TensorFlow 代码阅读指南

郑乘帆 ACSA
2017.02.21

一、以 dbg 选项安装

为了方便在调试中对照源代码，需要从源码附加 dbg 选项进行编译并且安装。因此需要使用 source + pip + virtualenv 的方式即：使用 bazel 编译工具编译源代码，生成 pip 包，然后用 virtualenv 创建虚拟环境之后，把 pip 包装到虚拟环境中去。

首先创建 virtualenv 虚拟环境，并且用 souce 命令载入。

在源代码目录下：

```
# 配置选项
./configure

# 正常
$ bazel build -c opt //tensorflow/tools/pip_package:build_pip_package
# 调试选项
$ bazel build -c dbg //tensorflow/tools/pip_package:build_pip_package
# 加 GPU
$ bazel build -c opt --config=cuda //tensorflow/tools/pip_package:build_pip_package
```

执行 build 指令之后，TensorFlow 所需的软件库都已经编译完成，调用 build_pip_package 命令将其打包成可用于 pip 环境的安装包，命令最后是安装包的存放位置：

```
$ bazel-bin/tensorflow/tools/pip_package/build_pip_package /tmp/tf
```

在 virtualenv 环境中使用 pip install 命令将安装包装上即可。

二、源码目录分析

源代码中，主要的内容在"tensorflow"这一目录下，上一个版本统计的 c/c++部分代码量为 48 万行，1.0 版"tensorflow"目录下的 c/c++部分代码量达到了 63 万行，加上 python 部分代码量达到了 111 万行（其中有很大一部分为注释）。

目录	内容
/c	面向用户的 C++编程 API
/cc	
/contrib	额外的 Python 库
/core	TensorFlow 运行时环境的所有源码

/core/common_runtime	单节点运行时环境
/core/distributed_runtime	分布式运行时环境
/core/……	
/examples	
/g3doc	官方提供的文档
/go	面向用户的 Go 语言编程 API
/models	预设的一些模型
/python	面向用户的 Python 编程 API
/stream_executor	流运算器，与 GPU 相关
/tensorboard	TensorFlow 提供的一个训练可视化工具
/tools	其他的一些工具
/user_ops	

三、pip 包目录分析

使用 pip + virtualenv 环境进行安装后，假定 virtualenv 的目录为 tf-master，则 TensorFlow 的所有文件都在 `tf-master/lib/python/site-packages/tensorflow` 目录中，这是一个标准的 Python 包目录。

Python 包目录中包含的文件：

目录/文件	说明
/contrib	TensorFlow 中用到的一些子模块
/core	运行时核心部分
/example	运行示例
/include	包含了一些头文件
/python	TensorFlow 运行时中的 python API
/python/pywrap_tensorflow.so	TensorFlow 的 C 和 C++ 部分源码编译之后都在这个运行库中，整个库文件的大小有 1.7G
/python/pywrap_tensorflow.py&pyc	Python 通过 SWIG 来调用 C 和 C++ 的运行库，这两个是 SWIG 生成的接口
/python/client	
/python/client/session.py&pyc	
/tensorboard	用于训练可视化，启动一个网页服务器，可供在浏览器中打开
/tools	其他工具

四、TensorFlow 的 gdb 调试方法

目前常用的 TensorFlow API 还是 Python 版的，因此需要用到 gdb 的附加到进程选项载入 Python 进程，随后进入 C/C++部分的代码。

启动 Python 进程，通过 os 包中的命令获取当前的进程号。

```
$ python
>>> import tensorflow as tf
>>> import os
>>> os.getpid()
12345
```

得到当前 Python 进程的进程号之后即可在另一个窗口中通过 gdb 启动进程调试，以 TF_NewBuffer 函数的断点为例：

```
$ gdb -p 12345
(gdb) break TF_NewBuffer
(gdb) c
```

回到刚刚的 Python 窗口：

```
>>> a = tf.constant(1)
>>> sess = tf.Session()
>>> sess.run(a)
```

