Advanced Concurrency Topics

We talked about low-level concepts and tools to manage concurrency in java.

The Thread class is placed in the java.lang package.

There are some High-level APIs to help us manage concurrency that were added since Java5 and are available in the **java.util.concurrency** package

These APIs are usually a better approach to handle concurrency compared to those old, low-level tools that we saw in the first document:

* They **usually** have a better performance, they don’t simply make everything synchronized for example.
* They can make a better use of modern multi-core CPUs that can perform parallel processing
* They are easier to use
* They offer different solutions to different use cases and for example you’re not left with only the synchronized keyword to handle concurrency.

# Thread-Safety and Thread-Safe Classes

Instances of classes that are thread safe can be used as a shared resource in multiple threads in a safe way without causing a synchronization problem.

To use the instances of these classes you won’t need to use locks, synchronized block/methods, etc.

They are either inherently thread-safe like immutable classes, or all the precautions are built in these classes and we won’t be thinking about them like ConcurrentHashmap.

**Every time you want to use a thread-safe, utility class you have to read the documentation and use the methods that are thread-safe because these classes can have some methods that don’t do their job in a thread-safe way.**

**You must also check for how they acquire locks for example, do they block on read operations?**

Examples of such classes are:

|  |  |
| --- | --- |
| Non-Thread-Safe Class | The Thread-Safe Equivalent |
| ArrayList | Vector |
| HashMap | ConcurrentHashMap |
| StringBuilder | StringBuffer |

**Some notes:**

* As you know the String class is thread-safe cause it’s immutable. The StringBuffer and StringBuilder are mutable classes for manipulating strings. The StringBuffer class makes its instances thread-safe by making its methods synchronous. However, there **are still some non-thread safe methods in this class that aren’t synchronized**.
* So, pay attention to the source code of the methods you use on this kind of thread-safe classes
* Some of the synchronized methods are an overridden form of their parent interface that don’t have the synchronized key-word in their signature. This was interesting

## Thread-Safe Classes or Normal Classes, which One to Use?

Why don’t we just always use thread-safe classes?

* First of all, we should know that immutable classes like *String* or *Integer* are inherently thread-safe since they’re always read not modified. So, in that sense they are not different from normal classes
* But when it comes to classes that provide a support for concurrency by for example synchronized blocks or other approaches, they have a worse performance due to these precautions
* So, if you don’t need to use an object as a shared resource where multiple threads will work with it, you better off using the normal classes

# Concurrent Collections

Side notes:

Maps are not subtypes of the Collection interface but I think they’re all part of the Java Collection API, I’m like 10% sure about this.

* A simple way to make a class thread-safe is to make all the methods synchronized but this certainly won’t have a good performance
* For example, you could only use one of the methods at a time as they all need to acquire the lock to this
* The java.util.concurrency offers data-structures that are thread-safe, and implement this thread-safety in an efficient way with decent performance in concurrent applications
* for example, they may acquire/release locks in a more efficient way

some of the examples are: **ArrayBlockingQueue, CopyOnWriteArrayList, ConcurrentHashMap**

## BlockingQueue

There are multiple implementations of this interface like *ArrayBlockingQueue* that uses an array as the underlying data structure and LinkedBlockingQueue that uses linked lists.

They are basically queues that are thread-safe an provide some additional handy features that you will get when you use the ***put()*** and ***take()*** methods to insert and remove data:

If you try to read/remove from an empty queue the thread waits until the queue has a value in it

If you try to insert into a full queue, the thread waits until there is room.

So basically, this will help you solve the Producer/Consumer Problem with ease.

## ConcurrentHashMap

In my quick investigation that **needs more attention and time**, I realized that concurrent hash maps don’t acquire locks on retrievals(it needs more introspection) so they’re a highly-concurrent implementation, on the other hand, HashTables simply lock the entire table for reads and writes so you may want to use ConcurrentHashMaps where you need a stricter synchronization

# Synchronizers

A synchronizer object is used to control the execution of multiple threads.

These objects have **an inner state** which is used to manage the involved threads to let them wait or resume execution for example.

* Note that there are some nuances to each one of these classes and you should read the API document to use them correctly

There are some utility classes to facilitate this process that I think are alternatives to methods like wait() and notify():

## Semaphore

* It controls the number of threads that are accessing a shared resources **through using a counter**
* the counter maintains the number of available permits
* When a thread wants to access the shared resource, it calls the acquire() method, if the counter is greater than 0, a permit is granted and the counter reduces by 1
* When a thread is done with the shared resource, it calls release() method, the counter is incremented by1, and the permit will be available for others to acquire
* If the counter number is **not** greater than 0, a thread will be blocked upon trying to acquire a permit until a permit can be acquired
* A binary Semaphore is a semaphore with only two states, either one permit is available or non. This kind of semaphore can act as a mutually exclusive lock.

