Java theory and practice: Concurrency made simple (sort of)

An introduction to the util.concurrent package

Brian Goetz (brian@quiotix.com)
Principal Consultant
Quiotix Corp

01 November 2002

Like many other application infrastructure services, concurrency utility classes such as work queues and thread pools are often needlessly rewritten from scratch for every project. This month, Brian Goetz offers an introduction to Doug Lea's `util.concurrent` package, a high-quality, widely used, open-source package of concurrency utilities.

View more content in this series

Most of us would never think of writing our own XML parser, text indexing and search engine, regular expression compiler, XSL processor, or PDF generator as part of a project that needs one of these utilities. When we need these facilities, we use a commercial or open source implementation to perform these tasks for us, and with good reasons -- the existing implementations do a good job, are easily available, and writing our own would be a lot of work for relatively little (or no) gain. As software engineers, we like to believe that we share Isaac Newton's enthusiasm for standing on the shoulders of giants, and this is sometimes, but not always, the case. (In his Turing Award lecture, Richard Hamming suggested that computer scientists instead prefer to "stand on each other's feet.")

Wheels reinvented, inquire within

When it comes to low-level application framework services such as logging, database connection pooling, caching, and task scheduling, which nearly every server application requires, we see these basic infrastructure services rewritten over and over again. Why is this? It's not necessarily because the existing options were inadequate or because the custom versions are better or more well-suited to the application at hand. In fact, the custom versions are often not any better suited to the application for which they are developed than the widely available, general-purpose implementations, and may well be inferior. For example, while you might not like log4j, it gets the job done. And while homegrown logging systems may have specific features log4j lacks, for most applications you'd be hard-pressed to argue that a full-blown custom logging package would be worth the cost of writing it from scratch instead of using an existing, general-purpose

implementation. And yet, many project teams end up writing their own logging, connection pooling, or thread scheduling packages, over and over again.

Deceptively simple

One of the reasons we wouldn't consider writing our own XSL processor is that it would be a tremendous amount of work. But these low-level framework services are deceptively simple, and so writing our own doesn't seem that hard. But they are harder to do correctly than it might first appear. The primary reason these particular wheels keep getting reinvented is that the need for these facilities in a given application often starts small, but grows as you run into the same issues that countless other projects have. The argument usually goes like this: "We don't need a full-blown logging/scheduling/caching package, we just need something simple, so we'll just write something to do that, and it will be tailored for our specific needs." But often, you quickly outgrow the simple facility you've written, and are tempted to add a few more features, and a few more, until you've written a full-blown infrastructure service. And at that point, you're usually wedded to what you've already written, whether it's better or not. You've already paid the full cost of building your own, so in addition to the actual migration cost of moving to a general-purpose implementation, you'd have to overcome the "sunk cost" barrier as well.

A treasure trove of concurrency building blocks

Scheduling and concurrency infrastructure classes are definitely harder to write than they look. The Java language provides a useful set of low-level synchronization primitives -- `wait()`, `notify()`, and `synchronized` -- but the details of using these primitives are tricky and there are many performance, deadlock, fairness, resource management, and thread-safety hazards to avoid. Concurrent code is hard to write and harder to test -- and even the experts sometimes get it wrong the first time. Doug Lea, author of *Concurrent Programming in Java* (see Resources), has written an excellent free package of concurrency utilities, including locks, mutexes, queues, thread pools, lightweight tasks, efficient concurrent collections, atomic arithmetic operations, and other basic building blocks of concurrent applications. This package, generally referred to as `util.concurrent` (because the real package name is too long), will form the basis of the `java.util.concurrent` package in JDK 1.5, being standardized under Java Community Process JSR 166. In the meantime, `util.concurrent` is well-tested and is used in many server applications, including the JBoss J2EE application server.

