面向高并发的 Java 应用性能提升方法研究

段庆军 [1]
DUAN Qingjun

摘 要 面向高并发场景下 Java 应用经常出现的性能瓶颈问题，文章开展了系统性的优化实践工作。借助监控体系的搭建以及压力测试的执行工作，把性能瓶颈精准定位到线程模型中的锁竞争以及内存管理中的垃圾回收压力。为处理这类问题，从多维度的综合优化部署工作。在并发层面，选用线程池隔离、异步编程模型以及无锁数据结构；在内存管理层面，对 JVM 堆内存参数进行精细化调整，并且从 G1 垃圾收集器迁移到 ZGC，同时借助堆外内存技术分担 GC 压力。优化后的数据显示，应用的最大吞吐量提高约 89.9%，99 百分位响应时间降低约 77.3%，最大 GC 停顿时间被缩短至 5 ms 以内。这次实践验证了所提方法的有效性，并成功把系统瓶颈由应用层迁移到数据层。

关键词 高并发；Java 性能；内存管理；异步编程；垃圾回收

doi：10.3969/j.issn.1672-9528.2026.06.016

0 引言

随着互联网业务的发展，Java 应用在高并发以及低延迟场景下的性能表现变得愈发重要。但在实际生产环境中，这类应用常因并发处理策略不合理、内存管理机制不完善，导致性能瓶颈凸显，具体表现为吞吐量下降、响应时间延长以及系统运行不稳定。为了应对这类挑战，一个典型的高并发 Java 应用当作研究对象来使用，旨在探索出一套成体系的性能瓶颈定位以及优化方法。借助结合实时性能剖析、并发压力测试等手段，深入分析线程阻塞以及垃圾回收等深层次缘由，同时据此提出并落实一系列针对性的并发模型以及内存管理优化方案，以期显著提高应用在高负载下的核心性能指标，为同类系统的性能调优工作提供工程实践参考 [1]。

1 应用性能瓶颈分析

1.1 性能剖析与监控系统构建

为准确定位系统性能瓶颈，首先构建了一套覆盖应用、（Java virtual machine, JVM）及操作系统层面的监控体系。该体系利用 Prometheus 收集时间序列数据，通过 Grafana 进行可视化展示，并结合 Arthas 等字节码增强工具进行实时应用诊断。监控系统核心指标包括：系统吞吐量（transactions per second, TPS）、请求响应时间（99th percentile latency）、JVM 堆内存使用率、垃圾回收（garbage collector，GC）活动频率与停顿时间、线程池状态以及中央处理器（central processing unit, CPU）利用率 [2]。初步监控数据显示，在高并发场景下，系统 TPS 增长乏力，响应时间出现明显抖动，CPU 利用率在部分时段达到 90% 以上，同时伴有频繁的 Young GC 和偶发的 Full GC 事件，单次 Full GC 停顿时间超 500 ms。

1.2 并发压力测试与瓶颈定位

基于监控系统提供的基线数据，使用 Gatling 设计并执行了阶梯式并发压力测试，模拟从 1 000~10 000 个虚拟用户逐步增加的负载。测试场景聚焦于核心的订单创建接口，该接口涉及库存查询、价格计算、用户验证等多个内部服务调用。压力测试期间，通过对线程堆栈进行采样分析，发现大量线程处于 BLOCKED 状态，等待获取同一个锁资源，该资源与一个全局订单号生成器相关。同时，对 JVM 的内存剖析显示，订单数据传输对象等临时对象的创建速率极高，在 10 000 个并发用户时，对象的分配速率达到约 800 MB/s，这直接导致了 Young Generation 区域的频繁 GC。

2 并发模型与异步处理优化

2.1 线程池参数合理化与隔离设计

针对应用中线程资源使用不当的问题，对原有的全局共享线程池进行了拆分与参数优化。根据业务逻辑的特性，创建了两种独立的线程池：CPU 密集型线程池与 IO 密集型线程池。IO 密集型业务（如数据库访问、外部服务调用）的线程池核心线程数根据特定公式进行设定。线程数的计算公式为：

1. 大庆油田有限责任公司试油试采分公司信息中心
黑龙江大庆 163000

$$N_{threads} = N_{cpu} \times U_{cpu} \times (1+\frac{W}{C}) \quad (1)$$

式中：$N_{threads}$ 为核心线程数，N_{cpu} 为 CPU 核心数，U_{cpu} 为目标 CPU 使用率，W 为线程等待时间，C 为线程计算时间。

