分　类　号：TP391　　单位代码：10190
研究生学号：2202003080　　密　　级：公 开

長春工業大學

硕士专业学位论文

姜　猛

2023 年 6 月

基于 Pod 流量数据的 Kubernetes 平台调度算法的研究

Research on Scheduling Algorithm of Kubernetes Platform Based on Pod Traffic Data

硕 士 研 究 生：姜猛

导　　师：刘 钢教授

校 外 导 师：杨 琼高级工程师

申 请 学 位：电子信息硕士

领　　域：计算机技术

所 在 单 位：计算机科学与工程学院

答 辩 日 期：2023 年 6 月

授予学位单位：长春工业大学

摘 要

随着容器技术的不断发展，越来越多的企业和机构选择将自家开发的应用以容器的形式部署到容器平台中进行统一调度管理，调度算法既要保障调度后的集群能够拥有提供良好服务的能力，同时对调度过程本身也具有一定的时间要求。目前，Kubernetes 作为一种被大部分企业和机构使用的主流容器编排技术，其调度算法存在负载不均衡与调度时间开销大的问题，影响了集群整体提供服务的能力与应用的调度部署效率。因此，本文针对 Kubernetes 调度算法存在的负载不均衡以及调度时间开销大的问题，提出了相应优化方案，主要工作内容如下：

针对 Kubernetes 默认调度算法存在的负载不均衡问题，本文通过对标准 Kubernetes 调度模型进行分析，定位到负载不均衡问题的产生两点原因：1.Kubernetes 默认调度算法缺失了对集群节点网络环境的评估。2.默认调度算法使用节点剩余资源数据进行优选打分的计算逻辑忽略了集群节点间的配置差异问题，对节点的负载程度容易产生误判。基于这两点优化方向，本文提出了一种基于 Pod 网络改进的调度策略 TLB（Traffic and load balancing，TLB），在调度策略中引入 Istio 及 Prometheus 监控组件构建 Pod 真实负载获取模型，计算比带宽占用率更细粒度的 Pod 真实网络分配率以弥补默认调度算法对网络环境评估的缺失。结合 Pod 的真实 CPU、内存以及网络分配律三项指标设计节点理想负载偏离度指标进行调度优选打分计算。实验结果表明，TLB 调度策略在解决负载不均衡问题上具有有效性和可行性。

针对 Kubernetes 默认调度算法存在的调度时间开销大的问题，本文定位到问题的产生原因在于默认调度算法对集群节点采用了串行化的优选打分机制，导致优选阶段在整个调度过程中占用了过长的时间。基于这一优化方向，本文设计了一种基于模拟退火思想改进的自适应遗传算法（Improved Genetic Algorithm，IGA），相比于标准遗传算法（Standard Genetic Algorithm，SGA）在 Kubernetes 调度问题上拥有更快的收敛速度，应用该算法构建并行化调度优选模型，将默认调度算法串行化的优选打分机制替换成对 Pod 分配方案的整体寻优迭代，降低调度的时间开销。并将 IGA 与 TLB 调度策略进行融合优化，提出了一种基于 IGA 改进的并行调度策略 TLB-IGA(Traffic and load balancing-IGA，TLB-IGA)。实验结果表明，TLB-IGA 调度策略在解决负载不均衡问题和调度时间开销大的问题上同时具有有效性和可行性，并且相比于其他没有使用本文 Pod 真实负载获取模型的负载均衡调度策略在提升负载均衡效果上具有一定优越性。

关键词： Kubernetes Istio 负载均衡 调度时间 遗传算法

Abstract

With the continuous development of container technology, more and more enterprises and organizations choose to deploy their own developed applications in the form of containers to the container platform for unified scheduling management, scheduling algorithms to ensure that the scheduling cluster can have the ability to provide good services, while the scheduling process itself also has certain time requirements. Currently, Kubernetes is a mainstream container scheduling technology used by most enterprises and organizations, but its scheduling algorithm has the problems of load imbalance and high scheduling time overhead, which affects the overall ability of the cluster to provide services and the efficiency of application scheduling and deployment. Therefore, this paper proposes an optimization solution to address the load imbalance and high scheduling time overhead of Kubernetes scheduling algorithm, and the main work is as follows:

To address the load imbalance problem of the Kubernetes default scheduling algorithm, this paper analyzes the standard Kubernetes scheduling model and locates two reasons for the load imbalance problem: 1. the Kubernetes default scheduling algorithm is missing the evaluation of the cluster node network environment. 2. the default scheduling algorithm uses the node residual resource data for preference The calculation logic of scoring ignores the configuration differences among cluster nodes, and it is easy to misjudge the load level of nodes. Based on these two optimization directions, this paper proposes a scheduling policy TLB (Traffic and load balancing, TLB) based on Pod network improvement, which introduces Istio and Prometheus monitoring components in the scheduling policy to build a Pod real load acquisition model and calculate the Pod real network allocation rate at a finer granularity than the bandwidth occupation rate to compensate for the lack of network environment evaluation by the default scheduling algorithm. The node ideal load deviation metric is designed by combining the three metrics of Pod's real CPU, memory and network allocation law for scheduling preference scoring calculation. The experimental results show that the TLB scheduling strategy is effective and feasible in solving the load imbalance problem.

To address the problem of high scheduling time overhead in the default scheduling algorithm of Kubernetes, this paper locates that the problem arises because the default scheduling algorithm adopts a serialized preference scoring mechanism for cluster nodes,

which causes the preference phase to take up too much time in the whole scheduling process. Based on this optimization direction, this paper designs an Improved Genetic Algorithm (IGA) based on the idea of simulated annealing, which has a faster convergence speed than the Standard Genetic Algorithm (SGA) for the Kubernetes scheduling problem. This algorithm is applied to build a parallelized scheduling preference model, replacing the serialized preference scoring mechanism of the default scheduling algorithm with an overall preference-seeking iteration of the Pod allocation scheme to reduce the scheduling time overhead. And the IGA and TLB scheduling policies are fused and optimized, and a parallel scheduling policy TLB-IGA (Traffic and load balancing-IGA, TLB-IGA) based on IGA improvement is proposed. The experimental results show that the TLB-IGA scheduling strategy is both effective and feasible in solving the load imbalance problem and the problem of large scheduling time overhead, and it is superior in improving the load balancing effect compared with other load balancing scheduling strategies that do not use the real load acquisition model of Pod in this paper.

Key words: Kubernetes Istio Load Balancing Scheduling Time Genetic-Algorithm

目　录

第 1 章 绪 论

1.1 研究背景及意义

随着容器平台技术的不断发展，越来越多的企业和厂商选择将自家开发的应用进行容器化部署[1]，打破机器间的硬件隔离，实现资源共享[2]，使异构机器的硬件计算资源尽可能得到充分而统一的利用[3]。容器平台成为众多企业和厂商使用的主流应用虚拟化方案[4]。目前，以 Kubernetes[5]容器平台为基础的统一调度[6]与管理系统占据了市场的大半份额。Kubernetes 具有轻量级、高性能、组件化和开源社区生态等诸多优点，其内置的默认调度器 kube-scheduler[7]是一个支持高度自由开发的基础组件，为了追求更高效、准确、稳定的任务调度，对 Kubernetes 调度算法的研究[8]也成为了目前企业和厂商研究的热点方向之一，众多知名大厂例如 IBM、华为、阿里巴巴都开始将 Kubernetes 的研发重视起来[9]，使用 Kubernetes 作为主要框架[10]搭建容器平台，通过自动化工具对容器进行部署、运行和管理[11]，实现了如灵雀云、DaoCloud 等专业的容器服务平台，向企业提供定制化的容器解决方案[12]。

对 Kubernetes 的广泛应用，也让企业和厂商对 Kubernetes 调度算法的要求随之提高[13]，既要保障集群调度结果拥有良好服务质量[14]，又对调度过程本身也具有一定的时间开销[15]要求。在这两方面需求下，Kubernetes 默认调度算法已不足以支持工作负载[16]的正确调度，其原因在于 Kubernetes 默认调度算法缺失了对集群节点网络运行情况的评估以及算法根据节点剩余资源情况进行优选打分[17]的逻辑都会导致的集群负载不均衡问题，集群中真实负载已经很严重的节点还会被分配应用 Pod，而其他节点却容易出现资源空余的状况，对集群应用整体的服务能力产生影响，从而出现卡顿，响应时间慢等服务质量差的问题，甚至节点的压力过高时，产生了宕机的风险[19]。同时 Kubernetes 默认调度策略串行化的优选打分机制[20]也造成了调度时间开销大的问题[21]。综上，一个能够以较低时间开销来实现负载均衡效果的调度策略对于 Kubernetes 容器平台的企业应用具有重要意义，因此本文将针对负载均衡和调度时间开销两方面问题，深入分析 Kubernetes 默认调度算法的不足，对此做出相应优化方案，从理论和实验两方面出发，优化 Kubernetes 调度算法。

1.2 国内外研究现状

在 Kubernetes 平台调度算法的负载均衡与调度时间开销问题上，国内外的众多学者和研究机构对此做出了研究改进，国内外的研究现状如下：

针对集群调度的负载均衡问题，在国内：黄志成[22]设计了一种资源监控的架构，并提出了一种基于集群负载均衡的资源调度算法，考虑节点自身的负载情况，在预选

阶段将历史负载较高的节点过滤掉，在优选阶段根据历史负载数据进行打分，实现负载均衡效果；李华东[23]等人使用实时监控的手段获取服务请求的实际资源使用情况,以实现服务的动态调度并建立物理节点间的合作博弈模型,以减少集群资源碎片化现象，提升负载均衡效果；熊衍捷[24]等人设计了一种基于谱聚类的 BaaS 资源负载均衡调度算法，实现了负载均衡效果；胡程鹏[25]设计了基于 Spark 的并行 GA 资源调度策略 SP-GA,并提出了四种动态资源调度触发机制，实现了调度结果的负载均衡。陈丰琴[26]提出了一种改进的动态负载均衡调度优选策略，保证 Pod 高并发调度下集群趋于理想化的负载均衡。罗永安[27]通过建立多 Pod 调度的优化模型，将集群整体的负载均衡度作为目标函数，设计了一种 K-DPSO 算法得到使集群负载均衡的静态调度方案。在国外：Nguyen[28]提出了一种资源自适应代理的增强型负载均衡器，定期监控每个 pod 的资源状态和工作节点之间的网络状态，以帮助做出负载平衡决策；Dua[29]等人通过配置专用于特定任务类型的集群，为每个作业定义标签进行分类，使用任务迁移引入负载平衡技术；Karypiadis[30]等人通过提出了 SCAL-E 智能代理，生成负载均衡调度策略。

针对调度时间开销问题，在国内：刘哲源[31]等人通过使用模拟退火优化粒子群算法提升调度算法性能;赵飞鸿[32]等人设计基于深度强化学习的资源调度方法，学习调度策略提升调度执行速度；蒋筱斌[33]等人提出了一种创建时调度的 ISAMatch 模型,综合考虑指令集亲和性、同种指令集架构节点数量等多个方面, 快速实现任务的最佳分配；戴志明[34]等人提出基于遗传算法改进的区间划分遗传调度算法，对智能工厂中的容器应用进行调度分配，提升算法效率；徐胜超[35]等人提出多集群容器云资源调度优化方法，具有较短调度时延。在国外：EI[36]等人通过对考虑 Pod 时间表和有关容器执行的历史信息来优化新容器的部署，降低调度任务执行时间；Chima[37]等人考虑物理、操作和网络参数以及软件状态动态编排应用；Raush[38]等人提出了一种方法来自动微调调度约束的权重，以优化高层操作目标，可选择最小化任务执行时间；Panda[39]等人设计了一种新的分布式容器调度方案 pMACH 用于优化数据中心的功耗和任务完成时间；Harichane[40]等人提出了一个用于 Kubernetes 环境的调度程序 KubeSC-RTP，使用基于应用程序运行时预测的机器学习，快速寻找调度最优方案。

众多研究改进对调度算法有着显著优化成效的同时，仍存在优化方向可以提升负载均衡问题的解决效果，在本文定位的负载均衡问题产生原因中，填补默认调度算法缺失的网络评估指标时，可以进一步细粒度划分节点 Pod 网络占用率和节点综合网络占用率，以达到对 Pod 调度进行针对性更好的资源评估。在调度过程中，对节点的网络、CPU 以及内存资源均采取同样的评估方式可以提升集群负载均衡效果，本文对这一优化方向展开了相关工作。

1.3 研究内容及创新点

本文主要针对 Kubernetes 默认调度算法存在的负载不均衡问题以及调度时间开销大的问题进行研究，具体研究内容如下：

（1）针对 Kubernetes 默认调度算法存在的负载不均衡问题，本文提出了一种基于 Pod 网络改进的调度策略 TLB，通过构建 Pod 真实负载获取模型计算节点的网络、CPU、以及内存三项 Pod 真实资源分配率，利用三项 Pod 真实资源分配率指标设计节点实时综合资源利用率指标反应节点 Pod 的真实负载，弥补了默认调度策略对节点网络环境评估的缺失；基于节点实时综合资源利用率指标设计节点理想负载偏离度指标以反应节点距离理想负载均衡状态的偏离程度，对默认调度算法的优选打分逻辑进行了优化，实现了负载均衡的调度效果。

（2）针对 Kubernetes 默认调度算法存在的调度时间开销大的问题，本文提出了一种基于 IGA 改进的并行调度策略 TLB-IGA，通过使用基于模拟退火思想改进的自适应遗传算法 IGA，构建并行化调度优选模型，对本文提出的 TLB 调度策略做出调度时间开销上的优化改进，同时实现了较低的调度时间和较好的负载均衡效果，提升了集群服务能力。

在上述研究内容中，本文的创新点如下：

（1）本文提出的 TLB-IGA 调度策略使用 Pod 真实负载获取模型，用节点单项资源的 Pod 占用率与节点单项资源的综合占用率占比，计算节点 Pod 真实资源分配率指标，以反应节点对 Pod 的资源分配程度，在 Kubernetes 调度问题上实现了对 Pod 更加具有针对性的资源评估，实验结果表明 TLB-IGA 调度策略相比于其他没有使用本文 Pod 真实负载获取模型的负载均衡调度策略在提升负载均衡效果上具有一定优越性。

（2）本文提出 TLB-IGA 调度策略设计并使用了一种基于模拟退火思想改进的自适应遗传算法，在标准遗传算法中加入扰动机制，使用节点理想负载偏离度指标设计自适应交叉算子和自适应变异算子，相比于标准遗传算法在 Kubernetes 调度问题上实现了更快的收敛速度，实验结果表明 TLB-IGA 调度策略使用的改进遗传算法 IGA 相比于标准遗传算法 SGA 在 Kubernetes 调度问题的寻优能力上具有优越性。

1.4 论文组织结构

本文共分为六个章节，论文结构安排如下所示：

第 1 章为绪论。本章节首先对 Kubernetes 平台调度算法的研究背景及研究意义进行了介绍，接着对国内外针对 Kubernetes 平台调度算法的的负载均衡和时间开销两方面问题的研究现状进行描述，再对本文的研究内容及创新点进行了说明，最后介绍了论文组织结构。

第 2 章为相关理论与技术介绍。本章节对 Kubernetes 平台调度算法的研究所需要涉及的众多理论与关键技术，包括 Kubernetes 的核心功能及其核心元素概念，遗传算法核心元素与算法流程，以及 Prometheus 和 Isito 监控组件技术的作用等，为本文后续内容的输出与研究做技术准备支持。

第 3 章为基于 Pod 网络改进的 Kubernetes 调度算法。本章节首先针对 Kubernetes 默认调度算法的不足进行分析，随后针对 Kubernetes 默认调度算法对集群节点网络运行情况评估的缺失以及默认优选策略根据节点剩余资源情况进行优选打分的逻辑所导致的集群负载不均衡问题，提出一种基于 Pod 网络改进的调度策略 TLB。后通过介绍 TLB 调度策略的调度模型，对调度策略优化点进行定位；随后对该调度策略所使用的 Pod 真实负载获取模型进行介绍；再对该调度策略所使用的节点实时综合资源利用率指标以及节点理想负载偏离度指标的计算逻辑及作用进行了说明；最后对该调度策略的调度流程和打分逻辑进行了详细阐述，完成对 TLB 调度策略的技术说明。

第 4 章为基于 IGA 改进的并行调度策略 TLB-IGA。本章节针对 Kubernetes 默认调度算法优选阶段中的串行化调度优选打分逻辑所导致的调度时间开销大的问题，提出了一种基于 IGA 改进的并行调度策略 TLB-IGA。后通过介绍该调度策略的调度模型，对该调度策略优化点进行定位，随后对该调度策略所使用的并行化调度优选模型进行介绍；再对该调度策略所使用的基于模拟退火思想改进的自适应遗传算法 IGA 的设计与算法流程进行了详细阐述；最后对 TLB-IGA 调度策略的调度流程和打分逻辑进行了介绍。完成对 TLB-IGA 调度策略的技术说明。

第 5 章为实验设计与结果分析。本章包括了三个实验：首先是基于 TLB-IGA 调度策略的资源调度实验，该实验通过使用第 3、4 章节提出的 TLB 和 TLB-IGA 两种调度策略与 Kubernetes 默认调度策略进行调度对比实验，利用调度结果的节点负载标准差与调度时间两个指标，验证本文提出的调度策略的优化效果；其次是基于 TLB-IGA 调度策略的集群性能实验，该实验通过压力测试对 TLB-IGA 调度策略与 Kubernetes 默认调度策略发起多线程访问，以 QPS(Queries Per Second，每秒处理请求条数)和 RT (Response time，平均响应时间）两个指标验证 TLB-IGA 调度策略对于集群整体提供服务的能力的提升；最后是与标准遗传算法（SGA）的对比试验，该实验以算法迭代次数和调度时间两个指标验证改进后的 IGA 在 Kubernetes 调度问题上寻优能力的提升。三个实验都确立了实验目标、说明了实验相关软件和硬件环境及实验搭建和实验的具体执行过程及结果分析。

第 6 章为总结与展望。对本文主要的研究内容进行了总结，以及对于未来的一些可能的研究方向进行了展望。

第 2 章　相关理论与技术介绍

本章对与本文研究内容相关的 Kubernetes、遗传算法、istio、prometheus 技术进行了详细介绍，为本文工作内容的输出提供技术支持。

2.1 Kubernetes 技术概述

Kubernetes 是目前最为主流的容器编排平台，其研究和应用现状非常活跃，具有庞大且完善的集群管理功能，具体涵盖了内置智能化的故障查找[41]和自我诊断处理功能、服务滚动升级和在线扩容等功能[42]。

2.1.1 核心功能介绍

Kubernetes 作为开源的容器编排平台，其核心功能主要用于管理和自动化容器化应用程序的部署、扩展和管理[43]，核心功能的作用机制主要包括以下几个方面：

（1）容器编排：Kubernetes 通过定义抽象层来管理容器集群，抽象层包括应用程序、服务、副本集和标签等。Kubernetes 通过这些抽象层对容器进行编排和管理，从而实现容器的快速部署和扩展。

（2）容器调度：Kubernetes 采用基于声明的方法来定义容器的资源需求和约束条件，从而可以对容器进行有效的调度，如图 2.1 所示，Kubernetes 通过调度器来确定容器的调度位置，并通过节点管理器来管理和监控容器的运行状态，完成调度流程。

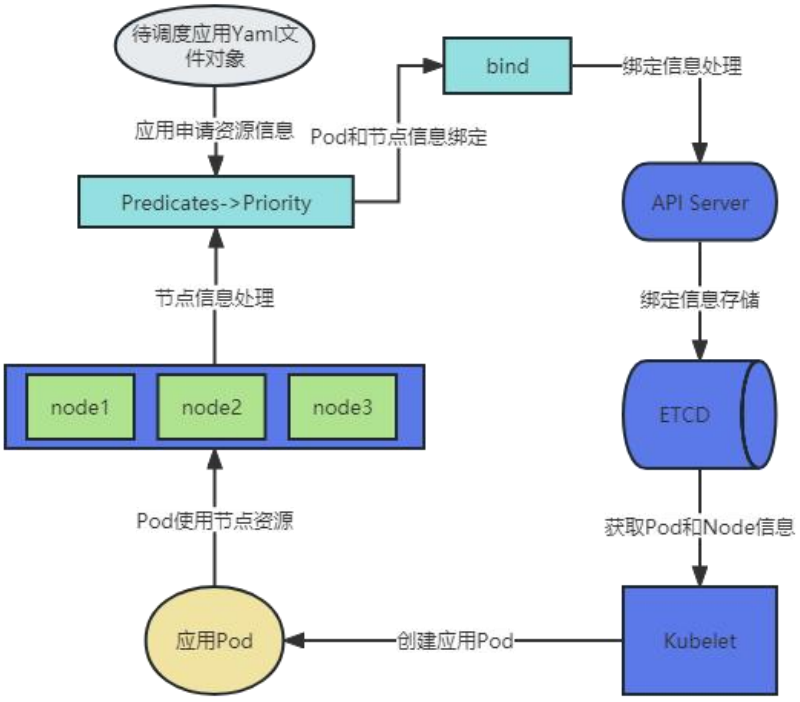

图 2.1 Kubernetes 容器调度流程

（3）资源管理：Kubernetes 可以自动化地管理容器的资源分配和利用，从而最大化容器集群的利用率。Kubernetes 可以通过资源管理器来监控容器的资源使用情况，并根据需要对容器进行动态扩展或收缩。

