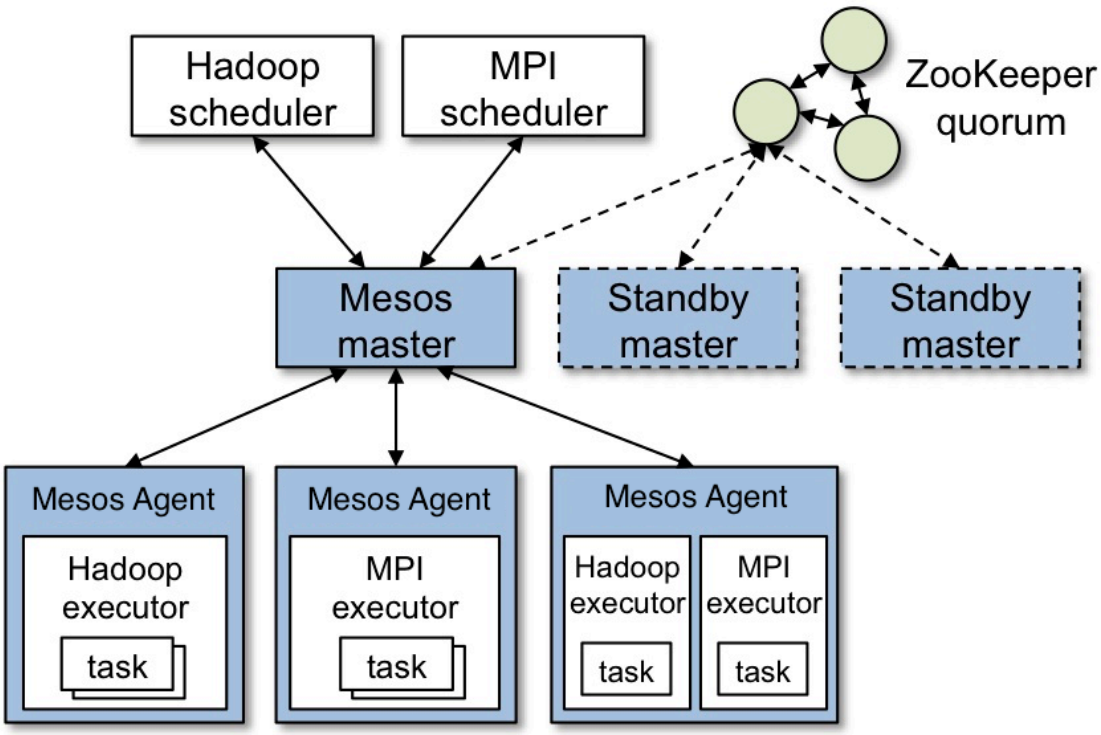
作业二

学号：1300013022

姓名：武守北

一、用自己的语言描述 Mesos 的组成结构，指出它们在源码中的具体位置，简单描述一下它们的工作流程：

Mesos 的组成结构图：



Mesos 架构主要由四部分组成：master、slave(agent)、Framework 的 scheduler 和 executor。外加一个故障恢复的 zookeeper。

1、master

master 代码位于 /src/master 目录下。

master 在 Mesos 架构中居于核心位置。master 负责管理每一个集群节点上运行的 agent 守护进程，以及 agent 上执行具体任务的 frameworks. master 进行对 agent 的管理的重要手段是统筹管理集群资源。master 利用 resource offer 机制实现这一点。resource offer 记录每个节点上有哪些处于空闲状态的资源（包括 CPU、内存、磁盘、网络等），资源列表的维护和更新由 master 完成。master 会根据这一列表中的信息、利用某种分配策略决定下一步为各个节点分配哪些资源。master 的资源分配机制可能需要根据不同运行状况和节点

需求而有所改变。为了适应多种分配机制的需求，master 使用了一种模块化的体系结构来使得添加新的分配策略变得更为容易。在实际应用中，常常会有多个 master 同时存在的情况，它们互为备份，防止由于某一个 master 终止运行造成整个系统意外停止。Mesos 使用 Zookeeper 管理多个 master，并选择其中一个作为主节点执行各项功能。

2、agent

agent 在一些旧的文档中被称为 slave，代码位于 /src/sched 和 /src/slave 目录下。agent 一方面向处于运行状态的 master 报告目前节点上的空闲资源，从而更新 resource offer 的资源列表，另一方面接收 master 关于分配资源和执行任务的指令，将资源分配给具体 framework 的 executor.

