Java Concurrency

lecture notes

Anton Shchastnyi

11/24/2011

A brief overview of the Java Concurrency. Main definitions, liveness issues, high level concurrency objects of the Java Platform, Standard Edition.

Table of Contents

Contents

Concurrency Definitions

Concurrency is the ability to run several parts of a program in parallel.
Concurrency can highly improve the speed of a program if certain tasks could be performed asynchronously or in parallel.

Process VS Thread

- ***Process***: A process runs independently and isolated of other processes. It cannot directly access shared data in other processes. The resources of the process are allocated to it via the operating system, e.g. memory and CPU time.

- ***Thread***: Threads are so called ***lightweight processes*** which have their own call stack but can access shared data. Every thread has its own memory cache. If a thread reads shared data it stores this data in its own memory cache. A thread can re-read the shared data, when this happens in Java will be explained in Java memory model part of this article.

Concurrency in Java

A Java application runs in its own process.
Within a Java application you work with ***several threads*** to achieve parallel processing or asynchronously behavior.

Java supports threads as part of the Java language. Java 1.5 also provides improved support for concurrency with the in the package ***java.util.concurrent***.

Java also provides ***locks*** to protect certain parts of the coding to be executed by several threads at the same time.

The simplest way of locking a certain method or Java class is to use the keyword "***synchronized***" in a method declaration.

The Java Memory Model

The Java memory model describes

- the communication between the memory of the threads and the main memory
- which operations are ***atomic***
- the ordering of the operations.

Atomic Operation

An atomic operation is an operation which is performed as a ***single unit of work*** without the possibility of interference from other operations.

In Java the language specification guarantees that that reading or writing a variable is atomic (unless the variable is of type ***long*** or ***double***). Long and double are only atomic if they declared as volatile.

Volatile

The ***volatile*** modifier tells the JVM that writes to the field should always be synchronously flushed to memory, and that reads of the field should always read from memory.

If a variable is declared as volatile then is guaranteed that any thread which reads the field will see the most recently written value.

This means that fields marked as volatile can be safely accessed and updated in a multi-thread application without using native or standard library-based synchronization.

(This does not apply to ***long*** or ***double*** fields, which may be non-atomic on some JVMs. However, it always applies to reference-typed fields.)

The operation ***i++*** is ***not atomic*** in Java for the primitives.
It first reads the value which is currently stored in **i** (atomic operations).
Then it increments it (atomic operation). But between the read and the write the value of **i** might have changed.

Since Java 1.5 the java language provides ***atomic variables***	
AtomicInteger, ***AtomicLong***	atomic methods: ***getAndDecrement****(),* ***getAndIncrement****()* ***getAndSet****()*
BigDecimal is only about to be more precise than Float or Double	

Nonblocking Algorithms

Java 5.0 provides supports for additional atomic operations. This allows to develop algorithm which are ***non-blocking algorithm***, e.g. which do not require synchronization, but are based on low-level atomic hardware primitives such as ***compare-and-swap (CAS)***.

A CAS operation checks if the variable has a certain value and if it has this value it will perform this operation.

Non-blocking algorithm are usually much faster than blocking algorithms as the synchronization of threads appears on a much lower level (hardware).

The JDK itself extensively uses non-blocking algorithms to increase the platform performance. Developing correct non-blocking algorithm is not a trivial task.

Liveness Issues

Deadlock
Deadlock describes a situation where two or more threads are blocked forever, waiting for each other.

Livelock
A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then ***livelock*** may result.

As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work.

This is comparable to two people attempting to pass each other in a corridor: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass. Seeing that they are still blocking each other, Alphone moves to his right, while Gaston moves to his left. They're still blocking each other, so...

Starvation
Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads.

For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

Guarded Blocks
Threads often have to coordinate their actions. The most common coordination idiom is the ***guarded block.***

Such a block begins by polling a condition that must be true before the block can proceed.

Immutable Objects
An object is considered ***immutable*** if its ***state cannot change after it is constructed***.

Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code.

Immutable objects are particularly useful in concurrent applications. Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state.

High Level Concurrency Objects

Synchronized blocs and ***monitors*** with java.lang.Thread class are adequate for very basic tasks,
but ***higher-level building blocks*** are needed for more advanced tasks.

This is especially true for massively concurrent applications that fully exploit today's multiprocessor and multi-core systems.

In this section we'll look at some of the high-level concurrency features introduced with Java 5.0.
Most of these features are implemented in the new ***java.util.concurrent*** packages. There are also new concurrent data structures in the ***Java Collections Framework***.

