学校代号：10532
学　　号：S131020005
密　　级：普通

湖南大学硕士学位论文

Hadoop平台中基于预释放资源列表的任务调度算法研究

学位申请人姓名：陈 京
导师姓名及职称：李智勇教授
培 养 单 位：信息科学与工程学院
专 业 名 称：计算机科学与技术
论文提交日期：2016年5月17日
论文答辩日期：2016年5月29日
答辩委员会主席：徐成教授

Research on Task Scheduling Algorithms Based on Pre-Release Resource List in Hadoop

by

CHEN Jing

B.E. (Hunan University of Science and Technology) 2013

A thesis submitted in partial satisfaction of the

Requirements for the degree of

Master of Engineering

in

Computer Science and Technology

in the

Graduate School

of

Hunan University

Supervisor

Professor LI Zhiyong

May, 2016

湖 南 大 学

学位论文原创性声明

本人郑重声明：所呈交的论文是本人在导师的指导下独立进行研究所取得的研究成果。除了文中特别加以标注引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写的成果作品。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律后果由本人承担。

作者签名：陈京 日期：2016 年 6 月 1 日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权湖南大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本学位论文属于

1、保密□，在____年解密后适用本授权书。

2、不保密☑。

(请在以上相应方框内打“✓”)

作者签名：陈京 日期：2016 年 6 月 1 日

导师签名：李绍勇 日期：2016 年 6 月 1 日

摘 要

互联网技术催生了大数据时代的来临。当前，大数据已经成为了炙手可热的研究焦点。由于海量的数据使得单个的计算机已经无法满足存储及计算的要求，各种大数据的计算模式及其对应的分布式计算系统开始不断涌现。MapReduce无疑是其中最为经典并且应用最为广泛的大数据计算模式。任务调度与资源分配一直以来都是大规模分布式集群研究的关键技术，这对提高大数据集群的计算效率尤其重要。

Hadoop是通过各个从节点在不同的时间向主节点以“pull”的方式发送心跳请求获取任务的。现有的任务调度器只根据当前请求任务的从节点状况，来选择任务进行分配。而没有将更多的资源与集群中各个作业的具体需求联系起来，做出更优的调度方案。因此本文在对MapReduce的任务推测执行算法和任务调度算法进行分析和研究的基础上，设计了相关的改进算法。具体工作如下：

（1）提出了预释放资源列表。通过Hadoop记录的历史信息和集群当前状况的监控信息，对节点的任务处理速率以及任务的剩余完成时间进行评估，可以预测出哪些资源即将释放，以此构建出预释放资源列表。从而使得Hadoop在进行任务调度的时候有着更大的优化空间。

（2）设计了一种基于预释放资源列表的任务推测执行算法。从构建出来的预释放资源列表中，选择使慢任务更快完成的资源，从而达到使慢任务更快完成的目的。实验表明，基于预释放资源列表的推测执行算法能够有效的使有慢任务的作业更快完成。

（3）设计了一种基于预释放资源列表的任务调度算法。根据作业在各个节点的本地性以及节点的任务处理速率，为每个作业构建一个预释放资源列表。通过将作业与预释放资源列表进行匹配调度，从而为每个作业找到最适合的资源。实验表明，本算法能有效提高集群的性能。

关键词：任务调度；任务推测执行；MapReduce；Hadoop；大数据

Abstract

Big Data era has come because of the development of Internet technology. Currently, Big Data has become one of the most important research issues. In the case of the mass of data, a single computer is unable to meet the requirements of storage and computing. A variety of big data computing model and distributed computing systems began to emerge. MapReduce is one of the most classic and most popular big data computing model. Task scheduling and resource allocation have always been a key technologies of large-scale distributed clusters, which is especially important to improve the efficiency of large-scale distributed clusters.

The method of hadoop assigns task is each slave send a heartbeat request to the master. Existing task scheduler only based on the current task request from the slave to select task assignment. Without more resources to the real needs of jobs linked to make better scheduling scheme. Therefore, this paper analysis and research MapReduce speculative execution algorithm and task scheduling algorighm, and put forward relevant improvement algorithms. Details are as follows:

(1) Proposed pre-release resource list. According to Job history information and current status of Hadoop cluster, which can evaluate task processing rate of node and task remaining time. Then build a pre-release resource list, making task sheduling of hadoop has more room for optimization.

(2) Design a task speculative execution algorithm based on pre-release resource list. From the pre-release resource list, select a faster resource to make the slow task done faster. Experiments show that task speculative execution algorithm based on pre-release resource list can effectively make the presence of the slow task of the job done faster.

(3) Design a task scheduling algorithm based on pre-release resource list. Based on the task processing rate of job and the data locality on each node to build a pre-release resource list for each job. By mathing the jobs and pre-realease resource list to find the most suitable resource for each job. Experiments show that the algorithm can effectively improve the performance of the cluster.

Key word: Task Scheduling; Speculative Execution; Big Data; MapReduce; Hadoop;

目 录

插图索引

附表索引

第1章 绪 论

1.1 研究背景与意义

当前，大数据已经成为移动互联网、大规模物联网、分布式集群计算、数据科学与机器学习等领域炙手可热的研究焦点。据2014年5月美国白宫发布的“大数据”白皮书统计，近年来美国产业界推进对大数据进行获取、组织与分析，以发现有效信息的工具与相关技术发展的力度迅猛增长[1]。2015年12月中国大数据技术大会讨论中，“大数据基础设施”已与“深度学习、工业与制造大数据、金融大数据”等研究主题成为关注热点，大数据研究已经从数据本身越来越走向行业与技术纵深[2]。Gartner 2015新兴技术发展周期简评分析，也表明在“大数据实用化”与“大数据机器学习”技术趋势迅速崛起的同时，“大数据处理的软硬件基础设施”已成为业内关注重点[3]。

目前，大数据分析技术已经在互联网、商业智能、网络与信息安全、医疗服务、智能交通、生物信息等行业广泛应用，并产生了巨大的社会价值和产业影响。美国俄亥俄州运输部(ODOT)早在2011年就利用INTRIX的云计算分析处理大数据来了解和处理恶劣天气的道路状况，减少了冬季连环撞车发生的概率[4]；微软研究院于2013年左右提出来城市计算的理念，其通过收集交通、天气、空气质量等数据，进行多源数据融合的大数据挖掘，以服务于改进城市规划、交通拥堵、空气污染等城市发展问题[5]；2016年3月份，AI领域最大新闻就是Google公司DeepMind小组研究的AlphaGo战胜世界围棋大师李世石，其关键技术之一就是采用了深度学习技术进行棋谱大数据分析与学习[6]。目前，数据智能已成为产业界、学术界的研究与关注热点，而支持大数据分析的大数据处理系统是其重要的技术基础之一，随着大数据分析技术的发展，各种大数据的计算模式与其对应的集群计算系统不断涌现。

MapReduce最早是由Google公司提出的一种面向大规模数据处理的分布式计算模型和方法。Google公司设计MapReduce的初衷主要是为了解决其搜索引擎中大规模网页数据的并行化处理。Google公司发明了MapReduce之后首先用其重新改写了其搜索引擎中的Web文档索引处理系统。但由于MapReduce可以普遍应用于很多大规模数据的计算问题，因此自发明MapReduce以后，Google公司内部进一步将其广泛应用于很多大规模数据处理问题。自2003年以来，Google公司在国际会议上陆续发表了三篇分布式计算的著名论文，公布了Google的GFS[7]和MapReduce[8]以及BigTable[9]的基本原理和主要设计思想。2004年，开源项目

Lucene（搜索索引程序库）和Nutch（搜索引擎）的创始人Doug Cutting发现MapReduce正是其所需要的解决大规模Web数据处理的重要技术，因而模仿Google MapReduce，基于Java设计开发了一个称为Hadoop的开源MapReduce并行计算框架和系统。Hadoop的分布式文件系统HDFS则是模仿了Google的GFS。Hadoop的子项目HBase[10]，则提供了类似于BigTable的能力。自此，Hadoop成为Apache开源组织下最重要的项目之一[11]，自其推出后很快得到了全球学术界和工业界的普遍关注，并得到推广和普及应用[12-14]。

任务调度与资源分配一直以来都是大规模分布式集群研究的关键技术，这对提高大数据处理系统的计算效率尤其重要[15-17]。大数据处理系统中不合理的数据存储[18-20]、任务调度与资源配置可能造成频繁的数据迁移[21]、任务同步等待以及通信开销[22]，是目前大数据集群计算系统亟待解决的重要问题之一。据美国CMU的邢波教授研究小组统计，目前一般大数据机器学习平台80%左右（比如Hadoop的通讯时间占到90%甚至更高）的CPU计算时间处于任务同步等待与数据通信[23]。通信开销主要是由于任务的数据与计算不在本地、通信强关联任务部署与任务同步调度不合理引起。在任务调度与资源配置优化中合理考虑数据计算本地性约束、通信强关联约束与任务同步约束，可大大降低系统因数据迁移、模型通信与任务同步造成的通信开销与同步等待。如何实现该优化机制是目前分布式计算系统主要难题之一。

MapReduce无疑是最为经典的大数据计算模式，Apache Hadoop则是MapReduce的开源实现。近年来，尽管各种大数据的计算模式与其对应的分布式计算系统开始不断涌现，Hadoop出现了众多的竞争对手，例如目前非常流行的Apache Spark[24]和Apache Flink[25]等分布式内存计算系统。但Hadoop凭借着成熟稳定的系统，以及Hadoop丰富的生态圈，仍然是众多企业和机构首选的分布式计算系统。甚至Spark等新兴的大数据计算模式，也都支持运行在Apache Hadoop YARN[26]上。其中，YARN是新一代的Hadoop资源管理器，Spark运行在YARN平台上的技术叫做Spark on YARN[27]。

因此，基于Hadoop计算平台进行任务调度算法的研究，并提出性能更好的任务调度算法具有重要的意义。

1.2 国内外研究现状

1.2.1 任务推测执行算法的国内外研究现状

在分布式集群环境下，因为任务负载不均或者机器处理性能的差异等原因，会造成任务之间处理速率不一致[28-29]。有些任务的进度会明显比其他任务慢，比如一个作业的某个任务进度只有30%，而其他任务却已经执行完毕，则该任务会

拖慢作业的整体完成进度。为了处理这种情况，Hadoop采用了任务推测执行机制。它利用集群监控信息，推测出“拖后腿”的任务，并为这样的任务启动一个备份任务，让该备份任务与原始任务同时处理同一份数据，并最终选用最先成功运行完成任务的计算结果作为最终结果。

任务推测执行机制实际上采用了经典的算法优化方法：以空间换时间。它同时启动多个相同任务处理相同的数据，并让这些任务竞争以缩短数据处理时间。显然，这种方法需要占用更多的计算资源。在集群资源紧缺的情况下，应合理使用该机制，争取在使用少量资源的情况下，减少作业的计算时间。

目前国内外学者针对 Hadoop 的任务推测执行算法做出了一些研究。下面分别对任务推测执行算法的国外研究现状以及国内研究现状进行介绍。

1.2.1.1 任务推测执行算法的国外研究现状

MapReduce提出的早期，集群的规模还并不是很大，集群间机器的异构性差异也较小，提交到集群中处理的作业类型也比较单一。所以，这个阶段Hadoop中采用的主要是适用于同构环境的任务推测执行算法。Google公司在2004年发表的关于MapReduce的论文中提出了一种简单的任务推测执行策略[8]。当map阶段或者reduce阶段快要完成的时候，将仍在运行的任务在所有可用的Slot上备份执行。当原任务和备份任务的其中一个执行完成了，就保存该任务的输出作为最终的结果，同时杀死另外一个任务。虽然该方法非常简单和直接，但Google发现这可以明显减少作业的完成时间。例如，在一个排序程序中，采用推测执行机制完成任务的时间比不采用推测执行的时间减少了44%。但是显然这种不加判断，备份执行所有正在运行的任务，会造成集群中资源的大量浪费。微软公司的分布式集群系统——Dryad[30]，在早期的时候也存在着同样的问题。Apache Hadoop最初使用的推测执行策略则对正在运行任务的快慢进行了判断，只为那些慢任务启动备份任务。但该算法只是简单的设定一个阈值，当一个任务的进度落后于平均任务进度达到这个阈值（设定为20%）时，就判定为慢任务。但该版本实现的推测执行策略缺乏保证备份任务执行速度的机制。如果新启动的备份任务不能比原任务更快完成，则备份任务占用的资源就被浪费了。并且也没有对节点的处理速度加以判断，所以该算法并不适用于异构环境。

随着MapReduce的逐渐流行和功能的逐渐强大，用户开始将越来越多的任务提交到MapReduce集群中处理。为了保证任务处理速率，集群中开始不断加入新的机器。集群内的机器异构性以及作业的异构性开始变得越明显。原来的推测执行算法已经不能满足异构环境的需要。伯克利大学的Zaharia M针对Hadoop的任务推测执行算法中存在的不足，提出了适用于异构环境的LATE（Longest Approximate Time to End）[31]算法。LATE算法对任务的快慢和节点速度的快慢都

进行了考虑。但LATE算法同样也还存在一些不足，由于LATE算法依然采用了静态方式计算任务的进度，可能导致性能仍然比较低下。Mantri[32]算法不同于之前的任务推测执行算法，之前的算法都是在已经没有未运行的任务的时候才将资源用于推测执行。而Mantri算法在集群比较忙的时候，如果检测到某个任务有非常高的可能性会是慢任务，就直接杀死该任务；如果集群有多余的空闲资源，则为该任务启动一个备份任务。MapReduce2.0采用了一种新的任务推测执行机制[33]。该机制认为只有当备份任务能够比原任务更快完成，那么启动该备份任务才有意义。否则只会造成资源浪费。并且认为备份任务比原任务越快完成越好。所以它总是选择备份任务完成时间与原任务完成时间差值最大的慢任务进行备份执行。新加坡的学者提出了SPEB[34]推测执行算法，该算法认为任务推测执行可以减少单一作业的完成时间，但是却以影响集群的整体性能为代价。所以提出了SPEB来扮演单一作业的完成时间和集群性能之间的效率平衡器的角色。

1.2.1.2 任务推测执行算法的国内研究现状

Quan Chen 等人在 LATE 算法基础上进行了改进，提出了 SAMR[35]（Self-Adaptive MapReduce Scheduling Algorithm）算法。该算法通过历史信息调整各个参数以提高估算任务进度和任务剩余时间的准确性，同时分别针对 Map Task 和 Reduce Task 将节点分成快节点和慢节点。但是该算法没有考虑任务负载不均的情况，所以当以任务进度增长率判断一个节点快慢的时候，同样会造成误判。杨立身等人在 SAMR 算法基础上进行了改进，提出了一种增强的自适应 MapReduce 调度算法[36]。该算法采用 K-means 聚类算法[37]动态地调整任务各阶段的进度值，以找到真正的慢任务。李丽英等人在 LATE 算法的基础上，提出了一种基于数据局部性的改进算法[38]。该算法解决了数据局部性导致的慢任务备份执行时，需要占用大部分时间而影响其处理速率的问题。Guo Z 等人提出一种效益感知的任务推测执行算法[39]。通过评估备份任务的成功率，来减少不必要的备份执行。北京大学陈琪、肖臻等人提出了 MCP[40]算法，该算法使用任务进度增长率和处理带宽来选择慢任务，然后使用指数加权移位平均法来预测处理速度和计算任务的剩余时间，最后基于集群的负载使用成本-利益模型来决定备份执行哪个任务。

然而目前的任务推测执行算法都是在当前发送心跳请求的资源上考虑任务推测执行。并没有充分利用 Hadoop 记录的历史信息和集群当前状况监控信息，找到能使慢任务更快完成的资源。

1.2.2 MapReduce 任务调度算法的国内外研究现状

在一个分布式的集群环境中，任务调度器的好坏直接影响了集群的性能。

Hadoop 最初的设计目的是支持大数据批处理作业，如日志挖掘、Web 索引等作业，为此，Hadoop 仅提供了一个非常简单的调度机制：FIFO，即先来先服务。在该调度机制下，所有作业被统一提交到一个队列中，Hadoop 按照提交顺序依次运行这些作业。但随着 Hadoop 的普及，单个 Hadoop 集群的用户量越来越大，不同用户提交的作业往往具有不同的服务质量要求（Quality Of Service，QoS），CPU 密集型作业、I/O 密集型作业等不同类型的作业对硬件的要求也是不同的。因此，简单的 FIFO 调度算法很难满足不同作业的多样化需求，同时也难以对硬件资源进行充分的利用。另外，FIFO 还会导致饥饿现象，即前面的大作业占据了所有的资源，导致后面的作业长时间得不到资源的现象。

为了克服单队列 FIFO 调度器的不足，多用户多队列调度器诞生了。这种调度器允许管理员按照应用需求对用户或者应用程序分组，并为不同的分组分配不同的资源量，同时通过添加各种约束防止单个用户或者应用程序独占资源，进而能够满足各种 QoS 需求。下面将介绍国内外学者在 Hadoop 的任务调度算法的一些研究成果。

1.2.2.1 MapReduce 任务调度算法的国外研究现状

多队列调度器的典型代表是Yahoo!的计算能力调度器[41]（Capacity Scheduler）和Facebook的公平调度器[42]（Fair Scheduler）。公平调度器和计算能力调度器都通过队列来对资源进行划分。通过为每个队列指定一定比例的资源，保证了每个队列都能得到指定比例的资源，避免了出现单队列调度器占用过多资源的问题。同时，当某个队列的资源有剩余时，可以暂借给别的队列，有效避免了资源的闲置浪费。当队列需要资源时，又可以将借出去的资源抢占回来，保证了队列可以得到它应有的资源量。随着Hadoop版本的演化，公平调度器和计算能力调度器支持的功能都越来越健全，很多功能在两种调度器中都能找到。所以可以发现，两个调度器正变得越来越相似。另外，这两种调度器都采用了延迟调度算法[43]，来提高作业的本地性。

除了这些在Apache Hadoop中实现了的调度器，还存在一些其他的调度器。Gregory A等人提出了一种基于作业截止时间约束的调度算法[44]。该算法的目标就是尽量满足用户期望的作业运行完成时间。调度器通过分析各个作业在集群上已完成任务的历史信息，估算出各个作业在分配一定资源的情况下的剩余时间。然后根据作业的期望完成时间，动态地调整给每个作业分配的资源数目，尽量使每个作业都能在期望的时间内完成。还有一些专门针对Reduce任务进行优化的调度器[45-47]。这些文章认为之前的调度器在进行任务调度的时候都只考虑了Map任务的本地性，而没有考虑Reduce任务其实也存在本地性。这类调度器通过将Reduce任务的机架本地性和节点本地性进行量化，然后把Reduce任务调度到本地性最大