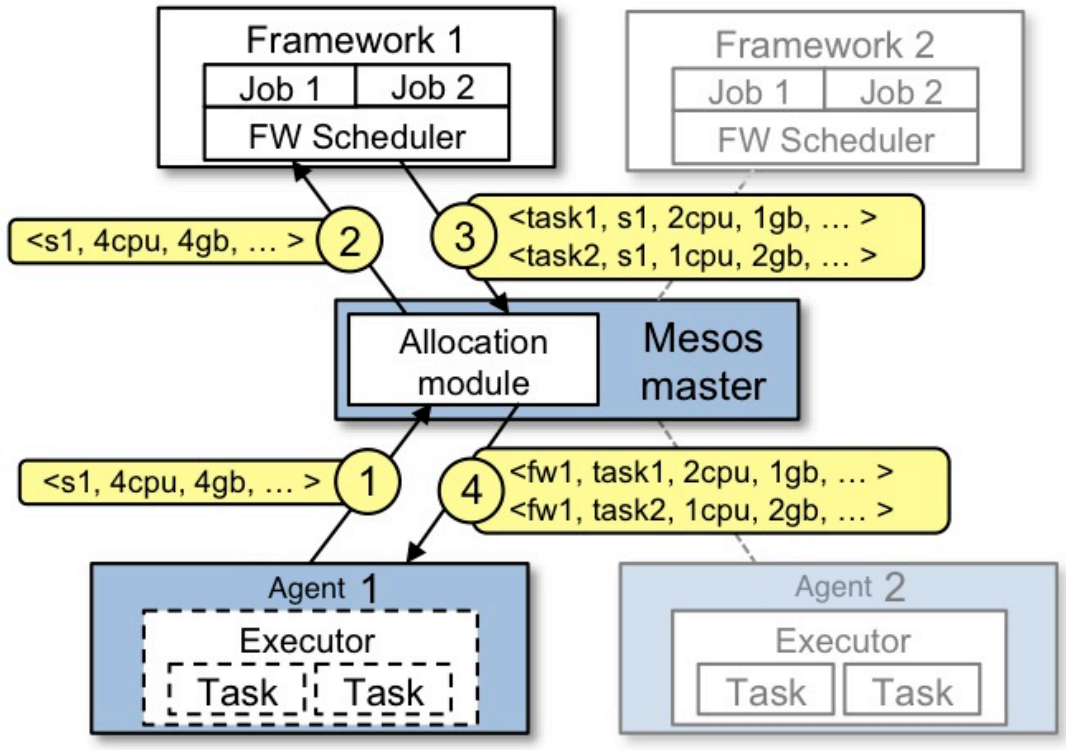
3、frameworks

frameworks 分为 scheduler 和 executor 两部分，负责具体执行某一个任务时的资源调度和执行工作，代码分别位于 /src/scheduler 和 /src/executor 目录下。 scheduler 负责与 master 交流目前 framework 运行需要哪些资源，以及 master 能够提供哪些资源。scheduler 向 master 注册框架信息后，master 会不断告知 scheduler 目前有哪些资源可用，由 scheduler 决定是否接受。若是，scheduler 还需要在接收资源并在节点内部进行分配后，告知 master 各项资源的具体分配信息。executor 负责在接收资源后具体执行任务。新的框架加入集群时也需要 executor 启动框架。

4、Zookeeper

Zookeeper 是一个 Apache 顶级项目。它是一个针对大型应用的数据管理、提供应用程序协调服务的分布式服务框架，提供的功能包括：配置维护、统一命名服务、状态同步服务、集群管理等。在生产环境中， Zookeeper 能够通过同时监控多个 master 在前台或后台运行或挂起，为 Mesos 提供一致性服务。代码位于 /src/zookeeper 目录下。

