集群调度技术研究综述

Author	Taosheng Shi
WeChat Contact	data-lake
Mail Contact	tshshi@126.com
Organization	NOKIA
Document category	Distributed System
Document location	https://github.com/stone-note/articles

Version	Status	Date	Author	Description of changes
0.1	Draft	12/7/2017	Taosheng Shi	Initiate
0.2	Draft	DD-MM-YYYY	YourNameHere	TypeYourCommentsHere
1.0	Approved	DD-MM-YYYY	YourNameHere	TypeYourCommentsHere

Contents

1 引言

什么是调度？个人理解最初的调度是和时间有关的。时间作为唯一的不可逆转的资源，一般是划分为多个时间片来使用（如图 1 所示）。就计算机而言，由于 CPU 的速度快的多，所以就有了分时系统，就有了针对 CPU 时间片的调度，让多个任务在同一个 CPU 上运行起来。然而这是一个假象，某一时刻 CPU 还是单任务运行的。

图 1 时间片的划分

为了在同一时间运行更多的任务，或者多个处理器一起工作完成一个任务目标，就需要一个协调者——这就成为一个分布式系统，就单个数据中心或者小范围来说，这就是集群。如果让一个分布式系统运行多个任务，每个任务对分布式系统中的资源必然产生竞争，调度问题发展到资源调度。

宏观上来说调度主题包括了单机操作系统、C/S 系统、B/S 系统、P2P 系统、集群系统、分布式系统等等，以及网络协议栈、存储协议栈的各种调度机制。本文主要总结了集群调度发展的三个阶段：宏调度、两层调度和状态共享调度，并比较了三者之间的优缺点。

2 集群调度

2.1 宏调度 (Monolithic schedulers)

宏调度：在同一个代码模块中实现调度策略，单个实例，没有并行。常见于 HPC（high-performance computing）世界中。

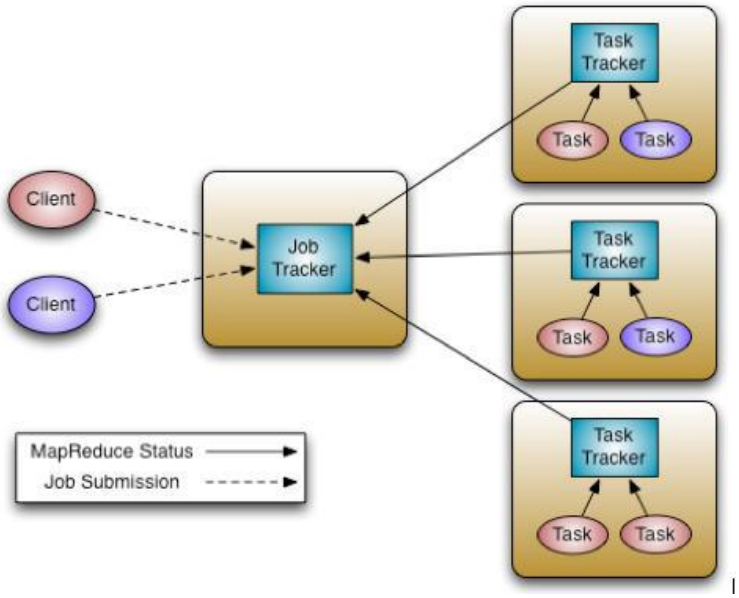

图 2 Hadoop1 和 MapReduce 的宏调度架构

如图 2 所示，以为 MapReduce 为例，一个称之为 JobTracker 的 Master 进程是所有 MapReduce 任务的中心调度器。每一个节点上面都运行一个 TaskTracker 进程来管理各个节点上的任务。各个 TaskTracker 要和 Master 节点上的 JobTracker 通信并接受 JobTracker 的控制。和大多数资源管理器类似，MapReduce 的 JobTracker 支持两种调度策略，Capacity 调度策略和 Fair 调度策略。

在 JobTracker 中，资源的调度和作业的管理功能全部放到一个进程中完成。这种设计方式的缺点是扩展性差：首先，集群规模受限；其次，新的调度策略难以融入现有代码中，比如之前仅支持批处理作业，现在要支持流式作业，而将流式作业的调度策略嵌入到中央式调度器中是一项很难的工作。

2.2 静态分区（Statically partitioned schedulers）

基于静态分区的资源划分和调度也被称为云计算中的调度，通过在云平台中分配和定义虚拟机角色，实现资源集合的全面控制。业务系统往往部署在专门的、静态划分的集群的一个子集上——把集群划分为不同的部分，分别支持不同的业务。现在大多数企业级的云计算都是采用这样计划经济式的资源分配方式——在系统部署之前做好容量规划和资源分配。