In this case you set the initial permits to 1, at the beginning of each thread you acquire a permit and in the end you release it, this way no two threads can acquire a permit at the same time and there will be a MUTEX.

* You can set a fairness Boolean on the semaphore objects and by setting that to true it kind of respects the order the threads has called the acquire method so the sooner a thread calls the acquire method, the sooner it will granted a permit**( but this one needs more investigation**).
* The number you pass to the constructor is the initial number of permits not the maximum number of permits
* You can for example set it to 0 so no one can acquire a permit until somebody has released a permit
* There is no need for a thread to first acquire a permit to be able to release a permit. The initial permits for example can be negative, so multiple releases must occur before any thread can acquire a permit. The contract on which methods acquire permits and which methods release permits are established by the application.

Let’s see an example of using a semaphore in solving the producer/consumer problem:

We are going to address two challenges in the P/C problem: first we need to make sure no two threads are simultaneously accessing the shared queue, and second we need to wait for the queue to have some value before we can remove from it:

The first challenge can be solved with using the synchronized keyword.

The second one means that there has to be at least one permit for a thread to be able to remove form the queue. This is solved by setting the initial number of permits to 0, and each remove operation has to acquire a permit. As the initial permit is set to 0, no one can remove form the least unless we release a permit on an add operation:

Semaphore sem = new Semaphore(0);

sem.acquire()

synchronized (list){ synchronized (list){

list.add(int); list.remove()

} }

sem.release();

Question: why are the acquire() and release() methods outside the synchronized blocks?

Because if they were inside it, nobody else could add something to the list while another one is waiting for it to have value.

## CountDownLatch

A synchronization aid that allows one or more threads to wait until **a set of operations** being performed in **other** threads completes.

A CountDownLatch is initialized with a given count. The ***await()*** methods block until the current count reaches zero due to invocations of the ***countdown()*** method, after which **all** waiting threads are released and **any subsequent invocations of await return immediately**.

**This is a one-shot phenomenon** -- **the count cannot be reset**. If you need a version that resets the count, consider using a CyclicBarrier

A CountDownLatch initialized with a count of 1 will simply act as an on/off latch or a gate where all the threads calling await have to wait until one thread calls the countdown method.

The example below shows a thread that has to wait until two other threads call latch.countDown()

CountDwonLach latch = new CountDownLatch(2);

Thread1:

Latch.await()

Do sth…

An example use case is when you want to make sure that two other processes have finished before you execute a task. Using a CountDownLatch you won’t have to pass the thread objects that are running those two to the waiting thread and call join() on them. You implement the synchronization using a CountDownLatch that has an initial value of 2 and in the waiting thread you’ll call await() on this object.

## Exchanger

A synchronization point at which threads can **pair** and **swap elements** **within pairs**. Each thread presents some object **on entry to the exchange method**, matches with a partner thread, and receives its partner's object on return.

1. A thread calls the exchange() method on an Exchanger object and passes a value to it
2. This thread **waits** for another thread to call the exchange() method on the same Exchanger object unless it( the current thread) gets interrupted
3. When the second thread calls the exchange() method, the first thread then transfers the given object to it, receiving its object in return.
4. Two threads then continue execution

Exchanger<String> exchanger = new Exchanger<>();

Thread1 --------- > exchanger.exchange(“x=2”)

Thread2 --------- > exchanger.exchange(“y=2”)

**Question: why exchangers are a good option for implementing pipelines.**

## CyclicBarrier

A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. CyclicBarriers are useful in programs involving a fixed sized party of threads that must occasionally wait for each other. The barrier is called cyclic because it can be re-used after the waiting threads are released.

A CyclicBarrier supports an optional Runnable command that is run once per barrier point, **after the last thread in the party arrives**, but **before any threads are released**. This barrier action is useful for updating shared-state before any of the parties continue.

It takes a single integer that denotes the number of threads that **need to call the await() method** on the barrier instance to signify reaching the common execution point.

CyclicBarrier barrier = new CyclicBarrier(3) -> it means 3 threads must call the wait so the barrier trips. The fist two will wait until the third one arrives

barrier.wait()

barrier.wait()

barrier.wait()

then all the threads are released

Some Points that I’m not so sure about:

* I think by reusability they mean after all the waiting threads has been released or the barrier has broken, the barrier resets automatically but it could also mean the presence of a reset method that resets the barrier to the initial state and all the waiting threads will throw a *BrokenBarrierException*. But Javadoc says you should not use reset method for other than breaking the barrier. I don’t now
* I’m like 90% sure about this one though: the last thread that enters the barrier point will execute the Runnable command and then all the threads are released

## Phaser

We don’t go deep on this one. Just be aware that this is similar to what Cyclic Barriers and Count Down Latches do with more flexibility

## SynchronousQueue

Read about it later on …..