Lock objects	Support locking idioms that simplify many concurrent applications.
Executors	Define a high-level API for launching and managing threads. Executor implementations provided by ***java.util.concurrent*** provide thread pool management suitable for large-scale applications.
Concurrent collections	Make it easier to manage large collections of data, and can greatly reduce the need for synchronization.
Atomic variables	Have features that minimize synchronization and help avoid memory consistency errors.
ThreadLocalRandom	Provides efficient generation of pseudorandom numbers from multiple threads (in JDK 7).

Locks

Lock implementations provide more extensive locking operations than can be obtained using *synchronized methods and statements*.
Locks allow more flexible structuring, may have quite different properties, and may support multiple associated Condition objects.

With this increased flexibility comes additional responsibility. The absence of block-structured locking removes the automatic release of locks that occurs with synchronized methods and statements. In most cases, the following idiom should be used:

```
Lock l = ...;
l.lock();
try {
    // access the resource protected by this lock
} finally {
    l.unlock();
}
```

Lock implementations provide additional functionality over the use of synchronized methods and statements.

A Lock class can also provide behavior and semantics that is quite different from that of the implicit monitor lock, such as guaranteed ordering, non-reentrant usage, or deadlock detection. If an implementation provides such specialized semantics then the implementation must document those semantics.

java.util.concurrent.locks

Interface Lock

Method Summary	
void	**lock**() Acquires the lock (similar to entering the ***synchronized*** section).
void	**lockInterruptibly**() Acquires the lock unless the current thread is interrupted.
Condition	**newCondition**() Returns a new Condition instance that is bound to this Lock instance.
boolean	**tryLock**() Acquires the lock only if it is free at the time of invocation.
boolean	**tryLock**(long time, TimeUnit unit) Acquires the lock if it is free within the given waiting time and the current thread has not been interrupted.
void	**unlock**() Releases the lock.
All Known Implementing Classes	
	ReentrantLock, ReentrantReadWriteLock.ReadLock, ReentrantReadWriteLock.WriteLock

Executors

In large-scale applications, it makes sense to separate thread management and creation from the rest of the application. Objects that encapsulate these functions are known as ***executors***.

Executors Types

Executor Interfaces	***Executor***	A simple interface that supports launching new tasks. ***(new Thread(r)).start();*** → ***e.execute(r);***
	ExecutorService	A subinterface of Executor, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself. `execute()` accepts ***Runnable***; `submit()` accepts ***Runnable*** and ***Callable.*** Returns a Future object, which is used to retrieve the Callable return value and to manage the status of both Callable and Runnable tasks. Provides methods for submitting large collections of Callable objects. Provides a number of methods for managing the shutdown of the executor. To support immediate shutdown, tasks should handle interrupts correctly.
	ScheduledExecutorService	A subinterface of ExecutorService, supports future and/or periodic execution of tasks. The ScheduledExecutorService interface supplements the methods of its parent ExecutorService with schedule, which executes a Runnable or Callable task after a specified delay. In

		addition, the interface defines scheduleAtFixedRate and scheduleWithFixedDelay, which executes specified tasks repeatedly, at defined intervals.
Thread Pools		The most common kind of executor implementation.
Fork/Join		A framework (new in JDK 7) for taking advantage of multiple processors.

Futures and Callables

Callable<V> method: V **call**() throws Exception Computes a result, or throws an exception if unable to do so.	A task that returns a result and may throw an exception. Implementors define a single method with no arguments called ***call()***. The Callable interface is similar to Runnable, in that both are designed for classes whose instances are potentially executed by another thread. A Runnable, however, does not return a result and cannot throw a checked exception. The Executors class contains utility methods to convert from other common forms to Callable classes.

java.lang.Thread VS the Executor Framework

Using Threads directly has the following disadvantages:

- Creating a new thread causes some performance overhead
- Too many threads can lead to reduced performance, as the CPU needs to switch between these threads
- You cannot easily control the number of threads, therefore you may run into out of memory errors due to too many threads

The ***java.util.concurrent*** package offers improved support for concurrency compared to threads.

java.lang.Thread	***Threads pools with the Executor Framework***

new Thread(new(RunnableTask()).start()
for each of a set of tasks
	Executor executor = *anExecutor*;
	executor.execute(new RunnableTask1());
	executor.execute(new RunnableTask2());

Thread pool manages a pool of worker threads. The thread pool contains a work queue which holds tasks waiting to get executed.

