Chapter 10. Thread Pools

For various reasons, thread pools are a very common tool in a multithreaded developer's toolkit. Most programs that use a lot of threads benefit in some way from using a thread pool.

J2SE 5.0 comes with its own thread pool implementation. Prior to this release, developers were left to write their own thread pool or use any number of commonly available implementations (including one we developed in earlier editions of this book and which is discussed in Appendix A). In this chapter, we discuss the thread pool implementation that comes with J2SE 5.0. If you can't use that implementation yet, the information in this chapter is still useful: you'll find out how and when using a thread pool can be advantageous. With that understanding, it's simple to use any thread pool implementation in your own program.

10.1 Why Thread Pools?

The idea behind a thread pool is to set up a number of threads that sit idle, waiting for work that they can perform. As your program has tasks to execute, it encapsulates those tasks into some object (typically a `Runnable` object) and informs the thread pool that there is a new task. One of the idle threads in the pool takes the task and executes it; when it finishes the task, it goes back and waits for another task.

Thread pools have a maximum number of threads available to run these tasks. Consequently, when you add a task to a thread pool, it might have to wait for an available thread to run it. That may not sound encouraging, but it's at the core of why you would use a thread pool.

Reasons for using thread pools fall into three categories.

The first reason thread pools are often recommended is because it's felt that the overhead of creating a thread is very high; by using a pool, we can gain some performance when the threads are reused. The degree to which this is true depends a lot on your program and its requirements. It is true that creating a thread can take as much as a few hundred microseconds, which is a significant amount of time for some programs (but not others; see Chapter 14).

The second reason for using a thread pool is very important: it allows for better program design. If your program has a lot of tasks to execute, you can perform all the thread management for those tasks yourself, but, as we've started to see in our examples, this can quickly become tedious; the code to start a thread and manage its lifecycle isn't very interesting. A thread pool allows you to delegate all the thread management to the pool itself, letting you focus on the logic of your program. With a thread pool, you simply create a task and send the task to the pool to be executed; this leads to much more elegant programs (see Chapter 11).

The primary reason to use a thread pool is that they carry important performance benefits for applications that want to run many threads simultaneously. In fact, anytime you have more active threads than CPUs, a thread pool can play a crucial role in making your program seem to run faster and more efficiently.

If you read that last sentence carefully, in the back of your mind you're probably thinking that we're being awfully weasely: what does it mean that your program "seems" to run faster? What we mean is that the throughput of your CPU-bound program running multiple calculations will be faster, and that leads to the perception that your program is running faster. It's all a matter of throughput.

10.1.1 Thread Pools and Throughput

In Chapter 9, we showed an example of what happens when a system has more threads than CPU resources. The way in which the threads perform the calculation has a big effect on the output. In particular, our first example produces this output:

```
Starting task Task 2 at 00:04:30:324

Starting task Task 0 at 00:04:30:334

Starting task Task 1 at 00:04:30:345

Ending task Task 1 at 00:04:38:052 after 7707 milliseconds

Ending task Task 2 at 00:04:38:380 after 8056 milliseconds

Ending task Task 0 at 00:04:38:502 after 8168 milliseconds
```

In this case, we have three threads and one CPU. The three threads run at the same time, are time-sliced by the operating system, and all completed execution in around eight seconds. Imagine that we have written this program as a server where each time a client connects, it is given a separate thread. When the three clients each request the service (that is, the calculation of the Fibonacci number), each will wait eight seconds for its answer.

In our second example, we run the threads sequentially and see this output:

```
Starting task Task 0 at 00:04:30:324

Ending task Task 0 at 00:04:33:052 after 2728 milliseconds

Starting task Task 1 at 00:04:33:062

Ending task Task 1 at 00:04:35:919 after 2857 milliseconds

Starting task Task 2 at 00:04:35:929

Ending task Task 2 at 00:04:38:720 after 2791 milliseconds
```

In this case, the total time to complete the calculation is still about 8 seconds, but each thread completes its execution in about 2.7 seconds. A server that runs the calculations sequentially will provide its first answer in 2.7 seconds, and the average waiting time for the clients will be 5.4 seconds.

