高性能分布式执行框架——Ray

Ray是UC Berkeley RISELab新推出的高性能分布式执行框架，它使用了和传统分布式计算系统不一样的架构和对分布式计算的抽象方式，具有比Spark更优异的计算性能。

Ray目前还处于实验室阶段，最新版本为0.2.2版本。虽然Ray自称是面向AI应用的分布式计算框架，但是它的架构具有通用的分布式计算抽象。

一、简单开始

首先来看一下最简单的Ray程序是如何编写的。

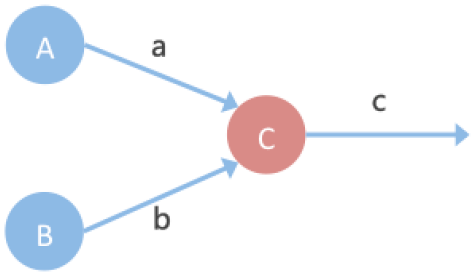
```
# 导入ray，并初始化执行环境
import ray
ray.init()

# 定义ray remote函数
@ray.remote
def hello():
    return "Hello world !"

# 异步执行remote函数，返回结果id
object_id = hello.remote()

# 同步获取计算结果
hello = ray.get(object_id)

# 输出计算结果
print hello
```

在Ray里，通过Python注解`@ray.remote`定义remote函数。使用此注解声明的函数都会自带一个默认的方法`remote`，通过此方法发起的函数调用都是以提交分布式任务的方式异步执行的，函数的返回值是一个对象id，使用`ray.get`内置操作可以同步获取该id对应的对象。熟悉Java里的Future机制的话对此应该并不陌生，或许会有人疑惑这和普通的异步函数调用没什么大的区别，但是这里最大的差异是，函数hello是分布式异步执行的。

remote函数是Ray分布式计算抽象中的核心概念，通过它开发者拥有了动态定制计算依赖（任务DAG）的能力。比如：

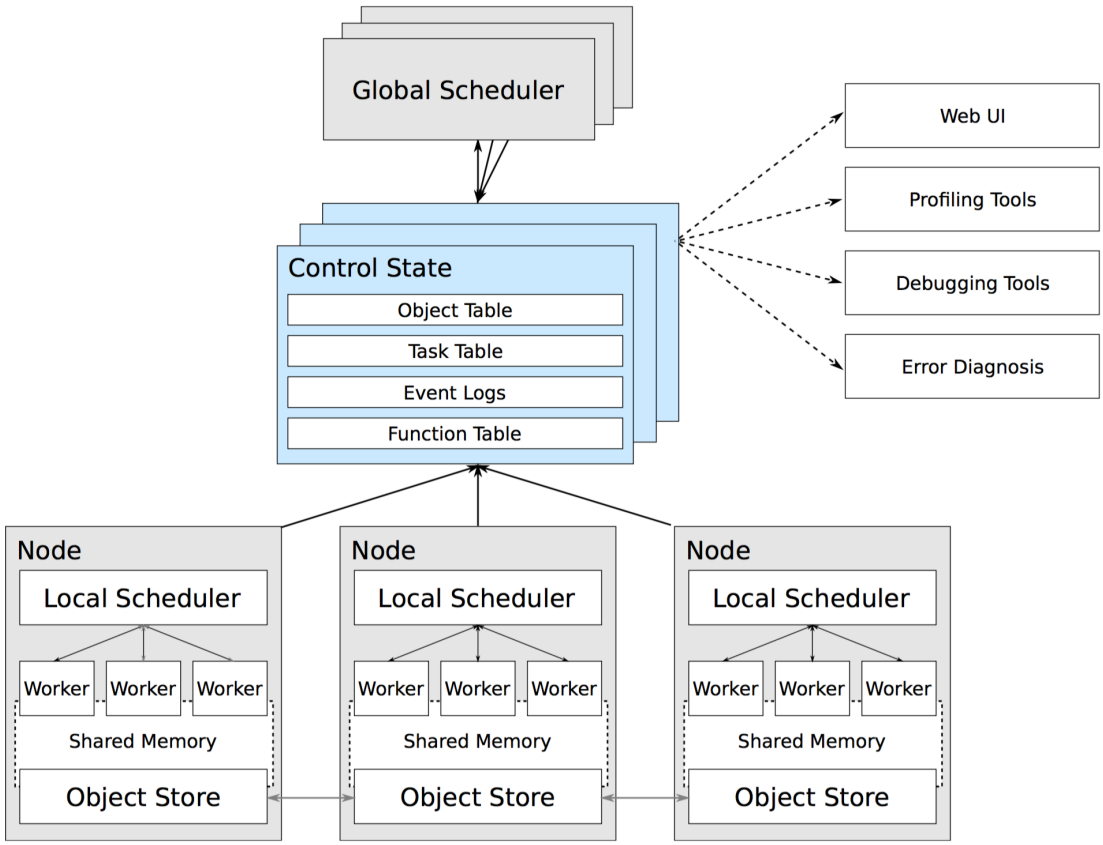
```
@ray.remote
def A():
    return "A"

@ray.remote
def B():
    return "B"

@ray.remote
def C(a, b):
    return "C"

a_id = A.remote()
b_id = B.remote()
c_id = C.remote(a_id, b_id)
print ray.get(c_id)
```

