(19)中华人民共和国国家知识产权局

(12)发明专利申请

(10)申请公布号 CN 107370796 A
(43)申请公布日 2017.11.21

(21)申请号 201710525971.1

(22)申请日 2017.06.30

(71)申请人 香港红鸟科技股份有限公司
地址 中国香港湾仔港湾道6-8号瑞安中心17楼1701室

(72)发明人 陈力 夏嘉诚 陈凯

(74)专利代理机构 北京中恒高博知识产权代理有限公司 11249
代理人 宋敏

(51)Int.Cl.
H04L 29/08(2006.01)
G06F 17/30(2006.01)
G06N 99/00(2010.01)
G06F 9/50(2006.01)

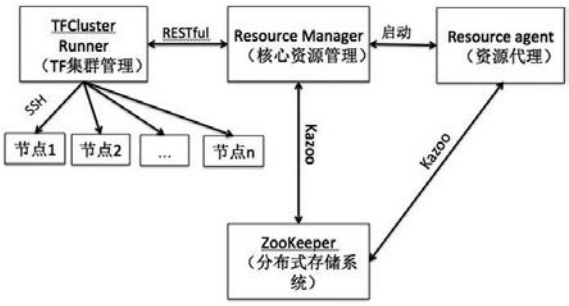
权利要求书1页 说明书9页 附图2页

(54)发明名称

一种基于Hyper TF的智能学习系统

(57)摘要

本发明公开了一种基于Hyper TF的智能学习系统,主要包括:TF集群管理模块、核心资源管理模块、分布式存储模块和资源代理模块;所述核心资源管理模块分别与所述TF集群管理模块、分布式存储模块和资源代理模块双向通信,所述分布式存储模块与资源代理模块进行双向通信。本发明的一种基于Hyper TF的智能学习系统可以实现高效率的、轻量化以及低耦合度的优点。



CN 107370796 A

1.一种基于Hyper TF的智能学习系统,其特征在于,主要包括:TF集群管理模块、核心资源管理模块、分布式存储模块和资源代理模块;所述核心资源管理模块分别与所述TF集群管理模块、分布式存储模块和资源代理模块双向通信,所述分布式存储模块与资源代理模块进行双向通信。

2.根据权利要求1所述的一种基于Hyper TF的智能学习系统,其特征在于,所述TF集群管理模块包括用户任务提交模块、用户任务处理模块、集群资源管理模块、数据存储模块和节点资源统计模块;

所述用户任务提交模块,向所述集群资源管理模块提交参数信息,并请求资源;

所述用户任务处理模块,通过所述集群资源管理模块获得资源分配信息;

所述集群资源管理模块,对所述节点资源统计模块的节点资源信息进行查询和获取;

所述节点资源统计模块,通过所述数据存储模块对节点资源信息进行写入和更新;

所述数据存储模块,通过所述集群资源管理模块对集群资源信息进行获取和更新。

3.根据权利要求2所述的一种基于Hyper TF的智能学习系统,其特征在于,所述TF集群管理模块提供用户直接使用的接口,与集群节点相连接,集群节点的节点数目为多个;

在集群节点上用自己输入的参数和分布式机器数目启动多机多卡分布式训练程序。

4.根据权利要求3所述的一种基于Hyper TF的智能学习系统,其特征在于,所述TF集群管理模块通过SSH(安全外壳协议)登录到每台节点上运行命令。

5.根据权利要求1所述的一种基于Hyper TF的智能学习系统,其特征在于,所述TF集群管理模块与核心资源管理模块通过RESTful(REpresentational State Transfer)模式来建立连接。

6.根据权利要求5所述的一种基于Hyper TF的智能学习系统,其特征在于,所述TF集群管理模块与核心资源管理模块建立连接后,其具体运行步骤包括:

S1:客户输入参数,所述TF集群管理模块与核心资源管理模块建立连接;

S2:所述TF集群管理模块向核心资源管理模块发送资源请求;

S3:若请求不满足条件,则客户重新输入参数;若请求满足条件,TF集群管理模块获得资源信息,并运行分布式TensorFlow程序,结束后释放资源。

7.根据权利要求1所述的一种基于Hyper TF的智能学习系统,其特征在于,所述分布式存储模块与核心资源管理模块、资源代理模块均采用Kazoo建立连接;

Kazoo是一个提供ZooKeeper高级接口服务的Python库,通过IP地址和端口建立客户端与ZooKeeper服务器取得连接,并进行读取、修改、创建、删除节点的操作。

8.根据权利要求1所述的一种基于Hyper TF的智能学习系统,其特征在于,所述系统中的信息包括服务器资源信息、网络地址、显卡设备、显卡利用率和显卡内存利用率。

9.根据权利要求8所述的一种基于Hyper TF的智能学习系统,其特征在于,所述服务器资源信息采用Python的字典数据结构进行存储。

10.根据权利要求8所述的一种基于Hyper TF的智能学习系统,其特征在于,所述显卡设备的可用性采用服务器个数乘以显卡个数的矩阵进行存储。

一种基于Hyper TF的智能学习系统

[0001]

