ForkJoinPool

# ForkJoinPool介绍

## 继承关系介绍

存在于java.util.concurrent包中。

public class **ForkJoinPool** extends **AbstractExecutorService**

All Implemented Interfaces: Executor, ExecutorService



Since: 1.7

## 功能介绍

An **ExecutorService** for running **ForkJoinTasks**. A **ForkJoinPool** provides the entry point for submissions from **non-ForkJoinTask clients**, as well as management and monitoring operations.

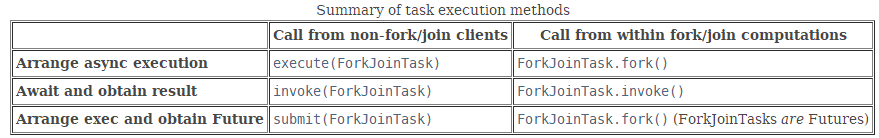
A **ForkJoinPool** differs from other kinds of **ExecutorService** mainly by virtue of employing work-stealing: all threads in the pool attempt to find and execute tasks submitted to the pool and/or created by other active tasks (eventually blocking waiting for work if none exist). This enables efficient processing when most tasks spawn other subtasks (as do most ForkJoinTasks), as well as when many small tasks are submitted to the pool from external clients. Especially when setting **asyncMode** to true in constructors, **ForkJoinPools** may also be appropriate for use with event-style tasks that are never joined.

A static **commonPool**() is available and appropriate for most applications. The common pool is used by any **ForkJoinTask** that is not explicitly submitted to a specified pool. Using the common pool normally reduces resource usage (its threads are slowly reclaimed during periods of non-use, and reinstated upon subsequent use).

For applications that require separate or custom pools, a **ForkJoinPool** may be constructed with a given target parallelism level; by default, equal to the number of available processors. The pool attempts to maintain enough active (or available) threads by dynamically adding, suspending, or resuming internal worker threads, even if some tasks are stalled waiting to join others. However, no such adjustments are guaranteed in the face of **blocked I/O** or other unmanaged synchronization. The nested **ForkJoinPool.ManagedBlocker** interface enables extension of the kinds of synchronization accommodated.

In addition to execution and lifecycle control methods, this class provides status check methods (for example ***getStealCount***()) that are intended to aid in developing, tuning, and monitoring fork/join applications. Also, method toString() returns indications of pool state in a convenient form for informal monitoring.

As is the case with other **ExecutorServices**, there are three main task execution methods summarized in the following table. These are designed to be used primarily by clients not already engaged in **fork/join** computations in the current pool. The main forms of these methods accept instances of **ForkJoinTask**, but overloaded forms also allow mixed execution of plain Runnable- or Callable- based activities as well. However, tasks that are already executing in a pool should normally instead use the within-computation forms listed in the table unless using async event-style tasks that are not usually joined, in which case there is little difference among choice of methods.



The common pool is by default constructed *with default parameters*, but these may be controlled by setting three system properties:

* java.util.concurrent.ForkJoinPool.common.*parallelism* - the parallelism level, a non-negative integer
* java.util.concurrent.ForkJoinPool.common.*threadFactory* - the class name of a ForkJoinPool.ForkJoinWorkerThreadFactory
* java.util.concurrent.ForkJoinPool.common.*exceptionHandler* - the class name of a Thread.UncaughtExceptionHandler

If a SecurityManager is present and no factory is specified, then the default pool uses a factory supplying threads that have no Permissions enabled. *The system class loader is used to load these classes*. Upon any error in establishing these settings, default parameters are used. It is possible to disable or limit the use of threads in the common pool by setting the parallelism property to zero, and/or using a factory that may return null. However doing so may cause unjoined tasks to never be executed.

Implementation notes: *This implementation restricts the maximum number of running threads to* ***32767***. Attempts to create pools with greater than the maximum number result in **IllegalArgumentException**.

This implementation rejects submitted tasks (that is, by throwing RejectedExecutionException) only when the pool is shut down or internal resources have been exhausted.

## 构造方法

4个参数，可以设定也可以采用默认的。

ForkJoinPool()

Creates a ForkJoinPool with parallelism equal to *Runtime.availableProcessors()*, using the default thread factory, no UncaughtExceptionHandler, and non-async LIFO processing mode.

ForkJoinPool(int parallelism)

Creates a ForkJoinPool with **the indicated parallelism level**, the default thread factory, no UncaughtExceptionHandler, and non-async LIFO processing mode.

