

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。

英语术语

因为我所撰写的文章，是供相关专业人士阅读，而非科普读物，因此在书中保留专业领域中朗朗上口的英语术语，故意不做翻译。例如，书中直接使用 OP 的术语，而不是将其翻译为「操作」。

但是，这会造成大面积的中英混杂的表达方式。幸运的是，绝大部分所使用的英语术语都是名词，极少出现动词或者形容词。但是，无论如何都不会丢失原本的主体语义和逻辑。

万事都有例外，对于无歧义的，表达简短，且语义明确的术语，会使用中文术语表示。特殊地，对于中文术语表达存在歧义时，会同时标注中文术语和英语术语。例如，检查点 (Checkpoint)，协调器 (Coordinator)。

一般地，无歧义的中文术语表定义在表1（第vi页）。

英文	中文
Variable	变量，参数
Session	会话
Device	设备

表 1 规范约定

在线帮助

为了更好地与读者交流，在我的 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一件。

目录

前言	v
I 基础知识	1
1 介绍	3
1.1 前世今生	3
1.1.1 DistBelief	3
1.1.2 TensorFlow	4
1.2 社区发展	6
1.2.1 开源	6
1.2.2 里程碑	6
1.2.3 工业应用	6
2 编程环境	9
2.1 代码结构	9
2.1.1 克隆源码	9
2.1.2 源码结构	9
2.1.3 内核	10
2.1.4 Python 接口	10
2.1.5 StreamExecutor	11
2.2 工程构建	11
2.2.1 环境准备	11
2.2.2 配置	13
2.2.3 构建	13
2.2.4 安装	13
2.2.5 验证	13
2.3 技术栈	14
II 系统架构	15
3 系统架构	17
3.1 系统架构	17
3.1.1 Client	18
3.1.2 Master	18
3.1.3 Worker	18
3.1.4 Kernel	19
3.2 图控制	19
3.2.1 组建集群	19
3.2.2 图构造	20
3.2.3 图执行	20
3.3 会话管理	22
3.3.1 创建会话	22
3.3.2 迭代运行	23
3.3.3 关闭会话	25
4 C API：分水岭	27
4.1 Swig：幕后英雄	27
4.2 客户端代理	27
4.3 会话生命周期	28
4.3.1 Python 前端	28
4.3.2 C++ 后端	29
4.4 创建会话	31
4.4.1 编程接口	31
4.4.2 C API	32
4.4.3 后端系统	32
4.5 创建/扩展图	33

4.5.1	编程接口	33
4.5.2	C API	35
4.5.3	后端系统	35
4.6	迭代运行	36
4.6.1	编程接口	36
4.6.2	C API	37
4.6.3	后端系统	38
4.7	关闭会话	38
4.7.1	编程接口	38
4.7.2	C API	39
4.7.3	后端系统	40
4.8	销毁会话	40
4.8.1	编程接口	40
4.8.2	C API	41
4.8.3	后端系统	41
4.9	性能调优	41
4.9.1	创建会话	42
4.9.2	销毁会话	42
4.9.3	删除图实例	43
III	编程模型	45
5	计算图	47
5.1	Python 前端	47
5.1.1	Operation	47
5.1.2	Tensor	48
5.1.3	Graph	50
5.1.4	图构造	50
5.2	后端 C++	52
5.2.1	边	52
5.2.2	节点	54
5.2.3	图	55
5.2.4	OpDef 仓库	58
5.3	图传递	58
6	设备	59
6.1	设备规范	59
6.1.1	形式化	59
6.1.2	上下文管理器	60
6.1.3	设备函数	62
7	会话	63
7.1	资源管理	63
7.1.1	关闭会话	63
7.1.2	上下文管理器	63
7.1.3	图实例	63
7.2	默认会话	65
7.2.1	张量求值	66
7.2.2	OP 运算	66
7.2.3	线程相关	66
7.3	会话类型	67
7.3.1	Session	67
7.3.2	InteractiveSession	68
7.3.3	BaseSession	68

8 变量	71
8.1 实战：线性模型	71
8.2 初始化模型	72
8.2.1 操作变量	72
8.2.2 初始值	73
8.2.3 初始化器	73
8.2.4 快照	74
8.2.5 变量子图	74
8.2.6 初始化过程	74
8.2.7 同位关系	75
8.2.8 初始化依赖	76
8.2.9 初始化器列表	77
8.3 变量分组	78
8.3.1 全局变量	78
8.3.2 本地变量	78
8.3.3 训练变量	79
8.3.4 global_step	79
8.4 源码分析：构造变量	79
8.4.1 构造初始值	80
8.4.2 构造变量 OP	80
8.4.3 构造初始化器	80
8.4.4 构造快照	80
8.4.5 变量分组	80
9 队列	83
9.1 队列	83
9.1.1 FIFOQueue	83
9.1.2 用途	85
9.2 协调器	85
9.2.1 使用方法	85
9.2.2 异常处理	86
9.2.3 实战：LoopThread	87
9.3 QueueRunner	87
9.3.1 注册 QueueRunner	87
9.3.2 执行 QueueRunner	87
9.3.3 关闭队列	89
IV 运行模型	91
10 分布式 TensorFlow	93
10.1 运行模式	93
10.1.1 本地模式	93
10.1.2 分布式模式	93
10.2 领域模型	95
10.2.1 Cluster	95
10.2.2 Job	95
10.2.3 Task	95
10.2.4 Server	96
10.2.5 Master Service	96
10.2.6 Worker Service	96
10.3 组建集群	97
10.3.1 ClusterSpec	97
10.3.2 Protobuf 描述	97
10.4 Server	98
10.4.1 领域模型	98
10.4.2 Protobuf 描述	98
10.4.3 服务互联	99

V 模型训练	101
11 BP 算法	103
11.1 TensorFlow 实现	103
11.1.1 计算梯度	103
11.1.2 应用梯度	107
12 数据加载	111
12.1 数据注入	111
12.2 数据预加载	112
12.2.1 使用 Const	112
12.2.2 使用 Variable	112
12.2.3 批次预加载	113
12.3 数据管道	113
12.3.1 构建文件名队列	114
12.3.2 读取器	114
12.3.3 解码器	114
12.3.4 构建样本队列	115
12.3.5 输入子图	115
12.4 数据协同	115
12.4.1 阶段 1	116
12.4.2 阶段 2	118
12.4.3 阶段 3	119
12.4.4 Pipeline 节拍	120
13 Saver	121
13.1 Saver	121
13.1.1 使用方法	121
13.1.2 文件功能	122
13.1.3 模型	123
14 MonitoredSession	125
14.1 引入 MonitoredSession	125
14.1.1 使用方法	126
14.1.2 使用工厂	126
14.1.3 装饰器	127
14.2 生命周期	127
14.2.1 初始化	128
14.2.2 执行	129
14.2.3 关闭	129
14.3 模型初始化	130
14.3.1 协调协议	130
14.3.2 SessionManager	130
14.3.3 引入工厂	131
14.3.4 Scaffold	131
14.3.5 初始化算法	133
14.3.6 本地变量初始化	133
14.3.7 验证模型	135
14.4 异常安全	136
14.4.1 上下文管理器	136
14.4.2 停止 QueueRunner	136
14.5 回调钩子	137

A 代码阅读	141
A.1 工欲善其事，必先利其器	141
A.2 力行而后知之真	141
A.3 发现领域模型	142
A.4 挖掘系统架构	142
A.5 细节是魔鬼	143
A.6 适可而止	144
A.7 发现她的美	144
A.8 尝试重构	144
A.9 形式化	145
A.10 实例化	146
A.11 独乐乐，不如众乐乐	147
B 持续学习	149
B.1 说文解字	149
B.1.1 选择	149
B.1.2 抽象	149
B.1.3 分享	150
B.1.4 领悟	150
B.2 成长之路	150
B.2.1 消除重复	150
B.2.2 提炼知识	150
B.2.3 成为习惯	151
B.2.4 更新知识	151
B.2.5 重构自我	151
B.2.6 专攻术业	151

Part I

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 是一个支持大规模和异构环境的机器学习系统。它使用 **数据流图** (Dataflow Graph) 表示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。[\[1\]](#)

数据流图的节点被映射在集群中的多个机器，在一个机器内被映射在多个计算设备 (Device) 上，包括 CPU, GPU, TPU。TensorFlow 灵活的架构支持多种计算平台，包括台式机，服务器，移动终端等。

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方面的研究，但 TensorFlow 优异的通用性使其也可广泛用于其他领域的数值计算。

1.1 前世今生

TensorFlow 是 DistBelief 的后继者，它站在巨人的肩膀上，革命性地重新设计架构设计，使得 TensorFlow 在机器学习领域一鸣惊人，在社区中产生了重大的影响。

为了更好地理解 TensorFlow 系统架构的优越性，得先从 DistBelief 谈起。

1.1.1 DistBelief

DistBelief 是一个用于训练大规模神经网络的分布式系统，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。

编程模型

DistBelief 的编程模型是基于层 (Layer) 的 DAG(Directed Acyclic Graph) 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实施非线性变换。

架构

DistBelief 使用参数服务器 (Parameter Server) 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，用于模型训练的计算；有状态的 PS(Parameter Server) 进程，用于维护模型参数。

如图1-1（第4页）所示，在分布式训练过程中，各个模型备份（Model Replica）异步地从 PS 上拉取（Fetch）训练参数 w ，当完成一步迭代运算后，推送（Push）参数的梯度 ∇w 到 PS 上去，并完成参数更新。

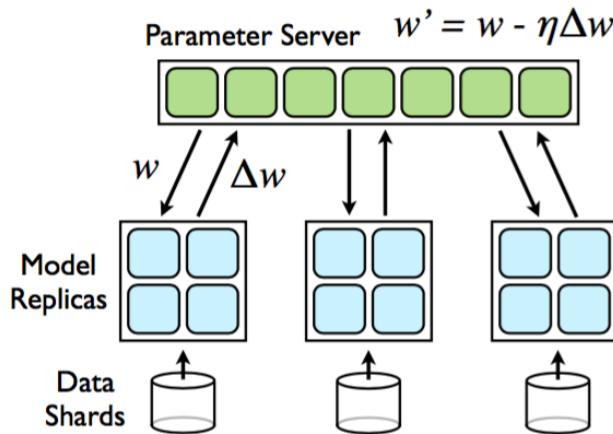


图 1-1 DistBelief: Parameter Server 架构

缺陷

但是，对于高级用户，DistBelief 的编程模型，及其 Parameter Server 的系统架构，缺乏如下几个方面的扩展性。

1. 优化算法：添加新的优化算法，必须修改 Parameter Server 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具有很大的挑战性，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 GPU；遗留的系统架构对支持新的计算设备缺乏弹性空间。

1.1.2 TensorFlow

正因为 DistBelief 遗留的架构和设计，不再满足潜在的深度学习与日俱增的需求，Google 毅然决定在 DistBelief 基础上做全新的架构设计，从而诞生了 TensorFlow。

编程模型

TensorFlow 使用数据流图（Dataflow Graph）表示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1-2（第5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，由 Gradients 构建了与前向子图对应的反向子图，用于训练参数的梯度计算；最后，使用‘SGD’的优化算法，构造参数更新子图，完成参数的更新。

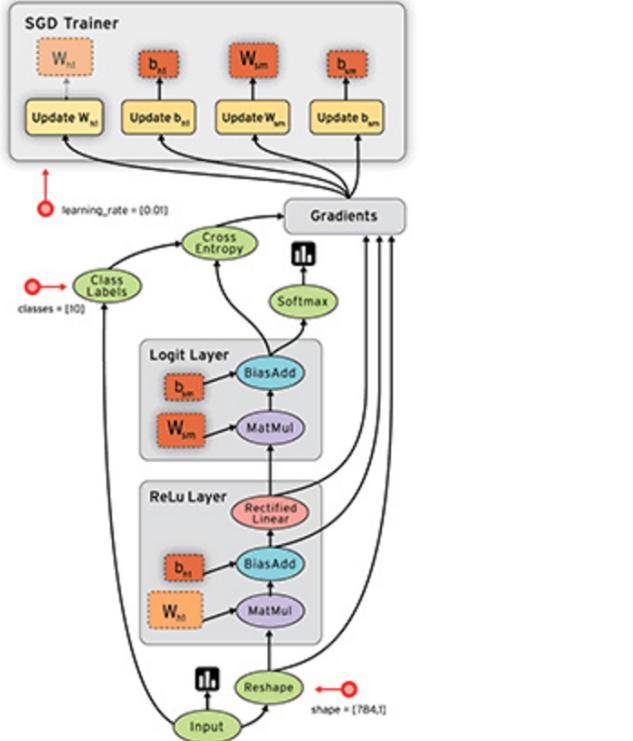


图 1-2 TensorFlow 数据流图

设计原则

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, TPU 多种异构计算设备类型；
4. 抽象任务：基于 Task 的 PS 任务，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. 跨平台：支持多 CPU/GPU/TPU 运算；支持台式机/服务器/移动设备；支持 Windows, Linux, MacOS；
2. 分布式：支持本地和分布式的模型训练和推理；
3. 多语言：支持 Python, C++, Java, Go 等多种程序设计语言的 API；
4. 通用性：支持各种复杂的网络模型的设计和实现；
5. 可扩展：支持 OP 扩展，Kernel 扩展，Device 扩展；

6. 可视化：使用TensorBoard可视化整个训练过程，包括计算图。

1.2 社区发展

TensorFlow是目前最为流行的机器学习框架。自开源以来，TensorFlow社区相当活跃。来自众多的非Google员工拥有数万次代码提交，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的问题被回答；在各类技术大会上，TensorFlow也是一颗闪亮的明星，得到众多开发者的青睐。

1.2.1 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1-3（第6页）所示，TensorFlow的社区活跃度已远远超过其他竞争对手，逐渐成为目前最为炙手可热的机器学习和深度学习框架，已然成为事实上的工业标准。

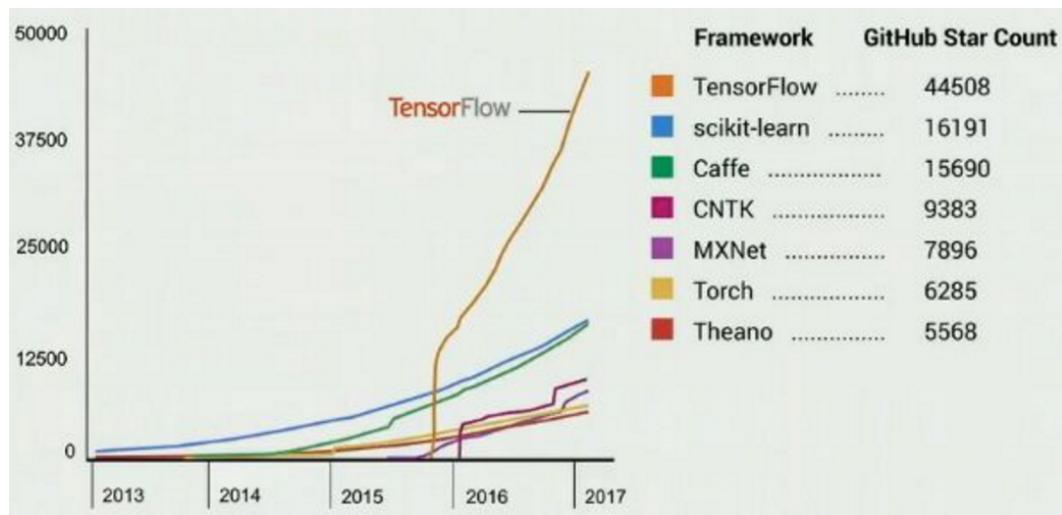


图 1-3 TensorFlow 社区活跃度

毫无疑问，TensorFlow的开源对学术界和工业界产生了巨大的影响，其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。

1.2.2 里程碑

TensorFlow自2015.11开源依赖，平均一个多月发布一个版本。如图1-4（第7页）所示，展示了TensorFlow几个重要特性的发布时间。

1.2.3 工业应用

TensorFlow自开源发展一年多以来，在生产环境中被大量应用使用。在医疗方面，使用

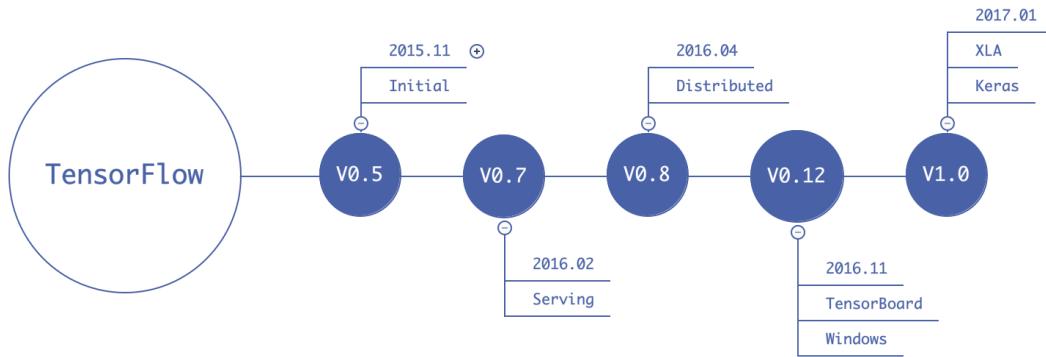


图 1-4 TensorFlow 重要里程碑

TensorFlow 构建机器学习模型，帮助医生预测皮肤癌；在音乐、绘画领域，使用 TensorFlow 构建深度学习模型，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，用于翻译等工作。

如图 1-5（第 7 页）所示，TensorFlow 在 Google 内部项目应用的增长也十分迅速，多个产品都有相关应用，包括：Search, Gmail, Translate, Maps 等等。

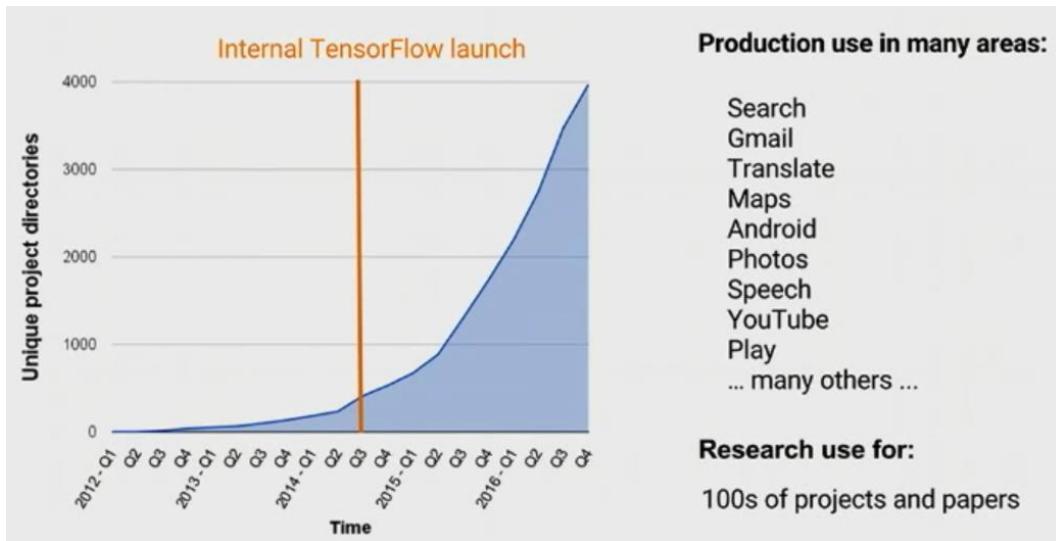


图 1-5 TensorFlow 在 Google 内部使用情况

2

编程环境

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

- Martin Flower

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，环境验证，以便对 TensorFlow 有一个基本的感性认识。

2.1 代码结构

2.1.1 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，`r1.2` 分支。

```
$ cd tensorflow
$ git checkout r1.2
```