在一个 16 核 CPU 的服务器上，目标 CPU 使用率设为 0.8，经性能剖析测得 IO 操作的等待与计算时间比值（W/C）约为 4。应用该公式计算得出 IO 密集型线程池的核心线程数约为 16×0.8×(1+4) = 64。对于 CPU 密集型任务（如复杂计算），线程池核心线程数设定为 CPU 核心数加一，即 17。同时，为避免任务堆积导致内存溢出，线程池选用了容量有限的链式阻塞队列，并配置了“调用者运行”拒绝策略，在高负载时将任务执行压力反馈给调用方线程，增强系统的自我保护能力。为了便于问题排查与性能监控，还为不同的线程池配置了自定义的线程工厂，使得每个线程都带有明确的业务前缀。此外，线程池的 keepAliveTime 参数也进行了差异化设置，IO 密集型线程池设置了较长（如 60 s）的存活时间，以复用空闲线程，而 CPU 密集型线程池则设置了较短的时间，以在任务量下降时快速释放核心资源。

2.2 基于异步编程模型的应用

订单创建流程中包含了对库存、用户和风控等多个下游服务的远程过程调用（remote procedure call, RPC），这些调用在原始设计中是同步串行执行的，总耗时等于各调用耗时之和，成为响应时间的主要瓶颈 [3-4]。为优化此流程，引入了 Java 8 的异步化编程工具进行改造。将每个独立的 RPC 调用使用该异步编程工具的工厂方法进行封装，并提交到前述定义的 IO 密集型线程池执行。例如，获取库存信息和用户信息的操作可以并行发起。之后，使用其提供的组合算子等待所有并行的异步任务完成，并对结果进行聚合处理。为了增强系统的健壮性，为每个异步调用都增加了异常处理逻辑。通过链接异常处理回调，即使某个下游服务调用失败（例如超时或返回错误码），整个组合任务也不会被中断，而是可以返回一个默认值或记录一个特定的失败状态，从而保证主流程的可用性。同时，为防止某个慢服务拖垮整个请求，还为每个异步任务设置了独立的超时机制。利用一个独立的、低优先级的调度线程池，为每个异步操作注册一个超时检查任务，若在规定时间（例如 200 ms）内未完成，则会以编程方式使其以超时异常结束，避免了无限期的等待。通过将串行等待转变为并行执行，多个 IO 等待时间得以重叠，理论上可以将此部分的耗时降低到所有 RPC 调用中最长的一个 [5]。

2.3 无锁数据结构与原子操作实践

压力测试中定位的全局订单号生成器锁竞争问题，是由于使用了同步关键字保护一个共享的序列号变量。在高并发下，该全局锁点成为了严重的性能瓶颈。为消除这种显式锁带来的线程上下文切换和调度开销，采用无锁编程思想对其进行改造 [6]。利用 Java 并发包中的原子长整型类替换了原有的 long 类型序列号。该原子类的原子性自增并获取值的方法基于（compare-and-swap, CAS）指令实现，这是一种硬件级别的原子操作。CAS 操作的过程是：处理器在修改共享变量时，会先比较内存中的值与自己预期的旧值是否一致，只有在一致的情况下才会执行更新，否则操作失败并进行重试。这个过程设定为乐观的，其假设冲突很少出现，所以规避了无竞争状态下开展加锁以及解锁操作的额外开销。在订单号生成的具体场景里，因为竞争程度非常高，单纯选用 CAS 自旋的方式可能会消耗掉大量 CPU 资源。所以，后续进一步优化为选用一个性能更高的原子累加器来使用。原子累加器的核心优势，在于其内部采用的分散计算核心思想。其内部包含一个基础数值以及一个单元格数组。处在无竞争或者低竞争的状态时，更新操作会直接作用在基础值上。当竞争程度加剧的时候，线程会借助哈希算法被分配到不同的单元格上开展独立的累加操作。只有在读取总和数值的时候，才会把基础值以及所有单元格的数值进行汇总操作 [7]。该累加器会在内部维护一个 Cell 数组，开展并发更新工作时，线程凭借哈希计算分散到不同的 Cell 上进行累加，最后借助求和方法汇总所有 Cell 的数值。