的位置，尽可能地降低了Reduce任务的通信开销。Mashayekhy L等人从能耗的角度进行考虑，提出了一种能耗感知的任务调度算法[48]。该调度算法在满足SLA（Service-Level Agreement）的条件下，尽可能地降低MapReduce集群的能耗。自学习调度器（Learning Scheduler）[49]是一种基于贝叶斯分类算法的资源感知调度器。该调度器选取了若干个作业特征作为作业分类属性，通过用户标注好的一些作业可训练得到一个分类器。从而判断作业在某个从节点上运行的好坏。动态优先级调度器[50-52]允许用户动态调整自己获取的资源量以满足其服务质量要求。动态优先级调度器的核心思想是在一定的预算约束下，根据用户提供的消费率按比例分配资源。相比于其他调度器，动态优先级调度器允许用户根据需要（比如完成时间）动态调整资源，进而可以对作业运行质量进行精细地控制。

1.2.2.2 MapReduce 任务调度算法的国内研究现状

李婴等人提出了一种基于作业类别和截止时间的任务调度算法[53]。该算法将作业分为 CPU 密集型作业和 I/O 密集型作业，并根据作业的截止时间来设置优先级，实现作业的调度。宋杰等人提出了一种优化 MapReduce 系统能耗的任务分发算法[54]。该算法通过动态调整任务大小，来保证任务的并行性，降低 MapReduce 系统的能耗。郑晓薇等人提出了一种基于节点能力的任务自适应调度分配方法[55]。该算法在计算性能较高的节点上，分配较多的任务和数据量。而对于计算性能低的的节点，相应地减轻任务和数据量负载。并且根据节点历史和当前的负载状态，以节点性能、任务特征、节点失效率等作为节点任务量调度分配的依据，并使各节点能自适应地对运行的任务量进行调整。从节点负载更均衡的角度提升集群的总任务完成时间。

但目前这些任务调度器，只是简单的把提交任务请求的空闲资源分配给调度器指定的作业，而没有充分考虑当前集群中其他资源的情况。进而从整体上做出更优的调度方案。

1.3 研究内容与主要工作

本文深入分析了Hadoop平台的基本架构与实现原理，重点研究了Hadoop的资源管理模块中的任务推测执行机制与任务调度算法。本文的研究工作主要包括如下内容：

（1）提出了预释放资源列表。目前Hadoop任务调度器只考虑当前接收到的某个从节点的心跳请求，就做出任务调度的决策。由于任务调度器没有获取更多的从节点信息，从而无法做出更优的任务调度方案。本文通过Hadoop记录的历史信息和对集群当前状况监控信息，构建出预释放资源列表。从而使得Hadoop在进

行任务调度和任务推测执行的时候有着更大的优化空间。

（2）设计了基于预释放资源列表的任务推测执行算法。现有的推测执行算法都是直接将最慢的任务分配给当前申请任务的空闲资源，而本文的算法不局限在当前申请任务的资源上考虑推测执行，通过构建预释放资源列表，在这些短时间内即将释放的资源上考虑推测执行。从而能找到使慢任务更快完成的资源。

（3）设计了基于预释放资源列表的任务调度器。当出现一个空闲资源的时候，现有的任务调度器都是基于自身的调度原则，挑选出哪个作业中的哪个任务应该得到这个空闲资源，然后就直接将这个空闲资源分配给这个任务。而本文的调度器使用了预释放资源列表，通过资源列表与任务列表的匹配调优，可以使得集群中各个作业获得更适合自身的资源。从而提高集群的整体性能。

1.4 本文组织结构

本文的内容组织结构具体如下：

第一章：绪论。这一章介绍了本文的研究背景及其意义，主要概述了Hadoop任务推测执行算法和任务调度算法的研究现状。接着介绍了本文的研究内容和主要工作，并对文章的内容组织结构进行了说明。

第二章：Hadoop基本架构、Hadoop集群状态监控与资源管理、Hadoop推测执行算法及任务调度算法介绍。该章节对Hadoop的两大基本模块进行了介绍，包括MapReduce计算模块和HDFS存储模块。然后介绍了JobTracker是如何进行状态监控与资源管理的。接着介绍了Hadoop的任务推测执行算法，包括Hadoop-1.0.0的任务推测执行算法和LATE算法。最后，介绍了Hadoop的经典调度器，包括FIFO调度器和公平调度器。详细介绍了公平调度器的资源三级调度模型和延迟调度算法。

第三章：基于预释放资源列表的任务推测执行算法。本章首先分析了现有推测执行算法的一些问题，针对这些问题设计了基于预释放资源列表的任务推测执行算法。然后对该算法进行了详细介绍，包括如何一步步构建慢任务列表和预释放资源列表，以及整个算法的详细步骤。最后，通过实验将该算法与原有算法进行了对比，并对实验结果进行了总结。

第四章：基于预释放资源列表的任务调度器。本章首先分析了现有任务调度器的一些问题，针对这些问题设计了基于预释放资源列表的任务调度器。然后对该调度器进行了详细介绍。最后，通过实验将该调度器与现有的调度器进行了对比，并对实验结果进行了总结。

最后，对本文所做的主要工作进行了总结，并对本文的进一步研究进行了展望和设想。

第2章 相关研究

MapReduce是一个分布式计算框架，MapReduce的编程模型为用户提供了非常易用的编程接口，编程人员只需要像编写串行程序一样实现几个简单的函数即可实现一个分布式程序。而其他比较复杂的工作，如节点间的通信、容错备份、资源调度等，全部由Hadoop内部来完成。在本章中，将对跟本文研究相关的Hadoop内部的一些实现细节进行介绍。

2.1 Hadoop 基本架构

掌握Hadoop的基本架构是深入理解Hadoop的任务调度器的基础。因此，接下来先对Hadoop的基本架构进行介绍。Hadoop由两部分组成，分别是分布式文件系统HDFS和分布式计算框架MapReduce。其中，分布式文件系统主要用于大规模数据的分布式存储，而MapReduce则是对存储在分布式文件系统中的数据进行分布式计算。

2.1.1 HDFS 架构

HDFS是一个具有高度容错性的分布式文件系统，适合部署在廉价的机器上。HDFS能提供高吞吐量的数据访问，非常适合大规模数据集上的应用。

HDFS的架构如图2.1所示，总体上采用了master/slave架构，主要由以下几个组件组成：Client、NameNode、SecondaryNameNode和DataNode。下面分别就这几个组件进行介绍[56]。

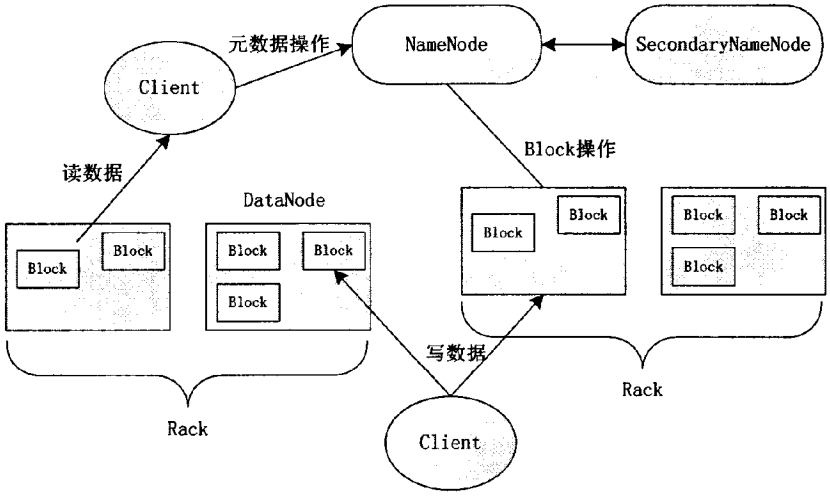


图 2.1 HDFS 架构图

（1）Client

Client提供了一个供用户调用的文件系统接口，用户通过该接口可以对HDFS文件系统进行操作，并隐藏了底层与NameNode和DataNode的交互过程。

（2）NameNode

HDFS的主节点。一个HDFS文件系统通常只有一个NameNode。它的功能是负责管理HDFS文件元数据信息以及监控所有从节点的健康状态。当需要访问HDFS上的文件时，就需要通过NameNode保存的元数据信息查找文件所在的位置。并且如果某个DataNode宕掉，NameNode可以知道这个DataNode上保存的是什么文件，从而在别的DataNode上重新备份这些文件。

（3）Secondary NameNode

Secondary NameNode其实并不是NameNode的热备份。NameNode宕掉后，Secondary NameNode并不能接替NameNode的职责使集群继续正常运行。它只是定期合并fsimage（HDFS元数据镜像文件）和edits日志（HDFS文件改动日志），并传送给NameNode。起到了为NameNode减小工作压力的作用。

（4）DataNode

HDFS的Slave节点。一个HDFS系统可以有很多个DataNode。通常每个Slave节点上运行一个DataNode进程，HDFS上的数据就实际存储于DataNode节点上。DataNode进行读写操作的最小单元是block，一般将block大小设为64MB或者128MB。当用户上传一个大文件到HDFS上时，该文件会以block为单位，切分成若干份，分别存储到不同的DataNode；同时，为了保证数据不丢失，会将同一个block备份若干份（默认是3，该参数可配置）到不同的DataNode上。这样，即使一个DataNode宕掉了，仍然可以通过别的DataNode上的备份还原丢失的数据。

2.1.2 MapReduce 架构

Hadoop MapReduce采用了Master/Slave架构，具体如图2.2所示。它主要由以下几个组件组成：Client、JobTracker、TaskTracker和Task。下面分别对这几个组件进行介绍[57]。

（1）Client

用户通过Client端将编写好的MapReduce程序提交到JobTracker端；用户还可以通过Client提供的一些接口对作业运行状态进行查看。

（2）JobTracker

JobTracker就是MapReduce架构的主节点。一个Hadoop集群中只有一个JobTracker进程。JobTracker主要功能是监控资源和作业调度。JobTracker监控所有TaskTracker的健康状况，一旦发现某个TaskTracker失败后，就会在别的TaskTracker上重新执行原来分配在该TaskTracker上的任务；同时，JobTracker还会监控任务的

执行进度、资源使用量等信息。任务调度器可以根据这些信息来更加合理地进行任务调度。

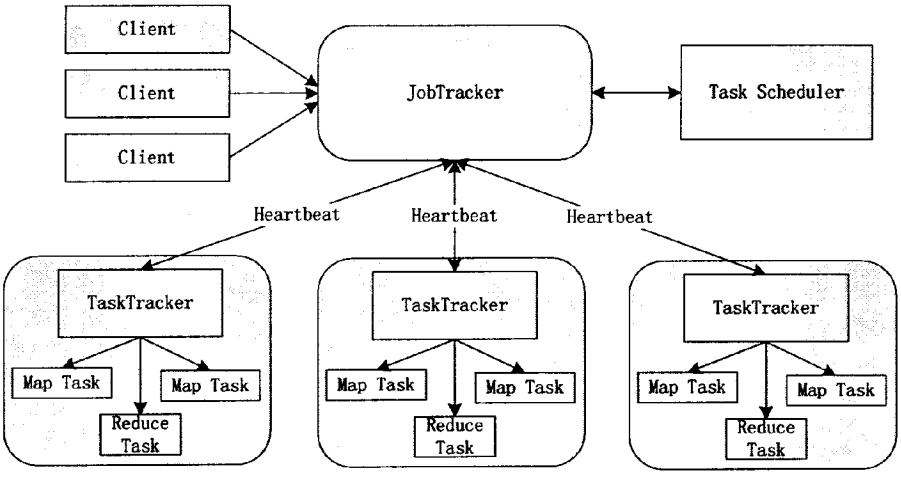


图 2.2 MapReduce 架构图

（3）TaskTracker

TaskTracker是MapReduce架构中的从节点。TaskTracker可以有许多个，甚至一台机器上可以设置多个TaskTracker。TaskTracker会周期性地向JobTracker发送心跳信息，将本节点上资源的使用情况和任务的运行进度汇报给JobTracker。JobTracker则根据TaskTracker的情况，向TaskTracker发送启动新任务、杀死任务等命令。TaskTracker通过“Slot”打包该节点上的资源。每个TaskTracker都可以设置该节点上一个Slot包含多少CPU和内存。Task只有获取Slot后才可以运行，而Hadoop调度器的作用就是将Task分配到合适的TaskTracker上的空闲Slot上执行。通过设置一个TaskTracker上的Slot数目，可以限定该TaskTracker运行Task的并发度。

（4）Task

一般一个MapReduce程序就对应一个作业，每个作业会被分解成若干个Map/Reduce任务（Task）。任务调度器将Task分配给指定的TaskTracker后，TaskTracker就负责启动和运行Task。MapReduce的处理单位是split。即每个Task处理的数据大小为一个split。因此split的多少决定了Map Task数目。Map Task执行过程如图2.3所示。Map Task先将对应的split迭代解析成一个个<key,value>对，然后将<key,value>对作为用户编写的map()函数的输入，map()函数执行完后会产生一个个新的<key,value>对，这些<key,value>对会根据key值来被划分成若干个partition（partition的个数由用户自己设定）。每个partition将被一个Reduce Task处理。因此，map任务产生的partition的数量决定了reduce任务的个数。

Reduce Task执行过程如图2.4所示.该过程分为三个阶段：（1）Shuffle阶段：从远程节点上拷贝Map Task产生的对应的partition；（2）Sort阶段：按照key值对partition中的<key,value>对进行排序；（3）Reduce阶段：将sort的结果迭代解析成

一个个的<key, value list>，然后把<key,value list>作为用户编写的reduce()函数的输入处理，将reduce()函数的最终结果保存到HDFS上。

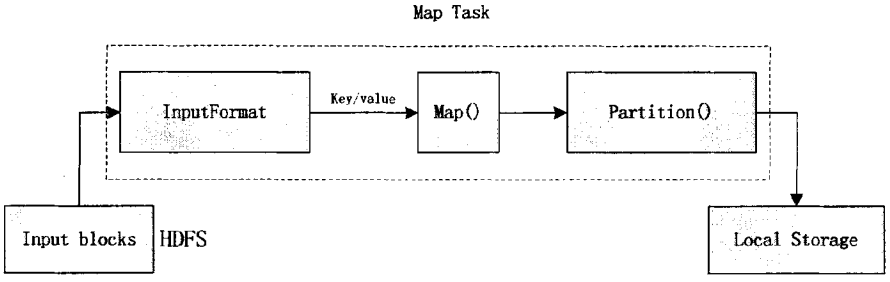


图 2.3 Map Task 执行过程

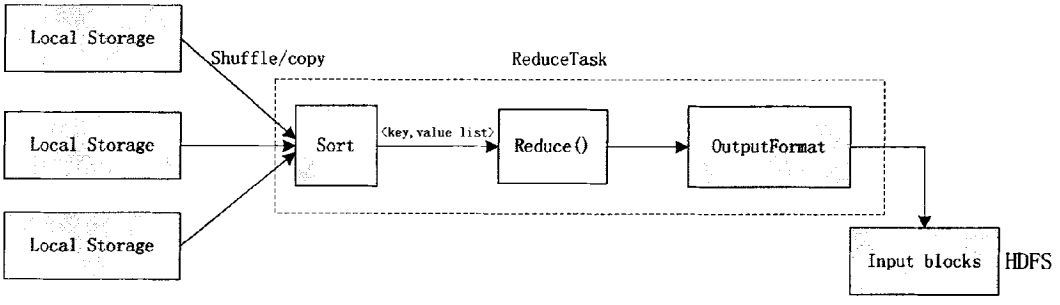


图 2.4 Reduce Task 执行过程

2.1.2.1 MapReduce 作业的生命周期

下面从一个MapReduce作业从提交到运行完成的生命周期，来了解MapReduce架构中的各个组件是如何相互协作的。用户提交到Hadoop系统中运行的MapReduce应用程序，就是一个作业（Job）。一个作业从提交到开始执行的过程如图2.5所示，整个过程需要以下步骤。

步骤1：用户通过Client端提交作业到JobTracker端；

步骤2：JobTracker通过任务调度器对作业执行初始化操作；

步骤3：某个TaskTracker发送心跳信息给JobTracker，向Jobracker汇报当前空闲的Slot数和任务的运行情况；

步骤4：如果该TaskTracker存在空闲的Slot，则JobTracker通过任务调度器来为该TaskTracker选择任务；

步骤5：任务调度器按照一定的调度策略为该TaskTracker选择最合适的任务列表，然后返回该列表给JobTracker；

步骤6：JobTracker以心跳请求响应的形式将任务列表返回给对应的TaskTracker；

步骤7：TaskTracker收到心跳应答后，就会着手处理这些任务。TaskTracker会周期性地向JobTracker汇报每个Task的完成进度。

步骤8：待所有Task执行完成，整个作业执行成功。

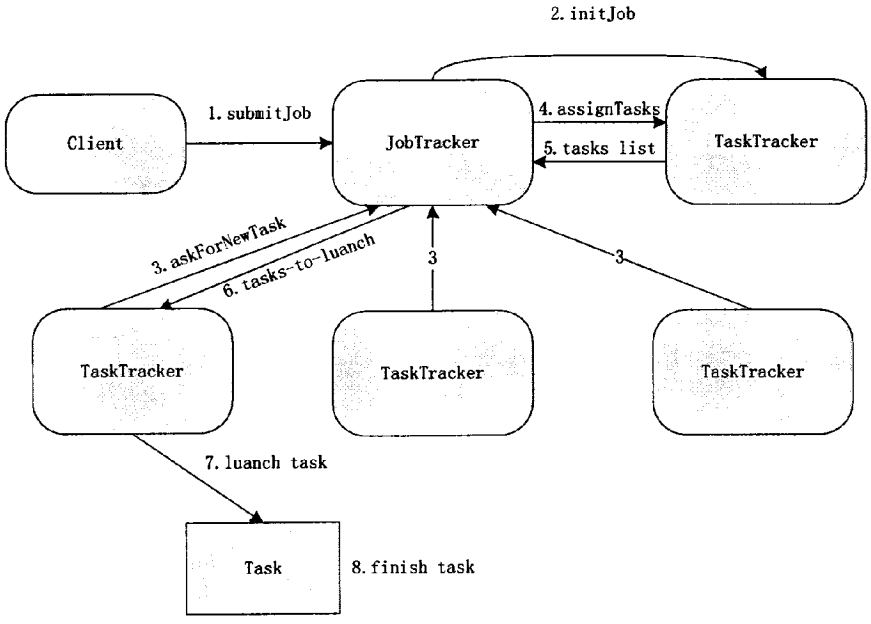


图 2.5 MapReduce 作业的生命周期

2.2 Hadoop 集群状态监控与资源管理

Hadoop集群的状态监控与资源管理主要由JobTracker这个组件负责。Hadoop MapReduce是一个Master/Slave架构。JobTracker便是该架构中的Master。JobTracker是整个集群中唯一的全局“管理者”，涉及的功能包括作业管理、状态监控、任务调度器等。它的设计思路直接决定着Hadoop MapReduce计算框架的容错性和可扩展性的好坏，因此，它是整个系统中最重要的组件，是系统高效运转的关键。下面将介绍JobTracker是如何进行状态监控与资源管理的。

