J[ava Concurrency - Part 7 : Executors and thread pools](http://baptiste-wicht.com/posts/2010/09/java-concurrency-part-7-executors-and-thread-pools.html)

 Baptiste Wicht

[2010-09-15 07:17](http://baptiste-wicht.com/posts/2010/09/java-concurrency-part-7-executors-and-thread-pools.html)

[43 Comments](http://baptiste-wicht.com/posts/2010/09/java-concurrency-part-7-executors-and-thread-pools.html#disqus_thread)

[Source](http://baptiste-wicht.com/posts/2010/09/java-concurrency-part-7-executors-and-thread-pools.wp)

Let's start with a new post in the Java concurrency series.

This time we'll learn how to start cleanly new threads and to manage thread pools. In Java, if you have a Runnable like this :

Runnable runnable = **new** Runnable(){

**public** void run(){

System.out.println("Run");

}

}

You can easily run it in a new thread :

**new** Thread(runnable).start();

This is very simple and clean, but what if you've several long running tasks that you want to load in parralel and then wait for the completion of all the tasks, it's a little bit harder to code and if you want to get the return value of all the tasks it becomes really difficult to keep a good code. But like for almost any problems, Java has a solution for you, the Executors. This simple class allows you to create thread pools and thread factories.

A thread pool is represented by an instance of the class ExecutorService. With an ExecutorService, you can submit task that will be completed in the future. Here are the type of thread pools you can create with the Executors class :

* **Single Thread Executor** : A thread pool with only one thread. So all the submitted task will be executed sequentially. Method :*Executors.newSingleThreadExecutor()*
* **Cached Thread Pool** : A thread pool that create as many threads it needs to execute the task in parralel. The old available threads will be reused for the new tasks. If a thread is not used during 60 seconds, it will be terminated and removed from the pool. Method :*Executors.newCachedThreadPool()*
* **Fixed Thread Pool** : A thread pool with a fixed number of threads. If a thread is not available for the task, the task is put in queue waiting for an other task to ends. Method : *Executors.newFixedThreadPool()*
* **Scheduled Thread Pool** : A thread pool made to schedule future task. Method : *Executors.newScheduledThreadPool()*
* **Single Thread Scheduled Pool** : A thread pool with only one thread to schedule future task. Method :*Executors.newSingleThreadScheduledExecutor()*

In terms of resources, the newFixedThreadPool will keep all the threads running until they are explicitly terminated. In the newCachedThreadPool Threads that have not been used for sixty seconds are terminated and removed from the cache.

Once you have a thread pool, you can submit task to it using the different submit methods. You can submit a Runnable or a Callable to the thread pool. The method return a Future representing the future state of the task. If you submitted a Runnable, the Future object return null once the task finished.

By example, if you have this Callable :

**private** **final** **class** **StringTask** **implements** Callable<String> {

**public** String call(){

*//Long operations*

**return** "Run";

}

}

If you want to execute that task 10 times using 4 threads, you can use that code :

ExecutorService pool = Executors.newFixedThreadPool(4);

**for**(int i = 0; i < 10; i++){

pool.submit(**new** StringTask());

}

But you must shutdown the thread pool in order to terminate all the threads of the pool :

pool.shutdown();

If you don't do that, the JVM risk to not shutdown because there is still threads not terminated. You can also force the shutdown of the pool using shutdownNow, with that the currently running tasks will be interrupted and the tasks not started will not be started at all.

But with that example, you cannot get the result of the task. So let's get the Future objects of the tasks :

ExecutorService pool = Executors.newFixedThreadPool(4);

List<Future<String>> futures = **new** ArrayList<Future<String>>(10);

**for**(int i = 0; i < 10; i++){

futures.add(pool.submit(**new** StringTask()));

}

**for**(Future<String> future : futures){

String result = future.get();

*//Compute the result*

}

pool.shutdown();

But this code is a bit complicated. And there is a disadvantage. If the first task takes a long time to compute and all the other tasks ends before the first, the current thread cannot compute the result before the first task ends. Once again, Java has the solution for you, CompletionService.

