主讲老师: Fox

> 有道笔记地址： https://note.youdao.com/s/USwdjLcz
> 并发知识体系：https://www.processon.com/view/link/615d4a610e3e74663e97fa0e

1. 由一道算法题引发的思考

算法题：如何充分利用多核CPU的性能，快速对一个2千万大小的数组进行排序?

这道算法题可以拆解来看：
1）首先这是一道排序的算法题，而且是需要使用高效的排序算法对2千万大小的数组进行排序，可以考虑使用快速排序或者归并排序。
2）可以使用多线程并行排序算法来充分利用多核CPU的性能。

关于排序算法，还不太清楚同学的可以学习下面的课程

> **大厂高频笔试题Top20、精选LeetCode热题100详解**
> https://vip.tulingxueyuan.cn/p/t_pc/course_pc_detail/big_column/p_61a344ffe4b09240f0e4b59e

这套课程中讲解了十大排序算法：

2. 基于归并排序算法实现

快速对一个大小为2千万的数组进行排序，可以使用高效的归并排序算法来实现。

2.1 什么是归并排序

归并排序（Merge Sort）是一种基于分治思想的排序算法。归并排序的基本思想是将一个大数组分成两个相等大小的子数组，对每个子数组分别进行排序，然后将两个子数组合并成一个有序的大数组。因为常常使用递归实现(由先拆分后合并的性质决定的)，所以我们称其为归并排序。

归并排序的步骤包括以下几个方面：

- 将数组分成两个子数组
- 对每个子数组进行排序
- 合并两个有序的子数组

归并排序的时间复杂度为O(nlogn)，空间复杂度为O(n)，其中n为数组的长度。

> 分治思想是将一个规模为N的问题分解为K个规模较小的子问题，这些子问题相互独立且与原问题性质相同。求出子问题的解，就可得到原问题的解。
>
> 分治思想的步骤如下：
>
> 1. 分解：将要解决的问题划分成若干规模较小的同类问题；
> 2. 求解：当子问题划分得足够小时，用较简单的方法解决；
> 3. 合并：按原问题的要求，将子问题的解逐层合并构成原问题的解。
>
> 计算机十大经典算法中的归并排序、快速排序、二分查找都是基于分治思想实现的算法
>
> 分治任务模型图如下：

2.2 归并排序动图演示

> 归并排序演示：https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

2.3 使用归并排序实现上面的算法题

单线程实现归并排序

单线程归并算法的实现，它的基本思路是将序列分成两个部分，分别进行递归排序，然后将排序好的子序列合并起来。

```java
public class MergeSort {

    private final int[] arrayToSort; //要排序的数组
    private final int threshold;  //拆分的阈值，低于此阈值就不再进行拆分

    public MergeSort(final int[] arrayToSort, final int threshold) {
        this.arrayToSort = arrayToSort;
        this.threshold = threshold;
    }

    /**
     * 排序
     * @return
     */
    public int[] sequentialSort() {
        return sequentialSort(arrayToSort, threshold);
    }

    public static int[] sequentialSort(final int[] arrayToSort, int threshold) {
        //拆分后的数组长度小于阈值，直接进行排序
        if (arrayToSort.length < threshold) {
            //调用jdk提供的排序方法
            Arrays.sort(arrayToSort);
            return arrayToSort;
        }

        int midpoint = arrayToSort.length / 2;
        //对数组进行拆分
        int[] leftArray = Arrays.copyOfRange(arrayToSort, 0, midpoint);
        int[] rightArray = Arrays.copyOfRange(arrayToSort, midpoint,
arrayToSort.length);
        //递归调用
        leftArray = sequentialSort(leftArray, threshold);
        rightArray = sequentialSort(rightArray, threshold);
        //合并排序结果
        return merge(leftArray, rightArray);
    }

```

```
    public static int[] merge(final int[] leftArray, final int[] rightArray) {
        //定义用于合并结果的数组
        int[] mergedArray = new int[leftArray.length + rightArray.length];
        int mergedArrayPos = 0;
        int leftArrayPos = 0;
        int rightArrayPos = 0;
        while (leftArrayPos < leftArray.length && rightArrayPos < rightArray.length) {
            if (leftArray[leftArrayPos] <= rightArray[rightArrayPos]) {
                mergedArray[mergedArrayPos] = leftArray[leftArrayPos];
                leftArrayPos++;
            } else {
                mergedArray[mergedArrayPos] = rightArray[rightArrayPos];
                rightArrayPos++;
            }
            mergedArrayPos++;
        }

        while (leftArrayPos < leftArray.length) {
            mergedArray[mergedArrayPos] = leftArray[leftArrayPos];
            leftArrayPos++;
            mergedArrayPos++;
        }

        while (rightArrayPos < rightArray.length) {
            mergedArray[mergedArrayPos] = rightArray[rightArrayPos];
            rightArrayPos++;
            mergedArrayPos++;
        }

        return mergedArray;
    }
}
```