2.3 两层调度（Two-level scheduling）

为了处理宏调度和集群静态分区的种种限制，一个直接的解决方案就是两层调度。通过引入一个中央的协调组件来决定每一个子集群需要分配的资源数量，从而动态的调整分配给每一个调度器（框架调度器）的资源。两层调度本质上是在调度中分离资源分配和任务分配，让渡一部分决策权力给应用框架，解决不同应用框架的需求异构问题。如图 3 所示，YARN 为上层不同的应用框架提供了统一的资源调度层。后文对 Mesos 的介绍一节会详细介绍两层调度的需求背景。

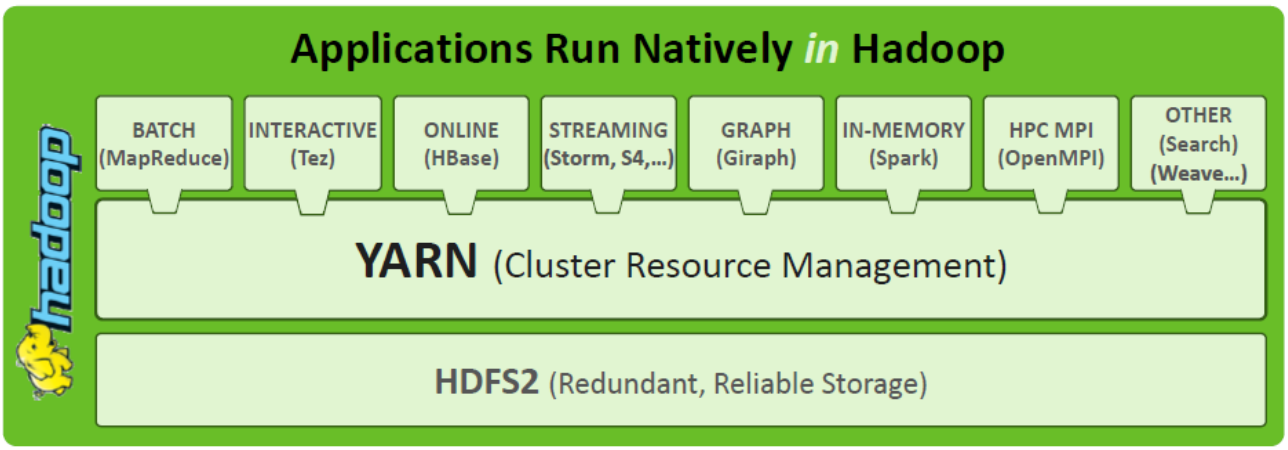

图 3 YARN 为上层不同的应用框架提供了统一的调度层

各个框架调度器并不知道整个集群资源使用情况，只是被动的接收资源。中央协调组件仅将可用的资源推送给各个框架，而框架自己选择使用还是拒绝这些资源。一旦框架（比如 JobTracker）接收到新资源后，再进一步将资源分配给其内部的各个应用程序（各个 MapReduce 作业），进而实现双层调度。

双层调度器有两个缺点，其一，各个框架无法知道整个集群的实时资源使用情况；其二，采用悲观锁，并发粒度小。

2.3.1 YARN

YARN 被称之为 Apache Hadoop Next Generation Compute Platform，是 hadoop1 和 hadoop2 之间最大的区别如图 4 所示。

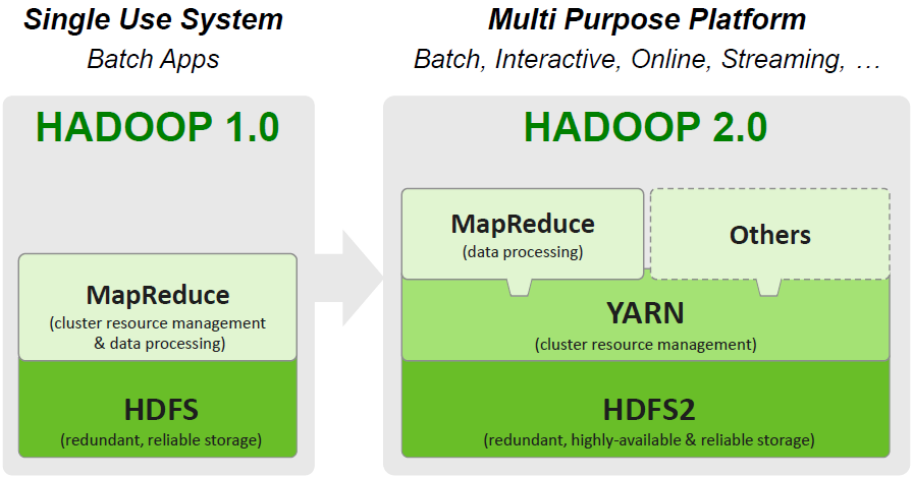

图 4 在 Hadoop 2.0 中引入 YARN

Hadoop2（MRv2）的基础思想就是把 JobTracker 的功能划分成两个独立的进程：全局的资源管理 ResourceManager 和每个进程的监控和调度 ApplicationMaster。 这个进程可以是 Map-Reduce 中一个任务或者是 DAG 中一个任务。

