密级:_____

中国科学院大学
University of Chinese Academy of Sciences

博士学位论文

分布式天文海量数据处理与控制研究

作者姓名：卫 守 林

指导教师：王 锋 教 授

中国科学院云南天文台

学位类别：理 学 博 士

学科专业：天文技术与方法

培养单位：中国科学院云南天文台

2017 年 5 月

Research on Massive Data Distributed Processing and Control for Astronomy

By

Shoulin Wei

A Dissertation Submitted to

University of Chinese Academy of Sciences

in Partial Fulfillment of the Requirement

for the Degree of

Doctor of Philosophy

Yunnan Observatories

Chinese Academy of Sciences

May, 2017

中国科学院大学

研究生学位论文原创性声明

本人郑重声明：所呈交的学位论文是本人在导师的指导下独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明或致谢。

作者签名：

日　　期：2017年4月8日

中国科学院大学

学位论文授权使用声明

本人完全了解并同意遵守中国科学院有关保存和使用学位论文的规定，即中国科学院有权保留送交学位论文的副本，允许该论文被查阅，可以公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存、汇编本学位论文。

涉密的学位论文在解密后适用本声明。

作者签名：　　导师签名：

日　　期：2017年4月8日　　日　　期：2017年4月8日

学位论文版权使用授权书

本人完全了解中国科学院大学有关保留、使用学位论文的规定，即：学校有权保存学位论文的印刷本和电子版，并提供目录检索与阅览服务；学校可以公布论文的全部或部分内容，可以采用影印、缩印、数字化或其它复制手段保存学位论文。

本人同意《中国优秀博硕士学位论文全文数据库》出版章程的内容，愿意将学位论文提交《中国学术期刊（光盘版）》电子杂志社，编入CNKI学位论文全文数据库并充实到“学位论文学术不端行为检测系统”比对资源库，同意按章程规定享受相关权益。

保密论文在解密后遵守此规定。

论文作者签名：王宇林　　指导教师签名：

日期：2017 年 4 月 8 日

摘 要

天文仪器的制造工艺的提高，高分辨率设备的应用，现代的大型望远镜都面临着海量观测数据处理的挑战。我国明安图射电频谱日像仪(MingantU SpEctral Radioheliograph，MUSER)以高空间、高时间、高频率分辨率对太阳活动进行观测，按照每天 8 小时观测时间计算，每天的原始数据量可达 2.6TB，且包含实时计算和批量计算模式。在 MUSER 海量数据处理的需求背景下，传统的单机多线程、多核并行技术，表现出诸多的局限性。当前一些主流的开源分布式数据处理技术，例如 Hadoop，因为依赖特定存储技术，编程接口，数据处理对象和模式等因素限制了它们在天文数据处理中的应用，并不适合在射电数据处理中。因此，MUSER 海量数据处理就需要设计和实现一套具有高性能、高扩展性，易编程的分布式数据处理框架。

澄江一米红外望远镜（New Vacuum Solar Telescope，NVST）目前已投入观测，由于观测的需要，陆续新增了多个设备，但新增的各类终端设备均是独立工作，导致当前的观测过程严重依赖于人工现场处理，缺少统一的控制与观测调度。因此为发挥 NVST 的作用，必须进一步提高整体系统的自动化、信息化，把当前分离的各个子系统进行整合，实现望远镜自主观测。

因此，本论文以分布式数据处理技术为核心，研究分布式计算框架的设计、分布式计算在 MUSER 中的具体应用和基于 ZeroMQ 的望远镜观测控制系统的网络通信模型设计，论文主要研究工作包括：

1) Spark Streaming 实时计算框架在 MUSER 的准实时的抽样观测数据处理中的应用。针对 MUSER 的需求，设计了自定义的接收器，选择自定义分区方式，等多种方式优化 Spark Streaming 在 MUSER 实时处理中的性能，通过异步执行方式提高处理的稳定性；

2) 面向天文海量数据处理的分布式计算框架（OpenCluster）的设计。使用在天文中广泛应用的 Python 语言编写，提供了简易的编程接口，多类型工作节点，方便天文学家将已有数据处理代码简单快速地扩展成分布式应用，节点失效检查使用心跳机制，支持一主多备提高主节点的高可用性，实现了简单的领导者选举机制。将该框架应用在 MUSER 的实时和批量数据处理中，设计了 MUSER 数据处理的应用界面；

3) 多种集群资源调度研究。深入分析了 Mesos 集群资源管理器，针对设计的分布式计算框架的独立调度模式的不足，结合 Mesos 实现了单任务单框架和集中仓储式调度模式。解决了集群中多种计算框架的资源隔离和共享，优先级的任务调度问题；

4) 基于 Docker 的 CaaS（Container as a Service）构建天文轻量级私有云环境。为提高 MUSER 中长期运行服务的可靠性，使用了 Mesos+Marathon+Docker 组合进行应用的创建

和容器的调度，使用 Kubernetes 容器管理工具创建可靠的 MUSER 长期运行服务；

5) 基于 ZeroMQ 的望远镜观测控制系统的网络通信模型设计。分析了开源天文望远镜控制系统 RTS2 中基于 Socket 网络通信的局限性，根据物联网应用中和望远镜控制系统中都是对设备控制的相似性，讨论了物联网通信协议 MQTT 在望远镜控制系统中适用性，给出了基于 ZeroMQ 的天文望远镜控制中通信部分设计。

论文研究的面向海量观测数据的分布式处理技术和基于 Docker 的轻量级容器云部署应用解决了 MUSER 观测数据的实时、历史积分数据分布式处理和可靠服务应用的难题，实际应用性较强，基于 ZeroMQ 的望远镜观测控制系统的网络通信模型设计为未来观测控制系统的实现打下了良好基础。研究方法也为未来国内外类似射电望远镜海量数据的分布式处理和望远镜观测控制系统的设计提供了参考，具有一定的应用和推广价值。

关键词：海量数据，射电数据处理，分布式计算，资源调度，观测控制

Abstract

Currently, with the advance of manufacture technology and the application of astronomical instruments with high resolution, modern large telescopes are faced with the challenges of massive data processing. MingantU SpEctral Radioheliograph (MUSER) is a synthetic aperture radio interferometer. As a solar-dedicated interferometric array, MUSER is capable of producing high quality radio images with high temporal, spatial, and spectral resolution. If normal observating time is 8 hours per day, the amount of raw data is about 2.6T bytes. In contrast to tranditional historical data processing, MUSER is faced with the requirements of real-time and batching data processing. In the context of massive data processing and management in MUSER, traditional single host with multi-thread or multi-core parallel computing technology perform many limitations. The current mainstream open source distributed computing frameworks, such as Hadoop, because of dependencies on specific storage, complex programming interface, exclusive data object and mode, are not well suited for astronomical data processing. Therefore, it is an urgent need to develop a new platform/framework to accelerate astronomical data processing. The platform/framework has to provide high-speed, scalability and easy programming interface, and can be dynamically tuned to demand of MUSER.

The New Vacuum Solar Telescope (NVST) has began routine observation. Due to the new needs of observation, the new terminal devices have been added to NVST. But these additional terminals are operated independently. There is lack of unified control and observation scheduling. It causes that the current observational procedure relies heavily on artificial operation. In order to play a greater role of NVST, it is nessecary to improve automation and informationization of NVST, and integrate current subsystems to realize automatic observations.

Therefore, this dissertation takes distributed data processing technology as the core focus, and studies design of distributed computing framework, the concrete applications of distributed computing in MUSER and design of network communication model for telescope observation control system based on ZeroMQ. This dissertation mainly includes the following aspects:

1) Applied research on real-time data processing of MUSER based on Spark Streaming. A customized receiver was created for real-time binary stream of MUSER. A customized partition and other ways was used to optimize real-time processing performance. Asynchronous execution was adopted to improve processing stability.

2) Design and implementation of a distributed computing framework for astronomical data

processing. We design a distributed computing framework called OpenCluster, by Python which is widely used in astronomy. OpenCluster provides simple programming interfaces that facilitate astronomers to extend existing codes into distributed applications quickly and easily. A heartbeat mechanism was implemented for node failure check in OpenCluster. A lightweight courtesy method was adopted in leader selection of factories to achive high availability. OpenCluster was applied to MUSER real-time and batch data processing successfully. This dissertation also present the usages and design of web interface for MUSER data processing application.

3) Research on cluster resource scheduling. On account of shortage standalone scheduling model in OpenCluster, we implemented one framework per job and centralized storage scheduling mode which solve the problem of resource isolation and sharing, priority scheduling for multiple computing frameworks in the cluster.

4) Based on Docker to build an astronomical lightweight private cloud environment. To improve reliability of long-running services in MUSER, we used the combination of "Mesos + Marathon + Docker" to dispatch containers. We also deployed an environment with Kubernetes to create reliable long-running services in MUSER.

5) Design of network communication model for telescope observation control system based on ZeroMQ. Firstly, we analyzed the limitation of network communication in RTS2 which is an open source astronomical telescope control system. Secondly, applicability of MQTT in telescope control system is also discussed in term of the similarity of device control between Internet of Thing and telescope control system. Finnally, we presented design of network communication based on ZeroMQ.

In this dissertation, distributed processing technology and lightweight container cloud based docker are used to solve the problem of real-time, historical data processing and reliable services deployment in MUSER. The design of network communication model based on ZeroMQ has laid a good foundation for the realization of future observation control system. The research methods also provides a reference for massive distributed data processing of similar telescopes, and the design of telescope observation control system.

KEY WORDS： Massive data, radio data processing, distributed processing, resource dispatching, observation control

目　录

图目录

表目录

第1章 绪论

科学发现已进入“数据密集型科学”的第四种新模式[1]，这种模式以数据为中心、基于对海量数据的处理和分析去发现新知识为基本特征。伴随着大型望远镜观测设备元器件性能的有效提升，数据的获取能力得到了空前的加强，天文学研究领域也正面临海量数据处理的挑战。如 Sloan Digital Sky Survey(SDSS) (1999-2005)产生的巡天数据达到了 80TB 的数据量；Large Synoptic Survey Telescope (LSST)每晚产生超过 30TB 的数据，10 年的总数据量达到了 30PB[2, 3]；据估计，Low-Frequency Array（LOFAR）的 36 个天线站每天产生的数据超过了 100TB[4]；而最终拥有 3000 个直径为 15 米的抛物面蝶形天线的 SKA[5]，产生的数据更是其五个数量级之上。受摩尔定律的影响[6, 7]，天文数据在未来的时间里仍会继续增长，增长速度也只会更快[8]。

在海量观测数据处理需求的背景下，天文领域迫切需求高可靠，高扩展的分布式计算技术和平台作为支撑。分布式计算研究如何把一个需要非常巨大的计算能力才能解决的问题分成许多小的部分，然后把这些部分配给许多计算机进行处理，最后把这些计算结果综合起来得到最终的结果[9]。面临的关键技术问题包括：并行算法、异构处理平台的分布式协同计算、批处理与实时计算多模式的科学数据处理、高速分布式的消息处理和计算资源统一管理与调度。

一个望远镜系统的正常运转需要多种类型的设备和系统协同工作，如圆顶，CCD，光谱仪等。单一类型的设备也有多个，需要协调工作，如 90 年代怀柔太阳观测基地的多通道望远镜已经配置了多台 CCD。大天区面积多目标光纤光谱望远镜（LAMOST）采用由 32 台 CCD 和 16 台光谱仪组成他们的焦面仪器。因此，研制一套具有较好的扩展性，稳定、可靠的望远镜设备集群控制系统，实现对运行在不同操作系统平台上（Windows, Linux 等）的不同厂商，不同设备类型的统一控制，以最小的代价快速完成新设备的接入，对强化望远镜的维护管理，提升望远镜观测能力无疑具有较好的意义与应用价值。同时结合大数据量分布式处理技术也是解决天文望远镜的观测控制系统基于多信息源和事件驱动的智能决策调度的关键。

1.1 研究背景

近几年来，我国太阳观测设备的开发研制取得了较大的进展。我国明年图射电频谱日像仪(MingantU SpEctral Radioheliograph，MUSER)是由我国天文学家提出的 400 MHz 至 15

GHz 范围内的厘米、分米波日像仪[10]。MUSER 首次在多波段上实现同时以高空间、高时间和高频率分辨率观测太阳活动的动力学性质，对太阳从色球层到高日冕的广大区域中进行高分辨率的三维动态成像观测，填补国际科学界在太阳耀斑、日冕物质抛射等爆发活动的能量初始释放区高分辨射电成像观测的空白，揭示太阳剧烈活动的起源和发生规律，并取得一批原创性的研究成果。MUSER 的投入观测能大大提升我国在太阳活动预报方面的能力，为航空、航天、卫星通讯和国防建设等提供有力保障，对推动学科发展及相关高技术领域的建设具有重要作用。

MUSER 观测地址位于内蒙古锡林格勒盟正镶白旗[11]，分为低频阵（MUSER-I）和高频阵（MUSER-II）。MUSER-I 由 40 面 4.5 米天线及接收设备组成，在 64 个频率点上成像; MUSER-II 由 60 面 2 米天线及接收设备组成，在 528 个观测频点上成像[12]。MUSER-I 每分钟产生 1.92GB 的数据，MUSER-II 每分钟产生 3.6GB 的数据，按照每天 8 小时观测时间计算，每天的原始数据量可达 2.6TB[13]。

海量数据处理已成为射电观测数据的一个显著特点。这种前所未有的海量观测数据的处理问题给各个观测台站带来了巨大挑战，成为当前十分迫切的技术研究方向[14]。互联网技术的蓬勃发展，开源的大数据的分布式数据处理技术已在互联网企业中广泛使用，但适用于天文计算，特别是射电数据处理方面的系统研究工作还较为缺乏。因此，针对日益膨胀的海量天文数据，尤其是 MUSER 这样的即将开始投入使用的设备，迫切需要有更好的解决方案对海量数据进行高速的处理。不但满足现阶段的需求而且考虑到未来的发展，提高观测结果的科研产出率和利用率，最终提高设备的使用价值。

随着先进天文观测设备越来越朝着多功能、复杂化发向发展，基于人工来实现操作与控制进行观测越来越困难，对望远镜观测控制系统（OCS）的要求也就越来越高。目前国外一般认为，观测控制系统的核心是观测调度与协同观测，目的是在 OCS 基础上，通过数据融合与实时数据处理分析方法，集成所有的可用观测设备，根据科学目标的需要选择最优的观测目标，进行有效地自主（自动）观测，以期提高观测数据的获取效率与质量，最终提高科学产出。国内在这方面虽然也开始起步但发展较慢，特别是国内以往的太阳观测设备均以主题观测为特色，对 OCS 的需求不是非常迫切。但随着澄江一米红外望远镜（New Vacuum Solar Telescope，NVST）和 MUSER 相继投入使用，以及太阳物理界正在推动的下一代项目 CGST 提出，OCS 的重要性日显突出。

1.1.1 MUSER 数据处理

MUSER 是采用综合孔径技术方法对太阳进行成像观测的射电望远镜，天线接收的射电噪声信号经过低噪声放大器放大后，再进行电/光转换，将电信号转换为光信号并通过光纤传至机房。经过模拟接收机和数字接收机的处理后，最终生成约定格式的数据并以文件

的方式进行存储，为了实现在线实时状态监控等功能，数字接收机在保存观测数据文件的同时，也可以将部分观测数据通过网络发送到指定地址，供实时数据分析使用。主要硬件由天线、传输、模拟接收机、数字接收机和后续的数据处理及控制设备等部分组成，硬件的结构及数据流如下图 1-1 所示[13]。

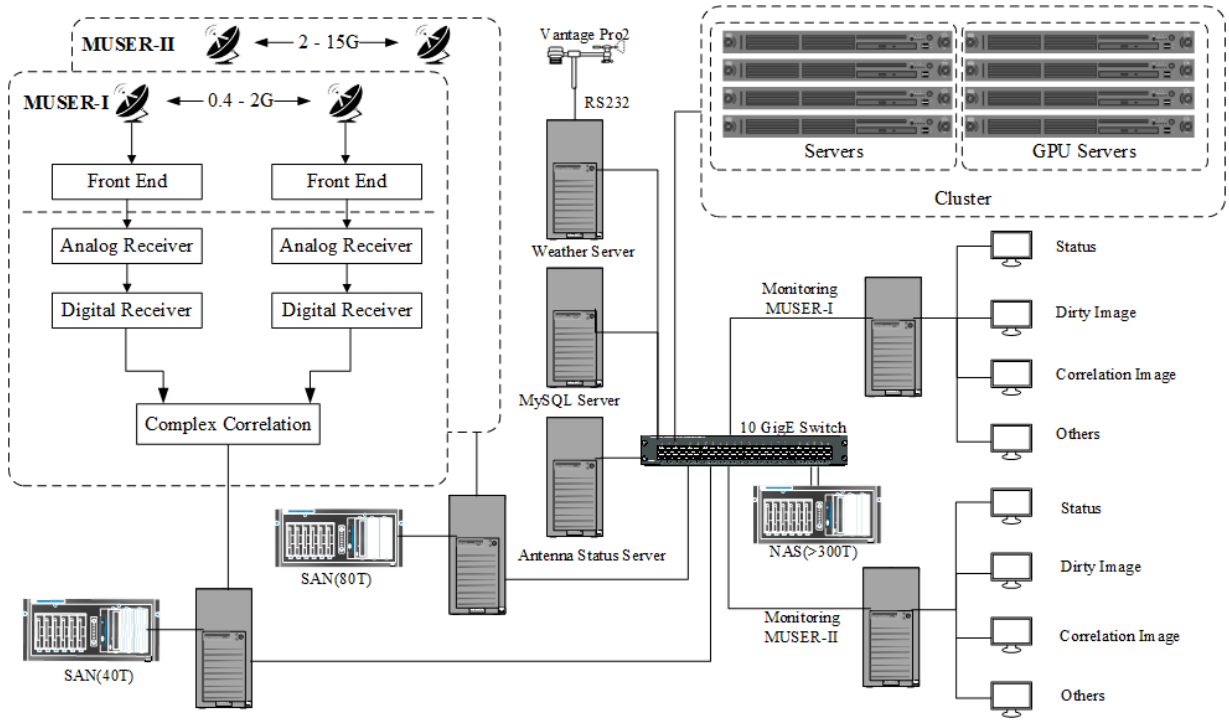

图 1-1 MUSER 硬件及数据流示意图

综合孔径成像原理按信号处理过程大致可以分为如下几步，首先通过分离的单天线接收来自辐射源的射电信号；然后将各天线接收到的信号做互相关处理并积分得到复可见度函数；最后通过对复可见度函数傅立叶反变换，得到源的亮度分布。MUSER 的数据处理采用相似的流程，数据处理流程如下图 1-2 所示[13]。

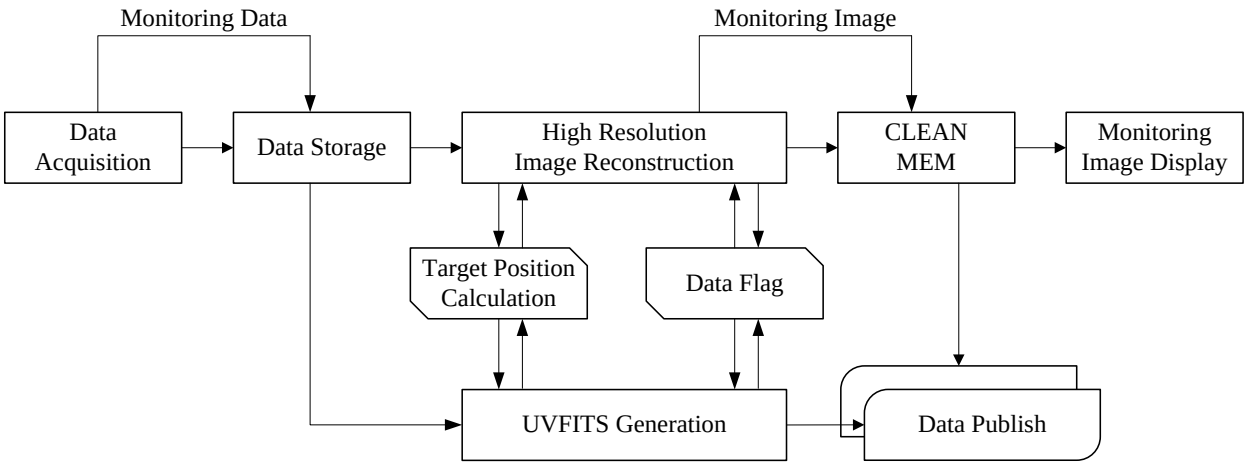

图 1-2 MUSER 数据处理流程图

如上图 1-1 和图 1-2 所示，在 MUSER 的数据处理流程中，涉及了高性能数据检索、

数据存储、实时数据处理、GPU 加速、数据归档和数据共享发布等一系列功能，构成射电数据从产生到成图的一个完整的过程。综合孔径成像原理有许多成熟的理论与实践，但 MUSER 需要的是一个能覆盖全方位从产生到发布的数据处理平台，在这方面，MUSER 有着自己的特殊要求，具体表现为：

1) 数据量大

MUSER-I 包含了 64 个通道，每 3ms 生成一帧数据（100000 字节），每分钟产生 1.92GB 的原始数据；MUSER-II 包含了 528 个通道，每 3.125ms 生成一帧数据（204800 字节），每分钟产生 3.6GB 的原始数据，按照每天 8 小时观测时间计算，每天的原始数据量可达 2.6TB，加上数据处理的中间结果，数据存储量还会成倍增长。

2) 实时计算需求

MUSER 将观测数据数据写入文件存储中，同时产生实时数据流，实时数据流需要经过处理并将结果推送到监控端。这和传统的基于历史观测资料的数据处理有很大的不同。图像处理部分的退卷积的计算过程即使在 GPU 上操作仍是非常耗时的，低频部分的实时数据每 5 秒产生 8 帧数据，需要产生 128 个脏图，再加上洁化和成图的计算过程，这对实时性要求较高的实时发布是一个很大的挑战。

3) 计算过程复杂

原始数据的格式转换、GPU 上的成图都需要多个处理步骤，计算过程复杂，同时要与多个系统交互协作完成。如数据校验过程中要考虑仪器状态、天气状态，数据处理中需要查询星表位置数据等，而这些状态数据分布在不同的存储系统中。

4) 异构计算节点

集群中包含了普通的节点和带有 GPU 卡的节点，而图像处理等操作需要在 GPU 节点上执行，多环境，异构平台导致应用配置维护难。因此，在分布式资源调度中，需要考虑不同类型的硬件资源调度，最优化的利用不同硬件的优势。

5) 存储模型多样

MUSER 数据处理中，除了原始数据文件，还需要取得仪器状态，光纤延迟，天线位置，气象条件，星表位置等多种数据，而这些数据来源不一，存储格式不同，即包含结构化数据又包括非结构化数据，有使用关系型数据库来存储，也有使用 NoSQL，还有文件系统，这些多种类型数据的并存增加了存储系统的复杂性，也增加了数据查询、管理的难度。

6) 多样的数据访问模式

与数据处理相关的部件有如下几部分：原始文件存储服务器、普通计算集群、GPU 集群、原始文件索引服务器、归档服务器、天气仪器状态采集服务器、Web 发布服务器和实时监控服务器等，这些部件都需要与数据处理系统进行数据交互，这些部件能否提供持续高速访问也是影响数据处理效率的关键因素。

7) 高扩展性

观测数据处理不但要求能够满足当前的数据观测，还要适应未来一定时间的观测需求，随着历史数据的积累，数据量持续增长，这就意味着数据处理系统要能够满足这些数据处理量的动态性特征，以及对未来投入新的计算节点的扩展需求。所以数据处理系统要能够适应数据容量和计算节点的伸缩，具备高扩展特性。

综上，MUSER 在如此高速产出并且兼具实时数据和历史数据的多种处理模式，同时需要融合多种数据来源，运行在异构的处理节点上，对海量数据进行高速的处理，就成为一件非常困难的事。分布式计算技术从体系结构上为解决这个问题提供了一种可行的方法，在分布式和云计算在互联网领域如火如荼发展的今天，开展面向海量天文数据处理的分布式计算技术研究，符合技术的发展趋势，针对性强，应用点具体，也具有一定的前瞻性，具有明显的研究与应用价值。

1.1.2 NVST 观测控制

NVST 位于云南天文台抚仙湖太阳观测基地，是“我国 21 世纪初主要的地面光学及近红外太阳观测设备”。其有效口径为 1 米，真空窗直径达到了 1.2 米，是目前世界上最大口径的真空望远镜，可在 0.3～2.5 微米波段对太阳进行高分辨率成像和多波段光谱观测，测量太阳磁场的精细结构、高时空分辨率的演化过程。

受限于前期有限的技术储备，NVST 在软件方面的建设一直滞后，整个望远镜系统缺少统一的控制与观测调度，新增的各类终端设备均是独立工作，导致当前的观测过程严重依赖于人工现场处理。从 NVST 的观测控制、数据采集及后续的数据处理需求分析来看，当前 NVST 面临的观测控制问题总结归纳如下：

1) 整合与完善

OCS 受限于技术与经费，根据观测结果实现最优控制始终未能开展。NVST 当前观测过多地依赖于人工辅助观测，差错率高，虽然可以观测但不利于未来的发展，迫切需要一个具有自动、智能的观测控制系统，降低观测人员工作量与难度，提高观测效率。国际上其它望远镜在建成初期实际也面临类似的问题，在建设的初期以基本功能实现为主，在试观测或观测期间才开始逐级完善。

2) 新设备集成与集中控制

天文望远镜所配套的观测终端设备是发挥一个天文望远镜作用与提高观测产出的核心。随着 NVST 的发展，更多的新一代观测终端设备给当前 NVST 的观测带来了发展的契机。但在观测中如何充分发挥这些终端的使用是困扰 NVST 的一个问题。实现越来越多的终端设备与越来越复杂的观测过程的统一，将复杂困难的技术与管理问题封装在幕后，使得观测人员可以以简单的方式进行复杂的观测成为当前 NVST 最为迫切的问题。比如，观

测员知道观测波长，色散要求，面源还是点源后，后台系统就可以自动选择合适的光谱仪，决定是否启动狭缝扫描及 AO。显然，这样的自然观测模式也是现代望远镜技术的追求与发展趋势。

3) 缺少观测过程联动与自动自主观测能力

现代望远镜的一个重要特色就是自主联动观测。NVST 投入观测以来，其观测目标的确定与观测还基本靠人工实现，与 ONSET 等其它望远镜观测结果的联动还未全部实现。同时，当前观测过程中与 Seeing 观测设备、气象站数据的联动一直没有完成。即使在 NVST 的系统中，各个子系统（或终端设备）间所需要的联动与互锁过程还没有实现自动化，“观测靠手、通信靠吼”经常是现场观测过程的真实写照。NVST 未来的发展，必须进一步提高整体系统的自动化、信息化，把当前分离的各个子系统进行整合，实现在信息融合前提下望远镜自主观测，这样才可以进一步发展 NVST 的作用。

4) 没有统一的平台软件标准与数据处理体系

作为一台大口径的太阳光学望远镜，NVST 提供了一个优秀的光学平台。针对未来发展需要，NVST 必然需要根据科学目标的不同，在望远镜这一平台上实现对应的数据处理 PIPELINE 和相应的数据处理模块。同时确保在功能扩展与模式扩展的同时，系统仍具有良好的可用性，确保望远镜的正常、稳定观测。这样的需求，就必然要求 NVST 有一个系统的底层软件设计，实现一套完整的数据交换标准与统一控制接口。

综上所述，完成了基础建设的 NVST 已经具有了良好的基础，其观测成果被国内外众多科学家们所期待。但当前的整体仪器仍面临观测过程严重依靠人工，联动观测尚无法开展，整体系统集成度偏低，可用性偏弱等问题，严重制约了 NVST 观测效率与科学产出能力。参考国外大型望远镜的发展与信息系统建设经验，当前在国内开展观测控制系统关键技术的研究并在 NVST 上应用需求迫切。

1.2 研究意义及价值

当前天文数据处理技术已经进入天文信息学时代，以海量数据为基础的现代天文观测迫切需要先进、智能的处理手段来加快数据的处理，加快科研进程。MUSER 已经投入试观测，当前的天文数据高性能处理技术已无法满足 MUSER 海量数据处理的需求，研究并解决 MUSER 数据的高速处理需求，才能确保 MUSER 实现从试观测阶段到常规观测阶段的顺利过渡，进而充分发挥其作为新一代太阳射电观测设备的重要作用。

本论文的研究工作以 MUSER 的海量射电观测数据为对象，以分布式体系为核心，主要运用基于内存的分布式计算技术，资源调度，容器等技术研究大型望远镜海量天文数据分布计算中的关键问题。本研究实现了具有高性能、良好扩展性、简单编程接口的分布式

计算框架，从而从根本上为高速海量天文数据的分布式计算提供了新的解决方案。除了满足 MUSER 日常观测数据处理和科学产出的需要。同时，项目成果也将为并行处理和分布式技术在太阳射电天文中的应用打下基础，为国内大型天文仪器的海量数据的分布式处理提供重要的技术借鉴。

天文望远镜的建设是一个复杂的系统工程，从项目的提出到最终的建成可能凝聚几代人的心血。NVST 自 2012 年进入试观测以来，所获得的优异观测资料得到了国内外同行的广泛关注，如何进一步提高与保障 NVST 的观测能力成为 NVST 今后发展的一个共识。在现阶段，有必要分析与借鉴国外先进望远镜的观测控制系统，针对 NVST 的观测控制要求，研究并实现一套开放、易扩展、可定制的 NVST 观测控制系统，突破其中的关键技术，对 NVST 无疑具有十分重要的意义。

针对在 NVST 建设阶段较为薄弱的观测控制系统，通过对其关键技术的研究，最终实现一套具有智能、高效、便捷、鲁棒的的 NVST 观测控制系统，实现异构设备的统一集成与调度，实现高效的自动化观测调度。显然这会有效地降低广大天文研究人员与观测人员的工作难度，加快科研产出，更好地利用 NVST 进行天文研究和发现。

本论文的另一项主要研究就是以实现高效、多通信模式的观测控制底层通信架构为目标，分析多种通信协议在观测控制系统中的适用性，从基础理论方面全面的掌握当今观测控制系统的设计原理、核心代码与分布式计算模型，并针对 NVST 的仪器特色对关键技术进行突破，未来最终设计的系统可以满足 NVST 的智能、高效、便捷、鲁棒的观测控制需要。本项目虽然针对 NVST 进行研究，但关键技术均可以应用与直接移植到 LAMOST 和 MUSER 系统，也可以为未来我国下一代大型望远镜（如 CGST）的信息系统与观测控制提供借鉴，研究意义明显。

1.3 论文研究范畴

本论文研究内容可分为两个部分，一个是分布式计算在天文射电海量观测数据处理上的应用，另一个是望远镜观测的远程控制系统。通过本文的研究，拟解决 MUSER 的海量数据分布式高性能处理以及 NVST 在观测控制方面的需求。整篇论文研究如何解决当前天文海量数据的高性能处理、可扩展问题、集群资源调度及望远镜观测控制系统的底层网络通信问题。也包括这些研究过程中的一些基础和容易忽略的问题，例如可扩展的任务优先级的研究。在论文中多处提到天文数据的处理和分析，但是天文数据处理算法本身不是本论文重点需要研究的内容。MUSER 数据处理涉及到多个数据存储系统，如气象数据，仪器状态，星表数据等，对这部分数据的存取，也不是本论文重点研究的内容。在第 3 章中 Spark 在 MUSER 中应用研究，主要是研究 Spark Streaming 在 MUSER 实时处理的应用。

在第 8 章中提到的观测控制系统 CTS，目前处于设计和代码的编写阶段，尽管在论文中提到了 MQTT 协议的分析和设计，但最终在 CTS 的网络通信设计中使用的是 ZeroMQ。另外，观测控制系统（OCS）与望远镜控制系统（TCS）易被混淆，在本文中我们定义 OCS 是比 TCS 更高的层次，OCS 研究的重点在于如何实现数据的融合进而进行观测调度决策，TCS 的核心任务是底层的望远镜控制。OCS 与 TCS 之间通过 ZeroMQ 的协议进行通信。

1.4 论文主要研究工作

本论文结合我国明安图射电频谱日像仪（MUSER）的海量数据处理需求和澄江一米红外望远镜（NVST）的观测控制系统的需求，进行了如下的研究：

1) 针对 MUSER 的海量数据处理需求，对当前在互联网领域广泛使用的分布式计算框架进行了分析并进行多个方面的测试。从 MUSER 数据分布式处理的需求的角度对这些计算框架的可用性进行了深入的分析，具体将 Spark Streaming 实时计算框架应用在 MUSER 的准实时的抽样观测数据处理中；

2) 设计了一个面向天文海量数据处理的分布式计算框架。设计的思想基于传统工厂和车间的商品制造流程，针对于互联网应用中的开源分布式计算框架在天文应用中的诸多困难，提供了简易的编程接口，将天文学家已有数据处理代码快速地扩展成分布式处理，对分布式计算中的心跳机制、高可用性、服务提供等技术进行了研究和分析。将该计算框架运用在 MUSER 的实际数据处理中，设计了 MUSER 数据处理的应用界面；

3) 为了充分利用集群的资源，为不同的任务分配不同资源，解决集群中多种计算框架的资源隔离和共享，就需要一个工具来进行整个数据中心资源的管理和分配。本文深入分析了 Mesos 集群资源管理器，结合 Mesos 使设计的分布式计算框架支持多种资源调度模式，使集群的资源更有效地调度和隔离、解决了优先级的任务调度问题；

4) 为进一步提高物理资源的利用率，以及业务部署和交付的效率，提高 MUSER 中长期运行服务的可靠性。研究了基于 Docker 的 CaaS（Container as a Service）构建天文轻量级私有云环境，使用了 Mesos+Marathon+Docker 组合进行应用的创建和容器的调度，以及 Kubernetes 容器管理工具创建可靠的 MUSER 长期运行服务；

5) 一个完整的望远镜系统的正常运转离不开多种设备、组件之间相互配合，因此网络通信将是观测控制系统的重要底层支撑。论文中分析了开源天文望远镜控制系统 RTS2 中基于 Socket 网络通信的局限性，根据物联网应用中和望远镜控制系统中都是对设备控制的相似性，讨论了物联网通信协议 MQTT 在望远镜控制系统中适用性，给出了基于 ZeroMQ 的天文望远镜控制中通信部分设计。