2.1.2 源码结构

可以运行如下命令，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`, `c` 组件，部分涉及 `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
```

2.1.3 内核

内核的源码物理组织如下所示。主要包括平台，实用函数库，Protobuf 定义，本地和分布式运行时，框架，图结构，及其 OP 定义与 Kernel 实现等组成，这也是本书重点剖析的对象。

示例代码 2-2 内核源码结构

```
./tensorflow/core  
└── common_runtime  
└── debug  
└── distributed_runtime  
└── example  
└── framework  
└── graph  
└── grappler  
└── kernels  
└── lib  
└── ops  
└── platform  
└── profiler  
└── protobuf  
└── public  
└── user_ops  
└── util
```

2.1.4 Python 接口

Python 编程接口的源码物理组织如下所示，它定义和实现了程序员的编程接口，这也是本书重点剖析的对象。

示例代码 2-3 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
```

2.1.5 StreamExecutor

StreamExecutor 是 Google 另一个组件库，但它也是 TensorFlow 计算的核心引擎，本书将大体介绍其系统架构与工作原理。

示例代码 2-4 StreamExecutor 源码结构

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

2.2 工程构建

因篇幅受限，本文仅以 Mac OS 为例，讲述 TensorFlow 的源码编译、安装、及其验证过程。其他操作系统，及其 CUDA 环境安装，请查阅 TensorFlow 官方文档。

2.2.1 环境准备

在构建 TensorFlow 前，需要事先准备构建环境。TensorFlow 的前端是一个支持多语言的编程接口，后端是一个使用 C++ 实现的执行系统。

因此，编译 TensorFlow 源代码之前，需要事先安装前端系统所使用编程语言的编译器、解释器、及其运行时环境。例如，使用 Python 的前端编程接口，需要事先安装 Python2，或者 Python3。下文以 Python2 为例，讲述环境准备过程；如果使用 Python3，请查阅相关文档。

其次，也需要事先安装 GCC, Clang 等 C++ 的编译器，用于编译后端系统实现。本文将不再冗述这两个方面的环境安装过程。

另外，TensorFlow 使用 Bazel 的构建工具。因此，需要事先安装 Bazel。不幸的是，因为 Bazel 依赖于 JDK，因此在安装 Bazel 之前需要先安装 JDK。

安装 JDK

可以从 Oracle 官网上下载至少 1.8 及以上版本的 JDK 版本，然后安装在系统中。

```
$ sudo mkdir -p /usr/lib/jvm  
$ sudo tar zxvf jdk-8u51-linux-x64.tar.gz -C /usr/lib/jvm  
$ sudo ln -s /usr/java/jdk1.8.0_51 /usr/java/default
```

创建 Java 相关环境变量，并添加到`~/.bashrc` 配置文件中。

```
$ echo 'export JAVA_HOME=/usr/java/default' >> ~/.bashrc  
$ echo 'export PATH="$JAVA_HOME/bin:$PATH"' >> ~/.bashrc
```

生效环境变量。

```
$ source ~/.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，需要事先安装 Ruby 解释器，在此不再冗述。

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

安装 Swig

TensorFlow 使用 Swig 构建多语言编程的环境，因此需要事先安装 Swig 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
```

如果系统未安装 pip，则可以使用 brew 先安装 pip：

```
| $ brew install pip
```

2.2.2 配置

编译环境准备就绪之后，便可以执行`./configure` 配置 TensorFlow 的编译环境了。特殊地，当系统不支持 GPU，则可以不需要配置和安装 CUDA，及其 cuDNN。

```
| $ ./configure
```

2.2.3 构建

当配置成功后，使用 Bazel 启动 TensorFlow 的编译。特殊地，当需要支持 GPU 时，添加`--config=cuda` 编译选项。

```
| $ bazel build --config=opt //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
```

2.2.4 安装

当 Whell 包构建成功后，便可以使用 pip 安装 TensorFlow 了。

```
| $ sudo pip install /tmp/tensorflow_pkg/tensorflow-1.2.0-py2-none-any.whl
```

2.2.5 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!
```

2.3 技术栈

通过构建 TensorFlow 源码，应该对 TensorFlow 所依赖的构建工具、组件库、及其第三方工具包有了初步的感性认识。

按照 TensorFlow 的系统软件层次，通过一张表格罗列 TensorFlow 所使用的技术栈，以便更清晰地了解 TensorFlow 的生态系统。

层	功能	组件
视图层	计算图可视化	TensorBoard
工作流层	数据集准备，存储，加载	Keras/TF Slim
计算图层	计算图构造与优化 前向计算/后向传播	TensorFlow Core
高维计算层	高维数组处理	Eigen
数值计算层	矩阵计算 卷积计算	BLAS/cuBLAS/ cuRAND/cuDNN
网络层	通信	gRPC/RDMA
设备层	硬件	CPU/GPU/TPU

图 2-1 TensorFlow 技术栈

Part II

系统架构

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

- Martin Flower

3 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。

3.1 系统架构

TensorFlow 的系统结构以 C API 为界，¹将整个系统分为前端和后端两个子系统：

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。

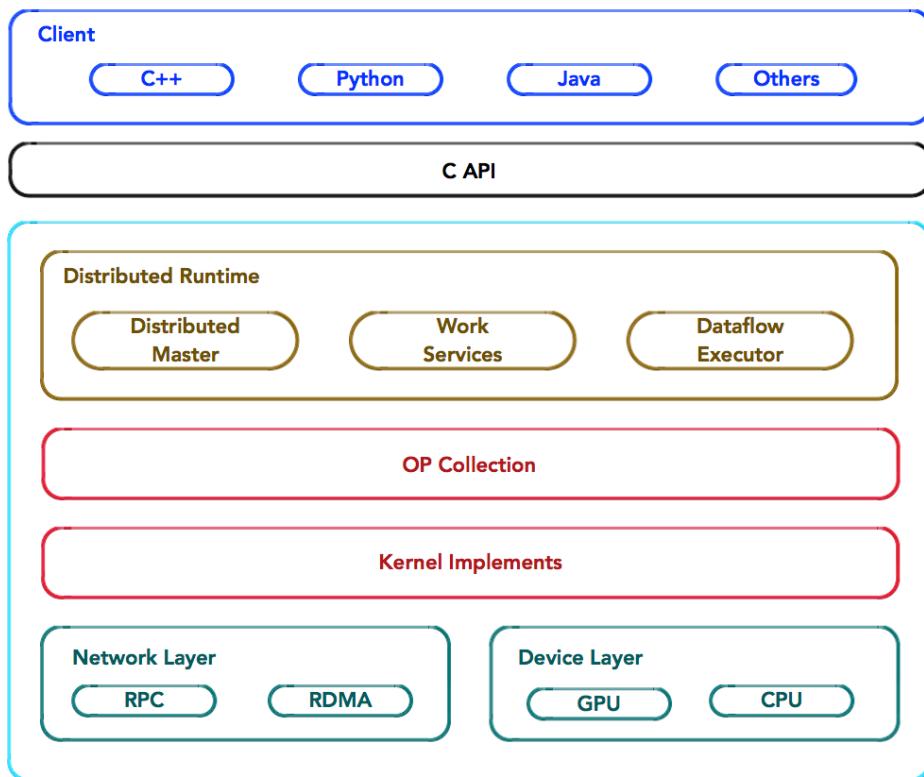


图 3-1 TensorFlow 系统架构

如图 3-1（第 17 页）所示，重点关注系统中如下 4 个基本组件，它们是系统分布式运行时的核心。

¹事实上，后端系统中也存在 Client 的代码，并常称它为前端系统在后端系统实现中的代理 Client。在后面的章节，将详细地讨论这个问题。

3.1.1 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，将 Protobuf 格式的 GraphDef 序列化后发送至 Master，启动计算图的执行过程。

3.1.2 Master

在分布式的运行时环境中，Client 根据 `Session.run` 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 Client 通过 `Session.run` 传递 `fetches` 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。

随后，Master 负责将 Client Graph 按照任务的名称分裂 (split-by-task) 为多个子图片段，常称为 (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 上并发执行这些子图片段。

最后，Master 将通知所有 Work 启动相应子图片段的执行；其中，Work 之间可能存在数据交互，Master 不参与两者之间的数据交换，它们两两互相通信，独立地完成交换数据，直至完成所有计算。

3.1.3 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. 按照拓扑排序算法执行本地子图，并调度 OP 的 Kernel 实现；
3. 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 也会执行图剪枝，得到最小依赖的 Client Graph。

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 Client Graph 分裂 (split-by-device) 为多个 Graph Partition；其中，每个计算设备对应一个 Graph Partition；随后，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。

3.1.4 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 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。

3.2 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。

3.2.1 组建集群

如图3-2（第20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。

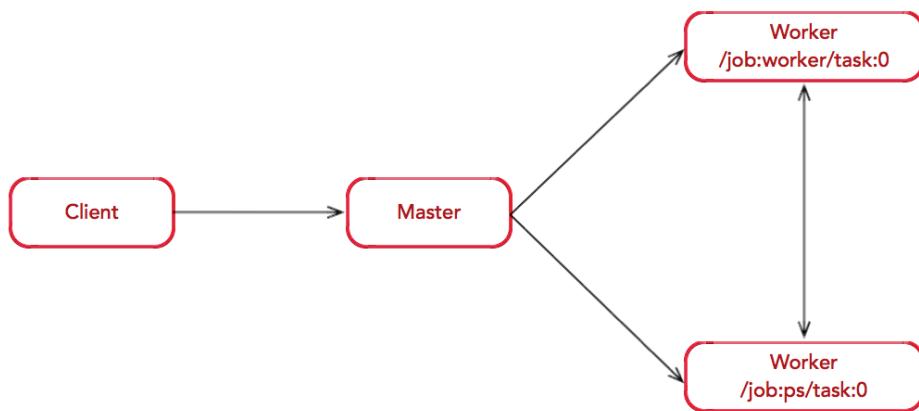


图 3-2 TensorFlow 集群: 1 PS + 1 Worker

3.2.2 图构造

如图3.2.2（第20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。

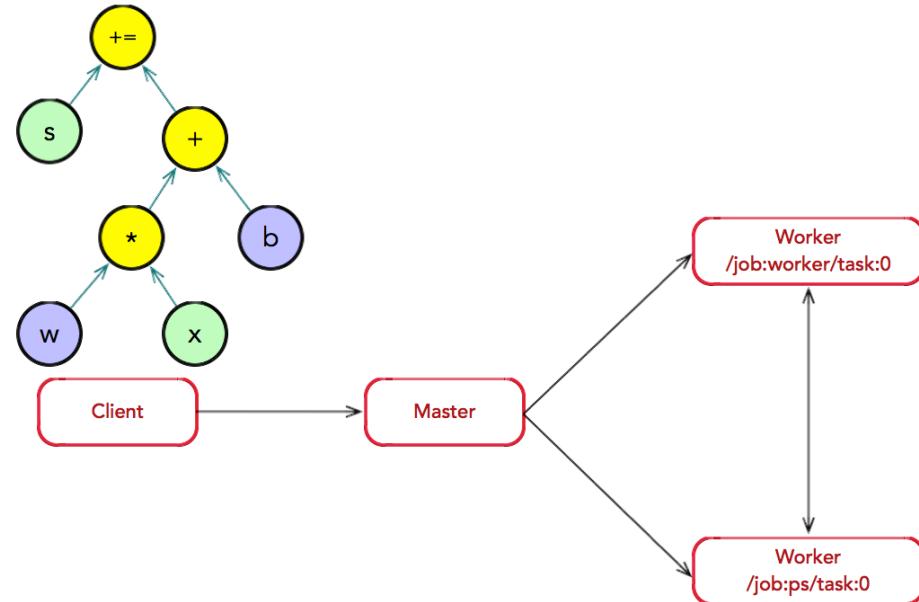


图 3-3 图构造

3.2.3 图执行

如图3.2.3（第21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。

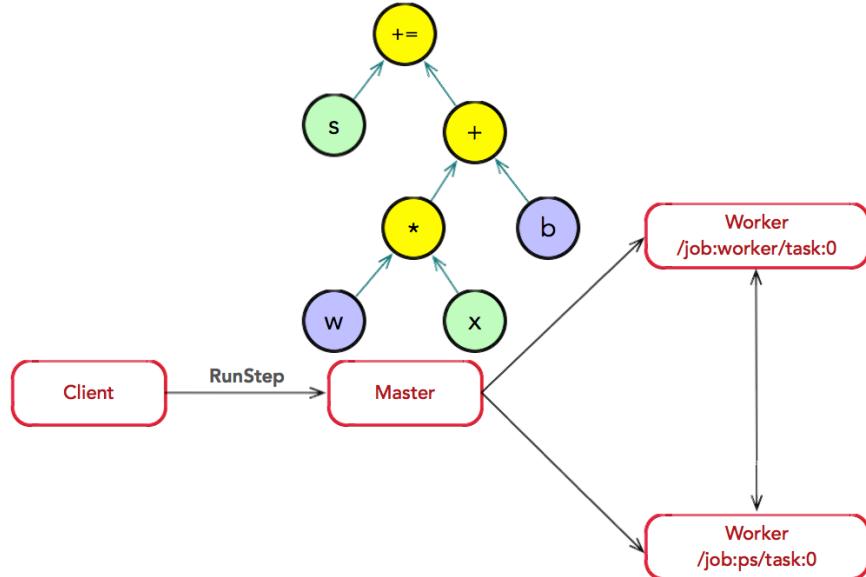


图 3-4 图执行

图分裂

如图3.2.3（第21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。

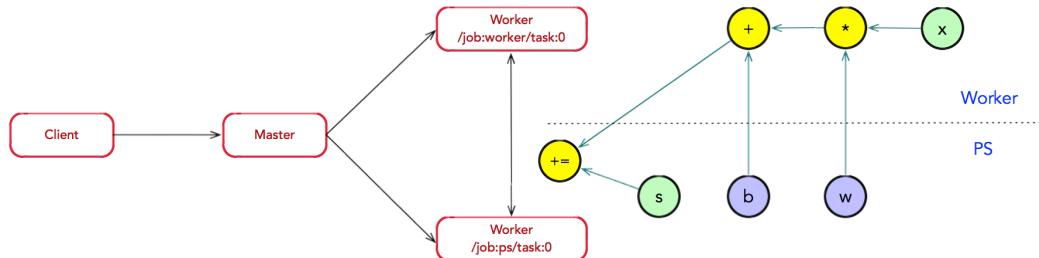


图 3-5 图分裂：按任务划分

子图注册

如图3.2.3（第22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。

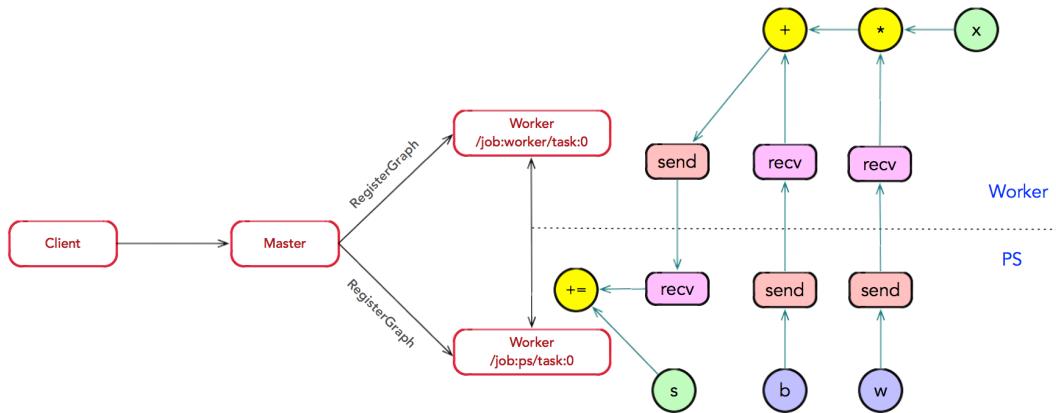


图 3-6 子图注册：插入 Send 和 Recv 节点

子图运算

如图 3.2.3（第 22 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。

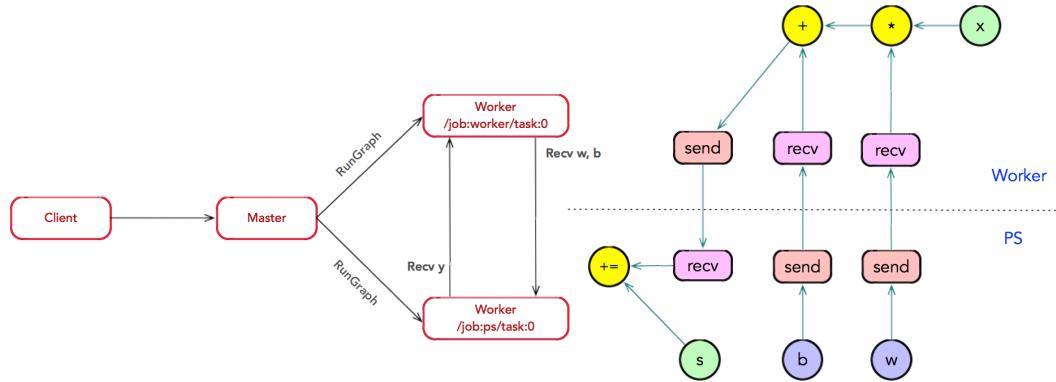


图 3-7 子图执行

3.3 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。

3.3.1 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 3.3.1（第 23 页）所示。

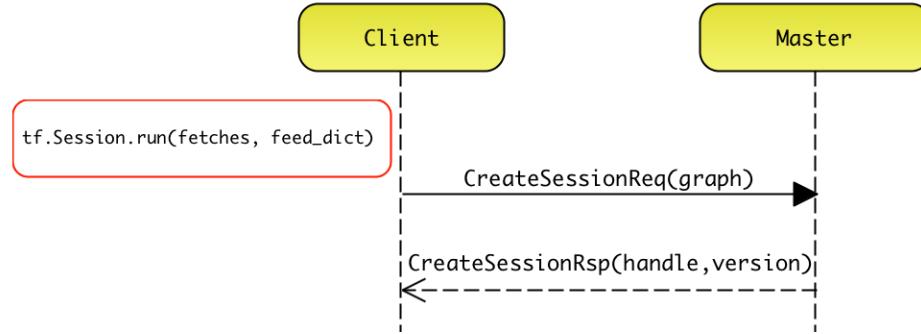


图 3-8 创建会话

3.3.2 迭代运行

随后，Client 会启动迭代执行的过程，并且称每次迭代为一次 Step。此时，Client 发送 `RunStepRequest` 消息给 Master；并且消息携带 `handle` 标识，用于 Master 索引相应的 `MasterSession` 实例。如图 3.3.2（第 23 页）所示。

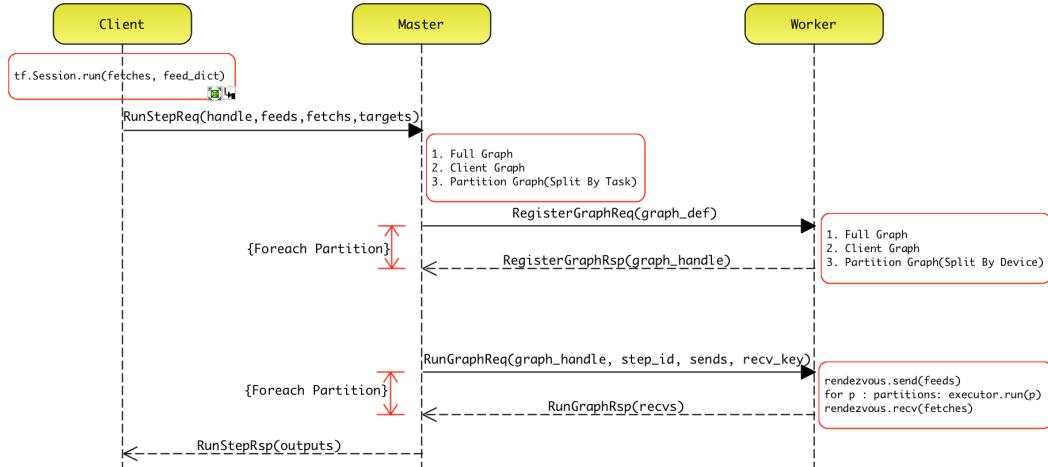


图 3-9 迭代执行

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，再次执行图剪枝，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 `graph_handle` 标识，用于 Worker 索引相应的子图。

Worker 收到消息 `RunGraphRequest` 消息后，按照如下形式化了的算法执行子图的运算。