A CompletionService is a service that make easier to wait for result of submitted task to an executor. The implementation is ExecutorCompletionService who's based on an ExecutorService to work. So let's try :

ExecutorService threadPool = Executors.newFixedThreadPool(4);

CompletionService<String> pool = **new** ExecutorCompletionService<String>(threadPool);

**for**(int i = 0; i < 10; i++){

pool.submit(**new** StringTask());

}

**for**(int i = 0; i < 10; i++){

String result = pool.take().get();

*//Compute the result*

}

threadPool.shutdown();

As soon as a task is completed, it is put in an internal java.util.concurrent.BlockingQueue (a highly efficient queue for Producer/Consumer problems and communication between threads).

From that queue, you can get the results of the finished tasks with take. If no task is yet available, take will wait until something is available.  
In this case we just print the result (the name of the current threat executing the Callable).

This is all you need to know to use a CompletionService. It is really simple. There is a lot of cool stuff in the JDK and in the java.util.concurrent package. Make sure to browse through the docs from time to time before inventing your own solution.

With that, you have the result in the order they are completed and you don't have to keep a collection of Future.

Here we are, you have the tools in hand to launch tasks in parralel using performing thread pools. Using Executors, ExecutorService and CompletionService you can create complex algorithm using several taks. With that tools, it's really easy to change the number of threads performing in parralel or adding more tasks without changing a lot of code.

I hope that this post will help you to write better concurrent code.

\*\*\*\*\*\*------IMP

This is where **ExecutorCompletionService** steps in. It is a thin wrapper around [ExecutorService](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html) that "remembers" all submitted tasks and allows you to wait for the first *completed*, as opposed to first *submitted* task. In a way ExecutorCompletionService keeps a handle to all intermediate Future objects and once any of them finishes, it's returned. Crucial API method is [CompletionService.take()](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CompletionService.html" \l "take()) that blocks and waits for *any* underlying Future to complete. Here is the submit step with ExecutorCompletionService:

## Task Delegation

Here is a diagram illustrating a thread delegating a task to an ExecutorService for asynchronous execution:

|  |
| --- |
| A thread delegating a task to an ExecutorService for asynchronous execution. |
| **A thread delegating a task to an ExecutorService for asynchronous execution.** |

Once the thread has delegated the task to the ExecutorService, the thread continues its own execution independent of the execution of that task.

## ExecutorService Implementations

Since ExecutorService is an interface, you need to its implementations in order to make any use of it. TheExecutorService has the following implementation in the java.util.concurrent package:

* [**ThreadPoolExecutor**](http://tutorials.jenkov.com/java-util-concurrent/threadpoolexecutor.html)
* [**ScheduledThreadPoolExecutor**](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html)

## Creating an ExecutorService

How you create an ExecutorService depends on the implementation you use. However, you can use theExecutors factory class to create ExecutorService instances too. Here are a few examples of creating anExecutorService:

ExecutorService executorService1 = Executors.newSingleThreadExecutor();

ExecutorService executorService2 = Executors.newFixedThreadPool(10);

ExecutorService executorService3 = Executors.newScheduledThreadPool(10);

## ExecutorService Usage

There are a few different ways to delegate tasks for execution to an ExecutorService:

* execute(Runnable)
* submit(Runnable)
* submit(Callable)
* invokeAny(...)
* invokeAll(...)

I will take a look at each of these methods in the following sections.

### execute(Runnable)

The execute(Runnable) method takes a java.lang.Runnable object, and executes it asynchronously. Here is an example of executing a Runnable with an ExecutorService:

ExecutorService executorService = Executors.newSingleThreadExecutor();

executorService.execute(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

executorService.shutdown();

There is no way of obtaining the result of the executed Runnable, if necessary. You will have to use aCallable for that (explained in the following sections).

### submit(Runnable)

The submit(Runnable) method also takes a Runnable implementation, but returns a Future object. ThisFuture object can be used to check if the Runnable as finished executing.