This is what we mean by the throughput of the program. In both cases, we've done the same amount of work, but in the second case, users of the program are generally happier with the performance.

Now consider what happens if additional requests come in while the server is executing. If we create a new thread for every client, the server could quickly become overloaded: the more threads it starts, the slower it provides an answer for each request. With three simultaneous threads, our calculation takes eight seconds. If a new request arrives every 2.7 seconds or so, we never finish. The server starts more and more threads, each thread gets less and less CPU time, and none ever finish.

On the other hand, if we run the requests sequentially using only one thread, the server reaches a steady state. With three requests in the queue, each subsequent request arrives as another one finishes. We can supply an endless number of answers to the clients; each client waits about eight seconds for a response.

This reasoning applies to programs other than servers. For instance, an image processing application may nicely partition its image and be able to work on each partition in a separate thread. If a user is watching the image on screen, you might want to display the results of one partition while another one is being manipulated.

The similarity to programs like this and servers is that the results of each thread are interesting. The result of a single calculation is interesting to the client that requested it, the result of a partition of the image is interesting to the user viewing the screen, and so on. In these cases, throttling the number of threads provides a better experience for the users of the application.

Clearly, parts of this discussion are contrived; we've selected the numbers in the best way possible to make our point, and we've used a calculation that needs only CPU resources to complete. In the real world, requests arrive at the server in random bursts, and processing the request involves making database calls or something else that is likely to block. Those things complicate using a thread pool, but they do not eliminate its benefits.

The fact that threads may block means that we need to have more threads than CPUs in our pool. So far, we've considered cases where there is one CPU and have seen that one CPU-intensive thread gives us the best throughput. If the thread spends 50% of its time blocked, you want two threads per CPU; if the thread blocks 66% of the time, you want three threads per CPU, and so on.

Of course, you're unlikely to be able to model your program in such detail. And any model becomes far harder to calculate once you start to account for random bursts in traffic. In the end, you'll need to run some tests to determine an appropriate size for your thread pool. But if CPU resources are sometimes scarce, throttling the number of threads (while still keeping the CPUs utilized) increases the throughput of your application.

10.1.2 Why Not Thread Pools?

If your program is doing batch processing, or simply providing a single answer or report, it doesn't really matter if you use as many threads as possible or a thread pool: if no one is interested in the results given by each thread, it doesn't matter if some of them finish before others. That doesn't mean that you can expect to create thousands of threads with impunity: threads take memory, and the more memory you use, the more impact you'll have on your system performance. Additionally, there is some slight overhead when the operating system manages thousands of threads instead of just a few. Still, if your program design nicely separates into multiple threads and you're interested only in the end result of all those threads, a thread pool isn't necessary.

Thread pools are also not necessary when available CPU resources are adequate to handle all the work the program needs to do. In fact, in this case a thread pool may do more harm than good. Obviously, if your system has eight CPUs and you have only four threads in your thread pool, tasks wait for a thread even though four CPUs are idle. With a thread pool, you want to throttle the total number of threads so that they don't overwhelm your system, but you never want to have fewer runnable threads than CPUs.

PREV < Day Day Up > NEXT
PREV < Day Day Up > NEXT

10.2 Executors

Java's implementation of thread pools is based on an executor. An executor is a generic concept modelled by this interface:

```
package java.util.concurrent;

public interface Executor {

    public void execute(Runnable task);

}
```

Executors are a useful design pattern for multithreaded programs because they allow you to model your program as a series of tasks. You don't need to worry about the thread details associated with the task: you simply create the task and pass it to the `execute()` method of an appropriate executor.

