caffinc
Caffeine Induced Code and Ramblings

Share

Sriram Follow Creator of geeky things, mostly unused junk

0 Comments

Writing your own Threadpool from scratch in Java

Introduction

Multi-threaded and multi-process programming is a great way to optimize CPU usage and get things done quickly. The concept can be seen as the backbone to the larger distributed processing which occurs in systems like `Spark` and `Hadoop`.

Knowing how to write good multi-threaded code helps to easily scale up the performance of a program on one machine. However, maintaining a bunch of threads is like opening a can of worms. If you have a large can and a couple of worms, things work great. If you have a small can and tightly packed worms, once you pop open the lid, there's no easy way to keep things under control.

Luckily, threadpools offer a super-can with tame worms which don't crawl all over the place once opened. You can use threadpools to control how much CPU your multi-threaded program should use, and you can throw in as many threads (up to a limit) as you want and not have your CPU blow up.

Every time I see someone complain about multi-threading being unwiedly or too confusing, a part of me wants to know even more about threads and how they work. And what better way to learn than to write a Threadpool from scratch?

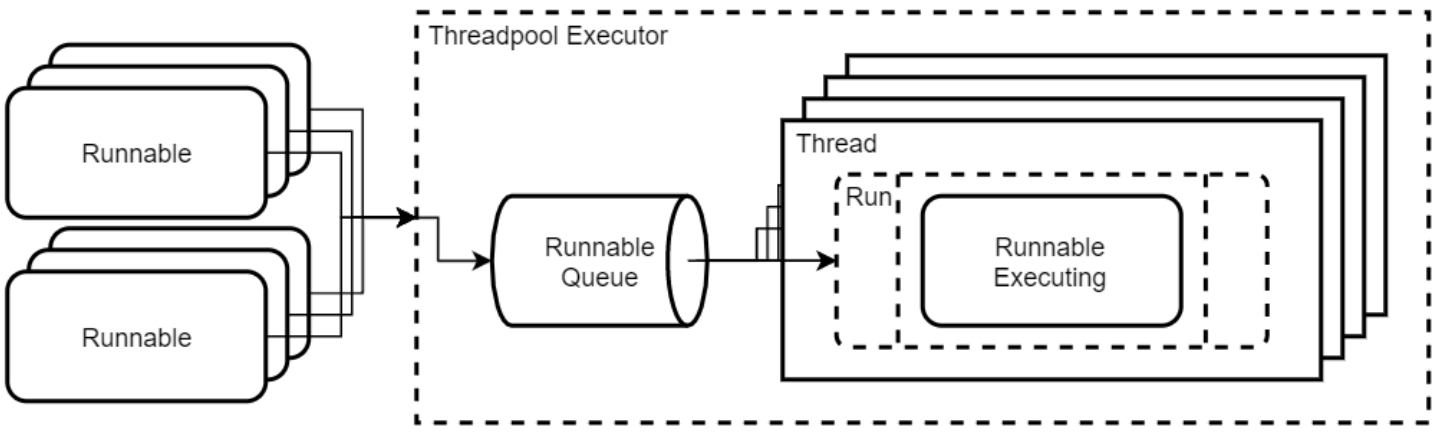
How they work

A long time ago when I was just starting my career, I had to use a connection pool called `C3P0`. Maybe everyone on the team was new to connection pools and we had a whole lot of trouble getting it to work correctly. We eventually figured it out, but since I wasn't actively working on it, I never found out what or how it worked. To me it was all a bit of magic.

Years later when I interview people with 5+ years of experience, and watch them wince when I mention Threads and Threadpools, it annoys me. So I decided to implement my own Threadpool to see just how hard it was to make one.

Turns out, Fixed Threadpools are rather easy to implement if you know what you're doing.

Let's look at *what* a threadpool is before the *how*:



So here we can see that the `Threadpool Executor` accepts `Runnable` objects and puts it into a `Runnable Queue`. This queue represents all the tasks that are sent to be executed by the `Threadpool`. The `Threadpool` itself is a bunch of threads that are waiting to pull out `Runnable`s from the queue and execute them in their own `run()` methods.

When the Threadpool is `running`, or in other words, the threads in the Threadpool are alive and ready to execute runnables, the `run()` method inside them is constantly polling the queue for any new objects. When there's a new Runnable in the queue, one of the threads pulls it out and calls the `run()` method of the Runnable.

In this way, we restrict the number of running threads to the number of threads set by the creator of the Threadpool. If you have 4 CPU threads in your machine, and only want your application to take a maximum of 50% (In practice, there are other threads spawned by the JVM, or other libraries that you might be using, which might cross this limit), you would restrict your pool to only run two threads at a time. However, if a Runnable being executed by the Thread in the Threadpool goes to sleep, that thread is effectively blocked until the Runnable decides that it's time to continue working again. There is no way to stop a Runnable, not cleanly.

Writing your own Threadpool

Now it's time to look at how you can make your own Threadpool from scratch. While this offers insight into how they work, it can also be used as a starting point for creating your own systems that cannot use one of the built-in Threadpools offered by Java.

