****



**研 究 生 毕 业 论 文**

**（申请工程硕士学位）**

|  |  |
| --- | --- |
| **论文题目** | 异步工作流框架的  设计与实现 |
| **作者姓名** | 付 晔 |
| **学科、专业名称** | 工程硕士(软件工程方向) |
| **研究方向** | 软件工程 |
| **指导教师** | 王崇骏　教授 |

**2018年 5 月 16** **日**

**学 号： MF1632017**

**论文答辩日期： 年 月 日**

**指 导 教 师： （签字）**

**异步工作流框架的**

**设计与实现**

|  |  |
| --- | --- |
| **作 者:** | **付晔** |
| 指导教师: | **王崇骏　教授** |

|  |
| --- |
| **南京大学研究生毕业论文** |
| **(申请工程硕士学位)** |

|  |
| --- |
| **南京大学软件学院** |
| **2018年5月** |

**The Design and Implementation of Async Workflow Framework**

**Fu, Ye**

**Submitted in partial fulfillment of the requirements for the degree of Master of Engineering**

Supervised by

Professor **Wang, Chongjun**

Software Institute

**NANJING UNIVERSITY**

Nanjing, China

May, 2018

# 摘 要

工作流作为一种对领域间业务场景或业务规则的抽象建模工具，在企业中得到了十分广泛的应用。工作流技术通过抽象建模，使各种复杂的企业业务处理流程能够被计算机所理解并实际执行，从而提高了企业业务执行的效率。现如今，随着互联网技术的高速发展，企业业务处理流程的日益复杂，对工作流建模和执行效率的要求也变得越来越高。

本论文提出了一个全新的业务工作流执行引擎解决方案。与以往的大多数基于同步的工作流执行引擎不同，该框架完全基于异步调用与回调的方式，具体业务操作的执行不会再阻塞其调用者线程，从而能够以更少的系统资源消耗实现更高的性能。同时，作为一个通用的工作流执行引擎，本框架还可以被广泛应用于对性能要求较高的各企业业务流程中，在节省企业计算资源的同时为其带来更高的性能体验。

本项目主要使用Java语言进行实现，在工作流结点执行器Executor模块以及调度器Scheduler模块中大量采用了CAS、Lock-Free队列等非阻塞算法等技术，在降低系统锁同步开销的同时实现了更高的运行性能。同时，这些模块还针对计算型结点居多的工作流进行了一些特殊的设计，以期进一步提高框架运行时的整体性能。另外，在LnP（Load and Performance）模块中，本项目还通过与持续集成工具Jenkins进行集成，自动将每一次基准测试的结果发送给框架开发人员，以便及时发现框架中的潜在性能瓶颈，从而保证了框架开发过程中的持续高质量。

目前，该异步工作流框架已集成到企业内部的核心计算服务平台中，正准备在生产环境中上线。Simulation测试显示，本框架能带来近50%的系统性能提升。

**关键词：**异步、工作流、Lock-Free队列、高性能

# Abstract

Workflow, as an important abstracting and modeling tool for domain specific business scenarios or business rules, is widely used in today’s companies. Through abstraction and modeling, workflow technologies can make complex business process much easier, so that it can be understood and then executed by computers. Nowadays, with the rapid development of Internet technology and the increasing complexity of business process, the requirements for workflow modeling and execution efficiency have also become much higher.

The Async Workflow Framework, subject of this thesis, proposed a brand new solution for business workflow execution engine. Different from previous execution engines mainly based on synchronous approaches, this async framework is purely based on asynchronous call and callback. In this way, specific business logic will no longer block its caller thread, resulting in less system resource overhead and better performance. Meanwhile, as a generic workflow execution engine, this framework can also be integrated into various business processes requiring high performance.

Async Workflow Framework is implemented mainly using the Java programming language. By leveraging some non-blocking algorithms such as CAS, lock-free queues in the Executor Module and Scheduler Module, this framework can achieve high system throughput with low overhead. Besides, by integrating the LnP (Load and Performance) Module with CI, benchmark reports can be sent to developers in time and this is helpful to locate performance related issues early and ensure high project quality during development.

Currently, Async Workflow Framework has already been integrated into the core compute service system in company, and is ready for live release. Simulation testing has shown a 50 percent throughput improvement with this framework.

**Keywords:** async, workflow, lock-free queue, high performance

**目 录**

[摘 要 I](#_Toc514108651)

[Abstract II](#_Toc514108652)

[图目录 V](#_Toc514108653)

[表目录 VII](#_Toc514108654)

[第一章 绪论 1](#_Toc514108655)

[1.1 引言 1](#_Toc514108656)

[1.2 工作流发展概况 2](#_Toc514108657)

[1.3 本文的主要工作 4](#_Toc514108658)

[1.4 本文的组织结构 5](#_Toc514108659)

[第二章 技术综述 6](#_Toc514108660)

[2.1 引言 6](#_Toc514108661)

[2.2 Netty Event Loop 6](#_Toc514108662)

[2.3 Jenkins 8](#_Toc514108663)

[2.4 非阻塞算法 9](#_Toc514108664)

[2.4.1 CAS 10](#_Toc514108665)

[2.4.2 Lock Free队列 11](#_Toc514108666)

[2.5 Yammer Metrics 12](#_Toc514108667)

[2.6 Apache FreeMarker 13](#_Toc514108668)

[2.7 本章小结 14](#_Toc514108669)

[第三章 异步工作流框架的分析与设计 15](#_Toc514108670)

[3.1 引言 15](#_Toc514108671)

[3.2 系统需求分析 17](#_Toc514108672)

[3.2.1 系统功能性需求 17](#_Toc514108673)

[3.2.2 系统非功能性需求 19](#_Toc514108674)

[3.3 系统总体设计与模块设计 20](#_Toc514108675)

[3.4 执行器Executor模块详细设计 21](#_Toc514108676)

[3.4.1 执行器模块类图 21](#_Toc514108677)

[3.4.2 执行器模块顺序图 23](#_Toc514108678)

[3.4.3 结点状态状态机 25](#_Toc514108679)

[3.5 调度器Scheduler模块详细设计 26](#_Toc514108680)

[3.5.1 调度器模块类图 26](#_Toc514108681)

[3.5.2 调度器模块顺序图 28](#_Toc514108682)

[3.5.3 失败处理策略 30](#_Toc514108683)

[3.6 LnP模块详细设计 31](#_Toc514108684)

[3.6.1 LnP模块类图 31](#_Toc514108685)

[3.6.2 LnP模块流程图 34](#_Toc514108686)

[3.6.3 LnP模块顺序图 35](#_Toc514108687)

[3.7 本章小结 36](#_Toc514108688)

[第四章 异步工作流框架的实现 37](#_Toc514108689)

[4.1 引言 37](#_Toc514108690)

[4.2 执行器Executor模块实现 37](#_Toc514108691)

[4.2.1 Trivial结点执行器实现 37](#_Toc514108692)

[4.2.2 Non-Trivial结点执行器实现 39](#_Toc514108693)

[4.2.3 结点任务队列实现 39](#_Toc514108694)

[4.3 调度器Scheduler模块实现 41](#_Toc514108695)

[4.3.1 调度下游节点执行实现 42](#_Toc514108696)

[4.3.2 Least Cost调度器实现 44](#_Toc514108697)

[4.4 LnP模块实现 45](#_Toc514108698)

[4.4.1 LnP配置文件示例 45](#_Toc514108699)

[4.4.2 LnP驱动脚本实现 49](#_Toc514108700)

[4.4.3 性能指标统计实现 50](#_Toc514108701)

[4.4.4 工作流图构建实现 51](#_Toc514108702)

[4.5 工作流图结构优化实现 54](#_Toc514108703)

[4.6 本章小结 56](#_Toc514108704)

[第五章 总结与展望 57](#_Toc514108705)

[5.1 总结 57](#_Toc514108706)

[5.2 进一步工作展望 57](#_Toc514108707)

[参考文献 59](#_Toc514108708)

[致 谢 64](#_Toc514108709)

[版权及论文原创性说明 65](#_Toc514108710)

# 图目录

[图1. 1 简单工作流示例图 3](#_Toc514108492)

[图2. 1 Netty EventLoop类层次结构 7](#_Toc514108493)

[图2. 2 非阻塞算法运行时表现 9](#_Toc514108494)

[图2. 3 阻塞算法运行时表现 9](#_Toc514108495)

[图2. 4 CAS处理流程 11](#_Toc514108496)

[图2. 5 Lock-Free队列出队操作实现 12](#_Toc514108497)

[图2. 6 FreeMarker处理流程 14](#_Toc514108498)

[图3. 1 工作流生命周期 15](#_Toc514108499)

[图3. 2 异步工作流框架总体架构 20](#_Toc514108500)

[图3. 3 工作流执行引擎模块划分 20](#_Toc514108501)

[图3. 4 执行器模块类图 22](#_Toc514108502)

[图3. 5 执行器队列结构模型 23](#_Toc514108503)

[图3. 6 执行器模块顺序图 24](#_Toc514108504)

[图3. 7 Trivial执行器等待策略 25](#_Toc514108505)

[图3. 8 结点状态转换状态机 26](#_Toc514108506)

[图3. 9 调度器模块类图 27](#_Toc514108507)

[图3. 10 调度器调度过程 28](#_Toc514108508)

[图3. 11 调度器模块顺序图 29](#_Toc514108509)

[图3. 12 调度策略流程图 30](#_Toc514108510)

[图3. 13 结点失败处理策略 31](#_Toc514108511)

[图3. 14 LnP模块主要类类图 32](#_Toc514108512)

[图3. 15 LnP模块构建工作流相关类类图 33](#_Toc514108513)

[图3. 16 LnP模块流程图 34](#_Toc514108514)

[图3. 17 LnP模块运行顺序图 35](#_Toc514108515)

[图4. 1 Trivial结点执行器实现 38](#_Toc514108516)

[图4. 2 Non-Trivial结点执行器实现 39](#_Toc514108517)

[图4. 3 执行器队列工作负载统计的实现 40](#_Toc514108518)

[图4. 4 LongAccumulator类多线程并发累加错误 41](#_Toc514108519)

[图4. 5 调度器模块调度下游结点执行实现 42](#_Toc514108520)

[图4. 6 检查结点就绪状态实现 43](#_Toc514108521)

[图4. 7 检测结点就绪状态调优之前的实现 43](#_Toc514108522)

[图4. 8 检查结点就绪状态实现性能提升 44](#_Toc514108523)

[图4. 9 LeastCostEstimation调度器实现 45](#_Toc514108524)

[图4. 10 LnP配置文件示例 46](#_Toc514108525)

[图4. 11 LnP模块基准测试报告示例 48](#_Toc514108526)

[图4. 12 LnP驱动脚本详细流程图 49](#_Toc514108527)

[图4. 13 工作流运行延时统计实现 51](#_Toc514108528)

[图4. 14 digraph文件格式示例 52](#_Toc514108529)

[图4. 15 菱形工作流图构建流程实现 52](#_Toc514108530)

[图4. 16 示例菱形工作流图 53](#_Toc514108531)

[图4. 17 Transitive Reduction算法基本思想 54](#_Toc514108532)

[图4. 18 图剪枝优化算法实现 55](#_Toc514108533)

[图4. 19 图剪枝优化带来的框架性能提升 56](#_Toc514108534)

# 

# 表目录

[表2. 1 Yammer Metrics支持的Metrics类型 13](#_Toc511852937)

# 绪论

## 引言

工作流作为一种对领域间业务场景或业务规则的抽象建模工具，在现代企业中得到了十分广泛的应用。工作流技术通过对不同领域中的业务场景、或者是不同工作流程间的业务规则等进行抽象，以方便地对这些业务场景或规则进行建模，从而使得复杂的业务处理流程能够被计算机所理解，进而实现其在计算机中的自动处理和执行[Workflow, 2018]。对于不同的领域特点，相应的工作流建模方式也会有所不同。

本论文所针对项目——异步工作流框架（Async Workflow Framework）——主要针对的是支付平台风险控制领域，为该领域提供了一个高效率低延时的工作流执行解决方案，旨在解决传统同步工作流框架在工作流执行上存在的一些痛点。但该框架并不是仅能用于执行支付平台风险控制领域的业务工作流，它通过提供一些通用的接口，可以很方便地集成到各个不同领域的工作流处理系统中。

在支付平台风险控制领域中，为了保证用户信息的安全，同时防止一些如诈骗、洗钱等违规行为的发生[Chande, 2008] [刘建伟，2010]，一个简单的用户操作背后，往往涉及了及其复杂的工作流计算和处理流程，包括但不限于对不同数据源中数据的加载、网络I/O、大量变量和模型的计算、不同业务规则的执行等等。最终，基于这些不同的模型或规则计算结果，支付平台将对用户的操作给予反馈。以用户登陆操作为例，可能的平台反馈结果包括同意用户登陆、要求用户输入验证码、怀疑账号被盗而拒绝登陆请求并要求用户提供相关证明材料等等。由于用户的后续操作取决于支付平台所给予的反馈，因此在具体工作流的计算和处理过程中，用户是需要同步进行等待的。为了尽可能不影响用户的使用体验，同时又尽可能地检测出平台中的一些可能的异常操作行为，这就要求背后工作流执行引擎的高效性和准确性。以本人实习所在企业为例，一个与用户提交支付请求相关的业务工作流拥有上万个的待执行单元，且这些执行单元之间的依赖关系也非常复杂，如一个模型结果单元就可能依赖于上千个变量计算单元，而工作流的总执行延时要求则在数百毫秒以内。同时，为了保证较好的用户体验，此操作要求拒绝率不能超过6%，也即每100个用户支付请求中至多能拒绝6个请求。此项准确性的要求主要为相关变量和模型的设计带来了较高的复杂性，但如此高的性能要求则给具体工作流执行框架的设计与实现带来了巨大的挑战。

本异步工作流框架就是为了解决这一挑战而出现的，其解决了以往大多基于同步解决方案的工作流执行框架所带来的一些痛点，实现了以更少的资源使用达到更高的执行性能。该框架提供的具体解决方案包括：通过合理利用非阻塞的队列、异步调用和回调的方式，减少操作系统线程和同步的开销以及系统上下文切换[Hook et al., 2018]；通过同时对数据依赖和流依赖进行建模，对数据之间的依赖关系提供了一个全局的认识，以实现在数据不再使用时尽早释放其所占用的内存空间，减少系统垃圾回收的压力[Virouleau et al., 2016]。同时，该框架作为一个通用的工作流执行框架实现，其提供了大量的通用接口，可以方便地与现有的业务服务系统进行集成，具体的业务服务系统并不需要太多的代码变更，就可以享受到该异步工作流框架所带来的执行性能上的提升。

## 工作流发展概况

工作流技术最初是在80年代中期由FileNet和ViewStar等公司提出的，这些公司通过推出各自的工作流产品，如工作流管理系统（WfMSs: Workflow Management Systems）等，将日常工作中经常用到的一些业务处理流程集成到了一起，一定程度上简化了企业的管理[罗海滨等，2000]。在进入90年代之后，随着计算机技术的不断发展，企业业务流程复杂度的不断增加，对业务流程处理的自动化要求也随之提高，这一趋势则进一步促进了工作流相关技术的发展和成熟。

工作流作为对实际业务处理流程的一种抽象，它定义了各个具体的业务操作之间执行的先后顺序，同时这些执行过程必须遵从一定的业务处理规则[Van et al., 2004]。大多数数据科学应用都可以用一组待执行的任务来表示，这些任务互相之间存在依赖关系，通过此种建模方式，可以将实际应用抽象成工作流应用[Calheiros et al., 2015]。转化之后的工作流应用通常可以用一个有向无环图（DAG: Directed Acyclic Graph [Wang, 2013]）来表示，图1.1显示了一个简单的工作流示例图，图中的顶点表示待执行的具体任务，边则代表任务之间的相互依赖关系，如一条A->B的边则表示任务B需要在任务A执行完成后才能够被调度执行。

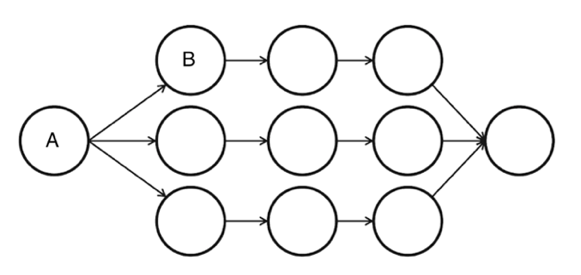


图1. 1 简单工作流示例图

在当前环境下，计算机技术飞速发展，各种云服务层出不穷，一个具体的业务执行流程往往会涉及到企业内外的多个不同的服务系统，这些服务系统通常都是分布在不同地方的[Pandey et al., 2011]。工作流管理系统就是用来将这些分布式的服务系统联系在一起，并通过利用其各自所提供的服务，来最终解决企业或终端用户的需求。

现在已有的一些工作流系统包括DAGMan（Directed Acyclic Graph MANager）、Chiron和FireWorks等等。DAGMan支持在一个较高的层次上管理工作流中的任务依赖关系，它通过与开源高吞吐量计算框架HTCondor集成，按照DAG图中定义的依赖顺序不断地将任务提交到HTCondor中执行，并处理任务的执行结果[HTCondor,2017]。Chiron通过利用数据库来管理工作流的并行执行，并支持工作流的执行优化，其适用于使用了大量数据的业务工作流[Ogasawara et al., 2013]。FireWorks是一个可以实现计算型工作流高运行时吞吐量的工作流软件，它通过实现任务打包并发执行、错误检测和恢复以及工作流运行时更新等等一系列措施，从而能够满足大多数计算型工作流对吞吐量的要求[Jain et al, 2015]。

在本论文所主要针对的支付平台风险控制领域，其相关的计算服务系统是性能攸关的。这些系统所涉及的业务工作流通常及其复杂，包含的具体业务操作量巨大。同时，系统对总体延时的要求又非常高，因此此类系统不仅要求实现高运行时吞吐量，同时还要保证较低的工作流系统平均延时，甚至是99分位的延时（即99%的工作流延时上限，只有1%的工作流延时允许超过该99分位值），这无疑对工作流执行引擎的运行时性能提出了非常大的挑战。例如本人实习所在的企业，一个用户支付验证相关的工作流中，就包含了数百个数据加载操作，且其加载的数据源还不尽相同，还包含了上万个变量计算以及几十个模型的计算操作，而所有这些操作都必须在数百毫秒内完成，并将最终计算结果返回给上层服务。如此高的性能要求，若是使用以往的基于同步的工作流执行引擎解决方案，工作流中一个具体业务操作的执行会同时阻塞其调用者线程，因此，要想达到更高的并发度，从而提高系统的运行性能，就必须使用更大的线程池。但是，更多的线程数将导致更加强烈的CPU资源竞争和更大的系统上下文切换开销，使系统的总体负荷增大，反而会降低系统的整体运行性能。

本论文中所介绍的异步工作流框架提出了一个全新的工作流执行引擎解决方案，其完全基于异步调用与回调的方式，具体业务操作的执行不再会阻塞其调用者线程。此种实现方案相比于以往的基于同步的解决方案，可以通过使用更少的线程数量达到同样的性能要求，从而实现了在相同情况下减少系统资源的消耗。同时，作为一个通用的工作流执行引擎，本框架可以被广泛应用于对性能要求较高的企业业务流程中，在节省企业计算资源的同时为其带来更高的性能体验。

## 本文的主要工作

本文所研究的异步工作流框架为执行单元多、单元之间依赖关系复杂、延时要求高的业务工作流的执行提供了一个高效率的解决方案。作为一个实际投入使用的商业平台框架，本项目对代码的质量以及框架整体的可扩展性等要求比较高。

本项目主要使用Java语言进行实现，通过采用Netty EventLoop以及CAS、Lock-Free队列等非阻塞算法等技术，在降低系统开销的同时实现了更高的吞吐量和性能。同时，本框架还针对特定领域工作流的特点进行了一些特殊的设计，以期进一步提高框架运行的整体性能，例如将工作流中的结点根据其预计执行时间长短划分为Trivial和Non-Trivial类型，并对不同类型的结点采用不同的执行策略等，以充分利用CPU等操作系统资源。另外，在本项目中还使用了Yammer Metrics对实际工作流执行过程中的性能数据指标进行统计，并通过FreeMarker将LnP基准测试的结果以清晰易读的方式发送出来。同时，该框架还通过shell脚本编程以及与持续集成工具Jenkins进行集成，实现了LnP模块的自动化，以方便框架开发人员及时发现框架中的潜在性能瓶颈，从而保证了项目开发过程中的持续高质量。

在将此异步工作流解决方案实际运用到企业内的核心计算服务系统后，实现了以更少的系统开销提供了更低的系统延时和更高的吞吐量（One Box测试显示系统99分位延时提升了近60%，吞吐量提升了近40%）。