例子代码中，对函数A、B的调用是完全并行执行的，但是对函数C的调用依赖于A、B函数的返回结果。Ray可以保证函数C需要等待A、B函数的结果真正计算出来后才会执行。如果将函数A、B、C类比为DAG的节点的话，那么DAG的边就是函数C参数对函数A、B计算结果的依赖，自由的函数调用方式允许Ray可以自由地定制DAG的结构和计算依赖关系。另外，提及一点的是Python的函数可以定义函数具有多个返回值，这也使得Python的函数更天然具备了DAG节点多入和多出的特点。



二、系统架构

Ray是使用什么样的架构对分布式计算做出如上抽象的呢，一下给出了Ray的系统架构（来自Ray论文，参考文献1）。



作为分布式计算系统，Ray仍旧遵循了典型的Master-Slave的设计：Master负责全局协调和状态维护，Slave执行分布式计算任务。不过和传统的分布式计算系统不同的是，Ray使用了**混合任务调度**的思路。在集群部署模式下，Ray启动了以下关键组件：

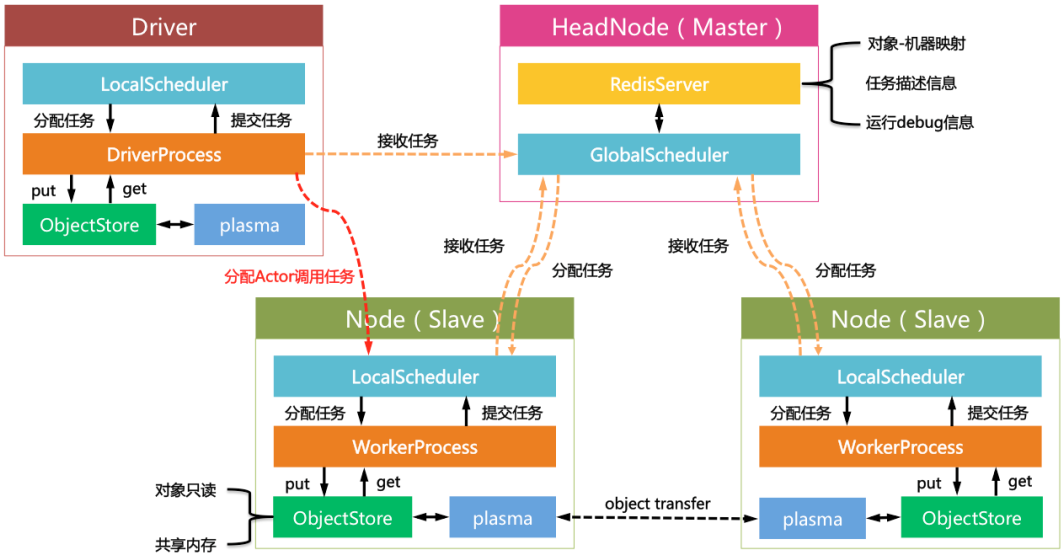
1. **GlobalScheduler**：Master上启动了一个全局调度器，用于接收本地调度器提交的任务，并将任务分发给合适的本地任务调度器执行。
2. **RedisServer**：Master上启动了一到多个RedisServer用于保存分布式任务的状态信息（ControlState），包括对象机器的映射、任务描述、任务debug信息等。
3. **LocalScheduler**：每个Slave上启动了一个本地调度器，用于提交任务到全局调度器，以及分配任务给当前机器的Worker进程。
4. **Worker**：每个Slave上可以启动多个Worker进程执行分布式任务，并将计算结果存储到ObjectStore。
5. **ObjectStore**：每个Slave上启动了一个ObjectStore存储只读数据对象，Worker可以通过共享内存的方式访问这些对象数据，这样可以有效地减少内存拷贝和对象序列化成本。ObjectStore底层由Apache Arrow实现。
6. **Plasma**：每个Slave上的ObjectStore都由一个名为Plasma的对象管理器进行管理，它可以在Worker访问本地ObjectStore上不存在的远程数据对象时，主动拉取其它Slave上的对象数据到当前机器。