1.5 论文章节安排

本论文的研究可以概括为两方面：基础性研究工作和应用性研究工作。第 4 章、第 6 章偏重于基础研究；而第 3 章、第 5 章、第 8 章和第 8 章则是应用性的研究。论文围绕 MUSER 面临的分布式数据处理和 NVST 的观测控制相关技术展开讨论，以如何解决海量天文数据的分布式计算的高性能、可靠性问题和观测控制系统实现为主线，重点放在海量数据分布式计算的关键问题的研究上。

论文在组织结构上，把各个相关研究点分散到各个章节中，每个章节可以独立，章节开头说明该章主要研究问题，章节的结尾总结该章节的工作。各章节的主要内容如下：

第 2 章从 MUSER 对数据处理的需求角度出发，对 MUSER 所涉及的与分布式有关的技术，进行了一个整体的回顾，探讨了在天文领域中的具体应用，也对当前的一些前沿分布式计算问题进行了讨论，调研了当前天文上的分布式计算和望远镜观测控制的相关技术；

第 3 章选取开源的分布式计算框架 Spark 来研究开源系统对天文数据分布式计算的支持，尤其是如何满足高性能的计算以及扩展性。针对 MUSER 的实时数据处理要求，使用 Spark Streaming 从性能和可扩展方面进行测试分析和研究；

第 4 章在分布式计算框架研究的基础上，针对天文分布式计算领域需求设计了面向天文计算的轻量级可扩展的分布式计算框架——OpenCluster，从设计思路，关键技术，编程接口，部署等多方面介绍了 OpenCluster；

第 5 章研究了 OpenCluster 在 MUSER 的历史数据和实时数据处理中的应用，如何使用 OpenCluster 提供的简单的编程接口，集成已有的代码，运行在分布式系统中；

第 6 章在前面章节的研究工作的基础上，讨论如何更有效的利用集群资源，使用 Mesos 使 OpenCluster 支持粗粒度和细粒度的任务调度，解决集群资源统一调度，结合 Kafka 实现任务优先级调度问题；

第 7 章讨论了 OpenCluster 对 Docker 的支持，以及 Docker 使用过程中的镜像构建，容器编排与调度的关键技术。研究了在 MUSER 的数据处理中，如何使用 Mesos 和 Kubernetes 平台下的容器编排和调度；

第 8 章主要研究研究基于 ZeroMQ 的天文望远镜控制中通信系统设计与实现。讨论了开源天文望远镜控制系统 RTS2 中基于 Socket 网络通信的局限性和物联网通信协议 MQTT 在望远镜控制系统中适用性，给出了基于 ZeroMQ 的天文望远镜控制中通信部分设计；

第 9 章，是整个论文研究工作的总结及未来工作的展望。

第2章 研究现状与趋势

太阳射电观测中已经有较成熟的数据处理方法可以借鉴，但具体的数据处理过程以及相应的处理算法设计和参数选择都与具体的设备紧密相关。针对 MUSER 的结构设计、观测精度和模式，需要研究相应的误差模型和数据处理算法，特别是适合 MUSER 的分布式计算框架的设计。在开源社区，已有成熟的分布式计算框架（Hadoop，Spark，Flink 等），在天文数据处理领域中，也有使用这些开源系统作为大规模图像处理的基础框架。但更多使用的是基于 MPI 的并行计算，或者内部实现的分布式计算框架作为项目组内高性能计算数据处理流水线的主要技术。

观测控制系统(OCS)的内涵随着天文望远镜技术的发展而不断地发展变化。目前国外一般认为，观测控制系统的核心是观测调度与协同观测，目的是在 TCS 基础上，通过数据融合与实时数据处理分析方法，集成所有的可用观测设备，根据科学目标的需要选择最优的观测目标，进行有效地自主（自动）观测，以期提高观测数据的获取效率与质量，最终提高科学产出。

本章围绕解决大型望远镜海量观测数据的分布式处理问题，对当前涉及分布式高速数据处理的诸多方面，例如对解决天文计算中高性能计算，并行计算、分布式计算、资源调度、影响计算性能的各个因素和相关理论基础等进行全面的回顾，并对前沿海量数据的分布式计算技术、框架，以及在天文中的具体应用进行调研分析；对国内外研究望远镜观测控制系统应用的现状进行分析，以及讨论了观测控制系统相关支撑技术的发展现状。

2.1 高性能计算

高性能计算作为计算机科学的一个分支，致力于开发高性能计算机和运行在高性能计算机上的应用软件。高性能计算系统利用大量处理单元的聚合计算能力来满足应用巨大的计算需求，其关键问题是实现众多计算节点的大规模集成和高效协同计算，核心技术涉及高性能计算机和大规模并行应用程序。回顾历史，高性能计算作为一个强大的计算工具，与科学研究的发展密不可分。一方面，科学研究对计算能力永无止境的需求促进了高性能计算技术向前发展；另一方面，高性能计算技术的每一次巨大进步都为科学研究提供了全新手段[15]。在当前核爆模拟、天气预报、工程计算、火箭发射和医学等众多领域，都面临着海量数据带来的挑战，高性能计算已被广泛应用于这些领域。

2009 年 11 月，在第 34 届高性能计算机 500 强的排名中，CRayXT5 超级计算机“Jaguar”，

最终以 2331Tflops 的峰值运算速度和 1759Tflops Linpack 测试值，战胜了上届排名第一的 IBM“Roadrunner”超级计算机，荣登榜首。曙光 5000A 和“天河一号”的问世，使我国一跃成为高性能计算的强国。第一台国产超千万亿次的超级计算机“天河一号”，在 2009 年 10 月正式亮相，峰值速度为 1206.19Tflops，Linpack 实测速度为 563.1TFlops，使“天河一号”计算机在最新的全球高性能计算机 500 强中占据了第五的位置，在当时成为中国超级计算机之首[16]。继“天河一号”后，由中国国防科技大学与浪潮公司共同研发的“天河二号”超级计算机，于 2013 年 6 月 17 日的国际超级计算机大会(International Supercomputer Conference)发布全球最快的 500 台超级计算机排行榜上位列第一[17]。在 2016 年 6 月 20 日，国际超级计算机大会发布了全球最快的 500 台超级计算机排行榜，由国家并行计算机工程技术研究中心研发的“神威太湖之光”超级计算机以超第二名近三倍的运算速度夺得第一[18]。

由西澳大学 ICRAR 研究所 SKA 技术团队牵头包括上海天文台参加在内的国际联合团队研发了一款数据流管理系统，取名为 DAliuGE，DAliuGE 的最终目的是为 SKA 科学数据处理器提供了一个高效的分布式数据管理平台和具有良好拓展性的管线系统执行环境。目前已在“天河二号”超级计算平台上成功部署了 SKA 数据流管理系统 DAliuGE 并完成了 1000 计算节点的大规模集成测试[1]。

这里的高性能计算主要是指高性能的超级计算机，毫无疑问，高性能计算作为世界高技术领域的战略制高点，已经成为科技进步的重要标志之一，在国民、生产、军事、科学等多领域都能发挥重要作用，但受数据传输、成本、编程模型等因素制约，并不适合在天文望远镜的数据处理中。而现在高性能计算，云计算，大数据正呈融合的趋势，硬件资源上也将融合 GPU[19]、FPGA[20]等技术提供更高性能的计算环境。

2.2 并行计算

并行计算是解决大规模数据处理的方法，其思路是将大量数据分散到多个节点上，将计算并行化，利用多机的计算资源，从而加快数据处理的速度。MPI[21] 作为目前国际上最流行的并行编程环境之一，因其良好的可移植性和易用性、完备的异步通信功能等优点，而在集群高性能计算中得到广泛应用。在基于 MPI 编程模型中，计算任务是由一个或多个彼此间通过调用库函数进行消息收、发通信的进程所组成。绝大部分 MPI 实现在程序初始化时生成一组固定的通信进程，这些进程在不同的节点上运行（通常一个处理器一个进程），执行着相同或不同的程序，以点对点通信或者集合通信的方式进行进程间交互，共同协作完成同一个计算任务。并行计算主要是通过消息传递的技术实现，包含用于共享内存编程

[1] http://www.yicas.cn/portal.php?mod=view&aid=2801

的 OpenMP 和用于分布式内存编程的 MPI。

OpenMP 是一种面向共享存储体系结构的多线程并行编程接口，所有处理器都被连接到一个共享的内存单元上，处理器在访问内存的时候使用的是相同的内存空间[22]。OpenMP 在并行执行程序时，串行区域由主线程执行，并行程序通过派生多个线程来执行，每个线程通过局部栈指针能够拥有“私有”的变量。因其具有基于指令的简单接口和逐一处理程序中的循环而不必大量重构代码的增量式的并行化方式，能极大地降低串行程序并行化的编程难度。在天文数据处理中，OpenMP 也被广泛使用。比如，使用 OpenMP 技术提高图像相减测光算法的效率[23]，基于 OpenMP 对天文学软件 Gridding 的优化方法[24]，在 LOFAR 中使用 OpenMP 在多核 CPU 和 GPU 上作波束合成的性能对比[25]。

MPI 是面向分布式存储体系结构的消息传递接口标准，已成为并行编程模型的事实标准。MPI 消息传递的并行编程是基于大粒度的进程级别上并行，具有最好的可移植性，几乎被当前流行的各类并行机所支持，且具有很好的可扩展性。但是，消息传递并行编程只能支持进程间的分布存储模式，即各个进程只能直接访问其局部内存空间，而对其它进行的局部内存空间的访问只能通过消息传递来实现。目前广泛使用的是 MPICH，MPICH 是 MPI 的一个应用实现，支持几乎所有操作系统平台。在天文领域中，我们使用 MPI 对 MUSER 的 UVFITS 的合成作了性能优化和测试[26]。乌鲁木齐天文站依托现有的南山 25m 射电望远镜和 VLBI 记录终端 MK5A 系统，开发的相干消色散处理软件，也使用了 MPI[27]。NVST 实现了基于 OpenMP 和 MPI 的高分辨重建[28]。

基于 OpenMP 和 MPI 编程模型已被广泛应用于天文的图像处理或者算法加速中，但并不适用于建立一个完整的分布式的数据处理平台，OpenMP 采用共享存储，因此不适合集群，而 MPI 编程模型复杂，具有较高的实现难度。现代的并行计算也不是单纯只使用一种并行模型，而是结合共享内存类型和分布式存储类型各自的优点，建立混合模型的并行系统。

2.3 分布式计算

面对海量的观测数据，采用提高 CPU 性能和改用更大容量的内存和磁盘的做法，已经变得越来越困难。在这种背景下，以网络和网络通信技术为依托，将分散在不同地理位置的计算机连接起来，组成空间上分散、逻辑上统一的数据存储和计算集群，成为当前实现大规模数据处理的主要选择。集群计算的优势在于它强调总体的处理能力，每台计算机做为单个节点参与计算过程，承担其中一部分计算任务，处理能力的强弱由全部节点共同决定。集群在运行过程中可以动态地调整自己的计算能力，赋与了集群计算近乎无限增长的可能，这是传统的集中式计算无法比拟的。

目前，谈到分布式计算，就不得不提 Apache Hadoop，Hadoop 是一个能够对大量数据进行分布式处理的软件框架，是目前最流行的大数据处理框架[29]。新版 Hadoop 主要包含分布式文件系统 HDFS、资源管理器 YARN 和批处理引擎 MapReduce。Hadoop 以一种可靠、高效、可伸缩的方式进行数据处理，它实现了并行与分布式计算 MapReduce 的编程思想[30]。同时 Hadoop 也不仅仅是一个框架，而是已经演变为一种分布式计算的生态圈，包含了多种计算框架，比如可伸缩的分布式迭代图处理系统 Giraph[31]，大规模的科学计算 Hama[32]，机器学习 Mahout [33]。

Spark 是 UC Berkeley AMP Lab 所开源的类似 Hadoop MapReduce 的通用的并行计算框架，与 Hadoop 的 MapReduce 引擎基于各种相同原则开发而来的 Spark 主要侧重于通过完善的内存计算和处理优化机制加快批处理工作负载的运行速度，采用 Scala 语言实现，提供类似于 DryadLINQ 的集成语言编程接口，使用户可以非常容易地编写并行任务[34]。拥有 Hadoop MapReduce 所具有的优点，但不同于 Hadoop 的是 Spark 的 Job 中间输出和结果可以保存在内存中，从而不再需要读写 HDFS，因此 Spark 能更好地适用于数据挖掘与机器学习等需要迭代的 MapReduce 算法[35]。除了支持 MapReduce 的批处理计算，它还支持流式计算，SQL 查询，机器学习和图表数据处理。Spark Streaming 基于微批量方式的计算和处理，可以用于处理实时的流数据。它使用 DStream，简单来说就是一个弹性分布式数据集（RDD）系列，处理实时数据[36]。

Apache Flink[37]同样也是支持批量和流式处理的基于 Java 实现的通用大数据分析引擎，提供了基于 Java 和 Scala 的 API，Python 的 API 也正在测试中。和 Spark 类似，Flink 利用基于内存的数据流并将迭代处理算法深度集成到了系统的运行时中，这就使得系统能够以极快的速度来处理数据密集型和迭代任务[38]。支持 StandAlone 独立集群和 YARN 的集群部署模式，生态圈也在正在发展中。

Apache Samza[39]是一种与 Apache Kafka 消息系统紧密绑定的流处理框架，并使用资源管理器 Apache Hadoop YARN 实现容错处理、处理器隔离、安全性和资源管理。Samza 非常适用于实时流数据处理的业务，它能够帮助开发者进行高速消息处理,同时还具有良好的容错能力。在 Samza 流数据处理过程中，每个 Kafka 集群都与一个能运行 Yarn 的集群相连并处理 Samza 作业。

目前，流式处理也是分布式计算中的一个重要分支，除了 Spark Streaming，Flink，Samza，还包括 Google Dremel、Apache Drill、Apache Storm 以及 Apache S4 等框架[40, 41]。

基于 MapReduce 的分布式计算框架已被广泛应用于科学数据处理[42-44]，天文数据处理中，Hadoop 也被用于图像叠加处理[45]，星表交叉认证[46]，使用 Spark 处理图像源信息提取[47]，AstroSpark 是在 Spark 的基础进行扩展的适合天文数据处理的计算框架[48]，实时数据处理也被广泛应用于射电数据处理中[49]。

开源的分布式计算框架可以为我们提供在互联网应用中久经考验的处理模型，但很多天文或者科学计算领域的算法并不适合用 MapReduce 描述[50]，另外 MapReduce 的编程模型学习曲线较长，特别是对于很多天文学家已经编写好的算法，很难使用 MapReduce 模型实现，或者具有较大的工作量，这是限制这些框架在天文领域中被广泛使用的主要原因。

2.4 统一资源管理

分布式计算，是在一批普通配置的机器，也就是集群，将计算任务拆分到各机器上计算，然后汇总结果。在一个集群中，存在很多资源需求各异的分布式应用，集群是分布式计算的基础，为了充分利用集群的资源（譬如为不同的应用分配不同资源，按任务优先级分配资源等），我们就需要一个工具来进行整个数据中心资源的管理、分配等，而这个工具就是集群资源调度管理工具，目前被广泛使用的是 Apache 基金会的 YARN 和 Mesos[51]。

2.4.1 YARN

YARN[52]是源于管理 Hadoop 规模的需求。在 YARN 出现之前，集群资源管理集成在 Hadoop MapReduce V1 架构中，为了有助于 MapReduce 的扩展而将其独立到 YARN 中实现，采用资源调度器取代原有的任务调度器，YARN 的体系结构如下图 2-1[53]所示，资源管理器（Resource Manager，RM）负责对各个节点管理器（NodeManager，NM）上的资源进行统一管理和调度，NM 负责每个节点上资源和任务的管理，定时向 RM 汇报本节点的资源使用情况。提交应用时，需要提供一个跟踪和管理这个程序的应用程序主控节点（ApplicationMaster，AM），由它向 RM 申请资源，并要求 NM 按 AM 申请到的 Container 资源信息来启动任务[54]。

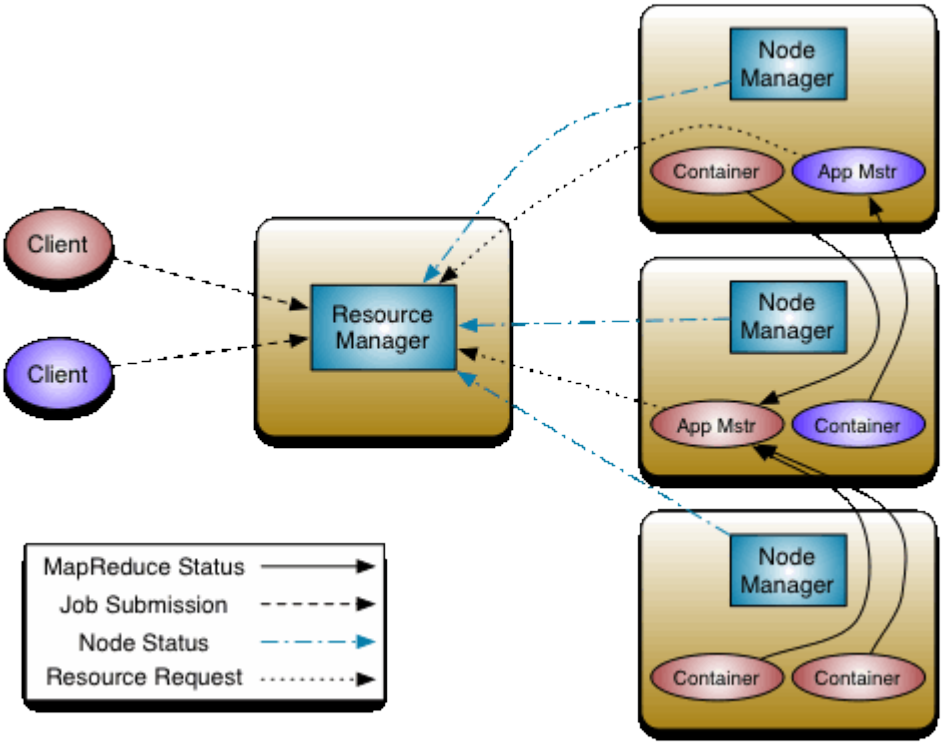

图 2-1 YARN 体系结构

资源调度机制是 YARN 资源调度器的核心。YARN 资源调度器采用的调度机制主要包括：双层资源调度机制、层级队列管理机制、资源保证和资源抢占机制。其中双层资源调度机制是其核心，是 YARN 进行资源分配的总体架构；层级队列管理机制是 YARN 对上层资源分配队列的管理方式；资源保证和资源抢占机制是 YARN 保证任务资源需求的机制[54]。

2.4.2 Mesos

Mesos 和 YARN 类似，是属于第二代的双层调度管理器，Mesos 诞生于 UC Berkeley 的一个研究项目，现已成为 Apache 基金会下的顶级项目。Mesos 能够在同一个集群上运行多种分布式系统，提供失败侦测，任务发布，任务跟踪，任务监控，低层次资源管理和细粒度的资源共享，可以扩展伸缩到数千个节点。原生支持 CPU、内存、磁盘网络和 GPU 的资源隔离，也支持编写自定义模块实现扩展的资源隔离。

Mesos 提供了分布式的、容错的架构，并且可以对资源进行细粒度的分配、调度。这个架构主要由三个模块组成：Masters，Slaves（新版本为 Agent），和框架（Frameworks）。Master 是管理节点，一个或多个 Masters 负责管理集群中运行在每台机器上的 Mesos Slaves。Slaves 是集群中负责执行某个框架的任务的机器，Mesos Master 依靠 Apache ZooKeeper，一种用于在集群内协同领导者选举的分布式数据库，来进行领导节点的选举，实现 Master 的高可用性，体系结构如下图 2-2[55]所示。

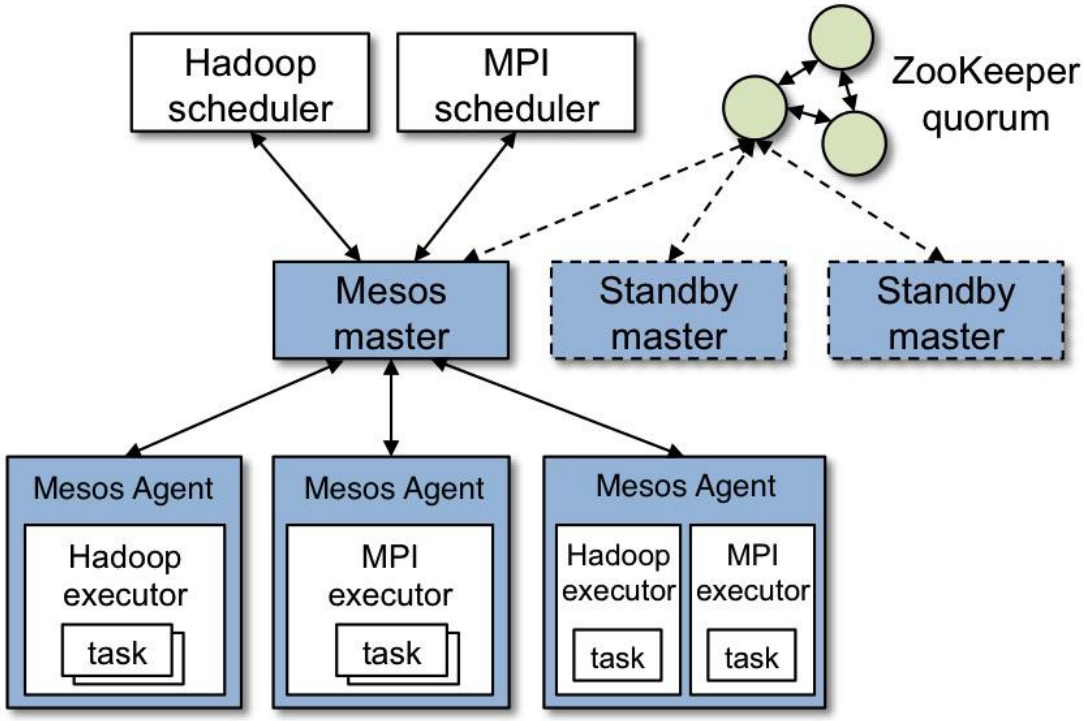

图 2-2 Mesos 体系架构

框架是运行在 Mesos 上的应用程序，每一种框架都包括调度器（Scheduler）和执行器（Executor），调度器负责决定接受还是拒绝资源提供（offer），当某个框架接受了一个来自于 Master 的资源提供后，就在该资源的 Slave 上启动一个或多个 Executors。这个 Executors 是负责执行框架任务的资源消费者。

支持 Mesos 调度的分布式计算框架包括了 Spark，Storm，Flink 等，在大的商业公司（如苹果 Siri、Twitter、Airbnb）Mesos 也被广泛使用，同时结合 Marathon 和 Docker，Mesos 也成为容器的分布式调度的资源管理工具。

在天文项目的集群调度中，智利虚拟天文台数据处理和 Web 服务发布的框架支持在 YARN 和 Mesos 资源调度[56]，Fillatre 和 Lepiller 用 YARN 对 NoSQL 的数据库和计算框架的集群资源调度[57]。

2.5 容器化应用

在软件开发过程中，将应用程序以及依赖打包为一个标准、独立、轻量的环境，这个环境其实就是容器，同时容器可运行在进程级别隔离并使用宿主机的内核，容器有效地将由单个操作系统管理的资源划分到孤立的组中，以更好地在孤立的组之间平衡有冲突的资源使用需求[58]，面对 Docker 在集群应用的需求，围绕 Docker，已经形成了很多分布式容器管理相关的生态圈。

2.5.1 Docker

Docker[59]自开源后受到广泛的关注和讨论，是目前使用最多的容器化软件。Docker 通过 Namespace 实现了资源隔离，通过 Cgroups 实现了资源限制，通过写时复制实现了高效的文件操作，与其它已经存在的容器化系统相比，Docker 可以更加简单的创建和管理容器，并与其它开源软件集成。与传统的虚拟化相比，基于 LXC 的轻量虚拟化 Docker 可以做到启动快且占用资源少。Docker 的出现使得一直以来长期存在的但并未被重视的容器技术得到快速的发展和广泛应用。它基于 Go 语言开发并遵从 Apache 2.0 协议，源代码托管在 Github 上。Docker 可在容器内快速自动化地部署应用，并通过操作系统的内核技术为容器提供资源隔离与安全保障。

相比系统虚拟化技术，容器技术的优势在于节省宿主机的资源。使用虚拟机的好处在于可以上下扩展，可控的计算资源，安全隔离，并可以通过 API 进行部署；尽管虚拟机在效率上一直在提升，现在使用虚拟机造成的 CPU 消耗只有很少的几个百分点，但毕竟每一台虚拟机还是会消耗了一部分资源用于运转一个完整的操作系统，同时还有较长的虚拟机操作系统启动的等待时间（分钟级），而容器的启动时间通常是在秒级[60]。

2.5.2 容器调度

单机的 Docker 引擎和单一的容器镜像只能解决单一服务的打包和测试问题。而要运行生产级的企业级应用，就需要容器调度管理系统。Docker 的迅速发展，生态圈也出现了多种容器的集群调度和管理工具。

Swarm[61] 是 Docker 官方管理 Docker 集群的工具，提供了集群的 Docker 节点的发现，注册机制，并根据客户端的请求，在 Docker 节点集群上根据策略进行调度。Swarm 最大的特点是提供了标准的 Docker API 接口，也就是说各种形式的 Docker Client 均可以直接与 Swarm 通信。在 Docker1.12 以上的版本，Swarm 已经集成在 Docker 引擎内，目前缺少商业应用的验证，但作为官方的容器化管理工具，未来的发展值得关注。

Kubernates[62]是 Google 的一款管理 Linux 容器集群的开源项目，支持 Docker 容器，可将集群作为一个单一的系统来对待，能够跨多主机来管理和运行 Docker 容器，并提供容器的定位、服务发现以及复制控制。作为 Google 多年大规模容器管理技术的开源版本，Kubernetes 已成为 Docker 生态圈中的重要一员[63]。

Mesos 开发之初就采用了 Linux 的容器技术，如 Cgroups，到目前为止依然是默认的。从 0.20.0 版本开始，Mesos 开始支持将 Docker 作为运行任务的隔离机制。分布式环境中，容器调度首先需要解决的就是宿主机的选择。Mesos 可以帮助我们解决复杂的容器调度，根据资源需求选择合适的宿主机，我们只需要专注于实现自己的业务逻辑。结合 Marathon，能够提供强大的在集群环境中部署容器应用和服务的平台。

2.5.3 CaaS

随着 Docker 技术的发展和广泛流行，云原生应用和容器调度管理系统也成为 IT 领域大热的词。按照云计算的服务模型划分，云计算可分为基础设施即服务（IaaS）、平台即服务（PaaS）和软件即服务（SaaS）[64]，在传统的云计算的三层服务模型基础上，衍生了基于容器调度的轻量级的云平台，即容器即服务 CaaS（Container as a Service）[65]模型。如微软的 Azure Container Service，提供在 Azure 云上部署 Docker 容器化应用，支持使用 Marathon、DC/OS 和 Docker Swarm 管理 Docker 的容器集群。Amazon EC2 Container Service (ECS)是一个高度可扩展的高性能软件容器管理服务，它支持 Docker，使用户可以轻松地在 Amazon EC2 实例集群上运行应用程序。阿里百川的 TAE2.0（Taobao App Engine）是针对移动互联网场景的定制化 PaaS 云服务，基于阿里云基础设施，采用 Docker 容器技术封装应用运行环境。Google 也在其 PaaS 产品中广泛应用 Docker。开源的 PaaS 框架 OpenStack 也已经支持 Docker。综上，作为新兴的轻量级容器引擎，Docker 为 PaaS 的构建提供了可靠的支持，并且有了非常常熟的商业应用。

Docker 作为新兴的技术，已在互联网广泛应用，对于 Docker 和及 Docker 容器云在天文领域的应用，目前国内、外相关可借鉴的研究资料较少。未来随着在天文项目组的应用逐渐增多， Docker 凭借其高效的资源利用，快速的交付和部署，更轻松的迁移和扩展应用等优势，在天文数据处理中发挥更大的作用。

2.6 望远镜观测控制系统

随着计算机技术和网络技术的快速发展，天文望远镜的整体控制思想有了较大的发展，从传统的电控逐渐发展到分布望远镜信息系统，模块化设计以及分布调用与集成的概念得到广泛共识，观测控制系统（Observation Control System - OCS）、望远镜控制系统（Telescope Control System- TCS）、焦面设备控制系统（Instrument Control System - ICS）、数据处理系统（Data Handling System - DHS）等已经成为系统设计时的基本概念[66]。

ALMA 控制系统(ALMA Control System – ACS)是当前天文界一个观测控制系统（OCS）的经典代表，事实上后续很多望远镜观测控制系统（OCS）的研究，都参考了 ACS 的基本框架与概念。ACS 是一个 CORBA 为底层的通用软件系统，支持组建和容器模型的开发。使用 CORBA 技术的好处是可以集成不同的开发技术，如 JAVA 用于控制系统的高层开发，C++用于底层时间关键开发，Python 则用于天文学家查询和分析数据[67-70]。正在建设中的 Daniel K. Inouye Solar Telescope - DKIST (原 Advanced Technology Solar Telescope，ATST)望远镜以 ACS 为基础框架，完成从观测调度到设备配置，从数据收集、数据处理、数据归档到数据查询的整个管理，各个子系统使用通信中间件进行通信[71, 72]。

同样，欧洲太阳望远镜（EST）望远镜的高级控制和软件部分包括了 OCS、TCS 和全局互锁系统 GIS。各个子系统之间通过通用的服务架构进行通信，通用服务架构提供容器或者组件的架构，提供通信连接、事件、通知、日志、配置等服务[66, 73]。国内的 LAMOST 的软件系统也是基于分布体系，使用 CORBA 技术，通过子系统之间、运行模块之间的软硬件接口来构成的多层次的、集中与分散相结合的系统[74-77]。

此外，近几年开展快速发展的自主观测技术，也给大型望远镜的信息系统建设提供了良好的借鉴。国外在观测调度方面发展的比较早也比较好的较大口径望远镜是 LCOGT(Lus Cumbres Observatory Global Telescope Network)[78]计划中的 Faulks South 和 Faulks North 望远镜和位于西班牙 La Palma 岛的 Liverpool 望远镜。该系列望远镜中将整个系统分为望远镜控制系统(Telescope Control System，TCS)、终端控制系统(Instrument Control System，ICS)和自主控制系统(Robotic Control System，RCS)三个部分，由 RCS 来实施观测调度[79]。它通过一个调度程序从预先输入的观测目标数据库中提取观测目标进行观测，通过实时的大气状况(视宁度、大气消光等)来决定是观测科学目标还是观测标准星，并且在有机会源产生的时候能够及时响应并实施观测[80]，目前 LCO 正在对 RCS 进行升级使其可以满足今后的 LCOGT 的不同地点望远镜观测调度的需求。日本 Sabaru 8.2 米望远镜成功地实施了 OCS 的研究，并取得了非常好的成果[81]。此外，在小望远镜中 OCS 的概念更为普及。主要观测目标是伽马暴的 BOOTES(Burst Optical Observer and Transient Exploring System)望远镜网络运用 RTS2(Remote Telescope System，Second Version)系统实现了对整个自主天文台的管理[82, 83]；RTS2 是一个开放源码的天文望远镜自主观测软件系统，采用面向对象的 C++ 开发，具有良好的模块化设计。实现了对观测目标的一些基本调度，它通过队列的模式对观测目标进行调度，实现了先入先出、高优先级目标优先等基本的调度[76, 84]。

针对望远镜观测控制系统中的组件部分，也开始出现大量的分布控制方法。欧南台的 Very Large Telescope（VLT），有近 40 个 CCD 相机用来自动导星，视场检查和波前检测。同时有 10 多个科学 CCD 相机在对应的光学波段上工作[85, 86]。CCD 的集群控制系统是基于 VLT 通用软件包开发的，是通过事件驱动的分布式实时控制系统，采用 C++语言实现，遵循标准架构，位于不同机器上的进程通过消息系统和数据库进行通信；运行在工作站的协调进程处理不同种类的消息。

2.7 相关理论基础

2.7.1 阿姆达尔定律

科学计算中许多问题都涉及大量实验数据的处理，如果计算可以被并行执行，即多处理器在不同的数据上进行并行计算，这样增加了处理器就加快了执行速度，从而达到了加

速的目的。然而多核面对一个似乎无法回避的难题，那就是由计算机体系结构专家吉恩.阿姆达尔在 1967 年提出的阿姆达尔定律（Amdahl's Law）[87]，它代表了处理器并行计算之后效率提升的能力。其形式如下：

$$Speedup_{Amdahl} = \frac{1}{(1-f)+{}^{f}/_{m}} \qquad (2\text{-}1)$$

其中 f 为问题中可被并行处理的部分的比例，m 为并行处理机的数量，Speedup 为并行后相比串行时的提速。

Amdahl's law 成立前提解决的问题的大小是固定的，可并行化的比例是固定的，当问题的可并行部分不大时，增加处理机的数量并不能显著地加快解决问题的时间。在实际中，当我们的计算能力增加（m 更大）之后，可以也应该去解决更大的问题。当问题更大的时候，通常情况下，这个问题也有更大的可能被分为可并行化的小问题（或者说处理多个相互独立的问题），也就意味着 f 更大（更接近 1），能得到更大的加速比[88]。

2.7.2 MapReduce

MapReduce 是一种分布式的计算模型，最初由 Google 提出[89]，用于解决海量数据的分布式计算问题。简单的说就是将进行大批量数据计算的工作分解执行（Map 过程），然后再将结果合并成最终结果（Reduce 过程）。这样做的目的是任务被分解后，可以通过大量机器进行分布式计算，加快任务执行。在 2.3 节分布式计算介绍的 Hadoop，Spark 等开源分布式框架都是 MapReduce 的开源实现。