## 本文的组织结构

本篇论文的组织结构如下：

第一章 绪论。介绍了异步工作流框架开发的项目背景，分析了工作流执行框架的发展概况，阐释了本项目拟解决的实际问题。同时，简要介绍了本项目所涉及的主要研究工作和采用的相关技术。

第二章 技术综述。介绍了该异步工作流框架在开发过程中所使用到的主要技术和框架，包括Netty提供的EventLoop，CAS、Lock-Free队列等非阻塞算法，持续集成工具Jenkins，系统性能指标统计工具Yammer Metrics以及模板框架FreeMarker等。

第三章 异步工作流框架的分析与设计。从系统的总体规划出发，对该框架所涉及的功能性需求和非功能性需求等进行了详细的分析。而后，介绍了该框架的整体架构和具体模块划分，并对执行器Executor模块、调度器Scheduler模块和LnP模块等三个核心模块所采用的具体设计方案进行了详细的阐述。

第四章 异步工作流框架的实现。在第三章对各个核心模块详细设计介绍的基础上，给出了各模块的具体实现方案，包括关键算法的具体实现流程，框架实际运行性能数据等。同时，还给出了若干示例，展示了在框架开发过程中如何发现并定位性能相关问题，并实现相应的优化，从而实现框架性能的提升。

第五章 总结与展望。对本文在该异步工作流框架开发过程中所做的工作进行了总结，同时对该框架的未来发展做了进一步的展望。

# 技术综述

## 引言

本章将简要介绍在设计与实现本异步工作流框架的过程中所应用和涉及到的相关技术，包括网络应用程序框架Netty所提供的Event Loop、CAS以及Lock-Free队列等非阻塞算法、持续集成工具Jenkins、性能数据统计工具Yammer Metrics、模板引擎FreeMarker等。同时，还将阐述本项目技术选型的具体原因。

## Netty Event Loop

Netty是一个全异步的、事件驱动的网络应用程序框架，其基于Java NIO进行实现[Maurer et al., 2015][Eckel, 2016]。利用Netty可以更高效地开发出高性能、低延时的网络应用，Twitter、Facebook和Instagram等大型公司都使用了Netty来提供相关服务。Finagle是Twitter提供的一个可扩展的RPC框架，可以用它来方便地构建支持高并发度的网络服务[Dirksen, 2015]。Finagle使用Scala语言实现，并同时提供了Scala和Java版本的API以供开发者使用[Finagle, 2018]。Facebook提供的Wangle则类似于Netty和Finagle的一个C++语言版本实现，它是一个异步的、事件驱动的客户端-服务器应用框架 [Wangle, 2018]。Wangle提供了流水线（Pipeline）和服务（Service）两种抽象，可以用于构建C++服务。

在本文所述项目——异步工作流框架——中，主要利用了Netty内置的强大线程模型以及其所提供的Lock-Free队列实现，实现了在减少系统资源消耗的同时提高框架的运行时吞吐量。

在Netty的线程模型中，使用了EventLoop接口来代表软件系统中待处理事件的循环。图2.1展示了EventLoop相关类的类层次结构（仅显示与本项目相关联的类和接口）。

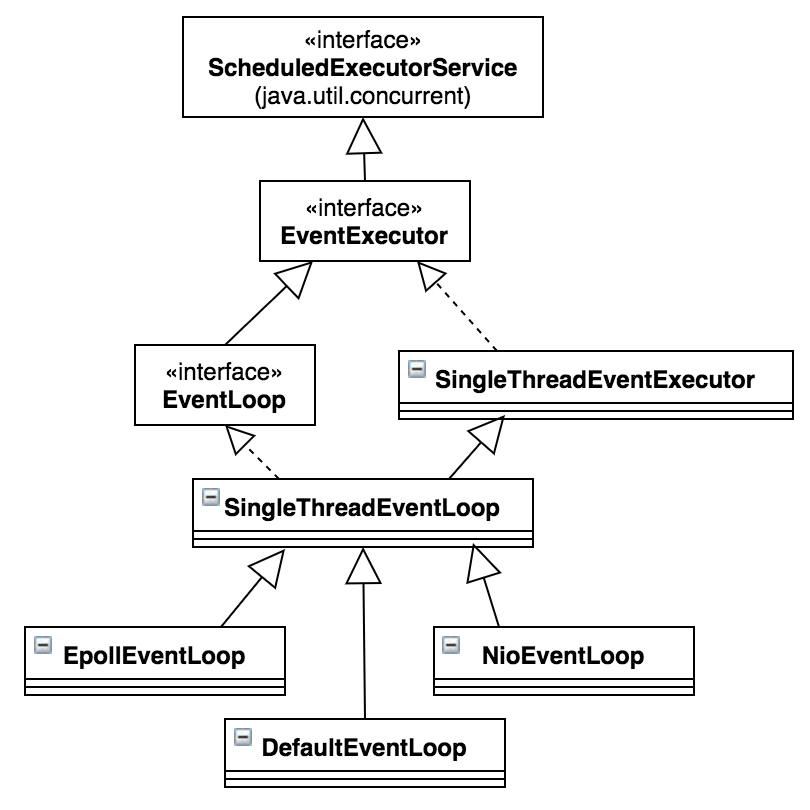


图2. 1 Netty EventLoop类层次结构

其中，顶层接口ScheduledExecutorService接口来自于Java原生类库java.util.concurrent包，EventLoop接口通过继承该接口，使得其具体实现类可以直接接受任务的提交与执行。每一个EventLoop都维护了一个自己的任务队列和一个驱动线程，这个驱动线程的本质就是不断在当前的任务队列中轮询任务并将其取出执行。当提交任务的线程与当前EventLoop的驱动线程是同一个线程时，任务可以立即被执行，否则，任务将被放入其相应的任务队列中等待EventLoop的下一次执行，这种实现可以在任务量大时显著地减少线程的上下文切换，从而提高应用程序运行时的吞吐量。

EventLoop接口主要有三个具体的实现类实现了单线程任务队列的线程模型，分别是EpollEventLoop类、NioEventLoop类以及DefaultEventLoop类。其中，EpollEventLoop类和NioEventLoop类在提供事件处理能力的基础上，还提供了执行I/O操作的能力。在Netty的实现中，每一个I/O Channel对应于一个独立的EventLoop，不同的I/O Channel之间互不影响，从而解决了I/O处理中的线程安全相关问题。

在本异步工作流框架的应用场景中，网络I/O相关操作已经交由整个异步实现蓝图中的异步RPC系统进行处理，实际工作流中的各个执行结点多为计算型结点，因此最终选择EventLoop继承结构中的DefaultEventLoop类实现。

## Jenkins

Jenkins是一个开源的软件工具，利用它可以实现软件项目的持续集成[Jenkins, 2018]。持续集成（CI: Continuous Integration），就是在软件项目的开发过程中长期持续地对项目代码进行集成测试，以便尽早发现项目中的潜在问题，保证软件开发的质量[兰洋等，2016]。此项实践在团队开发中尤为重要。

Jenkins实际上是各种不同软件工具的一个集合，它通过利用许多其他的工具，实现了软件项目持续集成过程的自动化。Jenkins可以通过监测GitHub或SVN等代码管理仓库，在检测到代码变更时自动触发maven或ant等构建工具对软件项目进行构建、测试，其还可以通过配置PMD、FindBugs等代码检查工具来在项目构建成功后执行代码质量检测[许晓斌，2011] [FindBugs, 2017]。另外，Jenkins还提供图表的形式展示项目构建的历史趋势和稳定性。当项目构建失败时，可以通过配置Jenkins发送e-mail或RSS提醒通知团队成员，开发人员也可以通过Jenkins提供的详尽构建日志来进行具体的错误定位。

在本文所述项目——异步工作流框架——的实现中，也搭建了Jenkins用于实现项目持续集成的自动化。另一方面，由于业务的需求，该框架对性能的要求较高，因此需要针对每一次代码提交执行LnP（Load and Performance）测试，以及时发现每次代码变更对框架性能的影响，方便进行性能问题的定位和恢复。Jenkins所支持的丰富的第三方插件使这一需求得以很方便的实现。在本项目中，使用了Jenkins提供的SSH Plugin插件，配置其在项目构建完成后执行远程机器（LnP测试专用机器）上的特定shell脚本文件，从而自动触发LnP基准测试的执行[SSH-Plugin,2017]。

## 非阻塞算法

在并发上下文中，非阻塞算法（Non-blocking Algorithm）是指线程之间可以相互交互通信、访问共享状态的算法，且在线程通信或访问共享数据的过程中，一个线程的操作并不会阻塞其他相关的线程。也就是说，如果一个线程的阻塞和挂起并不会导致算法中所涉及的其他线程的阻塞和挂起，则该算法就可被称为非阻塞算法[Hendler, 2011]。

图2.2展示了非阻塞算法中不同线程在访问共享数据结构时的表现。在线程A访问共享数据结构期间，若另一线程B试图访问同一数据结构，其调用会立即返回失败，线程B可以继续运行其它的操作。

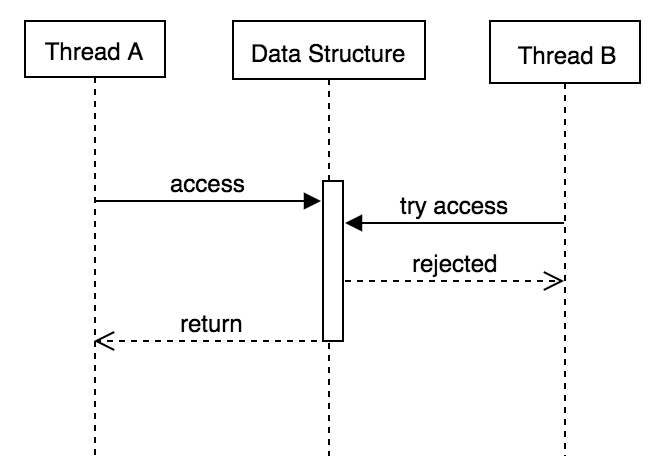


图2. 2 非阻塞算法运行时表现

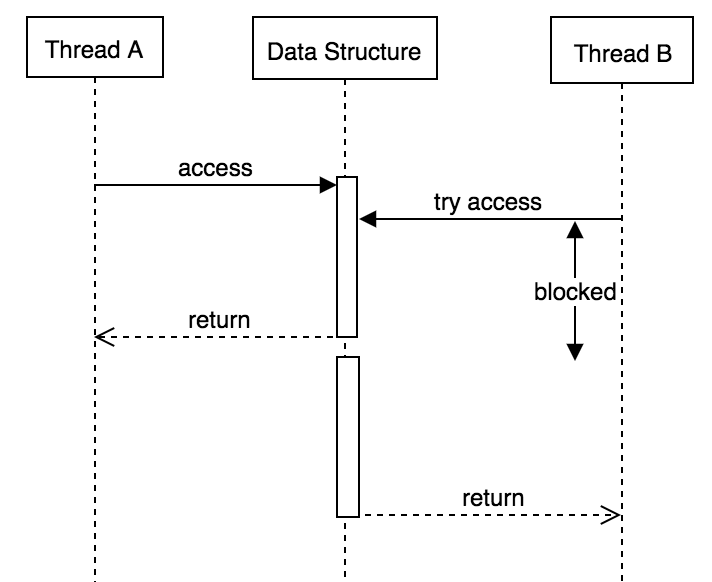


图2. 3 阻塞算法运行时表现

而在阻塞算法（Blocking Algorithm）的实现中，线程B将被一直阻塞直到它可以访问到共享的数据结构，也即线程A访问结束返回之后，图2.3展示了此种情形。

阻塞算法在访问共享数据结构时有加锁和释放锁的操作，而通常情况下，这些锁相关操作的代价是很高的。同时，线程B在等待获取锁的过程中无法进行其他任何操作，且线程B由于阻塞可能会被挂起，增加了线程上下文切换的开销。因此，在系统的并发量较大时，相比较而言，非阻塞算法通常可以提供比阻塞算法更高的运行性能。

非阻塞算法可以分为两种类型，Lock-Free（无锁）和Wait-Free（无等待）。Lock-Free算法是指在系统运行过程中，部分线程的阻塞不会影响整个系统的继续运行[Herlihy et al., 2011]。所以在Lock-Free系统中，虽然某些操作的延时因相关线程的阻塞会增加，但因为其不影响整个系统的运行，系统的总体吞吐量并不会下降。另一方面，Wait-Free算法在Lock-Free算法的基础上，还要求不同线程之间也不能相互影响，也即，一个线程的执行不能阻塞另一线程的执行[Herlihy, 1991]。因此，Wait-Free算法可以在不影响单个操作延时的前提下提高系统的整体吞吐量。相对的，Wait-Free算法的实现也要比Lock-Free算法更复杂。

在本项目中，主要使用了Lock-Free编程的方式来提高工作流框架的整体吞吐量和性能。以下分别对项目中所采用的几种方式做简要介绍。

### CAS

CAS（Compare and Swap，比较并交换）是一种Lock-Free算法，它基于CPU的指令架构，将对共享变量的更新实现为原子操作。CAS包含三个操作数，变量当前值、变量预期值和变量更新值。当且仅当变量当前值与提供的变量预期值相同时，CAS才将变量值更新为更新值，否则，说明在此操作过程中有其他线程对变量值进行了修改，CAS将不做任何操作直接返回。图2.4展示了这一过程。

Java1.5起在java.util.concurrent.atomic包中提供了许多原子变量类，用于对基本数据类型和引用类型进行CAS操作。JVM利用了底层硬件的支持来实现这一操作，若底层指令架构不支持CAS指令，JVM将使用自旋锁[Goetz et al., 2006]。同时，Java并发包中的许多其他类也在一定程度上使用了这些原子变量类。

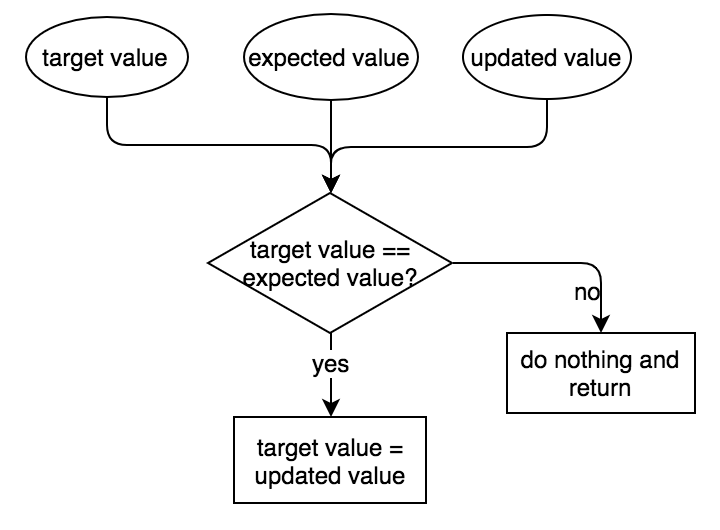


图2. 4 CAS处理流程

本项目中，在实现工作流中结点的状态机模型时即采用了Java并发类库中提供的原子变量类，保证在多线程并发时，每次只有一个线程可以成功更新特定结点的状态，如从READY到RUNNING，从RUNNING到COMPLETED或FAILURE等，从而确保每个结点只会被调度器调度执行一次。同时，更新结点状态失败的线程也无需一直循环尝试直到结点状态更新成功，从而避免了大量的CPU 空转，降低了CPU的执行开销。因为在一次工作流执行过程中，相同结点并不会被重复执行，因此只要有一个线程更新结点状态成功就能保证当前结点和其所有下游结点都能正确得到调度执行。

### Lock Free队列

Lock-Free队列是指线程安全的队列，多个线程可以并发地对队列进行存取操作，而无需担心会出现冲突和不一致等问题。同时，Lock-Free队列在实现上并不使用互斥锁，因此可以提供比传统基于锁的队列更高的并发性能，和更少的系统上下文切换。

Lock-Free队列支持所有队列的基本操作，包含入队、出队等，其实现方式有基于链表和环形数组两种。而不论采用哪种方式，其具体实现机制多是通过CAS等原子操作和一个Re-Try-Loop，不断重试直至操作成功，而非阻塞线程。这实际上与Hibernate的乐观锁机制非常类似，即在访问资源时乐观假设不会出现并发问题，不加锁直接访问，只在最后真正发生更新操作时才检查是否有并发冲突。其中Hibernate是通过增加一个额外的版本号来判断冲突的[孙卫琴，2010]。

图2.5所示代码片段展示了一个基于链表的Lock-Free队列出队操作的可能实现[Ladan et al., 2004]。

|  |
| --- |
| DeQueue() {  do {  p = head;  if (p.next == null) {  return null;  }  } while(!CAS(head, p, p.next))  return p.next.val;  } |

图2. 5 Lock-Free队列出队操作实现

在本文所述项目——异步工作流框架——中，每一个结点执行器会维护一个自己的队列用于存放其待执行的任务，而根据不同的任务调度策略，可能会有多个线程对其任务队列执行入队操作。通过以上分析可知，本项目中的任务队列为多生产者单消费者模式，因此实际选用的队列实现为JCTools提供的MpscArrayQueue[JCTools, 2018]，所谓MPSC，也即Multi Producer Single Consumer。该队列不仅实现了Lock-Free，同时还定义了大量的long型成员变量来实现CPU缓存行的填充，从而使得生产者索引producerIndex和消费者索引consumerIndex在CPU缓存中处于不同的缓存行，修改其中一个index并不会导致另一个index缓存的失效，进一步提高了并发性能[Meyers, 2013]。

## Yammer Metrics

Yammer Metrics是一个工具包，主要用来为Java应用程序的各项运行指标提供度量，以便开发人员更直观地了解系统的运行性能[Metrics, 2017]。其使用方法非常简单，只需在项目的pom文件中导入metrics-core包后，通过在Java程序中嵌入Yammer Metrics的代码，便可以轻松实现对应用各项指标的监控。

Yammer Metrics提供了多种不同的Metrics类型，以满足不同应用程序对不同系统运行指标的度量。表2.1展示了Yammer Metrics支持的五种Metrics类型、作用及其主要可使用的场景。

表2. 1 Yammer Metrics支持的Metrics类型

|  |  |  |
| --- | --- | --- |
| **Metrics类型** | **Metrics作用** | **Metrics使用场景示例** |
| Gauges （计量表） | 度量瞬时状态的数据信息 | 当前队列中待执行的任务数量 |
| Counters （计数器） | Gauges的特例，提供计数器功能 | 当前队列中待执行的任务数量，无须查询queue.size()，较Gauges效率更高 |
| Meters （公尺） | 度量某一时间段内事件发生的频率 | 每秒请求数TPS |
| Histograms（直方图） | 度量流式数据的统计分布，包括最大值、最小值、平均值以及各个分位值 | 系统响应延时的分布 |
| Timers （计时器） | 基于Histograms和Meters，度量某一代码块的执行时间及其分布情况 | 系统请求处理时间及分布 |

除了以上五种基本Metrics外，Yammer Metrics还提供了一个独立的Health Checks模块，支持将应用程序中所有健康检测相关代码集中在一处。

在本项目中，Yammer Metrics主要被用于LnP模块，以实现对工作流执行的总体延时情况及其分布的度量，主要使用了其提供的Histograms Metrics的相关功能。

## Apache FreeMarker

Apache FreeMarker是一个用纯Java编写的模板引擎，可以基于模板和数据来生成各种文本输出，包括HTML页面、e-mail、配置文件和源代码等[Freemarker,2018]。FreeMarker实现了数据展示和数据准备的分离，页面设计人员可以专注于数据的展示部分，而不用关心数据从何而来，后端开发人员则专注于处理数据的获取、准备和计算，并通过FreeMarker将处理完成后的数据注入到模板中，最终实现数据的显示。图2.6简单展示了FreeMarker的这一处理流程。

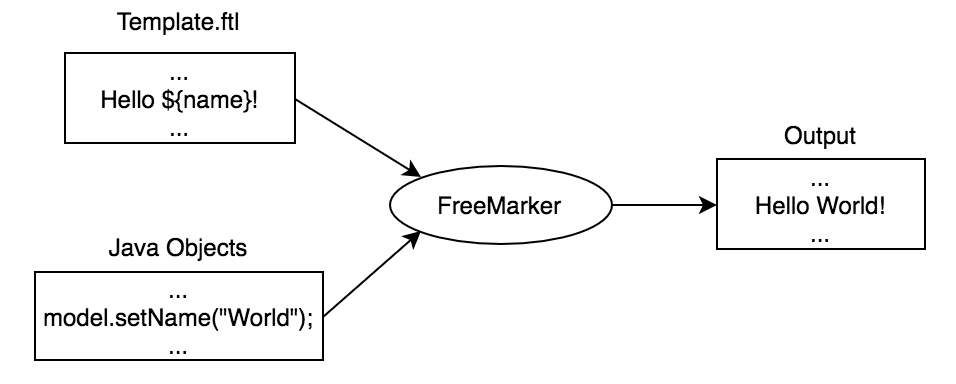


图2. 6 FreeMarker处理流程

FreeMarker支持从数据库、本地文件等不同源进行模板的载入，并根据应用程序的需要生成不同的文本输出。其处理结构使得FreeMarker非常适用于开发MVC模式的web应用，但是FreeMarker本身并不依赖于任何容器或Servlet环境，因此也可以被用于非web应用程序中。FreeMarker这一轻量级的特性使得将其嵌入到应用程序代码变得非常容易。

