TensorFlow 内核剖析

TensorFlow Internals

刘光聪

This page is intentionally left blank.

前言

本书定位

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

面向的读者

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

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

阅读方式

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

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

版本说明

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

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

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

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，绝对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iv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

This page is intentionally left blank.

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。

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 前世：DistBelief

DistBelief 是一个用于训练大规模神经网络的的分布式系统，是 Google 第一代分布式机器学习框架。自 2011 年以来，在 Google 内部大量使用 DistBelief 训练大规模的神经网络。

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2页）所示，在分布式训练过程中，各个模型备份 (Model Relica) 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。

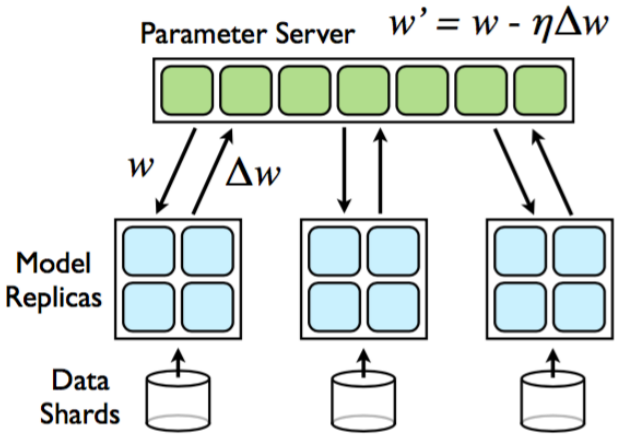

图 1-1 DistBelief: Parameter Server 架构

缺陷与不足

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 今生：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3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。

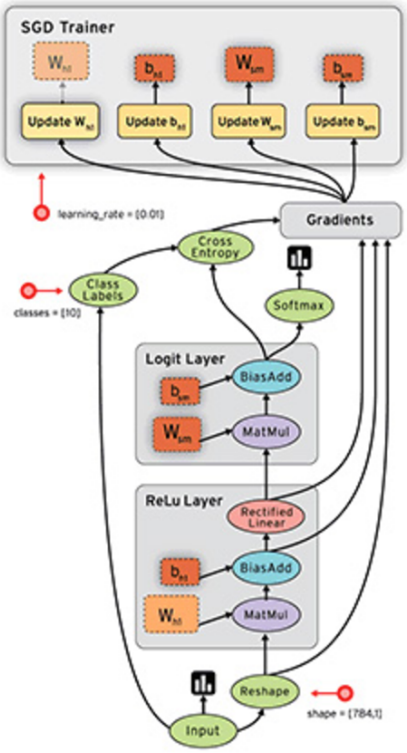

图 1-2 TensorFlow 数据流图

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4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。

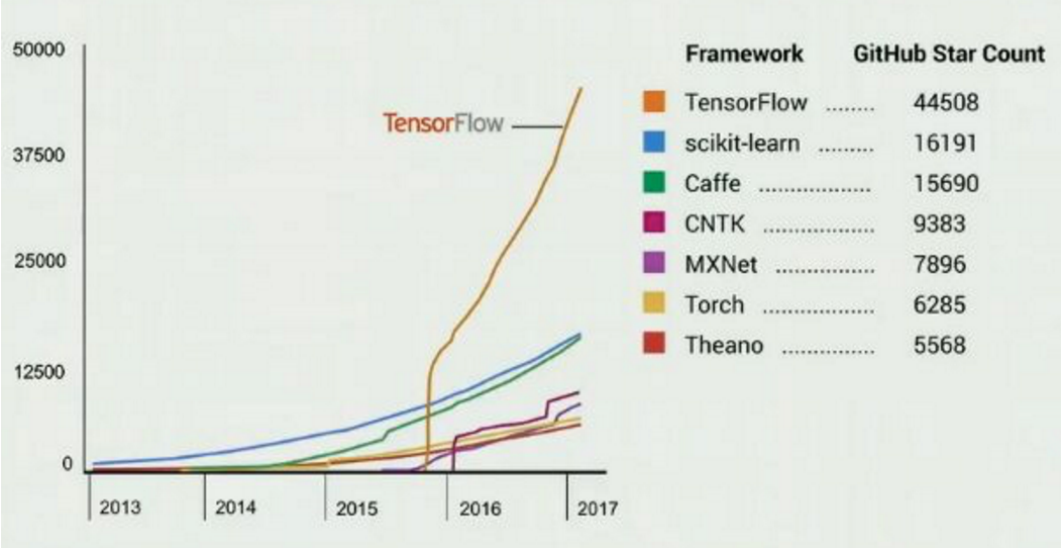

图 1-3 TensorFlow 社区活跃度

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5页）所示，展示了 TensorFlow 几个重要特性的发布时间。

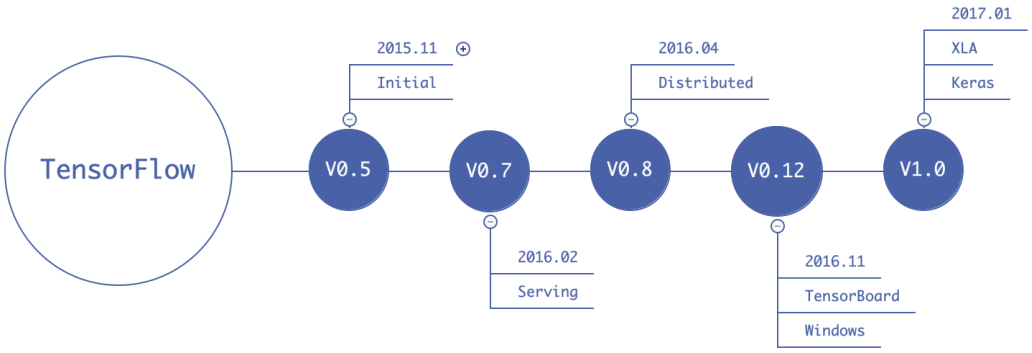

图 1-4 TensorFlow 重要里程碑

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 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5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。

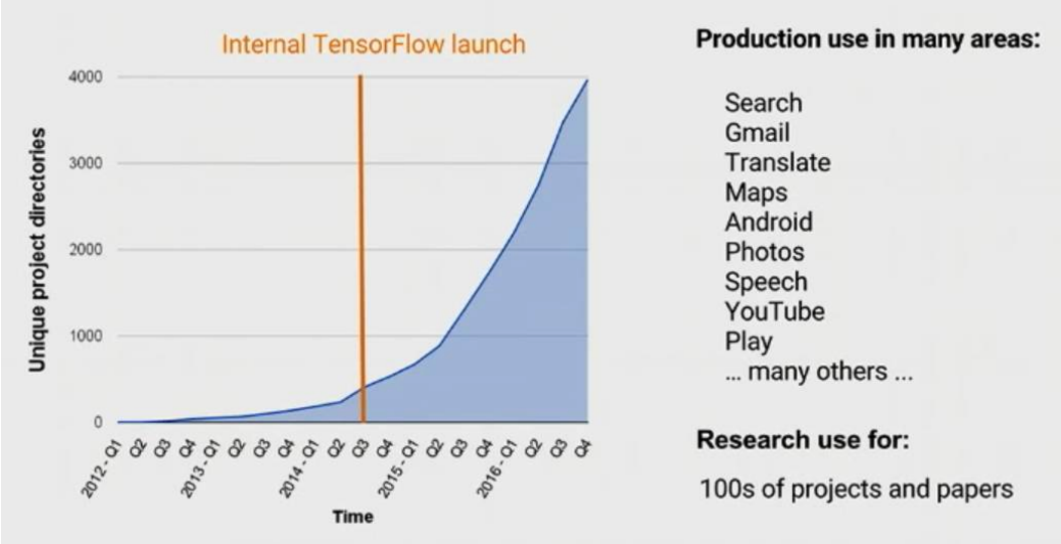

图 1-5 TensorFlow 在 Google 内部使用情况

This page is intentionally left blank.

