Java Thread Pool Detailed Explanation

Guide Java Java Concurrency About 8577 words About 29 minutes

Pooling technology is undoubtedly familiar to everyone. Thread pools, database connection pools, HTTP connection pools, and more are all applications of this concept. The main purpose of pooling technology is to reduce the consumption of each resource acquisition and improve resource utilization.

In this article, I will introduce the basic concepts and core principles of thread pools in detail.

Thread Pool Introduction

As the name suggests, a thread pool is a resource pool that manages a series of threads. It provides a way to limit and manage thread resources. Each thread pool also maintains some basic statistics, such as the number of completed tasks.

Here I borrow some content from the book "The Art of Java Concurrency Programming" to summarize the benefits of using thread pools:

- **Reduce resource consumption** . By reusing existing threads, you can reduce the consumption caused by thread creation and destruction.
- **Improve responsiveness** . When a task arrives, it can be executed immediately without waiting for a thread to be created.
- **Improve thread manageability** . Threads are a scarce resource. If they are created without limit, they will not only consume system resources but also reduce system stability. Using a thread pool allows for unified allocation, tuning, and monitoring.

Thread pools are generally used to execute multiple unrelated time-consuming tasks. Without multithreading, tasks are executed sequentially. Using a thread pool allows multiple unrelated tasks to be executed simultaneously.

Introduction to the Executor Framework

`Executor` The framework was introduced after Java 5. After Java 5, `Executor` starting threads through is better than using the method `Thread` of . `start` In addition to being easier to manage and more efficient (implemented with a thread pool to save overhead), there is another key point: it helps avoid the "this" escape problem.

> This escape means that other threads hold a reference to the object before the constructor returns. Calling methods on objects that have not been fully constructed may cause confusing errors.

`Executor` The framework not only includes thread pool management, but also provides thread factories, queues, and rejection strategies. `Executor` The framework makes concurrent programming easier.

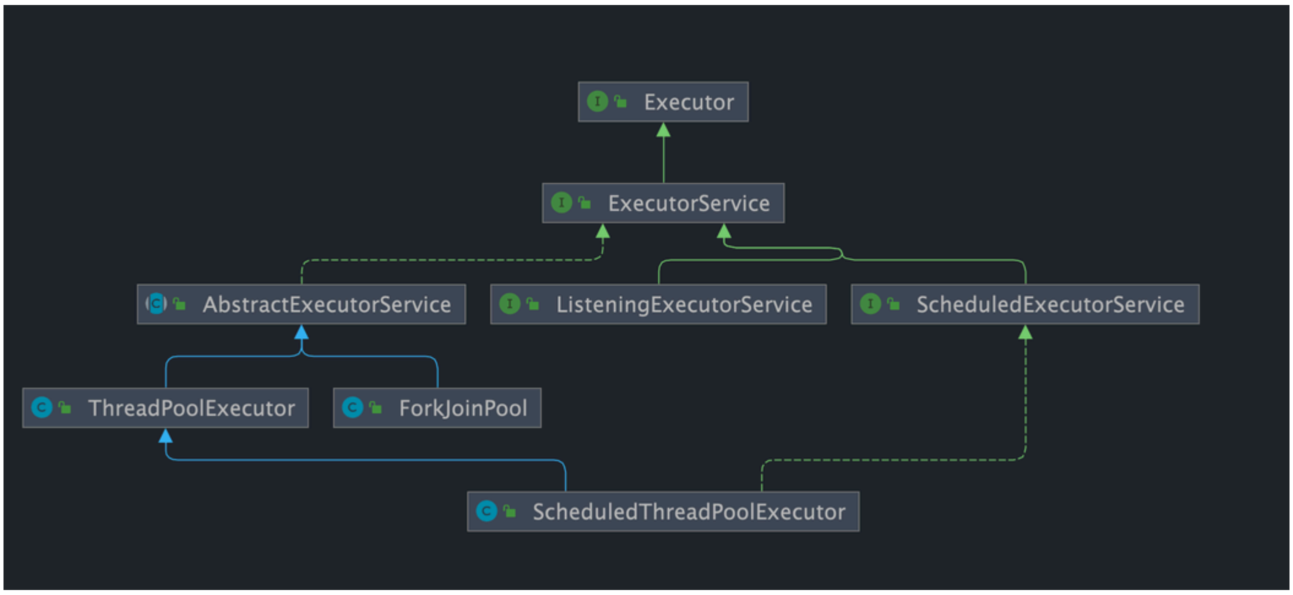
`Executor` The framework structure mainly consists of three parts:

1. Task (`Runnable` / `Callable`)

`Runnable` The interface or **`Callable` interfaces** that need to be implemented to perform the task . **`Runnable` Interfaces** or **`Callable` interface** implementation classes can be executed by **`ThreadPoolExecutor`** or **`ScheduledThreadPoolExecutor`** .

2. Task execution (`Executor`)

As shown in the figure below, it includes the core interface of the task execution mechanism **`Executor`** and the interface inherited `Executor` from the interface **`ExecutorService`** **.** **`ThreadPoolExecutor`** The two key classes , and , **`ScheduledThreadPoolExecutor`** implement **`ExecutorService`** the interface.



A lot of underlying class relationships are mentioned here, but in fact we need to pay more attention to `ThreadPoolExecutor` this class, which is used very frequently in our actual use of thread pools.

Note: By looking at `ScheduledThreadPoolExecutor` the source code, we find `ScheduledThreadPoolExecutor` that actually inherits `ThreadPoolExecutor` and implements `ScheduledExecutorService` , which `ScheduledExecutorService` in turn implements `ExecutorService` , as shown in the class relationship diagram given above.

`ThreadPoolExecutor` Class Description:

```java
//AbstractExecutorService实现了ExecutorService接口
public class ThreadPoolExecutor extends AbstractExecutorService
```

`ScheduledThreadPoolExecutor` Class Description:

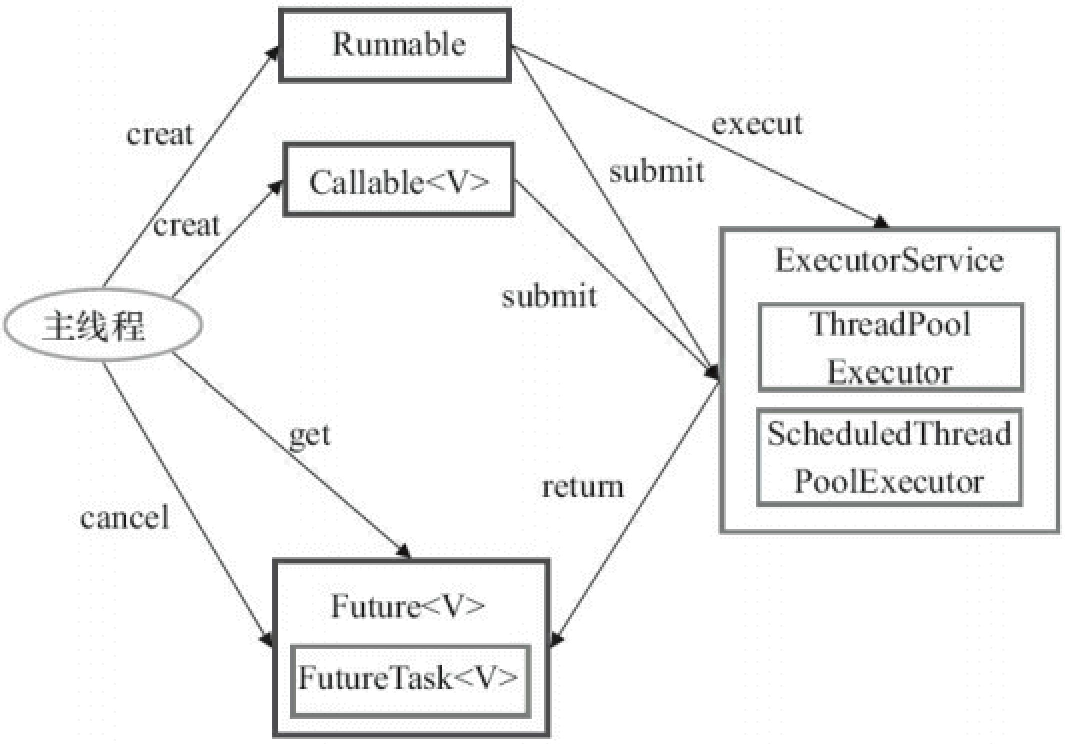
```java
//ScheduledExecutorService继承ExecutorService接口
public class ScheduledThreadPoolExecutor
        extends ThreadPoolExecutor
        implements ScheduledExecutorService
```

3. Results of asynchronous calculations (`Future`)

`Future` Both interfaces and `Future` interface implementation classes **`FutureTask`** can represent the results of asynchronous computations.

