数

`T`：元素类型。`NX`：每个线程连续读取 `NX` 列数据。`NY`：每个线程读取 `NY` 行数据，当前仅支持为 `NY = 1`。`BlockSize`：设备属性，标识当前设备线程索引方式。对于 GPU, `threadIdx.x` 用作线程索引，当前该参数暂不支持。`IsBoundary`：标识是否进行访存边界判断。当 Block 处理的数据总数小于 `NX * NY * blockDim.x` 时，需要进行边界判断以避免访存越界。

函数参数

dst；当前 Block 的输出数据指针。src；寄存器指针，大小为 $NX * NY$ 。num；当前 Block 最多读取 num 个元素，参数仅在 IsBoundary = true 时使用。

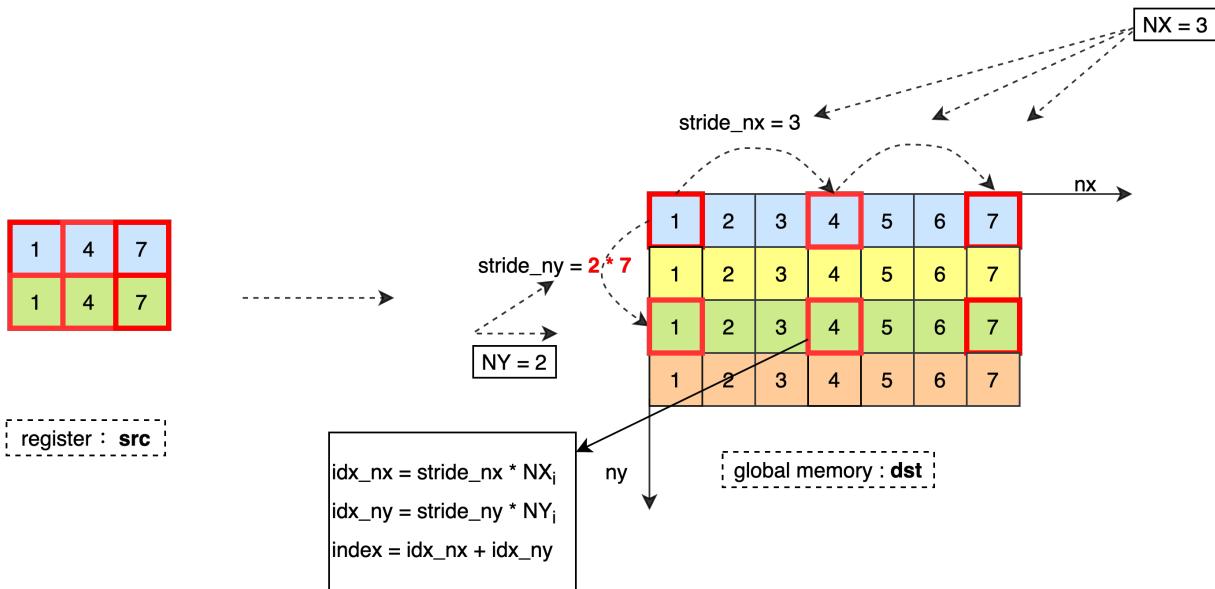
WriteData

函数定义

```
template <typename Tx, typename Ty, int NX, int NY, int BlockSize, bool IsBoundary = false>
__device__ void WriteData(Ty* dst, const Tx* src, int size_nx, int size_ny, int stride_nx, int stride_ny);
```

函数说明

将 Tx 类型的 2D 数据从寄存器中写入到全局内存，并按照 Ty 类型存储到全局内存 dst 中。最低维每写入 1 个元素需要偏移 stride_nx 个元素，最高维每写入 1 个元素需要偏移 stride_ny 个元素，直到将寄存器 $NX * NY$ 个数据全部写入到全局内存 dst 中。当 IsBoundary = true 需要保证当前最高维偏移个数不超过 size_ny，列偏移个数不超过 size_nx。数据处理过程如下：



模板参数

Ty : 数据存储在全局内存中的数据类型。Tx : 数据存储到寄存器上的类型。NX : 每个线程读取 NX 列数据。NY : 每个线程读取 NY 行数据。BlockSize : 设备属性, 标识当前设备线程索引方式。对于 GPU, threadIdx.x 用作线程索引, 当前该参数暂不支持。IsBoundary : 标识是否进行访存边界判断。当 Block 处理的数据总数小于 NX * NY * blockDim.x 时, 需要进行边界判断以避免访存越界。

函数参数

dst : 当前 Block 的输出数据指针。src : 寄存器指针, 数据类型为 Tx。size_nx : Block 最多偏移 size_nx 个元素, 参数仅在 IsBoundary = true 时参与计算。size_ny : Block 最多偏移 size_ny 个元素, 参数仅在 IsBoundary = true 时参与计算。stride_nx : 最低维每读取 1 个元素需要跳转 stride_nx 个元素。stride_ny : 最高维每读取 1 个元素需要跳转 stride_ny 个元素。

Init

函数定义

```
template <typename T, int NX>
__device__ void Init(T* dst, T init_data);
```

函数说明

将寄存器 dst 中的所有元素初始化为 init_data。

模板参数

T : 元素类型。NX : 初始化 NX 个元素。

函数参数

dst ; 寄存器指针。init_data ; 初始值。

Init

函数定义

```
template <typename T, int NX, int IsBoundary = false>
__device__ void Init(T* dst, T* src, int num);
```

函数说明

使用 src 寄存器中的元素对 dst 中的 NX 个元素进行初始化，当 IsBoundary = true 时，初始化个数不超过 num。

模板参数

T : 元素类型。NX : 初始化 NX 个元素。IsBoundary : 是否为初始化边界，当 NX > num 时，IsBoundary = true。

函数参数

dst ; 输出寄存器指针。src ; 输入寄存器指针。num ; 初始化的个数。

API 介绍 - Compute

ElementwiseUnary

函数定义

```
template <typename InT, typename OutT, int NX, int NY, int BlockSize, class OpFunc>
__device__ void ElementwiseUnary(OutT* out, const InT* in, OpFunc compute)
```

函数说明

按照 OpFunc 中的计算规则对 in 进行计算，将计算结果按照 OutT 类型存储到寄存器 out 中。

模板参数

InT：输入数据的类型。OutT：存储到 out 寄存器中的类型。NX：每个线程需要计算 NX 列数据。NY：每个线程需要计算 NY 行数据。BlockSize：设备属性，标识当前设备线程索引方式。对于 GPU，threadIdx.x 用作线程索引，当前该参数暂不支持。OpFunc：计算规则，定义方式请参考 OpFunc 小节。

函数参数

out：输出寄存器指针，大小为 NX * NY。in：输入寄存器指针，大小为 NX * NY。compute：计算函数，声明为 OpFunc<InT, OutT>()。

ElementwiseBinary

函数定义

```
template <typename InT, typename OutT, int NX, int NY, int BlockSize, class OpFunc>
__device__ void ElementwiseBinary(OutT* out, const InT* in1, const InT* in2, OpFunc
→compute)
```

函数说明

按照 OpFunc 中的计算规则对 in1、in2 进行计算，将计算结果按照 OutT 类型存储到寄存器 out 中。

模板参数

InT：输入数据的类型。OutT：存储到 out 寄存器中的类型。NX：每个线程需要计算 NX 列数据。NY：每个线程需要计算 NY 行数据。BlockSize：设备属性，标识当前设备线程索引方式。对于 GPU，threadIdx.x 用作线程索引，当前该参数暂不支持。OpFunc：计算规则，定义方式请参考 OpFunc 小节。

函数参数

out：输出寄存器指针，大小为 NX * NY。in1：左操作数寄存器指针，大小为 NX * NY。in2：右操作数寄存器指针，大小为 NX * NY。compute：声明为 OpFunc<InT>() 的计算对象。

CycleBinary

函数定义

```
template <typename InT, typename OutT, int NX, int NY, int BlockSize, class OpFunc>
__device__ void CycleBinary(OutT* out, const InT* in1, const InT* in2, OpFunc compute)
```

函数说明

按照 OpFunc 中的计算规则对 in1、in2 进行计算，将计算结果按照 OutT 类型存储到寄存器 out 中。in1 的 Shape 为 [1, NX]，in2 的 Shape 为 [NY, NX]，实现 in1、in2 的循环计算，out 的 Shape 是 [NY, NX]。

模板参数

InT：输入数据的类型。OutT：存储到 out 寄存器中的类型。NX：每个线程需要计算 NX 列数据。NY：每个线程需要计算 NY 行数据。BlockSize：设备属性，标识当前设备线程索引方式。对于 GPU，threadIdx.x 用作线程索引，当前该参数暂不支持。OpFunc：计算规则，定义方式请参考 OpFunc 小节。

函数参数

out：输出寄存器指针，大小为 NX * NY。in1：左操作数寄存器指针，大小为 NX。in2：右操作数寄存器指针，大小为 NX * NY。compute：声明为 OpFunc<InT>() 的计算对象。

ElementwiseTernary

函数定义

```
template <typename InT, typename OutT, int NX, int NY, int BlockSize, class OpFunc>
__device__ void ElementwiseTernary(OutT* out, const InT* in1, const InT* in2, const
→ InT* in3, OpFunc compute)
```

函数说明

按照 OpFunc 中的计算规则对 in1、in2、in3 进行计算，将计算结果按照 OutT 类型存储到寄存器 out 中。

模板参数

InT：输入数据的类型。OutT：存储到 out 寄存器中的类型。NX：每个线程需要计算 NX 列数据。NY：每个线程需要计算 NY 行数据。BlockSize：设备属性，标识当前设备线程索引方式。对于 GPU，threadIdx.x 用作线程索引，当前该参数暂不支持。OpFunc：计算规则，定义方式请参考 OpFunc 小节。

函数参数

out：输出寄存器指针，大小为 NX * NY。in1：操作数 1 的寄存器指针，大小为 NX * NY。in2：操作数 2 的寄存器指针，大小为 NX * NY。in3：操作数 3 的寄存器指针，大小为 NX * NY。compute：声明为 `OpFunc<InT>()` 的计算对象。

ElementwiseAny

函数定义

```
template <typename InT, typename OutT, int NX, int NY, int BlockSize, int Arity,>
class OpFunc>
__device__ void ElementwiseAny(OutT* out, InT (*ins) [NX * NY], OpFunc compute)
```

函数说明

按照 OpFunc 中的计算规则对 ins 中的输入进行计算，将计算结果按照 OutT 类型存储到寄存器 out 中，所有输入输出的维度相同。

模板参数

InT：输入数据的类型。OutT：存储到 out 寄存器中的类型。NX：每个线程需要计算 NX 列数据。NY：每个线程需要计算 NY 行数据。BlockSize：设备属性，标识当前设备线程索引方式。对于 GPU，threadIdx.x 用作线程索引，当前该参数暂不支持。Arity：指针数组 ins 中指针个数。OpFunc：计算规则，定义方式请参考 OpFunc 小节。

函数参数

`out`：输出寄存器指针，大小为 $NX * NY$ 。`ins`：由多输入指针构成的指针数组，大小为 `Arity`。
`compute`：声明为 `OpFunc<InT>()` 的计算对象。

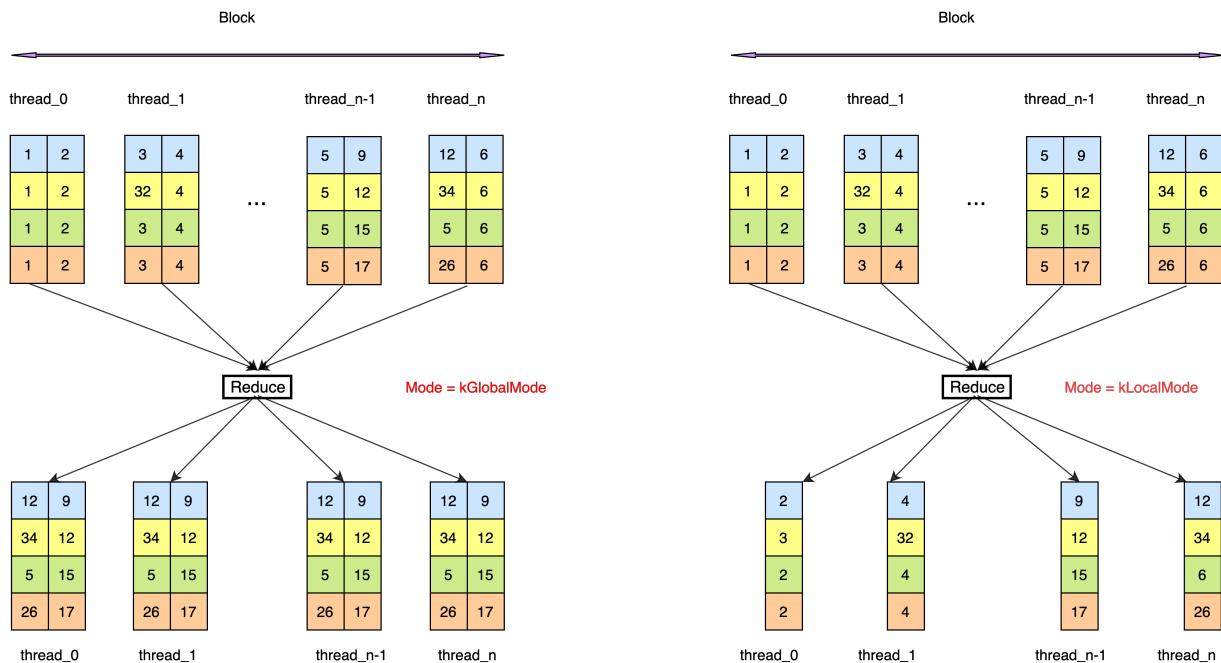
Reduce

函数定义

```
template <typename T, int NX, int NY, int BlockSize, class ReduceFunctor, 
         ~details::ReduceMode Mode>
__device__ void Reduce(T* out, const T* in, ReduceFunctor reducer, bool reduce_last_
         ~dim)
```

函数说明

根据 `reducer` 对 `in` 中的数据进行数据规约，输入 `in` 的 Shape 为 $[NY, NX]$ ，当 `Mode = kLocalMode` 时，对 `in` 沿着 `NX` 方向进行规约，完成线程内规约，`out` 为 $[NY, 1]$ ；当 `Mode = kGlobalMode` 时，使用共享内存完成 `block` 内线程间的规约操作，`in` 和 `out` 的 size 相同，均为 $[NY, NX]$ 。ReduceMax 数据处理过程如下：



模板参数

T : 输入数据的类型。NX : 每个线程需要计算 NX 列数据。NY : 每个线程需要计算 NY 行数据。

BlockSize : 设备属性, 标识当前设备线程索引方式。对于 GPU, threadIdx.x 用作线程索引, 当前该参数暂不支持。ReduceFunctor : 计算规则, 定义方式请参考 OpFunc 小节。Mode : 规约模式, 可以取值为 kGlobalMode、kLocalMode。

函数参数

out : 输出寄存器指针, 大小为 NX * NY。in : 输入寄存器指针, 大小为 NX * NY。reducer : 规约方式, 可以使用 ReduceFunctor<T>() 进行定义。reduce_last_dim : 表示原始输入的最后一维是否进行规约。

API 介绍 - OpFunc

介绍 Kernel Primitive API 定义的 Functor, 当前一共有 13 个 Functor 可以直接使用。

Unary Functor

ExpFunctor

定义

```
namespace kps = paddle::operators::kernel_primitives;
template <typename Tx, typename Ty = Tx>
struct kps::ExpFunctor<Tx, Ty>();
```

功能介绍

对 Tx 类型的输入数据做 Exp 操作, 并将结果转成 Ty 类型返回。

模板参数

Tx : 输入数据的类型。Ty : 返回类型。

使用示例

```
const int VecSize = 1;
float data[VecSize];
float out[VecSize];

kps::ElementwiseUnary<float, float, VecSize, 1, 1, kps::ExpFunctor<float>>(out, data,_
↪kps::ExpFunctor<float>());

// data[0] = 0;
// out[0] = exp(data);
// out[0] = 1
```

IdentityFunctor

定义

```
namespace kps = paddle::operators::kernel_primitives;
template <typename Tx, typename Ty = Tx>
struct kps::IdentityFunctor<Tx, Ty>();
```

功能介绍

将 Tx 类型的输入数据转成 Ty 类型返回。

模板参数

Tx : 输入数据的类型。Ty : 返回类型。

使用示例

```
const int VecSize = 1;
float data[VecSize];
int out[VecSize];

kps::ElementwiseUnary<float, int, VecSize, 1, 1, kps::IdentityFunctor<float, int>>
↪(out, data, kps::IdentityFunctor<float, int>());

// data[0] = 1.3;
// out[0] = 1
```

DivideFunctor

定义

```
namespace kps = paddle::operators::kernel_primitives;
template <typename Tx, typename Ty = Tx>
struct kps::DivideFunctor<Tx, Ty> (num);
```

功能介绍

将 Tx 类型的输入数据除以 num，并将结果转成 Ty 类型返回。

模板参数

Tx : 输入数据的类型。Ty : 返回类型。

使用示例

```
const int VecSize = 1;
float data[VecSize];
float out[VecSize];
float num = 10.0;

kps::ElementwiseUnary<float, float, VecSize, 1, 1, kps::DivideFunctor<float>>(out,
    ↪data, kps::DivideFunctor<float>(num));

// data[0] = 3.0
// out[0] = (3.0 / 10.0)
// out[0] = 0.3
```

SquareFunctor

定义

```
namespace kps = paddle::operators::kernel_primitives;
template <typename Tx, typename Ty = Tx>
struct kps::SquareFunctor<Tx, Ty>();
```

功能介绍

对 Tx 类型的输入数做 Square 操作，并将结果转成 Ty 类型返回。

模板参数

Tx : 输入数据的类型。Ty : 返回类型。

使用示例

```
const int VecSize = 1;
float data[VecSize];
float out[VecSize];

kps::ElementwiseUnary<float, float, VecSize, 1, 1, kps::SquareFunctor<float>>(out,_
    ↪data, kps::SquareFunctor<float>());

// data[0] = 3.0
// out[0] = (3.0 * 3.0)
// out[0] = 9.0
```

Binary Functor

MinFunctor

定义

```
namespace kps = paddle::operators::kernel_primitives;
template <typename T>
struct kps::MinFunctor<T>();
```

功能介绍

返回两个输入中的最小值。MinFunctor 提供了用于数据初始化的 initial() 函数，返回 T 类型表示的最大值。

模板参数

T : 数据类型。

使用示例

```
const int VecSize = 1;
float input1[VecSize];
float input2[VecSize];
float out[VecSize];

kps::ElementwiseBinary<float, float, VecSize, 1, 1, kps::MinFunctor<float>>(out, input1, input2, kps::MinFunctor<float>());

// input1[0] = 3.0
// input2[0] = 1.0
// out = input1[0] < input2[0] ? input1[0] : input2[0]
// out[0] = 1.0
```

MaxFunctor

定义

```
namespace kps = paddle::operators::kernel_primitives;
template <typename T>
struct kps::MaxFunctor<T>();
```

功能介绍

返回两个输入中的最大值。MaxFunctor 提供了用于数据初始化的 initial() 函数，返回 T 类型表示的最小值。

模板参数

T : 数据类型。

使用示例

```
const int VecSize = 1;
float input1[VecSize];
float input2[VecSize];
float out[VecSize];

kps::ElementwiseBinary<float, float, VecSize, 1, 1, kps::MaxFunctor<float>>(out,_
→input1, input2, kps::MaxFunctor<float>());

// input1[0] = 3.0
// input2[0] = 1.0
// out = input1[0] > input2[0] ? input1[0] : input2[0]
// out[0] = 3.0
```

AddFunctor

定义

```
namespace kps = paddle::operators::kernel_primitives;
template <typename T>
struct kps::AddFunctor<T>();
```

功能介绍

返回两个输入之和。AddFunctor 提供了用于数据初始化的 initial() 函数，返回 T 类型表示的数据 0。

模板参数

T: 数据类型。

使用示例

```
const int VecSize = 1;
float input1[VecSize];
float input2[VecSize];
float out[VecSize];

kps::ElementwiseBinary<float, float, VecSize, 1, 1, kps::AddFunctor<float>>(out,_
→input1, input2, kps::AddFunctor<float>());
```

(下页继续)

(续上页)

```
// input1[0] = 3.0
// input2[0] = 1.0
// out = input1[0] + input2[0]
// out[0] = 4.0
```

MulFunctor

定义

```
namespace kps = paddle::operators::kernel_primitives;
template <typename T>
struct kps::MulFunctor<T>();
```

功能介绍

返回两个输入的乘积。MulFunctor 提供了用于数据初始化的 initial() 函数，返回 T 类型表示的数据 1。

模板参数

T : 数据类型。

使用示例

```
const int VecSize = 1;
float input1[VecSize];
float input2[VecSize];
float out[VecSize];

kps::ElementwiseBinary<float, float, VecSize, 1, 1, kps::MulFunctor<float>>(out,_
    input1, input2, kps::MulFunctor<float>());

// input1[0] = 3.0
// input2[0] = 1.0
// out = input1[0] * input2[0]
// out[0] = 3.0
```

LogicalOrFunctor

定义

```
namespace kps = paddle::operators::kernel_primitives;
template <typename T>
struct kps::LogicalOrFunctor<T>();
```

功能介绍

返回两个输入元素逻辑或操作后的结果。LogicalOrFunctor 提供了用于数据初始化的 initial() 函数，返回 T 类型表示的数据 false。

模板参数

T : 数据类型。

使用示例

```
const int VecSize = 1;
bool input1[VecSize];
bool input2[VecSize];
bool out[VecSize];

kps::ElementwiseBinary<bool, bool, VecSize, 1, 1, kps::LogicalOrFunctor<bool>>(out,_
    ↵input1, input2, kps::LogicalOrFunctor<bool>());

// input1[0] = false
// input2[0] = true
// out = input1[0] || input2[0]
// out[0] = true
```

LogicalAndFunctor

定义

```
namespace kps = paddle::operators::kernel_primitives;
template <typename T>
struct kps::LogicalAndFunctor<T>();
```

功能介绍

返回两个输入元素逻辑与操作后的结果。LogicalAndFunctor 提供了用于数据初始化的 initial() 函数，返回 T 类型表示的数据 true。

模板参数

T : 数据类型。

使用示例

```
const int VecSize = 1;
bool input1[VecSize];
bool input2[VecSize];
bool out[VecSize];

kps::ElementwiseBinary<bool, bool, VecSize, 1, 1, kps::LogicalAndFunctor<bool>>(out,_
    ↪input1, input2, kps::LogicalAndFunctor<bool>());

// input1[0] = false
// input2[0] = true
// out = input1[0] && input2[0]
// out[0] = false
```

SubFunctor

定义

```
namespace kps = paddle::operators::kernel_primitives;
template <typename T>
struct kps::SubFunctor<T>();
```

功能介绍

两个输入进行减法操作。SubFunctor 提供了用于数据初始化的 initial() 函数，返回 T 类型表示的数据 0。

模板参数

T : 数据类型。

使用示例

```
const int VecSize = 1;
float input1[VecSize];
float input2[VecSize];
float out[VecSize];

kps::ElementwiseBinary<float, float, VecSize, 1, 1, kps::SubFunctor<float>>(out,_
    ↪input1, input2, kps::SubFunctor<float>());

// input1[0] = 3.0
// input2[0] = 1.0
// out = input1[0] - input2[0]
// out[0] = 2.0
```

DivFunctor

定义

```
namespace kps = paddle::operators::kernel_primitives;
template <typename T>
struct kps::DivFunctor<T>();
```

功能介绍

两个输入进行除法操作。DivFunctor 提供了用于数据初始化的 initial() 函数，返回 T 类型表示的数据 1。

模板参数

T : 数据类型。

使用示例

```
const int VecSize = 1;
float input1[VecSize];
float input2[VecSize];
float out[VecSize];

kps::ElementwiseBinary<float, float, VecSize, 1, 1, kps::DivideFunctor<float>>(out,
    input1, input2, kps::DivideFunctor<float>());

// input1[0] = 3.0
// input2[0] = 1.0
// out = input1[0] / input2[0]
// out[0] = 3.0
```

FloorDivFunctor

定义

```
namespace kps = paddle::operators::kernel_primitives;
template <typename T>
struct kps::FloorDivFunctor<T>();
```

功能介绍

两个输入进行除法操作，返回整数部分。FloorDivFunctor 提供了用于数据初始化的 initial() 函数，返回 T 类型表示的数据 1。

模板参数

T : 数据类型。

使用示例

```
const int VecSize = 1;
float input1[VecSize];
float input2[VecSize];
float out[VecSize];
```

(下页继续)

(续上页)

```
kps::ElementwiseBinary<float, float, VecSize, 1, 1, kps::FloorDivFunctor<float>>(out, _  
→input1, input2, kps::FloorDivFunctor<float>());  
  
// input1[0] = 3.0  
// input2[0] = 2.0  
// out = input1[0] / input2[0]  
// out[0] = 1.0
```

Functor 定义规则

当前计算函数中仅 ElementwiseAny 支持 Functor 参数设置为指针，其他计算函数的 Functor 仅能设置为普通参数。

普通参数传递

除 ElementwiseAny API 外其他计算函数仅支持普通参数传递。例如需要实现 $(a + b) * c$ 可将 Functor 定义如下：

ExampleTernaryFunctor:

```
namespace kps = paddle::operators::kernel_primitives;  
template <typename T>  
struct ExampleTernaryFunctor {  
    inline HOSTDEVICE T operator()(const T &input1, const T &input2, const T &input3) _  
→const {  
        return ((input1 + input2) * input3);  
    }  
};
```

示例

```
template<int VecSize, typename InT, typename OutT>  
__global__ void ElementwiseTernaryKernel(InT *input0, InT *input1, InT *input2, OutT _  
→*output, int size) {  
    // global memory input pointer input0, input1, input2  
    auto functor = ExampleTernaryFunctor<InT>();  
  
    const int NX = 4;  
    const int NY = 1;  
    const int BlockSize = 1;  
    const bool IsBoundary = false;
```

(下页继续)

(续上页)

```

int data_offset = NX * blockIdx.x * blockDim.x;
int num = size - data_offset;
// if num < NX * blockDim.x set IsBoundary = true

InT in0[NX * NY];
InT in1[NX * NY];
InT in2[NX * NY];
OutT out[NX * NY];

// each thread reads NX data continuously, and each block reads num data continuously
kps::ReadData<InT, NX, NY, BlockSize, IsBoundary>(in0, input0 + data_offset, num);
kps::ReadData<InT, NX, NY, BlockSize, IsBoundary>(in1, input1 + data_offset, num);
kps::ReadData<InT, NX, NY, BlockSize, IsBoundary>(in2, input2 + data_offset, num);
kps::ElementwiseTernary<InT, OutT, NX, NY, BlockSize, ExampleTernaryFunctor<InT>>(out,
    ↳ in0, in1, in2, functor);

...
}

```

指针传递

在进行 ElementwiseAny 的 Functor 定义时，需要保证 operate() 函数的参数是数组指针。例如要实现功能： $(a + b) * c + d$ ，则可以结合 ElementwiseAny 与 Functor 完成对应计算。

ExampleAnyFunctor 定义：

```

namespace kps = paddle::operators::kernel_primitives;
template <typename T>
struct ExampleAnyFunctor {
    inline HOSTDEVICE T operator()(const T * args) const { return ((arg[0] + arg[1]) *_
    ↳ arg[2] + arg[3]); }
};

```

示例

```

template<int VecSize, typename InT, typename OutT>
__global__ void ElementwiseAnyKernel(InT *input0, InT *input1, InT * input2, InT *_
    ↳ input3, OutT *output, int size) {
// global memory input pointer input0, input1, input2, input3
auto functor = ExampleAnyFunctor<InT>();

```

(下页继续)

(续上页)

```

const int NX = 4;
const int NY = 1;
const int BlockSize = 1;
const bool IsBoundary = false;
const int Arity = 4; // the pointers of inputs
int data_offset = NX * blockIdx.x * blockDim.x;
int num = size - data_offset;
// if num < NX * blockDim.x set IsBoundary = true

InT inputs[Arity][NX * NY];
OutT out[NX * NY];

// each thread reads NX data continuously, and each block reads num data continuously
kps::ReadData<InT, NX, NY, BlockSize, IsBoundary>(inputs[0], input0 + data_offset,_
↪num);
kps::ReadData<InT, NX, NY, BlockSize, IsBoundary>(inputs[1], input1 + data_offset,_
↪num);
kps::ReadData<InT, NX, NY, BlockSize, IsBoundary>(inputs[2], input2 + data_offset,_
↪num);
kps::ReadData<InT, NX, NY, BlockSize, IsBoundary>(inputs[3], input3 + data_offset,_
↪num);
kps::ElementwiseAny<InT, OutT, NX, NY, BlockSize, Arity, ExampleAnyFunctor<InT>>(out,_
↪inputs, functor);

...
}

```

API 示例

- ElementwiseAdd : 加法操作，输入和输出具有相同 Shape。
- Reduce : 针对最高维进行规约操作。
- Model : Resnet50 执行流程。

示例 - ElementwiseAdd

功能说明

- 完成相同 Shape 的两数相加，输入为 InT 类型，输出为 OutT 类型，根据 OpFunc 完成对应的计算。

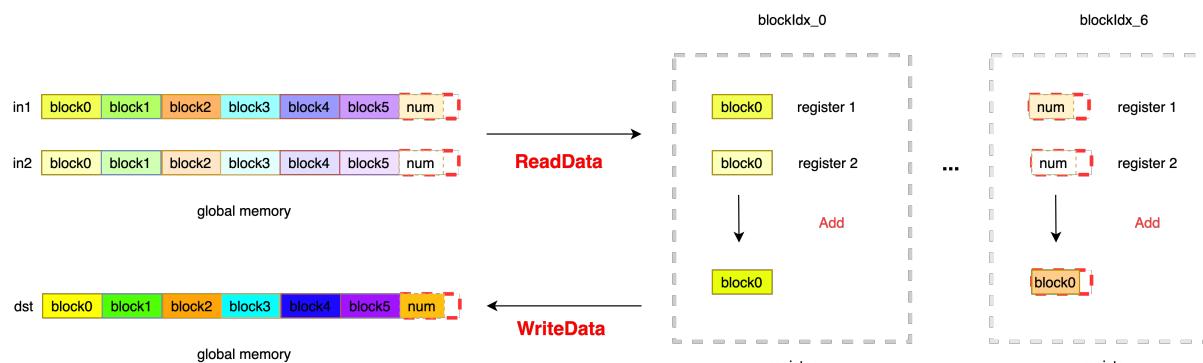
OpFunc 定义

OpFunc：用于定义当前数据的计算规则，AddFunctor 定义如下：

```
template <typename Int>
struct AddFunctor {
    HOSTDEVICE Int operator()(const Int &a, const Int &b) const { return (a + b); }
};
```

Kernel 实现说明

每个线程连续读取 VecSize 个元素，根据剩余元素 num 与 VecSize * blockDim.x 的关系，将数据处理分为 2 部分，第一部分，当 $\text{VecSize} * \text{blockDim.x} > \text{num}$ 表示当前数据处理需要进行边界处理，将 IsBoundary 设置为 true，避免访存越界；第二部分，不需要进行边界处理，设置 IsBoundary = false。根据当前 block 的数据指针，将数据从全局内存中读取到寄存器中，完成加法操作后，将数据写入全局内存中。注意此处使用 Init 函数对寄存器 arg0, arg1 进行初始化，避免当 arg0 或者 arg1 作为分母时出现为 0 的情况。根据 OpFunc 完成两数求和操作，当需要进行两数相乘，可以直接修改对应的 Functor 即可，可以直接复用 Kernel 代码，提升开发效率。



数据处理过程如下：

Kernel 代码

```

#include "kernel_primitives/kernel_primitives.h"

template<int VecSize, typename InT, typename OutT, typename OpFunc, bool IsBoundary>
__device__ void ElementwiseAddImpl(InT *in0, InT *in1, OutT *out, OpFunc func, int num) {

    InT arg0[VecSize];
    InT arg1[VecSize];
    OutT result[VecSize];

    // init arg0 and arg1
    Init<InT, VecSize>(arg0, static_cast<OutT>(1.0f));
    Init<InT, VecSize>(arg1, static_cast<OutT>(1.0f));

    // read data from global memory
    ReadData<InT, InT, VecSize, 1, 1, IsBoundary>(arg0, in0, num);
    ReadData<InT, InT, VecSize, 1, 1, IsBoundary>(arg1, in1, num);

    // compute result[i] = args[i] + arg1[i]
    ElementwiseBinary<InT, OutT, VecSize, 1, 1, OpFunc>(result, arg0, arg1, func);

    // write data
    WriteData<OutT, VecSize, 1, 1, IsBoundary>(out, result, num);
}

template<int VecSize, typename InT, typename OutT>
__global__ void ElementwiseAdd(InT *in0, InT *in1, OutT *out, int size) {

    // get the data offset of this Block
    int data_offset = VecSize * blockDim.x * blockDim.x;

    // get the stride offset the block
    int stride = gridDim.x * blockDim.x * VecSize;

    for (int offset = data_offset; offset < size; offset += stride) {
        if (offset + blockDim.x * VecSize < size) { // set IsBoundary = false

            ElementwiseAddImpl<VecSize, InT, OutT, AddFunctor<InT, OutT>, false>(in0 + offset,
                in1 + offset, out + offset, AddFunctor<InT, OutT>(), size - offset);

        } else { // left num is smaller than blockDim.x * VecSize, IsBoundary must be true
    }
}

```

(下页继续)

(续上页)

```

ElementwiseAddImpl<VecSize, InT, OutT, AddFunctor<InT, OutT>, true>(in0 +_
↪offset, in1 + offset, out + offset, AddFunctor<InT, OutT>(), size - offset);

}
}

}

```

示例 - Reduce

功能说明

- 根据 ReduceOp 中定义的计算规则对最高维度进行规约操作，例如输入为 $x[N, H, W, C]$, axis 取值为 0, 规约后为 $out[1, H, W, C]$ ，此处以 ReduceSum 为例进行介绍。

ReduceOp 定义

```

template <typename Tx, typename Ty = Tx>
struct IdentityFunctor {
    HOSTDEVICE explicit inline IdentityFunctor(int n) {}

    HOSTDEVICE inline Ty operator()(const Tx& x) const {
        return static_cast<Ty>(x);
    }
};

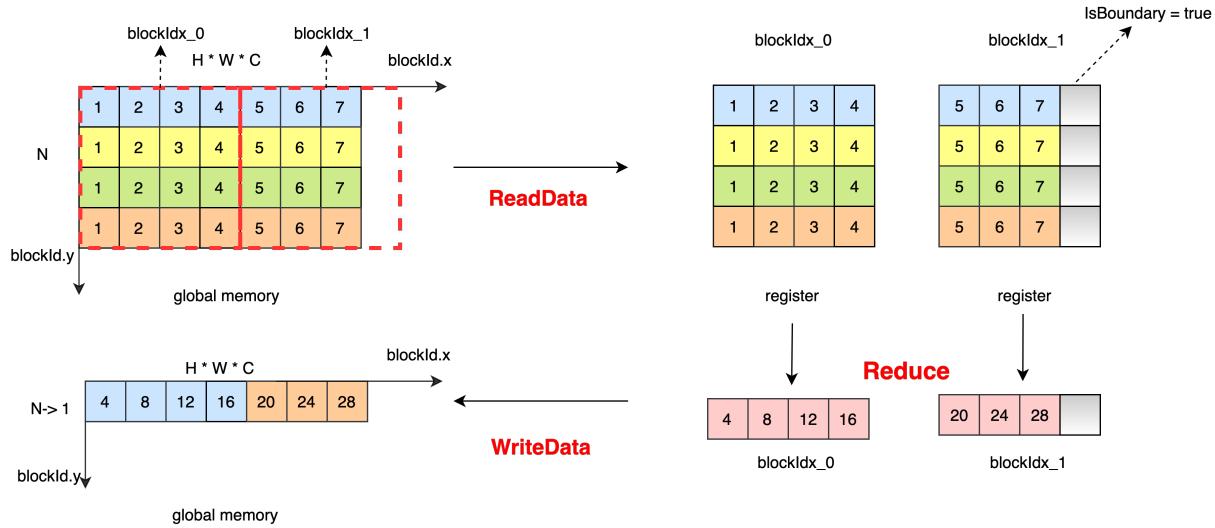
template <typename Tx, typename Ty = Tx>
struct AddFunctor {
    inline Ty initial() { return static_cast<Ty>(0.0f); }

    __device__ __forceinline__ Ty operator()(const Ty &a, const Ty &b) const {
        return b + a;
    }
};

```

kernel 实现说明

对最高维进行规约操作，将不需要进行规约的维度进行合并，根据 NX 和 blockDim.x 对 $H * W * C$ 进行 block 划分。对于 blockIdx_1 ，数据个数小于 $\text{blockDim.x} * NX$ ，则设置 $\text{IsBoundary} = \text{true}$ ，避免访存越界。将数据从全局内存中读取到寄存器中，每个线程读取 4 个元素，线程间数据没有依赖，进行线程内规约操作得到最终结果。将数据从寄存器写入全局内存中。ReduceSum 数据处理过程如下：



kernel 代码

```

template <typename Tx, typename Ty, typename MPType, typename ReduceOp, typename_
<TransformOp, bool IsBoundary = false>
__device__ void HigherDimImpl(const Tx* x, Ty* y, ReduceOp reducer,
                             TransformOp transform, MPType init,
                             int reduce_num, int left_num,
                             int block_num) {

    const int NY = 2;
    int idx = blockIdx.x * blockDim.x;
    int idy = blockIdx.y * block_num; // block_offset of rows
    Tx reduce_input[NY];
    MPType reduce_compute[NY];
    MPType result = init;

    int block_offset = idy * left_num + idx + blockIdx.z * reduce_num * left_num; //_
    ↪the offset of this block
    int store_offset = blockIdx.y * left_num + blockIdx.z * gridDim.y * left_num + idx;

    const Tx* input = x + block_offset;

```

(下页继续)

(续上页)

```

// how many columns left
int num = left_num - idx;

// how many rows have to be reduced
int loop = reduce_num - idy;
loop = loop > block_num ? block_size : loop;

for (int loop_index = 0; loop_index < loop; loop_index += NY) {
    kps::ReadData<Tx, Tx, 1, NY, 1, IsBoundary>(&reduce_input[0], input + loop_index_
→* left_num, num, NY, 1, left_num);
    kps::ElementwiseUnary<Tx, MPType, REDUCE_VEC_num, 1, 1, TransformOp>(&reduce_
→compute[0], &reduce_input[0], transform);
    kps::Reduce<MPType, NY, 1, 1, ReduceOp, kps::details::ReduceMode::kLocalMode>(&
→result, &reduce_compute[0], reducer, false);
}

Ty temp_data = static_cast<Ty>(result);
kps::WriteData<Ty, 1, 1, 1, IsBoundary>(y + store_offset, &temp_data, num);
}

template <typename Tx, typename Ty, typename MPType, typename ReduceOp, typename_
→TransformOp>
__global__ void ReduceHigherDimKernel(const Tx* x, Ty* y, ReduceOp reducer,
                                       TransformOp transform, MPType init,
                                       int reduce_num, int left_num,
                                       int blocking_num) {

    // get the remaining data of this kernel
    int num = left_num - blockIdx.x * blockDim.x;

    if (num >= blockDim.x) {

        // The remaining data is larger than blockdim.x
        HigherDimImpl<Tx, Ty, MPType, AddFunctor<Tx, Ty>, IdentityFunctor<Tx, Ty>, false>(
            x, y, AddFunctor<Tx, Ty>(), IdentityFunctor<Tx, Ty>(), init, reduce_num, left_
→num, blocking_num);

    } else {

        // The remaining data is smaller than blockdim.x, IsBoundary must be true
        HigherDimImpl<Tx, Ty, MPType, AddFunctor<Tx, Ty>, IdentityFunctor<Tx, Ty>, true>(
            x, y, AddFunctor<Tx, Ty>(), IdentityFunctor<Tx, Ty>(), init, reduce_num, left_
→num, blocking_num);
}