```
def run_graph(feeds):
    rendezvous.send(feeds)
    for p in partitions:
        executor.run(p)
    rendezvous.recv(fetches)
```

首先，Worker 根据 `graph_handle` 索引相应的子图；然后，并发执行所包含的所有子图片段。其中，每个子图片段放置在单独的 `Executor` 中执行，`Executor` 将按照拓扑排序算法完成子图片段的计算。

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，则需要涉及跨主机，或跨进程的通信。

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3.3.2（第24页）所示。

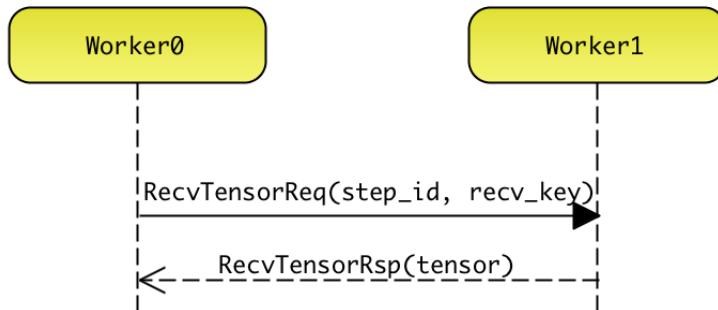


图 3-10 Worker 之间的数据交换

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。

3.3.3 关闭会话

经过许多次迭代执行后，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3.3.3（第25页）所示。

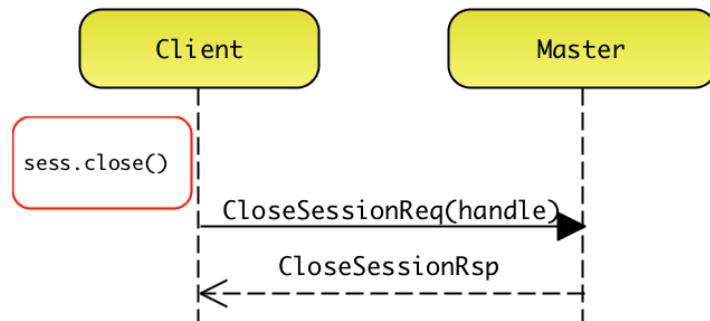


图 3-11 关闭会话

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

- Martin Flower

4

C API：分水岭

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. 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。

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。

4.1 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 `tf_session.i` 自动生成了两个适配 (Wrapper) 文件：

1. `pywrap_tensorflow.py`: 负责对接上层 Python 调用；
2. `pywrap_tensorflow.cpp`: 负责对接下层 C API 调用。

`pywrap_tensorflow.py` 模块首次被导入时，自动地加载 `_pywrap_tensorflow.so` 的动态链接库。从而在运行时实现了 `pywrap_tensorflow.py` 到 `pywrap_tensorflow.cpp` 的函数调用关系。

在 `pywrap_tensorflow.cpp` 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，最终实现到 `c_api.c` 具体实现的调用关系。如图 4.1 (第 28 页) 所示。

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。

4.2 客户端代理

在上一章提及，C API 并非是 Client 与 Master 的分界线。如图 4.2 (第 28 页) 所示，Client 有一部分 C++ 实现，`tf.Session` 实例负责 C++ 实现的 `tensorflow::Session` 实例的创建和销毁，并且它直接持有 `tensorflow::Session` 实例的句柄。

根据运行时环境，`tensorflow::Session` 可能存在多种实现。例如，存在 `tensorflow::DirectSession` 负责本地模式的会话控制；而 `tensorflow::GrpcSession` 负责分布式模式的会话控制。

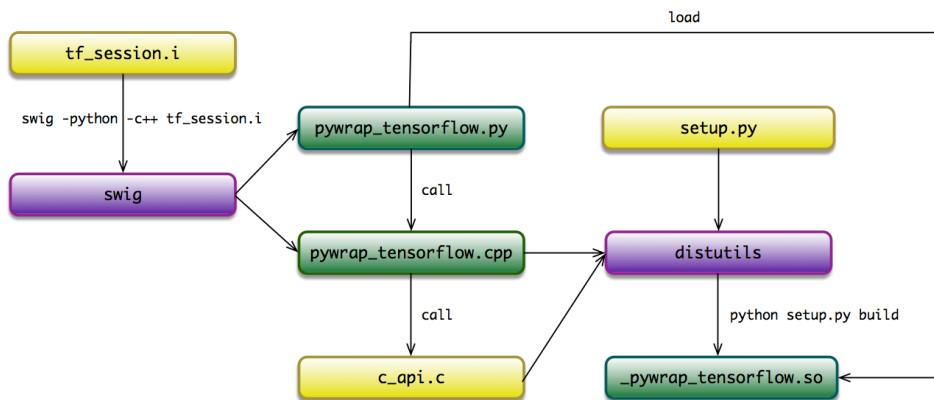


图 4-1 Swig 代码生成器

一般地，用户使用的是 `tf.Session`，而非 `tensorflow::Session`。因此，后者常常称为前者的代理。

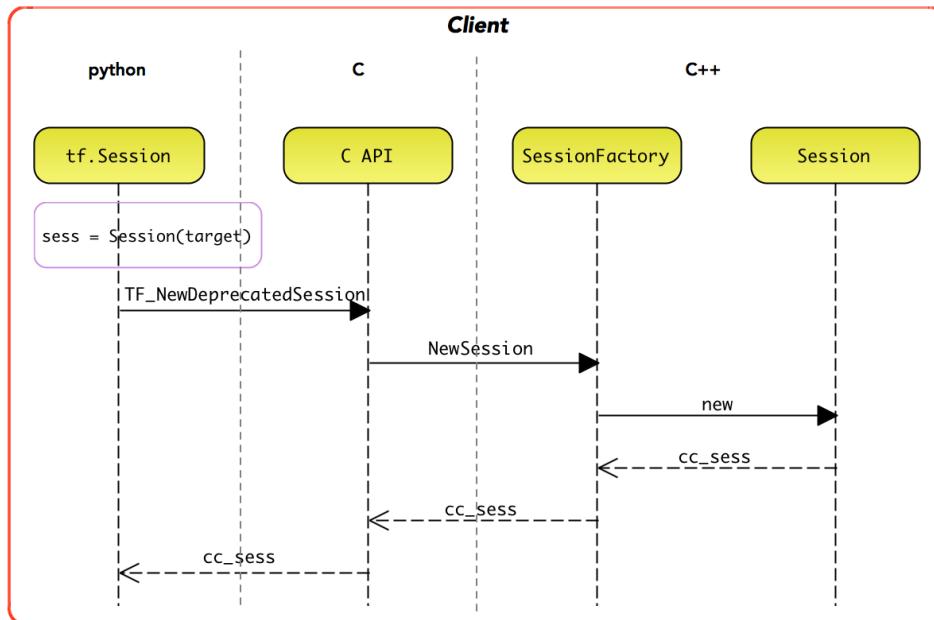


图 4-2 客户端：Session 实例创建过程

4.3 会话生命周期

会话的生命周期包括会话的创建，加载计算图，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。

4.3.1 Python 前端

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);`

 1. `Session._extend_graph(graph);`
 2. `Session.TF_Run(feeds, fetches, targets);`

3. 关闭 `Session;`
4. 销毁 `Session;`

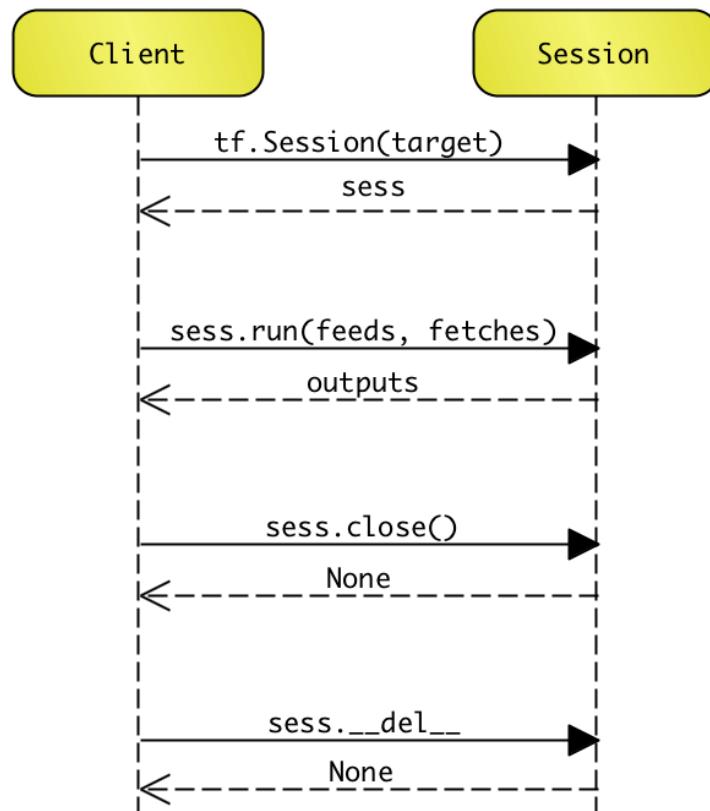


图 4-3 Python: Session 生命周期

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()
  
```

4.3.2 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。

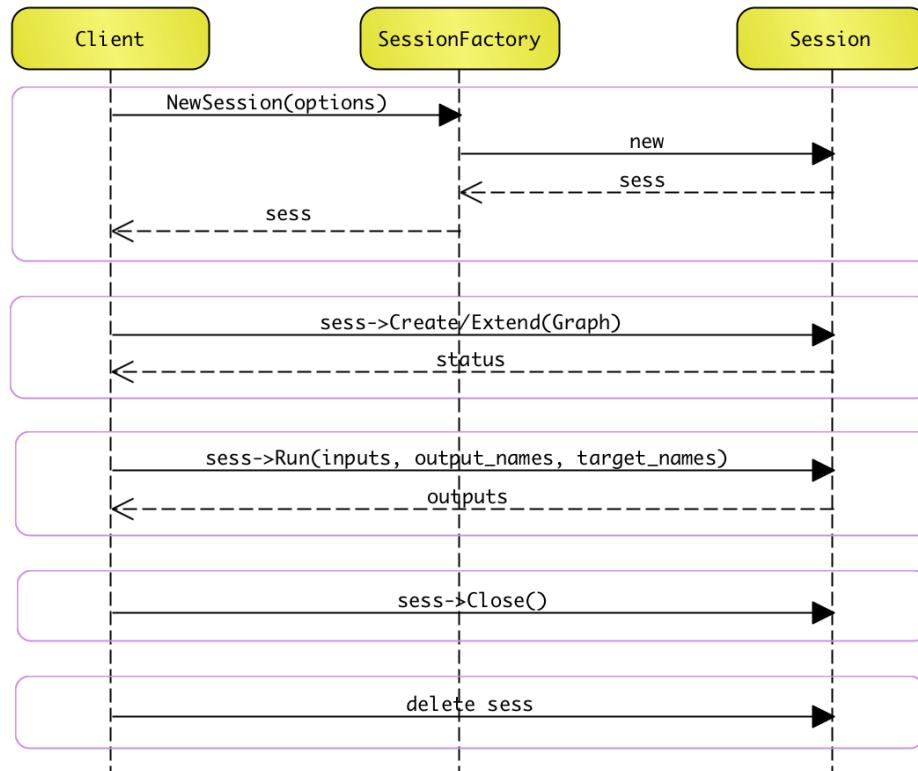


图 4-4 C++: Session 生命周期

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();

```

4.4 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。后端 C++ 系统根据前端传递的 Session.target，使用 SessionFactory 多态创建 tensorflow::Session 对象。

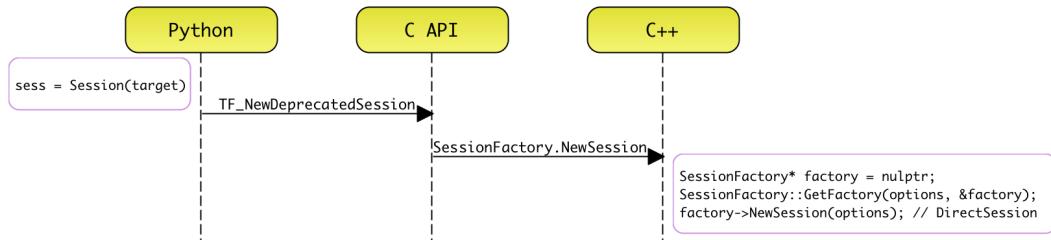


图 4-5 创建会话

4.4.1 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。

```
# tensorflow/python/client/session.py
class Session(BaseSession):
    def __init__(self, target='', graph=None, config=None):
        super(Session, self).__init__(target, graph, config=config)
        # ignoring implements...
```

在 `BaseSession` 的构造函数中，将调用 `pywrap_tensorflow` 模块中的函数。其中，`pywrap_tensorflow` 模块自动由 Swig 生成。

```
# 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):
        # ignoring implements...
        with errors.raise_exception_on_not_ok_status() as status:
            self._session = tf_session.TF_NewDeprecatedSession(opts, status)
```

Python 包装器

在 `pywrap_tensorflow` 模块中，通过 `_pywrap_tensorflow` 的转发，实现了从 Python 到动态连接库 `_pywrap_tensorflow.so` 中调用对应的 C++ 函数实现。

```
# tensorflow/bazel-bin/tensorflow/python/pywrap_tensorflow.py
def TF_NewDeprecatedSession(opts, status):
    return _pywrap_tensorflow.TF_NewDeprecatedSession(opts, status)
```

C++ 包装器

在 `pywrap_tensorflow.cpp` 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。

```
// tensorflow/bazel-bin/tensorflow/python/pywrap_tensorflow.cpp
static PyMethodDef SwigMethods[] = {
    // ...
    { "TF_NewDeprecatedSession",
        _wrap_TF_NewDeprecatedSession, METH_VARARGS, NULL},
};
```

最终，`_wrap_TF_NewDeprecatedSession` 将调用 `c_api.h` 对其开放的 API 接口：`TF_NewDeprecatedSession`。也就是说，自动生成的 `pywrap_tensorflow.cpp` 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。

4.4.2 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。

```
// tensorflow/c/c_api.c
TF_DeprecatedSession* TF_NewDeprecatedSession(
    const TF_SessionOptions* opt, TF_Status* status) {
    Session* session;
    status->status = NewSession(opt->options, &session);
    return status->status.ok() ? new TF_DeprecatedSession({session}) : NULL;
}
```

4.4.3 后端系统

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

```
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++ 系统中，`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` 为 `grpc` 开头，则创建 `GrpcSession` 实例，启动基于 RPC 的分布式运行模式。如图4.4.3（第33页）所示。

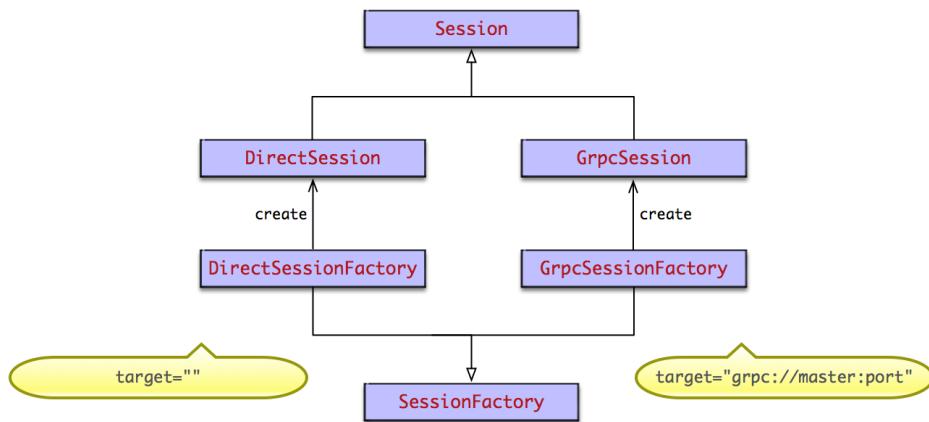


图 4-6 `tensorflow::Session` 创建：抽象工厂方法

4.5 创建/扩展图

随后，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。

4.5.1 编程接口

```

# tensorflow/python/client/session.py
class Session(BaseSession):
    def run(self, fetch_list, feed_dict=None, options=None, run_metadata=None):

```

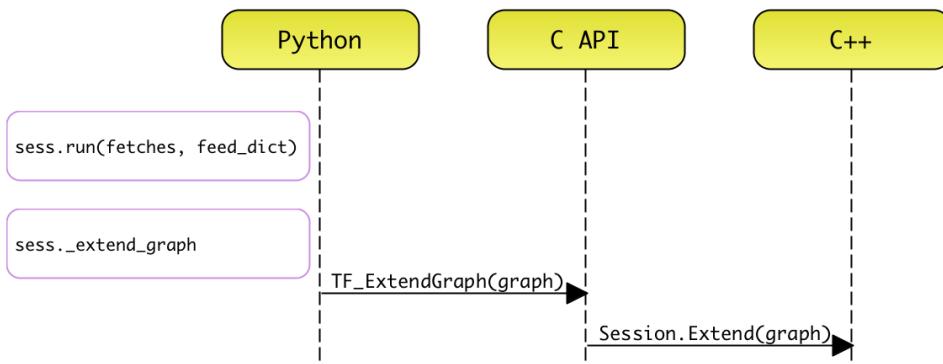


图 4-7 创建图

```

# ignores implements...
self._extend_graph()
with errors.raise_exception_on_not_ok_status() as status:
    return tf_session.TF_Run(
        self, options, feeds, fetches, targets, status, run_metadata)

```

其中，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。

```

from tensorflow.python import pywrap_tensorflow as tf_session

class Session(BaseSession):
    def _extend_graph(self):
        # ignores implements...
        with errors.raise_exception_on_not_ok_status() as status:
            tf_session.TF_ExtendGraph(
                self._session, graph_def.SerializeToString(), status)

```

Python 包装器

```

# tensorflow/bazel-bin/tensorflow/python/pywrap_tensorflow.py
def TF_ExtendGraph(sess, graph_def, status):
    return _pywrap_tensorflow.TF_ExtendGraph(sess, graph_def, status)

```

C++ 包装器

```

// tensorflow/bazel-bin/tensorflow/python/pywrap_tensorflow.cpp
static PyMethodDef SwigMethods[] = {
    // ignore implements...
    { (char *)"TF_ExtendGraph", _wrap_TF_ExtendGraph, METH_VARARGS, NULL},
};

```

4.5.2 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。

```
// 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);
}
```

4.5.3 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。

```
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

首先，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 `Master` 的 `handle`。