The Queue

First, let's look at the Queue component. The responsibility of the Queue is to hold Runnables, and to have a way to poll them and check if the Queue is empty or not in order to help the threads in the pool utilize their resources better.

```
1 private ConcurrentLinkedQueue<Runnable> runnables;
```

A `ConcurrentLinkedQueue` is perfect for this task. It offers the `add()`, `poll()` and `isEmpty()` methods which are crucial for us. It is also thread-safe, which helps us create a thread-safe Threadpool :) It is backed by a `LinkedQueue` which helps ensure that we don't assign more memory than we need, and also allows our Threadpool to hold onto a lot of Runnables in memory when needed.

The Pool Thread

Next comes the component that is responsible for actually polling the Queue and executing the Runnables.

```
private class SimpleThreadpoolThread extends Thread {
    private AtomicBoolean execute;
    private ConcurrentLinkedQueue<Runnable> runnables;

    public SimpleThreadpoolThread(String name, AtomicBoolean execute,
ConcurrentLinkedQueue<Runnable> runnables) {
        super(name);
        this.execute = execute;
        this.runnables = runnables;
    }

    @Override
    public void run() {
        try {
            // Continue to execute when the execute flag is true, or when there are
runnables in the queue
            while (execute.get() || !runnables.isEmpty()) {
                Runnable runnable;
                // Poll a runnable from the queue and execute it
                while ((runnable = runnables.poll()) != null) {
                    runnable.run();
                }
                // Sleep in case there wasn't any runnable in the queue. This helps
to avoid hogging the CPU.
                Thread.sleep(1);
            }
        } catch (RuntimeException | InterruptedException e) {
            throw new ThreadpoolException(e);
        }
    }
}
```

The `SimpleThreadpoolThread` is the underlying Thread in our Threadpool which does the polling and execution. Since it is only used by our Threadpool, it is `private`.

In the constructor, we accept two parameters apart from the name:

1. `AtomicBoolean execute`: This is used for controlling the execution of the `while` loop inside the `run()` method. If the `execute` flag is `false`, and the Queue is empty, the Thread stops.
2. `ConcurrentLinkedQueue<Runnable> runnables`: This is the Queue which holds the runnables.

In the `run()` method, we have an outer loop which keeps the Thread alive for as long as the `execute` flag is `true` or the Queue has elements. In the inner loop, we poll the Queue for as long as there are elements inside it and call he `runnable.run()` method to execute the Runnable.

We also have a `Thread.sleep(1)` call in the outer loop. This is to prevent the Thread from behaving like an infinite loop and hogging all the CPU available to it. The loop enters this line only when the Queue is empty, which shouldn't happen very often in a well optimized system.

A bunch of these threads are started up and they form the Threadpool.

