TensorFlow 内核剖析

TensorFlow Internals

刘光聪 著

献给我的女儿刘楚溪

前言

本书定位

这是一本剖析 TensorFlow 内核工作原理的书籍，并非讲述如何使用 TensorFlow 构建机器学习模型，也不会讲述应用 TensorFlow 的最佳实践。本书将通过剖析 TensorFlow 源代码的方式，揭示 TensorFlow 的系统架构、领域模型、工作原理、及其实现模式等相关内容，以便揭示内在的知识。

面向的读者

本书假设读者已经了解机器学习相关基本概念与理论，了解机器学习相关的基本方法论；同时，假设读者熟悉 Python, C++ 等程序设计语言。

本书适合于渴望深入了解 TensorFlow 内核设计，期望改善 TensorFlow 系统设计和性能优化，及其探究 TensorFlow 关键技术的设计和实现的系统架构师、AI 算法工程师、和 AI 软件工程师。

阅读方式

初次阅读本书，推荐循序渐进的阅读方式；对于高级用户，可以选择感兴趣的章节阅读。首次使用 TensorFlow 时，推荐从源代码完整地构建一次 TensorFlow，以便了解系统的构建方式，及其理顺所依赖的基本组件库。

另外，推荐使用 TensorFlow 亲自实践一些具体应用，以便加深对 TensorFlow 系统行为的认识和理解，熟悉常见 API 的使用方法和工作原理。强烈推荐阅读本书的同时，阅读 TensorFlow 关键代码；关于阅读代码的最佳实践，请查阅本书附录 A 的内容。

版本说明

本书写作时，TensorFlow 稳定发布版本为 1.2。不排除本书讲解的部分 API 将来被废弃，也不保证某些系统实现在未来版本发生变化，甚至被删除。

同时，为了更直接的阐述问题的本质，书中部分代码做了局部的重构；删除了部分异常处理分支，或日志打印，甚至是某些可选参数列表。但是，这样的局部重构，不会影响读者理解系统的主要行为特征，更有利于读者掌握系统的工作原理。

同时，为了简化计算图的表达，本书中的计算图并非来自 TensorBoard，而是采用简化了的，等价的图结构。同样地，简化了的图结构，也不会降低读者对真实图结构的认识和理解。

在线帮助

为了更好地与读者交流，已在 Github 上建立了勘误表，及其相关补充说明。由于个人经验与能力有限，在有限的时间内难免犯错。如果读者在阅读过程中，如果发现相关错误，

请帮忙提交 Pull Request，避免他人掉入相同的陷阱之中，让知识分享变得更加通畅，更加轻松，我将不甚感激。

同时，欢迎关注我的简书。我将持续更新相关的文章，与更多的朋友一起学习和进步。

1. Github: https://github.com/horance-liu/tensorflow-internals-errors
2. 简书：http://www.jianshu.com/u/49d1f3b7049e

致谢

感谢我的太太刘梅红，在工作之余完成对本书的审校，并提出了诸多修改的意见。

目录

第 I 部分

基础知识

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

- Martin Flower

1 介绍

TensorFlow 是一个使用**数据流图** (Dataflow Graph) 表达数值计算的开源软件库。它使用**节点**表示抽象的数学计算，并使用 OP 表达计算的逻辑；而**边**表示节点间传递的数据流，并使用 Tensor 表达数据的表示。[?]。数据流图是一种**有向无环图** (DAG)，当图中的 OP 按照特定的拓扑排序依次被执行时，Tensor 在图中流动形成数据流，TensorFlow 因此而得名。

在分布式运行时，数据流图的被分裂为多个子图，并被有效地部署到集群中的多个机器上并发执行。在一个机器内，注册的子图被二次分裂为更小的子图，它们被部署在**本地设备集**上并发执行。TensorFlow 支持多种异构设备的分布式计算，包括 CPU, GPU, ASIC。TensorFlow 跨平台的卓越表现，使得它能够灵活地部署在各种计算平台上，包括台式机、服务器、移动终端。

TensorFlow 最初由 Google Brain 的研究员和工程师们开发出来，用于开展机器学习和深度神经网络的研究，包括语言识别、计算机视觉、自然语言理解、机器人、信息检索。但是，TensorFlow 系统架构的通用性和灵活性，使其广泛地用于其他科学领域的数值计算。

1.1 前世今生

Google Brain 项目始于 2011 年，用于研究超大规模的深度神经网络。在项目早期阶段，Google Brain 构建了第一代分布式深度学习框架 DistBelief，并在 Google 内部的产品中得到了大量的应用。

基于 DistBelief 的经验，Google Brain 对深度学习训练和推理的需求，及其深度学习框架的系统行为和属性，有了更全面更深刻地理解，并于 2015.11 重磅推出第二代分布式深度学习框架 TensorFlow。TensorFlow 作为 DistBelief 的后继者，革命性地对既有系统架构做了全新设计和实现，TensorFlow 一经发布，便在深度学习领域一鸣惊人，在社区中形成了巨大的影响力。

DistBelief

DistBelief 是一个用于训练大规模神经网络的的分布式系统，是 Google 第一代分布式机器学习框架。自 2011 年以来，在 Google 内部大量使用 DistBelief 训练大规模的神经网络，并广泛地用于机器学习和深度学习领域的研究和应用，包括非监督学习、语言表示、图像分类、目标检测、视频分类、语音识别、序列预测、行人检测、强化学习等。

编程模型

DistBelief 的编程模型是基于**层**的 DAG 图。层可以看做是一种组合多个运算操作符的复合运算符，它完成特定的计算任务。例如，全连接层完成 $f(W^T x + b)$ 的复合计算，包括输入与权重的矩阵乘法，随后再与偏置相加，最后在线性加权值的基础上应用激活函数，实施非线性变换。

架构

DistBelief 使用**参数服务器** (Parameter Server, 常称为 PS) 的系统架构，训练作业包括两个分离的进程：无状态的 Worker 进程，用于模型的训练；有状态的 PS 进程，用于维护模型的参数。如图**??**（第**??**页）所示，在分布式训练过程中，各个**模型副本**异步地从 PS 上拉取训练参数 w，当完成一步迭代运算后，推送参数的梯度 Δw 到 PS 上去，并完成参数的更新。

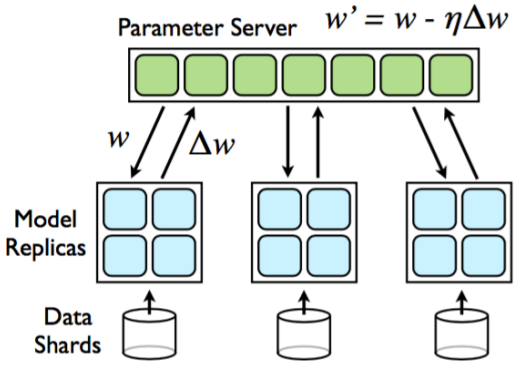


图 1-1 DistBelief: Parameter Server 架构

缺陷

但是，对于深度学习领域的高级用户，DistBelief 的编程模型，及其基于 PS 的系统架构，缺乏足够的灵活性和可扩展性。

1. 优化算法：添加新的优化算法，必须修改 PS 的实现；`get()`，`put()` 的抽象方法，对某些优化算法并不高效。

2. 训练算法：支持非前馈的神经网络面临巨大的挑战性；例如，包含循环的 RNN，交替训练的对抗网络，及其损失函数由分离的代理完成的增强学习模型。

3. 加速设备：DistBelief 设计之初仅支持多核 CPU，并不支持多卡的 GPU，遗留的系统架构对支持新的计算设备缺乏良好的可扩展性。

TensorFlow

DistBelief 遗留的架构和设计，不再满足深度学习与日俱增的需求变化，Google 毅然放弃了既有的 DistBelief 实现，并决定在其基础上做全新的系统架构设计，TensorFlow 应运而生，开创了深度学习领域的新纪元。

编程模型

TensorFlow 使用数据流图表达计算过程和共享状态，使用节点表示抽象计算，使用边表示数据流。如图**??**（第**??**页）所示，展示了 MNIST 手写识别应用的数据流图。在该模型中，前向子图使用了 2 层全连接网络，分别为 ReLU 层和 Softmax 层。随后，使用 SGD 的优化算法，构建了与前向子图对应的反向子图，用于计算训练参数的梯度。最后，根据参数更新法则，构造训练参数的更新子图，完成训练参数的迭代更新。

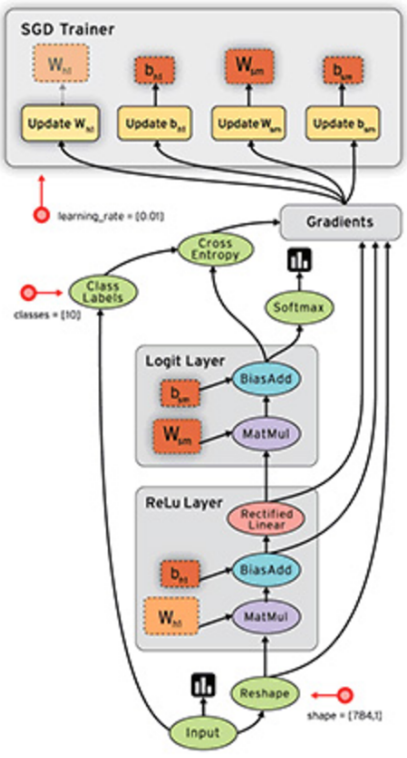


图 1-2 TensorFlow: 数据流图

设计原则

TensorFlow 的系统架构遵循了一些基本的设计原则，用于指导 TensorFlow 的系统实现。

1. 延迟计算：图的构造与执行分离，并推迟计算图的执行过程；

2. 原子 OP：OP 是最小的抽象计算单元，支持构造复杂的网络模型；

3. 抽象设备：支持 CPU, GPU, ASIC 多种异构计算设备类型；

4. 抽象任务：基于任务的 PS，对新的优化算法和网络模型具有良好的可扩展性。

优势

相对于其他机器学习框架，TensorFlow 具有如下方面的优势。

1. 高性能：TensorFlow 升级至 1.0 版本性能提升显著，单机多卡 (8 卡 GPU) 环境中，Inception v3 的训练实现了 7.3 倍的加速比；在分布式多机多卡 (64 卡 GPU) 环境中，Inception v3 的训练实现了 58 倍的加速比；

2. 跨平台：支持多 CPU/GPU/ASIC 多种异构设备的运算；支持台式机，服务器，移动设备等多种计算平台；支持 Windows，Linux，MacOS 等多种操作系统；

3. 分布式：支持本地和分布式的模型训练和推理；

4. 多语言：支持 Python, C++, Java, Go 等多种程序设计语言；

5. 通用性：支持各种复杂的网络模型的设计和实现，包括非前馈型神经网络；

6. 可扩展：支持 OP 扩展，Kernel 扩展，Device 扩展，通信协议的扩展；

7. 可视化：使用 TensorBoard 可视化整个训练过程，极大地降低了 TensorFlow 的调试过程；

8. 自动微分：TensorFlow 自动构造反向的计算子图，完成训练参数的梯度计算；

9. 工作流：TensorFlow 与 TensorFlow Serving 无缝集成，支持模型的训练、导入、导出、发布一站式的工作流，并自动实现模型的热更新和版本管理。

1.2 社区发展

TensorFlow 是目前炙手可热的深度学习框架。自开源以来，TensorFlow 社区相当活跃。在 Github 上收获了来自众多的非 Google 员工的数万次代码提交，并且每周拥有近百个 Issue 被提交。在 Stack Overflow 上也拥有上万个关于 TensorFlow 的问题被提问和回答。在各种技术大会上，TensorFlow 也是一颗闪亮的明星，得到众多开发者的青睐。

开源

2015.11，Google Research 发布文章：TensorFlow: Google's latest machine learning system, open sourced for everyone，正式宣布新一代机器学习系统 TensorFlow 开源。随后，TensorFlow 在 Github 上代码仓库短时间内获得了大量的 Star 和 Fork。如图**??**（第**??**页）所示，TensorFlow 的社区活跃度已远远超过其他竞争对手，成为目前最为流行的深度学习框架。

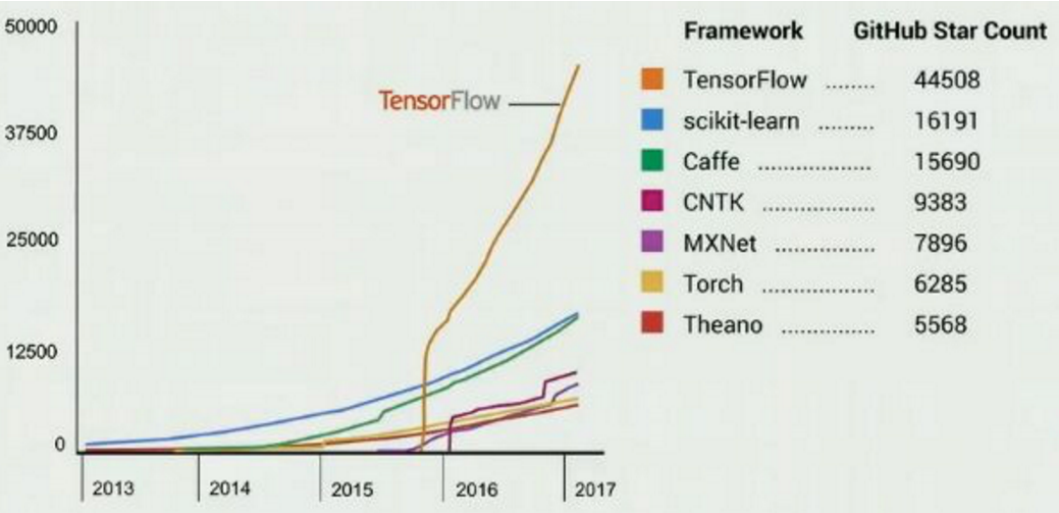


图 1-3 TensorFlow：社区活跃度

毫无疑问，TensorFlow 的开源对学术界和工业界产生了巨大的影响，它极大地降低了深度学习在各个行业中应用的难度。众多的学者、工程师、企业、组织纷纷地投入到了 TensorFlow 社区，并一起完善和改进 TensorFlow，推动其不断地向前演进和发展。

里程碑

TensorFlow 自 2015.11 开源以来，平均一个多月发布一个版本。如图**??**（第**??**页）所示，展示了 TensorFlow 几个重要特性的发布时间。

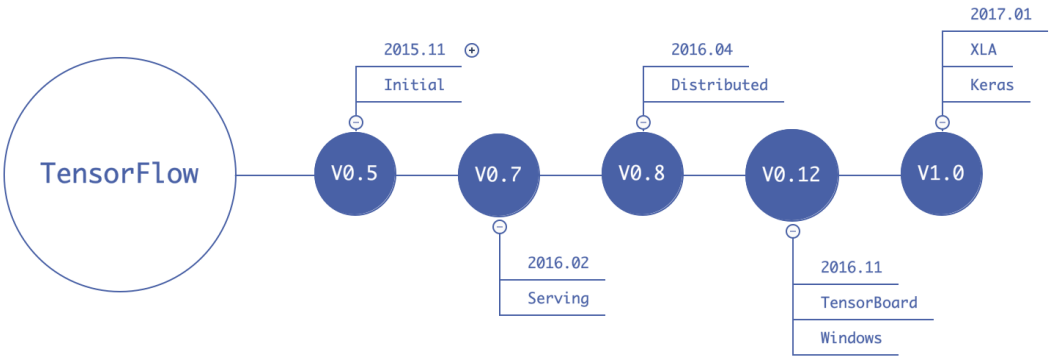


图 1-4 TensorFlow 重要里程碑

工业应用

TensorFlow 自开源发展两年以来，在生产环境中被大量应用使用。在医疗方面，帮助医生预测皮肤癌发生的概率；在音乐、绘画领域，帮助人类更好地理解艺术；在移动端，多款移动设备搭载 TensorFlow 训练的机器学习模型，用于翻译工作。TensorFlow 在 Google 内部应用的增长也十分迅速，多个重量级产品都有相关应用，包括：Google Search, Google

Gmail, Google Translate, Google Maps 等，如图**??**（第**??**页）所示。

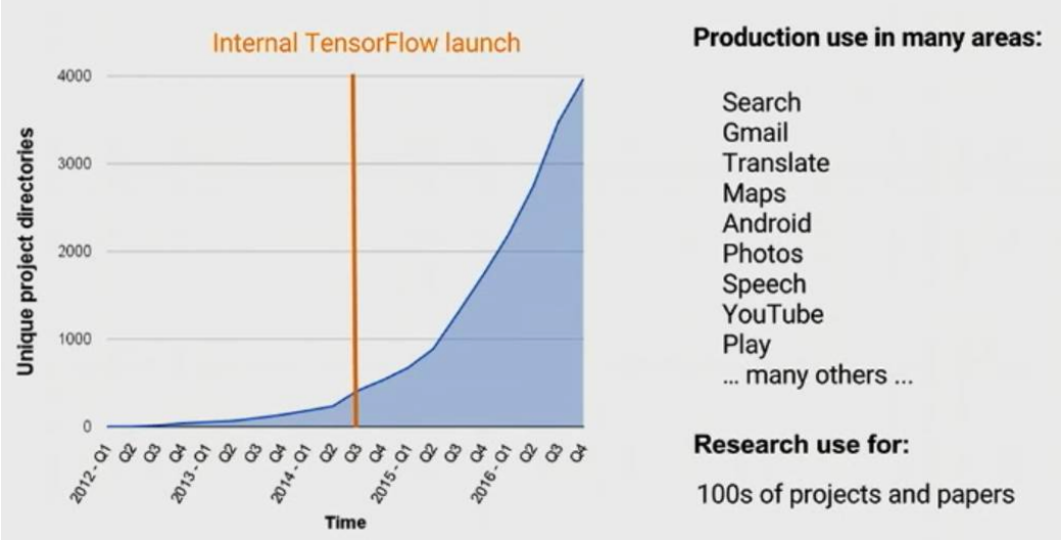


图 1-5 TensorFlow：在 Google 内部使用情况

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

- Martin Flower

2 编程环境

为了实现 TensorFlow 的快速入门，本章将介绍 TensorFlow 的编程环境，包括代码结构，工程构建，以便对 TensorFlow 的系统架构建立基本的感性认识。

2.1 代码结构

克隆源码

首先，从 Github 上克隆 TensorFlow 的源代码。

```
$ git clone git@github.com:tensorflow/tensorflow.git
```

然后，切换到最新的稳定分支上。例如，`r1.4` 分支。

```
$ cd tensorflow
$ git checkout r1.4
```

源码结构

运行如下命令，打印出 TensorFlow 源码的组织结构。

```
$ tree -d -L 1 ./tensorflow
```

其中，本书将重点关注 `core`，`python` 组件，部分涉及 `c`，`cc`，`stream_executor` 组件。

示例代码 2-1 TensorFlow 源码结构

```
./tensorflow
├── c
├── cc
├── compiler
├── contrib
├── core
├── docs_src
├── examples
├── g3doc
└── go
```

```
├── java
├── python
├── stream_executor
├── tools
└── user_ops
```

截止当前最新发布的 1.4 版本，TensorFlow 代码库拥有大约 100 万代码。其中，包括 53 万行 C/C++ 代码，37 万行 Python 代码，而且代码规模在不断膨胀之中。其中，Python 提供的 API 是最完善的；相比之下，其他编程语言的 API 尚未成熟，甚至处于起步阶段。

示例代码 2-2 TensorFlow 代码统计

```
-------------------------------------------------------------
Language              files    blank    comment    code
-------------------------------------------------------------
C++                    2238    77610      68275    443099
Python                 1881    92018     151807    369399
C/C++ Header           1175    27392      46215     86691
Markdown                218     8859          2     30925
CMake                    50     2183        986     16398
Go                       28     1779      13290     15003
Java                     72     1789       3111      7779
Bourne Shell            103     1487       3105      6074
Protocol Buffers         87     1313       3339      3452
Objective C++             9      227        181      1201
C                         8      157        130       941
make                      4      105        136       612
XML                      25      135        265       315
Groovy                    3       46         74       246
Maven                     5       21          4       245
DOS Batch                 9       30          0       143
Dockerfile                7       41         69       133
Perl                      2       29         38       130
Bourne Again Shell        3       24         63       111
JSON                      3        0          0        23
Objective C               1       10         13        21
YAML                      1        3         24        15
-------------------------------------------------------------
SUM:                   5932   215258     291127    982956
-------------------------------------------------------------
```

Core

内核的源码结构如下所示，主要包括平台，实用函数库，基础框架，Protobuf 定义，本地运行时，分布式运行时，图操作，OP 定义，以及 Kernel 实现等组成，这是本书重点剖析的组件之一，将重点挖掘基础框架中隐藏的领域模型，追踪整个运行时环境的生命周期和图操作的详细过程，并揭示常见 OP 的 Kernel 实现原理和运行机制。

示例代码 2-3 Core 源码结构

```
./tensorflow/core
├── common_runtime
├── debug
├── distributed_runtime
├── example
├── framework
└── graph
```

```
├── grappler
├── kernels
├── lib
├── ops
├── platform
├── profiler
├── protobuf
├── public
├── user_ops
└── util
```

其中，core 主要由 C++ 实现，大约拥有 26 万行代码。

示例代码 2-4 Core 代码统计

```
-------------------------------------------------------------
Language              files    blank   comment      code
-------------------------------------------------------------
C++                    1368    44791     38968    259289
C/C++ Header            653    15040     24474     50506
Protocol Buffers         57      736      2371      1806
Markdown                 11      327         0      1285
JSON                      2        0         0        18
-------------------------------------------------------------
SUM:                   2091    60894     65813    312904
-------------------------------------------------------------
```

Python

Python 定义和实现了 TensorFlow 的编程模型，并对外开放 API 供程序员使用，其源码结构如下所示，这也是本书重点剖析的部分。

示例代码 2-5 Python 源码结构

```
./tensorflow/python
├── client
├── debug
├── estimator
├── feature_column
├── framework
├── grappler
├── kernel_tests
├── layers
├── lib
├── ops
├── platform
├── profiler
├── saved_model
├── summary
├── tools
├── training
├── user_ops
└── util
```

其中，该组件由 Python 实现，大约有 18 万行代码。

示例代码 2-6 Python 代码统计

```
-------------------------------------------------------
Language             files     blank   comment      code
-------------------------------------------------------
Python                 714     45769     69407    179565
C++                     20       496       506      3658
C/C++ Header            15       207       387       363
Markdown                 4        48         0       200
Protocol Buffers         3        16        10        71
Bourne Shell             1        13        28        68
-------------------------------------------------------
SUM:                   757     46549     70338    183925
-------------------------------------------------------
```

Contrib

contrib 是第三方贡献的编程库，它也是 TensorFlow 标准化之前的实验性编程接口，犹如 Boost 社区与 C++ 标准之间的关系。当 contrib 的接口成熟后，便会被 TensorFlow 标准化，并从 contrib 中搬迁至 core, python 中，并正式对外发布。

示例代码 2-7 Contrib 源码结构

```
./tensorflow/contrib
├── android
├── batching
├── bayesflow
├── benchmark_tools
├── boosted_trees
├── cloud
├── cluster_resolver
├── cmake
├── compiler
├── copy_graph
├── crf
├── cudnn_rnn
├── data
├── decision_trees
├── deprecated
├── distributions
├── eager
├── factorization
├── ffmpeg
├── framework
├── fused_conv
├── gdr
├── graph_editor
├── grid_rnn
├── hooks
├── hvx
├── image
├── imperative
├── input_pipeline
├── integrate
├── keras
├── kernel_methods
├── labeled_tensor
├── layers
├── learn
├── legacy_seq2seq
├── linalg
├── linear_optimizer
├── lookup
```

```
    losses
    makefile
    memory_stats
    meta_graph_transform
    metrics
    mpi
    nccl
    ndlstm
    nearest_neighbor
    nn
    opt
    pi_examples
    predictor
    quantization
    reduce_slice_ops
    remote_fused_graph
    resampler
    rnn
    saved_model
    seq2seq
    session_bundle
    signal
    slim
    solvers
    sparsemax
    specs
    staging
    stat_summarizer
    stateless
    tensor_forest
    tensorboard
    testing
    text
    tfprof
    timeseries
    tpu
    training
    util
    verbs
    xla_tf_graph
```

由于 TensorFlow 社区相当活跃，`contrib` 的变更相当频繁，截止 1.4 版本，大约有 23 万行代码，主要由 Python 设计和实现的编程接口，部分运行时由 C++ 实现。

示例代码 2-8 Contrib 代码统计

```
-------------------------------------------------------------
Language            files      blank    comment       code
-------------------------------------------------------------
Python               1007      41436      75096     170355
C++                   201       5500       5313      32944
CMake                  48       2172        955      16358
C/C++ Header           99       1875       2867       6583
Markdown               46       1108          0       3485
Bourne Shell           18        232        386       1272
C                       7        151        118        931
Protocol Buffers       20        224        454        680
make                    4        105        136        612
Java                    2         77        209        335
Groovy                  1         10         20         75
Bourne Again Shell      1          6         15         59
Dockerfile              1          2          1         14
XML                     2          3          0          9
-------------------------------------------------------------
SUM:                 1457      52901      85570     233712
-------------------------------------------------------------
```

StreamExecutor

StreamExecutor 是 Google 另一个开源组件库，它提供了主机端 (host-side) 的编程模型和运行时环境，实现了 CUDA 和 OpenCL 的统一封装。使得在主机端的代码中，可以将 Kernel 函数无缝地部署在 `CUDA` 或 `OpenCL` 的计算设备上执行。

目前，StreamExecutor 被大量应用于 Google 内部 GPGPU 应用程序的运行时。其中，TensorFlow 运行时也包含了一个 StreamExecutor 的快照版本，用于封装 CUDA 和 `OpenCL` 的运行时。本书将简单介绍 CUDA 的编程模型和线程模型，并详细介绍 StreamExecutor 的系统架构与工作原理，揭示 Kernel 函数的实现模式和习惯用法。

示例代码 2-9 StreamExecutor 源码结构

```
./tensorflow/stream_executor
├── cuda
├── host
├── lib
└── platform
```

其中，StreamExecutor 使用 `C++` 实现，大约有 2.5 万行代码。

示例代码 2-10 StreamExecutor 代码统计

```
-------------------------------------------------------
Language            files     blank   comment      code
-------------------------------------------------------
C++                    43      2440      1196     16577
C/C++ Header           81      2322      5080      8625
-------------------------------------------------------
SUM:                  124      4762      6276     25202
-------------------------------------------------------
```

Compiler

众所周知，灵活性是 TensorFlow 最重要的设计目标和核心优势，因此 TensorFlow 的系统架构具有良好的可扩展性。TensorFlow 可用于定义任意图结构，并使用异构的计算设备有效地执行。但是，熊掌与鱼翅不可兼得，当低级 OP 组合为计算子图时，并期望在 GPU 上有效执行时，运行时将启动更多的 Kernel 的运算。

因此，TensorFlow 分解和组合 OP 的方法，在运行时并不能保证以最有效的方式运行。此时，XLA 技术孕育而生，它使用 JIT 编译技术来分析运行时的计算图，它将多个 OP 融合在一起，并生成更高效的本地机器代码，提升计算图的执行效率。

示例代码 2-11 Compiler 源码结构

```
./tensorflow/compiler
├── aot
└── jit
```

```
    plugin
    tests
    tf2xla
    xla
```

XLA 技术目前处于初级的研发阶段，是 TensorFlow 社区较为活跃研究方向，截止目前代码规模大约为 12.5 万行，主要使用 C++ 实现。

示例代码 2-12 Compiler 代码统计

```
-------------------------------------------------------------
Language             files     blank   comment      code
-------------------------------------------------------------
C++                    455     19010     18334    102537
C/C++ Header           250      5939     10323     15510
Python                  37      1255      1416      6446
Protocol Buffers         5       312       501       781
Markdown                 2         0         0         3
-------------------------------------------------------------
SUM:                   749     26516     30574    125277
-------------------------------------------------------------
```

2.2 工程构建

在开始之前，尝试 TensorFlow 源码的构建过程，了解 TensorFlow 的基本构建方式、工具，及其依赖的组件库、第三方工具包，对于理解 TensorFlow 工作原理具有很大的帮助。但是，因篇幅受限，本章仅以 Mac OS 系统为例，讲述 TensorFlow 的源码编译、安装、及其验证过程。其他操作系统，请查阅 TensorFlow 发布的官方文档。

环境准备

TensorFlow 的前端是一个支持多语言的编程接口。因此，编译 TensorFlow 源代码之前，需要事先安装相关的编译器、解释器、及其运行时环境。例如，使用 Python 作为编程接口，需要事先安装 Python 解释器。其次，在构建系统之前，也需要事先安装 GCC 或 Clang 等 C++ 编译器，用于编译后端系统实现。因为 TensorFlow 使用 C++11 语法实现，所以要保证安装 C++ 编译器要支持 C++11。另外，TensorFlow 使用 Bazel 的构建工具，可以将其视为更高抽象的 Make 工具。不幸的是，Bazel 使用 Java8 实现，其依赖于 JDK。因此在安装 Bazel 之前，还得需要事先安装 1.8 及以上版本的 JDK。

安装 JDK

推荐从 Oracle 官网上下载 1.8 版本的 JDK，然后创建相关的环境变量，并将其添加到 `~/.bashrc` 的配置文件中。

```
$ echo 'export JAVA_HOME=$(/usr/libexec/java_home)' >> ~/.bashrc
$ echo 'export PATH="$JAVA_HOME/bin:$PATH"' >> ~/.bashrc
```

安装 Bazel

在 Mac OS 上，可以使用 brew 安装 Bazel。

```
$ brew install bazel
```

如果系统未安装 brew，可以执行如下命令先安装 brew。当然，安装 brew 需要事先安装 Ruby 解释器，在此不再冗述。

```
$ ruby -e "$(curl -fsSL \
  https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

安装 Swig

TensorFlow 使用 Swig 构建多语言的编程环境，自动生成相关编程语言的包装器。因此，在构建之前需要事先安装 Swig 的工具包。

```
$ brew install swig
```

安装 Python 依赖包

使用 pip 安装 TensorFlow 所依赖的 Python 包。

```
$ sudo pip install six numpy wheel autograd
```

如果系统未安装 pip，则可以使用 brew 预先安装 pip。

```
$ brew install pip
```

安装 CUDA 工具包

当系统安装了 CUDA 计算兼容性大于或等于 3.0 的 GPU 卡时，则需要安装 CUDA 工具包，及其 cuDNN，实现 TensorFlow 运行时的 GPU 加速。推荐从 NVIDIA 官网上下载

CUDA Toolkit 8 及以上版本，并安装到系统中，配置相关环境变量。

```
$ echo 'export CUDA_HOME=/usr/local/cuda' >> ~/.bashrc
$ echo 'export LD_LIBRARY_PATH=$CUDA_HOME/lib:$LD_LIBRARY_PATH' >> ~/.bashrc
```

然后，再下载 cuDNN 5.1 及以上版本，并将其解压至 `CUDA_HOME` 相关系统目录中。

```
$ sudo tar -xvf cudnn-8.0-macos-x64-v5.1.tgz -C /usr/local
```

配置

至此，编译环境一切就绪，执行`./configure` 配置 TensorFlow 的编译环境了。当系统支持 GPU，则需要配置相关的 CUDA/cuDNN 编译环境。

```
$ ./configure
```

构建

当配置成功后，使用 Bazel 启动 TensorFlow 的编译。在编译启动之前，会尝试从代码仓库中下载相关依赖库的源代码，包括 gRPC, Protobuf, Eigen 等，并自动完成编译。

```
$ bazel build --config=opt //tensorflow/tools/pip_package:build_pip_package
```

当支持 GPU 计算时，添加`--config=cuda` 编译选项。

```
$ bazel build -c opt --config=cuda \
  //tensorflow/tools/pip_package:build_pip_package
```

编译成功后，便可以构建 TensorFlow 的 Wheel 包。

```
$ bazel-bin/tensorflow/tools/pip_package/build_pip_package \
  /tmp/tensorflow_pkg
```

安装

当 Wheel 包构建成功后，使用 pip 安装 TensorFlow 到系统中。

```
$ sudo pip install /tmp/tensorflow_pkg/tensorflow-1.4.0-py2-none-any.whl
```

验证

启动 Python 解释器，验证 TensorFlow 安装是否成功。

```
$ python
>>> import tensorflow as tf
>>> hello = tf.constant('Hello, TensorFlow!')
>>> sess = tf.Session()
>>> print(sess.run(hello))
Hello, TensorFlow!
```

IDE

在阅读代码之前，选择一个适宜的 IDE 可以改善代码阅读的质量和速度。推荐使用 Eclipse CDT 阅读 C++ 代码，安装 PyDev 的插件阅读 Python 代码。同时，也推荐 JetBrains 出品的 Clion 阅读 C++，PyCharm 阅读 Python。但是，当阅读 C++ 代码时，需要配置 TensorFlow, CUDA, Eigen3 头文件的搜索目录，并添加相关预定义的宏，以便 IDE 正确解析代码中的符号。本章以 Eclipse CDT 为例讲述相关配置方法。

创建 Eclipse 工程

创建一个 Eclipse C++ 工程，如图**??**（第**??**页）所示。确定唯一的项目名称，手动地指定 TensorFlow 源代码的根目录，并选择 Makefile 的空工程。然后，按照 Properties > C/C++ General > Paths and Symbols > Includes 配置头文件的搜索目录。

配置项	目录
TensorFlow	/usr/local/lib/python2.7/site-packages/tensorflow/include
CUDA	/usr/local/cuda/include

表 2.1 头文件搜索目录

配置 Eigen

不幸的是，Eigen 对外公开的头文件缺少.h 的后缀名，CDT 无法解析相关的符号。请参阅http://eigen.tuxfamily.org/index.php?title=IDEs相关说明，按照 Preferences > C/C++ > Coding Style > Organize Includes > Header Substitution 导入 eigen-header-substitution.xml 文件，如图**??**（第**??**页）所示。

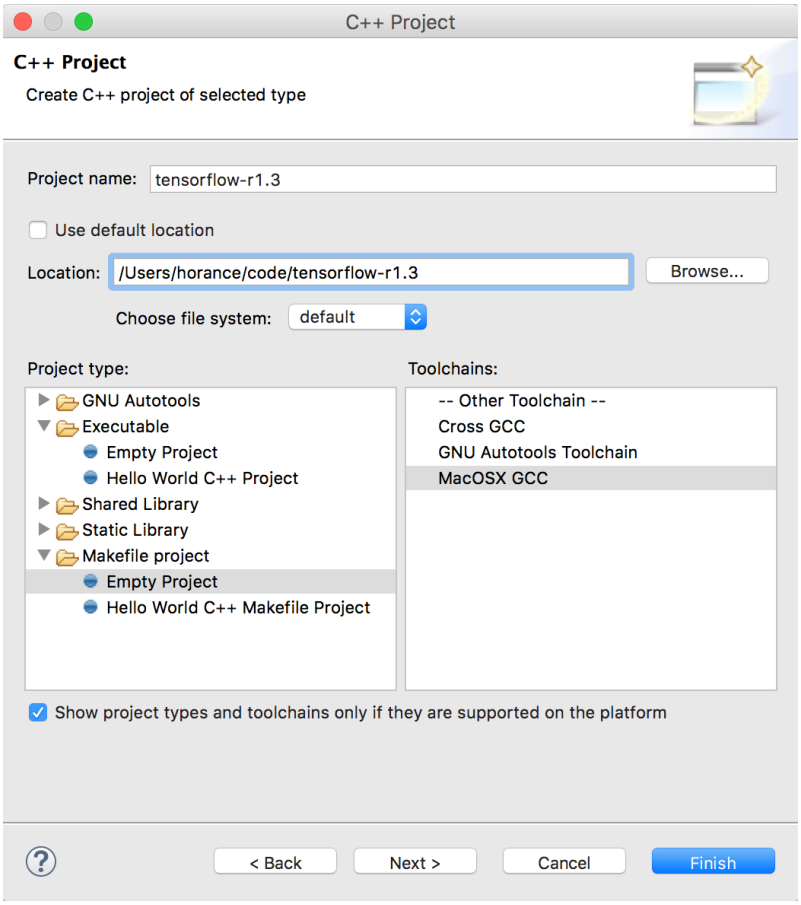


图 2-1 创建 Eclipse C++ 工程

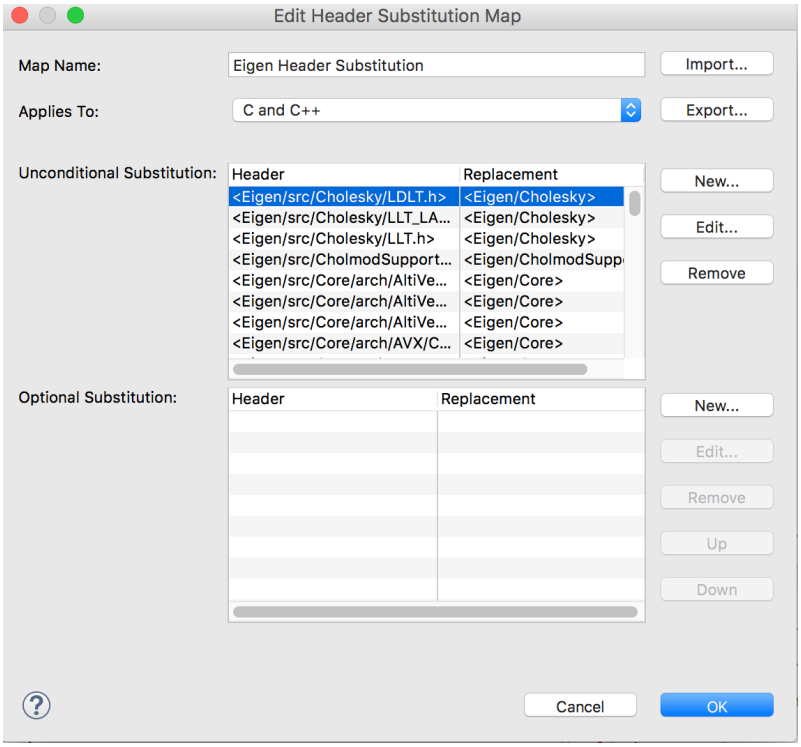


图 2-2 替换 Eigen 的头文件

2.3 代码生成

在构建 TensorFlow 系统时，Bazel 或 CMake 会自动生成部分源代码。理解代码生成器的输出结果，可以加深理解系统的行为模式。

2.4 技术栈

如图??（第??页）所示，按照系统的层次结构展示了 TensorFlow 的技术栈，构成了 TensorFlow 生态系统的核心。

层	功能	组件
视图层	可视化	TensorBoard
工作流层	数据集准备，存储，加载	Keras
计算图层	图构造/操作/优化/执行 前向计算/后向传播	TensorFlow Core
数值计算层	Kernel实现 矩阵乘法/卷积计算	Eigen/cuBLAS/cuDNN
网络层	通信	gRPC/RDMA
设备层	硬件	CPU/GPU/ASIC

图 2-3 TensorFlow 技术栈

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

- Martin Flower

3 破冰之旅

在开始探究 TensorFlow 内核之前，亲自动手实践模型的训练，熟悉模型训练的基本方法和调优技术，对于理解后续章节的内容将大有裨益。通过本文学习和实践，将了解到如何构建并训练出能够识别手写数字的神经网络[1]。

本章将循序渐进地分别使用单层感知器模型，多层感知器模型，最后尝试使用卷积神经网络。并在训练过程中，介绍了算法调优的一些常用技术，包括选择更好的激活函数，应用学习速率衰减的技术，实施 Dropout 技术等。最终，将模型的准确率提升至 99%以上。

在介绍每一种网络模型之前，将简单给出该模型的算法理论知识，帮助大家更好地理解程序的内容。但是，本书不是介绍机器学习算法的专业书籍，如需了解更多相关算法内容，请查阅相关文献和论文。

3.1 问题提出

本章使用 MNIST 数据集完成手写数字的网络模型训练，它包含了 60000 个训练样本数据；其中，包括 10000 个测试样本数据。如图**??**（第**??**页）所示，对于任意一个样本数据 x，使用 28×28 像素的数字矩阵表示。为了简化，将 28×28 的矩阵实施扁平化处理，得到长度为 784 的一维向量。

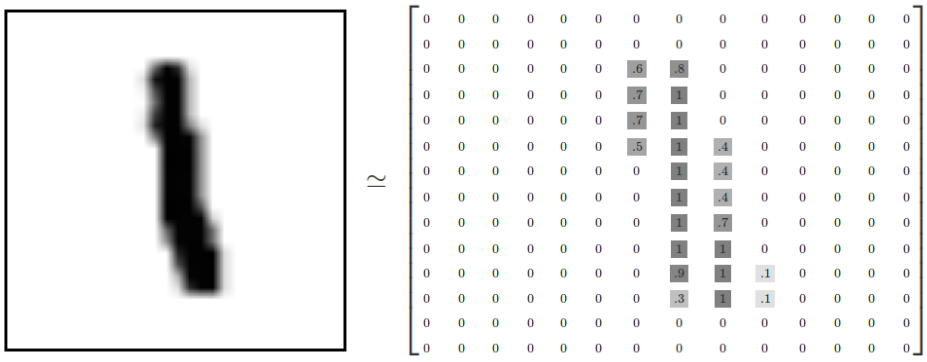


图 3-1 MNIST 样本数据表示

[1]本章内容摘自 Martin Görner 在 Codelabs 上发表的文章：Tensorflow and deep learning, without a PhD，经由 Martin Görner 同意，授权该文章在本书中发表。

样本数据集

因此，在 MNIST 训练数据集中，`mnist.train.images` 是一个 `[60000, 784]` 的二维矩阵。其中，矩阵中每一个元素，表示图片中某个像素的强度值，其值介于 0 和 1 之间。如图**??**（第**??**页）所示。

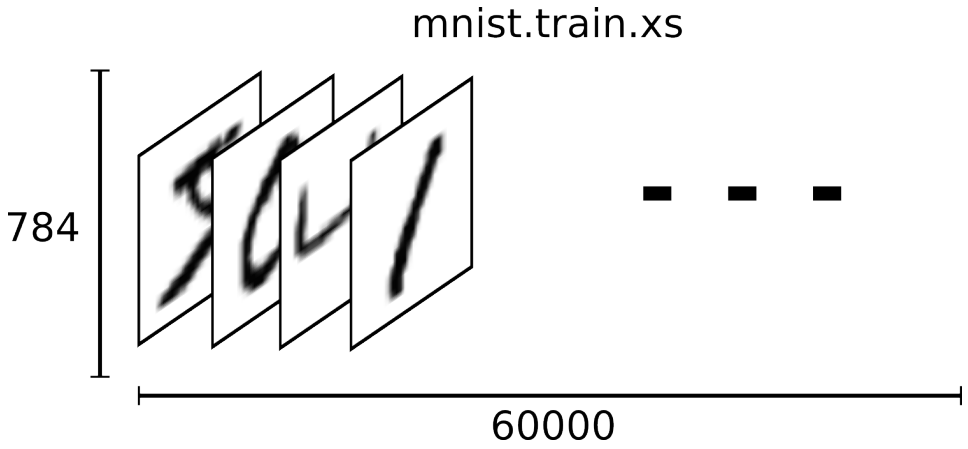


图 3-2 MNIST 训练数据集：输入数据集

相对应地，MNIST 数据集的标签是介于 0 到 9 的数字，`mnist.train.labels` 是一个 `[60000, 10]` 的二维矩阵，其中每一行是一个“one-hot”向量。如图**??**（第**??**页）所示。

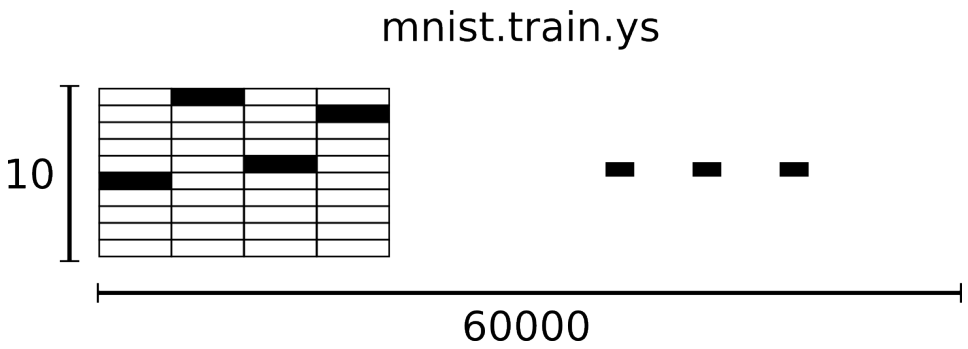


图 3-3 MNIST 训练数据集：标签数据集

图示说明

为了更好地可视化整个训练过程，使用 matplotlib 工具包绘制了 5 种类型的画板。如图**??**（第**??**页）所示，表示一个 mini-batch 的训练样本数据集。其中，`batch_size = 100`，白色背景表示数字被正确识别；而红色背景表示数字被误分类，手写数字的左侧标识了正确的标签值，而右侧标识了错误的预测值。

MNIST 拥有 50000 个训练样本，如果 `batch_size` 为 100，则需要迭代 500 次才能完整地遍历一次训练样本数据集，常称为一个 epoch 周期。

R 本章示例代码未使用 TensorBoard，而是用了 matplotlib，在训练时可以实时观察误差和精度的曲线变化。

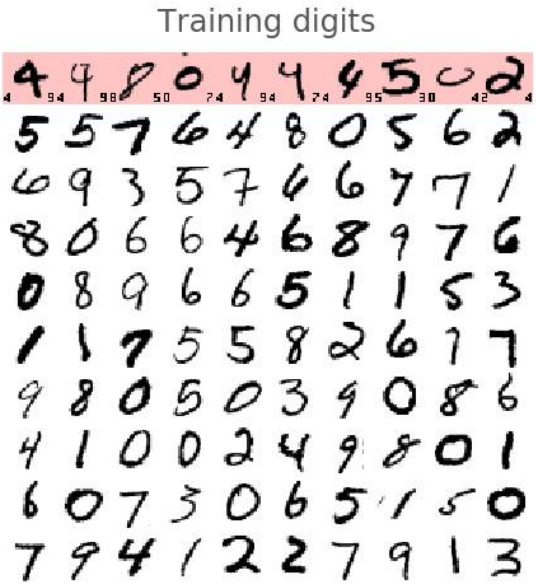


图 3-4 一次 mini-batch 的训练样本数据集，其中 `batch_size=100`

如图**??**（第**??**页）所示，MNIST 使用了规模为 10000 的测试样本数据集测试模型的当前精度。其中，左侧表示目前模型的大致精度；同样地，白色背景表示数字被正确识别；而红色背景表示数字被误分类，手写数字的左侧标识了正确的标签值，而右侧标识了错误的预测值。

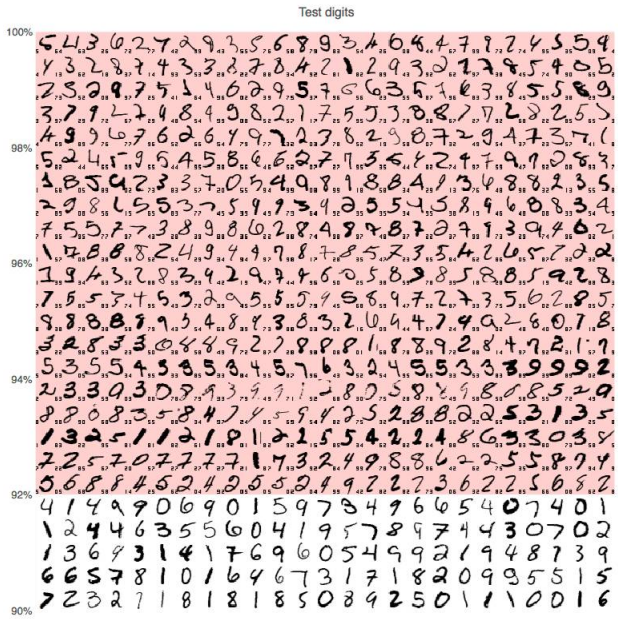


图 3-5 当前的模型精度：基于测试样本数据集

如图??（第??页）所示，使用交叉熵函数量化预测值与标签值之前的误差。其中，x 轴表示迭代的次数，y 轴表示损失值。另外，基于训练样本数据集，损失值的曲线抖动较大；而基于测试样本数据集，损失值的曲线抖动较小。

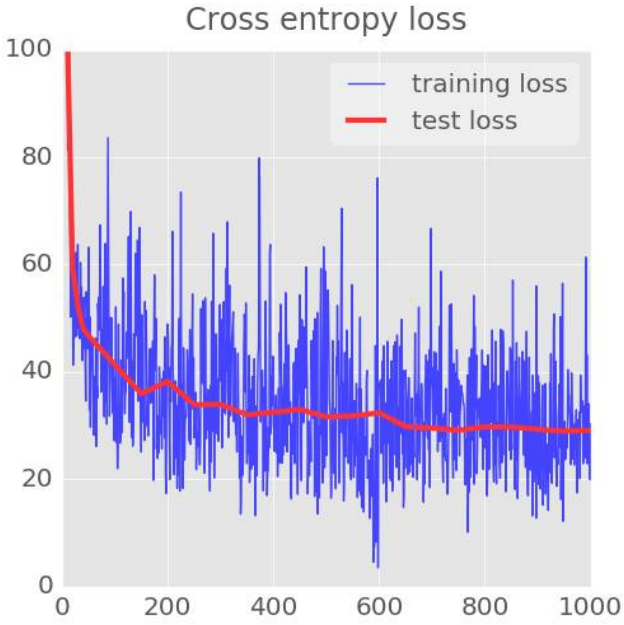


图 3-6 训练和测试的交叉熵损失

如图??（第??页）所示，可以实时计算得到模型在当前训练数据集和测试集上的精度。其中，x 轴表示迭代的次数，y 轴表示精度值。同理，基于训练样本数据集，精度曲线抖动较大；而基于测试样本数据集，精度曲线抖动较小。

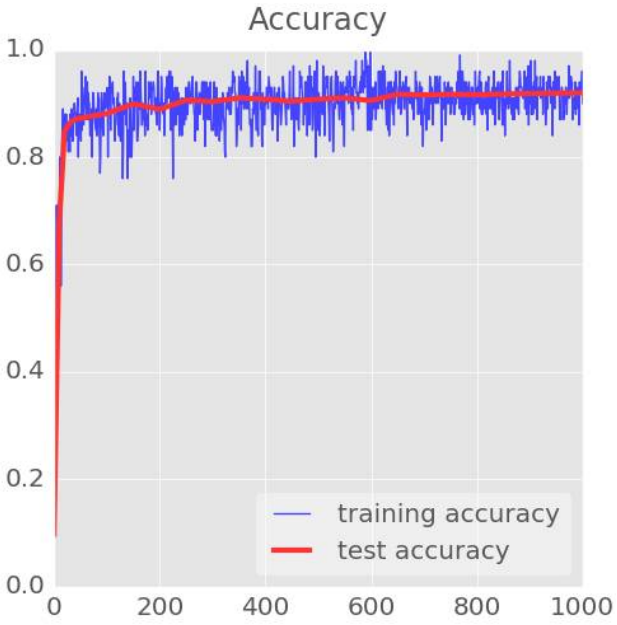


图 3-7 训练和测试的精度

如图??（第??页）所示，对于模型的每一个训练参数 (包括偏置)，可以统计得到其对应的数值分布图。当模型不能收敛时，参数的数值分布图能够给出有帮助的提示信息。

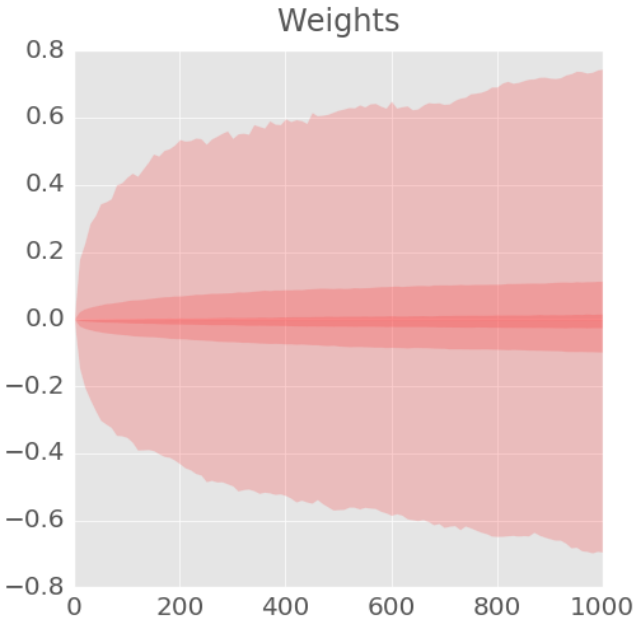


图 3-8 权重分布图

3.2 单层感知器

首先，尝试构建 10 个神经元的单层感知器。如图??（第??页）所示，对于诸如手写数字识别的多分类问题，理论上常使用 softmax 的激活函数。

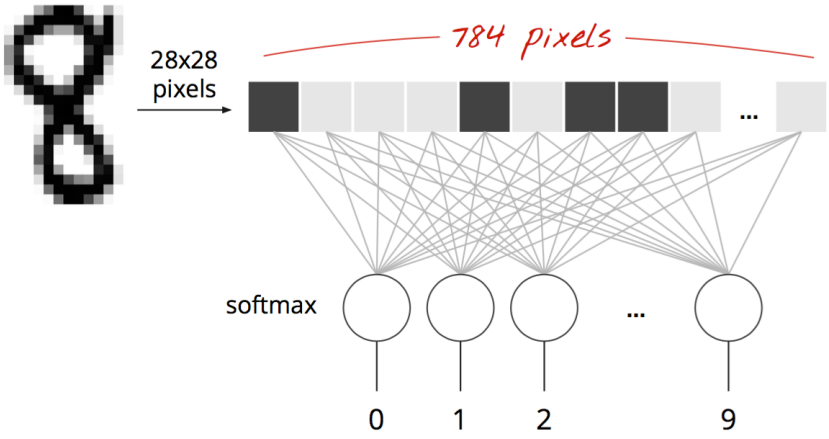


图 3-9 单层感知器

理论基础

理论上，softmax 回归是 logistic 回归的广义扩展。其中，logistic 回归是为了解决二分类问题，即 $y \in \{0, 1\}$；而 softmax 回归是为了解决 k 分类问题，即 $y \in \{1, 2, ..., k\}$。

符号定义

为了形式化地描述 softmax 回归问题，此处定义了一些常用符号。

- 训练样本集: $S = \{(x^{(i)}, y^{(i)}); i = 1, 2, ..., m\}$
- 第 i 个训练样本: $(x^{(i)}, y^{(i)})$
- 样本输入: $x = (x_1, x_2, ..., x_n)^T \in \mathbb{R}^n$
- 样本标签 (one-hot): $y = (y_1, y_2, ..., y_k)^T \in \mathbb{R}^k$
- 权重: $W \in \mathbb{R}^{n \times k}$
- 偏置: $b \in \mathbb{R}^k$
- softmax 函数: $softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad i = 1, 2, ..., k$

softmax 函数

如图**??**（第**??**页）所示，模型先求取线性加权和 z，然后求取 e^z，最后再实施归一化操作。

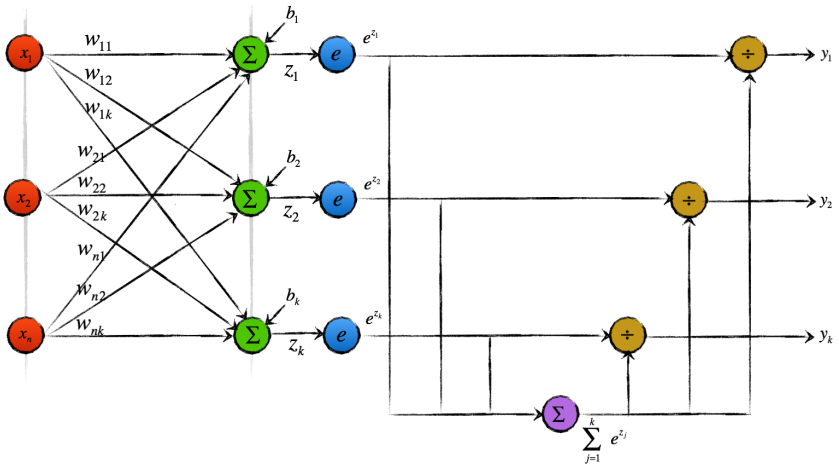


图 3-10 softmax 函数

权重与偏置

权重 W 为一个 $n \times k$ 的二维矩阵。

$$W = (W_1, W_2, ..., W_k) = \begin{pmatrix} w_{11} & \cdots & w_{1k} \\ \vdots & \ddots & \vdots \\ w_{n1} & \cdots & w_{nk} \end{pmatrix} \in \mathbb{R}^{n \times k}$$

其中，W_j 是一个长度为 n 的向量。

$$W_j = (w_{1j}, w_{2j}, ..., w_{nj})^T \in \mathbb{R}^n, j = 1, 2, ..., k$$

而偏置 b 是一个长度为 k 的“one-hot”向量。

$$b = (b_1, b_2, ..., b_k)^T \in \mathbb{R}^k$$

模型定义

多分类问题的单层感知器模型，使用 softmax 激活函数，可以如此定义。

$$\begin{aligned} y =& h_{W,b}(x) = softmax(z) = softmax(W^T x + b) \\ =& (y_1, y_2, ..., y_k)^T \\ =& \frac{1}{\sum_{j=1}^{k} e^{z_j}} (e^{z_1}, e^{z_2}, ..., e^{z_k})^T \\ =& \frac{1}{\sum_{j=1}^{k} e^{W_j^T x + b_j}} \left(e^{W_1^T x + b_1}, e^{W_2^T x + b_2}, ..., e^{W_k^T x + b_k} \right)^T \end{aligned}$$

其中，对于任意给定的样本 $(x, y) \in S$，z_i 表示 $W_i^T x + b_i$ 的线性加权和，而 $y_i (i = 1, 2, ..., k)$ 表示将其划归为类 i 的概率。

$$P(y = i | x; W, b) = \frac{e^{W_i^T x + b_i}}{\sum_{j=1}^{k} e^{W_j^T x + b_j}}$$

$$i = 1, 2, ..., k$$

交叉熵函数

基于样本数据集 $S = \{(x^{(i)}, y^{(i)}); i = 1, 2, ..., m\}$，交叉熵损失函数可以如此定义。

$$\begin{aligned} J(W, b) &= -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log\left(\widehat{y}^{(i)}\right) \\ &= -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{k} y_j^{(i)} \log\left(\widehat{y}_j^{(i)}\right) \end{aligned}$$

softmax 多分类问题，就是求取最优解 (W^*, b^*)，使得

$$W^*, b^* = \underset{W,b}{\arg\min}\, J(W, b)$$

计算梯度

对于任意一个样本 $(x, y) \in S$，可以推导出 $J(W, b)$ 相对于 W 与 b 的梯度公式。

$$\begin{aligned} \nabla_W J\left(W, b; x, y\right) &= \left(\widehat{y} - y\right) x \\ \nabla_b J\left(W, b; x^{(i)}, y^{(i)}\right) &= \left(\widehat{y} - y\right) \end{aligned}$$

参数更新

对于训练样本数据 S，根据 W, b 的梯度公式，完成本次迭代的参数更新。

$$\begin{aligned} W &\leftarrow W - \alpha\frac{\sum_{i=1}^{m} \nabla_W J\left(W, b; x^{(i)}, y^{(i)}\right)}{m} \\ b &\leftarrow b - \alpha\frac{\sum_{i=1}^{m} \nabla_b J\left(W, b; x^{(i)}, y^{(i)}\right)}{m} \end{aligned}$$

定义模型

接下来，使用 TensorFlow 完成该模型的搭建和训练。需要注意的是，理论上的公式与 TensorFlow 具体实现存在微妙的差异。理论上，公式中的 x 常表示一个样本，但 TensorFlow 中的 `x` 常表示一个 mini-batch 的样本数据集。因此，使用 TensorFlow 设计网络模型时，需要特别关注各个张量大小的变化是否符合预期。

输入和标签

首先，使用 tf.placeholder 分别定义训练样本的输入和标签。

```
x = tf.placeholder(tf.float32, [None, 28, 28, 1])
t = tf.placeholder(tf.float32, [None, 10])
```

tf.placeholder 定义了一个占位的 OP。None 表示未确定的样本数目，此处表示 batch_size 的大小；当 Session.run 时，将通过 feed_dict 的字典提供一个 mini-batch 的样本数据集，从而自动推导出 tf.placeholder 的大小。

另外，每张图片使用 $28 \times 28 \times 1$ 的三维数据表示 (灰度为 1)。为了简化问题，此处将输入的样本数据扁平化，将其变换为长度为 784 的一维向量。其中，-1 表示 mini-batch 的样本数目，由运行时自动推演其大小。

```
x = tf.reshape(x, [-1, 784])
```

定义变量

然后，使用 tf.Variable 定义模型参数。定义训练参数时，必须指定参数的初始化值；训练参数将根据初始值，自动推演数据的类型，及其大小。

```
w = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
```

此外，变量在使用之前，必须完成初始化。此处，init_op 将初始化所有全局的训练参数。

```
init_op = tf.global_variables_initializer()
```

模型定义

接下来，便可以很容易地得到多分类问题的单层感知器模型。

```
y = tf.nn.softmax(tf.matmul(x, w) + b)
```

如图**??**（第**??**页）所示，首先计算 `x` 与 `w` 的矩阵乘法，让后将 `b` 广播 (broadcast) 到矩阵的每一行相加，最终得到训练参数的线性加权和。

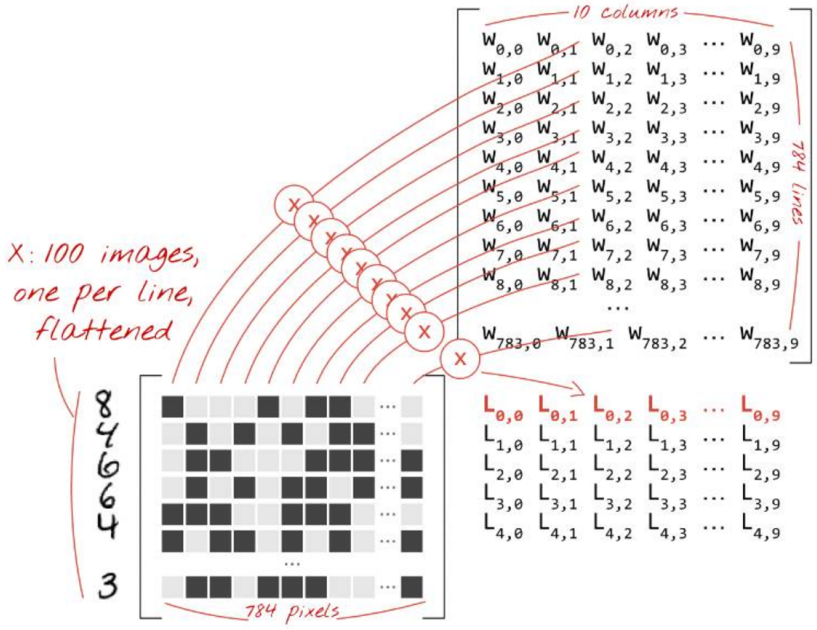


图 3-11 线性加权和

如图**??**（第**??**页）所示，softmax 将逐行实施运算，最终，`y` 的大小为 `[100, 10]`。

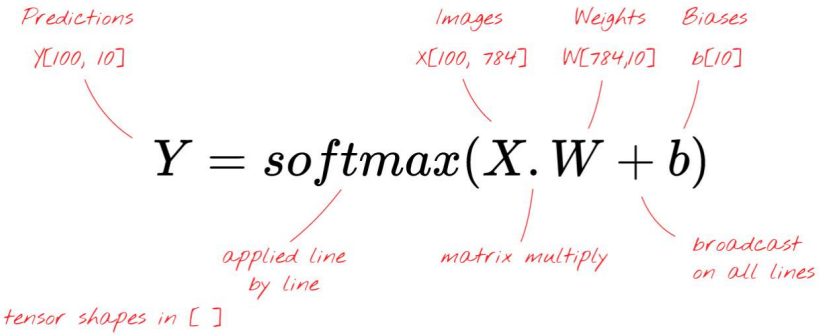


图 3-12 激活函数：softmax

损失函数

对于多分类问题，可以使用交叉熵的损失函数。

```
cross_entropy = -tf.reduce_sum(t * tf.log(y))
```

如图**??**（第**??**页）所示，`t` 和 `y` 的大小都为 `[100, 10]`；特殊地，`t` 的每一行都是一个

“one-hot”向量。

对 y 实施 tf.log 操作，也将得到一个大小为 [100, 10] 的矩阵。然后，t 与 tf.log(y) 逐位相乘 (并非矩阵相乘)，也将得到大小为 [100, 10] 的矩阵。最终，tf.reduce_sum 将矩阵中所有元素相加，得到一个标量 (scalar) 值。

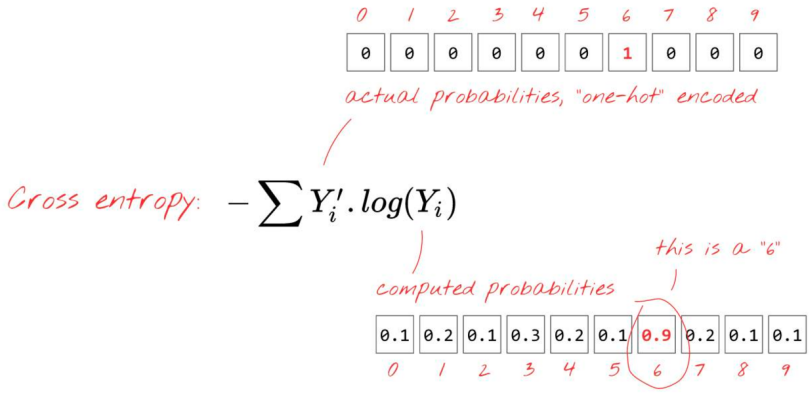


图 3-13 交叉熵损失函数

精度

tf.argmax(y,1) 将按第 1 个维度计算最大值的索引。既按照 $y_{100\times10}$ 的每一行，计算得到在每一行中最大值的的索引值。因此，tf.argmax(y,1) 将得到大小为 [100, 1] 的矩阵，或大小为 100 的向量。同样地，tf.argmax(t,1) 也是一个大小为 100 的向量。

然后，使用 tf.equal 将它们逐元素 (element-wise) 进行相等性比较，得到大小为 100 的布尔向量。为了计算精度，先将布尔向量转别为数值向量，最终求取该数值向量的均值。

```
is_correct = tf.equal(tf.argmax(y,1), tf.argmax(t,1))
accuracy = tf.reduce_mean(tf.cast(is_correct, tf.float32))
```

优化算法

接下来，使用梯度下降算法实现交叉熵损失函数的最小化。其中，learning_rate 表示学习速率，描述参数更新的快慢和步伐大小，是一个典型的超参。

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.003)
train_step = optimizer.minimize(cross_entropy)
```

如图**??**（第**??**页）所示，可以将损失函数比作一座山，登山者试图寻找最佳的行动方案达到山谷。登山者站在某个山坡上环顾四周，决定沿梯度的反方向向下走一小步，直到达到

局部最优。

当实施梯度下降更新算法时，初始点不同，获得的最小值也不同，因此梯度下降求得的只是局部最小值。另外，越接近最小值时，下降速度越慢。下降的步伐大小也非常重要，如果太小，则找到函数最小值的速度就很慢；如果太大，则可能会越过极值点。

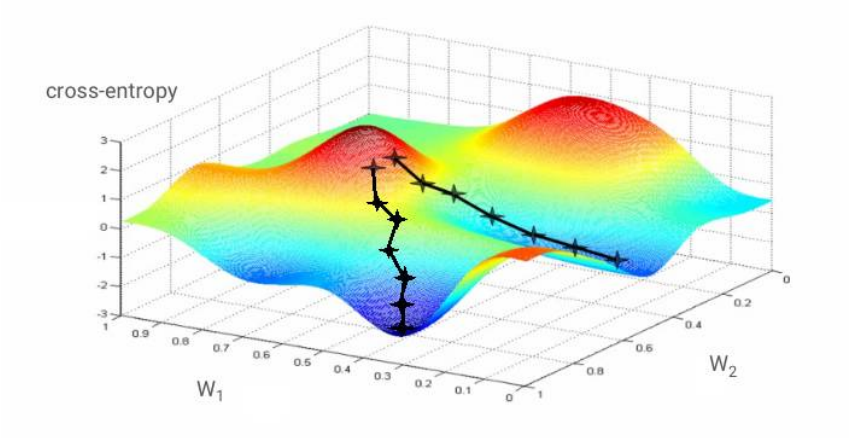


图 3-14 梯度下降算法

训练模型

在此之前，TensorFlow 仅构造计算图，并没有启动计算图的执行。接下来，客户端创建一个会话，建立与本地或远端计算设备集的通道，启动计算图的执行过程。

首先，完成训练参数的初始化。通过运行模型参数的初始化子图，并发地执行各个训练参数的初始化器，将初始值就地修改到相应的训练参数内。

```
with tf.Session() as sess:
  sess.run(init_op)
```

然后，开始迭代地执行 `train_step`，完成模型的一次迭代训练。其中，每 100 次迭代，计算当前模型在训练数据集及测试数据集的精度和损失。

```
with tf.Session() as sess:
  for step in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, t: batch_ys})

    if step % 100 == 0:
      acc, loss = sess.run([accuracy, cross_entropy],
        feed_dict={x: batch_xs, t: batch_ys})
      acc, loss = sess.run([accuracy, cross_entropy],
        feed_dict={x: mnist.test.images, t: mnist.test.labels})
```

据统计，经过 1000 次迭代，可得到大约 92% 的精度。

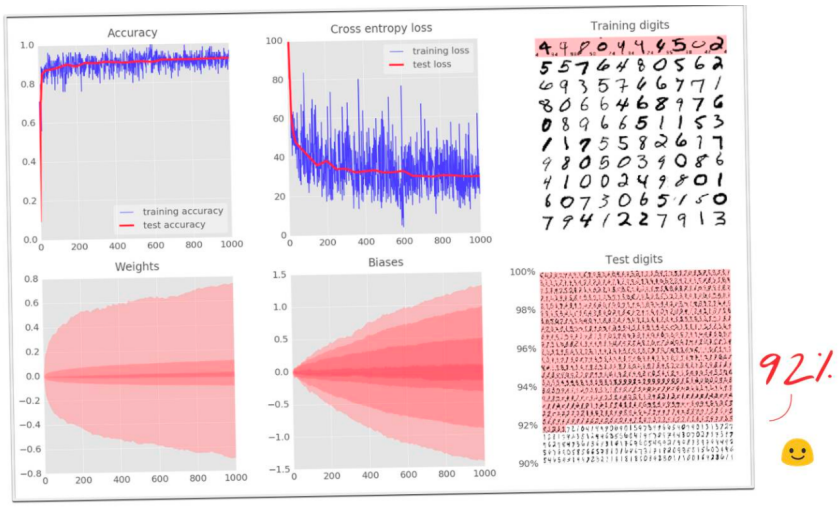


图 3-15 可视化：单层感知器，运行 1000 次 step

3.3 多层感知器

为了进一步提高精度，接下来尝试搭建 5 层的多层感知器模型。

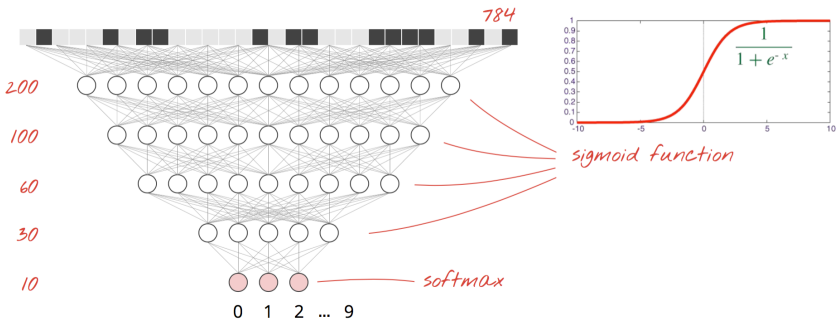


图 3-16 5 层感知器

理论基础

符号定义

为了形式化地描述多层感知器模型，此处定义了一些常用符号。

- n_ℓ: 网络层数，其中第 0 层为输入层，第 n_ℓ 层为输出层
- s_ℓ: 第 ℓ 层的节点数，$\ell = 0, 1, ..., n_\ell$
- $w_{ji}^{(\ell)}$: 第 $(\ell - 1)$ 层节点 i 与第 ℓ 层节点 j 之间的权重，$\ell = 1, ..., n_\ell$

- $b_i^{(\ell)}$: 第 ℓ 层节点 i 的偏置项，$\ell = 1, ..., n_\ell$
- $a_i^{(\ell)}$: 第 ℓ 层节点 i 的输出，$\ell = 1, ..., n_\ell, x = a^{(0)}, y = a^{(n_\ell)}$
- $z_i^{(\ell)}$: 第 ℓ 层节点 i 的权重和，$\ell = 1, ..., n_\ell$
- $\delta_i^{(\ell)}$: 第 ℓ 层节点 i 的误差项，$\ell = 1, ..., n_\ell$
- $S = \{(x^{(t)}, y^{(t)}); t = 1, 2, ..., m\}$: 样本空间

前向传播

$z^{(\ell)}$ 表示 ℓ 层的线性加权和，它由第 $\ell - 1$ 层的输出 $a^{(\ell-1)}$ 与第 ℓ 层的权重矩阵 $w^{(\ell)}$ 相乘，再加上第 ℓ 层的偏置向量所得。

推而广之，第 ℓ 层的输出，由激活函数 $f(z^{(\ell)})$ 所得。其中，$a^{(0)} = x, y = a^{(n_\ell)}$。

$$
\begin{aligned}
z^{(\ell)} &= w^{(\ell)} a^{(\ell-1)} + b^{(\ell)} \\
a^{(\ell)} &= f(z^{(\ell)}) \\
a^{(0)} &= x \\
y &= a^{(n_\ell)}
\end{aligned}
$$

后向传播

然后，反向计算各层的误差。其中，第 ℓ 层的误差，由 $\ell + 1$ 层的误差计算所得。特殊地，在输出层，预测值 $a^{(n_\ell)}$ 与 y 之间的误差，可以直接计算得到。

$$
\delta^{(\ell)} = \begin{cases} (w^{(\ell+1)})^T \delta^{(\ell+1)} \circ f'(z^{(\ell)}); & \ell \neq n_\ell \\ (a^{(\ell)} - y) \circ f'(z^{(\ell)}); & \ell = n_\ell \end{cases}
$$

损失函数 $J(w, b)$ 相对于各层的权重矩阵与偏置向量的梯度便可以计算得到。

$$
\begin{aligned}
&\nabla_{w^{(\ell)}} J(w, b; x, y) = \delta^{(\ell)} \left(a^{(\ell-1)} \right)^T \\
&\nabla_{b^{(\ell)}} J(w, b; x, y) = \delta^{(\ell)} \\
&\ell = 1, 2, ..., n_\ell
\end{aligned}
$$

一般地，在实际系统实现中，下游层传递梯度给上层，上层直接完成梯度的计算。

参数更新

对于给定样本数据集 $S = \{(x^{(t)}, y^{(t)}); t = 1, 2, ..., m\}$，根据梯度反传公式，可以计算得到参数更新的变化量。

$$\begin{aligned}\Delta w^{(\ell)} &\leftarrow \Delta w^{(\ell)} + \nabla_{w^{(\ell)}} J\left(w, b; x^{(t)}, y^{(t)}\right)\\ \Delta b^{(\ell)} &\leftarrow \Delta b^{(\ell)} + \nabla_{b^{(\ell)}} J\left(w, b; x^{(t)}, y^{(t)}\right)\\ t &= 1, 2, ..., m; \ell = 1, 2, ..., n_\ell\end{aligned}$$

最后，执行梯度下降算法，完成训练参数一个迭代的更新。

$$\begin{aligned}w^{(\ell)} &\leftarrow w^{(\ell)} - \alpha\left(\frac{\Delta w^{(\ell)}}{m}\right)\\ b^{(\ell)} &\leftarrow b^{(\ell)} - \alpha\frac{\Delta b^{(\ell)}}{m}\\ \ell &= 1, 2, ..., n_\ell\end{aligned}$$

定义模型

相对于上一节中尝试的单层感知器，此处定义每一个隐式层的权重时，并没有使用常量定义变量的初始值，而使用满足某种数据分布特征的随机值。

```
K = 200
L = 100
M = 60
N = 30

w1 = tf.Variable(tf.truncated_normal([28*28, K] ,stddev=0.1))
b1 = tf.Variable(tf.zeros([K]))

w2 = tf.Variable(tf.truncated_normal([K, L], stddev=0.1))
b2 = tf.Variable(tf.zeros([L]))

w3 = tf.Variable(tf.truncated_normal([L, M], stddev=0.1))
b3 = tf.Variable(tf.zeros([M]))

w4 = tf.Variable(tf.truncated_normal([M, N], stddev=0.1))
b4 = tf.Variable(tf.zeros([N]))

w5 = tf.Variable(tf.truncated_normal([N, 10], stddev=0.1))
b5 = tf.Variable(tf.zeros([10]))
```

在定义每一个隐式层时，采用 sigmoid 的激活函数。而在最后的输出层，采用 softmax 的激活函数。

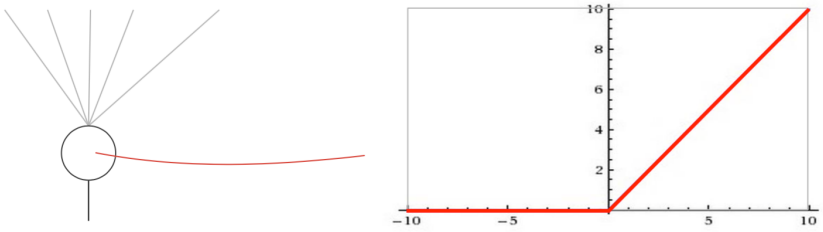
```
y1 = tf.nn.sigmoid(tf.matmul(x,  w1) + b1)
y2 = tf.nn.sigmoid(tf.matmul(y1, w2) + b2)
y3 = tf.nn.sigmoid(tf.matmul(y2, w3) + b3)
y4 = tf.nn.sigmoid(tf.matmul(y3, w4) + b4)
y  = tf.nn.softmax(tf.matmul(y4, w5) + b5)
```

经过迭代的模型训练，可以得到大约 97% 左右的精度。但是，网络随着层次的增加，模型变得越来越难以收敛。接下来，尝试一些常见的调优技术，改善网络的性能。

优化技术

激活函数：ReLU

在深度模型中，不适合使用 sigmoid 激活函数。它将把所有的值都挤到了 0 到 1 之间；随着网络层次的增加，引发梯度消失的问题。




```
Y = tf.nn.relu(tf.matmul(X, W) + b)
```

图 3-17 ReLU 激活函数

可以使用 ReLU(Rectified Linear Unit) 替代 sigmoid，不仅避免了 sigmoid 导致的一些问题，而且能够加快初始的收敛速度。

```
y1 = tf.nn.relu(tf.matmul(x,  w1) + b1)
y2 = tf.nn.relu(tf.matmul(y1, w2) + b2)
y3 = tf.nn.relu(tf.matmul(y2, w3) + b3)
y4 = tf.nn.relu(tf.matmul(y3, w4) + b4)
y  = tf.nn.softmax(tf.matmul(y4, w5) + b5)
```

另外，如果使用 ReLU 激活函数，偏置向量常常初始化为小的正值，使得神经元在一开始就会工作在 ReLU 的非零区域内。

```
K = 200
L = 100
M = 60
N = 30

w1 = tf.Variable(tf.truncated_normal([28*28, K] ,stddev=0.1))
b1 = tf.Variable(tf.ones([L])/10)

w2 = tf.Variable(tf.truncated_normal([K, L], stddev=0.1))
b2 = tf.Variable(tf.ones([L])/10)

w3 = tf.Variable(tf.truncated_normal([L, M], stddev=0.1))
b3 = tf.Variable(tf.ones([L])/10)

w4 = tf.Variable(tf.truncated_normal([M, N], stddev=0.1))
b4 = tf.Variable(tf.ones([L])/10)

w5 = tf.Variable(tf.truncated_normal([N, 10], stddev=0.1))
b5 = tf.Variable(tf.ones([L])/10)
```

如图**??**（第**??**页）所示，前 300 次迭代，使用 ReLU 相对于使用 sigmoid，其初始收敛速度提升显著。

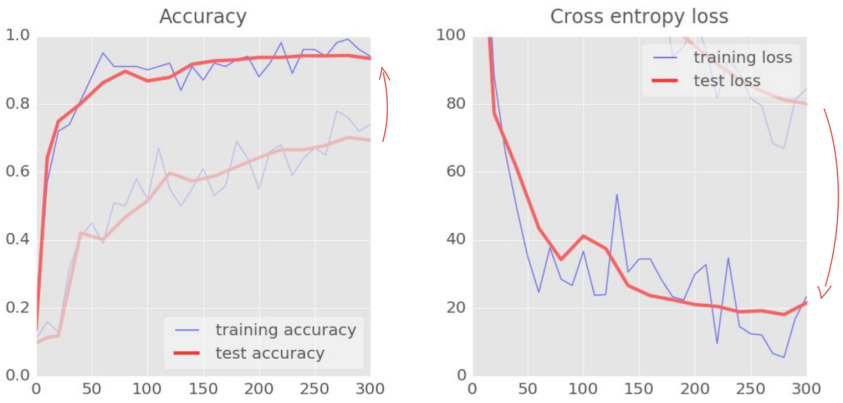


图 3-18 应用 ReLU 激活函数：初始收敛速度提升显著

不定值

为了得到稳定的数值计算结果，避免出现精度突降为 0 的情况发生。追溯实现代码，可能引入 `log(0)` 计算得到 `NaN` 不定值的问题。可以使用 `softmax_cross_entropy_with_logits` 计算交叉熵损失，并将线性加权和作为其输入 (常称为 logits)。

```
logits = tf.matmul(y4, w5) + b5
y = tf.nn.softmax(logits)

cross_entropy = tf.nn.softmax_cross_entropy_with_logits(
  logits=logits, labels=t)
```