```
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 uninitialized,
    // so simply initialize the session with 'graph'.
    return Create(graph);
}
// ignore implements...
}

```

4.6 迭代运行

接着，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。

每次 Step，计算图将正向计算网络的输出，然后反向传递梯度，并完成一次训练参数的更新。首先，后端系统根据 `fetches`, `targets`，对计算图（常称为 Full Graph）进行剪枝，得到一个最小依赖的子图（常称为 Client Graph）。

然后，运行时启动设备分配算法，如果节点之间的边横跨设备，则将该边分裂，插入相应的 `Send` 与 `Recv` 节点，实现跨设备节点的通信机制。

随后，将分裂出来的子图片段（常称为 Graph Partition）注册到相应的 Worker 上，并启动各个 Worker 并发执行这些子图片段。如图 4.6（第36页）所示。

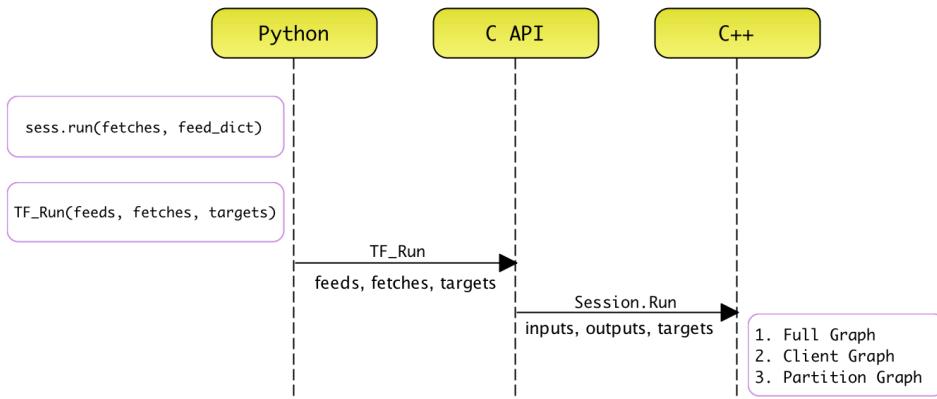


图 4-8 迭代执行

4.6.1 编程接口

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

```

# 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:
        return tf_session.TF_Run(
            self, options, feeds, fetches, targets, status, run_metadata)
        # ignores other implements...
```

Python 包装器

```
# tensorflow/bazel-bin/tensorflow/python/pywrap_tensorflow.py
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

```
// tensorflow/bazel-bin/tensorflow/python/pywrap_tensorflow.cpp
static PyMethodDef SwigMethods[] = {
    // ...
    { (char *)"TF_Run", _wrap_TF_Run, METH_VARARGS, NULL},
};
```

最终，_wrap_TF_Run 将转调 C API 对应的 TF_Run 接口函数。

4.6.2 C API

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。

```
// 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 ntargs,
    // 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...
}
```

4.6.3 后端系统

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

```
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。

4.7 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 4.7（第 39 页）所示。。

4.7.1 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`。

```
# tensorflow/python/client/session.py
from tensorflow.python import pywrap_tensorflow as tf_session
```

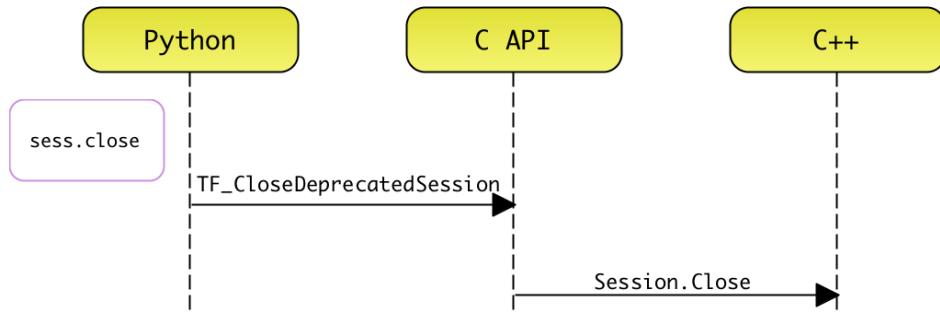


图 4-9 关闭会话

```

class Session(BaseSession):
    def close(self):
        # ignores other implements...
        with errors.raise_exception_on_not_ok_status() as status:
            tf_session.TF_CloseDeprecatedSession(self._session, status)
    
```

Python 包装器

```

# tensorflow/bazel-bin/tensorflow/python/pywrap_tensorflow.py
def TF_CloseDeprecatedSession(sess, status):
    return _pywrap_tensorflow.TF_CloseDeprecatedSession(sess, status)
    
```

C++ 包装器

```

// tensorflow/bazel-bin/tensorflow/python/pywrap_tensorflow.cpp
static PyMethodDef SwigMethods[] = {
    // ...
    { (char *)"TF_CloseDeprecatedSession",
        _wrap_TF_CloseDeprecatedSession, METH_VARARGS, NULL},
};
```

最终，_wrap_TF_CloseDeprecatedSession 将转调 C API 对应的 TF_CloseDeprecatedSession 接口函数。

4.7.2 C API

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

```

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

4.7.3 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。

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

4.8 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 4-10（第 40 页）所示。

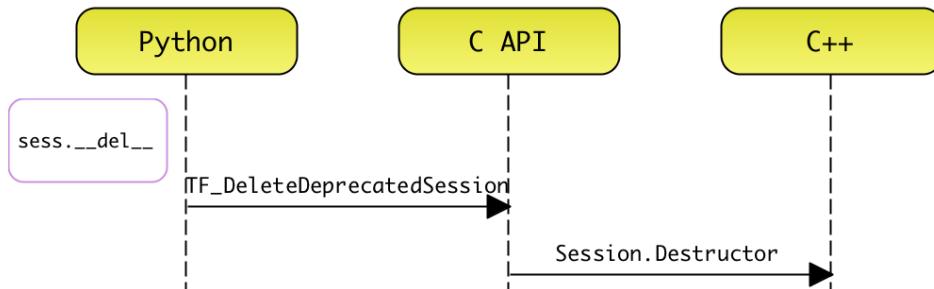


图 4-10 销毁会话

4.8.1 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_DeleteDeprecatedSession`。

```
# 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 = tf_session.TF_NewStatus()
                tf_session.TF_DeleteDeprecatedSession(self._session, status)
            finally:
                tf_session.TF_DeleteStatus(status)
            self._session = None
```

Python 包装器

```
# tensorflow/bazel-bin/tensorflow/python/pywrap_tensorflow.py
def TF_DeleteDeprecatedSession(sess, status):
    return _pywrap_tensorflow.TF_DeleteDeprecatedSession(sess, status)
```

C++ 包装器

```
// tensorflow/bazel-bin/tensorflow/python/pywrap_tensorflow.cpp
static PyMethodDef SwigMethods[] = {
    // ...
    { (char *)"TF_DeleteDeprecatedSession",
        _wrap_TF_DeleteDeprecatedSession, METH_VARARGS, NULL},
};
```

最终，_wrap_TF_DeleteDeprecatedSession 将转调 C API 对应的 TF_DeleteDeprecatedSession 接口函数。

4.8.2 C API

TF_DeleteDeprecatedSession 直接完成 Session(C++) 对象的释放。

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

4.8.3 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。

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

4.9 性能调优

一般地，一个 Session 只能运行一个计算图。如果一个 Session 要运行其他的计算图，必须先关掉 Session，然后再将新图注册到此 Session 中，最后启动新的计算图的执行过程。

事实上，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。如图 7-1（第 64 页）所示。

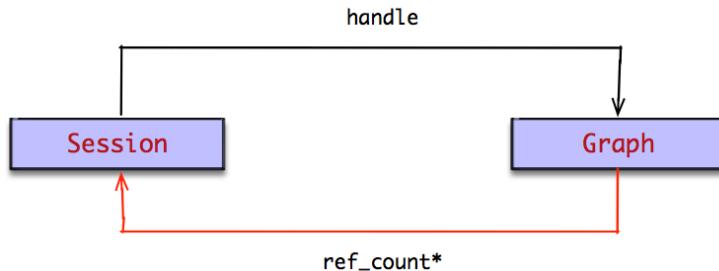


图 4-11 计算图：会话引用计数器技术

当需要删除图实例时，如果引用的 `Session` 数目为 0，则删除该图实例；否则，不删除图实例。

R 因此，之前实现操作 `Session` 的 C API 都被标识为已废弃（直至 1.2 版本还未删除），实现提供了一套新的 C API 管理 `Session` 的生命周期，以便改善系统的性能。

4.9.1 创建会话

在 `Session` 创建时，在该图实例的 `Session` 数目加 1。

```

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 NULL;
    }
}
  
```

4.9.2 销毁会话

在 `Session` 销毁时，在该图实例的 `Session` 数目减 1。

```

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;
```

4.9.3 删除图实例

当需要删除图实例时，如果引用的 Session 数目为 0，则删除该图实例；否则，不删除该图实例。

```
void TF_DeleteGraph(TF_Graph* g) {
    g->mu.lock();
    g->delete_requested = true;
    const bool del = g->num_sessions == 0;
    g->mu.unlock();
    if (del) delete g;
}
```


Part III

编程模型

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

- Martin Flower



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。

5.1 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。

5.1.1 Operation

`Operation` 表示某种抽象计算，它以零个或多个 `Tensor` 作为输入，经过计算后，输出零个或多个 `Tensor`。

如图5-1（第48页）所示。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，例如名称，输入输出的属性名等信息。而 `NodeDef` 描述了 OP 的动态属性值信息。

`Operation` 根据上游节点的输出，经过计算输出到下游。其中，`Operation` 的输入和输出以 `Tensor` 的形式存在。从而上下游产生了数据依赖关系。

此外，`Operation` 可能持有上游的控制依赖边的集合，表示其潜在的计算依赖关系。

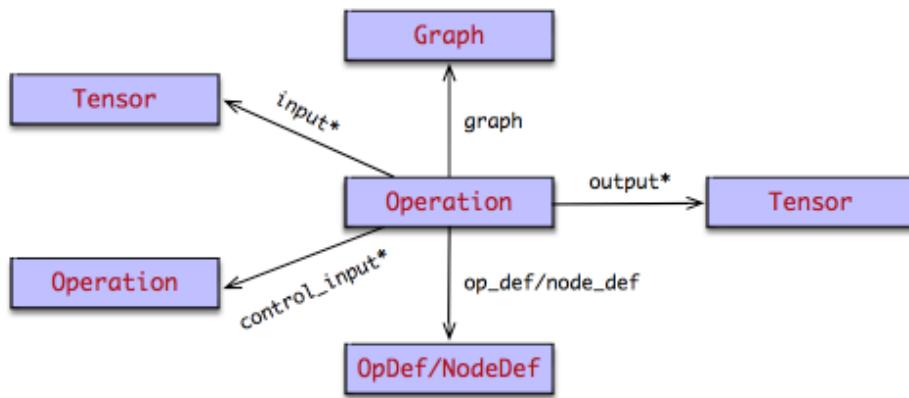


图 5-1 领域对象：Operation

5.1.2 Tensor

一个 `Tensor` 表示 `Operation` 的某个输出的符号句柄，它并不持 `Operation` 输出的真实数据。可以通过 `Session.run` 计算得到 `Tensor` 所持有的真实数据。

如图 5-2（第48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-消费者」的关系。

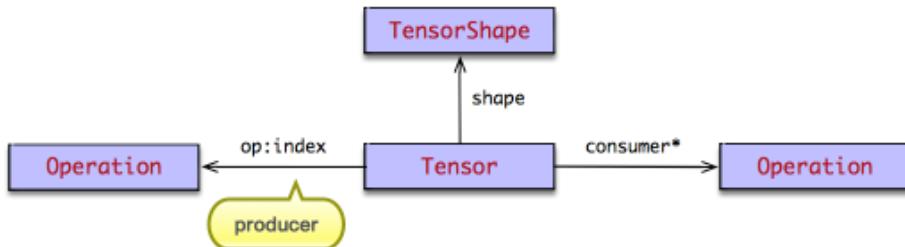


图 5-2 领域对象：Tensor

其中，`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。计算图以 `Tensor` 为边，构建 `Operation` 之间的数据连接，从而实现了整个计算图的数据依赖构建。

生产者与消费者

如图 5-3（第49页）所示。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 `index` 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 5-3 `Tensor`: 生产者-消费者关系

建立关联

最后，参看 `Operation` 与 `Tensor` 的部分实现，很容易找两者「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):
        # _inputs as consumers
        self._inputs = list(inputs)
        for a in self._inputs:
            a._add_consumer(self)

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

同样地，`Tensor` 在构造函数中持有作为上游的的生产者 `Operation`，及其它在该 `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)
  
```

5.1.3 Graph

如图 A-1 (第 142 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。

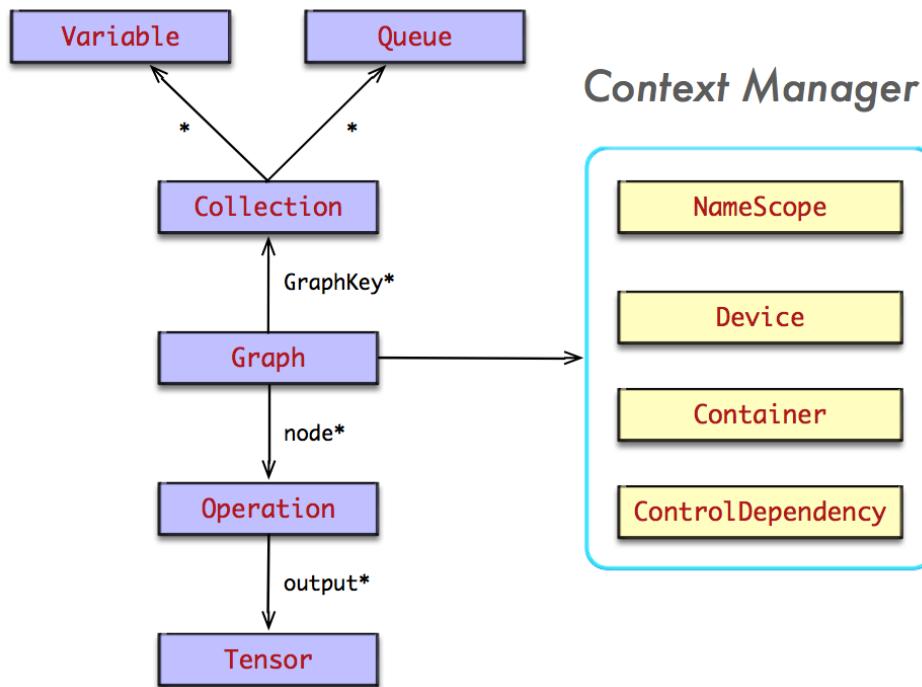


图 5-4 领域对象：Graph

5.1.4 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 5-5（第 51 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。

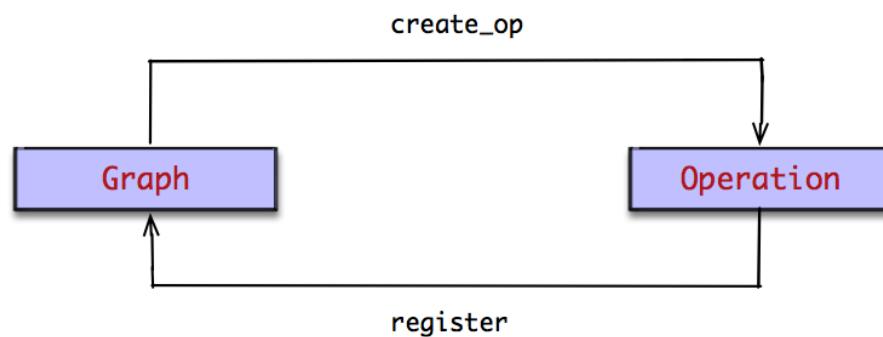


图 5-5 Graph: OP 工厂 + OP 仓库

OP 构造器

如图 5-6(第 52 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。

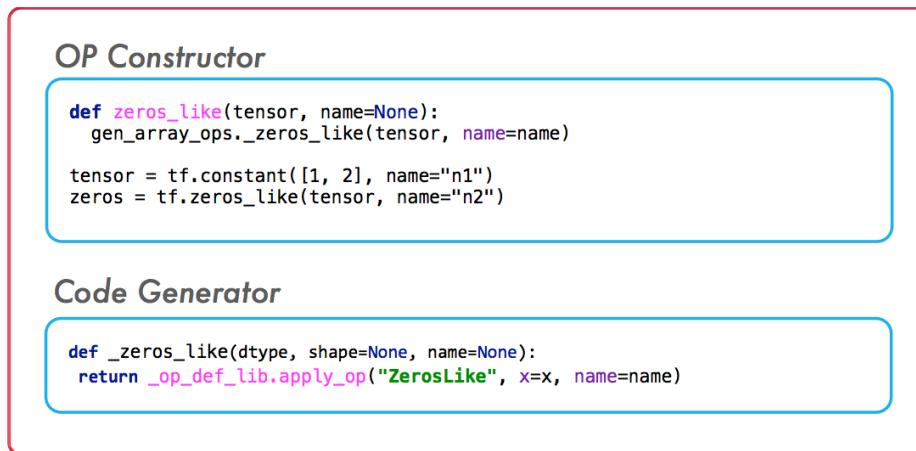


图 5-6 OP 构造器与代码生成器

构造 OpDef 与 NodeDef

然后, 如图 5-7 (第 53 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。

5.2 后端 C++

在 C++ 后端, 计算图是 TensorFlow 领域模型的核心。

5.2.1 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。

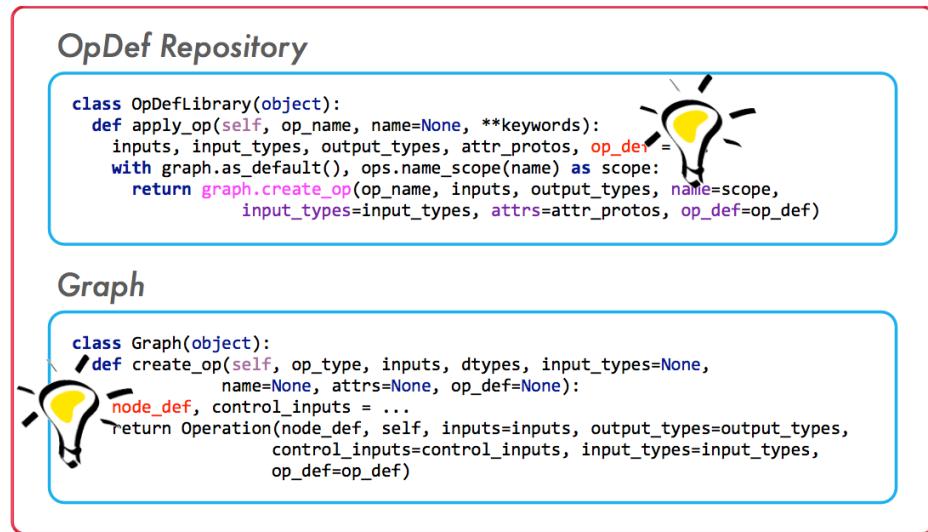


图 5-7 创建 Operation 实例: 创建 OpDef, NodeDef 实例

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。

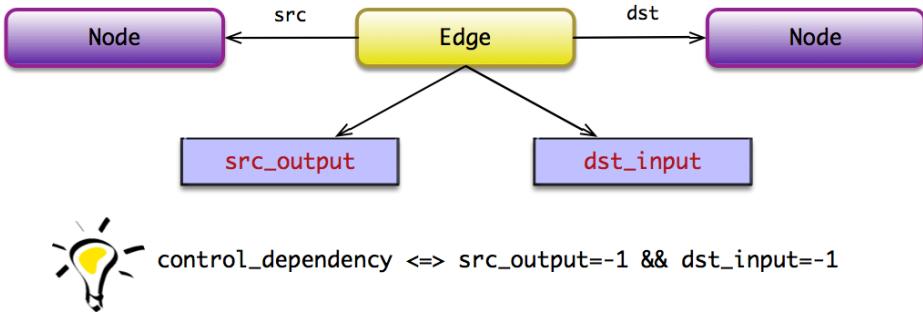


图 5-8 领域对象: Edge

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。

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。

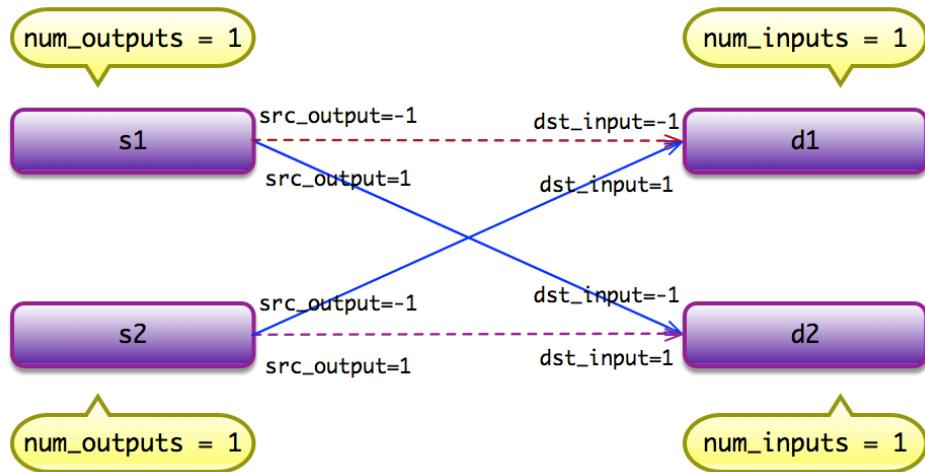


图 5-9 边例子