下图是一个简单的 Mesos 工作流程：



1、当出现以下几种事件中的一种时，会触发资源分配行为：新框架注册、框架注销、增加节点、出现空闲资源等；

2、Mesos Master 中的 Allocator 模块为某个框架分配资源，并将资源封装到 ResourceOffersMessage（Protocal Buffer Message）中，通过网络传输给 SchedulerProcess；

3、SchedulerProcess 调用用户编写的 Scheduler 中的 resourceOffers 函数（不能版本可能有变动），告之有新资源可用；

4、用户的 Scheduler 调用 MesosSchedulerDriver 中的 launchTasks()函数，告之将要启动的任务；

5、SchedulerProcess 将待启动的任务封装到 LaunchTasksMessage（Protocal Buffer Message）中，通过网络传输给 Mesos Master；

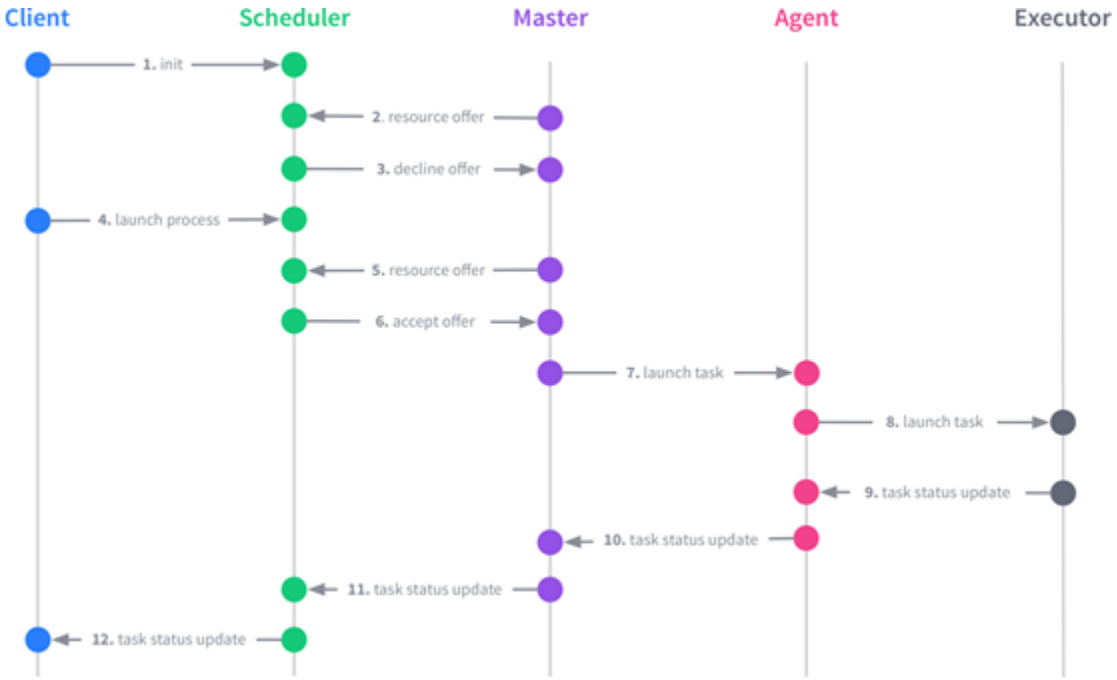
6、Mesos Master 将待启动的任务封装成 RunTaskMessage 发送给各个 Mesos Slave；

7、Mesos Slave 收到 RunTaskMessage 消息后，将之进一步发送给对应的 ExecutorProcess；

8、ExecutorProcess 收到消息后，进行资源本地化，并准备任务运行环境，最终调用用户编写的 Executor 中的 launchTask 启动任务（如果 Executor 尚未启动，则先要启动 Executor。

