涂小妹, 闫东升. 基于实时监控的动态调节线程池方法研究[J]. 智能计算机与应用,2024,14(9):76-81. DOI:10.20169/j.issn.2095-2163.240911

基于实时监控的动态调节线程池方法研究

涂小妹[1], 闫东升[2]
(1 浙江广厦建设职业技术大学 建筑工程学院, 浙江 东阳 322100; 2 杭州微易信息科技有限公司, 杭州 310018)

摘 要: 在数字社会对强算力的要求下,为使 CPU 性能最大化,需要使用线程池来提高 CPU 并行计算能力,然而在开发中线程池的使用往往伴随着线程池状态混乱以及难以监控的问题。针对该问题,本文提出了一种基于实时监控的动态调节线程池方法。该方法通过实时监控线程池参数,动态调整核心线程数与最大线程数的数量来规避由于线程资源的配置不足导致业务故障的问题;引入监控中心与报警中心模块,以便直观监测线程池的执行状态、线程数量、队列容量等来实现实时监控。引入配置中心模块实时同步修改线程池的参数,快速的调整线程池参数并同步到线程池中。实验结果表明,本文所提出的方法能有效的动态调节线程池,缓解线程池状态与任务混乱的问题。

关键词: 线程池; 动态调节; 实时监控; 配置中心; 并行计算
中图分类号: TP311.1 **文献标志码**: A **文章编号**: 2095-2163(2024)09-0076-06

Research on dynamic thread pool method based on real-time monitoring

TU Xiaomei[1], YAN Dongsheng[2]
(1 College of Civil Engineering and Architecture, Zhejiang Guangsha Vocational and Technical University of Construction, Dongyang 322100, Zhejiang, China; 2 Hangzhou Weiyi Information Technology Co., Ltd, Hangzhou 310018, China)

Abstract: Under the requirement of strong computing power in digital society, in order to maximize CPU performance, it is necessary to use thread pools to improve CPU parallel computing capability, however, the use of thread pools in development is often accompanied by the problem of chaotic thread pool status and difficult to monitor. In order to address this problem, this paper proposes a method for dynamically adjusting the thread pool based on real-time monitoring. The method dynamically adjusts the number of core threads and the maximum number of threads by monitoring the thread pool parameters in real time to avoid the problem of business failures due to insufficient configuration of thread resources; introduces a monitoring center and an alarm center module in order to visually monitor the execution state of the thread pool, the number of threads, the capacity of the queue and so on to achieve real-time monitoring. The configuration center module is introduced to synchronously modify the parameters of the thread pool in real time, so as to quickly adjust the thread pool parameters and synchronize them to the thread pool. Experimental results show that the method proposed in this paper can effectively adjust the thread pool dynamically and alleviate the problem of thread pool state and task confusion.

Key words: thread pool; dynamic adjustment; real-time monitoring; configuration center; parallel computing

0 引 言

现代计算机处理系统呈现出多样化趋势,从小型手持设备、笔记本电脑到大型集群系统,无不如此[1]。数字化时代的到来,系统软件日趋多样和复杂。因此,针对单一的处理器结构,即使微架构设计经过充分的优化也无法同时满足多样化的系统需求。对于企业级软件开发,常采用多线程方式控制程序并发计算以达到充分利用机器资源、提高计算能力的效果。从系统架构演化历程来看,当遇到并发计算瓶颈时,优化多线程开发是重要选择之一[2]。

在不同的软件系统中,多线程的开发方法不同,然而底层逻辑实现都大同小异。基于面向对象语言

基金项目: 浙江省教育厅一般科研项目(Y202250677);校青年培育项目(2020QNPY004)。
作者简介: 闫东升(1993-),男,学士,主要研究方向:软件开发,智能信息处理。
通讯作者: 涂小妹(1995-),女,硕士,主要研究方向:视频图像识别,人工智能,模式识别。Email: txm_95@163.com
收稿日期: 2023-04-17

开发的大型软件通常采用线程池来解决并发效率问题，线程池是一种基于资源池思想的线程管理工具。李哲等[3]利用缩减搜索空间，有效提高并行算法的工作效率；Minutoli[4]和 Papadopoulos[5]基于反馈机制异构多核处理器将不同类型的处理核心集成在处理器芯片，以满足各种应用程序的需求；周彭等[6]针对多线程的并行化方法有较大的额外通信开销，提出了使不同工作节点共享约束求解信息的加速求解方法；Inverso 等[7]通过延迟顺序化对多线程程序进行有界验证；Wang 等[8]提出基于多线程的多核和云资源的协同调度方法。