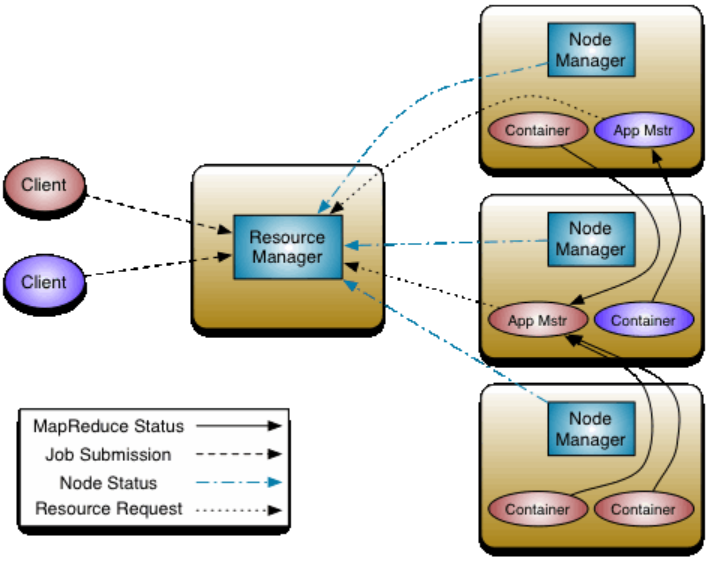

图 5 YARN 的调度框架

在 YARN 的设计中，集群中可以有多个 ApplicationMasters，每一个 ApplicationMasters 可以有多个 Containers （例如，图 5 中有两个 ApplicationMasters，红色和蓝色。红色的有三个 Containers，蓝色的有一个 Container）。 关键的一点是 ApplicationMasters 不是 ResourceManager 的部分，这就减轻了中心调度器的压力，并且，每一个 ApplicationMasters 都可以动态的调整自己控制的 container。

而 ResourceManager 是一个纯粹的调度器（不监控和追踪进程的执行状态，也不负责重启故障的进程），它唯一的目的就是在多个应用之间管理可用的资源（以 Containers 的粒度）。ResourceManager 是资源分配的终极权威。如果说 ResourceManager 是 Master，NodeManager 就是其 slave。ResourceManager 并且支持调度策略的插件化，CapacityScheduler 和 FairScheduler 就是这样的插件。

ApplicationMaster 负责任务提交，通过协商和谈判从 ResourceManager 那里以 Containers 的形式获得资源（负责谈判获得适合其应用需要的 Containers）。然后就 track 进程的运行状态。ApplicationMasters 是特定于具体的应用，可以根据不同的应用来编写不同的 ApplicationMasters。例如，“YARN includes a distributed Shell framework that runs a shell script on multiple nodes on the cluster. ” 另外，ApplicationMaster 提供自动重启的服务。ApplicationMaster 可以理解为应用程序可以自己实现的接口库。

ApplicationMasters 请求和管理 Containers。Containers 指定了一个应用在某一台主机上可以使用多少资源 (包括 memory, CPU 等) ，这类似于 HPC 调度中的资源池。ApplicationMaster 一旦从 ResourceManager 那里获得资源，它就会联系 NodeManager 来启动某个特定的任务。例如如果使用 MapReduce 框架，这些任务可能就是 Mapper 和 Reducer 进程。不同的框架会有不同的进程。

NodeManager 是每一个机器上框架代理，负责该机上的 Containers，并且监控可用的资源(CPU, memory, disk, network)。并且资源状态报告给 ResourceManager。

表面上看来，YARN 也是一个两层调度。在 YARN 中，资源请求从 Application Masters 发出到一个中心的全局调度器上，中心调度器根据应用的需要在集群中的多个节点上分配资源。但是 YARN 中的

Application Masters 提供的仅仅是一个任务管理服务，并不是一个真正的二层调度器。因此本质上 YARN 仍旧是一个宏调度架构。截止目前，YARN 只支持资源类型（内存）的调度。

Hadoop2（MRv2）的 API 是后向兼容的，支持 Map-Reduce 的任务只需要重新编译一下就可以运行在 Hadoop2（MRv2）上。

2.3.2 Mesos

大量分布式计算框架（Hadoop, Giraph, MPI, etc）的出现，每一个计算框架需要管理自己的计算集群。这些计算框架往往把任务分割成很多小任务，让计算靠近数据，从而可以提高集群的利用率。但是这些框架都是独立开发的，不可能在应用框架之间共享资源。形象的表示如图 6 所示。

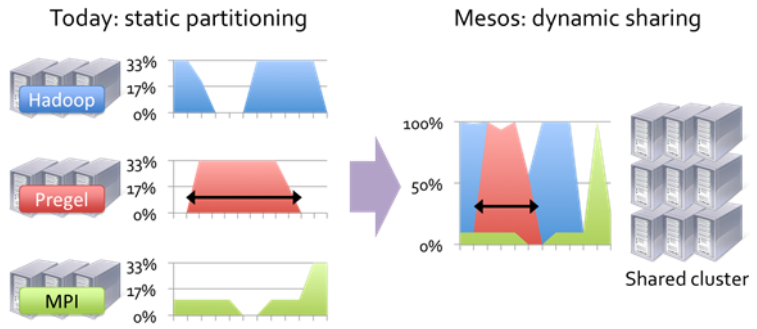

图 6 集群的静态分区和动态共享

我们希望在同一个集群上可以运行多个应用框架。 Mesos 是通过提供一个通用资源共享层，多个不同的应用框架可以运行在这个资源共享层之上。形象的表示如图 7 所示。