Here is a ExecutorService submit() example:

Future future = executorService.submit(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

future.get(); //returns null if the task has finished correctly.

### submit(Callable)

The submit(Callable) method is similar to the submit(Runnable) method except for the type of parameter it takes. The Callable instance is very similar to a Runnable except that its call() method can return a result. The Runnable.run() method cannot return a result.

The Callable's result can be obtained via the Future object returned by the submit(Callable) method. Here is an ExecutorService Callable example:

Future future = executorService.submit(new Callable(){

public Object call() throws Exception {

System.out.println("Asynchronous Callable");

return "Callable Result";

}

});

System.out.println("future.get() = " + future.get());

The above code example will output this:

Asynchronous Callable

future.get() = Callable Result

### invokeAny()

The invokeAny() method takes a collection of Callable objects, or subinterfaces of Callable. Invoking this method does not return a Future, but returns the result of one of the Callable objects. You have no guarantee about which of the Callable's results you get. Just one of the ones that finish.

If one of the tasks complete (or throws an exception), the rest of the Callable's are cancelled.

Here is a code example:

ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 1";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 2";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 3";

}

});

String result = executorService.invokeAny(callables);

System.out.println("result = " + result);

executorService.shutdown();

This code example will print out the object returned by one of the Callable's in the given collection. I have tried running it a few times, and the result changes. Sometimes it is "Task 1", sometimes "Task 2" etc.

### invokeAll()

The invokeAll() method invokes all of the Callable objects you pass to it in the collection passed as parameter. The invokeAll() returns a list of Future objects via which you can obtain the results of the executions of each Callable.

Keep in mind that a task might finish due to an exception, so it may not have "succeeded". There is no way on aFuture to tell the difference.

Here is a code example:

ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 1";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 2";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 3";

}

});

List<Future<String>> futures = executorService.invokeAll(callables);

for(Future<String> future : futures){

System.out.println("future.get = " + future.get());

}

executorService.shutdown();

## ExecutorService Shutdown

When you are done using the ExecutorService you should shut it down, so the threads do not keep running.

For instance, if your application is started via a main() method and your main thread exits your application, the application will keep running if you have an active ExexutorService in your application. The active threads inside this ExecutorService prevents the JVM from shutting down.

To terminate the threads inside the ExecutorService you call its shutdown() method. TheExecutorService will not shut down immediately, but it will no longer accept new tasks, and once all threads have finished current tasks, the ExecutorService shuts down. All tasks submitted to the ExecutorServicebefore shutdown() is called, are executed.

If you want to shut down the ExecutorService immediately, you can call the shutdownNow() method. This will attempt to stop all executing tasks right away, and skips all submitted but non-processed tasks. There are no guarantees given about the executing tasks. Perhaps they stop, perhaps the execute until the end. It is a best effort attempt.

# Java Fork and Join using ForkJoinPool

   
By [Jakob Jenkov](http://jakob.jenkov.com/" \t "jakob-jenkov)

 Connect with me:

Rate article:

Share article:

Table of Contents

* [Fork and Join Explained](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#fork-and-join-explained)
  + [Fork](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#fork)
  + [Join](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#join)
* [The ForkJoinPool](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#the-forkjoinpool)
  + [Creating a ForkJoinPool](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#creating-a-forkjoinpool)
  + [Submitting Tasks to the ForkJoinPool](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#submittting-tasks-to-forkjoinpool)
* [RecursiveAction](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#recursive-action)
* [RecursiveTask](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#recursive-task)
* [ForkJoinPool Critique](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#forkjoinpool-critique)

Last updated: 2015-02-03

The ForkJoinPool was added to Java in Java 7. The ForkJoinPool is similar to the [**Java ExecutorService**](http://tutorials.jenkov.com/java-util-concurrent/executorservice.html)but with one difference. The ForkJoinPool makes it easy for tasks to split their work up into smaller tasks which are then submitted to the ForkJoinPool too. Tasks can keep splitting their work into smaller subtasks for as long as it makes to split up the task. It may sound a bit abstract, so in this fork and join tutorial I will explain how the ForkJoinPool works, and how splitting tasks up work.