When we submit the implementation class of **`Runnable` interface** or **`Callable` interface** **`ThreadPoolExecutor`** to or **`ScheduledThreadPoolExecutor`** for execution. (`submit()` When calling the method, a **`FutureTask`** object will be returned)

`Executor` Framework usage diagram :



1. The main thread must first create a task object that implements the `Runnable` or `Callable` interface.
2. Submit the created implementation `Runnable` / `Callable` interface object directly `ExecutorService` to Execute: `ExecutorService.execute（Runnable command）`) or you can submit `Runnable` the object or `Callable` object to `ExecutorService` Execute (`ExecutorService.submit（Runnable task）` or `ExecutorService.submit（Callable <T> task）`).
3. If executed `ExecutorService.submit（…）` , `ExecutorService` it will return an `Future` object that implements the interface (we have just mentioned the difference between the execution `execute()` method and `submit()` the method, `submit()` which will return an `FutureTask 对象）。由于 FutureTask` implementation `Runnable` . We can also create it `FutureTask` and then directly hand it over `ExecutorService` to execution.
4. Finally, the main thread can execute `FutureTask.get()` methods to wait for the task to complete. The main thread can also execute `FutureTask.cancel（boolean mayInterruptIfRunning）` to cancel the execution of this task.

Introduction to the ThreadPoolExecutor class (important)

The thread pool implementation class `ThreadPoolExecutor` is `Executor` the core class of the framework.

Thread pool parameter analysis

`ThreadPoolExecutor` The class provides four constructors. Let's look at the longest one, and the other three are generated based on this constructor (the other constructors are basically constructors that give certain default parameters, such as what the default rejection policy is).

```java
    /**
     * 用给定的初始参数创建一个新的ThreadPoolExecutor。
     */
    public ThreadPoolExecutor(int corePoolSize,//线程池的核心线程数量
                              int maximumPoolSize,//线程池的最大线程数
                              long keepAliveTime,//当线程数大于核心线程数时，多余的空闲线程存活的最长时间
                              TimeUnit unit,//时间单位
                              BlockingQueue<Runnable> workQueue,//任务队列，用来储存等待执行任务的队列
                              ThreadFactory threadFactory,//线程工厂，用来创建线程，一般默认即可
                              RejectedExecutionHandler handler//拒绝策略，当提交的任务过多而不能及时处理时，我们可以定制策略来处理任务
                               ) {
        if (corePoolSize < 0 ||
            maximumPoolSize <= 0 ||
            maximumPoolSize < corePoolSize ||
            keepAliveTime < 0)
            throw new IllegalArgumentException();
        if (workQueue == null || threadFactory == null || handler == null)
            throw new NullPointerException();
        this.corePoolSize = corePoolSize;
        this.maximumPoolSize = maximumPoolSize;
        this.workQueue = workQueue;
        this.keepAliveTime = unit.toNanos(keepAliveTime);
        this.threadFactory = threadFactory;
        this.handler = handler;
    }
```

The following parameters are very important and you will definitely need them later when using the thread pool! So, be sure to take a small notebook and write them down clearly.

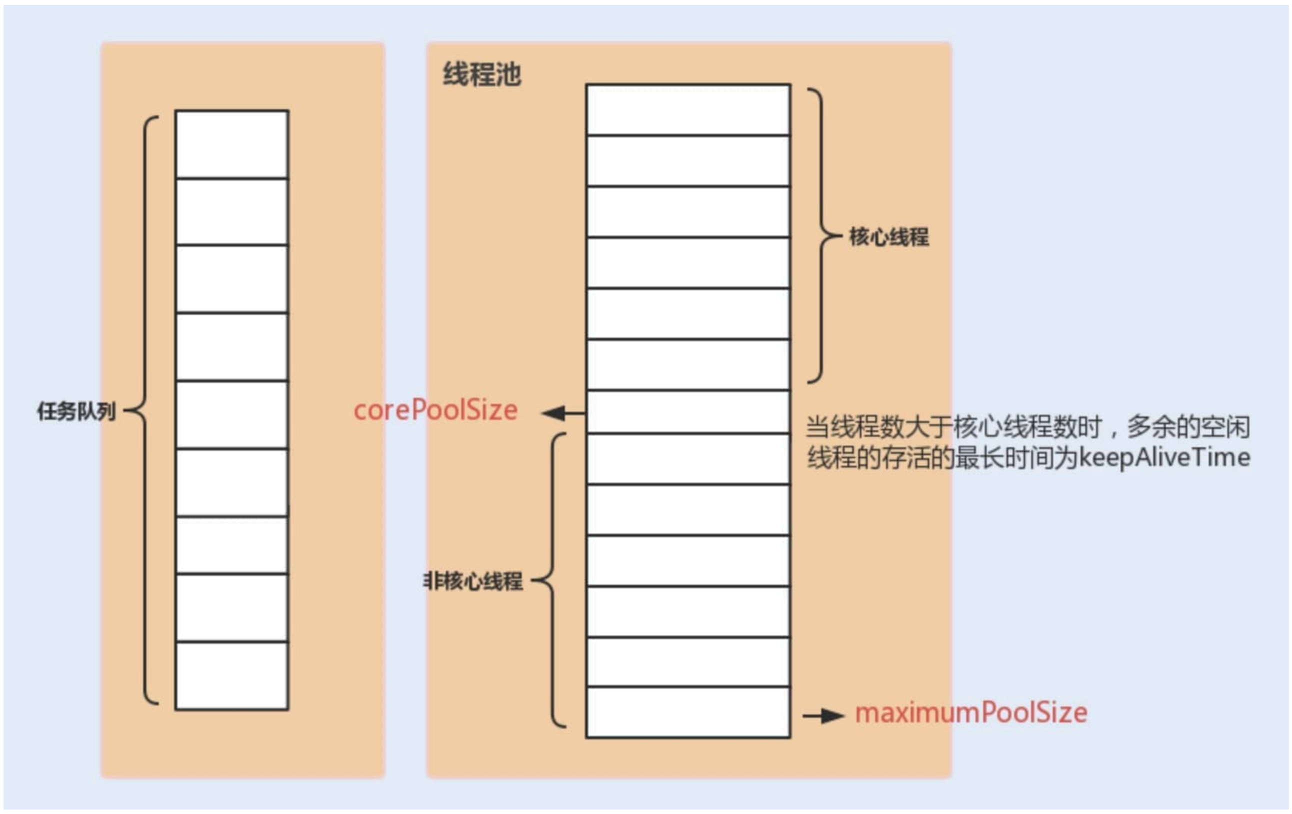
`ThreadPoolExecutor` The 3 most important parameters are:

- `corePoolSize` : The maximum number of threads that can run simultaneously when the task queue has not reached the queue capacity.
- `maximumPoolSize` : When the number of tasks stored in the task queue reaches the queue capacity, the number of threads that can currently run simultaneously becomes the maximum number of threads.
- `workQueue` : When a new task comes, it will first determine whether the number of currently running threads reaches the number of core threads. If so, the new task will be stored in the queue.

`ThreadPoolExecutor` Other common parameters:

- `keepAliveTime` : When the number of threads in the thread pool is greater `corePoolSize` than , if no new tasks are submitted at this time, the threads outside the core threads will not be destroyed immediately, but will wait until the waiting time exceeds and `keepAliveTime` then be recycled and destroyed.
- `unit` : `keepAliveTime` The time unit of the parameter.
- `threadFactory` :executor is used when creating a new thread.
- `handler` :Rejection strategy (will be introduced in detail later).

The following diagram can help you better understand the relationship between various parameters in the thread pool (Image source: "Java Performance Tuning Practice"):


线程池
任务队列
核心线程
corePoolSize
当线程数大于核心线程数时，多余的空闲线程的存活的最长时间为keepAliveTime
非核心线程
maximumPoolSize

`ThreadPoolExecutor` Deny policy definition:

If the number of threads currently running simultaneously reaches the maximum number of threads and the queue is full of tasks, `ThreadPoolExecutor` define some strategies:

- `ThreadPoolExecutor.AbortPolicy` : Throw `RejectedExecutionException` to reject the processing of new tasks.
- `ThreadPoolExecutor.CallerRunsPolicy` Calls the execution thread to run the task, that is, `execute` runs the task directly in the calling thread. (`run`) Rejected tasks are discarded if the executor is shut down. Therefore, this strategy slows down the submission of new tasks, affecting overall application performance. If your application can tolerate this delay and you require that all task requests be executed, you can choose this strategy.
- `ThreadPoolExecutor.DiscardPolicy` : Do not process new tasks and discard them directly.
- `ThreadPoolExecutor.DiscardOldestPolicy` : This strategy will discard the oldest unprocessed task request.