ForkJoinPool(int parallelism, ForkJoinPool.ForkJoinWorkerThreadFactory factory, Thread.UncaughtExceptionHandler handler, boolean asyncMode)

Creates a ForkJoinPool with the given parameters.

## 内部接口

static interface **ForkJoinPool.ForkJoinWorkerThreadFactory**

Factory for creating new *ForkJoinWorkerThreads*.

static interface *ForkJoinPool.ManagedBlocker*

Interface for extending managed parallelism for tasks running in ForkJoinPools.

## Field：defaultForkJoinWorkerThreadFactory

static ForkJoinPool.ForkJoinWorkerThreadFactory **defaultForkJoinWorkerThreadFactory**

Creates a new **ForkJoinWorkerThread**.

## 重要方法介绍

### 静态方法：获取commonPool

static **ForkJoinPool** **commonPool**()

Returns the common pool instance.

static int **getCommonPoolParallelism**()

Returns the targeted parallelism level of the common pool.

### execute:执行任务

void **execute**(ForkJoinTask<?> task)

Arranges for (asynchronous) execution of the given task.

void **execute**(Runnable task)

Executes the given command at some time in the future.

### invoke

<T> T **invoke**(ForkJoinTask<T> task)

Performs the given task, returning its result upon completion.

<T> List<Future<T>> **invokeAll**(Collection<? extends **Callable**<T>> tasks)

Executes the given tasks, returning a list of Futures holding their status and results when all complete.

### submit

<T> ForkJoinTask<T> submit(Callable<T> task)

Submits a value-returning task for execution and returns a Future representing the pending results of the task.

<T> ForkJoinTask<T> submit(ForkJoinTask<T> task)

Submits a ForkJoinTask for execution.

ForkJoinTask<?> submit(Runnable task)

Submits a Runnable task for execution and returns a Future representing that task.

<T> ForkJoinTask<T> submit(Runnable task, T result)

Submits a Runnable task for execution and returns a Future representing that task.

### getXxx

get构造函数的那些参数；

int **getActiveThreadCount**()

ForkJoinPool.ForkJoinWorkerThreadFactory getFactory()

Returns the factory used for constructing new workers.

int getPoolSize()

Returns the number of worker threads that have started but not yet terminated.

int **getQueuedSubmissionCount**()

Returns an estimate of the number of tasks submitted to this pool that have not yet begun executing.

long **getQueuedTaskCount**()

Returns an estimate of the total number of tasks currently held in queues by worker threads (but not including tasks submitted to the pool that have not begun executing).

int **getRunningThreadCount**()

Returns an estimate of the number of worker threads that are not blocked waiting to join tasks or for other managed synchronization.

long **getStealCount**()

Returns an estimate of the total number of tasks stolen from one thread's work queue by another.

### isXxx

isTerminated、isTerminating、isShutdown、isQuiescent()

# ForkJoinTask<V>

## 继承关系

public abstract class **ForkJoinTask<V>** extends Object

implements **Future<V>,** Serializable



All Implemented Interfaces: Serializable, **Future<V>**

Direct Known Subclasses: CountedCompleter, RecursiveAction, RecursiveTask

## 功能介绍

Abstract base class for tasks that run within a **ForkJoinPool**. A ForkJoinTask is a thread-like entity that is much lighter weight than a normal thread. Huge numbers of tasks and subtasks may be hosted by a small number of actual threads in a **ForkJoinPool**, at the price of some usage limitations.

A "main" **ForkJoinTask** begins execution when it is explicitly submitted to a ForkJoinPool, or, if not already engaged in a ForkJoin computation, commenced in the **ForkJoinPool.commonPool()** via fork(), invoke(), or related methods. Once started, it will usually in turn start other subtasks. As indicated by the name of this class, many programs using ForkJoinTask employ only methods **fork() and join(),** or derivatives such as **invokeAll**. However, this class also provides a number of other methods that can come into play in advanced usages, as well as extension mechanics that allow support of new forms of **fork/join processing**.