gbd 即断在了 TF_NewBuffer 这个函数上，

```
Breakpoint 1, TF_NewBuffer () at tensorflow/c/c_api.cc:280
280         TF_Buffer* TF_NewBuffer() { return new TF_Buffer{nullptr, 0, nullptr}; }
(gdb) bt
#0  TF_NewBuffer () at tensorflow/c/c_api.cc:280
#1  0x00007f68a1004ea5 in _wrap_TF_NewBuffer (args=0x7f69050ab050) at bazel-out/local-dbg/bin/tensorflow/python/pywrap_tensorflow.cc:5364
#2  0x00007f6904c02af0 in PyEval_EvalFrameEx () from /lib64/libpython2.7.so.1.0
#3  0x00007f6904c04e3d in PyEval_EvalCodeEx () from /lib64/libpython2.7.so.1.0
#4  0x00007f6904c0233c in PyEval_EvalFrameEx () from /lib64/libpython2.7.so.1.0
#5  0x00007f6904c04e3d in PyEval_EvalCodeEx () from /lib64/libpython2.7.so.1.0
#6  0x00007f6904c04f42 in PyEval_EvalCode () from /lib64/libpython2.7.so.1.0
#7  0x00007f6904c1e37f in run_mod () from /lib64/libpython2.7.so.1.0
#8  0x00007f6904c20430 in PyRun_InteractiveOneFlags () from /lib64/libpython2.7.so.1.0
#9  0x00007f6904c2061e in PyRun_InteractiveLoopFlags () from /lib64/libpython2.7.so.1.0
#10 0x00007f6904c20cae in PyRun_AnyFileExFlags () from /lib64/libpython2.7.so.1.0
#11 0x00007f6904c3191f in Py_Main () from /lib64/libpython2.7.so.1.0
#12 0x00007f6903e59b35 in __libc_start_main () from /lib64/libc.so.6
#13 0x000000000040071e in _start ()
```

可以看到 libpython 首先到 `pywrap_tensorflow` 里面调了 `_wrap_TF_NewBuffer` 这个函数，这部分是 SWIG 的封装，之后再是对 c_api.cc 中的 `TF_NewBuffer` 函数的调用。

采用类似的方法可以对代码中任何函数进行断点及调试。

五、单节点运行时

Session

Session 是 Python API 中用于管理和调度计算资源，并且控制启动 graph 计算过程的一个类。

Python 中 Session 类的继承关系：

```
class SessionInterface(object):
...
class BaseSession(SessionInterface):
...
class Session(BaseSession):
...
class InteractiveSession(BaseSession):
...
```

Session 和 InteractiveSession 是 TensorFlow 的 Python 代码中直接能用到的接口。

sess = tf.Session()

Session() 类创建时首先是调用自己的 `__init__` 构造方法，代码中用 super 函数调用了其父类 BaseSession 的 `__init__` 方法。

通过查看代码找到 C/C++部分的接口，即可使用 gdb 层层深入。

BaseSession.`__init__` 中创建 Session 的函数调用层次：

```
# session.py
4 BaseSession.__init__
    opts = tf_session.TF_NewSessionOptions(target=self._target, config=config)
    self._session = tf_session.TF_NewDeprecatedSession(opts, status)
# c_api.cc
3 TF_DeprecatedSession* TF_NewDeprecatedSession(const TF_SessionOptions* opt, TF_Status* status);
    status->status = NewSession(opt->options, &session);
    return new TF_DeprecatedSession({session});
# session.cc
```

```
2 Status NewSession(const SessionOptions& options, Session** out_session);
    Status s = SessionFactory::GetFactory(options, &factory);
    *out_session = factory->NewSession(options);
    return Status::OK();
# session_factory.cc
1 Status SessionFactory::GetFactory(const SessionOptions& options, SessionFactory**
out_factory);
# direct_session.cc
1 Session* NewSession(const SessionOptions& options);
    DirectSession* session = new DirectSession(options, new DeviceMgr(devices), this);
    return session;
0 DirectSession::DirectSession(const SessionOptions& options, const DeviceMgr* device_mgr,
DirectSessionFactory* const factory);
```

Session 在 C++ 层的代码中是一个集成了计算资源和用于 graph 处理的对象。

Session 在创建时能获取当前机器上所有可用的计算资源（CPU、GPU 等），并使用 device_mgr_类对其进行管理。Session 根据计算资源维护一个动态的线程池，当获取到一个 graph 并启动运行时，Session 就会将 graph 调度到线程池中交由空闲的计算资源来完成计算。

sess.run(...)

