摘要：在当前的软件系统开发中，多任务处理已经是提升系统性能和响应速度的关键技术之一。然而，传统的线程模型在资源占用、上下文切换等方面存在一定的局限性。随着技术的发展，协程作为一种轻量级的并发模型逐渐受到关注。本文将对Java中的线程与协程进行性能对比分析，探讨它们在资源占用、上下文切换、并发性能等方面的差异，以期为开发系统提供选择合适并发模型的依据。

关键词：Java协程，Java线程，并发性能

Abstract：In current software development, multitasking is a pivotal technology for enhancing system performance and responsiveness. However, traditional thread models have certain limitations in terms of resource consumption and context switching. with technological advancements, coroutines, as a lightweight concurrency model, have garnered attention. This paper aims to compare the performance of threads and coroutines in Java, exploring their differences in resource consumption, context switching, and concurrency performance, aiming to guide developers in selecting the appropriate concurrency model.

1.引言

Java作为一门广泛使用的编程语言，其线程模型在多任务处理中扮演着重要角色。然而，线程的创建和销毁开销大，上下文切换成本高，尤其在处理大量并发任务时，这些问题尤为突出。相比之下，Java协程，作为一种用户态的轻量级线程，被视为解决高并发问题的新途径。尽管Java标准库目前并未直接支持协程，但借助第三方库如Quasar或Project Loom（实验性项目），Java开发者也能体验到协程带来的便利。本文将对Java协程与线程的性能进行对比分析，以期有助于在开发系统时更好地理解和选择适合的并发编程方式。

2.Java协程与线程概述

2.1线程

Java的线程模型基于操作系统内核线程，每个Java线程对应一个操作系统线程。这种模型在多核CPU环境下能够实现真正的并行处理，但同时也带来了较高的上下文切换开销和资源消耗。

2.2协程

Java协程是一种用户态的轻量级线程，其调度不依赖于操作系统，由Java虚拟机管理。协程可以在同一线程内暂停和恢复执行，共享栈空间，从而大大减少了上下文切换和内存占用。

3.性能对比分析

3.1并发性能

在并发性能方面，线程和协程各有优势。线程可以充分利用多核处理器的并行执行能力，适用于需要并行执行多个阻塞或计算密集型任务的场景。而协程则更适合处理IO密集型任务，通过避免不必要的线程切换和上下文切换开销，提高并发性能。

然而理论上，在高并发场景下，线程的创建和销毁等操作会带来较大的性能和资源消耗，限制了其扩展性。而协程则是一种轻量级的并发编程方式，即使创建了上千个协程，对于系统来说也不是很大的负担，通过有效的调度策略（如协作式调度），依然能有效提升系统的并发处理能力，尤其是在I/O密集型应用中表现突出。

3.2内存占用

在线程编程中，每个Java线程都有自己的执行上下文和栈空间，默认占用约1MB的堆外内存（不含栈内存），在大规模并发场景下会迅速消耗系统资源。而协程则是一种轻量级的线程，每个协程只需要少量的栈空间，内存占用远低于线程，通常只有几十到几百字节，极大降低了内存使用。

3.3实验设计与结果分析

为了更准确地评估Java协程与线程的性能差异，以下进行了一组实验进行对比测试。分别使用Java线程和Java协程（虚拟线程）实现了相同的并发任务，并记录了任务的执行时间、内存消耗等性能指标。

3.3.1运行环境与工具：

笔记本为搭载Intel Core i5处理器和16GB内存的计算机，操作系统为Windows 10

Java21

Spring Boot 3.3.1

3.3.2测试代码

带虚拟线程的Spring Boot程序：

1. */\*\**
2. \* @description: 虚拟线程测试
3. \* @author: yang.jiuju
4. \* @time: 2024/6/29
5. \*/
6. public String runVT(long count) {
7. long startTime = System.currentTimeMillis();
8. try (ExecutorService executorService = Executors.newVirtualThreadPerTaskExecutor()) {
9. for (int i = 0; i < count; i++) {
10. executorService.submit(() -> {
11. try {
12. Thread.sleep(TWO\_SECOND);
13. } catch (InterruptedException ignored) {
14. }
15. });
16. }
17. Long tm = System.currentTimeMillis() - startTime;
18. System.out.printf("Thread(virtual) is completed,threadCount: %d,, spend time: %d ms.\n", count, tm);
19. return tm.toString();
20. }
21. }