3 内存管理与垃圾回收调优

3.1 JVM 堆内存结构与参数配置

系统所面对的高对象分配率相关问题，其根源在于 JVM 堆内存的配置工作没有契合业务负载的特性。最初的配置工作把 JVM 堆的初始大小以及最大大小都设为 4 GB，还选用了默认的新生代以及老年代比例。高并发场景下，大量生命周期较短的订单数据传输对象很快填满了 Eden 区，引发了频繁的 Minor GC[8]。部分对象因为 Survivor 区的空间不够或者达到晋升年龄阈值，过早地被晋升到 Old Generation，最终引发了耗时更长的 Major GC 或 Full GC。针对这个问题，相关人员对堆内存参数开展了调整工作。首先，把堆的初始大小以及最大大小都扩大到 8 GB，为对象的分配工作提供更充足的空间。其次，借助调整 JVM 参数的方式，把新生代以及老年代的大小比例设置为 1:2，即新生代大概占 2.67 GB。比例调整工作基于 Java Flight Recorder 以及 VisualVM 等工具对应用运行时对象生命周期的详细分析，超过 95% 的对象在第一次 Minor GC 之后就可以被回收。除此之外，相关人员还对对象的晋升阈值开展了微调工作。默认设置下，对象需要经历 15 次 Minor GC 才能完成到老年代的晋升。借助分析可以发现，少量存活时间稍长的对象（比如会话相关的信息）在第 5 到 6 次 GC 之后依旧存活，但是最终还是会被回收。所以，把最大晋升阈值从 15 下调到 6，让这些中期对象也能在新生代被回收，进一步降低了对老年代的内存污染。最后，

凭借调整相关参数的操作，将 Eden 区以及单个 Survivor 区的空间比例维持在 8:1。

3.2 面向低延迟场景的 GC 选型与调优

尽管开展堆内存结构的调整工作缓解了 GC 压力，原有选型 G1（Garbage-First）垃圾收集器在峰值负载场景下，GC 的 STW（Stop-The-World）停顿时长突破 100 ms，无法满足业务低延迟指标。G1 虽为服务端专用垃圾收集器，但在对象高速分配的极端场景中，混合回收效率跟不上老年代内存上涨速率，极易触发 Full GC[9-10]。为获得更低且稳定性更强的 GC 停顿指标，方案将垃圾收集器替换为 ZGC。ZGC 自 JDK11 版本推出，是面向高伸缩场景的低延迟垃圾收集器，设计目标将 GC 停顿控制在 10 ms 以内。借助调整 JVM 启动参数来启用 ZGC，并且分配 10 GB 的最大堆内存，系统正式切换至新的收集器方案。切换至 ZGC 还需要对其并发执行的线程数进行合理配置。ZGC 的并发标记和整理工作是由其内部的 GC 线程完成的，这些线程会与应用线程竞争 CPU 资源。若配置不当，可能影响应用吞吐量。通过设置并发 GC 线程数参数，将其值设定为 CPU 核心数的 1/4 左右，例如在一个 16 核服务器上设置为 4。还需要对其并发执行的线程数进行合理配置。ZGC 的并发标记和整理工作是由其内部的 GC 线程完成的，这些线程会与应用线程竞争 CPU 资源。若配置不当，可能影响应用吞吐量。通过设置并发 GC 线程数参数，将其值设定为 CPU 核心数的 1/4 左右，例如在一个 16 核服务器上设置为 4。ZGC 的关键技术，如着色指针和读屏障，使得并发标记、并发整理等几乎所有工作都能在 Java 线程运行时进行，极大地缩短了 STW 的时间。切换到 ZGC 后，即使在最高的并发压力下，通过 GC 日志分析，观测到的最大 GC 停顿时间也稳定在 5 ms 以下，彻底解决了由 GC 停顿引起的响应时间抖动问题。