Filling a void

A useful set of high-level synchronization tools, such as mutexes, semaphores, and blocking, thread-safe collection classes, was a glaring omission from the core Java class libraries. The Java language's concurrency primitives -- `synchronization`, `wait()`, and `notify()` -- are too low-level for the needs of most server applications. What happens if you need to try to acquire a lock, but time out if you don't get it within a certain period of time? Abort an attempt to acquire a lock if a thread is interrupted? Create a lock that at most N threads can hold? Support multi-mode locking, such as concurrent-read with exclusive-write? Or acquire a lock in one method and release it in another? The built-in locking supports none of these directly, but all of them can be built on the basic concurrency primitives that the Java language provides. But doing so is tricky and easy to get wrong.

Server application developers need simple facilities to enforce mutual exclusion, synchronize responses to events, communicate data across activities, and asynchronously schedule tasks. The low-level primitives that the Java language provides for this are difficult to use and error-prone. The `util.concurrent` package aims to fill this void by providing a set of classes for locking, blocking queues, and task scheduling, which provide the ability to deal with common error cases or bound the resources consumed by task queues and work-in-process.

Scheduling asynchronous tasks

The most widely used classes in `util.concurrent` are those that deal with scheduling of asynchronous events. In the July installment of this column, we looked at thread pools and work queues, and how the pattern of "queue of `Runnable`" is used by many Java applications to schedule small units of work.

It is very tempting to fork a background thread to execute a task by simply creating a new thread for the task:

```
new Thread(new Runnable() { ... } ).start();
```

While this notation is nice and compact, it has two significant disadvantages. First, creating a new thread has a certain resource cost, and so spawning many threads, each of which will perform a short task and then exit, means that the JVM may be doing more work and consuming more resources creating and destroying threads than actually doing useful work. Even if the creation and teardown overhead were zero, there is still a second, more subtle disadvantage to this execution pattern -- how do you bound the resources used in executing tasks of a certain type? What would prevent you from spawning a thousand threads at once, if a flood of requests came in all of a sudden? Real-world server applications need to manage their resources more carefully than this. You need to limit the number of asynchronous tasks executing at once.

Thread pools solve both of these problems -- they offer the advantages of improved scheduling efficiency and bounding resource utilization at the same time. While one can easily write a work queue and thread pool that executes `Runnable`s in pool threads (the example code in July's column will do the job), there is a lot more to writing an effective task scheduler than simply synchronizing access to a shared queue. A real-world task scheduler should deal with threads that die, kill excess pool threads so they don't consume resources unnecessarily, manage the pool size dynamically based on load, and bound the number of tasks queued. The last item, bounding the number of tasks queued, is important for keeping server applications from crashing due to out-of-memory errors when they become overloaded.

Bounding the task queue requires a policy decision -- if the work queue overflows, what do you do with the overflow? Throw away the newest item? Throw away the oldest item? Block the submitting thread until space is available on the queue? Execute the new item in the submitting thread? There are a variety of viable overflow-management policies, each of which is appropriate in some situations and inappropriate in others.

Executor

`Util.concurrent` defines an interface, `Executor`, to execute `Runnable`s asynchronously, and defines several implementations of `Executor` that offer different scheduling characteristics. Queuing a task to an executor is quite simple:

```
Executor executor = new QueuedExecutor();
...
Runnable runnable = ... ;
executor.execute(runnable);
```

The simplest implementation, `ThreadedExecutor`, creates a new thread for each `Runnable`, and provides no resource management -- much like the `new Thread(new Runnable() {}).start()` idiom. But `ThreadedExecutor` has one significant advantage: by changing only the construction of your executor, you can move to a different execution model without having to crawl through your entire application source to find all the places where you create new threads. `QueuedExecutor` uses a single background thread to process all tasks, much like the event thread in AWT and Swing. `QueuedExecutor` has the nice property that tasks are executed in the order they were queued, and because they are all executed within a single thread, tasks don't necessarily need to synchronize all accesses to shared data.