（4）容器网络：Kubernetes 可以为容器提供灵活、可靠的网络连接，支持多种网络协议和网络模型。Kubernetes 可以为容器分配 IP 地址，并通过服务发现和负载均衡来确保容器之间的通信和协作。

（5）存储管理：Kubernetes 可以管理容器的存储资源，支持多种存储类型和存储模型。Kubernetes 可以为容器分配存储卷，并通过存储管理器来管理存储资源的分配和使用。

总的来说，Kubernetes 的作用机制主要包括容器编排、容器调度、资源管理、容器网络和存储管理等方面。通过这些机制，Kubernetes 可以自动化地管理和扩展容器集群，提高容器化应用程序的可靠性和可管理性，实现快速部署和弹性扩展的目标。

2.1.2 核心元素概念

Kubernetes 作为用于管理容器化应用程序的开源平台，其中包含了许多重要的概念。以下是一些重要的概念，按照功能元素和架构组成可以分为两个部分：

其中功能元素部分的概念包括：

（1）Pod：Pod 是 Kubernetes 中最小的可部署单元，如图 2.2 所示，它是一个或多个容器的组合，这些容器共享网络和存储。Pod 提供了一个封装的环境，以便于在 Kubernetes 中进行应用程序部署和管理。

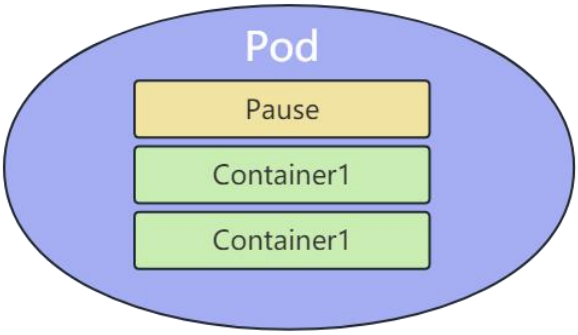

图 2.2 Pod 的结构

（2）Service：Service 是一组 Pod 的抽象，可以将它们打包在一起并提供一个统一的访问入口。Service 提供了一种稳定的 IP 地址和端口，以便其他应用程序可以使用它来访问应用程序。

（3）Replication Controller：Replication Controller 是 Kubernetes 中的一个核心概念，它用于确保在任何时候都运行着指定数量的 Pod 实例。Replication Controller 监

视 Pod 实例的运行状态，并在需要时启动或停止实例，以保持其数量不变。

（4）Deployment：Deployment 是 Replication Controller 的高级版本，如图 2.3 所示，它提供了更多的功能，例如滚动更新、回滚等。Deployment 通过控制 ReplicaSet 来管理 Pod 实例的创建和销毁。

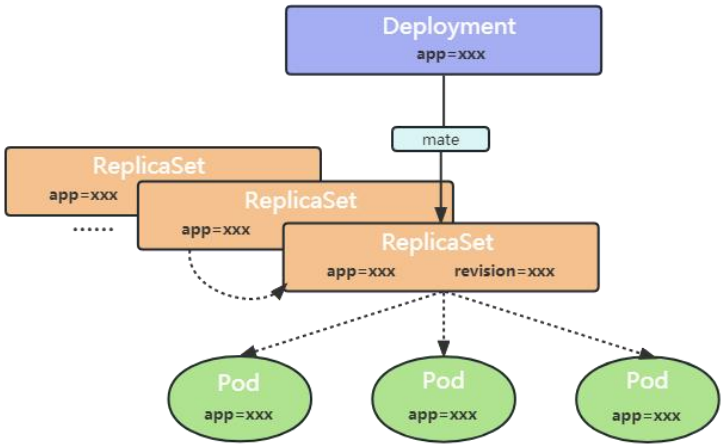

图 2.3 Deployment 的功能

（5）ConfigMap 和 Secret：ConfigMap 和 Secret 是 Kubernetes 中用于管理应用程序配置信息和敏感信息的机制。ConfigMap 用于存储应用程序配置信息，例如数据库连接字符串、环境变量等。Secret 用于存储敏感信息，例如密码、证书等。

（6）Namespace：Namespace 是 Kubernetes 中用于隔离不同组织或应用程序的逻辑分区。通过使用 Namespace，可以将不同的应用程序、环境或团队隔离开来，从而更好地管理资源和安全。

（7）Node：Node 是 Kubernetes 集群中的工作节点，它可以是物理主机或虚拟机。Node 上运行着 Pod 实例，并负责处理 Pod 实例的网络和存储。

（8）Master：Master 是 Kubernetes 集群的控制节点，它负责管理和调度集群中的所有资源和任务。Master 通常由多个组件组成，例如 API Server、Scheduler、Controller Manager 等。

架构组成部分的概念包括：

（1）API Server：API Server 是 Kubernetes 集群中的核心组件，它提供了 Kubernetes API 的访问入口。如图 2.4 所示，API Server 接受来自客户端的请求，处理和验证这些请求，并将它们转发到相应的组件进行处理。

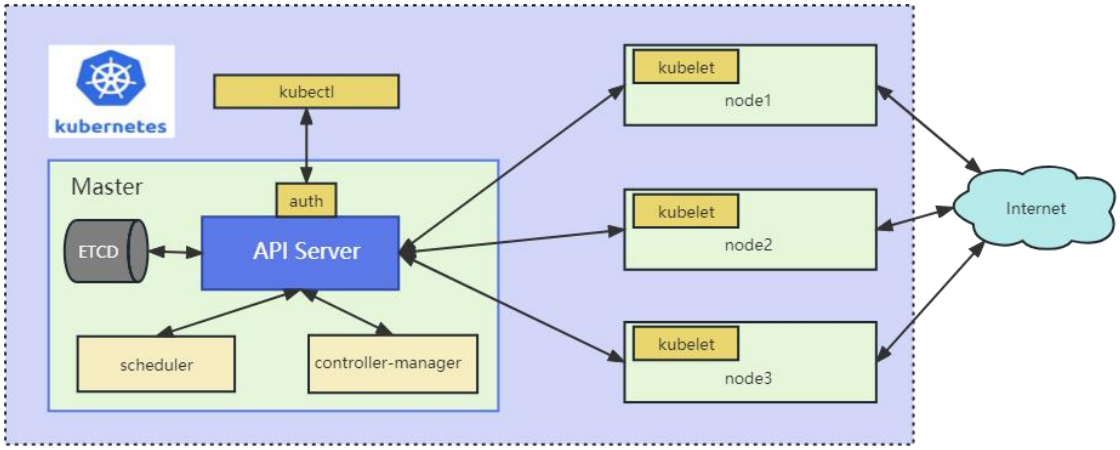

图 2.4 API Server 的功能

（2）etcd：etcd 是 Kubernetes 集群中用于存储集群状态的分布式键值存储系统。etcd 存储集群中的所有资源和配置信息，例如 Pod、Service、ConfigMap 等。

（3）Scheduler：Scheduler 是 Kubernetes 集群中的一个核心组件，如图 2.5 所示，它负责将 Pod 实例分配给集群中的 Node。Scheduler 根据各种条件和策略选择最佳的 Node，并将 Pod 调度到该 Node 上运行。

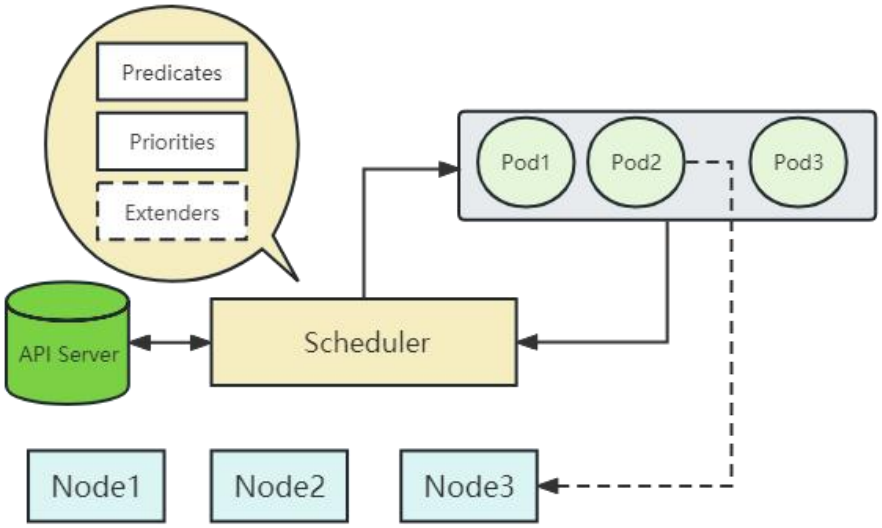

图 2.5 Scheduler 的功能

（4）Controller Manager：Controller Manager 是 Kubernetes 集群中的另一个核心组件，它负责管理各种控制器，例如 Replication Controller、Deployment 等。Controller Manager 监视集群中的各种资源，确保它们处于期望的状态，并在需要时采取行动。

（5）Volume：Volume 是 Kubernetes 中用于管理持久化数据的机制。Volume 提供了一种抽象的方法来管理存储，使得应用程序可以将数据存储到磁盘、网络存储等

不同的存储介质中。

（6）StatefulSet：StatefulSet 是 Kubernetes 中用于管理有状态应用程序的机制。StatefulSet 确保有状态应用程序的 Pod 实例在启动和停止时具有稳定的标识符和网络标识符，以便其他应用程序可以访问它们。

这些概念构成了 Kubernetes 平台的基本组成部分，对于在 Kubernetes 上进行应用程序部署和管理至关重要。

2.2 遗传算法概述

遗传算法（Genetic Algorithm，GA）是一种基于生物进化理论的启发式搜索算法[44]，常用于解决复杂的优化问题。它是由美国学者约翰·荷兰（John Holland）在 20 世纪 60 年代提出的。遗传算法是一种通用的优化算法，能够处理多种类型的问题。它具有全局搜索能力和并行处理能力，可以处理非线性、非凸、多峰和约束等各种优化问题。遗传算法的优点在于其可以处理大规模、复杂、多目标优化问题，并且不需要对目标函数的数学性质做出任何假设。遗传算法还可以通过并行计算来提高求解效率，具有很高的适应性和鲁棒性，适用于各种类型的问题。遗传算法可以在多种领域中发挥作用，如在工业制造中优化生产过程、在农业中优化种植方案、在金融领域中优化资产配置等。同时，在人工智能和机器学习领域，遗传算法也被广泛应用于神经网络结构优化、特征选择和分类问题等。遗传算法是一种强大的求解优化问题的工具，具有广泛的应用前景。

2.2.1 遗传算法核心元素介绍

遗传算法的核心元素从算法过程角度可以分为基础概念和进化概念。

遗传算法的基础概念包括：

（1）染色体：指个体的编码方式，染色体中包含了问题的解或者解的部分信息，它是遗传算法中基因的集合。

（2）种群：是指一组具有相似特征的个体集合，它是遗传算法的基础。种群中每个个体都代表了一个可能的解，而种群的适应度值则反映了该解的优劣程度。种群中的个体可以是任何问题的解，例如函数的最优解、参数寻优、排列问题的解等。

遗传算法的进化概念包括：

（1）适应度函数：用于评估个体的优劣程度，也就是个体适应程度。适应度函数是问题的关键，它将每个个体映射到一个实数值，这个实数值反映了个体的优越性。适应度函数的选择对算法的性能有很大影响，通常需要根据具体问题的特点设计。

（2）选择策略：用于从当前种群中选择出适应度高的个体，作为下一代个体的父代。选择策略可以根据适应度值进行比例选择、竞争选择等。

（3）交叉操作：用于产生新的个体，通过将两个父代个体的染色体进行交叉操作，产生新的个体。交叉操作的种类有单点交叉、两点交叉、均匀交叉等。

（4）变异操作：用于增加种群的多样性，通过改变某些个体的染色体基因，产生新的个体。变异操作的种类有位变异、交换变异、反转变异等。

（5）种群管理：用于控制种群大小，确保种群的多样性和进化的速度。种群管理包括种群初始化、种群复制、种群淘汰等。

这些元素共同作用，使遗传算法能够在搜索空间中进行有效的探索，并逐步寻找到问题的最优解。

2.2.2 遗传算法算法流程介绍

遗传算法的算法流程如下所示：

步骤（1）初始化种群：首先随机生成一定数量的染色体作为初始种群，并对每个染色体进行适应度评估，即计算其适应度值。

步骤（2）进行选择操作：从当前种群中选择适应度较高的个体作为下一代种群的父代。通常使用轮盘赌选择或者竞争选择等方法进行选择。

步骤（3）进行交叉操作：将被选择的父代染色体进行交叉操作，生成新的后代染色体。交叉操作可以采用单点交叉、多点交叉或者均匀交叉等方式。

步骤（4）进行变异操作：对交叉后的染色体进行变异操作，引入新的基因信息。变异操作通常是随机选择染色体的某些基因，然后进行变异。

生成新种群：将交叉和变异后的染色体作为新的个体加入下一代种群。

步骤（5）评估适应度：对新的种群进行适应度评估，计算每个染色体的适应度值。

步骤（6）终止条件检查：检查终止条件是否满足，如果满足则输出最优解，否则返回步骤（2）。

在遗传算法的演化过程中，种群中个体的适应度值越高，被选中作为父代的概率就越大。交叉和变异操作则通过改变染色体的基因信息，产生新的染色体，并保持种群的多样性。重复执行上述步骤，直到达到预设的终止条件为止，例如迭代次数达到预定值或者满足特定的收敛准则等。最终，通过遗传算法求解出问题的最优解或近似最优解。

2.3 Istio 技术概述

Istio 是一种服务网格（service mesh）技术，提供了对微服务架构的流量管理、安全、可观测性等方面的支持[45]。Istio 最初由 Google、IBM 和 Lyft 等公司开发，并于 2017 年发布。它为 Kubernetes 和其他容器平台上的微服务应用程序提供了一些核

心功能，例如流量管理、故障恢复、安全性、监控和跟踪等[46]。Istio 使用 Envoy 作为数据面代理，通过注入 sidecar 代理来监控和管理微服务之间的网络通信。Istio 还提供了丰富的 API 和命令行工具，可以对服务的流量进行细粒度控制，例如路由、限流和负载均衡[47]。此外，Istio 还提供了灰度发布、A/B 测试和金丝雀发布等功能，可以让开发人员更加轻松地进行部署和测试。Istio 还提供了安全性方面的支持[48]，例如服务间的身份验证、授权和加密通信等。Istio 可以通过定义安全策略来保护微服务之间的通信，同时还提供了统一的审计和访问控制机制。Istio 为云原生应用程序提供了一种更加可靠、安全和可管理的方式。作为一个开源项目，Istio 正在不断发展，社区中有越来越多的贡献者加入其中，使其成为现代微服务架构中不可或缺的一部分。

如图 2.6 所示，Istio 的架构包括了数据平面（Data Plane）和控制平面（Control Plane）。数据平面由一组专用的 sidecar 代理组成，负责在应用程序之间进行网络通信和数据交换。控制平面负责配置和管理 sidecar 代理，以及控制数据平面中的服务发现、流量路由和流量管理等行为。Istio 提供了一系列的 API 和 CLI 工具，用于操作和管理控制平面的各种资源。

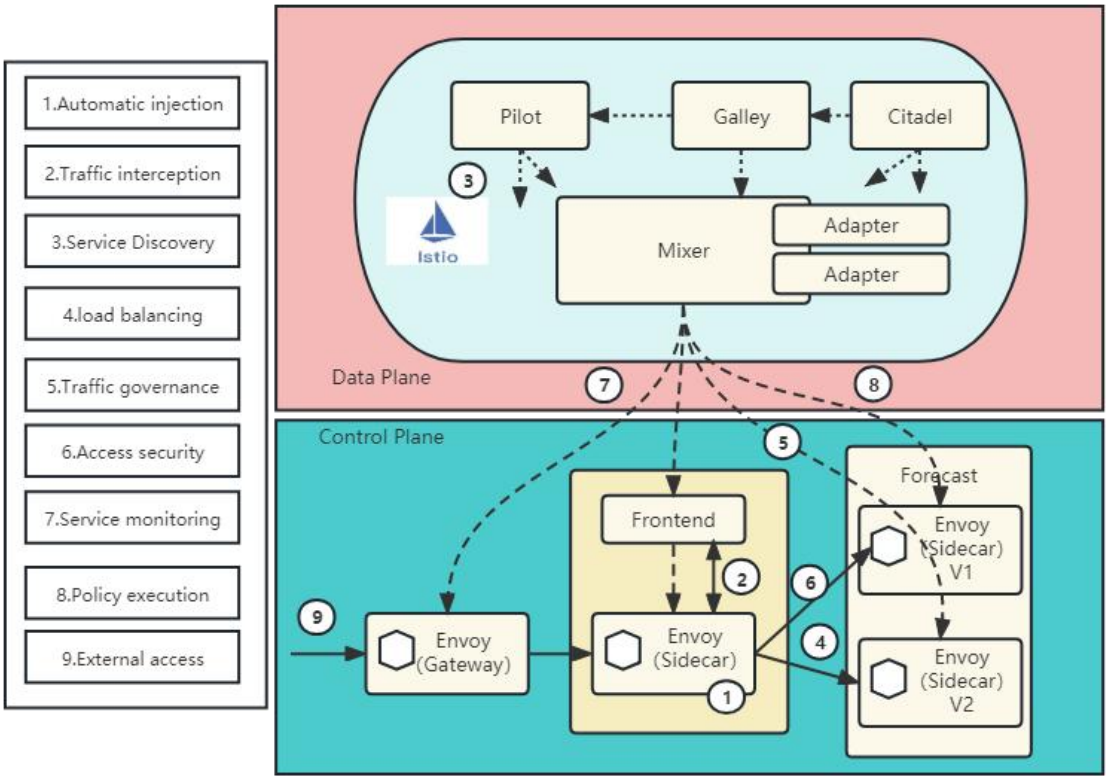

图 2.6 Istio 架构图

Istio 中，所有的网络流量都通过 sidecar 代理进行路由和控制，通过在每个应用程序实例旁部署一个 sidecar 代理，控制每个服务实例的入站和出站流量，这使得 Istio 能够提供多种服务治理功能，例如负载均衡、流量路由、故障恢复和熔断等。

2.4 Prometheus 技术概述

Prometheus 是一种开源的系统监测和警报工具，主要用于收集和记录系统的各种度量指标，包括 CPU 占用率、内存使用率、磁盘 IO 等[49]。它提供了一个灵活的查询语言 PromQL，可以让用户对收集的数据进行实时的查询和分析[50]。

Prometheus 的数据模型采用了一种时间序列的数据结构，如图 2.7 所示，每个时间序列对应一个指标的值在不同时间点的变化。Prometheus 通过 HTTP 协议暴露了一个标准的接口，使得用户可以通过自己编写的应用程序或者第三方工具来访问数据。除了数据收集和查询功能之外，Prometheus 还提供了灵活的告警机制，可以让用户根据自己的需要设置各种告警规则，当某个指标的值达到一定阈值时，Prometheus 会自动发送告警通知。同时，Prometheus 还提供了一个可视化的面板[51]，可以展示收集到的数据，并提供实时更新的图表和指标。

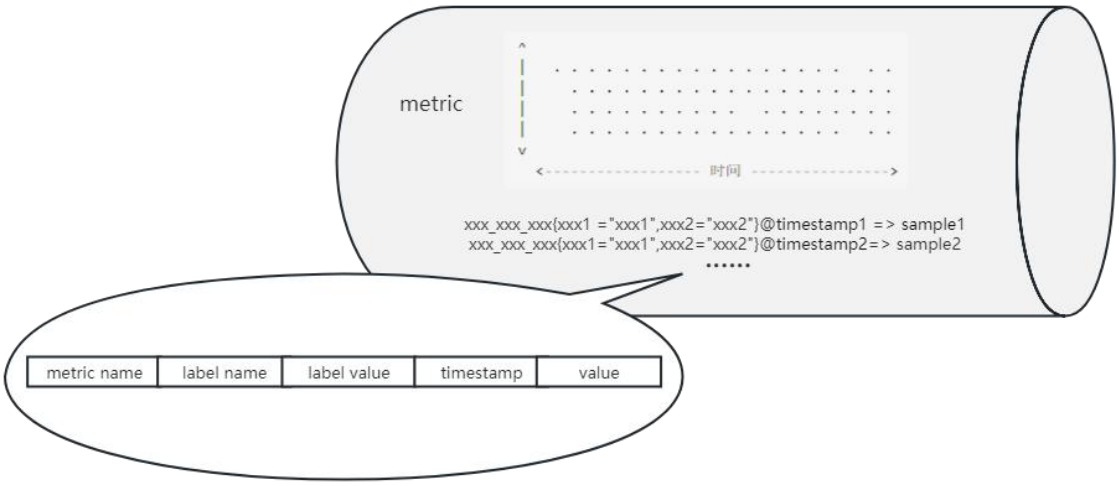

图 2.7 Prometheus 数据结构示意图

Prometheus 提供了一个强大的、可扩展的、开源的监测和警报解决方案，可以帮助用户实时监测和分析系统的各种指标，帮助用户快速识别和解决问题，提高系统的可靠性和稳定性。