在本项目中，FreeMarker主要被用于在LnP模块中载入Yammer Metrics所统计的工作流运行延时等相关信息，并生成e-mail输出，以使得开发人员能够更便捷清晰地了解每一次代码变更给框架性能所带来的影响。

## 本章小结

本章主要介绍了在设计与实现异步工作流框架的过程中，所应用和涉及到的相关技术。通过直接使用Netty提供的EventLoop以及CAS、Look-Free队列等非阻塞算法，简化了开发过程的复杂度，同时提升了框架在执行具体工作流时的效率，增大了系统的吞吐量。另外本项目还配置了Jenkins以实现持续集成，从而保证了项目的代码工程质量。通过利用Yammer Metrics收集系统的各项运行指标，并通过FreeMarker解析，生成格式化、易读的e-mail，也方便开发人员时刻了解系统的运行性能信息，便于及时定位可能的性能瓶颈。

# 异步工作流框架的分析与设计

## 引言

在具体介绍本文所述项目之前，先了解一下工作流执行的生命周期过程，图3.1展示了一个典型的工作流执行生命周期。

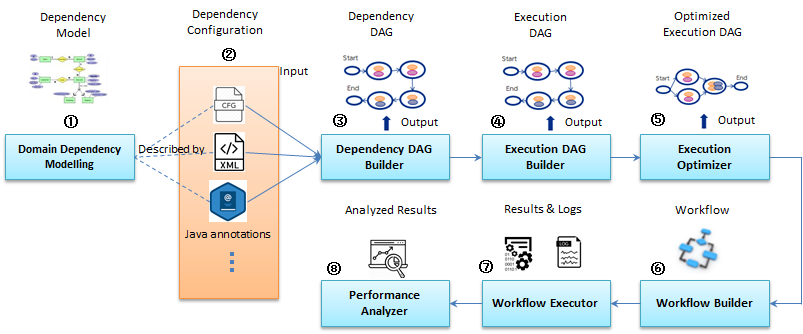


图3. 1 工作流生命周期

首先，需要针对不同领域业务流程中执行对象之间的依赖关系进行建模，以使得计算机能够理解不同的工作流并执行。关于对象依赖关系的配置，可以通过DSL（Domain Specific Language，领域特定语言）、XML配置文件或Java注解等不同方式。

然后，根据具体的依赖关系配置，可以通过解析构建出一张领域全局的依赖DAG（Directed Acyclic Graph，有向无环图）图。在构建出依赖DAG图之后，可以根据需要将其转化为具体的执行DAG图。执行DAG图中的结点将包含一些与工作流执行相关的信息，包括该结点的优先级、预计执行时间等。同时，还可以基于一些特定的规则对执行DAG图进行可能的优化，如按照数据源实现数据的批量加载、图中多余边的剪枝等等。

最后，根据不同的用户请求从执行DAG图中抽取出待执行的子图，并将其转化为真正的工作流执行，以获取相应的计算结果。其中，具体工作流的构建主要是在执行DAG图子图的基础上构建一些执行上下文信息，初始化一些Metrics统计等。在工作流执行完成返回结果之后，还可以根据运行过程中记录的各项性能指标或产生的日志文件等对整个工作流框架的执行性能进行分析，发现潜在的性能瓶颈并进行优化。

本异步工作流框架作为整个异步实现蓝图（还包括异步RPC系统、异步数据访问系统）中的一个核心系统，其并不直接服务于最终用户，而是作为一个工作流执行引擎与具体的平台服务系统进行集成，以提高平台上具体业务工作流执行的效率，从而实现在更短的时间内返回上层服务所需要的数据或计算结果，以实现更快的响应时间和更好的用户体验。举例来说，在本人实习企业内部，该异步工作流框架的主要用户之一就是一个统一计算服务（Unified Compute Service）系统，以下简称UCS系统。该UCS系统是企业风控部门提供的一个核心计算服务系统，其主要负责接收用户请求，并根据不同的用户请求类型进行各种变量和模型的计算，同时它还会调用下层数据访问服务加载变量和模型计算所需的数据。本异步工作流框架就是通过集成在该UCS系统中，用于实现变量和模型结果的高效计算，从而使得UCS能在更短的时间内返回上层调用，提高整个系统的可用性。

该异步工作流框架主要采用Java语言进行实现，其主要可以分为以下四个大块：依赖DAG构建部分，执行DAG构建部分，工作流构建部分和工作流执行引擎部分。

依赖DAG构建部分负责对领域对象之间的依赖关系进行建模，将不同的依赖关系配置抽象为一个有向无环图。在实际进行建模时，本框架除了实现传统工作流框架都关注的执行流依赖（flow dependency）之外，还对数据依赖（data dependency）进行了建模。因此在具体工作流的执行过程中，可以尽早地释放不再需要的数据结构所占用的内存空间，而无需等到工作流执行完成后一次性释放，从而缓解了JVM垃圾回收的压力。执行DAG构建部分负责加载并初始化结点执行的一些相关信息，从而将依赖DAG图转化为执行DAG图。工作流构建部分负责根据具体的用户请求类型将执行DAG图转化为实际用于执行的工作流。工作流执行引擎部分则负责具体工作流的执行和计算结果的返回。本框架针对一些特殊的业务工作流还实现了一些特别的优化，从而进一步提高了整个系统的吞吐量。

本章后续部分的结构安排如下：3.2节将详细阐述本框架的需求分析；3.3节将描述系统的总体设计；3.4至3.6节将分别介绍执行器Executor模块、调度器Scheduler模块和LnP模块等三大核心模块的详细设计。

## 系统需求分析

本节将主要从功能性需求和非功能性需求两个角度出发，对本项目的需求进行详细分析。

### 系统功能性需求

本论文中所介绍的异步工作流框架并不直接与终端用户进行交互，而是作为一个具体的工作流执行引擎集成到具体的平台服务系统中，用户只需向集成了本执行引擎的平台服务系统发送请求，并获取最终的计算结果即可。因此，本小节将围绕3.1节中所描述的工作流执行基本生命周期过程来对本框架的功能性需求进行分析与介绍。

针对依赖DAG的构建，主要包含以下3个需求：

（1）支持不同的领域依赖关系配置：要构建依赖DAG，则首先必须了解待解决领域的对象之间的依赖关系。如某一模型的结果依赖于哪几个变量的值，某一变量的计算依赖于哪些数据的加载等等。本框架需要支持多种不同的领域依赖关系建模方式，包括XML配置文件、Java注解等，如有可能，还应包括DSL描述配置[Neeraj et al., 2017]；

（2）根据依赖关系配置构建依赖DAG图：本框架需要解析各种不同的配置方式，并据此构建出一张DAG图来表示领域对象间的依赖关系，包括数据依赖和执行流依赖；

（3）配置准实时更新：针对特定的依赖关系配置方式，如通过XML配置文件的配置方式，在配置文件发生变更时，框架需要检测到并重新触发整个工作流构建流程。

针对执行DAG的构建，主要包含以下3个需求：

（1）将依赖DAG图转化为执行DAG图：本框架需要将依赖DAG图中的结点一一转化为具体的执行结点，构建出一张整体的执行DAG图。执行结点需要包含结点的优先级、预计执行时间等信息；

（2）支持从文本文件中获取结点执行相关信息：结点的预计执行时间信息需支持从文本文件中获取；

（3）优化执行DAG图：本框架需要在原始的执行DAG图上实现一定程度的优化，包括通用的整体优化，如图剪枝，和特定的局部优化，如聚合结点以实现相关数据的批量加载。

针对工作流的构建，主要包含以下1个需求：

根据用户请求构建工作流：执行DAG图包含了所有结点的相关执行信息，是一个全局的图，而具体的用户请求往往只涉及到图中一部分结点的执行。因此，本框架需要根据不同的用户请求，从构建完成的执行DAG图中抽取出不同的子图构建成具体的执行工作流。

针对工作流的执行，主要包含以下5个需求：

（1）调度执行主工作流：对于一个具体的工作流图，本框架需要能够正确的进行调度和执行。所谓正确，即调度顺序需满足结点之间的依赖关系，相同结点不会被重复执行，根据策略合理处理执行失败结点，并最终执行到END结点结束；

（2）执行post工作流：除了执行主工作流外，本框架还需支持post工作流的调度执行。Post工作流主要是用于执行一些如日志记录、计算结果存储等操作，其与具体的用户请求无关，通常在主工作流执行完成后执行，不影响用户请求的返回；

（3）计算结果返回：本框架需支持在主工作流执行完成后立即将计算结果返回给上层调用者，以减少系统整体的响应时间，因为post工作流的执行与返回结果的计算无关；

（4）记录运行性能数据指标：在调度执行主工作流的过程中，框架需要能够记录相关的运行数据指标，包括各结点的实际运行时间、失败结点的个数及其失败原因、框架的整体响应时间等，从而方便在出现性能问题时对框架运行情况进行分析；

（5）定时更新结点的执行相关信息：本框架需支持在运行过程中，根据各结点的实际执行情况，定时更新其预计执行时间等执行相关信息，以确保后续的调度策略与结点的实际运行情况尽可能地一致。

### 系统非功能性需求

3.2.1节主要从功能性的角度分析了本异步工作流框架所需要实现的具体行为，除此之外，系统对运行时的性能也非常关注。

在本人实习企业内部，该框架作为一个要集成在平台级统一计算服务系统中的一个工作流执行引擎，其运行性能必须满足整个计算服务系统对性能的要求，这些要求具体可分为以下3项非功能性需求：

（1）可用性

由于企业内的统一计算服务（Unified Compute Service）系统是直接与终端用户进行交互，接收用户请求并返回计算结果，其可用性通常要求达到99.2%以上，这一可用性除了UCS系统本身的可用性外，还包含了如网络等外界因素的影响。因此，本框架作为UCS系统中核心的工作流计算和执行引擎，其对可用性的要求将会更高；

（2）高性能

不同的用户操作会触发不同的工作流图执行，而对于不同类型的工作流图，其对运行延时的要求也会有所不同。如登陆操作相关的工作流会要求在100毫秒内返回验证结果，而支付操作相关的工作流由于其涉及的计算更为复杂，则是要求在800毫秒内返回结果即可。对于不同的用户操作，系统在保证稳定运行的同时，还需保证即使在并发量比较大的情况下，也能满足各个不同工作流图对于响应延时的要求；

（3）可靠性

系统需保证较高的可靠性，在调度执行工作流的过程中，非关键路径上结点的失败不能影响整个工作流的执行和最终计算结果的返回。

## 系统总体设计与模块设计

图3.2显示了本异步工作流框架的总体架构。该框架主要可以分为以下四大块：依赖DAG构建部分、执行DAG构建部分、工作流构建部分和工作流执行引擎部分，其中工作流执行引擎部分是整个异步工作流框架的核心，由它来实现工作流调度和执行逻辑。

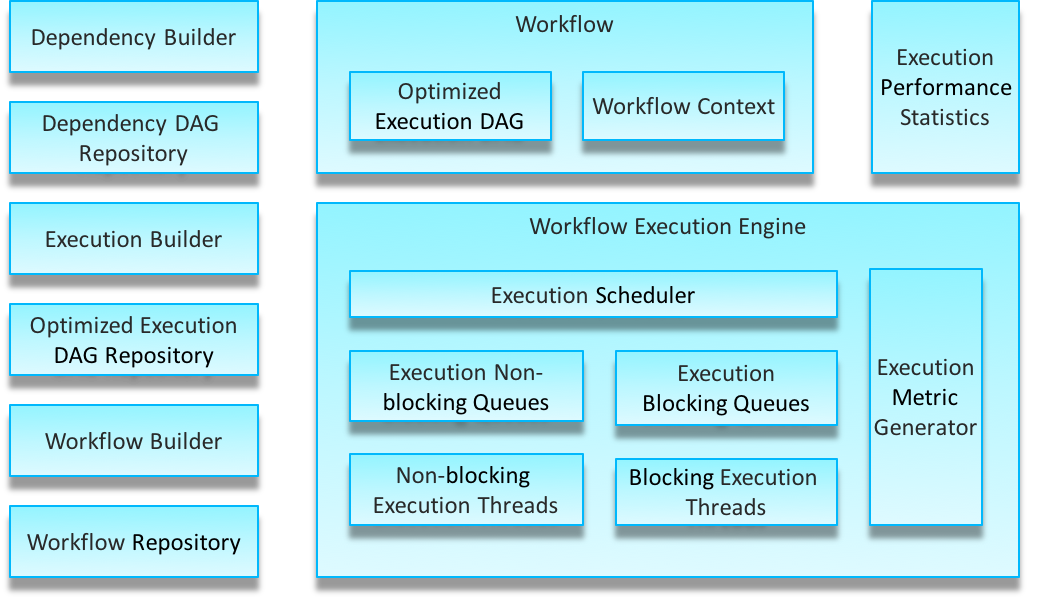


图3. 2 异步工作流框架总体架构

图3.3展示了工作流执行引擎部分的具体模块结构划分，其主要可分为以下五个模块：执行Execution模块、执行器Executor模块、度量Metrics模块、调度器Scheduler模块以及LnP（Load and Performance）模块

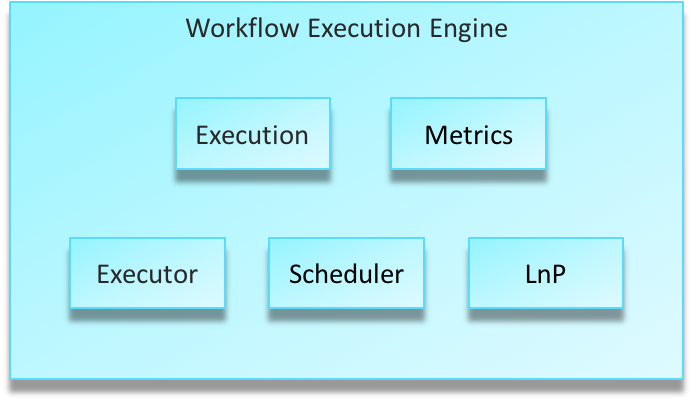


图3. 3 工作流执行引擎模块划分

执行Execution模块主要负责与工作流执行相关类的定义与实现，包括基本执行图（Execution DAG）的定义和构建，基本同步、异步结点类的定义，执行上下文（Execution Context）的实现，同时还包括监听器listener的定义，用于实现异步结点执行的各种回调函数。

执行器Executor模块主要负责各种执行器的实现，用于针对不同类型特征的工作流结点，以最大化工作流结点执行的效率。同时该模块还包含了执行器队列的定义及实现。

度量Metrics模块主要负责在实际工作流执行过程中记录一些结点的运行时间等相关信息，以便于在相同工作流的后续执行过程中根据结点的实际运行时间调整结点的类型，并根据不同的类型实现相关的优化，进一步提高工作流执行的效率。

调度器Scheduler模块主要负责实现不同的结点调度策略，以适应不同的应用场景，满足不同条件下对系统运行延时的要求。

LnP模块主要负责整个框架的性能测试。通过与Jenkins进行集成，在每次代码变更和项目构建成功后，执行相同的工作流图来比较本次代码变更所带来的系统性能影响，以便于及时发现性能相关问题并解决，从而保证框架开发过程中的持续高质量。

在本篇论文中，将选取本人主要参与的执行器Executor模块、调度器Scheduler模块和LnP模块来进行重点分析和介绍。

## 执行器Executor模块详细设计

本节主要阐述了执行器Executor模块的详细设计，包括该模块的类图、执行顺序图以及工作流图中结点的状态转换状态机。

### 执行器模块类图

图3.4显示了执行器模块的类图。其中AbstractNodeExecutor类提供结点执行的一些通用逻辑，如判断当前结点是否可执行以及根据结点类型（同步结点还是异步结点）实际调用结点业务逻辑执行等。在本框架中，结点的类型可以分为同步（Sync）的和异步（Async）的，不同类型的结点将对应不同的执行方式。

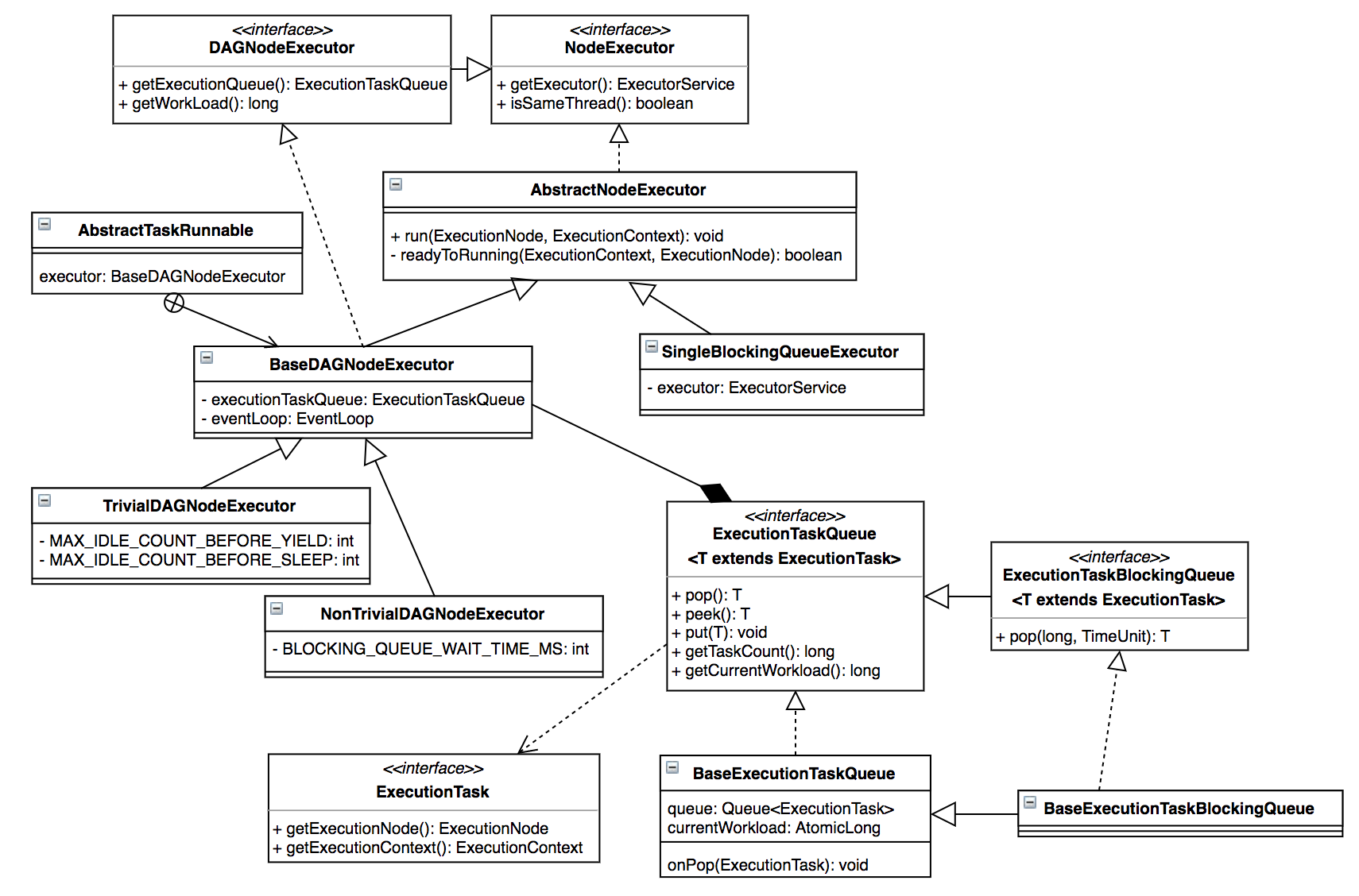


图3. 4 执行器模块类图

AbstractNodeExecutor类的两个子类BaseDAGNodeExecutor类和SingleBlockingQueueExecutor类分别提供了两种不同的线程模型来实现结点的执行。

SingleBlockingQueueExecutor类，正如其类名所示，其利用了Java原生的线程池来实现工作流中结点的执行，线程池中的所有线程都将共享同一个阻塞队列，对其进行入队和出队操作。此种实现方式是实现工作流结点执行的一种最简单的方式。