J2SE 5.0 comes with two kinds of executors. It comes with a thread pool executor, which we'll show next. It also provides a task scheduling executor, which we examine in Chapter 11. Both of these executors are defined by this interface:

```
package java.util.concurrent;

public interface ExecutorService extends Executor {

    void shutdown( );

    List shutdownNow( );

    boolean isShutdown( );

    boolean isTerminated( );

    boolean awaitTermination(long timeout, TimeUnit unit)

            throws InterruptedException;

    <T> Future<T> submit(Callable<T> task);

    <T> Future<T> submit(Runnable task, T result);

    Future<?> submit(Runnable task);

    <T> List<Future<T>> invokeAll(Collection<Callable<T>> tasks)

            throws InterruptedException;

    <T> List<Future<T>> invokeAll(Collection<Callable<T>> tasks,

                                  long timeout, TimeUnit unit)

            throws InterruptedException;

    <T> T invokeAny(Collection<Callable<T>> tasks)

            throws InterruptedException, ExecutionException;

    <T> T invokeAny(Collection<Callable<T>> tasks,  long timeout, TimeUnit unit)

            throws InterruptedException, ExecutionException, TimeoutException;

}
```

This interface provides a means for you to manage the executor and its tasks. The `shutdown()` method gracefully terminates the executor: any tasks that have already been sent to the executor are allowed to run, but no new tasks are accepted. When all tasks are completed, the executor stops its thread(s). The `shutdownNow()` method attempts to stop execution sooner: all tasks that have not yet started are not run and are instead returned in a list. Still, existing tasks continue to run: they are interrupted, but it's up to the runnable object to check its interrupt status and exit when convenient.

So there's a period of time between calling the `shutdown()` or `shutdownNow()` method and when tasks executing in the executor service are all complete. When all tasks are complete (including any waiting tasks), the executor service enters a terminated state. You can check to see if the executor service is in the terminated state by calling the `isTerminated()` method (or you can wait for it to finish the pending tasks by calling the `awaitTerminated()` method).

An executor service also allows you to handle many tasks in ways that the simple `Executor` interface does not accommodate. Tasks can be sent to an executor service via a `submit()` method, which returns a `Future` object that can be used to track the progress of the task. The `invokeAll( )` methods execute all the tasks in the given collection. The `invokeAny()` methods execute the tasks in the given collection, but when one task has completed, the remaining tasks are subject to cancellation. We'll discuss `Future` objects and cancellation later in this chapter.

PREV < Day Day Up > NEXT
PREV < Day Day Up > NEXT

10.3 Using a Thread Pool

To use a thread pool, you must do two things: you must create the tasks that the pool is to run, and you must create the pool itself. The tasks are simply `Runnable` objects, so that meshes well with a standard approach to threading (in fact, the task that we'll use for this example is the same `Runnable` task we use in Chapter 9 to calculate a Fibonacci number). You can also use `Callable` objects to represent your tasks (which we'll do later in this chapter), but for most simple uses, a `Runnable` object is easier to work with.

The pool is an instance of the `ThreadPoolExecutor` class. That class implements the `ExecutorService` interface, which tells us how to feed it tasks and how to shut it down. We'll look at the other aspects of that class in this section, beginning with how to construct it.

```
package java.util.concurrent;

public class ThreadPoolExecutor implements ExecutorService {

    public ThreadPoolExecutor(int corePoolSize,

                              int maximumPoolSize,

                              long keepAliveTime,

                              TimeUnit unit,

                              BlockingQueue<Runnable> workQueue);

    public ThreadPoolExecutor(int corePoolSize,

                              int maximumPoolSize,

                              long keepAliveTime,

                              TimeUnit unit,
                              BlockingQueue<Runnable> workQueue,

                              ThreadFactory threadFactory);

    public ThreadPoolExecutor(int corePoolSize,

                              int maximumPoolSize,

                              long keepAliveTime,

                              TimeUnit unit,

                              BlockingQueue<Runnable> workQueue,

                              RejectedExecutionHandler handler);

    public ThreadPoolExecutor(int corePoolSize,

                              int maximumPoolSize,

                              long keepAliveTime,

                              TimeUnit unit,

                              BlockingQueue<Runnable> workQueue,

                              ThreadFactory threadFactory,

                              RejectedExecutionHandler handler);

}
```

The core pool size, maximum pool size, keep alive times, and so on control how the threads within the pool are managed. We describe each of these concepts in our next section.