线程的开辟、调度、销毁是一个相当消耗资源的过程，过多的线程会带来资源的过度消耗[9-10]。但线程池可统一管理线程的生命周期，能够有效降低有限资源消耗，提高项目响应速度，提升线程的可管理性[11]。然而，由于线程调度是通过机器 CPU 来管理，即便检测到线程池参数存在问题，调整之后服务器的重启也是无法接受的[12-13]。本文将从线程池原理和线程池使用出发，提出企业级可控线程池设计方案。

1　并发任务的线程池设计原理分析

1.1　线程池整体架构

JAVA 线程池顶层执行器接口（Executor）中设计了执行方法（Execute），开发人员无需关心方法内部实现，仅传入 Runnable 任务即可，旨在达到解耦内聚的效果[14]。ExecutorService 接口继承自 Executor，对 Executor 功能进行了扩展，提供一些操作线程池的基础功能，如 Shutdown、Submit 等。抽象类 AbstractExecutorService 为 ExecutorService 接口中部分方法提供了默认实现。ThreadPoolExecutor 类为线程池提供了核心实现，其内部实现了一个生产者消费者模型，用来接收任务与执行任务，其中缓冲队列达到复用线程的效果，线程池内部运行如图 1 所示。

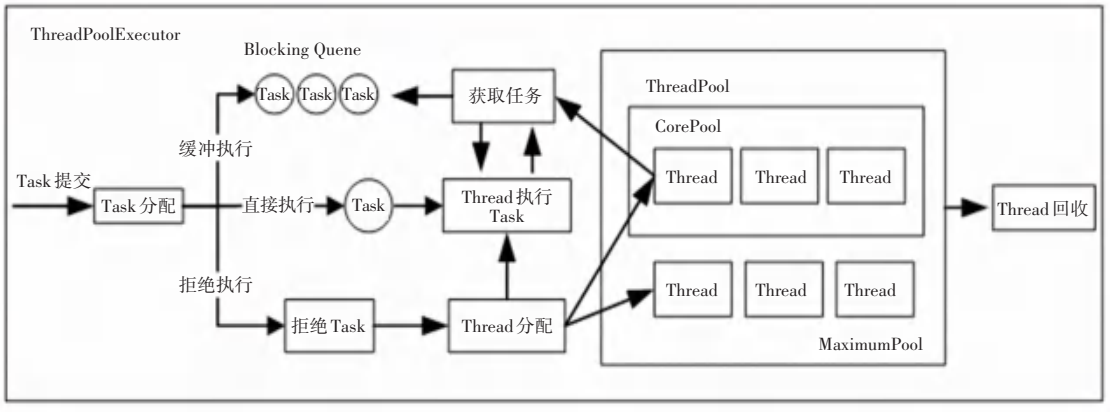

图 1　线程池内部运行图

Fig. 1　Thread pool internal operation graph

线程池内两大角色主要是线程和任务，任务管理者相当于生产者，线程管理者相当于消费者[15-16]。任务请求到达线程池后总体分为 3 种处理方式：（1）主线程直接执行任务；（2）将该任务暂时放到缓冲队列等待线程调用；（3）执行拒绝策略。以上 3 种方式的执行策略与线程池内部的状态紧密关联。

1.2　线程池生命周期

线程池的状态随着线程池的运转由内部程序维护，JDK 有个巧妙设计，使用一个变量同时维护了线程池的运行状态和线程数量[17]。利用 AtomicInteger 变量来同时维护线程状态和线程数量，该变量共 32 位，高 3 位表示线程状态，低 29 位表示线程数量，两个值之间互不干扰[18]。用一个变量去存储两个值，可避免在做相关决策时，出现不一致的情况。线程池状态之间的转换关系如图 2 所示。