MapReduce 借用函数式编程的思想，通过把海量数据集的常见操作抽象为 Map（映射过程）和 Reduce（聚集过程）两种集合操作，而不用过多考虑分布式相关的操作[90]。MapReduce 开发只需要通过定义 Map 和 Reduce 两个接口，读取大规模数据后，借助于计算机集群，运行用户编写的程序，将输入的大规模数据集合拆分成大量的数据片段，生成一系列的 Key/Value（键值对）作为中间数据，将每一个数据片段分配给一个 Map 任务，而 MapReduce 框架将这些子任务分配到大规模计算集群中的节点，根据分配到的 Key/Value 进行计算，执行用户指定的代码，生成中间结果 Key/Value 集合，最后输出到 Reduce 阶段，执行用户指定代码聚合中间结果并输出最终结果，其中含有相同 Key（键值）的数据会在同一个节点运行产生一个新的 Key/Value 集合[91]。

为了理解 MapReduce 编程模型，以经典的单词计数为例来说明。假设需要从给定的输入文件中获得每一个单词出现的次数。将其转化成为 MapReduce 作业，单词计数作业通过以下几个步骤定义[92]。

1，将输入的文件按行拆分成多个记录；

2，Map 函数处理上述记录，并对每个单词生成键值对（单词，出现频次）；

3，根据 Key（单词）合并 Map 函数输出的所有键值对，并根据键分组、排序。

4，将中间结果发送给 Reduce 函数，由 Reduce 函数产生最终输出。

这个 MapReduce 应用的完整步骤如下图 2-3 所示。

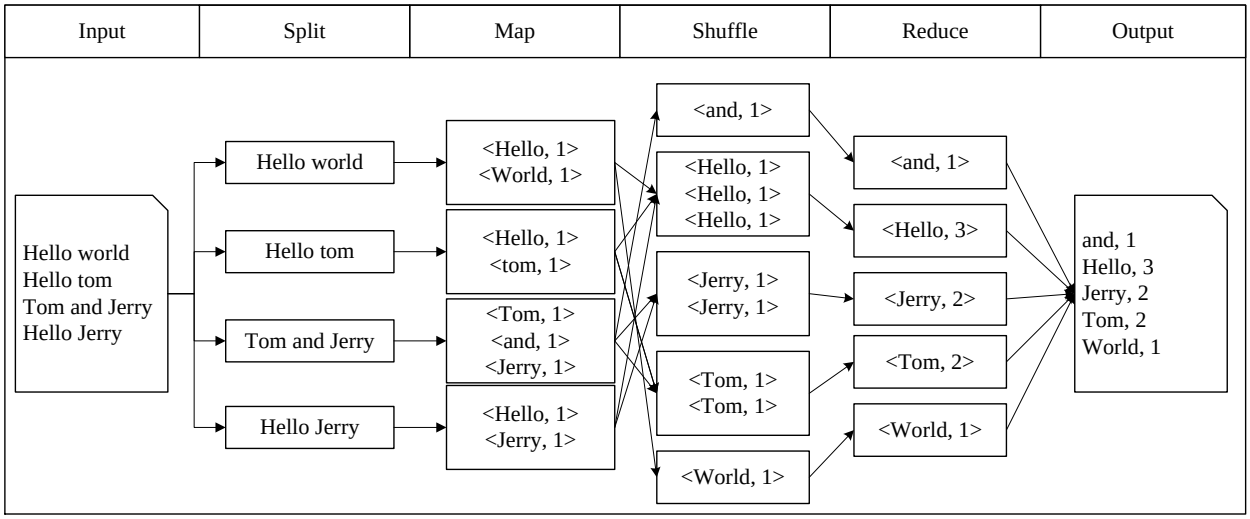

图 2-3 单词计数的 MapReduce 执行过程

2.7.3 ZeroMQ

ZeroMQ 是一个开源的、跨平台、高性能、精简灵活的网络消息中间件，将操作系统的异步、消息缓冲区和多线程处理机制封装在内，跨越多种传输协议（如进程内通信、IPC、TCP 和 UDP）、网络连接建立、数据打包成帧、路由选择等底层网络通信行为进行了抽象。它可以用于多种语言编写程序，如 C，C ++，Java，.NET，Python 和支持多种主流操作系统平台，如 Linux，Windows，也被称为 ØMQ、0MQ 或 ZMQ。

ZeroMQ 并不是如其名所示的是一个消息队列，它更像是一个面向消息的中间件或者网络通信程序库，定义了多种基础的通信模型，如请求回应模型（Request-Reply）、发布订阅模型（Publish-Subscribe）、推拉模型（Push-Pull），Router-Dealer，并可灵活组合使用这些基础模型构建更复杂的分布式计算网络通信框架。

相对于同类中间件 MSMQ、ActiveMQ 和 RabbitMQ 在部署时需要专门的一个服务器，ZeroMQ 只需要让应用程序引用 ZeroMQ 程序库，就可以在多个进程间进行消息发送，使得部署起来非常简单。ZeroMQ 采用 C/C++ 开发，并且协议格式定义得很简洁，所以性能远远高于其他的消息中间件[93]。缺点是 ZeroMQ 被设计成侧重于消息传输的轻量级消息中间件，缺少消息服务器来存储和转发消息，所以不支持消息持久化及崩溃恢复。

在早期版本的 ZeroMQ 中，API 基于 AMQP 的交换和队列模型，作者于 2009 重写了 API，改用 BSD Socket API，降低了 API 的学习曲线，方便 ZMQ 与现有技术的集成。例如，将 ZMQ 对象暴露为“套接字”或“文件描述符”允许在同一事件循环中处理 TCP，UDP，管道，文件和 ZMQ 事件。

2.7.4 有向无环图

有向无环图（Directed Acycline Graph），简称 DAG 图[94]，DAG 图是一类较有向树更一般的特殊有向图，从某个顶点出发经过若干条边不能再回到该顶点。分布式计算领域中 DAG 计算模型是指将计算任务在内部分解为若干个子任务，这些子任务之间由逻辑关系或运行先后顺序等因素被构建成有向无环图[95]。DAG 计算模型一般分为三层结构[96]：

1) 应用表达层，即是通过一定手段将计算任务分解成由若干子任务形成的 DAG 结构，其核心是表达的便捷性，主要是方便应用开发者快速描述或构建应用。在 Spark 中，就是用户定义的 RDD 的 map 和 reduce 等转换操作。

2) DAG 执行引擎层，主要目的是将上层以特殊方式表达的 DAG 计算任务通过转换和映射，将其部署到下层的物理机集群中运行，这层是 DAG 计算的核心部件，计算任务的调度，底层硬件的容错，数据与管理信息的传递，整个系统的管理与正常运转等都需要由这层来完成。在 Spark 中，就是 DAGScheduler，将 RDD 的依赖关系转换为 Stage 的 DAG。

3) 计算层，由大量物理机器搭建的分布式计算环境，是计算任务最终执行的场所。

DAG 在分布式计算中是非常常见的一种结构，Dryad，Apache Tez、Storm、Spark 都是实现了 DAG 计算模型的计算框架。

2.7.5 高可用性

现代的分布式计算目标都是运行在普通的 PC 服务器上，因此，在一个分布式系统中出现局部和临时错误是大概率事件[97]。错误可能来自于物理系统故障，外部系统故障也可能来自于系统自身的代码错误，依靠自己实现的代码不会出错来保证系统稳定其实也是难以实现的，因此要设计对任何可能错误的错误检测和容错处理，发生错误后，不影响系统的持续运行。

分布式环境下，失效检测是高可用性的基础，心跳模型被广泛应用于小规模集群中，相对于上百个节点的大规模集群来说，小规模的集群使用心跳模型检测对系统造成的负荷较小，并且小规模集群的网络带宽大速度较快延迟较小，这样心跳模型可以作出较为精确的判断。当心跳模型应用于两个服务器的集群时，两个服务器采用活动/备份或者活动/活动集群模型工作，它们之间使用心跳模型进行相互检测[98]。

对于大规模集群来说，目前普遍使用高可用、强一致性的服务发现存储仓库，来实现高可用性、Leader 选举操作，这在当前的分布式集群中比较流行，比如 Mesos 使用 Zookeeper[99]，Docker Swarm 高可用集群同时支持利用 Consul、Etcd、ZooKeeper 进行 Leader 的选举[100]，Kubernetes 也采用了 Etcd 等实现了自身的高可用[101]。

2.7.6 资源隔离

虚拟机技术利用虚拟环境为操作系统提供虚拟的硬件平台，从而可以实现系统的隔离。良好的资源隔离使得虚拟机技术成为近年来云计算的支撑技术[102]。虚拟化技术在许多领域有了非常广泛的用途。虚拟化技术能提高服务器的利用率，显著降低信息系统的运维成本；虚拟化技术也为网格计算、可信计算提供了新的技术手段。基于虚拟化技术可以实现操作系统和应用软件的动态迁移，从而能够实现服务器的在线维护和升级，也为构建高可靠系统提供了革新的技术手段。

运行在同一物理服务器上的多个虚拟机之间相互隔离，虚拟机与虚拟机之间互不影响。包括：计算隔离、数据隔离、存储隔离、网络隔离、访问隔离，虚拟机之间不会泄露数据，应用程序只能通过配置的网络连接进行通信[103]。

而目前容器作为资源隔离的另一种技术，越来越受到关注。容器有效地将由单个操作系统管理的资源划分到孤立的组中，以更好地在孤立的组之间平衡有冲突的资源使用需求[104]。容器与资源隔离并不是新的概念，如 Solaris Zones 和 BSD Jails 就是非 Linux 操作系统上的容器，在 Linux 平台下 的容器技术，有 Linux-Vserver、OpenVZ 和 FreeVPS。虽然这些技术都已经成熟，但是这些解决方案还没有将它们的容器支持集成到主流 Linux 内核中[105]。

在 Linux 内核提供的容器技术中，最常用的是 LXC，它是一种共享 Kernel 的操作系统级别的虚拟化解决方案，通过在执行时不重复加载内核，且虚拟容器(Container)与宿主机(Host)之间通过共享内核来加快启动速度和减少内存消耗。LXC 在 2008 年，被添加到了系统内核中，它综合了内核 Cgroup 和命名空间来实现轻量级的进程隔离[58]。LXC 的目的就是使用宿主机的内核，在宿主机上创建一个独立的和标准 Linux 类似的环境。Cgroup 是内核里用来把用户进程分组的机制，在此基础上每个子系统（CPU、Memory、Disk 和 Networking）有相应的机制来监控和限制资源利用。提供每个任务运行环境的隔离，从而保证任务性能的稳定性。

2.8 本章小结

本章总体上对分布式计算相关的技术进行了比较全面系统的阐述，其中包括对当前高性能计算、并行计算、分布式计算的相关计算的介绍，以及在天文中的具体应用，也涉及了资源调度的一些技术，并对前沿的容器化应用的进行了概要的介绍，章节最终选用分布式计算作为本文的研究目标。

调研表明，国外的望远镜非常重视观测控制系统的研究。特别是 DKIST(ATST)的设计中提出，OCS 是整个望远镜的灵魂，是整个望远镜信息系统的最高层，负责整个望远镜观

测过程的整体调度和事件触发，如果把 TCS、ICS 等看成是望远镜的不同的驱动的话，OCS 就是望远镜真正的操作系统。当前信息技术领域的新技术对天文技术的发展起到了重要的支撑作用，但这些技术如何应用于天文技术，真正解决天文科学问题还有很多的底层工作需要开展，需要从核心算法和可靠性等多方面进行拓展与改进，满足天文学的要求。

接下来，在第 3 章中，将选择基于内存迭代计算的分布式计算框架 Spark，研究在 MUSER 实时计算应用中的关键技术，在第 4 章中，将对在分布式计算技术中最核心的分布式计算框架的设计进行研究和分析，研究适合天文海量观测数据计算的分布式计算框架。

第3章 Spark 在 MUSER 中的应用研究

Spark 是一个用来实现快速而通用的开源的分布式计算平台，Spark 适用于各种各样原先需要多种不同的分布式平台的场景，包括批处理、迭代算法、交互式查询、流处理[106]。通过在一个统一的框架下支持这些不同的计算，Spark 使我们可以简单而低耗地把各种处理流程整合在一起。而这样的组合，在实际的数据分析过程中是很有意义的。不仅如此，Spark 的这种特性还大大减轻了原先需要对各种平台分别管理的负担。

Spark 提供简单易用的 API，编程语言支持用户使用 Python、Java、Scala 和 R 语言编写的分布式处理程序。特别是 Python 语言的支持，在天文数据处理领域尤为重要。Spark 还能和其他大数据工具密切配合使用。例如，Spark 可以运行在 YARN 和 Mesos 集群上，访问包括 HDFS、Kafka、Cassandra 在内的数据源[107]。

在 MUSER 的日常数据处理中，包含了两方面的需求，一是对历史数据的处理，另外一个是 5 秒钟的抽样观测数据的实时处理。本章针对 MUSER 的实时数据处理要求，研究使用 Spark Streaming 提供的实时计算框架来满足 MUSER 的实时数据处理需求。章节首先介绍了 MUSER 中使用 Spark Streaming 的数据处理 Pipeline，实现 MUSER 的自定义数据接收器，讨论了使用 Spark 中的核心组件 RDD（Resilient Distributed Dataset）弹性分布式数据集的影响处理性能的分区方式，持久化选择。最后结合 MUSER 中实时数据处理的需要，搭建了模拟环境对 Spark Streaming 从性能和可扩展方面进行测试分析和研究。

3.1 实时处理 Pipeline

为了能够实时监测当前的设备状态和观测情况，MUSER 每 5 秒钟产生的抽样观测数据，以 Socket 方式向外写出，也就是说需要在 5 秒内完成观测抽样数据的处理，并将处理结果发送到监控终端显示，否则就会产生数据积压，监控终端也就不能以准实时的方式反应望远镜观测情况。MUSER 的实时数据处理的主要过程包括，实时数据接收，预处理，天气数据标识，仪器状态校验和数据标定等[13]，同时根据需求生成 UVFITS 文件生成或者进一步生成脏图，再洁化，成图。在每个处理过程中还需要与其他系统交互。如与天气信息的查询，仪器状态查询，星表位置计算等。如下图 3-1 显示了 MUSER 实时数据处理的流程图。

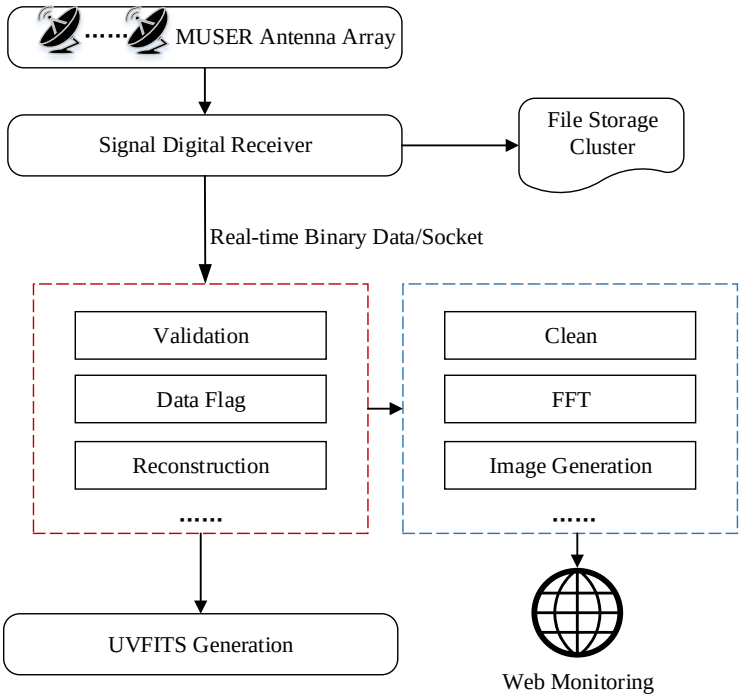

图 3-1 MUSER 实时数据处理流程图

由数字接收机产生的实时数据以 Socket/TCP 的方式向外写出数据，数据经过图 3-1 中左侧红色虚线框内的数据校验，重建等处理过程后生成 UVFITS 文件，再经过图 3-1 右侧蓝色虚线框内的处理过程后生成实时发布到 Web 监控端。其中蓝色框内的处理过程需要在安装了 GPU 卡的机器上执行。

3.2 自定义 Receiver

目前在 MUSER 低频部分的日常观测中，每 3 毫米产生 1 帧（100000 Bytes）的观测数据，每 5 秒产生 8 帧（8*100000 Bytes）的观测抽样数据[108]。观测抽样数据需要实时处理，并将处理结果发送到监控终端显示。Spark Streaming 提供的 Python 接口，都是加载文本对象的文件，或是连接 Flume、Kafka、HDFS 类型的特定存储系统，Spark Streaming 提供的例子也都是关于处理文本型的数据对象，而 MUSER 的实时数据是二进制的，且具有特定数据格式（每 100000 Bytes 大小为一帧），因此需要自定义 Spark Streaming 的接收器(Receiver)，这个接收器能够接收到二进制数据，并能分析数据，并转换成 RDD。

Spark Streaming 支持从内置数据源包括 Flume、Kafka、Kinesis、HDFS、文件、套接字等加载数据，也可以自定义接收器来从任意的流中接收数据。Spark 提供了 org.apache.spark.streaming.receiver.Receiver 这个抽象类，只需要继承这个抽象类，实现相关的方法，就可以实现自定义接收器，编程语言可以使用 Scala 或者 Java。

在 Receiver 抽象类中，有两个个关键的方法需要重写：

1) onStart()：这个函数主要负责在接收器启动时，做数据的接收工作；

2) onStop()：这个函数主要负责在接收器停止时，做清理的工作，停止接收线程。

不管是 onStart() 和 onStop() 方法都不可以无限期地阻塞。通常情况下， onStart()方法会启动一个新的线程负责接收数据，而 onStop()会保证接收数据的线程被终止。接收数据的线程也可以使用 Receiver 类提供的 isStopped() 方法来检测是否可以停止接收数据。数据一旦被接收，这些数据可以通过调用 store(data) 方法存储在 Spark 中， store(data) 方法是由 Receiver 类提供的。store 方法是一个阻塞调用，只有当所有的数据都被存储到 Spark 里面才会返回。

自定义的接收器可以通过使用 streamingContext.receiverStream() 方法来在 Spark Streaming 应用程序中使用，但在 Python 的 API 中，没有提供实现自定义接收器的方法，不能直接创建自定义接收器。因此需要一个中间的代理来调用自定义的接收器，我们创建了 MUSERStreamHelper。MUSERStreamHelper 是一个单例的类，提供了一个静态方法，这个方法通过 Py4J 可以在 Python 环境中直接调用，如下图 3-2 所示，MUSERStreamHelper 和自定义的接收器 MUSERStreamReceiver 都是使用 Java 编写，运行在 JVM 中。

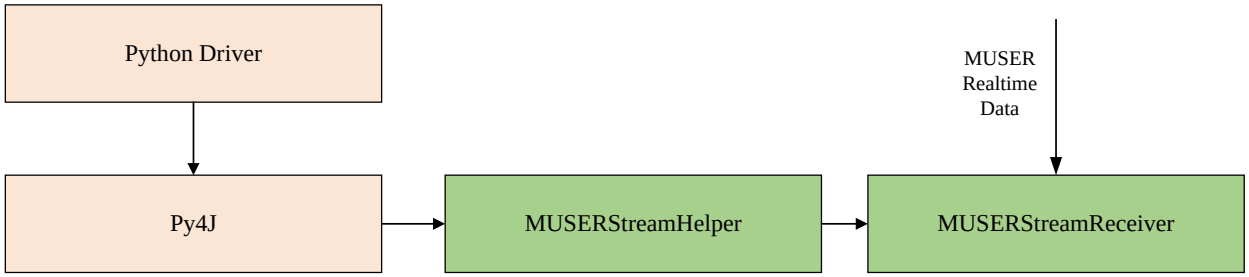

图 3-2 MUSER 自定义 Receiver 实时数据处理流程图

3.3 弹性分布式数据集

Spark 的核心是 RDD，包括 MapReduce，Streaming，SQL，Machine Learning 的不同的组件都是建立在 RDD 的基础上。RDD 是一个只读的，可分区的分布式数据集，这个数据集的全部或部分可以缓存在内存中，在多次计算间重用。RDD 具有良好的扩展性，可以方便的扩展 Spark 数据源[109]。目前数据源方面已经支持文件系统，HDFS，JDBC 等，数据格式支持 Avro、CSV、JSON、Parquet 等[110]。

用户可以显式地将数据存储到磁盘和内存中，并能控制数据的分区。同时，RDD 还提供了一组丰富的操作来操作这些数据。例如 map、flatMap、filter 等转换操作，以及如 count、collect、save 等执行操作。

3.4 自定义分区方式

RDD 作为数据结构，本质上是一个只读的分区记录集合。一个 RDD 可以包含多个分区（Partition），每个分区就是一个数据集片段。RDD 可以相互依赖。如果 RDD 的每个分区最多只能被一个子 RDD 的一个分区使用，则称之为窄依赖；若多个子 RDD 分区都可以依赖，则称之为宽依赖。不同的操作依据其特性，可能会产生不同的依赖。例如 map 操作会产生窄依赖，而 join 操作则产生宽依赖。

加载 Scala 集合或外部数据来创建 RDD 时，是可以指定分区个数的，若指定了具体值，那么分区的个数就等于该值，比如：

val rdd = sc.textFile(filePath, 5)　// 指定分区数为 5

Spark 内部提供了 HashPartitioner 和 RangePartitioner 两种分区策略，HashPartitioner 是计算数据项的 Key 的 Hash 值，Hash 值相同的元素放入同一个分区之内；RangePartitioner 是将数据项的 Key 同一数据范围的数据放入同一分区。为了减少通信开销，尽量将需要进行相同操作的数据放在同一分区，另外为了提高集群的利用率，使 RDD 的各个分区上的数据量尽量均匀。

MUSER 中使用观测时间和频段作为键值对 RDD 的 Key，使用 HashPartitioner 作为分区方式，因每个时间都不同，Hash 值也就不同，会造成分区过于分散，在做多个数据帧的积分操作，会有较大的通信开销，使用 RangePartitioner 也不能完全满足要求。因此需要实现自定义的分区方式，可以使用 Spark 提供的 org.apache.spark.Partitioner 抽象类，继承 Partitioner 类并重写下面三个方法：

- numPartitions: Int：返回创建出来的分区数。
- getPartition(key: Any): Int：返回给定键的分区编号（0 到 numPartitions-1）。
- equals()：判断相等性的标准方法。这个方法的实现非常重要，Spark 需要用这个方法来检查你的分区器对象是否和其他分区器实例相同，这样 Spark 才可以判断两个 RDD 的分区方式是否相同。

在 Python 中，因为没有 Partitioner 抽象类，实现自定义的分区方式，不需要继承 Partitioner 类，只需要在 rdd.partitionBy()里使用一个 Hash 函数，MUSER 中使用观测时间和频段作为键值，格式为：时间_频段，如：

20151101120854.354161_400

如下表格 3-1 的伪代码表示将一天一个小时中的所有频段相同的放在一个分区中。

表格 3-1 MUSER 的自定义分区的伪代码

```
def customize_partition(key):
    keys = key.split("_")
```

```
    hashHour = hash(keys[0][6:8])  #计算小时的 Hash 值
    hashBand = hash(keys[1])  #计算频段的 Hash 值
    return hashHour + hashBand

rdd.partitionBy(4, customize_partition)
```

3.5 持久化选择

使用 cache()和 persist()方法可以将 RDD 的中间计算结果缓存起来，cache()方法使用了默认的存储级别（MEMORY_ONLY），persist()方法可以传递一个 StorageLevel 对象来设置存储级别。使用 cache()方法，如果缓存新的 RDD 分区时，空间不够，旧的分区就会直接被删除。当用到这些分区数据时，由于计算结果已经被删除，需要进行重算。而 persist()方法允许传递一个存储级别，可以结合内存和磁盘存储获得更好的效果。如 StorageLevel.MEMORY_AND_DISK 是将 RDD 作为非序列化的 Java 对象存储在虚拟机（JVM）内存中，如果内存中放不下，则会将旧的分区写入磁盘，当再次需要用到的时候再从磁盘上读取回来。这样的代价有可能要比重新计算各分区要低很多，也可以带来更稳定的性能表现。当 RDD 分区的重算代价很大（比如计算一个很大的数据集计算）时，这种设置尤其有用。

3.6 时间切片

Spark Streaming 流式数据处理的本质是将连续的数据持久化，离散化，然后进行批量处理。Spark Streaming 将输入的实时数据流以时间片为单位进行拆分，目前是以秒级为单位拆分，然后以类似批处理的方式处理每个时间片的数据[36]。互联网应用的实时流式数据由于数据流动态持续的特性，其数据项到达的次序与速度无法控制。并且随着时间的延续，数据流的体积在理论上会是无限的。MUSER 的实时数据是观测设备按观测时间顺序产生的，其次序和速度（8*100000 Bytes/5s）是确定的。使用 Spark Streaming 的编写程序，首先需要声明 StreamingContext 对象，StreamingContext 对象的构造函数中有一个 batchDuration 参数，通过该参数定义 Spark Streaming 对数据流的切分间隔，batchDuration 参数会显著的影响数据处理速率，这个参数值可以通过检查端到端的延迟来判断（可以在 Spark 驱动程序的日志中查看"Total delay"或者利用 StreamingListener 接口）。如果延迟维持稳定，那么系统是稳定的。如果延迟持续增长，那么系统无法跟上数据处理速率，是不稳定的。如果系统是不稳定的，除了可以适当的减小 batchDuration 的值，同时也要考虑集群的处理能力，结合 Spark UI 任务的执行时间，找出延迟持续增长的原因。这里我们设置该值为 5，即 5 秒。

实时数据流到达后，系统按照时间切片间隔将数据流切片后，返回 DStream，其本身

封装了按时间片离散化了的数据流。DStream 中包含一个类型为 HashMap 成员变量 generatedRDDS，其中 Key 是时间片段，Value 就是 RDD，DStream 操作和 RDD 的操作类似。

3.7 共享变量

共享变量是一种可以在 Spark 任务中使用的特殊类型的变量，Spark 中有两种类型的共享变量，广播变量和累加器。广播变量用来高效分发较大的对象，累加器用来对信息进行聚合。

MUSER 的数据处理中，需要对高精度的观测目标的视位置进行相关计算，比如在每个相位差校正和生成 UVW 数据阶段。观测时为了确保相位补偿精度，MUSER 需要观测目标的视位置计算精度能够优于 1 个毫角秒，为此，采用了精密的 JPL 星历表[111]。使用 Spark 广播变量，将星历表高效地发送到所有的工作节点，提高星表查询和计算的速度。当广播一个比较大的值时，选择既快又好的序列化格式是很重要的，因为如果序列化对象的时间很长或者传送花费的时间太久，这段时间很容易就成为性能瓶颈。

JPL 星历表是以二进制文件存储，频繁的文件打开和关闭是一个耗时的工作，如果能在多个数据元素间共享一次文件配置就会比较有效率。Spark 中使用基于分区对数据进行操作以避免为每个数据元素进行重复的配置工作，基于分区的 map（mapPartitions 函数）和 foreach（foreachPartitions 函数），只对 RDD 的每个分区运行一次，这样可以帮助降低文件打开和关闭的频次，从而提高性能。

3.8 检查点

Spark Streaming 中不断的有数据流进来，它会把数据积攒起来，积攒的依据是以 Batch Interval 的方式进行积攒的，例如 5 秒钟，但是这 5 秒钟里面，就会收到 8 帧的原始数据，这 8 个帧的数据就构成了一个集合，而 RDD 处理的时候，就是根据这个固定的集合产生 RDD。对于连续不断的流进来的数据，就会根据这个连续不断帧的数据构成 Batch，因为时间间隔是固定的，所以每个时间间隔产生的数据也是固定的。

Spark 尽管可以通过操作图回溯之前执行的操作，并重新计算失败的任务来提高容错性。不过，如果只用谱系图来恢复的话，重算有可能会花很长时间，因为需要处理从程序启动以来的所有数据。在实时计算的场景下，Spark 可以利用记录日志的方式来支持容错，也就是检查点机制（Check Point）。检查点机制是我们在 Spark Streaming 中用来保障容错性的主要机制,它可以使 Spark Streaming 阶段性地把应用数据存储到诸如 HDFS 这样的可靠存储系统中，以供恢复时使用。在恢复数据时，Spark Streaming 只需要回溯到上一个检

查点即可。

3.9 异步执行

在 Spark Streaming 应用中，提交一个应用后，驱动器开始初始化接收器，通常在 onStart() 方法中启动一个新的线程负责接收数据，一旦接收到数据，驱动器按照批处理的时间（batchDuration）定期调度任务处理这些数据。应用和节点上的执行器在刚启动时需要几分钟做初始化的工作，会导致最初接收的任务一直没有完成，在这段时间内驱动器不会调度新的任务。这时候在接收器端就会有数据积压，随着积压的数据量越来越大，新的数据会进入内存，旧的数据移出内存，进而给 Java GC（垃圾回收）带来严重的压力，容易引发应用程序崩溃。

这里参考[2]使用异步处理任务中的业务逻辑来解决。如下表格 3-2 和表格 3-3 的代码所示，在同步方案中，Task 内执行业务逻辑，处理时间不定；异步方案中，Task 把业务逻辑嵌入线程，交给线程池执行，Task 立刻结束，执行器向驱动器报告执行完毕，异步处理的时间非常短。另外，当线程池中积压的线程数量太大时，会暂时使用同步处理。经实验验证，该方案大大提高了系统的吞吐量。

表格 3-2 同步处理逻辑

```
#函数 runBusinessLogic 是 Task 中的业务逻辑，执行时间不定
rdd.mapPartition(lambda partition : runBusinessLogic(partition))
```

表格 3-3 使用线程池异步处理逻辑

```
#异步处理，threadPool 是线程池
rdd.mapPartition(lambda partition : runThread(partition))
def runThread(partition):
   if threadPool.size > 100 : #线程池中积压的线程数大于 100 时
      runBusinessLogic(partition) #暂时同步处理
   else :
      threadPool.process(runBusinessLogic(partition))
```

3.10 实验

以下所有实验均在汉柏 PowerCube 上创建的虚拟机上进行，使用 5 台配合相同的虚拟机，每个虚拟机配备了 16G DDR2 内存，两路的 Intel Xeon E5-2640 v2 CPUs, 2.0 GHz, 16 核的 CPU，16G 主存，40GB 的硬盘。操作系统使用 CentOS7，Linux 内核版本 3.10.0。 JDK

[2] http://geek.csdn.net/news/detail/78414

版本为 1.8，Spark 的版本为 2.0.2，其中一个节点作为 Master 节点，同时也是 Worker 节点，其他四个节点作为 Worker 节点。 如下图 3-3 为 Spark 集群的 Web 监控界面。

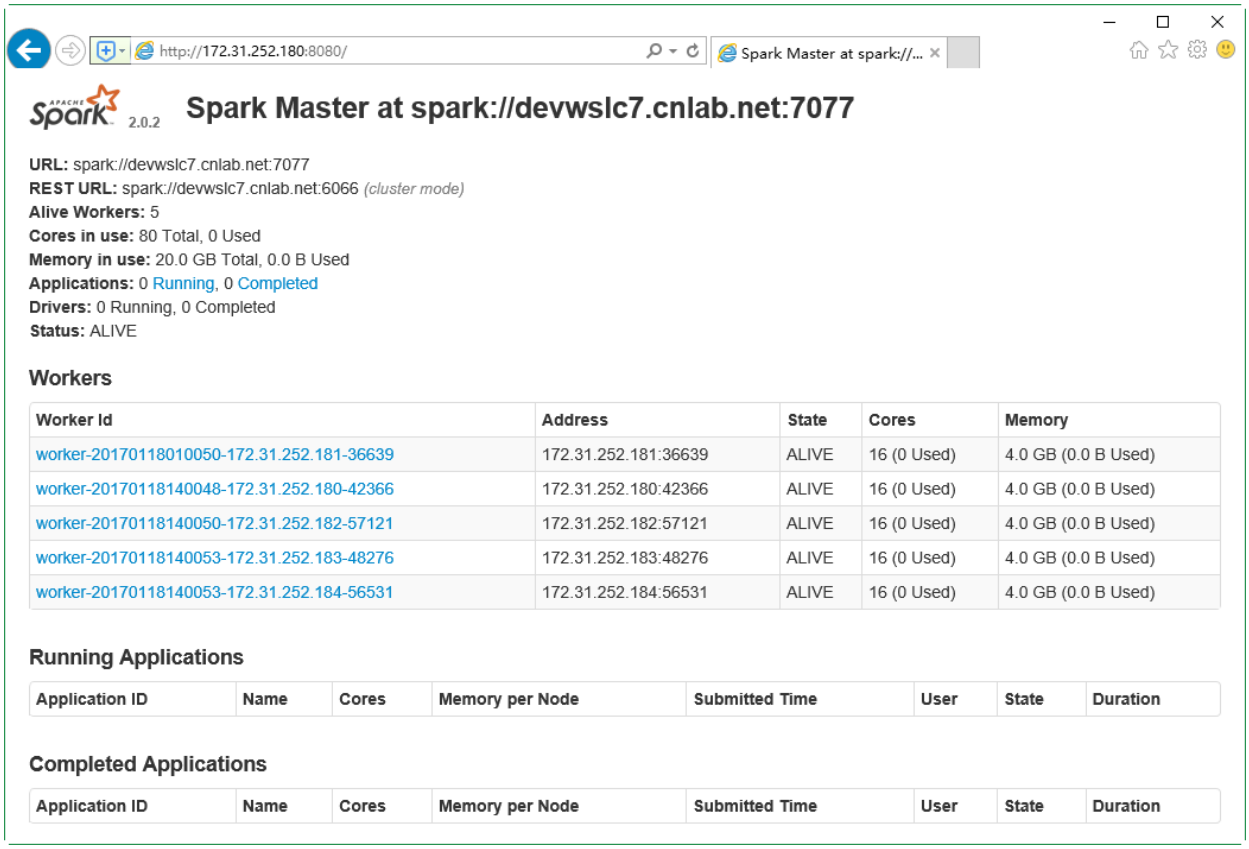

图 3-3 Spark 集群的 Web 监控界面

每个节点的 SPARK_WORKER_MEMORY 配置为 4096M，该实验主要测试 MUSER 时间计算自定义的接收器，处理时间，及整个系统的延迟，batchDuration 设置为 5 秒。测试数据使用真实的原始观测数据，每 5 秒发送 8 个小帧，选取了 10 分钟的测试结果，实际测试结果如下图 3-4，图 3-5，图 3-6 和图 3-7。