# Atomic Operations and Classes

**Get back to 00:56’:20” -01:07’:40”**

# The java.util.concurent.locks package

**You Can get back to this section later on as I didn’t spend much time to go through all the documents, I simply introduce the terms.**

## The Lock Interface

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

Using synchronized blocks is called implicit locks I guess and using objects of the Lock interface is called explicit lock.

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 by:
  + providing a non-blocking attempt to acquire a lock (**tryLock()**)
  + an attempt to acquire the lock that can be interrupted (**lockInterruptibly())**
  + and an attempt to acquire the lock that can timeout (**tryLock(long,TimeUnit)**).

### ReentrantLock

An implementation of the Lock interface. When a thread that owns the lock calls the lock method, it’ll return immediately.

## The ReadWriteLock Interface

When using synchronized methods/statements, if two threads are trying to read the second one is blocked but we know that it will be safe if they both perform a read operation.

This interface provides you with writeLock() and readLock() methods to acquire an exclusive lock for write operations and a lock for read operations that can be shared between threads that tr to perform a read operation.

So, a read lock can be acquired if no write locks are being held but write locks can be acquired if neither a read lock nor a write lock is being held by another thread

rwl. readLock().unlock()/lock();

rwl. writeLock().lock()/unlock();

**Look at Write Operation as a modifying operation and read operation as an operation that doesn’t change the data. For example a consumer that removes data from a queue performs a write operation.**

**Look at how you can use this concept to implement caches**

**Look at the difference between the performance of these two(Locks/ReadWriteLocks) when there is not many read operations.**

### ReentrantReadWriteLock

An implementation of the ReadWriteLock interface

## Conditions

Read About This Interface in the above package

# The Executers Framework

**You Can get back to this section later on as I didn’t spend much time to go through all the documents, I simply introduce the terms.**

Earlier we created new threads by using the Thread class and starting a thread object that contains the task that has to be done.

This Is not how it’s done in complex systems:

* They define the task that must be done in one part of the application(Task Submission)
* In another part of the app they manage creation and scheduling of the threads responsible for performing these tasks(Task execution)

## Executers

An object that executes submitted Runnable tasks. This interface provides a way of **decoupling task submission from the mechanics of how each task will be run, including details of thread use, scheduling**, etc.

An Executor is normally **used instead of explicitly creating threads**. For example, rather than invoking new Thread(new RunnableTask()).start() for each of a set of tasks, you might use:

Executor executor = anExecutor();

executor.execute(new RunnableTask1());

executor.execute(new RunnableTask2());

public interface Executor {  
  
 /\*\*  
 \* Executes the given command at some time in the future. **The command**  
 \* **may execute in a new thread, in a pooled thread, or in the calling  
 \* thread, at the discretion of the {@code Executor} implementation**.  
 \*  
 \* @param command the runnable task  
 \* @throws RejectedExecutionException if this task cannot be  
 \* accepted for execution  
 \* @throws NullPointerException if command is null  
 \*/  
 void execute(**Runnable** command);  
}

### Thread Pool

Executors normally keep a pool of threads and don’t create a new thread for each task

They can reuse these threads to save the overhead of creating a new thread. So based on the implementation they for example try to reuse an idle thread from the thread pool and if they fail to do so they create a new one and keep it in the pool.

### The Executors class

factory and utility methods for Executor, ExecutorService, ScheduledExecutorService, ThreadFactory, and Callable classes defined in this package. This class supports the following kinds of methods:

Methods that create and return an ExecutorService set up with commonly useful configuration settings.

Methods that create and return a ScheduledExecutorService set up with commonly useful configuration settings.

Methods that create and return a "wrapped" ExecutorService, that disables reconfiguration by making implementation-specific methods inaccessible.

Methods that create and return a ThreadFactory that sets newly created threads to a known state.

Methods that create and return a Callable out of other closure-like forms, so they can be used in execution methods requiring Callable.