技术领域

[0002] 本发明涉及智能学习系统技术领域，具体地，涉及一种基于Hyper TF的智能学习系统。

背景技术

[0003] 在大数据时代，深度学习的数据量和模型复杂程度导致对计算量的需求无比巨大，而TensorFlow是目前最流行的分布式机器学习框架。它是一个采用数据流图，用于数值计算的开源库，主要应用在机器学习和神经网络的研究。

[0004] 但是深度学习系统往往需要多台服务器做并行计算，甚至多个集群的服务器来共同训练神经网络。

[0005] 尽管TensorFlow支持分布式学习，但是它需要进行复杂，甚至可能是大量的手动设置参数。在多机多卡(显卡)的分布式TensorFlow程序的调试和运行过程中，每次启动和调试的过程都比较繁琐，需要太多的人力进行手动设置集群参数，具体包括：

1.每次执行时需要对每一个程序代码手动修改hyperparameters(超参数)，比如batch size(批量大小)；

2.登陆到每台服务器上来启动训练程序；

3.人工地管理/分配/释放硬件资源。

[0006] 手动指定这些集群规范对于用户来说是相当困难而且麻烦的，容易出错并且用户需要等待很长时间才能进行标杆测试，特别是对于大型集群。

[0007] 目前的google(谷歌)发布的TensorFlow，不能实现自动化的多显卡(显卡)训练，例如自动化放置设备节点以及并行计算。谷歌建议使用一些集群管理工具比如Kubernetes或者Mesos，但考虑到使用这样的工具需要有很多依赖，因为这些集群管理系统都是比较重量级的系统，使用比较繁琐，而且它们都不是专门为TensorFlow设计的，需要再进行额外支持配置，所以对很多用户来说，这并不是一个最方便的选择。

发明内容

[0008] 本发明的目的在于，针对上述问题，提出一种基于Hyper TF的智能学习系统，以实现高效率的、轻量化的以及低耦合度的优点。

[0009] 为实现上述目的，本发明采用的技术方案是：一种基于Hyper TF的智能学习系统，主要包括：TF集群管理模块、核心资源管理模块、分布式存储模块和资源代理模块；所述核心资源管理模块分别与所述TF集群管理模块、分布式存储模块和资源代理模块双向通信，所述分布式存储模块与资源代理模块进行双向通信。

[0010] 进一步地，所述TF集群管理模块包括用户任务提交模块、用户任务处理模块、集群资源管理模块、数据存储模块和节点资源统计模块；

所述用户任务提交模块，向所述集群资源管理模块提交参数信息，并请求资源；

所述用户任务处理模块，通过所述集群资源管理模块获得资源分配信息；

所述集群资源管理模块，对所述节点资源统计模块的节点资源信息进行查询和获取；

所述节点资源统计模块，通过所述数据存储模块对节点资源信息进行写入和更新；

所述数据存储模块，通过所述集群资源管理模块对集群资源信息进行获取和更新。

[0011] 进一步地，所述TF集群管理模块提供用户直接使用的接口，与集群节点相连接，集群节点的节点数目为多个；

在集群节点上用自己输入的参数和分布式机器数目启动多机多卡分布式训练程序。

[0012] 进一步地，所述TF集群管理模块通过SSH(安全外壳协议)登录到每台节点上运行命令。

[0013] 进一步地，所述TF集群管理模块与核心资源管理模块通过RESTful(REpresentational State Transfer)模式来建立连接。

[0014] 进一步地，所述TF集群管理模块与核心资源管理模块建立连接后，其具体运行步骤包括：

S1:客户输入参数，所述TF集群管理模块与核心资源管理模块建立连接；

S2:所述TF集群管理模块向核心资源管理模块发送资源请求；

S3:若请求不满足条件，则客户重新输入参数；若请求满足条件，TF集群管理模块获得资源信息，并运行分布式TensorFlow程序，结束后释放资源。

[0015] 进一步地，所述分布式存储模块与核心资源管理模块、资源代理模块均采用Kazoo建立连接；

Kazoo是一个提供ZooKeeper高级接口服务的Python库，通过IP地址和端口建立客户端与ZooKeeper服务器取得连接，并进行读取、修改、创建、删除节点的操作。

[0016] 进一步地，所述系统中的信息包括服务器资源信息、网络地址、显卡设备、显卡利用率和显卡内存利用率。

[0017] 进一步地，所述服务器资源信息采用Python的字典数据结构进行存储。

[0018] 进一步地，所述显卡设备的可用性采用服务器个数乘以显卡个数的矩阵进行存储。

[0019] 本发明的一种基于Hyper TF的智能学习系统，主要包括:TF集群管理模块、核心资源管理模块、分布式存储模块和资源代理模块；所述核心资源管理模块分别与所述TF集群管理模块、分布式存储模块和资源代理模块双向通信，所述分布式存储模块与资源代理模块进行双向通信，可以实现高效率的、轻量化的以及低耦合度的优点。

