4. Lecture 4 - Advanced multithreading

IT Learning &
Outsourcing Center

www.pragmatic.bg

Lector: Stefan Vadev
Skype: stefanvadev77
E-mail: stefan.vadev@gmail.com
Facebook:
https://www.facebook.com/stefan.vadev.7

Table of contents

- Advanced multithreading
- Executor Service, Thread pools
- java.util.concurrent package
- Exercises

Producer consumer

- In computing, the **producer–consumer** problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.

- The **producer**'s job is to generate data, put it into the buffer, and start again.
- At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

Problem

To make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Producer consumer

- **Solution**

The producer is to either go to sleep or discard data if the buffer is full.

- The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.
- An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

Producer consumer demo

Why use thread pools?

- Most of the executor implementations in **java.util.concurrent** use thread pools, which consist of worker threads. This kind of thread exists separately from the Runnable and Callable tasks it executes and is often used to execute multiple tasks.

- Using worker threads minimizes the overhead due to thread creation. Thread objects use a significant amount of memory, and in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead.

Why use thread pools?

- One common type of thread pool is the fixed thread pool. This type of pool always has a specified number of threads running; if a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread. Tasks are submitted to the pool via an internal queue, which holds extra tasks whenever there are more active tasks than threads.

Why use thread pools?

- An important advantage of the fixed thread pool is that applications using it degrade gracefully. To understand this, consider a web server application where each HTTP request is handled by a separate thread. If the application simply creates a new thread for every new HTTP request, and the system receives more requests than it can handle immediately, the application will suddenly stop responding to all requests when the overhead of all those threads exceed the capacity of the system. With a limit on the number of the threads that can be created, the application will not be servicing HTTP requests as quickly as they come in, but it will be servicing them as quickly as the system can sustain.

Executor service

- The java.util.concurrent.ExecutorService interface represents an asynchronous execution mechanism which is capable of executing tasks in the background.
- An ExecutorService is thus very similar to a thread pool. In fact, the implementation of ExecutorService present in the java.util.concurrent package is a thread pool implementation.

Executor service

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

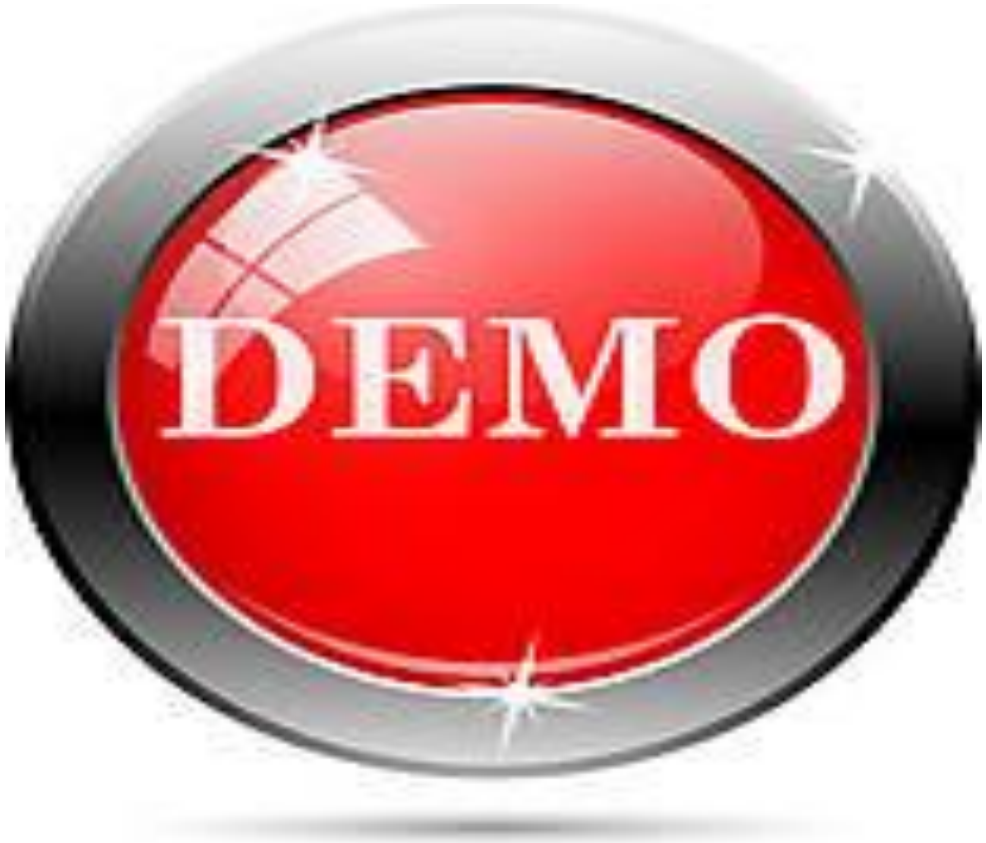
- For instance, if your application is started via a main() method and your main thread exits your application, the application will keep running if you have an active ExexutorService in your application. The active threads inside this ExecutorService prevents the JVM from shutting down.

