任务类型

思考： 线程池的线程数设置多少合适?

我们调整线程池中的线程数量的最主要的目的是为了充分并合理地使用 CPU 和内存等资源，从而最大限度地提高程序的性能。在实际工作中，我们需要根据任务类型的不同选择对应的策略。

CPU密集型任务

CPU密集型任务也叫计算密集型任务，比如加密、解密、压缩、计算等一系列需要大量耗费 CPU 资源的任务。对于这样的任务最佳的线程数为 CPU 核心数的 1~2 倍，如果设置过多的线程数，实际

上并不会起到很好的效果。此时假设我们设置的线程数量是 CPU 核心数的 2 倍以上，因为计算任务非常重，会占用大量的 CPU 资源，所以这时 CPU 的每个核心工作基本都是满负荷的，而我们又设置了过多的线程，每个线程都想去利用 CPU 资源来执行自己的任务，这就会造成不必要的上下文切换，此时线程数的增多并没有让性能提升，反而由于线程数量过多会导致性能下降。

IO密集型任务

IO密集型任务，比如数据库、文件的读写，网络通信等任务，这种任务的特点是并不会特别消耗 CPU 资源，但是 IO 操作很耗时，总体会占用比较多的时间。对于这种任务最大线程数一般会大于 CPU 核心数很多倍，因为 IO 读写速度相比于 CPU 的速度而言是比较慢的，如果我们设置过少的线程数，就可能导致 CPU 资源的浪费。而如果我们设置更多的线程数，那么当一部分线程正在等待 IO 的时候，它们此时并不需要 CPU 来计算，那么另外的线程便可以利用 CPU 去执行其他的任务，互不影响，这样的话在工作队列中等待的任务就会减少，可以更好地利用资源。

线程数计算方法

《Java并发编程实战》的作者 Brain Goetz 推荐的计算方法：

```
线程数 = CPU 核心数 *（1+平均等待时间/平均工作时间）
```

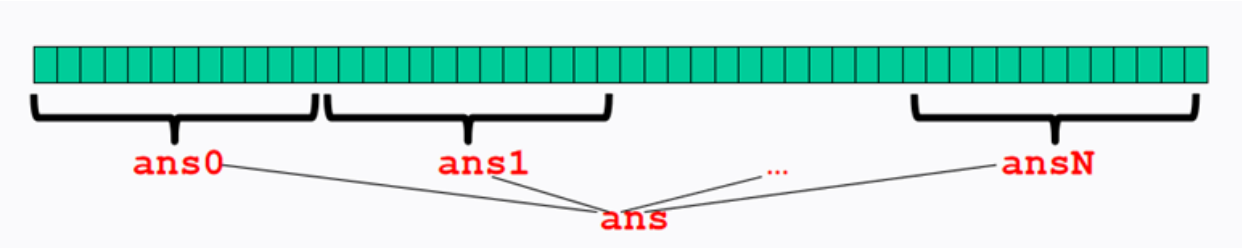
通过这个公式，我们可以计算出一个合理的线程数量，如果任务的平均等待时间长，线程数就随之增加，而如果平均工作时间长，也就是对于我们上面的 CPU 密集型任务，线程数就随之减少。

太少的线程数会使得程序整体性能降低，而过多的线程也会消耗内存等其他资源，所以如果想要更准确的话，可以进行压测，监控 JVM 的线程情况以及 CPU 的负载情况，根据实际情况衡量应该创建的线程数，合理并充分利用资源。

算法题

思考： 如何充分利用多核CPU的性能，计算一个很大数组中所有整数的和?