For now, we can use a constructor to create the tasks and put them in the thread pool:

```
package javathreads.examples.ch10.example1;


import java.util.concurrent.*;

import javathreads.examples.ch10.*;


public class ThreadPoolTest {


    public static void main(String[] args) {

        int nTasks = Integer.parseInt(args[0]);

        long n = Long.parseLong(args[1]);

        int tpSize = Integer.parseInt(args[2]);


        ThreadPoolExecutor tpe = new ThreadPoolExecutor(
            tpSize, tpSize, 50000L, TimeUnit.MILLISECONDS,
            new LinkedBlockingQueue<Runnable>( ));

        Task[] tasks = new Task[nTasks];
        for (int i = 0; i < nTasks; i++) {
            tasks[i] = new Task(n, "Task " + i);
            tpe.execute(tasks[i]);
        }
        tpe.shutdown( );
    }
}
```

In this example, we're using the tasks to calculate Fibonacci numbers as we do in Chapter 9. Once the pool is constructed, we simply add the tasks to it (using the `execute()` method). When we're done, we gracefully shut down the pool; the existing tasks run to completion, and then all the existing threads exit. As you can see, using the thread pool is quite simple, but the behavior of the pool can be complex depending on the arguments used to construct it. We'll look into that in the next section.

PREV < Day Day Up > NEXT
PREV < Day Day Up > NEXT

10.4 Queues and Sizes

The two fundamental things that affect a thread pool are its size and the queue used for the tasks. These are set in the constructor of the thread pool; the size can change dynamically while the queue must remain fixed. In addition to the constructor, these methods interact with the pool's size and queue:

```
package java.util.concurrent;
public class ThreadPoolExecutor implements ExecutionService {
    public boolean prestartCoreThread( );
    public int prestartAllCoreThreads( );
    public void setMaximumPoolSize(int maximumPoolSize);
    public int getMaximumPoolSize( );
    public void setCorePoolSize(int corePoolSize);
    public int getCorePoolSize( );
    public int getPoolSize( );
    public int getLargestPoolSize( );
    public int getActiveCount( );
    public BlockingQueue<Runnable> getQueue( );


    public long getTaskCount( );
    public long getCompletedTaskCount( );
}
```

The first set of methods deal with the thread pool's size, and the remaining methods deal with its queue.

Size

> The size of the thread pool varies between a given minimum (or core) and maximum number of threads. In our example, we use the same parameter for both values, making the thread pool a constant size.
>
> If you specify different numbers for the minimum and maximum number of threads, the thread pool dynamically alters the number of threads it uses to run its tasks. The current size (returned from the `getPoolSize()` method) falls between the core size and the maximum size.

Queue

> The queue is the data structure used to hold tasks that are awaiting execution. The choice of queue affects how certain tasks are scheduled. In this case, we've used a linked blocking queue, which places the least constraints on how tasks are added to the queue. Once you've passed this queue to the thread pool, you should not call any methods on it directly. In particular, do not add items directly to the queue; add them through the `execute()` method of the thread pool. The `getQueue()` method returns the queue, but you should use that for debugging purposes only; don't execute methods directly on the queue or the internal workings of the thread pool become confused.

These parameters allow considerable flexibility in the way the thread pool operates. The basic principle is that the thread pool tries to keep its minimum number of threads active. If it gets too busy (where busy is a property of the particular queue that the thread pool uses), it adds threads until the maximum number of threads is reached, at which point it does not allow any more tasks to be queued.

There are some nuances in this, particularly in how the queue interacts with the number of threads. Let's take it step by step:

1. The thread pool is constructed with M core threads and N maximum threads. At this point, no threads are actually created (though you can specify that the pool create the M core threads by calling the thread pool's `prestartAllCoreThreads()` method or that it preallocate one core thread by calling the `prestartCoreThread()` method).