2.5 本章小结

本章主要介绍了与本课题研究内容相关的一些重要理论和关键技术，详细介绍了 Kubernetes 容器编排技术的核心功能与元素，描述了 Kubernetes 中的部分核心组件及其作用，然后介绍了遗传算法，描述了算法的基本思想及算法过程，最后对本文提出的调度策略所需使用的 Istio 和 Prometheus 监控组件技术进行了介绍。

第 3 章　基于 Pod 网络改进的 Kubernetes 调度算法

本章对 Kubernetes 默认调度算法进行不足分析并提出优化方案，具体针对 Kubernetes 默认调度算法在优选阶段中对集群节点网络环境评估的缺失以及默认优选策略根据节点剩余资源情况进行优选打分的逻辑所导致的调度结果负载不均衡问题，提出了一种基于 Pod 网络改进的调度策略 TLB 以解决问题。首先对该调度策略调度模型进行了介绍，与 Kubernetes 标准调度模型对比将 TLB 调度策略的优化点进行定位，后对该调度策涉及的 Pod 真实负载获取模型、节点实时综合资源利用率和节点理想负载偏离度的概念设计进行详细介绍，最后对 TLB 调度策略调度流程与打分逻辑进行描述，完成对 TLB 调度策略的技术说明。

3.1 Kubernetes 默认调度算法

3.1.1 标准调度模型与调度策略

在 Kubernetes 集群中，资源的调度是在将应用资源包装成为 Kubernetes 的基础单位 Pod，并交由调度器进行调度管理的，如图 3.1 所示，Kubernetes 默认调度算法具有标准的 Kubernetes 调度模型，其一定具有 Predicates 预选阶段到 Priority 优选阶段到 Bind 绑定阶段的固定结构，只有经过预选和优选阶段，Kubernetes 才会生成应用 Pod 并完成调度至相应集群节点，最后完成绑定阶段。

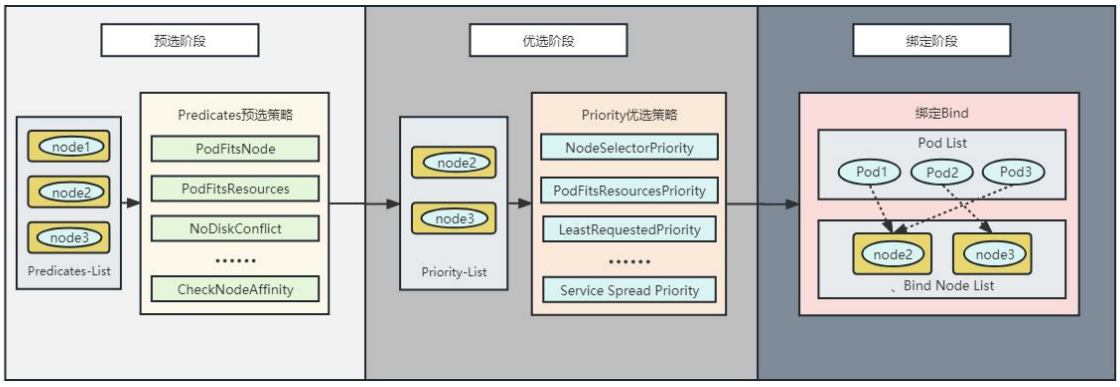

图 3.1 Kubernetes 标准调度模型

预选阶段是三大阶段的第一个阶段，在预选阶段，调度器实质上会遍历自己所记录的预选策略，并生成一个层级式的筛选的过滤环境对集群中的节点。预选策略主要用于过滤掉不符合要求的节点，从而减少后续的计算量，提高调度效率，下面是 Kubernetes 中的一些 Predicates 预选策略：

（1）PodFitsNode：检查 Pod 是否适合特定的节点，主要用于限制 Pod 只能在

特定的节点上运行，或者在节点上不允许运行特定的 Pod。

（2）PodFitsResources：检查 Pod 的资源请求是否能够满足节点的资源容量，包括 CPU 和内存等。

（3）NoDiskConflict：检查 Pod 是否会与已经存在的 Pod 在同一个节点上使用相同的存储。

（4）MatchNodeSelector：检查节点是否匹配 Pod 的标签选择器。

（5）PodToleratesNodeTaints：检查节点的污点和 Pod 的容忍是否匹配。一个节点的污点表示它不能被某些 Pod 调度使用，而 Pod 的容忍表示它可以在带有某些污点的节点上运行。

优选阶段是三大阶段的第二个阶段，优选过程则是通过打分操作对于预选阶段筛选出来的节点再一次进行择优比较，同预选策阶段类似，调度器会在优选阶段遍历自己所记录的优选策略，同样生成一个层级式的优选打分环境对集群中的节点进行优选打分操作。以下是 Kubernetes 中常用的几种 Priorities 优选策略：

（1）NodeSelectorPriority：该调度策略基于 Node 和 Pod 之间的标签匹配度来对 Node 进行排序，匹配度越高的 Node 得到的优先级越高。

（2）NodeAffinityPriority：该调度策略会对 Pod 定义的节点亲和性（Node Affinity）规则进行评估，如果规则匹配当前节点，则该节点得到更高的优先级。

（3）PodFitsResourcesPriority：该调度策略会计算每个 Node 上还剩余的资源量，并将剩余资源量最大的 Node 排在优先级最高的位置。

（4）Node Affinity Priority 这个策略会根据 Pod 与 Node 的亲和性来对 Node 进行打分。亲和性定义了哪些 Node 适合运行哪些 Pod，如 Pod 需要运行在特定的可用区或者节点类型上。

（5）Balanced Resource Allocation Priority：在负载均衡的时候，此策略会优先考虑 Pod 在所有 Node 上的资源利用率是否相同。如果相同，它将均匀地分配负载。这个策略对于需要长时间运行的 Pod 非常有用。

不同的优选策略对应着不同的打分函数，打分函数中记录着每种优选策略有着自己不同的功能性以及所设置的权重值，并可以返回节点得分，最终利用权重值与分数的加权和来计算出节点最终的得分，节点得分的计算公式如公式 3-1 所示：

$$FinalScoreNode = \sum_{i=1}^{n} weight_i * priorityFunc_i \qquad (3\text{-}1)$$

公式 3-1 中，FinalScoreNode 表示节点的最终得分，$weight_i$表示各个函数的权重值，$priorityFunc_i$表示具体的优选函数得分，通过优选策略和预选策略来优化 Pod 的调度，并为 Pod 选择最适合的节点。这些策略可以通过各种条件来衡量节点的适用性，如资源利用率、节点亲和性、节点选择器等等，从而满足不同应用场景的需求。

绑定阶段是三大阶段的最后阶段，当 Pod 绑定方案经过预选和优选两各阶段的层

层策略后，应用 Pod 就会加入到集群内部网络进行管理，同时将 Pod 和 Node 的绑定情况信息完整的保存到 Kubernetes 使用的 ETCD 数据中。当绑定成功后，绑定的节点即可根据绑定时所记录的 Pod 信息获取到 Pod 的资源清单配置文件，以此来促使应用容器的正常运行。

3.1.2 默认调度算法的不足分析

Kubernetes 默认调度算法的不足具体分析为如下两点：

（1）Kubernetes 默认调度算法会导致集群的负载分不均衡问题，其原因首先是因为默认调度算法优选阶段的所有优选策略都缺失了对网络环境的计算评估，其次默认优选策略是根据节点剩余资源情况进行优选打分，但考虑节点配置差异，高配置节点的剩余资源往往比低配置节点的剩余资源要更充沛，导致当集群中部分低配置节点的资源真实使用率远低于 Pod 的 Yaml 文件所申请的标准时，却并没有被调度更多的 Pod，造成了比较大的资源浪费，而集群中的另外一些高配置节点，其资源的真实使用率事实上已经过载，却因为配置过高无法被调度器所感知到而被继续调度 Pod，造成了集群负载不均衡问题，对集群整体的服务能力产生影响。针对这个问题，本文设计了一种基于 Pod 网络改进的调度策略 TLB，优化调度优选阶段产生的负载不均衡问题，在本文 3.2 章节中对该调度策略进行了详细阐述。

（2）Kubernetes 默认调度算法会导致调度时间开销大的问题，其原因子在于默认调度算法中优选阶段的众多优选策略对于 Pod 调度方案中所涉及的节点列表均采取了串行化的打分机制，当集群规模较大或当大量 Pod 一次性排入调度队列时，众多优选策略无法快速的制定出一个合理的分配方案，具有较大的时间开销问题。针对这个问题，本文在 TLB 调度策略的基础上设计了一种基于 IGA 改进的并行调度策略 TLB-IGA，保持 TLB 调度策略负载均衡效果的同时，优化调度优选阶段产生的时间开销大的问题，在本文第 4 章对该调度策略进行了详细阐述。

3.2 基于 Pod 网络改进的调度策略 TLB

3.2.1 TLB 调度策略调度模型

为解决 Kubernetes 默认调度算法在优选阶段中对集群节点网络环境评估的缺失以及默认优选策略根据节点剩余资源情况进行优选打分的逻辑所导致的集群负载不均衡的问题，本文提出一种基于 Pod 网络改进的调度策略 TLB，该调度策略实现了调度结果的负载均衡效果。使用 TLB 调度策略的 Kubernetes 调度模型如图 3.2 所示，与图 3.1 的 Kubernetes 标准调度模型对比，TLB 调度策略优化了默认调度算法中的优选阶段。

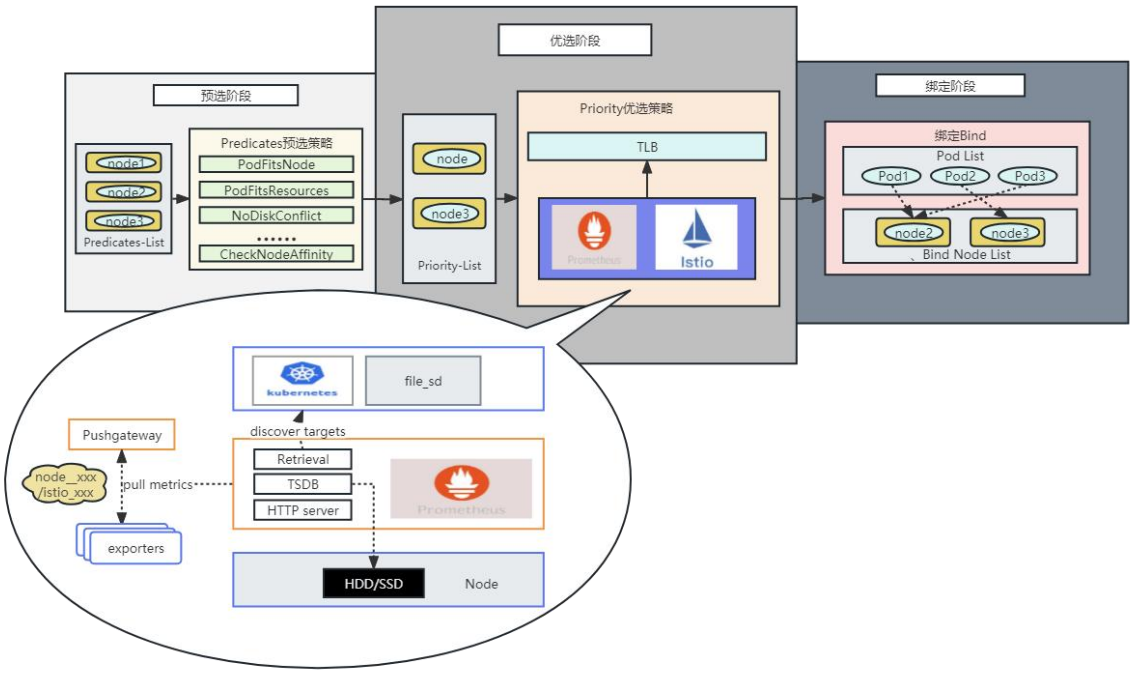

图 3.2 应用 TLB 调度策略的 Kubernetes 调度模型

TLB 调度策略通过构建 Pod 真实负载获取模型，计算节点实时综合资源利用率和节点理想负载偏离度指标参与优选打分计算，实现调度结果的负载均衡效果。本章后续章节的内容将对 TLB 调度策略涉及的 Pod 真实负载获取模型、节点实时综合资源利用率、节点理想负载偏离度以及 TLB 调度策略调度流程与打分逻辑进行详细介绍，以完成对于 TLB 调度策略的技术说明。

3.2.2 Pod 真实负载获取模型

由前文可知，Kubernetes 默认调度算法在优选阶段中对集群节点网络环境评估的缺失以及默认优选策略根据节点剩余资源情况进行优选打分的逻辑都会导致集群负载的不均衡问题。针对缺失的网络环境评估，在 Kubernetes 集群中，节点网络环境评估可以进一步划分为节点 Pod 网络占用率和节点综合网络占用率两种情况，节点 Pod 网络占用率是指节点上已部署的应用 Pod 的进出站流量数据总和占用节点带宽的比例、节点综合网络占用率则是指整个节点的实际带宽占用比例，而 Kubernetes 是基于 Pod 进行资源调度的，所以应该用节点上 Pod 占用资源的大小与节点实际已经被占用的该项资源大小的比值来反应该节点对于 Pod 资源的分配程度，因此节点网络环境评估的缺失问题更应该考虑细粒度的 Pod 网络占用率与节点实际使用的节点综合网络占用率的比值，去除相同的带宽分母，也就是 Pod 进出站流量数据总和与节点实际的上下行流量数据总和的占比。

Kubernetes 默认优选策略参考节点剩余资源情况进行优选打分的计算逻辑之所

以会造成负载不均衡问题，是因为集群中的节点配置往往存在一定差异，高配置节点的剩余资源情况往往好于低配置节点的剩余情况，这就导致高配置节点即便相对自身而言以达到一定的负载程度，却仍不断被分配 Pod 直至过载，然而使用节点真实的 CPU 和内存占用率则可以忽略节点配置差异，让每个节点的负载情况都被调度器所感知，同网络环境评估一样，对于 Kubernetes 来说，对 CPU 和内存占用情况的评估也应当使用细粒度的 Pod 占用率与节点综合占用率的比值进行计算。

综上，解决负载不均衡问题的关键在对节点 Pod 流量进出站数据以及节点 Pod 的实时 CPU 和内存占用数据进行获取。针对这点，本文提出的 TLB 调度策略通过构建一种 Pod 真实负载获取模型，获取节点的真实网络及 CPU 和内存占用情况对负载不均衡问题进行优化，下面首先对该模型进行介绍。

Pod 真实负载获取模型通过引入 Istio 和 Prometheus 监控组件，可以直接对集群中节点上已部署的应用 Pod 的实时进出站流量数据以及节点综合网络占用率进行监控获取，以计算比值来弥补默认 Kubernetes 调度算法对节点网络环境评估的缺失，并同时对节点 PodCPU 占用率和 Pod 内存占用率以及节点综合 CPU 占用率和节点综合内存占用率进行获取，代替默认优选策略使用的节点剩余资源指标以解决负载不均衡问题。Pod 真实负载获取模型架构如图 3.3 所示，监控数据统一由 Prometheus 进行收集，Istio 监控组件采用 Exporters 的方式将监控收集的节点应用 Pod 流量进出站数据数据通过暴露的 HTTP 接口让 Prometheus 定时抓取。

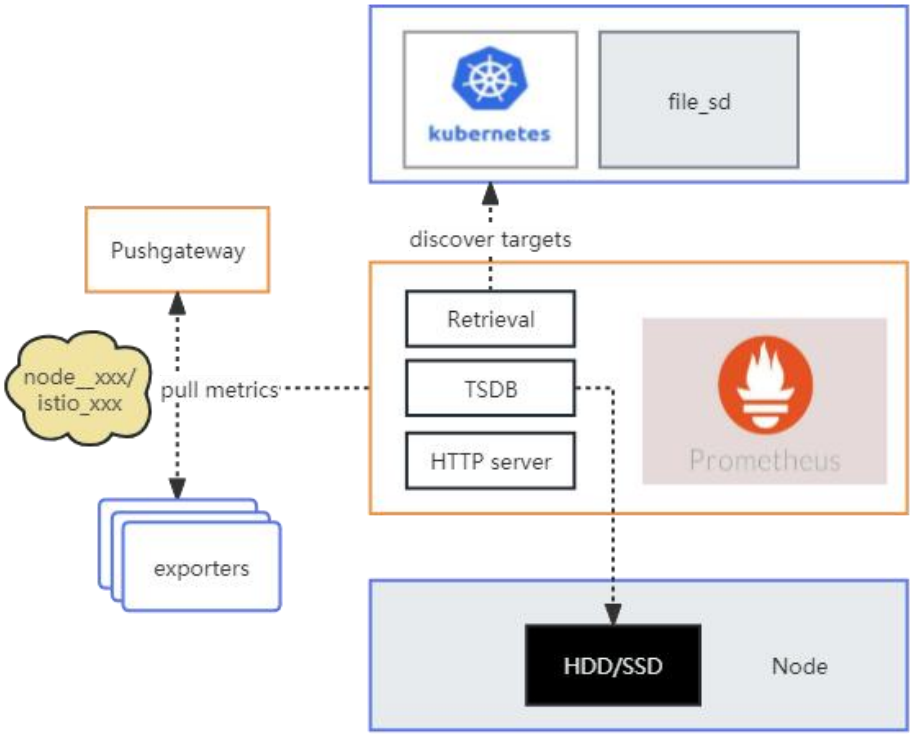

图 3.3 Pod 真实负载获取模型

这种 Exportes 的方式使得动态的节点网络环境可以由 Prometheus 集中管控，实现调度器计算与 Istio 监控 Kubernetes 节点服务的解耦，使得调度器可以直接通过 Promethes 同时获取到集群节点已有 Pod 的进出站流量数据以及节点 Pod 的 CPU 和内存占用率数据和节点的综合 CPU 和内存占用率数据。数据 3-1 展示 Pod 真实负载获取模型中，Prometheus 抓取的 istio_requests_total 的部分字段数据：

数据 3-1 **istio_requests_total 部分拉取字段数据**

{app="istio-ingressgateway",chart="gateways",connection_security_policy="unknown",destination_app="productpage",destination_canonical_revision="v1",destination_service="productpage.default.svc cluster.local",kubernetes_ namespace"istio-system"}

利用 Pod 真实负载获取模型可以对节点 Pod 的 CPU、内存占用率、进出站流量数据以及节点的综合 CPU、内存占用率、网络占用率数据进行直接精准获取，调度器可以感知节点 Pod 的实时负载情况以参与优选计算。

3.2.3 节点实时综合资源利用率

TLB 调度策略构建了一种 Pod 真实负载获取模型，该模型可以同时获取节点 Pod 的 CPU、内存占用率、进出站流量数据以及节点的综合 CPU、内存占用率、网络占用率数据。然而单一的 Pod 占用率数据与节点占用率数据的占比无法直接代表节点 Pod 的负载占用情况，如图 3.4 所示，集群节点 Pod 的任一资源占用情况都会随着时间而不断变化，所以需要设计一个集合了节点 Pod 的实时 CPU 和内存占用以及网络占用的综合考量指标来代表节点的真实负载情况，参与调度的优选计算。