Session 类中没有 run 这个方法，因此去它的父类 BaseSession 中找。

```
def run(self, fetches, feed_dict=None, options=None, run_metadata=None):
```

BaseSession 中对 run 这个方法有详细的说明，调用一次 run 是执行一遍数据流图，在 TensorFlow 的训练代码中通常是在一个循环中多次调用 sess.run()，一次 run 即为训练过程中的一步。

fetches 是 run 方法的一个输入参数，这个参数可以是很多种形式的数据，run 最后的返回值也会和 fetches 有相同的结构。

C++部分的代码入口是 TF_Run 函数：

```
void TF_Run(TF_DeprecatedSession* s, const TF_Buffer* run_options,
            // Input tensors
            const char** c_input_names, TF_Tensor** c_inputs, int ninputs,
            // Output tensors
            const char** c_output_names, TF_Tensor** c_outputs, int noutputs,
            // Target nodes
            const char** c_target_oper_names, int ntargets,
            TF_Buffer* run_metadata, TF_Status* status)
```

gdb 调用栈如下：

```
#0  TF_Run () at tensorflow/c/c_api.cc:590
```

```
#1 0x00007fd00f7da65a in tensorflow::TF_Run_wrapper_helper () at tensorflow/python/client/tf_session_helper.cc:499
#2 0x00007fd00f7dacea in tensorflow::TF_Run_wrapper () at tensorflow/python/client/tf_session_helper.cc:549
#3 0x00007fd00f7a6646 in _wrap_TF_Run (args=0x35b02f0) at bazel-out/local-dbg/bin/tensorflow/python/pywrap_tensorflow.cc:5894
```

可以看到中间经过了多层封装。

追踪到最底层是 tensorflow::DirectSession::Run()函数，其中具体的代码执行流程是：

- 累加 session 计数器
- 根据输入输出 tensor、目标节点，从当前 Session 中已有的 executor 中找是否存在一个相同任务的 executor，找到则将其返回，否则创建一个新的 executor
- executor 用于具体的执行
- executor->RunAsync 启动 graph 的运行
- 接收输出

RunAsync 函数在 gdb 中断点的详细位置是：

```
tensorflow::(anonymous namespace)::ExecutorImpl::RunAsync
```

Executor

Executor 是执行 graph 的最底层类，一般的执行过程：

- 创建一个 graph
- 创建一个 Executor，输入的参数有设备信息以及 graph
- 创建一个 Rendezvous
- 把 tensor 输入到 Rendezvous 中
- 调用 executor->Run 运行，参数有 ExecutorOpts 运行选项和前面创建的 Rendezvous
- 从 Rendezvous 中提取输出 tensor，执行结束
- 允许多个线程同时调用 Executor::Run

`executor->Run()` 这个函数是对上面 DirectSession::Run() 里面调用的 `RunAsync()` 的一个同步封装

`ExecutorImpl::RunAsync()` 中最终调用的是 `ExecutorState::RunAsync()`。

论文中提到：TensorFlow 中对 graph 的计算是准备一个调度用的 ready 队列，为 graph 中的每一个 node 维护一个依赖计数器，当计数器到零的时候这个 node 就可以调度给空闲的计算资源进行运算了。

上面的这个计数器应该是用于维护 graph 拓扑结构用的，在 RunAsync 的代码中也得到了体现。

tensorflow::(anonymous namespace)::ExecutorState::RunAsync 中的执行流程：

- 创建一个 ready 队列
- 把入度为 0 的 node 加入 ready 队列

- 调用 ScheduleReady() 把 ready 队列中的内容调度起来运行

RunAsync()似乎是遍历整个 graph，从里面找到入度为 0 的 node（即没有前置依赖的节点）加入 ready 队列，之后对 ready 队列的调度具体由 Session 中维护的设备管理器和线程池来执行。

到这里为止的函数调用栈：

```
#0 tensorflow::(anonymous namespace)::ExecutorState::ScheduleReady (this=0x192a440, ready=..., inline_ready=0x0) at tensorflow/core/common_runtime/executor.cc:1985
#1 0x00007ff5756268b3 in tensorflow::(anonymous namespace)::ExecutorState::RunAsync (this=0x192a440, done=...) at tensorflow/core/common_runtime/executor.cc:1391
#2 0x00007ff57562cb96 in tensorflow::(anonymous namespace)::ExecutorImpl::RunAsync (this=0x160e3c0, args=..., done=...) at tensorflow/core/common_runtime/executor.cc:2502
#3 0x00007ff575514dc6 in tensorflow::DirectSession::Run() at tensorflow/core/common_runtime/direct_session.cc:491
#4 0x00007ff57276d402 in TF_Run_Helper () at tensorflow/c/c_api.cc:557
#5 0x00007ff57276d9ee in TF_Run () at tensorflow/c/c_api.cc:619
#6 0x00007ff572611a20 in tensorflow::TF_Run_wrapper_helper () at tensorflow/python/client/tf_session_helper.cc:499
#7 0x00007ff5726120b0 in tensorflow::TF_Run_wrapper () at tensorflow/python/client/tf_session_helper.cc:549
#8 0x00007ff5725d00d1 in _wrap_TF_Run () at remote 0x7ff5d6001400>))
    at bazel-out/local-py3-dbg/bin/tensorflow/python/pywrap_tensorflow.cc:5945
```