[0020] 本发明的其它特征和优点将在随后的说明书中阐述，并且，部分地从说明书中变得显而易见，或者通过实施本发明而了解。

[0021] 下面通过附图和实施例，对本发明的技术方案做进一步的详细描述。

附图说明

[0022] 附图用来提供对本发明的进一步理解，并且构成说明书的一部分，与本发明的实施例一起用于解释本发明，并不构成对本发明的限制。在附图中：

图1为本发明所述一种基于Hyper TF的智能学习系统的结构示意图；

图2为本发明所述一种基于Hyper TF的智能学习系统的TF集群管理模块中Hyper TF模块结构示意图；

图3为本发明所述一种基于Hyper TF的智能学习系统的TF集群管理模块运行流程图。

具体实施方式

[0023] 以下结合附图对本发明的优选实施例进行说明，应当理解，此处所描述的优选实施例仅用于说明和解释本发明，并不用于限定本发明。

[0024] 如图1所示，一种基于Hyper TF的智能学习系统，主要包括：TF集群管理模块、核心资源管理模块、分布式存储模块和资源代理模块；所述核心资源管理模块分别与所述TF集群管理模块、分布式存储模块和资源代理模块双向通信，所述分布式存储模块与资源代理模块进行双向通信。

[0025] 其中用户任务提交模块与用户任务处理模块在组件TFCluster runner上实现，集群资源管理模块对应resource manager，数据存储模块对应distributed storage，节点显卡资源统计模块对应 resource agent。

[0026] Resource agent(资源代理)在每个物理机器上获得硬件、网络资源信息，并发送至resource manager进行更新，resource manager连接到一个分布式存储系统，将资源信息保存在上面并进行更新。

[0027] 所述分布式存储模块与核心资源管理模块、资源代理模块均采用Kazoo建立连接；在resource manager和resource agent上都需要和ZooKeeper建立连接来获取和更新资源信息。ZooKeeper服务其实是在服务器上开启了一个服务器服务，我们连接到ZooKeeper服务器上进行节点的操作。Kazoo是一个提供ZooKeeper高级接口服务的Python库，通过IP地址和端口建立客户端与ZooKeeper服务器取得连接，并进行读取、修改、创建、删除节点的操作。

[0028] Zookeeper(分布式存储系统)用来在集群上存储和共享资源信息，确保集群上每台机器上时刻看到的都是一致的信息，并且保证信息不会丢失。

[0029] 在实现中我们选择了Apache ZooKeeper作为分布式存储系统，它是一个为分布式应用程序保存配置信息、进行分布式同步、提供组服务的高性能协调服务器，在Apache Kafka、Alibaba Dubbo、Apache Mesos等流行的分布式程序框架中都有应用。

[0030] Resource manager和ZooKeeper服务需要在集群的某台机器上长期运行，每次使用TFCluster runner来启动分布式程序，仅需要对原本的程序添加一行代码的改动便可兼容HyperTF，并且可以多用户运行多任务，只需要启动多个TFCluster runner即可，具体运行代码如下所示。

```
# 1 parameter, 3 workers
python tfcluster_runner.py
    --training_script complex_mnist.py
    --ps 1 --worker 3
    --batch_size 100
    --learning_rate 0.001
    ...
```

[0031] TFCluster runner(TensorFlow 集群管理)为用户直接使用的接口，用于在集群

上用自己输入的参数和分布式机器数目启动多机多卡分布式训练程序。

[0032] 所述TF集群管理模块提供用户直接使用的接口，与集群节点相连接，集群节点的节点数目为多个；

在集群节点上用自己输入的参数和分布式机器数目启动多机多卡分布式训练程序。

[0033] 所述TF集群管理模块通过SSH(安全外壳协议)登录到每台节点上运行命令；TFCluster runner在获得资源后，会将集群信息和训练参数传递到TensorFlow，在集群节点上启动训练程序。这里我们考虑了三种连接思路：

1、SSH(安全外壳协议)登陆到每台节点上运行命令。

[0034] 2、建立客户端/服务器连接，发送命令到服务器服务，由其运行命令。

[0035] 3、在Python程序中直接调用要执行的训练Python程序。

[0036] 第三种方法里，由于HyperTF和TensorFlow训练代码同为Python语言写成，可以实现直接调用，但是这样做需要对训练程序代码做出很多修改，使之能在另一个Python程序中被调用。这种方法会破坏HyperTF的兼容性，不符合最初的减小工作量的设计初衷，还是保持通过命令行启动TensorFlow的做法。第二种方法是在每台机器上保持运行着一个服务器程序，在TFCluster runner上创建客户端，通过gRPC或套接字(BSD Socket)与服务器端建立连接，然后把命令发送到服务器，由其格式化好命令并通过Python的subprocess库运行命令。通过广泛的调研，我们发现开源集群管理工具TFMesos就采用了这种连接方法。TFMesos的开发采用这种方法是因为Mesos中集群节点中传递信息的缺省方式是gRPC。由于Mesos是个通用系统，只能采用这种客户端/服务器的方法，这么做是为了符合规范。我们的初衷之一是降低与其他系统的耦合度，所以设计中避免对通用集群系统的依赖。最后决定采用第一种方法，用SSH(安全外壳协议)建立连接。