2.2.1 状态监控

JobTracker的主要功能之一是作业控制，包括作业的分解和状态监控。其中，最重要的是状态监控，包括TaskTracker状态监控、作业状态监控和任务状态监控，其中，TaskTracker状态监控比较简单，只要记录其最近心跳汇报时间和健康状况（由TaskTracker端的监控脚本检测，并通过心跳将结果发送给JobTracker）即可。

JobTracker在其内部以“三层多叉树”的方式描述和跟踪每个作业的运行状态。如图2.6所示，“三级多叉树”模型分为三层，从高到低依次为JobInProgress、TaskInProgress和TaskAttempt三种对象集合。JobTracker为每个作业创建一个JobInProgress对象以跟踪和监控其运行状态。该对象存在于作业的整个运行过程中：它在作业提交时创建，作业运行完成时销毁。同时，为了采用分而治之的策

略解决问题，JobTracker会将每个作业拆分成若干个任务，并为每个任务创建一个TaskInProgress对象以跟踪和监控其运行状态，而任务在运行过程中，可能会因为软件bug、硬件故障等原因运行失败，此时JobTracker会按照一定的策略重新运行该任务，也就是说，每个任务可能会尝试运行多次，直到运行成功或者因超过尝试次数而失败。JobTracker将每运行一次任务称为一次“任务运行尝试”，即Task Attempt。对于某个任务，只要有一个Task Attempt运行成功，则相应的TaskInProgress对象会标注该任务运行成功，而当所有的TaskInProgress均标注其对应的任务运行成功后，JobInProgress对象会标识整个作业运行成功。

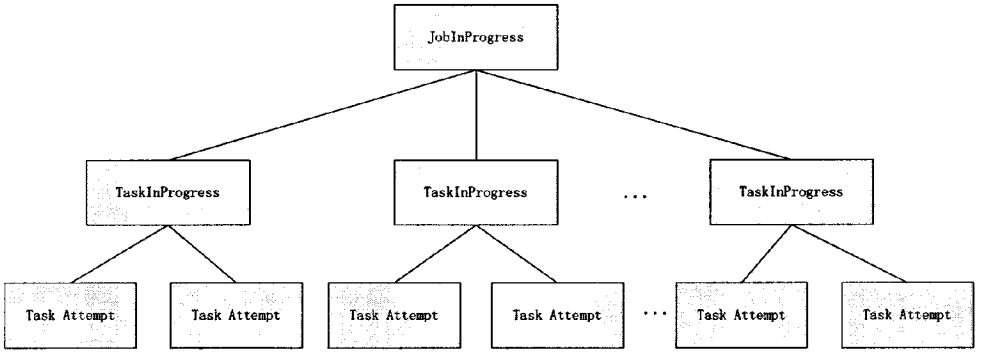


图 2.6 “三层多叉树”作业描述方式队列

JobInProgress类主要用于监控和跟踪作业运行状态，并为调度器提供最底层的调度接口。JobInProess维护了两种作业信息：一种是静态信息，这此信息是作业提交之时就已经确定好的，如Map/Reduce任务个数、作业优先级等信息；另一种是动态信息，这些信息随着作业的运行而动态变化，如正在运行的Map/Reduce任务个数、运行完成的Map/Reduce任务个数、作业开始执行时间等信息。这些信息对于发现TaskTracker、Job、Task故障非常有用，也可以为调度器进行任务调度提供决策依据。

TaskInProgress类维护了一个Task运行过程中的全部信息。比如任务运行进度、任务开始运行时间、运行状态等信息。在Hadoop中，由于一个任务可能会被推测执行或者重新执行，所以会存在多个Task Attempt，且同一时刻，可能有多个处理相同数据的任务尝试同时在执行，而这些任务被同一个TaskInProgress对象管理和跟踪，只要任何一个任务尝试运行成功，TaskInProgress就会标注该任务执行成功。

2.2.2 资源管理

JobTracker的另一个功能是资源管理，它由两部分组成：资源表示模型和资源管理模型。其中，资源表示模型用于描述资源的组织方式，Hadoop采用一个抽象概念Slot来表示各节点上的资源，从而大大简化了资源管理问题，每个Slot封装了

一定数目的CPU和内存；而资源分配模型则决定如何将资源分配给各个作业/任务，这部分工作主要是由任务调度器完成。

在分布式计算领域中，资源分配问题实际上是一个任务调度问题。它的主要任务是根据当前集群中各个节点上的资源（包括CPU、内存等资源）剩余情况与各个用户作业的服务质量（Quality of Service）要求，在资源和作业/任务之间做出最优的匹配。由于用户对作业服务质量的要求是多样化的，因此，分布式系统中的任务调度是一个多目标优化问题，更进一步说，它是一个典型的NP问题。

在Hadoop中，任务调度器和JobTracker之间存在函数相互调用的关系，它们彼此都拥有对方需要的信息或者功能。对于JobTracker而言，它需要调用任务调度器中的assignTasks函数为TaskTracker分配新的任务，同时，JobTracker内部保存了整个集群中的节点、作业和任务的运行时状态信息，这些信息是任务调度器进行调度决策时需要用到的。

Hadoop以队列为单位管理作业的资源，每个队列分配有一定量的资源，同时管理员可指定每个队列中的资源的使用者以防止资源滥用。现有的Hadoop调度器本质上均采用了三级调度模型。当一个TaskTracker出现空闲资源时，调度器会依次选择一个队列、（选中队列中的）作业和（选中作业中的）任务，并最终将这个任务分配给TaskTracker。

在Hadoop中，不同任务调度器的主要区别在于队列选择策略和作业选择策略不同，而任务选择策略通常是相同的，也就是说，给定一个节点，从一个作业中选择一个任务需要考虑的因素是一样的，均主要为数据本地性。

2.3 任务推测执行算法

任务备份是各种计算系统中最常见的容错调度技术。在Hadoop中，与任务备份相关的问题是任务推测执行机制。在分布式集群环境下，因为负载不均衡或者资源分布不均等原因，会造成同一个作业的多个任务之间运行进度不一致。有些任务的运行进度可能明显慢于其他任务，则这些任务会拖慢作业的整体执行进度。为了避免这种情况发生，Hadoop采用了任务推测执行机制。它根据一定的法则推测出“拖后腿”的任务，并为这样的任务启动一个备份任务，让该任务与原始任务同时处理同一份数据，只要有一个任务先完成就代表这个任务已完成。

2.3.1 Hadoop-1.0.0 版本的推测执行算法

Hadoop 在设计之初隐含了一些假设，而正是这些假设影响了 Hadoop 最初的推测执行设计算法。总结起来，共有以下 5 个假设：

（1）每个节点的计算能力是一样的。

（2）任务的执行进度随时间线性增加。

（3）启动一个备份任务的代价可以忽略不计。

（4）一个任务的进度可以表示成已完成工作量占总工作量的比例。

（5）同一个作业同种类型的任务工作量是一样的，所用总时间相同。

很明显，上述假设完全是基于同构集群并且负载均衡的前提下，所以并不适合异构集群和负载不均的情况。

Apache Hadoop 在 Hadoop-1.0.0 版本中，为一个任务启动备份任务需要满足以下条件：

（1）该任务没有其他正在运行的备份任务。（当前 Hadoop 最多允许一个任务同时启动两个 Task Attempt）。

（2）该任务已经运行时间超过 60 秒且当前正在运行的 Task Attempt 落后（同一个作业所有 Task Attempt 的）平均进度的 20%。

当以上条件都满足时，该任务将会被当作“拖后腿”任务，进而需为其启动备份任务。当该任务的某个 Task Attempt 成功运行完成后，JobTracker 会杀掉另外一个 Task Attempt。

该版本实现的推测执行机制还存在许多问题，以下是几个常见的问题：

（1）适用情况考虑不全：当作业的大部分任务已经运行完成时，如果存在若干个 Task Attempt 的运行进度等于或者超过 80%，即使这些是慢任务，但此时也不会启动备份任务。

（2）缺乏保证备份任务执行速度的机制：由于新启动的备份任务需要首先处理原始 Task Attempt 已经处理完的数据，因此需保证备份任务的运行速度不低于原始 Task Attempt，否则将失去启动备份任务的意义。

（3）参数不可配置：比如上面的数值“60”秒和 20%均不可配置，这不能满足用户根据自己集群特点定制参数的要求。

2.3.2 Hadoop LATE 调度算法

Matei Zaharia等人提出了适用于异构环境的LATE[31]推测执行算法。LATE算法后被Apache Hadoop应用于Hadoop-0.21等版本中。LATE算法解决了1.0.0版本中存在的一些问题。对于“适用情况考虑不全”问题，它采用了基于任务运行速度和任务最大剩余时间的策略，尽可能地提高发现“拖后腿”任务的可能性；对于“缺乏保证备份任务执行速度的机制”问题，它根据历史任务运行速度对节点进行性能评测，以识别出快节点和慢节点，并将新启动的备份任务分配给快节点；对于“参数不可配置”问题，它增加了多个配置选项，使一些常量数据尽可能地可配置，进而方便用户按照自己的应用特点和集群特点定制相应的参数值。

当一个存在空闲资源的TaskTracker申请任务时，LATE算法选择备份任务的步

骤如下：

（1）判断该TaskTracker是否是一个慢TaskTracker，如果是，则不能启动任何备份任务。

为了判断一个TaskTracker是否适合启动备份任务，Hadoop通过该TaskTracker上已完成任务的性能表现对其进行评估，如果满足以下条件，则认为该TaskTracker有能力启动一个备份任务：

$$\overline{progressRate(TT)} - \overline{progressRate(*)} \le \sigma \times slowNodeThreshold \quad (2.1)$$

其中 $\overline{progressRate(TT)}$ 表示当前申请任务的TaskTracker平均任务进度增长率；$\overline{progressRate(*)}$ 表示所有TaskTracker的平均任务进度增长率；σ 表示作业中所有任务进度增长率的标准方差；slowNodeThreshold表示任意一个TaskTracker上已运行完成任务的平均进度增长率与所有已运行完成任务的平均进度增长率的最大允许差距（标准方差的倍数），如果超过该阈值，则认为对该作业而言，该TaskTracker性能过低，不会在其上启动一个备份任务。

（2）检查作业已经启动的任务数是否超过限制。

由于一个任务一旦启动了备份任务，则需要两倍的计算资源处理同样的数据，为了防止推测执行机制滥用，Hadoop要求同时启动的备份任务数目与所有正在运行任务的比例不能超过speculativeCap，即满足以下条件：

$$speculativeTaskCount / numRunningTask < SpeculativeCap \quad (2.2)$$

其中，speculativeTaskCount 表示作业已经启动的备份任务数目，numRunningTask 表示作业正在运行的任务总数，speculativeCap 用于该作业允许启动备份任务的任务数目占正在运行任务的百分比。其默认值为 0.1，表示可为一个作业启动推测执行功能的任务数不能超过正在运行任务的 10%。

（3）筛选出作业中满足以下条件的所有任务，并保存到candidates数组中。

条件1：该任务未在该TaskTracker上运行失败过。

条件2：该任务没有其他正在运行的备份任务。

条件3：该任务已运行时间超过60秒。

条件4：该任务已经出现“拖后腿”的迹象，主要判断准则是：

$$\overline{progressRate}_{finished} - progressRate > progressRateStd_{finished} * slowTaskThreshold \quad (2.3)$$

其中，$\overline{progressRate}_{finished}$ 表示已完成任务的平均任务进度增长率；progressRate表示任务当前的任务进度增长率；$progressRateStd_{finished}$ 表示已完成任务的任务进度增长率的标准差；slowTaskThreshold是一个用于限定慢任务个数的阈值。

（4）按照任务运行剩余时间由大到小对candidates中的任务进行排序，并选择剩余时间最大的任务为其启动备份任务。

LATE算法倾向于选择剩余时间最长的任务，因为这样的任务使得其备份任务

替代自己的可能性最大。为此，LATE算法采用了一个简单的线性模型估算一个任务的剩余时间TimeLeft：

$$TimeLeft = (1 - progress) / progressRate \tag{2.4}$$

其中，progress 表示任务当前已完成的任务进度。

LATE算法同样也还存在一些不足，由于LATE算法采用了静态方式计算任务的进度，可能导致性能仍然比较低下，主要体现在以下两个方面。一方面，任务进度和任务剩余时间估算不准确，这会导致部分正常任务被误认为是"拖后腿"任务，从而造成资源浪费。另一方面，未针对任务类型对节点分类，尽管LATE算法可通过任务执行速度识别出慢节点，但它未分别针对Map Task和Reduce Task做出更细粒度的识别。而实际应用中，一些节点对于Map Task而言是慢节点，但对Reduce Task而言则是快节点。

2.4 任务调度器

当前Hadoop自带了多个任务调度器，有FIFO调度器，计算能力调度器和公平调度器。其中FIFO调度器是适用于单一类型作业的单队列任务调度器，而计算能力调度器和公平调度器则是适合于多用户多作业的多队列任务调度器。下面分别介绍FIFO调度器和公平调度器。

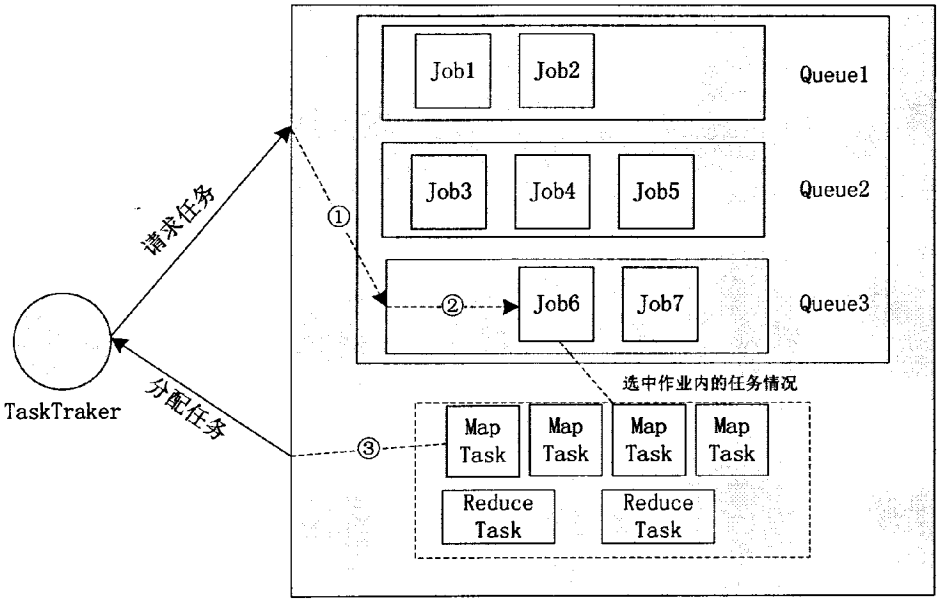
2.4.1 FIFO 调度器

早期的Hadoop应用中，Hadoop处理的作业主要是单个用户的单一类型的批处理作业，故JobTracker使用FIFO调度算法进行作业调度。FIFO的基本调度策略是：将所有用户的作业都提交到一个队列中，然后从队列中优先选择优先级最高的作业，当优先级相同时，优先选择提交时间最早的作业。FIFO算法的缺点是长作业会导致饥饿现象，同时使得系统的吞吐率下降，作业的平均响应时间变长。FIFO不利于集群资源的共享，不支持多用户的并发作业执行。虽然FIFO调度算法在多用户和多种类作业的情况下，存在许多缺点。但是在处理单一用户单一类型的作业时，却因为算法调度机制简洁，所以对系统的调度开销很小，此时效率反而是最好的。

2.4.2 公平调度器

公平调度器是Facebook开发的多用户调度器[42]。该调度器的核心设计思想是基于队列的最小资源量和公平共享量进行任务调度。其中，最小资源量是管理员配置的，而公平共享量是根据队列或作业权重计算得到的。

2.4.2.1 资源的三级调度模型



①选择一个队列②选择一个作业③选择一个任务

图 2.7 资源三级调度模型

当集群中出现一个存在空闲资源的TaskTracker请求任务时，该空闲资源的调度过程遵循如图2.7所示的资源三级调度模型。具体调度过程如下：

步骤1：选择队列。将集群中所有的队列按照公平原则进行排序。然后选择排在最前面的队列。

其中队列的公平原则是指：当存在资源使用量小于最小资源量的队列时，优先选择资源使用率最低的队列，即runningTasks/minShare最小的队列。其中runningTasks是队列当前正在运行的Task数目，minShare为队列的最小资源量，它等于用户配置的队列最小资源量与该队列当前的真实资源需求量的最小值。否则选择任务权重比最小的队列，其中队列的任务权重比为：tasksToWeightRatio = runningTasks / poolWeight。其中poolWeight是管理员配置的队列权重。

步骤2：选择作业。将选中队列中的作业按照公平原则或FIFO原则进行排序。选择最前面的作业。

作业的公平原则：优先将资源分配给资源池中任务权重比最小的作业，其中作业的任务权重比为：tasksToWeightRatio = runningTasks / jobWeight。其中jobWeight是管理员配置的作业权重；当作业的任务权重比也一样时，则优先选择提交时间较早的作业。