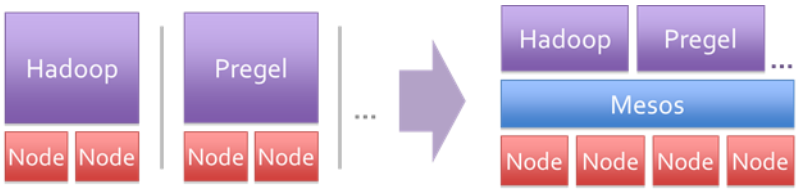

图 7 Mesos 为不同应用框架提供统一调度接口

但我们不希望使用简单的静态分区的方法，如图 8 所示。

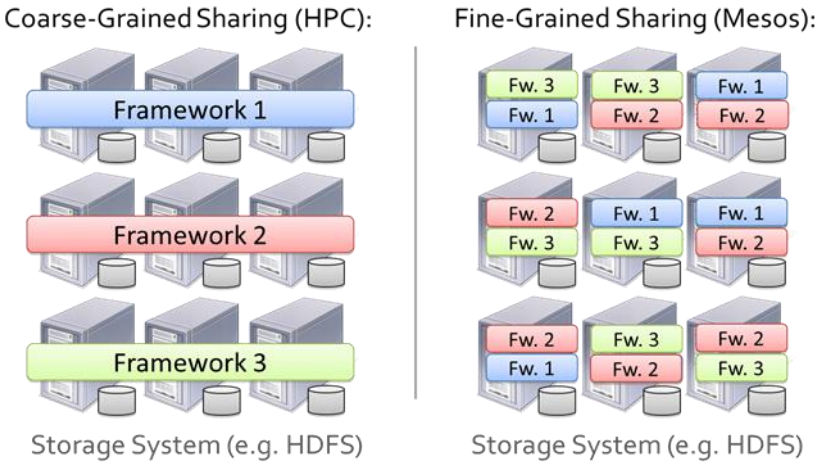

图 8 集群内静态分区

Mesos 的英文定义为：“ Mesos, which is an open source platform for fine-grained resource sharing between multiple diverse cluster computing frameworks. ”

Mesos 最大的好处就是提高集群的利用率。可以很好的隔离产品环境和实验环境，可以同时并发运行多个框架。其次，可以在多个集群之间共享数据。第三，可以降低维护成本。 Mesos 最大挑战是如何支持大量的应用框架。因为每一个框架都有不同的调度需求：编程模型、通信范型、任务依赖和数据放置。另外 Mesos 的调度系统需要能够扩展到数千个节点，运行数百万个任务。由于集群中的所有任务都依赖于 Mesos，调度系统必须是容错和高可用的。

Mesos 的设计决策（设计哲学）：不采用中心化的，设计周全的（应用需求，可用资源，组织策略），适用于所有任务的全局调度策略。而采用委派调度任务给应用框架（把调度和执行的功能交给应用框架）。 Mesos 声称：这样的设计策略可能不会达到全局最优的调度，但在实际运行中出奇的好，可以使得应用框架的近乎完美的达到目标。其声称中的优点主要有两个：应用框架的演化独立和保持 Mesos 的简洁。

Mesos 的主要组件包括 Master daemon，Slave daemons 和在 slaves 之上运行的 Mesos applications (也被称为 frameworks)（如图 9 所示）。Master 根据相应的策略（公平调度，优先级调度等）决定给每一个应用分配多少资源。模块化架构支持多种策略。Resource offer 是资源的抽象表示，基于该资源，应用框架可以在集群中的某个 node 上实例化分配 offer，并运行任务。每一个 Resource offer 就是一个分布在多个 node 上的空闲资源列表。 Mesos 基于一定的算法策略（如公平调度）决定有多少资源可以分配给应用框架，而应用框架决定使用（接受）哪些资源，运行哪些任务。Mesos 上运行的应用框架由两部分组成：应用调度器和 slave 上运行代理。应用调度器向 Mesos 注册。Master 决定向注册的框架提供多少资源，应用调度器决定 Master 分配的资源中哪些来使用。调度完成之后，应用调度器把接受的资源发送给 Mesos，从而决定了使用哪些 slave。然后应用框架中的任务的执行可以在 slave 上运行。当任务很小并且是短期任务（每个任务都频繁的让渡自己握着的资源的时），Mesos 工作的很好。