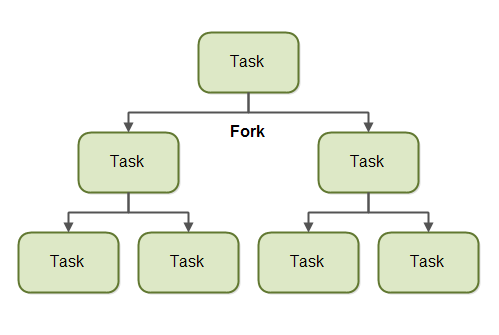
## Fork and Join Explained

Before we look at the ForkJoinPool I want to explain how the fork and join principle works in general.

The fork and join principle consists of two steps which are performed recursively. These two steps are the fork step and the join step.

### Fork

A task that uses the fork and join principle can *fork* (split) itself into smaller subtasks which can be executed concurrently. This is illustrated in the diagram below:



By splitting itself up into subtasks, each subtask can be executed in parallel by different CPUs, or different threads on the same CPU.

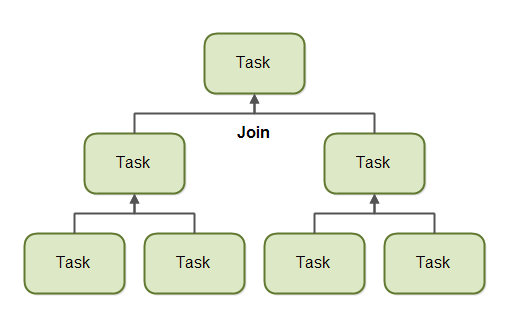
A task only splits itself up into subtasks if the work the task was given is large enough for this to make sense. There is an overhead to splitting up a task into subtasks, so for small amounts of work this overhead may be greater than the speedup achieved by executing subtasks concurrently.

The limit for when it makes sense to fork a task into subtasks is also called a threshold. It is up to each task to decide on a sensible threshold. It depends very much on the kind of work being done.

### Join

When a task has split itself up into subtasks, the task waits until the subtasks have finished executing.

Once the subtasks have finished executing, the task may *join* (merge) all the results into one result. This is illustrated in the diagram below:



Of course, not all types of tasks may return a result. If the tasks do not return a result then a task just waits for its subtasks to complete. No result merging takes place then.

## The ForkJoinPool

The ForkJoinPool is a special thread pool which is designed to work well with fork-and-join task splitting. TheForkJoinPool located in the java.util.concurrent package, so the full class name isjava.util.concurrent.ForkJoinPool.

### Creating a ForkJoinPool

You create a ForkJoinPool using its constructor. As a parameter to the ForkJoinPool constructor you pass the indicated level of parallelism you desire. The parallelism level indicates how many threads or CPUs you want to work concurrently on on tasks passed to the ForkJoinPool. Here is a ForkJoinPool creation example:

ForkJoinPool forkJoinPool = new ForkJoinPool(4);

This example creates a ForkJoinPool with a parallelism level of 4.

### Submitting Tasks to the ForkJoinPool

You submit tasks to a ForkJoinPool similarly to how you submit tasks to an ExecutorService. You can submit two types of tasks. A task that does not return any result (an "action"), and a task which does return a result (a "task"). These two types of tasks are represented by the RecursiveAction and RecursiveTaskclasses. How to use both of these tasks and how to submit them will be covered in the following sections.

## RecursiveAction

A RecursiveAction is a task which does not return any value. It just does some work, e.g. writing data to disk, and then exits.

A RecursiveAction may still need to break up its work into smaller chunks which can be executed by independent threads or CPUs.