Fork/Join并行归并排序

并行归并排序是一种利用多线程实现的归并排序算法。它的基本思路是将数据分成若干部分，然后在不同线程上对这些部分进行归并排序，最后将排好序的部分合并成有序数组。在多核CPU上，这种算法也能够有效提高排序速度。

可以使用Java的Fork/Join框架来实现归并排序的并行化

```java
public class MergeSortTask extends RecursiveAction {

    private final int threshold; //拆分的阈值，低于此阈值就不再进行拆分
    private int[] arrayToSort; //要排序的数组

    public MergeSortTask(final int[] arrayToSort, final int threshold) {
        this.arrayToSort = arrayToSort;
        this.threshold = threshold;
    }

    @Override
    protected void compute() {
        //拆分后的数组长度小于阈值，直接进行排序
        if (arrayToSort.length <= threshold) {
            // 调用jdk提供的排序方法
            Arrays.sort(arrayToSort);
            return;
        }

        // 对数组进行拆分
        int midpoint = arrayToSort.length / 2;
        int[] leftArray = Arrays.copyOfRange(arrayToSort, 0, midpoint);
        int[] rightArray = Arrays.copyOfRange(arrayToSort, midpoint, arrayToSort.length);

        MergeSortTask leftTask = new MergeSortTask(leftArray, threshold);
        MergeSortTask rightTask = new MergeSortTask(rightArray, threshold);

        //提交任务
        leftTask.fork();
        rightTask.fork();
        //阻塞当前线程，直到获取任务的执行结果
        leftTask.join();
        rightTask.join();

        // 合并排序结果
        arrayToSort = MergeSort.merge(leftTask.getSortedArray(),
rightTask.getSortedArray());
    }
```

```
38
39      public int[] getSortedArray() {
40          return arrayToSort;
41      }
42  }
```

在这个示例中，我们使用Fork/Join框架实现了归并排序算法，并通过递归调用实现了并行化。使用Fork/Join框架实现归并排序算法的关键在于将排序任务分解成小的任务，使用Fork/Join框架将这些小任务提交给线程池中的不同线程并行执行，并在最后将排序后的结果进行合并。这样可以充分利用多核CPU的并行处理能力，提高程序的执行效率。

测试结果对比

测试代码

```java
public class Utils {

    /**
     * 随机生成数组
     * @param size 数组的大小
     * @return
     */
    public static int[] buildRandomIntArray(final int size) {
        int[] arrayToCalculateSumOf = new int[size];
        Random generator = new Random();
        for (int i = 0; i < arrayToCalculateSumOf.length; i++) {
            arrayToCalculateSumOf[i] = generator.nextInt(100000000);
        }
        return arrayToCalculateSumOf;
    }
}
public class ArrayToSortMain {

     public static void main(String[] args) {
         //生成测试数组  用于归并排序
         int[] arrayToSortByMergeSort = Utils.buildRandomIntArray(20000000);
         //生成测试数组  用于forkjoin排序
         int[] arrayToSortByForkJoin = Arrays.copyOf(arrayToSortByMergeSort, arrayToSortByMergeSort.length);
         //获取处理器数量
         int processors = Runtime.getRuntime().availableProcessors();


         MergeSort mergeSort = new MergeSort(arrayToSortByMergeSort, processors);
         long startTime = System.nanoTime();
         // 归并排序
         mergeSort.mergeSort();
         long duration = System.nanoTime()-startTime;
         System.out.println("单线程归并排序时间: "+(duration/(1000f*1000f))+"毫秒");


         //利用forkjoin排序
```

```
37          MergeSortTask mergeSortTask = new MergeSortTask(arrayToSortByForkJoin,
    processors);
38          //构建forkjoin线程池
39          ForkJoinPool forkJoinPool = new ForkJoinPool(processors);
40          startTime = System.nanoTime();
41          //执行排序任务
42          forkJoinPool.invoke(mergeSortTask);
43          duration = System.nanoTime()-startTime;
44          System.out.println("forkjoin排序时间: "+(duration/(1000f*1000f))+"毫秒");
45
46      }
47  }
48
49
```

