compact1, compact2, compact3

java.util.concurrent

Class ThreadPoolExecutor

java.lang.Object
    java.util.concurrent.AbstractExecutorService
        java.util.concurrent.ThreadPoolExecutor

All Implemented Interfaces:

Executor, ExecutorService

Direct Known Subclasses:

ScheduledThreadPoolExecutor

```
public class ThreadPoolExecutor
extends AbstractExecutorService
```

An ExecutorService that executes each submitted task using one of possibly several pooled threads, normally configured using Executors factory methods.

Thread pools address two different problems: they usually provide improved performance when executing large numbers of asynchronous tasks, due to reduced per-task invocation overhead, and they provide a means of bounding and managing the resources, including threads, consumed when executing a collection of tasks. Each ThreadPoolExecutor also maintains some basic statistics, such as the number of completed tasks.

To be useful across a wide range of contexts, this class provides many adjustable parameters and extensibility hooks. However, programmers are urged to use the more convenient Executors factory methods Executors.newCachedThreadPool() (unbounded thread pool, with automatic thread reclamation), Executors.newFixedThreadPool(int) (fixed size thread pool) and Executors.newSingleThreadExecutor() (single background thread), that preconfigure settings for the most common usage scenarios. Otherwise, use the following guide when manually configuring and tuning this class:

Core and maximum pool sizes

A ThreadPoolExecutor will automatically adjust the pool size (see getPoolSize()) according to the bounds set by corePoolSize (see getCorePoolSize()) and maximumPoolSize (see getMaximumPoolSize()). When a new task is submitted in method execute(Runnable), and fewer than corePoolSize threads are running, a new thread is created to handle the request, even if other worker threads are idle. If there are more than corePoolSize but less than maximumPoolSize threads running, a new thread will be created only if the queue is full. By setting corePoolSize and maximumPoolSize the same, you create a fixed-size thread pool. By setting maximumPoolSize to an essentially unbounded value such as Integer.MAX_VALUE, you allow the pool to accommodate an arbitrary number of concurrent tasks. Most typically, core and maximum pool sizes are set only upon construction, but they may also be changed dynamically using setCorePoolSize(int) and setMaximumPoolSize(int).

On-demand construction

By default, even core threads are initially created and started only when new tasks arrive, but this can be overridden dynamically using method prestartCoreThread() or prestartAllCoreThreads(). You probably want to prestart threads if you construct the pool with a non-empty queue.

Creating new threads

New threads are created using a ThreadFactory. If not otherwise specified, a Executors.defaultThreadFactory() is used, that creates threads to all be in the same ThreadGroup and with the same NORM_PRIORITY priority and non-daemon status. By supplying a different ThreadFactory, you can alter the thread's name, thread group, priority, daemon status, etc. If a ThreadFactory fails to create a thread when asked by returning null from newThread, the executor will continue, but might not be able to execute any tasks. Threads should possess the "modifyThread" RuntimePermission. If worker threads or other threads using the pool do not possess this permission, service may be degraded: configuration changes may not take effect in a timely manner, and a shutdown pool may remain in a state in which termination is possible but not completed.

Keep-alive times

If the pool currently has more than corePoolSize threads, excess threads will be terminated if they have been idle for more than the keepAliveTime (see getKeepAliveTime(TimeUnit)). This provides a means of reducing resource consumption when the pool is not being actively used. If the pool becomes more active later, new threads will be constructed. This parameter can also be changed dynamically using method setKeepAliveTime(long, TimeUnit). Using a value of Long.MAX_VALUE TimeUnit.NANOSECONDS effectively disables idle threads from ever terminating prior to shut down. By default, the keep-alive policy applies only when there are more than corePoolSize threads. But method allowCoreThreadTimeOut(boolean) can be used to apply this time-out policy to core threads as well, so long as the keepAliveTime value is non-zero.

Queuing

Any BlockingQueue may be used to transfer and hold submitted tasks. The use of this queue interacts with pool sizing:

- If fewer than corePoolSize threads are running, the Executor always prefers adding a new thread rather than queuing.
- If corePoolSize or more threads are running, the Executor always prefers queuing a request rather than adding a new thread.
- If a request cannot be queued, a new thread is created unless this would exceed maximumPoolSize, in which case, the task will be rejected.