3.3 堆外内存使用与 GC 压力缓解

为了从根本上减少 JVM 堆内存的对象数量，从而降低 GC 的频率和压力，引入了堆外内存技术来管理一部分生命周期较长且数据结构固定的缓存数据 [11]。应用中存在一个产品信息本地缓存，该缓存之前使用一个线程安全的哈希表实现，存储了数百万个产品信息对象实例，占用近 1 GB 的堆内存。这些对象会在老年代中长期留存，进而成为 GC 运行过程中的负担。优化方案选用开源缓存库 Ehcache 3，并且将其进行配置，作为堆外存储使用。数据先进行序列化处理，再借助 Java NIO 接口分配的直接内存区域来开展存储工作。序列化方式选用了效率更高的第三方序列化框架 Kryo，Kryo 基于字节来开展相关设计，提供了紧凑的序列化格式，其序列化以及反序列化的速度比 Java 默认的处理方式要快数倍。同时，为了开展部分不受 GC 控制的内存的精细化管理工作，相关人员对 Ehcache 的堆外存储大小进行了严格限制，同时还为其进行配置了 LRU 淘汰策略。当堆外内存的使用量达到预先设定的上限，如 1.5 GB 时，最长未被访问的数据会被自动驱逐，以此防止堆外内存的无限增长以及潜在的内存泄漏问题。当应用要开展缓存数据的访问工作时，相关数据会被反序列化为对应对象。虽然这样做引入了序列化以及反序列化的额外开销，但是对于读多写少的缓存场景来说，其带来的影响是可控的 [12]。借助部分缓存数据移出堆内的操作，JVM 堆中老年代的占用率降低了约 1 GB，减少了 Mixed GC 以及 Full GC 的触发可能性。而堆外内存不受 GC 的管理，其分配以及释放工作需要手动开展或者借助框架进行精确控制，有效避免了内存泄露的风险。

4 性能提升效果验证与分析

4.1 关键性能指标优化前后对比

为量化性能提升的效果，选用和前期压力测试完全相同的硬件环境以及测试脚本，对完成优化后的应用系统开展全面的性能回归测试工作。测试过程记录了多个关键性能指标在优化前后的变化情况。把核心业务接口的性能对比数据整理在表 1 中进行展示。

表 1 优化前后关键性能指标对比

指标	优化前	优化后	提升 / 降低率
最大吞吐量 /TPS	4 821	9 156	+89.9%
平均响应时间 /ms	185	72	-61.1%
99 百分位响应时间 /ms	652	148	-77.3%
最大 GC 停顿时间 /ms	530	4.8	-99.1%

数据显示，系统最大吞吐量从 4 821 TPS 提高到 9 156 TPS，增幅约为 89.9%。平均响应时间从 185 ms 极大程度上降低到 72 ms。更为关键的 99 百分位响应时间，从 652 ms 优化到 148 ms，表明系统在处理绝大多数请求工作时都能提供更快的响应，消除了长尾请求带来的不良用户体验。最大 GC 停顿时间从 530 ms（G1 GC 下的 Full GC）降低到 4.8 ms（ZGC），显示出新 GC 策略在控制应用暂停方面的有效性。

4.2 系统资源利用率与稳定性评估

除吞吐量以及延迟之外，系统运行的稳定性也是开展优化效果评估工作的重要维度。借助优化前后在 10 000 名并发用户持续加压 30 min 期间的系统资源监控数据开展对比工作，分析了资源利用率的具体变化。优化前后 CPU 利用率的对比结果如图 1 所示。

优化前的老年代内存占用率呈现稳步上升态势，最终由于触发 Full GC 出现断崖式下跌的情况。优化后，鉴于对象晋升率降低以及堆外内存的运用，老年代内存占用的增长速度极为缓慢，系统可以在更长时段内保持稳定运行，不需要开展全局垃圾回收的相关工作。我们把相关资源利用率的统

计数据整理后，以表 2 的形式进行呈现。

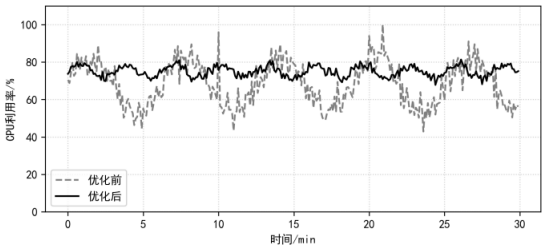

图 1 优化前后高负载下 CPU 利用率对比

表 2 优化前后高负载下系统资源利用率对比

资源指标	优化前	优化后
平均 CPU 利用率	68%	75%
峰值 CPU 利用率	96%	82%
CPU 利用率标准差	15.2%	3.5%
30 min 内 Full GC 次数	5	0
老年代峰值占用率	92%	45%