需要说明的是，Ray的论文中提及，全局调度器可以启动一到多个，而目前Ray的实现文档里讨论的内容都是基于一个全局调度器的情况。我猜测可能是Ray尚在建设中，一些机制还未完善，后续读者可以留意此处的细节变化。

Ray的任务也是通过类似Spark中Driver的概念的方式进行提交的，有所不同的是：

1. Spark的Driver提交的是任务DAG，一旦提交则不可更改。
2. 而Ray提交的是更细粒度的remote function，任务DAG依赖关系由函数依赖关系自由定制。

论文给出的架构图里并未画出Driver的概念，因此我在其基础上做了一些修改和扩充。



Ray的Driver节点和和Slave节点启动的组件几乎相同，不过却有以下区别：

1. Driver上的工作进程DriverProcess一般只有一个，即用户启动的PythonShell。Slave可以根据需要创建多个WorkerProcess。
2. Driver只能提交任务，却不能接收来自全局调度器分配的任务。Slave可以提交任务，也可以接收全局调度器分配的任务。
3. Driver可以主动绕过全局调度器给Slave发送Actor调用任务（此处设计是否合理尚不讨论）。Slave只能接收全局调度器分配的计算任务。

三、核心操作

基于以上架构，我们简单讨论一下Ray中关键的操作和流程。

1. ray.init()

在PythonShell中，使用`ray.init()`可以在本地启动ray，包括Driver、HeadNode（Master）和若干Slave。

```
import ray
ray.init()
```

如果是直连已有的Ray集群，只需要指定RedisServer的地址即可。

```
ray.init(redis_address="<redis-address>")
```

本地启动Ray得到的输出如下：

```
>>> ray.init()
Waiting for redis server at 127.0.0.1:58807 to respond...
Waiting for redis server at 127.0.0.1:23148 to respond...
Allowing the Plasma store to use up to 13.7439GB of memory.
Starting object store with directory /tmp and huge page support disabled
Starting local scheduler with 8 CPUs, 0 GPUs

======================================================================
View the web UI at http://localhost:8888/notebooks/ray_ui62614.ipynb?token=7c253b0fd66fe41294
======================================================================

{'object_store_addresses': [ObjectStoreAddress(name='/tmp/plasma_store73540254', manager_name
>>>
```

本地启动Ray时，可以看到Ray的WebUI的访问地址。

2. ray.put()

使用`ray.put()`可以将Python对象存入本地ObjectStore，并且异步返回一个唯一的ObjectID。通过该ID，Ray可以访问集群中任一个节点上的对象（远程对象通过查阅Master的对象表获得）。

对象一旦存入ObjectStore便不可更改，Ray的remote函数可以将直接将该对象的ID作为参数传入。使用ObjectID作为remote函数参数，可以有效地减少函数参数的写ObjectStore的次数。

```
@ray.remote
def f(x):
    pass

x = "hello"

# 对象x往ObjectStore拷贝里10次
[f.remote(x) for _ in range(10)]

# 对象x仅往ObjectStore拷贝1次
x_id = ray.put(x)
[f.remote(x_id) for _ in range(10)]
```

3. ray.get()

使用`ray.get()`可以通过ObjectID获取ObjectStore内的对象并将之转换为Python对象。对于数组类型的对象，Ray使用共享内存机制减少数据的拷贝成本。而对于其它对象则需要将数据从ObjectStore拷贝到进程的堆内存中。

如果调用`ray.get()`操作时，对象尚未创建好，则get操作会阻塞，直到对象创建完成后返回。get操作的关键流程如下：

1. Driver或者Worker进程首先到ObjectStore内请求ObjectID对应的对象数据。
2. 如果本地ObjectStore没有对应的对象数据，本地对象管理器Plasma会检查Master上的对象表查看对象是否存储其它节点的ObjectStore。
3. 如果对象数据在其它节点的ObjectStore内，Plasma会发送网络请求将对象数据拉到本地ObjectStore。