[0037] 如图2所示，所述TF集群管理模块包括用户任务提交模块、用户任务处理模块、集群资源管理模块、数据存储模块和节点资源统计模块；

所述用户任务提交模块，向所述集群资源管理模块提交参数信息，并请求资源；

所述用户任务处理模块，通过所述集群资源管理模块获得资源分配信息；

所述集群资源管理模块，对所述节点资源统计模块的节点资源信息进行查询和获取；

所述节点资源统计模块，通过所述数据存储模块对节点资源信息进行写入和更新；

所述数据存储模块，通过所述集群资源管理模块对集群资源信息进行获取和更新。

[0038] Resource manager(核心资源管理)承担：核心的资源管理、调度功能。

[0039] 所述TF集群管理模块与核心资源管理模块通过RESTful(REpresentational State Transfer)模式来建立连接；在运行分布式程序时，TFCluster runner会向resource manager请求本次计算所需要的集群资源，运行结束后再释放资源，这两种请求类似HTTP中的GET/PUT方法。在这里，我们采用了低耦合度的RESTful(REpresentational State Transfer)模式来建立客户端/服务器连接。

[0040] 1、RESTful是一种web服务设计风格，主要有：

(1)资源是由URI(Uniform Resource Identifier，统一资源标志符)来指定。

[0041] (2)对资源的操作包括获取、创建、修改和删除资源，这些操作正好对应HTTP协议提供的GET、POST、PUT和DELETE方法。

[0042] (3)通过操作资源的表现形式来操作资源。

[0043] 2、RESTful的优点有：

(1)可更高效利用缓存来提高响应速度。

[0044] (2)通讯本身的无状态性可以让不同的服务器的处理一系列请求中的不同请求，提高服务器的扩展性，兼容性好。

[0045] (3)不需要额外的资源发现机制。

[0046] 这里选择了 Flask-RESTful来方便地实现 RESTful-API。

[0047] (一)Resource manager的设计与实现

resource manager是系统中负责资源同步、存储、调度和分配的模块，和系统中其他组件都有连接。resource manager会在运行时在集群中的每一台服务器上启动resource agent，resource agent会在本机上查询网络地址、显卡设备、显卡利用率和显卡内存利用率等信息，得出可用的显卡列表，并将这些资源信息返回到resource manager。每台服务器上的资源的字典结构如下所示：

```
resource_node = {
    "eth0": eth0,
    "eth2": eth2,
    "port": port,
    "gpu": gpu,
    "gpu_avail_list": gpu_avail_list
}
```

其中eth0和eth2分别为192.168开头和10.40开头的Ethernet地址，分别在管理系统和运行应用程序时连接；port为resource agent获得的空闲端口号；显卡为这台服务器上的显卡数量，GPU_avail_list为可用于本次任务的显卡列表，比如这台服务器上有四台显卡，其中1,2,4号空闲，那么返回的显卡为4，GPU_avail_list为[0,1,3]。

[0048] 在每台服务器上，resource agent会通过netifaces获取以太网ip地址，用socket获取空闲端口号，用CUDA的pynvml函数获取显卡利用率、已用内存和总内存等信息，综合利用率和内存判断此台显卡是否可用于执行任务。在resource agent取得资源信息后，会将信息写入在集群中一台节点上运行的ZooKeeper服务，通过ZooKeeper可以保证我们的资源信息不会丢失，并且在集群上每台机器都能共享。

[0049] 接下来resource manager会连接到ZooKeeper服务器，从上面获得所有集群上所有服务器的资源信息及使用情况，构建出资源矩阵，将其也存放到ZooKeeper服务器上，并且和资源信息分隔开来。ZooKeeper上信息存放的结构和Linux文件系统类似，我们的信息如下所示：

```
--------/resources---<list of nodes>
    |
    |---/matrix
```

随后要在resource manager上配置RESTful服务器。Python有很多网络服务器框架，这里我选择了Flask-RESTful来方便地实现RESTful-API。在服务器服务中，我们按照REST方式定义了我们需要的资源，在客户端一端可以容易地使用GET方法获得资源，用PUT方法修改resource manager上的资源：

```
# resource list
api.add_resource(ResourceList, '/resources')
# single resource node
api.add_resource(Single_machine, '/resources/<resource_id>')
# resource matrix
api.add_resource(ResourceMatrix, '/matrix')
```