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

- Martin Flower

2 编程环境

为了实现 TensorFlow 的快速入门，本章将介绍 TensorFlow 的编程环境，包括代码结构，工程构建，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 TensorFlow 源码

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 之需要前先安装 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 \
  https://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

图 2-1 TensorFlow 技术栈

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

- Martin Flower

3 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，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，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。

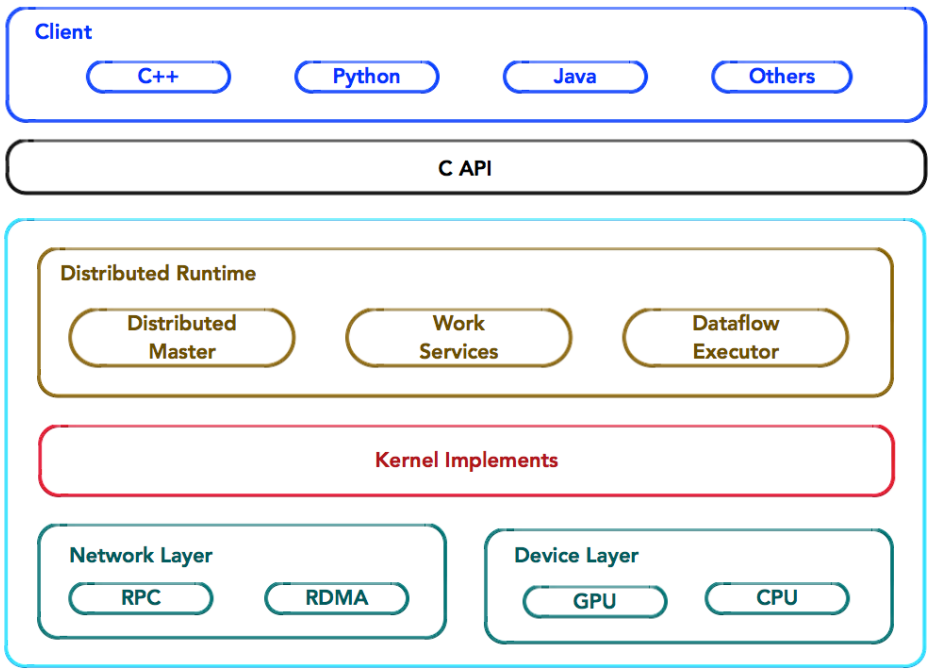

图 3-1 TensorFlow 系统架构

如图3-1（第13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。

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 支持最佳；并且，对其他编程语言的编程接口的支持日益完善。

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 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，最终计算得到所依赖的「最小子图」，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 启动相应的 Graph Partition 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，它们自行通信，独立交换数据即可，直至计算完成。

3.1.3 Worker

对于每以个任务，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. 调度 OP 的 Kernel 实现，执行本地子图；
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 Cliet 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 启动所有当前设备的 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 Work；或者接受来自其他 Worker 发送给它运算的结果，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。

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 的设备约束贵方，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。其中，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，其扩展性表现相当优越。

3.2 图控制

随后，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16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。

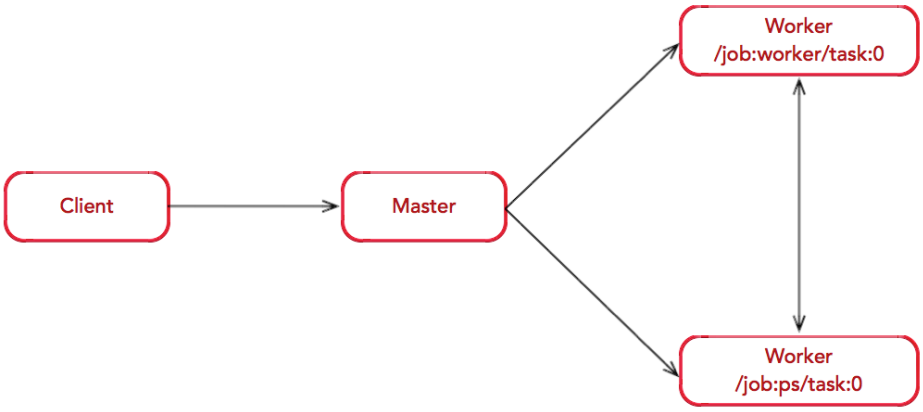

图 3-2 TensorFlow 集群：1 PS + 1 Worker

3.2.2 图构造

如图3.2.2（第16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。

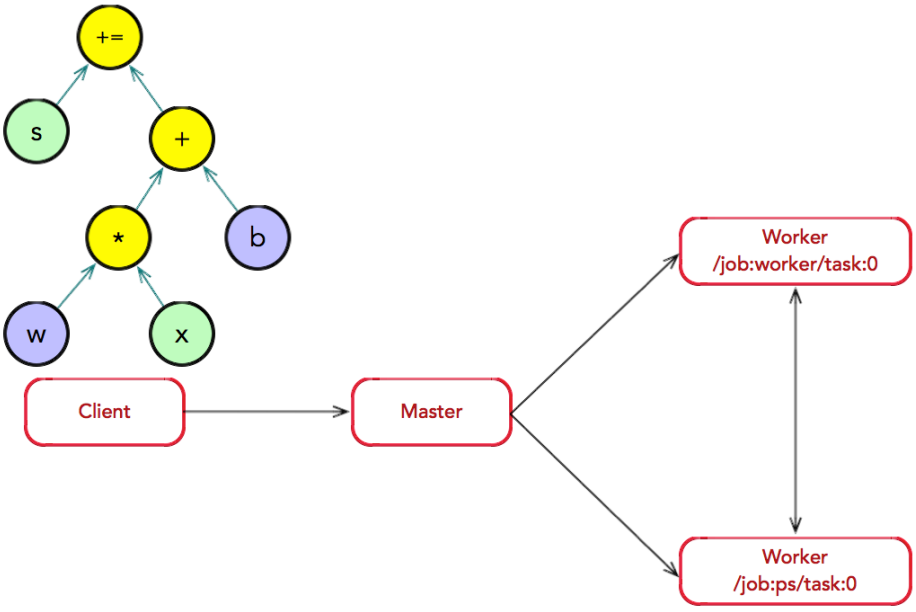

图 3-3 图构造

3.2.3 图执行

如图3.2.3（第17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 `Sess.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，执行优化后的计算子图。

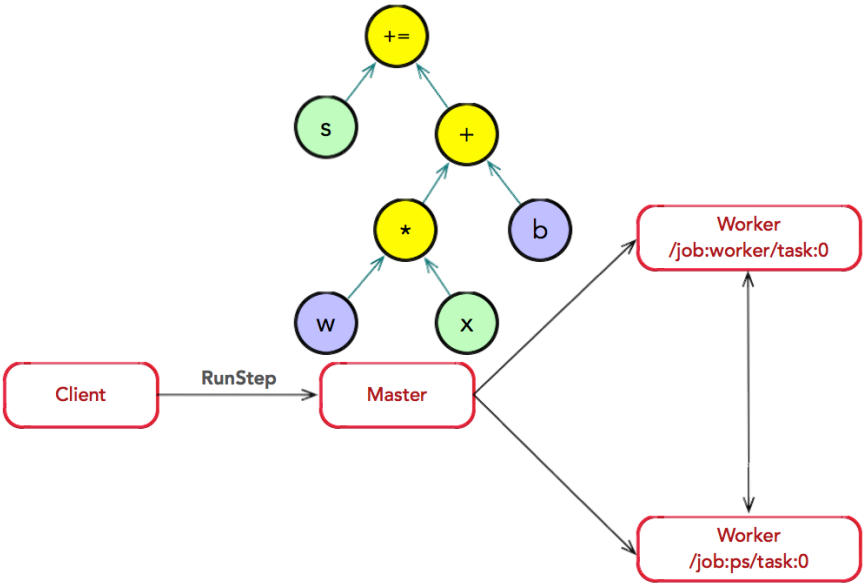

图 3-4 图执行

图分裂

如图3.2.3（第17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。

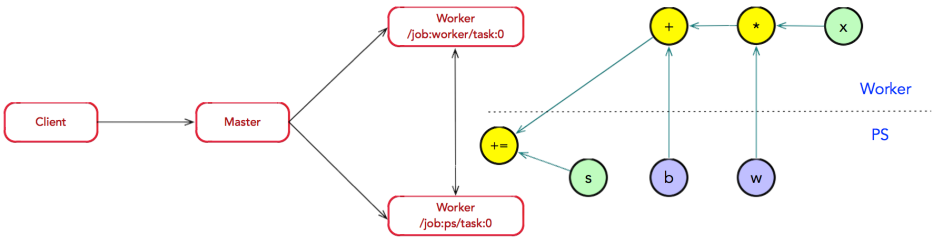

图 3-5 图分裂：按任务划分

子图注册

如图3.2.3（第18页）所示。在图分离过程中，如果计算图的边跨越任务节点，Master 将该边进行分裂，在两个任务之间插入 Send 和 Recv 节点，实现数据的传递。

