性能分析

测试环境

服务器：

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

当比较FixedThreadPool和ForkJoinPool作为协程调度器时，以下是可以考虑的数据和分析方面：

1. 性能比较：可以设计一组具有相似特征的并发任务，并分别使用FixedThreadPool和ForkJoinPool作为协程调度器来执行这些任务。通过测量执行时间、吞吐量或其他性能指标，来比较两种调度器的性能优劣。
2. 资源利用：观察和比较FixedThreadPool和ForkJoinPool作为协程调度器时的资源利用情况。可以监控线程的使用情况、内存消耗等指标，并分析两种调度器在不同负载下的资源利用效率。
3. 可伸缩性：通过逐渐增加并发任务的数量，测试FixedThreadPool和ForkJoinPool作为协程调度器的可伸缩性。记录并分析两种调度器在不同任务负载下的表现，包括任务执行时间的增长曲线、系统资源的利用情况等。
4. 并行性能：如果任务可以进行并行拆分和执行，可以使用具有并行特性的任务来比较FixedThreadPool和ForkJoinPool作为协程调度器的性能。可以测量任务拆分和合并的开销、并行执行的效率等指标，并进行对比分析。

```
package com.example.benchmark;

import org.openjdk.jmh.annotations.*;

import java.util.concurrent.*;

@Fork(1)
@Warmup(iterations = 3, time = 5)
@Measurement(iterations = 5, time = 5)
@State(Scope.Benchmark)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
public class DatabaseBenchmarkTest {
    @Param({"1","2"})
    private int testOption;

    @Param({"100","1000"})
    private int threadCount;

    @Param({"1000","10000"})
    private int taskCount;

    private ExecutorService dbExecutor;

    @Setup(Level.Trial)
```

```
    public void setup() {
        if (testOption == 1) {
            dbExecutor = Executors.newFixedThreadPool(threadCount);
        } else if (testOption == 2) {
            dbExecutor = new ForkJoinPool(threadCount);
        } else {
            throw new IllegalArgumentException("Invalid test option: " +
testOption);
        }
    }

    @TearDown(Level.Trial)
    public void teardown() {