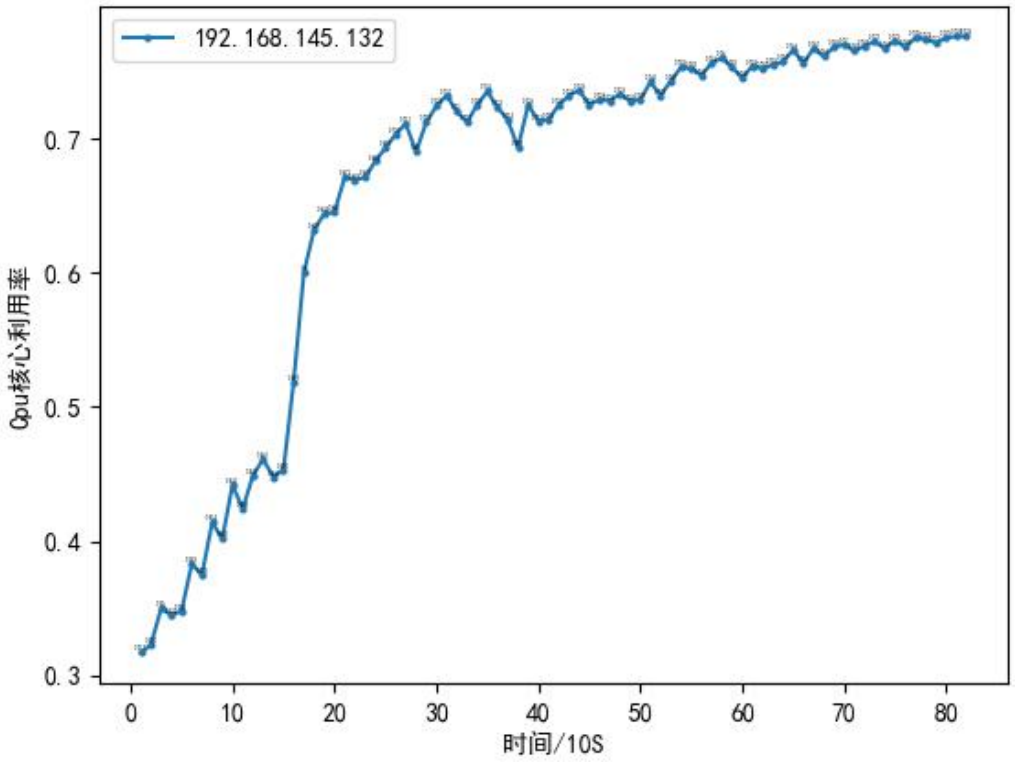

图 3.4 节点 Pod 的 CPU 核心利用率

因此，本文设计了一种节点实时综合资源利率指标以反应节点 Pod 真实负载情况。节点实时综合资源利用率的具体计算由Kubernetes集群中节点的Pod网络占用率与节点综合网络占用率比值、节点的 Pod 内存占用率与节点的综合内存占用率比值、节点的 Pod 的 CPU 占用率比值与节点的综合 CPU 占用率比值三个实时的监控比例数据来源组成，并考虑集群中节点的工作目标问题，将节点的功能性方向的权重计算设计为资源权重，加入到节点实时综合资源利用率指标的计算中，通过引入节点资源权重，将三项监控比例数据指标按照 Kubernetes 工作节点功能性方向多元的特点，动态结合起来，组成可以代表节点 Pod 实时的负载程度的新指标，即节点实时综合资源利用率。

节点实时综合资源利用率获取数据源到计算完成的完整步骤为：

（1）在 Kubernetes 集群中部署 Istio 监控服务组件和 Prometheus 监控服务组件，对 Kubernetes 集群节点上每一个应用服务 Pod 容器连接和监控，并对数据采用 Exporters 的方式将监控到的节点应用 Pod 流量进出站数据数据统一通过暴露 HTTP 接口的方式让 Prometheus 定时抓取，完成 Pod 真实负载获取模型。

（2）利用 Kubernetes 集群中部署并连接完成的 Prometheus 监控服务，对集群中所有节点 Pod 的实时内存占用情况进行监控收集。

（3）利用 Kubernetes 集群中部署并连接完成的 Prometheus 监控服务，对集群中所有节点 Pod 的实时 CPU 占用情况进行监控收集。

（4）计算 Kubernetes 集群中每一个节点的内存大小，CPU 核心数量，以及网络

带宽大小这三类单项静态资源与集群所有节点的该单项资源综合的总占比数据，作为节点类型权重。

（5）从 Prometheus 拉取到节点 Pod 的进出站流量数据与节点综合网络占用率数据，并计算总和与该节点的静态网络带宽数据占比计算节点的 Pod 网络占用数据，将 Pod 网络占用数据再与节点综合网络占用率的比值作为该节点 Pod 真实网络分配率。

（6）从 Prometheus 拉取到节点 Pod 的内存占用率数据与节点综合内存占用率数据，并计算占比作为该节点的 Pod 真实内存分配率。

（7）从 Prometheus 拉取到节点 Pod 的 CPU 占用率数据与节点综合 CPU 占用率数据，并计算占比作为该节点的 Pod 真实 CPU 分配率。

（8）将每个 Kubernetes 集群节点 Pod 真实网络分配率、Pod 真实内存分配率、Pod 真实 CPU 分配率与节点该单项资源的节点类型权重做乘积加和，并将输出结果取资源项数均值，作为最终可以代表一个 Kubernetes 集群中每一个节点 Pod 的实时资源占用情况的指标，即节点实时综合资源利用率。

节点实时综合资源利用率$L(Syn_i)$计算公式如公式 3-2 所示；

$$L(Syn_i)=\frac{W(CPU_i)*L(CPU_i)+W(Mem_i)*L(Mem_i)+W(Tra_i)*L(Tra_i)}{3}\times 100\% \quad (3\text{-}2)$$

公式 3-2 中，$L(CPU_i)$表示节点$node_i$的 Pod 真实 CPU 分配率，计算公式如公式 3-3 所示：

$$L(CPU_i)=\frac{\sum_{j=1}^{N}L(CPUPod)_{ij}}{L(CpuNode_i)}\times 100\% \quad (3\text{-}3)$$

公式 3-3 中，$L(CPUPod)_{ij}$表示的是节点$node_i$上Pod_j的 CPU 占用率，可以从 Prometheus 直接获取，$\sum_{j=1}^{N}L(CPUPod)_{ij}$表示的是节点$node_i$上的所有 Pod 的 CPU 占用率总和，$L(CpuNode_i)$表示的是节点$node_i$的综合 CPU 占用率，可以从 Prometheus 直接获取；

公式 3-2 中，$L(Mem_i)$表示的是节点$node_i$的 Pod 真实内存分配率，计算公式如公式 3-4 所示:

$$L(Mem_i)=\frac{\sum_{j=1}^{N}L(MemPod)_{ij}}{L(MemNode_i)}\times 100\% \quad (3\text{-}4)$$

公式 3-4 中，$L(MemPod)_{ij}$表示的是节点$node_i$上Pod_j的内存占用率，可以从 Prometheus 直接获取，$\sum_{j=1}^{N}L(MemPod)_{ij}$表示的是节点$node_i$上的所有 Pod 的内存占用

率总和，$L(MemNode_i)$表示的是节点$node_i$的综合内存占用率，可以从 Prometheus 直接获取；

公式 3-2 中，$L(Tra_i)$表示节点$node_i$的 Pod 真实网络分配率，计算公式如公式 3-5 所示:

$$L(Tra_i)=\frac{\sum_{j=1}^{N}（TrainPod_{ij}+TraoutPod_{ij}）}{Bandw_i*L(TraNode_i)}\times 100\% \tag{3-5}$$

公式 3-5 中，$TrainPod_{ij}$表示的是节点$node_i$上的Pod_j的进站流量数据，可以从 Prometheus 直接获取，$TroutPod_{ij}$表示的是节点$node_i$上的Pod_j的出站流量数据，可以从 Prometheus 直接获取，$Bandw_i$表示的$node_i$的静态网络带宽大小，$L(TraNode_i)$表示的是$node_i$的节点综合网络占用率，可以从 Prometheus 直接获取。

公式 3-2 中，$W(CPU_i)$、$W(Mem_i)$、$W(Tra_i)$表示的节点的单项资源权重，对于不同类型的工作节点，利用节点单项资源除以总节点该单项资源总和的比值作为节点该单项资源权重值。

公式 3-2 中，$W(CPU_i)$表示为节点$node_i$的 CPU 资源权重，计算公式如公式 3-6 所示:

$$W(CPU_i)=\frac{CPUCore_i}{CPUCount}\times 100\% \tag{3-6}$$

公式 3-6 中，$CPUCore_i$表示的是节点$node_i$的 CPU 核心数，$CPUCount$表示的是集群所有节点的 CPU 核心总数；

公式 3-2 中，$W(Mem_i)$表示为节点的$node_i$的内存资源权重，计算公式如公式 3-7 所示:

$$W(Mem_i)=\frac{Mem_i}{MemCount}\times 100\% \tag{3-7}$$

公式 3-7 中，Mem_i表示的是节点$node_i$的内存大小，$MemCount$表示的集群中所有节点的内存总和；

公式 3-2 中，$W(Tra_i)$表示的是节点$node_i$的网络资源权重，计算公式如公式 3-8 所示:

$$W(Tra_i)=\frac{Bandw_i}{BandwCount}\times 100\% \tag{3-8}$$

公式 3-8 中，$Bandw_i$表示的是节点$node_i$的带宽大小，$BandCount$表示的集群节点的总带宽大小。

获取节点实时综合资源利用率的伪代码如算法 3-1 所示：

算法 3-1 **获取节点实时综合资源利用率**

Input：CPURTO，memRTO，podNRTO，CPUWeight，memWieght，podNWeight

Output：nodeRTCRUArry

1：**For each** $node_i \in nodeList$

2： 计算$node_i$的 $L(Syn_i)$

3： 使用 nodeRTCRUArry 存储节点的$L(Syn_i)$

4：**End for**

5：**Return** nodeRTCRUArry

3.2.4 节点理想负载偏离度

TLB 调度策略利用 Prometheus 对于 Kubernetes 集群节点 Pod 的 CPU 以及内存运行情况的监控能力，计算集群所有节点的平均节点实时综合资源利用率，即可代表 Kubernetes 集群此刻负载均衡的理想状态。

平均节点实时综合资源利用率$L(Ava)$的计算公式如公式 3-9 所示

$$L(Ava) = \frac{\sum_{i=1}^{N} L(Syn_i)}{CluNum} \times 100\% \quad (3\text{-}9)$$

与每一个节点的节点实时综合资源利用率做差并取绝对值得到节点理想负载偏离度，即可表示为该节点的负载程度于平均水平的偏离度。

节点理想负载偏离度$NodeDev_i$计算公式如公式 3-10 所示：

$$NodeDev_i = \sqrt{(L(Syn_i) - L(Ava))^2} \quad (3\text{-}10)$$

节点理想负载偏离度计算基于对 Kubernetes 集群节点的节点实时综合资源利用率的计算，在计算得到节点实时综合资源利用率后，获取集群节点的节点理想负载偏离度的伪代码如算法 3-2 所示：

算法 3-2 **获取集群节点的节点理想负载偏离度**

Input：nodeRCTOArry

Output：nodeDevArry

1：计算集群$L(Ava)$

2：**For each** $node_i \in nodeList$

3： 计算$node_i$的$L(Syn_i)$

4： 计算$node_i$的$NodeDev_i$

5： 使用 nodeDevArry 存储$NodeDev_i$

6：**End for**

7：**Return** nodeDevArry

3.2.5 TLB 调度策略调度流程与打分逻辑

TLB 调度策略使用 Pod 真实负载获取模型，优化 Kubernetes 默认调度策略对于节点网络环境评估的缺失以及优选打分逻辑，以节点理想负载偏离度作为打分计算指标完成调度优选打分计算，如图 3.5 所示，TLB 调度策略调度的整体流程如下：

（1）通过筛选阶段后，利用本文 3.1.1 节点实时综合资源利用率的计算步骤计算该 Kubernetes 集群节点的节点实时综合资源利用率。

（2）利用每一个节点的节点实时综合资源利用率与集群的平均节点实时综合资源利用率计算差值绝对值，得出节点理想负载偏离度，作为 Kubernetes 调度优选阶段打分函数计算，完成节点打分操作。

（3）根据节点得分选取调度最优方案，并开始绑定应用 Pod 和节点，完成调度。

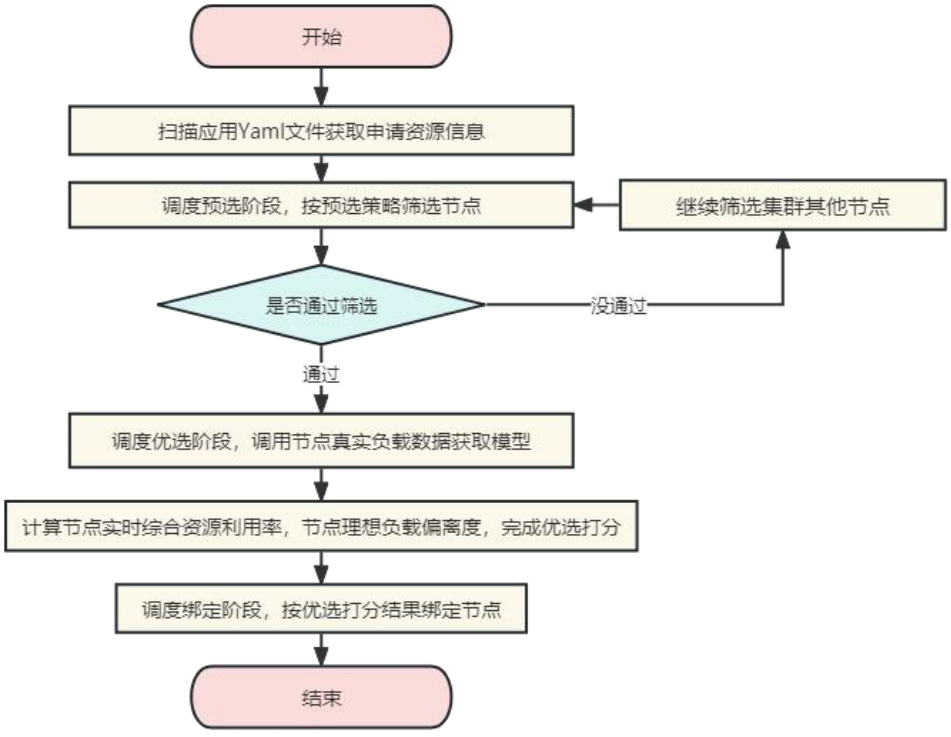

图 3.5 TLB 调度算法调度流程图

TLB 调度策略在优选打分函数中，通过遍历节点列表，拉取对应 Prometheus 和 Istio 监控组件收集的节点上 Pod 的 CPU 占用数据和内存占用数据以及节点上 Pod 的

进出站流量数据，对每个节点单独进行节点实时综合资源利用率和节点理想负载偏离度的计算，最终返回每个节点打分函数的结果，完成调度。TLB 调度策略优选打分计算的伪代码如算法 3-3 所示：

算法 3-3 **TLB 调度策略优选打分计算**

Input：cpuRTO，memRTO，podNRTO，cpuWeight，memWieght，podNWeight，nodNum

Output：nodeScoreArry

1：计算集群$L(Ava)$

2：**For each** $node_i \in nodeList$

3：　计算$node_i$的$L(Syn_i)$

4：　计算$node_i$的$NodeDev_i$

5：**End for**

6：使用 nodeDevArry 存储$NodeDev_i$

7：nodeScoreArry=nodeDevArry

8：**Return** nodeScoreArry

3.3 本章小结

本章针对 Kubernetes 默认调度算法在优选阶段中对集群节点网络环境评估的缺失以及默认优选策略根据节点剩余资源情况进行优选打分的逻辑所导致的集群负载不均衡问题提出了一种 TLB 调度策略，介绍了 TLB 调度策略调度模型，并对该调度策略构建的 Pod 真实负载获取模型的功能和模型架构进行了详细阐述，并描述了应用该模型设计计算的节点实时综合资源利用率和节点理想负载偏离度两个指标的计算流程，最后对 TLB 调度策略的应用调度流程和优选打分函数计算逻辑进行了详细阐述。

第 4 章　基于 IGA 改进的并行调度策略 TLB-IGA

本章针对 Kubernetes 默认调度算法串行化调度优选打分逻辑所导致的调度时间开销大的问题，设计并使用了一种基于模拟退火思想改进的自适应遗传算法 IGA 构建一种并行化调度优选模型，对本文提出的 TLB 调度策略做出了调度时间开销上的优化，提出了一种基于 IGA 改进的并行调度策略 TLB-IGA，该调度策略在实现 TLB 调度策略负载均衡效果的同时，解决了调度时间开销大的问题。本章首先通过 TLB-IGA 调度策略调度模型的介绍，对 TLB-IGA 调度策略的优化点进行定位，后对 TLB-IGA 调度策略涉及的并行化调度优选模型和基于模拟退火思想改进的自适应遗传算法 IGA 的概念设计进行了详细阐述，最后对 TLB-IGA 调度策略调度流程与打分逻辑进行了介绍，完成对 TLB-IGA 调度策略的技术说明。

4.1 TLB-IGA 调度策略调度模型

为解决 Kubernetes 默认调度算法优选阶段中的串行化调度优选打分逻辑所导致的调度时间开销大的问题，本文提出了一种基于 IGA 改进的并行调度策略 TLB-IGA，该调度策略实现 TLB 调度策略负载均衡效果的同时，降低了调度的时间开销。应用 TLB-IGA 调度策略的 Kubernetes 调度模型如图 4.1 所示，与图 3.1 的标准 Kubernetes 模型对比，TLB-IGA 调度策略通过结合 Pod 真实负载获取模型及并行化调度优选模型，优化了优选阶段的数据评估以及打分逻辑。

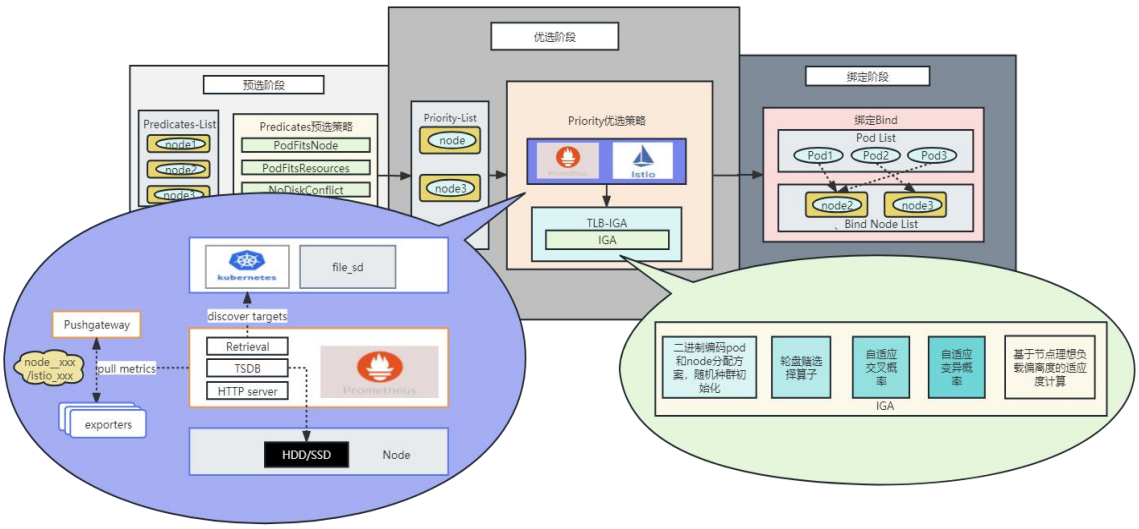

图 4.1 应用 TLB-IGA 调度策略的 Kubernetes 调度模型

TLB-IGA 调度策略通过构建一种并行化调度优选模型，结合 Pod 真实负载获取模型，优化优选阶段众多调度策略串行化节点打分逻辑所带来的时间开销问题的同时

实现调度结果的负载均衡效果。本章后续内容将对 TLB-IGA 调度策略涉及的并行化优选模型、基于模拟退火思想改进的自适应遗传算法 IGA 以及 TLB-IGA 调度策略调度流程与打分逻辑进行详细阐述，以完成对 TLB-IGA 调度策略的技术说明。

4.2 并行化调度优选模型

由前文可知，Kubernetes 默认调度算法在优选阶段串行化的优选打分机制存在调度时间开销大的问题，这是因为默认调度算法的众多优选策略对于 Pod 调度方案组成的优选节点列表均采取了串行化的打分机制，当集群规模较大或当大量 Pod 一次性排入调度队列时，调度器需要为预调度方案组成的节点列表顺序执行优选打分计算，以至于众多优选策略无法快速的制定出一个合理的分配方案，具有较大的时间开销问题。而本文提出的 TLB 调度策略保留了这一串行化的优选打分机制，为了达到调度结果的负载均衡，在串行化优选打分机制的基础上，使用了 Pod 真实负载获取模型，为节点列表再次添加了拉取 Prometheus 监控数据的逻辑，如图 4.2 所示，有几种分配方案就会出现几次数据拉取和优选打分计算的逻辑，数据传输与优选打分计算两种时间开销的串行叠加更是较为巨大而不可接受的。