FIFO原则：优先选择优先级最高的作业；优先级相同的情况下，选择作业提交时间最早的作业。

Hadoop将按照公平原则排序的队列称为公平队列，将按照FIFO原则排序的队列称为FIFO队列。公平调度器允许用户将不同的队列任意设置成公平队列或者FIFO队列。

步骤3：选择任务。通过延迟调度算法选择一个任务。

2.4.2.2 延迟调度算法

选择任务的时候，包括公平调度器和计算能力调度器都采用了延迟调度算法[43]。延迟调度的目的主要是提高数据本地性，减少数据在网络中的传输。延迟调度算法的核心设计思想是：当出现一个空闲资源时，如果选中的作业没有符合要求的本地性任务的时候，则暂时把资源让给其他作业，直到找到一个满足数据本地性的任务或者达到一个时间阈值，此时不得不为之选择一个非本地性的任务。

为了实现延迟调度，公平调度器为每个作业j维护三个变量：level、wait和skipped，分别表示最近一次调度时作业的本地性级别（0、1、2分别对应node-local、rack-local和off-switch）、已等待时间和最近一次调度是否被延迟调度，并依次初始化为：j.level=0、j.wait=0和j.skipped=false。此外，当不存在node-local任务时，为了尽可能选择一个本地性较好的任务，公平调度器采用了双层延迟调度算法：为了找到一个node-local任务最长可等待W1时间或者继续等待W2时间找一个rack-local任务。

2.4.2.3 公平调度算法的伪代码描述

下面通过伪代码的形式描述公平调度算法一次完整的公平调度的过程：

算法 2.1 公平调度算法

输入： 发送心跳请求的空闲 TaskTracker

```
1.  Function Task assignTasks(TaskTracker tt)
2.      for each job in allJobs do
3.          if job.skipped == true do
4.              job.updateLocalityWaitTimes()
5.              job.skipped = false
6.          end if
7.      end for
8.      while n.availableSlots > 0 then
9.          sort queues using fair policy
10.         for each queue in queues do
11.             sort jobs using fair policy or FIFO policy
```

```
12.                for each job in jobs do
13.                    if (task = job.obtainNewNodeLocalMapTask()) != null then
14.                        job.wait = 0, job.level = 0
15.                        return task
16.                    else if (task = job.obtainNewNodeOrRackLocalMapTask()) !=
    null and (job.level >= 1 or job.wait >= W1) then
17.                        job.wait = 0, job.level = 1
18.                        return task
19.                    else if job.level == 2 or (job.level = 1 and job.wait >= W2) or
    (job.level = 0 and job.wait >= W1 + W2) then
20.                        job.wait = 0, job.level = 2
21.                        task = job.obtainNewMapTask()
22.                        return task
23.                    else
24.                        job.skipped = true
25.                    end if
26.                end for
27.            end for
28.        end while
29.        return taskList
30. end Function
```

2.5 小结

本章对Hadoop的基本架构进行了介绍。包括HDFS和MapReduce的基本架构。然后对Hadoop如何对集群进行状态监控和资源管理进行了介绍，这些内容与本文设计的任务推测执行算法和任务调度算法具有密切的关系。接下来介绍了Hadoop解决集群中慢任务的方法——任务推测执行。最后介绍了Hadoop常用的任务调度器，尤其是公平调度器。公平调度器采用了资源三级调度模型，详细说明了公平调度器如何选择队列、选择作业以及如何通过延迟调度算法选择任务。下面两个章节将重点介绍本文设计的基于预释放资源列表的推测执行算法和基于预释放资源列表的任务调度算法。

第3章 基于预释放资源列表的任务推测执行算法

3.1 研究动机

Hadoop的JobTracker与TaskTracker之间通过"pull"方式进行通信。即JobTracker从不会主动向TaskTracker发送任何信息，而是由TaskTracker主动通过心跳"领取"属于自己的信息。所以JobTracker只能通过心跳应答的形式为各个TaskTracker分配任务。并且每个TaskTracker是在不同的时间向JobTracker发送心跳请求。所以即使当前有一批存在空闲资源的TaskTracker，任务调度器也是逐个接收到它们的心跳请求的。现有的推测执行算法都只根据当前接收到的一个TaskTracker的心跳请求信息，来做出推测执行的决策。由于在异构环境下，不同节点的任务处理速率是不一样的。所以很有可能当前还有更快的空闲资源，只是JobTracker暂时没有接收到它们的心跳请求。任务调度器可以等待更快的TaskTracker发送心跳请求，然后将慢任务分配给更快的资源。甚至有的TaskTracker当前并没有空闲资源，但它上面的任务马上就要完成，资源也将会空闲出来。如果该TaskTracker的任务处理速率足够快的话，有可能使得当前慢任务即使等待该TaskTracker上面的任务完成，然后再处理慢任务，也能够使得慢任务在当前空闲资源上完成得更快。

本章设计了基于预释放资源列表的任务推测执行算法用于解决上述问题。预释放资源列表是根据Hadoop记录的历史信息和集群当前状况监控信息，预测出的能使慢任务更快完成的一批资源。本章算法使得慢任务列表不仅是局限在当前申请任务的资源上考虑推测执行，而是在短时间内即将释放的一些资源上考虑推测执行。从而能找到使慢任务更快完成的资源，减少单个作业的完成时间。同时，预先判断备份任务是否能比原任务更快完成，只有当备份任务能更快完成的时候，才启动该备份任务。这有效减少了不必要的备份任务的执行，避免了系统中资源的浪费。从而有效降低了集群的整体运行时间。

3.2 基于预释放资源列表的推测执行算法

Hadoop对集群系统底层的计算资源状态与任务运行进度进行了监控与维护。通过记录的集群运行时状态信息，可以对资源状态和任务进度进行分析和预测。目前在该方面，Hadoop已提供了一定的研究基础，并应用于任务推测执行机制，用于解决慢任务问题。本章在现有的研究的基础上设计了基于预释放资源列表的任务推测执行算法。本章算法有两个核心的数据结构——慢任务列表和预释放资

源列表。需要说明的是，慢任务列表是包括 LATE 在内的现有推测执行算法都使用到的数据结构，而本章算法的创新之处在于提出的预释放资源列表，以及基于预释放资源列表设计的任务推测执行算法。下面将详细说明这两个列表如何构建。

3.2.1 构建慢任务列表

本章算法中慢任务列表的构建，直接采用了 Apache Hadoop 中 LATE 算法的任务预测模型。以下是具体的构建过程：

（1）预测任务进度

Hadoop的任务进度计算方法：Hadoop将任务的进度用已完成工作量占总工作量的比例来表示。对于Map Task而言，就表示成已读取数据量占总数据量的比例；对于Reduce Task而言，Hadoop将一个Reduce任务分为三个子阶段：Shuffle、Sort和Reduce，Hadoop简单的将Reduce任务每个阶段都设置成占总时间的1/3。所以任务进度的计算方法总结如下：

$$progress = \begin{cases} M / N & \text{for Map task} \\ 1/3 \times (K + M / N) & \text{for Reduce task} \end{cases} \tag{3.1}$$

其中，M表示已读取的数据量；N表示总数据量；K=0，1，2，分别对应Reduce的三个阶段。

（2）预测任务进度增长率

根据任务进度与任务已运行时间预测任务进度增长率：

$$progressRate = progress / T_{run} \tag{3.2}$$

其中 T_{run} 表示任务已运行时间。

（3）筛选慢任务

通过整体比较所有任务的任务进度增长率就可以筛选出慢任务，LATE算法根据如下公式划分慢任务，

$$\overline{progressRate}_{finished} - progressRate > progressRateStd_{finished} * slowTaskThreshold \tag{3.3}$$

其中，$\overline{\text{progressRate}}_{\text{finished}}$ 表示已完成任务的平均任务进度增长率；$\text{progressRateStd}_{\text{finished}}$ 表示已完成任务的任务进度增长率的标准差；slowTaskThreshold 是一个用于限定慢任务个数的阈值。

（4）预测任务剩余运行时间

任务剩余进度与任务进度增长率的比值即是任务剩余运行时间：

$$TimeLeft = (1 - progress) / progressRate \tag{3.4}$$

（5）生成慢任务列表

由公式（3.3）从所有运行的任务中筛选出慢任务，再由公式（3.4）计算出所有慢任务的剩余运行时间，将所有的慢任务根据剩余运行时间由大到小排序，

即构成慢任务列表。

3.2.2 构建预释放资源列表

任务运行完成，节点就会释放该资源。因此，可以将任务的预测模型用于预测资源的状况。预释放资源列表的具体构建过程如下所示：

（1）预测资源释放时间

当节点上的一个任务运行完成后，Hadoop的资源管理器就会释放该任务所占有的资源。所以，资源释放时间即是任务的剩余运行时间。由公式（3.4）就可计算出资源释放时间。

（2）预测节点运行任务的任务进度增长率

Hadoop将节点上已完成任务的平均任务进度增长率作为该节点运行任务的任务进度增长率。

（3）预测备份任务在预释放资源和已释放资源上的运行完成时间

已释放资源是指当前申请任务的空闲资源。备份任务在预释放资源和已释放资源上的运行完成时间的计算：

$$FinishedTime_{speculativeTask} = 1.0 / progressRate_{node} \tag{3.5}$$

其中，$progressRate_{node}$表示节点运行任务的任务进度增长率。

如果真的要将备份任务放在预释放资源上运行，除了实际需要的运行时间，还要考虑上等待预释放资源空闲出来的时间。所以，备份任务在预释放资源上的运行完成时间表示如下：

$$FinishedTime'_{speculativeTask} == 1.0 / progressRate_{node} + TimeLeft_{nodeFree} \tag{3.6}$$

其中，$TimeLeft_{nodeFree}$表示等待资源释放时间。

（4）生成预释放资源列表

本章算法只保留在预释放资源上备份任务运行完成时间小于已释放资源上备份任务运行完成时间的预释放资源。并将这些预释放资源按照完成时间由小到大进行排序，最后再加上已释放资源，即生成本章算法的预释放资源列表。

3.2.3 基于预释放资源列表的备份任务选择算法

构建出慢任务列表和预释放资源列表之后，就可以在这两个列表的基础上进行备份任务选择。具体的备份任务选择算法如下所示：

（1）首先，假设生成慢任务列表的大小为N，生成的预释放资源列表的大小为M。

（2）如果N=0，表示当前集群系统中不存在慢任务，所以不需要进行推测执行。算法结束。

（3）如果M>N，表明当前空闲资源处理任务比较慢，在预释放资源列表中，已释放资源前面存在足够多的更快的待释放资源，则不为该空闲资源分配慢任务，算法结束。

（4）如果M<=N，判断慢任务M的剩余完成时间是否大于备份任务在已释放资源M上的最终完成时间。如果大于，则选择慢任务M进行备份执行；否则，不为该空闲资源分配慢任务。算法结束。

在上述算法中，如果慢任务M的剩余完成时间不大于备份任务在已释放资源M上的最终完成时间，就不为已释放资源M分配慢任务了。这是因为慢任务列表是由慢到快排好序的，预释放资源列表是按由快到慢排好序的。当慢任务M的剩余完成时间不大于备份任务在已释放资源M上的最终完成时间的时候，慢任务M之后任务的剩余完成时间肯定也不大于已释放资源M上的最终完成时间。而慢任务M之前的M-1个任务，因为还有M-1个更快的预释放资源，所以也不需要在已释放资源M上运行。因此，本章算法直接比较慢任务M的剩余运行时间是否大于资源M的预测完成时间，如果大于就分配第M个任务，否则不为该资源分配任务。

3.2.4 基于预释放资源列表的任务推测执行算法

3.2.4.1 基于预释放资源列表的任务推测执行算法的描述

上文分别介绍了本章算法中慢任务列表的构建、预释放资源列表的构建以及基于预释放资源列表的备份任务选择算法。下面将完整的表述本章的基于预释放资源列表的任务推测执行算法。算法的流程如下所示：

（1）在作业的执行过程中，空闲资源向主节点提交任务请求。

（2）如果当前作业存在沿未开始运行的任务，则从尚未开始运行的任务中选择一个任务在空闲资源上执行，算法结束。

（3）否则通过记录的作业执行信息，生成作业的慢任务列表。

（4）如果慢任务列表大小为0，则表明没有慢任务，不需要进行推测执行，算法结束。

（5）否则继续生成预释放资源列表。

（6）假设慢任务列表大小为N，预释放资源列表大小为M，根据本章的备份任务选择算法，如果N>=M，并且慢任务M的剩余运行时间大于已释放资源M完成备份任务的时间，则选择任务M分配给申请任务的资源，算法结束。

（7）否则不为该空闲资源分配慢任务，算法结束。

具体的流程图如图3.3所示：

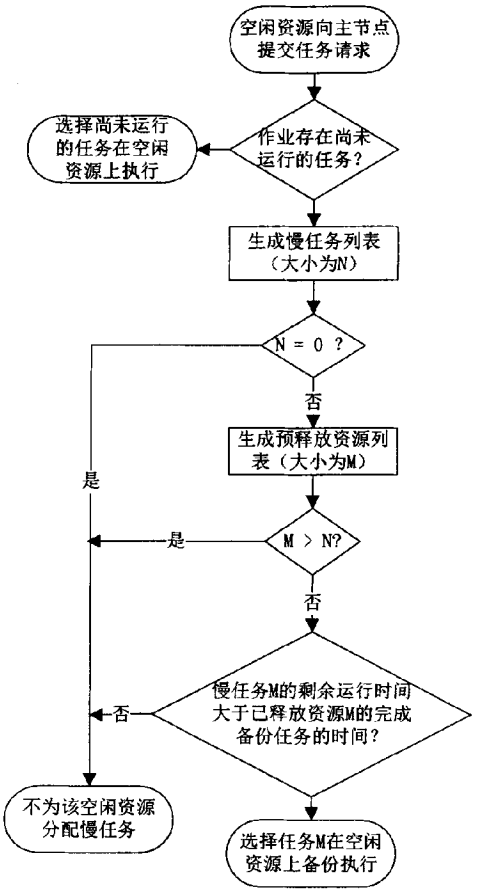


图 3.1 基于预释放资源列表的任务推测执行算法流程图

基于预释放资源列表的任务推测执行算法的具体实现，如下面的伪代码所示：

算法 3.1 基于预释放资源列表的任务推测执行算法

输入： 发送心跳请求的空闲 TaskTracker，任务调度器选择的作业

```
1.  Function Task assignTask(TaskTracker tt, Job job)
        // 如果作业还存在尚未运行的任务
2.      if job.haveNonRunningTask() then
3.          // 从作业中选择一个尚未运行的任务，分配给该 TaskTracker
4.          return job.chooseOneNonRuningTask()
5.      else
            // 生成慢任务列表
6.          slowTaskList = createSlowTaskList()
7.      end if
        // 如果慢任务列表大小为空
8.      if slowTaskList.isEmpty() then
```

```
9.          return null //因为没有慢任务
10.     else
            // 生成预释放资源列表
11.         preReleaseResourceList = createPreReleaseResourceList()
12.     end if
13.     N = slowTaskList().size()
14.     M = preReleaseResourceList.size()
        // 比较慢任务列表与预释放资源列表的大小
15.     if M > N then
16.         return null // 存在足够多的更快的预释放资源
17.     else if slowTaskList[M].taskFinishedTime <=
                preReleaseResourceList[M].taskFinishedTime then
18.         return null // 备份任务不能比原任务更快完成
19.     else
20.         return slowTaskList[M] //本章算法选择的备份任务
21.     end if
22. end Function
```

从算法 3.1 的伪代码中可以看到，备份任务的选择是一个直接选择操作，所以时间复杂度为 O(1)。构建预释放资源列表的时间复杂度等于构建慢任务列表的时间复杂度。所做操作都是对当前作业中所有正在运行的任务进行一次遍历，所以构建预释放资源列表的时间复杂度为 O(n)，其中 n 为当前作业中所有正在运行的任务的数量。所以最终，本章算法的时间复杂度还是等于 LATE 算法的时间复杂度，为 O(n)。

3.2.4.2 算法的分析

从上述伪代码中可以看出，本章算法存在三种可能的调度结果。

第一种结果：当作业中还存在尚未开始运行的任务的情况下，优先执行尚未开始运行的任务。

第二种结果：返回null。它的意思就是不为该请求任务的TaskTracker分配任务。又有三种情况可能会导致产生该结果。一，慢任务列表的大小为空，即当前集群中不存在慢任务，所有就不需要运行备份任务；二，预释放资源列表的大小大于慢任务列表的大小，因为预释放资源列表中存放的是比当前资源更快的资源，所以当预释放资源列表的大小大于慢任务列表的大小的时候，就表明存在足够多的更快的资源，能够使慢任务列表中的所有任务都更快完成，也间接地表明了当

前资源为慢资源；三，慢任务M的剩余完成时间不大于已释放资源M的备份任务完成时间，当备份任务不能比原任务更快完成，启动备份任务只会浪费资源。至于为什么慢任务M与已释放资源M，已经在本章的基于预释放资源列表的备份任务选择算法中解释过了。

第三种结果：选择慢任务M进行备份执行。

对比本章算法与LATE算法备份任务选择的结果，本章算法是选择慢任务M进行备份执行，而LATE算法总是选择最慢的任务（即慢任务1）进行备份执行。下面两幅图表示它们之间的区别：