前面创建 Session 的时候已经准备好了等待的线程池，这一步调用 ScheduleReady 之后，等待中的闲置线程就可以开始执行了。

ScheduleReady 代码执行了 tensorflow::(anonymous namespace)::ExecutorState::Process 这个函数。从 gdb 向 Process 函数继续追踪，可以看到已经切换到了另外一个线程上了，这个线程之前是由线程池管理着的空闲线程，现在被调度起来开始执行了：

```
#0 tensorflow::(anonymous namespace)::ExecutorState::Process (this=0x1fad090, tagged_node=..., scheduled_usec=0) at tensorflow/core/common_runtime/executor.cc:1441
#1 0x00007f75f2d73917 in tensorflow::(anonymous namespace)::ExecutorState::__lambda4::operator() (__closure=0x29f4370) at tensorflow/core/common_runtime/executor.cc:1995
#2 0x00007f75f2d7a768 in std::_Function_handler<void(), tensorflow::(anonymous namespace)::ExecutorState::ScheduleReady()> (__functor=...) at /usr/lib/gcc/x86_64-redhat-linux/4.8.5/../../../../include/c++/4.8.5/functional:2071
#3 0x00007f75efd7f8ec in std::function<void ()>::operator()() const (this=0x29f3ad0) at /usr/lib/gcc/x86_64-redhat-linux/4.8.5/../../../../include/c++/4.8.5/functional:2471
#4 0x00007f75f2f65960 in tensorflow::thread::EigenEnvironment::ExecuteTask (this=0x28d8a38, t=...) at tensorflow/core/lib/core/threadpool.cc:82
#5 0x00007f75f2f678a4 in Eigen::NonBlockingThreadPoolTempl<tensorflow::thread::EigenEnvironment>::WorkerLoop (this=0x28d8a30, thread_id=18) At external/eigen_archive/unsupported/Eigen/CXX11/src/ThreadPool/NonBlockingThreadPool.h:199
```

```
#6 0x00007f75f2f6630a in Eigen::NonBlockingThreadPoolTempl<tensorflow::thread::EigenEnvironment>::NonBlockingThreadPoolTempl(int, tensorflow::thread::EigenEnvironment)::{lambda()#1}::operator()() const (
    __closure=0x29ea660) at external/eigen_archive/unsupported/Eigen/CXX11/src/ThreadPool/NonBlockingThreadPool.h:59
#7 0x00007f75f2f6884f in std::_Function_handler<void (), Eigen::NonBlockingThreadPoolTempl<tensorflow::thread::EigenEnvironment>::NonBlockingThreadPoolTempl(int, tensorflow::thread::EigenEnvironment)::{lambda()#1}>::_M_invoke(std::_Any_data const&) (__functor=...) at /usr/lib/gcc/x86_64-redhat-linux/4.8.5/../../../../include/c++/4.8.5/functional:2071
……
#15 0x00007f75ecaf3230 in std::(anonymous namespace)::execute_native_thread_routine (__p=<optimized out>) at ../../../../../libstdc++-v3/src/c++11/thread.cc:84
#16 0x00007f7652fa3dc5 in start_thread (arg=0x7f75267fc700) at pthread_create.c:308
#17 0x00007f76525c973d in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:113
```

从 Process 中需要再往下钻一下。

六、分布式运行时

分布式版的 TensorFlow 从启动上跟单节点版不同，上层的运行时环境也有所区别。

首先明确一下 TensorFlow 官方给出的几个术语：

从底层角度来看：

Server：通常一台机器上运行一个 Server，用于管理机器上的计算资源、运行 graph 以及完成对远程服务的通信。每个 Server 中都会运行 2 种服务，Master Service 和 Worker Service。

Master Service：一个 RPC 进程，用于启动 Session 和管理远程的 Worker Service。

Worker Service：一个 RPC 进程，用于执行具体的运算。

Client：用户操作的那个程序进程叫 Client，即写代码的那一个进程。

再高一层：

Task：Task 对应了一个 Server，用于具体处理某些任务。

Job：一组 Task 可以共同完成一个 Job。

Cluster：一整个分布式运行时的集合，包括运行任务、设备等等。一个 Cluster 中包含了一个或多个 Job，每个 Job 又可分为一个或多个 Task。

例如：

tf.train.ClusterSpec({ "worker": ["worker0.example.com:2222", "worker1.example.com:2222",	/job:worker/task:0 /job:worker/task:1 /job:worker/task:2 /job:ps/task:0