        dbExecutor.shutdown();
        try {
            if (!dbExecutor.awaitTermination(1, TimeUnit.MINUTES)) {
                dbExecutor.shutdownNow();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
            dbExecutor.shutdownNow();
        }
    }

    @Benchmark
    public void testHeavyCpuTask() throws InterruptedException {
        CountDownLatch latch = new CountDownLatch(taskCount);
        for(int i = 0; i < taskCount; i++) {
            CompletableFuture<Void> cf = CompletableFuture.runAsync(() -> {
                try {
                    Thread.ofVirtual().scheduler(dbExecutor).start(() -> {
                        longRunningTask();
                        latch.countDown();
                    });
                } catch (Exception e) {
                    e.printStackTrace();
                }
            });
        }
        //等待所有任务完成
        latch.await();
    }

    public void heavyCpuTask() {
        // 执行计算密集型任务
        for (int i = 0; i < 1000000; i++) {
            Math.sqrt(i);
        }
    }

    public void longRunningTask() {
        // 模拟长时间运行的任务，例如睡眠5秒
```

```
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

以上代码是一个基准测试类，通过JMH框架进行性能测试。它包括了不同的基准测试选项、线程数量和任务数量，并使用线程池执行任务。其中包含了CPU密集型任务和模拟长时间运行的任务。通过运行基准测试可以评估和比较不同配置下的性能表现。

Benchmark	(taskCount)	(testOption)	(threadCount)	Mode	Cnt	Score	Error	Units
DatabaseBenchmarkTest.testHeavyCpuTask	10000	1	1000	avgt	5	26.777	±5.502	ms/op
DatabaseBenchmarkTest.testHeavyCpuTask	10000	2	1000	avgt	5	82.368	±45.204	ms/op