The Executor framework provides example implementation of the ***java.util.concurrent.Executor***.
The ***ExecutorService*** adds lifecycle methods to the Executor, which allows to shutdown the Executor and to wait for termination.

Thread Pools

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.

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.

Executor	An object that executes submitted Runnable tasks
ExecutorService	An Executor that provides methods to manage termination and methods that can produce a Future for tracking progress of one or more asynchronous tasks. ***All Known Implementing Classes:*** AbstractExecutorService, ScheduledThreadPoolExecutor, ThreadPoolExecutor
	The ThreadPoolExecutor class provides an extensible thread pool implementation. The Executors class provides convenient factory methods for these Executors.

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.

java.util.concurrent.Executors class	***Methods to create an executors that use threas pools***
newFixedThreadPool()	Creates an executor with a ***fixed thread pool***.
newCachedThreadPool()	Creates an executor with an ***expandable thread pool***. This executor is suitable for applications that launch many short-lived tasks.
newSingleThreadExecutor()	Creates an executor that ***executes a single task at a time***.

If none of the executors provided by the above factory methods meet your needs, constructing instances of java.util.concurrent.ThreadPoolExecutor or java.util.concurrent.ScheduledThreadPoolExecutor will give you additional options.

Fork/Join in Java 7

New in the Java SE 7 release, **the *fork/join framework*** is an implementation of the ***ExecutorService*** interface that helps you take advantage of multiple processors.

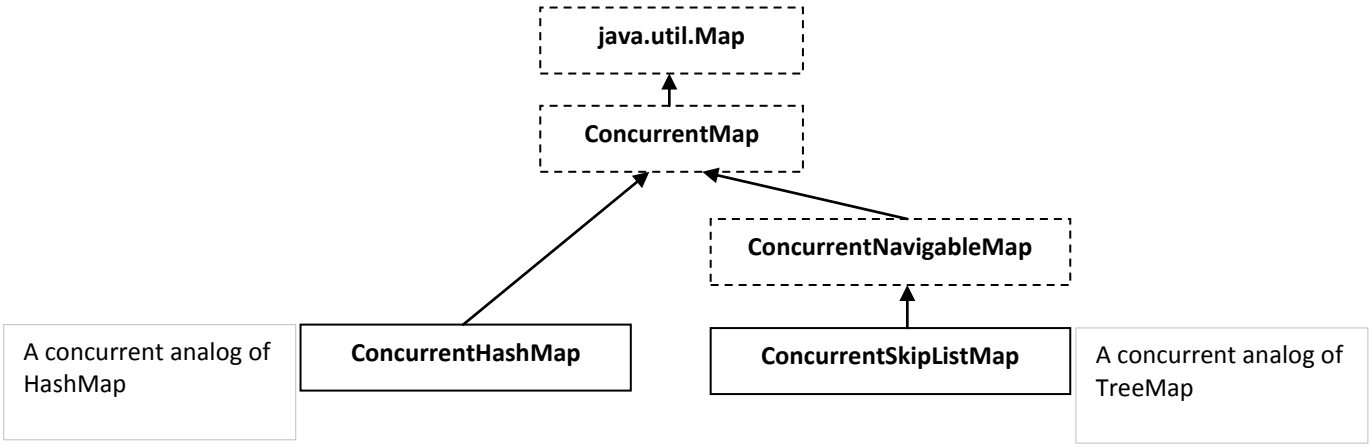
The fork/join framework allows you to distribute a certain task on several workers and then wait for the result. It is designed for work that can be broken into smaller pieces recursively. The goal is to use all the available processing power to make your application wicked fast.

For Java 6 you have to add this framework manually (jsr166.jar from http://g.oswego.edu/dl/concurrency-interest/).

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 FIFO data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue.
ConcurrentMap	The interface 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.
ConcurrentNavigableMap	This interface supports approximate matches.



References

[1] Lars Vogel, *Java Concurrency / Multithreading – Tutorial*,
http://www.vogella.de/articles/JavaConcurrency/article.html

[2] Oracle, Inc, *Java Tutorials – Concurrency*,
http://download.oracle.com/javase/tutorial/essential/concurrency/

[3] B. Goetz, with T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, D. Lea, *Java Concurrency in Practice*,
http://www.informit.com/store/product.aspx?isbn=0321349601

About the Author

Anton Shchastnyi is a software engineer living in Ukraine. He mainly focuses on developing web and desktop applications on Java platform. He also enjoys writing, and does quite a bit of tutorials and learning materials on Java and related technologies (http://antonshchastnyi.blogspot.com/, http://schaan.habrahabr.ru/blog).