总的来说 Mesos 是一个二级调度机制，第一级是向框架提供总的资源，第二级由框架自

身进行二次调度然后将结果返回给 Mesos。



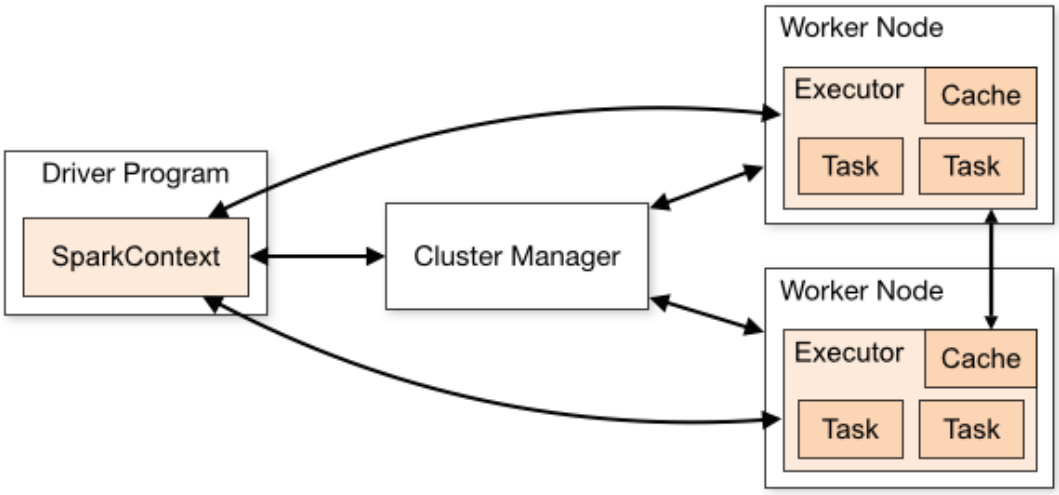
Master 部分在 /path/to/mesos/src/master 下，main.cpp 是入口程序，其内部会生成一个 master 对象，该对象开始监听信息。

Slave 部分在 /path/to/mesos/src/slave 下，同样 main.cpp 是 slave 的入口程序，在处理完若干参数后会生成一个 slave 对象，该对象开始监听信息并发送状态给 master 对象。

MesosSchedulerDriver 的启动模块在 /path/to/mesos/src/sched/sched.cpp 下，它创建一个 scheduler 的进程等待 framework 通过 http 的方式来注册，相当于给外部 framework 提供了一个接口。

MesosExecutorDriver 的启动模块在 /path/to/mesos/src/exec/exec.cpp 下，同理它创建了一个 executor 进程等待 framework 通过 http 的方式注册。

二、用自己的语言描述框架（如 Spark On Mesos）在 Mesos 上的运行过程，并与在传统操作系统上运行程序进行对比：



将上图的 Cluster Manager 替换为 Mesos，就得到了 Spark on Mesos 的结构图。其工作流程为：

1、Spark 启动后像 Mesos 注册，成为 Mesos Framework 中的一个；

2、用户向 Driver Program 提交 Task；

3、Mesos 对 Worker Nodes 进行调度，来决定在哪个 Slave(Worker Nodes)上运行哪一个 Task；

4、Slave 上的 executor 接受 SparkContext，从而运行特定的 task；

与传统操作系统的对比：

相同点：

1、Mesos 和传统操作系统都提供了一个基本性的功能：对硬件资源（CPU、内存）进行抽象，将硬件接口等底层实现进行屏蔽，从而对上层的结构（如应用）提供资源的抽象与分配管理。

2、Mesos 和传统操作系统都实现了隔离功能：传统操作系统实现了进程与进程之间的隔离；而 Mesos 实现了框架与框架之间、任务与任务之间资源的隔离。

不同点：

1、Mesos 管理的是集群中的资源，一般拥有多台设备；而传统操作系统一般针对单台设备。

2、Mesos 对 IO 设备的速度要求高于传统操作系统。为了调度与协调不同 slave 之间的工作，slave 节点之间、master 与 slave 节点之间的通信速度有较高要求。

3、Mesos 分配资源时，先对某个框架中的任务提供可用资源的数量，任务可以选择性的接受资源，与选择性的拒绝资源，然后 Mesos 再返回被接受的资源。在传统操作系统中一