- 单线程相加，一个for循环搞定
- 利用多线程进行任务拆分，比如借助线程池进行分段相加，最后再把每个段的结果相加。

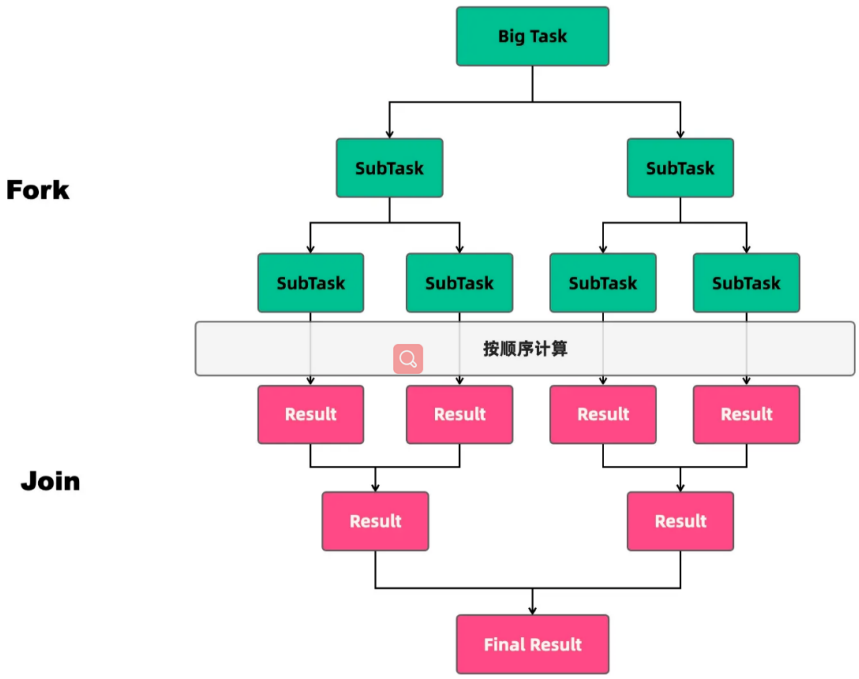


分治算法

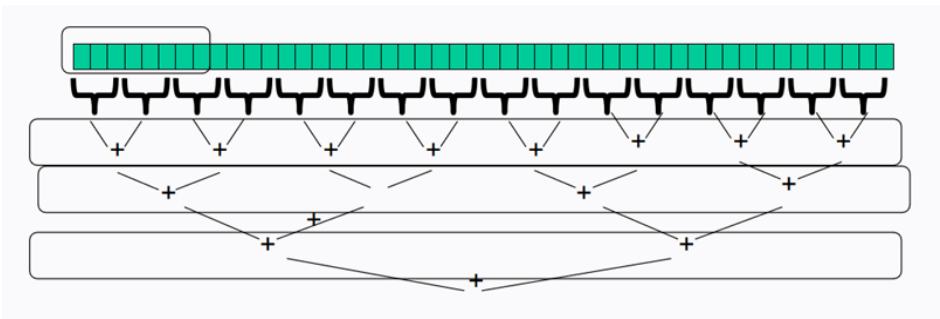
分治算法的基本思想是将一个规模为N的问题分解为K个规模较小的子问题，这些子问题相互独立且与原问题性质相同。求出子问题的解，就可得到原问题的解。

分治算法的步骤如下：

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。



在分治法中，子问题一般是相互独立的，因此，经常通过递归调用算法来求解子问题。



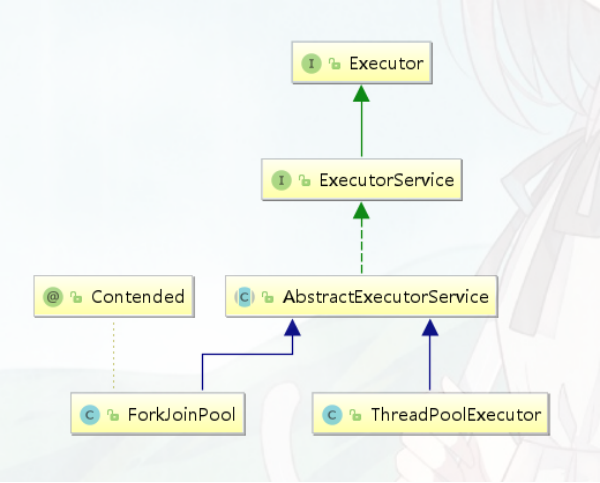
应用场景

分治思想在很多领域都有广泛的应用，例如算法领域有分治算法（归并排序、快速排序都属于分治算法，二分法查找也是一种分治算法）；大数据领域知名的计算框架 MapReduce 背后的思想也是分治。既然分治这种任务模型如此普遍，那 Java 显然也需要支持，Java 并发包里提供了一种叫做 Fork/Join 的并行计算框架，就是用来支持分治这种任务模型的。

Fork/Join框架介绍

传统线程池ThreadPoolExecutor有两个明显的缺点：一是无法对大任务进行拆分，对于某个任务只能由单线程执行；二是工作线程从队列中获取任务时存在竞争情况。这两个缺点都会影响任务的执