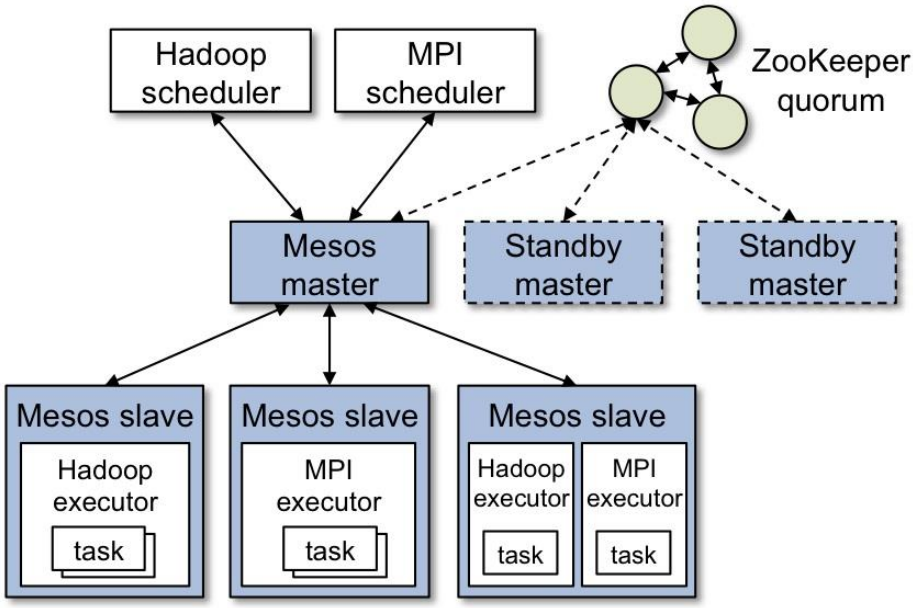

图 9 Mesos 调度框架

在 Mesos 中，一个中央资源分配器动态的划分集群，分配资源给不同的调度框架（scheduler frameworks）。资源可以在不同的调度框架之间以“offers”的形式任意分配，offers 表示了当前可用的资源。资源分配器为了避免不同调度框架对同一资源冲突申请，只允许一次只能分配给一个调度框架。在调度决策的过程中，资源分配器实质上起到了锁的作用。因此 Mesos 中的并发调度是悲观策略的。

Master 使用 resource offer 机制在多个框架之间细粒度的共享资源。每一个 resource offer 空闲资源列表，分布在多个 slave 上。Master 决定给每一个应用框架提供多少资源，依据是公平方法或者优先级方法。第三发可以以可插拔模块的方式定制策略。

Reject 机制：驳回 Mesos 提供的资源方案。为了保持接口的简单性， Mesos 不允许应用框架指定资源需求的限制信息，而是允许应用框架拒绝 Mesos 提供的资源方案。应用框架如果遇到没有满足其需求的资源提供方案，则会拒绝等待。 Mesos 声称拒绝机制可以支持任意复杂的资源限制，同时保持扩展性和简单。

Reject 机制带来的一个问题是在应用框架收到一个满足其需求的方案之前可能需要等待很长时间。由于不知道应用框架的需求， Mesos 可能会把同一个资源方案发给多个应用框架。因此，引入 fliter 机制： Mesos 中的一个调度框架使用 filter 来描述它期望被服务的资源类型(允许应用框架设置一个 filter 表示该应用框架会永远的拒绝某类资源)。因此，它不需要访问整个集群，它只需要访问它被 offer 的节点即可。这种策略带来的缺点是不能支持基于整个集群状态的抢占和策略：一个调度框架不知道分配给其他调度框架的资源。 Mesos 提供了一种资源储存的策略来支持 gang 调度。例如，应用框架可以指定一个其可以运行 node 白名单列表。这不是动态的集群分区吗？ Mesos 进一步解释 filter 机制：filter 只是一个资源分配模型的性能优化方案，应用框架有哪些任务运行在哪些 node 上最终决定权。

