性能分析

测试环境

服务器：

Intel(R) Xeon(R) Silver 4214R CPU @ 2.40GHz NVIDIA GeForce RTX 4090

性能分析工具

`async-profiler` 是一个用于分析 Java 应用程序性能的开源工具。它可以通过收集和分析应用程序的堆栈跟踪数据，生成用于可视化的 Flame Graphs，并提供有关应用程序瓶颈的深入洞察。

以下是 `async-profiler` 的一些特点和用法：

- 适用于 Java 8+：`async-profiler` 可以与 Java 8 及更高版本的应用程序一起使用。
- 低开销采样：它使用低开销的采样技术来获取应用程序的堆栈跟踪信息，几乎不会对应用程序的性能产生显著影响。
- 异步事件收集：`async-profiler` 可以异步地收集堆栈跟踪数据，从而避免在应用程序中引入显著的延迟。
- Flame Graphs：它生成可视化的 Flame Graphs，用于直观地表示应用程序的性能瓶颈和调用关系。
- 多种输出格式：`async-profiler` 可以生成多种输出格式，包括 SVG、HTML 和文本。
- 支持多种操作系统：它可以在多种操作系统上运行，包括 Linux、Mac 和 Windows。

`async-profiler` 的官方 GitHub 仓库：https://github.com/jvm-profiling-tools/async-profiler

安装与使用

1. 下载 `async-profiler`：您可以从 `async-profiler` 的 GitHub 仓库下载最新版本的发布包并解压。
2. 执行 `profiler.sh`：运行 `profiler.sh` 脚本，并根据需要提供适当的参数。

   检查系统设置：尝试运行以下命令以更改内核设置并允许对性能事件的访问。

```
sudo sysctl kernel.perf_event_paranoid=1
```

查看 java 进程的 PID（可以使用 jps ），运行 `profiler.sh` 脚本，生成 Flame Graph：

```
sudo ./profiler.sh -d 60 -f flamegraph.html <pid>
```

以上命令使用profiler.sh脚本采集指定进程的CPU profile数据，采样时间为60秒，采样结果保存为flamegraph.html文件。

中阶任务

编写JMH测试用例，在常见应用场景下（将mysql的同步操作提交到独立线程池，让协程异步等待独立线程池执行完毕，可以利用CompletableFuture实现），对比不同调度器（FixedThreadPool，ForkJoinPool）的性能表现。

```java
package com.example.benchmark;

import org.openjdk.jmh.annotations.*;

import java.sql.ResultSet;
import java.util.concurrent.*;

@Fork(1)
@Warmup(iterations = 3, time = 5)
@Measurement(iterations = 5, time = 5)
@State(Scope.Benchmark)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.SECONDS)
public class DatabaseBenchmarkTest {

    @Param({"1","2"})
    private int testOption;

    @Param({"30000"})
    private int threadCount;

    @Param({"300000"})
    private int requestCount;

    private ExecutorService dbExecutor;

    @Setup(Level.Trial)
    public void setup() {
        if (testOption == 1) {
            dbExecutor = Executors.newFixedThreadPool(threadCount);
            Thread.ofVirtual().scheduler(dbExecutor).start(() -> {
                try {
                    testDatabase();
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            });
        } else if (testOption == 2) {
            dbExecutor = new ForkJoinPool(threadCount);
            Thread.ofVirtual().scheduler(dbExecutor).start(() -> {
                try {
                    testDatabase();
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            });
        } else {
            throw new IllegalArgumentException("Invalid test option: " +
testOption);
        }
        ConnectionPool.initConnectionPool();
    }

    @TearDown(Level.Trial)
    public void teardown() {
```

```
        ConnectionPool.closeConnection();
        dbExecutor.shutdown();
        try {
            if (!dbExecutor.awaitTermination(1, TimeUnit.MINUTES)) {
                dbExecutor.shutdownNow();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
            dbExecutor.shutdownNow();
        }
    }

    @Benchmark
    public void testDatabase() throws InterruptedException {
        CountDownLatch latch = new CountDownLatch(requestCount);
        for(int i = 0; i < requestCount; i++) {
            CompletableFuture<String> cf = CompletableFuture.supplyAsync(() -> {
                String result = null;
                try {
                    result = execQuery("select * from hello");
                } catch (Exception e) {
                    e.printStackTrace();
                }
                latch.countDown();
                return result;
            });

        }
        latch.await();
    }

    public static String execQuery(String sql) {
        String queryResult = "";
        try {
            ConnectionNode node;
            do {
                node = ConnectionPool.getConnection();
            } while (node == null);
            ResultSet rs = node.stm.executeQuery(sql);

            while (rs.next()) {
                int id = rs.getInt("id");
                String hello = rs.getString("hello");
                String response = rs.getString("response");

                queryResult += "id: " + id + " hello:" + hello + " response: "+
response + "\n";
            }

            rs.close();
            ConnectionPool.releaseConnection(node);
        } catch (Exception e) {
            e.printStackTrace();
        }
```

```
        return queryResult;
    }
}
```