其中，Send 和 Recv 节点也是 OP，这是两个特殊的 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 Graph Partition 注册给相应的任务中，并由相应的 Worker 负责执行。

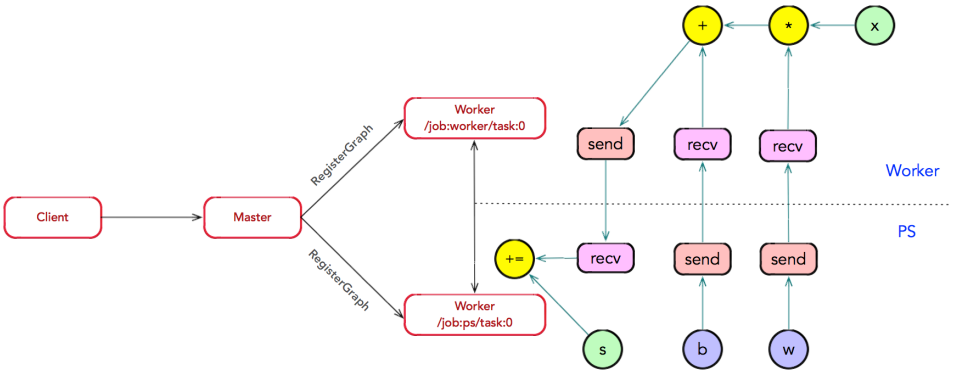

图 3-6 子图注册：插入 Send 和 Recv 节点

子图运算

如图3.2.3（第18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 之间通过调用 `RecvTensor` 接口，完成数据的交互。

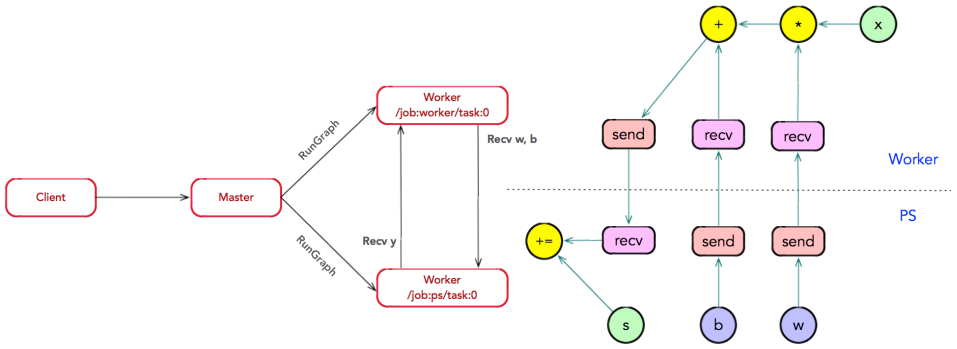

图 3-7 子图执行

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

- Martin Flower

4 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。

4.1 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。

4.1.1 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4-1（第20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。

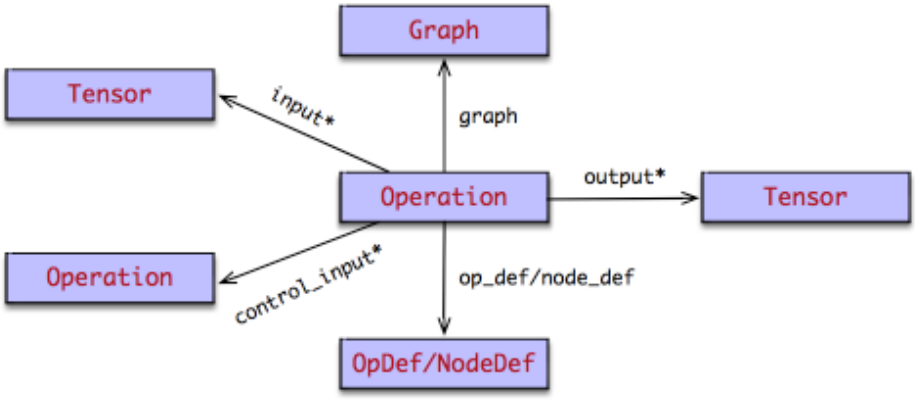

图 4-1 领域对象：Operation

4.1.2 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4-2（第20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。

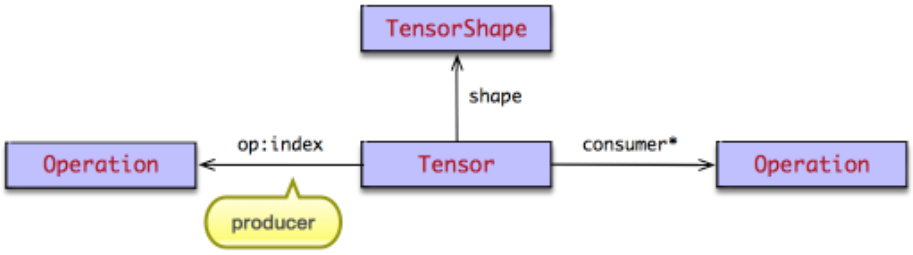

图 4-2 领域对象：Tensor

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4-3（第21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。

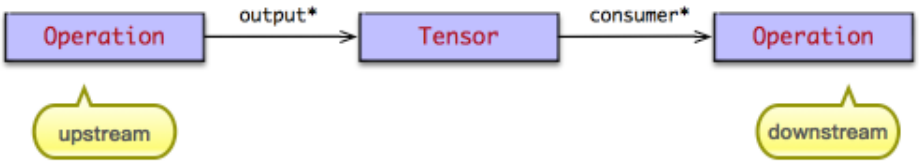

图 4-3 Tensor: 生产者-消费者关系

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)
```

4.1.3 Graph

如图4-4（第22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。

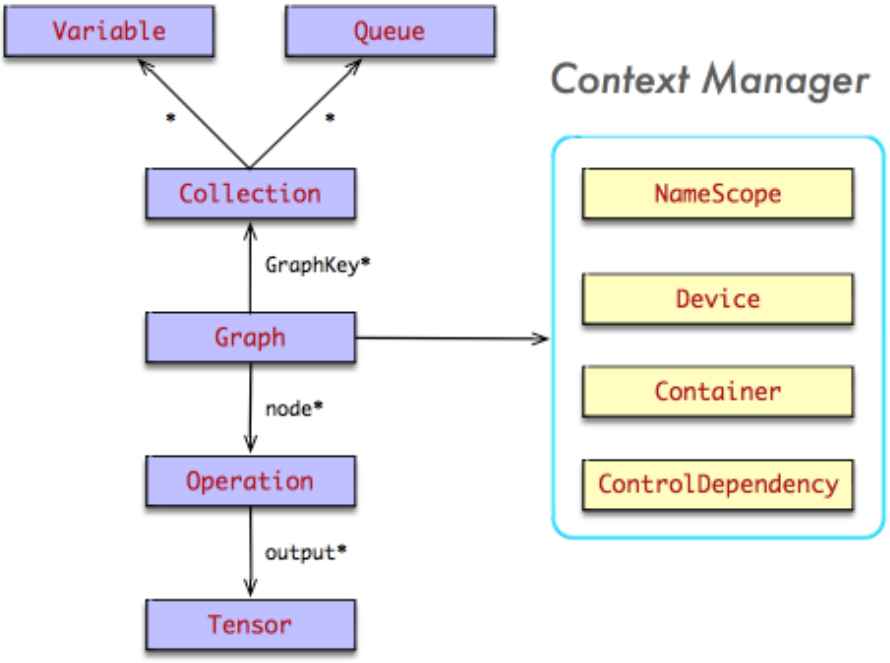

图 4-4 领域对象：Graph

4.1.4 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，对于某中类型的 OpDef 仓库，_InitOpDefLibrary 模块首次导入时，扫描 op_list_ascii 表示的所有 OP，并将其转换为 Protobuf 格式的 OpList 实例，最终将其注册到 OpDefLibrary 实例之中。

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