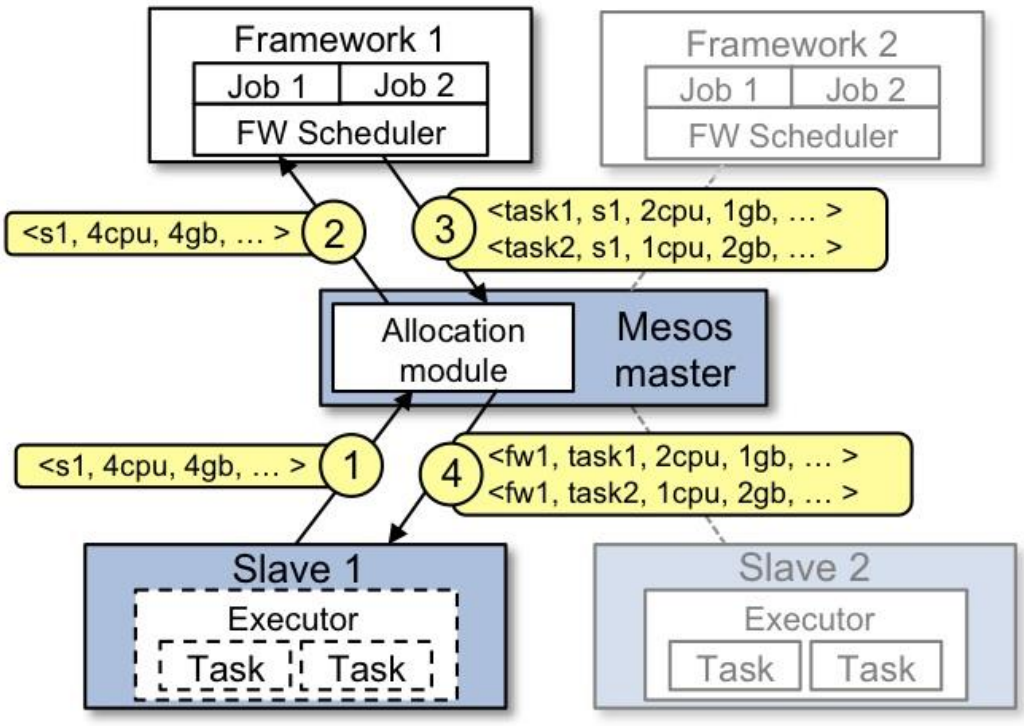

图 10 Mesos 调度流程

Mesos 任务调度过程如图 10 所示，具体流程如下：

1.Slave 1 向 Master 汇报它有 4 个 CPUs 和 4 GB 的空闲内存，Master 的 allocation 模块会根据相应的分配策略通知 framework 1 可以使用所有可用资源。

2. Master 把 slave 1 上的可用资源发送给 framework 1（以 resource offer 的方式）。

3.framework 的调度器响应 Master 调度器：准备在 slave 上运行两个任务，使用的资源分别是：第一个任务<2 CPUs, 1 GB RAM> ，第二个任务 <1 CPUs, 2 GB RAM> 。

4.最后，Master 把任务发送给 slave，然后把相应的资源分配给 framework 的执行器。然后执行器启动两个任务 。由于 slave1 上还有 1 CPU 和 1 GB 的内存没有分配，分配模块可以把资源分配给 framework 2。

另外 Mesos Master 的 Allocation module 是 pluggable。使用 ZooKeeper 来实现 Mesos Master 的 Failover。

Mesos 的 API 如图 11 所示。

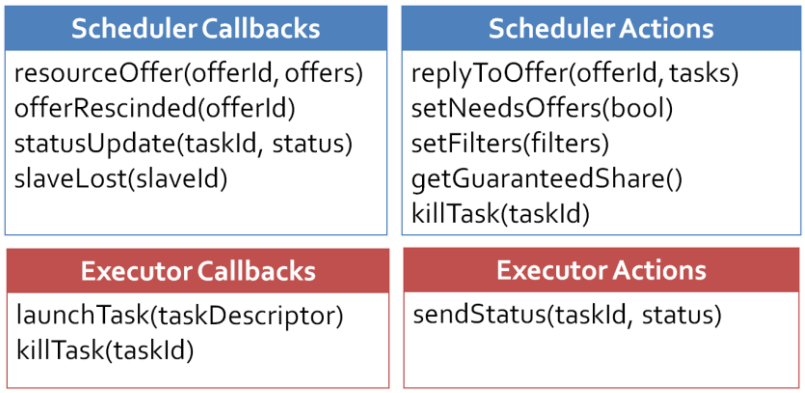

图 11 Mesos 的 API

2.4 状态共享调度（Shared-state scheduling）

在状态共享调度中，每一个调度器都可以访问整个集群状态。当多个调度器同时更新集群状态时使用乐观并发控制。Shared-state 调度可以解决两层调度的两个问题：悲观并发控制所带来的并行限制和调度框架对整个集群资源的可见性。乐观并发控制所带来的问题是当乐观假设不成立时，需要重新调度。

为了克服双层调度器的以上两个缺点（Omega paper 主要关注了这个问题），Google 开发了下一代资源管理系统 Omega，Omega 是一种基于共享状态的调度器，该调度器将双层调度器中的集中式资源调度模块简化成了一些持久化的共享数据（状态）和针对这些数据的验证代码，而这里的“共享数据”实际上就是整个集群的实时资源使用信息。一旦引入共享数据后，共享数据的并发访问方式就成为该系统设计的核心，而 Omega 则采用了传统数据库中基于多版本的并发访问控制方式（也称为“乐观锁”, MVCC, Multi-Version Concurrency Control），这大大提升了 Omega 的并发性。在 Omega 中没有中心的资源分配器，调度器自己做出资源分配的决策。

2.4.1 Omega

宏调度的缺点是难以增加调度策略和专门的实现，并且不能随着集群的扩展而扩展。两层调度确实可以提供灵活性和并行性，但是在实践中他们的资源可见性却是保守的，难以适应一些挑剔型的任务和