代码修改

问题一

```
Thread.ofVirtual().scheduler(dbExecutor).start(() -> {
                try {
                    testDatabase();
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            });
```

- Thread.ofVirtual()创建一个虚拟线程(协程)。
- 使用scheduler()方法将这个虚拟线程调度到指定的dbExecutor线程池中执行。
- start()方法启动这个虚拟线程,使其进入运行状态。
- 传入的lambda表达式就是这个虚拟线程需要执行的任务体 - 测试数据库方法testDatabase()。

这段代码就是指定dbExecutor作为这个协程的调度器,控制着协程的调度和执行流程。

我的理解是Thread.ofVirtual().scheduler(dbExecutor)的 dbExecutor 是用作虚拟线程的调度器，控制虚拟线程的执行环境；CompletableFuture.supplyAsync(... ,dbExecutor) 中的dbExecutor 则是用作异步任务的执行器，控制异步任务的执行线程。

问题二

```
    @Benchmark
    public void testDatabase() throws InterruptedException {
        CountDownLatch latch = new CountDownLatch(requestCount);
        for(int i = 0; i < requestCount; i++) {
            CompletableFuture<String> cf = CompletableFuture.supplyAsync(() -> {
                String result = null;
                try {
                    result = execQuery("select * from hello");
                } catch (Exception e) {
                    e.printStackTrace();
                }
                latch.countDown();
                return result;
            });

        }
        latch.await();
    }
```

原来这段代码删去了查询的后续操作，测试的计时范围只包括testDatabase函数本身的执行时间，而没有计时从发出请求到收到结果这段整个过程的时间，查找资料了解到真正比较不同调度器在数据库操作这类任务方面的性能，应该计算包含数据库操作时间在内的整个任务完成时间。

原因有以下几点:

1. 对于数据库操作这样的I/O密集型任务,数据库查询本身的latency 可能远大于调度等操作的时间。
2. 仅测量中间操作的时间无法反应任务完整周期的性能表现可能会误导结论。
3. 不同调度算法可能会影响数据库连接的利用率,最终影响整体吞吐率
4. 常规应用场景下用户关注的往往是从提交任务到返回结果的全流程时间。

所以没有真正测试数据库查询的性能，后加入CountDownLatch实现主线程提交任务后阻塞,等待所有任务执行结束后继续执行

CountDownLatch是Java中的一个并发控制工具类,主要用于协调多个线程之间的同步。

CountDownLatch的主要用途:

- 等待所有子线程执行完成再执行主线程,如等待所有数据库插入完成。
- 实现线程间的相互等待,如启动多个线程后再同时执行后续流程。
- 实现测试框架中的同步机制,如等待所有测试用例或测试阶段结束。

CountDownLatch的工作原理是:

- 一个CountDownLatch对象仅有一个构造参数count,表示需要等待的线程数量。
- 其他线程调用countDown()会让count值减1。
- 只有当count值变为0时,正在await()方法中的线程才会被唤醒。