学习速率衰减

随着网路层次的增加，及其应用相关优化技术后，模型的精度能够能得到 98% 左右，但很难得到一个稳定的精度。如图??（第??页）所示，精度和损失抖动相当明显。

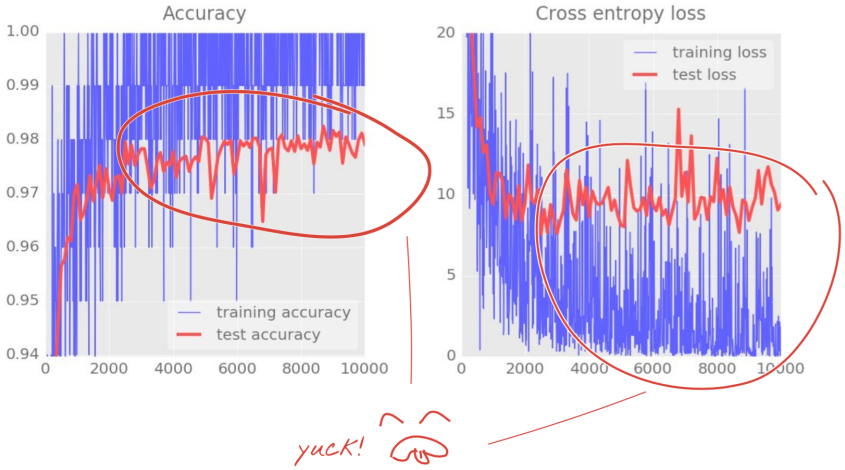


图 3-19 噪声抖动：学习速率过大

可以采用更好的优化算法，例如 AdamOptimizer。随着迭代过程的次数，学习速率将指数级衰减，在模型训练后期可以得到一个更稳定的精度和损失曲线。

```
lr = tf.placeholder(tf.float32)
train_step = tf.train.AdamOptimizer(lr).minimize(cross_entropy)
```

在每个迭代训练过程中，根据当前 step 的值，实时计算当前迭代的学习速率 lr，然后通过 feed_dict 传递给 Session.run 执行。其中，学习速率衰减方程如下代码所示，随着迭代次数的增加，学习速率指数衰减。

```
def lr(step):
  max_lr, min_lr, decay_speed = 0.003, 0.0001, 2000.0
  return min_lr + (max_lr - min_lr) * math.exp(-step/decay_speed)

with tf.Session() as sess:
  for step in range(10000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step,
      feed_dict={x: batch_xs, t: batch_ys, pkeep: 0.75, lr: lr(step)})
```

如图??（第??页）所示，应用学习速率衰减方法后，可以得到一个更稳定的精度和损失曲线。

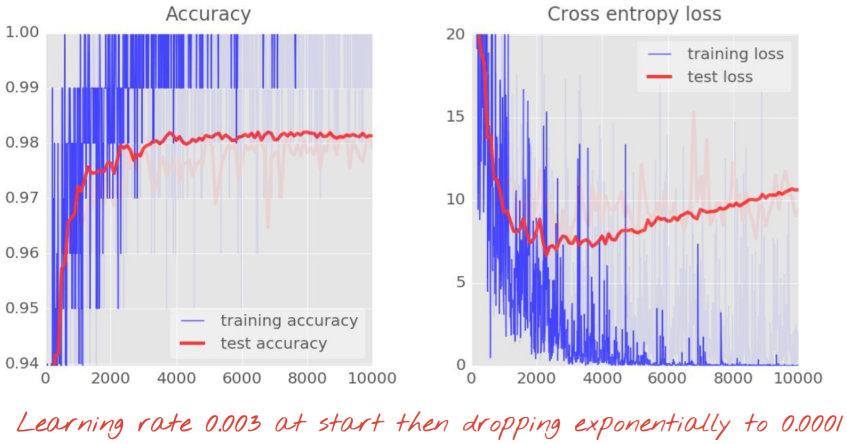


图 3-20 应用 Adam 优化算法后，精度和损失趋于稳定

应用 Dropout

但是，损失曲线在训练集与测试集上相分离，出现明显的过拟合现象。即模型在训练数据集上表现良好，但在测试数据集上出现反弹，模型缺乏足够的泛化能力。

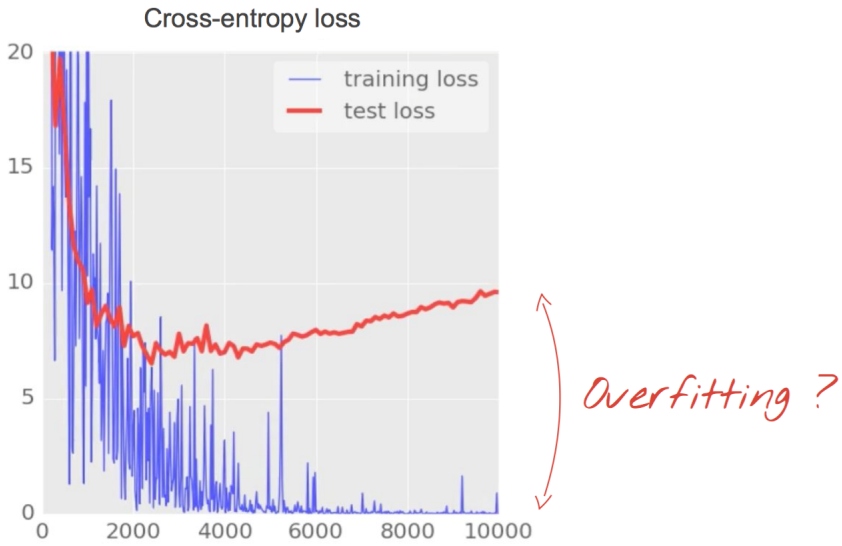


图 3-21 过拟合

如图**??**（第**??**页）所示，在训练时对隐藏层的输出实施 dropout 操作，以 `1 - pkeep` 的概率随机丢弃神经元的输出，并在反向传播梯度时不再更新相应的权重。而在推理时恢复所有神经元的输出，间接改善了网络的泛化能力。

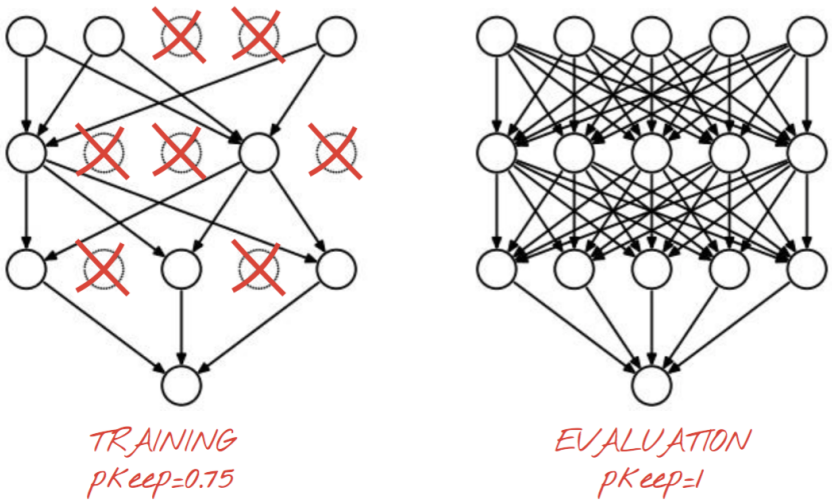


图 3-22 Dropout 方法

使用 TensorFlow 实现 dropout 操作时，先定义一个超级参数 pkeep，表示隐藏层的神经元以概率 pkeep 随机保留，以概率 1 - pkeep 随机丢弃。

```
pkeep = tf.placeholder(tf.float32)

y1 = tf.nn.relu(tf.matmul(x,  w1) + b1)
y1d = tf.nn.dropout(y1, pkeep)

y2 = tf.nn.relu(tf.matmul(y1d, w2) + b2)
y2d = tf.nn.dropout(y2, pkeep)

y3 = tf.nn.relu(tf.matmul(y2d, w3) + b3)
y3d = tf.nn.dropout(y3, pkeep)

y4 = tf.nn.relu(tf.matmul(y3d, w4) + b4)
y4d = tf.nn.dropout(y4, pkeep)

logits = tf.matmul(y4d, w5) + b5
y = tf.nn.softmax(Ylogits)
```

在训练时，置超参 pkeep 的值小于 1；而在推理时，置超参 pkeep 的值为 1。

```
with tf.Session() as sess:
  for step in range(10000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step,
      feed_dict={x: batch_xs, t: batch_ys, pkeep: 0.75, lr: lr(step)})

    if step % 100 == 0:
      acc, loss = sess.run([accuracy, cross_entropy],
        feed_dict={x: batch_xs, t: batch_ys, pkeep: 1})
      acc, loss = sess.run([accuracy, cross_entropy],
        feed_dict={x: mnist.test.images, t: mnist.test.labels, pkeep: 1})
```

在每一隐藏层实施 dropout 操作之后，训练集与测试集的损失曲线再次相交。但是，精度和损失曲线相比又出现小幅的抖动，而且训练集与测试集的损失曲线相重合的程度不是

很理想，过拟合问题依然很突出。

也就是说，过拟合问题存在其他更深刻的原因。例如，将 28×28 的图片实施扁平化操作，将其变换为一个长度为 784 的一维向量，这将完全丢失了像素的空间排列信息。

接下来，通过尝试构造卷积神经网络，从原图像中提取特征，从而保留了像素的空间排列信息，进而提升网络的性能。

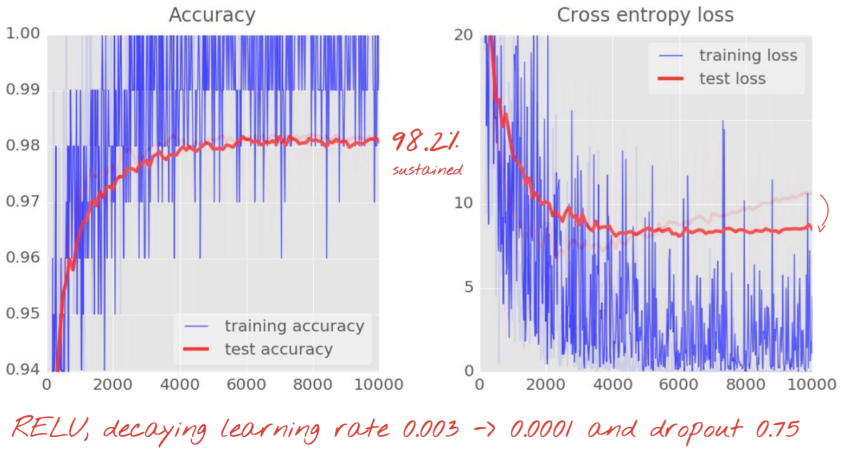


图 3-23 实施 Dropout 后，训练集与测试集的损失曲线再次重合

3.4 卷积网络

特征与优势

随着网络层次增加，全连接网络的梯度消失的问题将越发突出，收敛速度变得越来越慢。相对于全连接网络，卷积网络具有 3 个主要特征，它减少了网络参数的数量，提升网络的泛化能力。

局部连接

相对于全连接网络，卷积网络实现了局部连接，即每个神经元并不与上一层的神经元都存在连接。如图**??**（第**??**页）左侧所示，假如存在一张 1000×1000 像素的图像，及其 10^6 个隐藏层的神经元。在全连接网络里，将拥有 $10^3 \times 10^3 \times 10^6 = 10^{12}$ 个训练参数。

事实上，每个神经元没有必要与上一层神经元都存在连接。如图**??**（第**??**页）右侧所示，假如每个隐藏层的神经元仅与上一层 10×10 的局部图像存在连接，10^6 个隐藏层的神经元则需要 $10^6 \times 10^2 = 10^8$ 个网络连接，相比减少了 4 个数量级。

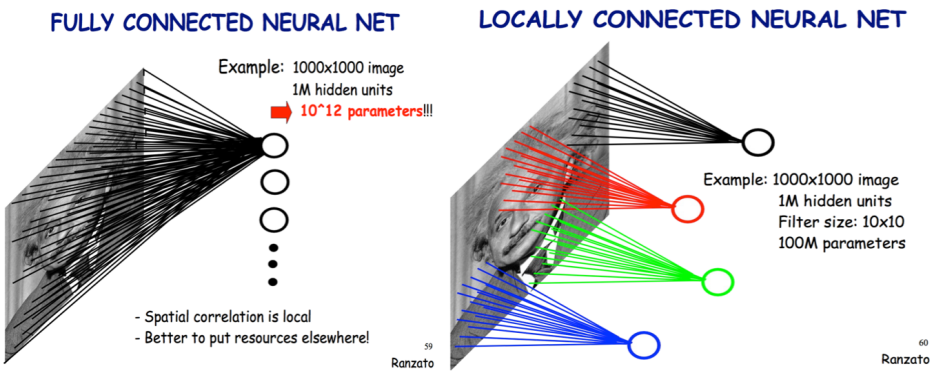


图 3-24 局部连接

权值共享

为了进一步减少网络连接，卷积网络还实现了权重共享；即每一组连接共享相同的权重，而不是每个连接存在不同的权重。如图**??**（第**??**页）右侧所示，每个隐藏层的神经元仅与 10×10 的局部图像存在连接，且共享 10×10 的权重矩阵，与隐藏层的神经元的数目无关。相对于如图**??**（第**??**页）左侧的局部连接网络，需要 10^8 个参数，卷积层仅需 10^2 个参数。

如图**??**（第**??**页）右侧所示，为了提取不同特征，例如不同边缘的图像特征，可以使用多个过滤器。例如，存在 100 个过滤器，则需要 10^4 个参数。

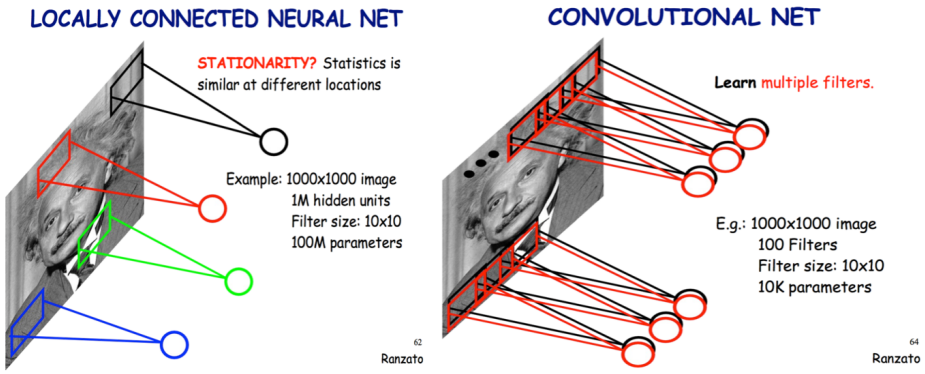


图 3-25 权值共享，多个过滤器

下采样

如图**??**（第**??**页）所示，可选地实施下采样，进一步减少网络的参数，提升模型的鲁棒性。

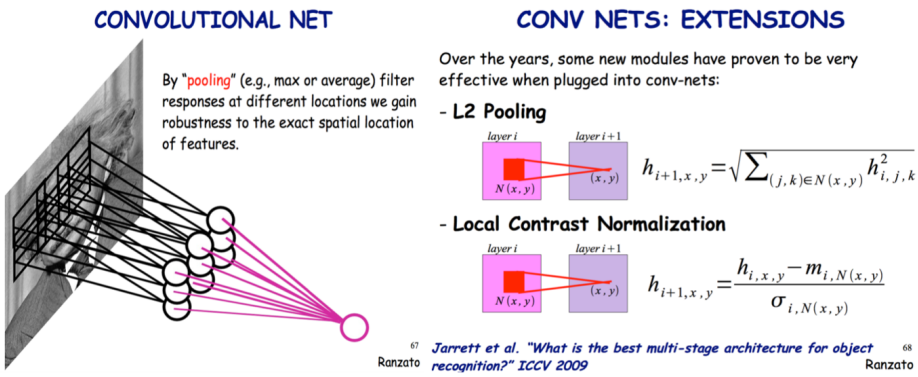


图 3-26 下采样

卷积运算

卷积运算是一个计算密集型的 OP。如图**??**（第**??**页）所示，存在一个权重向量 `w[3,3,3,2]`，其中输入通道数为 3，输出通道数为 2，卷积核大小为 3×3。

显而易见，输入图像的通道数，等价于卷积核的深度；卷积核的数目，等于 Feature Map 的输出通道数。另外，为了采集图像的边缘特征，在原图像外围补了 (padding) 一圈零值。每次卷积计算，移动的步长 (stride) 为 2。因此，最终输出的 Feature Map 大小为 3×3。

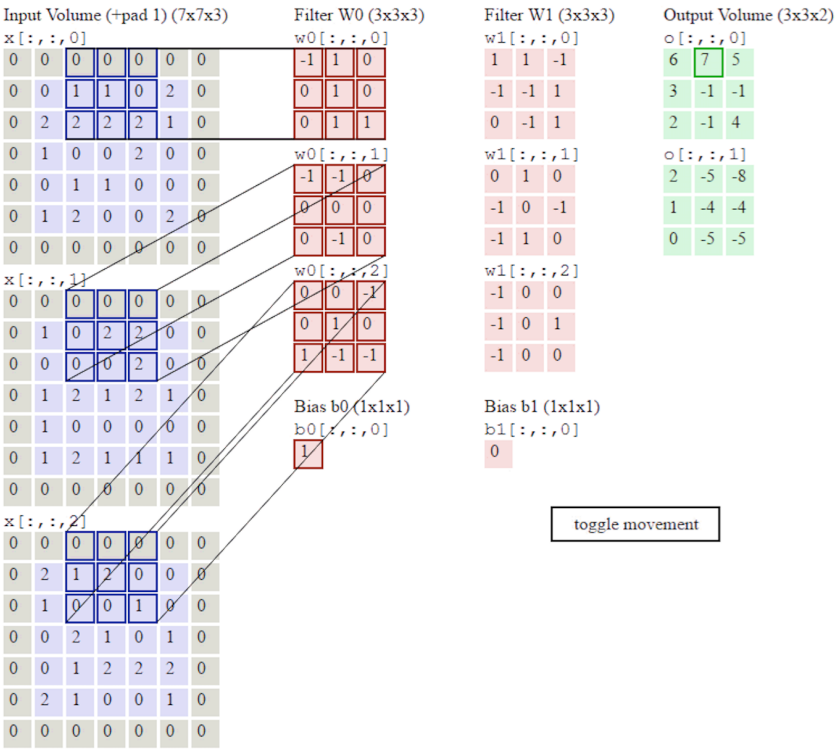


图 3-27 卷积运算

例子

假如存在一个 $32 \times 32 \times 3$ 的图片，卷积核大小为 $5 \times 5 \times 3$。其中，卷积核的深度等于图片的输入通道数。如图**??**（第**??**页）所示，卷积核与图片中大小为 $5 \times 5 \times 3$ 的块实施点积运算，得到一个值。

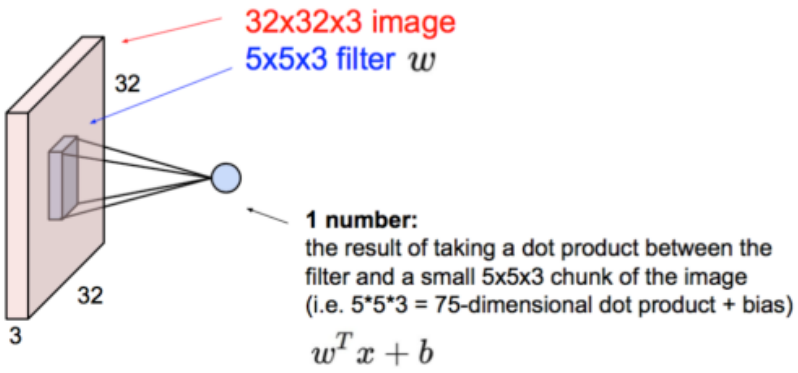


图 3-28 卷积运算：卷积核与图片块的点积运算

如图**??**（第**??**页）所示，卷积核遍历整个图片空间，最终得到一个大小为 $28 \times 28 \times 1$ 的 Feature Map。

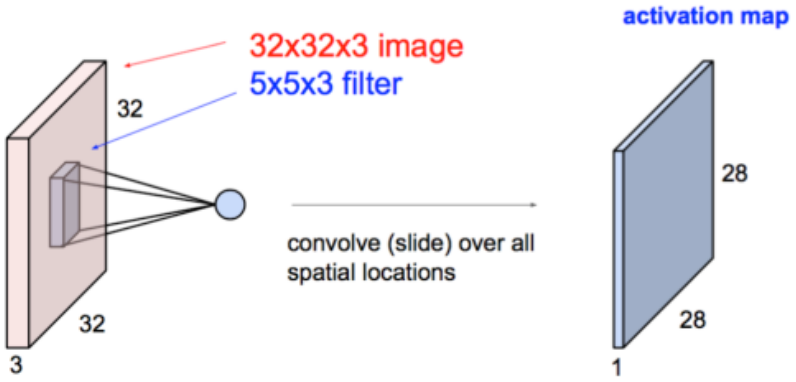


图 3-29 卷积运算：卷积核遍历图片，步长为 1

如图**??**（第**??**页）所示，如果存在多个卷积核，则得到多个 Feature Map。

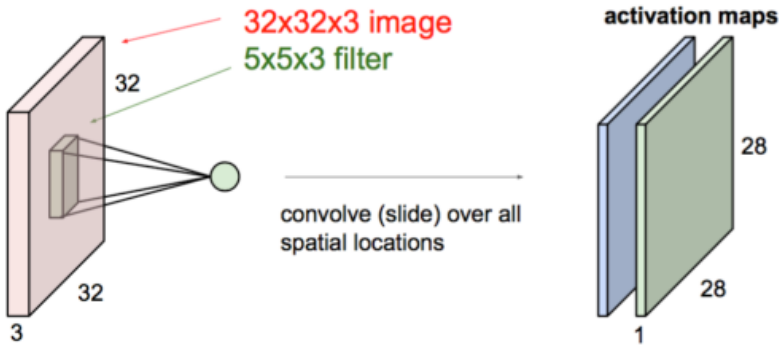


图 3-30 卷积运算：多个卷积核

公式推导

前向传播

$Z^{(\ell)}$ 表示 ℓ 层的线性加权和，它由第 $\ell-1$ 层的输出 $A^{(\ell-1)}$ 与第 ℓ 层的权重矩阵 $W^{(\ell)}$ 卷积，再加上第 ℓ 层的偏置向量所得。

推而广之，第 ℓ 层的输出，由激活函数 $f(Z^{(\ell)})$ 所得。其中，$A^{(0)}=x, y=A^{(n_\ell)}$。

$$
\begin{aligned}
Z^{(\ell)} &= A^{(\ell-1)} * W^{(\ell)} + b^{(\ell)} \\
A^{(\ell)} &= f\left(Z^{(\ell)}\right)
\end{aligned}
$$

后向传播

然后，反向计算各层的误差。其中，第 ℓ 层的误差，由 $\ell+1$ 层的误差计算所得。相对于全连接网络，此处运用的是卷积运算，并非矩阵乘法运算。

$$
\delta^{(\ell)} = \delta^{(\ell+1)} * W^{(\ell+1)} \circ f'\left(z^{(\ell)}\right)
$$

损失函数 $J(w,b)$ 相对于各层的权重矩阵与偏置向量的梯度便可以计算得到。

$$
\begin{aligned}
\nabla_{W^{(\ell)}} J(W,b) &= A^{(\ell-1)} * \delta^{(\ell)} \\
\nabla_{b^{(\ell)}} J(W,b) &= \delta^{(\ell)}
\end{aligned}
$$

实现卷积网络

实现卷积网络时，首先需要定义每层过滤器的权重矩阵，用于提取图像的特征。权重矩阵在图像里表现的像一个从原始图像矩阵中提取特定信息的过滤器。一个权重矩阵可能用来提取图像边缘信息，一个权重矩阵可能是用来提取一个特定颜色，另一个权重矩阵可能对不需要的噪声进行模糊化。

当存在多个卷积层时，初始层往往提取较多的一般特征；随着网络结构变得更深，权值矩阵提取的特征越来越复杂，并且越来越适用于眼前的具体问题。

一般地，过滤器常使用一个 4 维的张量表示，前两维表示过滤器的大小，第三维表示输入的通道数，第四维表示输出的通道数。如图**??**（第**??**页）所示。

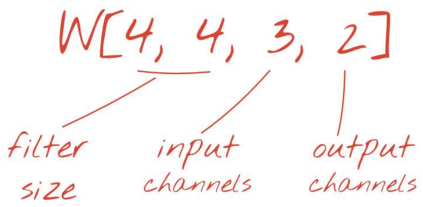


图 3-31 卷积层的过滤器

如图**??**（第**??**页）所示，构造了 3 个卷积层和 2 个全连接层。其中，中间隐藏层使用 ReLU 的激活函数，最后的输出层采用 softmax 的激活函数。

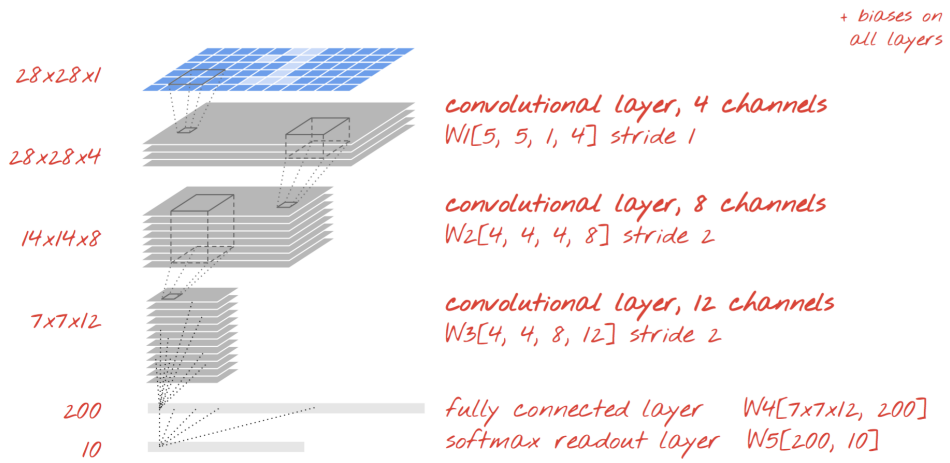


图 3-32 实现卷积神经网络

使用 TensorFlow 实现卷积网络，如下代码所示。

```
K = 4
L = 8
M = 12
N = 200
```

```
w1 = tf.Variable(tf.truncated_normal([5, 5, 1, K], stddev=0.1))
b1 = tf.Variable(tf.ones([K])/10)

w2 = tf.Variable(tf.truncated_normal([5, 5, K, L], stddev=0.1))
b2 = tf.Variable(tf.ones([L])/10)

w3 = tf.Variable(tf.truncated_normal([4, 4, L, M], stddev=0.1))
b3 = tf.Variable(tf.ones([M])/10)

w4 = tf.Variable(tf.truncated_normal([7 * 7 * M, N], stddev=0.1))
b4 = tf.Variable(tf.ones([N])/10)

w5 = tf.Variable(tf.truncated_normal([N, 10], stddev=0.1))
b5 = tf.Variable(tf.ones([10])/10)

y1 = tf.nn.relu(tf.nn.conv2d(
       x,  w1, strides=[1, 1, 1, 1], padding='SAME') + b1)
y2 = tf.nn.relu(tf.nn.conv2d(
       y1, w2, strides=[1, 2, 2, 1], padding='SAME') + b2)
y3 = tf.nn.relu(tf.nn.conv2d(
       y2, w3, strides=[1, 2, 2, 1], padding='SAME') + b3)

yy = tf.reshape(Y3, shape=[-1, 7 * 7 * M])
y4 = tf.nn.relu(tf.matmul(yy, w4) + b4)

logits = tf.matmul(y4, w5) + b5
y = tf.nn.softmax(logits)
```

如图**??**（第**??**页）所示，经过 10^4 次训练，可以得到大约 98.9% 的准确率。

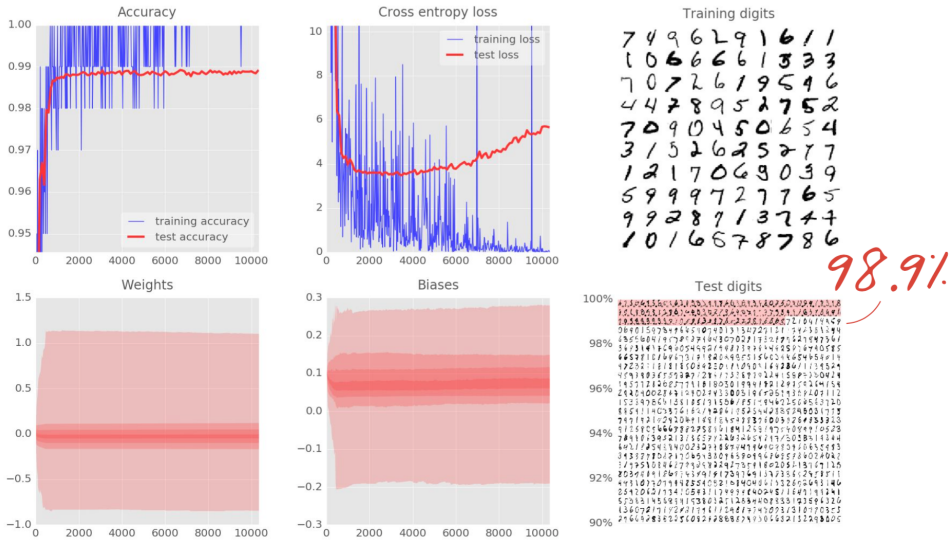


图 3-33 实现卷积网络：可以得到 98.9% 的准确率

增强卷积网络

如图**??**（第**??**页）所示，保留之前的网络层次结构，构造了 3 个卷积层和 2 个全连接层。其中，中间隐藏层使用 ReLU 的激活函数，最后的输出层采用 softmax 的激活函数。

但是，相对于之前的卷积网络，使用了更多的通道提取更多的特征。同时，在全连接的隐藏层中实施 dropout 操作，增强网络的泛化能力。

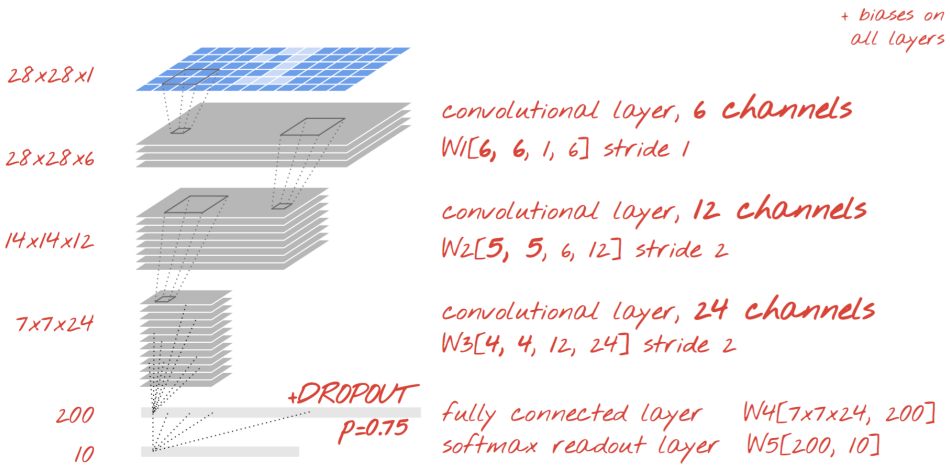


图 3-34 改善卷积神经网络

使用 TensorFlow 实现更大的卷积网络，如下代码所示。

```
K = 6
L = 12
M = 24
N = 200

w1 = tf.Variable(tf.truncated_normal([6, 6, 1, K], stddev=0.1))
b1 = tf.Variable(tf.ones([K])/10)

w2 = tf.Variable(tf.truncated_normal([5, 5, K, L], stddev=0.1))
b2 = tf.Variable(tf.ones([L])/10)

w3 = tf.Variable(tf.truncated_normal([4, 4, L, M], stddev=0.1))
b3 = tf.Variable(tf.ones([M])/10)

w4 = tf.Variable(tf.truncated_normal([7 * 7 * M, N], stddev=0.1))
b4 = tf.Variable(tf.ones([N])/10)

w5 = tf.Variable(tf.truncated_normal([N, 10], stddev=0.1))
b5 = tf.Variable(tf.ones([10])/10)

y1 = tf.nn.relu(tf.nn.conv2d(
       x,  w1, strides=[1, 1, 1, 1], padding='SAME') + b1)
y2 = tf.nn.relu(tf.nn.conv2d(
       y1, w2, strides=[1, 2, 2, 1], padding='SAME') + b2)
y3 = tf.nn.relu(tf.nn.conv2d(
       y2, w3, strides=[1, 2, 2, 1], padding='SAME') + b3)

yy = tf.reshape(Y3, shape=[-1, 7 * 7 * M])
y4 = tf.nn.relu(tf.matmul(yy, w4) + b4)
y4d = tf.nn.dropout(y4, pkeep)

logits = tf.matmul(y4d, w5) + b5
y = tf.nn.softmax(logits)
```

如图**??**（第**??**页）所示，经过 10^4 次训练，可以得到大约 99.3% 的准确率。

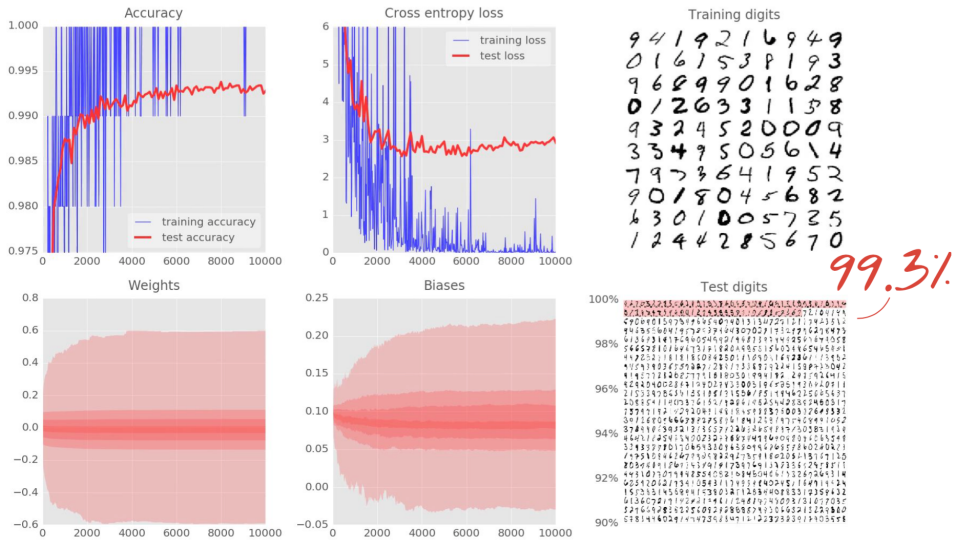


图 3-35 增强卷积网络：可以得到 99.3% 的准确率

同时，相对于之前实现的卷积网络，过拟合问题得到了明显地改善，如图**??**（第**??**页）所示。

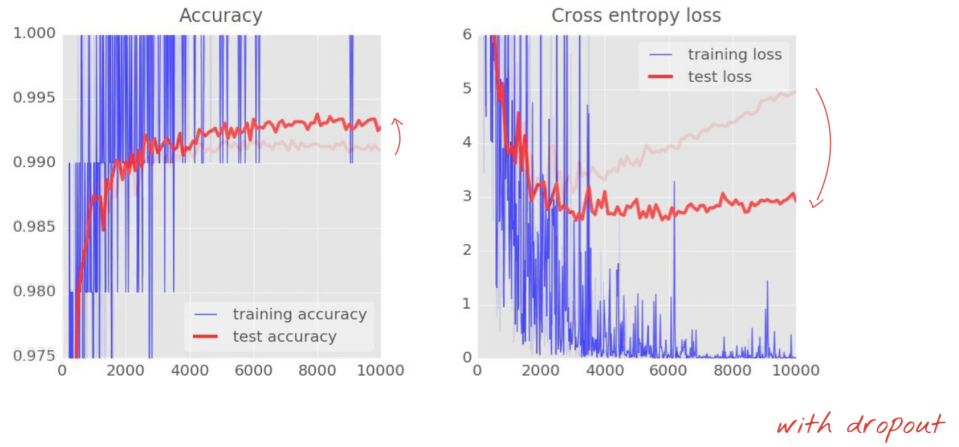


图 3-36 增强卷积网络：过拟合问题明显改善

第 II 部分

系统架构

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

- Martin Flower

4

系统架构

本章将阐述 TensorFlow 的系统架构，并一个简单的例子，讲述图结构的变换过程。最后，通过挖掘会话管理的工作机制，加深理解 TensorFlow 运行时的工作机理。

4.1 系统架构

如图**??**（第**??**页）所示，TensorFlow 的系统结构以 C API 为界，将整个系统分为**前端**和**后端**两个子系统[1]。

1. 前端系统：提供编程模型，负责构造计算图；
2. 后端系统：提供运行时环境，负责执行计算图。

TensorFlow 的系统设计遵循良好的分层架构，后端系统的设计和实现可以进一步分解为 4 层。

1. 运行时：分别提供本地模式和分布式模式，并共享大部分设计和实现；
2. 计算层：由各个 OP 的 Kernel 实现组成；在运行时，Kernel 实现执行 OP 的具体数学运算；
3. 通信层：基于 gRPC 实现组件间的数据交换，并能够在支持 IB 网络的节点间实现 RDMA 通信；
4. 设备层：计算设备是 OP 执行的主要载体，TensorFlow 支持多种异构的计算设备类型。

从图操作的角度看待系统行为，TensorFlow 运行时就是完成计算图的构造、编排、及其运行。

1. 表达图：构造计算图，但不执行图；
2. 编排图：将计算图的节点以最佳的执行方案部署在集群中各个计算设备上执行；

[1]事实上，后端系统中也存在 Client 的代码，前端系统是 TensorFlow 对外的编程接口。在后面的章节，将详细地讨论这个问题。

3. 运行图：按照拓扑排序执行图中的节点，并启动每个 OP 的 Kernel 计算。

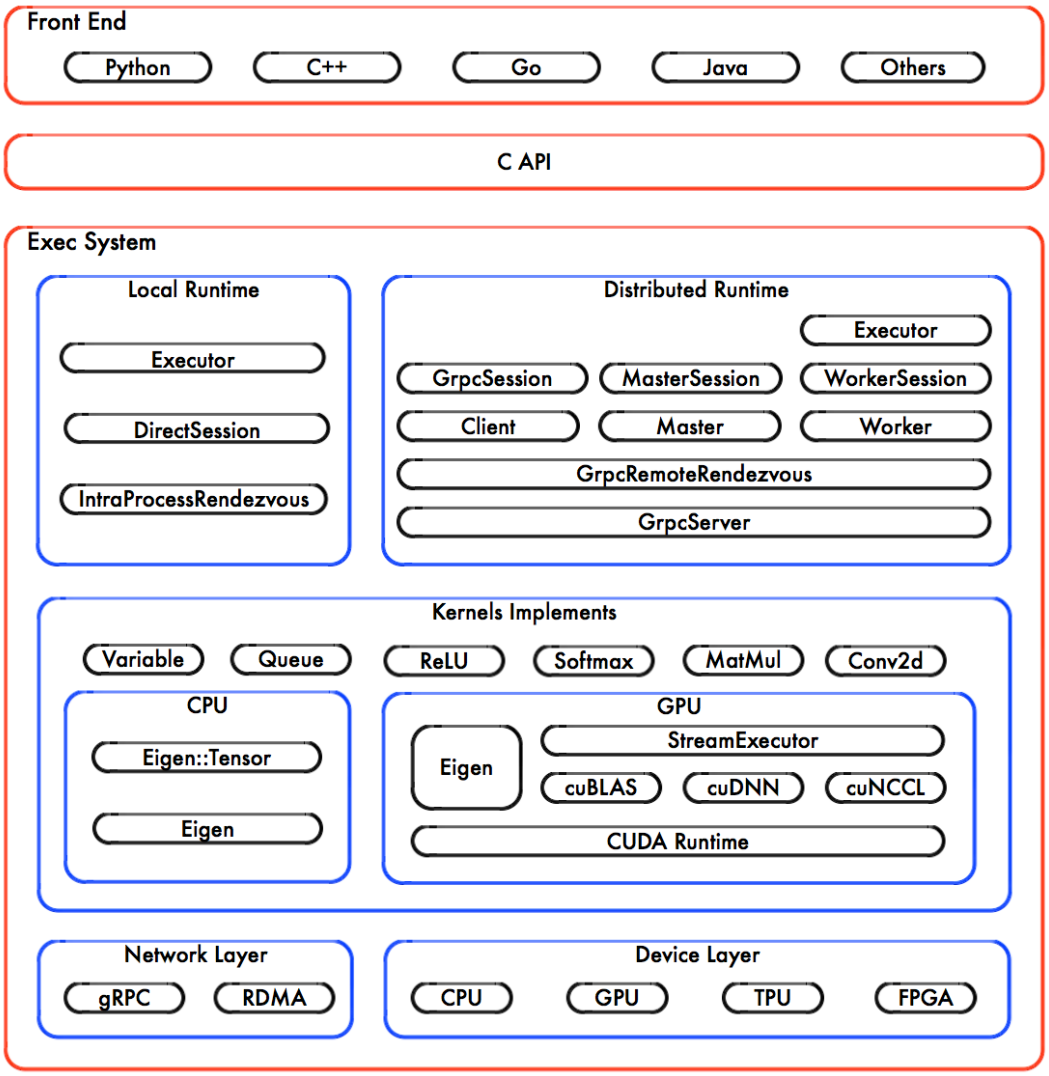


图 4-1 TensorFlow 系统架构

Client

Client 是前端系统的主要组成部分，它是一个支持多语言的编程环境。Client 基于 TensorFlow 的编程接口，构造计算图。目前，TensorFlow 支持 Python 和 C++ 的编程接口较为完善，尤其对 Python 的 API 支持最为全面。并且，对其他编程语言的 API 支持日益完善。

此时，TensorFlow 并未执行任何的图计算，直至与后台计算引擎建立 Session，并以 Session 为桥梁，建立 Client 与 Master 之间的通道，并将 Protobuf 格式的 GraphDef 序列化后传递给 Master，启动计算图的执行过程。

Master

在分布式的运行时环境中，Client 执行 `Session.run` 时，传递整个计算图给后端的 Master。此时，计算图是完整的，常称为 Full Graph。随后，Master 根据 `Session.run` 传递给它的 `fetches`，`feeds` 参数列表，反向遍历 Full Graph，并按照依赖关系，对其实施剪枝，最终计算得到最小的依赖子图，常称为 Client Graph。

接着，Master 负责将 Client Graph 按照任务的名称分裂 (`SplitByTask`) 为多个 Graph Partition；其中，每个 Worker 对应一个 Graph Partition。随后，Master 将 Graph Partition 分别注册到相应的 Worker 上，以便在不同的 Worker 上并发执行这些 Graph Partition。最后，Master 将通知所有 Work 启动相应 Graph Partition 的执行过程。

其中，Work 之间可能存在数据依赖关系，Master 并不参与两者之间的数据交换，它们两两之间互相通信，独立地完成交换数据，直至完成所有计算。

Worker

对于每一个任务，TensorFlow 都将启动一个 Worker 实例。Worker 主要负责如下 3 个方面的职责：

1. 处理来自 Master 的请求；
2. 对注册的 Graph Partition 按照本地计算设备集实施二次分裂 (`SplitByDevice`)，并通知各个计算设备并发执行各个 Graph Partition；
3. 按照拓扑排序算法在某个计算设备上执行本地子图，并调度 OP 的 Kernel 实现；
4. 协同任务之间的数据通信。

首先，Worker 收到 Master 发送过来的图执行命令，此时的计算图相对于 Worker 是完整的，也称为 Full Graph，它对应于 Master 的一个 Graph Partition。随后，Worker 根据当前可用的硬件环境，包括 (GPU/CPU) 资源，按照 OP 设备的约束规范，再将图分裂 (`SplitByDevice`) 为多个 Graph Partition；其中，每个计算设备对应一个 Graph Partition。接着，Worker 启动所有的 Graph Partition 的执行。最后，对于每一个计算设备，Worker 将按照计算图中节点之间的依赖关系执行拓扑排序算法，并依次调用 OP 的 Kernel 实现，完成 OP 的运算 (一种典型的多态实现技术)。

其中，Worker 还要负责将 OP 运算的结果发送到其他的 Worker 上去，或者接受来自其他 Worker 发送给它的运算结果，以便实现 Worker 之间的数据交互。TensorFlow 特化实现了源设备和目标设备间的 Send/Recv。

1. 本地 CPU 与 GPU 之间，使用 `cudaMemcpyAsync` 实现异步拷贝；
2. 本地 GPU 之间，使用端到端的 DMA 操作，避免主机端 CPU 的拷贝。

对于任务间的通信，TensorFlow 支持多种通信协议。

1. gRPC over TCP；
2. RDMA over Converged Ethernet。

此外，TensorFlow 已经初步开始支持 cuNCCL 库，用于改善多 GPU 间的通信。

Kernel

Kernel 是 OP 在某种硬件设备的特定实现，它负责执行 OP 的具体运算。目前，TensorFlow 系统中包含 200 多个标准的 OP，包括数值计算，多维数组操作，控制流，状态管理等。

一般每一个 OP 根据设备类型都会存在一个优化了的 Kernel 实现。在运行时，运行时根据 OP 的设备约束规范，及其本地设备的类型，为 OP 选择特定的 Kernel 实现，完成该 OP 的计算。

其中，大多数 Kernel 基于 `Eigen::Tensor` 实现。`Eigen::Tensor` 是一个使用 C++ 模板技术，为多核 CPU/GPU 生成高效的并发代码。但是，TensorFlow 也可以灵活地直接使用 cuDNN, cuNCCL, cuBLAS 实现更高效的 Kernel。

此外，TensorFlow 实现了矢量化技术，在高吞吐量、以数据为中心的应用需求中，及其移动设备中，实现更高效的推理。如果对于复合 OP 的子计算过程很难表示，或执行效率低下，TensorFlow 甚至支持更高效的 Kernel 注册，其扩展性表现非常优越。

4.2 图控制

通过一个最简单的例子，进一步抽丝剥茧，逐渐挖掘出 TensorFlow 计算图的控制与运行机制。

组建集群

如图**??**（第**??**页）所示。假如存在一个简单的分布式环境：1 PS + 1 Worker，并将其划分为两个任务：

1. ps0*:* 使用`/job:ps/task:0` 标记，负责模型参数的存储和更新；
2. worker0*:* `/job:worker/task:0` 标记，负责模型的训练。

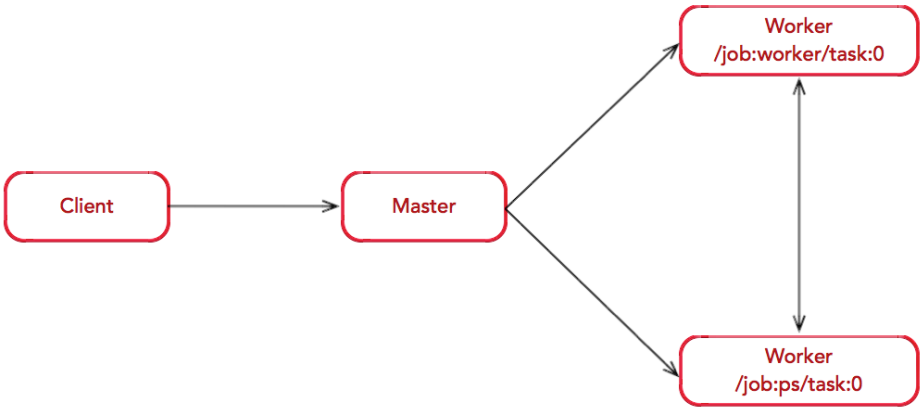


图 4-2 TensorFlow 集群：1 PS + 1 Worker

图构造

如图??（第??页）所示。Client 构建了一个简单计算图；首先，将 w 与 x 进行矩阵相乘，再与截距 b 按位相加，最后更新至 s 中。

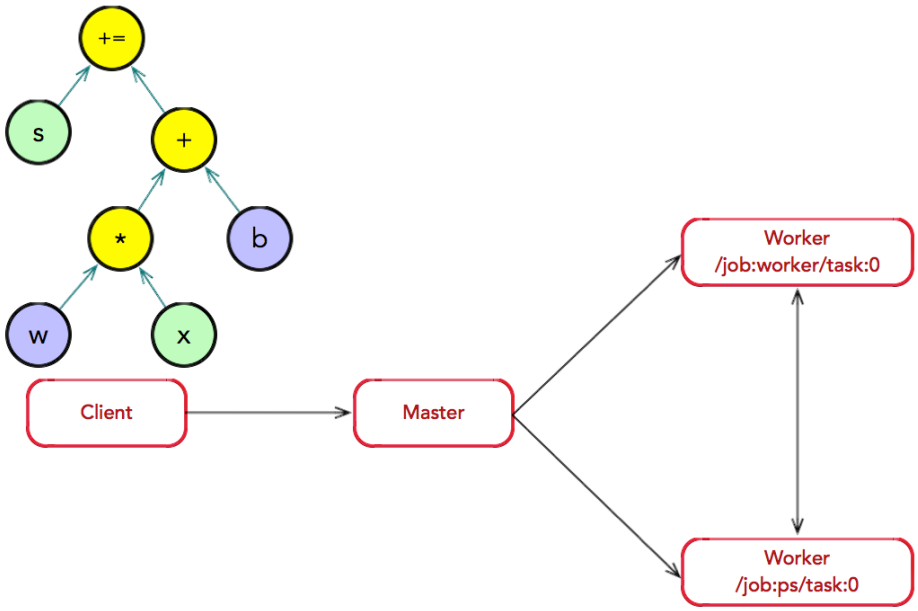


图 4-3 图构造

图执行

如图??（第??页）所示。首先，Client 创建一个 `Session` 实例，建立与 Master 之间的通道；接着，Client 通过调用 `Session.run` 将计算图传递给 Master。

随后，Master 便开始启动一次 Step 的图计算过程。在执行之前，Master 会实施一系

列优化技术，例如**公共表达式消除**，**常量折叠**等。最后，Master 负责任务之间的协同，执行优化后的计算图。

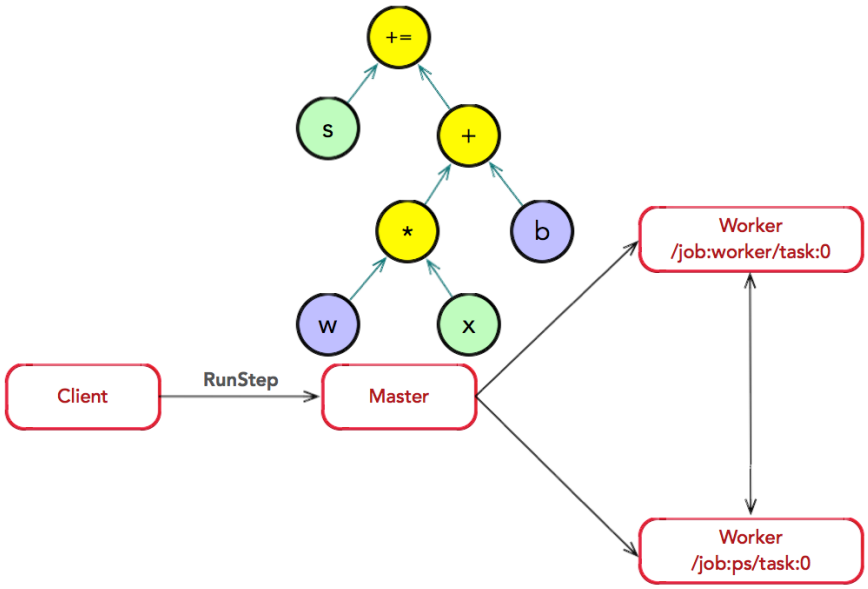


图 4-4 图执行

图分裂

如图**??**（第**??**页）所示，存在一种合理的图划分算法。Master 将模型参数相关的 OP 划分为一组，并放置在 ps0 任务上；其他 OP 划分为另外一组，放置在 worker0 任务上执行。

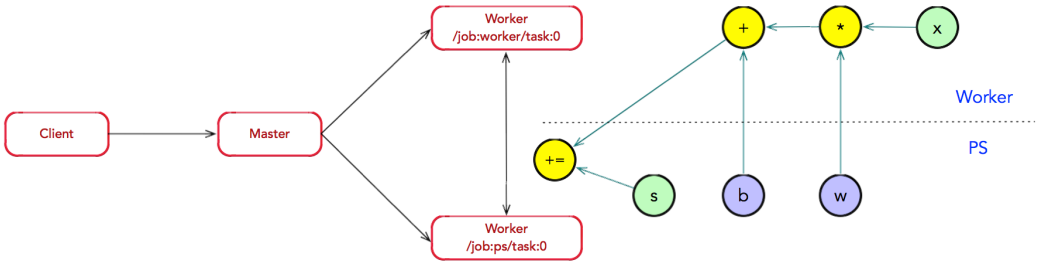


图 4-5 图分裂：按任务划分

子图注册

如图**??**（第**??**页）所示。在图分裂过程中，如果计算图的边跨越节点或设备，Master 将该边实施分裂，在两个节点或设备之间插入 Send 和 Recv 节点，实现数据的传递。

其中，`Send` 和 `Recv` 节点也是 OP，只不过它们是两个特殊的 OP，由内部运行时管理

和控制，对用户不可见；并且，它们仅用于数据的通信，并没有任何数据计算的逻辑。

最后，Master 通过调用 `RegisterGraph` 接口，将子图注册给相应的 Worker 上，并由相应的 Worker 负责执行运算。

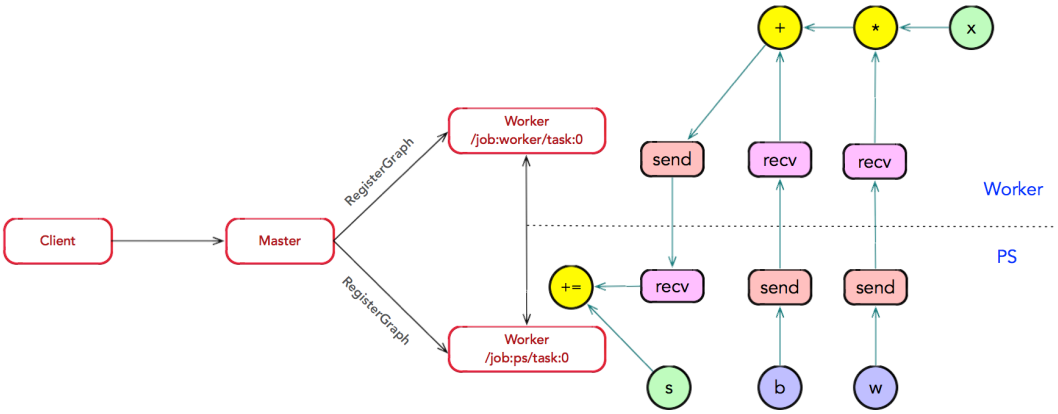


图 4-6 子图注册：插入 Send 和 Recv 节点

子图运算

如图**??**（第**??**页）所示。Master 通过调用 `RunGraph` 接口，通知所有 Worker 执行子图运算。其中，Worker 之间可以通过调用 `RecvTensor` 接口，完成数据的交换。

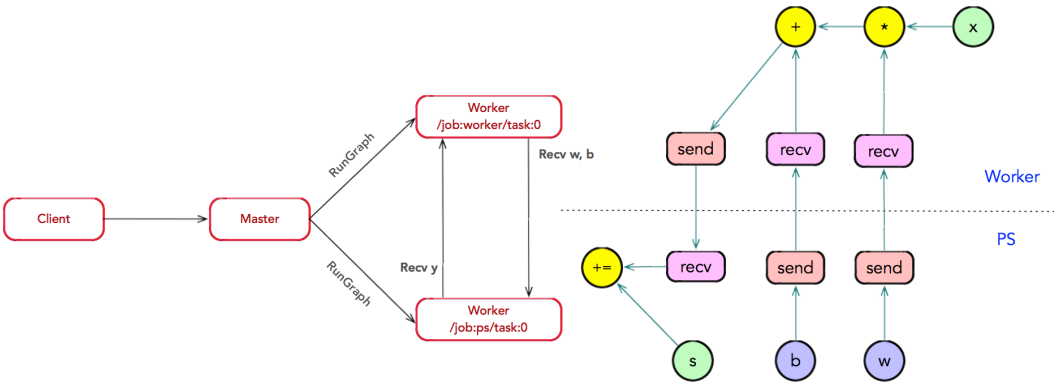


图 4-7 子图执行

4.3 会话管理

接下来，通过概述会话的整个生命周期过程，及其与图控制之间的关联关系，进一步揭开运行时的内部运行机制。

创建会话

首先，Client 首次执行 `tf.Session.run` 时，会将整个图序列化后，通过 gRPC 发送 `CreateSessionRequest` 消息，将图传递给 Master。

随后，Master 创建一个 `MasterSession` 实例，并用全局唯一的 `handle` 标识，最终通过 `CreateSessionResponse` 返回给 Client。如图**??**（第**??**页）所示。

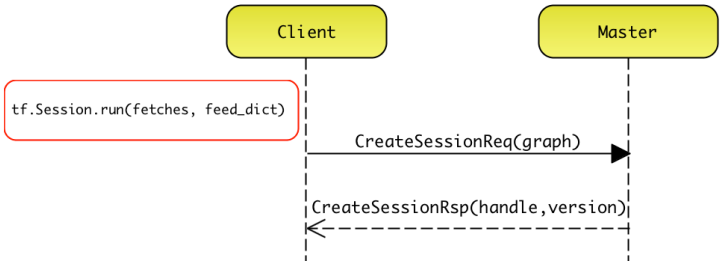


图 4-8 创建会话

迭代运行

随后，Client 会启动迭代执行的过程，并称每次迭代为一次 Step。此时，Client 发送 `RunStepRequest` 消息给 Master，消息携带 `handle` 标识，用于 Master 索引相应的 `MasterSession` 实例。如图**??**（第**??**页）所示。

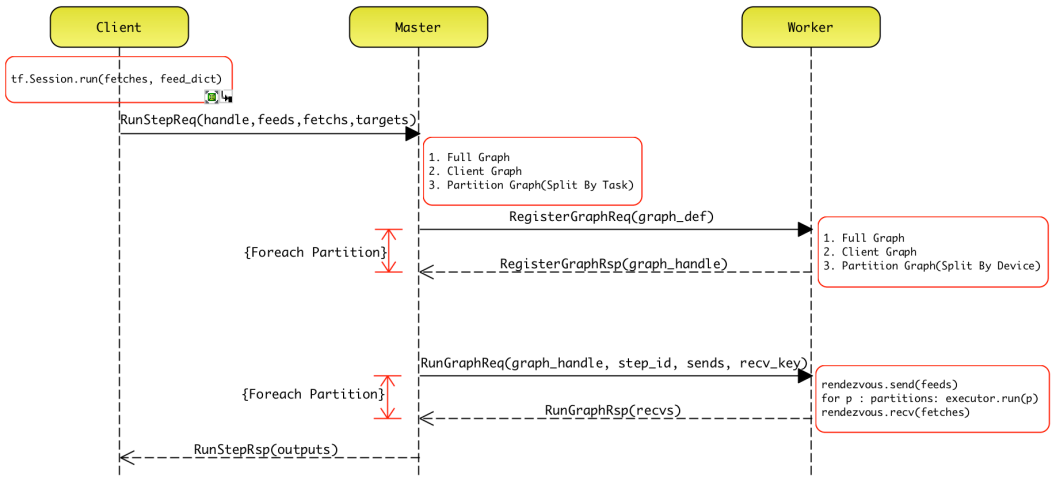


图 4-9 迭代执行

注册子图

Master 收到 `RunStepRequest` 消息后，将执行图剪枝，分裂，优化等操作。最终按照任务 (Task)，将图划分为多个子图片段 (Graph Partition)。随后，Master 向各个 Worker 发

送 RegisterGraphRequest 消息，将子图片段依次注册到各个 Worker 节点上。

当 Worker 收到 RegisterGraphRequest 消息后，再次实施分裂操作，最终按照设备 (Device)，将图划分为多个子图片段 (Graph Partition)。[1]

当 Worker 完成子图注册后，通过返回 RegisterGraphReponse 消息，并携带 graph_handle 标识。这是因为 Worker 可以并发注册并运行多个子图，每个子图使用 graph_handle 唯一标识。

运行子图

Master 完成子图注册后，将广播所有 Worker 并发执行所有子图。这个过程是通过 Master 发送 RunGraphRequest 消息给 Worker 完成的。其中，消息中携带 (session_handle, graph_handle, step_id) 三元组的标识信息，用于 Worker 索引相应的子图。

Worker 收到消息 RunGraphRequest 消息后，Worker 根据 graph_handle 索引相应的子图。最终，Worker 启动本地所有计算设备并发执行所有子图。其中，每个子图放置在单独的 Executor 中执行，Executor 将按照拓扑排序算法完成子图片段的计算。上述算法可以形式化地描述为如下代码。

```
def run_partitions(rendezvous, executors_and_partitions, inputs, outputs):
  rendezvous.send(inputs)
  for (executor, partition) in executors_and_partitions:
    executor.run(partition)
  rendezvous.recv(outputs)
```

交换数据

如果两个设备之间需要交换数据，则通过插入 Send/Recv 节点完成的。特殊地，如果两个 Worker 之间需要交换数据，则需要涉及跨进程间的通信。

此时，需要通过接收端主动发送 RecvTensorRequest 消息到发送方，再从发送方的信箱里取出对应的 Tensor，并通过 RecvTensorResponse 返回。如图**??**（第**??**页）所示。

关闭会话

当计算完成后，Client 向 Master 发送 CloseSessionReq 消息。Master 收到消息后，开始释放 MasterSession 所持有的所有资源。如图**??**（第**??**页）所示。

[1]在分布式运行时，图分裂经过两级分裂过程。在 Master 上按照任务分裂，而在 Worker 按照设备分裂。因此，得到结果都称为子图片段，它们仅存在范围，及其大小的差异。

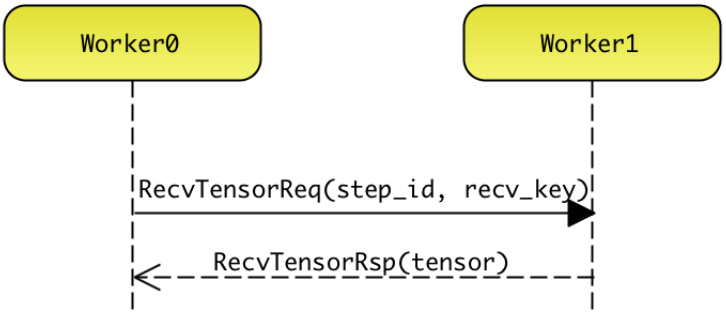


图 4-10 Worker 之间的数据交换

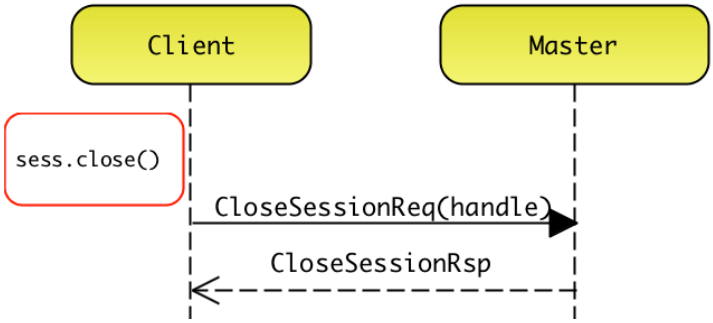


图 4-11 关闭会话

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

- Martin Flower

5

C API：分水岭

本章通过客户端 Session 生命周期的实现为例，揭示前端 Python 与后端 C++ 系统的实现通道，揭示 TensorFlow 多语言编程的奥秘。

5.1 Swig：幕后英雄

前端多语言编程环境与后端 C++ 实现系统的通道归功于 Swig 的包装器。TensorFlow 使用 Bazel 的构建工具，在系统编译之前启动 Swig 的代码生成过程，通过 `tensorflow.i` 自动生成了两个适配 (Wrapper) 文件：

1. `pywrap_tensorflow_internal.py`：负责对接上层 Python 调用；

2. `pywrap_tensorflow_internal.cc`：负责对接下层 C API 调用。

如图**??**（第**??**页）所示，`pywrap_tensorflow_internal.py` 模块首次被导入时，自动地加载 `_pywrap_tensorflow_internal.so` 的动态链接库；其中，`_pywrap_tensorflow_internal.so` 包含了整个 TensorFlow 运行时的所有符号。在 `pywrap_tensorflow_internal.cc` 的实现中，静态注册了一个函数符号表，实现了 Python 函数名到 C 函数名的二元关系。在运行时，按照 Python 的函数名称，匹配找到对应的 C 函数实现，最终实现 Python 到 `c_api.c` 具体实现的调用关系。

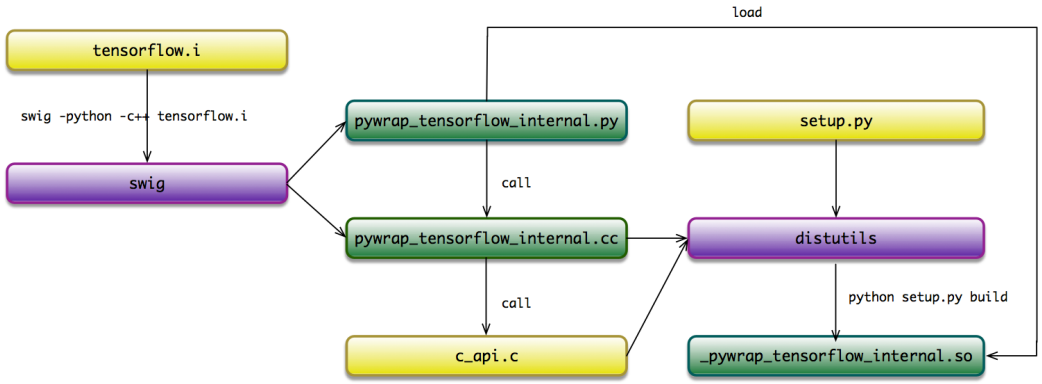


图 5-1 Swig 代码生成器

其中，Bazel 的生成规则定义于`//tensorflow/python:pywrap_tensorflow_internal`，如下代码所示。

```
tf_py_wrap_cc(
    name = "pywrap_tensorflow_internal",
    srcs = ["tensorflow.i"],
    swig_includes = [
        "client/device_lib.i",
        "client/events_writer.i",
        "client/tf_session.i",
        "client/tf_sessionrun_wrapper.i",
        "framework/cpp_shape_inference.i",
        "framework/python_op_gen.i",
        "grappler/cost_analyzer.i",
        "grappler/model_analyzer.i",
        "grappler/tf_optimizer.i",
        "lib/core/py_func.i",
        "lib/core/strings.i",
        "lib/io/file_io.i",
        "lib/io/py_record_reader.i",
        "lib/io/py_record_writer.i",
        "platform/base.i",
        "pywrap_tfe.i",
        "training/quantize_training.i",
        "training/server_lib.i",
        "util/kernel_registry.i",
        "util/port.i",
        "util/py_checkpoint_reader.i",
        "util/stat_summarizer.i",
        "util/tfprof.i",
        "util/transform_graph.i",
    ]
)
```

下文以客户端 Session 生命周期的实现为例，揭示前端 Python 与后端 C++ 系统的实现通道。

5.2 会话控制

严格意义上，C API 并非是 Client 与 Master 的分界线。如图??（第??页）所示，Client 存在部分 C++ 实现，即 `tensorflow::Session`。其中，`tf.Session` 实例直接持有 `tensorflow::Session` 实例的句柄。在实际运行时环境中，`tensorflow::Session` 可能存在多种实现。例如，`DirectSession` 负责**本地模式**的会话控制。而 `GrpcSession` 负责基于 gRPC 协议的**分布式模式**的会话控制。一般地，用户使用的是 `tf.Session` 实施编程，而非 `tensorflow::Session`。

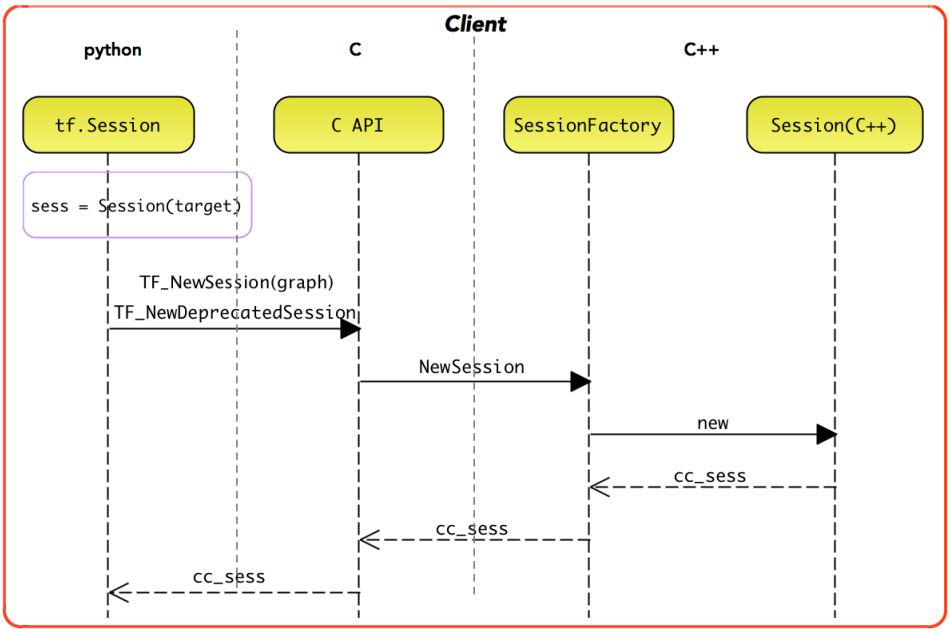


图 5-2 客户端：tensorflow::Session 实例创建过程

5.3 会话生命周期

会话的生命周期包括会话的创建，创建计算图，扩展计算图，执行计算图，关闭会话，销毁会话的基本过程。在前端 Python 和后端 C++ 表现为两套相兼容的接口实现。

Python 前端

如图??（第??页）所示，在 Python 前端，`Session` 的生命周期主要体现在：

1. 创建 `Session(target)`；
2. 迭代执行 `Session.run(fetches, feed_dict)`；
   (a) `Session._extend_graph(graph)`；
   (b) `Session.TF_Run(feeds, fetches, targets)`；

3. 关闭 Session；

4. 销毁 Session；

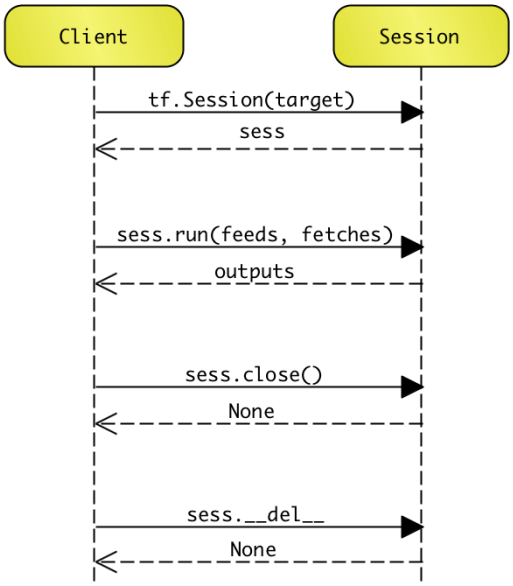


图 5-3 Python: Session 生命周期

例如，此处创建了本地模式的 Session 实例，并启动 mnist 的训练过程。

```
sess = tf.Session()
for _ in range(1000):
  batch_xs, batch_ys = mnist.train.next_batch(100)
  sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
sess.close()
```

C++ 后端

相应地，在 C++ 后端，Session 的生命周期主要体现在：

1. 根据 target 多态创建 Session；

2. Session.Create(graph)：有且仅有一次；

3. Session.Extend(graph)：零次或多次；

4. 迭代执行 Session.Run(inputs, outputs, targets)；

5. 关闭 Session.Close；

6. 销毁 Session 对象。

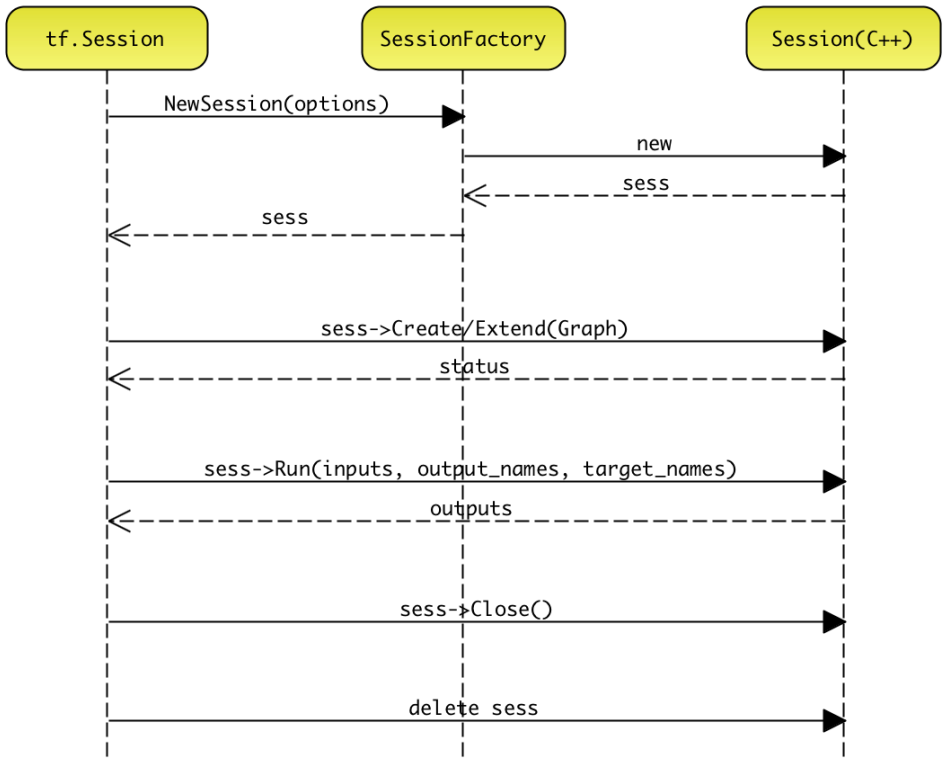


图 5-4 C++: Session 生命周期

例如，此处创建了本地模式的 DirectSession 实例，并启动计算图的执行过程。

```
// create/load graph ...
tensorflow::GraphDef graph;

// local runtime, target is ""
tensorflow::SessionOptions options;

// create Session
std::unique_ptr<tensorflow::Session>
sess(tensorflow::NewSession(options));

// create graph at initialization.
tensorflow::Status s = sess->Create(graph);
if (!s.ok()) { ... }

// run step
std::vector<tensorflow::Tensor> outputs;
s = session->Run(
  {},               // inputs is empty
  {"output:0"},     // outputs names
  {"update_state"}, // target names
  &outputs);        // output tensors
if (!s.ok()) { ... }

// close
session->Close();
```

5.4 创建会话

下面介绍 Session 创建的详细过程，从 Python 前端为起点，通过 Swig 自动生成的 Python-C++ 的包装器，并以此为媒介，实现了 Python 到 TensorFlow 的 C API 的调用。其中，C API 是前端系统与后端系统的分水岭。

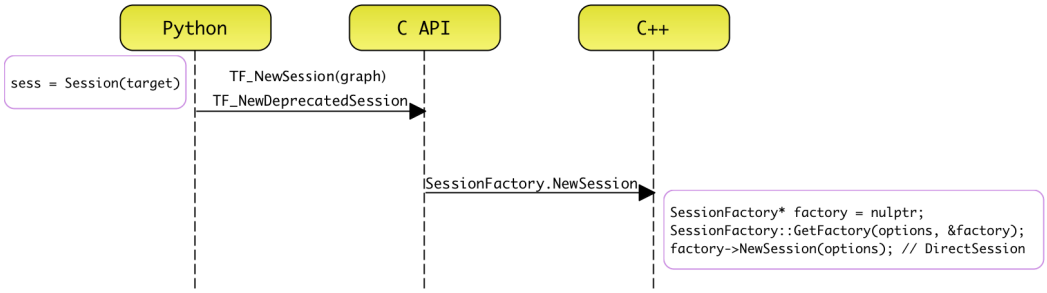


图 5-5 创建会话

编程接口

当 Client 要启动计算图的执行过程时，先创建了一个 `Session` 实例，进而调用父类 `BaseSession` 的构造函数。

示例代码 5-1 tensorflow/python/client/session.py

```
class Session(BaseSession):
  def __init__(self, target='', graph=None, config=None):
    super(Session, self).__init__(target, graph, config=config)
    self._default_graph_context_manager = None
    self._default_session_context_manager = None
```

在 `BaseSession` 的构造函数中，将调用 `pywrap_tensorflow` 模块中的函数。其中，`TF_NewDeprecatedSession` 是遗留下来的，已被废弃的接口实现。在新的 API 中，直接将图实例传递给后端 C++，避免了前后端图实例序列化的开销。

示例代码 5-2 tensorflow/python/client/session.py

```
from tensorflow.python import pywrap_tensorflow as tf_session

class BaseSession(SessionInterface):
  def __init__(self, target='', graph=None, config=None):
    # default graph
    if graph is None:
      self._graph = ops.get_default_graph()
    else:
      self._graph = graph

    # handle to tensorflow::Session
    self._session = None
    opts = tf_session.TF_NewSessionOptions(target=self._target,
```

```
                                                config=config)
    try:
      with errors.raise_exception_on_not_ok_status() as status:
        if self._created_with_new_api:
          self._session = tf_session.TF_NewSession(
              self._graph._c_graph, opts, status)
        else:
          self._session = tf_session.TF_NewDeprecatedSession(opts, status)
    finally:
      tf_session.TF_DeleteSessionOptions(opts)
```

如图**??**（第**??**页）所示，ScopedTFGraph 是对 TF_Graph 的包装器，完成类似于 C++ 的 RAII 的工作机制。而 TF_Graph 持有 ternsorflow::Graph 实例。其中，self._graph._c_graph 返回一个 TF_Graph 实例，后者通过 C API 创建的图实例。

示例代码 5-3 tensorflow/python/framework/ops.py

```
class Graph(object):
  def __init__(self):
    if _USE_C_API:
      self._scoped_c_graph = c_api_util.ScopedTFGraph()
    else:
      self._scoped_c_graph = None

  def _c_graph(self):
    if self._scoped_c_graph:
      return self._scoped_c_graph.graph
    return None
```

示例代码 5-4 tensorflow/python/framework/c_api_util.py

```
class ScopedTFGraph(object):
  def __init__(self):
    self.graph = c_api.TF_NewGraph()

  def __del__(self):
    if c_api.TF_DeleteGraph is not None:
      c_api.TF_DeleteGraph(self.graph)
```

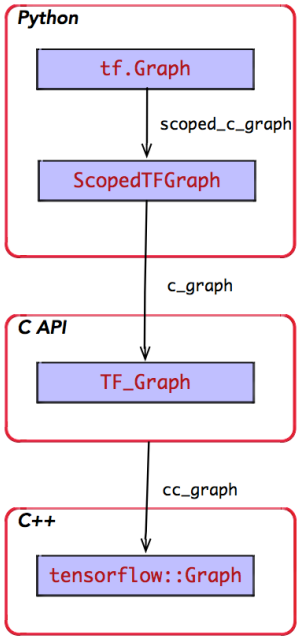


图 5-6 前后端：图实例传递

Python 包装器

在 pywrap_tensorflow 模块中，通过 _pywrap_tensorflow_internal 的转发，实现了从 Python 到动态连接库 _pywrap_tensorflow_internal.so 的函数调用。

示例代码 5-5 tensorflow/bazel-bin/tensorflow/python/pywrap_tensorflow_internal.py

```
def TF_NewDeprecatedSession(opts, status):
  return _pywrap_tensorflow_internal.TF_NewDeprecatedSession(opts, status)

def TF_NewSession(graph, opts, status):
  return _pywrap_tensorflow_internal.TF_NewSession(graph, opts, status)
```

C++ 包装器

在 pywrap_tensorflow_internal.cc 的具体实现中，静态注册了函数调用的符号表，实现 Python 的函数名称到 C++ 函数实现的具体映射。

示例代码 5-6 tensorflow/bazel-bin/tensorflow/python/pywrap_tensorflow_internal.cc

```
static PyMethodDef SwigMethods[] = {
  // ...
  { (char *)"TF_NewDeprecatedSession",
    _wrap_TF_NewDeprecatedSession, METH_VARARGS, NULL},

  { (char *)"TF_NewSession",
```

```
        _wrap_TF_NewSession, METH_VARARGS, NULL},
    };
```

最终，_wrap_TF_NewSession/_wrap_TF_NewDeprecatedSession 将分别调用 c_api.h 对其开放的 API 接口：TF_NewSession/TF_NewDeprecatedSession。也就是说，自动生成的 pywrap_tensorflow_internal.cc 仅仅负责 Python 函数到 C/C++ 函数调用的转发，最终将调用底层 C 系统向上提供的 API 接口。

C API

c_api.h 是 TensorFlow 的后端执行系统面向前端开放的公共 API 接口。其中，新的接口实现采用了引用计数的技术，实现图实例在多个 Session 实例中共享。