一些需要访问整个集群资源的任务。Omega 的解决方案是提出了一个新的并行调度框架：基于共享状态的、无锁的、乐观并发控制，可扩展的。

如图 12 所示，Omega 中没有中心的资源分配器，所有的资源分配决策都是由应用的调度器自己完成的。Omega 维护了一个成为 cell state 的资源分配状态信息主拷贝。每一个应用的调度器都维护了一个本地私有的，频繁更新的 cell state 拷贝，用来做调度决策。调度器可以看到全局的所有资源，并根据权限和优先级来自以为是的要求需要的资源。当调度器决定资源方案时，以原子的方式更新共享的 cell state：大多数时候这样 commit 将会成功（这就是乐观方法）。当冲突发生时，调度决策将会以事务的方式失败。无论调度成功还是失败，调度器都会重新同步本地的 cell state 和共享的 cell state。然后，如果需要，重启调度过程。

Omega 的调度器完全是并行的，不需要等待其他调度器。为了避免冲突造成的饥饿，Omega 调度器使用增量调度——Accept all but the conflict things，这样可以避免资源囤积。如果使用 all or nothing 的策略可以使用 Gang 调度（Either all tasks of a job are scheduled together, or none are, and the scheduler must try to schedule the entire job again.）。Gang 调度要等待所有资源就绪，才 commit 整个任务，就造成了资源囤积。

每一个应用调度器都可以实现自己的调度策略。但是它们必须就资源分配和任务的优先级达成一致。两层调度的中心资源管理器可以轻松实现这一点。这里可以进一步讨论：Google 认为公平性不是一个关键需求，各个调度器只是满足自己的业务需求。因此，限制每一个应用调度器的资源上限和任务提交上限。