"worker2.example.com:2222"], "ps": ["ps0.example.com:2222", "ps1.example.com:2222"]})	/job:ps/task:1

这个 Cluster 中包含了 5 台机器，这 5 台机器中有 3 台负责 worker 的 Job，2 台负责 ps 的 Job。

注意这里的 worker 是 Job 的名字，与上面的 Worker Service 的概念是不一样的。

每个 Task 都是一台机器，每台机器上都运行一个 TensorFlow Server 进程。每个 TensorFlow Server 进程中都包含一个 Master Service 和 Worker Service。

若以 worker0 机器作为 Master，则 worker0 的 Master Service 会启动一个 Session，并负责与其他 4 台机上的 Worker Service 进行通信，将任务分发给它管理的这些 Worker Service 进行处理。

server = tf.train.Server()

创建的 Server 的默认走的是 gRPC 协议。

一路跟踪到底是一个 RendezvousMgr 结构，用于管理生成的一系列 Rendezvous 实例，Rendezvous 用于处理 node 之间的通信，是最底层 send 和 recv 操作的一个后端。

创建到 RpcRendezvousMgr 时的函数调用栈：

```
#0  tensorflow::RpcRendezvousMgr::RpcRendezvousMgr (this=0x2492470, env=0x11cfbd8)
at tensorflow/core/distributed_runtime/rpc/rpc_rendezvous_mgr.cc:337
#1  0x00007ff9c777fce6 in tensorflow::GrpcServer::Init (this=0x11cfaf0) at
tensorflow/core/distributed_runtime/rpc/grpc_server_lib.cc:202
#2  0x00007ff9c7780a9f in tensorflow::GrpcServer::Create (server_def=..., env=0x15e7cb0,
out_server=0x7ffd0279fa30) at
tensorflow/core/distributed_runtime/rpc/grpc_server_lib.cc:286
#3  0x00007ff9c7780bcc in tensorflow::(anonymous
namespace)::GrpcServerFactory::NewServer (this=0x15e7800, server_def=...,
out_server=0x7ffd0279fa30)
    at tensorflow/core/distributed_runtime/rpc/grpc_server_lib.cc:301
#4  0x00007ff9c78a23ad in tensorflow::NewServer (server_def=...,
out_server=0x7ffd0279fa30) at tensorflow/core/distributed_runtime/server_lib.cc:70
#5  0x00007ff9c772dc0c in PyServer_New (server_def=..., out_server=0x7ffd0279fa30,
out_status=0x11863c0) at bazel-out/local-py3-
dbg/bin/tensorflow/python/pywrap_tensorflow.cc:4117
#6  0x00007ff9c773adf5 in _wrap_PyServer_New
(args=(b'\n2\n0\n\x06worker\x12\x11\x12\x0flocalhost:22222\x12\x13\x08\x01\x12\x0flocal
host:22223\x12\x06worker*\x04grpc', <SwigPyObject at remote 0x7ffa2b1c0510>))
    at bazel-out/local-py3-dbg/bin/tensorflow/python/pywrap_tensorflow.cc:8340
```

sess.run(...)

考虑这样一个简单的分布式测试程序：

worker1:

```
$ python
>>> import tensorflow as tf
>>> cluster = tf.train.ClusterSpec({"worker":["localhost:22222","localhost:22223"]})
>>> server = tf.train.Server(cluster, job_name="worker", task_index=1)
```

worker0:

```
$ python
>>> import tensorflow as tf
>>> cluster = tf.train.ClusterSpec({"worker":["localhost:22222","localhost:22223"]})
>>> server = tf.train.Server(cluster, job_name="worker", task_index=0)
>>> with tf.device("/job:worker/task:1"):
...         a = tf.constant(1)
...         b = tf.constant(2)
...         c = a + b
>>> sess = tf.Session("grpc://localhost:22222")
>>> sess.run(c)
```

在本机上开两个 Server，在 worker0 这个窗口编写程序代码，但在代码中指定让 worker1 来运行，以 worker0 来启动 Session，则 worker0 上的 master service 会负责将数据发送给 worker1 上的 worker service 来执行。