示例代码 5-7 tensorflow/c/c_api.c

```
TF_Session* TF_NewSession(TF_Graph* graph, const TF_SessionOptions* opt,
                          TF_Status* status) {
  Session* session;
  status->status = NewSession(opt->options, &session);
  if (status->status.ok()) {
    if (graph != nullptr) {
      mutex_lock l(graph->mu);
      graph->num_sessions += 1;
    }
    return new TF_Session(session, graph);
  } else {
    return nullptr;
  }
}

TF_DeprecatedSession* TF_NewDeprecatedSession(const TF_SessionOptions* opt,
                                              TF_Status* status) {
  Session* session;
  status->status = NewSession(opt->options, &session);
  if (status->status.ok()) {
    return new TF_DeprecatedSession({session});
  } else {
    DCHECK_EQ(nullptr, session);
    return nullptr;
  }
}
```

后端系统

NewSession 将根据前端传递的 target，使用 SessionFactory 多态创建不同类型的 tensorflow::Session 实例。

示例代码 5-8 tensorflow/c/c_api.c

```
Status NewSession(const SessionOptions& options, Session** out_session) {
  SessionFactory* factory;
  Status s = SessionFactory::GetFactory(options, &factory);
```

```
    if (!s.ok()) {
      *out_session = nullptr;
      return s;
    }
    *out_session = factory->NewSession(options);
    if (!*out_session) {
      return errors::Internal("Failed to create session.");
    }
    return Status::OK();
  }
```

工厂方法

在后端 C++ 实现中，tensorflow::Session 的创建使用了抽象工厂方法。如果 SessionOptions 中的 target 为空字符串 (默认的)，则创建 DirectSession 实例，启动本地运行模式；如果 SessionOptions 中的 target 以 grpc://开头，则创建 GrpcSession 实例，启动基于 RPC 的分布式运行模式。如图**??**（第**??**页）所示。

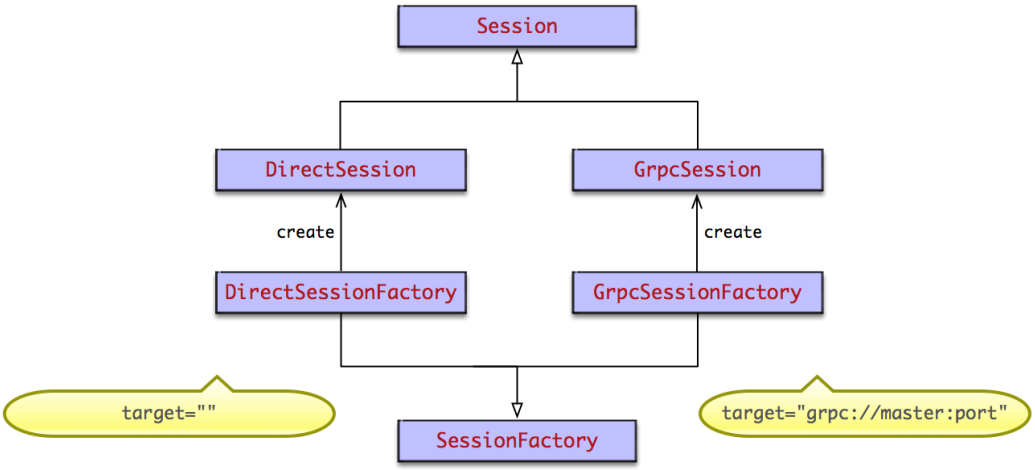


图 5-7 tensorflow::Session 创建：抽象工厂方法

5.5 创建/扩展图

在既有的接口实现中，需要将图构造期构造好的图序列化，并传递给后端 C++ 系统。而在新的接口实现中，无需实现图的创建或扩展。因为在创建 OP 时，节点实时添加至后端 C++ 系统的图实例中。但是，既有接口实现，对于理解系统行为较为重要，在次做简单地阐述。

Python 前端将迭代调用 Session.run 接口，将构造好的计算图，以 GraphDef 的形式发送给 C++ 后端。其中，前端每次调用 Session.run 接口时，都会试图将新增节点的计算图发送给后端系统，以便将新增节点的计算图 Extend 到原来的计算图中。特殊地，在首次调用 Session.run 时，将发送整个计算图给后端系统。

后端系统首次调用 Session.Extend 时，转调 (或等价实现)Session.Create。以后，后端系统每次调用 Session.Extend 时将真正执行 Extend 的语义，将新增的计算图的节点追加至原来的计算图中。

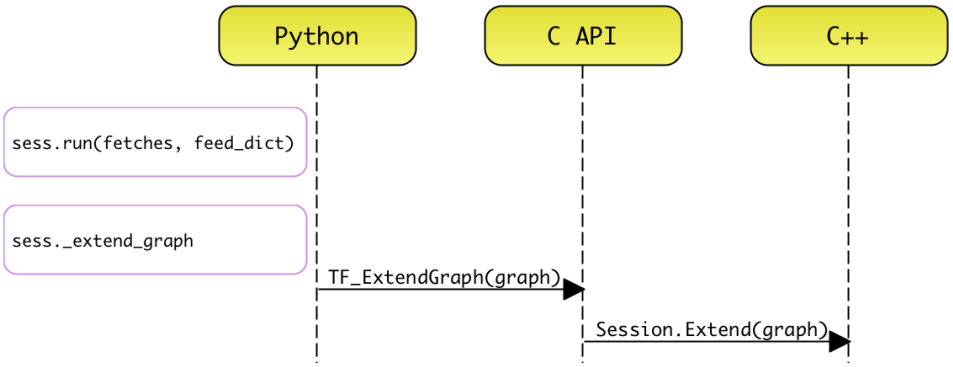


图 5-8 创建图

编程接口

在既有的接口实现中，通过 _extend_graph 实现图实例的扩展。

示例代码 5-9 tensorflow/python/client/session.py

```
class Session(BaseSession):
  def run(self, fetch_list, feed_dict=None, options=None, run_metadata=None):
    # ignores implements...
    self._extend_graph()
    # ignores implements...
```

在首次调用 self._extend_graph 时，或者有新的节点被添加至计算图中时，对计算图 GraphDef 实施序列化操作，最终触发 tf_session.TF_ExtendGraph 的调用。

示例代码 5-10 tensorflow/python/client/session.py

```
from tensorflow.python import pywrap_tensorflow as tf_session

class Session(BaseSession):
  def _extend_graph(self):
    if self._created_with_new_api: return

    with self._extend_lock:
      if self._graph.version > self._current_version:
        graph_def, self._current_version = self._graph._as_graph_def(
            from_version=self._current_version,
            add_shapes=self._add_shapes)

        with errors.raise_exception_on_not_ok_status() as status:
          tf_session.TF_ExtendGraph(
              self._session, graph_def.SerializeToString(), status)
```

Python 包装器

示例代码 5-11 tensorflow/bazel-bin/tensorflow/python/pywrap_tensorflow_internal.py

```
def TF_ExtendGraph(sess, graph_def, status):
  return _pywrap_tensorflow.TF_ExtendGraph(sess, graph_def, status)
```

C++ 包装器

示例代码 5-12 tensorflow/bazel-bin/tensorflow/python/pywrap_tensorflow_internal.cc

```
static PyMethodDef SwigMethods[] = {
  // ignore implements...
  { (char *)"TF_ExtendGraph",
    _wrap_TF_ExtendGraph, METH_VARARGS, NULL},
};
```

C API

TF_ExtendGraph 是 C API 对接上层编程环境的接口。首先，它完成计算图 GraphDef 的反序列化，最终调用 tensorflow::Session 的 Extend 接口。

示例代码 5-13 tensorflow/c/c_api.c

```
void TF_ExtendGraph(TF_DeprecatedSession* sess,
  const void* proto, size_t proto_len, TF_Status* status) {
  GraphDef g;
  if (!tensorflow::ParseProtoUnlimited(&g, proto, proto_len)) {
    status->status = InvalidArgument("Invalid GraphDef");
    return;
  }
  status->status = sess->session->Extend(g);
}
```

后端系统

tensorflow::Session 在运行时根据 Session 的动态类型，将多态地调用相应子类的实现。

示例代码 5-14 tensorflow/core/common_runtime/session.h

```
class Session {
public:
  virtual Status Create(const GraphDef& graph) = 0;
  virtual Status Extend(const GraphDef& graph) = 0;
};
```

其中，Create 表示在当前的 tensorflow::Session 实例上注册计算图，如果要注册新的计算图，需要关闭该 tensorflow::Session 对象。Extend 表示在 tensorflow::Session 实例上已注册的计算图上追加节点。Extend 首次执行时，等价于 Create 的语义，因为首次 Extend 时，已注册的计算图为空。事实上，系统就是按照如上方案实现的，此处以 GrpcSession 实现为例。

首次扩展图: GrpcSession

如果判断引用 Master 的 handle 不为空，则执行 Extend；否则，执行 Create 的语义，建立与 Master 的连接，并持有 MasterSession 的 handle。

示例代码 5-15 tensorflow/core/distributed_runtime/rpc/grpc_session.cc

```
Status GrpcSession::Extend(const GraphDef& graph) {
  CallOptions call_options;
  call_options.SetTimeout(options_.config.operation_timeout_in_ms());
  return ExtendImpl(&call_options, graph);
}

Status GrpcSession::ExtendImpl
  (CallOptions* call_options, const GraphDef& graph) {
  if (handle_is_empty()) {
    // Session was unitialized,
    // so simply initialize the session with 'graph'.
    return Create(graph);
  }
  // ignore implements...
}
```

5.6 迭代运行

如图??（第??页）所示，Python 前端 Session.run 实现将 fetches，feed_dict 传递给后端系统，后端系统调用 Session.Run 接口。后端系统的一次 Session.Run 执行常常被称为一次 Step。其中，Step 的执行过程是 TensorFlow 运行时的关键路径。

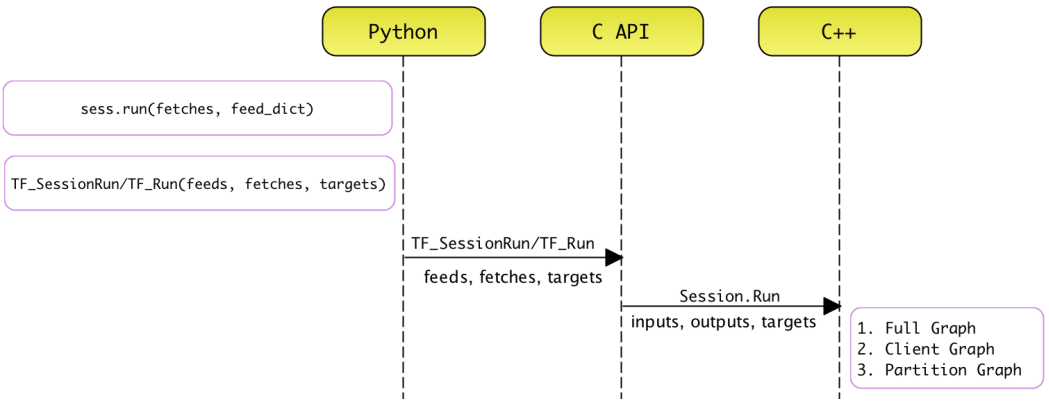


图 5-9 迭代执行

编程接口

当 Client 调用 Session.run 时，最终会调用 pywrap_tensorflow_internal 模块中的函数。

示例代码 5-16 tensorflow/python/client/session.py

```
from tensorflow.python import pywrap_tensorflow as tf_session

class Session(BaseSession):
  def run(self, fetch_list, feed_dict=None, options=None, run_metadata=None):
    # ignores other implements...
    self._extend_graph()
    with errors.raise_exception_on_not_ok_status() as status:
      if self._created_with_new_api:
        return tf_session.TF_SessionRun_wrapper(
            session, options, feed_dict, fetch_list, target_list,
            run_metadata, status)
      else:
        return tf_session.TF_Run(session, options,
                                 feed_dict, fetch_list, target_list,
                                 status, run_metadata)
```

Python 包装器

示例代码 5-17 tensorflow/bazel-bin/tensorflow/python/pywrap_tensorflow_internal.py

```
def TF_SessionRun_wrapper(session, run_options, inputs,
  outputs, targets, run_metadata, out_status):
  return _pywrap_tensorflow_internal.TF_SessionRun_wrapper(
    session, run_options, inputs, outputs, targets, run_metadata, out_status)

def TF_Run(sess, options, feeds, outputs,
  targets, status, run_metadata):
  return _pywrap_tensorflow.TF_Run(
    sess, options, feeds, outputs, targets, status, run_metadata)
```

C++ 包装器

示例代码 5-18 tensorflow/bazel-bin/tensorflow/python/pywrap_tensorflow_internal.cc

```
static PyMethodDef SwigMethods[] = {
  // ...
  { (char *)"TF_Run",
    _wrap_TF_Run, METH_VARARGS, NULL},

  { (char *)"TF_SessionRun_wrapper",
    _wrap_TF_SessionRun_wrapper, METH_VARARGS, NULL},
};
```

最终，_wrap_TF_Run/_wrap_TF_SessionRun_wrapper 将分别转调 C API 对应的 TF_Run/TF_SessionRun 接口函数。

C API

在既有的接口中，TF_Run 是 C API 对接上层编程环境的接口。首先，它完成输入数据从 C 到 C++ 的格式转换，并启动后台的 tensorflow::Session 的执行过程。当执行完成后，再将 outputs 的输出数据从 C++ 到 C 的格式转换。TF_SessionRun 与 TF_Run 工作机制差不多，在此不再冗述。

示例代码 5-19 tensorflow/c/c_api.c

```
void TF_Run(TF_DeprecatedSession* s,
  // session options
  const TF_Buffer* run_options,
  // Input tensors
  const char** c_input_names, TF_Tensor** c_inputs, int ninputs,
  // Output tensors
  const char** c_output_names, TF_Tensor** c_outputs, int noutputs,
  // Target nodes
  const char** c_target_oper_names, int ntargets,
  // run_metadata
  TF_Buffer* run_metadata, TF_Status* status) {
  // convert data format, ignore implements...
  s->session->Run(options_proto, input_names, output_names,
                  target_names, &outputs, &run_metadata);
  // store results in c_outputs...
}

void TF_SessionRun(TF_Session* session,
  const TF_Buffer* run_options,
  // Input tensors
  const TF_Output* inputs, TF_Tensor* const * input_values, int ninputs,
  // Output tensors
  const TF_Output* outputs, TF_Tensor** output_values, int noutputs,
  // Target nodes
  const TF_Operation* const* target_opers, int ntargets,
  // run_metadata
  TF_Buffer* run_metadata, TF_Status* status) {
  // ignore implements.
}
```

后端系统

tensorflow::Session 在运行时按照其动态类型，将多态地调用相应的子类实现。

示例代码 5-20 tensorflow/core/common_runtime/session.h

```
class Session {
public:
  virtual Status Run(
    const RunOptions& options,
    const vector<pair<string, Tensor> >& inputs,
    const vector<string>& output_names,
    const vector<string>& target_names,
    vector<Tensor>* outputs, RunMetadata* run_metadata) {
      return errors::Unimplemented(
        "Run with options is not supported for this session.");
  }
};
```

输入包括：

1. options：Session 的运行配置参数；
2. inputs：输入 Tensor 的名字列表；
3. output_names：输出 Tensor 的名字列表；
4. targets：无输出，待执行的 OP 的名字列表。

输出包括：

1. outputs：输出的 Tensor 列表；
2. run_metadata：运行时元数据的收集器。

其中，输出的 outputs 列表与输入的 output_names 一一对应，如果运行时因并发执行，导致 outputs 乱序执行，最终返回时需要对照输入的 output_names 名字列表，对 outputs 进行排序。

5.7 关闭会话

当计算图执行完毕后，需要关闭 tf.Session，以便释放后端的系统资源，包括队列，IO 等。会话关闭流程较为简单，如图**??**（第**??**页）所示。。

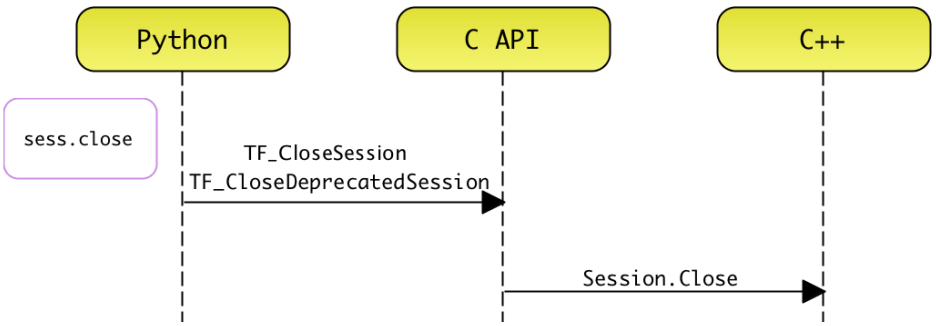


图 5-10 关闭会话

编程接口

当 Client 调用 Session.close 时，最终会调用 pywrap_tensorflow 模块中的函数：TF_CloseDeprecatedSession。

示例代码 5-21 tensorflow/python/client/session.py

```
from tensorflow.python import pywrap_tensorflow as tf_session
class Session(BaseSession):
```

```
  def close(self):
    if self._created_with_new_api:
      if self._session and not self._closed:
        self._closed = True
        with errors.raise_exception_on_not_ok_status() as status:
          tf_session.TF_CloseSession(self._session, status)
    else:
      with self._extend_lock:
        if self._opened and not self._closed:
          self._closed = True
          with errors.raise_exception_on_not_ok_status() as status:
            tf_session.TF_CloseDeprecatedSession(self._session, status)
```

Python 包装器

示例代码 5-22 tensorflow/bazel-bin/tensorflow/python/pywrap_tensorflow_internal.py

```
def TF_CloseSession(sess, status):
    return _pywrap_tensorflow_internal.TF_CloseSession(sess, status)

def TF_CloseDeprecatedSession(sess, status):
  return _pywrap_tensorflow.TF_CloseDeprecatedSession(sess, status)
```

C++ 包装器

_wrap_TF_CloseSession/_wrap_TF_CloseDeprecatedSession 将分别转调 C API 对应的 TF_CloseSession/TF_CloseDeprecatedSession 接口函数。

示例代码 5-23 tensorflow/bazel-bin/tensorflow/python/pywrap_tensorflow_internal.cc

```
static PyMethodDef SwigMethods[] = {
  // ...
  { (char *)"TF_CloseSession",
    _wrap_TF_CloseSession, METH_VARARGS, NULL},

  { (char *)"TF_CloseDeprecatedSession",
    _wrap_TF_CloseDeprecatedSession, METH_VARARGS, NULL},
};
```

C API

TF_CloseSession/TF_CloseDeprecatedSession 直接完成 tensorflow::Session 的关闭操作。

示例代码 5-24 tensorflow/c/c_api.c

```
void TF_CloseSession(TF_Session* s, TF_Status* status) {
  status->status = s->session->Close();
}
```

```
void TF_CloseDeprecatedSession(TF_DeprecatedSession* s, TF_Status* status) {
  status->status = s->session->Close();
}
```

后端系统

Session(C++) 在运行时其动态类型，将多态地调用相应的子类实现。

示例代码 5-25 tensorflow/core/common_runtime/session.h

```
class Session {
public:
  virtual Status Close() = 0;
};
```

5.8 销毁会话

当 tf.Session 不在被使用，由 Python 的 GC 释放。Session.__del__ 被调用后，将启动后台 tensorflow::Session 对象的析构过程。如图??（第??页）所示。

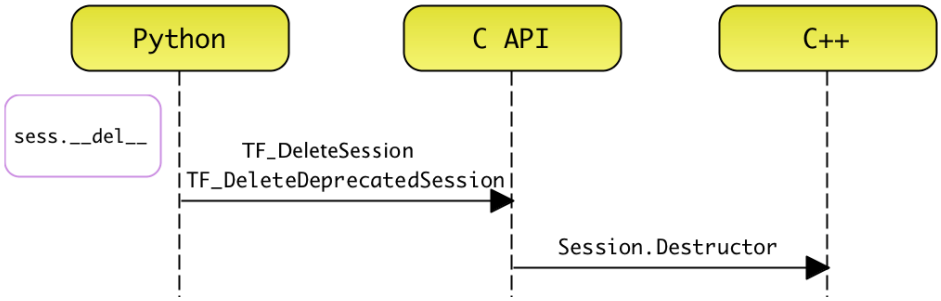


图 5-11 销毁会话

编程接口

当 Client 调用 Session.__del__ 时，先启动 Session.close 的调用，最终会调用 pywrap_tensorflow 模块中的函数 TF_DeleteSession/TF_DeleteDeprecatedSession。

示例代码 5-26 tensorflow/python/client/session.py

```
from tensorflow.python import pywrap_tensorflow as tf_session

class Session(BaseSession):
  def __del__(self):
    # 1. close session unconditionally.
    try:
      self.close()
```

```
      except Exception:
        pass
      # 2. delete session unconditionally.
      if self._session is not None:
        try:
          status = c_api_util.ScopedTFStatus()
          if self._created_with_new_api:
            tf_session.TF_DeleteSession(self._session, status)
          else:
            tf_session.TF_DeleteDeprecatedSession(self._session, status)
        except AttributeError:
          pass
        self._session = None
```

Python 包装器

示例代码 5-27 tensorflow/bazel-bin/tensorflow/python/pywrap_tensorflow_internal.py

```
def TF_DeleteSession(sess, status):
    return _pywrap_tensorflow_internal.TF_DeleteSession(sess, status)

def TF_DeleteDeprecatedSession(sess, status):
  return _pywrap_tensorflow.TF_DeleteDeprecatedSession(sess, status)
```

C++ 包装器

`_wrap_TF_DeleteSession/_wrap_TF_DeleteDeprecatedSession` 将分别转调 C API 对应的 `TF_DeleteSession/TF_DeleteDeprecatedSession` 接口函数。

示例代码 5-28 tensorflow/bazel-bin/tensorflow/python/pywrap_tensorflow_internal.cc

```
static PyMethodDef SwigMethods[] = {
  // ...
  { (char*)"TF_DeleteSession",
    _wrap_TF_DeleteSession, METH_VARARGS, NULL},

  { (char*)"TF_DeleteDeprecatedSession",
    _wrap_TF_DeleteDeprecatedSession, METH_VARARGS, NULL},
};
```

C API

`TF_DeleteDeprecatedSession` 直接完成 `tensorflow::Session` 对象的释放。而新的接口 `TF_DeleteSession` 实现中，当需要删除 `tensorflow::Session` 实例时，相应的图实例的计数器减 1。当计数器为 0 时，则删除该图实例；否则，不删除该图实例。

示例代码 5-29 tensorflow/c/c_api.c

```
void TF_DeleteSession(TF_Session* s, TF_Status* status) {
  status->status = Status::OK();
  TF_Graph* const graph = s->graph;
  if (graph != nullptr) {
    graph->mu.lock();
    graph->num_sessions -= 1;
    const bool del = graph->delete_requested && graph->num_sessions == 0;
    graph->mu.unlock();
    if (del) delete graph;
  }
  delete s->session;
  delete s;
}

void TF_DeleteDeprecatedSession(TF_DeprecatedSession* s, TF_Status* status) {
  status->status = Status::OK();
  delete s->session;
  delete s;
}
```

后端系统

tensorflow::Session 在运行时其动态类型，多态地调用相应子类实现的析构函数。

示例代码 5-30 tensorflow/core/common_runtime/session.h

```
class Session {
public:
  virtual ~Session() {};
};
```

5.9 性能调优

相比遗留的接口实现，新的接口实现存在若干优化技术提升系统的性能。虽然，截止本书撰写时，新的接口并未完全对外发布，但可以预期未来将删除既有已废弃的接口，替换为了新的接口实现。

共享图实例

如图**??**（第**??**页）所示，一个 Session 只能运行一个图实例。如果一个 Session 要运行其他的图实例，必须先关掉 Session，然后再将新的图实例注册到此 Session 中，最后启动新的计算图的执行过程。

但反过来，一个计算图可以运行在多个 Session 实例上。如果在 Graph 实例上维持 Session 的引用计数器，在 Session 创建时，在该图实例上增加 1；在 Session 销毁时 (不是关闭 Session)，在该图实例上减少 1；当计数器为 0 时，则自动删除图实例。在新的接口

实现中，实现了该引用计数器的技术。

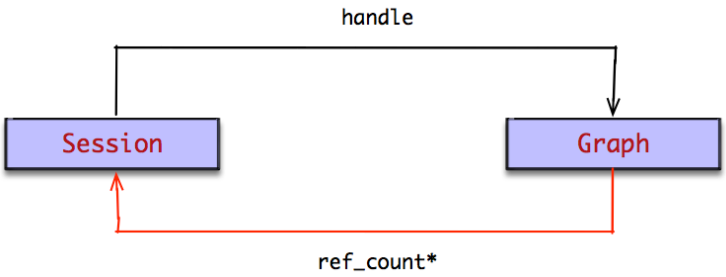


图 5-12 计算图：Session 引用计数器技术

消除序列化

如图??（第??页）所示，在遗留的接口实现中，前端 Python 在图构造期，将图构造完成后，并将其序列化，最后通过 `Session::Create` 或 `Session::Extend` 传递给后端 C++ 系统。这本质是一个图实例的拷贝过程，具有很大的时延开销。

图 5-13 图实例：序列化/反序列化

如图??（第??页）所示，在新的接口实现中，可以去除 `Session` 的 `Create/Extend` 的语义。在图的构造器，前端 Python 在构造每个 OP 时，直接通过 C API 将其追加至后端 C++ 的图实例中，从而避免了图实例在前后端的序列化和反序列化的开销。

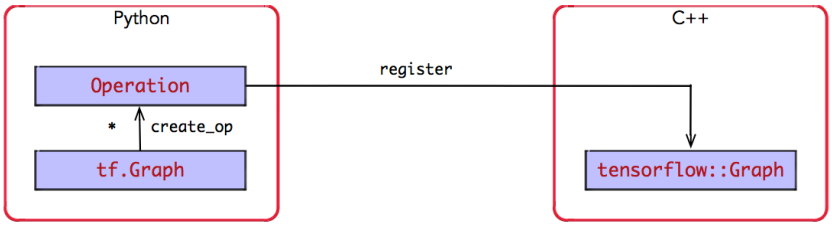


图 5-14 图实例：实施注册 asciiOP

第 III 部分

编程模型

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

- Martin Flower

6 计算图

在 TensorFlow 的计算图中，使用 OP 表示节点，根据 OP 之间计算和数据依赖关系，构造 OP 之间生产与消费的数据依赖关系，并通过有向边表示。其中，有向边存在两种类型，一种承载数据，并使用 `Tensor` 表示；另一种不承载数据，仅表示计算依赖关系。

本章将阐述 TensorFlow 中最重要的领域对象：**计算图**。为了全面阐述计算图的关键实现技术，将分别探讨前后端的系统设计和实现，并探究前后端系统间计算图转换的工作流原理。

6.1 Python 前端

在 Python 的前端系统中，并没有 `Node, Edge` 的概念，仅存在 `Operation, Tensor` 的概念。事实上，在前端 Python 系统中，`Operation` 表示图中的 `Node` 实例，而 `Tensor` 表示图中的 `Edge` 实例。

Operation

`OP` 用于表达某种抽象的数学计算，它表示计算图中的节点。`Operation` 是前端 Python 系统中最重要的一个领域对象，也是 TensorFlow 运行时最小的计算单位。

领域模型

如图**??**（第**??**页）所示，`Operation` 表示某种抽象计算，上游节点输出的零个或多个 `Tensor` 作为其输入，经过计算后输出零个或多个 `Tensor` 至下游的节点，从而上下游的 `Operation` 之间产生了数据依赖关系。特殊地，`Operation` 可能持有上游的控制依赖边的集合，表示潜在的计算依赖关系。

在计算图构造期间，通过 OP 构造器 (OP Constructor)，构造 `Operation` 实例，并将

其注册至默认的图实例中。与此同时，Operation 反过来通过 graph 直接持有该图实例。

Operation 的元数据由 OpDef 与 NodeDef 持有，它们以 ProtoBuf 的格式存在，它描述了 Operation 最本质的东西。其中，OpDef 描述了 OP 的静态属性信息，例如 OP 的名称，输入/输出参数列表，属性集定义等信息。而 NodeDef 描述了 OP 的动态属性值信息，例如属性值等信息。

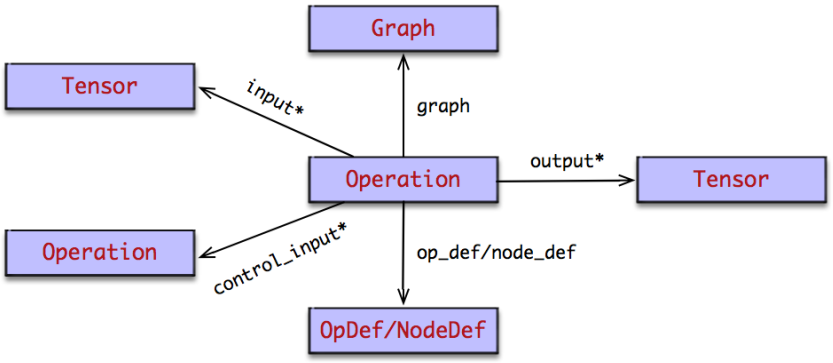


图 6-1 领域对象：Operation

构造器

```
class Operation(object):
  def __init__(self, node_def, g, inputs=None, output_types=None,
               control_inputs=None, input_types=None, original_op=None,
               op_def=None):
    # 1. NodeDef
    self._node_def = copy.deepcopy(node_def)

    # 2. OpDef
    self._op_def = op_def

    # 3. Graph
    self._graph = g

    # 4. Input types
    if input_types is None:
      input_types = [i.dtype.base_dtype for i in self._inputs]
    self._input_types = input_types

    # 5. Output types
    if output_types is None:
      output_types = []
    self._output_types = output_types

    # 6. Inputs
    if inputs is None:
      inputs = []
    self._inputs = list(inputs)

    # 7. Control Inputs.
    if control_inputs is None:
      control_inputs = []

    self._control_inputs = []
    for c in control_inputs:
      c_op = self._get_op_from(c)
```

```
        self._control_inputs.append(c_op)

      # 8. Outputs
      self._outputs = [Tensor(self, i, output_type)
                       for i, output_type in enumerate(output_types)]

      # 9. Build producter-consumer relation.
      for a in self._inputs:
        a._add_consumer(self)

      # 10. Allocate unique id for opeartion in graph.
      self._id_value = self._graph._next_id()
```

属性集

Operation 定义了常用了属性方法，用于获取该 OP 的元数据。其中，name 表示图中节点的名称，包括 name_scope 的层次名称，在图实例的范围内是唯一的，例如 layer_2/MatMul；type 则表示该 OP 类型唯一的名称，例如 MatMul, Variable。

```
class Operation(object):
  @property
  def name(self):
    """The full name of this operation."""
    return self._node_def.name

  @property
  def type(self):
    """The type of the op (e.g. "MatMul")."""
    return self._node_def.op

  @property
  def graph(self):
    """The Graph that contains this operation."""
    return self._graph

  @property
  def node_def(self):
    """Returns the NodeDef proto that represents this operation."""
    return self._node_def

  @property
  def op_def(self):
    """Returns the OpDef proto that represents the type of this op."""
    return self._op_def

  @property
  def device(self):
    """The name of the device to which this op has been assigned."""
    return self._node_def.device
```

运行 OP

可以从该 OP 为末端反向遍历图，寻找最小依赖的子图，并在默认的 Session 中执行该子图。

```
class Operation(object):
  def run(self, feed_dict=None, session=None):
    """Runs this operation in a Session.

    Calling this method will execute all preceding operations that
    produce the inputs needed for this operation.
    """
    _run_using_default_session(self, feed_dict, session)
```

其中，_run_using_default_session 将使用默认的 Session 运行该 OP。

```
def _run_using_default_session(operation, feed_dict, session=None):
  """Uses the default session to run "operation".
  """
  if session is None:
    session = get_default_session()
  session.run(operation, feed_dict)
```

Tensor

在图构造期，Tensor 在图中并未承载数据，它仅表示 Operation 输出的一个符号句柄。事实上，需要通过 Session.run 计算才能得到 Tensor 所持有的真实数据。

生产者与消费者

如图??（第??页）所示，Tensor 是两个 Operation 数据交换的桥梁，它们之间构造了典型的**生产者与消费者**之间的关系。上游 Operation 作为生产者，经过某种抽象计算，生产了一个 Tensor，并以此作为该上游 Operation 的输出之一，并使用 output_index 作为标识。该 Tensor 被传递给下游 Operation，并作为下游 Operation 的输入，下游 Operation 充当该 Tensor 的消费者。

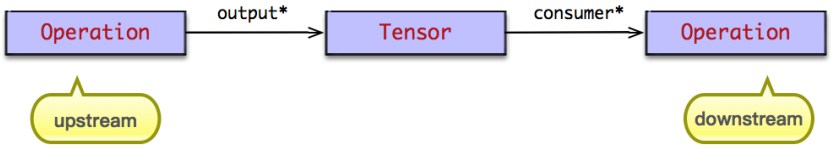


图 6-2 Tensor: 生产者-消费者

领域模型

如图??（第??页）所示，Tensor 通过 op 持有扮演生产者角色的 Operation，并且使用 index 表示该 Tensor 在该 Operation 输出列表中的索引。也就是说，可以使用 op:index 的二元组信息在图中唯一标识一个 Tensor 实例。

此外，Tensor 持有 Operation 的消费者列表，用于追踪该 Tensor 输出到哪些 Operation 实例了。因此，Tensor 充当了计算图的边，构建了 Operation 之间的数据依赖关系。

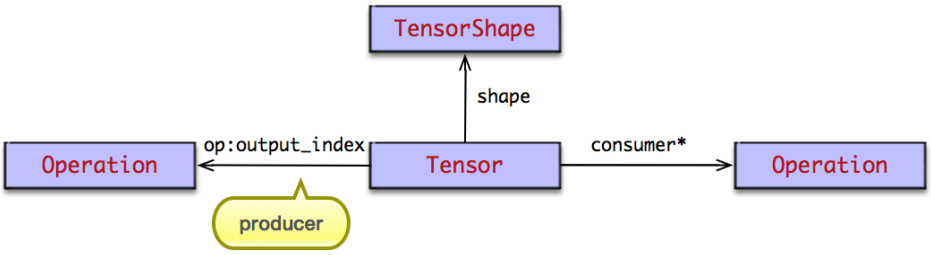


图 6-3 领域对象：Tensor

建立关联

最后，参考 Operation 与 Tensor 的部分实现，很容易找两者之间生产者-消费者的关联关系。当 Tensor 列表作为输入流入 Operation 时，此时建立了下游 Operation 与输入的 Tensor 列表之间的消费关系。

```
class Operation(object):
  def __init__(self, node_def, graph, inputs=None, output_types=None):
    # self(Operation) as consumer for input tensors.
    self._inputs = list(inputs)
    for a in self._inputs:
      a._add_consumer(self)

    # self(Operation) as producer for output tensors.
    self._output_types = output_types
    self._outputs = [Tensor(self, i, output_type)
                     for i, output_type in enumerate(output_types)]
```

同样地，Tensor 在构造器中持有上游的的生产者 Operation，及其该 Tensor 实例在该 Operation 的 outputs 列表中的索引。此外，当调用 _add_consumer，将该下游 Operation 追加至消费者列表之中。

```
class Tensor(_TensorLike):
  def __init__(self, op, value_index, dtype):
    # Index of the OP's endpoint that produces this tensor.
    self._op = op
    self._value_index = value_index

    # List of operations that use this Tensor as input.
    # We maintain this list to easily navigate a computation graph.
    self._consumers = []

  def _add_consumer(self, consumer):
    if not isinstance(consumer, Operation):
      raise TypeError("Consumer must be an Operation: %s" % consumer)
    self._consumers.append(consumer)
```

属性集

通过 Tensor 可以追溯上游的 Operation，从而获取相关的元数据。可以推测，遍历计算图的算法是反向的，与拓扑排序算法的遍历方向刚好相反。其中，name 返回了 (node:output_index) 的二元组信息，在计算图的范围内唯一地标识了该 Tensor 实例。

```
class Tensor(_TensorLike):
  @property
  def op(self):
    """The Operation that produces this tensor as an output."""
    return self._op

  @property
  def dtype(self):
    """The DType of elements in this tensor."""
    return self._dtype

  @property
  def graph(self):
    """The Graph that contains this tensor."""
    return self._op.graph

  @property
  def name(self):
    """The string name of this tensor."""
    return "%s:%d" % (self._op.name, self._value_index)

  @property
  def device(self):
    """The name of the device on which this tensor will be produced."""
    return self._op.device

  @property
  def shape(self):
    """Returns the TensorShape that represents the shape of this tensor.
    """
    return self._shape

  @property
  def value_index(self):
    """The index of this tensor in the outputs of its Operation."""
    return self._value_index
```

评估

```
class Tensor(_TensorLike):
  def eval(self, feed_dict=None, session=None):
    """Evaluates this tensor in a Session.

    Calling this method will execute all preceding operations that
    produce the inputs needed for the operation that produces this
    tensor.
    """
    return _eval_using_default_session(self, feed_dict, self.graph, session)
```

其中，_eval_using_default_session 将使用默认的 Session 评估该 Tensor 实例。注意，tf.Session.run 的 fetches 列表可以混合接收 Operation，Tensor 实例。

```
def _eval_using_default_session(tensors, feed_dict, graph, session=None):
  """Uses the default session to evaluate one or more tensors."""
  if session is None:
    session = get_default_session()
  return session.run(tensors, feed_dict)
```

TensorShape

Tensor 使用 TensorShape 描述其形状信息。它持有该 Tensor 的数据类型，及其 Dimension 列表，每个 Dimension 描述了该维度的大小。其中，TensorShape 与 Dimension 都是值对象，包含了一些实用的数学计算方法，例如计数，合并，兼容性检查等。

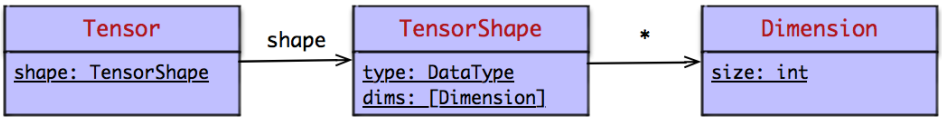


图 6-4 TensorShape

显而易见，可以使用 TensorShape 推算出 Tensor 所包含的元素个数。

```
class TensorShape(object):
  def num_elements(self):
    if self.is_fully_defined():
      size = 1
      for dim in self._dims:
        size *= dim.value
      return size
    else:
      return None
```

工厂方法

存在几个实用的工厂方法，scalar，vector，matrix 用于分别构造 0 维，1 维，2 维的 TensorShape 实例。

```
def scalar():
  return TensorShape([])

def vector(length):
  return TensorShape([length])

def matrix(rows, cols):
  return TensorShape([rows, cols])
```

部分定义

当构造计算图时，对其 TensorShape 暂时不能确定，则可以使用 None 表示。存在两种情况，如果 rank 大小未知，称该 TensorShape 未知；如果 rank 大小已知，则称该 TensorShape **部分定义**。

```
def unknown_shape(ndims=None):
  if ndims is None:
    return TensorShape(None)
  else:
    return TensorShape([Dimension(None)] * ndims)
```

完全定义

相反地，当 TensorShape 每个维度的大小都已确定，则称**完全定义**。

```
class TensorShape(object):
  def is_fully_defined(self):
    return (self._dims is not None and all(dim.value is not None
                                           for dim in self._dims))
```

属性集

可以使用 ndims 属性返回 TensorShape 的 rank 大小，使用 dims 属性返回 Dimension 列表。

```
class TensorShape(object):
  @property
  def dims(self):
    return self._dims

  @property
  def ndims(self):
    if self._dims is None:
      return None
    else:
      return len(self._dims)
```

转换

可以使用 as_proto 将其转换为 TensorShapeProto 表示。特殊地，当某一个 Dimension 未知，为了能够实施序列化，需要将 None 转换为-1。

```
class TensorShape(object):
  def _dims_as_proto(self):
    def _size(dim):
      return -1 if dim.value is None else dim.value

    return [tensor_shape_pb2.TensorShapeProto.Dim(size=_size(d))
            for d in self._dims]

  def as_proto(self):
    if self._dims is None:
      return tensor_shape_pb2.TensorShapeProto(unknown_rank=True)
    else:
      return tensor_shape_pb2.TensorShapeProto(dim=self._dims_as_proto())
```

也可以使用 as_list 将其转为 Dimension 列表。如果 TensorShape 的 rank 大小未知，则抛出 ValueError 异常。

```
class TensorShape(object):
  def as_list(self):
    if self._dims is None:
      raise ValueError("as_list() is not defined on an unknown TensorShape.")
    return [dim.value for dim in self._dims]
```

相反地，使用 as_shape 将 Deimension 列表，或 TensorShapeProto 转换为了 TensorShape 实例。

```
def as_shape(shape):
  if isinstance(shape, TensorShape):
    return shape
  else:
    return TensorShape(shape)
```

特殊地，当构造 TensorShape 时，当 TensorShapeProto 的某一维度大小为-1 时，将其转换为 None 的表示。

```
class TensorShape(object):
  def __init__(self, dims):
    if dims is None:
      self._dims = None
    elif isinstance(dims, tensor_shape_pb2.TensorShapeProto):
      if dims.unknown_rank:
        self._dims = None
      else:
        self._dims = [
          as_dimension(dim.size if dim.size != -1 else None)
          for dim in dims.dim
        ]
    elif isinstance(dims, TensorShape):
      self._dims = dims.dims
    else:
      try:
        dims_iter = iter(dims)
      except TypeError:
        # Treat as a singleton dimension
        self._dims = [as_dimension(dims)]
```

```
        else:
          # Got a list of dimensions
          self._dims = [as_dimension(d) for d in dims_iter]
```

Graph

Graph 是 TensorFlow 最重要的领域对象，TensorFlow 的运行时就是完成 Graph 的构造、传递、剪枝、优化、分裂、执行。因此，熟悉 Graph 的领域模型，对于理解整个 TensorFlow 运行时大有裨益。

领域模型

如图??（第??页）所示，一个 Graph 对象将包含一系列 Operation 对象，表示计算单元的集合。同时，它间接持有一系列 Tensor 对象，表示数据单元的集合。

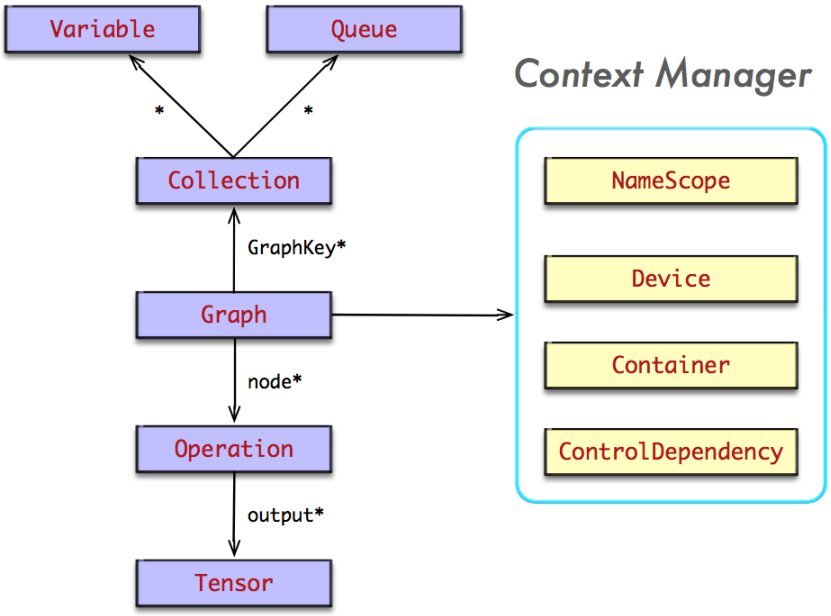


图 6-5 领域对象：Graph

为了快速索引图中的节点信息，在当前图的作用域内为每个 Operation 分配唯一的 id，并在图中存储 _nodes_by_id 的数据字典。同时，为了可以根据节点的名字快速索引节点信息，在图中也存储了 _nodes_by_name 的数据字典。

```
class Graph(object):
  def __init__(self):
    self._lock = threading.Lock()
    self._nodes_by_id = dict()    # GUARDED_BY(self._lock)
    self._next_id_counter = 0     # GUARDED_BY(self._lock)
    self._nodes_by_name = dict()  # GUARDED_BY(self._lock)
```

```
      self._version = 0             # GUARDED_BY(self._lock)
```

在图构造期，OP 通过 OP 构造器创建，最终被添加至当前的 Graph 实例中。当图被冻结后，便不能往图中追加节点了，使得 Graph 实例在多线程中被安全地共享。

```
class Graph(object):
  def _add_op(self, op):
    self._check_not_finalized()
    with self._lock:
      self._nodes_by_id[op._id] = op
      self._nodes_by_name[op.name] = op
      self._version = max(self._version, op._id)
```

分组

为了更好地管理 Graph 中的节点，在每个 Operation 上打上特定的标签，实现了节点的分类。相同类型的节点被划归在同一个 Collection 中，并使用唯一的 GraphKey 标识该集合。随后，便可以根据 GraphKey 快速索引相关的节点信息。其中，系统预定义了常用的 GraphKey，同时也支持自定义的 GraphKey。

```
class GraphKeys(object):
  # Key to collect Variable objects that are global (shared across machines).
  # Default collection for all variables, except local ones.
  GLOBAL_VARIABLES = "variables"

  # Key to collect local variables that are local to the machine and are not
  # saved/restored.
  LOCAL_VARIABLES = "local_variables"

  # Key to collect model variables defined by layers.
  MODEL_VARIABLES = "model_variables"

  # Key to collect Variable objects that will be trained by the
  # optimizers.
  TRAINABLE_VARIABLES = "trainable_variables"

  # Key to collect summaries.
  SUMMARIES = "summaries"

  # Key to collect QueueRunners.
  QUEUE_RUNNERS = "queue_runners"

  # Key to collect table initializers.
  TABLE_INITIALIZERS = "table_initializer"

  # Key to collect asset filepaths. An asset represents an external resource
  # like a vocabulary file.
  ASSET_FILEPATHS = "asset_filepaths"

  # Key to collect Variable objects that keep moving averages.
  MOVING_AVERAGE_VARIABLES = "moving_average_variables"
  # Key to collect regularization losses at graph construction.

  REGULARIZATION_LOSSES = "regularization_losses"

  # Key to collect concatenated sharded variables.
  CONCATENATED_VARIABLES = "concatenated_variables"
```

```
  # Key to collect savers.
  SAVERS = "savers"

  # Key to collect weights
  WEIGHTS = "weights"

  # Key to collect biases
  BIASES = "biases"

  # Key to collect activations
  ACTIVATIONS = "activations"

  # Key to collect update_ops
  UPDATE_OPS = "update_ops"

  # Key to collect losses
  LOSSES = "losses"

  # Key to collect BaseSaverBuilder.SaveableObject instances for
    checkpointing.
  SAVEABLE_OBJECTS = "saveable_objects"

  # Key to collect all shared resources used by the graph which need to be
  # initialized once per cluster.
  RESOURCES = "resources"

  # Key to collect all shared resources used in this graph which need to be
  # initialized once per session.
  LOCAL_RESOURCES = "local_resources"

  # Trainable resource-style variables.
  TRAINABLE_RESOURCE_VARIABLES = "trainable_resource_variables"

  # Key to indicate various ops.
  INIT_OP = "init_op"
  LOCAL_INIT_OP = "local_init_op"
  READY_OP = "ready_op"
  READY_FOR_LOCAL_INIT_OP = "ready_for_local_init_op"
  SUMMARY_OP = "summary_op"
  GLOBAL_STEP = "global_step"

  # Used to count the number of evaluations performed during a
  # single evaluation run.
  EVAL_STEP = "eval_step"
  TRAIN_OP = "train_op"

  # Key for control flow context.
  COND_CONTEXT = "cond_context"
  WHILE_CONTEXT = "while_context"
```

当某个 Opeartion 创建时，可以将其划归在特定的集合中，便于后期根据 GraphKey 快速索引。

```
class Graph(object):
  def add_to_collection(self, name, value):
    self._check_not_finalized()
    with self._lock:
      if name not in self._collections:
        self._collections[name] = [value]
      else:
        self._collections[name].append(value)
```

图实例

一般地，OP 注册到一个全局的、唯一的、隐式的、默认的图实例中。特殊地，TensorFlow 也可以显式地创建新的图实例 `g`，并调用 `g.as_default()` 使其成为当前线程中唯一默认的图实例，并在该上下文管理器中所创建的 OP 都将自动注册到该图实例中。

```
with tf.Graph().as_default() as g:
  c = tf.constant(5.0)
  assert c.graph is g
```

事实上，`g.as_default` 从当前线程的图栈中返回了一个上下文管理器，使得当前的图实例 `g` 覆盖原来默认的图实例；当退出了上下文管理器后，则恢复原来默认的图实例。但是，在任何一个时刻，在当前线程中有且仅有一个图实例成为**默认的**，可以调用 `tf.get_default_graph()`，返回该默认的图实例。

```
_default_graph_stack = _DefaultGraphStack()

def get_default_graph():
  """Returns the default graph for the current thread."""
  return _default_graph_stack.get_default()

class Graph(object):
  def as_default(self):
    """Returns a context manager that makes this Graph the default graph."""
    return _default_graph_stack.get_controller(self)
```

其中，`_DefaultStack` 中的 `get_controller` 在栈顶追加新的图实例；当退出上下文管理器后，则从栈顶移除该图实例，恢复之前的图实例。当调用 `get_default` 时，如果栈为空则通过 `_GetGlobalDefaultGraph` 返回全局的、唯一的、隐式的图实例。在大部分 TensorFlow 程序中，如果没有显式创建多个图实例，则所有 OP 都默认地注册到该图实例中。

```
class _DefaultStack(threading.local):
  """A thread-local stack for providing implicit defaults."""

  def __init__(self):
    super(_DefaultStack, self).__init__()
    self.stack = []

  def get_default(self):
    return self.stack[-1] if len(self.stack) >= 1 else None

  @tf_contextlib.contextmanager
  def get_controller(self, default):
    """A context manager for manipulating a default stack."""
    try:
      self.stack.append(default)
      yield default
    finally:
      self.stack.remove(default)

class _DefaultGraphStack(_DefaultStack):
  """A thread-local stack for providing an implicit default graph."""
```

```
  def __init__(self):
    super(_DefaultGraphStack, self).__init__()
    self._global_default_graph = None

  def get_default(self):
    """Override that returns a global default if the stack is empty."""
    ret = super(_DefaultGraphStack, self).get_default()
    if ret is None:
      ret = self._GetGlobalDefaultGraph()
    return ret

  def _GetGlobalDefaultGraph(self):
    if self._global_default_graph is None:
      self._global_default_graph = Graph()
    return self._global_default_graph
```

名字空间

为了更好地管理图中的节点，使用 name_scope 对图中的节点实施层次化命名。例如，想要在全宇宙的范围内定位故宫的位置，可以实施层次化的命名：宇宙/银河系/太阳系/地球/中国/北京/故宫。这对于 TensorBoard 实现计算图的可视化非常有帮助，当计算图比较大时，可以通过折叠的方式展示图，也可以通过展开的方式一窥究竟。

内嵌的 name_scope 将继承外围的 name_scope；如果内嵌的 name_scope 以/结尾，将重置为指定的 name_scope；如果内嵌的 name_scope 为空字符串或 None，将重置整个 name_scope。

```
with tf.Graph().as_default() as g:
  with g.name_scope("nested") as scope:
    nested_c = tf.constant(10.0, name="c")
    assert nested_c.op.name == "nested/c"

    # Create a nested scope called "inner".
    with g.name_scope("inner"):
      nested_inner_c = tf.constant(30.0, name="c")
      assert nested_inner_c.op.name == "nested/inner/c"

      # Treats `scope` as an absolute name scope,
      # and switches to the "nested/" scope.
      with g.name_scope(scope):
        nested_d = tf.constant(40.0, name="d")
        assert nested_d.op.name == "nested/d"

        # reset name scope
        with g.name_scope(""):
          e = tf.constant(50.0, name="e")
          assert e.op.name == "e"
```

事实上，name_scope 是一个上下文管理器，在嵌套的 name_scope 中，实现了 name_scope 栈式管理。当 name_scope 出了作用域，便自动恢复外围的 name_scope。

```
def _name_from_scope_name(name):
  return name[:-1] if name[-1] == "/" else name

class Graph(object):
  def __init__(self):
    self._name_stack = ""

  @tf_contextlib.contextmanager
  def name_scope(self, name):
    try:
      old_stack = self._name_stack
      if not name:
        new_stack = None
      elif name and name[-1] == "/":
        new_stack = _name_from_scope_name(name)
      else:
        new_stack = self.unique_name(name)
      self._name_stack = new_stack
      yield "" if new_stack is None else new_stack + "/"
    finally:
      self._name_stack = old_stack
```

在图构造期，OP 构造器更习惯于使用 `tf.name_scope`，它从输入的 `Operation` 或 `Tensor` 列表中尝试获取图实例；如果未能获取到，则返回默认的图实例。然后，再在该图实例上追加新的 `name_scope`。

```
@tf_contextlib.contextmanager
def name_scope(name, default_name=None, values=[]):
  n = default_name if name is None else name
  g = _get_graph_from_inputs(values)
  with g.as_default(), g.name_scope(n) as scope:
    yield scope
```

控制依赖

可以通过内嵌的 `control_dependencies` 合并外围的 `control_dependencies`，或通过 `None` 重置控制依赖集合为空。

```
with g.control_dependencies([a, b]):
  # Ops constructed here run after `a` and `b`.
  with g.control_dependencies(None):
    # Ops constructed here not waiting for either `a` or `b`.
    with g.control_dependencies([c, d]):
      # Ops constructed here run after `c` and `d`,
      # also not waiting for either `a` or `b`.
  with g.control_dependencies([e, f]):
    # Ops constructed here run after `a, b, e, f`.
```

事实上，`control_dependencies` 返回了一个上下文管理器，用于指定 OP 的控制依赖关系。其中，`control_ops` 记录了当前层所依赖的 Operation 列表，而 `current` 记录了当前层，及其外围所依赖的所有 Operation 列表。

```
class Graph(object):
  def control_dependencies(self, control_inputs):
    if control_inputs is None:
      return self._ControlDependenciesController(self, None)

    control_ops = []
    current = self._current_control_dependencies()
    for c in control_inputs:
      c = self.as_graph_element(c)
      if isinstance(c, Tensor):
        c = c.op
      if c not in current:
        control_ops.append(c)
        current.add(c)
    return self._ControlDependenciesController(self, control_ops)
```

_ControlDependenciesController 实现了一个控制依赖的控制器。control_inputs 为 None，将启用一个新的作用域，从而实现了新栈替代旧栈；当退出当前上下文的作用域后，恢复之前的旧栈，从而清除了所有之前的控制依赖关系。否则，每进入一层 control_inputs，便叠加当前作用域到当前的栈中。

```
class Graph(object):
  def __init__(self):
    self._control_dependencies_stack = []

  def _push_control_dependencies_controller(self, controller):
    self._control_dependencies_stack.append(controller)

  def _pop_control_dependencies_controller(self):
    self._control_dependencies_stack.pop()

  class _ControlDependenciesController(object):
    """Context manager for control_dependencies()."""

    def __init__(self, graph, control_inputs):
      self._graph = graph
      if control_inputs is None:
        self._control_inputs = []
        self._new_stack = True
      else:
        self._control_inputs = control_inputs
        self._new_stack = False
      self._seen_nodes = set()
      self._old_stack = None

    def __enter__(self):
      if self._new_stack:
        # Clear the control_dependencies.
        self._old_stack = self._graph._control_dependencies_stack
        self._graph._control_dependencies_stack = []
      self._graph._push_control_dependencies_controller(self)

    def __exit__(self, unused_type, unused_value, unused_traceback):
      self._graph._pop_control_dependencies_controller()
      if self._new_stack:
        self._graph._control_dependencies_stack = self._old_stack
```

_current_control_dependencies 用于规约所有外围的 control_inputs，直至当前层所依赖的 Operation 列表。

```
class Graph(object):
  def _current_control_dependencies(self):
    ret = set()
    for controller in self._control_dependencies_stack:
      for op in controller.control_inputs:
        ret.add(op)
    return ret
```

容器

```
with g.container('experiment0'):
  # All stateful Operations constructed in this context will be placed
  # in resource container "experiment0".
  v1 = tf.Variable([1.0])
  v2 = tf.Variable([2.0])
  with g.container("experiment1"):
    # All stateful Operations constructed in this context will be
    # placed in resource container "experiment1".
    v3 = tf.Variable([3.0])
    q1 = tf.FIFOQueue(10, tf.float32)
  # All stateful Operations constructed in this context will be
  # be created in the "experiment0".
  v4 = tf.Variable([4.0])
  q1 = tf.FIFOQueue(20, tf.float32)
  with g.container(""):
    # All stateful Operations constructed in this context will be
    # be placed in the default resource container.
    v5 = tf.Variable([5.0])
    q3 = tf.FIFOQueue(30, tf.float32)

# Resets container "experiment0", after which the state of v1, v2, v4, q1
# will become undefined (such as uninitialized).
tf.Session.reset(target, ["experiment0"])
```

```
class Graph(object):
  @tf_contextlib.contextmanager
  def container(self, container_name):
    """Returns a context manager that specifies the resource container."""
    original_container = self._container
    try:
      self._container = container_name
      yield self._container
    finally:
      self._container = original_container
```

图构造

在计算图的构造期间，不执行任何 OP 的计算。简单地说，图的构造过程就是根据 OP 构造器完成 `Operation` 实例的构造。而在 `Operation` 实例的构造之前，需要实现完成 `OpDef` 与 `NodeDef` 的构造过程。

OpDef 仓库

OpDef 仓库在系统首次访问时，实现了 OpDef 的延迟加载和注册。也就是说，对于某中类型的 OpDef 仓库，_InitOpDefLibrary 模块首次导入时，扫描 op_list_ascii 表示的所有 OP，并将其转换为 Protobuf 格式的 OpList 实例，最终将其注册到 OpDefLibrary 实例之中。

例如，模块 gen_array_ops 是构建版本时自动生成的，它主要完成所有 array_ops 类型的 OpDef 的定义，并自动注册到 OpDefLibrary 的仓库实例中，并提供按名查找 OpDef 的服务接口。

```
_op_def_lib = _InitOpDefLibrary()

def _InitOpDefLibrary():
  op_list = _op_def_pb2.OpList()
  _text_format.Merge(_InitOpDefLibrary.op_list_ascii, op_list)
  op_def_lib = _op_def_library.OpDefLibrary()
  op_def_lib.add_op_list(op_list)
  return op_def_lib

_InitOpDefLibrary.op_list_ascii = """op {
  name: "ZerosLike"
  input_arg {
    name: "x"
    type_attr: "T"
  }
  output_arg {
    name: "y"
    type_attr: "T"
  }
  attr {
    name: "T"
    type: "type"
  }
}
# ignore others
"""
```

工厂方法

如图**??**（第**??**页）所示。当 Client 使用 OP 构造器创建一个 Operation 实例时，将最终调用 Graph.create_op 方法，将该 Operation 实例注册到该图实例中。

也就是说，一方面，Graph 充当 Operation 的工厂，负责 Operation 的创建职责；另一方面，Graph 充当 Operation 的仓库，负责 Operation 的存储，检索，转换等操作。

这个过程常称为计算图的构造。在计算图的构造期间，并不会触发运行时的 OP 运算，它仅仅描述计算节点之间的依赖关系，并构建 DAG 图，对整个计算过程做整体规划。

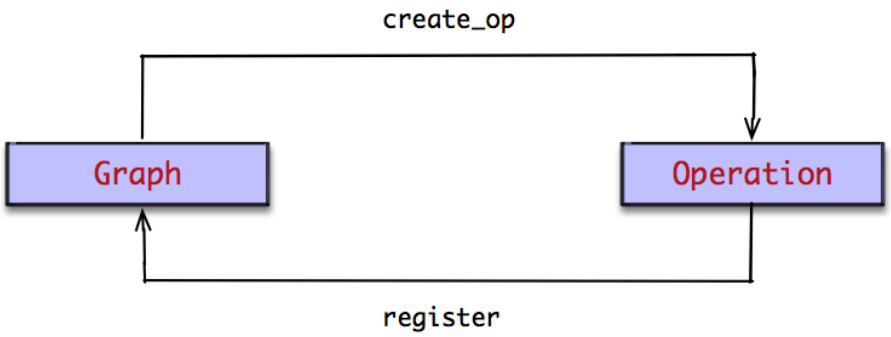


图 6-6 Graph: OP 工厂 + OP 仓库

OP 构造器

如图??(第??页)所示。在图构造期, Client 使用 `tf.zeros_like` 构造一个名为 `ZerosLike` 的 OP，该 OP 拥有一个输入，输出一个全 0 的 Tensor；其中，`tf.zeros_like` 常称为 OP 构造器。

然后，OP 构造器调用一段自动生成的代码，进而转调 `OpDefLibrary.apply_op` 方法。

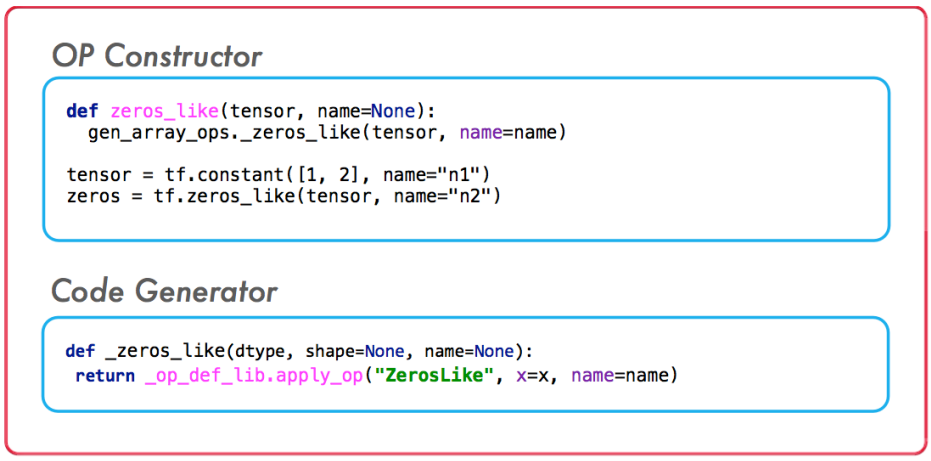


图 6-7 OP 构造器与代码生成器

构造 OpDef 与 NodeDef

然后，如图??（第??页）所示。`OpDefLibrary` 根据 OP 的名字从 `OpDefLibrary` 中，找到对应 `OpDef` 实例；最终，通过 `Graph.create_op` 的工厂方法，创建 `NodeDef` 实例，进而创建 `Operation` 实例，将其自身注册到图实例中。

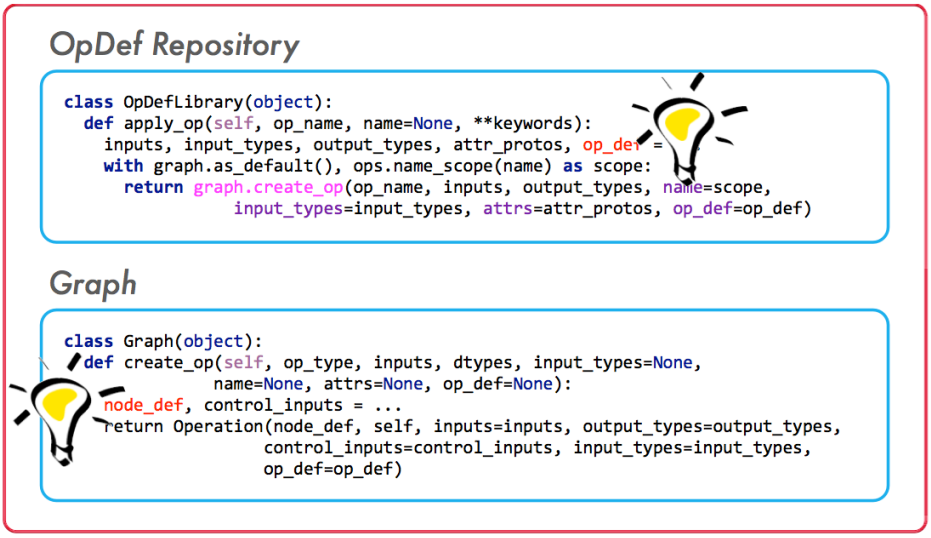


图 6-8 创建 Operation 实例：创建 OpDef, NodeDef 实例

6.2 后端 C++

在 C++ 后端，计算图是 TensorFlow 领域模型的核心。

边

Edge 持有前驱节点与后驱节点，从而实现了计算图的连接。一个节点可以拥有零条或多条输入边，与可以有零条或多条输出边。一般地，计算图中存在两类边：

1. 普通边：用于承载数据 (以 Tensor 表示)，表示节点间“生产者-消费者”的数据依赖关系，常用实线表示；
2. 控制依赖：不承载数据，用于表示节点间的执行依赖关系，常用虚线表示。

两个标识

Edge 持有两个重要的索引：

1. src_output：表示该边为「前驱节点」的第 src_output 条输出边；
2. dst_input：表示该边为「后驱节点」的第 dst_input 条输入边。

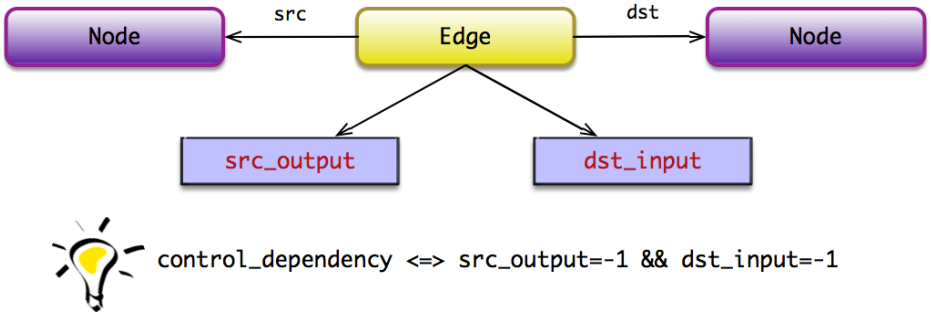


图 6-9 领域对象：Edge

例如，存在两个前驱节点 s1，s2，都存在两条输出边；存在两个后驱节点 d1，d2，都存在两条输入边。

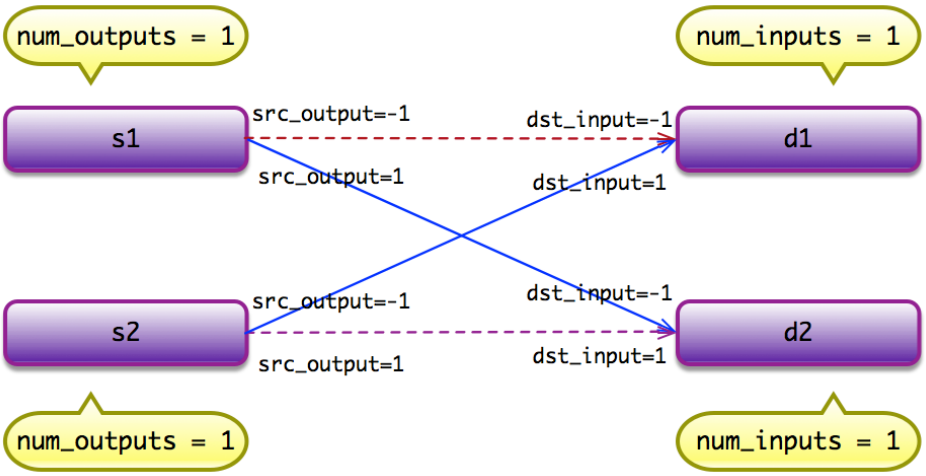


图 6-10 边例子

控制依赖

对于控制依赖边，其 src_output，dst_input 都为-1(Graph::kControlSlot)，暗喻控制依赖边不承载任何数据。

```
bool Edge::IsControlEdge() const {
   // or dst_input_ == Graph::kControlSlot;
   return src_output_ == Graph::kControlSlot;
}
```

Tensor 标识

一般地，计算图的「普通边」承载 Tensor，并使用 TensorId 标识。Tensor 标识由源节点的名字，及其所在边的 src_output 唯一确定。

```
TensorId ::= node_name:src_output
```

缺省地，src_output 默认为 0；也就是说，node_name 与 node_name:0 两者等价。特殊地，当 src_output 等于-1 时，表示该边为「控制依赖边」，TensorId 可以标识为 ^node_name，标识该边依赖于 node_name 所在的节点。

节点

Node(节点) 可以拥有零条或多条输入/输出的边，并使用 in_edges，out_edges 分别表示输入边和输出边的集合。另外，Node 持有 NodeDef，OpDef。其中，NodeDef 包含设备分配信息，及其 OP 的属性值列表；OpDef 持有 OP 的元数据，包括 OP 输入输出类型等信息。

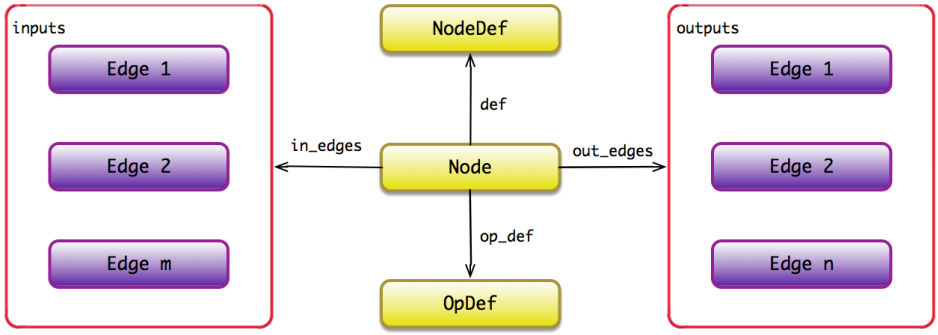


图 6-11 领域对象：Node

输入边

在输入边的集合中，可以按照索引 (dst_input) 线性查找。当节点输入的边比较多时，可能会成为性能的瓶颈。依次类推，按照索引 (src_output) 查找输出边，算法类同。

```
Status Node::input_edge(int idx, const Edge** e) const {
  for (auto edge : in_edges()) {
    if (edge->dst_input() == idx) {
      *e = edge;
      return Status::OK();
    }
  }
  return errors::NotFound("not found input edge ", idx);
}
```

前驱节点

首先通过 idx 索引找到输入边，然后通过输入边找到前驱节点。依次类推，按照索引查找后驱节点，算法类同。

```
Status Node::input_node(int idx, const Node** n) const {
  const Edge* e = nullptr;
  TF_RETURN_IF_ERROR(input_edge(idx, &e));
  *n = e == nullptr ? nullptr : e->src();
  return Status::OK();
}
```

图

Graph(计算图) 就是节点与边的集合。计算图是一个 DAG 图，计算图的执行过程将按照 DAG 的拓扑排序，依次启动 OP 的运算。其中，如果存在多个入度为 0 的节点，TensorFlow 运行时可以实现并发，同时执行多个 OP 的运算，提高执行效率。

图 6-12 领域模型：图

空图

计算图的初始状态，并非是一个空图。实现添加了两个特殊的节点：Source 与 Sink 节点，分别表示 DAG 图的起始节点与终止节点。其中，Source 的 id 为 0，Sink 的 id 为 1；依次论断，普通 OP 节点的 id 将大于 1。

Source 与 Sink 之间，通过连接「控制依赖」的边，保证计算图的执行始于 Source 节点，终于 Sink 节点。它们之前的控制依赖边，其 src_output，dst_input 值都为-1。

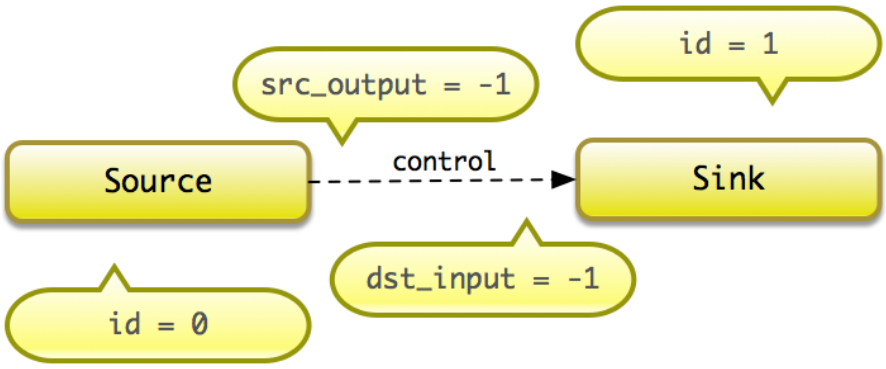


图 6-13 空图

Source 与 Sink 是两个内部实现保留的节点，其节点名称以下划线开头，分别使用 _SOURCE 和 _SINK 命名；并且，它们都是 NoOp，表示不执行任何计算。

```
Node* Graph::AddInternalNode(const char* name, int id) {
  NodeDef def;
  def.set_name(name);
  def.set_op("NoOp");

  Status status;
  Node* node = AddNode(def, &status);
  TF_CHECK_OK(status);
  CHECK_EQ(node->id(), id);
  return node;
}

Graph::Graph(const OpRegistryInterface* ops)
    : ops_(ops), arena_(8 << 10 /* 8kB */) {
  auto src  = AddInternalNode("_SOURCE", kSourceId);
  auto sink = AddInternalNode("_SINK",   kSinkId);
  AddControlEdge(src, sink);
}
```

习惯上，仅包含 Source 与 Sink 节点的计算图也常常称为空图。

非空图

在前端，用户使用 OP 构造器，将构造任意复杂度的计算图。对于运行时，实现将用户构造的计算图通过控制依赖的边与 Source/Sink 节点连接，保证计算图执行始于 Source 节点，终于 Sink 节点。

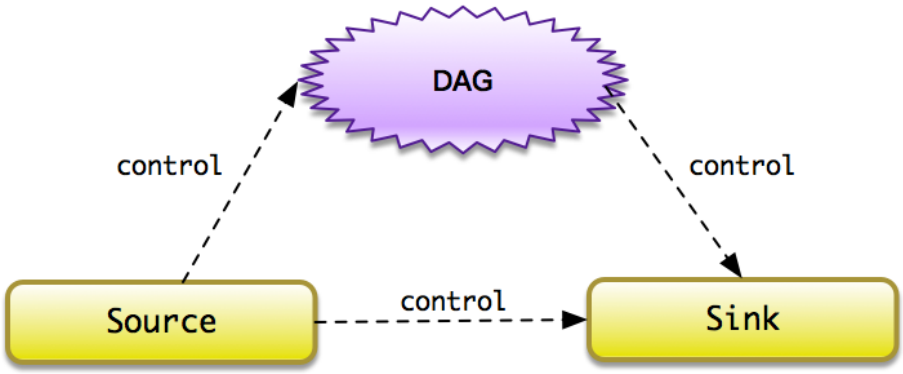


图 6-14 非空图

添加边

计算图的构造过程非常简单，首先通过 Graph::AddNode 在图中放置节点，然后再通过 Graph::AddEdge 在图中放置边，实现节点之间的连接。

```
const Edge* Graph::AllocEdge() const {
  Edge* e = nullptr;
  if (free_edges_.empty()) {
    e = new (arena_.Alloc(sizeof(Edge))) Edge;
  } else {
    e = free_edges_.back();
    free_edges_.pop_back();
  }
  e->id_ = edges_.size();
  return e;
}

const Edge* Graph::AddEdge(Node* source, int x, Node* dest, int y) {
  auto e = AllocEdge();
  e->src_ = source;
  e->dst_ = dest;
  e->src_output_ = x;
  e->dst_input_ = y;

  CHECK(source->out_edges_.insert(e).second);
  CHECK(dest->in_edges_.insert(e).second);

  edges_.push_back(e);
  edge_set_.insert(e);
  return e;
}
```

添加控制依赖边

添加控制依赖边，则可以转发调用 Graph::AddEdge 实现；此时，src_output，dst_input 都为-1。

```
const Edge* Graph::AddControlEdge(Node* src, Node* dst) {
  return AddEdge(src, kControlSlot, dst, kControlSlot);
}
```

OpDef 仓库

同样地，OpDef 仓库在 C++ 系统 main 函数启动之前完成 OpDef 的加载和注册。它使用 REGISTER_OP 宏完成 OpDef 的注册。

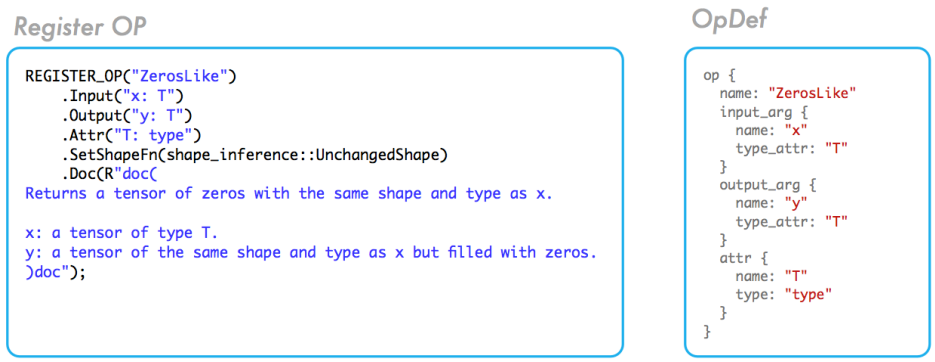


图 6-15 OpDef 注册：使用 REGISTER_OP

6.3 图传递

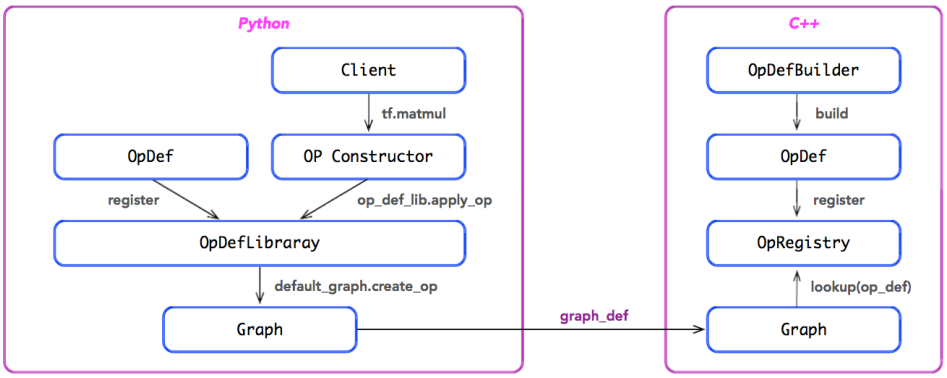


图 6-16 图的序列化与反序列化

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

- Martin Flower

7 设备

7.1 设备规范

设备规范 (Device Specification) 用于描述 OP 存储或计算设备的具体位置。

形式化

一个设备规范可以形式化地描述为:

```
DEVICE_SPEC ::= COLOCATED_NODE | PARTIAL_SPEC
COLOCATED_NODE ::= "@" NODE_NAME
PARTIAL_SPEC ::= ("/" CONSTRAINT) *
CONSTRAINT ::= ("job:" JOB_NAME)
             | ("replica:" [1-9][0-9]*)
             | ("task:" [1-9][0-9]*)
             | ( ("gpu" | "cpu") ":" ([1-9][0-9]* | "*") )
```

完整指定

如下例所示，完整地描述某个 OP 被放置在 PS 作业，0 号备份，0 号任务，GPU 0 号设备。

```
/job:ps/replica:0/task:0/device:GPU:0
```

部分指定

设备规范也可以部分指定，甚至为空。例如，下例仅描述了 `GPU0` 号设备。

```
/device:GPU:0
```

特殊地，当设备规范为空时，则表示对 OP 未实施设备约束，运行时自动选择设备放置 OP。

同位

使用 COLOCATED_NODE 指示该 OP 与指定的节点被同时放置在相同的设备上。例如，该节点与 other/node 放置在相同的设备上。

```
@other/node  # colocate with "other/node"
```

DeviceSpec

一个设备规范可以使用字符串，或者 DeviceSpec 表示。其中，DeviceSpec 是一个值对象，使用如下 5 个标识确定设备规范。

1. 作业名称
2. 备份索引
3. 任务索引
4. 设备类型
5. 设备索引

例如，使用 DeviceSpec 构造的设备规范。

```
# '/job:ps/replica:0/task:0/device:CPU:0'
DeviceSpec(job="ps", replica=0, task=0, device_type="CPU", device_index=0)
```

上下文管理器

常常使用上下文管理器 device(device_spec) 指定 OP 设备规范，在该上下文的作用域内构造的 OP 在运行时将被放置在指定设备上运行。

```
with g.device('/gpu:0'):
  # All OPs constructed here will be placed on GPU 0.
```

其中，device 是 Graph 的一个方法，它设计了一个栈式结构的上下文管理器，实现设备规范的闭包、合并、覆盖等特性。

合并

可以对两个不同范围的设备规范进行合并。

```
with device("/job:ps"):
  # All OPs constructed here will be placed on PS.
  with device("/task:0/device:GPU:0"):
    # All OPs constructed here will be placed on
    # /job:ps/task:0/device:GPU:0
```

覆盖

在合并两个相同范围的设备规范时，内部指定的设备规范具有高优先级，实现设备规范的覆盖。

```
with device("/device:CPU:0"):
  # All OPs constructed here will be placed on CPU 0.
  with device("/job:ps/device:GPU:0"):
    # All OPs constructed here will be placed on
    # /job:ps/device:GPU:0
```

重置

特殊地，当内部的设备规范置位为 None 时，将忽略外部所有设备规范的定义。

```
with device("/device:GPU:0"):
  # All OPs constructed here will be placed on CPU 0.
  with device(None):
    # /device:GPU:0 will be ignored.
```

设备规范函数

当指定设备规范时，常常使用字符串，或 DeviceSpec 进行描述。也可以使用更加灵活的**设备规范函数**，它提供了一种更加灵活的扩展方式指定设备规范。设备规范函数是一个回调函数，入参为 Operation，生成一个字符串格式的设备规范。

```
def matmul_on_gpu(n):
 if n.type == "MatMul":
   return "/gpu:0"
 else:
   return "/cpu:0"

with g.device(matmul_on_gpu):
  # All OPs of type "MatMul" constructed in this context
  # will be placed on GPU 0; all other OPs will be placed
  # on CPU 0.
```