Java线程的Spring Boot测试程序：

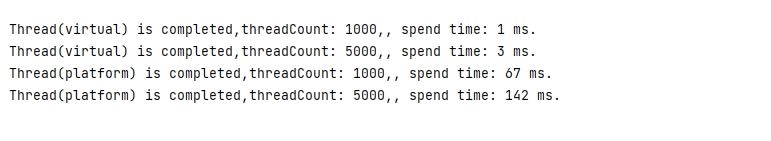
1. */\*\**
2. \* @description: 平台线程测试
3. \* @author: yang.jiuju
4. \* @time: 2024/6/29
5. \*/
6. public String runThread(int count) {
7. long startTime = System.currentTimeMillis();
8. try (ExecutorService executorService = Executors.newFixedThreadPool(count)) {
9. for (int i = 0; i < count; i++) {
10. executorService.submit(() -> {
11. try {
12. Thread.sleep(TWO\_SECOND);
13. } catch (InterruptedException ignored) {
14. }
15. });
16. }
17. Long tm = System.currentTimeMillis() - startTime;
18. System.out.printf("Thread(platform) is completed,threadCount: %d,, spend time: %d ms.\n", count, tm);
19. return tm.toString();
20. }
21. }

3.3.2测试结果

通过API访问接口测试耗时

1. public static int TWO\_SECOND = 2000;
2. @ResponseBody
3. @GetMapping("/all/{count}")
4. public Boolean runAll(@PathVariable(value = "count") Integer count) {
5. service.runVT(count);
6. service.runVT(count \* 5);
7. service.runThread(count);
8. service.runThread(count \* 5);
9. return true;
10. }

具体耗时见下图：



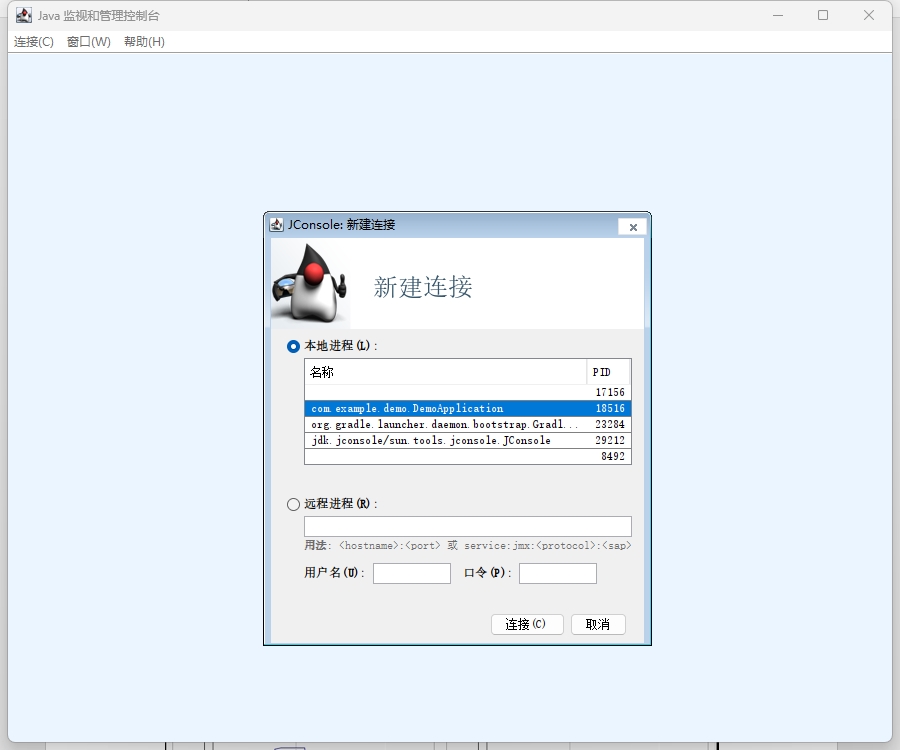
可以看出，在开启1000或者5000个协程/线程的情况下，协程的耗时都小于Java线程的耗时。