There are three general strategies for queuing:

1. *Direct handoffs*. A good default choice for a work queue is a SynchronousQueue that hands off tasks to threads without otherwise holding them. Here, an attempt to queue a task will fail if no threads are immediately available to run it, so a new thread will be constructed. This policy avoids lockups when handling sets of requests that might have internal dependencies. Direct handoffs generally require unbounded maximumPoolSizes to avoid rejection of new submitted tasks. This in turn admits the possibility of unbounded thread growth when commands continue to arrive on average faster than they can be processed.
2. *Unbounded queues*. Using an unbounded queue (for example a LinkedBlockingQueue without a predefined capacity) will cause new tasks to wait in the queue when all corePoolSize threads are busy. Thus, no more than corePoolSize threads will ever be created. (And the value of the maximumPoolSize therefore doesn't have any effect.) This may be appropriate when each task is completely independent of others, so tasks cannot affect each others execution; for example, in a web page server. While this style of queuing can be useful in smoothing out transient bursts of requests, it admits the possibility of unbounded work queue growth when commands continue to arrive on average faster than they can be processed.
3. *Bounded queues*. A bounded queue (for example, an ArrayBlockingQueue) helps prevent resource exhaustion when used with finite maximumPoolSizes, but can be more difficult to tune and control. Queue sizes and maximum pool sizes may be traded off for each other: Using large queues and small pools minimizes CPU usage, OS resources, and context-switching overhead, but can lead to artificially low throughput. If tasks frequently block (for example if they are

I/O bound), a system may be able to schedule time for more threads than you otherwise allow. Use of small queues generally requires larger pool sizes, which keeps CPUs busier but may encounter unacceptable scheduling overhead, which also decreases throughput.

Rejected tasks

New tasks submitted in method execute(Runnable) will be *rejected* when the Executor has been shut down, and also when the Executor uses finite bounds for both maximum threads and work queue capacity, and is saturated. In either case, the execute method invokes the RejectedExecutionHandler.rejectedExecution(Runnable, ThreadPoolExecutor) method of its RejectedExecutionHandler. Four predefined handler policies are provided:

1. In the default ThreadPoolExecutor.AbortPolicy, the handler throws a runtime RejectedExecutionException upon rejection.
2. In ThreadPoolExecutor.CallerRunsPolicy, the thread that invokes execute itself runs the task. This provides a simple feedback control mechanism that will slow down the rate that new tasks are submitted.
3. In ThreadPoolExecutor.DiscardPolicy, a task that cannot be executed is simply dropped.
4. In ThreadPoolExecutor.DiscardOldestPolicy, if the executor is not shut down, the task at the head of the work queue is dropped, and then execution is retried (which can fail again, causing this to be repeated.)

It is possible to define and use other kinds of RejectedExecutionHandler classes. Doing so requires some care especially when policies are designed to work only under particular capacity or queuing policies.

Hook methods 自定义扩展

This class provides protected overridable beforeExecute(Thread, Runnable) and afterExecute(Runnable, Throwable) methods that are called before and after execution of each task. These can be used to manipulate the execution environment; for example, reinitializing ThreadLocals, gathering statistics, or adding log entries. Additionally, method terminated() can be overridden to perform any special processing that needs to be done once the Executor has fully terminated.

If hook or callback methods throw exceptions, internal worker threads may in turn fail and abruptly terminate.

Queue maintenance

Method getQueue() allows access to the work queue for purposes of monitoring and debugging. Use of this method for any other purpose is strongly discouraged. Two supplied methods, remove(Runnable) and purge() are available to assist in storage reclamation when large numbers of queued tasks become cancelled.

Finalization

A pool that is no longer referenced in a program *AND* has no remaining threads will be shutdown automatically. If you would like to ensure that unreferenced pools are reclaimed even if users forget to call shutdown(), then you must arrange that unused threads eventually die, by setting appropriate keep-alive times, using a lower bound of zero core threads and/or setting allowCoreThreadTimeOut(boolean).

Extension example. Most extensions of this class override one or more of the protected hook methods. For example, here is a subclass that adds a simple pause/resume feature:

```
 class PausableThreadPoolExecutor extends ThreadPoolExecutor {
   private boolean isPaused;
   private ReentrantLock pauseLock = new ReentrantLock();
   private Condition unpaused = pauseLock.newCondition();

   public PausableThreadPoolExecutor(...) { super(...); }
```

```
  protected void beforeExecute(Thread t, Runnable r) {
    super.beforeExecute(t, r);
    pauseLock.lock();
    try {
      while (isPaused) unpaused.await();
    } catch (InterruptedException ie) {
      t.interrupt();
    } finally {
      pauseLock.unlock();
    }
  }

  public void pause() {
    pauseLock.lock();
    try {
      isPaused = true;
    } finally {
      pauseLock.unlock();
    }
  }

  public void resume() {
    pauseLock.lock();
    try {
      isPaused = false;
      unpaused.signalAll();
    } finally {
      pauseLock.unlock();
    }
  }
}
```

Since:

1.5

Nested Class Summary

Nested Classes

Modifier and Type	Class and Description
static class	**ThreadPoolExecutor.AbortPolicy** A handler for rejected tasks that throws a `RejectedExecutionException`.
static class	**ThreadPoolExecutor.CallerRunsPolicy** A handler for rejected tasks that runs the rejected task directly in the calling thread of the `execute` method, unless the executor has been shut down, in which case the task is discarded.
static class	**ThreadPoolExecutor.DiscardOldestPolicy** A handler for rejected tasks that discards the oldest unhandled request and then retries `execute`, unless the executor is shut down, in which case the task is discarded.
static class	**ThreadPoolExecutor.DiscardPolicy** A handler for rejected tasks that silently discards the rejected task.