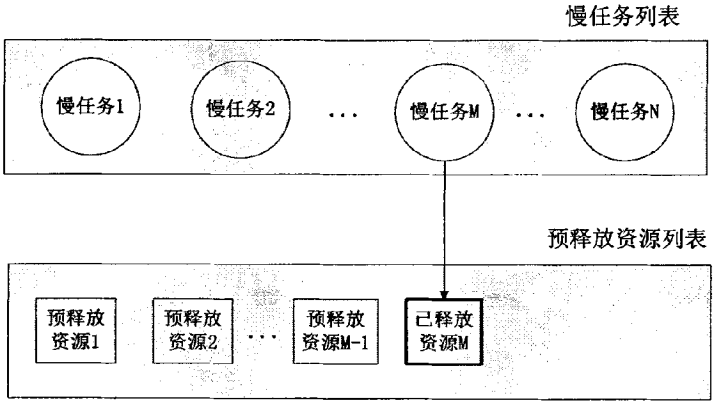


图 3.2 基于预释放资源列表的备份任务选择

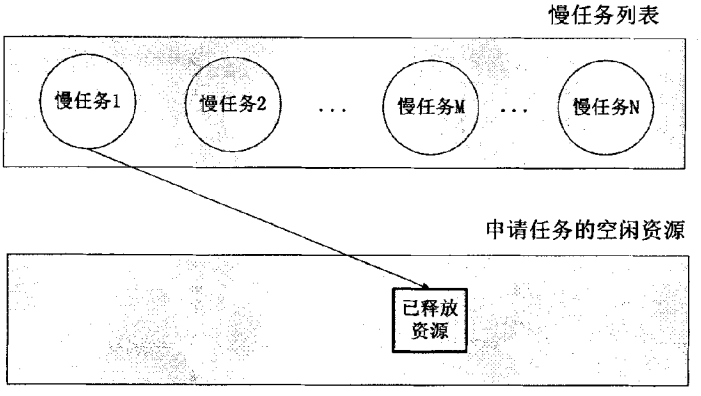


图 3.3 LATE 算法的备份任务选择

一个作业的最终完成时间是由最慢的任务决定的。上述两种备份任务选择策略都是为了使最慢的任务尽快完成。只是 LATE 算法是在一个已释放资源的基础上进行选择，而本章的算法则是在预释放资源列表的基础上进行选择。这样，就能找到真正能使最慢的任务最快完成的资源。而在没有比已释放资源更快的预释放资源的时候，本章算法的备份任务选择策略选择的备份任务将同 LATE 算法一样。

3.3 实验分析

为了评估本章算法的性能，下面将本章的算法与LATE算法以及不采用推测执行的算法进行了实验对比。并且分别从Hadoop的作业完成时间、推测执行任务数以及推测执行成功任务数几个方面进行对比。下面先对实验平台和相关配置进行介绍。

3.3.1 实验平台和配置

本章的实验环境是由9台普通PC机搭建的一个小型Hadoop集群。具体的节点硬件配置情况见表3.1。操作系统统一安装了64位的Ubuntu14.04，Hadoop的版本选择了Apache Hadoop-0.21，因为在该版本的Hadoop系统采用了LATE算法进行任务推测执行。同时，本章算法直接在Hadoop-0.21的源代码上进行修改实现，并重新编译并部署到集群中。

表 3.1 Hadoop 集群各节点的硬件配置情况

节点类型	CPU	内存
主节点	AMD Athlon(tm) 64 X2 Dual Core Processor 4400+	2G
Slave1	Pentium(R) Dual-Core CPU E6700 @ 3.20GHz	2G
Slave2	AMD Athlon(tm) 64 X2 Dual Core Processor 4400+	2G
Slave3	Intel(R) Pentium(R) CPU G630 @ 2.70GHz	2G
Slave4	AMD Athlon(tm) 64 X2 Dual Core Processor 4400+	2G
Slave5	Pentium(R) Dual-Core CPU E6700 @ 3.20GHz	2G
Slave6	Pentium(R) Dual-Core CPU E6700 @ 3.20GHz	2G
Slave7	Vmware Workstation 单核虚拟机	1G
Slave8	Intel(R) Dual-Core(TM) i5-2520M CPU @ 2.5GHZ	4G

本实验中Hadoop环境的参数配置情况见表3.2。由于集群中节点个数较少，而且不存在多个机架，所以HDFS的副本数的值设置为2。文件块大小设置为128M，而不是采用默认的64M，主要是为了增长任务的完成时间。因为Hadoop的任务推测执行算法要求任务运行时间超过1分钟，才考虑为其启动备份任务。因此将本集群的文件块大小设置为128M。mapreduce.map.speculative和mapreduce.reduce.speculative表示是否为Map任务和Reduce任务开启推测执行机制。实验过程中，将需要开启推测执行机制的算法就设置为true，不需要开启推测执行机制的算法设置为false。还有LATE算法需要配置的三个参数，实验中采用Hadoop设定的默认值。

表 3.2 一些重要的 Hadoop 参数配置

配置参数	值	说明
dfs.replication	2	HDFS 副本数
dfs.blocksize	128M	文件块大小
mapreduce.map.speculative	true	Map 任务是否开启推测执行
mapreduce.reduce.speculative	true	Reduce 任务是否开启推测执行
mapreduce.job.speculative.speculativecap	0.1	LATE 中限定备份任务数目的比例
mapreduce.job.speculative.slowtaskthreshold	1.0	LATE 中限定慢任务的阈值
mapreduce.job.speculative.slownodethreshold	1.0	LATE 中限定慢节点的阈值

另外，还对每个从节点上 Map Slot 的数目分别进行了设置，具体设置情况见表 3.3。

表 3.3 Hadoop 集群各节点的 Map Slot 数目配置

从节点名称	Map Slot 数目
Slave1	4
Slave2	4
Slave3	2
Slave4	4
Slave5	4
Slave6	2
Slave7	2
Slave8	4

Slot的数目限定了每个节点上最大的并发任务数。一般来说，一个Slot占用CPU的一个核。当配置的Slot数目超过CPU的核心数时，就会出现多个Slot进行CPU抢占的现象。这样会导致任务处理速率降低。本集群中的机器都是双核的，当把Map Slot数目配置为4个时，机器的任务处理速率大约会变成原来任务处理速率的一半。表3.3配置的目的就是为了使不同机器上的任务处理速率相距较大。这样，慢节点上就会产生慢任务，而快节点也有足够的能力使得慢任务更快完成。

为了使大家对集群中各个节点的任务处理速率有一个比较直观的感受，先在Hadoop中运行一个WordCount作业和一个求Pi值的作业。其中的WordCount应用程

序和求Pi值的应用程序为Apache Hadoop提供的经典应用程序。WordCount程序的功能是对指定的文本文件中所有单词进行计数。求Pi值的应用程序采用了quasi-Monte Carlo[58]方法，该方法通过投飞镖来计算圆周率的值。该应用程序不需要输入数据，用户只需要指定Map任务的个数以及每个Map任务投飞镖的次数即可。如图3.4和图3.5所示，分别表示本集群运行一个WordCount作业和运行一个求Pi值的作业，各个从节点完成一个任务大概所需要的时间。其中求Pi值的程序中本实验将每个Map任务投飞镖的次数设置为5*10E9次。可以观察到，各节点上完成一个Map任务所需的时间相差较大。大致可以观察到，Slave3、Slava6和Slave8的任务处理速率较快，Slave1和Slave5的任务处理速率一般，Slave2和Slave4的任务处理速率相对较慢，而Slave7因为虚拟机给的配置很低，所以任务处理速率非常慢。

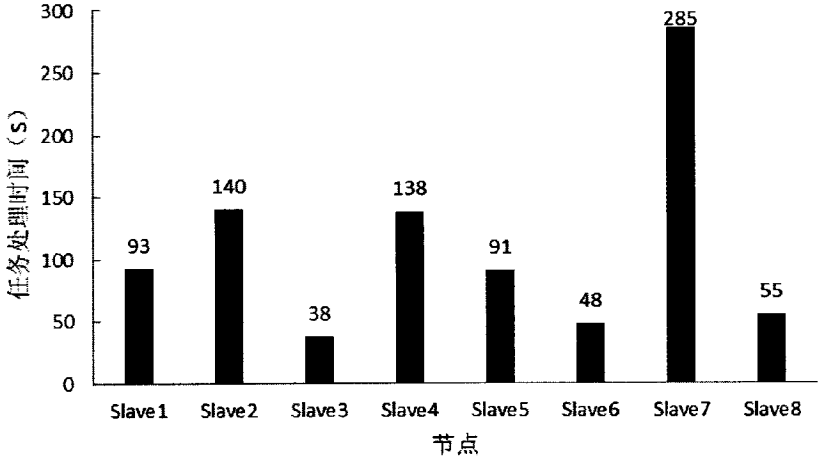


图 3.4 各节点 WordCount 任务处理时间

图 3.5 各节点 Pi 任务处理时间

3.3.2 实验结果分析

本次实验将本章的算法（用Hadoop-My表示）、LATE算法（用Hadoop-LATE表示）以及不使用推测执行的算法（用Hadoop-Non表示）进行对比实验。实验采用了WordCount应用程序和quasi-Monte Carlo方法求Pi值的应用程序作为作业，分别观察三种算法在运行一个作业和多个作业以及不同大小作业时的性能。

（1）Hadoop运行一个作业

将三种算法都运行数据大小为5G、10G和15G的WordCount应用程序各10次，运行40、80和120个Map任务的Pi应用程序各10次，其中每个Map任务投飞镖的次数为5*10E9次。然后记录下三种算法的作业完成时间以及推测执行的任务数和推测执行成功的任务数。从图3.6和图3.7中可以看到，Hadoop-Non的作业完成时间明显大于其余两个启用了任务推测执行机制的算法。但任务推测执行机制也不是每次都能起到良好的效果。比如在运行10G数据的WordCount作业的时候，从图中可以看到Hadoop-Non与Hadoop-LATE的作业完成时间是基本相同的。这是因为只有当备份任务比原任务更快完成了，才能起到使作业更快完成的目的。而Hadoop-My也不是每次都比Hadoop-LATE的作业完成时间更快，比如在运行5G数据的WordCount作业的时候，从图中可以看到两者的时间也是差不多的。这是因为，如果在预释放资源列表中找不到更快的资源，Hadoop-My与Hadoop-LATE选择的慢任务其实是一样的。总体上，从图3.6和图3.7的实验结果来看，Hadoop-My的作业完成时间是最短的。Hadoop-My能够比别的算法更快完成，是因为它能从预释放资源列表中找到更快的资源，从而使慢任务更快完成。

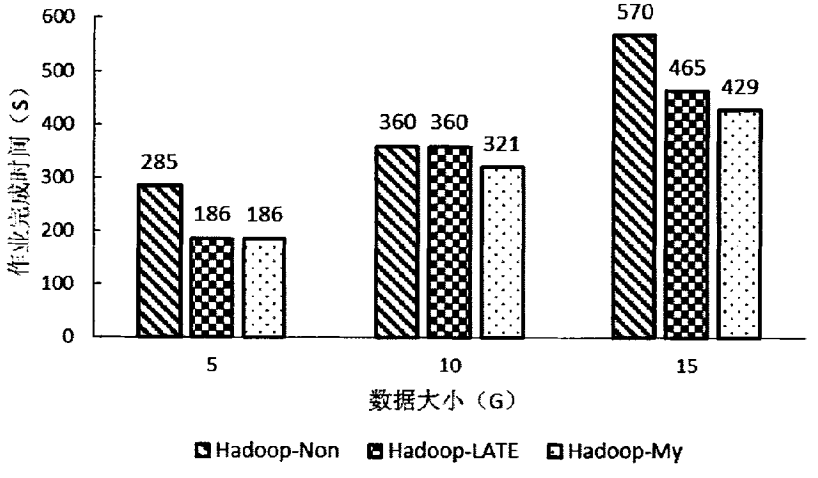


图 3.6 一个 WordCount 作业的完成时间

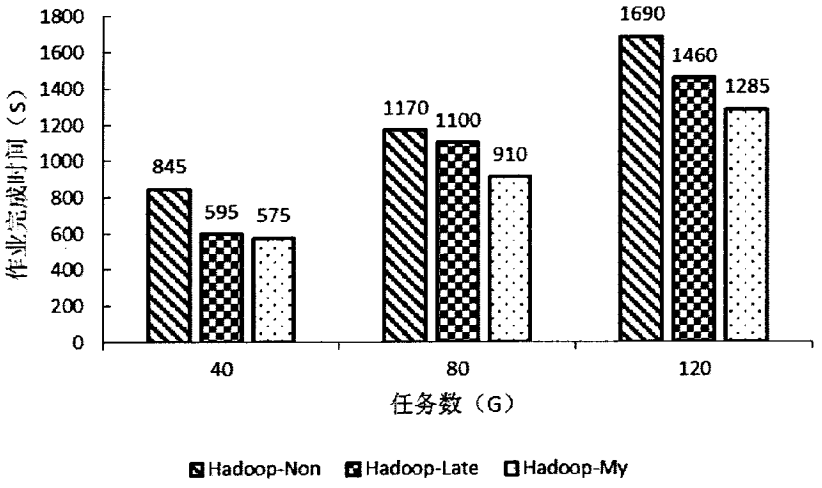


图 3.7 一个 Pi 作业的完成时间

表3.4和表3.5比较了Hadoop-LATE与Hadoop-My两个算法的推测任务数和推测成功任务数。因为Hadoop-Non没有推测任务，所以此项对比Hadoop-Non不参加。从表中可以看出，Hadoop-LATE比Hadoop-My启用了更多的推测任务数，同时Hadoop-LATE推测成功的任务数也少于Hadoop-My。显然，一个任务推测执行算法如果能启动更少的推测任务，同时又能取得更多的推测成功任务数，该算法的性能将是更好的。当Hadoop中同时运行多个作业的时候，将更能看到任务推测执行算法启动较少的推测任务对集群性能的提高。

表 3.4 一个 WordCount 作业的推测任务数及推测成功任务数

算法	5G 数据	10G 数据	15G 数据
Hadoop-LATE	2/4	6/10	8/20
Hadoop-My	2/2	5/7	10/13

表 3.5 一个 Pi 作业的推测任务数及推测成功任务数

算法	40 个任务	80 个任务	120 个任务
Hadoop-LATE	4/6	8/14	14/24
Hadoop-My	4/5	12/14	18/22

（2）Hadoop运行多个作业

下面比较不同算法在同时运行多个作业时的实验效果。图3.8和3.9分别表示同时运行2个WordCount作业和2个Pi作业时的作业完成时间，图3.10和图3.11分别表示同时运行3个WordCount作业和3个Pi作业时的作业完成时间。从实验结果可以看到， Hadoop-My算法的作业完成时间是最少的，Hadop-Non的作业完成时间也基本上是最长的。值得注意的是，图3.7中运行5G数据的WordCount作业的时候，以及图3.11中运行80个任务的Pi作业的时候，Hadoop-LATE的作业完成时间甚至大

于Hadoop-Non。这是因为，任务推测执行算法虽然很多时候能使慢任务更快完成，但是它是以占用集群更多资源为代价的。当集群中存在多个作业的时候，一个作业占用了更多的资源，必将导致别的作业能够得到的资源变少。这样，反而有可能导致集群的整体运行时间变长。所以，在没有好的任务推测执行算法的情况下，有些生产集群中，甚至直接禁用任务推测执行。这也说明了，一个好的任务推测执行算法的重要性。

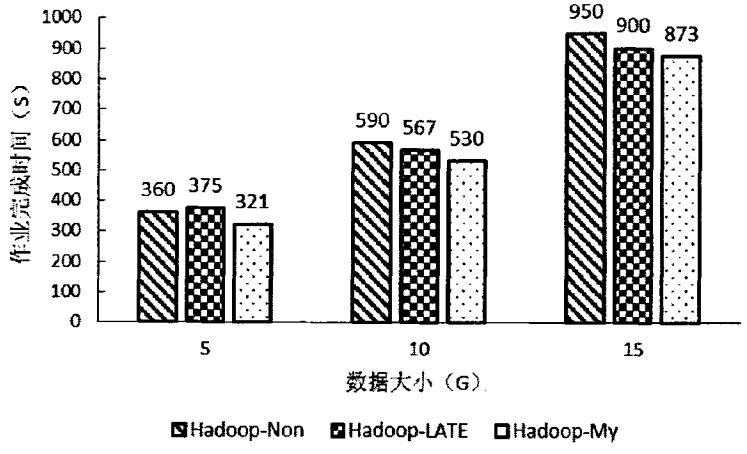


图 3.8 两个 WordCount 作业的完成时间

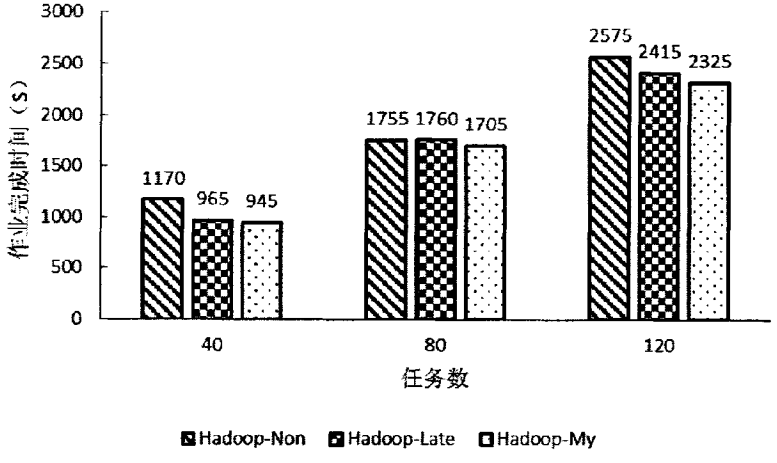


图 3.9 两个 Pi 作业的完成时间

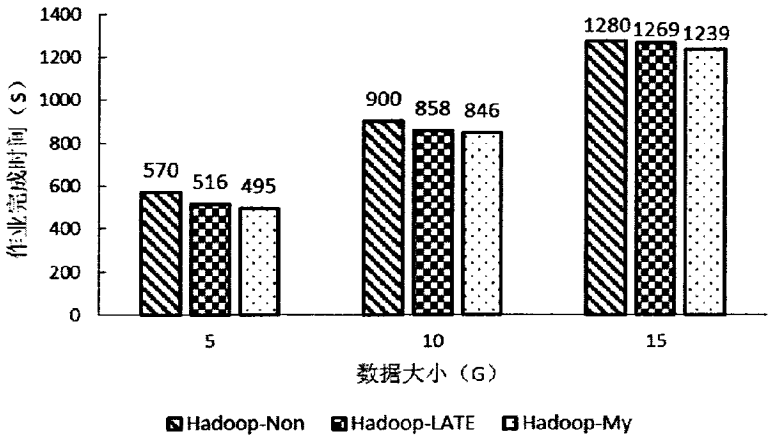


图 3.10 三个 WordCount 作业的完成时间

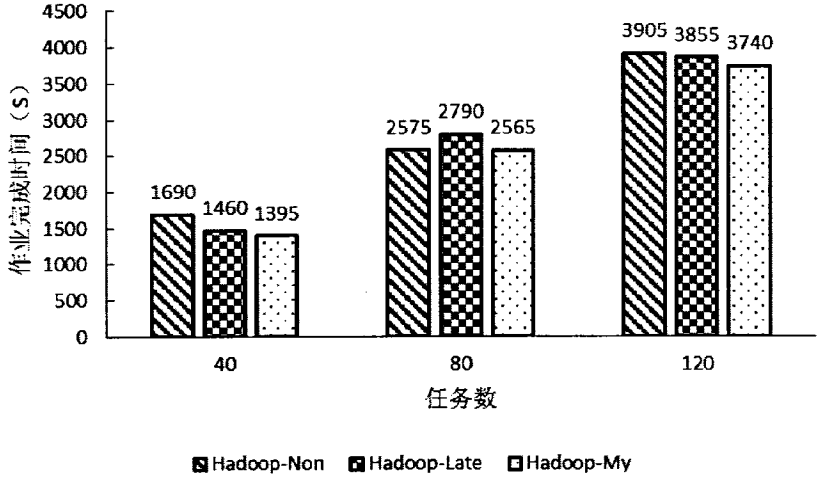


图 3.11 三个 Pi 作业的完成时间

由表3.6和表3.7所示，Hadoop-My启动的备份任务数要明显少于Hadoop-LATE的。但推测任务成功数则有时候会少于Hadoop-My。这是因为，Hadoop-My总是将最快的资源分配给最慢的任务，所以就有可能存在次慢的任务找不到更快的资源，从而使总的推测任务成功数减少了。Hadoop-My算法就是首先保证最慢的任务能够推测执行成功，因为作业的最终完成时间是由最慢的任务决定的。所以即使总的推测任务成功数减少了，Hadoop-My的作业完成时间也能更小。