通过JConsole监控工具，观察Java协程与Java协程的资源使用情况

以下是API访问接口测试资源使用

1. @ResponseBody
2. @GetMapping("/virtual/{count}")
3. public String runVT(@PathVariable(value = "count") Integer count) {
4. return service.runVT(count);
5. }
6. @ResponseBody
7. @GetMapping("/platform/{count}")
8. public String runThread(@PathVariable(value = "count") Integer count) {
9. return service.runThread(count);
10. }

在测试开始前，启动测试程序的同时，通过Java监控管理工具JConsole连接到测试程序，如下图：



连接成功后，具体测试步骤：

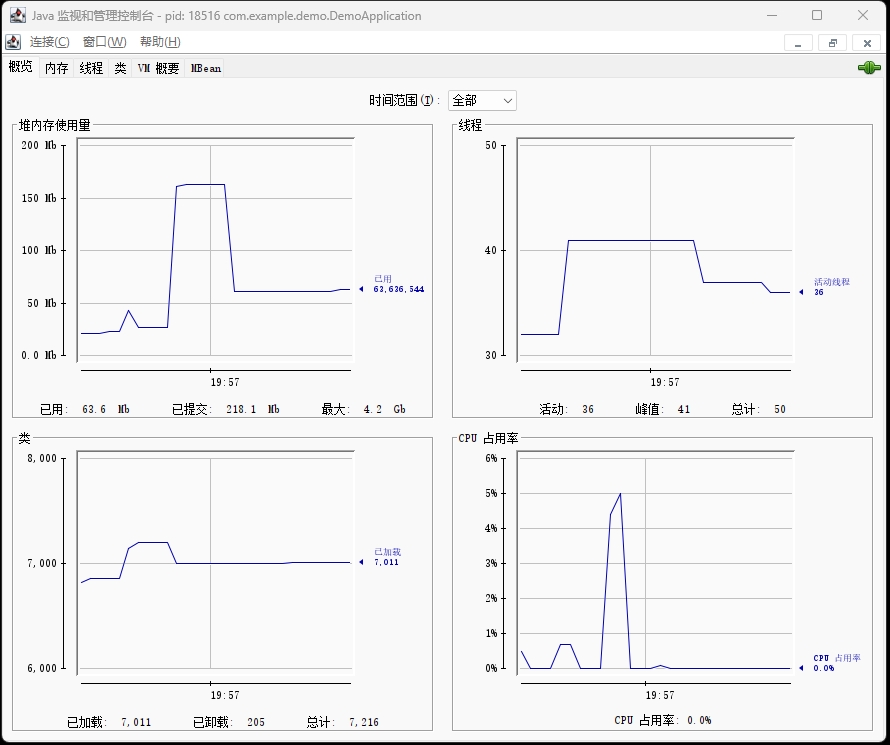
步骤一：发送一次接口请求，入参count为10000；

步骤二：等待5~10秒；

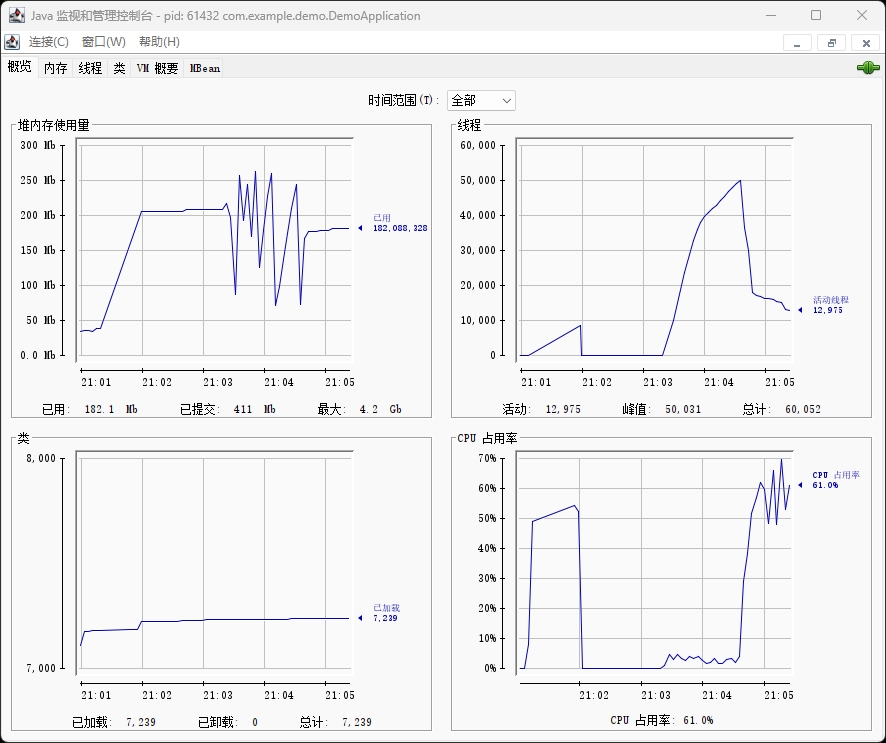
步骤三：发送一次接口请求，入参count为50000；

以上测试步骤针对Java虚拟线程（协程）和Java线程均保持一致。

Java虚拟线程（协程）两次请求接口的资源使用情况如下：



Java线程两次请求接口的资源使用情况如下：