实现

Graph.device(spec) 实现了一个栈式结构的上下文管理器，它可以接受字符串格式的设备规范，或者**设备规范函数**。事实上，当给 device 函数传递字符串，或者 DeviceSpec 时，首先会对该字符串，或 DeviceSpec 做一个简单的适配，统一转换为设备规范函数。

```
class Graph(object):
  def device(self, device_name_or_func):
    def to_device_func():
      if (device_name_or_func is not None
          and not callable(device_name_or_func)):
        return pydev.merge_device(device_name_or_func)
      else:
        return device_name_or_func

    try:
      self._device_function_stack.append(to_device_func())
      yield
    finally:
      self._device_function_stack.pop()
```

当用户使用 device 时，未显式地指定图实例，则隐式地使用全局唯一的默认图实例。也就是说，tf.device(spec) 函数事实上是对 get_default_graph().device(spec) 的一个简单包装。

```
# tensorflow/python/framework/ops.py
def device(device_name_or_function):
  return get_default_graph().device(device_name_or_function)
```

应用

在 Graph.device 实现中，pydev.merge_device 生成一个设备规范函数。该设备函数以输入的 spec 创建新的副本，通过调用 copy_spec.merge_from(current_device)，合并既有的 node_def.device 设备规范，并且 node_def.device 具有更高的优先级。实现较为隐晦，为什么 node_def.device 具有较高优先级呢？这取决于 _apply_device_functions 实现覆盖、合并、重置的需求。

```
def merge_device(spec):
  # replace string to DeviceSpec
  if not isinstance(spec, DeviceSpec):
    spec = DeviceSpec.from_string(spec or "")

  # returns a device function that merges devices specifications
  def _device_function(node_def):
    current_device = DeviceSpec.from_string(node_def.device or "")
    copy_spec = copy.copy(spec)

    # IMPORTANT: `node_def.device` takes precedence.
    copy_spec.merge_from(current_device)
    return copy_spec
  return _device_function
```

当 Graph.create_op 时，将调用 _apply_device_functions 设置 NodeDef 的设备规范。它将对 _device_function_stack 依次实施出栈操作，并调用相应的设备规范函数，并将结果直接设置为 NodeDef 的设备规范。如此，内嵌指定的 device 具有较高优先级，实现了合并、覆盖、重置外围指定的 device。

```
class Graph(object):
  def _apply_device_functions(self, op):
    for device_function in reversed(self._device_function_stack):
      if device_function is None:
        break
      # IMPORTANT: `node_def.device` takes precedence.
      op._set_device(device_function(op))
```

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

- Martin Flower

8 会话

客户端以 Session 为桥梁，与后台计算引擎建立连接，并启动计算图的执行过程。其中，通过调用 Session.run 将触发 TensorFlow 的一次计算 (Step)。

事实上，Session 建立了执行计算图的闭包环境，它封装了 OP 计算，及其 Tensor 求值的计算环境。

8.1 资源管理

在 Session 的生命周期中，将根据计算图的计算需求，按需分配系统资源，包括变量，队列，读取器等。

关闭会话

当计算完成后，需要确保 Session 被安全地关闭，以便安全释放所管理的系统资源。

```
sess = tf.Session()
sess.run(targets)
sess.close()
```

上下文管理器

一般地，常常使用上下文管理器创建 Session，使得 Session 在计算完成后，能够自动关闭，确保资源安全性地被释放。

```
with tf.Session() as sess:
  sess.run(targets)
```

图实例

一个 Session 实例，只能运行一个图实例；但是，一个图实例，可以运行在多个 Session 实例中。如果尝试在同一个 Session 运行另外一个图实例，必须先关闭 Session(不必销毁)，再启动新图的计算过程。

虽然一个 Session 实例，只能运行一个图实例。但是，可以 Session 是一个线程安全的类，可以并发地执行该图实例上的不同子图。例如，一个典型的机器学习训练模型中，可以使用同一个 Session 实例，并发地运行输入子图，训练子图，及其 Checkpoint 子图。

引用计数器

为了提高效率，避免计算图频繁地创建与销毁，存在一种实现上的优化技术。在图实例中维护一个 Session 的引用计数器，当且仅当 Session 的数目为零时，才真正地销毁图实例。

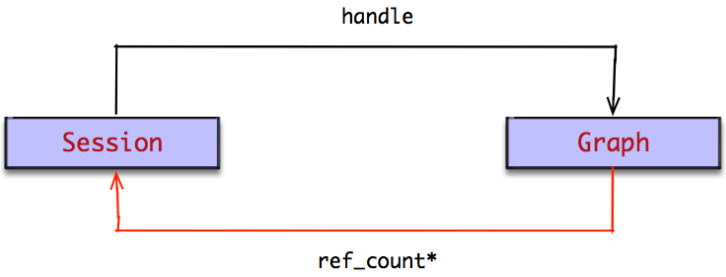


图 8-1 优化技术：会话实例的引用计数器

数据结构

此处，摘取 TF_Graph 部分关于 Session 引用计数器技术的关键字段；其中，TF_Graph 结构体定义于 C API 的头文件。

```
struct TF_Graph {
  TF_Graph();

  tensorflow::mutex mu;
  tensorflow::Graph graph GUARDED_BY(mu);

  // TF_Graph may only and must be deleted when
  // num_sessions == 0 and delete_requested == true

  // num_sessions incremented by TF_NewSession,
  // and decremented by TF_DeleteSession.
  int num_sessions GUARDED_BY(mu);
  bool delete_requested GUARDED_BY(mu);
};
```

同理，TF_Session 持有一个二元组：<tensorflow::Sesssion, TF_Graph>，它们之间是一对一的关系。其中，tensorflow::Sesssion 是 C++ 客户端侧的会话实例。

```
struct TF_Session {
  TF_Session(tensorflow::Session* s, TF_Graph* g)
      : session(s), graph(g), last_num_graph_nodes(0) {}
  tensorflow::Session* session;
  TF_Graph* graph;
  tensorflow::mutex mu;
  int last_num_graph_nodes;
};
```

创建会话

```
TF_Session* TF_NewSession(TF_Graph* graph, const TF_SessionOptions* opt,
                          TF_Status* status) {
  Session* session;
  status->status = NewSession(opt->options, &session);
  if (status->status.ok()) {
    if (graph != nullptr) {
      mutex_lock l(graph->mu);
      graph->num_sessions += 1;
    }
    return new TF_Session(session, graph);
  } else {
    DCHECK_EQ(nullptr, session);
    return nullptr;
  }
}
```

销毁会话

```
void TF_DeleteSession(TF_Session* s, TF_Status* status) {
  status->status = Status::OK();
  TF_Graph* const graph = s->graph;
  if (graph != nullptr) {
    graph->mu.lock();
    graph->num_sessions -= 1;
    const bool del = graph->delete_requested && graph->num_sessions == 0;
    graph->mu.unlock();
    if (del) delete graph;
  }
  delete s->session;
  delete s;
}
```

8.2 默认会话

通过调用 Session.as_default()，将该 Session 置为默认 Session，同时它返回了一个上下文管理器。在默认 Session 的上文中，可以直接实施 OP 的运算，或者 Tensor 的求值。

```
hello = tf.constant('hello, world')

sess = tf.Session()
with sess.as_default():
  print(hello.eval())
sess.close()
```

但是，Session.as_default() 并不会自动关闭 Session，需要用户显式地调用 Session.close 方法。

张量求值

如上例代码，hello.eval() 等价于 tf.get_default_session().run(hello)。其中，Tensor.eval 如下代码实现。

```
class Tensor(_TensorLike):
  def eval(self, feed_dict=None, session=None):
    if session is None:
      session = get_default_session()
    return session.run(tensors, feed_dict)
```

OP 运算

同理，当用户未显式提供 Session，Operation.run 将自动获取默认的 Session 实例，并按照当前 OP 的依赖关系，以某个特定的拓扑排序执行该计算子图。

```
class Operation(object):
  def run(self, feed_dict=None, session=None):
    if session is None:
      session = tf.get_default_session()
    session.run(self, feed_dict)
```

线程相关

默认会话仅仅对当前线程有效，以便在当前线程追踪 Session 的调用栈。如果在新的线程中使用默认会话，需要在线程函数中通过调用 as_default 将 Session 置为默认会话。

事实上，在 TensorFlow 运行时维护了一个 Session 的本地线程栈，实现默认 Session 的自动管理。

```
_default_session_stack = _DefaultStack()

def get_default_session(session):
  return _default_session_stack.get_default(session)
```

其中，_DefaultStack 表示栈的数据结构。

```
class _DefaultStack(threading.local):
  def __init__(self):
    super(_DefaultStack, self).__init__()
    self.stack = []

  def get_default(self):
    return self.stack[-1] if len(self.stack) >= 1 else None

  @contextlib.contextmanager
  def get_controller(self, default):
    try:
      self.stack.append(default)
      yield default
    finally:
      self.stack.remove(default)
```

8.3 会话类型

一般地，存在两种基本的会话类型：Session 与 InteractiveSession。后者常常用于交互式环境，它在构造期间将其自身置为默认，简化默认会话的管理过程。

此外，两者在运行时的配置也存在差异。例如，InteractiveSession 将 GPUOptions.allow_growth 置为 True，避免在实验环境中独占整个 GPU 的存储资源。

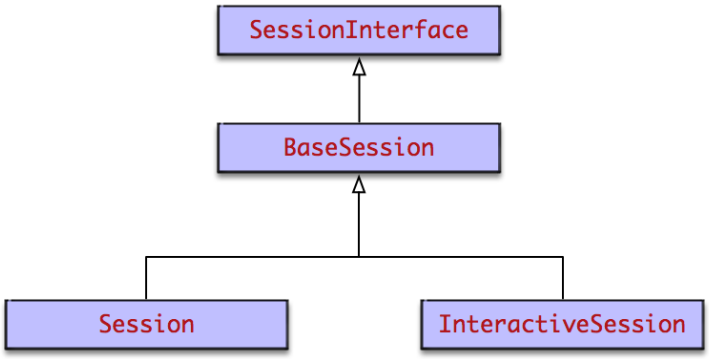


图 8-2 Session：类层次结构

Session

Session 继承 BaseSession，并增加了默认图与默认会话的上下文管理器的功能，保证系统资源的安全释放。

一般地，使用 with 进入会话的上下文管理器，并自动切换默认图与默认会话的上下文；退出 with 语句时，将自动关闭默认图与默认会话的上下文，并自动关闭会话。

```
class Session(BaseSession):
  def __init__(self, target='', graph=None, config=None):
    super(Session, self).__init__(target, graph, config=config)
    self._default_graph_context_manager = None
    self._default_session_context_manager = None

  def __enter__(self):
    self._default_graph_context_manager = self.graph.as_default()
    self._default_session_context_manager = self.as_default()

    self._default_graph_context_manager.__enter__()
    return self._default_session_context_manager.__enter__()

  def __exit__(self, exec_type, exec_value, exec_tb):
    self._default_session_context_manager.__exit__(
        exec_type, exec_value, exec_tb)
    self._default_graph_context_manager.__exit__(
        exec_type, exec_value, exec_tb)

    self._default_session_context_manager = None
    self._default_graph_context_manager = None

    self.close()
```

InteractiveSession

与 Session 不同，InteractiveSession 在构造期间将其自身置为默认，并实现默认图与默认会话的自动切换。与此相反，Session 必须借助于 with 语句才能完成该功能。在交互式环境中，InteractiveSession 简化了用户管理默认图和默认会话的过程。

同理，InteractiveSession 在计算完成后需要显式地关闭，以便安全地释放其所占用的系统资源。

```
class InteractiveSession(BaseSession):
  def __init__(self, target='', graph=None, config=None):
    super(InteractiveSession, self).__init__(target, graph, config)

    self._default_session_context_manager = self.as_default()
    self._default_session_context_manager.__enter__()

    self._default_graph_context_manager = graph.as_default()
    self._default_graph_context_manager.__enter__()

  def close(self):
    super(InteractiveSession, self).close()
    self._default_graph.__exit__(None, None, None)
    self._default_session.__exit__(None, None, None)
```

BaseSession

BaseSession 是两者的基类，它主要实现会话的创建，关闭，执行，销毁等管理生命周期的操作；它与后台计算引擎相连接，实现前后端计算的交互。

创建会话

通过调用 C API 的接口，self._session 直接持有后台计算引擎的会话句柄，后期执行计算图，关闭会话等操作都以此句柄为标识。

```
class BaseSession(SessionInterface):
  def __init__(self, target='', graph=None, config=None):
    # ignore implements...
    with errors.raise_exception_on_not_ok_status() as status:
      self._session =
        tf_session.TF_NewDeprecatedSession(opts, status)
```

执行计算图

通过调用 run 接口，实现计算图的一次计算。它首先通过 tf_session.TF_ExtendGraph 将图注册给后台计算引擎，然后再通过调用 tf_session.TF_Run 启动计算图的执行。

```
class BaseSession(SessionInterface):
  def run(self,
    fetches, feed_dict=None, options=None, run_metadata=None):
    self._extend_graph()
    with errors.raise_exception_on_not_ok_status() as status:
      return tf_session.TF_Run(session,
        options, feed_dict, fetch_list,
        target_list, status, run_metadata)

  def _extend_graph(self):
    with errors.raise_exception_on_not_ok_status() as status:
      tf_session.TF_ExtendGraph(self._session,
        graph_def.SerializeToString(), status)
```

关闭会话

```
class BaseSession(SessionInterface):
  def close(self):
    with errors.raise_exception_on_not_ok_status() as status:
      tf_session.TF_CloseDeprecatedSession(self._session, status)
```

销毁会话

```
class BaseSession(SessionInterface):
  def __del__(self):
    try:
      status = tf_session.TF_NewStatus()
      tf_session.TF_DeleteDeprecatedSession(self._session, status)
    finally:
      tf_session.TF_DeleteStatus(status)
```

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

- Martin Flower

9 变量

Variable 是一个特殊的 OP，它拥有状态 (Stateful)。从实现技术探究，Variable 的 Kernel 实现直接持有一个 Tensor 实例，其生命周期与变量一致。相对于普通的 Tensor 实例，其生命周期仅对本次迭代 (Step) 有效；而 Variable 对多个迭代都有效，甚至可以存储到文件系统，或从文件系统中恢复。

9.1 实战：线性模型

以一个简单的线性模型为例 (为了简化问题，此处省略了训练子图)。首先，使用 `tf.placeholder` 定义模型的输入，然后定义了两个全局变量，同时它们都是训练参数，最后定义了一个简单的线性模型。

```
x  = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784,10]), name='W')
b = tf.Variable(tf.zeros([10]), name='b')
y = tf.matmul(x, W) + b
```

在使用变量之前，必须对变量进行初始化。按照习惯用法，使用 `tf.global_variables_initializer()` 将所有全局变量的初始化器汇总，并对其进行初始化。

```
init = tf.global_variables_initializer()

with tf.Session() as sess:
  sess.run(init)
```

按照既有经验，其计算图大致如图**??**（第**??**页）所示。

事实上，正如图**??**（第**??**页）所示，实际的计算图要复杂得多，让我们从头说起。

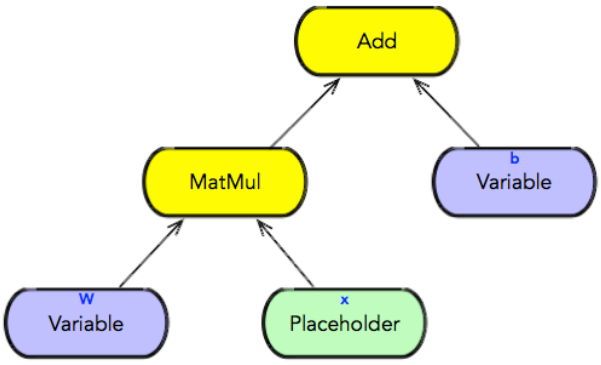


图 9-1 计算图：线性加权和

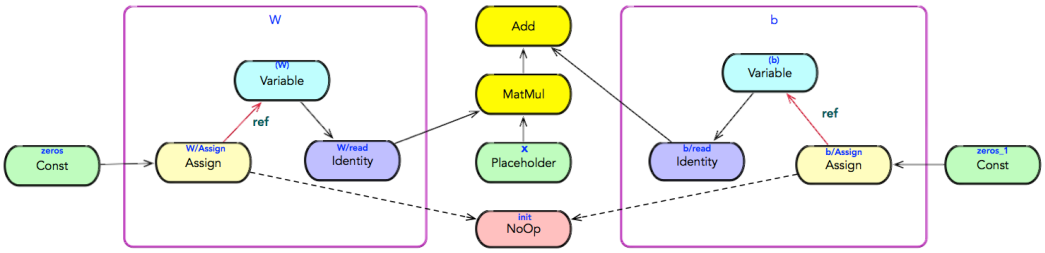


图 9-2 计算图：线性加权和

9.2 初始化模型

Variable 是一个特殊的 OP，它拥有状态 (Stateful)。如果从实现技术探究，Variable 的 Kernel 实现直接持有一个 Tensor 实例，其生命周期与 Variable 一致。相对于普通的 Tensor 实例，其生命周期仅对本次迭代 (Step) 有效；而 Variable 对多个迭代 (Step) 都有效，甚至可以持久化到文件系统，或从文件系统中恢复。

操作变量

存在几个操作 Variable 的特殊 OP 用于修改变量的值，例如 Assign, AssignAdd 等。Variable 所持有的 Tensor 以引用的方式输入到 Assign 中，Assign 根据初始值 (Initial Value) 或新值，就地修改 Tensor 内部的值，最后以引用的方式输出该 Tensor。

从设计角度看，Variable 可以看做 Tensor 的包装器，Tensor 所支持的所有操作都被 Variable 重载实现。也就是说，Variable 可以出现在 Tensor 的所有地方。例如，

```
# Create a variable
W = tf.Variable(tf.zeros([784,10]), name='W')

# Use the variable in the graph like any Tensor.
y = tf.matmul(x, W)
```

```
# The overloaded operators are available too.
z = tf.sigmoid(w + y)

# Assign a new value to the variable with assign/assign_add.
w.assign(w + 1.0)
w.assign_add(1.0)
```

初始值

一般地，在使用变量之前，必须对变量进行初始化。事实上，TensorFlow 设计了一个精巧的变量初始化模型。Variable 根据初始值 (Initial Value) 推演 Variable 的数据类型，并确定 Tensor 的形状 (Shape)。

例如，`tf.zeros` 称为 Variable 的初始值，它确定了 Variable 的类型为 `int32`，且 Shape 为 `[784, 10]`。

```
# Create a variable.
W = tf.Variable(tf.zeros([784,10]), name='W')
```

如下表所示，构造变量初始值的常见 OP 包括：

初始化器

另外，变量通过初始化器 (Initializer) 在初始化期间，将初始化值赋予 Variable 内部所持有 Tensor，完成 Variable 的就地修改。

在变量使用之前，必须保证变量被初始化器已初始化。事实上，变量初始化过程，即运行变量的初始化器。

证如上例 `W` 的定义，可以如下完成 `W` 的初始化。此处，`W.initializer` 实际上为 Assign 的 OP，这是 Variable 默认的初始化器。

```
# Run the variable initializer.
with tf.Session() as sess:
  sess.run(W.initializer)
```

一旦完成 Variable 的初始化，其类型与值得以确定。随后可以使用 Assign 族的 OP(例如 Assign, AssignAdd 等) 修改 Variable 的值。

需要注意的是，在 TensorBoard 中展示 Assign 的输入，其边使用特殊的 ref 标识。数据流向与之刚好相反，否则计算图必然出现环，显然违反了 DAG(有向无环图) 的基本需求。

快照

如果要读取变量的值，则通过 Identity 恒等变化，直接输出变量所持有的 Tensor。Identity 去除了 Variable 的引用标识，同时也避免了内存拷贝。

Identity 操作 Variable 常称为一个快照 (Snapshot)，表示 Variable 当前的值。

事实上，通过 Identity 将 Variable 转变为普通的 Tensor，使得它能够兼容所有 Tensor 的操作。

变量子图

例如，变量 W 的定义如下。

```
W = tf.Variable(tf.zeros([784,10]), name='W')
```

`tf.zeros([784,10])` 常称为初始值，它通过初始化器 Assign，将 `W` 内部持有的 Tensor 以引用的形式就地修改为该初始值；同时，Identity 去除了 Variable 的引用标识，实现了 Variable 的读取。

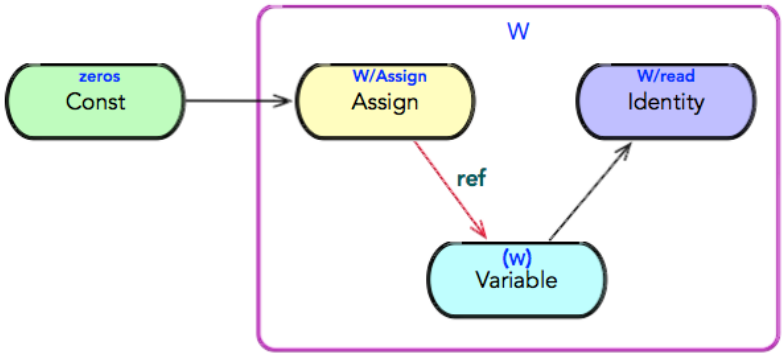


图 9-3 变量子图

初始化过程

更为常见的是，通过调用 `tf.global_variables_initializer()` 将所有变量的初始化器进行汇总，然后启动 Session 运行该 OP。

```
init = tf.global_variables_initializer()
```

事实上，搜集所有全局变量的初始化器的 OP 是一个 NoOp，即不存在输入，也不存在输出。所有变量的初始化器通过控制依赖边与该 NoOp 相连，保证所有的全局变量被初始化。

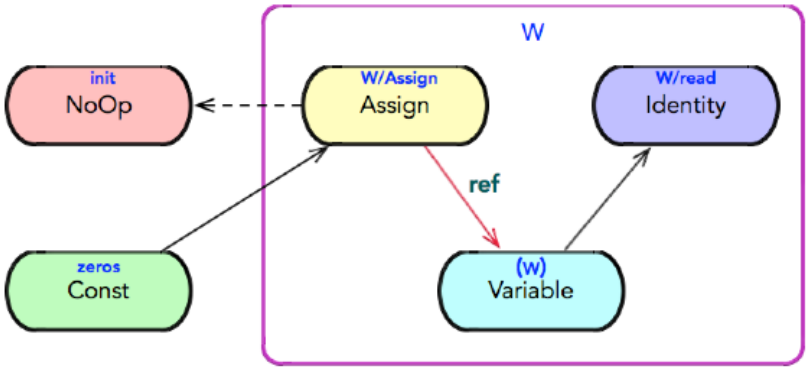


图 9-4 初始化 OP

同位关系

同位关系是一种特殊的设备约束关系。显而易见，Assign, Identity 这两个 OP 与 Variable 关系极其紧密，分别实现了变量的修改与读取。因此，它们必须与 Variable 在同一个设备上执行；这样的关系，常称为同位关系 (Colocation)。

可以在 Assign/Identity 节点上指定 `_class` 属性值：`[s: "loc:@W"]`，它表示这两个 OP 与 `W` 放在同一个设备上运行。

例如，以 `W/read` 节点为例，该节点增加了 `_class` 属性，指示与 `W` 的同位关系。

```
node {
  name: "W/read"
  op: "Identity"
  input: "W"
  attr {
    key: "T"
    value {
      type: DT_FLOAT
    }
  }
  attr {
    key: "_class"
    value {
      list {
        s: "loc:@W"
      }
    }
  }
}
```

初始化依赖

如果一个变量初始化需要依赖于另外一个变量的初始值，则需要特殊地处理。例如，变量 `V` 的初始值依赖于 `W` 的初始值，可以通过 `W.initialized_value()` 指定。

```
W = tf.Variable(tf.zeros([784,10]), name='W')
V = tf.Variable(W.initialized_value(), name='V')
```

事实上，两者通过 Identity 衔接，并显式地添加了依赖控制边，保证 `W` 在 `V` 之前初始化。此处，存在两个 Identity 的 OP，但职责不一样，它们分别完成初始化依赖和变量读取。

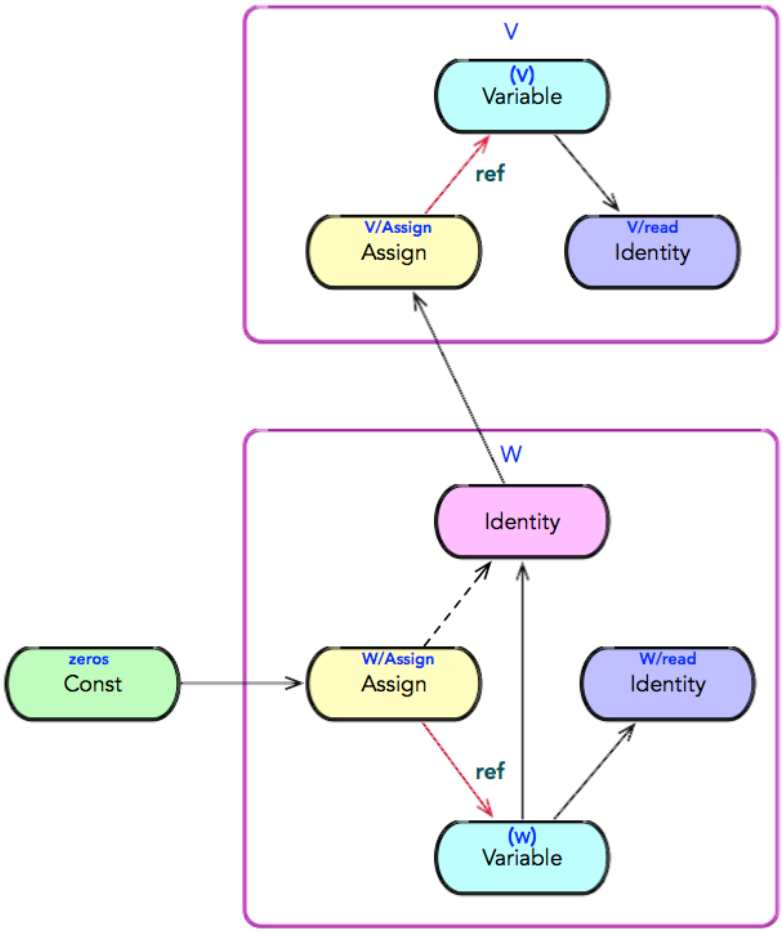


图 9-5 初始化依赖

同样地，可以通过调用 `tf.global_variables_initializer()` 将变量的所有初始化器进行汇总，然后启动 Session 完成所有变量的初始化。

```
init = tf.global_variables_initializer()
```

按照依赖关系，因为增加了 W/Assign 与 Identity 之间的控制依赖边，从而巧妙地实现了 W 在 V 之前完成初始化，并通过 W 当前的初始化值，最终完成 V 的初始化。

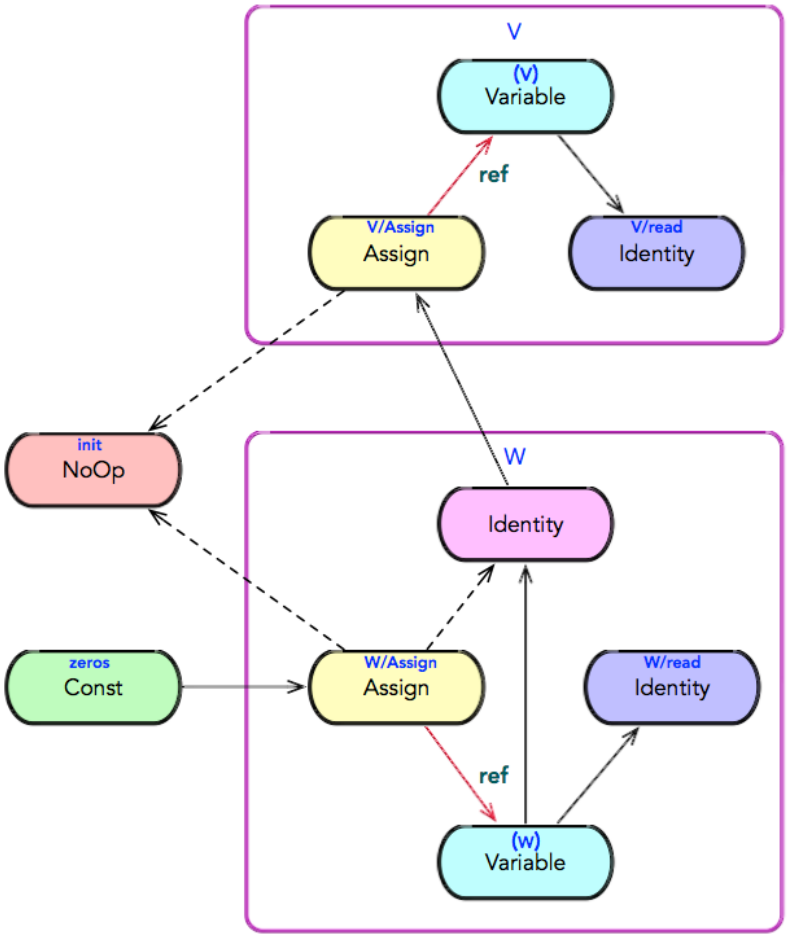


图 9-6 初始化 OP

初始化器列表

可以使用 variables_initializer 构建变量列表的初始化器列表。其中，group 将构造一个仅控制依赖于 _initialier_list() 的 NoOP。

```
def variables_initializer(var_list, name="init"):
  def _initialier_list():
    return *[v.initializer for v in var_list]
  return control_flow_ops.group(_initialier_list(), name=name)
```

例如，全局变量列表的初始化器列表可以如下构造。

```
def global_variables_initializer():
  return variables_initializer(global_variables())
```

9.3 变量分组

默认地，Variable 被划分在全局变量和训练变量的集合中。正如上例，`W`, `V` 自动划分至全局变量和训练变量的集合中。

全局变量

可以通过 `tf.global_variables()` 方便地检索全局变量的集合。在分布式环境中，全局变量能在不同的进程间实现参数共享。

```
def global_variables():
  return ops.get_collection(ops.GraphKeys.GLOBAL_VARIABLES)
```

本地变量

可以通过 `tf.local_variables()` 方便地检索本地变量的集合。

```
def local_variables():
  return ops.get_collection(ops.GraphKeys.LOCAL_VARIABLES)
```

可以使用 `local_variable` 的语法糖，构建一个本地变量。

```
def local_variable(initial_value, validate_shape=True, name=None):
  return variables.Variable(
      initial_value, trainable=False,
      collections=[ops.GraphKeys.LOCAL_VARIABLES],
      validate_shape=validate_shape, name=name)
```

本地变量表示进程内的共享变量，它通常不需要做断点恢复 (Checkpoint)，仅用于临时的计数器的用途。例如，在分布式环境中，使用本地变量记录该进程已读数据的 Epoch 数目。

训练变量

可以通过 tf.trainable_variables() 检索训练变量的集合。在机器学习中，训练变量表示模型参数。

```
def trainable_variables():
  return ops.get_collection(ops.GraphKeys.TRAINABLE_VARIABLES)
```

global_step

global_step 是一个特殊的 Variable，它不是训练变量，但它是一个全局变量。在分布式环境中，global_step 常用于追踪已运行 step 的次数，并在不同进程间实现数据的同步。

创建一个 global_step 可以使用如下函数：

```
def create_global_step(graph=None):
  graph = ops.get_default_graph() if graph is None else graph
  with graph.as_default() as g, g.name_scope(None):
    collections = [GLOBAL_VARIABLES, GLOBAL_STEP]
    return variable(
        GLOBAL_STEP,
        shape=[],
        dtype=dtypes.int64,
        initializer=init_ops.zeros_initializer(),
        trainable=False,
        collections=collections)
```

9.4 源码分析：构造变量

为了简化代码实现，此处对 Variable 做了简单的重构。

```
class Variable(object):
  def __init__(self, initial_value=None, trainable=True,
    collections=None, name=None, dtype=None):
    with ops.name_scope(name, "Variable", [initial_value]) as name:
      self._cons_initial_value(initial_value, dtype)
      self._cons_variable(name)
      self._cons_initializer()
      self._cons_snapshot()
    self._cons_collections(trainable, collections)
```

构造 Variable 实例，基本包括如下几个步骤：

构造初始值

```
def _cons_initial_value(self, initial_value, dtype):
  self._initial_value = ops.convert_to_tensor(
      initial_value, name="initial_value", dtype=dtype)
```

构造变量 OP

Variable 根据初始值的类型和大小完成自动推演。

```
def _cons_variable(self, name):
  self._variable = state_ops.variable_op_v2(
    self._initial_value.get_shape(),
    self._initial_value.dtype.base_dtype,
    name=name)
```

构造初始化器

Variable 的初始化器本质上是一个 Assign，它持有 Variable 的引用，并使用初始值就地修改变量本身。

```
def _cons_initializer(self):
  self._initializer_op = state_ops.assign(
    self._variable,
    self._initial_value).op
```

构造快照

Variable 的快照本质上是一个 Identity，表示 Variable 的当前值。

```
def _cons_snapshot(self):
  with ops.colocate_with(self._variable.op):
    self._snapshot = array_ops.identity(
      self._variable, name="read")
```

变量分组

默认地，Variable 被划分在全局变量的集合中；如果 `trainable` 为真，则表示该变量为训练参数，并将其划分到训练变量的集合中。

```
def _cons_collections(self, trainable, collections)
  if collections is None:
    collections = [GLOBAL_VARIABLES]
  if trainable and TRAINABLE_VARIABLES not in collections:
    collections = list(collections) + [TRAINABLE_VARIABLES]
  ops.add_to_collections(collections, self)
```

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

- Martin Flower

10

队列

TensorFlow 的 `Session` 是线程安全的。也就是说，多个线程可以使用同一个 `Session` 实例，并发地执行同一个图实例的不同 OP；TensorFlow 执行引擎会根据输入与输出对图实施剪枝，得到一个最小依赖的子图。

因此，通过多线程并使用同一个 `Session` 实例，并发地执行同一个图实例的不同 OP，最终实现的效果是子图之间的并发执行。

对于典型的模型训练，可以充分发挥 `Session` 多线程的并发能力，提升训练的性能。例如，输入子图运行在一个单独的线程中，用于准备样本数据；而训练子图则运行在另外一个单独的线程中，并按照 `batch_size` 的大小一个批次取走训练样本，并启动迭代的训练过程。

本文将讲解上述并发模型中的基础设施，包括队列，多线程的协调器，及其控制 `Enqueue OP` 执行的 QueueRunner。

10.1 队列

在 TensorFlow 的执行引擎中，Queue 是一种控制异步计算的强大工具。特殊地，Queue 是一种特殊的 OP，与 Variable 类似，它是一类有状态的 OP。

与之类似，Variable 拥有关联的 Assign 等修改其状态的 OP，Queue 也有与之关联的 OP，例如 `Enqueue, Dequeue, EnqueueMany, DequeueMany` 等 OP，它们都能直接修改 Queue 的状态。

FIFOQueue

举一个简单例子。首先，构造了一个 `FIFOQueue` 队列；然后，在计算图中添加了一个 `EnqueueMany`，该 OP 用于在队列头部追加 1 个或多个元素；其次，再添加一个出队的 `Dequeue`；最后，将出队元素的值增加 1，再将其结果入队。在启动计算图执行之前，计算图

的构造如下图所示。

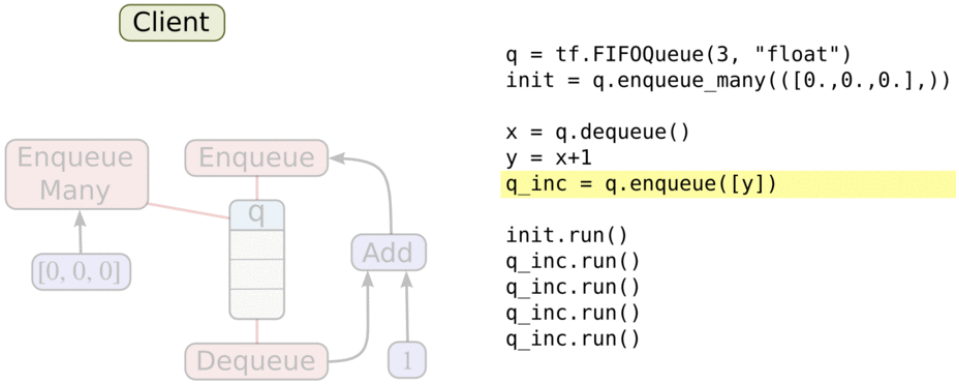


图 10-1 图构造期

执行 EnqueueMany 操作后，计算图的状态如下图所示。

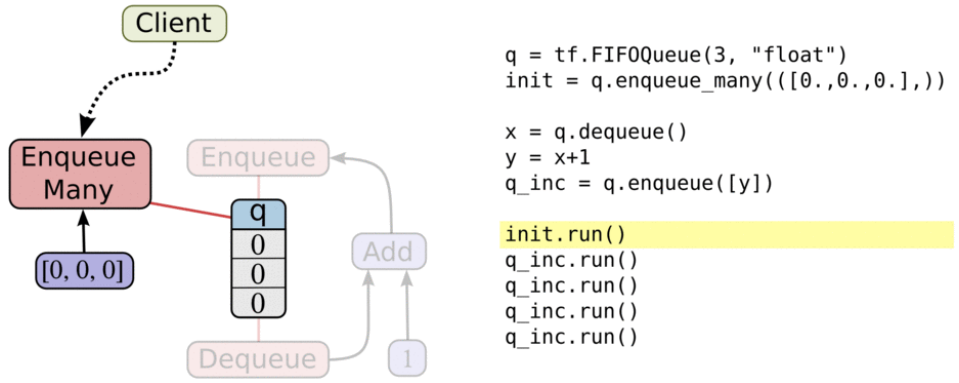


图 10-2 图执行期：执行一次 EnqueueMany

执行第一步 Enqueue 后，计算图的状态如下图所示。

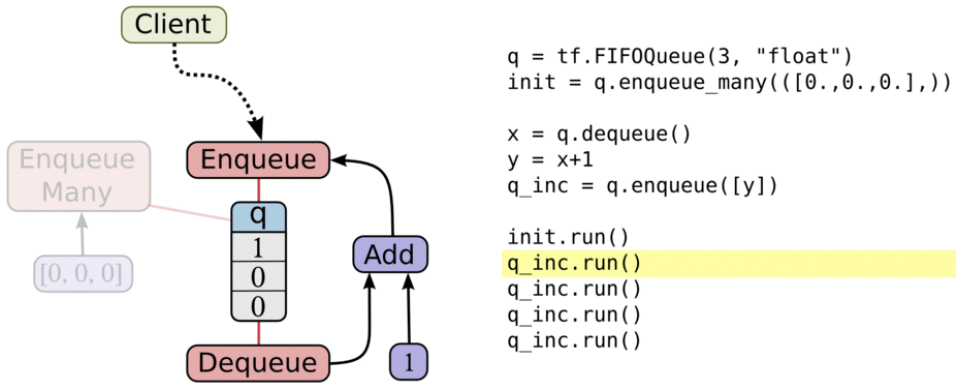


图 10-3 图执行期：执行一次 Enqueue

用途

队列在模型训练中扮演重要角色，后文将讲述数据加载的 Pipeline，训练模型常常使用 RandomShuffleQueue 为其准备样本数据。为了提高 IO 的吞吐率，可以使用多线程，并发地将样本数据追加到样本队列中；与此同时，训练模型的线程迭代执行 train_op 时，一次获取 batch_size 大小的批次样本数据。

显而易见，队列在 Pipeline 过程中扮演了异步协调和数据交换的功能，这给 Pipeline 的设计和实现带来很大的弹性空间。

需要注意的是，为了使得队列在多线程最大化发挥作用，需要解决两个棘手的问题：

1. 如何同时停止所有的线程，及其处理异常报告？
2. 如何并发地向队列中追加样本数据？

因此，TensorFlow 设计了 tf.train.Coordinator 和 tf.train.QueueRunner 两个类，分别解决上述两个问题。

这两个类相辅相成，Coordinator 协调多个线程同时停止运行，并且向等待停止通知的主程序报告异常；而 QueueRunner 创建了一组线程，并协作多个入队 OP(例如 Enqueue, EnqueueMany) 的执行。

10.2 协调器

Coordinator 提供了一种同时停止一组线程执行的简单机制。它拥有 3 个重要的方法：

1. should_stop: 判断当前线程是否应该退出
2. request_stop: 请求所有线程停止执行
3. join: 等待所有线程停止执行

使用方法

一般地，主程序常常使用如下模式使用 Coordinator。

```
# Create a coordinator.
coord = tf.train.Coordinator()

# Create 10 threads that run 'MyLoop()'
threads = [threading.Thread(target=MyLoop, args=(coord,))
          for i in xrange(10)]
```

```
# Start the threads.
for t in threads:
  t.start()

# wait for all of them to stop
coord.join(threads)
```

任何子线程，都可以通过调用 coord.request_stop，通知其他线程停止执行。因此，每个线程的迭代执行中，都要事先检查 coord.should_stop()。一旦 coord.request_stop 被调用，其他线程的 coord.request_stop() 将立即返回 True。

一般地，一个子线程的迭代执行方法遵循如下实现模式。

```
def MyLoop(coord):
  try
    while not coord.should_stop():
      # ...do something...
  except Exception as e:
    coord.request_stop(e)
```

异常处理

当某个线程发生了异常，则可以通过 coord.request_stop(e) 报告异常的发生。

```
try:
  while not coord.should_stop():
    # ...do some work...
except Exception as e:
  coord.request_stop(e)
```

为了消除异常代码处理的重复代码，可以使用 coord.stop_on_exception() 的上下文管理器。

```
with coord.stop_on_exception():
  while not coord.should_stop():
    # ...do some work...
```

其中，该异常也会在 coord.join 中被重新抛出。因此，在主程序也需要合理地处理异常。

```
try:
  # Create a coordinator.
  coord = tf.train.Coordinator()

  # Create 10 threads that run 'MyLoop()'
  threads = [threading.Thread(target=MyLoop, args=(coord,))
             for i in xrange(10)]
```

```
    # Start the threads.
    for t in threads:
      t.start()

    # wait for all of them to stop
    coord.join(threads)
  except Exception as e:
    # ...exception that was passed to coord.request_stop(e)
```

实战：LoopThread

10.3 QueueRunner

一个 QueueRunner 实例持有一个或多个 Enqueue 的入队 OP，它为每个 Enqueue OP 启动一个线程。

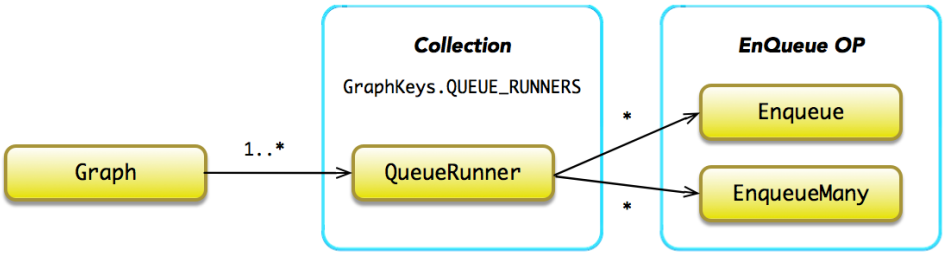


图 10-4 TensorFlow 系统架构

注册 QueueRunner

可以调用 tf.train.add_queue_runner 往计算图中注册 QueueRunner 实例，并且将其添加到 GraphKeys.QUEUE_RUNNERS 集合中。

```
def add_queue_runner(qr, collection=ops.GraphKeys.QUEUE_RUNNERS):
  ops.add_to_collection(collection, qr)
```

执行 QueueRunner

可以调用 tf.train.start_queue_runners 时，它会从计算图中找到所有 QueueRunner 实例，并从 QueueRunner 实例中取出所有 Enqueue OP，为每个 OP 启动一个线程。

```
def start_queue_runners(sess, coord, daemon=True, start=True,
                        collection=ops.GraphKeys.QUEUE_RUNNERS):
  with sess.graph.as_default():
    threads = []
```

```
    for qr in ops.get_collection(collection):
      threads.extend(qr.create_threads(
          sess, coord=coord, daemon=daemon, start=start))
  return threads
```

在 QueueRunner.create_threads 方法中，为其包含的每个 Enqueue 类型的 OP 启动单独的线程。

```
class QueueRunner(object):
  def create_threads(self, sess, coord, daemon, start):
    """Create threads to run the enqueue ops.
    """
    threads = [threading.Thread(
        target=self._run, args=(sess, op, coord))
        for op in self._enqueue_ops]
    if coord:
      threads.append(threading.Thread(
          target=self._close_on_stop,
          args=(sess, self._cancel_op, coord)))
    for t in threads:
      if coord:
        coord.register_thread(t)
      if daemon:
        t.daemon = daemon
      if start:
        t.start()
    return threads
```

迭代执行 Enqueue

每个 Enqueue 子线程将迭代执行 Enqueue OP。当发生 OutOfRangeError 异常时，将自动关闭队列，并退出子线程；但是，如果发生其他类型的异常，会主动通知 Coordinator 停止所有线程的运行，并退出子线程。

```
class QueueRunner(object):
  def _run(self, sess, enqueue_op, coord):
    try:
      enqueue_callable = sess.make_callable(enqueue_op)
      while True:
        if coord.should_stop():
          break
        try:
          enqueue_callable()
        except errors.OutOfRangeError:
          sess.run(self._close_op)
          return
    except Exception as e:
      coord.request_stop(e)
```

监听队列关闭

另外,如果给定 Coordinator 实例,QueueRunner 还会额外启动一个线程;当 Coordinator 实例被触发调用 request_stop 方法后，该线程将会自动关闭队列。

```
class QueueRunner(object):
  def _close_on_stop(self, sess, cancel_op, coord):
    """Close the queue, and cancel pending enqueue ops
       when the Coordinator requests stop.
    """
    coord.wait_for_stop()
    try:
      sess.run(cancel_op)
    except Exception:
      pass
```

其中，Queue 的 Cancel OP 与 Close OP 都会关闭队列，但是 Cancel OP 会撤销已缓存的 Enqueue OP 列表，但 Close OP 则保留已缓存的 Enqueue OP 列表。

关闭队列

当队列被关闭后，对于任何尝试 Enqueue 将会产生错误。但是，对于任何尝试 Dequeue 依然是成功的，只要队列中遗留元素；否则，Dequeue 将立即失败，抛出 OutOfRangeError 异常，而不会阻塞等待更多元素被入队。

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

- Martin Flower

11

OP 本质论

11.1 OP 的注册

在 C++ 后端系统，系统初始化时，系统完成所有 OP 的注册。OP 的注册是通过 `REGISTER_OP` 宏完成的。

REGISTER_OP

实施上，`REGISTER_OP` 定义了一套精致的内部 DSL，系统自动完成字符串表示的翻译表达，并将其转换为 `OpDef` 的内部表示，最后保存在 `OpDef` 的仓库中。

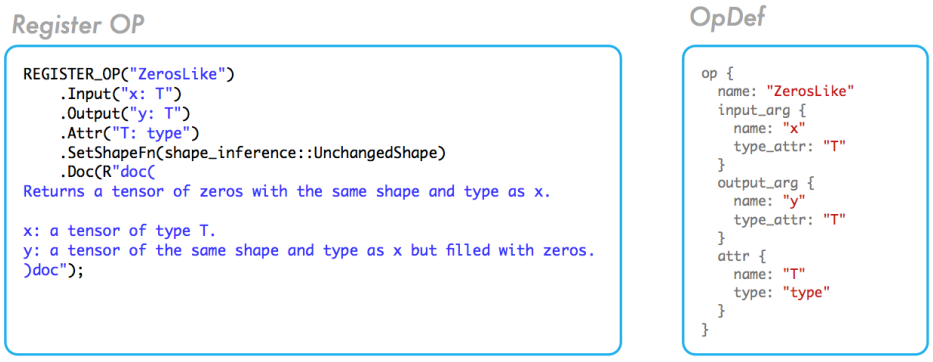


图 11-1 REGISTER_OP：注册 OP 的实用宏

查询接口

```
struct OpRegistryInterface {
  virtual ~OpRegistryInterface() {}

  virtual Status LookUp(
    const string& op_name,
    const OpRegistrationData** op_reg_data) const = 0;
```

```
  Status LookUpOpDef(const string& op_name, const OpDef** op_def) const;
};
```

其中，OpRegistrationData 描述了 OP 两种基本的元数据：OpDef 与 OpShapeInferenceFn；前者用于描述 OP 的输入/输出参数信息，属性名列表，及其约束关系。后者用于描述 OP 的 Shape 的推演规则。

```
struct OpRegistrationData {
  OpDef op_def;
  OpShapeInferenceFn shape_inference_fn;
};

using OpRegistrationDataFactory =
  std::function<Status(OpRegistrationData*)>;
```

OpDef 仓库

实现中，采用延迟初始化的技术。为了简化问题的描述，此处做了简单的代码重构，以便帮助大家理解 OpRegistry 的工作原理。

```
struct OpRegistry : OpRegistryInterface {
  OpRegistry();
  ~OpRegistry() override;

  void Register(const OpRegistrationDataFactory& factory);

 private:
  Status LookUp(
     const string& op_name,
     const OpRegistrationData** op_reg_data) const override;

 private:
  using Registry =
    std::unordered_map<string, OpRegistrationData*>;

  mutex mu_;
  Registry registry_;
};
```

```
Status OpRegistry::Register(
  const OpRegistrationDataFactory& factory) {
  auto op_reg_data(std::make_unique<OpRegistrationData>());
  Status s = factory(op_reg_data.get());
  if (s.ok()) {
    gtl::InsertIfNotPresent(&registry_,
      op_reg_data->op_def.name(),
      op_reg_data.get())
  }
  if (s.ok()) {
    op_reg_data.release();
  } else {
    op_reg_data.reset();
  }
  return watcher_status;
}
```

第 IV 部分

运行模型

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

- Martin Flower

12

本地执行

TensorFlow 可以独立地运行在一个进程内，完成计算图的执行过程。本章将重点介绍本地运行时的基本架构与运行机制；重点讨论计算图剪枝、分裂、优化、执行等实现技术细节；并且详细探究在本地模式下，跨设备间 OP 之间数据交互的工作机制，及其 OP 在设备集上的编排 (placement) 算法。

12.1 本地模式

如图**??**（第**??**页）所示，在本地模式下，Client, Master, Worker 部署在同一台机器同一进程内，并由 `DirectSession` 同时扮演这三个角色。`DirectSession` 运行在单独的进程内，各服务实体之间是函数调用关系。

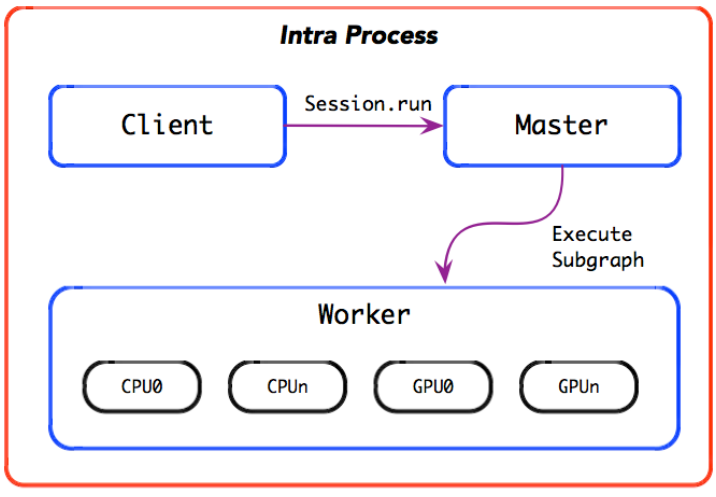


图 12-1 本地模式

Client 负责计算图的构造，通过调用 `Session.run`，启动计算图的执行过程。如图**??**（第**??**页）所示，在 `run_step` 执行过程之中，涉及计算图的剪枝、分裂、执行三个重要阶段。

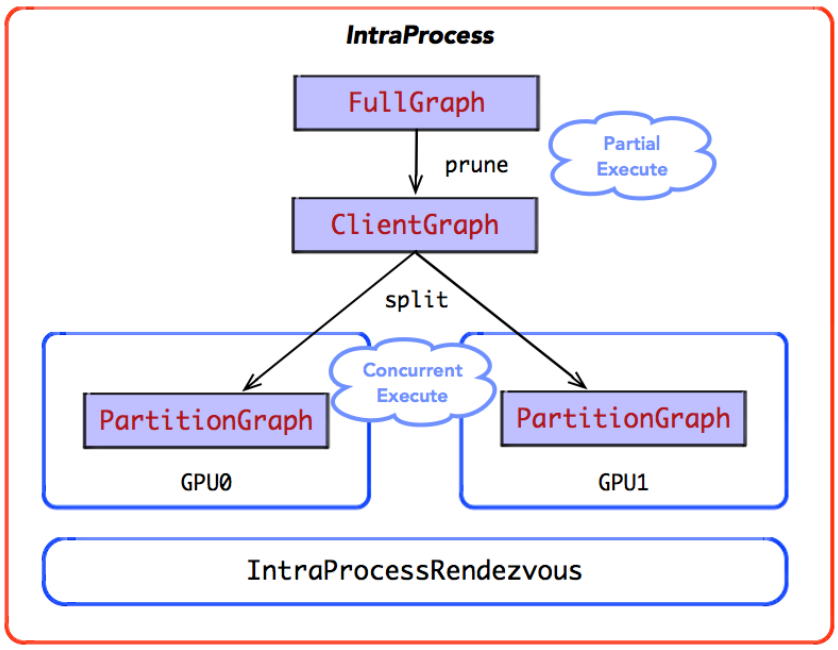


图 12-2 本地模式：图操作

部分执行

Master 收到计算图执行命令后，启动计算图的剪枝操作。它根据计算图的输入输出反向遍历图，寻找一个最小依赖的子图，常称为 `ClientGraph`。

也就是说，每次执行 `run_step` 时，并不会执行整个计算图 (`FullGraph`)，而是执行部分的子图。剪枝体现了 TensorFlow 部分执行的设计理念。

并发执行

然后，运行时按照当前设备集完成图的分裂，生成了很多子图，每个子图称为 `PartitionGraph`；然后触发各个 Worker 并发地执行每个 `PartitionGraph`；对于每一个 PartitionGraph，运行时将启动一个 Executor，按照其拓扑排序完成 `PartitionGraph` 的执行。

也就是说，分裂和执行体现了 TensorFlow 并发执行的设计理念。

12.2 会话控制

在本地模式下，其运行时由 `DirectSession` 控制。一般地，`DirectSession` 执行计算图时，各组件之间都是函数调用关系。但是，`DirectSession` 也存在清晰的生命周期管理机制，如图**??**（第**??**页）所示。

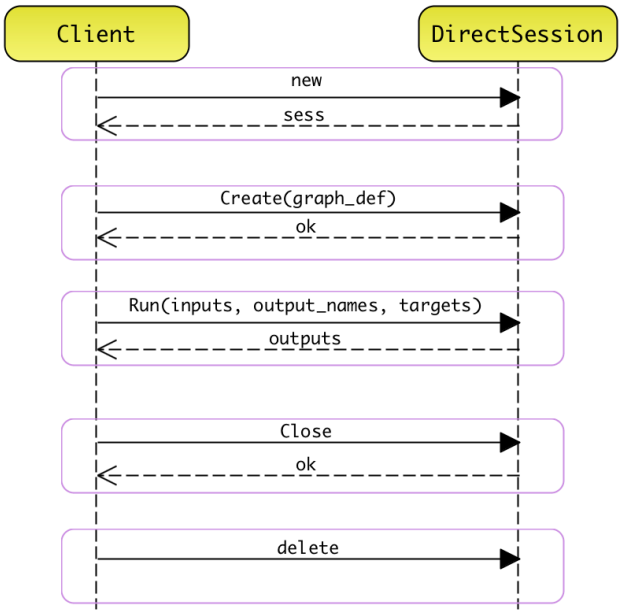


图 12-3 DirectSession 生命周期

领域模型

如图**??**（第**??**页）所示，DirectSession 持有 SimpleGraphExecutionState 实例，后者负责计算图的剪枝，生成 ClientGraph 实例。

DirectSession 同时持有一组线程池，但是没次 DirectSession.run 时，根据外部配置的索引，从线程池组里选择其一为其提供服务。因为 DirectSession 是线程安全的，支持多个并发执行的 DirectSession.run，即可以同时运行多个线程池实例。

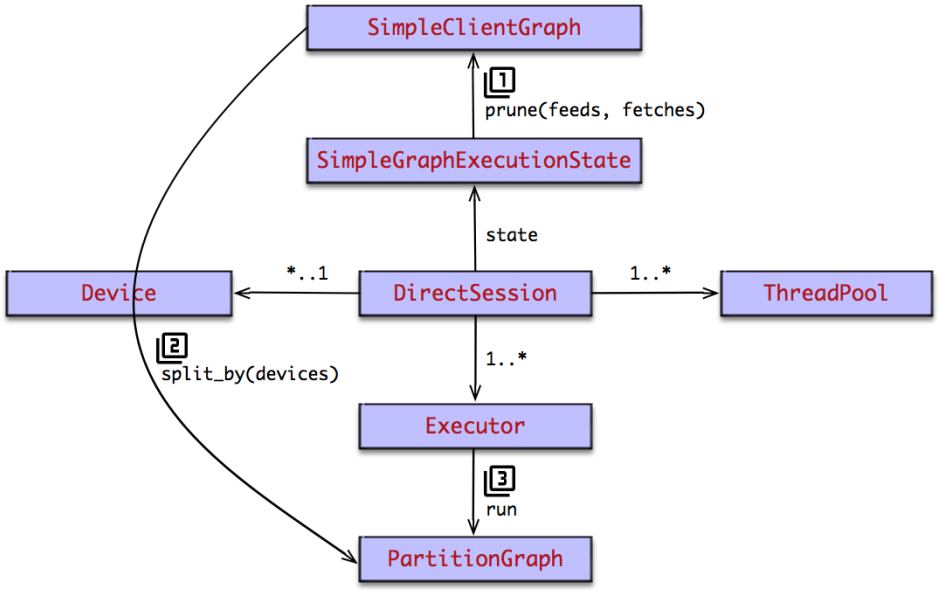


图 12-4 DirectSession 领域模型

创建会话

如图??（第??页）所示，DirectSession 由 DirectSessionFactory 多态创建。其中，DeviceFactory::AddDevices 将创建本地设备集。

其中，DirectSession 中主要完成线程池组的创建。

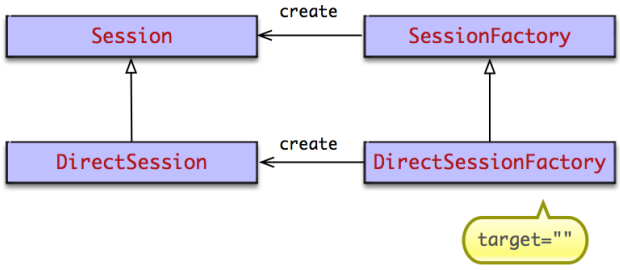


图 12-5 多态创建 DirectSession

```
struct DirectSessionFactory : SessionFactory {
  bool AcceptsOptions(const SessionOptions& options) override {
    return options.target.empty();
  }

  Session* NewSession(const SessionOptions& options) override {
    std::vector<Device*> devices;
    DeviceFactory::AddDevices(
        options, "/job:localhost/replica:0/task:0", &devices);
    return new DirectSession(options, new DeviceMgr(devices));
  }
};
```

其中，DirectSessionFactory::NewSession 由 C API 调用。

```
Status NewSession(const SessionOptions& options, Session** out_session) {
  SessionFactory* factory;
  Status s = SessionFactory::GetFactory(options, &factory);
  if (!s.ok()) {
    *out_session = nullptr;
    return s;
  }
  *out_session = factory->NewSession(options);
  if (!*out_session) {
    return errors::Internal("Failed to create session.");
  }
  return Status::OK();
}

TF_DeprecatedSession* TF_NewDeprecatedSession(
  const TF_SessionOptions* opt, TF_Status* status) {
  Session* session;
  status->status = NewSession(opt->options, &session);
  if (status->status.ok()) {
    return new TF_DeprecatedSession({session});
  } else {
    return nullptr;
  }
}
```

在 DirectSession 的构造函数中，主要负责其领域模型的初始化，包括线程池的创建，构建 CancellationManager 实例。

```
DirectSession::DirectSession(
    const SessionOptions& options,
    const DeviceMgr* device_mgr)
    : options_(options),
      device_mgr_(device_mgr),
      cancellation_manager_(new CancellationManager()) {
  // thread_pools_ = ...
}
```

销毁会话

由 SessionFactory 所 new 出来的 DirectSession，由 C API 负责 delete 掉。

```
void TF_DeleteDeprecatedSession(TF_DeprecatedSession* s, TF_Status* status) {
  status->status = Status::OK();
  delete s->session;  // delete DirectSession
  delete s;
}
```

随后，DirectSession 的析构函数被调用，它负责清理其负责管理的系统资源。主要包括 Executor 列表，ThreadPool 列表，CancellationManager 实例。

```
DirectSession::~DirectSession() {
  for (auto& it : partial_runs_) {
    it.second.reset(nullptr);
  }

  for (auto& it : executors_) {
    it.second.reset();
  }

  for (auto d : device_mgr_->ListDevices()) {
    d->op_segment()->RemoveHold(session_handle_);
  }

  delete cancellation_manager_;

  for (const auto& p_and_owned : thread_pools_) {
    if (p_and_owned.second) delete p_and_owned.first;
  }

  execution_state_.reset(nullptr);
  flib_def_.reset(nullptr);
}
```

创建/扩展图

首次扩展图，等价于创建图。扩展图就是在原有计算图的基础上，追加新的子图。当然，追加的子图中所包含的节点，在原有的计算图中不应该存在。

```
Status DirectSession::Create(const GraphDef& graph) {
  if (graph.node_size() > 0) {
    mutex_lock l(graph_def_lock_);
    return ExtendLocked(graph);
  }
  return Status::OK();
}

Status DirectSession::Extend(const GraphDef& graph) {
  mutex_lock l(graph_def_lock_);
  return ExtendLocked(graph);
}
```

当创建计算图时，DirectSession 主要完成 SimpleGraphExecutionState 实例的创建。如图**??**（第**??**页）所示，SimpleGraphExecutionState 实例持有 FullGraph 两种格式的实例：Graph 与 GraphDef，并由它负责管理和维护 FullGraph 的生命周期。

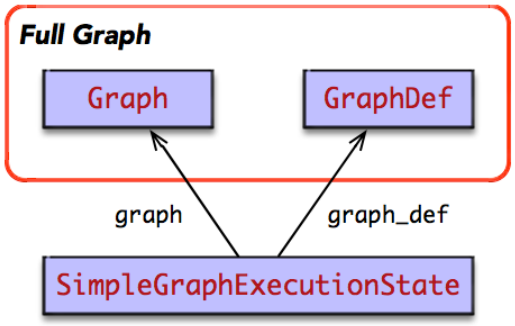


图 12-6 创建 SimpleGraphExecutionState 实例

其中，SimpleGraphExecutionState 的主要职责包括：

1. 构造 FullGraph：发生在 DirectSession.Create；
2. 执行简单的 OP 编排算法：发生在 DirectSession.Create；
3. 执行图的剪枝操作：发生在 DirectSession.Run。

当执行 DirectSession::Create 时，将创建 SimpleGraphExecutionState 实例，并完成 FullGraph 实例的构建和初始化。

```
Status SimpleGraphExecutionState::MakeForBaseGraph(
    GraphDef* graph_def, const SimpleGraphExecutionStateOptions& opts,
    std::unique_ptr<SimpleGraphExecutionState>* out_state) {
  auto ret = std::make_unique<SimpleGraphExecutionState>(graph_def, opts));

  AddDefaultAttrsToGraphDef(&ret->original_graph_def_, *ret->flib_def_, 0));
  if (!ret->session_options_->config.graph_options().place_pruned_graph()) {
    ret->InitBaseGraph();
  }
  *out_state = std::move(ret);
  return Status::OK();
}
```

其中，SimpleGraphExecutionState::InitBaseGraph 完成 FullGraph 从 GraphDef 到 Graph 的格式转换，并启动 SimplePlacer 的 OP 编排算法。

```
Status SimpleGraphExecutionState::InitBaseGraph() {
  auto ng = std::make_unique<Graph>(OpRegistry::Global());

  GraphConstructorOptions opts;
  ConvertGraphDefToGraph(opts, *original_graph_def_, ng.get());

  SimplePlacer placer(ng.get(), device_set_, session_options_);
  placer.Run();

  this->graph_ = ng.release();
  return Status::OK();
}
```

图构造：GraphDef -> Graph

刚开始，SimpleGraphExecutionState 得到的是 GraphDef，这是最原始的图结构。它由 Client 将序列化后传递到后端 C++，然后由后端反序列化得到的图结构。

如图??（第??页）所示，通过调用 ConvertGraphDefToGraph 将 GraphDef 实例变换为等价的 Graph 实例；同理，可以调用 Graph.ToGraphDef 将 Graph 实例变换为等价的 GraphDef 实例。

其中，GraphDef 是使用 protobuf 格式存在的图结构，它包含了图所有元数据；而 Graph 则是运行时系统中用于描述图结构的领域对象，它不仅仅持有 GraphDef 的元数据，并包含其它图结构的其它信息。

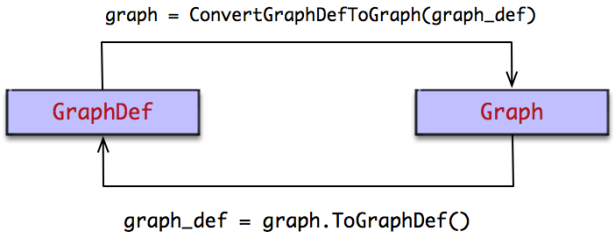


图 12-7 GraphDef 与 Graph 之间的格式转换

OP 编排：SimplePlacer

OP 的编排 (placement) 指的是，将计算图中包含的 OP 以最高效的方式置放在合适的计算设备上运算，以最大化计算资源的利用率，可以形式化地描述为图??（第??页）。

Input: $G = \{n_1, n_2, ..., n_p\}$, $D = \{d_1, d_2, ..., d_q\}$

Output: $\hat{d} = \arg\min_{d \in D} C(n,d)$ for $\forall n \in G$

Cost Model: *C(n,d)*表示节点*n*分配在设备*d*上执行的开销：

1. 如果节点*n*在设备*d*上不存在实现，其开销无穷大；
2. 节点*n*输入输出的大小估计；
3. 节点*n*在设备*d*上执行的时间估计；
4. 跨节点的输入传输至节点*n*的时间估计。

图 12-8 费用模型

求取最优的编排方案，我猜想这是一个 NP 问题。该问题取决于计算图的特征，网络拓扑与带宽，样本数目等多个复杂的因素，该问题也是社区中最活跃的问题之一。

迭代执行

DirectSession.Run 是 TensorFlow 运行时的关键路径，它负责完成一次迭代计算。首先，DirectSession 根据输入/输出对 FullGraph 实施剪枝，生成 ClientGraph；然后，根据所持有本地设备集，将 ClientGraph 分裂为多个 PartitionGraph；运行时为其每个 PartitionGraph 启动一个 Executor 实例，后者执行 PartitionGraph 的拓扑排序算法，完成计算图的执行。

具体实现，请参照**??**节 (**??**)，**??**节 (**??**)，**??**节 (**??**)。

图操作

如图**??**（第**??**页）所示，在本地模式下，计算图经历三个形态的变换，最终被分解至各个计算设备上，以便实现在各个计算设备上并发执行子图。

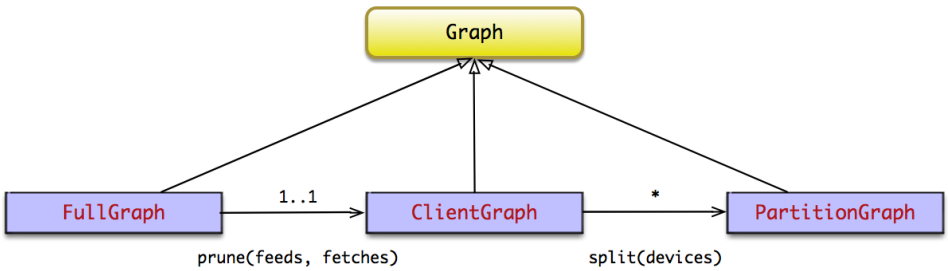


图 12-9 图变换

- FullGraph: Client 负责构造的完整的计算图，常称为 FullGraph；但是，一次 Session.run 并不会执行整个计算图；

- ClientGraph: Master 根据 Session.run 传递 feeds, fetches 输入输出列表，对 FullGraph 实施剪枝操作，计算得到本地迭代执行的最小依赖子图，常称为 ClientGraph；

- PartitionGraph: Master 根据当前计算设备集，及其 OP 的设备约束规范，将 ClientGraph 分裂为多个 PartitionGraph；其中，每个计算设备对应一个 PartitionGraph，计算设备负责 PartitionGraph 的执行。

但是，FullGraph, ClientGraph, PartitionGraph 的数据结构相同，它们都是 Graph 三种不同表现形式，仅仅大小和范畴存在差异。

形式化

在真实的系统实现中，本地模式的运行时是使用 C++ 实现。其中，TensorFlow 运行时的关键路径为 run_step。因为真实系统实现中涉及过多的细节，不易发现算法的主干和逻辑。为了简化问题的描述，将形式化地描述 run_step 的实现过程。

```
def do_run_partitions(executors_and_partitions):
  barrier = ExecutorBarrier(executors_and_partitions.size())
  for (executor, partition) in executors_and_partitions:
    executor.run(partition, barrier)
  barrier.wait()

def run_partitions(executors_and_partitions, inputs, outputs):
  frame = FunctionCallFrame()
  frame.set_args(inputs)
  do_run_partitions(executors_and_partitions)
  frame.get_ret_vals(outputs)

def run_step(devices, full_graph, inputs, outputs):
  client_graph = prune(full_graph, inputs, outputs)
  executors_and_partitions = split(client_graph, devices)
  run_partitions(executors_and_partitions, inputs, outputs)
```

其中，在每个计算设备上，启动一个 Executor 执行分配给它的 PartitionGraph。当某一个计算设备执行完所分配的 PartitionGraph 之后，ExecutorBarrier 的计数器加 1，直至所有设备完成 PartitionGraph 列表的执行，barrier.wait() 阻塞操作退出。

跨设备的 PartitionGraph 之间可能存在数据依赖关系，它们之间通过插入 Send/Recv 节点完成交互。事实上，在本地模式中，Send/Recv 通过 Rendezvous 完成数据交换的。Send 将数据放在 Rendezvous 上，而 Recv 则根据标识从 Rendezvous 取走。其中，Send 不阻塞，而 Recv 是阻塞的。

关闭会话

```
Status DirectSession::Close() {
  cancellation_manager_->StartCancel();
  {
    mutex_lock l(closed_lock_);
    if (closed_) return Status::OK();
    closed_ = true;
  }
  return Status::OK();
}
```

如图??（第??页）所示，将 Step 注册给 DirectSession 的 CancellationManager 之中。当 DirectSession 被关闭时，DirectSession 的 CancellationManager，将取消这次 step 的执行过程。

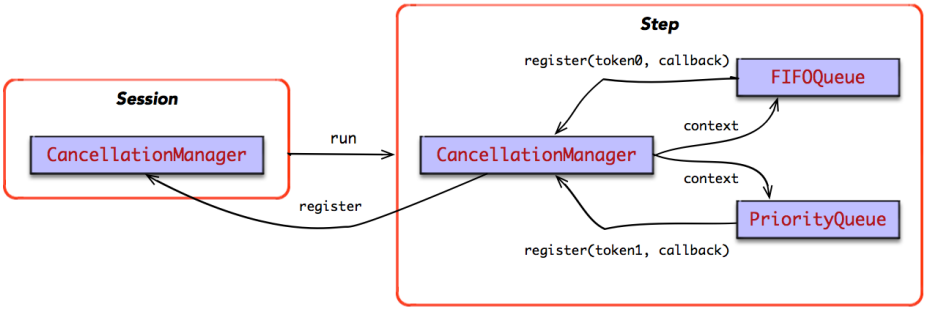


图 12-10 CancellationManager 工作原理

```
Status DirectSession::Run(
   const NamedTensorList& inputs,
   const std::vector<string>& output_names,
   const std::vector<string>& target_nodes,
   std::vector<Tensor>* outputs) {
  // step_cancellation_manager is passed to `OpKernelContext`
  CancellationManager step_cancellation_manager;

  // Register this step with session's cancellation manager, so that
  // `Session::Close()` will cancel the step.
  CancellationToken cancellation_token =
      cancellation_manager_->get_cancellation_token();
  bool already_cancelled = !cancellation_manager_->RegisterCallback(
      cancellation_token, [&step_cancellation_manager]() {
        step_cancellation_manager.StartCancel();
      });
  // ignore others...
}
```

当前 Step 的 CancellationManager 最终会传递给 OpKernelContext。Kernel 实现计算时，如果保存了中间状态，可以向其注册相应的回调钩子。其中，每个回调钩子都有唯一的 token 标识。

当 Step 被取消时，回调钩子被调用，该 Kernel 可以取消该 OP 的计算。例如，FIFOQueue 实现 TryEnqueue 时，便往本次 Step 的 CancellationManager 注册了回调钩子，用于取消该 Kernel 中间的状态信息。

```
void FIFOQueue::TryEnqueue(const Tuple& tuple, OpKernelContext* ctx,
                           DoneCallback callback) {
  CancellationManager* cm = ctx->cancellation_manager();
  CancellationToken token = cm->get_cancellation_token();
  bool already_cancelled;
  {
    mutex_lock l(mu_);
    already_cancelled = !cm->RegisterCallback(
        token, [this, cm, token]() { Cancel(kEnqueue, cm, token); });
  }
  // ignore others...
}
```

12.3 剪枝

DirectSession::Run 执行时，首先完成 ClientGraph 的构造。事实上，ClientGraph 的构造过程，主要完成 FullGraph 的剪枝算法，并生成 ClientGraph。

构建 ClientGraph

如图??（第??页）所示，SimpleGraphExecutionState 实例持有 FullGraph 实例，并根据输入/输出列表，生成 ClientGraph。

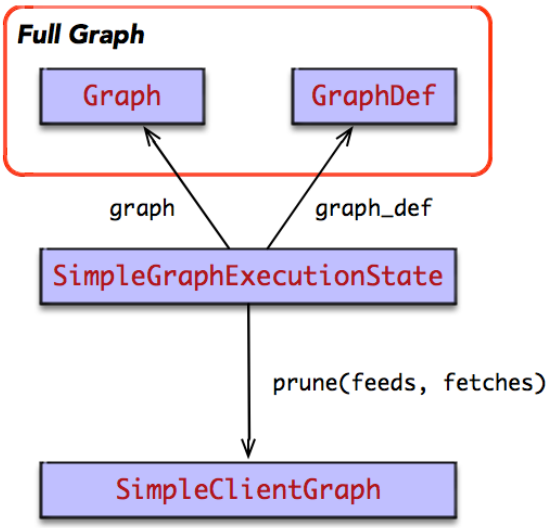


图 12-11 生成 ClientGraph

其中，BuildGraphOptions 包含了输入/输出列表，调用 SimpleGraphExecutionState::BuildGraph 生成 ClientGraph 实例。

```
namespace {
  BuildGraphOptions build_graph_options(
    const NamedTensorList& inputs,
    const std::vector<string>& outputs,
    const std::vector<string>& targets) {
    // sort inputs/outputs/targets
    std::vector<string> inputs_sorted(inputs.begin(), inputs.end());
    std::sort(inputs_sorted.begin(), inputs_sorted.end());

    std::vector<string> outputs_sorted(outputs.begin(), outputs.end());
    std::sort(outputs_sorted.begin(), outputs_sorted.end());

    std::vector<string> tn_sorted(targets.begin(), targets.end());
    std::sort(tn_sorted.begin(), tn_sorted.end());

    // build graph options
    BuildGraphOptions options;
    options.feed_endpoints = inputs_sorted;
    options.fetch_endpoints = outputs_sorted;
    options.target_nodes = tn_sorted;
    options.use_function_convention = !run_state_args->is_partial_run;
    return options;
  }
}

Status DirectSession::Run(
  const RunOptions& run_options,
  const NamedTensorList& inputs,
  const std::vector<string>& output_names,
  const std::vector<string>& target_nodes,
  std::vector<Tensor>* outputs,
  RunMetadata* run_metadata) {

  // 1. prune graph
  // client_graph = prune(full_graph, inputs, outputs)
  std::unique_ptr<SimpleClientGraph> client_graph;
  execution_state_->BuildGraph(
    build_graph_options(inputs, output_names, target_nodes),
    &client_graph);

  // 2. split graph into partition by devices
  // executors_and_partitions = split(client_graph, devices)

  // 3. lauch executor per partition
  // def run_partitions(executors_and_partitions, inputs, outputs):
  //   frame = FunctionCallFrame()
  //   frame.set_args(inputs)
  //   for (executor, partition) in executors_and_partitions:
  //     exec.run(part)
  //   frame.get_ret_vals(outputs)

  return Status::OK();
}
```

ClientGraph 初始来自原始的 FullGraph，调用 RewriteGraphForExecution 函数，将根据输入/输出，对 ClientGraph 实施改写操作，包括增加节点，或删除节点，最终生成 SimpleClientGraph 实例。

```
const DeviceAttributes&
SimpleGraphExecutionState::local_device_attr() const {
  return device_set_->client_device()->attributes();
}

Status SimpleGraphExecutionState::BuildGraph(
  const BuildGraphOptions& options,
```

```
    std::unique_ptr<SimpleClientGraph>* out) {
    // 1. create new_graph from origin graph,
    // which is client graph.
    std::unique_ptr<Graph> ng;
    ng.reset(new Graph(flib_def_.get()));
    CopyGraph(*graph_, ng.get());

    // 2. prune the client graph
    subgraph::RewriteGraphForExecution(
      ng.get(), options.feed_endpoints, options.fetch_endpoints,
      options.target_nodes, local_device_attr(),
      options.use_function_convention);
    }

    // 3. create SimpleClientGraph, and return it.
    std::unique_ptr<SimpleClientGraph> dense_copy(
        new SimpleClientGraph(std::move(flib)));
    CopyGraph(*ng, &dense_copy->graph);
    *out = std::move(dense_copy);

    return Status::OK();
  }
```

因此，构建 ClientGraph 过程，其关键路径为 RewriteGraphForExecution，即剪枝算法。剪枝算法根据输入/输出列表，反向遍历 FullGraph，找到最小的依赖子图 ClientGraph。

一般地，对于 ClientGraph 输入节点，扮演了起始节点；而输出节点，扮演了终止节点。因此，关于输入和输出，存在两个比较棘手的问题：

1. 输入：当 ClientGraph 计算开始前，外部的运行时如何传递 Tensor 给输入节点；
2. 输出：当 ClientGraph 计算完成后，外部的运行时又如何从输出节点获取 Tensor。

存在两种媒介：FunctionCallFrame 和 Rendezvous，外部运行时与输入/输出节点可以使用其中一种媒介交换数据。

FunctionCallFrame 用于 Arg/RetVal 函数调用的 OP，用于函数调用时传递函数参数值，及其返回函数值。但是，它们仅适用于单进程的运行时环境。

Rendezvous 用于 Send/Recv 消息发送的 OP，这是一种更为通用的通信方式，适用于分布式的运行时环境。

基于 Rendezvous

如图**??**（第**??**页）所示，根据 fetches 列表，反向搜索依赖的节点，直至 feeds，计算得到最小依赖的子图。

对于 Feed 的边实施剪枝，例如剪枝 ina:0) 边，并在此处插入节点 Recv，并按照输入边的名字命名该节点，例如 _recv_ina_0。

同理，对于 Fetch 的边也实施剪枝，例如剪枝 f:0 边，并在此处插入节点 Send 节点，并按照输出边的名字命名该节点，例如 _send_f_0。

最终，通过插入 Source/Sink 节点，将剪枝后得到各个联通的子图进行汇总，形成一个完整的 DAG 图。

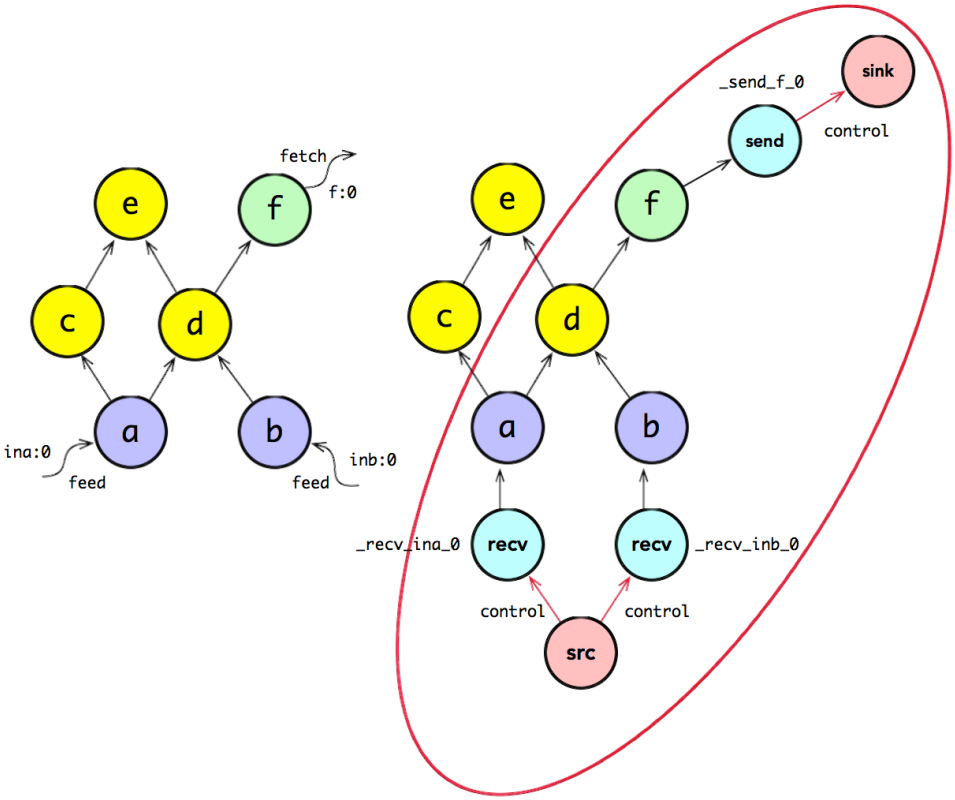


图 12-12 图剪枝：插入 Send/Recv 节点

基于 FunctionCallFrame

但是，输入/输出通过 Rendezvous 交换数据可能存在性能上的瓶颈。因为待发送的 Tensor 需要携带发送设备，接收设备，TensorId，共同组成了唯一的字符串标识，数据发送和接收需要花费很长的字符串解析的时间开销。