```
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。

5.2.2 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。

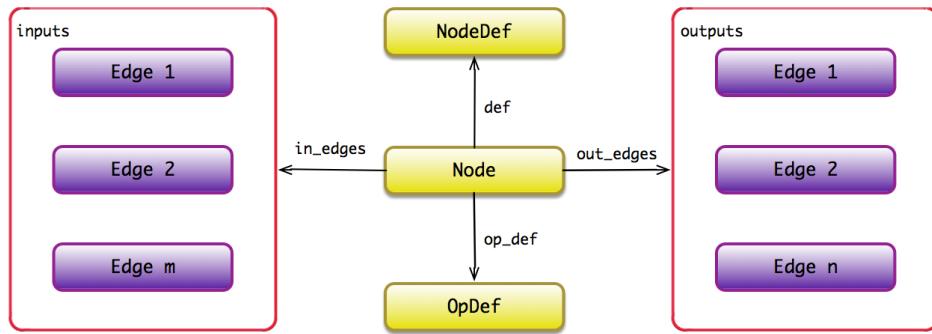


图 5-10 领域对象：Node

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();
    }
```

5.2.3 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 5-11 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。

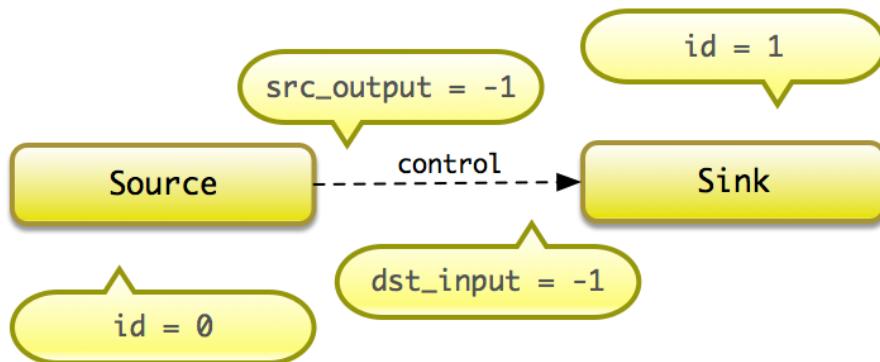


图 5-12 空图

`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。

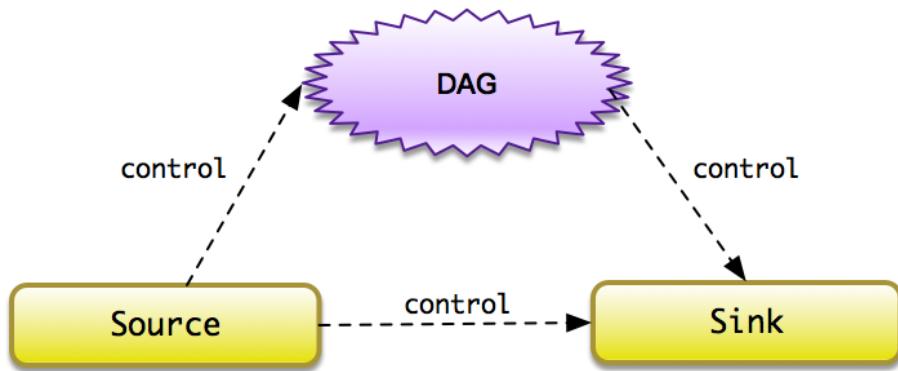


图 5-13 非空图

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);
}
```

5.2.4 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。

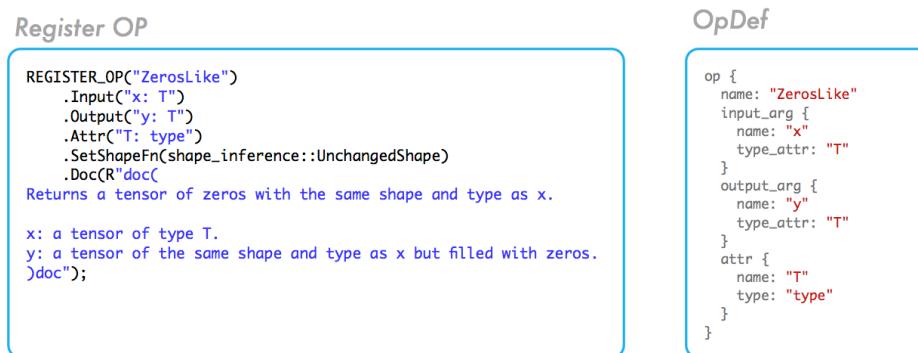


图 5-14 OpDef 注册：使用 REGISTER_OP

5.3 图传递

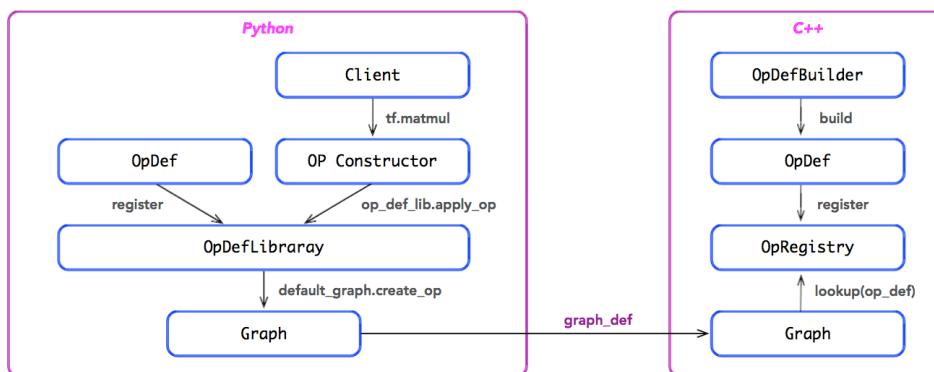


图 5-15 图的序列化与反序列化

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

- Martin Flower

6

设备

6.1 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。

6.1.1 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 GPU 的 0 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` 指定该 OP 与指定的节点被同时放置在特定设备上。例如：

```
| @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` 使用如下 5 个标识确定一个设备规范。

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)
```

6.1.2 上下文管理器

常常使用上下文管理器 (Context Manager) 指定 OP 默认的设备类型。

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

此处，显式地指定了图实例 `g`。其中，`device` 是 `Graph` 的一个方法，它定义实现了一个上下文管理器，从而实现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。

```
| def device(self, device_name_or_function):
|     device_function = to_device_function(device_name_or_function)
|     try:
|         self._device_function_stack.append(device_function)
|         yield
|     finally:
|         self._device_function_stack.pop()
```

包装器

当用户使用 `device` 未显式地指定图实例，其使用隐式存在的默认图实例。

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

也就是说，`device` 函数事实上是对 `get_default_graph().device` 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)
```

合并

可以对两个不同范围的设备规范进行合并。例如：

```
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

在合并两个不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。例如：

```
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.
```

6.1.3 设备函数

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函数，它提供了一种更加灵活的扩展方式指定设备。例如：

```
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.
```

事实上，当传递字符串，或者 `DeviceSpec` 给 `device` 函数时，首先会对该字符串，或 `DeviceSpec` 做一个简单的适配，使其具备设备函数的特性。

```
def device(self, device_name_or_function):
    if (device_name_or_function is not None
        and not callable(device_name_or_function)):
        device_function = pydev.merge_device(device_name_or_function)
    else:
        device_function = device_name_or_function

    # ignore others implements.
```

其中，`pydev.merge_device` 的设计是一种典型的函数式设计，它返回了一个函数，使得 `device` 函数在后续处理不用区分字符串或函数两种类型。

```
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: current_device takes precedence.
        copy_spec.merge_from(current_device)
        return copy_spec
    return _device_function
```

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

- Martin Flower

7 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。

7.1 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。

7.1.1 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()
```

7.1.2 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)
```

7.1.3 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。

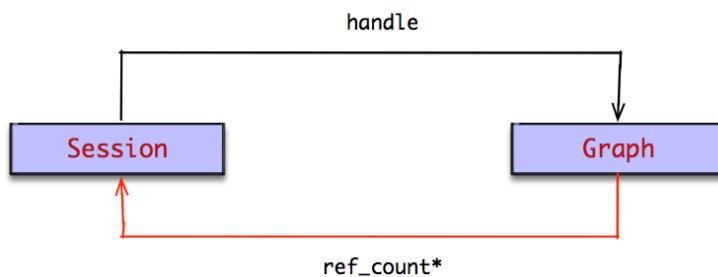


图 7-1 优化技术：会话实例的引用计数器

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::Session, TF_Graph>`，它们之间是一对一的关系。其中，`tensorflow::Session` 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;  
| }
```

7.2 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。

7.2.1 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)
```

7.2.2 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)
```

7.2.3 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)

```

7.3 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。

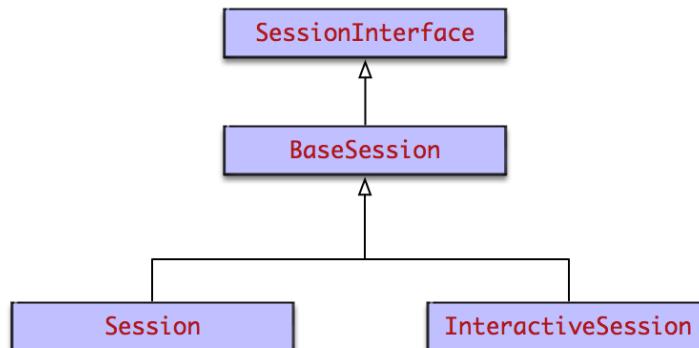


图 7-2 Session：类层次结构

7.3.1 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()

```

7.3.2 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)

```

7.3.3 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

8

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。

8.1 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 8-1 (第 72 页) 所示。

事实上，正如图 8-2 (第 72 页) 所示，实际的计算图要复杂得多，让我们从头说起。

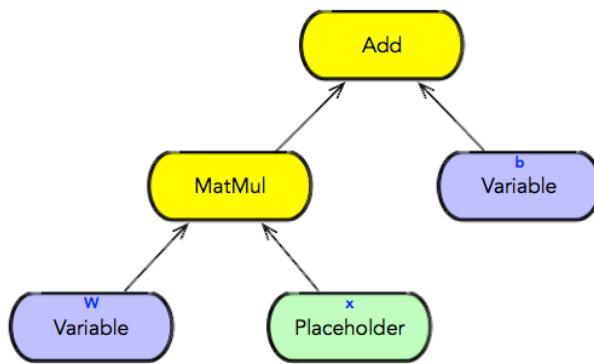


图 8-1 计算图：线性加权和

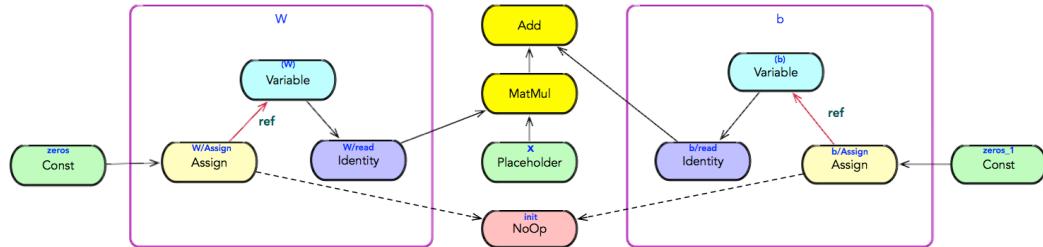


图 8-2 计算图：线性加权和

8.2 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。

8.2.1 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)
```

8.2.2 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：

8.2.3 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。

正如上例 `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。

8.2.4 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。

8.2.5 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。

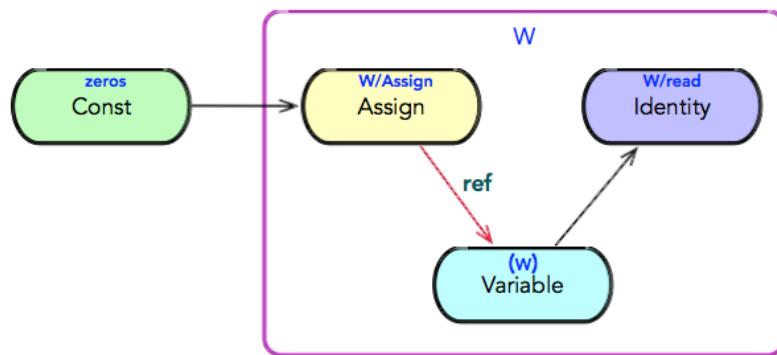


图 8-3 变量子图

8.2.6 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。

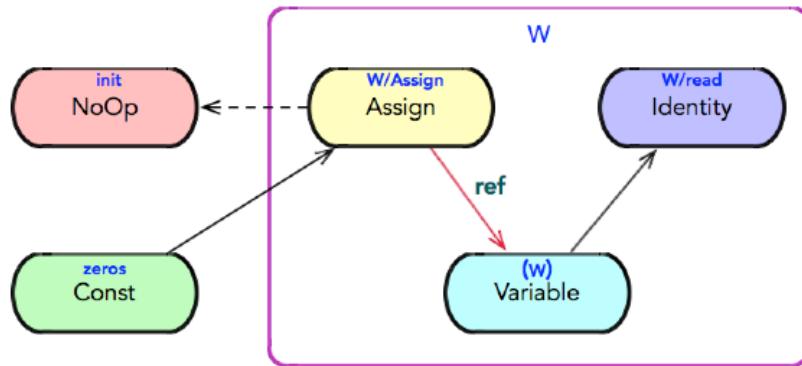


图 8-4 初始化 OP

8.2.7 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"
      }
    }
  }
}
```

8.2.8 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 链接，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。

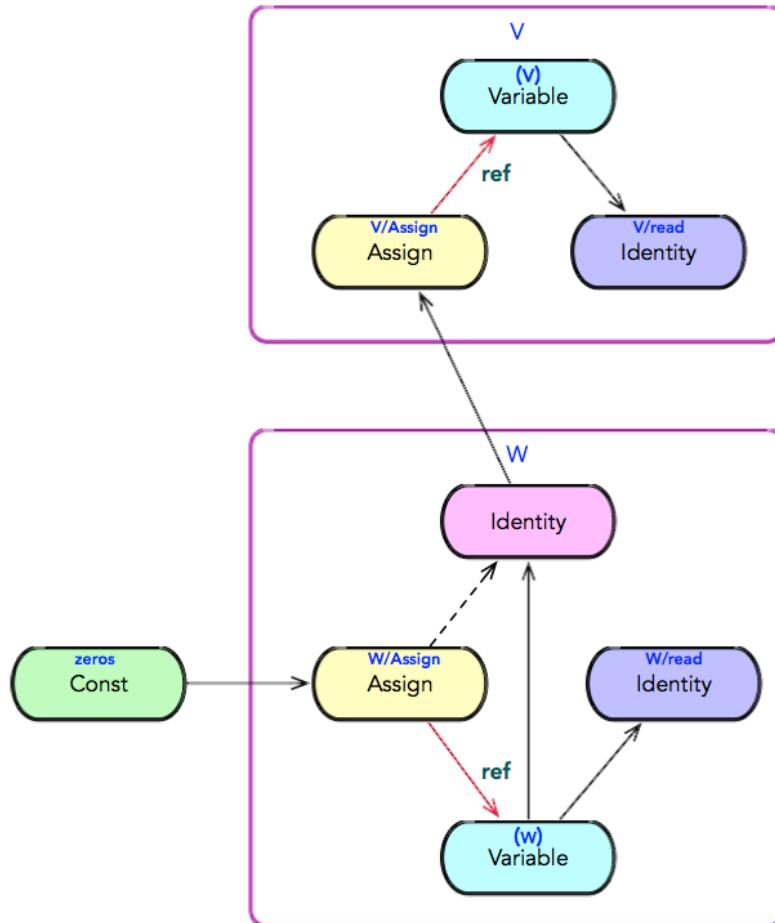


图 8-5 初始依赖

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。

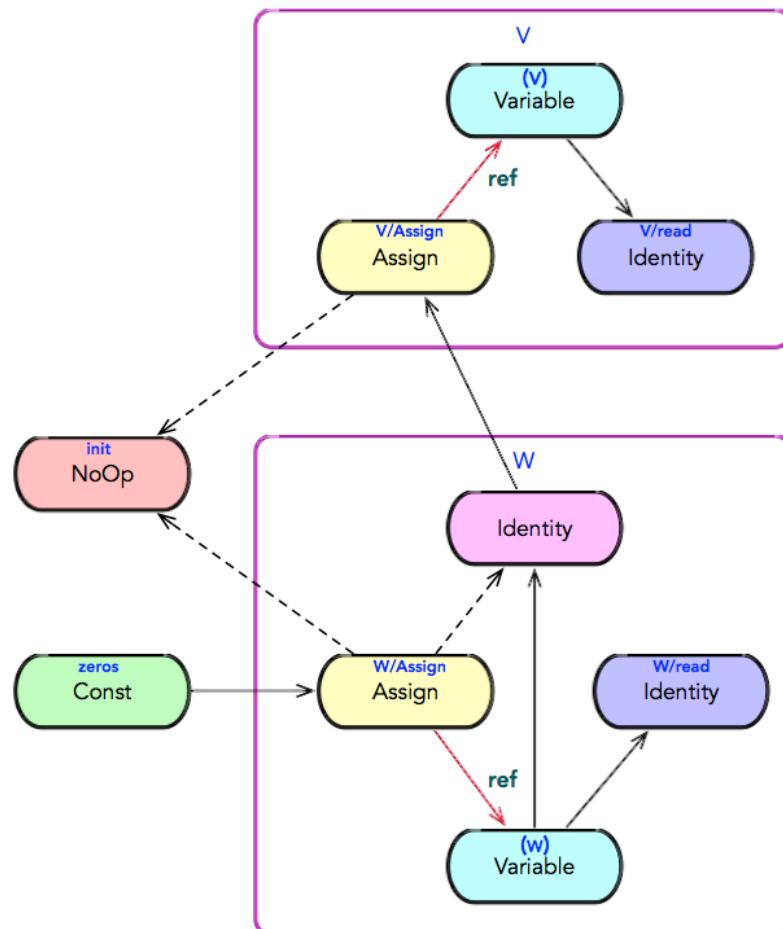


图 8-6 初始化 OP

8.2.9 初始器列表

可以使用 `variables_initializer` 构建变量列表的初始器列表。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器列表可以如下构造。

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

8.3 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。

8.3.1 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)
```

8.3.2 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。

8.3.3 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)
```