For example:

For example, when Spring creates a thread pool `ThreadPoolTaskExecutor` directly through `ThreadPoolExecutor` the constructor, if we do not specify `RejectedExecutionHandler` a rejection strategy to configure the thread pool, the default rejection strategy is used `AbortPolicy` . Under this rejection strategy, if the queue is full,

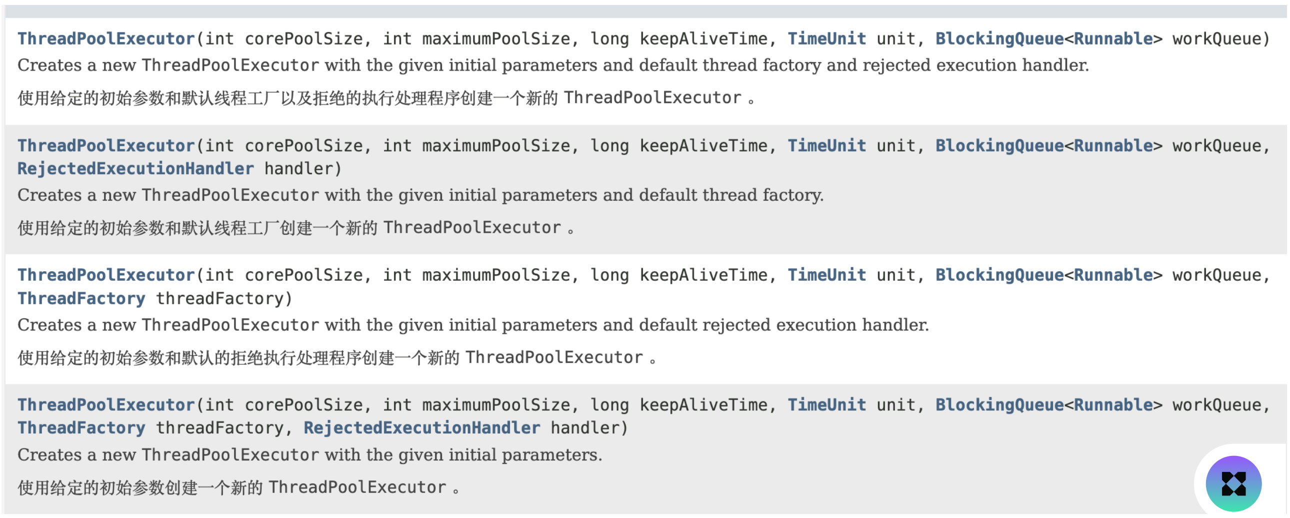
an exception `ThreadPoolExecutor` will be thrown `RejectedExecutionException` to reject new tasks, which means you will lose the processing of this task. If you do not want to discard tasks, you can use this strategy `CallerRunsPolicy` .
`CallerRunsPolicy` Unlike the other strategies, it neither discards tasks nor throws exceptions. Instead, it returns the task to the caller and uses the caller's thread to execute the task.

```java
public static class CallerRunsPolicy implements RejectedExecutionHandler {

        public CallerRunsPolicy() { }

        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
            if (!e.isShutdown()) {
                // 直接主线程执行，而不是线程池中的线程执行
                r.run();
            }
        }
    }
```

Two ways to create a thread pool

In Java, there are two main ways to create a thread pool:

Method 1: `ThreadPoolExecutor` Create directly through the constructor (recommended)



`ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue)`
Creates a new ThreadPoolExecutor with the given initial parameters and default thread factory and rejected execution handler.
使用给定的初始参数和默认线程工厂以及拒绝的执行处理程序创建一个新的 ThreadPoolExecutor 。

`ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, RejectedExecutionHandler handler)`
Creates a new ThreadPoolExecutor with the given initial parameters and default thread factory.
使用给定的初始参数和默认线程工厂创建一个新的 ThreadPoolExecutor 。

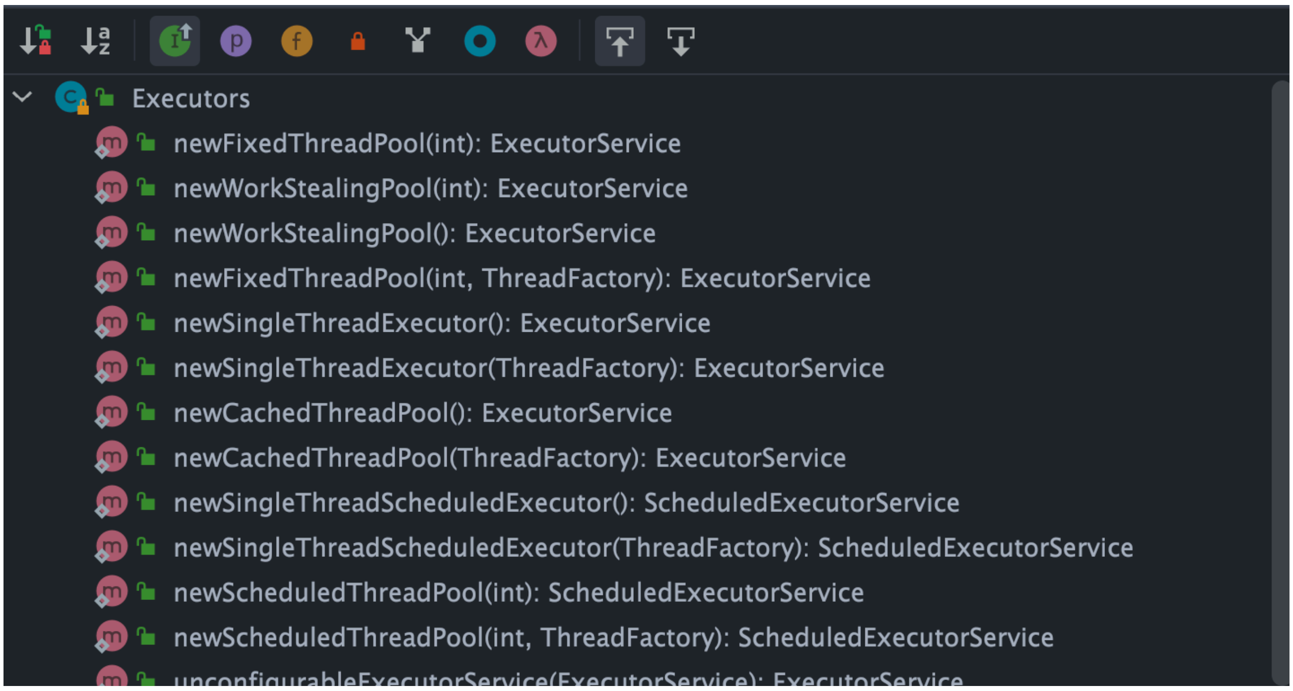
`ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory)`
Creates a new ThreadPoolExecutor with the given initial parameters and default rejected execution handler.
使用给定的初始参数和默认的拒绝执行处理程序创建一个新的 ThreadPoolExecutor 。

`ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory, RejectedExecutionHandler handler)`
Creates a new ThreadPoolExecutor with the given initial parameters.
使用给定的初始参数创建一个新的 ThreadPoolExecutor 。

This is the most recommended approach because it allows developers to explicitly specify the core parameters of the thread pool and have more precise control over the running behavior of the thread pool, thereby avoiding the risk of resource exhaustion.