根据测试结果可以看出，数组越大，利用Fork/Join框架实现的并行化归并排序比单线程归并排序的效率更高：

2.4 并行实现归并排序的优化和注意事项

在实际应用中，我们需要考虑数据分布的均匀性、内存使用情况、线程切换开销等因素，以充分利用多核CPU并保证算法的正确性和效率。

以下是并行实现归并排序的一些优化和注意事项：

- 任务的大小：任务大小的选择会影响并行算法的效率和负载均衡，如果任务太小，会造成任务划分和合并的开销过大；如果任务太大，会导致任务无法充分利用多核CPU并行处理能力。因此，在实际应用中需要根据数据量、CPU核心数等因素选择合适的任务大小。
- 负载均衡：并行算法需要保证负载均衡，即各个线程执行的任务大小和时间应该尽可能相等，否则会导致某些线程负载过重，而其他线程负载过轻的情况。在归并排序中，可以通过递归调用实现负载均衡，但是需要注意递归的层数不能太深，否则会导致任务划分和合并的开销过大。
- 数据分布：数据分布的均匀性也会影响并行算法的效率和负载均衡。在归并排序中，如果数据分布不均匀，会导致某些线程处理的数据量过大，而其他线程处理的数据量过小的情况。因此，在实际应用中需要考虑数据的分布情况，尽可能将数据分成大小相等的子数组。
- 内存使用：并行算法需要考虑内存的使用情况，特别是在处理大规模数据时，内存的使用情况会对算法的执行效率产生重要影响。在归并排序中，可以通过对数据进行原地归并实现内存的节约，但是需要注意归并的实现方式，以避免数据的覆盖和不稳定排序等问题。

- 线程切换：线程切换是并行算法的一个重要开销，需要尽量减少线程的切换次数，以提高算法的执行效率。在归并排序中，可以通过设置线程池的大小和调整任务大小等方式控制线程的数量和切换开销，以实现算法的最优性能。

3. Fork/Join框架介绍

3.1 什么是Fork/Join

Fork/Join是一个是一个并行计算的框架，主要就是用来支持分治任务模型的，这个计算框架里的 Fork 对应的是分治任务模型里的任务分解，Join 对应的是结果合并。它的核心思想是将一个大任务分成许多小任务，然后并行执行这些小任务，最终将它们的结果合并成一个大的结果。它适用于可以采用分治策略的计算密集型任务，例如大规模数组的排序、图形的渲染、复杂算法的求解等。

3.2 应用场景

1. **并行计算**:

ForkJoinPool 提供了一种方便的方式来执行大规模的计算任务，并充分利用多核处理器的性能优势。通过将大任务分解成小任务，并通过工作窃取算法实现任务的并行执行，可以提高计算效率。

2. **递归任务处理**:

ForkJoinPool 特别适用于递归式的任务分解和执行。它可以将一个大任务递归地分解成许多小任务，并通过工作窃取算法动态地将这些小任务分配给工作线程执行。

3. **并行流操作**:

Java 8 引入了 Stream API，用于对集合进行函数式编程风格的操作。ForkJoinPool 通常用于执行并行流操作中的并行计算部分，例如对流中的元素进行过滤、映射、聚合等操作。

4. **高性能任务执行**:

ForkJoinPool 提供了一种高性能的任务执行机制，通过对任务进行动态调度和线程池管理，可以有效地利用系统资源，并在多核处理器上实现任务的并行执行。

总的来说，ForkJoinPool 类在 Java 中具有广泛的应用场景，特别适用于大规模的并行计算任务和递归式的任务处理。它通过工作窃取算法和任务分割合并机制，提供了一种高效的并行计算方式，可以显著提高计算效率和性能。

3.3 Fork/Join使用

Fork/Join框架的主要组成部分是ForkJoinPool、ForkJoinTask。ForkJoinPool是一个线程池，它用于管理ForkJoin任务的执行。ForkJoinTask是一个抽象类，用于表示可以被分割成更小部分的任务。