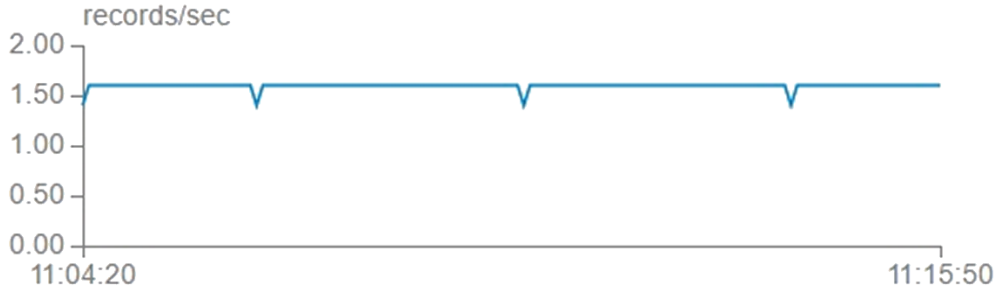

图 3-4 MUSER 自定义接收器每秒接收的记录数

图 3-5 MUSER 实时计算调度延迟

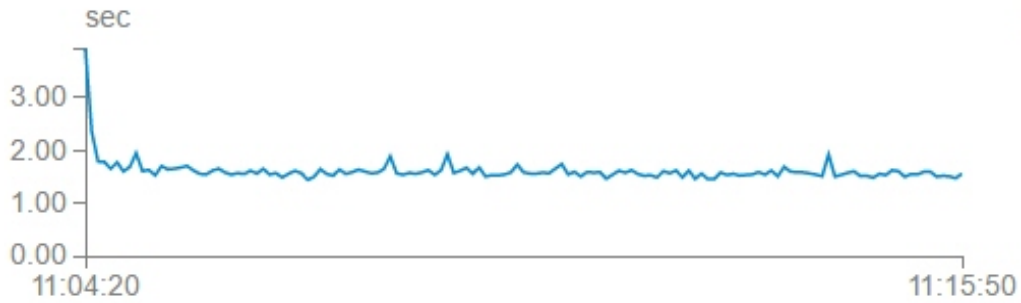

图 3-6 MUSER 实时计算处理时间

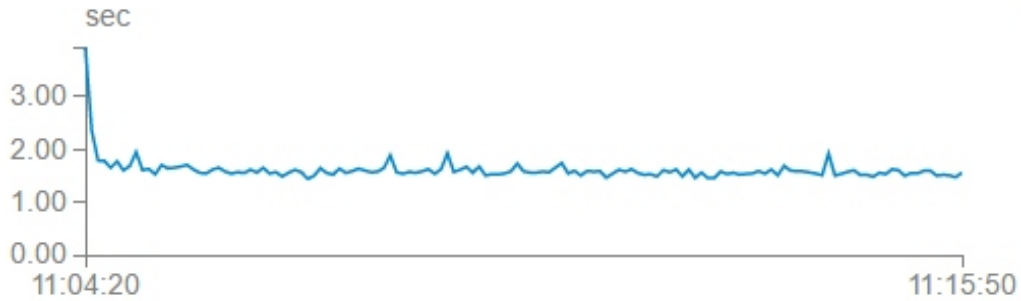

图 3-7 MUSER 实时计算总计延迟时间

从图 3-4 的数据接收的趋势图可以看出接收器每秒接收的记录数和发送的记录数一致，并且系统保持稳定。图 3-5 显示了 Spark 的调度延迟在 2 毫秒左右。从图 3-6 的数据处理时间随时间的变化图，可以看出在任务提交初期，由于驱动器和执行器的初始化延迟，在最初的任务执行时间较长，随后的任务的处理时间比较稳定，大约在 1.6 秒左右，满足 MUSER 中实时任务处理的需求。

3.11 讨论

借助于 Spark 的高性能、可扩展的体系结构，我们使用 Spark Streaming 实时计算框架来满足 MUSER 的实时数据处理需求。在 MUSER 成图的计算过程中，需要用到 CLEAN 算法[112]，CLEAN 是一个迭代的过程，目前没有还利用到 Spark 中基于内存迭代计算这个最重要的概念。Spark Streaming 已在 MUSER 小范围使用，将数据以小批量方式处理后，可以同时兼容批处理和实时处理的算法，特别适用于某些同时处理历史数据和实时数据的业务场景，也适用于对实时性要求不是很高的实时计算业务。但 Spark Streaming 作为一个

全新的实时计算框架，还需要更多的业务场景，特别是在天文数据处理中去验证其功能，稳定性也需要更长时间的考验。后续我们将会继续研究 Spark Streaming 在天文方面的应用，并强化 Spark Streaming 的作业监控机制。

3.12 本章小结

在 MUSER 的日常数据处理中，包含了两方面的需求，一是对历史数据的处理，另外一个是 5 秒钟的抽样观测数据的实时处理。本章针对 MUSER 的实时数据处理要求，研究使用 Spark Streaming 提供的实时计算框架来满足 MUSER 的实时数据处理需求。章节先 Spark 中的核心组件 RDD 弹性分布式数据集的介绍开始，讨论了影响 Spark 处理性能的分区方式，持久化选择。最后结合 MUSER 中实时数据处理的需要，设计了自定义的接收器，自定义分区，搭建了模拟环境对 Spark Streaming 从性能和可扩展方面进行测试分析和研究。但是以 Spark 为例，该系统虽然性能和可扩展性比较适合天文海量数据系统的特性要求，但在实验过程中发现，该框架的编程难度较大，系统的很多关键特性没有相应的 Python 的 API 支持，同时部署和管理较为复杂。

第4章 天文分布式计算框架研究

在上一章节研究了基于 Spark 的分布式计算在天文领域的应用，Spark 无疑能够提供高速的计算性能，具备良好的可扩展性。但是像 Hadoop、Spark 这些开源计算框架的安装、部署和管理的复杂性，导致其在实际应用时，增加了应用的复杂性，例如，需要有专门技术知识和经验的维护人员进行管理。这些框架大都基于 Java 和 Scala 开发，其能提供 API 也大都是 Java 和 Scala 接口，尽管 Spark 也提供了 Python 的 API，但通过 Py4J 的库进行转换，对性能也有较大的影响。其出身于互联网的特性，也更适合应用在互联网的应用中。

为了更好得使用分布式计算技术为天文计算服务，应对天文海量数据计算带来的挑战，本章节在分布式计算框架研究的基础上，针对天文分布式计算领域需求设计了面向天文计算的轻量级可扩展的分布式计算框架——OpenCluster。

4.1 框架设计

在开源分布式计算平台的实际部署和使用中，天文中大数据的特殊性，给分布式计算的框架选择带来了很大的困难，照搬基于互联网大数据的开源分布式数据处理平台并不能完全解决问题。同时这些平台大都基于 Java 和 Scala 语言开发，而不是在天文领域广泛使用的 Python 语言。天文计算领域需要一种新的分布式的处理框架满足科学数据处理的需求。这样的框架应该具有几个特点：

1，轻量级，框架只提供数据通信，不对数据处理流程和算法进行约束。

2，分布式，支持处理节点水平扩展，对集群节点状态的实时感知等。

3，任务粒度可定制，任务分配者与工作节点，分配者与分配者之间任务粒度可自由定制。

4，集成服务管理，将服务节点纳入集群管理。

5，处理多模式，处理环节之间，可以以同步与异步方式进行。

6，任务可追溯，对每个任务状态的完成情况可以标识。

7，跨平台，能在 Windows 和 Linux 平台下使用。

框架使用 Python 进行编写，选择 Python 有几个方面的原因：

1，跨平台，支持多种主流操作系统。

2，面向对象，使用继承方法实现不同类型的任务分配者（Manager）和执行者（Worker）。

3，丰富的第三方应用库，如 numpy，pymatplot，scipy。这对天文数据处理尤为重要。

基于这些原因，我们采用 Python 语言设计开发了 OpenCluster

4.1.1 计算模型

OpenCluster 的计算模型的设计思想来源于传统的工业生产的工厂模型，如下图 4-1 所示。

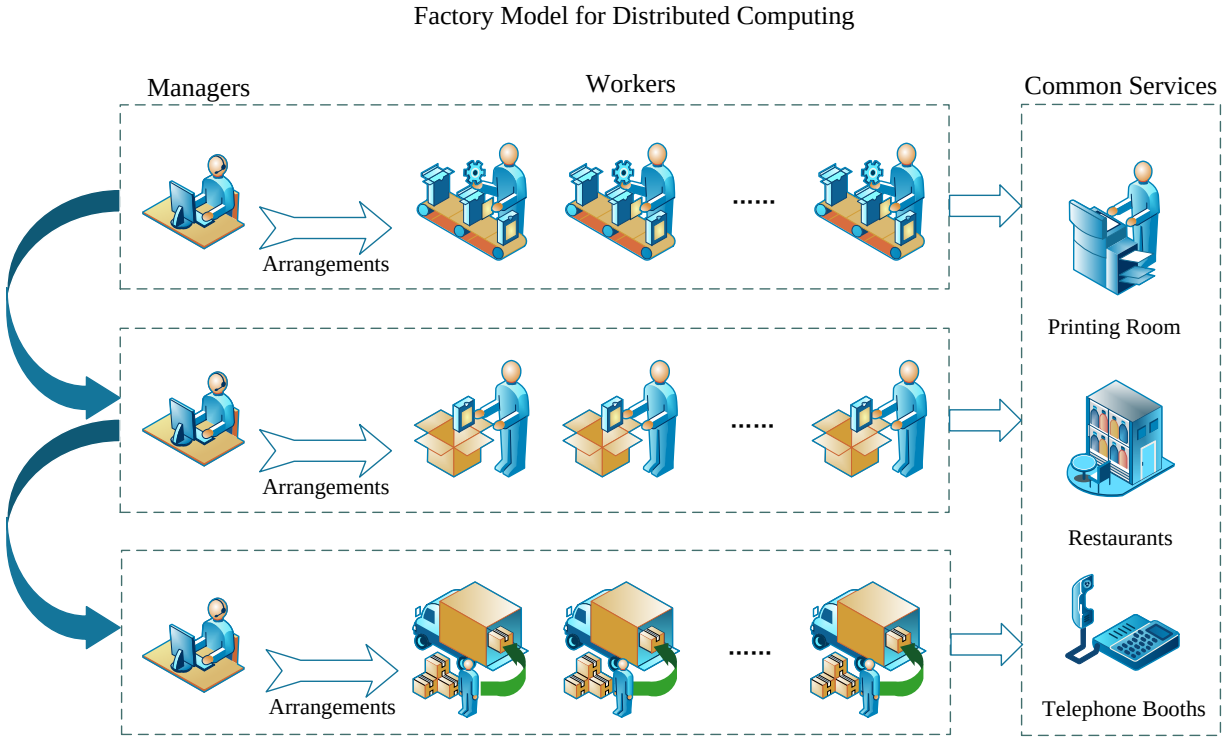

图 4-1 OpenCluster 的概念模型

在传统的工业生产中，一个产品在工厂的生产到销售要经过若干的流程，如加工，包装，发货等。工厂收到订单后，按照生产工艺将订单交由加工环节的工头，工头收到任务后，再将工作任务下发到工人。工作完成后再交由包装环节的工头，同理最后到发货环节。在整个生产过程中，工人和工头都需要一些公共服务，如打印，打电话，到餐厅吃饭等。这就构成了一个简单的工厂生产流程。

在天文数据处理中，也存在类似的流程。观测设备产生原始数据，原始数据的处理需要多个步骤，最终形成的数据才能被科学家使用。在处理过程中也需要调用公用服务来校验，修正数据。而这些处理步骤可能因为处理程序，处理平台的不同会被分散到不同的计算节点上进行。正因为这种相似性，促使我们采用工厂模型作为天文分布式计算概念模型。

4.1.2 整体框架

在工厂模型中，包含了工厂（Factory）、车间（Workshop）、车间经理（Manager）、工人（Worker）和服务（Service）这几种实体概念。他们之间的关系形成了 OpenCluster 的架构图，如下图 4-2 所示。

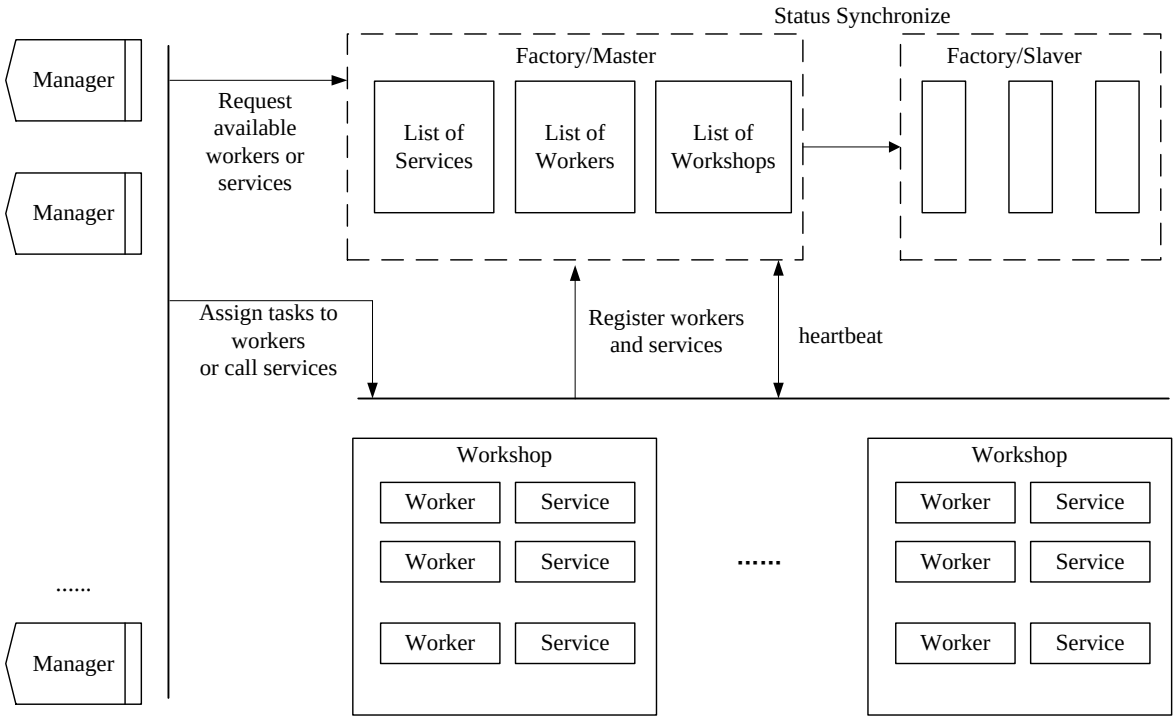

图 4-2 OpenCluster 的整体框架

在工厂中维护车间、工人和服务列表，车间、服务和工人向工厂注册，并定时向工厂发生心跳数据，工厂定时清理过时的车间、工人和服务节点，因此工厂、车间、工人和服务以驻留程序运行。车间代表了物理的计算节点，定时向工厂发送心跳和资源的使用情况。Manager 代表了一个计算任务，Manager 启动后，从工厂可以获得特定类型的工人，将计算任务切分成小的子任务分发给工人，工人完成任务后返回结果给 Manager，Manager 可以调用其它的 Manager 进行下个批量计算工作。在此期间，Manager 和工人都通过服务列表获得某一特定服务并调用。

4.2 关键技术

4.2.1 工厂单点故障

工厂是整个系统的核心部件，可以启动多个工厂实例，形成一主多备的关系。多个工厂实例同步备份服务和工人数据，并实现了简单的领导者选举机制。因此当 Master Factory 发生故障时，集群重新选取领导者，并实时切换到该 Factory。因此不存在工厂的单点故障。

OpenCluster 使用的工厂领导者选取机制与 Paxos 算法不同，Paxos 在维持领导者选举或者变量修改一致性上，采取一种类似议会投票的过半同意机制，比如设定一个领导者，需要将此看作一个议案，征求过半同意，每个节点通过一个议案会有编号记录，再次收到此领导者的不同人选，发现已经有编号记录便驳回，最后以多数通过的结果为准。而 OpenCluster 选取领导者采取的是一种谦让方式，当启动一个工厂实例，首先会询问其它工

厂实例是否愿意当领导者，没有人愿意它才担任；如果已经有了，它就谦让。正因为大家都谦让，不互相争抢，领导者之间能避免冲突保持一致性。一旦确定了领导者，工人、服务、经理都只跟该领导者打交道，所有对工厂中变量的操作都是通过该领导者进行，该领导者会自动复制变量修改的操作到其他工厂实例上。工厂负责工人和服务地址的注册与查找，相当于目录服务，工厂不转发请求，压力较小。

4.2.2 多类型工作节点

科学数据处理中，包含了多个处理环节，每个环节使用不同的处理程序。如在 MUSER 数据处理中，原数据的预处理，标识，积分处理可以在普通的计算节点上完成，而 Clean，图像生成需要在安装了 GPU 的计算节点上完成。这些不同的处理程序，需要特定类型的工人才能完成。因此系统应该允许多种类型的工人存在，并能根据计算需要获取指定类型的工人。这也要求工人实例在启动时，向工厂注册，标识自己是什么类型。工人实例可以是单个节点上的多个进程，也可以是多个节点中上的程序。

在主流的分布式计算，如 Hadoop、Spark 和 Storm 中，可以将计算任务 Map 和 Reduce，中间的计算任务分发到不同的节点上运行。我们可以定义数据如何 Map，如何 Reduce，但不能将任务 Map 到指定节点上，这也是我们为什么选择自己开发天文分布式计算框架的一个原因。

4.2.3 工人进程的数量

工人实例以进程方式启动，在单台计算节点上可启动多个同类型或不同类型的工人进程。多进程下再分多线程对性能提升的意义不大，反而在同步，加锁的地方会使编码难度变大。实际上一台机器 8 个进程，每个进程再开 8 个线程，总体跟 8 到 10 个进程的效果差不多。通常情况下单个节点上工人进程的数量设置成和节点的 CPU 的核心数一致。

4.2.4 批量计算与流式计算

我们把 Hadoop，Spark 归为批量计算，Storm，Spark Streaming 归为实时计算或者流式计算。两者的区别更多是从业务的数据对象而言，批量计算通常是分析处理存在的数据，实时计算的数据对象是不断生成的事件（数据，消息）。批量计算完成分析处理后退出，而实时流式计算不退出，循环处理新接收的数据，同时这个过程分散在多台计算机上并行完成，看上去就像数据连续不断地流经多个计算节点处理，形成一个实时流计算系统。

在 OpenCluster 中批量计算和流式计算的不同在于数据对象和 Manager 如何获取数据对象。比如经典的计算文本中单词出现的数量统计，使用批量计算，Manager 首先将需要计算的文本按行拆分，分发到工人节点上计算，Manager 统计汇总，输出计算结果，最后退出。而如果统计的网址访问流量日志，数据是通过 Socket 发生，则需要在 Manager 中定

义 Socket Server，当有新数据到达时，分发给工人处理，同时等待下一个数据到达，除非满足显示退出条件，否则永不退出。

4.2.5 心跳检测

心跳是分布式计算故障检测最常见的一种方式。如果工厂想确定工人节点是否发生故障，那么在每隔一段时间，比如 2 秒，向工人节点发送一个心跳包。如果工人节点一起正常，则响应工厂的心跳包；否则，工厂重试一定次数后认为工人节点发生故障。在这个过程中工厂与工人节点包含了双向的网络通信，为了减少心跳的网络通信次数，在工人节点启动后，启动单独的线程周期地向工厂发生心跳数据，在工厂中维护一个心跳列表，工厂周期性的检测心跳数据是否过期，这样工厂和工人间只有一次网络通信。

4.2.6 服务融合

集群中的 Manager 和 Worker 在数据处理过程中需要调用服务，这些服务可能已经存在的，并且是用多种不同的编程语言编写。这就需要有一种通用的通信模型，能够支持多种开发语言。因此我们选择使用 ZeroMQ 编写通信层，将服务注册到工厂中。注册时，服务和工人实例相似，需要提供服务的类型和地址。

节点启动后，向工厂注册服务，在单个节点上可以启动多个服务实例。而服务消费者（即 Manager 和 Worker），首先从工厂中获取服务地址列表，在本地缓存提供者列表，接着从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。服务消费者与服务节点的通信模型采用经典的 Request/Reply 通信模型。服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到工厂。

4.3 编程接口

OpenCluster 提供简单的编程接口（API）来帮助用户快速开发分布式应用程序。编写一个 OpenCluster 应用程序，首先必须通过继承基类 Manager 和 Worker 来实现用户自定义的 Manager 类和自定义的 Worker 类。每个 OpenCluster 应用程序由一个包含了 Main 方法的 Manager 启动运行程序，Manager 将任务拆分，并分发到特定的工人。Manager 和 Worker 的类的定义如图 4-3 所示，（红色框标注了必须实现的方法）。

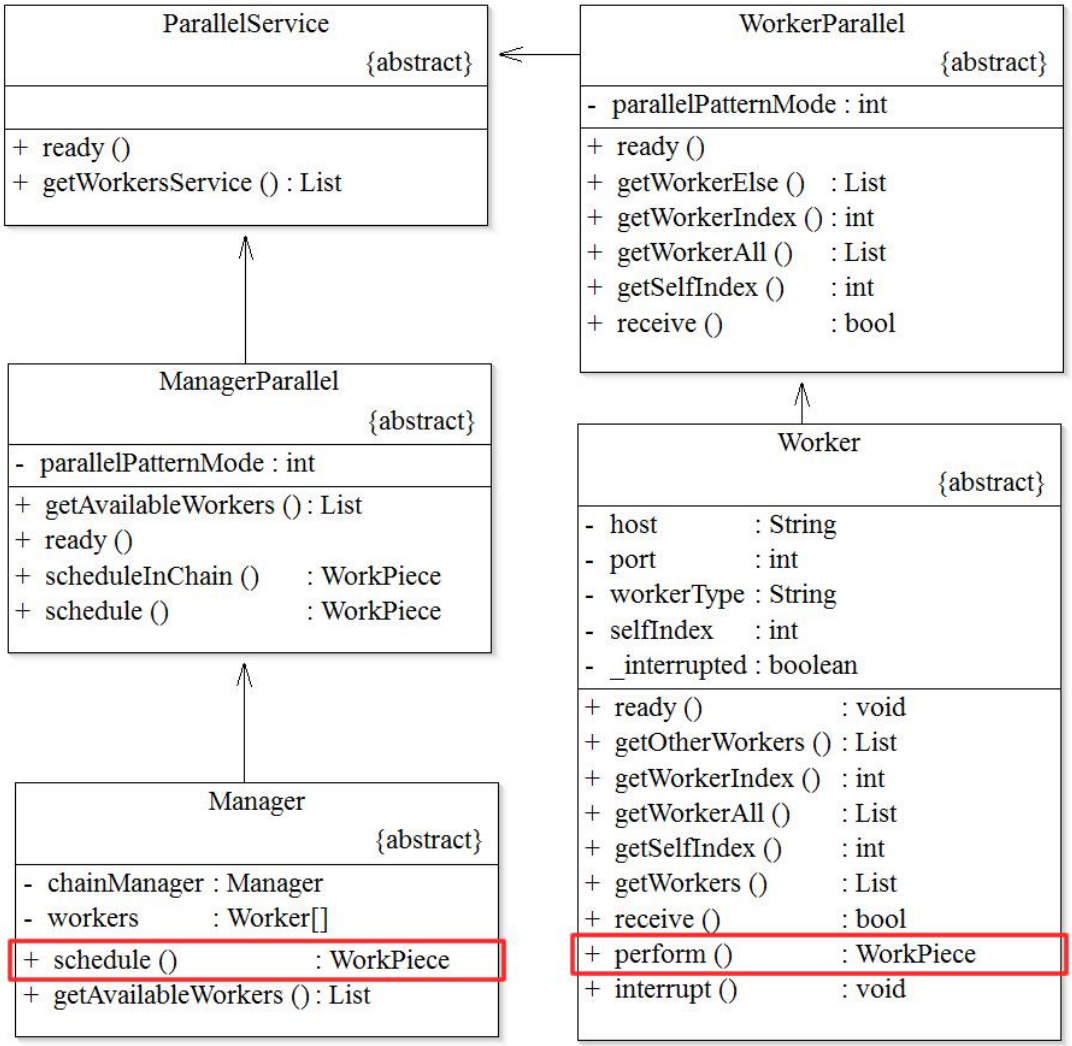

图 4-3 Manager 和 Worker 的类图

Manager 基类主要包含两个方法：

- getAvailableWorkers(worker_type)，调用这个方法返回可用的类型为 worker_type 的工人节点引用；
- schedule(task)，是一个空的函数，用户需要重写这个函数，将任务拆分，并将任务派发到工人节点。

Worker 基类主要包含了如下几个方法：

- ready(type, host, port)，当执行这个函数，表示该工人实例已经准备就绪，等待 Manager 调用，该实例在参数 host 和 port 指定的地址和端口上监听网络连接；
- perform(task)，是一个空的函数，用户需要重写这个函数，在这个函数是子任务执行的具体计算逻辑，参数 task 是需要计算的任务，函数返回计算的结果；
- interrupt()，执行该函数，可以中断正在执行的工人实例。

如上，使用 OpenCluster 开发分布式应用在框架方面的工作非常少，下面以经典的文本中单词数量的例子来进一步说明 OpenCluster 的编程接口的使用，如下表格 4-1 和表格

4-2 所示。

表格 4-1 使用 OpenCluster 实现的统计单词数量的 Manager 的伪代码

```
1  class WordCountManager(Manager) :
2      def __init__ (self):
3          super(WordCountManager,self).__init__()
4      def schedule(self, task):
5          workers = elf.getAvailableWorkers("WordCount")
6          wordcount = dict()
7          # split task into sub tasks
8          for worker in workers :
9              subwordcount = worker.perform(tasks[i])
10             wordcount.join(subwordcount)
11         return wordcount
12 if __name__ == "__main__" :
13     manager = WordCountManager()
14     result = manager.schedule("/logs/wordcount.log")
15     print result
```

表格 4-2 使用 OpenCluster 实现的统计单词数量的 Worker 的伪代码

```
1  class WordCountWorker (Worker) :
2      def __init__(self,name):
3          super(WordCountWorker,self).__init__()
4          self.name = name
5      def perform (self, work):
6          filepath = work.getObj("filepath")
7          offset = work.getObj("offset")
8          size = work.getObj("size")
9          wordcount = dict()
10         # count the number of words
11         wh = WorkPiece(True)
12         wh.setObj("word",wordcount)
13         return wh
14 if __name__ == "__main__" :
15     worker = WordCountWorker("worker1")
16     worker.ready("WordCount","*",9280)
17
```

类 WordCountManager 继承于 Manager 的基类，schedule（4 - 11 行）方法，首先查询当前集群中可用的类型为“WordCount”的工人，接着拆分任务，并将子任务分派给工人，最后将每个工人的结果合并成整个 wordcount(第 10 行)并返回结果。

类 WordCountWorker 继承于 Worker 的基类，perform（5 - 13 行）方法，是具体的计算的逻辑，从 WordCountManager 接收 task，参数 task 包含了要处理的文件、偏移和大小。然后打开文件，计算统计每个单词的数量，最后返回包含键-值对<Word, Count>的列表。在

主入口（15 - 16 行），首先声明了一个工人实例，接着调用了 ready 函数后，该类型为 WordCount 的工人实例将在所有网络接口的 9280 端口监听。

4.4 部署与监控

OpenCluster 设计之初的一个目标就是简化应用部署和管理的过程，OpenCluster 很容易部署在系统上。我们只需要将所有文件复制到指定目录，并设置适当的权限。同时我们使用 Web.py 框架编写了基于 Web 的 OpenCluster 集群监控界面。通过界面可以监测 OpenCluster 集群的运行信息，包含了所有操作节点的状态、任务、服务和工人信息，以及任务状态更改的更新和执行主机信息以及任务的统计数据。Web 服务器可以运行作为一个独立的守护进程运行，或者借助于 Web.py 实现的 WSGI 在任何兼容它的服务器上运行。此外，Web 界面另外提供的功能来控制工人，服务实例，如启动新的工人和服务实例，关闭这些实例。监控界面的其他功能包括图形显示资源的使用和任务的执行情况等。图 4-4 显示了一个 OpenCluster 监控界面的截图。OpenCluster 提供多种配置调整工厂、任务分配者、服务和工人的参数，这些参数可以通过命令行参数和配置文件定义。配置文件 config.ini 定义了这些参数的默认值。运行一个集群，首先用户必须创建一个工厂对象并执行工厂的启动函数。

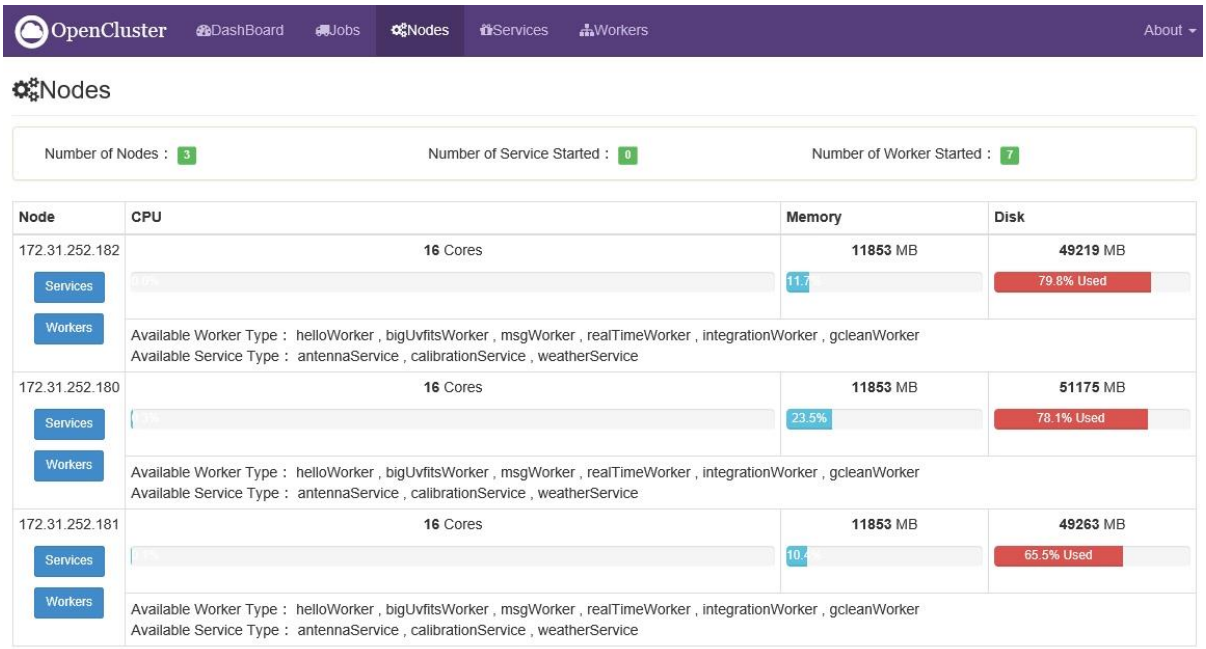

图 4-4 OpenCluster 的 Web 监控界面

4.5 实验

OpenCluster 采用 Python 语言开发，和更低层次的语言相比较可能存在性能上的差异。

通信层使用的是纯 Python 代码实现的分布式对象系统 Pyro4，本章节将对 OpenCluster 的通信效率和使用 MPICH2 的 C 语言实现同样数据通信的效率进行测试和比较。

4.5.1 环境

实验环境使用 5 台配合相同的服务器，每台服务器的配置相同，都配置了千兆网卡；每台服务器配备了 16G DDR2 内存，两路的 Intel Xeon E5-2640 v2 CPUs, 2.0 GHz, 16 核的 CPU，1TB 的硬盘。操作系统安装了 CentOS7.0。 在其中一个节点作为工厂节点，同时提供 Web 监控，其他四个节点作为工作节点。

4.5.2 测试

在测试中，我们比较了 MPI 和 OpenCluster 在二进制数据传输方面的效率。使用了 5 台服务器，每台服务器启动了 16 个工人实例来评估 OpenCluster 的性能。在测试 MPI（使用 MPICH）时，同样使用五个节点，每个节点上启动了 16 个进程。两个平台的机器配置和网络环境是完全相同的。测试中，模拟消息发送，发送一个特定大小的块从一个节点到其他节点（或从一个工人节点到其他工人节点）。我们分别测试了块大小为 100 KB 和 5MB 的传输效率（如下图 4-5 和图 4-6），测试结果表面当数据大小为 100 K 时，OpenCluster 的性能表现要比 MPI 差得多。当数据大小为 5M 时，OpenCluster 和 MPI 的性能表现接近。