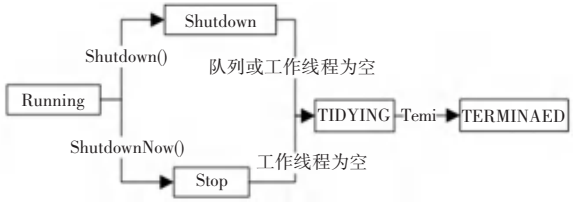

图 2　线程池状态转换图

Fig. 2　Thread pool internal operation graph

1.3　Worker 线程管理

Worker 线程是线程池的核心逻辑，其设计目的是为了让线程池能够掌握线程的生命周期[19]。Worker 线程是继承了 AbstractQueuedSynchronizer 的

Runnbale 类,Worker 的构建需传入一个 Runnbale 对象,由于 Worker 重写了 Run 方法,重写的 Run 方法执行 RunWork 函数,其持有当前线程和一个获取任务线程 FirstTask。当前线程主要用于判断线程执行内部状态等操作,FirstTask 用来表示传入线程池的第一个任务,若不为空,任务由核心线程执行,若为空,需要从缓冲队列中获取任务执行,Worker 线程执行任务过程如图 3 所示。

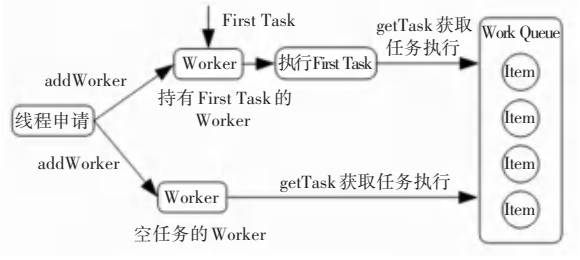

图 3 Worker 线程任务执行过程图

Fig. 3 Worker thread task execution process diagram

线程增加函数 addWorker,在检查线程池的运行状态以及给定边界合法之后,即可创建并启动一个新线程[20]。Worker 线程的销毁依赖于 JVM 的自动回收机制,线程池只需要根据线程的状态维护线程在 hash 表中的引用,消除引用的线程则会被 JVM 择机回收。

2 动态调度线程池方法设计

2.1 线程池调节机制

2.1.1 线程池缓冲队列

线程池能够稳定高效处理任务的核心是缓冲队列,线程池的本质是对任务的处理和对线程的管理。为了做到两者互不干扰且相互联系(解耦合),引入基于生产消费者模型的阻塞队列(BlockingQueue),生产者(Product)将任务压入队列中,由消费者(Custom)根据合适的策略消费队列中的任务。该队列遵循的原则:当队列为空时 Custom 会等待队列中重新压入任务,当队列满时 Product 会等待队列重新可用。队列的角色如同任务,临时存放需要被执行的任务,其缓冲队列执行过程如图 4 所示。

图 4 缓冲队列执行过程

Fig. 4 Buffer queue execution process

2.1.2 线程池申请模式

线程是稀缺资源,引入线程池的作用就是避免频繁的创建与销毁线程。第一种情况一般只出现在线程初始创建阶段,而第二种模式呈绝大多数情况。线程源源不断的从队列中获取任务执行,所以线程管理模块与缓冲队列之间需要一个通信过程。JDK 提供了 getTask 函数用于任务的申请,其大致流程:申请任务后首先判断线程池的状态,计算已存在的线程数量,在线程池未停止运行且线程数满足线程池要求的情况下,若线程为可回收线程,则从队列中获取任务执行,反之该线程继续执行当前任务,待其成为可回收线程返回前者。getTask 进行了多次判断,其目的是控制线程的数量,使其符合线程池的状态。如果线程池当前不能持有那么多线程,则会返回 null 值。

2.1.3 线程池拒绝策略

任务拒绝策略是线程池的保护部分,线程池有最大的容量,当线程池的任务缓存队列已满,并且线程池中的线程数目达到最大线程数时,就需要拒绝处理该任务,采取任务拒绝策略保护线程池。JDK 对该接口提供了 4 种实现,根据不同的要求可选择不同的拒绝策略。4 种拒绝策略:

(1) AbortPolicy:丢弃任务并抛出异常,在线程池不能接收新任务后及时抛出异常,常用于一些重要的业务场景;

(2) CallerRunsPolicy:由提交任务的线程处理该任务,该模式下所有的任务都将得到执行,可保证任务执行的正确性;

(3) DiscardOldestPolicy:丢弃队列最前面的任务,重新提交被拒绝的任务,该策略需要衡量队列中任务的重要性;