DatabaseBenchmarkTest.testHeavyCpuTask	50000	1	5000	avgt	5	147.657	±13.394	ms/op
DatabaseBenchmarkTest.testHeavyCpuTask	50000	2	5000	avgt	5	385.103	±126.250	ms/op
DatabaseBenchmarkTest.testHeavyCpuTask	100000	1	10000	avgt	5	275.195	±35.469	ms/op
DatabaseBenchmarkTest.testHeavyCpuTask	100000	2	10000	avgt	5	1498.102	±1030.298	ms/op

根据基准测试结果，可以看出使用不同的调度器（`testOption`参数为1和2）对于执行计算密集型任务的性能产生了显著的影响。两个调度器之间的区别和原因：

1. `testOption = 1`：使用`Executors.newFixedThreadPool`创建固定线程数的线程池作为调度器。

   结果表明，在这种情况下，随着任务数量（`taskCount`）的增加，性能呈现出相对稳定的状态。具体来说，随着任务数量的增加，执行时间（Score）也随之增加，但是增长幅度相对较小。这是因为固定线程池的线程数已经被限定为特定的数量（`threadCount`），线程池中的线程在处理任务时会被重复利用。因此，无论任务数量如何增加，线程池中的线程数保持不变，而调度和切换线程的开销相对较小。

2. `testOption = 2`：使用`ForkJoinPool`作为调度器。

   结果表明，使用`ForkJoinPool`作为调度器时，性能与任务数量和线程数之间存在较大的关联性。随着任务数量的增加，执行时间呈现出显著的增长。这是因为`ForkJoinPool`是基于工作窃取（work-stealing）算法的线程池，它将任务分割成更小的子任务并将其分配给线程池中的工作线程。但是，当任务数量增加时，线程池中的线程可能会被频繁地分割和创建，以及进行任务的调度和切换，这会增加额外的开销，导致性能下降。

综上所述，`Executors.newFixedThreadPool`相对于`ForkJoinPool`在处理计算密集型任务时的性能更好。这是因为`newFixedThreadPool`创建的线程池可以更好地控制线程数量，并且在处理任务时减少了线程调度和切换的开销。而`ForkJoinPool`适用于处理递归任务和I/O密集型任务等具有较高并行性的场景。

获取火焰图有两种命令：

```
（1）
java -agentpath:/data/lqy_dataset/async-profiler-2.9-linux-
x64/build/libasyncProfiler.so=start,event=cpu,file=profile1.html -jar
DatabaseBenchmarkTest.jar

（2）
sudo ./profiler.sh -d 60 -f flamegraph.html 9541
```

1. 第一个命令是在Java应用程序启动时使用Java代理（Java Agent）方式加载异步分析器（async-profiler）并进行性能分析。它的具体内容为：

   ```
   java -agentpath:/data/lqy_dataset/async-profiler-2.9-linux-