***A ForkJoinTask is a lightweight form of Future.*** The efficiency of ForkJoinTasks stems from a set of restrictions (that are only partially statically enforceable) reflecting their main use as computational tasks calculating pure functions or operating on purely isolated objects. The primary coordination mechanisms are fork(), that arranges asynchronous execution, and join(), that doesn't proceed until the task's result has been computed. Computations should ideally avoid synchronized methods or blocks, and should minimize other blocking synchronization apart from joining other tasks or using synchronizers such as Phasers that are advertised to cooperate with **fork/join scheduling**. Subdividable tasks should also not perform **blocking I/O**, and should ideally access variables that are completely independent of those accessed by other running tasks. These guidelines are loosely enforced by not permitting checked exceptions such as IOExceptions to be thrown. However, computations may still encounter *unchecked exceptions*, that are rethrown to callers attempting to join them. These exceptions may additionally include RejectedExecutionException stemming from internal resource exhaustion, such as failure to allocate internal task queues. Rethrown exceptions behave in the same way as regular exceptions, but, when possible, contain stack traces (as displayed for example using ex.printStackTrace()) of both the thread that initiated the computation as well as the thread actually encountering the exception; minimally only the latter.

It is possible to define and use **ForkJoinTasks** that may block, but doing do requires three further considerations: (1) Completion of few if any other tasks should be dependent on a task that blocks on external synchronization or I/O. Event-style async tasks that are never joined (for example, those subclassing CountedCompleter) often fall into this category. (2) To minimize resource impact, tasks should be small; ideally performing only the (possibly) blocking action. (3) Unless the ForkJoinPool.ManagedBlocker API is used, or the number of possibly blocked tasks is known to be less than the pool's ForkJoinPool.getParallelism() level, the pool cannot guarantee that enough threads will be available to ensure progress or good performance.

The primary method for awaiting completion and extracting results of a task is **join(),** but there are several variants: The Future.get() methods support interruptible and/or timed waits for completion and report results using **Future** conventions. Method **invoke**() is semantically equivalent to **fork();** join() but always attempts to begin execution in the current thread. The "quiet" forms of these methods do not extract results or report exceptions. These may be useful when a set of tasks are being executed, and you need to delay processing of results or exceptions until all complete. Method invokeAll (available in multiple versions) performs the most common form of parallel invocation: forking a set of tasks and joining them all.

In the most typical usages, a fork-join pair act like a call (fork) and return (join) from a parallel recursive function. As is the case with other forms of recursive calls, returns (joins) should be performed **innermost-first**. For example, a.fork(); b.fork(); b.join(); a.join(); is likely to be substantially more efficient than joining a before b.

The execution status of tasks may be queried at several levels of detail**: isDone()** is true if a task completed in any way (including the case where a task was cancelled without executing); **isCompletedNormally()** is true if a task completed without cancellation or encountering an exception; **isCancelled()** is true if the task was cancelled (in which case getException() returns a CancellationException); and **isCompletedAbnormally()** is true if a task was either cancelled or encountered an exception, in which case **getException()** will return either the encountered exception or CancellationException.

The **ForkJoinTask** class is not usually directly subclassed. Instead, you subclass one of the abstract classes that support a particular style of fork/join processing, typically RecursiveAction for most computations that do not return results, RecursiveTask for those that do, and CountedCompleter for those in which completed actions trigger other actions. Normally, a concrete ForkJoinTask subclass declares fields comprising its parameters, established in a constructor, and then defines a compute method that somehow uses the control methods supplied by this base class.

Method **join()** and its variants are appropriate for use only when completion dependencies are acyclic; that is, the parallel computation can be described as a directed acyclic graph (DAG). Otherwise, executions may encounter a form of deadlock as tasks cyclically wait for each other. However, this framework supports other methods and techniques (for example the use of Phaser, helpQuiesce(), and complete(V)) that may be of use in constructing custom subclasses for problems that are not statically structured as DAGs. To support such usages, a ForkJoinTask may be atomically tagged with a short value using setForkJoinTaskTag(short) or compareAndSetForkJoinTaskTag(short, short) and checked using getForkJoinTaskTag(). The ForkJoinTask implementation does not use these protected methods or tags for any purpose, but they may be of use in the construction of specialized subclasses. For example, parallel graph traversals can use the supplied methods to avoid revisiting nodes/tasks that have already been processed. (Method names for tagging are bulky in part to encourage definition of methods that reflect their usage patterns.)

Most base support methods are final, to prevent overriding of implementations that are intrinsically tied to the underlying lightweight task scheduling framework. Developers creating new basic styles of fork/join processing should minimally implement protected methods exec(), setRawResult(V), and getRawResult(), while also introducing an abstract computational method that can be implemented in its subclasses, possibly relying on other protected methods provided by this class.