Executor service

- To terminate the threads inside the ExecutorService you call its shutdown() method. The ExecutorService 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 ExecutorService before 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.

Executor service



Reentrant lock

Background

The traditional way to achieve thread synchronization in Java is by the use of synchronized keyword. While it provides a certain basic synchronization, the synchronized keyword is quite rigid in its use. For example, a thread can take a lock only once. Synchronized blocks don't offer any mechanism of a waiting queue and after the exit of one thread, any thread can take the lock. This could lead to starvation of resources for some other thread for a very long period of time.

Reentrant Locks are provided in Java to provide synchronization with greater flexibility.

Reentrant lock

The ReentrantLock class implements the Lock interface and provides synchronization to methods while accessing shared resources. The code which manipulates the shared resource is surrounded by calls to lock and unlock method. This gives a lock to the current working thread and blocks all other threads which are trying to take a lock on the shared resource.

As the name says, ReentrantLock allow threads to enter into lock on a resource more than once. When the thread first enters into lock, a hold count is set to one. Before unlocking the thread can re-enter into lock again and every time hold count is incremented by one. For every unlock request, hold count is decremented by one and when hold count is 0, the resource is unlocked.

Reentrant lock methods

lock(): Call to the lock() method increments the hold count by 1 and gives the lock to the thread if the shared resource is initially free.

unlock(): Call to the unlock() method decrements the hold count by 1. When this count reaches zero, the resource is released.

tryLock(): If the resource is not held by any other thread, then call to tryLock() returns true and the hold count is incremented by one. If the resource is not free then the method returns false and the thread is not blocked but it exits.

Reentrant lock methods

lock(): Call to the lock() method increments the hold count by 1 and gives the lock to the thread if the shared resource is initially free.

unlock(): Call to the unlock() method decrements the hold count by 1. When this count reaches zero, the resource is released.

tryLock(): If the resource is not held by any other thread, then call to tryLock() returns true and the hold count is incremented by one. If the resource is not free then the method returns false and the thread is not blocked but it exits.

Reentrant lock methods

tryLock(long timeout, TimeUnit unit): As per the method, the thread waits for a certain time period as defined by arguments of the method to acquire the lock on the resource before exiting.

lockInterruptibly(): This method acquires the lock if the resource is free while allowing for the thread to be interrupted by some other thread while acquiring the resource. It means that if the current thread is waiting for lock but some other thread requests the lock, then the current thread will be interrupted and return immediately without acquiring lock.

Reentrant lock methods

getHoldCount(): This method returns the count of the number of locks held on the resource.

isHeldByCurrentThread(): This method returns true if the lock on the resource is held by the current thread.

Reentrant lock demo

Atomic variables

The **java.util.concurrent.atomic** package defines classes that support atomic operations on single variables. All classes have get and set methods that work like reads and writes on volatile variables.

That is, a set has a happens-before relationship with any subsequent get on the same variable. The atomic compareAndSet method also has these memory consistency features, as do the simple atomic arithmetic methods that apply to integer atomic variables.

Atomic variables

AtomicBoolean
AtomicInteger
AtomicLong
AtomicReference
AtomicStampedReference
AtomicIntegerArray
AtomicLongArray
AtomicReferenceArray

Concurrent data structures

Concurrent Collections
The java.util.concurrent package includes a number of additions to the Java Collections Framework. These are most easily categorized by the collection interfaces provided:

BlockingQueue defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue.