`PooledExecutor` is a sophisticated thread pool implementation, which not only provides scheduling of tasks in a pool of worker threads, but also provides flexible pool-size tuning and thread life-cycle management, can bound the number of items on the work queue to prevent queued tasks from consuming all available memory, and offers a variety of available shutdown and saturation policies (block, discard, throw, discard-oldest, run-in-caller, and so on). All the `Executor` implementations manage thread creation and teardown for you, including shutting down all threads when the executor is shut down, and they also provide hooks into the thread creation process so that your application can manage thread instantiation if it wants to. This allows you to, for example, place all worker threads in a particular `ThreadGroup` or give them a descriptive name.

FutureResult

Sometimes you want to start a process asynchronously, in the hopes that the results of that process will be available when you need it later. The `FutureResult` utility class makes this easy. `FutureResult` represents a task that may take some time to execute and which can execute in another thread, and the `FutureResult` object serves as a handle to that execution process. Through it, you can find out if the task has completed, wait for it to complete, and retrieve its result. `FutureResult` can be combined with `Executor`; you can create a `FutureResult` and queue it to an executor, keeping a reference to the `FutureResult`. Listing 1 shows a simple example of `FutureResult` and `Executor` together, which starts the rendering of an image asynchronously and continues with other processing:

Listing 1. FutureResult and Executor in action

```
   Executor executor = ...
   ImageRenderer renderer = ...

        FutureResult futureImage = new FutureResult();
        Runnable command = futureImage.setter(new Callable() {
           public Object call() { return renderer.render(rawImage); }
        });

 // start the rendering process
        executor.execute(command);

        // do other things while executing
        drawBorders();
        drawCaption();

 // retrieve the future result, blocking if necessary
        drawImage((Image)(futureImage.get())); // use future
```

FutureResult and caching

You can also use `FutureResult` to improve the concurrency of load-on-demand caches. By placing a `FutureResult` into the cache, rather than the result of the computation itself, you can reduce the time that you hold the write lock on the cache. While it won't speed up the first thread to place an item in the cache, it *will* reduce the time that the first thread blocks other threads from accessing the cache. It will also make the result available earlier to other threads since they can retrieve `FutureTask` from the cache. Listing 2 is an example of using `FutureResult` for caching:

Listing 2. Using FutureResult to improve caching

```
public class FileCache {
  private Map cache = new HashMap();
  private Executor executor = new PooledExecutor();

  public void get(final String name) {
    FutureResult result;

    synchronized(cache) {
      result = cache.get(name);
      if (result == null) {
        result = new FutureResult();
        executor.execute(result.setter(new Callable() {
          public Object call() { return loadFile(name); }
        }));
        cache.put(result);
      }
    }
    return result.get();
  }
}
```

This approach allows the first thread to get in and out of the synchronized block quickly, and allows other threads to have the result of the first thread's computation as quickly as the first thread does, with no chance of two threads both trying to compute the same object.

Summary

The `util.concurrent` package contains many useful classes, some of which you may recognize as better versions of classes you've already written, perhaps even more than once. They

are battle-tested, high-performance implementations of many of the basic building blocks of multithreaded applications. `util.concurrent` was the starting point for JSR 166, which will be producing a set of concurrency utilities that will become the `java.util.concurrent` package in JDK 1.5, but you don't have to wait until then. In a future article, I'll look at some of the custom synchronization classes in `util.concurrent`, and explore some of the ways in which the `util.concurrent` and `java.util.concurrent` APIs differ.

Resources

- Doug Lea's *Concurrent Programming in Java, Second Edition* is a masterful book on the subtle issues surrounding multithreaded programming in Java applications.
- Download the `util.concurrent` package.
- The javadoc page for `PooledExecutor` explains the various queuing, overflow, and shutdown options offered.
- JSR 166 is standardizing the `util.concurrent` library for JDK 1.5.
- Read all of the articles in Brian's *Java theory and practice* column. Specifically applicable to this article are July's article on thread pools and work queues and September's article on thread leakage.
- Find hundreds of Java technology-related resources at the *developerWorks* Java technology zone.

About the author

Brian Goetz

Brian Goetz is a software consultant and has been a professional software developer for the past 15 years. He is a Principal Consultant at Quiotix, a software development and consulting firm located in Los Altos, California. See Brian's published and upcoming articles in popular industry publications.