行效率。为了解决传统线程池的缺陷，Java7中引入Fork/Join框架，并在Java8中得到广泛应用。Fork/Join框架的核心是ForkJoinPool类，它是对AbstractExecutorService类的扩展。ForkJoinPool允许其他线程向它提交任务，并根据设定将这些任务拆分为粒度更细的子任务，这些子任务将由ForkJoinPool内部的工作线程来并行执行，并且工作线程之间可以窃取彼此之间的任务。



ForkJoinPool最适合计算密集型任务，而且最好是非阻塞任务。ForkJoinPool是ThreadPoolExecutor线程池的一种补充，是对计算密集型场景的加强。

根据经验和实验，任务总数、单任务执行耗时以及并行数都会影响到Fork/Join的性能。所以，当你使用Fork/Join框架时，你需要谨慎评估这三个指标，最好能通过模拟对比评估，不要凭感觉冒然在生产环境使用。

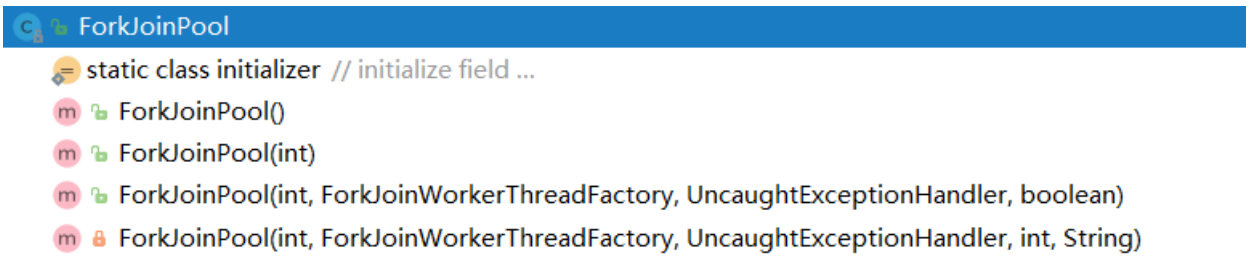
Fork/Join的使用

Fork/Join 计算框架主要包含两部分，一部分是分治任务的线程池 ForkJoinPool，另一部分是分治任务 ForkJoinTask

ForkJoinPool

ForkJoinPool 是用于执行 ForkJoinTask 任务的执行池，不再是传统执行池 Worker+Queue 的组合式，而是维护了一个队列数组 WorkQueue（WorkQueue[]），这样在提交任务和线程任务的时候大幅度减少碰撞。

ForkJoinPool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：asyncMode ? FIFO_QUEUE : LIFO_QUEUE。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。

按类型提交不同任务

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)

- execute类型的方法在提交任务后，不会返回结果。ForkJoinPool不仅允许提交ForkJoinTask类型任务，还允许提交Runnable任务

> 执行Runnable类型任务时，将会转换为ForkJoinTask类型。由于任务是不可切分的，所以这类任务无法获得任务拆分这方面的效益，不过仍然可以获得任务窃取带来的好处和性能提升。

- invoke方法接受ForkJoinTask类型的任务，并在任务执行结束后，返回泛型结果。如果提交的任务是null，将抛出空指针异常。
- submit方法支持三种类型的任务提交：ForkJoinTask类型、Callable类型和Runnable类型。在提交任务后，将返回ForkJoinTask类型的结果。如果提交的任务是null，将抛出空指针异常，并且当任务不能按计划执行的话，将抛出任务拒绝异常。

```
1 //递归任务  用于计算数组总和
2 LongSum ls = new LongSum(array, 0, array.length);
3 // 构建ForkJoinPool
4 ForkJoinPool fjp  = new ForkJoinPool(12);
5 //ForkJoin计算数组总和
```

ForkJoinTask

ForkJoinTask是ForkJoinPool的核心之一，它是任务的实际载体，定义了任务执行时的具体逻辑和拆分逻辑。ForkJoinTask继承了Future接口，所以也可以将其看作是轻量级的Future。

ForkJoinTask 是一个抽象类，它的方法有很多，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。