2. A task enters the pool (via the thread pool's `execute()` method). Now one of five things happens:

- If the pool has created fewer than M threads, it starts a new thread and runs the new task immediately. Even if some of the existing threads are idle, a new thread is created in the pool's attempt to reach M threads.
- If the pool has between M and N threads and one of those threads is idle, the task is run by an idle thread.
- If the pool has between M and N threads and all the threads are busy, the thread pool examines the existing work queue. If the task can be placed on the work queue without blocking, it's put on the queue and no new thread is started.
- If the pool has between M and N threads, all threads are busy, and the task cannot be added to the queue without blocking, the pool starts a new thread and runs the task on that thread.
- If the pool has N threads and all threads are busy, the pool attempts to place the new task on the queue. If the queue has reached its maximum size, this attempt fails and the task is rejected. Otherwise, the task is accepted and run when a thread becomes idle (and all previously queued tasks have run).

3. A task completes execution. The thread running the task then runs the next task on the queue. If no tasks are on the queue, one of two things happens:

- If the pool has more than M threads, the thread waits for a new task to be queued. If a new task is queued within the timeout period, the thread runs it. If not, the thread exits, reducing the total number of threads in the pool. The timeout period is a parameter used to construct the thread pool; in our example, we specified 50 seconds (50000L time units of `TimeUnit.MILLISECONDS`). Note that if the specified timeout is 0, the thread always exits, regardless of the requested minimum thread pool size.
- If the pool has M or fewer threads, the thread blocks indefinitely waiting for a new task to be queued (unless the timeout was 0, in which case it exits). It runs the new task when available.

What are the implications of all this? It means that the choice of pool size and queue are important to getting the behavior you want. For a queue, you have three choices:

- A `SynchronousQueue`, which effectively has a size of 0. In this case, whenever the pool tries to queue a task, it fails. The implication of this is tasks are either run immediately (because the pool has an idle thread or is below its threshold and, therefore, creates a new thread) or are rejected immediately. Note that you can prevent rejection of a task if you specify an unlimited maximum number of threads, but this prevents the throttling benefit of using a thread pool in the first place.
- An unbounded queue, such as a `LinkedBlockingQueue` with an unlimited capacity. In this case, adding a task to the queue always succeeds, which means that the thread pool never creates more than M threads and never rejects a task.
- A bounded queue, such as a `LinkedBlockingQueue` with a fixed capacity or an `ArrayBlockingQueue`. Let's suppose that the queue has a bounds of P. As tasks are added to the pool, it creates threads until it reaches M threads. At that point, it starts queueing tasks until the number of waiting tasks reaches P. As more tasks are added, the pool starts adding threads until it reaches N threads. If we reach a state where N threads are active and P tasks are queued, additional tasks are rejected.

In our example, we used a `LinkedBlockingQueue` with an unbounded capacity and a fixed pool size. This is perhaps the most common configuration of thread pools: it allows tasks to wait for an available thread, and a fixed number of threads is easier to monitor than a variable number of threads. A good alternative to this is to use a bounded queue with a fixed number of threads. In this model, if tasks start to arrive faster than they can be processed, they queue. Unlike the unbounded case, however, at some point the queue threshold is reached and your program must take appropriate action: if it's a server, it can reject future requests from clients, telling them that it's too busy right now and they should try again later.

If you use a thread pool, there is no magic formula that you can use to determine its optimal size and queuing strategy. When the operations are strictly CPU-bound, use only as many threads as there are CPUs. For more complex operations, choosing a thread pool size is a matter of testing different values to see which gives you the best program performance.

10.4.1 Rejected Tasks

Depending on the type of queue you use in the thread pool, a task may be rejected by the `execute()` method. Tasks are rejected if the queue is full or if the `shutdown( )` method has been called on the thread pool.