般是进程要求一定数目的资源，操作系统就相应地分配所需要的资源。

三、叙述 master 和 slave 的初始化过程：

1、Master

Master 的入口是 mesos-1.1.0/src/master/main.cpp

开一个 master::Flags 记录 flags。

flags 用到了 stout 库，主要是其中的 option 和 flag。

flags 涉及到了 LIBPROCESS_IP 等环境变量。

进行 libprocess 库的 process 的初始化。

进行日志 logging 的初始化。

将 warning 写入日志中。

新建一个 VersionProcess 线程用于返回 http 请求的版本号。

初始化防火墙。

初始化模块。

初始化 hooks（暂时不知道有什么作用）。

新建一个分配器的实例。

新建用于 state 的空间。

创建 State 实例。

创建 Registrar 实例。

创建 MasterContender 实例。

创建 MasterDetector 实例。

初始化 Authorizer 相关内容。

初始化 SlaveRemovalLimiter 相关内容。

创建 master 实例，创建 master 线程以监听请求。

等待 master 结束。

垃圾回收。

2、Slave

Slave 的入口是 mesos-1.1.0/src/slave/main.cpp

开一个 slave::Flags 进行 flags 的 chuli。

向 Master 提供资源，每隔"disk_watch_interval"的时间就调用一次 Slave::checkDiskUsage。

输出版本号。

利用 libprocess 生成一个 slave 的 ID。

进行 libprocess 库的 process 的初始化。

进行日志 logging 的初始化。

将 warning 写入日志中。

新建一个 VersionProcess 线程用于返回 http 请求的版本号。

初始化防火墙。

初始化模块。

创建 containerizer。

创建 detector。

Authorizer 管理。

创建 gc、StatusUpdateManager、ResourceEstimator。

创建 slave 实例，创建 slave 线程。

等待 slave 结束。

垃圾回收。

四、查找资料，简述 Mesos 的资源调度算法，指出在源代码中的具体位置并阅读，说说你对它的看法：

Mesos 使用的是 Dominant Resource Fairness 算法(DRF)。

其目标是确保每一个用户，即 Mesos 中的 Framework 能够接收到其最需资源的公平份额。首先定义主导资源(domainant resource)和主导份额。 主导资源为一个 Framework 的某个资源，Framework 所需除以 Master 所有，大于其它所有资源。Framework 所需除以 Master 所有，即为这个 Framework 的主导份额。DRF 算法会解方程，尽量让每一个 Framework 的主导份额相等，除非某个 Framework 不需要那么多的资源。如果是带权重的 DRF 算法，只需将权重归一化再执行 DRF 算法即可。核心算法如图所示：

Algorithm 1 DRF pseudo-code

$R = \langle r_1, \cdots, r_m \rangle$ ▷ total resource capacities
$C = \langle c_1, \cdots, c_m \rangle$ ▷ consumed resources, initially 0
$s_i \ (i = 1..n)$ ▷ user i's dominant shares, initially 0
$U_i = \langle u_{i,1}, \cdots, u_{i,m} \rangle \ (i = 1..n)$ ▷ resources given to user i, initially 0

pick user i with lowest dominant share s_i
$D_i \leftarrow$ demand of user i's next task
if $C + D_i \leq R$ **then**
    $C = C + D_i$ ▷ update consumed vector
    $U_i = U_i + D_i$ ▷ update i's allocation vector
    $s_i = \max_{j=1}^{m} \{u_{i,j}/r_j\}$
else
    return ▷ the cluster is full
end if

算法的主要部分位于 /src/master/allocator/mesos/hierarchical.cpp 中的 HierarchicallAllocatorProcess::allocate()；

五、写一个完成简单工作的框架(语言自选，需要同时实现 scheduler 和 executor)并在 Mesos 上运行，在报告中对源码进行说明并附上源码。

主要内容包括两个文件：

scheduler.py：GetSumScheduler 类的定义和整个 framework 的入口函数

executor.py：GetSumExecutor 类的定义

Scheduler 部分代码执行过程：

1、初始化 exeuctor 信息，包括名称、执行路径和资源信息等。

2、初始化 framework 信息，包括名称、用户信息和 Host。

3、初始化 scheduler 驱动，这个类封装在 pymesos 包中，使程序员摆脱了底层相关的事情，直接调用 API 即可。

4、增加信号处理函数，Ctrl + C。

5、开启运行的线程，然后用 while 等待线程。

6、进入 GetSumScheduler 类，其构造函数首先会读文件，然后将数据平均分成 10 份，然后创建 10 个 task，每个 task 执行一份数据。定义一个 frameworkMessage 方法来接收从 executor 执行回来的结果。