表 3.6 两个 WordCount 作业的推测任务数及推测任务成功数

算法	5G 数据	10G 数据	15G 数据
Hadoop-LATE	6/10	12/22	18/38
Hadoop-My	6/8	11/15	19/25

表 3.7 两个 Pi 作业的推测任务数及推测任务成功数

算法	40 个任务	80 个任务	120 个任务
Hadoop-LATE	14/20	24/34	32/50
Hadoop-My	12/15	23/27	31/38

表 3.8 三个 WordCount 作业的推测任务数及推测任务成功数

算法	5G 数据	10G 数据	15G 数据
Hadoop-LATE	12/16	22/36	30/58
Hadoop-My	11/13	19/24	30/36

表 3.9 三个 Pi 作业的推测任务数及推测任务成功数

算法	40 个任务	80 个任务	120 个任务
Hadoop-LATE	18/22	32/44	46/66
Hadoop-My	17/20	35/38	46/53

由以上所有的实验结果可以看出，不管是运行单个作业还是同时运行多个作业，Hadoop-My算法的作业完成时间都是最少的。这是因为Hadoop-My算法总是尽可能保证最慢的任务最先完成。同时Hadoop-My算法能够启动较少的推测任务，并且保证较多推测任务成功数。这可以使更多的空闲资源用于别的作业的执行，总体上提高了集群的效率。而Hadoop-LATE算法在运行单个作业的时候，虽然启动了许多推测失败的任务，但浪费的资源不会影响到Hadoop的整体完成时间。而当作业较多时，推测失败的任务浪费的资源就影响了别的作业的运行，从而延长了Hadoop的整体完成时间。至于Hadoop-Non算法，从整体的实验结果来说，它的作业完成时间是最慢的。虽然任务推测执机制会占用更多的资源，有可能会使集群的性能降低。但是通过改进任务推测执行算法，合理的启动备份任务，还是能使集群的性能得到提高。

最后对本章实验需要补充说明的是，由于本次实验集群的节点个数有限，所以每次推测执行的时候，慢任务可以选择的节点数量都是极其少的，同时节点间的任务处理速率相差不大，所以每次不同的节点选择也不能明显的拉开最终作业完成时间的差距。另外，本章实验中慢任务的来源只能是机器的任务处理速率较慢导致的，这类慢任务还是能在可预期的时间内完成的。但实际集群应用中，还有很多导致慢任务的原因。比如某个节点的速度可能突然变得很慢，需要等到任务运行时间过长，才会被Hadoop主动杀死。这类任务会使得不采用任务推测执行算法的集群，需要很长时间才能完成作业。而使用了推测执行算法的集群，则可以较好的避免这个问题。

3.4 小结

本章首先介绍了Hadoop主节点和从节点的pull通信方式，分析了在该通信方式下现有的任务推测执行算法存在的缺陷。接着设计了基于预释放资源列表的推测执行算法。算法部分首先介绍了如何构建慢任务列表和预释放资源列表。然后介绍了在构建出来的慢任务列表和预释放资源列表的基础上进行备份任务选择的算法。最后从整体上描述了基于预释放资源列表的任务推测执行算法。实验部分首先对本章实验采用的Hadoop集群配置进行了介绍。然后将本章的算法与LATE算法以及不使用推测执行的算法进行了对比。对比了不同算法在运行一个作业和同时运行多个作业，以及不同数据大小的作业的情况，并从Hadoop完成时间、启动推测任务数以及推测成功任务数几方面进行对比。实验结果表明，本章设计的算法启动推测任务数更少，同时推测成功率更高，从而有效节省了资源浪费，并能使Hadoop中的作业以更短时间完成。

第4章 基于预释放资源列表的任务调度算法

4.1 研究动机

Hadoop是通过各个从节点在不同的时间向主节点以“pull”的方式发送心跳请求来获取任务的。现有的任务调度算法都只根据当前请求任务的从节点状况，来选择任务进行分配。而没有将更多的资源与集群中各个作业的具体需求联系起来，从而做出更优的调度方案。

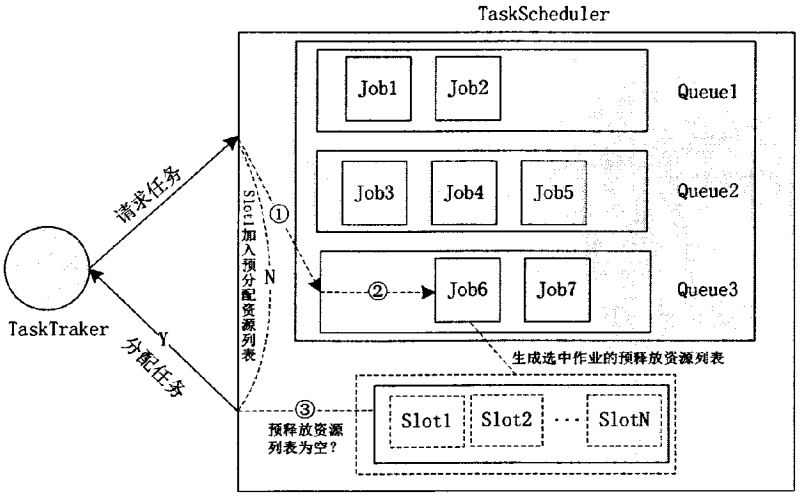
目前Hadoop中常用的任务调度器有计算能力调度器和公平调度器。这些调度器在选择队列和选择作业的时候都遵循公平的原则或者是计算能力的原则，但在选择任务的时候，为了满足任务的本地性，采用了延迟调度算法。在当前作业没有满足本地性要求的任务时，需要将资源让给下一个作业。所以公平调度器存在公平性和本地性之间的矛盾，计算能力调度器存在计算能力和本地性之间的矛盾。同时，延迟调度算法需要手动设置node-local等待时间和rack-local等待时间两个参数。而这两个参数跟集群情况和作业情况密切相关，用户很难找到适合当前集群中作业情况的参数设置，而且也不可能在作业情况改变的情况下，又重新去调度这两个参数。

本章设计了一种基于预释放资源列表的任务调度算法，充分利用了Hadoop记录的历史信息和集群当前状况监控信息来更好地帮助资源调度。本章算法无需手动设置延迟等待时间。并通过对预释放资源列表中的资源进行预调度，解决了公平性和本地性之间的矛盾。另外，本章设计的任务调度算法可以像延迟调度算法一样，同时应用于公平调度器和计算能力调度器。这样，原有的调度器可以满足原有调度规则的前提下（如时间、优先级、公平等），然后从预释放资源列表中获取更适合的资源分配给调度器指定的作业。本章以公平调度器为例，将本章的调度算法集成到公平调度器中。

4.2 基于预释放资源列表的任务调度算法

Hadoop大数据处理平台为每个计算任务维护的运行日志和内建计数器,可用于跟踪整个执行过程中各种事件的时序并获得相关的统计信息。Hadoop的推测执行机制利用这些统计信息来获取慢任务，并对慢任务进行备份执行。Hadoop的推测执行机制提供了计算任务进度、任务进度增长率、任务剩余运行时间以及节点的任务处理速率的方法。本章基于这些方法来构建本章算法的预释放资源列表，并基于预释放资源列表来实现任务调度。

4.2.1 基于预释放资源列表调度的资源三级调度模型



①Fair 原则选择队列②Fair 或 FIFO 原则选择作业③基于作业的预释放资源列表选择任务

图 4.1 基于预释放资源列表的资源三级调度模型

当集群中出现一个存在空闲资源的TaskTracker请求任务时，该空闲资源的调度过程遵循如图4.1所示的资源三级调度模型。具体调度过程如下：

步骤1：选择队列。按照公平原则选择最优先的队列。

这里的公平原则与原公平调度算法的公平原则基本上是一样的。只是在某些参数上还需要考虑预分配资源。基于预释放资源列表的公平调度算法的公平原则具体是指：当存在资源使用量小于最小资源量的队列时，优先选择资源使用率最低的队列，即(runningTasks+poolPreAssignNum)/minShare最小的队列。其中runningTasks是队列当前正在运行的Task数目，poolPreAssignNum是已经预分配给该队列的资源量，minShare为队列的最小资源量，它等于用户配置的队列最小资源量与该队列当前的真实资源需求量再减去poolPreAssignNum的最小值。否则选择任务权重比最小的队列，其中队列的任务权重比为：tasksToWeightRatio = (runningTasks+poolPreAssignNum) / poolWeight。其中poolWeight是管理员配置的队列权重。

步骤2：选择作业。从选中队列中的作业，按照公平原则或FIFO原则选择最优先的作业。

作业的公平原则：优先将资源分配给资源池中任务权重比最小的作业，其中作业的任务权重比为：tasksToWeightRatio = (runningTasks+jobPreAssignNum) / jobWeight。其中jobPreAssignNum 是已经预分配给该作业的资源量，jobWeight是管理员配置的作业权重；当作业的任务权重比也一样时，则优先选择提交时间较早的作业。

步骤3：选择任务。通过基于预释放资源列表的任务调度算法选择一个任务。

下面将具体描述如何构建预释放资源列表，以及基于预释放资源列表的公平调度算法。

4.2.2 预释放资源列表的构建

算法 4.1 构建预释放资源列表的伪代码：

输入参数依次为：按照公平原则被选中的作业，当前申请任务的空闲 TaskTracker 和预分配资源列表

```
Function List<Task> createPreReleaseResourceList(Job job,
        TaskTracker tt, List<Task> preAssignedTasks)
1.      preReleaseResourceList = null
2.      for each task in allRunningTasks do
3.          if !tabuTasks.contains(task) &&
              task.finishedTime() < job.getTaskFinishedTime(tt) do
4.              preReleaseResourceList.add(task)
5.          end if
6.      end for
7.      sort preReleaseResourceList comparing task.finishedTime()
8.      return preReleaseResourceList
9.  end Function
```

预释放资源列表是从当前所有正在运行的任务中挑选出来的。将满足如下条件的任务加入预释放资源列表：

条件1：任务所在资源不能包含在预分配资源列表中。

其中预分配资源列表是指一批已经预分配给Job的资源。在下节的基于预释放资源列表的任务调度算法中会进行预分配。

条件2：作业在任务所在资源的任务完成时间小于作业在当前空闲资源的任务完成时间。

其中作业在任务所在资源的任务完成时间是由三部分构成的，该任务的剩余完成时间、当前作业中的任务在该任务所在主机的完成所需时间和数据传输时间。作业中的任务在当前空闲资源的完成时间由两部分构成，当前作业中的任务在该任务所在主机的完成所需时间和数据传输时间。

任务的剩余完成时间可以公式（3.4）计算得到，当前作业中的任务在指定主机的完成所需时间可由公式（3.5）计算得到。数据传输时间是由任务的本地性决定的。对node-local任务的数据传输时间为0，rack-local和non-local任务的数据传输时间取决于任务数据大小和机架内的网络带宽及机架间的网络带宽。

在本章的任务调度器中，不同作业会生成不同的预释放资源列表。因为现在Hadoop集群的机器一般都是异构的，导致这些机器的CPU运算能力和I/O读写能力都是不同的。这样，Hadoop中不同类型的作业，如CPU密集型或I/O密集型作业面对同一个资源时，任务的处理时间是不一样的。所以需要针对不同作业生成不同的预释放资源列表。

4.2.3 基于预释放资源列表的公平调度算法

4.2.3.1 基于预释放资源列表的公平调度算法的伪代码描述

下面通过伪代码的形式描述基于预释放资源列表的公平调度算法一次完整的公平调度的过程：

算法 4.2 基于预释放资源列表的公平调度算法

输入： 发送心跳请求的空闲 TaskTracker

```
1.  Function Task assignTasks(TaskTracker tt)
2.      preAssignedTasks = null
        // 初始化每个作业的预分配资源数
3.      for each job in allJobs do
4.          job.preAssignNum = 0
5.      end for
        // 一直循环，直到做出任务调度决策
6.      while true then
            // 筛选出还需要资源的队列
7.          for each pool in allPools do
8.              if pool.getDemand() > pool.getPreAssignNum() do
9.                  needSlotPools.add(pool)
10.             end if
11.         end for
            // 当没有队列需要资源，则不为当前资源分配任务
12.         if needSlotPools.isEmpty() do
13.             return null
14.         end if
            // 通过公平原则选择一个队列
15.         chosedPool = choosePoolWithFair(needSlotPools)
            // 从选中的队列中选择还需要资源的作业
16.         for each job in chosedPool.getJobs() do
```

```
17.             if job.getDemand() > job.getPreAssignNum() do
18.                 needSlotJobs.add(job)
19.             end if
20.         end for
            // 通过公平原则或 FIFO 原则选择一个作业
21.         chosedJob = chooseJobWithFairOrFIFO(needSlotJobs)
            // 创建预释放资源列表
22.         preReleaseResourceList = createPreReleaseResourceList()
            // 如果预释放资源列表为空，按照任务调度原则选择一个任务
23.         if preReleaseResourceList.isEmpty() then
24.             return chosedJob.getTask()
25.         else
                // 将预释放资源列表的首个资源预分配给该作业
26.             chosedJob.preAssignNum++
27.             preAssignedTasks.add(preReleaseResourceList.get(0))
28.         end if
29.     end while
30. end Function
```

从上述伪代码可以看出，本章算法返回的结果可能为null，或者是从最终选择的作业中按照任务调度原则选择一个任务。当返回null的时候，一般是所有的队列都已经预分配了足够的预释放资源，也就是说存在足够多的能使任务更快完成的预释放资源，所以就不为这个较慢的资源分配任务了。任务调度原则是指优先选择满足node-local的任务，次优选择rack-local的任务，最后选择non-local的任务。

4.2.3.2 几种不同的调度算法举例对比

接下来通过一个简单的场景假设，将直接调度算法、延迟调度算法和本章的调度算法进行对比：假设当前集群只有一个队列，队列中有3个作业。Job1、Job2和Job3分别在Slot1、Slot2和Slot3上有数据本地性。作业优先级job1>job2>job3。Slot3、Slot1和Slot2将依次空闲。当前Slot3是请求任务的空闲资源。

基于预释放资源列表的调度过程如图4.2所示，图中的预释放资源列表根据数据本地性和节点速度等因素生成：

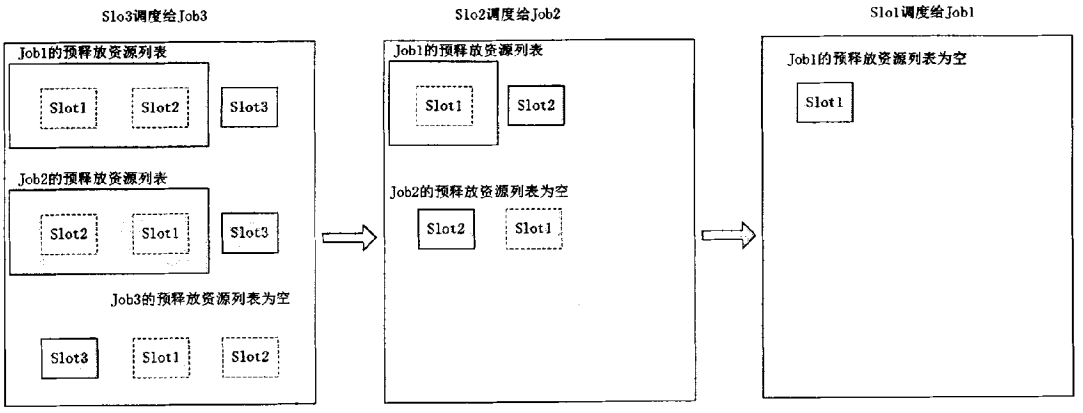


图 4.2 基于预释放资源列表的调度结果

（1）首先Slot3空闲资源开始请求任务，按照公平原则首先让Job1选择资源。按照本章算法Job1会选择预释放资源列表的第一个资源。于是将Slot1预调度给Job1；继续按照公平原则让Job2选择资源，同样选择预释放资源列表中的第一个资源。于是将Slot2预调度给Job2。最后按照公平的原则让Job3选择资源，Job3的预释放资源列表为空，因为Job3的任务在Slot1和Slot2上的完成时间要大于在Slot3的完成时间，此时就将Slot3分配给Job3，同时因为Job3有node-local任务在Slot3上，所以Job3会选择一个本地性的任务在Slot3上执行。

（2）接着Slot2空闲出来，并请求任务。按照公平的原则首先让Job1选择资源，将Slot1预分配给Job1；接着Job2选择资源，直接从Job2上选择一个本地任务在Slot2上执行。

（3）最后Slot1空闲出来，并请求任务。Job1选择一个本地任务在Slot1上执行。

通过本章的任务调度算法，所有的作业都获得满足本地性的资源。

接下来再看看延迟调度算法的结果：

（1）Slot3空闲并请求资源：因为Slot3不满足Job1的本地性，Job1把Slot3让给下一个Job；Slot3同样不满足Job2的本地性，Job2继续把资源让给下一个作业；Slot3满足Job3的本地性，所以就将Slot3分配给Job3。