When a task is rejected, the thread pool calls the rejected execution handler associated with the thread pool. These APIs deal with the rejected execution handler:

```
package java.util.concurrent;

public interface RejectedExecutionHandler {

    public void rejectedExecution(Runnable r, ThreadPoolExecutor executor);

}
```

```
package java.util.concurrent;

public class ThreadPoolExecutor implements ExecutorService {

    public void setRejectedExecutionHandler(RejectedExecutionHandler handler);

    public RejectedExecutionHandler getRejectedExecutionHandler( );

    public static class AbortPolicy implements RejectedExecutionHandler;

    public static class CallerRunsPolicy implements RejectedExecutionHandler;

    public static class DiscardPolicy implements RejectedExecutionHandler;

    public static class DiscardOldestPolicy implements RejectedExecutionHandler;

}
```

There is one rejected execution handler for the entire pool; it applies to all potential tasks. You can write your own rejected execution handler, or you can use one of four predefined handlers. By choosing a predefined rejected execution handler—or by creating your own handler—your program can take appropriate action when a task is rejected.

Here are the predefined handlers:

AbortPolicy

> This handler does not allow the new task to be scheduled when the queue is full (or the pool has been shut down); in that case, the `execute()` method throws a `RejectedExecutionException`. That exception is a runtime exception, so when using this policy, it's up to the program to catch the exception. Otherwise, the exception is propagated up the stack.
>
> This is the default policy for rejected tasks.

CallerRunsPolicy

> This handler executes the new task independently of the thread pool if the queue is full. That is, rather than queuing the task and executing it in another thread, the task is immediately executed by calling its `run()` method, and the `execute()` method does not return until the task has completed. If the task is rejected because the pool has been shut down, the task is silently discarded.

DiscardPolicy

> This handler silently discards the task. No exception is thrown.

DiscardOldestPolicy

> This handler silently discards the oldest task in the queue and then queues the new task. When used with a `LinkedBlockingQueue` or `ArrayBlockingQueue`, the oldest task is the one that is first in line to execute when a thread becomes idle. When used with a `SynchronousQueue`, there are never waiting tasks and so the `execute()` method silently discards the submitted task.
>
> If the pool has been shut down, the task is silently discarded.

To create your own rejected task handler, create a class that implements the `RejectedExecutionHandler` interface. Your handler (just like a predefined handler) can then be set using the `setRejectedExecutionHandler()` method of the thread pool executor.

PREV < Day Day Up > NEXT
PREV < Day Day Up > NEXT

10.5 Thread Creation

The thread pool dynamically creates threads according to the size policies in effect when a task is queued and terminates threads when they've been idle too long. Those policies are set when the pool is constructed, and they can be altered with these methods:

```
package java.util.concurrent;

public interface ThreadFactory {

    public Thread newThread(Runnable r);
}
```

```
package java.util.concurrent;

public class ThreadPoolExecutor implements ExecutorService {

    public void setThreadFactory(ThreadFactory threadFactory);

    public ThreadFactory getThreadFactory( );

    public void setKeepAliveTime(long time, TimeUnit unit);

    public long getKeepAliveTime(TimeUnit unit);

}
```

When the pool creates a thread, it uses the currently installed thread pool factory to do so. Creating and installing your own thread factory allows you to set up a custom scheme to create threads so that they are created with special names, priorities, daemon status, thread group, and so on.

The default thread factory creates a thread with the following characteristics:

- New threads belong to the same thread group as the thread that created the executor. However, the security manager policy can override this and place the new thread in its own thread group (see Chapter 13).
- The name of the thread reflects its pool number and its thread number within the pool. Within a pool, threads are numbered consecutively beginning with 1; thread pools are globally assigned a pool number consecutively beginning with 1.
- The daemon status of the thread is the same as the status of the thread that created the executor.
- The priority of the thread is `Thread.NORM_PRIORITY`.