ConcurrentMap is a subinterface of java.util.Map that defines useful atomic operations. These operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent. Making these operations atomic helps avoid synchronization. The standard general-purpose implementation of ConcurrentMap is ConcurrentHashMap, which is a concurrent analog of HashMap.

Concurrent data structures

Concurrent Collections
The java.util.concurrent package includes a number of additions to the Java Collections Framework. These are most easily categorized by the collection interfaces provided:

BlockingQueue defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue.

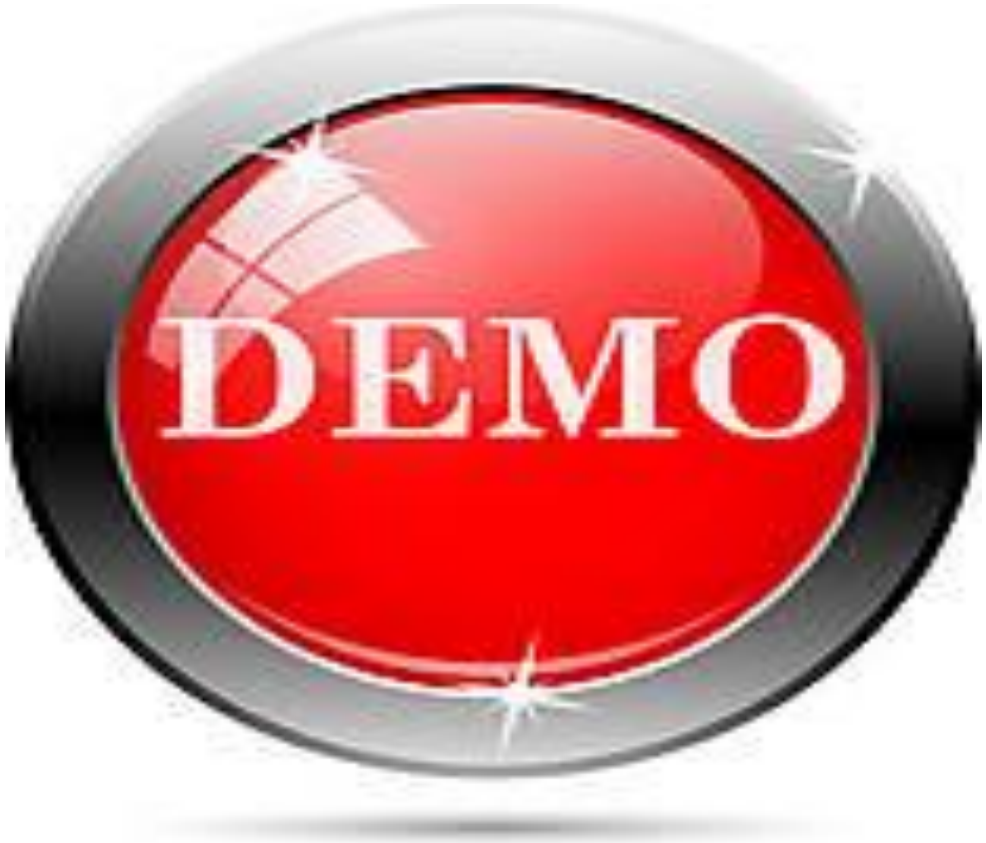
ConcurrentMap is a subinterface of java.util.Map that defines useful atomic operations. These operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent. Making these operations atomic helps avoid synchronization. The standard general-purpose implementation of ConcurrentMap is ConcurrentHashMap, which is a concurrent analog of HashMap.

Concurrent data structures

ConcurrentNavigableMap is a subinterface of **ConcurrentMap** that supports approximate matches. The standard general-purpose implementation of **ConcurrentNavigableMap** is **ConcurrentSkipListMap**, which is a concurrent analog of **TreeMap**.

All of these collections help avoid **Memory Consistency Errors** by defining a happens-before relationship between an operation that adds an object to the collection with subsequent operations that access or remove that object.

Data structures demo



Useful resources

https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

http://tutorials.jenkov.com/java-concurrency/index.html

https://www.tutorialspoint.com/java/java_multithreading.htm

Table of contents

- Advanced multithreading
- Executor Service, Thread pools
- java.util.concurrent package
- Exercises