8.3.4 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)
```

8.4 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：

8.4.1 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)
```

8.4.2 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)
```

8.4.3 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
```

8.4.4 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")
```

8.4.5 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

9

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`。

9.1 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。

9.1.1 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。

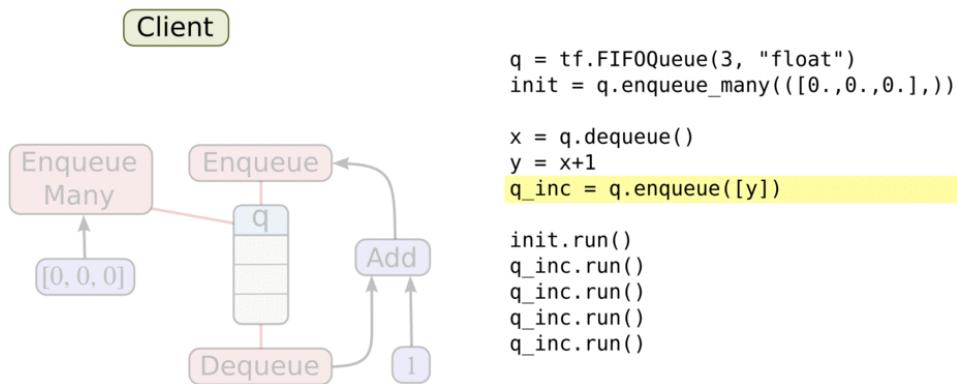


图 9-1 图构造期

执行 **EnqueueMany** 操作后，计算图的状态如下图所示。

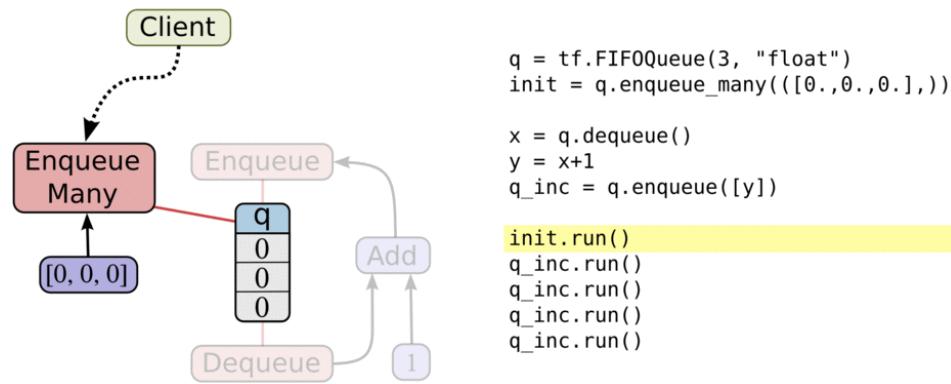


图 9-2 图执行期：执行一次 EnqueueMany

执行第一步 **Enqueue** 后，计算图的状态如下图所示。

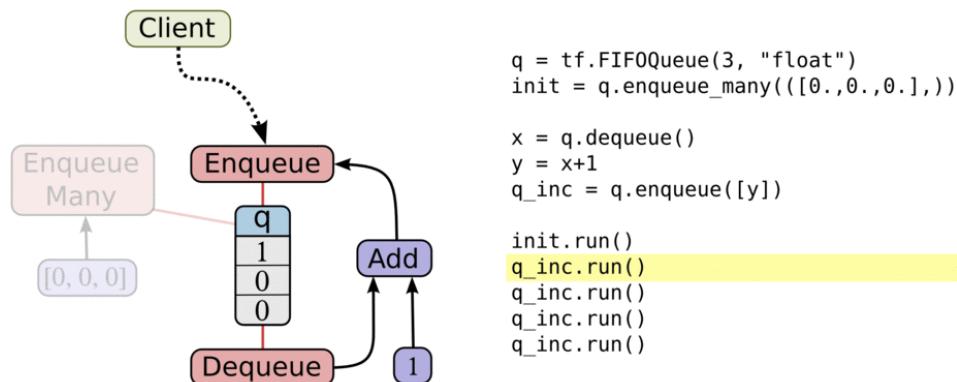


图 9-3 图执行期：执行一次 Enqueue

9.1.2 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。

9.2 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

9.2.1 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)
```

9.2.2 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)

```

9.2.3 实战：LoopThread

9.3 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。

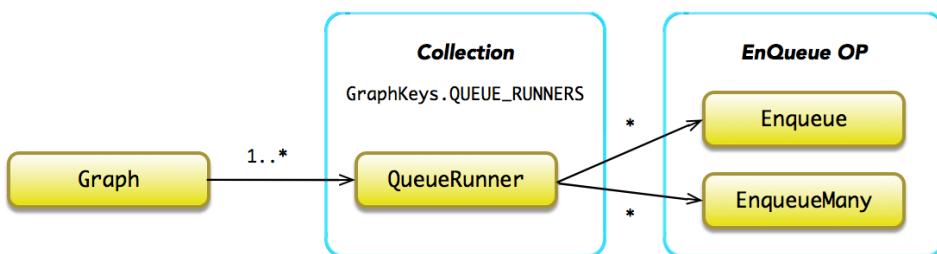


图 9-4 TensorFlow 系统架构

9.3.1 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)

```

9.3.2 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。

9.3.3 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。

Part IV

运行模型

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

- Martin Flower

10

分布式 TensorFlow

10.1 运行模式

TensorFlow 实现了两种基本的运行模式：

1. 本地模式：Client, Master, Worker 部署在同一台机器上，运行在单独的进程上；
2. 分布式模式：Client, Master, Worker 部署在不同机器上，运行在不同的进程上。

10.1.1 本地模式

对于本地模式，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 收到计算图执行消息后，启动计算图的剪枝操作；再将子图传递给 Worker，然后触发 Worker 执行子图。

Worker 根据本地计算设备资源，再将计算子图进行剪枝，分裂，将子图片段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片段。

因为，Client, Master, Worker 在同一进程中，各服务之间是函数调用关系。

10.1.2 分布式模式

对于分布式模式，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，然后触发各个 Worker 进程执行子图。

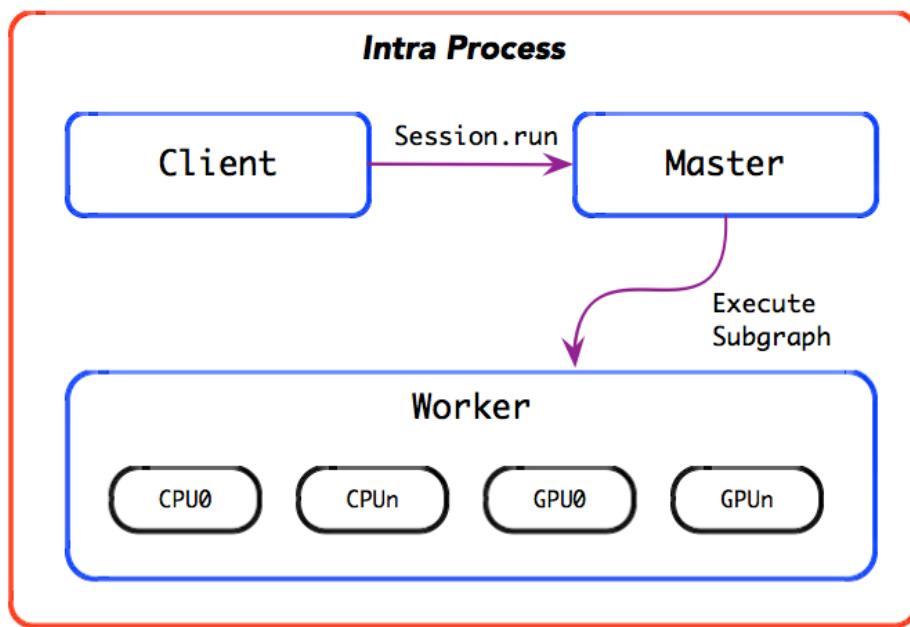


图 10-1 本地模式

Worker 进程收到子图注册的消息后，根据本地计算设备资源，再将计算子图进行剪枝，分裂，将子图片段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片段；如果 Worker 之间存在数据交换，可以通过设备间通信完成交互。

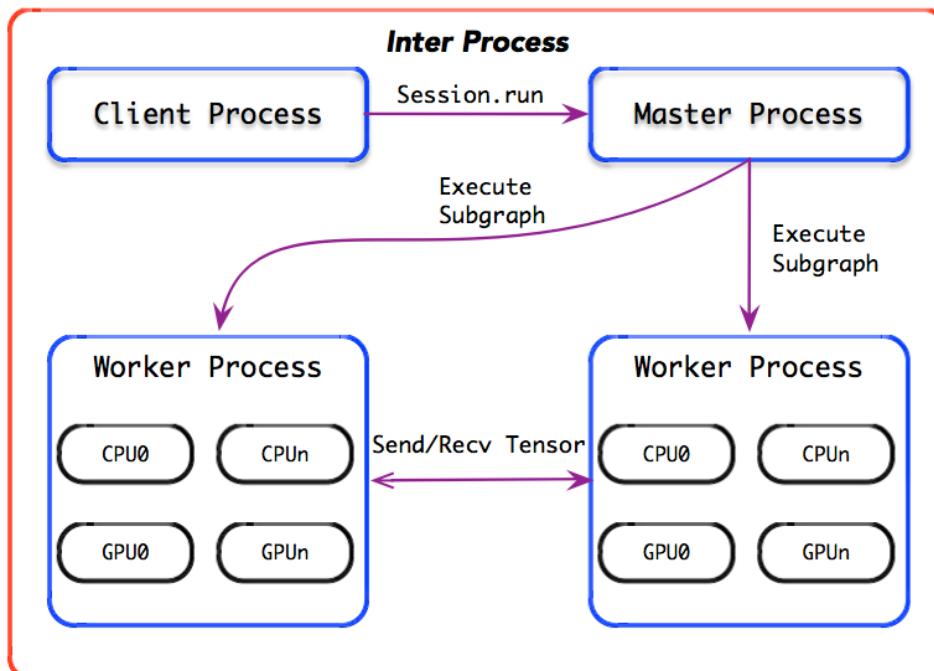


图 10-2 分布式模式

10.2 领域模型

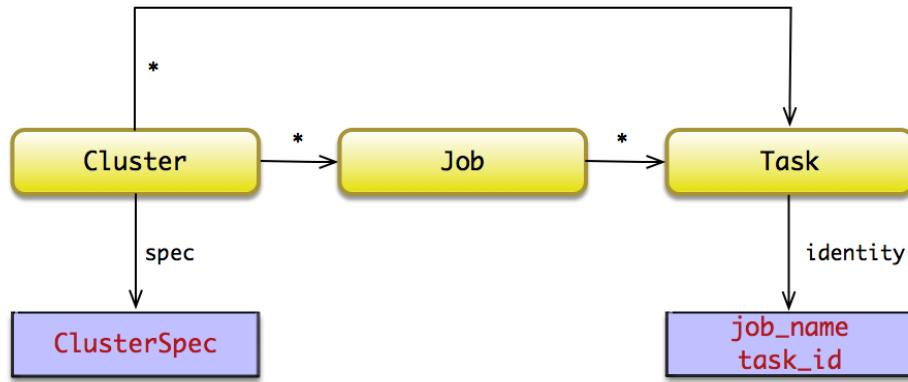


图 10-3 分布式模式：领域模型

10.2.1 Cluster

一个 Cluster 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。

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)。

其中，Cluster 使用 ClusterSpec 进行描述，后文详细描述。

10.2.2 Job

将目的相同的 Task 归类为一个 Job。一般地，在分布式深度学习训练过程中，将任务分为两类 Job：

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。

10.2.3 Task

在 Worker 节点上执行的独立任务。一般地，每个 Task 启动一个进程，并在其之上运行一个 `tf.train.Server` 实例。

其中，Task 使用 Job 名称与 Task 索引唯一标识：`job_name:task_index`。

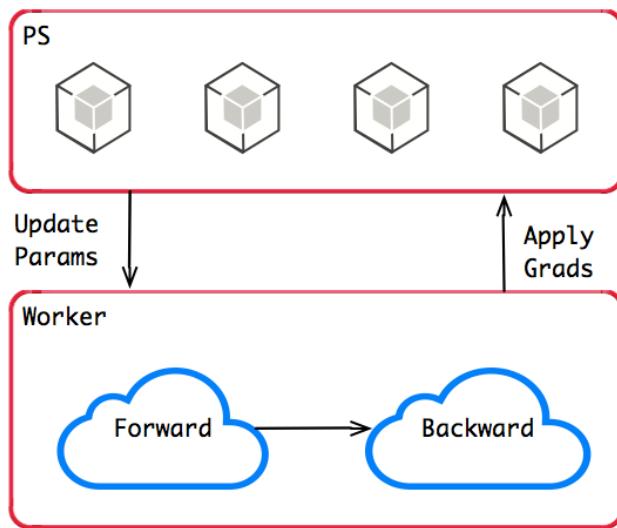


图 10-4 分布式模式：PS 任务与 Worker 任务

10.2.4 Server

Server 表示 Task 在运行时的服务进程，它对外提供 Master Service 服务和 Worker Service 服务。

10.2.5 Master Service

Master Service 是一个 RPC 服务，客户端通过 Session 目标 (target) 获取远端的一组分布式的设备集，并执行计算图的计算。

Master Service 定义了接入 Master 的接口定义，即 `master_service.proto` 中定义的接口；它负责协调和控制多个 Worker Service 的执行过程。

当 Client 根据 `target` 接入 Server 实例，Server 扮演了 Master 的角色，对外提供 Master Service 服务；其中，Client 与 Master 之间的交互遵循 Master Service 的接口规范。

10.2.6 Worker Service

Worker Service 也是一个 RPC 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，则通过设备间通信完成交互。其中，Master 与 Worker

之间，Worker 与 Worker 之间的交互遵循 Worker Service 的接口规范。

特殊地，Client 与 Master 可能在同一进程之内；Master 与本地的 Worker 也可能在同一进程中。

10.3 组建集群

为每个 Task 创建一个 Server，并启动相应的 Server 服务，如此便很容易地组建了一个 TensorFlow 集群。也就是说，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。

10.3.1 ClusterSpec

`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。因此，Task 索引在 Job 的 Task 集合中从 0 开始，按序自增的。

10.3.2 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`。

10.4 Server

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 Master Service 服务和 Worker Service 服务。

10.4.1 领域模型

一般地，一个 Task 实例上启动一个 Server 实例。因此，一个 Server 实例通过 `job_name`, `task_index` 唯一标识，与 Task 实例一一对应。

另外，一个 Server 实例通过 `ClusterSpec` 与其他 Server 实例实现互联。

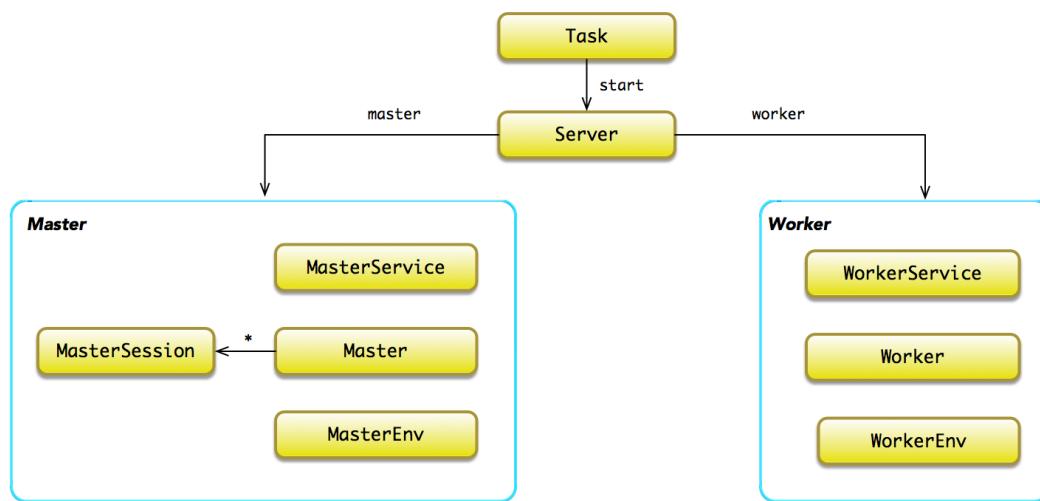


图 10-5 Server 架构

10.4.2 Protobuf 描述

当 `protocol` 为 `grpc` 时，则该 Server 实例将实现基于 GRPC 的服务；此外，可以通过 `ConfigProto` 实现运行时的参数配置。

```
message ServerDef {  
    ClusterDef cluster = 1;  
  
    string job_name = 2;  
    int32 task_index = 3;  
  
    ConfigProto default_session_config = 4;  
    string protocol = 5;  
}
```

10.4.3 服务互联

一个 Server 实例通过 `tf.train.ClusterSpec` 与集群中的其他 Server 实例实现互联。

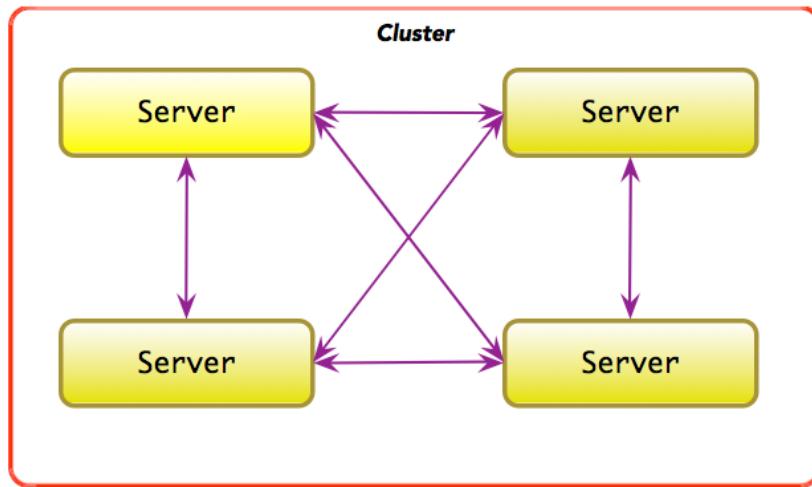


图 10-6 服务互联

当客户端接入其中一个 Server，此时它扮演了 Master 的角色，其他 Server 则扮演了 Worker 的角色。在分布式的深度学习训练模式中，存在多个客户端分别接入不同的 Server 实例。此时，客户端接入的 Server 实例，同时扮演了 Master 和 Worker 的角色。