PREV < Day Day Up > NEXT
PREV < Day Day Up > NEXT

10.6 Callable Tasks and Future Results

Executors in general operate on tasks, which are objects that implement the `Runnable` interface. In order to provide more control over tasks, Java also defines a special runnable object known as a callable task:

```
package java.util.concurrent;

public interface Callable<V> {

    public <V> call( ) throws Execption;

}
```

Unlike a runnable object, a callable object can return a result or throw a checked exception. Callable objects are used only by executor services (not simple executors); the services operate on callable objects by invoking their `call()` method and keeping track of the results of those calls.

When you ask an executor service to run a callable object, the service returns a `Future` object that allows you to retrieve those results, monitor the status of the task, and cancel the task. The `Future` interface looks like this:

```
public interface Future<V> {

    V get( ) throws InterruptedException, ExecutionException;

    V get(long timeout, TimeUnit unit)

        throws InterruptedException, ExecutionException, TimeoutException;

    boolean isDone( );

    boolean cancel(boolean mayInterruptIfRunning);

    boolean isCancelled( );

}
```

Callable and future objects have a one-to-one correspondence: every callable object that is sent to an executor service returns a matching future object. The `get()` method of the future object returns the results of its corresponding `call( )` method. The `get()` method blocks until the `call()` method has returned (or until the optional timeout has expired). If the `call()` method throws an exception, the `get()` method throws an `ExecutionException` with an embedded cause, which is the exception thrown by the `call( )` method.

The future object keeps track of the state of an embedded `Callable` object. The state is set to cancelled when the `cancel()` method is called. When the `call()` method of a callable task is called, the `call()` method checks the state: if the state is cancelled, the `call()` method immediately returns.

When the `cancel()` method is called, the corresponding callable object may be in one of three states. It may be waiting for execution, in which case its state is set to cancelled and the `call()` method is never executed. It may have completed execution, in which case the `cancel( )` method has no effect. The object may be in the process of running. In that case, if the `mayInterruptIfRunning` flag is false, the `cancel()` method again has no effect.

If the `mayInterruptIfRunning` flag is true, however, the thread running the callable object is interrupted. The callable object must still pay attention to this, periodically calling the `Thread.interrupted()` method to see if it should exit.

When an object in a thread pool is cancelled, there is no immediate effect: the object still remains queued for execution. When the thread pool is about to execute the object, it checks the object's internal state, sees that it has been cancelled, and skips execution of the object. So, cancelling an object on a thread pool queue does not immediately make space in the thread pool's queue. Future calls to the `execute()` method may still be rejected, even though cancelled objects are on the thread pool's queue: the `execute()` method does not check the queue for cancelled objects.

One way to deal with this situation is to call the `purge()` method on the thread pool. The `purge()` method looks over the entire queue and removes any cancelled objects. One caveat applies: if a second thread attempts to add something to the pool (using the `execute()` method) at the same time the first thread is attempting to purge the queue, the attempt to purge the queue fails and the canceled objects remain in the queue.

A better way to cancel objects with thread pools is to use the `remove()` method of the thread pool, which immediately removes the task from the thread pool queue. The `remove()` method can be used with standard runnable objects.

10.6.1 The FutureTask Class

You can associate a `Runnable` object with a future result using the `FutureTask` class:

```
public class FutureTask<V> implements Future<V>, Runnable {}
```

This class is used internally by the executor service: the object returned from the `submit()` method of an executor service is an instance of this class. However, you can use this class directly in programs as well. This makes sense when you need to monitor the status of a runnable object within an executor: you can construct a future task with an embedded runnable object and send the future task to the `execute()` method of an executor (or an executor service). You can then use the methods of the `Future` interface to monitor the status of the `run()` method of the embedded runnable object.

A `FutureTask` object can hold either an embedded runnable or callable object, depending on which constructor is used to instantiate the object:

```
public FutureTask(Callable<V> task);

public FutureTask(Runnable task, V result);
```

The `get()` method of a future task that embeds a callable task returns whatever is returned by the `call( )` method of that embedded object. The `get()` method of a future task that embeds a runnable object returns whatever object was used to construct the future task object itself.

We use this class in our next example and also in our examples in Chapter 15.