Constructor Summary

Constructors

Constructor and Description
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.
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.
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.
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.

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
protected void	**afterExecute**(**Runnable** r, **Throwable** t) Method invoked upon completion of execution of the given Runnable.
void	**allowCoreThreadTimeOut**(boolean value) Sets the policy governing whether core threads may time out and terminate if no tasks arrive within the keep-alive time, being replaced if needed when new tasks arrive.
boolean	**allowsCoreThreadTimeOut**() Returns true if this pool allows core threads to time out and terminate if no tasks arrive within the keepAlive time, being replaced if needed when new tasks arrive.
boolean	**awaitTermination**(long timeout, **TimeUnit** unit) Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.
protected void	**beforeExecute**(**Thread** t, **Runnable** r) Method invoked prior to executing the given Runnable in the given thread.
void	**execute**(**Runnable** command) Executes the given task sometime in the future.
protected void	**finalize**()

	Invokes `shutdown` when this executor is no longer referenced and it has no threads.
`int`	**`getActiveCount`**`()` Returns the approximate number of threads that are actively executing tasks.
`long`	**`getCompletedTaskCount`**`()` Returns the approximate total number of tasks that have completed execution.
`int`	**`getCorePoolSize`**`()` Returns the core number of threads.
`long`	**`getKeepAliveTime`**`(`**`TimeUnit`**` unit)` Returns the thread keep-alive time, which is the amount of time that threads in excess of the core pool size may remain idle before being terminated.
`int`	**`getLargestPoolSize`**`()` Returns the largest number of threads that have ever simultaneously been in the pool.
`int`	**`getMaximumPoolSize`**`()` Returns the maximum allowed number of threads.
`int`	**`getPoolSize`**`()` Returns the current number of threads in the pool.
`BlockingQueue``<`**`Runnable`**`>`	**`getQueue`**`()` Returns the task queue used by this executor.
`RejectedExecutionHandler`	**`getRejectedExecutionHandler`**`()` Returns the current handler for unexecutable tasks.
`long`	**`getTaskCount`**`()` Returns the approximate total number of tasks that have ever been scheduled for execution.
`ThreadFactory`	**`getThreadFactory`**`()` Returns the thread factory used to create new threads.
`boolean`	**`isShutdown`**`()` Returns `true` if this executor has been shut down.
`boolean`	**`isTerminated`**`()` Returns `true` if all tasks have completed following shut down.
`boolean`	**`isTerminating`**`()` Returns true if this executor is in the process of terminating after **`shutdown()`** or **`shutdownNow()`** but has not completely terminated.
`int`	**`prestartAllCoreThreads`**`()` Starts all core threads, causing them to idly wait for work.
`boolean`	**`prestartCoreThread`**`()` Starts a core thread, causing it to idly wait for work.
`void`	**`purge`**`()` Tries to remove from the work queue all **`Future`** tasks that have been cancelled.

boolean	**remove**(**Runnable** task) Removes this task from the executor's internal queue if it is present, thus causing it not to be run if it has not already started.
void	**setCorePoolSize**(int corePoolSize) Sets the core number of threads.
void	**setKeepAliveTime**(long time, **TimeUnit** unit) Sets the time limit for which threads may remain idle before being terminated.
void	**setMaximumPoolSize**(int maximumPoolSize) Sets the maximum allowed number of threads.
void	**setRejectedExecutionHandler**(**RejectedExecutionHandler** handler) Sets a new handler for unexecutable tasks.
void	**setThreadFactory**(**ThreadFactory** threadFactory) Sets the thread factory used to create new threads.
void	**shutdown**() Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.
List<**Runnable**>	**shutdownNow**() Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution.
protected void	**terminated**() Method invoked when the Executor has terminated.
String	**toString**() Returns a string identifying this pool, as well as its state, including indications of run state and estimated worker and task counts.

Methods inherited from class java.util.concurrent.AbstractExecutorService

invokeAll, invokeAll, invokeAny, invokeAny, newTaskFor, newTaskFor, submit, submit, submit

Methods inherited from class java.lang.Object

clone, equals, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

ThreadPoolExecutor

```
public 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. It may be more convenient to use one of the `Executors`