（2）Slot2空闲并请求资源：Slot2同样不满足Job1的本地性，但此时Job1是否会把资源让给下一个作业，取决于延迟调度算法延迟等待时间值的设置。如果当前等待时间小于W1，则让给Job2；如果当前等待时间大于W1且小于W1+W2，则看Slot2是否满足rack-local，满足的话，Slot2调度给Job1，否则让给Job2。如果大于W1+W2，则直接将Slot2调度给Job1。其中W1和W2是延迟调度算法需要设置的两个时间参数。

（3）Slot1空闲并请求资源。Slot1调度给剩下的那个作业。

另外，FIFO调度器采用了直接调度算法，即直接把当前空闲的资源调度给当

前优先级高的作业。所以调度结果就是Slot3分配给Job1，Slot2分配给Job2，Slot1分配给Job1。

通过下面的表格对比一下不同的调度算法的结果：

表 4.1 举例场景下不同算法的调度结果对比

任务调度算法	调度结果	本地性比例
直接调度	Slot3→Job1 Slot2→Job2 Slot1→Job3	1/3
延迟调度	Slot3→Job3 Slot2→Job1 Slot1→Job2 或 Slot3→Job3 Slot2→Job2 Slot1→Job1	1/3 或 3/3
基于预释放资源列表的调度	Slot3→Job3 Slot2→Job2 Slot1→Job1	3/3

从上表可以看出，本章的算法任务本地性的效果是最好的，而延迟调度算法的结果则取决于延迟等待时间的设置。而直接调度算法则是任务本地性最差的。

但本章算法的原则并不是为了使更多任务满足本地性，而是每次调度都让被选中的作业选择对该作业对说最快的资源。不同于延迟调度算法，在当前作业满足不了本地性的情况下，需要将调度机会让给下一个作业。也就是说，公平性需要让步于本地性。这就存在了公平性和本地性之间的矛盾。本章提出的预释放资源列表，保证了作业能够从更多的资源中发现更快的资源。然后通过预调度，保证了每次选中的作业都可以选择对该作业来说最好的资源。最好的资源不是指满足本地性的资源，而是指能使作业中的任务更快完成的资源。因为非本地性任务需要数据传输时间，所以，一般来说，满足本地性的任务完成得更快。所以，这也保证了本章算法能保持很高的任务本地性。

4.3 实验分析

本章将基于预释放资源列表的任务调度算法集成到Hadoop的公平调度器中，下面就将本章的基于预释放资源列表的公平调度算法和Hadoop的公平调度算法以及FIFO调度算法进行实验对比。

4.3.1 实验平台和配置

考虑到将一个调度算法直接在实际工作集群系统进行长时间测试计算、评估耗时太长，而且要找到集群规模、计算资源与计算场景完全满足每一次实验要求的实际系统也是非常困难的。因此，本章根据Hadoop底层原理，基于Java语言实现了一个Hadoop模拟器，用于验证和分析本调度算法的有效性。以下是一些相关实现细节：

（1）模拟的Hadoop硬件配置情况：3个机架，每个机架上分别有10个慢节点、10个普通节点和10个快节点。其中，慢节点的任务处理速率是普通节点的0.8倍，快节点的任务处理速率是普通节点的1.2倍。每个节点上都设置4个Slot。机架内的网络带宽是20M/S，机架间的带宽则是5M/S。

（2）HDFS的配置情况：每个数据块的大小设置为128M。然后每个数据块的备份数设置为3。其备份策略考虑了负载均衡，具体为：首先选择当前负载最小的节点保存第一份数据；第二份数据保存的节点要求与第一份数据所在节点同机架但不同节点，在满足要求的情况下选择负载最小的节点；第三份数据保存的节点与第一份数据不同机架，同时也选择负载最小的节点。每个数据块有3份备份，这表明每个任务在3个节点上会有节点本地性。

（3）延迟调度算法的两个时间参数：W1设置为5秒，W2设置为20秒。这两个时间参数的设置也是考虑了集群的网络带宽。因为在一个机架内，一个任务的数据传输时间约为6.4秒，通过BlockSize除以机架内网络带宽计算所得；而跨机架传输时，一个任务的数据传输时间约为25.6秒，通过BlockSize除以机架间网络带宽计算所得。保证了等待到本地性任务的收益要大于非本地性任务的数据传输时间。

（4）作业类型情况：根据作业大小，划分了三种类型的作业。

表 4.2 不同作业类型的任务处理时间

作业类型	数据量	普通节点任务处理时间
大作业	800*128M	800 秒
普通作业	300*128M	300 秒
小作业	60*128M	60 秒

一个作业数据量的大小决定了这个作业包含的任务数，比如大作业的数据量为800*128M，就表明大作业会被分成800个任务。

4.3.2 实验结果分析

实验将对本章的基于预释放资源列表的公平调度算法（Fair-PRRL，Pre-Release Resource List）和Hadoop的采用延迟调度的公平调度算法（Fair-DL，Delay Sheduling）以及FIFO调度算法进行对比。考察不同算法运行各种类型的作业的结果，然后对算法在整个Hadoop的完成时间、作业本地性任务的情况以及平均作业响应时间等方面进行评估。

本章总共运行了四组实验，具体如下：

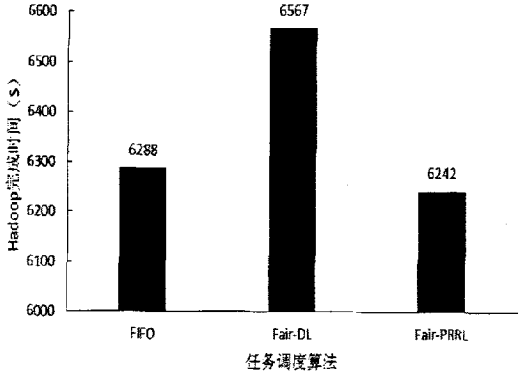
（1）创建3个队列，每个队列提交100个小作业，同时运行。

（2）创建3个队列，每个队列提交50个普通作业，同时运行。

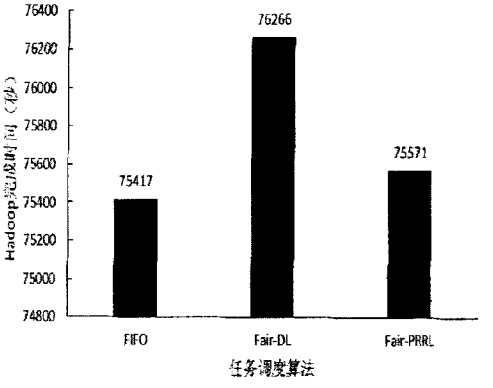
（3）创建3个队列，每个队列提交20个大作业，同时运行。

（4）创建3个队列，每个队列提交100个小作业、50个普通作业和20个大作业，同时运行。

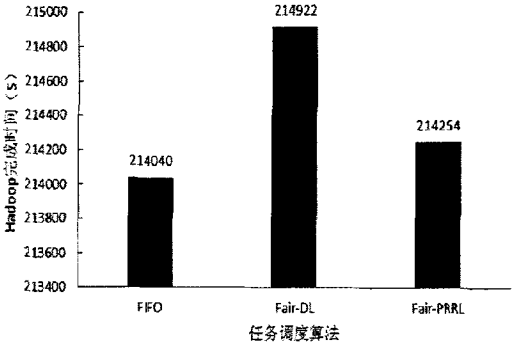
图4.3显示了各个调度算法分别运行完成Hadoop中所有作业所需要的时间。实验结果表明，Fair-PRRL算法在Hadoop完成时间方面明显好于Fair-DL算法。在运行小作业时，Fair-PRRL算法的Hadoop完成时间少于FIFO算法，而其它情况下，则略大于FIFO算法。这也说明，其实在执行批处理作业的时候，FIFO算法反而是效果最好的。因为FIFO算法只是按照作业提交时间逐个地执行作业，这样，作业执行的时候就不会有别的作业抢占资源。而Fair-DL算法，会同时运行很多作业，这样就可能导致某个作业的本地性资源，正好有其它作业的任务在这个资源上面执行。这就导致了这个作业只能选择非本地性的资源了，这也导致了更多的运行时间。Fair-PRRL算法也是同时运行多个作业，所以也存在着这样的问题。



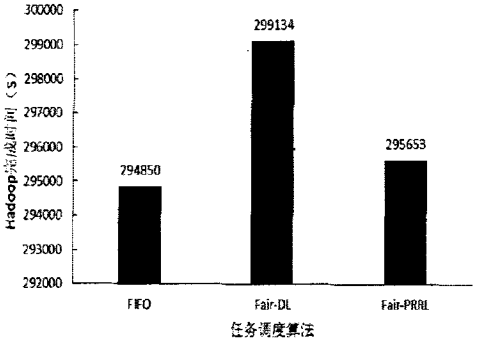
a)小作业 Hadoop 完成时间



b)普通作业 Hadoop 完成时间



c)大作业 Hadoop 完成时间



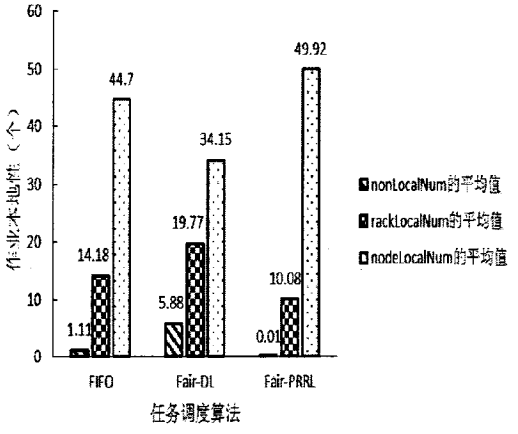
d)混合作业 Hadoop 完成时间

图 4.3 Hadoop 完成时间

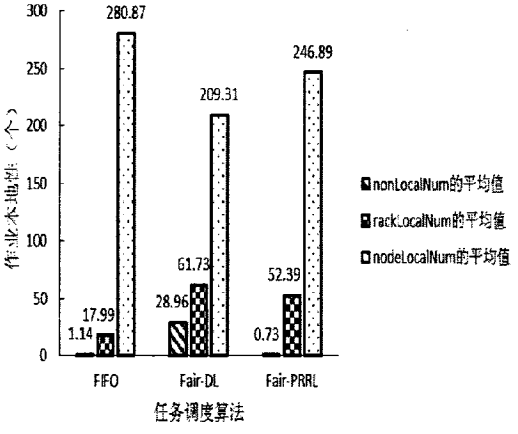
Fari-DL算法和Fair-PRRL算法都考虑到了这个问题，所以Fari-DL算法通过延迟调度来解决作业的本地性资源已经被别的作业占领的问题，Fair-PRRL算法则通过基于预释放资源列表的任务调度算法来解决这个问题。图4.4显示了各个调度算法的任务本地性情况。图中的nonLocalNum表示非本地性任务数，rackLocalNum

表示rack-local的任务数，nodeLocalNum表示node-local的任务数。从结果来看，Fari-DL的本地性情况是最差的，Fari-PRRL算法和FIFO算法效果差不多。并且在小作业的情况下，Fari-PRRL算法的任务本地性效果更好。

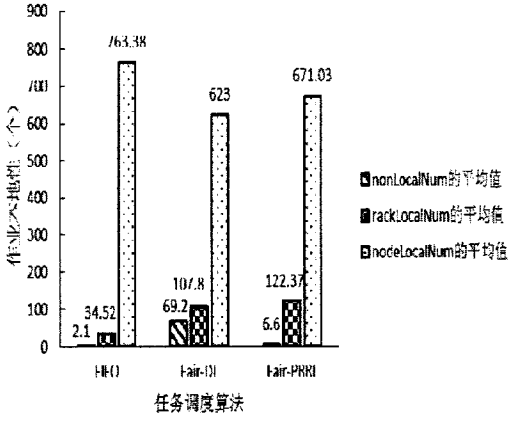
Fari-PRRL算法都是在执行小作业的情况下，Hadoop完成时间和任务本地性情况优于FIFO算法，原因是当小作业较多时，Fari-PRRL算法可以构建出足够大的预释放资源列表，从而有更多的选择可以获取到对当前作业更好的资源。



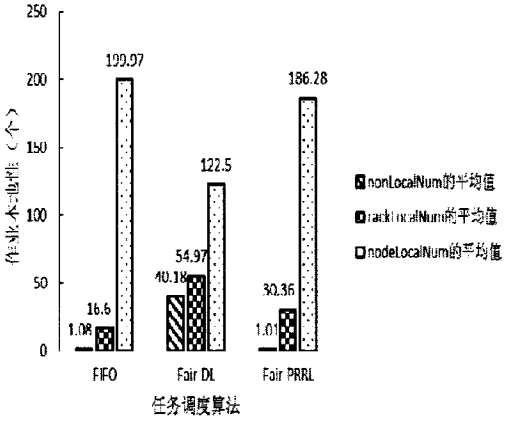
a)小作业的任务本地性



b)普通作业的任务本地性



c)大作业的任务本地性



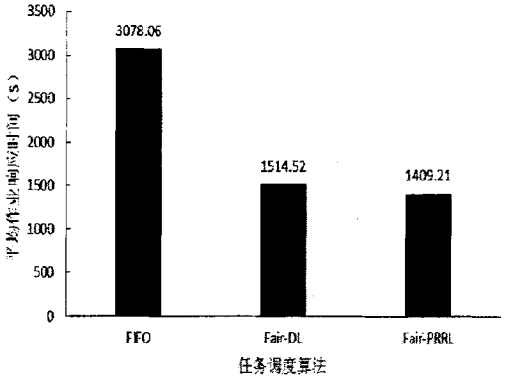
d)混合作业的任务本地性

图 4.4 作业的任务本地性

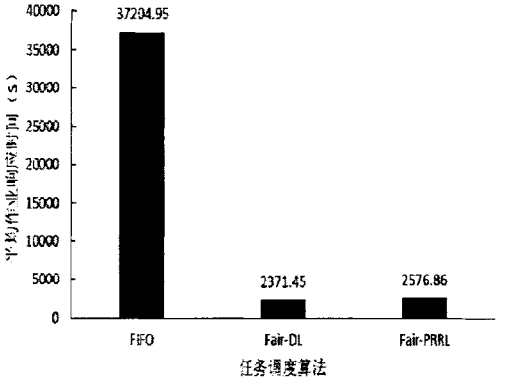
FIFO算法虽然在Hadoop完成时间和任务本地性方面表现很好，但它并不是一个适用多用户、多队列的一个调度算法。因为它逐个地按照作业提交时间执行作业，会导致后面的作业长时间处于等待状态。所以如果是在多用户、多队列的情况下，那么后提交作业的用户将会长时间的得不到反馈，处于后面的队列也是同样的道理。图4.5展示了各个算法的平均作业响应时间。平均作业响应时间是指作业从提交到开始执行所需要的时间。FIFO算法的平均作业响应时间远远大于另外

两个算法。而Fari-PRRL算法和Fari-DL算法则相差不大。

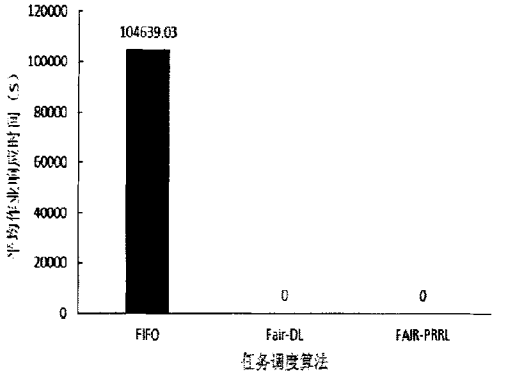
综合以上三个性能指标，本章的Fari-PRRL调度算法无疑是最好的。虽然FIFO算法在Hadoop完成时间和任务本地性方面效果都不错，但Fari-PRRL算法与之相差不大，甚至在小作业较多的情况下，Fari-PRRL算法还略好于FIFO算法。而且FIFO算法不适合于多用户、多队列的情形，也大大限制了它的使用场景。至于Fari-DL算法，Fari-PRRL算法在Hadoop完成时间和任务本地性方面都要明显更好，作业的平均响应时间因为都是采用的公平调度原则，所以相差不大。



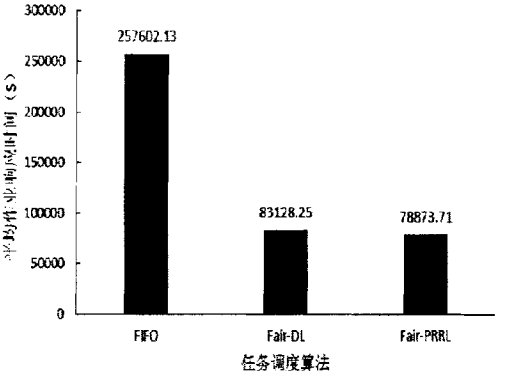
a)小作业的平均作业响应时间



b)普通作业的平均作业响应时间



c)大作业的平均作业响应时间



d)混合作业的平均作业响应时间

图 4.5 平均作业响应时间

4.4 小结

本章首先指出了现有的调度器选择任务时采用的延迟调度策略存在的一些缺陷，并分析了Hadoop的pull通信方式对任务调度的影响。针对这些不足，提出了本章的基于预释放资源列表的任务调度算法。并详细介绍了将本章的算法集成到公平调度器后的实现描述。首先建立了基于预释放资源列表调度的资源三级调度

模型。然后介绍了预释放资源列表的构建过程。最后以伪代码的形式描述了基于预释放资源列表的公平调度算法。并通过一个简单的样例，对比了预释放资源调度、延迟调度和直接调度算法，证明了本章算法在任务本地性和给作业分配更优资源的有效性。实验部分将本章的算法与公平调度算法以及FIFO调度算法进行了实验对比，分别采用了不同大小的作业，从Hadoop完成时间、任务本地性以及平均作业响应时间等方面进行了对比。实验结果表明，本章设计的算法无论在Hadoop完成时间、任务本地性，还是平均作业响应时间，都取得了更好的效果。