   x64/build/libasyncProfiler.so=start,event=cpu,file=profile1.html -jar
   DatabaseBenchmarkTest.jar
   ```

   - `-agentpath:/data/lqy_dataset/async-profiler-2.9-linux-x64/build/libasyncProfiler.so`：指定了异步分析器的路径和文件名。
   - `start,event=cpu,file=profile1.html`：启动异步分析器，并指定要分析的事件为CPU事件，将分析结果输出到profile1.html文件中。
   - `-jar DatabaseBenchmarkTest.jar`：启动了一个名为DatabaseBenchmarkTest.jar的Java应用程序。

   这个命令会在Java应用程序运行期间收集CPU事件的性能分析数据，并将结果保存到profile1.html文件中。

2. 第二个命令是直接运行了一个名为profiler.sh的脚本，并传递了一些参数进行性能分析。它的具体内容为：

   ```
   sudo ./profiler.sh -d 60 -f flamegraph.html 9541
   ```

   - `sudo ./profiler.sh`：运行了一个名为profiler.sh的脚本。
   - `-d 60`：指定了持续时间为60秒，即在60秒内进行性能分析。
   - `-f flamegraph.html`：指定了生成的性能分析结果文件名为flamegraph.html。
   - `9541`：指定了要进行性能分析的进程ID。

   这个命令会在指定的时间内对指定进程进行性能分析，并生成一个名为flamegraph.html的性能分析结果文件。

总结：

- 第一个命令是在Java应用程序启动时加载异步分析器进行性能分析，将结果保存到指定的文件中。
- 第二个命令是直接运行一个脚本对指定进程进行性能分析，生成一个性能分析结果文件。

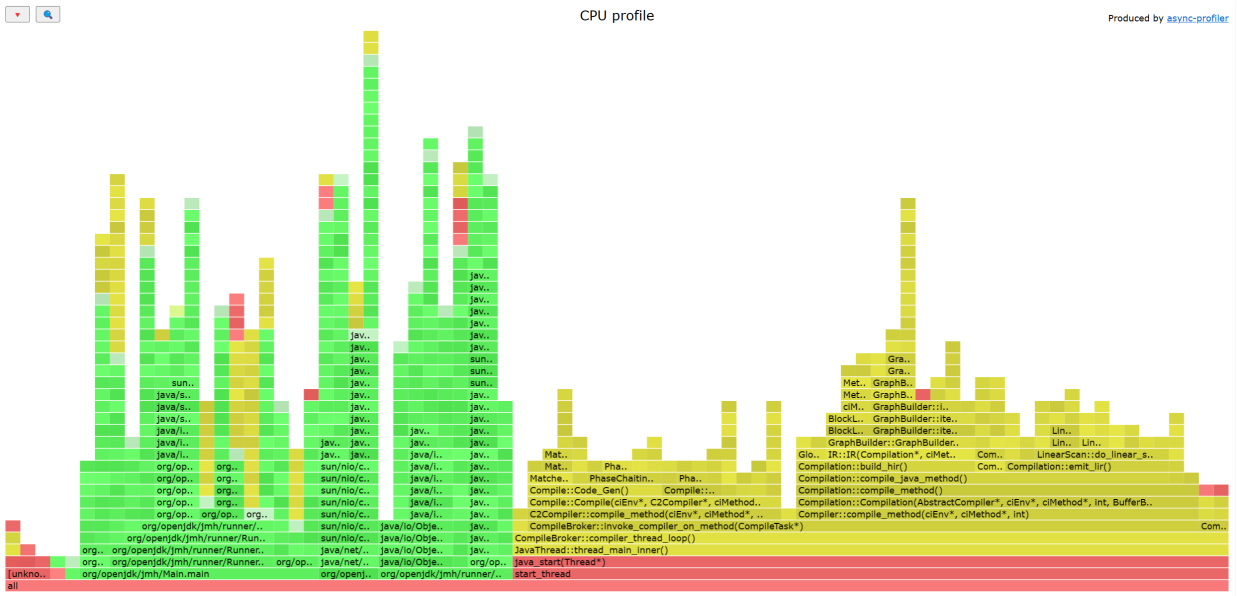
通过第一个命令，可以在应用程序运行期间进行实时的、针对CPU事件的性能分析，并将结果保存到文件中，方便后续分析和优化。这种方式对于需要长时间运行的应用程序或对实时性能监控要求较高的场景非常有用。第一个命令使用Java代理方式加载异步分析器进行性能分析具有以下优点：

1. 实时性能分析：通过在Java应用程序启动时加载异步分析器，可以实时地对应用程序进行性能分析，无需停止或重启应用程序。
2. 精确的事件选择：命令中的`event=cpu`指定了要进行CPU事件的性能分析，可以针对CPU消耗进行详细分析，帮助发现CPU瓶颈和性能热点。
3. 灵活的输出方式：通过指定`file=profile1.html`，可以将性能分析结果输出到指定的HTML文件中，便于后续查看和分析。
4. 高度可定制性：异步分析器本身提供了丰富的配置选项，可以根据具体需求进行定制，如选择不同的事件、调整采样率等。

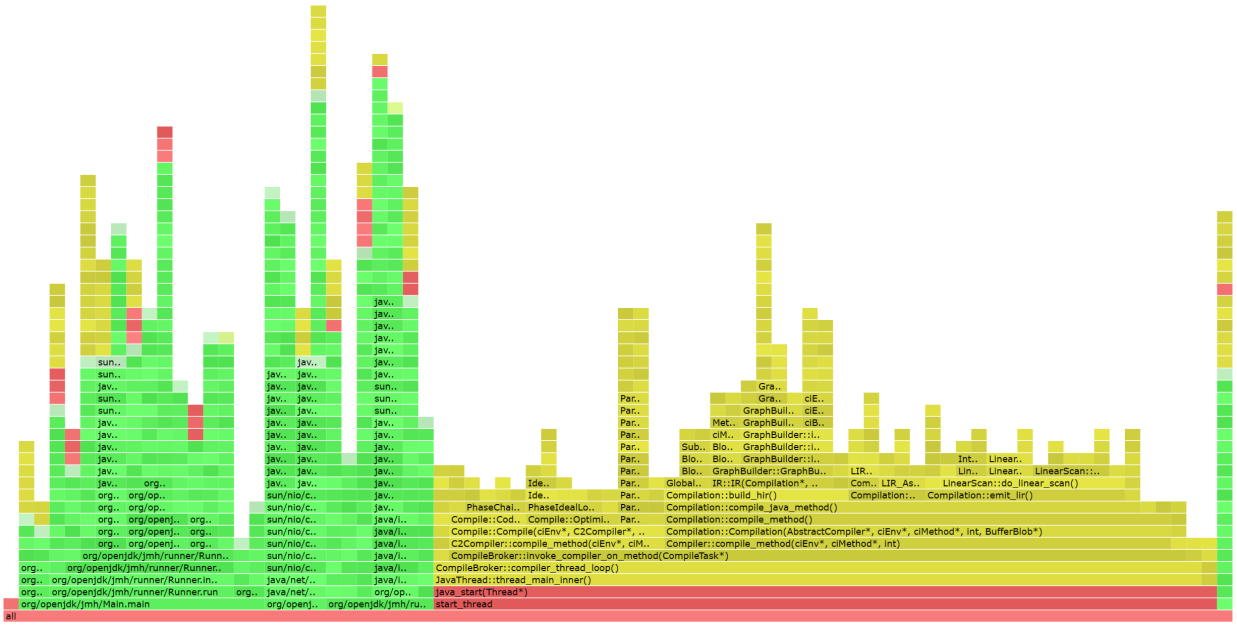
---

taskCount：10000，threadCount：1000情况下，火焰图：

（1）FixedThreadPool



（2）ForkJoinPool



Benchmark	(taskCount)	(testOption)	(threadCount)	Mode	Cnt	Score	Error	Units