Method 2: `Executors` Create through tool class (not recommended for production environment)

`Executors` The method for creating a thread pool provided by the tool class is shown in the following figure:



```
Executors
    newFixedThreadPool(int): ExecutorService
    newWorkStealingPool(int): ExecutorService
    newWorkStealingPool(): ExecutorService
    newFixedThreadPool(int, ThreadFactory): ExecutorService
    newSingleThreadExecutor(): ExecutorService
    newSingleThreadExecutor(ThreadFactory): ExecutorService
    newCachedThreadPool(): ExecutorService
    newCachedThreadPool(ThreadFactory): ExecutorService
    newSingleThreadScheduledExecutor(): ScheduledExecutorService
    newSingleThreadScheduledExecutor(ThreadFactory): ScheduledExecutorService
    newScheduledThreadPool(int): ScheduledExecutorService
    newScheduledThreadPool(int, ThreadFactory): ScheduledExecutorService
    unconfigurableExecutorService(ExecutorService): ExecutorService
```

`Executors` It can be seen that various types of thread pools can be created through tool classes, including:

- `FixedThreadPool` A thread pool with a fixed number of threads. The number of threads in this thread pool remains constant. When a new task is submitted, it is immediately executed if there are idle threads in the thread pool. If not, the new task is temporarily stored in a task queue and processed when a free thread becomes available.
- `SingleThreadExecutor` : A thread pool with only one thread. If more than one task is submitted to the thread pool, the tasks are stored in a task queue and executed in first-in, first-out order when a thread becomes idle.
- `CachedThreadPool` A thread pool with adjustable thread counts. The number of threads in the thread pool is fixed, but if there are idle threads available for reuse, they will be prioritized. If all threads are busy and a new task is submitted, a new thread will be created to handle the task. After all threads complete their current tasks, they w returned to the thread pool for reuse.

- `ScheduledThreadPool` : A thread pool that runs tasks after a given delay or executes tasks periodically.

The Alibaba Java Development Manual prohibits thread pools from `Executors` being created using , but rather through `ThreadPoolExecutor` the constructor. This approach allows coders to more clearly understand the thread pool's operating rules and avoid the risk of resource exhaustion.

`Executors` The disadvantages of returning the thread pool object are as follows (which will be described in detail later):

- `FixedThreadPool` And `SingleThreadExecutor` : A blocking queue is used `LinkedBlockingQueue` , and the maximum length of the task queue is `Integer.MAX_VALUE` , which can be regarded as unbounded. A large number of requests may accumulate, resulting in OOM.
- `CachedThreadPool` : Using a synchronous queue `SynchronousQueue` , the number of threads allowed to be created is `Integer.MAX_VALUE` . If the number of tasks is too large and the execution speed is slow, a large number of threads may be created, resulting in OOM.
- `ScheduledThreadPool` And `SingleThreadScheduledExecutor` : The unbounded delayed blocking queue used `DelayedWorkQueue` , the maximum length of the task queue is `Integer.MAX_VALUE` , a large number of requests may accumulate, resulting in OOM.

```java
public static ExecutorService newFixedThreadPool(int nThreads) {
    // LinkedBlockingQueue 的默认长度为 Integer.MAX_VALUE，可以看作是无界的
    return new ThreadPoolExecutor(nThreads, nThreads,0L, TimeUnit.MILLISECONDS,new LinkedBlockingQueue<Runnable>());

}

public static ExecutorService newSingleThreadExecutor() {
    // LinkedBlockingQueue 的默认长度为 Integer.MAX_VALUE，可以看作是无界的
    return new FinalizableDelegatedExecutorService (new ThreadPoolExecutor(1, 1,0L, TimeUnit.MILLISECONDS,new LinkedBlockingQueue<Runnable>()));

}

// 同步队列 SynchronousQueue，没有容量，最大线程数是 Integer.MAX_VALUE`
```

```java
public static ExecutorService newCachedThreadPool() {

    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,60L,
TimeUnit.SECONDS,new SynchronousQueue<Runnable>());

}

// DelayedWorkQueue（延迟阻塞队列）
public static ScheduledExecutorService newScheduledThreadPool(int
corePoolSize) {
    return new ScheduledThreadPoolExecutor(corePoolSize);
}
public ScheduledThreadPoolExecutor(int corePoolSize) {
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
          new DelayedWorkQueue());
}
```

Summary of commonly used blocking queues in thread pools

When a new task arrives, it will first determine whether the number of currently running threads reaches the number of core threads. If so, the new task will be stored in the queue.

Different thread pools will use different blocking queues, which we can analyze in combination with the built-in thread pool.

- (Unbounded queue) with capacity: `Integer.MAX_VALUE` and . At most , only the number of core threads can be created (the number of core threads and the maximum number of threads are equal), and only one thread can be created (the number of core threads and the maximum number of threads are both 1). The task queues of both will never be full. `LinkedBlockingQueue` `FixedThreadPool` `SingleThreadExector` `FixedThreadPool` `SingleThreadExector`
- `SynchronousQueue` (Synchronous Queue): It has no capacity and does not store elements `CachedThreadPool` . `SynchronousQueue` Its purpose is to ensure that if a free thread is available, it will be used to process the submitted task; otherwise, a new thread will be created to process the task. In other words, `CachedThreadPool` the maximum number of threads is `Integer.MAX_VALUE` , which can be understood as infinitely scalable. This may create a large number of threads, leading to OOM errors.
- `DelayedWorkQueue` Delayed Blocking Queue: The internal elements of the `ScheduledThreadPool` queue are not sorted by the time they were added, but rather by

the length of their delay. The internal "heap" data structure ensures that the task dequeued is the one with the earliest execution time in the queue. When the queue is full, it automatically expands to half its original capacity, meaning it never blocks. The maximum expansion capacity is achievable , so it can only create a maximum of the number of core threads. `SingleThreadScheduledExecutor` `DelayedWorkQueue` `DelayedWorkQueue` `Integer.MAX_VALUE`

Analysis of thread pool principles (important)

We have explained `Executor` the framework and class above. Now let's put it into practice and review the above content `ThreadPoolExecutor` by writing a small Demo. `ThreadPoolExecutor`

Thread pool sample code

First, create an `Runnable` implementation class of the interface (of course, it can also be `Callable` an interface, we will introduce the difference between the two later.)

`MyRunnable.java`

```java
import java.util.Date;

/**
 * 这是一个简单的Runnable类，需要大约5秒钟来执行其任务。
 * @author shuang.kou
 */
public class MyRunnable implements Runnable {

    private String command;

    public MyRunnable(String s) {
        this.command = s;
    }

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " Start. Time = " + new Date());
        processCommand();
```

```java
            System.out.println(Thread.currentThread().getName() + " End. Time = " + new Date());
        }

        private void processCommand() {
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        @Override
        public String toString() {
            return this.command;
        }
    }
```

When writing a test program, we use `ThreadPoolExecutor` the method recommended by Alibaba to create a thread pool using constructor custom parameters.

`ThreadPoolExecutorDemo.java`

```java
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class ThreadPoolExecutorDemo {

    private static final int CORE_POOL_SIZE = 5;
    private static final int MAX_POOL_SIZE = 10;
    private static final int QUEUE_CAPACITY = 100;
    private static final Long KEEP_ALIVE_TIME = 1L;
    public static void main(String[] args) {

        //使用阿里巴巴推荐的创建线程池的方式
        //通过ThreadPoolExecutor构造函数自定义参数创建
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
                CORE_POOL_SIZE,
                MAX_POOL_SIZE,
                KEEP_ALIVE_TIME,
                TimeUnit.SECONDS,
                new ArrayBlockingQueue<>(QUEUE_CAPACITY),
                new ThreadPoolExecutor.CallerRunsPolicy());

        for (int i = 0; i < 10; i++) {
            //创建WorkerThread对象（WorkerThread类实现了Runnable 接口）
            Runnable worker = new MyRunnable("" + i);
            //执行Runnable
            executor.execute(worker);
        }
        //终止线程池
        executor.shutdown();
        while (!executor.isTerminated()) {
        }
        System.out.println("Finished all threads");
    }
}

```

You can see that the code above specifies:

- `corePoolSize` : The number of core threads is 5.
- `maximumPoolSize` : Maximum number of threads: 10
- `keepAliveTime` : The waiting time is 1L.
- `unit` : The unit of waiting time is TimeUnit.SECONDS.
- `workQueue` ： The task queue is `ArrayBlockingQueue` and the capacity is 100;
- `handler` :Rejection strategy is `CallerRunsPolicy` .