Here are some examples:

*/\*\*  
 \* Creates an Executor that uses a single worker thread operating  
 \* off an unbounded queue. (Note however that if this single  
 \* thread terminates due to a failure during execution prior to  
 \* shutdown, a new one will take its place if needed to execute  
 \* subsequent tasks.) Tasks are guaranteed to execute  
 \* sequentially, and no more than one task will be active at any  
 \* given time. Unlike the otherwise equivalent  
 \* {@code newFixedThreadPool(1)} the returned executor is  
 \* guaranteed not to be reconfigurable to use additional threads.  
 \*  
 \* @return the newly created single-threaded Executor  
 \*/*public static ExecutorService newSingleThreadExecutor()

*/\*\*  
 \* Creates a thread pool that reuses a fixed number of threads  
 \* operating off a shared unbounded queue. At any point, at most  
 \* {@code nThreads} threads will be active processing tasks.  
 \* If additional tasks are submitted when all threads are active,  
 \* they will wait in the queue until a thread is available.  
 \* If any thread terminates due to a failure during execution  
 \* prior to shutdown, a new one will take its place if needed to  
 \* execute subsequent tasks. The threads in the pool will exist  
 \* until it is explicitly {@link ExecutorService#shutdown shutdown}.  
 \*  
 \* @param nThreads the number of threads in the pool  
 \* @return the newly created thread pool  
 \* @throws IllegalArgumentException if {@code nThreads <= 0}  
 \*/*public static ExecutorService newFixedThreadPool(int nThreads) {

*/\*\*  
 \* Creates a thread pool that creates new threads as needed, but  
 \* will reuse previously constructed threads when they are  
 \* available.* ***These pools will typically improve the performance  
 \* of programs that execute many short-lived asynchronous tasks.*** *\* Calls to {@code execute} will reuse previously constructed  
 \* threads if available. If no existing thread is available, a new  
 \* thread will be created and added to the pool. Threads that have  
 \* not been used for sixty seconds are terminated and removed from  
 \* the cache. Thus, a pool that remains idle for long enough will  
 \* not consume any resources. Note that pools with similar  
 \* properties but different details (for example, timeout parameters)  
 \* may be created using {@link ThreadPoolExecutor} constructors.  
 \*  
 \* @return the newly created thread pool  
 \*/*public static ExecutorService newCachedThreadPool() {

# The Callable Interface

If you ant to return something from a Runnable you would have to store the value into a shared resource and do some synchronization on it and… ooooof

You can use a Callable instead

*/\*\*  
 \* A task that returns a result and may throw an exception.  
 \* Implementors define a single method with no arguments called  
 \* {@code call}.  
 \*  
 \* <p>The {@code Callable} interface is similar to {@link  
 \* java.lang.Runnable}, in that both are designed for classes whose  
 \* instances are potentially executed by another thread. A  
 \* {@code Runnable}, however, does not return a result and cannot  
 \* throw a checked exception.  
 \*  
 \* <p>The {@link Executors} class contains utility methods to  
 \* convert from other common forms to {@code Callable} classes.  
 \*  
 \* @see Executor  
 \* @since 1.5  
 \* @author Doug Lea  
 \* @param* <*V*> *the result type of method {@code call}  
 \*/*@FunctionalInterface  
public interface Callable<V> {  
 */\*\*  
 \* Computes a result, or throws an exception if unable to do so.  
 \*  
 \* @return computed result  
 \* @throws Exception if unable to compute a result  
 \*/* V call() throws Exception;  
}

# The Future Interface

# ExecutorService

# ScheduledExecutorService

# ThreadLocal Objects

This class provides thread-local variables. These variables differ from their normal counterparts in that each thread that accesses one (via its get or set method) has its own, independently initialized copy of the variable.

**ThreadLocal instances are typically private static fields** in classes that wish **to associate state with a thread** (e. g., a user ID or Transaction ID).

For example, the class below generates unique identifiers local to each thread. A thread's id is assigned the first time it invokes ThreadId. get() and remains unchanged on subsequent calls.

public class ThreadId {

// Atomic integer containing the next thread ID to be assigned

**private static final AtomicInteger nextId = new AtomicInteger(0);**

// Thread local variable containing each thread's ID

**private static final ThreadLocal<Integer> threadId =**

**new ThreadLocal<Integer>() {**

**@Override protected Integer initialValue() {**

**return nextId. getAndIncrement();**

**}**

**};**

// Returns the current thread's unique ID, assigning it if necessary

**public static int get() {**

**return threadId. get();**

**}**

**}**

Each thread holds an **implicit** reference to its copy of a thread-local variable as long as the thread is alive and the ThreadLocal instance is accessible; after a thread goes away, all of its copies of thread-local instances are subject to garbage collection (unless other references to these copies exist).

Simply put, we can imagine that ThreadLocal stores data inside of a map with the thread as the key.

**From 01:52’:07” on is going to be skipped for now**