DatabaseBenchmarkTest.testLongRunningTask	1000	1	100	avgt	5	5015.453	±30.182	ms/op
DatabaseBenchmarkTest.testLongRunningTask	1000	1	1000	avgt	5	5020.847	±15.288	ms/op
DatabaseBenchmarkTest.testLongRunningTask	1000	2	100	avgt	5	5006.388	4.946	ms/op
DatabaseBenchmarkTest.testLongRunningTask	1000	2	1000	avgt	5	5028.961	150.740	ms/op
DatabaseBenchmarkTest.testLongRunningTask	10000	1	100	avgt	5	5232.626	286.466	ms/op
DatabaseBenchmarkTest.testLongRunningTask	10000	1	1000	avgt	5	5040.863	11.629	ms/op
DatabaseBenchmarkTest.testLongRunningTask	10000	2	100	avgt	5	5191.912	15.243	ms/op
DatabaseBenchmarkTest.testLongRunningTask	10000	2	1000	avgt	5	5067.596	108.506	ms/op

在代码中，我们可以观察到两个不同的调度器用于执行任务：
`Executors.newFixedThreadPool(threadCount)`和`new ForkJoinPool(threadCount)`。

1. `Executors.newFixedThreadPool(threadCount)`：
   这是一个基于线程池的调度器，使用了`Executors`工具类提供的`newFixedThreadPool()`方法创建一个固定大小的线程池。线程池中的线程数量由`threadCount`参数指定。每个任务都会被提交到线程池中的一个空闲线程执行。这种调度器适用于短期的、非阻塞的任务执行。
2. `new ForkJoinPool(threadCount)`：
   这是一个基于Fork-Join框架的调度器，使用了`ForkJoinPool`类创建一个Fork-Join线程池。Fork-Join框架是Java提供的一种并行任务执行模型，适用于处理可拆分的、递归的任务。在这种调度器中，任务会被分解成更小的子任务，并且这些子任务会被递归地分配给线程池中的工作线程执行。线程池中的线程数量由`threadCount`参数指定。

两个调度器之间的区别主要在于任务的调度和执行方式：

- 线程池调度器（`Executors.newFixedThreadPool()`）适用于短期、非阻塞的任务。它通过维护一个线程池来管理任务的执行。每个任务都会被提交到线程池中的一个空闲线程执行。线程池中的线程数是固定的，如果所有线程都在执行任务，新的任务将会等待直到有线程可用。这种调度器适用于大量的、相对较小的任务，可以提高任务执行的并发性。