结 论

科学技术及互联网的发展，推动着大数据时代的来临。各行各业每天都在产生数量巨大的数据。当前，大数据已经成为数据科学、机器学习与分布式计算等领域最重要的研究主题之一。同时，大数据技术已经在互联网、商业智能、网络与信息安全、医疗服务、智能交通、生物信息等行业广泛应用，并产生了巨大的社会价值和产业影响。但海量的数据使得单个的计算机已经无法满足存储及计算的要求，各种大数据的计算模式与其对应的集群计算系统不断涌现。MapReduce无疑是其中最为经典的大数据计算模式，Apache Hadoop则是MapReduce的开源实现。同时，随着Hadoop的不断改进和完善，以及Hadoop生态圈的不断丰富和成熟，越来越多的企业和机构开始使用Hadoop。目前，Hadoop仍是使用最为广泛的大数据处理系统。任务调度与资源分配一直以来是大规模分布式集群研究的关键技术，这对提高大数据处理系统的计算效率尤其重要。因此，对Hadoop的任务调度与资源分配技术进行研究，具有重要的意义。

Hadoop的主节点和从节点之间是采用pull模型进行通信，主节点只有在接收到从节点发送的心跳信息后，才能以响应的方式向该从节点分配任务。而且各个从节点是在不同的时间向主节点发送心跳请求，使得主节点不能在掌握了当前更多从节点信息的情况下，再做出更优的任务分配方案。现有的任务推测执行算法和任务调度算法都只根据当前请求任务的从节点状况，来选择任务进行分配。而没有分析稍后一些发送心跳信息的从节点状况。

本文主要研究了Hadoop的任务调度算法，以及针对任务执行过程中出现慢任务而采用的任务推测执行机制。并针对上述问题构建了一个预释放资源列表，再基于该预释放资源列表，分别针对任务推测执行算法和任务调度算法设计了相关的改进算法。下面对本文的工作进行总结，并指出下一步研究工作的展望与设想。

1. 论文工作总结

（1）提出了预释放资源列表。

本文提出了预释放资源列表。通过Hadoop记录的历史信息和集群当前状况的监控信息，利用现有Hadoop的任务推测执行机制采用的预测模型，对节点上正在运行的任务的剩余运行时间进行预测，从而可以知道该资源何时会释放。进一步根据节点的历史运行任务的任务进度增长率，预测出该节点运行该任务的可能完成时间。甚至可以再考虑任务在这些资源上数据本地性等信息，更进一步的评估任务在这些资源上的可能完成时间。本文就是将这些预测出的能使任务更快完成的资源组成预释放资源列表。然后将预释放资源列表应用于任务推测执行算法和任务调度算法。

（2）设计了基于预释放资源列表的任务推测执行算法。

Hadoop的任务推测执行算法是通过预测出任务完成时间，找到最慢的任务，然后将最慢的任务在当前空闲的快节点上备份执行，使慢任务能够更快完成。因为Hadoop是通过从节点向主节点以“pull”的方式发送心跳请求获取任务的，当有多个从节点空闲的时候，哪个从节点先向主节点发送心跳请求任务，任务就分配给了这个从节点。但后面发送心跳信息的空闲节点却有可能使慢任务更快完成。甚至当前即将释放的一些资源也有可能使慢任务更快完成。因此，本文提出一种基于预释放资源列表的任务推测执行算法，从构建出来的预释放资源列表中，选择使慢任务更快完成的资源，从而达到使慢任务更快完成的目的。实验表明，基于预释放资源列表的推测执行算法能够有效的使有慢任务的作业更快完成。

（3）设计了基于预释放资源列表的任务调度算法。

本文设计了一种基于预释放资源列表的任务调度算法。现有的任务调度器都只是根据当前发送心跳请求的节点状况，做出任务调度的决策。而本文的调度器根据Hadoop记录的历史信息和集群当前状况监控信息，为每个作业构建一个预释放资源列表。预释放资源列表的构建过程中，充分考虑了作业在各个节点的本地性、作业在节点上任务处理速率的历史信息。通过资源列表与任务列表的匹配调优，可以使得集群中各个作业获得更适合自身的资源。实验表明，本算法能有效提高集群的性能。

2. 进一步工作展望

由于时间仓促和作者的水平有限，本文虽然已经取得了一些工作成果，但仍有许多值得继续研究和可以改进的地方。

（1）Hadoop本身对任务进度的计算还是比较粗糙的，它简单的将任务的执行进度看成是随时间线性增加的；还将Reduce任务的三个子阶段简单的看成是时间均等的。显然这些都是不符合实际情况的。目前很多论文中都提出了改进的任务进度计算方法，但这些方法仍然不能很准确的计算出任务进度。所以，本文的算法仍然采用Hadoop源码中任务进度计算方法。但由于任务进度的计算不准确，难免对本文算法的结果造成影响。下一步可以找到或者提出更准确的任务进度计算方法应用于本算法，从而得到更好的效果。

（2）本文的算法在实现过程中，假设所有任务的资源需求都是同构的，所以就简单的为每一个任务分配一个Slot，并将不同节点上的Slot的资源量也设置为相同的。在Hadoop2.0版本中，Hadoop使用container来对资源进行抽象，用户可主动设置每个作业请求的container资源量需要的CPU数和内存大小。下一步研究中可以将本文的算法应用于资源需求异构的环境。

（3）本文虽然已经在预释放资源列表的基础上分别设计了一种任务推测执行算法和任务调度算法。但算法本身还是比较简单直接的，下一步工作完成可以基

于本文的预释放资源列表，提出更好的匹配调度算法。同时，预释放资源列表的构建过程中，也还可以考虑更多可能对调度有影响的因素，构建出更好的预释放资源列表。

参考文献

[1] Executive Office of the President. Big data: Seizing opportunities, preservin g values. https://www.whitehouse.gov/sites/default/files/docs/big_data_privacy_ report_may_1_2014.pdf, 2014-04-01

[2] 中国大数据技术大会. http://bdtc2015.hadooper.cn/dct/page/1, 2015-12-20

[3] Geethika Bhavya Peddibhotla, Kdnuggets. Gartner 2015 Hype Cycle: Big Da ta is Out, Machine Learning is in. http://www.kdnuggets.com/2015/08/gartner -2015-hype-cycle-big-data-is-out-machine-learning-is-in.html, 2015-08-30

[4] Chen H, Chiang R H L, Storey V C. Business Intelligence and Analytics: From Big Data to Big Impact. MIS quarterly, 2012, 36(4): 1165-1188

[5] Zheng Y, Capra L, Wolfson O, et al. Urban computing: concepts, methodologies, and applications. ACM Transactions on Intelligent Systems and Technology, 2014, 5(3): 38:1-38:55

[6] Silver D, Huang A, Maddison C J, et al. Mastering the game of Go with deep neural networks and tree search. Nature, 2016, 529(7587): 484-489

[7] Ghemawat S, Gobioff H, Leung S T. The Google file system. In: Proceedings of the nineteenth ACM symposium on Operating systems principles. New York: ACM, 2003, 29-43

[8] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. Communications of the ACM, 2008, 51(1): 107-113

[9] Chang F, Dean J, Ghemawat S, et al. Bigtable: A Distributed Storage System for Structured Data. Acm Transactions on Computer Systems, 2008, 26(2):205-218

[10] Vora M N. Hadoop-HBase for large-scale data. In: 2011 International Conference on Computer Science and Network Technology. Harbin, 2011, 601-605

[11] Apache Hadoop 官网. http://hadoop.apache.org, 2016-02-13

[12] 顾荣, 王芳芳, 袁春风, 等. YARM: 基于 MapReduce 的高效可扩展的语义推理引擎. 计算机学报, 2015, 38(1): 74-85

[13] 张继福, 李永红, 秦啸, 等. 基于 MapReduce 与相关子空间的局部离群数据挖掘算法. 软件学报, 2015, 26(5): 1079-1095

[14] Eldawy A, Mokbel M F. SpatialHadoop: A MapReduce framework for spatial data. In: IEEE 31st International Conference on Data Engineering. Seoul, 2015, 1352-1363

[15] Topcuoglu H, Hariri S, Wu M. Performance-effective and low-complexity task

scheduling for heterogeneous computing. IEEE Transactions on Parallel and Distributed Systems, 2002, 13(3): 260-274

[16] Ramamritham K, Stankovic J A. Dynamic task scheduling in hard real-time distributed systems. IEEE software, 1984, 1(3): 65-75

[17] Wu W, Bouteiller A, Bosilca G, et al. Hierarchical dag scheduling for hybrid distributed systems. In: IEEE International Parallel and Distributed Processing Symposium. Hyderabad, 2015, 156-165

[18] 王卓, 陈群, 李战怀, 等. 基于增量式分区策略的 MapReduce 数据均衡方法. 计算机学报, 2016, 39(1): 19-35

[19] Cheng Z, Luan Z, Meng Y, et al. ERMS: An Elastic Replication Management System for HDFS. In: IEEE International Conference on Cluster Computing Workshops. Beijing, 2012, 32-40

[20] 荀亚玲, 张继福, 秦啸. MapReduce 集群环境下的数据放置策略. 软件学报, 2015, 26(8): 2056-2073

[21] Kumar A, Mishra S, Mishra A. Priority with adoptive data migration in case of disaster using cloud computing use style. In: International Conference on Communication, Information & Computing Technology. Mumbai, 2015, 1-6

[22] Keren A, Barak A. Opportunity cost algorithms for reduction of i/o and interprocess communication overhead in a computing cluster. IEEE Transactions on Parallel and Distributed Systems, 2003, 14(1): 39-50

[23] Eric Xing. Big ML Software for Modern ML Algorithms. http://cci.drexel.edu/bigdata/bigdata2014/Ho_Xing_BigData2014_Tutorial.pdf, 2014-08-28

[24] Zaharia M, Chowdhury M, Franklin M J, et al. Spark: Cluster Computing with Working Sets. In: HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. Boston, 2010, 10-10

[25] Apache Flink 官网. http://flink.apache.org/, 2016-04-22

[26] Vavilapalli V K, Murthy A C, Douglas C, et al. Apache hadoop yarn: Yet another resource negotiator. In: Proceedings of the 4th annual Symposium on Cloud Computing. Santa Clara, 2013, 1-16

[27] Running Spark on YARN. http://spark.apache.org/docs/latest/running-on-yarn.html, 2016-03-09

[28] 周家帅, 王琦, 高军. 一种基于动态划分的 MapReduce 负载均衡方法. 计算机研究与发展, 2013, 50(z1): 369-377

[29] Elmeleegy K, Olston C, Reed B. SpongeFiles: Mitigating data skew in mapreduce using distributed memory. In: Proceedings of the 2014 ACM SIGMOD

international conference on Management of data. Snowbird,Utah, 2014, 551-562

[30] Isard M, Budiu M, Yu Y, et al. Dryad: distributed data-parallel programs from sequential building blocks. In: ACM SIGOPS Operating Systems Review. New York, 2007, 59-72

[31] Zaharia M, Konwinski A, Joseph A D, et al. Improving MapReduce Performance in Heterogeneous Environments. In: Proceedings of the 8th USENIX conference on Operating systems design and implementation. San Diego, 2008, 29-42

[32] Ananthanarayanan G, Kandula S, Greenberg A, et al. Reining in the Outliers in Map-Reduce Clusters using Mantri. In: 9th USENIX Symposium on Operating Systems Design and Implementation. Vancouver, 2010, 265-278

[33] 董西成. HADOOP 技术内幕：深入解析 YARN 架构设计与实现原理. 北京：机械工业出版社, 2013, 264-266

[34] Tang S, Lee B S, He B.DynamicMR: A Dynamic Slot Allocation Optimization Framework for MapReduce Clusters. IEEE Transactions on Cloud Computing, 2014, 2(3): 333-347

[35] Chen Q, Zhang D, Guo M, et al. Samr: A self-adaptive mapreduce scheduling algorithm in heterogeneous environment. In: IEEE 10th International Conference on Computer and Information Technology. Bradford, 2010, 2736-2743

[36] 杨立身, 余丽萍. 异构环境下增强的自适应 MapReduce 调度算法. 计算机工程与应用, 2013, 49(19):39-43

[37] Hartigan J A, Wong M A. Algorithm AS 136: A k-means clustering algorithm. Journal of the Royal Statistical Society, 1979, 28(1): 100-108

[38] 李丽英, 唐卓, 李仁发. 基于 LATE 的 Hadoop 数据局部性改进调度算法. 计算机科学, 2011, 38(11):67-70

[39] Guo Z, Fox G, Zhou M, et al. Improving resource utilization in mapreduce. In: 2012 IEEE International Conference on Cluster Computing. Beijing, 2012, 402-410

[40] Chen Q, Liu C, Xiao Z. Improving mapreduce performance using smart speculative execution strategy. IEEE Transactions on Computers, 2014, 63(4): 954-967

[41] Apache Hadoop. Hadoop: Capacity Scheduler. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html, 2016-01-26

[42] Apache Hadop. Hadoop: Fair Scheduler. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html, 2016-01-26

[43] Zaharia M, Borthakur D, Sen Sarma J, et al. Delay scheduling: a simple

technique for achieving locality and fairness in cluster scheduling. In: European Conference on Computer Systems. New York, 2010, 265-278

[44] Gregory A, Majumdar S. A Constraint Programming Based Energy Aware Resource Management Middleware for Clouds Processing MapReduce Jobs with Deadlines. In: Companion Publication for ACM/SPEC on International Conference on Performance Engineering. New York, 2016, 15-20

[45] Hammoud M, Sakr M F. Locality-Aware Reduce Task Scheduling for MapReduce. In: IEEE Third International Conference on Cloud Computing Technology and Science. Athens, 2011, 570-576

[46] Hammoud M, Rehman M S, Sakr M F. Center-of-Gravity Reduce Task Scheduling to Lower MapReduce Network Traffic. In: IEEE 5th International Conference on Cloud Computing. Honolulu, 2012, 49-58

[47] Zhang X, Hu B, Jiang J. An Optimized Algorithm for Reduce Task Scheduling. Journal of Computers, 2014, 9(4): 794-801

[48] Mashayekhy L, Nejad M M, Grosu D, et al. Energy-aware scheduling of mapreduce jobs for big data applications. IEEE Transactions on Parallel and Distributed Systems, 2015, 26(10): 2720-2733

[49] Jaideep. Learning Scheduler. https://issues.apache.org/jira/browse/MAPREDUCE-1439, 2015-05-13

[50] Sandholm T, Lai K. Dynamic proportional share scheduling in Hadoop. In: Job scheduling strategies for parallel processing. Atlanta: Springer Berlin Heidelberg, 2010, 110-131

[51] Tian C, Zhou H, He Y, et al. A Dynamic MapReduce Scheduler for Heterogeneous Workloads. In: Eighth International Conference on Grid and Cooperative Computing. Lanzhou, 2009, 218-224

[52] Tang S, Lee B S, He B. Dynamic Slot allocation technique for MapReduce clusters. In: IEEE International Conference on CLUSTER Computing. Indianapolis, 2013, 1-8

[53] 李婴, 滕飞, 李天瑞, 等. 一种 Hadoop 中基于作业类别和截止时间的调度算法. 计算机科学, 2015, 42(6): 28-31

[54] 宋杰, 徐澍, 郭朝鹏, 等. 一种优化 MapReduce 系统能耗的任务分发算法. 计算机学报, 2016, 39(2): 323-338

[55] 郑晓薇, 项明, 张大为, 等. 基于节点能力的 Hadoop 集群任务自适应调度方法. 计算机研究与发展, 2014, 51(3): 618-626

[56] Apache Hadoop. HDFS Architecture. http://hadoop.apache.org/docs/current/ha

doop-project-dist/hadoop-hdfs/HdfsDesign.html, 2016-01-26

[57] 董西成. Hadoop 技术内幕:深入解析 MapReduce 架构设计与实现原理. 北京：机械工业出版社, 2013, 34-36

[58] Niederreiter H. Quasi-Monte Carlo methods and pseudo-random numbers. Bulletin of the American Mathematical Society, 1978, 84(6): 957-1041

附录 A（攻读学位期间发表的学术论文与获得的成果）

学术论文：

[1] Zhiyong Li, Jing Chen, Shaomiao Chen, Jieqiong Hu. Chemical reaction multi-objective optimization for performance and cost of cloud computing[J]. Natural Computing（在审）

[2] 周润物，李智勇，陈少淼，陈京，李仁发. 面向大数据处理的并行优化抽样聚类K-means算法[J].计算机应用, 2016, 36(2): 311-315

国家发明专利：

[1] 李智勇;陈京;袁廷坤;陈少淼;杨波;李仁发. 基于待释放资源列表的MapReduce任务推测执行方法和装置，申请号：CN201510477121.X（实审）

附录 B（攻读学位期间所参与的项目目录）

科研项目：

[1] 国家自然科学基金项目：面向动态多目标优化的量子 Memetic 计算策略与算法研究，项目编号：61173107

致 谢

能够顺利完成我的研究生生涯，离不开众多给予我帮助的人。在此，向他们表示衷心的感谢。

感谢我的导师李智勇教授。李老师学术严谨，要求很高。三年来每周都坚持在组内召开学术会议讨论。面对频繁的会议，我们也曾感到过疲惫和压力。但是当坚持成为习惯之后，我们收获的不仅是知识，更多的是对自身素养的提高。李老师每天很早来到实验室，也经常很晚都还看到李老师办公室亮着的灯。李老师这种认真勤奋和不知疲倦的工作作风，也深深影响着我，鞭策着我前进。可以说，李老师不仅是我学术上的导师，也是人生方向上的导师。

感谢学校与学院的培养。学校优美的校园必定会使我怀念。感谢学院各位老师对我的关心和栽培，他们宽阔的知识视野和丰富的专业素养都常使我获益良多。特别要感谢学院领导提供的实验室和一批电脑，使我可以顺利完成 Hadoop 集群相关的实验工作。

感谢实验室的各位同门师兄弟们（杨波，陈少淼，乃科，张佳，高松，王东铭，汪维友，袁廷坤，黄滔，戴鑫，曾磊，王静如，王山泉，陈莉，林可），感谢他们在我的科研道路上提供的帮助与宝贵建议，也感谢他们在生活上对我的关心与照顾。

感谢我的家人。感谢他们无微不至的关爱以及物质上的帮助。他们一直都是我奋斗的源动力，以及不断向前的勇气。他们激励着我完成了学生阶段，接下来的工作阶段，我依旧离不开你们的关爱与支持，但我希望此刻自己有能力为你们做的更多。

最后，再次感谢所以帮助过我的人，祝愿他们生活越来越美好。

陈京

2016 年 5 月