```

(下页继续)

(续上页)

```

    }
}
```

示例 - Model

模型运行说明

- 在 GPU 平台上默认使用 Kernel Primitive API 编写的算子。
- 在昆仑芯 2 代 (XPU2) 平台上使用 Kernel Primitive API 编写的算子需要开启 FLAGS_run_kp_kernel 环境变量。

XPU Kernel Primitive API Paddle 模型运行

以 resnet50 为例展示昆仑芯 2 代 (XPU2) 平台 KP 模型运行的基本流程。

- 1. 安装 PaddlePaddle XPU2 KP 安装包，当前仅支持 python3.7

```
pip install https://paddle-wheel.bj.bcebos.com/2.3.0/xpu2/kp/paddlepaddle_xpu-2.3.0-
→cp37-cp37m-linux_x86_64.whl
```

- 1. 下载模型库并安装

```
git clone -b develop https://github.com/PaddlePaddle/PaddleClas.git
cd PaddleClas
python -m pip install -r requirements.txt
```

- 1. 下载数据集

```
cd dataset
rm -rf ILSVRC2012
wget -nc https://paddle-imagenet-models-name.bj.bcebos.com/data/whole_chain/whole_
→chain_CIFAR100.tar
tar xf whole_chain_CIFAR100.tar
ln -s whole_chain_CIFAR100 ILSVRC2012
cd ILSVRC2012
mv train.txt train_list.txt
mv test.txt val_list.txt
```

- 1. 模型运行

```

cd ../..
export FLAGS_selected_xpus=0
export FLAGS_run_kp_kernel=1
export XPUSIM_DEVICE_MODEL=KUNLUN2
nohup python tools/train.py \
-c ppcls/configs/ImageNet/ResNet/ResNet50.yaml \
-o Global.device=xpu > ResNet50_xpu2.log &

```

[2022/05/29 01:09:29] ppcls WARNING: The training strategy provided by PaddleClas is based on 4 gpus. But the number of gpu is 1 in current training. Please modify the strategy (learning rate, batch size and so on) if use this config to train.
[2022/05/29 01:09:31] ppcls INFO: [Train][Epoch 1/1][Iter: 0/782]lr(PiecewiseDecay): 0.10000000, top1: 0.00000, top5: 0.00000, CEloss: 6.95377, loss: 6.95377, batch_cost: 2.042668, reader_cost: 0.17086, ips: 31.33172 samples/s, eta: 0:26:37
[2022/05/29 01:09:48] ppcls INFO: [Train][Epoch 1/1][Iter: 10/782]lr(PiecewiseDecay): 0.10000000, top1: 0.00426, top5: 0.03551, CEloss: 10.42891, loss: 10.42891, batch_cost: 1.629738, reader_cost: 0.00020, ips: 39.27022 samples/s, eta: 0:20:58
[2022/05/29 01:10:04] ppcls INFO: [Train][Epoch 1/1][Iter: 20/782]lr(PiecewiseDecay): 0.10000000, top1: 0.00893, top5: 0.03869, CEloss: 9.06758, loss: 9.06758, batch_cost: 1.616988, reader_cost: 0.00020, ips: 39.15799 samples/s, eta: 0:20:32
[2022/05/29 01:10:20] ppcls INFO: [Train][Epoch 1/1][Iter: 30/782]lr(PiecewiseDecay): 0.10000000, top1: 0.01058, top5: 0.04637, CEloss: 8.54764, loss: 8.54764, batch_cost: 1.613428, reader_cost: 0.00032, ips: 39.66726 samples/s, eta: 0:20:13
[2022/05/29 01:10:36] ppcls INFO: [Train][Epoch 1/1][Iter: 40/782]lr(PiecewiseDecay): 0.10000000, top1: 0.01220, top5: 0.05450, CEloss: 8.08977, loss: 8.08977, batch_cost: 1.612708, reader_cost: 0.00028, ips: 39.68493 samples/s, eta: 0:19:56
[2022/05/29 01:10:52] ppcls INFO: [Train][Epoch 1/1][Iter: 50/782]lr(PiecewiseDecay): 0.10000000, top1: 0.01164, top5: 0.05300, CEloss: 7.66997, loss: 7.66997, batch_cost: 1.612058, reader_cost: 0.00026, ips: 39.70105 samples/s, eta: 0:19:40
[2022/05/29 01:11:08] ppcls INFO: [Train][Epoch 1/1][Iter: 60/782]lr(PiecewiseDecay): 0.10000000, top1: 0.01101, top5: 0.05277, CEloss: 7.33901, loss: 7.33901, batch_cost: 1.611258, reader_cost: 0.00025, ips: 39.72068 samples/s, eta: 0:19:23
[2022/05/29 01:11:24] ppcls INFO: [Train][Epoch 1/1][Iter: 70/782]lr(PiecewiseDecay): 0.10000000, top1: 0.01100, top5: 0.05348, CEloss: 7.03868, loss: 7.03868, batch_cost: 1.610988, reader_cost: 0.00024, ips: 39.72741 samples/s, eta: 0:19:07
[2022/05/29 01:11:41] ppcls INFO: [Train][Epoch 1/1][Iter: 80/782]lr(PiecewiseDecay): 0.10000000, top1: 0.01215, top5: 0.05382, CEloss: 6.82127, loss: 6.82127, batch_cost: 1.610538, reader_cost: 0.00023, ips: 39.73856 samples/s, eta: 0:18:50
[2022/05/29 01:11:57] ppcls INFO: [Train][Epoch 1/1][Iter: 90/782]lr(PiecewiseDecay): 0.10000000, top1: 0.01236, top5: 0.05460, CEloss: 6.62333, loss: 6.62333, batch_cost: 1.610578, reader_cost: 0.00022, ips: 39.73739 samples/s, eta: 0:18:34

- 1. 成功截图如下：

XPU2 Kernel Primitive API 模型列表

5.6 曙光开发指南

以下将说明 Paddle 适配曙光相关的开发指南：

- 曙光智算平台-Paddle 源码编译和单测执行：如何曙光曙光智算平台编译 Paddle 源码编译并执行单测。
- Paddle 适配 C86 加速卡详解：详解 Paddle 适配 C86 加速卡。
- Paddle 框架下 ROCm(HIP) 算子单测修复指导：指导 Paddle 框架下 ROCm(HIP) 算子单测修复。

5.6.1 曙光智算平台-Paddle 源码编译和单测执行

由于曙光智算环境下网路受限，直接编译飞桨源码会遇到第三方依赖库下载失败从而导致编译失败的问题。因此在曙光智算环境下进行飞桨源码编译与单测需要以下几个步骤：

第一章节：本地容器编译 Paddle 源码，并进行 Paddle 目录打包

第一步：本地启动编译容器(推荐使用 Paddle 镜像)

```

# 并拉取开发镜像
docker pull paddlepaddle/paddle:latest-dev-rocm4.0-miopen2.11

# 启动容器，注意目录映射按需改成自己环境中的路径
docker run -it --name paddle-dev --shm-size=128G --network=host \
--volume /home/test_user:/workspace --workdir=/workspace \

```

(下页继续)

(续上页)

```
--cap-add=SYS_PTRACE --security-opt seccomp=unconfined \
paddlepaddle/paddle:latest-dev-rocm4.0-miopen2.11 /bin/bash
```

第二步：在容器内进行源码编译，CMAKE 编译选项含义请参见编译选项表

```
# 拉取 Paddle 最新源码到本地目录，默认为 develop 分支
git clone https://github.com/PaddlePaddle/Paddle.git
cd Paddle

# 创建编译目录
mkdir build && cd build

# 执行 cmake
cmake .. -DPY_VERSION=3.7 -DWITH_ROCM=ON -DWITH_TESTING=ON \
-DWITH_DISTRIBUTE=ON -DWITH_MKL=ON

# 使用以下命令来编译
make -j8

# 编译完成之后将整个 Paddle 目录打包
cd /workspace
tar -zcvf Paddle.tar.gz ./Paddle
```

第二章节：将打包的 Paddle 源码包上传曙光智算平台个人目录

第一步：登录曙光智算 平台后进入菜单顶部的「文件 -> E-File」环境



第二步：在 E-File 页面点击文件上传，将压缩的 Paddle 包上传到智算平台的个人目录下

注意：Paddle.tar.gz 包大约 6G 左右大小，建议使用“快传”节省传输时间。

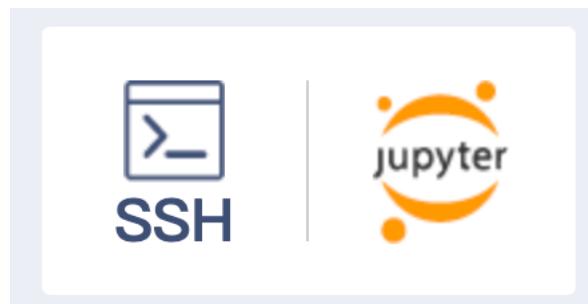
The screenshot shows the 'E-File' upload interface. At the top, it displays '区域 : 华东一区【昆山】 用户 : qili93'. Below this is a toolbar with buttons for '新建', '上传', '快传', '下载', '删除', '复制', and '移动'. The '快传' button is highlighted with a red box. The main area shows a file tree with a path '主目录 | 返回上一级 | 根目录 > pu'. A large red box highlights the '上传文件' (Upload File) button. Below the tree, there are filters for '文件名' (File Name), '文件类型' (File Type), and '大小' (Size).

第三章节：智算平台下启动 Paddle 开发容器，执行编译和单测

第一步：登录曙光智算 平台后进入页面中部的「我的服务 -> 智能计算服务」



第二步：在 AI 服务页面，点击页面底部的「SSH | Jupyter」图标



第三步：在弹出的「实例」页面中点击页面上部的「创建实例」按钮



第四步：在「创建实例」页面中选择和填入如下信息之后，点击「运行」按钮

注意：这里必须选择框架版本为 paddle:latest-dev-rocm4.0-miopen2.11，建议选择 CPU 数量 8, C86 加速卡数量 1，内存 64.0



第五步：在「实例」页面等待容器状态从「等待」转为「运行」后，点击右侧「SSH」按钮

如下图所示，第一行为容器刚创建时状态为「等待」，右侧「SSH」按钮为灰色不可点击；预计 10 分钟左右容器转为第二行的「运行」状态，右侧红框中的「SSH」按钮转为蓝色可点击状态。点击「SSH」按钮后弹出的「WebShell」页面即为刚创建的容器环境。



第六步：在「WebShell」页面中，链接 Paddle 源码目录并重新编译后，即可执行相关算子单测任务

注意：以下步骤目的是为了在容器内创建和本文第一章节中的本地容器一样的编译路径，否则 CMAKE 编译会出错

```
# 先将个人目录下的Paddle源码包进行解压
tar -zxvf Paddle.tar.gz

# 将解压后的源码目录软链接到/workspace/Paddle目录
sudo mkdir /workspace
sudo ln -s ~/Paddle /workspace/Paddle
sudo chown -R $USER:$USER /workspace

# 进入源码编译目录，重新执行编译命令
cd /workspace/Paddle/build && make -j8

# 编译成功之后，在build目录下执行单测
ctest -R test_atan2_op -VV
```

预期得到如下结果，即为编译和单测环境配置成功



计算服务

```
adding 'paddle/vision/transforms/functional_pil.py'
adding 'paddle/vision/transforms/functional_tensor.py'
adding 'paddle/vision/transforms/transforms.py'
adding 'paddlepaddle_rocm-0.0.0.data/scripts/paddle'
adding 'paddlepaddle_rocm-0.0.0.dist-info/METADATA'
adding 'paddlepaddle_rocm-0.0.0.dist-info/WHEEL'
adding 'paddlepaddle_rocm-0.0.0.dist-info/entry_points.txt'
adding 'paddlepaddle_rocm-0.0.0.dist-info/top_level.txt'
adding 'paddlepaddle_rocm-0.0.0.dist-info/RECORD'
removing build/bdist.linux-x86_64/wheel
[100%] Built target paddle_python
[ec12e51395ed:build {develop}] ctest -R test_atan2_op -VV
UpdateCTestConfiguration from :/workspace/Paddle/build/DartConfiguration.tcl
UpdateCTestConfiguration from :/workspace/Paddle/build/DartConfiguration.tcl
Test project /workspace/Paddle/build
Constructing a list of tests
Done constructing a list of tests
Updating test list for fixtures
Added 0 tests to meet fixture requirements
Checking test dependency graph...
Checking test dependency graph end
test 401
    Start 401: test_atan2_op

401: Test command: /opt/cmake-3.16/bin/cmake "-E" "env" "PYTHONPATH=/workspace/Paddle/bu
401: Test timeout computed to be: 10000000
401: W0303 10:18:41.801386 28370 gpu_context.cc:240] Please NOTE: device: 0, GPU Compute
401: /public/home/qili93/Paddle/build/python/paddle/fluid/tests/unittests/op_test.py:43:
by itself. Doing this will not modify any behavior and is safe. If you specifically wan
401: Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
401: np.dtype(np.bool)
1/1 Test #401: test_atan2_op ..... Passed 9.92 sec

The following tests passed:
    test_atan2_op

100% tests passed, 0 tests failed out of 1

Total Test time (real) = 12.36 sec
[ec12e51395ed:build {develop}]
```

✿ 选择文本复制，点击 [这里](#) 或ctrl+shift+v粘贴

5.6.2 Paddle 适配 C86 加速卡详解

当前百度飞桨 PaddlePaddle 已经适配了支持 C86 加速卡的 ROCm 软件栈，并提供了官方 Paddle ROCm 安装包的下载，以及官方 开发与运行镜像 的下载，并完成了 C86 加速卡上 70 个模型的训练与推理任务的支持。

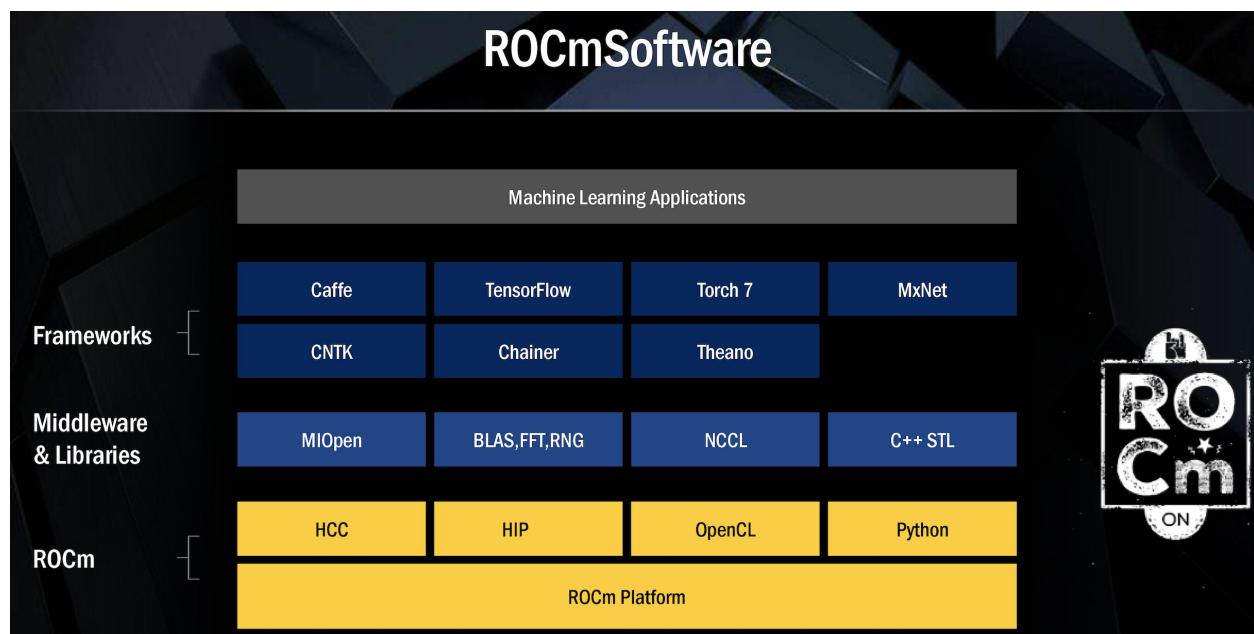
Paddle 适配 ROCm 软件栈

ROCM 软件栈简介

ROCM 软件栈整体架构如下，其中除了支持 C86 加速卡的 Driver/Fireware 之外，对上层应用还提供了一整套较为完整的开发套件，其中与深度学习框架适配相关的最为主要的几个部分包括：

- HIP: 支持异构计算的 C++ Driver/Runtime API，兼容 CUDA 并提供了与 CUDA 多个版本的 [API 对照表](#)
- HIP Kernel: 支持自定义 Kernel 编程，编程语法同 CUDA Kernel 一致，且支持直接编译 CUDA Kernel 源码为 HIP Kernel
- HIPCC: HIP Kernel 编译器，类同于 NVCC 编译器，支持将 CUDA Kernel 或者 HIP Kernel 编译为 ROCm 上的可执行程序
- 加速库、数学库及通讯库：包括 MIOpen, rocBLAS, RCCL 等，分别对标 CUDA 的 cuDNN, cuBLAS, NCCL 等([对照表](#))

ROCM 软件栈本身具备较高的成熟度与完备性，用户根据 ROCM 提供的 CUDA 到 HIP 的[代码移植手册](#)，可以较为方便的将 CUDA 上的工作移植到 HIP 上，使得用户程序可以支持跨 ROCM 与 CUDA 的异构计算。



Paddle 适配 ROCm

受益于 ROCm 本身的特性，Paddle 在适配 ROCm 的过程中可以大量迁移与复用已有的 Paddle 在 CUDA 上的工作。主要的适配开发工作包括以下几个部分：

1. 环境适配：包括 Paddle 基于 ROCm 软件栈的开发编译环境，以及算子注册等功能适配工作
 - `hip.cmake` 配置 HIP 相关编译环境，包括编译所需的头文件、依赖库，以及 CXX / HIPCC 的编译选项等
 - `miopen.cmake` 配置 MIOpen 算子相关编译环境，包括编译所需的 MIOpen 头文件及依赖库等
 - `rccl.cmake` 配置 ROCm 通讯库 RCCL 相关编译环境，主要为 RCCL 编译所需的头文件
 - `generic.cmake` 定义 Paddle 中的 `hip_library` 及 `hip_test` 等 cmake 函数，支持生成 HIP 对象
 - `operators.cmake` 配置 HIP Kernel 算子注册方式，自动映射 CUDA Kernel 算子文件为 HIP Kernel 算子文件
 - 其他相关 cmake 配置，包括依赖的第三方库如 `eigen.cmake` 和 `warpctc.cmake` 等
2. 设备接入：主要包括设备相关的 Driver/Runtime API 的接入，以及通讯库等底层加速库的接入工作
 - 动态库加载：在 `paddle/fluid/platform/dynload` 目录下动态加载 ROCm 加速库及所需 API，如 `hiprand.h` `miopen.h` `rocmblas.h` 等
 - Driver/Runtime 适配：主要在 `paddle/fluid/platform/device/gpu` 目录下对 HIP 和 CUDA 进行了相关 API 的封装，其中在 `gpu_types.h` 少量封装了部分与 CUDA 差异较小的数据类型定义，部分 ROCm 独有代码位于 `paddle/fluid/platform/device/gpu/rocm` 目录
 - Memory 管理：利用上一步封装好的 Driver/Runtime API 对 `memcpy.cc` 与 `paddle/fluid/memory/allocation` 目录下的多种 Memory Allocator 进行实现
 - Device Context 管理：利用封装好的 API 实现对设备上下文的管理及设备池的初始化，位于 `device_context.h`
 - 其他设备管理相关的适配接入，如 Profiler, Tracer, Error Message, NCCL 等，代码主要位于 `Paddle/fluid/platform` 目录下
3. 算子注册：主要包括 HIP Kernel 的算子注册，以及 MIOpen 的算子在 ROCm 平台上的注册
 - 数据类型支持：除通用数据类型外，还需适配 Paddle 支持的特殊数据类型包括 `float16.h` `complex.h` `bfloat16.h` 等
 - 数学库支持：通过 ROCm 的 rocBLAS 库，实现 Paddle 在 `blas.h` 中定义的 BLAS 函数，代码位于 `blas_impl.hip.h`
 - Kernel 算子注册：根据 `operators.cmake` 的修改，可以大部分复用 Paddle 框架下的 CUDA 已有算子 Kernel 文件，存在部分 Kernel 实现在 CUDA 与 ROCm 平台下有所区别，例如线程数、WarpSize 以及 thurst 库等；此类区别需要针对具体的算子实现进行相应的调整，通过 Paddle 自身的算子单测用例以及模型验证测试可以对此类问题进行定位并修复

- MIOpen 算子注册：MIOpen 与 cuDNN 的接口与类型设计较为类似，但在实际执行中还是存在一定区别，因为对于此类算子需根据 MIOpen API 进行适配，甚至对于差异较大的算子例如 `rnn_op.cu.cc` 需要进行 weight 数据重排
4. Python API 兼容适配：当前 Paddle ROCm 兼容所有 Paddle CUDA 相关的 Python API，这意味着对于所有目前 Paddle 可以支持 CUDA 的模型，无需修改任意代码可以直接运行在 C86 加速卡上

经过以上几个步骤的适配工作，用户可以无需修改任意代码就将之前在 Paddle CUDA 平台上的程序运行在 C86 加速卡的环境下，在保持用户已有的基于 Paddle CUDA 的编程习惯的同时，也减少了 Paddle 已有的模型套件在 CUDA 平台与 ROCm 平台之间的迁移工作。

例如以下这份代码可以同时运行于 Paddle 的 CUDA 环境和 C86 加速卡环境，且输出结果一致：

```
import paddle
import paddle.nn.functional as F
import numpy as np

paddle.set_device("gpu") # 设置运行在 CUDA 或 ROCm 上

x = np.arange(24).reshape(2, 3, 4).astype(np.float32)
x = paddle.to_tensor(x)
out = F.softmax(x)

# Nvidia V100 输出如下
W1102 10:17:50.512131 16379 device_context.cc:447] Please NOTE: device: 0, GPU ↵
Compute Capability: 7.0, Driver API Version: 11.2, Runtime API Version: 10.1
W1102 10:17:50.522437 16379 device_context.cc:465] device: 0, cuDNN Version: 7.6.
Tensor(shape=[2, 3, 4], dtype=float32, place=CUDAPlace(0), stop_gradient=True,
[[[0.03205860, 0.08714432, 0.23688284, 0.64391428],
 [0.03205860, 0.08714432, 0.23688284, 0.64391428],
 [0.03205860, 0.08714432, 0.23688284, 0.64391428]],

 [[0.03205860, 0.08714432, 0.23688284, 0.64391428],
 [0.03205860, 0.08714432, 0.23688284, 0.64391428],
 [0.03205860, 0.08714432, 0.23688284, 0.64391428]]))

# C86加速卡 输出如下
W1102 10:06:45.729085 875 device_context.cc:447] Please NOTE: device: 0, GPU ↵
Compute Capability: 90.0, Driver API Version: 321.0, Runtime API Version: 3.1
W1102 10:06:45.733167 875 device_context.cc:460] device: 0, MIOpen Version: 2.11.0
Tensor(shape=[2, 3, 4], dtype=float32, place=CUDAPlace(0), stop_gradient=True,
[[[0.03205860, 0.08714432, 0.23688284, 0.64391428],
 [0.03205860, 0.08714432, 0.23688284, 0.64391428],
 [0.03205860, 0.08714432, 0.23688284, 0.64391428]],

 [[0.03205860, 0.08714432, 0.23688284, 0.64391428],
```

(下页继续)

(续上页)

```
[0.03205860, 0.08714432, 0.23688284, 0.64391428],
[0.03205860, 0.08714432, 0.23688284, 0.64391428]]])
```

5.6.3 Paddle 框架下 ROCm(HIP) 算子单测修复指导

进行 ROCm(HIP) 算子修复之前, 请先仔细阅读 曙光智算平台-Paddle 源码编译和单测执行 并按照其中步骤准备好编译和单测环境。并阅读 Paddle 适配 C86 加速卡详解 文档了解当前 Paddle 与 ROCm(HIP) 的适配方案和具体的代码修改。

常见的 HIP 算子问题已经修复办法如下, 也可以在PaddlePR中搜索 [ROCM] 关键字查看 ROCm(HIP) 相关的代码修改, 更多问题请自行探索解决方法。

注: 打开 Paddle Debug Level 日志可以参考 Contribute Code 中的 Writing Logs 章节。

算子修复举例 1：无法找到对应算子的 GPU Kernel

这类问题常由相应的算子没有在 HIP 下注册成功引起, 首先根据报错信息中提示的算子名确认该算子的 Kernel 情况。执行以下步骤, 并观察输出结果中的 data_type 以及 place 的结果:

```
# 将以下路径输出到PYTHONPATH环境变量
export PYTHONPATH=/workspace/Paddle/build/python:$PYTHONPATH

# 执行如下命令打印算子的Kernel列表 - 将其中的xxx改为真正的算子名即可
python -c '$import paddle\nfor k in paddle.fluid.core._get_all_register_op_kernels() [
    ↪"XXX"] :print(k)'

# 例如如下输出, 表示存在数据类型为float, 算子硬件类型为GPU的Kernel
# 通常会有多行类似以下结果的输出, 请根据输出仔细分析算子Kernel结果
data_type[float]:data_layout[Undefined(AnyLayout)]:place[Place(gpu:0)]:library_
↪type[PLAIN]
```

情况 1：整个算子只有 CPU Kernel, 没有任何 GPU Kernel

例如如下输出表示: 算子只存在 Place 为 CPU 的 Kernel, 不存在任何 Place 为 GPU 的 Kernel

```
# 错误提示信息如下:
928:     NotFoundError: Operator (lu) does not have kernel for data_type[float]:data_
↪layout[Undefined(AnyLayout)]:place[Place(gpu:0)]:library_type[PLAIN].
928:         [Hint: Expected kernel_iter != kernels.end(), but received kernel_iter ==_
↪kernels.end().] (at /workspace/Paddle/paddle/fluid/framework/operator.cc:1503)
928:         [operator < lu > error]
```

(下页继续)

(续上页)

```
# 打印算子名对应的Kernel列表如下
data_type[double]:data_layout[Undefined(AnyLayout)]:place[Place(cpu)]:library_
→type[PLAIN]
data_type[float]:data_layout[Undefined(AnyLayout)]:place[Place(cpu)]:library_
→type[PLAIN]
```

这个原因通常是由于算子的 GPU Kernel 源码文件没有加入编译目标引起的。修复办法是先在 `operators.cmake` 中查看改算子的源码是否被移除，如下图代码所示：

```
245 elseif (WITH_ROCM)
246     list(REMOVE_ITEM miopen_cu_cc_srcs "affine_grid_cudnn_op.cu.cc")
247     list(REMOVE_ITEM miopen_cu_cc_srcs "grid_sampler_cudnn_op.cu.cc")
248     list(REMOVE_ITEM hip_srcs "cholesky_op.cu")
249     list(REMOVE_ITEM hip_srcs "cholesky_solve_op.cu")
250     list(REMOVE_ITEM hip_srcs "lu_op.cu")
251     list(REMOVE_ITEM hip_srcs "matrix_rank_op.cu")
252     list(REMOVE_ITEM hip_srcs "svd_op.cu")
```

移除的原因可以打开具体的源码文件进行查看，例如打开 `lu_op.cu`，可见如下结果：

```
14
15 #ifndef PADDLE_WITH_HIP
16 // HIP not support cusolver
17
18 #include "paddle/fluid/memory/memory.h"
19 #include "paddle/fluid/operators/lu_op.h"
20 #include "paddle/fluid/platform/dynload/cusolver.h"
```

根据注释，是由于初始适配时 ROCm 下的 rocSolver 库未曾适配导致的，需参考 `cuSovler` 代码以及 `hipSOLVER` 中 `rocSovler` 和 `cuSovler` 的 API 封装示例修改代码使改算子可以在 HIP 环境下正确运行。

情况 2：算子 GPU Kernel 存在，少了某个 LibraryType 下的 Kernel

这类问题的常见报错信息如下：

例如如下输出表示：算子只存在 GPU Kernel，但是只有 PLAIN 没有 CUDNN 类型实现

```
# 错误提示信息如下：
1000:      NotFoundError: Operator (pool2d_grad_grad) does not have kernel for data_
→type[float]:data_layout[Undefined(AnyLayout)]:place[Place(gpu:0)]:library_
→type[CUDNN].
1000:      [Hint: Expected kernel_iter != kernels.end(), but received kernel_iter ==_
→kernels.end()..] (at /workspace/Paddle/paddle/fluid/framework/operator.cc:1503)
```

(下页继续)

(续上页)

```
1000: [operator < pool2d_grad_grad > error]

# 打印算子名对应的Kernel列表如下
data_type[double]:data_layout[Undefined(AnyLayout)]:place[Place(gpu:0)]:library_
↪type[PLAIN]
data_type[::paddle::platform::float16]:data_
↪layout[Undefined(AnyLayout)]:place[Place(gpu:0)]:library_type[PLAIN]
data_type[float]:data_layout[Undefined(AnyLayout)]:place[Place(cpu)]:library_
↪type[PLAIN]
data_type[float]:data_layout[Undefined(AnyLayout)]:place[Place(gpu:0)]:library_
↪type[PLAIN]
data_type[double]:data_layout[Undefined(AnyLayout)]:place[Place(cpu)]:library_
↪type[PLAIN]
```

查看对应算子源码文件 `pool_cudnn_op.cu.cc` 可知对应的 `pool2d_grad_grad` 只在 CUDA 平台下注册了 CUDNN 的 `pool2d_grad_grad` 算子，但是没有在 HIP 平台下注册，因此修改代码在 HIP 平台下进行注册即可。

```

532 #ifdef PADDLE_WITH_HIP
533 // MIOPEN do not support double
534 REGISTER_OP_KERNEL(pool2d, CUDNN, plat::CUDAPlace,
535                     ops::PoolCUDNNOpKernel<float>,
536                     ops::PoolCUDNNOpKernel<plat::float16>);
537 REGISTER_OP_KERNEL(pool2d_grad, CUDNN, plat::CUDAPlace,
538                     ops::PoolCUDNNGradOpKernel<float>,
539                     ops::PoolCUDNNGradOpKernel<plat::float16>);
540
541 REGISTER_OP_KERNEL(pool3d, CUDNN, plat::CUDAPlace,
542                     ops::PoolCUDNNOpKernel<float>,
543                     ops::PoolCUDNNOpKernel<plat::float16>);
544 REGISTER_OP_KERNEL(pool3d_grad, CUDNN, plat::CUDAPlace,
545                     ops::PoolCUDNNGradOpKernel<float>);
546 #else
547 REGISTER_OP_KERNEL(pool2d, CUDNN, plat::CUDAPlace,
548                     ops::PoolCUDNNOpKernel<float>,
549                     ops::PoolCUDNNOpKernel<double>,
550                     ops::PoolCUDNNOpKernel<plat::float16>);
551 REGISTER_OP_KERNEL(pool2d_grad, CUDNN, plat::CUDAPlace,
552                     ops::PoolCUDNNGradOpKernel<float>,
553                     ops::PoolCUDNNGradOpKernel<double>,
554                     ops::PoolCUDNNGradOpKernel<plat::float16>);
555 REGISTER_OP_KERNEL(pool2d_grad_grad, CUDNN, plat::CUDAPlace,
556                     ops::PoolCUDNNGradGradOpKernel<float>,
557                     ops::PoolCUDNNGradGradOpKernel<double>,
558                     ops::PoolCUDNNGradGradOpKernel<plat::float16>);

```

情况 3：存在算子的 GPU Kernel，只是少了某几个数据类型

例如如下输出表示：存在 Place 为 GPU 的 GPU Kernel，只是少了数据类型 bfloat16

```

The following tests FAILED:
  364 - test_activation_op (Failed)
Errors while running CTest
[ 3@ec12e51395ed:build {develop} python -c '$import paddle\nfor k in paddle.fluid.core._get_all_register_op_ke
data_type[double]:data_layout[Undefined(AnyLayout)]:place[Place(gpu:0)]:library_type[PLAIN]
data_type[float]:data_layout[Undefined(AnyLayout)]:place[Place(gpu:0)]:library_type[PLAIN]
data_type[double]:data_layout[Undefined(AnyLayout)]:place[Place(cpu)]:library_type[PLAIN]
data_type[:paddle::platform::float16]:data_layout[Undefined(AnyLayout)]:place[Place(gpu:0)]:library_type[PLAIN]
data_type[:paddle::platform::bfloating16]:data_layout[MKLDNN]:place[Place(cpu)]:library_type[MKLDNN]
data_type[float]:data_layout[Undefined(AnyLayout)]:place[Place(cpu)]:library_type[PLAIN]
data_type[float]:data_layout[Undefined(AnyLayout)]:place[Place(gpu:0)]:library_type[CUDNN]
data_type[float]:data_layout[MKLDNN]:place[Place(cpu)]:library_type[MKLDNN]
[ 3@ec12e51395ed:build {develop} ]

```

这个原因通常是由于算子的 GPU Kernel 源码文件中没有注册对应数据类型引起的，修复办法是先在以下两

个算子 Kernel 源码目录中通过查找 REGISTER_OP_CUDA_KERNEL 关键字找到对应算子的源码文件中注册算子 Kernel 的代码。

```
Paddle 当前算子 Kernel 目录主要位于如下两个目录中  
cd /workspace/Paddle/paddle/fluid/operators  
cd /workspace/Paddle/paddle/phi/kernels
```

例如查找得到的 relu 算子的注册代码如下：

```
1597  /* ===== relu register ===== */  
1598  #ifdef PADDLE_WITH_HIP  
1599  REGISTER_ACTIVATION_CUDA_KERNEL(relu, Relu, CudaReluFunctor,  
1600      CudaReluGradFunctor);  
1601  REGISTER_OP_CUDA_KERNEL(  
1602      relu_grad_grad,  
1603      ops::ActivationDoubleGradKernel<paddle::platform::CUDADeviceContext,  
1604          ops::ReluGradGradFunctor<float>>,  
1605      ops::ActivationDoubleGradKernel<paddle::platform::CUDADeviceContext,  
1606          ops::ReluGradGradFunctor<double>>,  
1607      ops::ActivationDoubleGradKernel<plat::CUDADeviceContext,  
1608          ops::ReluGradGradFunctor<plat::float16>>);  
1609  #else  
1610  REGISTER_OP_CUDA_KERNEL(  
1611      relu, ops::ActivationCudaKernel<paddle::platform::CUDADeviceContext,  
1612          ops::CudaReluFunctor<float>>,  
1613      ops::ActivationCudaKernel<paddle::platform::CUDADeviceContext,  
1614          ops::CudaReluFunctor<double>>,  
1615      ops::ActivationCudaKernel<plat::CUDADeviceContext,  
1616          ops::CudaReluFunctor<plat::float16>>,  
1617      ops::ActivationCudaKernel<plat::CUDADeviceContext,  
1618          ops::CudaReluFunctor<plat::bfloat16>>);
```

注意观察其中用 PADDLE_WITH_HIP 的宏定义包围的代码才是 C86 加速卡相关算子，其中 REGISTER_ACTIVATION_CUDA_KERNEL 的定义如下，只为算子的前反向定义了 float,double,float16 三种数据类型，缺少错误提示中所说的 bfloat16 数据类型

```

1504 #define REGISTER_ACTIVATION_CUDA_KERNEL(act_type, op_name, functor, \
1505                                     grad_functor) \
1506     REGISTER_OP_CUDA_KERNEL( \
1507         act_type, ops::ActivationCudaKernel<paddle::platform::CUDADeviceContext, \
1508                                         ops::functor<float>>, \
1509                                         ops::ActivationCudaKernel<paddle::platform::CUDADeviceContext, \
1510                                         ops::functor<double>>, \
1511                                         ops::ActivationCudaKernel<plat::CUDADeviceContext, \
1512                                         ops::functor<plat::float16>>); \
1513     REGISTER_OP_CUDA_KERNEL( \
1514         act_type##_grad, \
1515         ops::ActivationGradCudaKernel<plat::CUDADeviceContext, \
1516                                         ops::grad_functor<float>>, \
1517                                         ops::ActivationGradCudaKernel<plat::CUDADeviceContext, \
1518                                         ops::grad_functor<double>>, \
1519                                         ops::ActivationGradCudaKernel<plat::CUDADeviceContext, \
1520                                         ops::grad_functor<plat::float16>>); \
1521

```

因此可以参考非 PADDLE_WITH_HIP 的宏定义包围的英伟达 GPU 相关代码，为 HIP 算子注册 bfloat16 数据类型。之后再在 HIP 环境下验证该算子的正确输出结果。

算子修复举例 2：单测的输出结果无法达到精度对齐

这类问题造成的原因较为多样，请先仔细阅读单测的报错信息，可能存在如下几种情况

情况 1：输出误差较小，则相应修改单测文件的误差阈值即可

这类问题的常见报错信息如下：

```

1066:   File "/public/home/qili93/Paddle/build/python/paddle/fluid/tests/unittests/ \
1066:   ↪test_poisson_op.py", line 60, in verify_output \
1066:     "actual: {}, expected: {}".format(hist, prob)) \
1066: AssertionError: False is not true : actual: [0.03375816 0.08399963 0.13975811 0. \
1066:   ↪17509079 0.17573357 0.14692497 \
1066:   ↪0.10456944 0.06567383 0.03586864], expected: [0.03368973499542734, 0. \
1066:   ↪0.08422433748856833, 0.14037389581428056, 0.1754673697678507, 0.1754673697678507, 0. \
1066:   ↪0.1462228081398756, 0.104444862957054, 0.06527803934815875, 0.03626557741564375]

```

从输出中可以观察到，actual 和 expected 结果较为接近，可以通过修改误差阈值来解决：

```

# 例如原有误差阈值为 0.01
self.assertTrue(np.allclose(hist, prob, rtol=0.01),
               "actual: {}, expected: {}".format(hist, prob))

# 将其修改为新的误差阈值如 0.05

```

(下页继续)

(续上页)

```
self.assertTrue(np.allclose(hist, prob, rtol=0.05),
               "actual: {}, expected: {}".format(hist, prob))
```

情况 2：输出误差较大，需要定位误差是代码实现导致还是硬件本身原因

这类问题的常见报错信息如下：

```
# 示例 2
373: AssertionError:
373: Not equal to tolerance rtol=1e-06, atol=0
373:
373: Mismatched elements: 1 / 1 (100%)
373: Max absolute difference: 6.5650682
373: Max relative difference: 3.80200248
373:   x: array([4.838329], dtype=float32)
373:   y: array(-1.72674)
```

从输出中观察到，此类算子误差非常大，可能是算子本身计算代码在 HIP 平台下存在问题。建议仔细调试该算子的 GPU Kernel，定位算子计算问题并进行修复。

情况 3：输出结果中出 Nan，需要定位算子内核函数的实现问题

这类问题的常见报错信息如下：

```
# 示例 3
356: AssertionError: False is not true : Output (Out) has diff at Place(gpu:0)
356: Expect [[[ 0.3687 -0.0764  0.1682  0.3389 -0.4622 ] ...
356: But Got[[[[nan nan nan nan nan] ...
```