ForkJoinPool

ForkJoinPool是Fork/Join框架中的线程池类，它用于管理Fork/Join任务的线程。ForkJoinPool类包括一些重要的方法，例如submit()、invoke()、shutdown()、awaitTermination()等，用于提交任务、执行任务、关闭线程池和等待任务的执行结果。ForkJoinPool类中还包括一些参数，例如线程池的大小、工作线程的优先级、任务队列的容量等，可以根据具体的应用场景进行设置。

构造器

ForkJoinPool中有四个核心参数，用于控制线程池的并行数、工作线程的创建、异常处理和模式指定等。各参数解释如下：

- int parallelism：指定并行级别（parallelism level）。ForkJoinPool将根据这个设定，决定工作线程的数量。如果未设置的话，将使用Runtime.getRuntime().availableProcessors()来设置并行级别；
- ForkJoinWorkerThreadFactory factory：ForkJoinPool在创建线程时，会通过factory来创建。注意，这里需要实现的是ForkJoinWorkerThreadFactory，而不是ThreadFactory。如果你不指定factory，那么将由默认的DefaultForkJoinWorkerThreadFactory负责线程的创建工作；
- UncaughtExceptionHandler handler：指定异常处理器，当任务在运行中出错时，将由设定的处理器处理；
- boolean asyncMode：设置队列的工作模式。当asyncMode为true时，将使用先进先出队列，而为false时则使用后进先出的模式。

```
//获取处理器数量
int processors = Runtime.getRuntime().availableProcessors();
//构建forkjoin线程池
ForkJoinPool forkJoinPool = new ForkJoinPool(processors);
```

任务提交方式

任务提交是ForkJoinPool的核心能力之一，提交任务有三种方式：

	返回值	方法
提交异步执行	void	execute(ForkJoinTask<?> task) execute(Runnable task)
等待并获取结果	T	invoke(ForkJoinTask<T> task)
提交执行获取Future结果	ForkJoinTask<T>	submit(ForkJoinTask<T> task) submit(Callable<T> task) submit(Runnable task) submit(Runnable task, T result)

ForkJoinTask

ForkJoinTask是Fork/Join框架中的抽象类，它定义了执行任务的基本接口。用户可以通过继承ForkJoinTask类来实现自己的任务类，并重写其中的compute()方法来定义任务的执行逻辑。通常情况下我们不需要直接继承ForkJoinTask类，而只需要继承它的子类，Fork/Join框架提供了以下三个子类：

- **RecursiveAction**：用于递归执行但不需要返回结果的任务。
- **RecursiveTask**：用于递归执行需要返回结果的任务。
- CountedCompleter<T>：在任务完成执行后会触发执行一个自定义的钩子函数

调用方法

ForkJoinTask 最核心的是 fork() 方法和 join() 方法，承载着主要的任务协调作用，一个用于任务提交，一个用于结果获取。

- **fork()——提交任务**

  fork()方法用于向当前任务所运行的线程池中提交任务。如果当前线程是ForkJoinWorkerThread类型，将会放入该线程的工作队列，否则放入common线程池的工作队列中。

- **join()——获取任务执行结果**

  join()方法用于获取任务的执行结果。调用join()时，将阻塞当前线程直到对应的子任务完成运行并返回结果。

处理递归任务

计算斐波那契数列

斐波那契数列指的是这样一个数列：1，1，2，3，5，8，13，21，34，55，89... 这个数列从第3项开始，每一项都等于前两项之和。

```java
public class Fibonacci extends RecursiveTask<Integer> {
    final int n;

    Fibonacci(int n) {
        this.n = n;
    }

    /**
     * 重写RecursiveTask的compute()方法
     * @return
     */
    protected Integer compute() {
        if (n <= 1)
            return n;
        Fibonacci f1 = new Fibonacci(n - 1);
        //提交任务
        f1.fork();
        Fibonacci f2 = new Fibonacci(n - 2);
        //合并结果
        return f2.compute() + f1.join();
    }

    public static void main(String[] args) {
        //构建forkjoin线程池
        ForkJoinPool pool = new ForkJoinPool();
        Fibonacci task = new Fibonacci(10);
        //提交任务并一直阻塞直到任务 执行完成返回合并结果。
        int result = pool.invoke(task);
        System.out.println(result);
    }
}
```

思考：如果n为100000，执行上面的代码会发生什么问题?