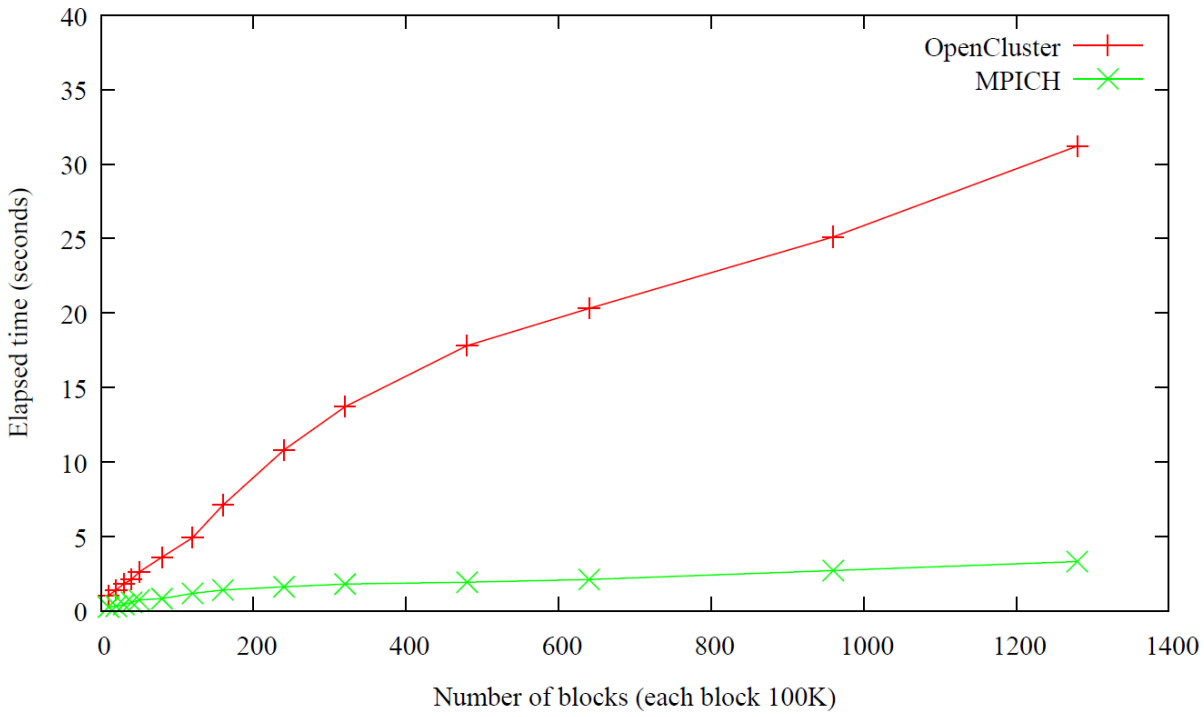

图 4-5 数据块大小为 100K，传输时间随传输数据量的变化图

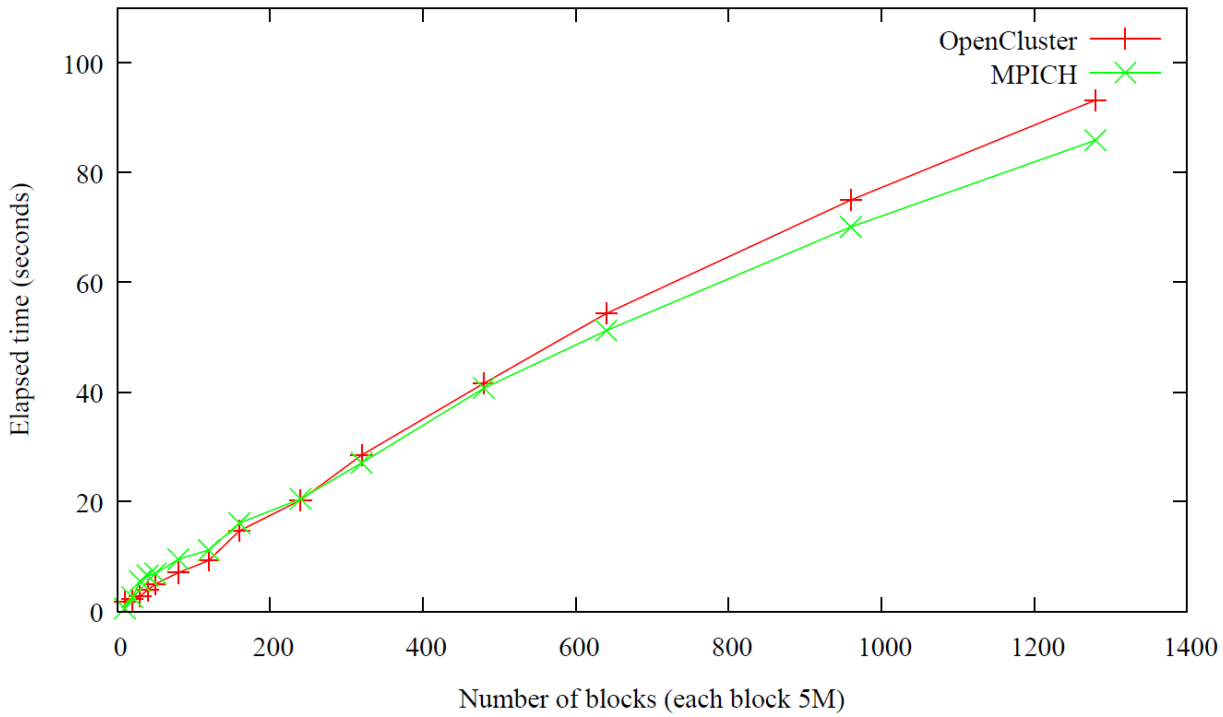

图 4-6 数据块大小为 5M，传输时间随传输数据量的变化图

综上，射电天文数据处理中，有较多的二进制数据的传输，而由数据性能趋势可以看出，OpenCluster 在数据库大小为 5M 时的有较好的性能表现，并且具有很好的扩展性特性，这一点很好的满足了 MUSER 射电数据的高性能、高扩展的数据处理需求。

4.6 讨论

受传统的工业制造操作流程的启发，本章节介绍了适合天文海量数据处理的框架 OpenCluster 的设计的整体思路和关键技术，阐述了 OpenCluster 的使用方法和界面监控，测试了其传输效率，选择了在天文计算中广泛使用到 Python 语言编写，网络通信框架使用了 Pyro4 这种远程对象调用的方法，能自动的对 Python 中对象进行远程代理，可以像操作本地对象一样操作远程对象，能极大的提高编程效率，并容易集成到自己的事件循环中。

在 OpenCluster 最初的设计中，Manager 从 Factory 中获得可用的工人连接后，就将任务分发给工人，它并不知道当前的工人所在的节点的计算负载（如 CPU 和内存的使用情况）。尽管我们可以定时收集集群中节点的资源使用情况，但我们也无法对任务计算的资源进行隔离。特别是节点的网络连接数，当有一个大的任务分成很多小的子任务时，如果将这些任务全部推到一个节点上，往往会因为节点的网络连接数到达上限，而造成节点的进程崩溃。在第 6 章中，将对 OpenCluster 的资源调度进行研究，主要解决任务拥塞、任务优先级调度和资源隔离等问题。

4.7 本章小结

本章节讨论了适合天文海量数据处理框架 OpenCluster 的设计和实现，使用了谦让的领导者选择机制解决 Factory 的单点故障问题；为了区分工人实例，每个工人实例都设置了其所属的类型，而这是传统开源分布式计算框架所不具备的；根据获取数据的方式的不同，OpenCluster 能灵活的适应批量计算和流式计算；引入了数据服务的设计，Manager 和 Worker 实例可以在数据拆分、分发或计算的过程中，调用公共服务和查询；OpenCluster 也提供了 Web 方式的监控管理界面，辅助集群管理者更快的了解集群内节点的状态信息和任务的执行情况。

结合第 3 章横向对比和本章纵向的分析，可以看出分布式 OpenCluster 能够很好的满足 MUSER 高扩展性和高性能的需要，另一方面，通过实际的部署和管理来看，OpenCluster 的安装部署和管理也比开源分布式计算框架如 Hadoop、Spark 要简单容易，在第 5 章中将讨论 OpenCluster 在 MUSER 的具体应用，如何使用简单易用的 API 方便地集成已有的数据处理代码。

第5章 OpenCluster 在 MUSER 中的应用研究

MUSER 的数据处理，根据处理对象主要包含日常准实时数据的处理和历史观测数据的处理。这两种模式，除了数据对象不同，采用的处理方法基本一致，另外实时数据处理不需要积分，在历史观测数据会根据不同的需要，会采用不同的积分时间进行数据处理。同时实时数据的处理要比历史数据处理有更高的优先级。

在上一章中主要讨论分布式计算框架 OpenCluster 的设计和实现，本章节主要讨论 OpenCluster 在 MUSER 的具体应用，工作主要包括：1) 介绍了 MUSER 数据处理流程；2) 使用 OpenCluster 设计历史数据的处理 Pipeline；3) 使用 OpenCluster 设计日常准实时数据处理的 Pipeline；4) 使用 Web.py 构建 MUSER 的参数配置和任务提交的 Web 应用系统；5)用 UVFITS 文件生成的处理过程讨论了 OpenCluster 的线性扩展性。

5.1 MUSER 数据处理 Pipeline

MUSER 是采用综合孔径技术方法对太阳进行成像观测的射电望远镜，数据处理方法与传统的综合孔径望远镜处理方法类似。从数字接收机接收的原始数据的处理要经过预处理，天气数据标识，数据标定，校验，洁化，成图等多次迭代操作，最后生成 UVFITS 和发布到 Web 的图像，如下图 5-1 所示。

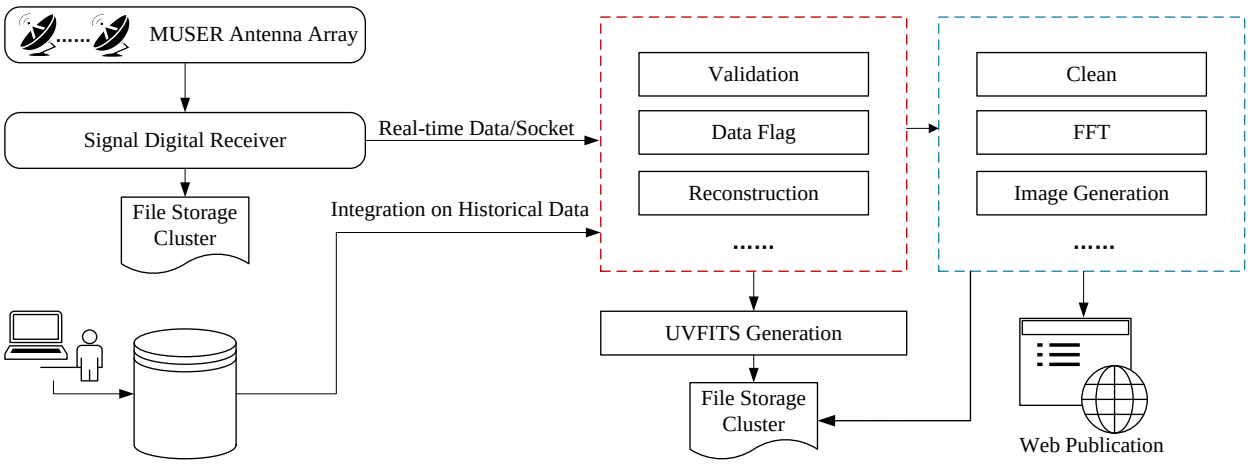

图 5-1 MUSER 数据处理 Pipeline

观测数据有两个来源，一是以 Socket/TCP 方式向外写出的实时数据，二是存储在磁盘的历史观测数据文件，两种数据的过程一致，其中红色虚线标注的处理过程可以在普通的计算节点上运行，蓝色虚线框内的 CLEAN、成图等操作需要在安装了 GPU 卡的机器上执

行。另外在数据处理过程中也需要查询卫星，星表，天线状态等其他数据源。

MUSER 是采用综合孔径技术方法对太阳进行成像观测的射电望远镜，数据从相关器输出后，一般需要进行相位调整、条纹停止、FLAG、格式转换、成像、去卷积等[113]。每部分中需要进行的工作简单介绍如下：

1）相位调整和条纹停止

MUSER 在相关器内已经完成了延时补偿，保证了来自太阳的射电信号同时到达相关器进行复相关运算。相关器内同时保留了每路信号的相位调整功能和条纹停止功能，理想情况下，通过每路信号的相位调整，确保所有基线的相位中心一致；通过条纹停止功能，补偿由于地球自转引起的基线的相位变化，确保在源不变的情况下，使得输出相位稳定。但在实际观测中，尤其是调试过程中，对于基线的相位中心确定，需要频繁的调整来逐渐逼近真实相位中心，因此可以把每路的相位调整工作在数据预处理中实现。

2）FLAG

MUSER-I 有 40 面，MUSER-II 有 60 面天线、接收机及规模庞大的相关器组成，日常观测中，即使单独的天线或接收通路出现故障而不工作，根据综合孔径成像原理，在数据处理中剔除这些不工作的数据也可对观测源成像。因此，在日常观测中，要记录每个天线的工作状态，每路接收信号的强弱变化，如发现某天线或通路不工作，在数据中标记掉这些数据，和 CASA 软件中的 FLAG 过程一样。

3）格式转换

MUSER 的原始文件为自定义的格式，为了方便利用现有通用软件（如 CASA）处理，需要把自定义格式转换为标准的 UVFITS、FITS-IDI 等格式[114]。

4）网格化（GRIDDING）

天线的位置决定 UV 平面上复可见度函数的位置，由于 MUSER 的天线是采用的螺旋型排列，而为了做快速傅立叶变换，要求 UV 平面上的复可见度函数需按网格化排列。这样，需要把落在网格其他位置中的点内插到网格节点上。

5）积分

MUSER-I 完成一次循环观测为 25ms, 一般需要对宁静太阳时期的数据需要积分到 1 秒钟再进行处理。即为把做完条纹停止和相位调整后的数据在对应频率、对应极化的数据进行累加求平均。实时数据处理不需要积分过程。

6）成像

成像过程把预处理过的可见度函数进行快速傅立叶变换得到脏图（Dirty map）。同时，把 UV 平面上的复可见度函数的幅值设 1，相位设 0，再进行快速傅立叶变换，得到脏束（Dirty Beam）。

7）CLEAN

使用 CLEAN 算法，根据得到的脏束对脏图进行去卷积处理，最大可能的还原真实的观测图像。

5.2 历史观测数据处理

MUSER 支持使用与 CASA 操作类似的命令行（CLI）在单机上做数据处理，同时提供了基于 Web 的数据处理任务提交系统，为了提高代码的重用率，这两种方式都使用了同样的处理代码。不同的是通过 Web 方式提交任务后，系统有专门的任务队列监控，一旦有新的任务，将数据处理任务调度到分布式数据处理系统 OpenCluster 中。

MUSER 产生的原始文件是自定义格式的，需要被转换为通用工具可以直接打开的 FITS 文件。在射电天文领域，被广泛采用的是随机组（Random groups）结构的 FITS 文件。因为主要保存 UV 复可见条纹数据，也经常被称为 UVFITS 文件。首先完成了 UVFITS 文件生成函数 generateUVFITS 的编写，generateUVFITS 接受时间段，高低频阵等参数，返回生成 UVFITS 文件的保存路径，generateUVFITS 函数可以成功运行在单机系统中。为了在 OpenCluster 平台中使用 generateUVFITS 函数，以分布式方式生成 UVFITS 文件，如在 4.3 编程接口章节所述，定义了工人类 UVFITSWorker 和任务分配类 UVFITSManager。

1），UVFITSWorker，UVFITSWorker 继承于 Worker 类，在 perform()函数中，也就是任务执行的函数中，调用 generateUVFITS 函数完成 UVFITS 文件的生成。

2）UVFITSManager，UVFITSManager 继承于 Manager 类，在 schedule()函数中，将任务拆分成小的子任务，将子任务按照不同的调度模式（在第 6 章中讨论调度模式），调度到 UVFITSWorker 实例，并获得由 UVFITSWorker 执行后返回的结果。

天文学家通常需要某一个时间段内的 UVFITS 文件，因此 Web 应用提供了 UVFITS 生成的任务提交界面，用户选择时间段及其他参数，系统生成任务后，将任务加入到任务队列。UVFITSTaskDispatcher，是任务调度器，作为后台守护进程运行，为每一个任务生成一个 UVFITSManager 并加入到线程池中，UVFITSManager 再将任务拆分成小的任务，调度到分布式集群中的 UVFITSWorker 的工人实例执行。任务完成后更新任务队列。在界面上可以查看任务的执行进度，对失败的任务可以重新调度。具体过程如下图 5-2 所示。

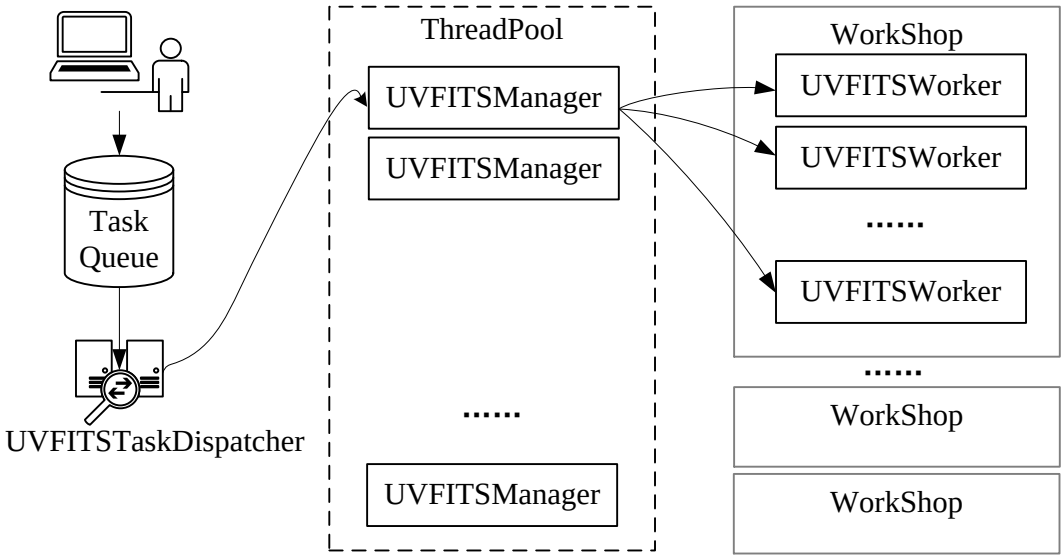

图 5-2 MUSER 历史数据处理组件图

为了提高任务调度器（UVFITSTaskDispatcher）的稳定性，在第 8 章中，将其介绍如何实现结合 Mesos，Marathon 和 Docker 实现了 Web 应用和任务调度器长服务的运行。

5.3 实时计算

MUSER 实时数据处理的是每 5 秒钟一个完整帧，帧的格式及处理过程和历史数据处理过程一致，需要进行预处理、网格化、成像、洁化等操作，不需要积分过程。首先编写了对一帧的数据处理程序 generateImage()函数，并在单机上测试成功，该函数接收一个帧的原始数据，处理成功后将生产的图像保存在预先配置的目录，返回生成图片的路径。根据 OpenCluster 的接口描述，定义了任务分配类 MUSERImageManager 和工人类 MUSERImageWorker：

1）MUSERImageManager，MUSERImageManager 继承于 Manager 类，在 schedule()函数中，将任务拆分成小的子任务，将子任务按照不同的调度模式，调度到 MUSERImageWorker 实例，并获得由 MUSERImageWorker 执行后返回的结果。

2）MUSERImageWorker，MUSERImageWorker 继承于 Worker 类，在 perform()函数中，也就是任务执行的函数中，调用 generateImage 函数生成发布的图像。

MUSER 的实时数据是采用 TCP/IP 协议，作为客户端向外写出数据流。因此首先需要构建一个 TCP Server 的数据接收端（MUSERSocketServer）。MUSERSocketServer 对接收到的数据做完整性的验证，如果是一个完整的数据帧，则创建 MUSERImageManager 任务分配对象，并加入到线程池，MUSERImageManager 再调度 MUSERImageWorker 完成任务的执行。MUSER 的实时数据的组件图如下图 5-3 所示。

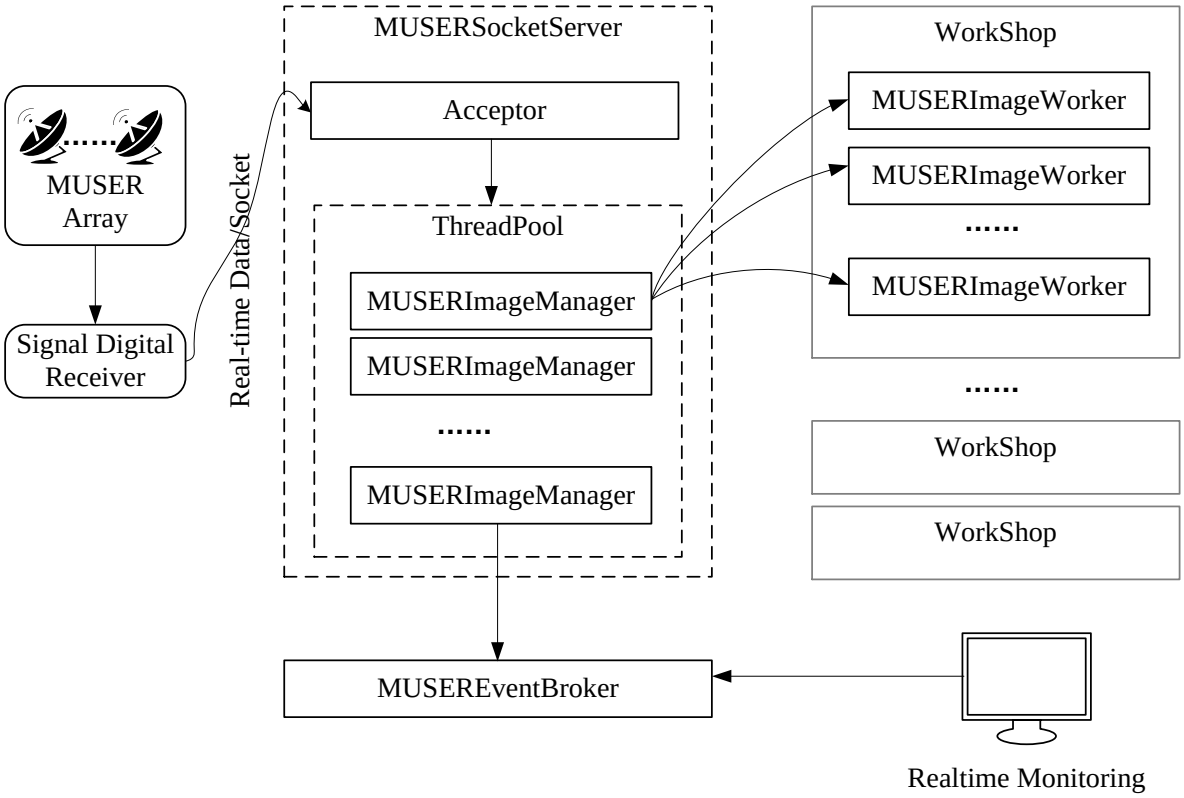

图 5-3 MUSER 实时数据处理组件图

MUSERImageWorker 在完成成图的过程中，同时产生可见度数据的互相关数据与自相关数据，作为返回值以 numpy 数组的形式返回给 MUSERImageManager，MUSERImageManager 需要等待分配的一个完整帧的所有子任务完成后，将互相关和自相关数据，发送到事件接收器（MUSEREventBroker）。MUSEREventBroker 使用 ZeroMQ 的 PUB 通信端将数据发布，采用 Publish 的方式，主要考虑到后期更多的系统接入到展示实时数据处理结果。实时数据监控作为订阅端从 MUSEREventBroker 订阅互相关和自相关数据，以及图像文件的路径信息。

MUSEREventBroker 采用一主一备的双子星模型，提供高可靠性的保障。客户端首先尝试连接一个，在无法获得服务时，连接备用节点。在任何时间，主备节点的一个（活跃的那个）为客户端程序提供服务。另一个（不活跃的那个）什么都不做。

5.4 界面操作

MUSER 软件系统采用 MySQL 数据库来存储数据处理中的若干关键参数，如光纤传输时延、气象数据等，同时为了提高查询速度，也支持将数据导出到 Redis 内存数据库。用户界面主要实现数据处理任务提交、天线状态维护、自动气象站和卫星数据修正配置等功能。

目前，MUSER 已经实现了与 CASA 操作类似的命令行（CLI）接口[11]，但命令行方

式存在命令多，表现方式不直观等不足。为了进一步提高 MUSER 数据处理系统的使用效率，我们使用 Web.py 开发了更为友好的 MUSER 数据处理系统交互界面。

Web.py 是一个纯 Python 实现的轻量级 Web 应用开发框架，支持 MVC 模式开发，可通过简单的编程将普通的 Python 类映射为请求处理对象，原生的支持编写 REST（REpresentational State Transfer）架构风格的 Web 服务，当前的版本是 0.38。Web.py 提供了最小化的 Web 服务器可以运行作为一个独立的守护进程运行，或者借助于 WSGI（Web Server Gateway Interface）在任何兼容它的服务器上运行。

基于 Web.py 开发的 MUSER 数据处理系统用户界面已基本开发完成并应用到实际项目数据处理中，前端使用 Bootstrap 构建了响应式页面，系统界面如下图 5-4 所示。系统实现了在 Web 界面下配置系统参数，为多方面数据（原始数据，异常天线信息，仪器状态，天气）的录入，维护，查询和使用提供了方便，确保了后续数据处理的有效性和可靠性。同时，Web 界面提供的数据处理参数的配置，方便了复杂任务的提交，与命令行界面相比，具有更好交互性和可用性。

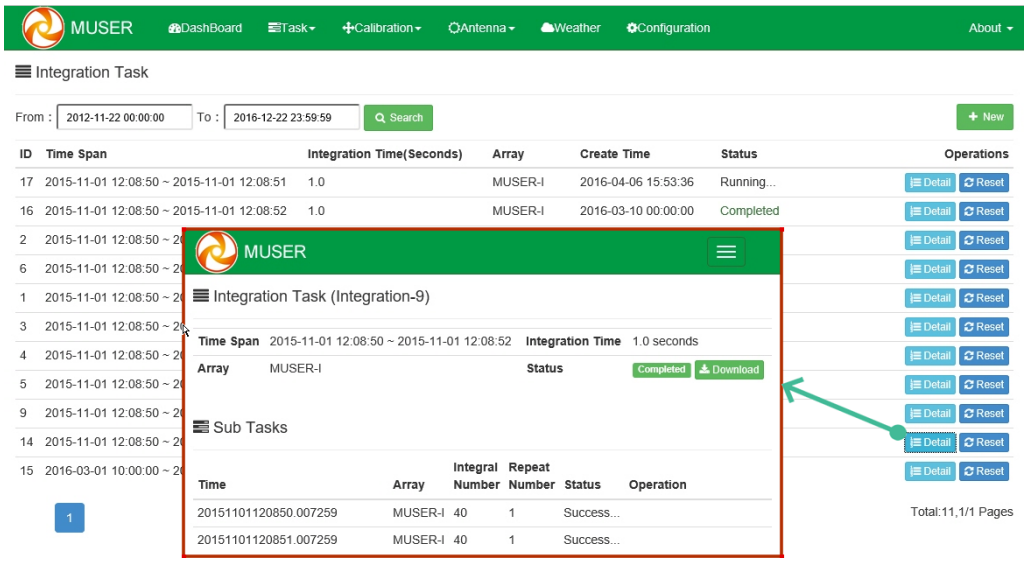

图 5-4 MUSER 数据处理系统用户界面

5.5 实验

实验环境的服务器的详细配置和第 4 章的实验部分的环境一致，测试数据选取了原始数据中的 1000 帧，测试在 OpenCluster 分布式计算环境中，使用不同的工人实例数量，完成这 1000 帧的 UVFITS 文件生成所消耗的时间。测试结果如下图 5-5 所示。

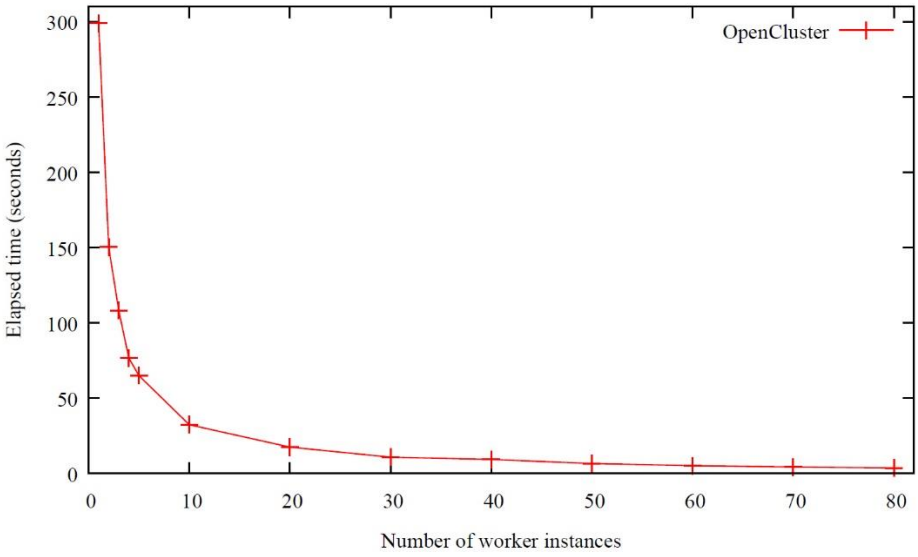

图 5-5 完成 1000 帧 UVFITS 文件生成所需时间随工人实例数量的变化图

从图中可以看出，随着工人实例数量的增加，完成 1000 帧 UVFITS 文件生成的所需时间逐步减少，加速比基本成线性关系。当工人实例超过 80 个后，所需时间没有减少，测试服务器均为 16 核 CPU，5 台机器的核心数为 80 个，当工人实例数量超过核心数，并没有性能提升，主要是因为 CPU 在多进程间切换同样会有很大的代价。

5.6 讨论

OpenCluster 已经成功运行在 MUSER 的内部集群中，用于历史数据和实时数据的分布式处理，能够将在单机运行的 Python 数据处理程序，通过 OpenCluster 提供的编程接口，快速的移植到分布式计算环境中。但在 OpenCluster 本身的集群管理中，工人实例的管理还较为薄弱，比如对崩溃的工人实例的恢复；在单个节点上，维持一定数量的工人实例；多种类型工人节点的资源分配和隔离等，还没有完善的功能。在第 6 章中讨论在 OpenCluster 中几种资源调度模式。

5.7 本章小结

本章首先简单地介绍了 MUSER 的数据处理流程，介绍了使用 OpenCluster 设计历史数据和实时数据的分布式处理流程和方法。实时数据处理中，设计了数据接收器服务端接收 MUSER 作为客户端发送的数据，设计了事件接收器，接收实时数据处理结果将数据发布到实时监控端。构建 MUSER 的参数配置和历史任务提交的 Web 应用系统。最后测试了在固定大小原始观测帧大小的情况下，随工人实例数量增加 UVFITS 文件生成的所需要的时间，实验表明 OpenCluster 具有较好的线性加速比。

第6章 OpenCluster 资源调度研究

前面章节研究了 Spark 在 MUSER 数据处理中的应用，在互联网领域广泛应用的开源的分布式计算平台在编程、管理和维护上存在较高的技术难度，数据处理对象也更适用于文本的分析，不适合处理射电天文的海量数据处理，因此我们开发了分布式计算框架 OpenCluster。通常在天文项目组的集群计算环境中，运行着多个计算平台或任务，考虑到资源利用率，运维成本，数据共享等因素，OpenCluster 应该具有这样功能和接口，和其他的计算平台共享集群的资源。这就需要一套能够感知和协调整个计算集群资源的分布式软件系统。在开源领域中，广泛使用的资源统一调度系统是 Yarn 和 Mesos。

Yarn 和 Mesos 都是 Apache 基金会下的集群资源管理工具，它通过抽象主机的 CPU、内存、存储等计算资源来搭建一套高效、容错、弹性的分布式系统。本章讨论基于 Mesos 使 OpenCluster 支持粗粒度和细粒度的任务调度，解决集群资源统一调度、任务优先级调度及异构资源的调度问题。

6.1 概述

在 OpenCluster 中，任务由 Manager 进行拆分，拆分的规则由开发者来定义，任务拆分后提交到 OpenCluster 的调度器。Manager 在启动时可以指定调度模式，目前 OpenCluster 支持五种调度模式：

1）本地模式（Local），单进程单线程模式，Manager 直接调用 Worker 的任务执行方法，Manager 和 Worker 属同一进程。

2）多线程模式（Thread），线程池的多线程模式，Manager 启动线程池，以多线程的方式执行 Worker 中定义的计算方法。

3）独立集群模式（Standalone），使用 OpenCluster 本身提供的集群管理方法调度工人实例，完成分布式计算。

4）Mesos 集群模式（Mesos），使用 Mesos 集群的资源调度方法，为每一个 Manager 创建一个 Mesos 框架（Framework）。

5）集中工厂模式（Factory），使用 Mesos 集群的资源调度方法，Kafka 作为任务仓库，为所有任务创建一个 Mesos 框架，进行全局调度。

在本地和多线程模式中，工人实例均是在提交 Manager 的节点上运行。本地模式启动一个工人实例，一般是用来测试算法的正确性。多线程模式是以线程池的方式运行多个工

人实例线程，一般是用来测试并行执行的算法的正确性。这两种主要是用来测试，不涉及资源的调度，在大规模分布式计算生产环境中，一般不使用。在余下的章节中，将主要讨论后面三种集群调度模式。

6.2 集群独立调度模式

在独立集群模式（Standalone）中，工人，服务和节点都是作为 Pyro4 服务启动，启动首先向工厂注册，在工厂中就保存了工人，服务和节点的服务列表。用户定义的 Manager 将大的计算任务切分成小的任务后，从 Factory 中获得可用的工人服务连接后，调用工人服务，将任务分发给工人，支持同步和异步地调用方式，工人实例完成计算后，将结果返回到 Manager，如下图 6-1 所示。