特殊地，Client 与 Master 可以部署在同一个进程中。此时，Client 与 Master 之间的交互更加简单，避免了 GRPC 的交互。

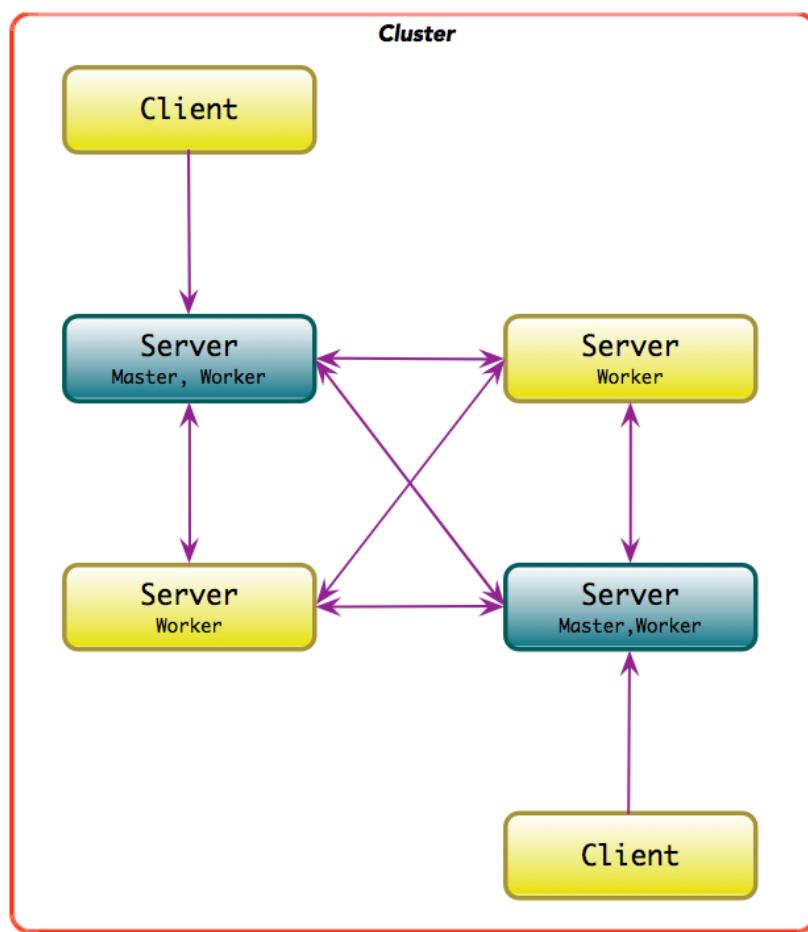


图 10-7 多客户端接入

Part V

模型训练

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

- Martin Flower

11

BP 算法

11.1 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)
```

11.1.1 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`。

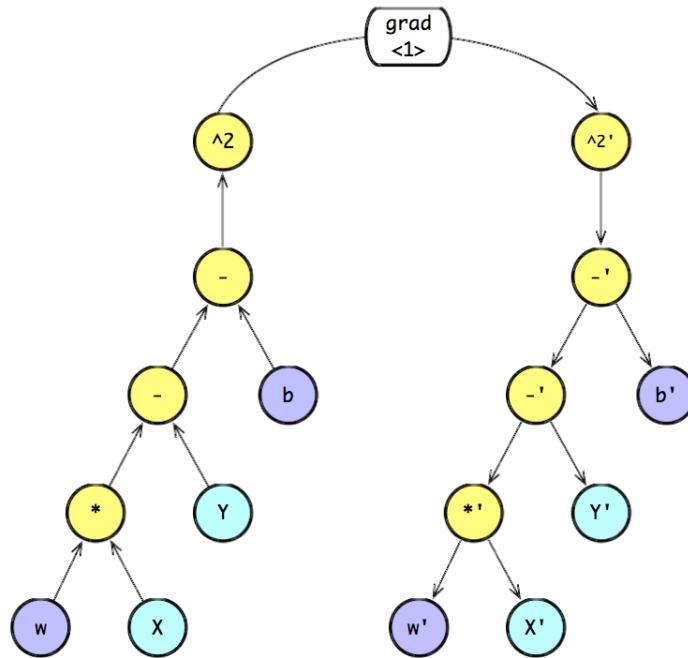


图 11-1 构造反向传播子图

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 11-2 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`。

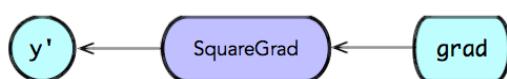


图 11-3 Square 函数：反向传播子图

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。

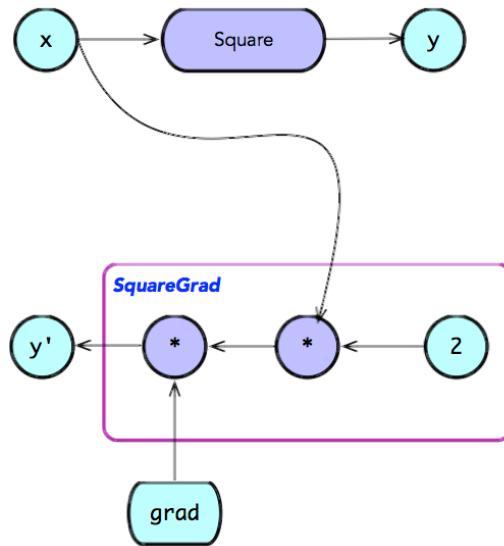


图 11-4 `Square` 函数：反向传播子图

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 $y'=y\exp(x)$ ，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，该局部子图还包含一个节点。

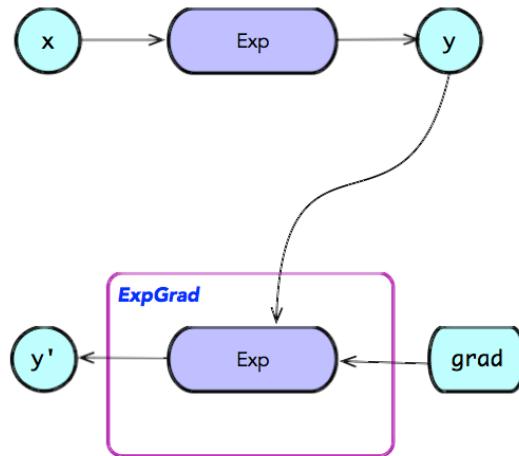


图 11-5 Exp 函数：反向传播子图

11.1.2 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。

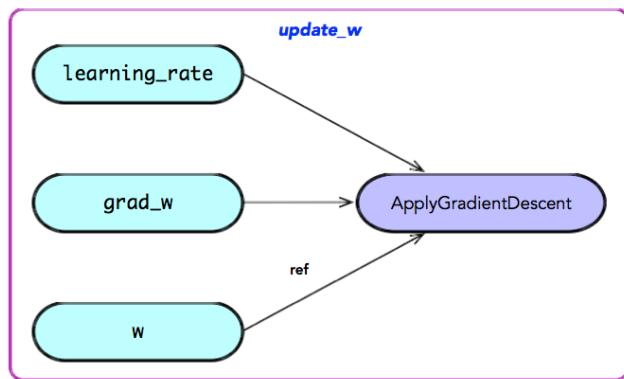


图 11-6 参数更新子图

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。

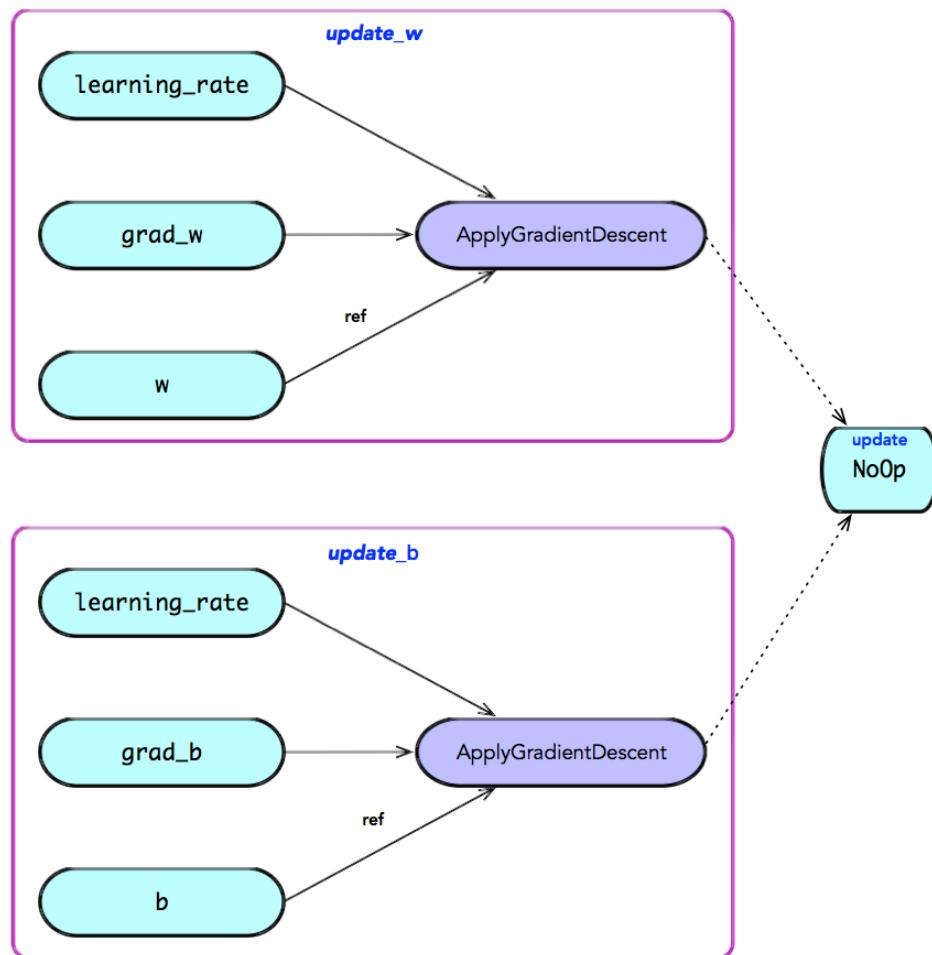


图 11-7 参数更新汇总

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。

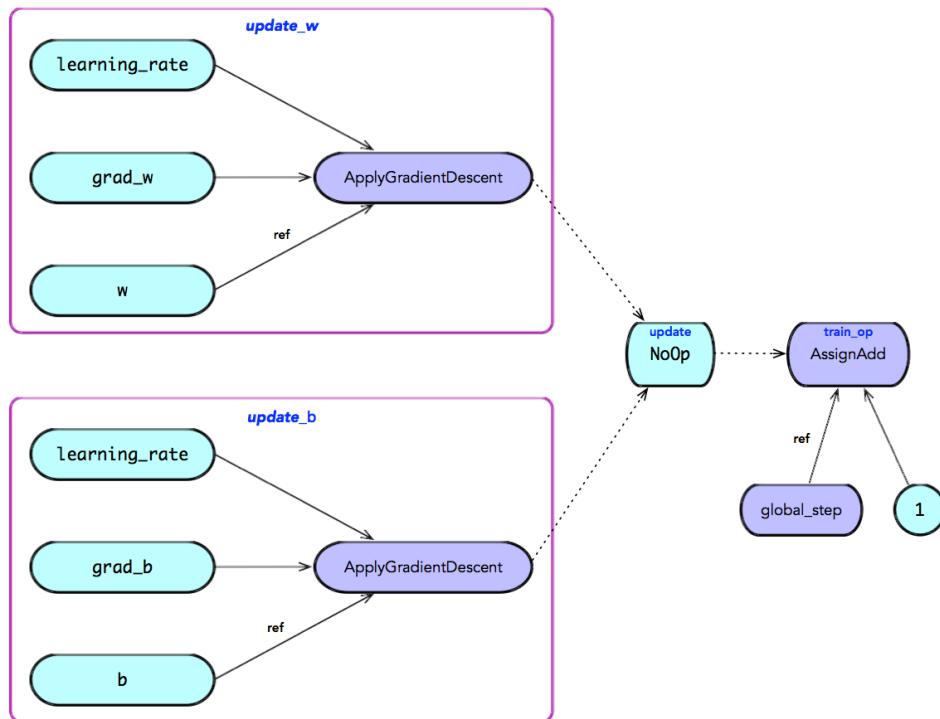


图 11-8 train_op

工作流

如图 11-9（第 110 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。

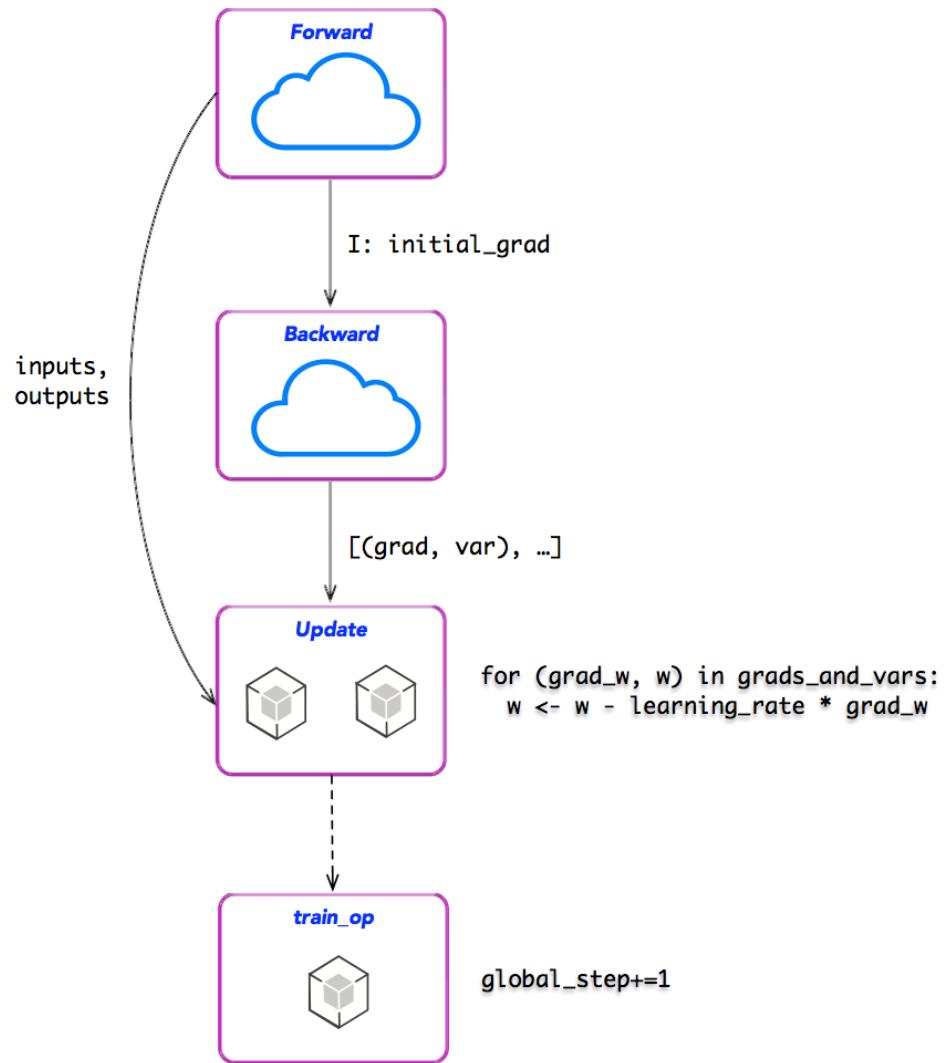


图 11-9 模型训练的工作流

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

- Martin Flower

12

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。

12.1 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。

12.2 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')
```

12.2.1 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)
```

12.2.2 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)
```

12.2.3 批次预加载

可以构建 Pipeline，结合数据预加载机制，实现样本的批次加载。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。

12.3 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。

12.3.1 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。

12.3.2 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
```

12.3.3 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)
```

12.3.4 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
```

12.3.5 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)
```

12.4 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。

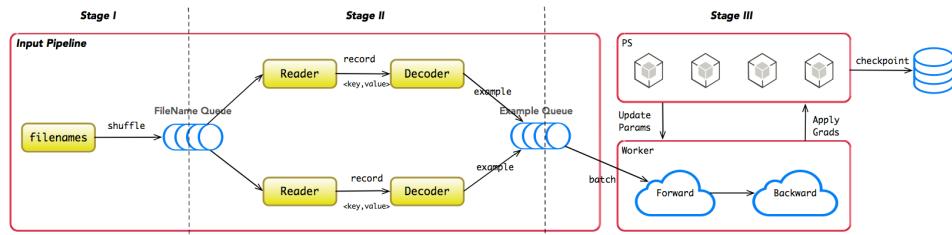


图 12-1 模型训练工作流

12.4.1 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。

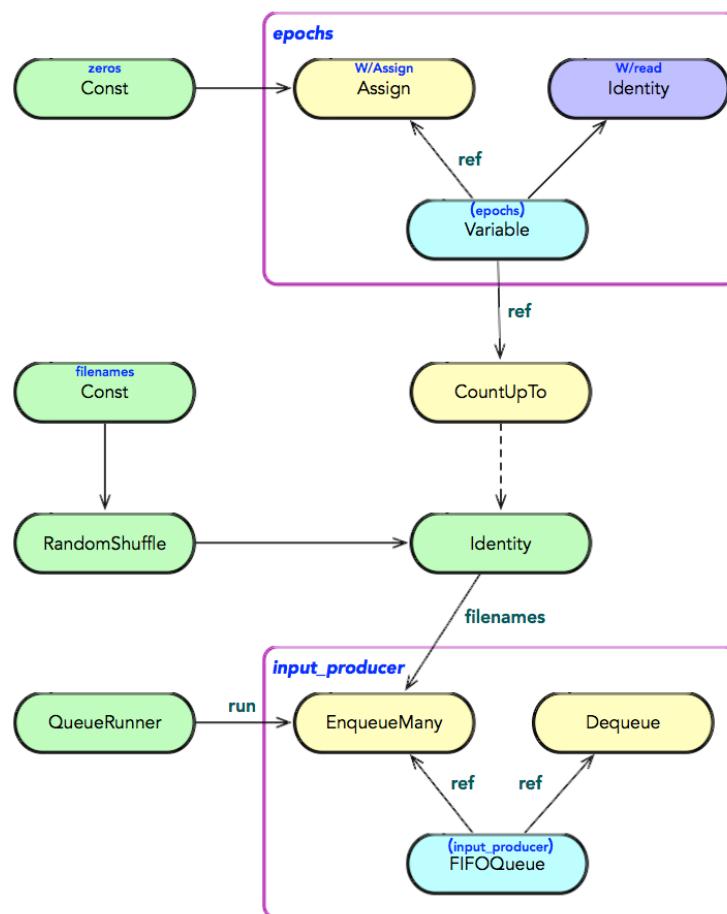


图 12-2 阶段 1：模型训练工作流

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。

12.4.2 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。

实际上，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。

12.4.3 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。

12.4.4 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

13

Saver

13.1 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。

13.1.1 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, 'ckpt')
```

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('ckpt.meta')  
    saver.restore(sess, 'ckpt')
```

13.1.2 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`) 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。

13.1.3 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。

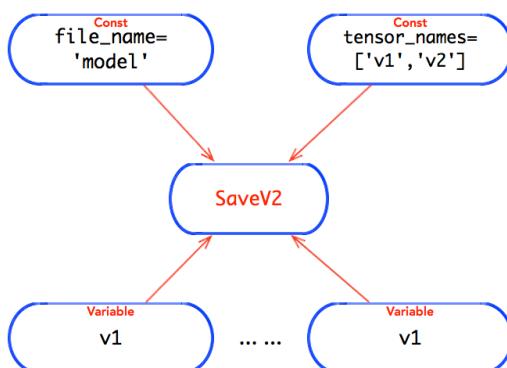


图 13-1 Saver: 持久化模型

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。

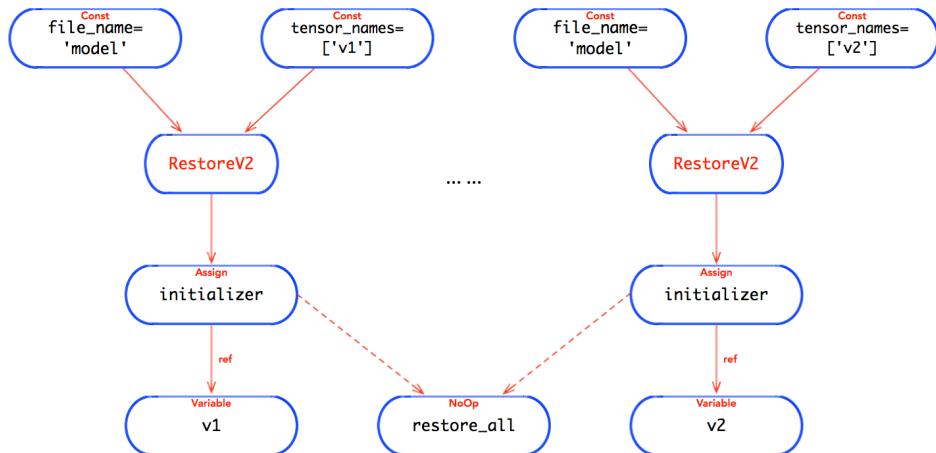


图 13-2 Saver: 恢复模型

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

- Martin Flower

14

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副本（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。