Output structure :

```plain
pool-1-thread-3 Start. Time = Sun Apr 12 11:14:37 CST 2020
pool-1-thread-5 Start. Time = Sun Apr 12 11:14:37 CST 2020
pool-1-thread-2 Start. Time = Sun Apr 12 11:14:37 CST 2020
pool-1-thread-1 Start. Time = Sun Apr 12 11:14:37 CST 2020
pool-1-thread-4 Start. Time = Sun Apr 12 11:14:37 CST 2020
pool-1-thread-3 End. Time = Sun Apr 12 11:14:42 CST 2020
pool-1-thread-4 End. Time = Sun Apr 12 11:14:42 CST 2020
pool-1-thread-1 End. Time = Sun Apr 12 11:14:42 CST 2020
pool-1-thread-5 End. Time = Sun Apr 12 11:14:42 CST 2020
pool-1-thread-1 Start. Time = Sun Apr 12 11:14:42 CST 2020
pool-1-thread-2 End. Time = Sun Apr 12 11:14:42 CST 2020
pool-1-thread-5 Start. Time = Sun Apr 12 11:14:42 CST 2020
pool-1-thread-4 Start. Time = Sun Apr 12 11:14:42 CST 2020
pool-1-thread-3 Start. Time = Sun Apr 12 11:14:42 CST 2020
pool-1-thread-2 Start. Time = Sun Apr 12 11:14:42 CST 2020
pool-1-thread-1 End. Time = Sun Apr 12 11:14:47 CST 2020
```

```
pool-1-thread-4 End. Time = Sun Apr 12 11:14:47 CST 2020
pool-1-thread-5 End. Time = Sun Apr 12 11:14:47 CST 2020
pool-1-thread-3 End. Time = Sun Apr 12 11:14:47 CST 2020
pool-1-thread-2 End. Time = Sun Apr 12 11:14:47 CST 2020
Finished all threads  // 任务全部执行完了才会跳出来，因为
executor.isTerminated()判断为true了才会跳出while循环，当且仅当调用
shutdown() 方法后，并且所有提交的任务完成后返回为 true
```

Analysis of thread pool principles

From the output of the previous code, we can see that **the thread pool will first execute 5 tasks. Then, when these tasks are completed, it will pick up new tasks to execute.** You can first analyze what is going on based on the above explanation. (Think about it for a while.)

Now, let's analyze the above output to briefly analyze the principle of thread pool.

In order to understand the principle of thread pool, we need to first analyze `execute` the method. In the sample code, we use `executor.execute(worker)` to submit a task to the thread pool.

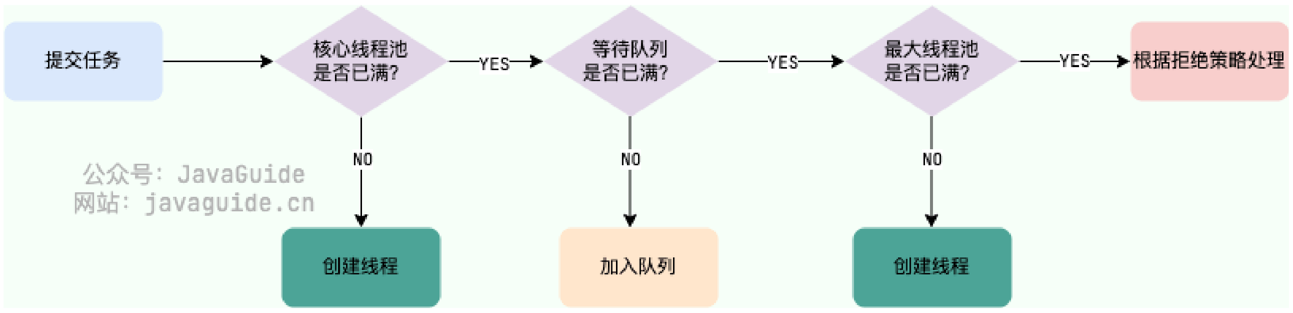
This method is very important. Let's take a look at its source code:

```java
   // 存放线程池的运行状态 (runState) 和线程池内有效线程的数量 (workerCount)
   private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));

    private static int workerCountOf(int c) {
        return c & CAPACITY;
    }
    //任务队列
    private final BlockingQueue<Runnable> workQueue;

    public void execute(Runnable command) {
        // 如果任务为null，则抛出异常。
        if (command == null)
            throw new NullPointerException();
        // ctl 中保存的线程池当前的一些状态信息
        int c = ctl.get();

        //  下面会涉及到 3 步 操作
        // 1.首先判断当前线程池中执行的任务数量是否小于 corePoolSize
        // 如果小于的话，通过addWorker(command, true)新建一个线程，并将任务(command)添加到该线程中；然后，启动该线程从而执行任务。
        if (workerCountOf(c) < corePoolSize) {
            if (addWorker(command, true))
                return;
            c = ctl.get();
        }
        // 2.如果当前执行的任务数量大于等于 corePoolSize 的时候就会走到这里，表明创建新的线程失败。
        // 通过 isRunning 方法判断线程池状态，线程池处于 RUNNING 状态并且队列可以加入任务，该任务才会被加入进去
        if (isRunning(c) && workQueue.offer(command)) {
            int recheck = ctl.get();
            // 再次获取线程池状态，如果线程池状态不是 RUNNING 状态就需要从任务队列中移除任务，并尝试判断线程是否全部执行完毕。同时执行拒绝策略。
            if (!isRunning(recheck) && remove(command))
                reject(command);
                // 如果当前工作线程数量为0，新创建一个线程并执行。
            else if (workerCountOf(recheck) == 0)
                addWorker(null, false);
        }
        //3. 通过addWorker(command, false)新建一个线程，并将任务(command)添加到该线程中；然后，启动该线程从而执行任务。
        // 传入 false 代表增加线程时判断当前线程数是否少于 maxPoolSize
        //如果addWorker(command, false)执行失败，则通过reject()执行相应的拒绝策略的内容。
        else if (!addWorker(command, false))
            reject(command);
    }
```

Here is a brief analysis of the entire process (the entire logic is simplified for easy understanding):

1. If the number of currently running threads is less than the number of core threads, a new thread will be created to execute the task.
2. If the number of currently running threads is equal to or greater than the number of core threads, but less than the maximum number of threads, then the task is placed in the task queue and waits for execution.
3. If the task fails to be put into the task queue (the task queue is full), but the numbe currently running threads is less than the maximum number of threads, a new thread is created to execute the task.

4. If the number of currently running threads is equal to the maximum number of threads, creating a new thread will cause the currently running threads to exceed the maximum number of threads, then the current task will be rejected and the rejection strategy will call `RejectedExecutionHandler.rejectedExecution()` the method.



In `execute` the method, call `addWorker` the method multiple times. `addWorker` This method is mainly used to create a new worker thread. If it returns true, it means that the worker thread is created and started successfully. Otherwise, it returns false.

```java
    // 全局锁，并发操作必备
    private final ReentrantLock mainLock = new ReentrantLock();
    // 跟踪线程池的最大大小，只有在持有全局锁mainLock的前提下才能访问此集合
    private int largestPoolSize;
    // 工作线程集合，存放线程池中所有的（活跃的）工作线程，只有在持有全局锁
mainLock的前提下才能访问此集合
    private final HashSet<Worker> workers = new HashSet<>();
    //获取线程池状态
    private static int runStateOf(int c)     { return c &
~CAPACITY; }
    //判断线程池的状态是否为 Running
    private static boolean isRunning(int c) {
        return c < SHUTDOWN;
    }


    /**
     * 添加新的工作线程到线程池
     * @param firstTask 要执行
     * @param core参数为true的话表示使用线程池的基本大小，为false使用线程池
最大大小
     * @return 添加成功就返回true否则返回false
     */
   private boolean addWorker(Runnable firstTask, boolean core) {
        retry:
        for (;;) {
            //这两句用来获取线程池的状态
            int c = ctl.get();
            int rs = runStateOf(c);

            // Check if queue empty only if necessary.
            if (rs >= SHUTDOWN &&
                ! (rs == SHUTDOWN &&
                   firstTask == null &&
                   ! workQueue.isEmpty()))
                return false;

            for (;;) {
               //获取线程池中工作的线程的数量
                int wc = workerCountOf(c);
                // core参数为false的话表明队列也满了，线程池大小变为
maximumPoolSize
                if (wc >= CAPACITY ||
                    wc >= (core ? corePoolSize : maximumPoolSize))
                    return false;
               //原子操作将workcount的数量加1
                if (compareAndIncrementWorkerCount(c))
                    break retry;
                // 如果线程的状态改变了就再次执行上述操作
                c = ctl.get();
                if (runStateOf(c) != rs)
                    continue retry;
                // else CAS failed due to workerCount change; retry
inner loop
            }
        }
        // 标记工作线程是否启动成功
        boolean workerStarted = false;
        // 标记工作线程是否创建成功
        boolean workerAdded = false;
        Worker w = null;
        try {

            w = new Worker(firstTask);
            final Thread t = w.thread;
            if (t != null) {
              // 加锁
                final ReentrantLock mainLock = this.mainLock;
                mainLock.lock();
        try {
          //获取线程池状态
          int rs = runStateOf(ctl.get());
         //rs < SHUTDOWN 如果线程池状态依然为RUNNING,并且线程的状态是存活的话，就会将工作线程添加到工作线程集合中
         //(rs=SHUTDOWN && firstTask == null)如果线程池状态小于STOP，也就是RUNNING或者SHUTDOWN状态下，同时传入的任务实例firstTask为null，则需要添加到工作线程集合和启动新的Worker
         // firstTask == null证明只新建线程而不执行任务
          if (rs < SHUTDOWN ||
            (rs == SHUTDOWN && firstTask == null)) {
            if (t.isAlive()) // precheck that t is startable
              throw new IllegalThreadStateException();
            workers.add(w);
           //更新当前工作线程的最大容量
            int s = workers.size();
            if (s > largestPoolSize)
              largestPoolSize = s;
           // 工作线程是否启动成功
            workerAdded = true;
          }
        } finally {
          // 释放锁
          mainLock.unlock();
        }
        //// 如果成功添加工作线程，则调用Worker内部的线程实例t的Thread#start()方法启动真实的线程实例
        if (workerAdded) {
          t.start();
         /// 标记线程启动成功
          workerStarted = true;
        }
      }
    } finally {
      // 线程启动失败，需要从工作线程中移除对应的Worker
      if (! workerStarted)
        addWorkerFailed(w);
    }
    return workerStarted;
  }
```