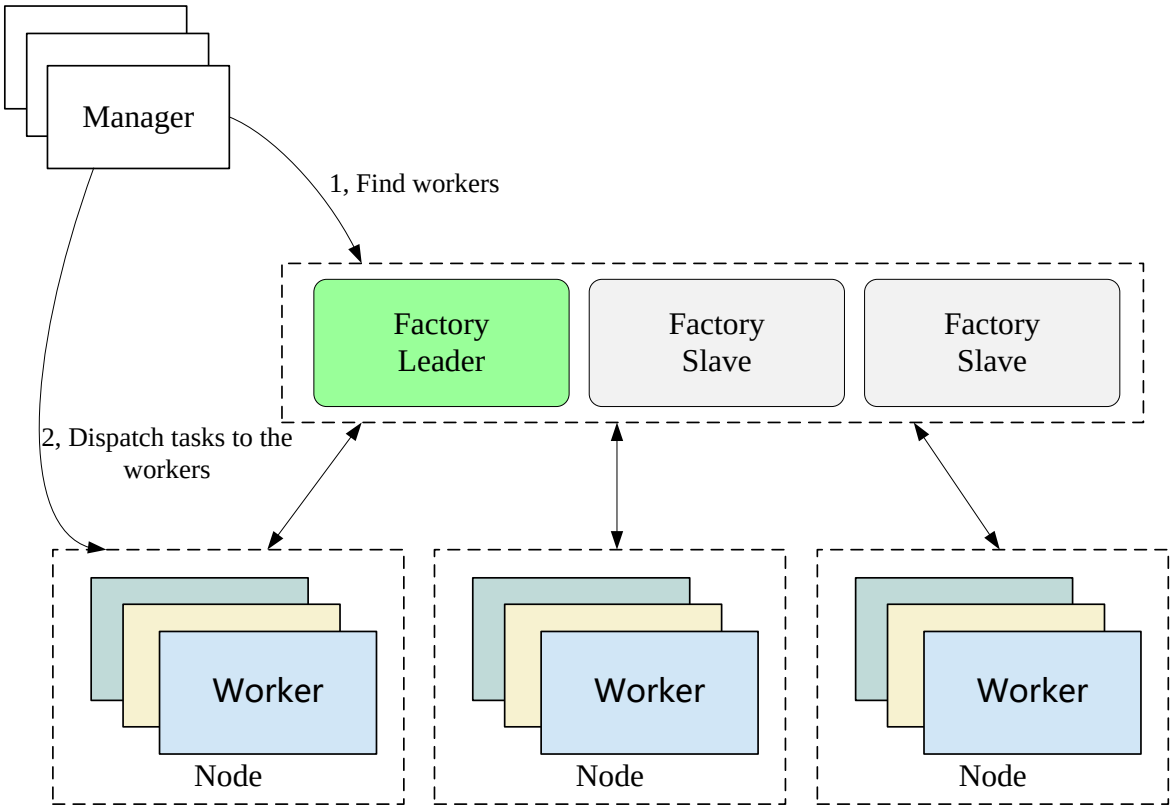

图 6-1 OpenCluster 独立调度模式

工人，服务和节点与工厂之间使用心跳机制，工厂能够感知当前集群可用的工人，服务的类型、数量。但它并不知道当前的工人所在节点的计算负载（如 CPU 和内存的使用情况）。节点服务定时向工厂发送当前节点的资源使用情况，但节点服务无法对节点上的工人计算的资源进行隔离，特别是节点的网络连接数。当有一个大的任务分成很多小的子任务时，如果集群中计算这批小任务的工人实例有限，每个工人实例可能会分配到非常多的任务，由于每次调度都需要建立一个网络连接，这个时候，往往会因为节点的网络连接数到达上限，而造成节点的工人实例进程崩溃。随着机器数量增加，软件管理越来越复杂，失

败管理维护越来越难，而且资源消耗非常不经济。

另外，在集群计算环境中，可能还运行着诸如 Hadoop，Spark 等其他计算平台，OpenCluster 需要和这些计算平台共享集群的计算资源。因此我们需要一个能够整合集群中的所有资源，根据应用的需要，动态分配资源的统一调度系统，Apache 基金会的 Mesos 和 Yarn 就是属于此类资源统一调度系统，Mesos 得益于 Google 的 Omega 系统设计的经验，构建了一个更好的两阶段的调度器，同时具有良好的 Python 编程接口，因此我们选用 Mesos。

6.3 基于 Mesos 的资源调度

在 Mesos 管理的集群中可以运行不同的分布式计算平台，如 Spark、Storm、Hadoop、Marathon 和 Chronos 等，这些不同的计算平台都作为框架使用 Mesos 提供的 SDK 向 Mesos 请求资源，然后自行调度计算任务。

Mesos 使用的两级调度架构和隔离技术让在同一个节点上运行多种不同类型的应用，应用程序调度程序及其执行器被称为 Framework（框架），框架接收到 Master 的资源提供（offer），允许用户编写调度算法，消耗提供的资源或者拒绝提供，继续等待另一个资源 offer。这个模型非常类似于单台机器上的同时运行的多个应用程序，他们在生成新线程向操作系统请求更多的 CPU 时间片或请求更多的内存。因此 Mesos 也被称作集群的操作系统。

编写基于 Mesos 的计算框架，需要实现调度器和执行器这两个组件。

调度器（Scheduler）：Scheduler 负责连接到 Mesos master，接受或者拒绝资源提供。Mesos 将调度的任务委派给 framework，而不是试图将所有的调度工作都放在 master 上。scheduler 可以根据在接到资源提供时是否有任务可以运行来接受或者拒绝一个资源提供。接收资源提供，向 Mesos Slaves 发送任务信息，获得任务执行的更新信息。

执行器（Executor）：从框架的环境变量配置中获得任务信息，在 Mesos Slaves 上运行任务。

6.3.1 Scheduler 设计

Mesos 框架可以使用 C、C++、Java/Scala 或 Python 编写，使用 Mesos 提供了直观的 API 方便用户编写自定义的框架。OpenCluster 使用 Python 编写，自定义调度器继承 Scheduler，实现 Scheduler 定义的接口，首先创建一个 FrameworkInfo，接着创建了 MesosSchedulerDriver，它是负责 Scheduler 和 Mesos Master 通信，维护 Scheduler 的生命周期，然后调用 MesosSchedulerDriver.start()方法启动这个调度器。

Scheduler 启动后，首先向 Master 注册，注册成功后调用 registered 函数；当重新与新的 Master 注册后，触发 reregistered 函数；断开连接后触发 disconnected 函数；当这个框架

请求的资源提供给这个框架时，调用 resourceOffers 函数；当资源请求被拒绝时调用 offerRescinded 函数；当发生如 slave 宕机，任务失败，任务结束等任务状态更新时 statusUpdate 函数被触发；当执行器发送消息到框架，触发 frameworkMessage 函数；当某个 slave 确定不能找到时调用（如设备故障，网络中断）slaveLost 函数，计算框架会在新的 slave 上重新启动所有任务的方式进行调度；执行器退出或者中断时调用 executorLost 函数；发生不能捕获的致命错误时调用 error 函数。

在所有的回调函数里面，最重要的是 resourceOffers，根据得到的 offers（每个 slave 都有 CPU，多少内存等资源），如果一个 offer 满足提交的任务所需的资源，就创建 Mesos 的调度任务，调度任务需要设置执行器，所用的资源，然后调用 MesosSchedulerDriver 的 launchTasks 函数提交任务。任务数据最终要发送到执行器，因此使用 lz4 的压缩库对数据进行压缩，降低传输成本。

更详细的框架开发指南可以参考官方文档[3]，在 Mesos1.0 及以上版本中，已支持使用 HTTP 的方式和 Mesos 交互。使用 HTTP 的方式，不需要本地库的支持，极大地方便了自定义组件的开发。

6.3.2 Executor 设计

执行器（Executor）是一个进程，它在 Mesos slave 节点上运行某个 Framework 的任务。当前内建的 Mesos 执行器允许框架执行 Shell 脚本和运行 Docker 容器。可以使用各种类型的语言来写新的执行器，将其绑定到 Framework 中，从而在某个任务需要时，由 Mesos slave 节点来取得执行器信息并启动执行器。如果运行一个普通的容器，或者命令行的应用，则不需要实现 Executor，仅仅 Mesos 默认的 Executor 就能够实现这个功能。如果你需要在 Executor 里面做很多自己定制化的工作，则需要自己写 Executor。

框架执行器（Framework Executor）的编写必须从 Executor 类继承, Executor 类提供了若干回调函数，在自定义执行器中需要覆盖。执行器第一次成功链接到 Mesos 时调用 registered 函数；节点重启后再次注册执行器时调用 reregistered 函数；执行器连接中断时调用 disconnected 函数；执行器启动任务时调用 launchTask 函数；调度器内正在运行的任务要终止时调用 killTask 函数；计算框架要传递给执行器信息时调用 frameworkMessage 函数，不要指望计算框架的信息以任何其他的可靠的方式重新传输；执行器需要终止所有现在运行任务时调用 shutdown 函数；发生致命性错误时调用 error 函数。

其中 launchTask 是最重要的函数，任务必须属于线程、进程、或者简单的计算，否则直到执行器返回回调时，该函数不会被调用。在 OpenCluster 的 Executor 的 launchTask 函数里，使用 Python 的 multiprocessing 线程池异步执行任务，任务执行过程中，发生错误或

[3] http://mesos.apache.org/documentation/latest/app-framework-development-guide/

执行器状态更新，使用 MesosExecutorDriver.sendStatusUpdate 方法发送状态更新，在调度器的 statusUpdate 函数里可以处理这些状态更新。

6.3.3 单任务单框架资源调度

在 OpenCluster 的单任务单框架资源调度模式中，每个 Manager 都是作为一个框架运行在 Mesos 的集群上，接收 Mesos 的资源，调度计算任务。如下图 6-2 所示。

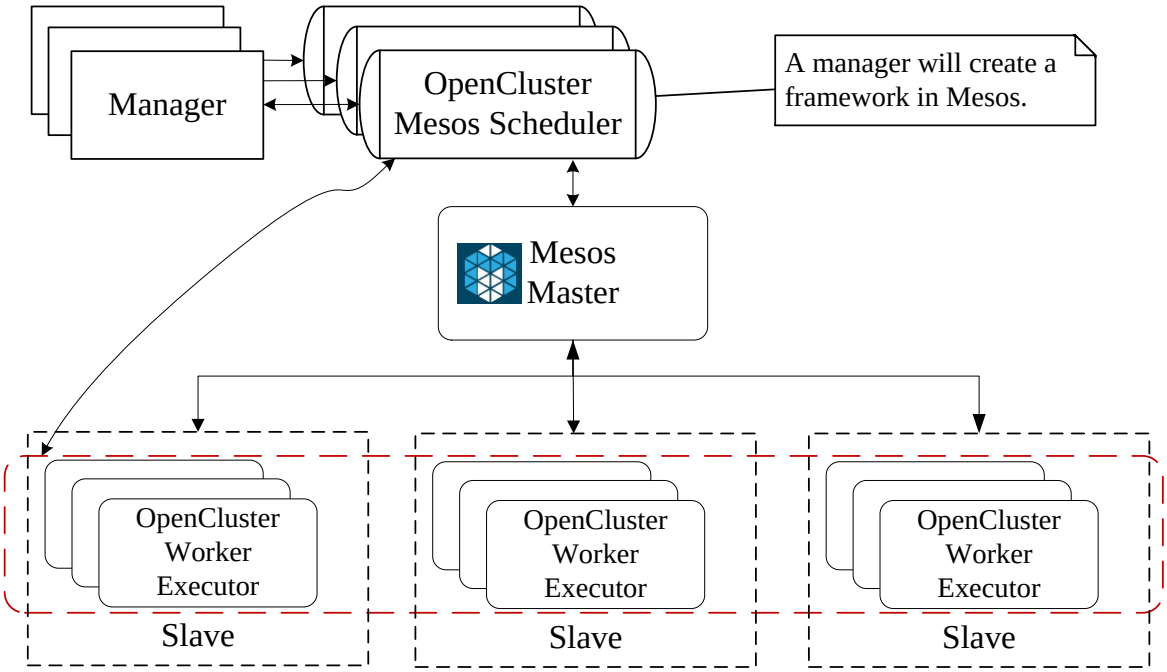

图 6-2 基于 Mesos 的多 Manager 的资源调度

在这个模型中，每个 Manager 都是 Framework，Mesos Master 任务的调度首先从 Slave 节点收集有关可用资源的信息，然后以资源邀约的形式，将这些资源提供给注册其上的 Framework（OpenCluster 里的 Manager），Framework 可以根据是否符合任务对资源的约束，选择接受或拒绝资源邀约。一旦资源邀约被接受，Framework 将与 Master 协作调度任务，并在数据中心的相应 Slave 节点上运行任务。资源分配模块确定 Framework 接受资源邀约的顺序，与此同时，确保在本性贪婪的 Framework 之间公平地共享资源。Mesos 资源分配的原则是在资源利用最大化的条件下公平地共享资源，但 Manager 的计算是有优先级的需求，比如在 MUSER 中，实时数据处理的 Manager 比历史数据处理的 Manager 具有更高的优先级，在单任务单框架的模式中，Framework 之间相对独立，彼此并不知道对方的存在，因此不能使用统一的策略将具有高优先级的任务优先调度。为此，我们增加另外一种基于 Mesos 的集中仓储式的资源调度模式。

6.3.4 集中仓储式资源调度

基于 Mesos 的集中仓储式的资源调度中，所有的 Manager 提交任务后，系统将任务序

列化后存入仓库（Kafka[115]）中，任务按优先级不同，分别存入不同的 Kafka 主题。OpenCluster 的调度器（Scheduler）负责从任务仓库（Kafka）订阅任务，调度任务，任务执行成功后，同样将结果存入仓库中。如下图 6-3 所示。

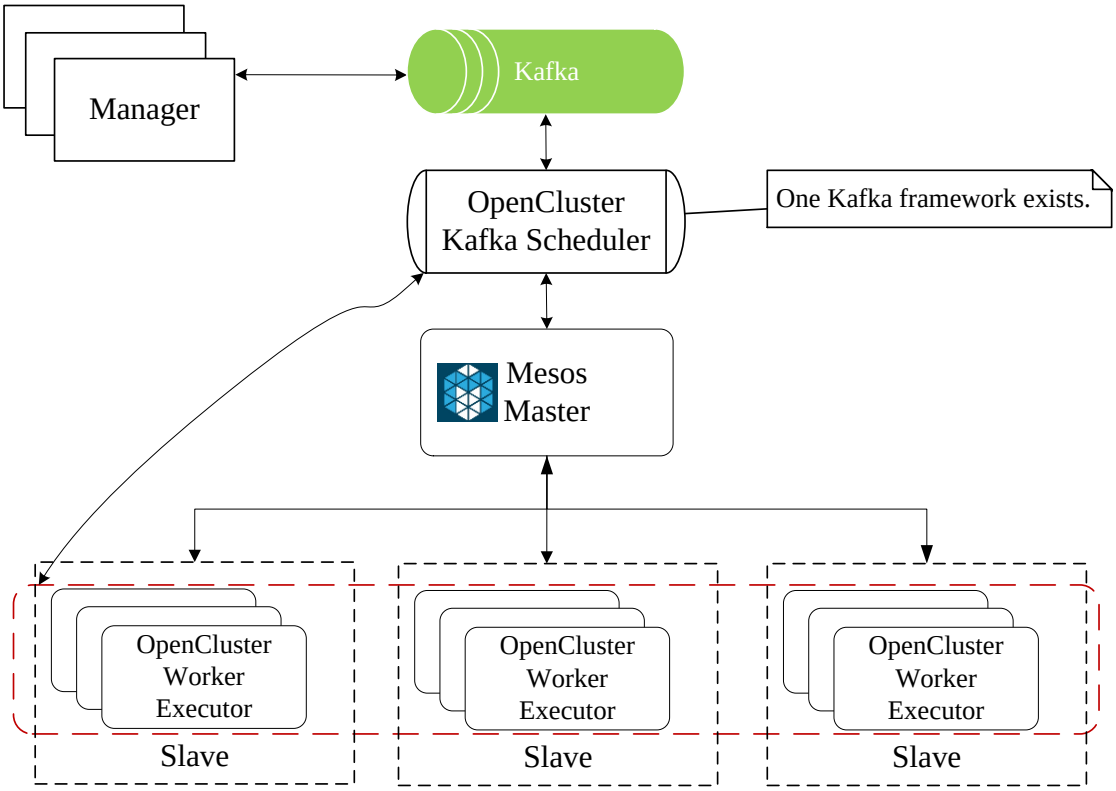

图 6-3 基于 Mesos 的集中仓储式资源调度

在 Manager 构建时，指定优先级（1-10，从低到高），并设置调度模式为集中仓储模式（Factory）时，在 Manager 启动是用户定义任务切分规则，将的任务切分为小的任务集，接着调用基类中 schedule 函数后，系统按照 Manager 的优先级将任务集提交到 Kafka 相应的主题中（OpenCluster-P，P 为 1~10，对应于 Manager 的优先级）。

集中仓储式的资源调度中，只存在一个调度器 OpenClusterKafkaScheduler，这个调度器是一个典型的长期运行的服务，从不同的优先级主题中（OpenCluster-P，P 为 1~10）订阅任务，消费主题中的任务，并存入缓存的队列中。当获得资源提供时，优先从优先级高的队列中读取任务，加载任务。

获得任务更新信息后，提取处理结果，将执行结果存入 Kafka 的结果主题中，执行结果主题的定义和任务主题相对应，为 OpenCluster-Result-P，P 为 1~10。后续的结果处理就可以从结果主题中订阅这些主题，做下一步的处理。

6.4 本章小结

本章介绍了 OpenCluster 支持的五种调度模式，其中本地和多线程模式，主要是用来测试，不涉及资源的调度。集群独立调度模式使用 OpenCluster 自身的资源调度，属于粗放

型调度，在节点和任务规模不大时，能提供稳定的运行支持。单任务单框架资源调度模式，为每一个 Manager 创建一个框架，OpenCluster 能和其他的计算平台共享集群的计算资源，但没有优先级机制控制多个 Manager 的资源分配。

采用集中仓储式的资源调度模式，显然能根据任务的优先级调度任务，优化集群资源的分配，提高应用业务的处理效率。但依赖于第三方存储（使用 Kafka）；优先级不能动态调整；Kafka 中任务和结果主题，需要事先创建；受限于 Kafka 消息大小对性能的影响，任务和结果中不能包含大容量的数据。Mesos 只运行 Linux 操作系统，在 OpenCluster 设计时，Mesos 使用的版本是 0.25，在 Mesos 1.0 及以上版本中，Mesos 提供了更多的特性，如在 CNI 支持，统一容器化工具，新的 HTTP API 支持，GPU 资源管理机制和 Windows 的支持也开始进入测试阶段。未来将继续以天文项目组分布式计算中的集群资源调度需求为基础，研究基于 Mesos 的资源调度，研究资源分配策略，GPU 资源调度和隔离等关键技术。

第7章 轻量级天文私有云应用

7.1 引言

分布式计算具有协同多台计算节点分布计算的能力，已在天文数据处理的被广泛使用，而在一个天文项目组内计算集群中，部署着多个计算框架以应对不同的计算需求，如何对不同的处理系统进行资源隔离，自动扩容，提高处理系统的稳定性和计算集群的使用效率，已成为内部集群管理的迫切需求。

Docker 是一个新兴的轻量级虚拟化技术，也是目前使用最多的容器化软件，已成为容器技术事实上的标准。Docker 最初只能在单宿主机上运行，不支持跨主机的容器部署、调度和管理。因为容器技术本身更适于解决大规模应用场景，所以通常都是集群基础上的部署、运维，但是目前对这一系列任务的自动化处理尚无统一的或者标准的框架。如果要让 Docker 真正在实际环境中发挥最大的效能并且易于维护，就需要有很成熟稳定的资源编排、资源调度和部署的支持，但是这方面暂时还没有很明显的最佳解决方案。当前主流的容器集群管理技术，包括了 Docker 官方的 Docker Swarm、Mesos 和 Google 的 Kubernetes。

本章在上四，五，六章研究分布式计算的基础上进一步研究分布式计算在私有云环境下的应用、开发、部署和运维。采用基于 Docker 的 CaaS（Container as a Service）层为上层提供云服务，以提高物理资源的利用率，以及业务部署和交付的效率，并促进应用架构的拆分和微服务化使用 Docker 提供资源隔离和内部依赖，本章的内容主要是结合资源调度框架 Mesos 和容器调度框架 Kubernetes 将 Docker 容器在数据中心自动伸缩，构建天文数据处理轻量级私有云。主要讨论了 Docker 镜像构建，容器编排与调度的关键技术，并给出了基于 Docker 的轻量级云在 MUSER 射电分布式数据处理中的实际应用。

7.2 天文计算容器化

通过 Docker 的使用，给 MUSER 的数据处理开发带来了业务的灵活性和稳定性。MUSER 中的数据处理应用可分为两类，一类是作业型的计算任务，任务处理完，应用退出。另一类是长服务型业务，如 Web 的监控服务，实时计算任务，这类应用需要一直在后台运行提供服务，永不退出。对于作业型任务，使用 OpenCluster 进行任务调度，在第 6 章中已对基于 Mesos 的资源调度做了讨论，为了提高业务隔离的安全性，OpenCluster 也支持使用 Docker 方式进行任务调度。本节主要介绍在 MUSER 中容器使用的关键技术，在 8.3

节和 8.4 节将介绍 MUSER 中容器化长服务型应用在集群环境下的调度。

7.2.1 OpenCluster 的容器支持

OpenCluster 实现的基于 Mesos 的细粒度调度模式中，提交任务时，支持多种调度模式（使用参数--mode 指定），当调度模式为 mesos 时，同时可以指定 Docker 的镜像文件，并设置执行器（Executor 为 Docker 的执行器），在这种模式下，分配到 Mesos 工作节点（Slave 节点）的任务，将以 Docker 容器的方式运行，这也要求所有的 Mesos 工作节点上都要支持 Docker。在 Mesos 工作节点上需修改配置文件 mesos-slave-env.sh（默认位置为 /usr/local/etc/mesos 目录），增加如下配置：

export MESOS_containerizers=docker,mesos

这个配置决定了 Mesos 可以直接支持 Docker。如果没有启用这个配置的话需要在每个 Slave 上修改这个配置，并在重新启动 Mesos 集群才能生效。

7.2.2 开发环境的容器编排

Docker 重新定义了程序开发测试、交付和部署过程。开发人员可以打包应用及依赖包到一个可移植的容器中，然后发布在宿主机上。一个应用包含了多个容器，比如 Web 容器依赖 Mysql 的数据库容器和卷容器，在开发环境下，多次停止启动多个容器，当容器之间还存在依赖关系时，启动容器时，还需按照先后顺序，使用--link，--volumes-from 来关联容器，这是非常繁琐并容易出错的工作。

Docker Compose 就是解决开发过程中面向单机的容器关系的问题，可以在一个文件中定义一个多容器的应用，然后使用一条命令来启动你的应用，然后所有相关的操作都会被自动完成。表格 7-1 列出了 MUSER 的 Web 应用的 Docker Compose 的配置文件，配置文件的详细指令，可以参考官方文档[4]。

表格 7-1 MUSER 的 Web 应用的 docker-compose.yaml 文件配置

docker-compose.yaml

```
muserweb:
    restart: always
    image: cnlab/muser:v1
    container_name: muser-web
    privileged: true
    ports:
        - 80:80
    links:
        - mysql
        - redis
    volumes:
        - /home/wsl/work/muser/web:/var/web
```

[4] https://docs.docker.com/compose/compose-file

```
    environment:
        WEB_REDIS_HOST: redis
        WEB_MYSQL_HOST: mysql
mysql:
    restart: always
    image: cnlab/mysql
    container_name: mysql-muser-web
    expose:
        - 27017
    volumes:
        - /opt/data/mysql:/data
redis:
    restart: always
    image: redis
    container_name: redis-muser-web
    expose:
        - 6379
    volumes:
        - /opt/data/redis:/data
```

上面的 YAML 文件定义了三个容器应用，第一个容器运行 Web 应用，第二个容器是 MySQL 数据库，第三个容器是 Redis 内存数据库。links 指令用来定义依赖，意思是 Web 应用依赖于 Mysql 和 Redis 容器。

配置文件定义完成后，在 docker-compose.yaml 文件的当前目录下，使用 docker-compose up 启动应用，或使用 docker-compose stop 停止应用。

7.2.3 镜像制作

容器镜像在某种程度上可被视作为轻量级虚机镜像。容器的封装性保证了开发和运维一致性，能够消除开发环境和生产环境的差异。开发人员使用镜像进行标准开发环境的构建，使用同样的镜像文件可以直接部署在生产环境中。MUSER 数据处理包含了众多第三方依赖库，通过 Docker 提供的 Dockerfile 方式，基础镜像选择的是官方镜像库中 centos 镜像。在 Dockerfile 文件中尽量地少使用 RUN 命令，以减少镜像层数，同时可以控制镜像文件的体积。为了有效地利用缓存，将操作相同部分放在 Dockerfile 的前面，将不同的部分放在后面。

Docker 镜像都来自官方的镜像源，由于国内连接 Docker 官方源的网速度太慢，而且对于 MUSER 这样的私有的镜像也不适宜直接放到 Docker 官方源上，因此我们搭建一个内部 Docker 镜像注册中心（Docker Registry）。将容器镜像上传到注册中心，团队开发人员可以从注册中心来拉取镜像，加速镜像文件的下载速度。

7.3 Mesos 中的容器调度

在第 6 章中已对 Mesos 的资源调度作了详细介绍，Mesos 的多个子任务在一个执行器上，可使用容器隔离技术，将不同的任务运行在不同的容器中，限制任务的资源使用。Mesos

除了提供自身的容器隔离，还支持 Docker 的容器隔离技术，可通过参数--containerizers=mesos,docker 在 Mesos Slaves 上配置。因此，使用 Mesos，可实现 Docker 容器在 Mesos 集群上调度，通常结合 Marathon 实现长服务的容器调度。

7.3.1 Marathon

Marathon 是作为一个 Mesos 框架运行在集群环境中，能够支持长服务的运行[116]，比如 Web 应用和后台调度等。用户通过 Marathon 的 Web 界面，或者其提供的 REST API 在 Mesos 管理的集群中调度 Docker 容器。可使用 JSON 格式描述任务配置，如下表格 7-2 展示了 MUSER 实时任务调度应用的 Marathon 的任务配置。完整的 JSON 格式及 REST API 说明可以参考官方网站[5]。

表格 7-2 MUSER 实时任务调度应用的 Marathon 任务配置

```
{
    "id": "/muser/realtime",
    "cpus": 2,
    "mem": 256.0,
    "instances": 1,
    "executor": "",
    "container": {
        "type": "DOCKER",
        "docker": {
            "image": "cnlab/muser:v1",
            "network": "HOST",
            "privileged": true,
            "parameters": [
                { "key": "user", "value": "muser" }
            ]
        },
        "volumes": [
             {
               "containerPath": "/work/opencluster",
               "hostPath": "/work/opencluster",
               "mode": "RW"
             },
        ]
    },
    "env": {
        "APP": "RealTime"
    }
}
```

在任务描述中定义了应用的 id，应用所需的 CPU 和内存等资源，实例数量，及 Docker 的镜像和相关参数。当新的应用提交到 Marathon 后，Marathon 首先向 Mesos Master 发送一个资源请求，并等待接收一个合适的资源 offer，一旦资源 offer 满足任务的要求，资源就会被接收，接着 Mesos Master 就将任务描述发送给提供资源的 Slave，Slave 启动容器运行，这个过程如下图 7-1 所示。

[5] https://mesosphere.github.io/marathon/docs/rest-api.html

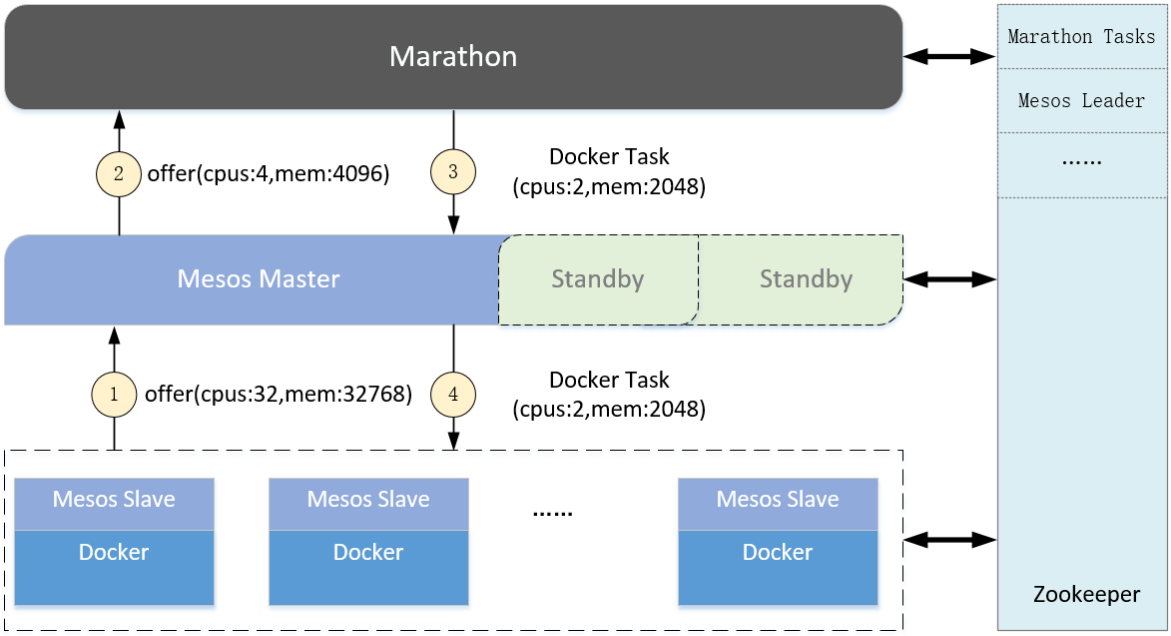

图 7-1 基于 Mesos 的 Marathon 容器调度结构图

MUSER 数据处理中需要长时间运行的服务如下图 7-2 所示，包括，参数配置及任务提交 Web 应用（/muser/confweb/），积分任务后台调度应用（/muser/integration），实时任务调度应用（/muser/realtime）和分布式计算框架 OpenCluster 的集群监控 Web 应用（/opencluster/web）。

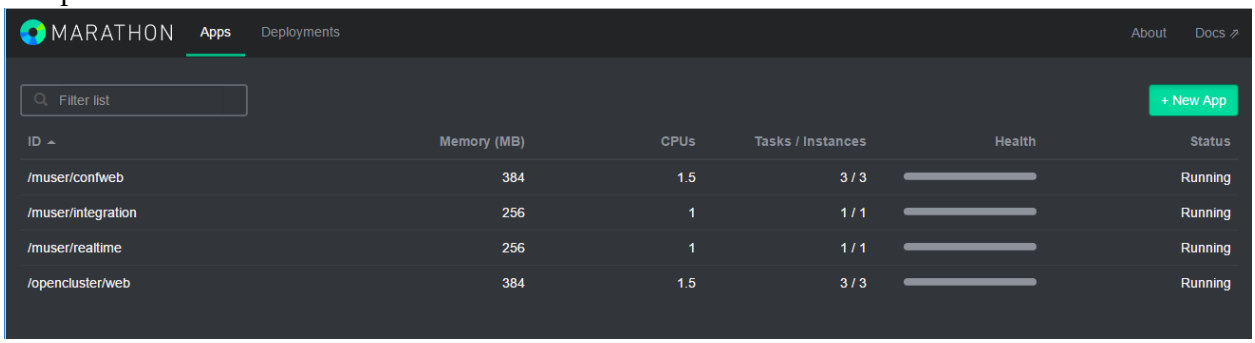

图 7-2 Marathon 在 MUSER 中的应用

7.3.2 服务发现

Marathon 提供的高可靠性运行能力，保障应用持续运行，也就说当一个节点不能提供服务时，Marathon 会将任务调度到集群中的其他节点上。积分任务和实时任务调度应用是无状态的，只启动一个实例，无论运行在哪台机器上都不影响，而对于 Web 应用来说，是需要对外提供服务的，Web 应用被调度到其他节点，IP 自然会发生变化，动态的 IP 显然不能满足要求。

为了解决这个问题，可以使用 Mesos-DNS 和 Marathon-lb。Mesos-DNS 会周期性从 Mesos master 获得所有运行任务的状态，并为每一个任务生成一个 DNS 的 SRV 记录。Mesos-DNS 的特点是轻量、无状态，易于部署和维护。另外一种方式是 Marathon-lb，

Marathon-lb 是一个容器化的应用，既是一个服务发现工具，也是负载均衡工具，它集成了 HAProxy，通过 Marathon 的 REST API 接口自动获取 Marathon 中应用的信息，并更新 HAProxy 配置，通过 servicePort 提供服务。

7.4 Kubernetes 容器管理

在容器云的生态圈已有多种容器编排和调度工具，Kubernetes 作为 Docker 生态圈中重要一员，是 Google 多年大规模容器管理技术 Borg 的开源版本，是 Google 在构建和管理大规模容器实践经验的最佳表现，已成为容器管理领域的领导者[62]。本节将讨论我们在 Kubernetes 应用过程中几个关键问题。

7.4.1 网络配置

Docker 原生的网络是桥（Bridge）的方式，Docker 启动时，会在宿主机器上创建一个名为 docker0 的虚拟网络接口，相当于虚拟交换机，每一个容器会在该 Bridge 上挂接一个端口，并与容器网卡联通。容器与主机相互通信，一个宿主机上的容器之前也可以互相通信。但不同宿主机上的容器之间是无法直接通信的。Kubernetes 的官方网站提供了几种方案，如 L2 网络，Flannel，OpenVSwitch 等来解决跨机器的容器互连[117]。本节讨论的是使用 Flannel 来搭建 VxLan，即叠加的隧道网络，选择 Flannel，是因为 Flannel 安装操作简单，同时依赖于 Etcd，而 Etcd 也是 Kubernetes 集群运行的必备组件。

Flannel 是 CoreOS 团队针对 Kubernetes 设计的一个网络规划服务，能够让集群中的不同节点主机创建的 Docker 容器都具有全集群唯一的虚拟 IP 地址，并使 Docker 容器可以互连。Flannel 默认使用 Etcd 作为网络配置的存储，以便每台主机知道整个集群的网络情况。安装 Flannel 后，会修改 Docker Daemon 启动时的网络参数，使用 Flannel 的提供的 IP 的范围。通过 Etcd，每个主机使用不同网段，单个主机上的容器获得的 IP 都在这个网段内，底层是通过 UDP/VxLan 等进行报文的封装和转发。这样，所有的容器都处在一个扁平化的网络地址空间中，所有容器直接的通信无需经过 NAT，所有集群节点与容器，容器与容器都可以直接通信。

7.4.2 服务发现

Services 是 Kubernetes 的真实应用服务的抽象，每一个服务后面都有很多对应的容器来支持，通过节点的 kube-proxy 进程和服务 selector 决定服务将请求传递给后端提供服务的容器。对每一个 Service, 它在本地打开一个端口，到这个端口的任意连接都会代理到后端 Pod 集合中的一个 Pod IP 和端口。在创建了服务后，服务 Endpoint 模型会体现后端 Pod 的 IP 和端口列表，kube-proxy 就是从这个 endpoint 维护的列表中选择服务后端的。对外