对这三种资源定义了三个class，每个class中定义了get()和put()函数。在ResourceList类中，当收到TFCluster runner发来的带有ps(训练节点)和worker(参数服务器)数量的GET请求，会调用schedule()函数，为其分配资源。

[0050] schedule()函数用来定义分配资源的算法，目前采用的是比较简单的round-robin即轮询算法，在每台服务器上先选择一台显卡，再向下一台服务器查找并选择一台显卡，一轮不够的话再开始第二轮分配，如下所示：

```
# resource list
api.add_resource(ResourceList, '/resources')
# single resource node
api.add_resource(Single_machine, '/resources/<resource_id>')
# resource matrix
api.add_resource(ResourceMatrix, '/matrix')
```

每分配出一台显卡后会在资源矩阵中将其标记为0，说明不再可用，然后将ZooKeeper服务器上的资源矩阵信息更新。当程序运行结束后，会对资源进行释放，也就是发送PUT请求，将resource manager上的资源矩阵里分配出去的元素更新为1，再同步到ZooKeeper服务器，便完成了一次资源分配和释放的过程。之后的开发可以将schedule()模块继续完善，实现更合理的分配算法，比如多用户时根据应用的重要程度优先级来先后分配资源，根据队列容量、数据的位置等等因素进行调度。

[0051] (二)TFCluster runner的设计与实现

TFCluster runner有读取用户输入的训练参数和服务器数量、与resource manager建立连接并请求资源、在集群上启动分布式TensorFlow训练程序、训练程序运行结束后释放资源等几个功能。每个程序都需要一个TFCluster runner来启动，在资源充足的情况下同一个集群上可以同时运行几个TFCluster runner而不相互影响。

[0052] 如图3为 TFCluster runner运行流程图，所述TF集群管理模块与核心资源管理模块建立连接后，其具体运行步骤包括：

S1:客户输入参数，所述TF集群管理模块与核心资源管理模块建立连接；

S2:所述TF集群管理模块向核心资源管理模块发送资源请求；

S3:若请求不满足条件，则客户重新输入参数；若请求满足条件，TF集群管理模块获得资源信息，并运行分布式TensorFlow程序，结束后释放资源。

[0053] 多任务的同时运行我们用两点来实现：

1.首先是REST API，服务器端一般是多线程的，用来处理多个客户端的连接。我们这里采用了REST API，客户端用PUT和GET操作来与服务器进行信息处理。这两个操作在我们系统中都是瞬时完成，不会产生冲突，所以并不需要多线程来处理请求。

[0054] 2.其次，TFCluster runner 在启动时会为每个任务根据当前的时间生成一个独

一无二的task key(任务键值),之后的程序运行、log日志输出、运行状态检查都会确定程序的身份,不会和其他程序搞混。

[0055] TFCluster runner通过RESTful API与resource manager建立连接,读取用户在命令行输入的参数后用GET命令发出请求,收到资源后将它们整理,再作为参数传递给TensorFlow程序。在每个节点上启动TensorFlow程序的方法是用SSH(安全外壳协议)与集群中的各个服务器建立连接。和Spark配置时一样,在配置HyperTF集群时需要设置免密码(password-less)的方式SSH(安全外壳协议)登陆,就是将主机器的SSH(安全外壳协议)公钥存放到每台从机上。TFCluster runner SSH(安全外壳协议)登陆到目标服务器上之后,会运行一个bash脚本程序,这个脚本也是系统的一部分,包含了激活TensorFlow所在的虚拟环境、打开本系统目录、根据输入的参数运行训练程序、输出log日志到文件等几步操作。在远程启动了每台机器上的训练程序之后,下一步TFCluster runner会检查输出的log文件以判断程序有没有运行完成。当所有参数服务器上的任务都运行完成后,TFCluster runner会向resource manager发送PUT消息,把上面的资源矩阵中之前分配给它的资源重新标记为可用,再写入ZooKeeper中,完成了以上的释放资源后TFCluster runner便会退出,作为主进程的resource manager仍会一直运行。

[0056] 所述系统中的信息包括服务器资源信息、网络地址、显卡设备、显卡利用率和显卡内存利用率。所述服务器资源信息采用Python的字典数据结构进行存储。所述显卡设备的可用性采用服务器个数乘以显卡个数的矩阵进行存储。

[0057] 在TensorFlow的计算中,主要是以显卡为计算设备,分布式运行时以一个显卡为一个计算单位,运行一个task。运行分布式需要本次计算要调用的集群中的每台设备的地址,包括IP地址和端口号,以及CUDA设备(即显卡)的编号。除此之外,为了调度资源还需要标记每张显卡是否被占用,因为TensorFlow默认会尽可能多的用掉内存,所以一张显卡一般来说只能同时运行一个程序。在最初的实现中,我们采用一个字典来存放所有的信息,用关键字idle对应的值为0或1,分别表示被占用和空闲状态。但通过实践发现,每次分配和释放资源后都要进行更新,其实只更新idle一项,但对每个资源节点都要进行操作,带来了不必要的开销,也复杂化了编程。所以,我们把资源是否空闲这一项专门取出,用一个矩阵resource matrix来表示。本质上resource matrix是一个二维数组Resource}[服务器, 显卡],其中服务器和显卡的编号唯一确定了这张显卡。这样,有关集群计算资源的信息以两种数据结构表示:

1.每台服务器上的资源,用Python的字典数据结构存储;

2.每台设备(显卡)的可用性,用服务器个数×显卡个数的矩阵来存储。

[0058]

本发明的优点:高效率,轻量化,低耦合度。

[0059] 一、高效率原因:

在分布式 TensorFlow 使用和开发的过程中,我们观察到,在使用多机多卡(显卡)的调试和运 行过程中,每次启动程序的过程都比较繁琐,需要比较多的人力来进行重复的工作:

1. 每次执行时需要对所有分布在不同机器上的每一个程序的代码手动修改hyperparameters,比 如 batch size,learning rate,training epoch。

[0060] 2．登陆到每台服务器上来分别启动训练程序。人工地管理、分配、释放硬件资源。这些步骤十分繁琐，尤其集群越大缺陷会更加明显，并且修改参数和分配资源的过程容易出错，

在进行大量全面的benchmark测试或者需要大量调试参数来获得更好准确率时，这些步骤会占去过 多的时间。比如运行一个由2台参数服务器和2台服务器 进行训练的程序，需要在四台服务器 上分别输入命令，如下所示：

```
# On Node 0:
python trainer.py \
    --ps_hosts=ps0.example.com:2222,ps1.example.com:2222 \
    --worker_hosts=worker0.example.com:2222,worker1.example.com:2222 \
    --job_name=ps --task_index=0

# On Node 1:
python trainer.py \
    --ps_hosts=ps0.example.com:2222,ps1.example.com:2222 \
    --worker_hosts=worker0.example.com:2222,worker1.example.com:2222 \
    --job_name=ps --task_index=1

# On Node 2:
python trainer.py \
    --ps_hosts=ps0.example.com:2222,ps1.example.com:2222 \
    --worker_hosts=worker0.example.com:2222,worker1.example.com:2222 \
    --job_name=worker --task_index=0

# On Node 3:
python trainer.py \
    --ps_hosts=ps0.example.com:2222,ps1.example.com:2222 \
    --worker_hosts=worker0.example.com:2222,worker1.example.com:2222 \
    --job_name=worker --task_index=1
```

这样无论是想进行参数调试还是给程序分配资源都很困难，需要在四台机器上进行操作，而且需要在每段程序代码中手动选择为每个任务分配的设备，手动输入hyperparameters。谷歌官方的建议是：“手动指定集群设定参数是十分冗长繁琐的，尤其是在大集群上”。谷歌建议使用集群管理工具比如Kubernetes或者Mesos，但考虑到使用这样的工具需要有很多依赖，与其他软件高度耦合(如Docker容器系统)。因为这些集群管理系统都是比较重量级的系统，使用比较繁琐，而且它们都不是专门为TensorFlow设计的，需要再进行额外支持配置，搭建集群，所以对很多用户来说，这并不是一个最方便的选择。

[0061] 如果采用我们的系统，参见第七章：具体实施方式可发现，只需输入几行代码，至于参数设置已经由系统自动化运行了。这样不仅避免的大量繁琐的参数设置，还可以同时并行地在多机多卡的运行环境下批量操作，集群管理，极大地提高了效率，也规避了手动设置出错的风险。

[0062] 二、轻量化，低耦合度原因

相比谷歌建议使用一些集群管理工具比如Kubernetes或者Mesos，这两种都是较为重量级的系统，而且需要进行额外的配置支持，存在较多依赖也并不是专门面向TensorFlow的。而HyperTF是专门为TensorFlow设计的，充分贴合用户的需求，不需要进行额外支持配置，配置操作简单便捷，具有轻量级，低耦合度的优点。

[0063] 本发明主要技术关键点：

1.粗粒度资源管理：HyperTF以显卡为单位进行资源管理，因为分布式TensorFlow深度

学习任务一般占用的显卡内存比较多，所花的时间也比较长，所以没有用更细粒度的资源比如内存以及时间来对其进行管理。

[0064] 2.多用户多任务共享资源：每个用户要运行一个分布式任务时，只需在集群中任意一台机器上启动TFCluster Runner，可以分别与Resource Manager连接，不同的任务运行在不同的显卡上，所以不会造成冲突，确保了资源的高效利用。另外我们给每个任务分配了唯一的task key，用来在运行中和输出的结果中区分不同的任务。

3.分布式系统通讯：在集群中我们根据不同的连接要求和特点，采用了几种不同的连接方法来进行节点间的通讯，比如和Resource Manager的连接采用RESTful HTTP，方便获取和更新资源；TFCluster Runner通过SSH(安全外壳协议)连接到服务器，直接地运行命令。