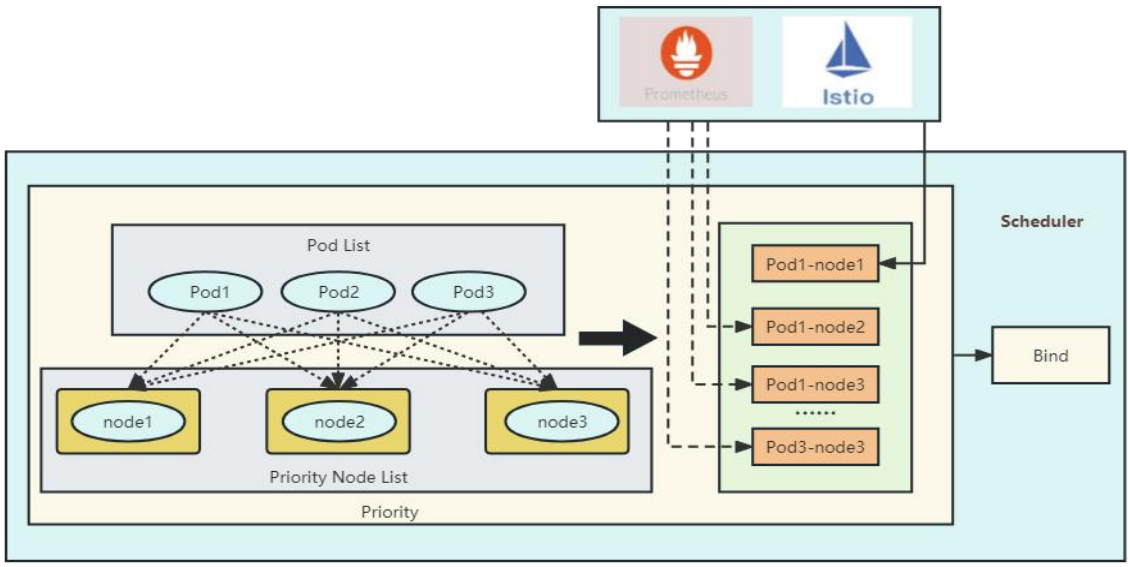

图 4.2 串行化优选打分机制示意图

针对这一问题，TLB-IGA 调度策略设计并使用了一种基于模拟退火思想改进的自适应遗传算法 IGA，构建并行化调度优选模型将调度方案映射为 IGA 的种群个体，进行所有调度方案集合的整体迭代。如图 4.3 所示，并行化调度优选模型结合 Pod 真实负载获取模型，集中式的对调度方案中涉及的多个节点拉取 Prometheus 捕获的实时 Pod 负载数据并计算优选打分，降低了数据传输和优化打分计算的执行频率，使得调度器不必再对节点列表顺序且单体的执行打分计算工作，迅速寻找到让调度结果更

趋近于 Kubernetes 整体负载均衡的方案，降低优选阶段的时间开销。

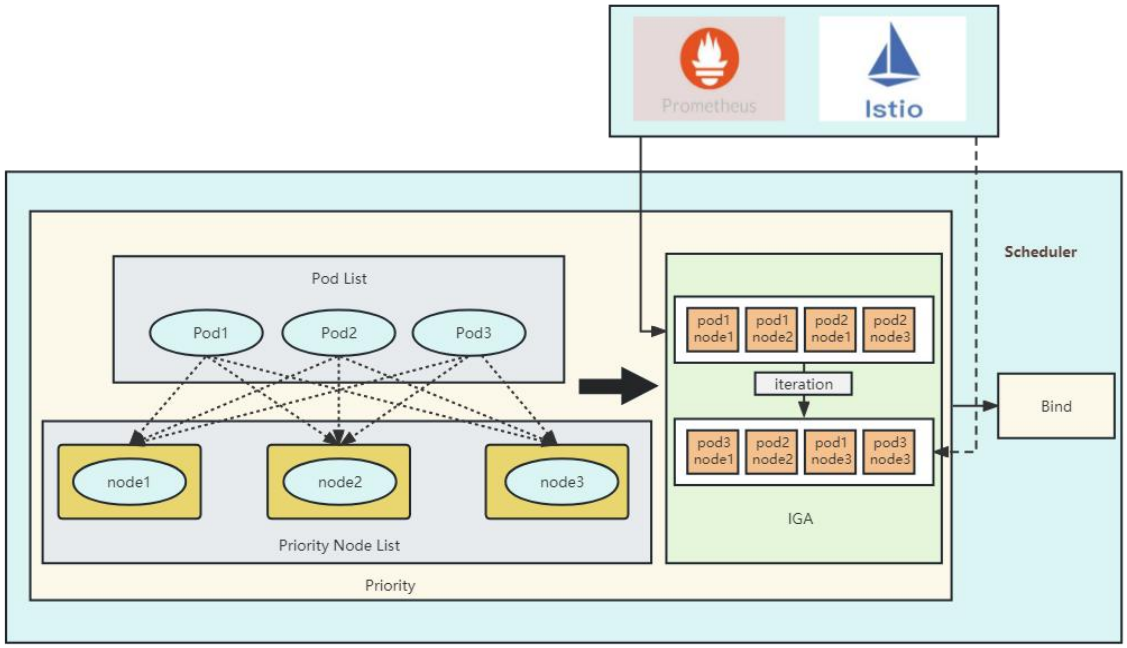

图 4.3 改进后的并行调度优选模型

并行化调度优选模型的核心计算逻辑是利用基于模拟退火思想改进的自适应遗传算法对集群调度方案的整体集合进行迭代。本章后续章节对基于模拟退火思想改进的自适应遗传算法 IGA 的算法流程与具体算子设计展开详细描述。

4.3 基于模拟退火思想改进的自适应遗传算法 IGA

4.3.1 改进遗传算法的必要性与改进方向

TLB-IGA 通过构建并行化调度优选模型，利用基于模拟退火思想改进的自适应遗传算法 IGA 对集群调度方案的整体集合进行迭代，以解决 Kubernetes 默认调度算法串行化调度优选打分逻辑的时间开销问题。标准遗传算法在 Kubernetes 调度问题上并不适合直接使用，其原因在于 Kubernetes 调度问题需要在 Pod 和集群节点的绑定方案组成的庞大的解空间中快速寻找出最优调度方案，而标准遗传算法在这个过程中会因为种群个体适应度相差不大，而很难产生最优个体，并且，产生的最优个体往往被差的个体所包围，限制了优秀基因的迭代留存，从而影响了算法的收敛速度，降低了一定的寻优能力。所以需要针对调度问题对标准遗传算法在收敛速度上的表现做出改进。

针对收敛速度问题，使用自适应遗传算法的改进设计则可以对这类问题进行解决，其关键点在于设计自适应的交叉概率和变异概率，自适应的概念即使概率能随适应度

的计算值变化而随之改变，以跳出局部最优情况，同时也更利于优良个体的生存，在保持群体多样性的同时，保证遗传算法的收敛性。所以本文结合在云 Kubernetes 调度问题上将模拟退火思想[52-53]中的扰动机制结合到遗传算法中，针对交叉算子和变异算子对标准遗传算法进行改进。

改进后的自适应遗传算法的流程图如图 4.4 所示，在算法迭代过程中可以自适应的对交叉概率和变异概率进行计算，以保证良好基因可以更好的留存。

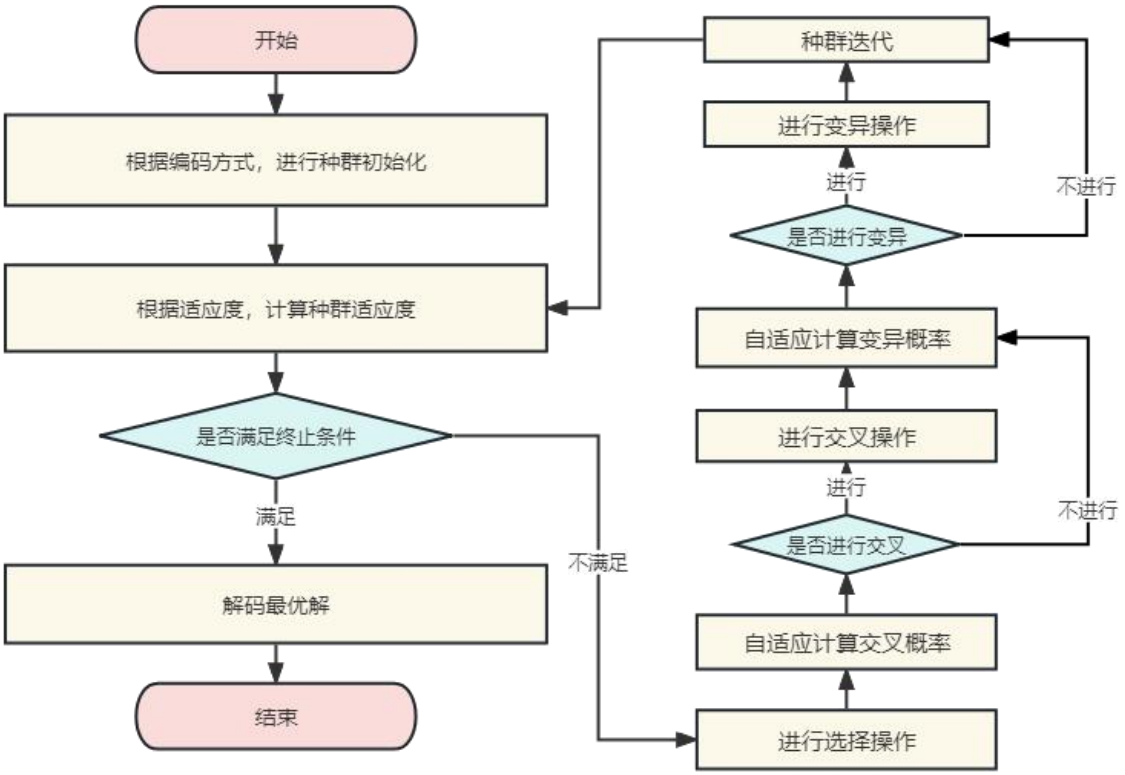

图 4.4 自适应遗传算法的算法流程

4.3.2 编码方案设计

Kubernetes 调度问题本质上来说是一种绑定选择问题，可以将应用 Pod 与 Kubernetes 集群节点的调度绑定选择视为一种 01 背包问题模型，如图 4.5 所示，本文根据 01 背包问题的核心解法模型与 Kubernetes 问题结合来设计遗传算法的编码核心要素模型。

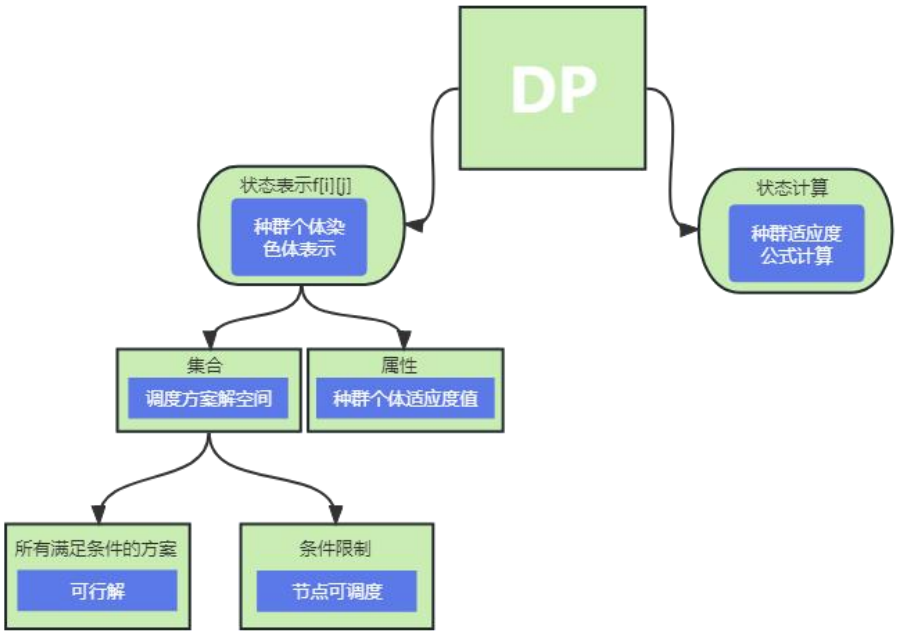

图 4.5 编码核心要素

采用二进制编码方案可以更加直观的反应 Pod 和节点的具体调度绑定方案，利用 0 和 1 两个数字，结合二维数组对种群个体的基因进行存储，使用二维数组的行来表示待调度的应用 Pod，用列来表示集群节点，二维数组中的某个元素的值则表示应用 Pod 与集群节点的一种匹配结果。用 Chromosome[i][j]=1 表示编号为 i 的应用 Pod 与编号为 j 的集群节点进行了绑定，反之 Chromosome[i][j]=0 则表示应用 Pod 与集群节点没有进行绑定。如表 4-1 所示，Chromosome[2][2]=0 则表示编号为 2 的待调度应用 Pod 没有与编号为 2 的 Kubernetes 集群节点 node2 进行调度绑定，根据表中内容可知，此时种群个体的染色体可以表示为{101001，000111，000101，101010，011010，111011}，其中元素的顺序则表示待调度应用 Pod 列表中个体的先后顺序，解码此二进制编码即可得到待调度应用 Pod 与 Kubernetes 集群节点的绑定结果，即此染色体中 101001 段表示 1 号待调度应用 Pod 被调度到了 Kubernetes 集群的 1、3、6 号节点上，染色体 000111 则表示 2 号待调度应用 Pod 被调度到了 Kubernetes 集群的 4、5、6 号节点上，后续四位元素同理。

表 4-1 染色体数组取值表

Chromosome	Node1	Node2	Node3	Node4	Node5	Node6
Pod1	1	0	1	0	0	1
Pod2	0	0	0	1	1	1
Pod3	0	0	0	1	0	1
Pod4	1	0	1	0	1	0
Pod5	0	1	1	0	1	0
Pod6	1	1	1	0	1	1

4.3.3 种群初始化方案设计

种群初始化是遗传算法真正开始的第一步，种群的初始化结果实际上对遗传算法的收敛性和最终迭代结果也非常的重要，不仅如此，种群初始化的优良同时还会影响遗传算法来寻找优良种群个体的速度，高质量的初始种群个体对解决遗传算法搜索能力差的问题有很好的帮助作用。现有遗传算法种群初始化方案根据不同的分类标准归纳总结分成了三类，分别是随机性(Randomness)、组合性(Compositionality)和通用性(Generality)，三类遗传算法种群初始化方案和其细粒度的划分如图 4.6 所示。

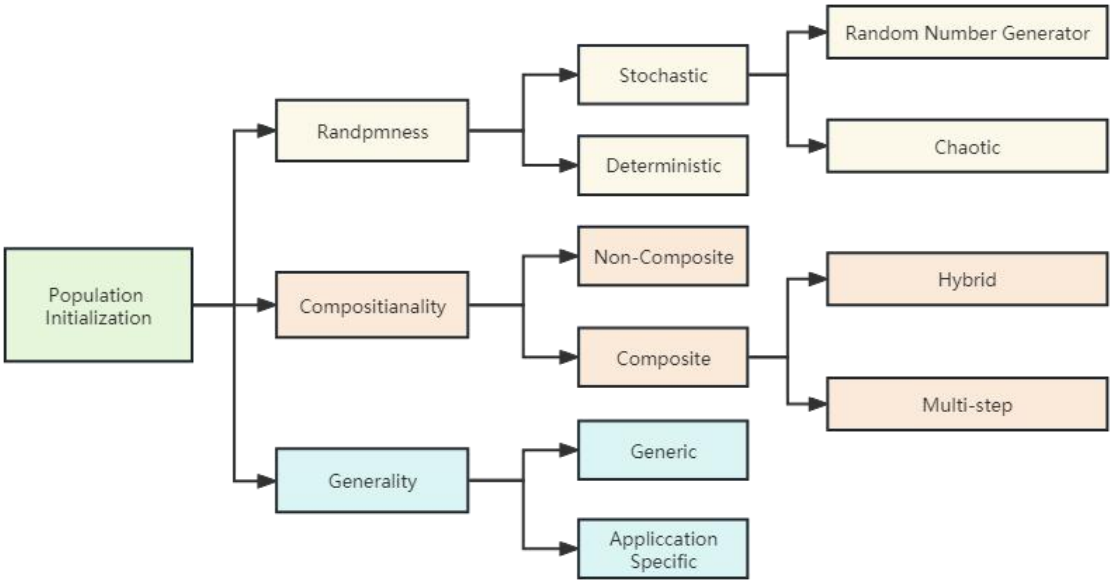

图 4.6 遗传算法种群初始化方案分类

从为了更好的利用遗传算法生物种群的随机性，以及对于应用 Pod 与集群节点绑定方案选择的两个角度考虑，本文选择使用随机初始化方案。

4.3.4 基于节点理想负载偏离度的适应度计算设计

在遗传算法中，种群的进化程度通常由适应度值来表示，适应度函数也可以称之为评价函数，是一个用来评判种群个体优劣程度的指标，遗传算法在种群的搜索进化过程中不再需要其他的外部信息，仅需要用适应度函数的结果对种群个体的优劣进行判断，并以此作为后续迭代的操作依据。

TLB-IGA 调度策略使用节点理想负载偏离度指标参与适应度函数的计算，当一种调度方案中包含的调度节点的理想负载偏离度值总值越大，代表该方案距离集群理想负载均衡状态的趋向性越差，则更应该为该套方案中包含的调度节点计算更低的优选得分，反之如果理想负载偏离度的数值越小，则代表该种方案可以让集群更接近负

载均衡状态，则该种方案中的节点应该得到更高的分数，这种反比关系与种群朝着适应度高值进化的正比关系正好相反，所以需要对节点理想负载偏离度计算公式做数学求解最小结果处理并将其映射为0到1取值范围的数学模型以适配遗传算法适应度公式计算要求，同时满足 Kubernetes 集群与理想负载不断接近的目标，基于节点理想负载偏离度的适应度算子$Fit(NodeDev_i)$计算公式如公式 4-1 所示：

$$Fit(NodeDev_i) = \frac{1}{(\sum_{i=1}^{N} NodeDev_i)^2 + 1} \times 10 \tag{4-1}$$

4.3.5 选择算子设计

遗传算法中，选择算子的作用即为种群进化迭代做出基因的自然选择过程，选择一个良好的选择算子设计方案，可以对种群进化的全局收敛效果起到正向的加速作用，并且可以对种群优良个体的良好基因进行最大程度的遗传。

目前被广泛使用的遗传算法选择算子设计方案有很多种，其中轮盘赌选择、锦标赛选择、随机遍历抽样、局部选择、截断选择等。本文选择使用轮盘赌选择算子，又称为比例选择方法，该选择算子的基本思想为各个个体被选中的概率与其适应度大小成正比，即种群个体的适应度数值越高，则该种群个体的基因被选择的概率也高，本文中轮盘赌选择算子$\delta(Fit(NodeDev_i))$的计算公式如公式 4-2 所示

$$\delta(Fit(NodeDev_i)) = \frac{Fit(NodeDev_i)}{\sum_{i=1}^{N} Fit(NodeDev_i)} \tag{4-2}$$

随机产生 6 个种群个体，对每个种群个体进行基于节点理想负载偏离度的适应度计算，并使用轮盘赌算子计算其各自的选择概率，如表 4-2 所示，种群个体的适应度值越大其选择概率就越高。6 个种群个体是由二进制编码映射的应用 Pod 与节点的调度方案，因此，当一种调度方案距离集群理想负载均衡状态越接近时，其代表的种群个体适应度值越大，则该方案被调度器选择的概率越高，以此达到种群不断朝着集群负载均衡的方向进化。

表 4-2 种群个体选择概率表

个体	染色体	适应度	选择概率	累计概率
1	100000	3	0.09375	0.09375
2	010000	6	0.1875	0.28125
3	001000	4	0.125	0.40625
4	000100	7	0.21875	0.625
5	000010	3	0.09375	0.71875
6	000001	9	0.28125	1

如图 4.7 所示，将表 4-2 中 6 个种群个体按照各自的适应度值大小生成一个公平转盘，6 个种群个体将转盘划分为 6 个连续区域，每个区域根据适应度数值的大小对应生成不同的面积，转盘中区域与转盘总面积的占比即代表种群个体的选择概率，在进行选择操作的时候，即在该轮盘赌示意图中拨动公平指针，指针完全按照转盘区域概率随机停留在 6 个体染色体所代表的区域之一，即代表此次选择操作的结果。

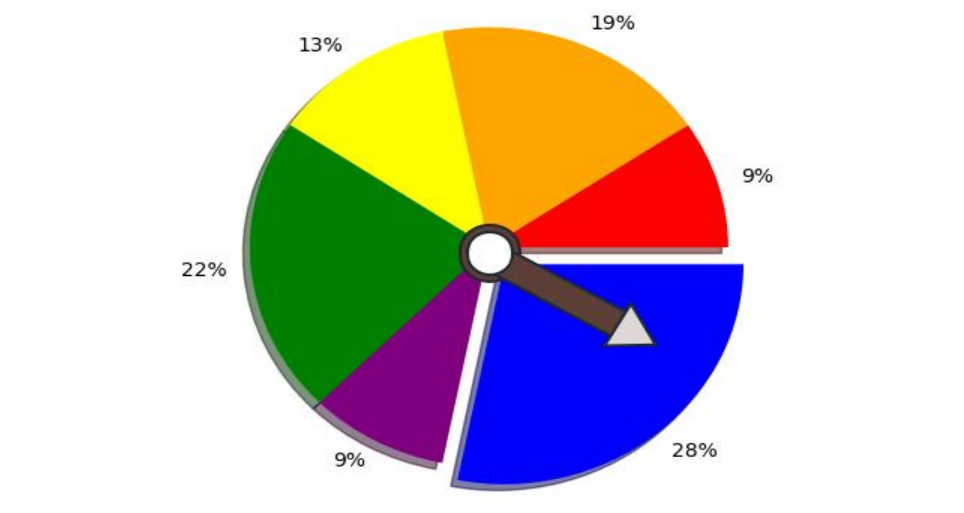

图 4.7 轮盘赌示意图

4.3.6 基于模拟退火思想的自适应交叉算子设计

模拟退火思想源于固体的退火过程，通过在搜索解空间的过程利用扰动机制，创造概率突跳，避免陷入局部最优解问题。将模拟退火思想的扰动机制结合到遗传算法的交叉运算中，设计出自适应的交叉概率，以自适应概率为判断基准，相互交换某两个个体之间的部分染色体，让种群中较优个体以较低概率进行交叉，增加种群中“弱势”个体的交叉，即其交叉概率应与选择个体的适应度值成反比，以此提高种群的多样性，减小算法陷入局部最优解的可能性。