**ForkJoinTasks** should perform relatively small amounts of computation. Large tasks should be split into smaller subtasks, usually via recursive decomposition. As a very rough rule of thumb, a task should perform more than 100 and less than 10000 basic computational steps, and should avoid *indefinite looping*. If tasks are too big, then parallelism cannot improve throughput. If too small, then memory and internal task maintenance overhead may overwhelm processing.

This class provides **adapt** methods for Runnable and Callable, that may be of use when mixing execution of **ForkJoinTasks** with other kinds of tasks. When all tasks are of this form, consider using a pool constructed in asyncMode.

**ForkJoinTasks** are Serializable, which enables them to be used in extensions such as remote execution frameworks. It is sensible to serialize tasks only before or after, but not during, execution. Serialization is not relied on during execution itself.

## 构造方法

public ForkJoinTask()

## 重要方法

### adapt

static <T> ForkJoinTask<T> adapt(Callable<? extends T> callable)

Returns a new **ForkJoinTask** that performs the call method of the given Callable as its action, and returns its **result** upon join(), translating any checked exceptions encountered into RuntimeException.

static ForkJoinTask<?> adapt(Runnable runnable)

Returns a new ForkJoinTask that performs the run method of the given Runnable as its action, and returns a null result upon join().

static <T> ForkJoinTask<T> adapt(Runnable runnable, T result)

Returns a new ForkJoinTask that performs the run method of the given Runnable as its action, and returns the given result upon join().

### fork()

ForkJoinTask<V> fork()

Arranges to asynchronously execute this task in the pool the current task is running in, if applicable, or using the **ForkJoinPool.commonPool()** if not inForkJoinPool().

### join()

V join()

Returns the result of the computation when it is done.

void quietlyJoin()

Joins this task, without returning its result or throwing its exception.

### invoke

V **invoke**()

Commences performing this task, awaits its completion if necessary, and returns its result, or throws an (unchecked) RuntimeException or Error if the underlying computation did so.

static <T extends ForkJoinTask<?>>Collection<T> **invokeAll**(Collection<T> tasks)

Forks all tasks in the specified collection, returning when isDone holds for each task or an (unchecked) exception is encountered, in which case the exception is rethrown.

static void **invokeAll**(ForkJoinTask<?>... tasks)

Forks the given tasks, returning when isDone holds for each task or an (unchecked) exception is encountered, in which case the exception is rethrown.

static void **invokeAll**(ForkJoinTask<?> t1, ForkJoinTask<?> t2)

**Forks the given tasks**, returning when isDone holds for each task or an (unchecked) exception is encountered, in which case the exception is rethrown.

void quietlyInvoke()

Commences performing this task and awaits its completion if necessary, without returning its result or throwing its exception.

# RecursiveTask<V>

## RecursiveTask<V>



public abstract class **RecursiveTask<V>** extends **ForkJoinTask<V>**

Since: 1.7

## 功能介绍

A recursive result-bearing **ForkJoinTask**.

For a classic example, here is a task computing Fibonacci numbers:

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, 144, 233，377，610

class **Fibonacci** extends RecursiveTask<Integer> {

final int n;

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

Integer compute() {

if (n <= 1)

return n;

Fibonacci f1 = new Fibonacci(n - 1);

f1.fork();

Fibonacci f2 = new Fibonacci(n - 2);

return f2.compute() + f1.join();

}

}

However, besides being a dumb way to compute **Fibonacci** functions (there is a simple fast linear algorithm that you'd use in practice), this is likely to perform poorly because the smallest subtasks are too small to be worthwhile splitting up. Instead, as is the case for nearly all **fork/join** applications, you'd pick some minimum granularity size (for example 10 here) for which you always sequentially solve rather than subdividing.

## 方法介绍

大部分方法都是继承ForkJoinTask。

**protected abstract V compute()**

**The main computation performed by this task.**

protected boolean exec()

Implements execution conventions for RecursiveTask.

V **getRawResult**()

Returns the result that would be returned by ForkJoinTask.join(), even if this task completed abnormally, or null if this task is not known to have been completed.

protected void **setRawResult**(V value)

Forces the given value to be returned as a result.

# RecursiveAction

## RecursiveAction



public abstract class **RecursiveAction** extends **ForkJoinTask<Void>**

## 功能介绍

A recursive **resultless** **ForkJoinTask**. This class establishes conventions to parameterize resultless actions as Void **ForkJoinTasks**. Because null is the only valid value of type Void, methods such as join always return null upon completion.