[0065] 4.资源信息更新和保存：我们采用Resource Agent组件来实时获取和更新信息，用ZooKeeper来分布式地存储信息，确保信息时刻更新、不会丢失且每台机器都能访问。

[0066] 至少可以达到以下有益效果：

本发明的一种基于Hyper TF的智能学习系统，主要包括：TF集群管理模块、核心资源管理模块、分布式存储模块和资源代理模块；所述核心资源管理模块分别与所述TF集群管理模块、分布式存储模块和资源代理模块双向通信，所述分布式存储模块与资源代理模块进行双向通信，可以实现高效率的、轻量化的以及低耦合度的优点。

[0067] 最后应说明的是：以上所述仅为本发明的优选实施例而已，并不用于限制本发明，尽管参照前述实施例对本发明进行了详细的说明，对于本领域的技术人员来说，其依然可以对前述各实施例所记载的技术方案进行修改，或者对其中部分技术特征进行等同替换。凡在本发明的精神和原则之内，所作的任何修改、等同替换、改进等，均应包含在本发明的保护范围之内。

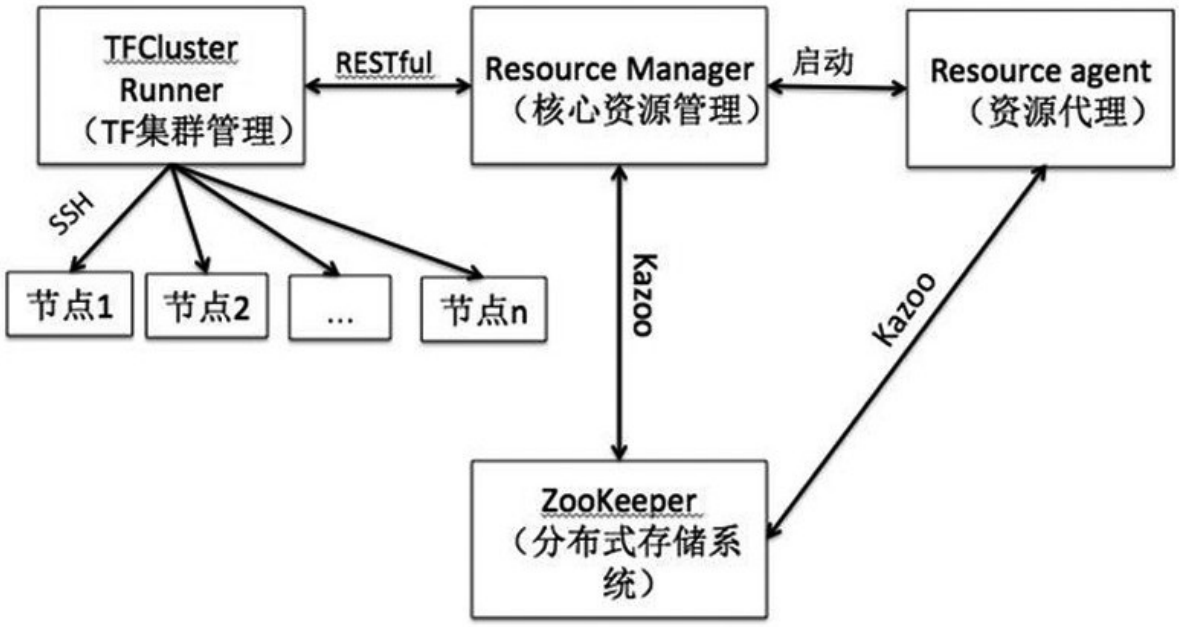

TFCluster
Runner
（TF集群管理）
RESTful
Resource Manager
（核心资源管理）
启动
Resource agent
（资源代理）
SSH
节点1
节点2
...
节点n
Kazoo
Kazoo
ZooKeeper
（分布式存储系统）