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

```
public class LongSum extends RecursiveTask<Long> {
    // 任务拆分最小阈值
    static final int SEQUENTIAL_THRESHOLD = 10000;

    int low;
    int high;
    int[] array;

    LongSum(int[] arr, int lo, int hi) {
        array = arr;
        low = lo;
        high = hi;
    }

    @Override
    protected Long compute() {

        //当任务拆分到小于等于阀值时开始求和
        if (high - low <= SEQUENTIAL_THRESHOLD) {

            long sum = 0;
            for (int i = low; i < high; ++i) {
                sum += array[i];
            }
            return sum;
```

```
        } else {  // 任务过大继续拆分
            int mid = low + (high - low) / 2;
            LongSum left = new LongSum(array, low, mid);
            LongSum right = new LongSum(array, mid, high);
            // 提交任务
            left.fork();
            right.fork();
            //获取任务的执行结果,将阻塞当前线程直到对应的子任务完成运行并返回结果
            long rightAns = right.join();
            long leftAns = left.join();
            return leftAns + rightAns;
        }
    }
```

ForkJoinTask使用限制

ForkJoinTask最适合用于纯粹的计算任务，也就是纯函数计算，计算过程中的对象都是独立的，对外部没有依赖。提交到ForkJoinPool中的任务应避免执行阻塞I/O。

ForkJoinPool 的工作原理

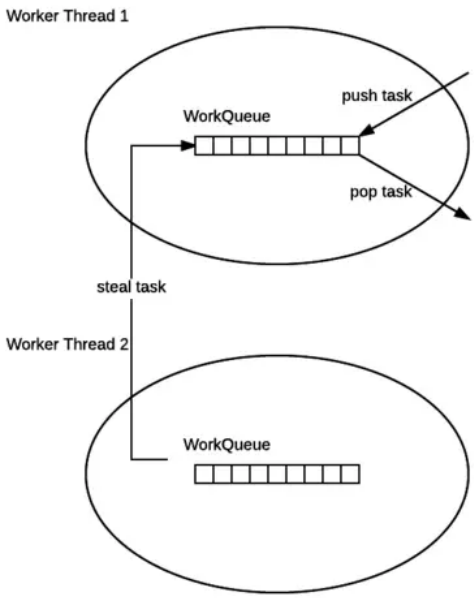
- ForkJoinPool 内部有多个工作队列，当我们通过 ForkJoinPool 的 invoke() 或者 submit() 方法提交任务时，ForkJoinPool 根据一定的路由规则把任务提交到一个工作队列中，如果任务在执行过程中会创建出子任务，那么子任务会提交到工作线程对应的工作队列中。
- ForkJoinPool 的每个工作线程都维护着一个工作队列（WorkQueue），这是一个双端队列（Deque），里面存放的对象是任务（ForkJoinTask）。
- 每个工作线程在运行中产生新的任务（通常是因为调用了 fork()）时，会放入工作队列的top，并且工作线程在处理自己的工作队列时，使用的是 LIFO 方式，也就是说每次从top取出任务来执行。
- 每个工作线程在处理自己的工作队列同时，会尝试窃取一个任务，窃取的任务位于其他线程的工作队列的base，也就是说工作线程在窃取其他工作线程的任务时，使用的是FIFO 方式。
- 在遇到 join() 时，如果需要 join 的任务尚未完成，则会先处理其他任务，并等待其完成。
- 在既没有自己的任务，也没有可以窃取的任务时，进入休眠 。

工作窃取

ForkJoinPool与ThreadPoolExecutor有个很大的不同之处在于，ForkJoinPool存在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。

ForkJoinPool的大部分操作都发生在工作窃取队列（work-stealing queues ） 中，该队列由内部类WorkQueue实现。它是Deques的特殊形式，但仅支持三种操作方式：push、pop和poll（也称为窃取）。在ForkJoinPool中，队列的读取有着严格的约束，push和pop仅能从其所属线程调用，而poll则可以从其他线程调用。

工作窃取的运行流程如下图所示：



- 工作窃取算法的优点是充分利用线程进行并行计算，并减少了线程间的竞争;
- 工作窃取算法缺点是在某些情况下还是存在竞争，比如双端队列里只有一个任务时。并且消耗了更多的系统资源，比如创建多个线程和多个双端队列。