特殊地，对于本地模式，因为在同一进程内，使用 Rendezvous 交换数据存在不必要的性能损耗。可以使用基于 FunctionCallFrame 函数调用替代之。

因此，在本地模式下，可以使用 Arg/RetVal 分别替代 Send/Recv 节点，从而实现了函数调用交换数据的方式，替代原有基于 Rendezvous 交互数据的方式。

如图??（第??页）所示。对于 Feed 的边实施剪枝，例如剪枝 ina:0) 边，并在此处插入节点 Arg，并按照输入边的名字命名该节点，例如 _arg_ina_0。

同理，对于 Fetch 的边也实施剪枝，例如剪枝 f:0 边，并在此处插入节点 RetVal 节点，并按照输出边的名字命名该节点，例如 _retval_f_0。

最终，通过插入 Source/Sink 节点，将剪枝后得到各个联通的子图进行汇总，形成一个完整的 DAG 图。

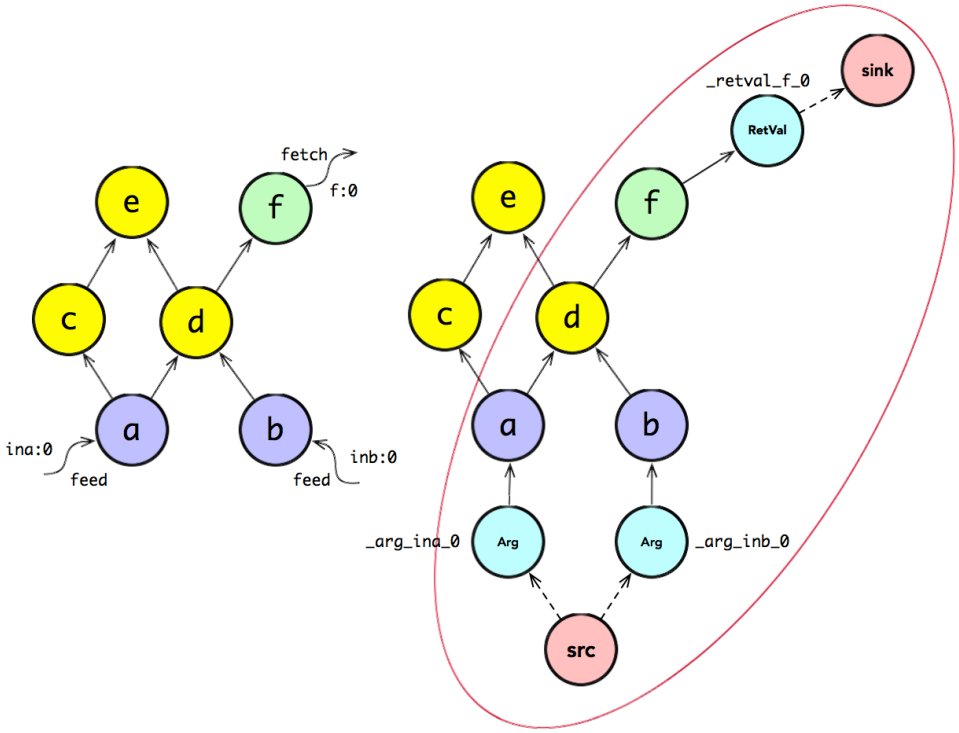


图 12-13 图剪枝：插入 Arg/RetVal 节点

剪枝算法实现

剪枝算法主要由 RewriteGraphForExecution 完成，主要包括 3 个子过程。

1. 追加输入节点
2. 追加输出节点
3. 反向剪枝

```
void RewriteGraphForExecution(Graph* g, bool use_function,
    const ArraySlice<string>& fed_outputs,
    const ArraySlice<string>& fetch_outputs,
    const ArraySlice<string>& target_node_names,
    const DeviceAttributes& device_info) {
  FeedInputs(g, use_function, device_info, fed_outputs);

  std::vector<Node*> fetch_nodes;
  FetchOutputs(g, use_function, device_info,
    fetch_outputs, &fetch_nodes);

  PruneForTargets(g, fetch_nodes, target_node_names);
}
```

追加输入节点

如图??（第??页）所示，对于任意一条输入边实施剪枝时，插入相应的 Arg 或 Recv 节点，删除既有的边，并重新连接相应的边。

在计算图中，一条边唯一地由 TensorId 标识，它由 op:src_output 二元组构成。前者表示边的上游节点，后者表示给边为上游节点的第几条边。

示例代码删除了部分不重要的逻辑，并搬迁了部分函数的职责，并在局部尝试部分函数提取，以便更好地还原算法的逻辑。其中，假设 Graph 可以按照 TensorId 索引节点和边。

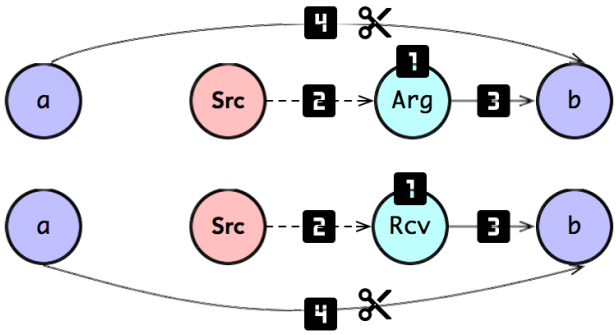


图 12-14 剪枝：输入边

```
namespace {
  DataType data_type(Graph& g, const TensorId& tensor_id) {
    Node* upstream_node = g.upstream_node(tensor_id);
    return BaseType(upstream_node->output_type(tensor_id.src_output()));
  }

  Node* AppendRecvNode(Graph& g,
    const TensorId& tensor_id, const DeviceAttributes& device_info) {
      Node* recv_node;
      NodeBuilder(strings::StrCat(
        "_recv_", tensor_id.op(), "_", tensor_id.src_output()), "_Recv")
        .Attr("tensor_type", data_type(g, tensor_id))
        .Attr("tensor_name", tensor_id.name())
        .Attr("send_device", device_info.name())
        .Attr("recv_device", device_info.name())
        .Attr("send_device_incarnation", device_info.incarnation())
        .Attr("client_terminated", true)
        .Finalize(g, &recv_node);
      return recv_node;
  }

  Node* AppendArgNode(Graph& g, size_t index,
    const TensorId& tensor_id, const DeviceAttributes& device_info) {
    Node* arg_node;
    NodeBuilder(strings::StrCat(
      "_arg_", tensor_id.op(), "_", tensor_id.src_output()), "_Arg")
      .Attr("T", data_type(g, tensor_id))
      .Attr("index", index)
      .Finalize(g, &arg_node);
    return arg_node;
  }

  // 1. append arg/recv node
  Node* AppendNewNode(Graph& g, bool use_function, size_t index,
```

```
      const TensorId& tensor_id, const DeviceAttributes& device_info) {
      if (use_function) {
        return AppendArgNode(g, index, tensor_id, device_info);
      } else {
        return AppendRecvNode(g, tensor_id, device_info);
      }
    }

    void AppendNewEdges(Graph& g,
      Node* new_node, const TensorId& tensor_id) {
      // 2. add control edge between source node and new node.
      g.AddControlEdge(g.source_node(), new_node);

      Edge* old_edge = g.edge(tensor_id);

      // 3. add edge between new node and downstream node.
      g.AddEdge(new_node, 0, old_edge->dst(), old_edge->dst_input());

      // 4. remove old edge.
      g.RemoveEdge(old_edge);
    }
  }

  void FeedInputs(Graph& g, bool use_function,
    const DeviceAttributes& device_info,
    const ArraySlice<TensorId>& feeds) {
    for (size_t i = 0; i < feeds.size(); ++i) {
      Node* new_node = AppendNewNode(use_function, i, feeds[i]);
      AppendNewEdges(g, new_node, feeds[i]);
    }
  }
```

追加输出节点

对于任意一条输出边实施剪枝时，插入相应的 `RetVal` 或 `Send` 节点，并将其与 `Sink` 节点通过控制依赖边连接。

如图**??**（第**??**页）所示，对输出边实施剪枝操作。新节点与上游节点的连接关系，在构造新节点时，通过 `Input` 已经指定。另外，函数直接返回了新节点 (`RetVal`/`Send`) 为终止节点，因此没必要删除原来的边，其算法与输入边的处理存在微妙的差异。

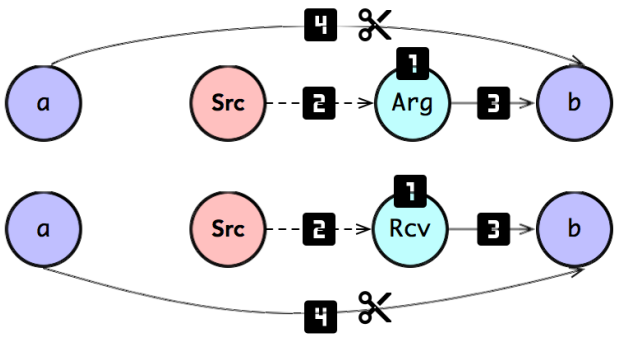


图 12-15 剪枝：输出边

```
namespace {
  Node* AppendSendNode(Graph& g,
    const TensorId& tensor_id, const DeviceAttributes& device_info) {
    Node* send_node;
    NodeBuilder(strings::StrCat(
      "_send_", tensor_id.op(), "_", id.src_output()), "_Send")
      // 2. add edge between upstream node and send node.
      .Input(g.upstream_node(tensor_id), tensor_id.src_output())
      .Attr("tensor_name", tensor_id.name())
      .Attr("send_device", device_info.name())
      .Attr("recv_device", device_info.name())
      .Attr("send_device_incarnation",
            device_info.incarnation())
      .Attr("client_terminated", true)
      .Finalize(g, &send_node);
    return send_node;
  }

  Node* AppendRetvalNode(Graph& g, size_t index,
    const TensorId& tensor_id, const DeviceAttributes& device_info) {
    Node* retval_node;
    NodeBuilder(strings::StrCat(
      "_retval_", tensor_id.op(), "_", tensor_id.src_output(), "_", index),
      "_Retval")
      // 2. add edge between upstream node and retval node.
      .Input(g.upstream_node(tensor_id), tensor_id.src_output())
      .Attr("T", data_type(g, tensor_id))
      .Attr("index", index)
      .Finalize(g, &retval_node))
    return retval_node;
  }

  // 1. append retval/send node
  Node* AppendNewNode(Graph& g, bool use_function, size_t index,
    const TensorId& tensor_id, const DeviceAttributes& device_info) {
    if (use_function) {
      return AppendRetvalNode(g, index, tensor_id, device_info);
    } else {
      return AppendSendNode(g, tensor_id, device_info);
    }
  }
}

void FetchOutputs(Graph& g, bool use_function,
  const DeviceAttributes& device_info,
  const ArraySlice<TensorId>& fetches,
  std::vector<Node*>& fetch_nodes) {
  for (size_t i = 0; i < fetches.size(); ++i) {
    Node* new_node = AppendNewNode(use_function, i, fetches[i]);

    // 3. add control edge between new node and sink node.
    g->AddControlEdge(new_node, g->sink_node());

    fetch_nodes.push_back(new_node);
  }
}
```

反向剪枝

剪枝操作，其本质就是 DAG 反向的宽度优先遍历算法。首先，创建了一个队列，及其一个 `visited` 数组，后者用于记录已经遍历过的节点。初始化时，队列仅包含输出节点和输入节点 (`targets`)。当图遍历完毕后，不再 `visited` 里面的节点，表示本此执行不依赖于

它，应从图中删除该节点，及其相关联的边。

经过剪枝后，将形成若干 DAG 子图。将入度为 0 的节点，与 Source 节点通过控制依赖边相连接；出度为 0 的节点，与 Sink 节点通过控制依赖边相连接，最终形成一个完整的 DAG 图。

```
namespace {
  void ReverseBFS(
    Graph* g, std::unordered_set<const Node*>& visited) {
    std::deque<const Node*> queue(visited.begin(), visited.end());
    while (!queue.empty()) {
      const Node* n = queue.front();
      queue.pop_front();
      for (const Node* in : n->in_nodes()) {
        if (visited.insert(in).second) {
          queue.push_back(in);
        }
      }
    }
  }

  void RemoveUnvisitedNodes(
    Graph* g, std::unordered_set<const Node*>& visited) {
    for (Node* n : g->nodes()) {
      if (visited.count(n) == 0 && !n->IsSource() && !n->IsSink()) {
        g->RemoveNode(n);
      }
    }
  }

  void PruneForReverseReachability(
    Graph* g, std::unordered_set<const Node*>& visited) {
    ReverseBFS(g, visited);
    RemoveUnvisitedNodes(g, visited);
  }

  void FixupSourceEdges(Graph* g, Node* n) {
    if (!n->IsSource() && n->in_edges().empty()) {
      g->AddControlEdge(g->source_node(), n);
    }
  }

  void FixupSinkEdges(Graph* g, Node* n) {
    if (!n->IsSink() && n->out_edges().empty()) {
      g->AddControlEdge(n, g->sink_node());
    }
  }

  void FixupSourceAndSinkEdges(Graph* g) {
    for (Node* n : g->nodes()) {
      FixupSourceEdges(g, n);
      FixupSinkEdges(g, n);
    }
  }

  void AppendTargetNodes(Graph& g,
    const ArraySlice<string>& target_names,
    std::unordered_set<const Node*>& targets) {
    for (auto name : target_names) {
      Node* target = g.GetNodeBy(name);
      targets.insert(target);
    }
  }
}

void PruneForTargets(Graph* g,
  std::vector<Node*>& fetch_nodes,
  const ArraySlice<string>& target_names) {
```

```
    std::unordered_set<const Node*> targets(
      begin(fetch_nodes), end(fetch_nodes));

    AppendTargetNodes(g, target_names, targets);
    PruneForReverseReachability(g, targets);
    FixupSourceAndSinkEdges(g);
  }
```

12.4 分裂

如图??（第??页）所示，假如 d 节点放置在 GPU0 上执行，而其他节点放置在 CPU0 上执行。其中，节点 a 与 b 通过 Arg 输入数据；节点 f 将其结果输出到 RetVal 节点上。

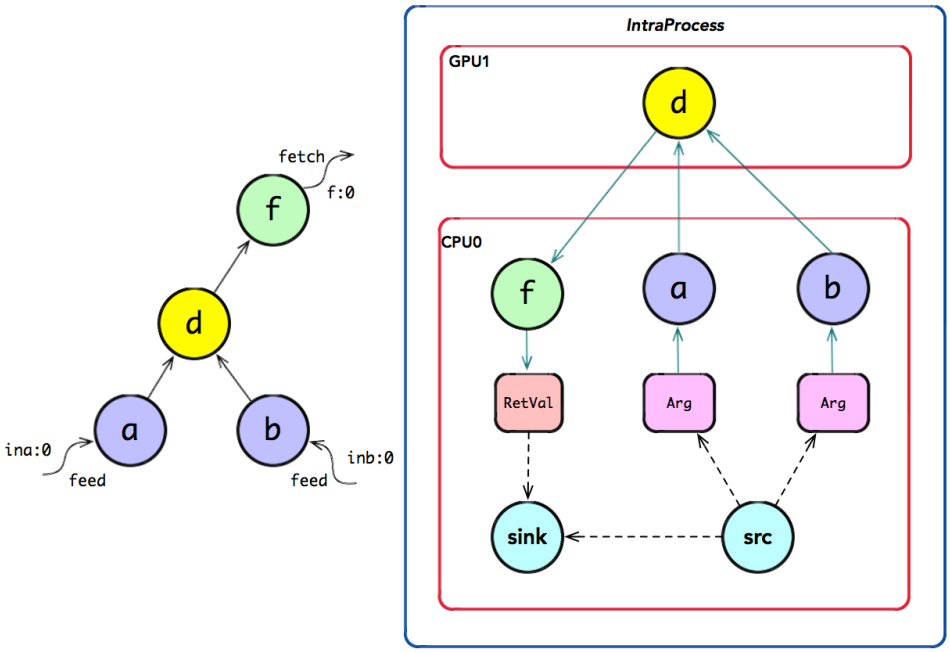


图 12-16 按本地设备集执行图分裂

因此，计算图中存在若干条边跨越设备。对于跨越设备的边，运行时将其分裂，并就地插入 Send/Recv 边，分别用于原设备上发送数据，并在目标设备上接受数据，完成设备间的数据交换。如图??（第??页）所示。

其中，Arg/RetVal 节点通过媒介 FunctionCallFrame 交换数据；Send/Recv 节点通过媒介 Rendezvous 交换数据。

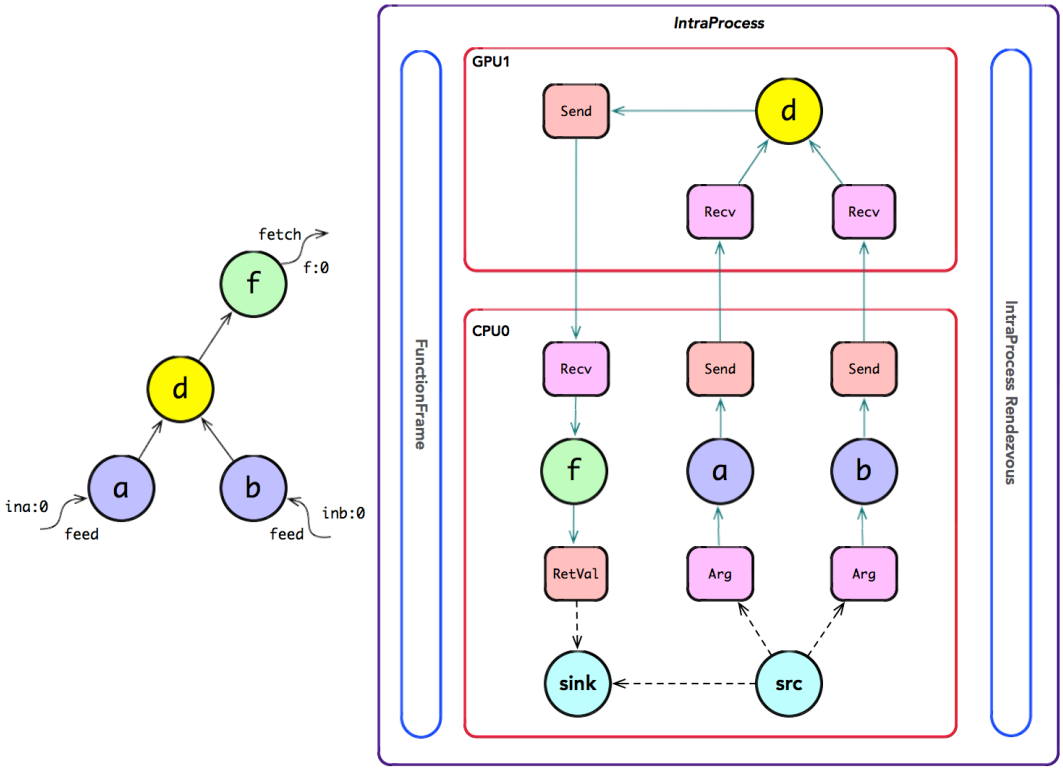


图 12-17 跨设备 OP 之间插入 Send/Recv 节点

情况 1

最简单的情况下，src 与 dst 在同一个 Partition 内。因此，直接将其划归在同一个 Partition 即可。

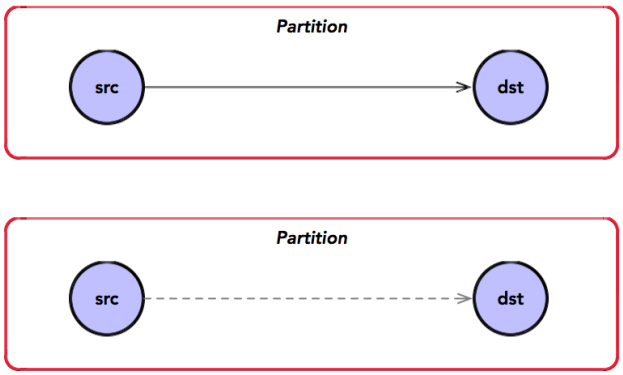


图 12-18 情况 1：src 与 dst 在同一个 Partition 内

情况 2

如果 src 与 dst 不在同一个 Partition 内，但两者之间原来是通过普通边连接在一起的。因此，仅需要在它们中间增加 Send 与 Recv 节点，将其划归在两个不同的 Partition

内。

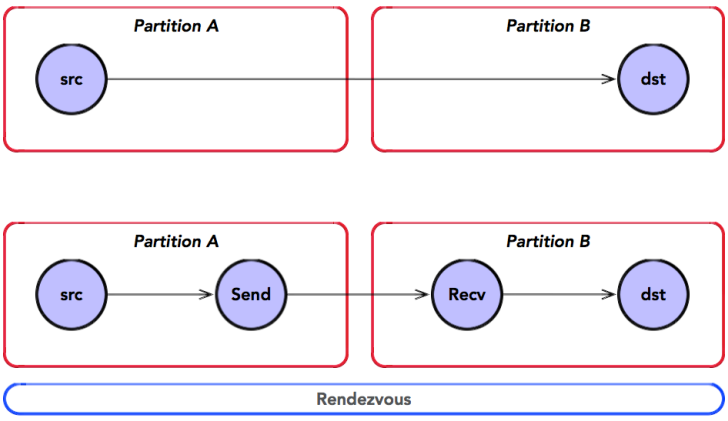


图 12-19 情况 2：src 与 dst 不在同一个 Partition 内，但两者之间是普通边

情况 3

如果 src 与 dst 不在同一个 Partition 内，但两者之间原来是通过控制依赖边连接在一起的。

此时，需要在 src 侧增加一个 Const 的 DummyNode，并作为 src 的下游通过控制依赖边相连；最终，在通过 Send 将其值发送到对端。

在 dst 侧，Recv 收到该值，使用 Identity 将其消费掉；最终，再将 Identity 与 dst 连接控制依赖边。

在这里，Const 扮演生产者，Identity 扮演消费者角色。既满足了跨设备间通信的需求，又满足原来 src 与 dst 之间的控制依赖的关系。但是，其缺点就是存在微妙的性能开销。

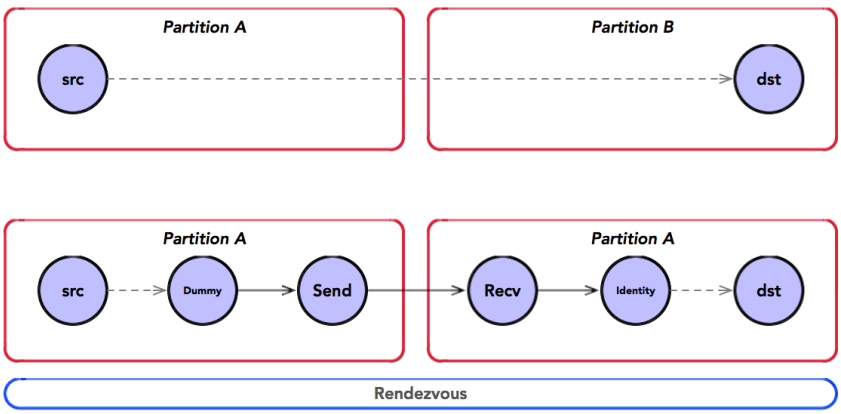


图 12-20 情况 3：src 与 dst 不在同一个 Partition 内，但两者之间是控制依赖边

分裂算法实现

分裂算法也是一个反向遍历图的算法。对于当前遍历的节点，将其标记为 `dst`；然后再寻找 `dst` 的所有输入边；遍历所有输入边，从而找到与改边相连的源节点，将其标记为 `src`。

因此，更具上述讨论的三种情况，迭代实现 `src` 与 `dst` 两者之前的 `Partition` 划分算法。

```
namespace {

  using Edges = std::vector<const Edge*>;
  using Partitions = std::unordered_map<string, GraphDef>;

  void AddInput(NodeDef* dst, StringPiece src_name, int src_slot) {
    if (src_slot == Graph::kControlSlot) {
      dst->add_input(strings::StrCat("^", src_name));
    } else if (src_slot == 0) {
      dst->add_input(src_name.data(), src_name.size());
    } else {
      dst->add_input(strings::StrCat(src_name, ":", src_slot));
    }
  }

  Edges InputsOf(const Node* dst) {
    Edges inputs(dst->num_inputs(), nullptr);
    for (auto edge : dst.in_edges()) {
      if (edge->IsControlEdge()) {
        inputs.push_back(e);
      } else {
        inputs[edge->dst_input()] = edge;
      }
    }
    return inputs;
  }

  NodeDef* InitDstNodeDef(const Node& dst, NodeDef* dst_def) {
    dst_def = dst.def();
    dst_def->set_device(dst.assigned_device_name());
    dst_def->clear_input();
    return dst_def;
  }

  NodeDef* AddDummyConst(const PartitionOptions& opts, GraphDef* gdef,
                         const Edge* edge, Status* status) {
    const Node* src = edge->src();
    Tensor tensor(DT_FLOAT, TensorShape({0}));
    NodeDef* result = gdef->add_node();
    *status = NodeDefBuilder(opts.new_name(src->name()), "Const")
                  .Device(src->assigned_device_name())
                  .Attr("dtype", DT_FLOAT)
                  .Attr("value", tensor)
                  .Finalize(result);
    return result;
  }

  NodeDefBuilder::NodeOut BuildSendFrom(
      const PartitionOptions& opts,
      GraphDef* src_graph,
      const Edge* edge,
      NodeDefBuilder::NodeOut& send_from) {
    if (edge->IsControlEdge()) {
      // Case 3: DummyNode(Const) -ctrl-> src -> send
      NodeDef* dummy = AddDummyConst(opts, src_graph, edge);
      AddInput(dummy, edge->src()->name(), Graph::kControlSlot);
```

```
      send_from.Reset(dummy->name(), 0, DT_FLOAT);
    } else {
      // Case 2: src -> send
      send_from.Reset(edge->src()->name(),
                      edge->src_output(),
                      EdgeType(edge));
    }
  }

  void SetSendRecvAttrs(
      const PartitionOptions& opts,
      const Edge* edge,
      NodeDefBuilder* builder) {
    builder->Attr("tensor_name",
                  strings::StrCat("edge_", edge->id(), "_", \
                    edge->src()->name()));
    builder->Attr("send_device", edge->src()->assigned_device_name());
    builder->Attr("send_device_incarnation",
                  static_cast<int64>(
                      opts.get_incarnation(edge->src()->assigned_device_name())));
    builder->Attr("recv_device", edge->dst()->assigned_device_name());
    builder->Attr("client_terminated", false);
  }

  NodeDef* AddSend(
      const PartitionOptions& opts,
      GraphDef* gdef,
      const Edge* edge,
      NodeDefBuilder::NodeOut send_from) {
    NodeDef* send = gdef->add_node();
    NodeDefBuilder builder(opts.new_name(edge->src()->name()), "_Send");
    SetSendRecvAttrs(opts, edge, &builder);
    builder.Device(edge->src()->assigned_device_name())
           .Input(send_from)
           .Finalize(send);
    return send;
  }

  NodeDef* AddRecv(const PartitionOptions& opts, const GraphInfo& g_info,
                   GraphDef* gdef, const Edge* edge, NodeDef** real_recv,
                   Status* status) {
    NodeDef* recv = gdef->add_node();
    NodeDefBuilder builder(opts.new_name(src->name()), "_Recv");
    SetSendRecvAttrs(opts, edge, &builder);
    builder.Device(dst->assigned_device_name())
           .Attr("tensor_type", EdgeType(edge))
           .Finalize(recv);
    return recv;

    if (edge->IsControlEdge()) {
      // Case 3: Recv -> Identity -contrl-> dst
      NodeDef* id = gdef->add_node();
      NodeDefBuilder(opts.new_name(src->name()), "Identity")
          .Device(dst->assigned_device_name())
          .Input(recv->name(), 0, cast_dtype)
          .Finalize(id);
      return id;
    } else {
      return recv;
    }
  }

  void InsertSendRecv(
      const PartitionOptions& opts,
      GraphDef* src_graph,
      Edge* edge,
      GraphDef* dst_graph,
      NodeDef* dst_def) {
    NodeDefBuilder::NodeOut send_from;
    BuildSendFrom(opts, src_graph, edge, send_from);

    NodeDef* send = AddSend(opts, src_graph, edge, send_from);
```

```
      NodeDef* recv = AddRecv(opts, dst_graph, edge);

      if (edge->IsControlEdge()) {
        // Case 3: In fact, recv is identity.
        AddInput(dst_def, recv->name(), Graph::kControlSlot);
      } else {
        AddInput(dst_def, recv->name(), 0);
      }
    }
  }

  Status Partition(const PartitionOptions& opts,
                   Partitions& partitions, Graph& client_graph) {
    for (const Node* dst : client_graph.op_nodes()) {
      // 1. find dst node
      GraphDef* dst_graph = &partitions[opts.node_to_loc(dst)];
      NodeDef* dst_def = InitDstNodeDef(*dst, dst_graph->add_node());

      // 2. search all input edges.
      for (const Edge* edge : InputsOf(dst)) {
        // 3. find src node: edge->src()
        GraphDef* src_graph = &partitions[opts.node_to_loc(src)];

        // skip sink/source nodes.
        if (!edge->src()->IsOp())
          continue;

        // Case 1: same partition
        if (src_graph == dst_graph) {
          AddInput(dst_def, src->name(), edge->src_output());
          continue;
        }

        // Case 2-3: different partition
        InsertSendRecv(opts, src_graph, edge, dst_graph, dst_def);
      }
    }
  }
```

回调函数

在 PartitionOptions 中，存在两个重要的回调函数。NodeToLocFunc 用于图分裂；NewNameFunc 用于给新增加的节点命名，例如 Send/Recv。

```
  struct PartitionOptions {
    typedef std::function<string(const Node*)> NodeToLocFunc;
    NodeToLocFunc node_to_loc = nullptr;

    typedef std::function<string(const string&)> NewNameFunc;
    NewNameFunc new_name = nullptr;

    // ignore others...
  };
```

对于图分裂，存在两种最基本的分裂方法。

```
  string SplitByDevice(const Node* node) {
    return node->assigned_device_name();
  }
```

```
string SplitByWorker(const Node* node) {
  string task, device;
  DeviceNameUtils::SplitDeviceName(
      node->assigned_device_name(), &task, &device);
  return task;
}
```

在本地模式下，NodeToLocFunc 被配置为 SplitByDevice。如图 intraprocess-splity-by-device 所示。

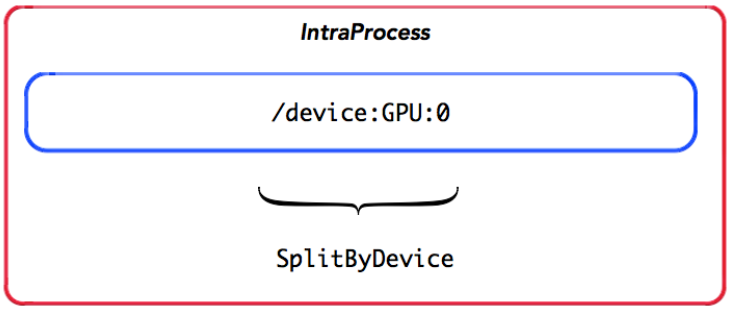


图 12-21 本地模式：SplitByDevice

在分布式模式下，Master 的 NodeToLocFunc 被配置为 SplitByWorker；而 Worker 的 NodeToLocFunc 被配置为 SplitByDevice。

因此，在分布式模式下，图分裂经历了两级分离。第一级按照 SplitByWorker 分裂，将图划分到各个 Worker 上去；第二级按照 SplitByDevice，再将图划分到各个计算设备上去。

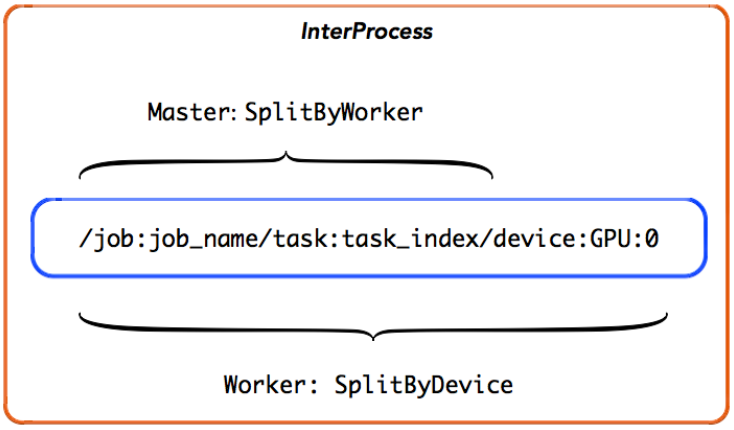


图 12-22 分布式模式：两级分裂

12.5 执行

接下来，运行时将并发执行各个 PartitionGraph。如图**??**（第**??**页）所示，每个 PartitionGraph 启动一个 Executor，实现并发执行图的计算。

每个 Executor 将执行 PartitionGraph 的拓扑排序算法，将入度为 0 的 OP 追加到 ready_queue 之中，并将其关联的 OP 的入度减 1。调度器调度 ready_queue 之中 OP ，并将其放入 ThreadPool 中执行对应的 Kernel 实现。

在所有 Partition 开始并发执行之前，需要外部将其输入传递给相应的 Arg 节点；当所有 Partition 完成计算后，外部再从 RetVal 节点中取走数据。其中，Arg/RetVal 节点之间的数据时通过 FunctionCallFrame 完成交互的。

如果 PartitionGraph 之间需要跨设备交换数据，生产者将其放在 Send 节点，消费者通过 Recv 节点获取数据。其中，发送方不阻塞；接收方如果数据未到，则发生阻塞直至超时。此外，Send/Recv 节点之间的数据是通过 Rendezvous 完成交互的。

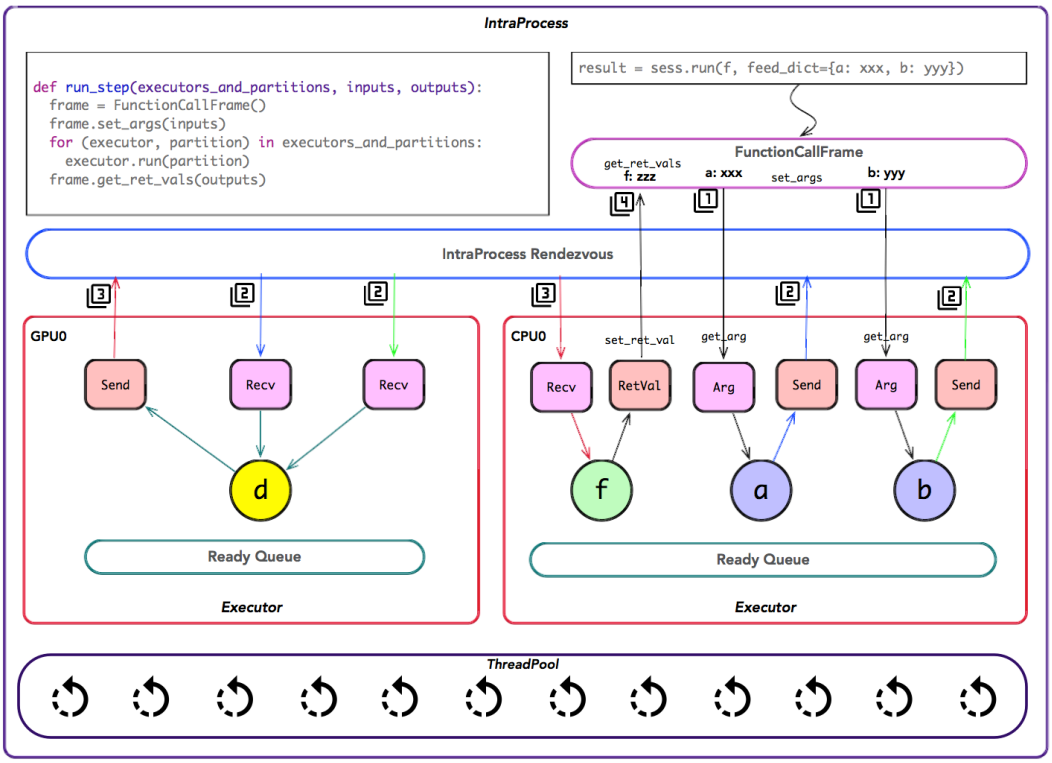


图 12-23 执行图

因此，执行图计算需要解决如下 3 个核心问题：

1. 输入/输出处理
2. 设备间数据交换
3. 执行 PartitionGraph

输入

在某个设备上，PartitionGraph 的起始节点为 Arg 节点，结束节点为 RetVal 节点。整个过程可以看成函数调用过程，Arg 用于传递函数参数，RetVal 用于返回函数值。

更确切地说，Arg 完成 PartitionGraph 的输入，RetVal 完成 PartitionGraph 的输出。对于 Arg 节点，其调用时序为：set_arg -> get_arg。其中，前者由 DirectSession 在启动 Executor 列表之前，通过调用 FunctionCallFrame.SetArgs(feeds)，传递输入参数列表的值；后者由 Arg 的 Kernel 实现调用。

```
Status DirectSession::Run(
  const RunOptions& run_options,
  const NamedTensorList& inputs,
  const std::vector<string>& output_names,
  const std::vector<string>& target_nodes,
  std::vector<Tensor>* outputs,
  RunMetadata* run_metadata) {

  // 1. prune graph
  // client_graph = prune(full_graph, inputs, outputs)

  // 2. split graph into partition by devices
  // executors_and_partitions = split(client_graph, devices)
  ExecutorsAndKeys* executors_and_keys = ... // ignore implements...

  // 3. lauch executor per partition
  // def run_partitions(executors_and_partitions, inputs, outputs):
  //   frame = FunctionCallFrame()
  //   frame.set_args(inputs)
  //   for (executor, partition) in executors_and_partitions:
  //     exec.run(part)
  //   frame.get_ret_vals(outputs)

  // 3.1 construct FunctionCallFrame
  FunctionCallFrame call_frame(
    executors_and_keys->input_types,
    executors_and_keys->output_types);

  // 3.2 frame.set_args(inputs)
  // 3.2.1 construct feeds list
  gtl::InlinedVector<Tensor, 4> feed_args(inputs.size());
  for (const auto& it : inputs) {
    // (first, second) => (tensor_name, tensor)
    feed_args[executors_and_keys->input_name_to_index[it.first]] = it.second;
  }

  // 3.2.2 frame.set_args(feeds)
  call_frame.SetArgs(feed_args);

  // 3.3 concurent execution
  // for (executor, partition) in executors_and_partitions:
  //   executor.run(partition)

  // 3.4 fetch outputs.
}
```

而 frame.get_arg 则有 Arg 来获取，并且 Arg 将其输出到 PartitionGraph 中的第一个计算节点。

```
struct ArgOp : OpKernel {
  explicit ArgOp(OpKernelConstruction* ctx) : OpKernel(ctx) {
    ctx->GetAttr("T", &dtype_);
    ctx->GetAttr("index", &index_);
  }

  void Compute(OpKernelContext* ctx) override {
    auto frame = ctx->call_frame();

    Tensor val;
    frame->GetArg(index_, &val);

    // put it into downsteram op's input.
    ctx->set_output(0, val);
  }

 private:
  int index_;
  DataType dtype_;
};
```

并发执行

经过图分裂后，运行为每个 Partition 启动一个 Executor。为了监听所有 Executor 是否全部完成，创建了一个 ExecutorBarrier。并且在启动所有 Executor 之后，调用 executors_done.Wait() 阻塞，等待所有 Executor 完成执行。

当完成一个 Executor 中完成，ExecutorBarrier 的计算器减 1(初始值为 num_executors)，直至为 0，将调用其完成钩子，最终触发 executors_done.Notify()。

```
Status DirectSession::Run(
  const RunOptions& run_options,
  const NamedTensorList& inputs,
  const std::vector<string>& output_names,
  const std::vector<string>& target_nodes,
  std::vector<Tensor>* outputs,
  RunMetadata* run_metadata) {

  // 1. prune graph
  // client_graph = prune(full_graph, inputs, outputs)

  // 2. split graph into partition by devices
  // executors_and_partitions = split(client_graph, devices)
  ExecutorsAndKeys* executors_and_keys = ... // ignore implements...

  // 3. lauch executor per partition
  // def run_partitions(executors_and_partitions, inputs, outputs):
  //   frame = FunctionCallFrame()
  //   frame.set_args(inputs)
  //   for (executor, partition) in executors_and_partitions:
  //     exec.run(part)
  //   frame.get_ret_vals(outputs)

  // 3.1 construct FunctionCallFrame
  FunctionCallFrame call_frame(
    executors_and_keys->input_types,
    executors_and_keys->output_types);

  // 3.2 frame.set_args(inputs)
  // 3.2.1 construct feeds list
  gtl::InlinedVector<Tensor, 4> feed_args(inputs.size());
```

```
  for (const auto& it : inputs) {
    // (first, second) => (tensor_name, tensor)
    feed_args[executors_and_keys->input_name_to_index[it.first]] = it.second;
  }

  // 3.2.2 frame.set_args(feeds)
  call_frame.SetArgs(feed_args);

  // 3.3 concurent execution
  // barrier = ExecutorBarrier(executors_and_partitions.size())
  // for (executor, partition) in executors_and_partitions:
  //   executor.run(partition)
  // barrier.wait()
  RunState run_state(&devices_);
  run_state.rendez = new IntraProcessRendezvous(device_mgr_.get());

  // 3.3.1 notify when finished.
  size_t num_executors = executors_and_keys->items.size();
  ExecutorBarrier* barrier = new ExecutorBarrier(
      num_executors, run_state.rendez, [&run_state](const Status& ret) {
        {
          mutex_lock l(run_state.mu_);
          run_state.status.Update(ret);
        }
        run_state.executors_done.Notify();
      });

  Executor::Args args;
  args.call_frame = &call_frame;
  args.rendezvous = run_state.rendez;
  args.runner = [this, pool](Executor::Args::Closure c) {
    SchedClosure(pool, std::move(c));
  };

  // 3.3.2 lauch all executors.
  for (const auto& item : executors_and_keys->items) {
    item.executor->RunAsync(args, barrier->Get());
  }

  // 3.3.3 wait until all executors finished.
  WaitForNotification(&run_state,
      &step_cancellation_manager,
      GetTimeoutInMs(run_options));

  // 3.4 fetch outputs.
}
```

输出

同理，对于 `RetVal` 节点，其调用时序为：`set_ret_val -> get_ret_val`。前者由 `RetVal` 完成，后者由 `DirectSession` 完成。

```
struct RetvalOp : OpKernel {
  explicit RetvalOp(OpKernelConstruction* ctx) : OpKernel(ctx) {
    ctx->GetAttr("T", &dtype_);
    ctx->GetAttr("index", &index_);
  }

  void Compute(OpKernelContext* ctx) override {
    // get upstream op's output.
    const Tensor& val = ctx->input(0);

    auto frame = ctx->call_frame();
    frame->SetRetval(index_, val);
```

```
  }

 private:
  int index_;
  DataType dtype_;
};
```

等所有 Executor 运行结束后，DirectSession 便可以从 FunctionCallFrame 中取出所有输出值，并将其放置在 outputs，并返回 Client。

```
Status DirectSession::Run(
  const RunOptions& run_options,
  const NamedTensorList& inputs,
  const std::vector<string>& output_names,
  const std::vector<string>& target_nodes,
  std::vector<Tensor>* outputs,
  RunMetadata* run_metadata) {

  // 1. prune graph
  // client_graph = prune(full_graph, inputs, outputs)

  // 2. split graph into partition by devices
  // executors_and_partitions = split(client_graph, devices)
  executors_and_keys = ... // ignore implements...

  // 3. lauch executor per partition
  // def run_partitions(executors_and_partitions, inputs, outputs):
  //   frame = FunctionCallFrame()
  //   frame.set_args(inputs)
  //   for (executor, partition) in executors_and_partitions:
  //     exec.run(part)
  //   frame.get_ret_vals(outputs)

  // 3.1 construct FunctionCallFrame
  FunctionCallFrame call_frame(
    executors_and_keys->input_types,
    executors_and_keys->output_types);

  // 3.2 frame.set_args(inputs)
  // 3.2.1 construct feeds list
  gtl::InlinedVector<Tensor, 4> feed_args(inputs.size());
  for (const auto& it : inputs) {
    // (first, second) => (tensor_name, tensor)
    feed_args[executors_and_keys->input_name_to_index[it.first]] = it.second;
  }

  // 3.2.2 frame.set_args(feeds)
  call_frame.SetArgs(feed_args);

  // 3.3 concurent execution
  // barrier = ExecutorBarrier(executors_and_partitions.size())
  // for (executor, partition) in executors_and_partitions:
  //   executor.run(partition)
  // barrier.wait()
  RunState run_state(&devices_);
  run_state.rendez = new IntraProcessRendezvous(device_mgr_.get());

  // 3.3.1 notify when finished.
  size_t num_executors = executors_and_keys->items.size();
  ExecutorBarrier* barrier = new ExecutorBarrier(
      num_executors, run_state.rendez, [&run_state](const Status& ret) {
        {
          mutex_lock l(run_state.mu_);
          run_state.status.Update(ret);
        }
        run_state.executors_done.Notify();
      });
```

```
    Executor::Args args;
    args.call_frame = &call_frame;
    args.rendezvous = run_state.rendez;
    args.runner = [this, pool](Executor::Args::Closure c) {
      SchedClosure(pool, std::move(c));
    };

    // 3.3.2 lauch all executors.
    for (const auto& item : executors_and_keys->items) {
      item.executor->RunAsync(args, barrier->Get());
    }

    // 3.3.3 wait until all executors finished.
    WaitForNotification(&run_state,
        &step_cancellation_manager,
        GetTimeoutInMs(run_options));

    // 3.4 fetch outputs.
    // 3.4.1 frame.get_get_ret_vals
    std::vector<Tensor> sorted_outputs;
    Status s = call_frame.ConsumeRetvals(&sorted_outputs);

    // 3.4.2 emplace to outputs, and return to client.
    outputs->reserve(sorted_outputs.size());
    for (int i = 0; i < output_names.size(); ++i) {
      const string& output_name = output_names[i];
      outputs->emplace_back(
        std::move(sorted_outputs[
          executors_and_keys->output_name_to_index[output_name]]));
    }
  }
```

至此，整个 DirectSession.Run 解读完毕。但是，Partition 中节点如何被调度执行的，Partition 之间的 Send/Recv 是如何工作的呢？

因此，在最后一公里，还要探究三件事情。

1. SendOp 与 RecvOp 的工作原理
2. IntraProcessRendezvous 的工作原理
3. Executor 的调度算法

12.6 设备间通信

SendOp/RecvOp 通过 Rendezvous 交换数据的；它实现了消息发送/接受，与具体消息传递相解耦。例如，在单进程内，SendOp/RecvOp 基于 IntraProcessRendezvous 传递数据的；而在多进程环境中，SendOp/RecvOp 则可以基于 GrpcRendezvous 传递数据。

首先，探究这两个 OP 的工作原理；然后，再探究本地模式下，IntraProcessRendezvous 的工作原理。

SendOp 实现

如图??（第??页）所示，进程内的 Send/Recv 通过唯一的标识 ParsedKey 实现数据的交换。

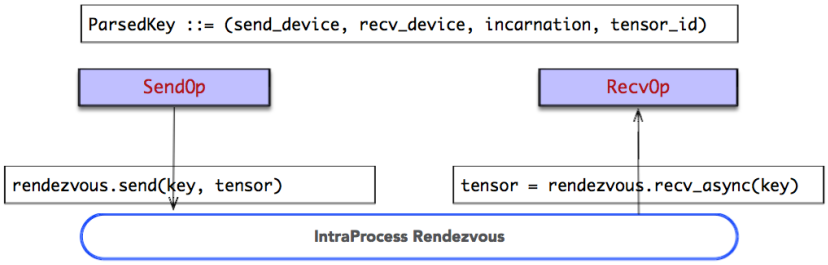


图 12-24 进程内 SendOp 与 RecvOp 的数据交换

参考 SendOp 的 Kernel 实现，看起来非常复杂，但是它实际上就做了一件事情。首先，它构造设备间通信的关键字 ParsedKey，然后调用 Rendezvous.Send 操作，将上游 OP 输入到 SendOp 的 Tensor 发送到 Rendezvous 缓存之中，该操作是非阻塞的。

其中，ParsedKey 包括：发送设备，接受设备，设备全局标识，及其待发送 Tensor 的标识 (src:output_index) 组成。

```
struct SendOp : OpKernel {
  explicit SendOp(OpKernelConstruction* ctx) : OpKernel(ctx) {
    string send_device;
    ctx->GetAttr("send_device", &send_device);

    string recv_device;
    ctx->GetAttr("recv_device", &recv_device);

    uint64 send_device_incarnation;
    ctx->GetAttr(
        "send_device_incarnation",
        reinterpret_cast<int64*>(&send_device_incarnation));

    string tensor_name;
    ctx->GetAttr("tensor_name", &tensor_name);

    key_prefix_ = GetRendezvousKeyPrefix(
        send_device, recv_device,
        send_device_incarnation, tensor_name);

    GetRendezvousKey(key_prefix_, {0, 0}, &parsed_key_.buf_);
    Rendezvous::ParseKey(parsed_key_.buf_, &parsed_key_);

    if (!ctx->GetAttr("_hostmem_sendrecv", &hostmem_sendrecv_).ok()) {
      hostmem_sendrecv_ = false;
    }
  }

  void Compute(OpKernelContext* ctx) override {
    Rendezvous::Args args;
    args.device_context = ctx->op_device_context();
    args.alloc_attrs = ctx->input_alloc_attr(0);

    // get it from upstream op's output, and as this op's input.
```

```
      ctx->rendezvous()->Send(
          CreateParsedkey(ctx), args, ctx->input(0),
          ctx->is_input_dead());
    }

   private:
    Rendezvous::ParsedKey CreateParsedkey(OpKernelContext* ctx) {
      FrameAndIter frame_iter = GetFrameAndIter(ctx, hostmem_sendrecv_);
      if (frame_iter == FrameAndIter(0, 0)) {
        return parsed_key_;
      } else {
        Rendezvous::ParsedKey in_loop_parsed;
        GetRendezvousKey(key_prefix_, frame_iter, &in_loop_parsed.buf_);
        Rendezvous::ParseKey(in_loop_parsed.buf_, &in_loop_parsed);
        return in_loop_parsed;
      }
    }

   private:
    string key_prefix_;
    Rendezvous::ParsedKey parsed_key_;
    bool hostmem_sendrecv_;
  };
```

RecvOp 实现

同理地，可以猜测 Recv 的 Kernel 的实现了。它首先构造 Rendezvous 的 ParsedKey，然后调用 Rendezvous.RecvAsync 操作，从 Rendezvous 取出相应的 Tensor。

这是一个异步操作，当 Rendezvous 中数据可获取，便开始执行回调函数 done_cb，它将其得到的 Tensor 输出到下游 OP。

```
  struct RecvOp : AsyncOpKernel {
    explicit RecvOp(OpKernelConstruction* ctx) : AsyncOpKernel(ctx) {
      string send_device;
      ctx->GetAttr("send_device", &send_device);

      string recv_device;
      ctx->GetAttr("recv_device", &recv_device);

      uint64 send_device_incarnation;
      ctx->GetAttr(
          "send_device_incarnation",
          reinterpret_cast<int64*>(&send_device_incarnation));

      string tensor_name;
      ctx->GetAttr("tensor_name", &tensor_name);

      key_prefix_ = GetRendezvousKeyPrefix(
          send_device, recv_device,
          send_device_incarnation, tensor_name);

      GetRendezvousKey(key_prefix_, {0, 0}, &parsed_key_.buf_);
      Rendezvous::ParseKey(parsed_key_.buf_, &parsed_key_);
      if (!ctx->GetAttr("_hostmem_sendrecv", &hostmem_sendrecv_).ok()) {
        hostmem_sendrecv_ = false;
      }
    }

    void ComputeAsync(OpKernelContext* ctx, DoneCallback done) override {
      Rendezvous::Args args;
      args.device_context = ctx->op_device_context();
      args.alloc_attrs = ctx->output_alloc_attr(0);
```

```
    ctx->rendezvous()->RecvAsync(
      CreateParsedKey(ctx), args, CreateDoneCallback(ctx));
  }

 private:
  Rendezvous::ParsedKey CreateParsedKey(OpKernelContext* ctx) {
    FrameAndIter frame_iter = GetFrameAndIter(ctx, hostmem_sendrecv_);
    if (frame_iter == FrameAndIter(0, 0)) {
      return parsed_key_;
    } else {
      Rendezvous::ParsedKey in_loop_parsed;
      GetRendezvousKey(key_prefix_, frame_iter, &in_loop_parsed.buf_);
      Rendezvous::ParseKey(in_loop_parsed.buf_, &in_loop_parsed);
      return in_loop_parsed;
    }
  }

  Rendezvous::DoneCallback CreateDoneCallback(OpKernelContext* ctx) {
    using namespace std::placeholders;
    return std::bind([ctx](DoneCallback done, const Status& s,
        const Rendezvous::Args&, const Rendezvous::Args&,
        const Tensor& val, bool is_dead) {
          ctx->SetStatus(s);
          if (s.ok()) {
            if (!is_dead) {
              // put it into downstream op's input.
              ctx->set_output(0, val);
            }
            *ctx->is_output_dead() = is_dead;
          }
          done();
        },
        std::move(done), _1, _2, _3, _4, _5);
  }

 private:
  string key_prefix_;
  Rendezvous::ParsedKey parsed_key_;
  bool hostmem_sendrecv_;
};
```

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

- Martin Flower

13

分布式 TensorFlow

TensorFlow 可以运行在分布式环境中，完成计算图的执行过程。本章将重点介绍分布式运行时的基本架构与运行机制；重点讨论各个服务进程之间的交互关系；并且深入剖析在分布式环境中图操作，及其会话生命周期控制的关键技术；

13.1 分布式模式

在分布式模式中，Client 负责计算图的构造，然后通过调用 `Session.run`，启动计算图的执行过程。

Master 进程收到计算图执行的消息后，启动计算图的剪枝，分裂，优化等操作；最终将子图分发注册到各个 Worker 进程上，然后触发各个 Worker 进程并发执行子图。

Worker 进程收到子图注册的消息后，根据本地计算设备资源，再将计算子图实施二次分裂，将子图分配在各个计算设备上，最后启动各个计算设备并发地执行子图；如果 Worker 之间存在数据交换，可以通过进程间通信完成交互。

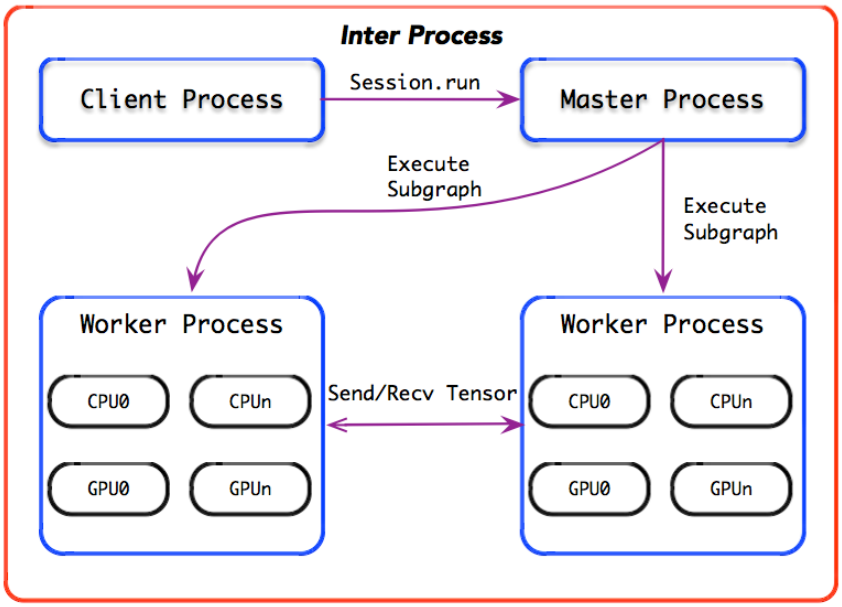


图 13-1 分布式模式

图操作

如图**??**（第**??**页）所示，在 `run_step` 执行过程之中，涉及计算图的剪枝、分裂、执行三个重要的图操作。其中，在分布式运行时，图分裂经历了两级分裂过程。

1. 一级分裂：由 `MasterSession` 完成，按照 `SplitByWorker` 或 `SplitByTask` 完成图分裂过程；

2. 二级分裂：由 `WorkerSession` 完成，按照 `SplitByDevice` 完成图分裂过程。

在分布式模式中，图剪枝也体现了 TensorFlow 部分执行的设计理念；而图分裂和执行也体现了 TensorFlow 并发执行的设计理念。其中，图剪枝仅发生在 Master 上，不发生在 Worker 上；而图分裂发生在 Master 和 Worker 上；图执行仅仅发生在 Worker 上，不发生在 Master 上。

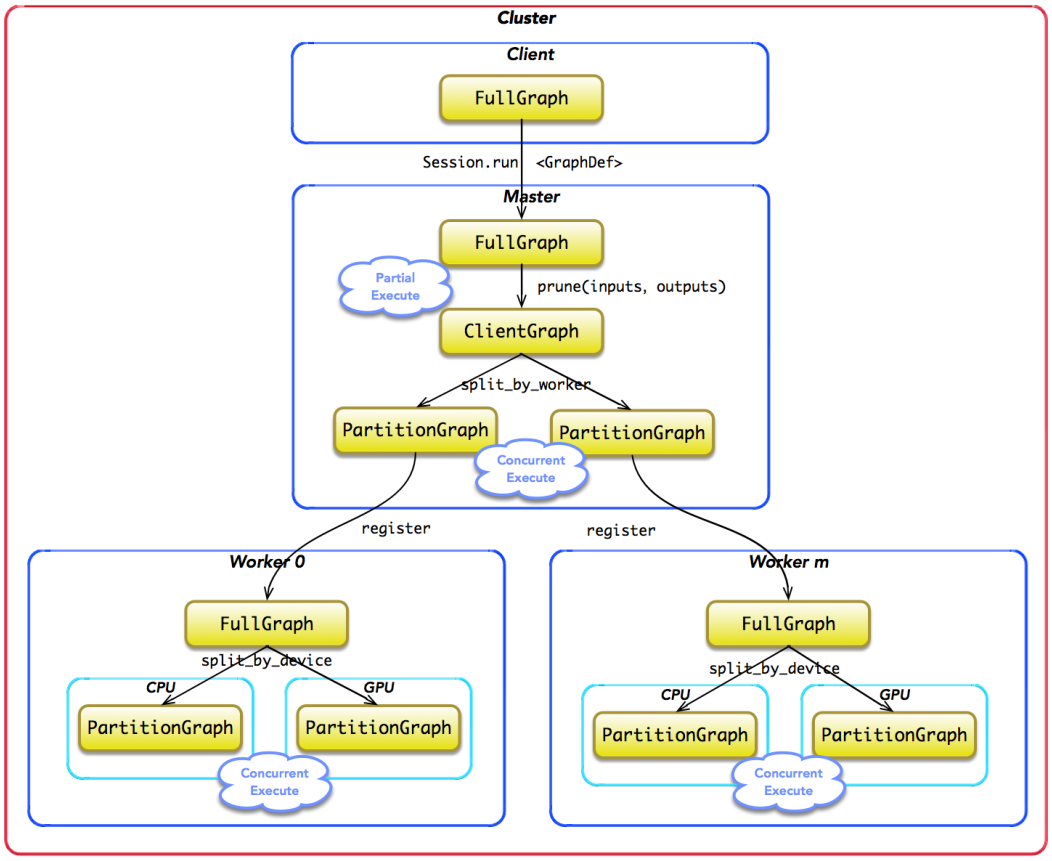


图 13-2 分布式：图操作

图分裂

为了更好地理解分布式运行时的工作原理，以一个简单的例子阐述图操作的具体过程。如图**??**（第**??**页）所示，假如存在一个简单的计算图，并且 `f`，`c`，`a` 部署在`/job:ps/task:0` 上，且分别被编排到 `CPU0`，`CPU1`，`CPU2` 上；`g`，`h` 部署在`/job:worker/task:0` 上，且同时被编排到 `GPU0` 上；`b`，`d`，`e` 部署在`/job:worker/task:1` 上，且 `d`，`e` 被编排到 `GPU0` 上，而 `b` 被编排到 `GPU1` 上。

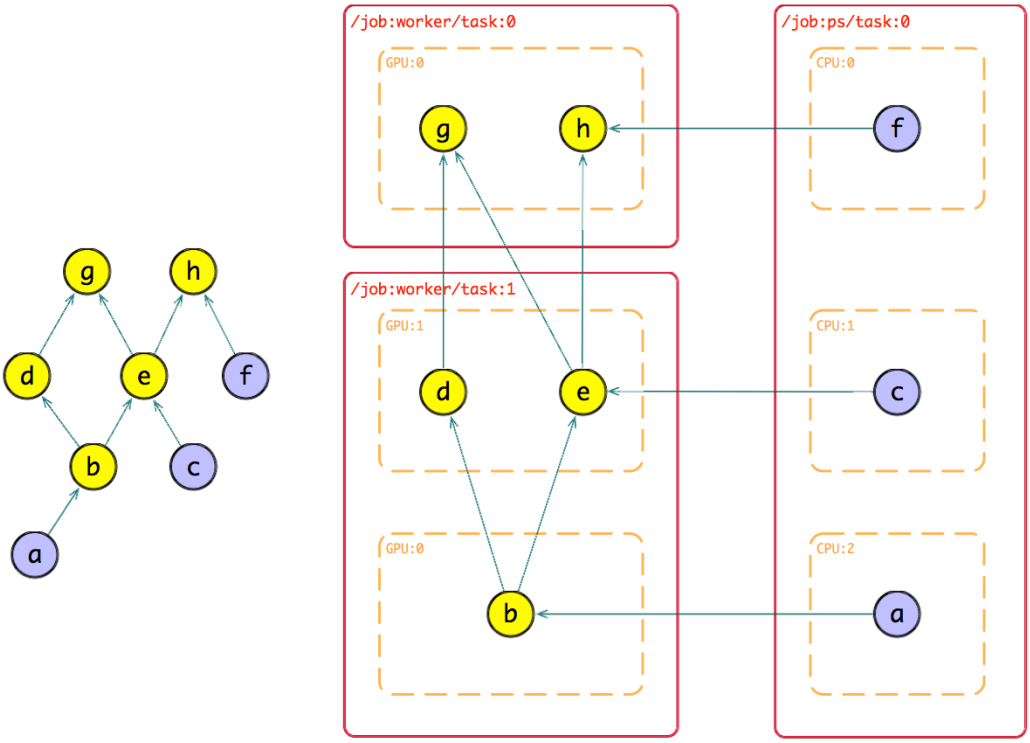


图 13-3 分布式：图分裂

数据交换

如图??（第??页）所示，对于跨设备的边，运行时自动实施边的分裂，分别在发送端和接收端插入 Send 和 Recv 两个末端节点。

进程间的 Send 和 Recv 节点，通过 GrpcRemoteRendezvous 实现数据交换。例如，/job:ps/task:0 与/job:worker/task:0,/job:ps/task:0 与/job:worker/task:1,或/job:worker/task:0 与/job:worker/task:1 之间是通过 GrpcRemoteRendezvous 完成数据交换的。

而进程内的 Send 和 Recv 节点，则通过 IntraProcessRendezvous 实现数据交换。例如，/job:worker/task:1 内存在两个 GPU，它们之间采用 IntraProcessRendezvous 实现数据交换。关于 Rendezvous 的具体实现过程，下文将还会重点讲述。

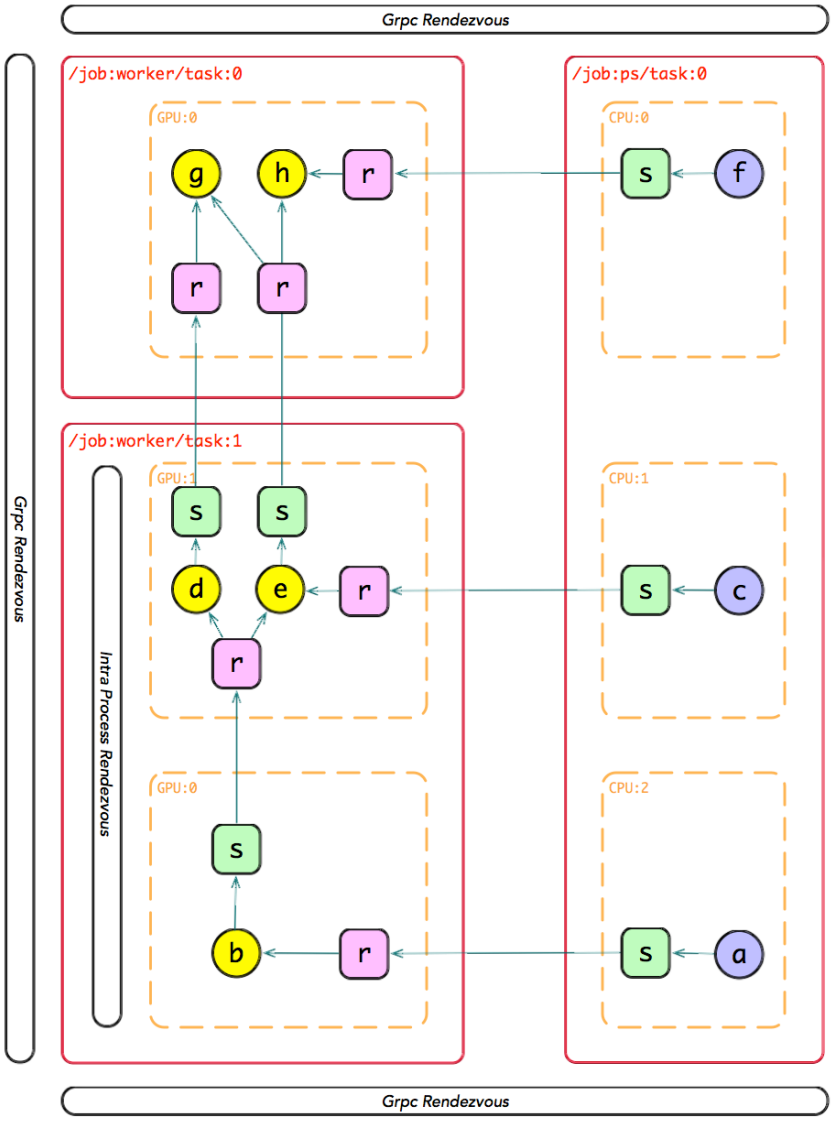


图 13-4 分布式：数据交换

形式化

在真实的系统实现中，分布式的运行时使用 C++ 实现。其中，TensorFlow 运行时的关键路径为 `run_step`。因为真实系统实现中涉及过多的细节，不易发现算法的主干和逻辑。为了简化问题的描述，将形式化地描述 `run_step` 的实现过程。

Master::RunStep

在 Master 上，主要完成 `FullGraph` 的剪枝操作，生成 `ClientGraph`；然后，按照 Worker 将 `ClientGraph` 分裂为多个 `PartitionGraph`；最后，将 `PartitionGraph` 列表注册给各个 Worker，并启动各个 Worker 并发执行 `PartitionGraph` 列表。

```
def run_step(workers, full_graph, inputs, outputs):
  client_graph = prune(full_graph, inputs, outputs)
  partition_graphs = split(client_graph, workers)
  register_graphs(partition_graphs, inputs, outputs)
  run_graphs(partition_graphs, inputs, outputs)
```

Worker::RunStep

在某个特定的 Worker 节点上，当收到 RegisterGraphRequest 消息后，将计算图按照本地设备集分裂为多个 PartitionGraph。然后，在每个计算设备启动一个 Executor，以便执行分配给它的 PartitionGraph。

当某一个计算设备执行完所分配的 PartitionGraph 之后，ExecutorBarrier 的计数器加 1，直至所有设备完成 PartitionGraph 列表的执行，barrier.wait() 阻塞操作退出。

跨设备的 PartitionGraph 之间可能存在数据依赖关系，它们之间通过插入 Send/Recv 节点完成交互。事实上，在分布式运行时，Send/Recv 通过 RpcRemoteRendezvous 完成数据交换的。

```
def send_inputs(remote_rendezvous, inputs):
  for (key, tensor) in inputs:
    remote_rendezvous.send(key, tensor)

def do_run_partitions(executors_and_partitions):
  barrier = ExecutorBarrier(executors_and_partitions.size())
  for (executor, partition) in executors_and_partitions:
    executor.run(partition, barrier.on_done())
  barrier.wait()

def recv_outputs(remote_rendezvous, outputs):
  for (key, tensor) in outputs:
    remote_rendezvous.recv(key, tensor)

def run_partitions(executors_and_partitions, inputs, outputs):
  remote_rendezvous = RpcRemoteRendezvous()
  send_inputs(remote_rendezvous, inputs)
  do_run_partitions(executors_and_partitions)
  recv_outputs(remote_rendezvous, outputs)

def run_step(devices, full_graph, inputs, outputs):
  executors_and_partitions = split(full_graph, devices)
  run_partitions(executors_and_partitions, inputs, outputs)
```

领域模型

如图**??**（第**??**页）所示，在 TensorFlow 分布式运行时，存在一个精巧的领域模型。

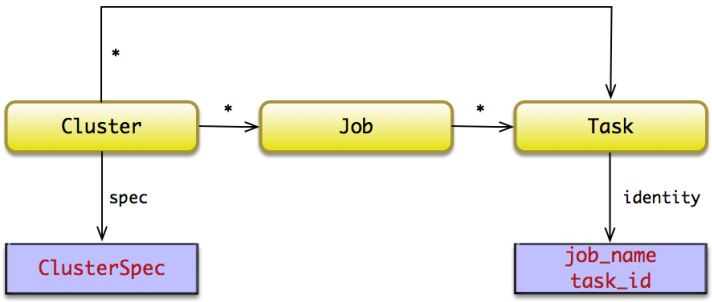


图 13-5 分布式：领域模型

Cluster

Cluster 使用 ClusterSpec 进行描述，它可以划分为一个或多个 Job，一个 Job 包含一个或多个 Task。也就是说，TensorFlow 集群是由执行计算图的任务集 (Task Set) 组成的。

每个 Task 可以独立运行在单独的机器上，也可以在一台机器上运行多个 Task(例如，单机多 CPU，或单机多 GPU)。

Job

将目的相同的 Task 划归在同一个 Job 中。每个 Job 使用 `job_id` 唯一标识。

一般地，在分布式深度学习的模型训练过程中，存在两种基本的 Job 类型：

1. ps：负责模型参数的存储和更新；
2. worker：负责计算密集型的模型训练和推理。

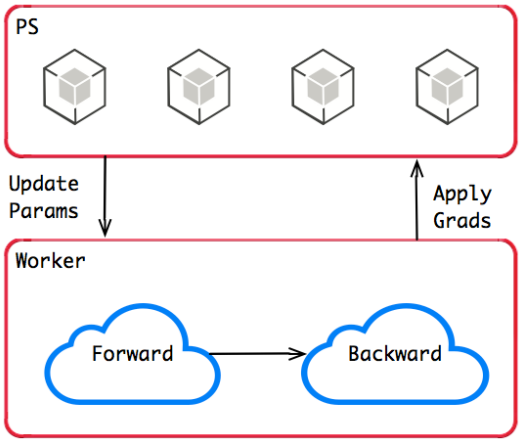


图 13-6 分布式模型训练：PS 与 Worker 之间的交互

Task

一般地，在分布式运行时中，Task 运行在独立的进程中，并在其上运行一个 `tf.train.Server` 实例。其中，Task 使用 `job_id:task_index` 的二元组唯一标识。

Server

Server 表示 Task 的服务进程，它对外提供 `MasterService` 和 `WorkerService` 服务。也就是说，Server 可以同时扮演 Master 和 Worker 两种角色。

组建集群

在分布式的 TensorFlow 运行时中，每个 Task 启动了一个 Server，并对外提供 `MasterService` 服务和 `WorkerService` 服务。其中，组建 TensorFlow 集群包括两个基本步骤:

1. 创建 `tf.train.ClusterSpec`，描述集群中 Task 的部署信息，并以 Job 的方式组织；
2. 对于每一个 Task，启动一个 `tf.train.Server` 实例。

集群配置

`ClusterSpec` 描述了集群中 Task 的部署信息，并以 Job 的方式组织。一般地，在分布式的执行模式中，为每个 Task 启动一个进程。因此，`ClusterSpec` 同时也描述了 TensorFlow 分布式运行时的进程分布情况。

例如，存在一个 TensorFlow 集群，它由 `ps` 和 `worker` 两个 Job 组成。其中，`ps` 部署在 `ps0:2222, ps1:2222` 上；`worker` 部署在 `worker0:2222, worker1:2222, worker2:2222` 上。

```
tf.train.ClusterSpec({
  "worker": [
    "worker0:2222",    # /job:worker/task:0
    "worker1:2222",    # /job:worker/task:1
    "worker2:2222"     # /job:worker/task:2
  ],
  "ps": [
    "ps0:2222",        # /job:ps/task:0
    "ps1:2222"         # /job:ps/task:0
  ]})
```

在此例中，未显式地指定 Task 的索引。默认地，一个 Job 的 Task 集合中，Task 索引从 0 开始按序自增的。

Protobuf 描述

```
message JobDef {
  string name = 1;
  map<int32, string> tasks = 2;
}

message ClusterDef {
  repeated JobDef job = 1;
}
```

其中，tasks 的关键字表示 task_index，值表示 host:port。

13.2 Master 服务

MasterService 是一个 RPC 服务。当 Client 根据 target 接入 Server 实例后，Server 扮演了 Master 的角色，对外提供 MasterService 服务。

其中，Client 与 Master 之间的交互遵循 MasterService 定义的接口规范。也就是说，MasterService 定义了 Client 接入 Master 的公共契约，负责协调和控制多个 WorkerService 的执行过程。

接口定义

在 master_service.proto 文件中，定义了 MasterService 的所有接口；而在 master.proto 文件中，定义了各个接口的消息体。

```
service MasterService {
  rpc CreateSession(CreateSessionRequest)
      returns (CreateSessionResponse);

  rpc ExtendSession(ExtendSessionRequest)
      returns (ExtendSessionResponse);

  rpc PartialRunSetup(PartialRunSetupRequest)
      returns (PartialRunSetupResponse);

  rpc RunStep(RunStepRequest)
      returns (RunStepResponse);

  rpc CloseSession(CloseSessionRequest)
      returns (CloseSessionResponse);

  rpc ListDevices(ListDevicesRequest)
      returns (ListDevicesResponse);
```

```
  rpc Reset(ResetRequest)
      returns (ResetResponse);
}
```

访问服务

一般地，Client 使用接口 `MasterInterface` 获取远端 `MasterService` 的服务。特殊地，`MasterInterface` 的所有接口都是同步接口，使得 Client 访问远端 `MasterService` 服务犹如调用本地函数一般。

需要注意的是，因为 `RunStepRequest/RunStepResponse` 消息中可能包含较大的 `Tensor` 实例。为了避免不必要的对象拷贝，实现特化实现了消息包装器。

```
// Abstract interface for communicating with the TensorFlow Master service.
//
// This interface supports both RPC-based master implementations, and
// in-process master implementations that do not require an RPC roundtrip.
struct MasterInterface {
  virtual ~MasterInterface() {}

  virtual Status CreateSession(
      CallOptions* call_options,
      const CreateSessionRequest* request,
      CreateSessionResponse* response) = 0;

  virtual Status ExtendSession(
      CallOptions* call_options,
      const ExtendSessionRequest* request,
      ExtendSessionResponse* response) = 0;

  virtual Status PartialRunSetup(
      CallOptions* call_options,
      const PartialRunSetupRequest* request,
      PartialRunSetupResponse* response) {
    return errors::Unimplemented(
      "Partial run not implemented for master");
  }

  virtual Status RunStep(
      CallOptions* call_options,
      RunStepRequestWrapper* request,
      MutableRunStepResponseWrapper* response) = 0;

  // Wrapper classes for the `MasterService.RunStep` message.
  //
  // The `RunStepRequest/RunStepResponse` message can contain
  // potentially large tensor data as part of its `feed/fetch`
  // submessages.
  virtual Status RunStep(
    CallOptions* call_options,
    const RunStepRequest* request,
    RunStepResponse* response) {
    std::unique_ptr<RunStepRequestWrapper> wrapped_request(
        new ProtoRunStepRequest(request));
    std::unique_ptr<MutableRunStepResponseWrapper> wrapped_response(
        new NonOwnedProtoRunStepResponse(response));
    return RunStep(call_options,
        wrapped_request.get(),
        wrapped_response.get());
  }
```

```
  // Returns a request object for use in calls to
  // `RunStep()`. Ownership is transferred to the caller.
  virtual MutableRunStepRequestWrapper* CreateRunStepRequest() {
    return new MutableProtoRunStepRequest;
  }

  // Returns a response object for use in calls to
  // `RunStep()`. Ownership is transferred to the caller.
  virtual MutableRunStepResponseWrapper* CreateRunStepResponse() {
    return new OwnedProtoRunStepResponse;
  }

  virtual Status CloseSession(
    CallOptions* call_options,
    const CloseSessionRequest* request,
    CloseSessionResponse* response) = 0;

  virtual Status ListDevices(
    CallOptions* call_options,
    const ListDevicesRequest* request,
    ListDevicesResponse* response) = 0;

  virtual Status Reset(
    CallOptions* call_options, const ResetRequest* request,
    ResetResponse* response) = 0;
};
```

如图**??**（第**??**页）所示，MasterInterface 存在两种基本实现。

1. 分布式：基于 gRPC 的 GrpcRemoteMaster 实现，Client 与 Master 分别部署在两个不同的进程；

2. 本地模式：基于函数调用的 LocalMaster 实现，Client 与 Master 在同一个进程内。

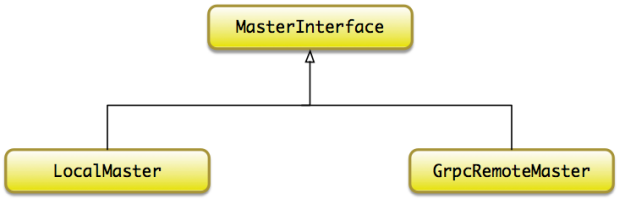


图 13-7 MasterInterface

在分布式模式中，GrpcRemoteMaster 使用如下类似的伪代码，并通过 gRPC 获取远端 MasterService 服务。

```
stub = NewStub("/job:worker/replica:0/task:0")
handle = stub->CreateSession({graph_def})
do {
  stub->RunStep(handle, feeds, fetches);
} while (!should_stop());
stub->CloseSession({handle})
```

RPC 过程

如图??（第??页）所示，Client 通过 MasterInterface 获取远端 MasterService 的服务。

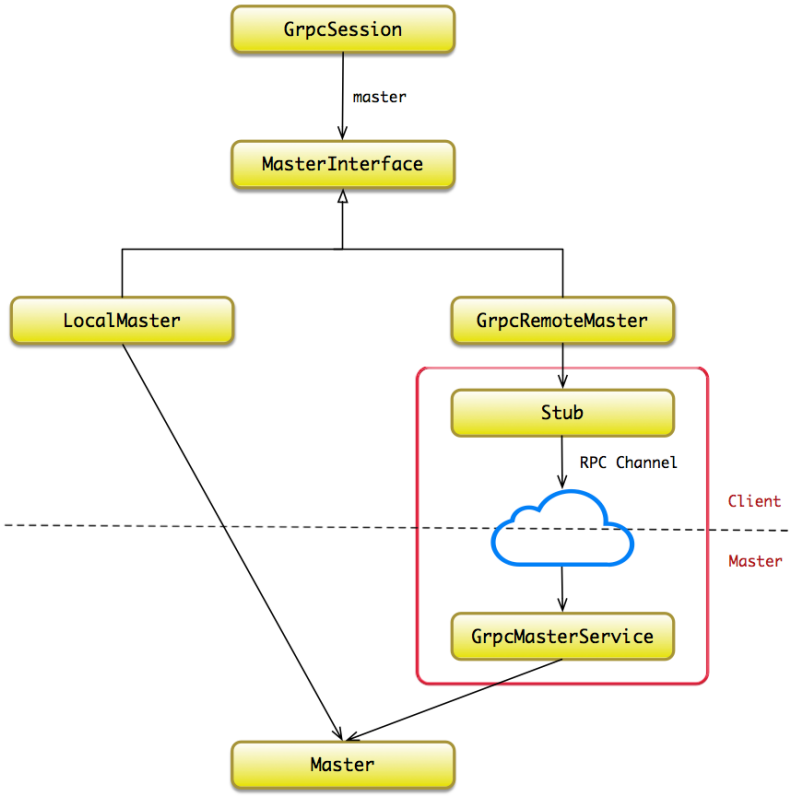


图 13-8 Client 获取 MasterService 的原理

其中，GrpcRemoteMaster 是 gRPC 客户端的一种实现，它最终通过 Stub 获取远端 Master 上的 GrpcMasterService 服务，使得其行为表现得犹如本地函数调用一般。其中，GrpcMasterService 实现了 MasterService 定义的所有服务接口，它是 MasterService 真正的服务实体。