(4) DiscardPolicy:丢弃任务,不抛异常,该策略用户无法发现线程池的异常状态、适合一些不重要的业务场景。

2.2 动态调节线程池问题分析

在实际项目开发中,不同业务需求对线程池的要求和影响是不一致的,其性能受任务类型的影响较大,不同类型的 CPU 在处理任务时有不同的表现,设置不合理的线程池可能会对业务系统造成伤害。若没有合理的考量系统访问量,线程池核心数与最大线程数设置过小,当大量的访问请求到达线程池后,超出线程池处理能力的任务就会抛出拒绝执行异常(RejectedExecutionException),导致系统报错,造成损失。因此,合理的设置线程池参数和正确使用线程池对高并发的项目有重要的影响,推荐设

置的线程数与机器 CPU 核心数、IO 耗时、CPU 耗时相关，计算关系如下：

$$N_{thread} = N_{cpu} \times (1 + T_{IO}/T_{CPU}) \quad (1)$$

其中，N_{thread} 表示线程数；N_{CPU} 表示 CPU 核数；T_{IO} 和 T_{CPU} 表示 IO 设备耗时和 CPU 设备耗时。

推荐设置的线程数与 CPU 核心数、阻塞系数(0~1 之间)相关，计算关系如下：

$$N_{thread} = N_{CPU}/(1 - \sigma) \quad (2)$$

其中，σ 表示阻塞系数。

参照现有公式设置线程池参数并非易事，上述 IO 耗时和阻塞系数并不能简单的计算出来，而是要根据线程池执行的不同的任务类型去估算，这种模糊不清的参数也是线程池中存在的隐患之一。

2.3　动态调节线程池方法设计

线程池的核心参数有：核心线程数(corePoolSize)、最大线程数(maximumPoolSize)、阻塞队列(workQueue)，是线程池的内部运转策略关键。线程池(ThreadPoolExecutor)内部提供一些基础的方法供开发人员查看与修改线程池的部分参数，如获取核心线程数(getCorePoolSize)、获取最大线程数(getMaximumPoolSize)可以查看当前线程池的核心线程数与最大线程数，设置核心线程数(setCorePoolSize)、设置最大线程数(setMaximumPoolSize)可以根据线程池目前的状态调整核心线程数与最大线程数的数量。根据以上函数可以设计两种线程池模型，线程池执行状态调整策略 1 的伪代码见表 1。

第二种当队列容量达到一定阈值的时候，动态调整 maxPoolSize 大小，伪代码见表 2。

表 1　线程池执行状态调整策略 1

Table 1　Thread pool execution state adjustment Strategy 1

线程池执行状态 1
buildExecutor(core,max,size)表示构造线程池，core、max、size 分别表示核心线程数、最大线程数与队列容量。
getTask()表示获取任务线程，内部打印线程池状态、休眠 1 000 ms。
execute(task)表示交由线程池执行任务，task 表示 Runable 对象。
executor :=buildExecutor(2,5,10); //创建一个核心线程数、最大线程数线程池，阻塞队列容量分别为 2,5,10 的线程池
For i: = 1 To 10 //循环执行任务
Runnable task:=getTask(); //获取可执行任务
executor.execute(task); //交由线程池执行任务
For End //循环结束

表 2　线程池执行状态调整策略 2

Table 2　Thread pool execution state adjustment Strategy 2

线程池执行状态 2
buildExecutor(core,max,size)表示构造线程池，core、max、size 分别表示核心线程数、最大线程数与队列容量
setMaximumPoolSize(size)表示设置线程池的最大线程数
getQueueSize()表示获取当前线程池大小
getTask()表示获取任务线程，内部打印线程池状态、休眠 1 000 ms
execute(task)表示交由线程池执行任务，task 表示 Runable 对象
executor :=buildExecutor(2,5,10); //创建一个核心线程数、最大线程数线程池，阻塞队列容量分别为 2,5,10 的线程池
For i: = 1 To 20 //循环执行任务
If<executor.getQueueSize() >= 10 * 0.8> Then //判断阻塞队列容量是否达到 80%
setMaximumPoolSize(10); //线程池扩容
Runnable task:=getTask() //获取可执行任务
executor.execute(task); //交由线程池执行任务
For End //循环结束