We now have the basic building blocks. Now let's look at the cement that holds it together:

```
public class SimpleThreadpool {
    // Count of threadpools created
    private static AtomicInteger poolCount = new AtomicInteger(0);
    // Queue of runnables
    private ConcurrentLinkedQueue<Runnable> runnables;
    // Flag to control the SimpleThreadpoolThread objects
    private AtomicBoolean execute;
    // Holds the "pool" of threads
    private List<SimpleThreadpoolThread> threads;

    /**
     * Thrown when there's a RuntimeException or InterruptedException when
executing a runnable from the queue, or awaiting termination
     */
    private class ThreadpoolException extends RuntimeException {
        public ThreadpoolException(Throwable cause) {
            super(cause);
        }
    }

    /**
     * Inner Thread class which represents the threads in the pool. It acts as a
wrapper for all runnables in the queue.
     */
    private class SimpleThreadpoolThread extends Thread {
        ...
    }

    /**
     * Private constructor to control the creation of threadpools. Increments the
poolcount whenever a new pool is created.
     *
     * @param threadCount Number of SimpleThreadpoolThreads to add to the pool
     */
    private SimpleThreadpool(int threadCount) {
        // Increment pool count
        poolCount.incrementAndGet();
        this.runnables = new ConcurrentLinkedQueue<>();
        this.execute = new AtomicBoolean(true);
        this.threads = new ArrayList<>();
        for (int threadIndex = 0; threadIndex < threadCount; threadIndex++) {
            SimpleThreadpoolThread thread = new
SimpleThreadpoolThread("SimpleThreadpool" + poolCount.get() + "Thread" +
threadIndex, this.execute, this.runnables);
            thread.start();
            this.threads.add(thread);
        }
    }

    /**
     * Gets a new threadpool instance with number of threads equal to the number
of processors (or CPU threads) available
     *
     * @return new SimpleThreadpool
     */
    public static SimpleThreadpool getInstance() {
        return getInstance(Runtime.getRuntime().availableProcessors());
    }

    /**
     * Gets a new threadpool instance with the number of threads specified
     *
     * @param threadCount Threads to add to the pool
     * @return new SimpleThreadpool
     */
```

```
66      public static SimpleThreadpool getInstance(int threadCount) {
67          return new SimpleThreadpool(threadCount);
68      }
69
70      /**
71       * Adds a runnable to the queue for processing
72       *
73       * @param runnable Runnable to be added to the pool
74       */
75      public void execute(Runnable runnable) {
76          if (this.execute.get()) {
77              runnables.add(runnable);
78          } else {
79              throw new IllegalStateException("Threadpool terminating, unable to
80  execute runnable");
81          }
82      }
83
84      /**
85       * Awaits up to <b>timeout</b> ms the termination of the threads in the
86  threadpool
87       *
88       * @param timeout Timeout in milliseconds
89       * @throws TimeoutException      Thrown if the termination takes longer than
90  the timeout
91       * @throws IllegalStateException Thrown if the stop() or terminate() methods
92  haven't been called before awaiting
93       */
94      public void awaitTermination(long timeout) throws TimeoutException {
95          if (this.execute.get()) {
96              throw new IllegalStateException("Threadpool not terminated before
97  awaiting termination");
98          }
99          long startTime = System.currentTimeMillis();
100         while (System.currentTimeMillis() - startTime <= timeout) {
101             boolean flag = true;
102             for (Thread thread : threads) {
103                 if (thread.isAlive()) {
104                     flag = false;
105                     break;
106                 }
107             }
108             if (flag) {
109                 return;
110             }
111             try {
112                 Thread.sleep(1);
113             } catch (InterruptedException e) {
114                 throw new ThreadpoolException(e);
115             }
116         }
117         throw new TimeoutException("Unable to terminate threadpool within the
118 specified timeout (" + timeout + "ms)");
119     }
120
121     /**
122      * Awaits the termination of the threads in the threadpool indefinitely
123      *
124      * @throws IllegalStateException Thrown if the stop() or terminate() methods
125 haven't been called before awaiting
126      */
127     public void awaitTermination() throws TimeoutException {
128         if (this.execute.get()) {
129             throw new IllegalStateException("Threadpool not terminated before
130 awaiting termination");
```

```
        }
        while (true) {
            boolean flag = true;
            for (Thread thread : threads) {
                if (thread.isAlive()) {
                    flag = false;
                    break;
                }
            }
            if (flag) {
                return;
            }
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                throw new ThreadpoolException(e);
            }
        }
    }

    /**
     * Clears the queue of runnables and stops the threadpool. Any runnables
currently executing will continue to execute.
     */
    public void terminate() {
        runnables.clear();
        stop();
    }

    /**
     * Stops addition of new runnables to the threadpool and terminates the pool
once all runnables in the queue are executed.
     */
    public void stop() {
        execute.set(false);
    }
}
```

In the constructor we create as many threads as requested and add them to a list. The constructor of our Threadpool is made `private` and we control the instantiation of the Threadpools by the `getInstance()` and `getInstance(int threadCount)` methods.

In the `getInstance()` method, we find the number of threads available to the process by calling the `Runtime.getRuntime().availableProcessors()` method.

The `execute(Runnable runnable)` method adds the Runnable object into the Queue.

We have two ways of stopping the Threadpool - `terminate()` or `stop()`. The `stop()` method sets the `execute` flag to false, and the Threadpool continues for as long as the Queue has runnables. The `terminate()` method is more abrupt as it clears the Queue and then calls `stop()`. If there are any executing Runnables, they continue to run until completion, but pending runnables do not execute.

In the end we have `awaitTermination()` and `awaitTermination(long timeout)` methods which blocks until the Threadpool has completed execution. The `awaitTermination()` method blocks indefinitely until termination of the pool, and

the `awaitTermination(long timeout)` throws a `TimeoutException` if the pool fails to terminate within the specified timeout period.

Seeing it in action

The Threadpool works the way it says on the box! You can find the actual code for the Threadpool, along with tests here: https://github.com/SriramKeerthi/SimpleThreadpool. I hope this helps you in your journey. Send me a message or comment below if you found this interesting or if you want more clarification!

17 Mar 2016

Java Threadpool

#java #queue #runnable #threadpool #threads

« CPU Load Generator in Java

Jersey Injection Source Error: Accepting user defined entities in your API »

uncategorized (2) mongodb (4) jenkins (3) java (29) android development (1)

api (5) swagger (1) mysql (1) redis (1) jetty (2) containerization (1) docker (2)

tensorflow (1) windows (2) virtualbox (1) github (1) amazon (2) aws (2)

influxdb (1) ubuntu (1) service (1) apache (1) reflection (1) junit (1) storm (5)

spark (6) scala (10) android (1) activator (1) cpu (1) threadpool (1) jersey (2)

maven (1) bintray (1) life (1) work (1) diskpart (1) disk management (1)

gparted (1) raspberry pi (1) sbt (1) functional programming (1)

distributed computing (1) remote procedure call (1) hsm (1) key variants (1)

bouncy castle (2) uber keystore (1) key check value (1) hardware security modules (1)

tools (2) variant (2) command (2)

Explore →

Theme Credits