从输出中观察到，算子输出直接出 nan 了，可能是算子本身计算代码在 HIP 平台下存在问题。同上个问题一样，需要仔细调试该算子的 GPU Kernel，定位算子计算问题并进行修复。可能的解决办法是请先检查对应算子 Kernel 的线程数，可以参考 [ROCm-Developer-Tools/HIP#2235](#) 中的回复，将 HIP 平台下的算子线程数控制在 256 及以内。

5.7 自定义新硬件接入指南

自定义硬件接入功能实现框架和硬件的解耦，提供一种插件式扩展 PaddlePaddle 硬件后端的方式。通过该功能，开发者无需为特定硬件修改 PaddlePaddle 代码，只需实现标准接口，并编译成动态链接库，则可作为插件供 PaddlePaddle 调用。降低为 PaddlePaddle 添加新硬件后端的开发难度。

自定义硬件接入功能由自定义 Runtime 与自定义 Kernel 两个主要组件构成，基于这两个组件，用户可按需完成自定义新硬件接入飞桨。

- [自定义 Runtime](#)：飞桨框架自定义 Runtime 介绍
- [自定义 Kernel](#)：飞桨框架自定义 Kernel 介绍
- [新硬件接入示例](#)：通过示例介绍自定义新硬件接入飞桨的步骤

5.7.1 自定义 Runtime

Title overline too short.

```
#####
# 定义 Runtime
#####
```

自定义 Runtime 为 PaddlePaddle 提供了一种插件式注册新硬件 Runtime 的方式。DeviceManager 管理 PaddlePaddle 的硬件设备以及 Runtime/Driver 接口，向上提供统一的接口供框架调用硬件功能，向下暴露一系列接口用于注册自定义 Runtime，通过 C API 形式保证二进制兼容性。这些接口可以在 `device_ext.h` 文件中查看，开发者只需要实现这些接口即可为 PaddlePaddle 添加自定义 Runtime。

- [数据类型](#)：介绍自定义 Runtime 的数据类型定义。
- [Device 接口](#)：介绍 Device 接口的定义和功能。
- [Memory 接口](#)：介绍 Memory 接口的定义和功能。
- [Stream 接口](#)：介绍 Stream 接口的定义和功能。
- [Event 接口](#)：介绍 Event 接口的定义和功能。

Device 接口

接口名称	功能简介
initialize	初始化硬件后端。
finalize	去初始化硬件后端。
init_device	初始化指定硬件设备。
deinit_device	去初始化指定硬件设备。
set_device	设置当前使用的硬件设备。
get_device	获取当前使用的硬件设备。
synchronize_device	同步指定的硬件设备。
get_device_count	查询可用设备数量。
get_device_list	查询可用设备号。
get_compute_capability	查询设备算力。
get_runtime_version	查询运行时版本号。
get_driver_version	查询驱动版本号。

Memory 接口

接口名称	功能简介
device_memory_allocate	分配设备内存。
device_memory_deallocate	释放设备内存。
host_memory_allocate	分配主机锁页内存。
host_memory_deallocate	释放主机锁页内存。
unified_memory_allocate	分配统一地址空间内存。
unified_memory_deallocate	释放统一地址空间内存。
memory_copy_h2d	主机到设备的同步内存拷贝。
memory_copy_d2h	设备到主机的同步内存拷贝。
memory_copy_d2d	设备内同步内存拷贝。
memory_copy_p2d	设备间同步内存拷贝。
async_memory_copy_h2d	主机到设备异步内存拷贝。
async_memory_copy_d2h	设备到主机异步内存拷贝。
async_memory_copy_d2d	设备内异步内存拷贝。
async_memory_copy_p2d	设备间异步内存拷贝。
device_memory_set	填充设备内存。
device_memory_stats	设备内存使用统计。
device_min_chunk_size	查询设备内存最小块大小。
device_max_chunk_size	查询设备内存最大块大小。
device_max_alloc_size	查询设备最大可分配内存大小。
device_extra_padding_size	查询设备内存额外填充大小。
device_init_alloc_size	查询设备初始化分配内存大小。
device_realloc_size	查询设备重分配内存大小。

Stream 接口

接口名称	功能简介
create_stream	创建一个 stream 对象。
destroy_stream	销毁一个 stream 对象。
query_stream	查询 stream 上任务是否完成。
synchronize_stream	同步 stream，等待 stream 上所有任务完成。
stream_add_callback	添加一个主机回调到 stream 上。
stream_wait_event	等待 stream 上的一个 event 完成。

Event 接口

接口名称	功能简介
create_event	创建一个 event 对象。
destroy_event	销毁一个 event 对象。
record_event	在 stream 上记录 event。
query_event	查询 event 是否完成。
synchronize_event	同步 event，等待 event 完成。

数据类型

C_Status

类型定义

```
typedef enum {
    C_SUCCESS = 0,
    C_WARNING,
    C_FAILED,
    C_ERROR,
    C_INTERNAL_ERROR
} C_Status;
```

说明

C_SUCCESS - 函数执行成功时返回值。

C_WARNING - 函数功能可能不符合预期时返回值，例如异步接口实际是同步。

C_FAILED - 资源耗尽或请求失败。

C_ERROR - 参数错误，用法错误或未初始化。

C_INTERNAL_ERROR - 插件内部错误。

C_Device

类型定义

```
typedef struct C_Device_st { int id; } * C_Device;
```

说明

描述一个 device 对象。

C_Stream

类型定义

```
typedef struct C_Stream_st* C_Stream;
```

说明

描述一个 stream 对象， stream 是框架内部用于执行异步任务的任务队列，同一 stream 中的任务按顺序执行。

C_Event

类型定义

```
typedef struct C_Event_st* C_Event;
```

说明

描述一个 event 对象， event 被框架内部用于同步不同 stream 之间的任务。

C_Callback

类型定义

```
typedef void (*C_Callback)(C_Device device,
                           C_Stream stream,
                           void* user_data,
                           C_Status* status);
```

说明

主机回调函数类型，具有 4 个参数，使用的设备，使用的 stream，用户数据，以及返回值。

CustomRuntimeParams

类型定义

```
struct CustomRuntimeParams {
    size_t size;
    C_DeviceInterface* interface;
    CustomRuntimeVersion version;
    char* device_type;
    char* sub_device_type;
    char reserved[32];
};
```

说明

插件入口函数 InitPlugin 的参数类型。

size - CustomRuntimeParams 的大小，框架和插件 CustomRuntimeParams 类型大小可能不一致，插件首先需要检查该大小，确保内存访问不会越界。可使用 PADDLE_CUSTOM_RUNTIME_CHECK_VERSION 宏完成检查。

interface - 设备回调接口，插件需要实现必要的接口，并填充该参数完成注册。

version - 使用 device_ext.h 头文件中定义的自定义 Runtime 版本填充，用于框架检查版本兼容性。

device_type - 设备类型名，用于框架区分设备，同时暴露到用户层，用于指定硬件后端，例如”CustomCPU”。

sub_device_type - 子设备类型名，可以用于说明插件版本，例如”V1.0”。

CustomRuntimeVersion

类型定义

```
struct CustomRuntimeVersion {
    size_t major, minor, patch;
};
```

说明

插件使用的自定义 Runtime 的版本号，用于框架检查版本兼容性。可使用 PAD-DLE_CUSTOM_RUNTIME_CHECK_VERSION 宏完成填充。

C_DeviceInterface

类型定义

C_DeviceInterface 的类型定义详见 device_ext.h。

说明

自定义 Runtime 回调接口集合。

Device 接口

initialize 【optional】

接口定义

```
C_Status (*initialize)()
```

接口说明

初始化硬件后端，例如初始化硬件 Runtime 或者 Driver。在注册硬件时，最先被调用，不实现该接口则不调用。

finalize 【optional】

接口定义

```
C_Status (*finalize)()
```

接口说明

去初始化硬件后端，例如硬件 Runtime 或者 Driver 退出时去初始化。在退出时最后被调用，不实现该接口则不调用。

init_device 【optional】

接口定义

```
C_Status (*init_device)(const C_Device device)
```

接口说明

初始化指定硬件设备，会在插件注册时对所有可用设备进行初始化，不实现该接口则不调用。该接口在 initialize 之后被调用。

参数

device - 需要初始化的设备。

deinit_device 【optional】

接口定义

```
C_Status (*deinit_device)(const C_Device device)
```

接口说明

去初始化指定硬件设备，释放所有该设备分配的资源，在退出时调用，不实现该接口则不调用。该接口在 finalize 之前被调用。

参数

device - 需要去初始化的设备。

接口定义

set_device [required]

```
C_Status (*set_device) (const C_Device device)
```

接口说明

设置当前使用的硬件设备，后续的任务执行在该设备上。

参数

device - 需要设置的设备。

get_device [required]

接口定义

```
C_Status (*get_device) (const C_Device device)
```

接口说明

获取当前使用的硬件设备。

参数

device - 存储当前使用的设备。

synchronize_device [required]

接口定义

```
C_Status (*synchronize_device)(const C_Device device)
```

接口说明

同步设备，等待指定设备上所有任务完成。

参数

device - 需要同步的设备。

get_device_count [required]

接口定义

```
C_Status (*get_device_count)(size_t* count)
```

接口说明

查询可用设备数量。

参数

count - 存储可用设备数量。

get_device_list 【required】

接口定义

```
C_Status (*get_device_list) (size_t* devices)
```

接口说明

获取当前可用所有设备的设备号列表。

参数

devices - 存储可用设备号。

get_compute_capability 【required】

接口定义

```
C_Status (*get_compute_capability) (size_t* compute_capability)
```

接口说明

获取设备算力。

参数

compute_capability - 存储设备算力。

get_runtime_version 【required】

接口定义

```
C_Status (*get_runtime_version) (size_t* version)
```

接口说明

获取运行时版本号。

参数

version - 存储运行时版本号。

get_driver_version [required]

接口定义

```
C_Status (*get_driver_version) (size_t* version)
```

接口说明

获取驱动版本号。

参数

version - 存储驱动版本号。

Memory 接口

device_memory_allocate [required]

接口定义

```
C_Status (*device_memory_allocate) (const C_Device device, void** ptr, size_t size)
```

接口说明

分配设备内存。

参数

device - 使用的设备。

ptr - 存储分配的设备内存地址。

size - 需要分配的设备内存大小 (字节形式)。

device_memory_deallocate 【required】

接口定义

```
C_Status (*device_memory_deallocate) (const C_Device device, void* ptr, size_t size)
```

接口说明

释放设备内存。

参数

device - 使用的设备。

ptr - 需要释放的设备内存地址。

size - 需要释放的设备内存大小 (字节形式)。

host_memory_allocate 【optional】

接口定义

```
C_Status (*host_memory_allocate) (const C_Device device, void** ptr, size_t size)
```

接口说明

分配主机锁页内存。

参数

device - 使用的设备。

ptr - 存储分配的主机内存地址。

size - 需要分配的内存大小（字节形式）。

host_memory_deallocate 【optional】

接口定义

```
C_Status (*host_memory_deallocate)(const C_Device device, void* ptr, size_t size)
```

接口说明

释放主机锁页内存。

参数

device - 使用的设备。

ptr - 需要释放的主机内存地址。

size - 需要释放的内存大小（字节形式）。

unified_memory_allocate 【optional】

接口定义

```
C_Status (*unified_memory_allocate)(const C_Device device, void** ptr, size_t size)
```

接口说明

分配统一地址空间内存。

参数

device - 使用的设备。

ptr - 存储分配的统一地址空间内存地址。

size - 需要分配内存的大小（字节形式）。

unified_memory_deallocate [optional]

接口定义

```
C_Status (*unified_memory_deallocate) (const C_Device device, void** ptr, size_t size)
```

接口说明

释放统一地址空间内存。

参数

device - 使用的设备。

ptr - 需要释放的统一地址空间内存地址。。

size - 需要释放的内存大小（字节形式）。

memory_copy_h2d [required]

接口定义

```
C_Status (*memory_copy_h2d) (const C_Device device, void* dst, const void* src, size_t  
size)
```

接口说明

主机到设备的同步内存拷贝。

参数

device - 使用的设备。

dst - 目的设备内存地址。

src - 源主机内存地址。

size - 需要拷贝的内存大小（字节形式）。

memory_copy_d2h [required]

接口定义

```
C_Status (*memory_copy_d2h)(const C_Device device, void* dst, const void* src, size_t  
                           ↵size)
```

接口说明

设备到主机的同步内存拷贝。

参数

device - 使用的设备。

dst - 目的主机内存地址。

src - 源设备内存地址。

size - 需要拷贝的内存大小（字节形式）。

memory_copy_d2d [required]

接口定义

```
C_Status (*memory_copy_d2d)(const C_Device device, void* dst, const void* src, size_t  
                           ↵size)
```

接口说明

设备内同步内存拷贝。

参数

device - 使用的设备。

dst - 目的设备内存地址。

src - 源设备内存地址。

size - 需要拷贝的内存大小 (字节形式)。

memory_copy_p2p [optional]

接口定义

```
C_Status (*memory_copy_p2p) (const C_Device dst_device, const C_Device src_device,  
    void* dst, const void* src, size_t size)
```

接口说明

设备间同步内存拷贝。

参数

dst_device - 目的设备。

src_device - 源设备。

dst - 目的设备内存地址。

src - 源设备内存地址。

size - 需要拷贝的内存大小 (字节形式)。

async_memory_copy_h2d 【optional】

接口定义

```
C_Status (*async_memory_copy_h2d) (const C_Device device, C_Stream stream, void* dst,_
                                     const void* src, size_t size)
```

接口说明

主机到设备的异步内存拷贝，如果没有实现，PaddlePaddle 会用同步接口代替。

参数

device - 使用的设备。

stream - 在该 stream 上执行。

dst - 目的设备内存地址。

src - 源主机内存地址。

size - 需要拷贝的内存大小（字节形式）。

async_memory_copy_d2h 【optional】

接口定义

```
C_Status (*async_memory_copy_d2h) (const C_Device device, C_Stream stream, void* dst,_
                                     const void* src, size_t size)
```

接口说明

设备到主机的异步内存拷贝，如果没有实现，PaddlePaddle 会用同步接口代替。

参数

device - 使用的设备。

stream - 在该 stream 上执行。

dst - 目的主机内存地址。

src - 源设备内存地址。

size - 需要拷贝的内存大小。

async_memory_copy_d2d 【optional】

接口定义

```
C_Status (*async_memory_copy_d2d) (const C_Device device, C_Stream stream, void* dst, const void* src, size_t size)
```

接口说明

设备内异步内存拷贝，如果没有实现，PaddlePaddle 会用同步接口代替。

参数

device - 使用的设备。

stream - 使用的 stream。

dst - 目的设备内存地址。

src - 源设备内存地址。

size - 需要拷贝的内存大小（字节形式）。

async_memory_copy_p2p 【optional】

接口定义

```
C_Status (*async_memory_copy_p2p) (const C_Device dst_device, const C_Device src_device, C_Stream stream, void* dst, const void* src, size_t size)
```

接口说明

设备间异步内存拷贝，如果没有实现，PaddlePaddle 会用同步接口代替。

参数

dst_device - 目的设备。

src_device - 源设备。

stream - 使用的 stream。

dst - 目的设备内存地址。

src - 源设备内存地址。

size - 需要拷贝的内存大小（字节形式）。

device_memory_set [optional]

接口定义

```
C_Status (*device_memory_set)(const C_Device device, void* ptr, unsigned char value,  
                           size_t size)
```

接口说明

使用 value 填充一块设备内存，如果没有实现，PaddlePaddle 会用 memory_copy_h2d 代替。

参数

device - 使用的设备。

ptr - 填充地址。

value - 填充值。

size - 填充大小（字节形式）。

device_memory_stats [required]

接口定义

```
C_Status (*device_memory_stats)(const C_Device device, size_t* total_memory, size_t*  
    ↪free_memory)
```

接口说明

设备内存使用统计。

参数

device - 使用的设备。

total_memory - 总内存（字节形式）。

free_memory - 剩余可用内存（字节形式）。

device_min_chunk_size [required]

接口定义

```
C_Status (*device_min_chunk_size)(C_Device device, size_t* size)
```

接口说明

获取设备内存的最小块大小（字节形式）。为避免频繁调用硬件 API 申请/释放内存，PaddlePaddle 会自行管理设备内存，申请内存时优先从 PaddlePaddle 管理的内存中分配。申请 size 大小的内存时，会分配 size + extra_padding_size 大小的内存，并按 min_chunk_size 对齐。

参数

device - 使用的设备。

size - 最小块的大小（字节形式）。

device_max_chunk_size 【optional】

接口定义

```
C_Status (*device_max_chunk_size)(C_Device device, size_t* size)
```

接口说明

PaddlePaddle 管理的设备内存一次最多分配该大小（字节形式），超过该大小时，将直接调用硬件 API 进行分配，如果没有实现，则大小等于 device_max_alloc_size。

参数

device - 使用的设备。

size - 最大块的大小（字节形式）。

device_max_alloc_size 【optional】

接口定义

```
C_Status (*device_max_alloc_size)(C_Device device, size_t* size)
```

接口说明

设备最多可分配的内存大小（字节形式），如果没有实现，则大小等于目前可用的内存。

参数

device - 使用的设备。

size - 最多可分配的内存的大小（字节形式）。

device_extra_padding_size 【optional】

接口定义

```
C_Status (*device_extra_padding_size)(C_Device device, size_t* size)
```

接口说明

分配设备内存需要的额外填充字节，如果没有实现，则默认为 0。为避免频繁调用硬件 API 申请/释放内存，PaddlePaddle 会自行管理设备内存，申请内存时优先从 PaddlePaddle 管理的内存中分配。申请 size 大小的内存时，会分配 size + extra_padding_size 大小的内存，并按 min_chunk_size 对齐。

参数

device - 使用的设备。

size - 额外填充的大小（字节形式）。

device_init_alloc_size 【optional】

接口定义

```
C_Status (*device_init_alloc_size)(const C_Device device, size_t* size)
```

接口说明

设备初始分配的内存大小（字节形式），如果没有实现，则大小等于 device_max_alloc_size。

参数

device - 使用的设备。

size - 初始分配的内存大小（字节形式）。

device_realloc_size 【optional】

接口定义

```
C_Status (*device_realloc_size)(const C_Device device, size_t* size)
```

接口说明

设备重分配的内存大小（字节形式），如果没有实现，则大小等于 device_max_alloc_size。

参数

device - 使用的设备。

size - 重分配的内存大小（字节形式）。

Stream 接口

create_stream 【required】

接口定义

```
C_Status (*create_stream)(const C_Device device, C_Stream* stream)
```

接口说明

创建一个 stream 对象，stream 是框架内部用于执行异步任务的任务队列，同一 stream 中的任务按顺序执行。硬件不支持异步执行时该接口需要空实现。

参数

device - 使用的设备。

stream - 存储创建的 stream 对象。

destroy_stream [required]

接口定义

```
C_Status (*destroy_stream) (const C_Device device, C_Stream stream)
```

接口说明

销毁一个 stream 对象。硬件不支持异步执行时该接口需要空实现。

参数

device - 使用的设备。

stream - 需要释放的 stream 对象。

query_stream [optional]

接口定义

```
C_Status (*query_stream) (const C_Device device, C_Stream stream)
```

接口说明

查询 stream 上的任务是否完成，如果没有实现，PaddlePaddle 会用 synchronize_stream 代替。

参数

device - 使用的设备。

stream - 需要查询的 stream。

synchronize_stream [required]

接口定义

```
C_Status (*synchronize_stream)(const C_Device device, C_Stream stream)
```

接口说明

同步 stream，等待 stream 上所有任务完成。硬件不支持异步执行时该接口需要空实现。

参数

device - 使用的设备。

stream - 需要同步的 stream。

stream_add_callback [optional]

接口定义

```
C_Status (*stream_add_callback)(const C_Device device, C_Stream stream, C_Callback_
↪callback, void* user_data)
```

接口说明

添加一个主机回调函数到 stream 上。

参数

device - 使用的设备。

stream - 添加回调到该 stream 中。

callback - 回调函数。

user_data - 回调函数的参数。

stream_wait_event 【required】

接口定义

```
C_Status (*stream_wait_event)(const C_Device device, C_Stream stream, C_Event event)
```

接口说明

等待 stream 上的一个 event 完成。硬件不支持异步执行时该接口需要空实现。

参数

device - 使用的设备。

stream - 等待的 stream。

event - 等待的 event。

Event 接口

create_event 【required】

接口定义

```
C_Status (*create_event)(const C_Device device, C_Event* event)
```

接口说明

创建一个 event 对象， event 被框架内部用于同步不同 stream 之间的任务。硬件不支持异步执行时该接口需要空实现。

参数

device - 使用的设备。

event - 存储创建的 event 对象。

destroy_event [required]

接口定义

```
C_Status (*destroy_event) (const C_Device device, C_Event event)
```

接口说明

销毁一个 event 对象。硬件不支持异步执行时该接口需要空实现。

参数

device - 使用的设备。

event - 需要释放的 event 对象。

record_event [required]

接口定义

```
C_Status (*record_event) (const C_Device device, C_Stream stream, C_Event event)
```

接口说明

在 stream 上记录 event。硬件不支持异步执行时该接口需要空实现。

参数

device - 使用的设备。

stream - 在该 stream 上记录 event。

event - 被记录的 event。

query_event [optional]

接口定义

```
C_Status (*query_event) (const C_Device device, C_Event event)
```

接口说明

查询 event 是否完成，如果没有实现，PaddlePaddle 会用 synchronize_event 代替。

参数

device - 使用的设备。

event - 需要查询的 event 对象。

synchronize_event [required]

接口定义

```
C_Status (*synchronize_event) (const C_Device device, C_Event event)
```

接口说明

同步 event，等待 event 完成。硬件不支持异步执行时该接口需要空实现。

参数

device - 使用的设备。

event - 需要同步的 event。

5.7.2 自定义 Kernel

内核函数（简称 Kernel）对应算子的具体实现，飞桨框架针对通过自定义 Runtime 机制注册的外部硬件，提供了配套的自定义 Kernel 机制，以实现独立于框架的 Kernel 编码、注册、编译和自动加载使用。自定义 Kernel 基于飞桨对外发布的函数式 Kernel 声明、对外开放的 C++ API 和注册宏实现。

- **Kernel 函数声明**：介绍飞桨发布的函数式 Kernel 声明。
- **Kernel 实现接口**：介绍自定义 Kernel 函数体实现所需的 C++ API。
- **Kernel 注册接口**：介绍自定义 Kernel 注册宏。

Kernel 函数声明

飞桨通过头文件发布函数式 Kernel 声明，框架内外一致。

编写自定义 Kernel 需基于具体的 Kernel 函数声明，头文件位于飞桨安装路径的 `include/paddle/phi/kernels/` 下。

Kernel 函数声明的格式如下：

```
template <typename T, typename Context>
void KernelNameKernel(const Context& dev_ctx,
                      InputTensor(s),
                      Attribute(s),
                      OutTensor(s));
```

约定：

1. 模板参数：固定写法，第一个模板参数为数据类型 `T`，第二个模板参数为设备上下文 `Context`。
2. 函数返回：固定为 `void`。
3. 函数命名：Kernel 名称 +Kernel 后缀，驼峰式命名，如 `SoftmaxKernel`。
4. 函数参数：依次为设备上下文参数，输入 Tensor 参数 (`InputTensor`)，属性参数 (`Attribute`) 和输出 Tensor 参数 (`OutTensor`)。其中：
 - 设备上下文参数：固定为 `const Context&` 类型；
 - 自定义 Kernel 对应 `CustomContext` 类型，请参照 `custom_context.h`
 - `InputTensor`：数量 ≥ 0 ，支持的类型包括：
 - `const DenseTensor&` 请参照 `dense_tensor.h`
 - `const SelectedRows&` 请参照 `selected_rows.h`
 - `const SparseCooTensor&` 请参照 `sparse_coo_tensor.h`
 - `const SparseCsrTensor&` 请参照 `sparse_csr_tensor.h`
 - `const std::vector<DenseTensor*>&`

- const std::vector<SparseCooTensor*>&
 - const std::vector<SparseCsrTensor*>&
- Attribute: 数量 ≥ 0 , 支持的类型包括:
 - bool
 - float
 - double
 - int
 - int64_t
 - phi::dtype::float16 请参照[float16.h](#)
 - const Scalar& 请参照[scalar.h](#)
 - DataType 请参照[data_type.h](#)
 - DataLayout 请参照[layout.h](#)
 - Place 请参照[place.h](#)
 - const std::vector<int64_t>&
 - const ScalarArray& 请参照[int_array.h](#)
 - const std::vector<int>&
 - const std::string&
 - const std::vector<bool>&
 - const std::vector<float>&
 - const std::vector<double>&
 - const std::vector<std::string>&

- OutTensor: 数量 > 0 , 支持的类型包括:

- DenseTensor*
- SelectedRows*
- SparseCooTensor*
- SparseCsrTensor*
- std::vector<DenseTensor*>
- std::vector<SparseCooTensor*>
- std::vector<SparseCsrTensor*>

示例, 如 softmax 的 Kernel 函数位于 softmax_kernel.h 中, 具体如下:

```
// Softmax 内核函数
// 模板参数: T - 数据类型
//           Context - 设备上下文
// 参数: dev_ctx - Context 对象
//       x - DenseTensor 对象
//       axis - int 类型
//       dtype - DataType 类型
//       out - DenseTensor 指针
// 返回: None
template <typename T, typename Context>
void SoftmaxKernel(const Context& dev_ctx,
                   const DenseTensor& x,
                   int axis,
                   DataType dtype,
                   DenseTensor* out);
```

注意:

1. Kernel 函数声明是自定义 Kernel 能够被注册和框架调用的基础，由框架发布，需要严格遵守
2. Kernel 函数声明与头文件可能不完全对应，可以按照函数命名约定等查找所需 Kernel 函数声明

Kernel 实现接口

Title overline too short.

```
#####
Kernel 实现 接口
#####
```

自定义 Kernel 函数体的实现主要依赖两部分：1. 飞桨发布的 API：如设备上下文 API、Tensor 相关 API 和异常处理 API 等；2. 硬件封装库的 API：根据具体硬件封装库使用。其中飞桨发布的 C++ API 已通过头文件方式发布。

- **Context API**：介绍设备上下文相关 C++ API。
- **Tensor API**：介绍 Tensor 相关 C++ API。
- **Exception API**：介绍异常处理相关 C++ API。

注：飞桨发布了丰富的 C++ API，此处重点介绍三类 API 并在相应页面列举相关联的类和文件供开发者参考查阅。

Context API

CustomContext

CustomContext 为自定义 Kernel 函数模板参数 Context 的实参，请参照custom_context.h

```
// 构造函数
// 参数: place - CustomPlace对象
// 返回: None
explicit CustomContext(const CustomPlace&);

// 析构函数
virtual ~CustomContext();

// 获取设备上下文Place信息
// 参数: None
// 返回: place - Place对象
const Place& GetPlace() const override;

// 获取设备上下文stream信息
// 参数: None
// 返回: stream - void*类型指针
void* stream() const;

// 等待stream上的操作完成
// 参数: None
// 返回: None
void Wait() const override;
```

DeviceContext

CustomContext 继承自 DeviceContext，请参照device_context.h

```
// 无参构造函数
DeviceContext();

// 拷贝构造函数
DeviceContext(const DeviceContext&);

// 移动构造函数
DeviceContext(DeviceContext&&);

// 移动赋值操作
```

(下页继续)

(续上页)

```
DeviceContext& operator=(DeviceContext&&);

// 析构函数
virtual ~DeviceContext();

// 设置Device Allocator
// 参数: Allocator指针
// 返回: None
void SetAllocator(const Allocator*);

// 设置Host Allocator
// 参数: Allocator指针
// 返回: None
void SetHostAllocator(const Allocator*);

// 设置zero-size Allocator
// 参数: Allocator指针
// 返回: None
void SetZeroAllocator(const Allocator*);

// 获取 Allocator
// 参数: None
// 返回: Allocator对象
const Allocator& GetAllocator() const;

// 获取 Host Allocator
// 参数: None
// 返回: Allocator对象
const Allocator& GetHostAllocator() const;

// 获取 zero-size Allocator
// 参数: None
// 返回: Allocator对象
const Allocator& GetZeroAllocator() const;

// 为Tensor分配Device内存
// 参数: TensorBase类型指针
//      dtype - DataType类型变量
//      requested_size - size_t类型变量, 默认值为0
// 返回: 数据指针 - void*类型指针
void* Alloc(TensorBase*, DataType dtype, size_t requested_size = 0) const;

// 为Tensor分配Device内存
```

(下页继续)

(续上页)

```

// 模板参数: T - 数据类型
// 参数: TensorBase类型指针
//      requested_size - size_t类型变量, 默认值为0
// 返回: 数据指针 - T*类型指针
template <typename T>
T* Alloc(TensorBase* tensor, size_t requested_size = 0) const;

// 为Tensor分配Host内存
// 参数: TensorBase指针
//      dtype - DataType类型变量
//      requested_size - size_t类型变量, 默认值为0
// 返回: 数据指针 - void*类型指针
void* HostAlloc(TensorBase* tensor,
                  DataType dtype,
                  size_t requested_size = 0) const;

// 为Tensor分配Host内存
// 模板参数: T - 数据类型
// 参数: TensorBase指针
//      requested_size - size_t类型变量, 默认值为0
// 返回: 数据指针 - T*类型数据指针
template <typename T>
T* HostAlloc(TensorBase* tensor, size_t requested_size = 0) const;

// 获取设备上下文Place信息, 子类实现
// 参数: None
// 返回: place - Place对象
virtual const Place& GetPlace() const = 0;

// 等待stream上的操作完成, 子类实现
// 参数: None
// 返回: None
virtual void Wait() const {}

// 设置随机数发生器
// 参数: Generator指针
// 返回: None
void SetGenerator(Generator*);

// 获取随机数发生器
// 参数: None
// 返回: Generator指针
Generator* GetGenerator() const;

```

(下页继续)

(续上页)

```
// 设置Host随机数发生器
// 参数: Generator指针
// 返回: None
void SetHostGenerator(Generator*);

// 获取Host随机数发生器
// 参数: None
// 返回: Generator指针
Generator* GetHostGenerator() const;
```

相关内容

- Place 与 CustomPlace: 请参照place.h
- Allocation 与 Allocator: 请参照allocator.h
- TensorBase: 请参照tensor_base.h
- DataType: 请参照data_type.h
- Generator: 请参照generator.h

Tensor API

飞桨发布多种 Tensor，基类均为 TensorBase，这里列举常用的 DenseTensor API，TensorBase 与其它 Tensor 类型请参照文后链接。

DenseTensor

DenseTensor 中的所有元素数据存储在连续内存中，请参照dense_tensor.h

```
// 构造DenseTensor并分配内存
// 参数: a - Allocator指针类型
//       meta - DenseTensorMeta对象
// 返回: None
DenseTensor(Allocator* a, const DenseTensorMeta& meta);

// 构造DenseTensor并分配内存
// 参数: a - Allocator指针类型
//       meta - DenseTensorMeta移动对象
// 返回: None
DenseTensor(Allocator* a, DenseTensorMeta&& meta);
```

(下页继续)

(续上页)

```

// 构造DenseTensor并分配内存
// 参数: holder - Allocation共享指针类型
//       meta - DenseTensorMeta移动对象
// 返回: None
DenseTensor(const std::shared_ptr<phi::Allocation>& holder,
            const DenseTensorMeta& meta);

// 移动构造函数
// 参数: other - DenseTensor移动对象
// 返回: None
DenseTensor(DenseTensor&& other) = default;

// 拷贝构造函数
// 参数: other - DenseTensor对象
// 返回: None
DenseTensor(const DenseTensor& other);

// 赋值操作
// 参数: other - DenseTensor对象
// 返回: DenseTensor对象
DenseTensor& operator=(const DenseTensor& other);

// 移动赋值操作
// 参数: other - DenseTensor对象
// 返回: DenseTensor对象
DenseTensor& operator=(DenseTensor&& other);

// 无参构造函数
DenseTensor();

// 析构函数
virtual ~DenseTensor() = default;

// 获取类名，静态函数
// 参数: None
// 返回: 字符串指针
static const char* name();

// 获取Tensor中元素数量
// 参数: None
// 返回: int64_t类型变量
int64_t numel() const override;

```

(下页继续)

(续上页)

```
// 获取Tensor的dims信息
// 参数: None
// 返回: DDim对象
const DDim& dims() const noexcept override;
```



```
// 获取Tensor的lod信息
// 参数: None
// 返回: LoD对象
const LoD& lod() const noexcept override;
```



```
// 获取Tensor的数据类型信息
// 参数: None
// 返回: DataType类型变量
DataType dtype() const noexcept override;
```



```
// 获取Tensor的内存布局信息
// 参数: None
// 返回: DataLayout类型变量
DataLayout layout() const noexcept override;
```



```
// 获取Tensor的Place信息
// 参数: None
// 返回: Place类型变量
const Place& place() const override;
```



```
// 获取Tensor的meta信息
// 参数: None
// 返回: DenseTensorMeta对象
const DenseTensorMeta& meta() const noexcept;
```



```
// 设置Tensor的meta信息
// 参数: meta - DenseTensorMeta移动对象
// 返回: None
void set_meta(DenseTensorMeta&& meta);
```



```
// 设置Tensor的meta信息
// 参数: meta - DenseTensorMeta对象
// 返回: None
void set_meta(const DenseTensorMeta& meta);
```



```
// 检查Tensor的meta信息是否有效
// 参数: None
```

(下页继续)

(续上页)

```
// 返回: bool类型变量
bool valid() const noexcept override;
```

```
// 检查Tensor是否被初始化
// 参数: None
// 返回: bool类型变量
bool initialized() const override;
```

```
// 为Tensor分配内存
// 参数: allocator - Allocator类型指针
//       dtype - DataType变量
//       requested_size - size_t类型变量
// 返回: void*类型指针
void* AllocateFrom(Allocator* allocator,
                     DataType dtype,
                     size_t requested_size = 0) override;
```

```
// 检查是否与其它Tensor共享内存
// 参数: b - DenseTensor对象
// 返回: bool类型变量
bool IsSharedWith(const DenseTensor& b) const;
```

```
// 修改Tensor的Dims信息并分配内存
// 参数: dims - DDim对象
// 返回: None
void ResizeAndAllocate(const DDim& dims);
```

```
// 修改Tensor的Dims信息
// 参数: dims - DDim对象
// 返回: DenseTensor对象
DenseTensor& Resize(const DDim& dims);
```

```
// 重置Tensor的LoD信息
// 参数: lod - LoD对象
// 返回: None
void ResetLoD(const LoD& lod);
```

```
// 获取Tensor的内存大小
// 参数: None
// 返回: size_t类型变量
size_t capacity() const;
```

```
// 获取Tensor的不可修改数据指针
```

(下页继续)

(续上页)

```
// 模板参数: T - 数据类型
// 参数: None
// 返回: 不可修改的T类型数据指针
template <typename T>
const T* data() const;

// 获取Tensor的不可修改数据指针
// 参数: None
// 返回: 不可修改的void类型数据指针
const void* data() const;

// 获取Tensor的可修改内存数据指针
// 模板参数: T - 数据类型
// 参数: None
// 返回: 可修改的T类型数据指针
template <typename T>
T* data();

// 获取Tensor的可修改内存数据指针
// 参数: None
// 返回: 可修改的void类型数据指针
void* data();
```

其它 Tensor 类型

- TensorBase: 请参照tensor_base.h
- SelectedRows: 请参照selected_rows.h
- SparseCooTensor: 请参照sparse_coo_tensor.h
- SparseCsrTensor: 请参照sparse_csr_tensor.h

相关内容

- Allocation 与 Allocator: 请参照allocator.h
- DenseTensorMeta: 请参照tensor_meta.h
- DDim: 请参照ddim.h
- LoD: 请参照lod_utils.h
- DataType: 请参照data_type.h
- DataLayout: 请参照layout.h

- Place: 请参照place.h

Exception API

PADDLE_ENFORCE

使用方式:

```
PADDLE_ENFORCE_{TYPE}(cond_a, // 条件A
                      cond_b, // 条件B, 根据TYPE可选
                      phi::errors::{ERR_TYPE} ("{ERR_MSG}"));
```

根据 TYPE 的不同, 分为:

ERR_TYPE 支持:

ERR_MSG 为 C 语言风格字符串, 支持变长参数。

示例:

```
// 如果num_col_dims >= 2 && num_col_dims <= src.size() 不为true则报InvalidArgumentException异常
// 和打印相关提示信息
PADDLE_ENFORCE_EQ(
    (num_col_dims >= 2 && num_col_dims <= src.size()),
    true,
    phi::errors::InvalidArgumentException("The num_col_dims should be inside [2, %d] "
                                         "in flatten_to_3d, but received %d.",
                                         src.size(),
                                         num_col_dims));
```

相关内容

- PADDLE_ENFORCE: 请参照enforce.h
- errors: 请参照errors.h

Kernel 注册接口

自定义 Kernel 通过飞桨框架提供的注册宏进行注册, 以便飞桨框架调用。

注册宏的位置需要放置在全局空间下。

注册宏的基本形式如下: 具体实现请参照kernel_registry.h

```
/** PD_REGISTER_PLUGIN_KERNEL
*
* Used to register kernels for plug-in backends.
* Support user-defined backend such as 'Ascend910'.
*/
PD_REGISTER_PLUGIN_KERNEL(kernel_name, backend, layout, meta_kernel_fn, ...)) {}
```

说明：

- 注册宏名称：固定为 `PD_REGISTER_PLUGIN_KERNEL`
- 第一个参数：`kernel_name`，即 `Kernel` 名称，飞桨内外一致，请参照 CPU 相同 `Kernel` 函数注册名称，如 `softmax`
- 第二个参数：`backend`，即后端名称，可自定义，但须与自定义 `Runtime` 设定的名称一致，如 `Ascend910`
- 第三个参数：`layout`，即内存布局，为 `DataLayout` 类型的枚举，按需设定，请参照 `layout.h`
- 第四个参数：`meta_kernel_fn`，即 `Kernel` 函数名，注意此处不加模板参数，如 `my_namespace::SoftmaxKernel`
- 不定长数据类型参数：C++ 的基础数据类型或飞桨定义的 `phi::dtype::float16`、`phi::dtype::bfloat16`、`phi::dtype::complex` 等类型，请参照 `data_type.h`
- 末尾：固定为函数体，其中可按需对 `Kernel` 进行必要设置，如果没有，保留 `{}`。

说明：末尾函数体对应的函数声明如下：

```
// Kernel 参数定义
// 参数： kernel_key - KernelKey 对象
//         kernel - Kernel 指针
// 返回： None
void __PD_KERNEL_args_def_FN_##kernel_name##_##backend##_##layout (
    const ::phi::KernelKey& kernel_key, ::phi::Kernel* kernel);
```

即函数体中可使用参数 `kernel_key` 与 `kernel`，在 `Kernel` 注册时对 `Kernel` 进行个性化调整。

示例，如 `softmax` 的 `CustomCPU` 后端 `Kernel` 注册如下：

```
// Softmax的CustomCPU后端Kernel注册
// 全局命名空间
// 参数： softmax - Kernel名称
//        CustomCPU - 后端名称
//        ALL_LAYOUT - 内存布局
//        custom_cpu::SoftmaxKernel - Kernel函数名
//        float - 数据类型名
//        double - 数据类型名
//        phi::dtype::float16 - 数据类型名
```

(下页继续)

(续上页)

```
PD_REGISTER_PLUGIN_KERNEL(softmax,
                          CustomCPU,
                          ALL_LAYOUT,
                          custom_cpu::SoftmaxKernel,
                          float,
                          double,
                          phi::dtype::float16) {}
```