另一方面，BaseDAGNodeExecutor类针对计算结点很多的业务工作流做了一些特殊的设计，以尽可能地减少系统运行时的上下文切换并提高吞吐量。

首先，根据结点的预计执行时间，将其划分为两种类型，Trivial结点和Non-Trivial结点。Trivial结点是指预计执行时间很短的结点，如小于1毫秒。Non-Trivial结点是指预计执行时间相对较长的结点，如大于1毫秒。对于Trivial结点的执行，每个线程将拥有一个自己的非阻塞队列，用于存储该线程待执行的工作流结点，该队列是多生产者单消费者模式的。而对于Non-Trivial结点的执行，由于其预计执行时间较长，同时，对于计算结点偏多的工作流而言，其数量相对于Trivial结点来说是比较少的，因此每个线程将维护一个自己的阻塞队列用于存储其待执行结点。另外，这些线程和队列都是全局可见的，可以供同时运行的多个工作流共享。图3.5显示了此种设计的线程与其队列的关系结构模型。

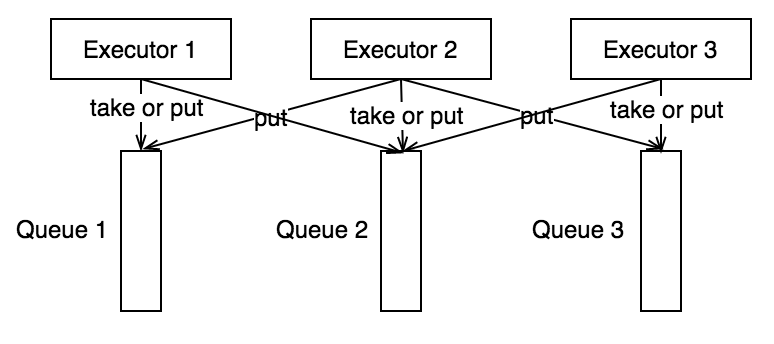


图3. 5 执行器队列结构模型

### 执行器模块顺序图

在执行器模块中，采用不同的线程和队列结构模型只会导致具体的入队、出队等操作模式不同，从而使得框架在整体的性能表现上出现差异。但在实际执行工作流中的结点时，框架的行为表现只跟待执行结点的类型有关，也就是说，框架的行为表现只取决于当前待执行的结点是同步结点还是异步结点。若当前待执行结点为同步节点，则执行器会同步等待该结点执行完成，然后通知调度器当前结点完成，以便调度器调度其下游结点执行。而如果当前待执行结点为异步结点，则执行器仅需触发该结点执行，并传递给其一个监听器。当该异步结点执行结束时，它会根据其不同的完成状态（成功或者失败）自动回调监听器中的相关方法，通知调度器的工作则由监听器来完成。在此异步结点的执行过程中，执行器线程可以继续执行其自身的业务逻辑，如开始执行下一个待执行结点等。

图3.6显示了执行器模块在处理不同类型结点执行时的系统顺序图。

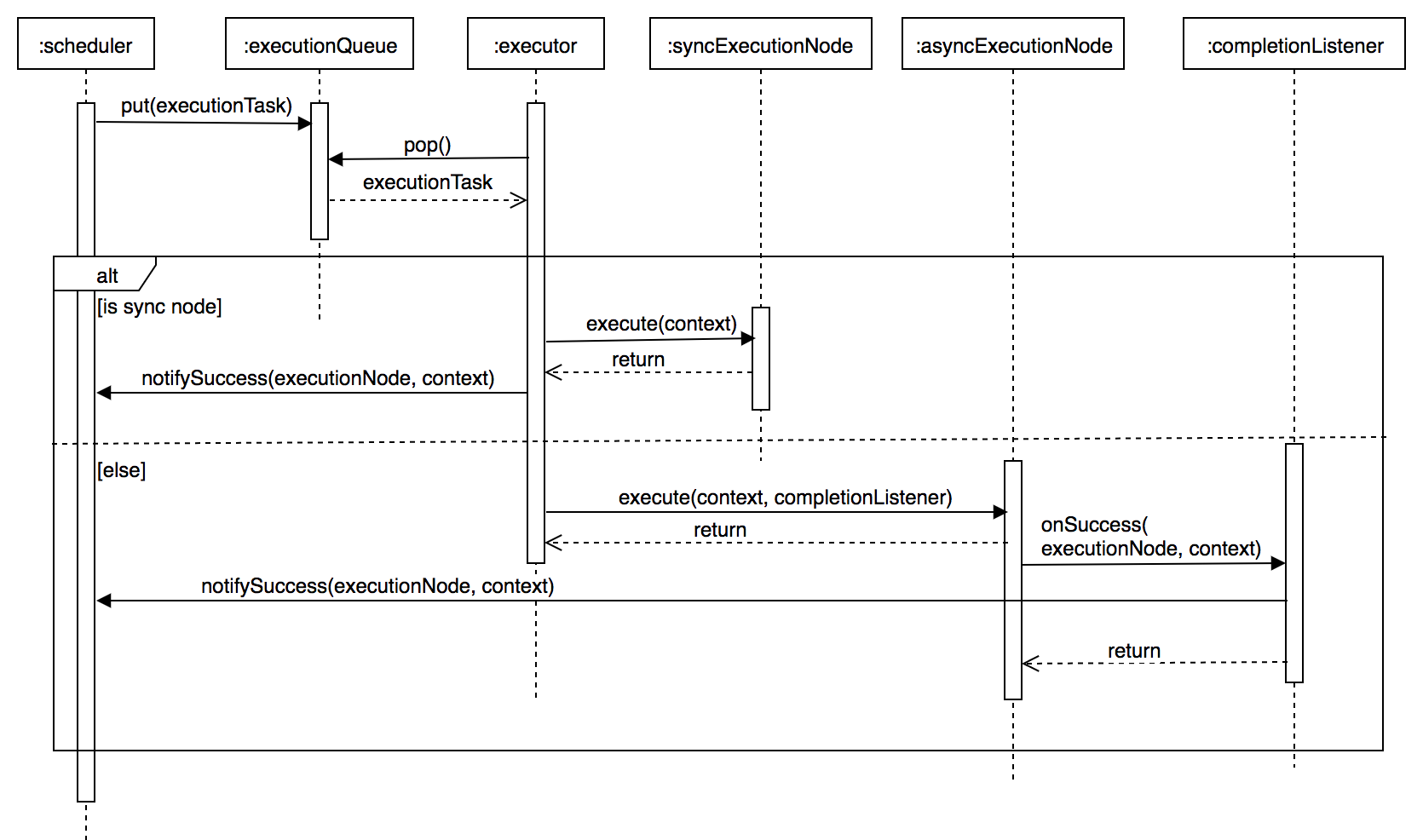


图3. 6 执行器模块顺序图

工作流中的结点除了可以根据操作行为划分为同步和异步结点之外，还可以根据其预计执行时间长短划分为Trivial和Non-Trivial结点。而由于Trivial结点的预计执行时间较短，其通常被视为同步结点执行，如此设计可以减少线程上下文切换的开销。因此对于Trivial结点而言，其相关的执行器队列是非阻塞的，因此当并发度不高，整个系统的负载量不大时，有可能导致相关执行器队列长时间为空。在这种情况下，对应的执行器线程可能会因为长时间获取不到待执行结点而空转，而Trivial结点的执行器线程数是和CPU核数相匹配的，这就会导致CPU利用率虚高，造成系统资源的浪费。针对此种情形，在参考了Disruptor的不同等待策略（Waiting Strategy）[Thompson, 2011]之后，本框架最终采取了yield和sleep相结合的方式，在CPU资源和工作流执行性能之间实现了权衡。图3.7描绘了此种等待策略的具体执行逻辑。当执行器线程空转次数超过yield阈值而未超过sleep阈值时，将采用yield策略尝试让出CPU资源；当执行器线程持续空转导致其超过了sleep阈值时，则会采用sleep策略强制其出让CPU资源。

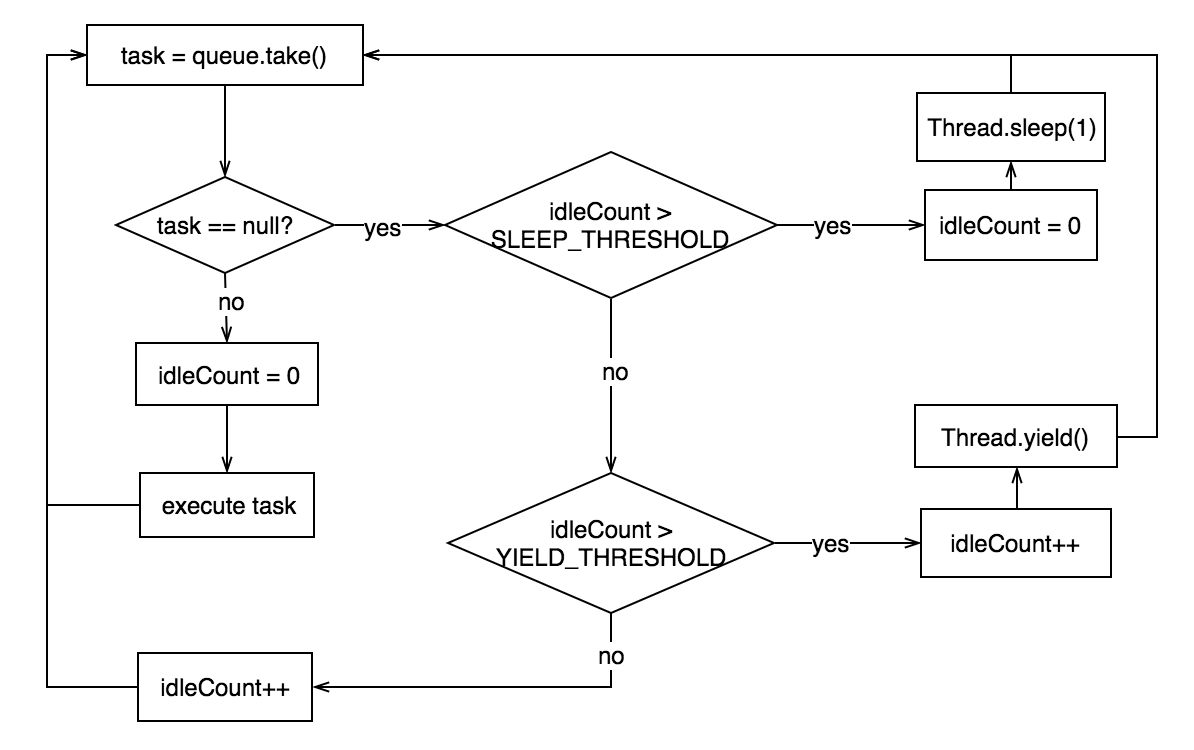


图3. 7 Trivial执行器等待策略

### 结点状态状态机

调度器在调度结点执行，以及执行器从队列中取出待执行结点并实际执行该结点时，会涉及到结点状态的改变。总的来说，一个工作流结点可以有以下五种可能的状态：Not Started（未开始）、Ready（就绪）、Running（运行）、Completed（完成）和Failed（失败）。图3.8描绘了这五种状态之间相互转换的一个状态机，以及触发不同转换的条件。

开始时，所有结点的状态都是Not Started。调度器将从工作流的起始结点开始，将可执行的结点状态变更为Ready，并根据其预计执行时间长短将结点放入相应的执行器队列。当执行器线程从队列中取出待执行结点时，会将结点状态从Ready变为Running，并根据具体的结点类型实际执行结点逻辑。若结点成功执行完成或者其ignorableIfFailure（忽略错误）标志被设置为TRUE，则其状态会变为Completed。若结点执行过程中遭遇异常或者执行超时，则其状态将变为Failed。

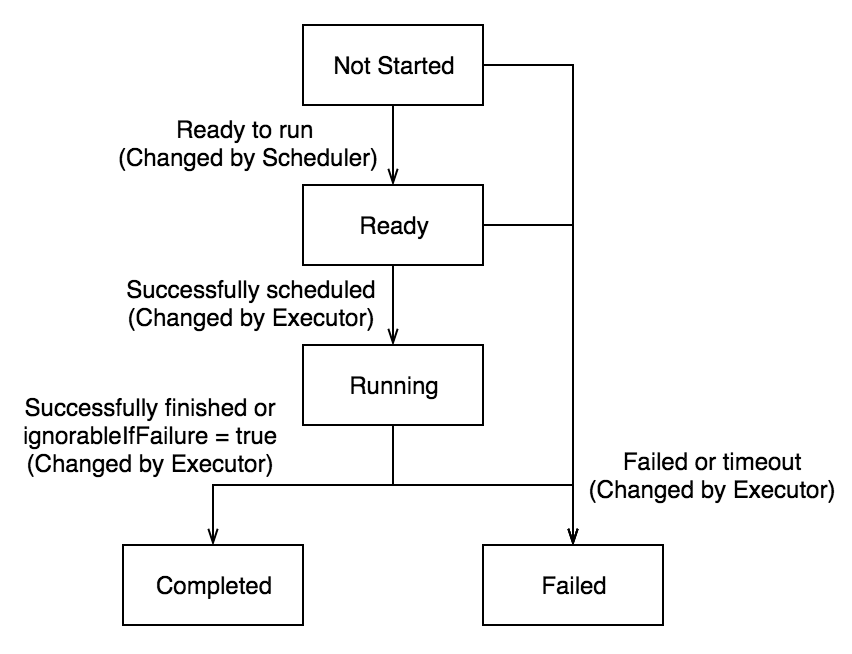


图3. 8 结点状态转换状态机

所有的结点状态转换都是通过CAS算法来实现的，从而确保在单个工作流执行过程中，每个结点只会被调度器调度执行一次。

## 调度器Scheduler模块详细设计

本节主要阐述了调度器Scheduler模块的详细设计，包括该模块的类图、执行顺序图和对工作流中失败结点的处理策略。

### 调度器模块类图

图3.9显示了调度器模块的类图。

ExecutionScheduler接口是该模块的顶层接口，其定义了调度器对外的表现行为，各个具体调度器类均实现了该接口。其中，BaseExecutionScheduler类是这些具体实现类的一个基类，它提供了一些不同调度器所需的公用逻辑实现，如实现结点执行成功或失败时的相关处理、调度下游结点的执行等。与执行器模块相类似，BaseExecutionScheduler类也有两个子类，用于针对不同的执行器线程和队列模型来实现不同的调度。

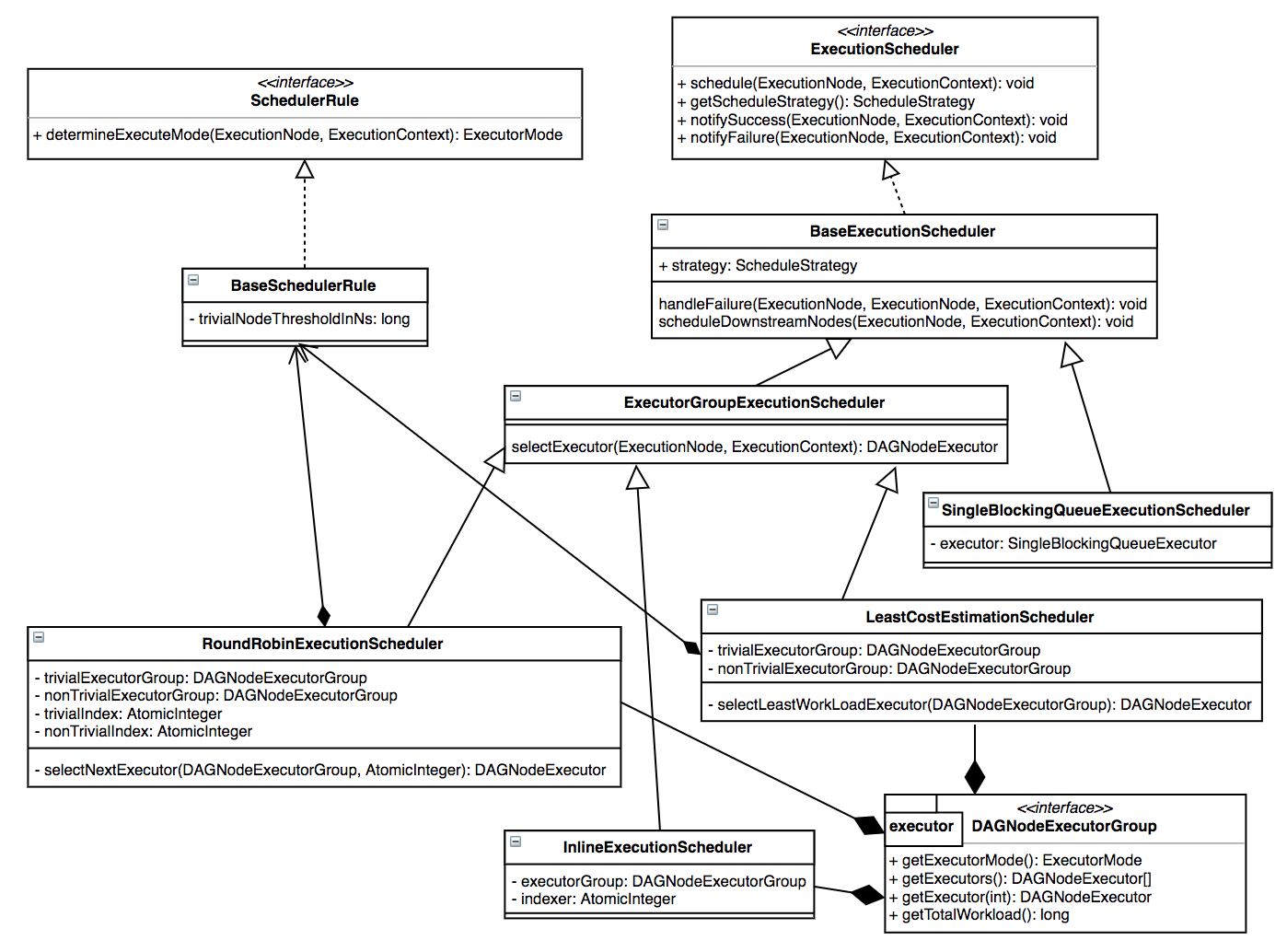


图3. 9 调度器模块类图

SingleBlockingQueueExecutionScheduler类针对采用了Java原生线程池和全局的单一阻塞队列的执行器实现，在调度就绪结点执行时，直接将其提交到对应的执行器执行即可。具体由哪个线程执行该结点，何时执行，这些都交由JVM底层调度实现。

另一方面，ExecutorGroupExecutionScheduler类针对每一个执行器线程都有一个单独的任务队列的执行器实现。该类的三个子类分别实现了三种不同的调度策略，以适应不同的应用场景，满足不同条件下对系统运行延时的要求。

RoundRobinExecutionScheduler类使用的调度策略最为简单，它每次将选取一组执行器线程中的下一个执行器线程，将待执行结点放入其相应的任务队列中。

InlineExecutionScheduler类则在round robin调度策略的基础上提供了一个捷径。其在选择具体的执行器线程之前，会先检查当前调度线程，如若当前调度线程恰好是一个执行器线程，则直接使用当前线程执行任务，否则就像round robin调度策略一样，选取下一个执行器线程执行。

LeastCostEstimationScheduler类所采用的调度策略则较为复杂，其在调度过程中将各个执行器线程的当前工作负载考虑在内，并总是将待执行结点放入当前工作负载最低的执行器队列中。执行器的工作负载定义为其相应队列中所有待执行任务的预计执行时间总和，采用此种调度策略可以使得当前结点被更快地出队执行，从而达到更好的执行性能。

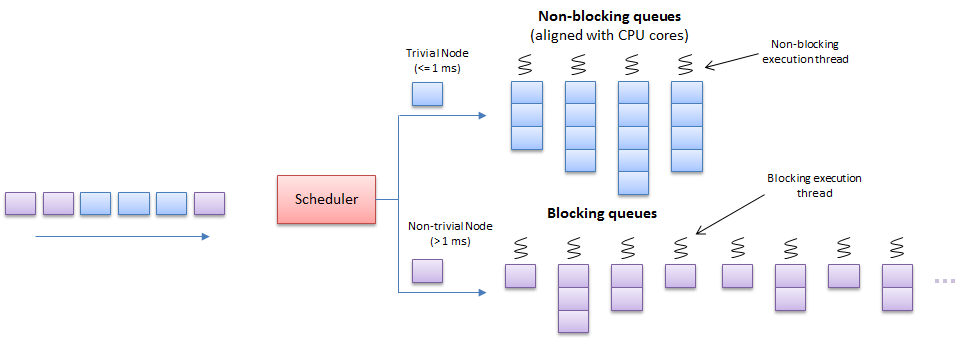


图3. 10 调度器调度过程

图3.10显示了ExecutorGroupExecutionScheduler类的调度过程，其根据不同结点的预计执行时间和不同的调度策略，将结点放入不同的执行器任务队列中，以实现结点的高效执行。