R 从严格意义上讲，GrpcSession，ClientMaster，GrpcRemoteMaster 都是 Client 实现的一部分。而不是通常理解的那样，Python 前端系统是完整的 Client 实现，后端 C++ 后端系统不包括 Client 的任何实现。

消息定义

接下来，将详细看看各个接口的消息定义。其中，最重要的就是识别出各个服务的标识。例如，Master 可以供多个 Client 接入，并为每个 Client 生成对应的 MasterSession 实例。因此 GrpcSession 持有 MasterSession 句柄，实现 Client 获取 Master 的服务。

CreateSession

如图??（第??页）所示，CreateSessionRequest 消息中携带初始的计算图，并与 target 指定的 Master 建立连接。当 Master 收到请求消息后，建立一个相对应的 MasterSession 实例，并使用 session_handle 唯一地标识该 MasterSession 实例。

待 Master 逻辑处理完成后，通过返回消息 CreateSessionResponse 给 Client。其中，CreateSessionResponse 消息中携带 session_handle，通过它 Client 端的 GrpcSession 与 Master 端的 MasterSession 建立关联关系。随后，Client 与 Master 的所有交互中，在请求消息中通过携带 session_handle，Master 通过它索引与之相对应的 MasterSession 实例。

此外，CreateSessionResponse 也携带了初始的 graph_version，用于后续发起 ExtendSession 操作，往原始的计算图中追加新的节点。

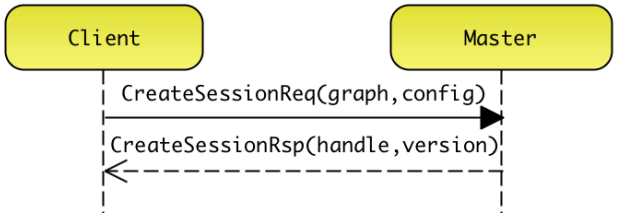


图 13-9 CreateSession

```
message CreateSessionRequest {
  GraphDef graph_def = 1;
  ConfigProto config = 2;
  string target = 3;
}

message CreateSessionResponse {
  string session_handle = 1;
  int64 graph_version = 2;
}
```

ExtendSession

当 CreateSession 成功后，后续 Client 可以通过 ExtendSession，携带待扩展的子图给 Master，增加原有计算图的规模 (只能追加子图，不能修改或删除节点)。

如图??(第??页)所示,在请求消息中需要携带 current_graph_version,Master 端进行版本匹配验证；待 ExtendSession 的逻辑处理完成后，在响应消息中携带 new_graph_version，用于下一此 ExtendSession 操作。其中，初始的 graph_version 由 CreateSessionResponse 携带给 Client 的。

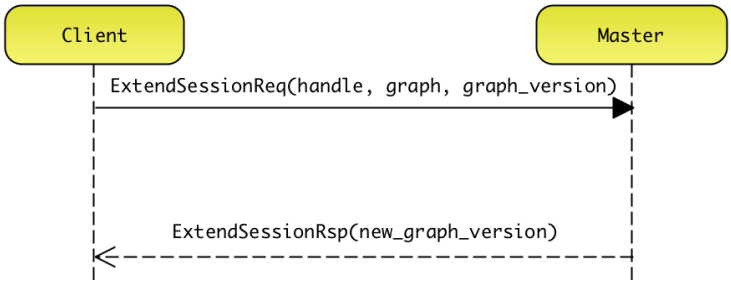


图 13-10 ExtendSession

```
message ExtendSessionRequest {
  string session_handle = 1;

  // REQUIRED: The nodes to be added to the session's graph.
  // If any node has the same name as an existing node,
  // the operation will fail with ILLEGAL_ARGUMENT.
  GraphDef graph_def = 2;

  // REQUIRED: The version number of the graph to be extended.
  // This will be tested against the current server-side version
  // number, and the operation will fail with FAILED_PRECONDITION
  // if they do not match.
  int64 current_graph_version = 3;
}

message ExtendSessionResponse {
  // The new version number for the extended graph,
  // to be used in the next call to ExtendSession.
  int64 new_graph_version = 4;
}
```

RunStep

一般地，在客户端迭代地执行 RunStep。如图??（第??页）所示，在每一次 RunStep 执行过程中，Client 在请求消息中携带 feed，fetch，target，分别表示输入的 NamedTensor 列表，待输出 Tensor 的名称列表，待执行 OP 的名称列表；在响应消息中携带 tensor，表示对应于 fetch 的名字列表，输出的 Tensor 列表。

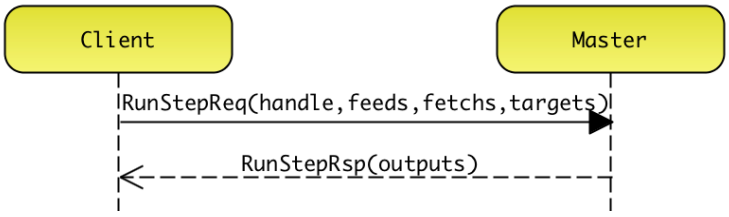


图 13-11 RunStep

```
message RunStepRequest {
  string session_handle = 1;

  repeated NamedTensorProto feed = 2;
```

```
  repeated string fetch = 3;
  repeated string target = 4;

  RunOptions options = 5;
  string partial_run_handle = 6;
}

message RunStepResponse {
  repeated NamedTensorProto tensor = 1;
  RunMetadata metadata = 2;
}
```

CloseSession

当计算完成后，需要关闭会话，释放系统计算资源。如图**??**（第**??**页）所示，Client 通过发送 `CloseSession` 给 Master，启动计算资源的释放过程。

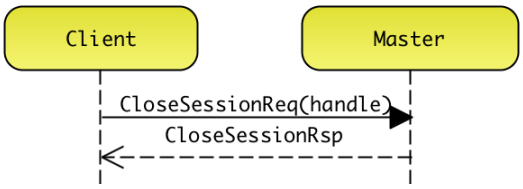


图 13-12 `CloseSession`

```
message CloseSessionRequest {
  string session_handle = 1;
}

message CloseSessionResponse {
}
```

13.3 Worker 服务

`WorkerService` 也是一个 gRPC 服务，负责调度本地设备集执行本地子图。它定义了接入 Worker 的接口规范，即 `master_service.proto` 中定义的接口。

Master 根据 `ClusterSpec` 信息，找到集群中其他的 Server 实例，此时这些 Server 实例将扮演 Worker 的角色。Master 将子图分发给各个 Worker 节点，并启动各个 Worker 节点的子图计算的执行过程。

如果 Worker 之间存在数据依赖，则通过进程间通信完成交互。其中，Master 与 Worker 之间，Worker 与 Worker 之间的交互遵循 `WorkerService` 定义的接口规范。

接口定义

在 worker_service.proto 文件中,定义了 WorkerService 的所有接口;而在 worker.proto 文件中，定义了各个接口的消息体。

```
service WorkerService {
  rpc GetStatus(GetStatusRequest)
      returns (GetStatusResponse);

  rpc CreateWorkerSession(CreateWorkerSessionRequest)
      returns (CreateWorkerSessionResponse);

  rpc RegisterGraph(RegisterGraphRequest)
      returns (RegisterGraphResponse);

  rpc DeregisterGraph(DeregisterGraphRequest)
      returns (DeregisterGraphResponse);

  rpc RunGraph(RunGraphRequest)
      returns (RunGraphResponse);

  rpc CleanupGraph(CleanupGraphRequest)
      returns (CleanupGraphResponse);

  rpc CleanupAll(CleanupAllRequest)
      returns (CleanupAllResponse);

  rpc RecvTensor(RecvTensorRequest)
      returns (RecvTensorResponse) {
  }

  rpc Logging(LoggingRequest)
      returns (LoggingResponse);

  rpc Tracing(TracingRequest)
      returns (TracingResponse);
}
```

访问服务

一般地，Master/Worker 使用接口 WorkerInterface 获取远端 WorkerService 的服务。其中，WorkerInterface 定义了异步访问 WorkerService 的接口；与 MasterInterface 类似，因为 RunGraphRequest/RunGraphResponse 中可能含有较大的 Tensor，为了避免不必要的对象拷贝，特化了实现了消息的包装器。

```
struct WorkerInterface {
  // async interfaces.
  virtual void GetStatusAsync(
      const GetStatusRequest* request,
      GetStatusResponse* response,
      StatusCallback done) = 0;

  virtual void CreateWorkerSessionAsync(
      const CreateWorkerSessionRequest* request,
      CreateWorkerSessionResponse* response,
      StatusCallback done) = 0;

  virtual void RegisterGraphAsync(
```

```
        const RegisterGraphRequest* request,
        RegisterGraphResponse* response,
        StatusCallback done) = 0;

    virtual void DeregisterGraphAsync(
        const DeregisterGraphRequest* request,
        DeregisterGraphResponse* response,
        StatusCallback done) = 0;

    virtual void RunGraphAsync(
        CallOptions* opts,
        RunGraphRequestWrapper* request,
        MutableRunGraphResponseWrapper* repsonse,
        StatusCallback done) = 0;

    // Wrapper classes for the `WorkerService.RunGraph` message.
    //
    // The `RunGraphRequest/RunGraphResponse` message can contain
    // potentially large tensor data as part of its `send/response`
    // submessages.
    virtual void RunGraphAsync(
        CallOptions* opts,
        const RunGraphRequest* request,
        RunGraphResponse* response,
        StatusCallback done) {
      RunGraphRequestWrapper* wrapped_request =
          new ProtoRunGraphRequest(request);
      MutableRunGraphResponseWrapper* wrapped_response =
          new NonOwnedProtoRunGraphResponse(response);
      RunGraphAsync(opts, wrapped_request, wrapped_response,
          [wrapped_request, wrapped_response, done](const Status& s) {
              done(s);
              delete wrapped_request;
              delete wrapped_response;
          });
    }

    // Returns a request object for use in calls to
    // `RunGraphAsync()`. Ownership is transferred to the caller.
    virtual MutableRunGraphRequestWrapper* CreateRunGraphRequest() {
      return new MutableProtoRunGraphRequest;
    }

    // Returns a response object for use in calls to
    // `RunGraphAsync()`. Ownership is transferred to the caller.
    virtual MutableRunGraphResponseWrapper* CreateRunGraphResponse() {
      return new OwnedProtoRunGraphResponse;
    }

    virtual void CleanupGraphAsync(
        const CleanupGraphRequest* request,
        CleanupGraphResponse* response,
        StatusCallback done) = 0;

    virtual void CleanupAllAsync(
        const CleanupAllRequest* request,
        CleanupAllResponse* response,
        StatusCallback done) = 0;

    virtual void RecvTensorAsync(
        CallOptions* opts,
        const RecvTensorRequest* request,
        TensorResponse* response,
        StatusCallback done) = 0;

    virtual void LoggingAsync(
        const LoggingRequest* request,
        LoggingResponse* response,
        StatusCallback done) = 0;

    virtual void TracingAsync(
        const TracingRequest* request,
```

```
        TracingResponse* response,
        StatusCallback done) = 0;
};
```

WorkerInterface 同时也定义了同步访问接口。同步接口通过 CallAndWait 的适配器，间接实现于异步接口之上。特殊地，同步接口使得 Master/Worker 调用远端 WorkerService 具有犹如调用本地函数一般。

```
struct WorkerInterface {
  // sync interfaces.
  Status GetStatus(
      const GetStatusRequest* request,
      GetStatusResponse* response) {
    return CallAndWait(&ME::GetStatusAsync, request, response);
  }

  Status CreateWorkerSession(
      const CreateWorkerSessionRequest* request,
      CreateWorkerSessionResponse* response) {
    return CallAndWait(&ME::CreateWorkerSessionAsync, request, response);
  }

  Status RegisterGraph(
      const RegisterGraphRequest* request,
      RegisterGraphResponse* response) {
    return CallAndWait(&ME::RegisterGraphAsync, request, response);
  }

  Status DeregisterGraph(
      const DeregisterGraphRequest* request,
      DeregisterGraphResponse* response) {
    return CallAndWait(&ME::DeregisterGraphAsync, request, response);
  }

  Status CleanupGraph(
      const CleanupGraphRequest* request,
      CleanupGraphResponse* response) {
    return CallAndWait(&ME::CleanupGraphAsync, request, response);
  }

  Status CleanupAll(
      const CleanupAllRequest* request,
      CleanupAllResponse* response) {
    return CallAndWait(&ME::CleanupAllAsync, request, response);
  }

  Status Logging(
      const LoggingRequest* request,
      LoggingResponse* response) {
    return CallAndWait(&ME::LoggingAsync, request, response);
  }

  Status Tracing(
      const TracingRequest* request,
      TracingResponse* response) {
    return CallAndWait(&ME::TracingAsync, request, response);
  }

 private:
  typedef WorkerInterface ME;

  template <typename Method, typename Req, typename Resp>
  Status CallAndWait(Method func, const Req* req, Resp* resp) {
    Status ret;
    Notification n;
    (this->*func)(req, resp, [&ret, &n](const Status& s) {
```

```
        ret = s;
        n.Notify();
      });
      n.WaitForNotification();
      return ret;
    }
  };
```

特殊地，WorkerInterface 生成的实例由 WorkerCacheInterface::ReleaseWorker 负责删除。因此，此处为了避免外部非法删除 WorkerInterface 实例，限制 WorkerInterface 的析构函数为 protected，并且声明 WorkerCacheInterface 为友元。

```
struct WorkerInterface {
 protected:
  virtual ~WorkerInterface() {}
  friend class WorkerCacheInterface;
};
```

如图**??**（第**??**页）所示，WorkerService 存在两种实现。其中，在本地模式中，直接使用 GrpcWorker；在分布式模式中，Worker 部署在另一个不同的进程内，使用 GrpcRemoteWorker。

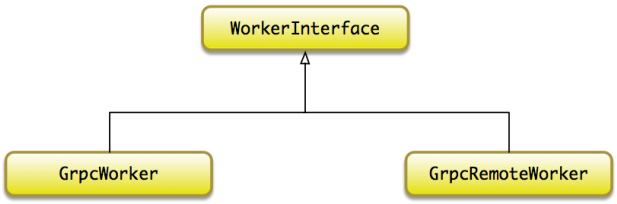


图 13-13 WorkerInterface 接口

RPC 过程

如图**??**（第**??**页）所示，在分布式模式中，GrpcRemoteWorker 是 gRPC 客户端的一种实现，它最终通过 Stub 获取远端 Worker 上的 GrpcWorkerService 服务，使得其行为表现得犹如本地函数调用一般。其中，GrpcWorkerService 实现了 WorkerService 定义的所有服务接口。

R 从严格意义上讲，GrpcRemoteWorker 是 Master 或者对端 Worker 实现的一部分。

而在本地模式中，通过 GrpcWorker 的函数调用，直接获取到了 WorkerService 的服务，避免了额外的网络传输开销。

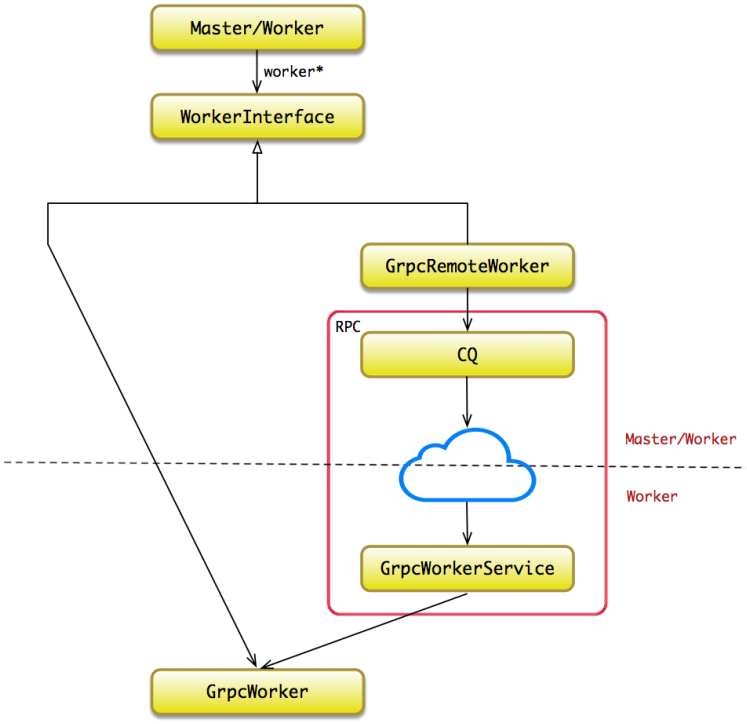


图 13-14 获取 MasterService 的 RPC 过程

消息定义

接下来，将详细看看 WorkerService 各个接口的消息定义。其中，最重要的就是识别出各个服务的标识。当创建 WorkerSession 时，MasterSession 的标识传递给 Worker，实现了 MasterSession 统一管理多个隶属的 WorkerSession 实例。

当 Worker 首次完成 RegisterGraph 后，向 Master 返回唯一的 graph_handle，以此标识该图实例。因此，在集群内可以使用 (session_handle, graph_handle) 二元组唯一标识该图实例。

当 Master 广播通知各个 Worker 并发地 RunGraph。为了区分不同 step，Master 生成全局唯一的 step_id，并通过 RunGraph 传递给各个 Worker。

1. session_handle: 创建 MasterSession 实例时自动生成，通过 CreateSessionResponse 携带给 Client；通过 CreateWorkerSessionRequest 携带给 Worker；

2. graph_id: 首次 RegisterGraph 时由 Worker 生成，通过 RegisterGraphResponse 携带给 Master；

3. step_id: 每次 RunStep 时，由 Master 生成唯一的标识，通过 RunGraphRequest 携带给 Worker。

CreateWorkerSession

如图??（第??页）所示，CreateWorkerSessionRequest 消息中携带 MasterSession 分配的 session_handle。当 Worker 收到请求消息后，生成一个 WorkerSession 实例，并使用 session_handle 在该 Worker 内唯一地标识该实例。

在同一个集群中，对于一个 MasterSession 实例，其他 Worker 收到相同的 session_handle。如此，该 MasterSession 实例便能统一管理隶属于它的所有 WorkerSession 实例。

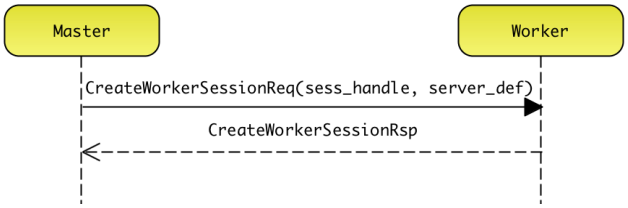


图 13-15 CreateWorkerSession

```
message CreateWorkerSessionRequest {
  string session_handle = 1;
  ServerDef server_def = 2;
}

message CreateWorkerSessionResponse {
}
```

RegisterGraph

如图??（第??页）所示，RegisterGraphRequest 消息中携带 MasterSession 分配的 session_handle，及其子图实例 graph_def。当 Worker 完成子图注册及其初始化后，向 Master 返回该子图的 graph_handle。

需要注意的是，Master 只会在执行一次 RegisterGraph，除非计算图的节点被重新编排，或者 Master 进程被重启。

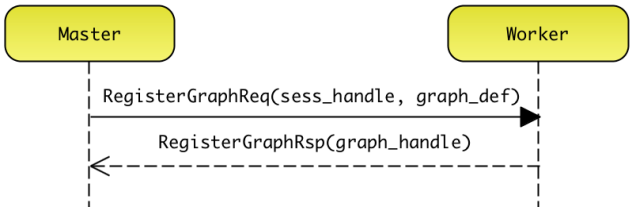


图 13-16 RegisterGraph

```
message RegisterGraphRequest {
  string session_handle = 1;

  GraphDef graph_def = 2;
  bool has_control_flow = 3 [deprecated = true];

  GraphOptions graph_options = 4;
  DebugOptions debug_options = 5;
}

message RegisterGraphResponse {
  string graph_handle = 1;
}
```

DeregisterGraph

如图??（第??页）所示，当 Worker 节点上的子图不再需要时 (例如，计算图被重新调度，图中节点被重新编排)，此时 Master 向 Worker 发送 DeregisterGraph 消息，以便 Worker 注销掉该子图实例。

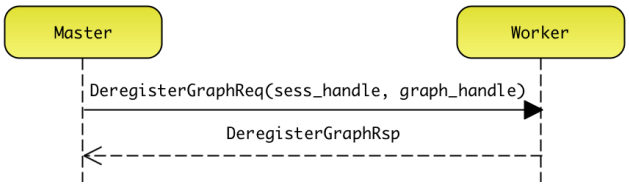


图 13-17 DeregisterGraph

```
message DeregisterGraphRequest {
  string session_handle = 2;
  string graph_handle = 1;
}

message DeregisterGraphResponse {
}
```

RunGraph

执行 Worker 节点上注册的子图时，为了区分不同 step，Master 生成唯一 step_id 并传递给各个 Worker，各个 Worker 通过 step_id 实现数据的协同。

此外，RunGraphRequest 携带了 send，recv_key，分别表示子图输入的 Tensor 标识和数据，及其子图输出的 Tensor 的标识。RunGraphResponse 返回 recv_key 相对应的 Tensor 列表。

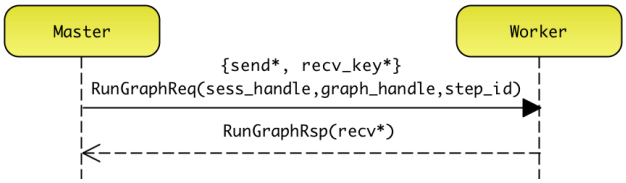


图 13-18 RunGraph

```
message RunGraphRequest {
  string session_handle = 8;
  string graph_handle = 1;
  int64 step_id = 2;

  ExecutorOpts exec_opts = 5;

  repeated NamedTensorProto send = 3;
  repeated string recv_key = 4;

  bool is_partial = 6;
  bool is_last_partial_run = 7;
}

message RunGraphResponse {
  repeated NamedTensorProto recv = 1;

  // execution stats
  StepStats step_stats = 2;
  CostGraphDef cost_graph = 3;
  repeated GraphDef partition_graph = 4;
}
```

RecvTensor

当执行某一次 step 中，如果两个 Worker 需要交互数据，消费者向生产者发送 RecvTensorRequest 消息，通过携带 (step_id, rendezvous_key) 二元组，请求对端 Worker 相应的 Tensor 数据，并通过 RecvTensorResponse 返回。

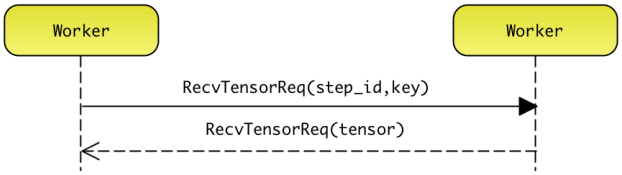


图 13-19 RecvTensor

```
message RecvTensorRequest {
  int64 step_id = 1;
  string rendezvous_key = 2;

  // If true, use an out-of-band DMA mechanism to transfer the
  // received tensor.
  bool dma_ok = 3;

  // Optional information on client-side device locality.
  DeviceLocality client_locality = 4;
```

```
  // Optional information on server-side device locality.
  DeviceLocality server_locality = 5;

  // Optional information needed by the RPC subsystem.
  google.protobuf.Any transport_options = 6;
}

message RecvTensorResponse {
  // The tensor as a proto.
  TensorProto tensor = 1;

  // If true, this tensor was the output of a dead node, and the
  // content is invalid.
  bool is_dead = 2;

  // The time at which tensor was available and started to be returned.
  int64 send_start_micros = 3;

  // Optional additional information about how to receive the tensor,
  // e.g. in the event that `RecvTensorRequest.dma_ok` was true.
  google.protobuf.Any transport_options = 4;
}
```

13.4 服务器

Server 是一个基于 gRPC 的服务器，负责管理本地设备集。它对外提供 MasterService 服务和 WorkerService 服务，具有同时扮演 Master 和 Worker 的角色。

领域模型

如图**??**（第**??**页）所示，GrpcServer 扮演 Master 的角色时，对外提供 MasterService 服务；其中，它为每一个接入的 Client 启动一个 MasterSession 实例，并使用全局唯一的 session_handle 标识它。也就是说，Master 可以接入多个 Client，而一个 Client 则只能接入一个特定的 Master。

GrpcServer 扮演 Worker 的角色时，对外提供 WorkerService 服务；其中，每个 Worker 可以为多个 Master 提供计算服务，它为每个向它请求计算服务的 MasterSession 生成一个相应的 WorkerSession 实例，等待相应的 MasterSession 下发计算图的**注册**和**执行**命令。

整个 GrpcServer 实例承载于 grpc::Server 进程之上，它监听特定端口的消息，当消息到达时自动派发到 MasterService 或 WorkerService 中相应的消息处理的回调函数。

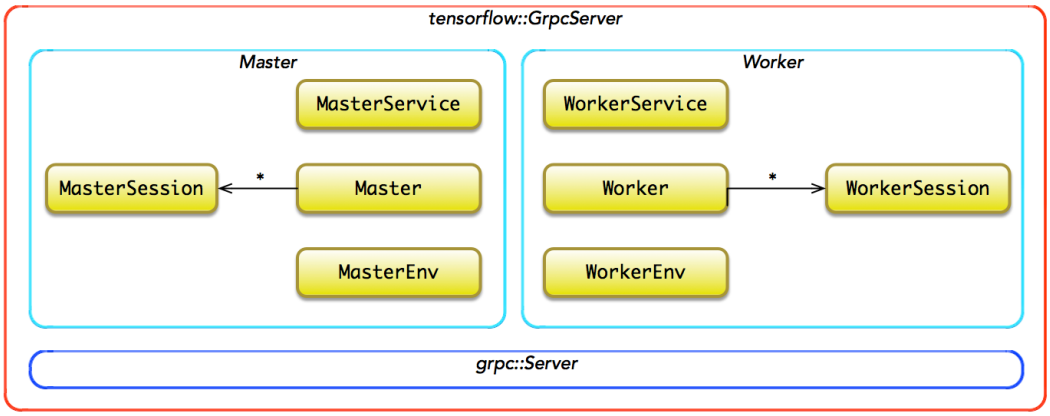


图 13-20 Server 领域模型

Protobuf 描述

当 protocol 为 grpc 时，系统运行时将启用基于 gRPC 实现的 GrpcServer 实例。此外，可以通过 ConfigProto 实现运行时参数的配置。也就是说，TensorFlow 的架构是对外开放的。例如，通过扩展 protocol 支持新的通信协议，实现基于新协议的 Server 实例。

```
message ServerDef {
  ClusterDef cluster = 1;

  string job_name = 2;
  int32 task_index = 3;

  ConfigProto default_session_config = 4;
  string protocol = 5;
}
```

服务互联

如图??（第??页）所示，一个 Server 实例通过 tf.train.ClusterSpec 与集群中的其他 Server 实例实现互联。

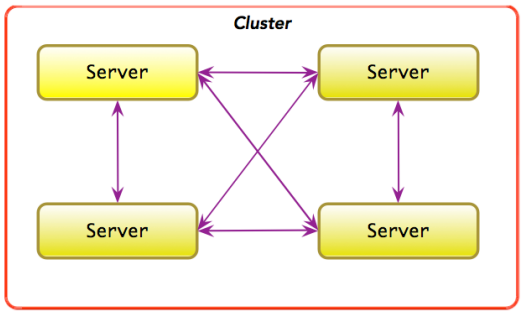


图 13-21 服务互联

如图??（第??页）所示，当 Client 接入其中一个 Server，此时它扮演了 Master 的角色，其他 Server 则扮演了 Worker 的角色。特殊地，Client 接入的 Server 也扮演了 Worker 的角色。

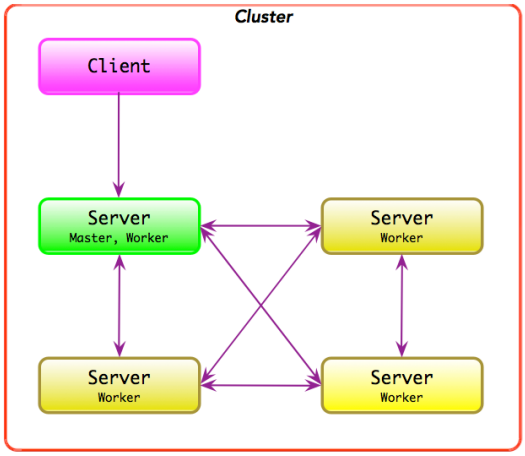


图 13-22 单 Client 接入集群

如图??（第??页）所示，可能存在多个 Client 分别接入不同的 Server 实例。此时，Client 接入的 Server 实例扮演了 Master 角色。但是，该 Server 实例，相对于集群中另外的 Server 实例，则扮演了 Worker 角色。

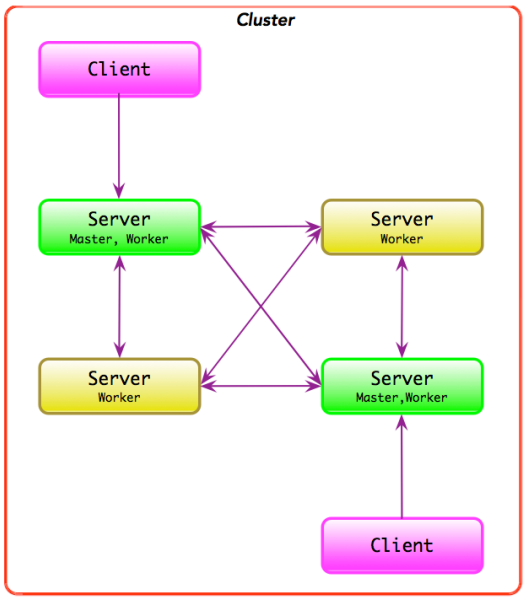


图 13-23 多 Client 接入集群

特殊地，Client 与 Master 可以部署在同一个进程内。此时，Client 与 Master 之间的交互更加简单，两者直接使用函数调用，避免了 gRPC 交互的额外开销。依次类推，在同一个 Server 内，Master 与 Worker 可以部署在同一进程内。此时，Master 与 Worker 之间直接使用函数调用。

状态机

如图??（第??页）所示，GrpcServer 是一个基于 grpc::Server 的服务器，它管理和维护了一个简单的状态机。

GrpcServer 在 New 状态上启动了 grpc::Server 服务，但对外并没有提供服务；而在 Started 状态上启动服务，对外提供 MasterService 和 WorkerService 的 RPC 消息服务；最终，在 Stopped 状态下停止 MasterService 和 WorkerService 服务。

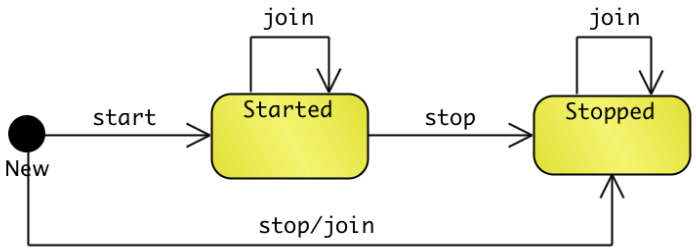


图 13-24 GrpcServer 状态机

创建服务

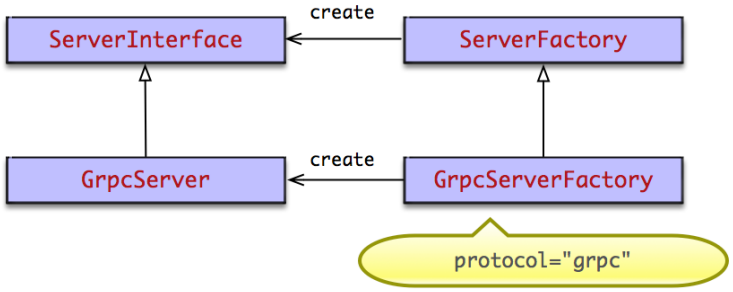


图 13-25 多态创建 Server 实例

```
struct GrpcServerFactory : ServerFactory {
  bool AcceptsOptions(const ServerDef& server_def) override {
    return server_def.protocol() == "grpc";
  }

  Status NewServer(const ServerDef& server_def,
      std::unique_ptr<ServerInterface>* out_server) override {
    GrpcServer::Create(server_def, Env::Default(), out_server);
    return Status::OK();
  }
};
```

```
void GrpcServer::Create(
    const ServerDef& server_def, Env* env,
    std::unique_ptr<ServerInterface>* out_server) {
  auto ret = std::make_unique<GrpcServer>(server_def, env);
  ret->Init();
  *out_server = std::move(ret);
}
```

如图**??**（第**??**页）所示，GrpcServer::Init 将完成 GrpcServer 领域对象的初始化，主要包括如下 3 个基本过程。

1. 初始化 MasterEnv 实例；
2. 初始化 WorkerEnv 实例；
3. 创建并启动 grpc::Server
   (a) 初始化 MasterService
      - 创建 Master 实例；
      - 创建 MasterService 实例；
   (b) 初始化 WorkerService
      - 创建 Worker 实例；
      - 创建 WorkerService 实例。

为了更好地理解整个 GrpcServer 实例的初始化过程，此处对实现做了局部的重构。首先，它初始化 MasterEnv, WorkerEnv 实例；然后，创建并启动 grpc::Server 服务器。

```
void GrpcServer::Init() {
  InitMasterEnv();
  InitWorkerEnv();
  StartGrpcServer();
}
```

初始化 MasterEnv

MasterEnv 持有 Master 运行时的上下文环境，它与 GrpcServer 具有相同的生命周期，因此整个 Master 的运行时都是可见的。

如图**??**（第**??**页）所示，LocalDevices 用于获取本地设备集；WorkerCacheFactory 用于创建 WorkerCacheInterface 实例；WorkerCacheInterface 用于创建 MasterInterface 实例，后者用于调用远端 MasterService 服务；MasterSessionFactory 用于创建 MasterSession 实例；OpRegisteryInterface 用于查询特定 OP 的元数据；Env 用于获取跨平台的 API 接口。其中，下文将重点讨论 WorkerCacheInterface 的创建过程。

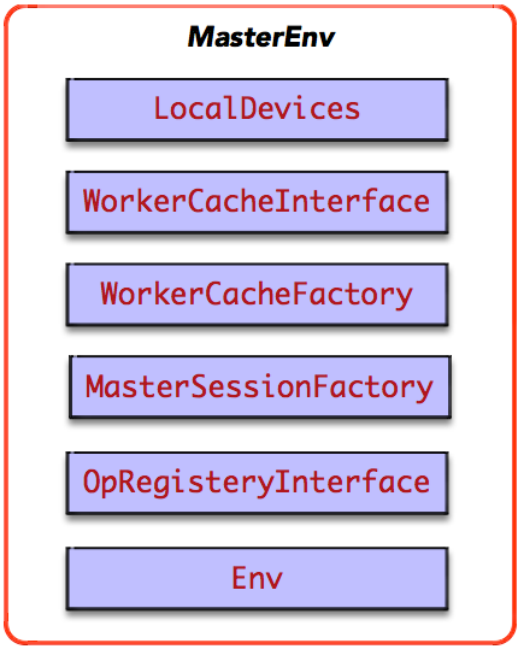


图 13-26 MasterEnv 模型

初始化 WorkerEnv

WorkerEnv 持有 Worker 运行时的上下文环境，它与 GrpcServer 具有相同的生命周期，因此整个 Worker 的运行时都是可见的。

如图??（第??页）所示，LocalDevices 用于获取本地设备集；DeviceManager 用于管理本地设备集和远端设备集；SessionManager 用于管理 WorkerSession 的集合；RendezvousManager 用于管理 Rendezvous 实例集；ThreadPool 将会自动从计算池中分配一个线程，启动 OP 的 Kernel 算子的执行；Env 用于获取跨平台的 API 接口。

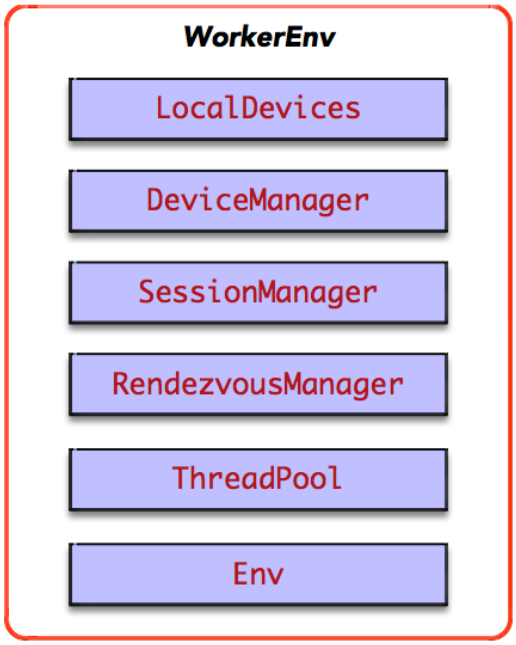


图 13-27 WorkerEnv 模型

启动 grpc::Server

系统实现采用构建器创建 grpc::Server 实例。首先，配置 grpc::Server 的服务选项；然后，分别构建 MasterService 实例和 WorkerService 实例。最后，调用 builder.BuildAndStart 方法启动 grpc::Server 服务器。

需要注意的是，grpc::Server 启动时，GrpcServer 依然处于 New 状态，grpc::Server 暂时还未对外提供 MasterService 服务和 WorkerService 服务。直至 GrpcServer 迁移至 Started 状态位置，grpc::Server 才真正对外提供 MasterService 服务和 WorkerService 服务。

```
void InitServerBuilder(::grpc::ServerBuilder& builder) {
  builder.AddListeningPort(
    strings::StrCat("0.0.0.0:", GetRequestedPort()),
    GetServerCredentials(server_def_), &bound_port_);
  builder.SetMaxMessageSize(std::numeric_limits<int32>::max());
  builder.SetOption(
      std::unique_ptr<::grpc::ServerBuilderOption>(new NoReusePortOption));
}

void GrpcServer::StartGrpcServer() {
  ::grpc::ServerBuilder builder;

  InitServerBuilder(builder);
  InitMasterService(builder);
  InitWorkerService(builder);

  server_ = builder.BuildAndStart();
}
```

很容易发现，grpc::Server 对外提供 MasterService 服务的实体是 GrpcMasterService 实例。当消息到达时，将自动回调 GrpcMasterService 实例中相应的消息处理函数。其中，在消息处理函数中，其业务逻辑的处理完全依托于 Master 的领域对象。

```
std::unique_ptr<Master> GrpcServer::CreateMaster(
    MasterEnv* master_env) {
  return std::make_unique<Master>(master_env);
}

AsyncServiceInterface* NewGrpcMasterService(
    Master* master, ::grpc::ServerBuilder* builder) {
  return new GrpcMasterService(master, builder);
}

void GrpcServer::InitMasterService() {
  master_impl_ = CreateMaster(&master_env_);
  master_service_ = NewGrpcMasterService(
      master_impl_.get(), &builder);
}
```

依次类推，grpc::Server 对外提供 WorkerService 服务的实体是 GrpcWorkerService 实例。当消息到达时，将自动回调 GrpcWorkerService 实例中相应的消息处理函数。其中，在消息处理函数中，其业务逻辑的处理完全依托于 GrpcWorker 的领域对象。

```
std::unique_ptr<GrpcWorker> NewGrpcWorker(WorkerEnv* env) {
  return std::unique_ptr<GrpcWorker>(new GrpcWorker(env));
}

AsyncServiceInterface* NewGrpcWorkerService(
    GrpcWorker* worker, ::grpc::ServerBuilder* builder) {
  return new GrpcWorkerService(worker, builder);
}

void GrpcServer::InitWorkerService(::grpc::ServerBuilder& builder) {
  worker_impl_ = NewGrpcWorker(&worker_env_);
  worker_service_ = NewGrpcWorkerService(
    worker_impl_.get(), &builder);
}
```

启动服务

在 New 状态，grpc::Server 已经启动，但暂时没有对外提供 MasterService 服务和 WorkerService 服务。通过调用 GrpcServer::Start 方法后，GrpcServer 的状态从 New 迁移 Started 状态，并启动了两个独立的线程，分别启动 MasterService 和 WorkerService 的消息处理器。此时，GrpcServer 正式对外提供 MasterService 和 WorkerService。

```
Status GrpcServer::Start() {
  mutex_lock l(mu_);
  switch (state_) {
    case NEW: {
      master_thread_.reset(
          env_->StartThread(ThreadOptions(), "TF_master_service",
```

```
                                [this] { master_service_->HandleRPCsLoop(); }));
        worker_thread_.reset(
            env_->StartThread(ThreadOptions(), "TF_worker_service",
                                [this] { worker_service_->HandleRPCsLoop(); }));
        state_ = STARTED;
        return Status::OK();
      }
      case STARTED:
        LOG(INFO) << "Server already started(" << target() << ")";
        return Status::OK();
      case STOPPED:
      default:
        CHECK(false);
    }
  }
```

等待终止服务

为了持久地对外提供 MasterService 服务和 WorkerService 服务，需要分别对线程 TF_master_service 和 TF_worker_service 实施 join 操作，使得主线程挂起，直至这两个线程终止。

通过调用 GrpcServer::Join 方法，当 GrpcServer 处于 Started 或 Stoped 状态时，它将自动调用 Thread 的析构函数。

```
 Status GrpcServer::Join() {
   mutex_lock l(mu_);
   switch (state_) {
     case NEW:
       // Prevent the server from being started subsequently.
       state_ = STOPPED;
       return Status::OK();
     case STARTED:
     case STOPPED:
       master_thread_.reset();
       worker_thread_.reset();
       return Status::OK();
     default:
       CHECK(false);
   }
 }
```

例如，基于 C++ 标准库实现的 StdThread 中，其析构函数将调用 std::thread 的 join 方法。

```
 struct StdThread : Thread {
   StdThread(const ThreadOptions&, const string&,
       std::function<void()> fn)
     : thread_(fn) {
   }

   ~StdThread() override {
     thread_.join();
   }

  private:
```

```
  std::thread thread_;
};
```

终止服务

遗憾的是，目前 GrpcServer 并不能优雅地退出。因此，在工程实践环境中，TensorFlow 的分布式运行时常常需要借助于 Kubernetes，实现 GrpcServer 服务的自动管理。

```
Status GrpcServer::Stop() {
  mutex_lock l(mu_);
  switch (state_) {
    case NEW:
      state_ = STOPPED;
      return Status::OK();
    case STARTED:
      return errors::Unimplemented(
          "Clean shutdown is not currently implemented");
    case STOPPED:
      LOG(INFO) << "Server already stopped(" << target() << ")";
      return Status::OK();
    default:
      CHECK(false);
  }
}
```

创建 WorkerCacheInterface

介绍完 GrpcServer 状态机模型之后，再回到之前遗留的一个问题。MasterEnv 持有 WorkerCacheInterface 实例，它用于查询或延迟创建 WorkerInterface；其中，WorkerInterface 用于访问远端 WorkerService 的服务。

工厂方法：GrpcServer::WorkerCacheFactory

在初始化 MasterEnv 时，通过调用工厂方法 GrpcServer::WorkerCacheFactory 创建 WorkerCacheInterface 实例。其中，WorkerCacheFactoryOptions 等价于 ServerDef，它包含 ClusterDef，及其 job_name:task_index 信息。因此，经过 ParseChannelSpec 得到的 GrpcChannelSpec 实例，等价于 ClusterSpec，它包含了集群的基本配置信息。

```
Status GrpcServer::WorkerCacheFactory(
    const WorkerCacheFactoryOptions& options,
    WorkerCacheInterface** worker_cache) {

  GrpcChannelSpec channel_spec;
  TF_RETURN_IF_ERROR(ParseChannelSpec(options, &channel_spec));

  std::unique_ptr<GrpcChannelCache> channel_cache(
      NewGrpcChannelCache(channel_spec, GetChannelCreationFunction()));
```

```
    string name_prefix = strings::StrCat(
        "/job:", *options.job_name, "/replica:0",
        "/task:", options.task_index);

    *worker_cache = NewGrpcWorkerCacheWithLocalWorker(
        channel_cache.release(), worker_impl_.get(), name_prefix);
    return Status::OK();
  }
```

工厂方法：NewGrpcChannelCache

NewGrpcChannelCache 用于创建 GrpcChannelCache 实例，GrpcChannelCache 可以根据 Worker 的名称，获取或延迟创建相应的 grpc::Channel 实例。其中，一个 Job 创建一个 SparseGrpcChannelCache 实例，MultiGrpcChannelCache 持有多个 SparseGrpcChannelCache，这是一种典型的组合模式的应用，下文将详细讲解 GrpcChannelCache 的设计。

```
  GrpcChannelCache* NewGrpcChannelCache(
      const GrpcChannelSpec& spec,
      ChannelCreationFunction channel_func) {
    std::vector<GrpcChannelCache*> caches;
    for (auto& job : spec.host_ports_jobs()) {
      caches.push_back(
          new SparseGrpcChannelCache(
              job.job_id, job.host_ports, channel_func));
    }
    return new MultiGrpcChannelCache(caches);
  }
```

工厂方法：NewGrpcWorkerCacheWithLocalWorker

而工厂方法 NewGrpcWorkerCacheWithLocalWorker，用于创建带有本地 Worker 的 GrpcWorkerCache 实例。

```
  WorkerCacheInterface* NewGrpcWorkerCacheWithLocalWorker(
      GrpcChannelCache* cc, WorkerInterface* local_worker,
      const string& local_target) {
    return new GrpcWorkerCache(cc, local_worker, local_target);
  }
```

工厂方法：GrpcServer::GetChannelCreationFunction

GetChannelCreationFunction 的设计使用了 C++ 函数式编程的思想，它返回了一个用于创建 grpc::Channel 实例的函数对象。但不幸的是，已存在的 NewHostPortGrpcChannel 函数与 ChannelCreationFunction 接口不匹配。因此，此处使用了名为 ConvertToChannelCreationFunction 的适配器，将 NewHostPortGrpcChannel 变换为 ChannelCreationFunction。

```
using SharedGrpcChannelPtr = std::shared_ptr<::grpc::Channel>;
using ChannelCreationFunction = std::function<SharedGrpcChannelPtr(string)>;

Status NewHostPortGrpcChannel(const string& target,
    SharedGrpcChannelPtr* channel) {
  ::grpc::ChannelArguments args;
  args.SetInt("grapc.arg.max.message_length",
              std::numeric_limits<int32>::max());
  args.SetInt("grpc.testing.fixed_reconnect_backoff_ms",
              1000);

  *channel = ::grpc::CreateCustomChannel(
      "dns:///" + target, ::grpc::InsecureChannelCredentials(), args);
  return Status::OK();
}

ChannelCreationFunction ConvertToChannelCreationFunction(
  const std::function<Status(string, SharedGrpcChannelPtr*)>& new_channel) {
  return [new_channel_func](const string& target) -> SharedGrpcChannelPtr {
    SharedGrpcChannelPtr channel_ptr;
    if (new_channel(target, &channel_ptr).ok()) {
      return channel_ptr;
    } else {
      return nullptr;
    }
  };
}

ChannelCreationFunction GrpcServer::GetChannelCreationFunction() const {
  return ConvertToChannelCreationFunction(NewHostPortGrpcChannel);
}
```

至此，我们理顺了 GrpcChannelCache 与 WorkerCacheInterface 的创建过程，但是它们是用来干什么呢？事实上，WorkerCacheInterface 用于获取 WorkerInterface 实例，后者用于访问远端 WorkerSerivice 服务的，其工作原理非常简单。

1. 获取集群中所有 Worker 的名字列表；
2. 根据 Worker 的名字创建 RPC 通道；
3. 根据 Worker 的 RPC 通道，创建 GrpcRemoteWorker 实例。

其中，GrpcRemoteWorker 是 WorkerInterface 的具体实现；GrpcChannelCache 负责获取 Worker 的名称，及其创建 Worker 相应的 grpc::Channel。

创建 Worker 的 RPC 通道

GrpcChannelCache 用于获取或创建集群中远端 Worker 的 RPC 通道。其中，ListWorkers 用于返回集群中 Worker 的名称列表。TranslateTask 用于将 Worker 的名称转换为 host:port 的地址信息。FindWorkerChannel 从缓存中查找 grpc::Channel 实例；如果未找到，则根据地址信息动态创建一个 grpc::Channel 实例，并添加至缓存中。

```
typedef std::shared_ptr<::grpc::Channel> SharedGrpcChannelPtr;

struct GrpcChannelCache {
  virtual ~GrpcChannelCache() {}
  virtual void ListWorkers(std::vector<string>* workers) const = 0;
  virtual SharedGrpcChannelPtr FindWorkerChannel(const string& target) = 0;
  virtual string TranslateTask(const string& task) = 0;
};
```

隐式树

如图??（第??页）所示，GrpcChannelCache 类层次结构遵循隐式的“树型”结构，SparseGrpcChannelCache 是树节点，其每个实例对应一个 Job 实例。而 MultiGrpcChannelCache 则持有多个 SparseGrpcChannelCache 实例，其每个实例对应多个 Job 实例。

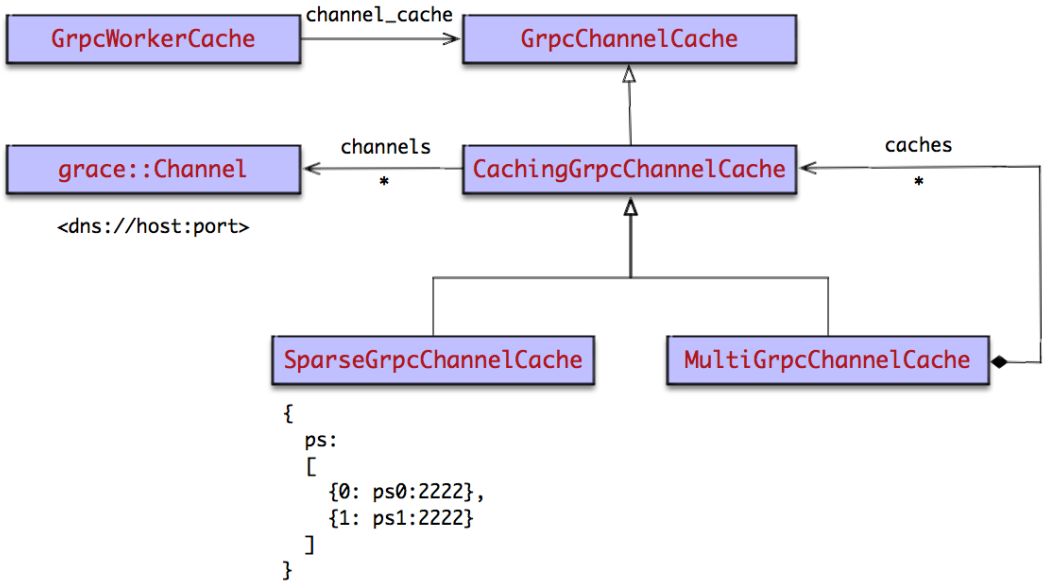


图 13-28 组合创建 GRPC 通道

缓存机制

为了避免每次实时创建 grpc::Channel 实例的开销，引入了 CachingGrpcChannelCache，它在查找 grpc::Channel 的过程使用了缓存技术。当缓存中查找失败时，通过调用 FindChannelOnce 延迟地动态创建 grpc::Channel 实例，并添加至缓存中。

```
struct CachingGrpcChannelCache : GrpcChannelCache {
  SharedGrpcChannelPtr FindWorkerChannel(const string& target) override {
    SharedGrpcChannelPtr ch = nullptr;
    {
      mutex_lock l(mu_);
      ch = gtl::FindPtrOrNull(channels_, target);
      if (ch) {
```

```
        return ch;
      }
    }
    ch = FindChannelOnce(target);
    if (ch) {
      mutex_lock l(mu_);
      channels_.insert({target, ch});
    }
    return ch;
  }

 protected:
  virtual SharedGrpcChannelPtr FindChannelOnce(const string& target) = 0;

 private:
  mutex mu_;
  std::unordered_map<string, SharedGrpcChannelPtr> channels_;
};
```

叶子节点

SparseGrpcChannelCache 的每个实例对应一个 Job 实例，它为某个 Job 创建相应的 grpc::Channel 实例集，每个 Task 对应一个 grpc::Channel。

其中，FindChannelOnce 通过调用 TranslateTask，从 Worker 名称中提取对应的 task_id，然后再从 host_ports_ 中索引出 host:port 的地址信息，以此地址调用工厂方法 channel_func_ 创建相应的 grpc::Channel 实例。因此，它主要负责如下三个职责：

1. 通过 ListWorkers 返回该 Job 对应的 Task 名称列表；例如，/job:ps 返回 [/job:ps/replica:0/task:0, /job:ps/replica:0/task:1]；
2. 通过 TranslateTask，并根据特定 Task 名称，索引 host:port 的地址信息；例如，/job:ps/replica:0/task:0 索引的地址为 ps0:2222；
3. 通过 FindChannelOnce，并根据特定 Task 名称，创建对应的 grpc::Channel 实例。例如，/job:ps/replica:0/task:0，创建以 ps0:2222 为地址的 grpc::Channel 实例。

```
static string MakeAddress(const string& job, int task) {
  return strings::StrCat("/job:", job, "/replica:0/task:", task);
}

struct SparseGrpcChannelCache : CachingGrpcChannelCache {
  SparseGrpcChannelCache(
      const string& job_id,
      const std::map<int, string>& host_ports,
      ChannelCreationFunction channel_func)
      : job_id_(job_id), host_ports_(host_ports),
        channel_func_(std::move(channel_func)) {
  }

  void ListWorkers(std::vector<string>* workers) const override {
    workers->reserve(workers->size() + host_ports_.size());
    for (const auto& id_host_port : host_ports_) {
      workers->emplace_back(MakeAddress(job_id_, id_host_port.first));
    }
  }
```

```
  string TranslateTask(const string& target) override {
    DeviceNameUtils::ParsedName parsed;
    if (!DeviceNameUtils::ParseFullName(target, &parsed)) {
      return "";
    }
    auto iter = host_ports_.find(parsed.task);
    return iter == host_ports_.end() ? "" : iter->second;
  }

 protected:
  SharedGrpcChannelPtr FindChannelOnce(const string& target) override {
    auto host_port = TranslateTask(target);
    if (host_port.empty()) {
      return nullptr;
    }
    return channel_func_(host_port);
  }

 private:
  const string job_id_;
  const std::map<int, string> host_ports_;
  const ChannelCreationFunction channel_func_;
};
```

非叶子节点

MultiGrpcChannelCache 通过 caches_ 持有多个 SparseGrpcChannelCache 实例，实现整个集群所有 Worker 节点的 grpc::Channel 的组合创建。为了进一步提高 SparseGrpcChannelCache 实例的查找过程，MultiGrpcChannelCache 缓存了已访问过的 SparseGrpcChannelCache 实例；只有当缓存中查找 SparseGrpcChannelCache 实例失败后，才尝试从 caches_ 列表中索引相应的 SparseGrpcChannelCache 实例，并自动添加至缓存中。

```
class MultiGrpcChannelCache : public CachingGrpcChannelCache {
 public:
  explicit MultiGrpcChannelCache(
      const std::vector<GrpcChannelCache*>& caches)
      : caches_(caches) {}

  ~MultiGrpcChannelCache() override {
    for (auto cache : caches_) {
      delete cache;
    }
  }

  void ListWorkers(std::vector<string>* workers) const override {
    for (auto cache : caches_) {
      cache->ListWorkers(workers);
    }
  }

  string TranslateTask(const string& target) override {
    mutex_lock l(mu_);  // could use reader lock
    auto cache = gtl::FindPtrOrNull(target_caches_, target);
    if (cache == nullptr) {
      for (auto c : caches_) {
        string r = c->TranslateTask(target);
        if (!r.empty()) {
          target_caches_.insert({target, c});
          cache = c;
          break;
```

```
        }
      }
    }
    return cache->TranslateTask(target);
  }

 protected:
  SharedGrpcChannelPtr FindChannelOnce(const string& target) override {
    for (auto cache : caches_) {
      auto ch = cache->FindWorkerChannel(target);
      if (ch) {
        mutex_lock l(mu_);
        target_caches_.insert({target, cache});
        return ch;
      }
    }
    return nullptr;
  }

 private:
  // List of channels used by this MultiGrpcChannelCache.
  const std::vector<GrpcChannelCache*> caches_;

  mutex mu_;
  // The same GrpcChannelCache can appear multiple times in the cache.
  std::unordered_map<string, GrpcChannelCache*> target_caches_;
};
```

创建 WorkerInterface

如图??（第??页）所示，GrpcWorkerCache 持有 GrpcChannelCache 对象，并通过它创建 grpc::Channel 实例，从而实现了 GrpcRemoteWorker 实例的动态创建。

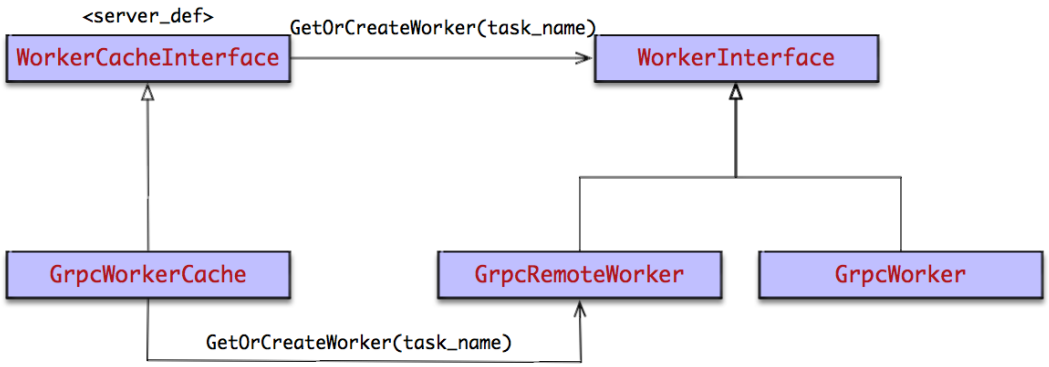


图 13-29 多态创建 WorkerInterface 实例

```
struct GrpcWorkerCache : WorkerCachePartial {
  GrpcWorkerCache(
      GrpcChannelCache* channel_cache,
      WorkerInterface* local_worker,
      const string& local_target)
      : local_target_(local_target),
        local_worker_(local_worker),
        channel_cache_(channel_cache) {}

  ~GrpcWorkerCache() override {
    live_rpc_counter_.WaitUntilUnused();
    delete channel_cache_;
```

```
    }

    void ListWorkers(std::vector<string>* workers) const override {
      channel_cache_->ListWorkers(workers);
    }

    WorkerInterface* CreateWorker(const string& target) override {
      if (target == local_target_) {
        return local_worker_;
      } else {
        auto channel = channel_cache_->FindWorkerChannel(target);
        if (!channel) return nullptr;
        return new GrpcRemoteWorker(&live_rpc_counter_, std::move(channel),
                                    &completion_queue_, &logger_);
      }
    }

    void ReleaseWorker(const string& target,
        WorkerInterface* worker) override {
      if (target != local_target_) {
        WorkerCacheInterface::ReleaseWorker(target, worker);
      }
    }

   private:
    string local_target_;
    WorkerInterface* local_worker_;  // Not owned.
    GrpcCounter live_rpc_counter_;
    GrpcChannelCache* channel_cache_;  // Owned.
    ::grpc::CompletionQueue completion_queue_;
    WorkerCacheLogger logger_;
  };
```

13.5 会话控制

会话控制是 TensorFlow 分布式运行时的核心，也是整个 TensorFlow 执行引擎的的关键路径。为了理顺会话控制的脉络，接下来的文章将重点讲述整个会话控制的详细过程。

会话协同

如图**??**（第**??**页）所示，在分布式模式中，会话控制通过 `GrpcSession`，`MasterSession`，`WorkerSession` 之间的协同实现的，它们分别驻留在 `Client`，`Master`，`Worker` 上，使用同一个 `session_handle` 实现协同工作的。

其中，`tf.Session` 使用 Python 实现，是 TensorFlow 对外提供的 API。它与 `GrpcSession` 在同一个进程内，并且直接持有 `GrpcSession` 的句柄 (或指针) 实现的。

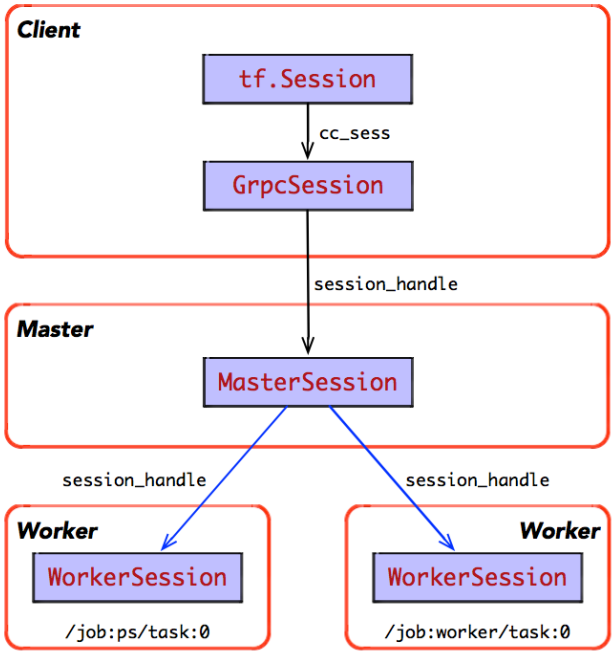


图 13-30 会话协同

如图??（第??页）所示，在分布式模式中，可能存在多个 Client 同时接入一个 Master，Master 为其每个接入的 Client 创建一个 `MasterSession` 实例。Worker 也可能同时为多个 Master 提供计算服务，Worker 为其每个请求计算的 Master 创建一个 `WorkerSession` 实例。为了区分不同的 Client 的计算服务，使用不同的 `session_handle` 区分。

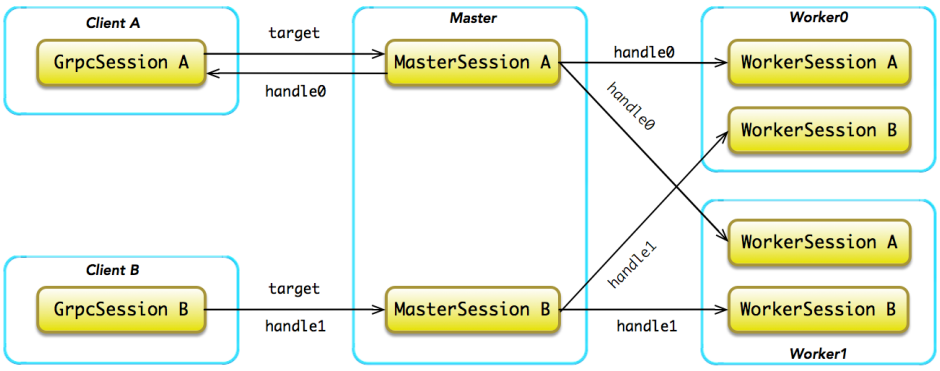


图 13-31 会话控制：领域模型

生命周期

`GrpcSession` 控制 Client 的会话生命周期，`MasterSession` 控制 Master 的会话生命周期，`WorkerSession` 控制 Worker 的会话生命周期，它们之间通过 `session_handle` 实现协同。

GrpcSession 生命周期

在分布式模式下，Client 的运行时由 GrpcSession 控制，GrpcSession 的生命周期过程如图??（第??页）所示。

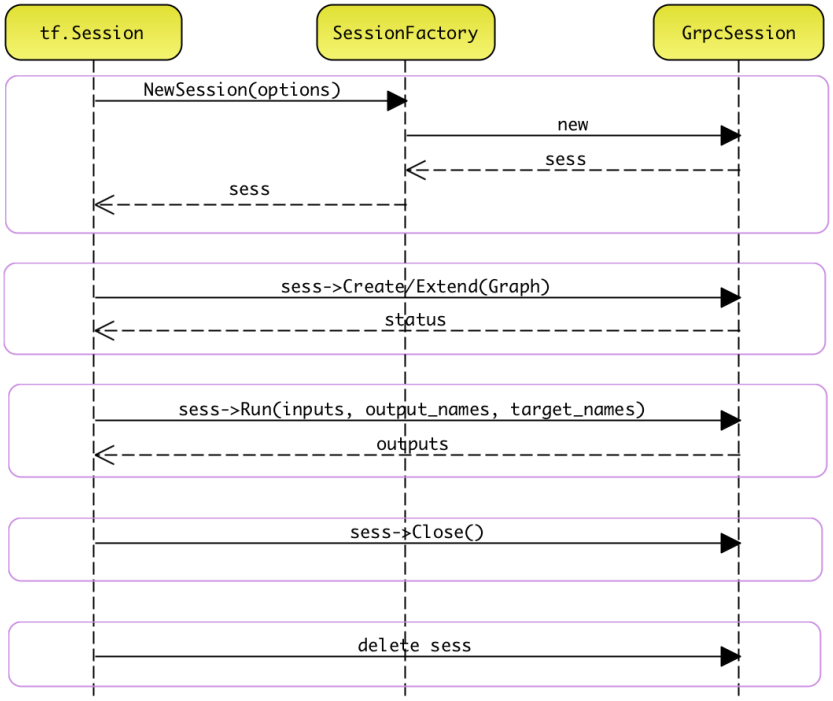


图 13-32 GrpcSession 生命周期

MasterSession 生命周期

在分布式模式下，Master 的运行时由 MasterSession 控制，MasterSession 生命周期过程如图??（第??页）所示。

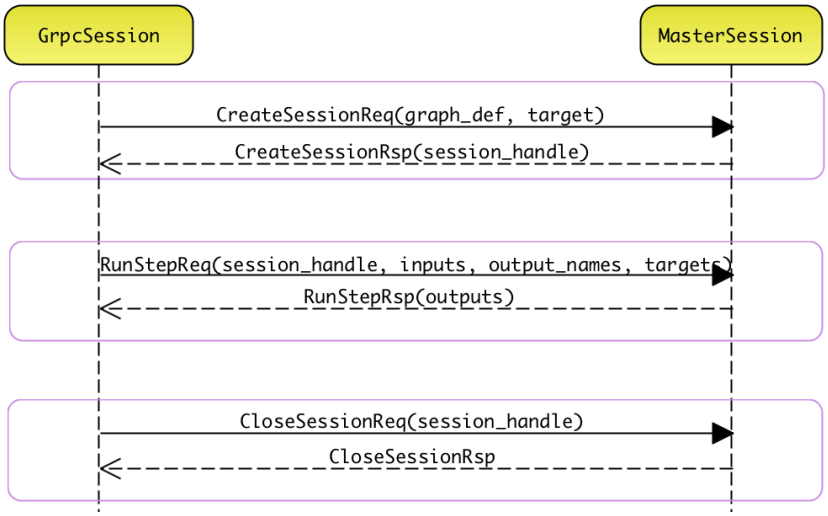


图 13-33 MasterSession 生命周期

WorkerSession 生命周期

在分布式模式下，Worker 的运行时由 WorkerSession 控制，WorkerSession 生命周期过程如图**??**（第**??**页）所示。

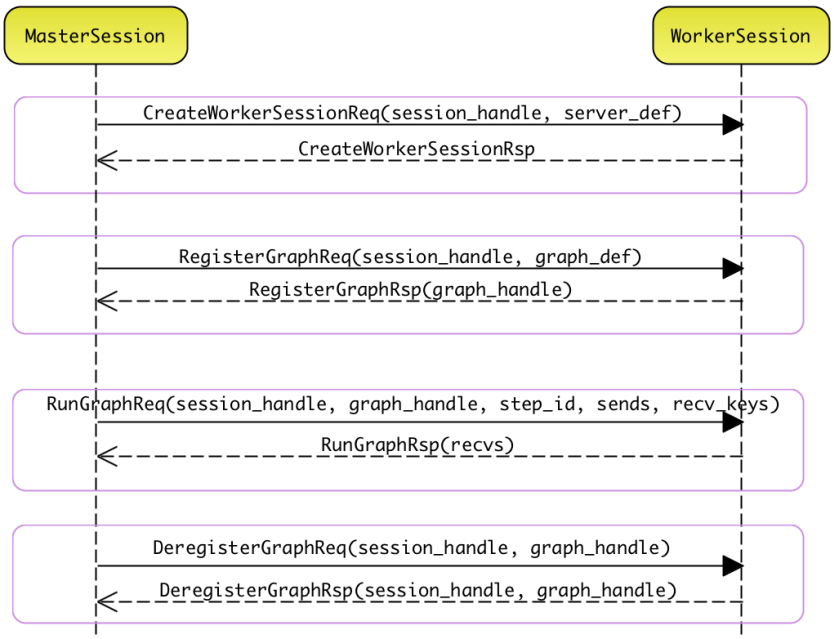


图 13-34 WorkerSession 生命周期

会话过程

在用户编程环境中，Client 从 tf.Session(target) 为起点，通过 Session.run 启动迭代执行，最终计算完成后调用 Session.close 关闭会话。但是，在分布式执行引擎的实现中，其过程要复杂得多。

- 创建会话
  1. 创建 GrpcSession；
  2. 获取远端设备集；
  3. 创建 MasterSession；
  4. 创建 WorkerSession；

- 迭代执行
  1. 启动执行；
  2. 图剪枝；
  3. 图分裂；
  4. 注册子图；
  5. 运行子图；

- 关闭会话
  1. 关闭 GrpcSession；
  2. 关闭 MasterSession；
  3. 关闭 WorkerSession；

13.6 创建会话

在启动计算之前，需要在 Client 端创建 GrpcSession 实例，在 Master 端创建 MasterSession 实例；在各个 Worker 上创建 WorkerSession 实例，三者通过 MasterSession 的 session_handle 实现协同，用于服务该接入的 Client 实例。

创建 GrpcSession

当 Client 调用 tf.Session(target) 时，通过调用 TF_NewDeprecatedSession 的 C API 接口，触发创建 GrpcSession 实例。其中，C API 是 TensorFlow 后端系统对外提供多语言

编程的标准接口。最终，tf.Session 将直接持有 GrpcSession 的句柄，如图??（第??页）所示。

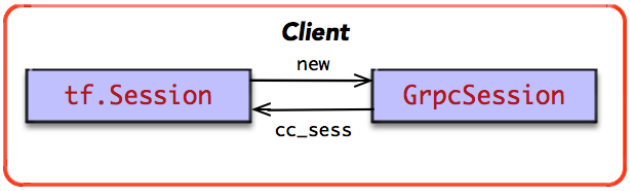


图 13-35 创建 GrpcSession: tf.Session 持有 GrpcSession 句柄

```
Status NewSession(const SessionOptions& options, Session** out_session) {
  SessionFactory* factory;
  Status s = SessionFactory::GetFactory(options, &factory);
  if (!s.ok()) {
    *out_session = nullptr;
    return s;
  }
  *out_session = factory->NewSession(options);
  if (!*out_session) {
    return errors::Internal("Failed to create session.");
  }
  return Status::OK();
}

TF_DeprecatedSession* TF_NewDeprecatedSession(
  const TF_SessionOptions* opt, TF_Status* status) {
  Session* session;
  status->status = NewSession(opt->options, &session);
  if (status->status.ok()) {
    return new TF_DeprecatedSession({session});
  } else {
    return nullptr;
  }
}
```

如图??（第??页）所示，GrpcSession 由 GrpcSessionFactory 多态创建。当 target 以 grpc://开头，则 SessionFactory::GetFactory 返回 GrpcSessionFactory 实例，而 GrpcSessionFactory::NewSession 的工厂方法委托 GrpcSession::Create 的静态工厂方法创建 GrpcSession 实例。

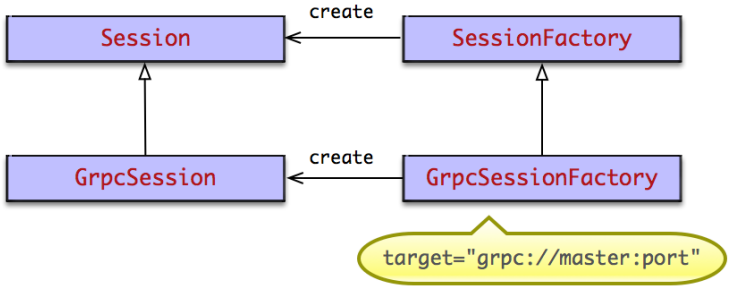


图 13-36 多态创建 GrpcSession

```
const char* kSchemePrefix = "grpc://";

struct GrpcSessionFactory : SessionFactory {
  bool AcceptsOptions(const SessionOptions& options) override {
    return StringPiece(options.target).starts_with(kSchemePrefix);
  }

  Session* NewSession(const SessionOptions& options) override {
    std::unique_ptr<GrpcSession> ret;
    Status s = GrpcSession::Create(options, &ret);
    if (s.ok()) {
      return ret.release();
    } else {
      return nullptr;
    }
  }
};
```

GrpcSession::Create 静态工厂方法主要负责创建 GrpcSession 实例,并完成相应的初始化工作。在初始化过程中,最重要的就是构建 MasterInterface 实例; 其中,MasterInterface 用于 Client 访问 Master 上的 MasterService 远端服务，它存在两种子类实现，分别对应两种不同应用场景:

1. LocalMaster: Client 与 Master 在同一进程内，调用 LocalMaster::Lookup 直接获取 LocalMaster 实例;

2. GrpcRemoteMaster: Client 与 Master 不在同一进程内，调用工厂方法 NewGrpcMaster 生成 GrpcRemoteMaster 实例。

GrpcRemoteMaster 实例是一个 RPC 的客户端实现，创建 GrpcRemoteMaster 实例时，需要先根据 target 指定的 Master 地址和服务端口，创建与之相连的 RPC 通道。

```
Status GrpcSession::Create(
    const SessionOptions& options,
    std::unique_ptr<GrpcSession>* out_session) {
  std::unique_ptr<GrpcSession> session(new GrpcSession(options));
  std::unique_ptr<MasterInterface> master;
  // intra-process between client and master.
  if (!options.config.rpc_options().use_rpc_for_inprocess_master()) {
    master = LocalMaster::Lookup(options.target);
  }
  // inter-process between client and master.
  if (!master) {
    SharedGrpcChannelPtr master_channel;
    TF_RETURN_IF_ERROR(NewHostPortGrpcChannel(
        options.target.substr(strlen(kSchemePrefix)), &master_channel));
    master.reset(NewGrpcMaster(master_channel));
  }
  session->SetRemoteMaster(std::move(master));
  *out_session = std::move(session);
  return Status::OK();
}
```

创建 MasterSession

如图??（第??页）所示，当 GrpcSession 实例创建成功后，随后将触发 GprcSession::Create 的调用，将初始的计算图通过 CreateSessionRequst 消息发送给 Master；当 Master 收到 CreateSessionRequst 消息后，生成相对应的 MasterSession 实例，并使用全局唯一的 session_handle 标识该实例，最终通过 CreateSessionResponse 消息带回给 GrpcSession。

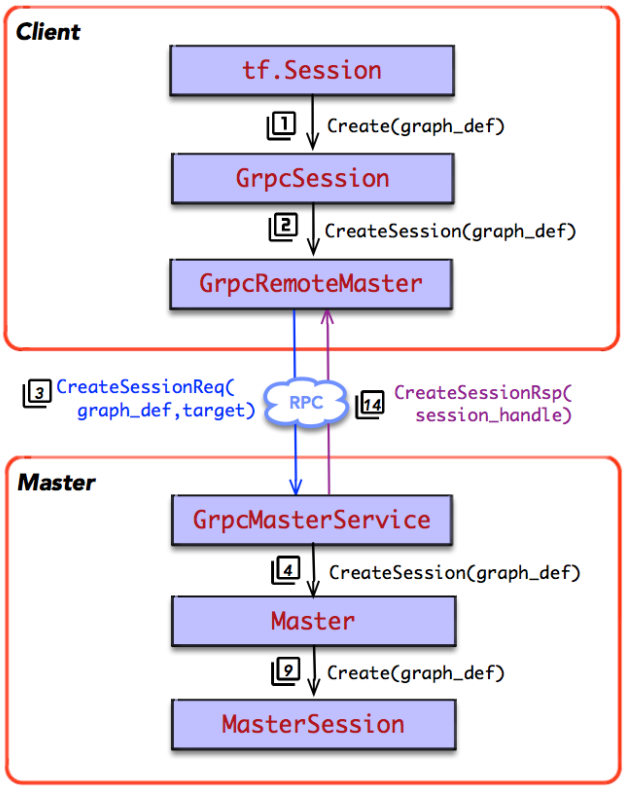


图 13-37 创建 MasterSession

GrpcSesion::Create(graph_def)

GrpcSession::Create(graph_def) 方法主要用于 Client 请求 Master 创建 MasterSession 实例。首先，GrpcSession::Create 方法完成构造 CreateSessionRequst 消息，然后通过 GrpcRemoteMaster 将其发送给 Master。

当 GrpcSession 收到 CreateSessionResponse 消息后，保存 MasterSession 的 handle，及其初始计算图的版本号 graph_version。其中，handle 用于标识 Master 侧的 MasterSession 实例，graph_version 用于后续扩展计算图使用。

```
void GrpcSession::BuildCreateSessionReq(
    const GraphDef& graph,
    CreateSessionRequest& req) {
  *req.mutable_config() = options_.config;
  *req.mutable_graph_def() = graph;
  req.set_target(options_.target);
}

void GrpcSession::SaveCreateSessionRsp(
    CreateSessionResponse& rsp) {
  mutex_lock l(mu_);
  swap(handle_, *(resp.mutable_session_handle()));
  current_graph_version_ = resp.graph_version();
}

Status GrpcSession::CreateImpl(CallOptions* call_options,
                               const GraphDef& graph) {
  CreateSessionRequest req;
  CreateSessionResponse resp;

  BuildCreateSessionReq(graph, req);
  Status s = master_->CreateSession(call_options, &req, &resp);
  if (s.ok()) {
    SaveCreateSessionRsp(resp);
  }
  return s;
}

Status GrpcSession::Create(const RunOptions& run_options,
                           const GraphDef& graph) {
  CallOptions call_options;
  call_options.SetTimeout(run_options.timeout_in_ms());
  return CreateImpl(&call_options, graph);
}

Status GrpcSession::Create(const GraphDef& graph) {
  CallOptions call_options;
  call_options.SetTimeout(options_.config.operation_timeout_in_ms());
  return CreateImpl(&call_options, graph);
}
```

GrpcRemoteMaster::CreateSession

`GrpcRemoteMaster` 是一个 gRPC 的客户端实现。它的实现非常简单，通过 gRPC 的一个 `stub` 调用远端 Master 相应的服务接口。

```
Status GrpcRemoteMaster::CreateSession(
    CallOptions* call_options,
    const CreateSessionRequest* request,
    CreateSessionResponse* response) override {
  ::grpc::ClientContext ctx;
  SetClientContext(*call_options, ctx);
  return FromGrpcStatus(stub_->CreateSession(&ctx, *request, response));
}
```

GrpcMasterService::CreateSessionHandler

GrpcMasterService 是一个 gRPC 服务，它实现了 MasterService 的 RPC 服务接口。当收到 CreateSession 消息后，将由 GrpcMasterService::CreateSessionHandler 回调处理该消息，它将委托 Master 处理该消息。

当 Master 处理完成后，将回调完成时的 lambda 表达式，向 Client 返回 CreateSessionResponse 的响应消息。

```
void GrpcMasterService::CreateSessionHandler(
  MasterCall<CreateSessionRequest, CreateSessionResponse>* call) {
  master_impl_->CreateSession(
    &call->request, &call->response,
    [call](const Status& status) {
        call->SendResponse(ToGrpcStatus(status));
    });
  ENQUEUE_REQUEST(CreateSession, true);
}
```

Master::CreateSession

Master::CreateSession 将会在线程池里启动一个线程，并在线程内按照 cluster_spec 信息，寻找所有的 Worker，收集远端设备集的信息。最后，创建了一个 MasterSession。

R 查找远端设备集的过程，待下一节详细说明，本节示例代码中略去此部分代码的实现。

当 MasterSession 创建成功后，Master 会保存 (handle, master_session) 的二元组信息，以便后续 Master 能够通过 handle 索引相应的 MasterSession 实例。

```
using RemoveDevices = unique_ptr<vector<unique_ptr<Device>>>;

void Master::CreateSession(const CreateSessionRequest* req,
                           CreateSessionResponse* resp, MyClosure done) {
  SchedClosure([this, req, resp, done]() {
    // 1. Find all remote devices.
    WorkerCacheInterface* worker_cache = env_->worker_cache;
    RemoveDevices remote_devices(new vector<unique_ptr<Device>>());

    Status status = DeviceFinder::GetRemoteDevices(
        req->config().device_filters(), env_,
        worker_cache, remote_devices.get())

    if (!status.ok()) return;

    // 2. Build DeviceSet
    std::unique_ptr<DeviceSet> device_set(new DeviceSet);
    for (auto&& d : *remote_devices) {
      device_set->AddDevice(d.get());
    }

    int num_local_devices = 0;
    for (Device* d : env_->local_devices) {
```

```
        device_set->AddDevice(d);
        if (num_local_devices == 0) {
          // Uses the first local device as the client device.
          device_set->set_client_device(d);
        }
        num_local_devices++;
      }

      // 3. Create MasterSession
      SessionOptions options;
      options.config = req->config();

      MasterSession* session = env_->master_session_factory(
          options, env_, std::move(remote_devices),
          std::move(worker_cache_ptr), std::move(device_set));

      GraphDef* gdef =
          const_cast<CreateSessionRequest*>(req)->mutable_graph_def();

      // ignore worker_cache_factory_options implements.
      WorkerCacheFactoryOptions worker_cache_factory_options;
      Status status = session->Create(gdef, worker_cache_factory_options);
      resp->set_session_handle(session->handle());

      // 4. Store <handle, master_session> pair.
      {
        mutex_lock l(mu_);
        CHECK(sessions_.insert({session->handle(), session}).second);
      }
    });
  }
```

MasterSession::Create(graph_def)

`MasterSession::Create(graph_def)` 主要完成两件事情。

1. 初始化计算图，并生成 `SimpleGraphExecutionState` 实例；
2. 如果动态配置集群，则广播所有 Worker 创建相应的 `WorkerSession` 实例。