```
_op_def_lib = _InitOpDefLibrary()

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

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


工厂方法

如图4-5（第23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。

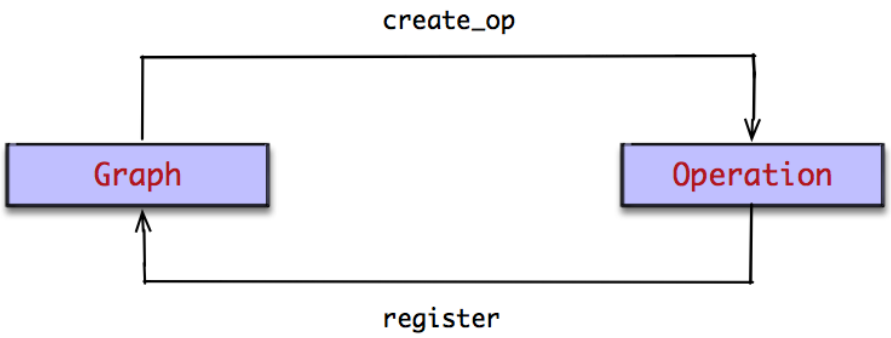

图 4-5 Graph: OP 工厂 + OP 仓库

OP 构造器

如图4-6(第24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。

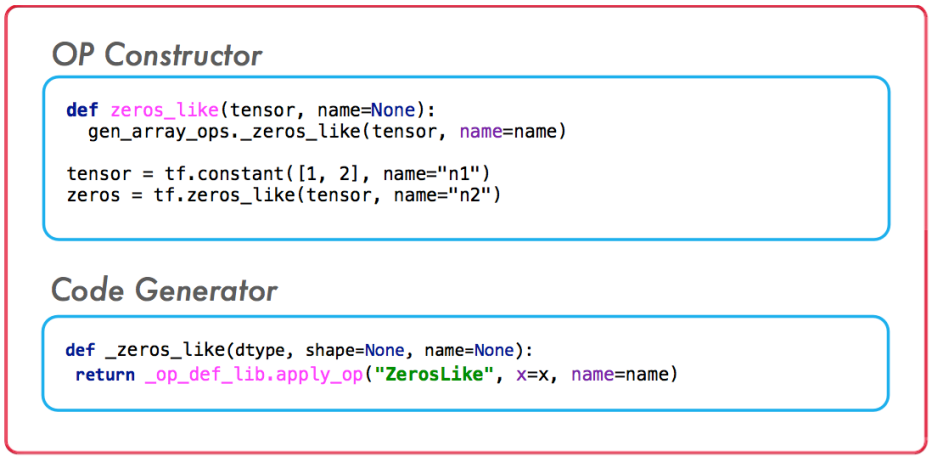

图 4-6 OP 构造器与代码生成器

构造 OpDef 与 NodeDef

然后，如图4-7（第25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。

4.2 后端 C++

在 C++ 后端，计算图是 TensorFlow 领域模型的核心。

4.2.1 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。

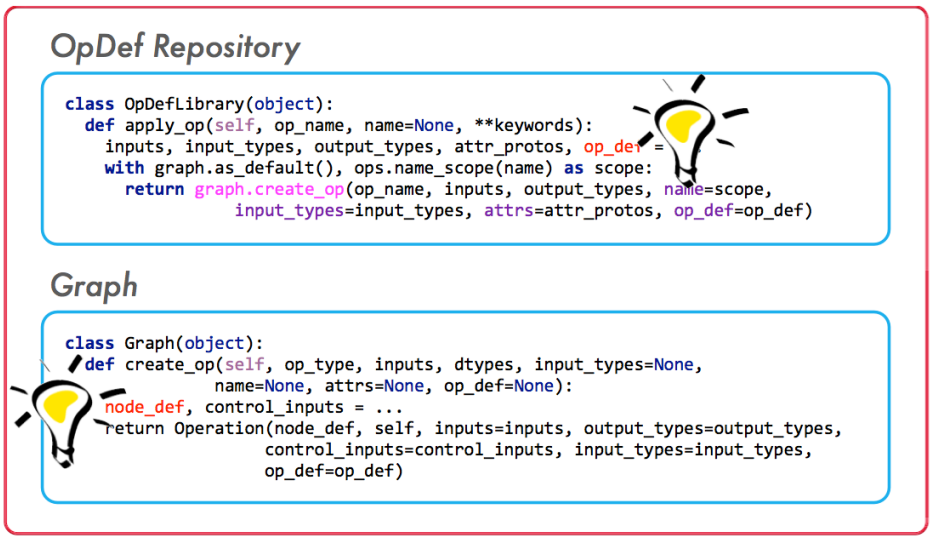

图 4-7 创建 Operation 实例：创建 OpDef, NodeDef 实例

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。

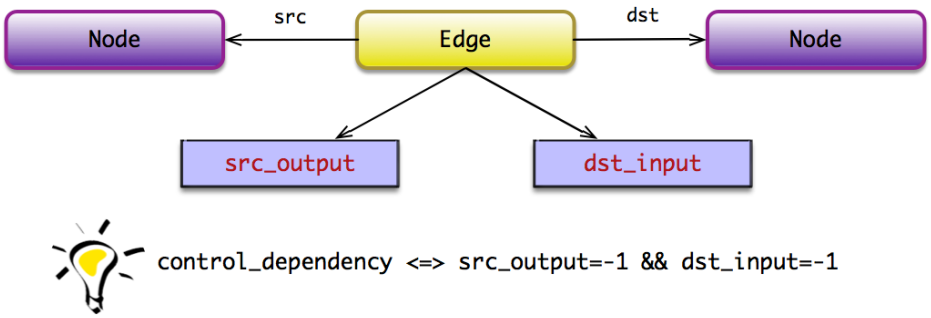

图 4-8 领域对象：Edge

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。

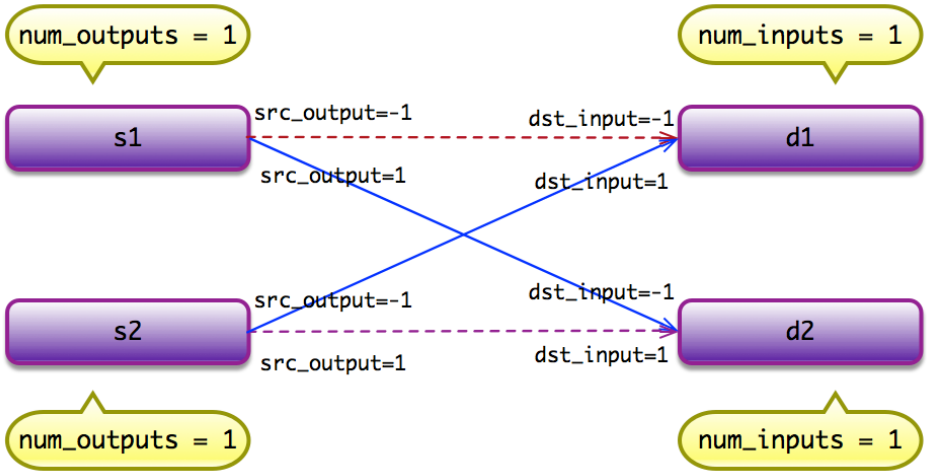

图 4-9 边例子

```
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。

4.2.2 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。

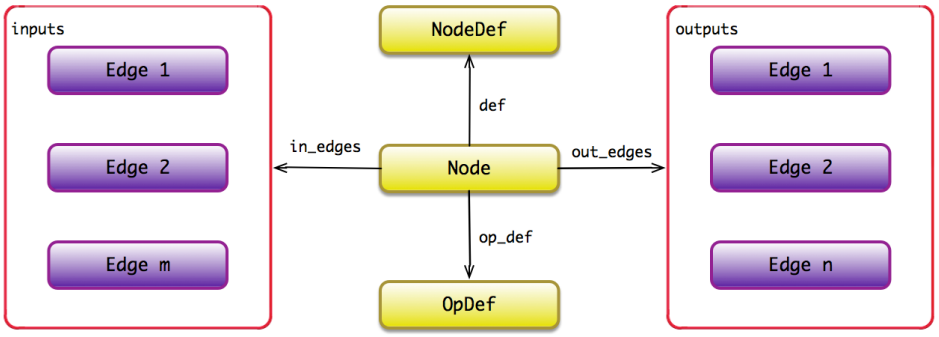

图 4-10 领域对象：Node

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();
}
```