For more information about thread pool source code analysis, please refer to this article: Hard-core dry goods: 4W words from the source code to analyze the implementation principle of JUC thread pool ThreadPoolExecutor

Now, let's go back to the example code. Shouldn't it be easy to understand its principle now?

If you don’t understand, it doesn’t matter, you can take a look at my analysis:

> We simulated 10 tasks in our code, configured the number of core threads to be 5 and the waiting queue capacity to be 100. Therefore, only 5 tasks can be executed simultaneously at any one time, and the remaining 5 tasks are placed in the waiting queue. If any of the current 5 tasks is completed, the thread pool will pick up a new task to execute.

Several common comparisons

`Runnable` vs `Callable`

`Runnable` While interfaces have existed since Java 1.0 but `Callable` were introduced in Java 1.5 to handle `Runnable` unsupported use cases, `Runnable` they don't return results or throw checked exceptions. However, `Callable` interfaces can. Therefore, if a task doesn't need to return results or throw exceptions, it's recommended to use `Runnable` interfaces; this will result in cleaner code.

The tool class `Executors` can convert `Runnable` the object into `Callable` the object. (`Executors.callable(Runnable task)` or `Executors.callable(Runnable task, Object result)`).

`Runnable.java`

```java
@FunctionalInterface
public interface Runnable {
   /**
    * 被线程执行，没有返回值也无法抛出异常
    */
    public abstract void run();
}
```

`Callable.java`

```java
@FunctionalInterface
public interface Callable<V> {
    /**
     * 计算结果，或在无法这样做时抛出异常。
     * @return 计算得出的结果
     * @throws 如果无法计算结果，则抛出异常
     */
    V call() throws Exception;
}
```

`execute()` vs `submit()`

`execute()` There `submit()` are two ways to submit tasks to the thread pool, with some differences:

- **Return Value** : `execute()` This method is used to submit tasks that do not require a return value. It is typically used to execute `Runnable` tasks, where it is impossible to determine whether the task was successfully executed by the thread pool. The __register__ `submit()` method is used to submit tasks that require a return value. You can submit either __register__ `Runnable` or `Callable` __register__ tasks. This `submit()` method returns a __register__ object, which can be used to determine the success of the task and obtain the task's return value. (This method blocks the current thread until the task completes and has a timeout period; if the task is not completed within the __register__ timeout, an error message will be thrown .) `Future` `Future` `get()` `get (long timeout, TimeUnit unit)` `timeout` `java.util.concurrent.TimeoutException`
- **Exception handling** : `submit()` When using the method, you can use `Future` the object to handle exceptions thrown during task execution; when using `execute()` the method, exception handling needs to be handled by custom (setting objects to handle exceptions `ThreadFactory` when the thread factory creates a thread) or the method of `UncaughtExceptionHandler` `ThreadPoolExecutor` `afterExecute()`

Example 1: Using `get()` a method to get the return value.

```java
// 这里只是为了演示使用，推荐使用 `ThreadPoolExecutor` 构造方法来创建线程池。
ExecutorService executorService = Executors.newFixedThreadPool(3);

Future<String> submit = executorService.submit(() -> {
    try {
        Thread.sleep(5000L);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return "abc";
});

String s = submit.get();
System.out.println(s);
executorService.shutdown();
```

Output:

```plain
abc
```

Example 2: Using `get (long timeout, TimeUnit unit)` a method to get the return value.

```java
ExecutorService executorService = Executors.newFixedThreadPool(3);

Future<String> submit = executorService.submit(() -> {
    try {
        Thread.sleep(5000L);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return "abc";
});

String s = submit.get(3, TimeUnit.SECONDS);
System.out.println(s);
executorService.shutdown();
```

Output:

```plain
Exception in thread "main" java.util.concurrent.TimeoutException
  at java.util.concurrent.FutureTask.get(FutureTask.java:205)
```

`shutdown()` VS `shutdownNow()`

- **`shutdown ()`** : Close the thread pool, and the state of the thread pool becomes `SHUTDOWN` . The thread pool no longer accepts new tasks, but the tasks in the queue

must be completed.
- `shutdownNow ()` : Close the thread pool, and the thread pool status changes to `STOP` . The thread pool terminates the currently running task, stops processing queued tasks, and returns a list of tasks waiting to be executed.

`isTerminated()` VS `isShutdown()`

- `isShutDown` When the method is called `shutdown()` , it returns true.
- `isTerminated` When the method is called `shutdown()` and all submitted tasks are completed, it returns true

Several common built-in thread pools

FixedThreadPool

introduce

`FixedThreadPool` It is called a thread pool that can reuse a fixed number of threads. `Executors` Let's take a look at the relevant implementation through the relevant source code in the class:

```java
   /**
     * 创建一个可重用固定数量线程的线程池
     */
    public static ExecutorService newFixedThreadPool(int nThreads, ThreadFactory threadFactory) {
        return new ThreadPoolExecutor(nThreads, nThreads,
                                      0L, TimeUnit.MILLISECONDS,
                                      new LinkedBlockingQueue<Runnable>(),
                                      threadFactory);
    }
```

There is another `FixedThreadPool` implementation method, which is similar to the above, so I will not elaborate on it here:

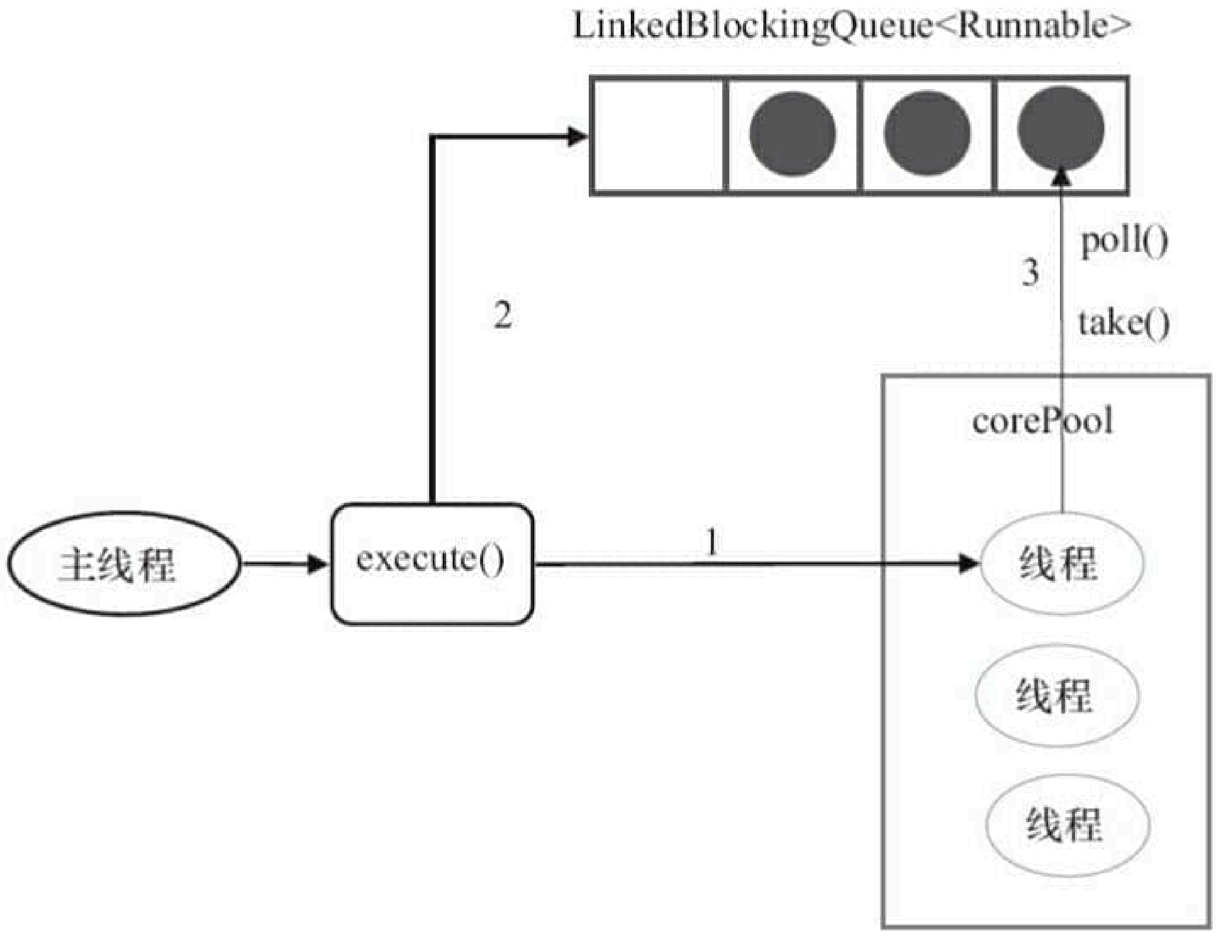
```java
    public static ExecutorService newFixedThreadPool(int nThreads) {
        return new ThreadPoolExecutor(nThreads, nThreads,
                                      0L, TimeUnit.MILLISECONDS,
                                      new LinkedBlockingQueue<Runnable>());
    }
```