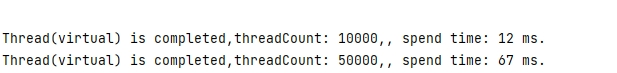


以上两张虚拟机运行数据，用以下表格列出接口请求时的高峰期对比数据：

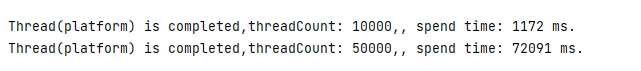
|  |  |  |
| --- | --- | --- |
|  | Java虚拟线程（协程） | Java线程 |
| 内存（Mb） | 160 | 250 |
| 线程数（个） | 41 | 50031 |
| CPU（%） | 5 | 70 |

以上两组测试的耗时打印截图如下：

Java虚拟线程（协程）耗时：



Java线程耗时：



可以明显看出，在开启上万个线程的情况下，Java线程明显占用了更多的资源（内存/CPU）。在测试5万个线程提交时，能明显感觉到笔记本运行时出现了卡顿问题。

测试结果表明，在处理大量并发任务时，Java协程模型相比传统线程模型有更低的平均响应时间和更高的吞吐量。协程的轻量级特性使其能够更快速地完成任务切换，减少了上下文切换带来的开销。

在资源消耗方面，Java协程模型显示出更低的内存占用和CPU使用率。由于协程的调度和管理在用户空间进行，避免了频繁的系统调用，从而降低了资源消耗。

4.总结

通过对Java协程与线程的性能对比分析，我们可以得出以下结论：在处理IO密集型任务时，协程的性能是优于线程的。在实际开发中，开发者应根据具体的应用场景和需求选择合适的并发编程方式。

4.参考

1. 《Java并发编程实战》

2. 《深入理解Java虚拟机》

3. 《协程在Java中的应用研究》