- Fork-Join调度器（`new ForkJoinPool()`）适用于可拆分、递归的任务。它使用Fork-Join框架来实现任务的并行执行。任务会被递归地分解成更小的子任务，并且这些子任务会被分配给线程池中的工作线程执行。线程池中的线程数量也是固定的，每个线程在处理一个任务时可能会进一步分解成更小的子任务。这种调度器适用于需要递归拆分任务并以并行方式执行的情况，可以提高任务的并行度和利用CPU资源的效率。

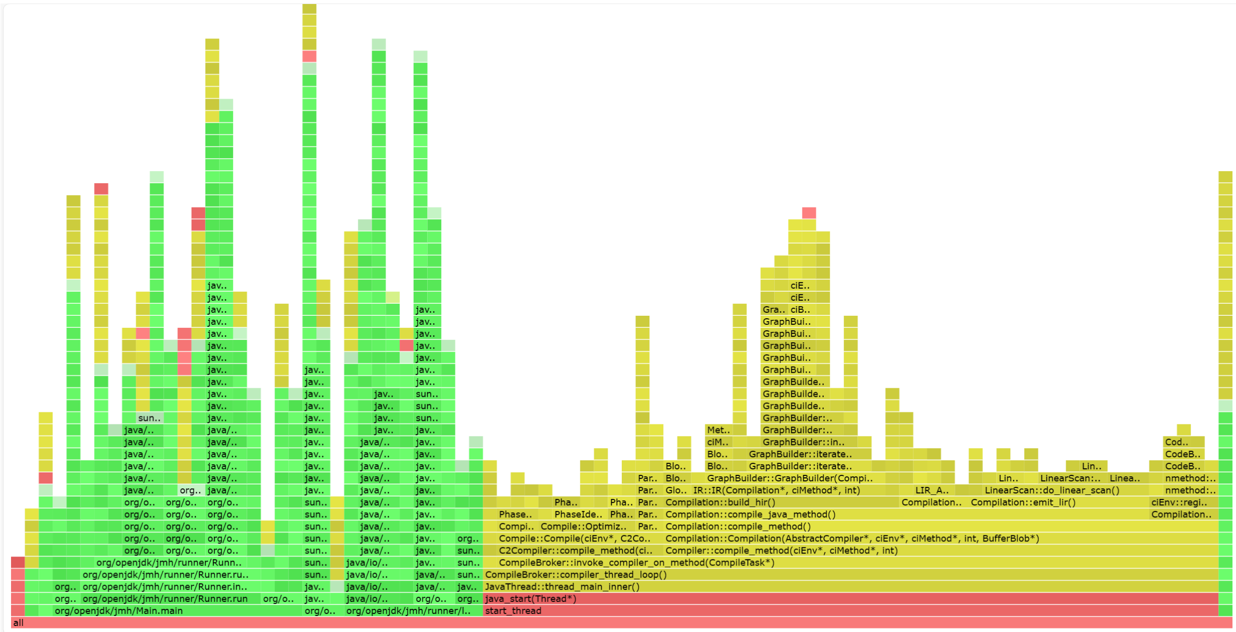
选择使用哪种调度器取决于任务的特性和需求。如果任务是短期的、非阻塞的，并且任务量较大，则线程池调度器可能更适合。如果任务是可拆分的、递归的，并且需要充分利用CPU资源进行并行执行，则Fork-Join调度器可能更适合。

需要注意的是，无论使用哪种调度器，都需要根据具体的需求和性能要求来选择合适的线程池大小和配置。过小的线程池可能导致任务等待执行，而过大的线程池可能会增加线程上下文切换的开销。因此，对于不同的场景，需要进行实际的性能测试和调优来确定最优的线程池配置。

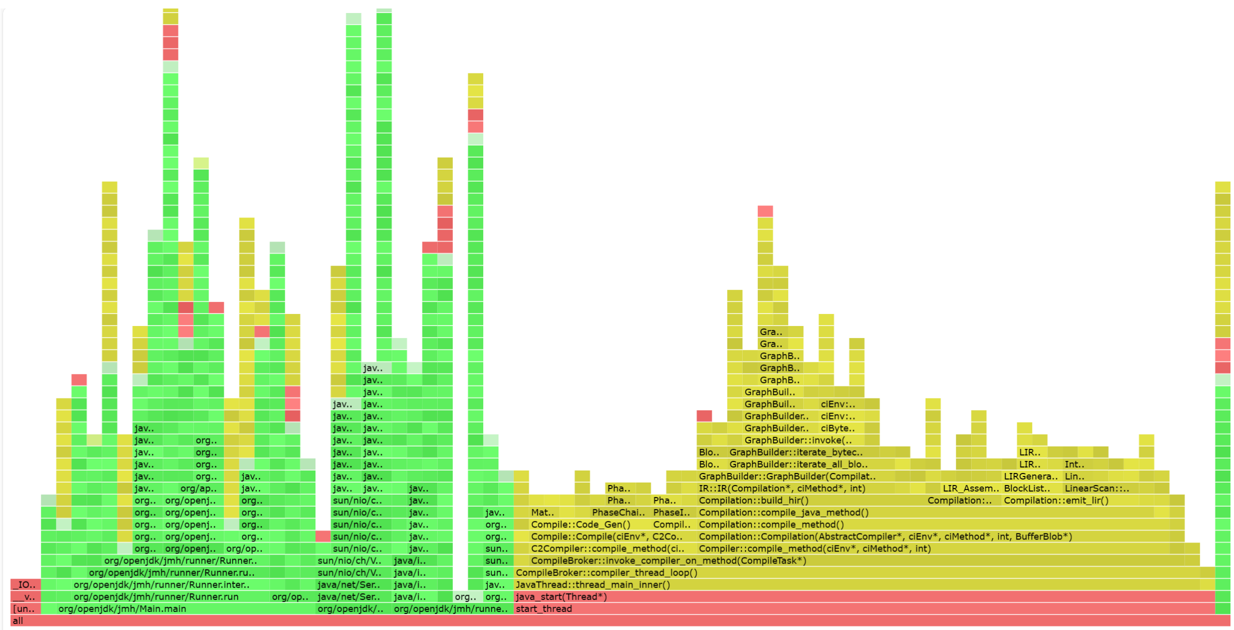
---

taskCount：1000，threadCount：100情况下，火焰图：

（1）FixedThreadPool



（2）ForkJoinPool



FixedThreadPool和ForkJoinPool作为协程调度器时的对比分析总结：

FixedThreadPool作为协程调度器的情况：

1. 并发性：FixedThreadPool作为协程调度器可以提供并发执行协程的能力，类似于多线程的并发执行模型。
2. 线程数量固定：FixedThreadPool作为协程调度器时，采用固定数量线程模型，每个线程独立执行任务，线程上下文切换开销小。
3. 资源利用：由于线程数量固定，无法根据协程的数量进行动态调整，可能存在线程资源浪费或协程等待的情况，特别是当协程数量超过线程数量时。
4. 适用性：FixedThreadPool作为协程调度器适用于需要并发执行的协程任务，但适用性相对较窄，不支持协程的轻量级和非阻塞特性。

ForkJoinPool作为协程调度器的情况：

1. 任务拆分与并行执行：ForkJoinPool作为协程调度器可以将大任务拆分为小任务并行执行，类似于协程的任务拆分和并发执行模型。
2. 动态负载平衡：ForkJoinPool使用工作窃取算法，允许线程间交换和执行任务，但上下文切换开销较大。在任务执行过程中可以动态调整线程的负载，提高并行性能。
3. 适应性：ForkJoinPool作为协程调度器适用于递归任务和可拆分的任务，能够以更细粒度的方式进行任务调度和并行计算。
4. 资源利用：由于工作窃取算法的负载平衡机制，ForkJoinPool可以更高效地利用线程资源，避免资源浪费和任务等待的情况。

总结：

- FixedThreadPool作为协程调度器提供并发执行协程的能力，但无法动态调整线程数量，可能存在资源利用问题。
- ForkJoinPool作为协程调度器具有任务拆分、并行执行和动态负载平衡的能力，适用于递归和可拆分的任务，并能更高效地利用线程资源。
- 总体性能

  CPU密集任务:FixedThreadPool更适用于CPU密集任务，利用多核高效执行，性能优良。

  IO密集任务:ForkJoinPool工作窃取能力处理横向切分的任务、IO密集任务更有效率，更好利用率。

在选择使用FixedThreadPool还是ForkJoinPool作为协程调度器时，需要考虑任务的特性、并发需求和资源利用要求。同时，还应注意协程调度器的实验性质，并在实际使用中进行测试和评估。