4. 如果对象数据还没有创建好，Master会在对象创建完成后通知请求的Plasma读取。
5. 如果对象数据已经被所有的ObjectStore移除（被LRU策略删除），本地调度器会根据任务血缘关系执行对象的重新创建工作。
6. 一旦对象数据在本地ObjectStore可用，Driver或者Worker进程会通过共享内存的方式直接将对象内存区域映射到自己的进程地址空间中，并反序列化为Python对象。

另外，`ray.get()`可以一次性读取多个对象的数据：

```
result_ids = [ray.put(i) for i in range(10)]
ray.get(result_ids)  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

4. @ray.remote

Ray中使用注解`@ray.remote`可以声明一个remote function。remote函数时Ray的基本任务调度单元，remote函数定义后会立即被序列化存储到RedisServer中，并且分配了一个唯一的ID，这样就保证了集群的所有节点都可以看到这个函数的定义。

不过，这样对remote函数定义有了一个潜在的要求，即remote函数内如果调用了其它的用户函数，则必须提前定义，否则remote函数无法找到对应的函数定义内容。

remote函数内也可以调用其它的remote函数，Driver和Slave每次调用remote函数时，其实都是向集群提交了一个计算任务，从这里也可以看到Ray的分布式计算的自由性。

Ray中调用remote函数的关键流程如下：

1. 调用remote函数时，首先会创建一个任务对象，它包含了函数的ID、参数的ID或者值（Python的基本对象直接传值，复杂对象会先通过`ray.put()`操作存入ObjectStore然后返回ObjectID）、函数返回值对象的ID。
2. 任务对象被发送到本地调度器。
3. 本地调度器决定任务对象是在本地调度还是发送给全局调度器。如果任务对象的依赖（参数）在本地的ObejctStore已经存在且本地的CPU和GPU计算资源充足，那么本地调度器将任务分配给本地的WorkerProcess执行。否则，任务对象被发送给全局调度器并存储到任务表（TaskTable）中，全局调度器根据当前的任务状态信息决定将任务发给集群中的某一个本地调度器。
4. 本地调度器收到任务对象后（来自本地的任务或者全局调度分配的任务），会将其放入一个任务队列中，等待计算资源和本地依赖满足后分配给WorkerProcess执行。
5. Worker收到任务对象后执行该任务，并将函数返回值存入ObjectStore，并更新Master的对象表（ObjectTable）信息。

`@ray.remote`注解有一个参数`num_return_vals`用于声明remote函数的返回值个数，基于此实现remote函数的多返回值机制。

```
@ray.remote(num_return_vals=2)
def f():
    return 1, 2

x_id, y_id = f.remote()
ray.get(x_id)  # 1
ray.get(y_id)  # 2
```

`@ray.remote`注解的另一个参数`num_gpus`可以为任务指定GPU的资源。使用内置函数`ray.get_gpu_ids()`可以获取当前任务可以使用的GPU信息。

```
@ray.remote(num_gpus=1)
def gpu_method():
    return "This function is allowed to use GPUs {}.".format(ray.get_gpu_ids())
```

5. ray.wait()

`ray.wait()`操作支持批量的任务等待，基于此可以实现一次性获取多个ObjectID对应的数据。

```
# 启动5个remote函数调用任务
results = [f.remote(i) for i in range(5)]
# 阻塞等待4个任务完成，超时时间为2.5s
ready_ids, remaining_ids = ray.wait(results, num_returns=4, timeout=2500)
```

上述例子中，results包含了5个ObjectID，使用`ray.wait`操作可以一直等待有4个任务完成后返回，并将完成的数据对象放在第一个list类型返回值内，未完成的ObjectID放在第二个list返回值内。如果设置了超时时间，那么在超时时间结束后仍未等到预期的返回值个数，则已超时完成时的返回值为准。

6. ray.error_info()

使用ray.error_info()可以获取任务执行时产生的错误信息。

```
>>> import time
>>> @ray.remote
>>> def f():
>>>     time.sleep(5)
>>>     raise Exception("This task failed!!")
>>> f.remote()
Remote function __main__.f failed with:

Traceback (most recent call last):
  File "<stdin>", line 4, in f
Exception: This task failed!!