### 调度器模块顺序图

图3.11显示了调度器模块与其他模块类交互的顺序图。

一个具体工作流的执行总是从起始结点START\_NODE开始的，调度器通过将该结点提交到线程池执行，或者将其放入选定执行器的待执行结点队列中，从而触发整个工作流的执行过程。在执行器执行完某一结点后，会根据结点的结束状态通知调度器。若结点执行成功，则通知调度器调度其下游结点执行，否则，通知调度器处理失败。这一通知和调度处理过程不断地循环进行，直到执行完工作流中的最后一个结点，也即END\_NODE，并将最终执行结果返回给客户端。

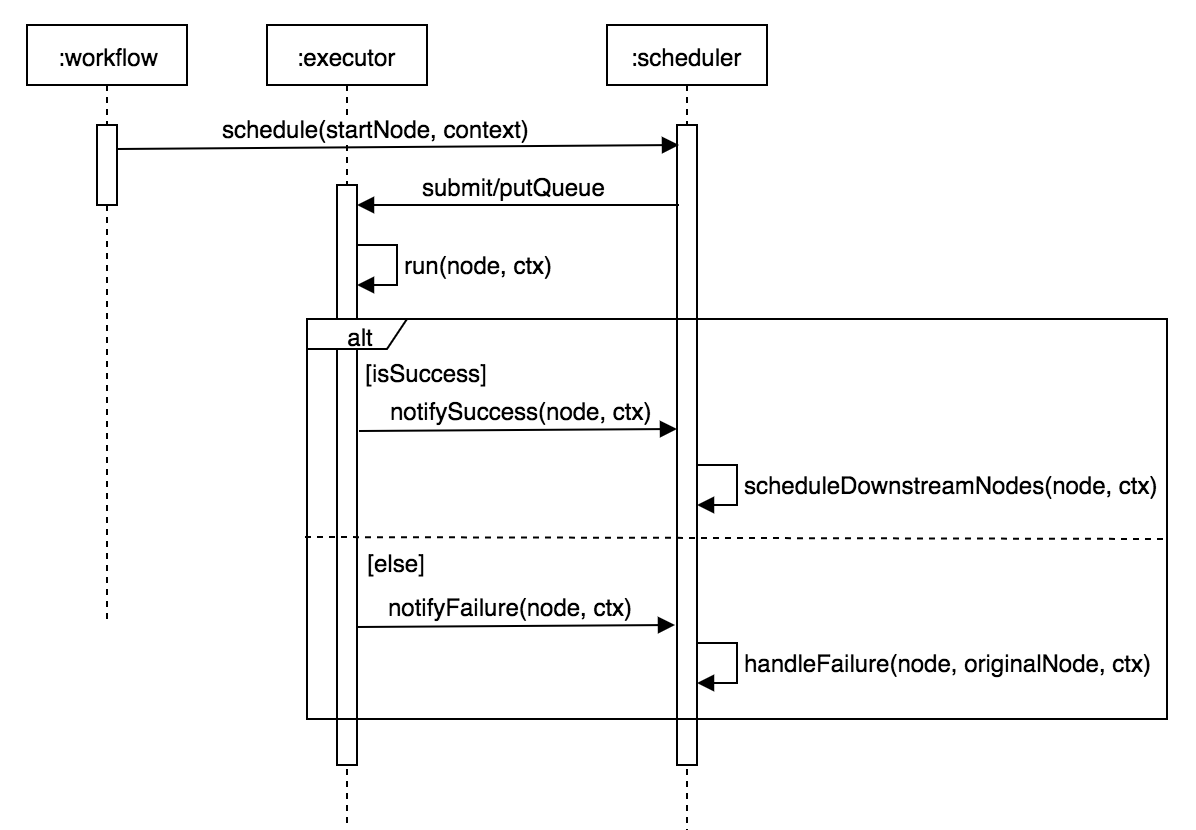


图3. 11 调度器模块顺序图

在调度器调度起始结点执行或后续调度下游结点执行时，与执行器模块在执行结点时只需考虑结点为同步的还是异步的不同，调度器模块的具体调度过程会随着执行器模块采用的线程和队列模型的不同而不同。对于执行器模块使用全局单一阻塞队列的情况而言，调度器模块只需向Java线程池提交任务即可，比较简单。而对于每个执行器线程维护一个单独任务队列的情况而言，调度器需先根据不同的调度策略从一组执行器线程中选择一个执行器线程，再将待执行结点放入其对应的任务队列中。

图3.12显示了在不同的调度策略下执行器线程的具体选择过程。首先需要判断当前使用的调度策略的具体类型，若为inline调度策略，且当前线程恰好是一个执行器线程，则选取当前线程，否则退化为round robin调度策略；若为round robin调度策略，则获取执行器线程组中的下一个线程；若为least cost调度策略，则遍历执行器线程组，并选取当前工作负载最小的执行器线程。在选定了具体的执行器线程后，直接将待执行结点放入其相应的任务队列中即可。

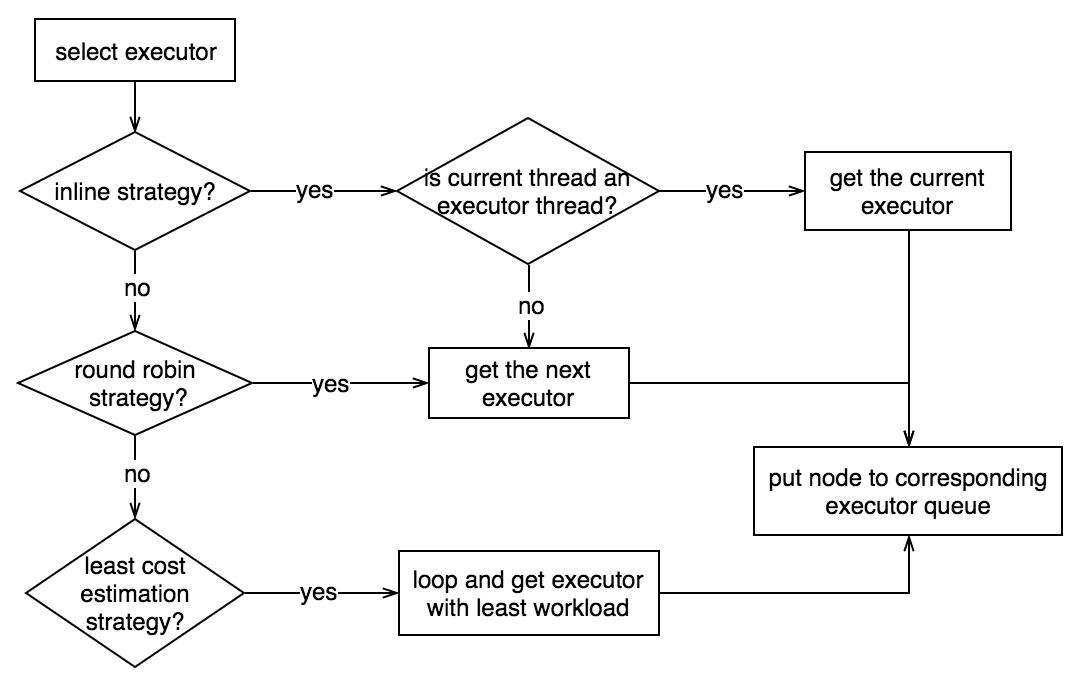


图3. 12 调度策略流程图

### 失败处理策略

除了调度结点的执行外，调度器模块还需处理结点执行失败的情形。图3.13显示了该工作流框架中所采取的结点失败处理策略。

通过给每一个结点增加一个IgnorableIfFailure的标志，若该标志为TRUE，则忽略该次失败，继续调度执行其下游结点，而若该标志为FALSE，则不再继续调度，而是将失败结点的下游结点都置为FAILURE并进行相应下游结点的失败处理，更下游的结点是否会得到调度执行则取决于当前失败结点的直接下游结点的IgnorableIfFailure标志值。

利用此种失败处理策略，可以很灵活地设置具体工作流的执行行为。例如可以将工作流中非关键路径上的结点的IgnorableIfFailure标志值设置为TRUE，则即使这些结点执行失败，也不会影响到工作流中其他结点的调度执行。

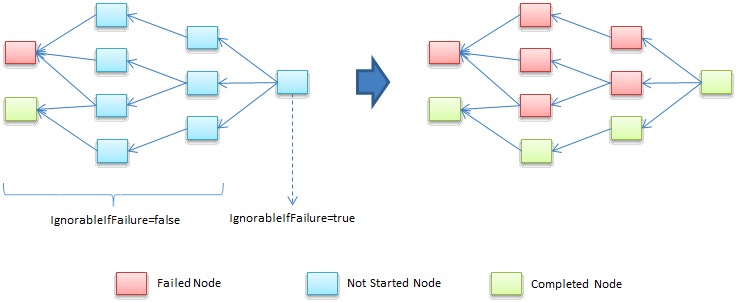


图3. 13 结点失败处理策略

## LnP模块详细设计

本节主要阐述了LnP模块的详细设计，包括该模块的类图、流程图和执行顺序图。

### LnP模块类图

LnP模块作为整个异步工作流框架中一个相对比较独立的模块，在保证开发过程中框架的运行性能方面起到了很关键的作用。图3.14显示了LnP模块中主要类的类图。

整个LnP模块由相关脚本驱动，具体测试工作流的生成、构建、运行和性能指标统计都由TestDriver类来驱动，而e-mail报告的发出则由OutputDriver类驱动。另外，关于LnP模块实际运行时该采用何种工作流图、如何创建工作流中的结点以及e-mail的接收者等，这些信息都包含在相关的配置文件中。TestSpec接口则是对这些相关配置的一个抽象，此种设计使得LnP模块在实现上更加灵活。MetricSet类使用了Yammer Metrics，以实现在测试工作流的运行过程中统计相关的性能指标，并最终显示在发出的e-mail报告中。EmailOutput类则是具体实现e-mail报告发送的类，其采用了FreeMarker以设计出结构更清晰易读的e-mail报告格式。

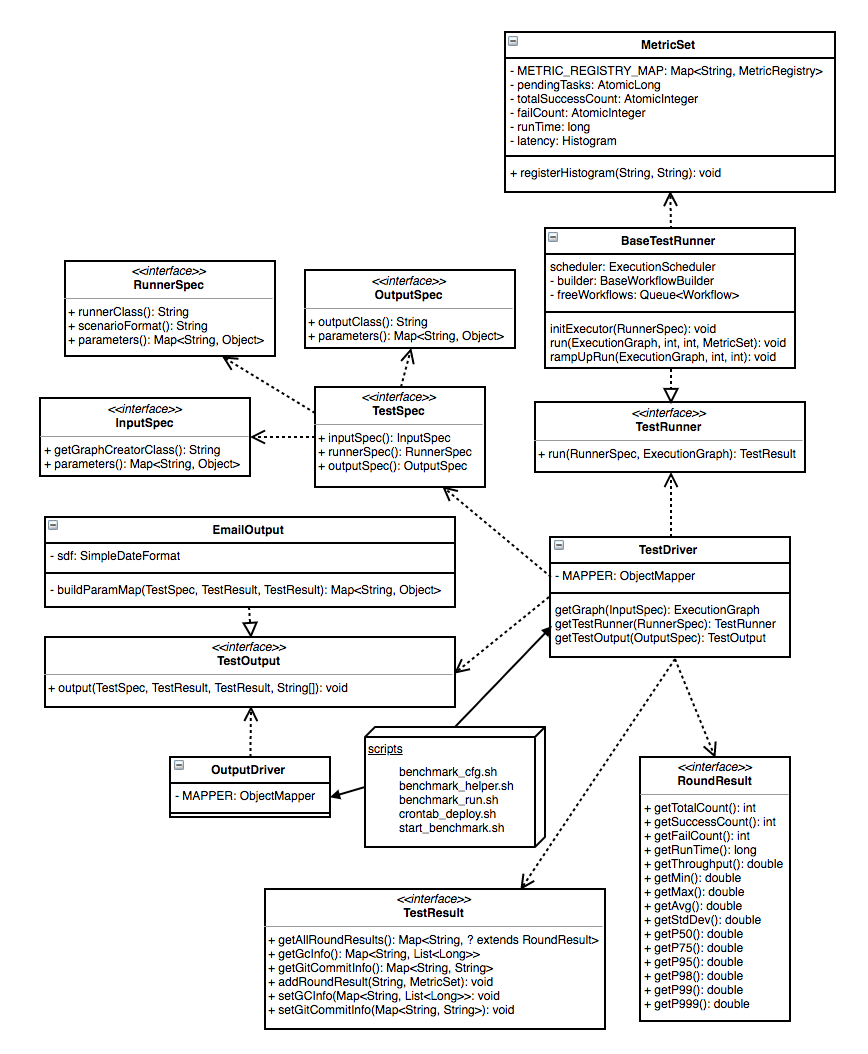


图3. 14 LnP模块主要类类图

上面提到，LnP模块具体运行的工作流图结构是由相关的配置文件来决定的。而根据不同配置文件中相关配置的不同，该模块在构建目标测试工作流和其中结点时的实现也有所不同。图3.15显示了LnP模块中与构建测试用工作流相关类的类图。

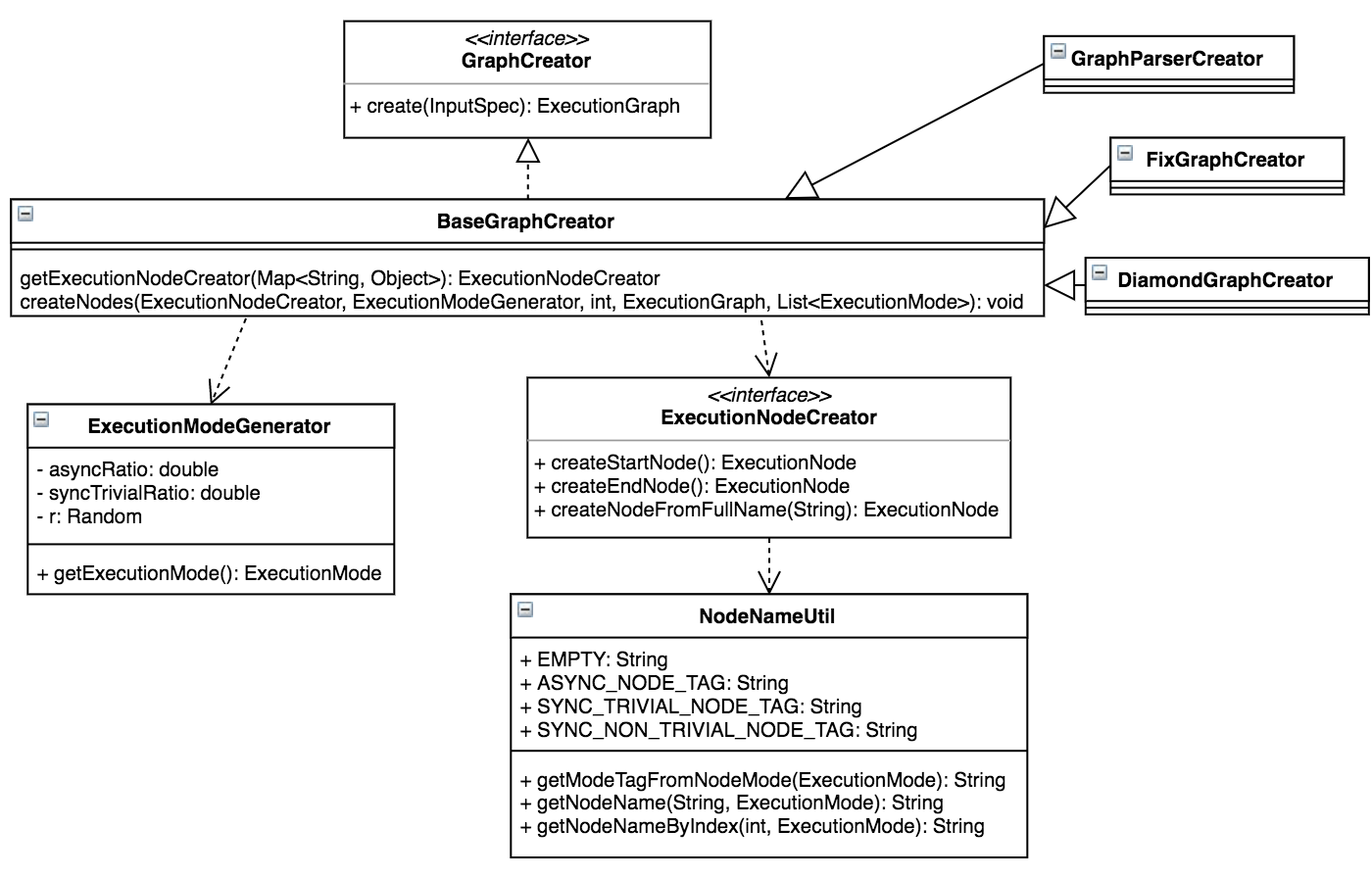


图3. 15 LnP模块构建工作流相关类类图

其中，BaseGraphCreator类实现了各个工作流图构建类所需的通用方法，如根据指定的结点构造器生成工作流结点等。该类有三个子类，分别实现了三种不同的工作流图构建策略，以供不同的应用场景使用。

GraphParserCreator类实现从特定格式的文本文件中解析并构造出工作流图。此种实现可以用于构造出与真实生产环境中完全相同的工作流图结构，从而可以将真实的工作流在该异步框架中的运行性能与其在原来老的框架中的运行性能进行比较，以验证采用了异步方案和多种特殊的巧妙设计后该框架对工作流执行性能的提升。

FixGraphCreator类则实现了根据配置的总结点数量，构造出一棵完全二叉树形状的工作流图。此种实现构造出来的工作流图较为简单，运行起来也比较快，整体的系统开销不会很大。它可用于单纯地测试框架对工作流调度执行的实现，对具体的调度和执行策略的变更比较敏感，因此很适合用来比较每一次代码变更对该异步工作流框架运行性能的影响。

DiamondGraphCreator类实现了根据给定的结点数、目标工作流层数和每个结点的最少子结点个数构造出一个菱形结构的工作流图。此种构造方式相较于简单的二叉树形式工作流图的构建更为复杂，支持通过不同的参数配置构造出复杂度不同的工作流图，因此可以更容易地比较并测试如结点数、结点边数等不同因素对框架整体运行性能的影响，从而更易于发现整个框架的性能瓶颈所在，并提前做出相应的优化。

### LnP模块流程图

LnP模块的主要目的是检测每一次代码变更对框架整体运行性能的影响，因此在每一次代码变更发生时，都需要运行此模块，执行相关的基准测试，收集测试工作流运行过程中的各项性能指标并生成相应的e-mail性能报告，发送给框架开发人员。此种需求使得LnP模块基准测试运行的自动化变得非常有必要，如此便可以降低开发人员的工作负担。在实际的实现上该模块采取了与Jenkins进行集成的设计方案，在每一次CI构建成功之后，自动调用LnP模块的相关脚本触发相应基准测试的执行。此种方案相较于通过简单的crontab任务去定期检查远程代码仓库是否有代码变更并触发LnP基准测试的运行方案相比，更加的方便，且可以实现更好的实时性。同时，通过将LnP的运行结果与CI构建的结果相关联，当发生运行错误时，也可以更便捷地通知到相关的框架开发人员。图3.16显示了LnP模块与Jenkins集成后的运行流程图。