通过动态调整线程池状态可以规避项目中由于线程资源的配置不足导致的业务故障，但是通过单一阈值自动调整线程池参数并不能直观的展示线程池内部状态和线程池变化过程，所以引入监控中心与报警中心模块来动态调整监控线程池，具体结构如图 5 所示。

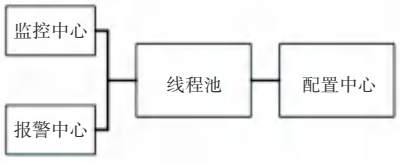

图 5 动态调整监控线程池结构图

Fig. 5 Dynamically adjust the structure of the monitoring thread pool

为了实时同步修改线程池的参数还需要引入配置中心模块,以便快速的调整线程池参数。配置中心能集中管理线程池的各个参数,在保证线程池执行任务高可用特性的同时可以实现线程池容量的实时更新,以匹配线程池处理任务的吞吐量。配置中心与应用保持长链接,以保证配置中心的改动能够及时的同步到应用端。监控中心实时获取到的数据可以根据时间分别选择日志记录或数据库持久化,为后期数据分析提供数据支持。预警中心基于监控平台数据结合业务需求设置合理的参数阈值,当所设参数达到预警阈值,对相关责任人进行预警通知。

3 实验结果与分析

实验模拟线程池执行任务的内部状态。实验一:线程池参数设置:corePoolSize = 2、maxPoolSize = 5、queueSize = 10,设置拒绝策略为 AbortPolicy,假设每执行一个任务需要 1 s,一次性提交 20 个任务,此时线程池同一时刻最大能执行的任务数是 15,预期当达到线程池容量上限的时候程序就会抛出异常,导致 5 个任务无法执行,执行 50 次,结果如图 6 所示符合预期。

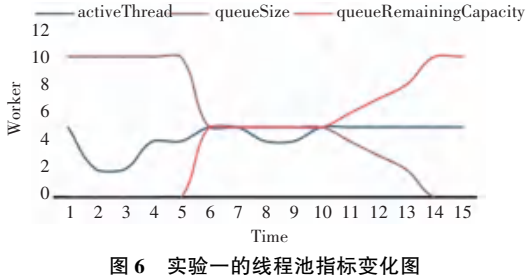

图 6 实验一的线程池指标变化图

Fig. 6 Thread pool indicator change graph for experiment one

实验二:线程池参数设置与实验一相同,不同的是当队列容量达到 80% 的时候,动态调整 maxPoolSize 为 10,此时预期线程池的预期最大执行能力为 20 个任务,线程池将刚好完成 20 个任务,不会抛出异常,如图 7 所示。实验二核心线程数由 5 到 10 的变化过程,实验数据变化并非线性变化,原因是线程并发执行,先后顺序不可控,所有任务都完成未产生异常,符合预期实验结果。

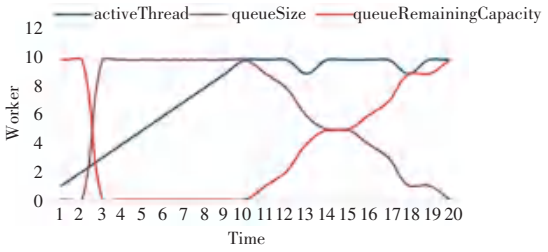

图 7 实验二的线程池指标变化图

Fig. 7 Thread pool indicator change graph for experiment two

实验表明,当线程池任务激增,在没有调整线程池执行能力的情况下,线程池中工作线程处于满负荷工作状态,缓冲队列被压满,多余任务被抛弃程序发生异常。而在引入动态调整之后,当线程池中工作线程达到一定阈值之后动态扩容,线程池的最大线程数增加,虽然队列在一定时间内处于队满状态,但是随着新增线程执行任务,缓解线程池压力,队列压力逐渐降低,所有任务均平滑执行完成,程序亦无异常。上述实验证明了在线程池运行过程中动态修改其参数的可行性。

4 结束语

在软件开发中线程池的应用普遍而又难以全面掌控,原因在于异步执行的任务不同性能的机器往往会有不同的表现,由于缺乏直观的数据展示,开发人员对线程池的整体状态难以全面掌握。本文结合线程池内部运行原理与开发中的具体案例,通过实验数据验证在线程池运行过程中,可通过配置中心动态修改线程池参数,使线程池性能达到瓶颈时动态调整容量,以缓解线程池状态与任务混乱的问题。

参考文献

[1] 赵姗, 杨秋松, 李明树. 性能非对称多核处理器下异构感知调度技术[J]. 软件学报, 2019, 30(4): 1164-1190.