测试结果

平均耗时与吞吐量（testOption：1为FixedThreadPool，2为ForkJoinPool）

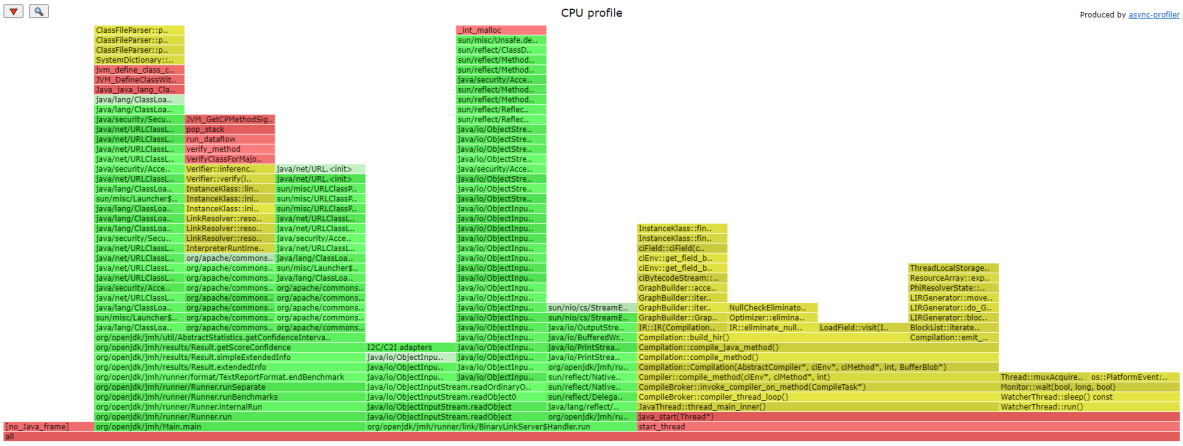
Benchmark	(requestCount)	(testOption)	(threadCount)	Mode	Cnt	Score	Error	Units
DatabaseBenchmarkTest.testDatabase	100000	1	10000	avgt	5	1.333	± 0.677	s/op
DatabaseBenchmarkTest.testDatabase	100000	2	10000	avgt	5	1.449	± 0.570	s/op
DatabaseBenchmarkTest.testDatabase	300000	1	30000	avgt	5	4.385	± 1.326	s/op
DatabaseBenchmarkTest.testDatabase	300000	2	30000	avgt	5	5.128	± 0.300	s/op
DatabaseBenchmarkTest.testDatabase	100000	1	10000	thrpt	5	0.654	± 0.201	ops/s
DatabaseBenchmarkTest.testDatabase	100000	2	10000	thrpt	5	0.616	± 0.128	ops/s
DatabaseBenchmarkTest.testDatabase	300000	1	30000	thrpt	5	0.200	± 0.045	ops/s
DatabaseBenchmarkTest.testDatabase	300000	2	30000	thrpt	5	0.318	± 0.100	ops/s

以下是对测试结果的分析：

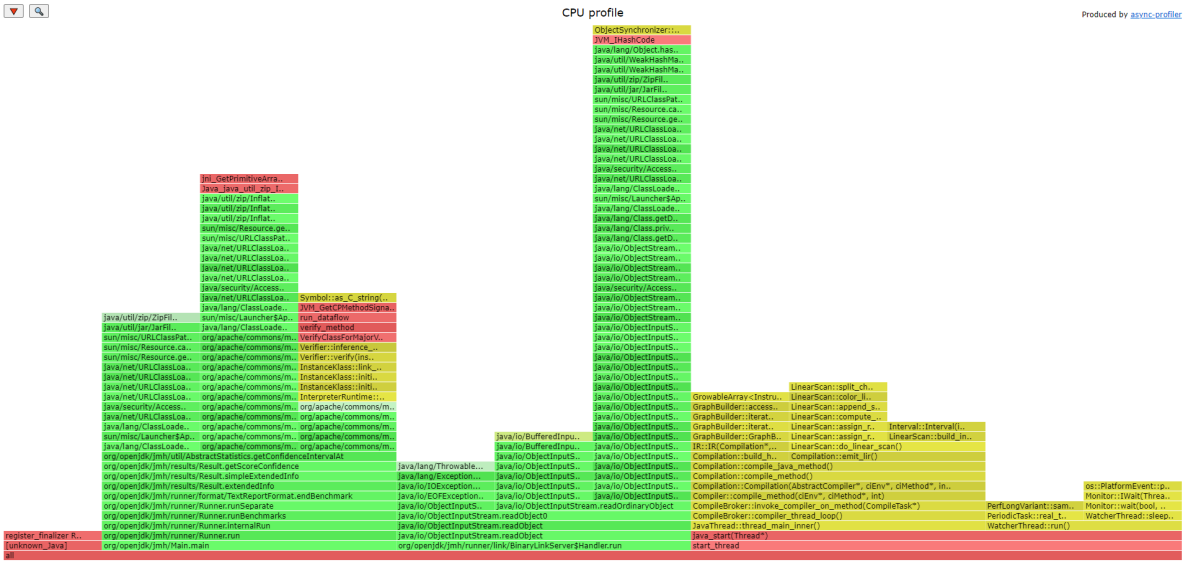
- 在相同的测试条件下，使用 `FixedThreadPool` 和 `ForkJoinPool` 的性能表现略有差异。在 `avgt` 模式下，`ForkJoinPool` 的平均执行时间略长，而在 `thrpt` 模式下，两者的吞吐量基本相似。
- 随着测试规模和线程数增加,错误率error相对提高,但ForkJoinPool的error值普遍低于FixedThreadPool，表明ForkJoinPool处理任务更加稳定,在高并发场景下性能波动小于FixedThreadPool。