14.1 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。

14.1.1 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)
```

14.1.2 使用工厂

使用 `MonitoredTrainingSession` 的工厂方法，可以简化 `MonitoredSession` 的创建过程。

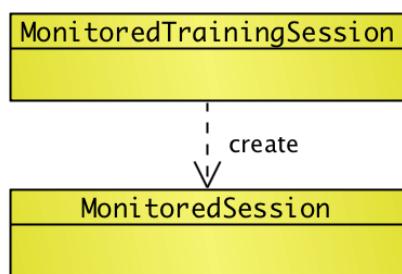


图 14-1 `MonitoredTrainingSession`: 工厂方法

```

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)

```

14.1.3 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。

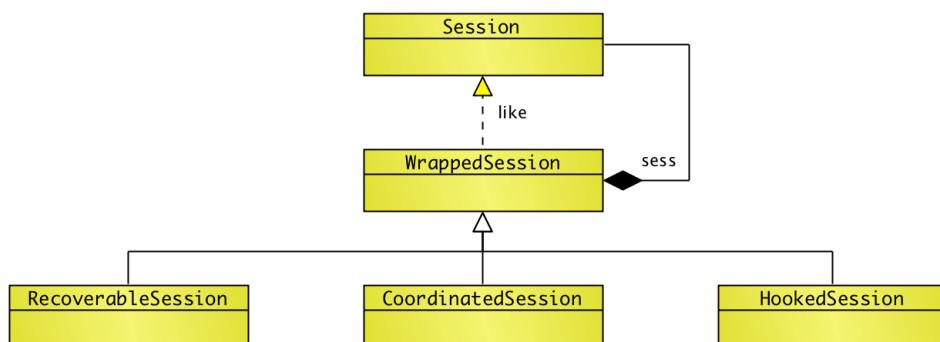


图 14-2 `MonitoredSession`: 装饰器

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)))))

```

14.2 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。

14.2.1 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 `Checkpointing` 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。

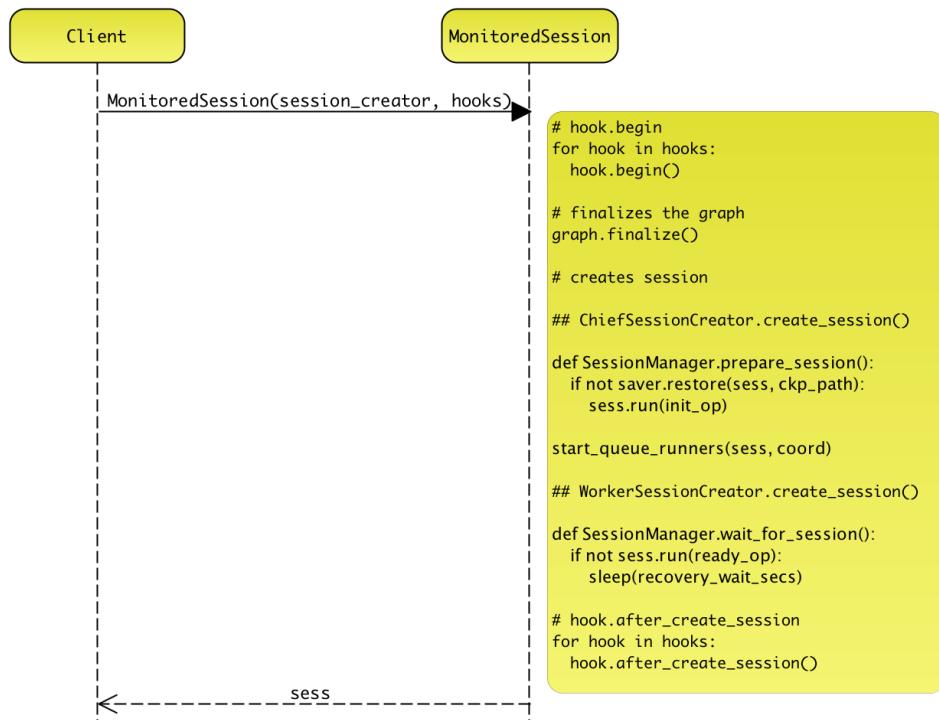


图 14-3 MonitoredSession：初始化

14.2.2 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。

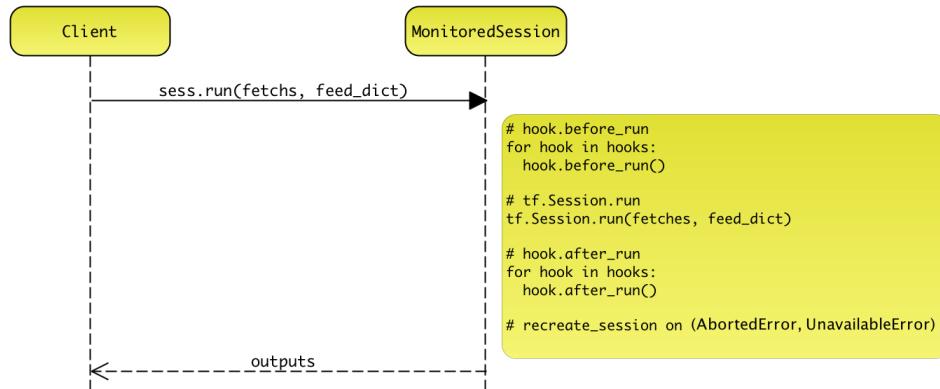


图 14-4 MonitoredSession：执行

14.2.3 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。

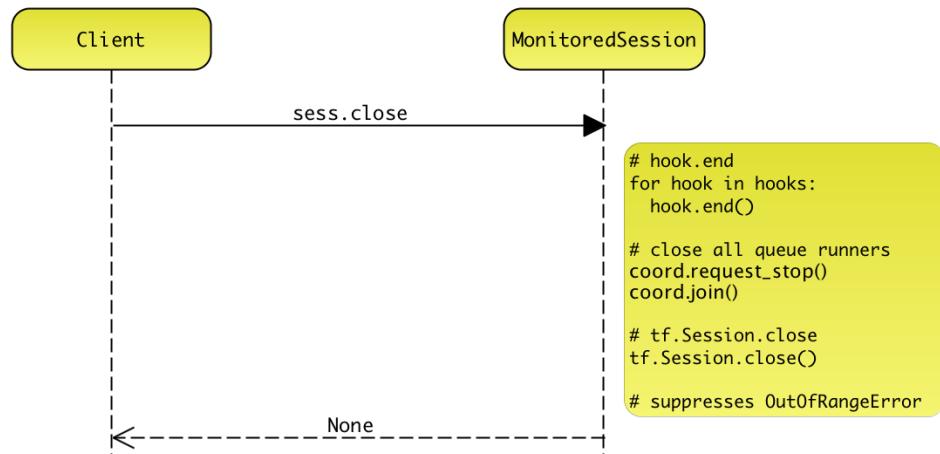


图 14-5 MonitoredSession：关闭

14.3 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，存在两种类型的 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。

14.3.1 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)
```

14.3.2 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。

14.3.3 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。

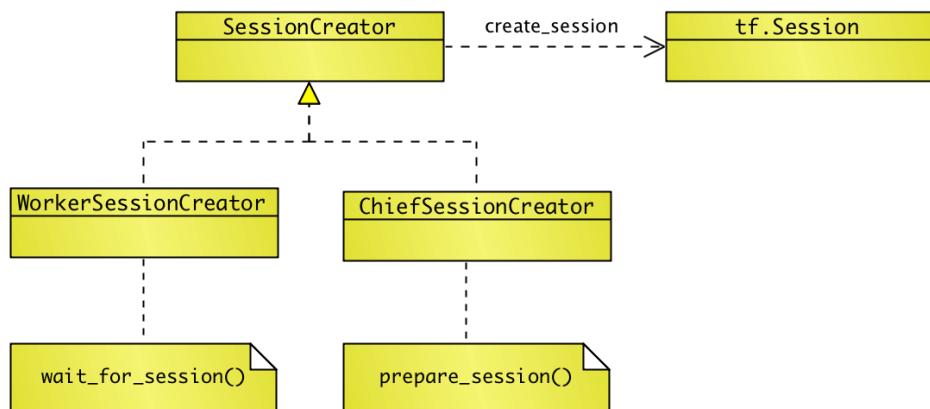


图 14-6 SessionManager

14.3.4 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。

14.3.5 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。

14.3.6 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。

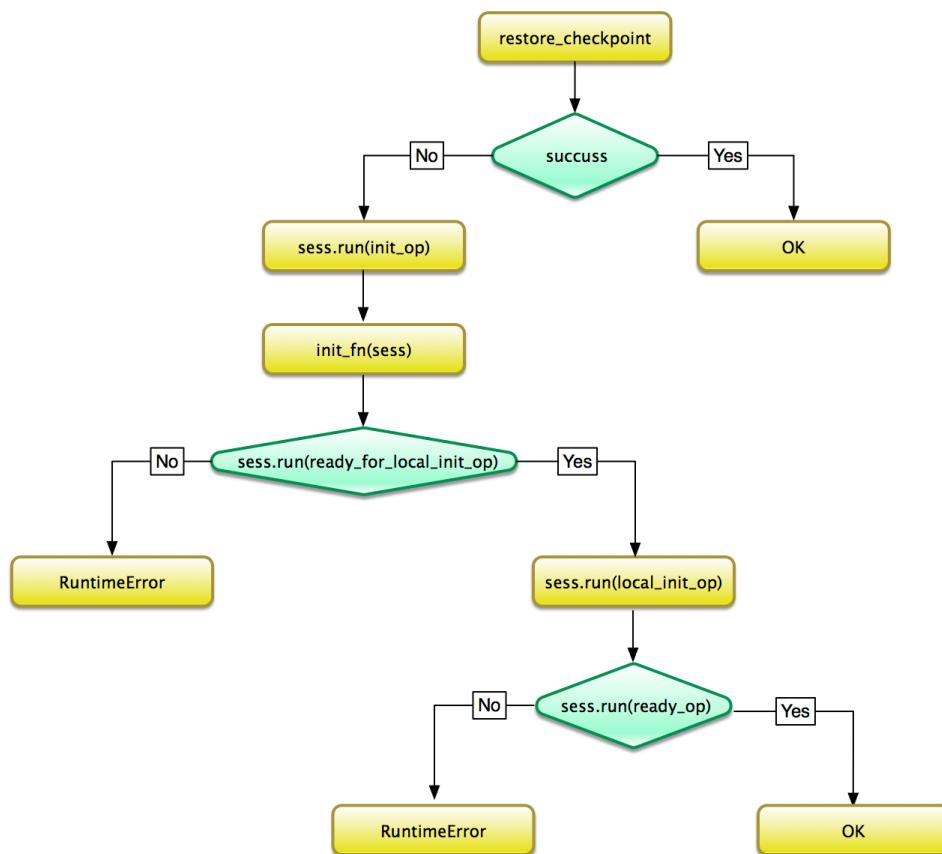


图 14-7 模型初始化算法

```
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
```

14.3.7 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
```

14.4 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。

14.4.1 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。

14.4.2 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
```

14.5 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。

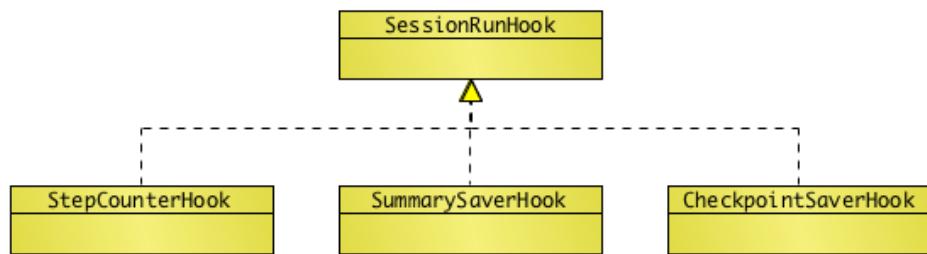


图 14-8 `SessionRunHook`

Part VI

附录



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。



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。

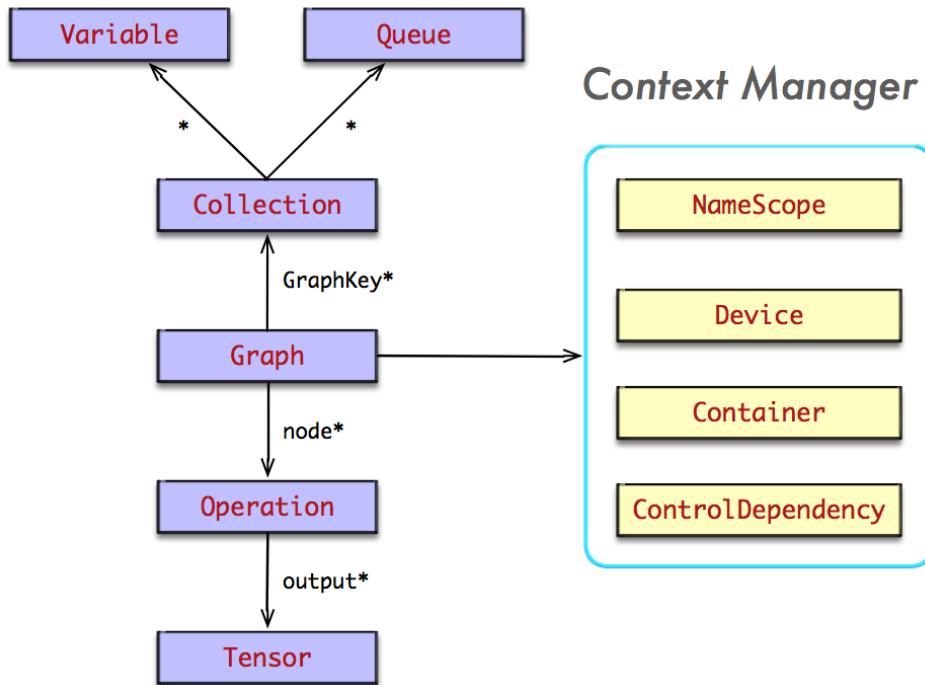


图 A-1 领域对象：Graph

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。

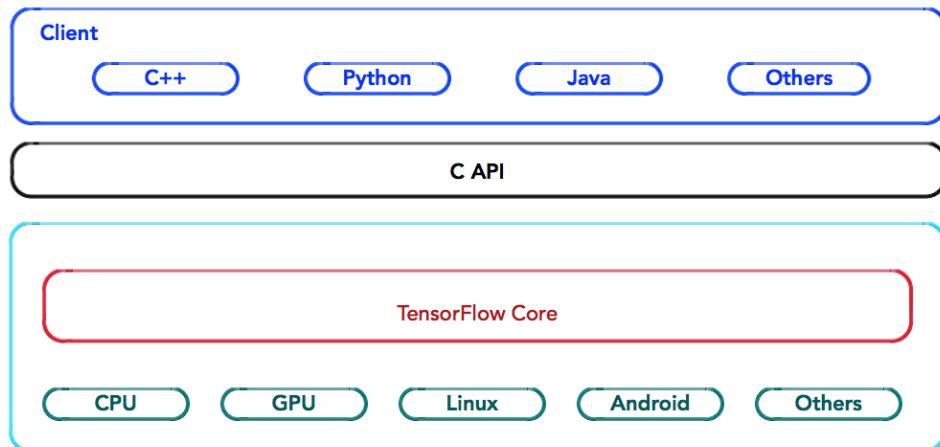


图 A-2 TensorFlow 系统架构

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
        array =
            new ArrayBuffer[elem.type](newSize)
        array += elem
    }
    else
        array += elem
    this
}
```

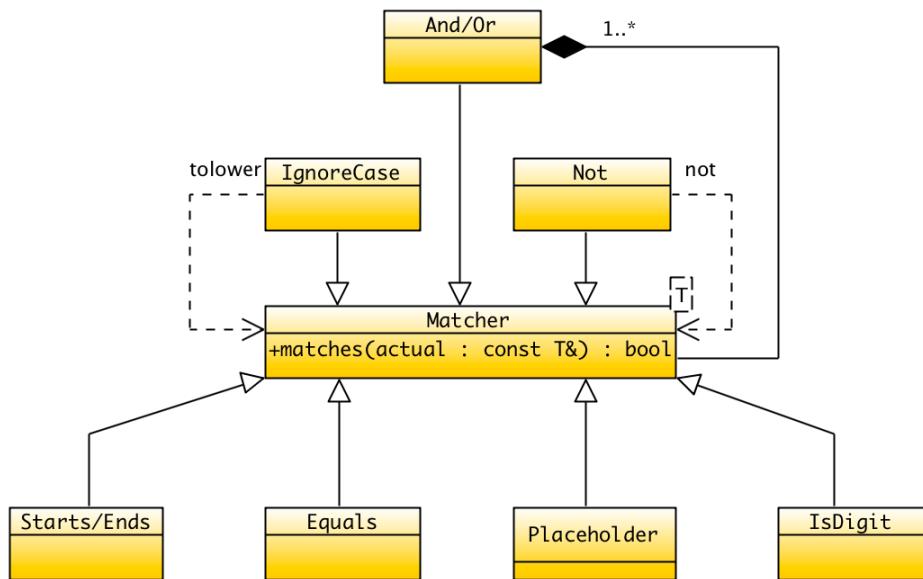


图 A-3 组合式设计

```

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。

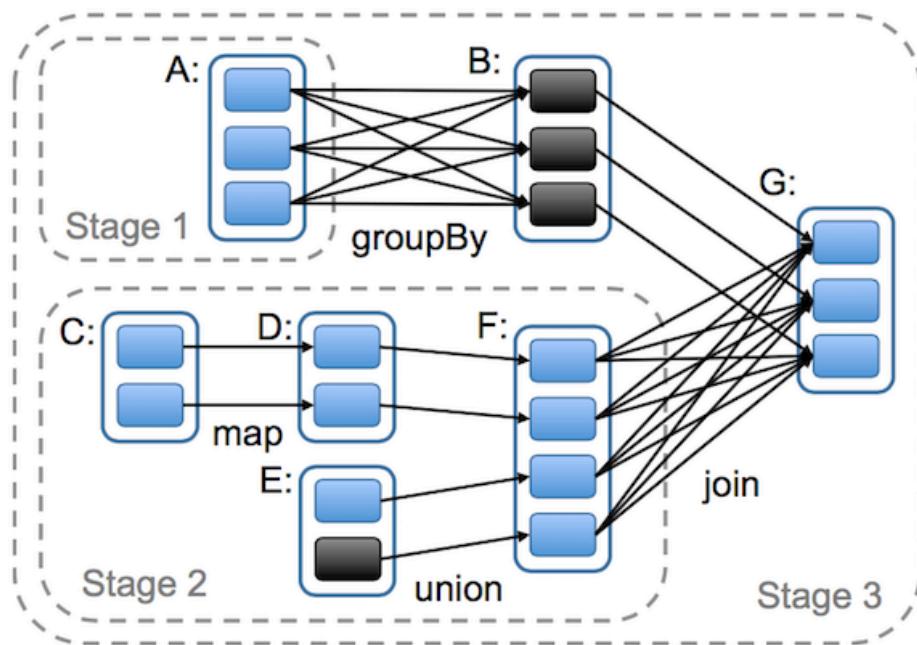


图 A-4 Spark: Stage 划分算法

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。



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。

B.1.1 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”。

B.1.2 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。

B.1.3 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。

B.1.4 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

B.2.1 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。

B.2.2 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。

B.2.3 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。

B.2.4 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。

B.2.5 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。

B.2.6 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.