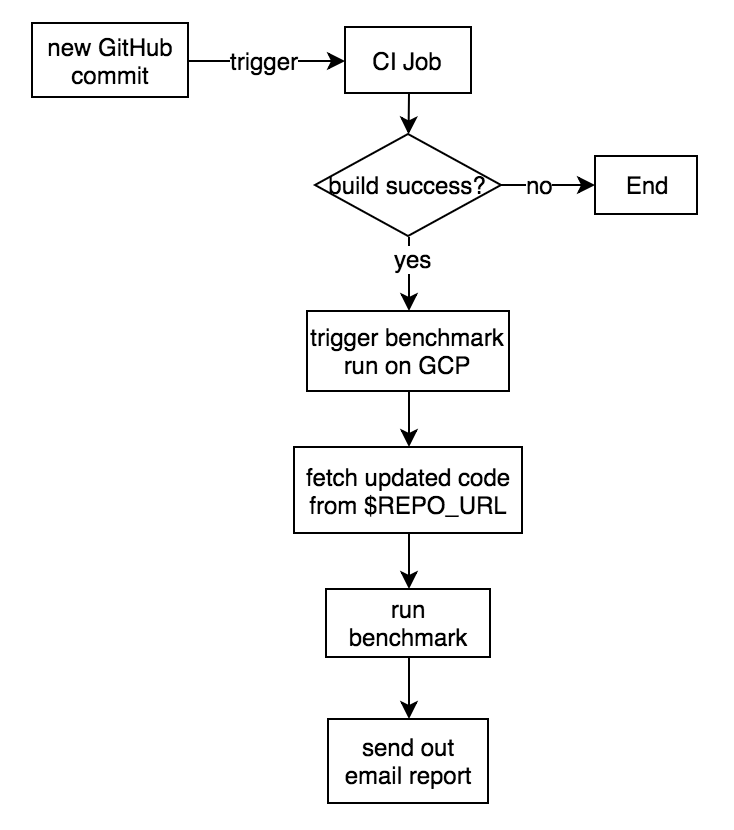


图3. 16 LnP模块流程图

### LnP模块顺序图

图3.17显示了LnP模块的运行顺序图。

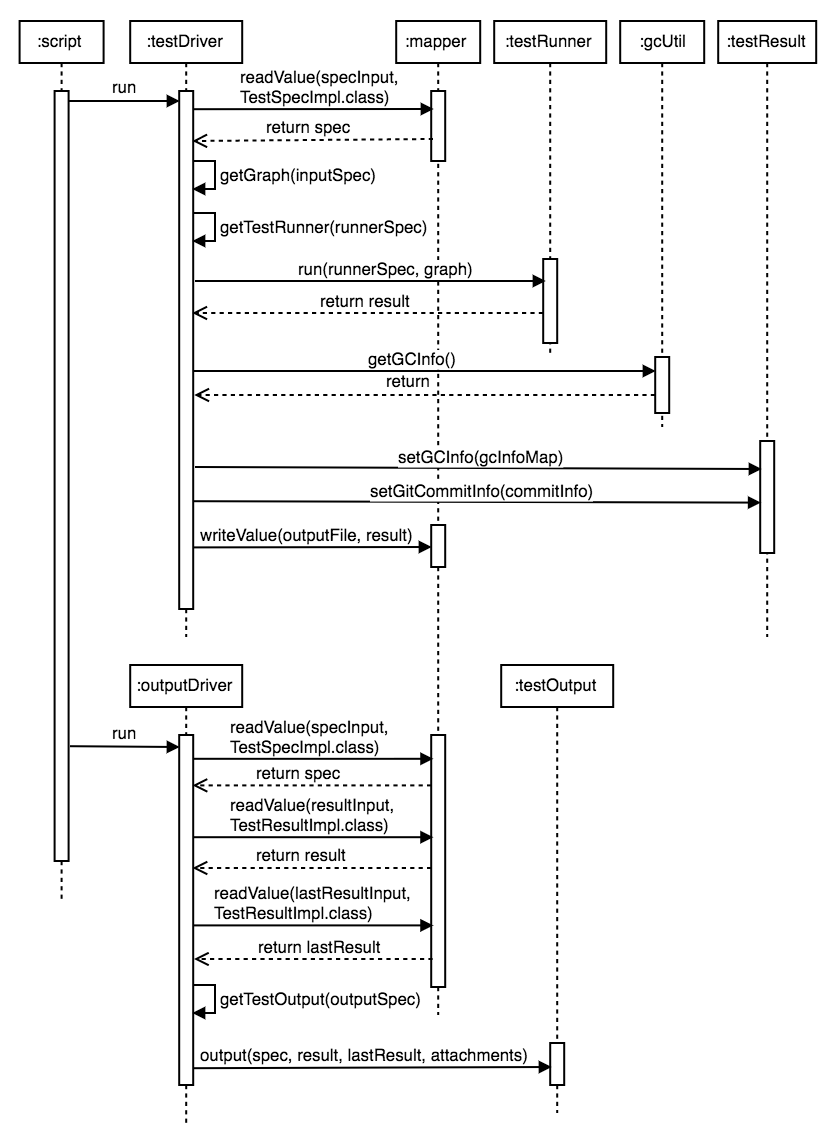


图3. 17 LnP模块运行顺序图

首先，LnP自动化脚本会触发TestDriver类运行，该类在解析完配置文件后调用TestRunner类，其会根据不同的配置值构建工作流图和工作流结点，并实际运行构建后的工作流，记录其运行时的性能数据信息。在运行完成后，TestDriver类会将系统垃圾回收信息和触发此次LnP的相关代码更新信息记录在运行结果中。随后，LnP自动化脚本会触发OutputDriver类的执行，其会根据相关配置文件的内容和工作流运行结果生成e-mail报告并发送给相关的框架开发人员。

## 本章小结

本章主要对异步工作流框架的需求和整体设计进行了分析和描述。首先，介绍了工作流执行的生命周期过程。而后，分别从系统的功能性需求和非功能性需求两个角度出发，对本异步工作流框架的需求进行了详细的分析。在系统设计部分，首先对整个系统的模块划分进行了一个总体的介绍，然后挑选了执行器Executor模块、调度器Scheduler模块和LnP模块等三个核心模块对其具体设计进行了详细的描述。

# 异步工作流框架的实现

## 引言

本章将在第三章对本异步工作流框架中三大核心模块（执行器Executor模块、调度器Scheduler模块以及LnP模块）详细设计介绍的基础上，详细阐述各个模块的具体实现细节。

同时，本章还将包含该框架在性能上实现的一些优化以及实际运行的相关性能数据。

## 执行器Executor模块实现

本节主要阐述了执行器Executor模块的具体实现，包括Trivial结点执行器和Non-Trivial结点执行器的实现，以及结点任务队列的实现。

### Trivial结点执行器实现

本异步工作流框架基于一个假设，也即待执行工作流中的大多数结点都属于计算型结点，它们的预计执行时间都很短，比如小于1毫秒，这些结点被称为Trivial结点。针对这些结点的执行，该模块使用了一些特殊的设计，通过利用线程本地的、Lock-Free的任务队列和特殊的等待策略，以实现在减少系统开销的同时提高框架运行的性能。

图4.1显示了Trivial结点执行器的具体实现方式。在Trivial结点执行器的实现中，采用了Netty提供的EventLoop接口的相关实现类，通过直接利用其内置的线程模型来实现具体结点执行任务的提交。同时，EventLoop中的每一个执行器线程都有一个属于自己的任务队列，用于存储该执行器线程待执行的任务。此任务队列的具体实现采用的是JCTools中的MpscArrayQueue实现，这个队列是一个多生产者单消费者的Lock-Free队列，可以降低并发情况下的锁同步开销，减少运行过程中系统上下文切换的次数，从而提高工作流中结点的执行效率。

|  |
| --- |
| eventLoop.submit(new AbstractTaskRunnable() {  public void run() {  int idleCount = 0;  while (true) {  // 省略异常处理代码  ExecutionTask task = executor.getExecutionQueue().pop();  if (task == null) {  if (idleCount >= MAX\_IDLE\_COUNT\_BEFORE\_SLEEP) {  idleCount = 0;  Thread.sleep(1);  } else if (idleCount >= MAX\_IDLE\_COUNT\_BEFORE\_YIELD) {  idleCount++;  Thread.yield();  } else {  idleCount++;  }  continue;  }  idleCount = 0;  executor.run(task);  }  }  }) |

图4. 1 Trivial结点执行器实现

同时，与Disruptor的SleepingWaitStrategy相类似，为了避免执行器线程在其任务队列为空时持续空转，白白浪费系统CPU资源，该模块在实现上采用了为每一个执行器线程维护一个idleCount计数的方案，该计数用于保存执行器线程从队列中获取待执行任务过程中持续空转的次数。若该计数大于某一值，则会调用Thread.yield方法，尝试让出CPU资源，但由于yield并不确保一定会出让CPU，若当前无其他线程竞争CPU资源或执行器线程的优先级较高，还是可能导致线程立刻被操作系统重新调度回来，无法真正实现CPU资源使用率的降低[Java-Yield,2016]。因此，为了避免此种情况的发送，在idleCount计数到达yield阈值时，将继续进行累加，若其超过了sleep阈值，则将调用Thread.sleep方法强制当前执行器线程让出CPU资源。测试结果表明，采用sleep和yield相结合的等待策略比仅采用yield的策略可以在系统负载较小时显著的降低CPU使用率（在16核的机器上配置12个Trivial执行器线程运行LnP 基准测试时，CPU利用率从70%左右降到了15%左右），同时在系统负载较大时不影响总体的吞吐量。

### Non-Trivial结点执行器实现

基于4.1.1节中所述相同的假设（待执行工作流中的大多数结点都属于计算型结点，它们的预计执行时间都很短），Non-Trivial结点是执行时间较长的结点，比如大于1毫秒，且其在工作流中的数量会远少于Trivial结点的数量（在本人实习企业内所使用的一个真实业务工作流中，Trivial结点数量与Non-Trivial结点数量的比值大约为9：1）。

因此，对于此类结点，若采用与Trivial结点执行器一样的实现策略，则会导致执行器线程空转的次数非常多，造成CPU资源的浪费，且一个Non-Trivial结点的执行会严重影响该执行器线程队列中后续结点的执行。于是，在具体实现上，Non-Trivial结点执行器同样利用了Netty的EventLoop接口，但却采用了阻塞队列的实现方案，以便在得不到待执行结点时可以及时地让出CPU资源，从而让其他可执行线程尽快得到执行。同时，在实际运行时，Non-Trivial结点执行器的线程数量应配置得比Trivial结点执行器的线程数量多，以支持更大的并发能力。

图4.2展示了Non-Trivial结点执行器的关键代码实现。

|  |
| --- |
| // 省略EventLoop提交任务代码和相关异常处理代码  while (true) {  ExecutionTaskBlockingQueue queue = executor.getExecutionQueue();  ExecutionTask task = queue.pop(BLOCKING\_QUEUE\_WAIT\_MS, TimeUnit.MILLISECONDS);  if (task != null) {  executor.run(task);  }  } |

图4. 2 Non-Trivial结点执行器实现

### 结点任务队列实现

不论是Trivial结点的执行器，还是Non-Trivial结点的执行器，其采用的都是同一个结点任务队列实现基类。这个基类在底层具体使用的Lock-Free队列或阻塞队列上进行了一层封装，既统一了对外接口，又可以实现在每一次入队和出队操作之后更新当前执行器线程任务队列的工作负载。此项工作负载的信息可以供特定的调度器在调度时使用，以实现更高效的结点执行和调度。图4.3中所示代码片段即展现了执行器模块中与记录执行器线程任务队列工作负载相关的具体实现。

|  |
| --- |
| protected final AtomicLong currentWorkload = new AtomicLong(0);  // 该方法在入队操作后调用  protected void onPut(ExecutionTask task) {  currentWorkload.addAndGet(task.getExecutionNode().getEstimatedCostInNs());  }  // 该方法在出队操作后调用  protected void onPop(ExecutionTask task) {  if (task != null) {  currentWorkload.accumulateAndGet(  -task.getExecutionNode().getEstimatedCostInNs(), (x, y) -> {  if (x + y <= 0 || getTaskCount() == 0) {  return 0;  }  return x + y;  });  }  } |

图4. 3 执行器队列工作负载统计的实现

注意在执行出队操作后对执行器工作负载的更新实现，并不是仅仅将出队结点的预计执行时间减掉即可。原因在于，本异步工作流框架是支持结点预计执行时间在运行时改变的，这样设计可以实现在同一工作流多次运行后，结点的预计执行时间会愈来愈接近其真实的执行时间，从而使得工作流的调度更加准确和高效。于是，对于此种特殊设计，为了避免执行器工作负载变为负值（结点预计执行时间在运行时增大），或者当执行器线程队列为空时工作负载仍大于0（结点预计执行时间在运行时减小），需要在结点出队时，对相应执行器工作负载的更新进行特殊实现。

另外，在本项目的具体开发过程中，曾遇到过这样一个有趣的问题。最开始，执行器工作负载累加器使用的是java.util.concurrent.atomic类库中提供的LongAccumulator类实现，结果在运行相关LnP基准测试的过程中却发现LeastCostEstimationScheduler调度器的表现很差。在增加框架运行时日志输出并分析之后发现，在仅采用两个Trivial结点执行器线程的情况下，least cost调度器在调度时总是选择同一个执行器线程，而另一个不被选择的执行器线程，其工作负载非常高，且一直维持着这一很高的工作负载不再改变。在进一步输出调度时各个Trivial执行器线程的待执行任务队列的大小时，发现有着很高工作负载的那个执行器线程的队列大小居然为0，也就是说它根本没有任务需要执行，一直在空转。而且，该问题的出现是时有时无的。因此，联想到是不是多线程并发产生的问题，而工作负载累加器的累加方法是正确无误的，于是便怀疑LongAccumulator类在实现上是否存在多线程并发相关问题。在经过一番搜索后，发现LongAccumulator类在JDK8中的实现确实存在bug，直到JDK10中才被修复[JavaBugDb,2017]。该类在多线程中实现累加时的顺序是不确定的，而执行器工作负载累加器在有结点出队时的更新操作跟累加的顺序十分相关，因此这个bug就可能导致一个实际为空的队列却有着很高的工作负载，图4.4模拟了这种情形。

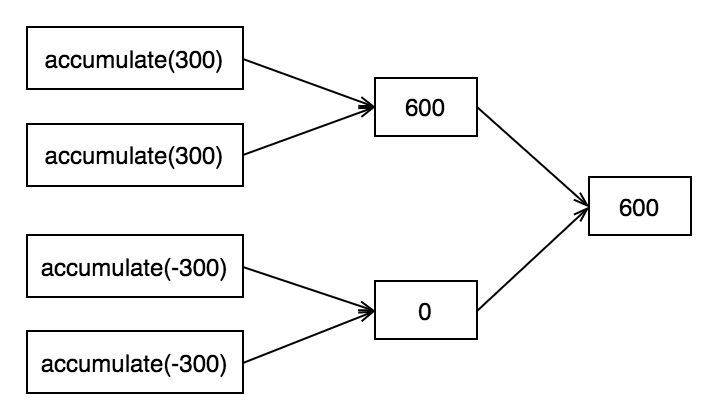


图4. 4 LongAccumulator类多线程并发累加错误

在把执行器工作负载累加器的实现从LongAccumulator类换成同一类库中的AtomicLong类后一切正常，使用LeastCostEstimationScheduler调度器进行工作流执行调度时的性能也随之提高。

## 调度器Scheduler模块实现

本节主要阐述了调度器Scheduler模块的具体实现，包括调度下游结点执行的实现和least cost调度器的实现。

### 调度下游节点执行实现

具体工作流的执行过程总是从调度器调度其起始结点执行开始的，在起始结点执行结束后，执行器线程会通知调度器调度执行其下游结点，此过程会循环发生，直到工作流的终止结点执行完成。而不论采用何种调度策略，在获取当前结点的下游结点并检查这些下游结点的就绪状态的实现上都是一样的。图4.5所示的代码片段展示了调度器模块中的这一通用逻辑。

|  |
| --- |
| void scheduleDownstreamNodes(ExecutionNode currentNode, ExecutionContext ctx) {  for (DAGNode n : currentNode.getDownStreamNodes().values()) {  final ExecutionNode node = (ExecutionNode) n;  ctx.getExecutionNodeContext(node).increaseCompletedUpstreamCount();  if (node.isReadyToExecute(ctx)) {  schedule(node, ctx);  }  }  } |

图4. 5 调度器模块调度下游结点执行实现

工作流中的每个执行结点都会有一个与其相对应的ExecutionNodeContext对象，该结点上下文对象用于存储当前结点的预计执行时间、当前状态、失败是否可忽略等信息。同时，该结点上下文对象中还维护了一个AtomicInteger类型的计数器，用于记录当前结点已完成的上游结点个数。在一个结点执行完毕后，调度器在调度其下游结点执行之前，会先将当前结点的所有下游结点的相应计数器的值加1，然后再检查其下游结点的就绪状态。若下游结点已就绪，则调度其执行，具体如何调度结点执行则根据不同调度策略有不同的实现。

图4.6展示了判断结点是否就绪的具体实现逻辑。

首先，会检查结点的当前状态是否为NOT\_STARTED，如果不是，则立即返回FALSE，从而保证在一次工作流执行过程中每个结点都只会被调度执行一次。然后，根据不同的调度策略，具体判断结点当前是否可以被调度执行。如果是采用了ANY的调度策略，即只要任意一个上游结点完成，该结点就可以被调度执行，则直接返回TRUE。因为在正确的实现下，该方法只应该在调度器调度下游结点执行时才会被调用。而如果采用的是ALL的调度策略，即只有所有的上游结点均完成了，当前结点才可以被调度执行，此种情况下则需检查该结点已完成的上游结点计数器的值是否等于其所有的上游结点数，若是则说明结点就绪，返回TRUE，否则返回FALSE。

|  |
| --- |
| boolean isReadyToExecute(ExecutionContext context) {  ExecutionNodeContext nodeCtx = context.getExecutionNodeContext(this);  final ExecutionState state = nodeCtx.getExecutionState();  if (!ExecutionState.NOT\_STARTED.equals(state))  return false;  final ExecutionStrategy strategy = this.getExecutionStrategy();  if (ExecutionStrategy.ANY.equals(strategy))  return true;  if (ExecutionStrategy.ALL.equals(strategy)  && nodeCtx.getCompletedUpstreamCount() != this.getUpStreamNodes().size())  return false;  return ExecutionStrategy.ALL.equals(strategy);  } |

图4. 6 检查结点就绪状态实现

此种基于已完成上游结点计数器的实现方案实际上是通过LnP模块基准测试调优之后的实现，图4.7中所示的部分代码片段展示了检测结点就绪状态方法在调优之前的实现方案。

|  |
| --- |
| // isReadyToExecute方法，省略方法前半段与图4.6中代码重复的部分  if (ExecutionStrategy.ALL.equals(strategy)) {  for (ExecutionNode node : getUpStreamNodes().values()) {  if (!node.isCompleted(context)) {  return false;  }  }  }  return ExecutionStrategy.ALL.equals(strategy); |

图4. 7 检测结点就绪状态调优之前的实现

从图4.6和图4.7中的代码片段可以看出，在之前的实现方案中，在判断当前结点是否就绪时，需要循环判断其所有上游结点的完成状态，而其每一个上游结点执行完成时都会调用当前下游结点的这一小段代码。因此，如果工作流中结点的依赖关系比较复杂，结点之间的边数比较多时，就可能会导致以上代码循环被反复执行非常多次，这样带来的延时开销是很大的。

在经过分析调优后，将图4.7中代码片段优化成了图4.6中展示的实现，通过使用已完成上游结点计数器，彻底去掉了这层循环检测。图4.8展示了此次优化所带来的框架性能提升，横轴表示工作流图中每个结点的最少直接下游结点个数，纵轴则表示工作流执行的整体延时。从图中可以看出，当结点的直接下游结点个数增多时，此次优化可以带来很显著的性能提升。如当每个结点的子结点个数达到100个时，99分位的系统延迟从1000+毫秒降到了100毫秒左右，性能提升了近90%。

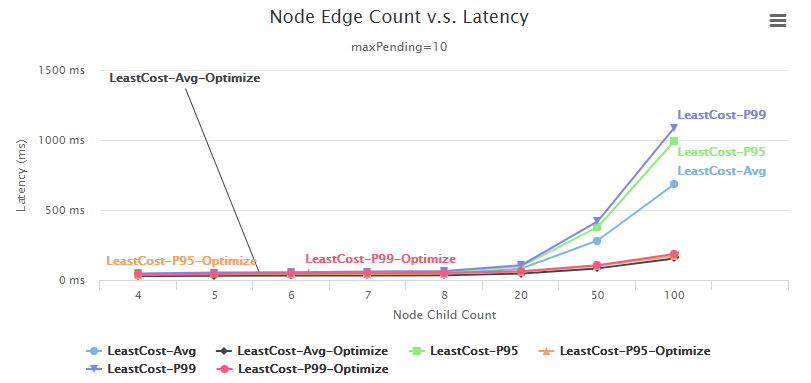


图4. 8 检查结点就绪状态实现性能提升

### Least Cost调度器实现

目前调度器模块中共有三种不同的调度策略实现：Inline调度策略、RoundRobin调度策略和LeastCostEstimation调度策略。

Inline调度器会优先选择当前正在运行的执行器线程执行被调度结点，若当前运行线程不是一个执行器线程，则退化为使用RoundRobin调度器。

RoundRobin调度器不考虑任何其他因素，其在内部维护了一个执行器数组的下标引用，该引用指向上一次被选择的执行器线程，在每次调度时，其将相应的下标引用加1并返回其所指向的执行器线程即可。

LeastCostEstimation调度器则不一样，它在调度时将各个执行器线程的当前工作负载考虑在内，总是优先选择工作负载最小的执行器线程，从而可以使被调度结点更快的得到执行，进而提升框架整体运行性能。图4.9展示了这一调度器的具体实现。