自适应交叉概率 P_c 的详细设计公式如公式 4-3 所示：

$$P_c = \frac{1}{a*T*(\frac{Fit(NodeDev_1)+Fit(NodeDev_2))}{2Fit(NodeDev_{avg})})^2+1} \quad (4\text{-}3)$$

公式 4-3 中，$\mathrm{Fit(NodeDev_1)}$和 $\mathrm{Fit(NodeDev_2)}$表示选取的两个个体的适应度值，$\mathrm{Fit(NodeDev_{avg})}$表示种群此时的适应度平均值，a 为退火系数，$T$ 为算法当前迭代温度，通过两个个体的适应度与适应度均值的比较，通过比值的形式判断所选的两个个体是否为较为优良的个体从而影响其是否交叉，形成一个 0 到 1 开区间的自适应交叉概率值域，以避免陷入局部最优解。

在交叉概率自适应得到值后，本文选用多点交叉，随机创建与种群个体染色体编

码串长度相等的屏蔽字串 $S = s_1 s_2 s_3 ... s_n$,其中 n 为种群个体染色体编码串长度，由 s_i 的值决定是该位置上的染色体字符是否需要进行交换，如图 4.8 所示，当通过了自适应交叉概率计算，并决定进行交叉时，使用 $S = 010010$ 的屏蔽字串来进行交叉运算后，配对的个体发生了染色体交叉。

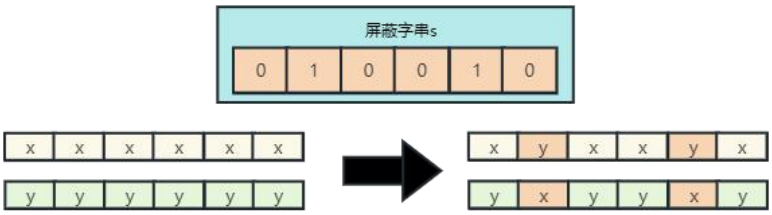

图 4.8 交叉示意图

4.3.7 基于模拟退火思想的自适应变异算子设计

变异操作是以一定概率改变染色体上的基因,产生略微不同的个体，结合模拟退火思想，设计计算自适应的变异概率，自适应变异概率 P_m 的计算公式设计如公式 4-4 所示：

$$P_m = \frac{1}{a*T*(\frac{Fit(NodeDev_i)}{Fit(NodeDev_{avg})})^2+1} \quad (4\text{-}4)$$

公式（4-4）中，$\text{Fit(NodeDev}_i\text{)}$表示当前选择的种群个体适应度值，利用种群个体与均值的比较，判断其是否为较优个体，从而判断是否需要变异，如果为较优个体则该公式可以尽可能保留其基因，反之同理。

经过变异概率的选择后，即完成决定是否变异，还需要设置一个变异操作方案，由于本文采用二进制编码方案对集群调度方案进行编码，其变异操作只需要进行非运算即可，所以本文采用均匀变异的方式来产生新个体，随机创建与种群个体染色体编码串长度相等的屏蔽字串 $S = s_1 s_2 s_3 ... s_n$,其中 n 为种群个体染色体编码串长度，由 s_i 的值决定是该位置上的染色体字符是否需要进行变异操作。如图 4.9 所示，使用 $S = 010010$ 的屏蔽字串来进行变异操作后，染色体发生了变异。

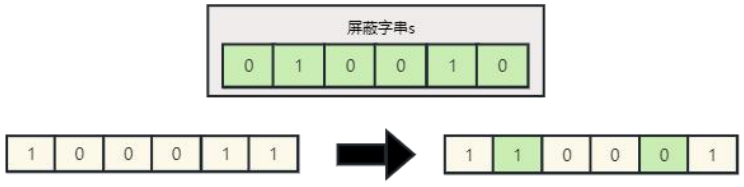

图 4.9 变异示意图

4.3.8 终止条件设计

遗传算法中，生物种群要进行连续不断的进化，以保证基因不停的靠近适应度的理想值，所以对于遗传算法的实际应用中一定要设计遗传算法的终止条件，保证对于遗传算法的应用不至于无法停止。

本论文为基于模拟退火思想改进的自适应遗传算法的共设置了两个终止条件。以 Kubernetes 集群负载均衡的调度目的出发，本论文设计了一个相对较大的适应度值终止值，即一个相对较小的节点理想负载偏离度值，如种群个体的适应值已比终止条件设定值还大，则代表该种群个体已经足够优良，即寻找到了最佳应用 Pod 与 Kubernetes 集群节点的调度方案，即可完成打分操作。以调度过程的时间开销为调度目的出发，本论文设计了自定义的遗传算法最大迭代次数，以防止种群在无限接近终止适应值的情况下超数迭代。两个终止条件只要满足其中的一个，则终止遗传算法的迭代，完成此次算法的运行。适应度终止值和最大迭代次数的具体数值在本文 5.1.2 章节中与退火系数和初始温度等参数于实验共同进行展示。

4.4 TLB-IGA 调度策略调度流程与打分逻辑

TLB-IGA 调度策略使用基于模拟退火思想改进的自适应遗传算法，优化 TLB 调度策略中对于优选阶段的串行式获取实时数据来进行优选打分的逻辑，利用基于节点理想负载偏离度的适应度计算，对调度分配方案进行整体的迭代，当满足任一终止条件时，完成最终迭代，解码种群个体并将分配方案中节点的单体负载偏离度值作为 Kubernetes 调度优选阶段的打分函数的计算结果，将应用 Pod 分配至对于节点并完成优选阶段。如图 4.10 所示，应用 TLB-IGA 调度策略的 Kubernetes 调度流程如下：

（1）扫描待调度应用 Pod 的 Yaml 清单文件，调度器获取申请资源信息。

（2）通过预选阶段，按照预选策略层级式筛选节点列表。

（3）进入优选阶段，根据本文 3.2.3 章节中记录的节点实时综合资源利用率计算步骤，计算出 Kubernetes 集群节点的节点实时综合资源利用率与节点理想负载偏离度。

（4）对 Pod 和节点的分配方案做二进制编码，随机初始化种群个体。

（5）IGA 开始迭代，采用基于节点理想负载偏离度的适应度计算，轮盘赌选择算子、自适应交叉、自适应变异操作，直至终止迭代，完成种群的最终进化。

（6）将最终的种群个体适应值解码映射的节点负载偏离度值作为 Kubernetes 集群此次调度优选环节的最终打分函数的运行结果，解析方案节点并完成打分，完成调度。

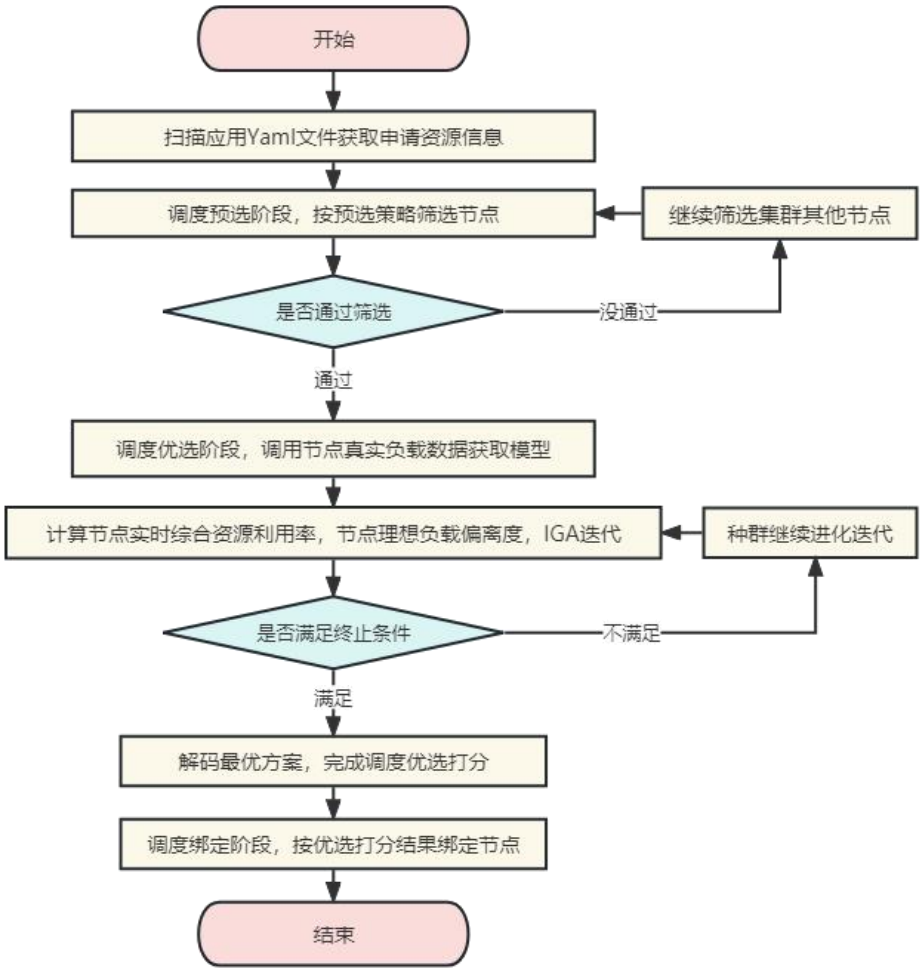

图 4.10 TLB-IGA 调度策略调度流程图

TLB-IGA 调度策略使用并行化调度优选模型在 IGA 迭代过程中集中式的拉取节点实时负载，并完成最优方案的寻找，在算法最终迭代完成后，打分函数将分配方案中节点的单体负载偏离度值作为节点的优选得分结果，其余未进入分配方案的节点此次调度优选打分函数的结果置 0，表示在此次调度中未被选中。TLB-IGA 调度策略优选打分函数计算的伪代码如算法 4-1 所示：

算法 4-1 **TLB-IGA 调度策略优选打分计算**

Input：cpuRTO，memRTO，podNRTO，cpuWeight，memWieght，podNWeight，popNum，fireT，fireA

Output：nodeScoreArry

1：随机初始化 popNum 个种群个体

2：使用 popList 存储种群

3：**While** terminationCondition == false **Do**

4：　解码 popList 映射 Node

5： 使用 popNodeList 存储 popList 映射的 Node

6： 计算 popList 的$L(Ava)$

7： **For each** $node_i \in nodeList$

8： 计算$node_i$的$L(Syn_i)$

9： 计算$node_i$的$NodeDev_i$

10： 计算 Fit($\mathrm{NodeDev_i}$)

11： **End for**

12： 计算$\delta(Fit(NodeDev_i))$

13： 自适应计算P_c，P_m

14： 对 popList 进行 Select_Crossover_Mutation 操作

15：**End while**

16：解码 popList 映射 Node

17：使用 popNodeList 存储 popList 映射的 Node

18：**For each** $node_i \in nodeList$

19： **If** $node_i \in popNodeList$

20： 使用 nodeScoreArry 存储$node_i$的$NodeDev_i$

21： **Else** 使用 nodeScoreArry 存储 0

22： **End if**

23：**Return** nodeScoreArry

4.5 本章小结

针对 Kubernetes 默认调度算法优选阶段中的串行化调度优选打分逻辑所导致的调度时间开销大的问题提出了一种基 IGA 改进的并行调度策略 TLB-IGA，对该调度策略调度模型进行了介绍，并详细阐述了该调度策略所构建的并行化调度优选模型以及其核心算法 IGA,对 IGA 的算法流程以及算子设计进行了详细阐述，最后介绍了 TLB-IGA 调度策略的执行流程和及优选打分函数的计算逻辑。

第 5 章　实验设计与结果分析

本章围绕本文提出的 TLB-IGA 调度策略和 TLB 调度策略建立三个实验，验证 TLB 调度策略在解决负载不均衡问题上的有效性和可行性、TLB-IGA 调度策略在解决负载不均衡问题上的有效性和可行性、TLB-IGA 调度策略在同时解决负载不均衡问题和调度时间开销大的问题上的有效性和可行性、TLB-IGA 调度策略相比于其他没有使用本文 Pod 真实负载获取模型的负载均衡调度策略在提升负载均衡效果上的优越性、TLB-IGA 调度策略在提升集群整体提供服务的能力上的有效性和可行性、TLB-IGA 调度策略使用的改进遗传算法 IGA 相比于标准遗传算法 SGA 在 Kubernetes 调度问题寻优能力上的优越性。

5.1 基于 TLB-IGA 调度策略的资源调度实验

5.1.1 实验目标

由本文前文可知，本文提出的 TLB 调度策略使用了 Pod 真实负载获取模型获取 Pod 负载数据以实现调度结果的负载均衡，本文提出的 TLB-IGA 调度策略使用并行化调度优选模型结合 Pod 负载数据获取模型，在 TLB 调度策略的基础上实现负载均衡效果的同时降低调度的时间开销。本实验部分从以下三点进行验证本文提出的调度策略的优化效果，具体验证目标如下：

（1）TLB 调度策略与 TLB-IGA 调度策略是否可以实现负载均衡的调度结果。

（2）TLB-IGA 调度策略是否可以在保持负载均衡的同时，相对于 Kubernetes 默认调度策略和 TLB 调度策略降低调度时间开销。

（3）TLB-IGA 调度策略在调度结果的负载均衡程度上，相比于其他未使用 Pod 真实负载获取模型的负载均衡调度策略是否具有提升。

通过对以上三点的验证结果分析来论证得出以下四个目标结论：

（1）TLB 调度策略在解决负载不均衡问题上具有有效性和可行性。

（2）TLB-IGA 调度策略在解决负载不均衡问题上具有有效性和可行性。

（3）TLB-IGA 调度策略在解决负载不均衡问题和调度时间开销大的问题上同时具有有效性和可行性。

（4）TLB-IGA 调度策略在提升调度结果的负载均衡效果上，相比于其他未使用 Pod 真实负载获取模型的负载均衡调度策略具有一定优越性。

5.1.2 实验设计与环境搭建

实验搭建了使用 TLB 调度策略的 Kubernetes 集群和使用 TLB-IGA 调度策略的

Kubernetes 集群以及使用 Kubernetes 默认调度策略的 Kubernetes 集群的共三种集群，分别在三种集群中对同一组应用使用各自的调度策略进行调度实验。对调度过程消耗的时间以及调度完成后集群节点的负载数据进行记录，建立 TLB 调度策略进群和 TLB-IGA 调度策略集群以及 Kubernetes 默认调度策略集群的调度结果对照实验。

本实验中，三种 Kubernetes 集群所应用的软件环境以及系统基础环境版本如表 5-1 所示。

表 5-1 基础环境版本表

Basic Environment	Version
Guest OS	CentOS7
Kubernetes	V1.23.0
Java Version	JDK 1.8
Golang Version	V1.16.6
Prometheus-Operator	V0.59
Istio	V1.15.0

本实验部分的设施基础使用了十五台虚拟机，如图 5.1 所示搭建了三个 Kubernetes 集群，三个 Kubernetes 集群各自使用五个节点，分别为一个 Master 和四个 Node 工作节点，且三个集群包括 master 在内的共十五台节点均已删除污点，且经过测试应用 Pod 均可调度到其上。

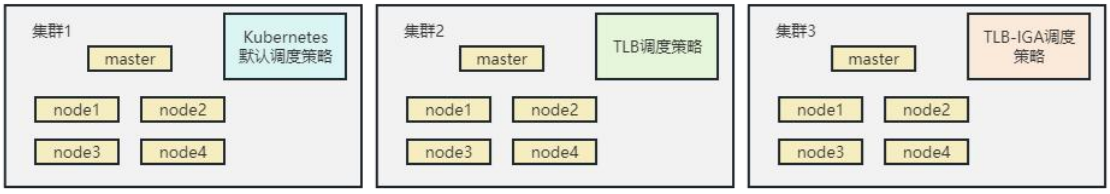

图 5.1 集群搭建示意图

三个 Kubernetes 集群节点的物理配置相同，如表 5-2 所示。

表 5-2 集群节点物理配置表

Node	CPU Core	Memory	Harddisk	Bandwidth
Master	2 核	8GB	70GB SSD	7Mps
Node1	2 核	4GB	70GB SSD	3Mbps
Node2	2 核	4GB	70GB SSD	3Mbps
Node3	2 核	4GB	70GB SSD	3Mbps
Node4	2 核	4GB	70GB SSD	3Mbps

为了避免实验的偶然性，本文在此实验中共设计了四组实验重复组,每组重复组分别采用五个应用 Pod，应用 Pod 配置文件除暴漏端口外，镜像和配置以及需求字段全部统一，应用镜像为基于 SpringBoot 框架开发的 web 应用，应用存在唯一接受无参查询的请求接口，测试数据库为 MySQL，查询数据实验前已基于 Redis 做好缓存，且数据库 MySQL 和缓存数据库 Redis 均为测试集群节点外的单体服务器，且接口均已通过测试，如数据 5-1 展示测试应用 Pod 的 Yaml 文件部分配置。

数据 5-1 **测试应用 Pod 的 Yaml 文件部分配置**

```
spec:
  containers:
  - name:Scheduletest
    image:192.168.145.132/tlbGA/Scheduletest:v1
    imagePullPolicy:IfNotPresent
    ports:
    - containerPort:8071
    resources:
      requests:
        CPU:100m
        memory:200Mi
      limits:
        CPU:200m
          memory:500Mi
```

实验中所使用 IGA 的参数配置如表 5-3 所示，其中 POP_SIZE 为种群规模，FIT_MAX 为 4.2.5 章中两个终止条件中之一的最大适应度，T_MAX 为 4.2.5 章中第二个终止条件的最大迭代次数，A 为退火系数，T0 为初始温度。

表 5-3 IGA 参数表

参数名	实验值
POP_SIZE	50
FIT_MAX	8
T_MAX	400
A	0.9
T0	10

实验利用节点实时综合资源利用率反应节点的真实负载，并对每组实验的集群节点计算其真实负载的标准差以反应集群的负载不均衡程度。标准差在数学意义上用来反应一组数据集的离散程度，标准差为方差的算术平方根，标准差越大代表，该组数据越离散，反之同理。负载标准差$\sigma(L(Ava))$计算公式如公式 5-1 所示：

$$\sigma(L(Ava)) = \sqrt{\frac{\sum_{i=1}^{N}(L(Syn_i)-L(Ava))^2}{CluNum}} \quad (5\text{-}1)$$

公式 5-1 中 CluNum 表示集群节点数量，集群节点负载数据的离散程度可以代表该集群的负载不均衡程度，负载标准差越大，其负载不均衡程度越高。

5.1.3 集群负载均衡程度结果与分析

实验根据 Prometheus 监控记录的集群节点，将使用 TLB 资源调度策略、TLB-IGA 调度策略、以及 Kubernetes 默认调度策略的三种集群共四组实验重复组调度后的集群节点负载数据进行数据收集，并绘制成折线图，结果如图 5.2 所示，从实验结果折线图的曲折程度和数据点值两方面进行分析如下：

从实验结果折线图的曲折程度中进行分析：

（1）Kubernetes 默认调度策略调度结果中，集群节点间的负载程度有着较大差异，导致折线图走向较为曲折，数据点上下浮动的程度较大。

（2）TLB 调度策略的调度结果同 TLB-IGA 调度策略有着较为相近的曲折程度，二者的折线均较为平滑，并无较大的波动，各自的节点之间具有较为相近的负载数据。

从实验结果折线图的数据点对应的负载数值进行分析：

（1）Kubernetes 默认调度策略结果中，由于调度器更倾向于将应用服务 Pod 调度至基础配置更高的节点 master 上，导致四组实验中，节点 master 的负载最终均达到了所有集群节点中的顶峰，分别为 45.9%，46.2%，46.2%，45.2%，而其他节点的负载情况除 node1 在重复组 1 中处于 45.4%外，均没有超过 45%，其中 node4 的节点负载甚至一直处于 35%-39%之间,处于该集群中负载最下游的水平，代表节点始终处于运行环境仍较为空闲的环境状态；同样，node2 节点仅在重复组 2 中出现了较高的负载 41.2%，在重复组 1、3、4 中均处于低于 40%的负载状态，处于集群负载的下游水平，四组重复组实验中出现过的最大负载差异为重复 2 中 master 节点与 node4 节点负载差异，达到了 10.2%；