4.3 可伸缩性测试与瓶颈迁移分析

为了评估系统的可伸缩性，执行了更大规模的并发用户测试，用户数从 5 000 逐步增加至 20 000。系统的吞吐量随并发用户数变化的趋势如图 2 所示。

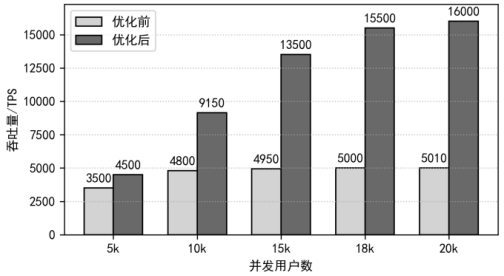

图 2 系统可伸缩性测试结果

从图 2 中可以看出，优化后的系统在用户数从 5 000 增长到 15 000 的过程中，TPS 呈现出近似线性的增长关系，表现出良好的水平扩展能力。然而，当并发用户数超过 18 000 时，TPS 的增长开始放缓，响应时间也随之开始延长。此时对系统进行分析，发现应用服务器的 CPU 和内存资源仍有余量，但数据库连接池的等待队列开始积压，数据库服务器的 IOPS 接近饱和。这一现象表明，原有的应用层性能瓶颈已经成功消除，系统的瓶颈已经迁移到了下游的数据库层面。这符合系统优化的普遍规律，即性能瓶颈会随着优化不断转移。

5 结论

研究工作成功地对一个面临高并发瓶颈的 Java 应用实施了全面的性能提升。通过对线程模型和内存管理的深度优化，证明了线程池隔离、异步化改造与无锁化设计相结合的并发策略，以及 JVM 堆结构调优、切换至低延迟垃圾收集器 ZGC 与应用堆外内存相结合的内存策略，是解决此类问题的有效途径。最终的性能验证结果表明，这些综合措施不仅大幅提升了系统的吞吐量与响应速度，还显著增强了运行稳定性。应用瓶颈从自身转移至下游依赖，也验证了本次优化工作的彻底性。该研究形成了一套从瓶颈分析到方案实施再到效果验证的完整优化流程，其技术方案与调优思路具有较强的实践指导意义。

参考文献：

[1] 赵博宇，余科谊，常振云，等 . 基于 Kubernetes 的 Java Web 应用系统高并发部署设计与优化分析 [C]// 中国高校校办产业协会终身学习专业委员会 . 第四届教育信息技术创新与发展学术研讨会论文集 . 北京：《中国教育信息化》杂志社，2025:841-849.

[2] 关安青 . 基于 Java 的电商软件开发与大数据分析 [J]. 数字技术与应用 ,2025,43(5):193-195.

[3] 王晶，杨豫娇 .Java 技术在分布式系统开发中的创新应用 [C]// 广西网络安全和信息化联合会 . 第九届工程技术管理与数字化转型学术交流会论文集 . 哈尔滨：哈尔滨信息工程学院 ,2025:129-130.

[4] 张烁 . 基于微服务架构的企业级软件开发实践探讨 [J]. 中国信息界 ,2024(2):48-50.

[5] 李方方，周亚凤，王校建，等 . 高并发条件下消息队列的设计与实现 [J]. 黑龙江科学 ,2023,14(24):72-74.

[6] 周亮 . 计算机软件开发中 Java 编程语言的应用研究 [J]. 科技资讯 ,2024,22(13):39-41.

[7] 王原昭，卢春雨，蒲鹏 . 面向高并发在线考试系统的性能优化 [J]. 软件 ,2024,45(2):14-18.

[8] 尹应荆 . 基于 Java 多线程的数据同步探究 [J]. 信息与电脑 (理论版),2024,36(10):143-145.

[9] 董庆杰 . 高并发场景下软件架构的实现与优化 [J]. 长江信息通信 ,2023,36(12):94-96.

[10] 张平，李慧春 . 高并发性能问题优化方案研究 [J]. 信息技术与信息化 ,2023(11):166-169.

[11] 王翔 . 高并发访问下的分布式系统架构设计 [J]. 智能城市，2023, 9(3):5-7.

[12] 张国芳 .Java 编程语言在计算机软件开发中的应用方向分析 [J]. 信息记录材料 ,2023,24(11):138-141.

【作者简介】

段庆军（1981—），男，黑龙江大庆人，硕士研究生，中级工程师，研究方向：信息工程软件技术方向。

（收稿日期：2025-10-23 修回日期：2026-06-05）