PREV < Day Day Up > NEXT
PREV < Day Day Up > NEXT

10.7 Single-Threaded Access

In Chapter 7, we saw the threading restrictions placed on developers using the Swing library. Swing classes are not threadsafe, so they must always be called from a single thread. In the case of Swing, that means that they must be called from the event-dispatching thread, using the `invokeLater()` and `invokeAndWait()` methods of the `SwingUtilities` class.

What if you have a different library that isn't threadsafe and want to use the library in your multithreaded programs? As long as you access that library from a single thread, your program won't run into any problems with data synchronization.

Here's a class you can use to accomplish that:

```
package javathreads.examples.ch10;


import java.util.concurrent.*;
import java.io.*;

public class SingleThreadAccess {

    private ThreadPoolExecutor tpe;

    public SingleThreadAccess( ) {
        tpe = new ThreadPoolExecutor(
              1, 1, 50000L, TimeUnit.SECONDS,
              new LinkedBlockingQueue<Runnable>( ));
    }

    public void invokeLater(Runnable r) {
        tpe.execute(r);
    }

    public void invokeAndWait(Runnable r)
                    throws InterruptedException, ExecutionException {
        FutureTask task = new FutureTask(r, null);
        tpe.execute(task);
        task.get( );
    }

    public void shutdown( ) {
        tpe.shutdown( );
    }
}
```

The methods of this class function exactly like their counterparts in the `SwingUtilities` class: the `invokeLater()` method runs its task asynchronously and the `invokeAndWait()` method runs it synchronously. Because the thread pool has only a single thread, all tasks passed to the `SingleThreadAccess` object are executed by a single thread, regardless of how many threads use the access object: the tasks run by the `SingleThreadAccess` object can call thread-unsafe classes.

In Chapter 9, we show the effect of running our Fibonacci calculations when the threads are serialized; our online examples for this chapter show (as example 2) how to use the `SingleThreadAccess` class to achieve that same behavior.

< Day Day Up >
< Day Day Up >

10.8 Summary

In this chapter, we began exploration of executors: utilities that process `Runnable` objects while hiding threading details from the developer. Executors are very useful because they allow programs to be written as a series of tasks; programmers can focus on the logic of their program without getting bogged down in details about how threads are created or used.

The thread pool executor is one of two key executors in Java. In addition to the programming benefits common to all executors, thread pools can also benefit programs that have lots of simultaneous tasks to execute. Using a thread pool throttles the number of threads. This reduces competition for the CPU and allows CPU-intensive programs to complete individual tasks more quickly.

The combination of individual tasks and a lack of CPU resources is key to when to use a thread pool. Thread pools are often considered important because reusing threads is more efficient than creating threads, but that turns out to be a red herring. From a performance perspective, you'll see a benefit from thread pools because when there is less competition for the CPU (because of fewer threads), the average time to complete an individual task is less than otherwise.

The key to effectively using Java's thread pool implementation is to select an appropriate size and queueing model for the pool. Selecting a queuing model is a factor of how you want to handle many requests: an unbounded queue allows the requests to accumulate while other models possibly result in rejected tasks that must be handled by the program. A little bit of work is required to get the most out of a thread pool. But the rewards—both in terms of the simplification of program logic and in terms of potential throughput—make thread pools very useful.

10.8.1 Example Classes

Here are the class names and Ant targets for the examples in this chapter:

Description	Main Java class	Ant target
Fibonacci Calculator with Thread Pool	`javathreads.examples.ch10.example1.ThreadPoolTest nRequests NumberToCalculate ThreadPoolSize`	ch10-ex1
Fibonacci Calculator using `SingleThreadAccess`	`javathreads.examples.ch10.example2.SingleThreadTest nRequests NumberToCalculate`	ch10-ex2

The properties for the Ant tasks are:

```
<property name="nThreads" value="10"/>

<property name="FibCalcValue" value="20"/>

<property name="ThreadPoolSize" value="5"/>
```

PREV < Day Day Up > NEXT