（2）TLB 调度策略调度结果中，所有节点的负载数据均处于 42%-47 之间%，五个节点在四组重复组中一直处于较为平衡的负载状态，最大负载差异为重复组 1 中 node3 与 node4 的负载差异，仅为 3.9%。

（3）TLB-IGA 调度策略调度结果同样达到了 5 个均衡负载的现象，所有节点的负载数据始终处于 43%-47%之间，处于负载均衡的状态，最大负载差异为实验重复组 1 中 node3 与 node4 的负载差异，仅为 3.3%。

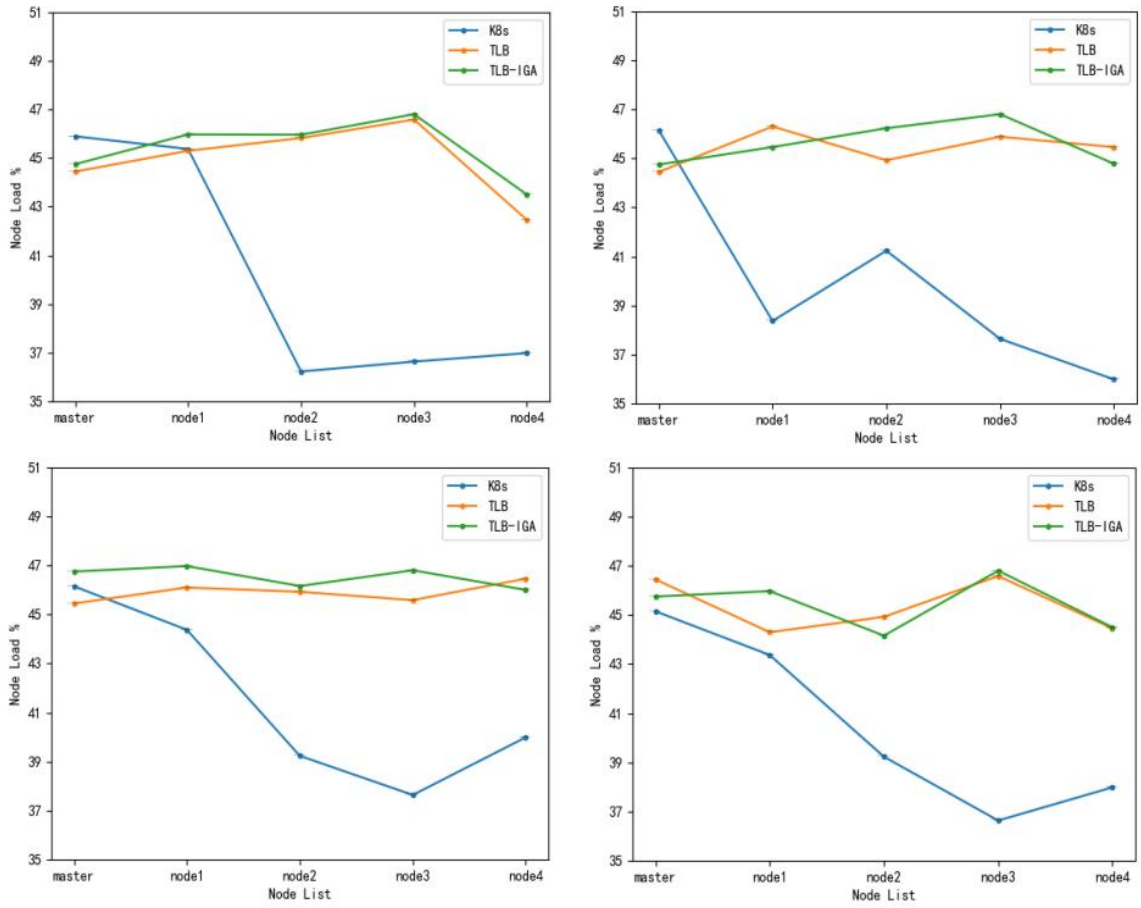

图 5.2 节点负载折线图

对四组重复组中三种集群各自五个节点的负载数据计算平均值，如表 5-4 所示，Kubernetes 默认调度策略四组实验组的整体平均负载为 40.8%，而 TLB 调度策略四组整体平均负载为 45.1%；相较于 Kubernetes 默认调度策略，TLB 调度策略由于引入了 Pod 真实负载获取模型，平均增加了 Kubernetes 集群 4.3%的节点负载；TLB-IGA 调度策略的四组整体平均负载为 45.4%，相较于 Kubernetes 默认调度策略，TLB-IGA 调度策略由于结合了 Pod 真实负载获取模型和并行化调度优选模型，平均增加了 Kubernetes 集群 4.7%的节点负载，提升了较小的负载消耗。

表 5-4 负载数据对比

实验重复组	K8s	TLB	TLB-IGA
重复组 1	40.6%	44.5%	45.1%
重复组 2	41.7%	45.6%	45.9%
重复组 3	40.7%	45.2%	45.4%
重复组 4	40.3%	45.1%	45.2%

将四组重复组实验的共 60 条节点负载数据记录统计共同绘制成柱状图，如图 5.3

所示。从树状图的结果可以看出：

（1）Kubernetes 默认调度策略的负载数据在图中所有区间均有出现，数据分布极为分散。

（2）TLB 调度策略的负载结果呈现节点的负载只在 41%-43%、43%-45%以及 45%-47%三个区间分配，相较于 Kubernetes 默认调度策略，减少了 3 个负载区间的集群节点出现。

（3）TLB-IGA 调度策略的负载结果呈现节点的负载只在 43%-45%、45%-47%两个区间分配，相较于 Kubernetes 默认调度策略，减少了 4 个负载区间的集群节点出现；相较于 TLB 调度策略的结果，二者的集中趋于较为重合，数据分布情况并无较大区别。

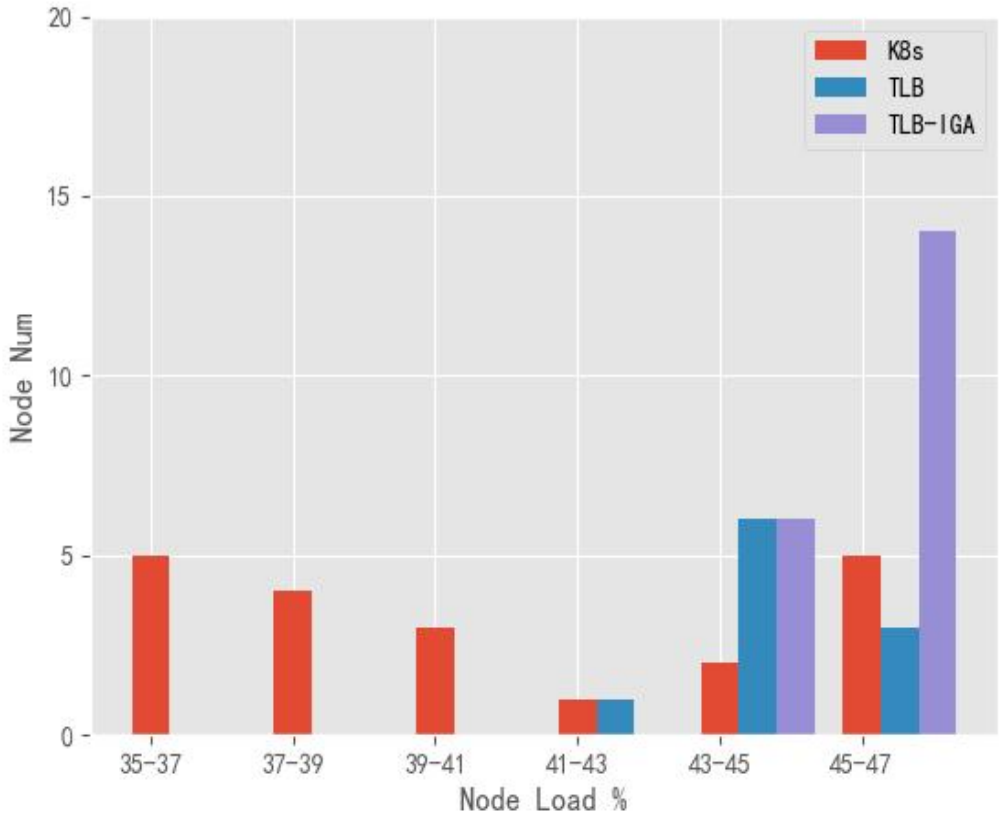

图 5.3 集群负载分布图

对三种调度策略的四组实验重复组计算节点的负载标准差反应调度后的 Kubernetes 集群各个节点负载不均衡程度。如表 5-5 所示，使用 TLB 调度策略以及 TLB-IGA 调度策略调度后的节点，其节点的负载标准差在四组实验重复组中，均低于使用 Kubernetes 默认调度策略的结果，TLB 调度策略四组实验的平均负载标准差为 1.0418，TLB-IGA 调度策略为 0.9933，二者并未较大差异。

表 5-5 节点负载标准差

实验重复组	K8S	TLB	TLB-IGA
重复组 1	3.5598	1.0585	1.0594
重复组 2	3.5621	0.9605	0.9492
重复组 3	3.7247	1.1451	1.0158
重复组 4	3.6598	1.0032	0.9487

使用四组重复组实验的标准差的均值作为代表 Kubernetes 默认调度策略和 TLB 调度策略以及 TLB-IGA 调度策略负载的各自的负载不均衡程度，对比参考文献[25]提出的没有使用本文 Pod 真实负载获取模型的负载均衡调度策略 SP-IGA 的负载标准差，如表 5-6 所示，TLB-IGA 调度策略相比于 SP-IGA 调度策略的节点负载标准差降低了 0.1094，降低了 9.9%的负载不均衡程度，即相比于其他负载均衡调度策略，TLB-IGA 调度策略由于使用了比带宽占用率更细粒度的 Pod 真实网络分配率指标，做到了进一步负载均衡程度的提升。

表 5-6 节点负载标准差对比

调度策略	节点负载标准差
K8s	3.6266
SP-IGA	1.1027
TLB	1.0418
TLB-IGA	0.9933

综合图 5.1、图 5.2 和表 5-4、表 5-5 以及表 5-6 的数据结果分析，可以说明 TLB 调度策略和 TLB-IGA 调度策略二者均实现了 Kubernetes 集群节点的负载均衡状态，即完成了对 5.1.1 章节实验目标中的（1）、（2）、（4）三个目标结论的论证，证明了 TLB 调度策略在解决负载不均衡问题上具有有效性和可行性、TLB-IGA 调度策略在解决负载不均衡问题上具有有效性和可行性、TLB-IGA 调度策略相比于其他没有使用本文 Pod 真实负载获取模型的负载均衡调度策略在提升负载均衡效果上具有一定优越性。

5.1.4 调度时间开销结果与分析

结果如表 5-7 所示，从实验的结果中可以看出，TLB-IGA 调度策略相较于 TLB 调度策略缩减了巨大的时间开销，相较于 Kubernetes 默认调度器调度策略处于适中的阶段。TLB 调度策略的平均时间开销为 0.59 秒，相比于 Kubernetes 默认调度策略平

均 0.20 秒的时间开销上升到了其 2.95 倍，时间开销提升了 195%；而 TLB-IGA 的平均时间开销为 0.16 秒，相比于 TLB 调度策略，降低了 74%的时间开销、相比于 Kubernetes 默认调度策略，降低了 22%的时间开销。

表 5-7 调度时间开销表

实验重复组	K8s（s）	TLB（s）	TLB-IGA（s）
重复组 1	0.21	0.57	0.17
重复组 2	0.20	0.62	0.15
重复组 3	0.22	0.64	0.16
重复组 4	0.17	0.53	0.15

综合图 5.1、图 5.2 及图 5.3 和表 5-4、表 5-5、表 5-6 及表 5-7 的数据结果分析，可以说明 TLB-IGA 调度策略实现了 Kubernetes 集群节点的负载均衡状态同时，用了较低的时间开销，完成了对 5.1.1 章节实验目标中的目标结论（3）的论证，证明了 TLB-IGA 调度策略在解决负载不均衡问题和调度时间开销大的问题上同时具有有效性和可行性。

5.2 基于 TLB-IGA 调度策略的集群性能实验

5.2.1 实验目标

为了验证 TLB-IGA 调度策略在提高 Kubernetes 集群整体提供服务的能力上的有效性和可行性，本章节实验通过 QPS(Queries Per Second，每秒处理请求条数)和 RT (Response time，平均响应时间）两方面指标的验证结果证明，TLB-IGA 调度策略在提升 Kubernetes 集群整体提供服务的能力上具有有效性和可行性。

5.2.2 实验设计与环境搭建

本章节实验的实验环境使用 5.1 章节中 Kubernetes 默认调度策略集群以及 TLB-IGA 调度策略集群，基础环境版本与节点物理配置以及测试应用清单文件，与 5.1 章节完全一致不做修改，这里不再展示，两个对比集群的选用如图 5.4 所示。测试应用的 Pod 数量设置为 50 个，待 50 个应用 Pod 全部完成调度部署后，使用 JMeter 分别向使用了 TLB-IGA 调度策略以及 Kubernetes 默认调度策略的两个 Kubernetes 集群发起压测，JMeter 每过 5 秒提升 500 线程数每秒作为跨度，记录线程从 500-5000 之间的 QPS 和 RT 值，共计 10 次数据记录。

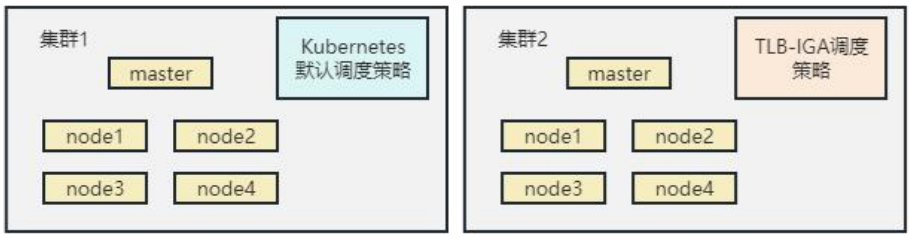

图 5.4 实验对比集群选用

5.2.3 集群性能实验结果与分析

TLB-IGA 调度策略和 Kubernetes 默认调度策略的 10 次 QPS 记录如图 5.5 所示。对两个集群的 QPS 结果进行分析如下：

（1）Kubernetes 默认调度策略集群随着 JMeter 测试线程数量的递增，在 35 秒前的 QPS 呈现上升趋势，与 TLB-IGA 调度策略集群并无较大差别。至 35 秒时，JMeter 达到每秒 3500 个测试线程，Kubernetes 默认调度策略集群的 master 节点出现节点宕机重启现象，集群整体 QPS 开始显著下降。

（2）TLB-IGA 调度策略集群的 QPS 在 10 次记录中均高于或等于 Kubernetes 默认调度策略集群，且并未出现节点宕机情况。实验结果表明，TLB-IGA 调度策略相较于 Kubernetes 默认调度策略拥有更高 QPS，即拥有更强的处理请求能力，其总体 QPS 提升了 27.8%

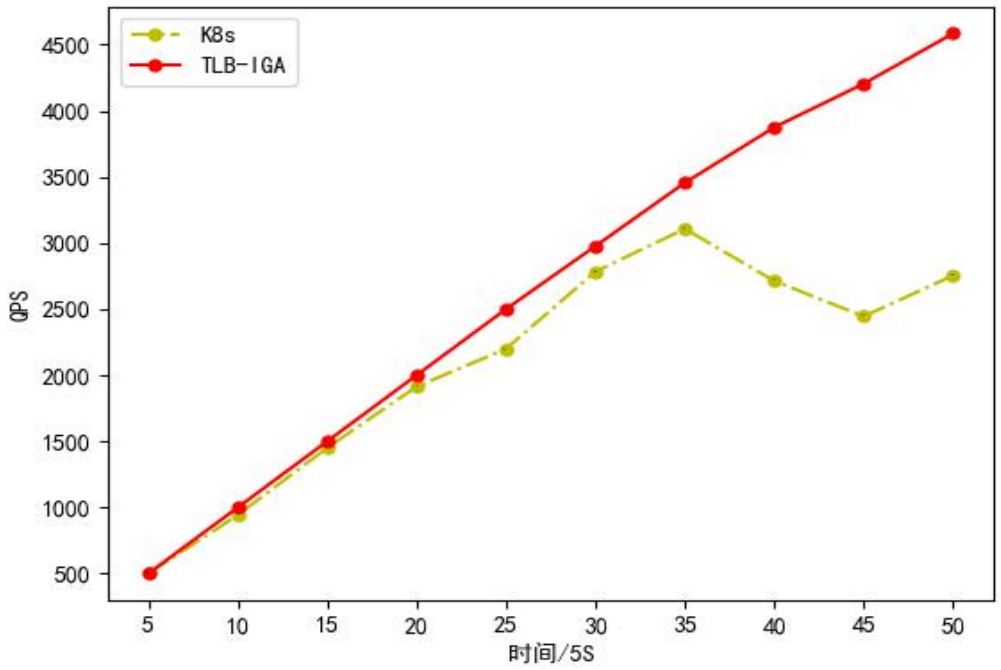

图 5.5 JMeter 压测集群 QPS

TLB-IGA 调度策略和 Kubernetes 默认调度策略的 10 次 RT 记录如图 5.6 所示。对两个集群的 RT 结果分析如下：

（1）Kubernetes 默认调度策略集群的 RT 值总体呈现先下降后上升趋势，前 35 秒，两种集群的 RT 值均较低且未超过 2.5 秒，35 秒时，Kubernetes 默认调度策略集群，master 节点的宕机重启导致集群处理响应能力下滑，RT 值开始明显上升，直到 40 秒记录时 RT 值已经超过了 3.5 秒；

（2）TLB-IGA 调度策略集群的 RT 值全程均未超过 2.5 秒，集群节点整体处于较为平稳有力的状态。总体可见，TLB-IGA 调度策略较 Kubernetes 默认调度策略拥有短的 RT 值，即拥有更快的响应时间，其总体 RT 值降低了 20.5%。

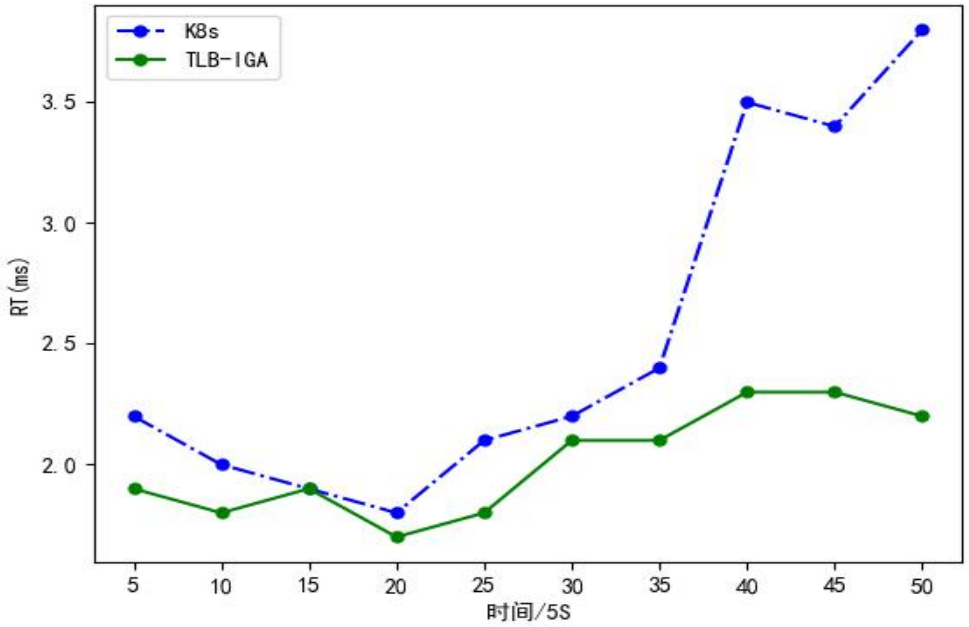

图 5.6 JMeter 压测集群 RT

综合图 5.5 和图 5.6 的数据分析，可以证明本文 TLB-IGA 调度策略在提升集群整体提供服务的能力上具有有效性和可行性。

5.3 与标准遗传算法（SGA）的对比实验

5.3.1 实验目标

为了验证 TLB-IGA 调度策略使用的改进遗传算法 IGA 相比于标准遗传算法 SGA 在 Kubernetes 调度问题寻优能力上的优越性，本实验通过在算法迭代次数和调度时间两个指标上的验证结果，证明本文 TLB-IGA 调度策略使用的改进遗传算法 IGA 相比于标准遗传算法 SGA 在 Kubernetes 调度问题的寻优能力上具有优越性。