4.2.3 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 4-11 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。

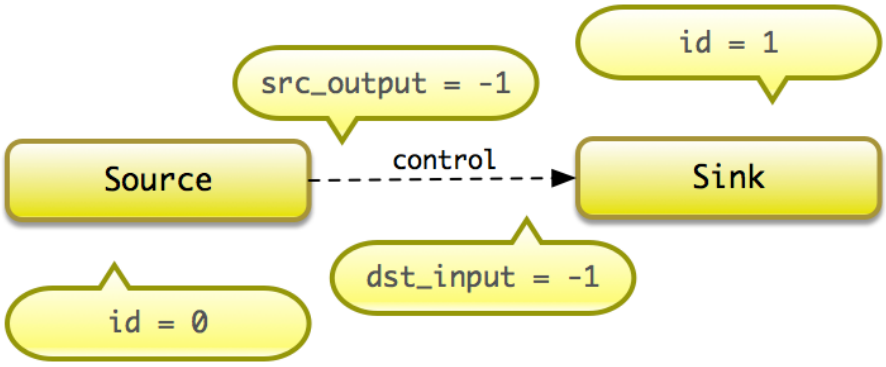

图 4-12 空图

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。

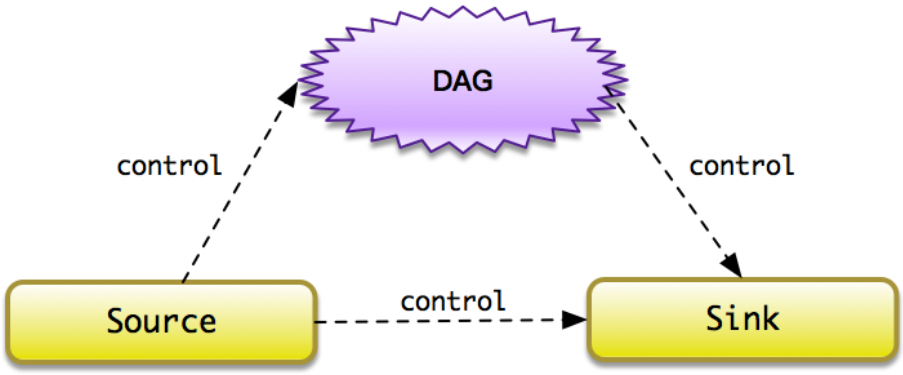

图 4-13 非空图

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);
}
```

4.2.4 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。

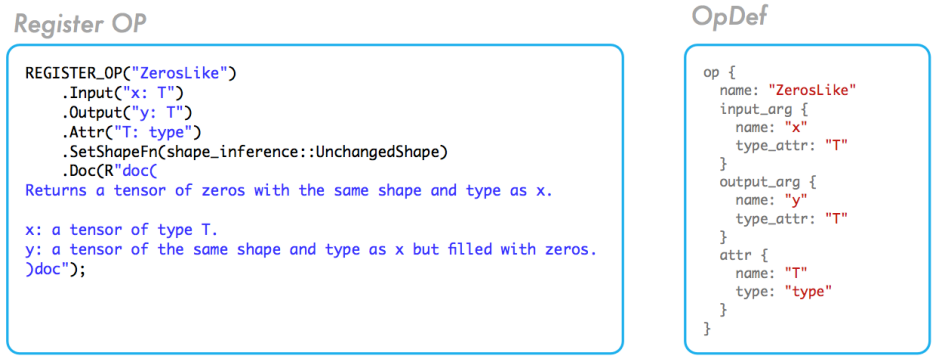

图 4-14 OpDef 注册：使用 REGISTER_OP

4.3 图传递

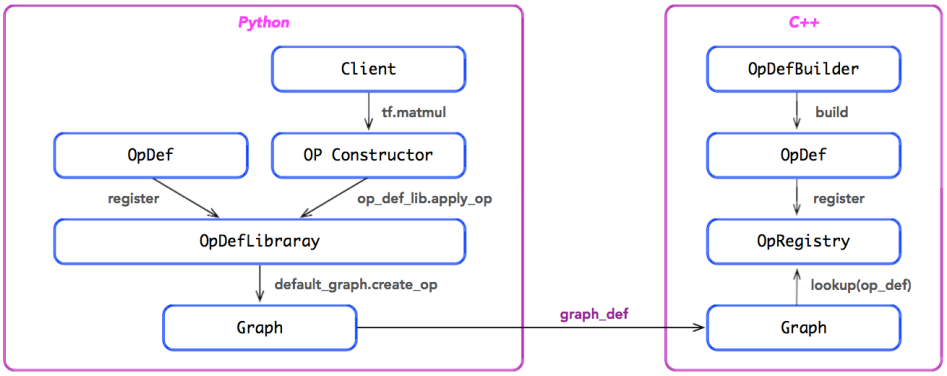

图 4-15 图的序列化与反序列化

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

- Martin Flower

5 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。

5.1 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5-1（第32页）所示。

事实上，正如图5-2（第32页）所示，实际的计算图要复杂得多，让我们从头说起。

5.2 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。

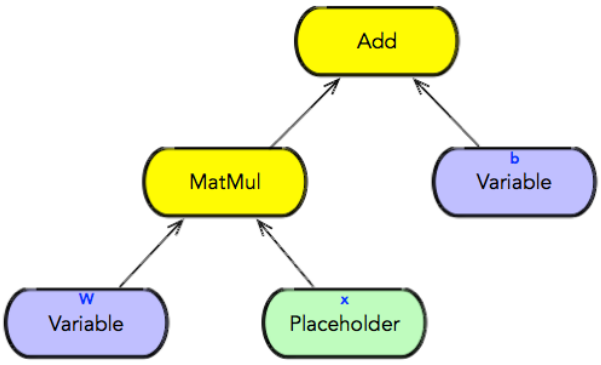

图 5-1 计算图：线性加权和

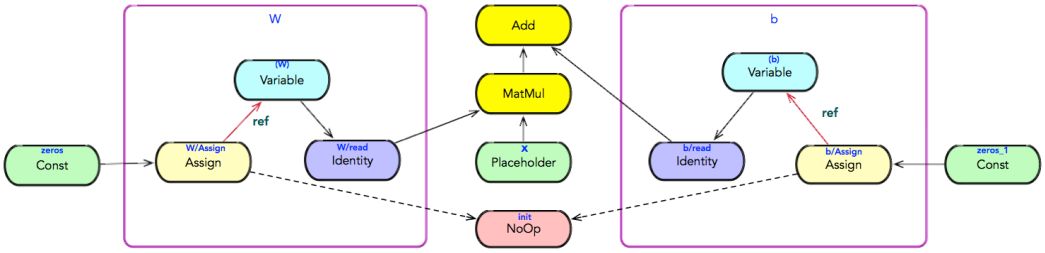

图 5-2 计算图：线性加权和

5.2.1 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)
```