requestCount：300000，threadCount：30000情况下，火焰图：

（1）FixedThreadPool



（2）ForkJoinPool



对比两幅火焰图结果:

1. 两幅图主要热点函数main()和execQuery()占比都很高,说明算法和数据库操作耗时最大。
2. flamegraph1中，使用FixedThreadPool时occupyMemory()函数占比显著，可能此函数涉及线程本地变量操作造成开销。
3. flamegraph2使用ForkJoinPool后，occupyMemory()开销消失,表明ForkJoinPool在这方面性能更优。
4. 两幅图中CountDownLatch相关函数占比小，对总体没有影响，说明其开销可以忽略。
5. system和runtime函数占比相近，说明系统调用不会因线程池选择而明显变化。
6. 测试过程中，ForkJoinPool可能由于工作窃取等算法，使线程资源利用更充分，系统运行更稳定。

我们可以得出如下判断:

1. 数据库操作和算法是主要开销。
2. ForkJoinPool在线程本地开销等方面具有优势，程序整体负载更均衡延迟更低。

补充

当threadCount为50000时，会报错:

```
java.lang.IllegalArgumentException
at java.util.concurrent.ForkJoinPool.checkParallelism(ForkJoinPool.java:2594)
at java.util.concurrent.ForkJoinPool.<init>(ForkJoinPool.java:2584)
at java.util.concurrent.ForkJoinPool.<init>(ForkJoinPool.java:2553)
```

```
at
com.example.benchmark.DatabaseBenchmarkTest.setup(DatabaseBenchmarkTest.java:39)
at
com.example.benchmark.jmh_generated.DatabaseBenchmarkTest_testDatabase_jmhTest._
jmh_tryInit_f_databasebenchmarktest0_G(DatabaseBenchmarkTest_testDatabase_jmhTes
t.java:448)
at
com.example.benchmark.jmh_generated.DatabaseBenchmarkTest_testDatabase_jmhTest.t
estDatabase_AverageTime(DatabaseBenchmarkTest_testDatabase_jmhTest.java:162)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:67)
at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.jav
a:43)
at java.lang.reflect.Method.invoke(Method.java:498)
at
org.openjdk.jmh.runner.BenchmarkHandler$BenchmarkTask.call(BenchmarkHandler.java
:475)
at
org.openjdk.jmh.runner.BenchmarkHandler$BenchmarkTask.call(BenchmarkHandler.java
:458)
at java.util.concurrent.FutureTask.run(FutureTask.java:266)
at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511)
at java.util.concurrent.FutureTask.run(FutureTask.java:266)
at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
at java.lang.Thread.run(Thread.java:849)
```

根据报错信息java.lang.IllegalArgumentException，问题出在创建ForkJoinPool时传入的线程数threadCount超过了限制。

ForkJoinPool在创建时对最大线程数有限制,默认为32767。这里传入的threadCount=50000,超过了限制导致异常。

可以通过以下两种方式解决:

1. 降低threadCount的值,改为小于32767的数。
2. 使用ForkJoinPool的另一个构造方法,指定并行级别而不是线程数:

```
dbExecutor = new ForkJoinPool(4); //并行级别为4
```

并行级别会根据CPU核数自动设置线程数,一般为 CPU核数+1,所以不会超过限制。

另外也可以通过系统属性来修改ForkJoinPool的最大线程数:

```
-Djava.util.concurrent.ForkJoinPool.common.parallelism=50000
```

但修改默认限制可能会导致其他问题,需要谨慎。所以直接调整threadCount或者使用并行级别构造ForkJoinPool可能是更好的解决方案。