5.3.2 实验设计与环境搭建

本章通过将 TLB-IGA 调度策略中的改进遗传算法（IGA）替换成标准标准遗传算法(SGA)，编译 TLB-SGA 调度策略搭建 Kubernetes 集群。使用编译后的 TLB-SGA 调度策略集群与 5.1 章节中使用的 TLB-IGA 调度策略集群进行对比实验，实验对比集群的选取如图 5.7 所示。对 TLB-SGA 调度策略集群进行 5.1.2 章节中与 TLB-IGA 调度策略集群完全相同的调度实验，实验过程完全一致不做任何改变，这里不再描述。记录调度过程中标准遗传算法的迭代次数变化与调度过程消耗的时间，并与本文改进的遗传算法 IGA 进行对比。

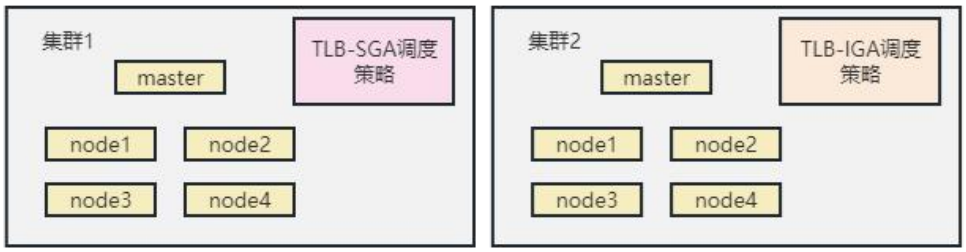

图 5.7 实验对比集群选用

本章节实验使用环境的基础环境版本与节点物理配置以及测试应用清单文件完全一致不做修改，这里不再展示。标准遗传算法 SGA 的参数如表 5-8 所示，其中 POP_SIZE 为种群规模，FIT_MAX 为 4.2.5 章中两个终止条件中之一的最大适应度，T_MAX 为 4.2.5 章中第二个终止条件的最大迭代次数，P_CRO 为交叉概率，P_MUT 为变异概率。

表 5-8 标准遗传算法参数表

参数名	实验值
POP_SIZE	50
FIT_MAX	8
T_MAX	400
P_CRO	0.6
P_MUT	0.1

5.3.3 算法迭代次数对比结果与分析

将四组重复组实验中标准遗传算法（SGA）的迭代次数与 5.1.2 章节中 TLB-IGA 调度策略集群的四组重复组实验的改进遗传算法（IGA）算法迭代次数进行对比记录，

如图 5.8 所示。

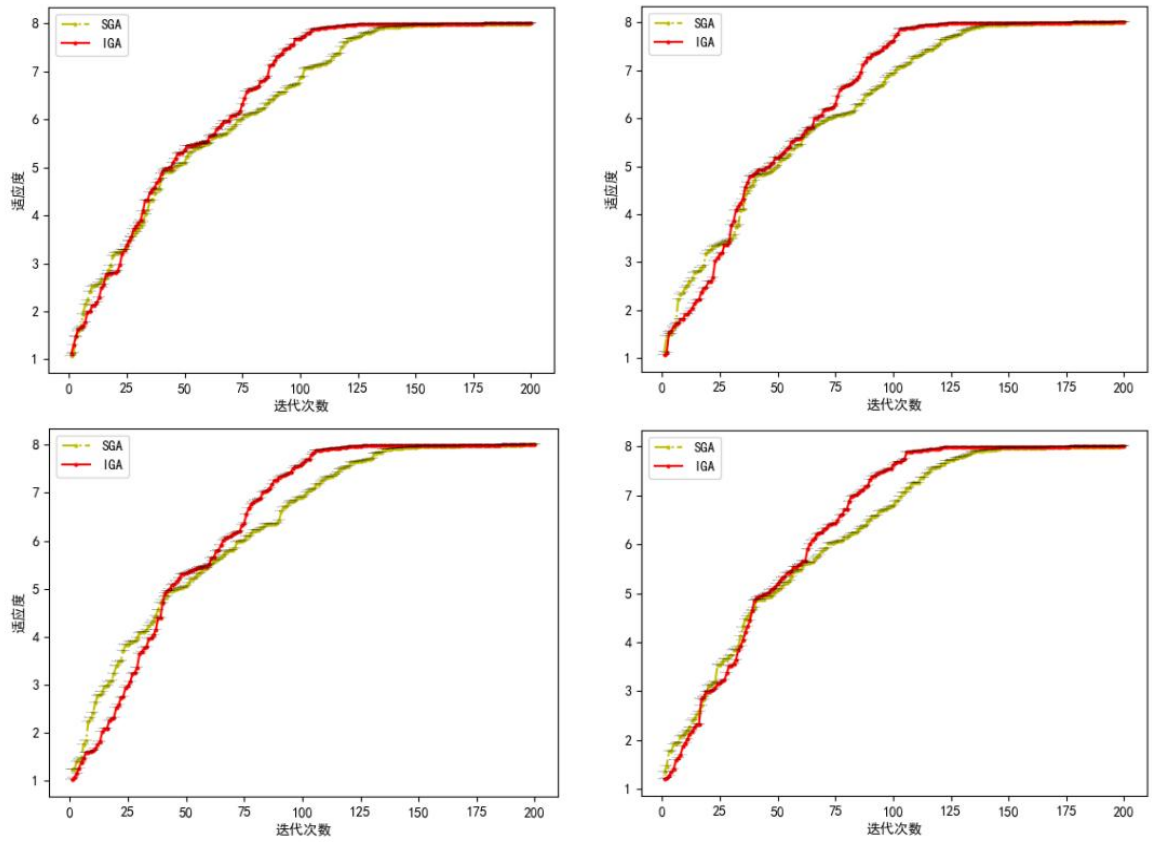

图 5.8 IGA 与 SGA 迭代对比

从四组实验图中可以看出，结合了模拟退火思想改进的 IGA 算法比标准遗传算法 SGA 具有更少的收敛迭代次数，可以用更少次的算法迭代完成最优解的寻找。

5.3.4 算法执行时间对比结果与分析

将使用标准遗传算法(SGA)编译的调度策略(TLB-SGA)的四组重复组实验的调度时间与 5.1.4 章节中 TLB-IGA 的四组调度时间进行记录对比，如表 5-9 所示。

表 5-9 调度时间开销表

实验重复组	TLB-SGA（s）	TLB-IGA（s）
重复组 1	0.26	0.17
重复组 2	0.22	0.15
重复组 3	0.23	0.16
重复组 4	0.25	0.15

四组重复组实验对比中，TLB-IGA 调度策略始终比 TLB-SGA 调度策略的调度时间短，TLB-SGA 的平均调度时间为 0.24 秒，TLB-IGA 调度策略相比于 TLB-SGA 调

度策略平均缩短了 34%的时间开销。

综合图 5.8、表 5-9 可以证明，TLB-IGA 调度策略使用的改进遗传算法 IGA 相比于标准遗传算法 SGA 在 Kubernetes 调度问题的寻优能力上具有优越性。

5.4 本章小结

本章围绕本文提出 TLB-IGA 调度策略和 TLB 调度策略建立了三个实验，并介绍了每个实验的实验目标以及具体的实验设计和环境搭建，最后对实验的结果进行了分析。实验结果表明，本文提出的 TLB-IGA 调度策略可以同时解决 Kubernetes 默认调度策略的负载不均衡和调度时间开销大的问题，并提升了集群整体提供服务的能力，且相比于没有使用本文 Pod 真实负载获取模型的其他负载均衡调度策略，在负载均衡效果上具有一定的优越性，其应用的 IGA 相比于 SGA 在 Kubernetes 调度问题上具有更好的寻优能力。

第 6 章　总结与展望

6.1 论文工作总结

面对容器平台庞大的用户体系和复杂多样的需求，调度算法不仅要求能够以较低的时间开销保障应用的调度部署，同时要求调度后承载容器的物理集群能够以健康的负载均衡状态来对外提供良好的服务能力。因此，本文以实现负载均衡效果和降低调度时间开销为主题，对目前最主流的 Kubernetes 容器平台展开调度算法的研究工作，分析 Kubernetes 默认调度算法存在负载不均衡问题和调度时间开销大问题的原因，设计优化方案，提出一种基于 IGA 改进的并行调度策略 TLB-IGA，以较低的调度时间开销实现了较好的负载均衡效果。下面对本文主要的工作内容进行总结：

（1）本文对 Kubernetes 标准调度模型进行分析，定位 Kubernets 默认调度算法负载不均衡问题的产生原因，通过分析发现负载不均衡问题的产生原因在于默认调度算法缺失了对集群节点网络运行情况的评估以及默认优选策略根据节点剩余资源情况进行优选打分的计算逻辑，针对这两点优化方向，本文提出了一种基于 Pod 网络改进的调度策略 TLB，在调度策略中通过引入 Prometheus 和 Istio 监控组件，构建了一种 Pod 真实负载获取模型，对节点上应用 Pod 的进出站流量数据以及 CPU 占用率、内存占用率进行获取，计算 Pod 的真实资源分配率以反应 Pod 的节点资源分配程度，并结三项资源分配率指标，设计节点实时综合资源利用率指标以反应节点的真实负载程度、设计节点理想负载偏离度指标以反应节点与理想负载均衡状态的偏离程度，以节点理想负载偏离度数值作为调度优选打分结果，代替默认调度算法使用节点剩余资源进行优选打分的计算逻辑，弥补了默认调度算法对节点网络环境评估的缺失，实现了负载均衡效果。

（2）通过对 Kubernetes 标准调度模型的分析，本文定位 Kubernetes 默认调度算法调度时间开销大问题的产生原因在于 Kubernetes 默认调度算法串行化的调度优选打分逻辑。针对这一优化方向，本文提出了一种基于 IGA 改进的并行调度策略 TLB-IGA，该策略通过使用基于模拟退火思想改进的自适应遗传算法，构建并行化调度优选模型，对本文提出 TLB 调度策略做出了时间开销上的优化。具体使用 Pod 调度方案的整体寻优迭代，代替 TLB 调度策略保留的默认调度算法的串行化调度优选打分机制，降低了 TLB 调度策略优选打分计算与数据传输频率，以较低的调度时间实现了较好的负载均衡效果，提升了集群提供服务的能力，且相比于其他负载均衡调度策略，TLB-IGA 调度策略在网络评估指标上，使用了比带宽占用率更细粒度的 Pod 真实网络分配率，在负载均衡程度上实现了一定的提升效果。

（3）针对于标准遗传算法在 Kubernetes 调度问题上存在的收敛速度慢的问题，本文设计了一种基于模拟退火思想改进的自适应遗传算法 IGA，结合模拟退火思想在

标准遗传算法中加入扰动机制，使用基于节点理想负载偏离度设计的自适应交叉算子与自适应变异算子，加快算法收敛速度，相比于标准遗传算法在 Kubernetes 调度问题上具有更好的寻优能力。

6.2 未来与展望

本文提出了一种基于 IGA 改进的并行调度策略 TLB-IGA，以较低的时间开销实现了集群调度的负载均衡效果，优化了 Kubernetes 部署应用的调度效率，提升了调度后的集群整体提供服务的能力。在未来，将持续针对 Kubernetes 平台调度算法展开以下方向的研究：

（1）对调度过程中执行计算逻辑所带来的节点工作负载上升的问题展开研究工作，减少节点在调度应用过程中的资源消耗，以提升调度过程内节点去执行用户应用的能力。

（2）对调度完成后的应用 Pod 迁移策略展开研究工作，使 Kubernetes 可以根据节点的负载程度变化自适应的调整对应用 Pod 的资源分配，持续保持集群的负载均衡效果，以长久提供良好的服务能力。

参考文献

[1] 付琳琳, 邹素雯. 微服务容器化部署的研究[J]. 计算技术与自动化, 2020 (4): 151-155.

[2] 马慧. 基于混合云平台的教育课程资源共享系统设计[J]. 现代电子技术, 2022.

[3] 赵军, 王晓. 云环境下国产可信根 TCM 虚拟化方案研究[J]. 信息技术与网络安全, 2020, 39(6): 44-48.

[4] 徐骁巍, 付晓轩. 虚拟化技术在计算机系统中的应用[J]. 信息与电脑, 2020, 32(1): 10-11.

[5] Burns B, Beda J, Hightower K, et al. Kubernetes: up and running[M]. " O'Reilly Media, Inc.", 2022.

[6] 郑忠斌,李世强,费海平.一种基于 Kubernetes 的工业物联网新型调度[J].单片机与嵌入式系统应用,2021,21(06):15-19.

[7] Stoyanov R, Armour W, Zilberman N. Network-accelerated cluster scheduler[M]//Proceedings of the SIGCOMM'22 Poster and Demo Sessions. 2022: 16-18.

[8] 吴逸文, 张洋, 王涛, 等. Development Exploration of Container Technology Through Docker Containers: A Systematic Literature Review Perspective[J]. Journal of Software, 2023: 1-25.

[9] 单朋荣,杨美红,赵志刚,李志鹏,杨丽娜.基于 Kubernetes 云平台的弹性伸缩方案设计与实现[J].计算机工程,2021,47(01):312-320.

[10] 夏畅. 基于 Kubernetes 的企业级容器云平台建设[J]. 电信快报, 2021, 1: 12-16.

[11] 马晋,赵思亮,赵芳.基于 Kubernetes 的云原生数据库集群部署方案[J].自动化与仪器仪表,2022(10):55-59.DOI:10.14016/j.cnki.1001-9227.2022.10.055.

[12] 胡晓亮.基于 Kubernetes 的容器云平台设计与实现[D].西安电子科技大学,2019.

[13] Carrión C. Kubernetes scheduling: Taxonomy, ongoing issues and challenges[J]. ACM Computing Surveys, 2022, 55(7): 1-37.

[14] 靳芳, 龙娟. 一种面向 Kubernetes 集群的网络流量弹性管理方法[J]. 北京交通大学学报, 2020, 44(05): 77.

[15] 温盈盈, 程冠杰, 邓水光, 等. APU: 一种精确评估超线程处理器算力消耗程度的方法[J]. 软件学报, 2023: 1-18.

[16] 何震苇, 黄丹池, 严丽云, 等. 基于 Kubernetes 的融合云原生基础设施方案与关键技术[J]. 电信科学, 2020, 36(12): 77-88.

[17] 刘可. 基于 Kubernetes 集群的资源调度策略研究[D].武汉纺织大学,2022.DOI:10.27698/d.cnki.gwhxj.2022.000375.

[18] Zhiming DAI,Mingtuo Z,Yang Y, etal.Fog computing resource scheduling in intelligent factories[J]. Journal of University of Chinese Academy of Sciences, 2021, 38(5):702.

[19] 潘远航,徐俊杰,颜开等. 基于BMC的Kubernetes集群物理节点的故障处理方法和系统[P]. 上海市：CN114218004A,2022-03-22.

[20] 马希琳. 基于 Kubernetes 容器集群资源调度策略研究[D].西安科技大学,2019.

[21] 徐盼望, 杜万和, 杨敬辉, 等. 边缘计算在制造业数据采集与处理中的应用[J]. 上海第二工业大学学报, 2021, 38(3).

[22] 黄志成.一种基于集群负载均衡的Kubernetes资源调度算法[J].电脑知识与技术,2023,19(05):39-41.DOI:10.14004/j.cnki.ckt.2023.0269.

[23] 李华东, 张学亮, 王晓磊, 等. Kubernetes 集群中多节点合作博弈负载均衡策略[J]. 西安电子科技大学学报, 2021.

[24] 熊衍捷,高镇, 李根, 等. 基于谱聚类的 BaaS 资源负载均衡调度算法[J]. 重庆大学学报, 2021, 44(11): 40-47.

[25] 胡程鹏.基于 Kubernetes 的容器云资源调度算法的研究[D].西安工程大　学,2021.DOI:10.27390/d.cnki.gxbfc.2021.000118.

[26] 陈丰琴.基于 Kubernetes 集群容器资源调度策略的研究与设计[D].西南交通大学,2020.DOI:10.27414/d.cnki.gxnju.2020.000777.

[27] 罗永安.基于 Kubernetes 的资源调度策略研究与优化[D].浙江理工大学,2021.DOI:10.27786/d.cnki.gzjlg.2021.000524.

[28] Nguyen Q M, Phan L A, Kim T. Load-balancing of kubernetes-based edge computing infrastructure using resource adaptive proxy[J]. Sensors, 2022, 22(8): 2869.

[29] Dua A, Randive S, Agarwal A, et al. Efficient load balancing to serve heterogeneous requests in clustered systems using kubernetes[C]//2020 IEEE 17th Annual Consumer Communications & Networking Conference (CCNC). IEEE, 2020: 1-2.

[30] Karypiadis E,Nikolakopoulos A,Marinakis A,etal.SCAL-E: An Auto Scaling Agent for Optimum Big Data Load Balancing in Kubernetes Environments[C]//2022 International Conference on Computer, Information and Telecommunication Systems (CITS). IEEE, 2022: 1-5.

[31] 刘哲源, 吕晓丹, 蒋朝惠. 基于模拟退火算法的粒子群优化算法在容器调度中的应用[J]. 计算机测量与控制, 2021, 29(12): 177-183.

[32] 赵飞鸿, 孙立峰. 一种基于深度强化学习的资源调度方法[J]. Computer Science and Application, 2021, 11: 2008.

[33] 蒋筱斌, 熊轶翔, 张珩, 等. 基于 Kubernetes 的 RISC-V 异构集群云任务调度系统[J]. 计算机系统应用, 2022, 31(9): 3-14.

[34] 戴志明, 周明拓, 杨旸, 等. 智能工厂中的雾计算资源调度[J]. 2021.

[35] 徐胜超, 熊茂华. 一种多集群容器云资源调度优化方法[J]. 计算机与数字工程, 2022, 50(11): 2490-2496.

[36] El Haj Ahmed Ghofrane,GilCastiñeira Felipe,CostaMontenegro Enrique. KubCG: A dynamic Kubernetes scheduler for heterogeneous clusters[J]. Software: Practice and Experience,2020,51(2).

[37] Chima Ogbuachi M, Reale A, Suskovics P, et al. Context-aware Kubernetes scheduler for edge-native applications on 5G[J]. Journal of communications software and systems, 2020, 16(1): 85-94.

[38] Rausch T, Rashed A, Dustdar S. Optimized container scheduling for data-intensive serverless edge computing[J]. Future Generation Computer Systems, 2021, 114: 259-271.

[39] Panda S, Ramakrishnan K K, Bhuyan L N. pmach: Power and migration aware container scheduling[C]//2021 IEEE 29th International Conference on Network Protocols (ICNP). IEEE, 2021: 1-12.

[40] Harichane I, Makhlouf S A, Belalem G. KubeSC‐RTP: Smart scheduler for Kubernetes platform on CPU‐GPU heterogeneous systems[J]. Concurrency and Computation: Practice and Experience, 2022, 34(21): e7108.

[41] 郝鹏海, 徐成龙, 刘一田. 基于 Kafka 和 Kubernetes 的云平台监控告警系统[J]. 计算机系统应用, 2020, 29(8): 121-126.

[42] A.Pereira Ferreira and R.Sinnott,"A Performance Evaluation of Containers Running on Managed Kubernetes Services," 2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom),Sydney,NSW,Australia,2019,pp.199-208,doi:10.1109/CloudCom.2019.00038.

[43] 常旭征,焦文彬.Kubernetes 资源调度算法的改进与实现[J].计算机系统应用, 2020, 29(07):256-259.

[44] V.M.Kureichik and J.A.Logunova,"The Implementation of the Genetic Algorithm

Using Cloud-Based Computing on the Internet," 2019 IEEE East-West Design & Test Symposium (EWDTS),Batumi,Georgia,2019,pp.1-4.

[45] 吴封斌,李笑瑜,蒲睿强,等.Istio:微服务架构服务治理升级研究[J].网络安全技术与应用,2022(7):2.

[46] 王冲,周甜,邓志伟,等.一种基于 Kubernetes 平台和 Istio 网格技术的灰度发布编排方法:,CN111176713A[P].2020.

[47] 张凯,王一,郑恺,等.一种自动适配多个外部注册中心到服务网格 Istio 的方法和装置:,CN202210671699.9[P].2022.

[48] 罗天.一种基于 istio 的微服务 API 网关的安全计费方法,CN110324341A[P].2019.

[49] 马永,吴跃,何李囡,等.基于 Prometheus 的基础软硬件全链路监控设计和实现[J].电子技术与软件工程,2019(24):2.

[50] 逄立业.一种 Prometheus 监控方法,装置及设备,CN112711512A[P].2021.

[51] Leppänen T. Data visualization and monitoring with Grafana and Prometheus[J]. 2021.

[52] 萧秋兰.遗传模拟退火算法的优化研究[J].信息记录材料,2022,23(12):95-98.DOI:10.16009/j.cnki.cn13-1295/tq.2022.12.052.

[53] 孙晨辉.基于改进模拟退火算法的云任务调度策略研究[D].南京邮电大学,2022.DOI:10.27251/d.cnki.gnjdc.2022.000892.