You implement a RecursiveAction by subclassing it. Here is a RecursiveAction example:

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.RecursiveAction;

public class MyRecursiveAction extends RecursiveAction {

private long workLoad = 0;

public MyRecursiveAction(long workLoad) {

this.workLoad = workLoad;

}

@Override

protected void compute() {

//if work is above threshold, break tasks up into smaller tasks

if(this.workLoad > 16) {

System.out.println("Splitting workLoad : " + this.workLoad);

List<MyRecursiveAction> subtasks =

new ArrayList<MyRecursiveAction>();

subtasks.addAll(createSubtasks());

for(RecursiveAction subtask : subtasks){

subtask.fork();

}

} else {

System.out.println("Doing workLoad myself: " + this.workLoad);

}

}

private List<MyRecursiveAction> createSubtasks() {

List<MyRecursiveAction> subtasks =

new ArrayList<MyRecursiveAction>();

MyRecursiveAction subtask1 = new MyRecursiveAction(this.workLoad / 2);

MyRecursiveAction subtask2 = new MyRecursiveAction(this.workLoad / 2);

subtasks.add(subtask1);

subtasks.add(subtask2);

return subtasks;

}

}

This example is very simplified. The MyRecursiveAction simply takes a fictive workLoad as parameter to its constructor. If the workLoad is above a certain threshold, the work is split into subtasks which are also scheduled for execution (via the .fork() method of the subtasks. If the workLoad is below a certain threshold then the work is carried out by the MyRecursiveAction itself.

You can schedule a MyRecursiveAction for execution like this:

MyRecursiveAction myRecursiveAction = new MyRecursiveAction(24);

forkJoinPool.invoke(myRecursiveAction);

## RecursiveTask

A RecursiveTask is a task that returns a result. It may split its work up into smaller tasks, and merge the result of these smaller tasks into a collective result. The splitting and merging may take place on several levels. Here is a RecursiveTask example:

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.RecursiveTask;

public class MyRecursiveTask extends RecursiveTask<Long> {

private long workLoad = 0;

public MyRecursiveTask(long workLoad) {

this.workLoad = workLoad;

}

protected Long compute() {

//if work is above threshold, break tasks up into smaller tasks

if(this.workLoad > 16) {

System.out.println("Splitting workLoad : " + this.workLoad);

List<MyRecursiveTask> subtasks =

new ArrayList<MyRecursiveTask>();

subtasks.addAll(createSubtasks());

for(MyRecursiveTask subtask : subtasks){

subtask.fork();

}

long result = 0;

for(MyRecursiveTask subtask : subtasks) {

result += subtask.join();

}

return result;

} else {

System.out.println("Doing workLoad myself: " + this.workLoad);

return workLoad \* 3;

}

}

private List<MyRecursiveTask> createSubtasks() {

List<MyRecursiveTask> subtasks =

new ArrayList<MyRecursiveTask>();

MyRecursiveTask subtask1 = new MyRecursiveTask(this.workLoad / 2);

MyRecursiveTask subtask2 = new MyRecursiveTask(this.workLoad / 2);

subtasks.add(subtask1);

subtasks.add(subtask2);

return subtasks;

}

}

This example is similar to the RecursiveAction example except it returns a result. The classMyRecursiveTask extends RecursiveTask<Long> which means that the result returned from the task is aLong .

The MyRecursiveTask example also breaks the work down into subtasks, and schedules these subtasks for execution using their fork() method.

Additionally, this example then receives the result returned by each subtask by calling the join() method of each subtask. The subtask results are merged into a bigger result which is then returned. This kind of joining / mergining of subtask results may occur recursively for several levels of recursion.

You can schedule a RecursiveTask like this:

MyRecursiveTask myRecursiveTask = new MyRecursiveTask(128);

long mergedResult = forkJoinPool.invoke(myRecursiveTask);

System.out.println("mergedResult = " + mergedResult);

Notice how you get the final result out from the ForkJoinPool.invoke() method call.

## ForkJoinPool Critique

It seems not everyone is equally happy with the new ForkJoinPool in Java 7. While searching for experiences with, and opinions about, the ForkJoinPool, I came across the following critique:

[**A Java Fork-Join Calamity**](http://coopsoft.com/ar/CalamityArticle.html)

It is well worth a read before you plan to use the ForkJoinPool in your own projects.