Java Concurrency

MAÁR ZOLTÁN

OCTOBER 1, 2016

Contents

1. Java memory model
2. Starting and stopping threads
3. Executors and Futures
4. Blocking synchronization
5. Non-blocking constructs

Why do we write concurrent applications?

Java memory model - Definitions

Thread

- A sequence of instructions, that may execute in parallel with others

Race condition

- When correct operation depends on the timing or the sequence of threads

Critical section

- Where race conditions might occur

Thread-safe

- Free of race condition

Java memory model – JMM

Visibility

- Defines under what conditions sees a thread the changes made by another thread to a shared variable

Ordering

- Defines ordering („within-thread as-if-serial”) and „happens before”

Guarantee

- within-thread as-if-serial: any reordering of statements is possible, as long as the result of the thread run in **isolation** is same as the statements would have been executed in program order

Java memory model – JMM – Ordering

Reordering

- Thread 1: a=1; b=1
- Thread 2: print(“” + a + b); // 00, 10, 11 but no 01 expected

Order of executions

- print; a = 1; b = 1; => 00
- a = 1; print; b = 1; => 10
- a = 1; b=1; print; => 11
- ~~b = 1; print; a = 1~~

Java memory model – JMM – Ordering

Happens-Before

- “To guarantee that the thread executing action B can see the results of action A (whether or not A and B occur in different threads), there must be a happens before relationship between A and B.” (JCIP)

Memory barriers

- Memory barriers either prevent out-of-order execution (by the CPU) of memory operations or
- prevent reordering of instructions (by the compiler)
- volatile, ...

Java memory model – JMM – Visibility

Atomic operations on variables

- Loading and storing of a single 32-bit quantity is atomic
- For 64-bit quantities, atomicity is not defined in the JVM specification
- Read and write of object references are always atomic

Initialization safety of final fields

- Once an object is properly constructed,
  - All threads will see the proper values of the final fields, without the need of additional synchronization
  - Any variables can be reached only through final fields are also guaranteed to be visible to other threads

Java memory model – JMM – Visibility

Immutable objects

- If an object can’t be changed then there is nothing to guard with synchronization. Immutable object:
- It’s state can’t be changed after construction
- Perfect for sharing data between threads
- All fields should be final, …, Anything else?

Not thread safe classes

- MessageFormat / DateFormat
- Matcher (but Pattern is thread-safe)
- Random – use ThreadLocalRandom instead, from Java 7
- Basic collections
- Date, Calendar

Contents

1 Java memory model

2 Starting and stopping threads

3 Executors and Futures

4 Blocking synchronization

5 Non-blocking constructs

Threads – Starting and stopping threads

Starting Threads

- Threads are always started with Thread.start()
- Submitting tasks to an Executor not always starts execution in a new thread

Stopping Threads

- Non-daemon threads are stopped when their run() method is finished
- Daemon threads are stopped when all other non-daemon threads are stopped
- Thread.stop() depricated

Proper ways to stop a thread

- React to interruptions / Poison pill if reading from a queue
- Thread.interrupt() - Thread.interrupted() / Thread.isInterrupted()
- Handling InterruptedException:
  1. Clean-up if required, 2. Set interrupted flag and/or re-throw InterruptedEx
- Thread.setUncaughtExceptionHandler useless with thread pools

Contents

1 Java memory model

2 Starting and stopping threads

3 Executors and Futures

4 Blocking synchronization

5 Non-blocking constructs

Executor Framework

Executor

- Top level interface
- Supports only Runnable, No life-cycle handling

ExecutorService

- Life-cycle, Future, Multiple task support
- Happens-Before:
  Actions before submit <= Actions in the task <= Actions after Future.get()
- Implementations: ThreadPool, ScheduledThreadPool, ForkJoinPool

Shutdown

- Cannot be restarted
- Shutdown is important otherwise JVM won’t exit
- Gracefully:shutdown(); awaitTermination(); shutdownNow(); awaitTermination()
- Tasks should react to interruptions

Executor Framework

Start task

- executorService.submit(): Runnable, Callable
- executorService.execute(): fire and forget
- For Runnables, it is possible to define a default result (submit parameter)
- Future.get() can be used instead of Thread.join() to wait for tasks to finish

Create ExecutorServices

- Executors provides static factory methods for the more common use cases
- ThreadFactory
- RejectedExecutionHandler
- Spring wrappers do not need explicit shutdown