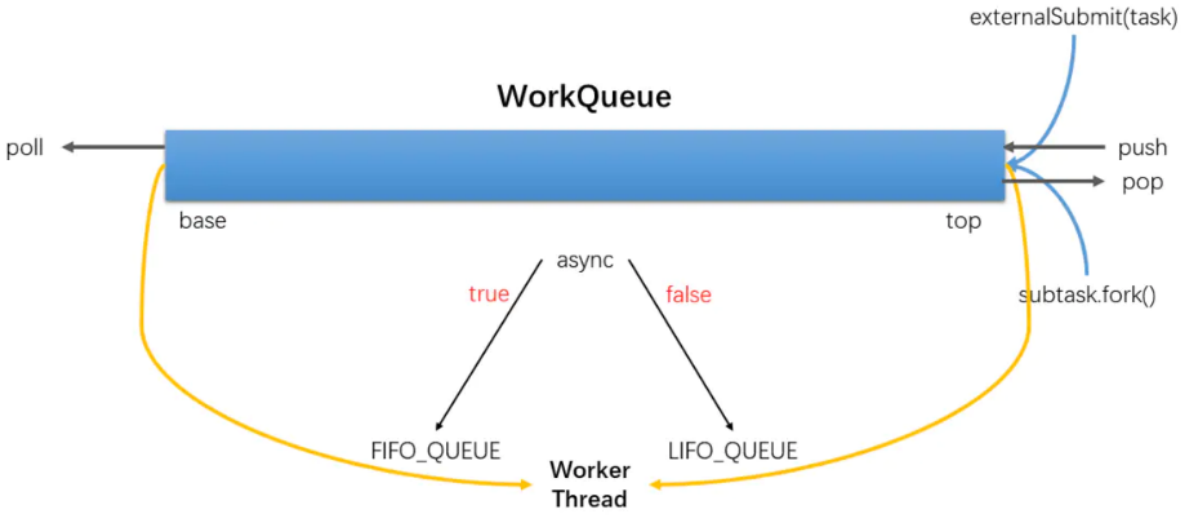
思考：为什么这么设计，工作线程总是从头部获取任务，窃取线程从尾部获取任务？

这样做的主要原因是为了提高性能，通过始终选择最近提交的任务，可以增加资源仍分配在CPU缓存中的机会，这样CPU处理起来要快一些。而窃取者之所以从尾部获取任务，则是为了降低线程之间的竞争可能，毕竟大家都从一个部分拿任务，竞争的可能要大很多。

此外，这样的设计还有一种考虑。由于任务是可分割的，那队列中较旧的任务最有可能粒度较大，因为它们可能还没有被分割，而空闲的线程则相对更有“精力”来完成这些粒度较大的任务。

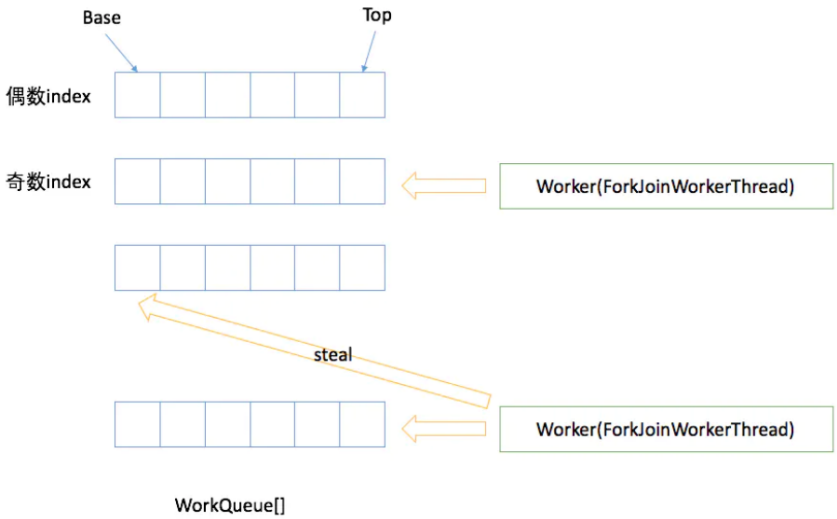
工作队列WorkQueue

- WorkQueue 是双向列表，用于任务的有序执行，如果 WorkQueue 用于自己的执行线程 Thread，线程默认将会从尾端选取任务用来执行 LIFO。
- 每个 ForkJoinWorkThread 都有属于自己的 WorkQueue，但不是每个 WorkQueue 都有对应的 ForkJoinWorkThread。
- 没有 ForkJoinWorkThread 的 WorkQueue 保存的是 submission，来自外部提交，在WorkQueues[] 的下标是 偶数位。