注意：

1. 对于通过自定义 Runtime 接入的后端，backend 参数须与之名称保持一致
2. 注册宏末尾函数体中除非有明确需要，否则保留空函数体即可，请参照飞桨框架内其它后端的使用

5.7.3 新硬件接入示例

本教程介绍如何为 PaddlePaddle 实现一个 CustomDevice 插件，添加一个名为 CustomCPU 的新硬件后端，并进行编译，打包，安装和使用。

注意：

- 请确保已经正确安装了飞桨 develop 最新版本
- 当前仅支持 Linux 平台，示例中使用 X86_64 平台

第一步：实现自定义 Runtime

InitPlugin

InitPlugin 作为自定义 Runtime 的入口函数，插件需要实现该函数，并在该函数中检查参数，填充硬件信息，注册 Runtime API。PaddlePaddle 初始化时加载插件并调用 InitPlugin 完成插件初始化，注册 Runtime（整个过程由框架自动完成，只要动态链接库位于 site-packages/paddle-plugins/ 或 CUSTOM_DEVICE_ROOT 环境变量指定目录即可）。

例子：

```
#include "paddle/phi/backends/device_ext.h"

void InitPlugin(CustomRuntimeParams *params) {
    // 将检查版本兼容性并填充插件使用的自定义 Runtime 版本信息
    PADDLE_CUSTOM_RUNTIME_CHECK_VERSION(params);

    // 填充 Runtime 基本信息
    params->device_type = "CustomCPU";
```

(下页继续)

(续上页)

```

params->sub_device_type = "V1";

// 注册 Runtime API
params->interface->set_device = set_device;
params->interface->get_device = get_device;
params->interface->create_stream = create_stream;
params->interface->destroy_stream = destroy_stream;
params->interface->create_event = create_event;
params->interface->destroy_event = destroy_event;
params->interface->record_event = record_event;
params->interface->synchronize_device = sync_device;
params->interface->synchronize_stream = sync_stream;
params->interface->synchronize_event = sync_event;
params->interface->stream_wait_event = stream_wait_event;
params->interface->memory_copy_h2d = memory_copy;
params->interface->memory_copy_d2d = memory_copy;
params->interface->memory_copy_d2h = memory_copy;
params->interface->device_memory_allocate = allocate;
params->interface->device_memory_deallocate = deallocate;
params->interface->get_device_count = get_device_count;
params->interface->get_device_list = get_device_list;
params->interface->device_memory_stats = memstats;
params->interface->device_min_chunk_size = get_min_chunk_size;
}

```

插件首先需要检查 InitPlugin 的参数，框架设置该参数成员 size 为其类型的大小并传入 InitPlugin，CustomRuntimeParams 与 C_DeviceInterface 的类型定义详见 device_ext.h。

然后，插件需要填充插件的基本信息以及版本号，以供 PaddlePaddle 管理插件以及检查版本兼容性。

- params->size 和 params->interface.size：自定义 Runtime 的后续版本中，会保证 size 和 interface 为 CustomRuntimeParams 类型的前两个成员。
- params->version：插件填充版本信息，其版本号在 device_ext.h 中定义，PaddlePaddle 在注册自定义 Runtime 时检查版本兼容性。
- params->device_type：硬件后端名，具有同名的插件已经注册时，则不会注册 Runtime。
- params->sub_device_type：硬件后端子类型名。

最后，插件需要填充 params->interface 中的回调接口（至少实现 Required 接口，否则 Runtime 不会被注册），完成自定义 Runtime 的初始化。具体 API 的说明详见[自定义 Runtime 文档](#)。

```

#include <malloc.h>

static size_t global_total_mem_size = 1 * 1024 * 1024 * 1024UL;

```

(下页继续)

(续上页)

```

static size_t global_free_mem_size = global_total_mem_size;

C_Status set_device(const C_Device device) {
    return C_SUCCESS;
}

C_Status get_device(const C_Device device) {
    device->id = 0;
    return C_SUCCESS;
}

C_Status get_device_count(size_t *count) {
    *count = 1;
    return C_SUCCESS;
}

C_Status get_device_list(size_t *device) {
    *device = 0;
    return C_SUCCESS;
}

C_Status memory_copy(const C_Device device, void *dst, const void *src, size_t size) {
    memcpy(dst, src, size);
    return C_SUCCESS;
}

C_Status allocate(const C_Device device, void **ptr, size_t size) {
    if (size > global_free_mem_size) {
        return C_FAILED;
    }
    global_free_mem_size -= size;
    *ptr = malloc(size);
    return C_SUCCESS;
}

C_Status deallocate(const C_Device device, void *ptr, size_t size) {
    if (!ptr) {
        return C_FAILED;
    }
    global_free_mem_size += size;
    free(ptr);
    return C_SUCCESS;
}

```

(下页继续)

(续上页)

```
C_Status create_stream(const C_Device device, C_Stream *stream) {
    stream = nullptr;
    return C_SUCCESS;
}

C_Status destroy_stream(const C_Device device, C_Stream stream) {
    return C_SUCCESS;
}

C_Status create_event(const C_Device device, C_Event *event) {
    return C_SUCCESS;
}

C_Status record_event(const C_Device device, C_Stream stream, C_Event event) {
    return C_SUCCESS;
}

C_Status destroy_event(const C_Device device, C_Event event) {
    return C_SUCCESS;
}

C_Status sync_device(const C_Device device) {
    return C_SUCCESS;
}

C_Status sync_stream(const C_Device device, C_Stream stream) {
    return C_SUCCESS;
}

C_Status sync_event(const C_Device device, C_Event event) {
    return C_SUCCESS;
}

C_Status stream_wait_event(const C_Device device, C_Stream stream, C_Event event) {
    return C_SUCCESS;
}

C_Status memstats(const C_Device device, size_t *total_memory, size_t *free_memory) {
    *total_memory = global_total_mem_size;
    *free_memory = global_free_mem_size;
    return C_SUCCESS;
}
```

(下页继续)

(续上页)

```
C_Status get_min_chunk_size(const C_Device device, size_t *size) {
    *size = 1;
    return C_SUCCESS;
}
```

第二步：添加自定义 Kernel

以 add 为例，介绍如何实现一个 kernel 并完成注册。

例子：

1. 确定 Kernel 声明

查找飞桨发布的头文件 `math_kernel.h` 中，其 Kernel 函数声明如下：

```
// Add 内核函数
// 模板参数： T - 数据类型
//             Context - 设备上下文
// 参数： dev_ctx - Context 对象
//         x - DenseTensor 对象
//         y - DenseTensor 对象
//         out - DenseTensor 指针
// 返回： None
template <typename T, typename Context>
void AddKernel(const Context& dev_ctx,
               const DenseTensor& x,
               const DenseTensor& y,
               DenseTensor* out);
```

2. Kernel 实现与注册

```
// add_kernel.cc

#include "paddle/phi/extension.h" // 自定义Kernel依赖头文件

namespace custom_cpu {

// Kernel 函数体实现
template <typename T, typename Context>
void AddKernel(const Context& dev_ctx,
```

(下页继续)

(续上页)

```
    const phi::DenseTensor& x,
    const phi::DenseTensor& y,
    phi::DenseTensor* out) {
// 使用dev_ctx的Alloc API为输出参数out分配模板参数T数据类型的内存空间
dev_ctx.template Alloc<T>(out);
// 使用DenseTensor的numel API获取Tensor元素数量
auto numel = x.numel();
// 使用DenseTensor的数据API获取输入参数x的模板参数T类型的数据指针
auto x_data = x.data<T>();
// 使用DenseTensor的数据API获取输入参数y的模板参数T类型的数据指针
auto y_data = y.data<T>();
// 使用DenseTensor的数据API获取输出参数out的模板参数T类型的数据指针
auto out_data = out->data<T>();
// 完成计算逻辑
for (auto i = 0; i < numel; ++i) {
    out_data[i] = x_data[i] + y_data[i];
}
}

} // namespace custom_cpu

// 全局命名空间内使用注册宏完成Kernel注册
// CustomCPU的AddKernel注册
// 参数：add - Kernel名称
//       CustomCPU - 后端名称
//       ALL_LAYOUT - 内存布局
//       custom_cpu::AddKernel - Kernel函数名
//       int - 数据类型名
//       int64_t - 数据类型名
//       float - 数据类型名
//       double - 数据类型名
//       phi::dtype::float16 - 数据类型名
PD_REGISTER_PLUGIN_KERNEL(add,
                         CustomCPU,
                         ALL_LAYOUT,
                         custom_cpu::AddKernel,
                         int,
                         int64_t,
                         float,
                         double,
                         phi::dtype::float16){}
```

第三步：编译与安装

CMake 编译

编写 CMakeLists.txt

```

cmake_minimum_required(VERSION 3.10)

project(paddle-custom_cpu CXX C)

set(PLUGIN_NAME      "paddle_custom_cpu")
set(PLUGIN_VERSION   "0.0.1")

set(PADDLE_PLUGIN_DIR "/path/to/site-packages/paddle-plugins/")
set(PADDLE_INC_DIR    "/path/to/site-packages/paddle/include/")
set(PADDLE_LIB_DIR    "/path/to/site-packages/paddle/fluid/")

##### 三方依赖，本示例中使用Paddle相同依赖
set(BOOST_INC_DIR     "/path/to/Paddle/build/third_party/boost/src/extern_boost")
set(GFLAGS_INC_DIR    "/path/to/Paddle/build/third_party/install/gflags/include")
set(GLOG_INC_DIR      "/path/to/Paddle/build/third_party/install/glog/include")
set(MKLDNN_INC_DIR    "/path/to/Paddle/build/third_party/install/mkldnn/include")
set(THIRD_PARTY_INC_DIR ${BOOST_INC_DIR} ${GFLAGS_INC_DIR} ${GLOG_INC_DIR} ${MKLDNN_
INC_DIR})

include_directories(${PADDLE_INC_DIR} ${THIRD_PARTY_INC_DIR})
link_directories(${PADDLE_LIB_DIR})

add_definitions(-DPADDLE_WITH_CUSTOM_DEVICE) # for out CustomContext
add_definitions(-DPADDLE_WITH_CUSTOM_KERNEL) # for out fluid seperate
add_definitions(-DPADDLE_WITH_MKLDNN) # for out MKLDNN compiling

##### 编译插件
add_library(${PLUGIN_NAME} SHARED runtime.cc add_kernel.cc)
target_link_libraries(${PLUGIN_NAME} PRIVATE :core_avx.so) # special name

##### 打包插件
configure_file(${CMAKE_CURRENT_SOURCE_DIR}/setup.py.in
${CMAKE_CURRENT_BINARY_DIR}/setup.py)

add_custom_command(TARGET ${PLUGIN_NAME} POST_BUILD
COMMAND ${CMAKE_COMMAND} -E remove -f ${CMAKE_CURRENT_BINARY_DIR}/python/
COMMAND ${CMAKE_COMMAND} -E make_directory ${CMAKE_CURRENT_BINARY_DIR}/python/
COMMAND ${CMAKE_COMMAND} -E make_directory ${CMAKE_CURRENT_BINARY_DIR}/python/
→paddle-plugins/                                         (下页继续)

```

(续上页)

```

COMMAND ${CMAKE_COMMAND} -E copy_if_different ${CMAKE_CURRENT_BINARY_DIR}/lib$  

→{PLUGIN_NAME}.so ${CMAKE_CURRENT_BINARY_DIR}/python/paddle-plugins/  

COMMENT "Creating plugin directories----->>>"  

)  
  

add_custom_command(OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/python/.timestamp  

COMMAND python3 ${CMAKE_CURRENT_BINARY_DIR}/setup.py bdist_wheel  

DEPENDS ${PLUGIN_NAME}  

COMMENT "Packing whl packages----->>>"  

)  
  

add_custom_target(python_package ALL DEPENDS ${CMAKE_CURRENT_BINARY_DIR}/python/.  

→timestamp)

```

编写 setup.py.in

CMake 根据 setup.py.in 生成 setup.py，再使用 setuptools 将插件封装成 wheel 包。

```

from setuptools import setup, Distribution  
  

packages = []
package_data = {}  
  

class BinaryDistribution(Distribution):
    def has_ext_modules(self):
        return True  
  

setup(
    name = '@CMAKE_PROJECT_NAME@',
    version='@PLUGIN_VERSION@',
    description='Paddle CustomCPU plugin',
    long_description='',
    long_description_content_type="text/markdown",
    author_email="Paddle-better@baidu.com",
    maintainer="PaddlePaddle",
    maintainer_email="Paddle-better@baidu.com",
    project_urls={},
    license='Apache Software License',
    packages= [
        'paddle-plugins',
    ],
    include_package_data=True,
    package_data = {
        '*': ['*.so', '*.h', '*.py', '*.hpp'],
    }
)

```

(下页继续)

(续上页)

```

},
package_dir = {
    '' : 'python',
},
zip_safe=False,
distclass=BinaryDistribution,
entry_points={
    'console_scripts': [
        ]
},
classifiers=[
],
keywords='Paddle CustomCPU plugin',
)

```

通过如下命令完成插件编译。

```

$ mkdir build
$ cd build
$ cmake ..
$ make

```

编译完成后在 build/dist 目录下生成 wheel 包。

setuptools 编译

编写 setup.py

setuptools 也可以用于编译插件，并直接打包

```

from setuptools import setup, Distribution, Extension
from setuptools.command.build_ext import build_ext
import os
import shutil

packages = []
package_data = {}

class BinaryDistribution(Distribution):
    def has_ext_modules(self):
        return True

    for pkg_dir in ['build/python/paddle-plugins/']:
        if os.path.exists(pkg_dir):

```

(下页继续)

(续上页)

```
shutil.rmtree(pkg_dir)
os.makedirs(pkg_dir)

include_dirs = [
    '/path/to/site-packages/paddle/include',
    "/path/to/Paddle/build/third_party/boost/src/extern_boost",
    "/path/to/Paddle/build/third_party/install/gflags/include",
    "/path/to/Paddle/build/third_party/install/glog/include",
    "/path/to/Paddle/build/third_party/install/mkldnn/include",
]

extra_compile_args = [
    '-DPADDLE_WITH_CUSTOM_KERNEL',
    '-DPADDLE_WITH_CUSTOM_DEVICE',
    '-DPADDLE_WITH_MKLDNN',
]

ext_modules = [Extension(name='paddle-plugins.libpaddle_custom_cpu',
                         sources=['runtime.cc', 'add_kernel.cc'],
                         include_dirs=include_dirs,
                         library_dirs=['/path/to/site-packages/paddle/fluid/'],
                         libraries=[':core_avx.so'],
                         extra_compile_args=extra_compile_args)]

setup(
    name='paddle-custom_cpu',
    version='0.0.1',
    description='Paddle CustomCPU plugin',
    long_description='',
    long_description_content_type="text/markdown",
    author_email="Paddle-better@baidu.com",
    maintainer="PaddlePaddle",
    maintainer_email="Paddle-better@baidu.com",
    project_urls={},
    license='Apache Software License',
    ext_modules=ext_modules,
    packages=[
        'paddle-plugins',
    ],
    include_package_data=True,
    package_data={
        '*': ['*.so', '*.h', '*.py', '*.hpp'],
    },
)
```

(下页继续)

(续上页)

```
package_dir={  
    '' : 'build/python',  
},  
zip_safe=False,  
distclass=BinaryDistribution,  
entry_points={  
    'console_scripts': [  
        ]  
},  
classifiers=[  
],  
keywords='Paddle CustomCPU plugin',  
)
```

通过如下命令完成插件编译。

```
$ python setup.py bdist_wheel
```

编译完成后在以 dist 目录下生成 wheel 包。

pip 安装

通过 pip 安装 wheel 包。

```
$ pip install build/dist/paddle_custom_cpu*.whl
```

第四步：加载与使用

安装插件到指定路径后 (site-packages/paddle-plugins)，我们就可以使用 PaddlePaddle 的 CustomCPU 硬件后端用于执行计算任务。

首先，需要查看 PaddlePaddle 目前已注册的自定义硬件。

```
>>> paddle.device.get_all_custom_device_type()  
['CustomCPU']
```

接下来设置要使用的硬件后端。

```
>>> paddle.set_device('CustomCPU')
```

最后，使用新硬件后端用于执行计算任务。

```
>>> x = paddle.to_tensor([1])
>>> x
Tensor(shape=[1], dtype=int64, place=Place(CustomCPU:0), stop_gradient=True,
       [1])
>>> x + x
Tensor(shape=[1], dtype=int64, place=Place(CustomCPU:0), stop_gradient=True,
       [2])
```

5.8 文档贡献指南

PaddlePaddle 的文档存储于 [PaddlePaddle/docs](#) 中，之后通过技术手段转为 HTML 文件后呈现至[官网文档](#)。官网文档和 docs 的对应关系如下：

5.8.1 一、修改前的准备工作

1.1 Fork

先跳转到 [PaddlePaddle/docs GitHub 首页](#)，然后单击 Fork 按钮，生成自己仓库下的目录，比如你的 GitHub 用户名为 USERNAME，则生成：<https://github.com/USERNAME/docs>。

1.2 Clone

将你目录下的远程仓库 clone 到本地。

```
git clone https://github.com/USERNAME/docs
cd docs
```

1.3 创建本地分支

docs 目前使用 [Git 流分支模型](#)进行开发，测试，发行和维护。

所有的 feature 和 bug fix 的开发工作都应该在一个新的分支上完成，一般从 develop 分支上创建新分支。

使用 git checkout -b 创建并切换到新分支。

```
git checkout -b my-cool-stuff
```

值得注意的是，在 checkout 之前，需要保持当前分支目录 clean，否则会把 untracked 的文件也带到新分支上，这可以通过 git status 查看。

1.4 下载 pre-commit 钩子工具（若有的话，可以跳过此步骤）

Paddle 开发人员使用 `pre-commit` 工具来管理 Git 预提交钩子。它可以帮助你格式化源代码 (C++, Python)，在提交 (`commit`) 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)。

`pre-commit` 测试是 `Travis-CI` 中单元测试的一部分，不满足钩子的 PR 不能被提交到 Paddle，首先安装并在当前目录运行它：

```
② pip install pre-commit  
② pre-commit install
```

Paddle 使用 `clang-format` 来调整 C/C++ 源代码格式，请确保 `clang-format` 版本在 3.8 以上。

注：通过 `pip install pre-commit` 和 `conda install -c conda-forge pre-commit` 安装的 `yapf` 稍有不同，Paddle 开发人员使用的是 `pip install pre-commit`。

5.8.2 二、正式修改文档

根据官网文档和 `docs` 的对应关系，确定要修改/新增的文档路径，然后修改或者新增。

2.1 新增文档

当你要新增文档时，需要参考上述的对应关系，找到合适的目录，新建 `Markdown` 或 `reStructuredText` 文件。中英文文档存储在同一路径下，其中，中文文档的后缀为 `_cn.md/rst`，英文文档的后缀为 `_en.md/rst`。

在新增文件后，还需要在目录文件中添加该文件的索引。目录文件一般是 `index_cn.rst/index_en.rst`，需要在文件的 `.. toctree::` 部分添加该文件的索引。

如在文档 `-> 使用教程 -> 动态图转静态图` 中新增《调试方法》，首先需要在 `docs/guides/04_dygraph_to_static` 中新建 `debugging_cn.md`, `debugging_en.md` 文件。之后，在 `docs/guides/04_dygraph_to_static/index_cn.rst` 的 `toctree` 部分，新增 `debugging_cn.md` 的索引，合入后即可展示到官网。

```
.. toctree::  
    :hidden:  
  
    basic_usage_cn.rst  
    program_translator_cn.rst  
    grammar_list_cn.rst  
    input_spec_cn.rst  
    error_handling_cn.md  
    debugging_cn.md    # 新增索引
```

2.2 修改文档

修改文档，可以通过文档的 URL，确定文档的源文件。如文档->使用教程->动态图转静态图中《调试方法》的文档 URL 为：https://www.paddlepaddle.org.cn/documentation/docs/zh/guides/04_dygraph_to_static/debugging_cn.html，URL 路径中，guides/04_dygraph_to_static/debugging_cn.html 即对应 (docs/docs/guides/04_dygraph_to_static/debugging_cn.md)，因此，可以很快的确定文档的源文件，然后直接修改即可。

5.8.3 三、提交 &push

3.1 提交 & 触发 CI 单测

- 修改 guides/04_dygraph_to_static/debugging_cn.md 这个文件，并提交这个文件

```
git status
On branch my-cool-stuff
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git restore <file>..." to discard changes in working directory)
modified:   paddle/tensor/math/all_cn.rst

no changes added to commit (use "git add" and/or "git commit -a")

git add  guides/04_dygraph_to_static/debugging_cn.md
```

如果你不想提交本次修改，使用 git checkout -- <file> 取消上面对 guides/04_dygraph_to_static/debugging_cn.md 文件的提交，可以将它恢复至上一次提交的状态：

```
git checkout -- guides/04_dygraph_to_static/debugging_cn.md
```

恢复后重新进行修改并提交文件即可。

- pre-commit：提交修改说明前，需要对本次修改做一些格式化检查：

```
pre-commit
CRLF end-lines remover.....(no files to check) Skipped
yapf.....Passed
Check for added large files.....Passed
Check for merge conflicts.....Passed
Check for broken symlinks.....Passed
Detect Private Key.....(no files to check) Skipped
Fix End of Files.....Passed
clang-format.....(no files to check) Skipped
cpplint.....(no files to check) Skipped
```

(下页继续)

(续上页)

pylint.....	Passed
copyright_checker.....	Passed

全部 Passed 或 Skipped 后，即可进入下一步。如果有 Failed 文件，则需要按照规范，修改出现 Failed 的文件后，重新 git add -> pre-commit，直至没有 Failed 文件。

```
☒ pre-commit
CRLF end-lines remover.....(no files to check)Skipped
yapf.....Failed
- hook id: yapf
- files were modified by this hook
Check for added large files.....Passed
Check for merge conflicts.....Passed
Check for broken symlinks.....Passed
Detect Private Key.....(no files to check)Skipped
Fix End of Files.....Passed
clang-format.....(no files to check)Skipped
cpplint.....(no files to check)Skipped
pylint.....Failed
- hook id: pylint-doc-string
- exit code: 127

./tools/codestyle/pylint_pre_commit.hook: line 11: pylint: command not found

copyright_checker.....Passed
```

- 填写提交说明：Git 每次提交代码，都需要写提交说明，让其他人知道这次提交做了哪些改变，可以通过 git commit 完成：

```
☒ git commit -m "fix docs bugs"
```

3.2 确保本地仓库是最新的

在准备发起 Pull Request 之前，需要同步原仓库（<https://github.com/PaddlePaddle/docs>）最新的代码。

首先通过 git remote 查看当前远程仓库的名字。

```
☒ git remote
origin
☒ git remote -v
origin      https://github.com/USERNAME/docs (fetch)
origin      https://github.com/USERNAME/docs (push)
```

这里 origin 是你 clone 的远程仓库的名字，也就是自己用户名下的 Paddle，接下来创建一个原始 Paddle 仓库的远程主机，命名为 upstream。

```
git remote add upstream https://github.com/PaddlePaddle/docs  
git remote  
origin  
upstream
```

获取 upstream 的最新代码并更新当前分支。

```
git fetch upstream  
git pull upstream develop
```

3.3 Push 到远程仓库

将本地的修改推送到 GitHub 上，也就是 <https://github.com/USERNAME/docs>。

```
# 推送到远程仓库 origin 的 my-cool-stuff 分支上  
git push origin my-cool-stuff
```

5.8.4 四、提交 PR

在你 push 后在对应仓库会提醒你进行 PR 操作，点击后，按格式填写 PR 内容，即可。

5.8.5 五、review&merge

提交 PR 后，可以指定 Paddle 的同学进行 Review。目前 Paddle 负责文档的同学是 @TCChenLong、@jzhang533、@saxon-zh、@Heeenrrry、@dingjiaweiww 等。

5.8.6 CI

Paddle 中与文档相关的 CI 流水线是 FluidDoc1 等，主要对以下几个方面进行检查：

- 检查 PR CLA
- 检查增量修改的 API 是否需要相关人员审核
- 若需要执行示例代码则执行看能否正常运行

如果无法通过该 CI，请点击对应 CI 的 details，查看 CI 运行的 log，并根据 log 修改你的 PR，直至通过 CI。未选择任何文件

Chapter 6

常见问题与解答

本栏目以问答对的形式收录了用户开发过程中遇到的高频咨询类问题，包含了 2.0 版本的变化、安装、数据与数据处理、模型组网、训练、预测、模型保存与加载、参数调整、分布式等几类常见问题。

除此之外，你也可以查看 [官网 API 文档](#)、[历史 Issues](#)、[飞桨论坛](#) 来寻求解答。

同时，你也可以在 [Github Issues](#) 中进行提问，飞桨会有专门的技术人员解答。

本栏目收录的问题集：

6.1 2.0 升级常见问题

6.1.1 环境变化

问题：paddle 2.0 是否支持 python2, python3.5 ?

- 答复：paddle 2.0 依然提供了 python2, python3.5 的官方安装包，但未来的某个版本将不再支持 python2, python3.5。（python 官方已停止对 python2, python3.5 的更新和维护）
-

问题：paddle 2.0 是否支持 CUDA9 ?

- 答复：本版本依然提供了 CUDA9 的官方安装包，但从未来的某个版本起，将不再支持 CUDA9。
-

问题：paddle 2.0 是否支持 CentOS6.0 ?

- 答复：对于 CentOS6.0，本版本仅提供了有限度的支持（仅发布了 CentOS6 下，Python3.7 CUDA10.2/CUDNN7 和 CPU 的安装包）。（CentOS6 官方已宣布了停止更新和维护）
-

问题：paddle 2.0 是否支持 AVX 指令的 x86 机器？

- 答复：2.0 版本依然提供了对不支持 AVX 指令的 x86 机器上运行飞桨的支持，但未来的某个版本将会废弃对不支持 AVX 指令的 x86 机器的支持。
-

问题：2.0 版本移除了哪些第三方库？

- 答复：2.0 版本移除的第三方依赖库为：nltk、opencv、scipy、rarfile、prettytable、pathlib、matplotlib、graphviz、objgraph。由于某些功能依然会有对 opencv 的依赖，在使用到时，会提示用户进行安装。

具体表现为：

- 删除依赖这些库的 API，原来 Paddle 基于这些第三方库提供了一些 API，在删除这些依赖库的同时也删除了这些 API，如：移除 matplotlib 的同时，paddle.utils.plot 也被删除了。
 - 删除依赖第三方库的数据集，如：移除 nltk 的同时，依赖 nltk 的数据集 sentiment(nltk.movie_review) 也被删除了。
 - 删除依赖的第三方库后，如果需要使用这些库，需要重新安装 pip install objgraph，直接 import objgraph 可能会报错。
-

问题：从 1.x 版本升级为 2.0 版本，哪些 API 有变动？

- 答复：飞桨框架 2.0.0 版本推荐用户使用位于 paddle 根目录下的 API，同时在 paddle.fluid 目录下保留了所有的 1.x 版本的 API，保留对之前版本 API 体系的支持。

查看 API 变动的两种方法：

- 依据 1.8 版本 API 到 2.0 版本 API 的对应关系表对 API 进行升级，请参考文档 [飞桨框架 API 映射表](#)
 - 飞桨提供了迁移工具，来方便用户将旧版本的代码迁移为 2.0.1 版本的代码，详情请见：[版本迁移工具](#)
-

问题：2.0 版本中是否有 LoDTensor？

- 答复：2.0 版本没有 LoDTensor 的概念，统一使用 Tensor 来表示数据。
- 解决方案：
 1. 使用 padding/bucketing 的方式对数据进行处理后，使用 Tensor 来表示数据，进行模型训练，具体示例请参考：[IMDB 数据集使用 BOW 网络的文本分类，使用注意力机制的 LSTM 的机器翻译](#)。
 2. 在使用 padding/bucketing 方案对性能影响极大的场景下，请谨慎升级，并请期待未来的 paddle 对该功能更加易用和高效的实现。

问题：为什么 paddle2.0 以后的版本要废弃 LoDTensor？

- 答复：在 2.0 之前的版本的 paddle 中，向用户暴露了以下的数据表示的概念：
 - `Tensor`：类似于 numpy ndarray 的多维数组。
 - `Variable`：可以简单理解为，在构建静态的计算图时的数据节点。
 - `LoDTensor`：用来表示嵌套的、每条数据长度不一的一组数据。（例：一个 batch 中包含了长度为 3, 10, 7, 50 的四个句子）

这三类不同类型的概念的同时存在，让使用 paddle 的开发者容易感到混淆，需要构建 LoDTensor 类型的数据的情况下具体的实践中，通常也可以使用 padding/bucketing 的最佳实践来达到同样的目的，因此 paddle 2.0 版本起，我们把这些概念统一为 `Tensor` 的概念。在 paddle 2.0 版本起，对于每条数据长度不一的一组数据的处理，您可以参看这篇 Tutorial：[使用注意力机制的 LSTM 的机器翻译](#)。

问题：1.8 开发的静态图代码能在 2.0 版本中运行吗？

- 答复：

所有 1.8 的静态图模型在 2.0 版本中都会报错。

 - 问题分析：

2.0 版本默认开启了动态图模式。即当调用 `import paddle` 后，此时 Paddle 已经运行在动态图模式下。基于 1.8 开发的静态图代码，在 2.0 版本下直接执行会出错。为此，在静态图的一些入口 API 中加入了报错检查，例如直接调用 `fluid.data` 会遇到如下错误：

 - 解决方案：
 1. 旧版本（1.8 及之前版本）静态图下的代码，需要在 `import paddle` 后的头部位置加入 `paddle.enable_static()` 来开启静态图模式，这样才能正常运行。

2. 原来通过 `dygraph guard` 写的动态图代码仍然可以正常执行。但在 2.0 下可以不需要像以前写 `dygraph guard`, 直接按照动态图模式编写代码。
 3. 同时, 请注意对于 GPU 版本的 `paddle`, 在 `import paddle` 时默认开启动态图会选择 `CUDAPlace` 作为默认 `place`。如果要修改 `place`, 可以通过 `paddle.set_device()` 来完成。
-

问题: 2.0 版本中 `loss.backward()` 是否默认清空上个 step 的梯度?

- 答复:

2.0 版本新增动态图梯度累加功能, 起到变相“扩大 BatchSize”的作用, `backward()` 接口默认不清空上个 step 梯度。

- 解决方案:

调用 `optimizer.minimize()` 后, 显式调用 `optimizer.clear_grad()` 来清空梯度。

6.2 安装常见问题

6.2.1 问题: 使用过程中报找不到 tensorrt 库的日志

- 问题描述:

TensorRT dynamic library (`libnvinfer.so`) that Paddle depends on is not configured correctly. (error code is `libnvinfer.so: cannot open shared object file: No such file or directory`)
Suggestions: Check if TensorRT is installed correctly and its version is matched with paddlepaddle you installed. Configure TensorRT dynamic library environment variables as follows:
Linux: set `LD_LIBRARY_PATH` by export `LD_LIBRARY_PATH=...`
Windows: set `PATH` by 'set `PATH=XXX;`

- 问题分析:

遇到该问题是因为使用的 `paddle` 默认开始了 TensorRT, 但是本地环境中没有找到 TensorRT 的库, 该问题只影响使用 [Paddle Inference](#) 开启 TensorRT 预测的场景, 对其它方面均不造成影响。

- 解决办法:

根据提示信息, 在环境变量中加入 TensorRT 的库路径。

6.2.2 问题: Windows 环境下, 使用 pip install 时速度慢, 如何解决?

- 解决方案:

在 pip 后面加上参数-i 指定 pip 源, 使用国内源获取安装包。

- 操作步骤:

1. Python2 情况下, 使用如下命令安装 PaddlePaddle。

```
pip install paddlepaddle -i https://mirror.baidu.com/pypi/simple/
```

2. Python3 情况下, 使用如下命令安装 PaddlePaddle。

```
pip3 install paddlepaddle -i https://mirror.baidu.com/pypi/simple/
```

你还可以通过如下三个地址获取 pip 安装包, 只需修改 -i 后网址即可:

1. <https://pypi.tuna.tsinghua.edu.cn/simple>
2. <https://mirrors.aliyun.com/pypi/simple/>
3. <https://pypi.douban.com/simple/>

6.2.3 问题: 使用 pip install 时报错, PermissionError: [WinError 5] , 如何解决?

- 问题描述:

使用 pip install 时报错, `PermissionError: [WinError 5]`,

`C:\\\\Program Files\\\\python35\\\\Lib\\\\site-packages\\\\graphviz.`

- 报错分析:

用户权限问题导致, 由于用户的 Python 安装到系统文件内 (如 `Program Files/`), 任何的操作都需要管理员权限。

- 解决方法:

选择“以管理员身份运行”运行 CMD, 重新执行安装过程, 使用命令 `pip install paddlepaddle`.

6.2.4 问题：使用 pip install 时报错，`ERROR: No matching distribution found for paddlepaddle`，如何解决？

- 问题描述：

使用 pip install 时报错，`ERROR: Could not find a version that satisfies the requirement paddlepaddle (from versions: none)`

`ERROR: No matching distribution found for paddlepaddle`

- 报错分析：

Python 版本不匹配导致。用户使用的是 32 位 Python，但是对应的 32 位 pip 没有 PaddlePaddle 源。

- 解决方法：

请用户使用 64 位的 Python 进行 PaddlePaddle 安装。

6.2.5 问题：本地使用 import paddle 时报错，`ModuleNotFoundError: No module named 'paddle'`，如何解决？

- 报错分析：

原因在于用户的计算机上可能安装了多个版本的 Python，而安装 PaddlePaddle 时的 Python 和 import paddle 时的 Python 版本不一致导致报错。如果用户熟悉 PyCharm 等常见的 IDE 配置包安装的方法，配置运行的方法，则可以避免此类问题。

- 解决方法：

用户明确安装 Paddle 的 python 位置，并切换到该 python 进行安装。可能需要使用 `python -m pip install paddlepaddle` 命令确保 paddle 是安装到该 python 中。

6.2.6 问题：使用 PaddlePaddle GPU 的 Docker 镜像时报错，`Cuda Error: CUDA driver version is insufficient for CUDA runtime version`, 如何解决？

- 报错分析：

机器上的 CUDA 驱动偏低导致。

- 解决方法：

需要升级 CUDA 驱动解决。

1. Ubuntu 和 CentOS 环境，需要把相关的驱动和库映射到容器内部。如果使用 GPU 的 docker 环境，需要用 nvidia-docker 来运行，更多请参考nvidia-docker。

-
2. Windows 环境，需要升级 CUDA 驱动。
-

6.2.7 问题：使用 PaddlePaddle 时报错，Error: no CUDA-capable device is detected，如何解决？

- 报错分析：

CUDA 安装错误导致。

- 解决方法：

查找“libcudart.so”所在目录，并将其添加到 LD_LIBRARY_PATH 中。

例如：执行 find / -name libcudart.so，发现 libcudart.so 在/usr/local/cuda-10.0/targets/x86_64-linux/lib/libcudart.so 路径下，使用如下命令添加即可。

```
export LD_LIBRARY_PATH=/usr/local/cuda-10.0/targets/x86_64-linux/lib/libcudart.so:$  
↪{LD_LIBRARY_PATH}
```

6.2.8 问题：如何升级 PaddlePaddle？

- 答复：

1. GPU 环境：

```
pip install -U paddlepaddle-gpu
```

或者

```
pip install paddlepaddle-gpu==需要安装的版本号（如2.0）
```

1. CPU 环境：

```
pip install -U paddlepaddle
```

或者

```
pip install paddlepaddle==需要安装的版本号（如2.0）
```

6.2.9 问题：在 GPU 上如何选择 PaddlePaddle 版本？

- 答复：

首先请确定你本机的 CUDA、cuDNN 版本，飞桨目前 pip 安装适配 CUDA 版本 9.0/10.0/10.1/10.2/11.0，CUDA9.0/10.0/10.1/10.2 配合 cuDNN v7.6.5+，CUDA 工具包 11.0 配合 cuDNN v8.0.4。请确定你安装的是适合的版本。更多安装信息见[官网安装文档](#)

6.2.10 问题：import paddle 报错，dlopen: cannot load any more object with static TLS，如何解决？

- 答复：

glibc 版本过低，建议使用官方提供的 docker 镜像或者将 glibc 升级到 2.23+。

6.2.11 问题：python2.7 中，如果使用 Paddle1.8.5 之前的版本，import paddle 时，报错，提示/xxxx/rarfile.py, line820, print(f.filename, file=file), SyntaxError: invalid syntax，如何解决？

- 答复：

rarfile 版本太高，它的最新版本已经不支持 python2.x 了，可以通过 pip install rarfile==3.0 安装 3.0 版本的 rarfile 即可。

6.3 数据及其加载常见问题

6.3.1 问题：如何在训练过程中高效读取数据量很大的数据集？

- 答复：当训练时使用的数据集数据量较大或者预处理逻辑复杂时，如果串行地进行数据读取，数据读取往往会成为训练效率的瓶颈。这种情况下通常需要利用多线程或者多进程的方法异步地进行数据载入，从而提高数据读取和整体训练效率。

paddle 中推荐使用 DataLoader，这是一种灵活的异步加载方式。

该 API 提供了多进程的异步加载支持，可以配置 num_workers 指定异步加载数据的进程数目从而满足不同规模数据集的读取需求。

具体使用方法及示例请参考 API 文档：[paddle.io.DataLoader](#)

6.3.2 问题：使用多卡进行并行训练时，如何配置 DataLoader 进行异步数据读取？

- 答复：paddle 中多卡训练时设置异步读取和单卡场景并无太大差别，动态图模式下，由于目前仅支持多进程多卡，每个进程将仅使用一个设备，比如一张 GPU 卡，这种情况下，与单卡训练无异，只需要确保每个进程使用的是正确的卡即可。

具体示例请参考飞桨 API [paddle.io.DataLoader](#)中的示例。

6.3.3 问题：有拓展 Tensor 维度的 Op 吗？

- 答复：请参考 API [paddle.unsqueeze](#)。
-

6.3.4 问题：如何给图片添加一个通道数，并进行训练？

- 答复：如果是在进入 paddle 计算流程之前，数据仍然是 numpy.array 的形式，使用 numpy 接口 `numpy.expand_dims` 为图片数据增加维度后，再通过 `numpy.reshape` 进行操作即可，具体使用方法可查阅 numpy 的官方文档。

如果是希望在模型训练或预测流程中完成通道的操作，可以使用 paddle 对应的 API [paddle.unsqueeze](#) 和 [paddle.reshape](#)。

6.3.5 问题：如何从 numpy.array 生成一个具有 shape 和 dtype 的 Tensor？

- 答复：在动态图模式下，可以参考如下示例：

```
import paddle
import numpy as np

x = np.ones([2, 2], np.float32)
y = paddle.to_tensor(x)

# 或者直接使用paddle生成tensor
z = paddle.ones([2, 2], 'float32')
```

6.3.6 问题：如何初始化一个随机数的 Tensor？

- 答复：使用 `paddle.rand` 或 `paddle.randn` 等 API。具体请参考：`paddle.rand` 和 `paddle.randn`

6.4 组网、训练、评估常见问题

6.4.1 问题：`stop_gradient=True` 的影响范围？

- 答复：如果静态图中某一层使用 `stop_gradient=True`，那么这一层之前的层都会自动 `stop_gradient=True`，梯度不再回传。

6.4.2 问题：请问 `paddle.matmul` 和 `paddle.multiply` 有什么区别？

- 答复：`matmul` 支持的两个 tensor 的矩阵乘操作。`multiply` 是支持两个 tensor 进行逐元素相乘。

6.4.3 问题：请问 `paddle.gather` 和 `torch.gather` 有什么区别？

- 答复：`paddle.gather` 和 `torch.gather` 的函数签名分别为：

```
paddle.gather(x, index, axis=None, name=None)
torch.gather(input, dim, index, *, sparse_grad=False, out=None)
```

其中，`paddle.gather` 的参数 `x`, `index`, `axis` 分别与 `torch.gather` 的参数 `input`, `index`, `dim` 意义相同。

两者在输入形状、输出形状、计算公式等方面都有区别，具体如下：

- `paddle.gather`
 - 输入形状：`x` 可以是任意的 N 维 Tensor。但 `index` 必须是形状为 [M] 的一维 Tensor，或形状为 [M, 1] 的二维 Tensor。
 - 输出形状：输出 Tensor `out` 的形状 `shape_out` 和 `x` 的形状 `shape_x` 的关系为：

$$\text{shape}_{\text{out}}[i] = \text{shape}_{\text{x}}[i] \text{ if } i \neq \text{axis} \text{ else } M.$$
 - 计算公式： $\text{out}[i_1][i_2] \dots [i_{\text{axis}}] \dots [i_N] = \text{x}[i_1][i_2] \dots [\text{index}[i_{\text{axis}}]] \dots [i_N].$
 - 举例说明：假设 `x` 的形状为 [N1, N2, N3], `index` 的形状为 [M], `axis` 的值为 1，那么输出 `out` 的形状为 [N1, M, N3]，且 $\text{out}[i_1][i_2][i_3] = \text{x}[i_1][\text{index}[i_2]][i_3]$ 。
- `torch.gather`

- 输入形状: `input` 可以是任意的 N 维 Tensor, 且 `index.rank` 必须等于 `input.rank`。
 - 输出形状: 输出 Tensor `out` 的形状与 `index` 相同。
 - 计算公式: $out[i_1][i_2]\dots[i_{dim}]\dots[i_N] = input[i_1][i_2]\dots[index[i_1][i_2]\dots[i_N]]\dots[i_N]$ 。
 - 举例说明: 假设 `x` 的形状为 `[N1, N2, N3]`, `index` 的形状为 `[M1, M2, M3]`, `dim` 的值为 1, 那么输出 `out` 的形状为 `[M1, M2, M3]`, 且 $out[i_1][i_2][i_3] = input[i_1][index[i_1][i_2][i_3]][i_3]$ 。
 - 异同比较
 - 只有当 `x.rank == 1` 且 `index.rank == 1` 时, `paddle.gather` 和 `torch.gather` 功能相同。其余情况两者无法直接互换使用。
 - `paddle.gather` 不支持 `torch.gather` 的 `sparse_grad` 参数。
-

6.4.4 问题: 在模型组网时, `inplace` 参数的设置会影响梯度回传吗? 经过不带参数的 op 之后, 梯度是否会保留下?