LeastCostEstimation调度器在选择执行器线程时，首先需要判断待调度结点的类型是Trivial还是Non-Trivial的，这个是由结点的预计执行时间所决定的，若其预计执行时间大于设定的某一阈值（默认设定为1毫秒），则为Non-Trivial结点，否则为Trivial结点。然后，根据不同的结点类型，选择相应的执行器线程组，并根据该线程组中各个执行器线程的当前工作负载，选择出负载最小的那一个，用于执行当前调度结点。

|  |
| --- |
| // 根据结点类型（Trivial or Non-Trivial）选择不同的执行器线程组  DAGNodeExecutor selectExecutor(ExecutionNode node, ExecutionContext ctx) {  // 将结点的预计执行时间与配置的TRIVIAL\_THRESHOLD比较，判断其类型  final ExecutionMode mode = rule.determineExecuteMode(node, ctx);  if (ExecutorMode.TRIVIAL.equals(mode)) {  return selectLeastWorkloadExecutor(trivialExecutorGroup);  }  return selectLeastWorkloadExecutor(nonTrivialExecutorGroup);  }  // 在给定的执行器线程组中选择当前工作负载最小的执行器线程  DAGNodeExecutor selectLeastWorkloadExecutor(DAGNodeExecutorGroup group) {  // for loop the executor group and find the executor with least workload  // skip…  } |

图4. 9 LeastCostEstimation调度器实现

## LnP模块实现

本节主要阐述了LnP模块的具体实现，包括该模块配置文件示例、驱动脚本的实现、基准测试运行性能指标统计的实现以及工作流图构建的实现。

### LnP配置文件示例

图4.10展示了一个LnP模块的示例配置文件，因保密性等相关因素，隐掉了与企业名称相关的一些字段值。整个LnP模块就是通过读取并解析这个配置文件，来确定如何构建工作流图以及其中的结点、采用何种调度策略、具体如何执行基准测试等信息的。

该配置可以分为相对独立的三个部分，input配置块、runner配置块和output配置块。

input配置块主要用于指明该如何构建工作流图以及如何创建工作流图中的结点。如果配置不同的工作流图构建类，其所需的具体参数也可能不同。示例中展示的是使用FixGraphCreator类来构建工作流图时所需指定的各项参数，而如果是使用DiamondGraphCreator类来构建具体的工作流图的话，除了同样需要指定各个不同类型结点的出现比率以及结点构建类等信息之外，还需要额外说明总结点数、每个结点的最少直接下游结点个数、工作流图的层数等相关信息。

|  |
| --- |
| {  “inputSpec” : {  “graphCreatorClass” : “xxx.xxx.workflow.benchmark.FixGraphCreator”,  “parameters” : {  “nodeCount” : 10000,  “asyncRatio” : 0.0,  “syncTrivialRatio” : 1.0,  “divergeCount” : 2,  “nodeCreatorClass” : “xxx.xxx.workflow.benchmark.ExecutionNodeCreatorImpl”  }  },  “runnerSpec” : {  “runnerClass” : “xxx.xxx.workflow.benchmark.SimpleTestRunner”,  “scenarioFormat” : “[Scheduler Name]-[Max Pendings Allowed]”,  “parameters” : {  “trivialExecutorCount” : 12,  “nonTrivialExecutorCount” : 0,  “scheduleClassFullName” : [xxx.xxx.workflow.scheduler.LeastCostEstimationScheduler”],  “maxPendingRound” : “1-10”,  “totalRound” : 1000,  “rampUpRound” : 1000  }  },  “outputSpec” : {  “outputClass” : “xxx.xxx.workflow.benchmark.EmailOutput”,  “parameters” : {  “smtpHost” : “xxx.xxx.com”,  “smtpPort” : 25,  “mailFrom” : “yefu@xxx.com”,  “mailTo" : “DL-xxx-xxx-Async-SCRUM@xxx.com”,  “mailTemplate” : “EmailTemplate.ftl”  }  }  } |

图4. 10 LnP配置文件示例

runner配置块用于指定以何种方式去执行构建出来的工作流图，包括Trivial和Non-Trivial结点执行器的个数、采用的具体调度器实现、框架运行时所允许的最大并发数以及运行的总轮次等信息。

output配置块则用于指定输出基准测试执行结果的方式。目前仅支持通过e-mail的形式输出，其使用的是EmailOutput类，需要指定的具体参数包括SMTP服务器的相关信息，e-mail的发送者和接收者地址，以及e-mail内容格式的模板等。

图4.11显示了一份真实的e-mail报告示例（隐去了企业名称相关信息），它是实际在框架开发过程中用于告诉开发人员每一次代码更新对框架性能影响的报告，由Jenkins在每次CI构建完成后自动触发LnP模块执行并发出。

该报告主体可以分为四个部分。

第一部分给出了触发此次LnP运行的GitHub代码提交信息和相关链接，因此若该报告显示框架运行性能较上次有显著下降，此部分信息可以使开发人员更便捷地定位到引起此次性能下降的具体代码变更，方便进行具体问题的定位；

第二部分指出了每一种不同的运行场景的命名规范，该部分信息来源于LnP配置文件；

第三部分展示了LnP配置文件中有关input和runner的各项配置信息；

第四部分则是整个基准测试报告的重点，包含了框架执行过程中垃圾回收和具体工作流运行延时等信息。对于配置文件中指定的每一个运行场景，报告中都给出了其总运行次数、运行成功次数、总运行时间、系统吞吐量以及运行延时的平均和各个具体分位值信息。同时，还包含此次运行结果与上一次基准测试运行结果的各项性能指标比较，很清晰直接地给出了此次代码变更对框架性能带来的影响。

另外，报告中还包含了两个附件，分别显示了工作流执行时所产生的系统调用统计（通过strace命令获得[strace, 2018]），以及CPU的使用率和上下文切换等相关信息（通过vmstat命令获得）。

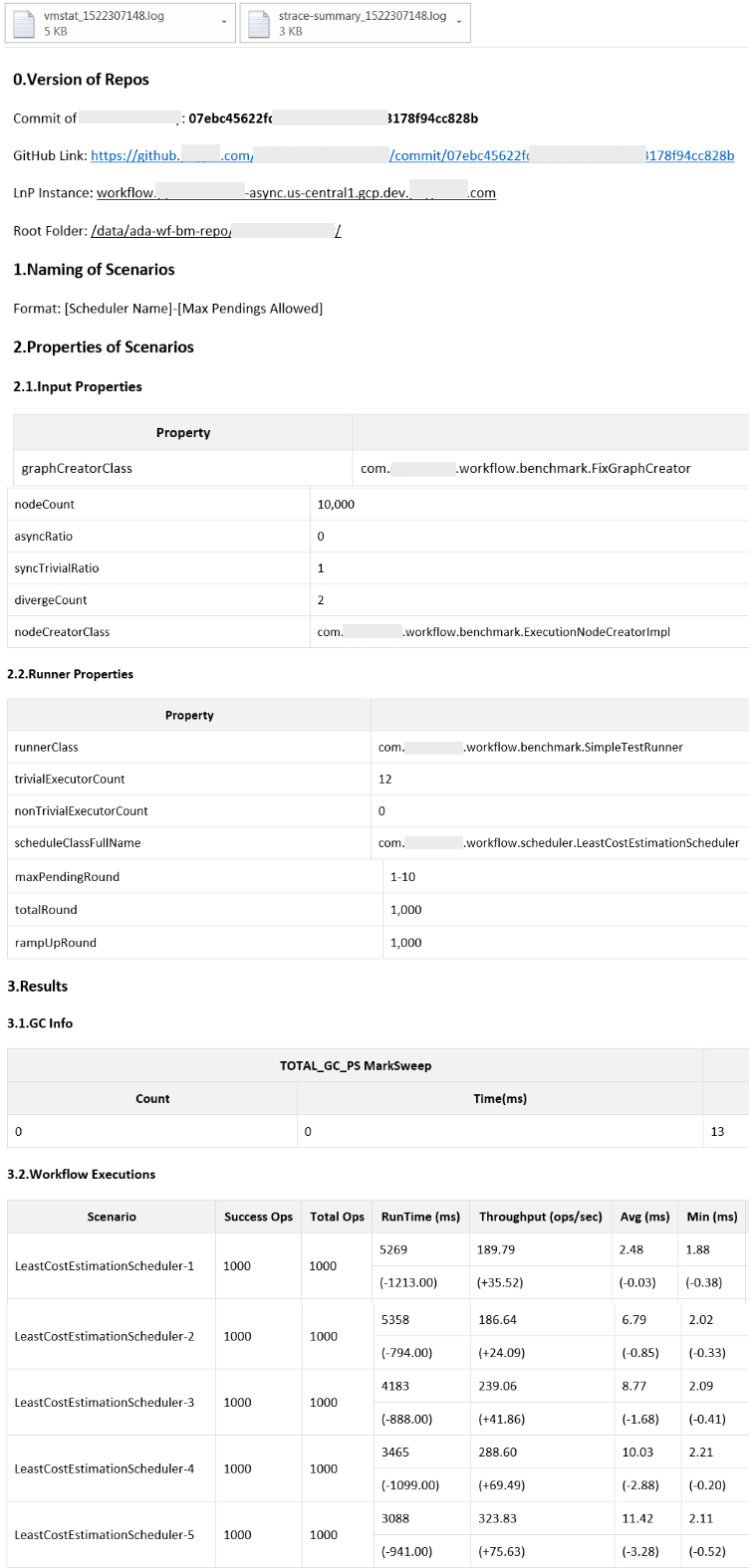


图4. 11 LnP模块基准测试报告示例

### LnP驱动脚本实现

图4.12显示了LnP模块驱动脚本的运行流程图。

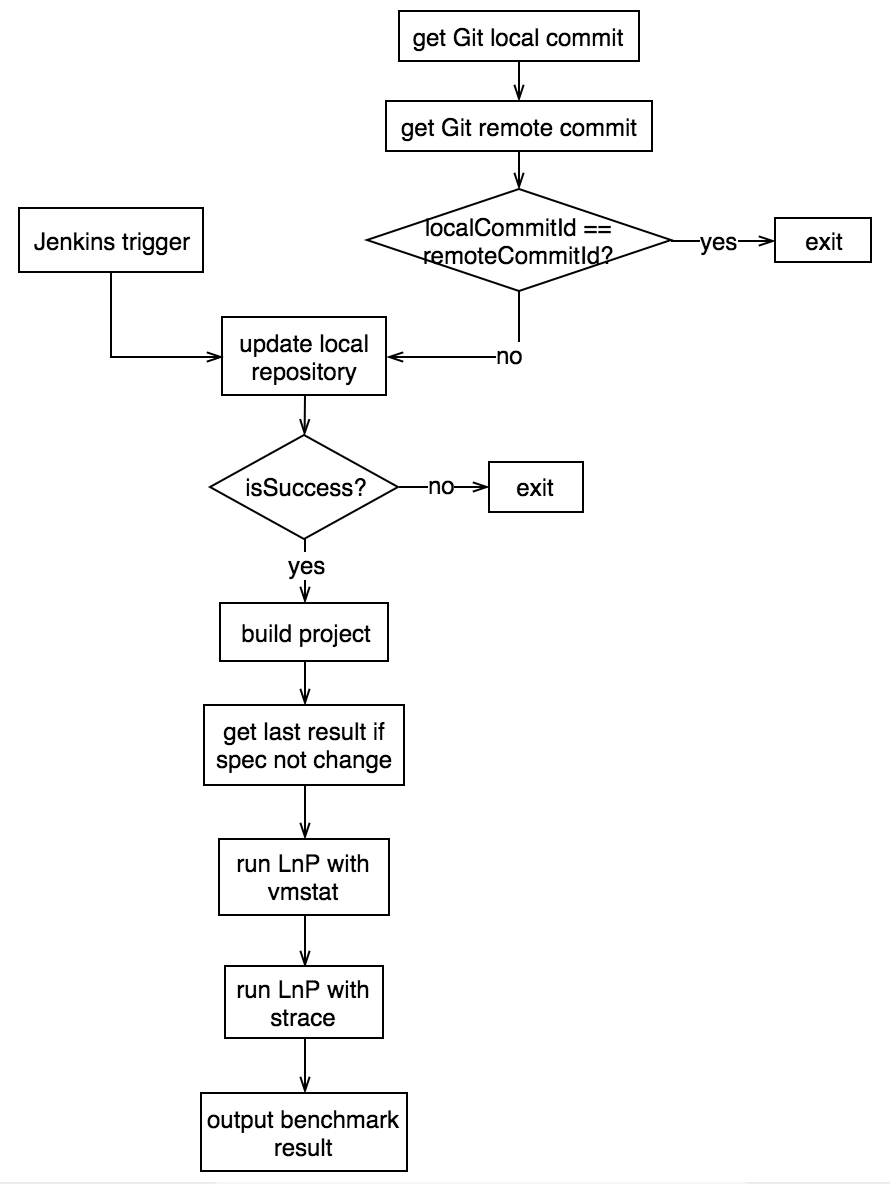


图4. 12 LnP驱动脚本详细流程图

有两种方式可以驱动LnP模块的运行，一种是由Jenkins在CI构建成功后自动触发，另一种是在本地手动触发。若是通过在本地手动触发的方式，则脚本会先获取本地和远程代码仓库的GitHub提交信息，只有在远程代码仓库有新的代码变更时才会触发后续操作，而如果是通过Jenkins自动触发的方式，则无需检测是否有新的代码变更，直接从远程代码仓库拉取并更新本地代码即可。若代码更新成功，则在本地执行maven构建并运行LnP基准测试。同时，LnP驱动脚本还支持根据参数中传递的具体GitHub提交ID信息将本地代码更新至指定的版本，便于重复测试指定代码变更对框架性能的影响。

在真正运行LnP基准测试之前，会先获取本地存储的上一次基准测试运行结果，用于与本次结果进行性能比较。同时，为了保证各项性能指标的可比性，需要检测本次运行与上次运行使用的LnP配置文件是否一致，这个一致性检测是通过MD5来实现的[Shotts et al., 2012]。实际的基准测试会执行两次，第一次与vmstat命令一起运行，以便记录框架运行过程中的上下文切换、CPU利用率等信息，第二次则与strace命令一起执行，用于记录系统调用的统计信息。之所以不能将这两次运行合并在一起，是因为strace带来的性能开销会比较大，它对于系统调用的统计是通过在每一次发生系统调用时中断应用程序来实现的[Gregg, 2013]，此种实现也会导致vmstat给出的上下文切换统计信息变得不准确，因此必须分为两次进行。

运行完成后，LnP脚本会将此次运行结果与上次运行结果作为参数传递给OutputDriver类用于输出e-mail报告。报告中采用的性能数据将是第一次运行的结果，也即与vmstat命令一起运行的结果，因为vmstat命令本身不会带来大的系统性能开销[vmstat, 2018]。

### 性能指标统计实现

要判断某一次代码变更对框架的性能是否有影响，就需要在工作流实际执行过程中统计一些可用于比较的性能指标，其中最主要的一项就是工作流的运行延时latency，其它一些还包括工作流执行的总成功和失败次数、系统吞吐量、垃圾回收时间以及一些操作系统层面的统计数据，如系统调用情况、CPU利用率等。另外，LnP模块在实际执行工作流时，对于同一个配置通常会执行多个轮次（通过totalRound参数配置），从而尽量减少其他外部因素对框架性能统计的干扰，以获得更为准确可信的性能指标值。

工作流的运行延时信息是通过Yammer Metrics提供的Histogram实现的，其可以展示统计指标的平均值、方差、最大最小值以及各个具体分位值的情况，从而使框架开发人员对延时信息有一个更全面清楚的认识，图4.13展示了其具体使用实现。在实际运行工作流之前，会注册一个新的Histogram用于记录当前工作流的运行延时信息。当每一次工作流执行完成后，其相应监听器中的onSuccess方法会计算此工作流的执行延时，并用此信息来更新相应的Histogram。最后在所有轮次都运行完毕后，直接从相应Histogram中获取具体的各项延时信息即可。

|  |
| --- |
| // in TestRunner, register the metrics before running workflow  MetricRegistry registry = new MetricRegistry();  Histogram latencySla = registry.histogram(histogramName + “-latencySla”);  // in TestCompletionListener, update the metrics when workflow success  Long runtime = getRunTimeUs(promise);  latencySla.update(runtime);  // in TestResult, get detailed latency info from the histogram  Snapshot snapshot = latencySla.getSnapshot();  snapshot.getMean();  // get all other info, such as min/max/99thPercentile etc. |

图4. 13 工作流运行延时统计实现

垃圾回收相关信息的统计是通过使用java.lang.management包中提供的ManagementFactory类来实现的，其getGarbageCollectorMXBeans方法可以获取系统运行过程中各个垃圾回收器运行的次数和总耗时[Meehan, 2006]。对于LnP模块来说，其主要是测试框架实现层面的性能，因此运行的工作流结点多是dummy结点，没有太多的实际行为，所以总体来说该模块的运行是CPU bound的，不会有太多的内存分配和回收动作。鉴于此特性，垃圾回收这一性能指标的参考价值将不如延时信息重要。

另外，操作系统层面的统计信息是通过执行vmstat、strace等Linux命令来获得的，具体实现在上一节中已提到。

### 工作流图构建实现

LnP模块主要提供了三种构建工作流图的方式，分别由FixGraphCreator类、GraphParserCreator类和DiamondGraphCreator类实现。

FixGraphCreator类实现了根据给定的结点总数构建一个完全二叉树形状的工作流图，此图的结构比较简单，每个结点最多只有两个直接下游结点，其图结构的简洁性使得它对框架的实现策略较为敏感，因此主要用于检测每次代码变更对框架运行性能的影响。

GraphParserCreator类实现从特定格式的文本文件中构建工作流图。通过从实际生产环境中将图结构dump下来，并用此类进行解析，便可以实现构建真正的业务工作流图，从而可以验证一些对复杂图结构的优化所能带来的框架执行性能提升，如Trivial结点聚合、对图进行多余边的剪枝等。

图4.14展示了一个目前LnP模块支持的用于描述工作流图结构的文本文件格式，其也可以通过使用Graphviz工具进行解析和可视化[Ellson et al., 2004]。

|  |
| --- |
| digraph G { // G代表该工作流图的名称  start -> a0; // 表示a0是start的直接下游结点  a0 -> a1;  a0 -> a2; // 同一结点的多个直接下游结点需分多行说明  // … 其他结点依赖关系  } |

图4. 14 digraph文件格式示例

DiamondGraphCreator类实现了根据给定的结点总数、层数和结点的最少直接下游结点个数构建一个菱形结构的工作流图。

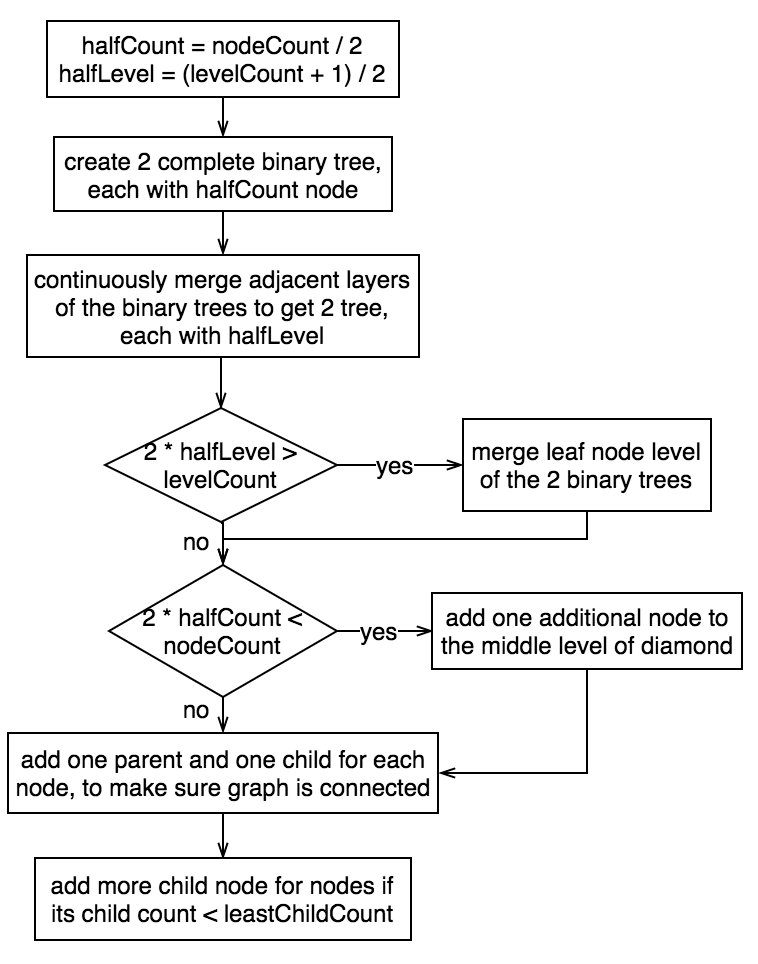


图4. 15 菱形工作流图构建流程实现

图4.15展示了该类构建菱形图的具体流程实现，结果菱形图实际上是由一正一反的两颗二叉树组成的，通过不断地合并二叉树图的相邻层以及改变中间层的结点个数以使得最终的结点总数和工作流图层数满足配置参数的要求。同时，为了保证最终工作流图的连通性，某些结点的直接下游结点个数可能会多于配置的最少直接下游结点数。