[2] 陈佳楠, 李哲, 李占山. 基于多核 CPU 的表约束并行传播模式研究[J]. 软件学报, 2021, 32(9): 2769-2782.

[3] 李哲,于哲舟,李占山. 一种高效的 FDE 并行传播算法[J]. 软件学报,2023,34(9):4153-4166.

[4] MINUTOLI M, CASTELLANA V G, SAPORETTI N, et al. Svelto: High-level synthesis of multi-threaded accelerators for graph analytics[J]. IEEE Transactions on Computers, 2021, 71(3): 520-533.

[5] PAPADOPOULOS A V, AGRAWAL K, BINI E, et al. Feedback-based resource management for multi-threaded applications[J]. Real-Time Systems, 2023, 59(1): 35-68.

[6] 周彭, 左志强. 基于多线程并行的符号执行引擎设计与实现[J]. 计算机研究与发展, 2023, 60(2): 248-261.

[7] INVERSO O, TOMASCO E, FISCHER B, et al. Bounded verification of multi-threaded programs via lazy sequentialization [J]. ACM Transactions on Programming Languages and Systems

(TOPLAS)，2021，44(1)：1-50.
[8] WANG Z Y，HAN Z，YAN L，et al. Cooperative scheduling of multi-core and cloud resources：multi-thread-based MCC offloading strategy[J]. Communications Iet，2019，13(14)：2146-2154.
[9] HERDT V，LE H M，GROßE D，et al. Combining sequentialization-based verification of multi-threaded C programs with symbolic partial order reduction[J]. International Journal on Software Tools for Technology Transfer，2019，21(5)：545-565.
[10] JIANG X，JI D，GUAN N，et al. Real-time scheduling and analysis of processing chains on multi-threaded executor in ROS 2[C]//Proceedings of 2022 IEEE Real-Time Systems Symposium (RTSS). IEEE，2022：27-39.
[11]谢浩山，刘晓楠，赵晨言，等. 基于 OpenMP 并行模型下 HHL 算法的经典模拟实现[J]. 计算机科学，2022，49(S2)：914-918.
[12]王建伟，郭春喜，白穆，等. 海量 IGS 数据实时线程池并发获取[J]. 测绘通报，2020，525(12)：93-96，105.
[13]SIDORENKO V G，AUNG K M，PETROV A S. Automation of subway train scheduling based on the multi-threaded software product[C]//Proceedings of 2020 International Multi-Conference on Industrial Engineering and Modern Technologies. IEEE，2020：1-6.
[14] TIAN Z，WANG Q，GAO C，et al. Plagiarism detection of multi-threaded programs using frequent behavioral pattern mining[J]. International Journal of Software Engineering and Knowledge Engineering，2020，30(11-12)：1667-1688.
[15]刘道清，扈红超，霍树民. 面向容器运行时安全威胁的 N 变体架构[J]. 计算机科学，2024，51(6)：399-408.
[16]YU T，PETOUMENOS P，JANJIC V，et al. Colab：A collaborative multi-factor scheduler for asymmetric multicore processors[C]//Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization. 2020：268-279.
[17]FAN W，ZHANG J，FAN X J，et al. Factual testing on multi-threaded scheduling for network convergence efficiency[C]//Proceedings of 2022 IEEE 10th Joint International Information Technology and Artificial Intelligence Conference. IEEE，2022：440-443.
[18]童林，陈庆奎. 一个并发数据流接入与动态配置模型[J]. 智能计算机与应用，2024，14(2)：106-113.
[19]LI T，NIE Y，WANG Y，et al. A dense sample surface defect detection algorithm based on deep learning and thread pool[C]//Proceedings of 2023 5th International Conference on Frontiers Technology of Information and Computer. IEEE，2023：1159-1163.
[20]胡汝博，陈庆奎. 面向并发数据流处理的存储接入方法[J]. 智能计算机与应用，2024，14(3)：37-45.