Executor Framework – Getting results from tasks

Simple Thread subclasses

- Use a shared, thread-safe data structures

ExecutorService

- Has got couple of methods for querying the status of the task
- Future.get()
  May or may not block
  Throws ExecutionException if the task threw an exception
  Should be called, otherwise exceptions from your task won't be propagated to the caller
- Future.cancel(mayInterruptIfRunning)
  Can be used to stop running tasks, if task reacts properly to interrupts

CompletionService

- Decouples task creation and result processing
- Results become available as tasks complete

Contents

1 Java memory model

2 Starting and stopping threads

3 Executors and Futures

4 Blocking synchronization

5 Non-blocking constructs

Blocking synchronization – Definitions

Contention

- Whenever one thread attempts to acquire a lock held by another thread

Deadlock

- Two or more threads are blocked forever, waiting for each other

Starvation

- Thread is unable to gain regular access to shared resources and unable to progress

Livelock

- Threads are continuously responding to each others actions and not progressing

Context switch

- Storing a state of process or thread so its execution can be resumed at a later time

Re-entrancy

- A thread may acquire the same lock multiple times. It must release it exactly the same times it acquired it.

Blocking synchronization – Intrinsic Locks

Synchronized keyword

- Tries to acquire the monitor of the associated object
- static method: the class instance/instance method: this/parameter
- blocks until monitor becomes available
- Encapsulate synchronization: use a private Object instance to lock on
  Users of your class can’t mess with the synchronization

Waiting for conditions

- Object.wait() & wait(timeout)
- Object.notify() and notifyAll()
- Both methods must be called while the object’s monitor is held

Blocking synchronization – Explicit Locks

Explicit locks

- Important interfaces: Lock, Condition and ReadWriteLock
- Default implementation: ReentrantLock
- All intrinsic locking operations can be mapped to explicit ones

Waiting for conditions

- Allows fairness but at a performance cost
- Support for multiple Conditions
  Threads will be notified only when the condition they're interested is signaled
  Checking in a loop is still required
- tryLock(): tries to get lock, returns immediately regardless the result
- lockInterruptibly() – threads can react to interrupts while trying to acquire a lock

Blocking synchronization – More

ReentrantReadWriteLock

- Composite-like structure with two locks

StampedLock – Java 8

- Three modes: Write, Read, Optimistic Read

Semaphore

- Has got a fixed numbers or permits which can be acquired or released

CountDownLatch

- One-shot synchronization point for a fixed number of threads.

CyclicBarrier

- Provides a common, reusable „meeting point” to a fixed number of threads. Executes a predefined action when all parties arrive at the barrier.

Phaser

- Similar to CyclicBarrier, but the number of participants can be changed dynamically. Multiple Phasers can be organized into a tree to reduce contention.

Contents

1 Java memory model

2 Starting and stopping threads

3 Executors and Futures

4 Blocking synchronization

5 Non-blocking constructs

Non-blocking constructs

Volatile modifier on fields

- Ensures visibility of changed values among multiple threads
- No synchronization is performed, not atomic
  - For single writers/multiple readers, this is not an issue
  - Can be used with multiple writers only when the next value does not depend on the current one (timestamps, for example)
  - Problematic cases: i++, b = !b, multiple fields

Guarantees

- Visibility: every write happened before a volatile write on the same thread, is visible after a volatile read on another thread
- Ordering: volatile reads/writes are not reordered
- Missed updates are possible with multiple writers (for example shared counter)

Non-blocking constructs

Atomic classes

- Atomic[Integer|Long|Reference]<Array>
- CAS is atomic (CPU instructions)
- Writes are atomic and happen only if the current value matches the expected one
- Usually performs better when contention is low
- ABA problem: AtomicStampedReference

Methods

- compareAndSet(old,new) - usually used in a loop Same memory effect as volatile read and write
- get() / set(value) - same memory effect as volatile read or write
- lazySet(new) - writes are not reordered, but subsequent reads might be
- weakCompareAndSet() - no ordering guarantees, but CAS is atomic

Non-blocking constructs

Java 8 Atomic classes

- Support for contended writes, slow reads

Long/DoubleAccumulator

- Maintains a running value
- Updated by a supplied function

Long/DoubleAdder

- Maintains a running sum
- Starts from zero
- Useful for various statistics, updated from many threads

• Thank you

- Books:
  Java Concurrency In Practice – Brian Goetz
  Concurrent Programming in Java™: Design Principles and Pattern – Doug Lea
  The Art of Multiprocessor Programming - Maurice Herlihy
- Blogs
  http://mechanical-sympathy.blogspot.hu/
  http://bad-concurrency.blogspot.hu/
  http://psy-lob-saw.blogspot.com/
  https://www.infoq.com/