- 答复: `inplace` 参数不会影响梯度回传。只要用户没有手动设置 `stop_gradient=True`, 梯度都会保留下。
-

6.4.5 问题: 如何不训练某层的权重?

- 答复: 在 `ParamAttr` 里设置 `learning_rate=0` 或 `trainable` 设置为 `False`。具体请参考文档
-

6.4.6 问题: 使用 CPU 进行模型训练, 如何利用多处理器进行加速?

- 答复: 在 2.0 版本动态图模式下, CPU 训练加速可以从以下两点进行配置:
 1. 使用多进程 `DataLoader` 加速数据读取: 训练数据较多时, 数据处理往往会成为训练速度的瓶颈, `paddle` 提供了异步数据读取接口 `DataLoader`, 可以使用多进程进行数据加载, 充分利用多处理的优势, 具体使用方法及示例请参考 API 文档: [paddle.io.DataLoader](#)。
 2. 推荐使用支持 `MKL` (英特尔数学核心函数库) 的 `paddle` 安装包, `MKL` 相比 `Openblas` 等通用计算库在计算速度上有显著的优势, 能够提升您的训练效率。
-

6.4.7 问题：使用 NVIDIA 多卡运行 Paddle 时报错 Nccl error，如何解决？

- 答复：这个错误大概率是环境配置不正确导致的，建议您使用 NVIDIA 官方提供的方法参考检测自己的环境是否配置正确。具体地，可以使用 NCCL Tests 检测您的环境；如果检测不通过，请登录 NCCL 官网 下载 NCCL，安装后重新检测。

6.4.8 问题：多卡训练时启动失败，Error: Out of all 4 Trainers，如何处理？

- 问题描述：多卡训练时启动失败，显示如下信息：
- 报错分析：主进程发现一号卡（逻辑）上的训练进程退出了。
- 解决方法：查看一号卡上的日志，找出具体的出错原因。`paddle.distributed.launch` 启动多卡训练时，设置 `--log_dir` 参数会将每张卡的日志保存在设置的文件夹下。

6.4.9 问题：训练时报错提示显存不足，如何解决？

- 答复：可以尝试按如下方法解决：
 - 检查是当前模型是否占用了过多显存，可尝试减小 `batch_size`；
 - 开启以下三个选项：

```
#一旦不再使用即释放内存垃圾，=1.0 垃圾占用内存大小达到10G时，释放内存垃圾
export FLAGS_eager_delete_tensor_gb=0.0
#启用快速垃圾回收策略，不等待cuda kernel 结束，直接释放显存
export FLAGS_fast_eager_deletion_mode=1
#该环境变量设置只占用0%的显存
export FLAGS_fraction_of_gpu_memory_to_use=0
```

详细请参考官方文档[存储分配与优化](#) 调整相关配置。

此外，建议您使用[AI Studio](#) 学习与实训社区训练，获取免费 GPU 算力，提升您的训练效率。

6.4.10 问题：如何提升模型训练时的 GPU 利用率？

- 答复：有如下两点建议：
 - 如果数据预处理耗时较长，可使用 `DataLoader` 加速数据读取过程，具体请参考 API 文档：[paddle.io.DataLoader](#)。
 - 如果提高 GPU 计算量，可以增大 `batch_size`，但是注意同时调节其他超参数以确保训练配置的正确性。

以上两点均为比较通用的方案，其他的优化方案和模型相关，可参考官方模型库 `models` 中的具体示例。

6.4.11 问题：如何处理变长 ID 导致程序内存占用过大的问题？

- 答复：请先参考显存分配与优化文档 开启存储优化开关，包括显存垃圾及时回收和 Op 内部的输出复用输入等。若存储空间仍然不够，建议：
 - 降低 `batch_size`；
 - 对 `index` 进行排序，减少 `padding` 的数量。

6.4.12 问题：训练过程中如果出现不收敛的情况，如何处理？

- 答复：不收敛的原因有很多，可以参考如下方式排查：
 - 检查数据集中训练数据的准确率，数据是否有错误，特征是否归一化；
 - 简化网络结构，先基于 `benchmark` 实验，确保在 `baseline` 网络结构和数据集上的收敛结果正确；
 - 对于复杂的网络，每次只增加一个改动，确保改动后的网络正确；
 - 检查网络在训练数据上的 Loss 是否下降；
 - 检查学习率、优化算法是否合适，学习率过大导致不收敛；
 - 检查 `batch_size` 设置是否合适，`batch_size` 过小会导致不收敛；
 - 检查梯度计算是否正确，是否有梯度过大的情况，是否为 `Nan`。

6.4.13 问题：Loss 为 NaN，如何处理？

- 答复：可能由于网络的设计问题，Loss 过大（Loss 为 NaN）会导致梯度爆炸。如果没有改网络结构，但是出现了 NaN，可能是数据读取导致，比如标签对应关系错误。还可以检查下网络中是否会出现除 0，log0 的操作等。

6.4.14 问题：训练后的模型很大，如何压缩？

- 答复：建议您使用飞桨模型压缩工具PaddleSlim。PaddleSlim 是飞桨开源的模型压缩工具库，包含模型剪裁、定点量化、知识蒸馏、超参搜索和模型结构搜索等一系列模型压缩策略，专注于模型小型化技术。

6.4.15 问题：load_inference_model 在加载预测模型时能否用 py_reader 读取？

- 答复：目前 load_inference_model 加载进行的模型还不支持 py_reader 输入。

6.4.16 问题：预测时如何打印模型中每一步的耗时？

- 答复：可以在设置 config 时使用 config.enable_profile() 统计预测时每个算子和数据搬运的耗时。对于推理 api 的使用，可以参考官网文档Python 预测 API 介绍。示例代码：

```
# 设置config:  
def set_config(args):  
    config = Config(args.model_file, args.params_file)  
    config.disable_gpu()  
    # enable_profile() 打开后会统计每一步耗时  
    config.enable_profile()  
    config.switch_use_fetch_ops(False)  
    config.switch_specify_input_names(True)  
    config.switch_ir_optim(False)  
    return config
```

6.4.17 问题：模型训练时如何进行梯度裁剪？

- 答复：设置 Optimizer 中的 grad_clip 参数值。

6.4.18 问题：静态图模型如何拿到某个 variable 的梯度？

- 答复：飞桨提供以下三种方式，用户可根据需求选择合适的方法：
 - 使用paddle.static.Print()接口，可以打印中间变量及其梯度。
 - 将变量梯度名放到 fetch_list 里，通过Executor.run()获取，一般 variable 的梯度名是 variable 的名字加上”@GRAD”。
 - 对于参数（不适用于中间变量和梯度），还可以通过Scope.find_var()接口，通过变量名字查找对应的 tensor。

后两个方法需要使用变量名，飞桨中变量的命名规则请参见Name。

```
# paddlepaddle>=2.0
import paddle
import numpy as np

paddle.enable_static()
data = paddle.static.data('data', shape=[4, 2])
out = paddle.static.nn.fc(x=data, size=1, num_flatten_dims=1, name='fc')

loss = paddle.mean(out)
loss = paddle.static.Print(loss) # 通过 Print 算子打印中间变量及梯度
opt = paddle.optimizer.SGD(learning_rate=0.01)
opt.minimize(loss)

exe = paddle.static.Executor()
exe.run(paddle.static.default_startup_program())
loss, loss_g, fc_bias_g = exe.run(
    paddle.static.default_main_program(),
    feed={'data': np.random.rand(4, 2).astype('float32')},
    fetch_list=[loss, loss.name + '@GRAD', 'fc.b_0@GRAD']) # 通过将变量名加入到fetch_list 获取变量

print(loss, loss_g, fc_bias_g)
print(paddle.static.global_scope().find_var('fc.b_0').get_tensor()) # 通过scope.find_var 获取变量
```

6.4.19 问题：paddle 有对应 torch.masked_fill 函数 api 吗，还是需要自己实现？

- 答复：由于框架设计上的区别，没有对应的 api，但是可以使用 paddle.where 实现相同的功能。

```
# paddlepaddle >= 2.0
import paddle

paddle.seed(123)
x = paddle.rand([3, 3], dtype='float32')
# Tensor(shape=[3, 3], dtype=float32, place=CUDAPlace(0), stop_gradient=True,
#        [[0.00276479, 0.45899123, 0.96637046],
#         [0.66818708, 0.05855134, 0.33184195],
#         [0.34202638, 0.95503175, 0.33745834]])

mask = paddle.randint(0, 2, [3, 3]).astype('bool')
# Tensor(shape=[3, 3], dtype=bool, place=CUDAPlace(0), stop_gradient=True,
#        [[True, True, False],
#         [True, True, True],
#         [True, True, True]])

def masked_fill(x, mask, value):
    y = paddle.full(x.shape, value, x.dtype)
    return paddle.where(mask, y, x)

out = masked_fill(x, mask, 2)
# Tensor(shape=[3, 3], dtype=float32, place=CUDAPlace(0), stop_gradient=True,
#        [[2., 2., 0.96637046],
#         [2., 2., 2.],
#         [2., 2., 2.]])
```

6.4.20 问题：在 paddle 中如何实现 torch.nn.utils.rnn.pack_padded_sequence 和 torch.nn.utils.rnn.pad_packed_sequence 这两个 API？

- 答复：目前 paddle 中没有和上述两个 API 完全对应的实现。关于 torch 中这两个 API 的详细介绍可以参考知乎上的文章 pack_padded_sequence 和 pad_packed_sequence：pack_padded_sequence 的功能是将 mini-batch 数据进行压缩，压缩掉无效的填充值，然后输入 RNN 网络中；pad_packed_sequence 则是把 RNN 网络输出的压紧的序列再填充回来，便于进行后续的处理。在 paddle 中，大家可以在 GRU、LSTM 等 RNN 网络中输入含有填充值的 mini-batch 数据的同时传入对应的 sequence_length 参数实现上述等价功能，具体用法可以参考 RNN。

6.4.21 问题：paddle 是否有爱因斯坦求和（einsum）这个 api?

- 答复：paddle 在 2.2rc 版本之后，新增了 paddle.einsum，在 develop 和 2.2rc 之后的版本中都可以正常使用。

At least one body element must separate transitions; adjacent transitions are not allowed.

6.4.22 问题：BatchNorm 在训练时加载预测时保存的模型参数时报错 AssertionError: Optimizer set error, batch_norm_1.w_0_moment_0 should in state dict.

- 答复：BatchNorm 在 train 模式和 eval 模式下需要的变量有差别，在 train 模式下要求传入优化器相关的变量，在 eval 模式下不管是保存参数还是加载参数都是不需要优化器相关变量的，因此如果在 train 模式下加载 eval 模式下保存的 checkpoint，没有优化器相关的变量则会报错。如果想在 train 模式下加载 eval 模式下保存的 checkpoint 的话，用 paddle.load 加载进来参数之后，通过 set_state_dict 接口把参数赋值给模型，参考以下示例：

```
import paddle

bn = paddle.nn.BatchNorm(3)
bn_param = paddle.load('./bn.pdparams')
bn.set_state_dict()
```

Document may not end with a transition.

6.5 模型保存常见问题

6.5.1 问题：静态图的 save 接口与 save_inference_model 接口存储的结果有什么区别？

- 答复：主要差别在于保存结果的应用场景：

1. save 接口 (2.0 的 paddle.static.save 或者 1.8 的 fluid.io.save)

该接口用于保存训练过程中的模型和参数，一般包括 *.pdmodel, *.pdparams, *.pdopt 三个文件。其中 *.pdmodel 是训练使用的完整模型 program 描述，区别于推理模型，训练模型 program 包含完整的网络，包括前向网络，反向网络和优化器，而推理模型 program 仅包含前向网络，*.pdparams 是训练网络的参数 dict，key 为变量名，value 为 Tensor array 数值，*.pdopt 是训练优化器的参数，结构与 *.pdparams 一致。

2. save_inference_model 接口 (2.0 的 paddle.static.save_inference_model 或者 1.8 的 fluid.io.save_inference_model)

该接口用于保存推理模型和参数，2.0 的 paddle.static.save_inference_model 保存结果为 *.pdmodel 和 *.pdiparams 两个文件，其中 *.pdmodel 为推理使用的模型 program 描述，*.pdiparams 为推理用的参数，这里存储格式与 *.pdparams 不同（注意两者后缀差个 i），*.pdiparams 为二进制 Tensor 存储格式，不含变量名。1.8 的 fluid.io.save_inference_model 默认保存结果为 __model__ 文件，和以参数名为文件名的多个分散参数文件，格式与 2.0 一致。

3. 关于更多 2.0 动态图模型保存和加载的介绍可以参考教程：[模型存储与载入](#)

6.5.2 问题：增量训练中，如何保存模型和恢复训练？

- 答复：在增量训练过程中，不仅需要保存模型的参数，也需要保存优化器的参数。

具体地，在 2.0 版本中需要使用 Layer 和 Optimizer 的 state_dict 和 set_state_dict 方法配合 paddle.save/load 使用。简要示例如下：

```
import paddle

emb = paddle.nn.Embedding(10, 10)
layer_state_dict = emb.state_dict()
paddle.save(layer_state_dict, "emb.pdparams")

scheduler = paddle.optimizer.lr.NoamDecay(
    d_model=0.01, warmup_steps=100, verbose=True)
adam = paddle.optimizer.Adam(
    learning_rate=scheduler,
    parameters=emb.parameters())
opt_state_dict = adam.state_dict()
paddle.save(opt_state_dict, "adam.pdopt")

load_layer_state_dict = paddle.load("emb.pdparams")
load_opt_state_dict = paddle.load("adam.pdopt")

emb.set_state_dict(para_state_dict)
adam.set_state_dict(opti_state_dict)
```

6.5.3 问题：paddle.load 可以加载哪些 API 产生的结果呢？

- 答复：

为了更高效地使用 paddle 存储的模型参数，paddle.load 支持从除 paddle.save 之外的其他 save 相关 API 的存储结果中载入 state_dict，但是在不同场景中，参数 path 的形式有所不同：

1. 从 paddle.static.save 或者 paddle.Model().save(training=True) 的保存结果载入：path 需要是完整的文件名，例如 model.pdparams 或者 model.opt；
2. 从 paddle.jit.save 或者 paddle.static.save_inference_model 或者 paddle.Model().save(training=False) 的保存结果载入：path 需要是路径前缀，例如 model/mnist，paddle.load 会从 mnist.pdmodel 和 mnist.pdiparams 中解析 state_dict 的信息并返回。
3. 从 paddle 1.x API paddle.fluid.io.save_inference_model 或者 paddle.fluid.io.save_params/save_persistables 的保存结果载入：path 需要是目录，例如 model，此处 model 是一个文件夹路径。

需要注意的是，如果从 paddle.static.save 或者 paddle.static.save_inference_model 等静态图 API 的存储结果中载入 state_dict，动态图模式下参数的结构性变量名将无法被恢复。在将载入的 state_dict 配置到当前 Layer 中时，需要配置 Layer.set_state_dict 的参数 use_structured_name=False。

6.5.4 问题：paddle.save 是如何保存 state_dict, Layer 对象, Tensor 以及包含 Tensor 的嵌套 list、tuple、dict 的呢？

- 答复：

1. 对于 state_dict 保存方式与 paddle2.0 完全相同，我们将 Tensor 转化为 numpy.ndarray 保存。
2. 对于其他形式的包含 Tensor 的对象（Layer 对象，单个 Tensor 以及包含 Tensor 的嵌套 list、tuple、dict），在动态图中，将 Tensor 转化为 tuple(Tensor.name, Tensor.numpy()); 在静态图中，将 Tensor 直接转化为 numpy.ndarray。之所以这样做，是因为当在静态图中使用动态保存的模型时，有时需要 Tensor 的名字因此将名字保存下来，同时，在 load 时区分这个 numpy.ndarray 是由 Tensor 转化而来还是本来就是 numpy.ndarray；保存静态图的 Tensor 时，通常通过 Variable.get_value 得到 Tensor 再使用 paddle.save 保存 Tensor，此时，Variable 是有名字的，这个 Tensor 是没有名字的，因此将静态图 Tensor 直接转化为 numpy.ndarray 保存。

此处动态图 Tensor 和静态图 Tensor 是不相同的，动态图 Tensor 有 name、stop_gradient 等属性；而静态图的 Tensor 是比动态图 Tensor 轻量级的，只包含 place 等基本信息，不包含名字等。

6.5.5 问题：将 Tensor 转换为 numpy.ndarray 或者 tuple(Tensor.name, Tensor.numpy()) 不是唯一可译编码，为什么还要做这样的转换呢？

- 答复：
 - 我们希望 `paddle.save` 保存的模型能够不依赖 paddle 框架就能够被用户解析（pickle 格式模型），这样用户可以方便的做调试，轻松的看到保存的参数的数值。其他框架的模型与 paddle 模型做转化也会容易很多。
 - 我们希望保存的模型尽量小，只保留了能够满足大多场景的信息（动态图保存名字和数值，静态图只保存数值），如果需要 Tensor 的其他信息（例如 `stop_gradient`），可以向被保存的对象中添加这些信息，`load` 之后再还原这些信息。这样的转换方式可以覆盖绝大多数场景，一些特殊场景也是可以通过一些方法解决的，如下面的问题。

6.5.6 问题：什么情况下 save 与 load 的结果不一致呢，应该如何避免这种情况发生呢？

- 答复：

以下情况会造成 `save` 与 `load` 的结果不一致：

 - 被保存的对象包含动态图 `Tensor` 同时包含 `tuple(string, numpy.ndarray)`；
 - 被保存的对象包含静态图 `Tensor`，同时包含 `numpy.ndarray` 或者 `tuple(string, numpy.ndarray)`；
 - 被保存的对象只包含 `numpy.ndarray`，但是包含 `tuple(string, numpy.ndarray)`。

针对这些情况我们有以下建议：

- 被保存的对象（包括 `Layer` 对象中的 `ParamBase`），避免包含形如 `tuple(string, numpy.ndarray)` 的对象；
- 如果被保存的对象包含 `numpy.ndarray`，尽量在 `load` 时设置 `return_numpy = True`。
- 对于 `Layer` 对象，只保存参数的值和名字，如果需要其他信息（例如 `stop_gradient`），请将手将这些信息打包成 `dict` 等，一并保存。

6.5.7 问题：paddle 2.x 如何保存模型文件？如何保存 paddle 1.x 中的 model 文件？

- 答复：
 - 在 paddle2.x 可使用 `paddle.jit.save` 接口以及 `paddle.static.save_inference_model`，通过指定 `path` 来保存成为 `path.pdmodel` 和 `path.pdiparams`，可对应 paddle1.x 中使用 `save_inference_model` 指定 `dirname` 和 `params_filename` 生成 `dirname/_model_` 和 `dirname/params` 文件。paddle2.x 保存模型文件详情可参考：
 - `paddle.jit.save/load`
 - `paddle.static.save/load_inference_model`

- 如果想要在 paddle2.x 中读取 paddle 1.x 中的 model 文件，可参考：
 - 兼容载入旧格式模型

6.5.8 问题：paddle 如何单独 load 存下来所有模型变量中某一个变量，然后修改变量中的值？

- 答复：
 - 如果目的是修改存储变量的值，可以使用 `paddle.save` 保存下来所有变量，然后再使用 `paddle.load` 将所有变量载入后，查找目标变量进行修改，示例代码如下：

```
import paddle

layer = paddle.nn.Linear(3, 4)
path = 'example/model.pdparams'
paddle.save(layer.state_dict(), path)
layer_param = paddle.load(path)
# 修改 fc_0.b_0 的值
layer_param["fc_0.b_0"] = 10
```

- 如果目的是单独访问某个变量，需要单独存储然后再单独读取，示例代码如下：

```
import paddle

layer = paddle.nn.Linear(3, 4)
path_w = 'example/weight.tensor'
path_b = 'example/bias.tensor'
paddle.save(layer.weight, path_w)
paddle.save(layer.bias, path_b)
tensor_bias = paddle.load(path_b)
tensor_bias[0] = 10
```

更多介绍请参考以下 API 文档：

- `paddle.save`
- `paddle.load`

6.6 参数调整常见问题

6.6.1 问题：如何将本地数据传入 paddle.nn.embedding 的参数矩阵中？

- 答复：需将本地词典向量读取为 NumPy 数据格式，然后使用 `paddle.nn.initializer.Assign` 这个 API 初始化 `paddle.nn.embedding` 里的 `param_attr` 参数，即可实现加载用户自定义（或预训练）的 Embedding 向量。

6.6.2 问题：如何实现网络层中多个 feature 间共享该层的向量权重？

- 答复：你可以使用 `paddle.ParamAttr` 并设定一个 `name` 参数，然后再将这个类的对象传入网络层的 `param_attr` 参数中，即将所有网络层中 `param_attr` 参数里的 `name` 设置为同一个，即可实现共享向量权重。如使用 `embedding` 层时，可以设置 `param_attr=paddle.ParamAttr(name="word_embedding")`，然后把 `param_attr` 传入 `embedding` 层中。

6.6.3 问题：使用 optimizer 或 ParamAttr 设置的正则化和学习率，二者什么差异？

- 答复：ParamAttr 中定义的 `regularizer` 优先级更高。若 ParamAttr 中定义了 `regularizer`，则忽略 Optimizer 中的 `regularizer`；否则，则使用 Optimizer 中的 `regularizer`。ParamAttr 中的学习率默认为 1.0，在对参数优化时，最终的学习率等于 optimizer 的学习率乘以 ParamAttr 的学习率。

6.6.4 问题：如何导出指定层的权重，如导出最后一层的 weights 和 bias？

- 答复：
 - 在动态图中，使用 `paddle.save` API，并将最后一层的 `layer.state_dict()` 传入至 `save` 方法的 `obj` 参数即可，然后使用 `paddle.load` 方法加载对应层的参数值。详细可参考 API 文档[save](#) 和[load](#)。
 - 在静态图中，使用 `paddle.static.save_vars` 保存指定的 `vars`，然后使用 `paddle.static.load_vars` 加载对应层的参数值。具体示例请见 API 文档：[load_vars](#) 和 [save_vars](#)。

6.6.5 问题：训练过程中如何固定网络和 Batch Normalization (BN)？

- 答复：
 - 对于固定 BN：设置 `use_global_stats=True`，使用已加载的全局均值和方差：`global_mean/variance`，具体内容可查看官网 API 文档[batch_norm](#)。
 - 对于固定网络层：如：`stage1 → stage2 → stage3`，设置 `stage2` 的输出，假设为 `y`，设置 `y.stop_gradient=True`，那么，`stage1→stage2` 整体都固定了，不再更新。

6.6.6 问题：训练的 step 在参数优化器中是如何变化的？

- 答复：
- `step` 表示的是经历了多少组 `mini_batch`，其统计方法为 `exe.run`(对应 `Program`) 运行的当前次数，即每运行一次 `exe.run`，`step` 加 1。举例代码如下：

```
# 执行下方代码后相当于step增加了N x Epoch总数
for epoch in range(epochs):
    # 执行下方代码后step相当于自增了N
    for data in [mini_batch_1, 2, 3...N]:
        # 执行下方代码后step += 1
        exe.run(data)
```

6.6.7 问题：如何修改全连接层参数，比如 weight, bias？

- 答复：可以通过 `param_attr` 设置参数的属性，`paddle.ParamAttr(initializer=paddle.nn.initializer.Normal(0.0, 0.02), learning_rate=2.0)`，如果 `learning_rate` 设置为 0，该层就不参与训练。也可以构造一个 `numpy` 数据，使用 `paddle.nn.initializer.Assign` 来给权重设置想要的值。

6.6.8 问题：如何进行梯度裁剪？

- 答复：Paddle 的梯度裁剪方式需要在 `Optimizer` 中进行设置，目前提供三种梯度裁剪方式，分别是 `paddle.nn.ClipGradByValue`（设定范围值裁剪）、`paddle.nn.ClipGradByNorm`（设定 L2 范数裁剪）、`paddle.nn.ClipGradByGlobalNorm`（通过全局 L2 范数裁剪），需要先创建一个该类的实例对象，然后将其传入到优化器中，优化器会在更新参数前，对梯度进行裁剪。

注：该类接口在动态图、静态图下均会生效，是动静统一的。目前不支持其他方式的梯度裁剪。

```

linear = paddle.nn.Linear(10, 10)
clip = paddle.nn.ClipGradByNorm(clip_norm=1.0) # 可以选择三种裁剪方式
sgd = paddle.optimizer.SGD(learning_rate=0.1, parameters=linear.parameters(), grad_
clip=clip)
sgd.step()                                     # 更新参数前，会先对参数的梯度进行裁剪

```

了解更多梯度裁剪知识

6.6.9 问题：如何在同一个优化器中定义不同参数的优化策略，比如 bias 的参数 weight_decay 的值为 0.0，非 bias 的参数 weight_decay 的值为 0.01？

- 答复：
 1. AdamW 的参数 apply_decay_param_fun 可以用来选择哪些参数使用 decay_weight 策略。
 2. 在创建 Param 的时候，可以通过设置 ParamAttr 的属性来控制参数的属性。

6.6.10 问题：paddle fluid 如何自定义优化器，自定义更新模型参数的规则？

- 答复：
 1. 要定义全新优化器，自定义优化器中参数的更新规则，可以通过继承 fluid.Optimizer，重写 _append_optimize_op 方法实现。不同优化器实现原理各不相同，一般流程是先获取 learning_rate，gradients 参数，可训练参数，以及该优化器自身特别需要的参数，然后实现更新参数的代码，最后返回更新后的参数。在实现更新参数代码时，可以选择直接调用 paddle 的 API 或者使用自定义原生算子。在使用自定义原生算子时，要注意动态图与静态图调用方式有所区别：需要首先使用 framework.in_dygraph_mode() 判断是否为动态图模式，如果是动态图模式，则需要调用 paddle._C_ops 中相应的优化器算子；如果不是动态图模式，则需要调用 block.append_op 来添加优化器算子。代码样例可参考 paddle 源码 中 AdamOptimizer 等优化器的实现。
 2. 使用现有的常用优化器，可以在创建 Param 的时候，可以通过设置 ParamAttr 的属性来控制参数的属性，可以通过设置 regularizer，learning_rate 等参数简单设置参数的更新规则。

6.7 分布式训练常见问题

6.7.1 综合问题

问题：怎样了解飞桨分布式 Fleet API 用法？

- 答复：可查看覆盖高低阶应用的分布式用户文档和分布式 API 文档

问题：机房训练模型的分布式环境用什么比较合适？

- 答复：推荐使用 K8S 部署，K8S 的环境搭建可参考文档

问题：目前飞桨分布式对哪些模型套件/工具支持？

- 答复：
- 多机多卡支持 paddlerec, PGL, paddleHelix, paddleclas, paddlenlp, paddledetection。
 - 单机多卡支持全部飞桨的模型套件和高层 API 写法，无需修改单卡训练代码，默认启用全部可见的卡。

问题：怎样自定义单机多卡训练的卡数量？

- 答复：如果直接使用飞桨模型套件（paddleclas, paddleseg 等）或高层 API 写的代码，可以直接用这条命令指定显卡启动程序，文档源代码不用改（文档内不要用 set_device 指定卡）：python3 -m paddle.distributed.launch --gpus="1, 3" train.py 使用基础 API 的场景下，在程序中修改三处：
 - 第 1 处改动，import 库 import paddle.distributed as dist
 - 第 2 处改动，初始化并行环境 dist.init_parallel_env()
 - 第 3 处改动，对模型增加 paddle.DataParallel 封装 net = paddle.DataParallel(paddle.vision.models.LeNet()) 修改完毕就可以使用 python3 -m paddle.distributed.launch --gpus="1, 3" xxx 来启动了。可参考[AIstudio 项目示例](#)

6.7.2 Fleet API 的使用

问题：飞桨 2.0 版本分布式 Fleet API 的目录在哪？

- 答复：2.0 版本分布式 API 从 paddle.fluid.incubate.fleet 目录挪至 paddle.distributed.fleet 目录下，且对部分 API 接口做了兼容升级。import 方式如下：

```
import paddle.distributed.fleet as fleet
fleet.init()
```

不再支持老版本 paddle.fluid.incubate.fleet API，2.0 版本会在分布式计算图拆分的阶段报语法相关错误。未来的某个版本会直接移除废弃 paddle.fluid 目录下的 API。

问题：飞桨 2.0 版本的 fleet 配置初始化接口 init 和 init_server 用法有什么变化？

- 答复：
 1. fleet.init 接口，2.0 版本支持 role_maker, is_collective, strategy 等参数，且均有缺省值，老版本仅支持 role_maker，且无缺省配置。[点击这里](#) 参考 2.0 Fleet API 的使用方式。
 2. fleet.init_server 接口，除支持传入 model_dir 之外，2.0 版本还支持传入 var_names，加载指定的变量。
-

问题：飞桨 2.0 版本的分布式 paddle.static.nn.sparse_embedding 和 paddle.nn.embedding 有什么差别？

- 答复：paddle.nn.embedding 和 paddle.static.nn.sparse_embedding 的稀疏参数将会在每个 PServer 段都用文本的一部分保存，最终整体拼接起来是完整的 embedding。推荐使用 paddle.static.nn.sparse_embedding 直接采用分布式预估的方案。虽然 nn.embedding 目前依旧可以正常使用，但后续的某个版本会变成与使用 paddle.static.nn.sparse_embedding 一样的保存方案。老版本中使用的 0 号节点的本地预测功能在加载模型的时候会报模型加载错误。
-

问题：飞桨 2.0 分布式可以用哪些配置类？

- 答复：2.0 之后统一为 paddle.distributed.fleet.DistributedStrategy()，与下述老版本配置类不兼容。2.0 之前的版本参数服务器配置类：paddle.fluid.incubate.fleet.parameter_server.distribute_transpiler.distributed_strategy.DistributedStrategy，2.0 之前的版本 collective 模式配置类：paddle.fluid.incubate.fleet.collective.DistributedStrategy
-

问题：飞桨 2.0 分布式配置项统一到 DistributedStrategy 后有哪些具体变化？

- 答复：2.0 版本之后，建议根据 [DistributedStrategy 文档](#) 和 [BuildStrategy 文档](#) 修改配置选项。
2.0 版本将 3 个环境变量配置变为 DistributedStrategy 配置项，3 个环境变量将不生效，包括
 - FLAGS_sync_nccl_allreduce → strategy.sync_nccl_allreduce
 - FLAGS_fuse_parameter_memory_size → strategy.fuse_grad_size_in_MB
 - FLAGS_fuse_parameter_groups_size → strategy.fuse_grad_size_in_TFLOPS

DistributedStrategy 中 exec_strategy 配置项不兼容升级为 execution_strategy。

DistributedStrategy 中 forward_recompute 配置项不兼容升级为 recompute。

DistributedStrategy 中 recompute_checkpoints 配置项不兼容升级为 recompute_configs 字典下的字段，如下：

```
import paddle.distributed.fleet as fleet
strategy = fleet.DistributedStrategy()
strategy.recompute = True
strategy.recompute_configs = {
    "checkpoints": ["x", "y"],
    "enable_offload": True,
    "checkpoint_shape": [100, 512, 1024]}
```

DistributedStrategy 中 use_local_sgd 配置项变为不兼容升级为 localsgd。

问题：飞桨 2.0 分布式 Fleet 的 program 接口是否还能继续用？

- 答复：2.0 版本后，fleet 接口下 main_program 和 _origin_program 均已废弃，会报错没有这个变量，替换使用 paddle.static.default_main_program 即可。

问题：怎样在本地测试 Fleet API 实现的分布式训练代码是否正确？

- 答复：首先写好分布式 train.py 文件
 - 在 PServer 模式下，命令行模拟启动分布式：python -m paddle.distributed.launch_ps --worker_num 2 --server_num 2 train.py
 - 在 Collective 模式下，命令改为 python -m paddle.distributed.launch --gpus=0,1 train.py

问题：Paddle Fleet 怎样做增量训练，有没有文档支持？

- 答复：增量训练可参考[文档示例](#)

问题：飞桨 2.0 分布式 `distributed_optimizer` 如何使用自动混合精度 amp 的 optimizer?

- 答复：amp_init 接口支持 pure_fp16，可以直接调用 `optimizer.amp_init`。
-

问题：Paddle Fleet 可以在 K8S GPU 集群上利用 CPU 资源跑 pserver 模式的 MPI 程序吗？

- 答复：可以，GPU 可设置为 trainer。
-

问题：使用 Fleet Collective 模式进行开发时，已使用 `fleet.distributed_optimizer` 对 `optimizer` 和 `fleet.DistributedStrategy` 包了一层。想确认模型是否也需要使用 `fleet.distributed_model` 再包一层？

- 答复：需要将 `fleet.distributed_model` 在封装一层。原因是动态图主要在 `fleet.distributed_model` 进行分布式设计，静态图是在 `fleet.distributed_optimizer` 进行分布式设计。所以，如果不区分动态图和静态图，两个接口都是需要的。
-

6.7.3 环境配置和训练初始化

问题：分布式环境变量 FLAGS 参数定义可以在哪查看，比如 `communicator` 相关的？

- 答复：参考使用 `DistributedStrategy` 配置分布式策略。
-

问题：2.0 分布式训练的启动命令有什么变化？

- 答复：为了统一启动分布式 Collective/PS 模式任务方式以及易用性考虑，2.0 版本中 `launch/fleetrun` 启动分布式任务时参数产生不兼容升级，`--cluster_node_ips` 改为`--ips`，`--selected_gpus` 改为`--gpus`、`--node_ip`、`--use_paddlecloud`、`--started_port`、`--log_level`、`--print_config` 5 个参数已废弃，使用旧参数会直接报错没有此参数。代码迁移至 `python/paddle/distributed/fleet/launch.py`。
-

问题：分布式环境依赖为什么出现第三方 libssl 库的依赖？

- 答复：分布式 RPC 从 GRPC 迁移至 BRPC，会导致在运行时依赖 libssl 库。使用 docker 的情况下，基础镜像拉一下官方最新的 docker 镜像，或自行安装 libssl 相关的依赖也可以。未安装 libssl 的情况下，import paddle 的时候，出现找不到 libssl.so 的库文件相关报错。使用 MPI 的情况下，需要将编译包时用到的 libssl.so、libcrypto.so 等依赖手动通过 LD_LIBRARY_PATH 进行指定。
-

6.7.4 分布式的动态图模式

问题：飞桨 2.0 版本动态图 DataParallel 用法有哪些简化？

+ 答复：老版本用法依然兼容，建议使用以下新用法：apply_collective_grads、scale_loss 可以删除不使用。loss 会根据环境除以相应的卡数，scale_loss 不再进行任何处理。

问题：飞桨 2.0 版本调用 model.eval 之后不再自动关闭反向计算图的构建，引入显存的消耗增加，可能会引入 OOM，怎么解决？

- 答复：动态图 no_grad 和 model.eval 解绑，应使用 with paddle.no_grad()：命令，显示关闭反向计算图的构建。
-

问题：飞桨 2.0 版本动态图环境初始化新接口怎样用？

- 答复：建议调用新接口 paddle.distributed.init_parallel_env，不需要输入参数。1.8 的 fluid.dygraph.prepare_context 依然兼容。
-

问题：分布式支持哪些飞桨 2.0 版本的模型保存和加载接口？

- 答复：与单机相同，分布式动态图推荐使用 paddle.jit.save 保存，使用 paddle.jit.load 加载，无需切换静态图，存储格式与推理模型存储一致。对比 1.8 动态图使用不含控制流的模型保存接口 TracedLayer.save_inference_model，含控制流的模型保存接口 ProgramTranslator.save_inference_model，加载模型需要使用静态图接口 fluid.io.load_inference_model。fluid.save_dygraph 和 fluid.load_dygraph 升级为 paddle.save 和 paddle.load，推荐使用新接口。paddle.save 不再默认添加后缀，建议用户指定使用标准后缀（模型参数：.pdparams，优化器参数：.pdopt）。

问题：飞桨 2.0 版本为什么不能使用 minimize 和 clear_gradient？

- 答复：2.0 版本中重新实现 optimizer，放在 paddle.optimizer，建议使用新接口和参数。老版本的 paddle.fluid.optimizer 仍然可用。

新版增加接口 step 替换 minimize。老版动态图需要调用 loss.backward()，用 minimize 来表示梯度的更新行为，词语意思不太一致。

新版使用简化的 clear_grad 接口替换 clear_gradient。

6.7.5 报错查错

问题：集合通信 Collective 模式报参数未初始化的错误是什么原因？

- 答复：2.0 版本需要严格先 run(startup_program)，然后再调用 fleet.init_worker() 启动 worker 端通信相关，并将 0 号 worker 的参数广播出去完成其他节点的初始化。先 init_worker，再 run(startup_program)，会报参数未初始化的错误

2.0 之前的版本是在 server 端做初始化，无需 0 号节点广播，所以 init_worker() 可以在 run(startup_program) 执行。

问题：分布式任务跑很久 loss 突然变成 nan 的原因？

- 答复：可设置环境变量 export FLAGS_check_nan_inf=1 定位出错的地方，可以从 checkpoint 开始训练，参数服务器和集合通信模式均可使用这种方式查错。
-

问题：任务卡在 role init 怎么解决？

- 答复：通常可能是 gloo 的初始化问题，需要检查是否有节点任务挂了。建议调小 train_data 配置的数据量，由于启动 trainer 前要下载数据，大量数据会导致拖慢。

6.8 其他常见问题

6.8.1 问题：使用 X2paddle 从 Caffe 转 Paddle model 时，报错

`TypeError: __new__() got an unexpected keyword argument
'serialized_options'`，如何处理？

- 答复：这是由于 ProtoBuf 版本较低导致，将 protobuf 升级到 3.6.0 即可解决。
-

6.8.2 问题：Windows 环境下，出现”Windows not support stack backtrace yet”，如何处理？

- 答复：Windows 环境下，遇到程序报错不会详细跟踪内存报错内容。这些信息对底层开发者更有帮助，普通开发者不必关心这类警告。如果想得到完整内存追踪错误信息，可以尝试更换至 Linux 系统。

Document may not end with a transition.