5.2.2 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：

5.2.3 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。

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

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

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

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

5.2.4 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。

5.2.5 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。

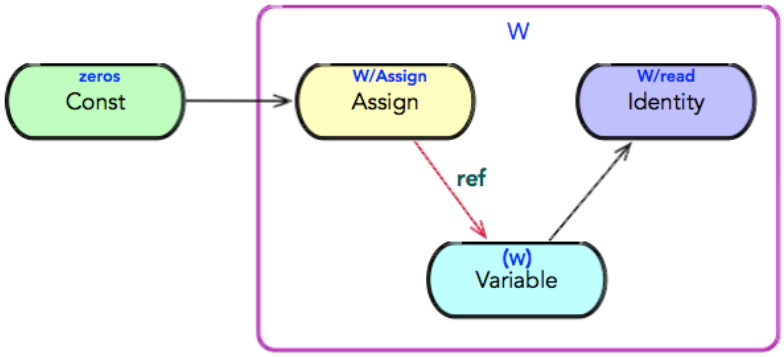

图 5-3 变量子图

5.2.6 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。

5.2.7 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)。

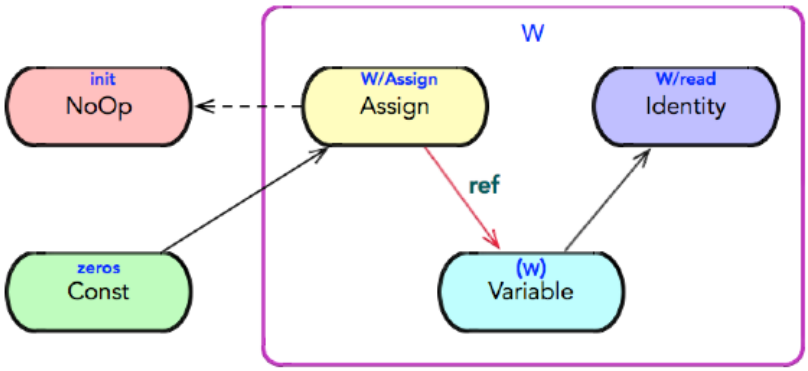

图 5-4 初始化 OP

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"
      }
    }
  }
}
```


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

同样地，可以通过调用 `tf.global_variables_initializer()` 将变量的所有初始化器进

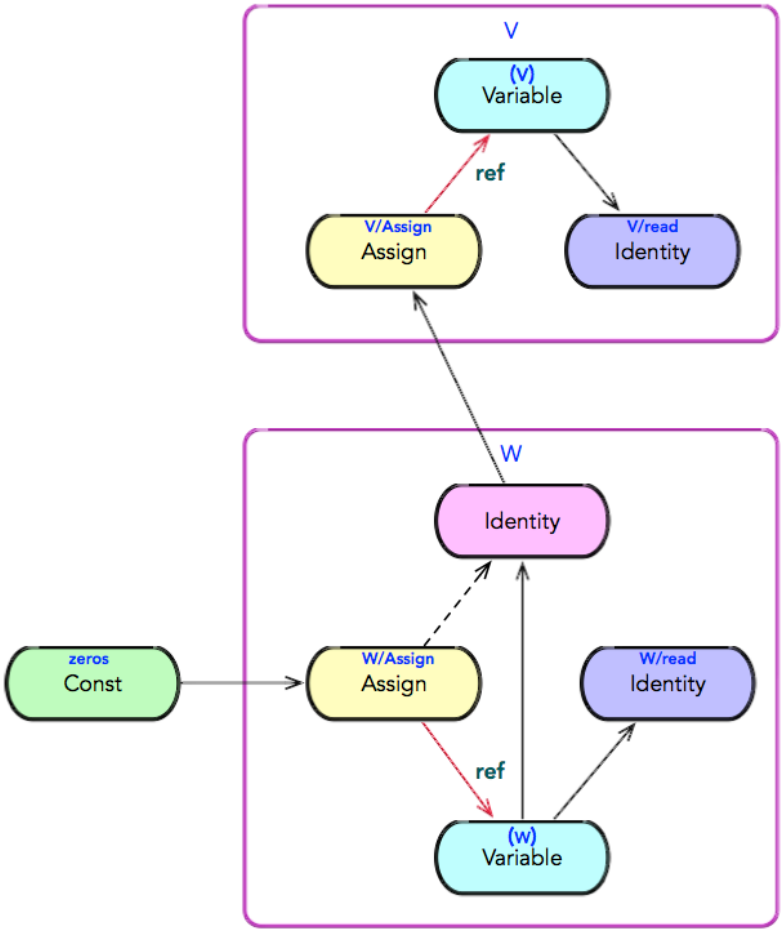

图 5-5 初始化依赖

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。

5.2.9 初始化器列表

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

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

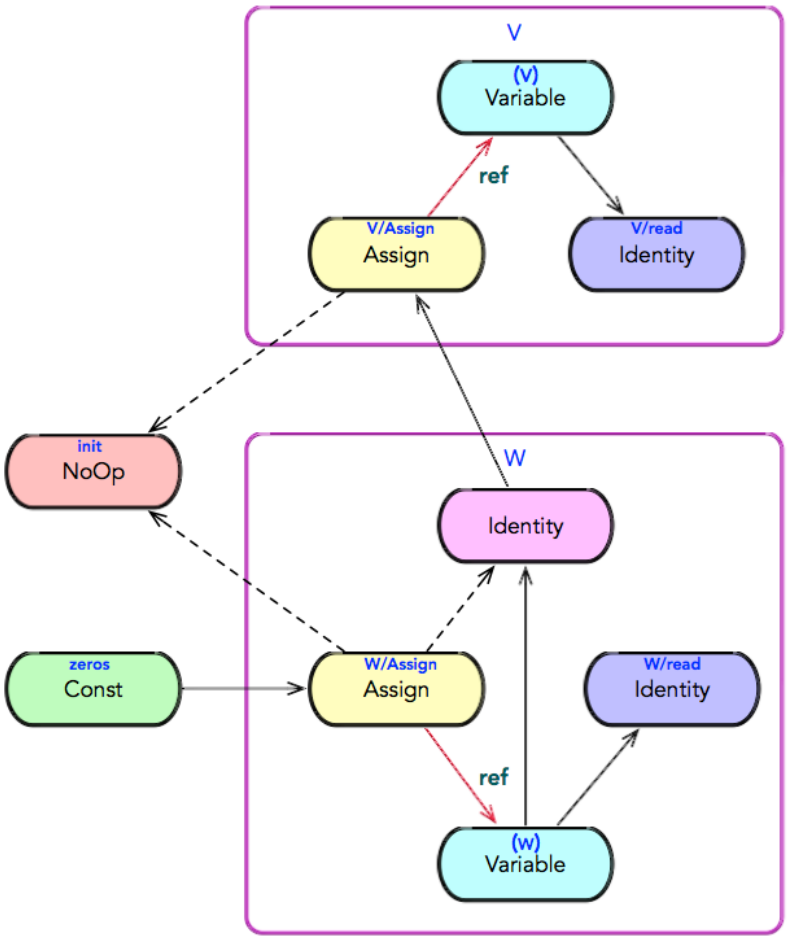

图 5-6 初始化 OP

例如，全局变量列表的初始化器列表可以如下构造。

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

5.3 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。

5.3.1 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)
```

5.3.2 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。

5.3.3 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)
```

5.3.4 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)
```

5.4 源码分析：构造变量

为了简化代码实现，此处对 Variable 做了简单的重构。

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

构造 Variable 实例，基本包括如下几个步骤：

构造初始值

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

5.4.1 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)
```

5.4.2 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
```

5.4.3 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")
```

5.4.4 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

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。

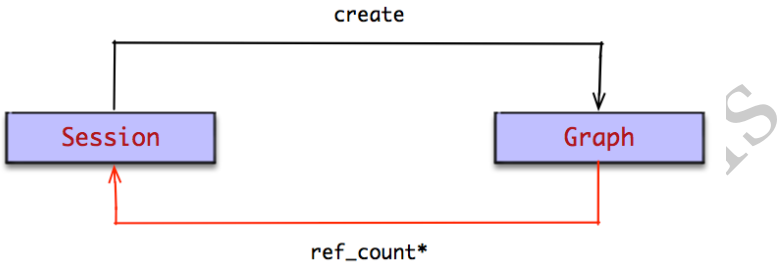

图 7-1 优化技术：会话实例的引用计数器

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::Sesssion, TF_Graph>，它们之间是一对一的关系。其中，tensorflow::Sesssion 是 C++ 客户端侧的会话实例。

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

```
  int last_num_graph_nodes;
};
```

创建会话

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

销毁会话

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


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。

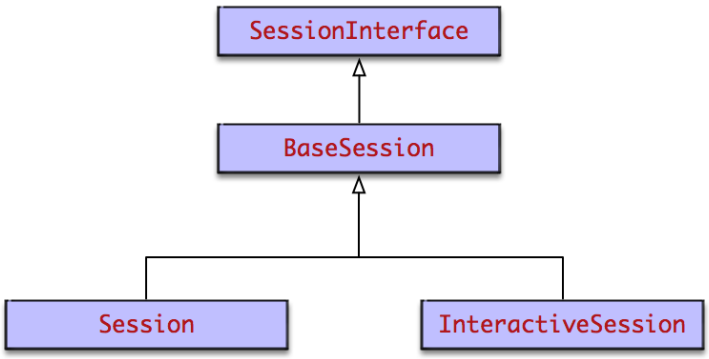

图 7-2 Session：类层次结构

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)
```

This page is intentionally left blank.