图4.16展示了实际构建出来的一个示例菱形工作流图（示例图使用参数为：结点总数=50，层数=7，最少直接下游结点数=2），其中最顶层和最底层分别为工作流的起始结点和终止结点，因此最终构建出来的工作流图为9层。

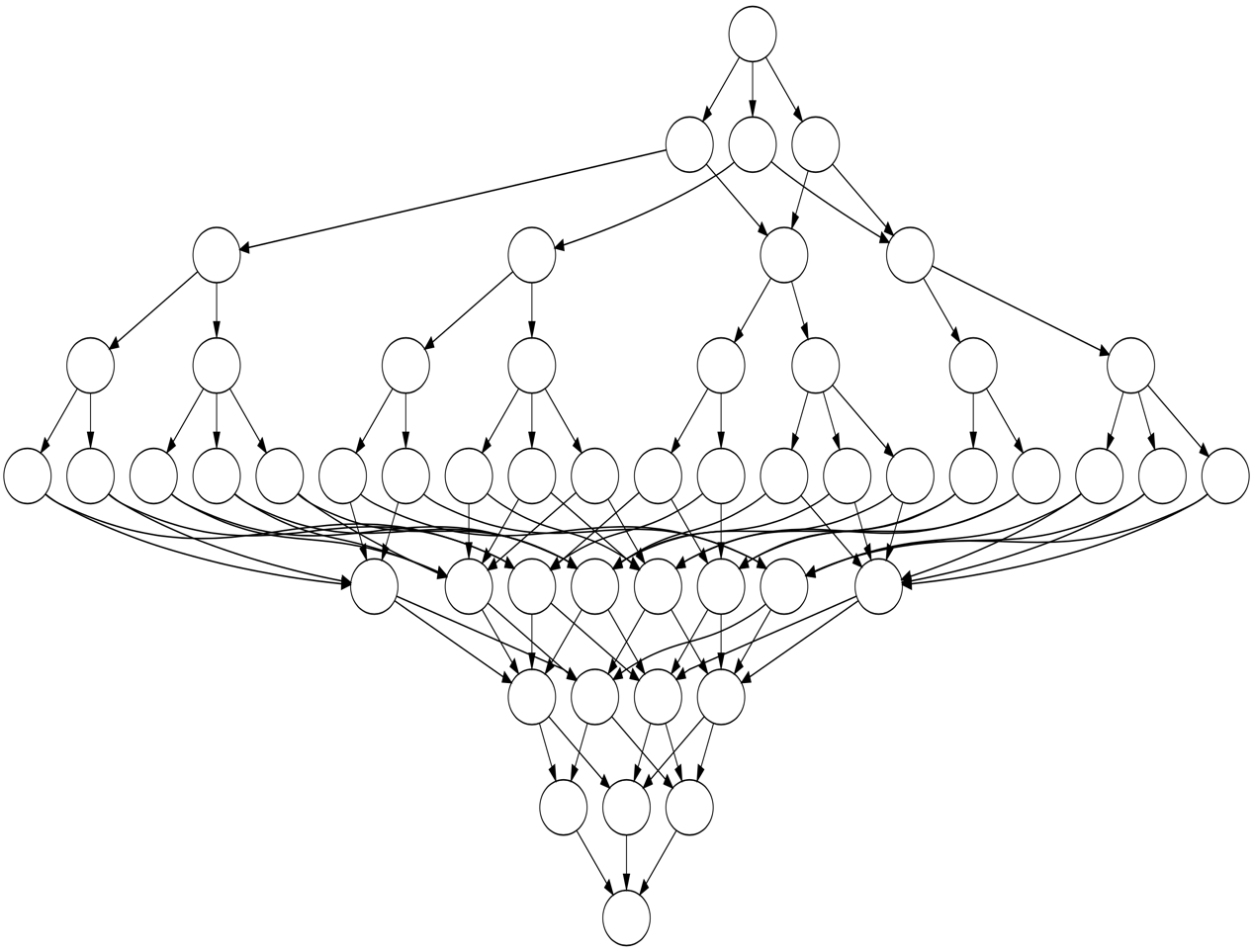


图4. 16 示例菱形工作流图

此种菱形结构的工作流图相较于简单的二叉树结构的图，其结构与实际的业务工作流图更为相似。另外，通过调整输入参数，可以很方便的改变结果菱形图的复杂度，如在给定结点总数和层数的情况下，增加工作流图中结点的边数等，从而能够测试不同因素对框架性能的影响，便于发现可能的性能瓶颈并及时做出相应的优化。4.2.1节给出了一个具体的优化示例，通过优化检测结点就绪状态的实现，大大降低了框架在工作流图中结点边数增多时性能下降的速度。

## 工作流图结构优化实现

图4.8中所示的折线图展现了工作流图中结点的直接下游结点个数，也即图中的边数增加时给工作流执行整体延时所带来的显著影响。因此，本异步工作流框架除了针对不同的结点类型和预计执行时间长短设计了专门的执行策略之外，还特别针对工作流图的结构实现了一系列的优化措施。这些优化措施可以静态作用于全局的执行DAG图上，通过简化工作流图的结构来达到其执行性能的提升。具体的优化措施包括多余边的剪枝、增加虚拟结点、聚合Trivial结点等。

工作流图中多余边的剪枝是基于Transitive Reduction算法实现的，图4.17展示了该算法的基本思想。对于一个有向无环图G，Transitive Reduction算法的目标是尽可能多地去除G中顶点之间的边，以产生一个新的有向无环图G’，同时保证消除后的图G’与原始图G有相同的连通性，也即若图G中存在顶点A到顶点B的一条边，则在图G’中也必然存在顶点A到顶点B的一条边[Transitive\_reduction,2018]。

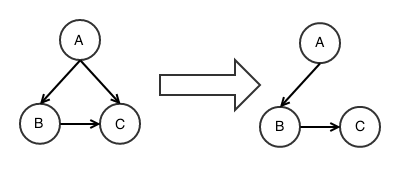


图4. 17 Transitive Reduction算法基本思想

如图4.17所示，顶点A到顶点C之间的边被去除，因为它们之间还有一条非直接路径A->B->C。具体到工作流的执行上，结点C要求结点A和结点B均执行完成后才可被调度执行，因此该条边的消除可以省去结点A执行完成后调度器检测结点C就绪状态的开销。若工作流图中此类冗余边较多，那么通过Transitive Reduction算法优化后，可以显著降低系统的总体延时，从而提升运行性能。

图4.18所示代码片段显示了本异步工作流框架中此项优化的具体实现方案。首先，使用深度优先搜索算法[Cormen, 2009]统计工作流图中各个顶点对之间的最长路径长度。然后，遍历此最长路径长度集合中的所有顶点对，若顶点A和顶点B之间的最长路径长度大于1，且顶点B是顶点A的直接下游结点，则说明顶点A到顶点B之间还存在另一条非直接路径，因此可以删除工作流图中顶点A和顶点B之间的上下游关系，同时并不影响工作流中结点的整体调度执行顺序。

|  |
| --- |
| // 图剪枝方法  void reduce(ExecutionGraph graph) {  for (ExecutionNode node : graph.getAllNodes().values()) {  Map<String, Integer> longestDistance = new HashMap<>();  findLongestDist(node, 0, longestDistance);  for (Map.Entry<String, Integer> entry : longestDistance.entrySet()) {  if (entry.getValue() > 1 && node.getDownStreamNodes().containsKey(entry.getKey())) {  graph.removeRelation(entry.getKey(), node.getName());  }  }  }  }  // 深度优先搜索获取图中顶点之间的最长路径长度  void findLongestDist(DAGNode node, int distance, Map<String, Integer> longestDist) {  String name = node.getName();  longestDist.put(name, Math.max(distance, longestDist.getOrDefault(name, 0)));  Map<String, ? extends DAGNode> map = node.getDownStreamNodes();  for (DAGNode child : map.values()) {  if (longestDist.getOrDefault(child.getName(), 0) < distance + 1) {  findLongestDist(child, distance + 1, longestDist);  }  }  } |

图4. 18 图剪枝优化算法实现

使用真实生产环境中的一个工作流图，并利用LnP模块对此项优化措施带来的性能提升进行测试，结果表明约15.7%的边可以被消除（从48189条边减少到40618条边），同时不论是对于least cost调度策略，还是round robin调度策略，工作流运行的总延时都能得到14%左右的提升。图4.19中的折线图显示了此项性能测试结果。

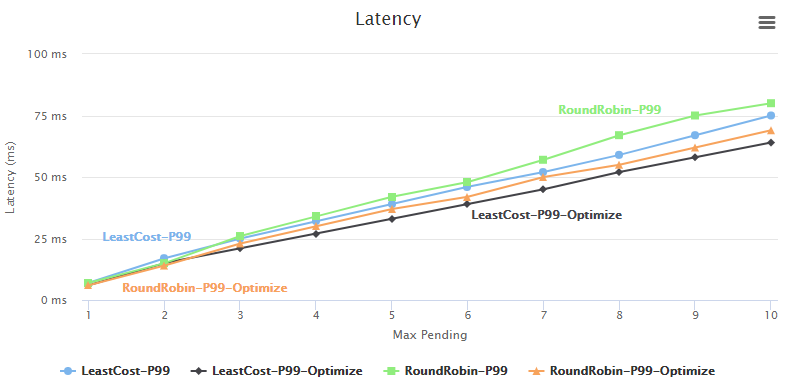


图4. 19 图剪枝优化带来的框架性能提升

## 本章小结

本章主要介绍了异步工作流框架的具体实现，主要包括执行器Executor模块、调度器Scheduler模块和LnP模块的核心功能实现以及一些关键的代码示例。同时，还描述了在开发过程中所实现的一些优化方案及其所带来的系统整体性能提升。

# 总结与展望

## 总结

本篇论文主要介绍了本人在实习企业参与的项目——异步工作流框架（Async Workflow Framework）的项目背景、框架整体结构以及系统关键模块的详细设计和具体实现方案。

首先，论文从具体的企业业务痛点出发，介绍了本异步工作流框架的开发背景和研究意义，揭示了其在减少系统开销的同时缩短当前工作流执行延时、提供系统吞吐量的最终实现目标。同时，简要描述了工作流的发展概况，并对项目中使用到的各项技术进行了较为详细的介绍，包括Netty EventLoop线程模型、CAS、Lock-Free队列等。

随后，介绍了工作流执行的典型生命周期过程，并以此为基础，分别从功能性需求和非功能性需求两个角度对该异步工作流框架的需求进行了详尽的分析，阐述了其需要满足的业务目标。

接着，介绍了该异步工作流框架的总体架构和具体模块划分，其主要包含依赖DAG构建部分，执行DAG构建部分，工作流构建部分和工作流执行引擎部分四个大块。

最后，选取了本人重点参与的三个核心模块：执行器Executor模块、调度器Scheduler模块和LnP模块，对其具体设计和实现方案进行了详细的论述，同时，还给出了一些具体的代码优化示例和运行结果显示，展示了此框架在具体提升工作流执行效率上的潜力。

## 进一步工作展望

目前本异步工作流框架的功能已经全部开发完成，并已经通过了LnP环境和QA环境的功能测试和其他相关测试，正在准备生产环境的上线过程。QA环境的测试结果显示，该框架在原有老框架的基础上可以将系统吞吐量提高近50%，99分位的系统延时可以降低近60%。当然，生产环境与QA环境相比还是有所不同，需要投入精力解决上线过程中可能遇到的各种问题，但是，其真正上线后可能带来的性能提升仍然非常值得期待。待顺利上线后，还可以将该框架与企业内部的其他服务系统进行集成，进一步提高企业内部服务系统的运行性能。同时，还可以考虑将此项目进行开源处理，便于拥有相似需求的企业或用户使用。

# 参考文献

[Calheiros et al., 2015] R. N. Calheiros, H. Kasim, T. Hung, X. Li, S. Lu, L. Wang, H. Palit, G. Lee, T. Ngo, and R. Buyya, Adaptive execution of scientific workflow applications on clouds, In *Cloud Computing with e-Science Applications*, pages 73-87, 2015.

[Chande, 2008] N. Chande, *A survey and risk analysis of selected non-bank retail payments systems*, Bank of Canada Discussion Paper, 2008.

[Cormen, 2009] T. H. Cormen, *Introduction to algorithms*, MA:MIT press, 2009.

[Dirksen, 2015] J. Dirksen, *RESTful Web Services with Scala*, UK:Packt Publishing Ltd, 2015.

[Eckel, 2016] B. Eckel, *Thinking in Java*, US:Prentice Hall, 2016.

[Ellson et al., 2004] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, Graphviz and dynagraph—static and dynamic graph drawing tools, In *Graph drawing software (Springer’2004)*, pages 127-148, 2004.

[Finagle, 2018] <https://github.com/twitter/finagle>, Twitter Finagle GitHub maintained by GitHub Inc, 2018.

[FindBugs, 2017] <https://github.com/findbugsproject/findbugs>, FindBugs GitHub maintained by GitHub Inc, 2017.

[Freemarker,2018] <https://freemarker.apache.org>, Apache FreeMarker maintained by The Apache Software Foundation, 2018.

[Goetz et al., 2006] B. Goetz and T. Peierls, Java concurrency in practice, London:Pearson Education, 2006.

[Gregg, 2013] B. Gregg, *Linux Performance Analysis and Tools*, Technical report, Joyent, 2013.

[Hendler, 2011] D. Hendler, Non-Blocking Algorithms, In *Padua D. (eds) Encyclopedia of Parallel Computing(Springer’2011)*, pages 1321-1328, 2011.

[Herlihy, 1991] M. Herlihy, Wait-free synchronization, In *ACM Transactions on Programming Languages and Systems (TOPLAS’1991)*, pages 124-149, 1991.

[Herlihy et al., 2011] M. Herlihy, and N. Shavit, *The art of multiprocessor programming*, US:Morgan Kaufmann Publishers, 2011.

[Hook et al., 2018] J. Hook, and N. Dingle, *Performance analysis of asynchronous parallel Jacobi*, Numerical Algorithms, 77(3), pages 831-866, 2018.

[HTCondor,2017] <http://research.cs.wisc.edu/htcondor/dagman/dagman.html>, DAGMan page maintained by cs.wisc.edu, 2017.

[Jain et al, 2015] A. Jain, S. P. Ong, W. Chen, B. Medasani, X. Qu, M. Kocher, M. Brafman, G. Petretto, G. M. Rignanese, G. Hautier, and D. Gunter, FireWorks: A dynamic workflow system designed for high throughput applications, In *Concurrency and Computation: Practice and Experience, 27(17)*, pages 5037-5059, 2015.

[JavaBugDb,2017] <https://bugs.java.com/bugdatabase/view_bug.do?bug_id=JDK-8178956>，Java Bug Database maintained by Oracle, 2017.

[Java-Yield,2016] <https://www.jianshu.com/p/0964124ae822>, blog maintained by jianshu, 2016.

[JCTools, 2018] <https://github.com/JCTools/JCTools>，JCTools GitHub maintained by GitHub Inc, 2018.

[Jenkins, 2018] <https://www.cloudbees.com/jenkins/about>, Jenkins Newsletter maintained by CloudBees Inc, 2018.

[Ladan et al., 2004] E. Ladan-Mozes, and N. Shavit, An optimistic approach to lock-free FIFO queues, In *International Symposium on Distributed Computing(Springer’2004)*, pages 117-131, 2004.

[Maurer et al., 2015] N. Maurer, and W. A. Marvin, *Netty in Action*, New York: Manning Publications, 2016.

[Meehan, 2006] A. Meehan, Performance implications of memory management in Java, In *WSEAS Transactions on Computers, 5(7)*, pages 1451-1457, 2006.

[Metrics, 2017] <http://metrics.dropwizard.io/4.0.0/index.html>，Metrics maintained by Dropwizard Team, 2017.

[Meyers, 2013] S. Meyers, *CPU Caches and Why You Care*, 2013.

[Neeraj et al., 2017] K. R. Neeraj, P. S. Janardhanan, A. B. Francis, and R. Murali, A domain specific language for business transaction processing, In *Signal Processing, Informatics, Communication and Energy Systems (SPICES’2017)*, pages 1-7, 2017.

[Ogasawara et al., 2013] E. Ogasawara, J. Dias, V. Silva, F. Chirigati, D. Oliveira, F. Porto, P. Valduriez, and M. Mattoso, Chiron: a parallel engine for algebraic scientific workflows, In *Concurrency and Computation: Practice and Experience 25, no. 16*, pages 2327-2341, 2013.

[Pandey et al., 2011] S. Pandey, D. Karunamoorthy, and R. Buyya, Workflow engine for clouds, In *Cloud computing: principles and paradigms*, pages 321-344, 2011.

[Shotts et al., 2012] J. Shotts, and E. William, *The linux command line: A complete introduction*, San Francisco:No Starch Press, 2012.

[SSH-Plugin,2017] <https://wiki.jenkins.io/display/JENKINS/SSH+plugin>,Jenkins plugin page maintained by Jenkins, 2017.

[strace, 2018] <https://github.com/strace/strace>, strace GitHub maintained by GitHub Inc, 2018.

[Thompson, 2011] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart, *Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads*, Technical paper, LMAX 206, 2011.

[Transitive\_reduction,2018] <https://en.wikipedia.org/wiki/Transitive_reduction>,Transitive Reduction Wikipedia maintained by Wikipedia Foundation Inc, 2018.

[Van et al., 2004] W. Van Der Aalst, and K. M. Van Hee, *Workflow management: models, methods, and systems*, MA:MIT press, 2004.

[Virouleau et al., 2016] P. Virouleau, F. Broquedis, T. Gautier, and F. Rastello, Using data dependencies to improve task-based scheduling strategies on NUMA architectures, In *European Conference on Parallel Processing (Springer’2016)*, pages 531-544, 2016.

[vmstat, 2018] <https://linux.die.net/man/8/vmstat>, vmstat man page maintained by linux.die.net, 2018.

[Wang, 2013] L. Wang, Directed acyclic graph, In *Encyclopedia of Systems Biology (Springer’2013)*, pages 574-574, 2013.

[Wangle, 2018] <https://github.com/facebook/wangle/>, Wangle GitHub maintained by GitHub Inc, 2018.

[Workflow, 2018] <https://en.wikipedia.org/wiki/Workflow> , Workflow Wikipedia maintained by Wikipedia Foundation Inc, 2018.

[兰洋等，2016] 兰洋，温迎福，*持续集成实践*，北京：电子工业出版社，2016。

[刘建伟，2010] 刘建伟，第三方支付平台风险控制问题探讨，*金融理论与实践, (12)：64-67*，2010。

[罗海滨等，2000] 罗海滨，范玉顺，吴澄，*工作流技术综述*，博士论文，清华大学自动化系，2000。

[孙卫琴，2010] 孙卫琴，*精通 Hibernate: Java 对象持久化技术详解*，北京：电子工业出版社，2010。

[许晓斌，2011] 许晓斌，*Maven实战*，北京：机械工业出版社，2011。

# 致 谢

在本篇论文即将完成之际，我谨向支持、帮助和指导本人的诸位表示由衷的感谢。

感谢王崇骏教授一直以来的悉心指导。从最初论文标题的选择，论文大纲的拟定，到论文最后的撰写、修改直至定稿的整个过程中，王教授一直耐心解答我的疑惑，提出很多建设性的意见，给予了我莫大的帮助。

感谢软件学院对我的照顾和培养，是学院提供的完善教学体系和所有教师的谆谆教诲，让我收获了知识，度过了人生中最难忘的两年。

感谢企业的各位领导、导师和同事。他们在我实习期间给了我极大的帮助，对我在公司完成的项目也给予了很大的肯定，使我的实习生活在过往的人生经历中成为了浓墨重彩的一笔。尤其是我的导师，工作之余也对本篇论文的撰写提供了很多宝贵的建议，在此表示真挚的感谢。

感谢家人一直以来在背后的默默支持，让我可以全身心地投入到工作、生活和学习当中。

最后，对参与本次论文审阅和答辩的各位老师致以诚挚的谢意，感谢你们的辛苦付出。

衷心祝愿你们身体健康，万事如意。

# 版权及论文原创性说明

任何收存和保管本论文的单位和个人，未经作者本人授权，不得将本论文转借他人并复印、抄录、拍照或以任何方式传播，否则，引起有碍作者著作权益的问题，将可能承担法律责任。

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。除文中已经注明引用的内容外，本论文不包含其他个人或集体已经发表或撰写的作品成果。本文所引用的重要文献，均已在文中以明确方式标明。本声明的法律结果由本人承担。

作者签名：

日期： 年 月 日