From the source code above, we can see that the newly created `FixedThreadPool` and `corePoolSize` are `maximumPoolSize` set to `nThreads` , and this `nThreads` parameter is passed by ourselves when we use it.

Even if `maximumPoolSize` the value of is `corePoolSize` greater than , at most `corePoolSize` threads will be created. This is because `FixedThreadPool` uses an unbounded queue with a capacity `Integer.MAX_VALUE` of , which will never be full. `LinkedBlockingQueue`

Introduction to the task execution process

`FixedThreadPool` Method `execute()` running diagram (the picture comes from "The Art of Java Concurrency Programming"):



The above picture shows:

1. If the number of currently running threads is less than `corePoolSize` , if a new task comes, a new thread will be created to execute the task;
2. When the number of currently running threads is equal to `corePoolSize` , if a new task comes, it will be added `LinkedBlockingQueue` ;
3. `LinkedBlockingQueue` After the threads in the thread pool complete the task at hand, they will repeatedly obtain tasks from it in a loop to execute;

Why is it deprecated `FixedThreadPool` ?

`FixedThreadPool` Using an unbounded queue `LinkedBlockingQueue` (the queue capacity is Integer.MAX_VALUE) as the work queue of the thread pool will have the following effects on the thread pool:

1. When the number of threads in the thread pool reaches `corePoolSize` , new tasks will wait in the unbounded queue, so the number of threads in the thread pool will not exceed `corePoolSize` ;

2. When using an unbounded queue, `maximumPoolSize` will be an invalid parameter, as it is impossible for the task queue to be full. Therefore, from the created source code, we can see that `FixedThreadPool` the created and are set to the same value. `FixedThreadPool` `corePoolSize` `maximumPoolSize`
3. `keepAliveTime` Due to 1 and 2, it will be an invalid parameter when using an unbounded queue ;
4. Running `FixedThreadPool` (unexecuted `shutdown()` or `shutdownNow()`) tasks will not be rejected, which will cause OOM (memory overflow) when there are too many tasks.

SingleThreadExecutor

introduce

`SingleThreadExecutor` It is a thread pool with only one thread. Let's look at **the implementation of SingleThreadExecutor:**

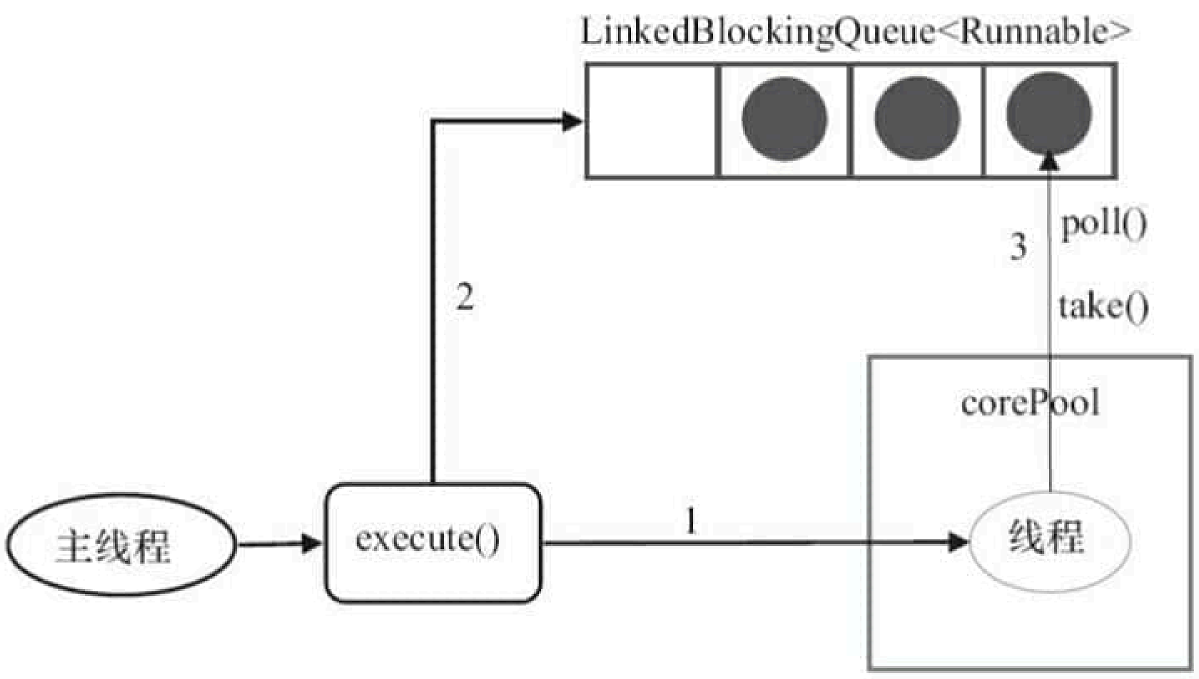
```java
   /**
     *返回只有一个线程的线程池
     */
    public static ExecutorService newSingleThreadExecutor(ThreadFactory threadFactory) {
        return new FinalizableDelegatedExecutorService
            (new ThreadPoolExecutor(1, 1,
                                    0L, TimeUnit.MILLISECONDS,
                                    new LinkedBlockingQueue<Runnable>(),
                                    threadFactory));
    }
```

```java
   public static ExecutorService newSingleThreadExecutor() {
        return new FinalizableDelegatedExecutorService
            (new ThreadPoolExecutor(1, 1,
                                    0L, TimeUnit.MILLISECONDS,
                                    new LinkedBlockingQueue<Runnable>()));
    }
```

From the source code above, we can see that the and `SingleThreadExecutor` of the newly created are both set to 1, and the other parameters are the same as . `corePoolSize` `maximumPoolSize` `FixedThreadPool`

Introduction to the task execution process

`SingleThreadExecutor` Schematic diagram of the operation (the picture comes from "The Art of Java Concurrency Programming"):



The above picture shows :

1. If the number of currently running threads is less than `corePoolSize` , a new thread is created to execute the task;
2. After there is a running thread in the current thread pool, add the task `LinkedBlockingQueue`
3. `LinkedBlockingQueue` After the thread finishes executing the current task, it will repeatedly obtain tasks from it in a loop to execute;

Why is it deprecated `SingleThreadExecutor` ?