表现为一个单一访问接口，外部不需要了解后端如何运行，这给扩展或维护后端带来很大的好处。

在一个集群内的服务，容器或集群内节点上发出访问的客户端我们称之为内部使用者。要把服务暴露给内部使用者，Kubernetes 可以使用环境变量和 DNS 这两种方式。其中环境变量是默认支持的，但是 Pod 必须在 Service 之后创建，而使用 DNS 则没有这个限制。我们在 MUSER 的容器云搭建过程中也使用的是 DNS 的方式，如下图 7-3 为 Kubernetes DNS 服务部署的 Pod 的详细信息，Kubernetes 的版本使用的是 1.2，在 1.2 版本中使用 DNS，需要部署 etcd，kube2sky，skyDNS 和 healthz，SkyDNS 和 Kube2Sky 分别部署在两个不同的容器中，由 SkyDNS 负责解析集群的 DNS 请求。在 Kubernetes1.3 版本中将 SkyDNS 和 Kube2Sky 整合到了一个程序中 KubeDNS，同时引入了 dnsmasq 容器，由它接受集群的 DNS 请求，目的就是利用 dnsmasq 的缓存模块，提高性能。

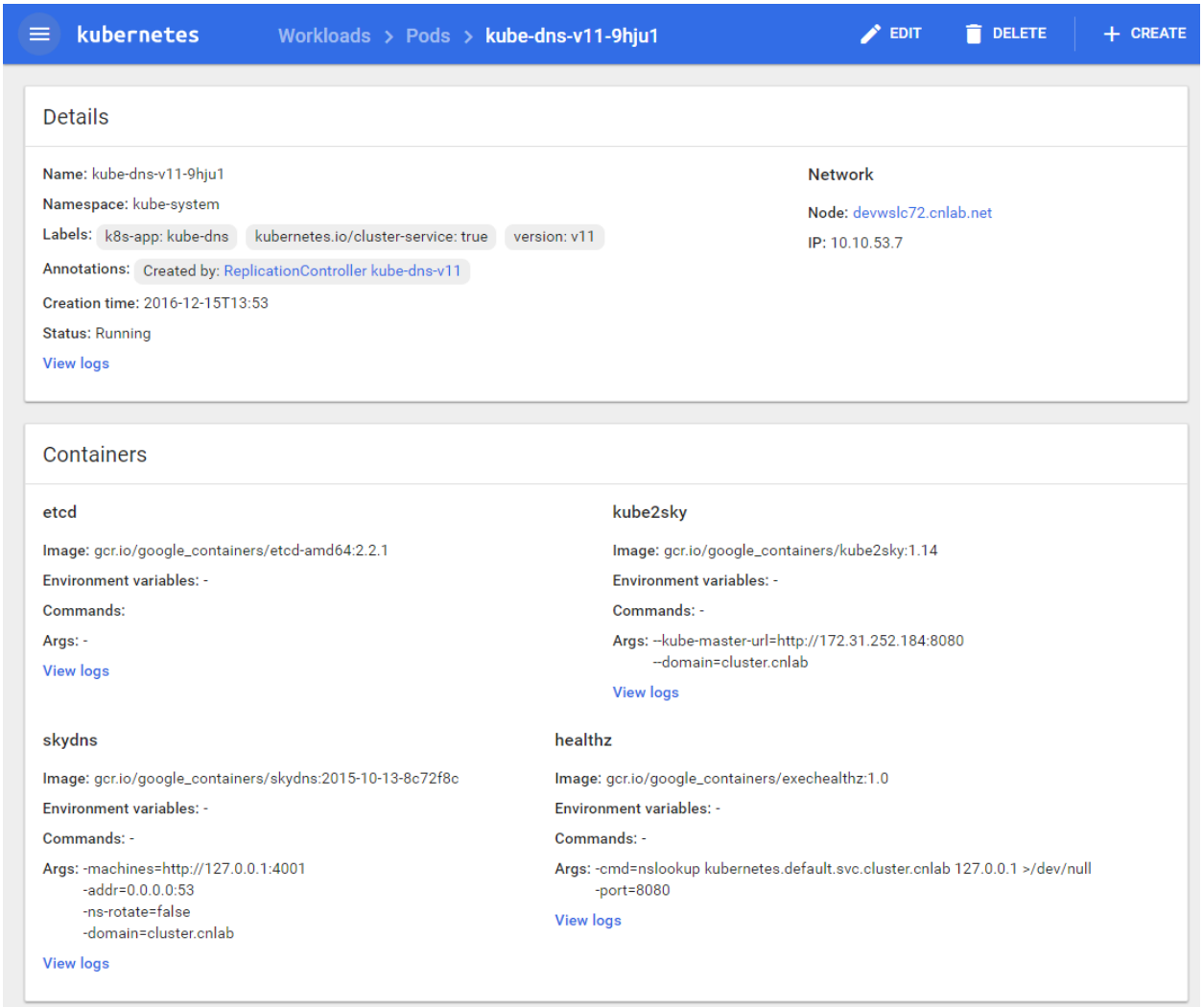

图 7-3 Kubernetes DNS 服务的 Pod 信息

对外的服务可以使用 HTTP 反向代理和负载据衡器（例如 NGINX），将外部请求代理到集群内的服务。动态抽取服务实例信息，激活一个实时 NGINX 配置文件的更新，实现

动态代理。由于使用的 Flannel 的网络模型，容器集群网络是封闭子网，可以提供平台内部应用之间基于四层和七层的调用，同时对外部使用反向代理提供应用基于域名（工作在七层）的直接访问，但无法满足用户在平台外部需要直接使用 IP 访问的需求。

7.4.3 应用部署

在 MUSER 中，容器只作为长服务类型的应用使用，而在 Kubernetes 中部署一个高可靠性的长服务应用主要涉及到 Pod，Service 和 Replication Controller 这三个对象。Pod 是 Kubernetes 中能够被创建、调度和管理的最小部署单位，一个 Pod 是由若干个 Docker 容器构成的容器组，容器组内的所有容器共享网络和存储。Replication Controller（RC）确保任何时候 Kubernetes 集群中有指定数量的 Pod 副本在运行，如果少于指定数量的 Pod 副本，RC 会启动新的 Pod，RC 是 Kubernetes 较早期的技术概念，适用于长服务的业务类型。未来对所有长服务业务的管理，都会通过 Deployment 来管理。Deployment 不仅可创建新的服务，还可以滚动升级服务。Service 是客户端需要访问的服务对象，每个 Service 会对应一个集群内部有效的虚拟 IP，集群内部通过虚拟 IP 访问一个服务。

以 MUSER 的参数配置及任务提交 Web 应用（/muser/confweb/）为例，来说明如何利用 Kubernetes 提供高可用性提交一个应用。Kubernetes 支持使用 JSON 和 YAML 两种格式部署各种资源，如下表格 7-3 使用 YAML 描述服务配置，表格 7-4 为复制控制器的配置。

表格 7-3 MUSER 参数配置及任务提交 Web 应用的服务配置

```
apiVersion: v1
kind: Service
metadata:
  name: muserweb
  namespace: muser
  labels:
    app: muserweb
spec:
  ports:
  - port: 31201
  selector:
    app: muserwebv1
```

表格 7-4 MUSER 参数配置及任务提交 Web 应用的复制控制器配置

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: muserweb
spec:
  replicas: 1
  selector:
    app: muserwebv1
  template:
    metadata:
      name: muserwebv1
      labels:
```