在上面的例子中，由于递归计算Fibonacci数列的任务数量呈指数级增长，当n较大时，就容易出现StackOverflowError错误。这个错误通常发生在递归过程中，由于递归过程中每次调用函数都会在栈中创建一个新的栈帧，当递归深度过大时，栈空间就会被耗尽，导致StackOverflowError错误。

栈溢出如何解决

我们可以使用迭代的方式计算Fibonacci数列，以避免递归过程中占用大量的栈空间。下面是一个使用迭代方式计算Fibonacci数列的例子：

```
public class Fibonacci {
    public static void main(String[] args) {
        int n = 100000;
        long[] fib = new long[n + 1];
        fib[0] = 0;
        fib[1] = 1;
        for (int i = 2; i <= n; i++) {
            fib[i] = fib[i - 1] + fib[i - 2];
        }
        System.out.println(fib[n]);
    }
}
```

处理递归任务注意事项

对于一些递归深度较大的任务，使用Fork/Join框架可能会出现任务调度和内存消耗的问题。
当递归深度较大时，会产生大量的子任务，这些子任务可能被调度到不同的线程中执行，而线程的创建和销毁以及任务调度的开销都会占用大量的资源，从而导致性能下降。
此外，对于递归深度较大的任务，由于每个子任务所占用的栈空间较大，可能会导致内存消耗过大，从而引起内存溢出的问题。
因此，在使用Fork/Join框架处理递归任务时，需要根据实际情况来评估递归深度和任务粒度，以避免任务调度和内存消耗的问题。如果递归深度较大，可以尝试采用其他方法来优化算法，如使用迭代方式替代递归，或者限制递归深度来减少任务数量，以避免Fork/Join框架的缺点。

处理阻塞任务

在ForkJoinPool中使用阻塞型任务时需要注意以下几点：

1. **防止线程饥饿**：当一个线程在执行一个阻塞型任务时，它将会一直等待任务完成，这时如果没有其他线程可以窃取任务，那么该线程将一直被阻塞，直到任务完成为止。为了避免这种情况，应该避免在ForkJoinPool中提交大量的阻塞型任务。
2. **使用特定的线程池**：为了最大程度地利用ForkJoinPool的性能，可以使用专门的线程池来处理阻塞型任务，这些线程不会被ForkJoinPool的窃取机制所影响。例如，可以使用ThreadPoolExecutor来创建一个线程池，然后将这个线程池作为ForkJoinPool的执行器，这样就可以使用ThreadPoolExecutor来处理阻塞型任务，而使用ForkJoinPool来处理非阻塞型任务。
3. **不要阻塞工作线程**：如果在ForkJoinPool中使用阻塞型任务，那么需要确保这些任务不会阻塞工作线程，否则会导致整个线程池的性能下降。为了避免这种情况，可以将阻塞型任务提交到一个专门的线程池中，或者使用CompletableFuture等异步编程工具来处理阻塞型任务。

下面是一个使用阻塞型任务的例子，这个例子展示了如何使用CompletableFuture来处理阻塞型任务：

```java
public class BlockingTaskDemo {
    public static void main(String[] args) {
        //构建一个forkjoin线程池
        ForkJoinPool pool = new ForkJoinPool();

        //创建一个异步任务，并将其提交到ForkJoinPool中执行
        CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
            try {
                // 模拟一个耗时的任务
                TimeUnit.SECONDS.sleep(5);
                return "Hello, world!";
            } catch (InterruptedException e) {
                e.printStackTrace();
                return null;
            }
        }, pool);

        try {
            // 等待任务完成，并获取结果
            String result = future.get();

            System.out.println(result);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        } finally {
            //关闭ForkJoinPool，释放资源
            pool.shutdown();
        }
    }
}
```

在这个例子中，我们使用了CompletableFuture来处理阻塞型任务，因为它可以避免阻塞ForkJoinPool中的工作线程。另外，我们也可以使用专门的线程池来处理阻塞型任务，例如ThreadPoolExecutor等。不管是哪种方式，都需要避免在ForkJoinPool中提交大量的阻塞型任务，以免影响整个线程池的性能。

3.4 ForkJoinPool工作原理

ForkJoinPool 内部有多个任务队列，当我们通过 ForkJoinPool 的 invoke() 或者 submit() 方法提交任务时，ForkJoinPool 根据一定的路由规则把任务提交到一个任务队列中，如果任务在执行过程中会创建出子任务，那么子任务会提交到工作线程对应的任务队列中。