7、在 updateStatus 中判断是否 10 个 tasks 的任务执行完毕，若完毕则可以结束 scheduler。

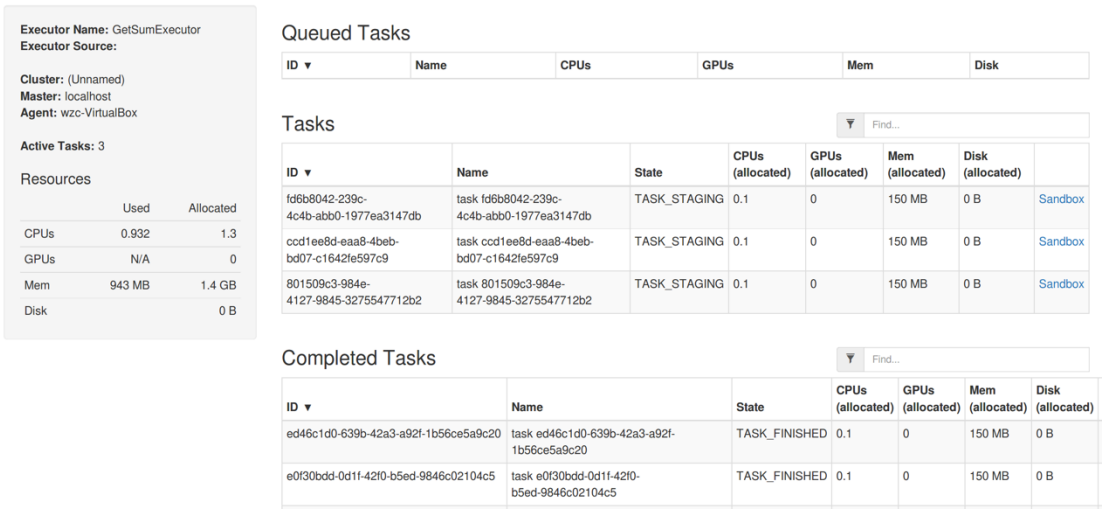
Executor 部分代码执行过程：

1、先发送当前的状态信息，初始化时状态为 RUNNING。

2、接着执行核心的部分。

3、执行结束后会再发送 FINISHED 的信息。（注意如果 executor 代码发生异常，则可能会卡在 RUNNING 和 FINISHED 之间，这里需要再处理）

运行结果如下：



Executor Name: GetSumExecutor
Executor Source:

Cluster: (Unnamed)
Master: localhost
Agent: wzc-VirtualBox

Active Tasks: 3

Resources

	Used	Allocated
CPUs	0.932	1.3
GPUs	N/A	0
Mem	943 MB	1.4 GB
Disk		0 B

Queued Tasks

ID ▾	Name	CPUs	GPUs	Mem	Disk

Tasks

Find...

ID ▾	Name	State	CPUs (allocated)	GPUs (allocated)	Mem (allocated)	Disk (allocated)	
fd6b8042-239c-4c4b-abb0-1977ea3147db	task fd6b8042-239c-4c4b-abb0-1977ea3147db	TASK_STAGING	0.1	0	150 MB	0 B	Sandbox
ccd1ee8d-eaa8-4beb-bd07-c1642fe597c9	task ccd1ee8d-eaa8-4beb-bd07-c1642fe597c9	TASK_STAGING	0.1	0	150 MB	0 B	Sandbox
801509c3-984e-4127-9845-3275547712b2	task 801509c3-984e-4127-9845-3275547712b2	TASK_STAGING	0.1	0	150 MB	0 B	Sandbox

Completed Tasks

Find...

ID ▾	Name	State	CPUs (allocated)	GPUs (allocated)	Mem (allocated)	Disk (allocated)
ed46c1d0-639b-42a3-a92f-1b56ce5a9c20	task ed46c1d0-639b-42a3-a92f-1b56ce5a9c20	TASK_FINISHED	0.1	0	150 MB	0 B
e0f30bdd-0d1f-42f0-b5ed-9846c02104c5	task e0f30bdd-0d1f-42f0-b5ed-9846c02104c5	TASK_FINISHED	0.1	0	150 MB	0 B