运行 sess.run(c)后，gdb 成功断在了 RpcRemoteRendezvous:: RecvFromRemoteAsync 这一函数上：

```
#0  tensorflow::(anonymous namespace)::RpcRemoteRendezvous::RecvFromRemoteAsync
() at tensorflow/core/distributed_runtime/rpc/rpc_rendezvous_mgr.cc:275
#1  0x00007ff9c77db0c3 in tensorflow::BaseRemoteRendezvous::RecvAsync() at
tensorflow/core/distributed_runtime/base_rendezvous_mgr.cc:272
#2  0x00007ff9ca84744f in
tensorflow::RecvOp::ComputeAsync(tensorflow::OpKernelContext*, std::function<void ()>)
(this=0x7ff894047fe0, ctx=0x7ff8bc000be8, done=...)
    at tensorflow/core/kernels/sendrecv_ops.cc:139
#3  0x00007ff9c7852abc in tensorflow::Device::ComputeAsync(tensorflow::AsyncOpKernel*,
tensorflow::OpKernelContext*, std::function<void ()>) (this=0x11cffd0,
op_kernel=0x7ff894047fe0, context=0x7ff8bc000be8,
    done=...) at ./tensorflow/core/common_runtime/device.h:88
#4  0x00007ff9ca78a96d in tensorflow::(anonymous namespace)::ExecutorState::Process
(this=0x7ff894048e20, tagged_node=..., scheduled_usec=0) at
tensorflow/core/common_runtime/executor.cc:1598
#5  0x00007ff9ca78c917 in tensorflow::(anonymous
namespace)::ExecutorState::__lambda4::operator() (__closure=0x7ff894005530) at
tensorflow/core/common_runtime/executor.cc:1995
```

```
#6 0x00007ff9ca793768 in std::_Function_handler<void(), tensorflow::(anonymous namespace)::ExecutorState::ScheduleReady() (__functor=...) at /usr/lib/gcc/x86_64-redhat-linux/4.8.5/../../../../include/c++/4.8.5/functional:2071
#7 0x00007ff9c77988ec in std::function<void ()>::operator()() const (this=0x7ff894000c00) at /usr/lib/gcc/x86_64-redhat-linux/4.8.5/../../../../include/c++/4.8.5/functional:2471
#8 0x00007ff9ca97e960 in tensorflow::thread::EigenEnvironment::ExecuteTask (this=0x2409a68, t=...) at tensorflow/core/lib/core/threadpool.cc:82
#9 0x00007ff9ca9808a4 in Eigen::NonBlockingThreadPoolTempl<tensorflow::thread::EigenEnvironment>::WorkerLoop (this=0x2409a60, thread_id=25)
    at external/eigen_archive/unsupported/Eigen/CXX11/src/ThreadPool/NonBlockingThreadPool.h:199
#10 0x00007ff9ca97f30a in Eigen::NonBlockingThreadPoolTempl<tensorflow::thread::EigenEnvironment>::NonBlockingThreadPoolTempl(int, tensorflow::thread::EigenEnvironment)::{lambda()#1}::operator()() const (
    __closure=0x2490c40) at external/eigen_archive/unsupported/Eigen/CXX11/src/ThreadPool/NonBlockingThreadPool.h:59
……
#19 0x00007ff9c450c230 in std::(anonymous namespace)::execute_native_thread_routine (__p=<optimized out>) at ../../../../../libstdc++-v3/src/c++11/thread.cc:84
#20 0x00007ffa2a9bcdc5 in start_thread (arg=0x7ff8cb7fe700) at pthread_create.c:308
#21 0x00007ffa29fe273d in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:113
```

从调用栈中可以看到，分布式版的执行流程与单节点版大致相同，只是设备管理等等方面会增加额外的步骤。以上调用栈的底层基本与单节点版运行时一致，也是将任务提交给线程池，线程池中的空闲进程进而执行 Process 函数。数据的收发应该是成为了 graph 的一部分，具体的有待进一步的解读和分析。

七、下一步代码分析待完成的工作

1. 单节点运行时的代码分析目前到了 Process()这一函数，输入参数是 graph 中的一个 node，但 graph 结构经过了较好的封装，后续需要对 graph、op、tensor 等存储结构作分析之后才能继续深入；
2. Process 是如何对应给 CPU 和 GPU 等不同的运算设备进行运算的；
3. Session 中线程池是如何进行管理的；
4. 分布式部分相对较底层的类是 RendezvousMgr 以及 Rendezvous，该类的调度和详细的运行机制；
5. TensorFlow 如何分解 graph，并在其中加入通信节点。