## 使用示例：文档介绍

Sample Usages. Here is a simple but complete ForkJoin sort that sorts a given long[] array:

static class SortTask extends RecursiveAction {

final long[] array; final int lo, hi;

SortTask(long[] array, int lo, int hi) {

this.array = array; this.lo = lo; this.hi = hi;

}

SortTask(long[] array) { this(array, 0, array.length); }

protected void compute() {

if (hi - lo < THRESHOLD)

sortSequentially(lo, hi);

else {

int mid = (lo + hi) >>> 1;

invokeAll(new SortTask(array, lo, mid),

new SortTask(array, mid, hi));

merge(lo, mid, hi);

}

}

// implementation details follow:

static final int THRESHOLD = 1000;

void sortSequentially(int lo, int hi) {

Arrays.sort(array, lo, hi);

}

void merge(int lo, int mid, int hi) {

long[] buf = Arrays.copyOfRange(array, lo, mid);

for (int i = 0, j = lo, k = mid; i < buf.length; j++)

array[j] = (k == hi || buf[i] < array[k]) ?

buf[i++] : array[k++];

}

}

You could then sort anArray by creating new SortTask(anArray) and invoking it in a ForkJoinPool. As a more concrete simple example, the following task increments each element of an array:

class IncrementTask extends RecursiveAction {

final long[] array; final int lo, hi;

IncrementTask(long[] array, int lo, int hi) {

this.array = array; this.lo = lo; this.hi = hi;

}

protected void compute() {

if (hi - lo < THRESHOLD) {

for (int i = lo; i < hi; ++i)

array[i]++;

}

else {

int mid = (lo + hi) >>> 1;

invokeAll(new IncrementTask(array, lo, mid),

new IncrementTask(array, mid, hi));

}

}

}

The following example illustrates some refinements and idioms that may lead to better performance: RecursiveActions need not be fully recursive, so long as they maintain the basic divide-and-conquer approach. Here is a class that sums the squares of each element of a double array, by subdividing out only the right-hand-sides of repeated divisions by two, and keeping track of them with a chain of next references. It uses a dynamic threshold based on method getSurplusQueuedTaskCount, but counterbalances potential excess partitioning by directly performing leaf actions on unstolen tasks rather than further subdividing.

double sumOfSquares(ForkJoinPool pool, double[] array) {

int n = array.length;

Applyer a = new Applyer(array, 0, n, null);

pool.invoke(a);

return a.result;

}

class Applyer extends RecursiveAction {

final double[] array;

final int lo, hi;

double result;

Applyer next; // keeps track of right-hand-side tasks

Applyer(double[] array, int lo, int hi, Applyer next) {

this.array = array; this.lo = lo; this.hi = hi;

this.next = next;

}

double atLeaf(int l, int h) {

double sum = 0;

for (int i = l; i < h; ++i) // perform leftmost base step

sum += array[i] \* array[i];

return sum;

}

protected void compute() {

int l = lo;

int h = hi;

Applyer right = null;

while (h - l > 1 && getSurplusQueuedTaskCount() <= 3) {

int mid = (l + h) >>> 1;

right = new Applyer(array, mid, h, right);

right.fork();

h = mid;

}

double sum = atLeaf(l, h);

while (right != null) {

if (right.tryUnfork()) // directly calculate if not stolen

sum += right.atLeaf(right.lo, right.hi);

else {

right.join();

sum += right.result;

}

right = right.next;

}

result = sum;

}

}

## 构造方法

RecursiveAction()

## 一般方法

大部分方法都是继承ForkJoinTask。

protected abstract void compute()

The main computation performed by this task.

protected boolean exec()

Implements execution conventions for RecursiveActions.

Void getRawResult()

Always returns null.

protected void setRawResult(Void mustBeNull)

Requires null completion value.

# CountedCompleter<T>

## 继承关系

public abstract class **CountedCompleter<T>** extends **ForkJoinTask<T>**

## 功能介绍

见官网

## 方法介绍

abstract void **compute**()

The main computation performed by this task.

void **tryComplete**()

If the pending count is nonzero, decrements the count; otherwise invokes onCompletion(CountedCompleter) and then similarly tries to complete this task's completer, if one exists, else marks this task as complete.

void **complete**(T rawResult)

Regardless of pending count, invokes onCompletion(CountedCompleter), marks this task as complete and further triggers tryComplete() on this task's completer, if one exists.

CountedCompleter<?> firstComplete()

If this task's pending count is zero, returns this task; otherwise decrements its pending count and returns null.

CountedCompleter<?> getCompleter()

Returns the completer established in this task's constructor, or null if none.