  You can inspect errors by running

      ray.error_info()

  If this driver is hanging, start a new one with

      ray.init(redis_address="127.0.0.1:65452")
>>> ray.error_info()
[{'type': 'task', 'message': 'Remote function \x1b[31m__main__.f\x1b[39m failed with:\n\nTrac
```

7. Actor

Ray的remote函数只能处理无状态的计算需求，有状态的计算需求需要使用Ray的Actor实现。在Python的class定义前使用`@ray.remote`可以声明Actor。

```
@ray.remote
class Counter(object):
    def __init__(self):
        self.value = 0

    def increment(self):
        self.value += 1
        return self.value
```

使用如下方式创建Actor对象。

```
a1 = Counter.remote()
a2 = Counter.remote()
```

Ray创建Actor的流程为：

1. Master选取一个Slave，并将Actor创建任务分发给它的本地调度器。
2. 创建Actor对象，并执行它的构造函数。

从流程可以看出，Actor对象的创建时并行的。

通过调用Actor对象的方法使用Actor。

```
a1.increment.remote()  # ray.get returns 1
a2.increment.remote()  # ray.get returns 1
```

调用Actor对象的方法的流程为：

1. 首先创建一个任务。
2. 该任务被Driver直接分配到创建该Actor对应的本地执行器执行，这个操作绕开了全局调度器（Worker是否也可以使用Actor直接分配任务尚存疑问）。
3. 返回Actor方法调用结果的ObjectID。

为了保证Actor状态的一致性，对同一个Actor的方法调用是串行执行的。

四、安装Ray

如果只是使用Ray，可以使用如下命令直接安装。

```
pip intall ray
```

如果需要编译Ray的最新源码进行安装，按照如下步骤进行（MaxOS）：

```
# 更新编译依赖包
brew update
brew install cmake pkg-config automake autoconf libtool boost wget
pip install numpy cloudpickle funcsigs click colorama psutil redis flatbuffers cython --ignor
# 下载源码编译安装
git clone https://github.com/ray-project/ray.git
cd ray/python
python setup.py install
# 测试
python test/runtest.py

# 安装WebUI需要的库［可选］
pip install jupyter ipywidgets bokeh

# 编译Ray文档［可选］
cd ray/doc
pip install -r requirements-doc.txt
make html
open _build/html/index.html
```

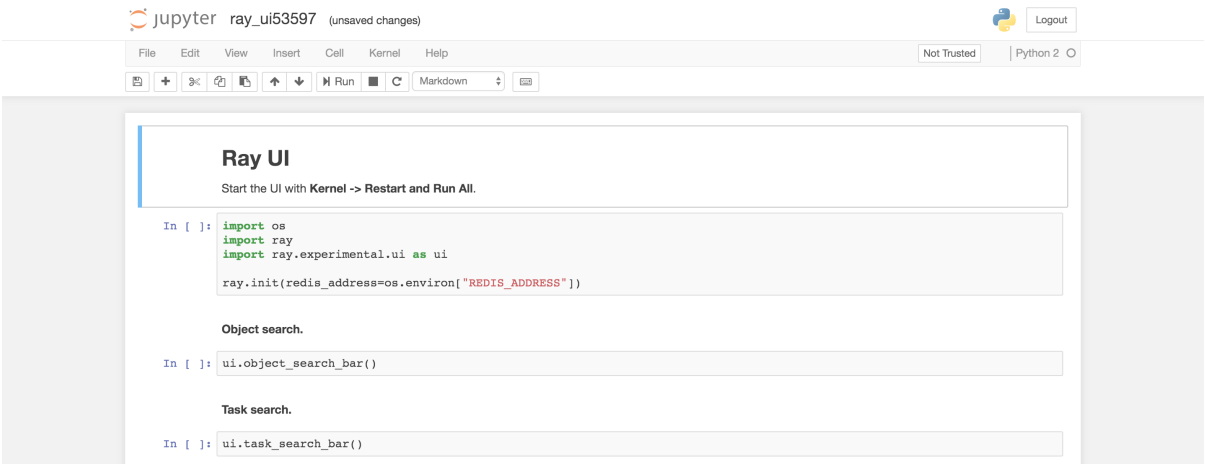
我在MacOS上安装jupyter时，遇到了Python的setuptools库无法升级的情况，原因是MacOS的安全性设置问题，可以使用如下方式解决：

1. 重启电脑，启动时按住`Command+R`进入Mac保护模式。
2. 打开命令行，输入命令`csrutils disable`关闭系统安全策略。
3. 重启电脑，继续安装jupyter。
4. 安装完成后，重复如上的方式执行`csrutils enable`，再次重启即可。

进入PythonShell，输入代码本地启动Ray：

```
import ray
ray.init()
```

浏览器内打开WebUI界面如下：



参考资料

1. Ray论文：[Real-Time Machine Learning: The Missing Pieces]

2. Ray开发手册：Welcome to the Ray documentation — Ray 2.1.0
3. Ray源代码：https://github.com/ray-project/ray