图1

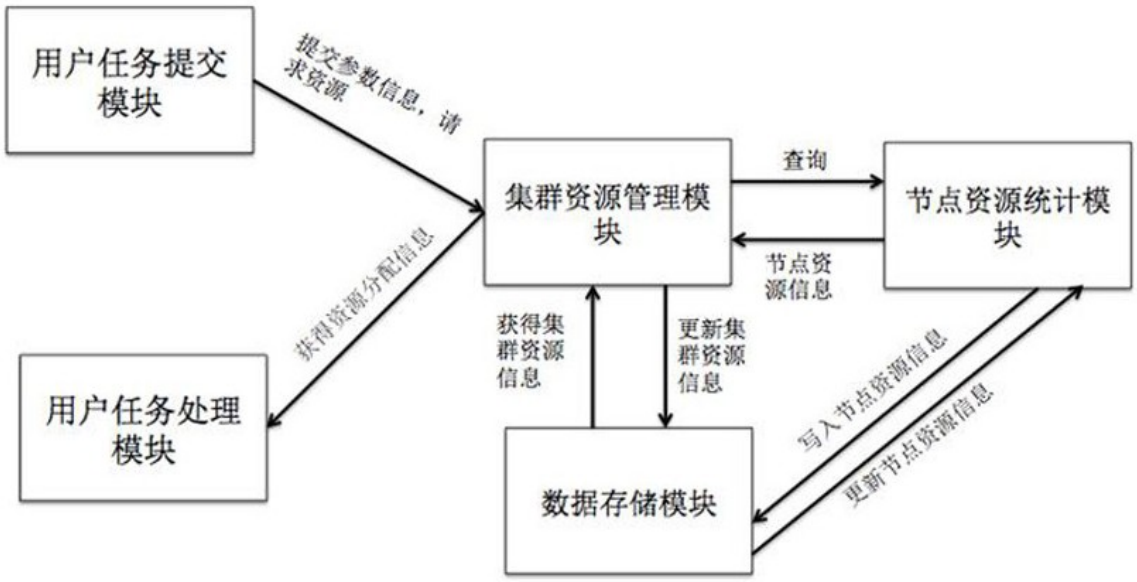

用户任务提交模块
提交参数信息，请求资源
集群资源管理模块
查询
节点资源统计模块
节点资源信息
获得资源分配信息
获得集群资源信息
更新集群资源信息
用户任务处理模块
数据存储模块
写入节点资源信息
更新节点资源信息

图2

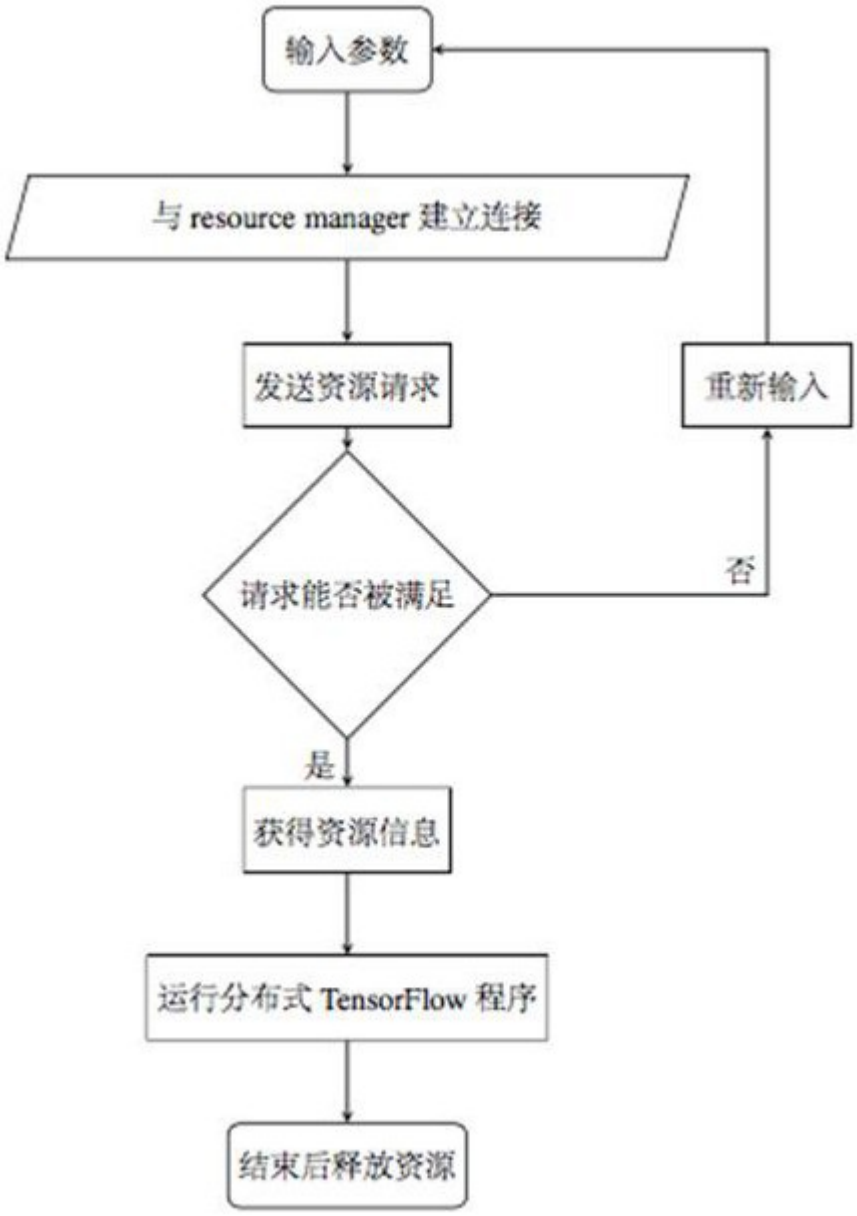

输入参数
与 resource manager 建立连接
发送资源请求
重新输入
请求能否被满足
否
是
获得资源信息
运行分布式 TensorFlow 程序
结束后释放资源

图3