

Using java.util.concurrent

ICS432 - Fall 2008 Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Concurrency and Java

- More and more people write concurrent Java applications
 - Because more and more people write concurrent applications!
- We have seen all the “low level” tools for writing concurrent applications
- Since JS2E 5.0, Java provides a package called `java.util.concurrent`
- This package is intended to make life easier when writing concurrent applications
 - developed and peer-reviewed by concurrency experts
 - impact on the programmer: less code to write, more reliable, faster
- There are a LOT of things in it, with MANY options
- In this lecture we just give an overview
- Some argue that one should do all concurrency with it
- But you most likely will have to do both
 - Besides, understanding how things work at the low level is always a good idea, unfortunately

The Thread Pool Concept

- When writing concurrent applications, one often ends up spawning off many short-lived threads that do some useful tasks, throughout program execution
- Doing this by hand has several drawbacks
 - It requires code to be written
 - We now know how to do it, but it would be nice not to have to
 - Especially if threads have to be terminated
 - One may want to control the maximum number of threads that are running simultaneously to avoid overload
 - More code to bound the number and to wait for “free” slots
 - Creating threads is actually expensive, and it may be a better idea to keep threads “around” and reuse them
 - Still more code
- What we really need is a **Thread Pool**

The Thread Pool

- A Thread Pool is a (possibly fixed) set of threads
- Example:
 - I have a pool that contains 6 threads
 - I keep giving things to do to the pool
 - Up to 6 tasks can be running at a given time
 - Extra tasks are queued and will be done later when previous tasks have completed
- `java.concurrent.util` provides just the right thing here: **ExecutorService**
- Let's see an example

ExecutorService Interface

```
public class Task implements Runnable {
    private String message;
    private int iterations;

    public Task(String s, int n) {
        message = s; iterations = n;
    }

    public void run() {
        for (int i=0; i < iterations; i++) {
            System.out.println(message);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {}
        }
    }
}
```

```
import java.util.concurrent.*;

...
ExecutorService pool;
pool = Executors.newFixedThreadPool(3);

pool.execute(new Task("three",3));
pool.execute(new Task("two",2));
pool.execute(new Task("five",5));
pool.execute(new Task("six",6));
pool.execute(new Task("one",1));

pool.shutdown();
```

let's run it

ExecutorService Interface

- The `shutdown()` method prevents new tasks from being submitted, but running and submitted tasks finish
- The `shutdownNow()` method prevents new tasks from being submitted, but (attempts) to let only currently running tasks finish
- The `isTerminated()` method returns true if there is no pending task
- It is possible to create thread pools that can grow, and tons of other bells and whistles that you can discover on the on-line documentation

Atomic Variables

- `java.util.concurrent` also provides many simple classes of variable that can be updated atomically
- Say you want to write a program that maintains a shared counter
- You'll have to create a new class with synchronized methods for increment, decrement, etc.
- Almost all Java developers who write concurrent programs have done this and will do it again
- `java.util.concurrent` provides all this
- Let's see an example of class `AtomicInteger`

AtomicInteger

| Constructor Summary | |
|--|---|
| <code>AtomicInteger()</code> | Create a new <code>AtomicInteger</code> with initial value 0. |
| <code>AtomicInteger(int initialValue)</code> | Create a new <code>AtomicInteger</code> with the given initial value. |

| Method Summary | |
|--|---|
| <code>int addAndGet(int delta)</code> | Atomically add the given value to current value. |
| <code>boolean compareAndSet(int expect, int update)</code> | Atomically set the value to the given updated value if the current value == the expected value. |
| <code>int decrementAndGet()</code> | Atomically decrement by one the current value. |
| <code>int get()</code> | Get the current value. |
| <code>int getAndAdd(int delta)</code> | Atomically add the given value to current value. |
| <code>int getAndDecrement()</code> | Atomically decrement by one the current value. |
| <code>int getAndIncrement()</code> | Atomically increment by one the current value. |
| <code>int getAndSet(int newValue)</code> | Set to the give value and return the old value. |
| <code>int incrementAndGet()</code> | |
| <code>void set(int newValue)</code> | Set to the given value. |
| ... | ... |

Atomic Variables

- And others:
 - `AtomicBoolean`
 - `AtomicIntegerArray`
 - `AtomicLong`
 - `AtomicLongArray`
 - ...
- So, rather than re-implementing the wheel each time, using these classes from `java.util.concurrent` may be a better idea
 - Some people do not find them very readable and end up writing wrapper functions around them
 - Still, removes the need to deal with synchronized

Concurrent Abstract Types

- Beyond mere variables, the concurrent application developer often creates abstract data types that must be thread-safe
- Examples
 - `ConcurrentLinkedQueue`, `ConcurrentHashMap`, `BlockingQueue`
 - Implement all the expected operations
 - Implemented very efficiently, by experts, with maximum allowed concurrency with many individual monitors
 - some weakening of global operations (e.g., `size()`)
- Nowadays, re-implementing a thread-safe `Queue` or `Hash Table` is probably not advisable if the ones in `java.util.concurrent` do the job

Other Useful Things

- Semaphore
 - Just what you did in a programming assignment, probably implemented more efficiently
- Barrier
 - All threads wait for each other
- Locks and Semaphores
- Callable interface
 - Makes it simple to retrieve return values from tasks executed by a thread pool
- Many on-line Java Docs and Tutorials, which have good descriptions and examples