Chapter 7

2.3.1 Release Note

7.1 1. 重要更新

- 2.3.1 版本是在 2.3 版本的基础上修复了已知问题，并且发布了支持 CUDA 11.6 的安装包。

7.2 2. 训练框架（含分布式）

7.2.1 (1) 功能优化

API

- 修 改 `paddle.nn.initializer.KaimingUniform` 和 `paddle.nn.initializer.KaimingNormal` 两种初始化方式，使其支持多种类型的激活函数。(#43721, #43827)
- 优化 `paddle.io.DataLoader` 的数据预读取功能，使其支持设置了 `prefetch_factor` 设定的预读取数据的缓存数量，避免在读取大块数据时出现 IO 阻塞。(#43674)

新动态图执行机制

- 修改新动态图 API 逻辑中 `optional` 类型 `Tensor` 的初始化方法，防止被提前析构导致数据异常。(#42561)

全新静态图执行器

- 延迟初始化执行器中的线程池，避免只执行一轮的 program (如 save、load、startup_program 等) 创建线程池。(#43768)

混合精度训练

- 设置 paddle.nn.Layer 中 set_state_dict 中禁用 state_dict hook。(#43407)

分布式训练

- 使 paddle.incubate.nn.functional.fused_attention 和 paddle.incubate.nn.functional.fused_feedforward 支持张量模型并行。(#43505)

其他

- 调整框架算子内核打印字符串的格式，便于进行自动化拆分解析。(#42931)
- 更新模型量化 API，支持 rounding to nearest ties to even 的四舍五入方式，支持量化取值范围 [-128, 127]。(#43829)
- 量化感知训练适配支持 AMP 混合精度训练。(#43689)
- 量化感知训练在启动时新增 progress bar，便于查看量化初始化进度，统计 out_threshold 时跳过 scale op，加速初始化过程。(#43454)
- 动态图量化训练支持 conv 和 bn 融合，静态图离线量化支持设置 skip_tensor_list 来跳过某些层不做量化。(#43301)

7.2.2 (2) 性能优化

- 优化 paddle.incubate.nn.functional.fused_attention 和 paddle.incubate.nn.functional.fused_feedforward 算子，增加 add_residual 属性，用以控制最后一步是否进行加 residual 操作，CAE 模型性能提升 7.7%。(#43719)
- 优化 linspace 算子，将 start、stop、num 三个输入 Tensor 初始化在 CPU 上，避免在算子中进行 GPU -> CPU 拷贝，SOLOv2 模型性能提升 6%。(#43746)

7.2.3 (3) 问题修复

API

- 修复 paddle.io.DataLoader 在 return_list=True 时因多线程冲突小概率报错问题。 (#43691)
- 修复 paddle.nn.Layer 的参数存在 None 类型参数时 to 方法报 NoneType 不存在 device 属性的错误。 (#43597)
- 修复 cumsum op 在某些 shape 下计算结果出错的问题。 (#42500, #43777)
- 修复静态图下 Tensor.__getitem__ 在使用 bool 索引时组网阶段输出结果维度为 0 的问题。 (#43246)
- 修复 paddle.slice 和 paddle.strided_slice 处理参数为负数时出现异常的问题。 (#43432)
- 修复 set_value op 在处理切片 step 为负数时赋值结果异常的问题。 (#43694)
- 修复 C++ 端 copy 接口不能在多卡设备间拷贝的问题。 (#43728)
- 修改 paddle.incubate.nn.functional.fused_attention 和 paddle.incubate.nn.functional.fused_feedforward 中属性命名引发的推理时的问题。 (#43505)
- 修复 ConditionalBlockGrad op 处理不需要 grad 的 Tensor 时异常的问题。 (#43034)
- 解决 C++ 的 einsum op 反向速度优化引起的显存增加问题，并将反向优化默认打开。 (#43397)
- 修复单卡下 paddle.io.DataLoader 多进程数据读取在固定随机种子时数据无法固定的问题。 (#43702)
- 修复 softmax op 在 Tensor 元素超过 2G 时，触发 CUDNN_STATUS_NOT_SUPPORT 的错误。 (#43719)
- 修复 trace op Event 字符串在不同算子无区分，导致性能分析不便利的问题。 (#42789)

其他

- 修复动转静多次 deepcopy 并保存导致的显存溢出问题。 (#43141)
- 修复自定义算子中使用的 PlaceType 类型升级引入的 device id 在多卡场景中出错的问题。 (#43830)
- 优化 paddle.profiler.Profiler timeline 可视化逻辑，将在 python 脚本中自定义的事件从 C++ 折叠层显示移动至 python 折叠层显示。 (#42790)

7.3 3. 部署方向 (Paddle Inference)

7.3.1 (1) 新增特性

新增功能

- CPU 上 ONNX Runtime 后端新增 PaddleSlim 量化模型支持。(#43774, #43796)

7.3.2 (2) 底层优化

CPU 性能优化

- EnableMkldnn 配置中移除 gpu_cpu_reshape2_matmul_fuse_pass，修复 ResNet50 性能下降的问题。(#43750)

GPU 性能优化

- 添加 bilinear_interp_v2 TensorRT convert 支持。(#43618)
- 添加 matmul_scale_fuse_pass、multihead_matmul_fuse_pass_v3 到 GPU pass，并添加单测。(#43765)
- 添加 GPU handle 延迟初始化支持。(#43661)

7.3.3 (3) 问题修复

框架及 API 修复

- 修复联编 Paddle-Lite XPU 时的编译报错问题。(#43178)
- 修复 ERNIE 3.0 pass 误触发的问题。(#43948)
- 修复 multihead op 中 int8 量化属性读不到的问题。(#43020)

后端能力修复

- 修复 MKLDNN 中 elementwise_mul 和 matmul 两个 op 在运行量化推理过程中崩溃的问题。(#43725)
- 修复同一模型在推理时 TensorRT 子图序列化文件反复生成的问题。(#42945, #42633)
- 修复 ONNX Runtime 后端与外部使用的 protobuf 冲突问题。(#43159, #43742)
- 修复 python 预测库 ONNX Runtime 后端在多输入情况下推理报错问题。(#43621)

7.4 4. 环境适配

7.4.1 编译安装

- 完成对 CUDA 11.6 的验证和适配，并在官网发布 CUDA 11.6 的安装包。(#43935, #44005)
- 修复在 Windows 上使用 CUDA 11.6 编译时的 cub 报错问题。(#43935, #44005)
- 修复 elementwise、reduce op 编译时间较长的问题。(#43202, #42779, #43205)

7.4.2 新硬件适配

- 寒武纪 MLU 支持飞桨 Profiler。(#42115)
- GraphCore IPU 支持显示编译进度。(#42078)

Chapter 8

2.3.0 Release Note

8.1 1. 重要更新

我们很高兴地发布飞桨框架 2.3.0 版本，本版本包含如下重要更新。

8.1.1 API

- 新增 100 多个 API，覆盖自动微分、线性代数、概率分布、稀疏张量、框架性能分析、硬件设备管理、视觉领域等方面。
- 新增 4 个自动微分 API，11 个线性代数 API，21 个概率分布类 API，更好地支持科学计算、强化学习等场景。
- 新增 11 个稀疏张量计算 API，支持创建 COO、CSR 格式的 Sparse Tensor 以及与 Tensor 互相转换等基础功能。
- 新增 9 个框架性能分析 API，以 `paddle.profiler.Profiler` 为核心，提供对训练、推理过程中性能数据的收集、导出和统计的功能。
- 新增 7 个硬件设备管理 API，更好支持硬件相关信息获取。
- 新增多个视觉、文本领域 API，方便复用 MobileNetV3, ResNeXt 等骨干网络，实现快速组网。

8.1.2 飞桨高可复用算子库 PHI

- 发布飞桨高可复用算子库 PHI (Paddle HIgh reusability operator library)，支持组合式算子功能复用。Primitive 算子内核复用、插件式硬件加速库复用。针对飞桨框架原算子库存在的算子接口不清晰、算子复用成本较高、调用性能不够快的问题，我们重构了飞桨框架的算子库，设计了灵活、高效的函数式算子库 Phi，通过对函数式算子接口组合调用的方式实现新算子。新算子库提供了 200 余个跟 python 开发接口保持一致的 C++ 运算类 API，以及近 500 个可供组合调用的前、反向函数式算子内核 Kernel，可大幅降低框架原生算子和自定义算子的开发成本。新算子库支持 Primitive API 方式开发算子内核，可

支持不同硬件（比如 GPU 和 XPU）的算子内核复用。新算子库支持以插件方式接入硬件（比如 NPU）的加速库，实现低成本复用硬件加速库。

8.1.3 分布式训练

- 全面升级自适应分布式训练架构，含弹性扩缩容、异步流水执行器、异构通信、自动并行等多个模块，支持了多种异构硬件下自动感知的分布式训练及分布式推理。
- 动态图混合并行下新增 MoE 并行策略、GroupSharded 并行策略、Pure FP16 等，进一步支持了动态图下大模型的高效并行训练。
- 全面升级优化了通用异构参数服务器架构，进行各模块的抽象简化，如通信、存储等，提升了参数服务器的二次开发体验；GPU 参数服务器在千亿参数百亿数据分钟级流式训练下性能提升 2.38 倍。

8.1.4 编译安装

- 从 2.3.0 版本开始，飞桨对框架支持的 GPU 架构种类进行了调整和升级。

8.1.5 推理部署

- 新增 Java API 和 ONNX Runtime CPU 后端。
- 支持 TensorRT 8.0 / 8.2 和结构化稀疏，针对 ERNIE 类结构模型性能深度优化。

8.1.6 硬件适配

- 新增自定义新硬件接入：提供一种插件式扩展 PaddlePaddle 硬件后端的方式。
- 新增对华为昇腾 910 / GraphCore IPU / 寒武纪 MLU / 昆仑芯 2 代多种异构芯片的训练/推理支持。

8.1.7 框架架构

- 这个版本中，我们在框架的执行器也做了大量工作，详情请见：[新动态图执行机制与全新静态图执行器](#)。

8.2 2. 不兼容升级

- 预编译安装包中移除 CUDA sm35 ARCH: 受到包体积大小的影响，在预编译的安装包中移除了 CUDA sm35 架构。(#41754)
- paddle.to_tensor 将一个 python int scalar 转换为 Tensor 时，在 Windows 上的默认数据类型由 int32 变为 int64，从而与 Linux/Mac 保持对齐。(#39662)
- 为了与 python3 下的除法行为保持一致，除法符号 / 从 rounding divide 变成 true divide，计算输出结果的数据类型从 int 切换成 float。(#40890)

```
>>> import paddle
>>> a = paddle.to_tensor([327])
>>> b = paddle.to_tensor([80])
>>> a / b
Tensor(shape=[1], dtype=int64, place=CUDAPlace(0), stop_gradient=True,
[4])
```

```
>>> import paddle
>>> a = paddle.to_tensor([327])
>>> b = paddle.to_tensor([80])
>>> a / b
Tensor(shape=[1], dtype=float32, place=Place(gpu:0), stop_gradient=True,
[4.08750010])
```

- 修正 ELU 的公式， $\alpha < 0$ 时的计算方式与原论文对齐，从而修复小部分情况下的计算结果错误。同时，由于在 $\alpha < 0$ 无法在数学上仅从输出计算反向梯度，因此 elu_ 在 $\alpha < 0$ 时将报错。(#37316)

```
# elu(x) = max(0, x) + min(0, a * (e^x - 1))
>>> import paddle
>>> x = paddle.to_tensor([-1., 6.])
>>> m = paddle.nn.ELU(-0.2)
>>> out = m(x)
>>> out
Tensor(shape=[2], dtype=float32, place=CUDAPlace(0), stop_gradient=True,
[ 0.          , -74.48576355])
>>> out = paddle.nn.functional.elu_(x, alpha=-0.2, name=None)
>>> out
Tensor(shape=[2], dtype=float32, place=CUDAPlace(0), stop_gradient=True,
[ 0.          , -74.48576355])
```

```
# elu(x) = x, if x > 0
# elu(x) = a * (e^x - 1), if x <= 0
>>> import paddle
```

(下页继续)

(续上页)

```

>>> x = paddle.to_tensor([-1., 6.])
>>> m = paddle.nn.ELU(-0.2)
>>> out = m(x)
>>> out
Tensor(shape=[2], dtype=float32, place=CUDAPlace(0), stop_gradient=True,
       [0.12642412, 6.])
>>> out = paddle.nn.functional.elu_(x, alpha=-0.2, name=None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.7/dist-packages/decorator.py", line 232, in fun
    return caller(func, *(extras + args), **kw)
  File "/usr/local/lib/python3.7/dist-packages/paddle/fluid/wrapped_decorator.py", line 25, in __impl__
    return wrapped_func(*args, **kwargs)
  File "/usr/local/lib/python3.7/dist-packages/paddle/fluid/dygraph/inplace_utils.py", line 34, in __impl__
    return func(*args, **kwargs)
  File "/usr/local/lib/python3.7/dist-packages/paddle/nn/functional/activation.py", line 89, in elu_
    assert alpha >= 0., "elu_ only support alpha >= 0, please use elu instead."
AssertionError: elu_ only support alpha >= 0, please use elu instead.

```

8.3 3. 训练框架 (含分布式)

8.3.1 (1) 新功能

API

- 新增 4 个自动微分类 API，支持科学计算需求，具体列表如下：(#40692)
 - paddle.incubate.autograd.vjp，计算向量-雅可比矩阵乘积。
 - paddle.incubate.autograd.jvp，计算雅可比矩阵-向量乘积。
 - paddle.incubate.autograd.Jacobian，计算雅可比矩阵。
 - paddle.incubate.autograd.Hessian，计算海森矩阵。
- 新增线性代数类 API
 - 新增 paddle.linalg.triangular_solve，计算具有唯一解的三角系数线性方程组。(#36714)
 - 新增 paddle.linalg.eig，计算一般方阵的特征分解。(#35764)
 - 新增 paddle.linalg.sovle，计算线性方程组的解。(#35715)

- 新增 paddle.linalg.lstsq, 计算线性方程组的最小二乘解。(#38585, #38621)
 - 新增 paddle.linalg.qr, 计算矩阵的 QR 分解。(#35742, #38824)
 - 新增 paddle.inner, 计算矩阵内积。(#37706)
 - 新增 paddle.outer, 计算矩阵外积。(#37706)
 - 新增 paddle.linalg.cov, 计算向量间协方差。(#38392)
 - 新增 paddle.linalg.cholesky_sovle, 计算方程 cholesky 解。(#38167)
 - 新增 paddle.linalg.lu、paddle.linalg.lu_unpack, 计算矩阵 lu 分解、解压缩 lu 矩阵。 (#38617, #38559, #38616)
- 新增 21 个概率分布类 API, 包括 6 个随机变量分布, 13 个随机变量变换, 2 个 KL 散度计算, 用于强化学习、变分推断、科学计算等场景, 具体列表如下: (#40536, #38820, #38558, #38445, #38244, #38047)
 - paddle.distribution.ExponentialFamily, 指数分布族基类。
 - paddle.distribution.Beta, Beta 分布。
 - paddle.distribution.Dirichlet, Dirichlet 分布。
 - paddle.distribution.Independent, 独立分布, 用于创建高阶分布。
 - paddle.distribution.TransformedDistribution, 变换分布, 用于通过基础分布及一系列变换生成高阶分布。
 - paddle.distribution.Multionomial, 多项分布。
 - paddle.distribution.Transform, 随机变量变换的基类。
 - paddle.distribution.AbsTransform, 取绝对值变换。
 - paddle.distribution.AffineTransform, 仿射变换。
 - paddle.distribution.ChainTransform, 变换的链式组合。
 - paddle.distribution.ExpTransform, 指数变换。
 - paddle.distribution.IndependentTransform, 独立变换, 用于扩展变换定义域的 event_dim。
 - paddle.distribution.PowerTransform, 幂变换。
 - paddle.distribution.ReshapeTransform, reshape 变换。
 - paddle.distribution.SigmoidTransform, sigmoid 变换。
 - paddle.distribution.SoftmaxTransform, softmax 变换。
 - paddle.distribution.StackTransform, stack 变换, 用于以 stack 方式组合多个变换。
 - paddle.distribution.StickBreakingTransform, stickbreaking 变换。
 - paddle.distribution.TanhTransform, tanh 变换。

- `paddle.distribution.kl_divergence`, 计算 KL 散度。
 - `paddle.distribution.register_kl`, 注册用户自定义 KL 散度计算函数。
- 新增高层 API
 - 新增 `paddle.vision.models.AlexNet`、`paddle.vision.models.alexnet`, 支持直接使用 AlexNet 模型。(#36058)
 - 新增 `paddle.vision.models.DenseNet`、`paddle.vision.models.densenet121`、`paddle.vision.models.densenet161`、`paddle.vision.models.densenet169`、`paddle.vision.models.densenet201`、`paddle.vision.models.densenet264`, 支持直接使用 DenseNet 模型。(#36069)
 - 新增 `paddle.vision.models.GoogLeNet`、`paddle.vision.models.googlenet`, 支持直接使用 GoogLeNet 模型。(#36034)
 - 新增 `paddle.vision.models.InceptionV3`、`paddle.vision.models.inception_v3`, 支持直接使用 InceptionV3 模型。(#36064)
 - 新增 `paddle.vision.models.MobileNetV3Small`、`paddle.vision.models.MobileNetV3Large`、`paddle.vision.models.mobilenet_v3_small`、`paddle.vision.models.mobilenet_v3_large`, 支持直接使用 MobileNetV3 模型。(#38653)
 - 新增 `paddle.vision.models.resnext50_32x4d`、`paddle.vision.models.resnext50_64x4d`、`paddle.vision.models.resnext101_32x4d`、`paddle.vision.models.resnext101_64x4d`、`paddle.vision.models.resnext152_32x4d`、`paddle.vision.models.resnext152_64x4d`, 支持直接使用 ResNeXt 模型。(#36070)
 - 新增 `paddle.vision.models.ShuffleNetV2`、`paddle.vision.models.shufflenet_v2_x0_25`、`paddle.vision.models.shufflenet_v2_x0_33`、`paddle.vision.models.shufflenet_v2_x0_5`、`paddle.vision.models.shufflenet_v2_x1_0`、`paddle.vision.models.shufflenet_v2_x1_5`、`paddle.vision.models.shufflenet_v2_x2_0`、`paddle.vision.models.shufflenet_v2_swish`, 支持直接使用 ShuffleNetV2 模型。(#36067)
 - 新增 `paddle.vision.models.SqueezeNet`、`paddle.vision.models.squeezeenet1_0`、`paddle.vision.models.squeezeenet1_1`, 支持直接使用 SqueezeNet 模型。(#36066)
 - 新增 `paddle.vision.models.wide_resnet50_2`、`paddle.vision.models.wide_resnet101_2`, 支持直接使用 WideResNet 模型。(#36952)
 - 新增 `paddle.vision.ops.nms` API, 支持单类别和多类别非极大抑制 (non-maximum suppression, nms) 算法, 用于目标检测预测任务加速。(#40962)
 - 新增 `paddle.vision.ops.roi_pool` 和 `paddle.vision.ops.RoIPool`, 支持检测任务中 ROI 区域池化操作。(#36154)
 - 新增 `paddle.vision.ops.roi_align` 和 `paddle.vision.ops.RoIAlign`, 支持检测任务中 ROI Align 操作。(#35102)

- 新增 paddle.text.ViterbiDecoder、paddle.text.viterbi_decode Viterbi 解码 API，主要用于序列标注模型的预测。(#35778)
- 新增 11 个 Sparse 类 API，支持创建 COO、CSR 格式的 Sparse Tensor，与 Tensor 互相转换等基础功能：
 - paddle.sparse.sparse_coo_tensor，创建 COO 格式的 Sparse Tensor。(#40780)
 - paddle.sparse.sparse_csr_tensor，创建 CSR 格式的 Sparse Tensor。(#40780)
 - paddle.sparse.ReLU，支持 SparseCooTensor 的 ReLU 激活层。(#40959)
 - paddle.sparse.functional.relu，支持 SparseCooTensor 的 ReLU 函数。(#40959)
 - Tensor.values()，获取 SparseCooTensor 或者 SparseCsrTensor 的非零元素方法。(#40608)
 - Tensor.indices()，获取 SparseCooTensor 的坐标信息的方法。(#40608)
 - Tensor.crows()，获取 SparseCsrTensor 的压缩行信息的方法。(#40608)
 - Tensor.cols()，获取 SparseCsrTensor 的列信息的方法。(#40608)
 - Tensor.to_sparse_coo()，将 DenseTensor 或者 SparseCsrTensor 转换为 SparseCooTensor。(#40780)
 - Tensor.to_sparse_csr()，将 DenseTensor 或者 SparseCooTensor 转换为 SparseCsrTensor。(#40780)
 - Tensor.to_dense()，将 SparseCooTensor 或者 SparseCsrTensor 转换为 DenseTensor。(#40780)
- 新增硬件相关 API
 - 新增 paddle.device.cuda.max_memory_allocated、paddle.device.cuda.max_memory_reserved、paddle.device.cuda.memory_allocated 和 paddle.device.cuda.memory_reserved 四个 GPU 显存监测相关 API，方便实时查看和分析模型显存占用指标。(#38657)
 - 新增 paddle.device.cuda.get_device_properties，支持返回 CUDA 设备属性信息。(#35661)
 - 新增 paddle.device.cuda.get_device_name 和 paddle.device.cuda.get_device_capability，支持返回 GPU 设备名称信息和计算能力的主要和次要修订号。(#35672)
- 新增 Tensor 操作 API
 - 新增 paddle.nansum，沿 axis 对输入 Tensor 求和，且忽略掉 NaNs 值。(#38137)
 - 新增 paddle.nanmean，沿 axis 对输入 Tensor 求平均，且忽略掉 NaNs 值。(#40472)
 - 新增 paddle.clone，返回输入 Tensor 的拷贝，并且提供梯度计算。(#38020)
 - 新增 paddle.Tensor.element_size，返回 Tensor 中的单个元素在计算机中所分配的 bytes 数量。(#38020)

- 新增 `paddle.Tensor.to_uva_tensor`, 支持将 `numpy` 对象转换为实际存储在 CPU, 但可作为 CUDA 对象进行虚拟地址访问的功能。(#39146, #38950)
- 新增 `paddle.rot90`, 沿 `axes` 指定的平面将 n 维 Tensor 旋转 90 度。(#37634)
- 新增 `paddle.logit` 和 `paddle.Tensor.logit`, 计算输入 Tensor 的 logit 函数值。(#37844)
- 新增 `paddle.repeat_interleave`, 沿着指定轴对输入进行复制, 创建并返回到一个新的 Tensor。(#37981)
- 新增 `paddle.renorm`, 把 Tensor 在指定的 `axis` 切分成多块后分别进行 p norm 操作。(#38130, #38459)
- 新增 `paddle.mode` 和 `paddle.Tensor.mode`, 沿指定轴查找输入 Tensor 的众数及对应的索引。 (#38446)
- 新增 `paddle.quantile` 和 `paddle.Tensor.quantile`, 沿指定轴计算 Tensor 的 q 分位数。 (#38567)
- 新增 `paddle.kthvalue` 和 `paddle.Tensor.kthvalue`, 查找 Tensor 中指定轴上第 k 小的数 及对应的索引。(#38386)
- 新增 `paddle.is_floating_point` 和 `paddle.Tensor.is_floating_point`, 判断输入 Tensor 是否为浮点类型。(#37885)
- 新增 `paddle.erfinv` 和 `paddle.Tensor.erfinv`, 计算输入 Tensor 的逆误差函数。(#38295)
- 新增 `paddle.lerp` 和 `paddle.Tensor.lerp`, 根据给定权重计算输入 Tensor 间的线性插值。 (#37253)
- 新增 `paddle.angle`, 用于计算复数 Tensor 的相位角。(#37689)
- 新增 `paddle.rad2deg` 和 `paddle.Tensor.rad2deg`, 将元素从弧度的角度转换为度。 (#37598)
- 新增 `paddle.deg2rad` 和 `paddle.Tensor.deg2rad`, 将元素从度的角度转换为弧度。 (#37598)
- 新增 `paddle.gcd` 和 `paddle.Tensor.gcd`, 计算两个输入的按元素绝对值的最大公约数。 (#37819)
- 新增 `paddle.lcm` 和 `paddle.Tensor.lcm`, 计算两个输入的按元素绝对值的最小公倍数。 (#37819)
- 新增 `paddleamax` 和 `paddle.Tensoramax`, 对指定维度上的 Tensor 元素求最大值, 正向 结果和 `max` 一样, 有多个相等的最大值时, 反向的梯度平均分到这多个值的位置上。 (#38417)
- 新增 `paddleamin` 和 `paddle.Tensoramin`, 对指定维度上的 Tensor 元素求最小值, 正向 结果和 `min` 一样, 有多个相等的最小值时, 反向的梯度平均分到这多个值的位置上。 (#38417)
- 新增 `paddle.isclose`, 用于判断两个 Tensor 的每个元素是否接近。 (#37135)

- 新增 paddle.put_along_axis 和 paddle.take_along_axis，用于提取或放置指定索引下标的元素。(#38608)
 - 新增 paddle.bincount 和 paddle.Tensor.bincount，用于统计 Tensor 中每个元素出现的次数。(#36317)
 - 新增 paddle.fmax、paddle.fmin，扩展了 max/min 的功能，支持比较的两个 Tensor 中有 NaN 值的情况，即如果对应位置上有 1 个 NaN 值，则返回那个非 NaN 值；如果对应位置上有 2 个 NaN 值，则返回 NaN 值。(#37826)
 - 新增 paddle.diff，用于计算沿给定维度的第 n 个前向差值，目前支持 n=1。(#37441)
 - 新增 paddle.asinh、paddle.acosh、paddle.atanh 反双曲函数类 API。(#37076)
 - 新增 paddle.as_real, paddle.as_complex 用于实数 Tensor 和复数 Tensor 之间的转换。 (#37784)
 - 新增 paddle.complex 用于给定实部和虚部构造复数 Tensor。(#37918, #38272)
 - 新增 paddle.det 与 paddle.slogdet，用于计算矩阵的行列式和行列式的自然对数。(#34992)
 - 新增 paddle.nn.utils.parameters_to_vector，可以将输入的多个 parameter 展平并连接为 1 个 1-D Tensor。(#38020)
 - 新增 paddle.nn.utils.vector_to_parameters，将 1 个 1-D Tensor 按顺序切分给输入的多个 parameter。(#38020)
- 新增组网类 API
 - 新增 paddle.nn.Fold、paddle.nn.functional.fold，支持将提取出的滑动局部区域块还原成 batch 的 Tensor。(#38613)
 - 新增 paddle.nn.CELU、paddle.nn.functional.celu，支持 CELU 激活层。(#36088)
 - 新增 paddle.nn.HingeEmbeddingLoss，增加计算 hinge embedding 损失的方式，通常用于学习 nonlinear embedding 或半监督学习。(#37540)
 - 新增 paddle.nn.ZeroPad2D API，按照 padding 属性对输入进行零填充。(#37151)
 - 新增 paddle.nn.MaxUnPool3D 和 paddle.nn.MaxUnPool1D，用于计算 3D 最大反池化和 1D 最大反池化。(#38716)
 - 新增 paddle.incubate.graph_khop_sampler、paddle.incubate.graph_sample_neighbors、paddle.incubate.graph_reindex API，支持图多阶邻居采样和图编号重索引操作，主要用于图神经网络模型训练。(#39146, #40809)
 - 新增随机数类 API
 - 新增 paddle.poisson，以输入 Tensor 为泊松分布的 lambda 参数，生成一个泊松分布的随机数 Tensor。(#38117)
 - 新增 paddle.randint_like API，支持新建服从均匀分布的、范围在 [low, high) 的随机 Tensor，输出的形状与输入的形状一致。(#36169)

- 新增 `paddle.Tensor.exponential_`, 为 `inplace` 式 API, 通过指数分布随机数来填充输入 `Tensor`。 (#38256)
- 新增参数初始化类 API
 - 新增 `paddle.nn.initializer.Dirac`, 通过迪拉克 `delta` 函数来初始化 3D/4D/5D 参数, 其常用于卷积层 `Conv1D/Conv2D/Conv3D` 的参数初始化。 (#37389)
 - 新增 `paddle.nn.initializer.Orthogonal`, 正交矩阵初始化, 被初始化后的参数是(半)正交向量。 (#37163)
 - 新增 `paddle.nn.initializer.calculate_gain`, 获取激活函数的推荐增益值, 增益值可用于设置某些初始化 API, 以调整初始化范围。 (#37163)
- 新增学习率类 API
 - 新增 `paddle.optimizer.lr.MultiplicativeDecay`, 提供 `lambda` 函数设置学习率的策略。 (#38250)
- 新增分布式相关 API
 - 新增 `paddle.incubate.optimizer.DistributedFusedLamb`, 使得 Lamb 优化器可分布式更新参数。 (#40011, #39972, #39900, #39747, #39148, #39416)
- 新增优化器相关 API(#40710)
 - `paddle.incubate.optimizer.functional.minimize_bfgs`, 增加二阶优化器 BFGS。
 - `paddle.incubate.optimizer.functional.minimize_lbfgs`, 增加二阶优化器 L-BFGS。
- 新增 `paddle.incubate.multiprocessing` 模块, 支持 `Tensor` (CPU/GPU) 在 python 进程间传输。 (#37302, #41339)
- 新增 `paddle.incubate.autotune.set_config` API, 支持多版本 Kernel 自动选择、混合精度数据布局自动转换、`DataLoader` 的 `num_workers` 自动选择, 以自动提升模型性能。 (#42301)
- 新增 `paddle.incubate.nn.FusedMultiTransformer` 和 `paddle.incubate.nn.functional.fused_multi_transformer` API, 可将多层 `transformer` 融合到一个 `op` 中, 提升模型推理性能, 注意: 仅支持前向推理。 (#42311)
- 新增动静统一的 `einsum_v2` op, 兼容原有 python 端 `paddle.einsum` 实现的同时支持动转静导出和更加完备的 Infershape 推导。 (#42495, #42327, #42397, #42105)

IR(Intermediate Representation)

- 动态图转静态图
 - 变量类型 StaticAnalysis 模块新增支持类似 `a, b = paddle.shape(x)` 的类型标记。(#39245)
 - 新增支持 `InputSpec.name` 作为 Program 缓存 hash key 的计算字段。(#38273)
 - 新增支持 `dict['key'] = x.shape` 语法。(#40611)
 - 新增支持 Pure FP16 训练。(#36944)
 - 新增支持 `for i in [x, y, z]` 语法。(#37259)
 - 新增支持 python3 的 type hint 语法。(#36544)
- Pass 开发
 - 新增基于 NVIDIA cuBlasLt Epilogue 的 FC + [relu|gelu] 的前向与反向融合。(#39437)
- Kernel Primitive API
 - 新增 GPU 平台 KP 算子, 包括 `cast`、`scale`、`clip`、`bce_loss`、`abs_grad`、`reduce_sum_grad`、`reduce_mean_grad`、`clip`、`bce_loss`、`full`、`full_like`、`distribution`、`random`、`masked_select_kernel`、`where_index`、`masked_select_grad`、`dropout`、`sigmoid`、`where`、`abs_grad`。(#36203, #36423, #39390, #39734, #38500, #38959, #39197, #39563, #39666, #40517, #40617, #40766, #39898, #39609)
 - 新增支持 XPU2 源码编译模式。(#37254, #40397, #38455)
 - 新增支持 KP 算子在 XPU2 和 GPU 中复用, 包括 `reduce`、`broadcast`、`elementwise_add`、`exp`、`log`、`relu`、`sigmoid`、`leaky_relu`、`softplus`、`hard_swish`、`reciprocal`。(#36904, #37226, #38918, #40560, #39787, #39917, #40002, #40364)
 - 新增 XPU2 平台 KP 算子单测, 包括 `brelu`、`ceil`、`celu`、`elu`、`floor`、`hard_shrink`、`hard_sigmoid`、`log1p`、`logsigmoid`、`relu6`、`silu`、`soft_relu`、`softsign`、`sqrt`、`square`、`swish`、`thresholded_relu`、`softshrink`。(#40448, #40524)
 - 新增 XPU2 KP 模型支持, 包括 resnet50、deepfm、wide_deep、yolov3-darknet53、det_mv3_db、bert、transformer、mobilenet_v3、GPT2。

混合精度训练

- 从混合精度训练 `paddle.amp.GradScaler` 的 `minimize` 中拆分出 `paddle.amp.GradScaler.unscale_` 方法, 提供恢复 loss 的独立接口。(#35825)
- 为 `paddle.nn.ClipByGlobalNorm` 动态图模式添加 FP16 支持, 为 clip op 添加 FP16 Kernel, 使 clip 相关操作支持 FP16。(#36198, #36577)
- 支持 `paddle.amp.decorate` 传入的 `optimizer` 参数为 `None`。(#37541)
- 为 `merged_momentum` op 添加支持输入多学习率、支持 `use_nesterov` 策略的计算、支持 `regularization` 计算。(#37527)

- 为 `paddle.optimizer.Momentum` 优化器添加 `multi_tensor` 策略、为 `Optimizer` 类的 `clear_grad` 添加 `set_to_zero` 分支。(#37564)
- 为 `paddle.optimizer.Adam` 优化器添加 `multi_tensor` 策略。(#38010)
- 为 `paddle.optimizer.SGD` 优化器添加 `multi_precision` 策略。(#38231)
- 为优化器 `state_dict` 方法添加存储 `master weight` 参数。(#39121)
- 添加支持 op CUDA bfloat16 混合精度训练，支持 O1、O2 模式，通过 `paddle.amp.auto_cast` 可开启上述训练模式。(#39029, #39815)
- 为如下 ops 添加 bfloat16 CUDA Kernel: matmul、concat、split、dropout、reshape、slice、squeeze、stack、transpose、unbind、elementwize_max、elementwize_add、elementwize_mul、elementwize_sub、scale、sum、layer_norm、p_norm、reduce_sum、softmax、log_softmax、sigmoid、sqrt、softplus、square、gaussian_random、fill_constant、fill_any_like。(#39485, #39380, #39395, #39402, #39457, #39461, #39602, #39716, #39683, #39843, #39999, #40004, #40027)
- 为如下 ops 添加 bfloat16 CPU Kernel: dropout、reshape、slice、squeeze、unsqueeze、stack、transpose、unbind、elementwize_max、elementwise_mul、elementwise_sub、gather。(#39380, #39395, #39402, #39457, #39461, #39602, #39716, #39683)
- 支持打印 bfloat16 类型的 Tensor。(#39375, #39370)
- 为 `p_norm`、`elementwise_max`、`fill_constant_batch_size_like`scatter` 增加 FP16 计算支持。(#35888, #39907, #38136, #38499)
- 为如下 ops 增加 `int16_t` 支持: cumsum、less_than、less_equal、greater_than、greater_equal、equal、not_equal、fill_any_like、gather_nd、reduce_sum、where_index、reshape、unsqueeze。(#39636)
- 为 `cross_entropy` op 增加 `int16_t` label 类型的支持。(#39409)
- 为 `embedding` op 增加 `int16_t` id 类型的支持。(#39381)
- 为 `reduce_mean` op 增加 FP16 类型的支持。(#38289)
- 为 `elementwise_min` op 增加 FP16 类型的支持。(#38123)
- 更新 bfloat16 AMP oneDNN 默认支持列表。(#39304)

飞桨高可复用算子库 PHI

针对飞桨框架原算子库存在的算子接口不清晰、算子复用成本较高、调用性能不够快的问题，我们重构了飞桨框架的算子库，设计了灵活、高效的函数式算子库 **PHI**，通过对函数式算子接口组合调用的方式实现新算子。新算子库提供了 200 余个跟 `python` 开发接口保持一致的 `C++` 运算类 `API`，以及近 500 个可供组合调用的前、反向函数式算子内核 `Kernel`，可大幅降低框架原生算子和自定义算子的开发成本。新算子库支持 `Primitive API` 方式开发算子内核，可支持不同硬件（比如 `GPU` 和 `XPU`）的算子内核复用。新算子库支持以插件方式接入硬件（比如 `NPU`）的加速库，实现低成本复用硬件加速库。主要可分为以下几部分工作：

- **算子库基础架构、核心组件与机制实现：**合理规划新算子库的目录结构，设计实现了新算子库的公共基础数据结构、新的函数式 `InferMeta` 和 `Kernel` 开发范式以及相应的注册和管理组件，并且支持 `Kernel` 文

件的自动化编译对象生成及编译依赖关系生成，使开发者仅需关注函数式 Kernel 的实现，开发范式简洁清晰。（#34425, #37107, #36946, #36948, #37876, #37916, #37977, 38078, #38861, #39123, #39131, #39748, #39790, #39941, #40239, #40635, #41091, #37409, #37942, #39002, #38109, #37881, #37517, #39870, #40975, #39475, #37304, #36910, #37120, #37146, #37215, #37255, #37369, #38258, #38257, #38355, #38853, #38937, #38977, #38946, #39085, #39153, #39228, #38301, #38275, #38506, #38607, #38473, #38632, #38811, #38880, #38996, #38914, #39101）

- 算子库 C++ API 体系建设：设计实现了基于 yaml 配置文件的算子定义范式、自动生成了 200 余个 C++ 运算类 API，供内外部开发者复用，降低了基础运算的重复开发成本。（#37668, #36938, #38172, #38182, #38311, #38438, #39057, #39229, #39281, #39263, #39408, #39436, #39482, #39497, #39651, #39521, #39760, #40060, #40196, #40218, #40640, #40732, #40729, #40840, #40867, #41025, #41368）
- 算子库兼容各执行体系：实现新的 InferMeta 及 Kernel 接入原动静态图执行体系、支持原 OpKernel 注册安全移除并迁移为新的 Kernel 形式。（#34425, #38825, #38837, #38842, #38976, #39134, #39140, #39135, #39252, #39222, #39351）
- 算子库底层数据结构及工具函数与框架解耦：解除 Phi 在核心数据结构上对框架的依赖，为后续 Phi 独立编译奠定基础，支持 infrt、自定义 Kernel 等一系列基于 Phi 的建设工作（#38583, #39188, #39560, #39931, #39169, #38951, #38898, #38873, #38696, #38651, #39359, #39305, #39234, #39098, #39120, #38979, #38899, #38844, #39714, #39729, #39889, #39587, #39558, #39514, #39502, #39300, #39246, #39124）
- 自定义算子机制与 Phi 整合并完善：支持在自定义算子编写时调用 Phi 自动生成的 200 余个 C++ 运算类 API，降低自定义算子开发成本，并进行一系列问题修复。（#37122, #37276, #37281, #37262, #37415, #37423, #37583, #38776, #39353, #41072）
- 算子规模化迁移改写：迁移了约 250 个高频算子的前、反向算子内核 Kernel 至新算子库，改写为函数式，支持在 C++ 端通过调用多个基础 Kernel 函数封装，快速组合实现高性能算子；同时，添加相应的 yaml 算子定义，并接入新动态图执行体系，提升 python API 调度性能。迁移改写的算子包括：