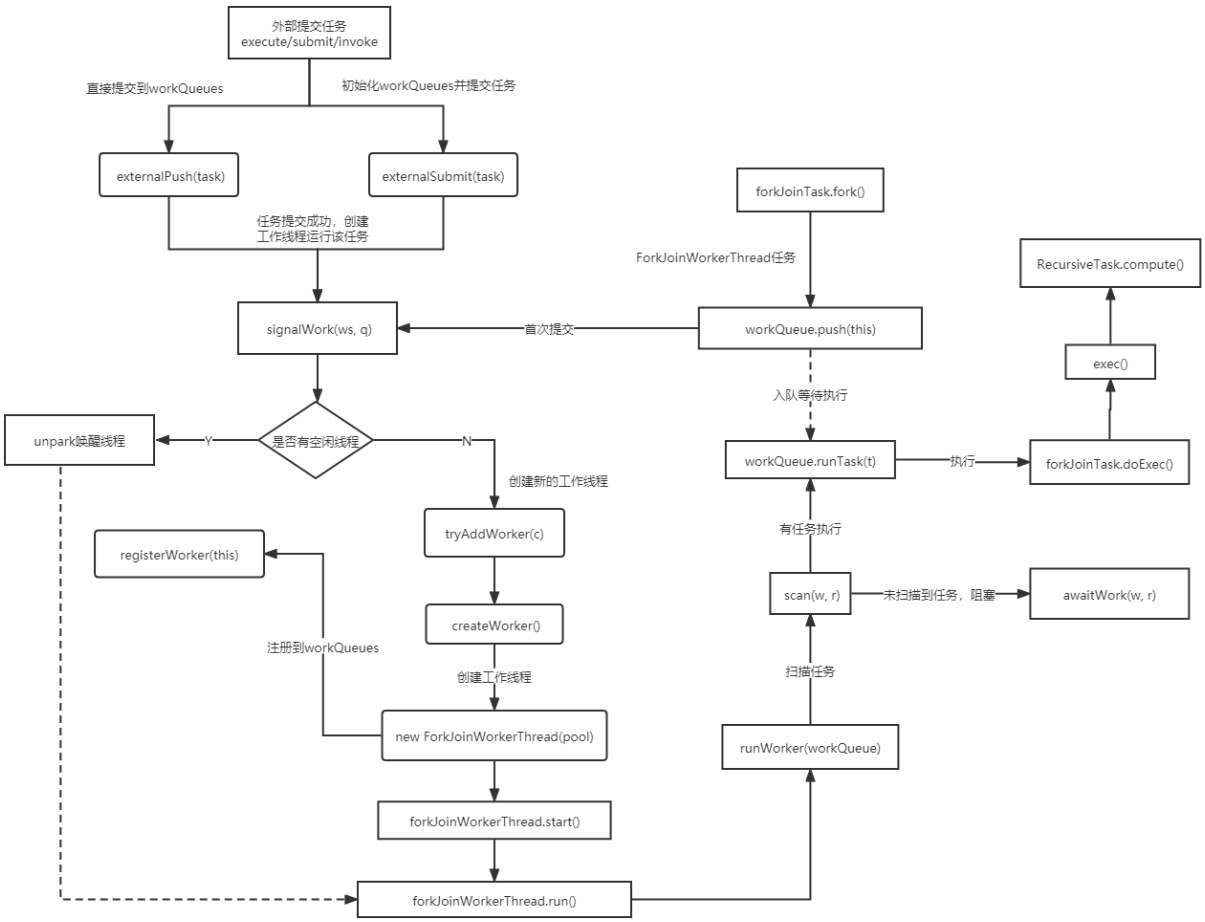
ForkJoinWorkThread

ForkJoinWorkThread 是用于执行任务的线程，用于区别使用非 ForkJoinWorkThread 线程提交 task。启动一个该 Thread，会自动注册一个 WorkQueue 到 Pool，拥有 Thread 的 WorkQueue 只能出现在 WorkQueues[] 的 奇数 位。



ForkJoinPool执行流程

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



总结

Fork/Join是一种基于分治算法的模型，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。