如果工作线程对应的任务队列空了，是不是就没活儿干了呢？不是的，ForkJoinPool 支持一种叫做“任务窃取”的机制，如果工作线程空闲了，那它可以“窃取”其他工作任务队列里的任务。如此一来，所有的工作线程都不会闲下来了。

核心设计：

- ForkJoinPool的任务会被内部存储了一个WorkQueue数组，提交给ForkJoinPool的任务会被分配到指定的WorkQueue上执行
- 每个WorkQueue内部维护了一个ForkJoinTask数组用来存储待执行的任务，以及一个独立的ForkJoinWorkerThread用来真正执行任务

工作线程ForkJoinWorkerThread

ForkJoinWorkerThread是ForkJoinPool中的一个专门用于执行任务的线程。当一个ForkJoinWorkerThread被创建时，它会自动注册一个WorkQueue到ForkJoinPool中。这个WorkQueue是该线程专门用于存储自己的任务的队列，只能出现在WorkQueues[]的奇数位。
ForkJoinWorkerThread工作线程启动后就会扫描偷取任务执行，另外当其在 ForkJoinTask#join() 等待返回结果时如果被 ForkJoinPool 线程池发现其任务队列为空或者已经将当前任务执行完毕，也会通过工作窃取算法从其他任务队列中获取任务分配到其任务队列中并执行。

工作队列WorkQueue

WorkQueue是一个双端队列，用于存储工作线程自己的任务。每个工作线程都会维护一个本地的WorkQueue，并且优先执行本地队列中的任务。当本地队列中的任务执行完毕后，工作线程会尝试从其他线程的WorkQueue中窃取任务。

WorkQueue 任务队列其实也分为了两种类型，一种是外部提交进来的任务所占用的队列，其在任务队列数组中的数组下标为偶数；另一种是属于工作线程私有的任务队列，保存大任务 fork 分解出来的任务，其在任务队列数组中的数组下标为奇数。

工作窃取

ForkJoinPool与ThreadPoolExecutor有个很大的不同之处在于，ForkJoinPool引入了工作窃取设计，它是其性能保证的关键之一。工作窃取，就是允许空闲线程从繁忙线程的双端队列中窃取任务。默认情况下，工作线程从它自己的双端队列的头部获取任务。但是，当自己的任务为空时，线程会从其他繁忙线程双端队列的尾部中获取任务。这种方法，最大限度地减少了线程竞争任务的可能性。

通过工作窃取，Fork/Join框架可以实现任务的自动负载均衡，以充分利用多核CPU的计算能力，同时也可以避免线程的饥饿和延迟问题

> 如果你对 ForkJoinPool 详细的实现细节感兴趣，也可以参考Doug Lea 的论文

3.5 ForkJoinPool执行流程

https://www.processon.com/view/link/5db81f97e4b0c55537456e9a

3.6 总结

Fork/Join是一种基于分治思想的模型，在并发处理计算型任务时有着显著的优势。其效率的提升主要得益于两个方面：

- **任务切分**：将大的任务分割成更小粒度的小任务，让更多的线程参与执行；
- **任务窃取**：通过任务窃取，充分地利用空闲线程，并减少竞争。

在使用ForkJoinPool时，需要特别注意任务的类型是否为纯函数计算类型，也就是这些任务不应该关心状态或者外界的变化，这样才是最安全的做法。如果是阻塞类型任务，那么你需要谨慎评估技术方案。虽然ForkJoinPool也能处理阻塞类型任务，但可能会带来复杂的管理成本。

和普通线程池之间的区别

- **工作窃取算法**

ForkJoinPool采用工作窃取算法来提高线程的利用率，而普通线程池则采用任务队列来管理任务。在工作窃取算法中，当一个线程完成自己的任务后，它可以从其它线程的队列中获取一个任务来执行，以此来提高线程的利用率。

- **任务的分解和合并**

ForkJoinPool可以将一个大任务分解为多个小任务，并行地执行这些小任务，最终将它们的结果合并起来得到最终结果。而普通线程池只能按照提交的任务顺序一个一个地执行任务。

- **工作线程的数量**

ForkJoinPool会根据当前系统的CPU核心数来自动设置工作线程的数量，以最大限度地发挥CPU的性能优势。而普通线程池需要手动设置线程池的大小，如果设置不合理，可能会导致线程过多或过少，从而影响程序的性能。

- **任务类型**

ForkJoinPool适用于执行大规模任务并行化，而普通线程池适用于执行一些短小的任务，如处理请求等。