- sqrt (#40727)
- square (#40727)
- sin (#40175)
- sinh (#40175)
- elementwise_fmax (#40140)
- elementwise_fmin (#40140)
- pool2d (#40208, #41053)
- max_pool2d_with_index (#40208, #41053)
- pool3d (#40208, #41053)
- max_pool3d_with_index (#40208, #41053)
- fill_constant (#36930, #39465)
- p_norm (#40819)

- fill_constant_batch_size_like (#40784)
- conv2d (#39354)
- conv2d_transpose (#40675, #41053)
- conv3d (#39354)
- conv3d_transpose (#40675, #41053)
- mish (#40727)
- gather_nd (#40090, #40043)
- gather (#40500)
- scatter (#40090, #40043)
- scatter_nd_add (#40090, #40043)
- sgd (40045)
- momentum (#41319)
- rmsprop (#40994)
- index_sample (#38130, #38459, #39905)
- adam (#40351)
- layer_norm (#40193)
- adagrad (#40994)
- adamax (#40173)
- adadelta (#40173)
- clip (#40602, #41661, #41675)
- ceil (#40913)
- cos (#40175)
- atan (#40175)
- cosh (#40175)
- erf (#40388)
- asin (#40175)
- acos (#40175)
- scale (#39278)
- elementwise_pow (#40993)
- elementwise_sub (#39225, #37260)

- round (#40913)
- floor (#40913)
- pow (#40913)
- elementwise_floordiv (#40993)
- reciprocal (#40727)
- log1p (#40785)
- allclose (#40469)
- mul (#40833)
- elementwise_max (#40590)
- elementwise_min (#40590)
- elementwise_mod (#40590)
- elementwise_add (#39048, #37043)
- matmul_v2 (#36844, #38713)
- elementwise_mul (#41042, #40252, #37471)
- elementwise_div (#40172, #40039, #37418)
- SelectedRows (#39037, #39087, #39128, #39162, #39236)
- fill_any_like (#39807)
- dot (#38359)
- sum (#40873)
- cumsum (#39976, #40200)
- diag_v2 (#39914)
- auc (#39976, #40200)
- log_loss (#39976, #40200)
- one_hot_v2 (39876)
- sigmoid_cross_entropy_with_logits (#39976, #40200)
- bce_loss (#39868)
- argsort (#40151)
- arg_max (#40222)
- arg_min (#40222)
- segment_pool (#40099)

- frobenius_norm (#40707, #41053)
- dist (#40178)
- isnan_v2 (#40076)
- logical_and (#39942)
- logical_not (#39942)
- isfinite_v2 (#40076)
- logical_or (#39942)
- isinf_v2 (#40076)
- is_empty (#39919)
- logical_xor (#39942)
- less_than (#39970)
- not_equal (#39970)
- equal (#39970)
- less_equal (#39970)
- equal_all (#39970)
- uniform_random (#39937)
- randint (#39876, #41375)
- randperm (#41265)
- unbind (#39789)
- bernoulli (#39590)
- increment (#39858, #39913)
- multinomial (#39858, #39913)
- addmm (#39858, #39913)
- cholesky (#39858, #39913)
- where (#39811)
- log10 (#40785)
- log2 (#40785)
- expm1 (#40727)
- atan2 (#39806)
- gaussian_random (#39932, #40122, #40191)

- empty (#38334)
- truncated_gaussian_random (#39971, #40191)
- mv (#39861, #39954)
- tan (#40175)
- set_value (#40195, #40478, #40636)
- bitwise_and (#40031)
- bitwise_not (#40031)
- bitwise_or (#40031)
- poisson (#39814)
- cholesky_solve (#40387)
- bitwise_xor (#40031)
- triangular_solve (#40417)
- sigmoid (#40626)
- atanh (#40175)
- softsign (#40727)
- thresholded_relu (#40385)
- tanh_shrink (#40565)
- stanh (#40727)
- reduce_mean (#37559)
- reduce_max (#40225)
- reduce_min (#40374)
- mean (#40872, #41319)
- reduce_all (#40374)
- reduce_any (#40374)
- logsumexp (#40790)
- softshrink (#40565)
- range (#41265, #40581)
- stack (#40581)
- tile (#40371)
- unique (#40581)

- unstack (#40581)
- slice (#40736)
- transpose2 (#39327)
- unsqueeze2 (#40596)
- squeeze2 (#40596)
- strided_slice (#40708)
- softmax (#39547)
- leaky_relu (#40385)
- gelu (#40393)
- prelu (#40393)
- log_softmax (#40393)
- elu (#40565)
- logsigmoid (#40626)
- psroi_pool (#40353, #41173)
- kthvalue (#40575)
- mode (#40571)
- yolo_box (#40112)
- yolov3_loss (#40944)
- temporal_shift (#40727)
- depthwise_conv2d (#39354)
- pad3d (#40701)
- pad (#40012)
- greater_equal (#39970)
- kldiv_loss (#39770)
- isclose (#39770)
- silu (#40565)
- unfold (#39778)
- batch_norm (39347)
- norm (#39324)
- roi_pool (#40574, #40682, #41173)

- roi_align (#40382, #40556, #41402)
- deformable_conv (#40700, #40794, #41644)
- deformable_conv_v1 (#40794, #41644)
- label_smooth (#39796)
- grid_sampler (#40585)
- greater_than (#39970)
- pixel_shuffle (#39949, #39712)
- nearest_interp_v2 (#40855)
- bilinear_interp_v2 (#40855)
- softmax_with_cross_entropy (#40832)
- rnn (#41007)
- reverse (#40791)
- trace (#39510)
- kron (#40427)
- accuracy (#39982)
- gather_tree (#40082, #39844)
- dropout (#40148)
- bincount (#39947)
- warpctc (#41389, #40023)
- multiplex (#40007, #40102)
- qr (#40007, #40007)
- assign_value (#40967)
- assign (#40022)
- cast (#37610)
- tril_triu (#40007, #41053)
- where_index (#40255)
- index_select (#40260, #41053)
- roll (#40257, #41053)
- cumprod (熊昆 #39770)
- shard_index (#40254)

- reshape2 (#40914, #39631, #38833, #37164)
- flip (#39822, #40974)
- eye (#39712, #40105, #41476)
- lookup_table_v2 (#39901)
- searchsorted (#40520, #41053)
- adamw (#40351)
- tanh (#40385)
- cross (#39829)
- concat (#38955, #41112)
- split (#39060)
- linspace (#40124)
- huber_loss (#39761)
- hierarchical_sigmoid (#40553)
- nll_loss (#39936)
- graph_send_recv (#40092, #40320)
- abs (#39492, #39762)
- exp (#40727)
- rsqrt (#40727)
- viterbi_decode (#40186)
- conj (#38247)
- real (#39777, #41173)
- imag (#39777, #41173)
- take_along_axis (#39959, #40270, #40974)
- put_along_axis (#39959, #40974)
- lgamma (#39770)
- relu (#40175)
- maxout (#39959, #40974)
- log (#40785)
- bilinear_tensor_product (#39903)
- flatten_contiguous_range (#38712, #36957, #41345)

- matrix_rank (#40074, #40519, #41466)
- logit (#37844)
- lerp (#40105, #39524)
- erfinv (#39949, #39712)
- broadcast_tensors (#40047)
- gumbel_softmax (#39873)
- diagonal (#39575)
- trunc (#39543, #39772)
- multi_dot (#40038)
- matrix_power (#40231)
- digamma (#39240)
- masked_select (#39193)
- determinant (#40539)
- eigh (#40213)
- size (#39949, #39712)
- shape (#40248)
- reduce_sum (#37559, #41295)
- reduce_prod (#39844)
- histogram (#39496)
- meshgrid (#41411)
- brelu (#40385)
- hard_swish (#40913)
- hard_shrink (#40565)
- selu (熊昆 #39819)
- expand_v2 (#39471)
- top_k_v2 (#40064)
- expand_as_v2 (#40373)
- swish (#40913)
- hard_sigmoid (#40626)

- exp, det, assign, gaussian_random, matrix_rank, eye, deformable_conv。([#41755]exp, det, assign, gaussian_random, matrix_rank, eye, deformable_conv。 (#41755, #41737

新动态图执行机制

针对飞桨原动态图执行机制的调度性能、二次开发能力差的问题，我们重构了动态图的底层执行机制。通过全新的调用执行方式，配合 Phi 算子库进行高效的运行时执行，对于 Phi 算子库支持的算子，切换到新动态图模式能体验到调度性能有较大幅度的提升。但是由于整体框架执行机制升级的工作量巨大，且该部分工作耦合了大量 Phi 算子库的工作，因此在这个版本下我们仍未默认使用该执行方式。如果想要试用可以通过设置环境变量 `FLAGS_enable_eager_mode=1` 来切换使用。具体包括如下内容：

- **新动态图执行机制基础架构、核心组件与机制实现：**静态化动态图相关执行代码，将原本的同质化的算子构建变成针对不同 Phi API 的特异化调用从而极大的优化了调度开销。(#36059, #37323, #37556, #37555, #37478, #37458, #37479, #37599, #37659, #37654, #39200, #39309, #39319, #39414, #39504, #39526, #39878, #39963)
- **新动态图执行机制子功能开发、适配：**支持了更加灵活，更加完备的动态图子功能例如 hook, pylayer, double_grad, inplace, amp 等等。(#41396, #40400, #40695, #41043, #40915, #41104, #41350, #41209, #40830, #40891, #36814, #37377, #37193, #36965, #37810, #36837, #38488, #39282, #39449, #39531, #39638, #39674, #39893, #40170, #40693, #40937, #41016, #41051, #41121, #41198, #41287, #41380, #41306, #41387, #40623, #40945, #39282, #39449, #38488)
- **新动态图执行的自动代码生成机制：**当我们为了将大量的同质化算子的计算和调度逻辑分化成不同的特异化的调度逻辑时，我们发现这是一个非常庞大的工作，因此我们引入了全新的自动代码生成逻辑来生成代码从而简化动态图的运行时逻辑。同时，为了能够适配之前框架中的各类运行时逻辑，我们也利用了一些复杂的编译手段来运行时的获取信息从而生成更加准确的调度代码。(#37574, #37575, #37639, #37723, #37753, #37812, #37837, #37910, #37943, #37992, #37959, #38017, #37969, #38160, #38085, #38562, #38573, #39192, #39215, #39355, #39358, #39328, #39233, #39628, #39767, #39743, #39897, #39797, #39997, #40058, #40080, #40107, #39962, #40132, #40276, #40266, #40480, #40482, #40368, #40650, #40815, #40907, #40935, #41089)
- **新动态图执行机制接入主框架，联合调试：**我们目前利用一些环境变量区分静态图模式和动态图模式（含新动态图和老动态图模式），这些模式下我们已经适配了大部分的动态图的逻辑，但是仍有大量问题正在修复中。(#37638, #37643, #37653, #38314, #38337, #38338, #39164, #39326, #40391, #40201, #40854, #40887)
- **更新了动态图下的一些判断逻辑，支持兼容形态下的动态图快速执行路径：**(#40786)
 - 非静态图模式（目前的过渡方案）：`_non_static_mode()`。
 - 在动态图模式下且判断在新动态图（推荐的判断逻辑）：`_in_dygraph_mode()`。
 - 在动态图模式下且判断在老动态图（不推荐的判断逻辑，在将来的版本中将废弃）：`_in_legacy_dygraph()`。
 - 在动态图模式下开启老动态图并关闭新动态图：`_enable_legacy_dygraph()` 或者退出 `_test_eager_guard()`。

- 在动态图模式下开启新动态图并关闭老动态图: `_disable_legacy_dygraph()` 或者 `with _test_eager_guard()`。
- 在静态图或者动态图模式下判断在新动态图: `_in_eager_without_dygraph_check()`。
- 动态图重构后支持 **inplace** 策略: 输入与输出为同一个 Tensor。
 - 为动态图重构中间态适配 **inplace** 策略。(#40400)
 - 为动态图重构最终态适配 **inplace** 策略。(#40695)
 - 动态图重构后, 为 PyLayer 功能添加 **inplace** 策略。(#41043)
 - 动态图重构后, 为 Tensor 的 `setitem` 功能添加 **inplace** 策略。(#40915)
 - 动态图重构后添加 `_reset_grad_inplace_version` 接口, 将 Tensor 的梯度的 **inplace version** 置为 0。(#41101)
 - 反向计算过程中如果不需要前向 Tensor 的值 (`no_need_buffer` 属性), 则不需要对该 Tensor 进行 **inplace version** 的检测操作。为 `no_need_buffer` 的 Tensor 跳过 **inplace version** 的检查。(#41350)
 - 统一动态图重构后与重构前对 **inplace version** 检查的报错信息。(#41209)
- 动态图重构后支持 **view** 策略: 输入与输出 Tensor 共享底层数据。
 - 为动态图重构中间态适配 **view** 机制。包括 `reshape`、`squeeze`、`unsqueeze`、`flatten` API。(#40830)
 - 为动态图重构最终态适配 **view** 机制。包括 `reshape` API。(#40891)
- 添加支持新动态图 **eager Tensor** 在 **python** 端的 **weakref**。(#41797)
- 增强新动态图 **DoubleGrad** 功能, 支持基础的 DoubleGrad 功能。(#41893, #41894, #41895)
- 新增 `core.eager.StringTensor` 接口, 支持在 python 端构造 StringTensor 以及使用 StringTensor 相关 API。(#41039)
- 为 `core.eager.Tensor` 新增 `*grad_name` 和 `_grad_value` API, 返回梯度的名称和值。(#41990)
- 为动态图中间态添加对 `no_need_buffer` 属性的处理。在 `inplace` 反向检查操作中, 会跳过具有 `no_need_buffer` 属性的 Tensor 的检查。(#41720)

全新静态图执行器

为了解决飞桨原静态图执行器在部分场景下调度性能不够理想, 不便于扩展多 stream 等问题, 我们实现了全新的性能优越, 易于扩展的静态图执行器, 充分利用了多 stream、多线程的异步调度能力。新执行器相当于原执行器是兼容升级, 目前已在单机单卡场景下默认使用, 用户不需要在训练代码中做任何修改即可自动使用。当然, 我们也提供了接口来切换回原执行器, 用户可以通过设置环境变量 `FLAGS_USE_STANDALONE_EXECUTOR=false` 来切换回原执行器。(#41179) 主要内容如下:

- 基础组件: 用于执行器中多线程算子调度的高性能线程池 (#35470, #35930, #36030, #36480, #36688, #36740, #38335, #40770) 及线程协同组件 (#38779, #40876, #40912), 算子执行后及时地显存回收 (#37642, #39617, #40859), 并行执行器新依赖分析算法 (#37231) 等。

- 调度逻辑：优化执行器中算子的调度方法，支持多 stream 的多线程异步调度机制，将数据类型、设备、布局等转换改为算子调度以提升性能，支持缓存算子 Kernel 选择，支持选择全新 Phi 算子等。（#35024, #34922, #35711, #35928, #39458, #36899）。
 - 接口兼容：兼容原执行器的用户接口和功能，如对齐 python 端 Executor.run()、支持 Scope 中管理 Tensor 等，确保用户可以无感知地切换新执行器。（#37278, #37379, #37445, #37510, #40955, #41778, #41058, #38584, #37957, #37672, #37474, #37085, #37061, #36945）
 - 增强多线程场景下调试和报错功能，将子线程的报错捕获到主线程中统一抛出，以提升用户体验。（#36692, #36802）
 - 修复新执行器通信流重置 Allocator 中 stream 缓存信息的问题，减少跨 stream 场景下的 RecordStream 开销，优化后 DeepFM 模型性能提升约 8%。（#42046）
 - 优化新执行器算子间的依赖分析方法，提升运行性能；为 send/recv 通信算子建立正确依赖以支持流水线并行。（#42009）

分布式训练

#38989, #39171, #39285, #39334, #39397, #39581, #39668, #40129, #40396, #40488, #40601, #37725, #37904, #38064)

- 静态图混合并行

- 新增 `scale_gradient` 标志位至 `gradient_scale_configs`, 用于控制流水线并行下梯度聚合运算对梯度进行求平均运算的位置。(#36384)
- 张量模型并行下, `dropout` 支持设置确定性随机种子生成器, 以确保非分布式变量的随机一致性和分布式变量的随机性。(#36228)
- NPU 混合并行支持 Offload, 可节约 40% 显存。(#37224)
- 为 `seed op` 增加 `force_cpu` 可选参数, 使 `dropout` 可以直接从 CPU 读取 `seed` 的值。(#35820)
- 完善 Automatic Sparsity (ASP)sharding 策略, 支持根据 `program` 选择 sharding 策略。(#40028)

- 自动并行

- 新增逻辑进程与物理设备自动映射后的进程重新启动 (relaunch)。(#37523, #37326)
- 完善自动并行底层机制和接口, 利于各个模块统一和添加优化 pass。(#36617, #38132)
- 新增统一资源表示, 支持逻辑进程与物理设备自动映射功能。(#37091, #37482, #37094)
- 完善自动并行计算图反向和更新部分的分布式属性补全功能。(#36744)
- 新增数据切分功能。(#36055)
- 新增张量重切分功能, 根据张量和算子的分布式属性对张量进行重新切分。(#40865, #41106)
- 新增资源数量或并行策略变化时分布式参数的自动转换功能。(#40434)
- 新增梯度累加功能 (GradientMerge), 减少通信次数, 提升训练效率。(#38259, #40737)
- 新增重计算功能 (Recompute), 优化显存。(#38920)
- 新增 Sharding 优化 pass, 支持 p-g-os 3 个 stage 的切分优化。(#38502)
- 新增 AMP + FP16 优化 pass。(#38764, #40615)
- 新增 Transformer 类模型的 QKV fuse 切分。(#39080)
- 新增 while op 的分布式属性推导功能, 确保迭代推导算法能收敛。(#39939, #39086, #39014)
- 支持子 block 和 while op 控制流的训练和推理。(#39612, #39895, #40077)

- 参数服务器

- GPUPS 下, 新增 NAN/INF 值检查工具。(#38131)
- GPUPS 下, 新增 `set_date` 接口, 适配增量训练。(#36194)
- GPUPS 下, 新增异步 `release dataset` 功能。(#37790)
- GPUPS 下, 支持 Dump 参数和中间层 (#36157);
- GPUPS 下, 支持优化器参数配置。(#39783, #39849)

- 统一参数服务器下，重构通信、存储等各个模块基类，提升各个模块的易二次开发性。(#41207, #41022, #40702, #39341, #39377, #39191, #39064)
- 统一参数服务器下，新增评估指标模块，支持 AUC/WuAUC/MaskAuc 等评估指标计算及可自定义扩展。(#38789)
- 支持在昆仑 2 芯片上的 XPU 参数服务器训练。(#41917, #42266, #41916)

Profiler

- Python 层新增性能分析模块 paddle.profiler: 提供对训推过程中性能数据的收集，导出和统计的功能。(#40065, #40357, #40888)
 - paddle.profiler.Profiler, 性能分析器，用户交互的接口。(#41029, #41524, #41157, #40249, #40111, #39964, #40133)
 - paddle.profiler.RecordEvent, 提供自定义打点来记录时间的功能。(#39693, #39694, #39695, #39675, #41445, #41132)
 - paddle.profiler.ProfilerTarget, 指定性能分析的目标设备。
 - paddle.profiler.ProfilerState, 表示性能分析器的状态。
 - paddle.profiler.SortedKeys, 指定统计表单内数据的排序方式。
 - paddle.profiler.make_scheduler, 生成性能分析器状态的调度器，实现采集范围的周期性控制。
 - paddle.profiler.export_chrome_tracing, 将性能数据保存到可供 chrome://tracing 插件查看的 google chrome tracing 文件。(#39316, #39984, #41029)
 - paddle.profiler.export_protobuf, 将性能数据保存到内部结构表示的 protobuf 文件。(#39519, #39109, #39474)
 - paddle.profiler.load_profiler_result, 载入所保存到 protobuf 文件的性能数据。
 - paddle.profiler.Profiler 通过指定 timer_only 参数，对模型进行数据读取、step 开销和吞吐量的统计。(#40386)
- C++ 层重构 Profiler 底层基础设施
 - 重构 Profiler 的控制器架构。(#38826, #39230, #39779)
 - 新增 Host Tracer，收集主机侧性能指标。(#37629, #37766, #37944, #38280, #39975, #40460)
 - 新增 CUDA Tracer，收集设备侧性能指标。(#39488)
 - Profiler 支持分级。(#39926)
- 修改新动态图下 op 的打点名称和类型。(#41771)
- 添加 Kernel 表单，以及优化表单内容的展示方式。(#41989)
- 消除 Profiler 关闭情况下对模型前向计算造成性能下降的影响。(#42142)

CINN 编译器接入

飞桨的编译器功能在逐步丰富中，针对 CINN ([GitHub - PaddlePaddle/CINN: Compiler Infrastructure for Neural Networks](#)) 的变更，Paddle 侧接入也进行了相对应的更改，以适配编译器 CINN 的功能。其中主要包括增加 Paddle-CINN 运行流程的子图管理相关功能，显存和速度性能的优化、开发过程发现的 bug 修复。

- 功能开发：

- 子图 op 相关：

- * 添加从计算图中找到并生成 CINN 子图的功能。(#36345)
- * 新增 cinn_launch op 作为运行时接入 CINN 的入口，负责调度 CINN 对子图进行编译、初始化数据空间、调度生成 Kernel 的执行。(#36600)
- * 为 cinn_launch op 的 Kernel 实现添加辅助类 CinnLaunchContext 管理子图编译、运行的中间数据，提升可扩展性和代码可读性。(#37938)
- * 为 CINN 子图添加额外的 fetch 结点，从而保证 CINN 外部结点能取到待 fetch 变量的值。(#37172, #37190)
- * 添加对 CINN 子图符号化的功能，符号化用于拓扑排序子图并返回 CINN 执行序列。(#36417)
- * 新增 CinnCompiler 类，用于调用 CINN 编译模型中可使用 CINN 算子替换的子图。(#36562, #36975)
- * 为 CINN 符号化类新增获取子图 fetch 变量名的接口，防止编译优化中将 fetch 变量融合消除。(#37218)

- 程序开发检查、debug、API 变更相关：

- * 同步更新 CINN 中 NetBuilder API 名称的变化。(#40392)
- * 为 Paddle-CINN 添加必要的用于 debug 的日志信息。(#36867)
- * 添加 Paddle desc 与 CINN desc 互转函数。(#36100)
- * 相比 Paddle，CINN 中实现的算子可能存在未使用到某些输入变量，因此在 cinn_launch op 中去除对输入变量必须被使用的检查。(#37119)
- * 新增 cinn_instruction_run op 用于调用 CINN 执行单个生成指令，便于 Paddle 侧构建 Graph 调度运行子图。(#39435, #39576)
- * 在 Paddle 中添加编译 CINN 所需的 CUDA/CUBLAS/MKL/CINN pass 应用等控制宏。(#37066, #36660)
- * 增加 FLAGS_allow_cinn_ops 和 FLAGS_deny_cinn_ops 两个控制标记，用于控制 Paddle 训练中使用 CINN 算子代替原生算子的种类。(#36842)

- 性能优化：

- 速度优化

- * 优化 CinnCacheKey 的计算耗时。(#37786, #37317)

- * 缓存 CINN 编译子图的变量 scope，降低运行参数构造开销。(#37983)
 - * 子图编译时接入 CINN 自动调优，支持通过 flag 启用，便于后续进一步调优训练性能。(#41795)
 - * 重构子图编译时对编译结果的正确性校验，避免运行时重复检查，降低调度开销。(#41777)
 - * 在 Paddle-CINN 训练功能中默认启用 TransposeFolding 和 GemmRewriter 优化 pass。(#41084)
 - * 将 Paddle 中创建的 cuda stream 传入 CINN，使得 Paddle 和 CINN 执行计算时共用同一个 CUDA stream。(#37337)
 - * 将 CINN 优化 pass 应用逻辑从 Paddle 中移动到 CINN 中。(#42047, #42070)
- 显存优化
- * 为 cinn_launch op 添加 NoNeedBufferVars 声明无须 buffer 的输入变量列表，以便显存优化提前释放无效空间。(#38367)
 - * 传入子图外部变量的引用计数信息，便于 cinn_launch 内子图复用显存优化 pass，降低使用 CINN 的显存开销。(#39209, #39622)
 - * 添加 CINN 编译生成的可执行指令集合转换为 Paddle Graph 的功能，支持复用 Paddle 调度器及显存优化 pass，进一步降低使用 CINN 的显存开销。(#39724, #39911)
 - * 添加 cinn_instruction_run op 的 Kernel 支持根据编译结果推断的数据类型动态申请空间。(#40920)
- 问题修复：
- 修复并优化 CINN 子图的生成逻辑。(#36503)
 - 修复 Paddle-CINN 不支持无输入子图的问题。(#40814)
 - 修复由于 CINN 无法处理 batch_norm 等算子中存在的无用输出而报错的问题。(#36996)
 - 修复若干 CINN 子图划分以及符号化中存在的 bug，解决 Paddle 训练接入 CINN 全流程打通过程中遇到的问题。(#36739, #36698)
 - CINN 尚不支持控制流，添加遇控制流跳过的逻辑。(#40812)

其他

- 模型量化
 - 升级量化存储格式，并统一动、静态图量化格式。(#41041)
 - 新增离线量化方法：EMD、Adaround。(#40421, #38460)
 - 支持更多 op 适配模 op 量化。(#40083)
 - 支持控制流中的 OP 量化。(#37498)
 - 新增支持 matmul_v2 OP 的量化。(#36469)
 - 新增支持量化后的 matmul_v2 在 TensorRT 上的推理。(#36594)

- 显存优化
 - 实现多 stream 安全 Allocator，支持在多 stream 异步计算场景下安全高效地使用显存。(#37290)
 - 新增运行时显存监控模块 (paddle.device.cuda.max_memory_allocated, paddle.device.cuda.max_memory_reserved, paddle.device.cuda.memory_allocated and paddle.device.cuda.memory_reserved)，支持高性能地实时统计显存数据。(#38657)
 - 实现 CPU-GPU 统一内存寻址 (CUDA Managed Memory)，支持在显存受限场景下训练超大模型。(#39075)
 - C++ 底层新增 GetBasePtr 接口，用来获取设备接口 CUDAAlloc 创建的设备地址。(#37978)
 - 减少 AutoGrowth Allocator 中 free blocks 的数量，提升显存分配性能。(#35732)
 - 对于 initializer.Normal 和 initializer.Constant 数据类型是 FP16 的 Tensor 去除多余的 float32 临时 Tensor 以及 cast，节省 2 倍显存。(#38818)
- 动态图高阶导数组网测试
 - 为动态图增加三阶导数组网测试，以及 Broadcast 情况的测试。(#36814, #37377)
- 自定义 op：支持 ROCm(HIP) 平台进行自定义 op 注册。(#36771)
- Cost Model：增加基于运行 Profile 的 Cost Model。(#35774)
- 提供定制化层 (nn.Layer) 的自动稀疏训练支持，让用户可根据自定义的 Prune 函数来对其设计的层进行稀疏剪枝。(#40253)
- 新增字符串张量底层数据结构表示，使框架具备字符串张量表示和计算的能力。(#39830, #40992)
- 新增或者升级 oneDNN FP32/int8/bfloat16 Kernel，包括：
 - ELU (#37149)
 - exp (#38624)
 - stack (#37002)
 - softplus (#36382)
 - round (#39653)
 - shape (#36033)
 - flatten and flatten2 (#35892)
 - slice (#37630)
 - elementwise_mul (#40546)
 - elementwise_add (#38176)
 - elementwise_div (#36158)
 - elementwise_sub (#35662)
 - roi_align (#37848)

- nearest_interp 和 nearest_interp_v2 (#37985, #38622, #39490)
 - assembly 优化 Adam (#39158)
 - logsoftmax (#39793)
 - activation (#40721)
 - mul (#38552)
 - mean (#37104)
 - relu (#36265)
 - pool2d (#37081)
 - concat (#35889)
 - conv2d (#38507, #38938, #36284)
 - LayerNorm (#40418)
- 增加基于 SSD-内存-GPU 显存的 3 级存储图检索引擎，支持大规模图神经网络训练。(#42472, #42321, #42027)
 - 增加异构多云训练通信模块 switch，实现 Send/Recv 接口，支持多云异构通信。(#40965 40911)

8.3.2 (2) 功能优化

API

- 为 paddle.Model 新增支持混合精度训练 O2 模式，即支持原来动/静态图的 Pure FP16 训练模式。 (#36441)
- 为 paddle.nn.Layer 支持 self chain 调用。 (#36609)
- 为 paddle.nn.Layer 的 to 方法添加 is_distributed 属性的设置，保证网络参数转换前后分布式属性保持一致。 (#36221)
- 完善 paddle.nn.Layer 的 to 方法的参数转换逻辑，降低转换过程占用的峰值显存，提高转换成功率。 (#36862)
- 为 paddle.incubate.graph_send_recv 支持设置输出 Tensor 的 shape，有利于减少实际计算过程的显存占用。 (#40509)
- 为 paddle.incubate.segment_sum、segment_mean、segment_max、segment_min 新增 int32、int64 数据类型支持。 (#40577)
- 为 transpose op 新增 bool 类型支持。 (#35886)
- 将 paddle.mm 底层算子从 matmul 切换到 matmul_v2。 (#35770)
- 为 paddle.einsum 支持静态图模式调用，支持未知 shape。 (#40360)

- 为 `paddle.nn.functional.margin_cross_entropy` 和 `paddle.nn.functional.class_center_sample` 支持数据并行。(#39852)
- 为 `paddle.nn.functional.grid_sample` 支持形状为 [1] 的输入。(#36183)
- 为 `paddle.nn.PRelu` 支持 NHWC 数据格式。(#37019)
- 为 `paddle.nn.functional.class_center_sample` 支持使用 `paddle.seed` 固定随机状态。 (#38248)
- 为 `paddle.fft` 下所有 API 新增 ROCM 后端支持，并优化 CUFFT 后端报错信息。(#36415, #36114)
- 为 `Tensorgetitem` 增加对切片部分维度为 0 的功能支持，即允许切片索引结果为空。(#37313)
- 为 `Tensor.setitem` 支持 int 和 bool 类型 Tensor 使用 bool 索引。(#37761)
- 为 `paddle.nn.functional.interpolate` 支持 nearest 模式时输入 shape 为 5D。(#38868)
- 为 `paddle.nn.Embedding`、`paddle.gather` 增加 int16 支持。(#40964, #40052)
- 为 `paddle.distributed.spawn` 添加 CPU 单机数据并行。(#35745, #36758, #36637)
- 新增 `depthwise_conv2dMKLDNN` 算子。(#38484)
- 为 `paddle.abs`、`paddle.transpose`、`paddle.squeeze`、`paddle.unsqueeze`、`paddle.matmul`、`paddle.full` 静态图数据类型检测中增加复数类型。(#40113)
- 为 `paddle.autograd.PyLayer` 支持 tuple/list 类型的参数。(#38146)
- 为 `paddle.autograd.PyLayer` 增加检查 `inplace` 策略下，输入叶子节点的 `Tensor` 的检查报错机制。 (#37931)
- 为 `paddle.autograd.PyLayer` 支持 HIP 库。(#38184)
- 为 `paddle.take_along_axis`、`paddle.put_along_axis` 支持更多 size 的输入，允许 `index` 矩阵的 shape size 大于 `arr` 矩阵的 shape size。(#39072)
- 优化 API `paddle.nn.Pad2D` 在 `replicate` 为 0 时的报错信息。(#36510)
- 支持 API `paddle.nn.Pad2D` 在 tuple 格式的 pad 输入。(#35985)
- 新增 `paddle.distributed.InMemoryDataset` 中 `tdm_sample` API 以支持 TDM 算法中的采样操作。 (#37044)
- 新增对于 `paddle.jit.save` 的 Pre-saving Hooks 机制。(#38186)
- 新增高阶微分相关 API：
 - `elementwise_add` 增加三阶 Kernel，支持三阶微分的计算。(#36508, #36618)
 - `matmul_v2` 增加三阶 Kernel，支持三阶微分的计算。(#36459)
 - `elementwise_mul` 增加三阶 Kernel，支持三阶微分的计算。(#37152)
- 完善 `paddle.amp.GradScaler` 调用 `check_finite_and_unscale op` 的逻辑，消除该处创建 bool 变量所引入的 `cudaMemcpy`。(#37770)

- 新增对 unstack 和 unique op 元素个数为 0 的 Tensor 增加检查。(#36021)
- 新增支持昆仑 2 的多层、双向 LSTM 功能，完善 RNN 前反向 op，支持时序类模型训练使用。(#42076)
- 新增支持昆仑 2 的 bce_loss 前反向 op。(#41610)
- 添加 paddle.linalg.det 的反向实现。(#36013)

IR(Intermediate Representation)

- 动态图转静态图
 - 优化动转静下 ProgramCache.last 接口行为，使其返回最近使用的 Program，而非最后生成的 Program。(#39541)
 - 优化动转静下 paddle.reshape API 的报错信息，新增推荐用法提示。(#40599)
 - 优化动转静代码转写时 is_api_in_module 函数中异常捕获类型。(#40243)
 - 优化动转静模块报错提示，默认隐藏 warning 信息。(#39730)
 - 增加动转静对于 type hint 语法的支持，提高变量类型分析的准确性。(#39572)
 - 优化 paddle.cond 功能，允许 bool、int 等基本类型支持值相等。(#37888)
 - 优化动转静 @to_static 装饰普通函数时，允许切换 train/eval 模式。(#37383)
 - 优化动转静报错栈，突出用户相关代码，减少框架冗余报错栈。(#36741)
 - 移除 paddle.cond 返回值中 no_value 占位符。(#36513、#36826)
 - 为动转静 run_program op 适配新动态图模式。(#40198, #40355)
 - 新增对于 zip 语法的检查。(#37846)
 - 修复 paddle.signal.frame、paddle.signal.stft、paddle.signal.istft 因维度和类型判断错误导致的动转静失败问题。(#40113)
 - 为 mean、pad3d ops 新增注册复数类型 Kernel。(#40113)

混合精度训练

- 为 amp 添加 GPU Compute Capability 环境检查，对无法产生训练加速效果的 GPU 环境添加使用警告。(#38086)
- 添加 paddle.amp.decorate 与 paddle.DataParallel 同时使用时调用顺序的检查。(#38785)

分布式训练

- 分布式训练基础功能
 - 优化 Fleet API 和 DistributedStrategy 配置以使用动态图并行功能，提升动态图易用性。(#40408)
 - 优化动态图混合并行 HybridParallelClipGrad 策略，支持 4D 混合并行 + Pure FP16 训练。(#36237, #36555)
 - 重构动态图数据并行策略，以支持新动态图和新通信库功能。(#40389, #40593, #40836, #41119, #41413, #39987)
 - 为 fused_attention op 支持分布式张量模型并行。(#40101)
 - 为 fused_feedforward op 支持分布式张量模型并行。(#40160)
- 图检索引擎
 - 优化图引擎的图采样接口返回的数据格式，采样速度提升 3 倍。(#37315)
 - 减少图引擎线程量以提升性能。(#37098)
 - 优化图引擎数据传输以提升性能。(#37341)
 - 利用模型中 embedding op 的拓扑关系，优化 embedding op 的合并逻辑以提升性能。(#35942)
- 通信库：重构通信库，提升通信库的易扩展性和二次开发性，支持异构通信。(#41398, #39720, #40911, #40579, #40629, #40437, #40430, #40228, #40181, #40100, #40097, #39892, #39384, #39737, #40040)
- 支持 paddle.incubate.distributed.models.moe 中 MoE 相关接口 (moe.GShardGate, moe.BaseGate, moe.SwitchGate, moe.MoELayer, moe.ClipGradForMOEByGlobalNorm) 的公开。(#42300)
- 修复 paddle.incubate.distributed.models.moe.MoELayer 中使用 recomputing 可能报错的问题。(#42128)
- 修复新动态图流水线并行因为数据类型不同导致的报错 (#41937 #42053)
- 修复新动态图张量模型并行因为数据类型不同导致的报错 (#41960)

自定义算子

- 增强 C++ 自定义算子机制对二阶反向算子编写功能，支持为二阶反向算子的梯度输入变量添加后缀作为输出使用。(#41781)
- 移除 Tensor API 成员方法中对废弃的枚举类型 PlaceType 的使用，进行相应兼容处理，并添加 deprecated warning 提示。(#41882)
- 为原 Tensor API 的一系列废弃接口，包括不完整构造函数、reshape、mutable_data、copy_to 方法添加 deprecated warning 提示。(#41882)

其他

- 报错调试优化
 - 优化 cross_entropy op 对 label 的边界检查报错信息。(#40001)
 - 为动态图添加 op 执行时 infer_shape 和 compute 方法的 profile record，用于在 timeline 中展示其开销。(#39023)
 - 替换了 Windows 下容易出现未知异常的 pybind::index_error 报错提示。(#40538)
 - 添加用户 scatter op 越界检查的报错信息。(#37429)
- 下载工具：针对 paddle.utils.download.get_path_from_url 中解压含多文件目录速度慢的问题，将原先循环遍历目录下文件逐一解压的方式替换为在目录上调用 extractall 一次解压的方式，解压速度大幅提升。(#37311)
- 加速 fake_quantize_range_abs_max、fake_quantize_abs_max、fake_quantize_dequantize_abs_max、fake_quantize_moving_average_abs_max 等量化训练。(#40491)

8.3.3 (3) 性能优化

分布式训练

- 混合并行优化器 sharding 支持 optimize_cast 优化，将前反向参数 cast 移到优化器阶段，性能提升 7%。 (#35878)
- GPUPS 优化：支持梯度 fuse allreduce 训练，训练提升 20%。(#35131)
- GPUPS 优化：dump CPU 优化提速 3.21 倍。(#40068)
- CPU 参数服务器流式训练优化：支持稀疏参数统计量自动统计、稀疏参数增量保存等功能，训练性能提升 20%。(#36465, #36601, #36734, #36909, #36943, #37181, #37194, #37515, #37626, #37995, #38582, #39250, #40762, #41234, #41320, #41400)

算子优化

- 优化 FasterTokenizer 性能，性能与优化前相比提升 10%。(#36701)
- 优化 index_select 反向计算，性能较优化前有 3.7~25.2 倍提升。(#37055)
- 优化 paddle.nn.ClipByGlobalNorm 的性能，以 10*10 的 paddle.nn.Linear 为例，性能与优化前相比提升 30% 左右。(#38209)
- 优化 pnorm 在 axis 维度极大或极小情况下的性能，前向速度提升 31~96 倍，反向速度提升 1.1~19 倍。 (#37685, #38215, #39011)