其中，`SimpleGraphExecutionState::MakeForBaseGraph` 实现与本地模式相同，在此不再重述。

```
 Status MasterSession::Create(
     GraphDef* graph_def,
     const WorkerCacheFactoryOptions& options) {
   SimpleGraphExecutionStateOptions execution_options;
   execution_options.device_set = devices_.get();
   execution_options.session_options = &session_opts_;
   {
     mutex_lock l(mu_);
     TF_RETURN_IF_ERROR(SimpleGraphExecutionState::MakeForBaseGraph(
         graph_def, execution_options, &execution_state_));
   }

   // CreateWorkerSessions should be called only with
   // dynamic cluster membership.
   if (options.cluster_def != nullptr) {
     return CreateWorkerSessions(options);
   }
   return Status::OK();
```

```
}
```

获取远端设备集

如图??（第??页）所示，在创建 MasterSession 之前，MasterSession 会轮询所有的 Worker 实例，获取远端所有 Worker 的设备信息。借助于 DeviceFinder 的设备查找器，调用 DeviceFinder::GetRemoteDevices 获取远端设备集。

它的工作原理非常简单，它根据 GrpcWorkerCache::ListWorkers 获取集群中所有的 Worker 的名字列表；然后，根据 worker_name 的名称，调用 GrpcWorkerCache::CreateWorker 工厂方法创建 WorkerInterface 实例，后者用于访问远端 WorkerService 服务。最后，通过 WorkerInterface 向远端 Worker 列表广播发送 GetStatusRequest 请求消息，从而实现远端设备集的获取。

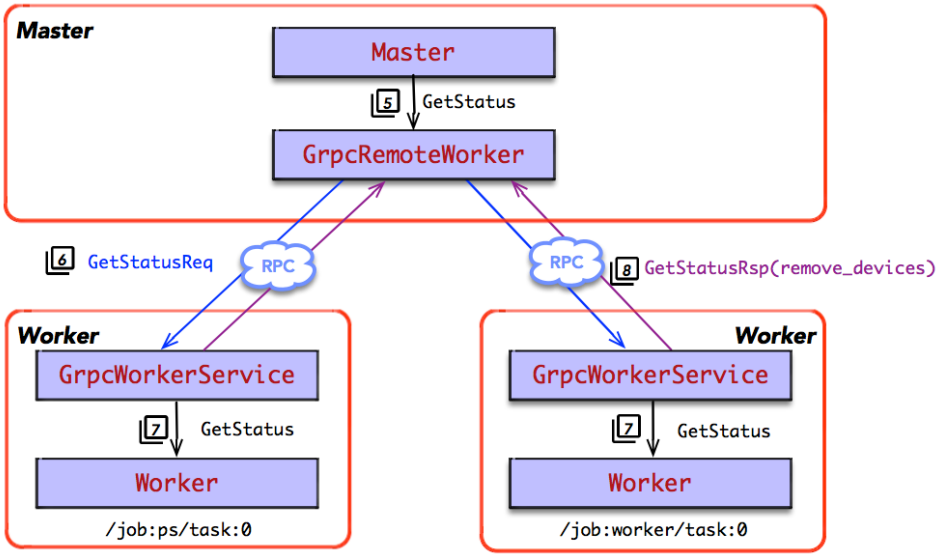


图 13-38 获取远端设备集

设备查找器

DeviceFinder 实现了一个函数对象，其实现了远端设备查找的算法，其过程分为三个步骤：

1. Start：并发地广播 GetStatusRequest 给集群中所有 Worker 实例；
2. Wait：收齐所有 Worker 返回的 GetStatusResponse 消息；
3. GetRemoteDevices：获取查询结果，并将其返回给客户；

```
struct DeviceFinder {
  static Status DeviceFinder::GetRemoteDevices(
      MasterEnv* env,
      WorkerCacheInterface* worker_cache,
      std::vector<std::unique_ptr<Device>>* out_remote) {
    DeviceFinder finder(env, worker_cache);
    finder.Start();
    TF_RETURN_IF_ERROR(finder.Wait());
    finder.GetRemoteDevices(env->local_devices, out_remote);
    return Status::OK();
  }
};
```

为了控制收齐多个 Worker 的 GetStatusResponse 消息，此处使用了 num_pending_ 的计数器，由 DeviceFinder::Start 设置初始值为 Worker 的数目。

当收到来自某个 Worker 的 GetStatusResponse 消息，则回调 WhenDone，将计数器减 1。当计数器减至为 0 时，通过调用 pending_zero_.notify_all，呼醒 pending_zero_.wait_for 语句，随后便可以通过 finder.GetRemoteDevices 获取查询结果了。

其中，在 DeviceFinder::Start 中，通过 NewRemoteDevices 向所有 Worker 广播 GetStatusRequest 消息查询相应的设备信息，下一节将重点描述其实现过程。

```
struct DeviceFinder {
 private:
  explicit DeviceFinder(
      MasterEnv* env,
      WorkerCacheInterface* worker_cache)
      : env_(env), worker_cache_(worker_cache) {
    worker_cache->ListWorkers(&targets_);
    seen_targets_.assign(targets_.size(), false);
  }

  ~DeviceFinder() {
    for (auto dev : found_) delete dev;
  }

  void Start() {
    {
      mutex_lock l(mu_);
      num_pending_ = targets_.size();
    }

    // Talk to all workers to get the list of available devices.
    using std::placeholders::_1;
    using std::placeholders::_2;
    for (size_t i = 0; i < targets_.size(); ++i) {
      NewRemoteDevices(env_->env, worker_cache_, targets_[i],
                       std::bind(&ME::WhenFound, this, i, _1, _2));
    }
  }

  // The caller takes the ownership of returned remote devices.
  void GetRemoteDevices(
      const std::vector<Device*>& local,
      std::vector<std::unique_ptr<Device>>* remote) {
    std::unordered_set<string> names(local.size());
    for (auto dev : local) {
      names.insert(dev->name());
    }

    mutex_lock l(mu_);
```

```
      for (auto dev : found_) {
        auto& name = dev->name();
        if (names.insert(name).second) {
          remote->push_back(std::unique_ptr<Device>(dev));
        } else {
          delete dev;
        }
      }
      found_.clear();
    }

    Status Wait() {
      mutex_lock l(mu_);
      while (num_pending_ != 0) {
        pending_zero_.wait_for(l, std::chrono::milliseconds(10 * 1000));
        if (num_pending_ != 0) {
          for (size_t i = 0; i < targets_.size(); ++i) {
            if (!seen_targets_[i]) {
              LOG(INFO)
                  << "CreateSession still waiting for response from worker: "
                  << targets_[i];
            }
          }
        }
      }
      return status_;
    }

    void WhenFound(int target_index, const Status& s,
                   std::vector<Device*>* devices) {
      mutex_lock l(mu_);
      seen_targets_[target_index] = true;
      if (!s.ok()) {
        status_.Update(s);
      } else {
        found_.insert(found_.end(), devices->begin(), devices->end());
        devices->clear();
      }
      --num_pending_;
      if (num_pending_ == 0) {
        pending_zero_.notify_all();
      }
    }

    typedef DeviceFinder ME;
    const MasterEnv* env_;
    WorkerCacheInterface* worker_cache_;

    mutex mu_;
    int num_pending_ GUARDED_BY(mu_);
    condition_variable pending_zero_;
    std::vector<Device*> found_ GUARDED_BY(mu_);

    std::vector<string> targets_;
    std::vector<bool> seen_targets_ GUARDED_BY(mu_);
    Status status_;
  };
```

温馨提示，当 num_pending_ 计数器不为零时，则主线程周期性睡眠 10 秒钟，醒来时如果发现存在 Worker 还未返回响应消息，则打印那些 Worker 的名称。当看到循环地打印如下信息时，此时应该明白/job:worker/task:2 对应的 Server 是否异常退出了，或 Master 与之相应的 Worker 之间网络是否发生了异常等，根据具体情况分析和处理。

```
CreateSession still waiting for response from worker: /job:worker/task:2
```

NewRemoteDevices

NewRemoteDevices 将根据 worker_name 查找 WorkerInterface 实例，并发送 GetStatusRequest 消息到对应的 Worker 获取其设备信息。当消息返回后，将回调 cb 的函数对象。其中，从远端 Worker 获取的设备信息并非完整，其不包含 worker_name 的信息，因此需要手动地追加上去。

```
void NewRemoteDevices(
    Env* env, WorkerCacheInterface* worker_cache,
    const string& worker_name, NewRemoteDevicesDone done) {
  struct Call {
    GetStatusRequest req;
    GetStatusResponse resp;
  };

  WorkerInterface* wi = worker_cache->CreateWorker(worker_name);
  Call* call = new Call;
  auto cb = [env, worker_cache, &worker_name, &done, wi, call](
      const Status& status) {
    Status s = status;
    std::vector<Device*> remote_devices;
    auto cleanup = gtl::MakeCleanup(
        [worker_cache, &worker_name, wi, &done, &remote_devices, &s, call] {
          worker_cache->ReleaseWorker(worker_name, wi);
          done(s, &remote_devices);
          delete call;
        });
    if (s.ok()) {
      DeviceNameUtils::ParsedName worker_name_parsed;
      DeviceNameUtils::ParseFullName(worker_name, &worker_name_parsed);

      remote_devices.reserve(call->resp.device_attributes_size());

      for (auto& da : call->resp.device_attributes()) {
        DeviceNameUtils::ParsedName device_name_parsed;
        DeviceNameUtils::ParseFullName(da.name(), &device_name_parsed);

        DeviceAttributes da_rewritten = da;
        da_rewritten.set_name(DeviceNameUtils::FullName(
            worker_name_parsed.job, worker_name_parsed.replica,
            worker_name_parsed.task, device_name_parsed.type,
            device_name_parsed.id));
        auto d = new RemoteDevice(env, da_rewritten);
        remote_devices.push_back(d);
      }
    }
  };
  wi->GetStatusAsync(&call->req, &call->resp, cb);
}
```

GrpcRemoteWorker::GetStatusAsync

GrpcRemoteWorker 是 WorkerInterface 的具体实现，它是 gRPC 的客户端实现，它通

过 stub 调用远端 WorkerService 相应的服务接口。

```
struct GrpcRemoteWorker : WorkerInterface {
  void GetStatusAsync(const GetStatusRequest* request,
                      GetStatusResponse* response,
                      StatusCallback done) override {
    IssueRequest(request, response, getstatus_, std::move(done));
  }
}
```

GrpcRemoteWorker::GetStatusAsync

GrpcWorkerService 是 WorkerService 的具体实现。当收到 GetStatusRequest 消息，将由 GetStatusHandler 回调处理。

```
struct GrpcWorkerService : AsyncServiceInterface {
  void GetStatusHandler(WorkerCall<GetStatusRequest, GetStatusResponse>* \
    call) {
    Schedule([this, call]() {
      Status s = worker_->GetStatus(&call->request, &call->response);
      call->SendResponse(ToGrpcStatus(s));
    });
    ENQUEUE_REQUEST(GetStatus, false);
  }
};
```

Worker::GetStatusAsync

Worker::GetStatusAsync 将委托 DeviceMgr 实现本地设备信息的汇总，并最终通过 GetStatusResponse 消息返回给对端。

```
void Worker::GetStatusAsync(const GetStatusRequest* request,
                            GetStatusResponse* response, StatusCallback \
                              done) {
  std::vector<DeviceAttributes> devices;
  env_->device_mgr->ListDeviceAttributes(&devices);
  response->mutable_device_attributes()->Reserve(devices.size());
  for (auto& d : devices) {
    response->add_device_attributes()->Swap(&d);
  }
  done(Status::OK());
}
```

DeviceMgr 持有本地设备集，实现极为简单。

```
void DeviceMgr::ListDeviceAttributes(
    std::vector<DeviceAttributes>* devices) const {
  devices->reserve(devices_.size());
  for (auto dev : devices_) {
```

```
      devices->emplace_back(dev->attributes());
    }
  }
```

创建 WorkerSession

当 MasterSession 创建成功后，如果没有动态配置集群 (默认的分布式配置环境)，则不会广播所有 Worker 动态地创建 WorkerSession。事实上，每个 Worker 都存在一个 SessionMgr 实例，它持有一个名为 legacy_session_ 的 WorkerSession 实例。因此，每个 Worker 存在一个全局唯一的 WorkerSession 实例。

```
SessionMgr::SessionMgr(
    WorkerEnv* worker_env,
    const string& default_worker_name,
    std::unique_ptr<WorkerCacheInterface> default_worker_cache,
    WorkerCacheFactory worker_cache_factory)
    : worker_env_(worker_env),
      legacy_session_(
          default_worker_name,
          std::move(default_worker_cache),
          std::unique_ptr<DeviceMgr>(worker_env->device_mgr),
          std::unique_ptr<GraphMgr>(
              new GraphMgr(worker_env,
              worker_env->device_mgr))),
      worker_cache_factory_(std::move(worker_cache_factory)) {}
```

如图**??**（第**??**页）所示，如果存在动态集群配置，则 Master 广播每个 Worker 各自创建一个 WorkerSession 实例，并且使用 sessin_handle 标识该 WorkerSession。这些 WorkerSession 隶属于此 MasterSession 实例，因为它们使用与 MasterSession 实例相同的 session_handle 标识。

其中，MasterSession 为了收齐所有 Worker 返回的 CreateWorkerSessionResponse 消息，引入了 BlockingCounter 计数器。BlockingCounter 计数器初始值为 Worker 的数目，当收到每个 Worker 的响应消息，则对计数器减 1，直至计数器为 0，done.Wait() 被呼醒。

此外，WorkerInterface 实例是通过 WorkerCacheInterface 查询或创建得到的，后文将详细讲解其工作原理。

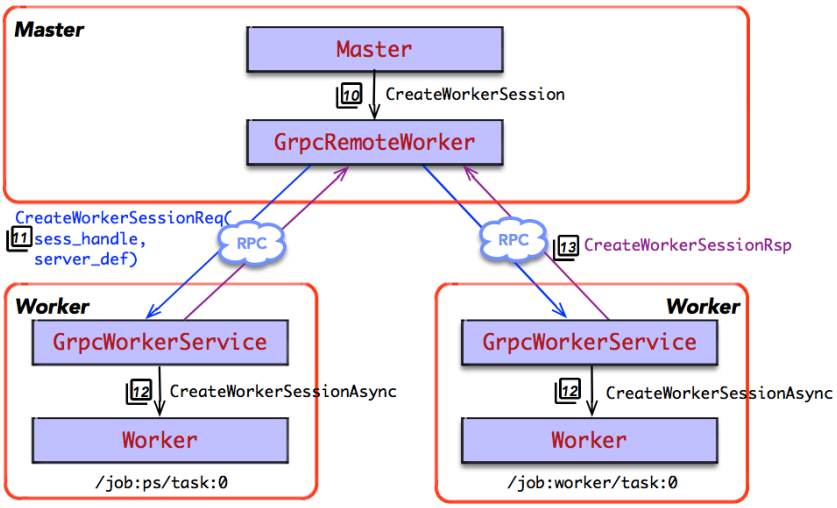


图 13-39 动态创建 WorkerSession

```
struct MasterSession::Worker {
  Worker(MasterSession* sess, const string& name,
         const DeviceNameUtils::ParsedName& parsed_name,
         const WorkerCacheFactoryOptions& opts)
      : sess(sess), name(&name), worker(GetOrCreateWorker()) {
    BuildRequest(parsed_name, opts);
  }

  void CreateWorkerSession(BlockingCounter& done, Status& status) {
    auto cb = [&status, &done](const Status& s) {
      status.Update(s);
      done.DecrementCount();
    };
    // IMPORTANT: notify worker to create worker session.
    worker->CreateWorkerSessionAsync(&request, &response, cb);
  }

  void Release() {
    if (worker != nullptr) {
      sess->worker_cache_->ReleaseWorker(*name, worker);
    }
  }

 private:
  WorkerInterface* GetOrCreateWorker() {
    return sess->worker_cache_->CreateWorker(*name);
  }

  void BuildRequest(const DeviceNameUtils::ParsedName& parsed_name,
                    const WorkerCacheFactoryOptions& opts) {
    request.set_session_handle(sess->handle_);
    BuildServerDef(parsed_name, opts, request.mutable_server_def());
  }

  void BuildServerDef(const DeviceNameUtils::ParsedName& parsed_name,
                      const WorkerCacheFactoryOptions& opts,
                      ServerDef* server_def) {
    *server_def->mutable_cluster() = *opts.cluster_def;
    server_def->set_protocol(*opts.protocol);
    server_def->set_job_name(parsed_name.job);
    server_def->set_task_index(parsed_name.task);
  }

 private:
  MasterSession* sess;
```

```
    // The worker name. (Not owned.)
    const string* name;

    // The worker referenced by name. (Not owned.)
    WorkerInterface* worker = nullptr;

    // Request and responses used for a given worker.
    CreateWorkerSessionRequest request;
    CreateWorkerSessionResponse response;
  };

  struct MasterSession::WorkerGroup {
    WorkerGroup(MasterSession* sess) : sess(sess) {}

    Status CreateWorkerSessions(const WorkerCacheFactoryOptions& opts) {
      TF_RETURN_IF_ERROR(CreateWorkers(opts));
      TF_RETURN_IF_ERROR(BroadcastWorkers());
      return Status::OK();
    }

    void ReleaseWorkers() {
      for (auto& worker : workers) {
        worker.Release();
      }
    }

   private:
    Status CreateWorkers(const WorkerCacheFactoryOptions& opts) {
      sess->worker_cache_->ListWorkers(&worker_names);
      for (auto& worker_name : worker_names) {
        TF_RETURN_IF_ERROR(AppendWorker(worker_name, opts));
      }
      return Status::OK();
    }

    // broadcast all workers to create worker session.
    Status BroadcastWorkers() {
      Status status = Status::OK();
      BlockingCounter done(workers.size());
      for (auto& worker : workers) {
        worker.CreateWorkerSession(done, status);
      }
      done.Wait();
      return status;
    }

    Status AppendWorker(const string& worker_name,
                        const WorkerCacheFactoryOptions& opts) {
      DeviceNameUtils::ParsedName parsed_name;
      TF_RETURN_IF_ERROR(ParseWorkerName(worker_name, &parsed_name));
      workers.emplace_back(Worker(sess, worker_name, parsed_name, opts));
      return Status::OK();
    }

    Status ParseWorkerName(const string& worker_name,
                           DeviceNameUtils::ParsedName* parsed_name) {
      if (!DeviceNameUtils::ParseFullName(worker_name, parsed_name)) {
        return errors::Internal("Could not parse name ", worker_name);
      }
      if (!parsed_name->has_job || !parsed_name->has_task) {
        return errors::Internal("Incomplete worker name ", worker_name);
      }
      return Status::OK();
    }

   private:
    MasterSession* sess;
    std::vector<string> worker_names;
    std::vector<Worker> workers;
  };

  Status MasterSession::CreateWorkerSessions(
```

```
      const WorkerCacheFactoryOptions& options) {
  CHECK(worker_cache_) << "CreateWorkerSessions should be called only with "
                       << "dynamic cluster membership.";

  WorkerGroup worker_group(this);

  // Release the workers.
  auto cleanup = gtl::MakeCleanup([&worker_group] {
    worker_group.ReleaseWorkers();
  });

  return worker_group.CreateWorkerSessions(options);
}
```

GrpcRemoteWorker

`GrpcRemoteWorker` 是访问远端 Worker 的 gRPC 客户端。它调用相应的 `stub` 调用远端服务。

```
struct GrpcRemoteWorker : WorkerInterface {
  void CreateWorkerSessionAsync(
      const CreateWorkerSessionRequest* request,
      CreateWorkerSessionResponse* response,
      StatusCallback done) override {
    IssueRequest(request, response, createworkersession_, std::move(done));
  }
};
```

GrpcWorkerService::CreateWorkerSessionHandler

在 Worker 端，`CreateWorkerSession` 消息由 `CreateWorkerSessionHandler` 回调处理。它在线程池中启动一个可运行的线程，触发 `Worker` 动态创建 `WorkerSession` 实例。

```
struct GrpcWorkerService : AsyncServiceInterface {
  void CreateWorkerSessionHandler(
      WorkerCall<CreateWorkerSessionRequest, CreateWorkerSessionResponse>*
          call) {
    Schedule([this, call]() {
      Status s = worker_->CreateWorkerSession(&call->request, \
        &call->response);
      call->SendResponse(ToGrpcStatus(s));
    });
    ENQUEUE_REQUEST(CreateWorkerSession, false);
  }
};
```

创建 WorkerSession 实例

Worker 将创建 WorkerSession 实例的职责委托给了 SessionMgr，由其统一管理和维护所有 WorkerSession 实例的生命周期。如图??（第??页）所示，SessionMgr 可能持有多个 WorkerSession 实例，每个 WorkerSession 实例使用 session_handle 标识。

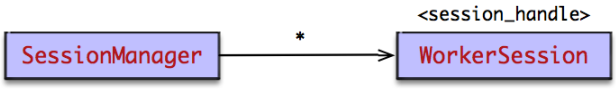


图 13-40 Session 管理器

```
void Worker::CreateWorkerSessionAsync(
    const CreateWorkerSessionRequest* request,
    CreateWorkerSessionResponse* response,
    StatusCallback done) {
  Status s = env_->session_mgr->CreateSession(
      request->session_handle(),
      request->server_def());
  done(s);
}
```

如图??（第??页）所示，WorkerSession 持有一个 GraphMgr 实例，用于注册和运行多个图实例。其中，每个图实例使用 graph_handle 标识。同时，每个 WorkerSession 持有一个 DeviceMgr 实例，用于管理本地计算设备的集合。

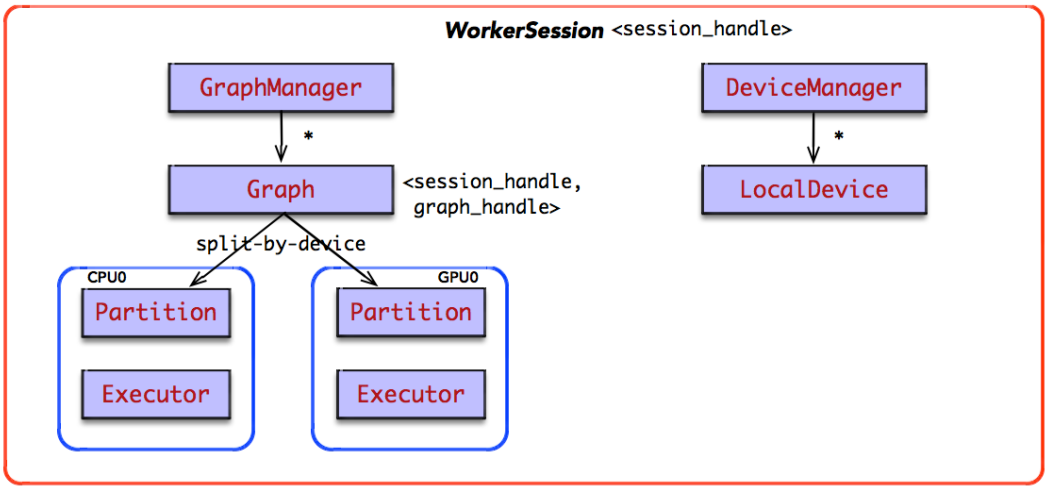


图 13-41 WorkerSession 领域模型：可注册和运行多个图实例

```
Status SessionMgr::CreateSession(const string& session,
                                 const ServerDef& server_def) {
  mutex_lock l(mu_);

  // 1. Create WorkerCacheInterface
  WorkerCacheInterface* worker_cache = nullptr;
  TF_RETURN_IF_ERROR(worker_cache_factory_(server_def, &worker_cache));
```

```
  // 2. Rename local devices
  auto worker_name = WorkerNameFromServerDef(server_def);
  std::vector<Device*> renamed_devices;
  for (Device* d : worker_env_->local_devices) {
    renamed_devices.push_back(
        RenamedDevice::NewRenamedDevice(worker_name, d, false));
  }
  std::unique_ptr<DeviceMgr> device_mgr(new DeviceMgr(renamed_devices));

  // 3. Create GraphMgr
  std::unique_ptr<GraphMgr> graph_mgr(
      new GraphMgr(worker_env_, device_mgr.get()));

  // 4. Create WorkerSession
  std::unique_ptr<WorkerSession> worker_session(new WorkerSession(
      worker_name, std::unique_ptr<WorkerCacheInterface>(worker_cache),
      std::move(device_mgr), std::move(graph_mgr)));

  // 5. Store (session_handle, WorkerSession) pair.
  sessions_.insert(std::make_pair(session, std::move(worker_session)));
  return Status::OK();
}
```

13.7 迭代执行

启动执行

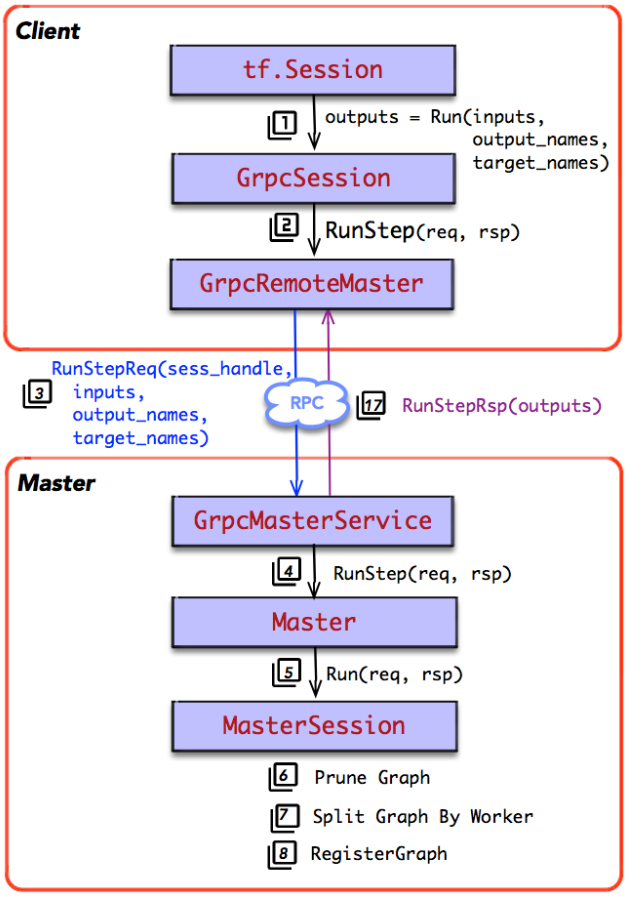


图 13-42 GprcSession: 启动 RunStep

GrpcSession::Run

```
namespace {
  using TensorIndex = std::unordered_map<string, int>;

  void BuildReqOptions(const SessionOptions& sess_options,
      const RunOptions& run_options,
      RunOptions& options) {
    options = run_options;
    if (run_options.timeout_in_ms() == 0) {
      options.set_timeout_in_ms(
          sess_options.config.operation_timeout_in_ms());
    }
  }

  void BuildReqFeeds(const vector<pair<string, Tensor>>& inputs,
      MutableRunStepRequestWrapper* req) {
    for (auto& it : inputs) {
```

```
      req->add_feed(it.first, it.second);
    }
  }

  void BuildReqFetches(const std::vector<string>& output_names,
      MutableRunStepRequestWrapper* req) {
    for (int i = 0; i < output_names.size(); ++i) {
      req->add_fetch(output_names[i]);
  }

  void BuildReqTargets(const std::vector<string>& target_names,
      MutableRunStepRequestWrapper* req) {
    for (string& target : target_names) {
      req->add_target(target);
    }
  }

  void BuildRunStepReq(
      const SessionOptions& sess_options,
      const RunOptions& run_options,
      const vector<pair<string, Tensor>>& inputs,
      const std::vector<string>& output_names,
      const std::vector<string>& target_names,
      MutableRunStepRequestWrapper* req) {
    BuildReqOptions(sess_options, run_options,
        req->mutable_options());
    BuildReqFeeds(inputs, req);
    BuildReqFetches(output_names, req);
    BuildReqTargets(target_names, req);
  }

  void BuildOuputNamesIndex(
      const std::vector<string>& output_names,
      TensorIndex& tensor_index) {
    for (int i = 0; i < output_names.size(); ++i) {
      const string& name = output_names[i];
      tensor_index.insert(make_pair(name, i));
    }
  }

  void BuildCallOptions(const RunOptions& options,
      CallOptions& call_options) {
    call_options.SetTimeout(options.timeout_in_ms());
  }

  Status DoSaveOutputs(const TensorIndex& tensor_index,
      const std::vector<string>& output_names,
      MutableRunStepResponseWrapper* resp,
      std::vector<Tensor>* outputs) {
    for (size_t i = 0; i < resp->num_tensors(); ++i) {
      auto fetch_it = tensor_index.find(resp->tensor_name(i));
      if (fetch_it == tensor_index.end()) {
        return errors::Internal(
           "unrequested fetch: ", resp->tensor_name(i));
      }

      Tensor output;
      TF_RETURN_IF_ERROR(resp->TensorValue(i, &output));
      (*outputs)[fetch_it->second] = output;
    }
  }

  Status SaveOutputs(const TensorIndex& tensor_index,
      const std::vector<string>& output_names,
      MutableRunStepResponseWrapper* resp,
      std::vector<Tensor>* outputs) {
    if (!output_names.empty()) {
      outputs->resize(output_names.size());
    }
    return DoSaveOutputs(tensor_index,
        output_names, rsep, outputs);
  }
```

```
  void SaveRunMetaData(MutableRunStepResponseWrapper* resp,
      RunMetadata* run_metadata) {
    if (run_metadata) {
      run_metadata->Swap(resp->mutable_metadata());
    }
  }

  Status SaveRspToOutputs(const TensorIndex& tensor_index,
      const std::vector<string>& output_names,
      MutableRunStepResponseWrapper* resp,
      std::vector<Tensor>* outputs,
      RunMetadata* run_metadata) {
    SaveRunMetaData(resp, run_metadata);
    return SaveOutputs(tensor_index, output_names, rsep, outputs);
  }
}

Status GrpcSession::Run(
    const RunOptions& run_options,
    const vector<pair<string, Tensor>>& inputs,
    const vector<string>& output_names,
    const vector<string>& target_names,
    std::vector<Tensor>* outputs,
    RunMetadata* run_metadata) {
  // 1. Build run step request.
  unique_ptr<MutableRunStepRequestWrapper> req(
      master_->CreateRunStepRequest());

  unique_ptr<MutableRunStepResponseWrapper> resp(
      master_->CreateRunStepResponse());

  BuildRunStepReq(options_, run_options, inputs,
      output_names, target_names, req.get());

  // 2. Build output tensor names index.
  TensorIndex tensor_index;
  BuildOuputNamesIndex(output_names, tensor_index);

  // 3. Build call options.
  CallOptions call_options;
  BuildCallOptions(req->options(), call_options)

  // 4. Do run step.
  TF_RETURN_IF_ERROR(RunProto(&call_options,
      req.get(), resp.get()));

  // 5. Save response to outputs.
  return SaveRspToOutputs(tensor_index, output_names,
      resp.get(), outputs, run_metadata);
}
```

```
Status GrpcSession::RunProto(
    CallOptions* call_options,
    MutableRunStepRequestWrapper* req,
    MutableRunStepResponseWrapper* resp) {
  {
    mutex_lock l(mu_);
    req->set_session_handle(handle_);
  }
  return master_->RunStep(call_options, req, resp);
}
```

GrpcRemoteMaster::RunStep

```
struct GrpcRemoteMaster : MasterInterface {
  using MasterServiceStub = ::grpc::MasterService::Stub;

  Status RunStep(CallOptions* call_options, RunStepRequestWrapper* request,
                 MutableRunStepResponseWrapper* response) override {
    ::grpc::ClientContext ctx;
    return Call(&ctx, call_options, &request->ToProto(),
                get_proto_from_wrapper(response),
                &MasterServiceStub::RunStep);
  }
};
```

GrpcMasterService::RunStepHandler

```
struct GrpcMasterService : AsyncServiceInterface {
  using RunStepCall = MasterCall<RunStepRequest, RunStepResponse>;

  void RunStepHandler(RunStepCall* call) {
    CallOptions* call_opts = CreateCallOptions(call);

    RunStepRequestWrapper* wrapped_request =
        new ProtoRunStepRequest(&call->request);

    MutableRunStepResponseWrapper* wrapped_response =
        new NonOwnedProtoRunStepResponse(&call->response);

    call->SetCancelCallback([call_opts]() {
        call_opts->StartCancel();
    });

    master_impl_->RunStep(call_opts, wrapped_request, wrapped_response,
      [call, call_opts, wrapped_request, wrapped_response](
          const Status& status) {
        call->ClearCancelCallback();
        delete call_opts;
        delete wrapped_request;
        call->SendResponse(ToGrpcStatus(status));
      });
    ENQUEUE_REQUEST(RunStep, true);
  }

 private:
  CallOptions* CreateCallOptions(RunStepCall* call) {
    CallOptions* call_opts = new CallOptions;
    if (call->request.options().timeout_in_ms() > 0) {
      call_opts->SetTimeout(call->request.options().timeout_in_ms());
    } else {
      call_opts->SetTimeout(default_timeout_in_ms_);
    }
    return call_opts;
  }
};
```

Master::RunStep

```
void Master::RunStep(CallOptions* opts,
    const RunStepRequestWrapper* req,
    MutableRunStepResponseWrapper* resp,
    DoneClosure done) {
  auto session = FindMasterSession(req->session_handle());
  SchedClosure([this, session, opts, req, resp, done]() {
    Status status = session->Run(opts, *req, resp);
    session->Unref();
    done(status);
  });
}
```

MasterSession::Run

```
Status MasterSession::Run(
    CallOptions* opts,
    const RunStepRequestWrapper& req,
    MutableRunStepResponseWrapper* resp) {
  Status status;
  if (!req.partial_run_handle().empty()) {
    status = DoPartialRun(opts, req, resp);
  } else {
    status = DoRunWithLocalExecution(opts, req, resp);
  }
  return status;
}
```

```
Status MasterSession::DoRunWithLocalExecution(
    CallOptions* opts, const RunStepRequestWrapper& req,
    MutableRunStepResponseWrapper* resp) {

  // 1. Prune: build ReffedClientGraph.
  BuildGraphOptions bgopts;
  BuildBuildGraphOptions(req, &bgopts);

  ReffedClientGraph* rcg = nullptr;
  int64 count = 0;
  TF_RETURN_IF_ERROR(StartStep(bgopts, &count, &rcg, false));

  // 2. Build and Register partitions to workers.
  core::ScopedUnref unref(rcg);
  TF_RETURN_IF_ERROR(BuildAndRegisterPartitions(rcg));

  // 3. Run partitions: notify all of workers to run partitions.
  uint64 step_id = (random::New64() & ((1uLL << 56) - 1)) | (1uLL << 56);
  Status s = rcg->RunPartitions(env_, step_id, count, &pss, opts, req, resp,
                                &cancellation_manager_, false);
  // 4. Cleaup Partitions: notify all of workers to clearup partitions.
  Ref();
  rcg->Ref();
  rcg->CleanupPartitionsAsync(step_id, [this, rcg](const Status& s) {
    rcg->Unref();
    Unref();
  });
  return s;
}
```

MasterSession::BuildAndRegisterPartitions

```
Status MasterSession::BuildAndRegisterPartitions(ReffedClientGraph* rcg) {
  PartitionOptions popts;
  popts.node_to_loc = SplitByWorker; // IMPORTANT
  popts.flib_def = rcg->client_graph()->flib_def.get();
  popts.control_flow_added = false;

  popts.new_name = [this](const string& prefix) {
    mutex_lock l(mu_);
    return strings::StrCat(prefix, "_S", next_node_id_++);
  };

  popts.get_incarnation = [this](const string& name) -> int64 {
    auto d = devices_->FindDeviceByName(name);
    return d->attributes().incarnation();
  };

  TF_RETURN_IF_ERROR(rcg->RegisterPartitions(popts));
  return Status::OK();
}
```

ReffedClientGraph::RegisterPartitions

```
Status ReffedClientGraph::RegisterPartitions(
    const PartitionOptions& popts) {
  {
    mu_.lock();
    if (!init_started_) {
      init_started_ = true;
      mu_.unlock();

      std::unordered_map<string, GraphDef> graph_defs;
      Status s = DoBuildPartitions(popts, &graph_defs);
      if (s.ok()) {
        s = DoRegisterPartitions(popts, std::move(graph_defs));
      }

      mu_.lock();
      init_result_ = s;
      init_done_.Notify();
    } else {
      mu_.unlock();
      init_done_.WaitForNotification();
      mu_.lock();
    }
    Status result = init_result_;
    mu_.unlock();
    return result;
  }
}
```

图分裂：SplitByWorker

ReffedClientGraph::DoBuildPartitions

```
Status MasterSession::ReffedClientGraph::DoBuildPartitions(
    PartitionOptions popts,
    std::unordered_map<string, GraphDef>* out_partitions) {
  // split full graph by worker name.
  return Partition(popts, &client_graph_->graph, out_partitions);
}
```

注册图

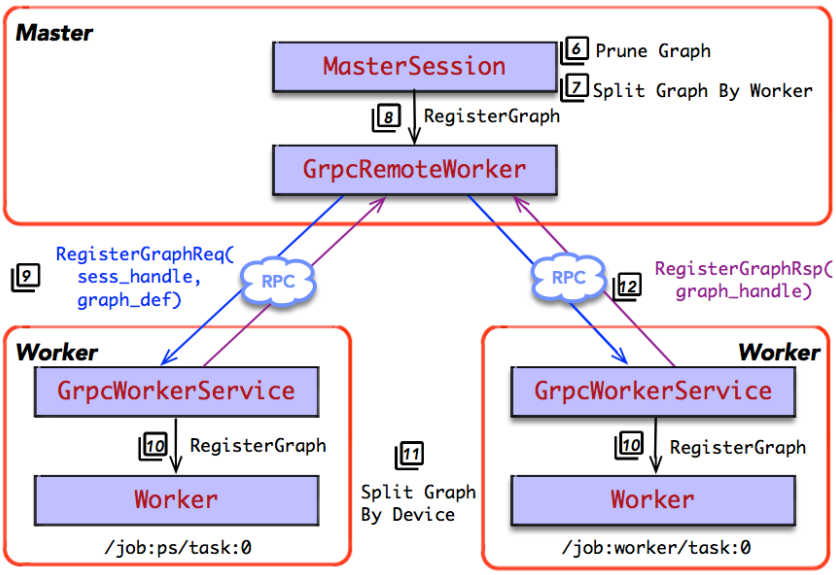


图 13-43 RegisterGraph

ReffedClientGraph::DoRegisterPartitions

```
Status ReffedClientGraph::DoRegisterPartitions(
    const PartitionOptions& popts,
    std::unordered_map<string, GraphDef> graph_partitions) {
  partitions_.reserve(graph_partitions.size());
  Status s;
  for (auto& name_def : graph_partitions) {
    partitions_.resize(partitions_.size() + 1);
    Part* part = &partitions_.back();
    part->name = name_def.first;
    TrackFeedsAndFetches(part, name_def.second, popts);
    part->worker = worker_cache_->CreateWorker(part->name);
  }

  struct Call {
```

```
      RegisterGraphRequest req;
      RegisterGraphResponse resp;
      Status status;
    };

    const int num = partitions_.size();
    gtl::InlinedVector<Call, 4> calls(num);

    BlockingCounter done(num);
    for (int i = 0; i < num; ++i) {
      const Part& part = partitions_[i];
      Call* c = &calls[i];

      c->req.set_session_handle(session_handle_);
      c->req.mutable_graph_def()->Swap(&graph_partitions[part.name]);
      *c->req.mutable_graph_options() = session_opts_.config.graph_options();
      *c->req.mutable_debug_options() = debug_opts_;

      auto cb = [c, &done](const Status& s) {
        c->status = s;
        done.DecrementCount();
      };
      part.worker->RegisterGraphAsync(&c->req, &c->resp, cb);
    }
    done.Wait();

    for (int i = 0; i < num; ++i) {
      Call* c = &calls[i];
      s.Update(c->status);
      partitions_[i].graph_handle = c->resp.graph_handle();
    }
    return s;
  }
```

GrpcRemoteWorker::RegisterGraphAsync

```
  class GrpcRemoteWorker : public WorkerInterface {
    void RegisterGraphAsync(const RegisterGraphRequest* request,
                            RegisterGraphResponse* response,
                            StatusCallback done) override {
      IssueRequest(request, response, registergraph_, std::move(done));
    }

    void IssueRequest(const protobuf::Message* request,
                      protobuf::Message* response, const ::grpc::string& \
                        method,
                      StatusCallback done, CallOptions* call_opts = nullptr) {
      new RPCState<protobuf::Message>(counter_, &stub_, cq_, method, *request,
                                      response, std::move(done), call_opts);
    }
  };
```

GrpcWorkerService::RegisterGraphHandler

```
  class GrpcWorkerService : public AsyncServiceInterface {
    void RegisterGraphHandler(
        WorkerCall<RegisterGraphRequest, RegisterGraphResponse>* call) {
      Schedule([this, call]() {
        Status s = worker_->RegisterGraph(&call->request, &call->response);
        call->SendResponse(ToGrpcStatus(s));
```

```
      });
      ENQUEUE_REQUEST(RegisterGraph, false);
    }
};
```

Worker::RegisterGraphAsync

```
void Worker::RegisterGraphAsync(
    const RegisterGraphRequest* request,
    RegisterGraphResponse* response,
    StatusCallback done) {
  auto session = FindWorkerSession(request);
  Status s = session->graph_mgr->Register(
      request->session_handle(),
      request->graph_def(),
      request->graph_options(),
      response->mutable_graph_handle());
  done(s);
}
```

GraphMgr::Register

```
Status GraphMgr::Register(
    const string& session,
    const GraphDef& gdef,
    const GraphOptions& graph_options,
    string* handle) {
  Item* item = new Item;
  Status s = InitItem(session, gdef, graph_options, item);
  if (!s.ok()) {
    item->Unref();
    return s;
  }

  // Generate unique graph_handle,
  // and register [graph_handle, graph_def] to table.
  {
    mutex_lock l(mu_);
    *handle = strings::Printf("%016llx", ++next_id_);
    item->handle = *handle;
    CHECK(table_.insert({*handle, item}).second);
  }
  return Status::OK();
}
```

图分裂：SplitByDevice

```
Status GraphMgr::InitItem(
    const string& session, const GraphDef& gdef,
    const GraphOptions& graph_options,
    Item* item) {
  item->session = session;
  item->lib_def.reset(
      new FunctionLibraryDefinition(OpRegistry::Global(), gdef.library()));
```

```
item->proc_flr.reset(new ProcessFunctionLibraryRuntime(
    device_mgr_, worker_env_->env, gdef.versions().producer(),
    item->lib_def.get(), graph_options.optimizer_options()));

// 1. Constructs the full graph out of "gdef"
Graph graph(OpRegistry::Global());
GraphConstructorOptions opts;
opts.allow_internal_ops = true;
opts.expect_device_spec = true;
TF_RETURN_IF_ERROR(ConvertGraphDefToGraph(opts, gdef, &graph));

// 2. Splits "graph" into multiple subgraphs by device names.
std::unordered_map<string, GraphDef> partitions;
PartitionOptions popts;
popts.node_to_loc = SplitByDevice;  // IMPORTANT.
popts.new_name = [this](const string& prefix) {
  mutex_lock l(mu_);
  return strings::StrCat(prefix, "_G", next_id_++);
};
popts.get_incarnation = [this](const string& name) -> int64 {
  Device* device = nullptr;
  Status s = device_mgr_->LookupDevice(name, &device);
  if (s.ok()) {
    return device->attributes().incarnation();
  } else {
    return PartitionOptions::kIllegalIncarnation;
  }
};
popts.flib_def = &graph.flib_def();
popts.control_flow_added = true;
popts.scheduling_for_recvs = graph_options.enable_recv_scheduling();

// IMPORTANT.
TF_RETURN_IF_ERROR(Partition(popts, &graph, &partitions));

// 3. convert GraphDef partitions to Graph partitions.
std::unordered_map<string, std::unique_ptr<Graph>> partition_graphs;
for (const auto& partition : partitions) {
  std::unique_ptr<Graph> device_graph(new Graph(OpRegistry::Global()));
  GraphConstructorOptions device_opts;
  // There are internal operations (e.g., send/recv) that we now allow.
  device_opts.allow_internal_ops = true;
  device_opts.expect_device_spec = true;
  TF_RETURN_IF_ERROR(ConvertGraphDefToGraph(device_opts, partition.second,
                                            device_graph.get()));
  partition_graphs.emplace(partition.first, std::move(device_graph));
}

// 4. Build executors_and_partitions(item->units) = [(e0, p0),
// (e1, p1), ...], and (e_n, p_n) is called ExecutionUnit.
LocalExecutorParams params;
item->units.reserve(partitions.size());
item->graph_mgr = this;

for (auto& p : partition_graphs) {
  const string& device_name = p.first;
  std::unique_ptr<Graph>& subgraph = p.second;
  item->units.resize(item->units.size() + 1);
  ExecutionUnit* unit = &(item->units.back());

  // Construct the root executor for the subgraph.
  params.device = unit->device;
  params.function_library = lib;
  params.create_kernel = [session, lib, opseg](
      const NodeDef& ndef, OpKernel** kernel) {
    // Caches the kernel only if the node is stateful.
    if (!lib->IsStateful(ndef.op())) {
      return lib->CreateKernel(ndef, kernel);
    }
    auto create_fn = [lib, &ndef](OpKernel** kernel) {
      return lib->CreateKernel(ndef, kernel);
```

```
        };
        // Kernels created for subgraph nodes need to be cached. On
        // cache miss, create_fn() is invoked to create a kernel based
        // on the function library here + global op registry.
        return opseg->FindOrCreate(session, ndef.name(), kernel, create_fn);
      };

      params.delete_kernel = [lib](OpKernel* kernel) {
        // If the node is stateful, opseg owns it. Otherwise, delete it.
        if (kernel && !lib->IsStateful(kernel->type_string())) {
          delete kernel;
        }
      };

      unit->graph = subgraph.get();
      TF_RETURN_IF_ERROR(
          NewLocalExecutor(params, subgraph.release(), &unit->root));
    }
    return Status::OK();
  }
```

运行图

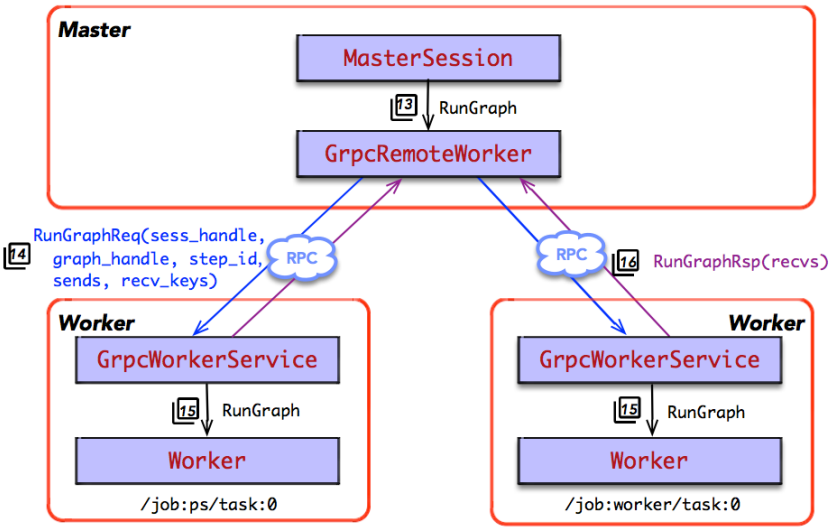


图 13-44 RunGraph

ReffedClientGraph::RunPartitions

```
Status MasterSession::ReffedClientGraph::RunPartitions(
    const MasterEnv* env, int64 step_id, int64 execution_count,
    PerStepState* pss, CallOptions* call_opts, const RunStepRequestWrapper& \
      req,
    MutableRunStepResponseWrapper* resp, CancellationManager* cm,
    const bool is_last_partial_run) {


  // 1. Prepares a number of calls to workers.
  // One call per partition.
  const int num = partitions_.size();
```

```
    RunManyGraphs calls(num);

    for (int i = 0; i < num; ++i) {
      const Part& part = partitions_[i];
      RunManyGraphs::Call* c = calls.get(i);
      c->req.reset(part.worker->CreateRunGraphRequest());
      c->resp.reset(part.worker->CreateRunGraphResponse());
      if (is_partial_) {
        c->req->set_is_partial(is_partial_);
        c->req->set_is_last_partial_run(is_last_partial_run);
      }
      c->req->set_session_handle(session_handle_);
      c->req->set_graph_handle(part.graph_handle);
      c->req->set_step_id(step_id);

      for (const auto& feed_key : part.feed_key) {
        const string& feed = feed_key.first;
        const string& key = feed_key.second;
        const int64 feed_index = feeds[feed];
        TF_RETURN_IF_ERROR(
            c->req->AddSendFromRunStepRequest(req, feed_index, key));
      }

      for (const auto& key_fetch : part.key_fetch) {
        const string& key = key_fetch.first;
        c->req->add_recv_key(key);
      }
    }

    // 2. Issues RunGraph calls.
    for (int i = 0; i < num; ++i) {
      const Part& part = partitions_[i];
      RunManyGraphs::Call* call = calls.get(i);
      part.worker->RunGraphAsync(
          &call->opts, call->req.get(), call->resp.get(),
          std::bind(&RunManyGraphs::WhenDone, &calls, i, \
            std::placeholders::_1));
    }

    // 3. Waits for the RunGraph calls.
    call_opts->SetCancelCallback([&calls]() { calls.StartCancel(); });
    auto token = cm->get_cancellation_token();
    bool success =
        cm->RegisterCallback(token, [&calls]() { calls.StartCancel(); });
    if (!success) {
      calls.StartCancel();
    }

    calls.Wait();

    call_opts->ClearCancelCallback();
    if (success) {
      cm->DeregisterCallback(token);
    } else {
      return errors::Cancelled("Step was cancelled");
    }

    // 4. Collects fetches.
    Status status = calls.status();
    if (status.ok()) {
      for (int i = 0; i < num; ++i) {
        const Part& part = partitions_[i];
        MutableRunGraphResponseWrapper* run_graph_resp = \
          calls.get(i)->resp.get();
        for (size_t j = 0; j < run_graph_resp->num_recvs(); ++j) {
          auto iter = part.key_fetch.find(run_graph_resp->recv_key(j));
          if (iter == part.key_fetch.end()) {
            status.Update(errors::Internal("Unexpected fetch key: ",
                                           run_graph_resp->recv_key(j)));
            break;
          }
          const string& fetch = iter->second;
```

```
          status.Update(
              resp->AddTensorFromRunGraphResponse(fetch, run_graph_resp, j));
          if (!status.ok()) {
            break;
          }
        }
      }
    }
    return status;
  }
```

GrpcRemoteWorker::RunGraphAsync

```
  struct GrpcRemoteWorker : public WorkerInterface {
    void RunGraphAsync(
        CallOptions* call_opts,
        RunGraphRequestWrapper* request,
        MutableRunGraphResponseWrapper* response,
        StatusCallback done) override {
      IssueRequest(&request->ToProto(),
          get_proto_from_wrapper(response),
          rungraph_, std::move(done), call_opts);
    }
  };
```

GrpcWorkerService::RunGraphHandler

```
  struct GrpcWorkerService : AsyncServiceInterface {
    void RunGraphHandler(WorkerCall<RunGraphRequest, RunGraphResponse>* call) {
      Schedule([this, call]() {
        auto wrapped_req = new ProtoRunGraphRequest(&call->request);
        auto wrapped_rsp = new NonOwnedProtoRunGraphResponse(&call->response);

        auto call_opts = new CallOptions;
        call->SetCancelCallback([call_opts]() {
            call_opts->StartCancel();
        });

        worker_->RunGraphAsync(call_opts, wrapped_req, wrapped_rsp,
          [call, call_opts, wrapped_req, wrapped_rsp](const Status& s) {
              call->ClearCancelCallback();
              delete call_opts;
              delete wrapped_req;
              delete wrapped_rsp;
              call->SendResponse(ToGrpcStatus(s));
          });
      });
      ENQUEUE_REQUEST(RunGraph, true);
    }
  };
```

Worker::RunGraphAsync

```
void Worker::RunGraphAsync(
    CallOptions* opts,
    RunGraphRequestWrapper* request,
    MutableRunGraphResponseWrapper* response,
    StatusCallback done) {
  if (request->is_partial()) {
    DoPartialRunGraph(opts, request, response, std::move(done));
  } else {
    DoRunGraph(opts, request, response, std::move(done));
  }
}
```

```
void Worker::DoRunGraph(
    CallOptions* opts,
    RunGraphRequestWrapper* request,
    MutableRunGraphResponseWrapper* response,
    StatusCallback done) {
  const int64 step_id = request->step_id();

  // 1. Prepare inputs and outputs.
  GraphMgr::NamedTensors in;
  GraphMgr::NamedTensors* out = new GraphMgr::NamedTensors;
  Status s = PrepareRunGraph(request, &in, out);
  if (!s.ok()) {
    delete out;
    done(s);
    return;
  }

  // 2. Register Cancellation callback.
  CancellationManager* cm = new CancellationManager;
  opts->SetCancelCallback([this, cm, step_id]() {
    cm->StartCancel();
    AbortStep(step_id);
  });

  CancellationToken token;
  {
    mutex_lock l(mu_);
    token = cancellation_manager_->get_cancellation_token();
    bool already_cancelled = !cancellation_manager_->RegisterCallback(
        token, [cm]() { cm->StartCancel(); });
    if (already_cancelled) {
      opts->ClearCancelCallback();
      delete cm;
      delete out;
      done(errors::Aborted("Call was aborted"));
      return;
    }
  }

  // 3. Start Execution.
  auto session =
      FindWorkerSession(request);

  session->graph_mgr->ExecuteAsync(
      request->graph_handle(), step_id, session,
      request->exec_opts(), response, cm, in,
      [ this, step_id, response, session, cm,
        out, token, opts, done](Status s) {

        // 4. Receive output tensors from grpc remote rendezvous.
        if (s.ok()) {
          s = session->graph_mgr->RecvOutputs(step_id, out);
        }

        // 5. Unregister Cancellation callback
        opts->ClearCancelCallback();
```

```
          {
            mutex_lock l(mu_);
            cancellation_manager_->DeregisterCallback(token);
          }
          delete cm;

          // 6. Save to RunStepResponse.
          if (s.ok()) {
            for (const auto& p : *out) {
              const string& key = p.first;
              const Tensor& val = p.second;
              response->AddRecv(key, val);
            }
          }
          delete out;
          done(s);
        });
  }
```

GraphMgr

```
def send_inputs(remote_rendezvous, inputs):
  for (key, tensor) in inputs:
    remote_rendezvous.send(key, tensor)

def do_run_partitions(executors_and_partitions):
  barrier = ExecutorBarrier(executors_and_partitions.size())
  for (executor, partition) in executors_and_partitions:
    executor.run(partition, barrier.on_done())
  barrier.wait()

def recv_outputs(remote_rendezvous, outputs):
  for (key, tensor) in outputs:
    remote_rendezvous.recv(key, tensor)

def run_step(executors_and_partitions, inputs, outputs):
  remote_rendezvous = RpcRemoteRendezvous()
  send_inputs(remote_rendezvous, inputs)
  do_run_partitions(executors_and_partitions)
  recv_outputs(remote_rendezvous, outputs)
```

```
class RpcRemoteRendezvous(Rendezvous)
  def __init__():
    self._local = LocalRendezvous()

  def send(key, tensor):
    self._local.send(key, tensor)

  def recv(key):
    if self.is_same_worker(key.src(), key.dst()):
      return self._local.recv(key)
    else:
      return rpc(recv_tensor_request(key.dst()))

  def is_same_worker(src, dst):
    return (src.job_name == dst.job_name and
      src.replica == dst.replica and
      src.task_index == dst.task_index)
```

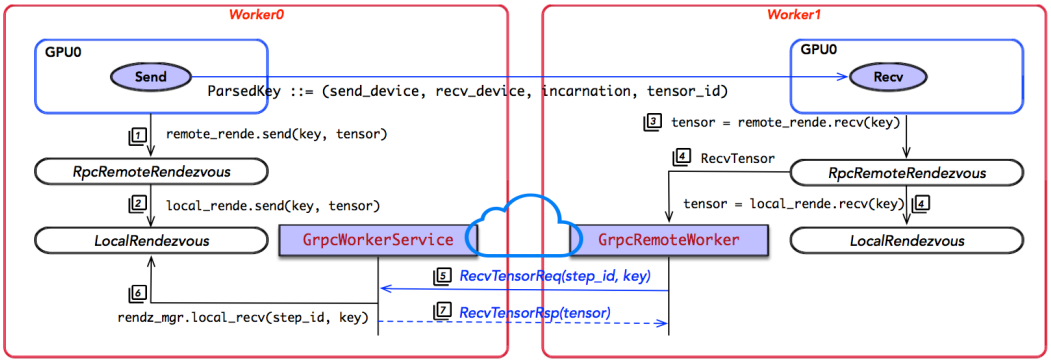


图 13-45 Worker: RunStep 形式化

```
void GraphMgr::ExecuteAsync(
    const string& handle, const int64 step_id,
    WorkerSession* session, const ExecutorOpts& opts,
    MutableRunGraphResponseWrapper* response,
    CancellationManager* cancellation_manager,
    const NamedTensors& in, StatusCallback done) {
  // 1. Lookup an item. Holds one ref while executing.
  // One item per registered graph.
  Item* item = nullptr;
  {
```

```
      mutex_lock l(mu_);
      auto iter = table_.find(handle);
      if (iter != table_.end()) {
        item = iter->second;
        item->Ref();
      }
    }

    RemoteRendezvous* rendezvous = worker_env_->rendezvous_mgr->Find(step_id);
    Status s = rendezvous->Initialize(session);

    // 2. Sends inputs to rendezvous.
    if (s.ok()) {
      s = SendInputsToRendezvous(rendezvous, in);
    }

    // 3. Start parallel executors.
    StartParallelExecutors(
        handle, step_id, item, rendezvous, collector,
        cost_graph, cancellation_manager,
        [this, item, rendezvous, done](const Status& s) {
            // 4. Recvs outputs from rendezvous.
            done(s);
            rendezvous->Unref();
            item->Unref();
        });
  }
```

```
  Status GraphMgr::SendInputsToRendezvous(
      Rendezvous* rendezvous, const NamedTensors& in) {
    Rendezvous::ParsedKey parsed;
    for (auto& p : in) {
      auto& key = p.first;
      auto& val = p.second;

      Status s = Rendezvous::ParseKey(key, &parsed);
      if (s.ok()) {
        s = rendezvous->Send(parsed, Rendezvous::Args(), val, false);
      }
      if (!s.ok()) {
        return s;
      }
    }
    return Status::OK();
  }
```

```
  void GraphMgr::StartParallelExecutors(
      const string& handle, int64 step_id,
      Item* item, Rendezvous* rendezvous,
      StepStatsCollector* collector,
      CancellationManager* cancellation_manager,
      StatusCallback done) {

    // 1. Wait until pending == 0, with default is num_units,
    // `pending -= 1` when one partition graph is done.
    int num_units = item->units.size();
    ExecutorBarrier* barrier =
        new ExecutorBarrier(
            num_units, rendezvous, [done](const Status& s) {
                done(s);
            });

    Executor::Args args;
    {
      mutex_lock l(mu_);
      args.step_id = ++next_id_;
    }
```

```
  args.rendezvous = rendezvous;
  args.cancellation_manager = cancellation_manager;
  args.stats_collector = collector;
  args.step_container = step_container;
  args.sync_on_finish = sync_on_finish_;

  using std::placeholders::_1;
  args.runner = std::bind(
      &thread::ThreadPool::Schedule,
      worker_env_->compute_pool, _1);

  2. Broadcast all partitions to run
  for (const auto& unit : item->units) {
    unit.root->RunAsync(args, barrier->Get());
  }
}
```

```
Status GraphMgr::RecvOutputsFromRendezvous(
    Rendezvous* rendezvous, NamedTensors* out) {
  // Receives values requested by the caller.
  Rendezvous::ParsedKey parsed;
  for (auto& p : *out) {
    auto& key = p.first;
    auto& val = p.second;

    bool is_dead = false;
    Status s = Rendezvous::ParseKey(key, &parsed);
    if (s.ok()) {
      s = rendezvous->Recv(parsed, Rendezvous::Args(), &val, &is_dead);
    }

    if (is_dead) {
      s = errors::InvalidArgument("The tensor returned for ", key,
                                  " was not valid.");
    }

    if (!s.ok()) {
      return s;
    }
  }
  return Status::OK();
}

Status GraphMgr::RecvOutputs(int64 step_id, NamedTensors* out) {
  Rendezvous* rendezvous = worker_env_->rendezvous_mgr->Find(step_id);
  Status s = RecvOutputsFromRendezvous(rendezvous, out);
  rendezvous->Unref();
  return s;
}
```

Rendzvous

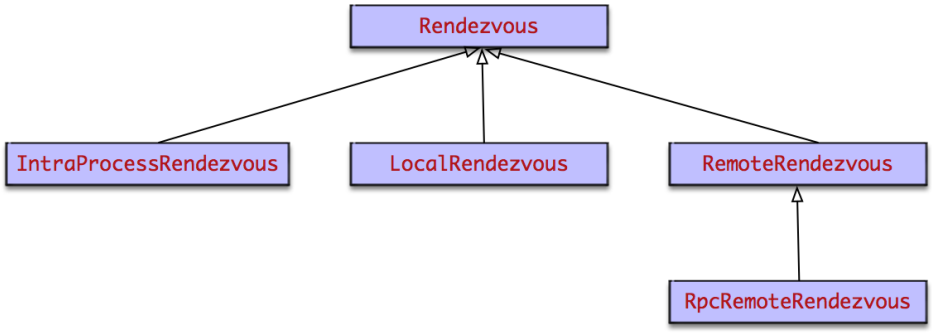


图 13-46 Rendezvous 层次结构

多态创建

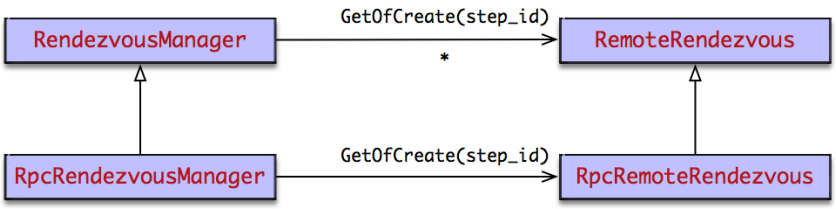


图 13-47 RemoteRendezvous 多态创建

发送

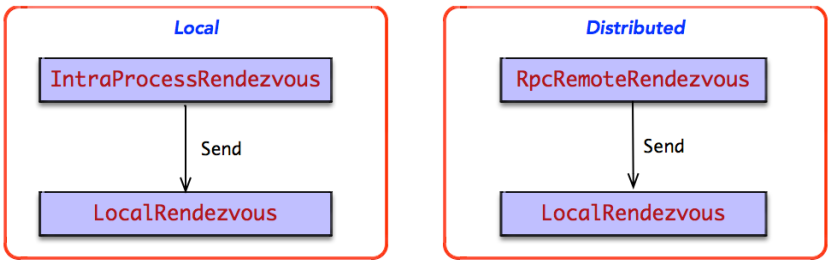


图 13-48 Rendezvous 发送

接收

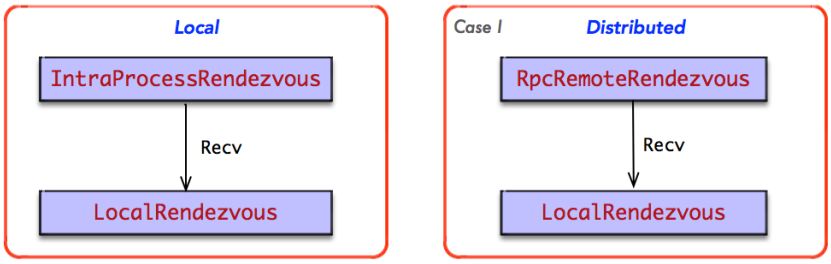


图 13-49 Rendezvous 接收：分布式发送端与接收端在同一个 Worker 内

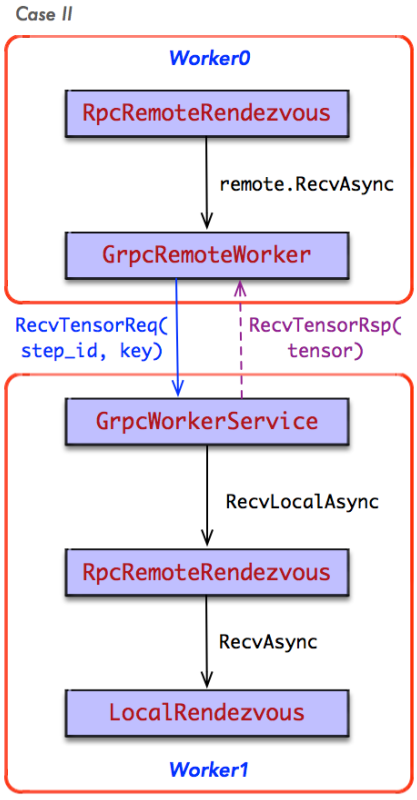


图 13-50 Rendezvous 接收：分布式发送端与接收端不在同一个 Worker 内

注销图

13.8 关闭会话

GrpcSession

```
Status GrpcSession::Close() {
  CloseSessionRequest req;
  {
    mutex_lock l(mu_);
    if (handle_.empty()) {
      return errors::InvalidArgument("A session is not created yet....");
    }
    req.set_session_handle(handle_);
    handle_.clear();
  }
  CloseSessionResponse resp;
  CallOptions call_options;
  call_options.SetTimeout(options_.config.operation_timeout_in_ms());
  return master_->CloseSession(&call_options, &req, &resp);
}
```

GrpcRemoteMaster

```
struct GrpcRemoteMaster : MasterInterface {
  Status CloseSession(CallOptions* call_options,
                      const CloseSessionRequest* request,
                      CloseSessionResponse* response) override {
    ::grpc::ClientContext ctx;
    ctx.set_fail_fast(false);
    SetDeadline(&ctx, call_options->GetTimeout());
    return FromGrpcStatus(stub_->CloseSession(&ctx, *request, response));
  }
};
```

GrpcMasterService

```
struct GrpcMasterService : AsyncServiceInterface {
  void CloseSessionHandler(
      MasterCall<CloseSessionRequest, CloseSessionResponse>* call) {
    master_impl_->CloseSession(&call->request, &call->response,
                               [call](const Status& status) {
                                 call->SendResponse(ToGrpcStatus(status));
                               });
    ENQUEUE_REQUEST(CloseSession, false);
  }
};
```

Master

```
void Master::CloseSession(const CloseSessionRequest* req,
                          CloseSessionResponse* resp, MyClosure done) {
  MasterSession* session = nullptr;
  {
    mu_.lock();
    auto iter = sessions_.find(req->session_handle());
    if (iter == sessions_.end()) {
      mu_.unlock();
      done(errors::Aborted(
          "Session ", req->session_handle(),
          " is not found. Possibly, this master has restarted."));
      return;
    }
    session = iter->second;
    sessions_.erase(iter);
    mu_.unlock();
  }

  // Session Close() blocks on thread shutdown. Therefore, we need to
  // delete it in non-critical thread.
  SchedClosure([session, done]() {
    Status s = session->Close();
    session->Unref();
    done(s);
  });
}
```

MasterSession

```
Status MasterSession::Close() {
  {
    mutex_lock l(mu_);
    closed_ = true;  // All subsequent calls to Run() or Extend() will fail.
  }
  cancellation_manager_.StartCancel();
  std::vector<ReffedClientGraph*> to_unref;
  {
    mutex_lock l(mu_);
    while (num_running_ != 0) {
      num_running_is_zero_.wait(l);
    }
    ClearRunsTable(&to_unref, &run_graphs_);
    ClearRunsTable(&to_unref, &partial_run_graphs_);
  }
  for (ReffedClientGraph* rcg : to_unref) rcg->Unref();
  return Status::OK();
}
```

ReffedClientGraph

```
ReffedClientGraph::~ReffedClientGraph() {
  DeregisterPartitions();
}
```

```
void ReffedClientGraph::DeregisterPartitions() {
  struct Call {
    DeregisterGraphRequest req;
    DeregisterGraphResponse resp;
  };
  for (Part& part : partitions_) {
    if (!part.graph_handle.empty()) {
      Call* c = new Call;
      c->req.set_session_handle(session_handle_);
      c->req.set_graph_handle(part.graph_handle);

      WorkerCacheInterface* worker_cache = worker_cache_;
      const string name = part.name;
      WorkerInterface* w = part.worker;

      auto cb = [worker_cache, c, name, w](const Status& s) {
        if (!s.ok()) {
          // This error is potentially benign, so we don't log at the
          // error level.
          LOG(INFO) << "DeregisterGraph error: " << s;
        }
        delete c;
        worker_cache->ReleaseWorker(name, w);
      };
      w->DeregisterGraphAsync(&c->req, &c->resp, cb);
    }
  }
}
```

GrpcWorkerService

```
struct GrpcWorkerService : AsyncServiceInterface {
  void CreateWorkerSessionHandler(
      WorkerCall<CreateWorkerSessionRequest, CreateWorkerSessionResponse>*
          call) {
    Schedule([this, call]() {
      Status s = worker_->CreateWorkerSession(&call->request, \
        &call->response);
      call->SendResponse(ToGrpcStatus(s));
    });
    ENQUEUE_REQUEST(CreateWorkerSession, false);
  }
};
```

Worker

```
void Worker::DeregisterGraphAsync(const DeregisterGraphRequest* request,
                                  DeregisterGraphResponse* response,
                                  StatusCallback done) {
  WorkerSession* session =
      env_->session_mgr->WorkerSessionForSession(request->session_handle());
  Status s = session->graph_mgr->Deregister(request->graph_handle());

  done(s);
}
```

GraphMgr

```
Status GraphMgr::Deregister(const string& handle) {
  Item* item = nullptr;
  {
    mutex_lock l(mu_);
    auto iter = table_.find(handle);
    if (iter == table_.end()) {
      return errors::Aborted("Graph handle is not found: ", handle,
                             ". Possibly, this worker just restarted.");
    }
    item = iter->second;
    table_.erase(iter);
  }
  item->Unref();
  return Status::OK();
}
```

```
GraphMgr::Item::~Item() {
  for (const auto& unit : this->units) {
    delete unit.root;
    unit.device->op_segment()->RemoveHold(this->session);
  }
}
```

第 V 部分

模型训练

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

- Martin Flower

14

BP 算法

14.1 TensorFlow 实现

TensorFlow 是一个实现了自动微分的软件系统。首先，它构造正向的计算图，实现计算图的前向计算。当调用 `Optimizer.minimize` 方法时，使用 `compute_gradients` 方法，实现反向计算图的构造；使用 `apply_gradients` 方法，实现参数更新的子图构造。

```
class Optimizer(object):
  def minimize(self, loss, var_list=None, global_step=None):
    """Add operations to minimize loss by updating var_list.
    """
    grads_and_vars = self.compute_gradients(
      loss, var_list=var_list)
    return self.apply_gradients(
      grads_and_vars,
      global_step=global_step)
```

计算梯度

`compute_gradients` 将根据 `loss` 的值，求解 `var_list=[v1, v2, ..., vn]` 的梯度，最终返回的结果为:`[(grad_v1, v1), (grad_v2, v2), ..., (grad_vn, vn)]`。其中，`compute_gradients` 将调用 `gradients` 方法，构造反向传播的子图。

以一个简单实例，讲解反向子图的构造过程。首先，构造前向的计算图。

```
X = tf.placeholder("float", name="X")
Y = tf.placeholder("float", name="Y")
w = tf.Variable(0.0, name="w")
b = tf.Variable(0.0, name="b")
loss = tf.square(Y - X*w - b)
global_step = tf.Variable(0, trainable=False, collections=[])
```

使用 `compute_gradients` 构造反向传播的子图。

```
sgd = tf.train.GradientDescentOptimizer(0.01)
grads_and_vars = sgd.compute_gradients(loss)
```

构造算法

反向子图的构建算法可以形式化地描述为：

```
def gradients(loss, grad=I):
  vrg = build_virtual_reversed_graph(loss)
  for op in vrg.topological_sort():
    grad_fn = ops.get_gradient_function(op)
    grad = grad_fn(op, grad)
```

首先，根据正向子图的拓扑图，构造一个虚拟的反向子图。之所以称为虚拟的，是因为真实的反向子图要比它复杂得多；更准确的说，虚拟的反向子图中的一个节点，对应于真实的反向子图中的一个局部子图。

同时，正向子图输出的最后一个节点，其输出梯度全为 1 的一个 `Tensor`，作为反向子图的初始的梯度值，常常记为 `I`。

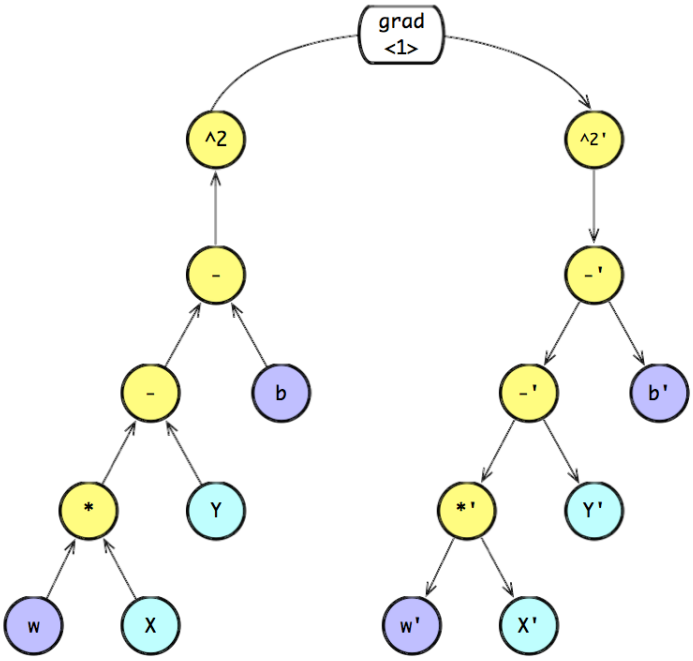


图 14-1 构造反向传播子图

接下来，根据虚拟的反向子图构造真实的反向子图。首先，根据该反向的虚拟子图执行拓扑排序算法，得到该虚拟的反向子图的一个拓扑排序；然后，按照该拓扑排序，对每个正向子图中的 OP 寻找其「梯度函数」；最后，调用该梯度函数，该梯度函数将构造该 OP 对应的反向的局部子图。

综上述，正向的一个 OP 对应反向的一个局部子图，并由该 OP 的梯度函数负责构造。当整个拓扑排序算法完成后，正向子图中的每个 OP 在反向子图中都能找到对应的局部子图。

例如，在上例中，正向图中最后一个 OP：求取平方的函数为例，讲述梯度函数的工作原理。

梯度函数原型

一般地，梯度函数满足如下原型：

```
@ops.RegisterGradient("op_name")
def op_grad_func(op, grad):
```

其中，梯度函数由 `ops.RegisterGradient` 完成注册，并放在保存梯度函数的仓库中。以后，便可以根据正向 OP 的名字，索引对应的梯度函数了。

对于一个梯度函数，第一个参数 `op` 表示正向计算的 OP，根据它可以获取正向计算时 OP 的输入和输出；第二个参数 `grad`，是反向子图中上游节点传递过来的梯度，它是一个已经计算好的梯度值 (初始梯度值全为 1)。

实战：平方函数

举个简单的例子，仅使用输入计算梯度。`y=square(x)`，用于求取 `x` 的平方。首先，构造正向计算图：

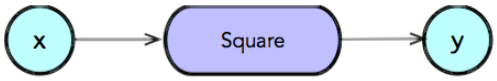


图 14-2 Square 函数：正向传播子图

然后，反向构造虚拟的反向子图。根据该虚拟的反向子图的拓扑排序，构造真正的反向计算子图。假如，当前节点为 `Square`，根据其 OP 名称，从仓库中找到对应的梯度函数 `SquareGrad`。

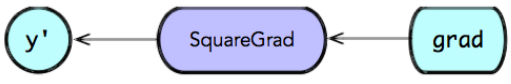


图 14-3 Square 函数：反向传播子图

因为，`y=Square(x)` 的导数为 `y'=2*x`。因此，其梯度函数 `SquareGrad` 的实现为：

```
@ops.RegisterGradient("Square")
def SquareGrad(op, grad):
  x = op.inputs[0]
  with ops.control_dependencies([grad.op]):
    x = math_ops.conj(x)
    return grad * (2.0 * x)
```

调用该梯度函数后，将得到正向 Square 的 OP，对应的反向子图 SquareGrad。它需要使用 Square 的输入，完成相应的梯度计算。

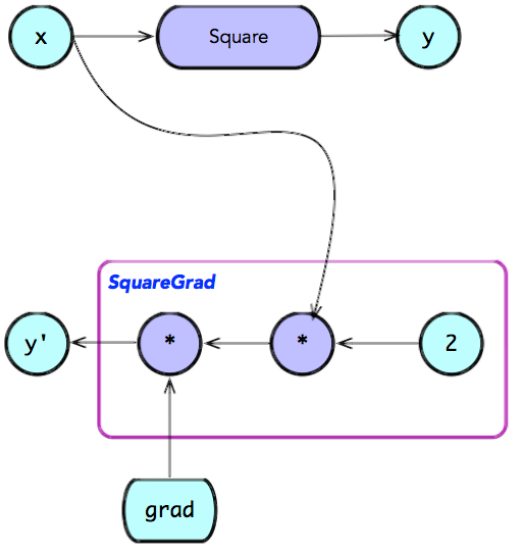


图 14-4 Square 函数：反向传播子图

一般地，正向子图中的一个 OP，对应反向子图中的一个局部子图。因为，正向 OP 的梯度函数实现，可能需要多个 OP 才能完成相应的梯度计算。例如，Square 的 OP，对应梯度函数构造了包含两个 2 个乘法 OP。

实战：指数函数

再举个简单的例子，仅使用输出计算梯度。y=exp(x)，指数函数；其导数为 y'=exp(x)，即 y'=y。因此，其梯度函数实现为：

```
@ops.RegisterGradient("Exp")
def _ExpGrad(op, grad):
  """Returns grad * exp(x)."""
  y = op.outputs[0]
  with ops.control_dependencies([grad.op]):
    y = math_ops.conj(y)
    return grad * y
```

如下图所示，正向子图中该 OP 的输出，用于对应的反向的局部子图的梯度运算。而且，该局部子图进包含一个节点。

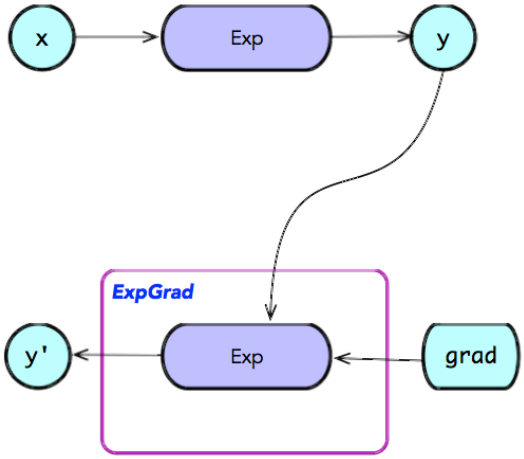


图 14-5 Exp 函数：反向传播子图

应用梯度

再做一个简单的总结，当调用 Optimizer.minimize 方法时，使用 compute_gradients 方法，实现反向计算图的构造；使用 apply_gradients 方法，实现参数更新的子图构造。

构造算法

首先，compute_gradients 在运行时将根据 loss 的值，求解 var_list=[v1, v2, ..., vn] 的梯度，最终返回的结果为：vars_and_grads = [(grad_v1, v1), (grad_v2, v2), ..., (grad_vn, vn)]。

然后，apply_gradients 迭代 grads_and_vars，对于每个 (grad_vi, vi)，构造一个更新 vi 的子图。其中，算法可以形式化地描述为：

```
def apply_gradients(grads_and_vars, learning_rate):
  for (grad, var) in grads_and_vars:
    apply_gradient_descent(learning_rate, grad, var)
```

其中，apply_gradient_descent 将构造一个使用梯度下降算法更新参数的计算子图。将 (grad, var) 的二元组，及其 learning_rate 的 Const OP 作为 ApplyGradientDescent 的输入。

ApplyGradientDescent 将应用 var <- var - learning*grad 的运算规则，实现 var 的就地更新。

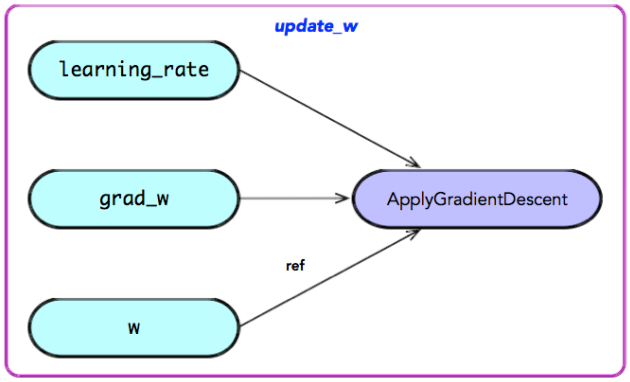


图 14-6 参数更新子图

参数更新汇总

如果存在多个训练的 Variable，最终生成多个更新参数的局部子图。它们通过一个名为 `update` 的 `NoOp`，使用控制依赖边汇总在一起。因为各个 `Variable` 之间相互独立，可以实现最大化的并发。