`SingleThreadExecutor` Like , an unbounded queue with a capacity of is used as the thread pool's work queue. Using `FixedThreadPool` an unbounded queue as the thread pool's work queue has the same impact on the thread pool as . Simply put, it can cause

OOM
errors. `Integer.MAX_VALUE` `LinkedBlockingQueue` `SingleThreadExecutor` `FixedThreadPool`

CachedThreadPool

introduce

`CachedThreadPool` It is a thread pool that creates new threads as needed. Let's take a look at `CachedThreadPool` the implementation of through the source code:

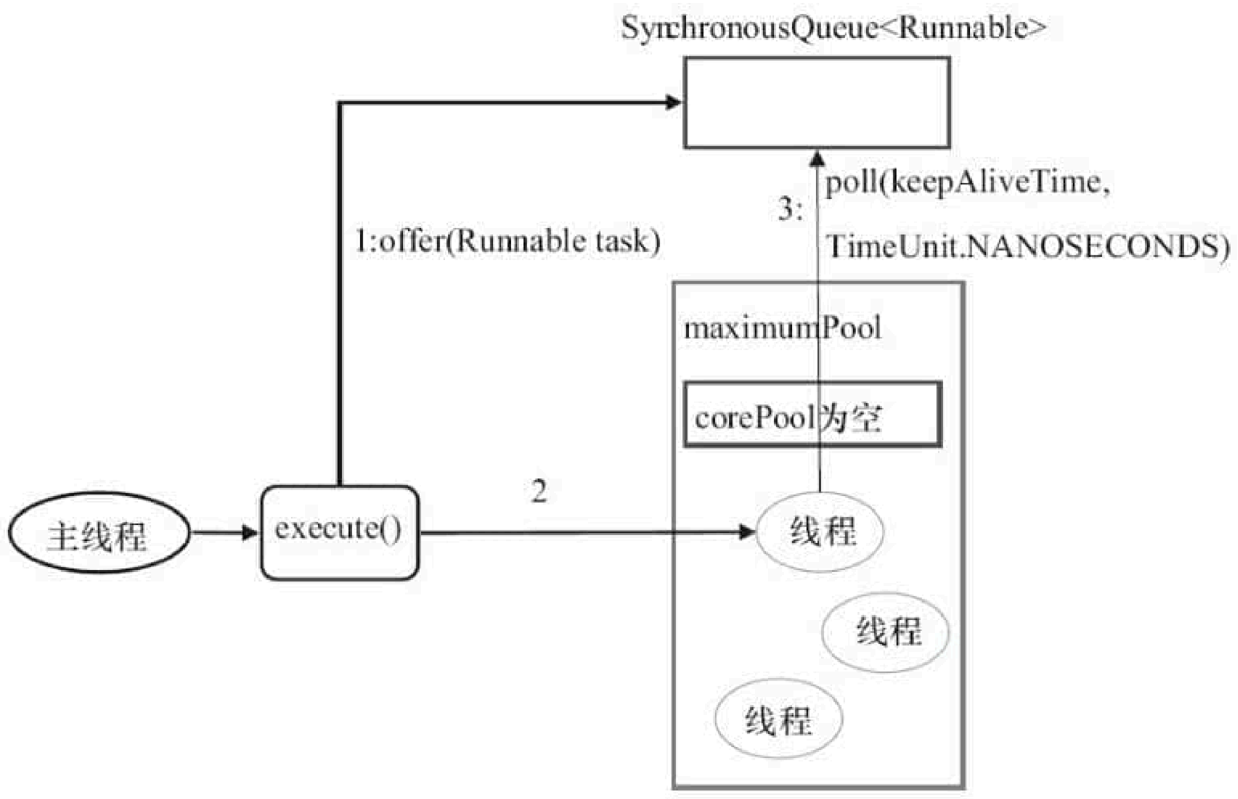
```java
    /**
     * 创建一个线程池，根据需要创建新线程，但会在先前构建的线程可用时重用它。
     */
    public static ExecutorService newCachedThreadPool(ThreadFactory threadFactory) {
        return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                      60L, TimeUnit.SECONDS,
                                      new SynchronousQueue<Runnable>(),
                                      threadFactory);
    }
```

```java
    public static ExecutorService newCachedThreadPool() {
        return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                      60L, TimeUnit.SECONDS,
                                      new SynchronousQueue<Runnable>());
    }
```

`CachedThreadPool` The `corePoolSize` is set to empty (0) and `maximumPoolSize` is set to `Integer.MAX.VALUE` , which means it is unbounded, which means that if the main thread submits tasks faster than `maximumPool` the threads in process tasks, `CachedThreadPool` new threads will be created continuously. In extreme cases, this will lead to exhaustion of CPU and memory resources.

Introduction to the task execution process

`CachedThreadPool` 's `execute()` method execution diagram (the picture comes from "The Art of Java Concurrency Programming"):



The above picture shows:

1. First, execute `SynchronousQueue.offer(Runnable task)` Submit Task to the Task Queue. If `maximumPool` an idle thread is currently executing `SynchronousQueue.poll(keepAliveTime,TimeUnit.NANOSECONDS)` , the main thread's offer operation `poll` is successfully paired with the idle thread's operation. The main thread then hands the task over to the idle thread, `execute()` completing the method execution. Otherwise, execute Step 2 below.
2. When the initial `maximumPool` is empty, or `maximumPool` there are no idle threads in , there will be no thread to execute `SynchronousQueue.poll(keepAliveTime,TimeUnit.NANOSECONDS)` . In this case, step 1 will fail, and `CachedThreadPool` a new thread will be created to execute the task, and the execute method will be executed.

Why is it deprecated `CachedThreadPool` ?

`CachedThreadPool` A synchronous queue is used `SynchronousQueue` , and the number of threads allowed to be created is `Integer.MAX_VALUE` , which may create a large number of threads and cause OOM.

ScheduledThreadPool

introduce

`ScheduledThreadPool` Used to run tasks after a given delay or periodically. This is rarely used in real projects and is not recommended. You only need to have a brief understanding of it.

```java
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) {
    return new ScheduledThreadPoolExecutor(corePoolSize);
}
public ScheduledThreadPoolExecutor(int corePoolSize) {
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
          new DelayedWorkQueue());
}
```

`ScheduledThreadPool` It is created by `ScheduledThreadPoolExecutor` using `DelayedWorkQueue` (delayed blocking queue) as the task queue of the thread pool.

`DelayedWorkQueue` The internal elements of the queue are not sorted by the time they were added, but rather by the length of time tasks have been delayed. The internal "heap" data structure ensures that each task dequeued is the one with the earliest execution time in the queue. `DelayedWorkQueue` When the queue is full, it automatically expands to 1/2 of its original capacity, meaning it never blocks and can `Integer.MAX_VALUE` only create as many threads as the number of core threads.

`ScheduledThreadPoolExecutor` Inherited `ThreadPoolExecutor` , so the creation `ScheduledThreadExecutor` is essentially to create a `ThreadPoolExecutor` thread pool, but the parameters passed in are different.

```java
public class ScheduledThreadPoolExecutor
        extends ThreadPoolExecutor
        implements ScheduledExecutorService
```

Comparison between ScheduledThreadPoolExecutor and Timer

- `Timer` sensitive to changes in the system clock, `ScheduledThreadPoolExecutor` no;

- `Timer` There is only one thread of execution, so long-running tasks can delay other tasks. `ScheduledThreadPoolExecutor` Any number of threads can be configured. In addition, `ThreadFactory` you can have full control over the threads created if you want (by providing);
- `TimerTask` A runtime exception thrown in will kill a thread, resulting in a `Timer` panic, meaning that scheduled tasks will no longer run. `ScheduledThreadExecutor` Not only does catch runtime exceptions, but it also allows you to handle them if needed (by overriding `afterExecute` the method `ThreadPoolExecutor`). The task that threw the exception will be canceled, but other tasks will continue to run.

For a detailed introduction to scheduled tasks, see this article: Detailed Explanation of Java Scheduled Tasks .

Thread Pool Best Practices

This article, Java Thread Pool Best Practices, summarizes some things you should pay attention to when using thread pools. You can read it before using thread pools in actual projects.

refer to

- The Art of Concurrent Programming in Java
- Java Scheduler ScheduledExecutorService ScheduledThreadPoolExecutor Example
- java.util.concurrent.ScheduledThreadPoolExecutor Example
- ThreadPoolExecutor – Java Thread Pool Example

Recently Updated2025/4/11 06:23

Contributors: Kou Shuang , wjch , cxyzjp , Jiabin , HTY , shuang.kou , liwenguang , jiayao , guide , LIU , zhuhao , vfmh , kaka2634 , errorog , Mr.HangdianGhost , Evan He , Verne.Chung , Li Jianxin , hyl1995 , Guide , Mr.Hope , paigeman , WangHuaming111 , smy1999