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

- Martin Flower

8 Saver

8.1 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。

8.1.1 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, 'ckp')
```

随后，可以根据断点文件存储的位置恢复模型。

```
with tf.Session() as sess:
  saver = tf.import_meta_graph('ckp.meta')
  saver.restore(sess, 'ckp')
```

8.1.2 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::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。

8.1.3 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。

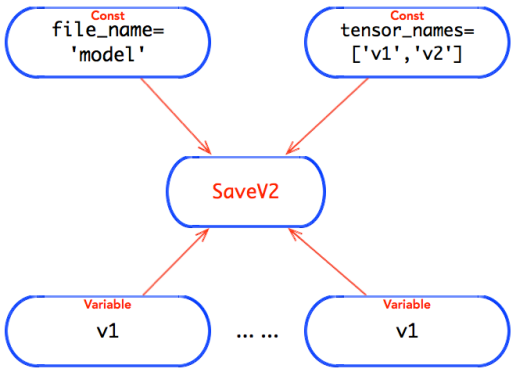

图 8-1 Saver：持久化模型

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。

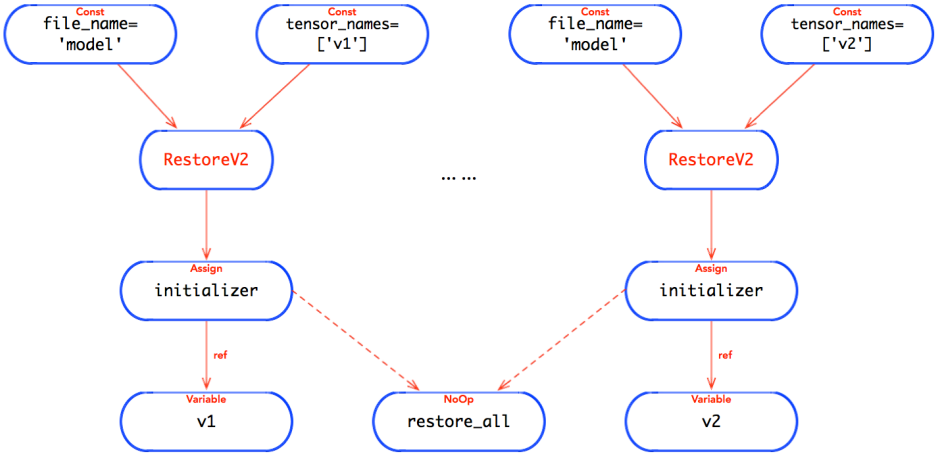

图 8-2 Saver：恢复模型

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

- Martin Flower

9

MonitoredSession

训练一个简单的模型，可以通过运行 `train_op` 数次直至模型收敛，最终将训练参数实施 Checkpoint，持久化训练模型。对于小规模的学习模型，这个过程至多需要花费数小时的时间。

但是，对于大规模的学习模型，需要花费数天时间；而且可能需要使用多份复本 (replica)，此时需要更加健壮的训练过程支持模型的训练。因此，需要解决三个基本问题：

1. 当训练过程异常关闭，或程序崩溃，能够合理地处理异常；
2. 当异常关闭，或程序崩溃之后，能够恢复训练过程；
3. 能够通过 TensorBoard 监控整个训练过程。

当训练被异常关闭或程序崩溃之后，为了能够恢复训练过程，必须周期性实施 Checkpoint。当训练过程重启后，可以通过寻找最近一次的 Checkpoint 文件，恢复训练过程。

为了能够使用 TensorBoard 监控训练过程，可以通过周期性运行一些 Summary 的 OP，并将结果追加到事件文件中。TensorBoard 能够监控和解析事件文件的数据，可视化整个训练过程，包括展示计算图的结构。

9.1 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`。

9.1.1 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)
```


9.1.2 使用工厂

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

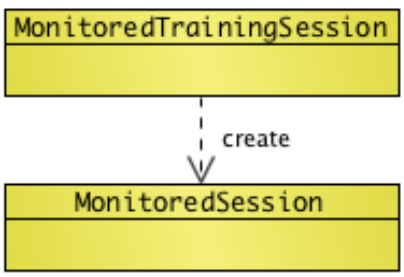

图 9-1 MonitoredTrainingSession：工厂方法

```
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)
```