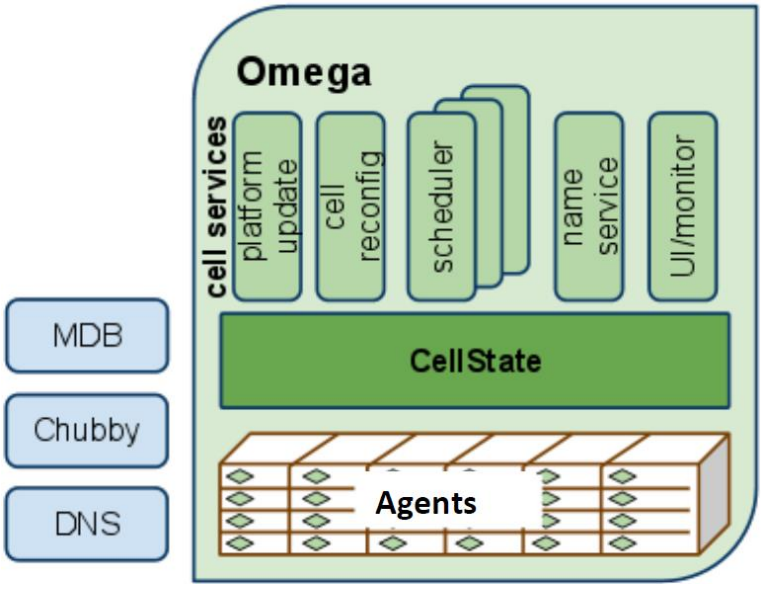

图 12 Omega 调度框架

2.5 对比分析

集群调度的主要目标是提高集群的利用率和使用效率。如图 13 所示为三种调度方式。

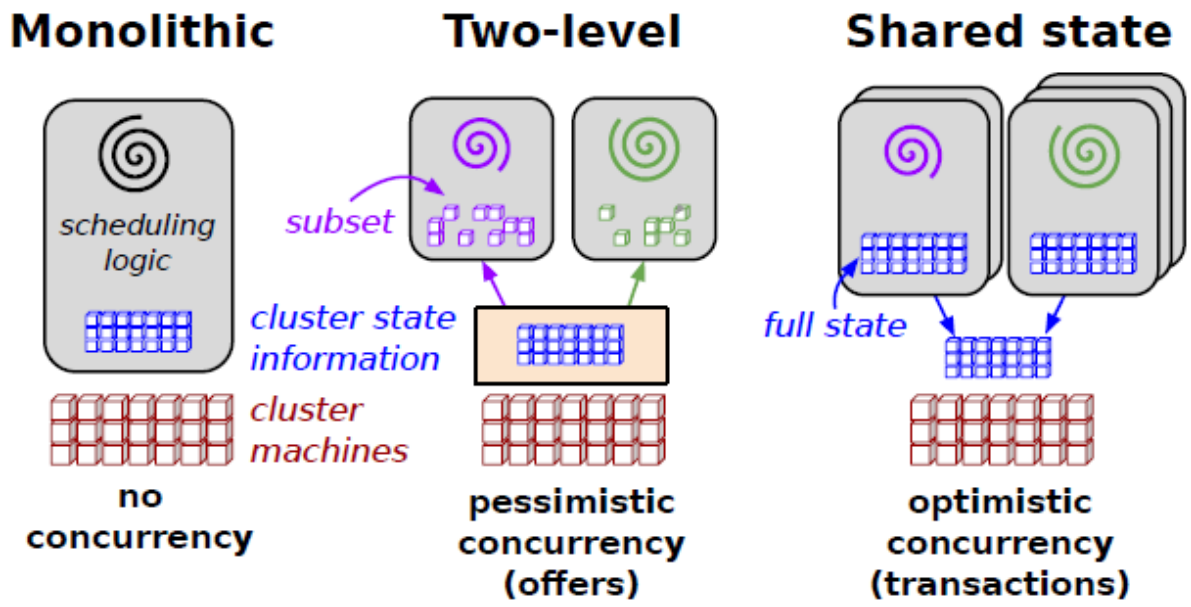

图 13 三种调度方式的对比

宏调度（Monolithic schedulers）为所有任务都使用一个中心调度算法。其缺点是不易增加新的调度策略，也不能随着集群的扩展而扩展。

两层调度（Two-level schedulers）本质上是在调度中分离资源分配和任务分配。使用一个动态资源管理器提供计算资源或者存储资源给多个并行的调度框架。每一个调度框架所拥有的都是一个整个资源的一个子集。为什么是一个动态资源管理器呢？是相对于静态的集群分区来说的。我们可以静态的把集群分为几个区，分别服务于不同的应用。上面的动态资源管理器完成工作就是把静态的分区工作动态化。由于两层调度无法处理难以调度的挑剔任务，且不能根据整个集群的状态做出决策，Google 引入共享状态调度架构。

共享状态调度（Shared state schedulers）使用无锁的乐观并发控制算法。Omega,Google 的下一代调度系统中使用了该架构。那么对比起来，两层调度（Two-level schedulers）本质上是悲观调度算法。

在 Omega 看来， Mesos 的 offer 的机制本质上是一个动态的过滤机制，这样 Mesos Master 向应用框架提供的只是一个资源池的子集。当然可以把这个子集扩大为一个全集，也就是 Share state 的，但其接口依然是悲观策略的。这一点可以讨论。

在 Omega 看来，YARN 中的 Application Masters 提供的仅仅是一个任务管理服务，并不是一个真正的二层调度器。其次，到目前为止，YARN 只支持一种资源类型。另外，尽管 YARN 中的 Application Masters 可以请求一个特定节点的资源，但是其具体策略是不清晰的。

图 14 所示几种调度策略的对比：

Approach	*Resource choice*	*Interference*	*Alloc. granularity*	*Cluster-wide policies*
Monolithic	all available	none (serialized)	global policy	strict priority (preemption)
Statically partitioned	fixed subset	none (partitioned)	per-partition policy	scheduler-dependent
Two-level (Mesos)	dynamic subset	pessimistic	hoarding	strict fairness
Shared-state (Omega)	all available	optimistic	per-scheduler policy	free-for-all, priority preemption

图 14 调度策略的对比（包含静态分区）

3 下一步工作

集群调度技术仍在发展之中，OSDI 16 将会发布一些最新的关于调度的文章，包括 Google 的 Rapid: Fast, Centralized Cluster Scheduling at Scale，后面会支持关注。

资源感知调度。利用机器学习从历史负载变化中预测资源需求模型，为调度决策提供依据。

4 参考文献

1、 Omega: flexible, scalable schedulers for large compute clusters.

2、 Mesos A Platform for Fine-Grained Resource Sharing in the Data Center.

3、 Apache Hadoop YARN: Yet Another Resource Negotiator.

4、 Multi-agent Cluster Scheduling for Scalability and Flexibility.

5、http://zhangjunhd.github.io/2013/08/16/multi-agent-cluster-scheduling.html

6、http://www.firmament.io/blog/scheduler-architectures.html

7、http://zh.linuxvirtualserver.org/node/29.

8、
http://www.austintek.com/LVS/meetings/Software_Freedom_Day_UNC_2007/Software_Freedom_Day_UNC.2007.09.html.

9、http://docs.huihoo.com/p2p/1/

10、http://www.admin-magazine.com/HPC/Articles/The-New-Hadoop

11、http:// Mesos.apache.org/documentation/latest/index.html

12、https://github.com/apache/ Mesos/tree/Master

13、http:// Mesos.apache.org/downloads/

14、http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

15、
https://confluence.inside.nsn.com/display/DataManagement/Policy+based+resource+management+based+on+YARN

16、http:// Mesos.apache.org/documentation/latest/ Mesos-presentations/

17、http:// Mesos.apache.org/documentation/latest/ Mesos-architecture/

18、http:// Mesos.apache.org/

19、http://dongxicheng.org/apache- Mesos/meso-architecture/

20、http://dongxicheng.org/mapreduce-nextgen/borg-yarn- Mesos-torca-corona/

21、http://djt.qq.com/bbs/thread-29998-1-1.html

22、http://www.tuicool.com/articles/NF3QJfQ

23、http://emuch.net/html/201310/6447521.html