- 优化 softmax 前、反向性能，对于 axis!= -1 的配置加速比为 2 倍左右。 (#38602, #38609, #32387, #37927)
- 优化 log_softmax 前、反向性能，对于 axis!= -1 的配置加速比为 6~20 倍左右。 (#38992, #40612)
- 优化 softmax_with_cross_entropy 前、反向性能，对于 hard_label 的配置加速比为 1.3 倍左右。 (#39553, #40424, #40643)
- 优化 top_k 性能，对于一维且 k 较大时 (k=5000) 的配置加速比为 22 倍以上。 (#40941)
- 优化 elementwise_mul 反向计算，较优化前有 1.85~12.16 倍性能提升。 (#37728)
- 优化 elementwise_min 反向和 elementwise_max 反向，较优化前打平或有 1.05~18.75 倍性能提升。 (#38236, #37906)
- 优化 nearest_interp 前向和反向计算，前向较优化前性能有 1.5~2.3 倍提升；反向性能较优化前有 60%~1.8 倍提升。 (#38528, #39067)
- 优化 bilinear_interp 前向和反向计算，前向较优化前性能有 0.4~2.3 倍提升；反向性能较优化前有 10%~30% 提升。 (#39243, #39423)
- 优化 dropout 前向和反向计算，性能提升约 20%。 (#39795, #38859, #38279, #40053)
- 优化 grid_sampler 前向和反向计算，前向较优化前性能有 10%~30% 提升；反向性能较优化前有 10%~60% 提升。 (#39751)
- 优化 group_norm 前向和反向计算，前向性能提升 1.04~2.35 倍，反向性能提升 1.12~1.18 倍。 (#39944, #40657, #39596)
- 优化 conv1d 前向和反向计算，前向性能提升 1.00~2.01 倍，反向性能提升 1.01~474.56 倍。 (#38425)
- 优化 elementwise_div 反向计算，反向性能提升 1.02~29.25 倍。 (#38044)
- 优化 gelu 前向和反向计算，前向性能提升 1.13~1.43 倍，反向性能提升 1.10~1.55 倍。 (#38188, #38263)
- 优化 elementwise_sub 反向计算，反向性能提升 1.04~15.64 倍。 (#37754)
- 优化 flip 在输入一维数据时前向性能，性能提升 100%。 (#37825)
- 优化 layer_norm 前向和反向计算，前向较优化前提升 2~5 倍，反向较优化前提升 20%~50%。 (#39167, #39247)
- 优化 embedding 前向和反向计算，前向较优化前最大提升 1.51 倍，反向较优化前提升 1.03~7.79 倍。 (#39856, #39886)
- 优化 gelu FP16 前向和反向计算，前向较优化前提升 9%~12%，反向较优化前提升 2%~9%。 (#38980)
- 移除 gather_nd 前反向算子中的 CPU -> GPU 显式数据传输操作，移除 index_select 前反向算子中的显式同步操作，将 scatter_nd 中的 GPU -> GPU 数据传输由同步操作改成异步操作。 (#40933)
- 优化 Lars optimzier 计算，优化后 Resnet50 FP16 模型训练性能较优化前提升 5.1%。 (#35652, #35476)
- 优化 AvgPool2dGrad 计算，优化后性能较优化前提升 2.6 倍。 (#35389)

- 优化 Elementwise 类计算对于多元输出的功能支持，优化后计算性能较优化前提升最多可达 15%。（#38329, #38410）
- 优化 Categorical 的 probs 计算，简化计算逻辑，性能提升 4~5 倍。（#42178）
- paddle.sum 性能优化，性能相比优化前提升约 20%。（#42309）

自动调优

新增训练全流程硬件感知性能自动调优功能，在图像分类、分割、检测和图像生成任务上与模型默认参数配置下的性能相比提升约 3%~50% 以上。通过 paddle.incubate.autotune.set_config API 设置自动调优状态，当前默认关闭。自动调优具体包括三个层次：

- paddle.io.DataLoader 新增自动调优功能，根据训练数据和设备资源选择最佳的模型 num_workers。（#42004）
- 新增混合精度训练数据布局自动调优功能，根据设备类型和数据类型选择最佳数据布局，并在运行时自动转换。（#41964）
- 新增 Conv 运行时所需 workspace size 阈值自动调整功能，根据 GPU 当前可申请显存资源情况来自动设置；基于通用的 AlgorithmCache 设计和 Kernel 计时组件，新增 Conv cuDNN 算法自动选择功能，支持数据变长模型。（#41833）

调度优化

- 移除 paddle.nn.ClipGradByGlobalNorm 中的 CudaStreamSync 隐藏操作，减少执行时的调度开销，在 ptb 模型上有 5% 的性能提升。（#42170）
- 优化一系列底层数据结构及原动态图执行体系中的细节实现，提升原动态图的调度性能。（#42010, #42171, #42224, #42256, #42306, #42329, #42340, #42368, #42425）
- 简化 paddle.distribution.Categorical 的 probs 计算逻辑，提升性能 4 到 5 倍。（#42178）

8.3.4 (4) 问题修复

API

- 修复 paddle.sum 输入参数类型和输出参数类型不一致且 axis 轴对应的 reduce 元素个数为 1 时，输出类型错误问题。（#36123）
- 修复 paddle.flops 在 layer 输出类型为 tuple 时的 AttributeError。（#38850）
- 修复 paddle.diag 因为没有反向 Kernel 而无法传播梯度的问题。（#40447）
- 修复 paddle.sort 输入存在 NaN 值排序错误。（#41070）
- 修复 paddle.full_like 输入存在 Inf 值构建 Tensor 错误。（#40232）

- 修复 paddle.strided_slice 在输入 starts 中数据小于 -rank 时， strided_slice 结果与 slice 不一致的 bug。(#39066)
- 修复 max_pool 系列算子在返回 index 时 infer_shape 计算错误的问题，受影响的 API 有 paddle.nn.functional.max_pool1d/2d/3d, paddle.nn.functional.adaptive_max_pool1d/2d/3d, paddle.nn.MaxPool1D/2D/3D, paddle.nn.AdaptiveMaxPool1D/2D/3D。(#40139)
- 修复 max_pool 系列算子返回的 pooling_mask 的 dtype 错误的问题，现在 pooling_mask 的 dtype 为 int32，受影响的 API 有 paddle.nn.functional.max_pool1d/2d/3d, paddle.nn.functional.adaptive_max_pool1d/2d/3d, paddle.nn.MaxPool1D/2D/3D, paddle.nn.AdaptiveMaxPool1D/2D/3D。(#39314)
- 修复 paddle.shape 默认存在反向梯度导致计算错误的问题。(#37340)
- 修复 paddle.nn.Layer 的 to 方法同时转换 dtype 和 place 存在的 bug。(#37007)
- 修复 paddle.amp.decorate 无法对非叶子网络层的参数改写为 FP16 的 bug。(#38402)
- 修复 paddle.amp.decorate 将 paddle.nn.BatchNorm1D、paddle.nn.BatchNorm2D、paddle.nn.BatchNorm3D 非输入参数改写为 FP16 的 bug。(#38541)
- 修复 paddle.amp.decorate 将 paddle.nn.SyncBatchNorm 非输入参数改写为 FP16 的 bug。(#40943)
- 修复 paddle.nn.Layer.to 当中多余的 warning。(#36700)
- 修复 paddle.nn.RNN 在控制流下使用报错的问题。(#41162)
- 修复 paddle.to_tensor 无法指定 Tensor 的 CUDA Place 的问题。(#39662)
- 修复 paddle.nn.Identity 没有公开的问题。(#39615)
- 修复动态图重构后，fill_ 和 zero_ inplace API 的输入在 CUDA Pinned Place 上时，输出值不正确的 bug。(#41229)
- 动态图重构后，修复使用 append op 的方式调用 assign op 导致输出 Tensor 的 inplace version 值不正确的 bug，修改为使用 _C_ops 的方式调用 assign op。(#41118)
- 移除 elementwise_add 三阶 Kernel 中不合理的代码，修复组网过程未初始化问题。(#36618)
- 修复 conv2d 执行 cuDNN Kernel 时属性缺失的问题。(#38827)
- 修复 multiclass_nms3 输出 shape 不正确的问题。(#40059)
- 修复 yolo_box 输出 shape 不正确的问题。(#40056)
- 修复高阶微分 gradients 接口在指定 target_grad 时未按预期生效的问题。(#40940)
- 修复动态图 op_BatchNormBase 基类中修改了 default_dtype，导致后续组网参数类型错误的问题，受影响的 API 有 paddle.nn.BatchNorm1D, paddle.nn.BatchNorm2D, paddle.nn.BatchNorm3D, paddle.nn.SyncBatchNorm。具体原因是当 get_default_dtype() == 'float16' 时，通过 set_default_dtype('float32') 修改默认参数数据类型，动态图组网的参数类型是通过 default_dtype 来创建的，因此当默认参数类型被修改后导致后续的组网参数类型错误。(#36376)

- 修复 batchnorm op 中，当数据类型为 FP32，且数据维度 dims = 2, data_layout = NHWC 时，反向 op 内中间变量未定义问题。(#37020)
- 修复静态图模式下，paddle.static.nn.prelu 对于 NHWC 输入格式且 mode==channel 权重的 shape 错误问题。(#38310)
- 修复多机情况下，paddle.nn.functional.class_center_sample CUDA 种子设置 bug。(#38815)
- 修复 paddle.nn.functional.one_hot 在输入不正确参数时，CUDA 版本无法正确报错的问题。(#41335)
- 修复 DCU 设备上回收显存的 callback 未及时触发导致显存 OOM 的问题。(#40445)
- 修复 setitem 索引赋值反向梯度传递异常以及动态图部分场景下 inplace 逻辑处理异常的问题。(#37023, #38298)
- 修复动转静下 Tensor array 使用 Slice 索引异常的问题。(#39251)
- 修复 paddle.Tensor.register_hook 接口使用时临时变量未析构，从而导致内存或显存泄漏的问题。(#40716)
- 修复 Tensorgetitem 当索引是全为 False 的 bool Tensor 时无法取值的问题。(#41297)
- 修复 Tensorgetitem 当索引是 bool scalar Tensor 时无法取值的问题。(#40829)
- 修复 paddle.index_select 在 index 为 0-shape Tensor 时报错的问题。(#41383)
- 修复 paddle.index_select, paddle.index_sample 申请的 GPU 线程数超过有限机器资源时报错的问题。(#41127, #37816, #39736, #41563)
- 修复 ReduceConfig、elemwise_grad、gather、gather_nd、scatter ops 申请 GPU 线程数超过有限机器资源时报错的问题。(#40813, #41127)
- 修复 Kernel Primitive API 中 ReadData, ReadDataBc, ReadDataReduce 在 NX != 1 时访存越界的问题。(#36373)
- 修复 IndexRandom 数据类型错误导致数据溢出计算结果异常的问题。(#39867, #39891)
- 修复 reduce op 在 reduce_num = 1 计算结果返回错误的问题。(#38771)
- 修复 reduce op 在 HIP 环境下 reduce 中间维度出现访存越界的问题。(#41273)
- 修复 matmul op 两个 FP16 一维向量计算时 Kernel 无法正常释放的问题。
- 修复部分算子在 CUDA 上因整型计算溢出导致的问题，包括：bernoulli、gaussian_random、gumbel_softmax、multinomial、truncated_gaussian_random、uniform_random_inplace、uniform_random ops。(#37670)
- 修复 paddle.nn.Sequential 在 for 循环遍历 sublayers 时会报 KeyError 错误的 bug。(#39372)
- 修复 paddle.nn.functional.unfold 在静态图下编译时检查 shape 错误的 bug。(#38907, #38819)
- 修复静态图使用 dropout 时如果指定了 axis 后会报错的问题。(#37223)
- 迁移 paddle.nn.MultiHeadAttention 中 matmul 算子到 matmul_v2 算子。(#36222)
- 修复 paddle.nn.functional.label_smooth 在输入为空 Tensor 时抛出 FPE 的问题。(#35861)

- 修复 reshape op 空 Tensor 形变问题，支持将空 Tensor reshape 成 [-1]。(#36087)
- 修复 fill_diagonal 参数 offset 非零时会造成修改值跨行问题。(#36212)
- 修改动态图模式下 range op 返回 stop gradient 设置成 True。(#37486)
- 修复 Lamb 优化器当 Beta1Pow 和 Beta2Pow 在 GPU 上时更新错误的 bug。(#38518)
- 修复 conv2d 算子 FLAGS_cudnn_deterministic 设置不生效的问题。(#37173)
- 修复因早期版本的 cufft 没有定义 CUFFT_VERSION 引发的问题。(#37312)
- 修复 paddle.ifftshift, paddle.fftshift 计算错误问题。(#36834, #36748)
- 修复 paddle.fft 系列 API 中的 axis 计算错误。(#36321)
- 修复 batch_norm_grad op 在 FP16 数据类型时输出数据类型注册的 bug，该 bug 会导致部分场景下编译失败，并且对 FP16 计算精度会有一定影响。(#42461)
- 修复 paddle.nn.functional.pad API 在模型动转静时，padding 为 Tensor 条件下的 Infershape 信息错误问题。(#42414)
- 修复 paddle.distribution.StickBreakingTransform 输入维度超过 2 时异常的问题。(#41762)
- 修复 fused_attention op 中 QK^T 计算出 nan/inf 的问题。(#42032)
- 修复 fused_attention op 中 FusedResidualDropoutBias 在 V100 上计算出 nan/inf 问题。(#42398)
- 修复 full_like op 在执行时引入的多余的 data transform 问题。(#41973)
- 修复 p_norm op 在 GPU 环境上计算 nan 的问题。(#41804)
- 修复 split op 在参数 sections 存在为 0 的 size 情况下，段错误的问题。(#41755)
- 修复 6 个 elementwise op (pow、complex、divide_double、multiply_double、fmax、fmin) 在需要 broadcast 的情况下，多卡训练时报 Place(gpu:0) 不支持的问题。(#42332)
- 修复 import paddle 时由于 PIL 版本升级导致的废弃接口报 warning 的问题。(#42307)
- 修复静态图下 paddle.linalg.matrix_rank 不支持 tol 为 FP64 Tensor 的问题。(#42085)

IR(Intermediate Representation)

- 动态图转静态图
 - 修复 tensor_array 搭配控制流使用时，在反向梯度累加时存在的类型推导错误问题。(#39585, #39689)
 - 修复动转静 AMP 训练时参数梯度类型未被正确设置的问题。(#40938)
 - 修复代码中存在错位注释时，动转静代码解析报错的问题。(#39035, #38003)
 - 修复动转静代码中调用非 forward 函数时，Tensor 未被正确转化为 Variable 的问题。(#37296, #38540)
 - 修复动转静代码转写时 paddle 被错误地作为变量传递的问题。(#37999)

- 修复模型动转静后调用 paddle.flops 时模型参数统计错误的问题。(#36852)
- 修复使用 paddle.jit.save/load 接口加载模型后，在 train 模式和 no_grad 上下文中，显存会一直增长的问题。(#36434)
- 添加在 convert_call 对 generator function 转换时的警告。(#35369)
- 修复 run_program op 依赖分析的问题。(#38470)
- 修复控制流 For 中返回单值时代码转换的问题。(#40683)
- 修复控制流 cond 的输入包含 LoDTensorArray 时，生成反向 op 会报错的问题。(#39585)
- 修复 paddle.jit.save 在导出动转静模型时丢失顶层 Layer 的 forward_pre_hook 和 forward_post_hook 的问题。(#42273)
- 修复 paddle.expand 中 shape 参数包含 Tensor 在动转静时会转换报错的问题。(#41973)

分布式训练

- 分布式训练基础功能
 - 修复分布式多机训练时，端口报错的问题。(#37274)
 - 修复 brpc 编译依赖问题。(#37064)
 - 修复 Fleet 启动时，由于 tcp 自连接产生的端口被占用的问题。(#38174)
 - 修复数据并行下，由于 FP16 参数在多卡下初始化不一致，导致精度下降的问题。(#38838, #38563, #38405)
 - 修复数据并行下，由于 FP16 梯度同步时，没有除以卡数，导致精度下降的问题。(#38378)
- 动态图混合并行
 - 修复在混合并行下，通过使用新 update 接口，FP16 模式不更新参数的问题。(#36017)
- 静态图混合并行
 - 修复分布式 dp 模式下 grad merge 与 ClipGradientByGlobalNorm 不兼容的问题。(#36334)
 - 修复混合并行下，张量模型并行的非分布式参数在初始化阶段未被广播，导致各卡非分布式参数不一致的问题。(#36186)
 - 修复 sharding 开启 offload 时，sharding 的 save_persistables 接口未保存 FP16 参数和 offload 持久化变量的问题。(#40477)
 - 修复开启 sharding 训练时，ema 参数在非 0 号卡上无法保存的问题。(#39860)
 - 修复 FC 按照列切分梯度计算错误的问题。(#38724)
 - 修复 DistributedStrategy 设置为 without_graph_optimizer 时和 rnn 一起使用报错的问题。(#36176)
- GPUPS 参数服务器训练
 - 修复 GPUPS 宏定义触发 CPU 分支编译问题。(#37248)

- 修复 GPUPS 流水线训练时在保存 delta 和 pullsparse 并发时引发的偶发报错问题。(#37233)
- 修复 HDFSClient 查询目录未返回全路径，引发下载报错问题。(#36590)
- 修复 GPUPS 流水线训练时拉取老参数问题。(#36512)
- 修复 GPUPS 多流 allocation 问题。(#37476)
- 修复 GPUPS pybind 出 core 的问题。(#37287)

其他

- 修复动态图量化训练保存模型时 clip_extra 的问题。(#38323)
- 修复动态图量化训练 abs_max scale 初始化的问题。(#39307)
- 修复动态图量化训练保存模型节点异常的问题。(#38102, #38012)
- 修复离线量化 flatten op 输出错误问题。(#37722)
- 修复了反量化 matmul op 时，维度对不上的问题。(#36982)
- 修复了量化无权重的 matmul_v2 时，错误添加量化 op 的问题。(#36593)
- 修复 conv op channel wise 量化在保存模型时 quant_axis 属性保存错误。(#39054)
- 修复 ChannelWise 量化训练速度慢的问题。(#40772)
- 修复量化训练初始化为 0 的 Tensor 出 NAN 的问题。(#36762)
- 修复多线程场景下混合精度 amp_level 设置错误问题。(#39198)
- 修复混合精度训练与 PyLayer, Recompute 等一起使用时，PyLayer 和 Recompute 中未正确设置混合精度的问题。(#39950, #40042)
- 修复了 Mac 下编译自定义算子时 D_GLIBCXX_USE_CXX11_ABI 未生效的问题。(#37878)
- 修复 initializer 相关 API 在 block=None 时动静行为不统一的问题。(#37827)
- 修复 python3.6 环境下没有 fluid 模块的 bug。(#35862)
- 修复优化器 paddle.optimizer.AdamW 错误调用 adam op 的 bug。(#36028)
- 修复 multi tensor 策略下 paddle.optimizer.Momentum 优化器参数 regularizer 属性为 None 时的逻辑错误。(#38344)
- 修复 multi tensor 策略下 paddle.optimizer.Momentum、paddle.optimizer.Adam 优化器会对 multi_precision 属性进行修改的错误。(#38991)
- 修复最终态 API amp 与 optional 类型 Tensor 组合使用的代码编译错误。(#40980)
- 修复 paddle+lite+xpu 预测库调用 lite CPU 预测时会报错的 bug，修复 paddle+lite(without NNAdapter) 编译时会报错的 bug。(#37449)
- 修复 Debug 编译模式下 LoDTensorArray 因 Pybind11 绑定不一致导致 crash 的 bug。(#37954)

- 修复 shape 参数为 Tensor 和 int 构成列表的极端情况下，无法正确构建 Tensor 的 bug。(#38284)
- 修复 paddle.optimizer.AdamW API 兼容性问题。(#37905)
- 修复 _InstanceNormBase 中 extra_repr 的返回错误。(#38537)
- 修复 联编开启 -DWITH_DISTRIBUTED 生成 Paddle Inference 缺少符号 paddle::distributed::TensorTable 的问题。(#41128)
- matmul_v2 op 新增 shape check，在 shape 中存在 0 值进行信息报错。(#35791)
- 修复动态图 recompute 对于没有梯度输入提示信息反复打印，改成用 warning 只打印一次的方式。(#38293)
- 修复 gelu op 在视觉模型中训练后期在验证集上精度低的问题。(#38450)
- 修复 adamw op 在数值计算上误差问题。(#37746)
- 补充 sparse_momentum_C_ops 接口 MasterParam 和 MasterParamOut 参数。(#39969)
- 修复 python3.6 环境下没有 distributed 模块的 bug。(#35848)
- 修复 eigh 单元测试数据初始化问题。(#39568)
- 修复 eigvalsh 单元测试数据初始化问题。(#39841)
- 修复 segment op 在 V100 上寄存器使用过多导致不能正常运行的问题。(#38113)
- 修复 conv 相关算子稀疏化维度错误的问题。(#36054)
- 提供自动稀疏训练(Automatic SParsity)静态图相关功能 Alias 至 paddle.static.sparsity。(#36525)
- 修复 divide op 整数除法还是整数的 bug。(#40890)
- 修复 paddle.multiplex 候选 Tensor 大小为 0 崩溃问题。(#34972)
- 修复 paddle.kl_div 参数 reduction 给定情况下速度异常的问题。(#37283)
- 修复 Cifar 数据集加载 data source 无序的问题。(#37272)
- 修复 ProgressBar 类中 loss 从 uint16 到 float 的转换。(#39231)
- 修复 ShareBufferWith 共享数据类型的问题。(#37464, #37247)
- 修复 paddle.io.DataLoader 使用 IterableDataset 并且 num_workers>0 时的性能问题。(#40541)
- 修复 paddle.vision.ops.yolo_loss 动态图返回值不全的问题。(#40185)
- 移出 paddle.io.BatchSampler 对输入参数 dataset 需要是 paddle.io.Dataset 类型的限制，扩大对用户自定义数据集的支持。(#40184)
- 修复 paddle.summary 报错 op_flops 不存在的问题。(#36489)
- 修复 lars_momentum op 在 lars_weight_decay=0 时公式错误的问题。(#40892)
- 修复 optimize-offload 无法保存 presistable var 的问题。(#36433)
- 修复 optimizer-offload 不支持 adamw op type 的问题。(#36432)

- 修复多线程场景下，Tracer 中 enable_program_desc_tracing_ 数据不安全的问题。(#39776)
- 修复模型读取时模型档案大小未初始化的问题。(#40518)
- 修复 Expand op 逻辑 bug，当输入 Tensor X 的维度，小于要拓展的 shape 时，可能导致取得 Out.Shape 是错误的。(#38677)
- 修复 Expand_As op 只取 y.shape，而没有 Y 变量输入时，导致的动转静报错。(#38677)
- 修复 Expand_As op 计算输出 shape 时逻辑的错误。(#38677)
- 修复 core.VarDesc.VarType.STRINGS 类型的变量获取 lod_level 属性报错的问题，并且设置其 lod_level 为 None。(#39077)
- 修复框架功能 PyLayer 不支持不同 dtype 的问题。(#37974)
- 修复了学习率衰减 API paddle.optimizer.lr.PolynomialDecay 的零除问题。(#38782)
- 修复调用 DisableGlogInfo() 接口后依旧残留部分日志的问题。(#36356)
- 修复 SimpleRNN、GRU 和 LSTM API CPU 训练时多层 RNN（dropout 设置为 0 时）反向计算出错的问题。(#37080)
- 为 cufft 和 hipfft 后端的 fft 添加了 cache。(#36646)
- 使 paddle.roll 的 shifts 参数支持传入 Tensor。(#36727)
- 为 fft 添加 onemkl 作为可选的计算后端。(#36414)
- 修复 mamtul_v2 和 elementwise_div 两个 op 在 bfloat16 类型下的精度问题。(#42479)
- 修复显存回收时 LoDTensorArray 只清理内部 Tensor 而未清空 Array 导致的下个 step 可能出错的问题。(#42398)

8.4 部署方向 (Paddle Inference)

8.4.1 (1) 新增特性

新增 API

- 增加 Java API，Java 开发者可以通过简单灵活的接口实现在服务端和云端的高性能推理。(#37162)
- 增加 GetTrtCompileVersion 和 GetTrtRuntimeVersion 接口，用于获取 TensorRT 版本信息。(#36429)
- 增加 ShareExternalData 接口，避免推理时对输入数据进行内存拷贝。(#39809)

新增功能

- 新增 ONNX Runtime 后端支持，当前集成版本只支持 CPU。(#39988, #40561)
- 基于 Paddle Lite 子图方式，新增昇腾 310 推理支持。(#35226)
- 新增原生 GPU FP16 推理功能。(#40531)
- switch_ir_debug 接口增加 dump 模型的功能。(#36581)
- 新增 TensorRT config 的配置接口: void UpdateConfigInterleaved(paddle_infer::Config* c, bool with_interleaved)，用于 int8 量化推理中特殊的数据排布。(#38884)
- log 中增加 TensorRT inspector 输出信息，仅在 TensorRT 8.2 及以上版本有效。(#38362, #38200))
- 增加 TensorRT ASP 稀疏推理支持。(#36413)

8.4.2 (2) 底层优化

CPU 性能优化

- 优化 MKLDNN 的缓存机制。(#38336, #36980, #36695)
- 新增 matmul_scale_fuse pass。(#37962)
- 新增 MKLDNN reshape_transpose_matmul_v2_mkldnn_fuse_pass。(#37847, #40948)
- 新增 MKLDNN conv_hard_sigmoid_mkldnn_fuse_pass。(#36869)
- 新增 MKLDNN matmul_v2_transpose_reshape_fuse_pass。(#36481)
- 新增 MKLDNN softplus_activation_mkldnn_fuse_pass。(#36657)
- 新增 MKLDNN elt_act_mkldnn_fuse_pass。(#36541)
- 新增 MKLDNN mish 算子及 conv_mish_mkldnn_fuse_pass。(#38623)

GPU 性能优化

- 将推理默认的显存分配策略由 naive_best_fit 变更为 auto_growth，解决部分模型占满 GPU 显存问题。(#41491)
- 支持 gelu、FC+gelu ops 使用 TensorRT 推理。(#38399) 合作团队
- 支持 deformable_conv 在静态 shape 下使用 TensorRT 推理。(#36612 #36850 #37345)
- 支持 nearest_interp_v2 op 使用 TensorRT 推理。(#34126)
- 增加 yolo_boxTensorRT plugin，支持输入参数 iou_aware 和 iou_aware_factor，使推理计算得到的 IoU 作为置信度的因子。(#34128)
- 支持 elementwise_sub 和 elementwise_div 调用 TensorRT 推理。(#40806 #41253)

- 支持 multiclass_nms3 使用 TensorRT 推理。 (#41181 #41344)
- 支持 flatten_contiguous_rang op 使用 TensorRT 推理。 (#38922)
- 支持 pool2d 属性 padding 的维度为 4、global_pooling 和 ceil_mode 为 True 情况下使用 TensorRT 推理。 (#39545)
- 支持 batch_norm 和 elementwise_add 为 5 维时使用 TensorRT 推理。 (#36446)
- 新增 pool3d 使用 TensorRT 推理。 (#36545, #36783)
- 增加 reduce int32 和 float 类型使用 TensorRT 推理，增加 reduce_mean GPU 算子 int32、int64 注册。 (#39088)
- 修改 MatmulV2ToMul pass，修改限定条件（不支持广播）和 op_teller 映射条件。 (#36652)
- 增加 TensorRT plugin 接口 AddPluginV2IOExt 的支持。 (#36493)
- 增加 roi_align op 中 aligned 属性并支持 TensorRT 推理。 (#38905)
- 增加 concat 属性 axis = -1 时支持 TensorRT 推理。 (#39096)
- 新增 TensorRT plugin：preln_emb_eltwise_layernorm、preln_skip_la、rnorm ops，用于 ERNIE 类模型性能优化。 (#39570)
- 新增 TensorRT fuse pass：preln_embedding_eltwise_layernorm_fuse_pass, preln_skip_layernorm_fuse_pass，用于 ERNIE 类模型性能优化。 (#39508)
- 将 matmul 融合相关的 pass 基于不同的后端（GPU、CPU、TensorRT）拆开，支持 FC 权重的转置功能。 (#39369)
- 新增 roll、strided_slice、slice op 在动态 shape 的情况下对 TensorRT 的支持。 (#41913, #41573, #41467)
- 新增 div op 对 TensorRT 的支持。 (#41243)
- 量化支持
 - PostTrainingQuantization API 新增支持 paddle.io.DataLoader 对象或者 Python Generator 的输入。 (#38686)
 - ERNIE 全量化模型推理支持 interleaved 数据排布。 (#39424)
 - 支持 PaddleSlim 新量化模型格式推理。 (#41049)
 - 新增 matmul int8 量化的推理 op converter 和 plugin。 (#37285)
 - 新增判断模型所有 op 能否支持 int8 量化的 pass。 (#36042)
 - 支持 multihead attention 非变长分支中 FC 部分的量化推理。 (#39660)

昇腾 NPU 相关功能

- 重构 shape 算子前向计算逻辑，支持在 NPU 上执行。(#39613)
- 重构 reshape 算子前向计算逻辑，支持 ShapeTensor 输入。(#38748)
- 模型权重加载时精度类型统一。(#39160)

8.4.3 (3) 问题修复

框架及 API 修复

- 修复保存静态图时模型剪裁的问题。(#37579)
- C API 增加对的字符串的封装 PD_Cstr，并提供构造和析构的方式，避免用户直接使用 C 运行时库来析构字符串。(#38667)
- 修复预测时内存复用的逻辑问题。(#37324)
- 修复多线程下内存复用报错问题。(#37894)
- 在没有权重文件时，允许传递空字符串进行推理。(#38579)
- 修复开启 TensorRT dynamic shape 后不支持 clone 问题。(#38520)
- 修复开启 TensorRT dynamic shape 后多线程 clone 报错问题。(#40067)
- 修复 TensorRT engine 析构问题。(#35842, #35938)
- lite xpu 接口修复无法选择 xpu 卡的问题。(#36610)
- TensorRT 动态 shape 参数自动生成接口增加文件存在性检查。(#36628)
- 修复 MKLDNN 不支持 conv3d 的问题。(#42055)

后端能力修复

- 修复预测时 cuDNN 默认算法选择配置，使用非 deterministic 策略。(#41491)
- 修复 deformable_conv op 在 TensorRT plugin 资源回收处理错误的问题。(#38374)
- 修复 deformable_conv op 在 TensorRT plugin 序列化错误问题。(#38057)
- 适配 TensorRT 8.0 新的构建引擎和系列化 API。(#36769)
- 修复 Flatten2MatmulFusePass、Squeeze2MatmulFusePass、Reshape2MatmulFusePass 没有生效问题。(#37644)
- 修复 TensorRT 输入数据在上时报错的问题。(#37427)
- 增加输入维度错误时的报错信息。(#38962)
- 修复 EmbEltwiseLayernorm 输出类型错误的问题。(#40015)
- 删除 conv_affine_channel_fuse_pass 以及对应的单元测试。(#39817)

- 修复 adaptive_pool2d pass 错误替换 pool 属性的问题。(#39600)
- 修复 shuffle_channel_detect_pass 错误生成 shuffle_channel op 的问题。(#39242)
- 修复 transpose 参数错误。(#39006)
- 修复 nearest_interp_v2 输入 scale 维度小于 1 时崩溃的问题。(#38725)
- 修复 prelu 在 dynamic shape 时不支持一维输入的问题。(#39389)
- 修复 slice 的 special_slice_plugin 的核函数计算错误的问题。(#39875)
- 暂时禁用 skip_layernorm 变长下的 int8 分支，防止精度下降。(#39991)
- 修复关于支持 preln_ernie 模型的一些 bug。(#39733)
- 修复 slice 在 ERNIE 中 threads 可能超过限制的 bug，修复 spacial_slice 误触的 bug。(#39096)
- 修复 elementwise 在维度相同时不支持广播的问题。(#37908)
- 修复 nearest_interp op 当 align_corners 为 True 时，TensorRT layer 的结果和原生 op 的结果有 diff，底层实现不一样。(#37525)
- 修复 qkv_plugin: 核函数计算错误。(#37096)
- 修复动态量化的推理 pass 的问题。(#35879)
- 当 Tensor 请求的内存容量低于已分配的 size 时直接复用。(#37880)
- 修复 ERNIE 定长模型开启 TensorRT 出现的 hang 问题。(#37839)
- 修复 TensorRT int8 时缺失 dynamic range 信息崩溃问题。(#36900)
- 修复 slice 反序列化代码问题。(#36588)
- 修复 yolo box 计算公式错误问题。(#36240)
- 修复老版本模型在使用新版本 roi_align 时崩溃问题。(#38788) 外部开发者
- 修复 softmax 在 python 和 C++ 上性能差异较大的问题。(#37130)
- 修复 matmul 在静态 shape 2 维输入和动态 shape 3 维输入情况下推理失败问题。(#36849)
- 修复 reshape_transpose_matmul_mkldnn_fuse_pass 对 shape 处理不当问题。(#36731)
- 修复输入为 2 维，但 TensorRT 获取到 4 维的问题。(#36614)
- 修复 interpolate_v2 MKLDNN 算子在 scale 属性为空时报错问题。(#36623)
- 修复 recurrent 算子在多线程场景性能差问题。(#36052)
- 移除 relu、sigmoid、tanh、relu6、batch_norm、clip、concat、gelu、hard_sigmoid、prelu、softmax、split、swish 对 TensorRT 2 维输入的限制。(#37097)
- 修复 reshape op 使用 TensorRT 推理。(#41090)
- 修复 matmul 相关 pass，兼容 matmul_v2。(#36424)
- 开启 TensorRT 时，conv2d 算子中 padding 方式支持 VALID 及 SAME 属性。(#38999)

- 修复 MKLDNN 多输入算子量化问题。(#39593, #39346, #40717)
- 修复 MKLDNN 量化场景下 conv+activation 的 scale 错误问题。(#38331)
- 修复 MKLDNN 无参数算子量化中，根据后续算子量化情况不同需做不同处理的问题。(#39342)
- 修复 MKLDNN cpu_bfloat16_placement_pass 中的数据类型相关问题。(#38702)
- 修复 MKLDNN bfloat16 推理中 split 算子执行问题。(#39548)
- 修复 MKLDNN matmul_v2 算子不支持 6 维问题。(#36342, #38665)
- 修复 MKLDNN matmul_v2_transpose_reshape 中的 MKLDNN DeviceContext 错误问题。(#38554)
- 修复分割模型在 MKLDNN 推理场景计算结果错误问题。(#37310)
- 修复 MKLDNN bfloat16 placement 算子列表并添加缺失算子。(#36291)
- 修复 MKLDNN 算子的格式问题，包括：FC、conv_transpose、6 维 Tensor 报错问题、conv 对 NHWC 输入的输出 format 错误问题。(#38890, #37344, #37175, #38553, #40049, #39097)
- 修复 MKLDNN 多线程推理场景因 cache 机制报错问题。(#36290, #35884)
- 修复 MKLDNN 因 matmul 及 FC 引起的量化模型精度异常问题。(#38023, #37618)
- 修复 MKLDNN 量化转换脚本因 pass 缺少引起的量化模型精度异常问题。(#37619, #40542, #38912)
- 修复 MKLDNN 开启量 op 因为数据类型不匹配崩溃的问题。(#38133)
- 修复 MKLDNN 某些 op 修改 layout 后需要改回原 layout 的问题。(#39422)
- 修复针对昇腾 910 推理场景下，由于未释放 GIL 锁，导致与昇腾软件栈冲突，python API 下报错的问题。(#38605)

8.5 5. 环境适配

8.5.1 编译安装

- 从 2.3.0 版本开始，飞桨对框架支持的 GPU 架构种类进行了调整和升级。(更多请参考：[飞桨支持的 GPU 架构](#))

备注：

- PIP 源安装是指用 pip install paddlepaddle 或 pip install paddlepaddle-gpu 从 PIP 官网下载安装包及依赖库的安装方式，支持架构种类少，安装包更轻量，下载源来自国外（相比 bos 源支持架构种类精简，安装包更轻量，只提供一种 CUDA 版本的安装包）。
 - 2.3 版本之前，飞桨 PIP 源安装包（CUDA10.2）支持的 GPU 架构为：3.5, 5.0, 5.2, 6.0, 6.1, 7.0, 7.5。
 - 2.3 版本之后，飞桨 PIP 源安装包（CUDA11.0）支持的 GPU 架构为：6.0, 6.1, 7.0, 7.5, 8.0
- 飞桨官网 bos 源是指从飞桨官网下载安装包及依赖库的安装方式，支持的 GPU 架构更多，下载源来自国内，速度较快。（相比 PIP 源支持架构种类多，提供多个 CUDA 版本的安装包）：

- 2.3 版本之前，飞桨官网 bos 源安装包支持的 GPU 架构：
 - * CUDA10 : 3.5, 5.0, 5.2, 6.0, 6.1, 7.0, 7.5;
 - * CUDA11 : 5.2, 6.0, 6.1, 7.0, 7.5, 8.0。
- 2.3 版本之后，飞桨官网 bos 源安装包支持的 GPU 架构
 - * CUDA10 : 3.5, 5.0, 5.2, 6.0, 6.1, 7.0, 7.5;
 - * CUDA11 : 3.5, 5.0, 6.0, 6.1, 7.0, 7.5, 8.0。
- 支持 Python 3.10，修复 Windows 下某些 PythonC API 变化导致的编译 bug。 (#41180)
- Windows 平台支持 Visual Studio 2019 编译。 (#38719)
- 消除 Windows 平台编译时出现的各种 warning。 (#38034, #37890, #37442, #37439, #36857)
- 修复底层数据结构升级引入的 jetson 编译问题。 (#39669, #39441)

8.5.2 新硬件适配

- 自定义新硬件接入：提供一种插件式扩展 PaddlePaddle 硬件后端的方式。通过该功能，开发者无需为特定硬件修改 PaddlePaddle 代码，只需实现标准接口，并编译成动态链接库，则可作为插件供 PaddlePaddle 调用。降低为 PaddlePaddle 添加新硬件后端的开发难度。当前支持自定义 Runtime 接入和自定义 Kernel 接入。
- 华为 NPU 芯片（Ascend910）训练/推理支持，支持 ResNet50、YoloV3、BERT、Transformer 等多个模型，支持静态图与混合精度训练，支持单卡、单机、多机分布式训练。
- Graphcore IPU 芯片（包括 IPU Mk2 GC200 和 Bow IPU）训练/推理支持，支持 ResNet50、BERT 等模型，支持静态图训练，支持单芯片、单机、多机分布式训练。
- 寒武纪 MLU 芯片（MLU370x4）训练/推理支持，支持 ResNet50 等模型，支持静态图 + 动态图训练，支持混合精度训练，支持单卡、单机、多机分布式训练。
- 昆仑芯 2 代芯片（昆仑芯 AI 加速卡 R200、R300）训练/推理支持，支持 ResNet50、YoloV3、OCR-DB、SSD、MobilnetV3、UNet、BERT、Transformer、GPT-2、Wide&Deep、DeepFM，支持静态图 + 动态图训练，支持混合精度训练，支持单机单卡、单机多卡训练。

8.6 Thanks to our Contributors

This release contains contributions from the project core team as well as :

Adam Osewski, Allen Guo, arlesniak, chenenquan, chenyanlann, fengkuangxiaxia, fuqianya, fwenguang, guguguzi, helen88, houj04, Jacek Czaja, jakpiase, jianghaicheng, joanna.wozna.intel, joeqiao12, Leo Chen, Leo Guo, Li-fAngyU, lidanqing, Liyulingyue, Matsumoto GAO, maxhuiy, Ming-Xu Huang, Nyakku Shigure, piotrekobi, piotrekobiIntel, Qing-shuChen, qipengh, Skr Bang, Sylwester Fraczek, Sławomir Siwek, taixiurong, tanzhipeng, Tomasz Socha, TTerror, Webbley, yaozhixin, ykkk2333, yujun, Zhangjingyu06, zhangxiaoci, zhangyikun02, zhangyk0314, zlsh80826, zn, Zuza