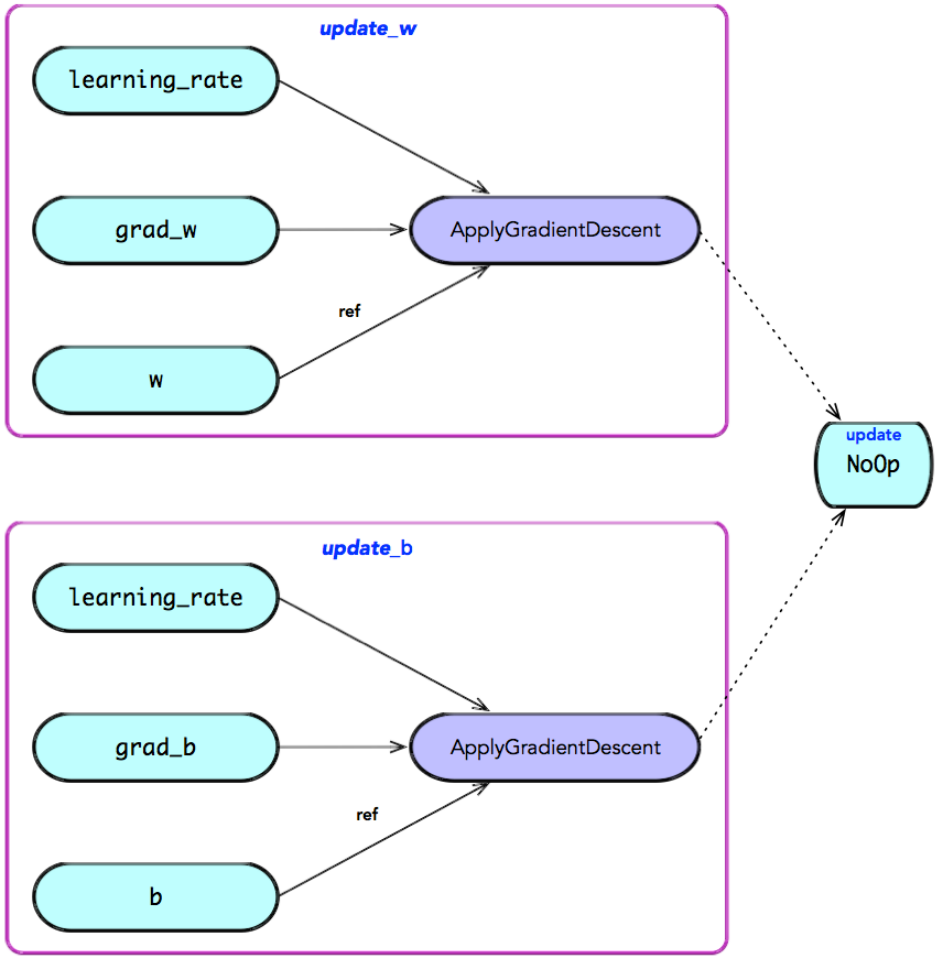


图 14-7 参数更新汇总

探秘 train_op

经过一轮 Step 运算，根据梯度，完成参数的更新，最终完成 `global_step` 加 1。而实现 `global_step` 加 1 的 OP 为 `AssignAdd`，并标记为 `train_op`；它持有 `global_step` 变量的引用，然后完成就地修改，使其值加 1。

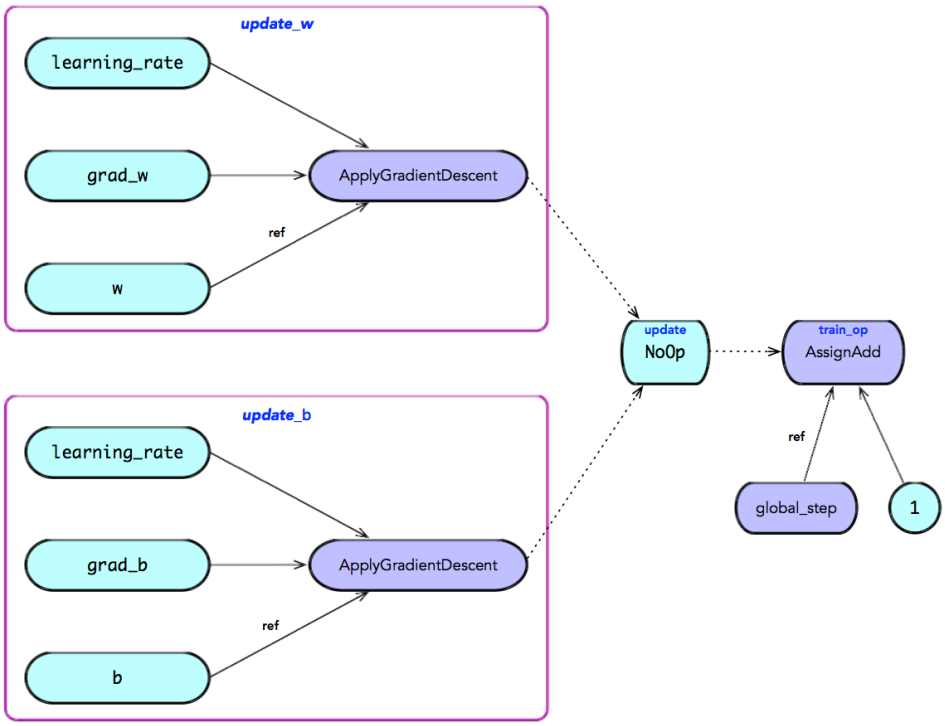


图 14-8 train_op

工作流

如图??（第??页）所示，整个训练过程，一次 Step 的训练过程由前向计算，反向梯度计算，参数更新，及其 `global_step` 计数四个基本过程组成。

其中，每一轮 Step 从开始 `Session.run` 执行开始。通过前向子图的计算，得到每个 OP 的输出，并作为下游 OP 的输入。

当前向子图完成计算后，再以初始梯度向量 I 为输入，反向计算各个训练参数的梯度，最终得到各个训练参数的梯度列表，并以 `grads_and_vars = [(grad_v1, v1), ..., (grad_vn, vn)]` 的二元组列表表示。

随后，参数更新子图以 `grads_and_vars` 为输入，执行梯度下降的更新算法；最后，通过 `train_op` 完成 `global_step` 值加 1，至此一轮 Step 执行完成。

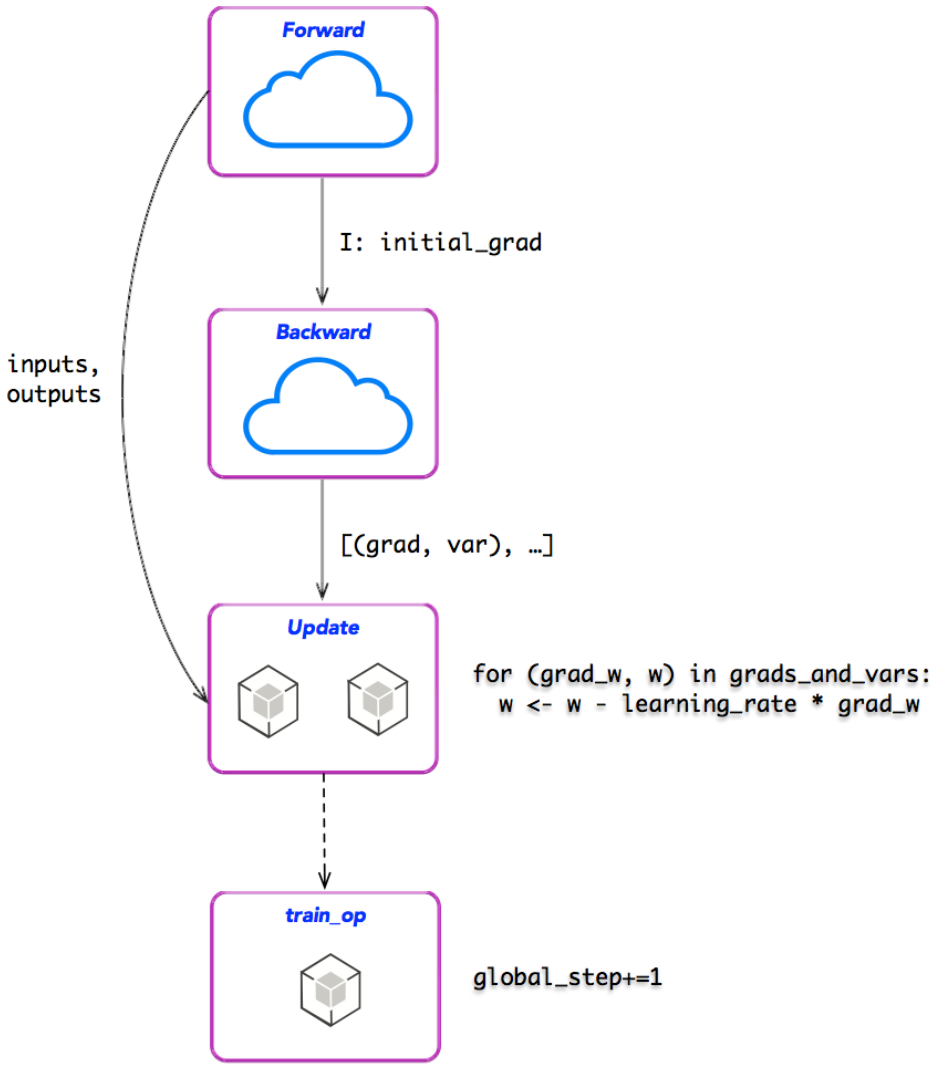


图 14-9 模型训练的工作流

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

- Martin Flower

15

数据加载

一般地，TensorFlow 输入样本数据到训练/推理子图中执行运算，存在三种读取样本数据的方法：

1. 数据注入：通过字典 feed_dict 将数据传递给 Session.run，以替代 Placeholder 的输出 Tensor 的值；
2. 数据管道：通过构造输入子图，并发地从文件中读取样本数据；
3. 数据预加载：对于小数据集，使用 Const 或 Variable 直接持有数据。

基于大型数据集的训练或推理任务，样本数据的输入常常使用数据的管道模式，确保高的吞吐率，提高训练/推理的执行效率。该过程使用队列实现输入子图与训练/推理子图之间的数据交互与异步控制。

本章将重点论述数据加载的 Pipeline 的工作机制，并深入了解 TensorFlow 并发执行的协调机制，及其队列在并发执行中扮演的角色。

15.1 数据注入

数据注入是最为常见的数据加载的方法，它通过字典 feed_dict 将样本数据传递给 Session.run，或者 Tensor.eval 方法；其中，字典的关键字为 Tensor 的名字，值为样本数据。

TensorFlow 将按照字典中 Tensor 的名字，将样本数据替换该 Tensor 的值。

```
x = tf.placeholder(tf.float32, [None, 784])
y_ = tf.placeholder(tf.float32, [None, 10])

with tf.Session():
  batch_xs, batch_ys = mnist.train.next_batch(100)
  sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

一般地，feed_dict 可以替代任何 Tensor 的值。但是，常常使用 Placeholder 表示其输出 Tensor 的值未确定，待使用 feed_dict 替代。

15.2 数据预加载

可以使用 Const 或 Variable 直接持有数据，将数据预加载至内存中，提升执行效率。该方法仅适用于小数据集，当样本数据集比较大时，内存资源消耗非常可观。这里以 mnist 数据集为例，讲解数据预加载的使用方法。

```
from tensorflow.examples.tutorials.mnist import input_data

data_sets = input_data.read_data_sets('/tmp/mnist/data')
```

使用 Const

由于 Const OP 输出 Tensor 的值是直接内联在计算图中。如果该 Const OP 在图中被使用多次，可能造成重复的冗余数据，白白浪费了不必要的内存资源。

```
with tf.name_scope('input'):
  input_images = tf.constant(data_sets.train.images)
  input_labels = tf.constant(data_sets.train.labels)
```

使用 Variable

可以使用不可变、非训练的 Variable 替代 Const。一旦初始化了该类型的 Variable，便不能改变其值，从而具备 Const 的属性。

用于数据预加载的 Variable 与用于训练的 Variable 之间存在差异，它将置位 trainable=False，系统不会将其归类于 GraphKeys.TRAINABLE_VARIABLES 集合中。在训练过程中，系统不会对其实施更新操作。

另外，在构造该类型的 Variable 时，还将设置 collections=[]，系统不会将其归类于 GraphKeys.GLOBAL_VARIABLES 集合中。在训练过程中，系统不会对其实施 Checkpoint 操作。

为了创建不可变、非训练的 Variable，此处写了一个简单的工厂方法。

```
def immutable_variable(initial_value):
  initializer = tf.placeholder(
    dtype=initial_value.dtype,
    shape=initial_value.shape)
  return tf.Variable(initializer, trainable=False, collections=[])
```

immutable_variable 使用传递进来的 initial_value 构造 Placeholder 的类型与形状信息，并以此作为 Variable 的初始值。可以使用 immutable_variable 创建不可变的，用于数据预加载的 Variable。

```
with tf.name_scope('input'):
  input_images = immutable_variable(data_sets.train.images)
  input_labels = immutable_variable(data_sets.train.labels)
```

批次预加载

可以构建 Pipeline，结合数据预加载机制，实现样本的批式加载。首先，使用 tf.train.slice_input_producer 在每个 epoch 开始时将整个样本空间随机化，每次从样本集合中随机采样获取一个训练样本。

```
def one(input_xs, input_ys, num_epochs)
  return tf.train.slice_input_producer(
    [input_xs, input_ys], num_epochs=num_epochs)
```

然后，使用 tf.train.batch 每次得到一个批次的样本数据。

```
def batch(x, y, batch_size)
  return tf.train.batch(
    [x, y], batch_size=batch_size)
```

对于使用 Variable 预加载数据，可以如下方式获取一个批次的样本数据。

```
with tf.name_scope('input'):
  input_images = immutable_variable(data_sets.train.images)
  input_labels = immutable_variable(data_sets.train.labels)

  image, label = one(input_images, input_labels, epoch=1)
  batch_images, batch_labels = batch(image, label, batch_size=100)
```

事实上，tf.train.slice_input_producer 将构造样本队列，通过 QueueRunner 并发地通过执行 Enqueue 操作，将训练样本逐一加入到样本队列中去。在每次迭代训练启动时，通过调用 DequeueMany 一次性获取 batch_size 个的批次样本数据到训练子图中去。

15.3 数据管道

一个典型的数据加载的 Pipeline(Input Pipeline)，包括如下几个重要数据处理实体：

1. 文件名称队列：将文件名称的列表加入到该队列中；

2. 读取器：从文件名称队列中读取文件名 (出队)；并根据数据格式选择相应的文件读取器，解析文件的记录；

3. 解码器：解码文件记录，并转换为数据样本；

4. 预处理器：对数据样本进行预处理，包括正则化，白化等；

5. 样本队列：将处理后的样本数据加入到样本队列中。

以 mnist 数据集为例，假如数据格式为 TFRecord。首先，使用 tf.train.string_input_producer 构造了一个持有文件名列表的 FIFOQueue 队列 (通过执行 EnqueueMany OP)，并且在每个 epoch 周期内实现文件名列表的随机化。

构建文件名队列

```
def input_producer(num_epochs):
  return tf.train.string_input_producer(
    ['/tmp/mnist/train.tfrecords'], num_epochs=num_epochs)
```

构造好了文件名队列之后，使用 tf.TFRecordReader 从文件名队列中获取文件名 (出队，通过调用执行 Dequeue OP)，并从文件中读取样本记录 (Record)。然后，使用 tf.parse_single_example 解析出样本数据。

读取器

```
def parse_record(filename_queue):
  reader = tf.TFRecordReader()
  _, serialized_example = reader.read(filename_queue)
  features = tf.parse_single_example(
      serialized_example,
      features={
          'image_raw': tf.FixedLenFeature([], tf.string),
          'label': tf.FixedLenFeature([], tf.int64),
      })
  return features
```

解码器

接着对样本数据进行解码，及其可选的预处理过程，最终得到训练样本。

```
def decode_image(features):
  image = tf.decode_raw(features['image_raw'], tf.uint8)
  image.set_shape([28*28])

  # Convert from [0, 255] -> [-0.5, 0.5] floats.
  image = tf.cast(image, tf.float32) * (1. / 255) - 0.5
  return image
```

```
def decode_label(features):
  label = tf.cast(features['label'], tf.int32)
  return label

def one_example(features):
  return decode_image(features), decode_label(features)
```

构建样本队列

可以使用 `tf.train.shuffle_batch` 构建一个 `RandomShuffleQueue` 队列，将解析后的训练样本追加在该队列中 (通过执行 `Enqueue OP`)；当迭代执行启动时，将批次获取 `batch_size` 个样本数据 (通过执行 `DequeueMany OP`)。

```
def shuffle_batch(image, label, batch_size):
    # Shuffle the examples and collect them into batch_size
    # batches.(Uses a RandomShuffleQueue)
    images, labels = tf.train.shuffle_batch(
      [image, label], batch_size=batch_size, num_threads=2,
      capacity=1000 + 3 * batch_size,
      # Ensures a minimum amount of shuffling of examples.
      min_after_dequeue=1000)
    return images, labels
```

输入子图

最后，将整个程序传接起来便构造了一个输入子图。

```
def inputs(num_epochs, batch_size):
  with tf.name_scope('input'):
    filename_queue = input_producer(num_epochs)
    features = parse_record(filename_queue)
    image, label = one_example(features)
    return shuffle_batch(image, label, batch_size)
```

15.4 数据协同

事实上，数据加载的 Pipeline 其本质是构造一个输入子图，实现并发 IO 操作，使得训练过程不会因操作 IO 而阻塞，从而实现 GPU 的利用率的提升。

对于输入子图，数据流的处理划分为若干阶段 (Stage)，每个阶段完成特定的数据处理功能；各阶段之间以队列为媒介，完成数据的协同和交互。

如下图所示，描述了一个典型的神经网络的训练模式。整个流水线由两个队列为媒介，将其划分为 3 个阶段。

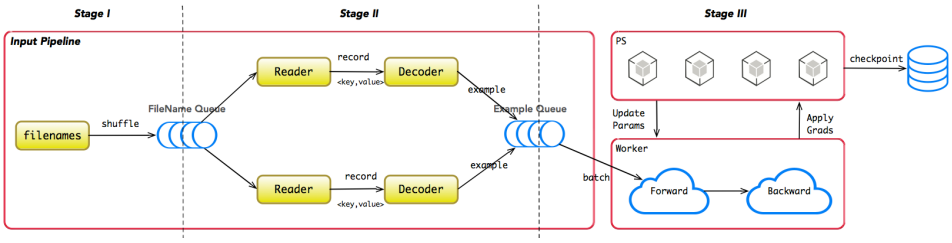


图 15-1 模型训练工作流

阶段 1

`string_input_producer` 构造了一个 `FIFOQueue` 的队列，它是一个有状态的 OP。根据 `shuffle` 选项，在每个 epoch 开始时，随机生成文件列表，并将其一同追加至队列之中。

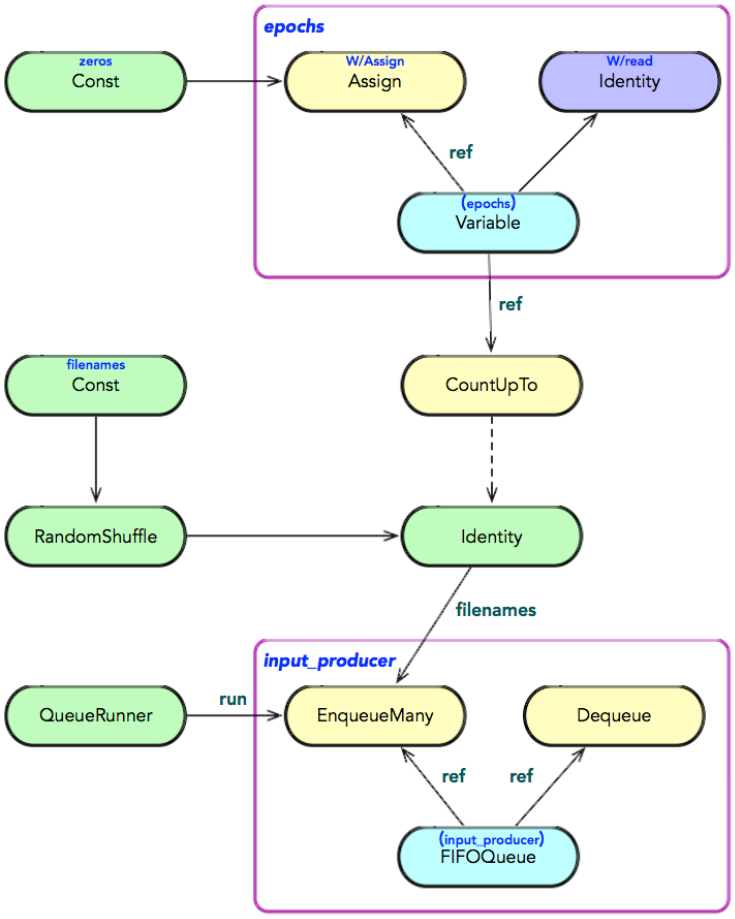


图 15-2 阶段 1：模型训练工作流

随机化

首先，执行名为 `filenames` 的 `Const` OP，再经过 `RandomShuffle` 将文件名称列表随机化。

Epoch 控制

为了实现 epoch 的计数，实现巧妙地设计了一个名为 epochs 的本地变量。其中，本地变量仅对本进程的多轮 Step 之间共享数据，并且不会被训练子图实施更新。

在 Session.run 之前，系统会执行本地变量列表的初始化，将名为 epochs 的 Variable 实施零初始化。

epoch 的计数功能由 CountUpTo 完成，它的工作原理类似于 C++ 的 i++。它持有 Variable 的引用，及其上限参数 limit。每经过一轮 epoch，使其 Variable 自增 1，直至达到 num_epochs 数目。

其中，当 epoch 数到达 num_epochs 时，CountUpTo 将自动抛出 OutOfRangeError 异常。详细实现可以参考 CountUpToOp 的 Kernel 实现。

```
template <class T>
struct CountUpToOp : OpKernel {
  explicit CountUpToOp(OpKernelConstruction* ctxt)
    : OpKernel(ctxt) {
    OP_REQUIRES_OK(ctxt, ctxt->GetAttr("limit", &limit_));
  }

  void Compute(OpKernelContext* ctxt) override {
    T before_increment;
    {
      mutex_lock l(*ctxt->input_ref_mutex(0));

      // Fetch the old tensor
      Tensor tensor = ctxt->mutable_input(0, true);
      T* ptr = &tensor.scalar<T>()();
      before_increment = *ptr;

      // throw OutOfRangeError if exceed limit
      if (*ptr >= limit_) {
        ctxt->SetStatus(errors::OutOfRange(
            "Reached limit of ", limit_));
        return;
      }
      // otherwise increase 1
      ++*ptr;
    }
    // Output if no error.
    Tensor* out_tensor;
    OP_REQUIRES_OK(ctxt, ctxt->allocate_output(
        "output", TensorShape({}), &out_tensor));
    out_tensor->scalar<T>()() = before_increment;
  }

private:
  T limit_;
};
```

入队操作

事实上，将文件名列表追加到队列中，执行的是 EnqueueMany，类似于 Assign 修改 Variable 的值，EnqueueMany 也是一个有状态的 OP，它持有队列的句柄，直接完成队列的状态更新。

在此处，EnqueueMany 将被 Session.run 执行，系统反向遍历，找到依赖的 Identity，发现控制依赖于 CountUpTo，此时会启动一次 epoch 计数，直至到达 num_epoch 数目抛出 OutOfRangeError 异常。同时，Identity 依赖于 RandomShuffle，以便得到随机化了的文件名列表。

QueueRunner

另外，在调用 tf.train.string_input_producer 时，将往计算图中注册一个特殊的 OP：QueueRunner，并且将其添加到 GraphKeys.QUEUE_RUNNERS 集合中。并且，一个 QueueRunner 持有一个或多个 Enqueue，EnqueueMany 类型的 OP。

阶段 2

Reader 从文件名队列中按照 FIFO 的顺序获取文件名，并按照文件名读取文件记录，成功后对该记录进行解码和预处理，将其转换为数据样本，最后将其追加至样本队列中。

读取器

事实上，实现构造了一个 ReaderRead 的 OP，它持有文件名队列的句柄，从队列中按照 FIFO 的顺序获取文件名。

因为文件的格式为 TFRecord，ReaderRead 将委托调用 TFRecordReader 的 OP，执行文件的读取。最终，经过 ReaderRead 的运算，将得到一个序列化了的样本。

解码器

得到序列化了的样本后，将使用合适的解码器实施解码，从而得到一个期望的样本数据。可选地，可以对样本实施预处理，例如 reshape 等操作。

入队操作

得到样本数据后，将启动 QueueEnqueue 的运算，将样本追加至样本队列中去。其中，QueueEnqueue 是一个有状态的 OP，它持有样本队列的句柄，直接完成队列的更新操作。

实施上，样本队列是一个 RandomShuffleQueue，使用出队操作实现随机采样。

并发执行

为了提高 IO 的吞吐率，可以启动多路并发的 Reader 与 Decoder 的工作流，并发地将样本追加至样本队列中去。其中，RandomShuffleQueue 是线程安全的，支持并发的入队或出队操作。

阶段 3

当数据样本累计至一个 batch_size 时，训练/推理子图将取走该批次的样本数据，启动一次迭代计算 (常称为一次 Step)。

出队操作

事实上，训练子图使用 DequeueMany 获取一个批次的样本数据。

迭代执行

一般地，一次迭代运行，包括两个基本过程：前向计算与反向梯度传递。Worker 任务使用 PS 任务更新到本地的当前值，执行前向计算得到本次迭代的损失。

然后，根据本次迭代的损失，反向计算各个 Variable 的梯度，并更新到 PS 任务中；PS 任务更新各个 Variable 的值，并将当前值广播到各个 Worker 任务上去。

Checkpoint

PS 任务根据容错策略，周期性地实施 Checkpoint。将当前所有 Variable 的数据，及其图的元数据，包括静态的图结构信息，持久化到外部存储设备上，以便后续恢复计算图，及其所有 Variable 的数据。

Pipeline 节拍

例如，往 FIFOQueue 的队列中添加文件名称列表，此时调用 EnqueueMany 起始的子图计算，其中包括执行所依赖的 CountUpTo。当 CountUpTo 达到 limit 上限时，将自动抛出 OutOfRangeError 异常。

扮演主程序的 QueueRunner，捕获 coord.join 重新抛出的 OutOfRangeError 异常，随后立即关闭相应的队列，并且退出该线程的执行。队列被关闭之后，入队操作将变为非法；而出队操作则依然合法，除非队列元素为空。

同样的道理，下游 OP 从队列 (文件名队列) 中出队元素，一旦该队列元素为空，则自动抛出 OutOfRangeError 异常。该阶段对应的 QueueRunner 将感知该异常的发生，然后捕获异常并关闭下游的队列 (样本队列)，退出线程的执行。

在 Pipeline 的最后阶段，train_op 从样本队列中出队批次训练样本时，队列为空，并且队列被关闭了，则抛出 OutOfRangeError 异常，最终停止整个训练任务。

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

- Martin Flower

16 Saver

16.1 Saver

在长期的训练任务过程中，为了实现任务的高可用性，TensorFlow 会周期性地执行断点检查 (Checkpoint)。

Saver 是实现断点检查功能的基础设施，它会将所有的训练参数持久化在文件系统中；当需要恢复训练时，可以载从文件系统中恢复计算图，及其训练参数的值。也就是说，Saver 承担如下两个方面的职责：

1. save: 将训练参数的当前值持久化到断点文件中；
2. restore: 从断点文件中恢复训练参数的值。

使用方法

例如，存在一个简单的计算图，包含两个训练参数。首先，执行初始化后，将其结果持久化到文件系统中。

```
# construct graph
v1 = tf.Variable([0], name='v1')
v2 = tf.Variable([0], name='v2')

# run graph
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  saver = tf.train.Saver()
  saver.save(sess, 'ckp')
```

随后，可以根据断点文件存储的位置恢复模型。

```
with tf.Session() as sess:
  saver = tf.import_meta_graph('ckp.meta')
  saver.restore(sess, 'ckp')
```

文件功能

当执行 Saver.save 操作之后，在文件系统中生成如下文件：

```
□□□ checkpoint
□□□ ckp.data-00000-of-00001
□□□ ckp.index
□□□ ckp.meta
```

索引文件

索引 (index) 文件保存了一个不可变表 (tensorflow::table::Table) 的数据；其中，关键字为 Tensor 的名称，其值描述该 Tensor 的元数据信息，包括该 Tensor 存储在哪个数据 (data) 文件中，及其在该数据文件中的偏移，及其校验和等信息。

数据文件

数据 (data) 文件记录了所有变量 (Variable) 的值。当 restore 某个变量时，首先从索引文件中找到相应变量在哪个数据文件，然后根据索引直接获取变量的值，从而实现变量数据的恢复。

元文件

元文件 (meta) 中保存了 MetaGraphDef 的持久化数据，它包括 GraphDef，SaverDef 等元数据。

将描述计算图的元数据与存储变量值的数据文件相分离，实现了静态的图结构与动态的数据表示的分离。因此，在恢复 (Restore) 时，先调用 tf.import_meta_graph 先将 GraphDef 恢复出来，然后再恢复 SaverDef，从而恢复了描述静态图结构的 Graph 对象，及其用于恢复变量值的 Saver 对象，最后使用 Saver.restore 恢复所有变量的值。

这也是在上例中，在调用 Saver.restore 之前，得先调用 tf.import_meta_graph 的真正原因；否则，缺失计算图的实例，就无法谈及恢复数据到图实例中了。

状态文件

Checkpoint 文件会记录最近一次的断点文件 (Checkpoint File) 的前缀，根据前缀可以找对对应的索引和数据文件。当调用 `tf.train.latest_checkpoint`，可以快速找到最近一次的断点文件。

此外，Checkpoint 文件也记录了所有的断点文件列表，并且文件列表按照由旧至新的时间依次排序。当训练任务时间周期非常长，断点检查将持续进行，必将导致磁盘空间被耗尽。为了避免这个问题，存在两种基本的方法：

1. `max_to_keep`: 配置最近有效文件的最大数目，当新的断点文件生成时，且文件数目超过 `max_to_keep`，则删除最旧的断点文件；其中，`max_to_keep` 默认值为 5;

2. `keep_checkpoint_every_n_hours`: 在训练过程中每 `n` 小时做一次断点检查，保证只有一个断点文件；其中，该选项默认是关闭的。

由于 Checkpoint 文件也记录了断点文件列表，并且文件列表按照由旧至新的时间依次排序。根据上述策略删除陈旧的断点文件将变得极其简单有效。

模型

持久化模型

为了实现持久化的功能，`Saver` 在构造时在计算图中插入 `SaveV2`，及其关联的 OP。其中，`file_name` 为一个 Const 的 OP，指定断点文件的名称；`tensor_names` 也是一个 Const 的 OP，用于指定训练参数的 Tensor 名称列表。

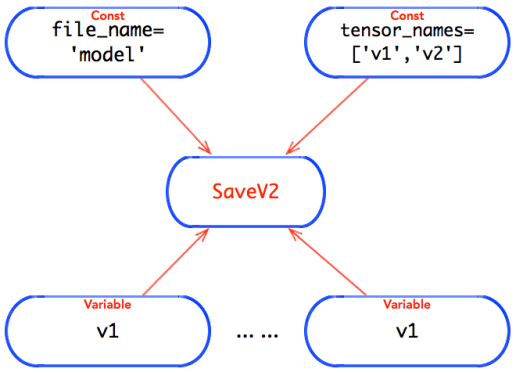


图 16-1 Saver：持久化模型

恢复模型

同样地，为了实现恢复功能，Saver 在构造期，为每个训练参数，插入了一个 RestoreV2，及其关联的 OP。其中，包括从断点文件中恢复参数默认值的初始化器 (Initializer)，其本质是一个 Assign 的 OP。

另外，file_name 为一个 Const 的 OP，指定断点文件的名称；tensor_names 也是一个 Const 的 OP，用于指定训练参数的 Tensor 名称列表，其长度为 1。

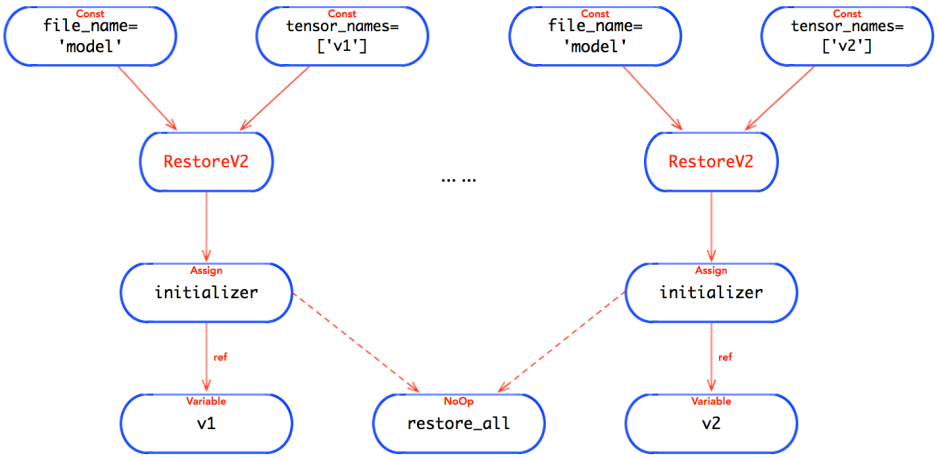


图 16-2 Saver：恢复模型

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

- Martin Flower

17

MonitoredSession

训练一个简单的模型，可以通过运行 `train_op` 数次直至模型收敛，最终将训练参数实施 Checkpoint，持久化训练模型。对于小规模的学习模型，这个过程至多需要花费数小时的时间。

但是，对于大规模的学习模型，需要花费数天时间；而且可能需要使用多份复本 (replica)，此时需要更加健壮的训练过程支持模型的训练。因此，需要解决三个基本问题：

1. 当训练过程异常关闭，或程序崩溃，能够合理地处理异常；
2. 当异常关闭，或程序崩溃之后，能够恢复训练过程；
3. 能够通过 TensorBoard 监控整个训练过程。

当训练被异常关闭或程序崩溃之后，为了能够恢复训练过程，必须周期性实施 Checkpoint。当训练过程重启后，可以通过寻找最近一次的 Checkpoint 文件，恢复训练过程。

为了能够使用 TensorBoard 监控训练过程，可以通过周期性运行一些 Summary 的 OP，并将结果追加到事件文件中。TensorBoard 能够监控和解析事件文件的数据，可视化整个训练过程，包括展示计算图的结构。

17.1 引入 MonitoredSession

`tf.train.MonitoredSession`，它可以定制化 `Hook`，用于监听整个 `Session` 的生命周期；内置 `Coordinator` 对象，用于协调所有运行中的线程同时停止，并监听，上报和处理异常；当发生 `AbortedError` 或 `UnavailableError` 异常时，可以重启 `Session`。

使用方法

一般地，首先使用 ChiefSessionCreator 创建 Session 实例，并且注册三个最基本的 tf.train.SessionRunHook：

1. CheckpointSaverHook：周期性地 Checkpoint；
2. SummarySaverHook：周期性地运行 Summary；
3. StepCounterHook：周期性地统计每秒运行的 Step 数目。

为了能够安全处理异常，并且能够关闭 MonitoredSession，常常使用 with 的上下文管理器。

```
session_creator = tf.train.ChiefSessionCreator(
  checkpoint_dir=checkpoint_dir,
  master=master,
  config=config)

hooks = [
  tf.train.CheckpointSaverHook(
    checkpoint_dir=checkpoint_dir,
    save_secs=save_checkpoint_secs),
  tf.train.SummarySaverHook(
    save_secs=save_summaries_secs,
    output_dir=checkpoint_dir),
  tf.train.StepCounterHook(
    output_dir=checkpoint_dir,
    every_n_steps=log_step_count_steps)
]

with tf.train.MonitoredSession(
  session_creator=session_creator,
  hooks=hooks) as sess:
  if not sess.should_stop():
    sess.run(train_op)
```

使用工厂

使用 MonitoredTrainingSession 的工厂方法，可以简化 MonitoredSession 的创建过程。

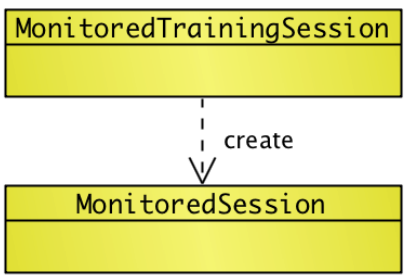


图 17-1 MonitoredTrainingSession：工厂方法

```
with MonitoredTrainingSession(
  master=master,
  is_chief=is_chief,
  checkpoint_dir=checkpoint_dir
  config=config) as sess:
  if not sess.should_stop():
    sess.run(train_op)
```

装饰器

为了得到复合功能的 MonitoredSession，可以将完成子功能的 WrappedSession 进行组合拼装。

1. RecoverableSession：当发生 AbortedError 或 UnavailableError 异常时，可以恢复和重建 Session；
2. CoordinatedSession：内置 Coordinator 对象，用于协调所有运行中的线程同时停止，并监听，上报和处理异常；
3. HookedSession：可以定制化 Hook，用于监听整个 Session 的生命周期。

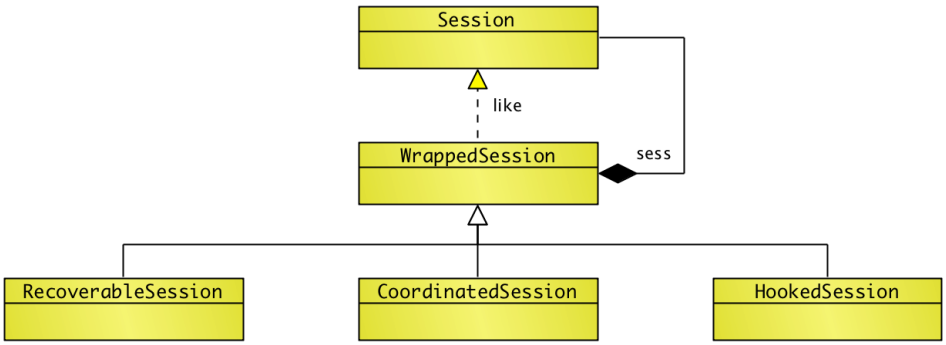


图 17-2 MonitoredSession：装饰器

最终，可以组合三者的特性，构建得到 MonitoredSession(伪代码实现，详情请查阅 MonitoredSession 的具体实现)。

```
MonitoredSession(
  RecoverableSession(
    CoordinatedSession(
      HookedSession(
        tf.Session(target, config)))))
```

17.2 生命周期

MonitoredSession 具有 Session 的生命周期特征 (但并非 IS-A 关系，而是 Like-A 关系，这是一种典型的按照鸭子编程的风格)。

在生命周期过程中，插入了 SessionRunHook 的回调钩子，用于监听 MonitoredSession 的生命周期过程。

初始化

在初始化阶段，MonitoredSession 主要完成如下过程：

1. 运行所有回调钩子的 begin 方法；
2. 通过调用 scaffold.finalize() 冻结计算图；
3. 创建会话：使用 SessionCreator 多态创建 Session
4. 运行所有回调钩子的 after_create_session 方法

其中，使用 SessionCreator 多态创建 Session 的过程，存在两种类型。

1. ChiefSessionCreator：调用 SessionManager.prepare_session，通过从最近的 Checkpoing 恢复模型，或运行 init_op，完成模型的初始化；然后，启动所有 QueueRunner 实例；
2. WorkerSessionCreator：调用 SessionManager.wait_for_session，等待 Chief 完成模型的初始化。

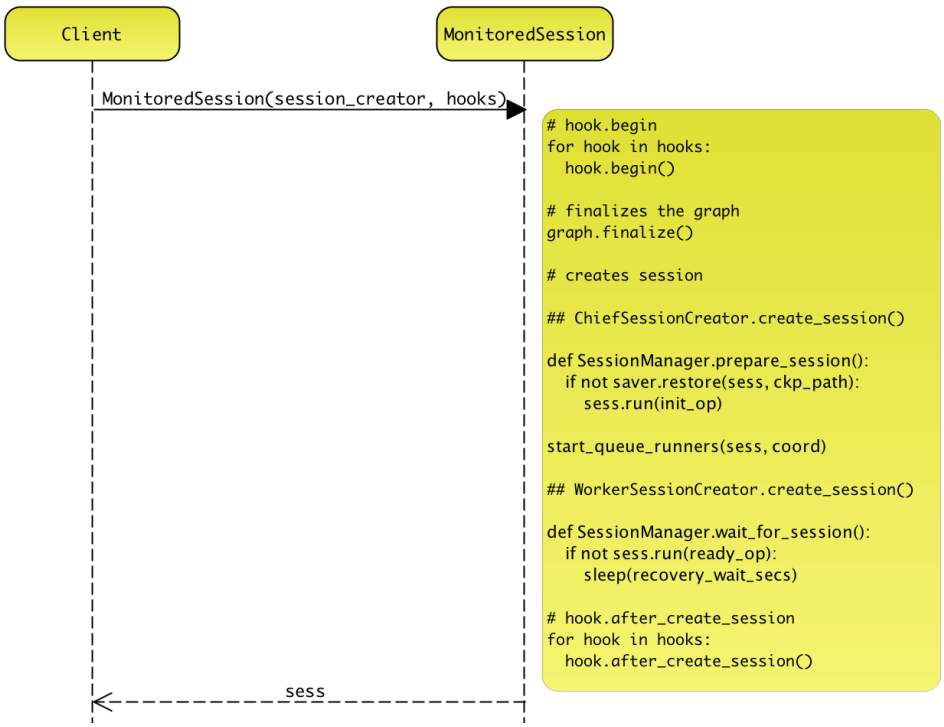


图 17-3 MonitoredSession：初始化

执行

在执行阶段，在运行 `Session.run` 前后分别回调钩子的 `before_run` 和 `after_run` 方法。如果在运行过程发生了 `AbortedError` 或 `UnavailableError` 异常，则重启会话服务。

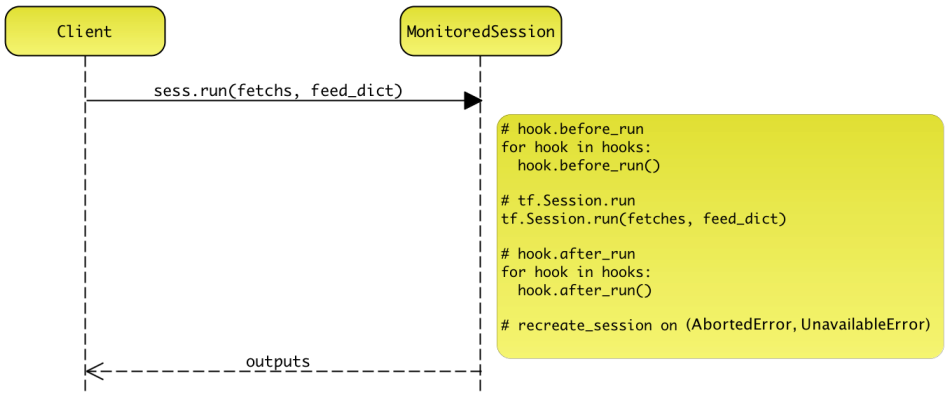


图 17-4 MonitoredSession：执行

关闭

当训练过程结束后，通过调用 `close` 方法，关闭 `MonitoredSession`，释放系统的计算资源。

此时，将回调钩子的 `end` 方法，并且会通过调用 `Coordinator.request_stop` 方法，停止所有 `QueueRunner` 实例。最终，听过调用 `tf.Session.close` 方法，释放系统的资源。

另外，如果发生 `OutOfRangeError` 异常，`MonitoredSession` 认为训练过程正常终止，并忽略该异常。

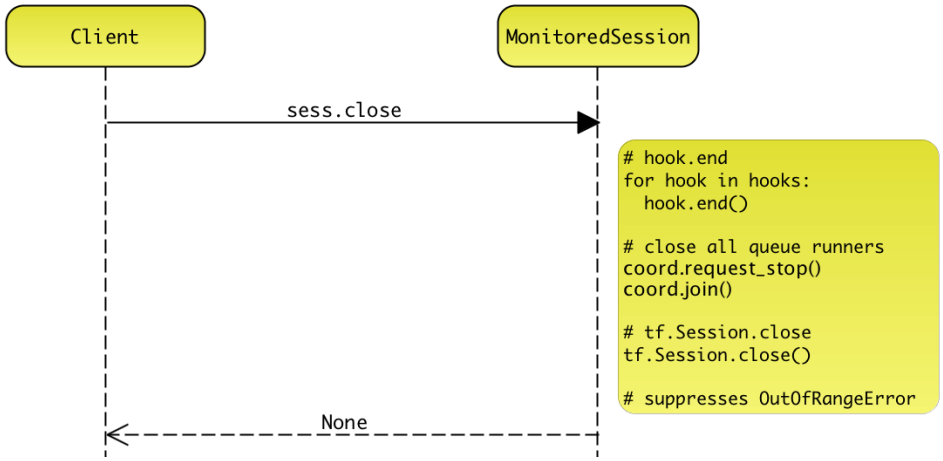


图 17-5 MonitoredSession：关闭

17.3 模型初始化

MonitoredSession 在初始化时，使用 SessionCreator 完成会话的创建和模型的初始化。

一般地，在分布式环境下，存在两中类型的 Worker:

1. Chief: 负责模型的初始化；
2. Non-Chief: 等待 Chief 完成模型的初始化。

两者之间，通过一个简单的协调协议共同完成模型的初始化。

协调协议

对于 Chief，它会尝试从 Checkpoint 文件中恢复模型；如果没有成功，则会通过执行 init_op 全新地初始化模型；其初始化算法，可以形式化描述为：

```
def prepare_session(master, init_op, saver, ckp_dir):
  if is_chief():
    sess = tf.Session(master)
    sess.run(init_op) if not saver.restore(sess, ckp_dir)
```

对于 Non-Chief，它会周期性地通过运行 ready_op，查看 Chief 是否已经完成模型的初始化。

```
def wait_for_session(master, ready_op, recovery_wait_secs):
  while True:
    sess = tf.Session(master)
    if sess.run(ready_op):
      return sess
    else:
      sess.close()
      time.sleep(recovery_wait_secs)
```

SessionManager

事实上，上述算法主要由 SessionManager 实现，它主要负责从 Checkpoint 文件中完成模型的恢复，或直接通过运行 init_op 完成模型的初始化，最终创建可工作的 Session 实例。

1. 对于 Chief，通过调用 prepare_session 方法，完成模型的初始化；
2. 对于 Non-Chief，通过调用 wait_for_session 方法，等待 Chief 完成模型的初始化。

详情可以参考 SessionManager 的具体实现。

引入工厂

使用工厂方法，分别使用 `ChiefSessionCreator` 和 `WorkerSessionCreator` 分别完成上述算法。

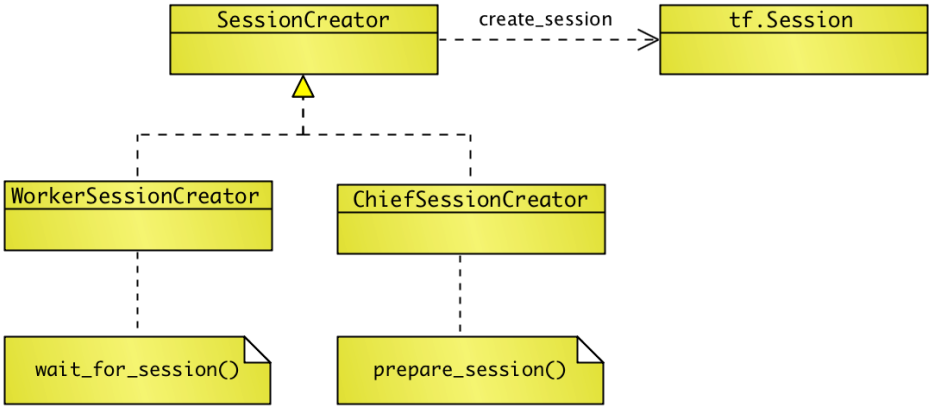


图 17-6 SessionManager

Scaffold

当要构建一个模型训练，需要 `init_op` 初始化变量；需要 `Saver` 周期性实施 Checkpoint；需要 `ready_op` 查看一个模型是否已经初始化完毕；需要 `summary_op` 搜集所有 Summary，用于训练过程的可视化。

一般地，在计算图中通过 `GraphKey` 标识了这些特殊的 OP 或对象，以便可以从计算图中检索出这些特殊的 OP 或对象。

在训练模型的特殊领域中，提供了一个基础工具库：`Scaffold`，用于创建这些 OP 或对象的默认值，并添加到计算图的集合中，并且 `Scaffold` 提供了查询接口可以方便地获取到这些 OP 或对象。

可以通过调用 `Scaffold.finalize` 方法，如果对应的 OP 或对象为 `None`，则默认创建该类型的实例。最终冻结计算图，之后禁止再往图中增加节点。

```
class Scaffold(object):
  def finalize(self):
    """Creates operations if needed and finalizes the graph."""

    # create init_op
    if self._init_op is None:
      def default_init_op():
        return control_flow_ops.group(
            variables.global_variables_initializer(),
            resources.initialize_resources(
              resources.shared_resources()))
      self._init_op = Scaffold.get_or_default(
          'init_op',
```

```
          ops.GraphKeys.INIT_OP,
          default_init_op)

    # create ready_op
    if self._ready_op is None:
      def default_ready_op():
        return array_ops.concat([
            variables.report_uninitialized_variables(),
            resources.report_uninitialized_resources()
        ], 0)
      self._ready_op = Scaffold.get_or_default(
          'ready_op',
          ops.GraphKeys.READY_OP,
          default_ready_op)

    # create ready_for_local_init_op
    if self._ready_for_local_init_op is None:
      def default_ready_for_local_init_op():
        return variables.report_uninitialized_variables(
            variables.global_variables())
      self._ready_for_local_init_op = Scaffold.get_or_default(
          'ready_for_local_init_op',
          ops.GraphKeys.READY_FOR_LOCAL_INIT_OP,
          default_ready_for_local_init_op)

    # create local_init_op
    if self._local_init_op is None:
      def _default_local_init_op():
        return control_flow_ops.group(
            variables.local_variables_initializer(),
            lookup_ops.tables_initializer())
      self._local_init_op = Scaffold.get_or_default(
          'local_init_op',
          ops.GraphKeys.LOCAL_INIT_OP,
          _default_local_init_op)

    # create summary_op
    if self._summary_op is None:
      self._summary_op = Scaffold.get_or_default(
          'summary_op',
          ops.GraphKeys.SUMMARY_OP,
          summary.merge_all)

    # create Saver
    if self._saver is None:
      self._saver = training_saver._get_saver_or_default()
    self._saver.build()

    ops.get_default_graph().finalize()
    return self
```

从 `finalize` 的实现可以看出，以下 OP 完成的功能为：

1. `init_op`：完成所有全局变量和全局资源的初始化；
2. `local_init_op`：完成所有本地变量和表格的初始化；
3. `ready_op`：查看所有的全局变量和全局资源是否已经初始化了；否则报告未初始化的全局变量和全局资源的列表；
4. `ready_for_local_init_op`：查看所有的本地变量和表格是否已经初始化了；否则报告未初始化的本地变量和表格的列表；
5. `summary_op`：汇总所有 Summary 的输出；

其中，本地变量不能持久化到 Checkpoint 文件中；当然，也就不能从 Checkpoint 文件

中恢复本地变量的值。

初始化算法

通过观测上面的 OP 的定义，理解 `prepare_session` 模型初始化的完整语义便不是那么困难了。

```
class SessionManager(object):
  def prepare_session(self,
                      master,
                      saver=None
                      checkpoint_filename=None
                      init_op=None,
                      init_feed_dict=None,
                      init_fn=None):
    """Creates a Session. Makes sure the model is ready."""

    def _restore_checkpoint():
      sess = session.Session(master)
      if not saver or not checkpoint_filename):
        return sess, False
      else:
        saver.restore(sess, checkpoint_filename)
        return sess, True

    def _try_run_init_op(sess):
      if init_op is not None:
        sess.run(init_op, feed_dict=init_feed_dict)
      if init_fn:
        init_fn(sess)

    sess, is_succ = self._restore_checkpoint()
    if not is_succ:
      _try_run_init_op(sess)
    self._try_run_local_init_op(sess)
    self._model_ready(sess)
    return sess
```

其初始化算法非常简单。首先，尝试从 Checkpoint 文件中恢复 (此处为了简化问题，省略了部分实现)；如果失败，则调用 `init_op` 和 `init_fn` 完成全局变量和资源的初始化；然后，才能实施本地变量和表格的初始化；最后，验证所有全局变量和资源是否已经初始化了。

本地变量初始化

对于非空的 `local_init_op`，必须等所有全局变量已经初始化完毕后才能进行初始化 (通过调用 `_ready_for_local_init_op`)；否则，报告未初始化的全局变量列表到 `msg` 字段中。

也就是说，本地变量初始化在全局变量初始化之后，且本地变量不会持久化到 Checkpoint 文件中。

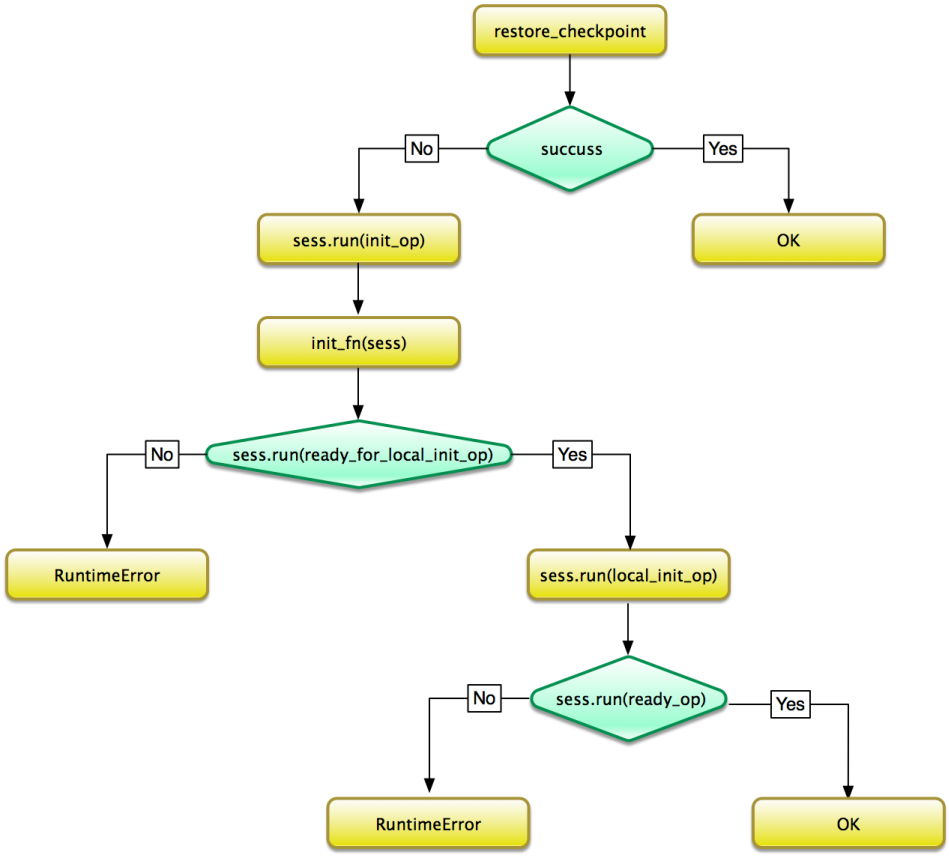


图 17-7 模型初始化算法

```
class SessionManager(object):
  def _ready_for_local_init(self, sess):
    """Checks if the model is ready to run local_init_op.
    """
    return _ready(self._ready_for_local_init_op, sess,
                  "Model not ready for local init")

  def _try_run_local_init_op(self, sess):
    """Tries to run _local_init_op, if not None,
       and is ready for local init.
    """
    if not self._local_init_op:
      return True, None:

    is_ready, msg = self._ready_for_local_init(sess)
    if is_ready:
      sess.run(self._local_init_op)
      return True, None
    else:
      return False, msg
```

验证模型

最后，通过执行 `_ready_op`，查看所有全局变量和全局资源是否都已经初始化了；否则，报告未初始化的变量列表到 `msg` 字段中。

```
class SessionManager(object):
  def _model_ready(self, sess):
    """Checks if the model is ready or not.
    """
    return _ready(self._ready_op, sess, "Model not ready")
```

其中，`_ready` 使用函数，用于运行相应的 `ready_op`，查看相应的变量或资源是否完成初始化。

```
def _ready(op, sess, msg):
  """Checks if the model is ready or not, as determined by op.
  """
  if op is None:
    return True, None

  ready_value = sess.run(op)
  if (ready_value.size == 0):
    return True, None
  else:
    uninitialized_vars = ", ".join(
        [i.decode("utf-8") for i in ready_value])
    return False, "initialized vars: " + uninitialized_vars
```

17.4 异常安全

一般地，常常使用 with 的上下文管理器，实现 MonitoredSession 的异常安全和资源安全释放。

上下文管理器

当退出 with 语句后，将停止运行所有 QueueRunner 实例，并实现 tf.Session 的安全关闭。

```
class _MonitoredSession(object):
  def __exit__(self, exception_type, exception_value, traceback):
    if exception_type in [errors.OutOfRangeError, StopIteration]:
      exception_type = None
    self._close_internal(exception_type)
    return exception_type is None

  def _close_internal(self, exception_type=None):
    try:
      if not exception_type:
        for h in self._hooks:
          h.end(self.tf_sess)
    finally:
      try:
        self._sess.close()
      finally:
        self._sess = None
        self.tf_sess = None
        self.coord = None
```

特殊地，当发生 OutOfRangeError 或 StopIteration，则认为正常终止，忽视该异常。如果抛出了其它类型的异常，则不会调用 end 的回调钩子。

停止 QueueRunner

另外，当执行 self._sess.close()，最终将调用 _CoordinatedSession 的 close 方法。通过调用 coord.request_stop 通知所有 QueueRunner 实例停止运行，并且通过调用 coord.join 方法等待所有 QueueRunner 实例运行完毕。

```
class _CoordinatedSession(_WrappedSession):
  def close(self):
    self._coord.request_stop()
    try:
      self._coord.join()
    finally:
      try:
        _WrappedSession.close(self)
      except Exception:
        pass
```

17.5 回调钩子

可以通过定制 SessionRunHook，实现对 MonitorSession 生命周期过程的监听和管理。

```
class SessionRunHook(object):
  def begin(self):
    pass

  def after_create_session(self, session, coord):
    pass

  def before_run(self, run_context):
    return None

  def after_run(self, run_context, run_values):
    pass

  def end(self, session):
    pass
```

其中，最常见的 Hook 包括：

1. CheckpointSaverHook：周期性地 Checkpoint；
2. SummarySaverHook：周期性地运行 Summary；
3. StepCounterHook：周期性地统计每秒运行的 Step 数目。

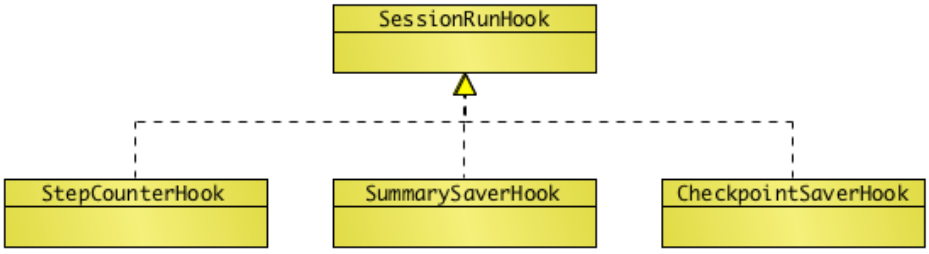


图 17-8 SessionRunHook

第 VI 部分

附录

A

代码阅读

在程序员的日常工作之中，绝大多数时间都是在**阅读代码**，而不是在写代码。但是，阅读代码往往是一件很枯燥的事情，尤其当遇到了一个不漂亮的设计，反抗的心理往往更加强烈。事实上，变换一下习惯、思路和方法，代码阅读其实是一个很享受的过程。

阅读代码的模式，实践和习惯，集大成者莫过于希腊作者 Diomidis Spinellis 的经典之作：Code Reading, The Open Source Perspective。本文从另外一个视角出发，谈谈我阅读代码的一些习惯，期待找到更多知音的共鸣。

A.1 工欲善其事，必先利其器

首先，阅读代码之前先准备好一个称心如意的工具箱，包括 IDE, UML，脑图等工具。我主要使用的编程语言包括 C++, Scala, Java, Ruby, Python；我更偏向使用 JetBrains 公司的产品，其很多习惯用法对程序员都很贴心。

其次，高效地使用快捷键，这是一个良好的代码阅读习惯，它极大地提高了代码阅读的效率和质量。例如，查看类层次关系，函数调用链，方法引用点等等。

R 拔掉鼠标，减低对鼠标的依赖。当发现没有鼠标而导致工作无法进行下去时，尝试寻找对应的快捷键。通过日常的点滴积累，工作效率必然能够得到成倍的提高。

A.2 力行而后知之真

阅读代码一种常见的反模式就是通过 Debug 的方式来阅读代码。作者不推荐这种代码阅读的方式，其一，因为运行时线程间的切换很容易导致方向的迷失；其二，了解代码调用栈对于理解系统行为并非见得有效，因为其包含太多实现细节，不易发现问题的本质。

但在阅读代码之前，有几件事情是必须做的。其一，手动地构建一次工程，并运行测试用例；其二，亲自动手写几个 Demo 感受一下。

先将工程跑起来，目的不是为了 Debug 代码，而是在于了解工程构建的方式，及其认识系统的基本结构，并体会系统的使用方式。

如果条件允许，可以尝试使用 ATDD 的方式，发现和挖掘系统的行为。通过这个过程，将自己当成一个客户，思考系统的行为，这是理解系统最重要的基石。

A.3 发现领域模型

阅读代码，不是为了了解每个类，每个函数干什么，而是为了挖掘更本质，更不易变化的知识。事实上，发现**领域模型**是阅读代码最重要的一个目标，因为领域模型是系统的灵魂所在。通过代码阅读，找到系统本质的知识，并通过自己的模式表达出来，才能真正地抓住系统的脉络，否则一切都是空谈。

例如，在阅读 TensorFlow 的 Python 实现的客户端代码时，理顺计算图的领域模型，对于理解 TensorFlow 的编程模型，及其系统运行时的行为极其重要。

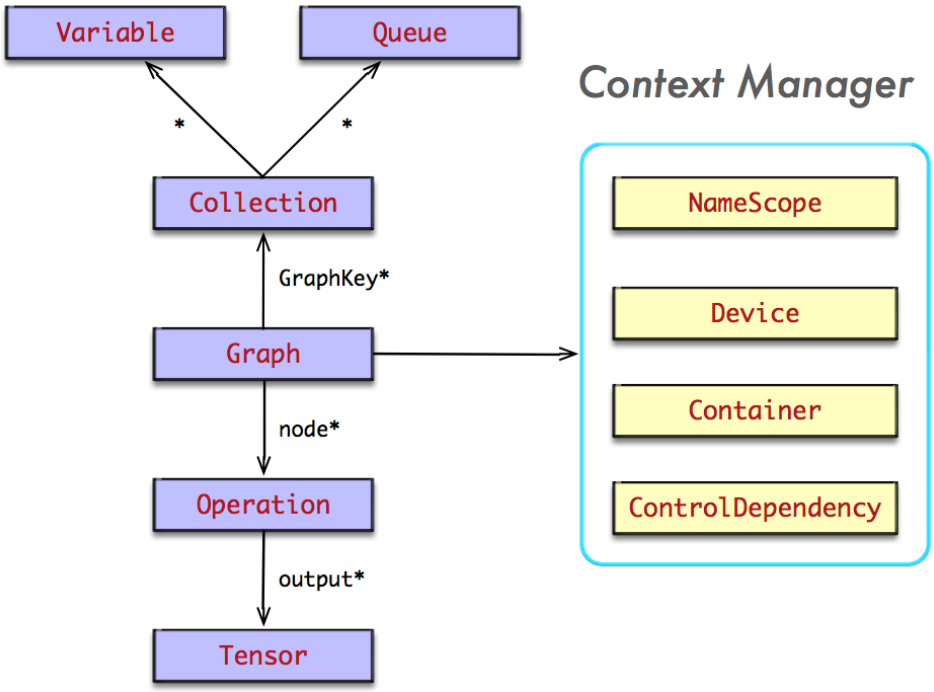


图 A-1 领域对象：Graph

A.4 挖掘系统架构

阅读代码犹如在大海中航行，系统架构图就是航海图。阅读代码不能没有整体的系统概念，否则收效不佳，阅读质量大大折扣。必须拥有系统思维，并明确目标，才不至于迷失方向。

首要的任务，就是找到系统的边界，并能够以抽象的思维思考外部系统的行为特征。其次，理清系统中各组件之间的交互，关联关系，及其职责，对于理解整个系统的行为极为重要。

例如，对于 TensorFlow，C API 是衔接前后端系统的桥梁。理解 C API 的设计，基本能够猜测前后端系统的行为。

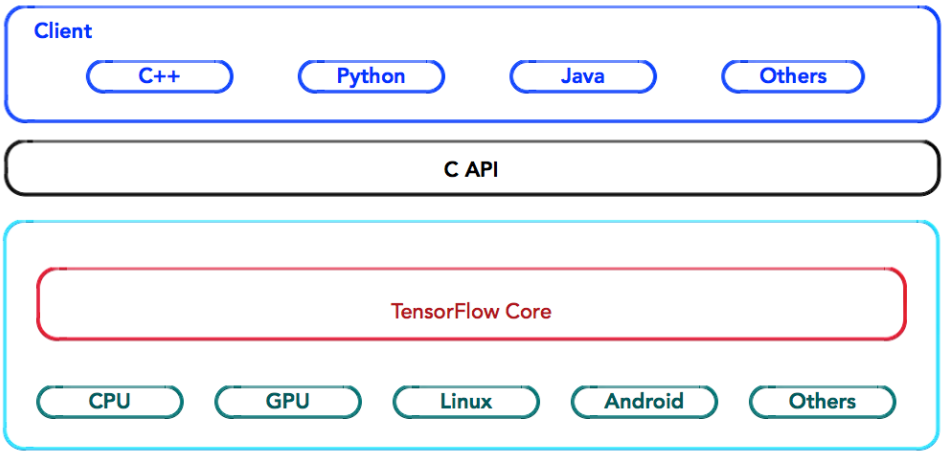


图 A-2 TensorFlow 系统架构

A.5 细节是魔鬼

纠结于细节，将导致代码阅读代码的效率和质量大大折扣。例如，日志打印，解决某个 Bug 的补丁实现，某版本分支的兼容方案，某些变态需求的锤子代码实现等等。

阅读代码的一个常见的反模式就是「给代码做批注」。这是一个高耗低效，投入产出比极低的实践。一般地，越是优雅的系统，注释越少；越是复杂的系统，再多的注释也是于事无补。

我有一个代码阅读的习惯，为代码阅读建立一个单独的 code-reading 分支，一边阅读代码，一边删除这些无关的代码。

```
$ git checkout -b code-reading
```

删除这些噪声后，你会发现系统根本没有想象之中那么复杂。现实中，系统的复杂性，往往都是不成熟的设计和实现导致的额外复杂度。随着对系统的深入理解，很多细节都会自然地浮出水面，所有神秘的面纱都将被揭开而公示天下。

A.6 适可而止

阅读代码的一个常见的反模式就是「一根筋走到底，不到黄河绝不死心」。程序员都拥有一颗好奇心，总是对不清楚的事情感兴趣。例如，消息是怎么发送出去的？任务调度工作原理是什么？数据存储怎么做到的?；虽然这种勇气值得赞扬，但在代码阅读时绝对不值得鼓励。

还有另外一个常见的反模式就是「追踪函数调用栈」。这是一个极度枯燥的过程，常常导致思维的僵化；因为你永远活在作者的阴影下，完全没有自我。

我个人阅读代码的时候，函数调用栈深度绝不超过 3，然后使用抽象的思维方式思考底层的调用。因为我发现，随着年龄的增长，曾今值得骄傲的记忆力，现在逐渐地变成自己的短板。当我尝试追踪过深的调用栈之后，之前的阅读信息完全地消失记忆了。

也就是说，我更习惯于「广度遍历」，而不习惯于「深度遍历」的阅读方式。这样，我才能找到系统隐晦存在的「分层概念」，并理顺系统的层次结构。

A.7 发现她的美

三人行，必有我师焉。在代码阅读代码时，当发现好的设计，包括实现模式，习惯用法等，千万不要错过；否则过上一段时间，这次代码阅读对你来说就没有什么价值了。

当我发现一个好的设计时，我会尝试使用类图，状态机，序列图等方式来表达设计；如果发现潜在的不足，将自己的想法补充进去，将更加完美。

例如，当我阅读 Hamcrest 时，尝试画画类图，并体会它们之间关系，感受一下设计的美感，也是受益颇多的。

A.8 尝试重构

因为这是一次代码阅读的过程，不会因为重构带来潜在风险的问题。在一些复杂的逻辑，通过重构的等价变换可以将其变得更加明晰，直观。

对于一个巨函数，我常常会提取出一个抽象的代码层次，以便发现它潜在的本质逻辑。例如，这是一个使用 Scala 实现的 ArrayBuffer，当需要在尾部添加一个元素时，既有的设计是这样子的。

```
def +=(elem: A): this.type = {
  if (size + 1 > array.length) {
    var newSize: Long = array.length
    while (n > newSize)
      newSize *= 2
```

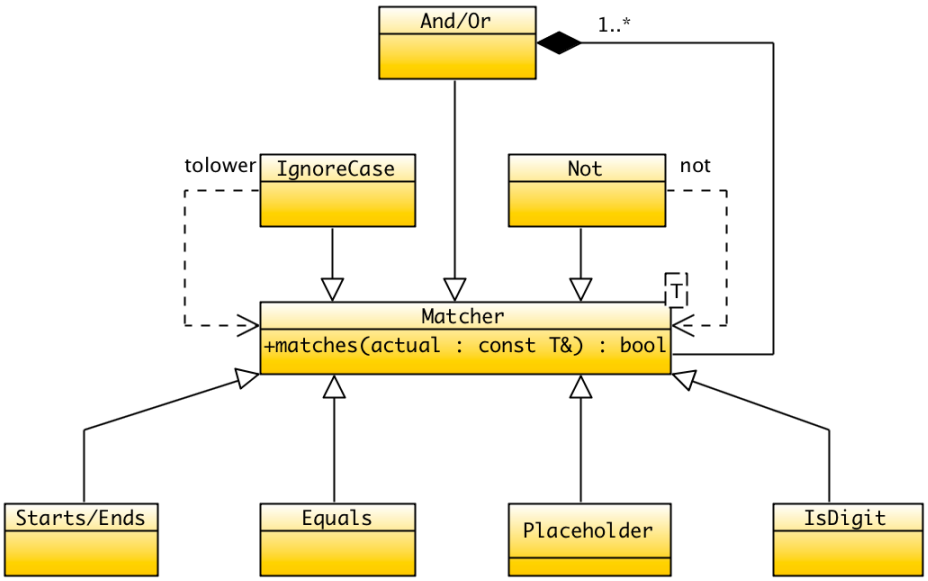


图 A-3 组合式设计

```
    newSize = math.min(newSize, Int.MaxValue).toInt

    val newArray = new Array[AnyRef](newSize)
    System.arraycopy(array, 0, newArray, 0, size)
    array = newArray
  }
  array(size) = elem.asInstanceOf[AnyRef]
  size += 1
  this
}
```

这段代码给阅读造成了极大的障碍，我会尝试通过快速的函数提取，发现逻辑的主干。

```
def +=(elem: A): this.type = {
  if (atCapacity)
    grow()
  addElement(elem)
}
```

至于 `atCapacity`，`grow`，`addElement` 是怎么实现的，压根不用关心，因为我已经达到阅读代码的效果了。

A.9 形式化

当阅读代码时，有部分人习惯画程序的「流程图」。相反，我几乎从来不会画「流程图」，因为流程图反映了太多的实现细节，而不能深刻地反映算法的本质。

我更倾向于使用「形式化」的方式来描述问题。它拥有数学的美感，简洁的表达方式，及其高度抽象的思维，对挖掘问题本质极其关键。

例如，对于 FizzBuzzWhizz 的问题，相对于冗长的文字描述，或流程图，形式化的方式将更加简单，并富有表达力。以 3, 5, 7 为输入，形式化后描述后，可清晰地挖掘出问题的本质所在。

```
r1: times(3) => Fizz ||
    times(5) => Buzz ||
    times(7) => Whizz

r2: times(3) && times(5) && times(7) => FizzBuzzWhizz ||
    times(3) && times(5) => FizzBuzz  ||
    times(3) && times(7) => FizzWhizz ||
    times(5) && times(7) => BuzzWhizz

r3: contains(3) => Fizz

rd: others => string of others

spec: r3 || r2 || r1 || rd
```

A.10 实例化

实例化是认识问题的一种重要方法，当逻辑非常复杂时，一个简单例子往往使自己豁然开朗。在理想的情况下，实例化可以做成自动化的测试用例，并以此描述系统的行为。

如果存在某个算法和实现都相当复杂时，也可以通过实例化探究算法的工作原理，这对于理解问题本身大有益处。

以 Spark 中划分 DAG 算法为例。以 G 为起始节点，从后往前按照 RDD 的依赖关系，依次识别出各个 Stage 的边界。

- Stage 3 的划分
    1. G 与 B 之间是窄依赖，规约为同一 Stage(3)；
    2. B 与 A 之间是宽依赖，A 为新的起始 RDD，递归调用此过程；
    3. G 与 F 之间是宽依赖，F 为新的起始 RDD，递归调用此过程。

- Stage 1 的划分
    1. A 没有父亲 RDD，Stage(1) 划分结束。特殊地 Stage(1) 仅包含 RDD A。

- Stage 2 的划分
    1. 因 RDD 之间的关系都为窄依赖，规约为同一个 Stage(2)；
    2. 直至 RDD C, E，因没有父亲 RDD，Stage(2) 划分结束。

最终，形成了 Stage 的依赖关系，依次提交 TaskSet 至 TaskScheduler 进行调度执行。

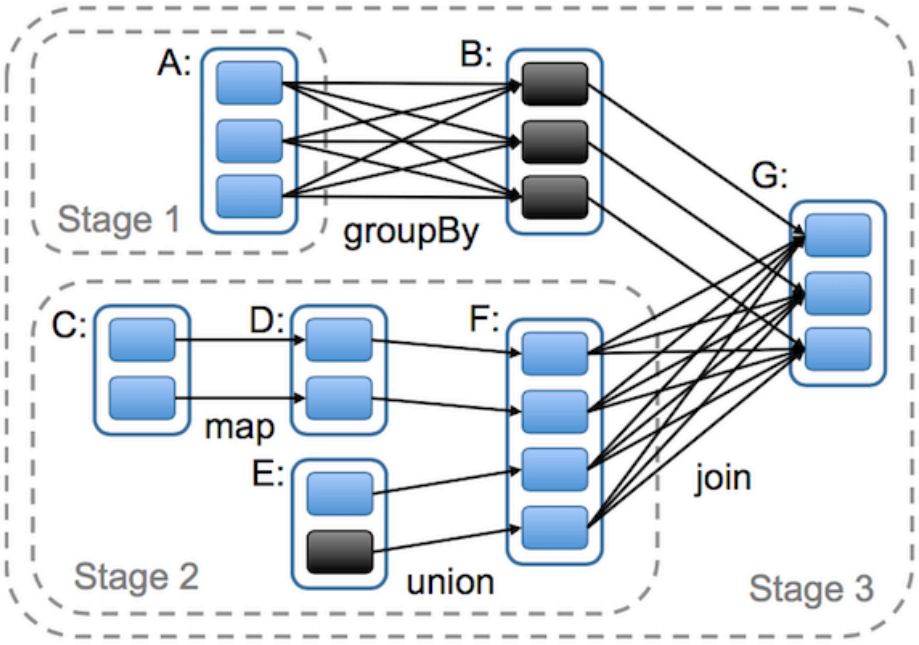


图 A-4 Spark：Stage 划分算法

A.11 独乐乐，不如众乐乐

与他人分享你的经验，也许可以找到更多的启发；尤其对于熟知该领域的人沟通，如果是 Owner 就更好了，肯定能得到意外的惊喜和收获。

也可以通过各种渠道，收集他人的经验，并结合自己的思考，推敲出自己的理解，如此才能将知识放入自己的囊中。

阅读代码，不是一个人的世界；应该走出去，多参加一些社区活动，了解生态圈中主流的研究方向，技术动态，产业发展，对于理解业务是极其有帮助的。

B

持续学习

B.1 说文解字

R 读书有三到，谓心到，眼到，口到。- 朱熹《训学斋规》

我出生时，父亲为我取名**刘光云**，承“光”辈，单字“云”。但自上学之后，便不知所云了。有一天语文老师说文解字道：“聪者，耳到，眼到，口到，心到也”。判若相识恨晚的感觉，我对“聪”字情有独钟，随将自己的名字改为**刘光聪**。

说来也巧，自那之后，妈妈再也没有担心过我的学习了。

选择

耳到，择其善者而从之，择不善者而改之。有人习惯于巨函数，大逻辑，问究为何如此，美其名曰：都是为了效 (hai) 率 (zi)；而我更偏爱具有层次感的代码风格，短小精干，意图明确。

他人的经验固然重要，但需要我们自己选择性地接收，而不是一味的听取。不要为大师所迷，大师有时也会犯错。关键在于自我思考，善于辨别。尤其在这个浮躁的世间里，能在地上跑的都喊自己是“大师”。

抽象

眼到，扫除外物，直觅本来也。一眼便能看到的都是假象，看不到，摸不着的往往才是本质。有人习惯平铺直叙的的逻辑，任其重复；而我更加偏爱抽象，并将揭示本质过程当成一种享受。

抽象，固然存在复杂度。但这样的复杂度是存在上下文的，如果大家具有类似的经验，

抽象自然就变成了模式。那是一种美，一种沟通的媒介。

如果对方缺乏上下文，抽象自然是困难的。所谓简单，是问题本质的揭示，并为此付出最小的代价；而不是平铺直叙，简单是那些门外汉永远也感受不到的美感。

过而不及，盲目抽象，必然增加不必要的复杂度。犹如大规模的预先设计，畅谈客户的各种需求，畅谈软件设计中各种变化，盲目抽象。

分享

口到，传道，授业，解惑也。分享是一种生活的信念，明白了分享的同时，自然明白了存在的意义。我喜欢分享自己的知识，并将其当成一种学习动力，督促自己透彻理解问题的本质。

因为能够分享，所以知识自然变成了自己的东西。每日的 Code Review，我常常鼓励团队成员积极分享，一则为了促就无差异的团队，二则协助分享者透彻问题的本质。

要让别人信服你的观点，关键是要给别人带来信服的理由。分享的同时，能够帮助锻炼自己的表达能力，这需要长时间的**刻意练习**。

领悟

心到，学而思之，思则得之，不思则不得也。只有通过自己独立思考，归纳总结的知识，才是真正属于自己的。

我偏爱使用图表来总结知识，一方面图的表达力远远大于文字；另外，通过画图也逼迫自己能够透彻问题的本质。

B.2 成长之路

消除重复

代码需要消除重复，工作的习惯也要消除重复。不要拘于固有的工作状态，重复的工作状态往往使人陷入舒服的假象，陷入**三年效应**的危机。

提炼知识

首先我们学习的不是信息，而是知识。知识是有价值的，而信息则没有价值。只有通过自己的筛选，提炼，总结才可能将信息转变为知识。

成为习惯

知识是容易忘记的，只有将知识付诸于行动，并将其融汇到自己的工作状态中去，才能永久性地成为自己的财产。

例如，快捷键的使用，不要刻意地去记忆，而是变成自己的一种工作习惯；不要去重复地劳动，使用 Shell 提供自动化程度，让 Shell 成为工作效率提升的利器，并将成为一种工作习惯。

更新知识

我们需要常常更新既有的知识体系，尤其我们处在一个知识大爆炸的时代。我痛恨那些信守教条的信徒，举个简单的例子，陈旧的代码规范常常要求 `if (NULL != p)` 这样的 `YODA Notation` 习惯用法。但是这样的表达编译器是高兴了，但对程序员是非常不友好的。

“if you are at least 18 years old”明显比“if 18 years is less than or equal to your age”更加符合英语表达习惯。

有人驳论此这个习惯用法，但是现代编译器对此类误用通常报告警告；而且保持 TDD 开发节奏，小步前进，此类低级错误很难逃出测试的法网。

重构自我

学，然后知不足；教，然后知困。不要停留在原点，应该时刻重构自己的知识体系。

在刚入门 OO 设计的时候，我无处不用设计模式；因为我看到的所有书籍，都是在讲设计模式如何如何地好。直至后来看到了演进式设计，简单设计和过度设计的一些观点后，让我重新回归到理性。

专攻术业

人的精力是有限的，一个人不可能掌握住世界上所有的知识。与其在程序设计语言的抉择上犹豫不决，不如透彻理解方法论的内在本质；与其在众多框架中悬而未决，不如付出实际，着眼于问题本身。

总之，博而不精，不可不防。

参考文献

[1] M. Abadi, A. Agarwal. Tensorflow, Large-scale machine learning on heterogeneous distributed systems. arXiv preprint, 1603.04467, 2016. https://arxiv.org/abs/1603.04467.

[2] R. Al-Rfou, G. Alain, Theano: A Python framework for fast computation of mathematical expressions. arXiv preprint, 1605.02688, 2016. https://arxiv.org/abs/1605.02688.

[3] T. Chen, M. Li. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. In Proceedings of LearningSys, 2015. www.cs.cmu.edu/~muli/file/mxnet-learning-sys.pdf.

[4] J. Dean, G. S. Corrado. Large scale distributed deep networks. In Proceedings of NIPS, pages 1232–1240, 2012. http://research.google.com/archive/large_deep_networks_nips2012.pdf.

[5] Gaël Guennebaud. Eigen: a c++ linear algebra library. http://downloads.tuxfamily.org/eigen/eigen_CGLibs_Giugno_Pisa_2013.pdf.