9.1.3 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。

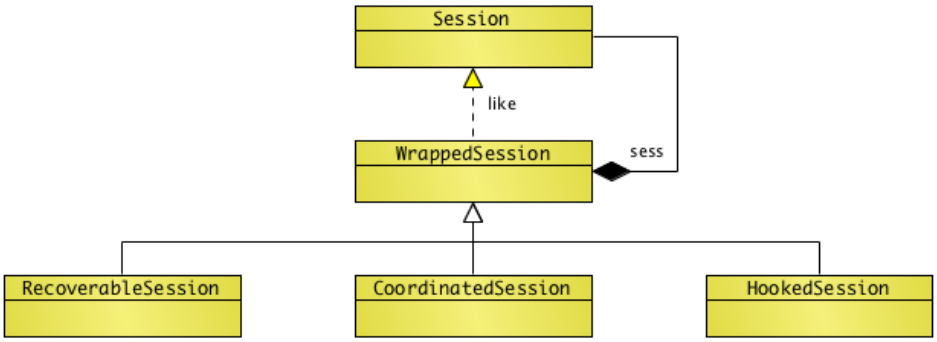

图 9-2 MonitoredSession：装饰器

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)))))
```

9.2 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。

9.2.1 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 Checkpoing 恢复模型，或运行 init_op，完成模型的初始化；然后，启动所有 QueueRunner 实例；
2. WorkerSessionCreator：调用 SessionManager.wait_for_session，等待 Chief 完成模型的初始化。

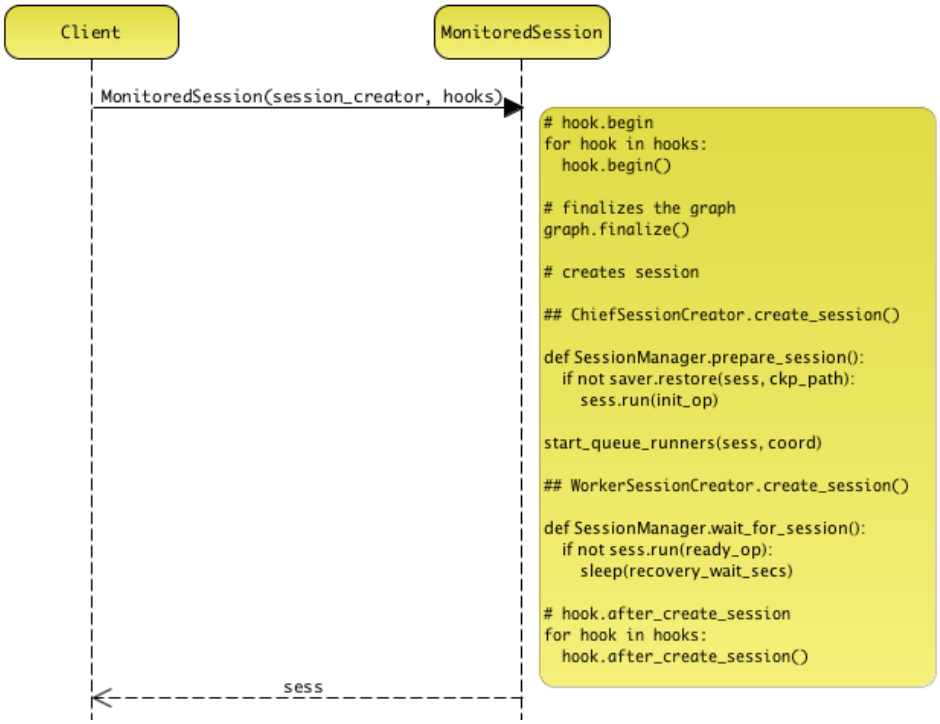

图 9-3 MonitoredSession：初始化

9.2.2 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。

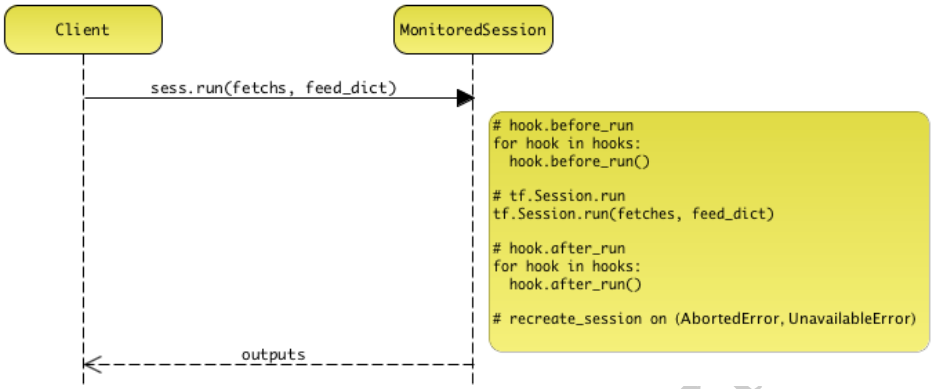

图 9-4 MonitoredSession：执行

9.2.3 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。

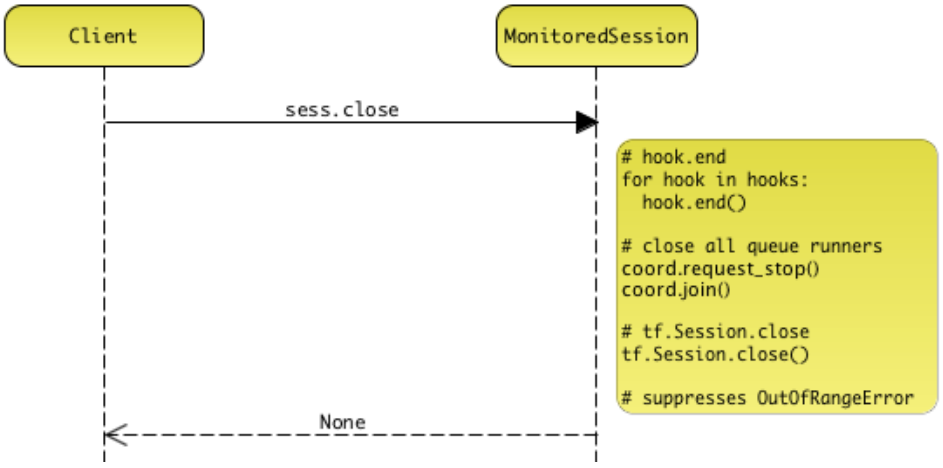

图 9-5 MonitoredSession：关闭

9.3 模型初始化

MonitoredSession 在初始化时，使用 SessionCreator 完成会话的创建和模型的初始化。

一般地，在分布式环境下，存在两中类型的 Worker:

1. Chief: 负责模型的初始化；
2. Non-Chief: 等待 Chief 完成模型的初始化。

两者之间，通过一个简单的协调协议共同完成模型的初始化。

9.3.1 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)
```

9.3.2 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。

9.3.3 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。

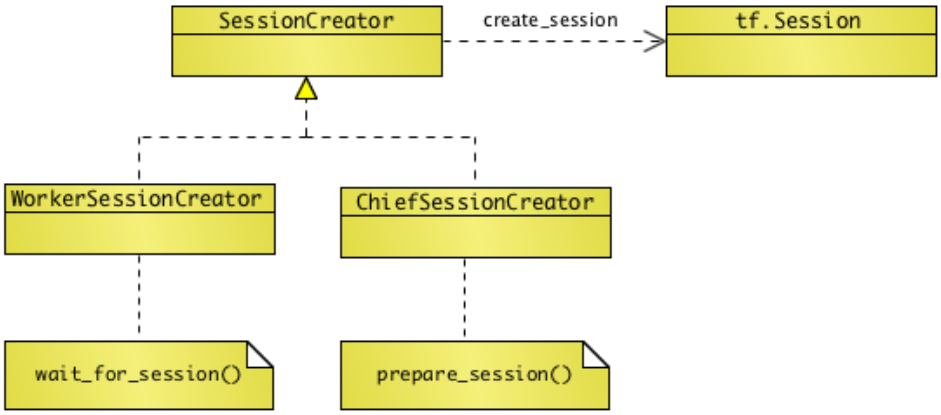

图 9-6 SessionManager

9.3.4 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。

9.3.5 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。

9.3.6 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。

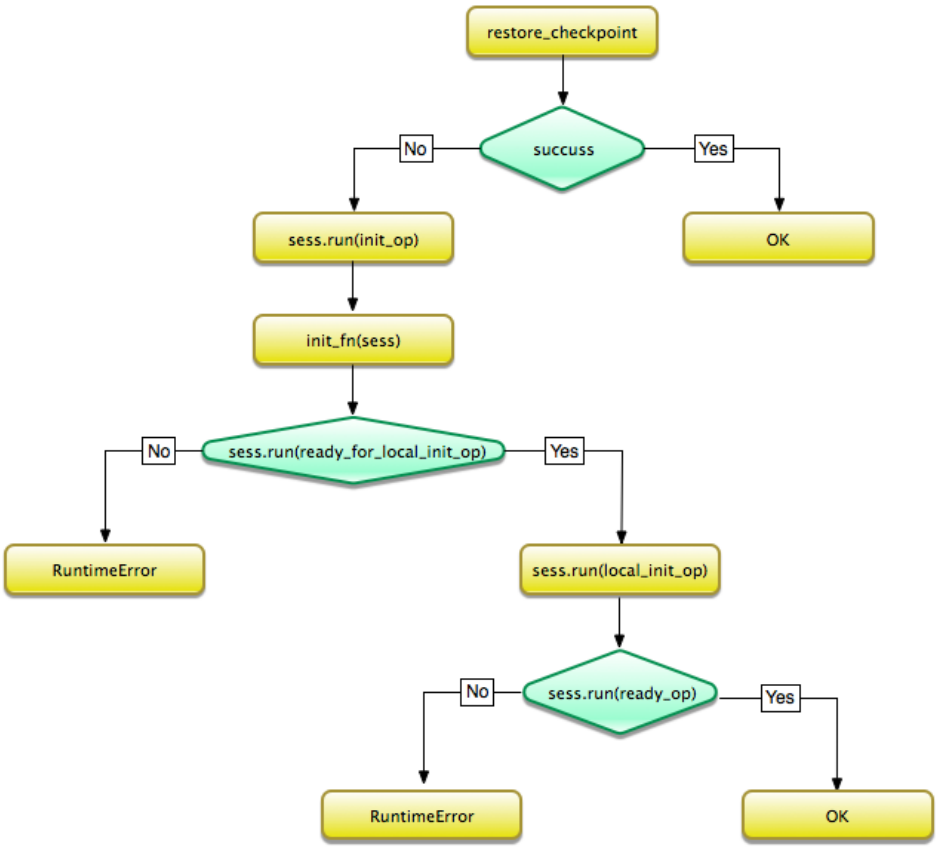
restore_checkpoint
succuss
No
Yes
sess.run(init_op)
OK
init_fn(sess)
sess.run(ready_for_local_init_op)
No
Yes
RuntimeError
sess.run(local_init_op)
sess.run(ready_op)
No
Yes
RuntimeError
OK

图 9-7 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
```

9.3.7 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
```

9.4 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。

9.4.1 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。

9.4.2 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
```

9.5 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。

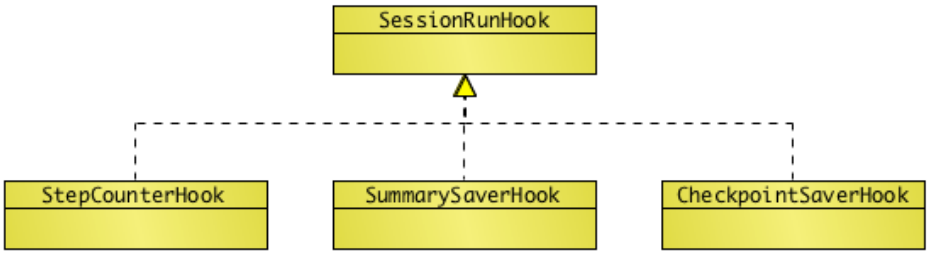

图 9-8 SessionRunHook

This page is intentionally left blank.