```
    app: muserwebv1
spec:
  containers:
  - name: muserweb
    image: cnlab/muser:v1
    ports:
    - containerPort: 9080
    args:
    -  --apiserver-host=http://172.31.252.184:8080
```

表格 7-4 中 spec（规范）描述了用户期望 Kubernetes 集群中分布式系统达到的期望状态（Desired State），如副本数（replicas）以及使用什么模板创建容器。

7.5 讨论

过去两年，容器和容器镜像已经成为了开发分布式和云计算应用中所必不可少的技术。在分布式集群中，容器的编排和调度则是无法回避的问题。Docker 在 1.12 的版本后，集成了之前独立的 Docker Swarm 编排引擎。Docker 将单节点 Docker 的使用概念扩展到 Swarm 集群中，操作命令几乎都是一致的，Docker 也正在迅速将添加更多的功能来与其他调度系统竞争。Google 的 Kubernetes 在 Google 经历了数年的实战检验，加上最近微软、RedHat、IBM 和 Docker 等也都宣布支持参与此项目，使得 Kubernetes 有着很好的前景。下一步将重点研究基于 Kubernetes 的容器调度在天文计算中的设计模式，包含了计算任务密集型任务、有状态的服务和跨区域等模式。

7.6 本章小结

本章研究了如何把容器和云原生计算等技术整合到传统的天文项目组的集群基础设施中，以得到一种新的服务环境，具体实践了在 Mesos 和 Kubernetes 平台下的容器编排和调度，并运用在 MUSER 的数据处理中，使用该技术能够简化开发、部署、运维的流程，同时能为天文数据服务提供高质量的服务保障。章节研究的容器技术和容器调度在 MUSER 数据处理中的应用，具有较强的实用价值，研究方法和结果可以为其他天文项目组的内部基于容器的云计算应用和虚拟天文台相关应用提供参考。

第8章 基于 ZeroMQ 的观测控制系统通信技术

一个望远镜系统的正常运转需要多种类型的设备和系统协同工作，如圆顶，CCD，光谱仪等。单一类型的设备也有多个，需要协调工作。在望远镜运行阶段，可能因为观测需要，增加新类型的设备或者新的制造厂商的设备，这些不同的设备通常都需要一个前置的计算机连接驱动，而完成一个观测计划就需要这些不同设备之间相互配合，协同工作。那么彼此之间的高效通信就变成能否完成有效观测的一个重要因素。

消息中间件技术为代表的新一代分布计算框架正快速发展，其中以 ZeroMQ、ActiveMQ 和 RabbitMQ 为代表。特别是 ZeroMQ，它扩展了传统 BSD socket 能力，提供简单的基于消息的通信，是一个非常简单的通信库[118]。ZeroMQ 专门为高吞吐量/低延迟的场景开发，非常适合天文望远镜观测控制这样的分布式、低延迟要求的场合。

本章主要研究基于 ZeroMQ 的天文望远镜控制中通信系统设计与实现。先分析和讨论开源天文望远镜控制系统 RTS2 中基于 Socket 网络通信的局限性，接着讨论了物联网通信协议 MQTT 在望远镜控制系统中适用性。最后给出了 ZeroMQ 的天文望远镜控制中通信系统设计与实现，并对给出了原型系统的通信效率的性能评估。

8.1 RTS2 的网络通信

RTS2 是开源领域中最著名的远程望远镜控制系统，是一套完整的自主天文台管理系统，包括设备控制、观测目标设置、观测任务调度、观测控制数据生成等功能[82]。RTS2 已经可以支持诸多的设备或模块，实现多种仪器设备如圆顶，望远镜，CCD，气象站的接入和控制。在网址 http://www.rts2.org/wiki/hw:start 中可以查看 RTS2 目前支持的设备。

8.1.1 体系结构

RTS2 体系中服务和设备是以组件的形式进行封装，对程序可扩展性就进行了较完善的考虑。所有的组件和服务都可以通过配置文件中进行配置，可以在任何时候开始工作、重启和停止，在这过程中不会影响系统其他模块的使用。RTS2 中设备、服务和客户端都作为独立组件运行，在操作系统层都是独立的进程，组件之间的网络通信使用基于 TCP 协议的原始套接字（Socket）方式[119]。RTS2 采用分层模块化设计，使用面向对象语言 C++编写，对象间具有清晰的继承关系，有利于程序的二次开发，对于未来新增的设备可以扩展相应的基类，快速的接入 RTS2 系统中。如下图 8-1 为 RTS2 的各个部分的组件组成了 RTS2

的体系结构。

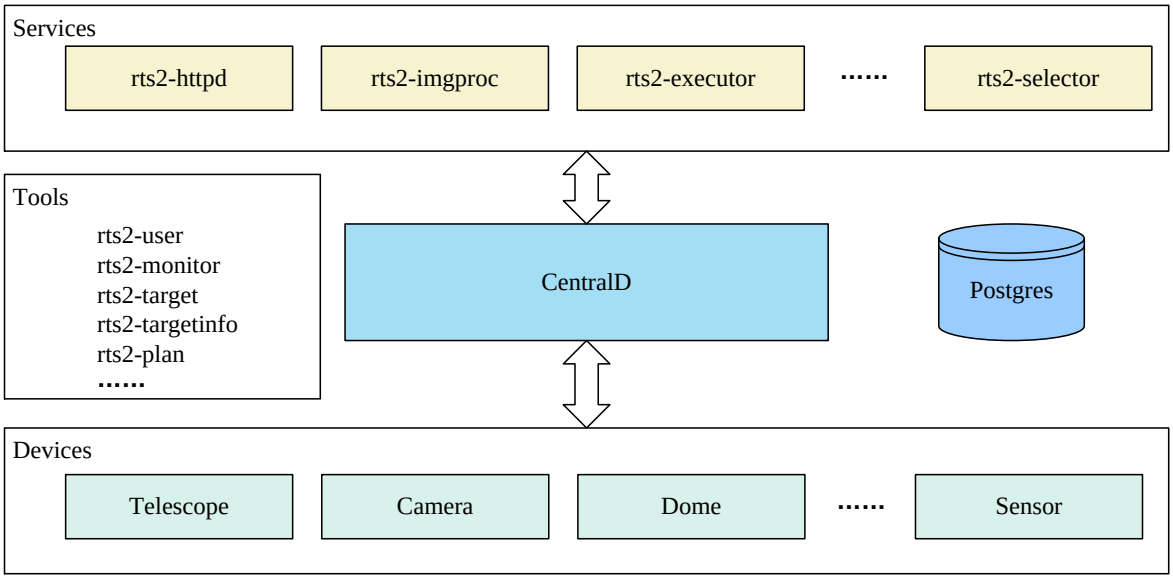

图 8-1 RTS2 体系结构

RTS2 系统中的组件主要分为中心控制程序（CentralD）、服务（Services）、设备（Devices）和工具类（Tools）的应用。 除此之外，RTS2 中的观测目标、观测计划的实施依赖于数据库，数据库服务器使用 Postgres，数据库中主要存储观测目标（target）、观测计划（plan）的详细信息等。CentralD 是 RTS2 的核心组件，在服务和设备、工具和设备之间起到桥梁的作用，CentralD 中保持着系统中所有设备的连接地址信息。RTS2 需要运行在 Linux 环境下，可进行分布式的部署，至少需要部署一个 CentralD。

RTS2 中的工具类应用，首先通过 CentralD 找到设备的地址连接信息，通过这个信息双方建立 Socket 连接进行通信，接收或发送相应的数据以此来控制设备。在这个过程中，CentralD 除了提供设备连接信息，不参与工具类和设备之间的通信。

8.1.2 网络协议

RTS2 中的设备类应用启动后，在一个动态的端口监听，并在 CentralD 中注册，注册成功后，CentralD 中就保存了设备的 IP 地址和端口信息。下面通过一个 MICCD 的启动过程来解释设备类的网络初始化过程。MICCD 启动的主函数（main 函数）是在 src/camd/miccd.cpp 文件中，这个文件也定义 MICCD 设备类。在 main 函数通过运行 run()函数启动 MICCD 程序，进行网络参数和硬件参数的设置，网络监听事件采用 select 的 I/O 多路复用的模式达到在一个线程内同时监视多个文件描述符的读写就绪状况。当有客户端命令连接到 MICCD 设备时，在 Device.cpp 中的 createConnection()函数执行产生一个 DevConnection 连接，在 block.cpp 中的 selectSuccess()函数调用该连接的 receive()函数接收客户端发送的数据，接着调用该连接的 processLine()函数进行解析（在 connection.cpp 文件中），processLine()函数中，对接收到的数据进行字符串匹配，进而解析出不同命令，再执

行相应的操作。在 processLine()中会调用 DevConnection 中的 command()，DevConnection 中的 command()调用 MICCD 的 commandAuthorized 函数，Camera::commandAuthorized()和 Device::commandAuthorized()函数。因此可以在 MICCD 的 commandAuthorized 函数中，对这类型设备的特殊命令编写相应的解析处理过程。

RTS2 网络通信使用基于 TCP 协议的纯文本的方式，数据传输时没有采用其他的序列化方法。因此，为了区分不同的操作命令，RTS2 使用不同的前缀代表不同的操作，如下表格 8-1 所示。

表格 8-1 RTS2 网络协议前缀描述

前缀	解释	示例
V	读取设备参数值	V infotime
X	修改设备参数值	X GAIN = 20
A	认证请求	A authorization_ok 2
S	设置设备状态	S 1 exposure started
P	当前状态进度	P 10 100
R	报告设备的状态及进度	R 1 10 100
B	读取阻塞状态	B 1 4
T	查看设备的响应状态	T ready
M	消息	M 1145788.876 C0 Exposure started
E	元数据	E value_type “name” description
F	列表形式的元数据	F “name” selection_vals
C	二进制通道	C connect_id 1 2 22876 32256
D	二进制通道数据	D connect_id ir1.osn.iaa.es 22876 32256
H	销毁二进制连接	H connect_id
I	共享内存的关键字	I connect_id 1 2 22876 32256
J	共享数据接收完	J connect_id
K	销毁共享内存段	K connect_id

除了上述基本的网络协议前缀，RTS2 中不同设备也有多种命令的解析处理过程，如 Telescope 类的 move 命令、Camera 类的 expose 命令解析操作。此外，还可以在设备的具体的类的定义的 commandAuthorized 函数中，编写自定义的命令，如 MICCD 中的 clear 命令的解析。命令完成后，返回以“+”或“-”字符开始,紧随其后的是 3 位数字的代码，接着空格分开的字符串信息，如“+000 OK”。“+”表示执行成功，“-”表示执行失败。

8.1.3 存在的问题

RTS2 使用 TCP 的原始套接字编程，每一个 TCP 连接都是由五个元素确定，源 IP 地址、源端口号、目的 IP 地址、目的端口号和通信协议。在实际程序的开发过程中一条连接建立之后，它可能需要在一段时间内一直被通信双方所保持，以备下次数据传输使用。在望远镜系统中，有多种设备，多种服务，通信的终端数目就有多个，同时也支持多客户端的连接，这就需要每个终端都要维持它所有的通信关系，使用 RTS2 就容易形成如下图 8-2 所示的网络连接关系。

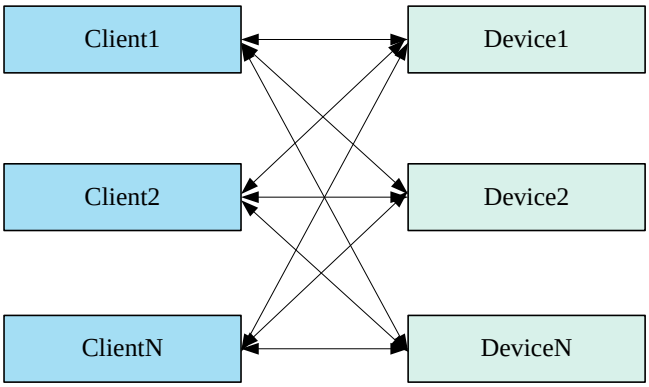

图 8-2 多对多通信

此时，每个参与通信的客户端所需维持的连接数量将非常庞大，这非常不利于程序的开发和实现。另外在 RTS2 中的连接类（Connection）的头文件和源文件的代码就超过 3000 行，显然不利于日后的升级和维护。

使用 select 的 I/O 多路复用，每次调用 select 函数，都需要把文件描述符集合从用户态拷贝到内核态，这个开销在文件描述符很多时会很大；同时每次调用 select 都需要在内核遍历传递进来的所有文件描述符，这个开销在文件描述符很多时也很大；select 支持的文件描述符数量有限，默认是 1024。

此外，望远镜控制系统中，经常需要同步操作多种设备。比如，多 CCD 的顺序观测，组合观测，同步曝光。RTS2 中一个客户端是同时建立和多个设备的独立的网络连接，这种情况下，只能通过循环的方式，依次向多个设备发送命令。尽管纯文本协议，协议设计小巧，同时通常控制系统中，终端不多，网络负载小，但仍不是严格意义上的广播数据通信方式，不能较好的解决协同控制问题。

8.2 基于 MQTT 通信分析

针对于 RTS2 使用的原始 TCP 套接字的局限，本节讨论物联网通信协议 MQTT（MessageQueuing Telemetry Transport，消息队列遥测传输）在望远镜控制系统通信中的适用性，主要是考虑到物联网设备的通信控制和望远镜控制系统有诸多的相似性。

1，设备控制，两者的核心都是对设备的控制，数据采集；

2，协同控制，都存在协同控制多种设备的需求；

3，可靠传输，低延迟，容错，高性能的传输是两者共同的需求。

8.2.1 MQTT 通信协议

MQTT 是 IBM 和合作伙伴开发的一个通讯协议，2014 年，正式成为推荐的物联网传输协议标准。它被设计用于轻量级的发布/订阅模式（Publish/Subscribe）的消息传输[120]，旨在为低带宽和不稳定的网络环境中的物联网设备提供可靠的网络服务。望远镜控制系统中的设备一般都需要一个前置机来驱动设备，而 MQTT 协议适用于低计算能力的设备，可显著降低对前置机的性能要求，节约控制成本。

相比较原始套接字，MQTT 也只引入了的 2 个字节头部传输开销，目前通用的协议很少有比 MQTT 还要低的传输开销，MQTT 协议固定头[6]格式如下图 8-3 所示。

Bit	7	6	5	4	3	2	1	0
Byte 1	Message Type				DUP Flag	Qos Level		RETAIN
Byte 2	Remaining Length							

图 8-3 MQTT 消息头格式

第一个字节（Byte 1）用于说明消息体的信息，其中 4-7 位表示消息类型，使用 4 位二进制可代表 16 种消息类型；DUP Flag 为打开标志；QoS(Quality of Service，服务质量)来保证消息的可靠性，它包括三种不同的服务质量（最多只传一次、最少被传一次、一次且只传一次）；RETAIN(保持)，只对 PUBLISH 消息有效，为 1 时表示发送的消息需要一直持久保存，为 0 时仅仅为当前订阅者推送此消息。

第二个字节（Byte 2）为剩余长度，保存变长头部和消息体的总大小的，但不是直接保存的。这一字节是可以扩展，其保存机制，前 7 位用于保存长度，后一部用做标识。当最后一位为 1 时，表示长度不足，需要使用二个字节继续保存。

从上面的消息头的分析可以看出使用 MQTT 作为望远镜控制系统中协议具有的优势：

1，可靠传输。设置高 QoS ，MQTT 可以保证消息可靠安全的传输。

2，消息推送。支持消息实时通知、灵活的 Publish/Subscribe 模式及消息存储和过滤。

3，低带宽、低耗能、低成本。占用带宽小，并且带宽利用率高，耗电量较少。

4，应用解耦。提供一对多的消息发布，解除应用程序间耦合。

8.2.2 基于 MQTT 的设计

MQTT 协议中有三种身份：发布者（Publisher）、代理（Broker）（服务器）、订阅者

[6] http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html#msg-format

（Subscriber）。其中，消息的发布者和订阅者都是客户端，消息代理是服务器，消息发布者可以同时是订阅者。因此使用 MQTT 作为望远镜控制系统的通信协议，系统的组件结构设计如下图 8-4 所示。

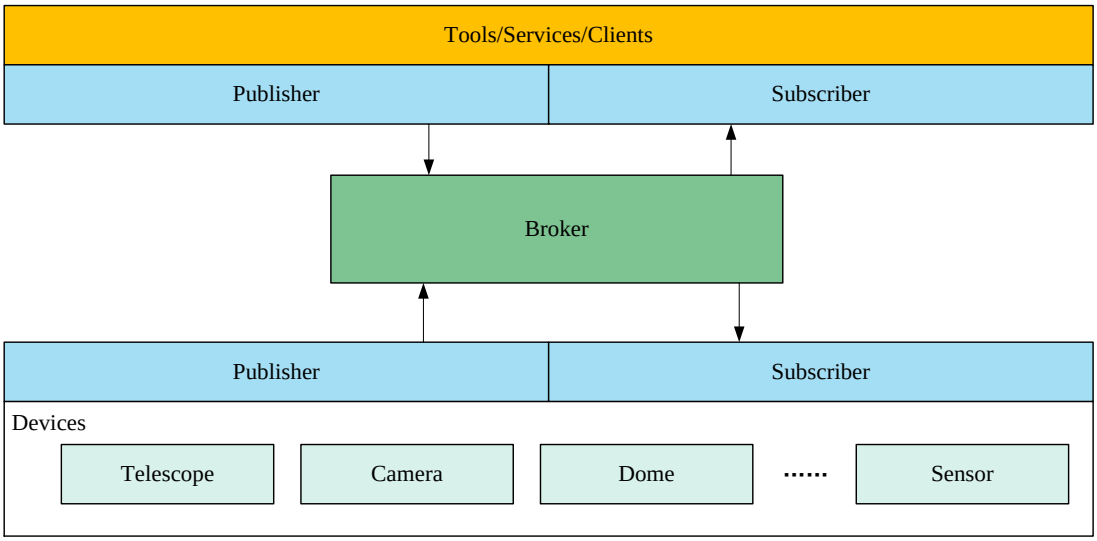

图 8-4 基于 MQTT 协议的望远镜控制系统组件结构图

其中 Tools/Services/Clients 以及 Devices 设备都是客户端，同时具有发布者和订阅者的身份，建立到 Broker 的持久的网络连接。客户端可以，发布其他客户端可能会订阅的信息；订阅其它客户端发布的消息；退订或删除应用程序的消息；断开与服务器连接。

MQTT 传输的消息分为主题（Topic）和负载（Payload）两部分，主题可以理解为消息的类型，订阅者订阅（Subscribe）后，就会收到该主题的消息内容（Payload），是指订阅者具体要使用的内容。基于 MQTT 的通信设计，最主要的也就是消息的设计，准确的说就是主题的设计。主题 Topic 通常是一个具有分层结构的字符串，可以基于有限数量的表达式进行过滤。基于 MQTT 的望远镜控制系统设备发布的主题定义如下表格 8-2，表格 8-3 定义的是客户端发布的主题。

表格 8-2 基于 MQTT 的望远镜控制系统设备发布的主题定义

主题	说明
device/info	设备信息
device/+/param	单个设备参数值
device/+/status	单个设备的状态更新
device/client/command/+/+	客户端命令执行的反馈
device/client/+/value/+/+	客户端修改设备参数值的反馈

表格 8-3 基于 MQTT 的望远镜控制系统客户端发布的主题定义

主题	说明
client/connect	客户端连接

client/disconnect	断开客户端连接
client/command/+	客户端向设备发送命令
client/device/+/value/+	客户端修改设备的参数值

8.2.3 存在的问题

1，需要安装第三方的 Broker，MQTT 有众多 Broker 的实现方式，可以参考[7]，其中 mosquitto 是一个开源基金会 eclipse 的轻量级的 C 开源实现，完全兼容了 MQTT 3.1 和 MQTT 3.1.1，有较完善的客户端编程接口，在验证中就是使用 mosquitto。功能完善、性能稳定的 Broker 是决定整个系统成功运行关键于因素。

2，尽管发布/订阅模式还提供了比传统的客户端/服务器模式更好的可扩展性，但要把发布/订阅模式扩展到支持数百万的连接，在技术上仍然是一个挑战。

3，异构连接的引入，比如有些设备需要使用串口连接，CCD 设备需要额外的网络连接来获取大尺寸的二进制图像，这些显然不适合使用 MQTT。如何更好的融入，协调不同协议的同步，也是需要关注的问题。

基于 MQTT 的望远镜控制系统的实现，仍在编程验证中，未来继续关注上述的问题解决，也将会继续完善消息主题的设计和对 Broker 性能及稳定性的验证。

8.3 基于 ZeroMQ 的设计

为了解决 NVST 的观测控制需求，设计和实现一个协同的自主望远镜观测控制系统，命名为 Collaborative Telescope System，CTS。目前处于设计和代码的编写阶段，基础框架已经搭建完毕，设备类终端内部控制通信及和观测控制系统的通信都是使用 ZeroMQ。

8.3.1 系统结构

CTS 的使用模块化设计（参考 ATST 设计），将望远镜系统控制系统分为：终端控制系统（TCS）,观测控制系统（OCS）和观测控制界面（OCS-UI），系统结构如下图 8-5 所示。

[7] https://github.com/mqtt/mqtt.github.io/wiki/servers

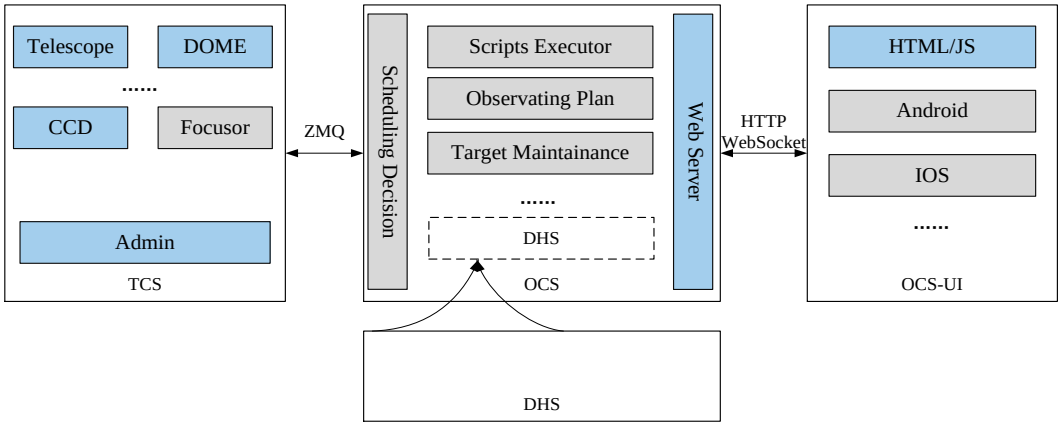

图 8-5 CTS 体系结构图

TCS 的核心任务是底层的设备控制，设备类的实现参考 RTS2 中对设备的定义，目的是快速理解设备参数的意义和支持的命令，以及各种设备之间的阻塞关系。编程语言采用 C++。TCS 中管理组件（Admin）和 RTS2 中的 CentralD 组件类似。

OCS 的重点在于如何实现数据的融合进而进行观测调度决策，同时对外提供 RESTful 和 Websocket 的 API，以便外部应用使用 OCS 的功能，使用 Python 语言编程。OCS 与 TCS 之间的通信采用 ZeroMQ，OCS 以客户端的方式连接到 TCS 的管理组件。同时考虑为 DHS 预留接口，之所以把 DHS 作为 OCS 的一部分，是因为高速实时数据处理的结果将能支持观测调度的决策，选择最优的观测目标。

UI 部分通过 OCS 提供的标准接口构建控制界面，目前仅构建 HTML 的界面，采用 React 的编程框架，目的是构建跨平台的页面，能够容易地移植到移动应用中。

8.3.2 套接字设计

分布式观测控制系统的核心是各个终端设备之间的数据通信，TCS 中的组件包括设备（Device）和管理组件（Admin），组件之间存在点到点、点到多点、多点对应多点的数据事件驱动的控制与处理需求。同时需要考虑终端实时状态、异常恢复、统一时序控制及可用性探测。客户端（Client）需要能够操作设备，接收设备广播的更新信息，但客户端不是直接连接到设备上，而是通过管理组件，因此管理组件需要为客户端创建一个接收请求命令的套接字和广播信息的套件字。对于设备来说，也需要接收来自客户端的命令，以及客户端的广播信息，管理组件也需要为设备创建一个接收请求命令的套接字和广播信息的套件字。如下图 8-6 所示为 TCS 内部的 ZeroMQ 通信套接字设计。

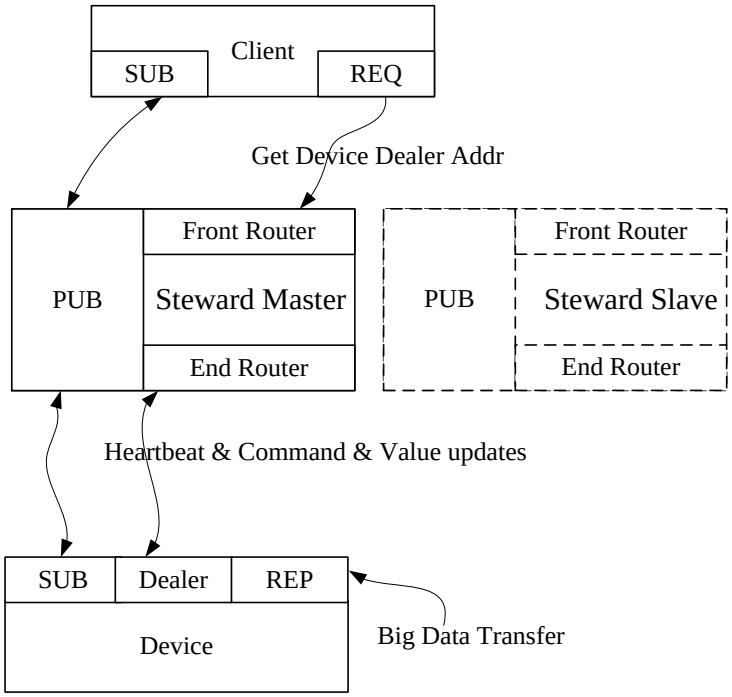

图 8-6 TCS 基于 ZeroMQ 的通信套接字设计

客户端有两个套接字分别为 SUB 和 Dealer 类型，命名为 client_sub 和 client_dealer。管理组件中的套接字包含了一个 PUB 和两个 Router 的类型，分别命名为 admin_pub、front_router 和 end_router。设备端中有三套接字，为 SUB、Dealer 和 REP 类型，分别命名为 device_sub、device_dealer 和 device_rep。

管理组件启动时需要 admin_pub、front_router 和 end_router 这 3 个 TCP 端口，admin 作为广播接口，将设备的更新信息，及广播信息进行广播。客户端启动时首先连接到 front_router，获得 admin_pub 的连接信息后，client_sub 再连接 admin_pub，接收广播信息。通过 front_router 接口，客户端向管理组件发送心跳，向特定的设备发送命令。

设备连接到 end_router，上报设备参数值，定时发送心跳，获得管理组件的 PUB 地址和端口，并连接到 PUB 端。对单个设备的操作，客户端连接到管理组件，查到设备的地址信息，进而发送操作命令，对于多个设备的并发控制，可以通过管理组件的 admin_pub 接口广播信息。

管理组件采用主从热备，客户端和设备在失去和管理组件主节点的心跳连接时，转向从节点，任一时间，某个节点会充当主机，接收所有客户端的请求，另一个则作为一种备机存在。主节点和从节点之间通过心跳来维持整个系统的高可用性，避免在管理组件上的单点故障。

在 ZeroMQ 中，套接字默认是瞬时的，它所连接的套接字（如 Router）则会给它生成一个 UUID 来标识消息的来源，与之相关联。Router 套接字会在所有收到的消息前添加消息来源的地址，为了客户端能操作特定的设备，就需要一个固定标识。每个设备都有一个固定的名称，在启动时可以通过参数指定，这个名称也作为设备的 Router 套接字的标识。

8.3.3 心跳机制

设备的可用性检查望远镜控制系统中的关键一环。可用性检测是通过固定时间的检测，判断各终端的可用性，自动移除不能工作的设备，在分布式系统中叫作心跳机制。使用ZeroMQ，在客户端、管理组件、设备中都需要读取多个套接字，主循环中都使用了 zmq_poll() 非阻塞的读取方式，使用另一个计时器来触发心跳。不使用主循环来控制心跳的发送，主要是因为这回导致过量地发送心跳（阻塞网络），或是发送得太少（导致节点断开，产生虚假故障）。心跳只在管理组件和设备之间维持，异步的在节点之间传递，任一节点都可以通过它来判断对方已经死亡，并中止通信。心跳频率是可以在文件中或启动参数中配置的，并在各个节点上维持一致的频率。

8.3.4 驱动转接

当前 NVST 的控制程序和数据采集程序使用 Windows 平台，而 CTS 在设计阶段考虑到通用性，选择在 Linux 平台下开发，因此需要对现有的 Windows 驱动做封装。好的一点在于，ZeroMQ 同样提供了在 Windows 平台下的开发库，因此同样可开发 Windows 下的虚拟设备类，最终接入到 CTS 中。未来我们也会考虑编写 Windows 下的设备基类，进一步减少自定义设备的代码工作量，实现 CTS 的跨平台操作。

8.3.5 序列化

在编写网络应用程序的时候往往需要将程序的某些数据存储在内存中，然后将其传输到网络中的另一台计算机上以实现通讯。这个将程序数据转化成能被存储并传输的格式的过程被称为“序列化”（Serialization），而它的逆过程则可被称为“反序列化”（Deserialization）。ZeroMQ仅解决网络通路的问题，并不提供序列化的方法，我们可以将对象强制转换为char*或者 void*类型的数据，然后进行数据的传输，那么对于每一个类的对象，都要编写不同的代码，工作量很大，通用性不高，同时还注意 CPU 字节序的问题。因此需要选择合适的序列化库，同时能最小化侵入编程。我们使用 MsgPack，MsgPack 是一个基于二进制的对象序列化库，具有跨语言的特性，同样非常容易使用。操作命令和更新消息通过 MsgPack 进行压包，然后写入 ZeroMQ 的消息结构体（zmq::message_t），通过 ZeroMQ 传递，最后接收者利用 MsgPack 进行解包，从而分析命令，获得数据。

8.4 本章小结

本文对 RTS2 的网络通信做了较为全面的研究，RTS2 采用的 Socket 方式，尽管理论上具有最好的性能，但实际上具有较大的开发难度和较差的维护性。RTS2 内部对多个设备的操作使用轮询的方式，实现数据同步广播的难度较大。因此，调研了在物联网领域广泛

使用 MQTT 协议，依据物联网和望远镜系统中设备控制和数据采集的相似性，分析验证了 MQTT 在望远镜系统中控制的可行性。MQTT 的方式，依赖于消息中心的稳定性，存在消息中心的单点故障的问题。最后本文采用了 ZeroMQ 这一新型的网络通信库，提出了一种基于消息中间件的消息通信架构，满足了望远镜控制系统中的请求/响应、广播等多种通信模式；采用了序列化方法，提高了通信系统的消息传递能力、系统的稳定性和数据的安全性。使用该消息通信架构的分布式网络具有智能化、适应不同环境要求、可靠性好、效率高和易于扩展等特点，可满足多种不同环境的控制任务要求。

第9章 总结与展望

本文首先从分布式数据处理系统的研究和分析开始，讨论了一些相对成熟、稳定的系统在 MUSER 射电数据处理中的可行性应用，通过对 Spark 在 MUSER 中具体应用研究表明 Spark 基于内存迭代的分布式数据处理框架具有很好的数据处理性能和扩展优势，但由于此类开源框架的编程复杂性和较复杂的维护性，造成了其直接应用的困境。另外，这些框架通常使用 Java 或 Scala 语言编写，而不是在天文中广泛使用的 Python，尽管提供了 Python 的接口，但不能完整地发挥系统性能的优势。Hadoop 或 Spark 等此类框架比较适合用于互联网中大数据处理与分析，天文的数据处理对象和互联网的数据有较大的不同，同时天文中已有的数据处理算法移植到基于 Map/Reduce 的计算模型中也有较大的困难。

结合天文数据的特点和天文学家的使用习惯，同时满足易用，高性能和可扩展性，本文有针对性的设计了一个分布式计算框架——OpenCluster。该系统是基于类似传统工业制造流程的思想，使用 Python 编写，提供简单易用的 API，能够容易地集成已有的数据处理代码，编写分布式处理程序。结合 Mesos 资源调度，提供多种资源调度模型，并应用在 MUSER 实际的数据处理系统中。同时也将流行的容器技术应用在 MUSER 项目中。

观测控制系统一直是望远镜系统中的一个重要组件。望远镜系统需要实现异构设备的统一集成与调度，数据处理的高效与有序，快捷的观测模式改变与设备的切换，高效的自动化观测调度，这一切都需要各个组件相互配合，因此也就需要网络通信和消息传递。为此，分析了开源的望远镜自动控制软件系统 RTS2，同时试验了物联网通信协议 MQTT，给出了基于 ZeroMQ 的通信架构的设计，并讨论相关的关键技术。拟采用 ZeroMQ 自主设计和实现观测控制系统的原型，目前正在编码实现中，网络通信部分的编码和测试已完成。

9.1 分布式计算

随着 MUSER 未来正式投入观测，历史数据的积累，MUSER 的历史数据的积分处理将对集群的处理性能形成巨大的挑战，如何提高数据处理性能、集群的水平扩展和资源优化，是解决未来 MUSER 数据处理的一项关健性任务。虽然部分分布式处理框架能够满足高性能的数据处理和具有横向可扩展能力，但是这些系统的管理和维护需要专业的技术知识和经验，对于天文观测和进行天文科学研究的工作人员来说，较难胜任这些维护工作。同时复杂的编程接口和较为固定的算法模式，限制了这些系统在天文海量数据分布式处理的使用。因此，提供一个轻量级、易编程、易扩展的分布式计算框架就很有必要。

对于文中提到的开源的分布式计算框架来说，在各个方面都具有不同的优势，适用于不同的应用场景，针对不同的性能考虑，没有一种通用的方法解决所有问题，互联网出身的基因决定了其并不适合在天文中应用。尽管初期能减少开发工作量，但后期应用方面的算法适配和调优方面的复杂性就会明显增多。因此，针对于天文海量数据的分布式计算的需求，设计了一个分布式计算框架 OpenCluster，主要对如何实现高性能、易编程和可扩展方面进行了研究。采用多类型工人实例，可以选择将特定的任务派发到支持该任务的节点；提供易用的编程接口降低扩展应用的代价；简化的领导者选举机制保障主节点高可用性；将数据查询服务融合在分布式计算中；利用 Mesos 提供的两级资源调度模式，实现了单任务单框架和集中仓储式调度模式，解决了集群中多种计算框架的资源隔离和共享，优先级的任务调度问题；将 OpenCluster 成功地应用在 MUSER 的实时和批量数据处理中；应用 Docker 在集群内调度保障了 MUSER 中长期服务运行的稳定性。

9.2 望远镜观测控制系统

目前我们正在设计和实现一个协同的自主望远镜观测控制系统，取名 Collaborative Telescope System，缩写为 CTS，主要目的是解决 NVST 观测控制问题，目前处于设计和代码的编写阶段，基础框架和网络通信框架已经搭建完毕。前期的工作包括了 TCS 基类设计，基础框架的搭建，网络通信 ZeroMQ 可用性进行的验证。本文中讨论了 CTS 在设计中，网络通信部分对 Socket，MQTT 和 ZeroMQ 通信技术的验证。为了实现在 NVST 上实际运行，驱动望远镜和实际观测的目标，还需要大量的工作和详细的设计。接下来的工作就集中在序列化实现，脚本引擎设计，观测计划的抽象和具体实现，先期以驱动 Camera，Dome 和 Telescope 为目标，对这几类的设备类进行详细设计。

9.3 展望

天文科学研究主要依赖于天文数据的处理结果，望远镜的口径和规模只会继续增大，观测的数据量也就会成倍的增长，如何有效地在科学数据的生命周期中做好数据的处理，是一项非常具有挑战性的工作，也是不可或缺的过程。在以 MUSER 为例的射电天文数据观测数据量的处理需求下，使用分布式数据处理技术，实现了高效的数据处理和分析，才能更有效地进行高时空分辨率的太阳数据观测活动，进一步也才能够提高科研的产出。

分布式处理系统的架构是一个很好的适应 MUSER 及未来的大型望远镜数据处理的技术，同时该技术也是目前天文大数据处理的必然选择。望远镜观测控制系统也越来越依赖于分布式实时计算结果，通过融合实时数据处理分析方法，集成所有的可用观测设备，根据科学目标和实时数据处理的结果需要选择最优的观测目标，进行有效地自主观测，提高

观测数据的获取效率与质量，这也突出了分布式计算技术的重要性。在未来的大型望远镜的数据处理中，面对更高性能的观测和处理需求，如何应对未来天文大数据的挑战，也是该技术需要推广和应用的原因。分布式处理框架的编程接口的容易程度也是影响其应用的重要因素，特别是和已有的天文数据处理代码的衔接。另外，在分布式处理框架的管理和维护上，也希望尽可能做到部署与扩展的便捷性，提高自动化运维机制，能够真正为天文科学的观测活动提供支持。

参考文献

[1] Hey T, Tansley S, Tolle K M. The Fourth Paradigm:Data-Intensive Scientific Discovery [M]. Berlin Heidelberg: Springer, 2012.

[2] Feigelson E D, Babu G J. Big data in astronomy [J]. Significance, 2012,9(4): 22–25.

[3] Andersen R. How Big Data Is Changing Astronomy (Again) [EB/OL]. [2015/03/16]. http://www.theatlantic.com/technology/archive/2012/04/how-big-data-is-changing-astronomy-again/255917/.

[4] Begeman K, Belikov A, Boxhoorn D, et al. LOFAR information system [J]. Future generation computer systems, 2011,27(3): 319-328.

[5] Onose A, Carrillo R E, Repetti A, et al. Scalable splitting algorithms for big-data interferometric imaging in the SKA era [J]. Monthly Notices of the Royal Astronomical Society, 2016,462(4): 4314-4335.

[6] Graham M J, Djorgovski S, Mahabal A, et al. Data challenges of time domain astronomy [J]. Distributed and Parallel Databases, 2012: 1-14.

[7] Bell G, Gray J, Szalay A. Petascale computational systems [J]. Computer, 2006,39(1): 110-112.

[8] Szalay A, Gray J. 2020 Computing: Science in an Exponential World [J]. Nature, 2006,440(7083): 413-414.

[9] 汤泳, 李长连, 吕英杰. 云计算在业务支撑系统中的应用初探 [J]. 邮电设计技术, 2011(10): 9-13.

[10] 颜毅华, 张坚, 陈志军, et al. 关于太阳厘米—分米波段频谱日像仪研究进展 [J]. 天文研究与技术, 2006,3(2): 91-98.

[11] 陈志军, 颜毅华, 刘玉英, et al. 关于中国厘米—分米波频谱日像仪（CSRH）选址与无线电环境监测 [J]. 天文研究与技术, 2006,3(2): 168-175.

[12] 王威, 颜毅华, 张坚, et al. CSRH 阵列设计研究及馈源设计的初步考虑 [J]. 天文研究与技术国家天文台台刊, 2006,3(2): 128-134.

[13] Wang F, Mei Y, Deng H, et al. Distributed Data-Processing Pipeline for Mingantu Ultrawide Spectral Radioheliograph [J]. Publications of the Astronomical Society of the Pacific, 2015,127(950): 383-396.

[14] 刘应波. 太阳望远镜海量数据存储关键技术研究 [D]; 中国科学院研究生院（云南天文台）, 2014.

[15] 臧大伟, 曹政, 孙凝晖. 高性能计算的发展 [J]. 科技导报, 2016,34(14): 22-28.

[16] 张军华, 臧胜涛, 单联瑜, et al. 高性能计算的发展现状及趋势 [J]. 石油地球物理勘探, 2010,45(6): 918-925.

[17] 王涛. “天河二号”超级计算机 [J]. 科学, 2013(4): 52-52.

[18] 王涛. “神威太湖之光”超级计算机 [J]. 科学, 2016(4): 5-5.

[19] 丁艺明, 刘波. 利用 GPU 进行高性能数据并行计算 [J]. 程序员, 2008(4): 97-99.

[20] Monmasson E, Cirstea M N. FPGA Design Methodology for Industrial Control Systems -- A Review [J]. IEEE Transactions on Industrial Electronics, 2007,54(4): 1824-1842.
[21] Jin H, Jespersen D, Mehrotra P, et al. High performance computing using MPI and OpenMP on multi-core parallel systems [J]. Parallel Computing, 2011,37(9): 562-575.
[22] 李梅. 基于 OpenMP 编程模型的多线程程序性能分析 [J]. 电子设计工程, 2014(23): 42-44.
[23] 李继良, 于策, 孙济洲, et al. 基于 OpenMP 的并行图像相减算法实现与分析 [J]. 天文研究与技术, 2011,08(2): 146-152.
[24] 吴韶华, 张广勇, 沈铂, et al. 一种基于 OpenMP 对天文学软件 Gridding 的优化方法 [M]. 2015.
[25] Sclocco A, Varbanescu A L, Mol J D, et al. Radio astronomy beam forming on Many-Core architectures[C]//Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International. IEEE, 2012: 1105-1116.
[26] 陈泰燃, 王威, 王锋, et al. 基于 MPI 的高性能 UVFITS 数据合成研究与应用 [J]. 天文研究与技术, 2016,13(2): 184-189.
[27] 刘立勇, 艾力·伊沙木丁, 张晋. 乌鲁木齐天文站建立脉冲星相干消色散观测系统 [J]. 天文研究与技术, 2007,4(1): 72-78.
[28] 李雪宝. 太阳望远镜海量数据并行处理技术研究 [D]; 中国科学院研究生院(云南天文台), 2015.
[29] Borthakur D. The hadoop distributed file system: Architecture and design [J]. Hadoop Project Website, 2007,11: 21.
[30] Dean J, Ghemawat S. MapReduce: A Flexible Data Processing Tool [J]. Commun ACM, 2010,53(1): 72-77.
[31] Koschel A, Heine F, Astrova I, et al. Efficiency Experiments on Hadoop and Giraph with PageRank[C]//Parallel, Distributed, and Network-Based Processing (PDP), 2016 24th Euromicro International Conference on. IEEE, 2016: 328-331.
[32] Seo S, Yoon E J, Kim J, et al. Hama: An efficient matrix computation with the mapreduce framework[C]//Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on. IEEE, 2010: 721-726.
[33] Rong C. Using mahout for clustering wikipedia's latest articles: A comparison between k-means and fuzzy c-means in the cloud[C]//Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on. IEEE, 2011: 565-569.
[34] Shanahan J G, Dai L. Large scale distributed data science using apache spark[C]//Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, 2015: 2323-2324.
[35] 黄文辉, 冯瑞. 基于 Spark Streaming 的视频/图像流处理与新的性能评估方法 [J]. 计算机工程与科学, 2015,37(11): 2055-2060.
[36] Ghesmoune M, Lebbah M, Azzag H. Micro-Batching Growing Neural Gas for Clustering

Data Streams Using Spark Streaming [J]. Procedia Computer Science, 2015,53: 158-166.
[37] Carbone P, Ewen S, Haridi S, et al. Apache flink: Stream and batch processing in a single engine [J]. Data Engineering, 2015: 28.
[38] Spangenberg N, Roth M, Franczyk B. Evaluating new approaches of big data analytics frameworks[C]//International Conference on Business Information Systems. Springer, 2015: 28-37.
[39] Namiot D. On big data stream processing [J]. International Journal of Open Information Technologies, 2015,3(8): 48-51.
[40] Philip Chen C L, Zhang C-Y. Data-intensive applications, challenges, techniques and technologies: A survey on Big Data [J]. Information Sciences, 2014,275(0): 314-347.
[41] Gorawski M, Gorawska A, Pasterak K. A survey of data stream processing tools [M]. Information Sciences and Systems 2014. Springer. 2014: 295-303.
[42] Luckow A, Mantha P, Jha S. Pilot-Abstraction: A Valid Abstraction for Data-Intensive Applications on HPC, Hadoop and Cloud Infrastructures [J]. 2015.
[43] Ekanayake J, Pallickara S, Fox G. MapReduce for Data Intensive Scientific Analyses[C]//eScience, 2008 eScience '08 IEEE Fourth International Conference on. Washington, DC, USA, 2008: 277 - 284.
[44] Szul P, Bednarz T. Productivity Frameworks in Big Data Image Processing Computations - Creating Photographic Mosaics with Hadoop and Scalding [J]. Procedia Computer Science, 2014,29(0): 2306-2314.
[45] Wiley K, Connolly A, Gardner J, et al. Astronomy in the Cloud: Using MapReduce for Image Coaddition [J]. Astronomy, 2011,123(901): 366-380.
[46] Mi C, Chen Q, Liu T. An Efficient Cross-Match Implementation Based on Directed Join Algorithm in MapReduce[C]//IEEE International Conference on Utility and Cloud Computing, Ucc 2011, Melbourne, Australia, December. 2011: 41-48.
[47] Zhang Z, Barbary K, Nothaft F A, et al. Scientific computing meets big data technology: An astronomy use case[C]//Big Data (Big Data), 2015 IEEE International Conference on. IEEE, 2015: 918-927.
[48] Brahem M, Lopes S, Yeh L, et al. AstroSpark: towards a distributed data server for big data in astronomy[C]//Proceedings of the 3rd ACM SIGSPATIAL PhD Symposium. ACM, 2016: 3.
[49] Mahmoud M S, Ensor A, Biem A, et al. Data Provenance and Management in Radio Astronomy: A Stream Computing Approach [J]. 2011.
[50] LaSalle D, Karypis G. MPI for Big Data: New Tricks for an Old Dog [J]. Parallel Computing, 2014,40(10): 754-767.
[51] 李永峰, 周敏奇, 胡华梁. 集群资源统一管理和调度技术综述 [J]. 华东师范大学学报(自然科学版), 2014,5: 17-30.
[52] Vavilapalli V K, Murthy A C, Douglas C, et al. Apache hadoop yarn: Yet another resource negotiator[C]//Proceedings of the 4th annual Symposium on Cloud Computing. ACM, 2013: 5.

[53] Apache. Apache Hadoop YARN [EB/OL]. [2017-02-08]. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.
[54] 董春涛, 李文婷, 沈晴霓, et al. Hadoop YARN 大数据计算框架及其资源调度机制研究[J]. 信息通信技术, 2015(1): 77-84.
[55] Apache. Mesos Architecture [EB/OL]. [2017-02-09]. http://mesos.apache.org/documentation/latest/architecture/.
[56] Solar M, Araya M, Farias H, et al. Cloud Services on an astronomy data center[C]//SPIE Astronomical Telescopes+ Instrumentation. International Society for Optics and Photonics, 2016: 99131G-99131G-99137.
[57] Fillatre L, Lepiller D. Processing Solutions for Big Data in Astronomy [J]. EAS Publications Series, 2016,78: 179-208.
[58] Helsley M. LXC：Linux 容器工具 [EB/OL]. [2016-12-01]. http://www.ibm.com/developerworks/cn/linux/l-lxc-containers/index.html.
[59] Merkel D. Docker: lightweight linux containers for consistent development and deployment [J]. Linux Journal, 2014,2014(239): 2.
[60] Dua R, Raja A R, Kakadia D. Virtualization vs containerization to support paas[C]//Cloud Engineering (IC2E), 2014 IEEE International Conference on. IEEE, 2014: 610-614.
[61] Docker. Docker Swarm | Docker [EB/OL]. [2017-02-03]. https://www.docker.com/products/docker-swarm.
[62] Google. What is Kubernetes? [EB/OL]. [2016-10-12]. https://kubernetes.io/docs/whatisk8s/.
[63] Bernstein D. Containers and cloud: From lxc to docker to kubernetes [J]. IEEE Cloud Computing, 2014,1(3): 81-84.
[64] Pahl C. Containerization and the paas cloud [J]. IEEE Cloud Computing, 2015,2(3): 24-31.
[65] Piraghaj S F, Dastjerdi A V, Calheiros R N, et al. Efficient virtual machine sizing for hosting containers as a service (services 2015)[C]//Services (SERVICES), 2015 IEEE World Congress on. IEEE, 2015: 31-38.
[66] Ferayorni A, Beard A, Berst C, et al. DKIST controls model for synchronization of instrument cameras, polarization modulators, and mechanisms[C]//SPIE Astronomical Telescopes+ Instrumentation. International Society for Optics and Photonics, 2014: 91520Z-91520Z-91513.
[67] Farris A, Marson R, Kern J. The ALMA Telescope Control System[C]//10th ICALEPCS Int Conf on Accelerator & Large Expt Physics Control Systems. Geneva, 2005.
[68] Schwarz J, Farris A, Sommer H. The alma software architecture[C]//SPIE Astronomical Telescopes+ Instrumentation. International Society for Optics and Photonics, 2004: 190-204.
[69] Chiozzi G, Sekoranja M, Caproni A, et al. The ALMA Common Software, ACS status and developments [J]. Geneva, Switzerland: ICALEPCS, 2005.
[70] Schwarz J, Sommer H, Jeram B, et al. The ALMA common software: dispatch from the trenches[C]//SPIE Astronomical Telescopes+ Instrumentation. International Society for Optics and Photonics, 2008: 70190W-70190W-70111.

[71] Keller C, Rimmele T, Hill F, et al. The Advanced Technology Solar Telescope [J]. Astronomische Nachrichten, 2002,323(3-4): 294-298.
[72] Ferayorni A. Instrument Control Software for the Visible Broadband Imager using ATST Common Services Framework and Base [J]. Software and Cyberinfrastructure for Astronomy II, 2012,8451.
[73] Keil S, Oschmann Jr J M, Rimmele T R, et al. Advanced Technology Solar Telescope: conceptual design and status[C]//SPIE Astronomical Telescopes+ Instrumentation. International Society for Optics and Photonics, 2004: 625-637.
[74] 罗阿理, 田园, 宋静, et al. LAMOST 观测控制系统设计与实现 [J]. 科研信息化技术与应用, 2012,3(4): 76-85.
[75] Jian W, Ge J, Xiaoqi Y, et al. The design of observatory control system of LAMOST [J]. Plasma Science and Technology, 2006,8(3): 347.
[76] 刘光曹, 王坚, 董健, et al. LAMOST 望远镜的流程控制与界面调用技术 [J]. 中国科学技术大学学报, 2010(9).
[77] 董健. 大型天文望远镜观测控制系统框架研究 [D]; 中国科学技术大学, 2011.
[78] Shporer A, Brown T, Lister T, et al. The LCOGT Network[C]//IAU Symposium. 2010: 553-555.
[79] Fraser S, Steele I A. Robotic telescope scheduling: the Liverpool Telescope experience [J]. Proc Spie, 2004,5493: 331-340.
[80] Brown T M, Baliber N, Bianco F B, et al. Las Cumbres Observatory Global Telescope Network [J]. Publications of the Astronomical Society of the Pacific, 2013,125(931): 1031-1055.
[81] Kosugi G, Sasaki T, Aoki T, et al. Subaru observation control system[C]//Optical Science, Engineering and Instrumentation'97. International Society for Optics and Photonics, 1997: 284-291.
[82] Kubánek P. RTS2—The Remote Telescope System [J]. Advances in Astronomy, 2010,2010: 902484l:902481-902484:902489.
[83] 范玉峰, 辛玉新, 白金明, et al. 丽江站 BOOTES-4 综述 [J]. 天文研究与技术, 2015,12(1): 78-88.
[84] Kubanek P. Genetic algorithm for robotic telescope scheduling [J]. arXiv preprint arXiv:10020108, 2010.
[85] Longinotti A. The ESO VLT CCD Detectors Software [J]. 1997,125(125): 418.
[86] Goodrich Bret D, Wampler Stephen B. Software controls for the ATST Solar Telescope[C]//Advanced Software, Control, and Communication Systems for Astronomy. Bellingham, 2004,5496: 518-527.
[87] Sun X-H, Chen Y. Reevaluating Amdahl's law in the multicore era [J]. Journal of Parallel and Distributed Computing, 2010,70(2): 183-188.
[88] manio. Amdahl’s law (阿姆达尔定律)的演化和思考 [EB/OL]. [2016-10-13]. http://www.manio.org/cn/progress-and-thoughts-of-amdahls-law/.

[89] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters [J]. Communications of the ACM, 2008,51(1): 107-113.
[90] Tapiador D, O'Mullane W, Brown A G A, et al. A framework for building hypercubes using MapReduce [J]. Computer Physics Communications, 2014,185(5): 1429-1438.
[91] 张兴旺, 李晨晖, 秦晓珠. 云计算环境下大规模数据处理的研究与初步实现 [J]. 现代图书情报技术, 2011,27(4): 17-23.
[92] Tannir K. Hadoop MapReduce 性能优化 [M]. 人民邮电出版社, 2015.
[93] 叶崧, 姚健东. 基于 ZeroMQ & JSON 的分布式测控系统消息通信架构设计 [J]. 现代电子技术, 2014(2): 105-109.
[94] Gerasoulis A, Yang T. On the granularity and clustering of directed acyclic task graphs [J]. IEEE Transactions on Parallel and Distributed Systems, 1993,4(6): 686-701.
[95] 田国忠, 肖创柏. 分布式系统下的 DAG 任务调度研究综述 [J]. 计算机工程与科学, 2015,37(5): 882-894.
[96] lesorb. 大数据处理之-DAG 计算 [EB/OL]. [2016-11-22]. http://lesorb.iteye.com/blog/2212600.
[97] 张新洲, 周敏奇. 大规模分布并行计算系统容错与恢复技术 [J]. 华东师范大学学报(自然科学版), 2014,5: 207-215.
[98] 尹康凯, 王明伟, 李善平. 高可用性集群中多个节点的心跳模型研究 [J]. 计算机工程, 2005,31(15): 102-103.
[99] 程学旗, 靳小龙, 杨婧, et al. 大数据技术进展与发展趋势 [J]. 科技导报, 2016,34(14): 49-59.
[100] Ismail B I, Goortani E M, Ab Karim M B, et al. Evaluation of docker as edge computing platform[C]//Open Systems (ICOS), 2015 IEEE Confernece on. IEEE, 2015: 130-135.
[101] Vohra D. Installing Kubernetes on a Multi-Node Cluster [M]. Kubernetes Microservices with Docker. Springer. 2016: 399-427.
[102] 顾宏久. 浅谈虚拟化与云计算的关系 [J]. 科学咨询:科技·管理, 2011(8): 68-69.
[103] 方海光, 陈蜜, FANGHai-guang, et al. KVM 在分布式资源共享和隔离中的应用 [J]. 计算机工程与应用, 2008,44(24): 103-105.
[104] 刘思尧, 李强, 李斌. 基于 Docker 技术的容器隔离性研究 [J]. 软件, 2015,36(4): 110-113.
[105] Jones M T. Libvirt 虚拟化库剖析 [EB/OL]. [2016-11-03]. http://www.ibm.com/developerworks/cn/linux/l-libvirt/index.html.
[106] 黎文阳. 大数据处理模型 Apache Spark 研究 [J]. 现代计算机:普及版, 2015(3): 55-60.
[107] 沈洪. Spark Streaming 在阿里的应用实践 [J]. 程序员, 2014(3): 110-112.
[108] Yan Y, Zhang J, Wang W, et al. The Chinese spectral radioheliograph—CSRH [J]. Earth, Moon, and Planets, 2009,104(1-4): 97-100.
[109] Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing[C]//Usenix Conference on Networked Systems

Design and Implementation. 2012: 2-2.
[110] Spark. Spark Programming Guide [EB/OL]. [2016-09-23]. http://spark.apache.org/docs/latest/programming-guide.html.
[111] Müller H, Thorwirth S, Roth D, et al. The Cologne database for molecular spectroscopy, CDMS [J]. Astronomy & Astrophysics, 2001,370(3): L49-L52.
[112] Cornwell T. A method of stabilizing the clean algorithm [J]. Astronomy and Astrophysics, 1983,121: 281-285.
[113] 王威. 中国厘米-分米波射电频谱日像仪(CSRH)阵列设计及相关技术方法研究 [D]; 中国科学院研究生院, 2007.
[114] 高姣姣, 王锋, 戴伟, et al. 面向射电日像仪的随机组结构剖析与文件设计 [J]. 天文研究与技术, 2013,10(4): 365-371.
[115] Kreps J, Narkhede N, Rao J. Kafka: A distributed messaging system for log processing[C]//Proceedings of the NetDB. 2011: 1-7.
[116] Saha P, Govindaraju M, Marru S, et al. Integrating Apache Airavata with Docker, Marathon, and Mesos [J]. Concurrency and Computation: Practice and Experience, 2015,28(7).
[117] Marmol V, Jnagal R, Hockin T. Networking in containers and container clusters [J]. Proceedings of netdev 01, 2015.
[118] Hintjens P. ZeroMQ: Messaging for Many Applications [M]. " O'Reilly Media, Inc.", 2013.
[119] Kubánek P, Jelínek M, French J, et al. The RTS2 Protocol[C]//Advanced Software and Control for Astronomy II. Marseille, France, 2008,7019: 70192S-70192S-70112.
[120] Happ D, Karowski N, Menzel T, et al. Meeting IoT platform requirements with open pub/sub solutions [J]. Annals of Telecommunications, 2016: 1-12.

致　谢

三年的博士学习生活犹如白驹过隙，转瞬即逝，忆回首，心中无限感慨。期间众多老师、同学、亲人、朋友们给予了我莫大的关心和帮助。使我在专业知识水平的扩展和个人能力的提升上获益匪浅，在此，我要向他们表达我最真诚的感谢。

感谢我的博士研究生导师王锋老师，从大学本科开始，到硕士研究生和博士阶段，王老师在学习和工作中，一直悉心教导。在我的课题研究上，特别是在学术论文的撰写过程中，提出了宝贵的修改建议。导师同样在生活上给了我非常多的关怀和照顾。在今后的工作中，我将更加努力，不辜负导师的培养。

感谢云南省计算机技术应用重点实验室的各位老师同仁，邓辉、戴伟、梁波、张晓丽、段晓红等多位老师多年来在工作上的支持和包容，在我读博士期间为我分担了很多的工作。多年的合作，让我始终感受到团队的温暖力量。

感谢中国科学院云南天文台，抚仙湖观测站的刘老师、徐老师等对我研究课题的支持与帮助！感谢季凯帆老师，在课题研究期间，提出很多建议和具体指导。

感谢中国科学院国家天文台，明安图观测基地的颜老师、王老师等多位老师同仁在 MUSER 数据处理中的悉心指导，你们在台站条件艰苦情况下，为 MUSER 项目建设如期完成的辛苦付出，值得我们尊敬和学习。

感谢我的父亲和母亲，感谢您们多年的养育之恩，您们的关怀和期盼为我提供着永不枯竭的克服困难的勇气和不断前进的力量。

特别感谢我的爱人，给了生活上的最大支持，在工作上给予了积极的鼓励和理解。感谢我的儿子，你给我们带来了欢声笑语，懂事可爱的你是爸爸学习的好榜样。

最后，对参加论文评审、答辩的各位老师表示衷心的感谢！

作者简介

姓名：卫守林　　性别：　男　　出生日期：1980.05.28　　籍贯：山西运城

【教育及工作经历】

2014.9 – 现在　　中国科学院云南天文台博士研究生
2006.7 – 现在　　昆明理工大学任教
2003.9 – 2006.6　　昆明理工大学硕士
1999.9 – 2003.7　　昆明理工大学本科

【攻读博士学位期间发表的论文】

[1] 卫守林, 石聪明, 高姣姣, 等. Vantage Pro 气象站实时数据采集与在 MUSER 中的应用研究[J]. 天文研究与技术, 2016, 13(1):117-123.
[2] Shoulin W, Feng W, Hui D, et al. OpenCluster: A Flexible Distributed Computing Framework for Astronomical Data Processing [J]. Publications of the Astronomical Society of the Pacific, 2017, 129(972): 024001.

【攻读博士学位期间参加的科研项目】

[1] 国家自然科学基金青年项目，11403009、CCD 集群分布式采集控制虚拟化技术研究、2015/01-2017/12、在研、主持。
[2] 云南省应用基础研究重点项目，2013FA032、面向决策服务的科技项目过程管理关键技术研究应用、2013/10-2016/09、结题、参与。
[3] 云南省应用基础研究重点项目，2013FA013、科技资源公共服务云关键技术与应用研究、2013/09-2016/08、结题、参与。
[4] 云南省应用基础研究面上项目，2013FZ018、基于流式处理的科学大数据分布式实时处理研究、2013/05-2016/04、结题、主持。
[5] 国家自然科学基金联合重点项目，U1231205、新一代厘米-分米波射电日像仪数据处理分析系统关键技术研究与实现、2013/01-2016/12、结题、参与。